Kathy Camenzind
Alex Huang
Xavier Zapien

Final Project Report: Pong.iRL

6.111 Fall 2018

Table of Contents

Overview

Inspired by the pong lab, our team has decided to build a robotic system to play air hockey against a human opponent, which is essentially the same game as pong but with an added degree of motion. The air hockey robot consists of a machine vision module, air hockey robot logic module, and a motion control module:

- *Machine vision module*: responsible for processing camera VGA data, buffering it into a frame and then processing the frame to isolate the position of the puck and paddle. It then passes this information to the air hockey robot logic. We anticipate this module to be the most challenging aspect of the design.

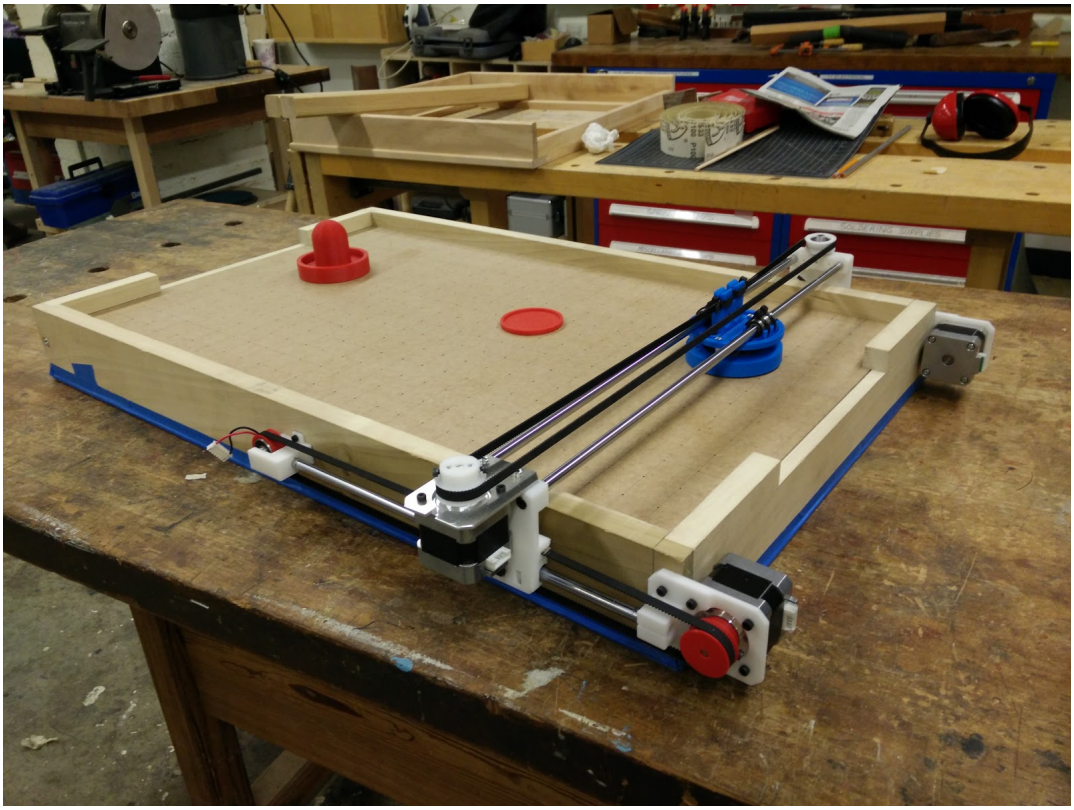
- *Air hockey robot logic module*: implements an “AI” that predicts the trajectory of the puck (accounting for bounce off the walls) and outputs a desired paddle position based on a game strategy which we will implement.

- *Motion control module*: controls how to actuate stepper motors to move paddle. An important consideration with stepper motors driving a load is that they will need to be ramped (accelerated) to avoid stalling. This module will ensure that we move the paddle in the most efficient way possible.

Implementation

The Air Hockey Robot

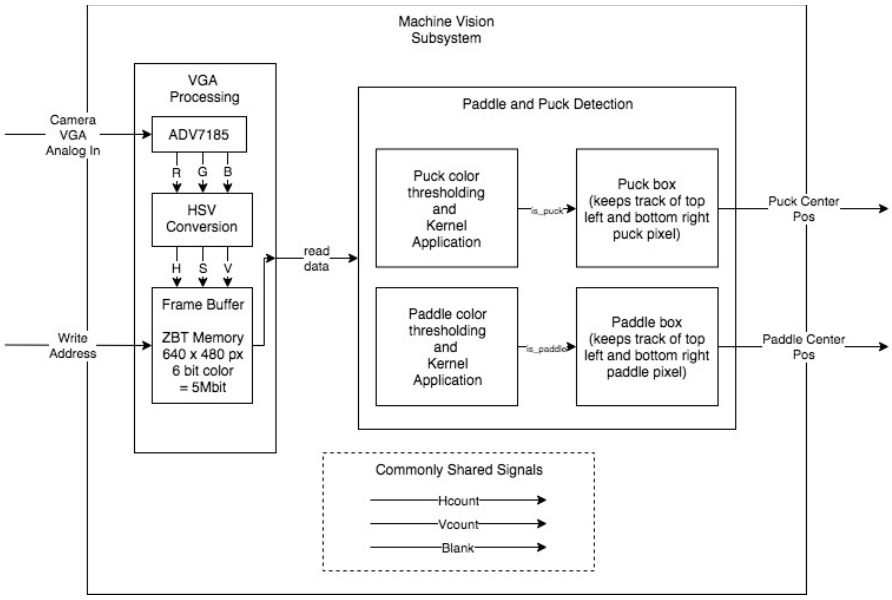
There is a substantial mechanical element to this project, but we already have it built:



This is a miniature air hockey table that we have custom built; it measures roughly 18” x 29”. There is a set of rails that allows the gantry (the paddle + rail + motor assembly) to move on the X axis and another set that allows the paddle (pictured in blue) to move on the Y axis. These parts are driven by 3 stepper motors. The table itself is essentially a big enclosed chamber with two computer fans underneath to draw air in. Tiny holes are drilled in a gridlike formation on the table top to allow air streams to escape. This airstream levitates the puck, which can slide around with little friction.

Machine Vision Module

Initial Description:



The machine vision module is responsible for handling camera data and identifying the paddle and puck. The camera will be mounted on the air hockey table and provide a birds eye view. Its data is passed to the ADV7185 NTSC decoder, which outputs RGB data. To save space, we are considering using only the top 6 bits of this color information. This data is then fed to the HSV conversion module which, as the name implies, outputs the HSV values of the pixel. We choose to use HSV because inevitably we will need to play with threshold values to pick up the colors of the paddle and puck and having HSV allows for more intuitive tuning of these parameters. The HSV data will be stored in a ZBT memory frame buffer.

The paddle and puck detection module will then read this data given a specific Hcount and Vcount. An erosion and dilation kernel will be applied to filter out noise and the top-left and bottom-right pixel will be kept. Potential problems of doing this may be noise causing issues if the erosion and dilation kernel does not adequately filter. A potential solution may be also to include some temporal filtering of the position of the puck and paddle as these are physical objects that cannot move very quickly relative to the processing time of the FPGA. In order to keep the filtering as simple as possible, we will also attach a generous light source so illumination is ideal and noise is reduced. In

addition to this, we will also require that players wear a monochrome sleeve when they are playing the game so as to not introduce false positives in our identification pipeline.

Seeing as it may almost be impossible to debug, we will have stream the camera view and overlay boxes which bound the paddle and puck positions so we can debug this module.

The output of this module - namely, the x and y positions of the paddle and puck will be passed to the AI subsystem using a shared register.

Hardware Usage:

Hcount: 10 bits to account for 640 pixels
Vcount: 9 bits to account for 480 pixels
Read/Write address: 19 bits to account for 640 * 480 possible addresses
Puck,Paddle positions: 10 and 9 bit registers for x and y
R, G, B: 6 bits for each color
H, S, V: 6 bits for each parameter

ZBT: 5.5 Mb of usage
HSV Conversion Module: 2x 16 bit multiplier IP

Changes in Implementation:

The implementation followed the initial description quite closely. However, there were changes to the data pathway. Instead of the RGB, the NTSC decoder actually output YCrCb. Fortunately, there was a staff provided module that converted this to RGB which we leveraged.

Instead of storing RGB data directly into ZBT, we actually decided to do HSV conversion and keying beforehand and store a pixel mask (such as pure black or pure green) if the pixel passes the threshold and is considered an object. In retrospect, it would probably be architecturally better to do this AFTER the ZBT as initially proposed in order to make the delay logic simpler. As it stands, we had to make changes to the ZBT control signals in order to take into account the cycle overhead created by the RGB to HSV conversion, which is arguably harder to understand than just synchronizing the pixels within vram_display.

As the picture was sufficiently noise free, we only implemented 1D erosion, which got rid of small pixel noises. It is also useful to note that by changing the focus on out camera input, we were essentially able to blur out our frames and kill some noise.

In calculating the centroid, we were initially going to keep the top left and bottom right most pixels in order to draw a box around the puck. However, we quickly found out that this is very prone to noise. In order to deal with this, we ended up averaging the hcount and vcount values of any pixels that had a mask. This method have us relatively good results for the centroid with noises of 1 to 2 pixels. A square box of precomputed dimensions can be centered around this centroid since the distance from the camera to board does not change. As we process each pixel as it is coming in from the camera, we were actually able to attain real time processing and position updates, which is something that we were uncertain could be achieved in the beginning of the project.

The paddle and puck detection logic remains the same as initially proposed, but we did add a module called hsv_threshold_select that allowed us to change HSV settings on the fly. Combined with the debug screen output, we were able to experiment with what hue, saturation and value gave the best results in identifying the colored puck and paddle that we had. This also allowed us to deal with any lighting changes in the lab. Note that using highly saturated colors helps greatly with chroma keying and providing a constant, dependable source of illumination will drastically make this process easier.

The debug screen actually ended up being incredibly helpful in seeing what was going on within the modules, which would have been impossible otherwise. We also added a shapes submodule that allowed us to paint crosshairs over identified objects or planned trajectories.

Challenges

The main challenge we faced in implementing machine vision was understanding the legacy staff code in interfacing the labkit to the NTSC camera. It was difficult to understand what all the control signals were doing and the code was not extensively documented. It is possible to get color video without completely understanding everything and in fact it is probably not worth spending a lot of time doing so (as we did). Here are a few tricky bits that might be useful to know:

- There are counters that increment which control when a pixel is stored in ntsc2zbt.v. These must be changed since going from BW to color images involve

storing fewer pixels in each ZBT location. Reference Mike Wang's final project writeup in Fall 2017 to see more exact locations in which it is changed.
- The delay logic in pixel must also be changed to wherever there is a delay of 4 pixels to that of 2 now.
- The ~22 black lines that you see at the top of the frame is control signals.

The other challenge was getting to know how data actually flowed from the camera to the labkit. In a nutshell, the NTSC camera outputted YCrCb, which was converted to RGB. This was stored in ZBT according to the control signals that ntsc2zbt.v generated. Then, the resulting pixels from ZBT is displayed to the screen by vram_display in zbt_6111_sample.v.

Lessons Learned

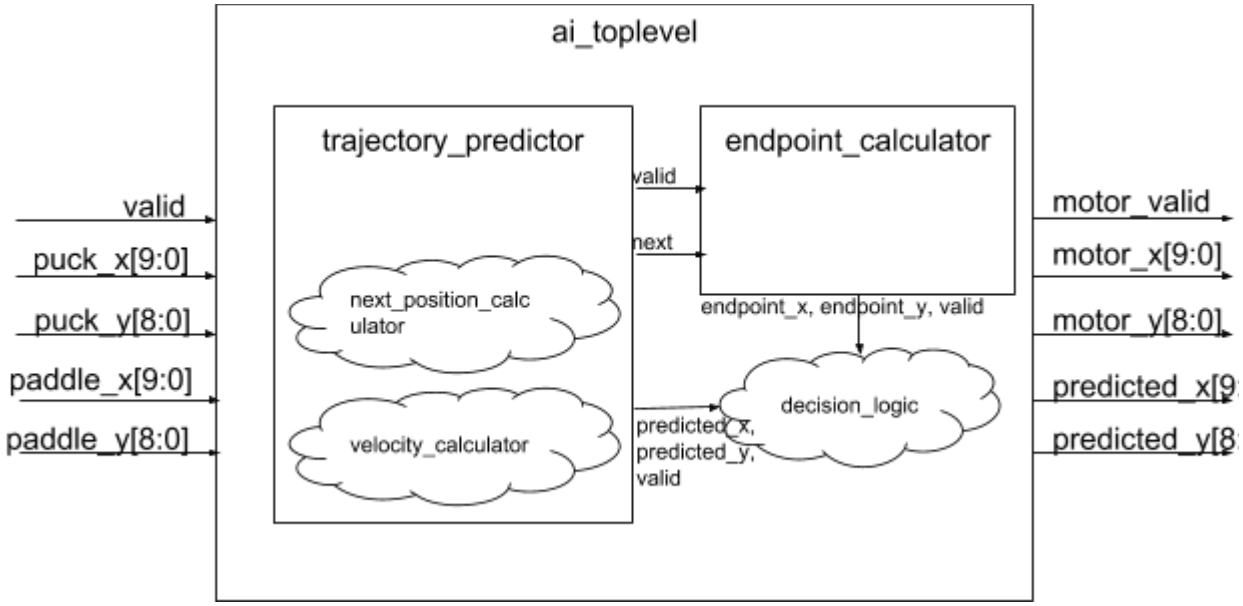
The main lesson we took away from the machine vision potion was that having visual feedback on the debug screen was much more important than anticipated. This turned out to be much more of a feature than just a debug tool, as we could give others a visual aid as to what is happening in our implementation on top of helping us figure out what was going wrong.

