Maestro

6.111 Final Project
Fall 2014

Evie Kyritsis
Sabina Maddila
Janelle Wellons

Abstract

One of the main reasons for playing a musical instrument is that the experience is fun and enjoyable. However, to be decent at playing an instrument requires practice, and if you play a string instrument, or have a condition such as arthritis, the experience of the musical instrument can immediately become painful and uncomfortable.

Maestro proposes an instrument that can be played without having to physically touch it. The user, wearing two distinctly colored gloves, simply has to wave their hands in front of a camera that will track their hand movements and parse the information into an X and Y coordinate. These coordinates are then mapped onto a sound and color wheel[1] that produces notes based on the user's hand positions and lights up the corresponding color of the note.

The user of Maestro gets to enjoy not only sound, but also aesthetically pleasing visuals that accompany the sound produced, thus creating an enjoyable, artistic experience.

[1] Detail regarding the color and sound wheel is described further below in Figure 12.

Contents

Abstract……… 2

I. Overview……… 4

II. Modules……… 5

1. TopLevel Module……… 5
2. Motion Tracking - Evie……… 6
   - 2.1 NTSC Decoder……… 7
   - 2.2 ZBT RAM……… 7
   - 2.3 Color Detection……… 9
   - 2.4 Crosshairs……… 11
   - 2.5 Mapper……… 12
3. Visualization - Janelle……… 13
   - 3.1 Sprite Generator……… 14
   - 3.2 Geometry……… 16
   - 3.3 Piano Logic & Color Wheel……… 16
4. Sound Generation - Sabina……… 20
   - 4.1 Generating an Oscillator……… 21
   - 4.2 Amplitude Modulation……… 25
   - 4.3 Post-Processing……… 27
   - 4.4 lab5audio……… 28
   - 4.5 Module Testing……… 28

III. Testing……… 28

IV. Review……… 29

V. Conclusion……… 29

Appendix I - Verilog Source Code ……… 30

I. Overview

As seen in Figure 1 (below), the system is comprised of three main components: motion tracking of the user's hands, mapping the generated coordinates in visualization to light up a color on the wheel that is displayed on a monitor, and generating the sound corresponding to the position and color information provided by the former two blocks. All of which are broken down into the Verilog modules that comprise the subparts of each block.

The **Motion Tracking** block processes the user input into the NTSC camera provided by the 6.111 staff (Figure 3) and sends off the relevant signals to the Visualization block. First we have the tracking of the hands, which is done using two distinctly colored gloves, red and green, so as to allow for the ability to play two simultaneous notes. Identification of the hands is done via processing the NTSC data from the camera. Taking in the movements of the user's hands, the pixels of the gloves are identified and then averaged in order to produce the X and Y coordinates of the center of mass for each hand. The center of mass coordinates of each hand are then mapped from the reference frame of the video image[2] to the coordinate system of the whole 1024 x 768 screen for the Visualization component of the system.

Visualization generates sprites, two white boxes, via a VGA digital to analog output that follow the movement of the user's hands on a monitor in real-time. The sprites move about the screen over the color wheel, and light up the slices of the wheel as they move over the corresponding areas. The only inputs that the visualization block takes in are the Cartesian coordinates of the center of masses of the user's hands from the motion tracking component. The colors that are lit up at any given time by the sprites are sent off to the sound generation block in order to be calculated into specific notes. The AD7125 on the FPGA board converts the XVGA signals and sprites into a VGA output on a computer monitor.

Sound Generation takes in the color outputs of the visualization block as well as external outputs from the labkit switches. The colors map to a specific note in the G major scale[3] which corresponds to a specific frequency. Sound is synthesized using a sinusoidal oscillator, whose frequency is determined by the visualization module. The resulting sinusoid is then shaped by an ADSR (or Attack Decay Sustain Release) volume envelope, which is triggered by one of the labkit switches (the envelope is then triggered every other second). Another labkit switch allows for chords, which is accomplished through additive synthesis. After these

---

[2] The image of the video fills an area of about 725 x 505 on the 1024 x 768 screen.

[3] Based on the physical representation of the sound wheel which is depicted in Figure 12.

modifications, the resulting wave is sent to the ac97 audio codec, which in turn drives a pair of external speakers.

Shown in Figure 1 (below) are the switches used on the labkit in the final implementation of Maestro, which are connected to each main component of the project. Switches[0:2] allow for the switching of output to the monitor between the graphics and video feed that is being captured by the NTSC camera. Switches[5:7] are the different debugging modes of the video image once switches[0:2] are set to display the NTSC video image. Switches[3] toggles continuous play mode while switch[4] toggles between the option to play with chords or not in sound generation.

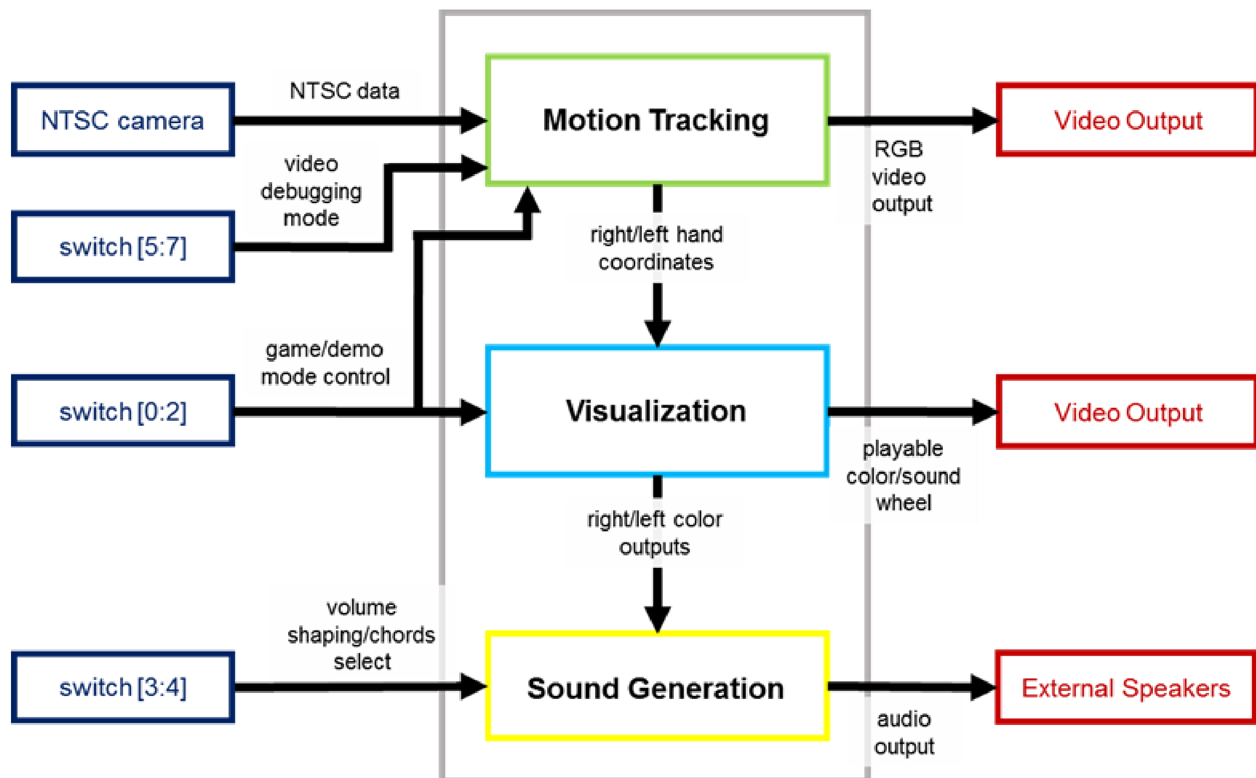

Figure 1. High Level Block Diagram. This diagram shows the high-level flow of the project as well as the external inputs and outputs into and out of the system. The general signals traveling between the main blocks of the system are also shown above. It can be seen that the system is divided into three main parts which have their own external inputs and outputs.

II. Modules

1. Top level Module - zbt_6111_sample.v

The top level module of the project began as the top level module for the sample NTSC video decoding code that was provided by the 6.111 staff. Building off this module for use as the

project's main module made the instantiation of the modules for all three main parts of the system more streamlined, as the IO assignments of the labkit's pins for the video decoding and storing of the NTSC camera was more involved than the IO assignments for sound generation from the ac97 or for visualization - which uses the same instantiation of the XVGA module as motion tracker in order to get the same hcount, vcount, and blank signals.

2. Motion Tracking – Evie

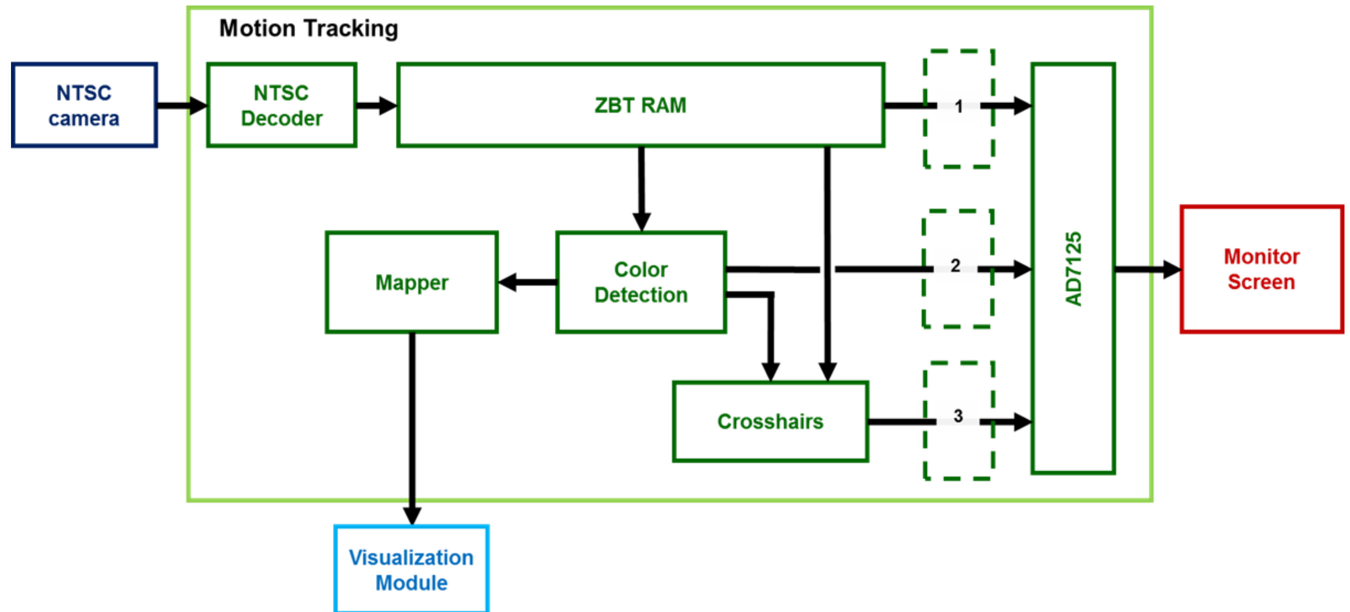

Figure 2. High Level Motion Tracking Diagram. Switches [5:7] on the labkit toggles between the 1, 2, and 3 output depicted in the dotted boxes. 1 is the RGB video feed. 2 is the video feed with the significant pixels shaded. 3 is the video feed with crosshairs drawn at the center of masses of the user's hands.

Motion tracking deals with receiving input from an NTSC camera that is capturing the movements of the user's hands and stores RGB pixel data into the ZBT memory on the FPGA. The pixel data is converted to the HSV color space, and based on threshold color values that were determined experimentally, identifies the significant pixels that correspond to the red and green glove worn by the user. The positions of these significant pixels are averaged and the result produces the XY coordinate of the center of mass for each hand which is then sent off to the visualization block.

Figure 3. NTSC Camera.

2.1 NTSC Decoder

This module is responsible for taking the input NTSC signal from the camera (see Figure 2), decoding, and storing the data in the ZBT RAM on the FPGA board. The module is a modified version of the NTSC camera Verilog that was provided by the 6.111 staff. The original 8-bit black and white YCrCb data was changed to incorporate 18 out of 24-bit RGB information. The ZBT ram stores only one frame at any given time and each frame consists of 1024x768 (786,432) pixels. Each stream of RGB information is stored as 3 bytes (1 for Red, 1 for Green, and 1 for Blue). Two pixels per address in the ZBT RAM are stored which is the reason why the 24-bits of RGB is truncated to 18-bits (6-bits for each color) - so as to accommodate the maximum of 36 bits per address line of storage in the memory.

2.2 ZBT RAM

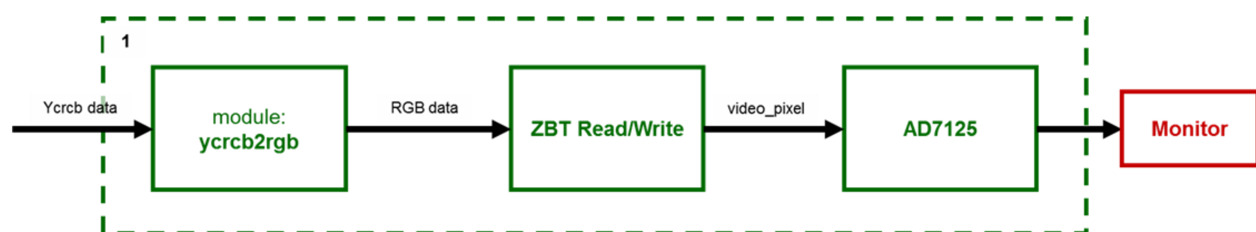

Figure 4. Video Display Mode. The NTSC YCrCb pixel data is converted into RGB data to be stored and read from the ZBT RAM. ZBT Read/Write contains the modified **ntsc2zbt** module that does the writing as well as the given **vram_display** module which does the reading. The AD7125 is what translates video_pixel, the video image, into VGA output.

The modifications to the sample NTSC code[4] happened primarily in the *ntsc2zbt* module which is responsible for the storing of the NTSC camera data. The camera data, which is decoded in

[4] Described in the section 2.1 - NTSC Decoder

the NTSC Decoder section (above), is parsed into YCrCb data. The provided *ycrcb2rgb* module is used to convert the YCrCb pixels into RGB before they are stored in the ZBT RAM using *ntsc2zbt*.

Vram_display (part of the provided code) reads from the ZBT RAM what *ntsc2zbt* stores and outputs the camera image in the form of vr_pixel[23:0]. Before any sort of pixel is outputted to the AD7125, the vr_pixel signal is modified so that the borders of the camera image remain black, becoming the output signal: video_pixel (seen in Figure 5).

The video image, which is about 725 x 505 pixels big, is displayed on a 1024x768 screen as shown in Figure 5 (below). Switch[7] of the labkit toggles the video image on the monitor screen on and off.

Figure 5. Video Image on 1024x768 Display.

2.3 Color Detection

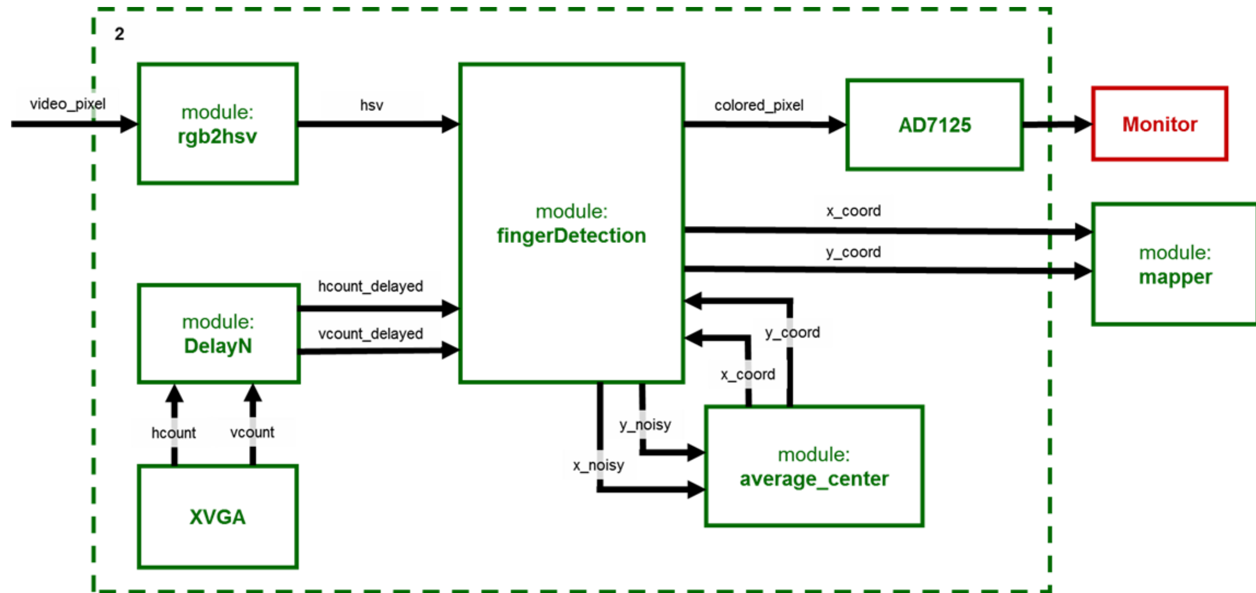

Figure 6. The Diagram of the Color Detection Sub-Part of Motion Tracking.

The detection of the hands is done primarily in the *fingerDetection* module whose task is to take the stream of HSV information and identify pixels that are within the threshold H,S,and V values. The HSV pixel data comes directly from the video_pixel signal (which is the RGB video image) using the provided *rgb2hsv* color conversion module. The video image is converted into HSV color values as to get an accurate color estimation in the poor lighting conditions of the lab. A hue-saturation-value(aka lightness) performs a lot better in the non-ideal environment. The actual conversion from RGB to HSV utilizes two 16x16 Xlinux Coregen Dividers and thus has a delay of about 18 clock cycles to the HSV pixel data.

As a result, the hcount and vcount signals from the *XVGA* module are delayed for 18 clock cycles, by using the *DelayN* module, so that the *fingerDetection* module is reading from the right HSV values per frame at any given time.

The *fingerDetection* module is instantiated once for each color that one wishes to detect. In the scope of our project, this module is instantiated twice, once to look for red, and once to look for green. The finger signal input is what determines the color that is to be identified and the value of this signal goes through an FSM that initiates the thresholds for that specific color. Then, if hcount and vcount are within the bounds of the video display, the module scans the image for the significant pixels that lie within the HSV color thresholds, keeping a count of how many pixels it detected, and accumulates their X and Y coordinates.

For debugging purposes, the pixels identified, known as significant pixels, are highlighted and once the module is finished scanning the video image, *fingerDetection* takes the average of the X and Y coordinates that it accumulates using the Coregen dividers in order to produce a center of mass of the significant pixels. At the beginning of each frame, both the count and accumulator of the pixels are cleared and a new center of mass calculation occurs at the end of each frame.

Figure 7. Shaded Pixel Debugging Mode.

Before sending the X and Y coordinates of the center of mass off to the *mapper* module (described in section 2.5), *average_center* records the four most current XY coordinates and averages them together. This is done so as to reduce the 'jumpy' characteristic of the center of masses. *Average_center* was provided by one of the staff who wrote the code for their 6.111 final project in the past.

The identification of the color thresholds for the red and green glove was done utilizing the alphanumeric hex display on the labkit to help identify the HEX code of the color of one specific pixel that the camera was capturing. White crosshairs were used to point to the place of this specific pixel. By recording several points on each glove, the ranges of the hue, saturation, and value of both the red and green glove were thus determined. In practice, blurring the image of the camera acted as a type of low-pass filter that improved the identification of just the hands by reducing the noise of the random pixels across the video image that would happen to fall within the threshold color ranges as well.

Recorded Pixel Hex Code (Red Glove)	H	S	V
D04830	9	76	81
D84450	355	68	84
803840	353	56	50
882438	348	73	53

Figure 8. Sample Recorded Color Data. In order to streamline the identification of threshold values, the hex codes were read from the hex display on the labkit which showed the color of the pixel at the center of white crosshairs.

2.4 Crosshairs

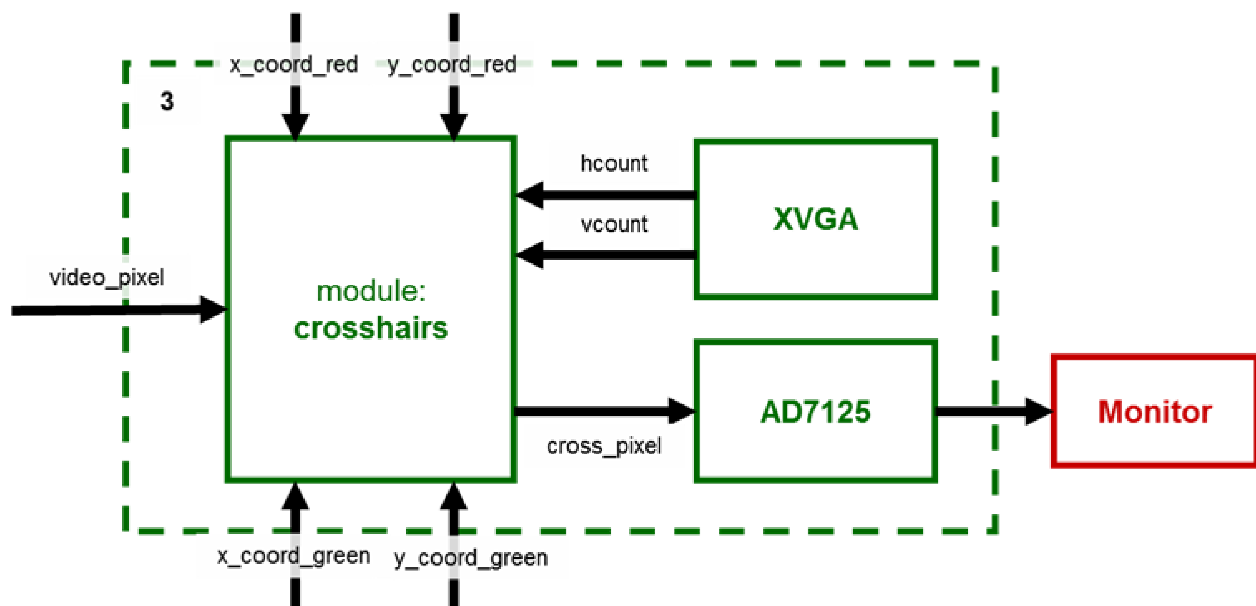

Figure 9. Crosshair Subpart of Motion Tracking.

The *crosshairs* module is the second half of the debugging process of the motion tracking block. It is responsible for outputting the video feed together with crosshairs that allow the developer to locate what the color detection portion of the block is determining as the center of mass of the hand in question.

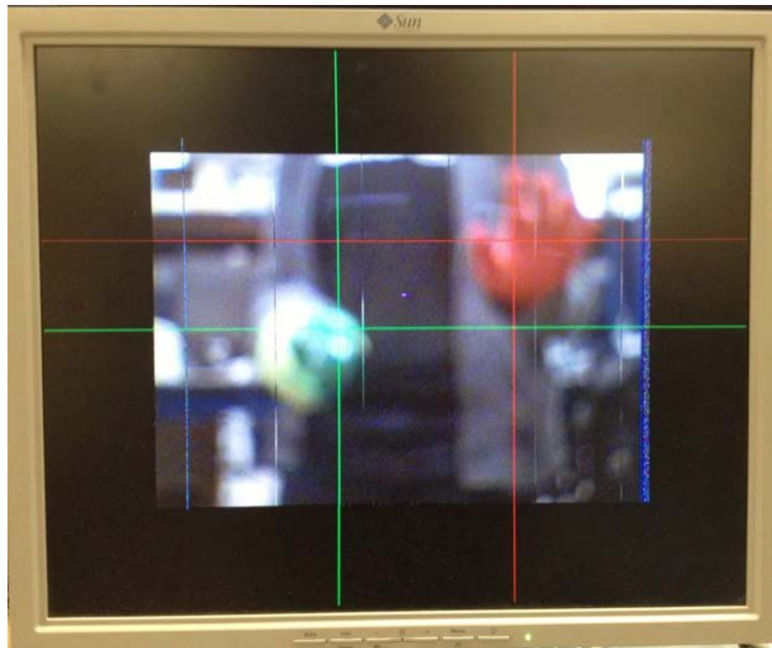

FIgure 10. Crosshair Debugging Mode.

2.5 Mapper

Since the image of the video feed (725x505) is smaller than the size of the whole screen (1024x768), the XY coordinate system of the image needed to be scaled to the XY of the monitor screen. The outer columns and rows of the video image corresponded to two columns and rows of the bigger screen. Moving inwards, the columns and rows get to a point where they map one to one with the columns and rows of the screen, just simply translated. We decided this was the best way to implement, given that the user should have finer control within the bounds of the color wheel. The spaces are wider outside the wheel, and thus allow for a stretching of the coordinate system. (See Figure 11 below)

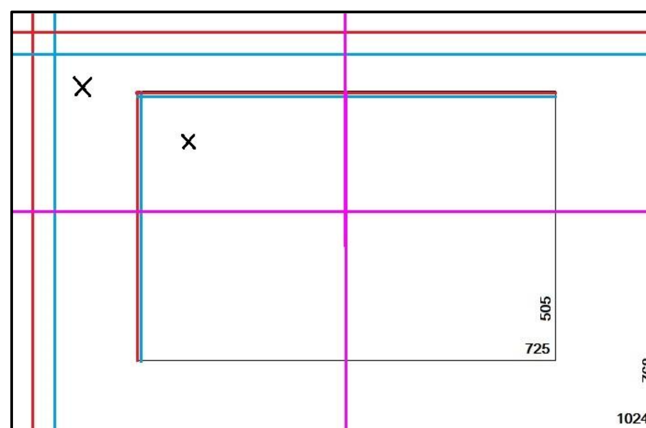

Figure 11. Mapping. The video image is smaller than the whole screen, so an 'X' in the depicted place on the video feed needs to correspond to the 'X' depicted in the larger rectangle that represents the larger screen.

3. Visualization - Janelle

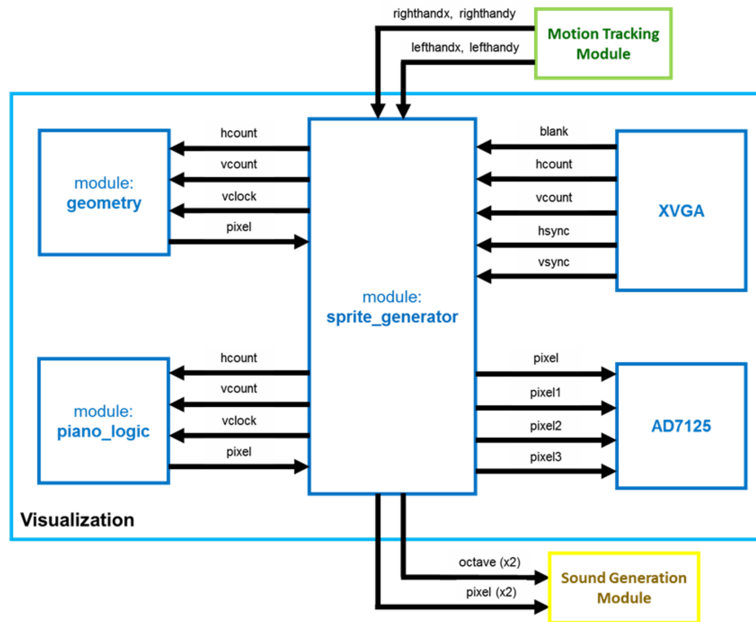

Figure 12: High Level Block Diagram of Visualization System

The visualization module has three main categories of modules to implement the design. The top level module generates sprites, the geometry module supplies the shapes, and the piano logic module uses an algorithm to determine when and where the proper colors are to be shown. Debugging the visuals was mostly done by trial and error. By stepping through each issue bit by bit, bugs could easily be spotted and, normally, resolved. As explained in more thorough detail in the following sections, the design choices were partially restricted by the difficulty working with irrational and/or fractional numbers with FPGA however they still yielded an aesthetically pleasing result that was similar to the goal as shown in Figure 13. Testing and developing happened concurrently and the documented visuals show a clear process of how the final product was achieved.

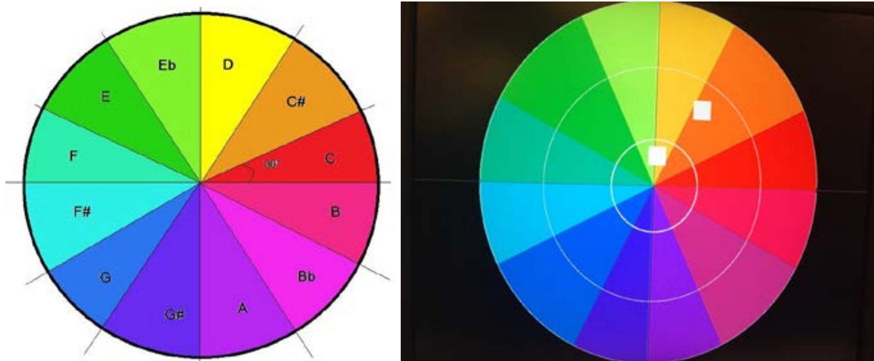

Figure 13. Color and Sound Wheel. As shown through our initial concept on the left and its implementation on the FPGA on the right. The color wheel on the right shows all the colors created in the most saturated color wheel along with an earlier version of the hand sprites (white squares).

3.1 High Level Module - Sprite Generator

The sprite generator module instantiates all the pixels that are displayed on the screen via the AD7125. It also utilizes alpha blending for the hand sprites (shown in Figure 13, Right). This was done using the logic first introduced in the pong game lab (an earlier lab in the class that implemented the basic pong game) and the gave the appearance of black squares when hand sprites overlapped with color wheel segments (see Figure 22).

3.2 Geometry

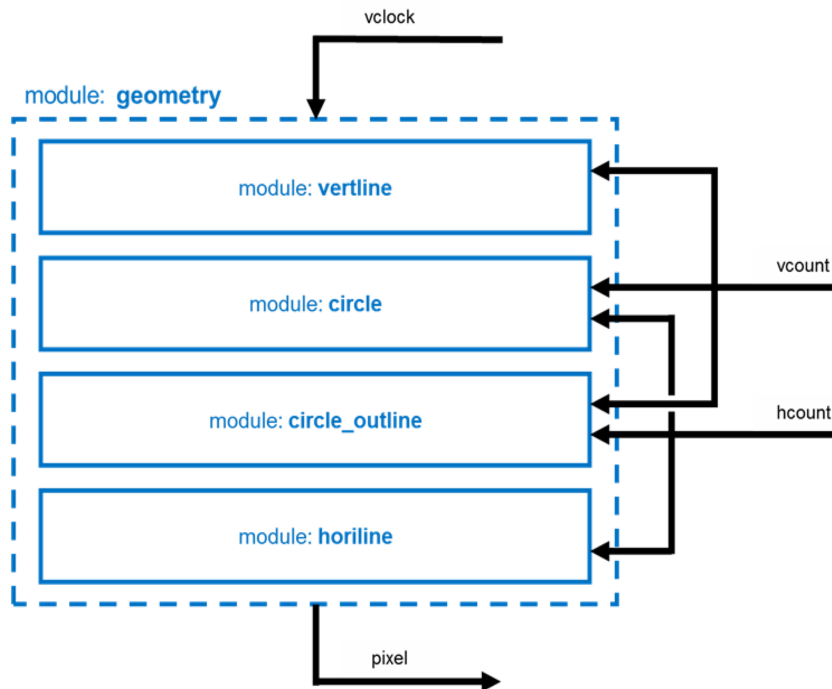

Figure 14: Geometry Block Diagram. The four modules that comprise the shapes and geometry of the visualization appear here.

Creating geometric shapes in Verilog can be challenging depending on the constraints of the system. Since monitors and displays are pixelated, irrational values do not hold the same meaning compared to, say, a MATLAB graph. In order to work around this, sometimes tolerances were implemented to get close representations of the shapes and figures. It was important to note that X and Y values in normal geometric equations would actually act as the hcount and vcount inputs produced by the XVGA.

Circle module[5] creates and fills in circles with a single RGB value by using an inequality in conjunction with the circle equation: $(x + x_0)^2 + (y + y_0)^2 \leq r^2$. This module was useful for understanding how to create all types of circles and was used in earlier development of creating the general shape of the color wheel

Circle outlines (or unfilled circles) were created by using tolerances as discussed above. This worked by comparing two circles -- let's say Circle A and Circle B where $radius_a < radius_b$. Using the center of the screen as the circle origin this tolerance basically locates pixels that have radiuses less than Circle A and greater than Circle B. This allowed for the customization of the "saturation" of the circle ring and was an easy fix for the otherwise non-visible pixelated circle equation.

