Beep Boop 9000

The basic idea of our project is an audio looper. The GUI allows the user to record up to 8 different tracks of approximately 30 seconds in length. The user can then choose to play any subset of the tracks that have been recorded while either just listening or playing additional sound over the top of the playback. The learning synthesizer will work by listening to a note from an instrument and using Fourier analysis to extract coefficients corresponding to each of the relevant frequencies. The user can save different sounds and then select which instrument to playback each track as. The user can also select the instrument to use for live playback. The full project was not integrated, although the individual pieces were nearly, if not completely developed.

Tyler:

I worked on three main parts of the project: the GUI (graphical user interface), the Memory Controller, and the Master FSM (finite state machine). Originally, I considered my contribution to only be two parts (the GUI and the Memory Controller), but the Master FSM ended up being different enough that I thought it deserved to be described separately. My contribution is located in three main files: *top_level.sv, mem_controller.sv, and GUI.sv*. The contents of these files can be found in Appendix A: Section 1: Parts 1, 2, & 3 respectively. Each part begins with the main module for each piece and is followed by any helper modules I used inside of them.

Master FSM

The Master FSM was the linkage between the subdivisions of the project and the different parts of the BeepBoop9000. It is the top level file where everything actually comes together and has to be realized as one whole project. After beginning to implement this piece, I had to go back and add multiple inputs and outputs to my modules to help with control flow, and in the case of the IFFT module, I had to greatly increase the depth of the fifo buffer.

The inputs to the FSM come from the GUI module and the ADC (Analog to Digital Converter) IP module. Even all hardware switch and button inputs are handled within the GUI and output from there as different signals dependent on the internal state, except for the reset switch (sw[14] in my case). Everything at this level is handled at 100MHz. The audio from the ADC is sampled at 100MHz and piped directly to the input of Brandon's FFT module, which is waiting for a signal to begin taking in this data. The ADC produces a 12-bit value, so I only use the top 8-bits for the FFT, and I convert it to two's complement the same way it is done in Lab 5A. The code used for interacting with the XADC IP module is taken almost directly from Lab 5A. This data does nothing in the FFT until a valid data signal is pulsed concurrently with the addition of new data from the ADC. The GUI also provides input data by taking in button presses and converting them to commands depending on the state of its internal FSM. The ones that concern the Master FSM include *play, record,* and *learn.* These change the state of the Master FSM depending on its current state. There is a basic handshake protocol between the GUI and the Master FSM that records the beginning and end of these changes. Basically, the GUI is a user of the Master FSM object. It calls the functions *play* and *record* (learn is not implemented), and the Master FSM sends a *return* signal when it has completed the operations of those functions. During those functions, the Master FSM sends various control signals to

operate the other modules. In the PLAYING state, the Master FSM must read from the Memory Controller and pipes the input to Matt's inverse FFT module. Both modules use an input and output pulse to signify asynchronously when to take in or output the next packet of data. When the last packet of data is read from the memory controller, the Master FSM receives a *return* signal from the memory controller, and the FSM itself sends a *return* signal to the GUI. It then returns to the WAITING state where it watches for input from the GUI. There is one other piece of code in the WAITING portion of the state switch-case statement. This is used to hold the *return* signal high for more than one clock cycle to ensure that the GUI sees it because the part of the GUI looking for that signal runs at 65MHz. Lastly, the RECORDING state is activated when the FSM receives a *record* command from the GUI while in the WAITING state. The logic for writing is a little more complicated because it must control both the input and output of the FFT along with the input to the Memory Controller. Luckily, we can leave the read flag high for the FFT and the write flag high for the Memory Controller for the duration of the "function" because both work on the asynchronous pulses I mentioned above. The FSM feeds in the audio as I mentioned before. It counts 60 seconds worth of samples, then cuts off data input to the FFT. For the sake of time and debugging, a stop recording signal was not implemented. Instead, a sample counter was used to sample up until 60 seconds worth of samples were recorded, although 80 seconds worth of space was allocated per track. After reading, the FSM just has to wait for an output pulse from the FFT and direct that with the accompanying data to the memory controller. If I was more sure about the timing of each module, I may have been able to take out some of the *if-statements* in this module and allow the modules to talk directly to each other. Finally, after the last sample has been sent to the Memory Controller, there is only one thing left before *returning.* The FSM has to cap the recorded data with a stop packet, which allows us to have variable-length recordings (with a maximum size of course). The Memory

Controller will use this on the other side to tell the FSM when it has finished reading a track. Now, the FSM can *return* and move to the WAITING state. There is one more unused state--GET_COEFFICIENTS. This would have been used to do a one page read from the Memory Controller to retrieve the instrument harmonics data from its stored position. This would be done in tandem with the PLAYING function, so Matt's module has the correct coefficients loaded in before it begins accepting data. However, this was replaced with hardcoded data in a case statement because we did not finish the learning module. Since the Master FSM is also the top level module, it also must deal with the hardware interface that is specified in the constraint file, but there is not too much to do with this.

I believe the RECORDING state of the FSM is a major node requiring further testing. It is one of the last put together and could definitely be the reason why the complete project is not working. There are several possibilities for where the failure is occurring, and the main reasons behind them are either my poor understanding of the ADC module and its output data stream or the FFT module and its input data stream. I spent a lot of time setting up correct debug states for interfacing between Matt's inverse FFT module and my memory controller. It was not easy getting the timing correct and state transitions correct between user-input, memory reading, and iFFT input, so I think expanding into an entire Master FSM would cause a lot more error. I was able to get the debug states to work and even the Master FSM connection with Matt's module to work, but I suspect something is wrong on the other side. I think the connection on the input side is overall more difficult to get correct, and it involves the more unsafe operation of writing to memory.

Memory Controller

I wanted to make the Memory Controller asynchronous so it is easier to connect it to other modules. The Memory Controller consists of three main parts: the FIFO buffer, the SD Controller, and the FSM. The other odds and ends that can be found in the file are the clock divider module used to convert the 100MHz into a 25MHz clock to drive the SD Controller, and the case-statement with lots of *localparams* to partition the memory into pieces and decide which memory address should be used to store or read a certain track or instrument.

FIFO Buffer

The FIFO Buffer is used for both reading and writing. It is 3 bytes wide and has a depth of 1024, and it runs on a 100MHz clock. The width is equal to the size of a single packet for Matt's iFFT. It consists of a frequency and amplitude value. When the Memory Controller is in the READING state, the FIFO Buffer is used to hold outputs. Reading from the SD card must be done in 512 byte pages (which is why the height of the buffer is 1024), but it is not guaranteed that your user will want an enormous 512-byte page. Plus, it would be ridiculous to have a register or bus that big. Instead, the Memory Controller buffers the outputs in the FIFO and waits for a user pulse to the ready-to-read flag before outputting a 3-byte packet. When the Memory Controller is in the WRITING state, the FIFO buffer is used to hold inputs. It is a similar process to reading except transactions occur after a ready-to-send pulse instead. Then, the contents of the FIFO are sent to the SD Controller. In both cases, the Memory Controller must read and write from the FIFO. This time, writing is actually slightly simpler because the FIFO's write-enable flag can be pulsed on the same clock cycle that you give it new data. Then, you can just leave the data on the line until the next pulse. When reading, I had to activate an extra

flag from the IP module to ensure the timing. The FIFO is kind enough to come with a *valid* flag that indicates when new read data is valid. So, the Memory Controller must pulse the read-enable flag then wait for the valid flag to pulse. Finally, it can pipe the new data to the SD Controller.

The FIFO was relatively easy to setup as long as you know the timing for its flags, which you can find in its online manual. The main tradeoff was deciding whether it should be 3 bytes or 1 byte wide. The project handles 3-byte packets in and out of the FFT modules, but the SD Controller only deals with single bytes. Although they were probably equal in difficulty of execution, I decided to use a 3-byte wide FIFO and use a 1-by-3 byte pseudo-buffer to interface between the lines of the FIFO and the SD Controller. I used a separate buffer for reading to and writing from the FIFO (and vice versa for the SD Controller) in order to keep write operations to registers on single clocks (this will be discussed more later). The depth of the FIFO Buffer certainly could have been reduced to 512, but it was an artifact of the 1-byte buffer and we had enough space to not bother changing it.

SD Controller

Thankfully, the staff was kind enough to provide a module that will interact with the SD card for us. That probably saved me a lot of time. Interfacing with this only took a little bit of time with the ILA to confirm I was doing things on the correct clock cycles. Most of the work went into the FSM and making sure it sent those signals when I wanted it too. I will discuss this in the next section, but the SD Controller (and the SD Card) run on a 25 MHz clock. So, I used a clock divider to turn the 100MHz master clock into a 25MHz clock. I then ran all the control signals in a separate 25MHz *always_ff* block.

FSM

The Memory Controller’s FSM is actually split into two FSMs that interact with each other through specific variables. There is one *always_ff* block that runs off the 25Mhz clock and one that runs off the 100MHz input clock. For the most part, variables in the two FSMs are kept completely separate. Specific variables are allowed to be read by both FSMs, but only one may write to it. These are to keep track of state and to pass data. The reason for doing this is to allow the FIFO Buffer to run at 100MHz, while the SD Controller must run at 25MHz. In order to pass data to and from the FIFO, I use two pseudo-buffers (mentioned above). One is written by the FIFO and the other by the SD Controller. The FSM’s states mimic those of the Master FSM with a WAITING, WRITING, & READING state instead of PLAYING and RECORDING. In the WRITING state, the FIFO fills the pseudo-buffer with a new data packet as soon as it is empty. Then, the SD controller tells its FSM whenever it is ready for the next byte, and the FSM can pass it in from the pseudo-buffer. On WRITING, the opposite occurs. The SD Controller slowly fills up the pseudo-buffer with a full data packet. Then, the whole thing is placed into the FIFO to eventually be read out. On WRITING, the FSM quietly finishes when the FIFO buffer is empty and the write line has gone low. On READING, the FSM completes and notifies its partner FSM when the FIFO outputs the stop packet. At this time, the FSM can return to the WAITING state and output a *return* pulse. If a stop packet was accidently not written, the FSM would stop reading when it has reached the maximum length a single track can be.

Probably the most inconvenient part of writing this module was having to do it in real time with an ILA instead of simulating it. The ILA adds a significant amount of compile time and can often be tricky to use if you are looking for signals in an FSM that are not firing when they should be. If a signal never fires, you can not see where things went wrong. So, you have to be

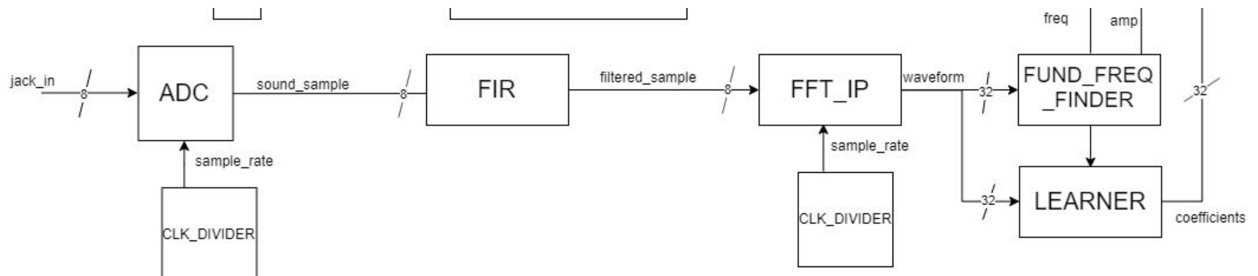
clever when adding your probes and trigger conditions. Setting up the two FSMs to interact with each other also took some extra thinking. I double and triple checked things every time I added a signal to be sure it would work between *always_ff* blocks. I also had to be careful when changing things in case I needed to change both FSMs. Also, I found I needed to reset the FIFO Buffer after a full *read* or *write* operation. I never figured out exactly why, but I shelved the problem instead by just resetting it at the end of an operation.

GUI

The GUI certainly contains the most Verilog I have written for the project. Everything that needs to be displayed adds up pretty quickly. The GUI was just the Status Box for a long time, while I figured out several timing issues and, in general, just how to create a decent GUI. I made sure everything was very modular and variable, so that once I had everything working properly, I could quickly build out the entire GUI from the buildings blocks I had created. I think this saved a lot of time. I spent a lot of time getting the timing of the *hcount* and *vcount* to cooperate. I still feel like there is a better way to do it than I had done it, but it worked well enough for the time I had. Unfortunately, adding one more rectangle for the Learning symbol caused some timing issues during the final presentation, which makes me wonder why that single rectangle was enough to push things over the edge. Displaying characters on screen from my custom coe file was probably the piece of the project that took up the most time for me. Even still, if I am not careful in the way I add components to the GUI, junk will appear on screen at the ends of character strings. I had to read over the timing reports from Vivado many times because they did not make sense to me. It took me at least a couple of days to realize that I needed to debounce all the buttons internally on the 65MHz clock to not have Vivado be upset at me for breaking timing constraints.

The GUI is a multi-layered design that is reflected in the states of the GUI's internal FSM. At the top layer, a user can toggle between the tracks menu and the three action buttons (play, learn, record). In a much further-down-the-road design, you would also be able to toggle to the status bar to pause/unpause, go back, trim, etc. The second tier is purely for the tracks menu because the action buttons do not require any intermediate selections. In this tier, you can toggle between the eight different tracks on screen. The final tier is the action tier. Everything at this level is an action. The current actions available are: *play, learn (*does not actually work), *record, mute track,* and *name track.* Every action or selection the user makes is displayed to the user using changes in color. Tracks and the tracks menu have a highlighted border when they are selected. The actions buttons change color completely. When actions occur, the button for that action changes color until the action is complete. In the case of renaming tracks, the track name will appear and the option to write the name in the status box will disappear. Action buttons will undergo a more drastic color change than when they were selected, and they will return to their original color when the action is complete. The mute button will remain changed until the user toggles it. All controls for the GUI are done with the buttons and switches on the FPGA's board. A stretch goal was to use mouse controls instead, but sadly, I did not have time to implement it. Instead, a user can confirm a selection and move down a tier by pressing the center button. The user can then toggle between options in that tier using the up and down buttons. Finally, the user can move back up a tier by pressing the left or right buttons. Names are written using the first five switches to create a binary number corresponding to the numerical value of the letter they desire (i.e. a=1, b=2, …). The left-most switch is used to toggle case (on for uppercase).

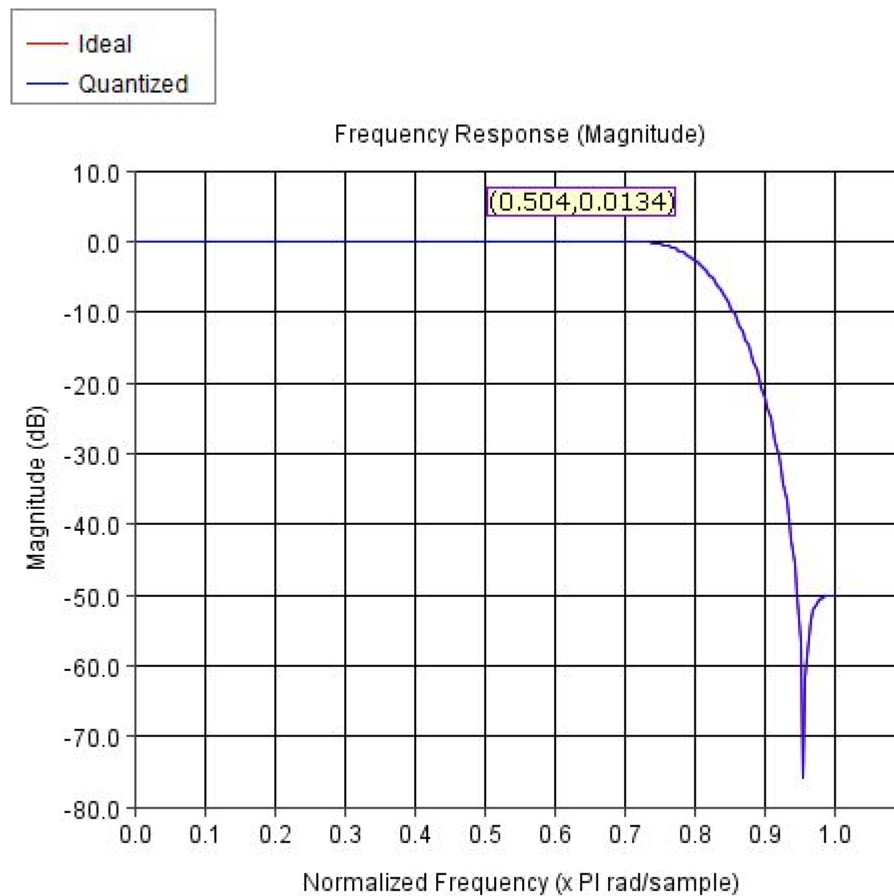
Brandon:



The objective of the input flow is to take an audio input and be able to identify the fundamental frequency of the note being played and being able to identify its harmonic coefficients. It's only data input is the 8-bit stream from the ADC and the output data consists of a 16 bit frequency, representing the bin number of the fft, and an 8 bit amplitude. When in learning mode, the module also drives a 32-bit bus comprised of 4 8-bit magnitudes of the first 4 harmonics. The input flow can be broken down into 4 pieces. The first is the ADC that samples the microphone. This was directly pulled from Lab 5 to save time and effort, since the configuration suited our needs, at 8 bits with down sampling to 48 kHz.

