6.111 Project Report: Sign Language Translator

Marc Felix and Deb Torres

Table of Contents

1 Introduction

American Sign Language (ASL) serves as a form of non-verbal communication for entire facets of society. However, the majority of Americans are not fluent in, let alone able to understand, ASL. This limits the means by which individuals that rely on ASL can communicate with the rest of the population. As such, we created a real-time sign language translator using the Nexys 4 DDR that recognizes the ASL alphabet. This alphabet consists of all 26 letters in the English alphabet, with 24 letters signed without motion and 2 letters signed including movement of the hand. While ASL letters are normally signed with the signer's dominant hand, our system only recognizes signs given with the right hand.

2 Materials

- Nexys 4 DDR FPGA
- OV7670 camera and ESP8266 microcontroller
- VGA compatible monitor
- Red, yellow, green, and blue LilyPad LEDs
- Conductive thread
- Coin-cell battery holder with switch
- CR2032 Batteries
- Black cotton gloves
- Black fabric paint
- Hot glue
- Assorted paper scraps

3 LED Glove

Figure 1: LED glove

To give our system a way to recognize points of interest on the user's hand, we designed and created an LED glove. Since we worked with letters signed with the right hand, the glove is worn on the user's right hand with the lights on the palm side. On the back of the hand is a coin-cell battery holder that powers our LEDs with a CR2032, 3-volt, lithium, coin-cell battery. As shown in Figure 1, the thumb and pinkie have matching blue lights, the palm and ring finger have matching red lights, the index finger has a yellow light, and the middle finger has a green light. The LED color corresponding to each point of interest was chosen arbitrarily, however the placement of the duplicated colors (red and blue) was a specific design choice. These elements are attached to the glove and the LEDs pull current from the battery in parallel through conductive thread. The positive terminal of each LED is wrapped in 2-3 loops of conductive thread. This thread then runs in mid-size running stitches to the positive terminal of the battery holder, which also gets looped around 2-3 times. Then, we connected the negative terminals using the same process. Figure 2 shows how our LEDs are connected to the battery without crossing positive and negative threads, as well as reducing potential for contact when adjacent fingers are touched together.

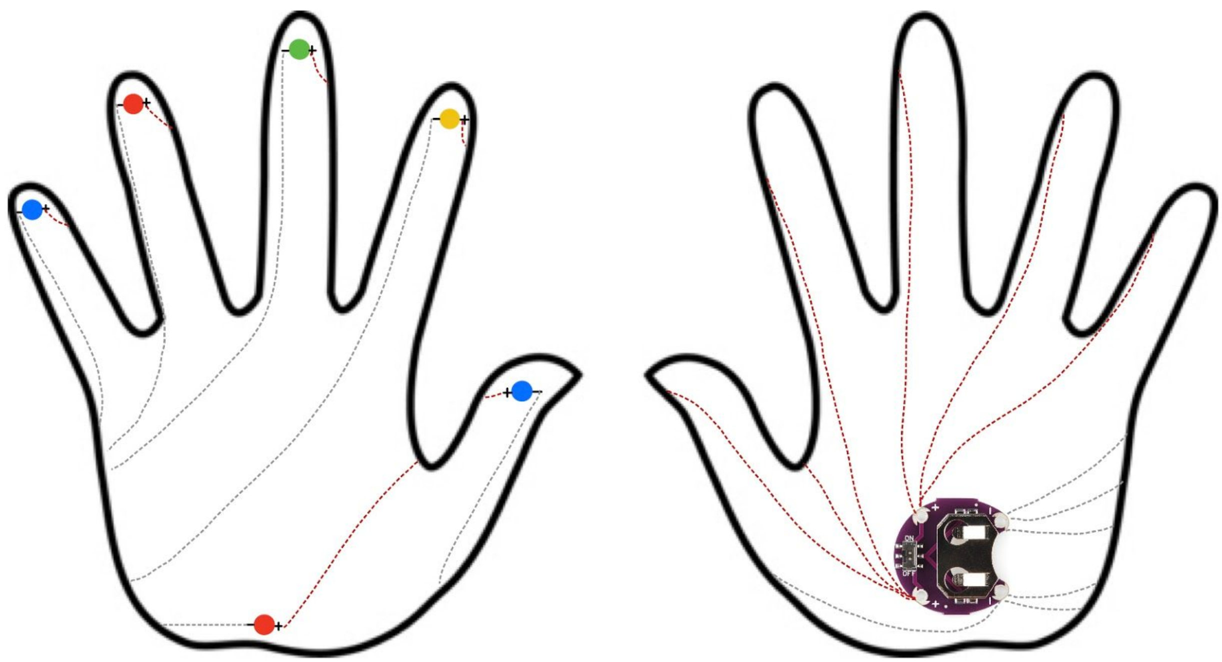

Figure 2: Connections on LED glove (positive is red, negative is gray)

At this point, the battery will power the LEDs, however, we took a few extra steps to further improve the glove. To start, threads are exposed and can easily touch other threads when moving through signs. In order to prevent this and protect the thread from fraying, we insulated the stitches with fabric paint and allowed it to dry. In addition, since the terminals of the LEDs are especially prone to bumping into each other, we used hot glue to protect the LED terminals and the thread looped around them. In addition, we needed to ensure the LEDs could be seen clearly by the camera. When LEDs are too bright, the center looks white when perceived by the camera, creating a ring of color around a white center rather than a dot of color, so we diffused our yellow LED with a strip of paper secured over the light. Finally, the green LED was dimmer than the other colors, making it appear smaller. To combat this, a dot hot glue (smudged to increase its opacity) covers the light, diffusing it so it appears larger to the camera. The glove in its final form is shown in Figure 1.

While we enjoyed improving our sewing skills, we faced some challenges designing and creating the glove. To start, the first glove prototype was sewn with very large running stitches, which created faulty connections and often didn't work. In addition, we didn't realize how frequently threads would touch when signing, so we had to go back and add more insulation multiple times. One shocking discovery we made is that, possibly due to sweat, wearing the glove for a long time often caused a slight buzzing sensation in the hand since the threads were uninsulated on the inside. Learning all of these lessons allowed us to create the best version of our glove possible, which is both comfortable for the user and functional for our system.

4 System Overview and Block Diagram

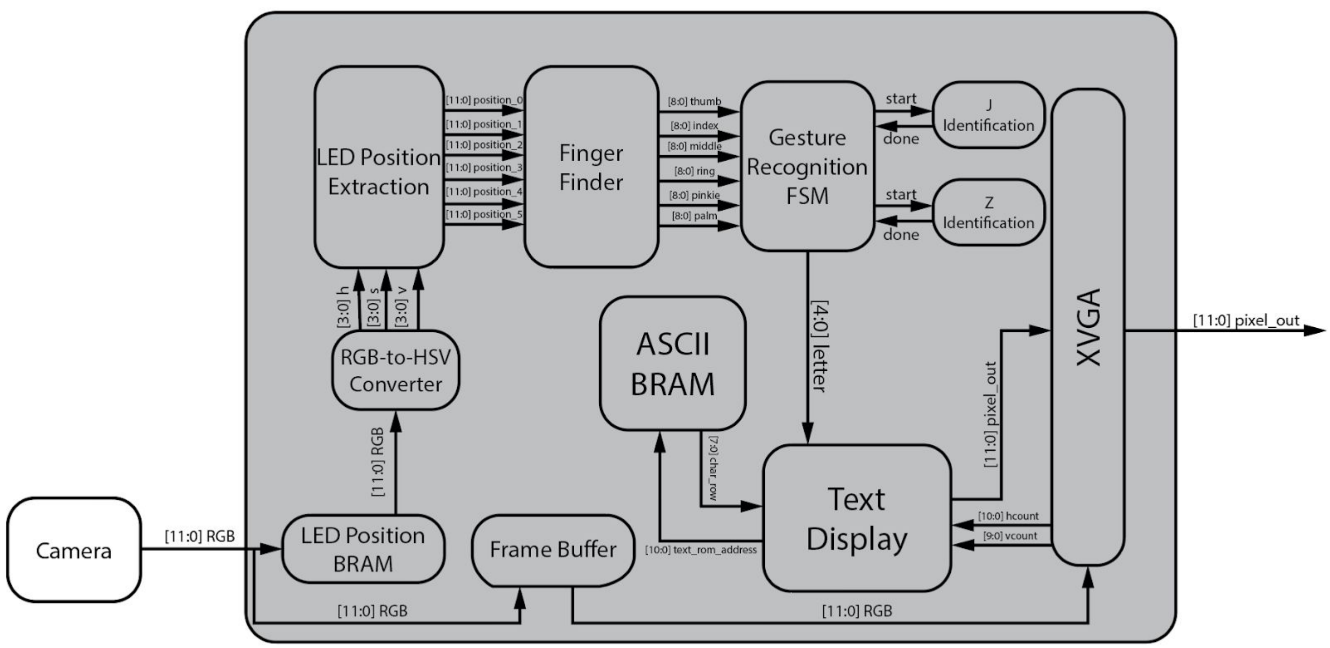

Figure 3: Top Level Module

The top level module of our design (depicted by the gray background in Figure 3) utilizes the Frame Buffer, LED Position BRAM, RGB to HSV converter, LED Position Extraction, Finger Finder, Gesture Recognition and J and Z Identification, Text Display, ASCII BRAM, and XVGA modules to generate VGA outputs given a camera input. Camera data is first passed to the LED Position BRAM to accumulate a full frame of data for processing. Once a frame is assembled, the RGB to HSV Converter takes in the RGB pixels in the frame and outputs pixel values in the HSV format. HSV pixels are then passed to the LED Position Extraction module where chroma keying is used to identify the positions of up to 6 of the glove-mounted LEDs. These positions and corresponding LED colors are passed to the Finger Finder module where the positions are assigned to the points of interest. The positions of the points of interest are passed into the gesture recognition module, which outputs the letter recognized from the LED positions either through static recognition or through interaction with J and Z Identification modules. The Text Display module uses this letter input as the input to the ASCII BRAM, which returns the appropriate pixel for the current pixel count. The Text Display module then outputs this pixel data and finally, the XVGA module formats this pixel data into a proper VGA output.

5 Modules

5.1 Camera

The camera input is created through a few hardware and software pieces, being an OV7670 camera, an ESP8266 microcontroller, the microcontroller Camera Control module, the FPGA Camera Read module, and the Frame Buffer. Starter code for these modules was provided, however the microcontroller Camera Control module needed brightness and saturation adjustments in order to suit the needs of the project. Once the camera data is received from the camera hardware, it is written to two separate BRAM modules. One of these is the aforementioned Frame Buffer which outputs raw camera data to the monitor; the other is an identical BRAM which provides data for the recognition pipeline. Despite containing the same data, these two BRAMs are accessed at different rates, therefore requiring two independent modules.

5.2 RGB to HSV Converter

The input from the camera is an RGB value for each pixel, which means each pixel has a separate value for the red, green, and blue intensity of the pixel. The converter, written by Kevin Zheng in 2010 (shoutout Kevin), produces an HSV representation of the pixel, giving a value (number) for the hue, saturation, and value (brightness). This hue value will be important in chroma keying. In 2010, when Kevin Zheng wrote this converter, the IP for a divider was slightly different than what Xilinx generates for users today, so we made slight alterations to fit the newer IP into the module. However, the current IP still has a 20 cycle latency, where the converter requires an 18 cycle latency. We were able to circumvent this issue, as the last two cycles of latency were a result of calculating the remainder, which is of no importance to us and can, as a result, be ignored.

5.3 LED Position Extraction (Marc)

The first of our two major modules, the LED Position Extraction module, takes in 8-bit hue, saturation, and value integers which represent a single RGB565 pixel. With this pixel, and the 76,799 (320x240) pixels that will follow, we need to determine where in the camera frame the (up to) 6 LEDs are positioned.

5.3.a Implementation Details

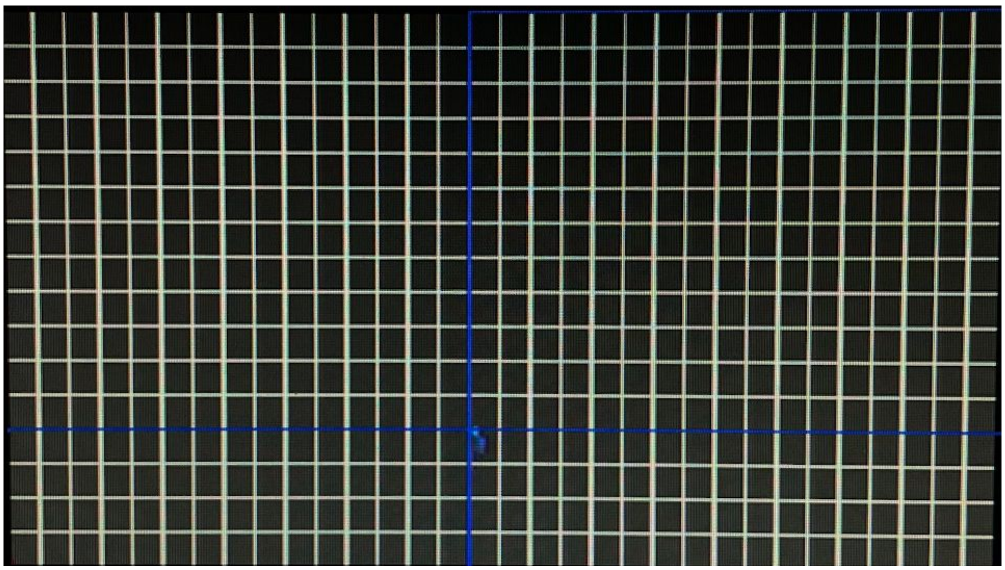

Figure 4: Grid of bins

This entire module is centered around the idea of a "bin" which we defined as a 15x10 (row-column) pixel area. These bins are arranged in a 16x32 grid, as shown in Figure 4, within the camera frame, such that all pixels within the camera frame fall into one of these 512 bins. In order to access each bin, all of which are represented by four dual-port RAMs (one for each LED color), we require a way to update the input address "automatically". We accomplish this through the Cell Determiner module, which uses the current x and y positions of a pixel and internal counters to output a BRAM address. With this groundwork in place, we can now describe the position extraction process.

To begin, the module remains idle until the start signal is asserted. Upon assertion, the HSV values currently on the input wire are passed to a simple combinational module which checks to see which color range the hue value resides in. Depending on this output, one of four temporary color registers is incremented to indicate that a pixel of the register's color was recognized. This process is repeated for 7 more cycles until the edge of the first bin is reached, at which point we sum the value (initially 0) at the current BRAM address with the temporary color register and replace the value in the BRAM; this occurs for each of the 4 colors in parallel. This is repeated for the following bins in the current row until the last pixel in the row is reached. If all 16 pixel rows have been counted for the current row of bins, the bin address is set to the first column of the next bin row; otherwise, the bin address is set back to the first column of the current bin row so that all pixel rows get counted for the current row of bins. This process continues until all 76,800 pixels in a single frame have been accounted for in the bin sums. Following the processing of the last pixel, the Cell Determiner module outputs a done signal to indicate that the LED Position Extraction module can proceed.

