Carlos Cuevas & Cody Winkleblack
6.111 Final Report
14 December 2018

Laser Cyclops

Final Report

Contents

Abstract

The main objective of this project was to have a laser fire at a target based on the feed of an Infrared-filtered NTSC camera. This feed detected the position of a light-emitting diode (LED), the coordinates of which were determined using a field-programmable gate array FPGA. From here, these coordinates were translated into the two offset angles needed by two servos oriented along orthogonal axes. Pulses were sent, the duty cycles of which were attenuated such that the servos achieved the angular offset needed to target the LED. Finally, once the LED's location was found, the laser fired.

Objectives

Expanding on the abstract, all the objects for the final projects were split into three different levels of complexity, listed as basic, intermediate, and advanced. We expected to at least hit all the targets on the basic level and most if not all of the intermediate. This will be elaborated in their respective sections below.

Overview

This section explains how a signal from a camera is converted to servo pulses for the laser to fire at the target.

The NTSC camera is directly connected to the Labkit. This camera sends pixel data in the format YCrCb, which then gets translated to the format RGB. This 30-bit wide pixel is cut down to 18-bits, in order to best use the ZBT memory; two pixels are stored in every memory address.

At this point, there is a major break in data flow. The first takes the 18-bit wide RGB signal and calculates the position of the target by taking all values found in one frame and comparing them with the known RGB value of the LED. At first, a simple comparison will be used on the infrared LED; however, once chroma keying is enabled, there is an extra conversion for all RGB values to its hue for this purpose. Additionally, the chroma key is selected by the user and the intersection of the crosshairs.

The target coordinates are then transposed so that origin is at the center of the frame, as opposed to its original location at the top-left. From there, the new coordinates are converted to angles [radians], using the arctan function and the known distance from the laser to the target.

These angles are finally fed to an interpreter that determines the length of the pulse to send to each servo, where the neutral position of $\pi/2$ equates to a pulse length of 1.5 ms. This pulse will have a range ≈ [1, 2] ms, which equates to the domain $\boldsymbol{\theta} = [0, \pi]$. The servos expect a refreshed pulse every 20 ms. The attached laser will be on when locked onto the target (i.e. servos are at rest and target is found by the FPGA). A siren also plays, when above conditions are met, the infamous laser soundbite from *Galaga*.

The other branch in the aforementioned break sends the RGB value to ZBT memory, which is stored in sets of two pixels per address. This is later extracted by another module that is responsible for pixel assignment that is sent to the VGA monitor. This module assigns said pixel to be the following, in order of highest priority: crosshairs, target image, and pixel data from the camera. In other words, if said pixel is in the range of the crosshairs, it will be assigned the designated color of the crosshairs; if it's within the box of the image, it will be assigned the respective value within that image; else, it will simply be given the pixel from the camera. Because of an image overlay on the target, the target's location must also be fed into this module from above.

Basic Goals

Given the overview of the project, the lowest level of difficulty includes meeting the following goals, as extracted from our "Project Checklist":

- Basic implementation of the laser tracking system functioning properly.
  - All hardware components operational, with the target being followed by the laser, and subsequent laser firing as soon as it determines the target location.
- The VGA should display the feed from the camera, so that the user is able to visualize the movement of the target

The original modules also included in the document were: interpreter, zbtMemory, displayManager, ledFinder, arctan, servo, servoClock. However, our actual implementation has substituted some of the names above. See the "Modules" section for more information, including module descriptions.

Intermediate Goals

The next level of difficulty contains the following objectives:

- Implement chroma keying, wherein the system would be able to lock onto a color hue that is specified by the user.
- An image will be imposed over the target and displayed on the VGA output (e.g. the Death Star or someone's face).
- Using a green screen, an additional background will be displayed behind the target. This would allow us to use various landscapes (e.g. outer space, a war zone, etc.).
- A firing sound should be generated every time the laser fires at the target. This should be implemented on the go using a siren and counters.

Advanced Goals

The highest level of difficulty contains the following:

- Add more sophisticated sounds into flash memory that will play on queue.

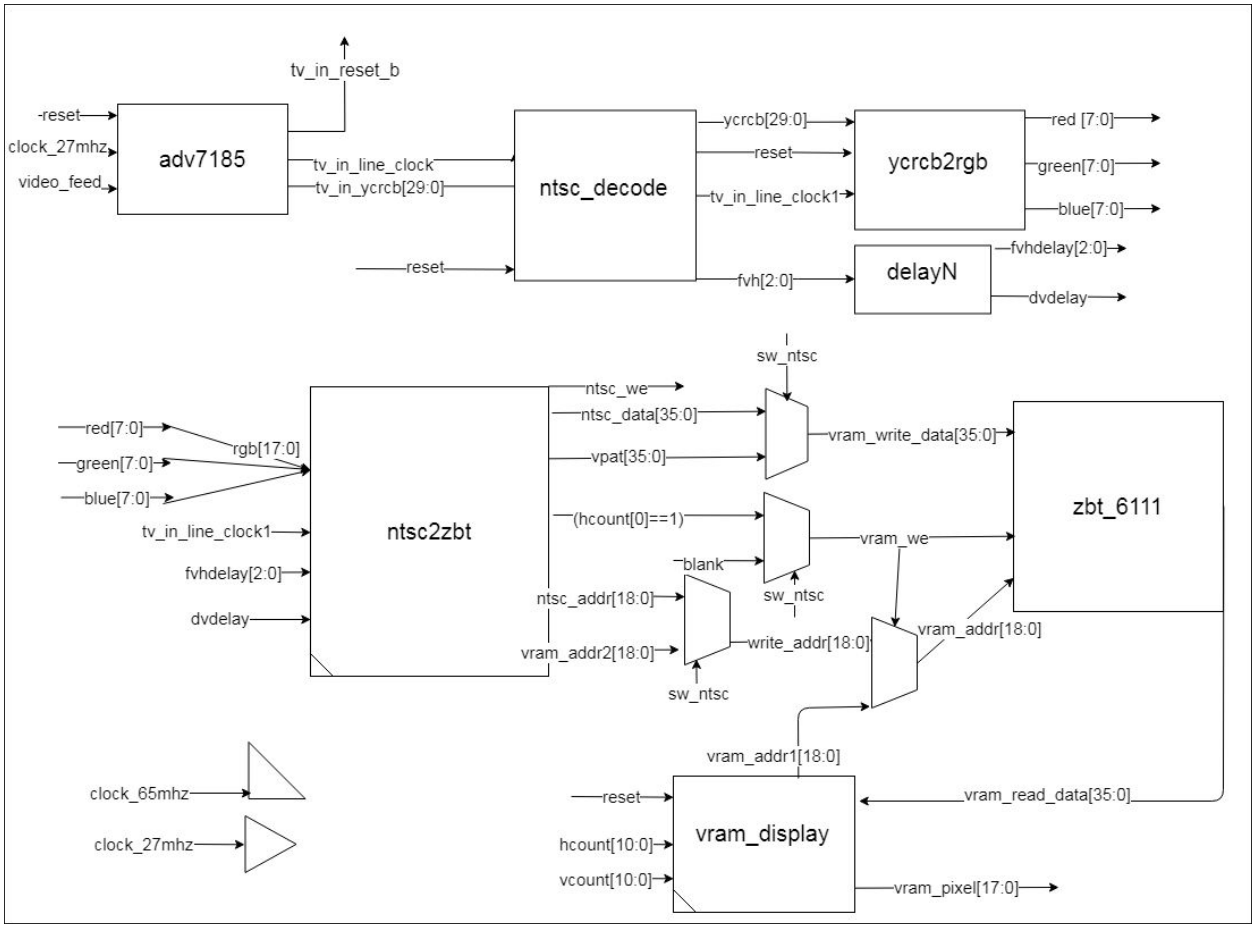
- A specific hue will be selected by the user using cross-hairs that are controlled via the Labkit buttons and displayed on the monitor.
    - From this, the distinct object that has the same hue as the cross-hair target will be targeted by the laser.

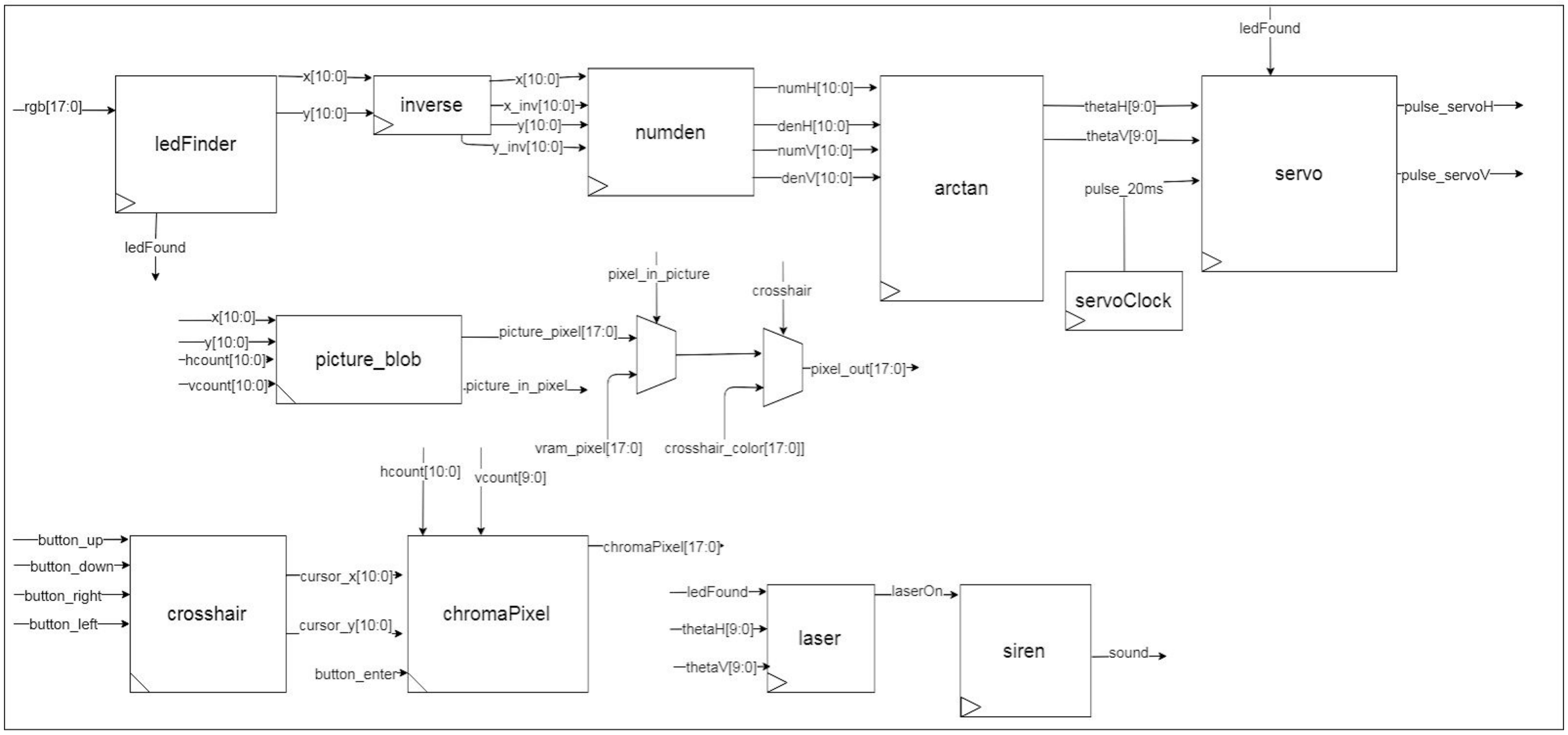
Hardware

The hardware that was used, with the exception of wires and tools, are listed below. Additionally, the method they were connected to the Labkit is listed as well, if applicable.