The second piece is the FIR filter. This is included to reduce aliasing in the FFT. I wanted to create a more advanced and flexible filter than what is used in the lab, so I implemented an IP module (FIR Compiler). The concept is quite simple, but the details proved to be numerous. The first began with the frequency response. Vivado's IP configuration tab gives a very useful tab with a graph of the FIR frequency response, which is preloaded with a low pass filter, which is what is required. Unfortunately the default filter did not have the correct cut-off frequency. I first attempted to use an online tool to generate the coefficients, but none would generate an acceptable response. I turned to MatLab which turned out to be quite easy.

Using the following code, I generated 31 coefficients that produced an acceptable frequency response with a cutoff around 20 kHz.

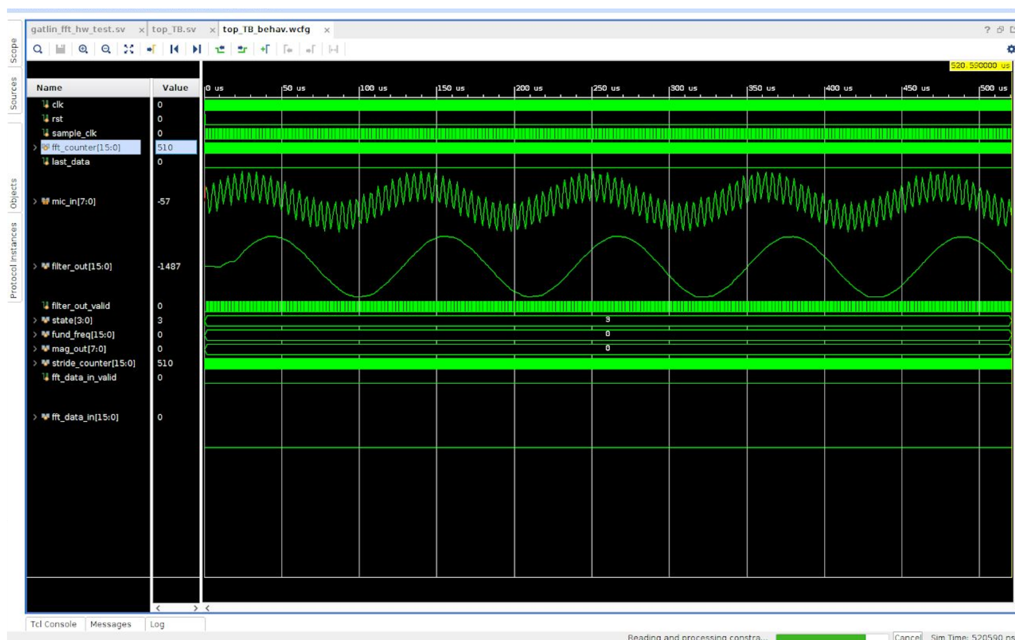



```
% b = fir1(15,.5);

f = [0 .83 .83 1];
m = [1 1 0 0];

b1 = fir2(30,f,m);
```

A higher cutoff frequency was chosen with the concern of attenuating the harmonics when the learner is trying to identify them. The coefficients can be directly copied and pasted into the coefficient vector field on the IP customizer.

I then began testing the filter.I ran a simulation generating a two sine waves, one within the cutoff frequency and one outside, summed them together, and put them through the filter. The output was garbage on the first attempt. It took me a while to figure out that the data_valid

line on the AXI-Stream protocol must be held low only during one clock for each sample. Otherwise, since the FIR is clocked at the system clock, it will read the same sample many times (once every clock), because the sample changes at 48 kHz and the system clock we were using was 100 MHz. The results in the filter progressing its iteration 2083 times on every sample. It created a zeroth order hold on the output even with the linear interpolation of the analog waveform by Vivado. The output did not make me start looking where the problem was, my first thought was messed up coefficients and so I spent a lot of time looking there. After fixing this issue the output looked similar to what is shown below.

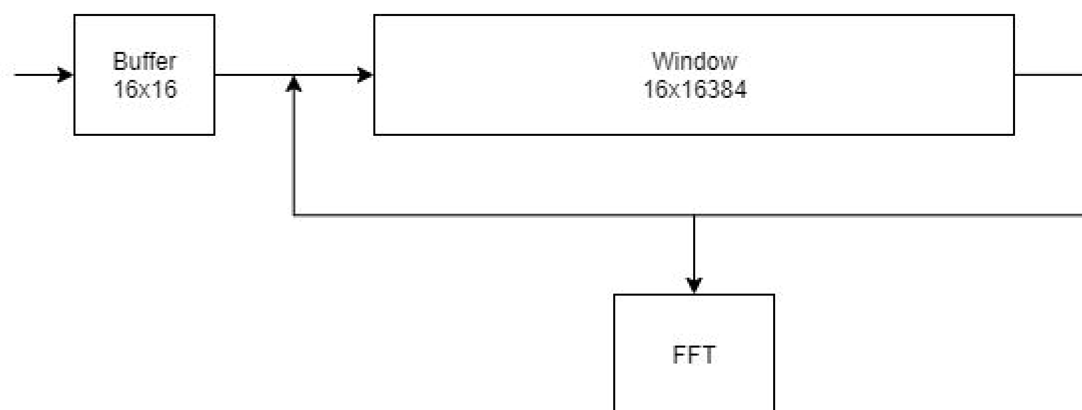


The filter very effectively filters out the higher frequency.

The next piece was the FFT. Again utilizing an IP core from Vivado, the greatest difficulty was getting it to play nice. The documentation to get started is a behemoth, but after familiarizing myself with AXIS protocol, it seemed less daunting.

I initially configured the FFT to synthesize in the pipelined mode, where you can load the next set of sample as it computes the previous windows. The plan was to get this working, then

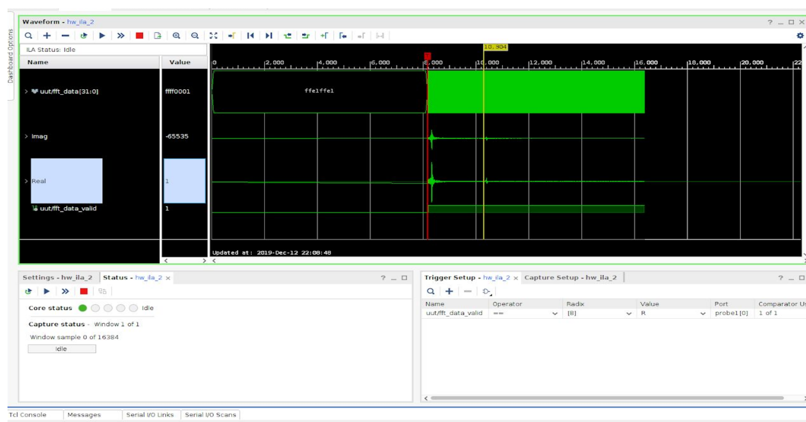
progress to running multiple FFT modules at one time to overlap the windows in order to increase the FFT rate. The first attempt again produced garbage output. After too long, I realized that the pipelined FFT, like the FIR was pulling the samples at the system clock rate. This was a bit confusing at first because you set the sample rate on the config page. I came to realize that the target number is only used in selecting the type of architecture, if left to the compiler. After coming to this realization, I made the data valid for only one clock cycle when a new sample came in. The simulation still showed odd results. The s_ready line would randomly deassert and reassert after the valid line was pulled high for a clock. The event halt flag would go high between valid pulses of the samples, which led me to believe I was on the right track, but it never output the correct data. The only way I could get it to work on pipelined was when the FFT clock was the sample clock. However, the FFT is way too slow when trying to do calculations at such a low rate. Eventually, I decided to switch to the burst mode. This required a new architecture for the FFT module, but I deemed it worth the time, since a more sophisticated version would have to move in that direction anyways. I could no longer stream the samples in at the natural rate. I now had to buffer them. I implemented this architecture with FIFOs.



The samples would come through the Buffer FIFO and would be immediately shifted into the window buffer if it was not performing an FFT. When it begins an FFT, the Window FIFO input

switched from the Buffer to the output of itself. The FIFO can then be completed cycled through, loading the FFT in the process, and returning the Window FIFO to its original state. The Buffer FIFO can hold a small amount of incoming samples while the FFT is being loaded, and then catch back up when the FFT loading is finished. This allows for a very easily configurable transform length (window), and a stride length (number of samples between transforms). With our choice of an FFT frequency of 50 hz, a single FFT can easily perform all of the calculations, which reduced the number of resources required. However, if a much higher rate as chosen, one that a single FFT could not do, more FFT modules could easily be added. The only additional development would be control logic to activate the FFT module at the right times when you are cycling the FIFO.

This implementation went remarkable smoothly. It began working very quickly, and the FFT output could finally be viewed.

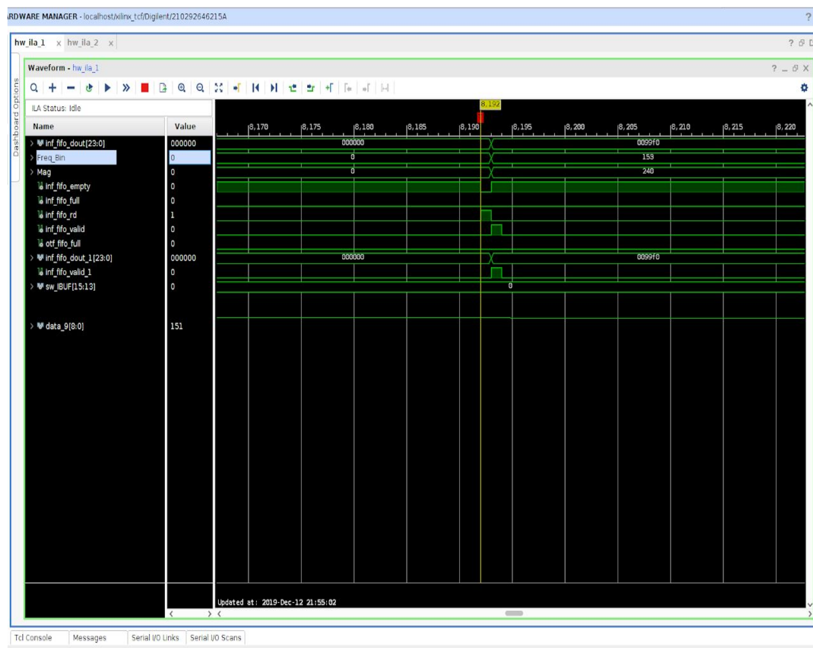


This view is the negative half of a naturally ordered FFT with 1 frequency within the cutoff and one outside (A different cutoff frequency was being used at this time).

The FFT data was passed to the next pieces, the learner and fundamental frequency identifier. The frequency identifier begins watching once the M_AXIS_DATA_VALID flag goes high. It calculates the magnitude given the real and imaginary parts of the transform, and keeps

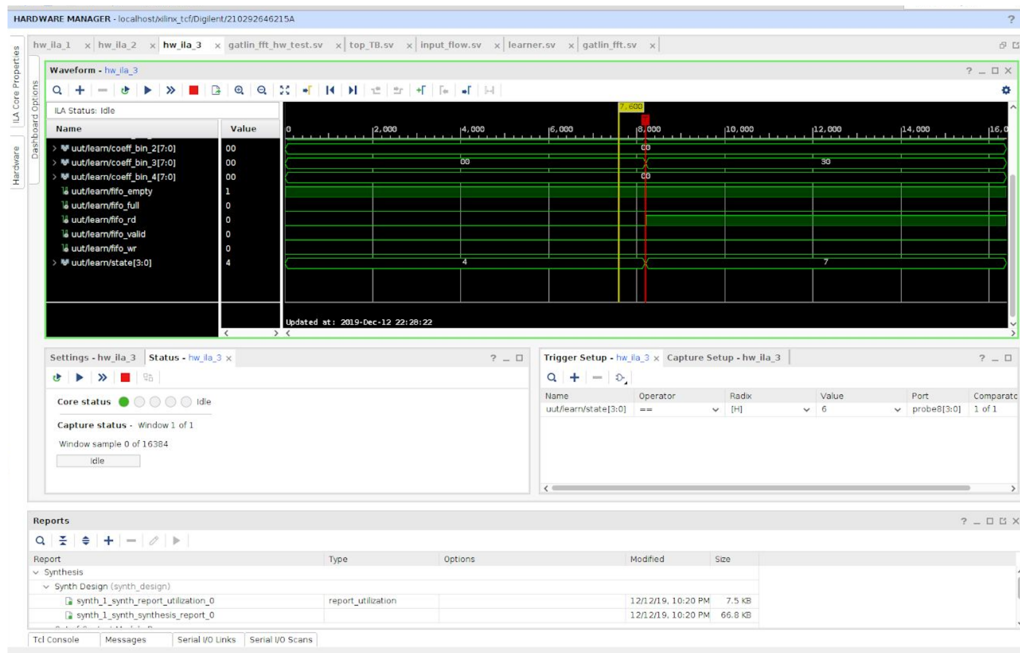
the highest magnitude seen as the bins are piped from the FFT to the magnitude calculator and then to the watcher. The identifier also counts the number of samples to determine the halfway point. Once it reaches half way it pulls its data flag high and holds it high for the second half of the FFT data stream.

The output of the fundamental identifier goes one of two ways, directly to the data management or to the learner module. When the learning input is not asserted, the frequency bin and magnitude are combined on a bus and put into an output FIFO. This is only used to help reduce the timing and piping requirements of the whole system. The exchange is quite standardized, and the sample can be produced and consumed out of phase.



If instead the learning line is asserted, the data is passed to the learner module. The module stores an entire frequency window in the fifo (I am now realizing I only need half). Once the fundamental identifier outputs the correct fundamental, the learner calculates the bin number of the coefficients. The FIFO can then be unloaded, and the magnitude calculated and

saved when the bin number matches the harmonic bin number. The coefficients are combined on the 32-bit output bus and sent to the data management module.



An overarching issue I dealt with was integer math scaling. I have not done much work with this type of computation, so it took a bit longer for me to figure out how the scaling would behave and what needs to be down to send it to the next module.

I think the greatest thing taken away from this project is familiarization with the AXI Streaming protocol. All of Vivado's IP Cores that I have seen use this protocol. Once you are comfortable with it, you can quickly develop really complicated systems utilizing all of the cores. It reduces the developer's job to just some creative use of the cores and logic for many applications.

Matthew:

I worked on three main parts of the project: the GUI (graphical user interface), the Memory Controller, and the Master FSM (finite state machine). Originally, I considered my contribution to only be two parts (the GUI and the Memory Controller), but the Master FSM ended up being different enough that I thought it deserved to be described separately. My contribution is located in three main files: *top_level.sv, mem_controller.sv, and GUI.sv*. The contents of these files can be found in Appendix A: Section 1: Parts 1, 2, & 3 respectively. Each part begins with the main module for each piece and is followed by any helper modules I used inside of them.

Overview / Motivation

I was primarily responsible for the IFFT and audio output side of the project. The goal of this portion of the design was to reconstruct an audio waveform by taking as an input data packets that correspond to a fundamental note value and amplitude, and then using an inverse Fourier transform to reproduce the desired waveform for the given sample. The IFFT takes as an input a set of frequency bin amplitudes, and while one of these was simply equal to the fundamental frequency of the sample, we had to generate the other non-zero frequency bin values by using a set of coefficients that represent a desired instrument sound in order to reproduce the first four harmonic frequency amplitudes. In this way, we were able to generate different sounds that represented several different instruments such as piano, horn, and clarinet. However, this was not a perfect approach, as the harmonic response of an instrument often changes depending on the volume of the note. Nevertheless, it served well as a rough

approximation and allowed us to generate a range of sounds and modify the sound manually to more closely approximate a desired instrument.

I was able to build an IFFT module that did just this, taking in a note value, volume, and instrument coefficients, and creating an appropriate output waveform. This should be enough to handle both real-time synthesis and synthesis from memory in our implementation. We were able to demonstrate the necessary data flow for synthesis from memory by manually writing data packets to the SD card and then playing the resulting transformed waveform out through headphones. This is exactly the operation that is required to play any more complicated arrangement of notes and volumes. We merely had difficulty integrating the output of the forward transform with the memory block. While we should have ideally tried to integrate these modules earlier, we spent probably 20 hours attempting to debug the connection of these two independently functioning modules and were just not able to find the missing link. We suspect that in less frantic debugging conditions, we would have been able to take a more methodical approach and link up these modules sufficiently.

After the IFFT, the audio output approach was just the same as in lab 5a, where we passed the output values through volume control and then PWM to the output.

Technical Details:

After getting a basic forward and inverse transform working, we experimented with a few of the different parameters in order to attain a reasonable frequency resolution while also keeping our FFT sample rate relatively high in order to sufficiently capture the dynamics of the sound. We settled on an FFT/IFFT size of 16384 samples mainly because this seemed like it would give a reasonable frequency resolution, at least for higher frequencies. Since our audio sample rate

was 48 kHz, this would give us a bin size of (48 kHz)/16384 samples, which means that each bin would represent about 2.93 Hz. Moreover, we wanted to ensure that our output would adequately capture the dynamics of the input sound. Thus, we settled on a 50 Hz FFT/IFFT sample rate. Accordingly, I would feed the IFFT a new data packet at 50 Hz, and take 960 samples from the resulting waveform so that over the course of the full second we would get 48000 output samples, giving us back the desired 48 kHz output frequency.

