6.111 Final Project: FPGA SDR

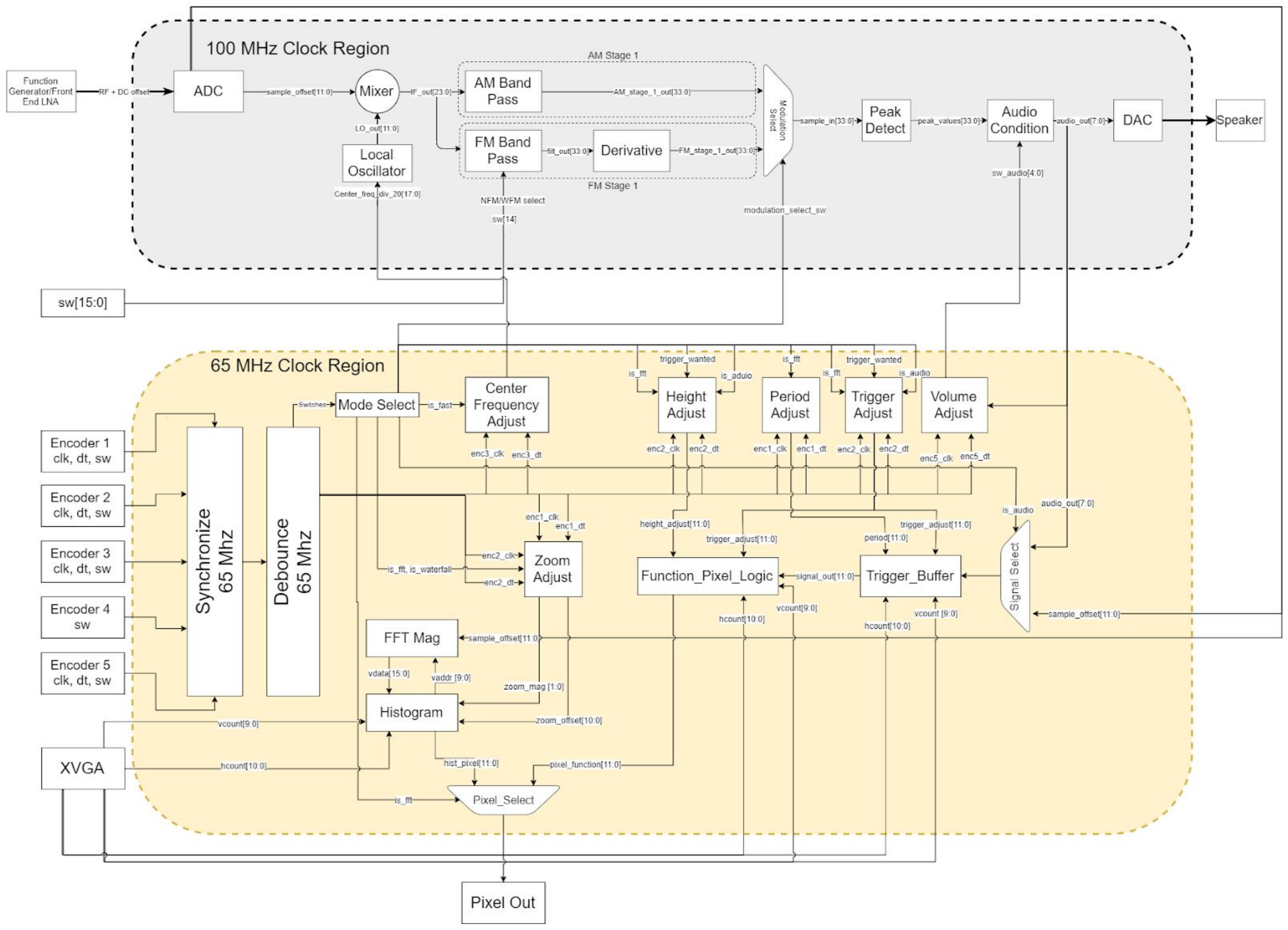
Colin Chaney
Charles Lindsay

Introduction

Software Defined Radios (SDRs) are special in the fact that they use very few analog components. In most cases the only analog components are the RF front ends that ensure a clean analog signal is fed into the ADC of the radio. Where SDRs have an advantage to analog radios is that they are flexible. A SDR is able to apply any number of signal processing tricks with the received data and not be locked into a single solution. This can be especially useful in applications such as satellites, where being able to reconfigure hardware on orbit can be essential to mission success.

For our project, our aim was to demodulate AM and FM audio while providing an intuitive user interface. Since we realized that demodulating real RF waves off of the air would be a stretch, we wanted to focus on getting the demodulation to work with function generators. For our user interface we wanted to be able to display any waveforms coming from the ADC as well as the demodulated audio. In addition we wanted to be able to view the frequency spectrum of our incoming signal, such that we could tune the center frequency much easier and identify radio signals such as broadcast stations.

FPGA Implementation Overview



In the FPGA we have two separate clock domains, both encompassing different parts of the project. The 100 MHz clock domain focuses on the signal processing of demodulating the audio while the 65 MHz domain focuses on user interaction through encoders and a VGA display. Both the raw ADC data and the demodulated audio are sent to the display modules to be shown. The raw ADC data is also sent to a Fast Fourier Transform (FFT) module, so that users can view the entire frequency spectrum of the ADC signal.

Integration for this project was hardly an issue since the two clock domains create a pretty distinct separation between the realms. Charles was able to experiment with DSP filters and not affect any of the VGA display modules that Colin was working on.

Digital Signal Processing Chain (Charles)

Overview

On a high level, the digital signal processing (DSP) chain implemented on the Artix-7 FPGA is simply an analog superheterodyne receiver. The user inputs a desired frequency to tune to, which adjusts the radio’s local oscillator frequency in order to mix the received RF signal down to the intermediate frequency the bandpass filter is centered around. The resulting signal output from the bandpass filter for AM will be a single carrier frequency mixed with an audio signal. By performing envelope detection the audio signal can be extracted from the high frequency carrier. In the case of FM, more complex DSP must take place for good quality signal output, which usually consists of a phase-locked loop (PLL) digital circuit. In order to reduce complexity given our short development period, our design demodulates FM via derivative demodulation and also uses existing AM demodulation modules.

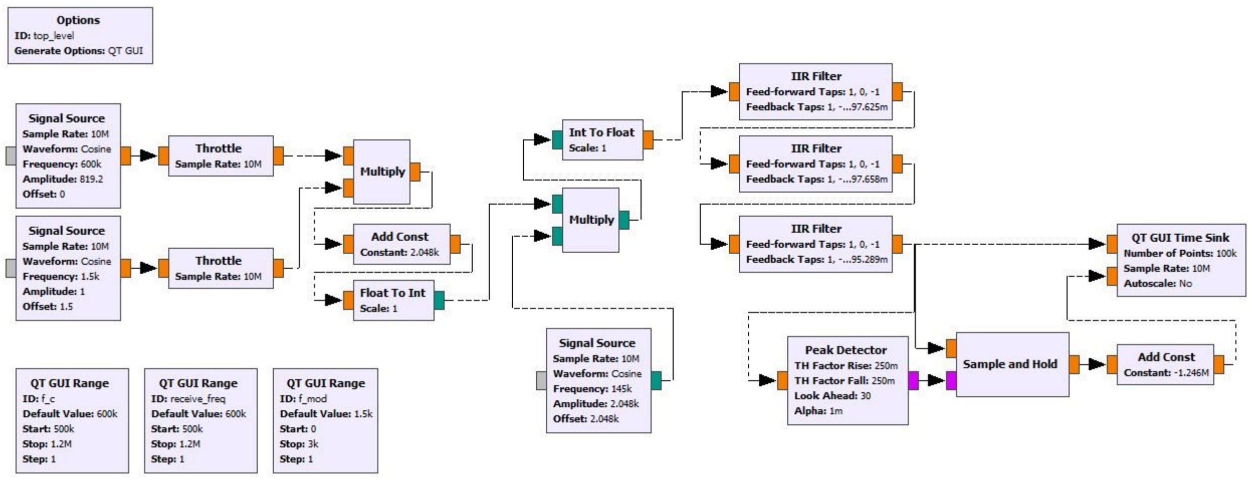
SDR Input Signal Generation and Testing

One or two function generators were used to drive the ADC analog input for all tests and demonstrations. Analog AM and FM signals ranging from 500 kHz to 5 MHz were output to test our system over the full frequency range of its intended design. We also output audio signals from our phones to the function generator in order to play music that could be modulated over any carrier frequency to simulate an AM broadcast station.

In addition, our radio is designed to operate with an RF front end biased above 0 V, although we did not have time to design a medium wave antenna and amplifier combination to receive AM broadcast radio.

GNU Radio Development

To model the complete AM demodulation from input signal generation through envelope detection, our system was simulated with GNU Radio blocks. This proved extremely useful for determining the maximum size of data passing through each module in order to help properly set our gain in our implementation and avoid overflow issues which are extremely difficult to detect in some cases. In addition, we could easily test FIR and IIR bandpass filter designs before simulating them on the FPGA which sped up the overall development process.



Options
ID: top_level
Generate Options: QT GUI

Signal Source
Sample Rate: 10M
Waveform: Cosine
Frequency: 600k
Amplitude: 819.2
Offset: 0

Signal Source
Sample Rate: 10M
Waveform: Cosine
Frequency: 1.5k
Amplitude: 1
Offset: 1.5

Throttle
Sample Rate: 10M

Throttle
Sample Rate: 10M

Multiply

Add Const
Constant: 2.048k

Float To Int
Scale: 1

QT GUI Range
ID: f_c
Default Value: 600k
Start: 500k
Stop: 1.2M
Step: 1

QT GUI Range
ID: receive_freq
Default Value: 600k
Start: 500k
Stop: 1.2M
Step: 1

QT GUI Range
ID: f_mod
Default Value: 1.5k
Start: 0
Stop: 3k
Step: 1

Int To Float
Scale: 1

Multiply

Signal Source
Sample Rate: 10M
Waveform: Cosine
Frequency: 145k
Amplitude: 2.048k
Offset: 2.048k

IIR Filter
Feed-forward Taps: 1, 0, -1
Feedback Taps: 1, -...97.625m

IIR Filter
Feed-forward Taps: 1, 0, -1
Feedback Taps: 1, -...97.658m

IIR Filter
Feed-forward Taps: 1, 0, -1
Feedback Taps: 1, -...95.289m

Peak Detector
TH Factor Rise: 250m
TH Factor Fall: 250m
Look Ahead: 30
Alpha: 1m

Sample and Hold

QT GUI Time Sink
Number of Points: 100k
Sample Rate: 10M
Autoscale: No

Add Const
Constant: -1.246M

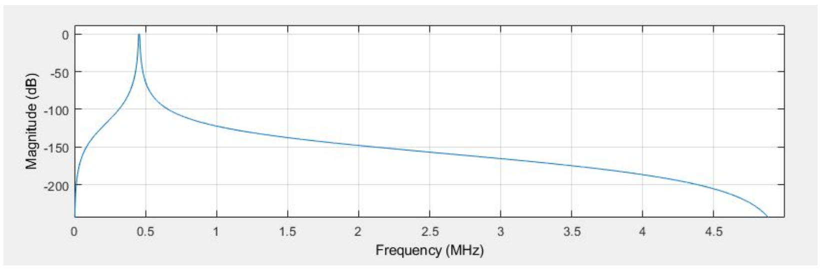
AM demodulation implementation built with GNU Radio.

Band Pass Filter Specifications

All bandpass filters were centered around the intermediate frequency $f_C = 455\ kHz$. For FM bandpass filters, the Carson Bandwidth Rule was used to determine the width of the passband given the maximum frequency deviation Δf and audio frequency $f_{i_{max}}$. For narrow band FM (NFM) $\Delta f = 5\ kHz$ and $f_{i_{max}} = 3\ kHz$, and for wideband FM (WFM) $\Delta f = 75\ kHz$ and $f_{i_{max}} = 15\ kHz$ which is typically used in US FM broadcast stations. The stopband for all filters were determined by the minimum spacing of broadcast stations in the US. The attenuation at the stop band was limited by the number of IIR filter sections we chose to implement, where more sections would have provided a steeper filter cutoff.