On the assertion of this signal, the module begins iterating over all 512 addresses of the completed bins. After a 2 cycle delay, to account for the BRAM read latency, the value from each address (the number of pixels of a color in a bin) is checked to see if it is above a specified threshold; if it is, the x and y positions of the bin, prepended with the color that was above the threshold, is set on one of the 6 possible indices in the output array. For simplicity sake, the thresholds are checked in the predetermined order: red, blue, green, then yellow. Once all 6 of the output positions have been assigned an output, or all the bins have been checked (whichever occurs first), the module asserts "done" and begins the resetting process. This process takes 512 cycles since the values at all the addresses of the bin BRAM must be set back to 0 in preparation of the next frame.

The entire process takes roughly 77,826 cycles at 65MHz, though the exact timing is not important for our purposes, so long as the module is complete before a new frame is available in the frame buffer. This happens to be the case for our system, however, even if this were not the case and we were limited to only processing, for example, every other frame, there would be no visible difference for the user.

5.2.b Color Recognition (or lack thereof)

This module, of course, did not not come without its own challenges in implementation. The most obvious example of this is the difficulty of color recognition. Starting with the camera sensors, we immediately lose some color data due to suboptimal camera initialization settings and to the oversaturation of the sensors by the LEDs (though this was slightly alleviated by diffusing the problematic colors). Additionally, we lose color data in the, arguably necessary, conversion of the RGB565 values to an 8-bit hue. All of this is on top of the disadvantageous similarities in wavelength of some of the colors (e.g. yellow and red). Even with all these factors, half of the colors, specifically blue and green, remained reliably recognizable. Red and yellow, however, started off as very unreliably differentiable.

Getting these colors properly differentiated required significant experimentation with upper and lower limits for each of the hue ranges that correspond to the four colors. A value threshold was also needed so we could filter out background colors that would be recognized unintentionally. However, we found even with these changes, there was still large enough variance in the appearance of the LEDs (due to subtle lighting changes and different brightnesses) that we required one more threshold in order to reach an acceptable level of reliability: the count of pixels of one color in each bin. We were very surprised to find that this threshold worked best when set to a value less than 20 pixels (i.e. ~13% of a bin) for all four colors.

All this experimenting was made possible through the use of the Virtual Input/Output IP in Vivado which allowed us to shift these limits as the module displayed an output to the monitor. This IP was also used for determining the pixel number threshold for the classification of bins. Given its usefulness throughout our project, we highly recommend using this IP for debugging a hardware implementation, as long as the trigger functionality of the ILA IP is not explicitly required.

5.3.c Grid Size Considerations

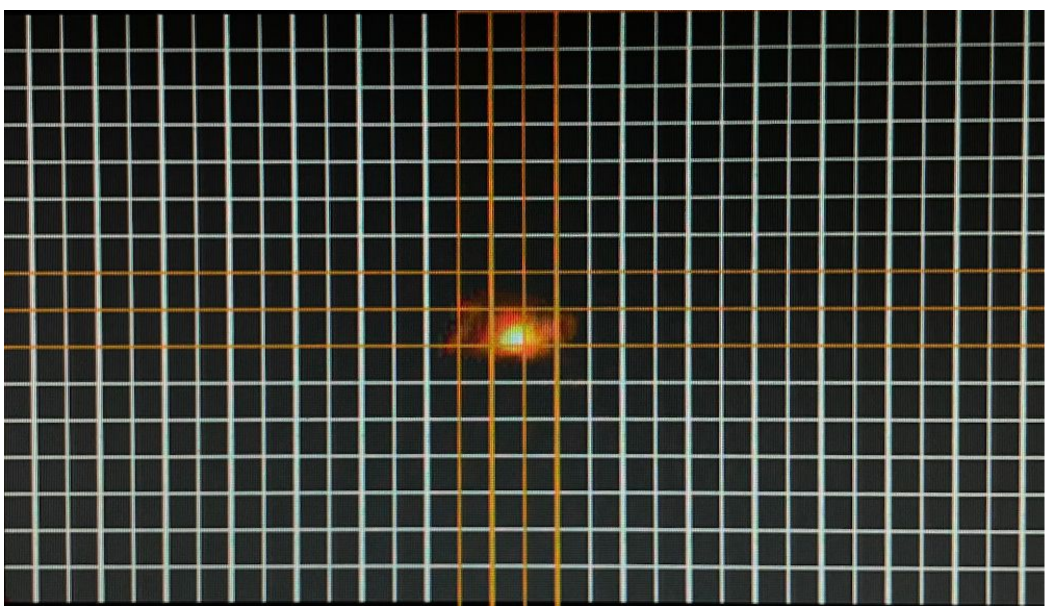

Figure 5: Extreme double counting of a single LED

The other main challenge in getting the LED Position Extraction module to function properly was determining the size of the grid (and therefore the size of the bins). Fundamentally, we were not able to make the bin area very small, since we would run the risk of counting a single LED in more than one bin, as shown in Figure 5 (this issue affects letters like "B", "C", and "O" the most, since there are only 6 position outputs available, meaning a double count causes another LED to be ignored). Conversely, we were not able to make the bins too large, since we would run the risk of grouping two LEDs into one bin which would limit us from recognizing signs with adjacent fingers. Additionally, if the bins are too large, we would lose too much position information for the Gesture Recognition module to be effective.

Since the issue of double counting would likely limit our lower bound of bin area too much, we opted to add logic which would explicitly avoid this. We do this by checking if any of the four adjacent bins to the bin in question was previously recognized as the same color as the bin in question; if this is the case, we ignore the current bin and proceed normally. This greatly reduces the amount of double counting, though if someone were to replicate this project, we would recommend increasing the radius of bins this checks, as there is still room to increase this without risking counting two legitimately different LEDs as one.

Luckily enough, with the aforementioned fix in place, our first guess of bin size worked well enough that we didn’t see any reason to try other bin sizes. Given this, we believe there is a decently sized range of bin areas that would result in the same, or even better, functionality. One aspect we did not experiment with was the shape of individual bins, though we believe a recreation of this project could benefit from a more square shaped bin that more accurately captures the circular appearance of the LEDs.

5.4 Finger Finder (Deb)

Between our two main modules is the Finger Finder module. The purpose of this module is to take the LED locations in a frame and assign them to their corresponding points of interest, being the thumb, palm, and the index, middle, ring, and pinkie fingers. This module takes the LED Position Extraction output, consisting of six, 12-bit values. The three most significant bits represent the color of the LED at the identified location (a value from one to four), the next five bits represent the bin number in the x-direction, and the last four bits represent the bin number in the y-direction. Since not all six LEDs will be present in every frame, the array will have valid color entries for however many LEDs are identified, and the remaining entries in the array will be 0, meaning the array contains no more valid positions.

With our current design, this module requires certain assumptions about hand positions in the ASL alphabet. These assumptions are necessary in order to assign LEDs of our repeated colors, but are effective and mostly unproblematic for gesture recognition. The first assumption is that, from the user’s perspective, the thumb is always to the left of the pinkie. This is mirrored from the camera’s perspective, giving the thumb a larger bin value than the pinkie in the x-direction. This assumption allows us to separate the thumb from the pinkie when both appear in the frame (two blue lights). Similarly, we assume the ring finger is always above the light on the palm, giving the ring finger a smaller bin value in the y-direction. This allows us to separate the two red LEDs as the ring finger and palm. These two assumptions are consistent in the recognition of all letters. We must also consider the case where only one light of a duplicated color is present. For red lights, the palm light is present in every letter we recognize and the ring finger is not, so if there is just one red LED we recognize it as the palm. For blue lights, many signs include the thumb and not the pinkie, but the pinkie does not appear without the thumb. As a result, a single blue light is recognized as the thumb. This is the only imperfect assumption thus far, due to the movement involved when signing "J". In this sign, the rotation of the hand covers the thumb light before the pinkie light. As a result, only the pinkie is visible for part of the motion. Anticipating this allowed us to switch our finger of interest to the thumb when looking for the pinkie in the J Identification module.

The module starts by waiting until the LED Position Extraction module signals it has finished processing the current frame, meaning the current positions are valid. Validity is only guaranteed for one cycle, so the array must be saved to continue using these values in subsequent clock cycles. When the start signal is asserted, the array is saved, the cycle count is set to 0, and the position of each finger is set to 0. Over the next six clock cycles, we use the cycle count to index into the array. On any of these cycles, if the color is 0 we skip the remainder of the array since there are no more valid positions. If a yellow or green color is found on any cycle, we assign the position in this entry to the index finger or middle finger, respectively. The first cycle is the simplest for the blue and red lights, since the assumption that a single light in these colors belongs to the thumb and palm means we assign these points at the first blue or red light in the array. In the next five cycles, if the entry is a blue light, we first check if we have stored a location as the thumb on a previous clock cycle. If we have not, we store this location as the thumb. If we have, we employ the assumption that the thumb is on the user's left and will store the position with the higher bin value in the thumb, and the position with the lower bin value in the pinkie. Similarly with a second red light, the higher bin value will be the palm and the lower bin value will be the ring finger.

This module asserts a done signal in two cases. Any time the color is 0, done is asserted to signal the locations are finished being assigned. In addition, when we have processed all the positions in the array, we assert the done signal. The output of the module is one, 9-bit variable for each LED on the hand, labeled as thumb, index, middle, ring, pinkie, and palm, containing the position of the bin where that point of interest is. These values are a valid position for any point of interest in the frame, and 0 for any not showing.

5.5 Gesture Recognition (Deb)

The Gesture Recognition module is the other main module in our system. The inputs correspond to the position of each point of interest from the Finger Finder module. The output of the module is a 5 bit representation of the letter (space = 0, "A" = 1, … , "Z" = 26) and a done signal when a letter is to be displayed. The module is mainly a static recognition state machine to identify the 24 static letters, with instances of the dynamic recognition modules to identify “J” and “Z”.

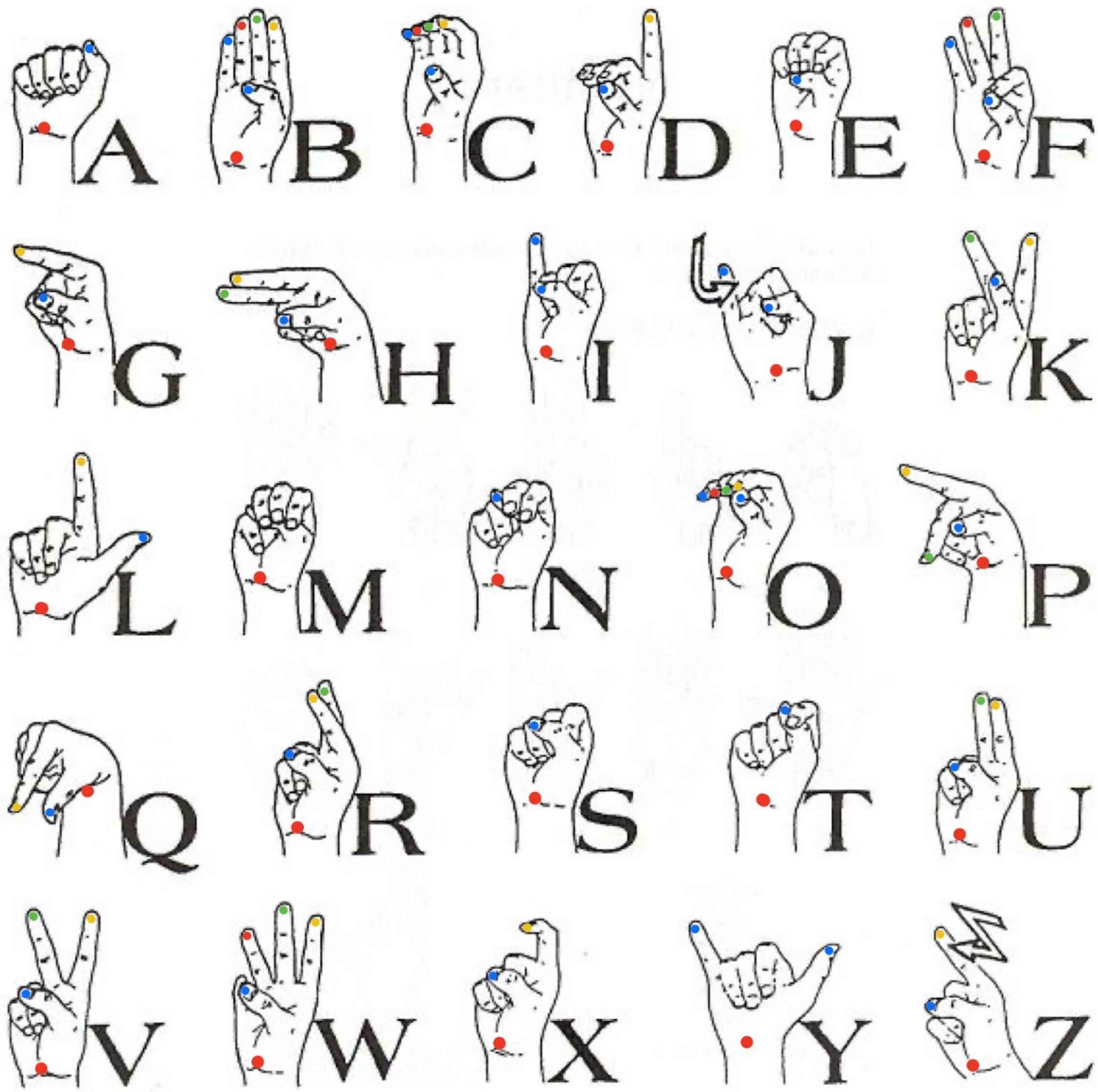

Figure 6: Lights visible to system in ASL alphabet

5.5.a Static Letter Recognition

The static recognition state machine begins in the idle state. When the Finger Finder module asserts the done signal, the state machine moves to the static recognition state to determine the letter created with the lights shown in the newly valid locations. The static recognition state is broken into the subsets of letters signed with the same lights showing. The lights showing in each letter are shown in Figure 6. For example, one of these subsets consists of the letters “A”, “E”, “N”, “S”, and “T”, which are all signed with the thumb and the palm showing and all other points hidden. If a frame is not showing lights corresponding to one of the eight subsets of letters, the state machine returns to the idle state, which will cause no change in the display.

When a subset is chosen, the given positions are checked for identifying features of each letter in the subset. For example, “A” creates the largest distance in the x-direction between the thumb and the palm. If the x-distance between the current positions is above the predetermined threshold value, the letter is recognized as “A”. The letter variable is set to 1, the number corresponding to “A”, and the state machine transitions to the debounce state.

The debounce state is how the user is saved from small variations or single frame mistakes causing changes in the displayed letter. This state uses a saved copy of the previous letter to see if our letter has stayed consistent. If the letter matches the previous letter, a counter is incremented. If not, the counter is reset. If the letter matches for four frames, the done signal is asserted so the display module knows the letter is ready to be shown on the monitor.