Lessons Learned / Advice

I spent a lot of time up front just reading documentation regarding the FFT/IFFT IP core, and I felt like this was mostly a fruitless use of my time. However, I had a hard time finding any relevant tutorials online, so I'm not sure if there was another better way to get a handle on all of the core's inputs and outputs aside from directly reading the documentation. I think it would be awesome to have a much more concise and colloquial documentation for the core's inputs and outputs, as a lot of the information in the documentation seemed to be rather tangential or a bit more confusing than it needed to be after I had actually figured things out.

I think this is still an important lesson, though, as it's very easy to look at the FFT/IFFT core as more of a black box and assume that you just put your inputs in and get out something that's easy to visualize and process as you might in a higher-level implementation of a Fourier transform.

Appendix A

Section 1

Part 1

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

module top_level(
    input clk_100mhz,
    input [15:0] sw,
    input btnc, btnr, btnl, btnu, btnd,
    input sd_dat_0,
    input vauxp3,
    input vauxn3,
    output [2:0] sd_dat,
    output sd_reset,
    output sd_sck,
    output sd_cmd,
    output vga_hs,
    output vga_vs,
    output [3:0] vga_r,
    output [3:0] vga_g,
    output [3:0] vga_b,
    output aud_pwm,
    output aud_sd
   );

   assign aud_sd = 1;
```

```
logic reset = sw[14];

logic c_clean;
logic u_clean;
logic d_clean;
logic l_clean;
logic r_clean;

debouncer btnc_clean (
    .rst_in(reset),
    .clk_in(clk_100mhz),
    .noisy_in(btnc),
    .clean_out(c_clean)
);

debouncer btnu_clean (
    .rst_in(reset),
    .clk_in(clk_100mhz),
    .noisy_in(btnu),
    .clean_out(u_clean)
);

debouncer btnd_clean (
    .rst_in(reset),
    .clk_in(clk_100mhz),
    .noisy_in(btnd),
    .clean_out(d_clean)
);

debouncer btnl_clean (
    .rst_in(reset),
    .clk_in(clk_100mhz),
    .noisy_in(btnl),
```

```
    .clean_out(l_clean)
);

debouncer btnr_clean (
    .rst_in(reset),
    .clk_in(clk_100mhz),
    .noisy_in(btnr),
    .clean_out(r_clean)
);

logic begin_playing;
logic begin_recording;
logic begin_learning;
logic finish;
logic finish_slow; // to account for 65mhz clock cycle
logic [2:0] track;

GUI gui (
    .clk_in(clk_100mhz),
    .rst_in(reset),
    .sw_char(sw[4:0]),
    .sw_case(sw[6]),
    .btnc(btnc),
    .btnu(btnu),
    .btnd(btnd),
    .btnl(btnl),
    .btnr(btnr),
    .action_done(finish),
    .hs_out(vga_hs),
    .vs_out(vga_vs),
    .r_out(vga_r),
    .g_out(vga_g),
    .b_out(vga_b),
```

```
    .play(begin_playing),
    .record(begin_recording),
    .learn(begin_learning),
    .current_track(track)
);

//// DEBUG ////
localparam INSTRUMENT_0 = 4'b0;
//// DEBUG ////

localparam MUSIC = 0;
localparam INSTRUMENT = 1;
logic mem_read;
logic mem_write;
logic read_flag;
logic write_flag;
logic read_valid_flag;
logic read_done_flag;
logic data_type;
logic [3:0] instrument;
logic [23:0] mem_data_in;
logic [23:0] mem_data_out;

mem_controller uut (
    .clk_in(clk_100mhz),
    .rst_in(reset),
    .track(track),
    .instrument(INSTRUMENT_0),
    .read(mem_read),
    .write(mem_write),
    .data_type(data_type),
    .ready_to_receive(read_flag),
    .ready_to_send(write_flag),
```

```
        .data_valid(read_valid_flag),
        .read_done(read_done_flag),
        .SD_DAT_0(sd_dat_0),
        .SD_DAT_1_3(sd_dat[2:0]),
        .data_in(mem_data_in),
        .SD_SCK(sd_sck),
        .SD_CMD(sd_cmd),
        .SD_RESET(sd_reset),
        .data_out(mem_data_out)
    );

    localparam WAITING = 0;
    localparam RECORDING = 1;
    localparam PLAYING = 2;
    localparam GET_COEFFICIENTS = 3;
    localparam LEARNING = 4;
    logic [2:0] state;

    logic c_clean_prev;

    logic [11:0] adc_data;
    logic adc_ready;
    logic [15:0] sample_counter;
    logic [31:0] samples;
    logic sample_trigger;
    localparam SAMPLE_COUNT = 2082;
    localparam MAX_SAMPLES = 2_800_000;
    assign sample_trigger = (sample_counter == SAMPLE_COUNT);

    xadc_wiz_0 my_adc ( .dclk_in(clk_100mhz), .daddr_in(8'h13), //read from 0x13 for channel
vaux3
                .vauxn3(vauxn3),.vauxp3(vauxp3),
                .vp_in(1),.vn_in(1),
```

```
                .di_in(16'b0),
                .do_out(adc_data),.drdy_out(adc_ready),
                .den_in(1), .dwe_in(0));

    localparam STOP_BIT = 24'h80_00_00;
    logic fft_read_flag;
    logic fft_valid_in_flag;
    logic fft_valid_out_flag;
    logic fft_valid_coefficients_flag;
    logic fft_empty_flag;
    logic fft_rst_flag;
    logic fft_full_flag;
    logic [7:0] fft_audio_in;
    logic [23:0] fft_data_out;
//    logic [23:0] fft_data_buf;
    logic [31:0] fft_coefficients_out;

//    assign fft_audio_in = {~adc_data[11], adc_data[10:4]};
//    assign fft_valid_in_flag = 1'b1;

    logic ifft_new_data_flag;
    logic [31:0] ifft_coefficients_in;
    logic [23:0] ifft_data_in;
    logic ifft_fifo_full_flag;

    logic [2:0] coefficient_byte_counter;

    input_flow (
        .clk(clk_100mhz),
        .rst(reset),
        .mic_in(fft_audio_in),
        .valid_in(fft_valid_in_flag),
        .fifo_rd(fft_read_flag),
```

```
    .fifo_full(fft_full_flag),
    .fifo_empty(fft_empty_flag),
    .fifo_dout(fft_data_out),
    .fifo_valid(fft_valid_out_flag),
    .coeffs(fft_coefficients_out),
    .coeffs_valid(fft_valid_coefficients_flag)
);

logic pwm_val;
assign aud_pwm = (pwm_val) ? 1'bZ : 1'b0;

output_flow (
    .clk(clk_100mhz),
    .rst(reset),
    .new_packet(ifft_new_data_flag),
    .packet_in(ifft_data_in),
    .coefficients(ifft_coefficients_in),
    .vol_in(4'hF), // Full Volume for now
    .fifo_in_full(ifft_fifo_full_flag),
    .pwm_out(pwm_val)
);

logic [1:0] finish_counter;

always_ff @(posedge clk_100mhz) begin
    if (reset) begin
        c_clean_prev <= 1'b0;


        finish <= 1'b0;
        finish_counter <= 2'b0;


        samples <= 32'b0;
        sample_counter <= 16'b0;
```

```
            state <= WAITING;
            coefficient_byte_counter <= 3'b0;

            mem_read <= 1'b0;
            mem_write <= 1'b0;
            read_flag <= 1'b0;
            write_flag <= 1'b0;

            fft_read_flag <= 1'b0;
            fft_valid_in_flag <= 1'b0;
//          fft_empty_flag <= 1'b0;
            fft_rst_flag <= 1'b0;
//          fft_data_buf <= 24'b0;

            ifft_new_data_flag <= 1'b0;
            ifft_data_in <= 24'b0;

        end else begin
            c_clean_prev <= c_clean;
            case (state)
                WAITING: begin
                    if (finish_counter == 3) begin
                        finish <= 1'b0;
                    end else begin
                        finish_counter <= finish_counter + 1;
                    end

                    case (1'b1)
                        begin_playing: begin
                            mem_read <= 1'b1;
                            read_flag <= 1'b1;
                            // TODO: connect instruments //
```

```
//                          data_type <= INSTRUMENT;
//                          state <= GET_COEFFICIENTS;
                        //// DEBUG ////
                        case (track)
                            0: ifft_coefficients_in <= {8'b0000_0110, 8'b0000_1000, 8'b0010_1000,
8'b0001_0000};
                            1: ifft_coefficients_in <= {32'h0000_0000};
                        endcase
                        state <= PLAYING;
                        //// DEBUG ////
                    end
                    begin_recording: begin
                        fft_rst_flag <= 1'b1;
                        mem_write <= 1'b1;
                        state <= RECORDING;
                    end
                    begin_learning: begin
                    end
                endcase
            end
            RECORDING: begin
                fft_read_flag <= 1'b1;
                if (samples < MAX_SAMPLES) begin
                    if (sample_counter == SAMPLE_COUNT)begin
                        sample_counter <= 16'b0;
                        samples <= samples + 1;
                    end else begin
                        sample_counter <= sample_counter + 16'b1;
                    end
                    if (sample_trigger) begin
                        fft_valid_in_flag <= 1'b1;
                        fft_audio_in <= {~adc_data[11],adc_data[10:4]}; //convert to signed. incoming data
is offset binary
```

```
            //https://en.wikipedia.org/wiki/Offset_binary
        end
    end else begin
        fft_read_flag <= 1'b0;
        if (fft_empty_flag) begin
            write_flag <= 1'b1;
            mem_data_in <= STOP_BIT;
            if (write_flag) begin
                write_flag <= 1'b0;
                mem_write <= 1'b0;
                fft_valid_in_flag <= 1'b0;
                sample_counter <= 16'b0;
                samples <= 32'b0;
                finish <= 1'b1;
                finish_counter <= 2'b0;
                state <= WAITING;
            end
        end
    end

    if (fft_valid_in_flag) begin
        fft_valid_in_flag <= 1'b0;
    end

    fft_rst_flag <= 1'b0;
    if (fft_valid_out_flag) begin
//          fft_data_buf <= fft_data_out;
        mem_data_in <= fft_data_out;
        write_flag <= 1'b1;
    end

    if (write_flag) begin
        write_flag <= 1'b0;
```

```
//                      mem_data_in <= fft_data_buf;
                end
            end
            PLAYING: begin
                mem_read <= 1'b1;
                read_flag <= 1'b1;
                data_type <= MUSIC;

                if (read_valid_flag) begin
                    ifft_new_data_flag <= 1'b1;
                    ifft_data_in <= mem_data_out;
                end else begin
                    ifft_new_data_flag <= 1'b0;
                end

                if (read_done_flag) begin
                    mem_read <= 1'b0;
                    read_flag <= 1'b0;
                    finish <= 1'b1;
                    finish_counter <= 2'b0;
                    state <= WAITING;
                end
            end
            GET_COEFFICIENTS: begin
                if (read_valid_flag) begin
                    coefficient_byte_counter <= coefficient_byte_counter + 1;
                    ifft_coefficients_in <= {ifft_coefficients_in[23:0], mem_data_out};
                end

                if (coefficient_byte_counter == 4) begin
                    mem_read <= 1'b0;
                    read_flag <= 1'b0;
                    coefficient_byte_counter <= 0;
```

```
                    state <= PLAYING;
                end
            end
            LEARNING: begin
            end
        endcase
    end
  end
endmodule
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

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

module debouncer(
        input rst_in,
        input clk_in,
        input noisy_in,
        output logic clean_out
    );

   logic [19:0] count;
   logic new_input;

   always_ff @(posedge clk_in)
     if (rst_in) begin
        new_input <= noisy_in;
        clean_out <= noisy_in;
        count <= 0; end
     else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0; end
     else if (count == 1000000) clean_out <= new_input;
     else count <= count+1;
endmodule
```

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

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

```
module mem_controller #(parameter SAMPLE_SIZE = 24)(
    input clk_in,
    input rst_in,
    input [2:0] track,
    input [3:0] instrument,
    input read,
    input write,
    input ready_to_receive,
    input ready_to_send,
    input SD_DAT_0,
    input [23:0] data_in,
    input data_type,
    output logic data_valid,
    output logic read_done,
    output logic [2:0] SD_DAT_1_3,
    output logic SD_SCK,
    output logic SD_RESET,
    output logic SD_CMD,
    output logic [23:0] data_out
    );

    assign SD_DAT_1_3[0] = 1'b1;
    assign SD_DAT_1_3[1] = 1'b1;

    // State //
    localparam WAITING = 0;
    localparam READING = 1;
    localparam WRITING = 2;
    logic [1:0] state;

    logic [12:0] bytes_seen;
    logic [15:0] bytes_used;
```

```
logic clk_25_MHZ;
logic [1:0] package_size;
logic [1:0] package_size_prev;

man_clk_divider (
    .clk_in(clk_in),
    .rst_in(rst_in),
    .clk_out(clk_25_MHZ)
);

// Data Type //
localparam MUSIC = 0;
localparam INSTRUMENT = 1;

// SD Card //
localparam PAGE_SIZE = 512;

logic sd_read;
logic sd_write;
logic [7:0] sd_byte_in;
logic [7:0] sd_byte_out;
logic sd_ready;
logic sd_byte_in_flag;
logic sd_byte_in_flag_prev;
logic sd_byte_out_flag;
logic sd_byte_out_flag_prev;
logic [31:0] sd_address;
logic [SAMPLE_SIZE-1:0] sd_data_package;

// Data //
localparam TRACK_ADDRESS_START = 32'h0000_2600;
localparam TRACK_LENGTH = 12_288;
localparam TRACK_1 = TRACK_ADDRESS_START;
```

```
    localparam TRACK_2 = TRACK_ADDRESS_START + TRACK_LENGTH;
    localparam TRACK_3 = TRACK_ADDRESS_START + TRACK_LENGTH + TRACK_LENGTH;
    localparam TRACK_4 = TRACK_ADDRESS_START + TRACK_LENGTH + TRACK_LENGTH +
TRACK_LENGTH;
    localparam TRACK_5 = TRACK_ADDRESS_START + TRACK_LENGTH + TRACK_LENGTH +
TRACK_LENGTH + TRACK_LENGTH;
    localparam TRACK_6 = TRACK_ADDRESS_START + TRACK_LENGTH + TRACK_LENGTH +
TRACK_LENGTH + TRACK_LENGTH + TRACK_LENGTH;
    localparam TRACK_7 = TRACK_ADDRESS_START + TRACK_LENGTH + TRACK_LENGTH +
TRACK_LENGTH + TRACK_LENGTH + TRACK_LENGTH + TRACK_LENGTH;
    localparam TRACK_8 = TRACK_ADDRESS_START + TRACK_LENGTH + TRACK_LENGTH +
TRACK_LENGTH + TRACK_LENGTH + TRACK_LENGTH + TRACK_LENGTH + TRACK_LENGTH;

    localparam INSTRUMENT_1 =  32'h0000_0200;
    localparam INSTRUMENT_2 =  32'h0000_0400;
    localparam INSTRUMENT_3 =  32'h0000_0800;
    localparam INSTRUMENT_4 =  32'h0000_0A00;
    localparam INSTRUMENT_5 =  32'h0000_0C00;
    localparam INSTRUMENT_6 =  32'h0000_0E00;
    localparam INSTRUMENT_7 =  32'h0000_1000;
    localparam INSTRUMENT_8 =  32'h0000_1200;
    localparam INSTRUMENT_9 =  32'h0000_1400;
    localparam INSTRUMENT_10 = 32'h0000_1800;
    localparam INSTRUMENT_11 = 32'h0000_1A00;
    localparam INSTRUMENT_12 = 32'h0000_1C00;
    localparam INSTRUMENT_13 = 32'h0000_1E00;
    localparam INSTRUMENT_14 = 32'h0000_2000;
    localparam INSTRUMENT_15 = 32'h0000_2200;
    localparam INSTRUMENT_16 = 32'h0000_2400;

    logic [31:0] data_address;
    always_comb begin
        if (data_type == MUSIC) begin
```

```
        case (track)
            0: data_address = TRACK_1;
            1: data_address = TRACK_2;
            2: data_address = TRACK_3;
            3: data_address = TRACK_4;
            4: data_address = TRACK_5;
            5: data_address = TRACK_6;
            6: data_address = TRACK_7;
            7: data_address = TRACK_8;
        endcase
    end else begin
        case (instrument)
            0:  data_address = INSTRUMENT_1;
            1:  data_address = INSTRUMENT_2;
            2:  data_address = INSTRUMENT_3;
            3:  data_address = INSTRUMENT_4;
            4:  data_address = INSTRUMENT_5;
            5:  data_address = INSTRUMENT_6;
            6:  data_address = INSTRUMENT_7;
            7:  data_address = INSTRUMENT_8;
            8:  data_address = INSTRUMENT_9;
            9:  data_address = INSTRUMENT_10;
            10: data_address = INSTRUMENT_11;
            11: data_address = INSTRUMENT_12;
            12: data_address = INSTRUMENT_13;
            13: data_address = INSTRUMENT_14;
            14: data_address = INSTRUMENT_15;
            15: data_address = INSTRUMENT_16;
        endcase
    end
end

sd_controller (
```

```
    .clk(clk_25_MHZ),
    .reset(rst_in),
    .miso(SD_DAT_0),
    .rd(sd_read),
    .wr(sd_write),
    .din(sd_byte_in),
    .address(sd_address),
    .mosi(SD_CMD),
    .cs(SD_DAT_1_3[2]),
    .sclk(SD_SCK),
    .ready(sd_ready),
    .byte_available(sd_byte_out_flag),
    .ready_for_next_byte(sd_byte_in_flag),
    .dout(sd_byte_out),
    .status()
);