It is also important to integrate early. We did not think about agreeing on a coordinate system early on so we each had a slightly different idea of where (0,0) corresponded to on the board and this led to problems later on when we were putting all the modules together. Although this seemed simple, it actually ended up causing some bugs that were uncaught until we really sat down and discussed the coordinate system.

Robot Logic Module

The Robot Logic module is responsible for determining where the paddle should go next based on the position of the puck and the robot paddle on the board. The module is passed the puck and paddle coordinates at a rate of 30Hz, and a `valid` wire is pulsed along with a new reading. The dimensions (in pixels) of the board as viewed by the camera was 446x700, so for both the puck and paddle the x coordinate was a $\log_2(446)=9$ bit number, and the y coordinate was a $\log_2(700)=10$ bit number. There is only one valid wire, since the paddle and puck position are given at the same time.

The AI module is broken into 3 main submodules: the top level AI, a trajectory predictor, and an endpoint calculator, as shown in the block diagram below.



Trajectory predictor: This submodule takes in puck_x and puck_y readings, and for every reading will generate a stream of next_pos readings based on where it thinks the puck will go next. This module's internal state includes the puck's previous position and previous velocity, and then when a new reading is passed in, it uses a combinational velocity_calculator module to determine what it thinks the current velocity of the puck, where the velocity consists of an unsigned x and y speed (in pixels/frame), and an x and y direction (1'b0 for negative, 1'b1 for positive). This is slightly more complicated than it seems because of wall bounces, i.e. if the puck in the previous timestep went from x=50 to x=75, we need to know if the x velocity is 25, or if it actually bounced off of the top wall (at pixel 446) and the speed is 767, or if it bounced off the bottom wall and the speed is -125 pixels/frame. The way this module ends up doing this is by calculating all possibilities in parallel and choosing the one with the least error in calculated current velocity vs previous velocity, because we expect the puck to not randomly change speed.

The submodule primarily outputs a stream of next_position values. After a new puck position is received (puck_pos[i]), on the next clock cycle the submodule starts calculating where the puck will be in the next cycle, i.e. the predicted puck_pos[i+1]. This calculation is performed by another combinational module, next_position_calculator, which takes in the current speed, direction, and position and calculates the next direction and position, assuming that the speed doesn't significantly change. The trajectory_predictor module then continuously calculates puck_pos[i+n] until a max n, and has a valid signal that it asserts as long as it is outputting valid predicted puck

positions. We're able to output a new position every clock cycle between frame readings.

This trajectory predictor also proved invaluable in debugging. We could output from the module the predicted position of the puck in x timesteps from now (generally x=15, which corresponds to 0.5 second ahead prediction in real time) to see where our AI thought the puck was going.

Endpoint Calculator: Since our motors had limited acceleration and relatively slow response time, it's not an ideal strategy to just track the current position of the puck. Instead, we decided that a good defensive strategy would be to have the robot go all the way back in the x direction, and in the y direction go to where the trajectory predictor thought the puck would be when it reached that goal line. So the endpoint_calculator module takes in the stream of predicted next puck positions, waits for one that crosses the goal line, and outputs that y position. However, since this corresponds to the direct command to the motor, this module also does some time averaging to reduce the noise of the signal sent to the motor. This means that the endpoint calculator actually only outputs data every 5 frames (so 6 per second) instead of every frame (30 per second).

Top-level AI: This last submodule is where the other two are instantiated, and directly takes in the input from the machine vision module and outputs to the motor module. It determines, based on the puck position and paddle position, whether we want the robot to have a defensive or offensive strategy, and then decides what command to actually send to the motor. When in an offensive strategy (the puck is on our half of the board, and the velocity is below a threshold so that we know we have time to hit it), the motor command is just the position of the puck, i.e. we want the paddle to move towards the puck to hit it. When in a defensive strategy, which is the default, the paddle moves to where the endpoint_calculator says the puck will cross the goal line. Lastly, if the puck is actually level with or behind our paddle, we want to move away from it, so that we don't accidentally score on ourselves by hitting the puck backwards into our own goal.

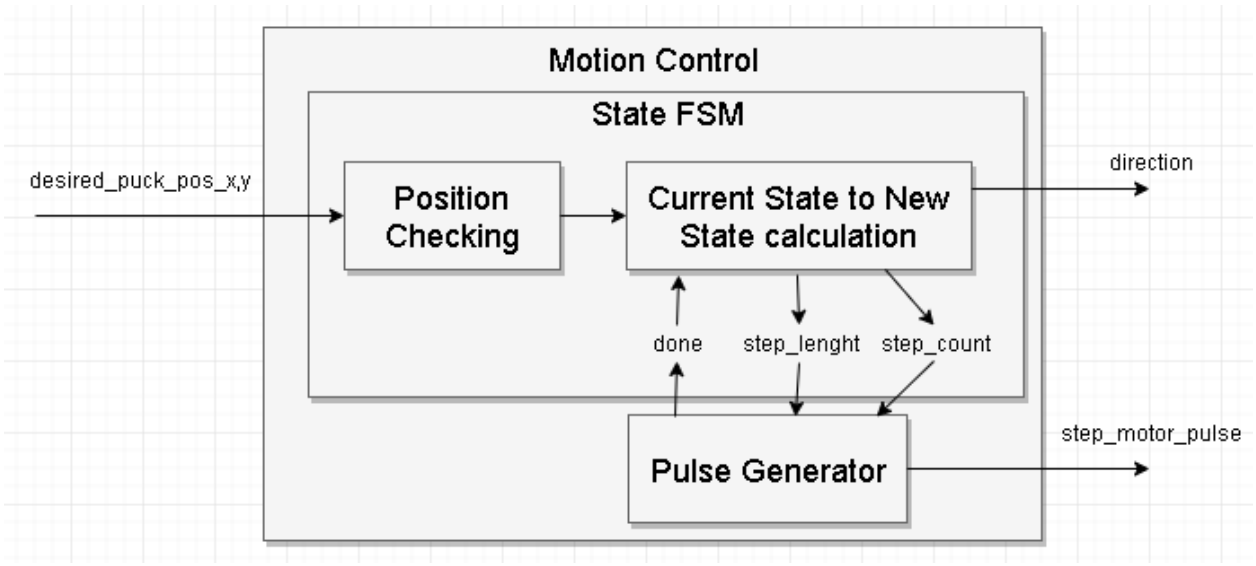
There are many ways to implement a more complex version of this AI. Some ideas are adding a second control phase: (1) moving into position to block the puck from scoring and (2) moving forwards to hit the puck. We could also implement a more aggressive strategy, where the trajectory submodule would output an array of (position, time) values of the puck's trajectory on the AI's side of the board, instead of just where it will reach the goal. It could then decide whether it has to move backwards at all, versus hitting the puck while far forward and giving the opponent less time to react.

Challenges:

The main challenge for these modules was debugging in real time. I wrote testbenches for each module and they worked as expected in simulation, but there are of course edge cases and interactions with the other modules that are unexpected until all of the modules are integrated. The screen display that we had was invaluable for debugging the AI module, as we could then draw crosshairs on calculated trajectories and endpoints and see under what conditions they were and weren't what we expected. Honestly, most of the bugs caught by this had to do with passing the wrong signal between modules (or missing signals), or signals that were combinational and changing instead of getting saved into a register.

One primary problem that we discovered even after the module was working as expected was the rate of data flow. The AI module is capable of sending continuously updated motor positions every frame, but the motor module was unhappy with suddenly changing directions, which is why we added some low pass filtering to clean up the signal we were sending to the motors.

Motion Control Module



Module Overview

Initially, the motion control module will be interfacing with one of the three stepper motors to control the Y position of the blue paddle. This module contain 3 sub-modules: State FSM, Boundary Checking (Position checking), and a pulse generator. Given the outputs of the AI Module, the boundary checking module will make sure that the inputs received from the AI module make sense and if they do not then the paddle will be moved to the center point. This error checking is vital for the safety of the paddle. If the position is valid, the 9 bits are sent to the current and new state calculator. This section will compare the last two desired puck positions and generate three outputs: direction, step_lenght, and step_count. Direction is fed directly onto the A4988 Stepper Motor Driver Carrier and will dictate on what direction the robot will move. Step_count will be a 15 bit value that will be fed into the pulse generator sub module. Step_count will be the number of steps needed for the motor to reach the new desired position. Step_length will also be sent to the pulse generator. This was fixed to make a step length of 2us, but after testing for stalling, this value might be changed to prevent damage to the motors.

The Pulse Generator submodule will create a pulse out of the step_count and step_lenght values. The pulse will be a 5V 2us pulse that will happen every 4us until the step_count reaches zero. Once the final pulse has been delivered, the robot assumes that it has reached it final destination and asserts a HIGH on its done output. This allows the state transition to only send new data to the pulse generator only when it has finished writing the last position to the paddle.

Hardware Usage:

Inputs
desired_puck_pos_x_y: 19 bits

Outputs:
direction: 1 bit
Step_motor_pulse: 1 bit at 500kHz

External: A4988 DMOS Microstepping Driver

Initial Implementation: Before starting the project, I had to think of how we were going to move the robot paddle across the board. Due to the nature of robot precision, it was evident that we had to choose stepper motors instead of regular DC motors. Given the fact that we will be wanting to stop a precision points in the board, the stall and gliding of the DC motors will cause us to overshoot or undershoot very frequently. This might

lead to an error build up and the need of a feedback circuit or control system. In order to simplify the problem to the actual verilog code, we chose to use two phase stepper motors. We chose a Nema 17 armature since it is very common and heavily used in 3D printers, which need to have precise and fine movements inside of an x-y axis. From our previous experience with stepper motors, we knew that we needed to have a stepper motor driver. We ended up choosing the A4988 from adafruit since it was compatible with our motors and it had lots of freedom in step size (Full, half, quarter, eight). It also only needed a direction and step signal in order to drive our motor. A sample schematic of the A4988 is observed in figure 3 below.

