Digital Audio Workstation
Sound Analysis through Hardware
6.111 Final Project Report - Fall 2015

Michelle Qiu
Germain Martinez
Gerzain Mata

Table of Contents

1. Abstract

Due to the advent of computers in the digital information age, many music aficionados moved their music production methodologies to the digital domain. Most use software-based Digital Audio Workstations to produce their music. With this in mind, we designed, tested, and implemented a hardware-based Digital Audio Workstation (DAW for short) which aims to implement the same kinds of functionality that a software-based implementation could achieve. This system allows the user to record their voice, save the audio samples in memory, and be able to playback the audio with a plethora of digital audio effects applied to them. At the same time, the sound output is run through a Fast Fourier Transform which serves to identify the predominant frequency and relative magnitude. The FFT information is also displayed on screen, as well as the length of the recording and the audio bank that was recorded into.

2. Acknowledgements

We would like to thank Gim Hom for providing us with guidance and informing us about what goals were feasible and what was unreasonable. We would especially like to thank him for providing us with a verilog FFT module, which helped us determine that our initial idea to create a percussive audio game was not possible due to the nature of the wideband frequency content that percussive sounds are made up of.

We would also like to thank I. Chuang for writing a ZBT driver and providing through the 6.111 site an example of a labkit project that interfaces with the ZBT SRAM.

Finally, we would like to thank Michael Trice for being flexible with meeting with us and giving us feedback on our reports and writeups.

3. Project Overview

In order to implement the DAW, there are several key functions that it must do. First, the DAW must be able to record and playback audio from various places in its memory. Next, it must be able to add effects to the audio. The DAW must be able to take a Fast Fourier Transform on the signal to obtain the frequency information contained in the signal. It must be able to perform other important calculations on the audio signal. Finally. the DAW must be interfaced in such a way that the user can interact with it without having to know all of the details of its function.

The two banks of ZBT SRAM on the 6.111 labkit are allocated to store audio information. The audio signal is recorded at a resolution (bit depth) of 12 bits and a sampling rate of 24 kHz. The system can record and playback audio from a total of 12 memory banks, all of which are on the ZBT memory. Memory banks 0 and 6 can each record up to 15 seconds of audio, while memory banks 1-5 and 7-11 can each record about 3 seconds of audio.

The project is controlled via the buttons and switches on the labkit. Important information is displayed on the VGA monitor and the labkit FPGA 40 led display. There are three states that the DAW can be in: standby, where nothing happens; record, where the user can choose a song to record to; and playback, where the user can choose a memory bank to play back audio from and apply audio effects to. The VGA output displays a Fast Fourier Transform of the audio playing through the speaker; it also shows the dominant frequency (the frequency at which the audio is the loudest), the maximum amplitude, the time elapsed in the current recording (in seconds), the number of effects being applied, and the memory bank choice. The central FSM state and number of seconds elapsed can be viewed on the labkit; the record mode, song choice, and effect levels can be set upon starting record or playback on the labkit.

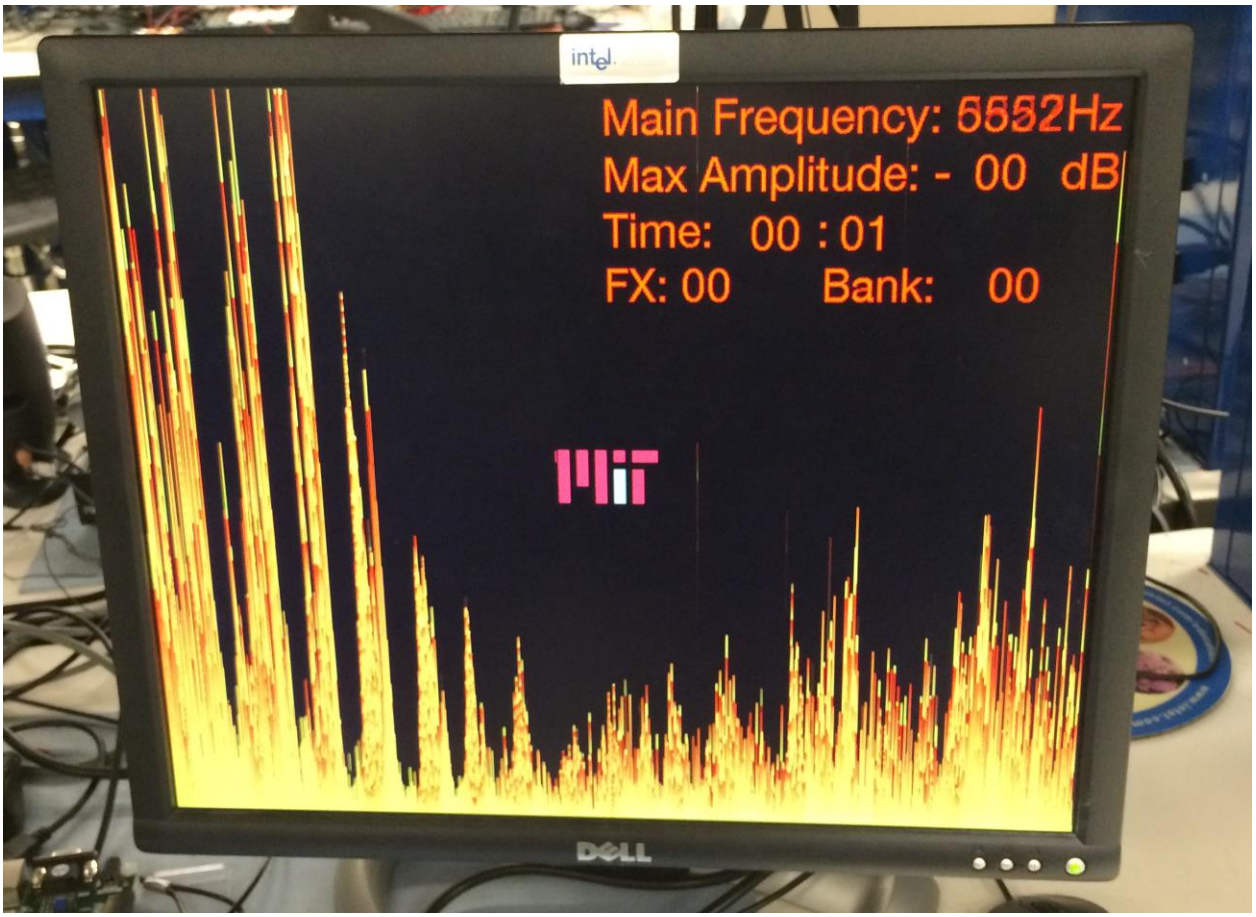

Figure 1: VGA Display showing song info and FFT spectrum

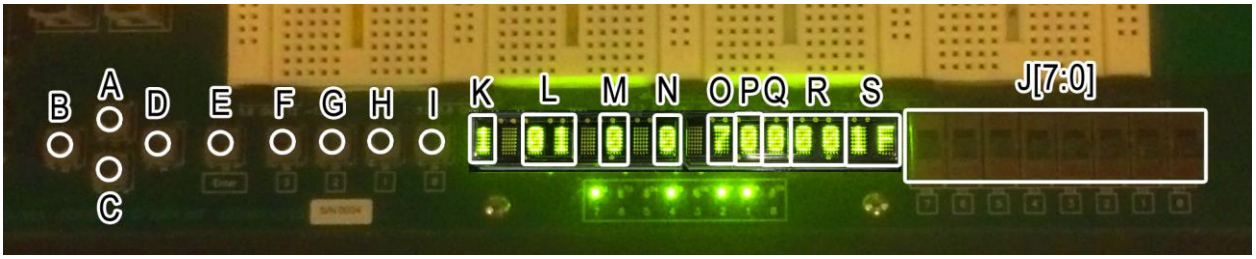

Figure 2: Labkit Control and Display.

Figure 1 key:

A,C: Increment (via A) and decrement (via C) selected parameter
B,D: Go to the next (via D) or previous (via B) selectable parameter
E: Enter, transition state
F: Reset
G: Volume down

H: Volume up
I: Pause/unpause
J [7:0]: Toggle effects on/off (0: delay/echo, 1: chorus, 2: compression, 3: soft limiter, 4: hard limiter, 5: distortion, 6: playback slow down x2, 7: playback speed up x2)
K: Central FSM state (0: standby, 1: playback, 2: record)
L: Seconds elapsed (0x00 to 0xFF)
M: Selectable parameter: (0: playback, 1: record)
N: Selectable parameter: song choice (0 to 11)
O: Selectable parameter: distortion level (0 to 7)
P: Selectable parameter: limiter level (0 to 3)
Q: Selectable parameter: compression level (0 to 3)
R: Selectable parameter: chorus level (0x00 to 0x1F)
S: Selectable parameter: echo level(0x00 to 0x1F)

4. Module Overview

The project is split into four main sections in order to facilitate implementation and debugging. The central FSM controls the other three modules, the memory module records from the microphone and plays back audio data, the audio module processes the audio data and sends it to the speaker, and the graphics module displays the sound analysis through a VGA signal.

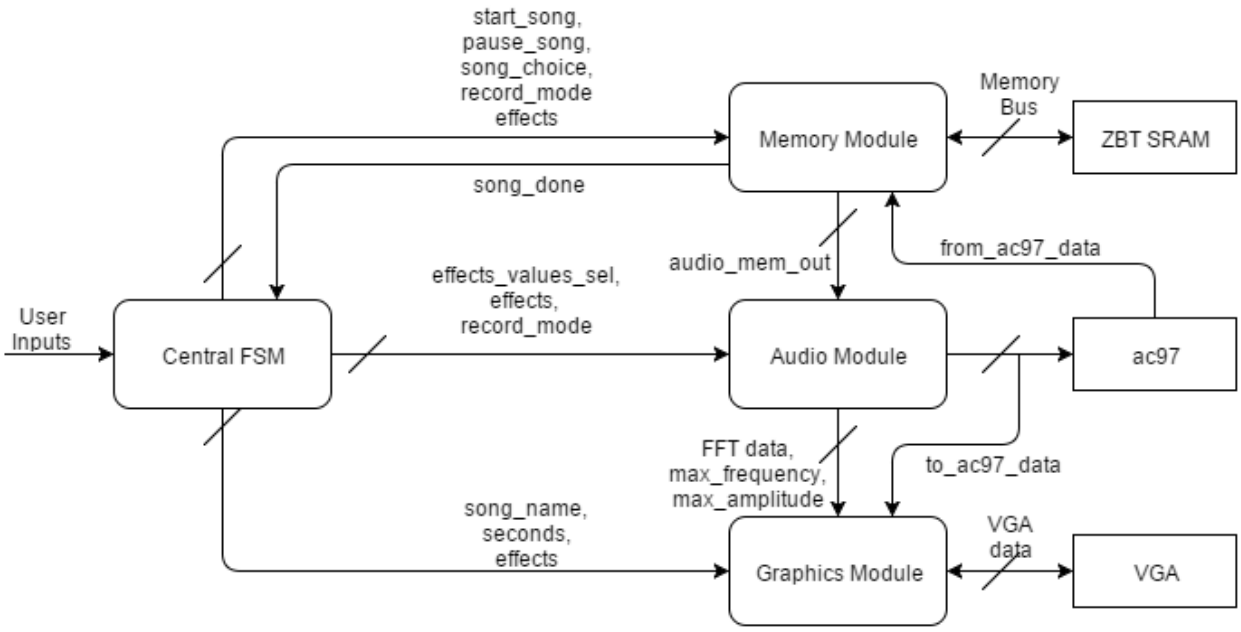

Figure 2: Block diagram showing the system as a whole and the signal flow between the main modules.

5. Module Implementation

5.1 Top-level Modules

The labkit_experiment module interfaces with the labkit and is where all of the integration between the modules take place. To write the labkit_experiment, the labkit.v files from lab 3 (the graphics lab) and from lab 5a (the audio processing lab) were used as a base; modifications were made to these modules to ensure that the proper functionality was achieved from the central FSM, the audio modules, the graphics modules, and the memory modules. These modules were modified to route any inputs (buttons, switches, ac97) and outputs (ac97, graphics out through VGA) on the labkit to their respective internal modules.

The debounce module, which was provided by previous labs, makes sure that that all the buttons and switches on the labkit are debounced and do not have buggy inputs.

5.2 Integration

In order to simplify integration of the modules, the protocol and bus widths for information shared between modules was determined before the modules were written. In addition, each module was tested on its own before integration. The modules were made to be modular; their functionality does not depend on the functionality of other modules. This was facilitated by using standardized inputs and outputs and making sure that the protocols and bus widths that were agreed upon from the beginning were being used in the construction of each module as they were needed. As a result, the integration of the modules with each other and the labkit was very smooth. Communication issues between the modules that occurred in our implementation were not because of incorrect bus widths or mismatching protocols; rather, any errors we ran into were due to issues with

5.3 Central FSM (Michelle)

5.3.1 Overview

In order to determine how the DAW functions, the central FSM is in charge of keeping track of the state of the project and transitions between record, playback, and

standby. It also sends signals to the other three modules to indicate what they should be outputting.

5.3.2 Details

The central FSM tells the memory module whether to start recording or playing back, whether to pause the song, and the song choice; tells the graphics module the song name, the effects applied, and the seconds elapsed; and tells the audio module the recording mode, effects, and effect values.

In order to know when and what values to send to the modules, the central FSM must keep track of its state and the parameters that the user can modify, like the record mode, the song choice, and effects and values while still allowing the user to easily see and modify these values. Since the number of parameters the user can modify exceeds the number of switches that the labkit has, A state machine module also needed to be implemented along with the central FSM to keep track of the modifiable values. In order to display these values, the display_16hex module from lab 5b was modified so that it could display a blank character and a completely filled in character. The blank character marked spaces between parameters while the filled in character signal was controlled by a blinking signal blink_fo so the user would know which modifiable parameter was selected.

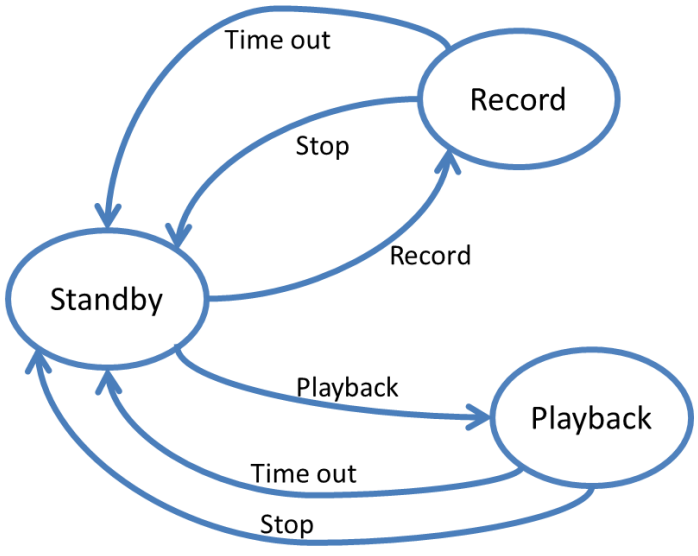

Figure 3: Central FSM

On transition from standby to a different state, the central FSM pulses a start signal and takes parameters chosen by the user to send to the other modules. When the state returns to standby, the FSM notifies the other main modules to not do anything.

5.3.3 Block Diagram:

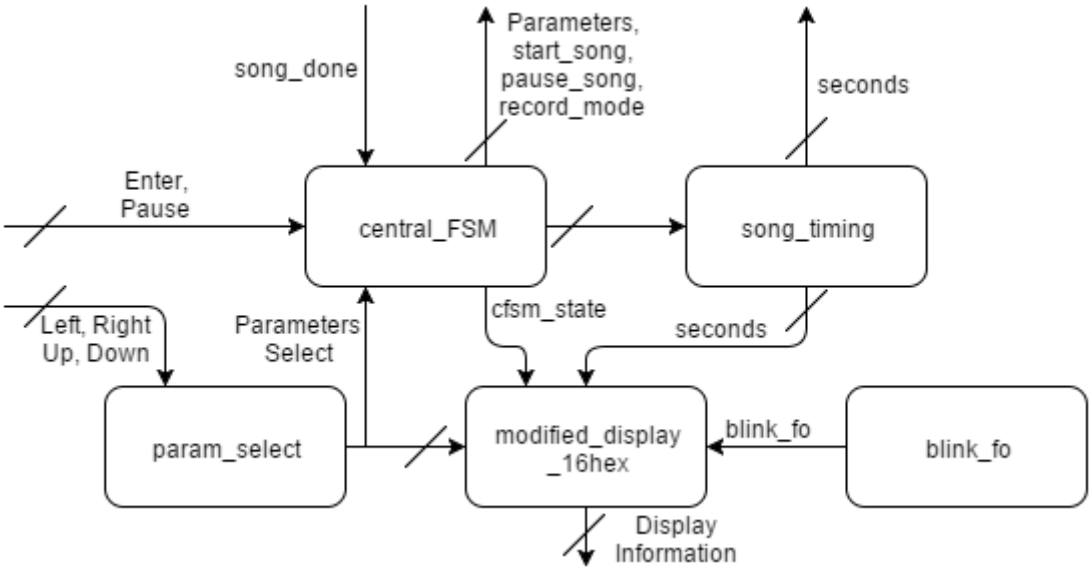

Figure 4: Central FSM Block Diagram

5.3.4 Implementation

5.3.4.1 Central FSM Module (central_fsm.v):

The central FSM controls the entire project by determining states and when to start and stop recording and playing back. When the user presses enter to transition to the record or playback state, the central FSM assigns the selections from the parameter selector module to record_mode, song_name, and effect_values and outputs these to the other modules. In addition, it translates the song_name to song_choice, which is what the memory module uses to determine which song to record to, since the memory module uses the highest bit to determine which ZBT bank to record to.

In addition, the central FSM asserts start_song one clock cycle after the enter button is pressed so that the other modules know when the song starts and the parameter registers from the previous cycle have been correctly updated.

Finally, the central FSM determines pause_song. When the state is in standby, pause_song is always asserted high. In record and playback, pause_song is initially low, and the bit is flipped every time the pause button is pressed.

5.3.4.2 Parameter Selector Module (param_sel.v):

The parameter selector module was an independent state machine from the central FSM that determined selection of the parameters. Each state meant a different parameter could be changed. The parameters that could be changed were the record_mode_sel, song_name_sel, and effect_values_sel, which determine values for distortion level, limiter level, compression level, chorus level, and echo level.. By pressing the right and left buttons on the labkit, the parameter selector transitions from one state to another. The up and down buttons changed the value of the selected parameter. Even though the values of the parameter selector module could be changed at any time, the central fsm module only uses the value at the clock cycle right before start_song is asserted.

5.3.4.3 Song Timing Module (song_timing.v):

The song timing module determines the number of seconds that a song has been recording or playing back. It uses the start_song signal from the central FSM and the song_done and pause_song to determine how long the song has been running. The counter resets and starts incrementing every second when start_song is asserted, and stops incrementing when song_done or pause_song is asserted.

5.3.4.4 Blink Module (blink_fo.v):

The blink module generates a signal, blink_fo, that alternates between high for a fourth of a second, and low for three fourths of a second. The output of the blink module is used in the modified display 16 hex module so that the user knows which parameter is selected for change.

5.3.4.5 Modified Display 16 Hex module (modified_display_16hex.v)

The modified display 16 hex module is used as a driver to display info on the labkit's 40 led display. It was modified from the display_16hex module from lab5b. The modified module takes in two additional inputs, blink_data and blank_data. blink_data is a 16 bit bus that determines if each of the 16 characters should be a solid color, and blank_data is a 16 bit bus that determines if each of the 16 characters should be a blank color, with blink_data taking precedence over blank_data. When a character has been determined to blink because the user has selected it, the corresponding bit in blink_data is set to blink_fo every clock cycle. The characters that have no information had the corresponding bit is blank_data set to 1.

5.4 Memory Modules (Michelle)

5.4.1 Overview

The memory module records audio taken from ac97 and plays it back to the audio modules to be processed upon request by the central FSM. In addition, the memory module is tasked with applying the speedup and slowdown effects since those effects are related to the rate at which audio samples are read from memory.

5.4.2 Details

Data comes directly from ac97 into the memory module, where it is stored into the specified song determined by the central FSM when the project is in record mode. When the allocated memory for the song is full or the user choses to stop recording, the memory stops writing. When the central FSM tells the memory module to playback, the memory starts reading out audio data from the selected bank. If either speed up or slow down, but not both, is selected, then the memory reads at either twice the rate or half the rate, depending on which effect is selected.

5.4.3 Block Diagram

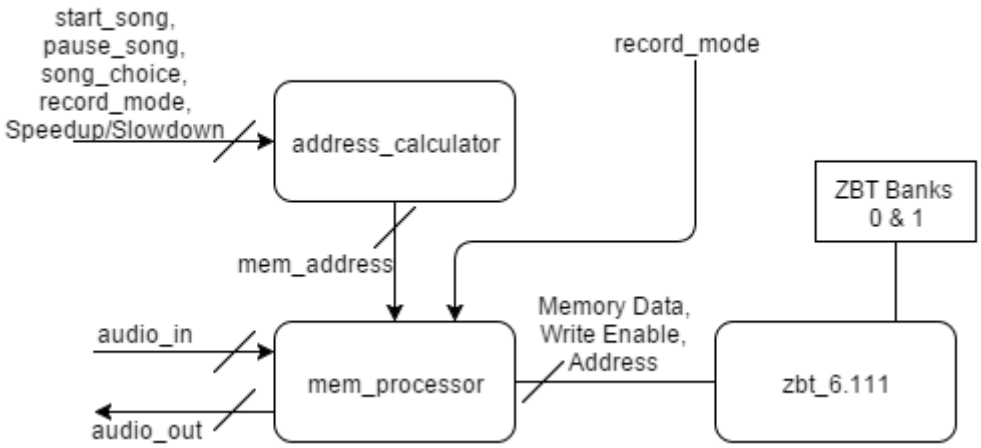

Figure 5: Memory Module Block Diagram

5.4.4 Implementation

5.4.4.1 Address Calculator Module (address_calculator.v)

The address calculator module determines which memory address, mem_address, the project needs to read from or write to based on inputs from the central FSM. The address calculator also signals song_done, which tells the central FSM if a song recording or playing has reached its max capacity/length. The address calcul

When start_song is asserted, the mem_address is reset to the base address of the song chosen by song_choice and song_done is deasserted. Then, with every assertion of ready, the mem_address is incremented until it reaches the highest address for that song choice and asserts song_done if record_mode is in playback.

If record_mode is in the record state, then when start_song is asserted the mem_address reset to the base address and song_done are deasserted. In addition, the highest address register is also reset to the base address and the maximum address register is updated to the maximum address of the song. With every assertion of ready, the mem_address and the highest_address for that song increment. When the current_address reaches the maximum address, song_done is asserted.

The address calculator does not need to consider the user ending a song recording early because when the user chooses to end a song early, the central FSM asserts pause_song, and the address stops incrementing. Pause_song is not deasserted by the central FSM until the next time start_song is asserted.