The recognition of letters once a subset is identified is the most difficult part of the static recognition state machine, and is the feature that required the most testing and fine tuning. Subsets containing one letter, such as “M” or “F”, are the simplest, since it is enough to recognize the correct lights to recognize the letter. However, having more letters within a subset, requires more distinctive features in order to separate the letters. For example, in the case of “A”, “E”, and “S”, the palm and thumb are the only lights present in the frame and the thumb has the same bin height for all three signs. So the signs must be separated by the thumb’s position in the x-direction relative to the palm. Each sign requires its own distance threshold between the two points, so that space must be partitioned into three sections to recognize all three letters. One challenge in writing this module is narrowing those threshold distances to catch all signs of a letter without including any other signs.

5.5.b Dynamic Letter Recognition

Dynamic characters “J” and “Z” cannot be recognized in the static recognition state machine because the locations of the points of interest must be processed over the course of many frames in order to recognize the letter. As such, two separate state machines handle the recognition of “J” and “Z” respectively. These modules have instances in the gesture recognition module and work in parallel with the static recognition. This is so a user can begin to sign a dynamic character, but not complete the character and move onto another letter without the dynamic recognition getting stuck due to the partially signed letter. This way, the new letter can be detected by the static recognition state machine and cause the dynamic module to exit.

The static recognition state machine interacts with the dynamic modules by providing the positions of relevant points of interest and a start signal for the module to begin recognition, while the dynamic modules signal when their dynamic character has been identified. The trigger for the “J” and “Z” modules are set when the static recognition state machine outputs a done signal with the letters “I” and “D” respectively. This is because the dynamic letters start with these hand positions before the movement begins. As previously stated, this does not disrupt the static recognition state machine, as it will continue through the states while the module works. Then, when a dynamic identification module is complete, it signals its letter has been recognized and should now be displayed. This is the only portion of dynamic recognition that is blocking, because the letter being output is set to the recognized dynamic character for 2 seconds before the static recognition state machine can process inputs again. This allows the user time to lock in the letter, since there isn’t a position to hold like in static recognition.

5.5.c False Recognition

Since the Gesture Recognition module is what decides the letter output on the monitor, this is where mistakes arise. Incorrect letters stem from two main types of errors: user correctable errors and position extraction error.

User correctable errors will cause the state machine to output a letter from the correct subset of letters, though the exact letter will be incorrect. This is because all the lights have been correctly identified as the necessary points of interest, but the user’s sign positioning does not meet the requirements the module has set for that letter. This can be due to the user tilting their hand while making the sign or altering the relative positions so they appear in range of another letter. One example of this is “T” and “N”, which are in the same subset of letters because they show only the palm and thumb and have the same relative distance between the thumb and palm in the y-direction. However, the thumb’s x position for either letter is just on opposite sides of the palm light, with the “T” having the thumb to the user’s left of the palm and the “N” on the right. Because the number of bins separating the two is not very large, if the user tilts their hand to their right while giving the sign for “T”, the state machine will recognize the sign as “N” since the thumb will cross to the other side of the palm and into the range for “N” recognition. This type of error is easy to correct, especially if the user is familiar with the system. In this case, the user will see the incorrect letter on the display, look at their light positions, and realize their hand is not straight up and down. Once the user reorients their sign, the display will show the intended letter. This error becomes less common as a user gets more familiar with the system and gets used to making signs the system recognizes as intended. If the system is used by just one user, the thresholds could even be adjusted to recognize their specific style of signs with little trouble.

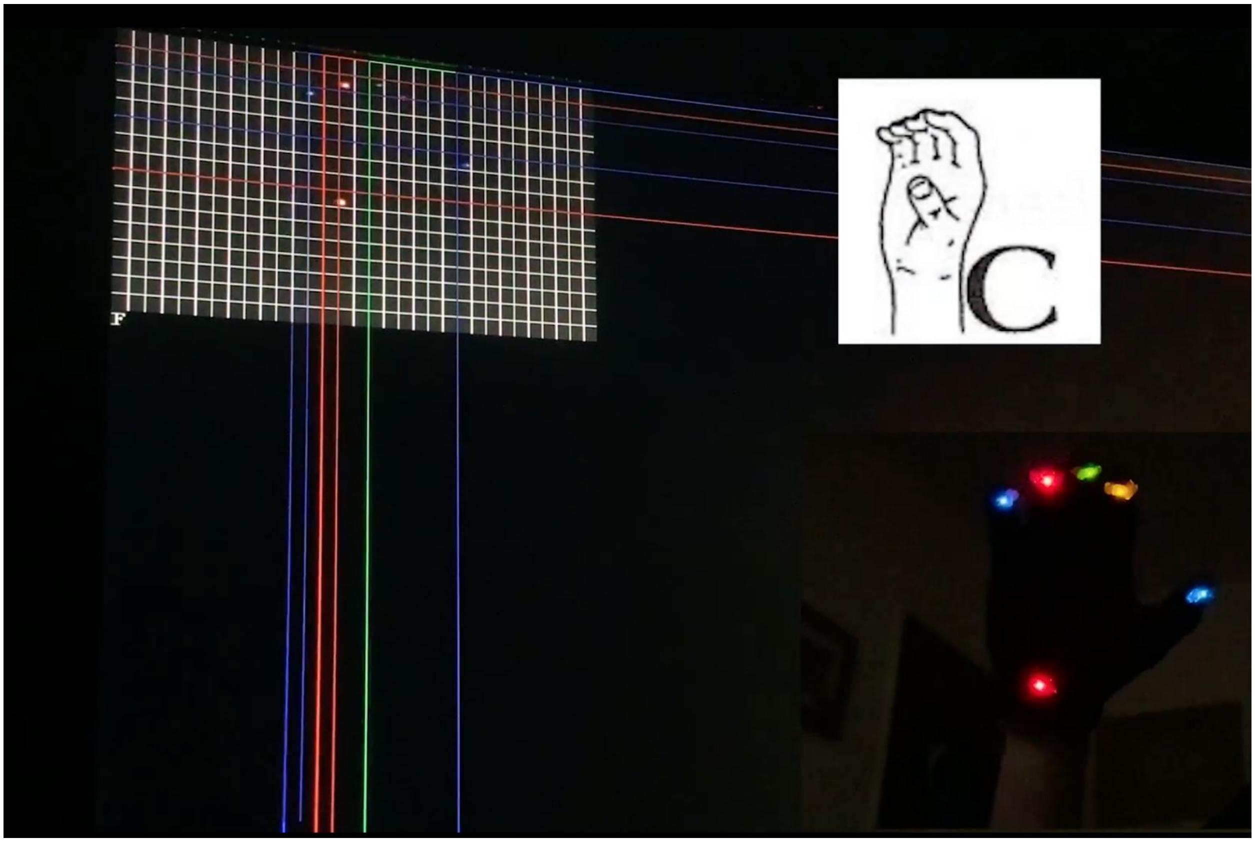

Figure 7: Misrecognition of “C”

The second kind of error that will cause an incorrect letter to appear on the display results is a fault in the recognition of LEDs located by the Position Extraction module. This can occur when an LED is unrecognized, recognized as the wrong color, or double counted. This is a more significant issue for recognizing the letter, since this will cause the static recognition state to select the incorrect subset of letters to choose from. As described in Section 5.3, the yellow light, which represents the index finger, is the most difficult to recognize. This lack of recognition causes signs produced with the index finger showing to register as a sign with all the same lights, but without the index finger showing. For example, “B”, “C”, and “O” are signed with all six lights visible and “F” is signed with all but the index finger visible. This means that if the user gives the sign for “B”, “C”, or “O” and their yellow light is too dim, their hand is too far from the camera, or their angle is such that the yellow LED is not recognized by the LED Position Extraction module (which the user will know due to the absence of the yellow crosshair on the grid), the module will produce the letter “F” as the output. This is shown in Figure 7, in which “C” is displayed as “F” due to the missed recognition of the index finger. Similar to the previous error discussed, an experienced user will see this issue and know to replace their battery or adjust their hand’s position relative to the camera until the light is more consistently recognized. However, this issue is more difficult for the user to know how to fix, since they might not have a sense for the best distance or angle for recognition. It can be difficult to adjust these thresholds to allow yellow to be recognized more easily, since its proximity to red and the difficulty distinguishing the boundary between the two can lead to a red LED registering as both a red and a yellow LED if the threshold for yellow is lowered too much, which leads to the same issues.

In a future iteration of the project, user correctable error is likely to remain due to the limitations of recognizing signs with a camera and lights. However, position extraction error could be mitigated in a future iteration by having more strict recognition conditions in order to avoid some recognition within an incorrect subset of letters. In the example letters given above, “C” and “O” are signed with middle and ring finger positions that could be distinguishable from the ones present in “F” by checking the palm to finger distance in the y-direction. As a result, a more thorough recognition process would only output “F” in the case that the positions of the index and middle finger don’t resemble those of “C” and “O”. However, even with these additional checks, the “B” to “F” error will likely still occur, since the other finger positions are not easily differentiable. Therefore, it is possible to avoid some incorrect recognition, but even with thorough analysis of misidentified letters, it does not appear to be absolutely avoidable without creating another distinguishable difference between the letters (such as shifting to a wider spread in the middle and ring fingers when signing “F” to create greater x-direction separation than in “B”).

5.6 J and Z Identification (Marc)

Generally, the structures of both the “J” and “Z” identification modules are similar with only minor differences between the actual recognition of the letters. As such, we will describe only one module in-depth and simply relate it to the other module after.

5.6.a Implementation

Using “J” as an example, the module begins with the assertion of “I”, as mentioned in Section 5.5.b. On the same cycle, current relevant finger positions (thumb and pinkie positions for “J”) are stored in registers and the state is moved from idle to the activated state. The module then waits until the real position of the pinkie (passed into the module as the thumb due to previous assumptions in the pipeline) is a specified distance below the initial thumb position. This detects the downward sweep that initially occurs when signing a “J”. Once this is detected, the state machine moves forward and similar logic is present to detect the leftward sweep at the end of the sign. Throughout this process, the only time the FSM resets to the idle state is when a different letter is output by the static recognition state machine (for reasons explained in Section 5.6.b). Once both these motions are detected, a done signal is asserted for one cycle and the FSM resets to idle. The only differences for the “Z” identification module are the storing of the index finger initially and different required motions (left-down/right-left).

5.6.b Accidental Recognition

Our initial goal when designing these modules was to require the user to follow a defined, though not strict, path to sign each letter, with large enough deviations from the path resulting in a reset of the FSM. This would make it very difficult to accidentally sign one of the letters in question. We quickly realized, though, that this wouldn't be possible due to the assumptions that we were required to make previously. Specifically, the way our module determines thumb and pinkie positions caused there to be unpredictable patterns in the positions given as the thumb and pinkie. This led to frequent, accidental invalidation of valid signs. In order to avoid this, we settled upon the current system where the only way to invalidate the sign is by the static recognition FSM recognizing a different letter than "I" or "D", depending on the first sign recognized. This, of course, causes "J" detection to be much less robust against false positives ("Z" is less affected due to the extra motions required). Since the error rate was somewhat low, we simply accepted this as a necessary consequence, however we believe it's possible to slightly reduce this false positive rate by adding more states/required movements to complete the sign.

5.7 Text Display (Marc)

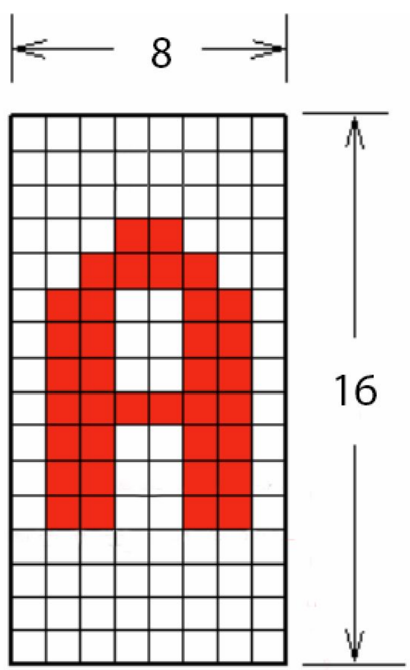

Figure 8: Character dimensions as represented in BRAM

Our final module of the pipeline, the Text Display module, is relatively simple compared to the previous ones. At its core, the Text Display module is a simple finite state machine with 64 states, one for each space a letter can occupy, resulting in a maximum character length of 64. The slight complexity of this module comes from the way in which the ASCII characters are displayed over VGA from a single-port ROM. Each character “drawn” by the COE file the ROM is initialized with is 16-bits tall by 8-bits wide, resulting in a depth of 2048 bits and a width of 8 bits. The module uses a specified start address along with an internal offset counter to access character data 8 bits at a time. Once read, these bits are stored in a temporary register where the bit at the 7th index is passed to the pixel_out output to be sent along the VGA connection to the monitor. Every cycle, this temporary register is shifted to the left by 1 to access the next appropriate pixel for displaying. This process is repeated for all 64 available character indices 16 times to capture all the rows of each character and once finished, the module resets all counters to prepare for the next frame.

6 Conclusion

As a team, we achieved all the goals we set out to accomplish and are satisfied with the final system. We are proud to have designed and created a system that recognizes all 26 letters of the ASL alphabet, including the two dynamic letters we initially categorized as a stretch goal, all the while adhering to our initial goal of staying as faithful to true ASL as much as possible. In addition, we feel the current system serves as a strong base upon which further functionality could be implemented, given a longer timeline. Were we to return to this project at a later date, our top priority would be implementing functionality of the system for left-handed users in addition to the currently supported right-handed users. Our vision for this would be to automatically detect which hand the user is signing with and use the appropriate internal logic to identify signs as before. At an overall system perspective, one change we would make is in the LED arrangement; knowing the current accuracy of LED position extraction, we would use a more reliably recognized LED, like green, as a duplicated color instead of the red we currently use. An ultimate goal for us would be to reach a high enough level of reliability in this base system that would allow us to pursue recognition of entire ASL words and phrases.