Horizontal and vertical line dividers used the simple line equations to create the lines that would visually divide the circle into the four Cartesian quadrants.

$$x = screen_center_y_value$$
$$y = screen_center_x_value$$

Figure 15: Non-Filled and Filled Targets. These target-like visuals feature all the components of the geometry modules alone.

[5] Verilog built from Problem 4, HW #8 from earlier in the semester.

3.3 Piano Logic and Color Wheel

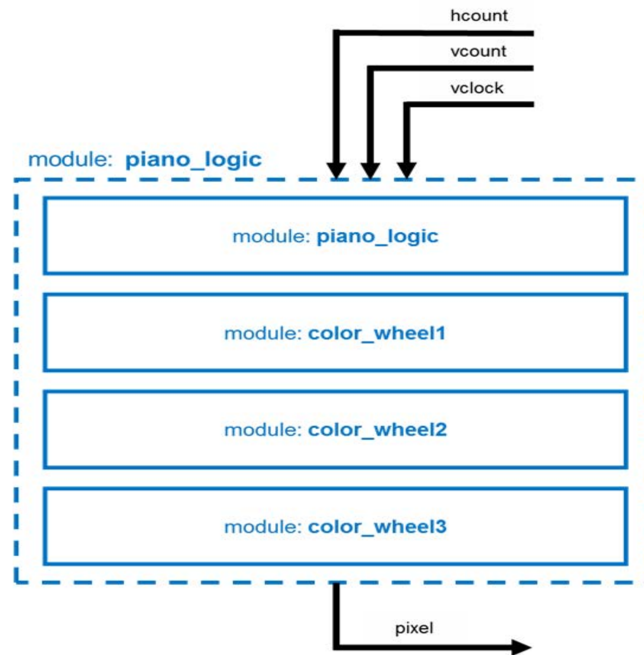

Figure 16: Piano Logic sub-section of Visualization Group

The color wheel modules lay the foundation for the piano logic. These modules take in twelve 24 bit RGB parameters that contain the colors of the wheel as shown in Figure 15. They also take in parameters (partially dependent on the initial parameters to create some of the earlier geometry) to determine the size and gradient of colors used for each. Utilizing partially the code from the circle module, and then a new set of code to display each pie slice of color, the color wheel lays the foundation to the piano logic module.

Creating an algorithm that accurately shows each pie slice on the color wheel is a challenge in itself. At first it seems the most straightforward approach is to use polar coordinates in order to define spaces on the display. The inequalities would have utilized the equation $\tan^{-1}\left(\frac{y}{x}\right) = \theta$ by dividing the pie slices into 30 degree increments. This method could have been easily implemented if Verilog offered a module that took in X and Y values to calculate the inverse tangent in degrees however the only option available outputs values in radians. This is an issue because in order to define spaces with accuracy, careful consideration to decimals would have needed to be taken. Verilog does not allow explicit inputs of fractions (unless the divider is 2^n) and the data sheet for the module supplied had many conditions listed in concern to decimal precision and utilization. So instead of continuing on that route, an alternative was taken.

The main replacement equations (shown on a plot in Figure 17) were those of diagonal lines and worked similarly as to how the circle module functioned; they utilized inequalities to define

spaces. The main drawback to this method was that the pie slices were not evenly spaced in increments of 30 degrees however they still had the desired effect of producing a clear color wheel.

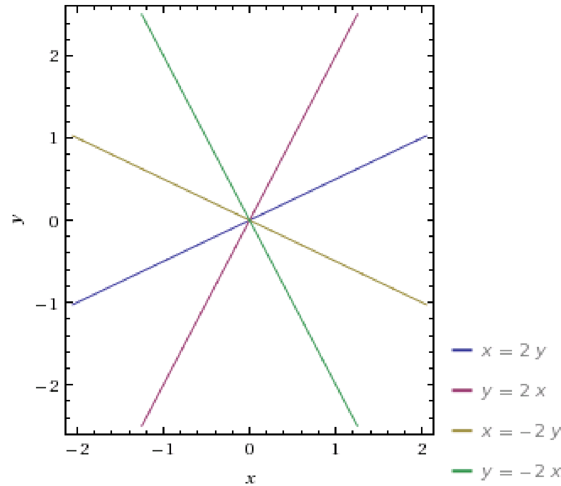

Figure 17. Equations used to describe pie slices of color wheel

The last important factor that went into the creation of the color wheel algorithm came from defining a new coordinate system. Contrary to the standard format of the Cartesian coordinate system, both the hcount (X) and vcount (Y) start in the left top corner of the screen. In order to shift the origin to a more familiar space, the X and Y registers were converted into signed values and adjusted accordingly depending on parameters of the display being used. The next issue arose strictly from the vcount which unconventionally started having an inverted axis. In other words the vcount was negative on the top half of the screen and positive on the lower half. The transformation equations were shown as follows:

$$y_0 = -(y - screen_center_y)$$
$$x_0 = x - screen_center_x$$

To test if the algorithm worked correctly, each quadrant was stepped through one piece at a time. These equations simply checked if the X and Y values supplied were greater or less than 0. Figure 18 shows how early debugging of this step through appeared. Every pixel within the proper radius appeared pink while any pixel in the first quadrant appeared blue.

Figure 18: Debugging through Quadrant step through of color wheel.

The next step to debugging took this process one step further. Beginning in the first quadrant, the first pie slice was assigned a pixel value of blue (for all X and Y greater than 0 and less than the radius of the circle) while the rest of the circle was assigned the color fuchsia as shown in Figure 18. Following that, debugging of an individual pie slice was done by entering arbitrary values for X and Y and testing whether they corresponded to the correct color as shown in Figure 19.

Figure 19. Pie Slices. Shows the individual process that occurred for debugging each pie slice color.

After the debugging was complete, implementing this color wheel for circles of different sizes and colors of different gradients was non-trivial since they all followed the same algorithm. Switches[0:2] from the labkit allows a user to see each separate color wheel as shown in Figure 20.

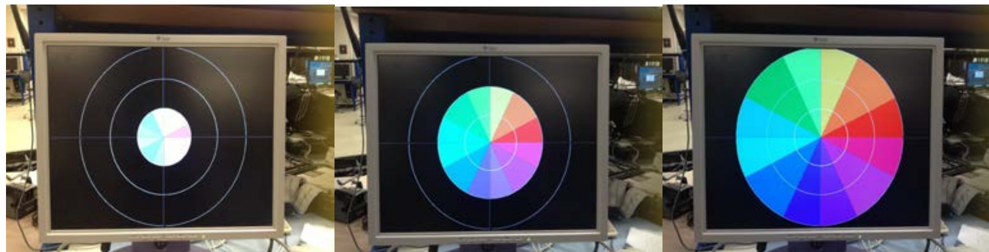

Figure 20. Varying the Color Wheel. All three variations of the color wheel are shown here. The color gradient went from light to dark and the radius went from short to long in the pictures from left to right respectively.

The piano logic module then took the concept of the color wheel and added two more features. Firstly, it now takes into consideration the X and Y positions supplied by the hand tracking modules (not to be confused with the X and Y positions supplied by the hcount and vcount) as well as the ones driven by the hcount and vcount. Secondly, it now outputs an octave -- a 2 bit register that is simply labeled a “zone” of the color wheel by an arbitrarily picked number. This is later fed to the sound generator. Even though the algorithms mirror each other exactly in the piano logic, a bug appeared in the 2nd quadrant that couldn’t really be explained. It created a weird shape in the dark green sections of the color as shown in Figure 21.

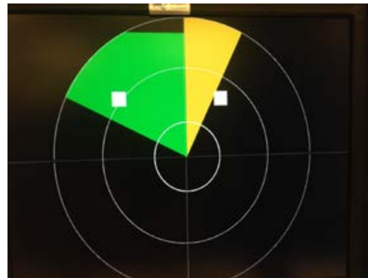

Figure 21. Glitch in Colorwheel.

Other than that, this module successfully used the X and Y inputs of the tracker to highlight the correct section of the color with the square hand sprites discussed earlier and also signaled for a sound of a corresponding color to be played as shown in Figure 22.

Figure 22. Still moment in color piano demonstration. This displays the combined pixels outputted by the piano logic combined with geometry, alpha blending, and tracking modules.

4. Sound Generation – Sabina

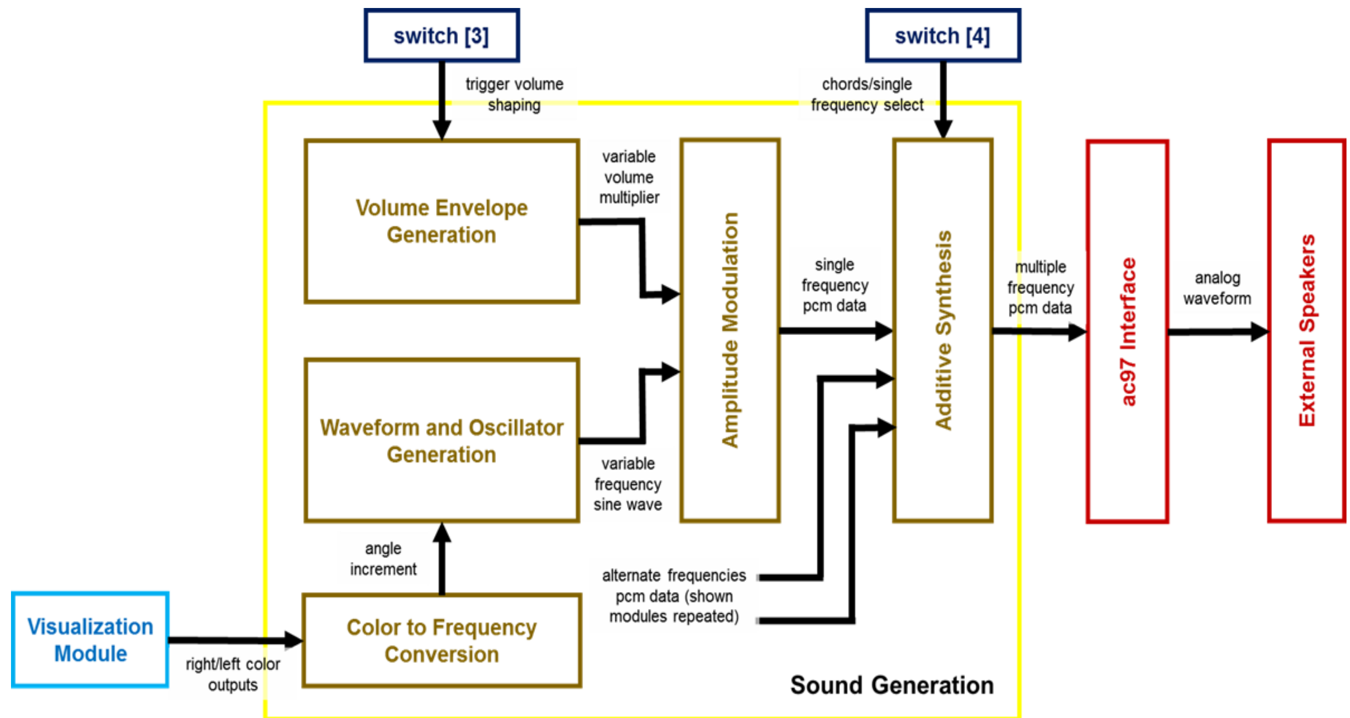

Figure 23. High Level Block Diagram of Sound Generation Module

The sound generation module synthesizes a sine-wave audio output from a 'color' input. This 'color', as well as a respective 'octave' input, indicate the block over which the user is hovering in the color wheel. This input comes from the visualization module. The sound generation module can further manipulate our output sound by using volume shaping and synthesizing additional frequencies for chords or harmonics. These settings are controlled using the spare switches on the labkit (switches [3:4]). The output of the sound generation module is a series of digital time samples that are in turn fed into the AC97, which we interfaced with using code provided to us in Lab 5a.

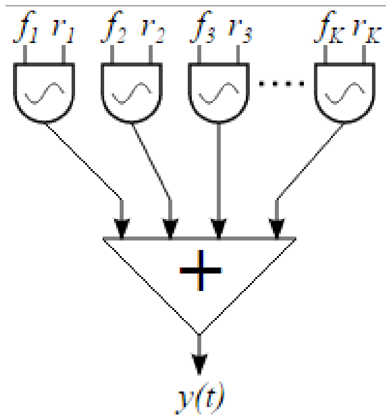

Figure 24. High level concept of additive synthesizers (from Wikimedia Commons)

We chose to model our sound generation module on early additive synthesizers like the teleharmonium: these are synthesizers that generate a variety of sounds by adding together fundamental waveforms, usually sinusoids, but also square, sawtooth, and triangle waves as well. We chose *this* implementation because it implemented our desired baseline audio and could be easily expanded to create chords or Shepard tones, which were sound applications we were interested in implementing if we wanted to expand our project (we ended up implementing chords in the higher level module *s_s_s_synth*). Though we originally planned on using a subtractive synthesis scheme, we didn't think it was necessary to use an FFT to generate only a small set of frequencies for the operation of our instrument.

In order to create our synthesizer (as is the case with any other additive synthesizer), the sound generation modulation was roughly broken into a frequency dependent component (generating the note and octave of our desired sound), an oscillator (generating a sinusoidal waveform) and a volume dependent component (variable volume envelope shaping).

4.1 Generating an oscillator

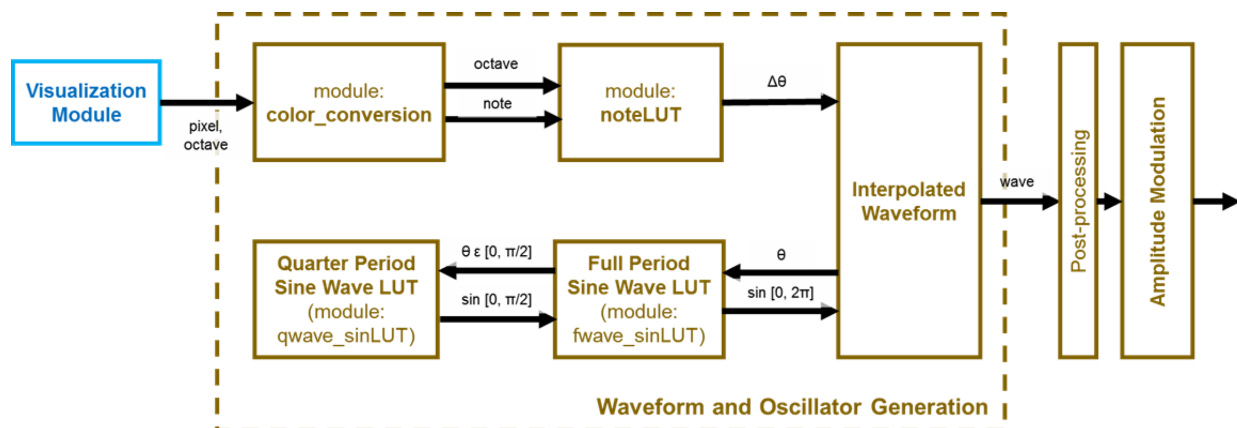

Figure 25. Waveform generation overview

‘

The first step to implementing the sound generation module was to synthesize an oscillator. For our oscillator, we chose to create a sinusoid, which resembles the sound and function of many of the first electronic instruments. Sinusoids are among the more difficult waveforms to implement digitally because they cannot be computed arithmetically-- instead they require a sinusoidal look-up table, in which information about wave amplitude is stored sequentially. The second step is determine how to step through the lookup table-- the faster the frequency is, the faster the oscillator must step through the sinusoid table. In this implementation, all frequencies step through the sinusoidal lookup table at the same sample rate, but instead step through the table by differing phase increments. For this implementation, the increments by which the wave is stepped through is defined by the *noteLUT* module.

Early on in our module, it was unclear exactly how many octaves and which octaves we wanted to implement. Thus in order to cover our bases, the implementation for the sound generation module began by attempting to be able to play as wide as a range of frequencies as possible (from octaves 0 to 7). Since the ac97 samples sounds at a fairly reliable rate (48 kHz), it is easy to determine how differently sized sine wavetables could resolve a range of frequencies. In order to resolve low frequencies (such as those in the zeroth octave), the size of the sine wave lookup table must be exponentially larger, sizing at the very least 2046 memory addresses to resolve the difference between the note C0 and C#0.

In order to limit the size of our sine wave table, *qwave_sinLUT*, to 1024 memory addresses, we used a quarter-wave table to effectively quadruple the number of memory spaces we could return sine values for while still maintaining a smaller memory. Using a quarter wave table required us using a simple finite state machine, *fwave_sinLUT*, to map all angle values to an address and value in the quarter wave table, and then finding a way to properly scale that value to reflect the positive-negative oscillations indicative of sine waves.

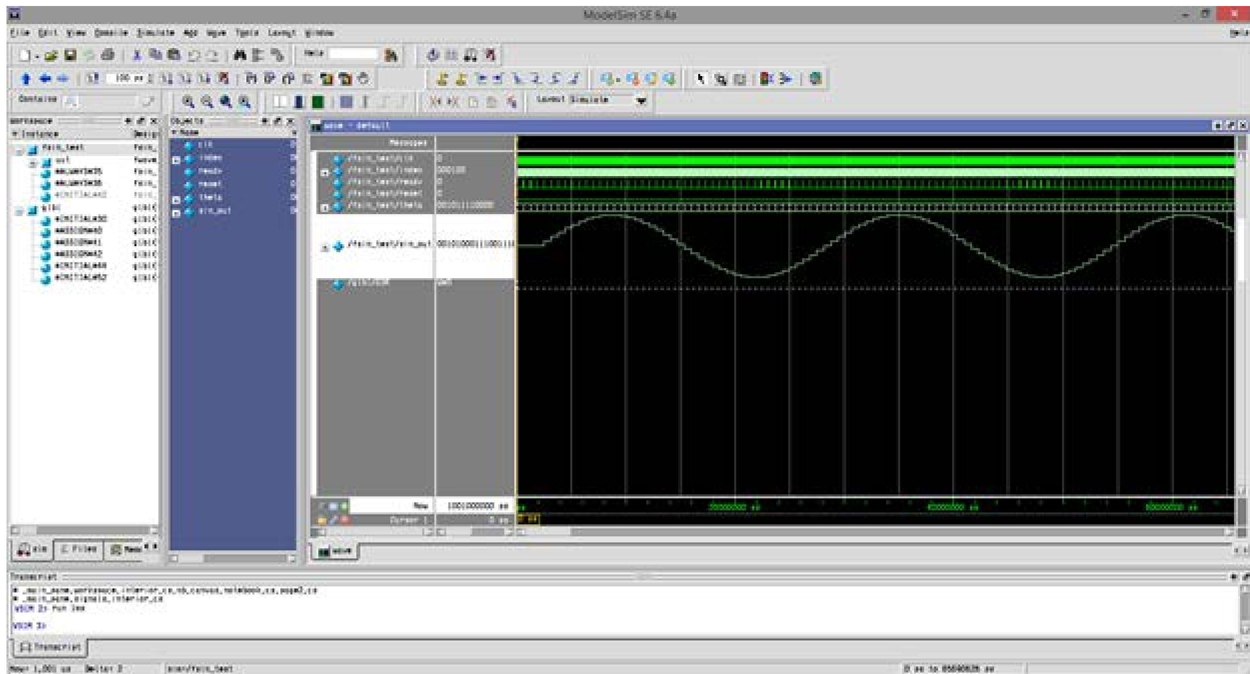

Figure 26. Generating a sinusoid in Model Sim by stepping through the full sine wave FSM

Wave table values were generated using the sin mathematical functions found in MATLAB and Microsoft Excel. Although, it came to our attention once implementing the module that Coregen itself may provide its own lookup table. In the future, it may be a useful on our behalf to test the effectiveness of either or both methods in determining the shape of an oscillator.

Another method we used to help resolve low frequencies was frequency interpolation. Frequency interpolation is in part possible because the angle increment data in the *noteLUT* is stored in floating point. We were able to use linear interpolation in the module *interpolate_wave* to determine the inferred sine value at a given decimal point angle. Because interpolation allows for us to step through the wave table at decimal increments, we were given further flexibility in determining low frequency waves. Linear interpolation can be be accomplished by computing weighted average between the provided integer table values. We implemented with following algorithm in order to achieve linear interpolation.

*interpolated value = low table value *(1 - fractional part of angle increment) + high table valu*fractional part of angle increment*

In practice, we found that the external speakers we used did a poor job themselves in playing the low frequencies in the 0th and 1st octaves, so we did not use these frequencies extensively. Still with our speaker capability, we could generate and listen to frequencies from octaves 2 through 7.

Using interpolation introduced errors in multiplication overflow. Overflow errors can result in generating high frequency noise in addition to the desired signal, as shown below.

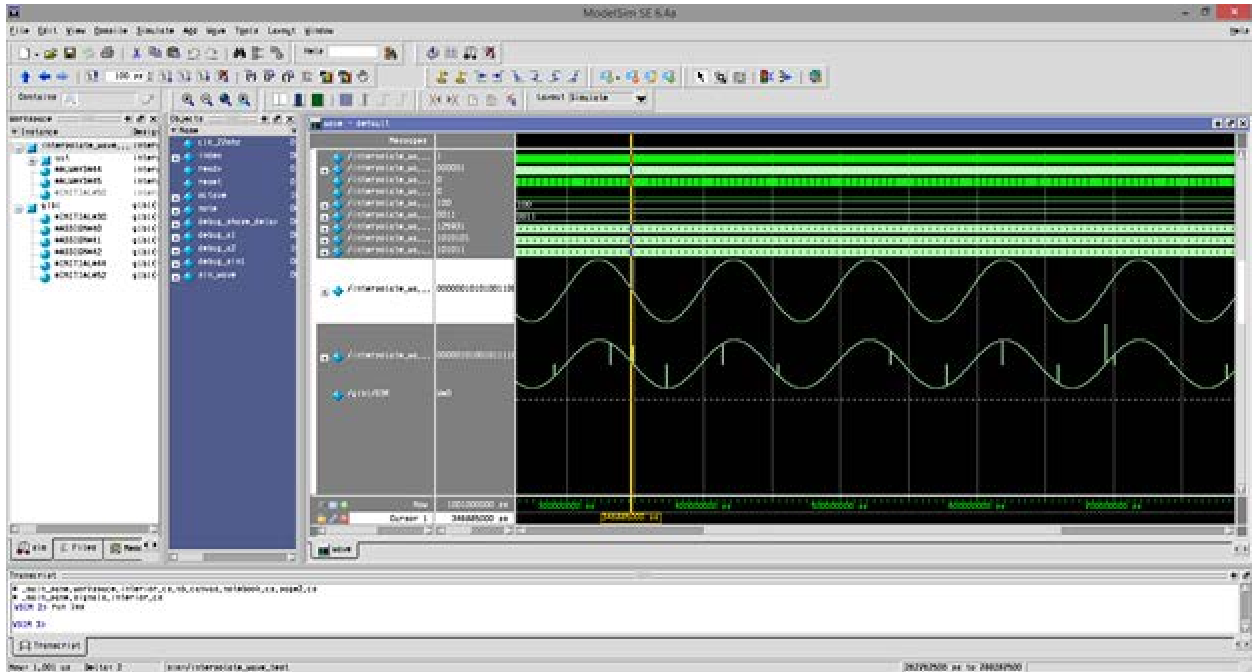

Figure 27. Distortion on simulated waveform due to multiplication overflow

There were also a number of problems with negative values and using sign extensions. By generating a sinusoidal wave centered around zero, it is easy to get caught up in sign extension errors. These sign extension errors can cause the oscillating wave to resemble a square wave. Figure 28 below displays this error both in the simulated output in ModelSim, and in the recorded analog audio output.

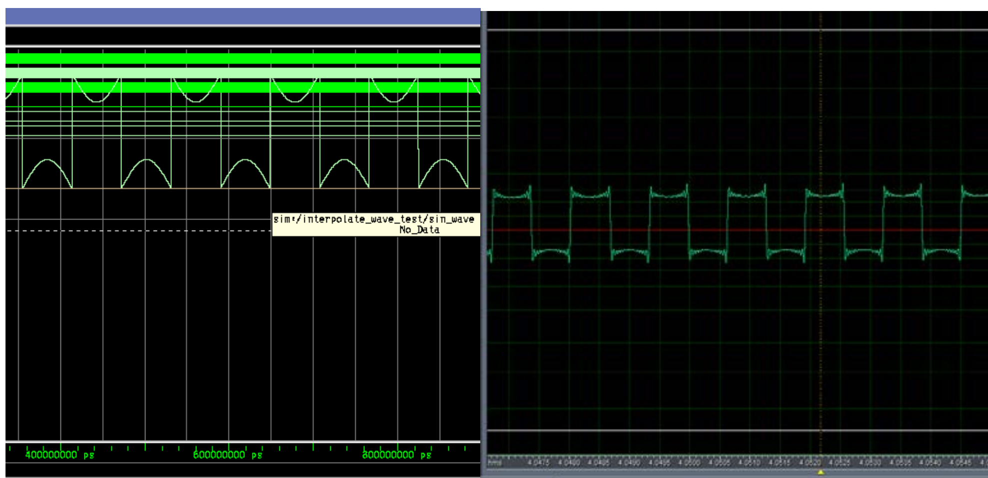

Figure 28. Distortion on simulated and recorded waves due to no sign extension

4.2 Amplitude modulation

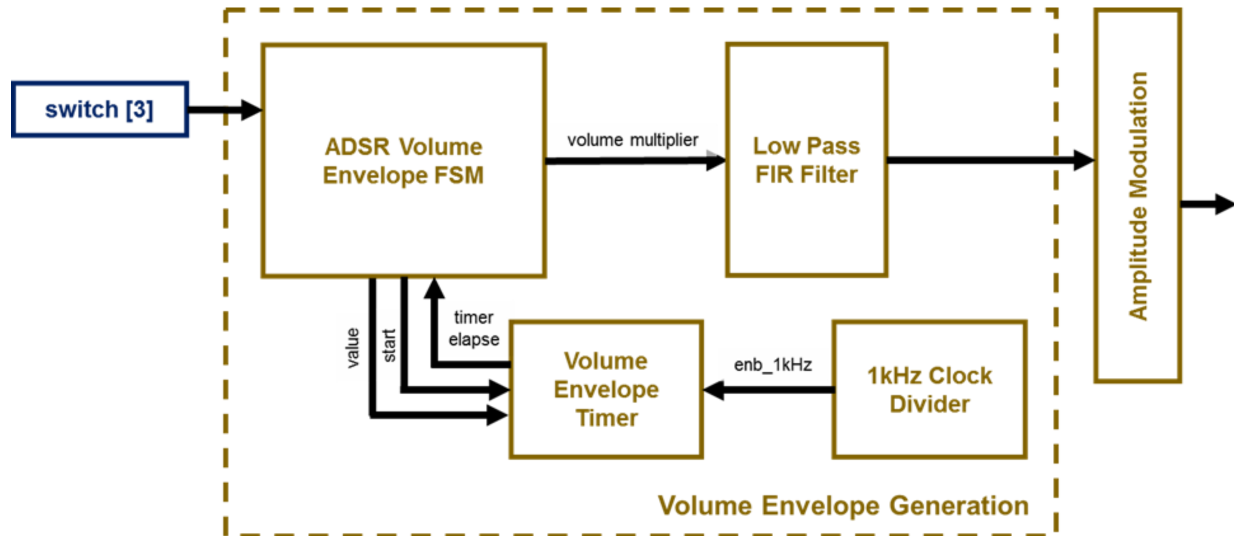

Figure 29. Volume shaping overview

Another important aspect to sound synthesis is through amplitude modulation, which people perceive as volume shaping. One of the most fundamental types of volume envelopes to music synthesis are the Attack-Decay-Sustain-Release volume shaping envelope, which was the primary method of volume shaping employed in the sound generation module.

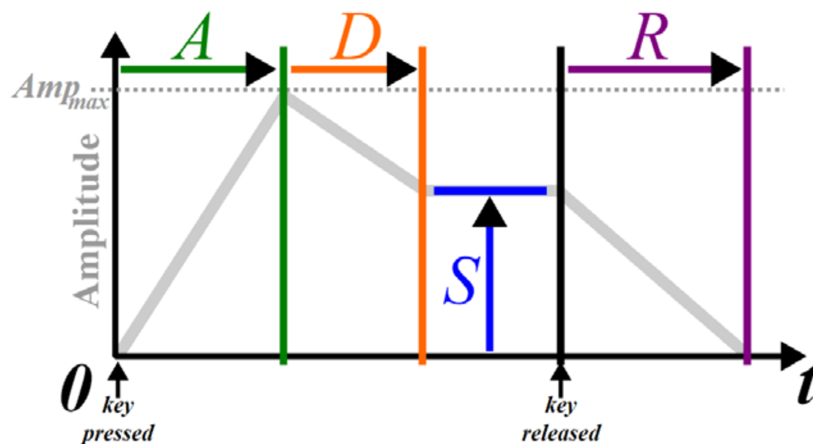

Figure 30. Diagram of common ADSR volume envelope. 'Attack' indicates the initial rise time of a sound, decay indicates the initial decay of the sound to a steady sustain amplitude and is also measured in time. The sustain portion of the envelope maintains a constant amplitude for a length of time until the sound is released (also a function of time) (from Wikimedia Commons)

In order to implement the ADSR volume envelope, we use a simple finite state machine to calculate the multiplier we would use to 'amplify' the sound we generate. Since the ADSR envelope is determine usually as a function of time that is typically defined in millisecond, we created a kHz clock divider and timer that runs on this divider to help time the envelope. The *kHz_divider* and *envelope_timer* modules are very similar to the modules developed for Lab 4 (Car Alarm Lab).

Since the amplitude envelope is evaluated at a sample rate of 1kHz, but the sound samples are evaluated at a sample rate of 48kHz, we passed the ADSR linear volume envelope through a low pass filter to help average the filter and interpolate the amplifying magnitudes every time the AC97 samples for audio data. This helped remove discontinuities from the resulting volume shaped waveform. The low pass filter *fir31* was adapted from a similar implementation done/provided for in Lab 5A.

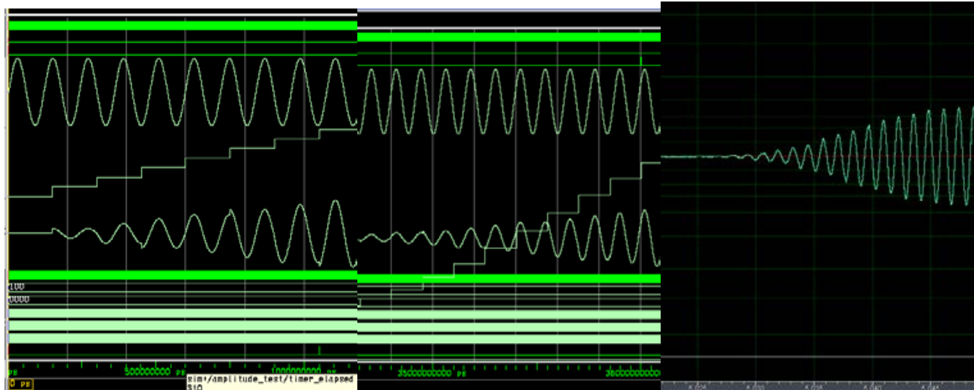

Figure 31. Low pass filtering helps remove some of the discontinuities found (in the leftmost screenshot) during a sharp attack. This can be seen in both ModelSim simulation (center) and actual recordings of our output (right)

We were also able to demonstrate our ability change the shape of the volume envelope based off of changing attack time, decay time, sustain time, sustain amplitude, and release time. However, we chose not making these parameters inputs, instead hardcoding these values, because we did not have enough labkit inputs to do so and we did not believe volume shaping to be a high enough priority use of those inputs in order to demonstrate our final project. Instead we can trigger a fixed volume shaping form every other second, or a fixed amplitude sound on one of the labkit switches, to demonstrate functionality.

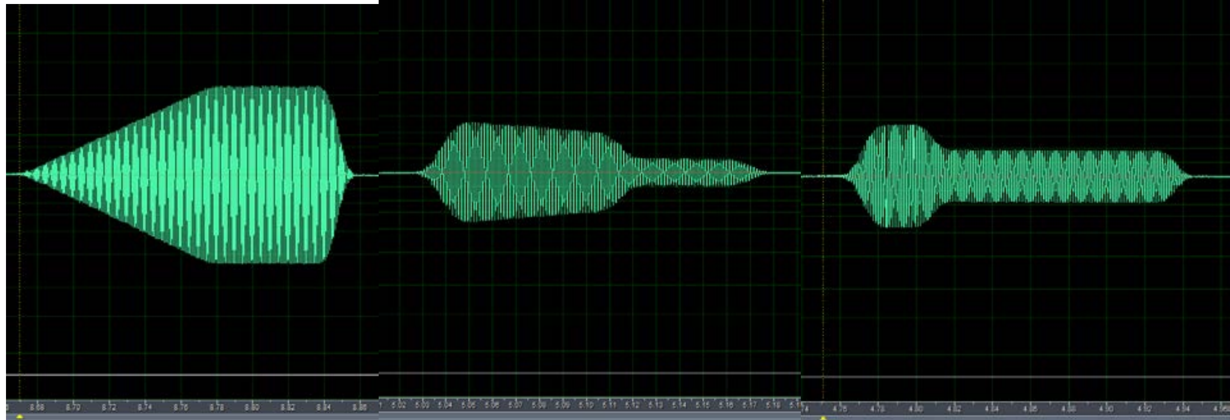

Figure 32. A few of the many types of recorded volume envelopes generated using varying parameters for ADSR

4.3 Post-processing: filtering and dithering

Post-processing involves the functions of filtering and dithering. Filtering, specifically low pass filtering, is used often in post-processing because it helps mitigate the appearance of error that might arise from any number of unresolved errors that could come from multiplication overflow or losing sign extension of a value. We adapted the *fir31* low pass filter developed in Lab 5a to perform this processing on the oscillator output and ADSR volume envelope.