Because the speedup and slowdown effects are directly related to how frequently the memory addresses are read, the address calculator also implements speed up and slow down. If the speed up bit is asserted, then the address calculator increments by 2 addresses every time ready is asserted, so that for every 3 audio samples read, 3 audio samples were skipped. If the slow down bit is asserted, the address calculator increments mem_address every other time ready is asserted, so every 3 audio samples are repeated once. This creates a basic implementation of speed up and slow down.

5.4.4.2 Memory Processor Module (mem_processor.v)

The memory processor module serves as a translator between the ZBT memory and the audio samples. Because each address in the ZBT has 36 bits and each audio sample is only 12 bits, we can store 3 audio samples for every memory address. Therefore, we only read or write to memory for every three ready asserts from ac97.

The bank in which the memory is written to is determined by the highest order bit in song choice. Therefore, song_choices 0-5 are written to bank 0 while song_choices 8-13 are written to bank 1. The memory processor calculates whether an address should be written to if ready is asserted, the song is not done, and the song is not paused. The data that is written to memory is a sliding window with the newest audio sample inserted into the right-most 12 bits while the other samples are pushed 12 bits to the left.

The memory processor also reads the memory every three ready assertions. When the memory processor reads the memory, it stores the memory to a different register last_read_mem. This ensures that if address incrementation and memory processor reading are asynchronous, the audio data will not be read out of order.

5.4.4.3 ZBT Driver (zbt_6.111.v)

The ZBT module was found on the 6.111 website in the Fall 2005 tools. This module was a driver between the ZBT SRAM used to interface with the ZBT.

5.5 Audio Modules (Germain)

5.5.1 Overview

When the recorded audio is played back, the user has the option to apply one or more effects to the audio. The audio modules each have the task of applying a predefined effect to the audio that is being played back. Each audio module is controlled by the switches on the labkit. Here is a list that contains the switch number on the labkit, the control signal it handles, and the effect that the control signal is linked to.

Switch 0, delay_enable: Delay/Echo
Switch 1, chorus_enable: Chorus
Switch 2, compression_enable: Compression
Switch 3, soft_limiter_enable: Soft Limiter
Switch 4, hard_limiter_enable: Hard Limiter
Switch 5, distortion_enable: Distortion
Switch 6: Playback Slow Down (controlled by ZBT memory handler module)
Switch 7: Playback Speed Up (controlled by ZBT memory handler module)

The block diagram is depicted below.

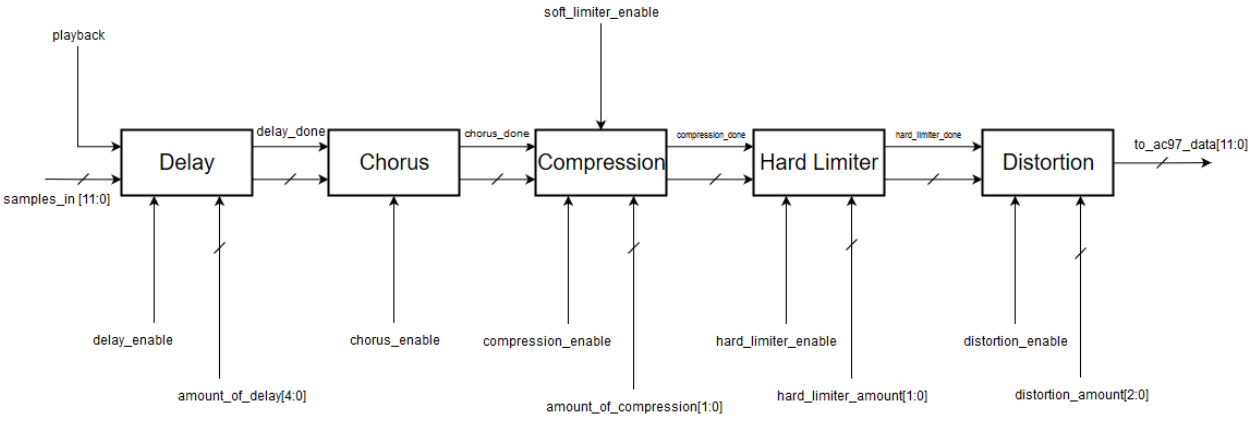

Figure 6: The block diagram illustrating the overall signal flow of the audio modules.

If an effect is not enabled, then the audio samples that pass into the effect are not affected, i.e., the samples going in will be the same coming out.

Playback speed up and slow down are handled by changing the frequency at which samples are read from memory. Speed up requires that every other sample is skipped; slow down requires that every sample is repeated once before the next sample is read.

The entire system runs on the 27 MHz clock on the FPGA. This includes all of the submodules in the system. Care was taken to reduce the calculations needed in each block such that the system runs with as little latency as possible. The most complex calculation that any part of the system has to do is limited to multiplication; this calculation would be pipelined every time it was used to guarantee that the system could generate valid outputs. The number of cycles available to process each sample is equal to the system clock divided by the sampling rate, or 27 MHz divided by 24 kHz. This number comes out to be a bit more than 1000 cycles, which gives the system plenty of time to do the audio processing in the worst-case where every effect is enabled at once. Minimizing the number of calculations needed in the audio processing block guarantees that the entire block looks like a one-cycle delay between the time that a recorded sample is released from memory to the time that it is output to the FFT module and the onboard ac97 audio.

5.5.2 Delay/Echo Module (delay_module.v)

5.5.2.1 Theory

The delay/echo module's main purpose is to mimic the auditory sensation of sound waves bouncing around inside a room right before they reach the listener. The echo module can be described using a linear, time-invariant system. Its behavior is summarized in the following diagram:

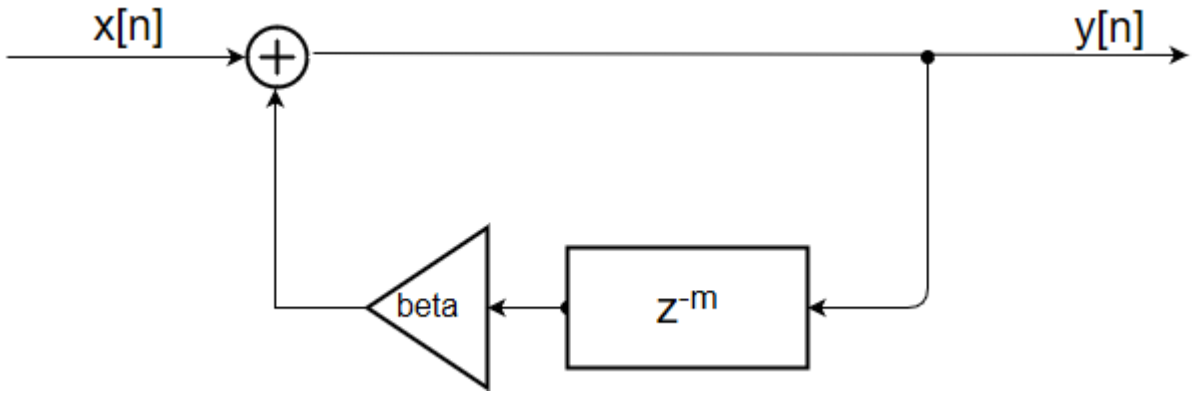

Figure 7: The block diagram representation of a theoretically sound feedback echo system.

In a usual echo system, the equation that describes the LTI system is $y[n] = x[n] + \beta y[n-m]$, where y[n] is the output signal, x[n] is the input signal, m is the delay factor, and β is the gain factor.

In theory, this system will combine the input signal with a time-delayed, attenuated version of the output signal. This would give the listener the impression of an echo. However, in practice this system will not give you the results you want. The reason for this is that there is a small, but important, nuance in digital systems. For any digital system that encodes signed integers with bits, the largest integer you can encode is 2^{n-1} , where n is the number of bits used to encode the numbers. This implementation uses 12 bits to encode audio, which means the largest possible value a sample can be is 2048. There's a limit to how large the samples can be. This means that if the first equation is implemented, there's a chance that samples can hit the maximum number. This will result in the output signal "clipping" or distorting at a certain input loudness, which is not desirable. Even though the positive feedback system is considered to be stable due to the feedback gain being less than one, the output will still distort for large enough inputs.

5.5.2.2 Implementation

This diagram shows a more practical system that implements the effect:

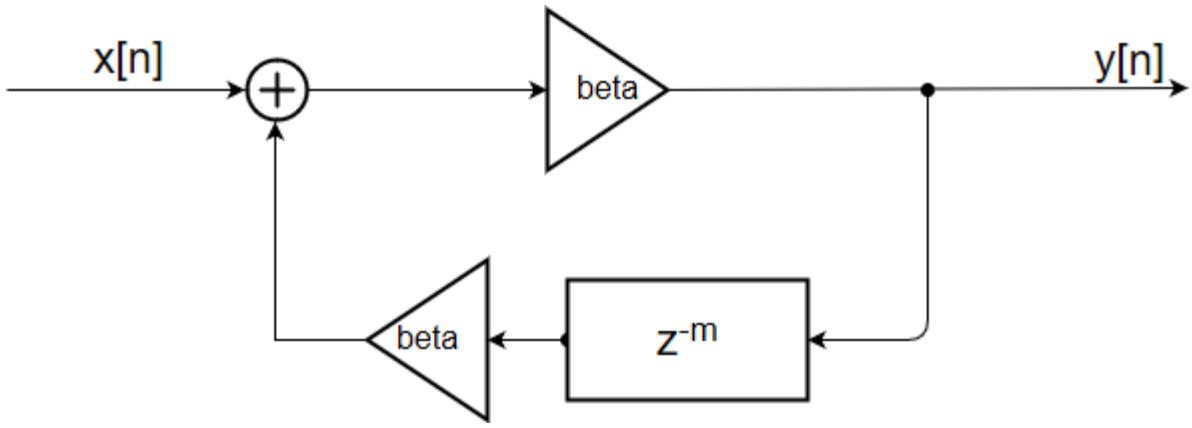

Figure 8: The block diagram representation of the delay/echo effect used in the project.

The equation that describes the delay/echo implementation in this project is $y[n] = \beta x[n] + \beta y[n-m]$,
where β is the gain factor. In the implementation, β = ½; this makes the calculation very simple in digital hardware since it can be done by arithmetically right shifting the binary value one time.

There is an extra attenuation factor applied to the input signal x[n]. This is included to prevent the output from distorting due to hitting the maximum value that can be encoded. There is a trade-off, though. Because the input is attenuated, the output will also be attenuated, so the recording will sound quieter with the delay effect on than with the delay effect off. This is a trade-off that was made to ensure a clean output and system stability.

The verilog module delay_module.v uses the delay system described in equation 2. The delay factor is variable; it can be set to values between 10 ms and 310 ms. These numbers are multiplied by 240 samples per millisecond to get the proper number of samples. The samples are stored using a module called mybram, which generates a single-port block RAM that has a memory capacity of 8192 x 12 bits. An IP core generated by the FPGA tools in ISE that implements a block RAM can also be used; however, it's much harder to test the functionality of the system using a simulator if one was to use this. A separate parameterized module was used rather than an IP core to simplify the design verification in simulation.

5.5.3 Chorus Module (chorus_effect.v)

5.5.3.1 Theory

The chorus effect mimics the auditory sensation of multiple voices playing the same part or making the same noises simultaneously.To create this effect, the system takes the input signal and adds multiple delayed versions of the input to it. These delays are rather short; four delay lines with 10-100 ms of delay on them is sufficient enough to recreate this effect. The delay lines are slightly out of phase with one another and with the input signal; this small difference in phase is enough to create the auditory phenomenon of multiple voices creating the same sound. The block diagram for chorus is shown below.

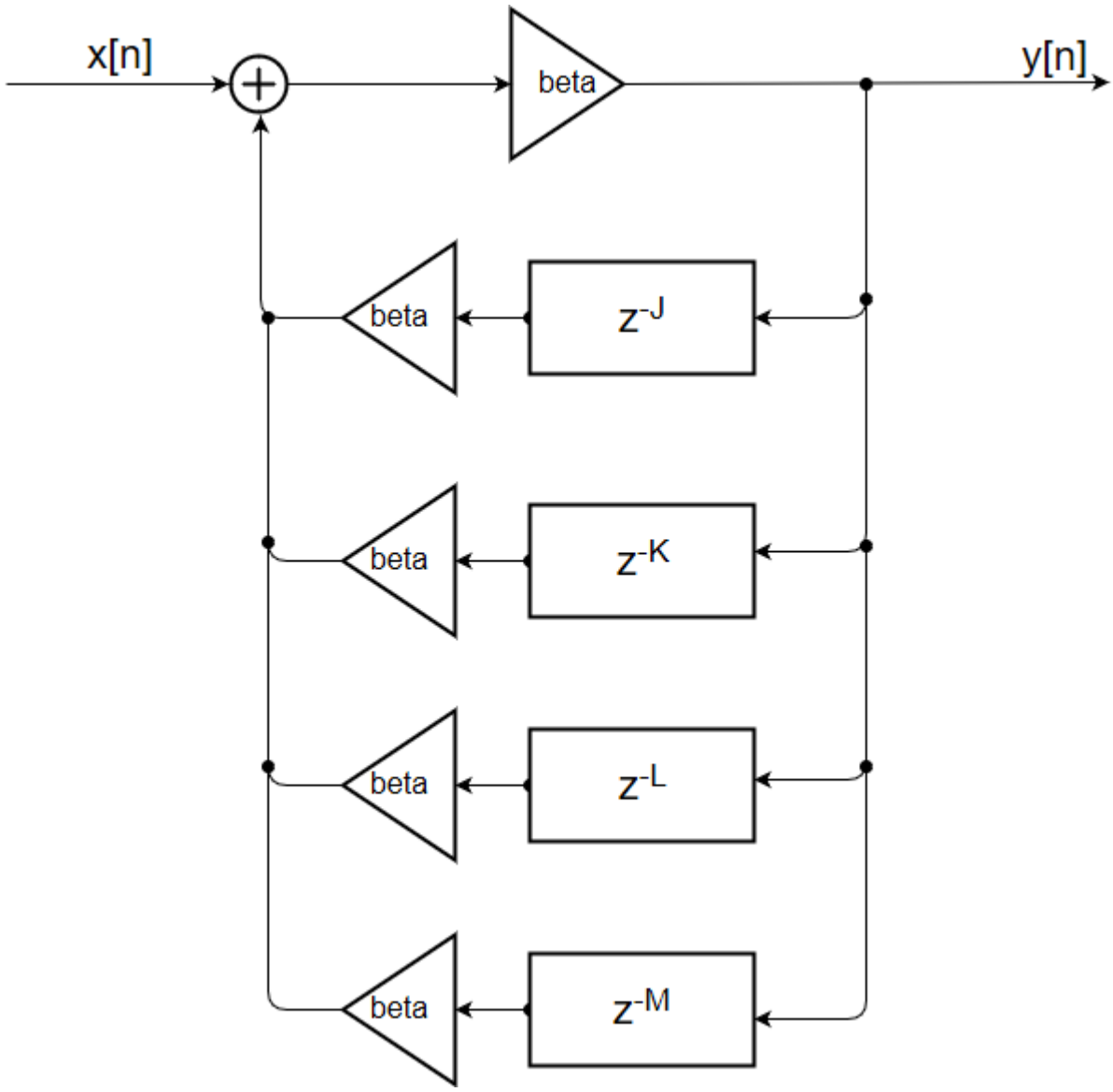

Figure 9: The block diagram representation of the chorus effect.

The equation that describes the chorus implementation in this project is

$y[n] = \beta x[n] + \beta y[n-J] + \beta\ y[n-K] + \beta\ y[n-L] + \beta\ y[n-M]$,

where y[n] is the input signal, y[n] is the output signal, β is the gain factor, and J, K, L, and M are all preset delay values.

5.5.3.2 Implementation

In chorus_effect.v, the delay parameters are set to J = 30 ms, K = 70 ms, L = 150 ms, and M is 310 ms. These numbers are multiplied by 240 samples per millisecond to get the proper number of samples. The gain factor β was chosen to be ½, which is simple to calculate in hardware since it can be done by an arithmetic right shift. Just like in the delay module, the input is attenuated by a factor of ½. This will ensure that the output does not clip and distort, although it also means that the audio volume is reduced from the original recording. Again, this is a trade-off that was made to ensure clean output audio and system stability.

5.5.4 Compression Module (compression.v and other modules)

5.5.4.1 Overview

In audio processing, dynamic range compression is the act of reducing the volume of loud sounds or increasing the volume of quiet sounds by reducing the audio signal's dynamic range. This effect is commonly employed in professional audio recordings to make the audio volume more consistent. The compression module does just that. The result of this effect is that the audio sounds like it is consistently at the same volume.

Compression is modeled as a variable gain block. The gain factor changes depending on the amplitude of the input system; this means that, unlike delay/echo and chorus, it is a nonlinear system because the gain factor is a function of the input signal amplitude. Compression can be modeled with the equation

$$y[n] = g(x[n]) \cdot x[n],$$

where g is a function of the input signal. Conventional LTI system analysis will not be sufficient to recreate this effect, even in hardware. The next figure shows the block diagram representation of the compression effect.

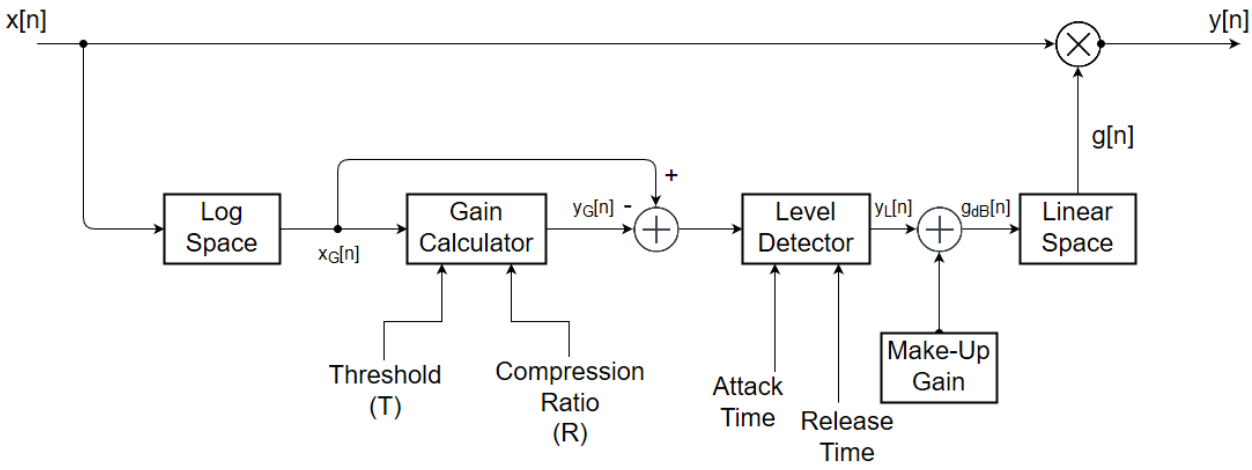

Figure 10: The block diagram modeling the feed-forward compression effect.

The feed-forward system of compression shown here is the preferred way of implementing the effect in digital systems because it introduces the least amount of distortion to the input signal compared to other methods (Giannoulis, Massberg, and Reiss). The variable gain is calculated by breaking up the process into steps. First, the sample is converted to its logarithmic space representation (in the implementation, the decibel representation is employed). Then, the system calculates the gain factor. This gain factor is put through a peak detector to adjust the gain based on whether or not the audio is at a peak. A make-up gain is applied to the gain factor to amplify the quieter parts of the audio. Finally, the resulting gain factor is converted back to a binary representation and is applied to the input signal.
In the following sections, the process for arriving at $g(x[n])$ is explained.

5.5.4.2 Logarithmic-Space Converter (signed_binary_12bit_to_dB.v)

5.5.4.2.1 Theory:

For the logarithmic-space converter, an algorithm that converts binary numbers to a decibel representation of the binary number was utilized (Weistroffer, Cooper, & Tucker, 2007). The decibel representation is a binary number, but it represents the value of the binary number in the decibel scale.

The algorithm works in the following way. The system takes the absolute value of the input sample. It then finds the place in the binary number that the most significant bit is located and saves that to a register. At the same time, the four bits after the msb are

saved into another register (this value is called R). The msb place is sent to a lookup table containing values of $k * (G_c - G_m)$, while the value of R is used to extract the value of $k * G_r$. The dB value is found by a subtraction:
$k * G = k * (G_c - G_m)$-$k * G_r$,
where G is the desired decibel representation of the sample (Weistroffer, Cooper, & Tucker, 2007). Getting the value of G is as simple as doing an arithmetic right shift, so long as an appropriate value of k is selected. The implementation uses k = 4.

5.5.4.2.2 Implementation:

The advantage of doing all the gain factor calculations in logarithmic space instead of on the raw binary samples is that the calculations are simplified to additions and subtractions, which saves a lot of resources on the FPGA. Exponentiation and logarithms are not trivial to implement in digital logic. This process is much simplified when the samples are converted to logarithmic space. The dB conversion algorithm's main advantage is that it is light; it only requires two small lookup tables, a subtraction, and a bit shift operation. The precision of the calculation is not a huge issue either; the algorithm was found to have a maximum error of 0.5 dB, according to Weistroffer, et al.
The module that implements this is called signed_binary_12bit_to_dB.

5.5.4.3 Gain Computer (compression_gain_computer.v)

5.5.4.3.1 Theory:

The gain computer calculates the amount of gain or attenuation that needs to be applied to the input signal. It does this based on the piecewise-defined equation

$$y_G[n] = x_G[n], x_G[n] \leq T;$$
$$y_G[n] = T + \frac{(x_G[n] - T)}{R}, x_G[n] > T;$$

where $y_G[n]$ is the output gain, $x_G[n]$ is the logarithmic representation of the input sample (in dB), T is the threshold value (in dB), and R is the compression ratio defined by the relation

$$R = \frac{x_G[n] - T}{y_G[n] - T}$$

(Giannoulis, Massberg, and Reiss). The output $y_G[n]$ is then subtracted from the input $x_G[n]$ to yield $x_L[n]$, the actual calculated gain:

$x_L[n] = x_G[n] - y_G[n]$.

5.5.4.3.2 Implementation:

The module compression_gain_calculator.v implements this function. The threshold value is set to -18 dB, which corresponds to a binary value of 256. The compression factor R can be set to different ratios. For the project, R is set by compression_amount. Here are the possible compression ratios in this implementation:

2'b00: No compression ratio
2'b01: Compression ratio of 2:1
2'b10: Compression ratio of 4:1
2'b11: Compression ratio of 8:1.

The compression ratios are chosen such that the division calculation is done through right shifts. This avoids having to use hardware dividers.

The gain calculator configuration from above yields what is known as a "hard knee". The name comes from the fact that $y_G[n]$ is a piecewise defined function in $x_G[n]$ and has a bent shape. Giannoulis, et al. also defines the configuration for a "soft knee", which has a smoother shape. In actual implementations of compression, the difference is hard to notice; this project uses the hard knee formulation to calculate the gain for simplicity.