- Labkit (FPGA)
- 2 servos
  - 3 wires: 5V source, signal (angle pulse), ground
- Laser
  - 2 wires: 3.3V source & signal, ground, 2n7000
- Parts holding up the servo-laser mechanism
  - 3 pieces: one held the laser and was attached to the vertical servo, another attached both servos together, and the last secured the horizontal servo to the table
  - All pieces were 3-D printed using precise measurements on the hardware
- NTSC camera
  - 1 yellow composite video cable connected to the Labkit
  - 1 power cable connected to the 120V power outlet
- infrared LED
  - A small 3.3V battery and a button were soldered to the LED
- infrared filter
  - A floppy disk was used as the filter
- VGA monitor
  - A VGA cable was connected to the Labkit

Block Diagram





Modules

This section provides a list of all modules that were used in the latest implementation of the final project. The top level module is *laser_cyclops*; this is based off the *zbt_6111_sample* module written by Daniel Moon and Thipok Rak-Amnouykit in the Fall of 2014. A lot of features were used from their project since it entailed chroma keying.

The Labkit's included 27 MHz clock was used primarily. However, a 65 MHz clock was also generated when dealing with any modules that led to the VGA monitor. Additionally, the NTSC camera contained a separate clock of its own.

Before the detailed and alphabetized list of modules, this following list portrays the hierarchy of modules with direct calls coming from the parent module. An asterisk before the name of a module indicates a module not originally intended or found in the project proposal or checklist. These marked modules added complexity to our initial design and took much time to write.

Finally, inputs and outputs will be given in the format *MdecN*, where *M* represents the number of bits used in the integer component of a number and *N* is the number of bits used in the fractional part of the number. If not noted on a variable of bus width greater than one, assume *(bus width)dec0*.

- laser_cyclops
    - *adv7185init
        - *i2c
    - arctan
    - chromaPixel
    - crosshair
    - debounce
    - *delayN
    - *display_16hex
    - inverse
    - laser
    - ledFinder
    - *ntsc2zbt
    - *ntsc_decode
    - numden
    - picture_blob
        - lab3rom
        - lab3colormap
    - *ramclock
    - *rgb2hue
        - modulo
    - servo
        - servoClock

- siren
  - eighthNote
  - pitchMaker
- *video_decoder
- vram_display
- *xvga
- ycrcb2rgb
- *zbt_6111

arctan [Xilinx & Carlos]

- **Inputs**: clock_27mhz, [10'] num, [10'] den (2dec8; input range = [-1,1])
- **Outputs**: [10'] θ (in radians; signed 3dec7)
- **Objectives**: a pre-written Verilog module found in ISE, this module will take in the two shorter sides of the triangle wishing to compute an angle for (in radians). For this project, the inputs will be a coordinate (x or y) and depth (z). This will be given in the normalized form of *num* and *den*. The following equations demonstrate this:

  $$\theta = arctan(\frac{num}{den})$$

  The inputs are required to be on the unit circle and thus must reside within [-1,1]. Xilinx has a pre-defined module for this task called "CONIC;" selecting the "arctan" function will yield the above result..

chromaPixel [Cody]

- **Inputs**: clock_65mhz, clean_enter, [11’] hcount, [10’] vcount, [11’] x_cursor, [10’] y_cursor, reset, [18’] vram_pixel
- **Outputs**: [18’] chromaPixel
- **Objectives**: Given that the user hits *button_enter*, the x and y locations of the cursor are stored. *Vram_display* is taken to be a continuous stream of pixels, and once *hcount* and *vcount* are the same as the x and y location of the cross hairs, the module outputs the pixel value at this location.

crosshair [Cody]

- **Inputs**: clock_65mhz, reset, clean_up, clean_down, clean_left, clean_right
- **Outputs**: [11’] cursor_x, [10’] cursor_y
- **Objectives**: Allows the user to change the location of the crosshairs for selection of the chromaPixel. In the top level module, if *hcount* and *vcount* fall within a +/- range of 5 to this location, the pixel is replaced by a green pixel.

debounce [6.111 & Cody]

- **Inputs**: clock_65mhz, reset, button (up, left, down, right, enter, 3, 2, 1, 0)
- **Outputs**: clean
- **Objectives**: In several instances of this module, this takes all buttons found on the Labkit and debounces them, returning a "clean" version of the input.

delayN [6.111 & Cody]

- **Inputs:** clk, in
- **Outputs:** out
- **Objectives:** By setting the appropriate parameter N, this modules delays the signal *in* by N clock cycles by storing the value into a series of registers.

display_16hex [6.111 & Carlos]

- **Inputs**: clock_65mhz, reset, [64'] dispdata
- **Outputs**: disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b, disp_data_out
- **Objectives**: primarily used for debugging, this module portrays critical signals to the hex display found on the Labkit. Depending on the test being performed, signals such as thetas, coordinates, or any other important value were displayed. However, these signals had to undergo a transformation from binary to decimal format so that the numbers portrayed would be easily readable.

inverse [Xilinx & Carlos]

- **Inputs**: clock_27mhz, [10'] n (10dec0)
- **Outputs**: [10'] n_inv (0dec10)
- **Objectives**: this module will prepare the input value for arctan by taking its reciprocal and outputting that. The input is 10 bits representing a whole number. The output is 10 bits representing a decimal with 0 bits used for the integer component. Negative signs are not saved and thus all I/O should be unsigned.
  ISE uses the Divider Module for this task. Use the following I/O interface:
    Inputs: clk (27MHz), [2'] dividend = 10'b1, [10'] divisor (10dec0)
    Outputs: rfd, [2'] quotient, [10'] fractional (0dec10)

laser [Carlos]

- **Inputs**: clock_27mhz, ledFound, [9'] θ_H, [9'] θ_V (2dec7)
- **Outputs**: laserOn

- **Objectives**: this module determines when the laser should be firing. Ideally, this should happen when the following conditional is met:
  ```
  if(ledFound &&
          abs(ΘH[n-1] - ΘH[n])<=E &&
          abs(ΘV[n-1] - ΘV[n])<=E)
      laserOn <= 1;
  else laserOn <= 0;
  ```
  The above conditional simply checks that (1) ledFound is high and (2) neither servo angle has changed (between the previous and current time steps) more than a specified error *E*.

ledFinder [Carlos]

- **Inputs**: clock_27mhz, reset, [18'] pixel_translated (RGB or hue), [11'] hcount, [10'] vcount, [9'] targetHue, mode
- **Outputs**: [11'] x, [11'] y, ledFound
- **Objectives**: this module is tasked with determining the center coordinates of the target at every frame, using the pixels that go through it. The input *mode* switches the computation between infrared LED and chroma keying. Pixel inputs still go through the same input *pixel_translated*; however, hues are only 9 bits wide. The hue that is being compared to is *targetHue*. The signal *ledFound* is high when at least one pixel in the frame is the designated infrared or chroma key value.

ntsc2zbt [6.111 & Cody]

- **Inputs**: clock_65mhz, [18'] rgb (2 pixels in format RGB), tv_in_line_clock1, [3'] fvhdelay
- **Outputs**: [19'] ntsc_addr, ntsc_we, [36'] ntsc_data, [36'] vpat, hcount[0]==1
- **Objectives**: This is responsible for sending a pixel to the VGA monitor for every (hcount, vcount) value. It extracts two pixels for every address request sent to the zbt_6111 module.

numden [Carlos]

- **Inputs**: clock, [10'] pos (x or y), [10'] pos_inv, [10'] z, [10'] z_inv
- **Outputs**: [10'] num, [10'] den (signed 2dec8)
- **Objectives**: this module will prepare the inputs for arctan by taking the reciprocal and original values for x, y, and z, and using them to normalize z/x or z/y. This will be done by using the following simplified conditionals:
  ```
  if(pos > z) begin
  num <= z * pos_inv;
  den <= 1;
  end
  ```

```
else if(z > pos) begin
num <= 1;
den <= pos * z_inv;
end

else ERROR;
```

The output format has to be in signed 2dec8 for the arctan module. This is simply done by taking the original result of num or den and appending two zeroes to the front of the decimal, except if that value is 1.

picture_blob [Cody]

- **Inputs**: clock_65mhz, [11’] x, [10’] y, [11’] hcount, [10’] vcount
- **Outputs**: [18’] picture_pixel, picture_in_pixel
- **Objectives**: This instantiation will overlay an image of the Death Star on top of the target location, given by x and y. *Picture_pixel* will be an input to the mux that determines what pixel (crosshair, death star, or camera, in that order of priority) gets sent to the display.

ramclock [6.111 & Cody]

- **Inputs:** ref_clock, clock_feedback_in
- **Outputs:** fpga_clock, ram0_clock, ram1_clock, clock_feedback_out, locked
- **Objectives:** This creates deskewed clocks specifically for the ZBT to use. It ensures that this clock is deskewed relative to all the other clocks sent to the other modules.

rgb2hue [Carlos]

- **Inputs**: clock_65mhz, [18’] rgb
- **Outputs**: [9’] hue
- **Objectives**: this module takes in a RGB value where each color contains the 6 MSB (the module internally appends 2 zeros so that the RGB value is once again at 24 bits). From this RGB value, the module returns its respective hue, from the HSL color format. A *modulo* and *inverse* is required for computation.

modulo [Carlos]

- **Inputs**: clock_65mhz, [9’] value, [3’] modulus
- **Outputs**: [9’] remainder
- **Objectives**: this module takes in a value and modulus and returns the remainder. It was specifically written to compute (value)mod6 but can be used with other modulis.

servo [Carlos]

- **Inputs**: clock_27mhz, [10'] theta (unsigned 2dec7), ledFound, pulse_20ms
- **Outputs**: pulse_servo
- **Objectives**: given the radian angle for a plane, this module will translate this angle to the appropriate pulse required by the servo. This pulse must fall roughly in the range [1,2] ms and be sent every 20 ms, which is what the input *pulse_20ms* is for. However, should *ledFound* be low (in which the target is not found), the module ignores the angle input and moves the servos to rest (π/2), using the neutral pulse value of 1.5 ms.

servoClock [Carlos]

- **Inputs**: clock_27mhz
- **Outputs**: pulse_20ms
- **Objectives**: a clock that uses the FPGA base clock and a counter in order to send out a pulse every 20 ms.

siren [Carlos & Cody]

- **Inputs**: clock_27mhz, sirenOn(laserOn), complexMelody
- **Outputs**: sound
- **Objectives**: This module is taken from Carlos's lab 4; upon *sirenOn* begin high, it turns plays the melody assigned by *complexMelody*. If this signal is low, it will simply play alternate a pitch and silence; if high, it will play a *Galaga* sounding laser fire.

ntsc_decoder [Cody]

- **Inputs:** clk, reset, [10'] tv_in_ycrcb
- **Outputs:** [30'] ycrcb, f, v, h, data_valid
- **Objectives:** This module takes in the 10 bit ycrcb data from adv1785init and encodes it into a 30 bit representation of ycrcb. It also outputs 3 sync lines indicating whether the field is even or odd (f), a vertical (v) and horizontal sync (h).

vram_display [Cody]

- **Inputs:** reset, clk, [11'] hcount, [10'] vcount, [36']vram_read_data
- **Outputs:** [18'] vr_pixel, [19'] vram_addr
- **Objectives:** This module generates the appropriate display pixels by reading the values at the ZBT ram at the address given by *vram_addr.* Two pixels are stored in each zbt memory space. The output, *vr_pixel,* gets sent to a mux wherein its decided whether the pixel should be replaced first by a crosshair or death star pixel, before being sent out to the display.

xvga [6.111 & Cody]

- **Inputs:** vclock
- **Outputs:** [11’] hcount, [10’] vcount, vsync, hsync, blank
- **Objectives:** This module generates the XVGA display signals, used widely by many other modules to determine which pixel is being accessed or generated (via the *hcount* and *vcount,* and the syncs at the end of each line).

ycrcb2rgb [Carlos]

- **Inputs**: clock_27mhz, [30'] pixel_camera (coming from the NTSC camera in the format YCrCb)
- **Outputs**: [18'] pixel_translated (format RGB)
- **Objectives**: Taking the input provided by the NTSC camera, a pixel is given in the format YCbCr. This module must then translate this pixel into the format RGB. Since there will be 24 bits generated after this translation, this must be cut down to 18 bits using the most significant bits from each color value so that two pixel values can be inserted into each ZBT-memory address.

zbt_6111 [6.111 & Cody]

- **Inputs**: clock_65mhz, vram_we, [36’] vram_write_data, [19’] vram_addr
- **Outputs**: [36'] vram_read_data
- **Objectives**: This saves the pixel that is coming in from the the set of muxes and the *ntsc2zbt* module into an address on ZBT memory. Two pixels per address are saved. With a given address, as requested by the *vram_display* module, the respective two pixels in that address will be outputted.

