SpaceSynth

Prajwal Tumkur Mahesh
Sage Simhon
11 December 2020

Table of Contents

Overview

SpaceSynth is a musical instrument controlled by movement. Through 3D tracking of the players hands and head, our gesturally controlled synthesizer allows its users to physically interact with their music, manipulate, and explore new sounds in their space. By extracting a number of features with the help of a camera, such as horizontal and vertical positions of both hands, their proximity to each other and to the device, as well as the spatial location of the player's head, SpaceSynth gives users the control and room for expression they need for creating and interacting with their audio in real time. The audio generation component, based around two subtractive synthesizer engines is designed for simplicity as well as customizability when needed. Adjustments to the frequency, amplitude, tuning, filter cutoff, and waveshapes of both synthesizers are possible either through the gesture-based controls or physical switches on the FPGA device, and allow for full control over the timbre and sound of the instrument. An LFO can optionally be enabled with its own configurable frequency, amplitude, and waveshape, to provide even greater variety and creative control of the sound.

Block Diagram

Figure 1 shows a high level overview of the system. The **camera_to_mask** module, shown with greater detail in figure 2, receives images from the camera and extracts data about the spatial positioning of the user's limbs. The **synthesizer** module, shown in figure 3, is one of the main building blocks of our audio system and accepts various audio parameters to generate and manipulate waveforms. The **top_level** module bridges the gap between these two, defining how the spatial data affects the various parameters of the synthesizers and adding extra features like an LFO and a VGA display to enhance the user experience. In the following sections, we explain how each of these modules work in greater detail.

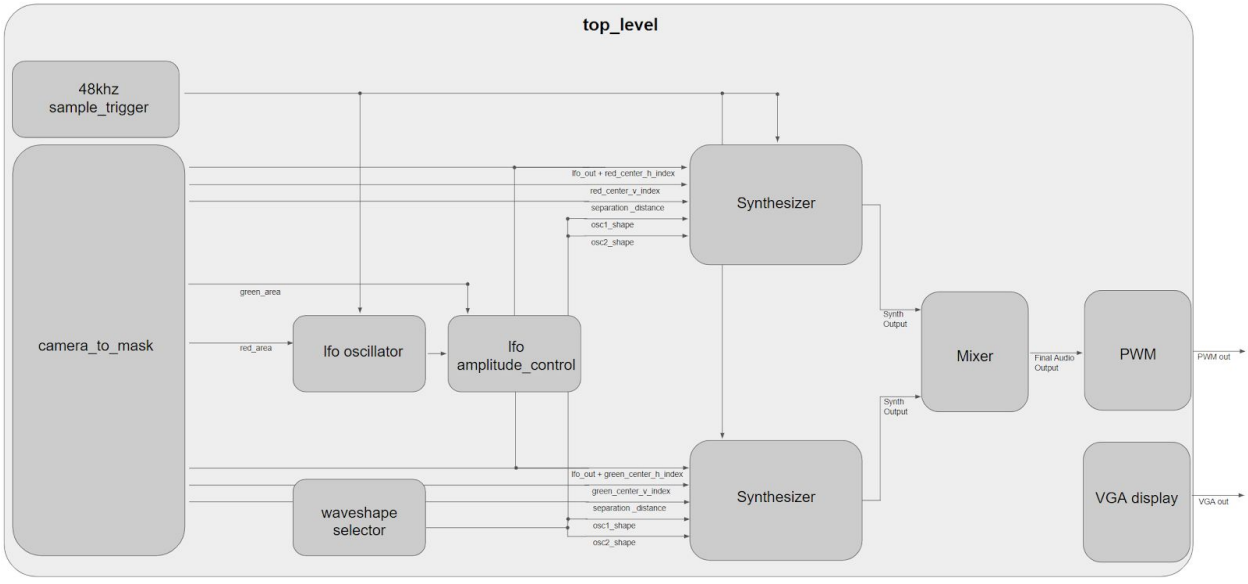

Figure 1: High-level block diagram of most important components of the top_level module

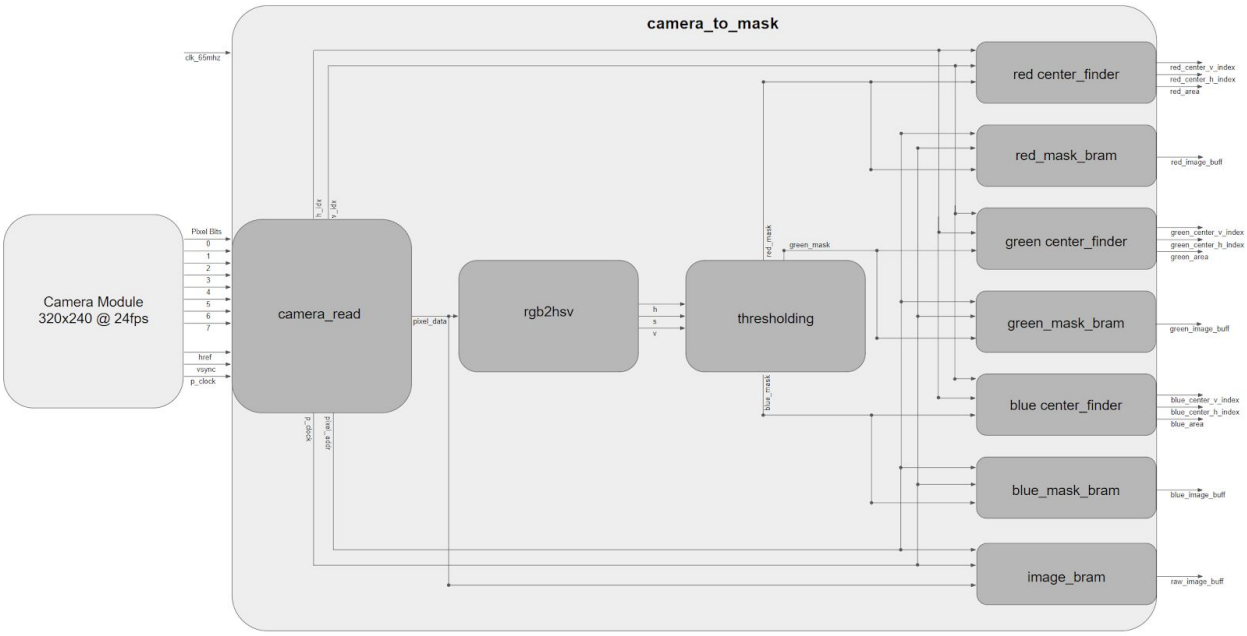

Figure 2: High-level block diagram describing the overall operation of the camera_to_mask module

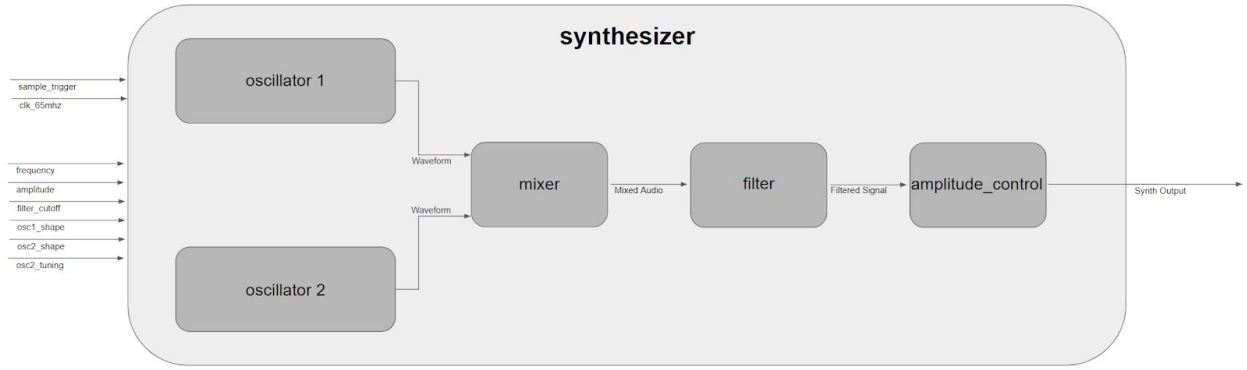

Figure 3: High-level block diagram describing the audio path through the synthesizer

Audio System

Oscillator (Prajwal)

The oscillator module is responsible for creating a signed 16-bit waveform with a configurable frequency and waveshape. The waveforms are generated with the help of a 32-bit phasor, which is simply a 32-bit counter that increments by a certain amount at the 48kHz sampling rate, resulting in a sawtooth waveform of a fixed amplitude as the counter repeatedly fills up at a constant rate and overflows. This sawtooth shape of the phasor can be seen in Figures 4 through 7 below. As the name suggests, this phasor generates the phase information needed to create the output waveforms. Because we want to be able to control the frequency of the output waveforms, we need to be able to control how quickly it completes one period, that is, how quickly the phase of that waveform goes from 0 to 2**π**. Since the phasor is responsible for providing the phase information for the output waveform, this means that we need to update the the phasor counter such that it goes from 0 (corresponding to a phase of 0) to $2x10^{32}$ (corresponding to a phase of 2**π**) at the desired frequency of the output waveform. The amount by which the counter is incremented during each sample is given by the equation below.

$$phase\ step\ = (desired\ frequency)\ \times\ \frac{2\times10^{32}}{48kHz}$$

The frequency input to the oscillator module is defined by a 12-bit number, so the maximum frequency is limited to 4095Hz. With the help of the phasor, we can create four fundamental waveshape types: sine, square, triangle, and sawtooth.

