FPGA Camera Controlled Traffic Lights

Jessica Quaye & Premila Rowles

6.111 Fall 2018

Table Of Contents

1 Project Summary

According to the World Health Organization, almost 1.25 million people die every year as a result of road accidents. Out of this horrifying number, 90% of these accidents occur in low- and middle-income countries which don't have resources for managing such accidents; those with resources prioritize other issues over road safety. For example, in Ghana, one cannot call any emergency service without airtime on your phone. This means that if there is an accident, people don't have immediate access to help unless somebody with airtime is present and can reach the ambulance services in time. Even in the event of successfully reaching the personnel, it is sometimes difficult to communicate the location of the emergency to them until it's too late.

In addition, almost everybody interacts with traffic lights on a daily basis; whether in the capacity of a pedestrian or of a driver, we use traffic lights in our daily commute. Most people have had unpleasant experiences with traffic lights when they are stuck in traffic for long periods of time but the signals don't move to balance the number of cars on both roads. This inability to effectively handle rush hour coupled with large volumes of traffic causes frustration on the road.

In order to optimize traffic on roads and help bridge the emergency communication gap, we have created a traffic light controller that takes in information from cameras observing the road and synthesizes the information using an FPGA (field-programmable gate array). This data is processed, analyzed, and sent to a finite state machine (FSM) that controls the traffic light signals based on the traffic direction that has more cars.

We wanted to create a system that was reliable for use in countries which have communication challenges when a road accident occurs, as well as improve traffic light efficiency for road users.

In addition to controlling the traffic light signals, the camera information is also shown as a live display (mirroring the street) on the Video Graphics Array (VGA) monitor. The system has playback functionality and can be used to review what happened after an accident or road danger occured. This paper provides a detailed description of our project, including an exposition on how the project was implemented and can be replicated, information on our design process and challenges, as well as reflections and lessons learned.

2 Project Goals

Minimum Product

- Given camera input, identify cars on a two-way street and provide information about their location.
- Convert the output of the NTSC camera to YCrCb space, then to RGB space, and finally to HSV space to simplify object detection.
- Implement traffic logic on a two-way street which changes car signal based on where there is a larger number of cars.
- Implement pedestrian push-to-walk signal for traffic lights.
- Draw visualization of camera input on the screen with signals changing in sync with traffic signals.

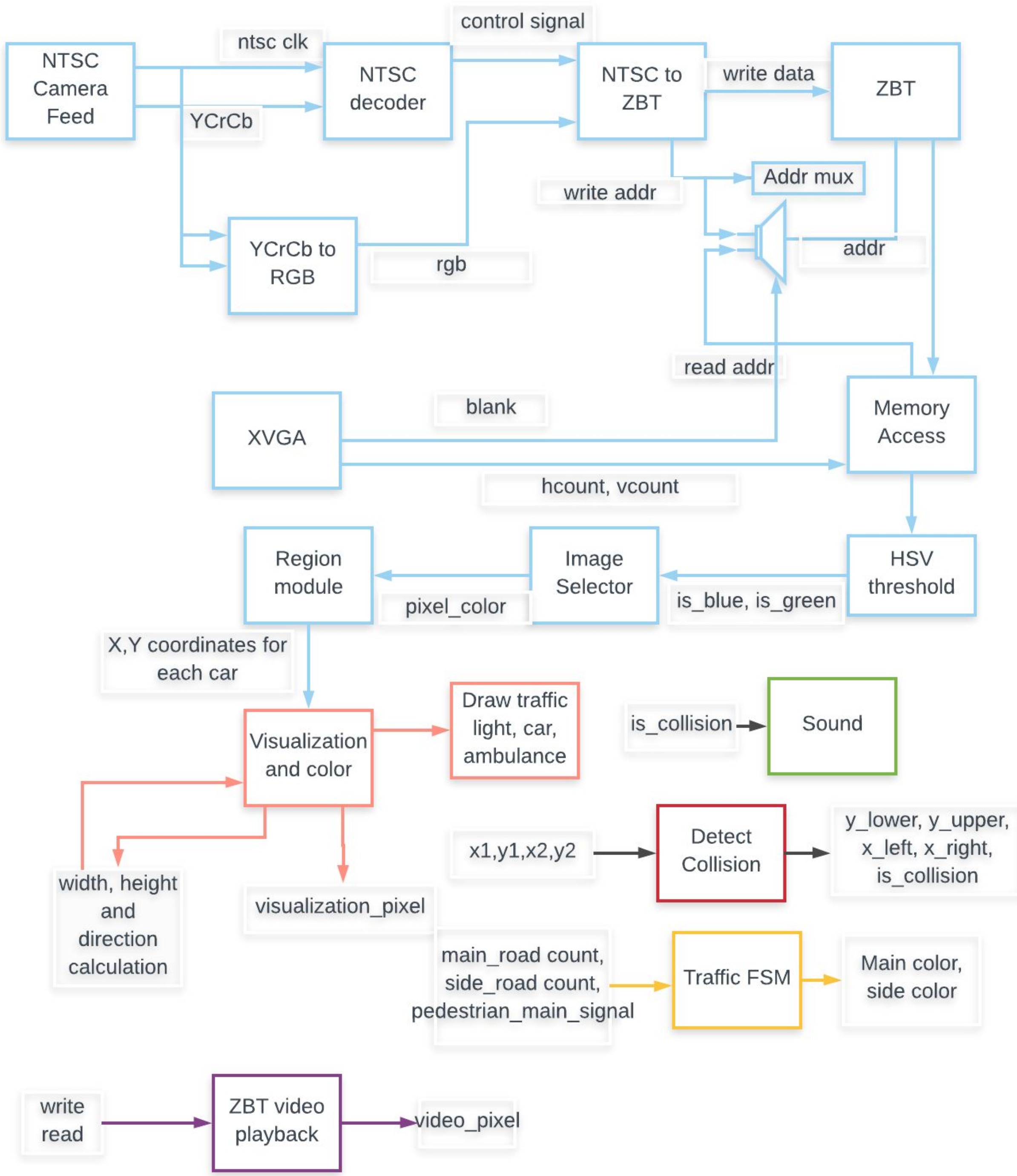
Expected Functionality

- Model collision of cars in visualization. When there is a collision, ambulance appears on screen and moves towards the location where the collision was detected. Before integration, this collision is simulated by a car that is continually moving on the screen, and a mouse input which controls the blob of “another car”.
- Add sound effects for ambulance.

Stretch Features

- Store the frames of images being shown on the VGA monitor and make it possible to playback a downsampled video.
- Detect reckless driving by using image processing to analyze car movement (swerving, overspeeding, etc).

3 System Block Diagram



4 Subsystems

4.1 Camera Input and Color Space Conversion - Premila

Modules: (All provided by staff but modified - see description below) zbt_6111, vram_display, adv7185init, ntsc_decode*, YCrCb2RGB, ntsc_to_zbt
Files: zbt_6111.v, ntsc_decode.v, ycrcb2rgb.v, ntsc_to_zbt.v

Basically, we take the raw NTSC (National Television Standards Committee) output which is in the form of analog signals and convert it to a digital format via the **adv7185** module. The three main signals coming from the camera output include
i) F (Field): 1 indicates even field, 0 indicates an odd field
ii) V: vertical sync signal indicating the start of a new frame
iii) H: the horizontal sync signal indicating the start of a new horizontal line

We need to store these digital bits in ZBT memory on the labkit in order to process the NTSC video data. The staff provided us with sample verilog that takes black and white NTSC video, stores it into ZBT and then displays it on the monitor with a screen size approximately 700 pixels wide x525 pixels tall.

We had to make several changes in order to display color on the monitor. The camera output is in YCrCb space; Y component: green is dominant but red and green have some input as well, Cr component: chromared (so in an image the red lights would be bright), and Cb : chromablue,(so blue objects will be bright). The sample code stores the Y value for four pixels in vr_pixel, which outputs a grayscale image. Since the ntsc_decode module provides a 30 bit YCrCb signal, and a location in ZBT memory is 36 bits wide, we can store 6 bits for each of Y, Cr, and Cb per pixel, allowing us to store 2 pixels in one location. Now we are extracting 18 bits from ntsc_decode instead of 8 bit pixels for the Y value and **ntsc2zbt** is now writing two 18 bit pixels, instead of four 8 bit pixels. vr_pixel is now 18 bits instead of 8 bits. Before, we were writing to memory when hcount[1:0] = 2 because we were storing four 8 bit pixels. Now we write to memory when hcount[0] = 1 because each pixel data is 18 bits wide. When we were sending four 8 bit pixels, the lower order two bits were not used, so the memory address changes every four pixels. Now each memory location contains only two pixels so memory addressing is twice as fast and only the LSB is not used. We adjusted myaddr2 to only ignore the lsb instead of ignoring two lower order bits. Specifically, in the ZBT address computation section, we remove the higher bit 0 and change x_addr to [9:1] from [9:2], in order to save every 2 addresses instead of every 4. We changed **vram_display** to read a new address every 2 pixels instead of every 4 as well.

The address signal for the vram_display module needs to be muxed since we are using this port for both read and write addresses. (read requests are interleaved between write requests using the write enable signal). We then pass in the YCrCb values to the **ycrcb2rgb** module. Once we have our rgb output, we can send those pixels to **ntsc_to_zbt** to store using **zbt_6111** and the **vram_display** module sends pixels to the VGA display.

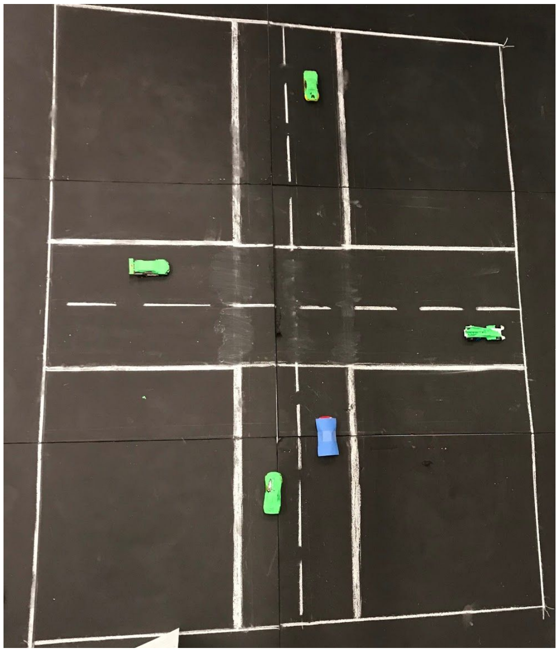


Figure 1: Bird eye camera view of street

4.2 Chroma Key Compositing - Premila

Modules: hsv_threshold, image_selector
Files: hsv_threshold.v, image_selector.v

The hsv_threshold module sets upper and lower bounds for thresholds on hue and value. We found the bounds by adjusting first the upper bound until the car was distinct from the rest of the image and most noise was filtered out. The module can detect green and blue cars. Pixels that have HSV values within the corresponding bounds are assigned to either the colors blue or green.

We used HSV space instead of RGB space because HSV ranges are used for chroma keying. In HSV space, the hue components of images are most likely to be similar. Unlike RGB, HSV can separate luma from chroma. The main characteristics of a color are better understood through the hue, value and saturation components, in that order. Saturation is the amount of gray in the color, from 0 to 100 percent(high saturation means the color is very apparent, low means it's washed out). Value works in conjunction with saturation and describes the brightness or intensity of the color, from 0–100 percent, where 0 is completely black, and 100 is the brightest and reveals the most color. Hue is the color portion of the color model, expressed as a number from 0 to 360 degrees.

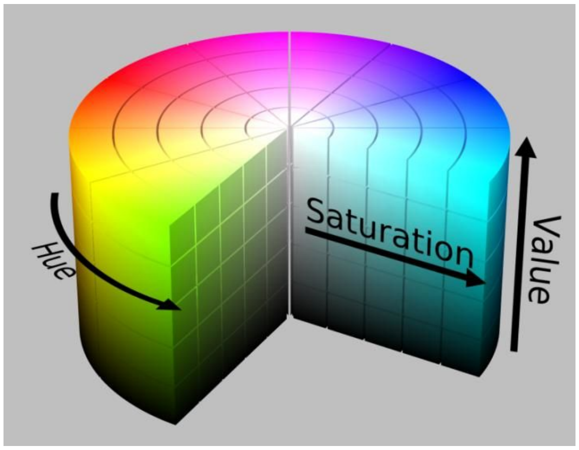


Figure 2: Diagram illustrating meaning of hue, saturation and value

It is much more feasible to adjust these HSV thresholds by looking at the image than to adjust thresholds in RGB space. RGB components are specific to an exact value for red, green and blue components and it's very difficult to guess those numbers. For example, in HSV space, you can guess the hue range for yellow to be between 60 and 120, while the RGB components are (251,253,124). To get rid of noise and ensure we are solely detecting the car we can adjust value and then saturation. Yellow is a 'washed out' color so it's half saturated and it's a bright color so it has a high value. The HSV diagram above shows how this combination of parameters leads to the color yellow in HSV space.

4.3 Object Detection - Premila

Modules: region_module
Files: region_module.v, final_proj.v

Now that we have converted to HSV space, we can leverage thresholding to do 'object detection' on our images. We created 9 regions in our traffic intersection (in figure below). This allows us to detect cars in each region based on hue and value ranges. We worked with blue and green cars in this project because most of the hot wheels we purchased were green and blue. We could have chosen to use other colors, but it's best if the shade of color we use is fairly saturated.

The cars are detected using a center of mass method, where we accumulate the sums of the x and y values per green/blue pixels and then divide by the number of pixels found, essentially taking an average of where a green or blue car is located. We can use this information to determine where the center point of the cars are and if there is a collision between a blue and green car in a given region. Because we are using this center of mass method we can only detect one car of each color in a given region because when we accumulate 'green' pixels, we are limiting our search to a specific region, so if there were two green cars we would end up getting the center point between those two cars.

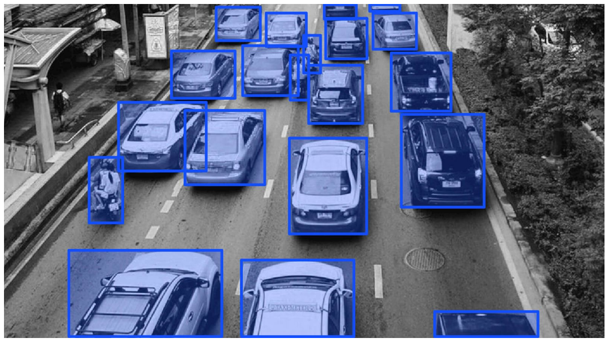


Figure 3: Image illustrating our object detection goal

Given more space in ZBT, we could have convolved the image with a template of a horizontal and vertical car. This would give us a single center point for each car in the image and then we would not need to limit our averaging method to specific regions. The issue is that this requires multiple line buffers in order to convolve, (about 20 line buffers the size of the image and one the size of the car). We also implemented erosion and dilation but did not include it in the pipeline because we were able to detect cars very well using only chroma thresholding and we didn't want to take up extra unnecessary space. Erosion and dilation are similar to the previous convolution method described. A 3x3 kernel is convolved with the image. For erosion, it's a kernel of all 0's and for dilation, all 1's. Basically we shift in one pixel at a time to 3 line buffers, two the size of the image and one the size of the kernel and then 'and' or 'or' the first 'x' elements of each line buffer. This result gives us the value of the current pixel we are operating on. This would help rid of noise and essentially do a form of blob tracking for the cars. Erosion is basically getting rid of pixels that are not really representing cars and are background noise, and dilation expands the real blobs back out so we can see them more easily. The template method described earlier is basically erosion, except our kernel is the size of the cars instead of a 3x3.

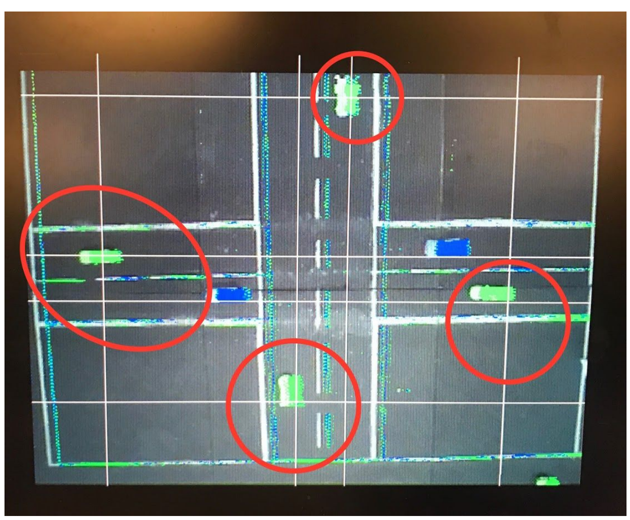


Figure 4: Illustration of cross hairs detecting cars

4.4 Traffic FSM Logic

Modules: traffic_fsm, led_controller
Files: traffic_fsm.v, led_controller.v

The main aim of our project was to have the traffic logic change based on which road (main road or side road) had a larger number of cars. The FMS works by checking consistently if the count on the other road is larger or smaller. If the main road has more cars, the traffic signal for the main road turns green and that of the side road turns red and vice versa.

However, pedestrians have the highest precedence. Thus, our pedestrian signal will cause a change ignoring everything happening on the road. If the push-to-walk button is pressed in one cycle, the lane that the pedestrian is in will be given precedence and will turn green on the next cycle. The FSM has the following states:

- ❏ MAIN_RED_SIDE_GREEN (default mode of traffic light if there are no cars or an equal number of cars) - 5 seconds
- ❏ MAIN_RED_SIDE_YELLOW - 2 seconds
- ❏ MAIN_GREEN_SIDE_RED -5 seconds
- ❏ MAIN_YELLOW_SIDE_RED - 2 seconds

Within each state, a check is made for the pedestrian signal while the countdown timer is running. In the states where either traffic light is on yellow, no comparisons of cars are made because it is a meta-state on it's way to a stable state. That is to say, when a traffic light is on yellow, it is definitely moving to red after the timer is up so there is no need to check for anything. When the timer runs out, the current cycle also completes comparison of cars and if necessary, moves to a next state.

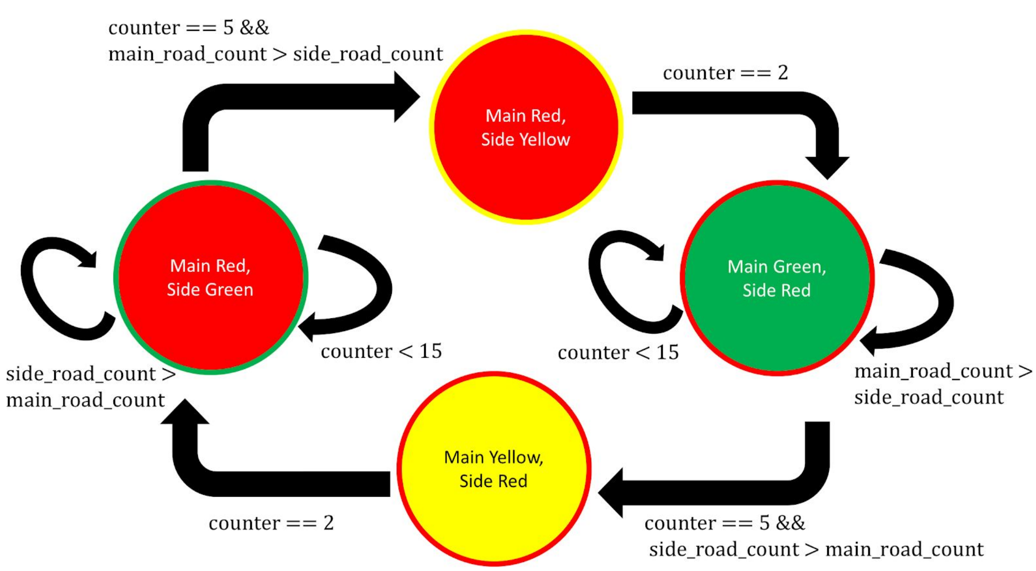


Figure 5: FSM illustrating operation of traffic lights

For easier debugging, we decoupled the output of the FSM from the individual signals. So the **traffic_fsm** module only outputs main_road_output as RED, BLUE or GREEN (these values are declared as constants in the params.v file and used universally). The **led_controller** module then takes those inputs and fans out the information into different signals, interpreting main_road_output == RED as main_road_red = ON, main_road_yellow = OFF, main_road_green = OFF, and same for the other colors.

4.5 LED Strip - Jessica

Modules: **led_strip**
File: led_strip.v

We used two LED strips to represent the outputs of our traffic signals on the "street" we had set up. The LED strip we used is an APA102 LED strip which uses a standard two wire SPI(serial peripheral interface) protocol. There are four inputs to the strip; the data wire, clock wire, ground and power (V_{CC}).

Figure 6: Wiring diagram of APA102 LED

To communicate with the FPGA labkit, we used the user1[1:0] pins - user1[0] drives clock and user1[1] drives data. We then grounded the ground pin and connected the V_{CC} pin to a wall power outlet. The datasheet stated that we needed to drive the LEDs with a clock of frequency 1MHz, a special start frame, and a specific end frame. The clock we use is generated from the MSB of a 6 bit counter which runs on a 65MHz clock and has a 50% duty cycle.

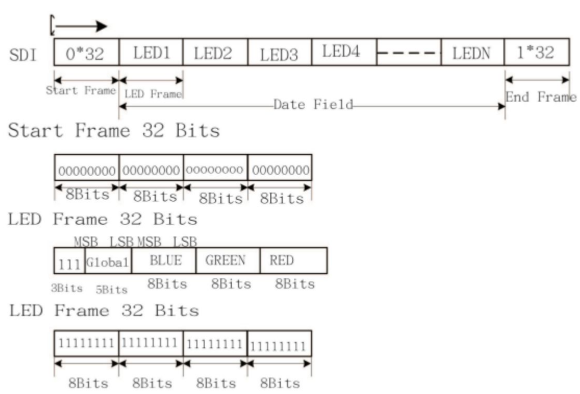


Figure 7: Diagram of APA102 LED protocol

The image above shows the breakdown of bits that are sent to the LEDs and what they mean. These bits are always sent in the form [32-bit START_FRAME, 32-bit for each LED, 32-bit END_FRAME]. On every rising edge of the clock, the value in the data register data bit is sent to the strip. Thus, after the end frame is sent, it is important to turn off the LED clock otherwise garbage data values will be sent across the strip.

For each LED , the "global" field within the frame is used to control the brightness so it can be tuned as desired. Since we were working on traffic lights, we focused on three colors - red, yellow (combination of red and green), and green. We had a 30 strip LED and needed only 18 LEDs (6 LEDs for each color) so the remaining 12 were sent blank (0,0,0 for R,G,B) frames.

Our **led_strip** module uses a state machine to send the frames. The four color frames we were using (red, yellow, green, blank) were pre-defined as registers so at each clock cycle, the index into the the register is updated to select a different bit from the current frame. The states are

- SEND_START_FRAME
- SEND_COLORED_FRAME
- SEND_BLANK_FRAME (used as filler because we are not using all the leds)
- SEND_END_FRAME
- READ_TRAFFIC_SIGNALS

The start and end frames are already defined in registers so we index into them just like the other colors. However, for the READ_TRAFFIC_SIGNALS state, we take input signals from the **led_controller** module. This will determine which LEDs are lit up in the SEND_COLORED_FRAME state. After this, blank frames are sent for the LEDs which are not used.

Each time, after the SEND_END_FRAME state is complete, the FSM goes to the READ_TRAFFIC_SIGNALS mode to check which color the traffic light currently has on. This is read into an array with the values [RED, YELLOW, GREEN]. The only possible values of this array are [1,0,0], [0,1,0] and [0,0,1]. If RED was 1, then only the strip allocated to the color red will be turned on in the SEND_COLORED_FRAME module. If RED was 0, then we send blank frames instead. We do same for the yellow and green LEDs. Afterwards, there are LEDs that we don't use so we fill them with blank frames.

We had to frequently update the LEDs because they were supposed to change in real time with the traffic signals. This means that as long as the system is running, the LED strip is constantly reading the traffic signals and updating colors. Thankfully, it does this so quickly that the human eye cannot perceive the individual signals being sent.