// FIFO //
localparam FIFO_SIZE = 1024;
localparam FIFO_LOG_SIZE = 10;
logic fifo_read;
logic fifo_write;
logic [SAMPLE_SIZE-1:0] fifo_data_in;
logic [SAMPLE_SIZE-1:0] fifo_data_out;
logic fifo_full;
logic fifo_empty;
logic fifo_valid;
logic first_write;
logic [FIFO_LOG_SIZE-1:0] fifo_length;
logic [SAMPLE_SIZE-1:0] fifo_data_package;
logic fifo_rst;
logic fifo_clear;
assign fifo_rst = rst_in | fifo_clear;
```

```
sd_fifo (
    .clk(clk_in),
    .srst(fifo_rst),
    .rd_en(fifo_read),
    .wr_en(fifo_write),
    .din(fifo_data_in),
    .dout(fifo_data_out),
    .full(fifo_full),
    .empty(fifo_empty),
    .data_count(fifo_length),
    .valid(fifo_valid)
);

always_ff @(posedge clk_25_MHZ) begin
    if (rst_in) begin
        bytes_seen <= 0;
        sd_address <= 32'b0;
        sd_read <= 1'b0;
        sd_write <= 1'b0;
        state <= WAITING;
        package_size <= 2'b0;
        sd_data_package <= 24'b0;
    end else begin
        case (state)
            WAITING: begin
                case ({write, read})
                    READING: begin
                        if (sd_ready) begin
                            state <= READING;
                            sd_address <= data_address;
                        end
                    end
```

```
        WRITING: begin
            if (sd_ready) begin
                state <= WRITING;
                sd_address <= data_address;
            end
        end
        default:;
    endcase
end
READING: begin
    if ((fifo_length <= FIFO_SIZE - PAGE_SIZE) &&
          sd_ready) begin
        sd_read <= 1'b1;
    end

    if (sd_byte_out_flag) begin
        bytes_seen <= bytes_seen + 1;
        package_size <= package_size + 1;
        sd_data_package <= {sd_data_package[15:0], sd_byte_out};
        if (bytes_seen == PAGE_SIZE - 1) begin
            sd_address <= sd_address + PAGE_SIZE;
            sd_read <= 1'b0;
        end
    end

    if (package_size == 3) begin
        package_size <= 0;
    end

    if (bytes_used == TRACK_LENGTH || read_done) begin
        sd_read <= 1'b0;
        sd_address <= 0;
        bytes_seen <= 0;
```

```
            package_size <= 0;
            state <= WAITING;
        end
    end
    WRITING: begin
        if (((fifo_length >= PAGE_SIZE) || (!fifo_empty && !write)) &&
              sd_ready) begin
            sd_write <= 1'b1;
        end

        if (sd_byte_in_flag) begin
            bytes_seen <= bytes_seen + 1;
            sd_byte_in <= fifo_data_package[(package_size+1 << 3)-1 -: 8];
            if (bytes_seen == PAGE_SIZE-1) begin
                sd_write <= 1'b0;
                bytes_seen <= 0;
                sd_address <= sd_address + PAGE_SIZE;
            end

            if (package_size == 2) begin
                package_size <= 0;
            end else begin
                package_size <= package_size + 1;
            end
        end

        if (fifo_empty && !write) begin
            state <= WAITING;
            sd_write <= 1'b0;
            package_size <= 0;
            bytes_seen <= 0;
            sd_address <= 0;
        end
```

```
            end
        endcase
    end
end

always_ff @(posedge clk_in) begin
    if (rst_in) begin
        data_out <= 24'b0;
        fifo_read <= 1'b0;
        fifo_write <= 1'b0;
        package_size_prev <= 2'b0;
        fifo_data_in <= 24'b0;
        bytes_used <= 0;
        first_write <= 1'b1;
        fifo_data_package <= 24'b0;
        data_valid <= 1'b0;
    end else begin
        case (state)
            READING: begin
                fifo_write <= 1'b0;
                fifo_read <= 1'b0;
                package_size_prev <= package_size;
                data_valid <= 1'b0;

                if ((package_size == 3) &&
                    (package_size_prev != package_size)) begin

                    fifo_write <= 1'b1;
                    fifo_data_in <= sd_data_package;
                end

                if (ready_to_receive) begin
                    fifo_read <= 1'b1;
```

```
        end

        if (fifo_valid) begin
            data_valid <= 1'b1;
            data_out <= fifo_data_out;
            bytes_used <= bytes_used + 3;
            if (fifo_data_out[SAMPLE_SIZE-1]) begin
                read_done <= 1'b1;
                fifo_clear <= 1'b1;
                data_out <= 0;
                data_valid <= 1'b0;
            end
        end
    end
    WRITING: begin
        fifo_read <= 1'b0;
        fifo_write <= 1'b0;
        package_size_prev <= package_size;

        if (((package_size == 0) && (package_size_prev != package_size)) ||
             (fifo_length >= 3 && first_write)) begin

            fifo_read <= 1'b1;
            if (first_write) begin
                first_write <= 1'b0;
            end
        end

        if (fifo_valid) begin
            fifo_data_package <= fifo_data_out;
        end

        if (ready_to_send) begin
```

```
                fifo_write <= 1'b1;
                fifo_data_in <= data_in;
            end
        end
        WAITING: begin
            first_write <= 1'b1;
            read_done <= 1'b0;
            fifo_clear <= 1'b0;
        end
      endcase
    end
  end
endmodule
//////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////

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

module man_clk_divider #(parameter DENOMINATOR = 4, parameter COUNTER_LENGTH = 2)(
  input clk_in,
  input rst_in,
  output logic clk_out
  );


  logic [COUNTER_LENGTH-1:0] counter;
  always_ff @(posedge clk_in) begin
    if (rst_in) begin
      counter <= 8'b0;
      clk_out <= 1'b0;
    end else begin
      if (counter == DENOMINATOR - 1) begin
        counter <= 0;
        clk_out <= 1'b1;
```

```
            end else begin
                counter <= counter + 1;
                clk_out <= 1'b0;
            end
        end
    end
endmodule

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

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

module sd_controller(
    output reg cs, // Connect to SD_DAT[3].
    output mosi, // Connect to SD_CMD.
    input miso, // Connect to SD_DAT[0].
    output sclk, // Connect to SD_SCK.
                // For SPI mode, SD_DAT[2] and SD_DAT[1] should be held HIGH.
                // SD_RESET should be held LOW.

    input rd,   // Read-enable. When [ready] is HIGH, asseting [rd] will
                // begin a 512-byte READ operation at [address].
                // [byte_available] will transition HIGH as a new byte has been
                // read from the SD card. The byte is presented on [dout].
    output reg [7:0] dout, // Data output for READ operation.
    output reg byte_available, // A new byte has been presented on [dout].

    input wr,   // Write-enable. When [ready] is HIGH, asserting [wr] will
                // begin a 512-byte WRITE operation at [address].
                // [ready_for_next_byte] will transition HIGH to request that
                // the next byte to be written should be presentaed on [din].
    input [7:0] din, // Data input for WRITE operation.
```

```
    output reg ready_for_next_byte, // A new byte should be presented on [din].

    input reset, // Resets controller on assertion.
    output ready, // HIGH if the SD card is ready for a read or write operation.
    input [31:0] address,   // Memory address for read/write operation. This MUST
                            // be a multiple of 512 bytes, due to SD sectoring.
    input clk,  // 25 MHz clock.
    output [4:0] status // For debug purposes: Current state of controller.
);

    parameter RST = 0;
    parameter INIT = 1;
    parameter CMD0 = 2;
    parameter CMD55 = 3;
    parameter CMD41 = 4;
    parameter POLL_CMD = 5;

    parameter IDLE = 6;
    parameter READ_BLOCK = 7;
    parameter READ_BLOCK_WAIT = 8;
    parameter READ_BLOCK_DATA = 9;
    parameter READ_BLOCK_CRC = 10;
    parameter SEND_CMD = 11;
    parameter RECEIVE_BYTE_WAIT = 12;
    parameter RECEIVE_BYTE = 13;
    parameter WRITE_BLOCK_CMD = 14;
    parameter WRITE_BLOCK_INIT = 15;
    parameter WRITE_BLOCK_DATA = 16;
    parameter WRITE_BLOCK_BYTE = 17;
    parameter WRITE_BLOCK_WAIT = 18;

    parameter WRITE_DATA_SIZE = 515;
```

```
reg [4:0] state = RST;
assign status = state;
reg [4:0] return_state;
reg sclk_sig = 0;
reg [55:0] cmd_out;
reg [7:0] recv_data;
reg cmd_mode = 1;
reg [7:0] data_sig = 8'hFF;

reg [9:0] byte_counter;
reg [9:0] bit_counter;

reg [26:0] boot_counter = 27'd100_000_000;
always @(posedge clk) begin
    if(reset == 1) begin
        state <= RST;
        sclk_sig <= 0;
        boot_counter <= 27'd100_000_000;
    end
    else begin
        case(state)
            RST: begin
                if(boot_counter == 0) begin
                    sclk_sig <= 0;
                    cmd_out <= {56{1'b1}};
                    byte_counter <= 0;
                    byte_available <= 0;
                    ready_for_next_byte <= 0;
                    cmd_mode <= 1;
                    bit_counter <= 160;
                    cs <= 1;
                    state <= INIT;
                end
```

```
    else begin
        boot_counter <= boot_counter - 1;
    end
end
INIT: begin
    if(bit_counter == 0) begin
        cs <= 0;
        state <= CMD0;
    end
    else begin
        bit_counter <= bit_counter - 1;
        sclk_sig <= ~sclk_sig;
    end
end
CMD0: begin
    cmd_out <= 56'hFF_40_00_00_00_00_95;
    bit_counter <= 55;
    return_state <= CMD55;
    state <= SEND_CMD;
end
CMD55: begin
    cmd_out <= 56'hFF_77_00_00_00_00_01;
    bit_counter <= 55;
    return_state <= CMD41;
    state <= SEND_CMD;
end
CMD41: begin
    cmd_out <= 56'hFF_69_00_00_00_00_01;
    bit_counter <= 55;
    return_state <= POLL_CMD;
    state <= SEND_CMD;
end
POLL_CMD: begin
```

```
    if(recv_data[0] == 0) begin
        state <= IDLE;
    end
    else begin
        state <= CMD55;
    end
end
IDLE: begin
    if(rd == 1) begin
        state <= READ_BLOCK;
    end
    else if(wr == 1) begin
        state <= WRITE_BLOCK_CMD;
    end
    else begin
        state <= IDLE;
    end
end
READ_BLOCK: begin
    cmd_out <= {16'hFF_51, address, 8'hFF};
    bit_counter <= 55;
    return_state <= READ_BLOCK_WAIT;
    state <= SEND_CMD;
end
READ_BLOCK_WAIT: begin
    if(sclk_sig == 1 && miso == 0) begin
        byte_counter <= 511;
        bit_counter <= 7;
        return_state <= READ_BLOCK_DATA;
        state <= RECEIVE_BYTE;
    end
    sclk_sig <= ~sclk_sig;
end
```

```
READ_BLOCK_DATA: begin
    dout <= recv_data;
    byte_available <= 1;
    if (byte_counter == 0) begin
        bit_counter <= 7;
        return_state <= READ_BLOCK_CRC;
        state <= RECEIVE_BYTE;
    end
    else begin
        byte_counter <= byte_counter - 1;
        return_state <= READ_BLOCK_DATA;
        bit_counter <= 7;
        state <= RECEIVE_BYTE;
    end
end
READ_BLOCK_CRC: begin
    bit_counter <= 7;
    return_state <= IDLE;
    state <= RECEIVE_BYTE;
end
SEND_CMD: begin
    if (sclk_sig == 1) begin
        if (bit_counter == 0) begin
            state <= RECEIVE_BYTE_WAIT;
        end
        else begin
            bit_counter <= bit_counter - 1;
            cmd_out <= {cmd_out[54:0], 1'b1};
        end
    end
    sclk_sig <= ~sclk_sig;
end
RECEIVE_BYTE_WAIT: begin
```

```
    if (sclk_sig == 1) begin
        if (miso == 0) begin
            recv_data <= 0;
            bit_counter <= 6;
            state <= RECEIVE_BYTE;
        end
    end
    sclk_sig <= ~sclk_sig;
end
RECEIVE_BYTE: begin
    byte_available <= 0;
    if (sclk_sig == 1) begin
        recv_data <= {recv_data[6:0], miso};
        if (bit_counter == 0) begin
            state <= return_state;
        end
        else begin
            bit_counter <= bit_counter - 1;
        end
    end
    sclk_sig <= ~sclk_sig;
end
WRITE_BLOCK_CMD: begin
    cmd_out <= {16'hFF_58, address, 8'hFF};
    bit_counter <= 55;
    return_state <= WRITE_BLOCK_INIT;
    state <= SEND_CMD;
            ready_for_next_byte <= 1;
end
WRITE_BLOCK_INIT: begin
    cmd_mode <= 0;
    byte_counter <= WRITE_DATA_SIZE;
    state <= WRITE_BLOCK_DATA;
```

```
    ready_for_next_byte <= 0;
end
WRITE_BLOCK_DATA: begin
    if (byte_counter == 0) begin
        state <= RECEIVE_BYTE_WAIT;
        return_state <= WRITE_BLOCK_WAIT;
    end
    else begin
        if ((byte_counter == 2) || (byte_counter == 1)) begin
            data_sig <= 8'hFF;
        end
        else if (byte_counter == WRITE_DATA_SIZE) begin
            data_sig <= 8'hFE;
        end
        else begin
            data_sig <= din;
            ready_for_next_byte <= 1;
        end
        bit_counter <= 7;
        state <= WRITE_BLOCK_BYTE;
        byte_counter <= byte_counter - 1;
    end
end
WRITE_BLOCK_BYTE: begin
    if (sclk_sig == 1) begin
        if (bit_counter == 0) begin
            state <= WRITE_BLOCK_DATA;
            ready_for_next_byte <= 0;
        end
        else begin
            data_sig <= {data_sig[6:0], 1'b1};
            bit_counter <= bit_counter - 1;
        end;
```

```
            end;
            sclk_sig <= ~sclk_sig;
        end
        WRITE_BLOCK_WAIT: begin
            if (sclk_sig == 1) begin
                if (miso == 1) begin
                    state <= IDLE;
                    cmd_mode <= 1;
                end
            end
            sclk_sig = ~sclk_sig;
        end
      endcase
    end
  end

  assign sclk = sclk_sig;
  assign mosi = cmd_mode ? cmd_out[55] : data_sig[7];
  assign ready = (state == IDLE);
endmodule

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

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

module GUI (
   input clk_in,
   input rst_in,
   input [4:0] sw_char,
   input sw_case,
   input btnc, btnu, btnd, btnl, btnr,
   input action_done,
```

```
 output hs_out,
 output vs_out,
 output logic [3:0] r_out,
 output logic [3:0] b_out,
 output logic [3:0] g_out,
 output logic play,
 output logic record,
 output logic learn,
 output logic [2:0] current_track
);

logic clk_65mhz;

// create 65mhz system clock, happens to match 1024 x 768 XVGA timing
clk_divider clk_div_65(.clk_in1(clk_in), .clk_out1(clk_65mhz));

logic [4:0] sw_char_buf;
logic sw_case_buf;
logic btnc_clean, btnl_clean, btnr_clean, btnu_clean, btnd_clean;

debouncer sw_0_clean (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(sw_char[0]),
    .clean_out(sw_char_buf[0])
);

debouncer sw_1_clean (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(sw_char[1]),
    .clean_out(sw_char_buf[1])
);
```

```
debouncer sw_2_clean (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(sw_char[2]),
    .clean_out(sw_char_buf[2])
);

debouncer sw_3_clean (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(sw_char[3]),
    .clean_out(sw_char_buf[3])
);

debouncer sw_4_clean (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(sw_char[4]),
    .clean_out(sw_char_buf[4])
);

debouncer sw_case_clean (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(sw_case),
    .clean_out(sw_case_buf)
);

debouncer btnc_clean_deb (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(btnc),
```

```
    .clean_out(btnc_clean)
);

debouncer btnl_clean_deb (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(btnl),
    .clean_out(btnl_clean)
);

debouncer btnr_clean_deb (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(btnr),
    .clean_out(btnr_clean)
);

debouncer btnu_clean_deb (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(btnu),
    .clean_out(btnu_clean)
);

debouncer btnd_clean_deb (
    .rst_in(rst_in),
    .clk_in(clk_65mhz),
    .noisy_in(btnd),
    .clean_out(btnd_clean)
);