The sawtooth waveform is the simplest to generate, since the phasor is already a sawtooth wave. If a sawtooth output is desired, we can simply take the top 16 bits of the phasor, which is unsigned, and invert the MSB to create the signed output waveform. Figure 4 shows the phasor waveform on top and the output sawtooth on the bottom.

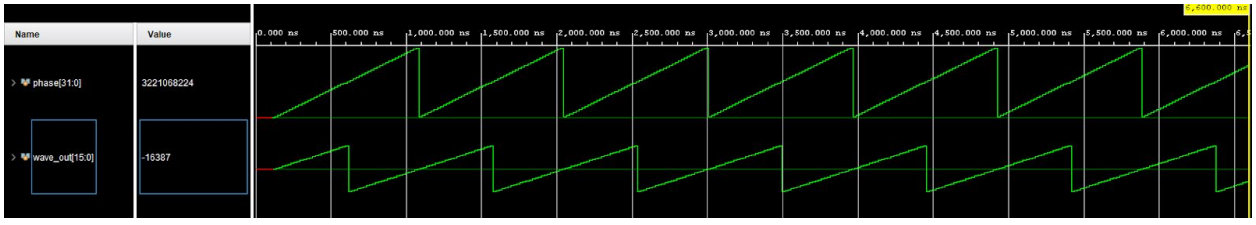

Figure 4: Phasor (top) and output sawtooth waveform (bottom) in simulation

The triangle waveform can be thought of as back to back sawtooth waveforms with half the period. In other words, in the time the phasor needs to ramp up from zero to its maximum value, the triangle needs to ramp up from zero to its maximum value and ramp back down from its maximum value to zero. To accomplish this, we take note that the MSB of the phasor, i.e. phasor[31], can be used as an indicator for the halfway point of the phase, since it has a value of 0 for half the time and a value of 1 for the other half. While this bit is 0, we simply take

phasor[30:15] which produces an upward sloping ramp with twice the frequency of the phasor. Then when phasor [31] is 1, we switch the direction of that ramp by using 16'hFFFF - phase[30:15], which produces a downward sloping ramp with twice the frequency of the phasor. Finally, the MSB is flipped to make it a signed waveform before being sent out as a triangle wave. The output waveform is shown in Figure 5.

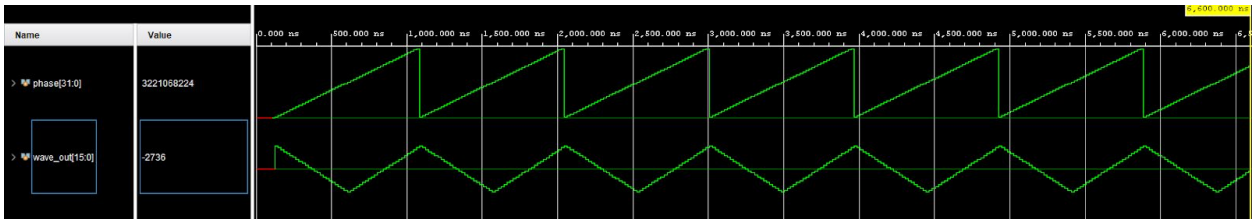

Figure 5: Phasor (top) and output triangle waveform (bottom) in simulation

The square wave, shown in Figure 6, is created in a similar way, except instead of using a ramp for each half of the phase, we just use a single high or low value. While phasor[31] is 0, we output a high value of 16'hFFFF, and when it is 1, we output a low value of 0. Once again, the MSB of this output is flipped to convert it to a signed value.

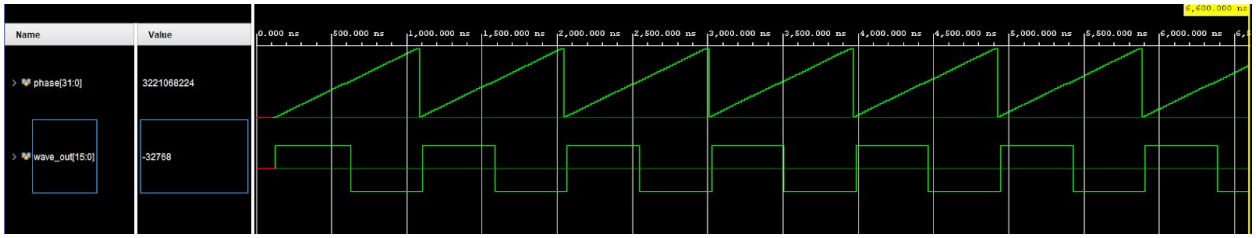

Figure 6: Phasor (top) and output square waveform (bottom) in simulation

The sine waveshape is slightly more complex to generate. Instead of calculating the value of sine(phase) for each sample, we instead use an 8-bit look-up-table (LUT) which stores the amplitude information for one period of a sine wave. The sine LUT, the code for which is generated by a python script, holds the 16-bit unsigned amplitudes of a sine waveform at 256 different points during its period. By passing the upper 8 bits of the phasor, phase[31:24], into this module, we can cycle through these 256 points and get on the output a sine waveform with the same frequency as the phasor. Similar to the rest, the MSB is flipped before being sent out as a signed 16-bit waveform, as shown in Figure 7.

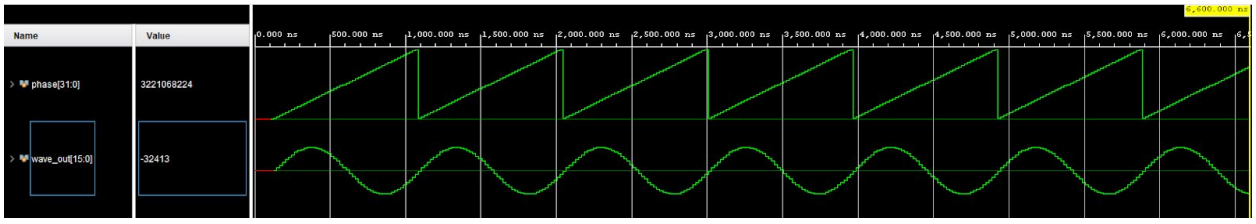

Figure 7: Phasor (top) and output square waveform (bottom) in simulation

Mixer (Prajwal)

The mixer module is used to combine two signed 16-bit waveforms into one without clipping. This is a combinational module which simply right-shifts both inputs to halve their gains and adds them together to form the output.

```
assign mixed_out = (wave1_in>>>1)+(wave2_in>>>1);
```

The mixer module is used to combine the output of the two oscillators inside each synthesizer module, as well as to combine the output of both synthesizers in top_level before they are sent out as the main audio.

Filter (Prajwal)

After the oscillators, the second most important component of a subtractive synthesizer is the low pass filter, which attenuates, or "subtracts", frequency components of the oscillator waveforms to shape the sound of the synth. For ease of implementation and because we don't need a perfect passband and stopband for this application, we chose to use a single first-order IIR filter to accomplish this. A first-order IIR filter is a simple recursive filter with the difference equation shown below.

$$y[n] = a_1 y[n-1] + b_0 x[n] + b_1 x[n-1]$$

The output of the filter is determined by the current input, the previous input, the previous output, as well as the coefficients a_1, b_0, and b_1. These coefficients are what determine the behavior of the filter, and most importantly, its cutoff frequency. Rather than calculating these coefficients on the FPGA for any given cutoff frequency, a python script was used to generate another 8-bit look up table with calculated coefficients corresponding to a range of cutoff frequencies.

The python script works by first choosing the range of cutoff frequencies for the filter. For the filter ultimately used in the project, this range was from 100Hz to 5000Kz. This range is then divided into 256 frequency steps which serve as the indices of the lookup table. For each frequency step, the ratio of the desired cutoff frequency to the system nyquist frequency (24kHz) is calculated and plugged into scipy's iir filter function, which outputs the values of the three coefficients for a lowpass iir filter with that desired cutoff.

These values are then multiplied by 2^{14} to convert them to fixed point numbers and formatted into a case statement and put into the filter_coeffs module, which delivers the appropriate filter coefficients for a chosen cutoff frequency among the 256 calculated. When the filter is updated by the user, the main filter module updates its a0 term, b0 term, and b1 term with the help of this filter_coeffs module.

To implement the difference function above, the main filter module needs to perform a number of multiplication and addition operations with fairly large numbers. To use resources most efficiently and to ensure that all the operations meet timing requirements, we employ an FSM to perform the steps of the difference equation sequentially rather than in parallel. Figure 8 summarizes the steps of this process along with the state transitions. First, a single, combinational multiplier is set up with two 16-bit factors—mult1 and mult2—with a 32-bit output mult_out, where mult_out = mult1*mult2. When a sample from a waveform enters the filter module, mult1 is set to b0, mult2 is set to the value of the input waveform, and the FSM enters its first state. The combinational multiplication is performed over the next clock cycle and as the FSM enters the new state, the result, left-shifted to compensate for the 2^{14} multiplication from earlier, is added to a running sum. Mult1 is then assigned to hold the next term b1, mult 2 is assigned to to hold the previous value of the waveform, and the FSM state is updated again. This process repeats until the difference equation has been calculated including all three terms, whereupon the sum now contains the output of the filter. The lower 15 bits of this sum along with the top sign bit are passed to the output. Because the filter module runs at a 65Mhz clock compared to the much slower sampling rate of 48kHz, there is plenty of time for calculating the filter output via this sequential method before the next sample comes in.