7 Appendix

Appendix

Top Level

```

`timescale 1ns / 1ps

module top_level(
   input clk_100mhz,
   input[15:0] sw,
   input btnc, btnu, btnl, btnr, btnd,
   input [7:0] ja, //pixel data from camera
   input [2:0] jb, //other data from camera (including clock return)
   output   jbclk, //clock FPGA drives the camera with
   input [2:0] jd,
   output   jdclk,
   output[3:0] vga_r,
   output[3:0] vga_b,
   output[3:0] vga_g,
   output vga_hs,
   output vga_vs,
   output led16_b, led16_g, led16_r,
   output led17_b, led17_g, led17_r,
   output[15:0] led,
   output ca, cb, cc, cd, ce, cf, cg, dp,  // segments a-g, dp
   output[7:0] an    // Display location 0-7
   );
   parameter CAM_WIDTH = 320;   //width of each camera frame in pixels
   parameter CAM_HEIGHT = 240;  //height of each camera frame in pixels

   localparam NO_COLOR = 0;
   localparam RED = 1;
   localparam BLUE = 2;
   localparam GREEN = 3;
   localparam ORANGE = 4;

        logic clk_65mhz;
    // create 65mhz system clock, happens to match 1024 x 768 XVGA timing
    clk_wiz_0 clkdivider(.clk_in1(clk_100mhz), .clk_out1(clk_65mhz));

    wire [6:0] segments;

    assign  dp = 1'b1;  // turn off the period

    assign led = sw;                        // turn leds on

    assign led16_r = btnl;                  // left button -> red led
    assign led16_g = btnc;                  // center button -> green led
    assign led16_b = btnr;                  // right button -> blue led
    assign led17_r = btnl;
    assign led17_g = btnc;
    assign led17_b = btnr;

    wire [10:0] hcount;    // pixel on current line
    wire [9:0] vcount;     // line number
```

```
    wire hsync, vsync, blank;
    wire [11:0] pixel;
    reg [11:0] rgb;
    xvga xvga1(.vclock_in(clk_65mhz),.hcount_out(hcount),.vcount_out(vcount),
          .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));


    // btnc button is user reset
    wire reset;
    debounce db1(.reset_in(reset),.clock_in(clk_65mhz),.noisy_in(btnc),.clean_out(reset));

    //sw[15] is used to confirm a letter
    logic text_display_trigger;
    debounce db5(.reset_in(reset),.clock_in(clk_65mhz),.noisy_in(sw[15]),.clean_out(text_display_trigger));

    //sw[14] is used to delete a letter
    logic text_display_trigger_delete;
    debounce db6(.reset_in(reset),.clock_in(clk_65mhz),.noisy_in(sw[14]),.clean_out(text_display_trigger_delete));

    logic xclk;
    logic[1:0] xclk_count;

    logic pclk_buff, pclk_in;
    logic vsync_buff, vsync_in;
    logic href_buff, href_in;
    logic[7:0] pixel_buff, pixel_in;

    logic [11:0] cam;                   //VGA output pixel data received from the camera
    logic [11:0] frame_buff_out;        //pixel data from the frame buffer
    logic [15:0] output_pixels;         //pixel data from camera
    logic [15:0] old_output_pixels;
    logic [12:0] processed_pixels;      //camera pixel data with truncated r, g, and b values
    logic valid_pixel;
    logic frame_done_out;               //true when the frame buffer contains the full previous frame

    logic [16:0] pixel_addr_in;         //frame buffer input address
    logic [16:0] pixel_addr_out;        //frame buffer output address

    logic [11:0] converter_out;         //input to the RGB-to-HSV converter

    //RGB-to-HSV values
    logic [7:0] hue;
    logic [7:0] saturation;
    logic [7:0] value;

    logic extraction_start;             //start signal for the position extraction module
    logic [8:0] extraction_x;           //current pixel x position
    logic [7:0] extraction_y;           //current pixel y position
    logic counting;                     //true when position extraction module is working
    logic [4:0] converter_delay;        //delay which accounts for BRAM read and RGB-to-HSV delay
```

```
    logic [16:0] pixel_addr_to_hsv;     //address for pixel input to RGB-to-HSV converter
    logic [11:0] led_positions [5:0];   //length 6 array of 12-bit wide led positions

    //number of pixels required to assign a bin a color
    logic [7:0] red_threshold;
    logic [7:0] blue_threshold;
    logic [7:0] green_threshold;
    logic [7:0] orange_threshold;

    //lower and upper hue bounds for each color
    logic [7:0] red_lower;
    logic [7:0] red_upper;
    logic [7:0] blue_lower;
    logic [7:0] blue_upper;
    logic [7:0] green_lower;
    logic [7:0] green_upper;
    logic [7:0] orange_lower;
    logic [7:0] orange_upper;

    logic [7:0] value_threshold;        //minimum value to count a pixel as a color

    //led positions from last position extraction cycle; used to create colored crosshairs
    logic [11:0] last_pos0;
    logic [11:0] last_pos1;
    logic [11:0] last_pos2;
    logic [11:0] last_pos3;
    logic [11:0] last_pos4;
    logic [11:0] last_pos5;

    //finger positions from finger finder module
    logic [8:0] thumb_pos;
    logic [8:0] index_pos;
    logic [8:0] middle_pos;
    logic [8:0] ring_pos;
    logic [8:0] pinkie_pos;
    logic [8:0] palm_pos;

    logic gesture_done;                 //true when gesture recognition fsm has found a letter
    logic [4:0] letter;                 //letter from GR-fsm

    logic [11:0] text_pixel_out;        //pixel used for text display
    logic text_display_confirm_letter;
    logic text_display_delete_letter;

    assign xclk = (xclk_count >2'b01);
    assign jbclk = xclk;
    assign jdclk = xclk;


    //BRAM used for storing and displaying camera data to the monitor
    blk_mem_gen_0 jojos_bram(.addra(pixel_addr_in),
                             .clka(pclk_in),
                             .dina(processed_pixels),
                             .wea(valid_pixel),
                             .addrb(pixel_addr_out),
                             .clkb(clk_65mhz),
```

```
                              .doutb(frame_buff_out));

    //BRAM used for inputting camera data to GR-fsm pipeline
    blk_mem_gen_0 led_position_bram(.addra(pixel_addr_in),
                                    .clka(pclk_in),
                                    .dina(processed_pixels),
                                    .wea(valid_pixel),
                                    .addrb(pixel_addr_to_hsv),
                                    .clkb(clk_65mhz),
                                    .doutb(converter_out));

    //instantiates the RGB-to-HSV converter
    rgb2hsv my_converter (.clock(clk_65mhz),
                          .reset(reset),
                          .r({converter_out[11:8], 4'b0}),
                          .g({converter_out[7:4], 4'b0}),
                          .b({converter_out[3:0], 4'b0}),
                          .h(hue),
                          .s(saturation),
                          .v(value));

    //Virtual input/output IP used for setting various thresholds for the led position extraction module
    vio_0 threshold_picker (
      .clk(clk_65mhz),
      .probe_in0(led_positions[0]),
      .probe_in1(led_positions[1]),
      .probe_in2(led_positions[2]),
      .probe_in3(led_positions[3]),
      .probe_in4(led_positions[4]),
      .probe_in5(led_positions[5]),
      .probe_out0(red_threshold),
      .probe_out1(blue_threshold),
      .probe_out2(green_threshold),
      .probe_out3(orange_threshold),
      .probe_out4(red_lower),
      .probe_out5(red_upper),
      .probe_out6(blue_lower),
      .probe_out7(blue_upper),
      .probe_out8(green_lower),
      .probe_out9(green_upper),
      .probe_out10(orange_lower),
      .probe_out11(orange_upper),
      .probe_out12(value_threshold)
    );

    //instantiates the position extraction module
    led_position_extraction my_extractor (.clock(clk_65mhz),
                                          .reset(reset),
                                          .start(extraction_start),
                                          .h(hue),
                                          .s(saturation),
                                          .v(value),
                                          .x(extraction_x),
                                          .y(extraction_y),
                                          .red_threshold(red_threshold),
                                          .blue_threshold(blue_threshold),
                                          .green_threshold(green_threshold),
                                          .orange_threshold(orange_threshold),
```

```
                                        .red_lower(red_lower),
                                        .red_upper(red_upper),
                                        .blue_lower(blue_lower),
                                        .blue_upper(blue_upper),
                                        .green_lower(green_lower),
                                        .green_upper(green_upper),
                                        .orange_lower(orange_lower),
                                        .orange_upper(orange_upper),
                                        .value_threshold(value_threshold),
                                        .led_positions(led_positions),
                                        .done(extraction_done),
                                        .ready(extraction_ready));

    //instantiates the finger_finder module; is assigns each position to a specific finger
    finger_finder my_finder (.clock(clk_65mhz),
                        .reset(reset),
                        .led_positions(led_positions),
                        .start(extraction_done),
                        .done(finder_done),
                        .thumb(thumb_pos),
                        .index(index_pos),
                        .middle(middle_pos),
                        .ring(ring_pos),
                        .pinkie(pinkie_pos),
                        .palm(palm_pos));

    //instantiates gesture recognition fsm
    gesture_recognition_fsm my_gesture (.clock(clk_65mhz),
                                    .reset(reset),
                                    .start(finder_done),
                                    .done(gesture_done),
                                    .thumb(thumb_pos),
                                    .index(index_pos),
                                    .middle(middle_pos),
                                    .ring(ring_pos),
                                    .pinkie(pinkie_pos),
                                    .palm(palm_pos),
                                    .letter(letter));

    //instantiates text_display; used to display the letters that are signed
    text_display my_text_display (.pixel_clk_in(clk_65mhz),
                              .reset(reset),
                              .hcount_in(hcount),
                              .vcount_in(vcount),
                              .letter(letter),
                              .done(gesture_done),
                              .confirm_letter(text_display_confirm_letter),
                              .delete_letter(text_display_delete_letter),
                              .pixel_out(text_pixel_out));

    assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];
    //display_8hex display(.clk_in(clk_65mhz),.data_in({thumb_pos[3:0], index_pos[3:0], middle_pos[3:0], ring_pos[3:0], pinkie_pos[3:0], palm_pos[3:0], 8'b0}), .seg_out(segments), .strobe_out(an));
    display_8hex display(.clk_in(clk_65mhz),.data_in(letter), .seg_out(segments), .strobe_out(an));

    always_ff @(posedge pclk_in)begin
```

```
        //handles addressing for camera data input to BRAM
        if (frame_done_out)begin
            pixel_addr_in <= 17'b0;
        end else if (valid_pixel)begin
            pixel_addr_in <= pixel_addr_in +1;
        end
    end

    //states used to avoid double-counting letter confirmations/deletions
    logic text_display_state;
    logic text_display_state_delete;

    always_ff @(posedge clk_65mhz) begin
        pclk_buff <= jb[0];//WAS JB
        vsync_buff <= jb[1]; //WAS JB
        href_buff <= jb[2]; //WAS JB
        pixel_buff <= ja;
        pclk_in <= pclk_buff;
        vsync_in <= vsync_buff;
        href_in <= href_buff;
        pixel_in <= pixel_buff;
        old_output_pixels <= output_pixels;
        xclk_count <= xclk_count + 2'b01;
        processed_pixels = {output_pixels[15:12],output_pixels[10:7],output_pixels
            [4:1]};

        //handles confirmation double counting
        if (text_display_state == 0 && text_display_trigger) begin
            text_display_confirm_letter <= 1;
            text_display_state <= 1;
        end else if (text_display_state == 1) begin
            text_display_confirm_letter <= 0;
            if (text_display_trigger == 0) text_display_state <= 0;
        end

        //handles deletion double counting
        if (text_display_state_delete == 0 && text_display_trigger_delete) begin
            text_display_delete_letter <= 1;
            text_display_state_delete <= 1;
        end else if (text_display_state_delete == 1) begin
            text_display_delete_letter <= 0;
            if (text_display_trigger_delete == 0) text_display_state_delete <= 0;
        end

        if (frame_done_out && extraction_ready) begin   //when position extraction is
            ready and a new frame is available
            counting <= 1;
            converter_delay <= 0;
            pixel_addr_to_hsv <= 0;
            extraction_x <= 0;
            extraction_y <= 0;
        end else if (counting) begin
            if (converter_delay < 23) begin                      //delays pixel data so
                 positions match up with correct data
                converter_delay <= converter_delay + 1;
                pixel_addr_to_hsv <= pixel_addr_to_hsv + 1;
            end else if (converter_delay == 23) begin
                extraction_start <= 1;
                pixel_addr_to_hsv <= pixel_addr_to_hsv + 1;
```

```
                converter_delay <= converter_delay + 1;
            end else if (extraction_x == CAM_WIDTH-1 && extraction_y == CAM_HEIGHT-1) begin //finished case
                counting <= 0;
            end else begin
                if (extraction_x == CAM_WIDTH - 1) begin        //next line
                    extraction_x <= 0;
                    extraction_y <= extraction_y + 1;
                end else extraction_x <= extraction_x + 1;      //next pixel
                extraction_start <= 0;
                pixel_addr_to_hsv <= pixel_addr_to_hsv + 1;
            end
        end

    end

    assign pixel_addr_out = hcount+vcount*32'd320;
    assign cam = ((hcount<320) &&  (vcount<240))?frame_buff_out:12'h000;

   //instantiates camera reading module
   camera_read  my_camera(.p_clock_in(pclk_in),
                          .vsync_in(vsync_in),
                          .href_in(href_in),
                          .p_data_in(pixel_in),
                          .pixel_data_out(output_pixels),
                          .pixel_valid_out(valid_pixel),
                          .frame_done_out(frame_done_out));

    //uncommenting below shows the white grid
    wire border = (hcount==0 | hcount==10 | hcount==20 | hcount==30 | hcount==40 | hcount==50 | hcount==60 | hcount==70 | hcount==80 | hcount==90 | hcount==100 | hcount==110 | hcount==120 | hcount==130 | hcount==140 | hcount==150 | hcount==160 | hcount==170 | hcount==180 | hcount==190 | hcount==200 | hcount==210 | hcount==220 | hcount==230 | hcount==240 | hcount==250 | hcount==260 | hcount==270 | hcount==280 | hcount==290 | hcount==300 | hcount==310 | vcount==0 | vcount==15 | vcount==30 | vcount==45 | vcount==60 | vcount==75 | vcount==90 | vcount==105 | vcount==120 | vcount==135 | vcount==150 | vcount==165 | vcount==180 | vcount==195 | vcount==210 | vcount==225);

    //display a colored crosshair for each position depending on the color detected
    wire border_0 = {hcount == (last_pos0[8:4]*10) | vcount == (last_pos0[3:0]*15)};
    wire border_1 = {hcount == (last_pos1[8:4]*10) | vcount == (last_pos1[3:0]*15)};
    wire border_2 = {hcount == (last_pos2[8:4]*10) | vcount == (last_pos2[3:0]*15)};
    wire border_3 = {hcount == (last_pos3[8:4]*10) | vcount == (last_pos3[3:0]*15)};
    wire border_4 = {hcount == (last_pos4[8:4]*10) | vcount == (last_pos4[3:0]*15)};
    wire border_5 = {hcount == (last_pos5[8:4]*10) | vcount == (last_pos5[3:0]*15)};

    reg b,hs,vs;
    always_ff @(posedge clk_65mhz) begin
         hs <= hsync;
         vs <= vsync;
         b <= blank;
         if (extraction_done) begin
            last_pos0 <= led_positions[0];
            last_pos1 <= led_positions[1];
            last_pos2 <= led_positions[2];
            last_pos3 <= led_positions[3];
            last_pos4 <= led_positions[4];
            last_pos5 <= led_positions[5];
```

```
        end

        //each of these conditionals handles assigning a color to a crosshair
        if (border_0) begin
           case (last_pos0[11:9])
               RED     :   rgb <= {4'b1111, 8'b0};
               BLUE    :   rgb <= {8'b0, 4'b1111};
               GREEN   :   rgb <= {4'b0, 4'b1111, 4'b0};
               ORANGE  :   rgb <= {4'b1111, 4'd9, 4'b0};
               default :   ;
           endcase
        end else if (border_1) begin
           case (last_pos1[11:9])
               RED     :   rgb <= {4'b1111, 8'b0};
               BLUE    :   rgb <= {8'b0, 4'b1111};
               GREEN   :   rgb <= {4'b0, 4'b1111, 4'b0};
               ORANGE  :   rgb <= {4'b1111, 4'd9, 4'b0};
               default :   ;
           endcase
        end else if (border_2) begin
           case (last_pos2[11:9])
               RED     :   rgb <= {4'b1111, 8'b0};
               BLUE    :   rgb <= {8'b0, 4'b1111};
               GREEN   :   rgb <= {4'b0, 4'b1111, 4'b0};
               ORANGE  :   rgb <= {4'b1111, 4'd9, 4'b0};
               default :   ;
           endcase
        end else if (border_3) begin
           case (last_pos3[11:9])
               RED     :   rgb <= {4'b1111, 8'b0};
               BLUE    :   rgb <= {8'b0, 4'b1111};
               GREEN   :   rgb <= {4'b0, 4'b1111, 4'b0};
               ORANGE  :   rgb <= {4'b1111, 4'd9, 4'b0};
               default :   ;
           endcase
        end else if (border_4) begin
           case (last_pos4[11:9])
               RED     :   rgb <= {4'b1111, 8'b0};
               BLUE    :   rgb <= {8'b0, 4'b1111};
               GREEN   :   rgb <= {4'b0, 4'b1111, 4'b0};
               ORANGE  :   rgb <= {4'b1111, 4'd9, 4'b0};
               default :   ;
           endcase
        end else if (border_5) begin
           case (last_pos5[11:9])
               RED     :   rgb <= {4'b1111, 8'b0};
               BLUE    :   rgb <= {8'b0, 4'b1111};
               GREEN   :   rgb <= {4'b0, 4'b1111, 4'b0};
               ORANGE  :   rgb <= {4'b1111, 4'd9, 4'b0};
               default :   ;
           endcase
        end else if (border && (vcount < 240 && hcount < 320)) begin
           rgb <= {12{1'b1}};
        end else begin
           rgb <= cam | text_pixel_out;
        end
    end

    // the following lines are required for the Nexys4 VGA circuit - do not change
```

```
    assign vga_r = ~b ? rgb[11:8]: 0;
    assign vga_g = ~b ? rgb[7:4] : 0;
    assign vga_b = ~b ? rgb[3:0] : 0;

    assign vga_hs = ~hs;
    assign vga_vs = ~vs;

endmodule

module synchronize #(parameter NSYNC = 3)  // number of sync flops.  must be >= 2
                   (input clk,in,
                    output reg out);

  reg [NSYNC-2:0] sync;

  always_ff @ (posedge clk)
  begin
    {out,sync} <= {sync[NSYNC-2:0],in};
  end
endmodule

////////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version - 24 bits)
//
////////////////////////////////////////////////////////////////////////////////

module debounce (input reset_in, clock_in, noisy_in,
                 output reg clean_out);

   reg [19:0] count;
   reg new_input;

   always_ff @(posedge clock_in)
     if (reset_in) begin
        new_input <= noisy_in;
        clean_out <= noisy_in;
        count <= 0; end
     else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0; end
     else if (count == 650000) clean_out <= new_input;
     else count <= count+1;


endmodule

///////////////////////////////////////////////////////////////////////////////////
// Engineer:   g.p.hom
//
// Create Date:    18:18:59 04/21/2013
// Module Name:    display_8hex
// Description:  Display 8 hex numbers on 7 segment display
//
///////////////////////////////////////////////////////////////////////////////////

module display_8hex(
    input clk_in,                 // system clock
    input [31:0] data_in,         // 8 hex numbers, msb first
    output reg [6:0] seg_out,     // seven segment display output
    output reg [7:0] strobe_out   // digit strobe
```

```
    );

    localparam bits = 13;

    reg [bits:0] counter = 0;  // clear on power up

    wire [6:0] segments[15:0]; // 16 7 bit memorys
    assign segments[0]  = 7'b100_0000;  // inverted logic
    assign segments[1]  = 7'b111_1001;  // gfedcba
    assign segments[2]  = 7'b010_0100;
    assign segments[3]  = 7'b011_0000;
    assign segments[4]  = 7'b001_1001;
    assign segments[5]  = 7'b001_0010;
    assign segments[6]  = 7'b000_0010;
    assign segments[7]  = 7'b111_1000;
    assign segments[8]  = 7'b000_0000;
    assign segments[9]  = 7'b001_1000;
    assign segments[10] = 7'b000_1000;
    assign segments[11] = 7'b000_0011;
    assign segments[12] = 7'b010_0111;
    assign segments[13] = 7'b010_0001;
    assign segments[14] = 7'b000_0110;
    assign segments[15] = 7'b000_1110;

    always_ff @(posedge clk_in) begin
      // Here I am using a counter and select 3 bits which provides
      // a reasonable refresh rate starting the left most digit
      // and moving left.
      counter <= counter + 1;
      case (counter[bits:bits-2])
          3'b000: begin  // use the MSB 4 bits
                  seg_out <= segments[data_in[31:28]];
                  strobe_out <= 8'b0111_1111 ;
                 end

          3'b001: begin
                  seg_out <= segments[data_in[27:24]];
                  strobe_out <= 8'b1011_1111 ;
                 end

          3'b010: begin
                   seg_out <= segments[data_in[23:20]];
                   strobe_out <= 8'b1101_1111 ;
                  end
          3'b011: begin
                  seg_out <= segments[data_in[19:16]];
                  strobe_out <= 8'b1110_1111;
                 end
          3'b100: begin
                  seg_out <= segments[data_in[15:12]];
                  strobe_out <= 8'b1111_0111;
                 end

          3'b101: begin
                  seg_out <= segments[data_in[11:8]];
                  strobe_out <= 8'b1111_1011;
                 end

          3'b110: begin
```

```
                    seg_out <= segments[data_in[7:4]];
                    strobe_out <= 8'b1111_1101;
                   end
          3'b111: begin
                   seg_out <= segments[data_in[3:0]];
                   strobe_out <= 8'b1111_1110;
                  end

      endcase
     end

endmodule

////////////////////////////////////////////////////////////////////////////////
// Update: 8/8/2019 GH
// Create Date: 10/02/2015 02:05:19 AM
// Module Name: xvga
//
// xvga: Generate VGA display signals (1024 x 768 @ 60Hz)
//
//                              ---- HORIZONTAL -----     ------VERTICAL -----
//                              Active                    Active
//                    Freq      Video   FP  Sync   BP      Video   FP  Sync  BP
//   640x480, 60Hz    25.175    640     16    96   48       480    11   2    31
//   800x600, 60Hz    40.000    800     40   128   88       600     1   4    23
//   1024x768, 60Hz   65.000    1024    24   136  160       768     3   6    29
//   1280x1024, 60Hz  108.00    1280    48   112  248       768     1   3    38
//   1280x720p 60Hz   75.25     1280    72    80  216       720     3   5    30
//   1920x1080 60Hz   148.5     1920    88    44  148      1080     4   5    36
//
// change the clock frequency, front porches, sync's, and back porches to create
// other screen resolutions
////////////////////////////////////////////////////////////////////////////////

module xvga(input vclock_in,
            output reg [10:0] hcount_out,    // pixel number on current line
            output reg [9:0] vcount_out,     // line number
            output reg vsync_out, hsync_out,
            output reg blank_out);

   parameter DISPLAY_WIDTH  = 1024;      // display width
   parameter DISPLAY_HEIGHT = 768;       // number of lines

   parameter  H_FP = 24;                 // horizontal front porch
   parameter  H_SYNC_PULSE = 136;        // horizontal sync
   parameter  H_BP = 160;                // horizontal back porch

   parameter  V_FP = 3;                  // vertical front porch
   parameter  V_SYNC_PULSE = 6;          // vertical sync
   parameter  V_BP = 29;                 // vertical back porch

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
   assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
   assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));  //
       1183
```

```
   assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP - 1)); //1343

   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));  // 771
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE - 1));  // 777
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE + V_BP - 1)); // 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_ff @(posedge vclock_in) begin
      hcount_out <= hreset ? 0 : hcount_out + 1;
      hblank <= next_hblank;
      hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

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

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

endmodule
```