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

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

logic [11:0] status_pixel;
logic box_status;
logic [47:0] track_name;

status_box (
    .clk_in(clk_65mhz),
    .rst_in(rst_in),
    .hcount_in(hcount),
    .vcount_in(vcount),
    .status(box_status),
    .char_val(sw_char_buf),
    .char_case(sw_case_buf),
    .enter(btnc_clean),
    .back(btnl_clean),
    .seconds(),
    .pixel_out(status_pixel),
    .name_out(track_name)
);

logic [11:0] tracks_pixel;
logic [7:0] track_select;
logic tracks_display_selected;
logic mute_track;
logic name_track;
logic mute_focused;
logic name_focused;

tracks (
```

```
    .clk_in(clk_65mhz),
    .rst_in(rst_in),
    .track_select(track_select),
    .track_name(track_name),
    .mute_track(mute_track),
    .name_track(name_track),
    .mute_focused(mute_focused),
    .name_focused(name_focused),
    .focused(tracks_display_selected),
    .hcount_in(hcount),
    .vcount_in(vcount),
    .pixel_out(tracks_pixel)
);

localparam PLAY_BUTTON_X = 760;
localparam PLAY_BUTTON_Y = 100;
logic play_button_select;
logic start_play;
logic [11:0] play_button_pixel;

play_button #(.WIDTH(64), .HEIGHT(64))(
    .clk_in(clk_65mhz),
    .rst_in(rst_in),
    .hcount_in(hcount),
    .vcount_in(vcount),
    .x_in(PLAY_BUTTON_X),
    .y_in(PLAY_BUTTON_Y),
    .play(start_play),
    .selected(play_button_select),
    .pixel_out(play_button_pixel)
);

localparam RECORD_BUTTON_X = 792;
```

```
    localparam RECORD_BUTTON_Y = 400;
    logic record_button_select;
    logic start_record;
    logic [11:0] record_button_pixel;

    record_button (
        .clk_in(clk_65mhz),
        .rst_in(rst_in),
        .hcount_in(hcount),
        .vcount_in(vcount),
        .x_in(RECORD_BUTTON_X),
        .y_in(RECORD_BUTTON_Y),
        .selected(record_button_select),
        .record(start_record),
        .pixel_out(record_button_pixel)
    );

    localparam LEARN_COLOR = 12'hF0F;
    localparam LEARN_FOCUS_COLOR = 12'hE24;
    localparam LEARN_SELECTED_COLOR = 12'hF1D;
    logic [11:0] learn_pixel;
    logic [11:0] learn_color;
    logic learn_select;
    logic learn_focus;
    assign learn_pixel = 12'b0;
//    assign learn_color = (learn_select) ? LEARN_SELECTED_COLOR : (learn_focus) ?
LEARN_FOCUS_COLOR : LEARN_COLOR;

//    draw_rectangle #(.WIDTH(64), .HEIGHT(64))
//    learn_background (
//        .clk_in(clk_in),
//        .rst_in(rst_in),
//        .x_outer_in(760),
```

```
//        .y_outer_in(550),
//        .x_inner_in(1024),
//        .y_inner_in(1204),
//        .hcount_in(hcount),
//        .vcount_in(vcount),
//        .color(learn_color),
//        .pixel_out(learn_pixel)
//    );

    assign rgb = tracks_pixel | status_pixel | play_button_pixel | record_button_pixel | learn_pixel;

    localparam buf_cycles = 3;
    logic b, hs, vs;

    synchronize #(.NSYNC(buf_cycles), .BUS_WIDTH(0)) (
        .clk_in(clk_65mhz),
        .in(vsync),
        .out(vs)
    );

    synchronize #(.NSYNC(buf_cycles), .BUS_WIDTH(0)) (
        .clk_in(clk_65mhz),
        .in(hsync),
        .out(hs)
    );

    synchronize #(.NSYNC(buf_cycles), .BUS_WIDTH(0)) (
        .clk_in(clk_65mhz),
        .in(blank),
        .out(b)
    );

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

```
    assign r_out = ~b ? rgb[11:8]: 0;
    assign g_out = ~b ? rgb[7:4] : 0;
    assign b_out = ~b ? rgb[3:0] : 0;

    assign hs_out = ~hs;
    assign vs_out = ~vs;

    localparam TOP = 0;
    localparam TRACKS = 1;
    localparam TRACK = 2;
    localparam NAME = 3;
    localparam ACTION = 4;
//    localparam PLAY = 4;
//    localparam RECORD = 5;
//    localparam LEARN = 6;

    logic [2:0] state;
    logic [2:0] presses;
    logic btnc_clean_prev;
    logic btnl_clean_prev;
    logic btnr_clean_prev;
    logic btnu_clean_prev;
    logic btnd_clean_prev;

    always_ff @(posedge clk_65mhz) begin
        if (rst_in) begin
            presses <= 2'b0;
            state <= 3'b0;
            tracks_display_selected <= 1'b0;
            play_button_select <= 1'b0;
            record_button_select <= 1'b0;
            learn_select <= 1'b0;
            btnc_clean_prev <= 1'b0;
```

```
        btnl_clean_prev <= 1'b0;
        btnr_clean_prev <= 1'b0;
        btnu_clean_prev <= 1'b0;
        btnd_clean_prev <= 1'b0;
    end else begin
        btnc_clean_prev <= btnc_clean;
        btnl_clean_prev <= btnl_clean;
        btnr_clean_prev <= btnr_clean;
        btnu_clean_prev <= btnu_clean;
        btnd_clean_prev <= btnd_clean;

        case (state)
            TOP: begin
                if (btnl_clean && !btnl_clean_prev) begin
                    presses <= (presses == 3) ? 2'b0 : presses + 1;
                end else if (btnr_clean && !btnr_clean_prev) begin
                    presses <= (presses == 0) ? 2'b10 : presses - 1;
                end
                start_record <= 1'b0;
                start_play <= 1'b0;
                learn_select <= 1'b0;

                case (presses)
                    0: begin
                        tracks_display_selected <= 1'b1;
                        play_button_select <= 1'b0;
                        record_button_select <= 1'b0;
                        learn_focus <= 1'b0;
                    end
                    1: begin
                        tracks_display_selected <= 1'b0;
                        play_button_select <= 1'b1;
                        record_button_select <= 1'b0;
```

```
                learn_focus <= 1'b0;
            end
            2: begin
                tracks_display_selected <= 1'b0;
                play_button_select <= 1'b0;
                record_button_select <= 1'b1;
                learn_focus <= 1'b0;
            end
            3: begin
                tracks_display_selected <= 1'b0;
                play_button_select <= 1'b0;
                record_button_select <= 1'b1;
                learn_focus <= 1'b1;
            end
        endcase

        if (btnc_clean && !btnc_clean_prev) begin
            case (presses)
                0: begin
                    tracks_display_selected <= 1'b0;
                    state <= TRACKS;
                    presses <= 3'b0;
                end
                1: begin
                    state <= ACTION;
//                      presses <= 0;
                    play <= 1'b1;
                    start_play <= 1'b1;
                end
                2: begin
                    state <= ACTION;
//                      presses <= 0;
                    record <= 1'b1;
```

```
                start_record <= 1'b1;
            end
            3: begin
                state <= ACTION;
//                  presses <= 0;
                learn <= 1'b1;
                learn_select <= 1'b1;
            end
        endcase
    end
end
TRACKS: begin
    case (presses)
        0: track_select <= 8'b0000_0001;
        1: track_select <= 8'b0000_0010;
        2: track_select <= 8'b0000_0100;
        3: track_select <= 8'b0000_1000;
        4: track_select <= 8'b0001_0000;
        5: track_select <= 8'b0010_0000;
        6: track_select <= 8'b0100_0000;
        7: track_select <= 8'b1000_0000;
    endcase

    current_track <= presses;

    if ((btnl_clean && !btnl_clean_prev) || (btnr_clean && !btnr_clean_prev)) begin
        presses <= 3'b0;
        tracks_display_selected <= 1'b1;
        track_select <= 8'b0;
        state <= TOP;
    end else if (btnu_clean && !btnu_clean_prev) begin
        presses <= presses - 1;
    end else if (btnd_clean && !btnd_clean_prev) begin
```

```
        presses <= presses + 1;
    end else if (btnc_clean && !btnc_clean_prev) begin
        presses <= 3'b0;
        state <= TRACK;
    end
end
TRACK: begin
    if ((btnl_clean && !btnl_clean_prev) || (btnr_clean && !btnr_clean_prev)) begin
        presses <= 3'b0;
        mute_focused <= 1'b0;
        name_focused <= 1'b0;
        state <= TRACKS;
    end else if (btnu_clean && !btnu_clean_prev) begin
        presses <= (presses) ? 3'b000 : 3'b001;
    end else if (btnd_clean && !btnd_clean_prev) begin
        presses <= (presses) ? 3'b000 : 3'b001;
    end else if (btnc_clean && !btnc_clean_prev) begin
        if (presses) begin
            mute_track <= 1'b1;
        end else begin
            box_status <= 1'b1;
            presses <= 3'b0;
            mute_focused <= 1'b0;
            name_focused <= 1'b0;
            state <= NAME;
        end
    end else begin
        mute_track <= 1'b0;
        name_track <= 1'b0;
    end

    if (presses) begin
        mute_focused <= 1'b1;
```

```
                    name_focused <= 1'b0;
                end else begin
                    mute_focused <= 1'b0;
                    name_focused <= 1'b1;
                end
            end
            NAME: begin
                if (btnc_clean && !btnc_clean_prev) begin
                    presses <= presses + 1;
                    if (presses == 7) begin
                        name_track <= 1'b1;
                        presses <= 3'b0;
                        box_status <= 1'b0;
                        state <= TRACK;
                    end
                end
            end
            ACTION: begin
                play <= 1'b0;
                learn <= 1'b0;
                record <= 1'b0;
                play_button_select <= 1'b0;
                record_button_select <= 1'b0;
                learn_focus <= 1'b0;
                if (action_done) begin
                    state <= TOP;
                end
            end
        endcase
    end
end

endmodule
```

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

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

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

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

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

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

  // horizontal: 1344 pixels total
  // display 1024 pixels per line
  reg hblank,vblank;
  wire hsyncon,hsyncoff,hreset,hblankon;
  assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
  assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
  assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));  // 1183
  assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP - 1));
//1343
```

```
   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));  // 771
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE - 1));  //
777
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE + V_BP -
1)); // 805

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always_ff @(posedge vclock_in) begin
//      if (rst_in) begin
//         hblank <= 0;
//         vblank <= 0;
//         hsync_out <= 0;
//         vsync_out <= 0;
//         blank_out <= 0;
//      end else begin
         hcount_out <= hreset ? 0 : hcount_out + 1;
         hblank <= next_hblank;
         hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

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

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

```
  end

endmodule

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

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

module status_box(
   input clk_in,
   input rst_in,
   input [1:0] status,
   input [4:0] char_val,
   input char_case,
   input selected,
   input enter, back,
   input [9:0] seconds,
   input [10:0] hcount_in,
   input [9:0] vcount_in,
   output logic [11:0] pixel_out,
   output logic [47:0] name_out
   );

   // Pixel Routing vars
   localparam PIXEL_INS = 3;
   localparam PIXEL_SIZE = 12;
   logic [PIXEL_SIZE-1:0] name_pixel_buf;
   logic [(PIXEL_INS * PIXEL_SIZE)-1:0] name_pixels;
   logic [PIXEL_SIZE-1:0] blank_pixel_buf;

   // Box //
   parameter BOX_WIDTH = 256;
```

```
    parameter BOX_HEIGHT = 64;
    parameter X_TOP_LEFT = 384;
    parameter Y_TOP_LEFT = 704;
    localparam BOX_BORDER_THICKNESS = 8;
    logic [PIXEL_SIZE-1:0] box_border_pixels;
    logic [PIXEL_SIZE-1:0] box_pixels;
    assign name_pixels[(PIXEL_SIZE*2)-1 -:  PIXEL_SIZE] = box_pixels;
    assign name_pixels[(PIXEL_SIZE*3)-1 -:  PIXEL_SIZE] = box_border_pixels;
    assign blank_pixel_buf = box_border_pixels | box_pixels;

    // STATE //
//    logic [1:0] state;
    localparam BLANK_STATE = 0;
    localparam NAME_STATE = 1;

    // "Enter Your Name" vars
    localparam enter_name_len = 18;
    logic [5:0] Enter_Your_Name [enter_name_len-1:0];

    // Name String vars
    localparam NAME_LEN = 8;
    localparam UNDERSCORE = 6'b11_1100;
    logic [5:0] Name [7:0];

    // Cursor vars
    logic [2:0] name_index;
    logic enter_prev;
    logic back_prev;
    logic name_done;

    always_ff @(posedge clk_in) begin
        if (rst_in) begin
```

```
        // Name String //
        Name[0] <= {char_case, char_val};
        Name[1] <= UNDERSCORE;
        Name[2] <= UNDERSCORE;
        Name[3] <= UNDERSCORE;
        Name[4] <= UNDERSCORE;
        Name[5] <= UNDERSCORE;
        Name[6] <= UNDERSCORE;
        Name[7] <= UNDERSCORE;
        // Name String END //

        // Pixel Routing //
        name_pixel_buf <= 12'b0;
//      blank_pixel_buf <= 12'b0;
        // Pixel Routing END //

        // Cursor Control //
        name_index <= 3'b0;
        enter_prev <= 1'b0;
        back_prev <= 1'b0;
        name_done <= 1'b0;
        // Cursor Control END //

    end else begin
        enter_prev <= enter;
        back_prev <= back;
        case (status)
            BLANK_STATE: begin
                name_done <= 1'b0;
                pixel_out <= blank_pixel_buf;
                // Name String //
                Name[0] <= {char_case, char_val};
                Name[1] <= UNDERSCORE;
```

```
        Name[2] <= UNDERSCORE;
        Name[3] <= UNDERSCORE;
        Name[4] <= UNDERSCORE;
        Name[5] <= UNDERSCORE;
        Name[6] <= UNDERSCORE;
        Name[7] <= UNDERSCORE;
        // Name String END //
    end
    NAME_STATE: begin
        // Pixel Routing ///
        for (integer i = 1; i <= PIXEL_INS; i = i + 1) begin
            name_pixel_buf = name_pixel_buf | name_pixels[(PIXEL_SIZE*i)-1 -: 12];
        end

        pixel_out <= name_pixel_buf;
        name_pixel_buf <= 12'b0;
        // Pixel Routing END //

        // Name String //
        Name[name_index] <= {char_case, char_val};
        name_out <= {Name[7], Name[6], Name[5], Name[4], Name[3], Name[2], Name[1],
Name[0]};
        // Name String END //

        // Cursor Control //
        if (enter && !enter_prev) begin
            name_index <= name_index + 1;
//              if (name_index == 3'b111) begin
//                  name_done <= 1'b1;
//              end
        end else if (back && !back_prev) begin
            name_index <= (name_index > 0) ? name_index - 1 : 3'b0; // Can't go below 0
            Name[name_index] <= UNDERSCORE;
```

```
                end
                // Cursor Control END //
            end
        endcase
    end
end

initial begin
    Enter_Your_Name[0] =  6'b01_1011; // _
    Enter_Your_Name[1] =  6'b10_0101; // E
    Enter_Your_Name[2] =  6'b00_1110; // n
    Enter_Your_Name[3] =  6'b01_0100; // t
    Enter_Your_Name[4] =  6'b00_0101; // e
    Enter_Your_Name[5] =  6'b01_0010; // r
    Enter_Your_Name[6] =  6'b01_1011; // _
    Enter_Your_Name[7] =  6'b11_1001; // Y
    Enter_Your_Name[8] =  6'b00_1111; // o
    Enter_Your_Name[9] =  6'b01_0101; // u
    Enter_Your_Name[10] = 6'b01_0010; // r
    Enter_Your_Name[11] = 6'b01_1011; // _
    Enter_Your_Name[12] = 6'b10_1110; // N
    Enter_Your_Name[13] = 6'b00_0001; // a
    Enter_Your_Name[14] = 6'b00_1101; // m
    Enter_Your_Name[15] = 6'b00_0101; // e
    Enter_Your_Name[16] = 6'b11_1011; // :
    Enter_Your_Name[17] = 6'b01_1011; // _
end

string_display #(.STRING_LEN(enter_name_len + NAME_LEN)) enter_name (
    .clk_in(clk_in),
    .rst_in(rst_in),
    .char_array({Name, Enter_Your_Name}),
    .hcount_in(hcount_in),
```

```
    .vcount_in(vcount_in),
    .x_in((X_TOP_LEFT + (BOX_WIDTH >> 1)) - (((enter_name_len + NAME_LEN) << 3) >> 1) - 3),
    .y_in((Y_TOP_LEFT + (BOX_HEIGHT >> 1)) - 4),
    .background_color(12'h00F),
    .pixel_out(name_pixels[((PIXEL_SIZE)-1) -: PIXEL_SIZE])
);

draw_rectangle #(.WIDTH(BOX_WIDTH), .HEIGHT(BOX_HEIGHT)) box (
    .clk_in(clk_in),
    .rst_in(rst_in),
    .x_outer_in(X_TOP_LEFT),
    .y_outer_in(Y_TOP_LEFT),
    .x_inner_in(1024),
    .y_inner_in(1024),
    .hcount_in(hcount_in),
    .vcount_in(vcount_in),
    .color(12'h00F),
    .pixel_out(box_pixels)
);

draw_rectangle #(.WIDTH(BOX_WIDTH + (BOX_BORDER_THICKNESS << 1)),
            .HEIGHT(BOX_HEIGHT + (BOX_BORDER_THICKNESS << 1)))
box_border (
    .clk_in(clk_in),
    .rst_in(rst_in),
    .x_outer_in(X_TOP_LEFT - BOX_BORDER_THICKNESS),
    .y_outer_in(Y_TOP_LEFT - BOX_BORDER_THICKNESS),
    .x_inner_in(X_TOP_LEFT),
    .y_inner_in(Y_TOP_LEFT),
    .hcount_in(hcount_in),
    .vcount_in(vcount_in),
    .color(12'hFFF),
    .pixel_out(box_border_pixels)
```

```
);

endmodule

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

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

module string_display #(parameter STRING_LEN = 8,
                        parameter FONT_COLOR = 12'hFFF) (
    input clk_in,
    input rst_in,
    input [5:0] char_array [STRING_LEN-1:0],
    input [10:0] hcount_in,
    input [9:0] vcount_in,
    input [11:0] background_color,
    input [10:0] x_in,
    input [9:0] y_in,
    output logic [11:0] pixel_out
    );

    localparam HEIGHT = 8;
    localparam WIDTH = 8;
    localparam LOG_WIDTH = 3;
    localparam CHAR_LEN = 6;
    localparam NEWLINE = 27;

    logic [10:0] x_current;
    logic [10:0] x_cursor = 11'b0;
    logic [9:0] y_current;
    logic [9:0] y_cursor = 10'b0;
    logic [5:0] char_value;
```

```
    logic [5:0] char_current;
    logic [11:0] pixel_buf;

    char_block #(.FONT_COLOR(FONT_COLOR))
    char_disp (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .x_in(x_current),
        .y_in(y_current),
        .hcount(hcount_in),
        .vcount(vcount_in),
        .char_val(char_value),
        .background_color(background_color),
        .pixel_out(pixel_buf)
    );

    always_comb begin
        char_current = (hcount_in - x_in) >> LOG_WIDTH;
        char_value = char_array[char_current];
        if (hcount_in-3 >= x_in &&
                hcount_in-3 < (x_in + (STRING_LEN << LOG_WIDTH)) &&
                vcount_in >= y_in &&
                vcount_in < (y_in + HEIGHT)) begin

            x_current = x_in + (((hcount_in - x_in) >> 3) << LOG_WIDTH);
            y_current = y_in; // For now, y_current does not change
            pixel_out = pixel_buf;
        end else begin
            pixel_out = 12'b0;
        end
    end

endmodule
```