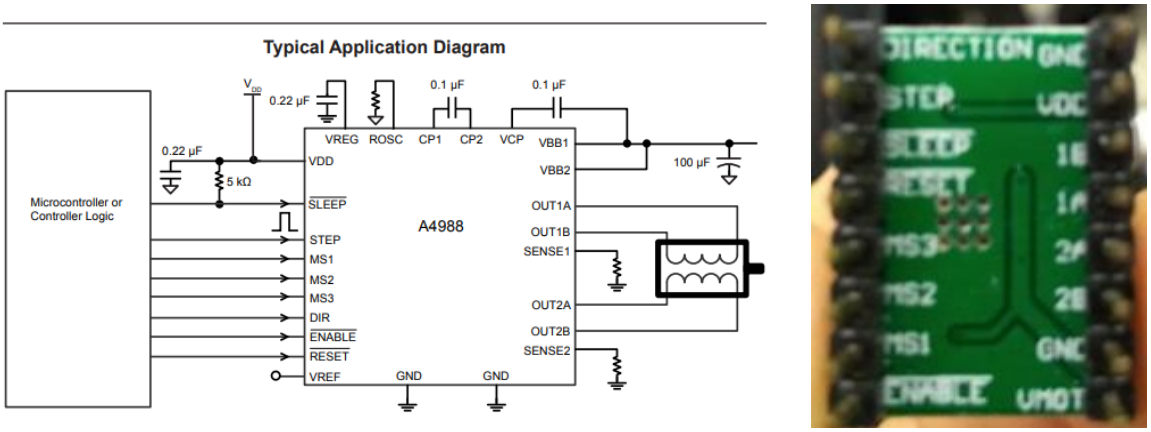
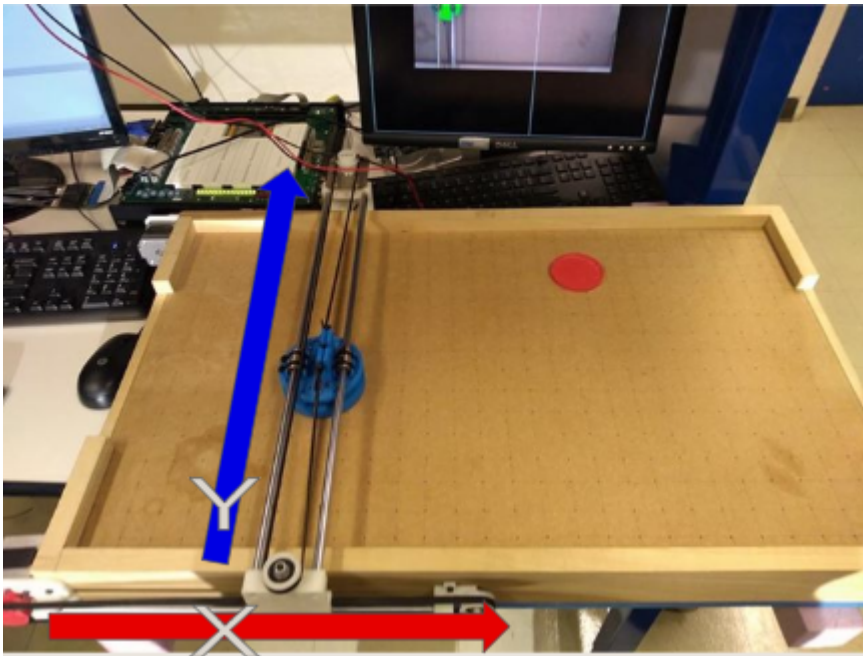


Figure 2: Typical A4988 Schematic

Since we were going to run three motors, we needed to have three of these A4988 drivers. One driver for the Y axis and two for the X axis. The Y axis only has to carry the load of the paddle meanwhile the X axis has to carry the entire Y system (paddle, pulley, belt, stepper motor), explaining the reasoning to have two stepper motors in our X axis.



Pong Lab X and Y

These stepper motors take in the following inputs: DIRECTION, STEP, MS1, MS2, and MS3. For the initial implementation, we assumed that we could work with half stepping, so based on the datasheet we could set MS1=1, MS2=0, and MS3=0. This left us only worrying about DIRECTION and STEP. DIRECTION will is a one bit signal that can be set at 0 for forward, and at 1 for backwards. Then STEP will take in a high step function for every step that the motors will take. One complete revolution will take 200 steps. After some calculations, we found out that we move at a rate of about .043 cm/step. Which translates to a maximum of 3256 half-steps to cover the entire board. This stepping sequence was assuming that we would be able to reach our desired position in the short amount of time in between new values. The FSM would not change to the next state until the value we wanted to get to was reached.

Changes to Initial Implementation

In our final implementation, we had to change how fast we moved the motor and how fast we were going to be accelerating, as both of these were not able to keep up with the speed of the other modules. One large constraint of this module are the physical restrictions of the friction and inertia that the paddle mechanism has. It requires a large amount of force to get it started prior to move it along the axis of motion. For that reason, we were forced to install a acceleration module in our motion control. During our integration, we were noticing that our paddle was not moving as fast as we expected. This was being caused by the lack of torque in our initial displacement. The acceleration module outputted a pulse to the pulse gen module. This pulse had an increasing frequency with a pulse being generated every 1ms down to .4ms, with a

.002ms decrease every time a pulse was generated. The pulse gen module received this signals and created a square wave every time it was activated with a start signal.

Sub Module Description

1. **Acceleration**
2. **Timer**
3. **FSM**
4. **Pulse Generator**

1. Acceleration:

```
module Acel_Timer(input clock, input start, output count);
```

Acel Timer takes in the system clock, and a start pulse (1 clock length) and generates pulses on the output count. As seen in the appendix, the variable `variable_timer` is loaded with TWO_MILI = 16'h6978, which is hex for 27000 counts, meaning that in this initial value, we will send a pulse every two millisecond (.5 ms high and .5ms low). This is a good initial pulse width to get the motors moving, after every clock cycle, we decrement the variable_timer by 4 counts, this gives us that acceleration we need. We also gave it a minimum of 16'h4500 for the max velocity of the paddle. This value was not calculated but it was determined by experimentation. Its count output is sent to the pulse generator module for the Y axis

2. Timer:

```
OneUsTimer timer(.clock(clock), .count(count_pulse_x));
```

The timer works very similar to Acceleration but it is simplified, as it does decrement the value of variable_timer. Since there is no change in variable timer, there is no need for a start signal, so no start signal is found here.

3. FSM:

```
Transition_y FSM_y(.clock(clock), .start(on_signal),
 .x_desired(x_val_wire),
.y_desired(desired_y),
.current_y(current_y),
.pulse_done(data_valid),
.motor_stop(stop_motor),
.reset(reset)
,.step_count(step_count_y),
.step_mode(step_mode_y),
.start_pulse(start_signal_y),
 .direction(direction_y),
```

.state_x(state_y));

NOTE: step_count, step_mode were not used in the final implementation. State_x is the output of the current state which was used for debugging and is not connected to other modules.

The FSM is timed with the global clock signal, and is started with the start signal. When started it takes in the following major inputs (y_desired (10 bits), current_y (9 bits), pulse_done, and reset. Pulse done has been repurposed to be a data valid input, meaning that this is pulsed high when a valid data set has been sent to the motion module. Once this is triggered high, the FSM can do another transition. When the start signal is received, it initializes all of the registers to be in the middle location, meaning that the paddle will be at the center points of its Y and X axii.

The FSM has 5 states:

BEGIN: waits for a signal in pulse done and moved to NEW_STATES, if not it stays in BEGIN

NEW_STATES: initializes both x_new and y_new with both of the new locations sent by the AI module, then moves to MATH.

MATH: Checks if the new location is greater or less than the current y value and changes the direction accordingly. If the direction is not going to be changed, it jumps back to begin, if the direction is going to change, it sets direction_reg to its opposite value and moved on to BEGIN_PULSE. Also, if the current_y and Y_desired and within the POS_BOUND, the motors are stopped, this gives us a bound in which we can have the motor not move and be able to defend the puck. Due to camera noise, without this check the paddle would shift back and forth.

BEGIN_PULSE: the old values of the y coordinate are saved in an old register, and the start pulse signal is raised high. Moves on to END_PULSE.

END_PULSE: lowers END_PULSE and moves on to BEGIN.

4. Pulse Generator

Module pulse_generator(input clock, input kill,
input start,
input count,
input[11:0] step_count,
input [2:0] step_mode,
output pulse,
output done,
output MS1, output MS2, output MS3);

NOTE: step_mode, done, MS1, MS2, and MS3 were not used in the final implementation of this module.

Pulse gen was started with the start signal, and whenever it received pulse in the count, it outputed a high signal until it received another count and then outputted a low signal and on and on, this allowed the pulses to be generated with a fixed or variable duty cycle based on the count input. Step_count was hardcoded to 12'hFF for the final implementation as that was a value that was safe for the final implementation.

Roadblocks and Issues

Communication Speeds between modules:

When interfacing a simulation module to real life, you always have to consider the speeds at which you send the data. This was very important when communicating the AI with the Motion module, since the AI could sent data at 100 samples per second, which was too fast for the paddle to move to those points. This was a very evident issue when we noticed that the paddle was twitching and not moving at all. From our implementation, the motion FSM was started every time it received a data valid signal from the AI, which was happening at 100 samples a second. This caused the FSM to restart every time and not do the entire sequence, leading to no movement.

Paddle jerking back and forth, starting stopping

Another issue we had in the implementation was that we were reaccelerating the robot everytime it received a new value. For instance, if the robot was moving up (+Y direction) from a value in 20 to a value in 40, it would start the acceleration sequence, but if it received a value of 60 before it reached 40, it restarted the acceleration and started moving to 60 now. This was very inefficient and caused the paddle to stop every time it was reaccelerating. In order to solve this, we inserted a check in our FSM, if we are moving in a direction already and we are asked to keep moving in that direction, then do not reaccelerate and keep going in that direction. Once you reach your desired destination, are you able to restart the accceleration.

Appendix

1. Top Level Code
2. Machine Vision Code
3. AI Code
4. Motion Module Code

Top Level Code

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

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

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

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

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

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

                clock_feedback_out, clock_feedback_in,

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

```
            rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

            mouse_clock, mouse_data, keyboard_clock, keyboard_data,

            clock_27mhz, clock1, clock2,

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

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

            switch,

            led,

            user1, user2, user3, user4,

            daughtercard,

            systemace_data, systemace_address, systemace_ce_b,
            systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

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

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

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

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

```
        tv_out_subcar_reset;

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

input  button0, button1, button2, button3, button_enter, button_right,
```

```
            button_left, button_down, button_up;
  input  [7:0] switch;
  output [7:0] led;

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

  inout [43:0] daughtercard;

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

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

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

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

  // Video Output
  assign tv_out_ycrcb = 10'h0;
  assign tv_out_reset_b = 1'b0;
  assign tv_out_clock = 1'b0;
  assign tv_out_i2c_clock = 1'b0;
  assign tv_out_i2c_data = 1'b0;
  assign tv_out_pal_ntsc = 1'b0;
```

```
  assign tv_out_hsync_b = 1'b1;
  assign tv_out_vsync_b = 1'b1;
  assign tv_out_blank_b = 1'b1;
  assign tv_out_subcar_reset = 1'b0;

  // Video Input
  //assign tv_in_i2c_clock = 1'b0;
  assign tv_in_fifo_read = 1'b1;
  assign tv_in_fifo_clock = 1'b0;
  assign tv_in_iso = 1'b1;
  //assign tv_in_reset_b = 1'b0;
  assign tv_in_clock = clock_27mhz;//1'b0;
  //assign tv_in_i2c_data = 1'bZ;
  // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
  // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

  // SRAMs

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

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

/* enable RAM pins */

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

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

  assign ram1_data = 36'hZ;
  assign ram1_address = 19'h0;
```

```
  assign ram1_adv_ld = 1'b0;
  assign ram1_clk = 1'b0;

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

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

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

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

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

  // LED Displays
/*
  assign disp_blank = 1'b1;
  assign disp_clock = 1'b0;
  assign disp_rs = 1'b0;
  assign disp_ce_b = 1'b1;
  assign disp_reset_b = 1'b0;
```

```
  assign disp_data_out = 1'b0;
*/
  // disp_data_in is an input

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

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

  // Daughtercard Connectors
  assign daughtercard = 44'hZ;

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

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

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

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

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

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

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

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

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

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

// ENTER button is user reset
wire reset,user_reset, motor_kill, motor_start, motor_up, motor_down;
debounce db1(power_on_reset, clk, ~button_enter, user_reset);
debounce db2(power_on_reset, clk, ~button0, motor_kill);
debounce db3(power_on_reset, clk, ~button1, motor_start);
debounce db4(power_on_reset, clk, ~button2, motor_up);
debounce db5(power_on_reset, clk, ~button3, motor_down);

assign reset = user_reset | power_on_reset;

// display module for debugging

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

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

// wire up to ZBT ram

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

wire ram0_clk_not_used;
zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
```

```
                    vram_write_data, vram_read_data,
                    ram0_clk_not_used,   //to get good timing, don't connect ram_clk to
zbt_6111
                    ram0_we_b, ram0_address, ram0_data, ram0_cen_b);



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

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

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

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


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

     // Extract rgb values from hsv, and delay other signals accordingly

     wire [7:0] r, g, b;
     wire [17:0] rgb;
     reg [24:0] dv_delay; // 3 delay for ycrcb2rgb, 22 for rgb2hsv
     reg [74:0] fvh_delay; // 25 delay (same as above) * 3
```