Camera Read

```
module camera_read(
        input  p_clock_in,
        input  vsync_in,
        input  href_in,
        input  [7:0] p_data_in,
        output logic [15:0] pixel_data_out,
        output logic pixel_valid_out,
        output logic frame_done_out
    );


        logic [1:0] FSM_state = 0;
    logic pixel_half = 0;

        localparam WAIT_FRAME_START = 0;
        localparam ROW_CAPTURE = 1;


        always_ff@(posedge p_clock_in)
        begin
        case(FSM_state)

        WAIT_FRAME_START: begin //wait for VSYNC
           FSM_state <= (!vsync_in) ? ROW_CAPTURE : WAIT_FRAME_START;
           frame_done_out <= 0;
           pixel_half <= 0;
        end

        ROW_CAPTURE: begin
           FSM_state <= vsync_in ? WAIT_FRAME_START : ROW_CAPTURE;
           frame_done_out <= vsync_in ? 1 : 0;
           pixel_valid_out <= (href_in && pixel_half) ? 1 : 0;
           if (href_in) begin
               pixel_half <= ~ pixel_half;
               if (pixel_half) pixel_data_out[7:0] <= p_data_in;
               else pixel_data_out[15:8] <= p_data_in;
           end
        end
        endcase
        end

endmodule
```

RGB to HSV Converter

```
`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;
                reg [31:0] hue_out;
                wire [17:0] s_res;
                wire [17:0] h_res;

                wire dout_valid_h;
                wire dout_valid_s;

                // Clocks 4-18: perform all the divisions
```

```
                //the s_divider (16/16) has delay 18
                //the hue_div (16/16) has delay 18

//              divider hue_div1(
//              .clk(clock),
//              .dividend(s_top),
//              .divisor(s_bottom),
//              .quotient(s_quotient),
//              // note: the "fractional" output was originally named "remainder" in this
//              // file -- it seems coregen will name this output "fractional" even if
//              // you didn't select the remainder type as fractional.
//              .fractional(s_remainder),
//              .rfd(s_rfd)
//              );
        div_gen_0 hue_div1(
                .aclk(clock),
                .s_axis_dividend_tdata(s_top),
                .s_axis_dividend_tvalid(1),
                .s_axis_divisor_tdata(s_bottom),
                .s_axis_divisor_tvalid(1),
                .m_axis_dout_tdata(s_res),
                // 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.
                .m_axis_dout_tvalid(dout_valid_s)
                );
                assign s_quotient = s_res[17:2];
                assign s_remainder = s_res[1:0];

                div_gen_0 hue_div2(
                .aclk(clock),
                .s_axis_dividend_tdata(h_top),
                .s_axis_dividend_tvalid(1),
                .s_axis_divisor_tdata(h_bottom),
                .s_axis_divisor_tvalid(1),
                .m_axis_dout_tdata(h_res),
                .m_axis_dout_tvalid(dout_valid_h)
                );

                assign h_quotient = h_res[17:2];
                assign h_remainder = h_res[1:0];

//              divider hue_div2(
//              .clk(clock),
//              .dividend(h_top),
//              .divisor(h_bottom),
//              .quotient(h_quotient),
//              .fractional(h_remainder),
//              .rfd(h_rfd)
//              );
                always @ (posedge clock) begin

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

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

```

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

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

        // Clock 3: compute the delta
        {my_r_delay2, my_g_delay2, my_b_delay2} <= {my_r_delay1, my_g_delay1, my_b_delay1};
        v_delay[0] <= max;
        delta <= max - min;

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


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

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


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

        v_delay[19] <= v_delay[18];
```

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

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

LED Position Extraction

```
`timescale 1ns / 1ps

module led_position_extraction(
                                input clock,
                                input reset,
                                input start,
                                input [7:0] h,
                                input [7:0] s,
                                input [7:0] v,
                                input [8:0] x,
                                input [7:0] y,
                                input [7:0] red_threshold,
                                input [7:0] blue_threshold,
                                input [7:0] green_threshold,
                                input [7:0] orange_threshold,
                                input [7:0] red_lower,
                                input [7:0] red_upper,
                                input [7:0] blue_lower,
                                input [7:0] blue_upper,
                                input [7:0] green_lower,
                                input [7:0] green_upper,
                                input [7:0] orange_lower,
                                input [7:0] orange_upper,
                                input [7:0] value_threshold,
                                output logic [11:0] led_positions [5:0],
                                output logic done,
                                output logic ready
    );
    parameter BIN_WIDTH = 10;               //width of each bin in pixels
    parameter BIN_HEIGHT = 15;              //height of each bin in pixels
    parameter NUM_BINS_X = 32;              //number of bins in the x direction
    parameter NUM_BINS_Y = 16;              //number of bins in the y direction
    parameter CAM_WIDTH = 320;
    parameter CAM_HEIGHT = 240;
//     parameter RED_THRESHOLD = 75;
//     parameter BLUE_THRESHOLD = 75;
//     parameter GREEN_THRESHOLD = 75;
//     parameter ORANGE_THRESHOLD = 75;
    localparam NO_COLOR = 0;
    localparam RED = 1;
    localparam BLUE = 2;
    localparam GREEN = 3;
    localparam ORANGE = 4;

    //tallies the number of red pixels found in the bin currently being accessed
    logic [3:0] temp_red_count;
    logic [3:0] temp_blue_count;
    logic [3:0] temp_green_count;
    logic [3:0] temp_orange_count;

    logic [2:0] current_pixel_color;        //color of the currently observed pixel

    //total value to be stored in the pixel count BRAM
    logic [7:0] value_to_red_bram;
    logic [7:0] value_to_blue_bram;
    logic [7:0] value_to_green_bram;
```

```
    logic [7:0] value_to_orange_bram;

    //previous total value from the pixel count BRAM
    logic [7:0] value_from_red_bram;
    logic [7:0] value_from_blue_bram;
    logic [7:0] value_from_green_bram;
    logic [7:0] value_from_orange_bram;

    logic valid_values;                    //true when the last pixel of the current bin has been counted

    logic [8:0] bin_bram_address;          //muxed bin address

    logic [5:0] x_counter;                 //counter used to track intra-bin position
    logic [8:0] x_delay;                   //used to delay x 1 cycle

    logic cell_determiner_done;            //true once all bins have been summed

    logic [8:0] cell_determiner_address;   //address of the bin current being summed
    logic [8:0] finished_bin_address;      //addresses bins after they have bin tallied to determine their overall color

    logic [2:0] color_grid [15:0][31:0];   //16x32 array used to avoid double counting of leds
    logic [4:0] x_pos;                     //x position used to access color_grid
    logic [3:0] y_pos;                     //y position used to access color_grid

    //used to reset color_grid
    integer i;
    integer j;