5.5.4.4 Level Detector (compression_level_detector.v)

5.5.4.4.1 Theory:

The gain computer can find the amount of gain or attenuation to apply to the input signal, but it has issues with signals that change quickly. In instances where the signal quickly changes in dB values or crosses the threshold quickly. setting the gain calculator output as the compression gain will result in sharp discontinuities in the output signal. In practice, this results in a distorted signal. To smooth the threshold transitions in faster signals, we use a level detector.

The level detector takes as input the gain factor calculated from the gain computer. It then smooths the signal using the piecewise-defined difference equation

$$y_L[n] = \alpha_A y_L[n-1] + , (1-\alpha_A)x_L[n], x_L[n] > y_L[n-1];$$
$$y_L[n] = \alpha_R y_L[n-1] + , (1-\alpha_R)x_L[n], x_L[n] \leq y_L[n-1];$$

where $x_L[n]$is the input gain from the gain computer, $y_L[n]$ is the output level, α_A is the attack coefficient, and $\alpha_R$is the release coefficient (Giannoulis, Massberg, and Reiss). To find $\alpha_A$and α_R, the attack and release coefficients, the following relations are true:

$$\alpha_A = exp\left(\frac{-1}{\tau_A f_s}\right)$$

$$\alpha_R = exp\left(\frac{-1}{\tau_R f_s}\right),$$

where τ_A and τ_R are the attack and release time constants, respectively, and f_s is the sampling frequency (which is 48 kHz for this project). The output $y_L[n]$ is a smoothed version of
$x_L[n]$ that filters out the fast transitions across the threshold.

5.5.4.4.2 Implementation:

The module compression_level_detector.v implements this part of the system. For a sampling frequency of 48 kHz and attack and release constants both set to 25 ms, the attack and release coefficients come out to be about 0.998. This is an issue because binary numbers don't have decimal points. To get this ratio in binary without resorting to fixed-point arithmetic, this implementation multiplies $y_L[n-1]$ by 510 and $x_L[n]$ by 2. These values are then added together and the result is divided by 512 to get $y_L[n]$.

5.5.4.5 Make-up Gain (compression_level_detector.v)

The make-up gain takes into account that signals that fall lower than the threshold also need to be amplified to aid in the process of reducing the dynamic range of the audio. The make-up gain is simple to implement in the logarithmic space; what would be a multiplication in binary ends up being an addition in logarithmic space. Thus, the make-up gain is applied to the output level through the equation
$g_{dB}[n] = y_L[n] + c_{dB}$,
where c_{dB} is the make-up gain. In the project, the make-up gain is set at 12 dB. Its function is controlled by soft_limiter_enable; when soft_limiter_enable is set high, the make-up gain

is turned off entirely and the compression turns into a soft limiter (more on this in the soft limiter section).

5.5.4.6 Conversion back to Linear Space (variable_gain.v):

The final gain value $g_{dB}[n]$ is in logarithmic space. To be able to use it as a gain, a conversion back to linear space is necessary. However, this conversion would yield fractions or decimal values, which isn't easy to work with. Instead, the input dB value is sent to a lookup table, which finds the corresponding multiplication factor (out of 1024). The input sample is then multiplied by this factor, and the result is then divided by 1024 (which is achieved in hardware by an arithmetic right shift of 10 places). This process of a multiply followed by a right shift is how the gain $g[n]$ is applied to the input sample. The output is then $y[n] = g(x[n]) \cdot x[n]$, which is the exact model used to describe the process in the overview.

5.5.4.7 Overall Implementation

In the project, the verilog modules signed_binary_12bit_to_dB, compression_gain_computer.v, compression_level_detector.v, and variable_gain.v together do the calculations necessary to modify the input samples. The compression module combines them all and routes the control, input, and output signals to and from their respective modules. Each module is constructed as a state machine; when one part is finished doing calculations, it sends the result and a done signal to the next module. The modules do not carry out calculations unless they receive a start signal from the last module (the done signal); the compression module acts as a pass-through when the compression effect is disabled.

The main challenge with implementing compression is in the gain calculations. With binary numbers, the largest dB value a number can be is 0 dB, which corresponds to the largest binary number allowed within a certain number of bits (this value is discussed in the delay/echo module section). This means that the positive decibel values found in the conversion process are actually *negative*; the smaller the sample is in amplitude, the smaller the sample is (not bigger). This makes setting the case necessary for the gain computer tricky; one has to keep in mind that the greater than and less than signs have to be reversed if this implementation is to be followed.

Another challenge with implementing compression lies within the amount of precision in the dB calculation. In the compression.v implementation, the dB conversion is precise to the ones digit. This can lead to discontinuities in the output waveform at boundaries where the output waveform amplitude changes slowly and the measured dB value changes. These discontinuities lead to the output waveform sounding a bit noisy. The solution to this is to use higher precision in the dB values by doing all the calculations with the $k * G$ value found from the lookup table. This solution was not put into the project at the time of writing, but it's a design consideration that should be made in the future should one need a clean compression effect module.

5.5.5 Soft Limiter Module (compression.v, soft_limiter_module)

The soft limiter module is a feature included in the compression module. Dynamic range compression reduces the volume of loud sounds and increases the volume of quiet sounds to reduce the loudness range that the output audio can have. A limiter, however, only reduces the volume of loud sounds; it does not increase the volume of quiet sounds. The output from a limiter has a larger dynamic range than the output from a compressor. One way to implement a limiter would be to make a compression effect in the fashion one described above but without the make-up gain block.

In the implementation, switch 3 controls the soft_limiting register; when it is turned on, the compression module skips the make-up gain block when calculating the gain that needs to be applied to the input audio.

5.5.6 Hard Limiter (Clipper) Module (limiter_module.v)

Another way to reduce the dynamic range of a signal is to cut the amplitude of the signal off when it reaches or exceeds a certain value. This is the basis of a hard limiter (otherwise known as a "clipper"). The module looks at the input signal; if the input amplitude is greater than or equal to the threshold value, then the output amplitude is set to be that threshold value.
The threshold values used in limiter_module.v are listed below.
hard_limiter_amount[1:0]:
2'b00: No Threshold
2'b01: Threshold = ±1024
2'b10: Threshold = ±512
2'b11: Threshold = ±256

5.5.7 Distortion Module (bitcrusher.v)

One more way to reduce the dynamic range of a signal is to reduce the bit depth of each sample, or the number of bits used to encode each sample. This is the basis of bitcrushing, or introducing distortion into the audio by reducing the bit depth. In the bitcrusher.v, the input bits_to_crush[2:0] determines how many bits to decrease the bit depth of each sample by. The system first does an arithmetic right shift by the number of binary places that bits_to_crush is set to (from 1 bit to 7 bits). Then, the system does an arithmetic left shift by the same number of binary places. This results in the bit depth being reduced; it can be reduced to a minimum of 5 bits. The resulting audio sounds more “lo-fi” and noisy, and reminds one of old audio devices like the Speak-and-Spell toys from the 1970s.

5.6 Graphics Modules (Gerzain)

5.6.1 Overview

The implementation of the graphics module involves being able to display text on the screen using the 6.111 labkit. It is infeasible to use calculations to represent line segments and endpoints that represent general letters and shapes because the labkit itself has a limited amount of onboard block ROM/ RAM capable of storing predefined data. It becomes practical to use the aforementioned memory to store images representing letters and numbers. One aspect to consider is how to allocate that same memory so that different instances of sprites can share the memory resources, thus allowing the capability to display an arbitrarily large amount of digits on the screen.

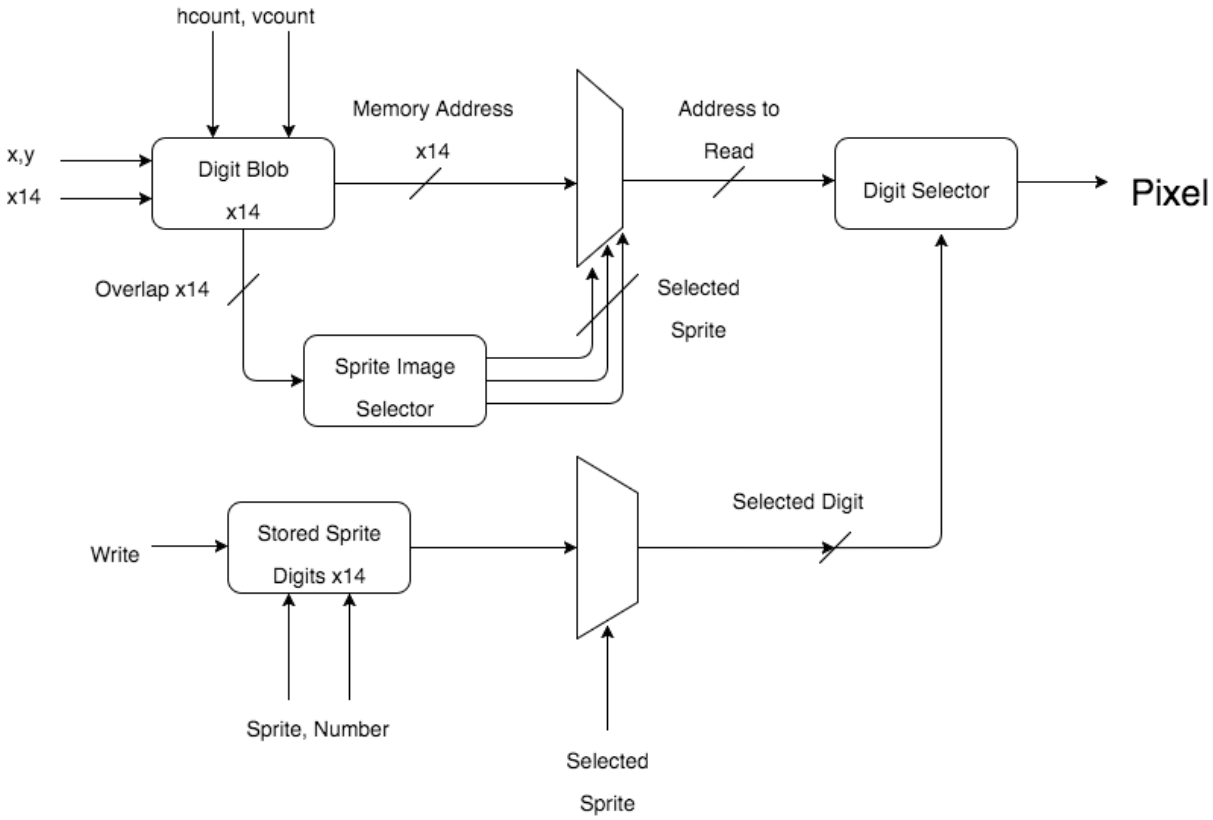

Figure 11: Block Diagram illustrating the behavior of the Graphics Module.

5.6.2 HUD Display

There is a separate image stored that serves as the core component of the HUD. It is simply a 512 by 240 pixel image containing text indicating which statistics the device needs to calculate: maximum frequency, maximum amplitude, elapsed time, number of effects applied, and which song bank the user has selected. The instantiation of the HUD digits module also occurs in this module.

5.6.3 HUD Digits

The instantiation of the digit sprites, the sprite image selector, and the digit selector occur within this module. The inputs include a write bit, three-bit values indicating which sprite and which number to assign to it, hcount, vcount, and the resulting output pixel. The write bit stores the digit number into a given sprite number. Whenever the overlap condition is triggered in any of the digit sprites the encoded four-bit value is output to the sprite image selector.

5.6.4 Sprite Image Selector

The overlap condition output by the digit sprites serves as input to this module. This is essentially a one-hot encoder, whereby whichever digit sprite out of a total of 14 digit sprites overlaps with the hcount and vcount of the screen is then encoded into a three-bit value. One of the underlying assumptions throughout the imaging process is that no digit sprites overlap with each other, which would otherwise mean that two different sprites would trigger a read condition on the shared image ROMs. This can be avoided by spacing the sprites appropriately.

5.6.5 Digit Selector

This module contains the instances of the digit ROMs and the singular color map for the two-to-eight color pixel conversion. A 11-bit address serves as input alongside a four-bit number selection value. A multiplexer then selects the ROM bits to read depending on the digit selected. The data bits from the read memory address are then sent to the color map where the resulting pixel value is sent out to be displayed on the VGA module.

5.6.6 Digit Blobs

Real-time statistics from the input waveform are extracted by the FFT module. This entails the use of individual sprites sharing the digit ROMs whenever the hcount and vcount of the screen intersected with the area covered by the sprite. In the usual implementation of a sprite whenever the current hcount and vcount of the screen intersect with the bounds of a given sprite that sprite outputs a predefined color. Whenever the current hcount and vcount intersect with a given digit sprite it triggers an overlap condition. This overlap condition is used in subsequent modules to select a corresponding digit rom

to display. At the same time the image address to pull the pixel value from is calculated within this module.

5.6.7 Color Map(s)

Ideally, each image displayed on the screen requires its own color map to convert the reduced number of bits per pixel to the 24 bits required by the VGA module. Since the Heads-Up Display alongside the digits were encoded with two bits per pixel and the only color used was red this allowed the use of one color map for most of the images displayed on the screen.

Figure 12: Example Images

There is one instance in which three color maps were necessary to display a secret function of the design. This specific image being displayed is the MIT logo

Figure 13: MIT Logo

Since this image contains equal amounts of the three primary colors along with a specific shade of red it was necessary to use three color maps for this image. Fortunately since there are only three specific colors on this image it's entirely possible to represent this image with two bits and use a 3-element lookup table for each primary color.

5.6.8 Number Digit ROMs

The implementation utilizes block ROM memory modules that are generated by the Xilinx IP Core Wizard to store information corresponding to the individual number digits. Since the Virtex2 FPGA used in the design has limited memory, it is wise to encode individual images with a reduced number of colors; this allows each individual image to take up less space in the board. Ten images representing the digits zero through nine are instantiated, with each pixel being represented with two bits of information. All the digits are colored red to reduce memory space. Ideally, each pixel would be represented with

one pixel. In the implementation, the pixels are encoded with two bits due to the effects of the image conversion. With each pixel extracted from memory, a color map table is necessary to provide the 24 bits of color required by the VGA module. Fortunately, there is no need to implement further color maps for green and blue color values since the digits only used a red color map. This results in decreased memory usage.

5.6.9 Process Audio

This module is primarily tasked with taking in the raw audio bits from the AC97 IC and performing a FFT on the audio stream. This is done through an instantiation of an FFT module which takes in both real and imaginary values in the time domain and outputs the real and imaginary coefficients in the frequency domain. The implementation of the FFT module itself is generated through Xilinx's IP Core generator. The magnitude of the coefficient is calculated from the real and imaginary parts. If the coefficient index is less than the width of the screen (1024 pixels) the calculated magnitude is output alongside the coefficient index. Once the square root of the sum of the squared imaginary and real components

5.6.10 Fast Fourier Transform Display (present in top module)

The FFT waveform is displayed on the screen whenever the vcount and hcount intersect with an area that falls under the spectrum waveform. At each hcount, the module executes a memory read on a 10-bit 2014-address RAM module that stores the calculated coefficients of the FFT. This memory read introduces a one-cycle delay, which is counteracted with a memory read pipeline that offsets the vertical and horizontal blanking periods by that same clock cycle. At the same time, there is a division calculation on every pixel that falls under the waveform. This division calculation utilizes a divider, which is also generated by the IP Core tools. The gradient for a single column is calculated by using the current vcount of the pixel and the coefficient's magnitude. This division also introduces a one cycle delay, which is counteracted with further pipeline stages. The gradient of the coefficient ranges from completely yellow at the bottom of the screen to red at the peaks of the spectrum coefficients.

5.6.11 Square Root

In order to calculate the magnitude of the frequency coefficients from the real and imaginary parts that the FFT sends out, it is necessary to use a square root implementation. This is implemented using a successive approximation algorithm. It is assumed that the square root of a given integer will have at most half the number of bits of the original integer. With this in mind, initially a trial value is assigned. The trial value is initialized such that it is equal to 1 bit shifted to the middle index of the total number of bits the given input value has. If the squared value of this trial number is less than the input value, then it can be added to the stored trial value. There is a bit shift that occurs whereby at each time step the module approaches the square root of the input value.

5.6.12 Heads-Up Display Blob

For the HUD there is a sprite which serves to indicate what the output digits represent. It is a two bits per pixel encoded image displaying the following phrases “Max Frequency”, “Max Amplitude”, “Time”, “FX” (current number of effects applied to the sound), and “Bank” (current song bank being used).

5.6.13 Binary-Coded Decimal Converter

The data provided by both the FSM and the audio modules is encoded in a binary value, whereas each sprite needs a four-digit value to save for later writing on the screen. This module simply converts a 12-bit value to four binary-coded decimals. This module is used by the song bank wires, the calculated maximum amplitude and maximum frequency, and by the elapsed time signals.

5.6.14 Maximum Frequency and Amplitude

During the horizontal scan of the screen, the FFT module accesses the RAM address of the frequency coefficient corresponding to the hcount of the current pixel. Whenever a new pixel is accessed, the hcount is used to access the RAM and obtain the magnitude of the FFT at the specific coefficient. This is used to determine whether or not this new FFT coefficient is larger than the currently saved maximum frequency and amplitude. If the new magnitude is larger than the current maximum amplitude of the current coefficient, then the new coefficient and the current memory address (multiplied by a factor) is used to obtain the maximum frequency and amplitude. During the start of a new scanline the maximum amplitude and magnitude are set to zero, after which the logic

comparison takes place whereby new values can be written to the max_amp and max_freq regs. This occurs whenever the hcount is updated, which means that the module will take into account frequencies larger than the ones being displayed on the screen. The maximum frequency capable of being displayed on the screen is 3 kHz. Since the VGA display has a horizontal resolution that is 1024 pixels wide, then it can be concluded that the FFT resolution is 3 hertz per pixel. Obtaining the maximum frequency can be calculated by multiplying the current largest hcount value by three. As for the maximum amplitude, the output is sent to a binary to decibel converter to obtain the amplitude in decibels.

5.6.15 Sprite Digit Assignment Interval

Whenever the vsync signal is asserted low, the current image frame is written to the screen. It is during this vertical blanking interval that allows for rewriting of the sprite number registers to assign them new numbers to display on the next image frame. Due to the sprite assignment scheme where only one sprite can be changed at a time, it is necessary to write to each sprite sequentially. The writing scheme is clocked using the same 27MHz clock that is used for the FFT module. This allows ample time to write the 14 sprite numbers onto their corresponding regs.

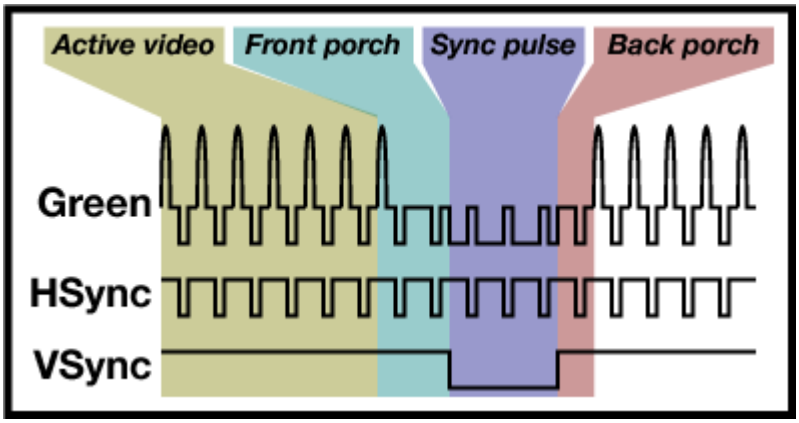

Figure ##: Diagram Showing Vertical Blanking Interval

5.6.16 Super-Secret Pong Mode

Whenever the user enables switches four through two a yellow paddle appears on the left side of the screen and if the audio output is enabled the center MIT logo image sprite starts to move. The image now becomes a puck and the user has the ability to play

pong. The movement speed of the puck depends on the largest coefficient written to the max_amp reg which is set as the x and y increment of the logo puck. The yellow paddle can be moved by pressing on the up and down button located on the 6.111 labkit.

5.6.16.1 MIT Logo Blob

The instantiation of the image ROM containing the data bits for the encoded pixels occurs here. The output of the image ROM is then passed onto three color maps for the red, green, and blue color components. Since this is the only image ROM that corresponds to this sprite the resulting colors from the color map are passed directly as an output pixel.

5.6.16.2 Pong Ball

Whenever the secret mode is activated with the toggle switches the logic controlling how the ball and paddle move is derived from this module. The trigger condition enables the MIT logo sprite to move along with the paddle. If the paddle is able to make contact with the puck (the MIT logo sprite) then it bounces back and continues bouncing off the rest of the bounds until it approaches the left wall again. Whenever the paddle is unable to make contact with the puck then the puck is placed back to its original position in the middle of the screen.

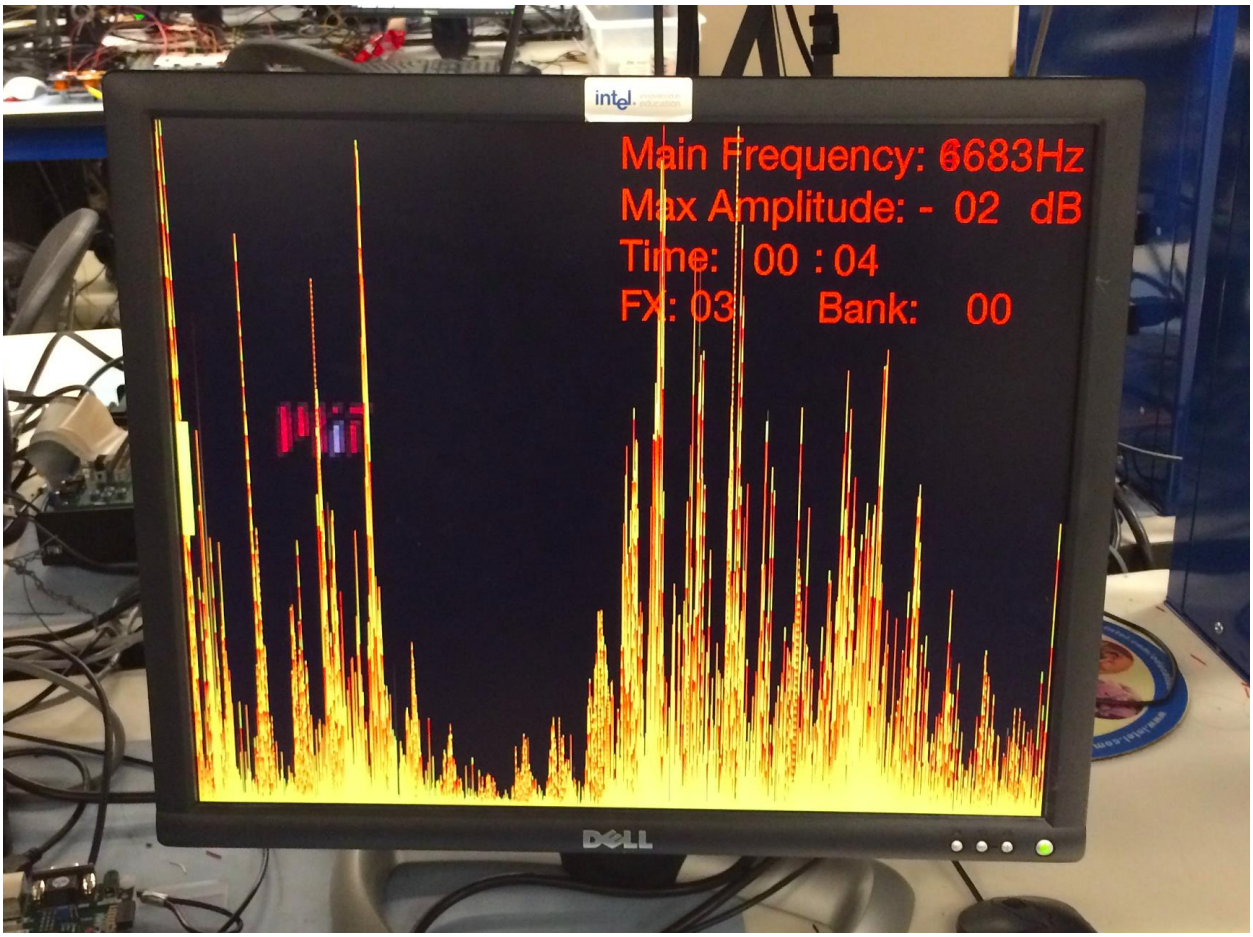

Figure 14: Demonstration of S.S. Pong Mode Enabled.