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

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

module char_block #(parameter FONT_COLOR = 12'hFFF)(
    input clk_in,
    input rst_in,
    input [10:0] x_in,
    input [9:0] y_in,
    input [10:0] hcount,
    input [9:0] vcount,
    input [5:0] char_val,
    input [11:0] background_color,
    output logic [11:0] pixel_out
   );

   localparam HEIGHT = 8; // chars are 8x8
   localparam WIDTH = 8;
   localparam ROM_WIDTH = 2;

   logic [11:0] pixel_addr; // 3841 lines in the coe file
   logic [ROM_WIDTH-1:0] color; // Currently only two colors
   // calculate rom address and read the location
   assign pixel_addr = ((hcount - x_in) + ((vcount - y_in) * WIDTH)) + (char_val * HEIGHT * WIDTH);
   char_rom letter (.clka(clk_in), .addra(pixel_addr), .douta(color));

   always @(posedge clk_in) begin
      if (rst_in) begin
         pixel_out <= 0;
      end else begin
         if (hcount >= x_in) begin
```

```
            case (color)
                2'b01: pixel_out <= background_color;
                2'b10: pixel_out <= FONT_COLOR;
                2'b11: pixel_out <= 12'b0;
                2'b00: pixel_out <= 12'b0;
            endcase
        end else begin
            pixel_out <= 12'b0;
        end
    end
end
endmodule

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

///////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////
module draw_rectangle #(parameter WIDTH = 256, HEIGHT = 128) (
    input clk_in,
    input rst_in,
    input [10:0] x_outer_in,
    input [9:0] y_outer_in,
    input [10:0] x_inner_in,
    input [9:0] y_inner_in,
    input [10:0] hcount_in,
    input [9:0] vcount_in,
    input [11:0] color,
    output logic [11:0] pixel_out
    );

    logic [10:0] x_thickness;
    logic [9:0] y_thickness;
    assign x_thickness = x_inner_in - x_outer_in;
```

```
    assign y_thickness = y_inner_in - y_outer_in;

    always_ff @(posedge clk_in) begin
        if (rst_in) begin
            pixel_out <= 12'h000;
        end else if (((hcount_in >= x_outer_in) &&
                    (hcount_in < (x_outer_in + WIDTH)) &&
                    (vcount_in >= y_outer_in) &&
                    (vcount_in < (y_outer_in + HEIGHT))) &&
                     !(((hcount_in >= x_inner_in) &&
                        (hcount_in < x_outer_in + WIDTH - x_thickness)) &&
                       ((vcount_in >= y_inner_in) &&
                        (vcount_in < y_outer_in + HEIGHT - y_thickness)))
                   ) begin
            pixel_out <= color;
        end else pixel_out <= 12'h000;
    end

endmodule

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

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

module tracks #(parameter NUM_TRACKS = 8, parameter LOG_NUM_TRACKS = 3)(
        input clk_in,
        input rst_in,
        input [7:0] track_select,
        input [47:0] track_name,
        input mute_track,
        input name_track,
        input mute_focused,
```

```
        input name_focused,
        input focused,
        input [10:0] hcount_in,
        input [9:0] vcount_in,
        output logic [11:0] pixel_out
    );

    //// TRACKS ////
    localparam TRACK_WIDTH = 338;
    localparam TRACK_HEIGHT = 78;
    localparam TRACK_COLOR = 12'h0F0;
    localparam TRACKS_START_X = 75;
    localparam TRACKS_START_Y = 60;
    localparam BORDER_SIZE = 4;

    logic [11:0] track_pixels [8:0];
    logic [47:0] track_names [7:0];
    logic [7:0] track_muted;
    logic [7:0] focus_mute;
    logic [7:0] focus_name;
    logic [7:0] track_named;

//    always_comb begin
//        for (integer i = 0; i < 8; i = i + 1) begin
//            if (track_select == i) begin
//                track_selected[i] = 1'b1;
//            end else begin
//                track_selected[i] = 1'b0;
//            end

//            if (i == 0) begin
//                pixel_out = track_pixels[i];
//            end else begin
```

```
//                pixel_out |= track_pixels[i];
//            end
//        end
//    end

    always_ff @(posedge clk_in) begin
        if (rst_in) begin
        end else begin
            for (integer i = 0; i < 8; i = i + 1) begin
                if (track_select[i]) begin
                    track_names[i] <= track_name;
                    track_muted[i] <= mute_track;
                    track_named[i] <= name_track;
                    focus_mute[i] <= mute_focused;
                    focus_name[i] <= name_focused;
                end else begin
                    track_names[i] <= 48'b0;
                    track_muted[i] <= 1'b0;
                    track_named[i] <= 1'b0;
                    focus_mute[i] <= 1'b0;
                    focus_name[i] <= 1'b0;
                end

                if (i == 0) begin
                    pixel_out = track_pixels[i];
                end else begin
                    pixel_out |= track_pixels[i];
                end
            end
            pixel_out |= track_pixels[8];
        end
    end
```

```
    localparam test_name =
48'b00_1000_00_0111_00_0110_00_0101_00_0100_00_0011_00_0010_00_0001;

    track track_1 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .selected(track_select[0]),
        .track_name(track_names[0]),
//        .track_name(test_name),
        .mute_track(track_muted[0]),
        .name_track(track_named[0]),
        .mute_focused(focus_mute[0]),
        .name_focused(focus_name[0]),
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
        .x_in(TRACKS_START_X),
        .y_in(TRACKS_START_Y),
        .pixel_out(track_pixels[0])
    );

    track track_2 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .selected(track_select[1]),
        .track_name(track_names[1]),
//        .track_name(test_name),
        .mute_track(track_muted[1]),
        .name_track(track_named[1]),
        .mute_focused(focus_mute[1]),
        .name_focused(focus_name[1]),
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
        .x_in(TRACKS_START_X),
```

```
        .y_in(TRACKS_START_Y + TRACK_HEIGHT),
        .pixel_out(track_pixels[1])
    );

    track track_3 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .selected(track_select[2]),
        .track_name(track_names[2]),
//        .track_name(test_name),
        .mute_track(track_muted[2]),
        .name_track(track_named[2]),
        .mute_focused(focus_mute[2]),
        .name_focused(focus_name[2]),
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
        .x_in(TRACKS_START_X),
        .y_in(TRACKS_START_Y + (TRACK_HEIGHT*2)),
        .pixel_out(track_pixels[2])
    );

    track track_4 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .selected(track_select[3]),
        .track_name(track_names[3]),
//        .track_name(test_name),
        .mute_track(track_muted[3]),
        .name_track(track_named[3]),
        .mute_focused(focus_mute[3]),
        .name_focused(focus_name[3]),
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
```

```
        .x_in(TRACKS_START_X),
        .y_in(TRACKS_START_Y + (TRACK_HEIGHT*3)),
        .pixel_out(track_pixels[3])
    );

    track track_5 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .selected(track_select[4]),
        .track_name(track_names[4]),
//      .track_name(test_name),
        .mute_track(track_muted[4]),
        .name_track(track_named[4]),
        .mute_focused(focus_mute[4]),
        .name_focused(focus_name[4]),
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
        .x_in(TRACKS_START_X),
        .y_in(TRACKS_START_Y + (TRACK_HEIGHT*4)),
        .pixel_out(track_pixels[4])
    );

    track track_6 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .selected(track_select[5]),
        .track_name(track_names[5]),
//      .track_name(test_name),
        .mute_track(track_muted[5]),
        .name_track(track_named[5]),
        .mute_focused(focus_mute[5]),
        .name_focused(focus_name[5]),
        .hcount_in(hcount_in),
```

```
        .vcount_in(vcount_in),
        .x_in(TRACKS_START_X),
        .y_in(TRACKS_START_Y + (TRACK_HEIGHT*5)),
        .pixel_out(track_pixels[5])
    );

    track track_7 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .selected(track_select[6]),
        .track_name(track_names[6]),
//        .track_name(test_name),
        .mute_track(track_muted[6]),
        .name_track(track_named[6]),
        .mute_focused(focus_mute[6]),
        .name_focused(focus_name[6]),
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
        .x_in(TRACKS_START_X),
        .y_in(TRACKS_START_Y + (TRACK_HEIGHT*6)),
        .pixel_out(track_pixels[6])
    );

    track track_8 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .selected(track_select[7]),
        .track_name(track_names[7]),
//        .track_name(test_name),
        .mute_track(track_muted[7]),
        .name_track(track_named[7]),
        .mute_focused(focus_mute[7]),
        .name_focused(focus_name[7]),
```

```
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
        .x_in(TRACKS_START_X),
        .y_in(TRACKS_START_Y + (TRACK_HEIGHT*7)),
        .pixel_out(track_pixels[7])
    );

    localparam FOCUSED_BORDER_THICKNESS = 6;
    localparam FOCUSED_COLOR = 12'h0FF;
    logic [11:0] focused_color;
    assign focused_color = (focused) ? FOCUSED_COLOR : 12'h000;

    draw_rectangle #(.WIDTH(TRACK_WIDTH + (FOCUSED_BORDER_THICKNESS << 1)),
                    .HEIGHT((TRACK_HEIGHT*8) + (FOCUSED_BORDER_THICKNESS << 1)))
    box_border (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .x_outer_in(TRACKS_START_X - FOCUSED_BORDER_THICKNESS),
        .y_outer_in(TRACKS_START_Y - FOCUSED_BORDER_THICKNESS),
        .x_inner_in(TRACKS_START_X),
        .y_inner_in(TRACKS_START_Y),
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
        .color(focused_color),
        .pixel_out(track_pixels[8])
    );

endmodule

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

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

```
module track #(parameter BORDER_THICKNESS = 4,
              parameter BORDER_COLOR = 12'hFFF)(
        input clk_in,
        input rst_in,
        input selected,
        input [47:0] track_name,
        input mute_track,
        input name_track,
        input mute_focused,
        input name_focused,
        input [10:0] hcount_in,
        input [9:0] vcount_in,
        input [10:0] x_in,
        input [9:0] y_in,
        output logic [11:0] pixel_out
    );

    localparam BACKGROUND_WIDTH = 300;
    localparam BACKGROUND_HEIGHT = 50;
    localparam MUTE_WIDTH = 30;
    localparam MUTE_HEIGHT = 70;
    localparam NAME_BACKGROUND_HEIGHT = 20;
    localparam NAME_BACKGROUND_WIDTH = BACKGROUND_WIDTH;

    localparam BORDER_SELECTED_COLOR = 12'hFF0;
    logic [11:0] border_color;
    logic [11:0] border_pixel;
    assign border_color = (selected) ? BORDER_SELECTED_COLOR : BORDER_COLOR;

    draw_rectangle #(.WIDTH(MUTE_WIDTH + BACKGROUND_WIDTH + (BORDER_THICKNESS <<
1)),
```

```
                .HEIGHT(NAME_BACKGROUND_HEIGHT + BACKGROUND_HEIGHT +
(BORDER_THICKNESS << 1)))
   border (
      .clk_in(clk_in),
      .rst_in(rst_in),
      .x_outer_in(x_in),
      .y_outer_in(y_in),
      .x_inner_in(x_in + BORDER_THICKNESS),
      .y_inner_in(y_in + BORDER_THICKNESS),
      .hcount_in(hcount_in),
      .vcount_in(vcount_in),
      .color(border_color),
      .pixel_out(border_pixel)
   );

   localparam BACKGROUND_COLOR = 12'h00F;
   logic [11:0] background_pixel;

   draw_rectangle #(.WIDTH(BACKGROUND_WIDTH),
                .HEIGHT(BACKGROUND_HEIGHT))
   background (
      .clk_in(clk_in),
      .rst_in(rst_in),
      .x_outer_in(x_in + BORDER_THICKNESS + MUTE_WIDTH),
      .y_outer_in(y_in + BORDER_THICKNESS + NAME_BACKGROUND_HEIGHT),
      .x_inner_in(1024),
      .y_inner_in(1024),
      .hcount_in(hcount_in),
      .vcount_in(vcount_in),
      .color(BACKGROUND_COLOR),
      .pixel_out(background_pixel)
   );
```

```
    localparam NAME_BACKGROUND_COLOR = 12'h999;
    localparam NAME_BACKGROUND_FOCUSED_COLOR = 12'h333;
    localparam NAME_FONT_COLOR = 12'hFFF;
    logic [5:0] name_array [8:0];
    logic [11:0] name_pixel;
    logic [11:0] name_background_color;
    assign name_background_color = (name_focused) ? NAME_BACKGROUND_FOCUSED_COLOR :
NAME_BACKGROUND_COLOR;

//    assign name_array[0] = 6'b00_0000;
//    assign name_array[1] = track_name[5:0];
//    assign name_array[2] = track_name[11:6];
//    assign name_array[3] = track_name[17:12];
//    assign name_array[4] = track_name[23:18];
//    assign name_array[5] = track_name[29:24];
//    assign name_array[6] = track_name[35:30];
//    assign name_array[7] = track_name[41:36];
//    assign name_array[8] = track_name[47:42];

    always_ff @(posedge name_track) begin
        name_array[0] <= 6'b00_0000;
        name_array[1] <= track_name[5:0];
        name_array[2] <= track_name[11:6];
        name_array[3] <= track_name[17:12];
        name_array[4] <= track_name[23:18];
        name_array[5] <= track_name[29:24];
        name_array[6] <= track_name[35:30];
        name_array[7] <= track_name[41:36];
        name_array[8] <= track_name[47:42];
    end

    string_display #(.STRING_LEN(9),
                .FONT_COLOR(NAME_FONT_COLOR))(
```

```
    .clk_in(clk_in),
    .rst_in(rst_in),
    .char_array(name_array),
    .hcount_in(hcount_in),
    .vcount_in(vcount_in),
    .x_in(x_in + BORDER_THICKNESS + MUTE_WIDTH + 10),
    .y_in(y_in + BORDER_THICKNESS + (NAME_BACKGROUND_HEIGHT >> 2)),
    .background_color(name_background_color),
    .pixel_out(name_pixel)
);

logic [11:0] name_background_pixel;

draw_rectangle #(.WIDTH(NAME_BACKGROUND_WIDTH),
                 .HEIGHT(NAME_BACKGROUND_HEIGHT))
name_background (
    .clk_in(clk_in),
    .rst_in(rst_in),
    .x_outer_in(x_in + BORDER_THICKNESS + MUTE_WIDTH),
    .y_outer_in(y_in + BORDER_THICKNESS),
    .x_inner_in(1024),
    .y_inner_in(1024),
    .hcount_in(hcount_in),
    .vcount_in(vcount_in),
    .color(name_background_color),
    .pixel_out(name_background_pixel)
);

logic [11:0] mute_pixel;
logic is_muted;

always_ff @(posedge mute_track) begin
    is_muted <= !is_muted;
```

```
    end

    mute_button (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .focused(mute_focused),
        .selected(is_muted),
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
        .x_in(x_in + BORDER_THICKNESS),
        .y_in(y_in + BORDER_THICKNESS),
        .pixel_out(mute_pixel)
    );

    assign pixel_out = background_pixel | border_pixel | name_pixel | name_background_pixel |
mute_pixel;

endmodule
/////////////////////////////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////////
module mute_button #(parameter WIDTH = 30, parameter HEIGHT = 70)(
        input clk_in,
        input rst_in,
        input focused,
        input selected,
        input [10:0] hcount_in,
        input [9:0] vcount_in,
        input [10:0] x_in,
        input [9:0] y_in,
        output logic [11:0] pixel_out
    );
```

```
    localparam BACKGROUND_COLOR = 12'h0F0;
    localparam FOCUSED_COLOR = 12'h5A5;
    localparam MUTED_COLOR = 12'hF00;
    localparam FONT_COLOR = 12'hFFF;
    localparam BOTH_STATES_COLOR = 12'hA55;

    logic [5:0] CAPITAL_M [2:0];
    logic [1:0] state;

    initial begin
        CAPITAL_M[0] = 6'b00_0000;
        CAPITAL_M[1] = 6'b10_1101;
        CAPITAL_M[2] = 6'b00_0000;
    end

    logic [11:0] char_pixel;
    logic [11:0] background_color;

    assign background_color = (focused) ?  ((selected) ? BOTH_STATES_COLOR :
FOCUSED_COLOR) :
                              (selected) ? MUTED_COLOR :
                                          BACKGROUND_COLOR;

    string_display #(.STRING_LEN(3),
            .FONT_COLOR(FONT_COLOR))
    M (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .char_array(CAPITAL_M),
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
        .x_in(x_in + (WIDTH >> 1) - 14),
```

```
        .y_in(y_in + (HEIGHT >> 1) - 12),
        .background_color(background_color),
        .pixel_out(char_pixel)
    );

    logic [11:0] background_pixel;
    draw_rectangle #(.WIDTH(WIDTH),
                     .HEIGHT(HEIGHT))
    name_background (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .x_outer_in(x_in),
        .y_outer_in(y_in),
        .x_inner_in(1024),
        .y_inner_in(1024),
        .hcount_in(hcount_in),
        .vcount_in(vcount_in),
        .color(background_color),
        .pixel_out(background_pixel)
    );

    assign pixel_out = char_pixel | background_pixel;

endmodule

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

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

module play_button #(parameter WIDTH = 64, parameter HEIGHT = 64)(
        input clk_in,
        input rst_in,
```

```
        input [10:0] hcount_in,
        input [9:0] vcount_in,
        input [10:0] x_in,
        input [9:0] y_in,
        input selected,
        input play,
        output logic [11:0] pixel_out
    );

    localparam PLAY_BUTTON_COLOR = 12'h0F0;
    localparam SELECTED_COLOR = 12'h0FF;
    localparam PLAYING_COLOR = 12'h00F;

    logic [12:0] image_addr;
    logic [1:0] color;
    logic [1:0] color_selected;
    // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   play_button_rom  rom (.clka(clk_in), .addra(image_addr), .douta(color));
//   play_button_selected_rom  rom_1 (.clka(clk_in), .addra(image_addr), .douta(color_selected));

   always_ff @(posedge clk_in) begin
        if (rst_in) begin
            pixel_out <= 12'b0;
        end else begin
            if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
                (vcount_in >= y_in && vcount_in < (y_in+HEIGHT))) begin
                case (color)
                    2'b00: pixel_out <= 12'h000;
                    2'b01: pixel_out <= (play) ? PLAYING_COLOR : (selected) ? SELECTED_COLOR :
PLAY_BUTTON_COLOR;
                    2'b10: pixel_out <= 12'h000;
                endcase
```

```
            end else begin
                pixel_out <= 12'h000;
            end
        end
    end

endmodule

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

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

module record_button(
        input clk_in,
        input rst_in,
        input [10:0] hcount_in,
        input [9:0] vcount_in,
        input [10:0] x_in,
        input [9:0] y_in,
        input selected,
        input record,
        output logic [11:0] pixel_out
    );

    localparam COLOR = 12'hF00;
    localparam SELECTED_COLOR = 12'hE93;
    localparam RECORDING_COLOR = 12'hFF0;

    logic inner_pixel;

    draw_circle #(.WIDTH(64), .HEIGHT(64)) inner (
        .clk_in(clk_in),
```

```
        .rst_in(rst_in),
        .x(x_in),
        .y(y_in),
        .r_o(32),
        .r_i(0),
        .hcount(hcount_in),
        .vcount(vcount_in),
        .pixel(inner_pixel)
    );

    logic outer_pixel;

    draw_circle #(.WIDTH(64), .HEIGHT(64)) outer (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .x(x_in),
        .y(y_in),
        .r_o(48),
        .r_i(40),
        .hcount(hcount_in),
        .vcount(vcount_in),
        .pixel(outer_pixel)
    );

    assign pixel_out = (inner_pixel || outer_pixel) ? ((record) ? RECORDING_COLOR : (selected) ?
SELECTED_COLOR : COLOR) : 12'h000;

endmodule

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

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