Process

Isolated Testing

Our plan from the very start was to get coding right away since we knew that we would be spending a lot of time on debugging Verilog. Thus, most of our modules were actually written the first weekend right after the project checklist.

The display module, being more difficult, took longer until we found the aforementioned sample code from a previous year. Given that the display portion of our project did not need to wait for hardware, since it was only using the camera, the Labkit, and the monitor, debugging immediately began on that.

The display (from the *adv7185* to *vram_display* modules) worked accordingly, including all the conversions to rgb values for the pixels. We were able to display a color image on the VGA display coming from the NTSC camera. Furthermore, this confirms that the use of the ZBT memory to store the pixels of each frame was also working accordingly, as the data was read by the *vram_display* before being sent out to the display.

The *crosshair* module was tested and worked as expected, overlaying the crosshairs on the display and obtaining the pixel that the user chose. Furthermore, the blob module was overlaying the appropriate image when we hardcoded a coordinate for display (in a full implementation we would have overlayed this on top of the x and y coordinates obtained from the *ledFinder* module). In testing, there seemed to be a strange bug wherein the screen would break for a moment immediately after programming, only to be overcome by turning the entire Labkit on and off again.

Once our parts for the mount were printed, work began on testing the other half of the modules. Before even assembling the whole mechanism, each servo and the laser was tested. The laser worked immediately; however, it was at this point we realized that the specifications on the manufacturer's website for the servos were a little unclear and not exact. But after spending some time fine tuning the pulse durations, an accurate conversion was developed to take incoming angles into pulses. The problems in our project then started here. Once the documentation was further investigated, it became apparent that the *arctan* module included in ISE was very restrictive. It only took in values for the numerator and denominator within the range [-1, 1]. This meant that we had to take our position and depth values, and normalize them. This was not a huge obstacle but took time, particularly when trying to avoid division as much as possible; in the end, it was required to do so, and thus the *inverse* module was created. Given standard numbers as input, the servos still ended up doing as asked. Thus, we thought it ready to attempt the first joint integration.

Joint Testing

As soon as we joined all the modules together for the first time, still leaving some, like the *ledFinder* unused—and just hardwiring coordinate values is where it really became a problem. It was quickly realized that the coordinate system that the pixels were using up until this point couldn't be used with the servos since our given neutral position of π/2 would not have computed given a coordinate of width/2 or height/2. We needed to transpose the coordinate system so that the origin was at the center, thus giving a value of $\theta = arctan(\frac{z}{0}) = \pi/2$. It was at this point, albeit with spending hours and hours of debugging, that we couldn't get the system to work. It should be noted that this was the first time any of our modules had to deal with negative numbers, and thus signed variables. Although arctan was designed to take in signed values as input and relay them signed as output, we were never able to get the system to correctly portray a hardcoded coordinate, despite successfully using a fixed decimal point format throughout. We think that the main problem was how the code was translating the coordinates and keeping track of the negative sign.

And because many of our last days in lab was spent in trying to get the laser to aim at the target, work was not able to be completed beyond this point, despite having already written modules to use a chroma key, the crosshairs, and portray the Death Star image. It was really frustrating that the bottleneck in our project was due to mathematical complications and Verilog constraints on both their *arctan* module as well as signed numbers.

For the Future

Joe gave an interesting proposal for how to determine depth. Instead of hard coding in a depth ('z'), we could instead use the angular offset of two different lasers to determine it, much like humans naturally do. In hindsight this was an interesting proposal, and had we needed a system with variable depths we could have implemented this.

The IR LED that was used was far too weak to be imaged. With the IR filter in place, we could only see the IR light when the LED was a few inches away from the lens of the NTSC. Instead, we resorted to using the powerful LED on our cell phones, which could be seen from many feet away. The filtering technique would still be pivotal to this end; although, a different targeted light source would surely need to be used.

Since signal conversion became a huge problem that was not foreseen, it should be emphasized the importance of laying down signals prior to coding. Additionally, if another method is found to create the pulses on the servos without needing trigonometry or especially negative values, that method might prove to be far easier to implement as well as far easier in computation.

Verilog

This final section contains the appended Verilog code used in the final project. The page numbers on the code will not be consistent with the final report’s numbering and should be noted separately.

```
// Carlos Cuevas & Cody Winkleblack
// 6.111 Final Project
// Laser Cyclops
// 10 Dec 2018

/*
This is the TOP LEVEL module for the entire final project. This file
   is a heavily modified version of the original "zbt_6111_sample.v"
   The comment block for the original file is found at the very bottom.

In short, this project's ultimate goal is the following:
   1. interpret the pixel-position of a target using a NTSC camera
      (tracking either an infrared LED or a chromakeyed object)
      a. chromakey is selected using crosshairs and user button;
         this value is stored as a hue
      b. YCrCb values from camera need to be converted to RGB
   2. convert that pixel coordinate to a real-distance coordinate
   3. compute two angles, in radians, that would have two servos
      aim at said target
   4. translate said angles to pulses for the servos
   5. convey a picture to the monitor; said picture contains the
      following per pixel, in order of priority:
      i. crosshairs
      ii. target's image (e.g. Death Star)
         determined by interpreting incoming RGB pixels,
         translating to the hue, and comparing that to
         chromakey
      iii. pixel value from NTSC camera
   6. fire the laser and a sound from a siren, when the target is locked on
      a. siren will alternate between a tone (A) and silence to
         make it sound like a blaster from Star Wars
      b. "target is locked on" means that the servos are not moving
         (or with a very small margin of error) and a position
         for the target is known

The 64-bit hex-display shall be used to convey the following values, using the
hex-format
   (every value is a hex-digit, or 4 bits in length; 16 hex-values total = 64 bits):
   hex[15:12] = x
   hex[11:8] = y
   hex[7:4] = thetaH
   hex[3:0] = thetaV

The eight LEDs on the Labkit will portray the following signals (note that the LED's
take
   inverted signals in practice):
   led[7]
   led[6]
   led[5] = complexMelody
   led[4] = ~mode (ledFinder; mode(1) = infrared; mode(0) = chromaKey)
   led[3] = laserOn
   led[2] = ledFound
   led[1] = reset
   led[0] = switch[0]
   assign led = ~{4'b0, laserOn, ledFound, reset, switch[0]};

The switches on the labkit serve to debug modules.  However, if button0 is active,
   then all switches become theta inputs to servos. Otherwise, What they are wired to is listed below:
   switch[7]
   switch[6] = sw (input for ntsc_to_zbt)
   switch[5] = ~complexMelody (sw(0) = complex)
   switch[4] = ~mode (for ledFound; sw(1) = infrared; sw(0) = chromaKey)
   switch[3]
   switch[2] = ~ledFound
   switch[1]
   switch[0] = display_debug (1: pixel <= {hcount[8:6],15'b0}; 0: (crosshairs --> image --> rgb_pixel))
      above switch selects between "red bar" and laser_cyclops modes
      "red bar" mode is as it sounds to check FPGA is not frozen

The following buttons are serving as user/degugging inputs (in practice also inverted):
   button_enter = user_reset
   button_up = crosshair UP
   button_down = crosshair DOWN
   button_left = crosshair LEFT
   button_right = crosshair RIGHT
   button3 = ~sw_ntsc
   button2
   button1 = grab chromaPixel on crosshair corrdinates
   button0 = manual override servo inputs to switches

The following provides an outline of all the modules being used (formatted as ISE provides):
   laser_cyclops
      adv7185init
         i2c
      chromaPixel
      crosshair
      debounce
      delayN
      display_16hex
      labkit.ucf
      laser
      ledFinder
      ntsc2zbt
      ntsc_decode
      ntsc_to_zbt
      picture_blob
         lab3rom
         lab3colormap
      ramclock
      rgb2hue
         inverse
      servo
      servoClock
      siren
         eighthNote
         pitchMaker
      video_decoder
      vram_display
      xvga
      ycrcb2rgb
      zbt_6111
```

```
*/

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

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

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

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

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

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

	      clock_feedback_out, clock_feedback_in,

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

	      rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

	      mouse_clock, mouse_data, keyboard_clock, keyboard_data,

	      clock_27mhz, clock1, clock2,

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

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

	      switch,

	      led,

	      user1, user2, user3, user4,

	      daughtercard,

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

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

```

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

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

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

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

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

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

    input  clock_feedback_in;
    output clock_feedback_out;

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

    output rs232_txd, rs232_rts;
    input  rs232_rxd, rs232_cts;

    input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

    input  clock_27mhz, clock1, clock2;

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

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

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

    inout [43:0] daughtercard;