6. Testbenches and Waveforms

Testing implemented modules became an essential part in deciphering errors present when creating the pieces of the DAW.

6.1 Digit Selector Testbench

Given an X and Y position on the screen, a sprite should take control of the ROMs output lines in order to select the appropriate pixel on the screen. During the implementation stage there were times when only one sprite would display on the screen.

Due to a lack of testbench implementation the error arose relatively late throughout the development process. Nevertheless the error became apparent when designing a testbench for the top module of the digit selector. The end result of the testbench was that it showed only one overlap condition being triggered due to an implementation error in the sprite selector.

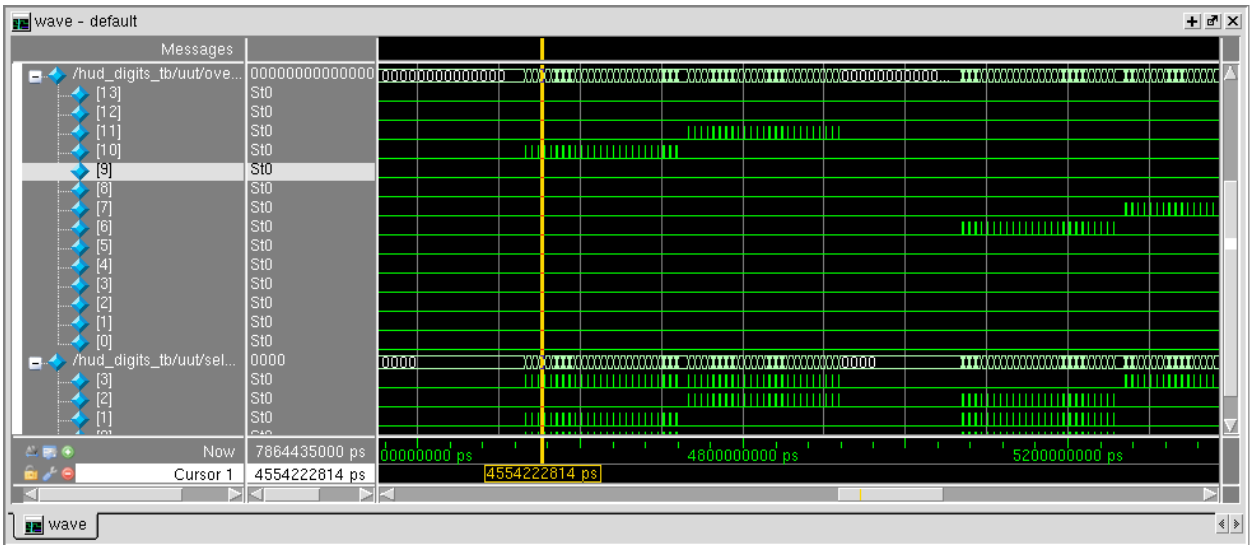

Figure 15: Digit Selector test bench waveform.

The following testbench shows the proper functionality of the binary to BCD decoder. This module is necessary to display the appropriate digits on each sprite.

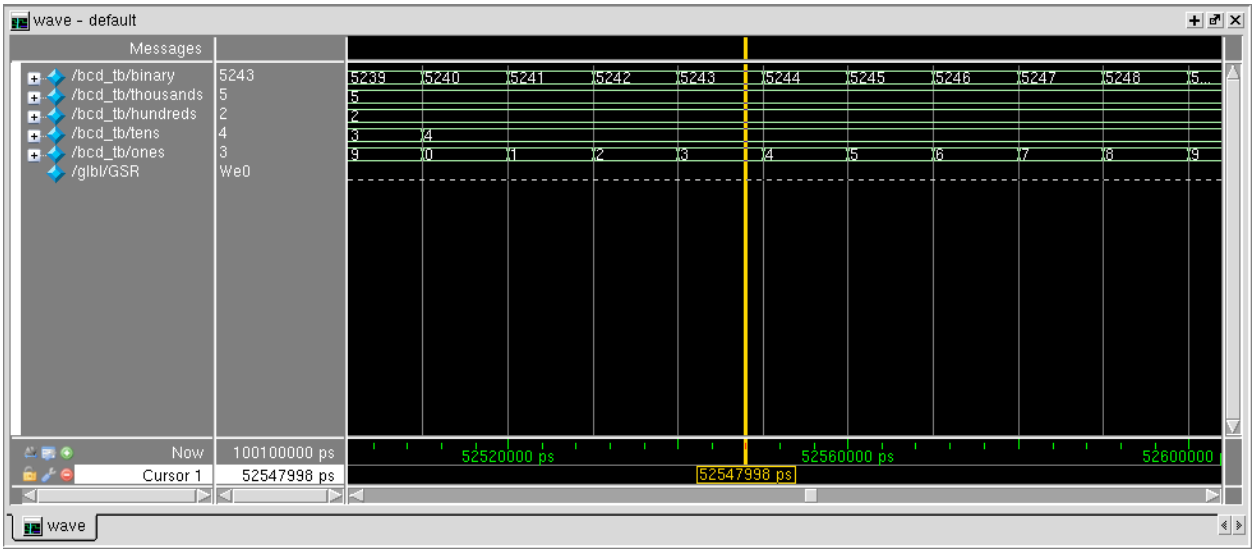

Figure 16: BCD module test bench waveform.

6.2 Testing the Memory

Because the modules directly interfaced with the SRAM on the labkit, creating testbenches for the memory processor was not feasible. Therefore, the memory module had to be tested manually on the labkit. Each song choice was tested to see if it would record and playback accurately and to the correct length, and each song was tested to see if interference from any input signal was interpreted correctly.

6.3 Testing the Audio Modules

The audio modules manipulate the audio samples; these audio samples are represented as signed 12-bit integers. This meant that it was easier to test each module's functionality through simulation. In Lab 5a (the audio processing lab), a test bench for evaluating the fir31.v module was provided along with a text file containing a long list of samples that can be used as input data for testing the module. The test benches for each of the audio effects were based on the test bench provided by Lab 5a; minimal modification of each test benches was required to make sure that certain parts of each module were outputting the correct data at each time for a given input value.

6.4 Testing the Central FSM

The Central FSM Modules were tested by running through all the possible states and transitions in param_select and central_fsm. These modules were tested by looking at the results of the Verilog code on the labkit. Unfortunately, there's no real way to test this module through simulation because the code is dependent on the hardware.

7. Conclusions

The overall design, implementation, and demonstration was a success. This was due in part to the modularization of the project. Each team member was given clearly defined objectives within each area of expertise. The signals necessary to communicate between the Finite State Machine, the memory modules, the audio processing modules, and the graphics modules were clearly defined since the planning phases. This gave each team member clear objectives which proper accountability as the end of the term was approaching.

The integration of the fields of digital signal processing, memory management, and computer graphics are present in the execution of this project. Nevertheless, there are some aspects of the project that could be improved upon. An attempt was made to change the sampling rate on the AC97 chipset which would have allowed longer audio recordings to be stored on the device. There are pipelining issues present when displaying the FFT waveform on the screen. Since the color gradient is calculated from a division whereby the denominator is a location in memory, there is latency introduced by the memory access.

This project serves as a proper platform from which digital audio enthusiasts can experiment with audio effects and frequency domain analysis of audio signals. The ability to record multiple samples, overlay effects on those samples, and having the ability to infer the effects of audio in the frequency spectrum is an invaluable asset to anyone interested improving the field of digital audio signal processing.

8. References

Weistroffer, G., Cooper, J. A., & Tucker, J. H. (2007). Quick and Easy Binary to dB Conversion.

SoutheastCon, 2007. Proceedings. IEEE. Richmond, VA: IEEE.

Giannoulis, D., Massberg., M., & Reiss, J. D. (2012). Digital Dynamic Range Compressor Design -- A Tutorial and Analysis. *Journal of the Audio Engineering Society, 60*(6), 399-408. http://www.aes.org/e-lib/browse.cfm?elib=16354

9. Verilog Code

9.1 Top-level Modules:

```
//////////////////////////////////////////////////////////////////////////////////
//
// Top level Labkit Module (Previous labs modified collaboratively)
//
//////////////////////////////////////////////////////////////////////////////////
module labkit_experiment(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'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 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

     /*
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_we_b = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
*/

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 ram0_we_b = 1'b0;
assign ram1_we_b = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
*/

assign ram1_ce_b = 1'b0;
assign ram1_oe_b = 1'b0;
assign ram1_adv_ld = 1'b0;
assign ram1_bwe_b = 4'h0;
assign clock_feedback_out = 1'b0;
```

```
// 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


// 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;

//////////////////////////////////////////////////////////////////////////
//
// final project: music effects and FFT
//
//////////////////////////////////////////////////////////////////////////

// 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));

// power-on reset generation

wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
                .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

wire [15:0] from_ac97_data, to_ac97_data;
wire ready;

// ENTER button is user reset
     wire vb3;
     wire reset;
     debounce bb3(.reset(reset),.clock(clock_27mhz),
                  .noisy(~button3),.clean(vb3));
assign reset = vb3 | power_on_reset;

          //SYNCHRONIZATION OF switches
  wire [7:0] switch_sync;
  genvar i;
  generate for(i =0; i < 8; i = i+1)
      begin: gen_modules //generate 8 debounce modules
          debounce d(reset,clock_27mhz, switch[i], switch_sync[i]);
      end
  endgenerate

 //debounced user inputs
 // button0, button1, button2, button3, button_enter, button_right,
 // button_left, button_down, button_up, and switches are inputs
 wire vup,vdown,vright,vleft;
 debounce bup(.reset(reset),.clock(clock_27mhz),.noisy(~button_up),.clean(vup));
```

```
   debounce bdown(.reset(reset),.clock(clock_27mhz),.noisy(~button_down),.clean(vdown));
   debounce
bright(.reset(reset),.clock(clock_27mhz),.noisy(~button_right),.clean(vright));
   debounce bleft(.reset(reset),.clock(clock_27mhz),.noisy(~button_left),.clean(vleft));
   wire venter;
   debounce
benter(.reset(reset),.clock(clock_27mhz),.noisy(~button_enter),.clean(venter));
   wire vb0,vb1,vb2;
   debounce bb0(.reset(reset),.clock(clock_27mhz),.noisy(~button0),.clean(vb0));
   debounce bb1(.reset(reset),.clock(clock_27mhz),.noisy(~button1),.clean(vb1));
   debounce bb2(.reset(reset),.clock(clock_27mhz),.noisy(~button2),.clean(vb2));


       wire [12:0] max_freq;
       wire [9:0] max_amp;
  reg [3:0] num, blob;
       reg write;
  reg [1:0] volume_or_selection;

  wire [18:0] mem_address; //memory address to write to
  wire [35:0] mem_read0; //read memory from zbt0
  wire [35:0] mem_read1; //read memory from zbt1
  wire [35:0] mem_write; //data to write to memory
  wire we0; //write enable for zbt0
  wire we1; //write enable for zbt1



       //relevant wires/modules
  wire [6:0] effects; //effects chosen, set by switches from central fsm
  wire [3:0] song_name; //song name, chosen from param and set by cfsm
  wire song_done; //song done, FROM MEMORY
  wire [3:0] song_choice; //song choice, from central fsm to memory
  wire start_song; //start song, from cfsm
  wire pause_song; //pause song, from cfsm
  wire [16:0] effect_values; //effect values, chosen from param and set by cfsm
  wire record_mode; //record mode, chosen from param and set by cfsm
  wire record_mode_sel; //variable record mode from param select
  wire [3:0] song_name_sel; //variable song name sel from param select
  wire [16:0] effect_values_sel; //variable effect values sel from param select
  wire [1:0] cfsm_state; //state of central fsm
                     //00: standby
                     //01: playback
                     //10: record
                     //11: default - standby
  wire blink_fo; //signal that is 1 for 1/4 of a second, 0 for 3/4, from blinkfo
  wire [7:0] seconds; //seconds elapsed since start of song
  wire [15:0] blink_fo_data; //data sent for modified hex display, from param select
       wire signed [11:0] audio_mem_out;
```

```
centralFSM central_fsm (
     .reset(reset),.clk(clock_27mhz), //from fpga, input
     .but_ent(venter),
     .switch(switch_sync),//from user inputs, all sync/deb'd, input
     .effects(effects),/*[6:0]*/
     .song_name(song_name),//to graphics module, output
     .song_done(song_done),//from memory module, input
     .song_choice(song_choice),
     .start_song(start_song),
     .pause_song(pause_song),//to memory module, output
     .effect_values(effect_values),
     .record_mode(record_mode),
     .record_mode_sel(record_mode_sel),
     .song_name_sel(song_name_sel),
     .effect_values_sel(effect_values_sel), //from param_select
     .cfsm_state(cfsm_state),
                    .vb0(vb0)
     );
blink_fo blinkfu(
     .reset(reset),
     .clk(clock_27mhz),
     .blink_fo(blink_fo)
     );
modified_display_16hex disp(reset,clock_27mhz,
    //data!!!!!!!!
     {2'b00,cfsm_state, //2+2
                    4'b0000, //4
                    seconds, //8
                    4'b0000, //4
                    3'b000,record_mode_sel, //3+1
                    4'b0000, //4
                    song_name_sel, //4 - 32
                    4'b0000, //4
     1'b0,effect_values_sel[16:14],//distortion 1+3
     2'b00,effect_values_sel[13:12],//limiter 2+2
     2'b00,effect_values_sel[11:10],//compression 2+2
     3'b000,effect_values_sel[9:5],//chorus 3+5
     3'b000,effect_values_sel[4:0]//echo 3+5
     },
     16'b0100_1010_1000_0000,//for modified hex blank data goes here (preset)
                    //16'b0000_0000_0000_0000, //debug
     blink_fo_data,//for modified hex blinkd ata goes here (from paramsel)
     disp_blank, disp_clock, disp_rs, disp_ce_b,
     disp_reset_b, disp_data_out);
song_timing stiming(
     .reset(reset),
     .clk(clock_27mhz),
     .start_song(start_song), //resets and starts incrementation, from fsm
     .song_done(song_done), //pauses incrementation, from memory
     .pause_song(pause_song), //pauses incrementation, from fsm
```

```
      .seconds(seconds) //8 bits seconds elapsed since start_song, for graphx/display
      );
 param_select parasel(
      .reset(reset),
      .clk(clock_27mhz),
      .blink_fo(blink_fo),
      .b_up(vup),
      .b_down(vdown),
      .b_right(vright),
      .b_left(vleft),
      .blink_fo_data(blink_fo_data), //16
      .song_name_sel(song_name_sel), //4
      .effect_choice_sel(effect_values_sel), //17
      .record_mode_sel(record_mode_sel) //1
      );

addresscalculator addr_calc(.reset(reset), //in
            .clk(clock_27mhz),.ready(ready),.record_mode(record_mode),//in
            .song_choice(song_choice),.start_song(start_song), //in
            .pause_song(pause_song), //in
            .mem_address(mem_address),.song_done(song_done),
                                    .spslsw(switch_sync[7:6]));//out
memprocessor mem_pros(
            .reset(reset),//in
            .clk(clock_27mhz),//in
            .ready(ready),//in
            .audio_in(from_ac97_data),
            .start_song(start_song),//in
            .song_choice(song_choice),
            .record_mode(record_mode),//in
            .pause_song(pause_song),//in
            .mem_read0(mem_read0),//in
            .mem_read1(mem_read1),//in
            .song_done(song_done),//in
            .we0(we0),//out
            .we1(we1),//out
            .mem_write(mem_write),//out
            .audio_out(audio_mem_out)//out
            );//out
//zbt drivers
zbt_6111 zbt0(.clk(clock_27mhz),.cen(1'b1),.we(we0),
            .addr(mem_address),.write_data(mem_write),
            .read_data(mem_read0),
            .ram_clk(ram0_clk),.ram_we_b(ram0_we_b),
            .ram_address(ram0_address),.ram_data(ram0_data),
            .ram_cen_b(ram0_cen_b));
zbt_6111 zbt1(.clk(clock_27mhz),.cen(1'b1),.we(we1),
            .addr(mem_address),.write_data(mem_write),
            .read_data(mem_read1),
            .ram_clk(ram1_clk),.ram_we_b(ram1_we_b),
```

```
            .ram_address(ram1_address),.ram_data(ram1_data),
            .ram_cen_b(ram1_cen_b));



// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync,vsync,blank;
xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
           .hsync(hsync),.vsync(vsync),.blank(blank));

// feed XVGA signals to user's pong game
wire [23:0] hud_pixel;
hud_display hd(.vclock(clock_65mhz),.reset(reset),
             .hcount(hcount),.vcount(vcount),
             .hud_pixel(hud_pixel),.write(write),
             .num(num),.blob(blob));




// allow user to adjust volume with up/down
reg [4:0] volume;
reg old_b2,old_b1;
always @ (posedge clock_27mhz) begin
  if (reset) volume <= 5'd8;
  else begin
     if (vb2 & ~old_b2 & volume != 5'd31) volume <= volume+1;
     if (vb1 & ~old_b1 & volume != 5'd0) volume <= volume-1;
  end
  old_b2 <= vb2;
  old_b1 <= vb1;
end

     initial begin
             num = 0;
             blob = 0;
             write = 0;
             volume = 5'd8;
     end

     // AC97 driver
modifiedlab5audio 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 sample_ready;

     // [0:4] echo - 32
```

```
 // [5:9] chorus - 32
 // [10:11] compression - 4
 // [12:13] limiter - 4
 // [14:16] distortion - 8

     audio_FSM gs_fsm(.clock(clock_27mhz),.reset(reset),.playback(~record_mode),
             .new_sample_ready(ready),.delay_enable(switch_sync[0]),
             .amount_of_delay(effect_values_sel[4:0]),.chorus_enable(switch_sync[1]),
             .compression_enable(switch_sync[2]),
             .compression_amount(effect_values_sel[11:10]),
             .soft_limiter_enable(switch_sync[3]),
             .hard_limiter_enable(switch_sync[4]),
             .hard_limiter_amount(effect_values_sel[13:12]),
             .distortion_enable(switch_sync[5]),
    .distortion_amount(effect_values_sel[16:14]),
             .samples_in(audio_mem_out),
             .to_ac97_data(to_ac97_data),.sample_ready(sample_ready));

// loopback incoming audio to headphones
//assign to_ac97_data = from_ac97_data;


// process incoming audio data, store results in histogram memory
wire [9:0] haddr;
wire [13:0] hdata;
wire hwe,sel;
process_audio a1(clock_27mhz,reset,ready,to_ac97_data,haddr,hdata,hwe);

// 1024x10 histogram memory: A port is write-only, B port is read-only
// use 1Kx(16+2) dual port BRAM
wire [15:0] dout;
RAMB16_S18_S18 histogram(
  .CLKA(clock_27mhz),.ADDRA(haddr),.DIA({2'b0,hdata}),
  .DIPA(2'b0),.WEA(hwe),
  .ENA(1'b1),.SSRA(1'b0),
  .CLKB(clock_65mhz),.ADDRB(hcount),.DOB(dout),
  .DIB(16'b0),.DIPB(2'b0),.WEB(1'b0),.ENB(1'b1),.SSRB(1'b0));

     max_freq_amp ma1 (.hcount(hcount),.amplitude(dout[9:0]),
             .max_freq(max_freq),.max_amp(max_amp));

     wire [3:0] freq_thousands, freq_hundreds, freq_tens, freq_ones;
     bcd my_bcd1 (.binary(max_freq),.thousands(freq_thousands),
             .hundreds(freq_hundreds),.tens(freq_tens),
             .ones(freq_ones));

     wire [3:0] max_amp_thousands, max_amp_hundreds, max_amp_tens, max_amp_ones;

     wire [8:0] calculated_db;
     bcd my_bcd2 (.binary(calculated_db),.thousands(max_amp_thousands),
```

```
                .hundreds(max_amp_hundreds),.tens(max_amp_tens),
                .ones(max_amp_ones));

        signed_binary_12bit_to_dB num_to_db(.clock(clock_65mhz),.reset(reset),
                .start(1'b1),.input_binary({1'b0,max_amp,1'b0}),
                .output_db(calculated_db));

        wire [3:0] seconds_thousands, seconds_hundreds, seconds_tens,seconds_ones;
        bcd
my_bcd3(.binary(seconds),.thousands(seconds_thousands),.hundreds(seconds_hundreds),
        .tens(seconds_tens),.ones(seconds_ones));

        wire [3:0] fx_bank;
        assign fx_bank = switch_sync[7] +
                switch_sync[6] +
                switch_sync[5] +
                switch_sync[4] +
                switch_sync[3] +
                switch_sync[2] +
                switch_sync[1] +
                switch_sync[0];

        wire [3:0] song_tens, song_ones;
        bcd my_bcd4(.binary(song_name),
        .tens(song_tens),.ones(song_ones));

        reg [3:0] write_dig;
        initial begin
                write_dig = 0;
        end
        always @(posedge clock_27mhz) begin
                if(~vsync) begin
                        case (write_dig)
                                1 : begin write <= 1; num <= freq_thousands; blob <= 0;end
                                2 : begin write <= 1; num <= freq_hundreds; blob <= 1;end
                                3 : begin write <= 1; num <= freq_tens; blob <= 2;end
                                4 : begin write <= 1; num <= freq_ones; blob <= 3;end
                                5 : begin write <= 1; num <= max_amp_tens; blob <=4; end
                                6 : begin write <= 1; num <= max_amp_ones; blob <=5; end
                                7 : begin write <= 1; num <= seconds_thousands; blob <=6;
end
                                8 : begin write <= 1; num <= seconds_hundreds; blob <=7;
end
                                9 : begin write <= 1; num <= seconds_tens; blob <=8; end
                                10 : begin write <= 1; num <= seconds_ones; blob <=9; end
                                11 : begin write<= 1; num <= fx_bank; blob <=11; end
                                12 : begin write<= 1; num <= song_tens; blob <=12; end
                                13 : begin write<= 1; num <= song_ones; blob <=13; end
                                default: write <= 0;
                        endcase
```

```
                    write_dig <= write_dig + 1;
                end
        end

 reg[23:0] fft_pixel;
 reg phsync,pvsync,pblank;
 reg xhsync,xvsync,xblank;
 reg yhsync,yvsync,yblank;
 reg [9:0] xvcount;
 reg [9:0] yvcount;

 wire [17:0] dividend;
 wire [17:0] quotient;
 wire [9:0] divisor;
 assign divisor = (dout[9:0] > 767) ? 767 : dout[9:0];
 assign dividend = yvcount * 8'hFF;
 grad_div my_div(.clk(clock_65mhz),.dividend(dividend),.quotient(quotient),
    .divisor(divisor));
 reg [7:0] red_grad_color, green_grad_color;

wire [23:0] secret_pixel;

pong_ball secret_ball (.vsync(vsync),.vclock(clock_65mhz),.up(vup),.down(vdown),
  .reset(vleft), .pspeed({1'b0,max_amp[9:7]}),.hcount(hcount),.vcount(vcount),
  .enabled(switch_sync[2] & switch_sync[3] & switch_sync[4]),
  .pixel(secret_pixel));

 always @ (posedge clock_65mhz) begin
   // first pipe stage: memory access
    yhsync <= hsync;
    yvsync <= vsync;
    yblank <= blank;
    yvcount <= 10'd767 - vcount;
   // second pipe stage: process memory result
    xhsync <= yhsync;
    xvsync <= yvsync;
    xblank <= yblank;
    xvcount <= yvcount;
    green_grad_color <=8'hFF - quotient[7:0];
              red_grad_color <= /*quotient[7:0]*/8'hFF;

   // third pipe stage: write divider to result
   phsync <= xhsync;
   pvsync <= xvsync;
   pblank <= xblank;
   fft_pixel <= xblank ? {24{1'b0}} :
        (dout[9:0] > xvcount) ?
        {red_grad_color,green_grad_color,{8{1'b0}}}:
        {24{1'b0}};
 end
```

```
  // switch[1:0] selects which video generator to use:
  //  00: user's pong game
  //  01: 1 pixel outline of active video area (adjust screen controls)
  //  10: color bars
  reg [23:0] rgb;
  wire border = (hcount==0 | hcount==1023 | vcount==0 | vcount==767);

  reg b,hs,vs;
  always @(posedge clock_65mhz) begin
        // default: pong
        hs <= phsync;
        vs <= pvsync;
        b <= pblank;
        rgb <= fft_pixel | hud_pixel | secret_pixel;
  end

  // VGA Output.  In order to meet the setup and hold times of the
  // AD7125, we send it ~clock_65mhz.
  assign vga_out_red = rgb[23:16];
  assign vga_out_green = rgb[15:8];
  assign vga_out_blue = rgb[7:0];
  assign vga_out_sync_b = 1'b1;    // not used
  assign vga_out_blank_b = ~b;
  assign vga_out_pixel_clock = ~clock_65mhz;
  assign vga_out_hsync = hs;
  assign vga_out_vsync = vs;

  assign led[5:0] = ~{1'b0,volume};
       assign led[6] = ~song_done; //track?!?!
       assign led[7] = ~pause_song; //need to know somehow

endmodule

//yu have been hacked!!!!!!
module modifiedlab5audio (
 input wire clock_27mhz,
 input wire reset,
 input wire [4:0] volume,
 output wire [11:0] audio_in_data,
 input wire [11:0] 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 [11:0] out_data;
 always @ (posedge clock_27mhz)
   if (ready) out_data <= audio_out_data;
 assign audio_in_data = left_in_data[19:8];
 assign left_out_data = {out_data, 8'b00000000};
 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
```

```
//////////////////////////////////////////////////////////////////////////////////
//
// Switch Debounce Module (Taken from previous labs)
//
//////////////////////////////////////////////////////////////////////////////////

module debounce (reset, clock, noisy, clean);

  input reset, clock, noisy;
  output clean;

  reg [18:0] count;
  reg new, clean;

  always @(posedge clock)
    if (reset)
      begin
         count <= 0;
         new <= noisy;
         clean <= noisy;
      end
    else if (noisy != new)
      begin
         new <= noisy;
         count <= 0;
      end
    else if (count == 270000)
      clean <= new;
    else
      count <= count+1;

endmodule
```