We also frequently worked with sound sample bit depths that far exceeded the 20 bits that could be accepted by the AC97. Rather than simply truncating these samples, we employed dithering; a widely used technique in digital audio is engineering, to reduce bit depth in order to help avoid the effects of quantization and the noise introduced because of it. Quantization happens when digital samples are truncated, because the truncation removes the extra information encoded in the 'lost' bits. This is effective in dividing a number but never having to account for the remainder or consider rounding. Dithering works by adding statistical noise to a signal before its bit depth is reduced. This, in theory, 'fills' or 'smears' in the sine wave so that the new wave amplitude averages its true value, even if the wave can only be represented by integers.

Once the noise has been factored in, the result is trimmed to the desired bit depth. The module *dither* accomplishes this by adding in triangular distribution noise stored in a lookup table. The values of this distribution were generated in MATLAB and are stored in floating point. The module computes addresses to the lookup table using a LFSR (linear feedback shift register) for a pseudo-random address.

We chose to use a triangle distribution for flexibility-- it is considered the best distribution to use if any audio has any possibility of being post-processed once again. We verified this by post-processing sine waves generated in SuperCollider and then dithered with different types of statistical noise in Adobe Audition. The triangle distribution features the least noticeable impact on sound that was further post-processed.

4.4 lab5audio

This module, and its successive sub-modules, was provided to us in Lab 5a in order to interface with the AC97. This module was larger unchanged, accept that it was modified to accept 20 bits of sound input rather than the 8 it had before.

4.5 Sound Generation Module Testing

The majority of testing for the sound module was done using ModelSim simulations. There are several reasons for: for one, many of the modules for this project needed to be tested individually in order to correctly and quickly isolate bugs This is also because the sound generation module is general fairly modular and can be easily expanded to incorporate more features or elements in the future. Luckily once getting the sound generation module onto the FPGA, it was relatively easy to find the remaining bugs (save for the large issue of timing) because most of those bugs were found at the higher up modules. I also found it useful, once getting on the labkit, to record the audio for me to analyze. Doing so made it easy for me to compare waveforms as track the positive (and negative) changes that my edits made to the overall module. Another reason for largely relying in ModelSim simulations was that because I could easily interpret my output using the simulator, running simulations let me iterate my code quicker. I was also less reliant on the labkit. That being said, there is always value in runinng off of the labkit, especially since it gave me the chance to verify my frequencies and debug sign extension issues by listening to the waveform(s) aurally.

III. Testing

Each main block of the project was first developed and debugged independently of each other. Development for all three components were able to occur concurrently as there were only one or two signals at most that would need to be transferred across blocks. The Hex alphanumeric display, buttons, LEDs, and switches were used extensively for debugging values relevant to the operation of the system. More details of how this debugging occurred are detailed within each module section above.

IV. Review

On their own, all of our modules functioned as planned however when integration took place, only the tracking and visualization sections worked together properly. They only needed to be connected in two places, one place for each hand in the *mapper* module. They both ran on the 65 MHz labkit clock, so there were no timing issues that occurred during integration.

Integrating sound was more difficult and we did not complete this integration in time. The piano logic module was feeding the correct pixel information to the color conversion module in the sound generation section however it was doing it much too fast to combat. All of the sound modules run in the 27 Mhz clock domain and the ac97 samples the output time domain sound samples at a rate of 48 kHz. As a way to get the color values from visualization to sound generation, we first attempted to switch all of sound generation to the 65 MHz clock cycle. However, this clock domain ran too fast for the sound calculations and only generated noise in the place of a pure tone. As a workaround, we then attempted to create a module, *color_storage* that would store the value of the color long enough for the sound generation to read it and process it. This method would create lag however it ended up not working as well.

We also thought about implementing an asynchronous FIFO to fix the clocking issue however found it too difficult to create with the limited time we had left. We did learn that Verilog Coregen has asynchronous FIFO modules already available to use and if we had known this before we would have used that instead of the hand-made module we attempted instead.

Lessons Learned

We would recommend paying close attention to the effects of different clocking between modules. If we could have done things differently, we would have tried integrating the sound generation and visualization with each other and also done an integration of the visualization and the motion tracking concurrently. Having these two integrations done in parallel, before bringing the whole project together, could have been better isolated the bugs in the system.

Make sure to dedicate at least two days for integration. (Assuming that all debugging in the separate modules have already been completed).

V. Conclusion

Even though we did not get the project done on time, by continuing on the project afterwards, we have successfully implemented a robust and fun way to enjoy creating music. We are satisfied with the performance of our system, especially after having used it ourselves. The

sound generation is able to produce pure, sinusoidal tones, by themselves or in chords, the hand sprites are able to accurately follow the movements of the user's hands, and the colors of the interactive color wheel light up at the correct times. We enjoyed working on this project so much that we are looking to further its robustness and capability either by ourselves as a personal project, or as a showcase exhibit with a third-party here at MIT!

Appendix I: Verilog Source Code

```
//Top-Level Module
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
// Sample code for the MIT 6.111 labkit demonstrating use of the ZBT
// memories for video display.  Video input from the NTSC digitizer is
// displayed within an XGA 1024x768 window.  One ZBT memory (ram0) is used
// as the video frame buffer, with 8 bits used per pixel (black & white).
//
// Since the ZBT is read once for every four pixels, this frees up time for
// data to be stored to the ZBT during other pixel times.  The NTSC decoder
// runs at 27 MHz, whereas the XGA runs at 65 MHz, so we synchronize
// signals between the two (see ntsc2zbt.v) and let the NTSC data be
// stored to ZBT memory whenever it is available, during cycles when
// pixel reads are not being performed.
//
// We use a very simple ZBT interface, which does not involve any clock
// generation or hiding of the pipelining.  See zbt_6111.v for more info.
//
// switch[7] selects between display of NTSC video and test bars
// switch[6] is used for testing the NTSC decoder
// switch[1] selects between test bar periods; these are stored to ZBT
//           during blanking periods
// switch[0] selects vertical test bars (hardwired; not stored in ZBT)
//
//
// Bug fix: Jonathan P. Mailoa <jpmailoa@mit.edu>
// Date   : 11-May-09
//
// Use ramclock module to deskew clocks;  GPH
// To change display from 1024*787 to 800*600, use clock_40mhz and change
// accordingly. Verilog ntsc2zbt.v will also need changes to change resolution.
//
// Date   : 10-Nov-11
/////////////////////////////
// TOP LEVEL MAESTRO MODULE
/////////////////////////////

////////////////////AUDIO MODULES////////////////////
module lab5audio (
  input wire clock_27mhz,
  input wire reset,
  input wire [4:0] volume,
  output wire [19:0] audio_in_data,
  input wire [19:0] audio_out_data,
  output wire ready,
  output reg audio_reset_b,   // ac97 interface signals
  output wire ac97_sdata_out,
  input wire ac97_sdata_in,
  output wire ac97_synch,
  input wire ac97_bit_clock
);

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

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

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

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

  reg [19:0] out_data;
  always @ (posedge clock_27mhz)
    if (ready) out_data <= audio_out_data;
  assign audio_in_data = left_in_data[19:0];
  assign left_out_data = out_data;
  assign right_out_data = left_out_data;

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

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

```
);
  reg [7:0] bit_count;

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

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

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

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

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

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

    // Latch user data at the end of each frame. This ensures that the
    // first frame after reset will be empty.
    if (bit_count == 255) begin
      l_cmd_addr <= {command_address, 12'h000};
      l_cmd_data <= {command_data, 4'h0};
      l_cmd_v <= command_valid;
      l_left_data <= left_data;
      l_left_v <= left_valid;
      l_right_data <= right_data;
      l_right_v <= right_valid;
    end

    if ((bit_count >= 0) && (bit_count <= 15))
      // Slot 0: Tags
      case (bit_count[3:0])
        4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
        4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
```

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

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

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

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

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

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

  wire [4:0] vol;
```

```
  assign vol = 31-volume;  // convert to attenuation

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

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

///////////////////END AUDIO MODULES///////////////////

module zbt_6111_sample(beep, audio_reset_b,
                    ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock,

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

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

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

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

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

             clock_feedback_out, clock_feedback_in,

             flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
```

```
           flash_reset_b, flash_sts, flash_byte_b,

           rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

           mouse_clock, mouse_data, keyboard_clock, keyboard_data,

           clock_27mhz, clock1, clock2,

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

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

           switch,

           led,

           user1, user2, user3, user4,

           daughtercard,

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

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

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

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

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

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

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

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

input  clock_feedback_in;
output clock_feedback_out;
```

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

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

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

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

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

   inout [43:0] daughtercard;

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

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

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

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

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

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

   // SRAMs

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

/*
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_clk = 1'b0;
   assign ram0_we_b = 1'b1;
   assign ram0_cen_b = 1'b0;     // clock enable
*/

/* enable RAM pins */

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

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

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

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

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

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

   // RS-232 Interface
   assign rs232_txd = 1'b1;
```

```
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

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

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

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

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

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

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

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

```
   //wire clk = clock_65mhz;  // gph 2011-Nov-10

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

   // use FPGA's digital clock manager to produce a
   // 40MHz clock (actually 40.5MHz)
   wire clock_40mhz_unbuf,clock_40mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_40mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 2
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 3
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_40mhz),.I(clock_40mhz_unbuf));

   wire clk = clock_40mhz;
*/
      wire locked;
      //assign clock_feedback_out = 0; // gph 2011-Nov-10

   ramclock rc(.ref_clock(clock_65mhz), .fpga_clock(clk),
                              .ram0_clock(ram0_clk),
                              //.ram1_clock(ram1_clk),   //uncomment if ram1 is used
                              .clock_feedback_in(clock_feedback_in),
                              .clock_feedback_out(clock_feedback_out), .locked(locked));


   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clk), .Q(power_on_reset),
                 .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
   wire reset,user_reset;
   debounce db1(power_on_reset, clk, ~button_enter, user_reset);
   assign reset = user_reset | power_on_reset;

   // display module for debugging

   reg [63:0] dispdata;
   display_16hex hexdisp1(reset, clk, dispdata,
                    disp_blank, disp_clock, disp_rs, disp_ce_b,
                    disp_reset_b, disp_data_out);

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);
/////////////////////////////////BEGIN VIDEO MODULES/////////////////////////////////////
   // wire up to ZBT ram
   wire [35:0] vram_write_data;
   wire [35:0] vram_read_data;
   wire [18:0] vram_addr;
   wire        vram_we;

   wire ram0_clk_not_used;
   zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
                 vram_write_data, vram_read_data,
```

```
                ram0_clk_not_used,   //to get good timing, don't connect ram_clk to zbt_6111
                ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

   // generate pixel value from reading ZBT memory
   wire [17:0]      vr_pixel; //Evie - size changed //7:0
   wire [18:0]      vram_addr1;

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

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

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

   ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                    .tv_in_ycrcb(tv_in_ycrcb[19:10]),
                    .ycrcb(ycrcb), .f(fvh[2]),
                    .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

       //Color Space conversions
       //convert to ycrcb -> RGB
       wire [7:0] R, G, B;
       YCrCb2RGB convert1( .R(R), .G(G), .B(B), .clk(clk), .rst(reset),

       .Y(ycrcb[29:20]), .Cr(ycrcb[19:10]), .Cb(ycrcb[9:0]) );
       //convert RGB -> hsv
       wire [23:0] hsv_data;
       rgb2hsv
hsvDATA(.clock(clk), .reset(reset), .r({vr_pixel[17:12],2'b0}), .g({vr_pixel[11:6],2'b0}),
                  .b({vr_pixel[5:0],2'b0}), .h(hsv_data[23:16]), .s(hsv_data[15:8]), .v(hsv_data[
7:0]));

       // code to write NTSC data to video memory
   wire [18:0] ntsc_addr;
   wire [35:0] ntsc_data;
   wire        ntsc_we;
   ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, {R[7:2],G[7:2],B[7:2]},
                ntsc_addr, ntsc_data, ntsc_we);

   // code to write pattern to ZBT memory
   reg [31:0]       count;
   always @(posedge clk) count <= reset ? 0 : count + 1;
   wire [18:0]      vram_addr2 = count[0+18:0];
   wire [35:0]      vpat = ( switch[1] ? {4{count[3+3:3],4'b0}}
                     : {4{count[3+4:4],4'b0}} );

   // mux selecting read/write to memory based on which write-enable is chosen
   wire      sw_ntsc = ~switch[7];
      wire   my_we = sw_ntsc ? (hcount[0]==1'd1) : blank;
   wire [18:0]      write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
   wire [35:0]      write_data = sw_ntsc ? ntsc_data : vpat;

   assign    vram_addr = my_we ? write_addr : vram_addr1;
   assign    vram_we = my_we;
```

```
   assign    vram_write_data = write_data;
/////////////////////////////////END VIDEO MODULES/////////////////////////////////////
/////////////////////////////////BEGIN HAND DETECTION MODULES/////////////////////////////////////
   // select output pixel data
   reg [23:0]      video_pixel;
   //reg     b,hs,vs;
   //vr_pixel is output of RBG information stored in ZBT
   always @(posedge clk)begin
        if((hcount >= 150 && hcount < 878) && (vcount >= 125 && vcount < 630))begin
             video_pixel <= {vr_pixel[17:12],2'b0,vr_pixel[11:6],2'b0,vr_pixel[5:0],2'b0};
        end
        else begin video_pixel <= 24'd0; end //make borders around camera image black
        //b <= blank; //put video in switch plate
        //hs <= hsync;
        //vs <= vsync;
    end

       //delay hcount  and  vcount  to  account  for  delay  in  hsv  converter
       wire  [10:0]  hcount_delayed;
       wire  [9:0]  vcount_delayed;
       delayN  #(22,11)  delay_h(clk,  hcount,  hcount_delayed);
       delayN  #(22,10)  delay_v(clk,  vcount,  vcount_delayed);


       //use the delayed hcount and vcount now
       //detection of red hand
       wire significant_red;
       wire [10:0] x_coord_red;
       wire [9:0] y_coord_red;
       wire [23:0] colored_pixel_red;
       wire [24:0] x_accum_red;
       wire [24:0] y_accum_red;
       fingerDetection
reddetect(.clk(clk), .finger(3'b001),.hsv(hsv_data), .h_count(hcount_delayed), .v_count(vcount_del
ayed),
       .pixel(video_pixel),.significant(significant_red),.colored_pixel(colored_pixel_red),.x_coor
d(x_coord_red),.y_coord(y_coord_red),.x_accumulator(x_accum_red),.y_accumulator(y_accum_red));

       //detection of green
       wire significant_green;
       wire [10:0] x_coord_green;
       wire [9:0] y_coord_green;
       wire [23:0] colored_pixel_green;
       wire [24:0] x_accum_green;
       wire [24:0] y_accum_green;
       fingerDetection
greendetect(.clk(clk), .finger(3'b100),.hsv(hsv_data), .h_count(hcount_delayed), .v_count(vcount_d
elayed),
       .pixel(video_pixel),.significant(significant_green),.colored_pixel(colored_pixel_green),.x_
coord(x_coord_green),.y_coord(y_coord_green),.x_accumulator(x_accum_green),.y_accumulator(y_accum_
green));

       wire [23:0] cross_pixel;
       crosshairs
Cross(.clk(clk), .hcount(hcount),.vcount(vcount),.x_red(x_coord_red),.y_red(y_coord_red),
                        .x_green(x_coord_green),.y_green(y_coord_green),.pixel(video_pixel),.cross
_pixel(cross_pixel));

       //mapper found in graphics section
/////////////////////////////////END HAND DETECTION MODULES/////////////////////////////////////
/////////////////////////////////BEGIN GRAPHICS MODULES/////////////////////////////////////
```

```
wire [23:0] sprite_pixel;
   wire [23:0] pixel1, pixel2,pixel3; //different variations of color wheel
   wire  [23:0] colors_right, colors_left;
       wire phsync,pvsync,pblank;
       wire [7:0] theta;
       wire [10:0] left_handx;
       wire [9:0] left_handy;
       wire [10:0] right_handx;
       wire [9:0] right_handy;
       wire [2:0] octave, octave1;
       //module that maps center of masses of hands from video to graphics
       mapper
redhand(.clk(clk),.y_in(y_coord_red),.x_in(x_coord_red),.x_new(left_handx),.y_new(left_handy));
       mapper
greenhand(.clk(clk),.y_in(y_coord_green),.x_in(x_coord_green),.x_new(right_handx),.y_new(right_han
dy));

   sprite_generator sg(.vclock(clk),
             .hcount(hcount),.vcount(vcount),.hsync(hsync),.vsync(vsync),.blank(blank),
             .left_handx(left_handx),.left_handy(left_handy),.right_handx(right_handx),
             .right_handy(right_handy),.phsync(phsync),.pvsync(pvsync),.pblank(pblank),
             .pixel(sprite_pixel),.pixel1(pixel1),.pixel2(pixel2),.pixel3(pixel3),

       .octave(octave),.octave1(octave1),.colors_right(colors_right),.colors_left(colors_left));


////////////////////////////////END GRAPHICS MODULES////////////////////////////////////
wire [23:0] finVideo_pixel;
assign finVideo_pixel = (switch[6] && ~switch[5]) ? (colored_pixel_red | colored_pixel_green):
(~switch[6] && switch[5]) ? cross_pixel : video_pixel;

reg [23:0] output_pixel;
   reg b,hs,vs;
   always @(posedge clock_65mhz) begin
      if (switch[2:0] == 3'b001) begin
        // Outer Filled Color Wheel
        hs <= hsync;
        vs <= vsync;
        b <= blank;
        output_pixel <= pixel1;
             end else if (switch[2:0] == 3'b011) begin
       // Middle Filled Color Wheel
        hs <= hsync;
        vs <= vsync;
        b <= blank;
        output_pixel <= pixel2;
      end else if (switch[2:0] == 3'b010) begin
        // Inner Filled color wheel
        hs <= hsync;
        vs <= vsync;
        b <= blank;
        output_pixel <= pixel3;
      end else if(switch[2:0] == 3'b110) begin
    // Theremin
        hs <= phsync;
        vs <= pvsync;
        b <= pblank;
        output_pixel <= sprite_pixel;
      end else if(switch[2:0] == 3'b100) begin
    // Video
        hs <= hsync;
```

```
      vs <= vsync;
      b <= blank;
      output_pixel <= finVideo_pixel; //test
     end else begin
      hs <= phsync;
      vs <= pvsync;
      b <= pblank;
      output_pixel <= pixel3; //test
     end
   end


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

   assign vga_out_red = output_pixel[23:16];
   assign vga_out_green = output_pixel[15:8];
   assign vga_out_blue = output_pixel[7:0];
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_pixel_clock = ~clk;
   assign vga_out_blank_b = ~b;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;
//////////////////////////BEGIN SOUND GEN////////////////////////////////////
wire [19:0] from_ac97_data, to_ac97_data;
  wire ready;

   // allow user to adjust volume
   wire vup,vdown;
   reg old_vup,old_vdown;
   debounce bup(.reset(reset),.clk(clock_27mhz),.noisy(~button_up),.clean(vup));
   debounce bdown(.reset(reset),.clk(clock_27mhz),.noisy(~button_down),.clean(vdown));
   reg [4:0] volume;
   always @ (posedge clock_27mhz) begin
     if (reset) volume <= 5'd8;
     else begin
      if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
      if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
     end
     old_vup <= vup;
     old_vdown <= vdown;
   end

   // AC97 driver
   lab5audio a(clock_27mhz, reset, volume, from_ac97_data, to_ac97_data, ready,
              audio_reset_b, ac97_sdata_out, ac97_sdata_in,
              ac97_synch, ac97_bit_clock);
      wire signed [15:0]pcm_dataRight;
      wire signed [15:0]pcm_dataLeft;
      wire Hz_enable;
      //wire cont_play;
      //wire [2:0]octave;
      //wire [3:0]note;

      s_s_s_synth SYNTHRIGHT
       (.clock(clock_27mhz),
        .reset(reset),
     .ready(ready),
        .play_chord(switch[3]),
     .color(colors_right),
     .cont_play(switch[4]),
```

```
        .usr_pulse(Hz_enable),
     .to_ac97_data(pcm_dataRight));

        s_s_s_synth SYNTHLEFT
       (.clock(clock_27mhz),
        .reset(reset),
     .ready(ready),
        .play_chord(switch[3]),
     .color(colors_left),
     .cont_play(switch[4]),
        .usr_pulse(Hz_enable),
     .to_ac97_data(pcm_dataLeft));

        //make sure data is padded (signed)
             assign to_ac97_data = pcm_dataLeft + pcm_dataRight;

      Hz_divider h(.clk_27mhz(clock_27mhz),
                   .reset(reset),
                                 .enb1kHz(Hz_enable));

////////////////////////////END SOUND GEN/////////////////////////////////////
   // debugging

      //Sabina
      assign led[7]=~to_ac97_data[6];
      assign led[3:0]=~{0,0,0,0};
      assign led[6:4]=~octave;
      reg [23:0] detector_pixel;
   always @(posedge clk)begin
             if(hcount == 363+150 && vcount == 252+125)begin
                   detector_pixel <= video_pixel; //was used for white crosshairs for glove
detecting
             end
     dispdata <= {colors_right[23:20],colors_right[19:16],colors_right[15:12],

      colors_right[11:8],colors_right[7:4],colors_right[3:0],16'b0,
                                                     1'b0,x_coord_green[10:0], 2'b0,
y_coord_green[9:0]};
      end

      //Sabina Logic Analyzer
      assign analyzer1_clock = ac97_bit_clock;
   assign analyzer1_data[0] = audio_reset_b;
   assign analyzer1_data[1] = ac97_sdata_out;
   assign analyzer1_data[2] = ac97_sdata_in;
   assign analyzer1_data[3] = ac97_synch;
   assign analyzer1_data[15:4] = 0;

   assign analyzer3_clock = ready;
   assign analyzer3_data = {from_ac97_data, to_ac97_data};
endmodule

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

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

```
   output    blank;

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

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

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

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

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

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

///////////////////Sprite Generator////////////////////////////////////////////////////
module sprite_generator(
      input [10:0] hcount,
      input [9:0] vcount,
      input hsync,       //horizontal sync
      input vsync,       //vertical sync
      input vclock,

      input [10:0] left_handx,  //x position of green hand
      input [9:0]  left_handy,  //y position of green hand
      input [10:0] right_handx, //x position of red hand
      input [9:0] right_handy,  //y position of red hand
      input blank,



      output phsync,
      output pvsync,
      output pblank,           //blanking
      output [23:0] pixel, //sprites
      output [23:0] pixel1,
      output [23:0] pixel2,
```

```
        output [23:0] pixel3,
        output [23:0] colors_right,
        output [23:0] colors_left,
        output [2:0] octave,
        output [2:0] octave1
     );

        assign pblank = blank; //delay replaces this
        assign phsync = hsync;
    assign pvsync = vsync;

        //Parameters
        parameter SCREEN_CENTER_X = 512;
        parameter SCREEN_CENTER_Y = 384;



              //Color Wheel Geometry

        wire [23:0] vertical_line;
        vertline vline(.hcount(hcount), .pixel(vertical_line));

        wire [23:0] horizontal_line;
        horiline hline(.vcount(vcount), .pixel(horizontal_line));

        wire [23:0] outline;
        circle_outline outline0(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                                  .pixel(outline));

        wire [23:0] outline1;
        circle_outline #(.RADIUS(250), .COLOR(24'hFF_FF_FF),.TOLERANCE(600))
              outline11(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                                  .pixel(outline1));

        wire [23:0] outline2;
        circle_outline #(.RADIUS(125),.COLOR(24'hFF_FF_FF),.TOLERANCE(500))
              outline22(.vclock(vclock),.hcount(hcount),.vcount(vcount),
                                  .pixel(outline2));

              // Colorwheel features

        //wire [23:0] colors_right;
        //wire [2:0] octave;
        piano_logic pl1(.vclock(vclock),.hcount(hcount),.vcount(vcount),
              .handx(right_handx),.handy(right_handy),.pixel(colors_right),.octave(octave));

        //wire [23:0] colors_left;
        //wire [2:0] octave1;
        piano_logic pl2(.vclock(vclock),.hcount(hcount),.vcount(vcount),
              .handx(left_handx),.handy(left_handy),.pixel(colors_left),.octave(octave1));

//Instantiation of Individual Sprites
              // Hand Sprites
        wire [23:0] left_hand;
        hand_sprite left_hand1(.handx(left_handx),.hcount(hcount),.vcount(vcount),
              .handy(left_handy),.pixel(left_hand));

        wire [23:0] right_hand;
        hand_sprite right_hand1(.handx(right_handx),.hcount(hcount),.vcount(vcount),
              .handy(right_handy),.pixel(right_hand));
```

```
	// Filled Color Wheel
		//Outer
	wire [23:0] color_wheel1;
	color_wheel color_wheel11(.hcount(hcount),.vcount(vcount),
		.vclock(vclock),.pixel(color_wheel1));
		//Middle
	wire [23:0] color_wheel_mid;
	color_wheel2 colorwheel22(.hcount(hcount),.vcount(vcount),
		.vclock(vclock),.pixel(color_wheel_mid));
		//Inner
	wire [23:0] color_wheel_inner;
	color_wheel3 colorwheel33(.hcount(hcount),.vcount(vcount),
		.vclock(vclock),.pixel(color_wheel_inner));

		//Nyan Cat - was going to be hand sprite picture
//	wire [23:0] nyan_cat;
//	picture_blob cat1(.vclock(vclock),.x(500),.y(300),.vcount(vcount),
//		.hcount(hcount),.pixel(nyan_cat));


	//ALPHA BLENDING BEGINNING
		//assign integers m and n:
	parameter m = 2;
	parameter n = 4;
	parameter logn = 3;

		//color blending components
	wire [23:16]r_blended = (right_hand[23:16]*m + colors_right[23:16]*(n-m))>>logn;
	wire [15:8]g_blended = (right_hand[15:8]*m + colors_right[15:8]*(n-m))>>logn;
	wire [7:0]b_blended = (right_hand[7:0]*m + colors_right[7:0]*(n-m))>>logn;

		//create new blended pixel
	wire [23:0] blended_pixel={r_blended,g_blended,b_blended};


		//color blending components
	wire [23:16]r_blended1 = (left_hand[23:16]*m + colors_left[23:16]*(n-m))>>logn;
	wire [15:8]g_blended1 = (left_hand[15:8]*m + colors_left[15:8]*(n-m))>>logn;
	wire [7:0]b_blended1 = (left_hand[7:0]*m + colors_left[7:0]*(n-m))>>logn;

		//create new blended pixel
	wire [23:0] blended_pixel1={r_blended,g_blended,b_blended};

//ALPHA BLENDING END

		//Assign output pixels
	assign pixel = (((right_hand!==0) && (colors_right!==0)) ||
		((left_hand!== 0) && (colors_left !== 0)))? //if neither is 0
		(right_hand | blended_pixel) && (left_hand | blended_pixel1) :
			(colors_right | horizontal_line| outline| outline1|
				outline2| vertical_line | right_hand | left_hand| colors_left);

	assign pixel1 = (color_wheel1| horizontal_line| outline1|outline|outline2|
		vertical_line);

	assign pixel2 =(color_wheel_mid| horizontal_line| outline1|outline|outline2|
		vertical_line);

	assign pixel3 =(color_wheel_inner| horizontal_line| outline1|outline|outline2|
		vertical_line);
```

```
endmodule

///////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.
//
// Bug due to memory management will be fixed. The bug happens because
// memory is called based on current hcount & vcount, which will actually
// shows up 2 cycle in the future. Not to mention that these incoming data
// are latched for 2 cycles before they are used. Also remember that the
// ntsc2zbt's addressing protocol has been fixed.