```
    reg [131:0] r_delay, g_delay, b_delay; // 22 for rgb2hsv only * 6

  YCrCb2RGB
color_conv(r,g,b,tv_in_line_clock1,reset,ycrcb[29:20],ycrcb[19:10],ycrcb[9:0]);
    assign rgb = is_puck ? 18'b0 : is_pad ? {6'b0, 6'hFF, 6'b0} : {r_delay[131:126],
g_delay[131:126], b_delay[131:126]};


    always @(posedge tv_in_line_clock1) begin
      dv_delay <= {dv_delay[23:0],dv}; // 22 + 3
      fvh_delay <= {fvh_delay[71:0], fvh};
      r_delay <= {r_delay[125:0], r[7:2]};
      g_delay <= {g_delay[125:0], g[7:2]};
      b_delay <= {b_delay[125:0], b[7:2]};
    end

  wire u,d;
  wire [7:0] h_max1pd, h_min1pd,
        v_maxpd, v_minpd, s_minpd;

  debounce up_debounce(.reset(reset), .clk(clk), .noisy(button_up), .clean(u));
  debounce down_debounce(.reset(reset), .clk(clk), .noisy(button_down), .clean(d));

  hsv_threshold_select #(.HMIN1('h91), .HMAX1('hB0),
                .SMIN('h51), .SMAX(255),
                .VMIN('h37), .VMAX(255))
  pad_hsv(.clk(tv_in_line_clock1), .reset(reset), .enable(switch[4]), .up(~u), .down(~d),
.minmax_sel(switch[5]), .hsv_sel(switch[2:1]), .h_max1(h_max1pd), .h_min1(h_min1pd),
.v_max(v_maxpd), .v_min(v_minpd), .s_min(s_minpd));

  wire [7:0] h_max1pk, h_min1pk,
        h_max2pk, h_min2pk,
        v_maxpk, v_minpk, s_minpk;

  hsv_threshold_select #(.HMIN1(0), .HMAX1(10),
                .HMIN2(230), .HMAX2(255),
                .SMIN(140), .SMAX(255),
                .VMIN('h55), .VMAX(255))
```

```
   puck_hsv(.clk(tv_in_line_clock1), .reset(reset), .enable(switch[6]), .up(~u), .down(~d),
.minmax_sel(switch[5]), .hsv_sel(switch[2:1]), .h_max1(h_max1pk), .h_min1(h_min1pk),
.h_max2(h_max2pk), .h_min2(h_min2pk),  .v_max(v_maxpk), .v_min(v_minpk),
.s_min(s_minpk));


   assign dispdata[31:0] = switch[6] ? (switch[2:1] == 0 ? {h_max1pk, 16'b0, h_min1pk} :
                switch[2:1] == 1 ? {h_max2pk, 16'b0, h_min2pk} :
                switch[2:1] == 2 ? {24'b0, s_minpk} :
                switch[2:1] == 3 ? {24'b0, v_minpk} : {24'b0, v_minpk}) :

                switch[4] ? (switch[2:1] == 0 ? {h_max1pd, 16'b0, h_min1pd} :
                switch[2:1] == 1 ? 32'hFFFFFFFF :
                switch[2:1] == 2 ? {24'b0, s_minpd} :
                switch[2:1] == 3 ? {24'b0, v_minpd} : {24'b0, v_minpd}) : 32'hFFFFFFFF ;


   reg [63:0] ddata;
   wire [7:0] h, s, v;
   reg [4:0] puck_buf, pad_buf;

   //TODO rename these variable names with puck
   reg [31:0] x_acc, x_count,
           y_acc, y_count;
   reg divide_start;
   wire x_divide_ready, y_divide_ready;
   wire [31:0] x_quotient,
            y_quotient;

   obj_divider #(.WIDTH(32)) puck_divider_x (.clk(clk), .sign(0), .start(divide_start),
.dividend(x_acc), .divider(x_count), .quotient(x_quotient), .ready(x_divide_ready));

   obj_divider #(.WIDTH(32)) puck_divider_y (.clk(clk), .sign(0), .start(divide_start),
.dividend(y_acc), .divider(y_count), .quotient(y_quotient), .ready(y_divide_ready));

   reg [31:0] x_acc_pad, x_count_pad,
           y_acc_pad, y_count_pad;
   reg divide_start_pad;
   wire x_divide_ready_pad, y_divide_ready_pad;
```

```
   wire [31:0] x_quotient_pad,
               y_quotient_pad;

   obj_divider #(.WIDTH(32)) pad_divider_x (.clk(clk), .sign(0), .start(divide_start_pad),
.dividend(x_acc_pad), .divider(x_count_pad), .quotient(x_quotient_pad),
.ready(x_divide_ready_pad));

   obj_divider #(.WIDTH(32)) pad_divider_y (.clk(clk), .sign(0), .start(divide_start_pad),
.dividend(y_acc_pad), .divider(y_count_pad), .quotient(y_quotient_pad),
.ready(y_divide_ready_pad));

   reg [10:0] cross_x, pad_x;
   // end of video screen 40 + 22 + 486
   reg [9:0] cross_y, pad_y;
   wire[17:0] cross_pix, pad_pix;
   crosshair puck_cross( .hcount(hcount), .x(cross_x), .vcount(vcount), .y(cross_y),
.pixel(cross_pix));
   crosshair #(.COLOR({6'hFF, 6'h00, 6'h00})) pad_cross (.hcount(hcount), .x(pad_x),
.vcount(vcount), .y(pad_y), .pixel(pad_pix));

   //draw boundary lines
   reg[9:0] top_border_pos, bot_border_pos;
   wire[17:0]  top_border_pix, bot_border_pix, defense_line_pix, pred_crosshair_pix,
motor_crosshair_pix;

         parameter MOTOR_MAX_Y = 10'h180;
         parameter MOTOR_MIN_Y = 10'h28;
         parameter MOTOR_MIN_X = 100;
         parameter PUCK_RADIUS_PIXELS = 37;
         parameter X_CAMERA_OFFSET_PIXELS = 152;
         parameter Y_CAMERA_OFFSET_PIXELS = 40;
         parameter BUFFER_PIXELS = 10;
         parameter DEAD_Y_TOP_PIXELS = 22;
         parameter Y_TABLE_PIXELS_TOP = 10;
         parameter Y_TABLE_PIXELS_BOTTOM = 10;
         parameter FRAME_HEIGHT = 486;
         parameter FRAME_WIDTH = 720;
         parameter TOP_BORDER_PIXELS_Y = Y_CAMERA_OFFSET_PIXELS +
BUFFER_PIXELS + DEAD_Y_TOP_PIXELS + Y_TABLE_PIXELS_TOP;
```

```
        parameter BOTTOM_BORDER_PIXELS_Y = Y_CAMERA_OFFSET_PIXELS +
FRAME_HEIGHT + Y_TABLE_PIXELS_BOTTOM - BUFFER_PIXELS -
Y_TABLE_PIXELS_BOTTOM;
        parameter DEFENSE_LINE_PIXELS_X = MOTOR_MIN_X +
X_CAMERA_OFFSET_PIXELS + BUFFER_PIXELS;

   bound_line top_border(.vcount(vcount), .y(TOP_BORDER_PIXELS_Y),
.pixel(top_border_pix));
   bound_line bot_border(.vcount(vcount), .y(BOTTOM_BORDER_PIXELS_Y),
.pixel(bot_border_pix));
   vertical_line defense (.hcount(hcount), .x(DEFENSE_LINE_PIXELS_X),
.pixel(defense_line_pix));

   wire left, right;
   debounce left_debounce(.reset(reset), .clk(clk), .noisy(button_left), .clean(left));
   debounce right_debounce(.reset(reset), .clk(clk), .noisy(button_right), .clean(right));

   rgb2hsv converter(.clock(tv_in_line_clock1), .r(r), .g(g), .b(b), .h(h), .s(s), .v(v));

   wire in_frame;
   assign in_frame = (hcount > X_CAMERA_OFFSET_PIXELS + BUFFER_PIXELS) &&
(hcount < (X_CAMERA_OFFSET_PIXELS + FRAME_WIDTH - BUFFER_PIXELS))
        && (vcount > (Y_CAMERA_OFFSET_PIXELS + DEAD_Y_TOP_PIXELS +
BUFFER_PIXELS)) && (vcount < (Y_CAMERA_OFFSET_PIXELS + DEAD_Y_TOP_PIXELS +
FRAME_HEIGHT - BUFFER_PIXELS));

   always @(posedge tv_in_line_clock1) begin
      if (switch[3]) begin
         puck_buf <= {
            puck_buf[3:0],
            (((h > h_min2pk && h < h_max2pk ||
            h > h_min1pk && h < h_max1pk)
            && v > v_minpk && s > s_minpk)
            ? 1'b1 : 1'b0)
         };
         is_puck <= puck_buf == 5'b11111 ? 1'b1 : 1'b0;

         pad_buf <= {
            pad_buf[3:0],
```

```
            (((h > h_min1pd && h < h_max1pd) &&
            v > v_minpd && s > s_minpd) ?
            1'b1 : 1'b0)
          };

          is_pad <= pad_buf == 5'b11111 ? 1'b1 : 1'b0;

       end else begin
          is_puck <= (((h > h_min2pk && h < h_max2pk || h > h_min1pk && h < h_max1pk)
&& v > v_minpk && s > s_minpk) ? 1'b1 : 1'b0);

          is_pad <= (((h > h_min1pd && h < h_max1pd) && v > v_minpd && s > s_minpd) ?
1'b1 : 1'b0);
       end

    end

    reg puck_visible;
    reg puck_data_valid;
    wire puck_data_actually_valid;
    reg puck_data_valid_counter;

    assign puck_data_actually_valid = puck_data_valid && puck_data_valid_counter;
    parameter PAD_MARKER = {6'h0, 6'hFF, 6'h0};

    always @(posedge clk) begin
          puck_data_valid <= 0;
          puck_data_valid_counter <= puck_data_valid_counter + puck_data_valid; //
toggles
       if (x_divide_ready) begin
          cross_x <= x_quotient[10:0];
             puck_data_valid <= 1;
          x_acc <= 0;
          x_count <= 0;

          cross_y <= y_quotient[10:0];
          y_acc <= 0;
          y_count <= 0;
```

```
        pad_x <= x_quotient_pad[10:0];
        x_acc_pad <= 0;
        x_count_pad <=0;

        pad_y <= y_quotient_pad[10:0];
        y_acc_pad <= 0;
        y_count_pad <= 0;

    end else begin
        if (in_frame) begin
            x_acc <= (vr_pixel == 18'b0) ? x_acc + hcount : x_acc;
            x_count <= (vr_pixel == 18'b0) ? x_count + 1 : x_count;
            y_acc <= (vr_pixel == 18'b0) ? y_acc + vcount : y_acc;
            y_count <= (vr_pixel == 18'b0) ? y_count + 1 : y_count;

            x_acc_pad <= (vr_pixel == PAD_MARKER) ? x_acc_pad + hcount : x_acc_pad;
            x_count_pad <= (vr_pixel == PAD_MARKER) ? x_count_pad + 1 : x_count_pad;
            y_acc_pad <= (vr_pixel == PAD_MARKER) ? y_acc_pad + vcount : y_acc_pad;
            y_count_pad <= (vr_pixel == PAD_MARKER) ? y_count_pad + 1 : y_count_pad;
        end

        if (hcount == 872 && vcount == 548) begin
            puck_visible <= x_count > 750;
            divide_start <= 1;
            divide_start_pad <= 1;
        end else begin
            divide_start <= 0;
            divide_start_pad <= 0;
        end
    end
 end

// code to write NTSC data to video memory

wire [18:0] ntsc_addr;
wire [35:0] ntsc_data;
wire        ntsc_we;
ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh_delay[74:72], dv_delay[24], rgb,
                 ntsc_addr, ntsc_data, ntsc_we, switch[6]);
```