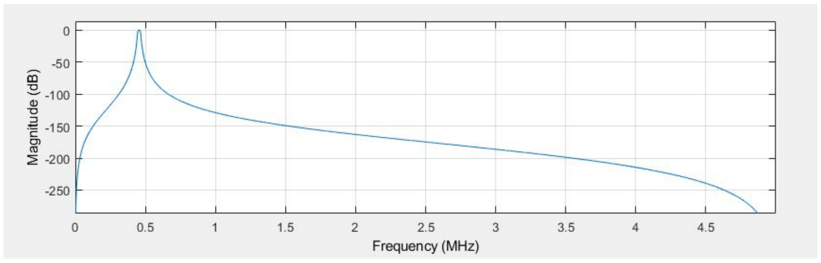
AM Bandpass

- Passband: f_C +/- 3 kHz
- Stopband: f_C +/- 5 kHz (8dB attenuation)



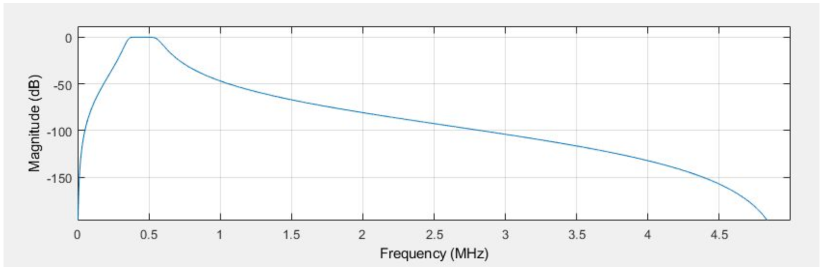
NFM Bandpass

- Passband: f_C +/- 8 kHz
- Stopband: f_C +/- 10 kHz (3dB attenuation)



WFM Bandpass

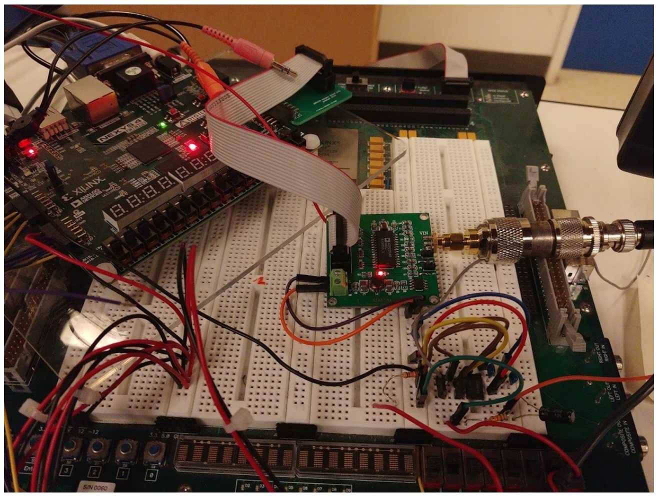
- Passband: f_C +/- 90 kHz
- Stopband: f_C +/- 110 kHz (3dB attenuation)



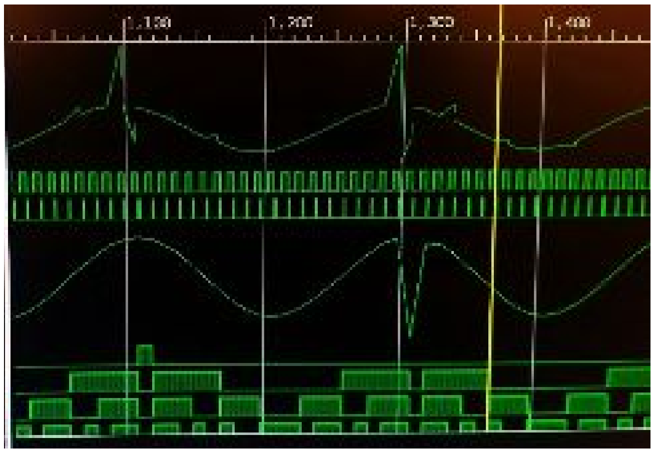
Modules

ADC (ADC_Interface.sv)

The AD9220 ADC we used has a maximum sampling rate of 10 MSPS, which limited the frequencies we could sample to below the nyquist frequency of 5 MHz. Since we were using a function generator for hardware testing and only outputting sinusoidal waveforms, we assumed that aliasing could be ignored as we could control the input spectrum.



An external breakout board for the AD9220 ADC was connected to the FPGA via two PMOD connectors. A clock divider was implemented to synthesize a 10 MHz clock which is connected directly to the PMOD port driving the ADC clock pin. The 12 data pins and out of range (OTR) pin are synchronized with two registers. Initially, the ADC module was designed to delay for four 100 MHz clock cycles after the rising edge of the synthesized 10 MHz clock before updating the module sample output. This includes two cycles to wait for both sync register outputs to reflect the most recent signal on the ADC pin, and two cycles to wait for the ADC data to become valid per AD9220 timing requirements. As we frequently observed glitches in the ADC output, we increased the delay to ten 100 MHz clock cycles and observed no future glitches. The sample output contains a DC bias therefore can be an unsigned value, but later in the signal chain this DC bias is removed and the processed signal is centered around 0 V.



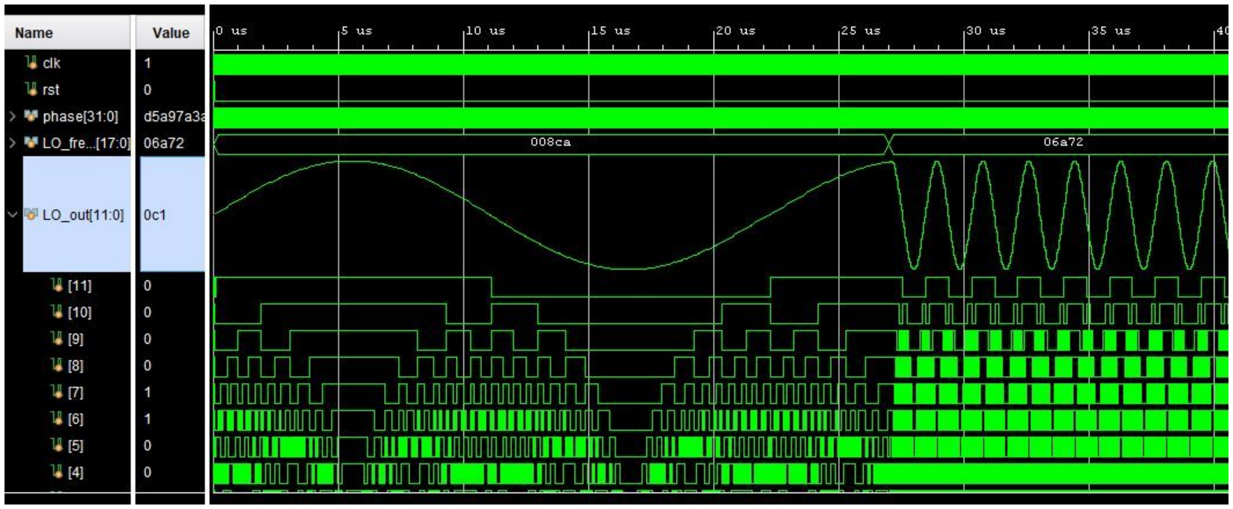
Debugging with ILA, where bottom analog waveform is the module output. An example of ADC glitch with four clock cycle delay

Local Oscillator (Local_Oscillator.sv)

Generates a sine wave that, when mixed with the sampled data output from the ADC, shifts the frequency content a user wants to demodulate into the desired pass band. The desired center frequency f_c is set by the user through our human interface, and the local oscillator (LO) synthesizes the frequency necessary to mix this signal down to the intermediate frequency (IF) where $f_{IF} = 455\ kHz$ and $f_{LO} = f_c - f_{IF}$. This is the typical IF used in most broadcast AM receivers.

A sine wave lookup table is implemented in a ROM with an 8 bit address that corresponds to the current phase, where $phase = \frac{2\pi * addr}{2^8 - 1}$ and each address contains a 12 bit value for the amplitude at that phase. In order to maintain signal resolution further down the DSP chain, the LO output must operate at 10 MSPS or greater to match the ADC sampling rate. This limits the lowest frequency we can tune our radio to since the number of samples per second will decrease as the LO frequency is decreased since the number samples n in our sine wave lookup table is constant. This can represented as $\frac{n}{T_c} \geq 10\ MSPS$ where T_c is the period of the sine wave being generated. Since the lowest frequency allocated to broadcast AM radio is 535 kHz, we chose to set the lowest frequency our SDR can receive to 500 kHz. This means a minimum $f_{LO} = 45\ kHz$, therefore n must be at least 8 bits to maintain an output sample rate of at least 10 MSPS, so we use an 8 bit ROM address.

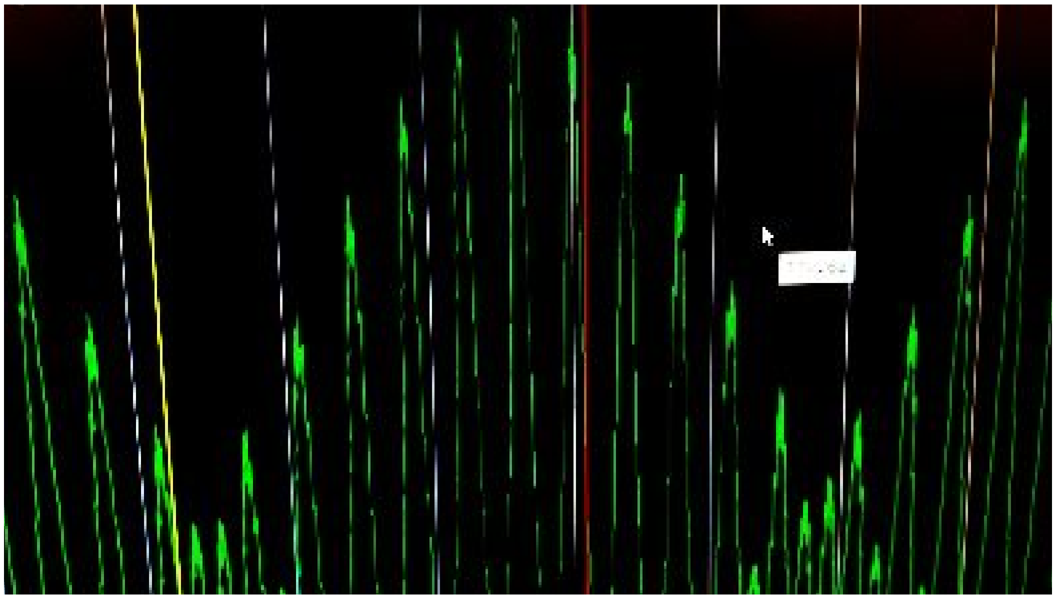
The top 8 bits of the 32 bit register value $phase$ are connected to the ROM address port. Once every 100 MHz clock cycle $phase$ is incremented by $phase\_inc$ which must be less than 32 bits, where $phase\_inc = \frac{f_{LO}}{f_{clk}}(2^{32} - 1)$. Letting parameter $\alpha = \frac{(2^{32}-1)}{f_{clk}}$, $phase\_inc = \alpha * f_{LO}$. If we want to be able to tune to all possible center frequencies from 0 Hz to 5 MHz with 1 Hz resolution, we would need f_c to be 23 bits. This means f_{LO} would be 23 bits in the maximum case, so that limits our α to a maximum of 9 bits given $phase$ and $phase\_inc$ are 32 bits. This results in a large loss of precision of the true output frequency of the LO with respect to the user set value, especially at high frequencies, due to floating point rounding errors when calculating the value of α used in implementation. To implement an accurate LO, α was constrained to a 14 bit value which meant our desired frequency had to be allocated to an 18 bit value now representing the desired center frequency f_c divided by 20. This is because with a frequency range of 0 Hz to 5 MHz, we can only achieve approximately 20 Hz of resolution using an 18 bit value. An error analysis in MATLAB showed that the error in desired LO frequency versus actual LO frequency increases linearly with frequency, and at 1 MHz the error is around 10 Hz if α is 14 bits which we considered sufficient to ensure no loss of information occurs during the subsequent bandpass filter stages.



Simulation verifying LO generates signal with desired frequency, that changes once during test

Mixer (Mixer.sv)

Multiples two 12 bit values, in this case the sampled data and local oscillator output where $IF_{out} = sample\_offset * LO_{out}$. Although a very simple implementation, this is one of the most powerful modules in the signal processing chain and produces very interesting outputs when viewed in the time domain, as seen on this synthesized ILA used to debug all stages up through mixing. This contains the desired low frequency signal and also undesired high frequency signal resulting from mixing, the latter of which must be filtered to obtain the original signal now shifted to the IF frequency of 455 kHz.