9.2 Central FSM Modules

```
//////////////////////////////////////////////////////////////////////////////////
//
// Central FSM Module (Written by Michelle Qiu)
//
//////////////////////////////////////////////////////////////////////////////////

//central fsm
/* :)
controls the whole system
outputs to audio module, graphics module, and memory module
3 states:
standby state
playback state
record state
```

```
*/

module centralFSM(reset, clk, //from fpga, input
     but_ent, switch,//from user inputs, all sync/deb'd, input
     effects,/*[6:0]*/
     song_name,//to graphics module, output
     song_done,//from memory module, input
     song_choice, start_song, pause_song,//to memory module, output
     effect_values,record_mode,
     record_mode_sel, song_name_sel, effect_values_sel, //from param_select
     cfsm_state, vb0
     );
  //fpga inputs
  input reset; //reset signal
  input clk; //clock

  //user inputs, all sync/deb'd, input
  input [7:0] switch; //we only care about switch 7 - 1 if paused
  input but_ent; //start/stop/transition
  reg but_ent_prev;


  //to audio module, output
  output reg [6:0] effects; //effect choice, a function of user input (necessary?????)
                // 0: echo
                // 1: chorus
                // 2: compression
                // 3: limiter
                // 4: distortion
                // 5: speed up 2x
                // 6: slow down 2x
  input [16:0] effect_values_sel; //effect values, effects values, from param_select
  output reg [16:0] effect_values; //controlled value
                // [0:4] echo - 32
                // [5:9] chorus - 32
                // [10:11] compression - 4
                // [12:13] limiter - 4
                // [14:16] distortion - 8

  input record_mode_sel; //1 = record, 0 = playback, from param select
  output reg record_mode; //controlled value

  input [3:0] song_name_sel; //song choice, from param_select
  output reg [3:0] song_name; //name of song, #s 1-12, controlled value
  output reg [3:0] song_choice; //calculated from song name, controlled value

  output reg start_song; //start record/playback,
                         //requires song choice and mode to be set 1 clock cycle before
  reg start_song_prev; //delay for start song
```

```
  output reg pause_song; //pause song!b0 controls in record and playback, otherwise pause
is 1

  input song_done; //abort! abort! stop dem incrementing, from memory

       input vb0; //pause button
       reg vb0_prev; //prev input

  reg reset_delay; //delay for reset

  //fsm
  output reg [1:0] cfsm_state;    //00: standby
                     //01: playback
                     //10: record
                     //11: n/a - standby

   always @(posedge clk) begin
       if (reset) begin
           //delay 1 clock cycle to set params????
           reset_delay <= 1;
       end else if (reset_delay) begin
           reset_delay <= 0;
           but_ent_prev <= but_ent;
           song_choice <= song_name_sel;
           song_name <= song_name_sel; //default is 0 anyways...
           record_mode <= record_mode_sel;
           effect_values <= effect_values_sel;
           pause_song <= 1; //default paused
           start_song_prev <= 0; //:)
           start_song <= 0; //NO
           cfsm_state <= 2'b00; //start in standby
           effects <= switch[6:0];
                              vb0_prev <= vb0;
       end else begin //actual fsm logic
           start_song <= start_song_prev; //delayed by 1 clock cycle
                              but_ent_prev <= but_ent;
                              vb0_prev <= vb0;
           case(cfsm_state)
               2'b01: begin //playback
                   if (start_song_prev) begin //de-assert start next clock cycle
                       start_song_prev <= 0;
                   //check if we should go back to record
                   end else if (song_done) begin  //return to playback and pause
                       cfsm_state <= 2'b00;
                       pause_song <= 1;
                   end else if (but_ent_prev == 0 & but_ent == 1) begin //return to
playback
                                                                        // and pause
                       cfsm_state <= 2'b00;
                       pause_song <= 1;
```

```
                end else if (vb0_prev == 0 & vb0 == 1) pause_song <= ~pause_song;
//else just
                                                                                //up
date pause
            end
            2'b10: begin //record
                if (start_song_prev) begin //de-assert start next clock cycle
                    start_song_prev <= 0;
                //check if we should go back to record
                end else if (song_done) begin //return to playback and pause
                    cfsm_state <= 2'b00;
                    pause_song <= 1;
                end else if (but_ent_prev == 0 & but_ent == 1) begin //return to
playback
                                                                    // and pause
                    cfsm_state <= 2'b00;
                    pause_song <= 1;
                end else if (vb0_prev == 0 & vb0 == 1) pause_song <= ~pause_song;
            end
            default: begin //same as 2'b00/ - standby
                //TBD: SHOULD VALUES BE SET TO DEFAULT IN THIS STATE FOR GERZAIN?
                if (but_ent_prev == 0 & but_ent == 1) begin //transition state
                    //where to transition to
                    if (record_mode_sel) cfsm_state <= 2'b10; //record
                    else cfsm_state <= 2'b01; //playback

                                                        //set values
                    start_song_prev <= 1; //assert start_song on next clock cycle
                    effect_values <= effect_values_sel; //set value
                    song_name <= song_name_sel; //set value
                    effects <= switch[6:0]; //set value
                    pause_song <= 0;
                    if (song_name_sel < 6) song_choice <= song_name_sel; //if so, name
is choice
                    else song_choice <= song_name_sel + 2; //correct nums for mem
address
                                                        //calculator

                    record_mode <= record_mode_sel; //set value
                end else pause_song <= 1;////redundant, but defs needs to be 1
            end //default
        endcase
    end //fsm logic
end //always

endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// Blink Module (Written by Michelle Qiu)
```

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

/*
1 for approx 1/4 of a second
0 for the rest of the second
used to control blinking display
*/
module blink_fo(
   reset,
   clk,
   blink_fo
   );

   input reset;
   input clk;
   output reg blink_fo;

   parameter [22:0] MAX_ONE = 23'b111_1111_1111_1111_1111_1111;
   reg [24:0] counter; // > 1sec but whatever

   always @(posedge clk) begin
     if (reset) begin //reset
        blink_fo <= 0;
        counter <= 25'b0;
     end else begin //logic
        counter <= counter + 1; //increment counter
        if (counter < MAX_ONE) begin // 1/4 of the time
           blink_fo <= 1; //assert 1
        end else begin
           blink_fo <= 0; // 3/4 of the time assert 0
        end
     end //if reset
   end //always

endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// Parameter Select Module (Written by Michelle Qiu)
//
//////////////////////////////////////////////////////////////////////////////////
/*
an fsm that determines parameter selection.
determines song choice, effect choice, and record mode
outputs will be displayed on the 16hex interfance on the labkit.
also determines the whole blink_fo data on the modified display 16 hex module to show
user
       where selection is.
```

```
   data:       {2'b00,state[1:0],4'b0000,seconds[7:0],
   4'b0000,000,play_record,4'b0000,songchoice,4'b0000,effectnums[27:0](tbd)}
   blink_data: 0000_0102_0334_4567
   blank_data: 1011_0101_0111_1111
               state(1), seconds(2), play/record(1), songchoice(1), effectnums(7)

   // [0:4] echo - 32
   // [5:9] chorus - 32
   // [10:11] compression - 4
   // [12:13] limiter - 4
   // [14:16] distortion - 8
*/


module param_select(
   reset,
   clk,
   blink_fo,
   b_up, b_down, b_right, b_left,
   blink_fo_data, //will be needed to be updated every clock cycle!!!!
   song_name_sel, effect_choice_sel, record_mode_sel
   );

   input reset;
   input clk;
   input blink_fo;
   input b_up, b_down, b_right, b_left;
   output reg [15:0] blink_fo_data;
   output reg [3:0] song_name_sel;
   output reg [16:0] effect_choice_sel;
   output reg record_mode_sel;

   reg [2:0] blink_state; //what gets modified.... 8 states WHOOT
   //assigned as 0-1-2-3-4-5-6-7 with 1 being the leftmost change and 7
   //being the rightmost and 0 being none

   reg b_up_prev;
   reg b_down_prev;
   reg b_right_prev;
   reg b_left_prev;

   always @(posedge clk) begin
       if (reset) begin
           blink_fo_data <= 16'b0000_0000_0000_0000;
           blink_state <= 3'b0; //default blank
           song_name_sel <= 4'b0; //default song 0
           effect_choice_sel <= 17'b0; //default no effect changes
           record_mode_sel <= 1; //default record
           //update prevs
           b_up_prev <= b_up;
```

```
    b_down_prev <= b_down;
    b_right_prev <= b_right;
    b_left_prev <= b_left;
end else begin

    //update prevs
    b_up_prev <= b_up;
    b_down_prev <= b_down;
    b_right_prev <= b_right;
    b_left_prev <= b_left;

    //handles state transitions (precedence over changes)
    if (b_right == 1 & b_right_prev == 0) begin
        blink_state <= blink_state + 1;
    end else if (b_left == 1 & b_left_prev == 0) begin
        blink_state <= blink_state - 1;
    end

    else begin //handles incrementation changes (based on state)
        //in every case handle:
        //1. assigns blink_fo_data
        //2. checks for up/down updates and updates

        case (blink_state) //yeah this may have been a bit bulkier/dense than
                           //I thought it would be
                            //better way for logic???? idk
            3'b000: begin //none
                blink_fo_data <= 16'b0; //no blink!
            end
            3'b001: begin //play/record, 0-1
                blink_fo_data <= {5'b0_0000,blink_fo,10'b00_0000_0000};
                if (b_up == 1 & b_up_prev == 0) begin
                    record_mode_sel <= ~record_mode_sel;
                end else if (b_down == 1 & b_down_prev == 0) begin
                    record_mode_sel <= ~record_mode_sel;
                end
            end
            3'b010: begin //song select 0-11
                blink_fo_data <= {7'b000_0000,blink_fo,8'b0000_0000};
                //special because max is 11!!!
                if (b_up == 1 & b_up_prev == 0) begin
                    if (song_name_sel == 11) song_name_sel <= 0;
                    else song_name_sel <= song_name_sel + 1;
                end else if (b_down == 1 & b_down_prev == 0) begin
                    if (song_name_sel == 0) song_name_sel <= 11;
                    else song_name_sel <= song_name_sel - 1;
                end
            end
            3'b111: begin //echo 0-31 0:4
                blink_fo_data <= {14'b00_0000_0000_0000,blink_fo,blink_fo};
```

```
                    if (b_up == 1 & b_up_prev == 0) begin
                        effect_choice_sel[4:0] <= effect_choice_sel[4:0] + 1;
                    end else if (b_down == 1 & b_down_prev == 0) begin
                        effect_choice_sel[4:0] <= effect_choice_sel[4:0] - 1;
                    end
                end
                3'b110: begin //chorus 0-31 5:9
                    blink_fo_data <= {12'b0000_0000_0000,blink_fo,blink_fo,2'b00};
                    if (b_up == 1 & b_up_prev == 0) begin
                        effect_choice_sel[9:5] <= effect_choice_sel[9:5] + 1;
                    end else if (b_down == 1 & b_down_prev == 0) begin
                        effect_choice_sel[9:5] <= effect_choice_sel[9:5] - 1;
                    end
                end
                3'b101: begin //compression 0-3 10:11
                    blink_fo_data <= {11'b000_0000_0000,blink_fo,4'b0000};
                    if (b_up == 1 & b_up_prev == 0) begin
                        effect_choice_sel[11:10] <= effect_choice_sel[11:10] + 1;
                    end else if (b_down == 1 & b_down_prev == 0) begin
                        effect_choice_sel[11:10] <= effect_choice_sel[11:10] - 1;
                    end
                end
                3'b100: begin //limiter 0-3 12:14
                    blink_fo_data <= {10'b00_0000_0000,blink_fo,5'b0_0000};
                    if (b_up == 1 & b_up_prev == 0) begin
                        effect_choice_sel[13:12] <= effect_choice_sel[13:12] + 1;
                    end else if (b_down == 1 & b_down_prev == 0) begin
                        effect_choice_sel[13:12] <= effect_choice_sel[13:12] - 1;
                    end
                end
                default: //3'b011,
                        begin //distortion 0-8 14:16
                    blink_fo_data <= {9'b0_0000_0000,blink_fo,6'b00_0000};
                    if (b_up == 1 & b_up_prev == 0) begin
                        effect_choice_sel[16:14] <= effect_choice_sel[16:14] + 1;
                    end else if (b_down == 1 & b_down_prev == 0) begin
                        effect_choice_sel[16:14] <= effect_choice_sel[16:14] - 1;
                    end
                end //default
            endcase
        end //blinkstate cases
    end //not reset
  end //always
endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// Song Timing Module (Written by Michelle Qiu)
//
//////////////////////////////////////////////////////////////////////////////////
```

```
//calculates elapsed time of song
module song_timing(
   reset,
   clk,
   start_song, //resets and starts incrementation, from fsm
   song_done, //pauses incrementation, from memory
   pause_song, //pauses incrementation, from fsm
   seconds //seconds elapsed since start_song, for graphx/display
   );

   input reset;
   input clk;
   input start_song;
   input song_done;
   input pause_song;
   output reg [7:0] seconds;

  reg [24:0] counter; //1 sec counter
  parameter DELAY = 27000000 -1;

  always @(posedge clk) begin
     if (reset) begin //reset
        seconds <= 4'b0;
        counter <= 25'b0;
     end else begin //else
        if (start_song) begin //reset seconds, counter if start_song
           seconds <= 4'b0;
           counter <= 25'b0;
        end
        else if (~song_done & ~pause_song) begin //if song not done/paused, increment
freely
           if (counter > DELAY) begin //1 second
              counter <= 0; //reset counter
              if (seconds == 8'hFF) seconds <= 8'hFF; //prevent overflow
              else seconds <= seconds + 1; //else increment seconds
           end else begin //increment counter
              counter <= counter + 1;
           end
        end
     end
  end

endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// (modified) Display Hex Module (lab5b module modified by Michelle Qiu)
//
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
```

```
// original specs:
// 6.111 FPGA Labkit -- Hex display driver
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
////////////////////////////////////////////////////////////////////////////
/*
display_16hex disp(reset, clock_27mhz, {my_hex_data,
                     data,
               disp_blank, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_out);

   data:       {2'b00,state[1:0],4'b0000,seconds[7:0],
          4'b0000,000,play_record,4'b0000,songchoice,4'b0000,effectnums[27:0](tbd)}
   blink_data: 0000_0102_0345_6677
   blank_data: 1011_0101_0111_1111
               state(1), seconds(2), play/record(1), songchoice(1), effectnums(7)
                // [0:4] echo - 32
                // [5:9] chorus - 32
                // [10:11] compression - 4
                // [12:13] limiter - 4
                // [14:16] distortion - 8

*/
//modified to handle blank/solid display by mqiu
module modified_display_16hex (reset, clock_27mhz, data,
     blank_data, blink_data,
               disp_blank, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_out);

  input reset, clock_27mhz;    // clock and reset (active high reset)
  input [63:0] data;           // 16 hex nibbles to display
  //modified:
  input [15:0] blank_data; //1 when all blank, 0 otherwise, takes precedence over number
  input [15:0] blink_data; //1 when all lit up, 0 otherwise, takes precedence over blank

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

```
reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

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

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

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

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

assign dreset = (reset_count != 0);

assign disp_clock = ~clock;

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

reg [7:0] state;             // FSM state
reg [9:0] dot_index;         // index to current dot being clocked out
reg [31:0] control;          // control register
```

```
reg [3:0] char_index;        // index of current character
reg [39:0] dots;             // dots for a single digit
reg [3:0] nibble;            // hex nibble of current character
reg [39:0] dots_hex;

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

always @(posedge clock)
  if (dreset)
    begin
       state <= 0;
       dot_index <= 0;
       control <= 32'h7F7F7F7F;
    end
  else



    casex (state)
      8'h00:
        begin
           // Reset displays
           disp_data_out <= 1'b0;
           disp_rs <= 1'b0; // dot register
           disp_ce_b <= 1'b1;
           disp_reset_b <= 1'b0;
           dot_index <= 0;
           state <= state+1;
        end

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

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

      8'h03:
        begin
```

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

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

        8'h05:
          begin
            // Latch the control register data / dot data
            disp_ce_b <= 1'b1;
            dot_index <= 39;          // init for single char
            char_index <= 15;         // start with MS char
            state <= state+1;
            disp_rs <= 1'b0;          // Select the dot register
          end

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

      endcase

  always @ (data or char_index)
    case (char_index) //will index need to be changed???????
      4'h0:             nibble <= data[3:0];
      4'h1:             nibble <= data[7:4];
```

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


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

    always @(posedge clock_27mhz) begin
     //modified:
       if (blink_data[char_index]) begin
           dots <= 40'b11111111_11111111_11111111_11111111_11111111;
       end else if (blank_data[char_index]) begin
           dots <= 40'b00000000_00000000_00000000_00000000_00000000;
       end else begin
           dots <= dots_hex;
       end
    end
endmodule
```