```
  // 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 = ~switch[7];
  wire  my_we = sw_ntsc ? (hcount[0]==1'b0) : 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

  reg [17:0]    pixel;
  reg   bl,hs,vs;

  always @(posedge clk)
    begin
        pixel <= switch[0] ? {hcount[8:6],5'b0} : ( pred_crosshair_pix ? pred_crosshair_pix :
motor_crosshair_pix ? motor_crosshair_pix : cross_pix ? cross_pix : pad_pix ? pad_pix :
(in_frame ? vr_pixel : 18'b0)) | top_border_pix | bot_border_pix | defense_line_pix;
        bl <= blank;
        hs <= hsync;
        vs <= vsync;
    end
```

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

        // wires for ai
        wire [10:0] predicted_x;
        wire [9:0] predicted_y;
        reg [10:0] display_predicted_x;
        reg [9:0] display_predicted_y;
        reg [9:0] display_motor_x;
        reg [8:0] display_motor_y;
        wire [10:0] motor_x;
        wire [9:0] motor_y;
        wire [10:0] puck_x;
        wire [9:0] puck_y;
        wire [10:0] x_speed;
        wire [9:0] y_speed;

        wire [9:0] paddle_x;
        wire [8:0] paddle_y;

        // puck "0" value (x and y) for ai is video frame edge + edge width (10) + puck
radius
        assign puck_x = cross_x - X_CAMERA_OFFSET_PIXELS - BUFFER_PIXELS;
        assign puck_y = cross_y - TOP_BORDER_PIXELS_Y;
        // same thing for paddle for motion control, except without puck radius
        assign paddle_x = pad_x - X_CAMERA_OFFSET_PIXELS - BUFFER_PIXELS;
        assign paddle_y = pad_y - TOP_BORDER_PIXELS_Y;

        // FIXED PADDLE DESIRED POSITION (to change back to ai output ,uncomment
and remove 10/9 bit values)
```

```
        always @(posedge clk) begin
                display_predicted_x <= {predicted_x[9:0]} + X_CAMERA_OFFSET_PIXELS +
BUFFER_PIXELS;
                display_predicted_y <= {predicted_y[8:0]} + TOP_BORDER_PIXELS_Y;
                if (motor_valid) begin
                        display_motor_x <= motor_x + X_CAMERA_OFFSET_PIXELS +
BUFFER_PIXELS;
                        display_motor_y <= motor_y + TOP_BORDER_PIXELS_Y;
                end
        end
        wire done_debug;
        wire pred_valid_debug;
        wire tneg_debug;
        wire tpos_debug;
        wire treg_debug;
        assign user4[30] = done_debug;
        assign user4[29] = pred_valid_debug;
        assign user4[28] = puck_data_valid;
        assign user4[27] = tneg_debug;
        assign user4[26] = tpos_debug;
        assign user4[25] = treg_debug;
        wire [1:0] ai_mode;
        ai_toplevel #(
                .MAX_X(700),
                .MAX_Y(446),
                .MAX_SPEED(100),
                .X_INTERCEPT(MOTOR_MIN_X),
                .PUCK_RADIUS(PUCK_RADIUS_PIXELS),
                .NUM_NEXT_POSITIONS(30),
                .NUM_NEXT_POSITIONS_WIDTH(8),
                .NUM_AHEAD_TO_PREDICT(10)
        ) ai (
                .rst(motor_start),
                .clk(clk),
                .puck_x(puck_x),
                .puck_y(puck_y),
                .paddle_x(paddle_x),
                .paddle_y(paddle_y),
                .puck_visible(puck_visible),
```

```
            .data_valid(puck_data_actually_valid),
            .predicted_x(predicted_x),
            .predicted_y(predicted_y),
            .done_debug(done_debug),
            .tneg(tneg_debug),
            .tpos(tpos_debug),
            .treg(treg_debug),
            .mode_reg(ai_mode),
            .pred_valid(pred_valid_debug),
            .x_speed(x_speed),
            .y_speed(y_speed),
            .motor_x(motor_x),
            .motor_y(motor_y),
            .motor_valid(motor_valid)
        );

        assign user4[31] = motor_valid;

        crosshair #(.COLOR({6'h00, 6'hFF, 6'h00})) pred_crosshair (.hcount(hcount),
.x(display_predicted_x), .vcount(vcount), .y(display_predicted_y),
.pixel(pred_crosshair_pix));
        crosshair #(.COLOR({6'h00, 6'h00, 6'hFF})) motor_crosshair (.hcount(hcount),
.x(display_motor_x), .vcount(vcount), .y(display_motor_y), .pixel(motor_crosshair_pix));


        wire motor_up_edge;
        wire motor_down_edge;
        reg old_motor_up;
        reg old_motor_down;

        assign motor_up_edge = motor_up & ~old_motor_up;
        assign motor_down_edge = motor_down & ~old_motor_down;

        // DEBUG: test boundaries
        reg [8:0] motor_test_y;
        always @(posedge clk) begin
            old_motor_up <= motor_up;
            old_motor_down <= motor_down;
            if (motor_start) begin
```

```
                motor_test_y <= 300;
            end else if (motor_up_edge) begin
                motor_test_y <= 9'd28;
            end else if (motor_down_edge) begin
                motor_test_y <= 9'h180;
            end
    end

    assign dispdata[63:48] = paddle_x;
    assign dispdata[47:32] = puck_x;

    wire [8:0] motor_y_bounded;

    assign motor_y_bounded = (motor_y > MOTOR_MAX_Y) ? MOTOR_MAX_Y:
            (motor_y < MOTOR_MIN_Y) ? MOTOR_MIN_Y : motor_y[8:0];

wire pulse_done_x_debug,
     pulse_done_y_debug;
wire [3:0] debug_out;

    motion_module motor_shit (
            .desired_x(motor_x[9:0]),
            .desired_y(motor_y_bounded),
            .current_x(paddle_x),
            .current_y(paddle_y),
            .reset(1'b0),
            .clock(clk),
            .kill(motor_kill),
            .on_signal(motor_start),
            .data_valid(motor_valid),
            .direction_x1(user4[0]),
            .direction_x2(user4[1]),
            .direction_y(user4[3]),
            .MS1(),
            .MS2(),
            .MS3(),
            .step_pulse_x(user4[2]),
            .step_pulse_y(user4[4]),
            .count_pul(user4[8]),
```

```
                .x_val_wired(),
                .y_val_wired(),
                .state_x(),
                .state_y(),
                .pulse_done_x(pulse_done_x_debug),
                .pulse_done_y(pulse_done_y_debug),
            .debug(debug_out)
        );

    // debugging
    assign led = {~puck_visible, ~motor_valid, ~ai_mode, ~debug_out}; //DEBUG

endmodule

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

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

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

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

  // vertical: 806 lines total
  // display 768 lines
```

```
  wire     vsyncon,vsyncoff,vreset,vblankon;
  assign   vblankon = hreset & (vcount == 767);
  assign   vsyncon = hreset & (vcount == 776);
  assign   vsyncoff = hreset & (vcount == 782);
  assign   vreset = hreset & (vcount == 805);

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

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

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

//////////////////////////////////////////////////////////////////////////
// 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, is_puck);

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


  //forecast hcount & vcount 4 clock cycles ahead to get data from ZBT
  wire [10:0] hcount_f = (hcount >= 1052) ? (hcount - 1052) : (hcount + 4);
  wire [9:0] vcount_f = (hcount >= 1052) ? ((vcount == 805) ? 0 : vcount + 1) : vcount;

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

  wire [1:0]      hc2 = hcount[0];
```

```
  reg [17:0]     vr_pixel;
  reg [35:0]     vr_data_latched;
  reg [35:0]     last_vr_data;

  always @(posedge clk) begin
   vr_data_latched <= (hc2==1'b1) ? vram_read_data : vr_data_latched;
          last_vr_data <= (hc2 == 1'b0) ? vr_data_latched : last_vr_data;
         end

  always @(*)                    // each 36-bit word from RAM is decoded to 4 bytes
   case (hc2)
     1'b1: vr_pixel = last_vr_data[17:0];
     1'b0: vr_pixel = last_vr_data[35:18];
   endcase

endmodule // vram_display

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

module delayN (clk,in,out);
  input clk;
  input in;
  output out;

        parameter NDELAY = 3;

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

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

endmodule // delayN

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

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

```
//
// 6.111 FPGA Labkit -- ZBT RAM clock generation
//
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////////
//
// This module generates deskewed clocks for driving the ZBT SRAMs and FPGA
// registers. A special feedback trace on the labkit PCB (which is length
// matched to the RAM traces) is used to adjust the RAM clock phase so that
// rising clock edges reach the RAMs at exactly the same time as rising clock
// edges reach the registers in the FPGA.
//
// The RAM clock signals are driven by DDR output buffers, which further
// ensures that the clock-to-pad delay is the same for the RAM clocks as it is
// for any other registered RAM signal.
//
// When the FPGA is configured, the DCMs are enabled before the chip-level I/O
// drivers are released from tristate. It is therefore necessary to
// artificially hold the DCMs in reset for a few cycles after configuration.
// This is done using a 16-bit shift register. When the DCMs have locked, the
// <lock> output of this mnodule will go high. Until the DCMs are locked, the
// ouput clock timings are not guaranteed, so any logic driven by the
// <fpga_clock> should probably be held inreset until <locked> is high.
//
//////////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

        assign ref_clk = ref_clock;

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

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

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

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

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

```
  // synthesis attribute PHASE_SHIFT of ext_dcm is 0

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


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

  assign locked = lock1 && lock2;

endmodule
```

