SLIME:
Single Lonely Instrumentalist Music Enhancer

Michal Szurek and Carine You

Fall 2021

1 Introduction

Alongside the components usually found in an acoustic guitar, a standard electric guitar setup consists of one or more pickups and an amplifier. A pickup is a transducer that converts mechanical string vibrations into small electric signals that are transmitted along a cord to the amplifier, which strengthens the signals to produce sound through one or more loudspeakers. The size, location, and components of the pickup are integral factors affecting the quality of the tone that is produced.

To achieve tones that are beyond the capability of standard guitar pickups, musicians often make use of guitar effect pedals. Each pedal is only able to produce a single effect due to its analog nature, so a guitarist generally requires an arsenal of pedals to create multiple effects during a performance. We sought to emulate these analog effects in digital hardware by taking advantage of the speed of FPGAs to produce unique tones with minimal delay. We were able to implement standard delay, distortion, overdrive, wah-wah, bass-boost, and tremolo guitar effects.

In addition to the aforementioned effects, we added a vocal component to the project with a harmonizer, which maps the pitch of a single vocal or guitar input to a variety of shifted pitches, each of which can be activated with a switch and added back to the original audio to create a several-note harmony. This setup allows a musician to sing by themselves with accompaniment from themselves.

2 Hardware

The project was developed in SystemVerilog on Xilinx Vivado 2019.2 for a Digilent Nexys A7-100T FPGA board, which, among various features, includes 16 switches, 4 Pmod connectors, and 12-bit VGA output.

To provide audio input, we used a generic electric guitar and microphone (models unknown), both adapted to produce output on a 3.5 mm audio cable. The audio signal was amplified with a Fender Frontman 10G electric guitar amplifier and then passed to a Digilent Pmod I2S2 connected to the JA Pmod port on the Nexys board. The Pmod I2S2 features a multi-bit audio A/D converter and a stereo D/A converter, each connected to 3.5mm audio jacks, which allows the Nexys board to transmit and receive stereo audio signals sampled at 44.1 kHz via the I2S protocol at a 24-bit resolution per channel. Processed audio was passed out of the Pmod I2S2 to a Bose SoundLink speaker using a standard AUX cable.

Aside from the sixteen Nexys on-board switches for pedal selection and a button for system reset, user input into the system was provided by signals from four optical rotary encoders connected to the JC and JD Pmod ports of the Nexys board.

To add a visual component to the project, frequency information from the magnitudes of a live short-time Fourier transform of the audio signal was displayed via VGA signals driving a standard monitor.

Figure 1: The main hardware setup. The four rotary encoders are connected to the left of the Neyxs board, while the Pmod I2S2 can be seen on the right.

3 System Design (joint)

Our system consisted of a series of user input, guitar effect, guitar/vocal harmonizer, and audio input/output modules centered around the main audio controller module. This controller module took processed rotary encoder signals for volume and effect magnitudes, as well as on-board switch states for pedal and harmonizer choices. It also received the raw vocal and guitar audio signals from the AXI-Stream I2S protocol module, which was responsible for both sending audio signals and receiving them from the Pmod I2S2 component. Upon taking in the audio and user control signals, the audio controller module provided these signals to the pedal and

harmonizer submodules and fetched corresponding effect-transformed audio signals that were fed back to the AXIS I2S protocol for system output.

A block diagram summarizing the organization of these modules is show in Figure 2, with module implementations to be described in more detail in the following sections. All modules were driven by a 22.591 MHz AXI clock with system reset signal controlled by the "down" button on the Nexys board.

3.1 Top Module

In terms of input, the top module took A/B control pins for each of four rotary encoders, sixteen on/off switch states, a reset button, and a serial data received via I2S protocol from the Pmod I2S2. The outputs of this module were VGA signals to show frequency content, AXIS TX/RX control signals (`tx_mclk`, `tx_lrck`, `tx_sclk`, `rx_mclk`, `rx_lrck`, `rx_sclk`, and serial data to drive the audio output for the Pmod I2S2.

These inputs and outputs interfaced with four rotary decoder instances, an AXIS I2S2 module instance, and an audio controller instance, which are described in more detail in the following subsection.

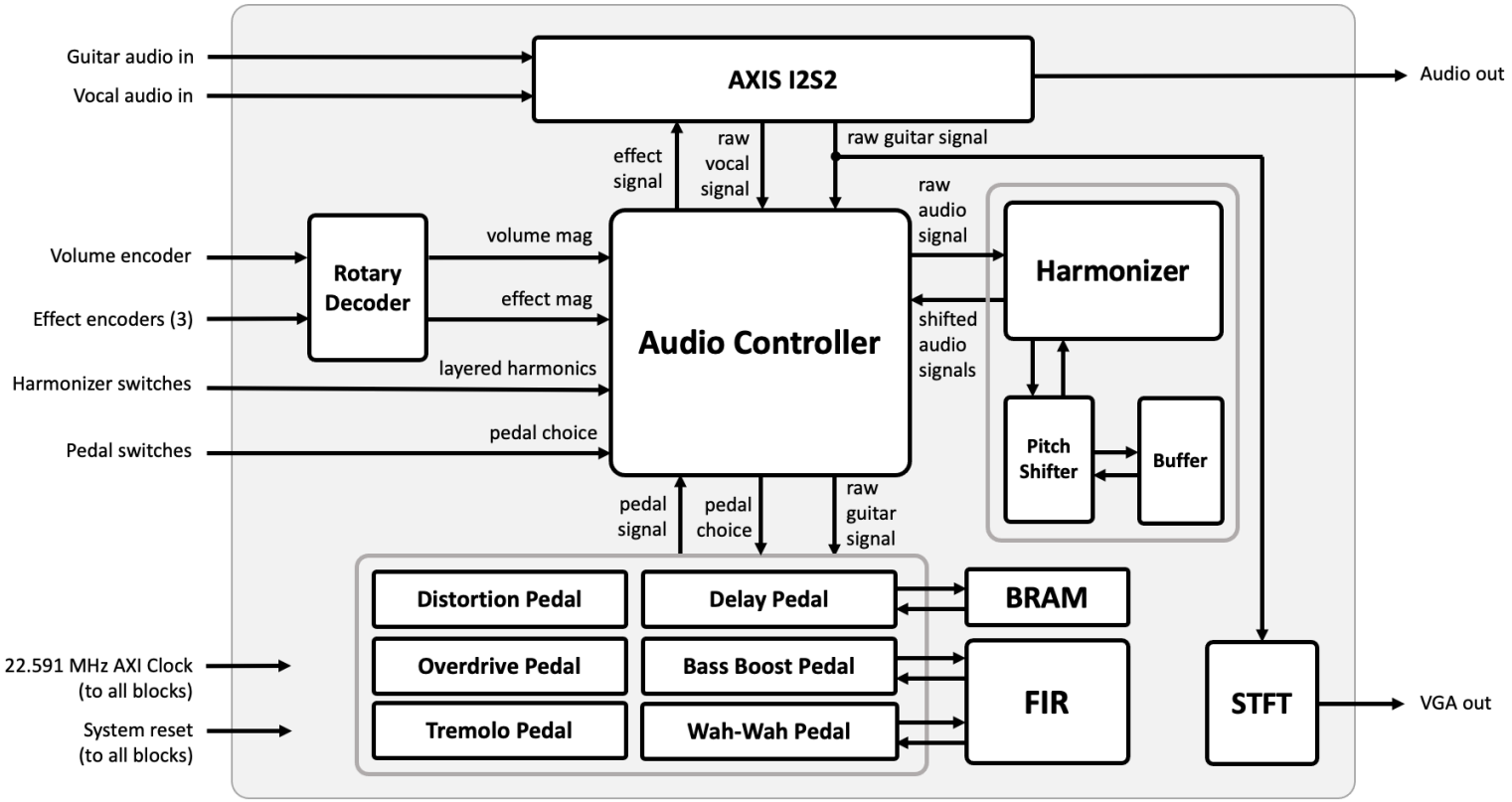
3.2 AXIS I2S2

The AXI-Stream I2S controller module was adapted from Digilent code written by Arthur Brown, which can be found on GitHub, though it was first modified to translate it from Verilog to SystemVerilog. This module took 32-bit TX AXI secondary interface data, 1-bit RX serial data from a Pmod pin, and AXI TX/RX inputs for TX secondary valid, TX secondary last, and RX primary signals. It produced the complementary signals for 32-bit RX AXI primary data and 1-bit TX AXI secondary ready, RX AXI primary valid, RX AXI primary last, and TX/RX Pmod pin drivers (`tx_mclk`, `tx_lrck`, `tx_sclk`, `tx_sdout`, `rx_mclk`, `rx_lrck`, `rx_sclk`).

Within the module, primary and secondary processes to process the AXIS serial data were implemented according to the I2S protocol: whenever a two-word packet was received on the secondary interface, it was transformed by the pedal modules in real-time over a few clock cycles and then sent over the primary interface, with reception of data on the secondary interface halted while processing and transfer took place.

An example of the timing for an I2S frame is shown in Figure 3, where SCLK is the Serial Clock, also known as the Bit Clock, and LRCK is the Left-Right or Word-Select Clock indicating whether a particular set of data corresponds to left or right channel audio for stereo sound. More details on the protocol can be found online.

Figure 2: Block diagram for the SLIME system.



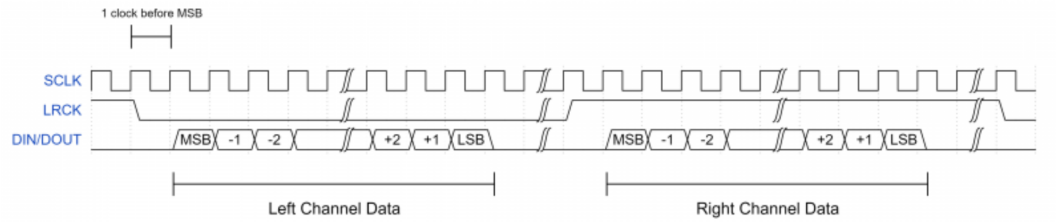


Figure 3: Example timing diagram for a single I2S protocol frame.

3.3 Audio Controller

The audio controller, which was the bread-and-butter of our pedal control mechanism, featured button, switch, and rotary encoder magnitude inputs; AXIS primary/secondary process control and data inputs; AXIS primary/secondary process control and data outputs; and VGA outputs.

fWithin the module, we instantiated either one or two instances of each pedal module (depending on whether the module processed left and right audio channels simultaneously or individually), with a switch-driven multiplexer to select from the outputs produced by each. These modules shared a similar data in/out processing pipeline: on each rising edge of the `s_new_word` signal signifying the arrival of a new 24-bit word on the secondary interface, new data were read into the pedal modules, effect-modified audio data were produced in the following few clock cycles, and output audio data were written on each rising edge of `s_new_packet_r`, occurring one clock cycle after the new two-word packet was ready.

This module additionally included an STFT module instance to produce the VGA signals corresponding to the frequency information for the audio signals computed via a short-time Fourier transform, though in theory it is not necessary for this module to reside in the audio controller, and it could instead be moved to the top-level module.

3.4 Rotary Decoder

Aside from the Nexys on-board switches for pedal selection, user input into the system was provided by signals from rotary encoders. The four aforementioned rotary decoder module instances drove four 8-bit magnitudes, three of which were used as effect magnitude control signals and one of which was used exclusively for volume control. These magnitude signals were passed to the audio controller module.

More specifically, each rotary encoder was connected to one of the available Pmod ports on the Nexys board, feeding in both of its quadrature signals ("A" and "B") for analysis. The rotary decoder module monitored the phasing of the encoder's signals and determined the direction in which the user was spinning the encoder's

knob. To do so, both "A" and "B" quadrature signals of the rotary encoder output were probed at a clock rate much faster than the rate at which they were changing. To prevent metastability, these signals were pushed into two registers as they came in and previous "A" and "B" signals were kept in a buffer.

We implemented a simple edge detection method on the "A" and "B" pulse trains and also defined a directional attribute based on the current and previous values of the "A" and "B" signals to see whether "A" or "B" was leading, specifying clockwise and counterclockwise movement, respectively. Each time an edge was detected, we inspected the directional attribute and incremented or decremented an internal magnitude counter accordingly. The counter magnitude was limited to between 0 and 255 for 8-bit precision. This number was then fed out of the rotary decoder module to act as effect magnitude or volume control for the system.

4 Guitar Effect Pedals

In this section, we describe the implementation of each guitar effect pedal in more detail.