9.3 Memory Modules

```
//////////////////////////////////////////////////////////////////////////////////
//
// Central FSM Module (written by Michelle Qiu)
//
//////////////////////////////////////////////////////////////////////////////////
//address calculator
/* :)
takes in address requests from the FSM
increments the address every 3 cycles of the clock
resets the address to the specified predetermined address
location when start_song is asserted
pauses the address when pause_song is asserted
when address reaches maximum for song_choice during write,
asserts song_done and stops incrementing til start
when address reaches max address during playback, asserts song_done
and stops incrementing til start
*/

module addresscalculator(reset, clk, ready, /*clkmultik,*/ record_mode,
           song_choice, start_song, pause_song,
           mem_address, song_done, spslsw);

   input reset; //reset, from switches
   input clk; // system clock, system
   input ready; //when data is available from ac97, from ac97 modules
   input [3:0] song_choice; //choice of 16 songs, i choose length, from fsm
   input start_song; //start to reset address, from fsm
   input pause_song; //pause to hold address incrementation, from fsm
   input record_mode; //1 if record, 0 if playback, from fsm
   input [1:0] spslsw;         //speed up, slow down
        reg everyotherready; //for slowdown

   output reg [18:0] mem_address; //address in memory that should be accessed
   output reg song_done; //when song has reached some kind of max
                //address (depending on mode), also used to suppress changes

   //start addresses
   parameter SONG1_ADDR = 0;
   parameter SONG2_ADDR = 240000;
   parameter SONG3_ADDR = 288000;
   parameter SONG4_ADDR = 336000;
   parameter SONG5_ADDR = 384000;
   parameter SONG6_ADDR = 432000;
   //max address
   parameter MAX_ADDR = 480000;

   reg [18:0] highest_addr[0:11]; //highest addr recorded for each song
```

```
    reg [3:0] addr_index; //index within highest_addr we are at
    reg [18:0] song_max; //max addr the song can go to when writing
    reg [1:0] counter3; //makes sure address is incremented every 3 sound bits

    reg record_state; //makes sure read/write is constant, fail-safe

    always @(posedge clk) begin
        if (reset) begin //reset
            counter3 <= 0;
            song_done <= 1; //not play a song until start is asserted
            highest_addr[0] <= SONG1_ADDR;
            highest_addr[1] <= SONG2_ADDR;
            highest_addr[2] <= SONG3_ADDR;
            highest_addr[3] <= SONG4_ADDR;
            highest_addr[4] <= SONG5_ADDR;
            highest_addr[5] <= SONG5_ADDR;
            highest_addr[6] <= SONG1_ADDR;
            highest_addr[7] <= SONG2_ADDR;
            highest_addr[8] <= SONG3_ADDR;
            highest_addr[9] <= SONG4_ADDR;
            highest_addr[10] <= SONG5_ADDR;
            highest_addr[11] <= SONG6_ADDR;
            record_state <= record_mode; //set record state
                                everyotherready <= 0; //set
        end else begin
            if (start_song) begin

                record_state <= record_mode; //set record state
                song_done <= 0; //free to start incrementing address, unlocks
functionality
                //uhm long case block :P based on song choice, choose song
                case(song_choice)
                    4'b0000: begin //song 1
                        mem_address <= SONG1_ADDR;
                        song_max <= SONG2_ADDR - 1;
                        if (record_mode) highest_addr[0] <= SONG1_ADDR;
                        addr_index <= 0;
                    end
                    4'b0001: begin //song 2
                        mem_address <= SONG2_ADDR;
                        song_max <= SONG3_ADDR - 1;
                        if (record_mode) highest_addr[1] <= SONG2_ADDR;
                        addr_index <= 1;
                    end
                    4'b0010: begin //song 3
                        mem_address <= SONG3_ADDR;
                        song_max <= SONG4_ADDR - 1;
                        if (record_mode) highest_addr[2] <= SONG3_ADDR;
                        addr_index <= 2;
                    end
```

```
4'b0011: begin //song 4
    mem_address <= SONG4_ADDR;
    song_max <= SONG5_ADDR - 1;
    if (record_mode) highest_addr[3] <= SONG4_ADDR;
    addr_index <= 3;
end
4'b0100: begin //song 5
    mem_address <= SONG5_ADDR;
    song_max <= SONG6_ADDR - 1;
    if (record_mode) highest_addr[4] <= SONG5_ADDR;
    addr_index <= 4;
end
4'b0101: begin //song 6
    mem_address <= SONG6_ADDR;
    song_max <= MAX_ADDR - 1;
    if (record_mode) highest_addr[5] <= SONG6_ADDR;
    addr_index <= 5;
end
4'b1000: begin //song 7
    mem_address <= SONG1_ADDR;
    song_max <= SONG2_ADDR - 1;
    if (record_mode) highest_addr[6] <= SONG1_ADDR;
    addr_index <= 6;
end
4'b1001: begin //song 8
    mem_address <= SONG2_ADDR;
    song_max <= SONG3_ADDR - 1;
    if (record_mode) highest_addr[7] <= SONG2_ADDR;
    addr_index <= 7;
end
4'b1010: begin //song 9
    mem_address <= SONG3_ADDR;
    song_max <= SONG4_ADDR - 1;
    if (record_mode) highest_addr[8] <= SONG3_ADDR;
    addr_index <= 8;
end
4'b1011: begin //song 10
    mem_address <= SONG4_ADDR;
    song_max <= SONG5_ADDR - 1;
    if (record_mode) highest_addr[9] <= SONG4_ADDR;
    addr_index <= 9;
end
4'b1100: begin //song 11
    mem_address <= SONG5_ADDR;
    song_max <= SONG6_ADDR - 1;
    if (record_mode) highest_addr[10] <= SONG5_ADDR;
    addr_index <= 10;
end
4'b1101: begin //song 12
    mem_address <= SONG6_ADDR;
```

```
                song_max <= MAX_ADDR - 1;
                if (record_mode) highest_addr[11] <= SONG6_ADDR;
                addr_index <= 11;
            end
            default: begin //default. no.
                mem_address <= MAX_ADDR;
                song_max <= MAX_ADDR;
            end
        endcase
    end //start song

    //if not paused and not song done and ready increment the
    // address and check if song finished
    else if (~pause_song & ~song_done & ready) begin
                                // every other!
                                everyotherready <= ~everyotherready;
        //increment counter
        if (counter3 == 2) counter3 <= 0; //reset
        else counter3 <= counter3 + 1; //increment

        //increment memaddress if counter3 == 0 if conditions are met
        if (counter3 == 0) begin

            if (record_state) begin //write
                if (mem_address < song_max) begin//increment mem_address &highest
addr
                    mem_address <= mem_address + 1;
                    highest_addr[addr_index] <= highest_addr[addr_index] + 19'b1;
                end else begin //if mem_address hit the song max
                    song_done <= 1; //stop incrementing addresses and writing
                end
            end

            else begin //playback
                if (mem_address < highest_addr[addr_index]) begin//increment mem
addr
                    if (spslsw[1] & ~spslsw[0]) begin //speed up
                        mem_address <= mem_address + 2;
                    end else if (spslsw[0] & ~spslsw[1]) begin //slow down
                        if (everyotherready) mem_address <= mem_address + 1;
                        end else begin //normal increment
                            mem_address <= mem_address + 1;
                        end
                    end else begin //if mem_addr hits the highest_addr recorded
                        song_done <= 1; //stop incrementing addresses
                end
            end

        end //read/write logic
```

```
            end //incrementing address logic

        end //non-reset logic
    end //always block


endmodule //

//////////////////////////////////////////////////////////////////////////////////
//
// Central FSM Module (Written by Michelle Qiu)
//
//////////////////////////////////////////////////////////////////////////////////
//memory processor
/* :)
determines when to write to memory and what should be written/read from memory
a translator between ac97 and the zbt memory output.
does not handle address calculation.
*/

module memprocessor(reset, clk, ready, audio_in, start_song,
     song_choice, record_mode, pause_song,
       mem_read0, mem_read1, song_done,
       we0, we1, mem_write, audio_out);

   input reset; //reset, from switches
   input clk; // system clock, system
   input ready; //when data is available from ac97, from ac97 modules
   input [11:0] audio_in; //audio that comes from ac97 that is written to memory, from
ac97
   input start_song; //reset what goes in/out??? from fsm
   input [3:0] song_choice; //from fsm
   input pause_song; //stop song, leaving everything where it is..., from fsm
   input record_mode; //from fsm, 1 if record, 0 if playback,from fsm

   input song_done; //from addresscalculator
   input [35:0] mem_read0;  //memory read from ram 0
   input [35:0] mem_read1;  //memory read from ram 1

   output reg [35:0] mem_write;//memory that will be written to memory
   output reg signed [11:0] audio_out;//audio that will go out to speaker
   output reg we0; //write enable to zbt bank 0
   output reg we1; //write enable to zbt bank 1

   reg [35:0] last_read_mem; //memory bus to read audio from
                             //(prevents out of order audio samples)
   reg [1:0] counter3; //asserted every 3 24hz clock cycles

   reg record_state;
   reg [3:0]current_song_choice;
```

```
   always @(posedge clk) begin
       if (reset) begin //reset eerything to 0
           mem_write <= 36'b0;
           audio_out <= audio_in;
           we0 <= 0;
           we1 <= 0;
           last_read_mem <= 0;
           counter3 <= 0;
           record_state <= record_mode;
           current_song_choice <= song_choice;
       end else begin


           //write logic
           //write if not paused, song not done, record mode, ready asserted, and
counter3 == 0
           if (~pause_song & ~song_done & record_state & ready & counter3 == 0) begin
               if (current_song_choice[3]) begin //write to correct zbt SRAM
                   we1 <= 1;
                   we0 <= 0;
               end else begin
                   we0 <= 1;
                   we1 <= 0;
               end
           end else begin //else do not write!
               we0 <= 0;
               we1 <= 0;
           end //end writing logic

           //rest of the logic - what to write, what audio is
           if (start_song) begin //start song, restart read/write
              mem_write <= 36'b0;
              last_read_mem <= 36'b0;
              counter3 <= 0;
              record_state <= record_mode;
              current_song_choice <= song_choice;
           end else
           if (ready & ~pause_song & ~song_done) begin //if ready asserted and song
                                                       //not paused or done

               //counter incrementation
               if (counter3 == 2) begin //counter3 ==2, update memory and reset counter3
to 0
                   //read new memory, make sure you read from correct bank, update
last_read_mem
                   if (current_song_choice[3]) last_read_mem <= mem_read1;
                   else last_read_mem <= mem_read0; //update memory if counter

                   counter3 <= 0; //reset counter3 to 0
```

```
               end else counter3 <= counter3 + 1; //increment counter3
               if (~record_state) begin //playback
                  case (counter3) //which part of last_read_mem you should be reading for
audio
                      2'b00: audio_out <= last_read_mem[35:24];
                      2'b01: audio_out <= last_read_mem[23:12];
                      2'b10: audio_out <= last_read_mem[11:0];
                      default: audio_out <= 12'b0; //invalid?
                  endcase
               end else begin //record
                  audio_out <= audio_in;
               end

               //update what will be written to memory, shift the mem_write
               mem_write <= {mem_write[23:0],audio_in};
            end
         end
      end


endmodule //

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

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

module zbt_6111(clk, cen, we, addr, write_data, read_data,
                ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

  input clk;                   // system clock
  input cen;                   // clock enable for gating ZBT cycles
  input we;                    // write enable (active HIGH)
  input [18:0] addr;           // memory address
```

```
  input [35:0] write_data;     // data to write
  output [35:0] read_data;     // data read from memory
  output        ram_clk;       // physical line to ram clock
  output        ram_we_b;      // physical line to ram we_b
  output [18:0] ram_address;   // physical line to ram address
  inout [35:0]  ram_data;      // physical line to ram data
  output        ram_cen_b;     // physical line to ram clock enable

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

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

  reg [1:0]   we_delay;

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

  // create two-stage pipeline for write data

  reg [35:0]  write_data_old1;
  reg [35:0]  write_data_old2;
  always @(posedge clk)
    if (cen)
      {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

  // wire to ZBT RAM signals

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

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

endmodule // zbt_6111
```

9.4 Audio Modules

```
//////////////////////////////////////////////////////////////////////////////
//
// audio_FSM Module (Written by Germain Martinez)
//
```

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

module audio_FSM
   (input clock,
   input reset,
   input playback,
   input new_sample_ready,
   input delay_enable,
   input [4:0] amount_of_delay,
   input chorus_enable,
   input compression_enable,
   input [1:0] compression_amount,
   input soft_limiter_enable,
   input hard_limiter_enable,
   input [1:0] hard_limiter_amount,
   input distortion_enable,
   input [2:0] distortion_amount,
   input signed [11:0] samples_in,
   output signed [11:0] to_ac97_data,
   output sample_ready);

   reg start_effects_modules;

   wire signed [11:0] delay_applied_sample;
   wire delay_done;
   delay_module delay(.clock(clock), .reset(reset), .start(start_effects_modules),
       .incoming_sample(samples_in), .delay_amount(amount_of_delay),
       .enable(delay_enable), .modified_sample(delay_applied_sample),
       .done(delay_done));


   wire signed [11:0] chorus_applied_sample;
   wire chorus_done;
   chorus_effect chorus(.clock(clock), .reset(reset), .start(delay_done),
       .incoming_sample(delay_applied_sample), .enable(chorus_enable),
       .modified_sample(chorus_applied_sample), .done(chorus_done));


   wire compression_done;
   wire signed [11:0] compression_applied_sample;
   compression compress(.clock(clock), .reset(reset), .start(chorus_done),
       .incoming_sample(chorus_applied_sample),
       .compression_amount(compression_amount), .soft_limiter(soft_limiter_enable),
       .enable(compression_enable), .modified_sample(compression_applied_sample),
       .done(compression_done));


   wire hard_limiter_done;
   wire signed [11:0] hard_limiter_applied;
   limiter_module hard_limiter(.clock(clock), .reset(reset),
```

```
        .start(compression_done), .incoming_sample(compression_applied_sample),
        .limiting_amount(hard_limiter_amount), .enable(hard_limiter_enable),
        .modified_sample(hard_limiter_applied), .done(hard_limiter_done));

    bitcrusher buttcrush(.clock(clock), .reset(reset), .start(hard_limiter_done),
        .enable(distortion_enable), .bits_to_crush(distortion_amount),
        .incoming_sample(hard_limiter_applied),.modified_sample(to_ac97_data),
        .done(sample_ready));

    always @(posedge clock) begin
       if (reset) begin
          start_effects_modules <= 1'b0;
       end

       start_effects_modules <= playback && new_sample_ready;

    end

endmodule


//////////////////////////////////////////////////////////////////////////////////
//
// to_fft_selector Module (Written by Germain Martinez)
//
//////////////////////////////////////////////////////////////////////////////////

module to_fft_selector #(parameter PLAYBACK = 1'b1,
        parameter RECORD = 1'b0)(
        input [11:0] from_fir_germain,
        input [11:0] from_ac97_audio,
        input [1:0] input_mode,
        output reg [11:0] to_fft);

        always @(input_mode) begin
                case(input_mode)
                        1:

        end

endmodule
//////////////////////////////////////////////////////////////////////////////////
//
// signed_binary_12bit_to_dB Module (Written by Germain Martinez)
//
//////////////////////////////////////////////////////////////////////////////////

module signed_binary_12bit_to_dB
    (input clock,
```

```
input reset,
input start,
input signed [11:0] input_binary,
output reg signed [8:0] output_db,
output reg done);

// This module takes a 12-bit integer and converts it to NEGATIVE dB.
// The maximum dB value we expect is 0dB. Anything else is considered to be
// a NEGATIVE NUMBER even though it's encoded as a positive integer.
// This module implements an algorithm that calculates dB of a binary number
// to a precision of 0.5 dB (maximum error is found to be 0.5dB, which is
// fine for this project).

reg [11:0] absolute_value_input;
reg [3:0] msb=4'd11;
reg [3:0] r;

wire [8:0] g_R;
wire [8:0] effective_g_M;
reg [1:0] state;

parameter IDLE=2'b00;
parameter FIND_MSB=2'b01;
parameter OUTPUT=2'b10;

lookup_g_R lookup1(.r(r), .g_R(g_R));
lookup_effective_g_M lookup2(.msb(msb), .effective_g_M(effective_g_M));

always @(posedge clock) begin
   if (reset) begin
      absolute_value_input <= 12'b0;
      msb <= 4'd11;
      r <= 4'b0;
      state <= IDLE;
   end

   case (state)
   // We want to take the input and find the absolute value of it.
   IDLE: begin
      done <= 1'b0;
      if (start) begin
         if (input_binary[11] == 1'b1) begin
            absolute_value_input <= -input_binary;
         end
         else begin
            absolute_value_input <= input_binary;
         end
         state <= FIND_MSB;
         msb <= 4'd11;
      end
```

```
        end

        // Find the most important bits (the most significant one and the four
        // bits after that (if there are any)
        FIND_MSB: begin
           if (absolute_value_input[11] == 1'b1) begin
              if (msb > 3) r <= absolute_value_input[10:7];
              // Account for the edge cases of msb being less than
              // or equal to 3
              else if (msb == 3) r <= {1'b0, absolute_value_input[10:8]};
              else if (msb == 2) r <= {2'b0, absolute_value_input[10:7]};
              else if (msb == 1) r <= {3'b0, absolute_value_input[10]};
              state <= OUTPUT;
           end
           else begin
              absolute_value_input <= absolute_value_input << 1;
              if (msb > 0) msb <= msb - 1;
              else begin
                 msb <= 0;
                 r <= 4'b0;
                 state <= OUTPUT;
              end
           end
        end
        // Once the corresponding values have been found in their
        // lookup tables, you can combine them and divide by 4 to get
        // the actual magnitude of the sample in dB.
        OUTPUT: begin
           output_db <= (effective_g_M - g_R) / 4;
           done <= 1'b1;
           state <= IDLE;
        end
        default: begin
           if (start) begin
              if (input_binary[11] == 1'b1) begin
                 absolute_value_input <= -input_binary;
              end
              else begin
                 absolute_value_input <= input_binary;
              end
              state <= FIND_MSB;
              msb <= 4'd11;
              done <= 1'b0;
           end
        end
        endcase
     end
endmodule

// These are the lookup tables for the signed_binary_to_dB module.
```

```
module lookup_g_R
   (input [3:0] r,
   output reg [8:0] g_R);

   always @(r) begin
      case (r)
      4'd0: g_R = 9'd0;
      4'd1: g_R = 9'd2;
      4'd2: g_R = 9'd4;
      4'd3: g_R = 9'd6;
      4'd4: g_R = 9'd8;
      4'd5: g_R = 9'd9;
      4'd6: g_R = 9'd11;
      4'd7: g_R = 9'd13;
      4'd8: g_R = 9'd14;
      4'd9: g_R = 9'd16;
      4'd10: g_R = 9'd17;
      4'd11: g_R = 9'd18;
      4'd12: g_R = 9'd19;
      4'd13: g_R = 9'd21;
      4'd14: g_R = 9'd22;
      4'd15: g_R = 9'd23;
      endcase
   end
endmodule

module lookup_effective_g_M
   (input [3:0] msb,
   output reg [8:0] effective_g_M);

   always @(msb) begin
      case (msb)
      4'd0: effective_g_M = 9'd265;
      4'd1: effective_g_M = 9'd241;
      4'd2: effective_g_M = 9'd217;
      4'd3: effective_g_M = 9'd193;
      4'd4: effective_g_M = 9'd169;
      4'd5: effective_g_M = 9'd145;
      4'd6: effective_g_M = 9'd120;
      4'd7: effective_g_M = 9'd96;
      4'd8: effective_g_M = 9'd72;
      4'd9: effective_g_M = 9'd48;
      4'd10: effective_g_M = 9'd24;
      4'd11: effective_g_M = 9'd0;
      default: effective_g_M = 9'd0;
      endcase
   end
endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
//
// bitcrusher Module (Written by Germain Martinez)
//
////////////////////////////////////////////////////////////////////////////////


module bitcrusher
   (input clock,
   input reset,
   input start,
   input enable,
   input [2:0] bits_to_crush,
   input signed [11:0] incoming_sample,
   output reg signed [11:0] modified_sample,
   output reg done);

   reg state;
   parameter  IDLE=1'b0;
   parameter  BITCRUSH=1'b1;

   reg signed [11:0] sample_to_bitcrush=12'b000;



   always @(posedge clock) begin
      if (reset) begin
         modified_sample <= 12'h000;
         done <= 1'b0;
         sample_to_bitcrush <= 12'h000;
         state <= IDLE;
      end

      if (start) begin
         case (enable)

         1'b0: begin
            modified_sample <= incoming_sample;
            state <= IDLE;
            done <= 1'b1;

         end
         1'b1: begin
            case (state)

               IDLE: begin
                  done <= 1'b0;
                  if (bits_to_crush>=3'b001) begin
                     sample_to_bitcrush <= incoming_sample >>> bits_to_crush;
```

```
                    state <= BITCRUSH;
                end
                else begin
                    modified_sample <= incoming_sample;
                    state <= IDLE;
                end

            end
            BITCRUSH: begin
                modified_sample <= sample_to_bitcrush <<< bits_to_crush;
                done <= 1'b1;
                state <= IDLE;
            end
         endcase
      end
      endcase
   end
end
endmodule

////////////////////////////////////////////////////////////////////////////////
//
// Chorus_effect Module
//
////////////////////////////////////////////////////////////////////////////////

module chorus_effect
   (input clock,
   input reset,
   input start,
   input enable,
   input signed [11:0] incoming_sample,
   output signed [11:0] modified_sample,
   output done);


   // The Chorus effect is achieved by concatenating 4 echo modules,
   // each of which having a different delay_amount.