4.6 Collision Detection - Jessica

Modules: collision_detector, calc_ambulance_params, get_amb_xy
Files: collision_detector.v

After the locations of the cars are detected and passed over from the image processing module, the values are analysed to determine if any pair of cars has collided. The (x,y) values of the top-left corner, as well as the width and height of the car, are sent into the collision_detector module which checks for overlap of values and interprets them as a collision. All these values are used to determine the top and right limits of the car. There is a check for an overlap in coordinates and if there is an overlap, the is_collision signal is turned on.

If there is a collision, the upper, lower, left and right limits of the cars involved in the accident are determined and sent to the calc_am. In a real life situation, this should be enough information for the ambulance to know where to go. However, because we were simulating the arrival of the ambulance in our visualization, we had to determine what direction the ambulance should appear from. Using the four directional limits computed from the collision, the ambulance determines what direction to come from in the **calc_ambulance_params** and then what direction to move towards in the **get_amb_xy** module. When the is_collision variable is TRUE, the ambulance calculations are made; the ambulance appears on the screen and starts moving towards the area where the collision occurred. The ambulance is treated like a car for the purposes of drawing on the screen, except that it reads the pixel bits from a different coe file if the is_ambulance variable is set to TRUE whenever it needs to be rendered on the screen.

4.7 VGA Output - Jessica

Modules: visualization, draw_car, draw_street, draw_traffic_light, xvga
Files: BMPtoCOE.m, visualization.v

The VGA module runs on a 65MHz clock. Since there were many different components that we needed to ensure were working, we started out by drawing a street on the VGA, using colored rectangular blobs to represent cars, and drawing out traffic lights to test the traffic FSM. All the pixel by pixel calculations are done in the **visualization** module and in the main module, these pixels are passed into the **xvga** module. To keep the code modular, we used different modules for each purpose.

Before we transitioned to using real life images from coe files, we drew out traffic lights as seen in the image below.

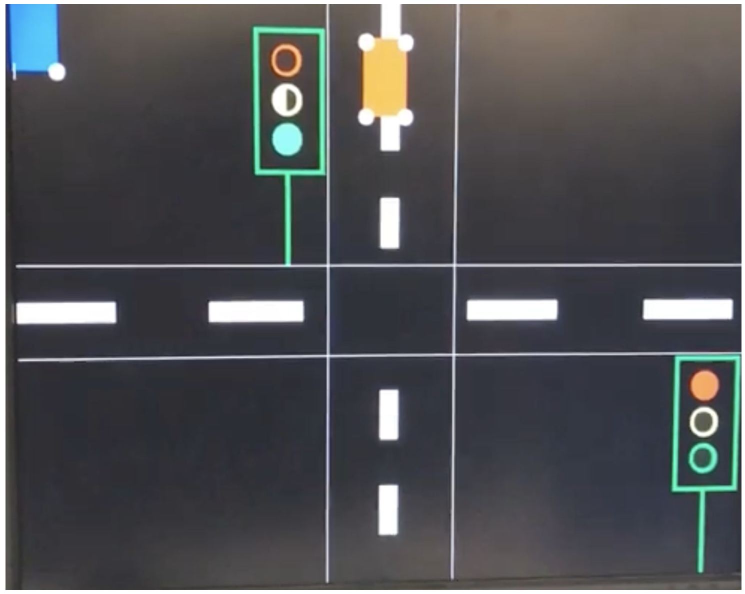


Figure 8: Initial drawings used for testing and FSM functionality verification

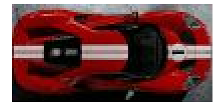
After we had adequately tested collisions and stress-tested the traffic light FSM, we updated the VGA output to use real-life images.

To draw images of the car, we took a .JPG image of a car, and used GIMP (a graphics editor pre-installed on the Athena) to scale it down and export it to .BMP format with only R,G and B components (no alpha). Then, using the BMPtoCOE.m file, we scaled the .bmp down to an 8 bit bitmap. However, the VGA monitor needs R,G and B values. The 8 bit bitmap includes a table which specifies the rgb values for each of the 8 bits in the image. So, each pixel in the image's coe file is represented by one byte, and that byte is an index into a table where each index specifies an R,G and B value separately. Thus, we loaded the image's pixel values as well as the R,G and B values which those indices map to. The image pixels (stored as indices) measure 8 bits in width by (image_width * image_height) in length and are loaded into a coe file. This coe file is loaded into ISE as a block of ROM memory which can be addressed into during image rendering on the VGA monitor.

In order to make good use of memory, we came up with different addressing equations to read out pixels from the memory block to obtain different orientations. For example, the original image we loaded for cars was oriented horizontally as seen in the image below.

Figure 9: JPEG image of car used for visualization

However, if you want to rotate the image without changing the x and y input, you can achieve this by computing a different address. The table below illustrates the different directions and how the address calculation is done for each image using only one block of memory.

Orientation, Direction	Equation	Resulting Image
Horizontal, facing right (original)	address = (hcount - x) + (vcount - y) * WIDTH;	
Horizontal, facing left	address = (WIDTH - (hcount - x)) + (vcount - y) * WIDTH;	
Vertical, facing up	address = (HEIGHT - (vcount - y)) + (WIDTH - (hcount - x)) * HEIGHT;	
Vertical, facing down	address = (vcount - y) + (hcount-x)*HEIGHT;	

In the .COE file, the original car color is red, but when the program is running, any color car may be displayed. This is achieved by thresholding on where the red concentration is high in the image and replacing that with whatever color of the car on the screen. Using the color pointer in GIMP, we played around with the different shades of red in the image and realized that the color of the car can be changed by replacing any region with a concentration of RED > 60 and BLUE < 50 and GREEN < 50 with a new color. Below are examples of Figure 8 shaded with new colors (blue and green).

Figure 10: Figure 8 colored green

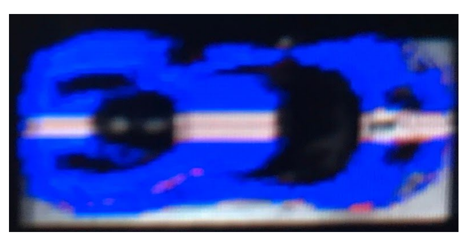


Figure 11: Figure 8 colored blue

The traffic lights are also implemented with the same idea. The original image of the traffic light looks like this:

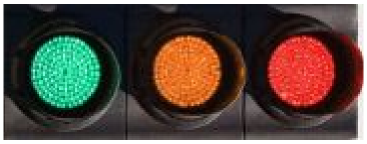


Figure 12: JPEG image of traffic lights used in visualization

We use a similar method as above to obtain different orientations of the traffic light. For our project, the traffic light signal on the screen only shows one color at a time, as is the case in real life. In order to accomplish this, we take signal outputs from the traffic FSM logic and gray out the signals that should be off using thresholding. For example, if the red light should be on, that means yellow and green should be off. Using thresholding, we determine that an area with GREEN > 150 is a green region and replace it with a gray pixel, and do similarly for yellow. The desired result is seen below.

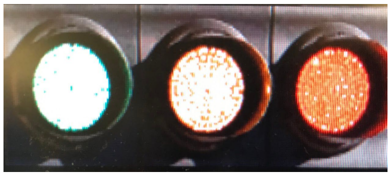


Figure 13: Traffic lights on red

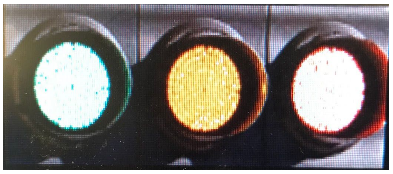


Figure 14: Traffic lights on yellow

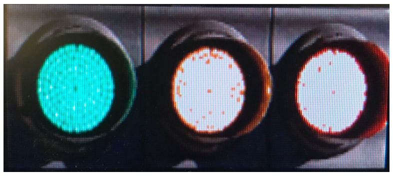


Figure 15: Traffic lights on green

4.8 Audio - Jessica

Modules: (All provided by staff - modified recorder) sound, recorder
Files: sound.v, WAVtoCOE.m

When a collision occurs, a signal is sent to the ambulance with information about where on the street the accident occured so that the ambulance drives onto the scene. To make the ambulance which appears on the screen more real, we added audio while it is in motion. We did this by converting an .mp3 file to a .wav file using an online converter.
We took the staff code which was used to playback a recorded audio and instead replaced the audio with the discrete sine wave values we had generated from the .wav file. After the conversion to .wav, we used the WAVtoCOE.m to write the discrete sine wave values into a coe file which is loaded as a block of ROM. The online converter produces unsigned integer (only positive numbers) values which shift the sine wave up and center it around 128. Thus, in the **recorder** module, a signed register is used to offset each of the discrete values by 128 to produce a value that is centered about 0. These values then drive the speakers to produce the siren sound.

4.9 Video Playback Module - Jessica

Modules: write_to_zbt, read_from_zbt, video
Files: video.v

When accidents occur, we would like to playback the accidents to determine the driver who caused them. Thus, we implemented a video playback module which worked by going across the screen and recording each pixels value into ZBT memory. On the labkit there are two ZBT banks which each have 512, 000 lines of 36 bits available for storage. The NTSC camera used the ZBT 0 bank so the video playback used ZBT 1 bank.

Each image on the screen is represented by a pixel. Since each image on the VGA monitor is 1024 wide * 768 tall, we need to store 1,024 *768 = 786,432 pixels. The VGA monitor takes in a 24-bit pixel value - (8R, 8G, 8B).

This means that we need to store (786,432 pixels * 24 $\frac{bits}{pixel}$)= 18,874,368 bits in ZBT memory for each image. To simplify the Mathematics, we considered pixels and not bits. Each ZBT address can hold 36 bits on one line. Instead of storing only one pixel (24 bits) per line and wasting the remaining 8 bits, we decided to store 18 bits for each pixel so that we can store 2 pixels on 1 line of ZBT memory.

This enables us to reason in terms of pixels and lines. We have 512,000 lines so we can store 512,000 lines * 2 $\frac{pixels}{line}$ = 1,024,000 pixels. The goal here is to store MULTIPLE IMAGES (since we want to show a video) as time progresses. If we were to store the entire image on the screen into ZBT, that would require 768,432 pixels for one image on screen which means we can store $\frac{1{,}024{,}000\ pixels}{768{,}432\ \frac{pixels}{frame}}$ = 1.33 frames. However, we need about 20 to 60 frames. Thus, we decided to downsample the images we were reading from the frames by reading every 4th vertical pixel and every 4th horizontal pixel.

This will shrink down our image size to $\frac{1024}{4}$ wide * $\frac{768}{4}$ tall = 256 wide * 192 tall BUT we will be able to store more frames.

Now, we are writing 256*192 = 49,152 pixels and there are 2 $\frac{pixels}{line}$, meaning we are using $\frac{49{,}152\ pixels}{2\ \frac{pixels}{line}}$ = 24, 576 lines for each frame. We have 512,000 lines available so we can store $\frac{512{,}000\ lines}{24{,}576\ \frac{lines}{frame}}$ ≈ 20 frames.

Thus, we implemented video playback to record 20 frames, recording a new frame each second. To illustrate how the playback works, consider the timeline for a minute. From 0 to 20 seconds, we record 20 frames, recording one frame per second. From 20 to 40 seconds, we overwrite those 20 frames. From 40 to 60 seconds, we overwrite those frames.

Our implementation accounts for the two pixels per line reading; on each address line, we read the first 18 bits, then move to read the next 18 bits for another pixel (on the same address line). We then increase the address and repeat. While increasing the address, we had to keep track of which frames we were reading because after reading the 20 frames that were written, there is noise in the .833 unwritten frames. Hence, we had to cycle back to address 0 after reading the 20th frame.

One tricky situation we had to deal with was handling a user request to watch a video in the middle of writing a frame. We needed to finish writing that frame so that we don't store partially written frames in memory. So, we designed the write module to detect the rising edge of the read_control variable (which signals a switch to video), completes the current frame it is writing, and sends a now_read signal from the **write_to_zbt** module to the **read_from_zbt** module. This now_read signal is then detected in the **read_from_zbt** module and the display switches to show the video playback.

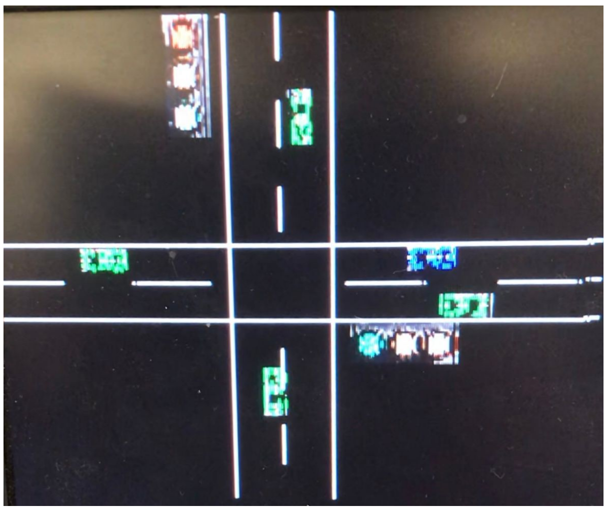


Figure 16: Downsampled video playback on monitor (cropped to focus on image)

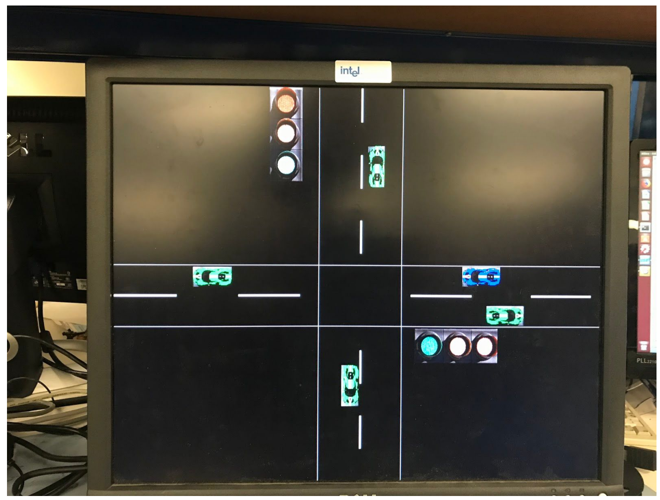


Figure 17: Actual display on VGA monitor

5 Testing and Debugging

Image Processing: Color thresholding

To test color thresholding, we adjusted the upper and lower bounds of the hue and value ranges using the buttons. We worked on hue, then value, and then saturation. This order is important because separating the hue component is easiest to perceive visually and has the most significant effect(it's the actual 'color' component of the image). Adjusting the value helps get rid of noise in the background and corners due to weird lighting or other objects(this is done in conjunction with saturation). We didn't adjust saturation for the cars because that component didn't make much difference. We tweaked the upper and lower bounds until we got ranges to distinguish the object (in our case, car) from the rest of the image. Once we got the best range, we parameterized the values we found for the objects we were working with and used them in the object detection algorithm.

Image Processing: Center of Mass

To test the algorithm for finding the center of mass of objects, we created crosshairs that centered on the cars that had been detected. We spent several hours debugging the crosshairs jumping around randomly, and not centering on any of the cars as we had expected. It turns out that we were reading the color of the vga_pixel variable, which was the output being sent to ZBT. The issue was that the accumulation was being done on pixels that the ZBT was outputting. Instead, we should have been using the output of ZBT, which is vr_pixel, to make the comparison.

Another major issue we ran into was dealing with noise from other parts of the screen. The crosshairs were being generated somewhat correctly but weren't consistent and partial correctness was certainly not enough for performing calculations based on these averages. Having many "pixel" variables was difficult to keep track of. We originally blacked out the pixels on the monitor that were not part of the camera output, so that we wouldn't have to worry about random noise from those parts of the screen. When calculating averages, we were using vr_pixel, which was the output from ZBT and NOT the camera output pixel. The bug was even harder to find because we had blacked out the part of the screen that had noise and that significantly affected our average values.

Traffic FSM

Our default mode for debugging and testing was using the LED display as well as the hex display. For example, when we implemented the traffic FSM, we segmented the LED display from 0-2 and from 4-6 to show the changing signals. We had the (traffic light - LED) outputs red-0, yellow-1, green-2 for the main road, and then for the side road, we had (traffic light - LED) outputs red -3, yellow-4, green-5.

So when that signal was off or on, we knew whether the FSM was working or not. We used switches to control the number of cars on each road: switch[7:6] was used for the main road count and switch[1:0] for the side road count. Thus, after we set the main_road_count < side_road_count using the switches, we would observe the impact of the change on the traffic lights via the LEDs. After incorporating pedestrian signals, we used the LEDs to once again view the different traffic signals and how they responded to different pedestrian signals.

Visualization

For the visualization, it was very convenient to have the VGA monitor showing what was being sent to it. That way, we could tell that something was off and saw the problem more easily. Addressing was definitely the most difficult to test because one couldn't tell what exactly made the image off but it just seemed off. Thus, we invested more time into drawing many different images and reasoned about the projection of points extensively to minimize error. We also chose to use a car that had a white strip in the middle so that it could guide us on what was wrong in the image as opposed to a solid colored car.

Collisions

Collisions were easiest to test with visual information. Since we wanted to be able to test all directions very easily, we drew a blob which was supposed to represent a car and connected the (x,y) top left corner of the blob to a mouse input. This allowed us to move the "car" blob around easily and simulate collisions from left, right, top and bottom and ensure that the is_collision variable was only triggered for collisions. The visual information also helped us to determine if the ambulance had detected the correct location of the collision because it would move towards the left, right, upwards or downwards based on the coordinates of the cars involved in the accident.

Audio

The audio was pretty straightforward to test; we had a pair of speakers that we drove with sound and could tell easily if the sound was off. While debugging, we found it useful to visualize the sine wave values that had been converted to COE file in Matlab because for a very long time the sound was just horrible. After visualizing the sound wave, we found that the online audio converter produced only uint(unsigned integer) values for the wave so it was centered around 128. Without visualizing the sine wave and understanding what was being loaded into the block memory, it would have been almost impossible to determine the problem since the audio module had a lot of different pieces.

Video Playback

The video playback interfacing with ZBT was one of the most difficult things to debug. Unfortunately, it is not as easy to see what is happening in the memory because you can't just drive the signal to an LED or hex display. Also, we are reading pixels so you also can't display ALL of them as they are rendered to the screen.

Since writing and reading both have very specific protocols that involve a lot of nuances, it was difficult to tell what was going wrong. The visual feedback was helpful but for a very long time, the screen was blank (which was due to some default setup values which hadn't been set to 0 in the main module). Thus, we had to do a lot of isolation of all the connected modules and set hardcoded values until we realized that the ZBT was actually not working after about 4 days of debugging.

After the ZBT started working, we tested with multiple images. First, with a screen of color red and to see the downsampled screen entirely red, then with a checkerboard pattern to see the downsampled checkerboard screen. Finally, we used the image of a car moving across the screen and verified that the downsampled video screen replicated that motion correctly. We then moved on to testing the visualization screen.

6 Challenges

Color Thresholding

We needed to threshold on hue *and* value, not just hue. This helped tremendously when trying to get the cars to be distinct on the screen. At first, it was impossible to get rid of all the noise because we were only adjusting hue. However, we found that hue is the most important component when identifying the general color, but does not help account for other objects with similar saturation/brightness levels. We were able to adjust value after optimizing the hue bounds and it helped get rid of most noise in the other parts of the image that may have been due to lighting or other objects with similar saturation levels to our object of interest.

Using the divider module to perform multiple divisions quickly

Another challenge was understanding the divider module and knowing how to use it to do many calculations at once. We needed to calculate the average for 6-10 cars every frame in order for the right information to be passed to the **visualization** and **traffic_fsm** modules. The division part of calculating the average, however, needed to happen after all the sums were calculated. This is because we are getting pixels one at a time, row by row, meaning we don't necessarily have the sums for each car in an orderly manner. We have a camera output size on the monitor of about 700x525 pixels. Thus, we used the time that would be used to render the remaining pixels on the monitor - (1024-700)x(768-525) - to perform divisions. We asserted the start signal as soon as we reach the end of the camera output, and have a state machine that runs through each division, one after the other. It is important to note that the start signal is held, so we have to deassert it on the next clock cycle. Otherwise, the average will be calculated over and over, causing the crosshairs to jump around.

ZBT

The hardest thing to debug was the ZBT interface because it was heavily reliant on staff code. Since ZBT was a stretch goal, we implemented everything before we worked on it. When we implemented ZBT, we were under the impression that it was staff code so it would work seamlessly. Unfortunately, there were a number of bugs in the way the default value settings which were found only after everything else in the code was commented out. ZBT works such that if any of the ram (A,B,C,D) values are driven with a 1, it is confused for both reads and writes and does nothing. Hence, all those values need to be defaulted to zero. ZBT was very difficult to debug but it was a great learning experience in dealing with memory.

Changing integration protocol due to object detection issues

Our original idea for integration was for the image processing module to send a start_frame pulse, iterate through the entire frame, and then send an end_frame pulse when it was done scanning through an image on the screen. While scanning over the image, if a car is detected, then a new_car_detected pulse is sent along with the new_car_x and new_car_y indicating the position of the new car on the screen.

Unfortunately, debugging the integration of both components of the project took a very long time and we did not have the bandwidth to implement that protocol. Instead, we decided to create instances of multiple cars (13) which would not be showed on the screen until their x and y values became non zero. We had to limit the number of cars to 13 because we had memory issues otherwise.

Effects of long wires for LEDs

When we tested our LED Strip and it was working, we assumed that it would be fine when integrated into the system which was true. However, when we created our physical set-up on the street, the LEDs had to be placed at a farther distance from the FPGA. We realized late that using longer wires for the LED strips caused very weird behaviour of glitchy blue signals instead of the colors that we had previously seen. We needed to frequently update our traffic signals so we shortened the wire which solved the problem.

Audio

It was much easier to load a COE file at the desired frequency and pitch that we wanted than to create the desired ambulance siren with a Direct Digital Synthesizer.

7 Design Decisions

Modularizing code

One thing we did early on which was very helpful throughout the project was to create a separate project for each new subsystem and test it working separately before moving it into the **final_project** module. This made it very easy to test things when we had put all the code together. If we were driving the LEDs in the **final_project** module and they were flickering, we could easily switch to the project which contained only the led implementation and test the code whose functionality we were certain of. This helped us to determine if the malfunction resulted from the integration, a hardware malfunction, or some other factor. It also helped us to keep things neatly segmented and narrow down sources of bugs when we saw unexpected behaviour.

Object Detection Algorithm

There were a few different ways we could have designed the algorithm for detecting cars in the image. We could have used a ‘center of mass’ method, a counting pixels method, or convolved with a template. The counting pixels method would involve hardcoding the size of the cars that should be known beforehand and then looking for a ‘1’ in HSV space to start counting until we saw a zero again. We would do this for however many rows there were for a car and count that as one car. The tricky thing is that it’s very difficult to know the exact number of pixels of a car and for this method to work, we need to have a pretty accurate number. This method also seemed like a hack and would not work well with larger scale versions of this project.

Our next idea was to create a template image of a horizontal and vertical car and convolve it with our image from the camera. This is ultimately the best method, would work on large scale versions of the project and would allow more complexity in object detection. Convolving the image with a template image containing a car of the same size would result in an image with a single pixel for each car. The issue is convolution in verilog requires line buffers to be the same size of the image and we didn’t have the space in memory for this. In addition, creating a template the exact size of the car may have been tricky because that heavily depends on the camera angle and distance from the road.

Our last idea, which we ended up implementing, was to calculate the center of mass of each car and pass that information on to the other modules. This method worked very well and was intuitive. The downside of this method is that it forced us to use cars of different colors. We worked around this by compartmentalizing the road into regions and only searching for chroma keyed pixels in certain regions. This is a limitation of our project and given more time and space, we would recommend using the convolution with a template method.

Filtering

When deciding how to best calculate the center of mass, we had to take noise into consideration. We used a median filtering method instead of a mean filtering method in order to avoid noise from random specks on the monitor. A median filter avoids this by only taking the center value and ignoring random noise. Another method of filtering that would solve this is erosion/dilation. This two step process, which we implemented but did not use, would help distinguish actual cars from random noise elsewhere on the screen. Our color thresholding ended up working really well and our cars were very distinguishable from other parts of the screen, so we didn't need to use any other filtering methods. If we had different lighting conditions and other potential sources of noise, these methods would definitely be necessary.

