FPGA iPad

Alex Leffell and Sienna Ramos

Introduction

Smart devices are ubiquitous in the modern world. One of the most significant features of a smartphone or an iPad is the touchscreen, which can sense the position of the user's finger, and can use that information to send commands to the phone. To send a message, we tap our fingers on the keys of a digital keyboard. To draw a picture, we can drag our fingers across the screen using a digital pen. And if we want to look more closely at a picture in our camera roll, we place both fingers on the picture, and pull them apart, causing the image to expand.

This project recreates the experience of playing with an image on a screen. We wanted to create a program in Verilog to process the scaling of an image. This task involves researching image processing algorithms, and creating verilog to implement the algorithms in a manner that was both quick and space-efficient.

Our project differs from a normal iPad functionality in that we opted to locate the fingers using a camera instead of with capacitive touch. The fingers are outfitted with infrared LEDs for better detection by the camera. The gesture recognition task allows the team to work with NTSC protocol and to develop a method by which to detect blobs of light using limited memory, time, and processing power.

Overview

Hardware

The FPGA iPad uses the labkit and the NTSC camera from the lab. We use two IR LEDs, each in series with a 51 Ω resistor and powered by 5V from the labkit, as shown in Figure 1.

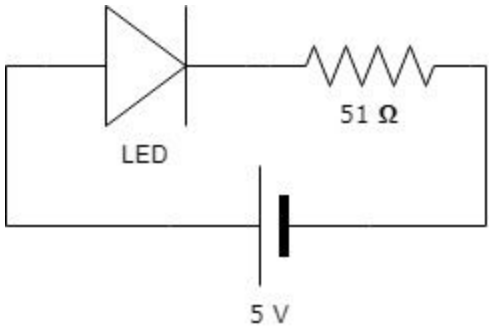

Figure 1: IR LED circuit

Each circuit is soldered and wrapped with heat-shrinking material to avoid short circuiting. We use two velcro straps to attach the two LEDs to the fingertips of the index and middle fingers.

The lens of the NTSC camera is covered by a filter made from a floppy disk. The floppy disk filters for IR light so as to reduce background noise created from other sources of light.

The FPGA iPad uses the labkit and the VGA screen provided at each lab station.

Software

We have designed two main modules for our system. The first is the gesture recognition module, which tracks the data stream coming from the NTSC to locate the two blobs of light from the IR LEDs. Once the two light sources have been detected, we perform calculations on the x and y coordinates of the points to determine the distance between the points, the angle between them, and the location of the center of the points.

The second is the image processing module, which takes an image stored in flash memory and the distance data from the gesture recognition module to scale the image to 50%-200% of its original size. The module uses a polyphasic finite impulse response filter to process the image so that the scaled versions appear smooth and avoid noise or aliasing.

The scaled image from the image processing module and the location of the center of the points from the gesture recognition module is sent to the VGA driver and displayed on the VGA screen.

A block diagram of the general software architecture is shown in Figure 2.