4.1 Delay (Carine)

The delay pedal added three attenuated versions of audio input with varying levels of delay to the original input to produce the pedal output audio signal. This was implemented with two block RAMs of width 24 and depth 65536 (created with the Xilinx IP Block Memory Generator), in which signed 24-bit samples of either left- or right- channel audio were stored and retrieved as appropriate.

Using the 8-bit effect magnitude from the rotary encoder, the amount of delay for the first echo was computed in terms of clock cycles (by multiplying the 8-bit magnitude by 20), after which the delays for the second and third echos were computed to be twice and thrice this amount, respectively. These delay amounts were translated combinationally into BRAM circular read addresses that incremented with every new sample.

Upon receipt of a new packet, the module stored the incoming audio data in the BRAM in the first three clock cycles, retrieved the samples for the first echo and performed a 1-bit signed right-shift in the next 3 cycles, retrieved the samples for the second echo and performed a 2-bit signed right-shift in the next 3 cycles, retrieved the samples for the third echo and performed a 3-bit signed right-shift in the following 3 cycles, and finally added these three echos to the current input signal to produce the output signal in the last cycle. We included three cycles between each retrieval as a conservative workaround for the fact that the read latency of the BRAM was 2 clock cycles.

4.2 Distortion & Overdrive (Michal)

When it comes to guitar pedals that implement harsh clipping effects, the nuances between the different processing techniques can be subtle. Many options exist for creatively clipping signals, each with a slightly varied sound; however, we decided to implement standard versions of "soft" and "hard" clipping to distinguish between "distortion" and "overdrive", respectively.

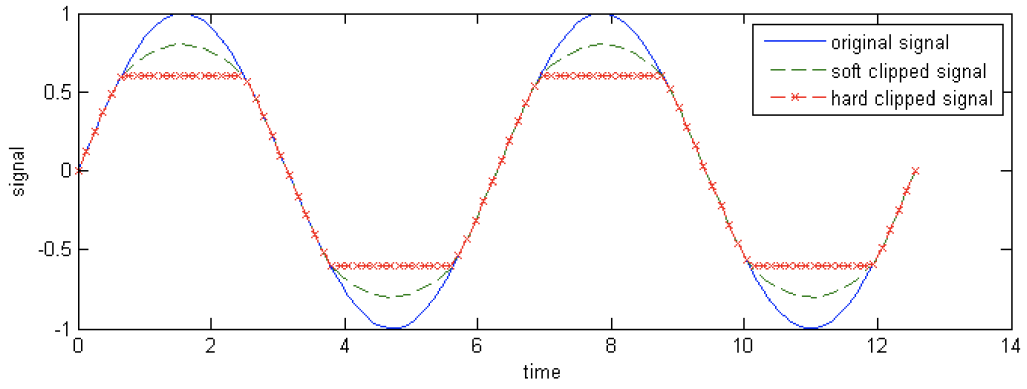


Figure 4: "Soft" clipping vs. "hard" clipping example for a simple sine wave.

4.2.1 Distortion

For the overdrive pedal, we implemented a form of "hard" clipping. In this technique, a threshold was defined to determine the intensity of the overdrive effect. Whenever the signal crossed said threshold, it was strictly clipped to the value of the threshold. We applied this threshold to both sides of the signal, meaning if either the positive or negative signal value exceeded this threshold, clipping was applied. If we consider a pure sine wave, this technique has an effect similar to converting a sinusoidal signal into a square wave, past a certain boundary, which can be seen in Figure 4.

Since the magnitude of this effect was controlled just by the threshold, we tied the threshold value to a multiplicative factor of the rotary encoder effect magnitude value to dynamically control the effect magnitude.

For a preliminary version of the distortion pedal, we also tried an arctangent-modulated version of the audio signal implemented with a CORDIC arctangent module generated by the Xilinx CORDIC IP core, though this did not seem to align with common practice, and so we discarded it in favor of the soft clipping method described above.

4.2.2 Overdrive

For the distortion pedal, we implemented a form of "soft" clipping. In this technique we similarly defined a threshold that determined the intensity of the effect. However, as opposed to flooring the value of the signal to the threshold once it was exceeded, we instead simply scaled down the signal beyond the threshold. This resulted in an effect that was less harsh compared to that of the overdrive pedal and maintained greater quality of sound while still distorting the signal. In terms of a pure sine wave, this technique preserves the shape of the sine wave but flattens out the peaks and troughs.

The magnitude of the overdrive effect was also controlled with the threshold as a function of the rotary encoder input value.

4.3 Bass Boost

As we worked to troubleshoot and implement FIR modules into our system using the Xilinx LogiCORE IP FIR Compiler core to generate FIR modules with taps produced by external programs, we first performed testing with a low-pass bass-isolating filter. Once we reached reliable functionality with the FIR compiler and the resulting modules, we chose to keep the bass-boosting filter. This pedal could be useful for playing accompaniment or producing more bass-heavy output for a different tone.

4.4 Wah-wah (Michal)

The wah-wah pedal, sometimes referred to as a "crybaby" pedal, produces a frequency sweeping effect reminiscent of the "wah-wah" of a crying baby. For this pedal, we implemented a collection of peak filters centered around frequencies in the mid-range of guitar audio. Each of these peak filters had a high-Q band around a single frequency, tapering off at all other frequencies. An example of such a peak filter is shown in Figure 5.

In operating a wah-wah pedal, the user sweeps through these peak filters in real time with some input device, in our case a rotary encoder. Depending on the scaled value of the controlling rotary encoder, our system selected between the set of peak filters, resulting in a wah-wah effect when quickly spinning the rotary encoder back and forth. All of the peak filters were produced through the Xilinx LogiCORE IP FIR Compiler core with taps produced by external programs. We found it more efficient to initialize many FIR modules as opposed to using the FIR module's "reload" capability to continuously pipe in new frequency taps.

If we were to expand our system further to the point of having utilization contractions, we could possibly pivot to having a single FIR module and reloading the FIR taps instead of selecting between multiple module outputs.

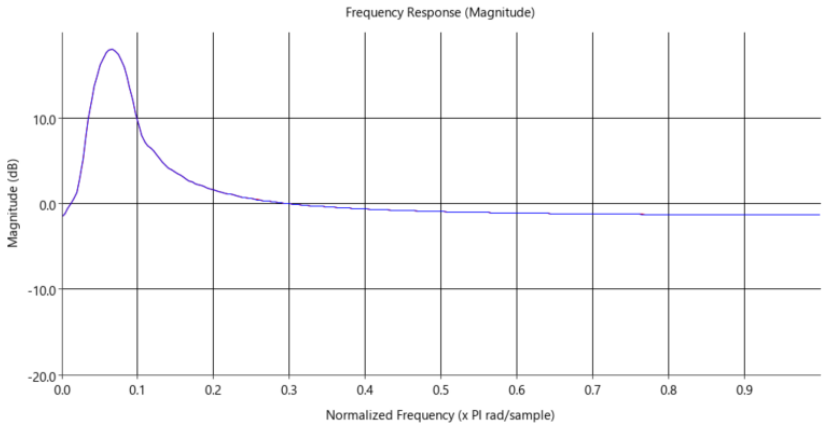


Figure 5: An example of a peak filter used in the wah-wah pedal.

4.5 Tremolo (Carine)

Tremolo pedals produce a modulation effect that creates a periodic change in volume in the guitar sound. Our tremolo pedal modulated the guitar audio signal multiplicatively with a triangle wave whose period was controlled by the an effect magnitude driven by a rotary encoder. This allowed us to modify the rate at which the volume oscillated.

We first generated a triangle wave by incrementing or decrementing a triangle-wave signal on the rising edge every valid input signal (i.e. on the arrival of every new packet) based on a directionality bit controlled by a counter. Upon reaching the half-period number of clock cycles, the directionality bit was flipped accordingly, causing the waveform to go from incrementing to decrementing or vice-versa.

This triangle wave was multiplied sample-wise with the incoming guitar audio signal to produce an output audio signal whose amplitude fluctuated periodically based on the triangle wave, producing a tremolo effect. An example of a triangle-wave-modulated since wave is shown in Figure 6. This pedal could also in theory be implemented with modulation by sawtooth, sine, or square waves, and it could be interesting to test such approaches in the future.

4.6 Harmonizer (Carine)

Most of the functionality of the harmonizer rested on a pitch-shifter module, which made use of a circular buffer that allowed us to play back the audio signal at varying speeds to alter the frequency of the harmonic audio signal. Alongside standard control and audio data signals, the pitch-shifter module took a 32-bit 3.29-format decimal number that regulated how quickly the values in the buffer would be played

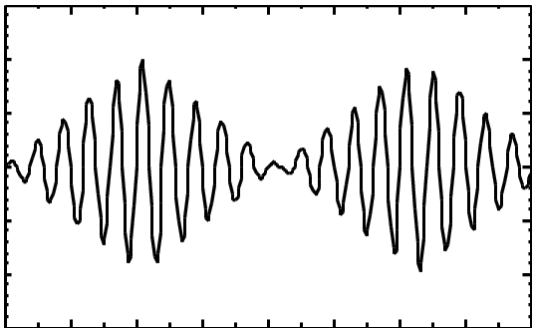


Figure 6: A sine wave with amplitude modulated by a triangle wave, as implemented in the tremolo pedal.

back.

In the first cycle after an audio data sample arrived, the pitch-shifter module stored the sample into a 1024-entry circular buffer. In the second cycle following the arrival of a new sample, the module incremented a pointer by adding the decimal value playback rate provided and then took the integer part to determine which address to read from in the following cycle.

Our harmonizer allowed us to add in the major 3rd, perfect 4th, and perfect 5th interval notes above the incoming audio signal. To compute the rate at which the buffer should be played, we calculated the number of semitones for each interval: since each musical semitone corresponds to a multiplicative factor of $2^{1/12}$ increase in frequency, the major third, which consists of four semitones, should be played back at a rate that is $2^{4/12} \approx 1.260$ times faster than the original playback rate, while the perfect fourth and fifth should be played back at $2^{5/12} \approx 1.335$ and $2^{7/12} \approx 1.498$ times the original rate, respectively.

The harmonizer module created six instances of the pitch-shifter (one left-channel and one right-channel shifter for each interval) each with appropriately computed decimal playback rates, and added these signals to the original tone based on three input switch states.

5 Visualization (Michal)

In order to create a live visualization of the audio signals produced by our system, we decided to implement a short-time Fourier transform of the audio output signal and plot the magnitudes produced.

The short-time Fourier transform module took signed 24-bit samples of audio data as input and computed a 1024-sample FFT. This was accomplished with a Xilinx Fast Fourier Transform (FFT) IP module impilementing the Cooley-Tukey FFT algorithm, a computationally efficient method for calculating the discrete Fourier

transform.

As data arrived into the system, it was pushed into the FFT module until 1024 samples were stored, at which point a trigger signal was sent to the FFT module to execute the computation. The outputs of the FFT module were real and imaginary values returned in a single buffer, with real values occupying the first half of the logic and imaginary the second. To derive a spectrograph, we squared and summed these values. With the use of custom AXIS protocol code written by Joe Steinmeyer in conjunction with the Xilinx CORDIC IP module for square rooting values, we computed the magnitude of each frequency component. Sandwiched in between the splitting-and-summing and the CORDIC squaring was a Xilinx FIFO IP module to allow for breathing room and avoid glitches.

The resulting magnitude data was then, through an XVGA protocol module written by Joe Steinmeyer, mapped to amplitude values and then displayed on a monitor taking VGA signals. An example of this visual output is shown in Figure 7.

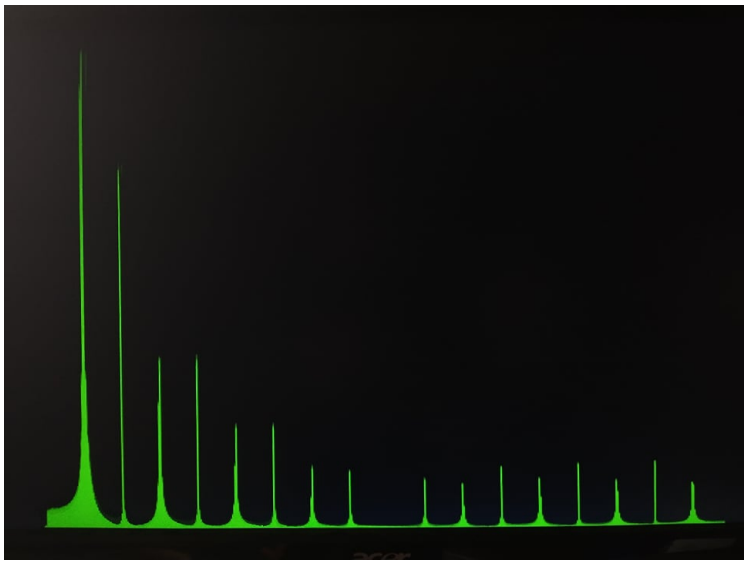


Figure 7: An example of the magnitude of a real-time STFT computed on the output audio data.