Memory Tradeoffs

Since there were a lot of different components of our project, we had to optimize for memory in every way possible. Rather than loading multiple images oriented in different directions and reading them out for the car, we decided to use only one image and use addressing to index in the image in different ways, to obtain different orientations and directions of the same image. This helped to create more space for loading other images for other uses instead. We would have liked to incorporate angled turns into the indexing but the timing of the pixel motion across the screen made it difficult to do so. Also, instead of loading different color maps, we chose to use thresholding to replace the colors of the images we had loaded in block ROM.

Demonstrating motion of cars from street to VGA monitor

When we were trying to replicate what was happening on the street on the VGA monitor, we spent time thinking about how we wanted to move the cars and if we would pass information about the speed of cars between modules. After considerable deliberation, we realized that at any point in time, we the position of the car was known so a car that would be moving fast would have larger displacement between two timestamps and a slower car would have smaller displacement between those timestamps; this would be captured by the camera when it detects the position of the cars so that wasn't something we needed to worry about.

Scaling

The field of view of the camera and the screen size were not proportional, so we had to make a decision as to whether we wanted to multiply the values by a factor to shift them to the correct location on the screen or wanted to shift them manually. After considering the impact on our system as a whole, we chose to modify the x and y values that were passed from the image processing module to the visualization module by adding and subtracting offsets. We chose not to use multiplication for scaling the images because that would introduce pipelining delays. We also made the decision of standardizing the size of the car on the screen (regardless of its size on the street). Further work can be done to improve this design and reflect the size of the car on the street.

8 Reflections

Premila

This has been the most exciting thing I've worked on at MIT. I learned so much in such a short amount of time about Verilog, image processing, debugging, and about working on a team. This project has sparked my interest in firmware development and image processing. My debugging skills improved so much and I gained the ability to look for the root of a bug, step by step, whether it was by outputting signals to the led or hex display or by breaking down the code into super small pieces and testing each and every one.

I learned the important principles of verilog and now have a solid understanding of how to write an algorithm in verilog using the pipelining methodology. I also learned how to modularize my code and put it together carefully, testing each piece before doing so. The most interesting part of this was the fact that everything I did was on a pixel by pixel basis, and I had to write all of my algorithms to match this.

The image processing part was the coolest thing I worked on in this project by far. I gained insight into working with camera output and sending that information one pixel at a time to ZBT. I enjoyed learning about the different color spaces and how to work in each of them as well as their benefits and downsides. This project definitely inspired me to think about future object detection work as well as potentially bringing machine learning principles into the algorithm.

Working with Jessica on this project was a very fun experience. We definitely had our disagreements and had to compromise when making design decisions, but I learned a ton from her debugging skills and algorithm knowledge as well as from her work ethic and organizational skills. We were able to successfully complete the project because we worked together so well and it would have been much more difficult otherwise.

Jessica

This project was a great learning experience which I learned many valuable lessons from. I came into the project seeking to learn how many different things work as much as possible; in other words, to cover much more breadth than depth. Thus, although I didn't work on the image processing directly, I gained experience working extensively with the VGA monitor, audio, ZBT, LEDs, etc.

While working on the project, my biggest takeaway was learning how to effectively debug a system. Sometimes, I would design something and when I didn't have the expected behavior, I found it difficult tracing the source of the issue. There are multiple ways to debug, but this class helped me gain the skill of knowing the most efficient way to debug a particular issue. Sometimes you're better of using visual cues but other times a display of the values makes more sense. Other times, using the switch to control multiple states in real time is the best way to go.

I did a lot of work related to reading from block memory and designing addresses to produce specific results so I learned more in-depth how the Core Generator works. I also learned how to think creatively as an engineer and to do more with less because the BRAM space was limited. This comes at a cost; your ability to create a solution to a limitation is highly dependent on the depth of your understanding of the system. So for every work around created, I had to invest more time learning.

Because we designed our project in a very modular way, one of the things I had to learn to balance was the trade off between having very modular and self-explanatory code versus the overhead cost associated with instantiating registers and passing them between multiple modules. This was especially crucial when we were adding new features because many things needed to be passed between submodules.

In addition, I learned a lot about reading documentation and understanding other people's code and how to alter it to achieve a desired outcome, which I think is an important skill not only for digital systems but for engineers.

Overall, this project was a great learning and growth experience for me. I learned a lot from working with Premila; though we had stressful times, we learned how to speak each other's language which helped us especially when we were headed towards the finish line.

9 Conclusion

This project was overall very successful. We hit all of our baseline and expected goals, and a few of our stretch goals. We predicted and thought through many of the issues we would run into at the beginning of the project, but not all of them. Thus, we had to think quickly and adapt every time we found a major bug or flaw in our design. There are a few key takeaways that we think would be helpful for future students to know.

We started working very early on our project and we believe this helped greatly and allowed us to pivot without fear of running out of time. We also received a great deal of assistance and feedback from the staff early because most of the time there was no one else around when we were in lab.

The biggest lesson we learned is that when you are stressed and tired, your attention to detail suffers greatly. On the Saturday before the project was due, we had spent so much time working in lab that we made the mistake of writing the boolean statement " if (x+60 > 0)". We were lucky enough to have our LA, Mike Wang, help us catch this error because we were both too exhausted from looking through hundreds of lines of code unable to understand what was going wrong.

We also learned the importance of taking things one step at a time. We started integration by combining both our labkit files by copying and pasting, and came to deeply regret it after spending a day and half debugging. In hindsight, it was naive for us to assume that blackboxing integration in one step would work. We ended up reverting everything and moving pieces of code in one step at a time, making sure things still worked at every step along the way. This was much easier to debug and we had a better understanding of what was going on.

The last key lesson we took away is to look at the bigger picture during every step of the project. It's very important to have a clear, detailed block diagram and to outline each module before coding anything. There were several times during the image processing component of the project when we had to re-do algorithms differently. We were trained to adapt and be flexible throughout the entire project and had to backtrack several times because we didn't think through the next steps. This made integration challenging because I (Premila) changed my implementation many times because it was hard to predict how things would play out. This affected the way I was passing information to Jessica, which would often involve redoing work she had done with the assumption that I was doing something different. Upon reflection, this was all all part of the process and contributed greatly to our learning experience as a whole.

10 Acknowledgements

We spent many hours in lab while working on our project and are very grateful to all the staff who pushed through many challenges with us. We are grateful to our mentor, Diana Wofk, who was very invested in our project and helped us creatively brainstorm whenever we needed to navigate roadblocks. We really appreciate your assistance scoping our project and constantly giving feedback as we progressed.

We are also grateful to Professor Steinmeyer for his help debugging many different parts of our project and equipping us with the skill of probing to find bugs.

Thanks to Professor Hom for helping work through many painful bugs over the course of this project! It was extremely helpful when lab was opened on Saturdays and he would sit through hours of debugging, helping to get us to the finish line.

Our LA, Mike Wang, was of incredible help throughout the project; he helped us solve many tough bugs and plan the design for the image processing algorithms which took a lot of time and careful thought. We appreciate your patience and willingness to work with us!

Thank you to the rest of the TAs and staff for encouraging us to do our best and helping us learn so much this semester.

Appendix I - System Usage

User Inputs

Switches

Signal	**Parameter**	**Function**
switch[0]	read_control	Switches to video playback mode when turned on, shows visualization otherwise
switch[1]	-	Selects between test bar periods; these are stored to ZBT during blanking periods
switch[2]	-	Used for testing the NTSC decoder
switch[3]	-	Selects between display of NTSC video and test bars
switch[4]	ntsc_to_vga	Shows camera output when turned on, displays visualization otherwise

Buttons

BUTTON	**Parameter**	**Function**
up	ped_cross_up	Used to cross road vertically - moving north
down	ped_cross_down	Used to cross road vertically-moving south
left	ped_cross_left	Used to cross road horizontally-moving west
right	ped_cross_right	Used to cross road horizontally-moving east
enter	system_reset	Resets the entire system

Hex Display Information

The hex display shows the current state of our traffic FSM as well as the number of cars on the main road and the number of cars on the side road in the form {traffic_fsm_state, main_road_count, side_road_count}.

Value	**FSM State**
1	main_red_side_green
2	main_red_side_yellow
3	main_green_side_red
4	main_yellow_side_red

Instructions for using the system

1. Turn off all switches and turn off the FPGA.
2. Make sure that the LEDs, the NTSC camera and the labkit are all connected to power.
3. Program the labkit using the bitfile.
4. The system is by default in the MAIN_GREEN_SIDE_RED FSM. Based on the input from the camera about the cars on the different sides of the road, the traffic signals will change and these can be seen on the LED strips as well as on the VGA monitor.
5. To view what the camera is seeing, flip switch[4] on . To return to the visualization screen, flip switch[4] off.
6. To view video playback, flip switch[0] on. To return to visualization screen, flip switch[5] off.

Appendix II - Matlab Code

- BMP to COE Conversion in Matlab (BMPtoCOE.m)
- WAV to COE Conversion in Matlab (WAVtoCOE.m)

Appendix III - Verilog Source Files

- clock_divider.v
- collision_detector.v
- debounce.v
- display_16hex.v
- divider.v
- erosion.v
- erosion_shift.v
- final_proj.v
- hsv_threshold.v
- image_selector.v
- led_controller.v
- led_strip.v
- ntsc2zbt.v
- ntsc_decoder.v
- params.v
- region.v
- rgb2hsv.v
- sound.v
- video.v
- visualization.v
- ycrcb2rgb.v
- zbt_6111.v

```
%% How to use this file
%Notice how %% divides up sections?  If you hit ctrl+enter, then MATLAB
%will execute all the lines within that section, but nothing else.  You can
%also navigate quickly through the file using ctrl+arrow_key
%% Getting 8 bit data
%When you store an 8 bit bitmap, things get a little complicated.  Now
%each pixel in the image only gets one 8 bit value.  But, you need to send
%the monitor an r,g, and b (each 8 bits long)!  How can this work?
%
%8 bit bitmaps include a table which specifies the rgb values for each of
%the 8 bits in the image.
%
%So each pixel is represented by one byte, and that byte is an index into a
%table where each index specifies an r, g, and b value separately.
%
%Because of this, now we need to load both the image and it's colormap.
 [picture color_table] = imread('car.bmp');

%% Displaying without the color table
%If we try to display the picture without the colormap, the image does not
%make sense
  figure
  image(picture)
  title('Per pixel values in 8 bit bitmap')

%% Displaying WITH the color table
%So to display the picture with the proper color table, we need to tell
%MATLAB to set its colormap to be in line with our colorbar.  The image
%quality is somewhat reduced compared to the 24 bit image, but not too bad.
  figure
  image(picture)
  colormap(color_table)   %This command tells MATLAB to use the image's color table
  colorbar                %This command tells MATLAB to draw the color table it is u
sing
  title('8 bit bitmap displayed using color table')

%% More about the color table
%The color table is in the format:
%
%color_table(color_index,1=r 2=g 3=b)
%
%So to get the r g b values for color index 3, we only need to say:
  disp('          r    g              b    for color 3 is:')
  disp(color_table(3,:))       %disp = print to console

%Although in the bitmap file the colors are indexed as 0-255 and each rgb
%value is an integer between 0-255, MATLAB images don't work like that, so
%MATLAB has automatically scaled them to be indexed 1-256 and to have a
%floating point value between 0 and 1.  To turn the floats into integer
%values between 0 and 256:

  color_table_8bit = uint8(round(256*color_table));

  disp('    r    g    b    for color 3 in integers is:')
  disp(color_table_8bit(3,:))

%Note that this doesn't fix the indexing (and it can't, since MATLAB won't
%let you have indexes below 1)

%another way to look at the color table is like this (don't worry about how
%to make this graph)
  figure
  stem3(color_table_8bit)
  set(gca,'XTick',1:3);
  set(gca,'YTick',[1,65,129,193,256]);
  set(gca,'YTickLabel',['  0';' 64';'128';'192';'255']);
  set(gca,'ZTick',[0,64,128,192,255]);

  xlabel('red = 1, green = 2, blue = 3')
  ylabel('color index')
  zlabel('value')
  title('Another way to see the color table')
```

```
%% Writing data to coe files for putting them on the fpga
%You can instantiate BRAMs to take their values from a file you feed them
%when you flash the FPGA.  You can use this technique to send them
%colortables, image data, anything.  Here's how to send the red component
%of the color table of the last example

  red = color_table(:,1);                   %grabs the red part of the colortable
  scaled_data = red*255;                    %scales the floats back to 0-255
  rounded_data = round(scaled_data);        %rounds them down
  data = dec2bin(rounded_data,8);           %convert the binary data to 8 bit binary #s

%open a file
  output_name = 'car_red.coe';
  file = fopen(output_name,'w');

%write the header info
  fprintf(file,'memory_initialization_radix=2;\n');
  fprintf(file,'memory_initialization_vector=\n');
  fclose(file);

%put commas in the data
  rowxcolumn = size(data);
  rows = rowxcolumn(1);
  columns = rowxcolumn(2);
  output = data;
  for i = 1:(rows-1)
      output(i,(columns+1)) = ',';
  end
  output(rows,(columns+1)) = ';';

%append the numeric values to the file
  dlmwrite(output_name,output,'-append','delimiter','', 'newline', 'pc');

% create color table for green (2)  and blue (3) and you're done!

%% Turning a 2D image into a 1D memory array
%The code above is all well and good for the color table, since it's 1-D
%(well, at least you can break it into 3 1-D arrays).  But what about a 2D
%array?  We need to turn it into a 1-D array:

  picture_size = size(picture);             %figure out how big the image is
  num_rows = picture_size(1);
  num_columns = picture_size(2);

  pixel_columns = zeros(picture_size(1)*picture_size(2),1,'uint8');  %pre-allocate a
 space for a new column vector

  for r = 1:num_rows
      for c = 1:num_columns
          pixel_columns((r-1)*num_columns+c) = picture(r,c);    %pixel# = (y*numColu
mns)+x
      end
  end

%so now pixel_columns is a column vector of the pixel values in the image
  rounded_data = round(pixel_columns);  %rounds them down
  data = dec2bin(rounded_data,8);           %convert the binary data to 8 bit binary #s

%open a file
  output_name = 'smaller_car.coe';
  file = fopen(output_name,'w');

%write the header info
  fprintf(file,'memory_initialization_radix=2;\n');
  fprintf(file,'memory_initialization_vector=\n');
  fclose(file);

%put commas in the data
  rowxcolumn = size(data);
  rows = rowxcolumn(1);
  columns = rowxcolumn(2);
  output = data;
  for i = 1:(rows-1)
```

```
        output(i,(columns+1)) = ',';
    end
    output(rows,(columns+1)) = ';';

%append the numeric values to the file
    dlmwrite(output_name,output,'-append','delimiter','', 'newline', 'pc');

%just to make sure that we're doing things correctly
    regen_picture = zeros(num_rows,num_columns,'uint8');
    for r = 1:num_rows
        for c = 1:num_columns
            regen_picture(r,c) = pixel_columns((r-1)*num_columns+c,1);
        end
    end

    figure
    subplot(121)
    image(picture)
    axis square
    colormap(color_table)
    colorbar
    title('Original Picture')

    subplot(122)
    image(regen_picture)
    axis square
    colormap(color_table)
    colorbar
    title('Regenerated Picture')
```

```
%% Read file
 [y,Fs] = audioread('amb.wav','native');
 a = audioplayer(y, Fs);
 play(a); %play the sound to make sure that it is the sound you are expecting
 plot(y); %plot the sine wave (helps debugging weird sound)

%% Writing data to coe files for putting them on the fpga
%You can instantiate BRAMs to take their values from a file you feed them
%when you flash the FPGA.  You can use this technique to send them
%colortables, image data, anything including sound bits.

 data = dec2bin(y,8);     %convert the decimal data to 8 bit binary #s

%open a file
 output_name = 'ambulance_siren.coe';
 file = fopen(output_name,'w');

%write the header info
 fprintf(file,'memory_initialization_radix=2;\n');
 fprintf(file,'memory_initialization_vector=\n');
 fclose(file);

%put commas in the data
 rowxcolumn = size(data);
 rows = rowxcolumn(1);
 columns = rowxcolumn(2);
 output = data;
 for i = 1:(rows-1)
     output(i,(columns+1)) = ',';
 end
 output(rows,(columns+1)) = ';';

%append the numeric values to the file
 dlmwrite(output_name,output,'-append','delimiter','', 'newline', 'pc');
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Jessica Quaye
// Create Date:    13:07:07 11/05/2018
// Design Name:
// Module Name:    divider
//////////////////////////////////////////////////////////////////////////////////

//assumes use of 65MHz clock, creates one second pulse each second
module clock_divider(
    input clk,
    output reg one_hz_enable );

         reg [25:0] counter = 26'b0;

         always @ (posedge clk) begin
                counter <= counter + 1;
                //generate 1hz signal
                if (counter == 26'd65_000_000 - 1)
                        begin
                                counter <= 0;
                                one_hz_enable <= 1;
                        end
                else one_hz_enable <= 0;
         end

endmodule

//creates 50% duty cycle 1mhz clock
module led_divider(
    input clk,
    output reg one_mhz_enable
    );

         reg [5:0] counter = 6'b0;

         always @ (posedge clk) begin
                counter <= counter + 1;
                if (counter[5] == 1'b1) begin //send a clock when the 2**6 bit is 1
                        one_mhz_enable <= 1;
                end

                else one_mhz_enable <= 0;
         end


endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Jessica Quaye
//
// Create Date:    21:21:17 11/18/2018
// Design Name:
// Module Name:    collision_detector
//////////////////////////////////////////////////////////////////////////////////
module collision_detector(
    input clk,
    input[10:0] car1_leftx,car2_leftx, street_leftx,
    input[10:0] car1_rightx,car2_rightx, street_rightx,
    input[9:0] car1_topy, car2_topy, street_topy,
    input[9:0] car1_bottomy, car2_bottomy, street_bottomy,
    output reg direction,
    output reg is_collision,
    output reg[10:0] leftx_threshold, rightx_threshold,
    output reg[9:0] uppery_threshold, lowery_threshold);

    `include "params.v"

    //TO DO
    always @ (posedge clk) begin
    //determine the ranges of the car to tell you if this will be VERTICAL or HORIZO
NTAL collision
    //determine if its on the HORIZONTAL street
    if ((street_topy <= car2_topy) && (car2_bottomy <= street_bottomy)) direction <=
 HORIZONTAL;
    else if ((street_leftx <= car2_leftx) && (car2_rightx <= street_rightx)) directi
on <= VERTICAL;

    //determine if a collision has occured
    if ((car1_leftx < car2_rightx) && (car1_rightx > car2_leftx) && (car1_topy < car
2_topy) && (car1_bottomy > car2_topy))
      begin
      is_collision <= TRUE;
      end

   else if ((car2_leftx < car1_rightx) && (car2_rightx > car1_leftx) && (car2_topy <
 car1_topy) && (car2_bottomy > car1_topy))
      begin
      is_collision <= TRUE;
      end

   else is_collision <= FALSE;


   //determine thresholds or stopping points for ambulances
   //determine y thresholds
   if (car2_topy < car1_topy) uppery_threshold <= car2_topy;
   else uppery_threshold <= car1_topy;

   if (car2_bottomy > car1_bottomy) lowery_threshold <= car2_bottomy;
   else lowery_threshold <= car1_bottomy;

   //determine x thresholds
   if (car2_leftx < car1_leftx) leftx_threshold <= car2_leftx;
   else leftx_threshold <= car1_leftx;

   if (car2_rightx > car1_rightx) rightx_threshold <= car2_rightx;
   else rightx_threshold <= car1_rightx;

   end //end always

endmodule


module calc_ambulance_params(input clk,
    input [10:0] leftx_threshold, rightx_threshold, street_leftx, street_rightx,
    input [9:0] uppery_threshold, lowery_threshold, street_topy, street_bottomy,
    input direction,
    input is_collision,
```

```
    output reg[1:0] ambulance_move_dir,
    output reg[10:0] ambulance_dest_x,
    output reg[9:0] ambulance_dest_y);

    `include "params.v"

    always @(posedge clk) begin
      if (direction == VERTICAL && is_collision == TRUE)
      begin
         if (lowery_threshold < street_topy) //ambulance move up to lower
         begin
            ambulance_move_dir <= MOVE_UP;
            ambulance_dest_y <= lowery_threshold;
         end

         else if (uppery_threshold > street_bottomy) //ambulance move down to upper
         begin
            ambulance_move_dir <= MOVE_DOWN;
            ambulance_dest_y <= uppery_threshold;
         end
      end

      if (direction == HORIZONTAL && is_collision == TRUE)
      begin
         if (leftx_threshold > street_rightx) //ambulance move left towards leftmost
         begin
            ambulance_move_dir <= MOVE_RIGHT;
            ambulance_dest_x <= leftx_threshold;
         end

         else if (rightx_threshold < street_leftx) // ambulance move right towards r
ightmost edge
         begin
            ambulance_move_dir <= MOVE_LEFT;
            ambulance_dest_x <= rightx_threshold;
         end
      end

    end //end always


endmodule

module get_amb_xy(input clk,
                   input one_hz_enable,
                   input is_collision,
                   input [1:0] ambulance_move_dir,
                   input [10:0]  ambulance_dest_x,
                   input [9:0] ambulance_dest_y,
                   output reg[10:0] ambulance_leftx,  ambulance_width,
                   output reg[9:0] ambulance_topy, ambulance_height);

   `include "params.v"
   reg amb_state = 0;

   always @(posedge clk)begin

         //determine if an ambulance is needed, ie, collision has occured
         if (is_collision == 1) begin
               begin
                  case(ambulance_move_dir)
                     MOVE_LEFT:
                     begin
                        ambulance_width <= 11'd100;
                        ambulance_height <= 10'd34;
                        ambulance_topy <= 10'd364;
                        if ((one_hz_enable == 1) && ((ambulance_leftx - CSPEED) > am
bulance_dest_x) ) ambulance_leftx <= ambulance_leftx - CSPEED;
                     end

                     MOVE_RIGHT:
                     begin
                        ambulance_width <= 11'd100;
```

```
                        ambulance_height <= 10'd34;
                        ambulance_topy <= 10'd364;
                        if ((one_hz_enable == 1) && ((ambulance_leftx + CSPEED) < am
bulance_dest_x - ambulance_width) ) ambulance_leftx <= ambulance_leftx + CSPEED;
                     end

                     MOVE_UP:
                     begin
                        ambulance_width <= 11'd34;
                        ambulance_height <= 10'd100;
                        ambulance_leftx <= 11'd520;
                        if ((one_hz_enable == 1) && ((ambulance_topy - CSPEED) > amb
ulance_dest_y) ) ambulance_topy <= ambulance_topy - CSPEED;
                     end

                     MOVE_DOWN:
                     begin
                        ambulance_width <= 11'd34;
                        ambulance_height <= 10'd100;
                        ambulance_leftx <= 11'd425;
                        if ((one_hz_enable == 1) && ((ambulance_topy + CSPEED) < (am
bulance_dest_y - ambulance_height)) ) ambulance_topy <= ambulance_topy + CSPEED;
                     end
                  default:;
                  endcase
               end //end of move amb state
         end //end if collision == 1

         else begin
                  case(ambulance_move_dir)
                     MOVE_LEFT:
                     begin
                        ambulance_width <= 11'd100;
                        ambulance_height <= 10'd34;
                        ambulance_leftx <= 11'd1024 - ambulance_width;
                        ambulance_topy <= 10'd364;
                     end

                     MOVE_RIGHT:
                     begin
                        ambulance_width <= 11'd100;
                        ambulance_height <= 10'd34;
                        ambulance_leftx <= 11'd0;
                        ambulance_topy <= 10'd364;
                     end

                     MOVE_UP:
                     begin
                        ambulance_width <= 11'd34;
                        ambulance_height <= 10'd100;
                        ambulance_leftx <= 11'd520;
                        ambulance_topy <= 10'd768 - ambulance_height;
                     end

                     MOVE_DOWN:
                     begin
                        ambulance_width <= 11'd34;
                        ambulance_height <= 10'd100;
                        ambulance_leftx <= 11'd425;
                        ambulance_topy <= 11'd0;
                     end
                  default:;
                  endcase
         end
       end //end always

endmodule
```