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

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


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

   input reset, clk;
   input [10:0] hcount;
   input [9:0]     vcount;
   output [17:0] vr_pixel; //7:0
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

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

   //wire [18:0]     vram_addr = {1'b0, vcount_f, hcount_f[9:2]};
      wire [18:0]   vram_addr = {vcount_f, ~hcount_f[9:1]};


   wire                 hc4 = hcount[0];
   reg [17:0]       vr_pixel; //7:0
```

```
   reg [35:0]       vr_data_latched;
   reg [35:0]       last_vr_data;

   always @(posedge clk)
        last_vr_data <= (hc4==1'd1) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
        vr_data_latched <= (hc4==1'd0) ? vram_read_data : vr_data_latched;

   always @(*)
                                     //each 36-bit word from RAM is decoded to 2 bytes
     case (hc4)
             1'd1: vr_pixel = last_vr_data[17:0];
         1'd0: vr_pixel = last_vr_data[35:18];
     endcase

endmodule // vram_display

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

module delayN#(parameter NDELAY = 3, parameter SIZE = 10)(clk,in,out);
   input clk;
   input [SIZE-1:0] in;
   output [SIZE-1:0] out;

   //parameter NDELAY = 3;

   reg [SIZE-1:0] shiftreg [NDELAY:0];
   reg [4:0] i;

   always @(posedge clk)begin
             shiftreg[0] <= in;
             for(i=1;i<NDELAY+1;i=i+1)begin
                   shiftreg[i] <= shiftreg[i-1];
             end
    end

      assign out = shiftreg[NDELAY];
endmodule // delayN

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

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ZBT RAM clock generation
//
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// This module generates deskewed clocks for driving the ZBT SRAMs and FPGA
// registers. A special feedback trace on the labkit PCB (which is length
// matched to the RAM traces) is used to adjust the RAM clock phase so that
// rising clock edges reach the RAMs at exactly the same time as rising clock
// edges reach the registers in the FPGA.
//
// The RAM clock signals are driven by DDR output buffers, which further
```

```
// ensures that the clock-to-pad delay is the same for the RAM clocks as it is
// for any other registered RAM signal.
//
// When the FPGA is configured, the DCMs are enabled before the chip-level I/O
// drivers are released from tristate. It is therefore necessary to
// artificially hold the DCMs in reset for a few cycles after configuration.
// This is done using a 16-bit shift register. When the DCMs have locked, the
// <lock> output of this mnodule will go high. Until the DCMs are locked, the
// ouput clock timings are not guaranteed, so any logic driven by the
// <fpga_clock> should probably be held inreset until <locked> is high.
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

	assign ref_clk = ref_clock;

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

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

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

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

   DCM ext_dcm (.CLKFB(fb_clk),
		    .CLKIN(ref_clk),
		    .RST(dcm_reset),
		    .CLK0(ram_clk),
		    .LOCKED(lock2));
   // synthesis attribute DLL_FREQUENCY_MODE of ext_dcm is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of ext_dcm is "TRUE"
   // synthesis attribute STARTUP_WAIT of ext_dcm is "FALSE"
   // synthesis attribute DFS_FREQUENCY_MODE of ext_dcm is "LOW"
   // synthesis attribute CLK_FEEDBACK of ext_dcm  is "1X"
```

```
   // synthesis attribute CLKOUT_PHASE_SHIFT of ext_dcm is "NONE"
   // synthesis attribute PHASE_SHIFT of ext_dcm is 0

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


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

   assign locked = lock1 && lock2;

endmodule

///end of top level module

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

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

   input      clk;  // system clock
   input      vclk; // video clock from camera
```

```
   input [2:0]      fvh;
   input          dv;
   input [17:0]     din;  //changed from 7:0 //this is ycrcb data
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output     ntsc_we;    // write enable for NTSC data
   //input    sw;         // switch which determines mode (for debugging) //not used in final
implementation


      //around center of screen
      parameter     COL_START = 10'd150;
    parameter       ROW_START = 10'd60;



   // here put the luminance data from the ntsc decoder into the ram
   // this is for 1024 * 768 XGA display

   reg [9:0]  col = 0;
   reg [9:0]  row = 0;
   reg [17:0]       vdata = 0;   // 7:0
   reg              vwe;
   reg              old_dv;
   reg              old_frame;  // frames are even / odd interlaced
   reg              even_odd;   // decode interlaced frame to this wire

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

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

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

   // synchronize with system clock

   reg [9:0] x[1:0],y[1:0];
   reg [17:0] data[1:0];   //changed from [7:0] data[1:0]
   reg       we[1:0];
   reg            eo[1:0];

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

```
  end

// edge detection on write enable signal

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

//// shift each set of four bytes into a large register for the ZBT
    // shift each set of two bytes into a large register for the ZBT
reg [35:0] mydata; //changed from [31:0] - now allows for 2, 18-bit bytes
always @(posedge clk)
  if (we_edge)
           mydata <= { mydata[17:0], data[1] };

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


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

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

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

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

//wire [18:0] myaddr = {1'b0, y_addr[8:0], eo_delay[3], x_addr[9:2]};
    //wire [18:0] myaddr = {y_delay[38:30], eo_delay[3], x_delay[39:31]};
    wire [18:0] myaddr = { y_addr[8:0], eo_delay[3], x_addr[9:1]};


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

```
   // alternate (256x192) image data and address
   //wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};
      wire [35:0] mydata2 = {data[1],data[1]};
   wire [18:0] myaddr2 = {y_delay[38:30], eo_delay[3], x_delay[37:29]};

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

   reg [18:0] ntsc_addr;
   reg [35:0] ntsc_data;
   //wire       ntsc_we = sw ? we_edge : (we_edge & (x_delay[30]==1'b0));
      wire       ntsc_we = (we_edge & (x_delay[30]==1'b0));

   always @(posedge clk)
     if ( ntsc_we )
       begin
         //ntsc_addr <= sw ? myaddr2 : myaddr;// normal and expanded modes
         //ntsc_data <= sw ? {mydata2} : mydata;
         //took away expanded mode
         ntsc_addr <= myaddr;
         ntsc_data <= mydata;
       end

endmodule // ntsc_to_zbt




////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:        Evie Kyritsis
//
// Create Date:    16:46:54 12/06/2014
// Design Name:
// Module Name:    fingerDetection
// Project Name:   Maestro
// Target Devices:
// Tool versions:
// Description:    Detects the color specified by the FSM and finger signal.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module fingerDetection(
             input clk,
             input [2:0] finger, //state of which color to tract, FSM below
             input [23:0] hsv,
             input [10:0] h_count,
             input [9:0]  v_count,
             input [23:0] pixel,
             output reg significant, //used for debugging
             output reg [23:0] colored_pixel,
             output [10:0] x_coord,
             output [9:0]  y_coord,
             output  reg  [24:0]  x_accumulator,
```

```
        output  reg  [24:0]  y_accumulator
                             );

        //Red HSV Bounds
        parameter RED_H_MAX = 359; //350-359
        parameter RED_H_MIN = 0;  //0-20
        parameter RED_S_MIN = 100; //160 //60
        parameter RED_V_MIN = 95; //100 //60

        //Green HSV Bounds
        parameter GREEN_H_MAX = 120;//   //120
        parameter GREEN_H_MIN = 50; //   //50
        parameter GREEN_S_MIN = 20;//    //100
        parameter GREEN_V_MIN = 100; //  //60

        //Frame Bounds - Camera Image 725x505
        parameter [9:0]  UP    = 10'd125;
        parameter [9:0]  DOWN  = 10'd630;
        parameter [10:0] LEFT  = 11'd150;
        parameter [10:0] RIGHT = 11'd875;

        reg [7:0] H_MAX;
        reg [7:0] H_MIN;
        reg [7:0] S_MIN;
        reg [7:0] V_MIN;

        parameter RED = 1;
        parameter GREEN = 4;
        always @(*)begin
              case(finger)
                    RED: begin   H_MAX = RED_H_MAX; H_MIN = RED_H_MIN; S_MIN = RED_S_MIN;
V_MIN = RED_V_MIN; end
                    GREEN: begin H_MAX = GREEN_H_MAX; H_MIN = GREEN_H_MIN; S_MIN =
GREEN_S_MIN; V_MIN = GREEN_V_MIN;end
              endcase
        end


        //Set Up averaging values for calculation of Center of Mass of Color Detected
        reg  [24:0]  count_x  =  0;
        reg  [24:0]  count_y  =  0;
        reg  [24:0]  dividend_x  =  0;
        reg  [24:0]  dividend_y  =  0;
        reg  [24:0]  divisor_x  =  0;
        reg  [24:0]  divisor_y  =  0;
        wire  [24:0]  x_quotient;
        wire  [24:0]  y_quotient;
        wire  [24:0]  x_remainder;
        wire  [24:0]  y_remainder;
        wire x_ready, y_ready;


        //Xlinux Coregen Dividers
        mass_divider  x_div(
                                          .clk(clk),
                                          .dividend(dividend_x),
                                          .divisor(divisor_x),
                                          .quotient(x_quotient),
                                          .fractional(x_remainder),
                                          .rfd(x_ready)
                                    );
```

```
            mass_divider  y_div(
                                                      .clk(clk),
                                                      .dividend(dividend_y),
                                                      .divisor(divisor_y),
                                                      .quotient(y_quotient),
                                                      .fractional(y_remainder),
                                                      .rfd(y_ready)
                                                );


          always @(posedge clk)begin
                  //reset values at beginning of new frame
                  if(h_count==11'd0  &&  v_count==10'd0)begin
                        x_accumulator  <=  0;
                        y_accumulator  <=  0;
                        count_x  <=  0;
                        count_y  <=  0;
                        colored_pixel  <=  pixel;
                  end

                  //detect  colors  if  in  frame
                  else  if(h_count>=LEFT && h_count<=RIGHT && v_count>= UP &&
v_count<=DOWN)begin

                        //detect red
                        if(finger==1 && ((hsv[23:16] > H_MIN && hsv[23:16] <= 10) ||
(hsv[23:16]<H_MAX && hsv[23:16]>350)) && hsv[15:8] > S_MIN && hsv[7:0] > V_MIN)begin
                              significant <= 1;
                              colored_pixel[23:16]  <=  8'd255; //shade bright red
                              colored_pixel[15:0]  <=  16'd0;
                              x_accumulator  <=  x_accumulator  +  h_count;
                              y_accumulator  <=  y_accumulator  +  v_count;
                              count_x  <=  count_x  +  1;
                              count_y  <=  count_y  +  1;
                        end //red

                        //detect green
                        else if(finger==4 && (hsv[23:16] >= H_MIN && hsv[23:16] < H_MAX) &&
hsv[15:8] > S_MIN && hsv[7:0] > V_MIN) begin
                              significant <= 1;
                              colored_pixel[23:16]  <=  8'd0; //shade green
                              colored_pixel[15:8]  <=  8'd255;
                              colored_pixel[7:0]  <=  8'd0;
                              x_accumulator  <=  x_accumulator  +  h_count;
                              y_accumulator  <=  y_accumulator  +  v_count;
                              count_x  <=  count_x  +  1;
                              count_y  <=  count_y  +  1;
                        end //green


                        else begin
                              significant <= 0;
                              colored_pixel <= pixel;
                              x_accumulator <= x_accumulator;
                              y_accumulator <= y_accumulator;
                              count_x <= count_x;
                              count_y <= count_y;
                              divisor_x <= divisor_x;
                              divisor_y <= divisor_y;
                              dividend_x <= dividend_x;
                              dividend_y <= dividend_y;
```

```
                          end

                    end //end of detect colors

                    //start divisors
                    else  if  (h_count == 11'd0 && v_count == 10'd630)begin

                          colored_pixel <=  pixel;
                          dividend_x <= (count_x > 50) ? x_accumulator : 0;
                          dividend_y <= (count_y > 50) ? y_accumulator : 0;
                          divisor_x  <= (count_x == 0) ? 1 : count_x;
                   divisor_y  <= (count_y == 0) ? 1 : count_y;

                    end

                    else begin //catch
                          significant <= 0;
                          colored_pixel <= pixel;
                    end

              end//end of always block

              wire [10:0] x_noisy;
              wire [9:0]  y_noisy;

              assign x_noisy = x_ready ? x_quotient[10:0] : x_noisy;
              assign y_noisy = y_ready ? y_quotient[9:0] : y_noisy;

              wire value;
              assign value = (h_count==11'd1 && v_count==10'd630);

              //reduce noise of shaded pixels
              average_center avg(clk,value,x_noisy,y_noisy,x_coord,y_coord);

              //x_coord, y_coord go to mapper.v

Endmodule //fingerdetection


`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Luis Fernandez - TA
//
// Create Date:    14:22:38 12/06/2014
// Design Name:
// Module Name:    average_center
// Project Name:
// Target Devices:
// Tool versions:
// Description:   Takes the last four recorded positions of x,y from fingerDetection and averages
them
//                so as to reduce the 'jumpiness' of the shaded pixels.
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module average_center(
```

```
		input clk,
		input value,
		input [10:0] x_noisy,
		input [9:0] y_noisy,
		output  [10:0] x_coord,
		output  [9:0] y_coord
	 );

	wire [12:0] x_sum;
	wire [11:0] y_sum;
	reg [10:0] buffer_x [3:0];
	reg [9:0] buffer_y [3:0];
	reg [1:0] offset;

	always @(posedge clk) begin
		if(value)begin
			offset <= offset +1;
			buffer_x[offset] <= x_noisy;
			buffer_y[offset] <= y_noisy;
		end
	end

	assign x_sum = buffer_x[0] + buffer_x[1] + buffer_x[2] + buffer_x[3];
	assign y_sum = buffer_y[0] + buffer_y[1] + buffer_y[2] + buffer_y[3];

	//divide by 4
	assign x_coord = x_sum >> 2;
	assign y_coord = y_sum >> 2;

endmodule //average_center

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:       Evie Kyritsis
//
// Create Date:    10:07:22 12/08/2014
// Design Name:
// Module Name:    mapper
// Project Name:
// Target Devices:
// Tool versions:
// Description: This module maps the x,y positions of the hands of the user on the video image
//              (about 725x505) to an x,y position on the larger, 1024x768 graphics screen.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module mapper(
			input clk,
			input [10:0] x_in,
			input [9:0]  y_in,
			output reg [10:0] x_new,
			output reg [9:0] y_new
    );

	 reg [9:0] y_coord;
```

```
	reg [10:0] x_coord;



//hcount - COL Mapping
always @(posedge clk)begin

		y_coord <= y_in - 10'd125; //shift image coord to be in 0,0 reference frame

		if(y_coord == 10'd0)begin y_new <= 10'd0;end
		else if (y_coord>=10'd1 && y_coord<=10'd28)begin
			y_new <= (y_coord << 1);
		end
		else if(y_coord>=10'd29&&y_coord<=10'd56)begin
			y_new <= (y_coord << 1) - 10'd2;
		end
		else if(y_coord>=10'd57&&y_coord<=10'd84)begin
			y_new <= (y_coord << 1) - 10'd3;
		end
		else if(y_coord>=10'd85&&y_coord<=10'd112)begin
			y_new <= (y_coord << 1) - 10'd4;
		end
		else if(y_coord>=10'd113&&y_coord<=10'd400)begin
			y_new <= y_coord + 10'd108 ; //
		end
		else if(y_coord>=10'd401&&y_coord<=10'd428)begin
			y_new <= ((y_coord - 10'd400) << 1) + 10'd507 ;
		end
		else if(y_coord>=10'd429&&y_coord<=10'd456)begin
			y_new <= ((y_coord - 10'd400) << 1) + 10'd506 ;
		end
		else if(y_coord>=10'd457&&y_coord<=10'd484)begin
			y_new <= ((y_coord - 10'd400) << 1) + 10'd505 ;
		end
		else if(y_coord>=10'd485&&y_coord<=10'd512)begin
			y_new <= ((y_coord - 10'd400) << 1) + 10'd504 ;
		end

end

//vcount - ROW Mapping
always @(posedge clk)begin

		x_coord <= x_in - 10'd150; //shift image coord to be in 0,0 reference frame

		if(x_coord>=10'd0 && x_coord<=10'd38)begin
			x_new <= x_coord << 1;
		end
		else if(x_coord>=10'd39 && x_coord<=10'd77)begin
			x_new <= (x_coord << 1) - 10'd1;
		end
		else if(x_coord>=10'd78 && x_coord<=10'd116)begin
			x_new <= (x_coord << 1) - 10'd2;
		end
		else if(x_coord>=10'd117 && x_coord<=10'd155)begin
			x_new <= (x_coord << 1) - 10'd3;
		end
		else if(x_coord>=10'd156 && x_coord<=10'd360)begin
			x_new <= x_coord + 10'd152;
		end
		else if(x_coord>=10'd360 && x_coord<=10'd367)begin
```

```
			x_new <= 10'd512;
		end
		else if(x_coord>=10'd368 && x_coord<=10'd572)begin
			x_new <= x_coord + 10'd145;
		end
		else if(x_coord>=10'd573 && x_coord<=10'd611)begin
			x_new <= ((x_coord - 10'd500) << 1) + 10'd572;
		end
		else if(x_coord>=10'd612 && x_coord<=10'd650)begin
			x_new <= ((x_coord - 10'd500) << 1) + 10'd571;
		end
		else if(x_coord>=10'd651 && x_coord<=10'd689)begin
			x_new <= ((x_coord - 10'd500) << 1) + 10'd570;
		end
		else if(x_coord>=10'd690 && x_coord<=10'd727)begin
			x_new <= ((x_coord - 10'd500) << 1) + 10'd569;
		end

end

endmodule //mapper

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:        Evie Kyritsis
//
// Create Date:    16:46:54 12/06/2014
// Design Name:
// Module Name:    crosshairs
// Project Name:   Maestro
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module crosshairs(
			input clk,
			input [10:0] hcount,
			input [9:0] vcount,
			input [10:0] x_red,
			input [9:0] y_red,
			input [10:0] x_green,
			input [9:0] y_green,
			input [23:0] pixel,
			output reg [23:0] cross_pixel
	);

	always @(posedge clk) begin

		//white cross hairs used for discovering color values of gloves at center pixel of
video
		/*if(hcount==362+150 && vcount!==252+125)begin
			cross_pixel <= {8'd255, 8'd255, 8'd255};
		end
```

```
            else if(hcount!==362+150 && vcount==252+125)begin
                  cross_pixel <= {8'd255, 8'd255, 8'd255};
            end
            else if(hcount==362+150 && vcount==252+125)begin
                  cross_pixel <= pixel;
            end
            else begin
                  cross_pixel <= pixel;
            end*/

            //display cross hairs on center of masses of 2 hand/color detections
            if(hcount == x_red || vcount == y_red)begin
                  cross_pixel[23:16] <= 8'd255; //red cross hair for red hand
                  cross_pixel[15:8] <= 8'd0;
                  cross_pixel[7:0] <= 8'd0;
            end
            else if(hcount == x_green || vcount == y_green)begin
                  cross_pixel[23:16] <= 8'd0;   //green cross hair for green hand
                  cross_pixel[15:8] <= 8'd255;
                  cross_pixel[7:0] <= 8'd0;
            end
            else begin
                  cross_pixel <= pixel;         //everywhere else is the original video_pixel
            end


        end


endmodule //crosshairs

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:       Janelle Wellons
//
// Create Date:    18:23:55 12/04/2014
// Design Name:
// Module Name:    color_wheel
// Project Name:
// Target Devices:
// Tool versions:
// Description: Creates the color wheel affect, showing specified color
//                                 when hands are in appropriate pi slice
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
//
// Piano Logic
//
////////////////////////////////////////////////////////////////////
module piano_logic
   #(parameter RADIUS = 384,            // default radius: 383 pixels
               COLOR = 24'hFF_FF_66)  // default color: white
   (input [10:0] hcount,
```

```
  input [9:0] vcount,
     input vclock,
     input [10:0] handx,  // x position of hand
     input [9:0] handy,  // y position of hand
  output reg [23:0] pixel,
     output reg [2:0] octave);

          reg signed [11:0] deltax;
          reg signed [10:0] deltay;
          reg signed [23:0] deltax_squared;
          reg signed [21:0] deltay_squared;
          reg signed [11:0] x0;
          reg signed [10:0] y0;
          reg signed [11:0] x1;
          reg signed [10:0] y1;


          parameter radius_squared = RADIUS*RADIUS;


          // Define colors of the colorwheel
          parameter RED = 24'hFF_00_00;
          parameter ORANGE = 24'hFF_66_00;
          parameter YELLOW = 24'hFF_CC_00;
          parameter LIGHT_GREEN = 24'h99_FF_33;
          parameter GREEN = 24'h00_CC_00;
          parameter TEAL = 24'h00_CC_99;
          parameter LIGHT_BLUE = 24'h00_CC_FF;
          parameter BLUE = 24'h00_66_FF;
          parameter VIOLET = 24'h66_00_FF;
          parameter PURPLE = 24'h99_00_FF;
          parameter FUCHSIA = 24'hCC_33_99;
          parameter PINK = 24'hFF_00_66;

          parameter SCREEN_CENTER_X = 512;
          parameter SCREEN_CENTER_Y = 384;

          parameter x = SCREEN_CENTER_X - RADIUS;
       parameter y = SCREEN_CENTER_Y - RADIUS;

          //make outputs octave 1, 2, 3
          parameter radius_squared2 = 250;
          parameter radius_squared3 = 100;

always @(posedge vclock) begin

          deltax <= (hcount > (x + RADIUS)) ? (hcount-(x+RADIUS)):
                 ((x+RADIUS)-hcount);
          deltay <= (vcount > (y + RADIUS)) ? (vcount-(y+RADIUS)):
                 ((y+RADIUS)-vcount);
      deltax_squared <= deltax * deltax;
      deltay_squared <= deltay * deltay;

      x0 <= handx - SCREEN_CENTER_X; // move origin to center of screen
      y0 <= -(handy - SCREEN_CENTER_Y);

          x1 <= hcount - SCREEN_CENTER_X;
          y1 <= -(vcount - SCREEN_CENTER_Y);
           //Inside the Circle
     if (deltax_squared + deltay_squared <= radius_squared) begin
          octave <= 1;
```

```
// First Quadrant
	if ((x0 > 0) && (y0 > 0) && (x1 > 0) && (y1 > 0)) begin
		if ((x0 > 2*y0)    && (x1 > 2*y1)) // theta between 0 and 30
			pixel <= RED;
		else if ((x0 < 2 * y0) && (y0 < 2 * x0) && // between 30, 60
				(x1 < 2 * y1) && (y1 < 2 * x1))
			pixel <= ORANGE;
		else if ((y0 > 2* x0) && (y1 > 2*x1)) // between 60,90
			pixel <= YELLOW;
		else
			pixel <= 0;
	end

 //Second Quadrant
	else if ((x0 < 0) && (y0 > 0) && (x1 < 0) && (y1 > 0)) begin
		if ((y0 > -2 * x0) && (y1 > -2 * x1))  // between 90,120
			pixel <= LIGHT_GREEN;
		else if ((x0 > -2*y0) && (y0 < -2*x0) && //between 120,150
				(y1 < -2*x1) && (x1 > - 2*y1))
			pixel <= GREEN;
		else if ((x0 < -2*y0) && (x1 < -2*y1)) // between 150,180
			pixel <= TEAL;
	end

//Third Quadrant
	else if ((x0 < 0) && (y0 < 0) && (x1 < 0) && (y1 < 0)) begin
		if ((x0 < 2*y0) && (x1 < 2*y1))  // between 180,210
			pixel <= LIGHT_BLUE;
		else if ((x0 > 2*y0) && (y0 > 2*x0) && // between 210,240
				(x1 > 2*y1) && (y1 > 2*x1))
			pixel <= BLUE;
		else if ((y0 < 2*x0) && (y1 < 2*x1)) // between 240,270
			pixel <= VIOLET;
		else
			pixel <= 0;
	end

//Fourth Quadrant
	else if ((x0 > 0) && (y0 < 0) && (x1 > 0) && (y1 < 0)) begin
		if ((y0 < -2*x0) && (y1 < -2*x1))      // between 270,300
			pixel <= PURPLE;
		else if ((y0 > -2*x0) && (x0 < -2*y0) && // between 300,330
			(y1 > -2*x1) && (x1 < -2*y1))
			pixel <= FUCHSIA;
		else if ((x0 > -2*y0) && (x1 > -2*y1)) // between 330,360
			pixel <= PINK;
		else
			pixel <= 0;
	end

//Other
	else
		pixel <= 0;

	if (deltax_squared + deltay_squared <= radius_squared2)
		octave <= 2;
	if (deltax_squared + deltay_squared <= radius_squared3)
		octave <= 3;

end
```

```
        //Outside the Circle
              else pixel <= 0;
              end
endmodule

//////////////////////////////////////////////////////////////////////
//
// Create Hand Sprites
//
//////////////////////////////////////////////////////////////////////

module hand_sprite
   #(parameter WIDTH = 20,            // default width: 64 pixels
               HEIGHT = 20,           // default height: 64 pixels
               COLOR = 24'hCC_CC_FF)  // default color: grey-blue
   (input [10:0] handx,hcount,
    input [9:0] handy,vcount,
    output reg [23:0] pixel);

   always @ * begin
      if ((hcount >= handx && hcount < (handx+WIDTH)) &&
        (vcount >= handy && vcount < (handy+HEIGHT)))
       pixel = COLOR;
      else pixel = 0;
   end
endmodule

//////////////////////////////////////////////////////////////////////
//
// Filled Color Wheel (outer)
//
//////////////////////////////////////////////////////////////////////
module color_wheel
   #(parameter RADIUS = 384,            // default radius: 383 pixels
               COLOR = 24'hFF_FF_66)  // default color: white
   (input [10:0]  hcount,
    input [9:0] vcount,
        input vclock,
    output reg [23:0] pixel);

              reg signed [11:0] deltax;
              reg signed [10:0] deltay;
              reg signed [23:0] deltax_squared;
              reg signed [21:0] deltay_squared;
              reg signed [11:0] x1;
              reg signed [10:0] y1;


              parameter radius_squared = RADIUS*RADIUS; //hardcoded for now


              // Define colors of the colorwheel
              parameter RED = 24'hFF_00_00;
              parameter ORANGE = 24'hFF_66_00;
              parameter YELLOW = 24'hFF_CC_00;
              parameter LIGHT_GREEN = 24'h99_FF_33;
              parameter GREEN = 24'h00_CC_00;
              parameter TEAL = 24'h00_CC_99;
              parameter LIGHT_BLUE = 24'h00_CC_FF;
              parameter BLUE = 24'h00_66_FF;
              parameter VIOLET = 24'h66_00_FF;
```

```
        parameter PURPLE = 24'h99_00_FF;
        parameter FUCHSIA = 24'hCC_33_99;
        parameter PINK = 24'hFF_00_66;

        parameter SCREEN_CENTER_X = 512;
        parameter SCREEN_CENTER_Y = 384;

        parameter x = SCREEN_CENTER_X - RADIUS;
     parameter y = SCREEN_CENTER_Y - RADIUS;


always @(posedge vclock) begin

        deltax <= (hcount > (x + RADIUS)) ? (hcount-(x+RADIUS)):
              ((x+RADIUS)-hcount);
        deltay <= (vcount > (y + RADIUS)) ? (vcount-(y+RADIUS)):
              ((y+RADIUS)-vcount);
    deltax_squared <= deltax * deltax;
    deltay_squared <= deltay * deltay;

        x1 <= hcount - SCREEN_CENTER_X;
        y1 <= -(vcount - SCREEN_CENTER_Y);

         //Inside the Circle
    if (deltax_squared + deltay_squared <= radius_squared) begin
        // First Quadrant
        if ((x1 > 0) && (y1 > 0)) begin
              if ( x1 > 2*y1) // theta between 0 and 30
                    pixel <= RED;
              else if ( // between 30, 60
                          (x1 < 2 * y1) && (y1 < 2 * x1))
                    pixel <= ORANGE;
              else if ( y1 > 2*x1) // between 60,90
                    pixel <= YELLOW;
              else
                    pixel <= 0;
        end

    // Second Quadrant
        else if ( (x1 < 0) && (y1 > 0)) begin
              if (y1 > -2*x1)     // between 90,120
                    pixel <= LIGHT_GREEN;
              else if (//between 120,150
                          (y1 < -2*x1) && (x1 > - 2*y1))
                    pixel <= GREEN;
              else if ( x1 < -2*y1) // between 150,180
                    pixel <= TEAL;
        end

    //Third Quadrant
        else if ( (x1 < 0) && (y1 < 0)) begin
              if ( x1 < 2*y1)     // between 180,210
                    pixel <= LIGHT_BLUE;
              else if (                                                // between 210,240
                          (x1 > 2*y1) && (y1 > 2*x1))
                    pixel <= BLUE;
              else if (y1 < 2*x1) // between 240,270
                    pixel <= VIOLET;
              else
                    pixel <= 0;
        end
```

```
	//Fourth Quadrant
		else if ( (x1 > 0) && (y1 < 0)) begin
			if ( (y1 < -2*x1))  // between 270,300
				pixel <= PURPLE;
			else if ( // between 300,330
				(y1 > -2*x1) && (x1 < -2*y1))
				pixel <= FUCHSIA;
			else if ( x1 > -2*y1) // between 330,360
				pixel <= PINK;
			else
				pixel <= 0;
		end
	//Other
		else
			pixel <= 0;
	end
	 //Outside the Circle
		else pixel <= 0;
	end
endmodule


//////////////////////////////////////////////////////////////////////
//
// Filled Color Wheel (Middle)
//
//////////////////////////////////////////////////////////////////////
module color_wheel2
   #(parameter RADIUS = 250,            // default radius: 383 pixels
               COLOR = 24'hFF_FF_66)  // default color: white
   (input [10:0]  hcount,
    input [9:0] vcount,
	input vclock,
    output reg [23:0] pixel);

		reg signed [11:0] deltax;
		reg signed [10:0] deltay;
		reg signed [23:0] deltax_squared;
		reg signed [21:0] deltay_squared;
		reg signed [11:0] x1;
		reg signed [10:0] y1;


		parameter radius_squared = RADIUS*RADIUS; //hardcoded for now


		// Define colors of the colorwheel
		parameter RED = 24'hFF_33_33;
		parameter ORANGE = 24'hFF_85_33;
		parameter YELLOW = 24'hFF_DB_4D;
		parameter LIGHT_GREEN = 24'h99_FF_33;
		parameter GREEN = 24'h4D_DB_4D;
		parameter TEAL = 24'h4D_DB_B8;
		parameter LIGHT_BLUE = 24'h66_E0_FF;
		parameter BLUE = 24'h4D_94_FF;
		parameter VIOLET = 24'h94_4D_FF;
		parameter PURPLE = 24'hBD_5C_FF;
		parameter FUCHSIA = 24'hD6_5C_AD;
		parameter PINK = 24'hFF_4D_94;
```

```
	parameter SCREEN_CENTER_X = 512;
	parameter SCREEN_CENTER_Y = 384;

	parameter x = SCREEN_CENTER_X - RADIUS;
    parameter y = SCREEN_CENTER_Y - RADIUS;


always @(posedge vclock) begin

	deltax <= (hcount > (x + RADIUS)) ? (hcount-(x+RADIUS)):
		((x+RADIUS)-hcount);
	deltay <= (vcount > (y + RADIUS)) ? (vcount-(y+RADIUS)):
		((y+RADIUS)-vcount);
      deltax_squared <= deltax * deltax;
      deltay_squared <= deltay * deltay;

	x1 <= hcount - SCREEN_CENTER_X;
	y1 <= -(vcount - SCREEN_CENTER_Y);

	 //Inside the Circle
    if (deltax_squared + deltay_squared <= radius_squared) begin
	// First Quadrant
	if ((x1 > 0) && (y1 > 0)) begin
		if ( x1 > 2*y1) // theta between 0 and 30
			pixel <= RED;
		else if ( // between 30, 60
				(x1 < 2 * y1) && (y1 < 2 * x1))
			pixel <= ORANGE;
		else if ( y1 > 2*x1) // between 60,90
			pixel <= YELLOW;
		else
			pixel <= 0;
	end

    // Second Quadrant
	else if ( (x1 < 0) && (y1 > 0)) begin
		if (y1 > -2*x1)     // between 90,120
			pixel <= LIGHT_GREEN;
		else if (//between 120,150
				(y1 < -2*x1) && (x1 > - 2*y1))
			pixel <= GREEN;
		else if ( x1 < -2*y1) // between 150,180
			pixel <= TEAL;
	end

    //Third Quadrant
	else if ( (x1 < 0) && (y1 < 0)) begin
		if ( x1 < 2*y1)     // between 180,210
			pixel <= LIGHT_BLUE;
		else if (                                                 // between 210,240
				(x1 > 2*y1) && (y1 > 2*x1))
			pixel <= BLUE;
		else if (y1 < 2*x1) // between 240,270
			pixel <= VIOLET;
		else
			pixel <= 0;
	end

    //Fourth Quadrant
	else if ( (x1 > 0) && (y1 < 0)) begin
		if ( (y1 < -2*x1))  // between 270,300
```

```
                        pixel <= PURPLE;
                  else if ( // between 300,330
                        (y1 > -2*x1) && (x1 < -2*y1))
                        pixel <= FUCHSIA;
                  else if ( x1 > -2*y1) // between 330,360
                        pixel <= PINK;
                  else
                        pixel <= 0;
            end
      //Other
            else
                  pixel <= 0;
      end
       //Outside the Circle
            else pixel <= 0;
      end
endmodule

//////////////////////////////////////////////////////////////////////
//
// Filled Color Wheel (Inner)
//
//////////////////////////////////////////////////////////////////////
module color_wheel3
   #(parameter RADIUS = 125,            // default radius: 383 pixels
               COLOR = 24'hFF_FF_66)  // default color: white
   (input [10:0]  hcount,
    input [9:0] vcount,
        input vclock,
    output reg [23:0] pixel);

            reg signed [11:0] deltax;
            reg signed [10:0] deltay;
            reg signed [23:0] deltax_squared;
            reg signed [21:0] deltay_squared;
            reg signed [11:0] x1;
            reg signed [10:0] y1;


            parameter radius_squared = RADIUS*RADIUS;


            // Define colors of the colorwheel
            parameter RED = 24'hFF_70_70;
            parameter ORANGE = 24'hFF_9D_5C;
            parameter YELLOW = 24'hFF_E6_82;
            parameter LIGHT_GREEN = 24'hB8_FF_70;
            parameter GREEN = 24'h71_E2_71;
            parameter TEAL = 24'h82_E6_CD;
            parameter LIGHT_BLUE = 24'h94_E9_FF;
            parameter BLUE = 24'h82_B4_FF;
            parameter VIOLET = 24'hB4_82_FF;
            parameter PURPLE = 24'hD1_8D_FF;
            parameter FUCHSIA = 24'hFF_99_FF;
            parameter PINK = 24'hFF_82_B4;

            parameter SCREEN_CENTER_X = 512;
            parameter SCREEN_CENTER_Y = 384;

            parameter x = SCREEN_CENTER_X - RADIUS;
         parameter y = SCREEN_CENTER_Y - RADIUS;
```

```
always @(posedge vclock) begin

        deltax <= (hcount > (x + RADIUS)) ? (hcount-(x+RADIUS)):
              ((x+RADIUS)-hcount);
        deltay <= (vcount > (y + RADIUS)) ? (vcount-(y+RADIUS)):
              ((y+RADIUS)-vcount);
     deltax_squared <= deltax * deltax;
     deltay_squared <= deltay * deltay;

        x1 <= hcount - SCREEN_CENTER_X;
        y1 <= -(vcount - SCREEN_CENTER_Y);

         //Inside the Circle
    if (deltax_squared + deltay_squared <= radius_squared) begin
        // First Quadrant
        if ((x1 > 0) && (y1 > 0)) begin
              if ( x1 > 2*y1) // theta between 0 and 30
                    pixel <= RED;
              else if ( // between 30, 60
                          (x1 < 2 * y1) && (y1 < 2 * x1))
                    pixel <= ORANGE;
              else if ( y1 > 2*x1) // between 60,90
                    pixel <= YELLOW;
              else
                    pixel <= 0;
        end

    // Second Quadrant
        else if ( (x1 < 0) && (y1 > 0)) begin
              if (y1 > -2*x1)     // between 90,120
                    pixel <= LIGHT_GREEN;
              else if (//between 120,150
                          (y1 < -2*x1) && (x1 > - 2*y1))
                    pixel <= GREEN;
              else if ( x1 < -2*y1) // between 150,180
                    pixel <= TEAL;
        end

    //Third Quadrant
        else if ( (x1 < 0) && (y1 < 0)) begin
              if ( x1 < 2*y1)     // between 180,210
                    pixel <= LIGHT_BLUE;
              else if (                                        // between 210,240
                          (x1 > 2*y1) && (y1 > 2*x1))
                    pixel <= BLUE;
              else if (y1 < 2*x1) // between 240,270
                    pixel <= VIOLET;
              else
                    pixel <= 0;
        end

    //Fourth Quadrant
        else if ( (x1 > 0) && (y1 < 0)) begin
              if ( (y1 < -2*x1))  // between 270,300
                    pixel <= PURPLE;
              else if ( // between 300,330
                    (y1 > -2*x1) && (x1 < -2*y1))
                    pixel <= FUCHSIA;
              else if ( x1 > -2*y1) // between 330,360
```