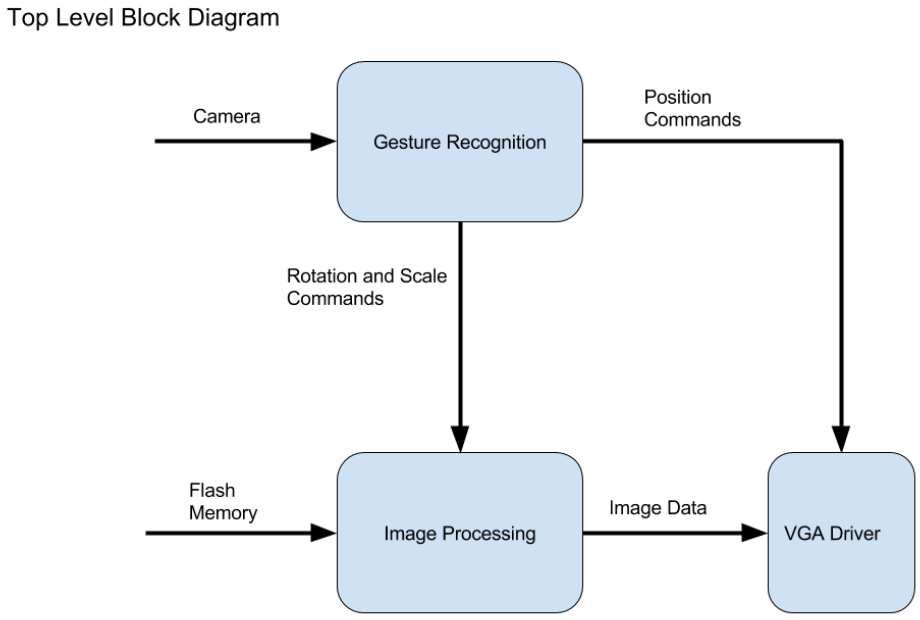

Figure 2: Top Level Block diagram

Technical Specifications

Gesture Recognition

All of the gesture recognition modules were developed by Sienna.

A block diagram of the gesture recognition architecture is shown in Figure 3.