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce #(parameter DELAY=1000000)   // .01 sec with a 100Mhz clock
                (input reset, clock, noisy,
                 output reg clean);

   reg [19:0] count;
   reg new;

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

endmodule
```

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

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

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

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

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

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

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

   always @(posedge clock_27mhz)
     begin
        if (reset)
          begin
             count = 0;
             clock = 0;
          end
        else if (count == 26)
          begin
             clock = ~clock;
             count = 5'h00;
          end
        else
          count = count+1;
     end

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

   assign dreset = (reset_count != 0);

   assign disp_clock = ~clock;

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

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

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

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

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

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

      8'h02:
        begin
           // Initialize dot register (set all dots to zero)
           disp_ce_b <= 1'b0;
           disp_data_out <= 1'b0; // dot_index[0];
           if (dot_index == 639)
             state <= state+1;
           else
             dot_index <= dot_index+1;
        end

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

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

      8'h05:
```

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

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

      endcase

   always @ (data or char_index)
     case (char_index)
       4'h0:            nibble <= data[3:0];
       4'h1:            nibble <= data[7:4];
       4'h2:            nibble <= data[11:8];
       4'h3:            nibble <= data[15:12];
       4'h4:            nibble <= data[19:16];
       4'h5:            nibble <= data[23:20];
       4'h6:            nibble <= data[27:24];
       4'h7:            nibble <= data[31:28];
       4'h8:            nibble <= data[35:32];
       4'h9:            nibble <= data[39:36];
       4'hA:            nibble <= data[43:40];
       4'hB:            nibble <= data[47:44];
       4'hC:            nibble <= data[51:48];
       4'hD:            nibble <= data[55:52];
       4'hE:            nibble <= data[59:56];
       4'hF:            nibble <= data[63:60];
     endcase

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

endmodule
```

```
//Engineer: Premila Rowles
//Module name: divider.v

// The divider module divides one number by another. It
// produces a signal named "ready" when the quotient output
// is ready, and takes a signal named "start" to indicate
// the the input dividend and divider is ready.
// sign -- 0 for unsigned, 1 for twos complement

// It uses a simple restoring divide algorithm.
// http://en.wikipedia.org/wiki/Division_(digital)#Restoring_division

module divider #(parameter WIDTH = 24)
  (input clk, sign, start,
   input [WIDTH-1:0] dividend,
   input [WIDTH-1:0] divider,
   output reg [WIDTH-1:0] quotient,
   output [WIDTH-1:0] remainder,
   output ready);

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

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

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

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

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

         negative_output = sign &&
                           ((divider[WIDTH-1] && !dividend[WIDTH-1])
                            ||(!divider[WIDTH-1] && dividend[WIDTH-1]));
       end
      else if ( bit > 0 ) begin
         diff = dividend_copy - divider_copy;
         quotient_temp = quotient_temp << 1;
         if( !diff[WIDTH*2-1] ) begin
            dividend_copy = diff;
            quotient_temp[0] = 1'd1;
         end
         quotient = (!negative_output) ?
                    quotient_temp :
                    ~quotient_temp + 1'b1;
         divider_copy = divider_copy >> 1;
         bit = bit - 1'b1;
      end
   end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Kevin Zheng Class of 2012
//           Dept of Electrical Engineering &  Computer Science
//
// Create Date:    18:45:01 11/10/2010
// Design Name:
// Module Name:    erosion
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module erosion(
                input wire clock,
                input wire reset,
                input reg a8,
                input reg a7,
                input reg a6,
                input reg a5,
                input reg a4,
                input reg a3,
                input reg a2,
                input reg a1,
                input reg a0,
                input binarized_value,
                input reg [19:0] count,
                input reg [23:0] pixel_value,
                output reg pixel_eroded,
                output reg frame_end);

                reg [19:0] num_bits_per_frame; //345600


                always @ (posedge clock) begin

                        //output if frame ended if we have counted total number of p
ixels in a frame

                        if (count > 345600) frame_end <= 1;


                        //pixel_eroded will be the first 3 elements of each line buf
fer to get a total of
                        //9 elements, we can logically 'and' these because the kerne
l is all 0's
                        pixel_eroded <= (a8 & a7 & a6 & a5 & a4 & a3 & a2
                        & a1 & a0) ? 24'hFF0000 : pixel_value;//and 9 elements


                end

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Kevin Zheng Class of 2012
//           Dept of Electrical Engineering &  Computer Science
//
// Create Date:    18:45:01 11/10/2010
// Design Name:
// Module Name:    erosion
////////////////////////////////////////////////////////////////////////////////
module erosion_shift(
                input wire clock,
                input wire reset,
                input reg binarized_value,
                input reg  [23:0] pixel_value,
                output reg [19:0] count_out,
                output reg [23:0] pixel_out,

                output reg  a8,
                output reg  a7,
                output reg  a6,
                output reg  a5,
                output reg  a4,
                output reg  a3,
                output reg  a2,
                output reg  a1,
                output reg  a0);

                reg buffer_one [699:0];
                reg buffer_two [699:0];
                reg pixel_buffer [702:0];
                reg buffer_three [2:0];
                reg [19:0] num_bits_per_frame; //345600
                reg [19:0] count;
                always @ (posedge clock) begin

                        count_out <= count + 1;
                        // new_pixel <= {h,s,v};
                        buffer_one <= {binarized_value, buffer_one[699:1]};
                        buffer_two <= {buffer_one[0], buffer_one[699:1]};
                        buffer_three <= {buffer_two[0], buffer_one[2:1]};

                        //shift pixel values in
                        pixel_buffer <= {pixel_value,pixel_buffer[701:1]};
                        pixel_out <= pixel_buffer[0];

                        a8 <= buffer_one [639];
                        a7 <= buffer_one [638];
                        a6 <= buffer_one [637];
                        a5 <= buffer_two [639];
                        a4 <= buffer_two [638];
                        a3 <= buffer_two [637];
                        a2 <= buffer_three [2];
                        a1 <= buffer_three [1];
                        a0 <= buffer_three [0];
                end
endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
//
// Created: December 1, 2018
// Authors: Jessica Quaye and Premila Rowles
//

module final_project(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; //unused because sound module drives these outputs
//   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; //used by NTSC
   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; //used by NTSC
   assign tv_in_clock = clock_27mhz;//1'b0;
   //assign tv_in_i2c_data = 1'bZ; //used by NTSC
   // 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;

/**********/
   //These values have to be set to 0 like ram0 since ram1 is used.
   assign ram1_adv_ld = 1'b0;
   assign ram1_ce_b = 1'b0;
   assign ram1_oe_b = 1'b0;
   assign ram1_bwe_b = 4'h0;

   // 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


   // LED Displays
/* USED in hex display
   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
    // 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; //used to drive LEDs
    assign user2 = 32'hZ;
//   assign user3 = 32'hZ; //used to drive LEDs
    assign user4 = 32'hZ;

    // Daughtercard Connectors
    assign daughtercard = 44'hZ;

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

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

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

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

    wire 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),
                                         .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;

    // display module used for debugging
    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 ram0_clk_not_used;
   zbt_6111 zbt0(clk, 1'b1, vram_we, vram_addr,
                 vram_write_data, vram_read_data,
                 ram0_clk_not_used,   //to get good timing, don't connect ram_clk
to zbt_6111
                 ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

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

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

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

   wire [29:0] ycrcb;   // video data (luminance, chrominance)
   wire [2:0] fvh;      // sync for field, vertical, horizontal
   wire       dv;       // data valid
   wire [23:0] rgb_out;//change
   wire [7:0] h,s,v;
   wire binarized_pixel;
   wire [23:0] vga_pixel;
   wire [23:0] pixel_out;
   wire is_red;
   wire is_blue;
   wire is_green;

   //convert NTSC raw analog output to digital
   ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                       .tv_in_ycrcb(tv_in_ycrcb[19:10]),
                       .ycrcb(ycrcb), .f(fvh[2]),
                       .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

   //convert from YCrCb space (camera output) to RGB color space
   YCrCb2RGB ycrcb2rgb_module(.R(rgb_out[23:16]),.G(rgb_out[15:8]),
                              .B(rgb_out[7:0]),.clk(tv_in_line_clock1),.rst(reset),
                              .Y(ycrcb[29:20]),.Cr(ycrcb[19:10]),.Cb(ycrcb[9:0]));


   //convert from rgb to hsv space for optimal color thresholding
   rgb2hsv rgb2hsv_module(.clock(tv_in_line_clock1),.reset(reset),
                          .r(rgb_out[23:16]),.g(rgb_out[15:8]),.b(rgb_out[7:0]),
                          .h(h),.s(s),.v(v));

   //define upper and lower bounds for testing during chroma keying
        reg [7:0] h_upper_bound = 255;
        reg [7:0] h_lower_bound = 0;
        reg [7:0] s_upper_bound = 255;
        reg [7:0] s_lower_bound = 0;
        reg [7:0] v_upper_bound = 255;
        reg [7:0] v_lower_bound = 0;
        reg [10:0] counter = 0;
        wire [23:0] vga_pixel_output;
```

```
  //hsv_threshold determines if a given pixel is within the bounds of hue and value
 ranges
  hsv_threshold #(.H_LOWER_BOUND_BLUE(8'hAA), .H_UPPER_BOUND_BLUE(8'hAA),
                .S_UPPER_BOUND_BLUE(s_upper_bound),
                .S_LOWER_BOUND_BLUE(s_lower_bound),
                .V_UPPER_BOUND_BLUE(8'hFF), .V_LOWER_BOUND_BLUE(8'hA3),
                .H_LOWER_BOUND_GREEN(8'h55),
                .H_UPPER_BOUND_GREEN(8'h55),.S_UPPER_BOUND_GREEN(s_upper_bound),
                .S_LOWER_BOUND_GREEN(s_lower_bound),
                .V_UPPER_BOUND_GREEN(8'hFF), .V_LOWER_BOUND_GREEN(8'h77))

  hsv_threshold_module(.rgb_pixel(rgb_out),.hsv_pixel({h,s,v}),
                      .pixel_out(pixel_out), .is_blue(is_blue),
                      .is_green(is_green));

  //image selector outputs a color for each pixel depending on hsv_threshold output
  image_selector image_selector_module(.pixel(pixel_out),.is_blue(is_blue),
                                      .is_green(is_green), .vga_pixel(vga_pixel),
                                      .binarized_pixel(binarized_pixel));

  //code to write NTSC data to video memory
  wire [18:0] ntsc_addr;
  wire [35:0] ntsc_data;
  wire        ntsc_we;

  //pass in 18 bit pixel to store in zbt memory
  ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, {vga_pixel[23:18],
                 vga_pixel[15:10], vga_pixel[7:2]},
                 ntsc_addr, ntsc_data, ntsc_we, switch[2]);

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

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

  //mux selecting read/write to memory based on which write-enable is chosen
  wire         sw_ntsc = ~switch[3];
  wire         my_we = sw_ntsc ? (hcount[0]==1'b1) : blank;
  wire [18:0]  write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
  wire [35:0]  write_data = sw_ntsc ? ntsc_data : vpat;

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



  wire[17:0]   pixel;
  reg  b,hs,vs;
  wire sign = 0;
  reg start = 0;


 //define output wires for each instance of the region module
  wire [4:0] state_center;
  wire [4:0] state_one;
  wire [4:0] state_two;
  wire [4:0] state_three;
  wire [4:0] state_four;
  wire [4:0] state_five;
  wire [4:0] state_six;
  wire [4:0] state_seven;
  wire [4:0] state_eight;

  wire end_frame;
  wire start_frame;

  //Clock Divider Outputs
  wire one_hz_enable;
```

```
wire one_mhz_enable;
reg[1:0] disp_state = 0;


reg[23:0] rgb;
wire display;

wire display_controller;
assign display_controller = switch[4];
wire [23:0] visualization_pixel;

wire [23:0] video_pixel;
wire use_video_pixel;

//assign read control to switches here
wire read_control;
assign read_control = switch[0];

always @(posedge clk) begin
   b <= blank;
   hs <= hsync;
   vs <= vsync;

   if (display_controller == 1) begin
      if (display == 1) begin
         rgb[23:16] <= {pixel[17:12],2'b0};
         rgb[15:8] <= {pixel[11:6],2'b0};
         rgb[7:0] <= {pixel[5:0],2'b0};
      end

     else rgb <= 24'b0;
   end

   else begin
      if (use_video_pixel == 1) rgb <= video_pixel;
      else rgb <= visualization_pixel;
   end
end

wire [23:0] x_avg_green_center;
wire [23:0] y_avg_green_center;
wire [23:0] x_avg_blue_center;
wire [23:0] y_avg_blue_center;

wire [23:0] x_avg_green_one;
wire [23:0] y_avg_green_one;
wire [23:0] x_avg_blue_one;
wire [23:0] y_avg_blue_one;

wire [23:0] x_avg_green_two;
wire [23:0] y_avg_green_two;
wire [23:0] x_avg_blue_two;
wire [23:0] y_avg_blue_two;

wire [23:0] x_avg_green_three;
wire [23:0] y_avg_green_three;
wire [23:0] x_avg_blue_three;
wire [23:0] y_avg_blue_three;

wire [23:0] x_avg_green_four;
wire [23:0] y_avg_green_four;
wire [23:0] x_avg_blue_four;
wire [23:0] y_avg_blue_four;

wire [23:0] x_avg_green_five;
wire [23:0] y_avg_green_five;
wire [23:0] x_avg_blue_five;
wire [23:0] y_avg_blue_five;

wire [23:0] x_avg_green_six;
wire [23:0] y_avg_green_six;
wire [23:0] x_avg_blue_six;
wire [23:0] y_avg_blue_six;
```

```
    wire [23:0] x_avg_green_seven;
    wire [23:0] y_avg_green_seven;
    wire [23:0] x_avg_blue_seven;
    wire [23:0] y_avg_blue_seven;

    wire [23:0] x_avg_green_eight;
    wire [23:0] y_avg_green_eight;
    wire [23:0] x_avg_blue_eight;
    wire [23:0] y_avg_blue_eight;

    //instance of region module for each region on the road
    region #(.UPPER_X(360), .UPPER_Y(300), .LOWER_X(445), .LOWER_Y(345))
       region_center_module(.clk(clk),.clock(clock_65mhz),.vr_pixel(vr_pixel),
                            .display(display),.hcount(hcount),.vcount(vcount),
                            .x_avg_green(x_avg_green_center),
                            .y_avg_green(y_avg_green_center),
                            .x_avg_blue(x_avg_blue_center),
                            .y_avg_blue(y_avg_blue_center),
                            .state(state_center));

    region #(.UPPER_X(360), .UPPER_Y(70), .LOWER_X(395), .LOWER_Y(285+35))
       region_one_module(.clk(clk),.clock(clock_65mhz),.vr_pixel(vr_pixel),
                         .display(display),.hcount(hcount),.vcount(vcount),
                         .x_avg_green(x_avg_green_one),
                         .y_avg_green(y_avg_green_one),
                         .x_avg_blue(x_avg_blue_one),
                         .y_avg_blue(y_avg_blue_one),
                         .state(state_one));

    region #(.UPPER_X(420), .UPPER_Y(95), .LOWER_X(455), .LOWER_Y(265+35))
       region_two_module(.clk(clk),.clock(clock_65mhz),.vr_pixel(vr_pixel),
                         .display(display),.hcount(hcount),.vcount(vcount),
                         .x_avg_green(x_avg_green_two),
                         .y_avg_green(y_avg_green_two),
                         .x_avg_blue(x_avg_blue_two),
                         .y_avg_blue(y_avg_blue_two),
                         .state(state_two));

    region #(.UPPER_X(50), .UPPER_Y(275), .LOWER_X(330), .LOWER_Y(305))
       region_three_module(.clk(clk),.clock(clock_65mhz),.vr_pixel(vr_pixel),
                           .display(display),.hcount(hcount),.vcount(vcount),
                           .x_avg_green(x_avg_green_three),
                           .y_avg_green(y_avg_green_three),
                           .x_avg_blue(x_avg_blue_three),
                           .y_avg_blue(y_avg_blue_three),
                           .state(state_three));


    region #(.UPPER_X(40), .UPPER_Y(330), .LOWER_X(338), .LOWER_Y(360))
       region_four_module(.clk(clk),.clock(clock_65mhz),.vr_pixel(vr_pixel),
                          .display(display),.hcount(hcount),.vcount(vcount),
                          .x_avg_green(x_avg_green_four),
                          .y_avg_green(y_avg_green_four),
                          .x_avg_blue(x_avg_blue_four),
                          .y_avg_blue(y_avg_blue_four),
                          .state(state_four));

    region #(.UPPER_X(465), .UPPER_Y(275), .LOWER_X(740), .LOWER_Y(305))
       region_five_module(.clk(clk),.clock(clock_65mhz),.vr_pixel(vr_pixel),
                          .display(display),.hcount(hcount),.vcount(vcount),
                          .x_avg_green(x_avg_green_five),
                          .y_avg_green(y_avg_green_five),
                          .x_avg_blue(x_avg_blue_five),
                          .y_avg_blue(y_avg_blue_five),
                          .state(state_five));


    region #(.UPPER_X(465), .UPPER_Y(330), .LOWER_X(740), .LOWER_Y(360))
       region_six_module(.clk(clk),.clock(clock_65mhz),.vr_pixel(vr_pixel),
                         .display(display),.hcount(hcount),.vcount(vcount),
                         .x_avg_green(x_avg_green_six),
                         .y_avg_green(y_avg_green_six),
```

```
                                   .x_avg_blue(x_avg_blue_six)
                                   ,.y_avg_blue(y_avg_blue_six),
                                   .state(state_six));

  region #(.UPPER_X(360), .UPPER_Y(368-40), .LOWER_X(395), .LOWER_Y(550))
      region_seven_module(.clk(clk),.clock(clock_65mhz),.vr_pixel(vr_pixel),
                                     .display(display),.hcount(hcount),.vcount(vcount),
                                     .x_avg_green(x_avg_green_seven),
                                     .y_avg_green(y_avg_green_seven),
                                     .x_avg_blue(x_avg_blue_seven),
                                     .y_avg_blue(y_avg_blue_seven),
                                     .state(state_seven));

  region #(.UPPER_X(420), .UPPER_Y(368-40), .LOWER_X(455), .LOWER_Y(550))
      region_eight_module(.clk(clk),.clock(clock_65mhz),.vr_pixel(vr_pixel),
                                     .display(display),.hcount(hcount),.vcount(vcount),
                                     .x_avg_green(x_avg_green_eight),
                                     .y_avg_green(y_avg_green_eight),
                                     .x_avg_blue(x_avg_blue_eight),
                                     .y_avg_blue(y_avg_blue_eight),
                                     .state(state_eight),
                                     .new_car(new_car),
                                     .end_frame(end_frame),.start_frame(start_frame));


   //create crosshairs for center of mass for each car to debug averaging method
   assign pixel = ((hcount == x_avg_green_center || vcount == y_avg_green_center
                                                  || hcount == x_avg_green_one
                                                  || vcount == y_avg_green_one
                                                  || hcount == x_avg_green_two
                                                  || vcount == y_avg_green_two
                                                  || hcount == x_avg_green_three
                                                  || vcount == y_avg_green_three
                                                  || hcount == x_avg_green_four
                                                  || vcount == y_avg_green_four
                                                  || hcount == x_avg_green_five
                                                  || vcount == y_avg_green_five
                                                  || hcount == x_avg_green_six
                                                  || vcount == y_avg_green_six
                                                  || hcount == x_avg_green_seven
                                                  || vcount == y_avg_green_seven
                                                  || hcount == x_avg_green_eight
                                                  || vcount == y_avg_green_eight
                                                  || hcount == x_avg_blue_one
                                                  || vcount == y_avg_blue_one
                                                  || hcount == x_avg_blue_two
                                                  || vcount == y_avg_blue_two
                                                  || count == x_avg_blue_three
                                                  || vcount == y_avg_blue_three)
                                                  ? 24'hFFFFFF : 0 ) | vr_pixel;
  //must include a check for display to ensure we don't have external noise
  //when calculating center of mass
  assign display = ((hcount > 40 && hcount < 734) && (vcount > 64 && vcount < 565));


   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clk.

   assign vga_out_red = rgb[23:16];
   assign vga_out_green = rgb[15:8];
   assign vga_out_blue = rgb[7:0];
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_pixel_clock = ~clk;
   assign vga_out_blank_b = ~b;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;