    //used to reset led positions
    integer k;

    logic [1:0] delay_counter;             //added delay to account for BRAM access latency
    logic [2:0] output_counter;            //counts the number of led positions that have been assigned a position

    logic started;                         //true after the module has been started
    logic internal_reset;                  //used to reset the module after a frame has been processed in preparation for the next fram
    logic cell_determiner_reset;           //resets the cell_determiner module
    logic resetting;                       //true when the module is interally resetting

    assign cell_determiner_reset = (internal_reset || reset);
    assign bin_bram_address = (cell_determiner_done) ? finished_bin_address : cell_determiner_address;

    //Instantiates cell_detmerminer module
    //Determines the BRAM address for the current bin
    cell_determiner #(.BIN_WIDTH(BIN_WIDTH), .BIN_HEIGHT(BIN_HEIGHT), .NUM_BINS_X(NUM_BINS_X), .NUM_BINS_Y(NUM_BINS_Y),. CAM_WIDTH(CAM_WIDTH),
        .CAM_HEIGHT(CAM_HEIGHT)) my_cell_determiner (.clock(clock), .reset(cell_determiner_reset), .start(start), .x(x), .y(y),
        .bin_num(cell_determiner_address), .done(cell_determiner_done));

    //Instantiates pixel_color_threshold module
```

```
    //Determines the color of the current pixel
    pixel_color_threshold my_color_threshold (.h(h), .v(v), .color(current_pixel_color
        ), .red_lower(red_lower),
        .red_upper(red_upper), .blue_lower(blue_lower), .blue_upper(blue_upper), .
            green_lower(green_lower),
        .green_upper(green_upper), .orange_lower(orange_lower), .orange_upper(
            orange_upper),
        .value_threshold(value_threshold));


    //each of the following BRAMs tracks the count of pixels of each
    //respective color for each bin in the 16x32 grid
    blk_mem_gen_1 red_BRAM (
      .clka(clock),
      .wea(valid_values),
      .addra(bin_bram_address),
      .dina(value_to_red_bram),
      .clkb(clock),
      .rstb(0),
      .addrb(bin_bram_address),
      .doutb(value_from_red_bram),
      .rsta_busy(red_rsta_busy),
      .rstb_busy(red_rstb_busy)
    );

    blk_mem_gen_1 blue_BRAM (
      .clka(clock),
      .wea(valid_values),
      .addra(bin_bram_address),
      .dina(value_to_blue_bram),
      .clkb(clock),
      .rstb(0),
      .addrb(bin_bram_address),
      .doutb(value_from_blue_bram),
      .rsta_busy(blue_rsta_busy),
      .rstb_busy(blue_rstb_busy)
    );

    blk_mem_gen_1 green_BRAM (
      .clka(clock),
      .wea(valid_values),
      .addra(bin_bram_address),
      .dina(value_to_green_bram),
      .clkb(clock),
      .rstb(0),
      .addrb(bin_bram_address),
      .doutb(value_from_green_bram),
      .rsta_busy(green_rsta_busy),
      .rstb_busy(green_rstb_busy)
    );

    blk_mem_gen_1 orange_BRAM (
      .clka(clock),
      .wea(valid_values),
      .addra(bin_bram_address),
      .dina(value_to_orange_bram),
      .clkb(clock),
      .rstb(0),
      .addrb(bin_bram_address),
```

```
      .doutb(value_from_orange_bram),
      .rsta_busy(orange_rsta_busy),
      .rstb_busy(orange_rstb_busy)
    );

    always_ff @(posedge clock) begin
        if (reset || internal_reset) begin
            temp_red_count <= 0;
            temp_blue_count <= 0;
            temp_green_count <= 0;
            temp_orange_count <= 0;
            x_counter <= 0;
            valid_values <= 0;
            finished_bin_address <= 0;

            //resets the color grid
            for (i=0; i<NUM_BINS_Y; i=i+1) begin
                for (j=0; j<NUM_BINS_X; j=j+1) begin
                    color_grid[i][j] <= 3'b0;
                end
            end
            x_pos <= 0;
            y_pos <= 0;
            delay_counter <= 0;
            output_counter <= 0;
            done <= 0;
            started <= 0;
            internal_reset <= 0;

            //reset the output array to 0
            for (k=0; k<6; k=k+1) begin
                led_positions[k] <= 12'b0;
            end
            resetting <= 0;
            ready <= 1;
        end else begin
            if (start) begin
                started <= 1;
                ready <= 0;
            end
            if (done || resetting) begin
                if (done) begin
                    resetting <= 1;
                    finished_bin_address <= 0;
                    done <= 0;
                    valid_values <= 1;
                end else if (finished_bin_address < 511) begin
                    finished_bin_address <= finished_bin_address + 1;
                end else begin
                    internal_reset <= 1;
                end

            end
            if (start || started) begin
                if (output_counter > 5) begin //Once all 6 outputs have been
                    determined
                    done <= 1;
                    started <= 0;
                end
```

```
if (cell_determiner_done) begin //After all pixels have been interated over, start interating over bins
    value_to_red_bram <= 0;
    value_to_blue_bram <= 0;
    value_to_green_bram <= 0;
    value_to_orange_bram <= 0;
    finished_bin_address <= finished_bin_address + 1;
    if (delay_counter < 2) begin //Delay to account for BRAM read time; matches the value from BRAM to the correct x,y position
        delay_counter <= delay_counter + 1;
    end else begin
        if(x_pos == NUM_BINS_X-1) begin //updates the x,y position for the current bin being accessed from BRAM
            x_pos <= 0;
            y_pos <= y_pos + 1;
        end else x_pos <= x_pos + 1;

        //Classify bin based on num of colored pixels of one color
        //if num is > threshold for that color, count the bin
        //color priority is red -> blue -> green -> orange
        if (value_from_red_bram > red_threshold) begin
            //Checks that no surrounding bins are the same color; eliminates the possibility of double counting one LED
            if (color_grid[y_pos-1][x_pos]!=RED && color_grid[y_pos+1][x_pos]!=RED && color_grid[y_pos][x_pos+1]!=RED && color_grid[y_pos][x_pos-1]!=RED) begin
                color_grid[y_pos][x_pos] <= RED;
                led_positions[output_counter] <= {3'd1, x_pos, y_pos};
                output_counter <= output_counter + 1;
            end
        end else if (value_from_blue_bram > blue_threshold) begin
            if (color_grid[y_pos-1][x_pos]!=BLUE && color_grid[y_pos+1][x_pos]!=BLUE && color_grid[y_pos][x_pos+1]!=BLUE && color_grid[y_pos][x_pos-1]!=BLUE) begin
                color_grid[y_pos][x_pos] <= BLUE;
                led_positions[output_counter] <= {3'd2, x_pos, y_pos};
                output_counter <= output_counter + 1;
            end
        end else if (value_from_green_bram > green_threshold) begin
            if (color_grid[y_pos-1][x_pos]!=GREEN && color_grid[y_pos+1][x_pos]!=GREEN && color_grid[y_pos][x_pos+1]!=GREEN && color_grid[y_pos][x_pos-1]!=GREEN) begin
                color_grid[y_pos][x_pos] <= GREEN;
                led_positions[output_counter] <= {3'd3, x_pos, y_pos};
                output_counter <= output_counter + 1;
            end
        end else if (value_from_orange_bram > orange_threshold) begin
            if (color_grid[y_pos-1][x_pos]!=ORANGE && color_grid[y_pos+1][x_pos]!=ORANGE && color_grid[y_pos][x_pos+1]!=ORANGE && color_grid[y_pos][x_pos-1]!=ORANGE) begin
                color_grid[y_pos][x_pos] <= ORANGE;
                led_positions[output_counter] <= {3'd4, x_pos, y_pos};
                output_counter <= output_counter + 1;
            end
        end else ; //no color in the bin was above the threshold
        if (x_pos == NUM_BINS_X-1 && y_pos == NUM_BINS_Y-1) begin
            done <= 1;
            started <= 0;
```

```
                        end
                    end
                end else begin  //Iterating over all the pixels from the camera (in HSV format)
                    if(x_counter == BIN_WIDTH-1) begin  //Adds the current amount of pixels counted to the amount previous counted and sends to BRAM
                        valid_values <= 1;
                        value_to_red_bram <= (current_pixel_color == RED) ? (temp_red_count + 1) + value_from_red_bram : temp_red_count + value_from_red_bram;
                        value_to_blue_bram <= (current_pixel_color == BLUE) ? (temp_blue_count + 1) + value_from_blue_bram : temp_blue_count + value_from_blue_bram;
                        value_to_green_bram <= (current_pixel_color == GREEN) ? (temp_green_count + 1) + value_from_green_bram : temp_green_count + value_from_green_bram;
                        value_to_orange_bram <= (current_pixel_color == ORANGE) ? (temp_orange_count + 1) + value_from_orange_bram : temp_orange_count + value_from_orange_bram;
                        temp_red_count <= 0;
                        temp_blue_count <= 0;
                        temp_green_count <= 0;
                        temp_orange_count <= 0;
                        x_counter <= 0;

                    end else begin  //adds the current pixel to a temp value if it falls into one of the four colors
                        valid_values <= 0;
                        x_counter <= x_counter + 1;
                        case (current_pixel_color)
                            RED     :   temp_red_count <= temp_red_count + 1;
                            BLUE    :   temp_blue_count <= temp_blue_count + 1;
                            GREEN   :   temp_green_count <= temp_green_count + 1;
                            ORANGE  :   temp_orange_count <= temp_orange_count + 1;
                            default :   ;
                        endcase
                    end
                end
            end
        end
    end

endmodule

//////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
module cell_determiner(
                        input clock,
                        input reset,
                        input start,
                        input [8:0] x,
                        input [7:0] y,
                        output logic [8:0] bin_num,
                        output logic done
    );
    parameter BIN_WIDTH = 10;
    parameter BIN_HEIGHT = 15;
    parameter NUM_BINS_X = 32;
    parameter NUM_BINS_Y = 16;
```

```
    parameter CAM_WIDTH = 320;
    parameter CAM_HEIGHT = 240;

    logic [5:0] x_counter;
    logic [5:0] y_counter;
    logic [8:0] x_delay;
    logic started;

    always_ff @(posedge clock) begin
        if(reset) begin
            x_counter <= 0;
            y_counter <= 0;
            bin_num <= 0;
            done <= 0;
            started <= 0;
            x_delay <= 0;
        end else if (done) begin
            //do nothing
        end else if (start || started) begin
            if (start) started <= 1;    //enter the started state after receiving a start signal
            x_delay <= x;
            if (x_delay==CAM_WIDTH-1) begin                          //at the edge of the fram
                x_counter <= 1;
                if (y == CAM_HEIGHT-1) begin                         //frame has been finished
                    done <= 1;
                    started <= 0;
                end else if (y_counter == BIN_HEIGHT-1) begin   //next line is in a different bin
                    y_counter <= 0;
                    bin_num <= bin_num + 1;
                end else begin                                       //next line is in the same bin
                    y_counter <= y_counter + 1;
                    bin_num <= bin_num - (NUM_BINS_X-1);
                end
            end else if (x_counter < BIN_WIDTH) begin           //continues within the same bin
                x_counter <= x_counter + 1;
            end else if (x_counter == BIN_WIDTH) begin          //increments the bin counter once the edge of the current bin has been reached
                x_counter <= 1;
                bin_num <= bin_num + 1;
            end
        end
    end
endmodule

/////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////
module pixel_color_threshold (
                                input [7:0] h,
                                input [7:0] v,
                                input [7:0] red_lower,
                                input [7:0] red_upper,
                                input [7:0] blue_lower,
                                input [7:0] blue_upper,
```

```
                                input [7:0] green_lower,
                                input [7:0] green_upper,
                                input [7:0] orange_lower,
                                input [7:0] orange_upper,
                                input [7:0] value_threshold,
                                output logic [2:0] color
    );
    parameter RED_LOWER_THRESH = 248;
    parameter RED_UPPER_THRESH = 4;
    parameter BLUE_LOWER_THRESH = 145;
    parameter BLUE_UPPER_THRESH = 177;
    parameter GREEN_LOWER_THRESH = 57;
    parameter GREEN_UPPER_THRESH = 99;
    parameter ORANGE_LOWER_THRESH = 28;
    parameter ORANGE_UPPER_THRESH = 50;
    parameter VALUE_THRESHOLD = 153; //60% value

    localparam NO_COLOR = 0;
    localparam RED = 1;
    localparam BLUE = 2;
    localparam GREEN = 3;
    localparam ORANGE = 4;

    always_comb begin
        color = 0; //default
        if (v > value_threshold) begin
            if (h > green_lower && h < green_upper) color = GREEN;
            else if(h > red_lower || h < red_upper) color = RED;
            else if (h > orange_lower && h < orange_upper) color = ORANGE;
            else if (h > blue_lower && h < blue_upper) color = BLUE;
        end
    end

endmodule
```

Finger Finder

```
`timescale 1ns / 1ps

module finger_finder(
                                input clock,
                                input reset,
                                input [11:0] led_positions [5:0],
                                input start,
                                output logic done,
                                output logic [8:0] thumb, //blue
                                output logic [8:0] index, //orange
                                output logic [8:0] middle, //green
                                output logic [8:0] ring, //red
                                output logic [8:0] pinkie, //blue
                                output logic [8:0] palm //red
);
    logic [11:0] pos_color [5:0];
    logic [3:0] count;
    logic [8:0] pos0;
    logic [2:0] col0;
    logic [8:0] pos1;
    logic [2:0] col1;
    logic [8:0] pos2;
    logic [2:0] col2;
    logic [8:0] pos3;
    logic [2:0] col3;
    logic [8:0] pos4;
    logic [2:0] col4;
    logic [8:0] pos5;
    logic [2:0] col5;

    localparam NO_COLOR = 0;
    localparam RED = 1;
    localparam BLUE = 2;
    localparam GREEN = 3;
    localparam ORANGE = 4;

    always_ff @(posedge clock) begin
        if (reset) begin //reset signal, new counter and values
            count <= 0;
            done <= 0;
            thumb <= 0;
            index <= 0;
            middle <= 0;
            ring <= 0;
            pinkie <= 0;
            palm <= 0;
        end else if(start) begin //new values to save, reset output and counter
            pos_color <= led_positions;
            count <= 0;
            done <= 0;
            thumb <= 0;
            index <= 0;
            middle <= 0;
            ring <= 0;
            pinkie <= 0;
            palm <= 0;
```

```
            pos0 <= led_positions[0][8:0]; //separates array into variables
            col0 <= led_positions[0][11:9];
            pos1 <= led_positions[1][8:0];
            col1 <= led_positions[1][11:9];
            pos2 <= led_positions[2][8:0];
            col2 <= led_positions[2][11:9];
            pos3 <= led_positions[3][8:0];
            col3 <= led_positions[3][11:9];
            pos4 <= led_positions[4][8:0];
            col4 <= led_positions[4][11:9];
            pos5 <= led_positions[5][8:0];
            col5 <= led_positions[5][11:9];