6 Reflection

The first stages of the development of this project were filled with great frustration because we attempted to integrate too many modules at once and could not figure out the root of a bug that produced very loud white noise as the system's output. This stemmed in part from the fact that some modules were based off of Verilog

starter code written by Digilent, and we did not understand their operation, even though we thought we did. As we struggled to debug the issue, we were forced to keep track of ready and valid signals more carefully, since the AXI and I2S protocols feature a huge array of these control signals, and it was initially very difficult to decide which signals to use as controls signals in our own modules because we did not adequately understand the protocols. Many of the first few weeks of the project were spent trying to decide when and where to pull the sampled data from the AXI modules to use in our own processing.

The bug turned out to be an issue with the processing of signed versus unsigned logics, since in some parts of the protocols the logics were declared as unsigned and in others they were interpreted as signed. Though simple to say, we learned the hard way that it is important to deeply understand modules written by others to keep track of both timing and functionality as systems become more complex. When debugging these issues, it also helps to isolate problems by changing the system incrementally and verifying functionality along the way, or writing testbenches for specific modules, of course. These practices can speed up synthesis and implementation drastically.

Throughout the project, we also became particularly familiar with utilization limits on the Nexys board, particularly when dealing with block RAM. More specifically, when developing the delay pedal, most of our initial implementations failed because the allocated buffers and BRAMs exceeded the memory capacity of the board, as we were working with the 48-bit data coming directly from the I2S module. To circumvent this issue, we eventually settled on a strategy that pulled the preliminary 24-bit data supplied before the sign-extend and multiply stages of the protocol, which allowed us to decrease our memory usage by half and consequently allowed us to create more delay with deeper BRAMs.

For future projects, or a redux of this project, we recommend that others be cognizant of utilization issues during the design and build phase, as opposed to coming up with solutions in the troubleshooting phase. Additionally, we are now more aware of the need to only store and process absolutely critical data. Zero-padded arrays, repeated data, and otherwise non-crucial bits can be forfeited or processed in a way as to reduce the take up of space.

We were also initially confused by bugs in the delay pedal that were due to the fact that the BRAM read latency was not one clock cycle, as we had originally thought. This led us to double-check the latency of all of our modules and keep them in mind as we designed modules around each other, and we would recommend that others do the same. We have truly come to terms with the importance of reading documentation. The documentation of the Xilinx IP cores is a treasure trove of information, allowing for efficient debugging or cross-referencing outputs to ensure that everything works well.