/////////////////////////////////////////////////////////////////////
// End of Image Processsing
// Beginning of Integration
```

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

   //Traffic FSM Input
   wire[2:0] main_road_count;
   wire[2:0] side_road_count;
   wire ped_cross_main_road;
   wire ped_cross_side_road;

   wire [1:0] car1_direction, car2_direction, car3_direction, car4_direction,
              car5_direction, car6_direction, car7_direction, car8_direction,
              car9_direction, car10_direction, car11_direction,car12_direction,
              car13_direction;

   //calculate number of cars on each road (using vertical and horizontal)
   assign main_road_count = (x_avg_green_one[10:0] > 0) +
                            (x_avg_green_two[10:0] > 0) +
                            (x_avg_blue_one[10:0] > 0) +
                            (x_avg_blue_two[10:0] > 0) +
                            (x_avg_green_seven[10:0] > 0) +
                            (x_avg_green_eight[10:0] > 0);
   assign side_road_count = (x_avg_green_three[10:0] > 0) +
                            (x_avg_blue_three[10:0] > 0) +
                            (x_avg_green_four[10:0] > 0) +
                            (x_avg_blue_four[10:0] > 0) +
                            (x_avg_green_five[10:0] > 0) +
                            (x_avg_blue_five[10:0] > 0) +
                            (x_avg_green_six[10:0] > 0);

   //debouncing signals for buttons
   wire ped_cross_up, ped_cross_down, ped_cross_right, ped_cross_left;
   debounce upbtn(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~button_up),.cle
an(ped_cross_up));
   debounce downbtn(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~button_down),
.clean(ped_cross_down));
   debounce rightbtn(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~button_right
),.clean(ped_cross_right));
   debounce leftbtn(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~button_left),
.clean(ped_cross_left));

   //use buttons to simulate pedestrian signals
   assign ped_cross_main_road = ped_cross_right | ped_cross_left;
   assign ped_cross_side_road = ped_cross_up | ped_cross_down;

   //Traffic FSM Outputs
   wire[1:0] main_out;
   wire[1:0] side_out;
   wire[2:0] out_state;


    //LED Outputs
   wire main_red;
   wire main_yellow;
   wire main_green;
   wire side_red;
   wire side_yellow;
   wire side_green;

    //Set hex display values
   always @(posedge clock_65mhz) begin
     dispdata <= {1'b0, out_state, 2'b0, main_out, 2'b0, side_out};
   end

   //LED strip Outputs
   //Main Road LED
   wire main_led_data;
   wire main_led_clock;
   wire main_enable_led_clock;
   assign main_led_clock = one_mhz_enable;

   //Side Road LED
   wire side_led_data;
   wire side_led_clock;
   wire side_enable_led_clock;
```

```
   assign side_led_clock = one_mhz_enable;


   //Instantiate all modules
   traffic_fsm traffic(.clk(clock_27mhz), .main_road_count(main_road_count),
                       .side_road_count(side_road_count),
                       .ped_cross_main_road(ped_cross_main_road),
                       .ped_cross_side_road(ped_cross_side_road),
                       .reset(reset), .one_hz_enable(one_hz_enable),
                       .main_out(main_out), .side_out(side_out),
                       .out_state(out_state));

   clock_divider one_hz(.clk(clk), .one_hz_enable(one_hz_enable));
   led_divider one_mhz(.clk(clk), .one_mhz_enable(one_mhz_enable));

   led_controller led_out(.clk(clock_27mhz),.main_out(main_out),
                          .side_out(side_out),
                          .main_red(main_red),.main_yellow(main_yellow),
                          .main_green(main_green),
                          .side_red(side_red),.side_yellow(side_yellow),
                          .side_green(side_green));

   led_strip main_led_strip(.clk(clk), .led_clock(main_led_clock),
                            .red_signal(main_red),.yellow_signal(main_yellow),
                            .green_signal(main_green),
                            .main_led_data(main_led_data),
                            .main_enable_led_clock(main_enable_led_clock));

   led_strip side_led_strip(.clk(clk), .side_led_clock(side_led_clock),
                            .red_signal(side_red),.yellow_signal(side_yellow),
                            .green_signal(side_green),
                            .side_led_data(side_led_data),
                            .side_enable_led_clock(side_enable_led_clock));

   //send traffic signals to led strip
   //1 is data (yellow wire)
   //0 is clock (blue wire)
   assign user1[1:0] = {main_led_data,(main_led_clock && main_enable_led_clock)};
   assign user3[1:0] = {side_led_data, (side_led_clock && side_enable_led_clock)};


   //Visualization signals
   //send signals to visualization module
   wire viz_hsync, viz_vsync, viz_blank;

   //declare car inputs
   wire [10:0] car1_leftx, car2_leftx, car3_leftx, car4_leftx, car5_leftx,
               car6_leftx,car7_leftx, car8_leftx, car9_leftx, car10_leftx;
   wire [10:0] car11_leftx, car12_leftx, car13_leftx;
   wire [9:0] car1_topy, car2_topy, car3_topy, car4_topy, car5_topy, car6_topy,
              car7_topy, car8_topy, car9_topy, car10_topy;
   wire [9:0] car11_topy, car12_topy, car13_topy;
   wire [10:0] car1_width, car2_width, car3_width, car4_width, car5_width,
               car6_width, car7_width,car8_width, car9_width, car10_width;
   wire [10:0] car11_width, car12_width, car13_width;
   wire [9:0] car1_height, car2_height, car3_height, car4_height, car5_height,
              car6_height, car7_height, car8_height, car9_height, car10_height;
   wire[9:0] car11_height, car12_height, car13_height;

   //REGIONS 1 To 3 FOR GREEN
   assign car1_leftx = (x_avg_green_one[10:0] > 0) ? (x_avg_green_one[10:0] + 80) :
0;
   assign car1_topy = (y_avg_green_one[9:0] > 0) ? (y_avg_green_one[9:0]) : 0;

   assign car2_leftx = (x_avg_green_two[10:0] > 0) ? (x_avg_green_two[10:0] + 90) :
0;
   assign car2_topy = (y_avg_green_two[9:0] > 0) ? (y_avg_green_two[9:0]) : 0;

   assign car3_leftx = (x_avg_green_three[10:0] > 0) ? (x_avg_green_three[10:0] + 30
) : 0; //HORIZ
   assign car3_topy = (y_avg_green_three[9:0] > 0) ? (y_avg_green_three[9:0] + 60) :
 0; //HORIZ
```

```
    //REGIONS 1 TO 3 FOR BLUE
    assign car4_leftx = (x_avg_blue_one[10:0] > 0) ? (x_avg_blue_one[10:0] + 80) : 0;

    assign car4_topy = (y_avg_blue_one[9:0] > 0 ) ? (y_avg_blue_one[9:0] ) : 0;

    assign car5_leftx = (x_avg_blue_two[10:0] > 0) ? (x_avg_blue_two[10:0] + 90) : 0;

    assign car5_topy = (y_avg_blue_two[10:0] > 0) ? (y_avg_blue_two[10:0] ) : 0;

    assign car6_leftx = (x_avg_blue_three[10:0] > 0) ? (x_avg_blue_three[10:0] + 30)
: 0; //HORIZ
    assign car6_topy = (y_avg_blue_three[9:0] > 0) ? (y_avg_blue_three[9:0] + 60) : 0
; //HORIZ

    //REGIONS 4 TO 6 FOR GREEN
    assign car7_leftx = (x_avg_green_four[10:0] > 0) ? (x_avg_green_four[10:0] + 30)
: 0; //HORIZ
    assign car7_topy = (y_avg_green_four[9:0] > 0) ? (y_avg_green_four[9:0] + 60) : 0
; //HORIZ

    assign car8_leftx = (x_avg_green_five[10:0] > 0) ? (x_avg_green_five[10:0] + 150)
 : 0; //HORIZ
    assign car8_topy = (y_avg_green_five[9:0] > 0) ? (y_avg_green_five[9:0] + 60) : 0
; //HORIZ

    assign car9_leftx = (x_avg_green_six[10:0] > 0) ? (x_avg_green_six[10:0] + 150) :
 0; //HORIZ
    assign car9_topy = (y_avg_green_six[9:0] > 0) ? (y_avg_green_six[9:0] + 80) : 0;
//HORIZ


    //REGIONS 7 AND 8 GREEN
    assign car10_leftx = (x_avg_green_seven[10:0] > 0) ? (x_avg_green_seven[10:0] + 1
00) : 0;
    assign car10_topy = (y_avg_green_seven[9:0] > 0) ? (y_avg_green_seven[9:0] + 80)
: 0;

    assign car11_leftx = (x_avg_green_eight[10:0] > 0) ? (x_avg_green_eight[10:0] + 1
00) : 0;
    assign car11_topy = (y_avg_green_eight[9:0] > 0) ? (y_avg_green_eight[9:0] + 80)
: 0;

  //REGIONS 4 TO 6 BLUE
    assign car12_leftx = (x_avg_blue_four[10:0] > 0) ? (x_avg_blue_four[10:0] + 0) :
0; //HORIZ
    assign car12_topy = (y_avg_blue_four[9:0] > 0) ? (y_avg_blue_four[9:0] + 60) : 0;
 //HORIZ

    assign car13_leftx = (x_avg_blue_five[10:0] > 0) ? (x_avg_blue_five[10:0] + 150)
: 0; //HORIZ
    assign car13_topy = (y_avg_blue_five[9:0] > 0) ? (y_avg_blue_five[9:0] + 60) : 0;
 //HORIZ

    //calculate width, height and direction of car
    w_and_h_calc wcalc1(.clk(clk),.car_x(car1_leftx),.car_y(car1_topy),.car_height(ca
r1_height),.car_width(car1_width),.car_direction(car1_direction));
    w_and_h_calc wcalc2(.clk(clk),.car_x(car2_leftx),.car_y(car2_topy),.car_height(ca
r2_height),.car_width(car2_width),.car_direction(car2_direction));
    w_and_h_calc wcalc3(.clk(clk),.car_x(car3_leftx),.car_y(car3_topy),.car_height(ca
r3_height),.car_width(car3_width),.car_direction(car3_direction));
    w_and_h_calc wcalc4(.clk(clk),.car_x(car4_leftx),.car_y(car4_topy),.car_height(ca
r4_height),.car_width(car4_width),.car_direction(car4_direction));
    w_and_h_calc wcalc5(.clk(clk),.car_x(car5_leftx),.car_y(car5_topy),.car_height(ca
r5_height),.car_width(car5_width),.car_direction(car5_direction));
    w_and_h_calc wcalc6(.clk(clk),.car_x(car6_leftx),.car_y(car6_topy),.car_height(ca
r6_height),.car_width(car6_width),.car_direction(car6_direction));
    w_and_h_calc wcalc7(.clk(clk),.car_x(car7_leftx),.car_y(car7_topy),.car_height(ca
r7_height),.car_width(car7_width),.car_direction(car7_direction));
    w_and_h_calc wcalc8(.clk(clk),.car_x(car8_leftx),.car_y(car8_topy),.car_height(ca
r8_height),.car_width(car8_width),.car_direction(car8_direction));
    w_and_h_calc wcalc9(.clk(clk),.car_x(car9_leftx),.car_y(car9_topy),.car_height(ca
r9_height),.car_width(car9_width),.car_direction(car9_direction));
    w_and_h_calc wcalc10(.clk(clk),.car_x(car10_leftx),.car_y(car10_topy),.car_height
```

```
(car10_height),.car_width(car10_width),.car_direction(car10_direction));
   w_and_h_calc wcalc11(.clk(clk),.car_x(car11_leftx),.car_y(car11_topy),.car_height
(car11_height),.car_width(car11_width),.car_direction(car11_direction));
   w_and_h_calc wcalc12(.clk(clk),.car_x(car12_leftx),.car_y(car12_topy),.car_height
(car12_height),.car_width(car12_width),.car_direction(car12_direction));
   w_and_h_calc wcalc13(.clk(clk),.car_x(car13_leftx),.car_y(car13_topy),.car_height
(car13_height),.car_width(car13_width),.car_direction(car13_direction));
   w_and_h_calc wcalc14(.clk(clk),.car_x(car14_leftx),.car_y(car14_topy),.car_height
(car14_height),.car_width(car14_width),.car_direction(car14_direction));


   //detect pairwise collision
   wire is_collision14;
   wire is_collision15;
   wire is_collision24;
   wire is_collision25;
   wire is_collision36;
   wire is_collision = is_collision14 || is_collision15 || is_collision24 || is_coll
ision25 || is_collision36;
   wire [9:0] street_topy, street_bottomy;
   wire [10:0] street_leftx, street_rightx;

   wire [10:0] leftx_threshold14, rightx_threshold14;
   wire [9:0] uppery_threshold14, lowery_threshold14;

   wire [10:0] leftx_threshold15, rightx_threshold15;
   wire [9:0] uppery_threshold15, lowery_threshold15;

   wire [10:0] leftx_threshold24, rightx_threshold24;
   wire [9:0] uppery_threshold24, lowery_threshold24;

   wire [10:0] leftx_threshold25, rightx_threshold25;
   wire [9:0] uppery_threshold25, lowery_threshold25;

   wire [10:0] leftx_threshold36, rightx_threshold36;
   wire [9:0] uppery_threshold36, lowery_threshold36;

   //calculate ambulance params
   wire [10:0] ambulance_dest_x, ambulance_leftx, ambulance_width;
   wire [9:0] ambulance_dest_y, ambulance_topy, ambulance_height;
   wire[1:0] ambulance_move_dir;
   wire direction14;
   wire direction15;
   wire direction24;
   wire direction25;
   wire direction36;

   assign street_leftx = 11'd420;
   assign street_rightx = 11'd600;
   assign street_topy = 10'd344;
   assign street_bottomy = 10'd464;

   //draw visualization
   visualization street_viz(.vclock(clk), .one_hz_enable(one_hz_enable),.hcount(hcou
nt), .vcount(vcount), .hsync(hsync), .vsync(vsync), .blank(blank),
                            .car1_leftx(car1_leftx), .car1_topy(car1_topy),
                            .car2_leftx(car2_leftx), .car2_topy(car2_topy),
                            .car3_leftx(car3_leftx), .car3_topy(car3_topy),
                            .car4_leftx(car4_leftx), .car4_topy(car4_topy),
                            .car5_leftx(car5_leftx), .car5_topy(car5_topy),
                            .car6_leftx(car6_leftx), .car6_topy(car6_topy),
                            .car7_leftx(car7_leftx), .car7_topy(car7_topy),
                            .car8_leftx(car8_leftx), .car8_topy(car8_topy),
                            .car9_leftx(car9_leftx), .car9_topy(car9_topy),
                            .car10_leftx(car10_leftx), .car10_topy(car10_topy),
                            .car11_leftx(car11_leftx), .car11_topy(car11_topy),
                            .car12_leftx(car12_leftx), .car12_topy(car12_topy),
                            .car13_leftx(car13_leftx), .car13_topy(car13_topy),
                            .car1_direction(car1_direction), .car2_direction(car2_di
rection),
                            .car3_direction(car3_direction), .car4_direction(car4_di
rection),
                            .car5_direction(car5_direction), .car6_direction(car6_di
```

```
rection),
                            .car7_direction(car7_direction), .car8_direction(car8_di
rection),
                            .car9_direction(car9_direction), .car10_direction(car10_
direction),
                            .car11_direction(car11_direction),  .car12_direction(car
12_direction),
                            .car13_direction(car13_direction),

                            .car1_width(car1_width), .car2_width(car2_width),
                            .car3_width(car3_width), .car4_width(car4_width),
                            .car5_width(car5_width), .car6_width(car6_width),
                            .car7_width(car7_width), .car8_width(car8_width),
                            .car9_width(car9_width), .car10_width(car10_width),
                            .car11_width(car11_width), .car12_width(car12_width),
                            .car13_width(car13_width),
                            .car1_height(car1_height),.car2_height(car2_height),
                            .car3_height(car3_height),.car4_height(car4_height),
                            .car5_height(car5_height),.car6_height(car6_height),
                            .car7_height(car7_height),.car8_height(car8_height),
                            .car9_height(car9_height), .car10_height(car10_height),
                            .car11_height(car11_height), .car12_height(car12_height)
,
                            .car13_height(car13_height),
                            .is_collision(is_collision),
                            .ambulance_leftx(ambulance_leftx),  .ambulance_width(amb
ulance_width),
                            .ambulance_topy(ambulance_topy), .ambulance_height(ambul
ance_height), .ambulance_direction(ambulance_move_dir),
                            .main_out(main_out), .side_out(side_out), .viz_hsync(viz
_hsync),
                            .viz_vsync(viz_vsync), .viz_blank(viz_blank), .pixel(vis
ualization_pixel));

   collision_detector coll_detect14( .clk(clk), .car1_leftx(car1_leftx),.car1_rightx
(car1_leftx + car1_width),
                                     .car1_topy(car1_topy),.car1_bottomy(car1_topy +
 car1_height),
                                     .car2_leftx(car4_leftx),.car2_rightx(car4_leftx
 + car4_width),
                                     .car2_topy(car4_topy), .car2_bottomy(car4_topy
+ car4_height),
                                     .street_topy(street_topy), .street_bottomy(stre
et_bottomy),
                                     .street_leftx(street_leftx), .street_rightx(str
eet_rightx),
                                     .leftx_threshold(leftx_threshold14), .rightx_th
reshold(rightx_threshold14),
                                     .uppery_threshold(uppery_threshold14), .lowery_
threshold(lowery_threshold14),
                                     .direction(direction14),.is_collision(is_collis
ion14));

   collision_detector coll_detect15( .clk(clk), .car1_leftx(car1_leftx),.car1_rightx
(car1_leftx + car1_width),
                                     .car1_topy(car1_topy),.car1_bottomy(car1_topy +
 car1_height),
                                     .car2_leftx(car5_leftx),.car2_rightx(car5_leftx
 + car5_width),
                                     .car2_topy(car5_topy), .car2_bottomy(car5_topy
+ car5_height),
                                     .street_topy(street_topy), .street_bottomy(stre
et_bottomy),
                                     .street_leftx(street_leftx), .street_rightx(str
eet_rightx),
                                     .leftx_threshold(leftx_threshold15), .rightx_th
reshold(rightx_threshold15),
                                     .uppery_threshold(uppery_threshold15), .lowery_
threshold(lowery_threshold15),
                                     .direction(direction15),.is_collision(is_collis
ion15));

   collision_detector coll_detect24( .clk(clk), .car1_leftx(car2_leftx),.car1_rightx
```

```
(car2_leftx + car2_width),
.car1_topy(car2_topy),.car1_bottomy(car2_topy + car2_height),
.car2_leftx(car4_leftx),.car2_rightx(car4_leftx + car4_width),
.car2_topy(car4_topy), .car2_bottomy(car4_topy + car4_height),
.street_topy(street_topy), .street_bottomy(street_bottomy),
.street_leftx(street_leftx), .street_rightx(street_rightx),
.leftx_threshold(leftx_threshold24), .rightx_threshold(rightx_threshold24),
.uppery_threshold(uppery_threshold24), .lowery_threshold(lowery_threshold24),
.direction(direction24),.is_collision(is_collision24));

   collision_detector coll_detect25( .clk(clk), .car1_leftx(car2_leftx),.car1_rightx
(car2_leftx + car2_width),
.car1_topy(car2_topy),.car1_bottomy(car2_topy + car2_height),
.car2_leftx(car5_leftx),.car2_rightx(car5_leftx + car5_width),
.car2_topy(car5_topy), .car2_bottomy(car5_topy + car5_height),
.street_topy(street_topy), .street_bottomy(street_bottomy),
.street_leftx(street_leftx), .street_rightx(street_rightx),
.leftx_threshold(leftx_threshold25), .rightx_threshold(rightx_threshold25),
.uppery_threshold(uppery_threshold25), .lowery_threshold(lowery_threshold25),
.direction(direction25),.is_collision(is_collision25));

   collision_detector coll_detect36( .clk(clk), .car1_leftx(car3_leftx),.car1_rightx
(car3_leftx + car3_width),
.car1_topy(car3_topy),.car1_bottomy(car3_topy + car3_height),
.car2_leftx(car6_leftx),.car2_rightx(car6_leftx + car6_width),
.car2_topy(car6_topy), .car2_bottomy(car6_topy + car6_height),
.street_topy(street_topy), .street_bottomy(street_bottomy),
.street_leftx(street_leftx), .street_rightx(street_rightx),
.leftx_threshold(leftx_threshold36), .rightx_threshold(rightx_threshold36),
.uppery_threshold(uppery_threshold36), .lowery_threshold(lowery_threshold36),
.direction(direction36),.is_collision(is_collision36));

   reg[10:0] leftx_threshold;
   reg[10:0] rightx_threshold;
   reg[9:0] uppery_threshold;
   reg[9:0] lowery_threshold;
   reg direction;

   always @ (posedge clk) begin
      if (is_collision == 1) begin
         if ((leftx_threshold14 > 0) && (rightx_threshold14 > 0)) begin
            leftx_threshold <= leftx_threshold14;
            rightx_threshold <= rightx_threshold14;
            uppery_threshold <= uppery_threshold14;
            lowery_threshold <= lowery_threshold14;
            direction <= direction14;
         end

         if ((leftx_threshold15 > 0) && (rightx_threshold15 > 0))  begin
```

```
            leftx_threshold <= leftx_threshold15;
            rightx_threshold <= rightx_threshold15;
            uppery_threshold <= uppery_threshold15;
            lowery_threshold <= lowery_threshold15;
            direction <= direction15;
         end

         if ((leftx_threshold24 > 0) && (rightx_threshold24 > 0)) begin
            leftx_threshold <= leftx_threshold24;
            rightx_threshold <= rightx_threshold24;
            uppery_threshold <= uppery_threshold24;
            lowery_threshold <= lowery_threshold24;
            direction <= direction24;
         end

         if ((leftx_threshold25 > 0) && (rightx_threshold25 > 0)) begin
            leftx_threshold <= leftx_threshold25;
            rightx_threshold <= rightx_threshold25;
            uppery_threshold <= uppery_threshold25;
            lowery_threshold <= lowery_threshold25;
            direction <= direction25;
         end

         if ((leftx_threshold36 > 0) && (rightx_threshold36 > 0)) begin
            leftx_threshold <= leftx_threshold36;
            rightx_threshold <= rightx_threshold36;
            uppery_threshold <= uppery_threshold36;
            lowery_threshold <= lowery_threshold36;
            direction <= direction36;
         end
      end
   end


   //Calculate thresholds and other factors that will affect ambulance direction
   calc_ambulance_params ambulance_calc(.clk(clk), .leftx_threshold(leftx_threshold)
, .rightx_threshold(rightx_threshold), .street_leftx(street_leftx), .street_rightx(s
treet_rightx),
                                        .uppery_threshold(uppery_threshold), .lowery
_threshold(lowery_threshold), .street_topy(street_topy), .street_bottomy(street_bott
omy),
                                        .direction(direction),
                                        .is_collision(is_collision),
                                        .ambulance_move_dir(ambulance_move_dir),
                                        .ambulance_dest_x(ambulance_dest_x),
                                        .ambulance_dest_y(ambulance_dest_y));

   //Use the ambulance parameters to determine where the ambulance should
   //start from and its destination limit

   get_amb_xy ambxny(.clk(clk), .one_hz_enable(one_hz_enable), .is_collision(is_coll
ision),
                     .ambulance_move_dir(ambulance_move_dir), .ambulance_leftx(ambul
ance_leftx),  .ambulance_dest_x(ambulance_dest_x), .ambulance_width(ambulance_width)
,
                     .ambulance_topy(ambulance_topy), .ambulance_dest_y(ambulance_de
st_y), .ambulance_height(ambulance_height));


   //Video playback
   //variables declared in RGB place
   video video_stuff(.clk(clk), .one_hz_enable(one_hz_enable),
                     .visualization_pixel(visualization_pixel),
                     .hcount(hcount), .vcount(vcount),
                     .read_control(read_control),
                     .video_pixel(video_pixel),
                     .ram1_we_b(ram1_we_b), .ram1_address(ram1_address), .ram1_data(
ram1_data), .ram1_cen_b(ram1_cen_b), .use_video_pixel(use_video_pixel));


   //Audio
   wire [7:0] from_ac97_data, to_ac97_data;
   wire ready;
```

```
   wire[4:0] volume = 5'd25;

   // AC97 driver
   audio a(clk, reset, volume, from_ac97_data, to_ac97_data, ready,
            audio_reset_b, ac97_sdata_out, ac97_sdata_in,
            ac97_synch, ac97_bit_clock);

   // record module
   recorder r(.clock(clk), .reset(reset), .ready(ready),
            .play_sound(is_collision), .from_ac97_data(from_ac97_data),
            .to_ac97_data(to_ac97_data));

   assign led = ~{main_red, main_yellow, main_green, side_red, side_yellow, side_gre
en, 1'b0, is_collision};


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;//CHANGE
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

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

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

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

   always @(posedge clk)
     last_vr_data <= (hc2==1'b1) ? vr_data_latched : last_vr_data;//CHANGE

   always @(posedge clk)
     vr_data_latched <= (hc2==1'b0) ? vram_read_data : vr_data_latched;//CHANGE

   always @(*)          // each 36-bit word from RAM is decoded to 4 bytes
     case (hc2) //CHANGE
//       2'd3: vr_pixel = last_vr_data[17:0];
//       2'd2: vr_pixel = last_vr_data[7+8:0+8];
       1'd1: vr_pixel = last_vr_data[17:0];
       1'd0: vr_pixel = last_vr_data[35:18];
     endcase

endmodule // vram_display


/////////////////////////////////////////////////////////////////////////////
// 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"
```

```verilog
   // 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
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Premila Rowles
//
// Create Date:    14:34:10 11/19/2018
// Design Name:
// Module Name:    hsv_threshold
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////
module hsv_threshold
        #(parameter H_UPPER_BOUND = 24'hFFFFFF,
        H_LOWER_BOUND = 24'hFFFFFF,
        S_UPPER_BOUND = 24'hFFFFFF,
        S_LOWER_BOUND = 24'hFFFFFF,
        V_UPPER_BOUND = 24'hFFFFFF,
        V_LOWER_BOUND = 24'hFFFFFF,
        CAR_UPPER_BOUND = 24'hFFA500,
        CAR_LOWER_BOUND = 24'hFFA500)

        (input [23:0] rgb_pixel,
    input [23:0] hsv_pixel,
    output [23:0] pixel_out,
    output is_blue,
         output is_green
    );
         wire h_satisfied;
         wire s_satisfied;
         wire v_satisfied;


         //pixels are assigned a color depending on upper and lower bounds of hue an
d value parameters
        assign is_blue = ((hsv_pixel[23:16] >= H_LOWER_BOUND_BLUE) && (hsv_pixel[23:
16] <= H_UPPER_BOUND_BLUE)) &&
        ((hsv_pixel[15:8] >= S_LOWER_BOUND_BLUE) && (hsv_pixel[15:8] <= S_UPPER_BOUN
D_BLUE)) &&
                        ((hsv_pixel[7:0] >= V_LOWER_BOUND_BLUE) && (hsv_pixel[7:0] <
= V_UPPER_BOUND_BLUE));

        assign is_green = ((hsv_pixel[23:16] >= H_LOWER_BOUND_GREEN) && (hsv_pixel[2
3:16] <= H_UPPER_BOUND_GREEN)) &&
        ((hsv_pixel[15:8] >= S_LOWER_BOUND_GREEN) && (hsv_pixel[15:8] <= S_UPPER_BOU
ND_GREEN)) &&
                        ((hsv_pixel[7:0] >= V_LOWER_BOUND_GREEN) && (hsv_pixel[7:0]
<= V_UPPER_BOUND_GREEN));
        //keep passing rgb pixel along so we can assign a pixel to either its rgb ou
tput or its hsv output
        assign pixel_out = rgb_pixel;


endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Premila Rowles
//
// Create Date:    14:35:44 11/19/2018
// Design Name:
// Module Name:    image_selector
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module image_selector(
    input [23:0] pixel,
    input is_blue,
         input is_green,
    output [23:0] vga_pixel,
    output binarized_pixel
    );

                //assign pixels to be the color detected in hsv space and determined
 by the bounds of hue and value
                assign vga_pixel = is_blue ? 24'h0000FF : is_green ? 24'h00FF00 : pi
xel;

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Jessica Quaye
//
// Create Date:    13:32:18 11/05/2018
// Design Name:
// Module Name:    led_controller
//////////////////////////////////////////////////////////////////////////////////

//default convention for color output. RED = 0, YELLOW = 1, GREEN = 2
module led_controller(
    input clk,
    input [1:0] main_out,
    input [1:0]side_out,
    output reg main_red,
    output reg main_yellow,
    output reg main_green,
    output reg side_red,
    output reg side_yellow,
    output reg side_green);

    `include "params.v"
    always @(posedge clk)
    begin
        case(main_out) //determine outputs for main traffic lights
                RED:
                begin
                        main_red <= ON;
                        main_yellow <= OFF;
                        main_green <= OFF;
                end

                YELLOW:
                begin
                        main_red <= OFF;
                        main_yellow <= ON;
                        main_green <= OFF;
                end

                GREEN:
                begin
                        main_red <= OFF;
                        main_yellow <= OFF;
                        main_green <= ON;
                end
        default:;
        endcase

        case(side_out) //determine outputs for side traffic lights
                RED:
                begin
                        side_red <= ON;
                        side_yellow <= OFF;
                        side_green <= OFF;
                end

                YELLOW:
                begin
                        side_red <= OFF;
                        side_yellow <= ON;
                        side_green <= OFF;
                end

                GREEN:
                begin
                        side_red <= OFF;
                        side_yellow <= OFF;
                        side_green <= ON;
                end
        default:;
        endcase
    end //end always
```

```
endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Jessica Quaye
//
// Create Date:    22:38:34 11/27/2018
// Design Name:
// Module Name:    led_strip
////////////////////////////////////////////////////////////////////////////////////
module led_strip(
    input clk,
    input led_clock,
    input red_signal,
    input yellow_signal,
    input green_signal,
    output reg main_led_data,
    output reg main_enable_led_clock
    );

   `include "params.v"

   //at each rising edge of the clock we have a new frame to send
   //initialize frames of different colors with format 3 intro bits, 5 global bits,
8'bB, 8'bG, 8'bR
   reg[31:0] red_frame =    32'b111_00011_0000_0000_0000_0000_1111_1111; //3 intro b
its, 5 global bits, 8'bB, 8'bG, 8'bR
   reg[31:0] green_frame =  32'b111_00011_0000_0000_1111_1111_0000_0000; //3 intro b
its, 5 global bits, 8'bB, 8'bG, 8'bR
   reg[31:0] yellow_frame = 32'b111_00011_0000_0000_1111_1111_1111_1111; //3 intro b
its, 5 global bits, 8'bB, 8'bG, 8'bR
   reg[31:0] blank_frame = 32'b111_00011_0000_0000_0000_0000_0000_0000; //3 intro bi
ts, 5 global bits, 8'bB, 8'bG, 8'bR

   //FSM parameters
   reg [2:0] state = 3'b000;

   //counters
   reg [4:0] start_counter = 5'b0; //initialize counter to count 32 bits for start
   reg [1:0] frame_color; //determine which color frame we are sending according to
RED = 0, YELLOW = 1, GREEN = 2
   reg [4:0] led_frame_counter = 5'd31; //initialize counter to count 32 bits for ea
ch frame
   reg [4:0] same_frame_counter = 5'd0; //need to send 27 frames of each color so us
ed to send same frame till 27
   reg [7:0] blank_counter = 8'd0; //send blank bits

   //used to control whether (RED, YELLOW, GREEN) LEDs will be on or off
   reg [2:0] switch_control_values = 3'b0;

   //store prev led clock
   reg prev_led_clock;

   always @(posedge clk) begin
   prev_led_clock <= led_clock;

   if (prev_led_clock == 0 && led_clock == 1) begin //begin at rising edge of led_cl
ock
      case(state)
         SEND_START_FRAME: //send 32'b0 to wire as start frame
         begin
            main_led_data <= 1'b0;
            main_enable_led_clock <= 1;

            if (start_counter == 5'd31)
               begin
                  start_counter <= 5'b0;
                  frame_color <= 2'b0; //initialize all these parameters for followi
ng state
                  led_frame_counter <= 5'd31;
                  same_frame_counter <= 5'd0;
                  state <= SEND_FRAME; //move to next frame when the 31st bit is sen
t
               end
```

```
            else start_counter <= start_counter + 1;

         end

         SEND_FRAME:
         begin
            //choose which color to send
            if (frame_color == RED) //current focus is on RED section
               begin
                  if (switch_control_values[0] == 1)main_led_data <= red_frame[led_f
rame_counter]; //if signal for RED is on, turn on RED
                  else main_led_data <= blank_frame[led_frame_counter]; //else suppl
y blank frames
               end
            else if (frame_color == YELLOW) //current focus is on YELLOW section
               begin
                  if (switch_control_values[1] == 1) main_led_data <= yellow_frame[l
ed_frame_counter]; //if signal for YELLOW is on, turn on YELLOW
                  else main_led_data <= blank_frame[led_frame_counter]; //else suppl
y blank frames
               end
            else if (frame_color == GREEN) //current focus is on GREEN section
               begin
                  if (switch_control_values[2] == 1)main_led_data <= green_frame[led
_frame_counter]; //if signal for GREEN is on, turn on GREEN
                  else main_led_data <= blank_frame[led_frame_counter]; //else suppl
y blank frames
               end
            else state <= SEND_BLANK_FRAME;

            if (led_frame_counter == 5'b0) //when you are done with one frame (one L
ED)
                  begin
                     led_frame_counter <= 5'd31;

                     if (same_frame_counter == 5'd6) //if all LEDs for one section a
re handled, move to another color's frame
                           begin
                              same_frame_counter <= 5'd0;

                              if (frame_color == GREEN) state <= SEND_BLANK_FRAME; /
/after GREEN, just fill blank frames
                              else frame_color <= frame_color + 1;
                           end

                     else same_frame_counter <= same_frame_counter + 1; //else stay
in same frame and keep sending more
                  end

            else led_frame_counter <= led_frame_counter - 1; //otherwise continue it
erating through the frame reg to index frame values
         end


         SEND_BLANK_FRAME:
         begin
            main_led_data <= blank_frame[led_frame_counter];

         if (led_frame_counter == 5'b0)
               begin
               led_frame_counter <= 5'd31;

                  if (blank_counter == 8'd62) //after sending 2 full blank LEDs, sen
d end frame
                  begin
                     state <= SEND_END_FRAME;
                     blank_counter <= 0;
                  end

                  else blank_counter <= blank_counter + 1;
               end
```

```
        else led_frame_counter <= led_frame_counter - 1;


        end

        SEND_END_FRAME:
        begin
           start_counter <= start_counter + 1;
           main_led_data <= 1'b1;

           if (start_counter == 5'd31)
           begin
              main_enable_led_clock <= 0; //turn off main_enable_led_clock to avoid
 sending data after all frames have ended
              state <= READ_TRAFFIC_SIGNALS; //time to check switches
           end
        end

        READ_TRAFFIC_SIGNALS:
        begin
           switch_control_values <= {green_signal, yellow_signal, red_signal}; //in
vert what you expect because of how signals are sent - actually {r,y,g}
           state <= SEND_START_FRAME;
        end

      default: state <= SEND_START_FRAME;
      endcase

      end //end if one_mhz_enable
   end // end always

endmodule
```

