STRUGGLEBOT

6.111 Final Project Report

Rebecca Greene & Kelly Snyder

12/12/13

Abstract

The strugglebot project controls a robot using two distinct FPGAs that interact wirelessly, using the labkit, a vga screen, and an ceiling-mounted camera as an interface. The user can select if they wish to control the hexapod manually, or guide it to specific points to which it will move autonomously. When given a point, the labkit will generate a set of directions to move the robot towards a desired location. This location might be a point clicked on the screen with a mouse, or the location of a second object being tracked.

The robot itself is a hexapod, consisting of six legs attached to a circular base, controlled by a nexys3 fpga that is mounted to it. Each leg has two small servo motors attached to it, and these twelve motors are used to propel the robot in any direction within a two dimensional plane. Instructions are transmitted to the hexapod wirelessly, using the XBee 802.15.4 RF module. Proximity sensors on the robot prevent it from crashing into any objects immediately in it’s path.

TABLE OF CONTENTS

Abstract *2*
Table of Contents *3*
Section 1: Overview *4*
Section 2: Modules *5*
  Section 2.1 Motor Control *5*
  Section 2.2 Motion Algorithm *6*
  Section 2.3 Distance Sensors *6*
  Section 2.4 Xbee Communication *8*
  Section 2.5 Display and Camera Module *9*
  Section 2.6 Color Detection and Tracking *11*
  Section 2.7 Mouse and Point Detection *13*
  Section 2.8 Types of Controlling Motion *14*
    Section 2.8.1 Buttons *14*
    Section 2.8.2 Move to Point *15*
    Section 2.8.3 Tracking to Second Object *16*
Section 3: Hardware Concerns *17*
  Section 3.1: Chassis Construction *17*
  Section 3.2: Zeroing Motors *18*
  Section 3.3 Wiring *18*
  Section 3.4 Power *18*
Section 4: Challenges and Considerations *18*
  Section 4.1 Walking and Timing *18*
  Section 4.2 Weight Consideration *18*
  Section 4.3 Move to Point Inaccuracies *18*
  Section 4.4 RGB vs HSV *18*
Section 5: Conclusion *19*
Appendix A: Verilog Code *20*

1. Overview - Both

The implementation of our project consists of two main sections each using a separate FPGA. One section is the movement and control of the hexapod itself, using a Nexys 3 development board mounted on the top of the robot. The other section uses the labkit FPGA to take input from the user, from a NTSC camera, and from a PS/2 mouse, and turns that input into instructions for the hexapod.

Figure 1, below, shows the basic flow of data in the two sections (in blue) and between them. Each of the smaller parts (in green) will be explained in greater detail in **section 2**.

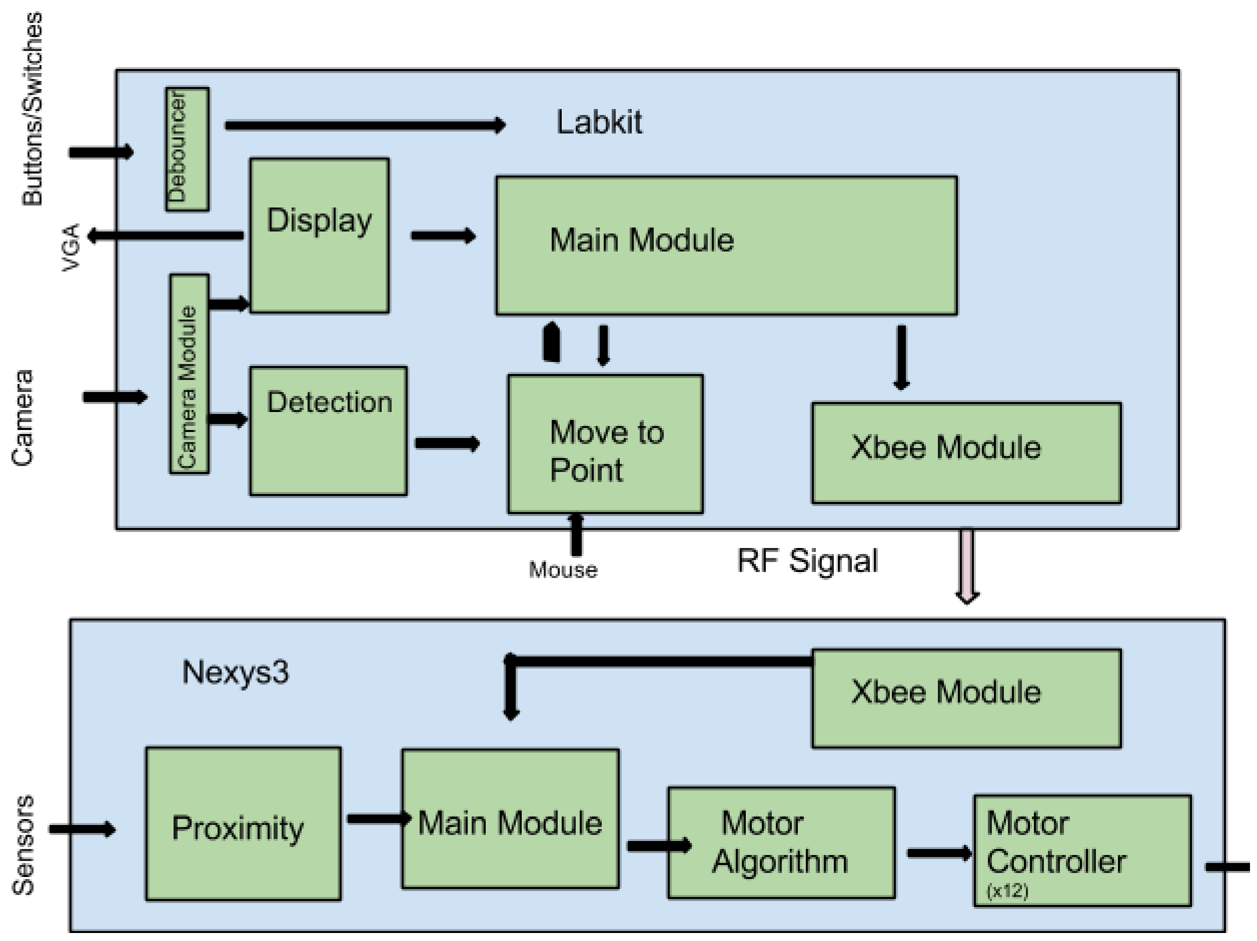

Figure 1. This diagram shows the basic high level flow of our system. It can be seen that the system is divided into two main parts, which then are divided into smaller parts.

Inputs to the system will consisted of data from the camera as well as user input on the labkit hardware. The user can control the hexapod's movement directly using buttons, or hexapod can move autonomously to a point specified by the user with a mouse or a second object.. In button mode, debounced signals from the buttons would be passed through the Main

Labkit module to the Xbee module, and transmitted to the hexapod. In the alternative case, data from the camera would feed into the Image Recognition Module and the Screen Module. The latter would display the robot's location, and interactions between the Screen module and the Labit Main Module would produce coordinates for the robot to travel towards. Using coordinates from the Main FSM and the Image Recognition Module, the Mapping Module would produce directions to pass through the Main Labkit FSM to the Xbee.

On the other side of the system, the Xbee module on the hexapod receives data from the labkit, and passes it to the hexapod's Main FSM. This will use data from the proximity sensor module to decide if it's appropriate to move, and selectively send the directions to the Motor FSM to eventually be translated into motion.

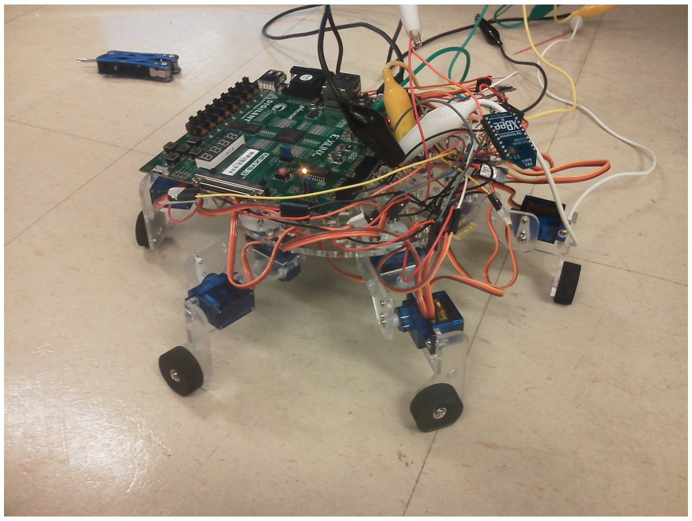

Image 1: The hexapod chassis with Xbee and Nexys 3 mounted on top

2. Modules

2.1 Motor Control - Becca

Each of the twelve servo motors on the joints of the robot is directly controlled via it's own motor control module. Servo motors are controlled using a pwm signal of 50hz, and the position is determined by varying the duty cycle of the signal. The input to the module is the number of milliseconds that the signal should be high (pulse width). The motor controller uses a 1mhz enable line to count every microsecond of the cycle, and maintains a high output while a register counts up to the desired length, and then hold the signal low until it counts up to the end of the cycle. It is important to note that the motor position can't be updated more than once per pwm cycle. Thus it is important to have very precise timing between the algorithm that generates the duty cycles and the modules that actually generate the pwm signal. This is discussed further in Section 4.2. Some experimentation was necessary to optimize this relationship. If the clock ration between the algorithm and the pwm generator was 1:1 or lower, movement was incredibly awkward and erratic. If the ration was too high, precision was lost. Although a separate control for speed existed within the algorithm generator, a combination of speed and optimal timing was necessary to create smooth motion.   It is also necessary to note that the servos can only accept signals with pulse widths ranging from 800-2200us, otherwise the motor can be damaged. . Fortunately, we used motors with 3.3v logic, so no level conversion circuitry was necessary.

2.2 Motion Algorithm - Becca

A rather complex algorithm governs the motors of the hexapod. It is based off suggestions from the company that produces the hexapod chassis kit we purchased, and has been altered to be feasible in verilog and to take advantage of the FPGA's ability to perform vector calculations . Each motor is treated as an index in a matrix, and different algorithms are used to independently calculate the motion for the knee and hip joints. These are then combined by multiplying by unit vectors. All motion is based on circles and sine waves- the motors move back and forth following sin and cos waves, and are oriented around the circle of the robot chassis. Even the algorithm itself is circular, using self-referential sine waves. Even and odd legs are 180 degrees out of phase.

There are four main parameters to the algorithm - speed, stride, step and angle. Angle determines the direction the robot will travel in, ranging from 0 to 360 degrees. Step is used to track the algorithm over time. It also ranges from 0 to 360 degrees, and is incremented every cycle by the speed.. Stride determines the range of motion of the leg, multiplying angles by a constant to create the desired range of motion of the robot- too small and it's inefficient, too large and limbs collide with both each other and the ground.

The calculation for the knees of the robot is always the same, independent of input angle. It is simply sin(Step) * Stride. Thus the knees of the robot are always moving up and down in a sine wave. Calculation for the hips is a bit more complication- in addition to the cos(Step)*Stride, it is dependent on the sine of the desired angle of travel added to the position of the leg on the body (0, 60, 120, 180, 240, and 300 degrees). The output from each cycle is added to the current motor pulse width to keep changes from being too abrupt.

This module contained a submodule sine/cos lookup table to perform trig calculations.

2.3 Distance Sensors - Becca

Our intention was to use four distance sensors, located at the cardinal directions of the robot, to prevent the robot from running into objects that the user or mapping algorithm didn't notice. Unfortunately, due to weight considerations (see Section 4.2), we were unable to actually mount the sensors on the robot, but we were able to demonstrate proof of concept by having the sensors seated on the table, and causing the robot to stop when the appropriate directional sensor was obstructed.

One of our stretch goals was to use the distance sensors to navigate around objects instead of just stopping in front of them and refusing to move. Unfortunately, this would have required gathering a lot of data about how the sensors reacted to objects approaching at different angles, and there was not enough time to attempt an undertaking of this scale.

We used Sony GP2Y0A51SK0F IR sensors to measure distance. These sensors measure distances from 2-15cm, and produce an analog signal from 0-2.5 volts, varying with according to fig. 2. Each sensor was associated with a cardinal direction (up, down, left right), and would cause the robot to stop if it was attempting to move in a direction whose sensor had been triggered to a certain threshold.

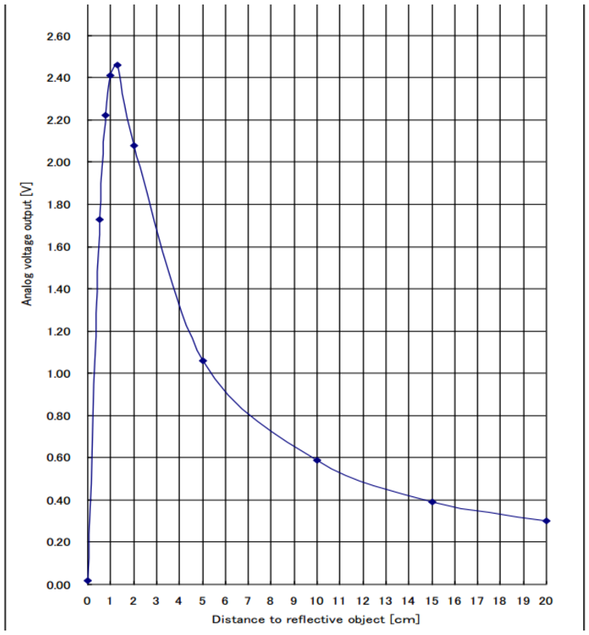

Figure 2: From the datasheet for the GP2Y0A51SK0F, a voltage-distance chart, using a piece of white paper to mark distance

The output of the sensors were hookup up to a ADC0804, an 8 bit microcontroller-compatible analog to digital converter, used in continuous conversion mode (**Figure 3**). After some experimentation, we decided to shame 6.115 teachings by only using bits [6:4] of each output, as these were the only ones that changed in a significant way. It would have been better practice to use chip select to alternate between full 8-bit data from each ADC, which would have been more than responsive enough with the quick clock cycle of the Nexys 3. However, the solution we used was much simpler, and we didn’t have time to go back and modify the design.

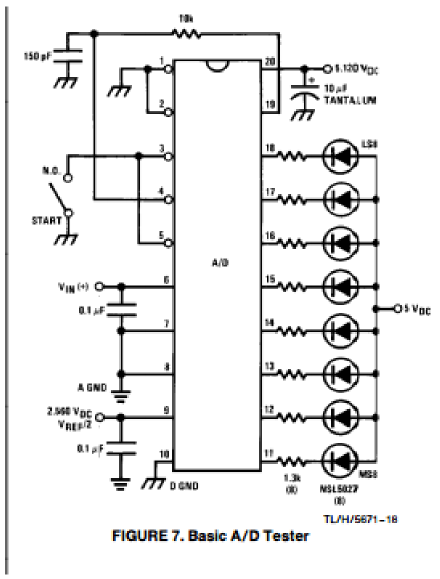

Figure 3: From the 804 datasheet, a sample application for testing with LEDs.

Figure 4 shows the wiring diagram of the chips, sensors and the Nexys 3.