Appendix: SystemVerilog Code

```
`timescale 1ns / 1ps


module top #(
    parameter NUMBER_OF_SWITCHES = 4,
        parameter VOL_ROTARY_WIDTH  = 7,
        parameter MAG1_ROTARY_WIDTH = 8,
        parameter MAG2_ROTARY_WIDTH = 8,
        parameter MAG3_ROTARY_WIDTH = 8,
        parameter TREMOLO_ROTARY_WIDTH = 8,
        parameter RESET_POLARITY = 0
) (
    input wire          clk_100mhz,
    input wire [15:0]   sw,
    input wire          btnd,
    input wire          rx_data,


    input wire btnc,btnu,btnr,btnl,
    output logic[3:0] vga_r,
    output logic[3:0] vga_b,
    output logic[3:0] vga_g,
    output logic vga_hs,
    output logic vga_vs,
    output logic [15:0] led,


    //Magnitude1 control pins
    input wire A_1,
    input wire B_1,
    input wire sw_r_1,

    //Magnitude2 control pins
    input wire A_2,
    input wire B_2,
    input wire sw_r_2,

    //Magnitude3 control pins
    input wire A_3,
    input wire B_3,
    input wire sw_r_3,

    //Volume control pins
    input wire A_v,
    input wire B_v,
    input wire sw_r_v,

    output logic        tx_mclk,
    output logic        tx_lrck,
    output logic        tx_sclk,
    output logic        tx_data,
    output logic        rx_mclk,
    output logic        rx_lrck,
    output logic        rx_sclk
);

    logic axis_clk; // approx 22.591MHz

    logic [23:0] axis_tx_data;

    logic axis_tx_valid;
    logic axis_tx_ready;
```

```
    logic axis_tx_last;

    logic [23:0] axis_rx_data;
    logic axis_rx_valid;
    logic axis_rx_ready;
    logic axis_rx_last;


        logic resetn = (btnd == RESET_POLARITY) ? 1'b1 : 1'b0;

    clk_wiz_0 m_clk (
        .clk_in1(clk_100mhz),
        .clk_out1(axis_clk) // 22.591 MHz
    );


    logic [VOL_ROTARY_WIDTH-1:0] vol_counter;
    logic [MAG1_ROTARY_WIDTH-1:0] mag1_counter;
    logic [MAG2_ROTARY_WIDTH-1:0] mag2_counter;
    logic [MAG3_ROTARY_WIDTH-1:0] mag3_counter;



    rotary_decoder #(.ROTARY_WIDTH(VOL_ROTARY_WIDTH))
                    volume_control (.clk(clk_100mhz),
                                .rst(btnd),
                                .sw_r(sw_r_v),
                                .A(A_v),.B(B_v),
                                .counter(vol_counter)
                                );

    rotary_decoder #(.ROTARY_WIDTH(MAG1_ROTARY_WIDTH))
                    mag1_control (.clk(clk_100mhz),
                                .rst(btnd),
                                .sw_r(sw_r_1),
                                //flipped intentioallt
                                .A(B_1),.B(A_1),
                                .counter(mag1_counter)
                                );

    rotary_decoder #(.ROTARY_WIDTH(MAG2_ROTARY_WIDTH))
                    mag2_control (.clk(clk_100mhz),
                                .rst(btnd),
                                .sw_r(sw_r_2),
                                //intentionally flipped
                                .A(B_2),.B(A_2),
                                .counter(mag2_counter)
                                );

    rotary_decoder #(.ROTARY_WIDTH(MAG3_ROTARY_WIDTH))
                    mag3_control (.clk(clk_100mhz),
                                .rst(btnd),
                                .sw_r(sw_r_3),
                                .A(A_3),.B(B_3),
                                .counter(mag3_counter)
                                );



    axis_i2s2 m_i2s2 (
        .axis_clk(axis_clk),
        .axis_resetn(resetn),
```

```
        .tx_axis_s_data(axis_tx_data),
        .tx_axis_s_valid(axis_tx_valid),
        .tx_axis_s_ready(axis_tx_ready),
        .tx_axis_s_last(axis_tx_last),

        .rx_axis_m_data(axis_rx_data),
        .rx_axis_m_valid(axis_rx_valid),
        .rx_axis_m_ready(axis_rx_ready),
        .rx_axis_m_last(axis_rx_last),

        .tx_mclk(tx_mclk),
        .tx_lrck(tx_lrck),
        .tx_sclk(tx_sclk),
        .tx_sdout(tx_data),
        .rx_mclk(rx_mclk),
        .rx_lrck(rx_lrck),
        .rx_sclk(rx_sclk),
        .rx_sdin(rx_data)
    );

    audio_controller #(
        .SWITCH_WIDTH(NUMBER_OF_SWITCHES),

                .VOL_ROTARY_WIDTH(VOL_ROTARY_WIDTH),
                .MAG1_ROTARY_WIDTH(MAG1_ROTARY_WIDTH),
                .MAG2_ROTARY_WIDTH(MAG2_ROTARY_WIDTH),
                .MAG3_ROTARY_WIDTH(MAG3_ROTARY_WIDTH),
                .TREMOLO_ROTARY_WIDTH(TREMOLO_ROTARY_WIDTH),

                .DATA_WIDTH(24)
    ) m_vc (
        .clk_in(axis_clk),
        .clk_100mhz(clk_100mhz),
        .rst_in(btnd), // OR BTND?
        .all_sw(sw[15:0]),
        .pedal_sw_in(sw[15:7]),
        .vol_sw_in(sw[3:0]), // for volume control

        .s_axis_data(axis_rx_data),
        .s_axis_valid(axis_rx_valid),
        .s_axis_ready(axis_rx_ready),
        .s_axis_last(axis_rx_last),

        .m_axis_data(axis_tx_data),
        .m_axis_valid(axis_tx_valid),
        .m_axis_ready(axis_tx_ready),
        .m_axis_last(axis_tx_last),

        .vol_val(vol_counter),
        .mag1_val(mag1_counter),
        .mag2_val(mag2_counter),
        .mag3_val(mag3_counter),

        .btnc(btnc),
        .btnu(btnu),
        .btnd(btnd),
        .btnr(btnr),
        .btnl(btnl),
        .vga_r(vga_r),
        .vga_b(vga_b),
        .vga_g(vga_g),
```

```
        .vga_hs(vga_hs),
        .vga_vs(vga_vs),
        .led(led)


    );
endmodule
```

```
`timescale 1ns / 1ps
`default_nettype none

//////////////////////////////////////////////////////////////////////////////////
// Company: Digilent
// Engineer: Arthur Brown
//
// Create Date: 03/23/2018 01:23:15 PM
// Module Name: axis_i2s2
// Description: AXI-Stream I2S controller intended for use with Pmod I2S2.
//              Generates clocks and select signals required to place each of the ICs on the
Pmod I2S2 into slave mode.
//              Data is 24-bit, left aligned, shifted one serial clock right from the LRCK
boundaries.
//              This module only supports 44.1KHz sample rate, and expects the frequency of
axis_clk to be approx 22.591MHz.
//              At the end of each I2S frame, a 2-word packet is made available on the AXIS
master interface. Further packets will be discarded
//              until the current packet is accepted by an AXIS slave.
//              Whenever a 2-word packet is received on the AXIS slave interface, it is
transmitted over the I2S interface on the next frame.
//              Each packet consists of two 3-byte words, starting with left audio channel
data, followed by right channel data.
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module axis_i2s2 (
    input  wire         axis_clk, // require: approx 22.591MHz
    input  wire         axis_resetn,
    input  wire [31:0]  tx_axis_s_data,
    input  wire         tx_axis_s_valid,
    input  wire         tx_axis_s_last,
    input  wire         rx_axis_m_ready,
    input  wire         rx_sdin,  // JA[7]

    output logic        tx_axis_s_ready = 1'b0,
    output logic [31:0] rx_axis_m_data,
    output logic        rx_axis_m_valid = 1'b0,
    output logic        rx_axis_m_last = 1'b0,
    output logic        tx_mclk,   // JA[0]
    output logic        tx_lrck,   // JA[1]
    output logic        tx_sclk,   // JA[2]
    output logic        tx_sdout,  // JA[3]
    output logic        rx_mclk,   // JA[4]
    output logic        rx_lrck,   // JA[5]
    output logic        rx_sclk    // JA[6]
);

    logic [8:0] count = 9'd0;
    localparam EOF_COUNT = 9'd455; // end of full I2S frame

    always_ff @(posedge axis_clk) begin
        count <= count + 1;
    end

    logic lrck;
    logic sclk;
    logic mclk;
```

```
    assign lrck = count[8];
    assign sclk = count[2];
    assign mclk = axis_clk;

    assign tx_lrck = lrck;
    assign tx_sclk = sclk;
    assign tx_mclk = mclk;
    assign rx_lrck = lrck;
    assign rx_sclk = sclk;
    assign rx_mclk = mclk;

    /* AXIS SLAVE CONTROLLER */
    logic [31:0] tx_data_l = 0;
    logic [31:0] tx_data_r = 0;

    always_ff @(posedge axis_clk) begin
        if (axis_resetn == 1'b0) begin
            tx_axis_s_ready <= 1'b0;
        end else if (tx_axis_s_ready == 1'b1 && tx_axis_s_valid == 1'b1 && tx_axis_s_last ==
1'b1) begin // end of packet, cannot accept data until current one has been transmitted
            tx_axis_s_ready <= 1'b0;
        end else if (count == 9'b0) begin // beginning of I2S frame, in order to avoid
tearing, cannot accept data until frame complete
            tx_axis_s_ready <= 1'b0;
        end else if (count == EOF_COUNT) begin// end of I2S frame, can accept data
            tx_axis_s_ready <= 1'b1;
        end
    end

    always_ff @(posedge axis_clk) begin
        if (axis_resetn == 1'b0) begin
            tx_data_r <= 32'b0;
            tx_data_l <= 32'b0;
        end else if (tx_axis_s_valid == 1'b1 && tx_axis_s_ready == 1'b1) begin
            if (tx_axis_s_last == 1'b1) begin
                tx_data_r <= tx_axis_s_data;
            end else begin
                tx_data_l <= tx_axis_s_data;
            end
        end
    end

    /* I2S TRANSMIT SHIFT REGISTERS */
    logic [23:0] tx_data_l_shift = 24'b0;
    logic [23:0] tx_data_r_shift = 24'b0;

    always_ff @(posedge axis_clk) begin
        if (count == 3'b000000111) begin
            tx_data_l_shift <= tx_data_l[23:0];
            tx_data_r_shift <= tx_data_r[23:0];
        end else if (count[2:0] == 3'b111 && count[7:3] >= 5'd1 && count[7:3] <= 5'd24) begin
            if (count[8] == 1'b1) begin
                tx_data_r_shift <= {tx_data_r_shift[22:0], 1'b0};
            end else begin
                tx_data_l_shift <= {tx_data_l_shift[22:0], 1'b0};
            end
        end
    end

    always_ff @(count, tx_data_l_shift, tx_data_r_shift) begin
        if (count[7:3] <= 5'd24 && count[7:3] >= 4'd1) begin
```

```
            if (count[8] == 1'b1) begin
                tx_sdout = tx_data_r_shift[23];
            end else begin
                tx_sdout = tx_data_l_shift[23];
            end
        end else begin
            tx_sdout = 1'b0;
        end
    end

    /* SYNCHRONIZE DATA IN TO AXIS CLOCK DOMAIN */
    logic [2:0] din_sync_shift = 3'd0;
    logic din_sync = din_sync_shift[2];
    always_ff @(posedge axis_clk) begin
        din_sync_shift <= {din_sync_shift[1:0], rx_sdin};
    end

    /* I2S RECEIVE SHIFT REGISTERS */
    logic [23:0] rx_data_l_shift = 24'b0;
    logic [23:0] rx_data_r_shift = 24'b0;
    always_ff @(posedge axis_clk) begin
        if (count[2:0] == 3'b011 && count[7:3] <= 5'd24 && count[7:3] >= 5'd1) begin
            if (lrck == 1'b1) begin
                rx_data_r_shift <= {rx_data_r_shift, din_sync};
            end else begin
                rx_data_l_shift <= {rx_data_l_shift, din_sync};
            end
        end
    end

    /* AXIS MASTER CONTROLLER */
    logic [31:0] rx_data_l = 32'b0;
    logic [31:0] rx_data_r = 32'b0;

    always_ff @(posedge axis_clk) begin
        if (axis_resetn == 1'b0) begin
            rx_data_l <= 32'b0;
            rx_data_r <= 32'b0;
        end else if (count == EOF_COUNT && rx_axis_m_valid == 1'b0) begin
            rx_data_l <= {8'b0, rx_data_l_shift};
            rx_data_r <= {8'b0, rx_data_r_shift};
        end
    end

    assign rx_axis_m_data = (rx_axis_m_last == 1'b1) ? rx_data_r : rx_data_l;

    always_ff @(posedge axis_clk) begin
        if (axis_resetn == 1'b0) begin
            rx_axis_m_valid <= 1'b0;
        end else if (count == EOF_COUNT && rx_axis_m_valid == 1'b0) begin
            rx_axis_m_valid <= 1'b1;
        end else if (rx_axis_m_valid == 1'b1 && rx_axis_m_ready == 1'b1 && rx_axis_m_last ==
1'b1) begin
            rx_axis_m_valid <= 1'b0;
        end
    end

    always_ff @(posedge axis_clk) begin
        if (axis_resetn == 1'b0) begin
            rx_axis_m_last <= 1'b0;
        end else if (count == EOF_COUNT && rx_axis_m_valid == 1'b0) begin
            rx_axis_m_last <= 1'b0;
```

```
        end else if (rx_axis_m_valid == 1'b1 && rx_axis_m_ready == 1'b1) begin
            rx_axis_m_last <= ~rx_axis_m_last;
        end
    end

endmodule

`default_nettype wire
```

```
`timescale 1ns / 1ps
`default_nettype none
////////////////////////////////////////////////////////////////////////////////
// adapted from Digilent Verilog code by Arthur Brown
// AXI-Stream audio controller intended for use with AXI Stream Pmod I2S2 controller.
// Whenever a 2-word packet is received on the slave interface, it is tranformed
// by the pedal modules, then sent over the master interface.
// Reception of data on the slave interface is halted while processing and
// transfer is taking place.
////////////////////////////////////////////////////////////////////////////////

module audio_controller #(