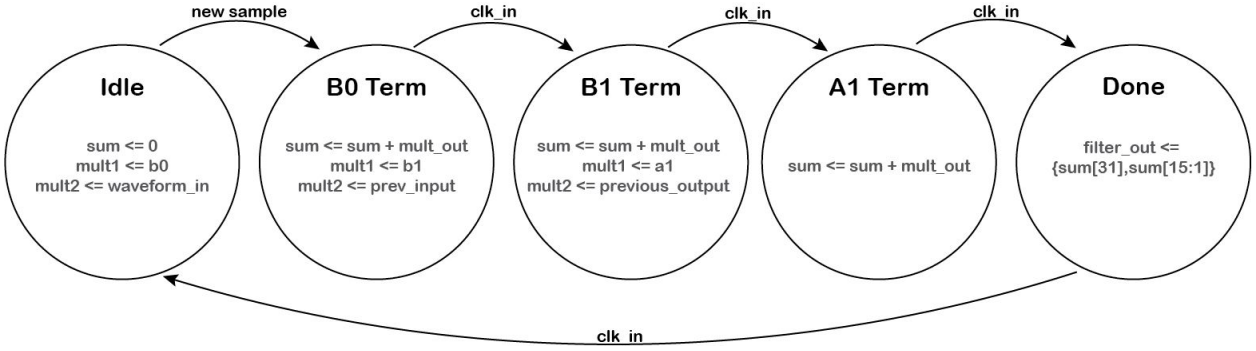

Figure 8: State transition diagram for first order IIR filter calculation

The size of the fixed-point normalized coefficients was chosen through a bit of trial and error. For most of the cutoff frequency steps, the IIR filter coefficients provided by the scipy iirfilter function were extremely small, so they needed to be multiplied by a fairly large number to scale them to fixed point numbers with enough precision. A larger scaling factor would give more usable digits and thus a higher precision representation of the filter coefficients, while a lower scaling factor would give fewer digits and less precision. However, going too large would mean more cascaded adders on the FPGA when it came time to multiply the terms and more difficulty getting the multiplications and additions to fit into a single clock cycle. A 2^{14} scaling was eventually chosen as the scaling factor by first starting at 2^{30} and lowering a couple of times until the implemented design met timing constraints.

Amplitude Control (Prajwal)

After the signal has been processed by the filter and before it leaves the synthesizer output, the gain of the signal is adjusted by the amplitude_control module. This is another combinational module that works by right-shifting the signal to reduce its amplitude. Since we are working with 16-bit signals, we can technically shift the signal by up to 16 bits, for 16 different volume levels with a -6dB drop in loudness between each step. Practically however, the lowest three or four volume levels are unusable, since at this point, there will not be enough bits of information remaining to accurately represent the audio waveform without adding in a lot of quantization noise. As an extreme example, shifting the signal by 15 bits to its lowest volume level will leave just 1 bit of signal information remaining, turning any input waveform into basically a square wave.

Synthesizer (Prajwal)

The synthesizer module is a higher-level module that collects all of the above components and defines the path the signal takes through them. The main audio generating components inside the synthesizer module are two oscillator modules. The frequency input to the synthesizer directly sets the frequency of one of these oscillators, referred to as osc_1. The frequency input to the other oscillator (osc_2) is instead tuned to a number of octaves above or below that of osc_1. Conveniently octaves are logarithmic units, where an increase of one octave corresponds to a doubling of the frequency. As a result, the frequency of osc_2 can be calculated simply by shifting the input frequency left or right by the number of octaves we want to be above or below it.

$$\text{osc2\_frequency} = \text{frequency\_in} << \text{octave}$$

The output of both oscillators is combined using the mixer module, the output of which is directed through the filter, followed by the amplitude controller, and finally sent to the output.

The waveshapes of both oscillators can be individually controlled, and when mixed together create an audio waveform that is rich in harmonics. By changing the filter cutoff frequency we can change how much of these harmonics are attenuated and consequently alter the timbre off the sound.

LFO (Prajwal)

An LFO, or low frequency oscillator is a separate oscillator whose frequency is typically below the audible frequency range, i.e. 20Hz or less. The purpose of an LFO is not to generate audio,

but to act as a modulation source to alter one or more parameters of the synthesizer. In our system, we used an LFO to modulate the frequency, or pitch, of both synthesizers.

The LFO was created in the top_level module, and consisted of an oscillator module feeding into an amplitude control module. Both of these modules work the same way as they do in the main synthesizer modules, with the exception that the LFO oscillator generally receives a much lower frequency input.

To modulate the frequency of both of the synthesizers, the 16-bit signed waveform is simply added to the frequency input to both synthesizers. When the LFO waveform has a positive value, it adds to the frequency input of the synthesizers and increases their pitch. A negative value of the LFO waveform decreases the frequency of the synthesizers and results in a lower pitch. The amplitude of the LFO affects how much the pitch changes by and the frequency of the LFO affects the rate at which the pitch changes occur.

This is not the most ideal way to implement a LFO on pitch, but it is the one that involves the least amount of computation. To see why this is not the best method, consider a situation where the synthesizer is set to a base frequency of 300Hz, and the LFO has an amplitude of 150. When added together, the synth frequency reaches a minimum of 150Hz, which is 1 octave lower than the base frequency, and a maximum of 450Hz, which is only 0.5 octaves above the base frequency. This discrepancy occurs because addition is a linear change while changes in pitch are perceived logarithmically. Moreover, with this addition approach, the LFO, at the same amplitude, has different effects on the pitch of the sound at different base frequencies. For example consider another situation where the synthesizer is set to a base frequency of 60Hz, and the LFO has the same amplitude of 150. In this case, when added together, the synth frequency reaches a maximum of 210Hz, which is a little less than 2 octaves above the base frequency, and a minimum of -90Hz, which causes the unsigned frequency variable to roll over. To prevent these problems and create an LFO that has the same effect on pitch regardless of the base frequency, the formula below should be used to calculate the new frequency.

$$(new\ frequency) = (base\ frequency) \times 2^{((LFO\ normalized)\times(modulation\ depth))}$$

LFO normalized refers to the LFO waveform normalized to oscillate between +1 and -, and the modulation depth is the number of octaves the LFO should shift the base frequency at maximum amplitude. Because of the complexity and additional fpga resources required to perform this calculation using fixed point numbers, and because we were satisfied with the pitch changes produced by the LFO through the addition method, we made the choice to stick with the former. This works well enough in our system because the range of base frequencies requested from the synthesizers—usually between 200Hz and 520Hz—is fairly small, so the effect that the LFO has on the pitch of the sound will be more or less consistent across the frequency range.

The effect of a large LFO amplitude on a small base frequency value was an issue that still needed to be dealt with. In these cases, the frequency value rolls over and produces unpleasant high frequency tones. To prevent this, a simple conditional statement, highlighted below, is used to check if the signed value of the frequency will go below zero. If so, the frequency is simply set to 0Hz, but if not, the LFO is added as usual.This effectively hard clips the frequency value so it never goes below zero .

```
synth1_frequency <= (red_area>=detection_threshold)?
(lfo_enabled?(($signed(lfo_out+12'd200+red_center_h_index)>=12'sd0)?(12'd200+red_center_h_index
+lfo_out):12'd0):(12'd200+red_center_h_index)):12'd0;
```

PWM Audio Out (Prajwal)

All the audio waveforms inside the FPGA are represented as signed 16-bit numbers that hold information about the amplitude level of that waveform at a particular point in time. However, these digital, signed 16-bit numbers can't directly be listened to as audio. The pwm module bridges this gap between numerical representation of waveforms and an actual analog audio signal we can listen to.

The audio output jack on the FPGA board is preceded by a 4th order Butterworth lowpass filter, which can be used to create an analog waveform from the digital output of the FPGA. This is done by using a high frequency digital carrier wave, whose pulse width is modulated by the amplitude of the desired waveform. As the carrier wave passes through the low pass filter, it essentially gets integrated, and the filter outputs an analog waveform whose amplitude is proportional to the width of the pulse.

A common way to create this digital PWM signal is to use a fast moving ramp signal to compare the digital waveform against as demonstrated in Figure 9. In our system, we use a 16-bit ramp that increments by 256 on every input clock edge. With a 65Mhz clock, this results in a ramp with a frequency of approximately 250kHz. This ramp is compared against the digital output waveform of the system, which is converted to an unsigned number before being sent into the module. When the amplitude value of the ramp is above that of the waveform, the PWM output to the filter is pulled low (i.e. 0) and when the amplitude value of the ramp is below that of our waveform, the PWM signal is pulled to a high impedance state (i.e. Z).