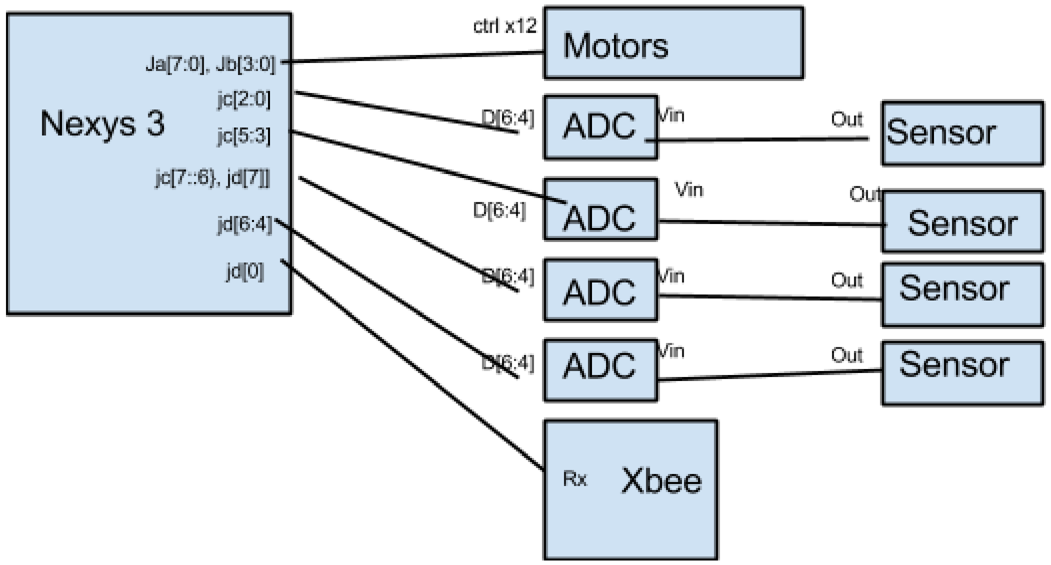

Figure 4: Block Diagram demonstration wiring between I/O pins on the Nexys 3

2.4 Xbee Communication- Becca

Wireless communication between the lab kit and the Nexys 3 was facilitated with the Xbee 802.15.4 RF transceiver, produced by Digi International. It's possible to program the Xbee for a variety of different functions, but for our purposes we programmed it to transmit UART data between the two FPGAs. I/O pins from the FPGAs were connected to the Rx and Tx lines of the Xbee; the only other necessary connections were power and ground (Figure 5). In order to use the data from the Xbee, I created two separate modules- a receiver and a transmitter.

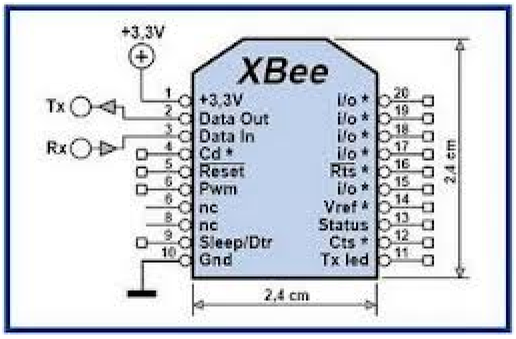

Figure 5: Pinout of the Xbee 802.15.4 RF Transceiver. The only pins used were +3.3, Tx, Rx and gnd.

The receiver module waits for a start bit and then calculates a sampling time for the 8 data bits that are being transmitted. Each bit is sampled 8 times. The sampled bits are added to a data register, and when the register is filled, a data_ready line goes high for one cycle. The transmitter takes in 8 bits of data and a enable signal. When enable goes high, the transmitter drives the output pin high or low for the appropriate amount of time, as determined by the data to be sent.

Both transmitter and receiver have an internal baud rate generator that uses the input clock frequency to create the desired baud rate. For this project, we used 9600 because that was the recommended baud for the Xbee, but other rates are possible. The generator module also outputs sample intervals for the receiver.

Packets between the two boards were 8 bits long. The high order bit determined whether or not the robot should be moving. The lower order bits were used to transmit the angle. Theoretically this could range from 0-360 degrees (incremented by steps of four, because bit shifting would be necessary to create full range). However, it was much more practical to stick to 0, 80, 180, and 270 degrees for motion, to correspond to the four pushbuttons and the locations of the four distance sensors. Thus in reality the bottom two bits were used to select angle, and the remaining five bits remained unused.

2.5 Display and Camera Module - Kelly

The purpose of this section of modules is to take video data from a 6.111 NTSC camera and display that data on a monitor. As the project became more advanced, other graphics were sometimes displayed, but these are very basic and the majority of the data displayed on the monitor still comes from the camera.

We use a modified version of the NTSC to VGA code provided by the 6.111 staff. The code takes NTSC video data from the camera and stores it in ZBT memory. That data is then displayed via VGA on a 1024x768 pixel screen. The original code stores 4 pixels worth of black and white data in each ZBT memory location, so the code has been modified to save and display RGB data instead. To do this, first the data itself is converted from YCrCb to RGB in a relatively simple linear conversion, again using code from the 6.111 staff. This produces 24 bits of RGB data, one byte for each color. In order to take full advantage of the ZBT memory, each slot of which is 36 bits wide, this data is truncated to 18 bits, and so we can store 2 pixels worth of RGB data in the memory. This data is then displayed to the monitor, along with any other graphics added. An overview of the data flow from the camera to the monitor is shown in Figure 6.

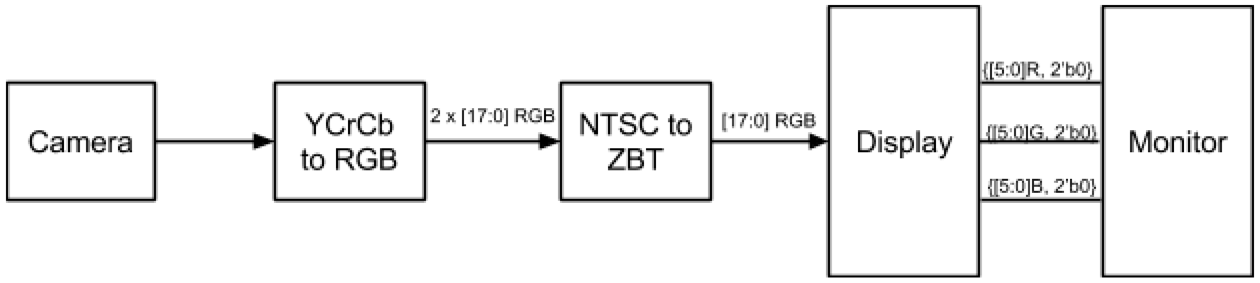

Figure 6. This figure shows the flow of data from the camera to the monitor.

2.6 Color Detection and Tracking - Kelly

This module detects a given color being displayed on the monitor and tracks it by determining the average pixel position of all colored pixels in the screen area. It displays a simple white square graphic to show the user where on the screen that average is. It is used primarily to track the location of the robot.

This detection module can be used to detect and track objects of any solid color. It is used mostly in this project to track the hexapod itself, which has a large red square on top of it, in order to make it easily identifiable. The detection and tracking process goes through several stages, and the full process of displaying the center coordinate is completed once per frame.

First, a color must be detected. When an instance of this module is instantiated, a range for each of the three colors, red, green, and blue is determined. So to detect the color red (0xFF0000 in RGB), we look for pixels with a high value for red and low values for green and blue. To detect the color yellow (0xFFFF00), we look for pixels with high values for red and green and low values for blue. Each pixel on the screen is then determined to have an RGB value inside or outside of this range. If the pixel is inside of this range, it is given a value of 18'h3FFFF (all ones), and if it is outside of the range it is given a value of 0. This creates a binary image, in which all the pixels of a certain color are white, while all other pixels are black. This makes it significantly easier to isolate and track an object of a specific color. This binary image is illustrated below in **Image 2**.

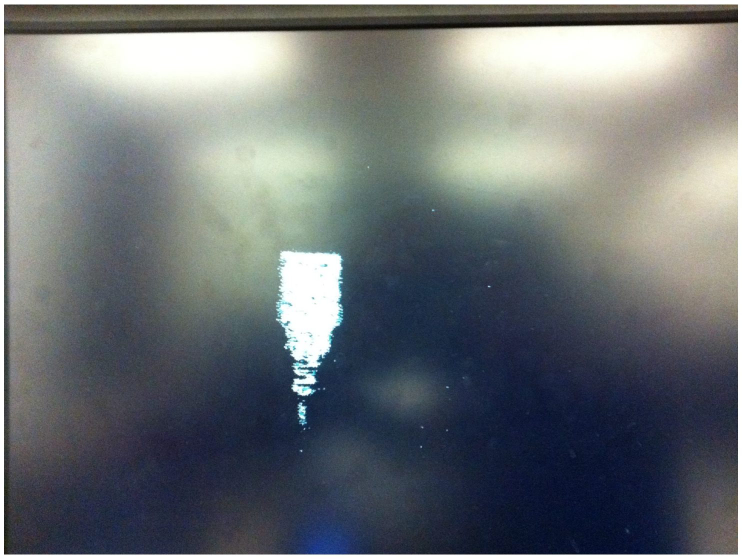

Image 2. The above image is a photograph of the code for the project running and displaying a binary image. The color being detected is red and a red object can be easily tracked (as shown by the red USB cable above).

This binary image can then be used to determine the average horizontal and vertical coordinates of all the white pixels. In theory, this image could also be used to eliminate some amount of noise, so that only the points contained within the largest cluster of white pixels were included in the average. However, I determined that, so long as the background did not contain other significant objects of the color being detected, this noise elimination was not particularly necessary.

The average or center coordinates are stored in a small buffer that also contains the three previous center coordinates. These are also averaged together to produce the final center coordinates. This is an effort to “smooth out” the tracked location.

A small (10 pixel by 10 pixel) white square is drawn on the screen, centered around the averaged coordinates. This is not actually necessary for any functionality, but simply allows the user to see that objects are being correctly tracked. It is also very simple to track two different objects of two different colors - it simply requires two instances of the module. **Image 3**, below, of the code running and tracking a red object (the hexapod), as well as a yellow object.

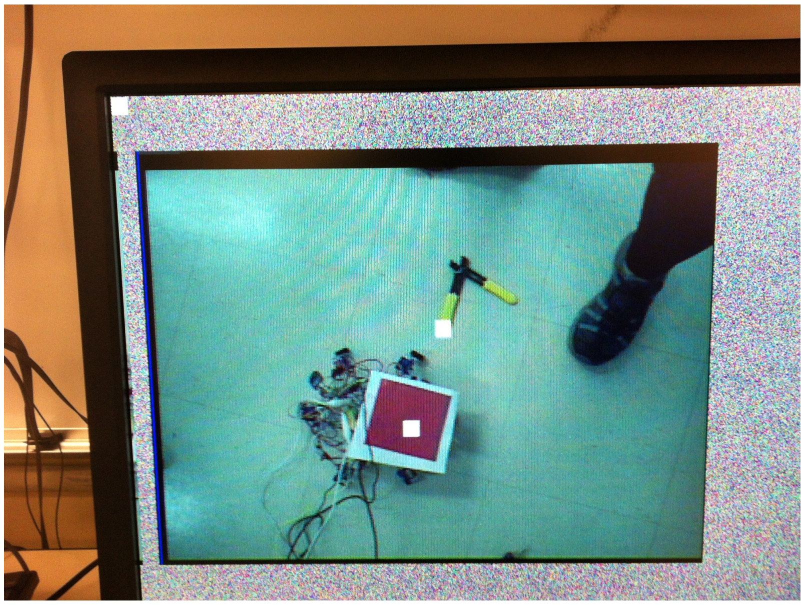

Image 3. The above images shows the code running and displaying the tracked location of both a red object and a yellow object.

2.7 Mouse and Point Detection - Kelly

This module communicates with a PS/2 mouse that is connected to the lab kit. It outputs x & y pixel coordinates of the locations of the mouse on the screen, and also detects if one of the three mouse buttons is being pressed.

The basic code for this module was obtained from the 6.111 website. It was modified slightly to fit into this project. Code was added to display another small (10 x 10 pixel) white square centered around the coordinates of the mouse, so that the user can see where the mouse is "located." Additionally, every time the left mouse button is clicked, the current coordinates are stored and saved until the button is clicked again, at which point they are replaced. We use only the left button simply because our project only requires the use of one button, as I will explain in the next section. **Image 3** shows the code running - it is tracking a red object, a yellow object, and displaying the location of the mouse.

2.8 Types of Controlling Motion - Kelly

Much of the functionality I have described so far has been independent of the hexapod itself. This is because the above sections are simply individual pieces of information collected for the greater purpose of moving the hexapod. The main purpose of the labkit fpga is to communicate wirelessly with the hexapod itself (explained in **Section 2.4**) and send it commands, directing its movement.

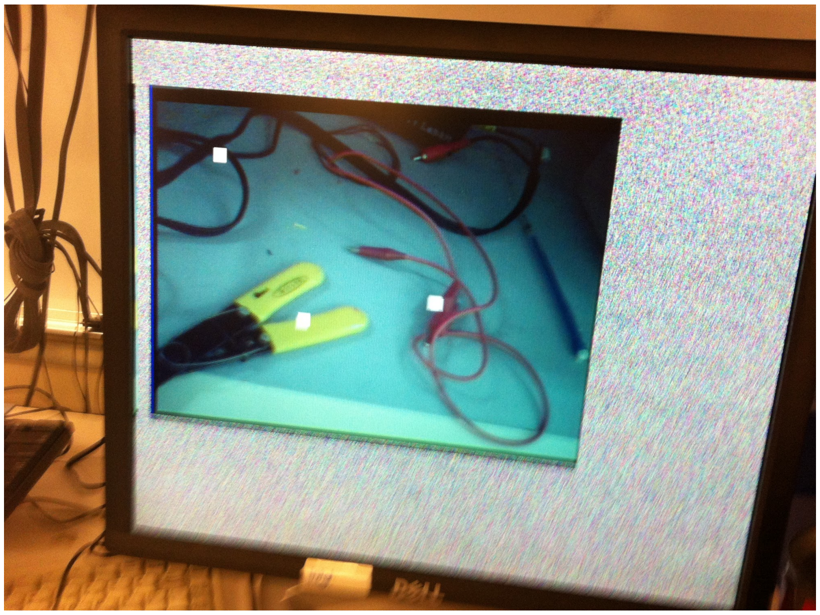

Image 3. This image shows the code running and displaying the tracked location of a red object and a yellow object, and the location of the mouse (upper left corner).

This is mostly brought together in the main labkit module, which is the top level that brings together the various modules on the labkit. With all of the information assembled from the previously described modules and from various inputs, the main module sends a packet to the hexapod every one second, containing information about movement and direction of movement.

There are three different ways in which the user can control the hexapod's movement. They will be described in greater detail in the following sections, but for a brief overview - the user can press buttons which tell the hexapod to move in a certain direction, the user can click a point on the screen with the mouse and the hexapod will move autonomously to that point, and finally, the user can present an object of a second color and the hexapod will move autonomously to that object.

2.8.1 Buttons

This is the simplest way of moving the robot. With this scheme the user can press one of four buttons (up, down, right, left) and the robot will move appropriately. Rather than strict directions, the four buttons actually represent four angles. As was discussed in **section n**, the command to control the hexapod consists of a speed and an angle. For this implementation, there is just one speed - moving or not moving - and four different angles represented by the four buttons - 0, 90, 180 and 270 degrees.

When any of the four buttons is pressed, this is detected by the main labkit module, and the press is stored until the next packet is sent out on the second. The packet consists of a command to move and the appropriate angle. If no button is being pressed, a command of all zeros - no movement and zero angle - is sent.

2.8.2 Move to Mouse Point

This module implements the second type of movement - autonomous movement by the hexapod to a point determined by the user. This type of movement is much more complex than the previous, and actually makes use of the information collected by the other modules - the location of the robot and the position of the mouse.

With this implementation, the robot is at rest until the user clicks somewhere on the screen with the mouse. Once the mouse has been clicked, the point that was clicked becomes the destination for the robot. This point remains the destination unless the mouse is clicked again at a different point. In that case, the new point becomes the destination, even if the robot had not reached the previous destination.

The command packets take the same form they did with the buttons. The most significant bit is a 1 or a 0 to determine if the hexapod is moving or not moving. The two least significant bits determine angle. Again, we only use the four cardinal direction angles, as this was the most practical and least complicated form to implement. The difference between the two implementations is that commands to the robot are automatically determined and adjusted based on the location of the robot in relation to its destination.