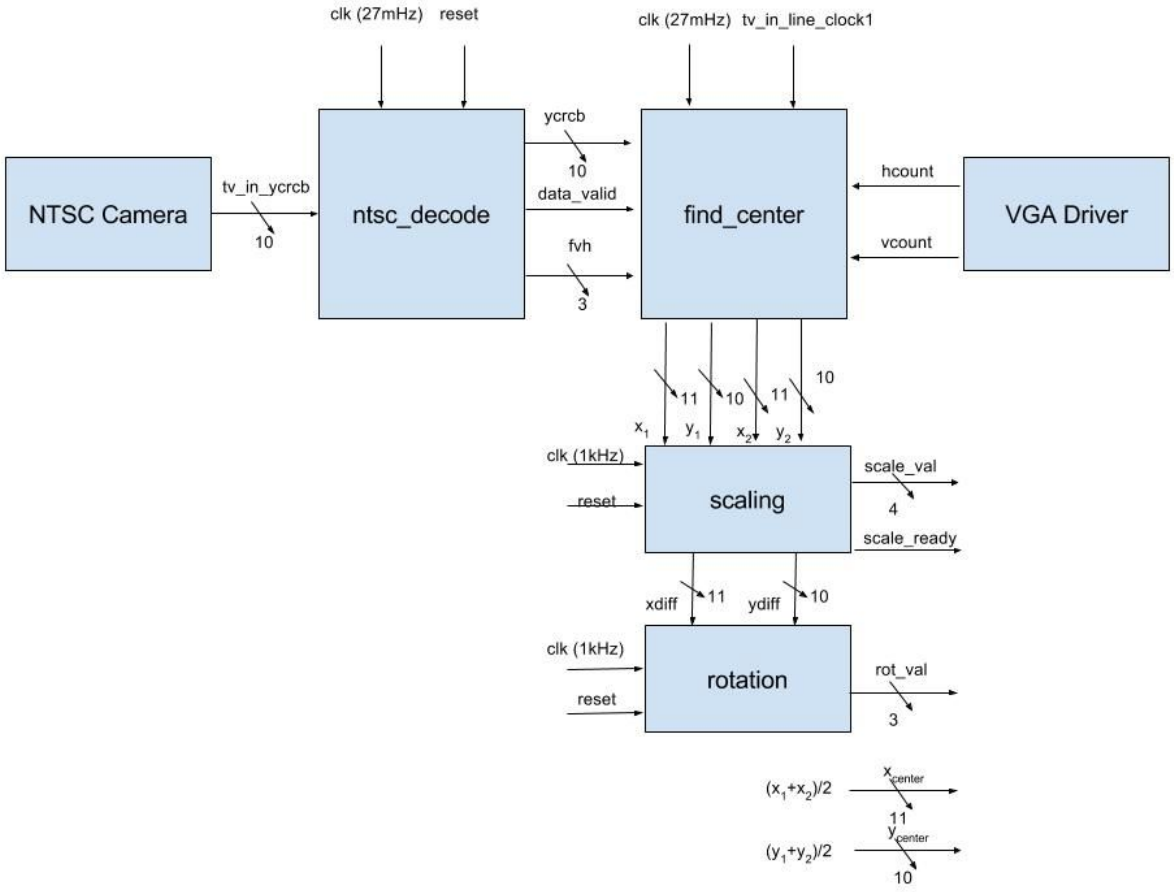

Figure 3: Gesture Recognition Block Diagram

NTSC_Decode

The ntsc_decode module consists of a state machine which watches the data stream coming from the NTSC camera, determines which values the stream is currently outputting, and updates its internal values accordingly. This module is largely unchanged from the sample NTSC code available on the course website, so I will not go into detail about how the code works, apart from the functionality that was changed for use in this project.

tv_in_ycrcb is the 10 bit input from the decoder chip. Depending on the state of the stream, it either contains luminance data, Y, or Chrominance data, Cr and Cb. ntsc_decode outputs the 30 bit value ycrcb, which contains the luminance and chrominance data for each pixel. For the purposes of this project, we only care about the luminance data because we want to track the fingertip LEDs, so although all YCrCb data is saved, we only change the way Y is calculated.

We set a threshold value of 10'h200. All tv_in_ycrcb values above this threshold are saved as 10'h3FF. All tv_in_ycrbc values below this threshold are saved as 10'h000. With this software filter and the hardware filter placed over the camera lens, the only nonzero pixels are the ones which correspond to the IR LEDs on the fingertips.

Find_Center

The first always block in the find_center module uses the 3 bit fvh (frame, vertical, and horizontal) and 1 bit dv (data_valid) inputs coming from the NTSC decoder, as well as the tv_in_line_clock1 created by the NTSC camera. This block is the same as the first part of the ntsc2zbt module provided in the sample NTSC code available on the course website, so I will not go into detail about how it works. This block of the code functions to assign data from ntsc_decode a row and column value as it corresponds to a pixel on a screen. From this code, we have an x and y coordinate of every pixel coming in.

Once we have the coordinate values of each pixel, we enter a finite state machine which scans the incoming data for the two contiguous masses of nonzero pixels. The machine is clocked on tv_in_line_clock1. This state machine has 5 states: LOOKING_FOR_BLOB1, BLOB1, LOOKING_FOR_BLOB2, BLOB2, and NOT_LOOKING.

The default state is LOOKING_FOR_BLOB1. In order to leave this state, an incoming pixel must meet the following criteria: the pixel must be white (&din==1), and it must be within a certain range of coordinates for which the camera is giving valid data. In this case, we use a range of 40<x<640. If this criteria is met, we proceed to the next state, BLOB1.

In BLOB1, we record the x and y values of the current pixel. This is the top left corner of the first light. We exist in BLOB1 for only one clock cycle.

We move on to LOOKING_FOR_BLOB2. The criteria for leaving this state is the same as for LOOKING_FOR_BLOB2, with the additional requirement that the next pixel must not be within 30 pixels in the x or y direction from the first pixel recorded. Once triggered, we move to BLOB2. In the chance that the VGA screen starts a new cycle of printing pixels onto the screen before we have recorded the second blob, we move back to LOOKING_FOR_BLOB1.

In BLOB2, we record the x and y values of the current pixel. This is the top left corner of the first light. We exist in BLOB2 for only one clock cycle.

Until the VGA screen starts a new cycle, we end in the NOT_LOOKING state.