Machine Vision Code

```
//////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// This module drives the labkit hex displays and shows the value of
// 8 bytes (16 hex digits) on the displays.
//
// 24-Sep-05 lke: updated to use new reset-once state machine, remove clear
```

```
// 02-Nov-05 Ike: updated to make it completely synchronous
//
// Inputs:
//
//   reset      - active high
//   clock_27mhz - the synchronous clock
//   data       - 64 bits; each 4 bits gives a hex digit
//
// Outputs:
//
//    disp_*    - display lines used in the 6.111 labkit (rev 003 & 004)
//
//////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data_in,
                      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_in;                   // 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 [5:0] count;
  reg [7:0] reset_count;
//   reg               old_clock;
  wire     dreset;
  wire     clock = (count<27) ? 0 : 1;
```

```
  always @(posedge clock_27mhz)
    begin
          count <= reset ? 0 : (count==53 ? 0 : count+1);
          reset_count <= reset ? 100 : ((reset_count==0) ? 0 : reset_count-1);
//        old_clock <= clock;
    end

  assign dreset = (reset_count != 0);
  assign disp_clock = ~clock;
  wire   clock_tick = ((count==27) ? 1 : 0);
//   wire   clock_tick = clock & ~old_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
  reg [63:0] data;

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

  always @(posedge clock_27mhz)
    if (clock_tick)
      begin
          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
        state <= state+1;
      end

  8'h04:
```

```
    begin
      // Setup the control register
      disp_rs <= 1'b1; // Select 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
      data <= data_in;
      state <= state+1;
    end

8'h06:
    begin
      // Load the user's dot data into the dot reg, char by char
      disp_rs <= 1'b0;                        // Select the dot register
      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
              data <= data_in;
              dot_index <= 39;
          end
      else
        dot_index <= dot_index-1;         // else loop thru all dots
```

```
                end

        endcase // casex(state)
    end

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

// module which holds the h,s,v threshold settings which can be reset on the
// fly in order to accomodate a wide range of lighting situations. This also
// helps in determining the best bounds for identifying each object as the user
// can experiment with it

// inputs:
//  - minmax_sel controls whether the maximum or minimum value is being set.
//    0 for min and 1 for max
//  - hsv_sel controls which of the h,s,v parameter is being set
//    0 for h1, 1 for h2, 2 for s, and 3 for v
//  - up and down controls whether the current value should be incremented or
//    decremented should be a debounced input

module hsv_threshold_select
#(parameter HMIN1 = 0, HMAX1 = 10,
          HMIN2 = 230, HMAX2 = 255,
          SMIN = 140, SMAX = 255,
          VMIN = 125, VMAX = 255)
(input clk, reset, enable,
        up, down,
        minmax_sel,
   input [1:0] hsv_sel, // 0:h1 1:h2 1:s 2:v
   output reg [7:0] h_max1, h_max2, s_max, v_max,
                h_min1, h_min2, s_min, v_min); // 2 h_max/h_min to account
                                               // for circular nature of
                                               // hue in HSV coordinates

   parameter H1SEL = 0,
             H2SEL = 1,
             SSEL = 2,
             VSEL = 3,
```

```
        VAL = 5;

reg up_old, down_old; // used for identifying a pulse
wire up_pulse, down_pulse;

assign up_pulse = up && ~up_old;
assign down_pulse = down && ~down_old;


always @(posedge clk) begin
   if (reset) begin
      h_min1 <= HMIN1;
      h_max1 <= HMAX1;
      h_min2 <= HMIN2;
      h_max2 <= HMAX2;
      s_min <= SMIN;
      s_max <= SMAX;
      v_min <= VMIN;
      v_max <= VMAX;
   end

   else begin
      if (enable) begin

         up_old <= up;
         down_old <= down;

         //TODO pulse detection
         case (hsv_sel)
            H1SEL:
               if (minmax_sel)
                  h_max1 <= (up_pulse && (h_max1 <= 255 - VAL)) ? (h_max1 + VAL) :
                        (down_pulse && (h_max1 >= VAL)) ? (h_max1 - VAL) : h_max1;
               else
                  h_min1 <= (up_pulse && (h_min1 <= 255 - VAL)) ? (h_min1 + VAL) :
                        (down_pulse && (h_min1 >= VAL)) ? (h_min1 - VAL) : h_min1;
            H2SEL:
               if (minmax_sel)
                  h_max2 <= (up_pulse && (h_max2 <= 255 - VAL)) ? (h_max2 + VAL) :
```

```
                    (down_pulse && (h_max2 >= VAL)) ? (h_max2 - VAL) : h_max2;
              else
                h_min2 <= (up_pulse && (h_min2 <= 255 - VAL)) ? (h_min2 + VAL) :
                    (down_pulse && (h_min2 >= VAL)) ? (h_min2 - VAL) : h_min2;
            SSEL:
              if (minmax_sel)
                s_max <= (up_pulse && (s_max <= 255 - VAL)) ? (s_max + VAL) :
                    (down_pulse && (s_max >= VAL)) ? (s_max - VAL) : s_max;
              else
                s_min <= (up_pulse && (s_min <= 255 - VAL)) ? (s_min + VAL) :
                    (down_pulse && (s_min >= VAL)) ? (s_min - VAL) : s_min;
            VSEL:
              if (minmax_sel)
                v_max <= (up_pulse && (v_max <= 255 - VAL)) ? (v_max + VAL) :
                    (down_pulse && (v_max >= VAL)) ? (v_max - VAL) : v_max;
              else
                v_min <= (up_pulse && (v_min <= 255 - VAL)) ? (v_min + VAL) :
                    (down_pulse && (v_min >= VAL)) ? (v_min - VAL) : v_min;
          endcase
        end
      end
    end


endmodule
//
// 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'd152; // center the display on a 1024 x 768 display
   parameter   ROW_START = 10'd20;

   // 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 && ~old_dv; // if data valid, write it
      old_frame <= frame;
      even_odd = frame_edge ? ~even_odd : even_odd;

      if (!frame) begin
          col <= fvh[0] ? COL_START :
                  (!fvh[1] && dv && (col < 1024)) ? col + 1 : col;
          row <= fvh[1] ? ROW_START :
                  (fvh[0] && (row < 768)) ? row + 1 : row;
          vdata <= dv ? 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[0], we[1]};
     eo_delay <= {eo_delay[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]};

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

   reg [18:0] ntsc_addr;
   reg [35:0] ntsc_data;
   wire       ntsc_we = (we_edge & (x_delay[10]==1'b1));

   always @(posedge clk)
     if ( ntsc_we )
       begin
            ntsc_addr <= myaddr;
            ntsc_data <= mydata;
       end

endmodule // ntsc_to_zbt

// The divider module divides one number by another. It
// produces a signal named "ready" when the quotient output
// is ready, and takes a signal named "start" to indicate
// the the input dividend and divider is ready.
```

```
// sign -- 0 for unsigned, 1 for twos complement

// It uses a simple restoring divide algorithm.
// http://en.wikipedia.org/wiki/Division_(digital)#Restoring_division
//
// Author Logan Williams, updated 11/25/2018 gph

module obj_divider #(parameter WIDTH = 8)
 (input clk, sign, start,
  input [WIDTH-1:0] dividend,
  input [WIDTH-1:0] divider,
  output reg [WIDTH-1:0] quotient,
  output [WIDTH-1:0] remainder,
  output ready);

  reg [WIDTH-1:0]  quotient_temp;
  reg [WIDTH*2-1:0] dividend_copy, divider_copy, diff;
  reg negative_output;

  assign remainder = (!negative_output) ?
          dividend_copy[WIDTH-1:0] : ~dividend_copy[WIDTH-1:0] + 1'b1;

  reg [5:0] bit = 0;
  reg del_ready = 1;
  assign ready = (bit==0) & ~del_ready;

  wire [WIDTH-2:0] zeros = 0;
  initial bit = 0;
  initial negative_output = 0;
  always @( posedge clk ) begin
    del_ready <= (bit==0);
    if( start ) begin

      bit = WIDTH;
      quotient = 0;
      quotient_temp = 0;
      dividend_copy = (!sign || !dividend[WIDTH-1]) ?
                  {1'b0,zeros,dividend} :
                  {1'b0,zeros,~dividend + 1'b1};
```

```
        divider_copy = (!sign || !divider[WIDTH-1]) ?
                    {1'b0,divider,zeros} :
                    {1'b0,~divider + 1'b1,zeros};

        negative_output = sign &&
                     ((divider[WIDTH-1] && !dividend[WIDTH-1])
                      ||(!divider[WIDTH-1] && dividend[WIDTH-1]));
      end
      else if ( bit > 0 ) begin
        diff = dividend_copy - divider_copy;
        quotient_temp = quotient_temp << 1;
        if( !diff[WIDTH*2-1] ) begin
          dividend_copy = diff;
          quotient_temp[0] = 1'd1;
        end
        quotient = (!negative_output) ?
                quotient_temp :
                ~quotient_temp + 1'b1;
        divider_copy = divider_copy >> 1;
        bit = bit - 1'b1;
      end
    end
endmodule
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Kevin Zheng Class of 2012
//           Dept of Electrical Engineering &  Computer Science
//
// 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

                divider hue_div1(
                .clk(clock),
                .dividend(s_top),
```

```
    .divisor(s_bottom),
    .quotient(s_quotient),
// note: the "fractional" output was originally named "remainder" in this
    // file -- it seems coregen will name this output "fractional" even if
    // you didn't select the remainder type as fractional.
    .fractional(s_remainder),
    .rfd(s_rfd)
    );
    divider 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
module crosshair #(
   parameter OFFSET = 30,
   parameter COLOR = 18'b111111_111111_111111 //white default color
)
(
   input [10:0] hcount, x,
   input [9:0] vcount, y,
   output reg [17:0] pixel
);

   always @ * begin
      if (hcount == 0 | hcount == 1023 | vcount == 0 | vcount == 767 |
         // Note: assumed that y > OFFSET and x > OFFSET always true
         (hcount == x && vcount > (y - OFFSET) && vcount < (y + OFFSET))|
         (vcount == y && hcount > ( x - OFFSET) && hcount < (x + OFFSET)))

         pixel = COLOR;

      else pixel = 0;
```

```
   end

endmodule

module bound_line #(
   parameter COLOR = 18'b111111_111111_111111 //white default color
)
(
   input [9:0] vcount, y,
   output reg [17:0] pixel
);
   always @ * begin
      if (vcount == y)
         pixel = COLOR;
      else
         pixel = 0;
   end
endmodule


//
// 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
```

```
        /************************************************************************
**
** 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

/* 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




//
// 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
```