         //look at first item in array
        end else if (count == 0) begin
            count <= count + 1;
            if(col0 == GREEN) begin //green light = middle finger
                middle <= pos0;
            end else if(col0 == ORANGE) begin //orange light = index finger
                index <= pos0;
            end else if (col0 == BLUE) begin //blue light = thumb (for now)
                thumb <= pos0;
            end else if (col0 == RED) begin //red light = palm (for now)
                palm <= pos0;
            end else begin //no color, we're done
                count <= 6;
                done <= 1;
            end

         //second item in array, index 1
        end else if (count == 1) begin
            count <= count + 1;
            if(col1 == GREEN) begin //green light = middle finger
                middle <= pos1;
            end else if(col1 == ORANGE) begin //orange light = index finger
                index <= pos1;
            end else if(col1 == BLUE) begin //blue light
                if(thumb == 0) begin //first blue light = thumb
                    thumb <= pos1;
                end else if(pos1[8:4] > thumb[8:4]) begin //this light left of previous, new thumb
                    pinkie <= thumb;
                    thumb <= pos1;
                end else begin //this light is to the right, is the pinkie
                    pinkie <= pos1;
                end
            end else if (col1 == RED) begin //red light
                if (palm == 0) begin //first red light = palm
                    palm <= pos1;
                end else if(pos1[3:0] > palm[3:0]) begin //this light is lower than previous, new palm
                    ring <= palm;
                    palm <= pos1;
                end else begin // this light is above the previous, ring finger
                    ring <= pos1;
                end
            end else begin //no color, we're done
                count <= 6;
                done <= 1;
```

```
            end

         //third item in array, index 2
        end else if (count == 2) begin
            count <= count + 1;
            if(col2 == GREEN) begin //green light = middle finger
                middle <= pos2;
            end else if(col2 == ORANGE) begin //orange light = index finger
                index <= pos2;
            end else if(col2 == BLUE) begin //blue light
                if(thumb == 0) begin //first blue light = thumb
                    thumb <= pos2;
                end else if(pos2[8:4] > thumb[8:4]) begin //this light left of previous, new thumb
                    pinkie <= thumb;
                    thumb <= pos2;
                end else begin //this light is to the right, is the pinkie
                    pinkie <= pos2;
                end
            end else if (col2 == RED) begin //red light
                if (palm == 0) begin //first red light = palm
                    palm <= pos2;
                end else if(pos2[3:0] > palm[3:0]) begin //this light is lower than previous, new palm
                    ring <= palm;
                    palm <= pos2;
                end else begin // this light is above the previous, ring finger
                    ring <= pos2;
                end
            end else begin //no color, we're done
                count <= 6;
                done <= 1;
            end

        //fourth item in array, index 3
        end else if (count == 3) begin
            count <= count + 1;
            if(col3 == GREEN) begin //green light = middle finger
                middle <= pos3;
            end else if(col3 == ORANGE) begin //orange light = index finger
                index <= pos3;
            end else if(col3 == BLUE) begin //blue light
                if(thumb == 0) begin //first blue light = thumb
                    thumb <= pos3;
                end else if(pos3[8:4] > thumb[8:4]) begin //this light left of previous, new thumb
                    pinkie <= thumb;
                    thumb <= pos3;
                end else begin //this light is to the right, is the pinkie
                    pinkie <= pos3;
                end
            end else if (col3 == RED) begin //red light
                if (palm == 0) begin //first red light = palm
                    palm <= pos3;
                end else if(pos3[3:0] > palm[3:0]) begin //this light is lower than previous, new palm
                    ring <= palm;
                    palm <= pos3;
                end else begin // this light is above the previous, ring finger
```

```
                    ring <= pos3;
                end
            end else begin //no color, we're done
                count <= 6;
                done <= 1;
            end

        //fifth item in array, index 4
        end else if (count == 4) begin
            count <= count + 1;
            if(col4 == GREEN) begin //green light = middle finger
                middle <= pos4;
            end else if(col4 == ORANGE) begin //orange light = index finger
                index <= pos4;
            end else if(col4 == BLUE) begin //blue light
                if(thumb == 0) begin //first blue light = thumb
                    thumb <= pos4;
                end else if(pos4[8:4] > thumb[8:4]) begin //this light left of previous, new thumb
                    pinkie <= thumb;
                    thumb <= pos4;
                end else begin //this light is to the right, is the pinkie
                    pinkie <= pos4;
                end
            end else if (col4 == RED) begin //red light
                if (palm == 0) begin //first red light = palm
                    palm <= pos4;
                end else if(pos4[3:0] > palm[3:0]) begin //this light is lower than previous, new palm
                    ring <= palm;
                    palm <= pos4;
                end else begin // this light is above the previous, ring finger
                    ring <= pos4;
                end
            end else begin //no color, we're done
                count <= 6;
                done <= 1;
            end

        //sixth item in array, index 5
        end else if (count == 5) begin
            count <= count + 1;
            if(col5 == GREEN) begin //green light = middle finger
                middle <= pos5;
            end else if(col5 == ORANGE) begin //orange light = index finger
                index <= pos5;
            end else if(col5 == BLUE) begin //blue light
                if(thumb == 0) begin //first blue light = thumb
                    thumb <= pos5;
                end else if(pos5[8:4] > thumb[8:4]) begin //this light left of previous, new thumb
                    pinkie <= thumb;
                    thumb <= pos5;
                end else begin //this light is to the right, is the pinkie
                    pinkie <= pos5;
                end
            end else if (col5 == RED) begin //red light
                if (palm == 0) begin //first red light = palm
                    palm <= pos5;
```

```
                end else if(pos5[3:0] > palm[3:0]) begin //this light is lower than previous, new palm
                    ring <= palm;
                    palm <= pos5;
                end else begin // this light is above the previous, ring finger
                    ring <= pos5;
                end
            end else begin //no color, we're done
                count <= 6;
                done <= 1;
            end
            done <= 1; //set done to 1 to signal we've finished

        //count > 5, all inputs have been processed
        end else begin
            done <= 0; //reset done signal
        end

    end
endmodule
```

Gesture Recognition

```
`timescale 1ns / 1ps

module gesture_recognition_fsm(
                                input clock,
                                input reset,
                                input [8:0] palm, //palm
                                input [8:0] thumb, //thumb
                                input [8:0] index, //index
                                input [8:0] middle, //middle
                                input [8:0] ring, //ring
                                input [8:0] pinkie, //pinkie
                                input start,
                                output logic [4:0] letter,
                                output logic done
    );
    parameter IDLE = 'b000;
    parameter LED_DET = 'b001;
    parameter STAT_REC = 'b011;
    parameter DYNAMIC = 'b111;
    parameter DEB = 'b010;

    parameter A_XDIFF1 = 'd7;
    parameter T_E_XDIFF1 = 'd0;
    parameter S_XDIFF1 = 'd4;
    parameter E_YDIFF1 = 'd5;
    parameter V_XDIFF1 = 'd3;
    parameter K_YDIFF1 = 'd6;
    parameter O_YDIFF1 = 'd5;
    parameter G_YDIFF1 = 'd3;
    parameter H_YDIFF = 'd3;
    parameter H_XDIFF = 'd10;

    parameter DEB_COUNT = 'd3;
    parameter J_TIME = 195_000_000;

    logic [4:0] prev_letter;
    logic [8:0] count;

    logic i;
    logic d;
    logic z;
    logic z_flag;
    logic [27:0] z_count;
    logic j;
    logic j_flag;
    logic [27:0] j_count;
    logic [2:0] state;

    is_jay jay (.clock(clock),.reset(reset),.start(i),.done(j), .letter(letter), .fsm_done(done),
                .palm(palm),.thumb(thumb),.index(index),.middle(middle),.ring(ring),.pinkie(pinkie));

    is_zee zee (.clock(clock),.reset(reset),.start(d),.done(z),.letter(letter),.fsm_done(done),
                .palm(palm),.thumb(thumb),.index(index),.middle(middle),.ring(ring),.
```

```
                    pinkie(pinkie));

    always_ff @ (posedge clock) begin
        if(reset) begin //reset signal
            done <= 0;
            state <= IDLE;
            i <= 0;
            d <= 0;
            z_count <= 0;
            z_flag <= 0;
            j_count <= 0;
            j_flag <= 0;
        end else if (j || j_flag) begin //is_jay module signals j has been signed
            i <= 0;                      //and we haven't displayed long enough yet
            letter <= 5'd10;
            done <= 1;
            if(j_count == J_TIME) begin //we have displayed for J_TIME cycles, stop
                j_flag <= 0;
                j_count <= 0;
            end else begin //continue displaying, increment time
                j_count <= j_count + 1;
                j_flag <= 1;
            end
        end else if (z || z_flag) begin //is_zee module signals z has been signed
            d <= 0;                      //and we haven't displayed long enough yet
            letter <= 5'd26;
            done <= 1;
            if (z_count == J_TIME) begin //we have displayed for J_TIME cycles, stop
                z_flag <= 0;
                z_count <= 0;
            end else begin //continue displaying, increment time
                z_count <= z_count + 1;
                z_flag <= 1;
            end

        end else begin //state machine for static recognition
            case (state)
                IDLE: begin //idle state waits for valid LED positions
                    done <= 0;
                    i <= 0;
                    d <= 0;
                    if(start) begin //positions are valid
                        state <= STAT_REC;
                    end
                end

                STAT_REC: begin //identify letter being signed
                    //a, e, t, s, or n
                    if (index == 0 && middle == 0 && ring == 0 && pinkie == 0 && thumb != 0 && palm != 0) begin
                        if(thumb[8:4] > palm[8:4] + A_XDIFF1) begin //a
                            letter <= 5'd1;
                            state <= DEB;
                        end else if(thumb[8:4] > palm[8:4] + S_XDIFF1) begin //s
                            letter <= 5'd19;
                            state <= DEB;
                        end else if(thumb[8:4] > palm[8:4] + T_E_XDIFF1) begin //e or t
                            if (thumb[3:0]  + E_YDIFF1 < palm[3:0]) begin //t
```

```
                            letter <= 5'd20;
                            state <= DEB;
                        end else begin //e
                            letter <= 5'd5;
                            state <= DEB;
                        end
                    end else begin //n
                        letter <= 5'd14;
                        state <= DEB;
                    end
                //space
                end else if (index == 0 && middle == 0 && ring == 0 && pinkie == 0 && thumb == 0 && palm == 0) begin
                    letter <= 0;
                    state <= DEB;
                //m
                end else if (index == 0 && middle == 0 && ring == 0 && pinkie == 0 && thumb == 0 && palm != 0) begin
                    letter <= 5'd13;
                    state <= DEB;
                //f
                end else if (index == 0 && middle != 0 && ring != 0 && pinkie != 0 && thumb != 0 && palm != 0) begin
                    letter <= 5'd6;
                    state <= DEB;
                //y or i
                end else if (index == 0 && middle == 0 && ring == 0 && pinkie != 0 && thumb != 0 && palm != 0) begin
                    if(thumb[8:4] > palm[8:4] + A_XDIFF1) begin //y
                        letter <= 5'd25;
                        state <= DEB;
                    end else begin //i
                        letter <= 5'd9;
                        state <= DEB;
                    end
                //q, g, l, d, or x
                end else if (index != 0 && middle == 0 && ring == 0 && pinkie == 0 && thumb != 0 && palm != 0) begin
                    if (thumb[3:0] > palm[3:0]) begin //q
                        letter <= 5'd17;
                        state <= DEB;
                    end else if(thumb[3:0] < index[3:0] + G_YDIFF1) begin //g
                        letter <= 5'd7;
                        state <= DEB;
                    end else if(thumb[8:4] > palm[8:4] + A_XDIFF1) begin //l
                        letter <= 5'd12;
                        state <= DEB;
                    end else if (thumb[8:4] > palm[8:4] + T_E_XDIFF1) begin //d
                        letter <= 5'd4;
                        state <= DEB;
                    end else begin //x
                        letter <= 5'd24;
                        state <= DEB;
                    end
                //c, o or b
                end else if (index != 0 && middle != 0 && ring != 0 && pinkie != 0 && thumb != 0 && palm != 0) begin
                    if(thumb[8:4] > palm[8:4] + A_XDIFF1) begin //c
                        letter <= 5'd3;
```

```
                state <= DEB;
            end else if (thumb[3:0]  + O_YDIFF1 < palm[3:0]) begin //o
                letter <= 5'd15;
                state <= DEB;
            end else begin //b
                letter <= 5'd2;
                state <= DEB;
            end
        //h, p, k, r, v, or u
        end else if (index != 0 && middle != 0 && ring == 0 && pinkie == 0 && thumb != 0 && palm != 0) begin
            if (middle[8:4] + H_XDIFF < palm[8:4]) begin
                letter <= 5'd8;
                state <= DEB;
            end else if (middle[3:0] > thumb[3:0]) begin //p
                letter <= 5'd16;
                state <= DEB;
            end else if (thumb[3:0]  + K_YDIFF1 < palm[3:0]) begin //k
                letter <= 5'd11;
                state <= DEB;
            end else if(middle[8:4] > index[8:4]) begin//r
                letter <= 5'd18;
                state <= DEB;
            end else if (index[8:4] > middle[8:4] + V_XDIFF1) begin //v
                letter <= 5'd22;
                state <= DEB;
            end else begin //u
                letter <= 5'd21;
                state <= DEB;
            end
        //w
        end else if (index != 0 && middle != 0 && ring != 0 && pinkie == 0 && palm != 0) begin
            letter <= 5'd23;
            state <= DEB;
        //no sign
        end else begin
            letter <= 0;
            state <= IDLE;
        end
    end

    DEB: begin //checks sign is consistent for long enough and sets done signal for display
        if(letter == prev_letter) begin //consistent
            if(count == DEB_COUNT) begin //for long enough, trigger display
                done <= 1;
                count <= 0;
                if(letter == 9) begin //triggers is_jay module
                    i <= 1;
                end else if (letter == 4) begin //triggers is_zee module
                    d <= 1;
                end
            end else begin
                count <= count + 1;
            end
        end else begin //new letter, reset count, no done signal to display
```

```
                        count <= 0;
                    end
                    prev_letter <= letter;
                    state <= IDLE;
                end
            endcase
        end
    end
endmodule
```

J Identification

```
module is_jay (
                input clock,
                input reset,
                input start,
                input fsm_done,
                input [4:0] letter,
                input [8:0] palm, //palm
                input [8:0] thumb, //thumb
                input [8:0] index, //index
                input [8:0] middle, //middle
                input [8:0] ring, //ring
                input [8:0] pinkie, //pinkie
                output logic done
    );
    localparam VALID_Y_SHIFT = 1;               //downward shift necessary to register as having moved down
    localparam VALID_X_SHIFT = 8;               //rightward shift necessary to register as having moved right

    localparam IDLE = 2'b00;
    localparam ACTIVATED = 2'b01;
    localparam SHIFTED_DOWN = 2'b11;
    localparam FINISHED = 2'b10;

    localparam BIN_X_WIDTH = 31;                //number of bins in the x direction
    localparam BIN_Y_HEIGHT = 15;               //number of bins in the y direction

    logic [1:0] recognition_state;
    logic [8:0] last_pinkie;                    //first recognized position of pinkie when starting the module
    logic [8:0] last_thumb;                     //first recognized position of thumb when starting the module
    logic [8:0] actually_pinkie;                //variable used for clarity