Mixer output on synthesized ILA

IIR Filter (AM_BP_filter.sv)

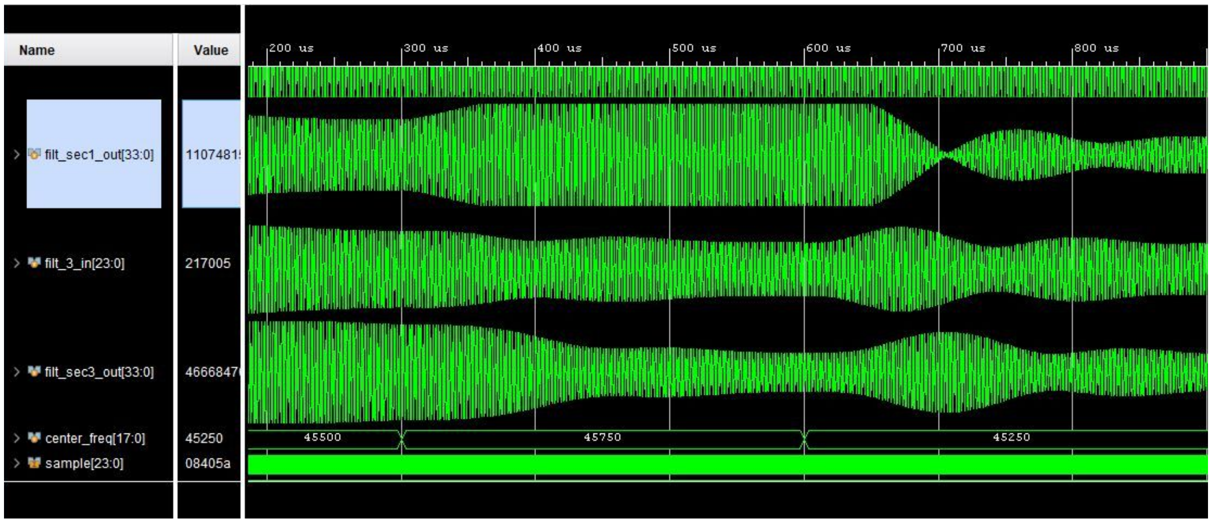
This filter module is used in both AM and FM demodulation, and consists of a customizable IIR filter section where any number of coefficients can be input to create an N^{th} order IIR filter.

An FIR filter was initially considered, but due to the 10 MSPS sampling rate there are only 10 clock cycles in which the summation of all filter values can be calculated as we are limited the FPGA's 100 MHz clock. This means we are limited to implementing bandpass filters with either a maximum of 10 taps if we choose to compute 1 value per clock cycle and sum with 1 register, or must use a more complicated implementation and much more hardware on the FPGA.

Given the bandpass filter specification for AM, the number of taps required is about 2000, therefore there were two proposed solutions to implement a FIR. It would be possible to downsample the IF signal to 1 MHz by first low passing to filter out high frequencies in the IF and avoid aliasing, and then after downsampling compute the output of a 400 tap FIR filter. By using four ROMS to hold filter tap coefficients and four BRAMS to hold previous sample values and operating them in parallel, you could sum 100 filter values separately over 100 clock cycles and add the four registers containing the final sums in order to compute the filter output just before the next sample arrived. This would be a fairly complex implementation and use a large amount of FPGA resources, so an IIR filter was selected as it could be easily implemented with cascaded sections each only containing 3 to 6 taps each for an AM or FM filter.

Each IIR filter consists of 1 section where $y[n] = \sum a_k y[n-k] + \sum b_l x[n-l]$, $k = [0, N-1]$ and $l = [1, M-1]$. The feedback coefficients a_k, or denominator, and feed-forward coefficients b_l were generated in MATLAB using the Filter Designer GUI. These signed coefficients are multiplied by 2^{16} and rounded to the nearest integer to eliminate some floating point precision errors in implementation. Some nonidealities in filter implementation were still observed, for example the simulated passband attenuation for AM was approximately 2.5 dB where it was designed to be 8 dB. This is not an issue since in reality most AM broadcast stations are further apart than the filter was designed at 10 kHz spacing, since stations tend to be at least 20 kHz apart in most cities.

When computing the filter output, the x and y sums tended to grow extremely high, so there are three 100 bit registers used in each IIR filter section. A 3 section IIR bandpass filter was simulated with a 455 kHz AM signal in order to determine an appropriate number of bits for these registers to avoid overflow and set proper gains, and determine a maximum filter output value.



Simulation verifying output of 3 IIR filter sections for the AM bandpass filter do not overflow, and properly attenuate frequencies outside the pass band. Center_freq varies throughout test.

AM Stage 1 (implemented in top_level.sv)

The first stage of the AM filter is implemented in *top_level.sv* and consists of 3 IIR filter sections in series to create a bandpass filter centered around the intermediate frequency (IF), where the input is IF_{out} converted to a signed value. The output of each filter is right shifted to ensure the value does not overflow when passed into the next section. See the DSP section overview for filter design specifications.

FM Stage 1 (FM_demod_state_1)

Includes both a narrow band (NFM) and wideband FM (WFM) IIR bandpass filter. The filter input comes directly from IF_{out} converted to a signed value. A push button in our human interface allows each modulation scheme to be selected.

Originally a frequency detection method by identifying zero crossings in the filtered signal was proposed, but due to the low frequency FM deviation relative to the carrier frequency it would be difficult to identify changes in frequency. For example, a 455 kHz carrier signal with 15 kHz frequency deviation will see a maximum period deviation corresponding to 7 clock cycles given our 100 MHz clock, so it would be difficult to achieve good resolution without first mixing the carrier down to a frequency close to 0 Hz and low pass filtering the signal. Due to the extra complexity in implementation, a derivative demodulation scheme was chosen to increase resolution and decrease resources.

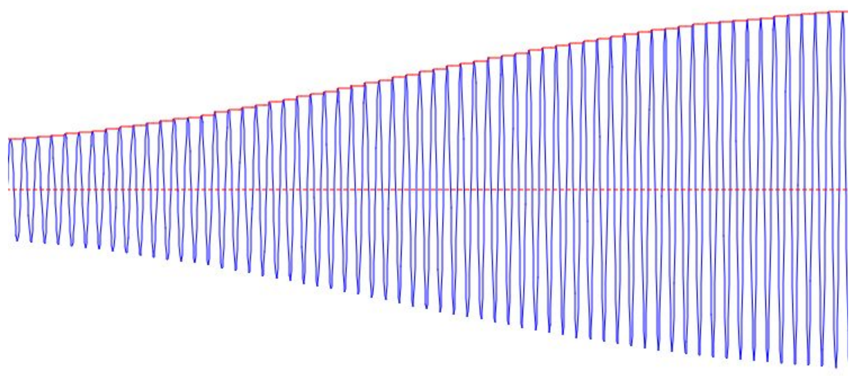
The difference of the past two values output from the final bandpass filter section is computed and output to the peak detection module. An FM signal $f(t) = A cos(w_c t + k\int s(t))$ where $s(t)$ is the audio signal being modulated and w_c is the carrier phase has a derivative $\frac{df(t)}{dt} = - A(w_c + ks(t)) * sin(w_c t + k\int s(t))$. This is essentially an AM signal, which allows the FM signal to be demodulated using the same modules as AM demodulation. This reduces system complexity greatly with the downside of some noise being introduced as the signal is not a simple AM signal.

We were able to successfully demodulate FM and output a tone with the correct frequency using this demodulation method, but several harmonics were present at times in addition to the modulated frequency and were noticeable when listening to the output tone. A simple averaging filter may have helped eliminate some of this noise.

Noisy FM signal output to speaker

Peak Detect and Hold (Peak_detect_hold.sv)

Demodulates an AM signal via envelope detection. This detects both positive and negative peaks of the signed input and will flip the sign if necessary to generate an output signal with the maximum resolution by identifying all extrema of the signal. The input signal will either be a true AM signal or FM signal derivative, both which update with a new input value at 10 MHz.


AM signal with 500 kHz carrier, generated in GNU Radio SDR simulation

The current input and previous input are used to calculate the derivative, and the sign of the derivative is encoded as a binary value and appended to an array containing only the previous 10 sign values. The sum of the array is constantly computed, and if the sum = 5 then it registers a peak has been detected. This works for both positive and negative peaks, as there will be an equal number of positive and negative derivatives surrounding a peak.

Once a peak is detected, an array of the previous 10 input values is searched to find the minimum or maximum value, which is determined by summing all values in the array. If the sum is less than zero, the algorithm will find the minimum value and set the output register to the magnitude of this value which is the value of the peak. This also effectively downsamples the signal to 0.91 MSPS, as only peaks register as valid output values and points in between are ignored.

Audio Condition (AM_audio_condition.sv)

This module takes the output signal from the peak detect and hold module and generates a 48 kHz, 8 bit audio signal for the DAC module which converts the digital audio signal into an analog output to drive a speaker.

A clock divider is used to down sample the 0.91 MSPS input to 48 kHz. This 48 kHz signal contains a DC offset which is estimated by averaging the previous 32 audio values. The final output `audio_out` is computed by removing the DC offset and shifting the zero centered signal according the input volume. Since `ds_audio_offset` is a 34 bit value, the maximum number of bits the signal must be shifted to avoid overflow during assignment to the 8 bit audio value is 26 bits, therefore `audio_level` must be a 5 bit value.

```
audio_out <= ( (ds_audio_offset - avg) >>> ('d26 - audio_level) )
```

Human Interface (Chaney)

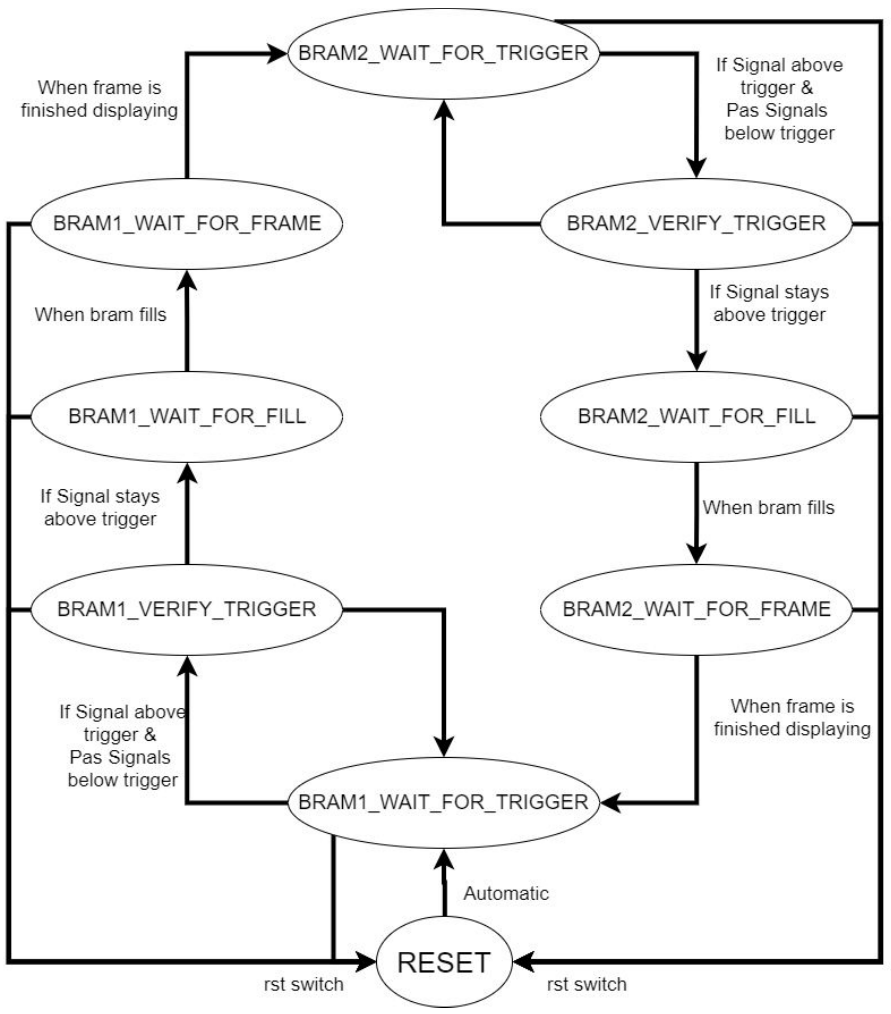
Overview

For human interfacing, our SDR has both a VGA monitor output and a laser cut wood piece with rotary encoders. The VGA monitor works at a resolution of 1024x768 at 60 Hz. The VGA monitor is able to display the incoming raw signal from the ADC or the demodulated audio. The VGA monitor can also be used to display the frequency information of our ADC signal, telling us what types of signals are available to our radio. This frequency spectrum is capped at the nyquist frequency divided by two, which in this case is 2.5 MHz. On this screen we are able to see where our current center frequency is set to, making it easy for users to adjust to signals on the screen. In this frequency display, a user could also zoom to different areas of the spectrum and gain more clarity. We also tried to implement a waterfall display, but were unsuccessful in getting it to work as expected.

Displaying a Signal

Trigger_Buffer.sv

To display a signal, we had to implement a somewhat complex state machine that wrote a signal between two BRAMs. We wouldn't need a complex state machine if we didn't allow our user to change the sample rate. Allowing the user to change the sampling rate lets users see signals of varying frequencies. The general process of this state machine is that it writes to one BRAM while the display grabs data from the other BRAM. Doing this allows the display to wait until a sample of any frequency captured before displaying it.


BRAM2_WAIT_FOR_TRIGGER
When frame is finished displaying
If Signal above trigger & Pas Signals below trigger
BRAM1_WAIT_FOR_FRAME
BRAM2_VERIFY_TRIGGER
When bram fills
If Signal stays above trigger
BRAM1_WAIT_FOR_FILL
BRAM2_WAIT_FOR_FILL
If Signal stays above trigger
When bram fills
BRAM1_VERIFY_TRIGGER
BRAM2_WAIT_FOR_FRAME
If Signal above trigger & Pas Signals below trigger
When frame is finished displaying
BRAM1_WAIT_FOR_TRIGGER
Automatic
RESET
rst switch
rst switch

Signal Display Block Diagram Overview

RESET: This is the default state when the FPGA is started. In this state we reset all the addresses to 0, the signal out to 0, and the past signal values to the trigger value (we do this so that all past values are filled before the trigger is evaluated). We go back to this state whenever the reset switch is flipped. It automatically transitions to BRAM1_WAIT_FOR_TRIGGER.