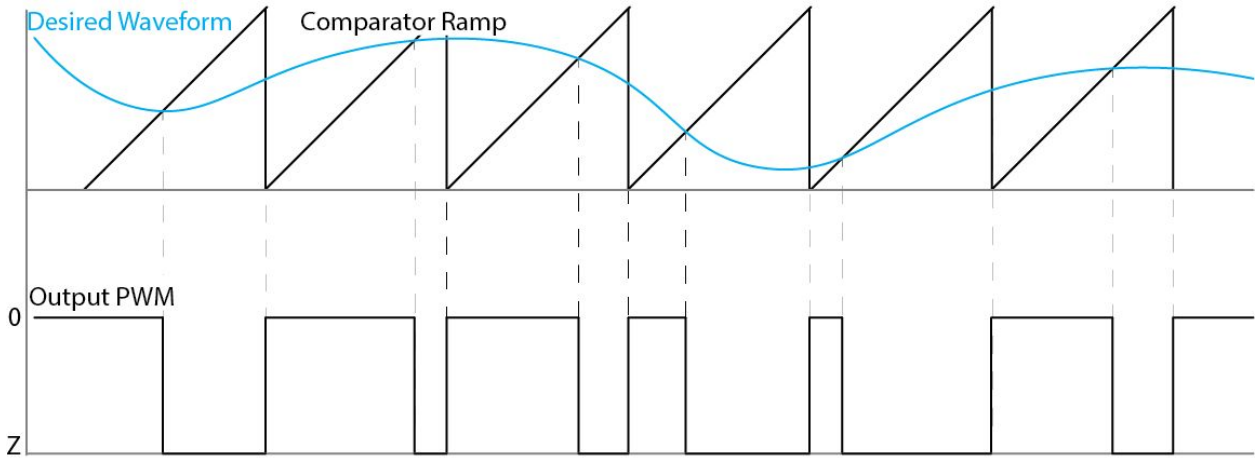

Figure 9: Ramp waveform and desired signal (top) with the resulting PWM waveform

Controls Extraction

Camera_to_mask (Sage)

The camera to mask module is responsible for reading from the camera and converting these readings into several pieces of processed information that will be used as inputs to the sound generation modules and for the displays. These pieces include per-color classifications (booleans for red, green, and blue) of the pixels every frame, and the centroid in (*x, y*) space and area of each group of classified pixels. This module effectively returns a "mask" filtering the video for each of the red, green, and blue LEDs. Each mask is simply a frame of pixels that are either black or one of {red, green, blue}. The mask can then be used to display a black screen with a moving, colored blob, representing the corresponding LED in space, and containing crosshairs meeting at the center. This information is stored in a dual-port BRAM frame buffer (1x76800 for each color) for such use. Additionally, the raw video pixels are stored in a 12x76800 BRAM.

The first step in implementation of **camera_to_mask** is obtaining the pixels from the camera by interfacing with the camera reading code. From the camera starter code, a 16.25MHz clock called *xclk* is created by counting every 4th rising edge of the 65MHz system clock (so, 25% the speed of our system clock). The camera is driven with this clock via the pin *jbclk_p* from the JB PMOD port on the FPGA side. The JB pin 0 holds the returning pixel clock, *pclk,* from the camera. This clock, along with *vsync* and *href* are used by the provided **camera_read** module to read and output the incoming 320x240 frames of pixels serially. The module outputs 16 bit pixels, a storage address for BRAM, the (*x, y*) coordinates of the pixel (a modification we made to the code), and booleans indicating whenever a new valid pixel is outputted as well as whenever all the pixels in a frame have been outputted.

The serially outputted pixels then enter a processing pipeline as shown in figure 10 below.

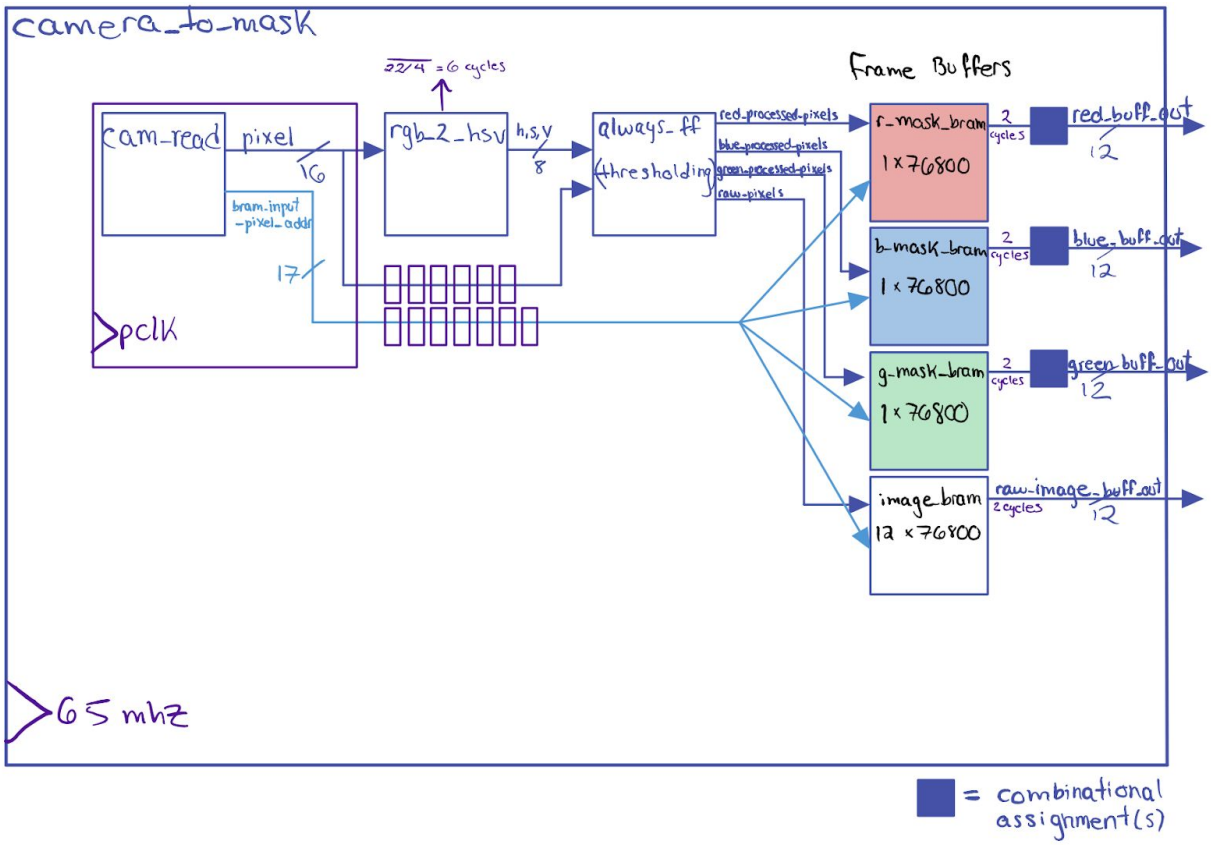

Figure 10: Block and Timing Diagrams of pixel storage pipelines in ***camera_to_mask***

of BRAM. The first option is to simply send the raw output pixel to image_bram, for which there is only a latency of 1 as it is sequentially converted from a 16 bit 5-6-5 RGB value to a 12 bit value holding the upper 4 bits of each color. The second option is to send the pixel into red, blue, and green BRAMs which contain the classified pixels. For example, a red LED pixel in red BRAM will be stored as a 1'b1 and anything else will be stored as a 0. This is done by first passing the pixel through an RGB to HSV converter module, **rgb_2_hsv**, which converts the pixels from RGB values to HSV values. We take full advantage of the available RGB resolution by passing all 5 bits of red (concatenated with three zeros for the 8-bit input requirement), all 6 bits of green (concatenated with two zeros), and all 5 bits of blue (concatenated with two zeros). The decision to switch to an HSV color space was made to reduce the amount of color bleeding/incorrect classifications that would be present in thresholding RGB values. This module takes twenty-two cycles per new pixel (at *pclk*), and since pixels are coming in at a rate of 1/4th the system clock, there is an overall latency of six 65MHz clock cycles.

Continuing on path 2, the hsv values are then thresholded and stored in BRAM as 1s or 0s at the per-pixel incrementing input address (1st port) provided by **cam_read.** depending on if they meet that color BRAM's thresholding criteria.

Thresholding (Sage)

Thresholding (chroma keying) the camera images was approached by using trial and error from several different angles.

Thresholds for hue were created by examining an HSV palette and choosing rough boundaries. Then, these values were fine-tuned experimentally by using the VGA display of the color masks. Despite a suggestion to try thresholding only based on hue from past experience, we found that doing this created too much noise and therefore saturation and value needed to also be considered. As value was raised, more noise was picked up, and as it was reduced, less of the LEDs were detected. We settled on an optimal value for each LED experimentally. For saturation, a number had to be chosen such that it is low enough to overcome the effects of light from the LEDs reducing the saturation of the pixels on the LED but high enough to filter out non-red, blue, or green sources of light. Additionally, other objects in the room that were in fact red, blue, or green needed to not fall into the thresholds. The success we had with this was mostly in thanks to the use of LEDs themselves and their characteristic lower saturation than non-illuminated objects. Still, we struggled to get the camera readings correctly classified even after tuning the threshold values:

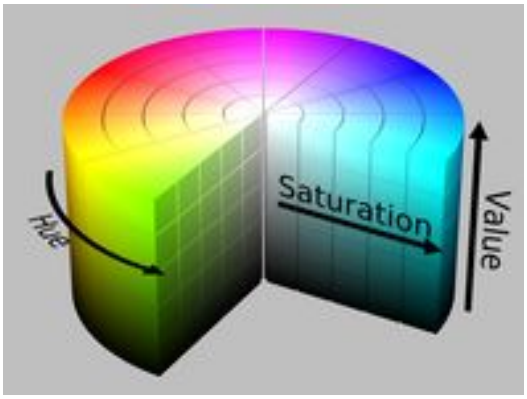

Figure 11: HSV palette

One challenge we faced with using LEDs was related to the aforementioned reduced saturation due to the light. Because the LEDs were so bright, the camera picked many of the pixels up as high value white pixels. We reduced the brightness on the physical LEDs to a much lower setting and found that this improved the issue of the light overpowering the color. Interestingly, the different colored lights responded differently to this tuning, with each color requiring a different optimal brightness. We also tuned the camera control settings in ov7670_control.ino, including the RBG gains and automatic gain and exposure control to optimize for noise reduction and picking up enough LED. Because of the success we had with this and thresholding, we did not find it necessary to rely on the ILA or a calibration module for experimentally determining the HSV value of a particular pixel, though this approach was considered.