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

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

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

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

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

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

    // SRAMs

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

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

```
      assign ram0_cen_b = 1'b0;  // clock enable
   */

   /* enable RAM pins */

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

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

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

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

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

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

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

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

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

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

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

      // Daughtercard Connectors
      assign daughtercard = 44'hZ;

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

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

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

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

      //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;

     // debounce all other buttons
     wire clean_left, clean_right, clean_up, clean_down, clean3, clean2, clean1, clean0;
     debounce db2(reset, clk, ~button_left, clean_left);
     debounce db3(reset, clk, ~button_right, clean_right);
     debounce db4(reset, clk, ~button_up, clean_up);
     debounce db5(reset, clk, ~button_down, clean_down);
     debounce db6(reset, clk, ~button3, clean3);
     debounce db7(reset, clk, ~button2, clean2);
     debounce db8(reset, clk, ~button1, clean1);
     debounce db9(reset, clk, ~button0, clean0);

     // display_16hex module for debugging
     // contents of dispdata on top of document
     reg [63:0] dispdata;
     display_16hex hexdisp1(reset, clk, dispdata,
             disp_blank, disp_clock, disp_rs, disp_ce_b,
             disp_reset_b, disp_data_out);

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

     // wire up to ZBT ram
     wire [35:0] vram_write_data;
     wire [35:0] vram_read_data;
     wire [18:0] vram_addr;
     wire        vram_we;
     //wire [35:0] chromaPixelData;
     //wire [18:0] chromaPixelAdress;
```

```

   //conversion of each pixel into their hue, to be fed into ledFinder
   wire [8:0] vr_hue; //checking each pixel against the chromaHue
   wire [17:0] chromaPixel;
   wire [8:0] chromaHue;
   wire [17:0] vr_pixel;
   rgb2hue rgb2hue1(.clock(clk),.rgb(vr_pixel),.hue(vr_hue));
   // height and width for video feed from NTSC
   parameter VIDEO_WIDTH = 11'd640;
   parameter VIDEO_HEIGHT = 11'd480;
   // hard-wired coord's below should have servos aim at thetaH = (pi/2, 3pi/4), thetaV = (pi/4, pi/2)
   wire [10:0] x = 11'd640; // x-coordinate of target; range = [0, 639]
   wire [10:0] y = 11'd0; // y-coordinate of target; range = [0, 479]
   wire [10:0] x_debug, y_debug; // debug
   wire mode = ~switch[4]; // 0 = chromaKey mode; 1 = infrared mode (ledFinder)
   // wire ledFound;
   wire debug_servo = clean0;
   wire ledFound = debug_servo ? 1 : ~switch[2]; // debugging
   wire fakeLed; //just used as initial output for debugging, modify below in call to ledFinder
   ledFinder ld(.clock(clock_27mhz),.reset(reset),.pixel_translated(vr_hue),
      .hcount(hcount),.vcount(vcount),.x(x_debug),.y(y_debug),
      .ledFound(fakeLed),.targetHue(chromaHue), .mode(mode)); //pixel translated is fed by vram_display

   // debug & display_16hex
   wire [15:0] disp_x, disp_y;
   bin2decInt dispX (.clock(clock_27mhz), .value(x), .out(disp_x));
   bin2decInt dispY (.clock(clock_27mhz), .value(y), .out(disp_y));

   // transpose x,y so that theta = pi/2 @ neutral position
   wire [10:0] x_trans, y_trans; // 11dec0
   wire x_neg, y_neg; // x|y is negative if asserted
   transpose transposeXY (.clock(clock_27mhz), .x(x), .y(y), .width(VIDEO_WIDTH), .height(VIDEO_HEIGHT),
      .x_out(x_trans), .y_out(y_trans), .x_neg(x_neg), .y_neg(y_neg));

   // invert x,y (already transposed), with negative sign nullated in this step
   wire [10:0] x_inv, y_inv; // 0dec11
   inverse xInv (.clk(clock_27mhz), .dividend(2'b1), .divisor(x_trans), .fractional(x_inv));
   inverse yInv (.clk(clock_27mhz), .dividend(2'b1), .divisor(y_trans), .fractional(y_inv));

   // generate numerator and denomenator, where theta = arctan(num/den)
   wire [10:0] numH, numV, denH, denV; // 2dec9
   parameter Z_COORD = 11'd640; // width of camera spans ~ 8 floor_tiles; z spans ~ 8 floor_tiles; thus, equal to VIDEO_WIDTH = 640
   parameter Z_INV = 11'b0000_0000_011; // 0dec11; z_inv = 1/640 = 0.0015625
   numden numdenX (.clock(clock_27mhz), .a(x_trans), .a_inv(x_inv), .b(Z_COORD), .b_inv(Z_INV), .num(numH), .den(denH));
   numden numdenY (.clock(clock_27mhz), .a(y_trans), .a_inv(y_inv), .b(Z_COORD), .b_inv(Z_INV), .num(numV), .den(denV));

   // den should now incorporate the negative sign
   wire [10:0] denH_neg, denV_neg; // 2dec9; signed
```

```verilog
   assign denH_neg = x_neg ? {(~denH[10:9])+1, denH[8:0]} : denH;
   assign denV_neg = y_neg ? {(~denV[10:9])+1, denV[8:0]} : denV;

   // arctan function (theta = arctan(num/den); num/den = y_in/x_in)
   wire signed [10:0] thetaH_exp, thetaV_exp; // 3dec8
   arctan arctanX (.clk(clock_27mhz), .x_in(0), .y_in(1), .phase_out(thetaH_exp));
   arctan arctanY (.clk(clock_27mhz), .x_in(0), .y_in(1), .phase_out(thetaY_exp));
   // transpose all angles by adding pi/2
   wire signed [10:0] thetaH_trans = thetaH_exp + 11'b001_1001_0010; // 3dec8; add pi/2
   wire signed [10:0] thetaV_trans = thetaV_exp + 11'b001_1001_0010;

   // debug & display_16hex
   wire [15:0] disp_thetaH, disp_thetaV;
   reg [8:0] thetaH, thetaV;
   bin2decAngle dispThetaH (.clock(clock_27mhz), .neutral(~ledFound), .value(thetaH),
.hex(disp_thetaH));
   bin2decAngle dispThetaV (.clock(clock_27mhz), .neutral(~ledFound), .value(thetaV),
.hex(disp_thetaV));

   // servo functions
   wire [7:0] debug_servo_angle = switch;
   always @(posedge clock_27mhz) begin
      if(debug_servo) begin // theta in format 2dec7
         case(debug_servo_angle[7:4])
            4'd0: thetaH <= 9'b0; // 0
            4'd1: thetaH <= 9'b00_1000011; // pi/6
            4'd2: thetaH <= 9'b00_1100100; // pi/4
            4'd3: thetaH <= 9'b01_0000110; // pi/3
            4'd4: thetaH <= 9'b01_1001001; // pi/2
            4'd5: thetaH <= 9'b10_0001100;
            4'd6: thetaH <= 9'b10_0101101;
            4'd7: thetaH <= 9'b10_1001110;
            default: thetaH <= 9'b11_0010010;
         endcase // thetaH

         case(debug_servo_angle[3:0])
            4'd0: thetaV <= 9'b0; // 2dec7
            4'd1: thetaV <= 9'b00_1000011;
            4'd2: thetaV <= 9'b00_1100100;
            4'd3: thetaV <= 9'b01_0000110;
            4'd4: thetaV <= 9'b01_1001001;
            4'd5: thetaV <= 9'b10_0001100;
            4'd6: thetaV <= 9'b10_0101101;
            4'd7: thetaV <= 9'b10_1001110;
            default: thetaV <= 9'b11_0010010;
         endcase // thetaV
      end

      else begin // ~debug
         thetaH <= thetaH_trans[9:1]; // 2dec7
         thetaV <= thetaV_trans[9:1];
      end
   end //always
   wire pulse_20ms, pulse_servoH, pulse_servoV;
   assign user1[1:0] = {pulse_servoH, pulse_servoV};
   servoClock clock_20ms (.clock(clock_27mhz), .pulse_20ms(pulse_20ms));
   servo servoH (.clock(clock_27mhz), .ledFound(ledFound), .pulse_20ms(pulse_20ms), .
theta(thetaH), .pulse_servo(pulse_servoH));
   servo servoV (.clock(clock_27mhz), .ledFound(ledFound), .pulse_20ms(pulse_20ms), .theta(thetaV), .pulse_servo(pulse_servoV));

   // laser module
   wire laserOn;
   assign user1[31] = laserOn;
   laser laserFire (.clock(clock_27mhz), .ledFound(ledFound), .thetaH(thetaH), .thetaV(thetaV), .laserOn(laserOn));

   // siren module
   wire sound;
   wire complexMelody = ~switch[5];
   assign user1[29] = sound;
   siren sr(.clock(clock_27mhz), .sirenOn(laserOn), .complexMelody(complexMelody), .sound(sound));

   //image overlay, based on locations of x and y from ledFinder, mux later based on ledFound, and if pixel is within range of image (width 256, height 240)
   wire [23:0] dstarPixel;
   picture_blob dstar1(.pixel_clk(tv_in_line_clock1),.x(11'd500),.hcount(hcount),.y(10'd400),.vcount(vcount),.pixel(dstarPixel)); //x and y here are the starting coordinates for the image display


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

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

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

   //NTSC Decode
   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));

   //conversion of YCrCb data to RGB data, rgb outputs are originally 8 bits wide, need to shorten that in a second
   wire [7:0] red;
   wire [7:0] green;
   wire [7:0] blue;
   YCrCb2RGB rgb(.R(red), .G(green), .B(blue), .clk(tv_in_line_clock1), .rst(reset), .Y(ycrcb[29:20]), .Cr(ycrcb[19:10]), .Cb(ycrcb[9:0]) );

   //6 bit r g and b
```

```verilog
   wire [5:0] red2 = red[7:2];
   wire [5:0] green2 = green[7:2];
   wire [5:0] blue2 = blue[7:2];

   // NTSC data written to ZBT memory
   //delay of fvh and delay to account for YCrCb Delay (3 clk cycles)
   wire fvhDelay0, fvhDelay1, fvhDelay2, dvDelay;
      delayN #(.NDELAY(3))
   fvh0(.clk(tv_in_line_clock1),.in(fvh[0]),.out(fvhDelay0));
      delayN #(.NDELAY(3))
   fvh1(.clk(tv_in_line_clock1),.in(fvh[1]),.out(fvhDelay1));
      delayN #(.NDELAY(3))
   fvh2(.clk(tv_in_line_clock1),.in(fvh[2]),.out(fvhDelay2));
      delayN #(.NDELAY(3))
   DvD(.clk(tv_in_line_clock1),.in(dv),.out(dvDelay));

   wire [18:0] ntsc_addr;
   wire [35:0] ntsc_data;
   wire        ntsc_we;

   ntsc_to_zbt n2z (.clk(clk), .vclk(tv_in_line_clock1), .fvh({fvhDelay2,fvhDelay1,fvhDelay0}), .dv(dvDelay), .din({red2,green2,blue2}),
          .ntsc_addr(ntsc_addr), .ntsc_data(ntsc_data), .ntsc_we(ntsc_we), .sw(switch[6]));

   // generate pixel value from reading ZBT memory
   // wire [17:0]    vr_pixel;   //each pixel requires 18 bits
   wire [18:0]    vram_addr1;
   //cursor additions for selecting hue to target on
   wire [10:0] cursor_x;
   wire [9:0] cursor_y;
   //wire [17:0] chromaPixelAdress;

   //crosshair instatiation
   parameter CROSS_MAX_X = 11'd700;
   parameter CROSS_MIN_X = 11'd12;
   parameter CROSS_MAX_Y = 10'd560;
   parameter CROSS_MIN_Y = 10'd75;
   crosshair #(.X_MAX(CROSS_MAX_X), .X_MIN(CROSS_MIN_X), .Y_MAX(CROSS_MAX_Y), .Y_MIN(CROSS_MIN_Y))
         cr (.reset(reset), .clk(clk), .up(clean_up), .down(clean_down), .left(clean_left), .right(clean_right), .cursor_x(cursor_x), .cursor_y(cursor_y)); //cursor location sent to chromakey

   //vram instatiation, sends vram_adress to ZBT to access vram_pixel to be displayed, and vram_pixel value to chromaPixel when hcount and vcount match the cursor location
   vram_display vd1(reset,clk,hcount,vcount,vr_pixel,vram_addr1,vram_read_data);

   //SETUP servos and all the associated modules
   //laser instatiation, sets pin 23 to high to pulse the laser
   // wire laserOn;
   // assign user1[23] = laserOn;
   //laser lase(.clock(clk), .ledFound(ledFound),.thetaH(),.thetaV(),.laserOn(laserOn));

   //chromaPixel grabs the value of the pixel found at the center of the cross hair,
translated into hue
   chromaPixel chroma(.button_enter(clean1),.hcount(hcount),.vcount(vcount),.x_cursor(cursor_x),.y_cursor(cursor_y),.clk(clk),.reset(reset),.vram_pixel(vr_pixel),.chromaPixel(chromaPixel));
   rgb2hue chromhue(.clock(clk),.rgb(chromaPixel),.hue(chromaHue));


   // 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];
   //vpat = 36'd0; //if outside of range of ntsc, pixels are black

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

   wire [35:0]    vpat = 36'd0; //all black instead of videopattern


   // mux selecting read/write to memory based on which write-enable is chosen
   wire  sw_ntsc = clean3;
   //wire   my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank; //my_we should be high for 2 clock cycles
   wire  my_we = sw_ntsc ? (hcount[0]==1'd1) : blank; //write enable should be high every other clock cylce
   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 using initial mux, decides whether pixel or
   //will be expanded to account for chromakeying, wherein if the pixel matches the chroma key, an image is overlayed

   reg [17:0] pixel;
   reg b,hs,vs, dstar;

   parameter HALFWIDTH = 178;     // default picture width
   parameter HALFHEIGHT = 120;
   parameter WIDTH = 356;
   parameter CROSS_WIDTH = 3'd3; // width of cursor boxes = CROSS_WIDTH * 2 - 1
   reg crosshair, xDepth, yDepth, xBorder, yBorder, Border; // determines pixel assignment for crosshairs

   wire display_debug = debug_servo ? 0 : switch[0];
   always @(posedge clk) begin
      //if ( ((((hcount+3) > cursor_x) && (hcount < (cursor_x+3))) || (((vcount+3) > cursor_y) && (vcount < (cursor_y + 3)))) &&
        //    ( (hcount < CROSS_MAX_X) && (hcount > CROSS_MIN_X) && (vcount < CROSS_MAX_Y) && (vcount > CROSS_MIN_Y) ) )
      xDepth <= ((hcount+CROSS_WIDTH) > cursor_x) && (hcount < (cursor_x+CROSS_WIDTH));
      //xBorder <= (hcount < CROSS_MAX_X) && (hcount > CROSS_MIN_X);
      //yBorder <= (vcount < CROSS_MAX_Y) && (vcount > CROSS_MIN_Y);
      yDepth <= ((vcount+CROSS_WIDTH) > cursor_y) && (vcount < (cursor_y + CROSS_WIDTH));
      //Border <= xBorder && yBorder;
      crosshair <= xDepth || yDepth;
      //(xDepth || yDepth); //(hcount==512) || (vcount==370); //0;


      if (ledFound)
         if(((hcount+HALFWIDTH) >= x && hcount < (x+HALFWIDTH)) && ((vcount+HALFHEIGHT) >= y && vcount < (y+HALFHEIGHT)))
            dstar <= 1;
      else
         dstar <= 0;

      pixel <= display_debug ? {hcount[8:6],15'b0} :
               ( crosshair  ?  {6'b0, 6'b1111_11, 6'b0}  :
               // ( crosshair  ?  {6'b0, 6'b1111_11, 6'b0} : vr_pixel);
               (dstar ? dstarPixel : vr_pixel)); //mux between test signal, crosshair, or vr_pixel

      //add to above (ledFinder) ? {dstarPixel[23:18],dstarPixel[15:10],dstarPixel[7:2]} : (all else)
      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.f

   //these values are for a black and white display, will need to add in instatiation of ycrcb2rgb for this to work
   assign vga_out_red = {pixel[17:12],2'b00}; //keep the higher order bits of r,g,b
   assign vga_out_green ={pixel[11:6],2'b00};
   assign vga_out_blue = {pixel[5:0],2'b00};
   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 via LEDs and hexDisplay

   // assign led = ~{vram_address[18:13], reset, switch[0]};
   assign led = ~{3'b0, mode, laserOn, ledFound, reset, display_debug};

   always @(posedge clock_27mhz)
      dispdata <= {disp_x, disp_y, disp_thetaH, disp_thetaV};
     // dispdata <= {vram_read_data,9'b0,vram_addr};
     // dispdata <= {ntsc_data,9'b0,ntsc_addr};

endmodule
```

```

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

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

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

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

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

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

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

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

///////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.
```

```
//
// Bug due to memory management will be fixed. The bug happens because
// memory is called based on current hcount & vcount, which will actually
// shows up 2 cycle in the future. Not to mention that these incoming data
// are latched for 2 cycles before they are used. Also remember that the
// ntsc2zbt's addressing protocol has been fixed.

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

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


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

   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;
   //input [35:0] chromaPixelData; //pixel data from ZBT
   //output [18:0] chromaPixelAddress; //Adress of pixel that need be looked up in
zbt_6111
   //input [10:0] cursor_x;
   //input [9:0] cursor_y;
   //input blank;

   //bunk code for address determination of chroma key pixel
   //always @(posedge clk)
      //if (clean_enter) //begin
         //chromaAddress <= {cursor_y,(~cursor-x[9:1]-9'd136)}; //address of the hue
we want to target
      //chromaPixel <= ;  //need vram_read_data that corresponds to the above address
      //end
      //wire [18:0] chromaAddress;

   //forecast hcount & vcount 8 clock cycles ahead to get data from ZBT
   //wire [10:0] hcount_f = (hcount >= 1048) ? (hcount - 1048) :
      //(hcount >=24) ? (hcount-24) : hcount;