BRAM1_WAIT_FOR_TRIGGER: In this stage, we setup the display to read whatever signal snapshot is saved in BRAM2 (`signal_out <= data_from_frame2`). We do that by making the address for BRAM2 dependent on the hcount of the frame (`frame2_addr <= (hcount_in - 'd101)`). As you go to the right of the screen you grab later values of the BRAM. As we wait for the trigger, we write back to 20 past values. For the trigger to be satisfied, all past 20 values have to be below the trigger line and the current signal value has to be at or above the trigger line. An additional constraint is placed where the difference between the current signal and the past signal can't be too high (`(signal_in - past_signal) < 'd1000`). Having this does create issues for extremely high frequency signals, as you can jump from low to high in a matter of a few cycles. This was done to prevent noise and was considered a good tradeoff, as high frequency information is hard to visually comprehend. Once the trigger is met we move to BRAM1_VERIFY_TRIGGER and setup BRAM1 to be written to.

BRAM1_VERIFY_TRIGGER: In this stage we verify that the trigger is in fact correctly triggered. We do this by making sure that the signal value stays above the trigger line for `NEEDED_HIGH` amount of samples. Once this happens we switch to the state BRAM1_WAIT_FOR_FILL. During this time we write to BRAM1 in normal operation, which will be explained in BRAM1_WAIT_FOR_FILL.

BRAM1_WAIT_FOR_FILL: In this state we keep writing to BRAM1 until we capture the full snapshot for our display. To make the sampling period variable we have a counter and the address to BRAM1 is not changed until the counter reaches the wanted period number. BRAM1 becomes filled when the address gets to 821, as we are displaying less than 821 samples on the display.

BRAM1_WAIT_FOR_FRAME: Since BRAM1 is full, we can theoretically switch it so that BRAM1 is what the display is using and BRAM2 is being written to. However, we need to wait until the frame ends to switch the BRAMs. We know the frame is over when the vcount reaches 767.

Afterward, we repeat the same states as before but for BRAM2 being written to and BRAM1 being what is used for the display. The cycle continues over and over again, sending out the signal we want to display at any given hcount.

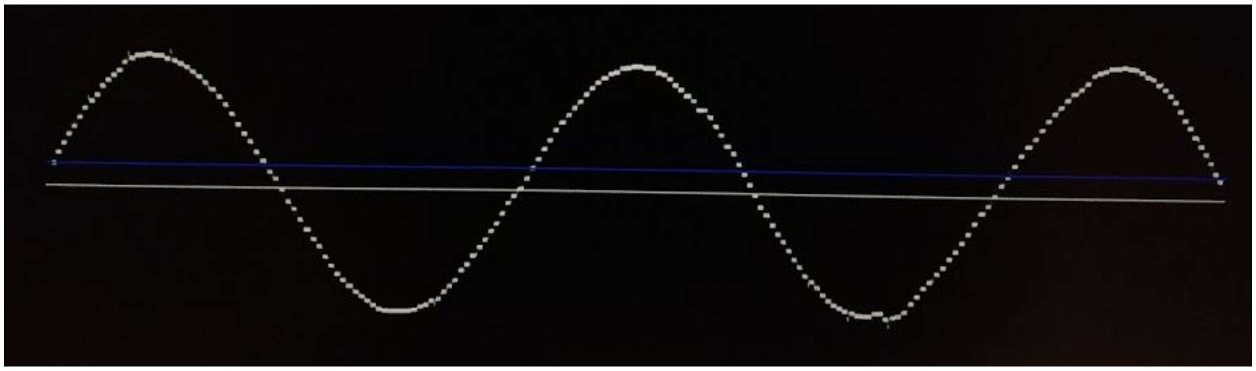
Function_Pixel_Logic.sv

When displaying a signal on the monitor, there are three separate components. Those components are the centerline, the function/signal itself, and the trigger line. The centerline shows what the length and 0 height scale are, as such it is just a white line across the screen. We declare the centerline as `{12{((vcount_in==384) & (hcount_in>100) & (hcount_in<923))}}`, meaning the 0 height line is at the vcount of 384 and the signal is displayed between an hcount of 101 to 922. The trigger line is very similar, but we have to scale it's height the same as we scale the signal height, which we will discuss later.

From Trigger_Buffer we are able to get whatever signal value we are supposed to display. This signal value should change based upon the hcount, as dictated in the Trigger_Buffer module. This means we only need to focus on the height of the function and make sure we are only displaying within a certain boundary.

We first have to scale the signal height so it is matched with the user input height_adjust. The concept of height_adjust is that it determined the highest possible pixel height. To make this so, we multiple our signal in with our height_adjust to form a new 24 bit value. We then bitshift this new value to the right by 11 bits so that the original height_adjust acts like a decimal multiplier. We do the exact same scaling for the trigger line so that we can display it. Displaying between demodulated audio and the raw ADC signal has different offsets. The raw ADC signal will have a 2.5 volt offset, as driven by the function generator or RF amplifier. The audio signal will have no offset since any DC bias in removed in the audio conditioning module.

Once we have the scaled heights in pixels, the logic becomes easy. The trigger line is exactly the same as the centerline, except it's height is offset by the scaled trigger height (`{4{((vcount_in==(12'sd384 - scaled_trigger_height)) & (hcount_in>100) & (hcount_in<923))}}`). The trigger line is also blue, as to differentiate it from the centerline. To display the actual signal/function we first have to make sure it is within the hcount range of the centerline. Afterward we see if our current vcount is equal to the centerline pixel height minus the scaled pixel height of the signal. To make the signal thicker we also check if it is one above or one below. If any of those conditions are met, it makes the pixel on the screen white. All in all, this results in a signal being shown that starts at the trigger line and continues until it has reached the end of the sample window

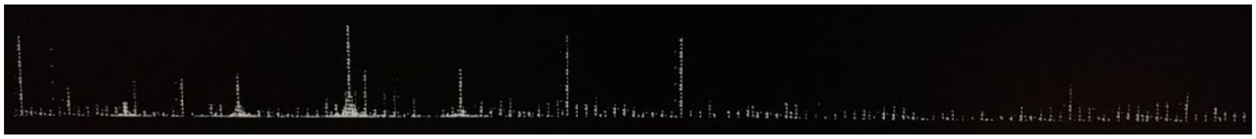

White horizontal line: Centerline
Blue horizontal line: Triggerline

FFT and Center Frequency Tuning (histogram.sv)

A Basic FFT

A radio is no good if you aren't able to see what types of signals you can tune to and know where you are currently tuned to. To be able to see what types of signals are available, we had to implement a fast fourier transform (FFT). This would let us see what type of frequencies were strong in the ADC signal. For actually implementing the FFT we modified the example code provided on the course website. The example code used a 4096 point FFT, which is fine for our purposes. Two big differences between our final code and the example code were that we did not oversample and we used a different clock domain. We wanted to be able to see a large spectrum, up to 2.5 MHz, which oversampling would ruin. Since our ADC interface is clocked at 100 MHz, our system clock would not be 104 MHz. In turn our `ADC_data_valid` was used in `fft_bram` to signify that a `fsample` was valid and that `fhead` be incremented.

Where most of our most important changes came into play were with the fft magnitude scaling and the actual histogram pixel generator. In the fft magnitude block, a `scaling` factor was used to differentiate between magnitudes. After playing around with the values I found that `12'b011111111000` proved to be the best scaling value and could differentiate between noise and peaks of data the best.


Full RF Spectrum FFT

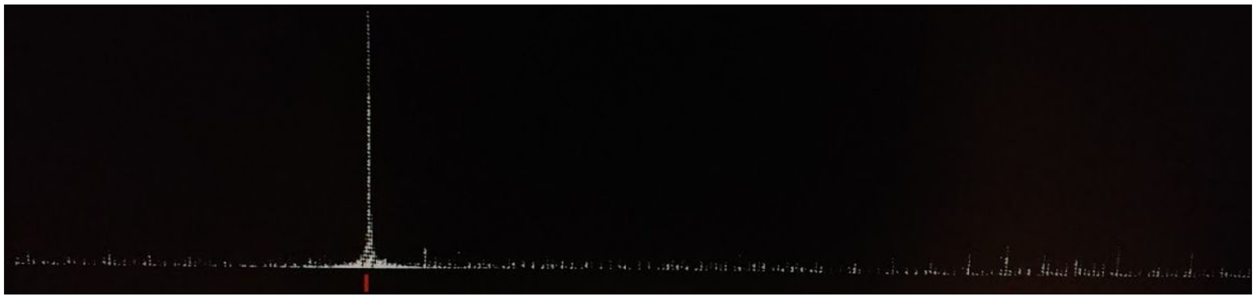
Our only initial changes to the histogram module were to move the FFT up the screen, more towards the middle, and to make the height higher for each individual bin. We accomplished the first by altering vheight such that it subtracted from a lower number, thus being higher up on the screen `(vheight = 10'd500 - vcount)`. The second was accomplished by making `hheight` be right shifted less, going from 7 to 4 (`hheight = vdata >> 4;`).

A Basic Frequency Marker

Making these small changes provided all of the effective functionality we needed to be able to see the frequency components of the signal from our ADC. From there we had to implement a marker that showed us what our current center frequency was. This would let us be able to tune the marker to different frequencies and hear different things.

The first step in determining where to put this marker would be to figure out how much frequency changes from pixel to pixel. Our screen takes up 1024 pixels horizontally, and in that span we cross half of our nyquist frequency, 2.5 MHz. That means that the frequency per pixel would be `'d2500000 >> 10` or around 2,441.

Our initial frequency always starts at 600 KHz. Dividing 600,000 by that number yields an initial hcount position of 246. Testing this out with a 600 KHz signals shows that this math is indeed correct.



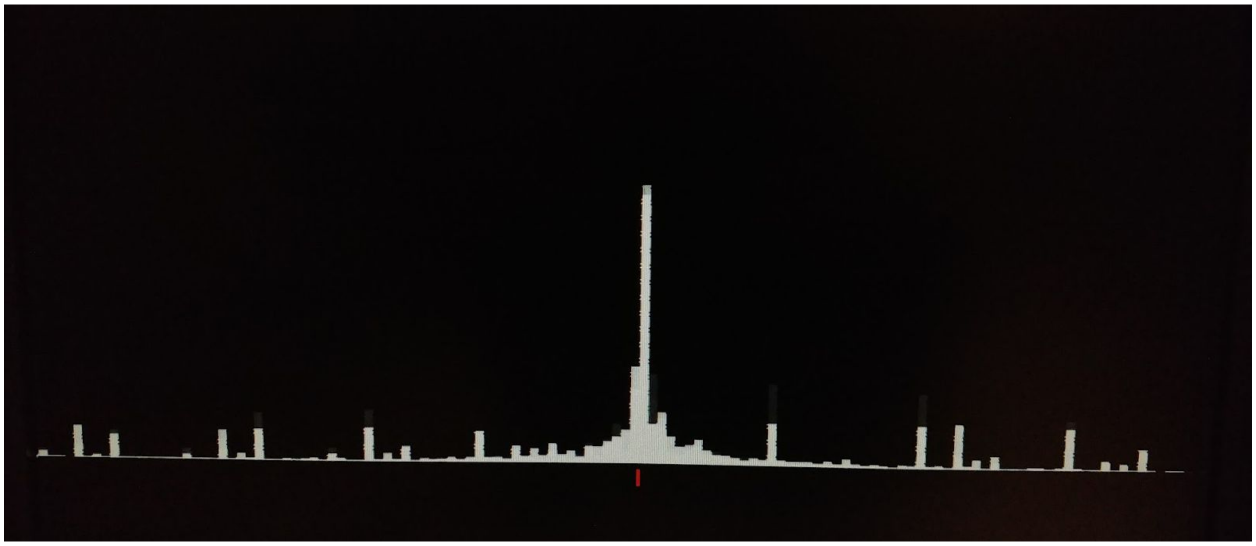
RF Spectrum FFT

From there all we had to do was adjust the position of this market based upon how our center frequency changed. Since our center frequency was mod 20, that meant we had to multiply our center frequency wire by 20 to get the real effective frequency. From there we just had to see if the difference between the current frequency and the new one was above this “spacer” value. A hard part about this though was distinguishing from negative and positive changes. For instance, center_freq - current_freq will be a small number if center_freq is larger, but if it is smaller instead, the number would overflow and be a large number. That would make a positive and negative frequency appear to be the same. To combat this we put an upper limit for how much the frequency could change in a single clock cycle, that being 100 KHz. This made sense for our system, as it would be impossible for a person to change their center frequency that fast.