AI Code

```
module ai_toplevel #( parameter
        NUM_NEXT_POSITIONS = 20,
        NUM_NEXT_POSITIONS_WIDTH = 5,
        NUM_AHEAD_TO_PREDICT = 15,
        X_INTERCEPT = 200,
        PUCK_RADIUS = 37,
        MAX_X = 700,
        MAX_Y = 446,
        MAX_SPEED = 100
) ( rst, clk, puck_x, puck_y, paddle_x, paddle_y, puck_visible, data_valid, predicted_x,
predicted_y, x_speed, y_speed, motor_x, motor_y, motor_valid, done_debug, pred_valid,
tpos, tneg, treg, mode_reg);
        input wire clk, rst;
        input wire [10:0] puck_x;
        input wire [9:0] puck_y;
        input wire [10:0] paddle_x;
        input wire [9:0] paddle_y;
        input wire puck_visible, data_valid;

        output reg [10:0] predicted_x;
        output reg [9:0] predicted_y;
        output reg [10:0] motor_x;
        output reg [9:0] motor_y;
        output reg motor_valid;
        output wire [10:0] x_speed;
        output wire [9:0] y_speed;
        output wire done_debug;

        parameter MAX_Y_AI = MAX_Y - 2*PUCK_RADIUS;
        parameter MAX_X_AI = MAX_X - 2*PUCK_RADIUS;
        parameter MID_Y = MAX_Y >> 1;

        function [10:0] get_ai_x;
                input [10:0] raw_x;
                get_ai_x = (raw_x < PUCK_RADIUS) ? 0 : raw_x - PUCK_RADIUS;
        endfunction

        function [10:0] get_raw_x;
                input [10:0] ai_x;
```

```
        get_raw_x = ai_x + PUCK_RADIUS;
endfunction

function [9:0] get_ai_y;
        input [9:0] raw_y;
        get_ai_y = (raw_y < PUCK_RADIUS) ? 0 : raw_y - PUCK_RADIUS;
endfunction

function [9:0] get_raw_y;
        input [9:0] ai_y;
        get_raw_y = ai_y + PUCK_RADIUS;
endfunction

reg [NUM_NEXT_POSITIONS_WIDTH-1:0] counter;
reg old_pred_valid;
wire [10:0] pred_x;
wire [9:0] pred_y;
output wire pred_valid;
output wire tneg;
output wire tpos;
output wire treg;
wire x_dir, y_dir;

trajectory_predictor # (
        .MAX_X (MAX_X_AI),
        .MAX_Y(MAX_Y_AI),
        .MAX_SPEED(MAX_SPEED),
        .NUM_NEXT_POSITIONS(NUM_NEXT_POSITIONS),
        .NUM_NEXT_POSITIONS_WIDTH(NUM_NEXT_POSITIONS_WIDTH)
) predictor (
        .clk(clk),
        .puck_x(get_ai_x(puck_x)),
        .puck_y(get_ai_y(puck_y)),
        .puck_visible(puck_visible),
        .data_valid(data_valid),
        .next_x(pred_x),
        .next_y(pred_y),
        .next_valid(pred_valid),
        .x_speed(x_speed),
```

```
        .y_speed(y_speed),
        .x_dir(x_dir),
        .y_dir(y_dir)
);

wire ep_valid;
wire [8:0] ep_y;
wire ep_found;

endpoint_calculator #(
        .X_INTERCEPT(X_INTERCEPT)
) ep_calc (
        // Inputs
        .reset(rst),
        .clk(clk),
        .x_trajectory(pred_x),
        .y_trajectory(pred_y),
        .trajectory_valid(pred_valid),
        // Outputs
        .done(done_debug),
        .tneg(tneg),
        .tpos(tpos),
        .treg(treg),
        .endpoint_valid(ep_valid),
        .endpoint_y(ep_y),
        .endpoint_found(ep_found)
);


reg motor_valid_defense;
reg [9:0] motor_x_defense;
reg [8:0] motor_y_defense;

wire pred_posedge;
wire pred_negedge;
assign pred_posedge = pred_valid && ~old_pred_valid;
assign pred_negedge = ~pred_valid && old_pred_valid;

// Set motor valid (has low pass filtering)
```

```
always @(posedge clk) begin
	motor_valid_defense <= puck_visible && ep_valid;
	if (ep_valid) begin
		if (ep_found) begin
			motor_y_defense <= get_raw_y(ep_y);
		end else begin
			motor_y_defense <= MID_Y;
		end
		motor_x_defense <= X_INTERCEPT;
	end
end

// Set predicted trajectory (1 per frame)
always @(posedge clk) begin
	old_pred_valid <= pred_valid;
	// counter for num values read in for a trajectory
	if (pred_valid) begin
		// new prediction is coming in
		counter <= counter + 1;
	end else if (pred_negedge) begin
		// reset counter
		counter <= 0;
	end

	if (counter == (NUM_AHEAD_TO_PREDICT - 1)) begin
		// about to get last position
		predicted_x <= get_raw_x(pred_x);
		predicted_y <= get_raw_y(pred_y);
	end
end

reg [8:0] y_offset;
reg [9:0] saved_paddle_x, saved_puck_x;
reg [8:0] saved_paddle_y, saved_puck_y;
reg [9:0] saved_x_speed;
reg [8:0] saved_y_speed;
reg saved_x_dir, saved_y_dir;
always @(posedge clk) begin
	if (data_valid) begin
```

```
                        y_offset <= (puck_y > paddle_y) ? puck_y - paddle_y : paddle_y -
puck_y;
                        saved_puck_x <= puck_x;
                        saved_puck_y <= puck_y;
                        saved_paddle_x <= paddle_x;
                        saved_paddle_y <= paddle_y;
                end
                if (pred_valid) begin
                        saved_x_speed <= x_speed;
                        saved_y_speed <= y_speed;
                        saved_x_dir <= x_dir;
                        saved_y_dir <= y_dir;
                end
        end

        reg [1:0] mode; // combinational
        output reg [1:0] mode_reg;
        always @(posedge clk) mode_reg <= mode;
        parameter AWAY = 2'b00;
        parameter STRIKE = 2'b01;
        parameter DEFENSE = 2'b10;

        // Set the mode (combinationally)
        always @(*) begin
                if (saved_paddle_x >= saved_puck_x) mode <= AWAY;
                else if (saved_x_speed <= 10 && saved_puck_x <= 300) mode <= STRIKE;
                else if (saved_x_dir==0 && y_offset < 100 ) mode <= STRIKE;
                else mode <= DEFENSE;
        end

        // Set the motor output
        always @(*) begin
                case (mode)
                        AWAY: begin
                                motor_valid <= puck_visible && pred_valid;
                                motor_x <= (saved_paddle_x > saved_puck_x) ?
saved_paddle_x + 20 : saved_paddle_x - 20;
                                motor_y <= saved_paddle_y;
                        end
```

```
                        STRIKE: begin
                                motor_valid <= puck_visible && pred_valid;
                                motor_x <= saved_puck_x;
                                motor_y <= saved_puck_y;
                        end
                        DEFENSE: begin
                                motor_valid <= motor_valid_defense;
                                motor_x <= motor_x_defense;
                                motor_y <= motor_y_defense;
                        end
                endcase
        end

endmodule

`timescale 1ns / 1ns
module endpoint_calculator #( parameter
                X_INTERCEPT = 200
) (
                // Inputs
                clk,
                reset,
                x_trajectory,
                y_trajectory,
                trajectory_valid,
                // Outputs
                done,
                tneg,
                tpos,
                treg,
                endpoint_valid,
                endpoint_y,
                endpoint_found
);

        input wire clk;
        input wire reset;
        input wire [9:0] x_trajectory;
        input wire [8:0] y_trajectory;
```

```
input wire trajectory_valid;

output reg endpoint_valid;
output reg endpoint_found;
output reg [8:0] endpoint_y;


reg old_traj_valid;
wire traj_valid_posedge;
wire traj_valid_negedge;
output wire tneg;
output wire tpos;
output wire treg;
assign treg = old_traj_valid;
assign tneg = traj_valid_negedge;
assign tpos = traj_valid_posedge;
assign traj_valid_posedge = trajectory_valid && (~old_traj_valid);
assign traj_valid_negedge = (~trajectory_valid) && old_traj_valid;

output reg done;
reg one_ep_valid;
reg [2:0] num_results; // counts 0->4
reg [2:0] num_valid; // 0->4
reg [8:0] trajectory_endpoint;

reg [8:0] y1, y2, y3, y4;

// read in points
always @(posedge clk) begin
        old_traj_valid <= trajectory_valid;
        if (reset) begin
                num_results <= 3;
                num_valid <= 0;
                done <= 0;
        end else if (traj_valid_posedge) begin
                // reset new trajectory
                done <= 0;
                one_ep_valid <= 0;
        end else if (trajectory_valid && ~one_ep_valid) begin
```

```
                        // looking for this trajectory's endpoint
                        if (x_trajectory < X_INTERCEPT) begin
                                // shift in a new value
                                trajectory_endpoint <= y_trajectory;
                                one_ep_valid <= 1;
                        end
                end else if (traj_valid_negedge) begin
                        // end of trajectory
                        done <= 1;
                        num_results <= (num_results == 3) ? 0 : num_results + 1;
                        num_valid <= (num_results == 3) ? one_ep_valid : num_valid +
one_ep_valid;
                        { y1, y2, y3, y4 } <= { trajectory_endpoint, y1, y2, y3 };
                end else begin
                        done <= 0;
                end
        end

        wire [10:0] endpoint_y_sum_all_valid;
        wire [10:0] endpoint_y_sum_one_invalid;
        wire [8:0] mid_value;
        assign endpoint_y_sum_all_valid = y1 + y2 + y3 + y4;
        assign endpoint_y_sum_one_invalid = y1 + y2 + y3 + mid_value;

        assign mid_value = (y4 >= y2 && y4 >= y3) ? ((y2 >= y3) ? y2 : y3) : // y4 is max
                (y2 >= y3 && y2 >= y3) ? ((y4 >= y3) ? y4 : y3) : // y2 is max
                (y4 >= y2) ? y4 : y2; // y3 is max

        // set output
        always @(posedge clk) begin
                endpoint_valid <= 0;
                if ((done==1) && (num_results == 3)) begin
                        endpoint_valid <= 1;
                        if (num_valid == 4) begin
                                endpoint_found <= 1;
                                endpoint_y <= endpoint_y_sum_all_valid >> 2;
                        end else if (num_valid == 3) begin
                                endpoint_found <= 1;
                                endpoint_y <= endpoint_y_sum_one_invalid >> 2;
```

```
                end else begin
                        endpoint_found <= 0;
                end
            end
        end
endmodule


`timescale 1ns / 1ns
// assuming never moves fast enough to do 2 x or 2 y bounces in same frame
module trajectory_predictor # (parameter
        NUM_NEXT_POSITIONS = 20,
        NUM_NEXT_POSITIONS_WIDTH = 5,
        MAX_X = 700,
        MAX_Y = 446,
        MAX_SPEED = 100
) (clk, puck_x, puck_y, puck_visible, data_valid, next_x, next_y, x_dir, y_dir, x_speed,
y_speed, next_valid);

        // I/O
        input wire clk;
        input wire data_valid, puck_visible;
        input wire [10:0] puck_x; // 10 x direction bits: 0 -> MAX_X
        input wire [9:0] puck_y; // 9 y direction bits: 0 -> MAX_Y
        output reg [10:0] next_x;
        output reg [9:0] next_y;
        output reg next_valid;
        output reg [10:0] x_speed;
        output reg [9:0] y_speed;
        output reg x_dir, y_dir;

        // State used to track actual history
        reg [10:0] prev_frame_x; // previous x pos received
        reg [9:0] prev_frame_y; // previous y pos received
        reg [10:0] prev_frame_x_speed; // x speed going into previous frame
        reg [9:0] prev_frame_y_speed; // y speed going into previous frame
        reg prev_frame_x_dir; // x direction (+ or -) going into previous frame
        reg prev_frame_y_dir; // y direction (+ or -) going into previous frame
```

```
// State used for predictions
reg [10:0] cur_x_pred;
reg [9:0] cur_y_pred;
reg cur_x_dir;
reg cur_y_dir;
reg [NUM_NEXT_POSITIONS_WIDTH-1:0] counter;

// Reset conditions
reg prev_puck_visible;
wire puck_reset;
wire [10:0] x_speed_to_use;
wire [9:0] y_speed_to_use;
wire x_dir_to_use;
wire y_dir_to_use;

// Intermediate wires
wire [10:0] calc_x_speed;
wire [9:0] calc_y_speed;
wire calc_x_dir;
wire calc_y_dir;
wire [10:0] pred_x;
wire [9:0] pred_y;
wire pred_x_dir, pred_y_dir;

assign puck_reset = puck_visible & ~prev_puck_visible;
assign x_speed_to_use = (puck_reset) ? 10'b0 : calc_x_speed;
assign y_speed_to_use = (puck_reset) ? 9'b0 : calc_y_speed;
assign x_dir_to_use = (puck_reset) ? 0 : calc_x_dir;
assign y_dir_to_use = (puck_reset) ? 0 : calc_y_dir;

always @(posedge clk) begin
        x_speed <= prev_frame_x_speed;
        y_speed <= prev_frame_y_speed;
        x_dir <= prev_frame_x_dir;
        y_dir <= prev_frame_y_dir;
end

// Update current and prev state whenever there's a new frame
always @(posedge clk) begin
```

```
            prev_puck_visible <= puck_visible;
            next_valid <= 0;
            if (data_valid) begin
                // Save historical state
                prev_frame_x <= puck_x;
                prev_frame_y <= puck_y;
                prev_frame_x_speed <= x_speed_to_use; // calculated from puck,
saved pos/speed
                prev_frame_y_speed <= y_speed_to_use; // calculated from puck,
saved pos/speed
                prev_frame_x_dir <= x_dir_to_use;
                prev_frame_y_dir <= y_dir_to_use;

                // Set state for predictions
                counter <= 0;
                cur_x_pred <= puck_x;
                cur_y_pred <= puck_y;
                cur_x_dir <= x_dir_to_use;
                cur_y_dir <= y_dir_to_use;
            end else if (counter < NUM_NEXT_POSITIONS) begin
                next_valid <= 1;
                counter <= counter + 1; // Increment counter
                next_x <= pred_x;
                next_y <= pred_y;
                cur_x_dir <= pred_x_dir;
                cur_y_dir <= pred_y_dir;
                cur_x_pred <= pred_x;
                cur_y_pred <= pred_y;
            end
        end

        // Module that calculates initial velocity and speed of the puck
        velocity_calculator #(
            .MAX_X(MAX_X),
            .MAX_Y(MAX_Y),
            .MAX_SPEED(MAX_SPEED)
        ) vel_calc (
            .x(puck_x),
            .y(puck_y),
```

```
                .prev_x(prev_frame_x),
                .prev_y(prev_frame_y),
                .prev_x_speed(prev_frame_x_speed),
                .prev_y_speed(prev_frame_y_speed),
                .prev_x_dir(prev_frame_x_dir),
                .prev_y_dir(prev_frame_y_dir),
                .cur_x_speed(calc_x_speed),
                .cur_y_speed(calc_y_speed),
                .cur_x_dir(calc_x_dir),
                .cur_y_dir(calc_y_dir)
        );

        // Need to update x and y dir per prediction in case of bounces
        next_position_calculator #(
                .MAX_X(MAX_X),
                .MAX_Y(MAX_Y),
                .MAX_SPEED(MAX_SPEED)
        ) next_pos_calc (
                .x(cur_x_pred),
                .y(cur_y_pred),
                .x_speed(prev_frame_x_speed),
                .y_speed(prev_frame_y_speed),
                .x_dir(cur_x_dir),
                .y_dir(cur_y_dir),
                .next_x(pred_x),
                .next_y(pred_y),
                .next_x_dir(pred_x_dir),
                .next_y_dir(pred_y_dir)
        );
endmodule

module velocity_calculator #( parameter
        MAX_X = 700,
        MAX_Y = 446,
        MAX_SPEED = 100
) (x, y, prev_x, prev_y, prev_x_speed, prev_y_speed, prev_x_dir, prev_y_dir, cur_x_speed,
cur_y_speed, cur_x_dir, cur_y_dir);

        input wire [10:0] x, prev_x, prev_x_speed;
```

```
        input wire [9:0] y, prev_y, prev_y_speed;
        input wire prev_x_dir, prev_y_dir;

        output reg [10:0] cur_x_speed;
        output reg [9:0] cur_y_speed;
        output reg cur_x_dir, cur_y_dir;

        wire [10:0] x_speed_bounce;
        wire [10:0] x_speed_no_bounce;
        wire [10:0] x_error_bounce;
        wire [10:0] x_error_no_bounce;
        wire [9:0] y_speed_bounce;
        wire [9:0] y_speed_no_bounce;
        wire [9:0] y_error_bounce;
        wire [9:0] y_error_no_bounce;
        wire x_dir_no_bounce;
        wire y_dir_no_bounce;

        wire x_bounced;
        wire y_bounced;

        assign x_speed_bounce = (prev_x_dir == 0) ? (x + prev_x) : ((MAX_X - x) + (MAX_X
- prev_x));
        assign y_speed_bounce = (prev_y_dir == 0) ? (y + prev_y) : ((MAX_Y - y) + (MAX_Y
- prev_y));
        assign x_speed_no_bounce = x_dir_no_bounce ? (x - prev_x) : (prev_x - x);
        assign y_speed_no_bounce = y_dir_no_bounce ? (y - prev_y) : (prev_y - y);
        assign x_dir_no_bounce = (x > prev_x);
        assign y_dir_no_bounce = (y > prev_y);

        // For bounce, we know it's least error when in same direction.
        assign x_error_bounce = (x_speed_bounce > prev_x_speed) ? (x_speed_bounce -
prev_x_speed) : (prev_x_speed - x_speed_bounce);
        assign y_error_bounce = (y_speed_bounce > prev_y_speed) ? (y_speed_bounce -
prev_y_speed) : (prev_y_speed - y_speed_bounce);

        // For no bounce, have to check if it's the same direction or not.
        assign x_error_no_bounce = (x_dir_no_bounce == prev_x_dir) ?
                ((x_speed_no_bounce > prev_x_speed) ? (x_speed_no_bounce -
```

```
prev_x_speed) : (prev_x_speed - x_speed_no_bounce)) :
                (x_speed_no_bounce + prev_x_speed);
        assign y_error_no_bounce = (y_dir_no_bounce == prev_y_dir) ?
                ((y_speed_no_bounce > prev_y_speed) ? (y_speed_no_bounce -
prev_y_speed) : (prev_y_speed - y_speed_no_bounce)) :
                (y_speed_no_bounce + prev_y_speed);

        // Decide whether bounced based on error
        assign x_bounced = (x_error_bounce < x_error_no_bounce);
        assign y_bounced = (y_error_bounce < y_error_no_bounce);

        always @(*) begin
                cur_x_dir <= (x_bounced) ? ~prev_x_dir : x_dir_no_bounce;
                cur_y_dir <= (y_bounced) ? ~prev_y_dir : y_dir_no_bounce;
                cur_x_speed <= (x_bounced) ? ((x_speed_bounce > MAX_SPEED) ?
MAX_SPEED : x_speed_bounce) : ((x_speed_no_bounce > MAX_SPEED) ? MAX_SPEED :
x_speed_no_bounce);
                cur_y_speed <= (y_bounced) ? ((y_speed_bounce > MAX_SPEED) ?
MAX_SPEED : y_speed_bounce) : ((y_speed_no_bounce > MAX_SPEED) ? MAX_SPEED :
y_speed_no_bounce);
        end

endmodule

module next_position_calculator #( parameter
        MAX_X = 700,
        MAX_Y = 446,
        MAX_SPEED = 100
) (x, y, x_speed, y_speed, x_dir, y_dir, next_x, next_y, next_x_dir, next_y_dir);
        input wire [10:0] x, x_speed;
        input wire [9:0] y, y_speed;
        input wire x_dir, y_dir;

        output reg [10:0] next_x;
        output reg [9:0] next_y;
        output reg next_x_dir, next_y_dir;

        always @(*) begin
                if (x_dir==0) begin
```

```
        // moving down
        if (x < x_speed) begin
                // switch directions, bounce off bottom
                next_x <= x_speed - x;
                next_x_dir <= 1;
        end else begin
                // still going down
                next_x <= x - x_speed;
                next_x_dir <= 0;
        end
end else begin
        // moving up
        if (x > MAX_X - x_speed) begin
                // switch directions, bounce off top
                next_x <= (MAX_X - x_speed) + (MAX_X - x);
                next_x_dir <= 0;
        end else begin
                // still going up
                next_x <= x + x_speed;
                next_x_dir <= 1;
        end
end
if (y_dir==0) begin
        // moving down
        if (y < y_speed) begin
                // switch directions, bounce off bottom
                next_y <= y_speed - y;
                next_y_dir <= 1;
        end else begin
                // still going down
                next_y <= y - y_speed;
                next_y_dir <= 0;
        end
end else begin
        // moving up
        if (y > MAX_Y - y_speed) begin
                // switch directions, bounce off top
                next_y <= (MAX_Y - y_speed) + (MAX_Y - y);
                next_y_dir <= 0;
```

```
                end else begin
                        // still going up
                        next_y <= y + y_speed;
                        next_y_dir <= 1;
                end
        end
    end
endmodule
```