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

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

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

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

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

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

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

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

        if (!fvh[2])
```

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

   // synchronize with system clock

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

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

   // edge detection on write enable signal

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

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

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

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


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

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

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

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

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

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


   // alternate (256x192) image data and address
   wire [35:0] mydata2 = {data[1],data[1],data[1],data[1]};//no change{data[1],data[
1],data[1],data[1]};
   wire [18:0] myaddr2 = {1'b0, y_addr[8:0], eo_delay[3], x_addr[7:0]};//no CHANGE//
{1'b0, y_addr[8:0], eo_delay[3], x_addr[7:0]};

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

   reg [18:0] ntsc_addr;
   reg [35:0] ntsc_data;
   wire       ntsc_we = sw ? we_edge : (we_edge & (x_delay[30]==1'b0));
   always @(posedge clk)
     if ( ntsc_we )
       begin
          ntsc_addr <= sw ? myaddr2 : myaddr;   // normal and expanded modes
          ntsc_data <= sw ? {4'b0,mydata2} : mydata;
  end

endmodule // ntsc_to_zbt
```

```
 `timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Jessica Quaye
//
// Create Date:    22:21:57 12/05/2018
// Design Name:
// Module Name:    params
//////////////////////////////////////////////////////////////////////////////////

//Street Dimensions
        parameter STREET_LEFTX = 11'd420;
        parameter STREET_RIGHTX = 11'd600;
        parameter STREET_VERT_MID = 11'd512;
        parameter STREET_TOPY = 10'd344;
        parameter STREET_BOTTOMY = 10'd464;
        parameter STREET_HORIZ_MID = 10'd402;

//Car Directions
        parameter MOVE_LEFT = 2'b01;
        parameter MOVE_RIGHT = 2'b00;
        parameter MOVE_UP = 2'b10;
        parameter MOVE_DOWN = 2'b11;

//General Constants
        parameter TRUE = 1;
        parameter FALSE = 0;
        parameter ON = 1;
        parameter OFF = 0;

//Line Directions
        parameter VERTICAL = 1'b1;
        parameter HORIZONTAL = 1'b0;

//Traffic Light Colors
         parameter RED = 2'b00;
         parameter YELLOW = 2'b01;
         parameter GREEN = 2'b10;

//Traffic Light FSM States
        parameter MAIN_RED_SIDE_GREEN = 3'b000;
        parameter MAIN_RED_SIDE_YELLOW = 3'b001;
        parameter MAIN_GREEN_SIDE_RED = 3'b010;
        parameter MAIN_YELLOW_SIDE_RED = 3'b011;

//Screen Limits
        parameter SCREEN_Y_LIMIT = 10'd768;
        parameter SCREEN_X_LIMIT = 11'd1024;

//LED Strip States
        parameter SEND_START_FRAME = 3'b000;
        parameter SEND_FRAME = 3'b001;
        parameter SEND_BLANK_FRAME = 3'b010;
        parameter SEND_END_FRAME = 3'b011;
        parameter READ_TRAFFIC_SIGNALS = 3'b100;

//      Ambulance Speed
parameter CSPEED = 4'd10;


//Video Parameters
                parameter NUMBER_OF_FRAMES = 6'd20;
                parameter NUM_FRAMES_X_LINES = 19'd491520; //24576*20frames
                parameter NUM_LINES_PER_FRAME = 19'd24576;

                //small frame params

                parameter FRAME_WIDTH = 11'd256;
                parameter FRAME_HEIGHT = 10'd192;
```

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Premila Rowles
//
// Create Date:    18:16:21 12/01/2018
// Design Name:
// Module Name:    region
//////////////////////////////////////////////////////////////////////////////////
module region

   #(parameter UPPER_X = 353,
    UPPER_Y = 272,
    LOWER_X = 459,
    LOWER_Y = 368)

    (input clk,
    input clock,
    input [17:0] vr_pixel,
    input [10:0] hcount,
    input [9:0] vcount,
    input display,
    output reg [23:0] x_avg_green,
    output reg [23:0] y_avg_green,
    output reg [23:0] x_avg_blue,
    output reg [23:0] y_avg_blue,
    output reg new_car,
    output reg [4:0] state,
    output reg end_frame,
    output reg start_frame

    );
   wire sign = 0;
   reg start = 0;

   reg [23:0] x_sum_g;
   reg [23:0] y_sum_g;

   reg [23:0] x_sum_b;
   reg [23:0] y_sum_b;


   reg [23:0] x_sum_green;
   reg [23:0] y_sum_green;
   reg [13:0] count_green;
   reg [23:0] x_sum_blue;
   reg [23:0] y_sum_blue;
   reg [13:0] count_blue;

   reg [13:0] count_avg_g;

   reg [13:0] count_avg_b;

   reg started_division;

   wire [11:0] x_quotient_g;
   wire [11:0] y_quotient_g;
   wire [10:0] remainder_x_g;
   wire [10:0] remainder_y_g;
   wire ready_x_g;
   wire ready_y_g;


   reg x_done_g;
   reg y_done_g;


   wire [11:0] x_quotient_b;
   wire [11:0] y_quotient_b;
   wire [10:0] remainder_x_b;
   wire [10:0] remainder_y_b;
   wire ready_x_b;
   wire ready_y_b;
```

```
   reg x_done_b;
   reg y_done_b;

   parameter RESET = 0;
   parameter LOAD_DATA = 1;
   parameter READY_X_READY_Y = 2;
   parameter BOTH_DONE = 3;
   parameter REGION_ONE_BLUE = 4;
   parameter REGION_TWO_GREEN = 5;
   parameter REGION_TWO_BLUE = 6;



    always @(posedge clk) // cross hairs generation
     begin

         if (hcount == 0 && vcount == 0) begin
               x_sum_green <= 0;
               y_sum_green <= 0;
               count_green <= 1;
               x_sum_blue <= 0;
               y_sum_blue <= 0;
               count_blue <= 1;

            end


         // start accumulations based on bounds and based on color (vr_pixel[11:6] i
s green)
         if ((display) && (hcount > UPPER_X) && (hcount < LOWER_X) && (vcount > UPPE
R_Y) &&
                (vcount < LOWER_Y) && (vr_pixel[11:6] == 6'b11_1111)) begin //region
 one and green
//
            x_sum_green <= x_sum_green + hcount;
            y_sum_green <= y_sum_green + vcount;
            count_green <= count_green + 1;

         end
         //vr_pixel[5:0] is blue
         if ((display) && (hcount > UPPER_X) && (hcount < LOWER_X) && (vcount > UPPE
R_Y) &&
                (vcount < LOWER_Y) && (vr_pixel[5:0] == 6'b11_1111)) begin //region
one and green

            x_sum_blue <= x_sum_blue + hcount;
            y_sum_blue <= y_sum_blue + vcount;
            count_blue <= count_blue + 1;
         end

        //state machine for dividing
      case(state)
         //reset state to set all values to 0 at start of frame
         RESET : begin
            if (hcount == 0 && vcount == 0) begin
               start <= 0;
               x_sum_g <= 0;
               y_sum_g <= 0;
               count_avg_g<= 0;
               x_sum_b <= 0;
               y_sum_b <= 0;
               count_avg_b <= 0;
               // once we reach the end of the frame, we have calculated all of our
sums
               // and we can start dividing
            if (hcount == 750 && vcount == 550) begin
               state <= LOAD_DATA;
               started_division <= 1;

            end else started_division <= 0;
```

```
            end

         //load data- pass in sums and count to dividend and divisor and assert star
t for a clock cycle
         LOAD_DATA : begin
               if (started_division) begin

                  start <= 1;

                  x_sum_g <= x_sum_green;
                  y_sum_g <= y_sum_green;
                  count_avg_g<= count_green;

                  x_sum_b <= x_sum_blue;
                  y_sum_b <= y_sum_blue;
                  count_avg_b<= count_blue;

                  started_division <= 0;
                  state <= READY_X_READY_Y;
               end else start <= 0;
            end

         // wait for divisions to end
         // set averages to 0 if count is less than 300 (most likely due to noise)
         READY_X_READY_Y : begin
               start <= 0;
               if (ready_x_g) begin
                  if (count_green < 300) x_avg_green <= 0;
                  else begin
                     new_car <= 1;
                     x_avg_green <= x_quotient_g;
                  end
                  x_done_g <= 1;
               end
               if (ready_y_g) begin
                  if (count_green < 300) y_avg_green <= 0;
                  else begin
                     new_car <= 1;
                     y_avg_green <= y_quotient_g;
                  end
                  y_done_g <= 1;
               end
               if (ready_x_b) begin
                  if (count_blue < 300) x_avg_blue <= 0;
                  else begin
                     new_car <= 1;
                     x_avg_blue <= x_quotient_b;
                  end
                  x_done_b <= 1;
               end

               if (ready_y_b) begin
                  if (count_blue < 300) y_avg_blue <= 0;
                  else begin
                     new_car <= 1;
                     y_avg_blue <= y_quotient_b;

                  end

                  y_done_b <= 1;
               end
               if (x_done_g && y_done_g && x_done_b && y_done_b) begin
                  state <= BOTH_DONE;
                  end_frame <= 1;
               end
            end

         // all averages calculated so we restart the FSM
         BOTH_DONE : begin

                  state <= RESET;
                  started_division <= 1;
```

```
                  x_done_g <= 0;
                  y_done_g <= 0;
                  x_done_b <= 0;
                  y_done_b <= 0;

            end

            default : state <= RESET;
      endcase
   end

         //instantiate divider module for x and y sums for two different colors to h
appen in parallel
         divider divider_module(.clk(clk), .start(start), .sign(1'b0), .dividend(x_s
um_g), .divider(count_avg_g),
         .quotient(x_quotient_g), .remainder(remainder_x_g), .ready(ready_x_g));

         divider divider_module2(.clk(clk), .start(start), .sign(1'b0), .dividend(y_
sum_g), .divider(count_avg_g),
         .quotient(y_quotient_g), .remainder(remainder_y_g), .ready(ready_y_g));

         divider divider_module3(.clk(clk), .start(start), .sign(1'b0), .dividend(x_
sum_b), .divider(count_avg_b),
         .quotient(x_quotient_b), .remainder(remainder_x_b), .ready(ready_x_b));

         divider divider_module4(.clk(clk), .start(start), .sign(1'b0), .dividend(y_
sum_b), .divider(count_avg_b),
         .quotient(y_quotient_b), .remainder(remainder_y_b), .ready(ready_y_b));



endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Kevin Zheng Class of 2012
//           Dept of Electrical Engineering &  Computer Science
//
// Create Date:    18:45:01 11/10/2010
// Design Name:
// Module Name:    rgb2hsv
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module rgb2hsv(clock, reset, r, g, b, h, s, v);
                input wire clock;
                input wire reset;
                input wire [7:0] r;
                input wire [7:0] g;
                input wire [7:0] b;
                output reg [7:0] h;
                output reg [7:0] s;
                output reg [7:0] v;
                reg [7:0] my_r_delay1, my_g_delay1, my_b_delay1;
                reg [7:0] my_r_delay2, my_g_delay2, my_b_delay2;
                reg [7:0] my_r, my_g, my_b;
                reg [7:0] min, max, delta;
                reg [15:0] s_top;
                reg [15:0] s_bottom;
                reg [15:0] h_top;
                reg [15:0] h_bottom;
                wire [15:0] s_quotient;
                wire [15:0] s_remainder;
                wire s_rfd;
                wire [15:0] h_quotient;
                wire [15:0] h_remainder;
                wire h_rfd;
                reg [7:0] v_delay [19:0];
                reg [18:0] h_negative;
                reg [15:0] h_add [18:0];
                reg [4:0] i;
                // Clocks 4-18: perform all the divisions
                //the s_divider (16/16) has delay 18
                //the hue_div (16/16) has delay 18

                divider hue_div1(
                .clk(clock),
                .dividend(s_top),
                .divider(s_bottom),
                .quotient(s_quotient),
                // note: the "fractional" output was originally named "remainder" in
 this
                // file -- it seems coregen will name this output "fractional" even
if
                // you didn't select the remainder type as fractional.
                .remainder(s_remainder),
                .ready(s_rfd)
                );
                divider hue_div2(
                .clk(clock),
                .dividend(h_top),
                .divider(h_bottom),
                .quotient(h_quotient),
                .remainder(h_remainder),
                .ready(h_rfd)
                );
```

```
                always @ (posedge clock) begin

                        // Clock 1: latch the inputs (always positive)
                        {my_r, my_g, my_b} <= {r, g, b};

                        // Clock 2: compute min, max
                        {my_r_delay1, my_g_delay1, my_b_delay1} <= {my_r, my_g, my_b
};

                        if((my_r >= my_g) && (my_r >= my_b)) //(B,S,S)
                                max <= my_r;
                        else if((my_g >= my_r) && (my_g >= my_b)) //(S,B,S)
                                max <= my_g;
                        else    max <= my_b;

                        if((my_r <= my_g) && (my_r <= my_b)) //(S,B,B)
                                min <= my_r;
                        else if((my_g <= my_r) && (my_g <= my_b)) //(B,S,B)
                                min <= my_g;
                        else
                                min <= my_b;

                        // Clock 3: compute the delta
                        {my_r_delay2, my_g_delay2, my_b_delay2} <= {my_r_delay1, my_
g_delay1, my_b_delay1};
                        v_delay[0] <= max;
                        delta <= max - min;

                        // Clock 4: compute the top and bottom of whatever divisions
 we need to do
                        s_top <= 8'd255 * delta;
                        s_bottom <= (v_delay[0]>0)?{8'd0, v_delay[0]}: 16'd1;


                        if(my_r_delay2 == v_delay[0]) begin
                                h_top <= (my_g_delay2 >= my_b_delay2)?(my_g_delay2 -
 my_b_delay2) * 8'd255:(my_b_delay2 - my_g_delay2) * 8'd255;
                                h_negative[0] <= (my_g_delay2 >= my_b_delay2)?0:1;
                                h_add[0] <= 16'd0;
                        end
                        else if(my_g_delay2 == v_delay[0]) begin
                                h_top <= (my_b_delay2 >= my_r_delay2)?(my_b_delay2 -
 my_r_delay2) * 8'd255:(my_r_delay2 - my_b_delay2) * 8'd255;
                                h_negative[0] <= (my_b_delay2 >= my_r_delay2)?0:1;
                                h_add[0] <= 16'd85;
                        end
                        else if(my_b_delay2 == v_delay[0]) begin
                                h_top <= (my_r_delay2 >= my_g_delay2)?(my_r_delay2 -
 my_g_delay2) * 8'd255:(my_g_delay2 - my_r_delay2) * 8'd255;
                                h_negative[0] <= (my_r_delay2 >= my_g_delay2)?0:1;
                                h_add[0] <= 16'd170;
                        end

                        h_bottom <= (delta > 0)?delta * 8'd6:16'd6;


                        //delay the v and h_negative signals 18 times
                        for(i=1; i<19; i=i+1) begin
                                v_delay[i] <= v_delay[i-1];
                                h_negative[i] <= h_negative[i-1];
                                h_add[i] <= h_add[i-1];
                        end

                        v_delay[19] <= v_delay[18];
                        //Clock 22: compute the final value of h
                        //depending on the value of h_delay[18], we need to subtract
 255 from it to make it come back around the circle
                        if(h_negative[18] && (h_quotient > h_add[18])) begin
                                h <= 8'd255 - h_quotient[7:0] + h_add[18];
                        end
                        else if(h_negative[18]) begin
                                h <= h_add[18] - h_quotient[7:0];
                        end
```

```
                    else begin
                            h <= h_quotient[7:0] + h_add[18];
                    end

                    //pass out s and v straight
                    s <= s_quotient;
                    v <= v_delay[19];
            end
endmodule
```

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Jessica Quaye
//
// Create Date:    18:16:14 12/05/2018
// Design Name:
// Module Name:    sound
//////////////////////////////////////////////////////////////////////////////////


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

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

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

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

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

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

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

```verilog
  // generate repeating sequence of read/writes to AC97 registers
  ac97commands cmds(.clock(clock_27mhz), .ready(ready),
                    .command_address(command_address),
                    .command_data(command_data),
                    .command_valid(command_valid),
                    .volume(volume),
                    .source(3'b000));     // mic
endmodule