    parameter SWITCH_WIDTH = 4,
        parameter VOL_ROTARY_WIDTH = 8,
        parameter MAG1_ROTARY_WIDTH = 8,
        parameter MAG2_ROTARY_WIDTH = 8,
        parameter MAG3_ROTARY_WIDTH = 8,
        parameter TREMOLO_ROTARY_WIDTH = 8,
    parameter DATA_WIDTH = 24
) (
    input wire clk_in,
    input wire clk_100mhz,
    input wire rst_in,
    input wire [15:0] all_sw,
    input wire [8:0] pedal_sw_in,
    input wire [SWITCH_WIDTH:0] vol_sw_in,

    input wire btnc,btnu,btnd,btnr,btnl,
    output logic[3:0] vga_r,
    output logic[3:0] vga_b,
    output logic[3:0] vga_g,
    output logic vga_hs,
    output logic vga_vs,
    output logic [15:0] led,

    //AXIS SLAVE INTERFACE
    input  wire [DATA_WIDTH-1:0] s_axis_data,
    input  wire     s_axis_valid,
    output logic    s_axis_ready = 1'b1,
    input  wire     s_axis_last,

    // AXIS MASTER INTERFACE
    output logic [DATA_WIDTH-1:0] m_axis_data = 1'b0,
    output logic    m_axis_valid = 1'b0,
    input  wire     m_axis_ready,
    output logic    m_axis_last = 1'b0,

    // VOLUME AND MAGNITUDE CONTROL
    input wire [VOL_ROTARY_WIDTH-1:0] vol_val,
    input wire [MAG1_ROTARY_WIDTH-1:0] mag1_val,
    input wire [MAG2_ROTARY_WIDTH-1:0] mag2_val,
    input wire [MAG3_ROTARY_WIDTH-1:0] mag3_val




);
    localparam MULTIPLIER_WIDTH = 24;

    logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0] data [1:0];
    logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0] pedal_data [1:0];
    logic [1:0] pedal_valid;
```

```
    logic [SWITCH_WIDTH-1:0] sw_sync_r [2:0];
    logic [SWITCH_WIDTH-1:0] sw_sync = sw_sync_r[2];

    logic [SWITCH_WIDTH:0] m = {1'b0, sw_sync} + 1;
    logic [MULTIPLIER_WIDTH:0] multiplier = 'b0; // range of 0x00:0x10 for width=4

    logic m_select = m_axis_last;
    logic m_new_word = (m_axis_valid == 1'b1 && m_axis_ready == 1'b1) ? 1'b1 : 1'b0;
    logic m_new_packet = (m_new_word == 1'b1 && m_axis_last == 1'b1) ? 1'b1 : 1'b0;

    logic s_select = s_axis_last;
    logic s_new_word = (s_axis_valid == 1'b1 && s_axis_ready == 1'b1) ? 1'b1 : 1'b0;
    logic s_new_packet = (s_new_word == 1'b1 && s_axis_last == 1'b1) ? 1'b1 : 1'b0;
    logic s_new_packet_r = 1'b0;

    // PEDAL CHOICE MUX
    localparam CLEAN        = 9'b000000000;
        localparam DELAY        = 9'b100000000;
        localparam DISTORTION   = 9'b010000000;
        localparam OVERDRIVE    = 9'b001000000;
        localparam WAHWAH       = 9'b000100000;
        localparam LOWPASS      = 9'b000010000;
        localparam TREMOLO      = 9'b000001000;
        localparam HARMONIZER   = 9'b000000100;
        localparam OVERDELAY    = 9'b000000010;
        localparam OVERTREMOLO  = 9'b000000001;
                //      localparam HIGHPASS     = 9'b0000110;

        logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0]    pedal_data_delay [1:0];
        logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0]    pedal_data_distortion [1:0];
        logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0]    pedal_data_overdrive [1:0];
        logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0]    pedal_data_wahwah [1:0];
        logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0]    pedal_data_lowpass [1:0];
//      logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0]    pedal_data_highpass [1:0];
        logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0]    pedal_data_tremolo [1:0];
        logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0]    pedal_data_harmonizer [1:0];
        logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0]    pedal_data_overdelay [1:0];
        logic [MULTIPLIER_WIDTH+DATA_WIDTH-1:0]    pedal_data_overtrem [1:0];


        logic layering_switches = all_sw[5:4];
        logic [1:0] pedal_valid_overdrive;
        logic  pedal_valid_delay;
        logic  pedal_valid_overdelay;
        logic [1:0] pedal_valid_distortion;
        logic [1:0] pedal_valid_wahwah;
        logic [1:0] pedal_valid_lowpass;
//      logic [1:0] pedal_valid_highpass;
        logic [1:0] pedal_valid_tremolo;
        logic [1:0] pedal_valid_overtrem;

        logic  pedal_valid_harmonizer;

    always_comb begin
        case(pedal_sw_in)
            CLEAN:      pedal_data = data;
            DELAY:      pedal_data = pedal_data_delay;
            DISTORTION: pedal_data = pedal_data_distortion;
            OVERDRIVE:  pedal_data = pedal_data_overdrive;
            WAHWAH:  pedal_data = pedal_data_wahwah;
            LOWPASS:  pedal_data = pedal_data_lowpass;
```

```
//            HIGHPASS:  pedal_data = pedal_data_highpass;
            TREMOLO:  pedal_data = pedal_data_tremolo;
            HARMONIZER: pedal_data = pedal_data_harmonizer;
            OVERTREMOLO: pedal_data = pedal_data_overtrem;
        endcase
    end // always_comb



    // DELAY PEDAL INSTANTIATION
    logic [DATA_WIDTH-1:0] small_data [1:0];
    logic [DATA_WIDTH-1:0] small_pedal_data_delay_0;
    logic [DATA_WIDTH-1:0] small_pedal_data_delay_1;
    assign pedal_data_delay[0] = {{MULTIPLIER_WIDTH{small_pedal_data_delay_0[DATA_WIDTH-1]}},
small_pedal_data_delay_0};
    assign pedal_data_delay[1] = {{MULTIPLIER_WIDTH{small_pedal_data_delay_1[DATA_WIDTH-1]}},
small_pedal_data_delay_1};

    logic [DATA_WIDTH-1:0] small_pedal_data_tremolo_0;
    logic [DATA_WIDTH-1:0] small_pedal_data_tremolo_1;
    assign pedal_data_tremolo[0] =
{{MULTIPLIER_WIDTH{small_pedal_data_tremolo_0[DATA_WIDTH-1]}}, small_pedal_data_tremolo_0};
    assign pedal_data_tremolo[1] =
{{MULTIPLIER_WIDTH{small_pedal_data_tremolo_1[DATA_WIDTH-1]}}, small_pedal_data_tremolo_1};



    delay_pedal #(.DELAY_ROTARY_WIDTH(MAG3_ROTARY_WIDTH)) delay_pedal_01 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag3_val),
        .small_guitar_in_0(small_data[0]),
        .small_guitar_in_1(small_data[1]),
        .small_guitar_out_0(small_pedal_data_delay_0),
        .small_guitar_out_1(small_pedal_data_delay_1),
        .valid_out(pedal_valid_delay)
        );


    overdrive_pedal #(.OD_ROTARY_WIDTH(MAG1_ROTARY_WIDTH)) pedal0 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag1_val),
        .guitar_in($signed(data[0])),
        .guitar_out(pedal_data_overdrive[0]),
        .valid_out(pedal_valid_overdrive[0])
    );
    overdrive_pedal #(.OD_ROTARY_WIDTH(MAG1_ROTARY_WIDTH)) pedal1 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag1_val),
        .guitar_in($signed(data[1])),
        .guitar_out(pedal_data_overdrive[1]),
        .valid_out(pedal_valid_overdrive[1])
    );

    // OVERTREMOLO

     overdrive_pedal #(.OD_ROTARY_WIDTH(MAG2_ROTARY_WIDTH)) overtrem_0 (
```

```
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag2_val),
        .guitar_in($signed(pedal_data_tremolo[0])),
        .guitar_out(pedal_data_overtrem[0]),
        .valid_out(pedal_valid_overtrem[0])
    );
    overdrive_pedal #(.OD_ROTARY_WIDTH(MAG2_ROTARY_WIDTH)) overtrem_1 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag2_val),
        .guitar_in($signed(pedal_data_tremolo[1])),
        .guitar_out(pedal_data_overtrem[1]),
        .valid_out(pedal_valid_overtrem[1])
    );


  // DISTORTION PEDAL INSTANTIATION
    distortion_pedal #(.DIST_ROTARY_WIDTH(MAG2_ROTARY_WIDTH)) distortion_pedal_0 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag2_val),
        .guitar_in($signed(data[0])),
        .guitar_out(pedal_data_distortion[0]),
        .valid_out(pedal_valid_distortion[0])
    );
    distortion_pedal #(.DIST_ROTARY_WIDTH(MAG2_ROTARY_WIDTH)) distortion_pedal_1 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag2_val),
        .guitar_in($signed(data[1])),
        .guitar_out(pedal_data_distortion[1]),
        .valid_out(pedal_valid_distortion[1])
    );


  // WAHWAH PEDAL INSTANTIATION

    wahwah_pedal_0 #(.WAH_ROTARY_WIDTH(MAG2_ROTARY_WIDTH)) wah_pedal_0 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag2_val),
        .guitar_in($signed(data[0])),
        .guitar_out(pedal_data_wahwah[0]),
        .valid_out(pedal_valid_wahwah[0])
    );
    wahwah_pedal_1 #(.WAH_ROTARY_WIDTH(MAG2_ROTARY_WIDTH)) wah_pedal_1 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag2_val),
        .guitar_in($signed(data[1])),
        .guitar_out(pedal_data_wahwah[1]),
        .valid_out(pedal_valid_wahwah[1])
    );

  // LOWPASS PEDAL INSTANTIATION
```

```
    lowpass_pedal_0  lowpass_0 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .guitar_in($signed(data[0])),
        .guitar_out(pedal_data_lowpass[0]),
        .valid_out(pedal_valid_lowpass[0])
    );
    lowpass_pedal_1  lowpass_1 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .guitar_in($signed(data[1])),
        .guitar_out(pedal_data_lowpass[1]),
        .valid_out(pedal_valid_lowpass[1])
    );



    // TREMOLO PEDAL INSTANTIATION
    tremolo_pedal #(.TREMOLO_ROTARY_WIDTH(MAG1_ROTARY_WIDTH)) tremolo_pedal_0 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag1_val),
        .small_guitar_in($signed(small_data[0])),
        .small_guitar_out(small_pedal_data_tremolo_0),
        .valid_out(pedal_valid_tremolo[0])
    );
    tremolo_pedal #(.TREMOLO_ROTARY_WIDTH(MAG1_ROTARY_WIDTH)) tremolo_pedal_1 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .mag_in(mag1_val),
        .small_guitar_in($signed(small_data[1])),
        .small_guitar_out(small_pedal_data_tremolo_1),
        .valid_out(pedal_valid_tremolo[1])
    );


    logic [DATA_WIDTH-1:0] small_pedal_data_harmonizer_0;
    logic [DATA_WIDTH-1:0] small_pedal_data_harmonizer_1;
    assign pedal_data_harmonizer[0] =
{{MULTIPLIER_WIDTH{small_pedal_data_harmonizer_0[DATA_WIDTH-1]}},
small_pedal_data_harmonizer_0};
    assign pedal_data_harmonizer[1] =
{{MULTIPLIER_WIDTH{small_pedal_data_harmonizer_1[DATA_WIDTH-1]}},
small_pedal_data_harmonizer_1};


  // HARMONIZER INSTANTIATION

     harmonizer harmonizer_pedal_0 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .valid_in(s_new_packet_r),
        .data_in_0($signed(small_data[0])),
        .data_in_1($signed(small_data[1])),
        .delta_opt_in(all_sw[3:0]),
        .data_out_0(small_pedal_data_harmonizer_0),
        .data_out_1(small_pedal_data_harmonizer_1),
```

```
        .valid_out(pedal_valid_harmonizer)
    );



    stft_joe my_joe_stft (.clk_100mhz(clk_100mhz),
                  .clk_in(clk_in),
                  .sw(all_sw),
                  .btnc(btnc),
                  .btnu(btnu),
                  .btnd(btnd),
                  .btnr(btnr),
                  .btnl(btnl),
                  .vga_r(vga_r),
                  .vga_b(vga_b),
                  .vga_g(vga_g),
                  .vga_hs(vga_hs),
                  .vga_vs(vga_vs),
                  .led(led),
                  .sample_trigger(s_new_word), //unsure for what this should be
                  .guitar_in(small_data[0])
                  );









    //
----------------------------------------------------------------------------------------------------

    always_ff @(posedge clk_in) begin
        sw_sync_r[2] <= sw_sync_r[1];
        sw_sync_r[1] <= sw_sync_r[0];
        sw_sync_r[0] <= vol_sw_in;

        multiplier <= {vol_val,{MULTIPLIER_WIDTH{1'b0}}} / {VOL_ROTARY_WIDTH{1'b1}};

        //How to make a multiplier
        //multiplier <= {value you want,{MULTIPLIER_WIDTH{1'b0}}} / {value length{1'b1}};
        s_new_packet_r <= s_new_packet;
    end

    always_ff @(posedge clk_in) begin
        if (s_new_word == 1'b1) begin // sign extend and register AXIS slave data
            data[s_select] <= {{MULTIPLIER_WIDTH{s_axis_data[DATA_WIDTH-1]}}, s_axis_data};
            small_data[s_select] <= s_axis_data; // added this
        end else if (s_new_packet_r == 1'b1) begin

            // ORIGINAL VERSION
//            data[0] <= $signed(data[0]) * multiplier; // core volume control algorithm,
infers a DSP48 slice
//            data[1] <= $signed(data[1]) * multiplier;

            data[0] <= $signed(pedal_data[0]) * multiplier;
            data[1] <= $signed(pedal_data[1]) * multiplier;
        end
    end

    always_ff @(posedge clk_in) begin
```

```
        if (s_new_packet_r == 1'b1) begin
            m_axis_valid <= 1'b1;
        end else if (m_new_packet == 1'b1) begin
            m_axis_valid <= 1'b0;
        end
    end

    always_ff @(posedge clk_in) begin
        if (m_new_packet == 1'b1) begin
            m_axis_last <= 1'b0;
        end else if (m_new_word == 1'b1) begin
            m_axis_last <= 1'b1;
        end
    end

    always_comb begin
        if (m_axis_valid == 1'b1) begin
            m_axis_data = data[m_select][MULTIPLIER_WIDTH+DATA_WIDTH-1:MULTIPLIER_WIDTH];
        end else begin
            m_axis_data = 'b0;
        end
    end