   // 30 ms delay
   wire signed [11:0] delay_sample_1;
   wire delay_1_done;
   delay_module delay_1(.clock(clock), .reset(reset), .start(start),
       .incoming_sample(incoming_sample), .delay_amount(5'b00011),
       .enable(enable), .modified_sample(delay_sample_1),
       .done(delay_1_done));
```

```
    // 70 ms delay
    wire signed [11:0] delay_sample_2;
    wire delay_2_done;
    delay_module delay_2(.clock(clock), .reset(reset), .start(delay_1_done),
        .incoming_sample(delay_sample_1), .delay_amount(5'b00111),
        .enable(enable), .modified_sample(delay_sample_2),
        .done(delay_2_done));


    // 150 ms delay
    wire signed [11:0] delay_sample_3;
    wire delay_3_done;
    delay_module delay_3(.clock(clock), .reset(reset), .start(delay_2_done),
        .incoming_sample(delay_sample_2), .delay_amount(5'b01111),
        .enable(enable), .modified_sample(delay_sample_3),
        .done(delay_3_done));


    // 310 ms delay
    delay_module delay_4(.clock(clock), .reset(reset), .start(delay_3_done),
        .incoming_sample(delay_sample_3), .delay_amount(5'b11111),
        .enable(enable), .modified_sample(modified_sample),
        .done(done));


endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// compression Module (Written by Germain Martinez)
//
//////////////////////////////////////////////////////////////////////////////////

module compression #(parameter SAMPLING_RATE=24000)
    (input clock,
    input reset,
    input start,
    input signed [11:0] incoming_sample,
    input [1:0] compression_amount,
    input soft_limiter,
    input enable,
    output reg signed [11:0] modified_sample,
    output reg done);

    wire db_convert_done;
    wire signed [8:0] sample_db;

    reg start_compression;

    wire signed [11:0] outgoing_sample;
```

```
wire compression_done;

// Always check to see if compression is enabled.
// If it is not enabled, the input samples should be the same
// as the output samples.
always @(posedge clock) begin
   if (reset) begin
      start_compression <= 1'b0;
      modified_sample <= 12'h000;
      done <= 1'b0;

   end

   case (enable)
   1'b0: begin
      start_compression <= 1'b0;
      modified_sample <= incoming_sample;
      done <= 1'b1;
   end


   // If compression is enabled, check to see if the compression
   // amount is not zero. If it is actually a number, we can
   // start up the compression module and wait until the
   // compression module is finished to output the done signal.
   1'b1: begin
      if (compression_amount == 2'b00) begin
         modified_sample <= incoming_sample;
         start_compression <= 1'b0;
         done <= 1'b1;
      end
      else begin
         start_compression <= 1'b1;
         modified_sample <= outgoing_sample;
         done <= compression_done;
      end

   end
   endcase
end

signed_binary_12bit_to_dB converter(.clock(clock), .reset(reset),
    .start(start_compression), .input_binary(incoming_sample), .output_db(sample_db),
    .done(db_convert_done));


wire gain_computer_done;
wire signed [8:0] computed_db;
wire signed [8:0] computed_level;
compression_gain_computer gain_computer(.clock(clock), .reset(reset),
```

```
        .start(db_convert_done), .compression_amount(compression_amount),
        .input_db(sample_db), .output_db(computed_db),
        .output_level(computed_level), .done(gain_computer_done));


    wire level_detector_done;
    wire signed [8:0] calculated_gain;
    compression_level_detector level_detector(.clock(clock), .reset(reset),
        .start(gain_computer_done), .soft_limiter(soft_limiter),
        .input_level(computed_level), .output_gain(calculated_gain),
        .done(level_detector_done));



    variable_gain apply_gain(.clock(clock), .reset(reset), .start(level_detector_done),
       .incoming_sample(incoming_sample), .input_gain(calculated_gain),
       .compressed_sample(outgoing_sample), .done(compression_done));

endmodule
//////////////////////////////////////////////////////////////////////////////////
//
// compression_gain_computer Module (Written by Germain Martinez)
//
//////////////////////////////////////////////////////////////////////////////////

module compression_gain_computer #(parameter THRESHOLD=18)
    (input clock,
    input reset,
    input start,
    input [1:0] compression_amount,
    input signed [8:0] input_db,
    output reg signed [8:0] output_db,
    output reg signed [8:0] output_level,
    output reg done);

    reg [1:0] state;
    reg [8:0] threshold_amount;
    reg signed [10:0] modified_db;

    reg [8:0] calculate_threshold_gain;

    parameter IDLE=2'b00;
    parameter INPUT_LESS_THAN_THRESHOLD=2'b01;
    parameter INPUT_GREATER_THAN_THRESHOLD=2'b10;
    parameter COMPUTE_LEVEL=2'b11;


    parameter LIMITER=1'b1;
    parameter COMPRESSOR=1'b0;
```

```
always @(posedge clock) begin

   if (reset) begin
      state <= IDLE;
      threshold_amount <= (THRESHOLD << compression_amount);
      output_db <= 9'h000;
      done <= 1'b0;
   end

   case (state)

   IDLE: begin
      done <= 1'b0;
      if (start) begin
         if (input_db >= THRESHOLD) state <= INPUT_LESS_THAN_THRESHOLD;
         else begin
         // Remember that input_db represents a NEGATIVE number in dB,
         // and that the maximum value we can encode in dB is 0dB.
             calculate_threshold_gain <= (threshold_amount + (input_db - THRESHOLD));
             state <= INPUT_GREATER_THAN_THRESHOLD;
         end
      end
   end

   INPUT_LESS_THAN_THRESHOLD: begin
      output_db <= input_db;
      state <= COMPUTE_LEVEL;
   end

   INPUT_GREATER_THAN_THRESHOLD: begin
      output_db <= calculate_threshold_gain >> compression_amount;
      state <= COMPUTE_LEVEL;
   end

   COMPUTE_LEVEL: begin
      output_level <= input_db - output_db;
      done <= 1'b1;
      state <= IDLE;
   end

   default: begin
      if (start) begin
         done <= 1'b0;
         if (input_db >= THRESHOLD) state <= INPUT_LESS_THAN_THRESHOLD;
         else begin
         // Remember that input_db represents a NEGATIVE number in dB,
         // and that the maximum value we can encode in dB is 0dB.
             output_db <= (threshold_amount + (input_db - THRESHOLD));
             state <= INPUT_GREATER_THAN_THRESHOLD;
```

```
            end
         end
      end

      endcase


   end

endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// delayModule (Written by Germain Martinez)
//
//////////////////////////////////////////////////////////////////////////////////

module delay_module #(parameter SAMPLING_RATE=24000, SAMPLES=240)
   (input clock,
   input reset,
   input start,
   input signed [11:0] incoming_sample,
   input [4:0] delay_amount,
   input enable,
   output reg signed [11:0] modified_sample=12'h000,
   output reg done);

   // Need to store anywhere between 10 ms worth of samples and
   // 320 ms worth of samples into BRAM. A better implementation would just pull
   // past samples from the ZBT memory, but I'm assuming that I don't have any
   // access to the ZBT memory.

   reg [12:0] addr=13'h0000;

   reg [12:0] current_pointer=12'h000;
   reg [12:0] delayed_pointer=12'h000;


   reg write=1'b0;
   reg signed [11:0] mem_in=12'h000;
   wire signed [11:0] mem_out;


   reg [12:0] wait_for_memory=13'h0000;

   // This reg stores a version of the stored, delayed sample that is multiplied by 7.

   reg signed [14:0] stored_sample=15'h0000;

   // To have a delay from 10 ms to 320 ms, need to count
```

```
// up to 32 *0.01 s * 240 samples per 0.01 s = 7680 samples. This means
// we need log(7680) / log(2) = about 13 bits for the memory address bank.
// Of course, that means we have 8192 memory locations by 12 bits
// worth of memory to work with.

mybram #(.LOGSIZE(13),.WIDTH(12))
   store_delay_samples(.addr(addr),.clk(clock),
   .we(write),.din(mem_in),.dout(mem_out));

reg [2:0] delay_state=3'b000;

parameter IDLE=3'b000;
parameter READ_DELAYED_SAMPLE=3'b001;
parameter SCALE_DELAYED_SAMPLE=3'b010;
parameter COMBINE_DELAYED_SAMPLE=3'b100;
parameter WAIT_ONE_SAMPLE=3'b101;
parameter WAIT_ANOTHER_SAMPLE=3'b110;

parameter GARBAGE_MEMORY=3'b111;

// This thing has 5 states:
// 00: do nothing until ready is asserted.
// 01: start up the delay effects, write current sample into memory location
// 02: read sample from delayed memory location
// 03: combine sample from delayed memory location with current sample.

always @(posedge clock) begin


   // If we change any parameters, we should engage reset and bring the echo
   // module back to normal.
   if (reset) begin
      current_pointer <= 12'h000;
      delayed_pointer <= 12'h000;
      modified_sample <= 12'h000;
      addr <= 12'h000;
      write <= 1'b0;
      stored_sample <= 15'h0000;
      done <= 1'b0;
      wait_for_memory <= 12'b0;
   end


   case(enable)

   1'b0: begin
      modified_sample <= incoming_sample;
      done <= 1'b1;
   end
```

```
1'b1: begin

   // If we don't set a delay amount in, then the incoming and outgoing
   // samples should be exactly the same.
   if (delay_amount == 5'b0) begin
      modified_sample <= incoming_sample;
      done <= 1'b1;
   end

   // The way echo works is through this difference equation:
   // y[n] = x[n] + c*y[n-m], where m is delay_amount,
   // x[n] is incoming_sample, y is modified_sample, and
   // c = a coefficient between 0 and 1 (I used 7/8).
   else begin
      case(delay_state)
         IDLE: begin
            if (start) begin
               done <= 1'b0;
               current_pointer <= current_pointer + 13'h1;
               delayed_pointer <= current_pointer - (SAMPLES*delay_amount);
               write <= 1'b0;
               if (wait_for_memory < (SAMPLES*delay_amount)) begin
                  wait_for_memory <= wait_for_memory + 13'h1;
                  delay_state <= GARBAGE_MEMORY;
               end
               else delay_state <= READ_DELAYED_SAMPLE;
            end
         end

         READ_DELAYED_SAMPLE: begin
            addr <= delayed_pointer;
            write <= 1'b0;
            delay_state <= WAIT_ONE_SAMPLE;
         end

         SCALE_DELAYED_SAMPLE: begin
            stored_sample <= mem_out;
            delay_state <= COMBINE_DELAYED_SAMPLE;
         end

         WAIT_ONE_SAMPLE: delay_state <= WAIT_ANOTHER_SAMPLE;

         WAIT_ANOTHER_SAMPLE: delay_state <= SCALE_DELAYED_SAMPLE;


         COMBINE_DELAYED_SAMPLE: begin
            modified_sample <= (incoming_sample >>> 1) + (stored_sample >>> 1);
            addr <= current_pointer;
            write <= 1'b1;
            mem_in <= (incoming_sample >>> 1) + (stored_sample >>> 1);
```

```
                delay_state <= IDLE;
                done <= 1'b1;
            end

            GARBAGE_MEMORY: begin
                addr <= current_pointer;
                write <= 1'b1;
                mem_in <= incoming_sample >>> 1;
                modified_sample <= incoming_sample >>> 1;
                delay_state <= IDLE;
                done <= 1'b1;
            end

            default: begin
                if (start) begin
                    done <= 1'b0;
                    current_pointer <= current_pointer + 12'd1;
                    delayed_pointer <= current_pointer - (SAMPLES*delay_amount);
                    write <= 1'b0;
                    if (wait_for_memory < (SAMPLES*delay_amount)) begin
                        wait_for_memory <= wait_for_memory + 13'h1;
                        delay_state <= GARBAGE_MEMORY;
                    end
                    else delay_state <= READ_DELAYED_SAMPLE;
                end
            end
        endcase
    end
  end
  endcase
 end
endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// Limiter Module (Written by Germain Martinez)
//
//////////////////////////////////////////////////////////////////////////////////

module limiter_module #(parameter SAMPLING_RATE=24000)
 (input clock,
 input reset,
 input start,
 input signed [11:0] incoming_sample,
 input [1:0] limiting_amount,
 input enable,
 output reg signed [11:0] modified_sample,
 output reg done
 );
```

```
// Note: since we are using a signed 12-bit encoding, remember that
// the encoded bit values can be from -(2^(n-1)) = -2048 to
// 2^(n-1) - 1 = 2047 (where n is the number of bits)


// This module implements "hard limiting", which clips the signal at
// a certain amplitude threshold that we set with the switches (limiting_amount)

reg [11:0] last_sample;

always @(posedge clock) begin
   if (reset) begin
      modified_sample <= 12'h000;
      done <= 1'b0;
      last_sample <= 12'h000;
   end


   case (enable)
   1'b0: begin
      modified_sample <= incoming_sample;
      done <= 1'b1;
   end

   1'b1: begin
      if (start) begin
         // Take one clock cycle up. Compare the last sample to this sample.
         // If they are not the same, then our results need to be
         // recalculated in the next clock cycle.
         case(limiting_amount)

         // In this setting, the limiter should do nothing.
         2'b00: begin
             modified_sample <= incoming_sample;
         end

         // This is the "90% limiting" setting.
         // If the incoming sample is larger than 90% of the maximum amplitude
         // we can encode, then we cut it off at 90%.
         2'b01: begin
            if (incoming_sample > 1024) modified_sample <= 1024;
            else if (incoming_sample < -1024) modified_sample <= -1024;
            else modified_sample <= incoming_sample;
         end

         // This is the "75% limiting" setting.
         // If the incoming sample is larger than 75% of the maximum amplitude
         // we can encode, then we cut it off at 75%.
         2'b10: begin
            if (incoming_sample > 512) modified_sample <= 512;
```

```
            else if (incoming_sample < -512) modified_sample <= -512;
            else modified_sample <= incoming_sample;
         end

         // This is the "50% limiting" setting.
         // If the incoming sample is larger than 50% of the maximum amplitude
         // we can encode, then we cut it off at 50%.
         2'b11: begin
            if (incoming_sample > 256) modified_sample <= 256;
            else if (incoming_sample < -256) modified_sample <= -256;
            else modified_sample <= incoming_sample;
         end
         endcase
         // In any case, it should take us one clock cycle to do
         // the compare. We can now assert done.
         done <= 1'b1;
      end
   end
   endcase
end
endmodule


////////////////////////////////////////////////////////////////////////////////
//
// variable_gain Module (Written by Germain Martinez)
//
////////////////////////////////////////////////////////////////////////////////


module variable_gain
   (input clock,
   input reset,
   input start,
   input signed [11:0] incoming_sample,
   input signed [8:0] input_gain,
   output reg signed [11:0] compressed_sample,
   output reg done);

   reg signed [8:0] modified_input_gain;
   reg signed [8:0] modified_input_sample;
   reg signed [12:0] multiplication_factor;

   reg signed [24:0] scaled_sample;


   reg [1:0] state;
```

```
// States that the variable_gain machine can be in.
parameter IDLE=2'b00;
parameter FIND_GAIN=2'b01;
parameter APPLY_GAIN=2'b10;
parameter DIVIDE_AND_OUTPUT=2'b11;

always @(posedge clock) begin

   // If the module needs to be reset, then bring all registers
   // to a known state.
   if (reset) begin
      compressed_sample <= 12'h000;
      done <= 1'b0;
      state <= IDLE;
      modified_input_gain <= 9'h000;
      modified_input_sample <= 9'h000;
      multiplication_factor <= 9'sd256;
      scaled_sample <= 25'b0;

   end

   case (state)



   IDLE: begin

      if (start) begin
         modified_input_gain <= input_gain >>> 1;
         state <= FIND_GAIN;
         done <= 1'b0;
      end
   end

   // Now we convert the dB gain into a "binary" gain that we can actually use.

   FIND_GAIN: begin
      if (modified_input_gain > 20) begin
         multiplication_factor <= 13'sd2560;
      end

      else if (modified_input_gain < -20) begin
         multiplication_factor <= 13'sd2560;
      end

      else begin
         case (modified_input_gain)
         9'sd20: multiplication_factor <= 13'sd2560;
         9'sd19: multiplication_factor <= 13'sd2281;
         9'sd18: multiplication_factor <= 13'sd2033;
```

```
            9'sd17: multiplication_factor <= 13'sd1812;
            9'sd16: multiplication_factor <= 13'sd1615;
            9'sd15: multiplication_factor <= 13'sd1439;
            9'sd14: multiplication_factor <= 13'sd1283;
            9'sd13: multiplication_factor <= 13'sd1143;
            9'sd12: multiplication_factor <= 13'sd1019;
            9'sd11: multiplication_factor <= 13'sd908;
            9'sd10: multiplication_factor <= 13'sd809;
            9'sd9: multiplication_factor <= 13'sd721;
            9'sd8: multiplication_factor <= 13'sd643;
            9'sd7: multiplication_factor <= 13'sd573;
            9'sd6: multiplication_factor <= 13'sd510;
            9'sd5: multiplication_factor <= 13'sd455;
            9'sd4: multiplication_factor <= 13'sd405;
            9'sd3: multiplication_factor <= 13'sd361;
            9'sd2: multiplication_factor <= 13'sd322;
            9'sd1: multiplication_factor <= 13'sd287;
            9'sd0: multiplication_factor <= 13'sd256;
            -9'sd1: multiplication_factor <= 13'sd228;
            -(9'sd2): multiplication_factor <= 13'sd203;
            -(9'sd3): multiplication_factor <= 13'sd181;
            -(9'sd4): multiplication_factor <= 13'sd161;
            -(9'sd5): multiplication_factor <= 13'sd143;
            -(9'sd6): multiplication_factor <= 13'sd128;
            -(9'sd7): multiplication_factor <= 13'sd114;
            -(9'sd8): multiplication_factor <= 13'sd102;
            -(9'sd9): multiplication_factor <= 13'sd91;
            -(9'sd10): multiplication_factor <= 13'sd81;
            -(9'sd11): multiplication_factor <= 13'sd72;
            -(9'sd12): multiplication_factor <= 13'sd64;
            -(9'sd13): multiplication_factor <= 13'sd57;
            -(9'sd14): multiplication_factor <= 13'sd51;
            -(9'sd15): multiplication_factor <= 13'sd46;
            -(9'sd16): multiplication_factor <= 13'sd41;
            -(9'sd17): multiplication_factor <= 13'sd36;
            -(9'sd18): multiplication_factor <= 13'sd32;
            -(9'sd19): multiplication_factor <= 13'sd29;
            -9'sd20: multiplication_factor <= 13'sd26;
            default: multiplication_factor <= 13'sd256;
            endcase
        end

        state <= APPLY_GAIN;

    end

    // One clock cycle for multiplies (because they can take a lot of time, especially
with
    // 12-bit by 13-bit multiplies.
    APPLY_GAIN: begin
```

```
            scaled_sample <= incoming_sample * multiplication_factor;
            state <= DIVIDE_AND_OUTPUT;
         end


         // Divide by 128.
         DIVIDE_AND_OUTPUT: begin
            compressed_sample <= scaled_sample[19:8];
            state <= IDLE;
            done <= 1'b1;
         end

         endcase
      end
endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// AC97 Module (taken from lab5a)
//
//////////////////////////////////////////////////////////////////////////////////
// assemble/disassemble AC97 serial frames
module ac97 (ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

  output ready;
  input [7:0] command_address;
  input [15:0] command_data;
  input command_valid;
  input [19:0] left_data, right_data;
  input left_valid, right_valid;
  output [19:0] left_in_data, right_in_data;

  input ac97_sdata_in;
  input ac97_bit_clock;
  output ac97_sdata_out;
  output ac97_synch;

  reg ready;

  reg ac97_sdata_out;
  reg ac97_synch;

  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;
reg [19:0] left_in_data, right_in_data;

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
```

```
//////////////////////////////////////////////////////////////////////////////////
//
// ac97commands Module (taken from lab5a)
//
//////////////////////////////////////////////////////////////////////////////////
// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                     command_valid, volume, source);

  input clock;
  input ready;
  output [7:0] command_address;
  output [15:0] command_data;
  output command_valid;
  input [4:0] volume;
  input [2:0] source;

  reg [23:0] command;
  reg command_valid;

  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
////////////////////////////////////////////////////////////////////////////////
//
// generate PCM data for 750hz sine wave (assuming f(ready) = 48khz)
//
////////////////////////////////////////////////////////////////////////////////

module tone750hz (clock, ready, pcm_data);
  input clock;
  input ready;
  output [19:0] pcm_data;

  reg [8:0] index;
  reg [19:0] pcm_data;

  initial begin
     // synthesis attribute init of old_ready is "0";
     index <= 8'h00;
     // synthesis attribute init of index is "00";
     pcm_data <= 20'h00000;
     // synthesis attribute init of pcm_data is "00000";
  end

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

  // one cycle of a sinewave in 64 20-bit samples
  always @(index) begin
     case (index[5:0])
       6'h00: pcm_data <= 20'h00000;
       6'h01: pcm_data <= 20'h0C8BD;
       6'h02: pcm_data <= 20'h18F8B;
       6'h03: pcm_data <= 20'h25280;
```

```
6'h04: pcm_data <= 20'h30FBC;
6'h05: pcm_data <= 20'h3C56B;
6'h06: pcm_data <= 20'h471CE;
6'h07: pcm_data <= 20'h5133C;
6'h08: pcm_data <= 20'h5A827;
6'h09: pcm_data <= 20'h62F20;
6'h0A: pcm_data <= 20'h6A6D9;
6'h0B: pcm_data <= 20'h70E2C;
6'h0C: pcm_data <= 20'h7641A;
6'h0D: pcm_data <= 20'h7A7D0;
6'h0E: pcm_data <= 20'h7D8A5;
6'h0F: pcm_data <= 20'h7F623;
6'h10: pcm_data <= 20'h7FFFF;
6'h11: pcm_data <= 20'h7F623;
6'h12: pcm_data <= 20'h7D8A5;
6'h13: pcm_data <= 20'h7A7D0;
6'h14: pcm_data <= 20'h7641A;
6'h15: pcm_data <= 20'h70E2C;
6'h16: pcm_data <= 20'h6A6D9;
6'h17: pcm_data <= 20'h62F20;
6'h18: pcm_data <= 20'h5A827;
6'h19: pcm_data <= 20'h5133C;
6'h1A: pcm_data <= 20'h471CE;
6'h1B: pcm_data <= 20'h3C56B;
6'h1C: pcm_data <= 20'h30FBC;
6'h1D: pcm_data <= 20'h25280;
6'h1E: pcm_data <= 20'h18F8B;
6'h1F: pcm_data <= 20'h0C8BD;
6'h20: pcm_data <= 20'h00000;
6'h21: pcm_data <= 20'hF3743;
6'h22: pcm_data <= 20'hE7075;
6'h23: pcm_data <= 20'hDAD80;
6'h24: pcm_data <= 20'hCF044;
6'h25: pcm_data <= 20'hC3A95;
6'h26: pcm_data <= 20'hB8E32;
6'h27: pcm_data <= 20'hAECC4;
6'h28: pcm_data <= 20'hA57D9;
6'h29: pcm_data <= 20'h9D0E0;
6'h2A: pcm_data <= 20'h95927;
6'h2B: pcm_data <= 20'h8F1D4;
6'h2C: pcm_data <= 20'h89BE6;
6'h2D: pcm_data <= 20'h85830;
6'h2E: pcm_data <= 20'h8275B;
6'h2F: pcm_data <= 20'h809DD;
6'h30: pcm_data <= 20'h80000;
6'h31: pcm_data <= 20'h809DD;
6'h32: pcm_data <= 20'h8275B;
6'h33: pcm_data <= 20'h85830;
6'h34: pcm_data <= 20'h89BE6;
6'h35: pcm_data <= 20'h8F1D4;
```

```
      6'h36: pcm_data <= 20'h95927;
      6'h37: pcm_data <= 20'h9D0E0;
      6'h38: pcm_data <= 20'hA57D9;
      6'h39: pcm_data <= 20'hAECC4;
      6'h3A: pcm_data <= 20'hB8E32;
      6'h3B: pcm_data <= 20'hC3A95;
      6'h3C: pcm_data <= 20'hCF044;
      6'h3D: pcm_data <= 20'hDAD80;
      6'h3E: pcm_data <= 20'hE7075;
      6'h3F: pcm_data <= 20'hF3743;
    endcase // case(index[5:0])
  end // always @ (index)
endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// mybram Module (taken from lab5a)
//
//////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps
module mybram #(parameter LOGSIZE=14, WIDTH=1)
              (input wire [LOGSIZE-1:0] addr,
               input wire clk,
               input wire [WIDTH-1:0] din,
               output reg [WIDTH-1:0] dout,
               input wire we);
  // let the tools infer the right number of BRAMs
  (* ram_style = "block" *)
  reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
  always @(posedge clk) begin
    if (we) mem[addr] <= din;
    dout <= mem[addr];
  end
endmodule
```

9.5 Graphics Modules

```
//////////////////////////////////////////////////////////////////////////////////
//
// bcd Module (Written by Gerzain Mata)
//
//////////////////////////////////////////////////////////////////////////////////

module bcd(
```

```
    input [12:0] binary,
    output reg [3:0] thousands,
    output reg [3:0] hundreds,
    output reg [3:0] tens,
    output reg [3:0] ones
    );

         integer i;
         always @(binary)begin
                thousands = 4'd0;
                hundreds = 4'd0;
                tens = 4'd0;
                ones = 4'd0;

                for(i=12; i>=0; i=i-1) begin
                        if (thousands >=5)
                                thousands = thousands + 3;
                        if (hundreds >=5)
                                hundreds = hundreds +3;
                        if(tens >= 5)
                                tens = tens + 3;
                        if(ones >=5)
                                ones = ones + 3;

                        thousands = thousands <<1;
                        thousands[0] = hundreds[3];
                        hundreds = hundreds <<1;
                        hundreds[0] = tens[3];
                        tens = tens << 1;
                        tens[0] = ones[3];
                        ones = ones << 1;
                        ones[0] = binary[i];
                end
         end


endmodule
//////////////////////////////////////////////////////////////////////////////////
//
// digit_blob Module (Written by Gerzain Mata)
//
//////////////////////////////////////////////////////////////////////////////////

module digit_blob
        #(parameter WIDTH = 25, HEIGHT = 52)
        (input pixel_clk,
        input [10:0] x,hcount,
        input [9:0] y,vcount,
        output [10:0] image_addr,
        output reg overlap
```