At the beginning of the new VGA cycle, we push the x and y coordinates for each of the two blobs into a 16-entry first in, first out array. The average value of this array is the output 11 and 10 bit x and y coordinates of the blobs. Because the two blobs sometimes switch places depending on which one is higher in the camera's view, the first in, first out arrays are reset so that all entries are equal to the most recent value of the x and y coordinates if one of the coordinates is 20 pixels or farther from its previously measured location.

Scaling

The scaling module is a finite state machine with four states. The state is changed every millisecond, and the output value is changed every four milliseconds.

In the first state, the scaling module uses the x and y coordinates of the two blobs found in the find_center module. For both the x and y coordinates, the module compares the values to see which is larger, and then subtracts the larger from the smaller to create the 11 bit output xdiff and 10 bit output ydiff.

In the second state, the sum of xdiff and ydiff is calculated.The sum of xdiff and ydiff serves as an approximation of the distance between the points of light. The sum is put into a first in, first out array which holds the most recent 16 sum values.

In the third state, the average value of the first in, first out array is calculated. This value serves as the distance value upon which all further calculation are based.

In the fourth state, the distance value is compared with the reference distance value. Based on the results of the comparison, the 4 bit output scale_val is assigned a value between 0 and 11. The value is assigned based on the following comparisons:

scale_val=0 if distance is 0-25% of the reference value
scale_val=1 if distance is 25-50% of the reference value
scale_val=2 if distance is 50-75% of the reference value
scale_val=3 if distance is 75-100% of the reference value
scale_val=4 if distance is 100-125% of the reference value
scale_val=5 if distance is 125-150% of the reference value
scale_val=6 if distance is 150-175% of the reference value
scale_val=7 if distance is 175-200% of the reference value
scale_val=8 if distance is 200-225% of the reference value
scale_val=9 if distance is 225-250% of the reference value
scale_val=10 if distance is 250-275% of the reference value
scale_val=11 if distance is 275-400% of the reference value

Upon reset, the reference distance value is set to the current distance value.

Rotation

The rotation module uses the distances between the two x coordinates and between the two y coordinates, xdiff and ydiff respectively, to make a calculation of the angle between the fingers and the horizontal axis. We compare the ratio between xdiff and the sum of xdiff and ydiff to determine the angle. We bin the data into the following categories:

0 is an angle from the horizontal axis between 0°-11.25°
1 is an angle from the horizontal axis between 11.25°-22.5°
2 is an angle from the horizontal axis between 22.5°-33.75°
3 is an angle from the horizontal axis between 33.75°-45°
4 is an angle from the horizontal axis between 45°-56.25°
5 is an angle from the horizontal axis between 56.25°-67.5°
6 is an angle from the horizontal axis between 67.5°-78.75°
7 is an angle from the horizontal axis between 78.75°-90°

This raw 3 bit rotation value is updated every millisecond and put into a 16-entry moving average filter. The output of the moving average filter is the 3 bit rotation value output of the module.

Translation

Translation is not a separate module and the calculation exists in the top file. The raw x_{center} and y_{center} values are two wires, 11 and 10 bits respectively, which hold the averages of the two x values and the two y values. The raw data is put through a moving average filter which averages the 32 most recent values for each wire. The filter is updated every millisecond. The output values of the moving average filter are the top left corner of the processed image on the VGA screen.

Image Processing

All of the image processing modules were developed by Alex.

The image processing module translates and scales a given image. In order to be able to keep the scaled image in BRAM, the original image size was restricted to 128 pixels square. To ensure that the original image is bandlimited, it was prefiltered in MATLAB. The image was uploaded to the FPGA by a .coe file and the BRAM wizard. The image scaling was performed by sequentially upsampling and downsampling. The ratio of upsampling to downsampling determines the amount of scaling, allowing for non-integer amounts of scaling. A polyphase filter structure was utilized to perform this scaling efficiently.