    //due to necessary previous design choices, the thumb position that is passed to this module while signing
    //j will actually be the pinkie of the user, as such, this is an alias for the thumb
    assign actually_pinkie = thumb[8:0];

    always_ff @(posedge clock) begin
        if (reset) begin
            done <= 0;
            recognition_state <= IDLE;
        end else begin
            case (recognition_state)
                IDLE        :   begin
                                    if (start) begin                        //once a "i" has been recognized
                                        recognition_state <= ACTIVATED;
                                        last_pinkie <= pinkie;
                                        last_thumb <= thumb;
                                    end
                                    done <= 0;
                                end
                ACTIVATED   :   begin
```

```
                    if (letter != 9 && fsm_done) begin      //reset if another letter is recognized
                        recognition_state <= IDLE;
                    end else if (last_thumb[3:0] < BIN_Y_HEIGHT - VALID_Y_SHIFT) begin
                        //if the pinkie has moved far enough down, proceed
                        if (actually_pinkie[3:0] >= last_thumb[3:0] + VALID_Y_SHIFT) recognition_state <= SHIFTED_DOWN;
                    end else recognition_state <= IDLE;
                end
        SHIFTED_DOWN:   begin
                    if (letter != 9 && fsm_done) begin
                        recognition_state <= IDLE;
                    end else if ((last_pinkie[8:4] < BIN_X_WIDTH - VALID_X_SHIFT)) begin
                        //go to FINISHED state if pinkie moves right enough
                        if (actually_pinkie[8:4] >= last_pinkie[8:4] + VALID_X_SHIFT) recognition_state <= FINISHED;
                    end else recognition_state <= IDLE;
                end
        FINISHED    :   begin
                    done <= 1;
                    recognition_state <= IDLE;
                end
            endcase
        end
    end
endmodule
```

Z Identification

```
module is_zee (
                input clock,
                input reset,
                input start,
                input fsm_done,
                input [4:0] letter,
                input [8:0] palm, //palm
                input [8:0] thumb, //thumb
                input [8:0] index, //index
                input [8:0] middle, //middle
                input [8:0] ring, //ring
                input [8:0] pinkie, //pinkie
                output logic done
    );

    //respective shifts required to register in a certain direction
    localparam VALID_LEFT_SHIFT = 8;
    localparam VALID_DOWN_SHIFT = 5;

    localparam IDLE = 3'b000;
    localparam ACTIVATED = 3'b001;
    localparam SHIFTED_LEFT = 3'b011;
    localparam SHIFTED_DOWN_RIGHT = 3'b111;
    localparam FINISHED = 3'b110;

    localparam BIN_X_WIDTH = 31;
    localparam BIN_Y_HEIGHT = 15;

    logic [2:0] recognition_state;
    logic [8:0] last_index;

    always_ff @(posedge clock) begin
        if (reset) begin
            done <= 0;
            recognition_state <= IDLE;
        end else begin
            case (recognition_state)
                IDLE                :   begin
                                            if (start) begin                    //once a "d" has been recognized
                                                recognition_state <= ACTIVATED;
                                                last_index <= index;
                                            end
                                            done <= 0;
                                        end
                ACTIVATED           :   begin
                                            if (letter != 4 && fsm_done) recognition_state <= IDLE;
                                            else if ((last_index[8:4] > VALID_LEFT_SHIFT) && (last_index[3:0] < BIN_Y_HEIGHT - VALID_DOWN_SHIFT)) begin
                                                //if the index finger has moved far enough left, proceed
                                                //the index != 0 check has been added to account for position instability
```

```
                                if ((index[8:4] <= last_index[8:4] - VALID_LEFT_SHIFT) && index[8:4] != 0) recognition_state <= SHIFTED_LEFT;
                            end else recognition_state <= IDLE;
                        end
                SHIFTED_LEFT      :   begin
                            if (letter != 4 && fsm_done) recognition_state <= IDLE;
                            //if the index finger has moved far enough down and right, proceed
                            else if ((index[8:4] >=  last_index[8:4]) && (index[3:0] >= last_index[3:0] + VALID_DOWN_SHIFT)) recognition_state <= SHIFTED_DOWN_RIGHT;
                        end
                SHIFTED_DOWN_RIGHT  :   begin
                            if (letter != 4 && fsm_done) recognition_state <= IDLE;
                            //if the index finger has moved left enough, finish
                            else if ((index[8:4] <= last_index[8:4] - VALID_LEFT_SHIFT) && index[8:4] != 0) recognition_state <= FINISHED;
                        end
                FINISHED          :   begin
                            done <= 1;
                            recognition_state <= IDLE;
                        end
                default :   ;
            endcase
        end
    end
endmodule
```

Text Display

```
module text_display (
                    input pixel_clk_in,
                    input reset,
                    input [10:0] hcount_in,
                    input [9:0] vcount_in,
                    input [4:0] letter,
                    input done,
                    input confirm_letter,
                    input delete_letter,
                    output logic [11:0] pixel_out);

    integer i;

    logic [3:0] start_offset;                   // num of bits for 256*240 ROM
    logic [10:0] start_addr;
    logic [7:0] char_row;                       //ROM output as 8 bit wide array
    logic [5:0] char_counter;                   //tracks the current character being displayed
    logic [7:0] current_row;                    //current char_row being used for display
    logic [4:0] current_letter;                 //current letter being display
    logic [4:0] letter_array [63:0];            //array which holds 5 bit numbers corresponding to letters
    logic [2:0] row_counter;                    //tracks the position in the current character row
    logic [5:0] letter_array_input_index;       //index of the next letter to be input
    logic [10:0] text_rom_addr;                 //addresses into the text ROM
    logic started;

    letter_to_addr addr_finder (.letter(current_letter), .start_addr(start_addr));

    assign text_rom_addr = start_addr + start_offset;


    //instantiates ROM containing COE file for ASCII characters
    text_rom your_instance_name (
      .clka(pixel_clk_in),
      .addra(text_rom_addr),
      .douta(char_row)
    );

    always_ff @ (posedge pixel_clk_in) begin
        if (reset) begin
            start_offset <= 0;
            char_counter <= 0;
            current_row <= 0;
            row_counter <= 0;
            started <= 0;
            letter_array_input_index <= 0;

            //resets all letters to empty spaces
            for (i=0; i<64; i=i+1) begin
                letter_array[i] <= 5'b0;
            end
        end else begin
            if (vcount_in >= 240 && vcount_in < 255) begin              //when the
```

```
                currently displayed pixel line is within this module’s domain
                if (hcount_in < 512) begin                              //when within the range of the text (8x64=512)
                    if (char_counter == 63 && row_counter == 4) begin   //almost reached the end of the character line; occurs 3 cycles in advance
                        start_offset <= start_offset + 1;
                        char_counter <= 0;
                        current_letter <= letter_array[0];
                    end else if (row_counter == 4) begin                //occurs 3 cycles in advance to account for ROM read latency
                        current_letter <= letter_array[char_counter + 1];
                        char_counter <= char_counter + 1;
                    end
                    if (row_counter == 7) begin                         //actually finished displaying the current character
                        current_row <= char_row;
                    end else if (hcount_in != 0) current_row <= current_row << 1;   // shift in the next bit for the current character

                    pixel_out <= (current_row[7]) ? {12{1’b1}} : 12’b0;
                    row_counter <= (hcount_in==0) ? 0 : row_counter + 1;
                end else begin                                          //prepares for the next line in the frame
                    row_counter <= 0;
                    current_row <= char_row;
                end

            end else begin                          //out of range; prepares for next frame
                current_row <= char_row;
                current_letter <= letter_array[0];
                start_offset <= 0;
                pixel_out <= 12’b0;
            end
        end
        if (done && !delete_letter) begin           //when a gesture is confirmed, set the letter at the cursor to the recognized letter
            letter_array[letter_array_input_index] <= letter;
        end
        if (confirm_letter) begin                   //increment the cursor, leaving the current letter at the previous index
            letter_array_input_index <= letter_array_input_index + 1;
        end else if (delete_letter) begin           //decrement the cursor, deleting the letter at the current index
            letter_array[letter_array_input_index] <= 0;
            letter_array_input_index <= (letter_array_input_index == 0) ? 0 :
                letter_array_input_index - 1;
        end
    end
endmodule

module letter_to_addr (
                        input [4:0] letter,
                        output logic [10:0] start_addr
    );
    parameter START_OFFSET = 1024;      //offset to the start of the caps alphabet; start at ascii value = 65
    parameter SPACE_VALUE = 0;
```

```
    always_comb begin
        start_addr = (letter != SPACE_VALUE) ? (letter * 16) + START_OFFSET : 0;
    end

endmodule
```

Camera Control

```
/*
   Wire - I2C Scanner
   The WeMos D1 Mini I2C bus uses pins:
   D1 = SCL
   D2 = SDA
*/

#include <Wire.h>

/*These are settings some of which have been found empirically and/or found
from random internet sites. When you see that there's a "magic" number it
isn't a magic number like in comp sci or something...it just means we have
no idea why this register value seems to help since the data sheet doesn't
give a ton of guidance.  I'm sure there's rational explanations for many of
these numbers, but sometimes I've just got bills to pay and life to live
and don't have time to figure out why. You know the deal.
*/

const byte ADDR = 0x21; //name of the camera on I2C

uint8_t settings[][2] = {
  {0x12, 0x80}, //reset
  {0xFF, 0xF0}, //delay
  {0x12, 0x14}, // COM7,     set RGB color output (QVGA and test pattern 0x6...for RGB video 0x4)
  {0x11, 0x80}, // CLKRC     internal PLL matches input clock
  {0x0C, 0x00}, // COM3,     default settings
  {0x3E, 0x00}, // COM14,    no scaling, normal pclock
  {0x04, 0x00}, // COM1,     disable CCIR656
  {0x40, 0xd0}, //COM15,     RGB565, full output range
  {0x3a, 0x04}, //TSLB       set correct output data sequence (magic)
  {0x14, 0x18}, //COM9       MAX AGC value x4
  {0x4F, 0xB3}, //MTX1       all of these are magical matrix coefficients
  {0x50, 0xB3}, //MTX2
  {0x51, 0x00}, //MTX3
  {0x52, 0x3d}, //MTX4
  {0x53, 0xA7}, //MTX5
  {0x54, 0xE4}, //MTX6
  {0x58, 0x9E}, //MTXS
  {0x3D, 0xC0}, //COM13      sets gamma enable, does not preserve reserved bits, may be wrong?
  {0x17, 0x14}, //HSTART     start high 8 bits
  {0x18, 0x02}, //HSTOP      stop high 8 bits //these kill the odd colored line
  {0x32, 0x80}, //HREF       edge offset
  {0x19, 0x03}, //VSTART     start high 8 bits
  {0x1A, 0x7B}, //VSTOP      stop high 8 bits
  {0x03, 0x0A}, //VREF       vsync edge offset
  {0x0F, 0x41}, //COM6       reset timings
  {0x1E, 0x00}, //MVFP       disable mirror / flip //might have magic value of 03
  {0x33, 0x0B}, //CHLF       //magic value from the internet
  {0x3C, 0x78}, //COM12      no HREF when VSYNC low
  {0x69, 0x00}, //GFIX       fix gain control
  {0x74, 0x00}, //REG74      Digital gain control
  {0xB0, 0x84}, //RSVD       magic value from the internet *required* for good color
  {0xB1, 0x0c}, //ABLC1
  {0xB2, 0x0e}, //RSVD       more magic internet values
```

```
  {0xB3, 0x80}, //THL_ST
  //begin mystery scaling numbers. Thanks, internet!
  {0x70, 0x3a},
  {0x71, 0x35},
  {0x72, 0x11},
  {0x73, 0xf0},
  {0xa2, 0x02},
  //gamma curve values
  {0x7a, 0x20},
  {0x7b, 0x10},
  {0x7c, 0x1e},
  {0x7d, 0x35},
  {0x7e, 0x5a},
  {0x7f, 0x69},
  {0x80, 0x76},
  {0x81, 0x80},
  {0x82, 0x88},
  {0x83, 0x8f},
  {0x84, 0x96},
  {0x85, 0xa3},
  {0x86, 0xaf},
  {0x87, 0xc4},
  {0x88, 0xd7},
  {0x89, 0xe8},
  //WB Stuff (new stuff!!!!)
  {0x00, 0x00}, //set gain reg to 0 for AGC
  {0x01, 0x8F}, //blue gain (default 80)
  //{0x02, 0x8F}, //reg gain (default 80)
  {0x02, 0x80}, //reg gain (default 80)
  {0x6a, 0x3F}, //green gain (default not sure!)
  //{0x6a, 0x4F}, //green gain (default not sure!)
  {0x13, 0x00}, //disable all automatic features!! (including automatic white balance)
  {0x10,0x20}, //exposure (default 40)
  //added value
  {0x55, 0x44}, //increases brightness
  {0xA0, 0xFF}, //increases brightness
  {0x9F, 0xFF}, //increases brightness
  //{0x55, 0x2F}, //increases brightness
  //{0xA0, 0xFF}, //increases brightness
  //{0x9F, 0xFF}, //increases brightness
};
uint8_t output_state;

void setup()
{
  Wire.begin();
  Serial.begin(115200);
  Serial.println("Starting");
  delay(1000);
  Wire.beginTransmission(ADDR);
  Wire.write(0x0A);
  Wire.requestFrom(ADDR, 2);
  byte LSB = Wire.read();
  byte MSB = Wire.read();
  uint16_t val = ((MSB << 8) | LSB);
  Wire.endTransmission();
  Serial.println(val);
  for (int i = 0; i < sizeof(settings) / 2; i++) {
    Wire.beginTransmission(ADDR);
```

```
    Wire.write(settings[i][0]);
    Wire.write(settings[i][1]);
    //    Wire.write(RegValues[i][1]);
    //    Wire.write(RegValues[i][2]);
    Wire.endTransmission();
  }
  //  Wire.write(0x12);
  //  Wire.write(0x4);
  Serial.println("OV7670 Setup Done");
  pinMode(4, INPUT_PULLUP);
  output_state = 0;
}


void loop()
{

}


void writeByte(uint8_t target_reg, uint8_t val) {
  Wire.beginTransmission(ADDR);
  Wire.write(target_reg);
  Wire.write(val);
  Wire.endTransmission();
}

void readBytes(uint8_t target_reg, uint8_t* val_out, uint8_t num_bytes) {
  Wire.beginTransmission(ADDR);
  Wire.write(target_reg);
  Wire.requestFrom(ADDR, num_bytes);
  uint8_t* ptr_to_out;
  ptr_to_out = val_out;
  for (int i = 0; i < num_bytes; i++) {
    *ptr_to_out = Wire.read();
    ptr_to_out++;
  }
}
```