The current center frequency and position is adjusted when a large enough discrepancy is seen. The addition and subtraction works 1 pixel per clock cycles. This means if someone jumped multiple pixels in one clock cycle, it would take more clock cycles to register that change. This was okay for the purposes of this project as a human cannot perceive this.

Zoom Functionality

Everything up to this point has created a frequency spectrum and allowed us to tune to different stations. As part of our goals we wanted to be able to zoom in on different parts of the FFT spectrum. The first part of this was to actually zoom and make the frequency window covered by the 1024 pixels shorter. This could be accomplished by bit shifting the address of the BRAM that gives us the FFT values to the right (`vaddr = (hcount[9:0] >> range)`). This would mean that a single address would last longer, 2 times longer in the case of a single bit shift and 4 times in the case of two bit shifts. This provided that functionality quite well.



Zoomed FFT

What was more complex was being able to change the window. Bit shifting like this made it so that we could zoom in on the smaller frequencies easier. To actually look at frequencies in the middle we had to add an offset (`vaddr = (hcount[9:0] >> range) + zoom_offset`). What made this more complex is how it changed the center frequency marker. The center frequency marker position also had to be bit shifted right with range to make it scale down too. The zoom offset, however, was separate from this and had to be done before the bit shifting (`display_position = ((current_position - zoom_offset) << range)`). The zoom offset also worked in the opposite direction for the center frequency line, as you move to the right the center frequency would move left.

Since the `display_position` wire could overflow and display when zooming in different domains where it shouldn't. We had to better establish the frequency window we were looking at and make sure the current center frequency was in the window. The minimum on the window would be determined by the pixel/frequency resolution and what our zoom offset was. The zoom offset was representative of how far away we were from a 0 frequency in terms of pixels, so the minimum frequency was just the product of the offset and the pixel/frequency resolution (`min_frequency_window = zoom_offset * spacer`). The max frequency of the window would be the minimum frequency of the window plus the total frequency we span in the screen (`max_frequency_window = min_frequency_window + max_frequency`). This max frequency span of the screen would be determined by the original max frequency bit shifted by the zoom amount (`max_frequency = 'd2500000 >> range;`). For example, if we our zoom factor was `2'b01` the max frequency span of the window would be 1.25 MHz.

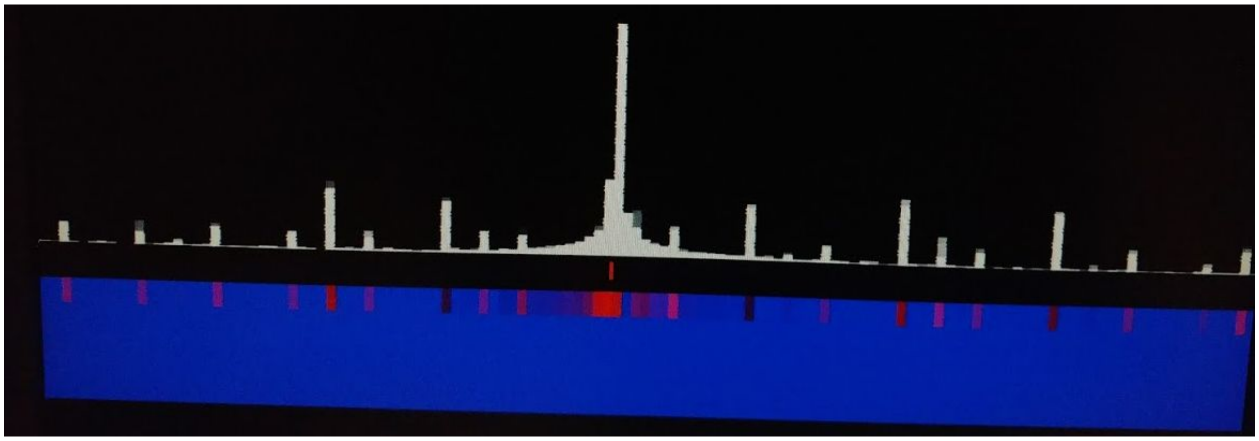
What worked so well with this solution is that we didn't have to change the calculations for how the center frequency marker position changed. The relative position of the different zooms and offset was calculated independent of the position of the marker when there is no zoom and offset.

Displaying the Intermediate Frequency (IF)

Since our radio is a superheterodyne, there is an IF that we operate at. We wanted to be able to display this IF on the frequency spectrum if need be. We added a special marker of a different color to show this when a certain switch it flipped. The original position of this IF doesn't change, with the hcount being 187. The same type of processing is applied to the IF (`if_position = ('d187 - zoom_offset) << range;`).

Waterfall Display

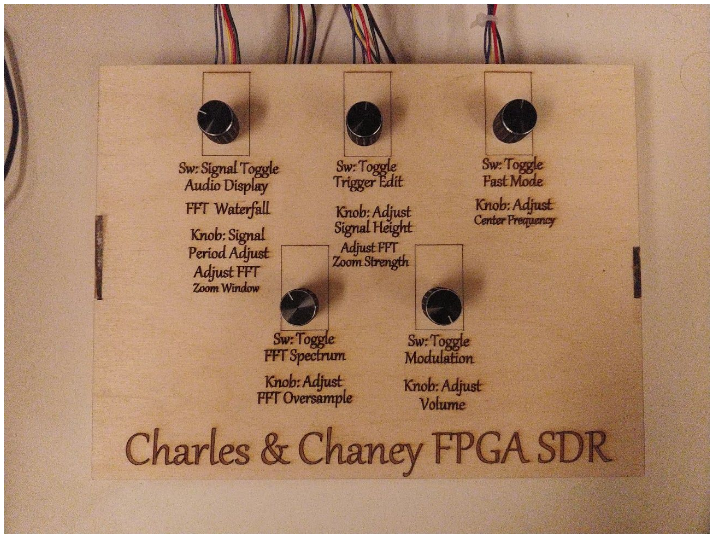
As part of our checklist we wanted to be able to have a waterfall display. We were not able to implement it fully. The concept of it was that we would set the zoom_offset to a number closer to the center frequency and set the zoom to the maximum. We would split the screen into 8 sections horizontally and 7 sections vertically. We would take the average of the areas by having an accumulator and bit shifting it then push the values down as they change. In our waterfall display we were not able to get the averaged values to work or flow down. We were able to test our color gradient in real time on the first section of pixels vertically. In the color gradient, cold colors signified low magnitude and warm colors were high magnitude.



Waterfall Implementation Test

User Input

All of the modules have talked about user inputs but not explained how they work. To make the project feel more like a real radio, we wanted to use knobs. At first we thought about using potentiometers but decided to use rotary encoders instead. We put the rotary encoders on a laser cut piece to make the display look nice.

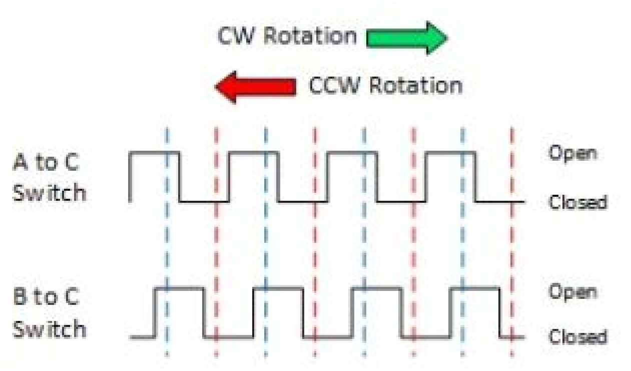


The rotary encoders had three different things they could do. They could turn clockwise, counter-clockwise, and have their switch be pressed. There are three pins important to this, those being clk, dt, and sw. All of these pins are active low. The simplest of the three is the switch, which has voltage go low whenever it is pressed.

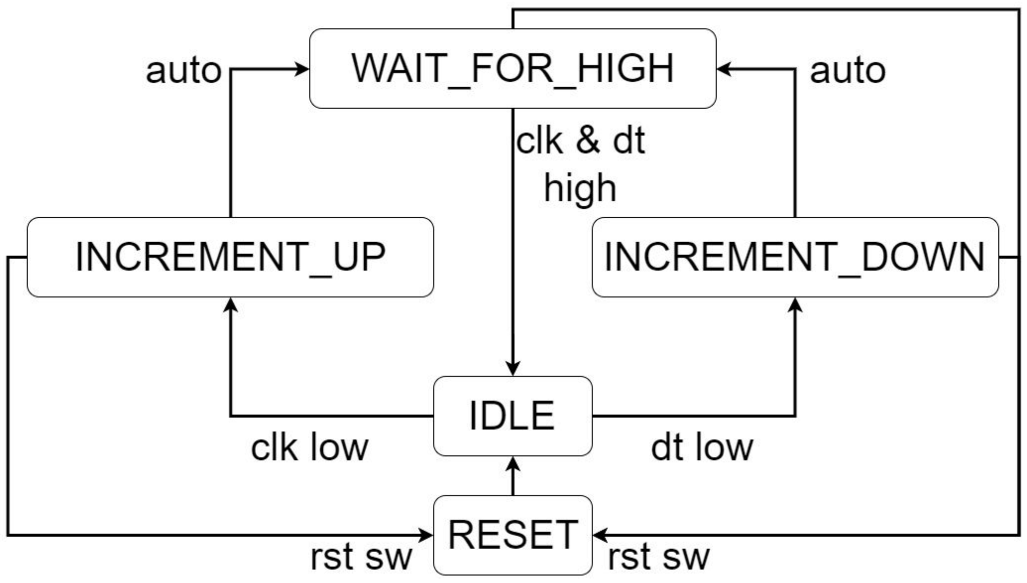
Generally we used the switches as a means of toggling between different modes. One of the switches, for example, toggled whether we wanted to see the FFT spectrum or if we wanted to see a signal be displayed. How we did this toggle was by creating a register (`reg is_fft = 0`) and storing the past value of the switch (`logic past_fft_value`). The top_level.sv would wait to see the switch go from high to low (`if(!encoder4_sw_db & past_fft_value ) begin`), and once that was done it would increment the register by 1 (`is_fft <= is_fft + 1;`). This would make the register 0 if it was 1 before or 1 if the register was 0 before. This created an effective toggle. These were all of the implemented toggles in top_level.sv:

- `Is_fast`
  - Fast mode for period and center frequency adjust
- `Modulation_select_sw`
  - AM or FM demodulation
  - Changed LED color on FPGA so user could see what mode they were on
- `Trigger_wanted`
  - Adjust signal height or trigger height
- `Is_fft`
  - Look at FFT spectrum or signal
- `Is_audio`
  - Look at demodulated audio or raw signal, only works if fft mode is off
- `Is_waterfall`
  - Look at regular FFT or waterfall display, only works if fft mode is on

The clk, and dt pins are a little more complex. Rotary encoders work through light sources and optical sensors. The light sources go through a track that can block light or not block light. The clk and dt pins have different tracks that cause them to go high and low at different times. If you turn the encoder clockwise you will find that the clk pin goes low first and then the dt pin does low. Once a turn is completed they both go high. If you turn counter-clockwise instead the dt pin goes low first. An important note is that these encoders can be finicky, so it was really important to synchronize and debounce them to the correct clock.



The general outline of a state machine for these switches is shown below. The idea is that we wait to see if either clk or dt pulses low first. If clk pulses low first then we increment up. If dt pulses low first then we increment down. We wait until both signals stabilize and go high before we wait to increment again.



Several versions of this were implemented in many different modules. I'll list them all briefly and talk about how each one is different.

- `Control_center_frequency.sv`
    - Maximum of `18'd250_000` and a minimum of `'d25_000`. Fast mode increments at `'d300` per tick and regular mode at `'d25` per tick.
- `Control_volume.sv`
    - Maximum of `'d26` and a minimum of `'d0` and changes at 1 per tick.
    - Has a special variable `is_max` to make sure that the volume does not get too loud. It makes sure the volume is below the bit shift point that would cause the number to overflow.
- `Control_zoom_magnitude.sv`

  - Maximum of `'d3` and a minimum of `'d0` and changes at 1 per tick.
  - Only works if fft mode is on and waterfall is off.
- `Control_zoom_window.sv`
  - Three separate state machines that depend on what the value of `zoom_magnitude` is.
  - Only works if fft mode is on and waterfall is off.
  - `Zoom_magnitude == 2'b00`
    - `Zoom_pos_out` is always 0
  - `Zoom_magnitude == 2'b01`
    - Maximum of `'d503` and a minimum of `'d0` and changes at `'d10` per tick.
  - `Zoom_magnitude == 2'b10`
    - Maximum of `'d779` and a minimum of `'d0` and changes at `'d20` per tick.
  - `Zoom_magnitude == 2'b11`
    - Maximum of `'d867` and a minimum of `'d0` and changes at `'d30` per tick.
- `Control_period.sv`
  - Maximum is `'d3950` and minimum is `'d0`.
  - If in fast mode it changes at `'d20` per tick and `'d1` per tick if not
  - Only works if fft mode is off.
- `Control_height`
  - Two separate state machines for whether audio is on or off
  - Audio off
    - Maximum of `'d390` and minimum of `'d5`.
    - Moves at `'d10` per tick.
  - Audio on
    - Maximum of `'d520` and minimum of `'d3`.
    - Moves at `'d20` per tick.
  - Only works if fft mode is off and trigger adjust is off.
- `Control_trigger_height.sv`
  - Two separate state machines for whether audio is on or off
  - Audio off
    - Maximum of `'d4000` and minimum of `'d5`.
    - Moves at `'d50` per tick.
  - Audio on
    - Maximum of `'d3970` and minimum of `'d33`.
    - Moves at `'d20` per tick.
  - Only works if fft mode is off and trigger adjust is on

Summary

All in all we were able to meet a good number of our goals for the project. Where we fell short was not being able to get the waterfall display to work or getting the front end properly put together. These issues might have been resolved a bit better had we started working on the project seriously earlier.