Another challenge we had was with noise created by the LEDs casting shadows on our clothing, arms, nearby objects, and the other 1-2 LEDs. While the aforementioned reduction in brightness significantly relieved this issue, and probably enough to not have significant error in the area calculations, there was still a non-trivial amount of noise due to shadows that would interfere with the center calculations. We decided to cover the LEDs with socks as a form of containing and softening/smoothing the light.

One observation was that the brightness of the circular lights on the LEDs overpowered the color and they were not picked up by the thresholding. Thus, the color masks appeared as colored circles with smaller black circles (the lights) inside of them. Each LED responded differently to the thresholding. Interestingly, this effect was prevalent in the green lights, and less in the red, and the opposite effect existed in the blue. Because of the uniform distribution of these lights on the circular LED face, this effect did not interfere with the center-finding calculations. However, there would certainly be a decrease in area. Since area is a relative control in our case, a simple scaling of the green area relative to the red and red relative to blue would have been a sufficient solution, but we found the effects of different area scales to be insignificant for our purposes. Another possible solution that might have been worth exploring was using the HSL color space, with the value from HSV replaced with lightness by HSL.

While most of these designs helped make the thresholding more robust to lighting changes, we left available different switches (7-10) containing different threshold settings for users to choose what works best for them.

Returning to the block diagram on page 10, we can see that **camera_to_mask** uses the booleans stored in BRAM to output 12 bit classified pixels that are either red (12'hF00), green (12'h0F0), or blue (12'h00F), or it returns the raw image pixel. The 2nd port address to select a value from BRAM is provided as an input to the module.

Finally, the last responsibility of **camera_to_mask** is outputting the centroid and area of each frame. This is done by passing the pixels and corresponding indices from **camera_read** directly into the **center_finder** module and routing the results as output to **camera_to_mask.**

The camera_to_mask module was built incrementally as we added increasing functionality and control extraction to it. In retrospect, earlier planning of strictly what will be contained in and outside of the module would have saved the module from growing very large and perhaps a bit too bulky. Breaking down **camera_to_mask** into smaller modules—especially separating the instantiation **center_finder** from the module—would have been a cleaner and safer design decision. One plausible breakdown would be: a module to receive the data from the camera hardware and pass it through **camera_read** (this would be entirely on pclk), a module that, receiving these values at pclk, passes them through the pipeline in figure 10, but still on pclk, and a third module that extracts the frames from BRAM (now crossing clock domains to 65 mhz) and interfaces with center finding to extract the center and area controls (using BRAM address to calculate indices). On page 19 is a discussion of challenges encountered with the overall timing of the pipelines which dives further into the design improvements we learned. First, we discuss the **center finder** module and displays.

Center Finder (Prajwal)

The **center_finder** module performs a simple center of mass calculation on each frame of the thresholded masks. There are three separate instances of **center_finder** inside **camera_to_mask**, each responsible for finding the centroid and area of one of the three thresholded colors. Each of these instances is fed with binary values corresponding to whether or not a pixel falls within its thresholding bounds along with that pixel's horizontal and vertical location in the frame.

Inside the module, an FSM, whose state transition diagram is shown in figure 12, performs the center finding operation on each frame. At the start of each frame, the all values are zeroed and initialized before the FSM enters an "IDLE" state. Here, we check whether or not a new pixel has been received, and whether or not that pixel has a value of 1, meaning it did fall inside the thresholding bounds for that color. If both of these conditions are met, the FSM moves to its "NEWPIXEL" state, where the horizontal index and vertical index of that pixel are added to a running sum. Another variable, num_pixel, keeps track of how many of these active pixels are counted in each frame. The FSM then returns to its "IDLE" state where it once again waits for the aforementioned conditions to be met. This process continues until the indices of the incoming pixel indicate that we are reaching the end of the frame. At this point, the FSM transitions to its "DONE" state, where a divider IP begins the process of dividing the horizontal index and vertical index sums by the number of pixels counted. The results of these divisions are the horizontal and vertical indices of the center of mass of thresholded pixels.

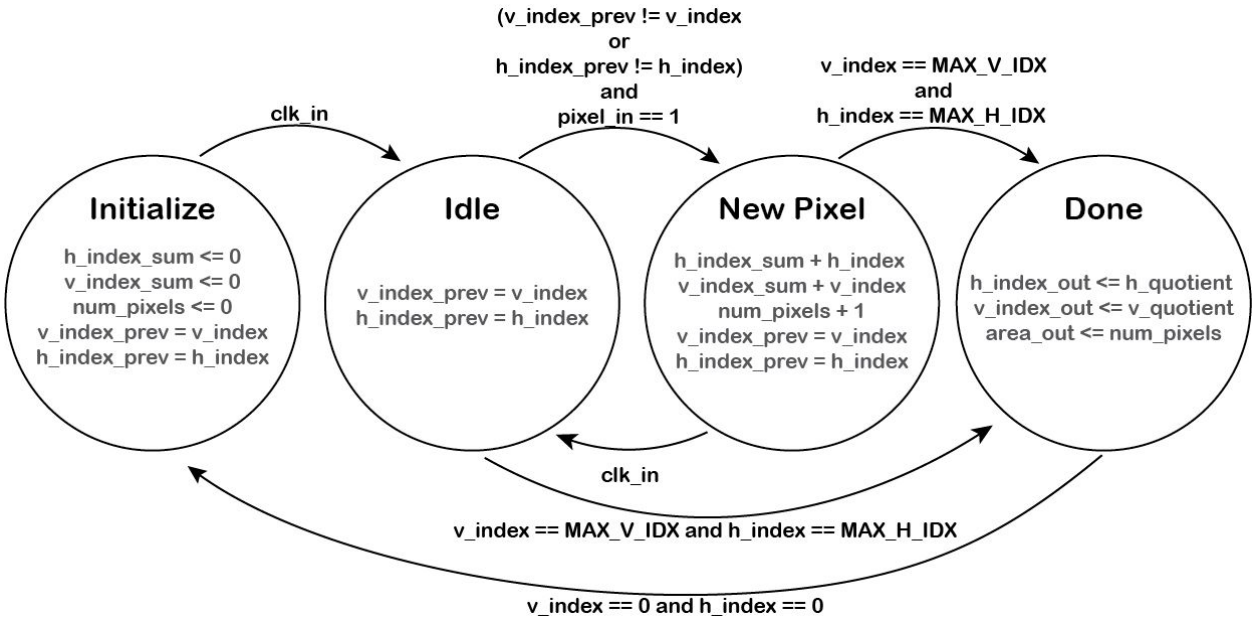

Figure 11: Simplified state transition diagram for center_finder FSM

Because division is a costly process in terms of resources and most importantly, time, the size of the division had to be chosen carefully so as to minimize the number of clock cycles needed to perform the division. The dividend would reach its maximum size if the index of every pixel in

the frame was added together. For the horizontal index sum, this would mean the sum (0+1+2+3…+319) for all 240 rows. The equation below, where n and m are the maximum horizontal and vertical indices, can be used to find that the largest number this will reach.

$$m \times n(n+1)/2 \;=\; 240 \times 319(319+1) \;= 12249600$$

This tells us that the minimum size of our dividend needs to be 24 bits to hold the maximum possible sum. The divisor, which is just a count of the number of active pixels, can reach a maximum of 76800—i.e.the number of pixels in one 320x240 frame—which can fit into a 17 bit number. With these sizes, an unsigned division with the help of the divider ip takes 26 clock cycles to divide. To accommodate for the delay, we simply stop the FSM a little bit early. Instead of letting it run to the last pixel in the frame, we transition to the "DONE" state as soon as we reach the beginning of the last row of pixels. This doesn't really affect the precision of the center finder, and gives us plenty of time before the next frame starts for the division to complete.

One thing to note is that the **center_finder** module runs on the common 65Mhz clock , while new pixels only enter at the slower p_clk_in. To compensate for this difference, the FSM only moves forward when a change is detected in the pixel's indices, which indicates that a new pixel has been received. When in "IDLE", the FSM checks the current indices and compares them to the values they had the last time it checked. The pixel is only considered for the "NEWPIXEL" state if these indices are new.

Displays (in top level) (Sage)

The code responsible for routing the correct pixels to the 1024 x 768 VGA is found in the top level. The VGA-related assignments are organized in a series of conditionals found in an *always_ff* block. In this block, assignments are made to *current_pixel,* which the *vga_r, vga_g,* and *vga_b* pins for the VGA circuit are assigned to combinationally.