```
        );

        initial begin
                overlap = 0;
        end

         always@ (posedge pixel_clk) begin
                if ((hcount >= x && hcount < (x+WIDTH)) &&
                        (vcount >= y && vcount < (y+HEIGHT)))
                        overlap <= 1;
                else overlap <= 0;
         end

         // calculate rom address and read the location
         assign image_addr = (hcount-x) + (vcount-y) * WIDTH;

endmodule
//////////////////////////////////////////////////////////////////////////////////
//
// digit_selector Module (Written by Gerzain Mata)
//
//////////////////////////////////////////////////////////////////////////////////

module digit_selector(
   input [10:0] address,
   input [3:0] num_sel,
   input clk,
   output [7:0] red_pixel
   );

         wire [1:0] zeroes, ones, twos, threes, fours, fives, sixes,
                sevens, eights, nines;
         reg [1:0]to_color_map;

         zero_img img_cero(.clka(clk),.addra(address),.douta(zeroes));
         one_img img_uno(.clka(clk),.addra(address),.douta(ones));
         two_img img_dos(.clka(clk),.addra(address),.douta(twos));
         three_img img_tres(.clka(clk),.addra(address),.douta(threes));
         four_img img_cuatro(.clka(clk),.addra(address),.douta(fours));
         five_img img_cinco(.clka(clk),.addra(address),.douta(fives));
         six_img img_seis(.clka(clk),.addra(address),.douta(sixes));
         seven_img img_siete(.clka(clk),.addra(address),.douta(sevens));
         eight_img img_ocho(.clka(clk),.addra(address),.douta(eights));
         nine_img img_nueve(.clka(clk),.addra(address),.douta(nines));

         always @(posedge clk) begin
                case (num_sel)
                        0: to_color_map <= zeroes;
                        1: to_color_map <= ones;
                        2: to_color_map <= twos;
```

```
                        3: to_color_map <= threes;
                        4: to_color_map <= fours;
                        5: to_color_map <= fives;
                        6: to_color_map <= sixes;
                        7: to_color_map <= sevens;
                        8: to_color_map <= eights;
                        9: to_color_map <= nines;
                        default: to_color_map <= 2'b00;
                endcase
        end

        red_hud digit_map(.clka(clk),.addra(to_color_map),.douta(red_pixel));

endmodule
//////////////////////////////////////////////////////////////////////////////
//
// bi-directional mono interface to AC97 (Provided by Gim Hom)
//
//////////////////////////////////////////////////////////////////////////////

module fft_audio (clock_27mhz, reset, volume,
                  audio_in_data, audio_out_data, ready,
                  audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                  ac97_synch, ac97_bit_clock);

  input clock_27mhz;
  input reset;
  input [4:0] volume;
  output [15:0] audio_in_data;
  input [15:0] audio_out_data;
  output ready;

  //ac97 interface signals
  output audio_reset_b;
  output ac97_sdata_out;
  input ac97_sdata_in;
  output ac97_synch;
  input ac97_bit_clock;

  wire [2:0] source;
  assign source = 0;      //mic

  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;

  reg audio_reset_b;
  reg [9:0] reset_count;
```

```
  //wait a little before enabling the AC97 codec
  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(ac97_ready, command_address, command_data, command_valid,
            left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
            right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
            ac97_bit_clock);

  // generate two pulses synchronous with the clock: first capture, then ready
  reg [2:0] ready_sync;
  always @ (posedge clock_27mhz) begin
    ready_sync <= {ready_sync[1:0], ac97_ready};
  end
  assign ready = ready_sync[1] & ~ready_sync[2];

  reg [15:0] out_data;
  always @ (posedge clock_27mhz)
    if (ready) out_data <= audio_out_data;
  assign audio_in_data = left_in_data[19:4];
  assign left_out_data = {out_data, 4'b0000};
  assign right_out_data = left_out_data;

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

//////////////////////////////////////////////////////////////////////////////////
//
// hud_blob Module (Written by Gerzain Mata)
//
//////////////////////////////////////////////////////////////////////////////////

module hud_blob
       #(parameter WIDTH = 512, HEIGHT = 240)
       (input pixel_clk,
   input [10:0] x,hcount,
        input [9:0] y,vcount,
        output reg [23:0] pixel);
```

```
        wire [16:0] image_addr;
        wire [1:0] image_bits;
        wire [7:0] red_mapped;


        always@ (posedge pixel_clk) begin
               if ((hcount >= x && hcount < (x+WIDTH)) &&
                       (vcount >= y && vcount < (y+HEIGHT)))
                       pixel <= {red_mapped, 8'b0, 8'b0};
               else pixel <= 0;
        end

        // calculate rom address and read the location
        assign image_addr = (hcount-x) + (vcount-y) * WIDTH;

        hud_img rom2(.clka(pixel_clk), .addra(image_addr),
               .douta(image_bits));

               // use color map to create 8bits R, 8bits G, 8 bits B;
       red_hud rcm (.clka(pixel_clk), .addra(image_bits), .douta(red_mapped));

endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// hud_digits Module  (Written by Gerzain Mata)
//
//////////////////////////////////////////////////////////////////////////////////

module hud_digits(
   input clk,
        input write,
        input [3:0] num,
        input [3:0] blob,
   input [10:0] hcount,
   input [9:0] vcount,
   output [23:0] pixel
   );
        wire [10:0] image_addr[13:0];
        reg [3:0] number [13:0];
        wire [13:0] overlap;

        genvar i;
        // generate the digit sprites/blobs
        generate
                for ( i = 0; i < 4; i = i+1 ) begin : ripple
                        digit_blob un_blob(.pixel_clk(clk),.x(860+(i*26)),.hcount(hcount),
                                .y(1),.vcount(vcount),.image_addr(image_addr[i]),
```

```
                                .overlap(overlap[i]));
                end
                for ( i = 4; i < 6; i = i+1 ) begin : ripples
                        digit_blob un_blob(.pixel_clk(clk),.x(880+(i-
4)*26),.hcount(hcount),
                                .y(54),.vcount(vcount),.image_addr(image_addr[i]),
                                .overlap(overlap[i]));
                end
                for ( i = 6; i < 8; i = i+1 ) begin : riggle
                        digit_blob un_blob(.pixel_clk(clk),.x(660+(i-
6)*26),.hcount(hcount),
                                .y(110),.vcount(vcount),.image_addr(image_addr[i]),
                                .overlap(overlap[i]));
                end
                for ( i = 8; i < 10; i = i+1 ) begin : riffle
                        digit_blob un_blob(.pixel_clk(clk),.x(750+(i-
8)*26),.hcount(hcount),
                                .y(110),.vcount(vcount),.image_addr(image_addr[i]),
                                .overlap(overlap[i]));
                end
                for ( i = 10; i < 12; i = i+1 ) begin : riggles
                        digit_blob un_blob(.pixel_clk(clk),.x(590+(i-
10)*26),.hcount(hcount),
                                .y(160),.vcount(vcount),.image_addr(image_addr[i]),
                                .overlap(overlap[i]));
                end
                for ( i = 12; i < 14; i = i+1 ) begin : ribbles
                        digit_blob un_blob(.pixel_clk(clk),.x(900+(i-
12)*26),.hcount(hcount),
                                .y(160),.vcount(vcount),.image_addr(image_addr[i]),
                                .overlap(overlap[i]));
                end
        endgenerate

        // write specified digit to sprite for display

        always @(posedge clk) begin
               if(write) begin
                       case(blob)
                               0 : number[0] <= num;
                               1 : number[1] <= num;
                               2 : number[2] <= num;
                               3 : number[3] <= num;
                               4 : number[4] <= num;
                               5 : number[5] <= num;
                               6 : number[6] <= num;
                               7 : number[7] <= num;
                               8 : number[8] <= num;
                               9 : number[9] <= num;
                               10 : number[10] <= num;
```

```
                                11 : number[11] <= num;
                                12 : number[12] <= num;
                                13 : number[13] <= num;
                                default: number[0] <= number[0];
                        endcase
                end
        end

        wire [3:0] selected_sprite;
        reg [10:0] address_out;
        reg [3:0] num_sel_out;

        // selects the sprite to pull the address from depending
        // if the hcount and vcount overlap with the sprite's location
        sprite_img_selector
the_sel(.clk(clk),.sprites(overlap),.selected(selected_sprite));


        always @(posedge clk) begin
                case(selected_sprite)
                                1 : begin address_out <= image_addr[0];
                                        num_sel_out <= number[0]; end
                                2 : begin address_out <= image_addr[1];
                                        num_sel_out <= number[1]; end
                                3 : begin address_out<= image_addr[2];
                                        num_sel_out<= number[2]; end
                                4 : begin address_out<= image_addr[3];
                                        num_sel_out<= number[3]; end
                                5 : begin address_out<= image_addr[4];
                                        num_sel_out<= number[4]; end
                                6 : begin address_out<= image_addr[5];
                                        num_sel_out<= number[5]; end
                                7 : begin address_out<= image_addr[6];
                                        num_sel_out<= number[6]; end
                                8 : begin address_out<= image_addr[7];
                                        num_sel_out <=number[7]; end
                                9 : begin address_out <=image_addr[8];
                                        num_sel_out<= number[8]; end
                                10 : begin address_out<= image_addr[9];
                                        num_sel_out<= number[9]; end
                                11 : begin address_out<= image_addr[10];
                                        num_sel_out<= number[10]; end
                                12 : begin address_out<= image_addr[11];
                                        num_sel_out<= number[11]; end
                                13 : begin address_out <=image_addr[12];
                                        num_sel_out<= number[12]; end
                                14 : begin address_out <=image_addr[13];
                                        num_sel_out <=number[13]; end
                        default : begin address_out<= 0;
                                num_sel_out<= 0; end
```

```
            endcase
        end

        wire [7:0] red_pixel;

        assign pixel = {red_pixel,8'b0, 8'b0};

        // selects which digit to pull from memory to display on the screen
        // then pulls the pixel out from the color map and pushes it out to
        // the VGA bus
        digit_selector the_dig (.address(address_out),.num_sel(num_sel_out),
            .clk(clk),.red_pixel(red_pixel));

endmodule

//////////////////////////////////////////////////////////////////////////////
//
// hud_display: display the statistics on the screen!!! (Written by Gerzain Mata)
//
//////////////////////////////////////////////////////////////////////////////

module hud_display (
  input vclock,         // 65MHz clock
  input reset,          // 1 to initialize module
       input write,
       input [3:0] num,
       input [3:0] blob,
  input [10:0] hcount, // horizontal index of current pixel (0..1023)
  input [9:0]  vcount, // vertical index of current pixel (0..767)
  output [23:0] hud_pixel      // pong game's pixel  // r=23:16, g=15:8, b=7:0
  );

       wire [23:0] hud_img_pixel;

       hud_blob thehud
       (.pixel_clk(vclock),.x(512),.hcount(hcount),.y(0),.vcount(vcount),
       .pixel(hud_img_pixel));

       wire [23:0] digit_pixel;

        hud_digits ma_digs(.clk(vclock),.write(write),.num(num),
            .blob(blob),.hcount(hcount),.vcount(vcount),.pixel(digit_pixel));

       assign hud_pixel = hud_img_pixel | digit_pixel;
endmodule

//////////////////////////////////////////////////////////////////////////////
//
// max_freq_amp Module (Written by Gerzain Mata)
```

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


module max_freq_amp #(parameter WIDTH = 1024)
       ( input [10:0] hcount,
       input [9:0] amplitude,
       output reg [12:0] max_freq,
       output reg [9:0] max_amp
       );

       wire [11:0] freq;
       assign freq = 3 * hcount;

       always @(*) begin
               if(hcount == 0) begin
                       max_freq <= freq;
                       max_amp <= amplitude;
               end else if(amplitude > max_amp && hcount < WIDTH) begin
                       max_freq <= freq;
                       max_amp <= amplitude;
               end
       end

endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// mit_logo_blob Module (Written by Gerzain Mata)
//
//////////////////////////////////////////////////////////////////////////////////

module mit_logo_blob
       #(parameter WIDTH = 110, HEIGHT = 59)
       (input pixel_clk,
       input [10:0] x,hcount,
       input [9:0] y,vcount,
       output reg [23:0] pixel
       );

       wire [12:0] image_addr;
       wire [7:0] red_mapped, green_mapped, blue_mapped;
       wire [1:0] image_bits;


        always@ (posedge pixel_clk) begin
               if ((hcount >= x && hcount < (x+WIDTH)) &&
                       (vcount >= y && vcount < (y+HEIGHT)))
                        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;


        mit_logo_rom rom1(pixel_clk, image_addr, image_bits);
        // use color map to create 8bits R, 8bits G, 8 bits B;
        mit_logo_red rcm (pixel_clk, image_bits, red_mapped);
        mit_logo_green gcm (pixel_clk, image_bits, green_mapped);
        mit_logo_blue bcm (pixel_clk, image_bits, blue_mapped);

endmodule

//////////////////////////////////////////////////
//
// picture_blob: display a picture (Written by Gerzain Mata)
//
//////////////////////////////////////////////////

module picture_blob
       #(parameter WIDTH = 320, HEIGHT = 240)
       (input pixel_clk,
   input [10:0] x,hcount,
        input [9:0] y,vcount,
        output reg [23:0] pixel);

        wire [16:0] image_addr;
        wire [5:0] image_bits;
        wire [7:0] red_mapped, green_mapped, blue_mapped;


        always@ (posedge pixel_clk) begin
            if ((hcount >= x && hcount < (x+WIDTH)) &&
                    (vcount >= y && vcount < (y+HEIGHT)))
                    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;

        image_rom rom1(.clka(pixel_clk), .addra(image_addr),
            .douta(image_bits));

            // use color map to create 8bits R, 8bits G, 8 bits B;
       red_table rcm (.clka(pixel_clk), .addra(image_bits), .douta(red_mapped));
       green_table gcm (.clka(pixel_clk), .addra(image_bits), .douta(green_mapped));
```

```
       blue_table bcm (.clka(pixel_clk), .addra(image_bits), .douta(blue_mapped));

endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// pong_ball: an instance of the pong ball! (Written by Gerzain Mata)
//
//////////////////////////////////////////////////////////////////////////////////

module pong_ball #(parameter SCRN_WIDTH = 1024, SCRN_HEIGHT = 768,
       PUCK_W = 110, PUCK_H = 59, PADDLE_H = 128, PADDLE_W = 16)
(
  input vsync,  // vsync
  input vclock,        // 65MHz clock
  input reset,         // 1 to initialize module
       input up,               // 1 when paddle should move up
  input down,         // 1 when paddle should move down
  input [3:0] pspeed,  // puck speed in pixels/tick
  input [10:0] hcount, // horizontal index of current pixel (0..1023)
  input [9:0] vcount, // vertical index of current pixel (0..767)
  input enabled,
  output [23:0] pixel  // pong game's pixel  // r=23:16, g=15:8, b=7:0
  );



       // wires and instatiations of blobs and alpha blender
       reg [10:0] x_pos;
       reg [9:0] y_pos;
       reg [1:0] direction;
       reg [9:0]  paddle_y;
       wire [23:0] paddle_pixel;


       blob #(.WIDTH(PADDLE_W),.HEIGHT(PADDLE_H),.COLOR(24'hFF_FF_00))   // yellow!
                 paddle(.x(11'd0),.y(paddle_y),.hcount(hcount),.vcount(vcount),
                 .pixel(paddle_pixel));

       // puck blob
       wire [23:0] puck_pixel;
       mit_logo_blob puck(.pixel_clk(vclock),.x(x_pos), .y(y_pos),.hcount(hcount),
                      .vcount(vcount),.pixel(puck_pixel));

       // gotta put them pixels on that screen :)
       assign pixel = enabled ? (puck_pixel | paddle_pixel) : puck_pixel;

       // initial state
       initial begin
```

```
                x_pos = (SCRN_WIDTH - PUCK_W) >> 1;
                y_pos = (SCRN_HEIGHT - PUCK_H) >> 1;
                direction = 2'b00;
                paddle_y <= (SCRN_HEIGHT - PADDLE_H) >> 1;
        end

        always@(negedge vsync) begin
                // conditions on reset
                if(reset) begin
                        x_pos <= (SCRN_WIDTH - PUCK_W) >> 1;
                        y_pos <= (SCRN_HEIGHT - PUCK_H) >> 1;
                        direction <= 2'b00;
                        paddle_y <= (SCRN_HEIGHT - PADDLE_H) >> 1;
                end
                else begin
                        // if puck hits upper bound
                        if(y_pos - pspeed <= 10)begin
                                direction[0] <= 1;
                        end
                        // if it hits lower bound
                        else if ((y_pos + PUCK_H) + pspeed >= SCRN_HEIGHT - 10) begin
                                direction[0] <= 0;
                        end
                        // if it hits left bound
                        if(x_pos - pspeed <= 10)begin
                                if ((paddle_y < y_pos+PUCK_H ) && (paddle_y+PADDLE_H >
y_pos)) begin
                                        direction[1] <= 0;
                                end
                                // otherwise end the game
                                else begin
                                        x_pos <= (SCRN_WIDTH - PUCK_W) >> 1;
                                        y_pos <= (SCRN_HEIGHT - PUCK_H) >> 1;
                                end
                        end
                        // if it hits right bound
                        else if((x_pos + PUCK_W) + pspeed >= SCRN_WIDTH-10) begin
                                direction[1] <= 1;
                        end
                        // change position of puck if the game is running
                        if (enabled) begin
                        case(direction)
                                2'b00: begin x_pos <= x_pos + pspeed; y_pos <= y_pos -
pspeed; end
                                2'b01: begin x_pos <= x_pos + pspeed; y_pos <= y_pos +
pspeed; end
                                2'b10: begin x_pos <= x_pos - pspeed; y_pos <= y_pos -
pspeed; end
                                2'b11: begin x_pos <= x_pos - pspeed; y_pos <= y_pos +
pspeed; end
```

```
                        endcase
                        end

                        // paddle y position
                        if(up && paddle_y > 0) begin
                                paddle_y <= paddle_y - 4;
                        end else if(down && (paddle_y + PADDLE_H) < 767) begin
                                paddle_y <= paddle_y + 4;
                        end

                end
        end

endmodule

//blob module
module blob
  #(parameter WIDTH = 64,            // default width: 64 pixels
              HEIGHT = 64,           // default height: 64 pixels
              COLOR = 24'hFF_FF_FF)  // default color: white
  (input [10:0] x,hcount,
   input [9:0] y,vcount,
   output reg [23:0] pixel);

  always @ * begin
     if ((hcount >= x && hcount < (x+WIDTH)) &&
        (vcount >= y && vcount < (y+HEIGHT)))
       pixel = COLOR;
     else pixel = 0;
  end
endmodule
//////////////////////////////////////////////////////////////////////////////////
//
// process_audio: process incoming audio samples, generate frequency histogram
//  (Written by Gerzain Mata)
//
//////////////////////////////////////////////////////////////////////////////////

module process_audio(clock_27mhz,reset,ready,from_ac97_data,haddr,hdata,hwe);
 input clock_27mhz;
 input reset;
 input ready;
 input [11:0] from_ac97_data;
 output [9:0] haddr,hdata;
 output hwe;

 wire signed [22:0] xk_re,xk_im;
 wire [13:0] xk_index;
 parameter sel = 4'b1000;
```

```
// IP Core Gen -> Digital Signal Processing -> Transforms -> FFTs
// -> Fast Fourier Transform v3.2
// Transform length: 16384
// Implementation options: Pipelined, Streaming I/O
// Transform length options: none
// Input data width: 12
// Phase factor width: 12
// Optional pins: CE
// Scaling options: Unscaled
// Rounding mode: Truncation
// Number of stages using Block Ram: 7
// Output ordering: Bit/Digit Reversed Order
fft16384u fft(.clk(clock_27mhz), .ce(reset | ready),
              .xn_re(from_ac97_data), .xn_im(12'b0),
              .start(1'b1), .fwd_inv(1'b1), .fwd_inv_we(reset),
              .xk_re(xk_re), .xk_im(xk_im), .xk_index(xk_index));

wire signed [13:0] xk_re_scaled = xk_re >> sel;
wire signed [13:0] xk_im_scaled = xk_im >> sel;

// process fft data
reg [2:0] state;
reg [9:0] haddr;
reg [27:0] rere,imim;
reg [27:0] mag2;
reg hwe;
wire sqrt_done;

always @ (posedge clock_27mhz) begin
  hwe <= 0;
  if (reset) begin
    state <= 0;
  end
  else case (state)
   3'h0: if (ready) state <= 1;
   3'h1: begin
           // only process data with index < 1024
           state <= (xk_index[13:10] == 0) ? 2 : 0;
           haddr <= xk_index[9:0];
           rere <= xk_re_scaled * xk_re_scaled;
           imim <= xk_im_scaled * xk_im_scaled;
         end
   3'h2: begin
           state <= 3;
           mag2 <= rere + imim;
         end
   3'h3: if (sqrt_done) begin
           state <= 0;
           hwe <= 1;
```

```
          end
    endcase
  end

  wire [13:0] mag;
  sqrt sqmag(clock_27mhz,mag2,state==2,mag,sqrt_done);
  defparam sqmag.NBITS = 28;

  assign hdata = mag;

endmodule
module sqrt(clk,data,start,answer,done);
  parameter NBITS = 8;  // max 32
  parameter MBITS = (NBITS+1)/2;
  input clk,start;
  input [NBITS-1:0] data;
  output [MBITS-1:0] answer;
  output done;

  reg [MBITS-1:0] answer;
  reg busy;
  reg [4:0] bit;

  wire [MBITS-1:0] trial = answer | (1 << bit);

  always @ (posedge clk) begin
    if (busy) begin
      if (bit == 0) busy <= 0;
      else bit <= bit - 1;
      if (trial*trial <= data) answer <= trial;
    end
    else if (start) begin
      busy <= 1;
      answer <= 0;
      bit <= MBITS - 1;
    end
  end

  assign done = ~busy;
endmodule
//////////////////////////////////////////////////////////////////////////////
//
// sprite_img_selector Module (Written by Gerzain Mata)
//
//////////////////////////////////////////////////////////////////////////////

module sprite_img_selector(
         input clk,
    input [13:0] sprites,
    output reg [3:0] selected
```

```
    );

        always @(posedge clk) begin
                case(sprites)
                        14'b00000000000001 : selected <= 1;
                        14'b00000000000010 : selected <= 2;
                        14'b00000000000100 : selected <= 3;
                        14'b00000000001000 : selected <= 4;
                        14'b00000000010000 : selected <= 5;
                        14'b00000000100000 : selected <= 6;
                        14'b00000001000000 : selected <= 7;
                        14'b00000010000000 : selected <= 8;
                        14'b00000100000000 : selected <= 9;
                        14'b00001000000000 : selected <= 10;
                        14'b00010000000000 : selected <= 11;
                        14'b00100000000000 : selected <= 12;
                        14'b01000000000000 : selected <= 13;
                        14'b10000000000000 : selected <= 14;
                        default: selected <= 0;
                endcase
        end
endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz) (provided in lab code)
//
//////////////////////////////////////////////////////////////////////////////////

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
```