Until the mouse is clicked, the robot simply does not move. When the mouse is clicked, the first command simply tells the robot to move in the 0 degree direction, whatever it thinks is "straight ahead." The tracking algorithm simply determines where the robot is, it does not determine orientation, so this first step is really a calibration step. At the start, the algorithm has no idea where the robot will actually go when told to move in a certain direction, so it simply tells it to go in a random direction, which might be completely away from the destination. This is illustrated in **Figure 7.** After that, it can begin adjustment.

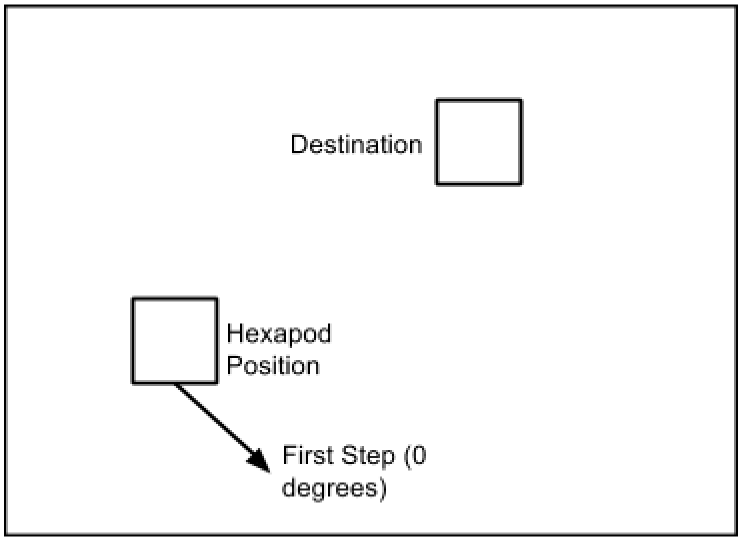

Figure 7. This figure shows how the hexapod might move at initialization (when a point is first clicked).

A key part of the algorithm is the fact that it saves and uses the last position of the hexapod in addition to using the current position. So, once a second, the module calculates the distance of the hexapod to the destination in both the x and y directions, as well the distance of the last position of the hexapod to the destination in the x and y directions. This allows for effective adjustment, by analyzing the progress of the hexapod.

Some adjustments are very simple - if the hexapod has gotten closer to the destination in both the x and y direction, it should continues moving at the same angle it currently is. If it has gotten farther away in both directions, it should adjust by 180 degrees and go back the way it came. If it has only gotten closer in one direction however, it is a bit more complicated, and using distance information is not enough. Additionally, the algorithm uses the position of the hexapod and the point destination. Whether the hexapod should adjust by 90 or -90 degrees is determined by where the hexapod is in relation to the destination, rather than how far away it is.

Figure 8 shows a possible trajectory of the hexapod. After some testing, we determined that adjusting by only multiples of 90 degrees was simple and sufficient. When the hexapod moves within a certain distance of the destination, it is determined that it has reached the destination and stops moving until given a new destination.

2.8.3 Tracking to 2nd Object

This turned out to be much simpler to implement than expected, with the ease of tracking a second object with the color detection module. As a result, the exact same algorithm can be applied to this type of movement. Instead of mouse coordinates, the move to point module is simply instantiated with the tracked location of a second object as the destination.

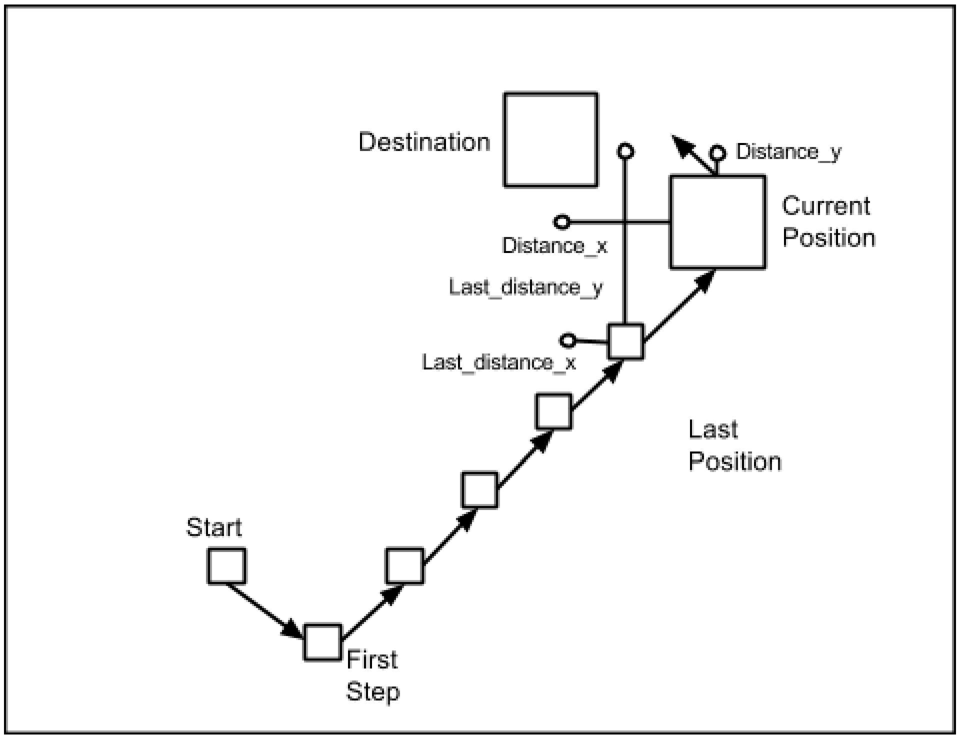

Figure 8. The above diagram shows a possible trajectory for the hexapod. Each movement is based on the distance of the hexapod from the destination (in both directions) the distance of the last location of the hexapod from the destination, and the position of the hexapod in relation to the destination.

3. Hardware Concerns

Beyond the programming of the fgpa and the construction of related circuits, the project contained several hardware concerns that were necessary for completion. Many of these were affected by load weight concerns, which is discussed in more detail in **Section 4.2**.

3.1 Chassis construction

Fortunately, the challenge of constructing the chassis was mostly avoided. After talking to Professor Steve Leeb for advice, he volunteered to buy us a hexapod chassis kit from Sparkfun. Thus we were saved many hours of mechanical design and machining, and the construction of the chassis was reduced to a ~1 hour problem of plastic parts and tiny screws.

3.2 Zeroing Motors

It was necessary to zero all the motors with a pulse width of 1500us before connecting any of the joints of the hexapod to the chassis. Because the robot was carrying such a heavy load (see **Section 4.2**), motors would frequently slip and need to be re-zeroed after several test runs.

3.3 Wiring

There were a bunch of wires connecting the various motors and sensors to the Nexys 3 devboard. Several different wiring schemes were attempted before we found one that was practical, and even then we sometimes had issues with servo motors pulling out their own wires, or the legs becoming entangled in power cords.

3.4 Power

Both the Nexys 3 and the ADC chips required 5v power supply, while the motors and the distance sensors needed 3.3V supply. All of the motors and the sensors drew a lot of current- it could push to over 1.5A, at certain stages of motion.   Our original plan was either to use both a 5v and a 3.3v battery, or use a 5v and a step down. However, it quickly became apparent that the robot could never support the weight of a battery, so instead we trailed wires from the labkit power supply.

4. Challenges - Both

4.1 Walking and Timing

We didn't initially realize how difficult it would be to optimize timing between the various module controlling motion. It is incredibly important to have very precise timing between the algorithm that generates the duty cycles and the modules that actually generate the pwm signal. Delay within the motors themselves was also a factor. A surprising amount effort was required to find timing that would actually create a walking motion, instead of random twitches or flails. .If the clock ratio between the algorithm and the pwm generator was 1:1 or lower, movement was incredibly awkward and erratic. If the ration was too high, precision was lost. Although a separate control for speed existed within the algorithm generator, a combination of speed and optimal timing was necessary to create smooth motion. The optimal ratio was only determined after many hours of experimentation.

4.2 Weight Considerations

Although it was obviously a better choice to use the hexapod chassis kit, the kit did come with certain limitation. The 9G servos it was designed with are not very powerful, meant to support the weight of the chassis, and perhaps a very small driving board. The Nexys 3, while not a terribly large board, is clearly much bigger and heavier than the chassis was designed to accommodate. As we performed more test runs, the motors became weaker and weaker, until finally they would slip and collapse underneath the weight of the devboard. In order for the robot to perform, it must be partially supported. Any weight that could be moved off the hexapod found a new home on the lab bench.

4.3 Move to Point Inaccuracies

There were multiple challenges with the move to point algorithm, both with the algorithm itself and with the real world non-idealities once it was implemented.

The first challenge was the lack of information about the orientation of the robot. We came up with a variety of possible solutions, including having two colors on the robot to determine a front and a back. Ultimately, however we decided the simplest strategy would just be to have the robot move "straight" and adjust based on visual feedback of where it went.

The second challenge was with the algorithm itself. Initially, we had hoped to implement a more sophisticated strategy where we actually calculated a more precise angle for the robot to move in, based on its distance from the destination in the x and y directions. This would have involved a relatively simple trigonometric calculation ($\theta = arctan(distx/disty)$). However, this trig is not particularly simple to implement in verilog. I began looking at ways to use the CORDIC math core to implement this function, but given time constraints, determined it would be simpler to implement a less complicated basic algorithm using easy angles and visual feedback.

Finally, once that algorithm was actually implemented, there were several non idealities in the execution. While the algorithm worked well on paper, and reasonably well in reality, there were many things that made it less than perfect. First was the motion of the robot itself, which did not always move in particularly straight lines and was often quite erratic. Second was the tracking, which did not actually maintain a fixed center for the object it was tracking, but rather bounced around a good amount. This was made a bit better by averaging the centered coordinates over time, but was still sometimes erratic enough to prevent the algorithm from really fully working.

4.4 RGB vs. HSV

Most of the reading that I did on color detection and object tracking recommended using the HSV (Hue Saturation Value) color space rather than RGB. However, I struggled to actually implement the RGB to HSV conversion. Though I used relatively straightforward code provided by 6.111, when I tried to set ranges for HSV values to detect a particular color, the code barely worked and picked up mostly noise. I'm sure this was an error in my implementation (though I compared my implementation to others, and couldn't find a difference), but after a while, I determined that using the RGB color space was perfectly sufficient for our purposes, and converting to HSV was not really worth the effort.

Other than that, there were still minor problems with the color detection. This code cannot track two items of the same color, for instance, only two items of different colors. In fact, if there are two distinct items of the same color on the screen, this code fails to track even one of them, and instead settles on a point somewhere in the middle. This is a disadvantage and often caused some problems - if one of us was wearing a red shirt for instance. This was often a common problem with tracking yellow - given that the hexapod was moving on the floor, the camera often picked up the yellow in the floor and became confused, making it difficult to track a single yellow object. However, this implementation works out fairly well for our purposes, and the drawbacks and complications are easy enough to avoid.

5. Conclusion

Over the course of five weeks, we were able to implement all of our goals as described in our project proposal, with the exception of object avoidance with the proximity sensors. Although we encountered a number of difficulties with both hardware and software, we were able to find solutions to these problems that were functional, if not always sophisticated. We learned that it is important not to abstract away hardware problems such as motor strength or power supplies, and these considerations are just as critical to a project's success as the code that controls it.

Without a doubt, use of ModelSim as a debugging tool was essential to the operations of the motion algorithm. However, a large amount of user testing was also necessary to see how the systems interacted in real life, which was not always exactly as expected. Small issues in one system would cause another to function poorly- if the robot wasn't walking quite straight,our original mapping algorithm was rendered practically useless. Finding appropriate timing for clocks between modules and signals between labkits was also an important aspect to the project that we hadn't originally considered. Overall, we think the project was a lot of fun and a great learning experience.

Appendix A: Verilog Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:  Becca Greene
//
// Create Date:    22:51:37 04/11/2013 C
// Design Name:
// Module Name:    labkit
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module labkit(
   input clk_100mhz,
   input [7:0] switch,

   input btn_up,       // buttons, depress = high
   input btn_enter,
   input btn_left,
   input btn_down,
   input btn_right,

   output [7:0] seg,   //output 0->6 = seg A->G ACTIVE LOW,
                  //output 7 = decimal point, all active low

   output [3:0] dig,   //selects digits 0-3, ACTIVE LOW
   output [7:0] led,   // 1 turns on leds

   output [2:0] vgared,
   output [2:0] vgagreen,
   output [2:1] vgablue,
   output hsync,
   output vsync,
         output[7:0] ja,
   output [7:0] jb,
   input [7:0] jc,
   input [7:0] jd,
   inout [19:0] exp_io_n,
   inout [19:0] exp_io_p
   );

   // all unused outputs must be assigned
   assign vgared = 3'b111;
   assign vgagreen = 3'b111;
   assign vgablue = 2'b11;
   assign hsync = 1'b1;
   assign vsync = 1'b1;

// next three lines turns the 7 seg display completely off
   assign seg = 7'b111_1111;           //output 0->6 = seg A->G ACTIVE LOW
   assign dp = 1'b1;                           //decimal point ACTIVE LOW
   assign dig = 4'hF;                  //selectives digits 0-3, ACTIVE LOW



////////////////////////////////////////////////////////////////////
// dcm_all is a general purpose digital clock manager. It is used
// to create clocks at desired frequncies and phases.
//
   wire clk_25mhz;
         wire clk_100mhz_buf;  // 100mhz buffered clock, not used

   dcm_all_v2 #(.DCM_DIVIDE(8), .DCM_MULTIPLY(2))
         my_clocks(
                                   .CLK(clk_100mhz),
                                   .CLKSYS(clk_100mhz_buf),
//                                 .CLK25(CLK25),
                                   .CLK_out(clk_25mhz)
         );
//
////////////////////////////////////////////////////////////////////
// clocks and button debouncers
////////////////////////////////////////////////////////////////////
   debounce  db_up(.reset(1'b0), .clock(clk_25mhz), .noisy(btn_up), .clean(up));
```

```
                  debounce  db_down(.reset(1'b0), .clock(clk_25mhz), .noisy(btn_down),
.clean(down));
                  debounce  db_left(.reset(1'b0), .clock(clk_25mhz), .noisy(btn_left), .clean(left));
                  debounce  db_right(.reset(1'b0), .clock(clk_25mhz), .noisy(btn_right),
.clean(right));
                  debounce  db_enter(.reset(1'b0), .clock(clk_25mhz), .noisy(btn_enter),
.clean(enter));

                  assign reset = enter;
                  wire en_1mhz;
                  reg [14:0] pulse_width = 12'h5DC;
                  wire pwm;
                  wire en_50hz;

                  divider #(26'h19) oneMhz (.clock_in(clk_25mhz), .enable(en_1mhz));
                  divider #(26'h3D090) fiftyhz (.clock_in(clk_25mhz), .enable(en_50hz));
                  genPWM pwmgen (.enable(en_1mhz), .pulse_width(pulse_width), .pwm(pwm));
                  //entoclk twomhz (.enable(en_1mhz), .clk(ja[0]));

         ///////////////////////////////////////////////////////
         //          Serial Handling
         ///////////////////////////////////////////////////////
                  reg TxD_start;
                  wire [7:0] TxD_data;
                  wire TxD;
                  wire TxD_busy;
                  wire RxD;
                  wire RxD_data_ready;
                  wire [7:0] RxD_data;
                  wire RxD_idle;
                  wire RxD_endofpacket;
                  reg [7:0] data_buffer;


                  transmitter xbeeout (.clk(clk_25mhz), .TxD_start(TxD_start),
.TxD_data(TxD_data), .TxD(TxD),
                                                                                    .TxD_busy(TxD_busy));

                  receiver xbeein (.clk(clk_25mhz), .RxD(RxD),
.RxD_data_ready(RxD_data_ready), .RxD_data(RxD_data),
                                                                     .RxD_idle(RxD_idle),
.RxD_endofpacket(RxD_enofpacket));
```