Motion Module Code

```
module motion_module(input [9:0] desired_x,
                        input [8:0] desired_y,
                    input [9:0] current_x,
                        input [8:0] current_y,
            input [9:0] puck_current_x,
                        input reset,
                        input clock,
                        input kill,
                        input on_signal,
                        input data_valid,
                        output direction_x1,
                        output direction_x2,
                        output direction_y,
                        output MS1,
                        output MS2,
                        output MS3,
                        output step_pulse_x,
                        output step_pulse_y,
                        output count_pul,
                        output [10:0] x_val_wired,
                        output [9:0] y_val_wired,
                        output [3:0] state_x,
                        output [3:0] state_y,
                        output pulse_done_x,
                        output pulse_done_y,
            output [3:0] debug);
```

```
        wire count_pulse_x;
        wire count_pulse_y;
        wire [10:0] x_val_wire;
        wire [9:0] y_val_wire;
        wire done_flag_x; // tells the FSM when movement is complete
        wire done_flag_y;
        wire [11:0] step_count_x;
        wire [11:0] step_count_y;
        wire [2:0] step_mode_x;
        wire [2:0] step_mode_y;
        wire start_signal_x;
        wire start_signal_y;
        wire MS1_x;
        wire MS2_x;
        wire MS3_x;
        wire stop_motor, below, above;

        OneUsTimer timer(.clock(clock), .count(count_pulse_x));

        Acel_Timer acel(.clock(clock),.count(count_pulse_y),.start(start_signal_y))

        Transition_y FSM_x(.clock(clock), .start(on_signal), .x_desired(desired_x),

.y_desired(desired_x),.current_y(current_x),

.pulse_done(data_valid),.motor_stop(stop_motor_x),

.reset(reset),.step_count(step_count_x),.step_mode(step_mode_x),
                                                .start_pulse(start_signal_x),
.direction(direction_x1),.state_x(state_x));

        Transition_y FSM_y(.clock(clock), .start(on_signal), .x_desired(x_val_wire),

.y_desired(desired_y),.current_y(current_y),

.pulse_done(data_valid),.motor_stop(stop_motor),.reset(reset),.step_count(step_count_
y),.step_mode(step_mode_y),
                                                .start_pulse(start_signal_y),
```

```
.direction(direction_y),.state_x(state_y));



        assign direction_x2 = ~direction_x1;

   assign below = (current_y > desired_y);
   assign above = (current_y <= desired_y);

   wire motor_desired_reached = ((current_y > desired_y) ? (((current_y - desired_y) < 10
)? 1'b1 : 0) :
                                 (((desired_y - current_y) < 10) ? 1'b1 : 0));
        wire motor_desired_reached_x = ((current_x > desired_x) ? (((current_x -
desired_x) < 10 )? 1'b1 : 0) :
                                 (((desired_x - current_x) < 10) ? 1'b1 : 0));
   assign stop_motor_y = (((current_y >= 9'h170) && direction_y) || ((current_y <= 9'h52)
&& ~direction_y) || motor_desired_reached)  ? 1'b1 : 1'b0;
   assign stop_motor_x = (((current_x >= 9'hFF ) && direction_x1) || ((current_x <= 9'h70)
&& ~direction_x1) || motor_desired_reached_x) ? 1'b1 : 1'b0;

        pulse_generator p_gen_x(.clock(clock), .kill(kill),.start(start_signal_x),
.count(count_pulse_x),.step_count(12'hFFF),

.step_mode(step_mode_x),.pulse(step_pulse_x),.done(done_flag_x),.MS1(MS1_x),
.MS2(MS2_x),.MS3(MS3_x));

        pulse_generator p_gen_y(.clock(clock),
.kill(stop_motor_y),.start(start_signal_y),.count(count_pulse_y),.step_count(12'hFFF),

.step_mode(step_mode_y),.pulse(step_pulse_y),.done(done_flag_y),.MS1(MS1_y),
.MS2(MS2_y),.MS3(MS3_y));

        assign MS1 = MS1_x;
        assign MS2 = MS2_x;
        assign MS3 = MS3_x;
        assign count_pul = start_signal_x;
        assign x_val_wired = x_val_wire;
        assign y_val_wired = y_val_wire;
```

```
   assign debug = {above, below, done_flag_y, stop_motor_y};
   assign pulse_done_x = stop_motor;
        assign pulse_done_y = done_flag_y;

endmodule


Module pulse_generator(input clock, input kill, input start, input count, input[11:0]
step_count, input [2:0] step_mode,
                                                        output pulse, output done,output
MS1, output MS2, output MS3);
        // This module will be used to increment the angle of the motor when we are in
acceleration mode
        // This will receive a start signal for every start sequence and continue the steps
for that step count
        // Maximum steps to be done is 2000 or 7D0

        // Step Modes MS1_MS2_MS3
                parameter FULL =                3'b000;
                parameter HALF =                3'b100;
                parameter QUARTER =             3'b010;
                parameter EIGHT     =           3'b110;
                parameter SIXTEENTH =    3'b111;

                reg start_flag;
                reg MS1_reg;
                reg MS2_reg;
                reg MS3_reg;
                reg [11:0] step_counter;
                reg done_reg;
                reg pulser;
                reg pulse_flag;


                        always @(posedge clock) begin
                                if(kill) begin
                                                pulser <=0; end
                                else begin
                                        if(start) begin
```

```
                                        start_flag <=1;
                                        done_reg <= 0;
                                        pulse_flag <= 0;
                                        step_counter <= step_count; end
                                    else if(count) begin
                                        if(pulse_flag) begin
                                            pulser <= 0;
                                            pulse_flag <= 0; end
                                        else if (start_flag) begin
                                            MS1_reg <= step_mode[2];
                                            MS2_reg <= step_mode[1];
                                            MS3_reg <= step_mode[0];
                                            start_flag <=0;end
                                        else if ((step_counter > 0))begin //&&(count))
begin
                                            pulser <= 1;
                                            step_counter <= step_counter -1;
                                            pulse_flag <=1; end
                                        else if (step_counter == 0) begin
                                            start_flag <=0;
                                            done_reg <= 1;
                                        end
                                    end
                                    end
                        end
                        assign done = done_reg;
                        assign MS1 = MS1_reg;
                        assign MS2 = MS2_reg;
                        assign MS3 = MS3_reg;
                        assign pulse = pulser;

                endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:07:08 12/04/2018
```

```
// Design Name:
// Module Name:    Transition_x
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module Transition_y(input clock, input start,
                    input [10:0] x_desired,
                    input [9:0] y_desired,
                    input [8:0] current_y,
                    input pulse_done,
                    input reset,
                    output [11:0] step_count,
                    output [2:0] step_mode,
                    output start_pulse,
                    output direction,
                    output motor_stop,
                    output [3:0] state_x);
          // this module will calculate how many steps we need to take to make it to our
next state
          // will hold our current position and plan the next movement
          // Reset will set out paddle to be at the midpoint of our board
          parameter MIDDLE_X = 350; // reset to 320; // reset value for X direction, midway
on the board
          parameter MIDDLE_Y = 223; // rest value for Y direction, midway on the board
          parameter BEGIN                    =         4'b0000;
          parameter NEW_STATES =          4'b0001;
          parameter MATH =                   4'b0010;
          parameter BEGIN_PULSE =         4'b0100;
          parameter END_PULSE      =         4'b1000;
          parameter POS_BOUND              =  16'D10;
```

```
        reg [10:0] x_new;
        reg [9:0]  y_new;
        reg [10:0] x_old;
        reg [9:0]  y_old;
        reg [10:0] x_diff;
        reg [9:0]  y_diff;
        reg direction_reg;
        reg start_pulse_reg;
        reg pulse_flag;
        reg [3:0] state;
        reg pulse_done_flag;
        reg stop_motor;

        always @(posedge clock) begin
          if (reset) begin
                x_new <= MIDDLE_X;
                x_old <= MIDDLE_X;
                y_new <= MIDDLE_Y;
                y_old <= MIDDLE_Y;
                state <= NEW_STATES; //TODO why does this go into NEW_STATES and
not BEGIN?
          end
          else if(start) begin
                state<= BEGIN;
                x_old <= MIDDLE_X;
                y_old <= MIDDLE_Y;
                pulse_done_flag <=1; end
          else begin
                case(state)
                      BEGIN: begin
                      if (pulse_done || pulse_done_flag) begin
                            state <= NEW_STATES;
                            pulse_done_flag <=0;end
                      else begin
                            state <= BEGIN; end
                      end

                      NEW_STATES: begin
```

```
        x_new <= x_desired; //TODO remove
        y_new <= y_desired;
        state <= MATH; end
MATH: begin
          if(current_y >= y_desired[8:0])begin
                direction_reg <= 1'b0;
                if (direction_reg == 1'b1) begin

                                    state <= BEGIN_PULSE; end

                            else state <= BEGIN;


                if ((current_y - y_desired[8:0]) < POS_BOUND)begin
                     stop_motor <=1;end
                else begin
                     stop_motor <=0;end
              end
              else begin
                direction_reg <= 1'b1;

                    if (direction_reg == 1'b0) begin

                                    state <= BEGIN_PULSE; end

                            else state <= BEGIN;
                if((y_desired[8:0] -current_y)< POS_BOUND) begin
                     stop_motor <=1; end
                else begin
                     stop_motor <=0;end
               end
             end
BEGIN_PULSE: begin
        x_old <= x_new; //TODO remove
        y_old[8:0] <= current_y[8:0];
        y_old[9] <= 1'b0;
        start_pulse_reg <= 1;
        state <= END_PULSE; end
END_PULSE: begin
```

```
                    start_pulse_reg <= 0;
                    state <= BEGIN; end
                endcase end

                end
                assign step_count[0] = 1'b0;
                assign step_count[10:1] = y_diff[9:0];
                assign step_count[11] = 1'b0;

                assign step_mode = 3'b0;
                assign start_pulse = start_pulse_reg;
                assign direction = direction_reg;
                assign state_x[3:0] = state[3:0];
                assign motor_stop = stop_motor;

endmodule


`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:35:17 12/07/2018
// Design Name:
// Module Name:    Accelerator
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module Acel_Timer(input clock, input start, output count);
```

```
                    // creates a pulse every 1us for the pulse to to the stepper driver to
work.
                    parameter TWO_MILI = 16'h6978;//34BC;//value equivalent to 1ms
with a 27mhz clock
                    parameter ONE_MILI = 16'h34BC;//1A5E;
                    parameter HALF_MILI = 16'h1A5E;//6978;
                    parameter MAX_MILI = 16'h4500;
                    reg [15:0] timer_value;
                    reg count_reg;
                    reg [15:0] variable_timer;
                    reg [15:0] counter;



                    always @(posedge clock) begin
                        if(start)begin
                            timer_value <= 16'h0;
                            variable_timer <= TWO_MILI;
                            counter <= 16'h0; end

                        else if (timer_value < variable_timer) begin
                            timer_value <= timer_value + 1;
                            count_reg <=0;end

                        else begin
                            timer_value <= 16'b0;
                            count_reg <=1;
                            if (variable_timer > MAX_MILI) begin
                                variable_timer <= variable_timer - 16'h4; end
                            counter <= counter +1;end
                        end

                        assign count = count_reg;
    endmodule
////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////
```