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

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

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

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

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

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

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

    // Latch user data at the end of each frame. This ensures that the
    // first frame after reset will be empty.
    if (bit_count == 255) begin
      l_cmd_addr <= {command_address, 12'h000};
```

```
      l_cmd_data <= {command_data, 4'h0};
      l_cmd_v <= command_valid;
      l_left_data <= left_data;
      l_left_v <= left_valid;
      l_right_data <= right_data;
      l_right_v <= right_valid;
    end

    if ((bit_count >= 0) && (bit_count <= 15))
      // Slot 0: Tags
      case (bit_count[3:0])
        4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
        4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
        4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
        4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
        4'h4: ac97_sdata_out <= l_right_v; // Right data valid
        default: ac97_sdata_out <= 1'b0;
      endcase
    else if ((bit_count >= 16) && (bit_count <= 35))
      // Slot 1: Command address (8-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;
    else if ((bit_count >= 36) && (bit_count <= 55))
      // Slot 2: Command data (16-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;
    else if ((bit_count >= 56) && (bit_count <= 75)) begin
      // Slot 3: Left channel
      ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
      l_left_data <= { l_left_data[18:0], l_left_data[19] };
    end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
      ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

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

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

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

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

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

```
  wire [4:0] vol;
  assign vol = 31-volume;  // convert to attenuation

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

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



module recorder(
  input wire clock,                 // 27mhz system clock
  input wire reset,                 // 1 to reset to initial state
  input wire play_sound,              // 1 for playback, 0 for record
  input wire ready,                 // 1 when AC97 data is available
  input wire [7:0] from_ac97_data,  // 8-bit PCM data from mic
  output reg [7:0] to_ac97_data     // 8-bit PCM data to headphone
);

        // read sound bits from rom address and send to ac97 module
        reg[17:0] sound_addr = 0;
        wire[7:0] sound_bits;
        wire signed [7:0] signed_sound_bits;
        sound_coe  rom1(.clka(clock), .addra(sound_addr), .douta(sound_bits));

        assign signed_sound_bits = {1'b0, sound_bits} - 128;

   always @ (posedge clock) begin
      if (ready) begin

                 // get here when we've just received new data from the AC97
                 to_ac97_data <= play_sound ? signed_sound_bits : 8'd0 ;

                         if (play_sound == 1) begin
                                if (sound_addr == 18'd113_219) sound_addr <= 18'd100
00;
                                else sound_addr <= sound_addr + 1;
                        end
      end //ready
   end //always @
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Jessica Quaye
//
// Create Date:    14:58:18 12/05/2018
// Design Name:
// Module Name:    video
//////////////////////////////////////////////////////////////////////////////////
module video( input clk,
               input one_hz_enable,
               input[23:0] visualization_pixel,
               input [10:0] hcount,
               input [9:0] vcount,
               input read_control,
               output [23:0] video_pixel,
               output ram1_we_b,
               output[18:0] ram1_address,
               inout [35:0] ram1_data,
               output ram1_cen_b,
               output reg use_video_pixel
    );

   //WRITE TO ZBT
   wire write_vram_we;
   wire [18:0] write_vram_addr;
   wire [35:0] vram_read_data;
   wire [35:0] vram_write_data;
   wire [1:0] write_state;
   wire out_write_position;
   wire now_read;

   write_to_zbt writing_section(.clk(clk), .one_hz_enable(one_hz_enable), .visualiza
tion_pixel(visualization_pixel),
                           .write_state(write_state), .read_control(read_control),
                           .out_write_position(out_write_position),
                           .we(write_vram_we), .addr_wire(write_vram_addr), .write_d
ata(vram_write_data),
                           .hcount(hcount), .vcount(vcount),
                           .now_read(now_read));


   //READ FROM ZBT
   wire start_read;
   wire read_vram_we;
   wire [18:0]read_vram_addr;
   wire read_state;
   wire out_reading_state;

   read_from_zbt reading_section(.clk(clk), .pixel_out(video_pixel), .start_read(rea
d_control), .one_hz_enable(one_hz_enable),
            .data_in(vram_read_data), .we(read_vram_we), .addr_out(read_vram_addr),
.hcount(hcount), .vcount(vcount),
            .read_state(read_state) , .out_reading_state(out_reading_state), .right(
right));


   //INTERFACE WITH STAFF ZBT MODULE
   reg[18:0] vram_addr;
  reg zbt_vram_we;

  //use read_control and now_read signal to determine when to read from memory
   always @(posedge clk) begin
      if (read_control == 1) begin
            //set everything to reading state
            if (now_read == 1) begin
               vram_addr <= read_vram_addr;
               zbt_vram_we <= 0;
               use_video_pixel <= 1;
            end
      end

      else begin
```

```
         vram_addr <= write_vram_addr;
         zbt_vram_we <= 1;
         use_video_pixel <= 0;
      end

   end

   zbt_6111 zbt1(.clk(clk), .cen(1'b1), .we(zbt_vram_we), .addr(vram_addr), .write_d
ata(vram_write_data), //REPLACE VRAM WRITE DATA
   .read_data(vram_read_data), .ram_we_b(ram1_we_b), .ram_address(ram1_address), .ra
m_data(ram1_data), .ram_cen_b(ram1_cen_b));


endmodule //end of video module

module write_to_zbt(input clk, input one_hz_enable, input[23:0] visualization_pixel,
 input read_control,
                     output reg we, output[18:0] addr_wire, output reg[35:0] write_d
ata, output [1:0] write_state,
                     output out_write_position,
                     output reg now_read,
                     input [10:0] hcount,
                     input [9:0] vcount);

      reg first = 0;
      reg [18:0] addr_counter;
      reg placeholder;

      `include "params.v"

      reg[1:0] state;
      parameter IDLE = 2'd0;
      parameter WRITING = 2'd1;

      reg write_position = 0;
      parameter FIRST = 0;
      parameter SECOND = 1;

      reg [18:0] addr = {19{1'b1}};

      always @ (posedge clk) begin
//         //have a reg that assigns 1 when one_hz_enable == 1 and don't turn it off
 until idle has seen it
         if (one_hz_enable == 1) placeholder <= 1;
         case(state)
            IDLE:
            begin
               if (read_control == 1) now_read <= 1;

               else begin
                  if ((placeholder == 1) && (hcount == 0) && (vcount == 0)) begin
                     we <= 0;
                     placeholder <= 0;
                     state <= WRITING; //each second, record a frame till the buffer
 is full
                     addr_counter <= 19'b0;
                     write_position <= FIRST;
                     now_read <= 0;
                  end
               end
            end

            WRITING:
            begin
               if (hcount[1:0] == 2'b00 && vcount[1:0] == 2'b00  && (hcount < 'd1024
) && (vcount < 'd768)) begin
                  case(write_position)
                  FIRST:
                     begin
                        write_data[35:18] <= {visualization_pixel[23:18],visualizati
on_pixel[15:10],visualization_pixel[7:2]} ; //write 18 pixels of data to the lhs
                        write_position <= SECOND;
                        we <= 1'b0;
```

```
                    end //end of first
                 SECOND:
                    begin
                       write_data[17:0] <= {visualization_pixel[23:18],visualizatio
n_pixel[15:10],visualization_pixel[7:2]} ;
                       write_position <= FIRST; //go to first because you need to h
ave data =(first, second)
                       we <= 1'b1; //send a write enable because we have a full add
ress

                       //at end of one frame, move to idle state and wait for
                       //one second before coming to write another frame
                       if (addr_counter == (NUM_LINES_PER_FRAME - 1)) begin //keep
track of address of multiple 24576 = 256 * 192  / (2  pixels per line)
                             addr_counter <= 0;
                             state <= IDLE;
                       end

                       else addr_counter <= addr_counter + 1;

                       //wrap around address when you hit the end of 20 frames
                       if (addr == NUM_FRAMES_X_LINES) addr <= 0; //24576*20frames
                       else addr <= addr + 1; //increment address by 1


                    end //end of SECOND
                    endcase
              end // if hcount and vcount are a multiple of 4


           end //end of WRITING state
        endcase //end of state machine
     end //end always

     assign write_state = state;
     assign out_write_position = write_position;

     assign addr_wire = addr;

endmodule //write_to_zbt

module read_from_zbt(input clk, output reg[23:0] pixel_out, input start_read, input
one_hz_enable,
                 input[35:0] data_in, output we, output reg[18:0] addr_out,
                 input [10:0] hcount,
                   input [9:0] vcount,
                 input right,
                 output read_state,
                 output out_reading_state);

     `include "params.v"

     assign we = 0; //indicate we are reading
      reg place_holder;

      reg state = 0;
     parameter INITIAL = 0;
     parameter READING = 1;

      reg[2:0] reading_state = 3'b0;
     parameter FIRST = 3'd0;
     parameter SECOND = 3'd1;

     reg [18:0] addr_counter;
    reg [5:0] frames_so_far = 6'b0;
      reg [18:0] addr_held_for_align;

     always @ (posedge clk) begin
        if (one_hz_enable == 1) place_holder <= 1;

        case(state)
           INITIAL: begin
              addr_out <= 19'd0;
```

```
                pixel_out <= 24'hFF_FF_FF;
                if ((start_read == 1) && (hcount == 0) && (vcount == 0))begin
                   //wait for another cycle before moving to read because you need 2
cycles of delay and this applies
                   state <= READING;
                end
             end

             READING: begin
                case(reading_state)
                   FIRST: begin
   //                   read_data[35:18]is 18 bits so append 0s after each 6
                      pixel_out <= {{data_in[35:30], 2'b0} , {data_in[29:24], 2'b0},
{data_in[23:18], 2'b0}};
                      reading_state <= SECOND;
                   end

                   SECOND: begin
   //                 read_data[17:0]is 18 bits so append 0s after each 6 bits
                      pixel_out <= {{data_in[17:12], 2'b0} , {data_in[11:6], 2'b0}, {
data_in[5:0], 2'b0}};

                      if ((hcount < FRAME_WIDTH) && (vcount < FRAME_HEIGHT)) begin //
if within region && hcount is even
                            if (addr_counter == (NUM_LINES_PER_FRAME - 1)) begin
                               if ((place_holder == 1)) begin
                                  place_holder <= 0;
                                  addr_out <= addr_out + 1;
                                  addr_counter <= 0;
                                  reading_state <= FIRST;

                               end

                               else begin
                                  addr_out <= addr_out - (NUM_LINES_PER_FRAME -
 1);

                                  addr_counter <=0;
                                  reading_state <= FIRST;
                               end
                            end

                            else begin //otherwise, it's business as usual. increm
ent address by 1 and progress
                               addr_counter <= addr_counter + 1;
                               addr_out <= addr_out + 1; //increment address by 1
                               reading_state <= FIRST;
                            end

                      end //hcount == FRAME WIDTH

                      else pixel_out <= 24'd0;

                   end //end SECOND

                endcase// endcase for reading state

                if (start_read == 0) state <= INITIAL;
             end //end READING

          endcase
       end //end always

       assign read_state = state;
       assign out_reading_state = reading_state;

endmodule //read_from_zbt
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Jessica Quaye
//
// Create Date:    19:17:09 11/12/2018
// Design Name:
// Module Name:    visualization
//////////////////////////////////////////////////////////////////////////////////


//draws vertical line on the screen
module vertical_line
        #(parameter COLOR = 24'hFF_FF_FF)
         (input [10:0] x,hcount,
    input [9:0] y,vcount,
         input [9:0] y_length,
         input [9:0] thickness,
    output reg [23:0] pixel);

         always @ * begin
                 if ( (hcount >= x && hcount <= (x+thickness)) && (vcount >= y && vc
ount <= (y + y_length + thickness)) ) pixel = COLOR; //we are at the same x but same
 or greater y
                 else pixel = 0;
         end //end always
endmodule

//draws dotted vertical line on screen
module vertical_dotted
        #(parameter WIDTH = 3,
                COLOR = 24'hFF_FF_FF)
        (input [10:0] x,hcount,
    input [9:0] y,vcount,
    output reg [23:0] pixel);

         always @ * begin
                if ((hcount >= x && hcount < (x+WIDTH)) &&
                   (vcount >= 0 && vcount <= 64 |
                         vcount >= 128 && vcount <= 192 |
                         vcount >= 256 && vcount <= 320 |
                         vcount >= 512 && vcount <= 576 |
                         vcount >= 640 && vcount <= 704 )) pixel = COLOR;
                else pixel = 0;
         end //end always

endmodule

//draws horizontal line on screen
module horizontal_line
        #(parameter COLOR = 24'hFF_FF_FF)
         (input [10:0] x,hcount,
    input [9:0] y,vcount,
         input [10:0] x_length,
         input [10:0] thickness,
    output reg [23:0] pixel);

         always @ * begin
                 if ((vcount >= y && vcount <= (y + thickness)) && (hcount >= x && h
count <= (x + x_length + thickness))) pixel = COLOR; //we are at the same x but same
 or greater y
                 else pixel = 0;
         end //end always
endmodule

//draws dotted horizontal line on screen
module horizontal_dotted
        #(parameter HEIGHT = 10,
                COLOR = 24'hFF_FF_FF)
        (input [10:0] x,hcount,
    input [9:0] y,vcount,
    output reg [23:0] pixel);
```

```
        always @ * begin
               if ((vcount >= y && vcount < (y+HEIGHT)) &&
                  (hcount >= 0 && hcount <= 128 |
                        hcount >= 256 && hcount <= 384 |
                        hcount >= 620 && hcount <= 748 |
                        hcount >= 876 && hcount <= 1004 )) pixel = COLOR;
               else pixel = 0;
        end //end always

endmodule

//draws traffic light given color that should be turned on
module draw_traffic_light
        (input clk,
         input [10:0] x,hcount,
    input [9:0] y,vcount,
         input [1:0] signal,
         input orientation,
    output reg [23:0] traffic_pixel);

         `include "params.v"

         reg [15:0] image_addr; //num of bits for 8*6000 ROM
         wire [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

        //vertical w = 68, h = 180

        wire[10:0] WIDTH = (orientation == VERTICAL) ?  11'd68 : 11'd180 ;
        wire[9:0]HEIGHT = (orientation == VERTICAL) ?  10'd180 : 10'd68;

        always @ (posedge clk) begin
                case(orientation)
                        VERTICAL: begin
                                image_addr <= (hcount-x + 3) + (vcount-y) * WIDTH;
                                end
                        HORIZONTAL: begin //rotate 90deg
                                image_addr <= (HEIGHT - (vcount-y)) +  (WIDTH- (hcou
nt - x))* HEIGHT;
                                end
                endcase
        end

         traffic_image_rom  traffic_rom(.clka(clk), .addra(image_addr), .douta(image
_bits));

   // use color map to create 8bits R, 8bits G, 8 bits B;
   traffic_red_coe traffic_rcm (.clka(clk), .addra(image_bits), .douta(red_mapped));
   traffic_green_coe traffic_gcm (.clka(clk), .addra(image_bits), .douta(green_mappe
d));
   traffic_blue_coe traffic_bcm (.clka(clk), .addra(image_bits), .douta(blue_mapped)
);

        always @(posedge clk) begin

      if ((hcount >= x && hcount < (x+WIDTH)) &&
          (vcount >= y && vcount < (y+HEIGHT)))
                                begin
                                if (signal == RED) begin
                                        //yellow off and green off
                                        if ((red_mapped > 8'd200) && (green_mapped >
 8'd100)) traffic_pixel <= {8'd170, 8'd170, 8'd170}; //yellow off
                                        else if (green_mapped > 8'd150) traffic_pixe
l <= {8'd170, 8'd170, 8'd170}; //green off
                                        else traffic_pixel <= {red_mapped, green_map
ped, blue_mapped};
                                end

                                if (signal == YELLOW) begin
                                        //red off and green off
                                        if ((red_mapped > 8'd200) && (green_mapped <
 8'd90)) traffic_pixel <= {8'd170, 8'd170, 8'd170}; //red off
                                        else if (green_mapped > 8'd150 && (red_mappe
d < 8'd90)) traffic_pixel <= {8'd170, 8'd170, 8'd170}; //green off
```

```
                                                else traffic_pixel <= {red_mapped, green_map
ped, blue_mapped};
                                         end

                                         if (signal == GREEN) begin
                                                //red off, yellow off
                                                if ((red_mapped > 8'd200) && (green_mapped <
 8'd100)) traffic_pixel <= {8'd170, 8'd170, 8'd170}; // red off
                                                else if ((red_mapped > 8'd200) && (green_map
ped > 8'd100)) traffic_pixel <= {8'd170, 8'd170, 8'd170}; //yellow off
                                                else traffic_pixel <= {red_mapped, green_map
ped, blue_mapped};
                                         end
                                  end
     else traffic_pixel <= 0;
        end


endmodule

//draws street on screen
module draw_street
( input clk,
  input[10:0]hcount,
  input[9:0] vcount,
  output [23:0] street_pixel);

  `include "params.v"

         //DRAW MAIN ROAD
         //generate left line
                wire [23:0] left_line_pixel;
                vertical_line #(.COLOR(24'hFF_FF_FF))
                  left_line(.x(11'd424),.y(10'd0),.hcount(hcount),.vcount(vcount), .
y_length(SCREEN_Y_LIMIT), .thickness(11'd0),//x = (1024/2) - width/2, y = (768/2) -
height/2
                                                   .pixel(left_line_pixel));

        //generate right line
                wire [23:0] right_line_pixel;
                vertical_line #(.COLOR(24'hFF_FF_FF))
                  right_line(.x(11'd600),.y(10'd0),.hcount(hcount),.vcount(vcount),
.y_length(SCREEN_Y_LIMIT), .thickness(11'd0),//x = (1024/2) - width/2, y = (768/2) -
 height/2
                                                   .pixel(right_line_pixel));

        //generate mid dotted vertical line
        wire [23:0] vert_mid_dot_pixel;
        vertical_dotted #(.WIDTH(4), .COLOR(24'hFF_FF_FF))
                mid_dot(.x(11'd512),.y(10'd0),.hcount(hcount),.vcount(vcount), //x =
 (right-left/2) - width/2
                                                   .pixel(vert_mid_dot_pixel));

        //DRAW SIDE ROAD
        //generate top line
        wire [23:0] top_line_pixel;
        horizontal_line #(.COLOR(24'hFF_FF_FF))
                 top_line(.x(11'd0),.y(10'd344),.hcount(hcount),.vcount(vcount), .x_
length(SCREEN_X_LIMIT), .thickness(11'd0),//y = (right-left/2) - width/2
                                                   .pixel(top_line_pixel));

        //generate bottom line
        wire [23:0] bottom_line_pixel;
        horizontal_line #(.COLOR(24'hFF_FF_FF))
                 bottom_line(.x(11'd0),.y(10'd464),.hcount(hcount),.vcount(vcount),
.x_length(SCREEN_X_LIMIT), .thickness(11'd0),//y = (right-left/2) - width/2
                                                   .pixel(bottom_line_pixel));

        //generate mid dotted horizontal line
        wire [23:0] horiz_mid_dot_pixel;
        horizontal_dotted #(.HEIGHT(4),.COLOR(24'hFF_FF_FF))
                 horiz_mid_dot(.x(11'd0),.y(10'd402),.hcount(hcount),.vcount(vcount)
, //y = (top-bottom/2) - height/2
```

```
                                        .pixel(horiz_mid_dot_pixel));

        assign street_pixel = left_line_pixel | vert_mid_dot_pixel | right_line_pixe
l | top_line_pixel | bottom_line_pixel | horiz_mid_dot_pixel;
endmodule

//draws a car given its color
module draw_car
        #(parameter COLOR = 24'hFF_00_00)
   (input clk,
         input [10:0] x,hcount, width,
    input [9:0] y,vcount, height,
         input [1:0] car_direction,
         input is_ambulance,
    output reg [23:0] pixel);

        (* ram_style =  "registers" *) reg [15:0] image_addr, amb_addr; //num of bit
s for 8*6000 ROM
        wire [7:0] image_bits, red_mapped, green_mapped, blue_mapped;
        wire [7:0] amb_bits, amb_red_mapped, amb_green_mapped, amb_blue_mapped;

        `include "params.v"

        always @ (posedge clk) begin
                case(car_direction)
                        MOVE_RIGHT: begin
                                amb_addr <= (hcount-x + 3) + (vcount-y) * width;
                                image_addr <= (hcount-x + 3) + (vcount-y) * width;
                                end
                        MOVE_LEFT: begin
                                amb_addr <= (width-(hcount-x) - 4) +  (vcount-y) * w
idth;
                                image_addr <= (width-(hcount-x) - 4) +  (vcount-y) *
 width;
                                end
                        MOVE_UP: begin
                                amb_addr <= (height - (vcount-y)) +  (width- (hcount
 - x))* height;
                                image_addr <= (height - (vcount-y)) +  (width- (hcou
nt - x))* height;
                                end
                        MOVE_DOWN: begin
                                amb_addr <= (vcount-y) +  (hcount - x)* height;
                                image_addr <= (vcount-y) +  (hcount - x)* height;
                                end
                endcase
         end

         smaller_car  car_rom(.clka(clk), .addra(image_addr), .douta(image_bits)); /
/CHANGE
         amb_image_rom  amb_rom(.clka(clk), .addra(amb_addr),   .douta(amb_bits));

   // use color map to create 8bits R, 8bits G, 8 bits B;
   smaller_car_red_coe car_rcm (.clka(clk), .addra(image_bits), .douta(red_mapped));
 //CHANGE
   smaller_car_green_coe car_gcm (.clka(clk), .addra(image_bits), .douta(green_mappe
d)); //CHANGE
   smaller_car_blue_coe car_bcm (.clka(clk), .addra(image_bits), .douta(blue_mapped)
); //CHANGE

        //use color map to create 8bits R, 8bits G, 8 bits B;
        amb_red_coe amb_rcm(.clka(clk), .addra(amb_bits), .douta(amb_red_mapped));
        amb_green_coe amb_gcm (.clka(clk), .addra(amb_bits), .douta(amb_green_mapped
));
   amb_blue_coe amb_bcm (.clka(clk), .addra(amb_bits), .douta(amb_blue_mapped));

        always @(posedge clk) begin
     if ((hcount >= x && hcount < (x+width)) &&
          (vcount >= y && vcount < (y+height)))
                          begin
                                if (is_ambulance == TRUE) pixel <= {amb_red_mapped,
amb_green_mapped, amb_blue_mapped};
                                else begin
```

```
                                                  if ((x > 0) && (y >0))begin
                                                          if ((red_mapped > 'd60) && (green_ma
pped < 'd50) && (blue_mapped < 50)) pixel <= COLOR;
                                                          else pixel <= {red_mapped, green_map
ped, blue_mapped};
                                                  end
                                                  else pixel <= 0;
                                          end
                                   end
     else pixel <= 0;
        end

endmodule

module visualization(
    input vclock,
         input one_hz_enable,
    input[10:0] hcount,
    input[9:0] vcount,
    input hsync, vsync, blank,
    input [1:0] main_out,
    input [1:0] side_out,
    input [1:0] car1_direction,car2_direction, car3_direction, car4_direction, car5_
direction,
                       car6_direction, car7_direction, car8_direction, car9_direction,
 car10_direction,
    input[1:0] car11_direction, car12_direction, car13_direction,
    input [1:0] ambulance_direction,
    input [10:0] car1_leftx,car2_leftx, car3_leftx,car4_leftx, car5_leftx, car6_left
x, car7_leftx, car8_leftx, car9_leftx, car10_leftx,
    input [10:0] car11_leftx, car12_leftx, car13_leftx,
    input [10:0] car1_width, car2_width, car3_width, car4_width, car5_width, car6_wi
dth, car7_width, car8_width, car9_width, car10_width,
    input [10:0] car11_width, car12_width, car13_width,
    input [9:0] car1_topy, car2_topy, car3_topy, car4_topy, car5_topy, car6_topy, ca
r7_topy, car8_topy, car9_topy, car10_topy,
    input [9:0] car11_topy, car12_topy, car13_topy,
    input [9:0] car1_height,car2_height, car3_height, car4_height, car5_height,car6_
height, car7_height, car8_height, car9_height, car10_height,
    input [9:0] car11_height,car12_height, car13_height,
    input is_collision,
    input[10:0] ambulance_leftx, ambulance_width,
    input[9:0] ambulance_topy, ambulance_height,
    output viz_hsync,
    output viz_vsync,
    output viz_blank,
    output[23:0] pixel
    );

    assign viz_hsync = hsync;
    assign viz_vsync = vsync;
    assign viz_blank = blank;

    `include "params.v"

       //draw street
        wire [23:0] street_pixel;
        draw_street street1(.clk(vclock),.hcount(hcount),.vcount(vcount),
                                         .street_pixel(street_pixel));

        //draw main traffic light
        wire [23:0] traffic1_pixel;
        draw_traffic_light traffic1(.clk(vclock),.x(320),.y(0),.hcount(hcount),.vcou
nt(vcount), .signal(main_out), .orientation(VERTICAL),
                                         .traffic_pixel(traffic1_pixel));

        //draw side traffic light
        wire [23:0] traffic2_pixel;
        draw_traffic_light traffic2(.clk(vclock),.x(624),.y(468),.hcount(hcount),.vc
ount(vcount), .signal(side_out), .orientation(HORIZONTAL),
                                         .traffic_pixel(traffic2_pixel));
```

```
        //draw car - GREEN
        wire[23:0] car1_pixel;
        draw_car #(.COLOR(24'h00_FF_00))
        car1(.clk(vclock), .x(car1_leftx), .y(car1_topy), .hcount(hcount), .vcount(v
count), .height(car1_height), .width(car1_width), .pixel(car1_pixel), .car_direction
(car1_direction), .is_ambulance(FALSE));

        //draw car - GREEN
        wire[23:0] car2_pixel;
        draw_car #(.COLOR(24'h00_FF_00))
        car2(.clk(vclock), .x(car2_leftx), .y(car2_topy), .hcount(hcount), .vcount(v
count), .height(car2_height), .width(car2_width), .pixel(car2_pixel), .car_direction
(car2_direction), .is_ambulance(FALSE));


        //draw car - GREEN
        wire[23:0] car3_pixel;
        draw_car #(.COLOR(24'h00_FF_00))
        car3(.clk(vclock), .x(car3_leftx), .y(car3_topy), .hcount(hcount), .vcount(v
count), .height(car3_height), .width(car3_width), .pixel(car3_pixel), .car_direction
(car3_direction), .is_ambulance(FALSE));

        //draw car - BLUE
        wire[23:0] car4_pixel;
        draw_car #(.COLOR(24'h00_00_FF))
         car4(.clk(vclock), .x(car4_leftx), .y(car4_topy), .hcount(hcount), .vcount(
vcount), .height(car4_height), .width(car4_width), .pixel(car4_pixel), .car_directio
n(car4_direction), .is_ambulance(FALSE));

        //draw car - BLUE
        wire[23:0] car5_pixel;
        draw_car #(.COLOR(24'h00_00_FF))
         car5(.clk(vclock), .x(car5_leftx), .y(car5_topy), .hcount(hcount), .vcount(
vcount), .height(car5_height), .width(car5_width), .pixel(car5_pixel), .car_directio
n(car5_direction), .is_ambulance(FALSE));

        //draw car - BLUE
        wire[23:0] car6_pixel;
        draw_car #(.COLOR(24'h00_00_FF))
        car6(.clk(vclock), .x(car6_leftx), .y(car6_topy), .hcount(hcount), .vcount(v
count), .height(car6_height), .width(car6_width), .pixel(car6_pixel), .car_direction
(car6_direction), .is_ambulance(FALSE));

        //draw car - GREEN
        wire[23:0] car7_pixel;
        draw_car #(.COLOR(24'h00_FF_00))
        car7(.clk(vclock), .x(car7_leftx), .y(car7_topy), .hcount(hcount), .vcount(v
count), .height(car7_height), .width(car7_width), .pixel(car7_pixel), .car_direction
(car7_direction), .is_ambulance(FALSE));

        //draw car - GREEN
        wire[23:0] car8_pixel;
        draw_car #(.COLOR(24'h00_FF_00))
        car8(.clk(vclock), .x(car8_leftx), .y(car8_topy), .hcount(hcount), .vcount(v
count), .height(car8_height), .width(car8_width), .pixel(car8_pixel), .car_direction
(car8_direction), .is_ambulance(FALSE));

        //draw car - GREEN
        wire[23:0] car9_pixel;
        draw_car #(.COLOR(24'h00_FF_00))
        car9(.clk(vclock), .x(car9_leftx), .y(car9_topy), .hcount(hcount), .vcount(v
count), .height(car9_height), .width(car9_width), .pixel(car9_pixel), .car_direction
(car9_direction), .is_ambulance(FALSE));

        //draw car - GREEN
        wire[23:0] car10_pixel;
        draw_car #(.COLOR(24'h00_FF_00))
        car10(.clk(vclock), .x(car10_leftx), .y(car10_topy), .hcount(hcount), .vcoun
t(vcount), .height(car10_height), .width(car10_width), .pixel(car10_pixel), .car_dir
ection(car10_direction), .is_ambulance(FALSE));

        //draw car - GREEN
```

```
        wire[23:0] car11_pixel;
        draw_car #(.COLOR(24'h00_FF_00))
        car11(.clk(vclock), .x(car11_leftx), .y(car11_topy), .hcount(hcount), .vcoun
t(vcount), .height(car11_height), .width(car11_width), .pixel(car11_pixel), .car_dir
ection(car11_direction), .is_ambulance(FALSE));

        //draw car - BLUE
        wire[23:0] car12_pixel;
        draw_car #(.COLOR(24'h00_00_FF))
        car12(.clk(vclock), .x(car12_leftx), .y(car12_topy), .hcount(hcount), .vcoun
t(vcount), .height(car12_height), .width(car12_width), .pixel(car12_pixel), .car_dir
ection(car12_direction), .is_ambulance(FALSE));


        //draw car - BLUE
        wire[23:0] car13_pixel;
        draw_car #(.COLOR(24'h00_00_FF))
        car13(.clk(vclock), .x(car13_leftx), .y(car13_topy), .hcount(hcount), .vcoun
t(vcount), .height(car13_height), .width(car13_width), .pixel(car13_pixel), .car_dir
ection(car13_direction), .is_ambulance(FALSE));

        //draw ambulance
        wire [23:0] ambulance_pixel;
        draw_car ambulance(.clk(vclock), .x(ambulance_leftx), .y(ambulance_topy), .h
count(hcount), .vcount(vcount), .height(ambulance_height), .width(ambulance_width),
.pixel(ambulance_pixel), .car_direction(ambulance_direction), .is_ambulance(TRUE));

        reg [23:0] dom_pixel;

        always @ * begin
            if (is_collision == 1) dom_pixel = street_pixel | car1_pixel | car2_pixe
l | car3_pixel | car4_pixel |
                                                car5_pixel| car6_pixel| car7_pixel| c
ar8_pixel| car9_pixel | car10_pixel |
                                                car11_pixel | car12_pixel | car13_pix
el | ambulance_pixel;
            else dom_pixel = street_pixel | car1_pixel | car2_pixel | car3_pixel | c
ar4_pixel | car5_pixel |
                                    car6_pixel | car7_pixel| car8_pixel | car9_pixel | car10
_pixel | car11_pixel |
                                    car12_pixel | car13_pixel;
        end //end always

        assign pixel =  dom_pixel| traffic1_pixel | traffic2_pixel;
endmodule


//module to determine the height, width and direction of a car
module w_and_h_calc(
         input clk,
         input [10:0] car_x,
         input [9:0] car_y,
         output reg[9:0] car_height,
         output reg[10:0] car_width,
         output reg[1:0] car_direction);

        `include "params.v"

        always @ (posedge clk) begin

                //determine if orientation is vertical or horizontal
                if (car_x < STREET_LEFTX || car_x > STREET_RIGHTX) //should be horiz
ontal
                        begin
                                car_height <= 11'd39; //CHANGE
                                car_width <= 10'd80; //CHANGE

                                        //given horizontal orientation, check if mov
ing left or moving right
                                        if (STREET_TOPY < car_y && car_y < STREET_HO
RIZ_MID) car_direction <= MOVE_LEFT;
                                        else car_direction <= MOVE_RIGHT;
                        end
```

```
                //else vertical orientation
                else begin
                                car_height <= 11'd80; //CHANGE
                                car_width <= 10'd39; //CHANGE

                                        //given vertical orientation, check if movin
g up or moving down
                                        if (STREET_LEFTX < car_x && car_x < STREET_V
ERT_MID) car_direction <= MOVE_DOWN;
                                        else car_direction <= MOVE_UP;
                end
        end //end always

endmodule





////////////////////////////////
////UNUSED CODE: previously used blobs and ORed pixels before transitioning to
//COE files
//////////

//module draw_car
//   #(parameter COLOR = 24'h22_8B_22)
//   (input clk,
//       input [10:0] x,hcount, width,
//    input [9:0] y,vcount, height,
//    output [23:0] pixel);
//
//       wire [23:0] rectangle_pixel;
//
//       //draw car rectangle
//       draw_filled_rectangle #(.COLOR(COLOR))
//                                car_skeleton(.x(x), .y(y), .hcount(hcount), .vcount(
vcount), .height(height), .width(width), .pixel(rectangle_pixel));
//
//       //draw four wheels all of radius 10
//       //draw top left wheel
//       wire [23:0] top_left_wheel_pixel;
//       draw_round_puck #(.RADIUS(10), .COLOR(24'hFF_FF_FF))
//              top_left_wheel(.clk(clk), .hcount(hcount), .vcount(vcount), .x(x -10
), .y(y - 5), .pixel(top_left_wheel_pixel));
//
//      //draw top right wheel
//       wire [23:0] top_right_wheel_pixel;
//       draw_round_puck #(.RADIUS(10), .COLOR(24'hFF_FF_FF))
//              top_right_wheel(.clk(clk), .hcount(hcount), .vcount(vcount), .x(x+wi
dth-15), .y(y-5), .pixel(top_right_wheel_pixel));
//
//      //draw bottom left wheel
//       wire [23:0] bottom_left_wheel_pixel;
//       draw_round_puck #(.RADIUS(10), .COLOR(24'hFF_FF_FF))
//              bottom_left_wheel(.clk(clk), .hcount(hcount), .vcount(vcount), .x(x-
10), .y(y+height-10), .pixel(bottom_left_wheel_pixel));
//
//       //draw bottom right wheel
//       wire [23:0] bottom_right_wheel_pixel;
//       draw_round_puck #(.RADIUS(10), .COLOR(24'hFF_FF_FF))
//              bottom_right_wheel(.clk(clk), .hcount(hcount), .vcount(vcount), .x(x
+width-15), .y(y + height-10), .pixel(bottom_right_wheel_pixel));
//
//       assign pixel = rectangle_pixel | top_left_wheel_pixel | top_right_wheel_pix
el | bottom_left_wheel_pixel | bottom_right_wheel_pixel;
//endmodule

//module used to create a circle of given radius on the screen by coloring pixels wi
th given color
//module draw_round_puck
//#(parameter RADIUS = 10'd30,
//                                COLOR = 24'hFF_00_00)
```

```
//( input clk,
//  input[10:0]x, hcount,
//  input[9:0] y, vcount,
//  output reg[23:0] pixel);
//
//  reg[100:0] radiussquared;
//  reg[10:0] deltax;
//  reg[9:0] deltay;
//  reg[120:0] deltaxsquared;
//  reg[80:0] deltaysquared;
//
//    always @(posedge clk)
//    // compute x-xcenter and y-ycenter
//    begin
//    radiussquared <= RADIUS*RADIUS;
//
//     // RADIUS is a paramater
//    deltax <= (hcount > (x+RADIUS)) ? (hcount-(x+RADIUS)) : ((x+RADIUS)-hcount);
//    deltay <= (vcount > (y+RADIUS)) ? (vcount-(y+RADIUS)) : ((y+RADIUS)-vcount);
//
//     deltaxsquared <= deltax * deltax;
//     deltaysquared <= deltay * deltay;
//
//        // check if distance is less than radius squared
//    if(deltaxsquared + deltaysquared <= radiussquared) pixel <= COLOR;
//    else pixel <= 0;
//    end //end always block
//endmodule


//module used to overwrite pixels of a larger circle.
//module draw_inner_circle
//#(parameter RADIUS = 10,
//                               COLOR = 24'h00_00_00)
//( input clk,
//  input[10:0] x,hcount,
//  input[9:0] y,vcount,
//  input activate_inner,
//  input [23:0] outer_pixel,
//  output [23:0] pixel);
//
//  reg[100:0] radiussquared;
//  reg[10:0] deltax;
//  reg[9:0] deltay;
//  reg[120:0] deltaxsquared;
//  reg[80:0] deltaysquared;
//
//  //alpha blending initialization
//      wire[2:0] m = 3'b010;
//      wire[2:0] n = 3'b100;
//
//      //inner register
//      reg[23:0] internal_pixel;
//
//    always @(posedge clk)
//    // compute x-xcenter and y-ycenter
//    begin
//    radiussquared <= RADIUS*RADIUS;
//
//    // RADIUS is a paramater
//    deltax <= (hcount > (x+RADIUS)) ? (hcount-(x+RADIUS)) : ((x+RADIUS)-hcount);
//    deltay <= (vcount > (y+RADIUS)) ? (vcount-(y+RADIUS)) : ((y+RADIUS)-vcount);
//
//     deltaxsquared <= deltax * deltax;
//     deltaysquared <= deltay * deltay;
//
//    // check if distance is less than radius squared
//    if(deltaxsquared + deltaysquared <= radiussquared && activate_inner == 1)
//                         begin
//                                internal_pixel[23:16] <= (outer_pixel[23:16] * m  >>
 n) + (COLOR[23:16] * (2**n - m) >>n);
//                                internal_pixel[15:8] <= (outer_pixel[15:8] * m  >> n
) + (COLOR[15:8] * (2**n - m) >> n);
```

```
//                              internal_pixel[7:0] <= (outer_pixel[7:0] * m  >> n)
+ (COLOR[7:0] * (2**n - m) >> n);
//                       end
//        else internal_pixel <= outer_pixel; //otherwise, maintain outer pixel color

//     end //end always block
//
//       assign pixel = internal_pixel;
//
//endmodule

//module draw_traffic_light
//       #(parameter THICKNESS = 5,
//                                        HORIZONTAL_LENGTH = 90,
//                                        VERTICAL_LENGTH = 180,
//                                        STICK_LENGTH = 120,
//                                        LIGHT_RADIUS = 20,
//                                        COLOR = 24'h22_8B_22)
//       (input clk,
//        input [10:0] x,hcount,
//     input [9:0] y,vcount,
//        input [1:0] signal,
//     output [23:0] traffic_pixel);
//
//        //draw main box
//        wire [23:0] main_box_pixel;
//        draw_empty_rectangle main_box(.x(x),.y(y),.hcount(hcount),.vcount(vcount),
.thickness(THICKNESS), .vertical_length(VERTICAL_LENGTH), .horizontal_length(HORIZON
TAL_LENGTH), .pixel(main_box_pixel));
//
//        //draw support stick
//        wire [23:0] support_stick_pixel; //x = x + (horiz_len/2 - thickness/2)
//               vertical_line #(.COLOR(COLOR))
//                 support_stick(.x(x + 42),.y(y+VERTICAL_LENGTH),.hcount(hcount),.vc
ount(vcount), .y_length(STICK_LENGTH), .thickness(THICKNESS), .pixel(support_stick_p
ixel));
//
//        //determine coordinates for circles
//        wire[10:0] circle_x = x + 20;
//     wire[9:0] red_y = y + 9'd20;
//        wire[9:0] yellow_y = red_y + 9'd50;
//        wire[9:0] green_y = yellow_y + 9'd50;
//
//        //declare switches to control lights. if light should be off, activate the
inner circle
//        reg activate_inner_r;
//        reg activate_inner_y;
//        reg activate_inner_g;
//
//        //constants
//        wire[1:0] red = 2'b00;
//        wire[1:0] yellow = 2'b01;
//        wire[1:0] green = 2'b10;
//        wire on = 1;
//        wire off = 0;
//
//        always @ * begin
//               if (signal != red) activate_inner_r = on;
//               else activate_inner_r = off;
//
//               if (signal != yellow) activate_inner_y = on;
//               else activate_inner_y = off;
//
//               if (signal != green) activate_inner_g = on;
//               else activate_inner_g = off;
//        end
//
//        //DRAW PUCKS FOR TRAFFIC LIGHTS
//
//        //DRAW RED PUCK
//        wire [23:0] red_puck_pixel;
//        draw_round_puck #(.RADIUS(LIGHT_RADIUS), .COLOR(24'hFF_00_00))
//               red_puck(.clk(clk), .hcount(hcount), .vcount(vcount), .x(circle_x),
```

```
.y(red_y), .pixel(red_puck_pixel));
//
//       //IF RED LIGHT IS OFF, DRAW A DARK INNER CIRCLE
//       wire [23:0] red_inner_pixel;
//      draw_inner_circle #(.RADIUS(15))
//        r_black (.clk(clk), .hcount(hcount), .vcount(vcount), .x(circle_x + 5), .y
(red_y + 5), .activate_inner(activate_inner_r), .outer_pixel(red_puck_pixel), .pixel
(red_inner_pixel));
//
//       //DRAW YELLOW PUCK
//       wire [23:0] yellow_puck_pixel;
//       draw_round_puck #(.RADIUS(LIGHT_RADIUS), .COLOR(24'hFF_FF_00))
//              yellow_puck(.clk(clk), .hcount(hcount), .vcount(vcount), .x(circle_x
), .y(yellow_y), .pixel(yellow_puck_pixel));
//
//      //IF YELLOW LIGHT IS OFF, DRAW A DARK INNER CIRCLE
//       wire [23:0] yellow_inner_pixel;
//      draw_inner_circle #(.RADIUS(15))
//       y_black (.clk(clk), .hcount(hcount), .vcount(vcount), .x(circle_x + 5), .y(
yellow_y + 5), .activate_inner(activate_inner_y), .outer_pixel(yellow_puck_pixel), .
pixel(yellow_inner_pixel));
//
//       //DRAW GREEN PUCK
//       wire [23:0] green_puck_pixel;
//       draw_round_puck #(.RADIUS(LIGHT_RADIUS) , .COLOR(24'h00_FF_00))
//              green_puck(.clk(clk), .hcount(hcount), .vcount(vcount), .x(circle_x)
, .y(green_y),.pixel(green_puck_pixel));
//
//      //IF GREEN LIGHT IS OFF, DRAW A DARK INNER CIRCLE
//       wire [23:0] green_inner_pixel;
//      draw_inner_circle #(.RADIUS(15))
//       g_black (.clk(clk), .hcount(hcount), .vcount(vcount), .x(circle_x + 5), .y(
green_y + 5), .activate_inner(activate_inner_g), .outer_pixel(green_puck_pixel), .pi
xel(green_inner_pixel));
//
//      assign traffic_pixel = main_box_pixel | support_stick_pixel | red_inner_pixe
l | yellow_inner_pixel | green_inner_pixel;
//endmodule

//module draw_empty_rectangle
//   #(parameter COLOR = 24'h22_8B_22)
//   (input [10:0] x,hcount,
//    input [9:0] y,vcount,
//       input [10:0] thickness,
//       input [7:0] vertical_length, horizontal_length,
//    output [23:0] pixel);
//
//      //DRAW VERTICAL BARS
//      //draw left bar
//      wire [23:0] vert_left_pixel;
//              vertical_line #(.COLOR(COLOR))
//                vert_left(.x(x),.y(y),.hcount(hcount),.vcount(vcount), .y_length(v
ertical_length), .thickness(thickness), .pixel(vert_left_pixel));
//
//      //draw right bar
//      wire [23:0] vert_right_pixel;
//              vertical_line #(.COLOR(COLOR))
//                vert_right(.x(x + horizontal_length),.y(y),.hcount(hcount),.vcount
(vcount), .y_length(vertical_length), .thickness(thickness), .pixel(vert_right_pixel
));
//
//   //DRAW HORIZONTAL BARS
//      //draw top bar
//      wire [23:0] horiz_top_pixel;
//              horizontal_line #(.COLOR(COLOR))
//                      horiz_top(.x(x),.y(y),.hcount(hcount),.vcount(vcount), .x_le
ngth(horizontal_length), .thickness(thickness), .pixel(horiz_top_pixel));
//
//      //draw bottom bar
//   wire [23:0] horiz_bottom_pixel;
//              horizontal_line #(.COLOR(COLOR))
//                      horiz_bottom(.x(x),.y(y + vertical_length),.hcount(hcount),.
vcount(vcount), .x_length(horizontal_length), .thickness(thickness), .pixel(horiz_bo
```

```
ttom_pixel));
//
//   assign pixel = vert_left_pixel | vert_right_pixel | horiz_top_pixel | horiz_bot
tom_pixel;
//
//endmodule

//module draw_filled_rectangle
//   #(parameter COLOR = 24'hFF_45_00)
//   (input [10:0] x,hcount, width,
//    input [9:0] y,vcount, height,
//    output reg [23:0] pixel);
//
//       always @ * begin
//              if ( (hcount >= x && hcount < (x+width)) && (vcount >= y && vcount <
 (y+height))) pixel = COLOR;
//              else pixel = 0;
//       end
//endmodule
```