The flow of conditional logic is as follows, in pseudocode:

For top half of the VGA display:
if sw[2] and in top left 320x240 pixels of screen:
    //want to display raw video
    current_pixel <= corresponding raw image pixel
    address to pull from BRAM <= current location of VGA pixel
else if ~sw[2] and in top left 320x240 pixels of screen:
    //want to display red mask
    current_pixel <= is this location on a crosshair ? is red area above threshold ? Magenta :
    corresponding red mask pixel : corresponding red mask pixel
else if ~sw[2] and in second 320 pixels of top 240 pixels screen:

```
        //want to display blue mask
        if this location is a region divider, current_pixel <= yellow
        else do same as red, but for blue mask

else if ~sw[2] and in third 320 pixels of top 240 pixels screen:
        //same as red, but for green mask
```

The first line of pseudocode indicates that sw[2] is an option for the user to see their raw video (of course with the synthesizers still working). Otherwise, the three side-by-side color masks will be displayed.

The magenta crosshairs are only displayed when the area of that LED is above a detection threshold. This was decided to provide a nicer visual that lets the user know the red pixels on the screen are only being tracked when the system believes it is an actual LED. The value of the detection threshold was experimentally chosen to be 200 so that the user can stand far away from the camera (~5 feet) and still control the synths. If the threshold is not passed, the red or black pixel that would otherwise be in that location is displayed instead of the crosshair pixel.

The word "corresponding" in the pseudocode refers to whatever pixel in the raw video/r/g/b mask frame buffer matches the indices of the 320x240 sub-screen on the 1024x768 VGA. The way this pixel is obtained is by a combinational assignment to the corresponding BRAM 2nd port address. The current coordinates on the VGA are converted to an index in the flattened, 76800 long array of BRAM address space, representing a single 320x240 camera frame:

```
assign blue_buff_output_pixel_addr = (hcount-11'd320)+vcount*32'd320;
assign red_buff_output_pixel_addr = hcount+vcount*32'd320;
assign green_buff_output_pixel_addr = (hcount-11'd640)+vcount*32'd320;
```

Note that hcount and vcount are variables in top-level that come from the xvga module.

Initially, these address assignments were updated in the sequential conditional blocks. For example, the pseudocode above for the red mask originally looked like this:

```
else if ~sw[2] and in top left 320x240 pixels of screen:
        //want to display red mask
        current_pixel <= is this location on a crosshair ? is red area above threshold ? Magenta :
        corresponding red mask pixel : corresponding red mask pixel
        red_buff_output_pixel_addr <= hcount+vcount*32'd320;
```

However, a cleaner code that avoids inconsistent pipeline delays between hcount, vcount, and the address assignment follows the previous design. See the two block diagrams below for the distinction.

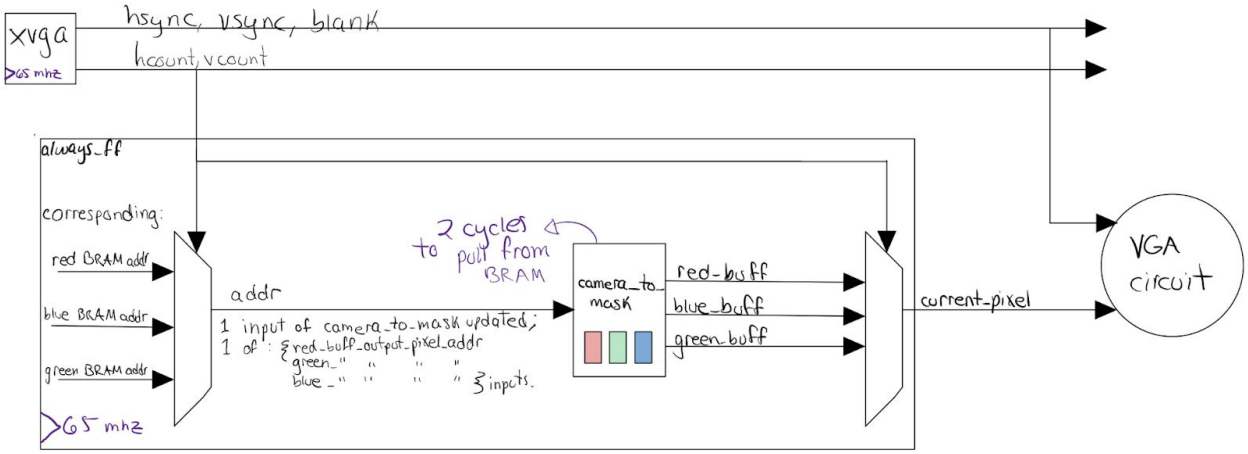

Figure 13: Original pipeline: inconsistent timings of hcount and vcount

Bypass 1st mux by updating all red, blue, and green addresses:

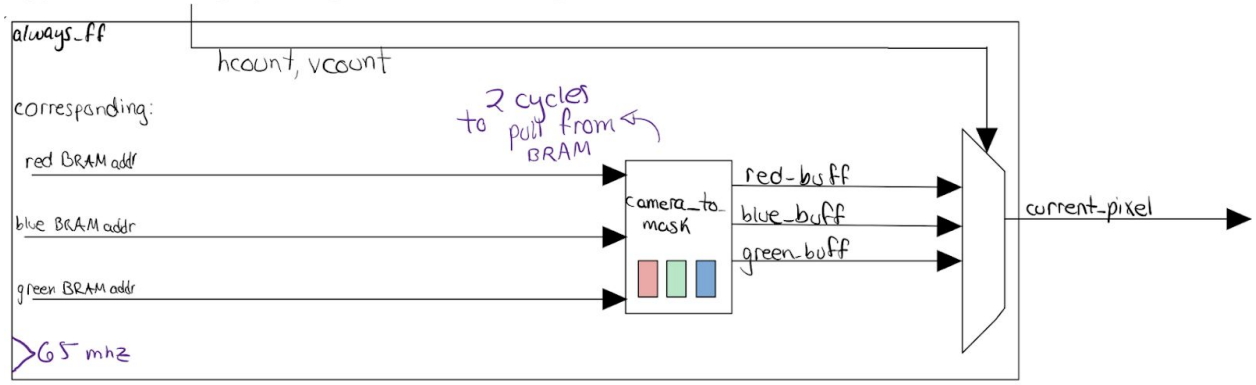

Figure 14: Improved, cleaner pipeline with only one timing for hcount and vcount

For the bottom half of the VGA, we follow a similar trend of conditionally setting the current pixel to the correct bar graph pixel. To create the bar graphs, we had to choose a mapping of ranges of values from the original camera frame to ranges of values in the bar graph (i.e., height of bar graph in number of pixels). To neatly fit onto the screen with labels, we had about 400 pixels of height to use.

The ranges for the bar graphs were chosen so that we could avoid the overhead of using divider IPs. For the horizontal position, the range is exactly the horizontal range of the camera 0-319, resulting in a 320-pixel tall bar. An analogous 240 range was chosen for y position. Area was a little more difficult because 76800 distinct possibilities (max area is 240*320) could not fit in a single bar graph. We chose a range of 300 pixels, so the 76800 range was bucketed by normalizing to 300. Fortunately, we could avoid scaling by 300 with a multiplier then dividing by 76800 with a divider by instead right shifting 76800 eight times as 76800/256 is exactly 300.

image_ROM_synth (Sage)

This module creates the picture blob for any picture displays using ROM that we would like to display. We wanted to display some visual of the controls and LED movements changing in real-time for users to understand the effect their movements have and gain an intuition for the response and range of motion they can adopt.

This module uses the code from lab 3's **picture_blob**, with modifications to BRAM sizes and address widths to support our particular image dimensions and the use of all three colors instead of a grayscale. After deciding between multiple images, the image we chose to display was indeed on a grayscale, so the code would produce sufficient results if kept in grayscale, but we chose to leave it using all three color maps should we wish to add colored pictures in the future.

We used the Python code from lab 3 to convert our custom image files into coe files.