```
		always @(posedge clk_25mhz) begin
			if (RxD_data_ready) begin
				data_buffer <= RxD_data;
				TxD_start <= 1;
			end else begin
				TxD_start <= 0;
			end

	////////////////////////////////////////////////////////////////
	// ADC and Xbee data Handling

			// speed
			if (data_buffer[7]) begin
				if ((jc[0]||jc[1]||jc[2]) && angle <= 9'd0) speed <=0; // check forward
ADC
				else if ((jc[1]||jc[0]||jd[7]) && angle <= 9'd90) speed <= 0;  // check
right ADC
				else if ((jc[4]||jc[3]||jc[2]) && angle <= 9'd180) speed <= 0; // check
backward ADC
				else if ((jc[7]||jc[6]||jc[5]) && angle <= 9'd270) speed <= 0; // check
left ADC
				else speed <=4;
			end
			else speed <= 0;
				//angle
				case (data_buffer[6:0])
				7'b00: angle<= 9'd0;
				7'b01: angle <= 9'd90;
				7'b10: angle <= 9'd180;
				7'b11: angle <= 9'd270;
				default: angle <= 9'd0;


			endcase
		end

		assign RxD = jd[0];
		assign jd[1] = TxD;
		assign TxD_data = data_buffer;


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

```
        // Motor Management
        //////////////////////////////////////////////////////////


                reg [8:0] angle = 0;
                reg [4:0] speed= 0;
          wire [15:0] knee0us;
                wire [15:0] hip0us;
                wire [15:0] knee1us;
                wire [15:0] hip1us;
                wire [15:0] knee2us;
                wire [15:0] hip2us;
                wire [15:0] knee3us;
                wire [15:0] hip3us;
                wire [15:0] knee4us;
                wire [15:0] hip4us;
                wire [15:0] knee5us;
                wire [15:0] hip5us;

        // If not moving, return to a neutral position- better for balance
                wire [15:0] knee0  = (speed == 0)? 1500: knee0us;
                wire [15:0] hip0  = (speed == 0)? 1500: hip0us;
                wire [15:0] knee1  = (speed == 0)? 1500: knee1us;
                wire [15:0] hip1  = (speed == 0)? 1500: hip1us;
                wire [15:0] knee2  = (speed == 0)? 1500: knee2us;
                wire [15:0] hip2 = (speed == 0)? 1500: hip2us;
                wire [15:0] knee3  = (speed == 0)? 1500: knee3us;
                wire [15:0] hip3  = (speed == 0)? 1500: hip3us;
                wire [15:0] knee4  = (speed == 0)? 1500: knee4us;
                wire [15:0] hip4  = (speed == 0)? 1500: hip4us;
                wire [15:0] knee5  = (speed == 0)? 1500: knee5us;
                wire [15:0] hip5  = (speed == 0)? 1500: hip5us;

                // Motor Algorithm
                alg motionAlgorithm (.clk(en_50hz), .angle(angle), .speed(speed),
.knee0(knee0us), .hip0(hip0us),
                                        .knee1(knee1us), .hip1(hip1us), .knee2(knee2us),
.hip2(hip2us), .knee3(knee3us),
                                        .hip3(hip3us), .knee4(knee4us), .hip4(hip4us),
.knee5(knee5us), .hip5(hip5us));


                reg [15:0] rset = 1500;
```

```
            reg [15:0] lowend = 800;
            reg [15:0] highend = 2200;



// pwm signal generation -> output
            genPWM knee0pwm (.enable(en_1mhz), .pulse_width(knee0), .pwm(ja[0]));
    genPWM hip0pwm (.enable(en_1mhz), .pulse_width(hip0), .pwm(ja[1]));
            genPWM knee1pwm (.enable(en_1mhz), .pulse_width(knee1), .pwm(ja[2]));
            genPWM hip1pwm (.enable(en_1mhz), .pulse_width(hip1), .pwm(ja[3]));
            genPWM knee2pwm (.enable(en_1mhz), .pulse_width(knee2), .pwm(ja[4]));
            genPWM hip2pwm (.enable(en_1mhz), .pulse_width(hip2), .pwm(ja[5]));
            genPWM knee3pwm (.enable(en_1mhz), .pulse_width(knee3), .pwm(ja[6]));
            genPWM hip3pwm (.enable(en_1mhz), .pulse_width(hip3), .pwm(ja[7]));
            genPWM knee4pwm (.enable(en_1mhz), .pulse_width(knee4), .pwm(jb[0]));
            genPWM hip4pwm (.enable(en_1mhz), .pulse_width(hip4), .pwm(jb[1]));
            genPWM knee5pwm (.enable(en_1mhz), .pulse_width(knee5), .pwm(jb[2]));
            genPWM hip5pwm (.enable(en_1mhz), .pulse_width(hip5), .pwm(jb[3]));

            // assign unused ports
            assign jb[7:4] = 4'h0;

endmodule
///////////////////////////////////////////////////////////////////////////////
// alg module
// Determines pulse width of servos for walking motion using matrix multiplication
// See report for full explanation
///////////////////////////////////////////////////////////////////////////////
module alg (input clk,
                        input [8:0] angle,
                        input [4:0] speed,
                        output wire [15:0] knee0,
                        output wire [15:0] hip0,
                        output wire [15:0] knee1,
                        output wire [15:0] hip1,
                        output wire [15:0] knee2,
                        output wire [15:0] hip2,
                        output wire [15:0] knee3,
                        output wire [15:0] hip3,
                        output wire [15:0] knee4,
                        output wire [15:0] hip4,
                        output wire [15:0] knee5,
                        output wire [15:0] hip5
```

```
			);


		reg [8:0] stride;					// range of motion
		reg delay = 0;						// used to sync timing
		wire [8:0] stepmatrix[11:0];	// used to include step in genvar statments
		reg [8:0] step;
		wire [8:0] start[11:0];			// starting positions for hips
		wire [9:0] xang[11:0];			// final hip angle
		wire signed [9:0] xcos[11:0];	// cos of xang
		wire signed[9:0]  xsin[11:0];	// sing of xang
		wire signed[15:0]  odds[11:0];		// knees
		wire signed[15:0]  eves[11:0]; // hips
		wire signed [18:0] x[11:0];	// value produced to account for hip angle changes,
unshifted. Used mostly for debugging
		wire signed [15:0] xf[11:0];	// x shifted to actual output values.
		reg signed [15:0] out[11:0];	// output as registers
		wire signed [15:0] outwire[11:0]; // output as wires. necessary for combining
blocking and unblocking statements
		wire [7:0] shift[11:0];			// used to put odd and even legs out of
phase
		wire [9:0] shifted[11:0];				// stepmatrix with odd and even legs
out of phase
		wire signed [9:0] outsin[11:0];		// sign of step angle
		wire signed [9:0] outcos[11:0];			// cos of step angle
		wire sel1[11:0];						// used to select for hips
		wire sel2[11:0];						// used to select for knees
		wire xsinneg[11:0];						// used to sign lookup tables
		wire outcosneg[11:0];				//
		wire outsinneg[11:0];				//

		// starting values
		assign start[0] = 0;
		assign start[1] = 0;
		assign start[2] = 60;
		assign start[3] = 60;
		assign start[4] = 120;
		assign start[5] = 120;
		assign start[6] = 180;
		assign start[7] = 180;
		assign start[8] = 240;
		assign start[9] = 240;
		assign start[10] = 300;
		assign start[11] =  300;

		assign shift[0] = 0;
		assign shift[1] = 0;
		assign shift[2] = 180;
		assign shift[3] = 180;
		assign shift[4] = 0;
		assign shift[5] = 0;
		assign shift[6] = 180;
		assign shift[7] = 180;
		assign shift[8] = 0;
		assign shift[9] = 0;
		assign shift[10] = 180;
		assign shift[11] = 180;

		genvar c;
		generate
			for (c = 0; c < 12; c = c+1) begin
				assign xang[c] = (start[c] + angle) < 360? start[c] + angle: start[c]
+ angle - 360; // calculate hip angle
				assign stepmatrix[c] = step;
				assign shifted[c] = (stepmatrix[c] + shift[c]) < 360? stepmatrix[c] +
shift[c]: stepmatrix[c] + shift[c] - 360; //puts legs out of phase
				assign x[c] = (xsin[c]*stride);  // hip angle * stride
				assign xf[c] = {x[c] >> 10};    // shifted
				assign sel1[c] = (c%2 == 0)? 0:1; // create sel1 and sel 2
				assign sel2[c] = (c%2 == 0)? 1:0;
				assign eves[c] = (outsinneg[c])? -1*((sel2[c]*stride*outsin[c]) >>
13): (sel2[c]*stride*outsin[c]) >> 13; //hip final output
				assign odds[c] = (outcosneg[c]^xsinneg[c])?
-1*((sel1[c]*xf[c]*outcos[c]) >> 13): (sel1[c]*xf[c]*outcos[c]) >> 13; // knee final output
				assign outwire[c] = out[c] +odds[c] + eves[c]; //overall output
				trig c_xang (.clk(clk), .angle(xang[c]), .cos(xcos[c]), .sin(xsin[c]),
.sinneg(xsinneg[c]));
				trig c_outang (.clk(clk), .angle(shifted[c][9:0]), .cos(outcos[c]),
.sin(outsin[c]), .cosneg(outcosneg[c]), .sinneg(outsinneg[c]));
			end
		 endgenerate


		assign knee0 = out[0] ;
		assign hip0 = out[1];
		assign knee1 = out[2];
assign hip1 = out[3];
assign knee2 = out[4];
assign hip2 = out[5];
assign knee3 = out[6];
assign hip3 = out[7];
assign knee4 = out[8];
assign hip4 = out[9];
assign knee5 = out[10];
assign hip5 = out[11];

initial begin
        step = 8'b0;
        stride = 0;
        out[0] = 12'd1500;
        out[1] = 12'd1550;
        out[2] = 12'd1550;
        out[3] = 12'd1450;
        out[4] = 12'd1500;
        out[5] = 12'd1400;
        out[6] = 12'd1500;
        out[7] = 12'd1550;
        out[8] = 12'd1500;
        out[9] = 12'd1500;
        out[10] = 12'd1500;
        out[11] = 12'd1400;
end

always @(posedge clk) begin
        if(speed==0) begin
                stride <= stride - 25;
                if(stride<0) stride=0;
        end else begin
                if(stride>400) stride <= 400;
                else stride<=stride + 25;
        end

        if (step + speed < 361) step <= step + speed;
        else step <= step + speed - 360;
        if (~delay) delay = 1;
        else begin
                // an additional safety margin in case changing stride is insufficient.
                if (outwire[0] < 1000) out[0] <= 1000;
                else if (outwire[0] > 2200) out[0] <= 2200;
                else out[0] <= outwire[0];

                if (outwire[1] < 1000) out[1] <= 1000;
                else if (outwire[1] > 2200) out[1] <= 2200;
                else out[1] <= outwire[1];

                if (outwire[2] < 1000) out[2] <= 1000;
                else if (outwire[2] > 2200) out[2] <= 2200;
                else out[2] <= outwire[2];

                if (outwire[3] < 1000) out[3] <= 1000;
                else if (outwire[3] > 2200) out[3] <= 2200;
                else out[3] <= outwire[3];

                if (outwire[4] < 1000) out[4] <= 1000;
                else if (outwire[4] > 2200) out[4] <= 2200;
                else out[4] <= outwire[4];

                if (outwire[5] < 1000) out[5] <= 1000;
                else if (outwire[5] > 2200) out[5] <= 2200;
                else out[5] <= outwire[5];

                if (outwire[6] < 1000) out[6] <= 1000;
                else if (outwire[6] > 2200) out[6] <= 2200;
                else out[6] <= outwire[6];

                if (outwire[7] < 1000) out[7] <= 1000;
                else if (outwire[7] > 2200) out[7] <= 2200;
                else out[7] <= outwire[7];

                if (outwire[8] < 1000) out[8] <= 1000;
                else if (outwire[8] > 2200) out[8] <= 2200;
                else out[8] <= outwire[8];

                if (outwire[9] < 1000) out[9] <= 1000;
                else if (outwire[9] > 2200) out[9] <= 2200;
                else out[9] <= outwire[9];

                if (outwire[10] < 1000) out[10] <= 1000;
                else if (outwire[10] > 2200) out[10] <= 2200;
                else out[10] <= outwire[10];

                if (outwire[11] < 1000) out[11] <= 1000;
```

```
                    else if (outwire[11] > 2200) out[11] <= 2200;
                    else out[11] <= outwire[11];
                end

        end

endmodule
////////////////////////////////////////////////////////////////////////////////
// trig function
// lookup table given angle in degrees, outputs sin and cos
////////////////////////////////////////////////////////////////////////////////

/* sine and cos values are multiplied by 1024 and rounded, to be bitshifted later */
module trig ( input clk,
                    input [9:0] angle,
                    output reg [9:0] sin,
                    output reg [9:0] cos,
                    output wire cosneg,
                    output wire sinneg );


        //reg signed[14:0] sin;
        wire [6:0] angle2;


        //reg signed [14:0] cos;

        assign sinneg = (angle > 180)? 1 : 0;
        assign angle2 = (angle < 91)? angle : (angle < 181) ? (180 - angle) : (angle <
271)? (angle - 180): (360- angle);
        assign cosneg = (90 < angle & angle < 270)? 1: 0;

        always @(posedge clk) begin

        case (angle2)
                0: sin <= 0000;
                1: sin <= 0017;
                2: sin <= 0036;
                3: sin <= 0052;
                4: sin <= 0072;
                5: sin <= 0088;
                6: sin <= 0108;
                7: sin <= 0126;
```

```
8: sin <= 0142;
9: sin <= 0160;
10: sin <= 0178;
11: sin <= 0196;
12: sin <= 0213;
13: sin <= 0234;
14: sin <= 0248;
15: sin <= 0265;
16: sin <= 0283;
17: sin <= 0299;
18: sin <= 0316;
19: sin <= 0333;
20: sin <= 0350;
21: sin <= 0367;
22: sin <= 0384;
23: sin <= 0400;
24: sin <= 0407;
25: sin <= 0416;
26: sin <= 0448;
27: sin <= 0464;
28: sin <= 0481;
29: sin <= 0497;
30: sin <= 0512;
31: sin <= 0527;
32: sin <= 0543;
33: sin <= 0558;
34: sin <= 0573;
35: sin <= 0588;
36: sin <= 0602;
37: sin <= 0616;
38: sin <= 0631;
39: sin <= 0644;
40: sin <= 0658;
41: sin <= 0672;
42: sin <= 0685;
43: sin <= 0698;
44: sin <= 0712;
45: sin <= 0724;
46: sin <= 0736;
47: sin <= 0749;
48: sin <= 0761;
49: sin <= 0773;
50: sin <= 0784;
```

```
51: sin <= 0795;
52: sin <= 0806;
53: sin <= 0818;
54: sin <= 0828;
55: sin <= 0839;
56: sin <= 0849;
57: sin <= 0859;
58: sin <= 0868;
59: sin <= 0878;
60: sin <= 0886;
61: sin <= 0896;
62: sin <= 0904;
63: sin <= 0912;
64: sin <= 0921;
65: sin <= 0928;
66: sin <= 0936;
67: sin <= 0943;
68: sin <= 0949;
69: sin <= 0956;
70: sin <= 0963;
71: sin <= 0969;
72: sin <= 0974;
73: sin <= 0979;
74: sin <= 0984;
75: sin <= 0989;
76: sin <= 0993;
77: sin <= 0997;
78: sin <= 1001;
79: sin <= 1005;
80: sin <= 1009;
81: sin <= 1012;
82: sin <= 1014;
83: sin <= 1018;
84: sin <= 1019;
85: sin <= 1020;
86: sin <= 1022;
87: sin <= 1023;
88: sin <= 1029;
89: sin <= 1024;
90: sin <= 1000;
default sin <= 0000;
endcase
```

```
case (angle2)
        90: cos <= 0000;
        89: cos <= 0017;
        88: cos <= 0035;
        87: cos <= 0053;
        86: cos <= 0070;
        85: cos <= 0087;
        84: cos <= 0105;
        83: cos <= 0122;
        82: cos <= 0139;
        81: cos <= 0156;
        80: cos <= 0174;
        79: cos <= 0191;
        78: cos <= 0208;
        77: cos <= 0230;
        76: cos <= 0242;
        75: cos <= 0259;
        74: cos <= 0276;
        73: cos <= 0293;
        72: cos <= 0309;
        71: cos <= 0326;
        70: cos <= 0342;
        69: cos <= 0358;
        68: cos <= 0375;
        67: cos <= 0391;
        66: cos <= 0407;
        65: cos <= 0423;
        64: cos <= 0438;
        63: cos <= 0454;
        62: cos <= 0469;
        61: cos <= 0485;
        60: cos <= 0500;
        59: cos <= 0515;
        58: cos <= 05230;
        57: cos <= 0545;
        56: cos <= 0559;
        55: cos <= 0574;
        54: cos <= 0588;
        53: cos <= 0602;
        52: cos <= 0616;
        51: cos <= 0629;
        50: cos <= 0643;
        49: cos <= 0656;
```

```
48: cos <= 0669;
47: cos <= 0682;
46: cos <= 0695;
45: cos <= 0707;
44: cos <= 0719;
43: cos <= 0731;
42: cos <= 0743;
41: cos <= 0755;
40: cos <= 0766;
39: cos <= 0777;
38: cos <= 0788;
37: cos <= 0799;
36: cos <= 0809;