    always_ff @(posedge clk_in) begin
        if (s_new_packet == 1'b1) begin
            s_axis_ready <= 1'b0;
        end else if (m_new_packet == 1'b1) begin
            s_axis_ready <= 1'b1;
        end
    end

endmodule // audio_controller
```

```
`timescale 1ns / 1ps

module rotary_decoder #( parameter ROTARY_WIDTH )
                       (
                         input wire clk,
                         input wire rst,
                         input wire sw_r, //jd[0] i.e. jd[1]
                         input wire B, //jd[1] i.e. jd[2]
                         input wire A, //jd[2] i.e. jd[3]
                         output logic [ROTARY_WIDTH-1:0] counter
                       );

    // Unless you're spinning your knob like a mad man, your clock is oversampling the rotary
signal
    // Need to check for the rising edge of each signal


    logic [2:0] A_buffer;
    logic [2:0] B_buffer;
    //To prevent metastability (since this is a werid system)
    //Use two registers are suggested in lecture
    //Remember with concatanation, the sequence is preserved
    //AABB = {AA,BB}
    // Each buffer contains 3 piece of data essentially
    // The LSB is the current A and B signal
    // The next bit is the "old" A
    always_ff @(posedge clk) begin
        A_buffer <= {A_buffer[1:0],A};
        B_buffer <= {B_buffer[1:0],B};
    end //always_ff end

    logic edge_detection;
    logic direction;
    always_comb begin
        edge_detection = A_buffer[1] ^ A_buffer[2] ^ B_buffer[1] ^ B_buffer[2];
        direction = A_buffer[1] ^ B_buffer[2];
        // direction == 1 is clockwise
        // direction == 0 is counter-clockwise
    end // always_comb end

    always_ff @(posedge clk)begin
        if (rst) begin
            counter <= (2**ROTARY_WIDTH - 1)/2 ;
        end
        if (edge_detection == 1) begin
            if (direction == 1) begin //clockwise
                if (counter  == (2**ROTARY_WIDTH)-1) begin
                    counter <= counter;
                end else begin
                    counter <= counter + 1;
                end
            end else begin
                if (counter == 0) begin
                    counter <= counter;
                end else begin
                    counter <= counter - 1;
                end
            end
       end
   end //alway_ff end

endmodule
```

```
module delay_pedal #(
    parameter DELAY_ROTARY_WIDTH =8
    ) (
    input wire  clk_in,
    input wire  rst_in,
    input wire  valid_in,
    input wire  [DELAY_ROTARY_WIDTH-1:0] mag_in,
    input wire signed [23:0] small_guitar_in_0,
    input wire signed [23:0] small_guitar_in_1,
    output logic [23:0] small_guitar_out_0,
    output logic [23:0] small_guitar_out_1,
    output logic valid_out
    );

    logic [15:0] scaled1 = (mag_in* 8'd20);
    logic [15:0] scaled2 = (mag_in* 8'd40);
    logic [15:0] scaled3 = (mag_in* 8'd60);

    logic [15:0] DELAY1  = 16'd2000 + scaled1;
    logic [15:0] DELAY2  = 16'd2000 + scaled2;
    logic [15:0] DELAY3  = 16'd2000 + scaled3;

    //OLD CODE USES PARAMETERS
//    parameter DELAY1 = 5000;
//    parameter DELAY2 = 10000;
//    parameter DELAY3 = 15000;

    logic [15:0] data_addr;     // read/write addr for bram
    logic [15:0] delay1_addr;   // read addr for bram
    logic [15:0] delay2_addr;   // read addr for bram
    logic [15:0] delay3_addr;   // read addr for bram
    logic [15:0] curr_addr;     // write addr for bram


    logic [15:0] valid_r;

    logic data0_wea;
    logic data1_wea;

    logic signed [23:0] data_to_bram_0;
    logic signed [23:0] data_from_bram_0;
    logic signed [23:0] data_to_bram_1;
    logic signed [23:0] data_from_bram_1;

    logic signed [23:0] guitar_1_delay1;
    logic signed [23:0] guitar_0_delay1;
    logic signed [23:0] guitar_1_delay2;
    logic signed [23:0] guitar_0_delay2;
    logic signed [23:0] guitar_1_delay3;
    logic signed [23:0] guitar_0_delay3;

    // BRAM SPECS: write width 24, depth 65536
    blk_mem_gen_0 data0_bram (
        .addra(data_addr),
        .clka(clk_in),
        .dina(data_to_bram_0),
        .douta(data_from_bram_0),
        .ena(1),
        .wea(data0_wea)
        );

    blk_mem_gen_1 data1_bram(
```

```
        .addra(data_addr),
        .clka(clk_in),
        .dina(data_to_bram_1),
        .douta(data_from_bram_1),
        .ena(1),
        .wea(data1_wea)
        );

    assign delay1_addr = curr_addr - DELAY1;
    assign delay2_addr = curr_addr - DELAY2;
    assign delay3_addr = curr_addr - DELAY3;

    assign data_to_bram_0 = small_guitar_in_0;
    assign data_to_bram_1 = small_guitar_in_1;

    always_ff @(posedge clk_in) begin
        valid_r <= {valid_r[14:0], valid_in};

        if (valid_r[0] && ~(valid_r[1])) begin         // write
            data1_wea <= 1'b1;
            data0_wea <= 1'b1;
            data_addr <= curr_addr;
            curr_addr <= curr_addr + 1;
        end else if (valid_r[3] && ~(valid_r[4])) begin
            data1_wea <= 1'b0;
            data0_wea <= 1'b0;
            data_addr <= delay1_addr;
            guitar_0_delay1 <= (data_from_bram_0);
            guitar_1_delay1 <= (data_from_bram_1);
        end else if (valid_r[6] && ~(valid_r[7])) begin
            data1_wea <= 1'b0;
            data0_wea <= 1'b0;
            data_addr <= delay2_addr;
            guitar_0_delay2 <= (data_from_bram_0);
            guitar_1_delay2 <= (data_from_bram_1);
            guitar_0_delay1 <= guitar_0_delay1 >>> 1;
            guitar_1_delay1 <= guitar_1_delay1 >>> 1;
        end else if (valid_r[9] && ~(valid_r[10])) begin
            data1_wea <= 1'b0;
            data0_wea <= 1'b0;
            data_addr <= delay3_addr;
            guitar_0_delay3 <= (data_from_bram_0);
            guitar_1_delay3 <= (data_from_bram_1);
            guitar_0_delay2 <= guitar_0_delay2 >>> 2;
            guitar_1_delay2 <= guitar_1_delay2 >>> 2;
        end else if (valid_r[12] && ~(valid_r[13])) begin
            guitar_0_delay3 <= guitar_0_delay3 >>> 3;
            guitar_1_delay3 <= guitar_1_delay3 >>> 3;
        end else if (valid_r[15]) begin
            small_guitar_out_0 <= small_guitar_in_0 + guitar_0_delay1 + guitar_0_delay2 +
guitar_0_delay3;
            small_guitar_out_1 <= small_guitar_in_1 + guitar_1_delay1 + guitar_1_delay2 +
guitar_1_delay3;
        end

   end


endmodule
```

```
`timescale 1ns / 1ps

module distortion_pedal #(
    parameter DIST_ROTARY_WIDTH = 8
) (
    input wire  clk_in,
    input wire  rst_in,
    input wire  valid_in,
    input wire [DIST_ROTARY_WIDTH-1:0] mag_in,   /// NEEDS A SIZE
    input wire signed [47:0]    guitar_in,
    output logic [47:0]         guitar_out,
    output logic valid_out
);

    parameter DATA_WIDTH = 24;
    parameter THRESHOLD = 100000900;
    parameter N_REGS = 3;


    logic signed [47:0] top_threshold = $signed(mag_in*40'd5000);  //apparently this sign
extneds?
    logic signed [47:0] bottom_threshold = $signed(-mag_in*40'd5000);

    logic [7:0] gain;
    logic [3:0] count;
    logic signed [31:0] boosted;

    logic signed [47:0] guitar_regs [N_REGS-1:0]; // N_REGS-element array each 8 bits wide
    logic [N_REGS-1:0] valid_regs;

//    assign guitar_out = guitar_regs[N_REGS-1];
    assign valid_out  = valid_regs[N_REGS-1];
    assign gain = 8'd1;


      always_comb begin
        if ($signed(guitar_in) > top_threshold) begin
            guitar_out = top_threshold + ($signed(guitar_in) >>> 2);
        end else if ($signed(guitar_in) < bottom_threshold) begin
            guitar_out = bottom_threshold + ($signed(guitar_in) >>> 2);
        end else begin
            guitar_out = guitar_in;
        end
     end

endmodule // distortion_pedal
```

```
module overdrive_pedal #(
    parameter OD_ROTARY_WIDTH = 8
    ) (
    input wire  clk_in,
    input wire  rst_in,
    input wire  valid_in,
    input wire [OD_ROTARY_WIDTH-1:0] mag_in,
    input wire signed [47:0]    guitar_in,
    output logic [47:0]         guitar_out,
    output logic valid_out
);

    parameter DATA_WIDTH = 24;
    parameter THRESHOLD = 100000900;
    parameter N_REGS = 3;

    //Needs to be fiddled around with of course
    logic signed [47:0] top_threshold = $signed(mag_in*40'd5000);  //apparently this sign
extneds?
    logic signed [47:0] bottom_threshold = $signed(-mag_in*40'd5000);
    logic [7:0] gain;
    logic [3:0] count;
    logic signed [31:0] boosted;

    logic signed [47:0] guitar_regs [N_REGS-1:0]; // N_REGS-element array each 8 bits wide
    logic [N_REGS-1:0] valid_regs;

//    assign guitar_out = guitar_regs[N_REGS-1];
    assign valid_out  = valid_regs[N_REGS-1];
    assign gain = 8'd1;


    always_comb begin
        if ($signed(guitar_in) > top_threshold) begin
            guitar_out = top_threshold;
        end else if ($signed(guitar_in) < bottom_threshold) begin
            guitar_out = bottom_threshold;
        end else begin
            guitar_out = guitar_in;
        end
     end


endmodule // overdrive_pedal
```

```
`timescale 1ns / 1ps

  module lowpass_pedal_0 (
    input wire  clk_in,
    input wire  rst_in,
    input wire  valid_in,
    input wire signed [47:0]    guitar_in,
    input wire ready_in,
    output logic [47:0]         guitar_out,
    output logic valid_out
);


 assign guitar_out = lowpass_out[71:16] >>> 3;
 logic [71:0] lowpass_out;
 logic ready_out;
 logic valid_out;


lowpass_0 lowpass_0_instance (
  .aclk(clk_in),                                  // input wire aclk
  .s_axis_data_tvalid(valid_in),  // input wire s_axis_data_tvalid
  .s_axis_data_tready(ready_out),  // output wire s_axis_data_tready
  .s_axis_data_tdata($signed(guitar_in)),    // input wire [47 : 0] s_axis_data_tdata
  .m_axis_data_tvalid(valid_out),  // output wire m_axis_data_tvalid
  .m_axis_data_tdata( lowpass_out)    // output wire [71 : 0] m_axis_data_tdata
);


endmodule
```

```
`timescale 1ns / 1ps

 module wahwah_pedal_0 #(
    parameter WAH_ROTARY_WIDTH = 8
  ) (
    input wire  clk_in,
    input wire  rst_in,
    input wire  valid_in,
    input wire [WAH_ROTARY_WIDTH-1:0] mag_in,   /// NEEDS A SIZE
    input wire signed [47:0]    guitar_in,
    input wire ready_in,
    output logic [47:0]         guitar_out,
    output logic valid_out
);

always_comb begin
//    guitar_out = guitar_in;
//    guitar_out = wah_1_out[63:16];
//    if (mag_in >= 8'd200) begin
//        guitar_out = guitar_in;
//    end else if (mag_in >= 8'd150 && mag_in <= 8'd199) begin
//        guitar_out = guitar_in;
//    end else if (mag_in >= 8'd100 && mag_in <= 8'd149) begin
//        guitar_out = wah_3_out[71:16];
//    end else if (mag_in >= 8'd50 && mag_in <= 8'd99) begin
//        guitar_out = wah_2_out[71:16];
//    end else if (mag_in >= 8'd0 && mag_in <= 8'd49) begin
//        guitar_out = wah_1_out[63:16];
//    end
//end


    if (mag_in >= 8'd224) begin
        guitar_out = wah_8_out[71:16];
    end else if (mag_in >= 8'd192 && mag_in <= 8'd223) begin
        guitar_out = wah_7_out[71:16];
    end else if (mag_in >= 8'd160 && mag_in <= 8'd191) begin
        guitar_out = wah_6_out[71:16];
    end else if (mag_in >= 8'd128 && mag_in <= 8'd159) begin
        guitar_out = wah_5_out[71:16];
    end else if (mag_in >= 8'd96 && mag_in <= 8'd127) begin
        guitar_out = wah_4_out[71:16];
    end else if (mag_in >= 8'd64 && mag_in <= 8'd95) begin
        guitar_out = wah_3_out[71:16];
    end else if (mag_in >= 8'd32 && mag_in <= 8'd63) begin
        guitar_out = wah_2_out[63:16];
    end else if (mag_in >= 8'd0 && mag_in <= 8'd31) begin
        guitar_out = wah_1_out[71:16];
    end
end



logic ready_out;
logic [71:0] wah_1_out;
logic [63:0] wah_2_out;
logic [71:0] wah_3_out;
logic [71:0] wah_4_out;
logic [71:0] wah_5_out;
logic [71:0] wah_6_out;
logic [71:0] wah_7_out;
logic [71:0] wah_8_out;
```

```
// assign guitar_out = fir_data_out[47:0];
// r_compiler_1 your_instance_name_1 (
//  .aclk(clk_in),                                // input wire
aclk .s_axis_data_tvalid(valid_in),  // input wire s_axis_data_tvalid
//  .s_axis_data_tready(ready_out),  // output wire s_axis_data_tready
//  .s_axis_data_tdata($signed(guitar_in)),    // input wire [47 : 0] s_axis_data_tdata
//  .m_axis_data_tvalid(valid_out),  // output wire m_axis_data_tvalid
//  .m_axis_data_tready(ready_in),  // input wire m_axis_data_tready
//  .m_axis_data_tdata(wah_1_out)    // output wire [55 : 0] m_axis_data_tdata
//);



logic ready_out_1;
logic valid_out_1;
wah_0_1 wah_1 (
  .aclk(clk_in),                                // input wire aclk
  .s_axis_data_tvalid(valid_in),  // input wire s_axis_data_tvalid
  .s_axis_data_tready(ready_out_1),  // output wire s_axis_data_tready