```
                        pixel <= PINK;
                else
                        pixel <= 0;
            end
        //Other
            else
                pixel <= 0;
        end
         //Outside the Circle
            else pixel <= 0;
        end
endmodule



////////////////////////////////////////////////////
//
//
//picture_blob: display a picture
//
//////////////////////////////////////////////////
module picture_blob
        #(parameter WIDTH = 64,
            parameter HEIGHT = 50)
        (input vclock,
         input [10:0] x,hcount,
         input [9:0] y, vcount,
         output reg [23:0] pixel);

         wire [11:0] image_addr;
         wire [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

         //note the one clock cycle delay in pixel
         always @(posedge vclock) begin
            if((hcount >= x && hcount < (x+WIDTH)) &&
                vcount >= y && vcount < (x +WIDTH))
                pixel <= {red_mapped, green_mapped, blue_mapped};
                else pixel <= 0;
         end

         //calculate rom address and read the location
         assign image_addr = (hcount-x) + (vcount - y)* WIDTH;
         ram_red rom1(image_addr, vclock, image_bits);
         rom_green rom2(image_addr, vclock, image_bits);
         rom_blue rom3(image_addr, vclock, image_bits);

         //use color map to create 8bits R,8bits G, 8bits B
         //red_color_map rcm (image_bits, vclock, red_mapped);
         //green_color_map gcm (image_bits, vclock, green_mapped);
         //blue_color_map bcm (image_bits, vclock, blue_mapped);

Endmodule //picture_blob

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Janelle Wellons
//
// Create Date:    13:39:28 12/02/2014
// Design Name:  Geometry
// Module Name:
```

```
// Project Name:
// Target Devices:
// Tool versions:
// Description: Where the geometry of whats shown on the display is done
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
//
// Circle
//
////////////////////////////////////////////////////////////////////
module circle
   #(parameter RADIUS = 384,            // default radius: 383 pixels
               COLOR = 24'hFF_FF_66)  // default color: white
   (input [10:0] hcount,
    input [9:0] vcount,
        input vclock,
    output reg [23:0] pixel);

       reg [11:0] deltax;
   reg [10:0] deltay;
       reg [23:0] deltax_squared;
       reg [21:0] deltay_squared;
       parameter radius_squared = RADIUS*RADIUS; //hardcoded for now
       parameter x0 = 512-RADIUS;
       parameter y0 = 384-RADIUS;

  always @(posedge vclock) begin
        deltax <= (hcount > (x0 + RADIUS)) ? (hcount-(x0+RADIUS)):
                                                    ((x0+RADIUS)-hcount);
    deltay <= (vcount > (y0 + RADIUS)) ? (vcount-(y0+RADIUS)):
                                                    ((y0+RADIUS)-vcount);
    deltax_squared <= deltax * deltax;
    deltay_squared <= deltay * deltay;
             if (deltax_squared + deltay_squared <= radius_squared)
                   pixel <= COLOR;
      else pixel <= 0;
   end
endmodule

////////////////////////////////////////////////////////////////////
//
// Circle Outline
//
////////////////////////////////////////////////////////////////////
module circle_outline
   #(parameter RADIUS = 384,            // default radius: 383 pixels
               COLOR = 24'hFF_FF_FF,
                                TOLERANCE = 750)  //for dealing with pixelated screen
   (input [10:0] hcount,
    input [9:0] vcount,
        input vclock,
    output reg [23:0] pixel);

       reg [11:0] deltax;
```

```
   reg [10:0] deltay;
      reg [23:0] deltax_squared;
      reg [21:0] deltay_squared;
      reg [23:0] sum;
      parameter SCREEN_CENTERX = 512;
      parameter SCREEN_CENTERY = 384;
      parameter RADIUS_SQUARED = RADIUS*RADIUS; //hardcoded for now
      parameter x0 = SCREEN_CENTERX-RADIUS;
      parameter y0 = SCREEN_CENTERY-RADIUS;


  always @(posedge vclock) begin
       deltax <= (hcount > (x0 + RADIUS)) ? (hcount-(x0+RADIUS)):
                                                     ((x0+RADIUS)-hcount);
    deltay <= (vcount > (y0 + RADIUS)) ? (vcount-(y0+RADIUS)):
                                                     ((y0+RADIUS)-vcount);
    deltax_squared <= deltax * deltax;
    deltay_squared <= deltay * deltay;
       sum <= deltax_squared + deltay_squared;
            if (sum <= RADIUS_SQUARED && sum >= (RADIUS_SQUARED - TOLERANCE))
                   pixel <= COLOR;
      else pixel <= 0;
   end
endmodule

//////////////////////////////////////////////////////////////////////
//
// Vertical Divider Line
//
//////////////////////////////////////////////////////////////////////
module vertline
   #(parameter B = 511,            // default position
               COLOR = 24'h66_66_66)  // default color: white
   (input [10:0] hcount,
    output reg [23:0] pixel);

   always @ * begin
      if (hcount == B)
       pixel = COLOR;
      else pixel = 0;
   end
endmodule
//////////////////////////////////////////////////////////////////////
//
// Horizontal Divider Line
//
//////////////////////////////////////////////////////////////////////
module horiline
   #(parameter B = 384,            // default position
               COLOR = 24'h66_66_66)  // default color: white
   (input [10:0] vcount,
    output reg [23:0] pixel);

   always @ * begin
      if (vcount == B)
       pixel = COLOR;
      else pixel = 0;
   end
endmodule //horiline

`timescale 1ns / 1ps
```

```
////////////////////////////////////////////////////////////////////////////////
// Sabina Maddila
// module: oscillator.v
// integration of sound generation module
///////////////////////////////////////////////////////////////////////////////
module oscillator
      #(parameter CLOCK_FREQUENCY=27,
        parameter WSIZE=36,
        parameter INT_DEPTH=12,
        parameter FRAC_DEPTH=6,
        parameter OSC_DEPTH=16,
        parameter DEFAULT_AMPL=1<<(OSC_DEPTH-1),
        parameter SUSTAIN_AMPL=1<<(OSC_DEPTH-1),
        parameter ATTACK_TIME=1000,
        parameter DECAY_TIME=200,
        parameter SUSTAIN_TIME=3000000,
        parameter RELEASE_TIME=5000000)
    (input clk,
        input ready,
        input reset,
        input [2:0]octave,
        input [3:0]note,
        input cont_play,
        input play,
        input usr_pulse,
        output [OSC_DEPTH-1:0]pcm_data);

        parameter MILLISEC_COUNT=CLOCK_FREQUENCY*1000-1;

        //from amplitude_test.v test bench
         wire [OSC_DEPTH-1:0] preamp;
            wire [OSC_DEPTH-1:0] amplifier;
            wire [2*OSC_DEPTH-1:0] postamp;

            assign pcm_data=postamp[31:16];

            amplitude amp (.clk_27mhz(clk),
                                    .reset(reset),
                                    .preamp(preamp),
                                    .amplifier(amplifier),
                                    .postamp(postamp));

            wire [WSIZE+9:0]osc_in;
            wire [WSIZE-1:0]osc_in_prefilter;
            wire [OSC_DEPTH-1:0]osc_out;
            wire timer_elapsed;
            wire [6:0]timer_value;
            wire [2:0]ADSR_state;
            wire enb1kHz;
            wire [6:0]count;
            wire [OSC_DEPTH-1:0]prefilter;
            wire [OSC_DEPTH+9:0]postfilter;

            assign amplifier = cont_play?DEFAULT_AMPL:
                                                    play?0:
                                postfilter[OSC_DEPTH+9:10];

            interpolate_wave wav(.clk_27mhz(clk),.ready(ready),
                                .reset(reset),.octave(octave),.note(note),
                                .sin_wave(osc_in_prefilter));
```

```
            fir_lo31 #(.WSIZE(WSIZE))
                                wav_fltr(.clock(clk),
                                          .reset(reset),
                                          .ready(ready),
                                          .x(osc_in_prefilter),
                                          .y(osc_in));

            dither #(.IN_SIZE(WSIZE+10))
                   dither_in(.clk(clk),.ready(ready),.reset(reset),
                                          .wave_in(osc_in),.wave_out(preamp));

            dither #(.IN_SIZE(2*OSC_DEPTH))
                          dither_out (.clk(clk),.ready(ready),.reset(reset),
                                              .wave_in(postamp),.wave_out(osc_out));

            ADSR #(.BIT_DEPTH(OSC_DEPTH),.SUSTAIN_AMPL(SUSTAIN_AMPL),
              .ATTACK_TIME(ATTACK_TIME),.DECAY_TIME(DECAY_TIME),
                         .SUSTAIN_TIME(SUSTAIN_TIME),.RELEASE_TIME(RELEASE_TIME))
                    adsr_env(.clk_27mhz(clk),
                                    .enb1kHz(enb1kHz),
                                    .reset(reset),
                                    .usr_enable(usr_pulse),
                                    .timer_elapsed(timer_elapsed),
                                    .start_timer(start_timer),
                                    .timer_value(timer_value),
                                    .ADSR_state(ADSR_state),
                                    .amplify(prefilter));

            fir_lo31 #(.WSIZE(OSC_DEPTH))
                         env_fltr(.clock(clk),
                                       .reset(reset),
                                       .ready(enb1kHz),
                                       .x(prefilter),
                                       .y(postfilter));

            envelope_timer timr(.clk_27mhz(clk),
                                              .reset(reset),
                                              .enb1kHz(enb1kHz),
                                              .value(timer_value),
                                              .count(count),
                                              .start(start_timer),
                                              .done(timer_elapsed));

            kHz_divider #(.MILLISEC_COUNT(MILLISEC_COUNT))
                                    kHzdivde(.clk_27mhz(clk),
                                                   .reset(reset),
                                                   .enb1kHz(enb1kHz));
endmodule

/////////////////////////////////////////////////////////////////////////////////
// module: s_s_s_synth
// incorporates the oscillator module with the AC97 driver
// **adapted using the recorder module frol Lab 5
/////////////////////////////////////////////////////////////////////////////////
module s_s_s_synth
       #(parameter CLOCK_FREQUENCY=27,
         parameter WSIZE=36,
         parameter INT_DEPTH=12,
         parameter FRAC_DEPTH=6,
         parameter OSC_DEPTH=16,
         parameter SUSTAIN_AMPL=1<<(OSC_DEPTH-1),
```

```
    parameter ATTACK_TIME=7,
    parameter DECAY_TIME=20,
    parameter SUSTAIN_TIME=30,
    parameter RELEASE_TIME=50,
    parameter OUTER_OCTAVE=5,
    parameter MIDDLE_OCTAVE=4,
    parameter INNER_OCTAVE=3)
   (input wire clock,                  // 27mhz system clock
    input wire reset,                 // 1 to reset to initial state
input wire ready,                 // 1 when AC97 data is available
input [23:0] color,                          // input from visualization module
input cont_play,                                    // indicates if in continous play mode
    input play_chord,                                   // indicar
    input usr_pulse,                                  // if not continous play,
output reg [19:0] to_ac97_data); // 16-bit PCM data to headphone + 2 bits fo' ovah'flo'


    wire [2:0]octave;
    wire [3:0]note;
    wire [15:0]pcm_data1;
    wire [15:0]pcm_data2;
    wire [15:0]pcm_data3;

    always @(posedge clock)begin
       if(ready) begin
             to_ac97_data <= {pcm_data1,4'b0}+{pcm_data2,4'b0}+{pcm_data3,4'b0};
          end
    end
    wire [23:0] color_to_sound;
    wire storing;
   color_storage  store(.clk(clk),.reset(reset),.color(color),.color_to_sound(color_to_sound),
                                                    .storing(storing));

    oscillator #(.CLOCK_FREQUENCY(CLOCK_FREQUENCY),.WSIZE(WSIZE),
                                .INT_DEPTH(INT_DEPTH),.FRAC_DEPTH(FRAC_DEPTH),
                                .OSC_DEPTH(OSC_DEPTH),.SUSTAIN_AMPL(SUSTAIN_AMPL),
                                .ATTACK_TIME(ATTACK_TIME),.DECAY_TIME(DECAY_TIME),
                                .SUSTAIN_TIME(SUSTAIN_TIME),.RELEASE_TIME(RELEASE_TIME))
                          s1(.clk(clock),
                 .reset(reset),
                              .ready(ready),
             .octave(octave),
                              .note(note),
                              .cont_play(cont_play),
                                 .play(1),
                              .usr_pulse(usr_pulse),
             .pcm_data(pcm_data1));

    oscillator #(.CLOCK_FREQUENCY(CLOCK_FREQUENCY),.WSIZE(WSIZE),
                                .INT_DEPTH(INT_DEPTH),.FRAC_DEPTH(FRAC_DEPTH),
                                .OSC_DEPTH(OSC_DEPTH),.DEFAULT_AMPL(1<<(OSC_DEPTH-3)),
                                .SUSTAIN_AMPL(1<<(OSC_DEPTH-3)),
                                .ATTACK_TIME(ATTACK_TIME),.DECAY_TIME(DECAY_TIME),
                                .SUSTAIN_TIME(SUSTAIN_TIME),.RELEASE_TIME(RELEASE_TIME))
             s2(.clk(clock),
                 .reset(reset),
                              .ready(ready),
             .octave(octave+1),
                              .note(note),
                              .cont_play(cont_play),
                                 .play(play_chord),
```

```
                                    .usr_pulse(usr_pulse),
                  .pcm_data(pcm_data2));

         oscillator #(.CLOCK_FREQUENCY(CLOCK_FREQUENCY),.WSIZE(WSIZE),
                                       .INT_DEPTH(INT_DEPTH),.FRAC_DEPTH(FRAC_DEPTH),
                                       .OSC_DEPTH(OSC_DEPTH),.DEFAULT_AMPL(1<<(OSC_DEPTH-2)),
                                       .SUSTAIN_AMPL(1<<(OSC_DEPTH-2)),
                                       .ATTACK_TIME(ATTACK_TIME),.DECAY_TIME(DECAY_TIME),
                                       .SUSTAIN_TIME(SUSTAIN_TIME),.RELEASE_TIME(RELEASE_TIME))
                  s3(.clk(clock),
                     .reset(reset),
                                    .ready(ready),
                  .octave(octave),
                                    .note(note+4),
                                    .cont_play(cont_play),
                                       .play(play_chord),
                                    .usr_pulse(usr_pulse),
                  .pcm_data(pcm_data3));

        color_conversion #(.OUTER_OCTAVE(OUTER_OCTAVE),
                                                .MIDDLE_OCTAVE(MIDDLE_OCTAVE),
                                                .INNER_OCTAVE(INNER_OCTAVE))
                                              clr (.clk_27mhz(clock),.ready(ready),.reset(reset),

            .color(color_to_sound),.octave(octave),.note(note),.storing(storing));

Endmodule //s_s_s_synth

`timescale 1ns / 1ps
/* Sabina Maddila
/* amplitude.v
/* modules: amplitude, ASDR
/**/

module amplitude
  #(parameter BIT_DEPTH=16)
   (input clk_27mhz,
        input reset,
        input signed [BIT_DEPTH-1:0]preamp,
        input [BIT_DEPTH-1:0]amplifier,
        output [2*BIT_DEPTH-1:0]postamp);

        reg signed[2*BIT_DEPTH-1:0]ampd_wave;
        wire signed[BIT_DEPTH-1:0]sgnd_amplifier=amplifier[BIT_DEPTH-1:0];

        always @(posedge clk_27mhz)begin
               ampd_wave <= preamp*sgnd_amplifier;
        end

        assign postamp=ampd_wave[2*BIT_DEPTH-1:0];

endmodule

/////////////////////////////////////////////////////////////////////////////////
// module: ADSR
// FSM for the ADSR (Attack-Delay-Sustain-Response) volume envelope
// **determines the amplitude multiplication factor (modulation)
/////////////////////////////////////////////////////////////////////////////////
module ADSR
 #(parameter BIT_DEPTH=16,
   parameter SUSTAIN_AMPL=1<<(BIT_DEPTH-1),
```

```
	parameter ATTACK_TIME=7,
	parameter DECAY_TIME=20,
	parameter SUSTAIN_TIME=30,
	parameter RELEASE_TIME=50)
   (input clk_27mhz,
    input enb1kHz,
    input reset,
	input usr_enable,
	input timer_elapsed,
	output start_timer,
	output [6:0]timer_value,
	output [2:0]ADSR_state,
	output [BIT_DEPTH-1:0]amplify);

    /*defined states, where state[2] determies if synth is @rest*/
	parameter [2:0]ATTACK=3'b100;
    parameter [2:0]DECAY=3'b101;
    parameter [2:0]SUSTAIN=3'b110;
    parameter [2:0]RELEASE=3'b111;
	parameter [2:0]REST=3'b000;

	parameter MAX_AMPL=1<<BIT_DEPTH;

	parameter ATTACK_INCR=MAX_AMPL/ATTACK_TIME;
	parameter DECAY_INCR=(MAX_AMPL-SUSTAIN_AMPL)/DECAY_TIME;
	parameter SUSTAIN_INCR=0;
	parameter RELEASE_INCR=SUSTAIN_AMPL/RELEASE_TIME;
	parameter REST_INCR=0;

	reg [2:0]state=REST;
	reg [BIT_DEPTH:0]amp_factor=0;
	reg [6:0]set_timer_val=0;
	reg timer_start=0;

	wire max_factor = (amp_factor>=MAX_AMPL);
	wire sustain_factor = (amp_factor==SUSTAIN_AMPL);
	wire decay_factor = (amp_factor<=DECAY_INCR);

	assign start_timer=timer_start;
	assign ADSR_state=state;
	assign amplify={1'b0,amp_factor[BIT_DEPTH-1:1]};
	assign timer_value=set_timer_val;

	always @(posedge clk_27mhz) begin
		case(state)
			REST:begin
				state <= usr_enable ? ATTACK: state;
				amp_factor <= 0;
				set_timer_val <= ATTACK_TIME;
				timer_start <= usr_enable ? 1: 0;
				end
			ATTACK:begin
				state <= timer_elapsed ? DECAY: state;
				amp_factor <= max_factor ? amp_factor:
							enb1kHz ? amp_factor+ATTACK_INCR:
amp_factor;
				set_timer_val <= timer_elapsed ? DECAY_TIME: set_timer_val;
				timer_start <= timer_elapsed ? 1: 0;
				end
			DECAY:begin
				state <= timer_elapsed ? SUSTAIN: state;
```

```
                              amp_factor <= sustain_factor ? amp_factor:
                                                          enb1kHz ? amp_factor-DECAY_INCR:
amp_factor;
                              set_timer_val <= timer_elapsed ? SUSTAIN_TIME: set_timer_val;
                              timer_start <= timer_elapsed ? 1: 0;
                              end
                        SUSTAIN:begin
                              state <= timer_elapsed ? RELEASE: state;
                              amp_factor <= SUSTAIN_AMPL;
                              set_timer_val <= timer_elapsed ? RELEASE_TIME: set_timer_val;
                              timer_start <= timer_elapsed ? 1: 0;
                              end
                        RELEASE:begin
                              state <= timer_elapsed ? REST: state;
                              amp_factor <= decay_factor ? 0:
                                                          enb1kHz ? amp_factor-DECAY_INCR:
amp_factor;
                              set_timer_val <= 0;
                              timer_start <= 0;
                              end
                        default: state <= REST;
                  endcase
            end


endmodule

////////////////////////////////////////////////////////////////////////////////
// module: envelope_timer
// generates determines the length oftime for an envelope (millisecond count)
// **adapted from similar module developed for Lab 4
////////////////////////////////////////////////////////////////////////////////
module envelope_timer
   (input clk_27mhz,
        input reset,
    input enb1kHz,
    input [6:0]value,
    input start,
        output [6:0]count,
    output done);

        reg [6:0]cnt=7'b0;
        reg [1:0]expire=2'b0;

        always @(posedge clk_27mhz) begin
              cnt <= reset ? 0: start ? value : enb1kHz&&(cnt>0) ? cnt+7'b1111111 : cnt;
              expire <= reset ? 0: start ? 2'b0: (expire>=2'b01)?2'b11: (cnt<=0)?2'b01 : expire;
        end

        assign done=^expire; //ensures that done is only asserted for a single cycle
        assign count=cnt;

endmodule //envelope_timer

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//Sabina Maddila
// module: color_conversion
// receives note information from THE VISUALIZATION MODULE
// **uses register to hold frequency information (b/c clock domains)
////////////////////////////////////////////////////////////////////////////////
```

```
module color_conversion
 #(parameter OUTER_OCTAVE=3,
   parameter MIDDLE_OCTAVE=4,
       parameter INNER_OCTAVE=5)
  (input clk_27mhz,
   input ready,
       input reset,
       input storing,
       input [23:0]color,
       output [2:0]octave,
       output [3:0]note);

       parameter RED=24'hFF_00_00;
       parameter RED2 = 24'hFF_70_70;
       parameter RED1 = 24'hFF_33_33;

       parameter ORANGE=24'hFF_66_00;
       parameter ORANGE2 = 24'hFF_9D_5C;
       parameter ORANGE1 = 24'hFF_85_33;

       parameter YELLOW=24'hFF_CC_00;
       parameter YELLOW2 = 24'hFF_E6_82;
       parameter YELLOW1 = 24'hFF_DB_4D;

       parameter LIGHT_GREEN=24'h99_FF_33;
       parameter LIGHT_GREEN2 = 24'hB8_FF_70;
       parameter LIGHT_GREEN1 = 24'h99_FF_33;

       parameter GREEN=24'h00_CC_00;
       parameter GREEN2 = 24'h71_E2_71;
       parameter GREEN1 = 24'h4D_DB_4D;

       parameter TEAL=24'h00_CC_99;
       parameter TEAL2 = 24'h82_E6_CD;
       parameter TEAL1 = 24'h4D_DB_B8;

       parameter LIGHT_BLUE=24'h00_CC_FF;
       parameter LIGHT_BLUE2 = 24'h94_E9_FF;
       parameter LIGHT_BLUE1 = 24'h66_E0_FF;

       parameter BLUE=24'h66_00_FF;
       parameter BLUE2 = 24'h82_B4_FF;
       parameter BLUE1 = 24'h4D_94_FF;

       parameter VIOLET=24'h99_00_FF;
       parameter VIOLET2 = 24'hB4_82_FF;
       parameter VIOLET1 = 24'h94_4D_FF;

       parameter FUCHSIA=24'hCC_33_99;
       parameter FUCHSIA1 = 24'hD6_5C_AD;
       parameter FUCHSIA2 = 24'hFF_99_FF;

       parameter PINK=24'hFF_00_66;
       parameter PINK2 = 24'hFF_82_B4;
       parameter PINK1 = 24'hFF_4D_94;

       reg [3:0]determine_note=15; //default values defined in noteLUT
       reg [2:0]determine_octave=7;

       assign octave=determine_octave;
       assign note=determine_note;
```

```
	always @(posedge clk_27mhz)begin
		if (ready&&storing) begin
			case(color)
				RED: {determine_note,determine_octave} <= {0,OUTER_OCTAVE};
		//C
				RED1: {determine_note,determine_octave} <= {0,MIDDLE_OCTAVE};
				RED2: {determine_note,determine_octave} <= {0,INNER_OCTAVE};
				ORANGE: {determine_note,determine_octave} <= {1,OUTER_OCTAVE};
//C#|Db
				ORANGE1: {determine_note,determine_octave} <= {1,MIDDLE_OCTAVE};
				ORANGE2: {determine_note,determine_octave} <= {1,INNER_OCTAVE};
				YELLOW: {determine_note,determine_octave} <= {2,OUTER_OCTAVE};	//D
				YELLOW1: {determine_note,determine_octave} <= {2,MIDDLE_OCTAVE};
				YELLOW2: {determine_note,determine_octave} <= {2,INNER_OCTAVE};
				LIGHT_GREEN: {determine_note,determine_octave} <= {3,OUTER_OCTAVE};
//D#|Eb
				LIGHT_GREEN1: {determine_note,determine_octave} <= {3,MIDDLE_OCTAVE};
				LIGHT_GREEN2: {determine_note,determine_octave} <= {3,INNER_OCTAVE};
				GREEN: {determine_note,determine_octave} <= {4,OUTER_OCTAVE};	//E
				GREEN1: {determine_note,determine_octave} <= {4,MIDDLE_OCTAVE};
				GREEN2: {determine_note,determine_octave} <= {4,INNER_OCTAVE};
				TEAL: {determine_note,determine_octave} <= {5,OUTER_OCTAVE};	//F
				TEAL1: {determine_note,determine_octave} <= {5,MIDDLE_OCTAVE};
				TEAL2: {determine_note,determine_octave} <= {5,INNER_OCTAVE};
				LIGHT_BLUE: {determine_note,determine_octave} <= {6,OUTER_OCTAVE};
//F#|Gb
				LIGHT_BLUE1: {determine_note,determine_octave} <= {6,MIDDLE_OCTAVE};
				LIGHT_BLUE2: {determine_note,determine_octave} <= {6,INNER_OCTAVE};
				BLUE: {determine_note,determine_octave} <= {7,OUTER_OCTAVE};
//G
				BLUE1: {determine_note,determine_octave} <= {7,MIDDLE_OCTAVE};
				BLUE2: {determine_note,determine_octave} <= {7,INNER_OCTAVE};
				VIOLET: {determine_note,determine_octave} <= {8,OUTER_OCTAVE};
//G#|Ab
				VIOLET1: {determine_note,determine_octave} <= {8,MIDDLE_OCTAVE};
				VIOLET2: {determine_note,determine_octave} <= {8,INNER_OCTAVE};
				FUCHSIA: {determine_note,determine_octave} <= {9,OUTER_OCTAVE};
//A#|Bb
				FUCHSIA1: {determine_note,determine_octave} <= {9,MIDDLE_OCTAVE};
				FUCHSIA2: {determine_note,determine_octave} <= {9,INNER_OCTAVE};
				PINK: {determine_note,determine_octave} <= {10,OUTER_OCTAVE};	//B
				PINK1: {determine_note,determine_octave} <= {10,MIDDLE_OCTAVE};
				PINK2: {determine_note,determine_octave} <= {10,INNER_OCTAVE};
				default: {determine_note,determine_octave} <=
{0,6}/*{determine_note,determine_octave} */;
			endcase
		end
		else if(ready&&~storing) {determine_note,determine_octave} <=
{0,6}/*{determine_note,determine_octave} */;
		else {determine_note,determine_octave} <= {0,6};
	end
endmodule //color_conversion

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//Sabina Maddila
// module: dither
// takes sinusoid and reduces bit_depth, try to reduces quantization
// **dithering improves performance comparred to amplitude truncation
// **reduces the impact of unwanted harmonics
```

```
// **uses triangular probability distribution dither (TPBD), storing PDF in LUT
////////////////////////////////////////////////////////////////////////////////
module dither
 #(parameter IN_SIZE=36,
   parameter OUT_SIZE=16)
  (input clk,
   input ready,
       input reset,
   input [IN_SIZE-1:0]wave_in,
   output [OUT_SIZE-1:0]wave_out);

       reg [IN_SIZE-1:0]triangl_dist=0; //triangle probability dist
       reg [7:0]rand_addr=8'b1; //used to generate random address

       wire rpoly = rand_addr[7]^rand_addr[5]^rand_addr[4]^rand_addr[3];
       reg [IN_SIZE-1:0]with_dither;

       assign wave_out = with_dither[IN_SIZE-1:(IN_SIZE-OUT_SIZE)];

       always @(posedge clk) begin
              if (ready) begin
                     with_dither <= wave_in + triangl_dist + wave_in[OUT_SIZE-1]; //also includes
rounding

                     // lfsr polynomial for random approximation; period of 255
                     rand_addr <= reset?1:{rand_addr[6:0],rpoly};

                     // generate rand triangle distribution lookup table
                     case(rand_addr)
                            8'h01: triangl_dist <= 36'h0000440A2;
                            8'h02: triangl_dist <= 36'h000065D8E;
                            8'h03: triangl_dist <= 36'hFFFFC75F5;
                            8'h04: triangl_dist <= 36'hFFFFB6E92;
                            8'h05: triangl_dist <= 36'h0001B09F3;
                            8'h06: triangl_dist <= 36'hFFFE8CA9B;
                            8'h07: triangl_dist <= 36'h00010AA20;
                            8'h08: triangl_dist <= 36'h00012AC7C;
                            8'h09: triangl_dist <= 36'h0000B91BB;
                            8'h0A: triangl_dist <= 36'hFFFEE37E9;
                            8'h0B: triangl_dist <= 36'hFFFF7288E;
                            8'h0C: triangl_dist <= 36'hFFFFA3503;
                            8'h0D: triangl_dist <= 36'h00006639D;
                            8'h0E: triangl_dist <= 36'hFFFF0B91B;
                            8'h0F: triangl_dist <= 36'h000081B1F;
                            8'h10: triangl_dist <= 36'hFFFEEC969;
                            8'h11: triangl_dist <= 36'h000055EF8;
                            8'h12: triangl_dist <= 36'hFFFFFD022;
                            8'h13: triangl_dist <= 36'h0000ABA58;
                            8'h14: triangl_dist <= 36'h00007D794;
                            8'h15: triangl_dist <= 36'h00011F53A;
                            8'h16: triangl_dist <= 36'h000110DC2;
                            8'h17: triangl_dist <= 36'hFFFFA290F;
                            8'h18: triangl_dist <= 36'h000072941;
                            8'h19: triangl_dist <= 36'hFFFF420A1;
                            8'h1A: triangl_dist <= 36'hFFFE7E8A2;
                            8'h1B: triangl_dist <= 36'h000091B22;
                            8'h1C: triangl_dist <= 36'h00000002F;
                            8'h1D: triangl_dist <= 36'hFFFFF59D6;
                            8'h1E: triangl_dist <= 36'h0001207F9;
                            8'h1F: triangl_dist <= 36'h00003BBC9;
                            8'h20: triangl_dist <= 36'h00004047C;
```

```
8'h21: triangl_dist <= 36'h0000F0893;
8'h22: triangl_dist <= 36'h0000C0A86;
8'h23: triangl_dist <= 36'h000028EA9;
8'h24: triangl_dist <= 36'hFFFF35AF1;
8'h25: triangl_dist <= 36'hFFFF62AC8;
8'h26: triangl_dist <= 36'h00010C12B;
8'h27: triangl_dist <= 36'hFFFE7A9C8;
8'h28: triangl_dist <= 36'hFFFFFACDA;
8'h29: triangl_dist <= 36'hFFFF28B81;
8'h2A: triangl_dist <= 36'h00019646C;
8'h2B: triangl_dist <= 36'h00007BE36;
8'h2C: triangl_dist <= 36'h0000003DD;
8'h2D: triangl_dist <= 36'hFFFFF0FA1;
8'h2E: triangl_dist <= 36'hFFFEB0CC4;
8'h2F: triangl_dist <= 36'h000067AA0;
8'h30: triangl_dist <= 36'hFFFE9526C;
8'h31: triangl_dist <= 36'hFFFEC18A7;
8'h32: triangl_dist <= 36'h00000B351;
8'h33: triangl_dist <= 36'hFFFEE132D;
8'h34: triangl_dist <= 36'h0000CB396;
8'h35: triangl_dist <= 36'h0000CAB6C;
8'h36: triangl_dist <= 36'h00008286F;
8'h37: triangl_dist <= 36'hFFFF184EE;
8'h38: triangl_dist <= 36'h0000598C9;
8'h39: triangl_dist <= 36'h0000099C6;
8'h3A: triangl_dist <= 36'h000188F74;
8'h3B: triangl_dist <= 36'h000053037;
8'h3C: triangl_dist <= 36'h0000BC735;
8'h3D: triangl_dist <= 36'hFFFFE7C57;
8'h3E: triangl_dist <= 36'hFFFFDC20B;
8'h3F: triangl_dist <= 36'h0000D15E5;
8'h40: triangl_dist <= 36'hFFFED131C;
8'h41: triangl_dist <= 36'hFFFF083C1;
8'h42: triangl_dist <= 36'hFFFF2D817;
8'h43: triangl_dist <= 36'hFFFFC4BAC;
8'h44: triangl_dist <= 36'h0000D6AB5;
8'h45: triangl_dist <= 36'h0000BEEB0;
8'h46: triangl_dist <= 36'hFFFEB20EA;
8'h47: triangl_dist <= 36'hFFFFC9859;
8'h48: triangl_dist <= 36'h00000DF35;
8'h49: triangl_dist <= 36'hFFFFD376E;
8'h4A: triangl_dist <= 36'h000057D95;
8'h4B: triangl_dist <= 36'h0000465B4;
8'h4C: triangl_dist <= 36'hFFFF87426;
8'h4D: triangl_dist <= 36'hFFFFDBB85;
8'h4E: triangl_dist <= 36'hFFFE5A1C0;
8'h4F: triangl_dist <= 36'h0001A497E;
8'h50: triangl_dist <= 36'hFFFF280C4;
8'h51: triangl_dist <= 36'hFFFEEBFBA;
8'h52: triangl_dist <= 36'hFFFFB9DF0;
8'h53: triangl_dist <= 36'hFFFF424A5;
8'h54: triangl_dist <= 36'hFFFFFAB15;
8'h55: triangl_dist <= 36'hFFFFA5E43;
8'h56: triangl_dist <= 36'h000160C0D;
8'h57: triangl_dist <= 36'h000133A02;
8'h58: triangl_dist <= 36'hFFFEA62FB;
8'h59: triangl_dist <= 36'h00008D462;
8'h5A: triangl_dist <= 36'hFFFF77A0B;
8'h5B: triangl_dist <= 36'hFFFFD6D65;
8'h5C: triangl_dist <= 36'h000019206;
8'h5D: triangl_dist <= 36'h000152BB0;
```