Getting our bandpass filter to work took much longer than expected, as we were expecting to implement a simple FIR filter as we had done in previous labs until we realized that given our tight timing restraints this would not be possible. This definitely proves that it is extremely important to have a really well thought out design before implementation, or at least allow for a lot of extra time in case issues like this are encountered. Although we did start later than planned, we were able to solve major issues such as these as we did allow extra time in our schedule. It just meant that we weren't able to get some of the more experimental components like an RF front end working.

Acknowledgments

We would like to think Gim and Joe for being great lecturers and also mentors throughout the project! We both put a lot of time into the project and enjoyed working on it, but we wouldn't have gotten it off the ground without Joe's help on the hardware side and also signal processing advice (we'd both only taken 6.003 / 16.002).

Appendix

All code can be found at the Github link https://github.com/cmll12/FPGA_SDR

```
module top_level(
        input clk_100mhz, //10 ns period
        input [15:0] sw, //reset switch
        input btnc, btnu, btnl, btnr, btnd,
        input [5:0] jb, //ADC data in
        input [7:1] ja, //ADC data in
        input [7:0] jc, //Encoders
        input [7:0] jd, //Encoders
        output logic ja_0, //ADC clk pin
        output logic led16_r,
        output logic led16_g,
        output logic led16_b,
        output logic [15:0] led,
        output logic aud_pwm,
        output logic aud_sd,
        output[3:0] vga_r,
        output[3:0] vga_b,
        output[3:0] vga_g,
        output vga_hs,
        output vga_vs
    );

    //system reset as sw[15]
    logic rst;
    assign rst = sw[15];


    //Creates both another 100mhz clock and a 65mhz clock.
    //A second 100mhz clock was needed because the original
    //clock is constrained to the clkdivider
    logic clk100mhz;
    logic clk_65mhz;
    clk_wiz_0 clkdivider(.clk_in1(clk_100mhz),
                      .clk_out1(clk100mhz),
                      .clk_out2(clk_65mhz));

    //A wire for each pin of an an encoder that we used. In
    //total we had 5 encoders with three wires being important
```

```
//for operation, those being clk, dt, and sw.
wire encoder1_clk;
wire encoder1_dt;
wire encoder1_sw;
wire encoder2_clk;
wire encoder2_dt;
wire encoder2_sw;
wire encoder3_clk;
wire encoder3_dt;
wire encoder3_sw;
wire encoder4_sw;
wire encoder5_clk;
wire encoder5_sw;
wire encoder5_dt;

//Each of the signals are synchronized to the 65 mhz clock
//based upon their PMOD input port. We chose 65 mhz because
//most modules that need the inputs from these devices run on
//65 mhz
synchronize encoder1_clk_synchronize(
   .clk(clk_65mhz),
   .in(jc[0]),
   .out(encoder1_clk));

synchronize encoder1_dt_synchronize(
   .clk(clk_65mhz),
   .in(jc[1]),
   .out(encoder1_dt));

synchronize encoder1_sw_synchronize(
   .clk(clk_65mhz),
   .in(jc[5]),
   .out(encoder1_sw));

synchronize encoder2_clk_synchronize(
   .clk(clk_65mhz),
   .in(jc[2]),
   .out(encoder2_clk));

synchronize encoder2_dt_synchronize(
   .clk(clk_65mhz),
   .in(jc[3]),
   .out(encoder2_dt));
```

```
synchronize encoder2_sw_synchronize(
    .clk(clk_65mhz),
    .in(jc[4]),
    .out(encoder2_sw));

synchronize encoder3_clk_synchronize(
    .clk(clk_65mhz),
    .in(jc[6]),
    .out(encoder3_clk));

synchronize encoder3_dt_synchronize(
    .clk(clk_65mhz),
    .in(jc[7]),
    .out(encoder3_dt));

synchronize encoder3_sw_synchronize(
    .clk(clk_65mhz),
    .in(jd[2]),
    .out(encoder3_sw));

synchronize encoder4_sw_synchronize(
    .clk(clk_65mhz),
    .in(jd[3]),
    .out(encoder4_sw));

synchronize encoder5_clk_synchronize(
    .clk(clk_65mhz),
    .in(jd[4]),
    .out(encoder5_clk));

synchronize encoder5_sw_synchronize(
    .clk(clk_65mhz),
    .in(jd[0]),
    .out(encoder5_sw));

synchronize encoder5_dt_synchronize(
    .clk(clk_65mhz),
    .in(jd[5]),
    .out(encoder5_dt));

//After synchronizing the signals we have to
//debounce them. These wires corespond to the
```

```
//debounces signal afterward
wire encoder1_clk_db;
wire encoder1_dt_db;
wire encoder1_sw_db;
wire encoder2_clk_db;
wire encoder2_dt_db;
wire encoder2_sw_db;
wire encoder3_clk_db;
wire encoder3_dt_db;
wire encoder3_sw_db;
wire encoder4_sw_db;
wire encoder5_clk_db;
wire encoder5_sw_db;
wire encoder5_dt_db;

//All of the encoder signals are debounced
debounce encoder1_clk_debounce(
   .reset_in(rst),
   .clock_in(clk_65mhz),
   .noisy_in(encoder1_clk),
   .clean_out(encoder1_clk_db));

debounce encoder1_dt_debounce(
   .reset_in(rst),
   .clock_in(clk_65mhz),
   .noisy_in(encoder1_dt),
   .clean_out(encoder1_dt_db));

debounce encoder1_sw_debounce(
   .reset_in(rst),
   .clock_in(clk_65mhz),
   .noisy_in(encoder1_sw),
   .clean_out(encoder1_sw_db));

debounce encoder2_clk_debounce(
   .reset_in(rst),
   .clock_in(clk_65mhz),
   .noisy_in(encoder2_clk),
   .clean_out(encoder2_clk_db));

debounce encoder2_dt_debounce(
   .reset_in(rst),
   .clock_in(clk_65mhz),
    .noisy_in(encoder2_dt),
    .clean_out(encoder2_dt_db));

debounce encoder2_sw_debounce(
    .reset_in(rst),
    .clock_in(clk_65mhz),
    .noisy_in(encoder2_sw),
    .clean_out(encoder2_sw_db));

debounce encoder3_clk_debounce(
    .reset_in(rst),
    .clock_in(clk_65mhz),
    .noisy_in(encoder3_clk),
    .clean_out(encoder3_clk_db));

debounce encoder3_dt_debounce(
    .reset_in(rst),
    .clock_in(clk_65mhz),
    .noisy_in(encoder3_dt),
    .clean_out(encoder3_dt_db));

debounce encoder3_sw_debounce(
    .reset_in(rst),
    .clock_in(clk_65mhz),
    .noisy_in(encoder3_sw),
    .clean_out(encoder3_sw_db));

debounce encoder4_sw_debounce(
    .reset_in(rst),
    .clock_in(clk_65mhz),
    .noisy_in(encoder4_sw),
    .clean_out(encoder4_sw_db));

debounce encoder5_clk_debounce(
    .reset_in(rst),
    .clock_in(clk_65mhz),
    .noisy_in(encoder5_clk),
    .clean_out(encoder5_clk_db));

debounce encoder5_sw_debounce(
    .reset_in(rst),
    .clock_in(clk_65mhz),
    .noisy_in(encoder5_sw),
    .clean_out(encoder5_sw_db));

debounce encoder5_dt_debounce(
    .reset_in(rst),
    .clock_in(clk_65mhz),
    .noisy_in(encoder5_dt),
    .clean_out(encoder5_dt_db));

//This is a general code block used many times.
//The purpose is to toggle a mode of operation
//by clicking the switch on an encoder. This one
//is meant to change whether fast mode is enabled
reg is_fast = 0;
//Used to compare against past value of operation.
logic past_fast_value;
always_ff @(posedge clk_65mhz) begin
   //This makes sure we go from a high to a low value.
   //The encoders are active low.
   if(!encoder3_sw_db & past_fast_value ) begin
      //If this happens then we need to change mode.
      //By adding 1 the bit will either overflow or not,
      //toggling between the modes
      is_fast <= is_fast + 1;
   end
   //Write past value so we can see the transition from
   //high to low.
   past_fast_value <= encoder3_sw_db;
end

//Used to select if we are on AM or FM demodulation.
reg modulation_select_sw = 0;
logic past_modulation_value;
always_ff @(posedge clk_65mhz) begin
   if(!encoder5_sw_db & past_modulation_value ) begin
      modulation_select_sw <= modulation_select_sw + 1;
   end
   past_modulation_value <= encoder5_sw_db;
end

//Used to see if we want to change or trigger height
//or if we want to change our signal height.
reg trigger_wanted = 0;
logic past_trigger_value;
```

```
always_ff @(posedge clk_65mhz) begin
  if(!encoder2_sw_db & past_trigger_value ) begin
    trigger_wanted <= trigger_wanted + 1;
  end
  past_trigger_value <= encoder2_sw_db;
end

//Used to toggle the FFT or signal display
reg is_fft = 0;
logic past_fft_value;
always_ff @(posedge clk_65mhz) begin
  if(!encoder4_sw_db & past_fft_value ) begin
    is_fft <= is_fft + 1;
  end
  past_fft_value <= encoder4_sw_db;
end

//Used only if we aren't display the FFT. This
//toggles whether we want to see the raw ADC in
//signal or if we want to see the demodulated
//audio signal.
reg is_audio = 0;
logic past_audio_value;
always_ff @(posedge clk_65mhz) begin
  //Relies on previous is_fft values so we don't switch
  //it if we are in the wrong display mode.
  if(!encoder1_sw_db & past_audio_value & !is_fft) begin
    is_audio <= is_audio + 1;
  end
  past_audio_value <= encoder1_sw_db;
end

//Used only if we are displaying the FFT. This toggles
//whether we want to do the narow band waterfall or not.
//Currently the waterfall doesn't work.
reg is_waterfall = 0;
logic past_waterfall_value;
always_ff @(posedge clk_65mhz) begin
  //Relies on previous is_fft values so we don't switch
  //it if we are in the wrong display mode.
  if(!encoder1_sw_db & past_waterfall_value & is_fft) begin
    is_waterfall <= is_waterfall + 1;
  end
  past_waterfall_value <= encoder1_sw_db;
end

//LED lights to help show what modulation scheme we are on
always_comb begin
  //If high, this means we are demodulating FM.
  if(modulation_select_sw) begin
    //If sw[14] is high, it means we are demodulating
    //a wide band FM. Else, it is narrow band FM.
    if(sw[14]) begin
      //Sets color blue
      led16_r <= 0;
      led16_g <= 0;
      led16_b <= 1;
    end else begin
      //Sets color green
      led16_r <= 0;
      led16_g <= 1;
      led16_b <= 0;
    end
  //We are demodulating AM
  end else begin
    //Sets color red
    led16_r <= 1;
    led16_g <= 0;
    led16_b <= 0;
  end
end

//Wires for VGA display
wire [10:0] hcount;      // Pixel on current line
wire [9:0] vcount;       // Line number
wire hsync, vsync, blank; // Signals to output on connector
reg [11:0] rgb;          // Pixel out color

//Given module that outputs the correct timing for each signal
xvga xvga1(.vclock_in(clk_65mhz),.hcount_out(hcount),.vcount_out(vcount),
    .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));

//Setup speaker output
assign aud_sd = 1;

//ADC variables
```

```
logic [11:0] sample;    //The actual output of the ADC
logic ADC_data_valid;   //Goes high if data is valid
logic B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12;   //Specific raw bits out
logic ADC_clk_gen;      //The clock we send out to get the ADC working
logic OTR;              //Overflow bit

//Map ADC pins to FPGA PMOD pins
assign B1  = ja[3];
assign B2  = ja[5];
assign B3  = ja[7];
assign B4  = jb[1];
assign B5  = jb[3];
assign B6  = jb[5];
assign B7  = jb[4];
assign B8  = jb[2];
assign B9  = jb[0];
assign B10 = ja[6];
assign B11 = ja[4];
assign B12 = ja[2];
assign OTR = ja[1];

assign ja_0 = ADC_clk_gen; //10MHz clock output from ADC interface

//Variable used for the center frequency
logic [17:0] center_freq_div_20;

//Module that adjusts the center frequency based on the knob inputs
control_center_frequency adjust_frequency(
   .clk(clk_65mhz),
   .right(encoder3_clk_db),
   .left(encoder3_dt_db),
   .reset(rst),
   .center_frequency_out(center_freq_div_20),
   .is_fast(is_fast));


logic [4:0] sw_audio;              //Determine audio level
logic signed [7:0] DAC_audio_in;   //Audio sent to DAC

//Sets the volume based on the knob inputs and the current DAC value
control_volume(
        .clk(clk_65mhz),
        .up(encoder5_clk_db),
          .down(encoder5_dt_db),
          .reset(rst),
          .DAC_in(DAC_audio_in),
          .volume_out(sw_audio));
  //------------------------------------------------------------------


  //Interface with AD9220
  ADC_Interface AD9220 (.clk_100mhz(clk100mhz),.rst(rst),.sample_offset(sample),
                  .ADC_data_valid(ADC_data_valid),.ADC_clk(ADC_clk_gen),.B1(B1),.B2(B2),
                  .B3(B3),.B4(B4),.B5(B5),.B6(B6),.B7(B7),.B8(B8),.B9(B9),.B10(B10),