```

```verilog
   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 [9:0] vcount_f = (hcount >= 1048) ? ((vcount == 768) ? 0 : vcount + 1) : vcount;

   // always@(posedge clk) //mux to send the chromakey address, only when blank is high //(blank) ?  {chromaAddress} :
   wire [18:0] vram_addr = {vcount_f, (~hcount_f[9:1] - 9'd136)};

   //wire [18:0]   vram_addr = {vcount, (~hcount[9:1] - 9'd150)};

   wire      hc2 = hcount[0]; //LSB of hcount, switches every clock cycle, used to select which pixel is displayed from the same address
   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'd1) ? vr_data_latched : last_vr_data; //each 18 bit pixel is latched for two clk cycle

   always @(posedge clk)
     vr_data_latched <= (hc2==1'd0) ? ((hcount_f < 10'd14)  ? 18'd0 : vram_read_data) : vr_data_latched;
     //((hcount_f < 10'd14)  ? 18'd0 : vram_read_data) : vr_data_latched; //fixing the far left hand side pixels to be black, based on forecasted pixels
     //(((hcount_f == cursor_x) || (vcount_f == cursor_y) || (hcount_f < 10'd11)) ? 18'd0 : vram_read_data): vr_data_latched; /
     //vram_read_data : vr_data_latched;
     //if hcount is less than 11 or the same value as the cursor's x position or y position, pixel is black

   always @(*) begin // each 36-bit word from RAM is decoded to 2 bytes
     case (hc2)
       1'd1: vr_pixel = last_vr_data[17:0];  //first pixel in address
       //(hcount_f < 10'd11) ? 18'd0 :
       //(hcount_f == cursor_x || vcount_f == cursor_y) ? 18'd0 :

       1'd0: vr_pixel = last_vr_data[35:18]; //second pixel in address
       //(hcount < 10'd11) ? 18'd0 :

     endcase
  end

endmodule // vram_display

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

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

   parameter NDELAY = 3;

```

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

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

  endmodule // delayN

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

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

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

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

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

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

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

```

   assign ref_clk = ref_clock;

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

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

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

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

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

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


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

   assign locked = lock1 && lock2;

endmodule

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

```

```verilog
// Carlos Cuevas & Cody Winkleblack
// 6.111 Laser Cyclops


/*
Purpose of module is to transpose x,y coordinates given from ledFinder
   to new coordinates in which both servos will be at neutral
   position when both angles given are pi/2

Transpose x,y so that theta = pi/2 @ neutral position
   thetaH_min @ x_max; thetaV_min @ y_min
   thus, flip negative sign for x so that thetaH_min @ x_min
*/

module transpose(
   input clock,
   input [10:0] x, y, width, height,
   output reg [10:0] x_out, y_out,
   output reg x_neg, y_neg // x|y is negative is asserted
   );

   wire [10:0] x_gain = width >> 1; // gain to x so that tan(pi/2) = z / 0 = undefined --> x + gain = 0
   wire [10:0] y_gain = height >> 1; // gain to y so that tan(pi/2) = z / 0 = undefined --> y + gain = 0

   always @(posedge clock) begin
      // x_transposed = -(x - width/2)
      if(x < x_gain) begin
         x_neg <= 0;
         x_out <= x_gain - x;
      end

      else begin
         x_neg <= 1;
         x_out <= x - x_gain;
      end

      // y_transposed = y - height/2
      if(y < y_gain) begin
         y_neg <= 1;
         y_out <= y_gain - y;
      end

      else begin
         y_neg <= 0;
         y_out <= y - y_gain;
      end

   end // always @(posedge clock)
endmodule // transpose
```

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

```verilog
//
// 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, chromaPixelData); //Need to add additonally mux at the tail end, so that chromakey value doesn't get sent back to display module

   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
   //input [18:0] chromaPixelAdress; //input for lookup table for chroma Pixel
   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
   output [35:0] chromaPixelData;

   // 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, required because there is a two cycle delay after we is enabled

   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