35: cos <= 0819;
34: cos <= 0829;
33: cos <= 0839;
32: cos <= 0848;
31: cos <= 0857;
30: cos <= 0866;
29: cos <= 0875;
28: cos <= 0883;
27: cos <= 0891;
26: cos <= 0899;
25: cos <= 0906;
24: cos <= 0914;
23: cos <= 0921;
22: cos <= 0927;
21: cos <= 0934;
20: cos <= 0940;
19: cos <= 0946;
18: cos <= 0951;
17: cos <= 0956;
16: cos <= 0961;
15: cos <= 0966;
14: cos <= 0970;
13: cos <= 0974;
12: cos <= 0978;
11: cos <= 0982;
10: cos <= 0985;
09: cos <= 0988;
08: cos <= 0990;
07: cos <= 0994;
06: cos <= 0995;
```

```
                05: cos <= 0996;
                04: cos <= 0998;
                03: cos <= 0999;
                02: cos <= 0999;
                01: cos <= 0999;
                00: cos <= 1000;
                default cos <= 00000;
            endcase
            end
        endmodule

        ////////////////////////////////////////////////////////////////////////
        // genPWM
        // Generate pwm signal @50hz (20ms)
        ////////////////////////////////////////////////////////////////////////
        module genPWM  (input enable,
                                    input [15:0] pulse_width,    // pulse width is in us.
    Must be between 450 and 2449
                                    output reg pwm);

            parameter cycle = 15'h4E20;

            reg [14:0] count;
            initial begin
                count = 0;
                pwm = 1'b1;
            end

            always @(posedge enable) begin
                if (count == cycle) begin
                    pwm <= 1'b1;
                    count <= 0;
                end else if (count == pulse_width) begin
                    pwm <= 0;
                    count <= count + 1;
                end else begin
                    count <= count + 1;
                end
            end
        endmodule
        ////////////////////////////////////////////////////////////////////////
        // Divider Module
        //   Create enable signals at different frequencies
```

```
//////////////////////////////////////////////////////////////////////////
module divider #(parameter [26:0] maxCount = 27'hCDFE60)
                                        (input clock_in,
                                        output reg enable);

        reg [26:0] count = 0;

        always @(posedge clock_in) begin
                if  (count == maxCount) begin
                        count <= 0;
                        enable <= 1;
                end else begin
                        enable <= 0;
                        count <= count + 1;
                end
        end
endmodule
//////////////////////////////////////////////////////////////////////////
// EnToClck
// Creates a clock from an enable signal
//////////////////////////////////////////////////////////////////////////
module entoclk (input enable,
                                         output reg clk);
                initial begin
                        clk = 0;
                end

                always @(posedge enable) clk = ~clk;
endmodule

//////////////////////////////////////////////////////////////////////
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce #(parameter DELAY=400000)   // .01 sec with a 49Mhz clock
             (input reset, clock, noisy,
              output reg clean);

  reg [19:0] count;
  reg new;

  always @(posedge clock)
    if (reset)
```

```
      begin
            count <= 0;
            new <= noisy;
            clean <= noisy;
      end
    else if (noisy != new)
      begin
            new <= noisy;
            count <= 0;
      end
    else if (count == DELAY)
      clean <= new;
    else
      count <= count+1;

endmodule

///////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Gim P. Hom 3/22/2007
//
// Create Date:    17:51:37 03/11/2007
// Design Name:
// Module Name:    vga_general
// Project Name:
// Revision 0.01 - File Created
// Additional Comments:
//
///////////////////////////////////////////////////////////////////////////////
module vga_general(
    input pixel_clk,
    output reg [10:0] hcount,
    output reg [9:0] vcount,
    output blank,
    output hblank,
    output  hsync,
    output  vsync
);


            wire vblank;
```

```
        // 1024 x 768 @ 60hz;  alternate resolution = 640x480 75hz
        parameter hfp = 24;  //24;  // 16;
        parameter hsy = 136;  //136; // 96;
        parameter hbp = 160;  //160; // 48;

        parameter vfp = 3;  //3;  //  11;
        parameter vsy = 6;   //6;  //  2;
        parameter vbp = 29;  //29; //  32;

        parameter hsize = 1023; //1023; //639;  // there are 640 pixels counting 0
        parameter vsize = 767; //767; //479;  // similarly there are 480 lines counting line 0

        wire  h_end = (hcount == (hsize + hfp + hsy + hbp));
        wire  v_end = (vcount == (vsize + vfp + vsy + vbp));

        assign hsync = ((hcount < hsize + hfp) || (hcount > hsize + hfp + hsy));
        assign vsync = ((vcount < vsize + vfp) || (vcount > vsize + vfp + vsy));

        assign hblank = (hcount <= hsize);
        assign vblank = (vcount <= vsize);

        assign blank = hblank && vblank;

        always @(posedge pixel_clk)
          begin
                hcount <= h_end ? 0 : hcount + 1;
                vcount <= h_end ? (v_end ? 0 : vcount + 1) : vcount;
                end

endmodule

////////////////////////////////////////////////////////////////////////////////
// Company: Digilent Inc 2011
// Engineer: Michelle Yu
// Create Date:    08/26/2011
// Module Name:    dcm_all
// Project Name:     PmodPS2_Demo
// Target Devices: Nexys3
// Tool version:     ISE 14.2
// Description: This file contains the design for a dcm that generates a 25MHz and a

//              40MHz clock from a 100MHz clock.
//
```

```
        // Revision:
        // Revision 0.01 - File Created
        // Revision 1.00 - Converted from VHDL to Verilog (Josh Sackos)

        // Revision 2.00 - removed CLK25, add parameters for divide/multiply

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

        //
==============================================================================
==================
        //                          Define Module
        //
==============================================================================
==================
        module dcm_all_v2  #(parameter DCM_DIVIDE = 4,

                                DCM_MULTIPLY = 2)

           (

            CLK,

        //  RST,

            CLKSYS,

        //    CLK25,

            CLK_out

        );


        //
==============================================================================
==================
        //                          Port Declarations
        //
==============================================================================
==================
        input   CLK;
//        input   RST;
        output  CLKSYS;
//        output  CLK25;
        output  CLK_out;


//
================================================================================
==================
//                    Parameters, Registers, and Wires
//
================================================================================
==================


        // Output registers

        wire CLKSYS;

        wire CLK25;

        wire CLK_out;



        // architecture of dcm_all entity
        wire    GND = 1'b0;
        wire    CLKSYSint;
        wire    CLKSYSbuf;

        assign CLKSYS = CLKSYSbuf;


//
================================================================================
==================
//                           Implementation
//
================================================================================
==================
        // buffer system clock and wire to dcm feedback
        BUFG BUFG_clksys(

                .O(CLKSYSbuf),

                .I(CLKSYSint)

        );

        // Instantiation of the DCM device primitive.
        // Feedback is not used.
        // Clock multiplier is 2
        // Clock divider is 5
        // 100MHz * 2/5 = 40MHz
        // The following generics are only necessary if you wish to change the default
behavior.
        DCM #(

                .CLK_FEEDBACK("1X"),
                .CLKDV_DIVIDE(4.0),                 //  Divide by:
1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5,6.0,6.5
                                                        //            7.0,7.5,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0
or 16.0
                .CLKFX_DIVIDE(DCM_DIVIDE),           //  Can be any interger from 2 to 32
                .CLKFX_MULTIPLY(DCM_MULTIPLY),       //  Can be any integer from 2 to 32
                .CLKIN_DIVIDE_BY_2("FALSE"),         //  TRUE/FALSE to enable CLKIN
divide by two feature
                .CLKIN_PERIOD(10000.0),              //  Specify period of input clock (ps)
                .CLKOUT_PHASE_SHIFT("NONE"),         //  Specify phase shift of NONE,
FIXED or VARIABLE
                .DESKEW_ADJUST("SYSTEM_SYNCHRONOUS"), //
SOURCE_SYNCHRONOUS, SYSTEM_SYNCHRONOUS or
                                                        //       an integer from 0 to 15
                .DFS_FREQUENCY_MODE("LOW"),          //  HIGH or LOW frequency mode
for frequency synthesis
                .DLL_FREQUENCY_MODE("LOW"),          //  HIGH or LOW frequency mode
for DLL
                .DUTY_CYCLE_CORRECTION("TRUE"),      //  Duty cycle correction, TRUE
or FALSE
                .FACTORY_JF(16'hC080),               //  FACTORY JF Values
                .PHASE_SHIFT(0),                     //  Amount of fixed phase shift from -255 to
255
```

```
            .STARTUP_WAIT("FALSE")          //  Delay configuration DONE until DCM
LOCK, TRUE/FALSE

        )

        DCM_inst(

            .CLK0(CLKSYSint),               // 0 degree DCM CLK ouptput

            .CLK180(),                      // 180 degree DCM CLK output

            .CLK270(),                      // 270 degree DCM CLK output

            .CLK2X(),                       // 2X DCM CLK output

            .CLK2X180(),                    // 2X, 180 degree DCM CLK out

            .CLK90(),                       // 90 degree DCM CLK output

            .CLKDV(), //(CLK25),            // Divided DCM CLK out (CLKDV_DIVIDE)

            .CLKFX(CLK_out),                // DCM CLK synthesis out (M/D)

            .CLKFX180(),                    // 180 degree CLK synthesis out

            .LOCKED(),                      // DCM LOCK status output

            .PSDONE(),                      // Dynamic phase adjust done output

            .STATUS(),                      // 8-bit DCM status bits output

            .CLKFB(CLKSYSbuf),              // DCM clock feedback

            .CLKIN(CLK),                    // Clock input (from IBUFG, BUFG or DCM)

            .PSCLK(GND),                    // Dynamic phase adjust clock input

            .PSEN(GND),                     // Dynamic phase adjust enable input

            .PSINCDEC(GND),                 // Dynamic phase adjust
increment/decrement

            .DSSEN(1'b0),
               .RST (1'b0)  //(RST)              // DCM asynchronous reset input

          );

     endmodule


     ////////////////////////////////////////////////////////
     // Asynchronous UART receiver
     //
     ////////////////////////////////////////////////////////
     module receiver(
          input clk,
          input RxD,
          output reg RxD_data_ready = 0,
          output reg [7:0] RxD_data = 0,  // data received, valid only (for one clock cycle)
when RxD_data_ready is high
          output RxD_idle,  // asserted when no data has been received for a while
     );

     parameter ClkFrequency = 25000000; // 25MHz
     parameter Baud = 9600;

     parameter Oversampling = 8;
     // sample each bit 8 times

     reg [3:0] RxD_state = 0;

     wire tick;
     baudGen #(ClkFrequency, Baud, Oversampling) tickgen(.clk(clk), .enable(1'b1),
.tick(tick));


     reg [1:0] sync = 2'b11;

     reg [1:0] count = 2'b11;
     reg RxD_bit = 1'b1;



     function integer log2(input integer v); begin log2=0; while(v>>log2) log2=log2+1; end
endfunction
```

```
        localparam l2o = log2(Oversampling);
        reg [l2o-2:0] OversamplingCnt = 0;
        always @(posedge clk) begin
        // synchronize RxD to clk
        if(tick) sync <= {sync[0], RxD}

        // decide when to sample the RxD line
        if(tick)
        begin
                if(sync[1]==1'b1 && count!=2'b11) count <= count + 1'd1;
                else
                if(sync[1]==1'b0 && count!=2'b00) count <= count - 1'd1;

                if(count==2'b11) RxD_bit <= 1'b1;
                else
                if(count==2'b00) RxD_bit <= 1'b0;
        end

        if(tick) OversamplingCnt <= (RxD_state==0) ? 1'd0 : OversamplingCnt + 1'd1;
        wire sample = tick && (OversamplingCnt==Oversampling/2-1);

        //accumulate the RxD bits in a register

        case(RxD_state)
                4'b0000: if(~RxD_bit) RxD_state <= 4'b0001 ;  // start bit found?
                4'b0001: if(sample) RxD_state <= 4'b1000;  // sync start bit to sample
                4'b1000: if(sample) RxD_state <= 4'b1001;  // bit 0
                4'b1001: if(sample) RxD_state <= 4'b1010;  // bit 1
                4'b1010: if(sample) RxD_state <= 4'b1011;  // bit 2
                4'b1011: if(sample) RxD_state <= 4'b1100;  // bit 3
                4'b1100: if(sample) RxD_state <= 4'b1101;  // bit 4
                4'b1101: if(sample) RxD_state <= 4'b1110;  // bit 5
                4'b1110: if(sample) RxD_state <= 4'b1111;  // bit 6
                4'b1111: if(sample) RxD_state <= 4'b0010;  // bit 7
                4'b0010: if(sample) RxD_state <= 4'b0000;  // stop bit
                default: RxD_state <= 4'b0000;
        endcase

        if(sample && RxD_state[3]) RxD_data <= {RxD_bit, RxD_data[7:1]};

        RxD_data_ready <= (sample && RxD_state==4'b0010 && RxD_bit);  // done when a
stop bit is received
        end
```

```
	endmodule


	module baudGen(
		input clk, enable,
		output tick  // generate a tick at the specified baud rate * oversampling
	);
	parameter ClkFrequency = 25000000;
	parameter Baud = 9600;
	parameter Oversampling = 1;

	function integer log2(input integer v); begin log2=0; while(v>>log2) log2=log2+1; end
endfunction
	localparam AccWidth = log2(ClkFrequency/Baud)+8;  // +/- 2% max timing error over a
byte
	reg [AccWidth:0] Acc = 0;
	localparam ShiftLimiter = log2(Baud*Oversampling >> (31-AccWidth));  // this makes
sure Inc calculation doesn't overflow
	localparam Inc = ((Baud*Oversampling <<
(AccWidth-ShiftLimiter))+(ClkFrequency>>(ShiftLimiter+1)))/(ClkFrequency>>ShiftLimiter);
	always @(posedge clk) if(enable) Acc <= Acc[AccWidth-1:0] + Inc[AccWidth:0]; else Acc
<= Inc[AccWidth:0];
	assign tick = Acc[AccWidth];
	endmodule








	//
	// File:   zbt_6111_sample.v
	// Date:   26-Nov-05
	// Author: I. Chuang <ichuang@mit.edu>
	//
	// Sample code for the MIT 6.111 labkit demonstrating use of the ZBT
	// memories for video display.  Video input from the NTSC digitizer is
	// displayed within an XGA 1024x768 window.  One ZBT memory (ram0) is used
	// as the video frame buffer, with 8 bits used per pixel (black & white).
	//
```