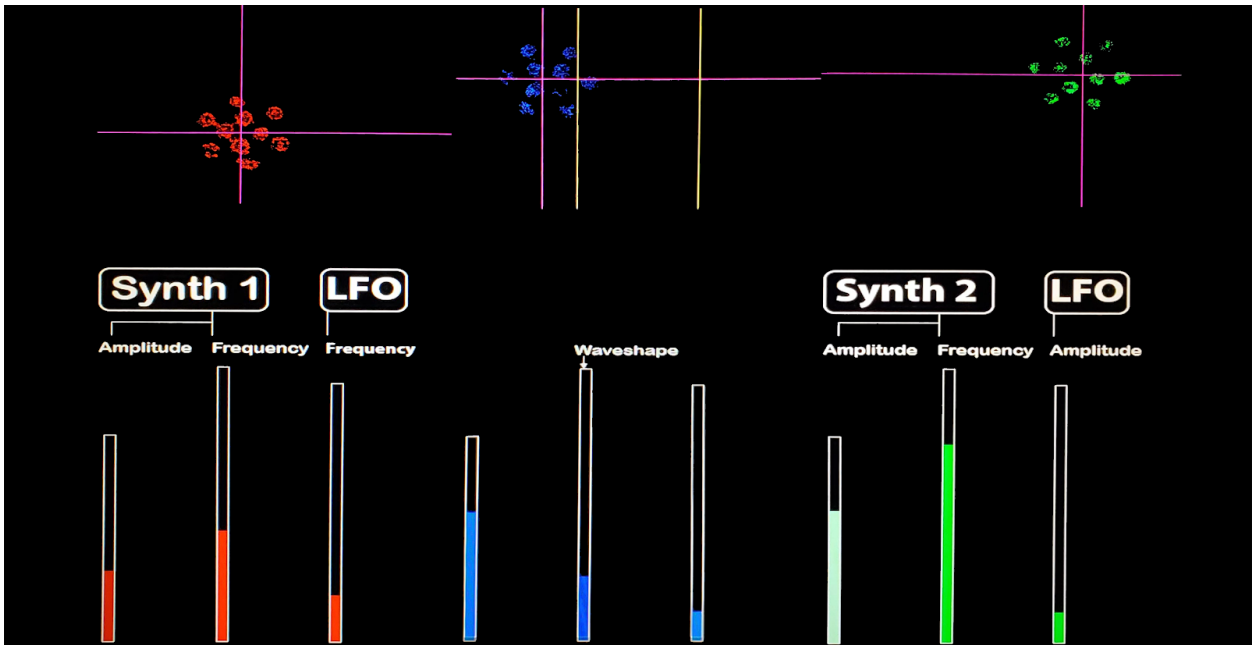

Figure 15[1]: Display of pixel masks and controls

Figure 15 shows the final display that was used. The words seen on the screen are the custom-created 1024x190 image that was stored in ROM.

Each bar graph shows the value of a different physical parameter relating to the above LED. From left to right, there are three bar graphs for each of red, blue, and g[1]reen. Of each triplet of bar graphs, the left bar represents the y-position of the LED, the middle bar represents the

[1] Notice that the lights are being picked up instead of their surroundings, contrary to the thresholding discussion. This occurred toward the end of the project for one of the two students in the team and is possibly attributed to lighting changes or use of a thicker sock.

x-position, and the right bar represents the area/depth. Above each bar graph is a description of which parameter is affected and which oscillator (LFO or Oscillator 1 of a particular Synthesizer) the parameter modulates. Ultimately, the left and right blue bars were not needed for any synth controls. Since adding the third blue LED was part of our stretch goals, we left the bar graphs less finalized, keeping the unused graphs so that the user can still see a visual of where the blue LED is on the screen and its area, just as the user can with the red and green LEDs. This also leaves the synthesizer more open ended should someone want to play with the code to add or change the controls.

Challenges with timing (Sage)

We now continue with a discussion of timing challenges that spanned pipelines in **camera_to_mask** and **top_level**. After implementation, we noticed a few timing issues that could be seen on the display and crosshair locations. Debugging these was one of the more tedious processes in the project. First, there were artifacts of pixels on the left edges of the display's sub-screens. These were bars a few pixels wide that should have been displayed on the right borders, at hcount = 319, 639, 959 but were instead displaying on the left borders.

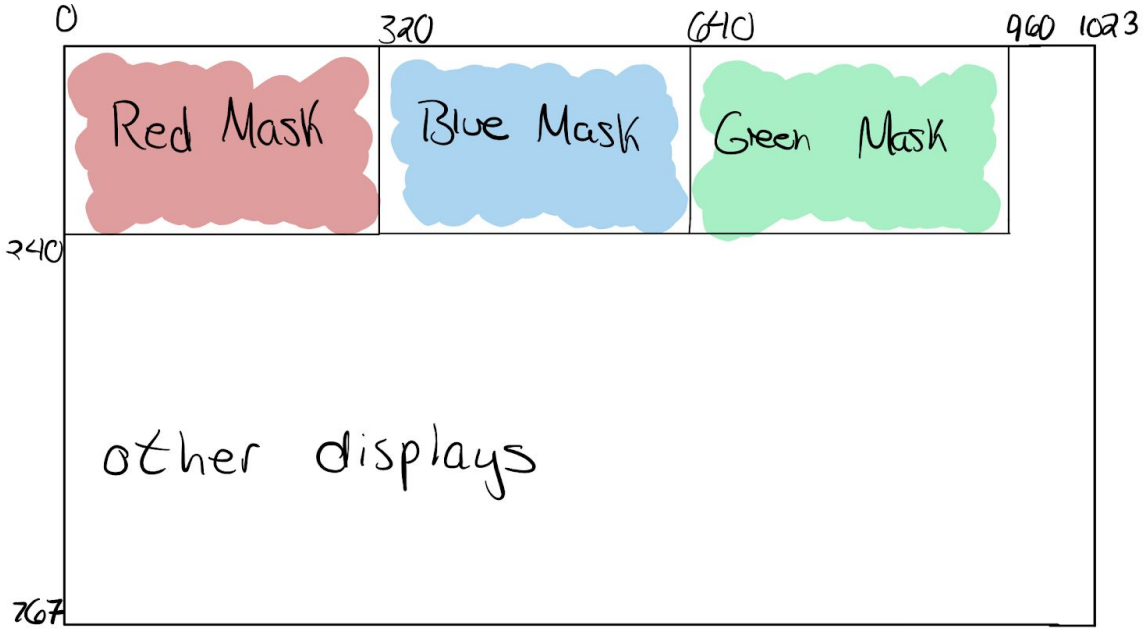

Figure 16: Organization of display when viewing masks, labeled with pixel values

Below are two images of what this effect looked like in the beginning. The bars appear to be several pixels wide. On the raw video, there is also a ~6 pixel wide black bar and a blue bar (the blue bar was determined to be a part of the camera setup rather than a pipeline issue, and the black bar was at some point resolved in our series of many timing changes/debug attempts).

We examined our pipelines and found multiple cycles of delay in current_pixel relative to hsync and vsync. Looking at the original **top_level** displays pipeline in Figure 10, it takes 4 cycles to update current_pixel after a vsync, hsync, and blank are provided. These delays would be responsible for the bar of pixels noticed in the raw video display, but not other displays, since, despite our creation of sub-screens, hcount still wraps around only once at 1023.

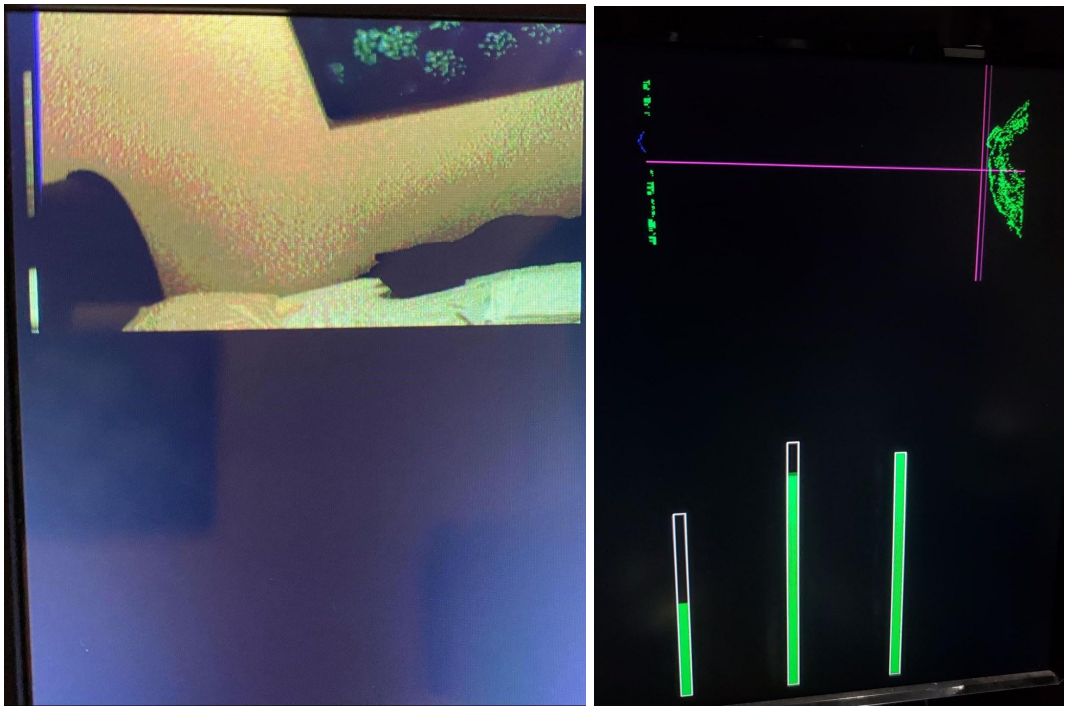

Figure 17: VGA timing artifacts in raw video (left) and green mask (right)

Upon attempts to delay vsync, hsync, and blank by 4 cycles using shifting arrays, we saw improvements in the artifact width but it was not entirely gone. It took 5 cycles of delay to fully remove the artifact, but we could not find the source of the 5th cycle. This was likely some other subtle delay, perhaps not in **top_level**, that we covered up without solving the root of the issue. We then turned our attention to the pixel masks and the **camera_to_mask** module. The red mask's bar was not gone, even though it consumes the same location as the raw video, however, as we then expected, it at least had an improved width relative to the green and blue masks' bars. Since entirely removing the raw video artifact was not enough to remove the red bar, it means there was another source of error unique to the color masks. Perhaps fixing things in **camera_to_mask** would resolve the extra unknown cycle in the raw video as well.