```

```
// Carlos Cuevas & Cody Winkleblack
// 6.111 Laser Cyclops
// ARCTAN TEST (after servoTest successful)

/*
This module takes a binary string and returns the decimal string
   in sets of 4 bits per decimal character.
This will be forwarded to the hex-display to portray these values
   in decimal form.
The format for this specific module using fractional decimals
   with input format 2dec7.
*/

module bin2decAngle(
   input clock, neutral, // neutral sets angle to pi/2, regardless of "value"
   input [8:0] value, // 2dec7; range-ish = [0, pi]
   output [15:0] hex // 16 bits for up to 4 possible decimal values
   );

   reg [4:0] ones, d1, d2, d3, d4, d5, d6, d7;
   assign hex = neutral ? {4'd1, 4'd5, 4'd7, 4'd1} : {ones[3:0], d1[3:0], d2[3:0], d3[
3:0]};

   reg [4:0] count = 0; // 5 bits counts in range = [0,31]
   reg [8:0] last_value;
   reg [5:0] carry;

   always @(posedge clock) begin
      if(count > 0) count <= count + 1;

      case(count)
         5'd0: begin
            if(last_value != value) begin
               count <= count + 1;
               last_value <= value;
               ones <= {2'b0, value[8:7]};
               d1 <= 0;
               d2 <= 0;
               d3 <= 0;
               d4 <= 0;
               d5 <= 0;
               d6 <= 0;
               d7 <= 0;
               carry <= 0;
            end
         end

         5'd1: begin
            d1 <= d1 + (last_value[6] ? 5 : 0);
         end

         5'd2: begin
            d1 <= d1 + (last_value[5] ? 2 : 0);
            d2 <= d2 + (last_value[5] ? 5 : 0);
         end

         5'd3: begin
```

```
                d1 <= d1 + (last_value[4] ? 1 : 0);
                d2 <= d2 + (last_value[4] ? 2 : 0);
                d3 <= d3 + (last_value[4] ? 5 : 0);
            end

            5'd4: begin
                d2 <= d2 + (last_value[3] ? 6 : 0);
                d3 <= d3 + (last_value[3] ? 2 : 0);
                d4 <= d4 + (last_value[3] ? 5 : 0);
            end

            5'd5: begin
                if(d2 > 9) begin
                    carry[5] <= 1;
                    d2 <= d2 - 10; end
                if(d3 > 9) begin
                    carry[4] <= 1;
                    d3 <= d3 - 10; end
            end

            5'd6: begin
                d1 <= d1 + carry[5];
                d2 <= d2 + carry[4] + (last_value[2] ? 3 : 0);
                d3 <= d3 + carry[3] + (last_value[2] ? 1 : 0);
                d4 <= d4 + carry[2] + (last_value[2] ? 2 : 0);
                d5 <= d5 + carry[1] + (last_value[2] ? 5 : 0);
                carry <= 0;
            end

            5'd7: begin
                if(d2 > 9) begin
                    carry[5] <= 1;
                    d2 <= d2 - 10; end
                if(d3 > 9) begin
                    carry[4] <= 1;
                    d3 <= d3 - 10; end
                if(d4 > 9) begin
                    carry[3] <= 1;
                    d4 <= d4 - 10; end
            end

            5'd8: begin
                d1 <= d1 + carry[5];
                d2 <= d2 + carry[4] + (last_value[1] ? 1 : 0);
                d3 <= d3 + carry[3] + (last_value[1] ? 5 : 0);
                d4 <= d4 + carry[2] + (last_value[1] ? 6 : 0);
                d5 <= d5 + carry[1] + (last_value[1] ? 2 : 0);
                d6 <= d6 + carry[0] + (last_value[1] ? 5 : 0);
                carry <= 0;
            end

            5'd9: begin
                if(d2 > 9) begin
                    carry[5] <= 1;
                    d2 <= d2 - 10; end
                if(d3 > 9) begin
                    carry[4] <= 1;
```

```
                  d3 <= d3 - 10; end
               if(d4 > 9) begin
                  carry[3] <= 1;
                  d4 <= d4 - 10; end
               if(d5 > 9) begin
                  carry[2] <= 1;
                  d5 <= d5 - 10; end
            end

            5'd10: begin
               d1 <= d1 + carry[5];
               d2 <= d2 + carry[4] + (last_value[0] ? 1 : 0);
               d3 <= d3 + carry[3] + (last_value[0] ? 5 : 0);
               d4 <= d4 + carry[2] + (last_value[0] ? 6 : 0);
               d5 <= d5 + carry[1] + (last_value[0] ? 2 : 0);
               d6 <= d6 + carry[0] + (last_value[0] ? 5 : 0);
               d7 <= d7 + (last_value[0] ? 5 : 0);
               carry <= 0;
            end

            5'd11: begin
               if(d2 > 9) begin
                  carry[5] <= 1;
                  d2 <= d2 - 10; end
               if(d3 > 9) begin
                  carry[4] <= 1;
                  d3 <= d3 - 10; end
               if(d4 > 9) begin
                  carry[3] <= 1;
                  d4 <= d4 - 10; end
               if(d5 > 9) begin
                  carry[2] <= 1;
                  d5 <= d5 - 10; end
               if(d6 > 9) begin
                  carry[1] <= 1;
                  d6 <= d6 - 10; end
            end

            5'd12: begin
               d1 <= d1 + carry[5];
               d2 <= d2 + carry[4];
               d3 <= d3 + carry[3];
               d4 <= d4 + carry[2];
               d5 <= d5 + carry[1];
               carry <= 0;
            end

            5'd13: begin
               if(d2 > 9) begin
                  carry[5] <= 1;
                  d2 <= d2 - 10; end
               if(d3 > 9) begin
                  carry[4] <= 1;
                  d3 <= d3 - 10; end
               if(d4 > 9) begin
                  carry[3] <= 1;
                  d4 <= d4 - 10; end
               if(d5 > 9) begin
                  carry[2] <= 1;
                  d5 <= d5 - 10; end
            end

            5'd14: begin
               d1 <= d1 + carry[5];
               d2 <= d2 + carry[4];
               d3 <= d3 + carry[3];
               d4 <= d4 + carry[2];
               carry <= 0;
            end

            5'd15: begin
               if(d2 > 9) begin
                  carry[5] <= 1;
                  d2 <= d2 - 10; end
               if(d3 > 9) begin
                  carry[4] <= 1;
                  d3 <= d3 - 10; end
               if(d4 > 9) begin
                  carry[3] <= 1;
                  d4 <= d4 - 10; end
            end

            5'd16: begin
               d1 <= d1 + carry[5];
               d2 <= d2 + carry[4];
               d3 <= d3 + carry[3];
               carry <= 0;
            end

            5'd17: begin
               if(d2 > 9) begin
                  carry[5] <= 1;
                  d2 <= d2 - 10; end
               if(d3 > 9) begin
                  carry[4] <= 1;
                  d3 <= d3 - 10; end
            end

            5'd18: begin
               d1 <= d1 + carry[5];
               d2 <= d2 + carry[4];
               carry <= 0;
            end

            5'd19: begin
               if(d2 > 9) begin
                  carry[5] <= 1;
                  d2 <= d2 - 10; end
            end

            5'd20: begin
               d1 <= d1 + carry[5];
               carry <= 0;
            end
```

```

          default: count <= 0;
        endcase // count

    end // always @(posedge clock)
endmodule // bin2dec
```

```verilog
// Carlos Cuevas & Cody Winkleblack
// 6.111 Laser Cyclops
// ARCTAN TEST (after servoTest successful)

/*
This module takes a binary string and returns the decimal string
   in sets of 4 bits per decimal character.
This will be forwarded to the hex-display to portray these values
   in decimal form.
The format for this specific module is integers only.

Table that converts value (if bit asserted) into decimal format {n3,n2,n1,n0}:
   value[10] = 1024
   value[9] = 512
   value[8] = 256
   value[7] = 128
   value[6] = 64
   value[5] = 32
   value[4] = 16
   value[3] = 8
   value[2] = 4
   value[1] = 2
   value[0] = 1
*/

module bin2decInt(
   input clock,
   input [10:0] value, // 11dec0; range = [0,2047]
   output [15:0] out // 16 bits for up to 4 possible decimal values
   );

   reg [6:0] n3, n2, n1, n0; // max possible number is 47 in n0 (need 7 bits)
   assign out = {n3[3:0], n2[3:0], n1[3:0], n0[3:0]};

   reg done = 1; // determines whether it accepts new values or not
   reg [10:0] last_value;
   reg [6:0] carry_n3, carry_n2, carry_n1;
   reg cascade; // used for computing decimal bits from value
   reg [4:0] count; // iterates down through bits in value [0,10]

   always @(posedge clock) begin
      if(done) begin // ready for new value to compute
         last_value <= value; // update

         if(last_value != value) begin // initialization
            done <= 0;
            carry_n3 <= 0;
            carry_n2 <= 0;
            carry_n1 <= 0;
            cascade <= 1;
            count <= 0;
            n3 <= 0;
            n2 <= 0;
            n1 <= 0;
            n0 <= 0;
         end
      end
```

```

        else begin // calculation block (~DONE)
           if(cascade) begin // initial calc phase in which all bits in value are translated to decimal form
              count <= count + 1; // increment count every time step

              case(count)
                 5'd0: begin
                    if(last_value[10]) begin
                       n3 <= n3 + 1;
                       n2 <= n2 + 0;
                       n1 <= n1 + 2;
                       n0 <= n0 + 4;
                    end
                 end

                 5'd1: begin
                    if(last_value[9]) begin
                       n3 <= n3 + 0;
                       n2 <= n2 + 5;
                       n1 <= n1 + 1;
                       n0 <= n0 + 2;
                    end
                 end

                 5'd2: begin
                    if(last_value[8]) begin
                       n3 <= n3 + 0;
                       n2 <= n2 + 2;
                       n1 <= n1 + 5;
                       n0 <= n0 + 6;
                    end
                 end

                 5'd3: begin
                    if(last_value[7]) begin
                       n3 <= n3 + 0;
                       n2 <= n2 + 1;
                       n1 <= n1 + 2;
                       n0 <= n0 + 8;
                    end
                 end

                 5'd4: begin
                    if(last_value[6]) begin
                       n3 <= n3 + 0;
                       n2 <= n2 + 0;
                       n1 <= n1 + 6;
                       n0 <= n0 + 4;
                    end
                 end

                 5'd5: begin
                    if(last_value[5]) begin
                       n3 <= n3 + 0;
                       n2 <= n2 + 0;
                       n1 <= n1 + 3;
```

```
                        n0 <= n0 + 2;
                     end
                  end

                  5'd6: begin
                     if(last_value[4]) begin
                        n3 <= n3 + 0;
                        n2 <= n2 + 0;
                        n1 <= n1 + 1;
                        n0 <= n0 + 6;
                     end
                  end

                  5'd7: begin
                     if(last_value[3]) begin
                        n3 <= n3 + 0;
                        n2 <= n2 + 0;
                        n1 <= n1 + 0;
                        n0 <= n0 + 8;
                     end
                  end

                  5'd8: begin
                     if(last_value[2]) begin
                        n3 <= n3 + 0;
                        n2 <= n2 + 0;
                        n1 <= n1 + 0;
                        n0 <= n0 + 4;
                     end
                  end

                  5'd9: begin
                     if(last_value[1]) begin
                        n3 <= n3 + 0;
                        n2 <= n2 + 0;
                        n1 <= n1 + 0;
                        n0 <= n0 + 2;
                     end
                  end

                  5'd10: begin
                     if(last_value[0]) begin
                        n3 <= n3 + 0;
                        n2 <= n2 + 0;
                        n1 <= n1 + 0;
                        n0 <= n0 + 1;
                     end
                     cascade <= 0; // done with initial computing
                  end

                  default: ;
               endcase // count
            end // cascade

            else begin // final part of computing (the carry) ~cascade
               if(n0 > 9) begin
                  carry_n1 <= carry_n1 + 1;
                  n0 <= n0 - 10; end
               else if(carry_n1 > 0) begin
                  n1 <= n1 + carry_n1;
                  carry_n1 <= 0; end
               else if(n1 > 9) begin
                  carry_n2 <= carry_n2 + 1;
                  n1 <= n1 - 10; end
               else if(carry_n2 > 0) begin
                  n2 <= n2 + carry_n2;
                  carry_n2 <= 0; end
               else if(n2 > 9) begin
                  carry_n3 <= carry_n3 + 1;
                  n2 <= n2 - 10; end
               else if(carry_n3 > 0) begin
                  n3 <= n3 + carry_n3;
                  carry_n3 <= 0; end
               else done <= 1; // end of computation
            end // ~cascade
         end // ~DONE

      end // always @(posedge clock)
   endmodule // bin2dec
```