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

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2011-Nov-10: Changed resolution to 1024 * 768.
//                                          Added back ramclok to deskew RAM clock
//
// 2009-May-11: Fixed memory management bug by 8 clock cycle forecast.
//              Changed resolution to  800 * 600.
//              Reduced clock speed to 40MHz.
//              Disconnected zbt_6111's ram_clk signal.
//              Added ramclock to control RAM.
//              Added notes about ram1 default values.
//              Commented out clock_feedback_out assignment.
//              Removed delayN modules because ZBT's latency has no more effect.
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
```

```
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//             value. (Previous versions of this file declared this port to
//             be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//             actually populated on the boards. (The boards support up to
//             72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

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

              clock_feedback_out, clock_feedback_in,

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

              rs232_txd, rs232_rxd, rs232_rts, rs232_cts,
               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

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

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

               switch,

               led,

               user1, user2, user3, user4,

               daughtercard,

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

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

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

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

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

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

```
inout  tv_in_i2c_data;

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

inout  mouse_clock, mouse_data;
        input  keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

inout  [15:0] systemace_data;
output [6:0]  systemace_address;
```

```
   output systemace_ce_b, systemace_we_b, systemace_oe_b;
   input  systemace_irq, systemace_mpbrdy;

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

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

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

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

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

   // SRAMs

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

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

/* enable RAM pins */

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

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

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

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

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

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

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

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

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

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

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

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

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

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

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


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

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

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

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


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

	// ENTER button is user reset
	wire reset,user_reset;
	debounce db1(power_on_reset, clk, ~button_enter, user_reset);
	assign reset = user_reset | power_on_reset;

	//up, down, left and right buttons are used as controls
		wire up,down, left, right;
	debounce db2(.reset(reset),.clk(clk),.noisy(~button_up),.clean(up));
	debounce db3(.reset(reset),.clk(clk),.noisy(~button_down),.clean(down));
		debounce db4(.reset(reset),.clk(clk),.noisy(~button_left),.clean(left));
		debounce db5(.reset(reset),.clk(clk),.noisy(~button_right),.clean(right));

	//divider is used to create a pulse every one second
		wire one_sec;

		one_divider one_div1 (.clk(clock_27mhz), .reset(reset),
.oneHz_enable(one_sec));

		wire txd, txd_busy; //transmit line for XBee

//	This code implements the button control mode - if a button is pressed at
//	any point, that data is latched and every second a packet is transmitted
//	with the appropriate movement command
```

```
//
//         reg move_data, up_latch, right_latch, down_latch, left_latch;
//         reg tx_start;
//         reg [6:0] ang_data;
//
//
//         always @(posedge one_sec) begin
//                 up_latch <= up;
//                 right_latch <= right;
//                 down_latch <= down;
//                 left_latch <= left;
//                 end
//
//         always @(posedge clk) begin
//                 if (one_sec) begin
//                         tx_start <= 1;  //raise line to signal packet
//                            //transmission
//                         if (up_latch|down_latch|left_latch|right_latch) begin //if any button
has been pressed
//                                 move_data <= 1'h1;                          //send appropriate
movement command
//                                 if (up_latch)
//                                         ang_data <= 7'h00;
//                                 else if (right_latch)
//                                         ang_data <= 7'h01;
//                                 else if (down_latch)
//                                         ang_data <= 7'h02;
//                                 else if (left_latch)
//                                         ang_data <= 7'h03;
//                                 end
//                         else begin
//                                 move_data <= 1'h0;                    //else, no movement,
default angle of 0 degrees
//                                 ang_data <= 7'h00;
//                                 end
//                 end
//                 else begin
//                         if (tx_start)
//                                 tx_start <= 0; //only keep line raised for one clock cycle
//                         end
//                 end
//
//
```

```
	//		transmitter trans(.clk(clk), .TxD_start(tx_start), .TxD_data({move_data, 
ang_data}), .TxD_busy(txd_busy), .TxD(txd)); //transmit movement command packet
	//



			wire [11:0] mx, my;  //mouse x and y coordinates
			wire [2:0] btn_click; //left - btn_click[2], middle - btn_click[1], right button - 
btn_click[0]
			wire [17:0] mouse_pixel; //used to draw square representing mouse

			ps2_mouse_xy mouse_click(.clk(clk), .reset(reset), .mx(mx), .my(my),

.ps2_clk(mouse_clock), .ps2_data(mouse_data),

.btn_click(btn_click)); //get mouse data

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

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

		blob mouse_track(.x(mx[10:0]), .hcount(hcount), .y(my[9:0]), .vcount(vcount),
								.alt_pixel(18'h0),
.pixel(mouse_pixel)); //square that represents mouse


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

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

	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));

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

		YCrCb2RGB convert (.clk(tv_in_line_clock1), .rst(reset), .Y(ycrcb[29:20]),
.Cr(ycrcb[19:10]),
									.Cb(ycrcb[9:0]), .R(R), .G(G),
.B(B)); //Convert 30 bits of ycrcb to 24 bits of RGB

		wire [7:0] vr_pixel_H, vr_pixel_S, vr_pixel_V;

		rgb2hsv con (.clock(tv_in_line_clock1), .reset(reset), .r({vr_pixel[17:12], 2'b0}),
.g({vr_pixel[11:6], 2'b0}), .b({vr_pixel[5:0], 2'b0}), .h(vr_pixel_H),
							.s(vr_pixel_S), .v(vr_pixel_V));

	// code to write NTSC data to video memory

	wire [18:0] ntsc_addr;
```

```
   wire [35:0] ntsc_data;
   wire        ntsc_we;
   ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, {R[7:2], G[7:2], B[7:2]},
                    ntsc_addr, ntsc_data, ntsc_we, switch[6]); //write 18 bits of RGB to ZBT

   // 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'd1) : 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   b,hs,vs;
//       wire vr_pix_h_range, vr_pix_s_range, vr_pix_v_range;
//
//       assign vr_pix_h_range = ((vr_pixel_H >= 8'h00) & (vr_pixel_H <= 8'h1E)) ? 1 : 0;
//       assign vr_pix_s_range = ((vr_pixel_S >= 8'h3C) & (vr_pixel_S <= 8'hFF)) ? 1 : 0;
//       assign vr_pix_v_range = ((vr_pixel_V >= 8'h3C) & (vr_pixel_V <= 8'hFF)) ? 1 : 0;

         wire [17:0] track_pixel, track_pixel2;
         wire [10:0] x_center, x_posit2;
         wire [9:0] y_center, y_posit2;

//   detection detect(.clock(clk), .reset(reset), .hsync(hsync), .vsync(vsync), .blank(blank),
	//							.vr_pixel(vr_pixel), .x_posit(x_posit),
.y_posit(y_posit), .hcount(hcount),
	//							.vcount(vcount),
.track_pixel(track_pixel));
	//
	  detection detect(.clock(clk), .reset(reset), .hsync(hsync), .vsync(vsync), .blank(blank),
								.vr_pixel(vr_pixel),
.x_center(x_center), .y_center(y_center), .hcount(hcount),
								.h(vr_pixel_H), .s(vr_pixel_S),
.v(vr_pixel_V), .vcount(vcount), .track_pixel(track_pixel));  //detects a red object

		detection #(.GREEN_MIN(6'h1F), .GREEN_MAX(6'h3F))
						detect2 (.clock(clk), .reset(reset), .hsync(hsync),
.vsync(vsync), .blank(blank),
								.vr_pixel(vr_pixel),
.x_center(x_posit2), .y_center(y_posit2), .hcount(hcount),
								.h(vr_pixel_H), .s(vr_pixel_S),
.v(vr_pixel_V), .vcount(vcount), .track_pixel(track_pixel2)); //detects a yellow object

		wire [10:0] distx;
		wire [9:0] disty;
		wire move_data;
		wire [1:0] ang_data;

		move2point m2p(.clk(clk), .reset(reset), .btn_click(btn_click[2]),
.one_sec(one_sec),

.x_posit(x_center), .y_posit(y_center), .point_x(x_posit2),

.point_y(y_posit2), .distx(distx), .disty(disty), .txd(txd), //implements move to point algorithm

.move_data(move_data), .ang_data(ang_data));		//currently goes to second tracked
object

		assign user1[0] = txd; //transmit line is pin 0 of user1 i/o


		reg [10:0] destx;
		reg [9:0] desty;

	//	always @(posedge clk) begin
	//		if (x_center > mx[10:0])
	//			distx <= x_center - destx;
```

```
//                      else
//                              distx <= destx - x_center;
//
//                      if (y_center > my[9:0])
//                              disty <= y_center - desty;
//                      else
//                              disty <= desty - y_center;
//              end


                always @(posedge btn_click[2]) begin //stores mouse button click
                        destx <= mx[10:0];
                        desty <= my[9:0];
                        end

           always @(posedge clk)
             begin
                pixel <= track_pixel|track_pixel2|mouse_pixel; //combines video data and squares
that mark tracked objects and mouse
                b <= 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_red = pixel;
//   assign vga_out_green = pixel;
//   assign vga_out_blue = pixel;
           assign vga_out_sync_b = 1'b1;    // not used
           assign vga_out_pixel_clock = ~clk;
           assign vga_out_blank_b = ~b;
           assign vga_out_hsync = hs;
           assign vga_out_vsync = vs;

           // debugging

           assign led = ~{vram_addr[18:13],reset, switch[0]};

           always @(posedge clk)
```

```
                    dispdata <= {distx, 7'b0, disty, 28'b0, move_data, 6'b0, ang_data};

endmodule

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

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

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

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

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

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

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

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

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

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

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

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

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

  // sync and blanking
  wire     next_hblank,next_vblank;
```

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

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

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

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

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

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


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

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

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

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

  wire   hc2 = hcount[0];
  reg [17:0]      vr_pixel;
  reg [35:0]      vr_data_latched;
  reg [35:0]      last_vr_data;

  always @(posedge clk)
    last_vr_data <= (hc2==1) ? vr_data_latched : last_vr_data;

  always @(posedge clk)
    vr_data_latched <= (hc2==0) ? vram_read_data : vr_data_latched;

  always @(*)            // each 36-bit word from RAM is decoded to 2 bytes
    case (hc2)
      0: vr_pixel = last_vr_data[17:0];
      1: 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

        module one_divider(
           input clk,
           input reset,
           output reg oneHz_enable = 0
           );

                reg[27:0] counter = 0;

        //This module simply counts to 27000000 and asserts a signal high on that count (so
roughly every 1
        //second).  It resets to 0 if reset is asserted or if a new timer needs to start (so that every
count
        //is a full second).

                always @(posedge clk) begin
                        if(reset) begin //reset conditions
                                counter <= 0;
                                oneHz_enable <= 0;
                                end
                        else
                                begin
                                if (counter == (27000000-1))
```

```
                                        begin
                                        oneHz_enable <= 1; //assert a signal every 27000000
counts
                                        counter <= 0;
                                        end
                                else
                                        begin
                                        counter <= counter + 1;
                                        oneHz_enable <= 0;
                                        end
                                end
                end

        endmodule

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

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

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

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

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

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

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

                assign ref_clk = ref_clock;

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

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

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

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

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

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


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

    assign locked = lock1 && lock2;

  endmodule

  module detection (clock, reset, hsync, vsync, blank, vr_pixel, hcount, h, s, v, vcount,
track_pixel, x_center, y_center);

  //This module detects a certain color, finds the center of the color on the screen and
tracks that point

        parameter RED_MIN = 6'h1F; //these set RGB ranges - default is to detect red
        parameter RED_MAX = 6'h3F;
        parameter GREEN_MIN = 6'h00;
        parameter GREEN_MAX = 6'h0F;
        parameter BLUE_MIN = 6'h00;
        parameter BLUE_MAX = 6'h0F;

        input clock, reset, hsync, vsync, blank;
```

```
 input [17:0] vr_pixel;
 input [10:0] hcount;
 input [9:0] vcount;
 input [7:0] h, s, v;
 output [17:0] track_pixel;
 output [10:0] x_center;
 output [9:0] y_center;

 wire new;
 wire [12:0] sum_up_x;
 wire [11:0] sum_up_y;
 reg [18:0] thresh_pixel;
 reg [24:0] x_accumulator = 0;
 reg [24:0] y_accumulator = 0;
 reg [24:0] x_count = 0;
 reg [24:0] y_count = 0;
 reg [24:0] dividend_x = 0;
 reg [24:0] dividend_y = 0;
 reg [24:0] divisor_x = 0;
 reg [24:0] divisor_y = 0;
 reg [43:0] x_pix_buffer = 0;
 reg [39:0] y_pix_buffer = 0;
 wire [24:0] x_quotient;
 wire [24:0] y_quotient;
 wire [24:0] x_remainder;
 wire [24:0] y_remainder;
 wire x_rfd;
 wire y_rfd;

//dividers for center coordinates
 center_divider x_div(.clk(clock),
                                                              .dividend(dividend_x),
                                                              .divisor(divisor_x),
                                                              .quotient(x_quotient),
                                                              .fractional(x_remainder),
                                                              .rfd(x_rfd));

 center_divider y_div(
                                                             .clk(clock),
                                                             .dividend(dividend_y),
                                                             .divisor(divisor_y),
                                                             .quotient(y_quotient),
                                                             .fractional(y_remainder),
                                                            .rfd(y_rfd));



          wire vr_pix_r_range, vr_pix_g_range, vr_pix_b_range;
          //wire vr_pix_h_range, vr_pix_s_range, vr_pix_v_range;

          assign vr_pix_r_range = ((vr_pixel[17:12] >= RED_MIN) & (vr_pixel[17:12] <=
RED_MAX)) ? 1 : 0; //determine if pixel is within range for RGB
          assign vr_pix_g_range = ((vr_pixel[11:6] >= GREEN_MIN) & (vr_pixel[11:6] <=
GREEN_MAX)) ? 1 : 0;
          assign vr_pix_b_range = ((vr_pixel[5:0] >= BLUE_MIN) & (vr_pixel[5:0] <=
BLUE_MAX)) ? 1 : 0;

          //assign vr_pix_h_range = (h >= 230) ? 1 : 0;
          //assign vr_pix_s_range = (s >= 160) ? 1 : 0;
          //assign vr_pix_v_range = (v >= 100) ? 1 : 0;



          always @(posedge clock) begin
               if ((hcount>11'd30 && hcount < 11'd740) && (vcount>10'd80 &&
vcount<10'd550)) //limit calculations to actual video feed
                    thresh_pixel <=(vr_pix_r_range & vr_pix_g_range &
vr_pix_b_range)? 18'h3FFFF : 0; //if within range, assign white pixel, else black
               else
                    thresh_pixel <= 0;
               end

          always @(posedge clock) begin
               if (hcount == 11'd0 & vcount == 10'd0) begin              //reset
accumulators with every frame
                    x_accumulator <= 0;
                    y_accumulator <= 0;
                    x_count <= 0;
                    y_count <= 0;
                    end
               else if ((hcount>11'd30 && hcount < 11'd740) & (vcount>10'd80 &&
vcount<10'd550)) begin
                    if (thresh_pixel == 18'h3FFFF) begin
                         x_accumulator <= x_accumulator + hcount; //if pixel is
within range, add it's position to accumulators, and add one to each count
                         y_accumulator <= y_accumulator + vcount;
```