                  .B11(B11),.B12(B12),.out_of_range(OTR));

  //Local Oscillator
  logic [11:0] LO_out;
  Local_Oscillator LO (.rst(rst), .clk_in(clk100mhz),.center_freq_div_20(center_freq_div_20),
                    .LO_out(LO_out));

  //Mixer
  logic [23:0] IF_out;
  Mixer sample_LO_mixer (.in_a(sample),.in_b(LO_out),.p_out(IF_out));

  //convert mixer IF out to signed integer
  logic signed [23:0] IF_signed;
  assign IF_signed = {~IF_out[23],IF_out[22:0]};

  //FM Demodulation Stage 1 (BP filter and derivative)
  logic signed [33:0] FM_stage_1_out;
  logic FM_sample_ready;

  FM_demod_stage_1 FM_stage_1
(.clk(clk100mhz),.rst(rst),.IF_in(IF_signed),.IF_data_valid(ADC_data_valid),

.FM_BP_width(sw[14]),.FM_derivative_out(FM_stage_1_out),.FM_data_valid(FM_sample_read
y));

  //end FM demod stage 1 ----------------------------------------

  //AM Demodulation Stage 1 (BP filter)
  //AM Bandpass Filter

  //section 1 ----------------------------------------
  //initialize coeffs
```

```
   parameter N = 3;
   logic signed [17:0] b1 [(N-1):0]; //N b feedforward coeffs [b(N-1)...b0), unpacked array
   logic signed [17:0] a1 [(N-2):0]; //N-1 feedback coeffs [a(N-1)...a1], unpacked array

   assign b1 [(N-1):0] = '{-18'sd65536,18'sd0,18'sd65536}; //b coeff MATLAB: [1,0,-1]
   assign a1 [(N-2):0] = '{18'sd65380,-18'sd125527}; //a coeff MATLAB:

   logic signed [33:0] filt_sec_1_out;
   logic sec_1_ready;

   //triggers on ADC_sample_valid
   AM_BP_Filter #(.N(N)) AM_BP_sec_1 (.clk_in(clk100mhz),.rst(rst),.b(b1),.a(a1),

.sample_ready(ADC_data_valid),.sample(IF_signed),.filt_out(filt_sec_1_out),.filt_valid(sec_1_re
ady));

   //section 2 -----------------------------------------
   //same N as section 1

   logic signed [23:0] filt_sec_2_in;
   //divide output from filter section 1 by 2^11 to fit 24 bit input parameter
   assign filt_sec_2_in = (filt_sec_1_out>>>11);

   //initialize coeffs
   logic signed [17:0] b2 [(N-1):0]; //N b feedforward coeffs [b(N-1)...b0), unpacked array
   logic signed [17:0] a2 [(N-2):0]; //N-1 feedback coeffs [a(N-1)...a1], unpacked array

   assign b2 [(N-1):0] = '{-18'sd65536,18'sd0,18'sd65536}; //b coeff MATLAB: [1,0,-1]
   assign a2 [(N-2):0] = '{18'sd65383,-18'sd125680}; //a coeff MATLAB:

   logic signed [33:0] filt_sec_2_out;
   logic sec_2_ready;

   AM_BP_Filter #(.N(N)) AM_BP_sec_2 (.clk_in(clk100mhz),.rst(rst),.b(b2),.a(a2),

.sample_ready(sec_1_ready),.sample(filt_sec_2_in),.filt_out(filt_sec_2_out),.filt_valid(sec_2_rea
dy));

   //section 3 -----------------------------------------
   //same N as section 1

   logic signed [23:0] filt_sec_3_in;
   //divide output from filter section 1 by 2^11 to fit 24 bit input parameter
```

```
    assign filt_sec_3_in = (filt_sec_2_out>>>11);

    //initialize coeffs
    logic signed [17:0] b3 [(N-1):0]; //N b feedforward coeffs [b(N-1)...b0), unpacked array
    logic signed [17:0] a3 [(N-2):0]; //N-1 feedback coeffs [a(N-1)...a1], unpacked array

    assign b3 [(N-1):0] = '{-18'sd65536,18'sd0,18'sd65536}; //b coeff MATLAB: [1,0,-1]
    assign a3 [(N-2):0] = '{18'sd65227,-18'sd125456}; //a coeff MATLAB:

    logic signed [33:0] AM_stage_1_out;
    logic AM_sample_ready;

    AM_BP_Filter #(.N(N)) AM_BP_sec_3 (.clk_in(clk100mhz),.rst(rst),.b(b3),.a(a3),

.sample_ready(sec_2_ready),.sample(filt_sec_3_in),.filt_out(AM_stage_1_out),.filt_valid(AM_sa
mple_ready));

    //end AM demod stage 1 ------------------------------------------

    //AM and FM Demodulation stage 2 (peak detect and audio condition)
    //Modulation_select_sw, sw[5], determines which demod to use (0 = AM, 1 = FM)
    logic signed [33:0] peak_detect_sample_in;
    logic peak_detect_sample_ready;
    always_comb begin
        if (!modulation_select_sw) begin
            peak_detect_sample_ready = AM_sample_ready;
            peak_detect_sample_in = AM_stage_1_out;
        end else begin
            peak_detect_sample_ready = FM_sample_ready;
            peak_detect_sample_in = FM_stage_1_out;
        end
    end

    //Peak Detect and Hold
    //magnitude of peak values of signal
    logic [34:0] peak_values;
    logic peak_ready;

    Peak_detect_hold AM_peak_detect
(.clk(clk100mhz),.rst(rst),.sample_ready(peak_detect_sample_ready),

.sample_in(peak_detect_sample_in),.peak_value(peak_values),.sample_ready_out(peak_ready
));
```

```
    //Audio Condition
    //output to DAC module
    AM_audio_condition condition_AM_for_DAC
(.clk(clk100mhz),.rst(rst),.audio_offset(peak_values>>2),.audio_level(sw_audio),
                        .audio_out(DAC_audio_in));

    //ila --------------------
    fm_stage_1_ila ila_fm_stage_1
(.clk(clk100mhz),.probe0(filt_sec_6_out),.probe1(filt_sec_5_out));
    //----------------------

    //end demod stage 2 ----------------------------------------

    //Drive DAC with demodulated audio
    logic pwm_val;
    DAC_stuff (.clk_in(clk100mhz), .rst_in(rst), .level_in({~DAC_audio_in[7],DAC_audio_in[6:0]}),
.pwm_out(pwm_val));
    assign aud_pwm = pwm_val?1'bZ:1'b0;


    //----------------------------------------------------------------
    //VGA Stuff

    //The height_adjust is used to see what relative height
    //is wanted when displaying signals
    logic [11:0] height_adjust;
    control_height my_height(
                .clk(clk_65mhz),
                .up(encoder2_clk_db),
                .down(encoder2_dt_db),
                .reset(rst),
                .sw(trigger_wanted),
                .is_audio(is_audio),
                .is_fft(is_fft),
                .height_out(height_adjust));

    //The trigger_adjust is used to set the trigger level
    //of the signals.
    logic [11:0] trigger_adjust;
    control_trigger_height my_trigger(
                .clk(clk_65mhz),
                .up(encoder2_clk_db),
                .down(encoder2_dt_db),
                .reset(rst),
                .sw(trigger_wanted),
                .is_audio(is_audio),
                .is_fft(is_fft),
                .height_out(trigger_adjust));

//This sets the period we wait between taking samples.
//Having a higher period lets people see lower frequency
//signals
logic [11:0] period;
control_period my_period(
                .clk(clk_65mhz),
                .is_fast(is_fast),
                .right(encoder1_clk_db),
                .left(encoder1_dt_db),
                .reset(rst),
                .is_fft(is_fft),
                .period_out(period));

//In the case where we want to display a signal, we
//need to decide between the demodulated audio or the
//raw ADC signal
logic [11:0] signal_to_display;
always_comb begin
  //Sets correct signal based on what the user wants
  //to display
  if(is_audio) begin
    signal_to_display = {DAC_audio_in, 4'b0};
  end else begin
    signal_to_display = sample;
  end
end

//The display signal is what actually gets sent to the
//display. Why this is different is because we want to
//buffer our signal between two brams. This lets us see
//signals of varying frequencies.
logic [11:0] display_signal;
trigger_buffer my_buffer(.clock_in(clk_65mhz),.reset_in(rst),
                         .signal_in(signal_to_display),
                         .trigger_height(trigger_adjust),
                    .hcount_in(hcount),.vcount_in(vcount),
                    .period(period),
                    .signal_out(display_signal));

//Based on the display signal, this module gives us what
//the pixel color should be for any given point on the screen.
wire [11:0] pixel_function;
function_pixel_logic plot(.vclock_in(clk_65mhz),.reset_in(rst),
          .height_adjust(height_adjust),
          .signal_in(display_signal),
          .trigger_height(trigger_adjust),
          .hcount_in(hcount),.vcount_in(vcount),
          .is_audio(is_audio),
          .pixel_out(pixel_function));

//A border on the display is created. This is there to auto
//adjust the display easier
wire border = (hcount==0 | hcount==1023 | vcount==0 | vcount==767 |
            hcount == 512 | vcount == 384);

//Registers for the blanking, horizontal sync, and vertical sync signals.
//This is here in case there is some pipelining.
reg b,hs,vs;


//FFT Stuff
// INSTANTIATE SAMPLE FRAME BLOCK RAM
// This 16x4096 bram stores the frame of samples
// The read port is read by the bram_to_fft module and sent to the fft.
logic fwe;             // Whether or not we want to write to the bram
logic [11:0] fhead = 0; // Frame head - a pointer to the write point, works as circular buffer
logic [15:0] fsample;   // The sample data from the ADC
logic [11:0] faddr;     // Frame address - The read address, controlled by bram_to_fft
logic [15:0] fdata;     // Frame data - The read data, input into bram_to_fft
fft_bram bram1 (
   .clka(clk100mhz),
   .wea(fwe),
   .addra(fhead),
   .dina(fsample),
   .clkb(clk100mhz),
   .addrb(faddr),
   .doutb(fdata));
```

```
  //If the ADC is valid we want to icrement fhead to move to the next
  //position. Will overflow with time.
  always_ff @(posedge clk100mhz) begin
    if (ADC_data_valid) begin
      fhead <= fhead + 1;
    end
  end
  //Write only when we finish a valid sample.
  assign fwe = ADC_data_valid;

  //Switch the input into the bram between the IF and the signal.
  //Used for demonstration purposes.
  always_comb begin
    if(sw[1]) begin
      fsample = IF_out[23:10];
    end else begin
      fsample = {sample, 2'b0};
    end
  end

  //SAMPLE FRAME BRAM READ PORT SETUP
  //For this project, we just need to display the FFT on 60Hz video, so let's only send the frame
of samples
  //once every 60Hz.

  //Synchronize the vsync to 100mhz since it is on 65mhz
  logic vsync_100mhz, vsync_100mhz_pulse;
  synchronize vsync_synchronize(
     .clk(clk100mhz),
     .in(vsync),
     .out(vsync_100mhz));

  //Since vsync goes low when active, we must invert to make it a pulse.
  level_to_pulse vsync_ltp(
     .clk(clk100mhz),
     .level(~vsync_100mhz),
     .pulse(vsync_100mhz_pulse));

  // INSTANTIATE BRAM TO FFT MODULE
  // This module handles the magic of reading sample frames from the BRAM whenever start is
asserted,
  // and sending it to the FFT block design over the AXI-stream interface.
  logic last_missing; // All these are control lines to the FFT block design
```

```
    logic [31:0] frame_tdata;
    logic frame_tlast, frame_tready, frame_tvalid;
    bram_to_fft bram_to_fft_0(
        .clk(clk100mhz),
        .head(fhead),
        .addr(faddr),
        .data(fdata),
        .start(vsync_100mhz_pulse),
        .last_missing(last_missing),
        .frame_tdata(frame_tdata),
        .frame_tlast(frame_tlast),
        .frame_tready(frame_tready),
        .frame_tvalid(frame_tvalid)
    );

    // This is the FFT module, implemented as a block design with a 4096pt, 16bit FFT
    // that outputs in magnitude by doing sqrt(Re^2 + Im^2) on the FFT result.
    // It's fully pipelined, so it streams 4096-wide frames of frequency data as fast as
    // you stream in 4096-wide frames of time-domain samples.
    logic [23:0] magnitude_tdata; // This output bus has the FFT magnitude for the current index
    logic [11:0] magnitude_tuser; // This represents the current index being output, from 0 to 4096
    logic magnitude_tlast, magnitude_tvalid;
    fft_mag fft_mag_i(
        .clk(clk100mhz),
        .event_tlast_missing(last_missing),
        .frame_tdata(frame_tdata),
        .frame_tlast(frame_tlast),
        .frame_tready(frame_tready),
        .frame_tvalid(frame_tvalid),
        .scaling(12'b011111111000),
        .magnitude_tdata(magnitude_tdata),
        .magnitude_tlast(magnitude_tlast),
        .magnitude_tuser(magnitude_tuser),
        .magnitude_tvalid(magnitude_tvalid));

    // Let's only care about the range from index 0 to 1023, which represents frequencies 0 to