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: 
// Engineer: 
// 
// Create Date:    16:39:31 12/06/2018 
// Design Name: 
// Module Name:    chromaPixel 
// Project Name: 
// Target Devices: 
// Tool versions: 
// Description: 
//
// Dependencies: 
//
// Revision: 
// Revision 0.01 - File Created
// Additional Comments: 
//
//////////////////////////////////////////////////////////////////////////////////
module chromaPixel(
    input button_enter,
    input [10:0] hcount,
    input [9:0] vcount,
   input [10:0] x_cursor,
   input [9:0] y_cursor,
    input clk,
    input reset,
    input [17:0] vram_pixel,
    output [17:0] chromaPixel
    );

   reg [10:0] x_select; //crosshair value as selected by the user
   reg [9:0] y_select;
   reg [17:0] chromaPixelReg;


   always @(posedge clk) begin
      if (button_enter)
         {x_select,y_select} <= {x_cursor,y_cursor};
      if (hcount == x_select && vcount == y_select)
         chromaPixelReg <= vram_pixel;
   end
   assign chromaPixel = chromaPixelReg;

endmodule

```

```


`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:16:59 12/05/2018
// Design Name:
// Module Name:    crosshair
// Project Name:
// Target Devices:
// Tool versions:
// Description: Takes input from up, down, left, and right buttons and edits the
crosshair location
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module crosshair(
    input reset, clk,
   input up,
    input down,
    input left,
    input right,
   output[10:0] cursor_x,
   output[9:0] cursor_y
    );

   reg [10:0] regCursor_x; //initial cursor position is at the center
   reg [9:0] regCursor_y;
   reg [19:0] count; // for timer; 20 bits holds up to 1,048,576 values
   parameter TIMER = 450_000; // sets a limit at how fast a cursor moves
   parameter X_MAX = 11'd1023;
   parameter X_MIN = 11'd0;
   parameter Y_MAX = 10'd768;
   parameter Y_MIN = 10'd0;
   parameter X_DEFAULT = 11'd512;
   parameter Y_DEFAULT = 10'd384;

   always@(posedge clk) begin
      if(reset) begin
         regCursor_x <= X_DEFAULT;
         regCursor_y <= Y_DEFAULT;
         count <= 0;
      end

      else if(count >= TIMER) begin
         count <= 0;
         if ((regCursor_x < X_MAX) && right)
            regCursor_x <= regCursor_x + 1;
         else if ((regCursor_x > X_MIN) && left)
            regCursor_x <= regCursor_x - 1;
         if ((regCursor_y < Y_MAX) && down)
            regCursor_y <= regCursor_y + 1;
         else if ((regCursor_y > Y_MIN) && up)
            regCursor_y <= regCursor_y - 1;
      end

      else count <= count + 1;
   end

   assign cursor_x = regCursor_x;
   assign cursor_y = regCursor_y;


endmodule

```

```
// Carlos Cuevas & Cody Winkleblack
// 6.111 Laser Cyclops

// this module is responsible for the laser's on/off state
module laser(
   input clock, ledFound,
   input [8:0] thetaH, thetaV, // in radians; 2dec7
   output reg laserOn
   );

   parameter ERROR = 9'b00_0001_000; // default angle ERROR = 0.0625 radians
   reg [8:0] thetaH_minus1, thetaV_minus1;

   always @(posedge clock) begin
      if(ledFound && // if ledFound=1 AND |theta[n-1] - theta[n]| <= ERROR (for both planes)
            (thetaH - thetaH_minus1) <= ERROR &&
            (thetaV - thetaV_minus1) <= ERROR) begin
         laserOn <= 1;
      end

      else laserOn <= 0;

      thetaH_minus1 <= thetaH;
      thetaV_minus1 <= thetaV;

   end // always @(posedge clock)
endmodule // laser
```

```
// Carlos Cuevas & Cody Winkleblack
// 6.111 Laser Cyclops

/*
This module takes a pixel value per (x,y) coordinate and determines if that RGB value is
   a part of the target.
There are two modes to this module: infrared, chromaKey
   1. infrared
      Assumes all pixels that are NOT from the target to be black.
      Returns center of all non-black pixels.
   2. chromaKey
      Given chromaKey as a hue, all incoming RGB values are first
         translated to its respective hue and then compared to the
         key (targetHue).
      Returns the center of all pixels that match the chromaKey
         (targetHue).
*/

module ledFinder(
   input clock, reset,
   input [8:0] pixel_translated, // hue values
   input [10:0] hcount,
   input [9:0] vcount,
   input [8:0] targetHue,
   input mode, // 1 = infrared; 0 = chromaKey
   output [10:0] x, y,
   output reg ledFound
   );

   wire [10:0] vcount_exp = {1'b0, vcount};

   reg [10:0] min_x, max_x, min_y, max_y;
   reg [8:0] last_pixel;

   assign x = (min_x + max_x) >> 1;
   assign y = (min_y + max_y) >> 1;

   always @(posedge clock) begin
      last_pixel <= pixel_translated; // serves as an auto_reset signal

      if(reset || (last_pixel != pixel_translated)) begin // reset || auto_reset
         min_x <= 11'b1111_1111_111;
         min_y <= 10'b1111_1111_11;
         max_x <= 11'b0;
         max_y <= 10'b0;
         ledFound <= 0;
      end // if(reset)

      else begin
         if (mode) begin // mode = 1 (infrared targetting)
            if(pixel_translated != 0) begin // if pixel hue is NOT black
               ledFound <= 1;
               if(hcount < min_x) min_x <= hcount;
               if(hcount > max_x) max_x <= hcount;
               if(vcount_exp < min_y) min_y <= vcount_exp;
               if(vcount_exp > max_y) max_y <= vcount_exp;
```

```
                end
            end

            else begin // mode = 0 (chromaKey targetting)
                if (pixel_translated == targetHue) begin
                    ledFound <= 1;
                    if(hcount < min_x) min_x <= hcount;
                    if(hcount > max_x) max_x <= hcount;
                    if(vcount_exp < min_y) min_y <= vcount_exp;
                    if(vcount_exp > max_y) max_y <= vcount_exp;
                end
            end
        end // else

    end // always @(posedge clock)
endmodule // ledFinder
```

```verilog
// Carlos Cuevas & Cody Winkleblack
// 6.111 Laser Cyclops
// ARCTAN TEST (after servoTest successful)

/*
This module takes two values and its reciprocals and
outputs the numerator and denomenator for arctan.

There's a module for reciprocal given by Xilinx.

theta = arctan(b/a)
if(a>b):
   num = b/a
   den = 1
else:
   num = 1
   den = a/b
*/

module numden(
   input clock,
   input [10:0] a, b, // 11dec0
   input [10:0] a_inv, b_inv, // 0dec11
   output [10:0] num, den // 2dec9
   );

   wire [21:0] a_exp = {a, 11'b0}; // 11dec11
   wire [21:0] a_inv_exp = {11'b0, a_inv};
   wire [21:0] b_exp = {b, 11'b0};
   wire [21:0] b_inv_exp = {11'b0, b_inv};

   reg [43:0] num_exp, den_exp; // 22dec22

   assign num = num_exp[23:13]; // integer bits = 23:22; fracional bits = 22:13
   assign den = den_exp[23:13];

   always @(posedge clock) begin
      if(a > b) begin
         num_exp <= a_inv_exp * b_exp;
         den_exp <= {22'b1, 22'b0};
      end

      else begin
         num_exp <= {22'b1, 22'b0};
         den_exp <= a_exp * b_inv_exp;
      end

   end // always @(posedge clock)
endmodule
```

```
`timescale 1ns / 1ps

module picture_blob
   #(parameter HALFWIDTH = 128,
               HALFHEIGHT = 120,
               WIDTH = 256,    // default picture width
               HEIGHT = 240)  // default picture height
   (input pixel_clk,
    input [10:0] x,hcount,
    input [9:0] y,vcount,
    output reg [17:0] pixel);

   wire [15:0] image_addr;   // num of bits for 256*240 ROM
   wire [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount-x) + (vcount-y) * WIDTH;
   lab3rom  DStar(.clka(pixel_clk), .addra(image_addr), .douta(image_bits)); //this
is the line we modify for changing the picture

   // use color map to create 8 bits R, 8 bits G, 8 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
   // use color map to create 8bits R, 8bits G, 8 bits B;
   lab3colormap  redmap(.clka(pixel_clk), .addra(image_bits), .douta(red_mapped));
   //green_coe gcm (.clka(pixel_clk), .addra(image_bits), .douta(green_mapped));
   //blue_coe bcm (.clka(pixel_clk), .addra(image_bits), .douta(blue_mapped));

   // note the one clock cycle delay in pixel!
   always @ (posedge pixel_clk) begin
     if (((hcount+HALFWIDTH) >= x && hcount < (x+HALFWIDTH)) && //change in this line
will align picture center and LED center, rather than having picture top, left start
at x,y
          ((vcount+HALFHEIGHT) >= y && vcount < (y+HALFHEIGHT)))
        pixel <= {red_mapped[7:2], red_mapped[7:2], red_mapped[7:2]}; // greyscale
        //pixel <= {red_mapped, 16h'0}; // only red hues
        else pixel <= 0;
   end
endmodule


```

```verilog
// Carlos Cuevas & Cody Winkleblack
// 6.111 Laser Cyclops

/*
This module takes a [18'] RGB value and returns its respective [9'] hue.
This module assumes that the 6 MSB per color are inputted.

Formula:
R' = R/255
G' = G/255
B' = B/255
Cmax = max(R', G', B')
Cmin = min(R', G', B')
delta = Cmax - Cmin

H = {0,                          delta = 0
   60 * ((G'-B')/delta)mod6),     Cmax = R'
   60 * ((B'-R')/delta) + 2),     Cmax = G'
   60 * ((R'-G')/delta) + 4),     Cmax = B'
   }
   range = [0,360]

S = {0,                       delta = 0
   delta / (1 - abs(2L-1)),   delta != 0
   }

L = (Cmax + Cmin) / 2
*/

module rgb2hue(
   input clock,
   input [17:0] rgb,
   output reg [8:0] hue // 9 bits needed for hue range = [0,360]
   );

   parameter INV_255 = 16'b0000_0001_0000_0001; // format 0dec16; 1/255 =
0.00392156862745
   wire [7:0] r = {rgb[17:12], 2'b0};
   wire [7:0] g = {rgb[11:6], 2'b0};
   wire [7:0] b = {rgb[5:0], 2'b0};
   reg [47:0] r_prime, g_prime, b_prime; // 16dec32
   wire [16:0] r_prime2 = r_prime[32:16]; // 1dec16
   wire [16:0] g_prime2 = g_prime[32:16];
   wire [16:0] b_prime2 = b_prime[32:16];
   reg [16:0] c_max, c_min; // 1dec16
   reg [16:0] delta; // 1dec16

   wire [16:0] delta_inv; // 16dec1
   inverse deltaInv (.clk(clock), .dividend(2'b1), .divisor(delta), .fractional(
delta_inv));

   reg [63:0] valueR, valueG, valueB; // 32dec32
   wire [8:0] remainder; // 9dec0
   modulo mod6 (.clock(clock), .value(valueR[40:32]), .modulus(3'd6), .remainder(
remainder));

   always @(posedge clock) begin
```

```
      r_prime <= {r, 16'b0} * {8'b0, INV_255}; // 2(8dec16) = 16dec32
      g_prime <= {g, 16'b0} * {8'b0, INV_255};
      b_prime <= {b, 16'b0} * {8'b0, INV_255};

      c_max <= (r_prime2 > g_prime2) ? ((r_prime2 > b_prime2) ? r_prime2 : b_prime2) :
                     ((g_prime2 > b_prime2) ? g_prime2 : b_prime2);
      c_min <= (r_prime2 < g_prime2) ? ((r_prime2 < b_prime2) ? r_prime2 : b_prime2) :
                     ((g_prime2 < b_prime2) ? g_prime2 : b_prime2);
      delta <= c_max - c_min;

      valueR <= {15'b0, (g_prime2 - b_prime2)} * {delta_inv, 15'b0}; // 2(16dec16) = 32dec32
      valueG <= {15'b0, (b_prime2 - r_prime2)} * {delta_inv, 15'b0};
      valueB <= {15'b0, (r_prime2 - g_prime2)} * {delta_inv, 15'b0};