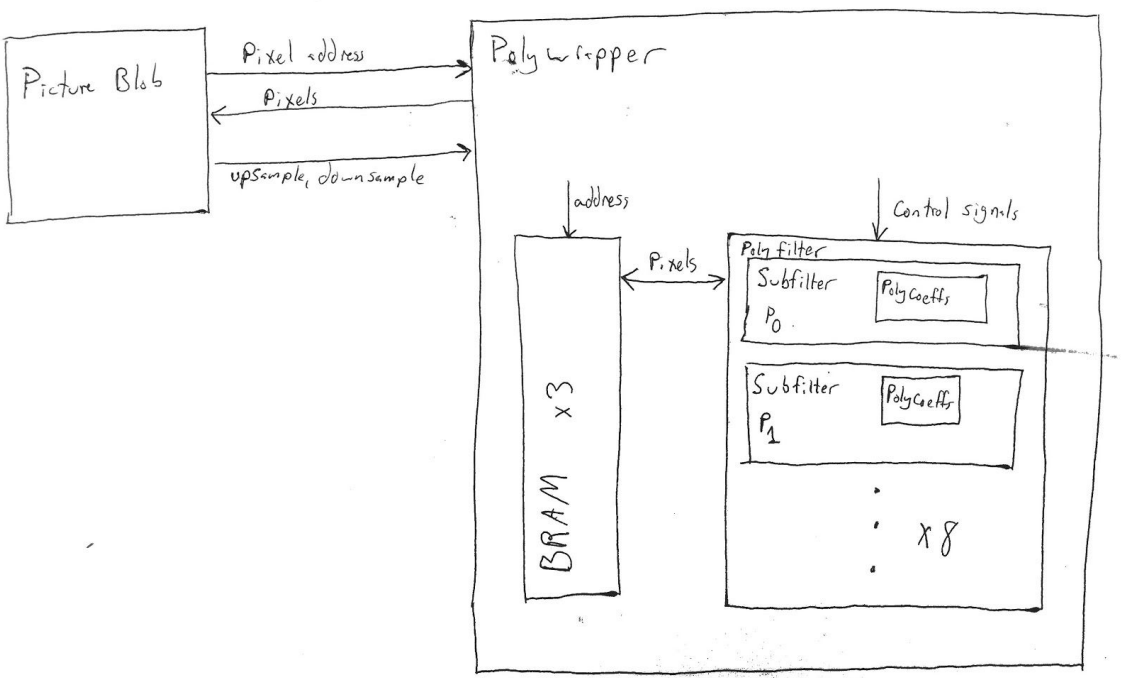

Figure 4: Block diagram of image processing module

The core of this module is a polyphase FIR filter. This is a filter architecture that performs the same function as a regular FIR filter except it is more efficient since it performs upsampling without actually having to insert zeros between samples. A polyphase filter for upsampling works by dividing a regular FIR filter in to subfilters that work in parallel. These subfilters are referred to as phases. In order to have 8 scaling steps between the original image and one twice as big, a 31-tap FIR filter was divided into 8 phases. The filter coefficients were determined using the MATLAB intfilt() command, which designs a filter specifically for interpolation. The data inputted to the polyphase filter is sent to each subfilter and then the output of each subfilter is commutated as shown in Figure 5. For example, to scale a signal by 3, 3 subfilters would be used. Each subfilter would get every third filter coefficient of the original 31-tap filter, offset by the phase (see Figure 5 for a clearer explanation). Then a data sample would be put into all the subfilters simultaneously and the output of each filter would be collected sequentially to get the upsampled data.