```
8'h5E: triangl_dist <= 36'hFFFFD3FE7;
8'h5F: triangl_dist <= 36'h0001A1BCD;
8'h60: triangl_dist <= 36'hFFFF8D8DD;
8'h61: triangl_dist <= 36'h000074223;
8'h62: triangl_dist <= 36'h00005DBF9;
8'h63: triangl_dist <= 36'h00001470D;
8'h64: triangl_dist <= 36'h000072280;
8'h65: triangl_dist <= 36'h00005DDDE;
8'h66: triangl_dist <= 36'hFFFF319A3;
8'h67: triangl_dist <= 36'hFFFF03118;
8'h68: triangl_dist <= 36'h0001EA0AE;
8'h69: triangl_dist <= 36'hFFFF2B871;
8'h6A: triangl_dist <= 36'hFFFE82BCC;
8'h6B: triangl_dist <= 36'h0000205B5;
8'h6C: triangl_dist <= 36'h000107217;
8'h6D: triangl_dist <= 36'h00005F87A;
8'h6E: triangl_dist <= 36'hFFFF3BFA4;
8'h6F: triangl_dist <= 36'hFFFFB7CB3;
8'h70: triangl_dist <= 36'hFFFFEB7B0;
8'h71: triangl_dist <= 36'h00019DE20;
8'h72: triangl_dist <= 36'hFFFF1E5BD;
8'h73: triangl_dist <= 36'h0000ECC6F;
8'h74: triangl_dist <= 36'h000050702;
8'h75: triangl_dist <= 36'hFFFFBC281;
8'h76: triangl_dist <= 36'hFFFF3C626;
8'h77: triangl_dist <= 36'hFFFFD9D80;
8'h78: triangl_dist <= 36'hFFFFF6B60;
8'h79: triangl_dist <= 36'hFFFEFB775;
8'h7A: triangl_dist <= 36'h000030160;
8'h7B: triangl_dist <= 36'hFFFF585D8;
8'h7C: triangl_dist <= 36'hFFFFC10E5;
8'h7D: triangl_dist <= 36'h00002C6A6;
8'h7E: triangl_dist <= 36'hFFFF6B581;
8'h7F: triangl_dist <= 36'hFFFF86394;
8'h80: triangl_dist <= 36'h00003FF18;
8'h81: triangl_dist <= 36'hFFFF74F07;
8'h82: triangl_dist <= 36'h0000D08EB;
8'h83: triangl_dist <= 36'h0001A0A97;
8'h84: triangl_dist <= 36'h000087EE8;
8'h85: triangl_dist <= 36'hFFFFA89B4;
8'h86: triangl_dist <= 36'h00002D05E;
8'h87: triangl_dist <= 36'hFFFEEDB39;
8'h88: triangl_dist <= 36'h0001225DC;
8'h89: triangl_dist <= 36'h000104CF8;
8'h8A: triangl_dist <= 36'h0000CAE52;
8'h8B: triangl_dist <= 36'hFFFF71B9A;
8'h8C: triangl_dist <= 36'h000032D58;
8'h8D: triangl_dist <= 36'hFFFE6CA46;
8'h8E: triangl_dist <= 36'hFFFFD82F4;
8'h8F: triangl_dist <= 36'hFFFF94E9D;
8'h90: triangl_dist <= 36'hFFFF22F8C;
8'h91: triangl_dist <= 36'hFFFF32253;
8'h92: triangl_dist <= 36'hFFFFD6DD7;
8'h93: triangl_dist <= 36'hFFFEDE44C;
8'h94: triangl_dist <= 36'h000035358;
8'h95: triangl_dist <= 36'hFFFFF0E3F;
8'h96: triangl_dist <= 36'h000070BCF;
8'h97: triangl_dist <= 36'h000073551;
8'h98: triangl_dist <= 36'h00004CAB0;
8'h99: triangl_dist <= 36'hFFFE84BBB;
8'h9A: triangl_dist <= 36'hFFFEBDEEA;
```

```
8'h9B: triangl_dist <= 36'hFFFF99580;
8'h9C: triangl_dist <= 36'h0000100DE;
8'h9D: triangl_dist <= 36'h0000565C0;
8'h9E: triangl_dist <= 36'hFFFFCE49A;
8'h9F: triangl_dist <= 36'h0000CCC8B;
8'hA0: triangl_dist <= 36'h00007FBBB;
8'hA1: triangl_dist <= 36'h00017FCB4;
8'hA2: triangl_dist <= 36'h0000104D7;
8'hA3: triangl_dist <= 36'hFFFF9CE15;
8'hA4: triangl_dist <= 36'hFFFEEB547;
8'hA5: triangl_dist <= 36'h00003C5EB;
8'hA6: triangl_dist <= 36'h0000AB745;
8'hA7: triangl_dist <= 36'hFFFFD72E4;
8'hA8: triangl_dist <= 36'hFFFEDA36F;
8'hA9: triangl_dist <= 36'hFFFF75C67;
8'hAA: triangl_dist <= 36'hFFFF1BD4E;
8'hAB: triangl_dist <= 36'hFFFF7FD54;
8'hAC: triangl_dist <= 36'hFFFFE0585;
8'hAD: triangl_dist <= 36'h00000E175;
8'hAE: triangl_dist <= 36'hFFFFE9B76;
8'hAF: triangl_dist <= 36'h000100618;
8'hB0: triangl_dist <= 36'h00000953E;
8'hB1: triangl_dist <= 36'h000154136;
8'hB2: triangl_dist <= 36'h00004C2C6;
8'hB3: triangl_dist <= 36'h00016B119;
8'hB4: triangl_dist <= 36'hFFFF633F0;
8'hB5: triangl_dist <= 36'h000063ED0;
8'hB6: triangl_dist <= 36'hFFFF854C5;
8'hB7: triangl_dist <= 36'h00006130C;
8'hB8: triangl_dist <= 36'h000070351;
8'hB9: triangl_dist <= 36'hFFFEBCCE6;
8'hBA: triangl_dist <= 36'hFFFF6D7DA;
8'hBB: triangl_dist <= 36'hFFFF56BA0;
8'hBC: triangl_dist <= 36'h00005EAF8;
8'hBD: triangl_dist <= 36'h0000E25F4;
8'hBE: triangl_dist <= 36'hFFFFA8F7C;
8'hBF: triangl_dist <= 36'h0000ACC77;
8'hC0: triangl_dist <= 36'h0000636C6;
8'hC1: triangl_dist <= 36'hFFFE3B560;
8'hC2: triangl_dist <= 36'h0000374C2;
8'hC3: triangl_dist <= 36'hFFFFC24F8;
8'hC4: triangl_dist <= 36'h00012E21B;
8'hC5: triangl_dist <= 36'hFFFE1890E;
8'hC6: triangl_dist <= 36'hFFFFEC661;
8'hC7: triangl_dist <= 36'hFFFFD7ADD;
8'hC8: triangl_dist <= 36'hFFFFEB950;
8'hC9: triangl_dist <= 36'h0000A4DD8;
8'hCA: triangl_dist <= 36'hFFFF9B2DD;
8'hCB: triangl_dist <= 36'h0000B00E4;
8'hCC: triangl_dist <= 36'hFFFFF11E6;
8'hCD: triangl_dist <= 36'hFFFE88EE3;
8'hCE: triangl_dist <= 36'hFFFF2FA8C;
8'hCF: triangl_dist <= 36'h0000820F2;
8'hD0: triangl_dist <= 36'hFFFFF23D7;
8'hD1: triangl_dist <= 36'hFFFF1AF76;
8'hD2: triangl_dist <= 36'hFFFFA6E76;
8'hD3: triangl_dist <= 36'h00003A4D8;
8'hD4: triangl_dist <= 36'hFFFF3D107;
8'hD5: triangl_dist <= 36'h00008DAD2;
8'hD6: triangl_dist <= 36'hFFFF64D2E;
8'hD7: triangl_dist <= 36'h00012FEDE;
```

```
                        8'hD8: triangl_dist <= 36'hFFFF77965;
                        8'hD9: triangl_dist <= 36'h0000A15D3;
                        8'hDA: triangl_dist <= 36'hFFFF3A813;
                        8'hDB: triangl_dist <= 36'hFFFF843DF;
                        8'hDC: triangl_dist <= 36'hFFFEDA901;
                        8'hDD: triangl_dist <= 36'h000028A1A;
                        8'hDE: triangl_dist <= 36'h0000688EE;
                        8'hDF: triangl_dist <= 36'h000018706;
                        8'hE0: triangl_dist <= 36'hFFFFD8720;
                        8'hE1: triangl_dist <= 36'h0000503E2;
                        8'hE2: triangl_dist <= 36'h0000522CB;
                        8'hE3: triangl_dist <= 36'h000065C57;
                        8'hE4: triangl_dist <= 36'h00004B04D;
                        8'hE5: triangl_dist <= 36'h000156753;
                        8'hE6: triangl_dist <= 36'hFFFF4AF8B;
                        8'hE7: triangl_dist <= 36'h000079970;
                        8'hE8: triangl_dist <= 36'hFFFF5FED6;
                        8'hE9: triangl_dist <= 36'hFFFEFA323;
                        8'hEA: triangl_dist <= 36'h00003A40F;
                        8'hEB: triangl_dist <= 36'hFFFFE5CCD;
                        8'hEC: triangl_dist <= 36'hFFFFEA699;
                        8'hED: triangl_dist <= 36'h00005B00D;
                        8'hEE: triangl_dist <= 36'h0000A4F5D;
                        8'hEF: triangl_dist <= 36'hFFFFAC80E;
                        8'hF0: triangl_dist <= 36'h00005B0B2;
                        8'hF1: triangl_dist <= 36'hFFFFD31AE;
                        8'hF2: triangl_dist <= 36'h0000E01ED;
                        8'hF3: triangl_dist <= 36'h0000D8071;
                        8'hF4: triangl_dist <= 36'hFFFF6EAC4;
                        8'hF5: triangl_dist <= 36'h00003DD32;
                        8'hF6: triangl_dist <= 36'h00002C00A;
                        8'hF7: triangl_dist <= 36'h0000154D4;
                        8'hF8: triangl_dist <= 36'h0000FADF0;
                        8'hF9: triangl_dist <= 36'hFFFF74961;
                        8'hFA: triangl_dist <= 36'hFFFF985D9;
                        8'hFB: triangl_dist <= 36'hFFFEFA017;
                        8'hFC: triangl_dist <= 36'h00014E62E;
                        8'hFD: triangl_dist <= 36'h000050EAA;
                        8'hFE: triangl_dist <= 36'hFFFFF5600;
                        8'hFF: triangl_dist <= 36'h00004D244;
                        default:{triangl_dist,rand_addr} <= {triangl_dist,8'h1};
                    endcase
                end
            end

endmodule //dither

// The divider module divides one number by another. It
// produces a signal named "ready" when the quotient output
// is ready, and takes a signal named "start" to indicate
// the the input dividend and divider is ready.
// sign -- 0 for unsigned, 1 for twos complement
//Sabina Maddila
// It uses a simple restoring divide algorithm.
// http://en.wikipedia.org/wiki/Division_(digital)#Restoring_division

module divider #(parameter WIDTH = 8)
  (input clk, sign, start,
   input [WIDTH-1:0] dividend,
   input [WIDTH-1:0] divider,
   output reg [WIDTH-1:0] quotient,
```

```
   output [WIDTH-1:0] remainder,
   output ready);

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

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

   reg [5:0] bit; //may cause problems if not big enough
   reg del_ready = 1;
   assign ready = (!bit) & ~del_ready;

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

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

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

// The divider module divides one number by another. It
// produces a signal named "ready" when the quotient output
// is ready, and takes a signal named "start" to indicate
// the the input dividend and divider is ready.
// sign -- 0 for unsigned, 1 for twos complement
//Sabina Maddila
// It uses a simple restoring divide algorithm.
// http://en.wikipedia.org/wiki/Division_(digital)#Restoring_division

module divider #(parameter WIDTH = 8)
```

```
  (input clk, sign, start,
   input [WIDTH-1:0] dividend,
   input [WIDTH-1:0] divider,
   output reg [WIDTH-1:0] quotient,
   output [WIDTH-1:0] remainder,
   output ready);

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

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

   reg [5:0] bit; //may cause problems if not big enough
   reg del_ready = 1;
   assign ready = (!bit) & ~del_ready;

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

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

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

`timescale 1ns / 1ps
/* Sabina Maddila
/* wave.v
/* modules: wave, interpolate_wave,dither, noteLUT, fwave_sinLUT, qwave_sin_LUT
/* generates the wave tables for a simple sine
/**/
```

```
////////////////////////////////////////////////////////////////////////////////
// module: wave
// generates wavetable for simple sine using integer,truncated theta increments
// **most basic implementation: quickest/least intensive
// **difficult to resolve notes in or lower than the 2nd octave
////////////////////////////////////////////////////////////////////////////////
module wave
  #(parameter BIT_WIDTH=18)
       (input clk_27mhz,
        input ready,
        input reset,
        input [2:0]octave,
        input [3:0]note,
        /*output [9:0]debug_theta,
        /*output [9:0]debug_incr_integr,/**/
        output [2*BIT_WIDTH-1:0]sin_wave);

        reg [11:0]theta=0;

        wire [9:0]incr_integr;
        wire [5:0]incr_fractn;
        wire [2*BIT_WIDTH-1:0]sin;

        assign sin_wave=sin;

        /*assign debug_theta=theta;
        /*assign debug_incr_integr=incr_integr;/**/

        always @(posedge clk_27mhz) begin if (ready) theta <= theta + {2'b0,incr_integr};
        end

        noteLUT thetaINCR(.clk(clk_27mhz),.reset(reset),.ready(ready),
                          .octave(octave),.note(note),
                          .incr_integr(incr_integr),.incr_fractn(incr_fractn));

        fwave_sinLUT  #(.WSIZE(2*BIT_WIDTH))
                      sinSAMPLE(.clk(clk_27mhz),.reset(reset),.ready(ready),
                                                .theta(theta),.sin_out(sin));

endmodule

////////////////////////////////////////////////////////////////////////////////
// module: interpolate_wave
// generates simple sine wave using linear interpolation @ given octave & note
// **linear interpolation smoothens waveform for high frequency notes
// **linear interpolation helps better resolve low frequency notes
////////////////////////////////////////////////////////////////////////////////
module interpolate_wave
  #(parameter WSIZE=36,
    parameter INT_DEPTH=12,
    parameter FRAC_DEPTH=6)
       (input clk_27mhz,
        input ready,
        input reset,
        input [2:0]octave,
        input [3:0]note,
        /*output [WSIZE+FRAC_DEPTH-1:0]debug_sin,
        /*output [INT_DEPTH+FRAC_DEPTH-1:0]debug_phase_accum,
        /*output [INT_DEPTH+FRAC_DEPTH-1:0]debug_phase_delay,
        /*output [FRAC_DEPTH:0]debug_a1,
```

```
	/*output [FRAC_DEPTH:0]debug_a2,
	/*output [WSIZE-1:0]debug_sin1,
	/*output debug_check_conditions,
	/*output [WSIZE-1:0]debug_sin2,/**/
	output [WSIZE-1:0]sin_wave);

	parameter integer CYCL_DELAY=4;
	integer i;

	reg [INT_DEPTH+FRAC_DEPTH-1:0]phase_accum=0; //max 18 bit phase accum
	reg [INT_DEPTH+FRAC_DEPTH-1:0]phase_delay[CYCL_DELAY:0];
	reg signed[WSIZE+FRAC_DEPTH+1:0]sin=0;

	wire signed[WSIZE-1:0]sin1,sin2;
	wire [9:0]incr_integr;
	wire [FRAC_DEPTH-1:0]incr_fractn;

	wire signed[FRAC_DEPTH:0]a2 = {1'b0, phase_delay[CYCL_DELAY][FRAC_DEPTH-1:0]};
	wire signed[FRAC_DEPTH:0]a1 = (1<<FRAC_DEPTH)-a2;

	assign sin_wave=sin[WSIZE+FRAC_DEPTH:FRAC_DEPTH];

	/*assign debug_sin=sin;
	/*assign debug_phase_accum=phase_accum;
	/*assign debug_phase_delay=phase_delay[CYCL_DELAY];
	/*assign debug_a1=a1;
	/*assign debug_a2=a2;
	/*assign debug_sin1=sin1;
	/*assign debug_check_conditions=~|a2;
	/*assign debug_sin2=sin2;/**/

	always @(posedge clk_27mhz) begin
	      if (ready) begin
	         phase_accum <= phase_accum + {incr_integr,incr_fractn};
	            sin <= (sin1*a1[FRAC_DEPTH:0])+(sin2*a2);

	            for (i=CYCL_DELAY; i>0; i=i-1) phase_delay[i] <= phase_delay[i-1];
	            phase_delay[0] <= phase_accum;

	      end
	end

	noteLUT #(.FRAC_DEPTH(FRAC_DEPTH))
	        angINCR(.clk(clk_27mhz),.reset(reset),.ready(ready),.octave(octave),.note(note),
	                .incr_integr(incr_integr),.incr_fractn(incr_fractn));

	fwave_sinLUT #(.WSIZE(WSIZE))
	             sinLO(.clk(clk_27mhz),.reset(reset),.ready(ready),
	                          .theta(phase_accum[11+FRAC_DEPTH:FRAC_DEPTH]),.sin_out(sin1));
	fwave_sinLUT #(.WSIZE(WSIZE))
	             sinHI(.clk(clk_27mhz),.reset(reset),.ready(ready),

	             .theta(phase_accum[11+FRAC_DEPTH:FRAC_DEPTH]+11'b1),.sin_out(sin2));
endmodule

///////////////////////////////////////////////////////////////////////////////
// module: noteLUT
// stores note frequency values & calculates theta incr value
// **input: 6 octaves, 12 notes per octave (+4 redundant notes per octave)
// **mem: values stored << 6 to preserve frac. vals; interpolating oscillator
// **output range: <350, 9 bit-width to >1, 1 bit-width
```

```
// **output bit depth --> 10 bits, allowing for 2s complement
////////////////////////////////////////////////////////////////////////////////
module noteLUT
 #(parameter FRAC_DEPTH=6) //max val=6
  (input clk,
   input ready,
   input reset,
   input [2:0]octave,
      input [3:0]note,
      output [9:0]incr_integr,
      output [FRAC_DEPTH-1:0]incr_fractn);

      reg [15:0]mem_out; // values are << 6 in memory

      assign incr_integr = reset?1: (mem_out>>6);
      assign incr_fractn = reset?0: mem_out[5:(6-FRAC_DEPTH)];

      wire [6:0]addr=reset?7'b0:{octave,note};

      always @(posedge clk) begin
            case(addr)
                  {3'd0,4'd00}: mem_out <= 16'h0059; //C0
                  {3'd0,4'd01}: mem_out <= 16'h005F; //C#0|Db0
                  {3'd0,4'd02}: mem_out <= 16'h0064; //D0
                  {3'd0,4'd03}: mem_out <= 16'h006A; //D#0|Eb0
                  {3'd0,4'd04}: mem_out <= 16'h0071; //E0
                  {3'd0,4'd05}: mem_out <= 16'h0077; //F0
                  {3'd0,4'd06}: mem_out <= 16'h007E; //F#0|Gb0
                  {3'd0,4'd07}: mem_out <= 16'h0086; //G0
                  {3'd0,4'd08}: mem_out <= 16'h008E; //G#0|Ab0
                  {3'd0,4'd09}: mem_out <= 16'h0096; //A0
                  {3'd0,4'd10}: mem_out <= 16'h009F; //A#0|Bb0
                  {3'd0,4'd11}: mem_out <= 16'h00A9; //B0
                  {3'd0,4'd12}: mem_out <= 16'h00B3;
                  {3'd0,4'd13}: mem_out <= 16'h00BD;
                  {3'd0,4'd14}: mem_out <= 16'h00C8;
                  {3'd0,4'd15}: mem_out <= 16'h00D4;
                  {3'd1,4'd00}: mem_out <= 16'h00B3; //C1
                  {3'd1,4'd01}: mem_out <= 16'h00BD; //C#1|Db1
                  {3'd1,4'd02}: mem_out <= 16'h00C8; //D1
                  {3'd1,4'd03}: mem_out <= 16'h00D4; //D#1|Eb1
                  {3'd1,4'd04}: mem_out <= 16'h00E1; //E1
                  {3'd1,4'd05}: mem_out <= 16'h00EE; //F1
                  {3'd1,4'd06}: mem_out <= 16'h00FD; //F#1|Gb1
                  {3'd1,4'd07}: mem_out <= 16'h010C; //G1
                  {3'd1,4'd08}: mem_out <= 16'h011C; //G#1|Ab1
                  {3'd1,4'd09}: mem_out <= 16'h012C; //A1
                  {3'd1,4'd10}: mem_out <= 16'h013E; //A#1|Bb1
                  {3'd1,4'd11}: mem_out <= 16'h0151; //B1
                  {3'd1,4'd12}: mem_out <= 16'h0165;
                  {3'd1,4'd13}: mem_out <= 16'h017A;
                  {3'd1,4'd14}: mem_out <= 16'h0191;
                  {3'd1,4'd15}: mem_out <= 16'h01A9;
                  {3'd2,4'd00}: mem_out <= 16'h0165; //C2
                  {3'd2,4'd01}: mem_out <= 16'h017A; //C#2|Db2
                  {3'd2,4'd02}: mem_out <= 16'h0191; //D2
                  {3'd2,4'd03}: mem_out <= 16'h01A9; //D#2|Eb2
                  {3'd2,4'd04}: mem_out <= 16'h01C2; //E2
                  {3'd2,4'd05}: mem_out <= 16'h01DD; //F2
                  {3'd2,4'd06}: mem_out <= 16'h01F9; //F#2|Gb2
                  {3'd2,4'd07}: mem_out <= 16'h0217; //G2
```

```
{3'd2,4'd08}: mem_out <= 16'h0237; //G#2|Ab2
{3'd2,4'd09}: mem_out <= 16'h0259; //A2
{3'd2,4'd10}: mem_out <= 16'h027C; //A#2|Bb2
{3'd2,4'd11}: mem_out <= 16'h02A2; //B2
{3'd2,4'd12}: mem_out <= 16'h02CA;
{3'd2,4'd13}: mem_out <= 16'h02F5;
{3'd2,4'd14}: mem_out <= 16'h0322;
{3'd2,4'd15}: mem_out <= 16'h0352;
{3'd3,4'd00}: mem_out <= 16'h02CA; //C3
{3'd3,4'd01}: mem_out <= 16'h02F5; //C#3|Db3
{3'd3,4'd02}: mem_out <= 16'h0322; //D3
{3'd3,4'd03}: mem_out <= 16'h0352; //D#3|Eb3
{3'd3,4'd04}: mem_out <= 16'h0384; //E3
{3'd3,4'd05}: mem_out <= 16'h03BA; //F3
{3'd3,4'd06}: mem_out <= 16'h03F2; //F#3|Gb3
{3'd3,4'd07}: mem_out <= 16'h042E; //G3
{3'd3,4'd08}: mem_out <= 16'h046E; //G#3|Ab3
{3'd3,4'd09}: mem_out <= 16'h04B1; //A3
{3'd3,4'd10}: mem_out <= 16'h04F9; //A#3|Bb3
{3'd3,4'd11}: mem_out <= 16'h0545; //B3
{3'd3,4'd12}: mem_out <= 16'h0595;
{3'd3,4'd13}: mem_out <= 16'h05EA;
{3'd3,4'd14}: mem_out <= 16'h0644;
{3'd3,4'd15}: mem_out <= 16'h06A3;
{3'd4,4'd00}: mem_out <= 16'h0595; //C4
{3'd4,4'd01}: mem_out <= 16'h05EA; //C#4|Db4
{3'd4,4'd02}: mem_out <= 16'h0644; //D4
{3'd4,4'd03}: mem_out <= 16'h06A3; //D#4|Eb4
{3'd4,4'd04}: mem_out <= 16'h0708; //E4
{3'd4,4'd05}: mem_out <= 16'h0773; //F4
{3'd4,4'd06}: mem_out <= 16'h07E5; //F#4|Gb4
{3'd4,4'd07}: mem_out <= 16'h085D; //G4
{3'd4,4'd08}: mem_out <= 16'h08DC; //G#4|Ab4
{3'd4,4'd09}: mem_out <= 16'h0963; //A4
{3'd4,4'd10}: mem_out <= 16'h09F2; //A#4|Bb4
{3'd4,4'd11}: mem_out <= 16'h0A89; //B4
{3'd4,4'd12}: mem_out <= 16'h0B2A;
{3'd4,4'd13}: mem_out <= 16'h0BD4;
{3'd4,4'd14}: mem_out <= 16'h0C88;
{3'd4,4'd15}: mem_out <= 16'h0D46;
{3'd5,4'd00}: mem_out <= 16'h0B2A; //C5
{3'd5,4'd01}: mem_out <= 16'h0BD4; //C#5|Db5
{3'd5,4'd02}: mem_out <= 16'h0C88; //D5
{3'd5,4'd03}: mem_out <= 16'h0D46; //D#5|Eb5
{3'd5,4'd04}: mem_out <= 16'h0E10; //E5
{3'd5,4'd05}: mem_out <= 16'h0EE7; //F5
{3'd5,4'd06}: mem_out <= 16'h0FC9; //F#5|Gb5
{3'd5,4'd07}: mem_out <= 16'h10BA; //G5
{3'd5,4'd08}: mem_out <= 16'h11B8; //G#5|Ab5
{3'd5,4'd09}: mem_out <= 16'h12C6; //A5
{3'd5,4'd10}: mem_out <= 16'h13E4; //A#5|Bb5
{3'd5,4'd11}: mem_out <= 16'h1513; //B5
{3'd5,4'd12}: mem_out <= 16'h1653;
{3'd5,4'd13}: mem_out <= 16'h17A7;
{3'd5,4'd14}: mem_out <= 16'h190F;
{3'd5,4'd15}: mem_out <= 16'h1A8D;
{3'd6,4'd00}: mem_out <= 16'h1653; //C6
{3'd6,4'd01}: mem_out <= 16'h17A7; //C#6||Db6
{3'd6,4'd02}: mem_out <= 16'h190F; //D6
{3'd6,4'd03}: mem_out <= 16'h1A8D; //D#6|Eb6
{3'd6,4'd04}: mem_out <= 16'h1C21; //E6
```

```
				{3'd6,4'd05}: mem_out <= 16'h1DCD; //F6
				{3'd6,4'd06}: mem_out <= 16'h1F93; //F#6|Gb6
				{3'd6,4'd07}: mem_out <= 16'h2173; //G6
				{3'd6,4'd08}: mem_out <= 16'h2370; //G#6|Ab6
				{3'd6,4'd09}: mem_out <= 16'h258C; //A6
				{3'd6,4'd10}: mem_out <= 16'h27C8; //A#6|Bb6
				{3'd6,4'd11}: mem_out <= 16'h2A25; //B6
				{3'd6,4'd12}: mem_out <= 16'h2CA7;
				{3'd6,4'd13}: mem_out <= 16'h2F4E;
				{3'd6,4'd14}: mem_out <= 16'h321E;
				{3'd6,4'd15}: mem_out <= 16'h3519;
				{3'd7,4'd00}: mem_out <= 16'h2CA7; //C7
				{3'd7,4'd01}: mem_out <= 16'h2F4E; //C#7|Db7
				{3'd7,4'd02}: mem_out <= 16'h321E; //D7
				{3'd7,4'd03}: mem_out <= 16'h3519; //D#7|Eb7
				{3'd7,4'd04}: mem_out <= 16'h3842; //E7
				{3'd7,4'd05}: mem_out <= 16'h3B9A; //F7
				{3'd7,4'd06}: mem_out <= 16'h3F25; //F#7|Gb7
				{3'd7,4'd07}: mem_out <= 16'h42E7; //G7
				{3'd7,4'd08}: mem_out <= 16'h46E1; //G#7|Ab7
				{3'd7,4'd09}: mem_out <= 16'h4B18; //A7
				{3'd7,4'd10}: mem_out <= 16'h4F8F; //A#7|Bb7
				{3'd7,4'd11}: mem_out <= 16'h544A; //B7
				{3'd7,4'd12}: mem_out <= 16'h4000;
				{3'd7,4'd13}: mem_out <= 16'h4000;
				{3'd7,4'd14}: mem_out <= 16'h4000;
				{3'd7,4'd15}: mem_out <= 16'h0001; //debug vale
				default: mem_out <= 16'h0001;
			endcase
		end
endmodule

////////////////////////////////////////////////////////////////////////////////
// module: fwave_sinLUT
// stores values for sin/cos values given theta-step value
// **there is a two ready-cycle delay on sine output for theta value
// **uses FSM to map quarterwave  to a fullwave sine LUT
////////////////////////////////////////////////////////////////////////////////
module fwave_sinLUT
 #(parameter WSIZE=36)
  (input clk,
   input ready,
	input reset,
	input [11:0]theta,
	/*output [9:0]debug_sin_addr,
	/*output [1:0]debug_quad,
	/*output [1:0]debug_quad_hld,
	/*output debug_zero_xing,/**/
	output [WSIZE-1:0]sin_out);

	parameter QUAD1=2'b00; //angle range: [pi/4096        , pi/2]
	parameter QUAD2=2'b01; //angle range: [pi/2+pi/4096   , pi]
	parameter QUAD3=2'b10; //angle range: [pi+pi/4096     , 3*pi/2]
	parameter QUAD4=2'b11; //angle range: [3*pi/2+pi/4096 , 2*pi]

	parameter integer CYCL_DELAY=2;
	integer i;

	wire [1:0]quad=theta[11:10];
	wire [WSIZE-1:0]sin_mem;
	wire zero_xing, pre_insrt0;
```

```
	reg [WSIZE-1:0]sin_val;
	reg [9:0]sin_addr;
	reg [1:0]quad_hld[CYCL_DELAY:0];
	reg insrt0;

	assign sin_out=reset?0:sin_val;
	assign pre_insrt0=zero_xing&&(~|theta[9:0]);
	assign zero_xing=quad[1]^quad_hld[1][1];

	/*assign debug_sin_addr=sin_addr;
	/*assign debug_quad=quad;
	/*assign debug_quad_hld=quad_hld[2];
	/*assign debug_zero_xing=zero_xing;/**/

	always @(posedge clk) begin
	   if (ready) begin
	      case(quad)
			QUAD1:begin
				sin_addr <= reset?0:pre_insrt0?0:theta[9:0]-1;
				sin_val <= insrt0?0:zero_xing?~{1'b0,sin_mem[WSIZE-1:1]}+1: //val
delayed by two clock cycles
						{1'b0,sin_mem[WSIZE-1:1]};
				insrt0 <= zero_xing&&(~|sin_addr);
				end
			QUAD2:begin
				sin_addr <= (~theta[9:0])-1;
				sin_val <= {1'b0,sin_mem[WSIZE-1:1]};
				end
			QUAD3:begin
				sin_addr <= reset?0:pre_insrt0?0:theta[9:0]-1;
				sin_val <= insrt0?0:zero_xing?{1'b0,sin_mem[WSIZE-1:1]}: //val
delayed by two clock cycles
							~{1'b0,sin_mem[WSIZE-1:1]}+1;
				insrt0 <= zero_xing&&(~|sin_addr);
				end
			QUAD4:begin
				sin_addr <= (~theta[9:0])-1;
				sin_val <= {1'b1,~sin_mem[WSIZE-1:1]}+1;
				end
			default: {sin_addr,sin_val} <= {sin_addr,sin_val};
		endcase

		for (i=CYCL_DELAY; i>0; i=i-1) quad_hld[i] <= quad_hld[i-1];
		quad_hld[0] <= quad;

	    end
	end

	qwave_sinLUT #(.WSIZE(WSIZE))
		qsin(.clk(clk),.ready(ready),.reset(reset),.theta(sin_addr),
			.sin_mem(sin_mem));

endmodule

////////////////////////////////////////////////////////////////////////////////
// module: qwave_sinLUT
// stores values for sin/cos values given theta-step value
// **adapted from Verliog provided by Michael Price
// **1024 memory addresses, mapping over range of [0,pi/2]
// **mapping quarterwave lowers wavetable noise w/ same memory size
```