  .s_axis_data_tdata($signed(guitar_in)),    // input wire [47 : 0] s_axis_data_tdata
  .m_axis_data_tvalid(valid_out_1),  // output wire m_axis_data_tvalid
  .m_axis_data_tdata(wah_1_out)    // output wire [63 : 0] m_axis_data_tdata
);


logic ready_out_2;
logic valid_out_2;
wah_0_2 wah_2 (
  .aclk(clk_in),                                // input wire aclk
  .s_axis_data_tvalid(valid_in),  // input wire s_axis_data_tvalid
  .s_axis_data_tready(ready_out_2),  // output wire s_axis_data_tready
  .s_axis_data_tdata($signed(guitar_in)),    // input wire [47 : 0] s_axis_data_tdata
  .m_axis_data_tvalid(valid_out_2),  // output wire m_axis_data_tvalid
  .m_axis_data_tdata(wah_2_out)    // output wire [63 : 0] m_axis_data_tdata
);


logic ready_out_3;
logic valid_out_3;
wah_0_3 wah_3 (
  .aclk(clk_in),                                // input wire aclk
  .s_axis_data_tvalid(valid_in),  // input wire s_axis_data_tvalid
  .s_axis_data_tready(ready_out_3),  // output wire s_axis_data_tready
  .s_axis_data_tdata($signed(guitar_in)),    // input wire [47 : 0] s_axis_data_tdata
  .m_axis_data_tvalid(valid_out_3),  // output wire m_axis_data_tvalid
  .m_axis_data_tdata(wah_3_out)    // output wire [63 : 0] m_axis_data_tdata
);


logic ready_out_4;
logic valid_out_4;
wah_0_4 wah_4 (
  .aclk(clk_in),                                // input wire aclk
  .s_axis_data_tvalid(valid_in),  // input wire s_axis_data_tvalid
  .s_axis_data_tready(ready_out_4),  // output wire s_axis_data_tready
  .s_axis_data_tdata($signed(guitar_in)),    // input wire [47 : 0] s_axis_data_tdata
  .m_axis_data_tvalid(valid_out_4),  // output wire m_axis_data_tvalid
  .m_axis_data_tdata(wah_4_out)    // output wire [63 : 0] m_axis_data_tdata
);
```

```
logic ready_out_5;
logic valid_out_5;
wah_0_5 wah_5 (
  .aclk(clk_in),                                // input wire aclk
  .s_axis_data_tvalid(valid_in),  // input wire s_axis_data_tvalid
  .s_axis_data_tready(ready_out_5),  // output wire s_axis_data_tready
  .s_axis_data_tdata($signed(guitar_in)),    // input wire [47 : 0] s_axis_data_tdata
  .m_axis_data_tvalid(valid_out_5),  // output wire m_axis_data_tvalid
  .m_axis_data_tdata(wah_5_out)    // output wire [63 : 0] m_axis_data_tdata
);


logic ready_out_6;
logic valid_out_6;
wah_0_6 wah_6 (
  .aclk(clk_in),                                // input wire aclk
  .s_axis_data_tvalid(valid_in),  // input wire s_axis_data_tvalid
  .s_axis_data_tready(ready_out_6),  // output wire s_axis_data_tready
  .s_axis_data_tdata($signed(guitar_in)),    // input wire [47 : 0] s_axis_data_tdata
  .m_axis_data_tvalid(valid_out_6),  // output wire m_axis_data_tvalid
  .m_axis_data_tdata(wah_6_out)    // output wire [63 : 0] m_axis_data_tdata
);


logic ready_out_7;
logic valid_out_7;
wah_0_7 wah_7 (
  .aclk(clk_in),                                // input wire aclk
  .s_axis_data_tvalid(valid_in),  // input wire s_axis_data_tvalid
  .s_axis_data_tready(ready_out_7),  // output wire s_axis_data_tready
  .s_axis_data_tdata($signed(guitar_in)),    // input wire [47 : 0] s_axis_data_tdata
  .m_axis_data_tvalid(valid_out_7),  // output wire m_axis_data_tvalid
  .m_axis_data_tdata(wah_7_out)    // output wire [63 : 0] m_axis_data_tdata
);


logic ready_out_8;
logic valid_out_8;
wah_0_8 wah_8 (
  .aclk(clk_in),                                // input wire aclk
  .s_axis_data_tvalid(valid_in),  // input wire s_axis_data_tvalid
  .s_axis_data_tready(ready_out_8),  // output wire s_axis_data_tready
  .s_axis_data_tdata($signed(guitar_in)),    // input wire [47 : 0] s_axis_data_tdata
  .m_axis_data_tvalid(valid_out_8),  // output wire m_axis_data_tvalid
  .m_axis_data_tdata(wah_8_out)    // output wire [63 : 0] m_axis_data_tdata
);
endmodule
```

```
module tremolo_pedal #(
    parameter int DATA_WIDTH = 24,
    parameter TREMOLO_ROTARY_WIDTH = 8
    )(
    input wire  clk_in,
    input wire  rst_in,
    input wire  valid_in,
    input wire  [7:0] mag_in,
    input wire signed [23:0] small_guitar_in,
    output logic [23:0] small_guitar_out,
    output logic valid_out
    );

    localparam WAVE_WIDTH = 22; // max req'd width. may be overextended
    localparam STAGES     = 3;

    logic [WAVE_WIDTH-1:0] wave_half_period;
    assign wave_half_period = 'd1048576 + mag_in * 'd8192;

    logic signed [  DATA_WIDTH-1:0] data_r   = '0;
    logic signed [2*DATA_WIDTH-1:0] data_m   = '0;
    logic signed [  DATA_WIDTH-1:0] data_eff = '0;
    logic        [      STAGES-1:0] vld_eff  = '0;

    logic cnt;
    logic nxt = &cnt;
    logic [WAVE_WIDTH-1:0] small_wav;
    logic [DATA_WIDTH-1:0] wav;
    assign wav = small_wav;


    always_ff @(posedge clk_in) begin
        if (rst_in) begin
            cnt <= 1'b0;
        end else begin
            cnt <= cnt + 1;
        end
    end

    tri_wav tri_wav_0 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .nxt_in(nxt),
        .N_in(wave_half_period),
        .wav_out(small_wav)
    );

    always_ff @(posedge clk_in) begin
        if (rst_in) begin
            data_r   <= '0;
            data_m   <= '0;
            data_eff <= '0;
        end else begin
            data_r   <= small_guitar_in;
            data_m   <= {{WAVE_WIDTH{data_r[DATA_WIDTH-1]}}, data_r} *
{{WAVE_WIDTH{wav[DATA_WIDTH-1]}}, wav};
            data_eff <= data_m >>> WAVE_WIDTH;
        end
    end


    always_ff @(posedge clk_in) begin
```

```
        if (rst_in) vld_eff <= '0;
        else        vld_eff <= (vld_eff << 1) | valid_in;
    end

    always_comb small_guitar_out = data_eff;
    always_comb valid_out = vld_eff[STAGES-1];

endmodule // tremolo_pedal



module tri_wav #(
    parameter int WAVE_WIDTH = 22
    )(
    input wire clk_in,
    input wire rst_in,
    input wire nxt_in, // pulse for output wave to advance to next value
    input wire [WAVE_WIDTH:0] N_in,
    output logic [WAVE_WIDTH-1:0] wav_out = '0
    );

    logic dir = 0;

    always_ff @ (posedge clk_in) begin
        if (rst_in) begin
            dir <= 0;
        end else if (wav_out == N_in - 1) begin
            dir <= 1;
        end else if (wav_out == 1) begin
            dir <= 0;
        end
    end

    always_ff @ (posedge clk_in) begin
        if (rst_in) begin
            wav_out <= '0;
        end else begin
            wav_out <= (dir) ? wav_out - nxt_in : wav_out + nxt_in;
        end
    end

endmodule
```

```
`timescale 1ns / 1ps

module harmonizer (
    input wire clk_in,
    input wire rst_in,
    input wire valid_in,
    input wire signed [23:0] data_in_0,
    input wire signed [23:0] data_in_1,
    input wire [3:0] delta_opt_in,
    output logic [23:0] data_out_0,
    output logic [23:0] data_out_1,
    output logic valid_out
    );

    logic h1en;
    logic h3en;
    logic h4en;
    logic h5en;

    assign h1en = delta_opt_in[3];
    assign h3en = delta_opt_in[2];
    assign h4en = delta_opt_in[1];
    assign h5en = delta_opt_in[0];

    logic signed [23:0] pitch_shift_out_3 [1:0];
    logic signed [23:0] pitch_shift_out_4 [1:0];
    logic signed [23:0] pitch_shift_out_5 [1:0];

    logic pitch_shift_valid_3 [1:0];
    logic pitch_shift_valid_4 [1:0];
    logic pitch_shift_valid_5 [1:0];

    // DELTA OPTIONS
    // major 3rd: 2^(4/12) = 2^(-2) + 2^(-7) + 2^(-9) + 2^(-13)
    logic [31:0] delta_major3;
    assign delta_major3 = 32'b00101000010100010000000000000000;

    // perfect 4th: 2^(5/12) = 2^(-2) + 2^(-4) + 2^(-6) + 2^(-8)
    //                       + 2^(-9) + 2^(-11) + 2^(-12) + 2^(-13)
    //                       + 2^(-20) + 2^(-25) + 2^(-29)
    logic [31:0] delta_perfect4;
    assign delta_perfect4 = 32'b00101010101101111000000100010001;

    // perfect 5th: 2^(7/12) = 2^(-2) + 2^(-3) + 2^(-4) + 2^(-5)
    //                       + 2^(-6) + 2^(-7) + 2^(-8) + 2^(-9)
    //                       + 2^(-12) + 2^(-16) + 2^(-21) + 2^(-22)
    //                       + 2^(-24) + 2^(-26) + 2^(-27) + 2^(-28)
    logic [31:0] delta_perfect5;
    assign delta_perfect5 = 32'b00101111111110010001000011010111O;

    pitch_shifter #(1024) d3_0 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .delta_in(delta_major3),
        .new_sample_valid_in(valid_in),
        .new_sample_data_in(data_in_0),
        .pitch_shift_data_out(pitch_shift_out_3[0]),
        .pitch_shift_valid_out(pitch_shift_valid_3[0])
        );

    pitch_shifter #(1024) d3_1 (
        .clk_in(clk_in),
```

```
        .rst_in(rst_in),
        .delta_in(delta_major3),
        .new_sample_valid_in(valid_in),
        .new_sample_data_in(data_in_1),
        .pitch_shift_data_out(pitch_shift_out_3[1]),
        .pitch_shift_valid_out(pitch_shift_valid_3[1])
        );

    pitch_shifter #(1024) d4_0 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .delta_in(delta_perfect4),
        .new_sample_valid_in(valid_in),
        .new_sample_data_in(data_in_0),
        .pitch_shift_data_out(pitch_shift_out_4[0]),
        .pitch_shift_valid_out(pitch_shift_valid_4[0])
        );

     pitch_shifter #(1024) d4_1 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .delta_in(delta_perfect4),
        .new_sample_valid_in(valid_in),
        .new_sample_data_in(data_in_1),
        .pitch_shift_data_out(pitch_shift_out_4[1]),
        .pitch_shift_valid_out(pitch_shift_valid_4[1])
        );

     pitch_shifter #(1024) d5_0 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .delta_in(delta_perfect5),
        .new_sample_valid_in(valid_in),
        .new_sample_data_in(data_in_0),
        .pitch_shift_data_out(pitch_shift_out_5[0]),
        .pitch_shift_valid_out(pitch_shift_valid_5[0])
        );

     pitch_shifter #(1024) d5_1 (
        .clk_in(clk_in),
        .rst_in(rst_in),
        .delta_in(delta_perfect5),
        .new_sample_valid_in(valid_in),
        .new_sample_data_in(data_in_1),
        .pitch_shift_data_out(pitch_shift_out_5[1]),
        .pitch_shift_valid_out(pitch_shift_valid_5[1])
        );

     // add harmonics or not
     logic signed [23:0] h3 [1:0];
     logic signed [23:0] h4 [1:0];
     logic signed [23:0] h5 [1:0];

     assign h3[0] = (h3en) ? pitch_shift_out_3[0] : 24'b0;
     assign h3[1] = (h3en) ? pitch_shift_out_3[1] : 24'b0;
     assign h4[0] = (h4en) ? pitch_shift_out_4[0] : 24'b0;
     assign h4[1] = (h4en) ? pitch_shift_out_4[1] : 24'b0;
     assign h5[0] = (h5en) ? pitch_shift_out_5[0] : 24'b0;
     assign h5[1] = (h5en) ? pitch_shift_out_5[1] : 24'b0;

     always_ff @(posedge clk_in) begin
        data_out_0 <= data_in_0 + h3[0] + h4[0] + h5[0];
```

```
        data_out_1 <= data_in_1 + h3[1] + h4[1] + h5[1];
     end

endmodule // harmonizer

module pitch_shifter #(
    parameter B = 1024              // depth of buffer
    )(
    input wire clk_in,
    input wire rst_in,
    input wire [31:0] delta_in,     // 3.29 format pitch-shift amount
    input wire new_sample_valid_in,
    input wire signed [31:0] new_sample_data_in,
    output logic signed [31:0] pitch_shift_data_out,
    output logic signed [31:0] pitch_shift_valid_out
    );

    logic signed [31:0] new_sample_data_reg;
    logic [2:0] state;

    // enables
    logic b1_rden;
    logic b1_wren;

    // fractional index for pitch shifting $clog2(B).29 format
    logic [$clog2(B)+28:0]  b1_index;

    // buffer read and write addresses
    logic [$clog2(B)-1:0]   b1_read_addr;
    logic [$clog2(B)-1:0]   b1_write_addr;

    // read data
    logic signed [31:0]     b1_read_out;

    // mux between smoothed and unsmoothed
    assign pitch_shift_data_out = b1_read_out;
    assign pitch_shift_valid_out = (state == 3'd2) ? 1'b1 : 1'b0;

    buffer #(B) b1 (
        .clk_in(clk_in),
        .rden_in(b1_rden),
        .wren_in(b1_wren),
        .rdaddr_in(b1_read_addr),
        .wraddr_in(b1_write_addr),
        .data_in(new_sample_data_reg),
        .data_out(b1_read_out)
        );

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

            state                   <= 3'd0;
            b1_index              <= 0;
            b1_write_addr         <= 0;
            b1_read_addr          <= 0;
            new_sample_data_reg    <= 0;
            b1_rden               <= 0;
            b1_wren               <= 0;

        end else if ((state == 3'd0) && new_sample_valid_in) begin // state 0 = waiting for
new samp
            new_sample_data_reg <= new_sample_data_in;
```

```
            b1_write_addr <= b1_write_addr + 1;   // increment write addr, modulo rb size
            b1_index <= (b1_index + delta_in);    // update fractional index, modulo rb size
            b1_read_addr <= b1_index[$clog2(B)+28 : 29]; // only take integer part for read
address
            b1_rden <= 1;
            b1_wren <= 1;;
            state   <= 3'd1;