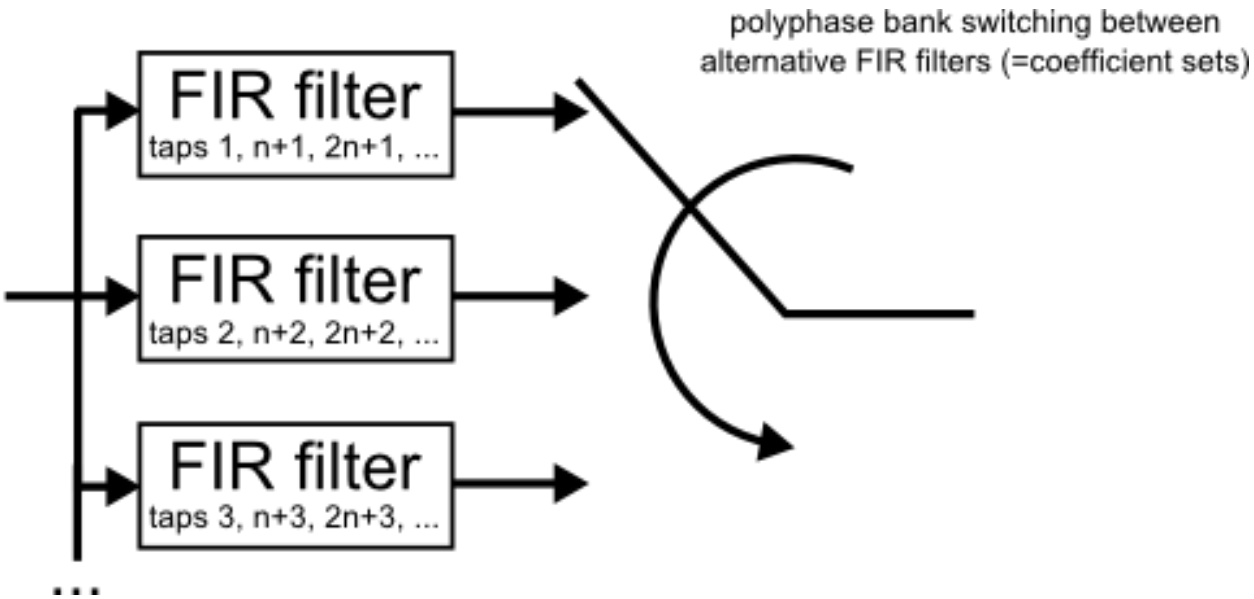

Figure 5: Diagram showing polyphase filter architecture. dsprelated.com

Poly_coeffs

This module returns a coefficient of the 31-tap FIR filter. Each subfilter contains an instance of this module. It takes as an input the upsampling factor *upfac*, the phase of its parent subfilter *phase* and the convolution index of that subfilter, *index*. It computes the following eqaution to determine which coefficient to output: coeff# = *index* * (*upfac* + 1) + *phase*.

Subfilter

This module implements an FIR filter. It is parameterized by its phase, making it easy to create multiple instances with different phases. This module is based around an FSM and the data to be filtered is stored in a shift register. It waits in an idle state until the ready input is toggled. Since the length of the convolutional window depends on the amount of upsampling, this window is first stored in a register *max*. In the next state, the input data is shifted into the shift register. Then, for a number of samples determined by *max*, the data in the shift register is convolved with the coefficients from poly_coeffs. This innovative design allows the subfilter to perform different amounts of scaling without having to recompile the verilog. Once finished, a done signal is strobed and the convolution accumulator is scaled and outputted. This module also has a reset strobe that clears the shift register and returns the FSM to the idle state.

Polyfilter

This module implements the complete polyphase FIR filter with 8 phases (8 instantiations of the subfilter module). It has output strobes that indicate when it is ready for a new sample and when new data is available at the output. This module transfers the input data

to all subfilters simultaneously. Then when the subfilters all have new data available, this high level module sequentially transfers the data from each active subfilter output to the output of this module. It only takes the data from the subfilters being used for the current amount of upsampling. This action performs the function of the upsampling commutator described above.