```
					x_count <= x_count + 1;
					y_count <= y_count + 1;
					end
				else begin
					x_accumulator <= x_accumulator; //else nothing changes
					y_accumulator <= y_accumulator;
					x_count <= x_count;
					y_count <= y_count;
					end
				end
			else if (hcount == 11'd0 & vcount == 10'd551) begin
				dividend_x <= (x_count > 30)?x_accumulator:0; //if there are more
than 30 points (noise) start dividers
				dividend_y <= (y_count > 30)?y_accumulator:0;
				divisor_x <= (x_count == 0)?1:x_count;
				divisor_y <= (y_count == 0)?1:y_count;
				end
			end

		assign new = (hcount==11'd1 && vcount == 10'd552); //for buffer

		always @(posedge clock) begin
			if (new) begin
				x_pix_buffer <= {x_pix_buffer[32:0], x_quotient[10:0]}; //buffer
stores 4 points
				y_pix_buffer <= {y_pix_buffer[29:0], y_quotient[9:0]};
				end
			end

		assign sum_up_x = x_pix_buffer[43:33] + x_pix_buffer[32:22] +
x_pix_buffer[21:11] + x_pix_buffer[10:0];
		assign sum_up_y = y_pix_buffer[39:30] + y_pix_buffer[29:20] +
y_pix_buffer[19:10] + y_pix_buffer[9:0];

		assign x_center = sum_up_x >> 2;	//average coordinates over time creates a
smoother tracking
		assign y_center = sum_up_y >> 2;


		blob tracker(.x(x_center), .hcount(hcount), .y(y_center), .vcount(vcount),
						.alt_pixel(thresh_pixel), .pixel(track_pixel)); //draw
square to track object
```

```
endmodule

module move2point(
        input clk,
        input reset,
        input btn_click,
        input one_sec,
        input [10:0] x_posit,
        input [9:0] y_posit,
        input [10:0] point_x,
        input [9:0] point_y,
        output txd,
        output reg [10:0] distx,
        output reg [9:0] disty,
        output reg move_data,
        output reg [1:0] ang_data
   );

        reg tx_start;
        reg stopped = 0;
        reg clicked;
        reg start = 0;
        //reg [1:0] ang_data;
        //reg move_data;
        reg [10:0] x_posit_last = 0;
        reg [9:0] y_posit_last = 0;
        reg [10:0] distx_last, x_posit_latched, point_x_latched;
        reg [9:0] disty_last, y_posit_latched, point_y_latched;
        wire txd_busy;

//This module implements the move to point algorithm.  It uses visual feedback to
determine where the hexapod is and where it should be going.

        always @(posedge one_sec) begin
                x_posit_latched <= x_posit; //latch points in case they change
                y_posit_latched <= y_posit;
                point_x_latched <= point_x;
                point_y_latched <= point_y;

                if (x_posit > point_x)
                        distx <= x_posit - point_x; //calculate distances, make sure that the
distance is positive.
```

```
                    else
                              distx <= point_x - x_posit;

                    if (y_posit > point_y)
                              disty <= y_posit - point_y;
                    else
                              disty <= point_y - y_posit;

                    if (x_posit_last > point_x)
                              distx_last <= x_posit_last - point_x;
                    else
                              distx_last <= point_x - x_posit_last;

                    if (y_posit_last > point_y)
                              disty_last <= y_posit_last - point_y;
                    else
                              disty_last <= point_y - y_posit_last;
                    end

          always @(posedge clk) begin
                    if (btn_click)  //only start moving if the mouse button has actually been
clicked
                              clicked <= 1;
                    else if (start)
                              clicked <= 0;

                    if (point_x < 11'd30 | point_x > 11'd740| point_y <10'd80 | point_y>10'd550)
//if a click occurs outside of the field, immediately stop
                              move_data <= 0;

                    else if (one_sec & !stopped) begin //every second, determine movement
to be taken
                              tx_start <= 1; //raise line to signal packet sending
                              if (clicked) begin
                                        start <= 1;
                                        move_data <= 1;  //initially move straigt to calibrate
                                        ang_data <= 2'd0;
                                        x_posit_last <= x_posit_latched; //save current point as last
point
                                        y_posit_last <= y_posit_latched;
                                        end

                              else if (start) begin      //compare distances and points
```

```
                                        if ((distx < 11'd150) & (disty < 10'd150)) begin //if hexapod
is close to destination, stop moving
                                                stopped <= 1;
                                                start <= 0;
                                                end
                                        else if ((distx < distx_last) & (disty < disty_last)) //if hexapod
got closer to destination, go in same direction
                                                ang_data <= ang_data;

                                        else if ((distx > distx_last) & (disty > disty_last)) //if it got
farther away, turn around (change by 180 degrees)
                                                ang_data <= ang_data + 2'd2;

                                        else if ((distx > distx_last) & (disty < disty_last)) begin //if
closer in one direction but not the other
                                                if
(((x_posit_latched<point_x_latched)&(y_posit_latched<point_y_latched))|((x_posit_latched>point
_x_latched)&(y_posit_latched>point_y_latched))) //compare positions and determine appropriate
90 degree turn
                                                        ang_data <= ang_data - 2'd1;
                                                else
                                                        ang_data <= ang_data + 2'd1;
                                                end

                                        else begin //see above
                                                if
(((x_posit_latched<point_x_latched)&(y_posit_latched<point_y_latched))|((x_posit_latched>point
_x_latched)&(y_posit_latched>point_y_latched)))
                                                        ang_data <= ang_data + 2'd1;
                                                else
                                                        ang_data <= ang_data - 2'd1;
                                                end
                                        end

                                else begin
                                        stopped <= 1; //don't move unless there's a point to move
to
                                        ang_data <= 2'd0;
                                        end
                                end
                        else if (stopped) begin //stop immediately
                                stopped <= 0;
                                move_data <= 0;
                                end
                        else begin
                                if (tx_start) //only keep line high for one clock cycle
                                        tx_start <= 0;
                                end
                        end

                transmitter transmit2 (.clk(clk), .TxD_start(tx_start), .TxD_data({move_data,
5'b00000, ang_data}), .TxD(txd), .TxD_busy(txd_busy));     //transmit movement packets

                                                        //via Xbee


endmodule

module debounce (reset, clk, noisy, clean);
  input reset, clk, noisy;
  output clean;

  parameter NDELAY = 650000;
  parameter NBITS = 20;

  reg [NBITS-1:0] count;
  reg xnew, clean;

  always @(posedge clk)
    if (reset) begin xnew <= noisy; clean <= noisy; count <= 0; end
    else if (noisy != xnew) begin xnew <= noisy; count <= 0; end
    else if (count == NDELAY) clean <= xnew;
    else count <= count+1;

endmodule

// ps2_mouse_xy gives a high-level interface to the mouse, which
// keeps track of the "absolute" x,y position (within a parameterized
// range) and also returns button presses.

module ps2_mouse_xy(clk, reset, ps2_clk, ps2_data, mx, my, btn_click);



  input clk, reset;
  inout ps2_clk, ps2_data;     // data to/from PS/2 mouse
  output [11:0] mx, my;          // current mouse position, 12 bits
  output [2:0]  btn_click;       // button click: Left-Middle-Right

  // module parameters
  parameter     MAX_X = 1023;
  parameter     MAX_Y = 767;

  // low level mouse driver

  wire [8:0]      dx, dy;
  wire [2:0]      btn_click;
  wire   data_ready;
  wire   error_no_ack;
  wire [1:0]      ovf_xy;
  wire   streaming;

//   original 6.111 fall 2005 Verilog - appears to be buggy  so it has been
//   commented out.
//   ps2_mouse m1(clk,reset,ps2_clk,ps2_data,dx,dy,ovf_xy, btn_click,
//                  data_ready,streaming);
//


// using ps2_mouse Verilog from Opencore

// divide the clk by a factor of two sot that it works with 65mhz and the original timing
// parameters in the open core source.
// if the Verilog doesn't work the user should update the timing parameters. This  Verilog
assumes
// 50Mhz clock; seems to work with 32.5mhz without problems. GPH  11/23/2008 with
// assist from BG

ps2_mouse_interface

  #(.WATCHDOG_TIMER_VALUE_PP(26000),
  .WATCHDOG_TIMER_BITS_PP(15),
  .DEBOUNCE_TIMER_VALUE_PP(246),
  .DEBOUNCE_TIMER_BITS_PP(8))

  m1(
 .clk(clk),
 .reset(reset),
 .ps2_clk(ps2_clk),
  .ps2_data(ps2_data),
  .x_increment(dx),
  .y_increment(dy),
  .data_ready(data_ready),
  .read(1'b1),  // force a read
  .left_button(btn_click[2]),
  .right_button(btn_click[0])  // rx_read_o
  );

//  error_no_ack  not used


   // Update "absolute" position of mouse

   reg [11:0]  mx, my;
   wire        sx = dx[8];              // signs
   wire        sy = dy[8];
   wire [8:0]  ndx = sx ? {0,~dx[7:0]}+1 : {0,dx[7:0]};    // magnitudes
   wire [8:0]  ndy = sy ? {0,~dy[7:0]}+1 : {0,dy[7:0]};

   always @(posedge clk) begin
     mx <= reset ? 0 :
            data_ready ? (sx ? (mx>ndx ? mx - ndx : 0)
                             : (mx < MAX_X - ndx ? mx+ndx : MAX_X)) : mx;
     // note Y is flipped for video cursor use of mouse
     my <= reset ? 0 :
            data_ready ? (sy ? (my < MAX_Y - ndy ? my+ndy : MAX_Y)
                             : (my>ndy ? my - ndy : 0))  : my;
//          data_ready ? (sy ? (my>ndy ? my - ndy : 0)
//                           : (my < MAX_Y - ndy ? my+ndy : MAX_Y)) : my;
   end

endmodule

//-------------------------------------------------------------------------------------
//
// Author: John Clayton
// Date  : April 30, 2001
// Update: 6/06/01 copied this file from ps2.v (pared down).
// Update: 6/07/01 Finished initial coding efforts.
// Update: 6/09/01 Made minor changes to state machines during debugging.
//              Fixed errors in state transitions.  Added state to m2
//              so that "reset" causes the mouse to be initialized.
//              Removed debug port.
//
//
//
//
//
// Description
//-----------------------------------------------------------------------------------
// This is a state-machine driven serial-to-parallel and parallel-to-serial
// interface to the ps2 style mouse.  The state diagram for part of the
// m2 state machine was obtained from the work of Rob Chapman, as published
// at:
// www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1998_w/mouse_notes.html
//
//
// Some aspects of the mouse interface are not implemented (e.g, verifying
// the FA response code from the mouse when enabling streaming mode.)
// However, the mouse interface was designed so that "hot plugging" a mouse
// into the connector should cause the interface to send the F4 code to the
// mouse in order to enable streaming.  By this means, the mouse begins to
// operate, and no reset pulse should be needed.
//
// Similarly, there is a "watchdog" timer implemented, so that during periods
// of inactivity, the bit_count is cleared to zero.  Therefore, the effects of
// a bad count value are corrected, and internal errors of that type are not
// propagated into subsequent packet receive operations.
//
// To enable the streaming mode, F4 is sent to the mouse.
// The mouse responds with FA to acknowledge the command, and then enters
// streaming mode at the default rate of 100 packets per second (transmission
// of packets ceases when the activity at the mouse is not longer sensed.)
//
// There are additional commands to change the sampling rate and resolution
// of the mouse reported data.  Those commands are not implemented here.
// (E8,XX = set resolution 0,1,2,3)
// (E7 = set scaling 2:1)
// (E6 = reset scaling)
// (F3,XX = set sampling rate to XX packets per second.)
//
// At this time I do not know any of the command related to using the
// wheel of a "wheel mouse."
//
// The packets consists of three bytes transmitted in sequence.  The interval
// between these bytes has been measured on two different mice, and found to
// be different.  On the slower (older) mouse it was approximately 345
// microseconds, while on a newer "wheel" mouse it was approximately 125
// microseconds.  The watchdog timer is designed to cause processing of a
// complete packet when it expires.  Therefore, the watchdog timer must last
// for longer than the "inter-byte delay" between bytes of the packet.
// I have set the default timer value to 400 usec, for my 49.152 MHz clock.
// The timer value and size of the timer counter is settable by parameters,
// so that other clock frequencies and settings may be used.  The setting for
// the watchdog timeout is not critical -- it only needs to be greater than
// the inter-byte delay as data is transmitted from the mouse, and no less
// than 60usec.
//
// Each "byte" of the packet is transmitted from the mouse as follows:
//
// 1 start bit, 8 data bits, 1 odd parity bit, 1 stop bit. == 11 bits total.
// (The data bits are sent LSB first)
//
// The data bits are formatted as follows:
//
// byte 0: YV, XV, YS, XS, 1, 0, R, L
// byte 1: X7..X0
// byte 2: Y7..Y0
//
// Where YV, XV are set to indicate overflow conditions.
//       XS, YS are set to indicate negative quantities (sign bits).
//        R,  L are set to indicate buttons pressed, left and right.
//
//
//
// The interface to the ps2 mouse (like the keyboard) uses clock rates of
// 30-40 kHz, dependent upon the mouse itself.  The mouse generates the
// clock.
// The rate at which the state machine runs should be at least twice the
// rate of the ps2_clk, so that the states can accurately follow the clock
// signal itself.  Four times oversampling is better.  Say 200kHz at least.
// In order to run the state machine extremely fast, synchronizing flip-flops
// have been added to the ps2_clk and ps2_data inputs of the state machine.
// This avoids poor performance related to slow transitions of the inputs.
//
// Because this is a bi-directional interface, while reading from the mouse
// the ps2_clk and ps2_data lines are used as inputs.  While writing to the
// mouse, however (which is done when a "packet" of less than 33 bits is
// received), both the ps2_clk and ps2_data lines are sometime pulled low by
// this interface.  As such, they are bidirectional, and pullups are used to
// return them to the "high" state, whenever the drivers are set to the
// high impedance state.
//
// Pullups MUST BE USED on the ps2_clk and ps2_data lines for this design,
// whether they be internal to an FPGA I/O pad, or externally placed.
// If internal pullups are used, they may be fairly weak, causing bounces
// due to crosstalk, etc.  There is a "debounce timer" implemented in order
// to eliminate erroneous state transitions which would occur based on bounce.
// Parameters are provided to configure the debounce timer for different
// clock frequencies.  2 or 3 microseconds of debounce should be plenty.
// You may possibly use much less, if your pullups are strong.
//
// A parameters is provided to configure a 60 microsecond period used while
// transmitting to the mouse.  The 60 microsecond period is guaranteed to be
// more than one period of the ps2_clk signal.
//
//
//-------------------------------------------------------------------------------------


`resetall
`timescale 1ns/100ps

`define TOTAL_BITS   33  // Number of bits in one full packet


module ps2_mouse_interface (
  clk,
  reset,
  ps2_clk,
  ps2_data,
  left_button,
  right_button,
  x_increment,
  y_increment,
  data_ready,       // rx_read_o
  read,             // rx_read_ack_i
  error_no_ack
  );

// Parameters
```

```
        // The timer value can be up to (2^bits) inclusive.
        parameter WATCHDOG_TIMER_VALUE_PP = 19660; // Number of sys_clks for
400usec.
        parameter WATCHDOG_TIMER_BITS_PP  = 15;    // Number of bits needed for timer
        parameter DEBOUNCE_TIMER_VALUE_PP = 186;   // Number of sys_clks for debounce
        parameter DEBOUNCE_TIMER_BITS_PP  = 8;     // Number of bits needed for timer


        // State encodings, provided as parameters
        // for flexibility to the one instantiating the module.
        // In general, the default values need not be changed.