Examining **camera_to_mask** allowed us to find remaining pipeline flaws, but a full fix was not achieved. As seen in Figure 10, the series purple registers in the diagram were placed as delays to align path timings. This reduced the mask artifacts down to a couple pixels wide.

Along our path of debugging, we also encountered an issue where, in contrast to Figure 17, the pixels on the left were the same pixel for a few locations in space. We resolved this by removing a bug that stretched the pixels, putting the same ones multiple times in BRAM.

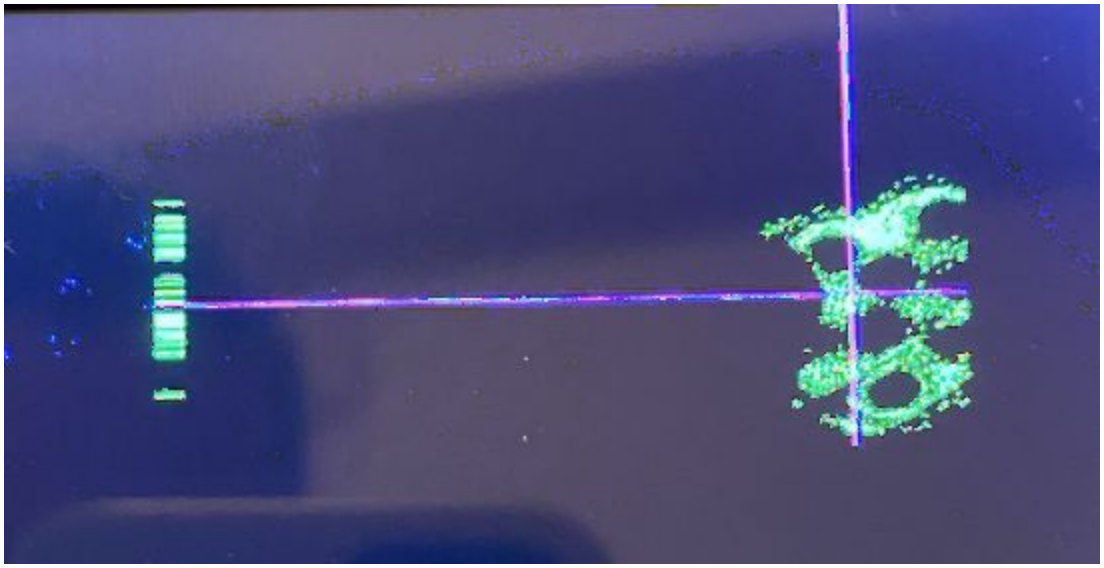

Figure 18: Stretched pixels on left-hand side of the screen

Another possible source of timing misalignment was the crossing of clock domains in **camera_to_mask**. Pixels were being outputted at a rate of pclk, but they were being analyzed at the speed of a 65 mhz clock. While the code ensured that new processing only occurred when a valid new pixel was introduced (so as to avoid the same pixel being stored in BRAM multiple times), we concluded it would be cleaner to only cross clock domains through one place: the BRAM. We changed **rgb_to_hsv** and the **always_ff** in figure 10 to run on pclk instead of the 65mhz clock, and then adjust the register delays accordingly. However, this produced no changes on the display, and even introduced some other strange display bugs. Despite cleaning the code by doing the above, the drawbacks that it introduced suggested that not everything had been exhaustively aligned, and it would be messy to continue to make those improvements one by one, so we had to revert to those modules using 65mhz and relying on their internal mechanisms for processing new pixels once every four cycles as mentioned above. In the future, a design that crosses clock domains more cautiously and meticulously is significant.

Another issue that was more pressing was the center finding calculations. As the LEDs moved to the right edge of the screen and began to go off the screen, the crosshairs would move significantly to the right. That is, much more than would be expected should the source of this only be the extra pixels on the left hand side. This only became prevalent after several pipeline alignments for the display bars had been made. We looked to camera_to_mask and center_finder to try to make further changes. These included delaying the horizontal and vertical indices going into **center_finder** so that they are associated with the correct pixel and trying a number of delays on the BRAM input address for testing whether this has an effect. We did not have much luck with these changes, and decided to revert a few of our earlier changes for the sake of sparing the effect it had on the crosshairs.

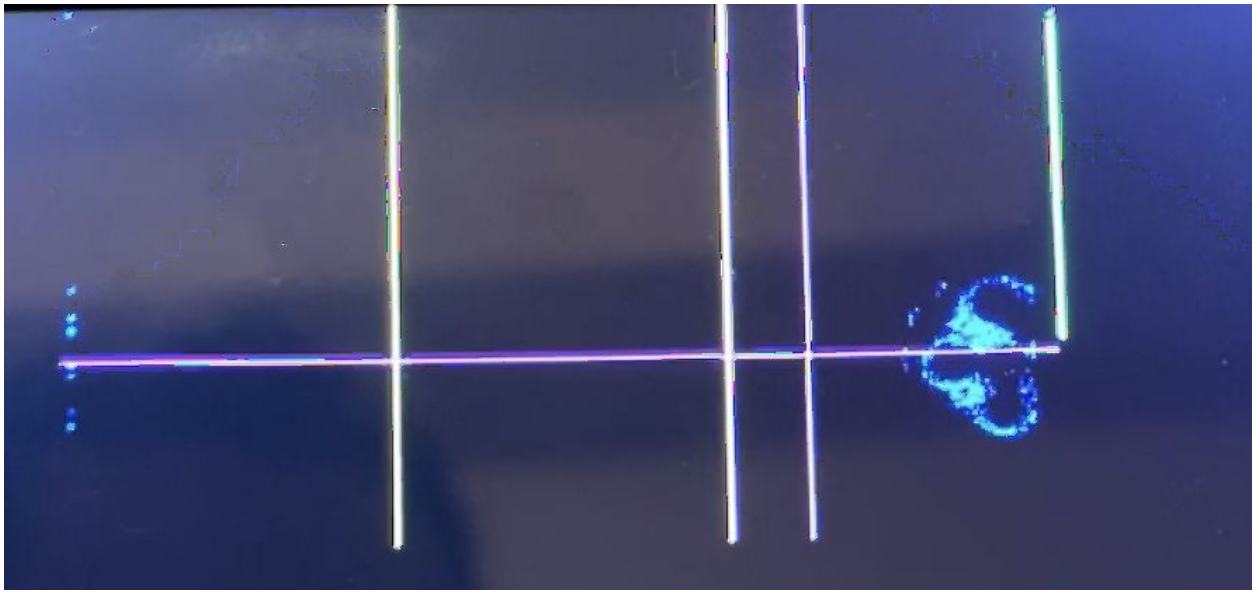

Figure 19: low crosshair accuracy after improvement to pixel bars (ignore the yellow bar on the right, used for debugging purposes)

Further, after changing to the improved displays pipeline that avoids inconsistencies with hcount and vcount, (Figure 14), the aforementioned delay to hsync, vsync, and blank is reduced to three. However, we still found that anything lower than five re-introduced the raw image pixel bar.

Ultimately, we made several attempts to discover and remove the somewhat scattered sources of timing delay across **camera_to_mask** and **top_level**, and some solutions worked and were kept, while others were discarded, and others did not work. It was a balance of tradeoffs in displays and crosshair calculations, and though we were able to reduce our pixel bars and remove the stretched pixels to an extent, we mostly erred on the side of crosshair calculations. Overall, several design insights were learned:

For any system whose calculations and displays rely on camera and frames of pixels, and more generally for any clocked system on which there is more than a single pipeline path (so, basically all clocked systems), planning the pipelines down to the clock cycle is very important. We spent a good deal of time considering the logical relationships of our modules, logics, and pipeline paths, but in the future we would spend more time drawing block diagrams on a per-cycle/pipeline path basis that would have allowed us to exhaustively consider the timing relationships. This would result in a less cautious approach of building these modules and their relationships, avoiding a build up of timing issues that were only faced by the end of most of the implementation.

Top_level (Sage)

The top level module assembles together the results of camera readings and the audio generation by linking the parameters received from **camera_to_mask** to the frequencies and amplitude of the two synthesizer's primary oscillators, the LFO amplitude and frequency, and the synth oscillators' waveshapes. The LFO waveshape (four options) is changed with sw[0:1]. The low pass filter cutoff frequency is tuned by the separation distance of the hands. We initially planned to use separation distance in two dimensions, but to avoid the cost of implementing a square root as seen in the quadratic formula, we chose to only evaluate the separation distance in the x dimension. Similar to the requirement for a threshold area to be surpassed for crosshair display, all controls only update the audio generation parameters if the responsible thresholded colorblob surpasses the threshold area of 200. Additionally, the LFO can be toggled on or off using sw14 on the FPGA.

External Code Sources and References

Camera starter code, Joe Steinmeyer

Includes:

camera setup code found in **camera_to_mask**

cam_read

rgb_2_hsv, Kevin Zheng

Image blob code, Lab 3

Audio starter code, Lab 5a

Acknowledgments

We'd like to thank Prof Joe Steinmeyer for the excellent lectures and mentoring that helped make this project possible. We'd also like to thank Brian Sennet, our industry mentor from Bose, as well as all the TAs that helped us get this project off the ground, solve problems, and make improvements to our system as we brought it to its final state.

Appendix

All project code can be found at the following Github link:
https://github.mit.edu/prajtm/SpaceSynth