```
module draw_circle #(parameter WIDTH = 1024, parameter HEIGHT = 768)(
    input clk_in,
    input rst_in,
    input [10:0] x,
    input [9:0] y,
    input [11:0] r_o,
    input [11:0] r_i,
    input [10:0] hcount,
    input [9:0] vcount,
    output logic pixel
);

    logic [11:0] x_1;
    logic [11:0] y_1;
    logic [23:0] x_2;
    logic [23:0] y_2;
    logic [23:0] r_o_2;
    logic [23:0] r_i_2;

    always_comb begin
        x_1 = (x > hcount) ? x - hcount : hcount - x;
        y_1 = (y > vcount) ? y - vcount : vcount - y;
    end

    always_ff @(posedge clk_in) begin
        if (rst_in) begin
            x_2 <= 24'b0;
            y_2 <= 24'b0;
            r_i_2 <= 24'b0;
            r_o_2 <= 24'b0;
        end else begin
            x_2 <= x_1 * x_1;
```

```
            y_2 <= y_1 * y_1;
            r_o_2 <= r_o * r_o;
            r_i_2 <= r_i * r_i;

            if ((x_2 + y_2 < r_o_2) && (x_2 + y_2 >= r_i_2)) begin
                pixel <= 1'b1;
            end else begin
                pixel <= 1'b0;
            end
        end
    end

endmodule

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

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

module synchronize #(parameter NSYNC = 2,
                     parameter BUS_WIDTH = 0
                   )  // number of sync flops.  must be >= 2
                   (input clk_in,
                    input [BUS_WIDTH:0] in,
                    output logic [BUS_WIDTH:0] out);

  reg [NSYNC-2:0] sync;

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

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

Section 2

Part 1

```
/////////////////////////////////////////////////////////////////////////////////////////////////////////
///////////////////////////////////////
`timescale 1ns / 1ps



module gatlin_fft #(parameter STRIDE=960, WINDOW=16384, DATA_WIDTH = 16) (
        input clk,
        input rst,
        input [DATA_WIDTH-1:0] data_in,
        input valid_in,
        output logic [15:0] freq_out,
        output logic [7:0] mag_out,
        output logic valid_out,
        output logic fft_valid,
        output logic [31:0] fft_data_out
    );

    logic [15:0] stride_counter = 16'h0000;
    logic start_fft = 1'b0;

    logic [3:0] state = 4'h0;

    logic [DATA_WIDTH-1:0] fft_index = 16'b0;
    logic fft_data_in_valid_start = 1'b0;
    logic fft_data_in_valid = 1'b0;
```

```
logic signed [DATA_WIDTH-1:0] fft_data_in = 8'h00;
logic fft_data_in_last = 1'b0;
logic ready = 1'b0;
//logic [31:0] fft_data_out;
logic fft_data_out_last;
logic fft_data_out_valid;
logic fft_ready;


logic w_fifo_full;
logic w_fifo_wr;
logic [DATA_WIDTH-1:0] w_fifo_data_in;
logic w_fifo_rd;
logic [DATA_WIDTH-1:0] w_fifo_data_out;

logic sb_fifo_full;
logic sb_fifo_empty;
logic sb_fifo_wr;
logic [DATA_WIDTH-1:0] sb_fifo_data_in;
logic sb_fifo_rd;
logic [DATA_WIDTH-1:0] sb_fifo_data_out;
logic loading_fft = 1'b0;
logic [1:0] fifo_trans_buf = 2'b00;
logic w_fifo_cycle_wr;
logic w_fifo_hold_rd;
logic w_fifo_start_rd = 1'b0;

logic [15:0] max_bin;

logic event_tlast_unexpected;
logic event_tlast_missing;

assign sb_fifo_data_in = data_in;
```

```
assign sb_fifo_wr = valid_in;

assign fft_valid = fft_data_out_valid;

assign w_fifo_wr = fifo_trans_buf[1] | w_fifo_cycle_wr;
assign sb_fifo_rd = fifo_trans_buf[0];
assign w_fifo_rd = w_fifo_hold_rd | (w_fifo_start_rd&~fft_ready);


always_comb begin
  loading_fft = (state==4'h2)?1'b1:1'b0;
  w_fifo_hold_rd = (state==4'h2)?1'b1:1'b0;
  w_fifo_cycle_wr = ((state==4'h2)|(state==4'h4))?1'b1:1'b0;
  fft_data_in_last = (state==4'h4)?1'b1:1'b0;
  //fft_data_in_valid = (state==5'h1)?1'b1:1'b0;
  w_fifo_data_in = (loading_fft)?w_fifo_data_out:sb_fifo_data_out;

end

always_ff @(posedge clk) begin
  if (rst) begin
    stride_counter <= 16'h0000;
    state <= 4'h3;
    fft_index <= 16'h0000;
  end else begin
    fifo_trans_buf <= {fifo_trans_buf[0],(~sb_fifo_empty&~loading_fft&~fifo_trans_buf[0])};
    case (state)
      4'h0 :  begin
            if (stride_counter<STRIDE) begin
              stride_counter <= valid_in?stride_counter + 1'b1:stride_counter;
            end else begin
              stride_counter <= 16'h0000;
              fft_index <= fft_index+1'b1;
```

```
            state <= 5'h1;
         end
      end

4'h1 :  begin
         fft_data_in_valid <= 1'b1;
         w_fifo_start_rd <= 1'b1;
         state <= (fft_ready)?4'h1:4'h2;
      end

4'h2 :  begin
         state <= (fft_index<WINDOW-2)?4'h2:4'h4;
         //fft_data_in_valid <= (fft_index<WINDOW-2)?1'b1:1'b0;
         fft_index <= fft_index+1'b1;
      end

4'h4 :  begin
         fft_data_in_valid <= 1'b0;
         fft_index <= 16'h0000;
         state <= 4'h0;
      end

4'h3 :  begin
         if (stride_counter<WINDOW-STRIDE-1) begin
            stride_counter <= valid_in?stride_counter +1'b1:stride_counter;
         end else begin
            stride_counter <= 16'h0000;
            state <= 5'h0;
         end
      end


endcase
```

```
    end
end



fifo_generator_0 window_fifo(
    .clk(clk),
    .srst(rst),
    .full(w_fifo_full),
    .din(w_fifo_data_in),
    .wr_en(w_fifo_wr),
    //.empty(),
    .dout(w_fifo_data_out),
    .rd_en(w_fifo_rd));

fifo_generator_1 sample_buffer_fifo(
    .clk(clk),
    .srst(rst),
    .full(sb_fifo_full),
    .empty(sb_fifo_empty),
    .din(sb_fifo_data_in),
    .wr_en(sb_fifo_wr),
    //.empty(),
    .dout(sb_fifo_data_out),
    .rd_en(sb_fifo_rd));


xfft_0 fft(
    .aclk(clk),
    .s_axis_data_tdata({16'h0000,w_fifo_data_out}),
    .s_axis_data_tlast(fft_data_in_last),
    .s_axis_data_tready(fft_ready),
    .s_axis_config_tdata(32'h0),
```

```
        .s_axis_config_tvalid(1'b0),
        .s_axis_data_tvalid(fft_data_in_valid),
        .m_axis_data_tdata(fft_data_out),
        .m_axis_data_tlast(fft_data_out_last),
        .m_axis_data_tvalid(fft_data_out_valid),

        .event_tlast_unexpected(event_tlast_unexpected),
        .event_tlast_missing(event_tlast_missing));

    max_ident mi(
        .clk(clk),
        .real_mag(fft_data_out[15:0]),
        .imag_mag(fft_data_out[31:16]),
        .rst(rst),
        .valid_in(fft_data_out_valid),
        .valid_out(valid_out),
        .max_bin(freq_out),
        .max_amp(mag_out));



endmodule
//////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////
```

Part 2

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////
`timescale 1ns / 1ps

module max_ident #(parameter WINDOW_SIZE = 16384)(
    input clk,
    input [15:0] real_mag,
```

```
 input [15:0] imag_mag,
 input rst,
 input valid_in,
 output logic valid_out,
 output logic [15:0] max_bin = 1'h0000,
 output logic [7:0] max_amp
 );

logic [15:0] bin_counter = 16'h0000;
logic [7:0] cur_bin_mag;
logic [7:0] max_bin_mag = 8'h00;
logic mag_valid;
logic [7:0] david_mom_gay;


assign max_amp = max_bin_mag;
assign david_mom_gay = cur_bin_mag;


 Mag_Calc mag(
     .real_in(real_mag),
     .imag_in(imag_mag),
     .clk(clk),
     .mag_out(cur_bin_mag),
     .valid_out(mag_valid),
     .valid_in(valid_in));

 always_ff @(posedge clk) begin
     if (rst) begin
         bin_counter <= 16'b0;
         bin_counter <= 16'b0;
         max_bin_mag <= 8'h00;
         max_bin <= 16'h0000;
```

```
            valid_out <= 1'b0;
        end else begin
            if (mag_valid) begin
                if (bin_counter<WINDOW_SIZE-1) begin
                    bin_counter <= bin_counter+1'b1;
                    if (bin_counter<WINDOW_SIZE/2-1) begin
                        if (cur_bin_mag>max_bin_mag) begin
                            max_bin_mag <= cur_bin_mag;
                            max_bin <= bin_counter;
                        end
                    end else begin
                        valid_out <= 1'b1;
                    end
                end
            end else begin
                valid_out <= 1'b0;
                bin_counter <= 16'b0;
                max_bin_mag <= 8'h00;
                max_bin <= 16'h0000;

            end
        end
    end

endmodule
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Part 3

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
`timescale 1ns / 1ps
```

```
module input_flow(
    input clk,
    input rst,
    input [7:0] mic_in,
    input valid_in,
    input fifo_rd,

    input learner_active,

    output logic fifo_full,
    output logic fifo_empty,

    output logic [23:0] fifo_dout,
    output logic fifo_valid,

    output logic [31:0] coeffs,
    output logic ceoffs_valid
  );

  logic [15:0] fund_freq;
  logic [7:0] mag_out;
  logic [15:0] filter_out;
  logic filter_out_valid;
  logic write_sample = 1'b0;
  logic filter_ready;

  logic fft_sample_ready = 1'b1;
  logic fft_sample_valid;

  logic fft_data_valid;
  logic [31:0] fft_data;
```

```
gatlin_fft fft(
    .clk(clk),
    .rst(rst),
    .data_in(filter_out),
    .valid_in(filter_out_valid),
    .freq_out(fund_freq),
    .mag_out(mag_out),
    .valid_out(fft_sample_valid),
    .fft_valid(fft_data_valid),
    .fft_data_out(fft_data));

learner learn(
    .clk(clk),
    .rst(rst),
    .fund_freq(fund_freq),
    .fund_mag(mag_out),
    .fund_valid(fft_sample_valid),
    .fft_data(fft_data),
    .fft_data_valid(fft_data_valid),
    .active(learner_active),
    .harmonic_coeffs(coeffs),
    .valid_out(ceoffs_valid));

output_fifo fifo(
    .clk(clk),
    .srst(rst),
    .full(fifo_full),
    .empty(fifo_empty),
    .din({fund_freq,mag_out<<4}),
    .dout(fifo_dout),
    .wr_en(write_sample),
    .valid(fifo_valid),
    .rd_en(fifo_rd));
```

```
    fir_0 filter(
        .aclk(~clk),
        .s_axis_data_tdata(mic_in),
        .s_axis_data_tready(filter_ready),
        .s_axis_data_tvalid(valid_in),
        .m_axis_data_tdata(filter_out),
        .m_axis_data_tvalid(filter_out_valid));

    always_ff @(posedge clk) begin
        if (fft_sample_valid) begin
            if (fft_sample_ready) begin
                fft_sample_ready <= 1'b0;
                write_sample <= 1'b1;
            end else begin
                write_sample <= 1'b0;
            end
        end else begin
            fft_sample_ready = 1'b1;
        end
    end

endmodule

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

Part 4

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////
`timescale 1ns / 1ps

module learner(
```

```
    input clk,
    input rst,
    input [15:0] fund_freq,
    input [7:0] fund_mag,
    input fund_valid,
    input [31:0] fft_data,
    input fft_data_valid,
    input active,
    output logic [31:0] harmonic_coeffs,
    output logic valid_out
);

logic [3:0] state = 4'h0;
logic [3:0] harm_index = 4'h0;

logic [7:0] coeff_1;
logic [7:0] coeff_2;
logic [7:0] coeff_3;
logic [7:0] coeff_4;

logic [7:0] coeff_bin_1;
logic [7:0] coeff_bin_2;
logic [7:0] coeff_bin_3;
logic [7:0] coeff_bin_4;

logic [15:0] fund_freq_reg;
logic [7:0] fund_mag_reg;

logic fifo_full;
logic fifo_empty;
logic [31:0] fifo_out;
logic fifo_wr;
logic fifo_rd;
```

```
logic [15:0] bin_counter = 16'h0000;
logic mag_out;
logic mag_valid;
logic fifo_valid;


learner_fifo fifo(
    .clk(clk),
    .srst(rst),
    .full(fifo_full),
    .empty(fifo_empty),
    .din(fft_data),
    .dout(fifo_out),
    .wr_en(fifo_wr),
    .rd_en(fifo_rd),
    .valid(fifo_valid));