        end else if (state == 3'd1) begin // done writing
            b1_wren <= 0;
            state   <= 3'd2;
        end else if (state == 3'd2) begin // done reading
            b1_rden <= 0;
            state   <= 3'd0;
        end else begin
            b1_rden    <= 0;
            b1_wren    <= 0;
        end
    end

endmodule // pitch-shifter


module buffer #(
    parameter B = 1024
    )(
    input wire                  clk_in,
    input wire                  rden_in,
    input wire                  wren_in,
    input wire [($clog2(B)-1):0]rdaddr_in,
    input wire [($clog2(B)-1):0]wraddr_in,
    input wire signed [31:0]    data_in,
    output logic signed [31:0]  data_out
    );

    logic [($clog2(B)-1):0] read_addr_r;
    logic signed [31:0]     mem [(B-1):0];
    logic                   rden_r;

    always_ff @(posedge clk_in) begin    // write latency 1 cycle
        if (wren_in) begin
            mem[wraddr_in] <= data_in;
        end
    end

    always_ff @(posedge clk_in) begin    // read latency 3 cycles
        read_addr_r <= rdaddr_in;
        rden_r <= rden_in;

        if (rden_r) begin
            data_out <= mem[read_addr_r];
        end

    end

endmodule
```

```
`timescale 1ns / 1ps
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 12/05/2021 02:21:30 PM
// Design Name:
// Module Name: stft
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module stft_joe(    input wire clk_100mhz,
                    input wire clk_in,
                    input wire [15:0] sw,
                    input wire btnc, btnu, btnd, btnr, btnl,
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // Put in an audio_in signal instead of getting the signal ourselves

                    output logic[3:0] vga_r,
                    output logic[3:0] vga_b,
                    output logic[3:0] vga_g,
                    output logic vga_hs,
                    output logic vga_vs,
                    output logic [15:0] led,
                    input logic sample_trigger,
                    input  logic signed [23:0] guitar_in //small data
    );


    logic [15:0] sample_counter;
    logic [11:0] adc_data;
    logic [11:0] sampled_adc_data;
    logic sample_trigger;
    logic adc_ready;
    logic enable;
    logic [7:0] recorder_data;
    logic [7:0] vol_out;
    logic pwm_val; //pwm signal (HI/LO)
    logic [15:0] scaled_adc_data;
    logic [15:0] scaled_signed_adc_data;
    logic [15:0] fft_data;
    logic       fft_ready;
    logic       fft_valid;
    logic       fft_last;
    logic [9:0] fft_data_counter;

    logic fft_out_ready;
    logic fft_out_valid;
    logic fft_out_last;
    logic [31:0] fft_out_data;
```

```
    logic sqsum_valid;
    logic sqsum_last;
    logic sqsum_ready;
    logic [31:0] sqsum_data;

    logic fifo_valid;
    logic fifo_last;
    logic fifo_ready;
    logic [31:0] fifo_data;

    logic [23:0] sqrt_data;
    logic sqrt_valid;
    logic sqrt_last;

    logic pixel_clk;


      clk_wiz_1 myvga(
            .clk_out1(pixel_clk),     // output clk_out1
            .clk_in1(clk_100mhz));      // input clk_in1

    assign led = sw; //just to look pretty
    //-------------------------------------------------------------------------------------
What to sample_rtigger on
    always_ff @(posedge clk_in) begin
        if (sample_trigger) begin
//            scaled_data = ...........  MIGHT NEED TO SCALE DATA HERE?
            if (fft_ready) begin
                fft_data_counter <= fft_data_counter + 1;
                //if we've sent 1023, indicate last sample of frame!
                fft_last <= fft_data_counter == 1023;
                fft_valid <= 1'b1;
                fft_data <= guitar_in[23:8]; /////// POTENTIAL PUTFALL
            end
        end else begin
            fft_data <= 0;
            fft_last <= 0;
            fft_valid <= 0;
        end
   end //always_ff end

xfft_joe joe_fft (
  .aclk(clk_in),                                                // input wire aclk
  .s_axis_config_tdata(0),     //Not optional so set to 0     // input wire [15 : 0]
s_axis_config_tdata
  .s_axis_config_tvalid(0),    //Not optional so set to 0     // input wire
s_axis_config_tvalid
  .s_axis_config_tready(),     //Dont need to read            // output wire
s_axis_config_tready
  .s_axis_data_tdata(fft_data),     /// input wire [31 : 0] s_axis_data_tdata
  .s_axis_data_tvalid(fft_valid),                   // input wire s_axis_data_tvalid
  .s_axis_data_tready(fft_ready),                   // output wire s_axis_data_tready
  .s_axis_data_tlast(fft_last),                     // input wire s_axis_data_tlast
  .m_axis_data_tdata(fft_out_data),                     // output wire [31 : 0]
m_axis_data_tdata
  .m_axis_data_tvalid(fft_out_valid),                   // output wire m_axis_data_tvalid
  .m_axis_data_tready(fft_out_ready),                   // input wire m_axis_data_tready
  .m_axis_data_tlast(fft_out_last),                     // output wire m_axis_data_tlast
  .event_frame_started(),                  // output wire event_frame_started
  .event_tlast_unexpected(),           // output wire event_tlast_unexpected
  .event_tlast_missing(),                  // output wire event_tlast_missing
```

```
  .event_status_channel_halt(),      // output wire event_status_channel_halt
  .event_data_in_channel_halt(),    // output wire event_data_in_channel_halt
  .event_data_out_channel_halt()  // output wire event_data_out_channel_halt
);



    joe_fifo myfifo (
      .s_axis_aresetn(1'b1),  // input wire s_axis_aresetn
      .s_axis_aclk(clk_in),        // input wire s_axis_aclk
      .s_axis_tvalid(sqsum_valid),    // input wire s_axis_tvalid
      .s_axis_tready(sqsum_ready),    // output wire s_axis_tready
      .s_axis_tdata(sqsum_data),
      .s_axis_tlast(sqsum_last),      // input wire [x : 0] s_axis_tdata
      .m_axis_tvalid(fifo_valid),    // output wire m_axis_tvalid
      .m_axis_tready(fifo_ready),
      .m_axis_tlast(fifo_last),    // input wire m_axis_tready
      .m_axis_tdata(fifo_data)      // output wire [x : 0] m_axis_tdata
    );

joe_sqrt mysqrt (
      .aclk(clk_in),                                        // input wire aclk
      .s_axis_cartesian_tvalid(fifo_valid),  // input wire s_axis_cartesian_tvalid
      .s_axis_cartesian_tready(fifo_ready),  // output wire s_axis_cartesian_tready
      .s_axis_cartesian_tlast(fifo_last),    // input wire s_axis_cartesian_tlast
      .s_axis_cartesian_tdata(fifo_data),    // input wire [x : 0] s_axis_cartesian_tdata
      .m_axis_dout_tvalid(sqrt_valid),           // output wire m_axis_dout_tvalid
      .m_axis_dout_tlast(sqrt_last),             // output wire m_axis_dout_tlast
      .m_axis_dout_tdata(sqrt_data)             // output wire [x : 0] m_axis_dout_tdata
    );



square_and_sum_v1_0 mysq(.s00_axis_aclk(clk_in), .s00_axis_aresetn(1'b1),
                        .s00_axis_tready(fft_out_ready),
                        .s00_axis_tdata(fft_out_data),.s00_axis_tlast(fft_out_last),
                        .s00_axis_tvalid(fft_out_valid),.m00_axis_aclk(clk_in),
                        .m00_axis_aresetn(1'b1),. m00_axis_tvalid(sqsum_valid),
                        .m00_axis_tdata(sqsum_data),.m00_axis_tlast(sqsum_last),
                        .m00_axis_tready(sqsum_ready));




    logic [9:0] addr_count;
    logic [9:0] draw_addr;
    logic [31:0] amp_out;
    logic [10:0] hcount;
    logic [9:0] vcount;
    logic       vsync;
    logic       hsync;
    logic       blanking;
    logic [11:0] rgb;

    always_ff @(posedge clk_in)begin
        if (sqrt_valid)begin
            if (sqrt_last)begin
                addr_count <= 'd1023; //allign
            end else begin
```

```
                addr_count <= addr_count + 1'b1;
            end
        end

    end



joe_bram mvb (
  .clka(clk_in),    // input wire clka
  .ena(1'b1),      // input wire ena
  .wea(sqrt_valid),      // input wire [0 : 0] wea
  .addra(addr_count+3),  // input wire [9 : 0] addra//-
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  .dina(sqrt_data),    // input wire [47 : 0] dina
  .douta(),  // output wire [47 : 0] douta
  .clkb(pixel_clk),    // input wire clkb
  .enb(1'b1),      // input wire enb
  .web(1'b0),      // input wire [0 : 0] web
  .addrb(draw_addr),  // input wire [9 : 0] addrb
  .dinb(0),    // input wire [47 : 0] dinb
  .doutb(amp_out)  // output wire [47 : 0] doutb
);


    //draw bargraphs from amp_out extracted (scale with switches)
    always_ff @(posedge pixel_clk)begin
//        if (!blanking)begin //time to draw!
//            rgb <= 12'b0011_0000_0000;
//        end
        draw_addr <= hcount/2;
        if ((amp_out>>sw[7:4])>=768-vcount)begin
            rgb <= sw[15:8];
        end else begin
            rgb <= 12'b0000_0000_0000;
        end

    end
    xvga myyvga (.vclock_in(pixel_clk),.hcount_out(hcount),
                .vcount_out(vcount),.vsync_out(vsync), .hsync_out(hsync),
                 .blank_out(blanking));

    assign vga_r = ~blanking ? rgb[11:8]: 0;
    assign vga_g = ~blanking ? rgb[7:4] : 0;
    assign vga_b = ~blanking ? rgb[3:0] : 0;

    assign vga_hs = ~hsync;
    assign vga_vs = ~vsync;

endmodule


// CUSTOM SQUARE AND SUM OMDULE
module square_and_sum_v1_0 #
    (
        // Users to add parameters here

        // User parameters ends
        // Do not modify the parameters beyond this line


        // Parameters of Axi Slave Bus Interface S00_AXIS
```

```
        parameter integer C_S00_AXIS_TDATA_WIDTH    = 32,

        // Parameters of Axi Master Bus Interface M00_AXIS
        parameter integer C_M00_AXIS_TDATA_WIDTH    = 32,
        parameter integer C_M00_AXIS_START_COUNT    = 32
    )
    (
        // Users to add ports here

        // User ports ends
        // Do not modify the ports beyond this line


        // Ports of Axi Slave Bus Interface S00_AXIS
        input wire  s00_axis_aclk,
        input wire  s00_axis_aresetn,
        output wire  s00_axis_tready,
        input wire [C_S00_AXIS_TDATA_WIDTH-1 : 0] s00_axis_tdata,
        input wire  s00_axis_tlast,
        input wire  s00_axis_tvalid,

        // Ports of Axi Master Bus Interface M00_AXIS
        input wire  m00_axis_aclk,
        input wire  m00_axis_aresetn,
        output wire  m00_axis_tvalid,
        output wire [C_M00_AXIS_TDATA_WIDTH-1 : 0] m00_axis_tdata,
        output wire  m00_axis_tlast,
        input wire  m00_axis_tready
    );

    reg m00_axis_tvalid_reg_pre;
    reg m00_axis_tlast_reg_pre;
    reg m00_axis_tvalid_reg;
    reg m00_axis_tlast_reg;
    reg [C_M00_AXIS_TDATA_WIDTH-1 : 0] m00_axis_tdata_reg;

    reg s00_axis_tready_reg;
    reg signed [31:0] real_square;
    reg signed [31:0] imag_square;

    wire signed [15:0] real_in;
    wire signed [15:0] imag_in;
    assign real_in = s00_axis_tdata[31:16];
    assign imag_in = s00_axis_tdata[15:0];

    assign m00_axis_tvalid = m00_axis_tvalid_reg;
    assign m00_axis_tlast = m00_axis_tlast_reg;
    assign m00_axis_tdata = m00_axis_tdata_reg;
    assign s00_axis_tready = s00_axis_tready_reg;

    always @(posedge s00_axis_aclk)begin
        if (s00_axis_aresetn==0)begin
            s00_axis_tready_reg <= 0;
        end else begin
            s00_axis_tready_reg <= m00_axis_tready; //if what you're feeding data to is
ready, then you're ready.
        end
    end

    always @(posedge m00_axis_aclk)begin
        if (m00_axis_aresetn==0)begin
            m00_axis_tvalid_reg <= 0;
```

```
            m00_axis_tlast_reg <= 0;
            m00_axis_tdata_reg <= 0;
        end else begin
            m00_axis_tvalid_reg_pre <= s00_axis_tvalid; //when new data is coming, you've got
new data to put out
            m00_axis_tlast_reg_pre <= s00_axis_tlast; //
            real_square <= real_in*real_in;
            imag_square <= imag_in*imag_in;

            m00_axis_tvalid_reg <= m00_axis_tvalid_reg_pre; //when new data is coming, you've
got new data to put out
            m00_axis_tlast_reg <= m00_axis_tlast_reg_pre; //
            m00_axis_tdata_reg <= real_square + imag_square;
        end
    end
endmodule

module xvga(input wire vclock_in,
            output logic [10:0] hcount_out,    // pixel number on current line
            output logic [9:0] vcount_out,     // line number
            output logic vsync_out, hsync_out,
            output logic 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
   logic hblank,vblank;
   logic 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
   logic 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
   logic 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
      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
endmodule
```