      //remainder <= valueR[40:32] % 6;

      if(delta == 0) hue <= 0;

      else if(c_max == r_prime2) begin
         // hue <= 60 * ((G'-B')/delta)mod6;
         hue <= 60 * remainder;
      end

      else if(c_max == g_prime2) begin
         // hue <= 60 * ((B'-R')/delta) + 2;
         hue <= 60 * (valueG[40:32] + 2);
      end

      else if(c_max == b_prime2) begin
         // hue <= 60 * ((R'-G')/delta) + 4;
         hue <= 60 * (valueB[40:32] + 4);
      end

      else $display("hue Error");
   end // always @(posedge clock)
endmodule
```

```
// Carlos Cuevas & Cody Winkleblack
// 6.111 Laser Cyclops
// ARCTAN TEST (after servoTest successful)

// this module sends a pulse to the servos every 20ms (param)
module servoClock(
   input clock, // given 27MHz clock
   output reg pulse_20ms
   );

   parameter TARGET = 540_000; // 27,000,000 * (20/1000) = 540,000
   reg [19:0] counter = 20'b0; // 20 bits holds up to 1,048,576 values

   always @(posedge clock) begin
      if(counter >= TARGET) begin // if TARGET is hit, pulse
         counter <= 0;
         pulse_20ms <= 1;
      end // if(counter >= TARGET)

      else begin
         counter <= counter + 1;
         pulse_20ms <= 0;
      end

   end // posedge clock
endmodule
```

```verilog
`timescale 1ns / 1ps

module siren(
   input clock, sirenOn, complexMelody, // clock MUST BE 27MHz from Labkit
   output reg sound
   );

   reg last_sirenOn; // for remembering when the melody has started
   reg [5:0] note; // the note in the melody; 5 measures * 8 eighth-notes = 40 notes (need 6 bits)
   reg [5:0] oldNote; // to know when to ping pitchStart;
   reg eighthStart; // input to eighthNote
   wire eighthPing; // output to eighthNote
   reg pitchStart; // input to pitchMaker; resets pitchMaker to given pitch
   reg [4:0] pitch; // input to pitchMaker: frequency to emit sound (pitch); 5 bits for max 32 pitches
   wire pitchOut; // output to pitchMaker, will be connected to sound if sirenOn
   wire [5:0] last_note = complexMelody ? 6'd14 : 6'd1;

   eighthNote eight(.clock(clock), .start(eighthStart), .pulse(eighthPing)); // shortened duration of eighthnote for final project
   pitchMaker soundFreq(.clock(clock), .start(pitchStart), .pitch(pitch), .sound(pitchOut));

   always @(posedge clock) begin
      if(!sirenOn) begin // if siren should be off
         sound <= 0;
         last_sirenOn <= 0;
      end

      else begin // if siren should be playing (gets more complicated due to changing pitches)
         sound <= pitchOut;

         if(!last_sirenOn) begin // reset variables to default
            last_sirenOn <= 1;
            note <= 0;
            oldNote <= 6'd1;
            eighthStart <= 1;
         end

         else begin // choose pitch based on what note you are in melody
            if(!complexMelody) begin // MELODY = Middle C pulses
               case(note)
                  6'd0: pitch <= 5'd0;
                  6'd1: pitch <= 5'd31;
                  default: ;
               endcase
            end

            else begin // MELODY = PEW PEW
               case(note)
                  6'd0: pitch <= 5'd1; // start at A5, progress downward in pitch
                  6'd1: pitch <= 5'd2;
                  6'd2: pitch <= 5'd3;
                  6'd3: pitch <= 5'd4;
                  6'd4: pitch <= 5'd5;
                  6'd5: pitch <= 5'd6;
                  6'd6: pitch <= 5'd7;
                  6'd7: pitch <= 5'd8; // A4
                  6'd8: pitch <= 5'd9;
                  6'd9: pitch <= 5'd10;
                  6'd10: pitch <= 5'd11;
                  6'd11: pitch <= 5'd12;
                  6'd12: pitch <= 5'd13;
                  6'd13: pitch <= 5'd14;
                  6'd14: pitch <= 5'd15; // A3
                  default: pitch <= 5'd31; // silent until last_note hit
               endcase
            end // Melody = Viva la Vida

            if(!(note == oldNote)) begin
               pitchStart <= 1;
            end

            oldNote <= note;

            if(pitchStart) begin // end pitchStart pulse after 1 tick
               pitchStart <= 0;
            end

            if(eighthStart) begin // end eightStart pulse after 1 tick
               eighthStart <= 0;
            end

            if(eighthPing) begin // increment note at end of eighth-note; restart melody if last_note hit
               note <= (note >= last_note) ? 0 : (note + 1);
            end

         end // choosing what pitch (!reset)

      end // if siren should be playing
   end // always @(posedge clock)
endmodule // siren


module eighthNote(
   input clock, start, // clock MUST BE clock_27mhz
   output reg pulse
   );

   reg [22:0] counter;
   parameter TARGET = 23'd421_875; // duration for Star Wars laser firing = 64 notes/s

   always @(posedge clock) begin
      if(start) begin // reset mode, if START=1
         counter <= 0;
         pulse <= 0;
      end

      else if(counter == TARGET) begin // pulse and reset counter
         pulse <= 1;
         counter <= 0;
      end

      else begin // increment otherwise and remove pulse
          counter <= counter + 1;
          pulse <= 0;
      end

   end // always @(posedge clock_27mhz)

endmodule // eighthNote


module pitchMaker(
   input clock, start, // clock = 27MHz
   input [4:0] pitch, // 5 bits --> up to 32 different pitches
   output reg sound
   );

   /*
   Table of Pitches
   0 - Middle C = C4 - 261.63 Hz (from the 4th octave)

   Key Signature: A major (tonic from 4th octave = A4)
   1 - A5   - 880 Hz
   2 - G#5  - 830.61 Hz
   3 - F#5  - 739.99 Hz
   4 - E5   - 659.25 Hz
   5 - D5   - 587.33 Hz
   6 - C#5  - 554.37 Hz
   7 - B4  - 493.88 Hz
   8 - A4  - 440 Hz
   9 - G#4 - 415.30 Hz
   10 - F#4 - 369.99 Hz
   11 - E4  - 329.63 Hz
   12 - D4  - 293.66 Hz
   13 - C#4 - 277.18 Hz
   14 - B3  - 246.94 Hz
   15 - A3  - 220 Hz

   16-31 SILENT      - 0 Hz
   */

   reg [24:0] counter; // 25 bits holds 33,554,432 values
   reg [24:0] target; // this is the value the "counter" needs to hit to send sound=1

   always @(posedge clock) begin
        if(start) begin
            counter <= 0;
            sound <= 0;
            case(pitch)
                5'd0: target <= 25'd103_199; // 27M / 261.63
                5'd1: target <= 25'd30_682; // 27M / 880
                5'd2: target <= 25'd32_506; // 27M / 830.61
                5'd3: target <= 25'd36_487; // 27M / 739.99
                5'd4: target <= 25'd40_956; // 27M / 659.25
                5'd5: target <= 25'd45_971; // 27M / 587.33
                5'd6: target <= 25'd48_704; // 27M / 554.37
```

```
                    5'd7: target <= 25'd54_669; // 27M / 493.88
                    5'd8: target <= 25'd61_363; // 27M / 440
                    5'd9: target <= 25'd65_013; // 27M / 415.30
                    5'd10: target <= 25'd72_975; // 27M / 369.99
                    5'd11: target <= 25'd81_910; // 27M / 329.63
                    5'd12: target <= 25'd91_943; // 27M / 293.66
                    5'd13: target <= 25'd97_410; // 27M / 277.18
                    5'd14: target <= 25'd109_338; // 27M / 246.94
                    5'd15: target <= 25'd122_727; // 27M / 220
                    default: target <= 0; // SILENT (pitch > 15)
                endcase // pitch
            end

            else if(target == 25'b0) begin
                sound <= 0;
            end

            else if(counter == (target >> 2)) begin // begin pulse SOUND
                sound <= 1;
            end

            else if(counter == target) begin // end pulse SOUND
                sound <= 0;
                counter <= 0; // restart (as long as desired)
            end


            if(counter < target) begin // increment counter if NOT at or past target
                counter <= counter + 1;
            end

        end // always @(posedge clock)
    endmodule // pitchMaker
```

```verilog
// Carlos Cuevas & Cody Winkleblack
// 6.111 Laser Cyclops
// ARCTAN TEST (after servoTest successful)

/*
/*
Pulse notes:
- lower bound = 0 radians = 0.5ms pulse = 13,500 count_steps
- neutral = pi/2 radians = 1.50ms pulse = 40,500 count_steps
- higher bound = pi radians = 2.5ms pulse = 67,500 count_steps

Conversion:
[pulse] = [rad]*2/pi + 0.5
[pulse_target] = 27_000_000 * [pulse]/1000 = 27_000 * [pulse]

1/pi = 0.318309886184 = 9'b00.0101_000 (2dec7)
27,000 = 15'b110_1001_0111_1000 (15dec0)

WARNING: watch use of decimals (keep decimal location consistent)
*/

module servo(
   input clock, ledFound, pulse_20ms, // clock must be 27MHz
   input [8:0] theta, // in radians and format 2dec7; 9 bits holds up to 512 values (need [0,pi] set-space)
   output reg pulse_servo
   );

   parameter TWO_OVER_PI = 9'b00_1010_001; // format 2dec7; value equal to 2/pi = 0.636619772368 = 0.10100010111110011
   reg [19:0] counter; // 20 bits can hold 1,048,576 values
   reg [19:0] pulse_target; // value that counter will count up to for a specified angle

   reg [17:0] pulse1; // format 4dec14; theta * 2/pi
   wire [8:0] pulse2 = pulse1[15:7]; // format 2dec7; cut to 9 bits
   reg [8:0] pulse3; // format 2dec7; adds 0.5
   reg [43:0] pulse4; // format 30dec14 (44 bits); 27,000 * pulse3
   wire [19:0] pulse5 = pulse4[33:14]; // format 20dec0; the target for count via calculations

   always @(posedge clock) begin
      pulse1 <=  theta * TWO_OVER_PI;
      pulse3 <= (pulse2[6] == 1) ? ((pulse2[7] == 1) ? {3'b100, pulse2[5:0]} : {3'b010, pulse2[5:0]})
                                          : {pulse2[8:7], 1'b1, pulse2[5:0]}; // format 2dec7; adding 0.5, or a 1 to the MSB in pulse2[6:0]
      pulse4 <= 22'b110_1001_0111_1000__0000_000 * {13'b0, pulse3}; // 30dec14 = 15dec7 * 15dec7 = 27,000 * pulse3_expanded

      if(pulse_20ms) begin // determine the pulse needed for "theta"
         pulse_servo <= 1;
         counter <= 0;

         if(ledFound) begin
            // format 20dec0; calculation finally translates to a target for counter
            // verifies that pulse_target WILL NOT go beyond bounds, which would
```

```
break servo
            pulse_target <= (pulse5 < 20'd16_500) ? 20'd16_500 :
                              ( (pulse5 > 20'd67_500) ? 20'd67_500 : pulse5 );
         end // if(ledFound)

         else begin // implement if you want to return to NEUTRAL STATE if !ledFound;
else, will use last pulse
            pulse_target <=  20'd40_500; // 27,000,000 * 1.5/1000 = 27,000 * 1.5 =
40,500
         end // if(!ledFound)
      end // if(pulse_20ms)

      else begin
         if(counter >= pulse_target) pulse_servo <= 0;
         if((counter + 1) > 0) counter <= counter + 1;
      end // else
   end // always @(posedge clock)


endmodule // servo
```