        // There are three state machines: m1, m2 and m3.
        // States chosen as "default" states upon power-up and configuration:
        //    "m1_clk_h"
        //    "m2_wait"
        //    "m3_data_ready_ack"

        parameter m1_clk_h = 0;
        parameter m1_falling_edge = 1;
        parameter m1_falling_wait = 3;
        parameter m1_clk_l = 2;
        parameter m1_rising_edge = 6;
        parameter m1_rising_wait = 4;

        parameter m2_reset = 14;
        parameter m2_wait = 0;
        parameter m2_gather = 1;
        parameter m2_verify = 3;
        parameter m2_use = 2;
        parameter m2_hold_clk_l = 6;
        parameter m2_data_low_1 = 4;
        parameter m2_data_high_1 = 5;
        parameter m2_data_low_2 = 7;
        parameter m2_data_high_2 = 8;
        parameter m2_data_low_3 = 9;
        parameter m2_data_high_3 = 11;
        parameter m2_error_no_ack = 15;
        parameter m2_await_response = 10;

        parameter m3_data_ready = 1;
        parameter m3_data_ready_ack = 0;
```

```
// I/O declarations
input clk;
input reset;
inout ps2_clk;
inout ps2_data;
output left_button;
output right_button;
output [8:0] x_increment;
output [8:0] y_increment;
output data_ready;
input read;
output error_no_ack;

reg left_button;
reg right_button;
reg [8:0] x_increment;
reg [8:0] y_increment;
reg data_ready;
reg error_no_ack;

// Internal signal declarations
wire watchdog_timer_done;
wire debounce_timer_done;
wire packet_good;

reg [`TOTAL_BITS-1:0] q;  // Shift register
reg [2:0] m1_state;
reg [2:0] m1_next_state;
reg [3:0] m2_state;
reg [3:0] m2_next_state;
reg m3_state;
reg m3_next_state;
reg [5:0] bit_count;      // Bit counter
reg [WATCHDOG_TIMER_BITS_PP-1:0] watchdog_timer_count;
reg [DEBOUNCE_TIMER_BITS_PP-1:0] debounce_timer_count;
reg ps2_clk_hi_z;     // Without keyboard, high Z equals 1 due to pullups.
reg ps2_data_hi_z;    // Without keyboard, high Z equals 1 due to pullups.
reg clean_clk;        // Debounced output from m1, follows ps2_clk.
reg rising_edge;      // Output from m1 state machine.
reg falling_edge;     // Output from m1 state machine.
reg output_strobe;    // Latches data data into the output registers
```

```
//----------------------------------------------------------------------
// Module code

assign ps2_clk = ps2_clk_hi_z?1'bZ:1'b0;
assign ps2_data = ps2_data_hi_z?1'bZ:1'b0;

// State register
always @(posedge clk)
begin : m1_state_register
  if (reset) m1_state <= m1_clk_h;
  else m1_state <= m1_next_state;
end

// State transition logic
always @(m1_state
         or ps2_clk
         or debounce_timer_done
         or watchdog_timer_done
         )
begin : m1_state_logic

  // Output signals default to this value, unless changed in a state condition.
  clean_clk <= 0;
  rising_edge <= 0;
  falling_edge <= 0;

  case (m1_state)
    m1_clk_h :
      begin
        clean_clk <= 1;
        if (~ps2_clk) m1_next_state <= m1_falling_edge;
        else m1_next_state <= m1_clk_h;
      end

    m1_falling_edge :
      begin
        falling_edge <= 1;
        m1_next_state <= m1_falling_wait;
      end

    m1_falling_wait :
      begin
        if (debounce_timer_done) m1_next_state <= m1_clk_l;
        else m1_next_state <= m1_falling_wait;
      end

    m1_clk_l :
      begin
        if (ps2_clk) m1_next_state <= m1_rising_edge;
        else m1_next_state <= m1_clk_l;
      end

    m1_rising_edge :
      begin
        rising_edge <= 1;
        m1_next_state <= m1_rising_wait;
      end

    m1_rising_wait :
      begin
        clean_clk <= 1;
        if (debounce_timer_done) m1_next_state <= m1_clk_h;
        else m1_next_state <= m1_rising_wait;
      end
    default : m1_next_state <= m1_clk_h;
  endcase
end


// State register
always @(posedge clk)
begin : m2_state_register
  if (reset) m2_state <= m2_reset;
  else m2_state <= m2_next_state;
end

// State transition logic
always @(m2_state
         or q
         or falling_edge
         or rising_edge
         or watchdog_timer_done
         or bit_count
         or packet_good
         or ps2_data
         or clean_clk
```

```
     )
begin : m2_state_logic

  // Output signals default to this value, unless changed in a state condition.
  ps2_clk_hi_z <= 1;
  ps2_data_hi_z <= 1;
  error_no_ack <= 0;
  output_strobe <= 0;

  case (m2_state)

    m2_reset :    // After reset, sends command to mouse.
      begin
        m2_next_state <= m2_hold_clk_l;
      end

    m2_wait :
      begin
        if (falling_edge) m2_next_state <= m2_gather;
        else m2_next_state <= m2_wait;
      end

    m2_gather :
      begin
        if (watchdog_timer_done && (bit_count == `TOTAL_BITS))
          m2_next_state <= m2_verify;
        else if (watchdog_timer_done && (bit_count < `TOTAL_BITS))
          m2_next_state <= m2_hold_clk_l;
        else m2_next_state <= m2_gather;
      end

    m2_verify :
      begin
        if (packet_good) m2_next_state <= m2_use;
        else m2_next_state <= m2_wait;
      end

    m2_use :
      begin
        output_strobe <= 1;
        m2_next_state <= m2_wait;
      end
```

```
// The following sequence of 9 states is designed to transmit the
// "enable streaming mode" command to the mouse, and then await the
// response from the mouse.  Upon completion of this operation, the
// receive shift register contains 22 bits of data which are "invalid"
// therefore, the m2_verify state will fail to validate the data, and
// control will be passed into the m2_wait state once again (but the
// mouse will then be enabled, and valid data packets will ensue whenever
// there is activity on the mouse.)
m2_hold_clk_l :
  begin
    ps2_clk_hi_z <= 0;   // This starts the watchdog timer!
    if (watchdog_timer_done && ~clean_clk) m2_next_state <= m2_data_low_1;
    else m2_next_state <= m2_hold_clk_l;
  end

m2_data_low_1 :
  begin
    ps2_data_hi_z <= 0;  // Forms start bit, d[0] and d[1]
    if (rising_edge && (bit_count == 3))
      m2_next_state <= m2_data_high_1;
    else m2_next_state <= m2_data_low_1;
  end

m2_data_high_1 :
  begin
    ps2_data_hi_z <= 1;  // Forms d[2]
    if (rising_edge && (bit_count == 4))
      m2_next_state <= m2_data_low_2;
    else m2_next_state <= m2_data_high_1;
  end

m2_data_low_2 :
  begin
    ps2_data_hi_z <= 0;  // Forms d[3]
    if (rising_edge && (bit_count == 5))
      m2_next_state <= m2_data_high_2;
    else m2_next_state <= m2_data_low_2;
  end

m2_data_high_2 :
  begin
    ps2_data_hi_z <= 1;  // Forms d[4],d[5],d[6],d[7]
    if (rising_edge && (bit_count == 9))
```

```
      m2_next_state <= m2_data_low_3;
    else m2_next_state <= m2_data_high_2;
  end

m2_data_low_3 :
  begin
    ps2_data_hi_z <= 0;  // Forms parity bit
    if (rising_edge) m2_next_state <= m2_data_high_3;
    else m2_next_state <= m2_data_low_3;
  end

m2_data_high_3 :
  begin
    ps2_data_hi_z <= 1;  // Allow mouse to pull low (ack pulse)
    if (falling_edge && ps2_data) m2_next_state <= m2_error_no_ack;
    else if (falling_edge && ~ps2_data)
      m2_next_state <= m2_await_response;
    else m2_next_state <= m2_data_high_3;
  end

m2_error_no_ack :
  begin
    error_no_ack <= 1;
    m2_next_state <= m2_error_no_ack;
  end

// In order to "cleanly" exit the setting of the mouse into "streaming"
// data mode, the state machine should wait for a long enough time to
// ensure the FA response is done being sent by the mouse.  Unfortunately,
// this is tough to figure out, since the watchdog timeout might be longer
// or shorter depending upon the user.  If the watchdog timeout is set to
// a small enough value (less than about 560 usec?) then the bit_count
// will get reset to zero by the watchdog before the FA response is
// received.  In that case, bit_count will be 11.
// If the bit_count is not reset by the watchdog, then the
// total bit_count will be 22.
// In either case, when this state is reached, the watchdog timer is still
// running and it is best to let it expire before returning to normal
// operation.  One easy way to do this is to check for the bit_count to
// reach 22 (which it will always do when receiving a normal packet) and
// then jump to "verify" which will always fail for that time.
m2_await_response :
  begin
```

```
        if (bit_count == 22) m2_next_state <= m2_verify;
        else m2_next_state <= m2_await_response;
      end

    default : m2_next_state <= m2_wait;
  endcase
end



// State register
always @(posedge clk)
begin : m3_state_register
  if (reset) m3_state <= m3_data_ready_ack;
  else m3_state <= m3_next_state;
end

// State transition logic
always @(m3_state or output_strobe or read)
begin : m3_state_logic
  case (m3_state)
    m3_data_ready_ack:
          begin
            data_ready <= 1'b0;
            if (output_strobe) m3_next_state <= m3_data_ready;
            else m3_next_state <= m3_data_ready_ack;
          end
    m3_data_ready:
          begin
            data_ready <= 1'b1;
            if (read) m3_next_state <= m3_data_ready_ack;
            else m3_next_state <= m3_data_ready;
          end
    default : m3_next_state <= m3_data_ready_ack;
  endcase
end

// This is the bit counter
always @(posedge clk)
begin
  if (reset) bit_count <= 0;  // normal reset
  else if (falling_edge) bit_count <= bit_count + 1;
  else if (watchdog_timer_done) bit_count <= 0;  // rx watchdog timer reset
```

```
        end

        // This is the shift register
        always @(posedge clk)
        begin
          if (reset) q <= 0;
          else if (falling_edge) q <= {ps2_data,q[`TOTAL_BITS-1:1]};
        end

        // This is the watchdog timer counter
        // The watchdog timer is always "enabled" to operate.
        always @(posedge clk)
        begin
          if (reset || rising_edge || falling_edge) watchdog_timer_count <= 0;
          else if (~watchdog_timer_done)
            watchdog_timer_count <= watchdog_timer_count + 1;
        end
        assign watchdog_timer_done =
(watchdog_timer_count==WATCHDOG_TIMER_VALUE_PP-1);

        // This is the debounce timer counter
        always @(posedge clk)
        begin
          if (reset || falling_edge || rising_edge) debounce_timer_count <= 0;
        //  else if (~debounce_timer_done)
          else debounce_timer_count <= debounce_timer_count + 1;
        end
        assign debounce_timer_done =
(debounce_timer_count==DEBOUNCE_TIMER_VALUE_PP-1);

        // This is the logic to verify that a received data packet is "valid"
        // or good.
        assign packet_good = (
                           (q[0]  == 0)
                        && (q[10] == 1)
                        && (q[11] == 0)
                        && (q[21] == 1)
                        && (q[22] == 0)
                        && (q[32] == 1)
                        && (q[9]  == ~^q[8:1])    // odd parity bit
                        && (q[20] == ~^q[19:12])  // odd parity bit
                        && (q[31] == ~^q[30:23])  // odd parity bit
                        );
```

```
// Output the special scan code flags, the scan code and the ascii
always @(posedge clk)
begin
  if (reset)
  begin
    left_button <= 0;
    right_button <= 0;
    x_increment <= 0;
    y_increment <= 0;
  end
  else if (output_strobe)
  begin
    left_button <= q[1];
    right_button <= q[2];
    x_increment <= {q[5],q[19:12]};
    y_increment <= {q[6],q[30:23]};
  end
end


endmodule

//`undefine TOTAL_BITS

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

//
```

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

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

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

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

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

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

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

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

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

   // synchronize with system clock

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

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

   // edge detection on write enable signal

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

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

   reg [31:0] mydata;
   always @(posedge clk)
     if (we_edge)
       mydata <= { mydata[17:0], data[1] };

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


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

```
//		To fix, delay col & row by 4 clock cycles.
//		Delay other signals as well.

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

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

// compute address to store data in
wire [8:0] y_addr = y_delay[38:30];
	wire [9:0] x_addr = x_delay[39:30];

wire [18:0] myaddr = {y_addr[8:0], eo_delay[3], x_addr[9:1]};

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


// alternate (256x192) image data and address
wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};
wire [18:0] myaddr2 = {1'b0, y_addr[8:0], eo_delay[3], x_addr[7:0]};

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

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

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

endmodule // ntsc_to_zbt
```

```
//
// 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



/*always @ (posedge clk or posedge rst)
  if (rst)
     begin
      R_int <= 0; G_int <= 0; B_int <= 0;
     end
  else
    begin
    X_int <= (const1 * (Y_reg - 'd64)) ;
    R_int <= X_int + (const2 * (Cr_reg - 'd512));
    G_int <= X_int - (const3 * (Cr_reg - 'd512)) - (const4 * (Cb_reg - 'd512));
    B_int <= X_int + (const5 * (Cb_reg - 'd512));
    end

*/
/* limit output to 0 - 4095, <0 equals o and >4095 equals 4095 */
```

```
assign R = (R_int[20]) ? 0 : (R_int[19:18] == 2'b0) ? R_int[17:10] : 8'b11111111;
assign G = (G_int[20]) ? 0 : (G_int[19:18] == 2'b0) ? G_int[17:10] : 8'b11111111;
assign B = (B_int[20]) ? 0 : (B_int[19:18] == 2'b0) ? B_int[17:10] : 8'b11111111;

endmodule


//
// 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



///////////////////////////////////////////
// Asynchronous UART Transmitter
```

```
//
////////////////////////////////////////////
module transmitter(
        input clk,
        input TxD_start,
        input [7:0] TxD_data,
        output TxD,
        output TxD_busy
);

// Assert TxD_start for (at least) one clock cycle to start transmission of TxD_data
// TxD_data is latched so it doesn't have to stay valid while it is being sent

parameter ClkFrequency = 25000000;          // 25MHz
parameter Baud = 9600;

reg [3:0] TxD_state = 0;
wire TxD_ready = (TxD_state==0);
assign TxD_busy = ~TxD_ready;

wire tick;
baudGen #(ClkFrequency, Baud) tickgen(.clk(clk), .enable(TxD_busy), .tick(tick));

reg [7:0] TxD_shift = 0;
always @(posedge clk)
begin
        if(TxD_ready & TxD_start)
                TxD_shift <= TxD_data;
        else
        if(TxD_state[3] & tick)
                TxD_shift <= (TxD_shift >> 1);

        case(TxD_state)
                4'b0000: if(TxD_start) TxD_state <= 4'b0100;
                4'b0100: if(tick) TxD_state <= 4'b1000;  // start bit
                4'b1000: if(tick) TxD_state <= 4'b1001;  // bit 0
                4'b1001: if(tick) TxD_state <= 4'b1010;  // bit 1
                4'b1010: if(tick) TxD_state <= 4'b1011;  // bit 2
                4'b1011: if(tick) TxD_state <= 4'b1100;  // bit 3
                4'b1100: if(tick) TxD_state <= 4'b1101;  // bit 4
                4'b1101: if(tick) TxD_state <= 4'b1110;  // bit 5
                4'b1110: if(tick) TxD_state <= 4'b1111;  // bit 6
                4'b1111: if(tick) TxD_state <= 4'b0010;  // bit 7
```

```
                    4'b0010: if(tick) TxD_state <= 4'b0011;  // stop1
                    4'b0011: if(tick) TxD_state <= 4'b0000;  // stop2
                    default: if(tick) TxD_state <= 4'b0000;
                endcase
        end

        assign TxD = (TxD_state<4) | (TxD_state[3] & TxD_shift[0]);  // put together the start,
data and stop bits
        endmodule
```