Polywrapper

This module implements an FSM that controls the BRAM and the polyphase filter, inputting the data into the filter in the correct order and putting the filtered data in the correct location. This module takes in as inputs up and downsample values, the pixel address and image width and start and reset strobes. The module instantiates 3 BRAMs, one that contains the original image, one that contains the image after being scaled horizontally, and one that contains the final bidirectionally scaled image. The FSM waits in an idle state, outputting pixels from the unmodified image based on the pixel address input. Then once the start signal is strobed, it begins to scale the image based on the up and downsample values. First the FSM puts data into the polyphase filter row-wise, resetting the filter after each row. The FSM keeps track of when to reset the filter based on the image size. Then the FSM puts data in to the filter column-wise, resetting after each column. The module calculates the limit for each column based on the image size and scale factor. In both steps, decimation is accomplished by ignoring filter output values according to the downsample value. It also scales the filter output according to the upsample value to counteract the passband gain of the filter. Finally the FSM waits in a done state where it outputs pixels from the modified image until reset is strobed.

Picture blob

This module is based on the blob module from lab 3. This module transferred up and downsample values to the Polywrapper module based on commands from the gesture recognition module and also calculated the correct pixel address for the current hcount and vcount. It adjusted the image dimensions based on the scale command. This module also translated the image by offsetting hcount and vcount like in the original blob module.

Challenges

One challenge we predicted during our presentation was that we needed to figure out a way to build both the image processing and gesture recognition modules without using too much memory. One way to mitigate this potential issue was to develop a gesture recognition module that used only a finite state machine and a few other calculation variables, as opposed to saving a pixel matrix in the ZBT and performing calculations on the matrix. The resulting state machine, although fast and barely using the memory, was extremely noisy and buggy. By the time we started integrating the project and realized we had a significant amount of unused memory, it was too late to develop an entirely new architecture to detect the IR lights in a more robust, more memory intensive way.

We had a lot of issues with the clocking. There were multiple clocks running for each module, and incorrect clocking gave us nonsensical data. We fixed many of our bugs by thinking through the requisite clocking for each module and incorporating ready strobes that allowed interfacing between different clock domains.

In the image processing module, it was difficult to keep track of the memory locations of the pixels, and to keep track of the new locations after scaling. Designing the architecture for the polyphase filter was also difficult, and required a lot of time spent prototyping the mechanism in MATLAB before attempting to build the filter in Verilog.

Takeaways and Advice

Alex's Notes

I really enjoyed this project since I got to implement a somewhat abstract signal processing concept in hardware. This involved researching polyphase filter architectures and really understanding them in order to design one in verilog. I found prototyping the filtering algorithms in MATLAB to be very helpful and I recommend doing that to future teams. I also felt that I jumped in to writing verilog too quickly after I figured out polyphase filters, leading to a high level module that was clunky and could have been much neater. I would recommend making sure you have a complete block diagram with exact inputs and outputs before writing any verilog code. I also spent a lot of time working on different 'test' versions of the filter. These ended up not working and I spent a lot of time trying to debug something that I wasn't even going to use in the end. Thus I recommend constant self evaluation to understand the work you are doing on any given day in the context of the whole project. This will prevent getting too sidetracked. While I was successful in scaling an image, if I had more time I would like to neaten up my code base to make it as neat as possible. Then it would be easier to debug some of the minute image artifacts that were present in my scaling implementation. Then it would not be too difficult to incorporate image rotation into the processing pipeline.

Sienna's Notes

I learned so much from this project. I had no idea what an NTSC camera was before this project, and now I'm very comfortable working with the protocol. I had a lot of fun developing code to accomplish a task while also working within constraints we didn't have during the lab assignments, such as using as little memory as possible, producing a result in near-real time, and making sure my code was usable and understandable for my partner. This project taught me how much I have yet to learn about optimizing memory, about using clocks correctly, and about interesting image processing algorithms and their applications in electrical engineering. If I'd had more time, I would be interested in developing a more rigorous method of blob detection. Although this task would have required more memory and computation time, it would produce far less noise than the method I used.