```
////////////////////////////////////////////////////////////////////////////////
module qwave_sinLUT
 #(parameter WSIZE=36)
  (input clk,
   input ready,
       input reset,
       input [9:0]theta,
       output [WSIZE-1:0]sin_mem);

       reg [35:0]mem_out;
       assign sin_mem=reset?0:mem_out[35:36-WSIZE];

       always @(posedge clk) begin
              if (ready) begin
                     case(theta)
            10'h000: mem_out <= 36'h003243F55;
            10'h001: mem_out <= 36'h006487E2F;
            10'h002: mem_out <= 36'h0096CBC11;
            10'h003: mem_out <= 36'h00C90F87F;
            10'h004: mem_out <= 36'h00FB532FC;
            10'h005: mem_out <= 36'h012D96B0E;
            10'h006: mem_out <= 36'h015FDA037;
            10'h007: mem_out <= 36'h01921D1FC;
            10'h008: mem_out <= 36'h01C45FFE1;
            10'h009: mem_out <= 36'h01F6A296A;
            10'h00A: mem_out <= 36'h0228E4E1B;
            10'h00B: mem_out <= 36'h025B26D77;
            10'h00C: mem_out <= 36'h028D68703;
            10'h00D: mem_out <= 36'h02BFA9A43;
            10'h00E: mem_out <= 36'h02F1EA6BB;
            10'h00F: mem_out <= 36'h03242ABEF;
            10'h010: mem_out <= 36'h03566A962;
            10'h011: mem_out <= 36'h0388A9E99;
            10'h012: mem_out <= 36'h03BAE8B18;
            10'h013: mem_out <= 36'h03ED26E63;
            10'h014: mem_out <= 36'h041F647FE;
            10'h015: mem_out <= 36'h0451A176D;
            10'h016: mem_out <= 36'h0483DDC33;
            10'h017: mem_out <= 36'h04B6195D6;
            10'h018: mem_out <= 36'h04E8543D9;
            10'h019: mem_out <= 36'h051A8E5BF;
            10'h01A: mem_out <= 36'h054CC7B0E;
            10'h01B: mem_out <= 36'h057F0034A;
            10'h01C: mem_out <= 36'h05B137DF5;
            10'h01D: mem_out <= 36'h05E36EA96;
            10'h01E: mem_out <= 36'h0615A48AE;
            10'h01F: mem_out <= 36'h0647D97C4;
            10'h020: mem_out <= 36'h067A0D75A;
            10'h021: mem_out <= 36'h06AC406F5;
            10'h022: mem_out <= 36'h06DE72619;
            10'h023: mem_out <= 36'h0710A344B;
            10'h024: mem_out <= 36'h0742D310D;
            10'h025: mem_out <= 36'h077501BE6;
            10'h026: mem_out <= 36'h07A72F457;
            10'h027: mem_out <= 36'h07D95B9E7;
            10'h028: mem_out <= 36'h080B86C19;
            10'h029: mem_out <= 36'h083DB0A72;
            10'h02A: mem_out <= 36'h086FD9475;
            10'h02B: mem_out <= 36'h08A2009A6;
            10'h02C: mem_out <= 36'h08D42698B;
            10'h02D: mem_out <= 36'h09064B3A7;
```

```
10'h02E: mem_out <= 36'h09386E77E;
10'h02F: mem_out <= 36'h096A90496;
10'h030: mem_out <= 36'h099CB0A72;
10'h031: mem_out <= 36'h09CECF896;
10'h032: mem_out <= 36'h0A00ECE87;
10'h033: mem_out <= 36'h0A3308BC9;
10'h034: mem_out <= 36'h0A6522FE0;
10'h035: mem_out <= 36'h0A973BA52;
10'h036: mem_out <= 36'h0AC952AA2;
10'h037: mem_out <= 36'h0AFB68054;
10'h038: mem_out <= 36'h0B2D7BAEE;
10'h039: mem_out <= 36'h0B5F8D9F3;
10'h03A: mem_out <= 36'h0B919DCE9;
10'h03B: mem_out <= 36'h0BC3AC352;
10'h03C: mem_out <= 36'h0BF5B8CB5;
10'h03D: mem_out <= 36'h0C27C3896;
10'h03E: mem_out <= 36'h0C59CC678;
10'h03F: mem_out <= 36'h0C8BD35E1;
10'h040: mem_out <= 36'h0CBDD8655;
10'h041: mem_out <= 36'h0CEFDB759;
10'h042: mem_out <= 36'h0D21DC871;
10'h043: mem_out <= 36'h0D53DB922;
10'h044: mem_out <= 36'h0D85D88F0;
10'h045: mem_out <= 36'h0DB7D3761;
10'h046: mem_out <= 36'h0DE9CC3F9;
10'h047: mem_out <= 36'h0E1BC2E3C;
10'h048: mem_out <= 36'h0E4DB75B0;
10'h049: mem_out <= 36'h0E7FA99D9;
10'h04A: mem_out <= 36'h0EB199A3C;
10'h04B: mem_out <= 36'h0EE38765D;
10'h04C: mem_out <= 36'h0F1572DC2;
10'h04D: mem_out <= 36'h0F475BFEF;
10'h04E: mem_out <= 36'h0F7942C69;
10'h04F: mem_out <= 36'h0FAB272B5;
10'h050: mem_out <= 36'h0FDD09258;
10'h051: mem_out <= 36'h100EE8AD6;
10'h052: mem_out <= 36'h1040C5BB6;
10'h053: mem_out <= 36'h1072A047B;
10'h054: mem_out <= 36'h10A4784AB;
10'h055: mem_out <= 36'h10D64DBCB;
10'h056: mem_out <= 36'h11082095F;
10'h057: mem_out <= 36'h1139F0CED;
10'h058: mem_out <= 36'h116BBE5FA;
10'h059: mem_out <= 36'h119D8940B;
10'h05A: mem_out <= 36'h11CF516A6;
10'h05B: mem_out <= 36'h120116D4E;
10'h05C: mem_out <= 36'h1232D978A;
10'h05D: mem_out <= 36'h1264994DF;
10'h05E: mem_out <= 36'h1296564D2;
10'h05F: mem_out <= 36'h12C8106E8;
10'h060: mem_out <= 36'h12F9C7AA7;
10'h061: mem_out <= 36'h132B7BF94;
10'h062: mem_out <= 36'h135D2D534;
10'h063: mem_out <= 36'h138EDBB0C;
10'h064: mem_out <= 36'h13C0870A3;
10'h065: mem_out <= 36'h13F22F57D;
10'h066: mem_out <= 36'h1423D4920;
10'h067: mem_out <= 36'h145576B12;
10'h068: mem_out <= 36'h148715AD8;
10'h069: mem_out <= 36'h14B8B17F7;
10'h06A: mem_out <= 36'h14EA4A1F6;
```

```
10'h06B: mem_out <= 36'h151BDF859;
10'h06C: mem_out <= 36'h154D71AA7;
10'h06D: mem_out <= 36'h157F00865;
10'h06E: mem_out <= 36'h15B08C119;
10'h06F: mem_out <= 36'h15E214448;
10'h070: mem_out <= 36'h161399179;
10'h071: mem_out <= 36'h16451A831;
10'h072: mem_out <= 36'h1676987F7;
10'h073: mem_out <= 36'h16A81304F;
10'h074: mem_out <= 36'h16D98A0C0;
10'h075: mem_out <= 36'h170AFD8D0;
10'h076: mem_out <= 36'h173C6D805;
10'h077: mem_out <= 36'h176DD9DE5;
10'h078: mem_out <= 36'h179F429F5;
10'h079: mem_out <= 36'h17D0A7BBD;
10'h07A: mem_out <= 36'h1802092C1;
10'h07B: mem_out <= 36'h183366E89;
10'h07C: mem_out <= 36'h1864C0E9B;
10'h07D: mem_out <= 36'h18961727C;
10'h07E: mem_out <= 36'h18C7699B3;
10'h07F: mem_out <= 36'h18F8B83C6;
10'h080: mem_out <= 36'h192A0303C;
10'h081: mem_out <= 36'h195B49E9B;
10'h082: mem_out <= 36'h198C8CE69;
10'h083: mem_out <= 36'h19BDCBF2D;
10'h084: mem_out <= 36'h19EF0706E;
10'h085: mem_out <= 36'h1A203E1B1;
10'h086: mem_out <= 36'h1A517127E;
10'h087: mem_out <= 36'h1A82A025B;
10'h088: mem_out <= 36'h1AB3CB0CE;
10'h089: mem_out <= 36'h1AE4F1D5F;
10'h08A: mem_out <= 36'h1B1614794;
10'h08B: mem_out <= 36'h1B4732EF3;
10'h08C: mem_out <= 36'h1B784D305;
10'h08D: mem_out <= 36'h1BA96334F;
10'h08E: mem_out <= 36'h1BDA74F58;
10'h08F: mem_out <= 36'h1C0B826A7;
10'h090: mem_out <= 36'h1C3C8B8C4;
10'h091: mem_out <= 36'h1C6D90535;
10'h092: mem_out <= 36'h1C9E90B82;
10'h093: mem_out <= 36'h1CCF8CB31;
10'h094: mem_out <= 36'h1D00843C9;
10'h095: mem_out <= 36'h1D31774D2;
10'h096: mem_out <= 36'h1D6265DD3;
10'h097: mem_out <= 36'h1D934FE54;
10'h098: mem_out <= 36'h1DC4355DB;
10'h099: mem_out <= 36'h1DF5163F0;
10'h09A: mem_out <= 36'h1E25F281A;
10'h09B: mem_out <= 36'h1E56CA1E1;
10'h09C: mem_out <= 36'h1E879D0CC;
10'h09D: mem_out <= 36'h1EB86B462;
10'h09E: mem_out <= 36'h1EE934C2D;
10'h09F: mem_out <= 36'h1F19F97B2;
10'h0A0: mem_out <= 36'h1F4AB9679;
10'h0A1: mem_out <= 36'h1F7B7480B;
10'h0A2: mem_out <= 36'h1FAC2ABEF;
10'h0A3: mem_out <= 36'h1FDCDC1AD;
10'h0A4: mem_out <= 36'h200D888CD;
10'h0A5: mem_out <= 36'h203E300D7;
10'h0A6: mem_out <= 36'h206ED2952;
10'h0A7: mem_out <= 36'h209F701C6;
```

```
10'h0A8: mem_out <= 36'h20D0089BD;
10'h0A9: mem_out <= 36'h21009C0BD;
10'h0AA: mem_out <= 36'h21312A64F;
10'h0AB: mem_out <= 36'h2161B39FB;
10'h0AC: mem_out <= 36'h219237B49;
10'h0AD: mem_out <= 36'h21C2B69C2;
10'h0AE: mem_out <= 36'h21F3304EE;
10'h0AF: mem_out <= 36'h2223A4C56;
10'h0B0: mem_out <= 36'h225413F81;
10'h0B1: mem_out <= 36'h22847DDF8;
10'h0B2: mem_out <= 36'h22B4E2745;
10'h0B3: mem_out <= 36'h22E541AEE;
10'h0B4: mem_out <= 36'h23159B87E;
10'h0B5: mem_out <= 36'h2345EFF7D;
10'h0B6: mem_out <= 36'h23763EF73;
10'h0B7: mem_out <= 36'h23A6887E9;
10'h0B8: mem_out <= 36'h23D6CC869;
10'h0B9: mem_out <= 36'h24070B07A;
10'h0BA: mem_out <= 36'h243743FA7;
10'h0BB: mem_out <= 36'h246777578;
10'h0BC: mem_out <= 36'h2497A5176;
10'h0BD: mem_out <= 36'h24C7CD32A;
10'h0BE: mem_out <= 36'h24F7EFA1E;
10'h0BF: mem_out <= 36'h25280C5DA;
10'h0C0: mem_out <= 36'h255823500;
10'h0C1: mem_out <= 36'h2588349D2;
10'h0C2: mem_out <= 36'h25B840120;
10'h0C3: mem_out <= 36'h25E845B5D;
10'h0C4: mem_out <= 36'h261845811;
10'h0C5: mem_out <= 36'h26483F6C7;
10'h0C6: mem_out <= 36'h267833707;
10'h0C7: mem_out <= 36'h26A82185C;
10'h0C8: mem_out <= 36'h26D809A4F;
10'h0C9: mem_out <= 36'h2707EBC6A;
10'h0CA: mem_out <= 36'h2737C7E37;
10'h0CB: mem_out <= 36'h27679DF40;
10'h0CC: mem_out <= 36'h27976DF0F;
10'h0CD: mem_out <= 36'h27C737D2D;
10'h0CE: mem_out <= 36'h27F6FB926;
10'h0CF: mem_out <= 36'h2826B9282;
10'h0D0: mem_out <= 36'h2856708CD;
10'h0D1: mem_out <= 36'h288621B91;
10'h0D2: mem_out <= 36'h28B5CCA57;
10'h0D3: mem_out <= 36'h28E5714AA;
10'h0D4: mem_out <= 36'h29150FA16;
10'h0D5: mem_out <= 36'h2944A7A23;
10'h0D6: mem_out <= 36'h29743945D;
10'h0D7: mem_out <= 36'h29A3C484F;
10'h0D8: mem_out <= 36'h29D349582;
10'h0D9: mem_out <= 36'h2A02C7B83;
10'h0DA: mem_out <= 36'h2A323F9DB;
10'h0DB: mem_out <= 36'h2A61B1015;
10'h0DC: mem_out <= 36'h2A911BDBD;
10'h0DD: mem_out <= 36'h2AC08025E;
10'h0DE: mem_out <= 36'h2AEFDDD82;
10'h0DF: mem_out <= 36'h2B1F34EB5;
10'h0E0: mem_out <= 36'h2B4E85581;
10'h0E1: mem_out <= 36'h2B7DCF173;
10'h0E2: mem_out <= 36'h2BAD12215;
10'h0E3: mem_out <= 36'h2BDC4E6F3;
10'h0E4: mem_out <= 36'h2C0B83F99;
```

```
10'h0E5: mem_out <= 36'h2C3AB2B91;
10'h0E6: mem_out <= 36'h2C69DAA67;
10'h0E7: mem_out <= 36'h2C98FBBA7;
10'h0E8: mem_out <= 36'h2CC815EDE;
10'h0E9: mem_out <= 36'h2CF729395;
10'h0EA: mem_out <= 36'h2D263595A;
10'h0EB: mem_out <= 36'h2D553AFB8;
10'h0EC: mem_out <= 36'h2D843963C;
10'h0ED: mem_out <= 36'h2DB330C71;
10'h0EE: mem_out <= 36'h2DE2211E3;
10'h0EF: mem_out <= 36'h2E110A61F;
10'h0F0: mem_out <= 36'h2E3FEC8B0;
10'h0F1: mem_out <= 36'h2E6EC7924;
10'h0F2: mem_out <= 36'h2E9D9B707;
10'h0F3: mem_out <= 36'h2ECC681E4;
10'h0F4: mem_out <= 36'h2EFB2D949;
10'h0F5: mem_out <= 36'h2F29EBCC2;
10'h0F6: mem_out <= 36'h2F58A2BDC;
10'h0F7: mem_out <= 36'h2F8752623;
10'h0F8: mem_out <= 36'h2FB5FAB25;
10'h0F9: mem_out <= 36'h2FE49BA6E;
10'h0FA: mem_out <= 36'h30133538B;
10'h0FB: mem_out <= 36'h3041C7609;
10'h0FC: mem_out <= 36'h307052176;
10'h0FD: mem_out <= 36'h309ED555E;
10'h0FE: mem_out <= 36'h30CD5114F;
10'h0FF: mem_out <= 36'h30FBC54D5;
10'h100: mem_out <= 36'h312A31F7F;
10'h101: mem_out <= 36'h3158970DA;
10'h102: mem_out <= 36'h3186F4873;
10'h103: mem_out <= 36'h31B54A5D9;
10'h104: mem_out <= 36'h31E398897;
10'h105: mem_out <= 36'h3211DF03D;
10'h106: mem_out <= 36'h32401DC59;
10'h107: mem_out <= 36'h326E54C77;
10'h108: mem_out <= 36'h329C84026;
10'h109: mem_out <= 36'h32CAAB6F5;
10'h10A: mem_out <= 36'h32F8CB071;
10'h10B: mem_out <= 36'h3326E2C28;
10'h10C: mem_out <= 36'h3354F29A8;
10'h10D: mem_out <= 36'h3382FA881;
10'h10E: mem_out <= 36'h33B0FA840;
10'h10F: mem_out <= 36'h33DEF2875;
10'h110: mem_out <= 36'h340CE28AC;
10'h111: mem_out <= 36'h343ACA876;
10'h112: mem_out <= 36'h3468AA761;
10'h113: mem_out <= 36'h3496824FB;
10'h114: mem_out <= 36'h34C4520D4;
10'h115: mem_out <= 36'h34F219A7B;
10'h116: mem_out <= 36'h351FD917E;
10'h117: mem_out <= 36'h354D9056D;
10'h118: mem_out <= 36'h357B3F5D7;
10'h119: mem_out <= 36'h35A8E624B;
10'h11A: mem_out <= 36'h35D684A59;
10'h11B: mem_out <= 36'h36041AD8F;
10'h11C: mem_out <= 36'h3631A8B7F;
10'h11D: mem_out <= 36'h365F2E3B6;
10'h11E: mem_out <= 36'h368CAB5C5;
10'h11F: mem_out <= 36'h36BA2013C;
10'h120: mem_out <= 36'h36E78C5AA;
10'h121: mem_out <= 36'h3714F029F;
```

```
10'h122: mem_out <= 36'h37424B7AC;
10'h123: mem_out <= 36'h376F9E460;
10'h124: mem_out <= 36'h379CE884C;
10'h125: mem_out <= 36'h37CA2A2FF;
10'h126: mem_out <= 36'h37F76340B;
10'h127: mem_out <= 36'h382493B00;
10'h128: mem_out <= 36'h3851BB76D;
10'h129: mem_out <= 36'h387EDA8E4;
10'h12A: mem_out <= 36'h38ABF0EF6;
10'h12B: mem_out <= 36'h38D8FE932;
10'h12C: mem_out <= 36'h39060372B;
10'h12D: mem_out <= 36'h3932FF870;
10'h12E: mem_out <= 36'h395FF2C94;
10'h12F: mem_out <= 36'h398CDD326;
10'h130: mem_out <= 36'h39B9BEBB8;
10'h131: mem_out <= 36'h39E6975DC;
10'h132: mem_out <= 36'h3A1367122;
10'h133: mem_out <= 36'h3A402DD1D;
10'h134: mem_out <= 36'h3A6CEB95D;
10'h135: mem_out <= 36'h3A99A0575;
10'h136: mem_out <= 36'h3AC64C0F5;
10'h137: mem_out <= 36'h3AF2EEB70;
10'h138: mem_out <= 36'h3B1F88478;
10'h139: mem_out <= 36'h3B4C18B9F;
10'h13A: mem_out <= 36'h3B78A0076;
10'h13B: mem_out <= 36'h3BA51E290;
10'h13C: mem_out <= 36'h3BD19317F;
10'h13D: mem_out <= 36'h3BFDFECD5;
10'h13E: mem_out <= 36'h3C2A61424;
10'h13F: mem_out <= 36'h3C56BA700;
10'h140: mem_out <= 36'h3C830A4FB;
10'h141: mem_out <= 36'h3CAF50DA7;
10'h142: mem_out <= 36'h3CDB8E097;
10'h143: mem_out <= 36'h3D07C1D5F;
10'h144: mem_out <= 36'h3D33EC390;
10'h145: mem_out <= 36'h3D600D2BE;
10'h146: mem_out <= 36'h3D8C24A7C;
10'h147: mem_out <= 36'h3DB832A5D;
10'h148: mem_out <= 36'h3DE4371F5;
10'h149: mem_out <= 36'h3E10320D7;
10'h14A: mem_out <= 36'h3E3C23697;
10'h14B: mem_out <= 36'h3E680B2C7;
10'h14C: mem_out <= 36'h3E93E94FD;
10'h14D: mem_out <= 36'h3EBFBDCCA;
10'h14E: mem_out <= 36'h3EEB889C4;
10'h14F: mem_out <= 36'h3F1749B7F;
10'h150: mem_out <= 36'h3F430118D;
10'h151: mem_out <= 36'h3F6EAEB84;
10'h152: mem_out <= 36'h3F9A528F8;
10'h153: mem_out <= 36'h3FC5EC97C;
10'h154: mem_out <= 36'h3FF17CCA6;
10'h155: mem_out <= 36'h401D0320A;
10'h156: mem_out <= 36'h40487F93D;
10'h157: mem_out <= 36'h4073F21D3;
10'h158: mem_out <= 36'h409F5AB60;
10'h159: mem_out <= 36'h40CAB957B;
10'h15A: mem_out <= 36'h40F60DFB7;
10'h15B: mem_out <= 36'h4121589AB;
10'h15C: mem_out <= 36'h414C992EB;
10'h15D: mem_out <= 36'h4177CFB0C;
10'h15E: mem_out <= 36'h41A2FC1A4;
```

```
10'h15F: mem_out <= 36'h41CE1E648;
10'h160: mem_out <= 36'h41F93688E;
10'h161: mem_out <= 36'h42244480C;
10'h162: mem_out <= 36'h424F48457;
10'h163: mem_out <= 36'h427A41D05;
10'h164: mem_out <= 36'h42A5311AD;
10'h165: mem_out <= 36'h42D0161E3;
10'h166: mem_out <= 36'h42FAF0D3F;
10'h167: mem_out <= 36'h4325C1357;
10'h168: mem_out <= 36'h4350873C0;
10'h169: mem_out <= 36'h437B42E12;
10'h16A: mem_out <= 36'h43A5F41E3;
10'h16B: mem_out <= 36'h43D09AECA;
10'h16C: mem_out <= 36'h43FB3745D;
10'h16D: mem_out <= 36'h4425C9234;
10'h16E: mem_out <= 36'h445050700;
10'h16F: mem_out <= 36'h447ACD506;
10'h170: mem_out <= 36'h44A53F931;
10'h171: mem_out <= 36'h44CFA73FB;
10'h172: mem_out <= 36'h44FA044FC;
10'h173: mem_out <= 36'h452456BCC;
10'h174: mem_out <= 36'h454E9E802;
10'h175: mem_out <= 36'h4578DB936;
10'h176: mem_out <= 36'h45A30DF00;
10'h177: mem_out <= 36'h45CD358F7;
10'h178: mem_out <= 36'h45F7526B4;
10'h179: mem_out <= 36'h4621647CE;
10'h17A: mem_out <= 36'h464B6BBDE;
10'h17B: mem_out <= 36'h46756827C;
10'h17C: mem_out <= 36'h469F59B41;
10'h17D: mem_out <= 36'h46C9405C4;
10'h17E: mem_out <= 36'h46F31C1A0;
10'h17F: mem_out <= 36'h471CECE6B;
10'h180: mem_out <= 36'h4746B2BC0;
10'h181: mem_out <= 36'h47706D937;
10'h182: mem_out <= 36'h479A1D668;
10'h183: mem_out <= 36'h47C3C22EF;
10'h184: mem_out <= 36'h47ED5BE62;
10'h185: mem_out <= 36'h4816EA85D;
10'h186: mem_out <= 36'h48406E078;
10'h187: mem_out <= 36'h4869E664C;
10'h188: mem_out <= 36'h489353975;
10'h189: mem_out <= 36'h48BCB598B;
10'h18A: mem_out <= 36'h48E60C627;
10'h18B: mem_out <= 36'h490F57EE6;
10'h18C: mem_out <= 36'h49389835F;
10'h18D: mem_out <= 36'h4961CD32E;
10'h18E: mem_out <= 36'h498AF6DED;
10'h18F: mem_out <= 36'h49B415337;
10'h190: mem_out <= 36'h49DD282A5;
10'h191: mem_out <= 36'h4A062FBD3;
10'h192: mem_out <= 36'h4A2F2BE5B;
10'h193: mem_out <= 36'h4A581C9D8;
10'h194: mem_out <= 36'h4A8101DE6;
10'h195: mem_out <= 36'h4AA9DBA1E;
10'h196: mem_out <= 36'h4AD2A9E1D;
10'h197: mem_out <= 36'h4AFB6C97E;
10'h198: mem_out <= 36'h4B2423BDC;
10'h199: mem_out <= 36'h4B4CCF4D3;
10'h19A: mem_out <= 36'h4B756F3FE;
10'h19B: mem_out <= 36'h4B9E038FA;
```

```
10'h19C: mem_out <= 36'h4BC68C361;
10'h19D: mem_out <= 36'h4BEF092D1;
10'h19E: mem_out <= 36'h4C177A6E4;
10'h19F: mem_out <= 36'h4C3FDFF38;
10'h1A0: mem_out <= 36'h4C6839B68;
10'h1A1: mem_out <= 36'h4C9087B12;
10'h1A2: mem_out <= 36'h4CB8C9DD1;
10'h1A3: mem_out <= 36'h4CE100343;
10'h1A4: mem_out <= 36'h4D092AB03;
10'h1A5: mem_out <= 36'h4D31494B0;
10'h1A6: mem_out <= 36'h4D595BFE5;
10'h1A7: mem_out <= 36'h4D8162C41;
10'h1A8: mem_out <= 36'h4DA95D960;
10'h1A9: mem_out <= 36'h4DD14C6E0;
10'h1AA: mem_out <= 36'h4DF92F45E;
10'h1AB: mem_out <= 36'h4E2106177;
10'h1AC: mem_out <= 36'h4E48D0DCB;
10'h1AD: mem_out <= 36'h4E708F8F5;
10'h1AE: mem_out <= 36'h4E9842295;
10'h1AF: mem_out <= 36'h4EBFE8A48;
10'h1B0: mem_out <= 36'h4EE782FAC;
10'h1B1: mem_out <= 36'h4F0F11260;
10'h1B2: mem_out <= 36'h4F3693202;
10'h1B3: mem_out <= 36'h4F5E08E31;
10'h1B4: mem_out <= 36'h4F857268B;
10'h1B5: mem_out <= 36'h4FACCFAAF;
10'h1B6: mem_out <= 36'h4FD420A3C;
10'h1B7: mem_out <= 36'h4FFB654D1;
10'h1B8: mem_out <= 36'h50229DA0D;
10'h1B9: mem_out <= 36'h5049C998F;
10'h1BA: mem_out <= 36'h5070E92F6;
10'h1BB: mem_out <= 36'h5097FC5E3;
10'h1BC: mem_out <= 36'h50BF031F4;
10'h1BD: mem_out <= 36'h50E5FD6CA;
10'h1BE: mem_out <= 36'h510CEB404;
10'h1BF: mem_out <= 36'h5133CC942;
10'h1C0: mem_out <= 36'h515AA1624;
10'h1C1: mem_out <= 36'h518169A4A;
10'h1C2: mem_out <= 36'h51A825555;
10'h1C3: mem_out <= 36'h51CED46E6;
10'h1C4: mem_out <= 36'h51F576E9B;
10'h1C5: mem_out <= 36'h521C0CC18;
10'h1C6: mem_out <= 36'h524295EFB;
10'h1C7: mem_out <= 36'h526912600;
10'h1C8: mem_out <= 36'h528F8237B;
10'h1C9: mem_out <= 36'h52B5E5459;
10'h1CA: mem_out <= 36'h52DC3B923;
10'h1CB: mem_out <= 36'h53028517B;
10'h1CC: mem_out <= 36'h5328C1D00;
10'h1CD: mem_out <= 36'h534EF1B56;
10'h1CE: mem_out <= 36'h537514C1D;
10'h1CF: mem_out <= 36'h539B2AEF8;
10'h1D0: mem_out <= 36'h53C13438A;
10'h1D1: mem_out <= 36'h53E730973;
10'h1D2: mem_out <= 36'h540D20056;
10'h1D3: mem_out <= 36'h5433027D6;
10'h1D4: mem_out <= 36'h5458D7F95;
10'h1D5: mem_out <= 36'h547EA0736;
10'h1D6: mem_out <= 36'h54A45BE5B;
10'h1D7: mem_out <= 36'h54CA0A4A8;
10'h1D8: mem_out <= 36'h54EFAB9C0;
```

```
10'h1D9: mem_out <= 36'h55153FD45;
10'h1DA: mem_out <= 36'h553AC6EDB;
10'h1DB: mem_out <= 36'h556040E25;
10'h1DC: mem_out <= 36'h5585ADAC8;
10'h1DD: mem_out <= 36'h55AB0D465;
10'h1DE: mem_out <= 36'h55D05FAA2;
10'h1DF: mem_out <= 36'h55F5A4D23;
10'h1E0: mem_out <= 36'h561ADCB8A;
10'h1E1: mem_out <= 36'h56400757D;
10'h1E2: mem_out <= 36'h566524AA0;
10'h1E3: mem_out <= 36'h568A34A97;
10'h1E4: mem_out <= 36'h56AF37506;
10'h1E5: mem_out <= 36'h56D42C993;
10'h1E6: mem_out <= 36'h56F9147E2;
10'h1E7: mem_out <= 36'h571DEEF99;
10'h1E8: mem_out <= 36'h5742BC05B;
10'h1E9: mem_out <= 36'h57677B9CF;
10'h1EA: mem_out <= 36'h578C2DB9A;
10'h1EB: mem_out <= 36'h57B0D2560;
10'h1EC: mem_out <= 36'h57D5696C9;
10'h1ED: mem_out <= 36'h57F9F2F79;
10'h1EE: mem_out <= 36'h581E6EF16;
10'h1EF: mem_out <= 36'h5842DD547;
10'h1F0: mem_out <= 36'h58673E1B1;
10'h1F1: mem_out <= 36'h588B913FB;
10'h1F2: mem_out <= 36'h58AFD6BCA;
10'h1F3: mem_out <= 36'h58D40E8C7;
10'h1F4: mem_out <= 36'h58F838A96;
10'h1F5: mem_out <= 36'h591C550DF;
10'h1F6: mem_out <= 36'h594063B4A;
10'h1F7: mem_out <= 36'h59646497C;
10'h1F8: mem_out <= 36'h598857B1D;
10'h1F9: mem_out <= 36'h59AC3CFD4;
10'h1FA: mem_out <= 36'h59D014749;
10'h1FB: mem_out <= 36'h59F3DE124;
10'h1FC: mem_out <= 36'h5A1799D0B;
10'h1FD: mem_out <= 36'h5A3B47AA8;
10'h1FE: mem_out <= 36'h5A5EE79A1;
10'h1FF: mem_out <= 36'h5A827999F;
10'h200: mem_out <= 36'h5AA5FDA4B;
10'h201: mem_out <= 36'h5AC973B4B;
10'h202: mem_out <= 36'h5AECDBC4A;
10'h203: mem_out <= 36'h5B1035CF0;
10'h204: mem_out <= 36'h5B3381CE5;
10'h205: mem_out <= 36'h5B56BFBD2;
10'h206: mem_out <= 36'h5B79EF961;
10'h207: mem_out <= 36'h5B9D1153A;
10'h208: mem_out <= 36'h5BC024F07;
10'h209: mem_out <= 36'h5BE32A672;
10'h20A: mem_out <= 36'h5C0621B23;
10'h20B: mem_out <= 36'h5C290ACC5;
10'h20C: mem_out <= 36'h5C4BE5B02;
10'h20D: mem_out <= 36'h5C6EB2583;
10'h20E: mem_out <= 36'h5C9170BF3;
10'h20F: mem_out <= 36'h5CB420DFB;
10'h210: mem_out <= 36'h5CD6C2B48;
10'h211: mem_out <= 36'h5CF956381;
10'h212: mem_out <= 36'h5D1BDB654;
10'h213: mem_out <= 36'h5D3E5236A;
10'h214: mem_out <= 36'h5D60BAA6E;
10'h215: mem_out <= 36'h5D8314B0C;
```