//16
Mag_Calc mc(
    .clk(clk),
    .real_in(fifo_out[15:0]),
    .imag_in(fifo_out[31:16]),
    .mag_out(mag_out),
    .valid_out(mag_valid),
    .valid_in(fifo_valid));



always_ff @(posedge clk) begin
    if (rst) begin
        state <= 4'h0;
    end else begin
        case (state)
```

```
4'h0 :  begin //idle
            state <= active?4'h1:4'h0;
            bin_counter <= 16'h0000;
        end
4'h1 :  begin //looks for low valid to make sure to start at the beggining of a cycle
            state <= fft_data_valid ? 4'h2: 4'h3;
        end
4'h2 :  begin
            fifo_wr <= fft_data_valid ? 1'b1: 1'b0;
            state <= fft_data_valid ? 4'h3 : 4'h2;
        end
4'h3 :  begin
            fifo_wr <= fft_data_valid ? 1'b1: 1'b0;
            state <= fft_data_valid ? 4'h3 : 4'h4;
        end
4'h4 :  begin //waiting for fft_calc to finish and
            state <= fund_valid ? 4'h5 : 4'h4;
        end
4'h5 :  begin
            fund_freq_reg <= fund_freq;
            fund_mag_reg <= fund_mag;
            state <= 4'h6;
        end
4'h6 :  begin
            coeff_bin_1 <= fund_freq<<1;
            coeff_bin_2 <= fund_freq<<2+fund_freq;
            coeff_bin_3 <= fund_freq<<2;
            coeff_bin_4 <= fund_freq<<2+fund_freq;
            state <= 4'h7;
            fifo_rd <= 1'b1;
        end
5'h7 :  begin
            if (mag_valid) begin
```

```
                    bin_counter <= bin_counter+ 1'b1;
                    if (bin_counter==coeff_bin_1) begin
                        coeff_1 <= mag_out;
                    end
                    if (bin_counter==coeff_bin_2) begin
                        coeff_2 <= mag_out;
                    end
                    if (bin_counter==coeff_bin_3) begin
                        coeff_3 <= mag_out;
                    end
                    if (bin_counter==coeff_bin_4) begin
                        coeff_4 <= mag_out;
                    end
                end
                if (bin_counter>16384/2) begin
                    state <= 4'h0;
                end
            end

        default : state <= 4'h0;
      endcase
    end
  end
endmodule

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

Section 3

Part 1

```
//////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////
`timescale 1ns / 1ps

module fundamental_ifft(input clk,
                    input rst,
                    // input new_sample, // high pulse when there is a new sample input to the module
--THIS WILL BE AN INTERNAL
                    input new_packet, // pulse to indicate new packet is sent to the module
                    input [23:0] packet_in, // with FIFO implemented, this will be a pack that contains the
fundamental and amplitude that we then throw into the FIFO
                    input [31:0] coefficients,
                    output logic ready_for_note = 1, // high indicates that the module is capable of
starting another new sample  -- we're not gonna need this anymore with fifo
                    output logic fifo_in_full,
                    output logic signed [7:0] sample_out // these will be the outputs from the FIFO --
modulated at the desired output rate (48 kHz now)

);

  // FFT_SIZE allows us to create input data
  parameter FFT_SIZE = 16384;
  parameter SAMPLES_OUT = 960;

  // INITIALIZING FFT MASTER/SLAVE
  logic [27:0] scale_sch_1 = 28'b0;
  logic [27:0] scale_sch_0 = 28'b0101010101010101010101010101;
  // config slave channel signals
  logic s_axis_config_tready; // slave is ready
  logic [31:0] s_axis_config_tdata = {5'b0, scale_sch_1, 1'b1};
  logic s_axis_config_tvalid = 1; // payload is valid
  // data slave channel signals
  logic s_axis_data_tvalid = 0; // payload is valid
  logic s_axis_data_tready; // slave is ready
  logic [31:0] s_axis_data_tdata = 32'b0; // data payload
  logic s_axis_data_tlast = 0; // indicates end of packet
```

```
// data master channel signals
logic m_axis_data_tvalid; // payload is valid
logic m_axis_data_tready = 1; // slave is ready
logic [31:0] m_axis_data_tdata; // data payload
logic m_axis_data_tlast; // indicates end of packet
// event signals
logic event_frame_started;
logic event_tlast_unexpected;
logic event_tlast_missing;
logic event_status_channel_halt;
logic event_data_in_channel_halt;
logic event_data_out_channel_halt;

// INITIALIZE INPUTS AND OUTPUTS FOR IFFT
logic [14:0] num_samples = 0; // keep track of number of samples
logic [15:0] ch_0_input_re;
logic [15:0] ch_0_input_im;
logic [15:0] ch_0_output_re;
logic [15:0] ch_0_output_im;

assign ch_0_input_re = s_axis_data_tdata[15:0];
assign ch_0_input_im = s_axis_data_tdata[31:16];
assign ch_0_output_re = m_axis_data_tdata[15:0];
assign ch_0_output_im = m_axis_data_tdata[31:16];

// INITIALIZE FFT
xfft_0_matt my_fft(.aclk(clk),
        .s_axis_config_tvalid(s_axis_config_tvalid),
        .s_axis_config_tready(s_axis_config_tready),
        .s_axis_config_tdata(s_axis_config_tdata),
        .s_axis_data_tvalid(s_axis_data_tvalid),
        .s_axis_data_tready(s_axis_data_tready),
        .s_axis_data_tdata(s_axis_data_tdata),
        .s_axis_data_tlast(s_axis_data_tlast),
        .m_axis_data_tvalid(m_axis_data_tvalid),
        .m_axis_data_tready(m_axis_data_tready),
        .m_axis_data_tdata(m_axis_data_tdata),
        .m_axis_data_tlast(m_axis_data_tlast),
        .event_frame_started(event_frame_started),
        .event_tlast_unexpected(event_tlast_unexpected),
        .event_tlast_missing(event_tlast_missing),
        .event_status_channel_halt(event_status_channel_halt),
        .event_data_in_channel_halt(event_data_in_channel_halt),
```

```
        .event_data_out_channel_halt(event_data_out_channel_halt)
    );

 // INITIALIZE INPUTS AND OUTPUTS FOR FREQUENCY GENERATOR
 logic freq_active = 0; // not active to start, only high while the freq_generator is feeding samples
to fft
 logic start_divide;
 logic [15:0] quotient;
 logic divide_ready;
 logic [15:0] index;
 logic [15:0] real_coeff;
 logic [15:0] imag_coeff;
 logic [15:0] fundamental;
 logic [7:0] amplitude;

 // INITIALIZE FREQUENCY GENERATOR
//  freq_generator my_gen(.clk(clk),
//                 .fundamental(fundamental),
//                 .amplitude(amplitude),
//                 .coefficients(coefficients),
//                 .active(freq_active),
//                 .real_coeff(real_coeff),
//                 .imag_coeff(imag_coeff),
//                 .start_divide(start_divide),
//                 .quotient(quotient),
//                 .divide_ready(divide_ready),
//                 .index(index));

 freq_generator_2 my_gen(.clk(clk),
                 .fundamental(fundamental),
                 .amplitude(amplitude),
                 .coefficients(coefficients),
                 .active(freq_active),
                 .real_coeff(real_coeff),
                 .imag_coeff(imag_coeff));



 // INITIALIZE INPUTS AND OUTPUTS FOR INPUT FIFO
 parameter INPUT_CLOCKS = 2000000; // divider for 50 Hz input
 //parameter INPUT_CLOCKS = 20000; // use this for simulation to send input right away
 logic fifo_in_write;
 logic fifo_in_read; // want to read from the fifo in at 50 Hz when it is not empty
```

```
  logic [23:0] packet_out; // this is the fundamental and amplitude we read from the fifo
  // logic fifo_in_full;
  logic fifo_in_empty;
  logic [31:0] counter = 0;
  logic new_sample;
  logic have_read = 0; // keep track of whether we have read anything at all yet

  // INITIALIZE INPUTS AND OUTPUTS FOR OUTPUT FIFO
  parameter OUTPUT_CLOCKS = 2082; // divider so we can output at the same rate we took data
in, 48 kHz
  //parameter OUTPUT_CLOCKS = 1; // faster output for the simulation
  logic [15:0] fifo_in;
  logic fifo_write;
  logic fifo_read;
  logic [15:0] fifo_out;
  logic fifo_full;
  logic fifo_empty;
  logic [12:0] fifo_count;
  logic [10:0] samples_written = 0;
  logic done_writing = 0; // tells us whether we already wrote the necessary samples frome the
IFFT to the FIFO
  logic [11:0] clock_counter = 0; // count how many clock cycles we have waited
  logic sample_trigger;
  assign sample_trigger = (clock_counter == OUTPUT_CLOCKS);

  // INITIALIZE FIFO FOR READING INPUT HERE
  fifo_generator_2 fifo_input(.clk(clk), //fifo_generator_1_matt
                        .srst(rst),
                        .din(packet_in),
                        .wr_en(fifo_in_write),
                        .rd_en(fifo_in_read),
                        .dout(packet_out),
                        .full(fifo_in_full),
                        .empty(fifo_in_empty));


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

  // INITIALIZE FIFO FOR OUTPUTTING NOTE VALUES
  fifo_generator_0_matt fifo_output(.clk(clk),
                         .srst(rst),
                         .din(fifo_in),
```

```
                        .wr_en(fifo_write),
                        .rd_en(fifo_read),
                        .dout(fifo_out),
                        .full(fifo_full),
                        .empty(fifo_empty)
                        //.data_count(fifo_count)
                        );

  // INITIALIZE ILA FOR DEBUGGING FIFO OUT AND SAMPLE OUT
//  ila_0 myila(.clk(clk),
//              .probe0(new_sample),
//              .probe1(packet_out),
//              .probe2(real_coeff),
//              .probe3(ch_0_output_re),
//              .probe4(sample_out));

  always @(posedge clk) begin
    // WRITING TO INPUT FIFO
    if (new_packet) begin // if we are receiving a new packet, write the information to the FIFO --
these will be coming in at roughly 50 Hz
       fifo_in_write <= 1; // write packet to FIFO -- it will already be on the FIFO input line
    end else begin
       fifo_in_write <= 0;
    end

//    if (!fifo_in_empty) begin // read samples at 50 Hz and pulse new sample when we read one
//       if (counter == INPUT_CLOCKS) begin // set fundamental and amplitude, pulse new_sample
-- it will already have the coefficients
//          fifo_in_read <= 1;
//          //fundamental <= packet_out[23:8];
//          //amplitude <= packet_out[7:0];
//          new_sample <= 0;
//          counter <= 0;
//          have_read <= 1;
//       end else if (counter == 2 && have_read) begin
//          fifo_in_read <= 0;
//          counter <= counter + 1;
//          new_sample <= 1;
//          fundamental <= packet_out[23:8];
//          amplitude <= packet_out[7:0];
//       end else begin // increment counter and set new_sample low
//          fifo_in_read <= 0;
//          counter <= counter + 1;
```

```
//           new_sample <= 0;
//       end
//    end else begin // if fifo is empty -- we should reset the have read
//       have_read <= 0;
//    end

    if (!fifo_in_empty) begin // read samples at 50 Hz and pulse new sample when we read one
        if (counter == INPUT_CLOCKS - 2) begin // set fundamental and amplitude, pulse
new_sample -- it will already have the coefficients
            fifo_in_read <= 1;
            counter <= counter + 1;
            new_sample <= 0;
            //have_read <= 1;
        end else if (counter == INPUT_CLOCKS) begin
            fundamental <= packet_out[23:8];
            amplitude <= packet_out[7:0];
            new_sample <= 1;
            counter <= 0;
        end else begin // increment counter and set new_sample low
            fifo_in_read <= 0;
            counter <= counter + 1;
            new_sample <= 0;
        end
    end else begin // if fifo is empty -- we should reset the have read
        //have_read <= 0;
    end


    // TAKES CARE OF INPUT SIDE OF THINGS -- needs a High new_samples pulse accompanied
by the sample data to work
    // after we get the FIFO working we can have ready_for_note be an internal instead of an output
    // we will also now make new_sample and internal that we pulse high when reading in packet
from fifo
    // we might not even need ready_for_note?
    if (new_sample && ready_for_note) begin // new note available and not busy -- start producing
coefficients and transforming
        freq_active <= 1;
        ready_for_note <= 0;
        num_samples <= 0; // reset number of samples
    end else if (num_samples < (FFT_SIZE - 1) && num_samples > 0) begin // keep going until we
accumulate all of the samples we need into the IFFT
        s_axis_data_tdata <= {imag_coeff, real_coeff};
        num_samples <= num_samples + 1;
```

```
    end else if (num_samples == FFT_SIZE - 1) begin // just need one more samples -- it will be zero
        freq_active <= 0;
        s_axis_data_tdata <= 32'b0;
        s_axis_data_tlast <= 1;
        num_samples <= num_samples + 1;
    end else if (num_samples == FFT_SIZE) begin // now we are completely done
        s_axis_data_tvalid <= 0;;
        s_axis_data_tlast <= 0;
        // freq_active <= 0;
    end else if (num_samples == 0 && !ready_for_note) begin // avoid the invalid sample at first
rising edge
        s_axis_data_tvalid <= 1;
        s_axis_data_tdata <= 32'b0;
        num_samples <= num_samples + 1;
    end
    if (m_axis_data_tlast) begin
        ready_for_note <= 1;
    end

    // NOW FOR OUTPUT SIDE OF THINGS -- WRITING TO FIFO
    if (m_axis_data_tvalid && !done_writing && samples_written < SAMPLES_OUT) begin // when
output is valid, write as loong as we have not accrued the necessary number of samples
        fifo_in <= ch_0_output_re;
        fifo_write <= 1;
        samples_written <= samples_written + 1;
    end else if (samples_written == SAMPLES_OUT) begin // time to stop writing to FIFO, we have
accrued the necessary number of samples
        fifo_write <= 0;
        samples_written <= 0;
        done_writing <= 1;
    end else if (m_axis_data_tlast) begin // last sample of the frame -- we reset samples_written
now and done writing
        done_writing <= 0;
    end

    // NOW FOR READING FROM FIFO -- JUST READ OUT THE RESULTS AT 48 kHz when it's not
empty
    if (!fifo_empty) begin // read the results to sample_out at 48 kHz
        if (clock_counter == OUTPUT_CLOCKS)begin
            clock_counter <= 16'b0;
        end else begin
            clock_counter <= clock_counter + 16'b1;
            fifo_read <= 0; // do not read from fifo
```

```
      end
      if (sample_trigger) begin
        fifo_read <= 1;
        sample_out <= fifo_out >> 5; // try setting to highest bits of fifo
      end
    end else begin
      fifo_read <= 0;
    end

  end
endmodule
```

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

Part 2

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 12/09/2019 02:05:23 PM
// Design Name:
// Module Name: output_flow
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
```

```
//       .clk(clk),
//       .rst(rst),
//       // input new_sample, // high pulse when there is a new sample input to the module --THIS
WILL BE AN INTERNAL
//       .new_packet(inf_fifo_valid), // pulse to indicate new packet is sent to the module
//       .packet_in(inf_fifo_dout), // with FIFO implemented, this will be a pack that contains the
fundamental and amplitude that we then throw into the FIFO
//       .coefficients({8'b00000110, 8'b00001000, 8'b00101000, 8'b00010000}),
//       //.ready_for_note = 1, // high indicates that the module is capable of starting another new
sample  -- we're not gonna need this anymore with fifo
//       .fifo_in_full(otf_fifo_full),
//       .sample_out(sample_out));

module output_flow(
        input clk,
        input rst,
        input new_packet,
        input [23:0] packet_in,
        input [31:0] coefficients,
        input [2:0] vol_in,

        output logic fifo_in_full,
        output logic pwm_out
    );

    logic signed [7:0] ifft_out;
    logic [7:0] vol_out;

    fundamental_ifft ifft(
        .clk(clk),
        .rst(rst),
        // input new_sample, // high pulse when there is a new sample input to the module --THIS
WILL BE AN INTERNAL
        .new_packet(new_packet), // pulse to indicate new packet is sent to the module
        .packet_in(packet_in), // with FIFO implemented, this will be a pack that contains the
fundamental and amplitude that we then throw into the FIFO
        .coefficients({8'b00000110, 8'b00001000, 8'b00101000, 8'b00010000}),
        //.ready_for_note = 1, // high indicates that the module is capable of starting another new
sample  -- we're not gonna need this anymore with fifo
        .fifo_in_full(fifo_in_full),
        .sample_out(ifft_out));

        volume_control vc(
```

```
        .vol_in(vol_in),
        .signal_in(ifft_out),
        .signal_out(vol_out));

    pwm pwm(
        .clk_in(clk),
        .rst_in(rst),
        .level_in({~vol_out[7],vol_out[6:0]}),
        .pwm_out(pwm_out));


endmodule



module volume_control (input [2:0] vol_in, input signed [7:0] signal_in, output logic signed[7:0]
signal_out);
  logic [2:0] shift;
  assign shift = 3'd7 - vol_in;
  assign signal_out = signal_in>>>shift;
endmodule

//PWM generator for audio generation!
module pwm (input clk_in, input rst_in, input [7:0] level_in, output logic pwm_out);
  logic [7:0] count;
  assign pwm_out = count<level_in;
  always_ff @(posedge clk_in)begin
    if (rst_in)begin
      count <= 8'b0;
    end else begin
      count <= count+8'b1;
    end
  end
endmodule

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