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

module YCrCb2RGB ( R, G, B, clk, rst, Y, Cr, Cb );

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

input clk,rst;
input[9:0] Y, Cr, Cb;

wire [7:0] R,G,B;
reg [20:0] R_int,G_int,B_int,X_int,A_int,B1_int,B2_int,C_int;
reg [9:0] const1,const2,const3,const4,const5;
reg[9:0] Y_reg, Cr_reg, Cb_reg;

//registering constants
always @ (posedge clk)
begin
 const1 = 10'b 0100101010; //1.164 = 01.00101010
 const2 = 10'b 0110011000; //1.596 = 01.10011000
 const3 = 10'b 0011010000; //0.813 = 00.11010000
 const4 = 10'b 0001100100; //0.392 = 00.01100100
 const5 = 10'b 1000000100; //2.017 = 10.00000100
end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
      Y_reg <= 0; Cr_reg <= 0; Cb_reg <= 0;
      end
   else
      begin
          Y_reg <= Y; Cr_reg <= Cr; Cb_reg <= Cb;
      end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
       A_int <= 0; B1_int <= 0; B2_int <= 0; C_int <= 0; X_int <= 0;
      end
   else
     begin
     X_int <= (const1 * (Y_reg - 'd64)) ;
     A_int <= (const2 * (Cr_reg - 'd512));
     B1_int <= (const3 * (Cr_reg - 'd512));
     B2_int <= (const4 * (Cb_reg - 'd512));
     C_int <= (const5 * (Cb_reg - 'd512));
     end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
       R_int <= 0; G_int <= 0; B_int <= 0;
      end
   else
     begin
     R_int <= X_int + A_int;
     G_int <= X_int - B1_int - B2_int;
     B_int <= X_int + C_int;
     end

/* limit output to 0 - 4095, <0 equals o and >4095 equals 4095 */
assign R = (R_int[20]) ? 0 : (R_int[19:18] == 2'b0) ? R_int[17:10] : 8'b11111111;
assign G = (G_int[20]) ? 0 : (G_int[19:18] == 2'b0) ? G_int[17:10] : 8'b11111111;
assign B = (B_int[20]) ? 0 : (B_int[19:18] == 2'b0) ? B_int[17:10] : 8'b11111111;

endmodule
```

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

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

module zbt_6111(clk, cen, we, addr, write_data, read_data,
                  ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);
   input clk;                   // system clock
   input cen;                   // clock enable for gating ZBT cycles
   input we;                    // write enable (active HIGH)
   input [18:0] addr;           // memory address
   input [35:0] write_data;     // data to write
   output [35:0] read_data;     // data read from memory
   output        ram_clk;       // physical line to ram clock
   output        ram_we_b;      // physical line to ram we_b
   output [18:0] ram_address;   // physical line to ram address
   inout [35:0]  ram_data;      // physical line to ram data
   output        ram_cen_b;     // physical line to ram clock enable

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

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.
   reg [1:0]   we_delay;

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

   // create two-stage pipeline for write data
   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

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


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

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

endmodule // zbt_6111
```