```
10'h216: mem_out <= 36'h5DA5604EE;
10'h217: mem_out <= 36'h5DC79D7C0;
10'h218: mem_out <= 36'h5DE9CC32E;
10'h219: mem_out <= 36'h5E0BEC6E4;
10'h21A: mem_out <= 36'h5E2DFE28C;
10'h21B: mem_out <= 36'h5E50015D3;
10'h21C: mem_out <= 36'h5E71F6066;
10'h21D: mem_out <= 36'h5E93DC1EF;
10'h21E: mem_out <= 36'h5EB5B3A1D;
10'h21F: mem_out <= 36'h5ED77C89A;
10'h220: mem_out <= 36'h5EF936D14;
10'h221: mem_out <= 36'h5F1AE2738;
10'h222: mem_out <= 36'h5F3C7F6B2;
10'h223: mem_out <= 36'h5F5E0DB30;
10'h224: mem_out <= 36'h5F7F8D45F;
10'h225: mem_out <= 36'h5FA0FE1EC;
10'h226: mem_out <= 36'h5FC260384;
10'h227: mem_out <= 36'h5FE3B38D5;
10'h228: mem_out <= 36'h6004F818E;
10'h229: mem_out <= 36'h60262DD5B;
10'h22A: mem_out <= 36'h604754BEB;
10'h22B: mem_out <= 36'h60686CCED;
10'h22C: mem_out <= 36'h60897600E;
10'h22D: mem_out <= 36'h60AA704FD;
10'h22E: mem_out <= 36'h60CB5BB68;
10'h22F: mem_out <= 36'h60EC382FF;
10'h230: mem_out <= 36'h610D05B71;
10'h231: mem_out <= 36'h612DC446B;
10'h232: mem_out <= 36'h614E73D9F;
10'h233: mem_out <= 36'h616F146BA;
10'h234: mem_out <= 36'h618FA5F6C;
10'h235: mem_out <= 36'h61B028766;
10'h236: mem_out <= 36'h61D09BE57;
10'h237: mem_out <= 36'h61F1003EE;
10'h238: mem_out <= 36'h6211557DC;
10'h239: mem_out <= 36'h62319B9D2;
10'h23A: mem_out <= 36'h6251D297E;
10'h23B: mem_out <= 36'h6271FA693;
10'h23C: mem_out <= 36'h6292130C0;
10'h23D: mem_out <= 36'h62B21C7B7;
10'h23E: mem_out <= 36'h62D216B28;
10'h23F: mem_out <= 36'h62F201AC5;
10'h240: mem_out <= 36'h6311DD63E;
10'h241: mem_out <= 36'h6331A9D46;
10'h242: mem_out <= 36'h635166F8D;
10'h243: mem_out <= 36'h637114CC5;
10'h244: mem_out <= 36'h6390B34A1;
10'h245: mem_out <= 36'h63B0426D2;
10'h246: mem_out <= 36'h63CFC230A;
10'h247: mem_out <= 36'h63EF328FB;
10'h248: mem_out <= 36'h640E93859;
10'h249: mem_out <= 36'h642DE50D5;
10'h24A: mem_out <= 36'h644D27223;
10'h24B: mem_out <= 36'h646C59BF5;
10'h24C: mem_out <= 36'h648B7CDFE;
10'h24D: mem_out <= 36'h64AA907F1;
10'h24E: mem_out <= 36'h64C994982;
10'h24F: mem_out <= 36'h64E889264;
10'h250: mem_out <= 36'h65076E24B;
10'h251: mem_out <= 36'h6526438EB;
10'h252: mem_out <= 36'h6545095F7;
```

```
10'h253: mem_out <= 36'h6563BF923;
10'h254: mem_out <= 36'h658266224;
10'h255: mem_out <= 36'h65A0FD0AF;
10'h256: mem_out <= 36'h65BF84477;
10'h257: mem_out <= 36'h65DDFBD31;
10'h258: mem_out <= 36'h65FC63A93;
10'h259: mem_out <= 36'h661ABBC51;
10'h25A: mem_out <= 36'h663904220;
10'h25B: mem_out <= 36'h66573CBB6;
10'h25C: mem_out <= 36'h6675658C7;
10'h25D: mem_out <= 36'h66937E90A;
10'h25E: mem_out <= 36'h66B187C35;
10'h25F: mem_out <= 36'h66CF811FC;
10'h260: mem_out <= 36'h66ED6AA17;
10'h261: mem_out <= 36'h670B4443C;
10'h262: mem_out <= 36'h67290E020;
10'h263: mem_out <= 36'h6746C7D7A;
10'h264: mem_out <= 36'h676471C01;
10'h265: mem_out <= 36'h67820BB6D;
10'h266: mem_out <= 36'h679F95B72;
10'h267: mem_out <= 36'h67BD0FBCA;
10'h268: mem_out <= 36'h67DA79C2B;
10'h269: mem_out <= 36'h67F7D3C4C;
10'h26A: mem_out <= 36'h68151DBE5;
10'h26B: mem_out <= 36'h683257AAE;
10'h26C: mem_out <= 36'h684F8185F;
10'h26D: mem_out <= 36'h686C9B4B0;
10'h26E: mem_out <= 36'h6889A4F58;
10'h26F: mem_out <= 36'h68A69E811;
10'h270: mem_out <= 36'h68C387E93;
10'h271: mem_out <= 36'h68E061296;
10'h272: mem_out <= 36'h68FD2A3D3;
10'h273: mem_out <= 36'h6919E3204;
10'h274: mem_out <= 36'h69368BCE1;
10'h275: mem_out <= 36'h695324424;
10'h276: mem_out <= 36'h696FAC786;
10'h277: mem_out <= 36'h698C246C0;
10'h278: mem_out <= 36'h69A88C18D;
10'h279: mem_out <= 36'h69C4E37A7;
10'h27A: mem_out <= 36'h69E12A8C7;
10'h27B: mem_out <= 36'h69FD614A7;
10'h27C: mem_out <= 36'h6A1987B03;
10'h27D: mem_out <= 36'h6A359DB95;
10'h27E: mem_out <= 36'h6A51A3616;
10'h27F: mem_out <= 36'h6A6D98A43;
10'h280: mem_out <= 36'h6A897D7D6;
10'h281: mem_out <= 36'h6AA551E8B;
10'h282: mem_out <= 36'h6AC115E1C;
10'h283: mem_out <= 36'h6ADCC9645;
10'h284: mem_out <= 36'h6AF86C6C2;
10'h285: mem_out <= 36'h6B13FEF4F;
10'h286: mem_out <= 36'h6B2F80FA8;
10'h287: mem_out <= 36'h6B4AF2788;
10'h288: mem_out <= 36'h6B66536AC;
10'h289: mem_out <= 36'h6B81A3CD1;
10'h28A: mem_out <= 36'h6B9CE39B3;
10'h28B: mem_out <= 36'h6BB812D0F;
10'h28C: mem_out <= 36'h6BD3316A1;
10'h28D: mem_out <= 36'h6BEE3F627;
10'h28E: mem_out <= 36'h6C093CB5F;
10'h28F: mem_out <= 36'h6C2429605;
```

```
10'h290: mem_out <= 36'h6C3F055D7;
10'h291: mem_out <= 36'h6C59D0A93;
10'h292: mem_out <= 36'h6C748B3F6;
10'h293: mem_out <= 36'h6C8F351C0;
10'h294: mem_out <= 36'h6CA9CE3AD;
10'h295: mem_out <= 36'h6CC45697C;
10'h296: mem_out <= 36'h6CDECE2ED;
10'h297: mem_out <= 36'h6CF934FBD;
10'h298: mem_out <= 36'h6D138AFAB;
10'h299: mem_out <= 36'h6D2DD0277;
10'h29A: mem_out <= 36'h6D48047E0;
10'h29B: mem_out <= 36'h6D6227FA4;
10'h29C: mem_out <= 36'h6D7C3A984;
10'h29D: mem_out <= 36'h6D963C53F;
10'h29E: mem_out <= 36'h6DB02D295;
10'h29F: mem_out <= 36'h6DCA0D146;
10'h2A0: mem_out <= 36'h6DE3DC112;
10'h2A1: mem_out <= 36'h6DFD9A1B9;
10'h2A2: mem_out <= 36'h6E17472FD;
10'h2A3: mem_out <= 36'h6E30E349D;
10'h2A4: mem_out <= 36'h6E4A6E65A;
10'h2A5: mem_out <= 36'h6E63E87F6;
10'h2A6: mem_out <= 36'h6E7D51931;
10'h2A7: mem_out <= 36'h6E96A99CD;
10'h2A8: mem_out <= 36'h6EAFF098B;
10'h2A9: mem_out <= 36'h6EC92682D;
10'h2AA: mem_out <= 36'h6EE24B575;
10'h2AB: mem_out <= 36'h6EFB5F124;
10'h2AC: mem_out <= 36'h6F1461AFE;
10'h2AD: mem_out <= 36'h6F2D532C3;
10'h2AE: mem_out <= 36'h6F4633837;
10'h2AF: mem_out <= 36'h6F5F02B1B;
10'h2B0: mem_out <= 36'h6F77C0B34;
10'h2B1: mem_out <= 36'h6F906D844;
10'h2B2: mem_out <= 36'h6FA90920D;
10'h2B3: mem_out <= 36'h6FC193854;
10'h2B4: mem_out <= 36'h6FDA0CADC;
10'h2B5: mem_out <= 36'h6FF274968;
10'h2B6: mem_out <= 36'h700ACB3BC;
10'h2B7: mem_out <= 36'h70231099C;
10'h2B8: mem_out <= 36'h703B44ACC;
10'h2B9: mem_out <= 36'h705367711;
10'h2BA: mem_out <= 36'h706B78E2F;
10'h2BB: mem_out <= 36'h708378FEA;
10'h2BC: mem_out <= 36'h709B67C07;
10'h2BD: mem_out <= 36'h70B34524C;
10'h2BE: mem_out <= 36'h70CB1127D;
10'h2BF: mem_out <= 36'h70E2CBC60;
10'h2C0: mem_out <= 36'h70FA74FB9;
10'h2C1: mem_out <= 36'h71120CC50;
10'h2C2: mem_out <= 36'h712993100;
10'h2C3: mem_out <= 36'h71410804A;
10'h2C4: mem_out <= 36'h71586B73A;
10'h2C5: mem_out <= 36'h716FBD67F;
10'h2C6: mem_out <= 36'h7186FDDDF;
10'h2C7: mem_out <= 36'h719E2CD22;
10'h2C8: mem_out <= 36'h71B54A40D;
10'h2C9: mem_out <= 36'h71CC56267;
10'h2CA: mem_out <= 36'h71E3507F9;
10'h2CB: mem_out <= 36'h71FA39488;
10'h2CC: mem_out <= 36'h7211107DE;
```

```
10'h2CD: mem_out <= 36'h7227D61C0;
10'h2CE: mem_out <= 36'h723E8A1F8;
10'h2CF: mem_out <= 36'h72552C84D;
10'h2D0: mem_out <= 36'h726BBD486;
10'h2D1: mem_out <= 36'h72823C66E;
10'h2D2: mem_out <= 36'h7298A9DCB;
10'h2D3: mem_out <= 36'h72AF05A68;
10'h2D4: mem_out <= 36'h72C54FC0B;
10'h2D5: mem_out <= 36'h72DB88280;
10'h2D6: mem_out <= 36'h72F1AED8E;
10'h2D7: mem_out <= 36'h7307C3CFF;
10'h2D8: mem_out <= 36'h731DC709C;
10'h2D9: mem_out <= 36'h7333B8830;
10'h2DA: mem_out <= 36'h734998384;
10'h2DB: mem_out <= 36'h735F66262;
10'h2DC: mem_out <= 36'h737522495;
10'h2DD: mem_out <= 36'h738ACC9E6;
10'h2DE: mem_out <= 36'h73A065220;
10'h2DF: mem_out <= 36'h73B5EBD0F;
10'h2E0: mem_out <= 36'h73CB60A7C;
10'h2E1: mem_out <= 36'h73E0C3A33;
10'h2E2: mem_out <= 36'h73F614BFF;
10'h2E3: mem_out <= 36'h740B53FAC;
10'h2E4: mem_out <= 36'h742081505;
10'h2E5: mem_out <= 36'h74359CBD6;
10'h2E6: mem_out <= 36'h744AA63EB;
10'h2E7: mem_out <= 36'h745F9DD0F;
10'h2E8: mem_out <= 36'h747483710;
10'h2E9: mem_out <= 36'h7489571B9;
10'h2EA: mem_out <= 36'h749E18CD7;
10'h2EB: mem_out <= 36'h74B2C8838;
10'h2EC: mem_out <= 36'h74C7663A7;
10'h2ED: mem_out <= 36'h74DBF1EF2;
10'h2EE: mem_out <= 36'h74F06B9E7;
10'h2EF: mem_out <= 36'h7504D3453;
10'h2F0: mem_out <= 36'h751928000;
10'h2F1: mem_out <= 36'h752D6C6C6;
10'h2F2: mem_out <= 36'h75419DE68;
10'h2F3: mem_out <= 36'h7555BD4BA;
10'h2F4: mem_out <= 36'h7569CA988;
10'h2F5: mem_out <= 36'h757DC5CA2;
10'h2F6: mem_out <= 36'h7591AEDD5;
10'h2F7: mem_out <= 36'h75A585CF2;
10'h2F8: mem_out <= 36'h75B94A9C7;
10'h2F9: mem_out <= 36'h75CCFD423;
10'h2FA: mem_out <= 36'h75E09DBD5;
10'h2FB: mem_out <= 36'h75F42C0AE;
10'h2FC: mem_out <= 36'h7607A827C;
10'h2FD: mem_out <= 36'h761B12111;
10'h2FE: mem_out <= 36'h762E69C3B;
10'h2FF: mem_out <= 36'h7641AF3CC;
10'h300: mem_out <= 36'h7654E2794;
10'h301: mem_out <= 36'h766803762;
10'h302: mem_out <= 36'h767B12309;
10'h303: mem_out <= 36'h768E0EA59;
10'h304: mem_out <= 36'h76A0F8D23;
10'h305: mem_out <= 36'h76B3D0B39;
10'h306: mem_out <= 36'h76C69646C;
10'h307: mem_out <= 36'h76D94988E;
10'h308: mem_out <= 36'h76EBEA770;
10'h309: mem_out <= 36'h76FE790E5;
```

```
10'h30A: mem_out <= 36'h7710F54BF;
10'h30B: mem_out <= 36'h77235F2D0;
10'h30C: mem_out <= 36'h7735B6AEB;
10'h30D: mem_out <= 36'h7747FBCE2;
10'h30E: mem_out <= 36'h775A2E889;
10'h30F: mem_out <= 36'h776C4EDB3;
10'h310: mem_out <= 36'h777E5CC32;
10'h311: mem_out <= 36'h7790583DB;
10'h312: mem_out <= 36'h77A241481;
10'h313: mem_out <= 36'h77B417DF7;
10'h314: mem_out <= 36'h77C5DC012;
10'h315: mem_out <= 36'h77D78DAA6;
10'h316: mem_out <= 36'h77E92CD88;
10'h317: mem_out <= 36'h77FAB988B;
10'h318: mem_out <= 36'h780C33B85;
10'h319: mem_out <= 36'h781D9B64A;
10'h31A: mem_out <= 36'h782EF08AF;
10'h31B: mem_out <= 36'h78403328A;
10'h31C: mem_out <= 36'h7851633B0;
10'h31D: mem_out <= 36'h786280BF7;
10'h31E: mem_out <= 36'h78738BB34;
10'h31F: mem_out <= 36'h78848413D;
10'h320: mem_out <= 36'h789569DE9;
10'h321: mem_out <= 36'h78A63D10E;
10'h322: mem_out <= 36'h78B6FDA82;
10'h323: mem_out <= 36'h78C7ABA1C;
10'h324: mem_out <= 36'h78D846FB2;
10'h325: mem_out <= 36'h78E8CFB1D;
10'h326: mem_out <= 36'h78F945C32;
10'h327: mem_out <= 36'h7909A92CA;
10'h328: mem_out <= 36'h7919F9EBC;
10'h329: mem_out <= 36'h792A37FE0;
10'h32A: mem_out <= 36'h793A6360D;
10'h32B: mem_out <= 36'h794A7C11C;
10'h32C: mem_out <= 36'h795A820E5;
10'h32D: mem_out <= 36'h796A75541;
10'h32E: mem_out <= 36'h797A55E07;
10'h32F: mem_out <= 36'h798A23B12;
10'h330: mem_out <= 36'h7999DEC39;
10'h331: mem_out <= 36'h79A987157;
10'h332: mem_out <= 36'h79B91CA44;
10'h333: mem_out <= 36'h79C89F6DA;
10'h334: mem_out <= 36'h79D80F6F3;
10'h335: mem_out <= 36'h79E76CA69;
10'h336: mem_out <= 36'h79F6B7115;
10'h337: mem_out <= 36'h7A05EEAD3;
10'h338: mem_out <= 36'h7A151377C;
10'h339: mem_out <= 36'h7A24256EB;
10'h33A: mem_out <= 36'h7A33248FB;
10'h33B: mem_out <= 36'h7A4210D87;
10'h33C: mem_out <= 36'h7A50EA46A;
10'h33D: mem_out <= 36'h7A5FB0D80;
10'h33E: mem_out <= 36'h7A6E648A4;
10'h33F: mem_out <= 36'h7A7D055B1;
10'h340: mem_out <= 36'h7A8B93484;
10'h341: mem_out <= 36'h7A9A0E4F9;
10'h342: mem_out <= 36'h7AA8766EC;
10'h343: mem_out <= 36'h7AB6CBA39;
10'h344: mem_out <= 36'h7AC50DEBE;
10'h345: mem_out <= 36'h7AD33D456;
10'h346: mem_out <= 36'h7AE159AE0;
```

```
10'h347: mem_out <= 36'h7AEF63237;
10'h348: mem_out <= 36'h7AFD59A3A;
10'h349: mem_out <= 36'h7B0B3D2C6;
10'h34A: mem_out <= 36'h7B190DBB9;
10'h34B: mem_out <= 36'h7B26CB4F0;
10'h34C: mem_out <= 36'h7B3475E4B;
10'h34D: mem_out <= 36'h7B420D7A6;
10'h34E: mem_out <= 36'h7B4F920E1;
10'h34F: mem_out <= 36'h7B5D039DA;
10'h350: mem_out <= 36'h7B6A6226F;
10'h351: mem_out <= 36'h7B77ADA81;
10'h352: mem_out <= 36'h7B84E61EE;
10'h353: mem_out <= 36'h7B920B896;
10'h354: mem_out <= 36'h7B9F1DE58;
10'h355: mem_out <= 36'h7BAC1D314;
10'h356: mem_out <= 36'h7BB9096A9;
10'h357: mem_out <= 36'h7BC5E28F9;
10'h358: mem_out <= 36'h7BD2A89E2;
10'h359: mem_out <= 36'h7BDF5B947;
10'h35A: mem_out <= 36'h7BEBFB706;
10'h35B: mem_out <= 36'h7BF888302;
10'h35C: mem_out <= 36'h7C0501D1C;
10'h35D: mem_out <= 36'h7C1168533;
10'h35E: mem_out <= 36'h7C1DBBB2B;
10'h35F: mem_out <= 36'h7C29FBEE4;
10'h360: mem_out <= 36'h7C3629040;
10'h361: mem_out <= 36'h7C4242F22;
10'h362: mem_out <= 36'h7C4E49B6B;
10'h363: mem_out <= 36'h7C5A3D4FD;
10'h364: mem_out <= 36'h7C661DBBC;
10'h365: mem_out <= 36'h7C71EAF8A;
10'h366: mem_out <= 36'h7C7DA5049;
10'h367: mem_out <= 36'h7C894BDDD;
10'h368: mem_out <= 36'h7C94DF829;
10'h369: mem_out <= 36'h7CA05FF11;
10'h36A: mem_out <= 36'h7CABCD278;
10'h36B: mem_out <= 36'h7CB727242;
10'h36C: mem_out <= 36'h7CC26DE52;
10'h36D: mem_out <= 36'h7CCDA168E;
10'h36E: mem_out <= 36'h7CD8C1AD9;
10'h36F: mem_out <= 36'h7CE3CEB19;
10'h370: mem_out <= 36'h7CEEC8731;
10'h371: mem_out <= 36'h7CF9AEF06;
10'h372: mem_out <= 36'h7D048227F;
10'h373: mem_out <= 36'h7D0F4217F;
10'h374: mem_out <= 36'h7D19EEBED;
10'h375: mem_out <= 36'h7D24881AF;
10'h376: mem_out <= 36'h7D2F0E2A9;
10'h377: mem_out <= 36'h7D3980EC2;
10'h378: mem_out <= 36'h7D43E05E1;
10'h379: mem_out <= 36'h7D4E2C7EB;
10'h37A: mem_out <= 36'h7D58654C8;
10'h37B: mem_out <= 36'h7D628AC5E;
10'h37C: mem_out <= 36'h7D6C9CE93;
10'h37D: mem_out <= 36'h7D769BB50;
10'h37E: mem_out <= 36'h7D808727C;
10'h37F: mem_out <= 36'h7D8A5F3FD;
10'h380: mem_out <= 36'h7D9423FBD;
10'h381: mem_out <= 36'h7D9DD55A2;
10'h382: mem_out <= 36'h7DA773594;
10'h383: mem_out <= 36'h7DB0FDF7D;
```

```
10'h384: mem_out <= 36'h7DBA75344;
10'h385: mem_out <= 36'h7DC3D90D2;
10'h386: mem_out <= 36'h7DCD29811;
10'h387: mem_out <= 36'h7DD6668E8;
10'h388: mem_out <= 36'h7DDF90341;
10'h389: mem_out <= 36'h7DE8A6706;
10'h38A: mem_out <= 36'h7DF1A9420;
10'h38B: mem_out <= 36'h7DFA98A79;
10'h38C: mem_out <= 36'h7E03749FB;
10'h38D: mem_out <= 36'h7E0C3D290;
10'h38E: mem_out <= 36'h7E14F2422;
10'h38F: mem_out <= 36'h7E1D93E9C;
10'h390: mem_out <= 36'h7E26221E8;
10'h391: mem_out <= 36'h7E2E9CDF2;
10'h392: mem_out <= 36'h7E37042A5;
10'h393: mem_out <= 36'h7E3F57FEB;
10'h394: mem_out <= 36'h7E47985B1;
10'h395: mem_out <= 36'h7E4FC53E1;
10'h396: mem_out <= 36'h7E57DEA68;
10'h397: mem_out <= 36'h7E5FE4932;
10'h398: mem_out <= 36'h7E67D702A;
10'h399: mem_out <= 36'h7E6FB5F3E;
10'h39A: mem_out <= 36'h7E778165A;
10'h39B: mem_out <= 36'h7E7F3956B;
10'h39C: mem_out <= 36'h7E86DDC5D;
10'h39D: mem_out <= 36'h7E8E6EB1E;
10'h39E: mem_out <= 36'h7E95EC19B;
10'h39F: mem_out <= 36'h7E9D55FC2;
10'h3A0: mem_out <= 36'h7EA4AC580;
10'h3A1: mem_out <= 36'h7EABEF2C3;
10'h3A2: mem_out <= 36'h7EB31E779;
10'h3A3: mem_out <= 36'h7EBA3A391;
10'h3A4: mem_out <= 36'h7EC1426F9;
10'h3A5: mem_out <= 36'h7EC8371A0;
10'h3A6: mem_out <= 36'h7ECF18374;
10'h3A7: mem_out <= 36'h7ED5E5C65;
10'h3A8: mem_out <= 36'h7EDC9FC61;
10'h3A9: mem_out <= 36'h7EE346359;
10'h3AA: mem_out <= 36'h7EE9D913C;
10'h3AB: mem_out <= 36'h7EF0585F9;
10'h3AC: mem_out <= 36'h7EF6C4180;
10'h3AD: mem_out <= 36'h7EFD1C3C2;
10'h3AE: mem_out <= 36'h7F0360CAF;
10'h3AF: mem_out <= 36'h7F0991C38;
10'h3B0: mem_out <= 36'h7F0FAF24D;
10'h3B1: mem_out <= 36'h7F15B8EDF;
10'h3B2: mem_out <= 36'h7F1BAF1DF;
10'h3B3: mem_out <= 36'h7F2191B3F;
10'h3B4: mem_out <= 36'h7F2760AF0;
10'h3B5: mem_out <= 36'h7F2D1C0E4;
10'h3B6: mem_out <= 36'h7F32C3D0C;
10'h3B7: mem_out <= 36'h7F3857F5B;
10'h3B8: mem_out <= 36'h7F3DD87C3;
10'h3B9: mem_out <= 36'h7F4345636;
10'h3BA: mem_out <= 36'h7F489EAA6;
10'h3BB: mem_out <= 36'h7F4DE4508;
10'h3BC: mem_out <= 36'h7F531654D;
10'h3BD: mem_out <= 36'h7F5834B69;
10'h3BE: mem_out <= 36'h7F5D3F750;
10'h3BF: mem_out <= 36'h7F62368F4;
10'h3C0: mem_out <= 36'h7F671A049;
```

```
10'h3C1: mem_out <= 36'h7F6BE9D45;
10'h3C2: mem_out <= 36'h7F70A5FD9;
10'h3C3: mem_out <= 36'h7F754E7FC;
10'h3C4: mem_out <= 36'h7F79E35A1;
10'h3C5: mem_out <= 36'h7F7E648BD;
10'h3C6: mem_out <= 36'h7F82D2146;
10'h3C7: mem_out <= 36'h7F872BF2F;
10'h3C8: mem_out <= 36'h7F8B7226E;
10'h3C9: mem_out <= 36'h7F8FA4AFA;
10'h3CA: mem_out <= 36'h7F93C38C7;
10'h3CB: mem_out <= 36'h7F97CEBCB;
10'h3CC: mem_out <= 36'h7F9BC63FD;
10'h3CD: mem_out <= 36'h7F9FAA152;
10'h3CE: mem_out <= 36'h7FA37A3C0;
10'h3CF: mem_out <= 36'h7FA736B40;
10'h3D0: mem_out <= 36'h7FAADF7C7;
10'h3D1: mem_out <= 36'h7FAE7494C;
10'h3D2: mem_out <= 36'h7FB1F5FC6;
10'h3D3: mem_out <= 36'h7FB563B2D;
10'h3D4: mem_out <= 36'h7FB8BDB79;
10'h3D5: mem_out <= 36'h7FBC040A0;
10'h3D6: mem_out <= 36'h7FBF36A9C;
10'h3D7: mem_out <= 36'h7FC255963;
10'h3D8: mem_out <= 36'h7FC560CEF;
10'h3D9: mem_out <= 36'h7FC858538;
10'h3DA: mem_out <= 36'h7FCB3C236;
10'h3DB: mem_out <= 36'h7FCE0C3E3;
10'h3DC: mem_out <= 36'h7FD0C8A37;
10'h3DD: mem_out <= 36'h7FD37152C;
10'h3DE: mem_out <= 36'h7FD6064BB;
10'h3DF: mem_out <= 36'h7FD8878DE;
10'h3E0: mem_out <= 36'h7FDAF518E;
10'h3E1: mem_out <= 36'h7FDD4EEC6;
10'h3E2: mem_out <= 36'h7FDF95080;
10'h3E3: mem_out <= 36'h7FE1C76B6;
10'h3E4: mem_out <= 36'h7FE3E6163;
10'h3E5: mem_out <= 36'h7FE5F1082;
10'h3E6: mem_out <= 36'h7FE7E840D;
10'h3E7: mem_out <= 36'h7FE9CBBFF;
10'h3E8: mem_out <= 36'h7FEB9B855;
10'h3E9: mem_out <= 36'h7FED57909;
10'h3EA: mem_out <= 36'h7FEEFFE17;
10'h3EB: mem_out <= 36'h7FF09477C;
10'h3EC: mem_out <= 36'h7FF215533;
10'h3ED: mem_out <= 36'h7FF382738;
10'h3EE: mem_out <= 36'h7FF4DBD89;
10'h3EF: mem_out <= 36'h7FF621821;
10'h3F0: mem_out <= 36'h7FF7536FD;
10'h3F1: mem_out <= 36'h7FF871A1C;
10'h3F2: mem_out <= 36'h7FF97C179;
10'h3F3: mem_out <= 36'h7FFA72D12;
10'h3F4: mem_out <= 36'h7FFB55CE6;
10'h3F5: mem_out <= 36'h7FFC250F1;
10'h3F6: mem_out <= 36'h7FFCE0932;
10'h3F7: mem_out <= 36'h7FFD885A6;
10'h3F8: mem_out <= 36'h7FFE1C64D;
10'h3F9: mem_out <= 36'h7FFE9CB25;
10'h3FA: mem_out <= 36'h7FFF0942D;
10'h3FB: mem_out <= 36'h7FFF62163;
10'h3FC: mem_out <= 36'h7FFFA72C7;
10'h3FD: mem_out <= 36'h7FFFD8858;
```

```
            10'h3FE: mem_out <= 36'h7FFFF6216;
            10'h3FF: mem_out <= 36'h800000000;
            default: mem_out <= 0;
                    endcase
             end
      end
endmodule

/**/
/*Filter Modules from Lab 5 - Audio
/*adapted to accept and receive variable bit_depths & different 3dB frequency
/**/

///////////////////////////////////////////////////////////////////////////////
// 31-tap FIR filter, 10-bit signed coefficients.
// ready is asserted whenever there is a new sample on the X input,
// the Y output should also be sampled at the same time.  Assumes at
// least 32 clocks between ready assertions.  Note that since the
// coefficients have been scaled by 2**10, so has the output (it's
// expanded from 8 bits to 18 bits).  To get an 8-bit result from the
// filter just divide by 2**10, ie, use Y[17:10].
///////////////////////////////////////////////////////////////////////////////
module fir31
#(parameter WSIZE=36)
 (input wire clock,reset,ready,
  input wire signed [WSIZE-1:0] x,
  output reg signed [9+WSIZE:0] y);

  reg [4:0]offst=5'b0; // offset, ranges from 0 (5'b00000) to 31 (5'b11111)
  reg [4:0]index=5'b0; // index
  reg signed [WSIZE-1:0]cbuff[31:0]; //circular buffer
  reg signed [9+WSIZE:0]accum;
  wire signed [9:0]coeff;

  wire max_index=&index;
  coeffs31 cf31(.index(index),.coeff(coeff));

  always @(posedge clock) begin
     if (ready) begin
        offst <= &offst ? 5'b0: offst+5'b1;
             cbuff[offst] <= x;
             {index, accum} <= 23'b0;
             end
     else begin
             index <= max_index ? index: index+1;
             accum <= max_index ? accum: accum + coeff*cbuff[(offst-index)%32];
             end

     y <= &index ? accum: y;
  end
endmodule

///////////////////////////////////////////////////////////////////////////////
// Coefficients for a 31-tap low-pass FIR filter with Wn=.0833 (2kHz for a
// 48kHz sample rate). Since we're doing integer arithmetic, we've scaled
// the coefficients by 2**10
// Matlab command: round(fir1(30,2*(2/48))*1024)
///////////////////////////////////////////////////////////////////////////////
module coeffs31(
  input wire [4:0] index,
  output reg signed [9:0] coeff
```

```
);
  // tools will turn this into a 31x10 ROM
  always @(index)
    case (index)
      5'd0:  coeff = -1;  //-10'sd1;
      5'd1:  coeff = -1;  //-10'sd1;
      5'd2:  coeff = -1;  //-10'sd3;
      5'd3:  coeff = 0;   //-10'sd5;
      5'd4:  coeff = 2;   //-10'sd6;
      5'd5:  coeff = 5;   //-10'sd7;
      5'd6:  coeff = 11;  //-10'sd5;
      5'd7:  coeff = 19;  //10'sd0;
      5'd8:  coeff = 28;  //10'sd10;
      5'd9:  coeff = 40;  //10'sd26;
      5'd10: coeff = 52;  //10'sd46;
      5'd11: coeff = 64;  //10'sd69;
      5'd12: coeff = 75;  //10'sd91;
      5'd13: coeff = 84;  //10'sd110;
      5'd14: coeff = 90;  //10'sd123;
      5'd15: coeff = 91;  //10'sd128;
      5'd16: coeff = 90;  //10'sd123;
      5'd17: coeff = 84;  //10'sd110;
      5'd18: coeff = 75;  //10'sd91;
      5'd19: coeff = 64;  //10'sd69;
      5'd20: coeff = 52;  //10'sd46;
      5'd21: coeff = 40;  //10'sd26;
      5'd22: coeff = 28;  //10'sd10;
      5'd23: coeff = 19;  //10'sd0;
      5'd24: coeff = 11;  //-10'sd5;
      5'd25: coeff = 5;   //-10'sd7;
      5'd26: coeff = 2;   //-10'sd6;
      5'd27: coeff = 0;   //-10'sd5;
      5'd28: coeff = -1;  //-10'sd3;
      5'd29: coeff = -1;  //-10'sd1;
      5'd30: coeff = -1;  //-10'sd1;
      default: coeff = 10'hXXX;
    endcase
endmodule //fir31

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:       Evie Kyritsis
//
// Create Date:    18:37:27 12/09/2014
// Design Name:
// Module Name:    color_storage
// Project Name:
// Target Devices:
// Tool versions:
// Description:  Doesn't work - instead we implemented an asynchronous FIFO
//                that does a more efficient job of transferring information
//                               across clock domains.
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module color_storage(
                   input clk, //27 Mhz
```

```
                    input reset,
                    input [23:0] color, //color Janelle puts in
                    output reg [23:0] color_to_sound, //color for sabina to read
                    output reg storing //wire that goes up and down //up = storing, down = not
storing
    );

reg [12:0] counter; //count up to 563 twice and then change colors
reg [23:0] buffer;
always @(posedge clk)begin
      if(reset) begin
            storing <= 0;
            buffer <= color;
      end
      else begin
            color_to_sound <= buffer;
            storing <= 1;
            counter <= counter +1;
            if(counter == 1126)begin //27E6/48E3 times 2
                  storing <= 0;
                  buffer <= color;
            end
      end

end

endmodule //color_storage
```