omega/2
    // where omega is the nyquist frequency (sample rate / 2)
    logic in_range = ~|magnitude_tuser[11:10]; // When 13 and 12 are 0, we're on indexes 0 to
1023

    // INSTANTIATE HISTOGRAM BLOCK RAM
    // This 16x1024 bram stores the histogram data.
// The write port is written by process_fft.
// The read port is read by the video outputter or the SD care saver
// Assign histogram bram read address to histogram module unless saving
logic [9:0] haddr; // The read port address
logic [15:0] hdata; // The read port data
histogram_bram bram2 (
    .clka(clk100mhz),
    .wea(in_range & magnitude_tvalid),  // Only save FFT output if in range and output is valid
    .addra(magnitude_tuser[9:0]),       // The FFT output index, 0 to 1023
    .dina(magnitude_tdata[15:0]),       // The actual FFT magnitude
    .clkb(clk100mhz),  // input wire clkb used to be clk_65mhz
    .addrb(haddr),     // input wire [9 : 0] addrb
    .doutb(hdata)      // output wire [15 : 0] doutb
);

//This module controls how much we zoom in on the FFT.
logic [1:0] fft_zoom_mag;
control_zoom_magnitude my_zoom(
    .clk(clk_65mhz),
    .up(encoder2_clk_db),
    .down(encoder2_dt_db),
    .reset(rst),
    .is_fft(is_fft),
    .is_waterfall(is_waterfall),
    .zoom_out(fft_zoom_mag));

//This module controls the offset where we are zooming
//in on. This in effect determine the windows of our zoom.
logic [9:0] zoom_offset;
control_zoom_window my_zoom_window(
    .clk(clk_65mhz),
    .up(encoder1_clk_db),
    .down(encoder1_dt_db),
    .reset(rst),
    .is_fft(is_fft),
    .is_waterfall(is_waterfall),
    .zoom_magnitude(fft_zoom_mag),
    .zoom_pos_out(zoom_offset));

//Module that outputs the pixel for the FFT.
//Includes other logic for the center frequency line
//and waterfall display.
logic [12:0] hist_pixel;
logic [1:0] hist_range;
histogram fft_histogram(
    .clk(clk_65mhz),
    .rst(rst),
    .hcount(hcount),
    .vcount(vcount),
    .blank(blank),
    .range_in(fft_zoom_mag), // How much to zoom on the first part of the spectrum
    .vaddr(haddr),
    .vdata(hdata),
    .freq(center_freq_div_20),
    .is_if(sw[1]),
    .is_waterfall(is_waterfall),
    .zoom_offset_in(zoom_offset),
    .pixel(hist_pixel));

// VGA OUTPUT
always_ff @(posedge clk_65mhz) begin
  //No pipelining so the signals are the same
  hs <= hsync;
  vs <= vsync;
  b <= blank;
  //If we want to display the fft the pixel is
  //whaterver we get from the histogram module
  if(is_fft) begin
     rgb <= hist_pixel;
  end else begin
     //otherwise, if sw[0] is on we want to show
     //our border for display calibration
     if (sw[0]) begin
        rgb <= {12{border}};
     //If not, then we display whatever signal
     end else begin
        rgb <= pixel_function;
     end
  end
end

//Assign connector values accordingly.
assign vga_r = ~b ? rgb[11:8]: 0;
assign vga_g = ~b ? rgb[7:4] : 0;
assign vga_b = ~b ? rgb[3:0] : 0;
```

```
   assign vga_hs = ~hs;
   assign vga_vs = ~vs;
endmodule

module ADC_Interface(
      input clk_100mhz, //10 ns period
      input rst, //system reset
      output logic [11:0] sample_offset, //most recent sample, 0-4095
      output logic ADC_data_valid, //triggered high at 10 MHz, otherwise off
      //ADC I/O
      output logic ADC_clk, //ADC clk pin
      input B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,//bits 1-12 from ADC
      input out_of_range
   );

  //format ADC values in array, including out of range bit
   logic [12:0] raw_values;
   assign raw_values[12:0] = {out_of_range, B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12};

   //10 mhz clk (100 ns) with 50% duty cycle with rising edge synced with 100mHz clk
   //rising edge. ADC sample rate = 10 MSPS
   logic clk_10mhz;
   logic [3:0] clk_count;

   assign ADC_clk = clk_10mhz;

   //trigger every time clk_10mhz goes high while creating 50% DC 10mhz clk
   logic trigger;
   always_comb begin
      if (!rst) begin
         if (clk_count <= 4) clk_10mhz = 1;
         else clk_10mhz = 0;
      end else
         clk_10mhz = 0;
   end
   always_ff @(posedge clk_100mhz) begin
      if (!rst) begin
         //reset clk every 10 counts of 100mHz clk, otherwise increment
         if (clk_count == 9) begin
            clk_count <= 0;
            trigger <= 1;
         end else begin
            clk_count <= clk_count + 1;
            trigger <= 0;
        end
    end else begin
        clk_count <= 9;
        trigger <=0;
    end //end else
end //end always_ff

//sync ADC pins by passing input through 2 registers
//registers for syncing ADC input data
logic [12:0] prev_data;
logic [12:0] sync_data;

always_ff @(posedge clk_100mhz) begin
    if (rst) begin
        prev_data <= 13'b0;
        sync_data <= 13'b0;
    end else begin
        //sync all 12 input pins with 2 registers
        for (int i=0; i<=12; i=i+1) begin
            prev_data[i] <= raw_values[i];
            sync_data[i] <= prev_data[i];
        end //for
    end //rst else
end //always_ff

//updates ouput value once ADC data has become valid
//AND passes through 2 registers (waits 10 clk periods for
//data valid, + 2 clk periods for sync registers. So delay_period = 2,
//which grabs value at 10 ns after last ADC clk high
//Asserts ADC_data_valid once new data is valid
parameter delay_period = 'd2;
logic [3:0] count;
always_ff @(posedge clk_100mhz) begin
    if (rst) begin
        count <= delay_period + 1; //ensures data won't become valid until after first trigger
        sample_offset[11:0] <= 12'h000;
        ADC_data_valid <= 0;
    end else begin
        if (trigger) begin
            count <= 1;
        end else begin
            //once waited for 2 clk periods...
            if (count == delay_period) begin
               //assert data valid
               ADC_data_valid <= 1;
               //ouput ADC synced data if not out of range, otherwise output max value
               sample_offset[11:0] <= !sync_data[12] ? sync_data[11:0] : 12'hFFF;
            end else ADC_data_valid <= 0;
            //just keep counting....should be okay becuase trigger will always reset count before it
overflows
            count <= count + 1;
         end //end trigger else
      end //rst else
   end  //end always_ff

Endmodule





module AM_BP_Filter #(parameter N = 8) //N = number of feedforward coeff, max 8
   (
      input clk_in,
      input rst,

      //coeffiecients multiplied by 2^16
      input signed [17:0] b [(N-1):0], //N b feedforward coeffs [b(N-1)...b0), unpacked array
      input signed [17:0] a [(N-2):0], //N-1 feedback coeffs [a(N-1)...a1], unpacked array
      input sample_ready, //ADC data valid
      input signed [23:0] sample, //IF_in

      output logic signed [33:0] filt_out,
      output logic filt_valid //indicate new filter value available
   );

   logic signed [23:0] x [7:0]; //store past 8 x values, up to and including x[n]
   logic signed [100:0] y [7:0]; //store past 8 y values up to and inlcuding y[n-1]

   //pointer to x[n] and y[n-1] in x and y arrays
   logic [2:0] index;
   //indicate begin calculating filter ouput
   logic sum_values;

   //used in computing x and y terms for filter output
```

```
logic signed [100:0] x_sum;
logic signed [100:0] y_sum;
logic [3:0] ii;

//ensures index overflow does not result in calling x or y at a negative val
logic [2:0] val_index;
assign val_index = index - ii;

always_ff @(posedge clk_in) begin
   if (rst) begin
      //intialize x & y values to all 0
      for (int i=0; i<8; i=i+1) begin
         x[i] <= 0;
         y[i] <= 0;
         //reset x and y sums
         x_sum <= 0;
         y_sum <= 0;
      end
      //reset current value pointer
      index <= 0;
   end else begin
      if (sample_ready) begin
         //update x with most recent sample
         x[index] <= sample[23:0];
         //start computing filter output, reset sums
         sum_values <= 1;
         //reset sum values
         x_sum <= 0;
         y_sum <= 0;
         //increment pointer index
         index <= index+1;
         //start ii at 1, since incrementing index
         //therefore x[n] is at [index-1]
         ii <= 1;
      end else begin //sample ready
      //calculate x and y sums for past N values
         if (sum_values && (ii <= N)) begin
            x_sum <= x_sum + ( (x[val_index]*b[ii-1])>>>16 );
            //only summing N-2 y terms
            y_sum <= (ii <= (N-1)) ? ( y_sum + ( (y[val_index]*a[ii-1])>>>16 ) ) : y_sum;
            ii <= ii + 1;
         end else if (sum_values) begin
            //stop summation
```

```
                sum_values <= 0;
                //compute filt_out
                filt_out <= x_sum - y_sum;
                filt_valid <= 1;
                //store this to be used as previous value of y
                y[index] <= x_sum - y_sum;
             end else begin
                filt_valid <= 0;
             end //sum values
          end //else sample ready

       end //rst else
    end //end always_ff
Endmodule

module AM_audio_condition(
       input clk, //should be 100 MHz for proper sampling
       input rst,
       //signal from peak detect, 10 MSPS
       input [33:0] audio_offset,
       //audio level, set by 3 switches
       input [4:0] audio_level,

       //8 bit audio signal, centered at 0
       output logic signed [7:0] audio_out
    );

    //downsample to 48 khz  -------
    parameter COUNT_48k = 2082;

    logic [11:0] count;
    logic trigger;
    //down-sampled, amplitude adjusted audio (48kHz). Still contains DC offset and is 24 bits
    logic [33:0] ds_audio_offset;

    //trigger once every COUNT_48k cycles
    assign trigger = (count == COUNT_48k);

    //HP filter trigger
    logic sample_ready;

    always_ff @(posedge clk) begin
       if (rst) count <= 0;
    else begin
        //rst count every COUNT_48k periods of 100MHz clock for 48kHz sample rate
        if (trigger) begin
            count <= 0;
            ds_audio_offset <= audio_offset;
            sample_ready <= 1;
        end else begin
            sample_ready <= 0 ;
            count <= count + 1;
        end //else trigger
    end //else rst
end

//Moving average to compute audio offset
//array of past peak detect outputs
logic [33:0] window [31:0];
//current index in window
logic [4:0] index;
logic [5:0] sum_i; //index for computing sum
logic [38:0] sum; //sum of values in window
logic [33:0] avg; //average value of window

always_ff @(posedge clk) begin
    if (rst) begin
        //initialize all values in window to 0
        for (int i=0; i<32; i=i+1) begin
            window[i] <= 0;
        end
        //reset index
        index <= 0;
        sum_i <= 0;
        sum <= 0;
    end else begin //rst else
        if (sample_ready) begin
            //store current down sampled offset audio
            window[index] <= ds_audio_offset;
            index <= index + 1;
            sum <= 0;
            sum_i <= 0;
        end else begin
            //no sample, still computing avg
            if (sum_i < 32) begin
                sum <= sum + window[sum_i];
                sum_i = sum_i + 1;
            end if (sum_i == 32) begin //done computing avg
                sum_i <= 33; //causes system to stay in next else condition until ready to compute
sum again
                avg <= (sum >> 5); //divide sum by 32 (# of window values) to get average
            end else begin
                //Shift level of audio and shrink to 8 bit value
                //uses ds_audio_offset grabbed when sample_ready = 1
                //shifts between 20 and 6 bits (since max audio level = 15)
                audio_out <= ( (ds_audio_offset - avg) >>> ('d26 - audio_level) );
            end //sum_i < 32
        end //sample_ready else
    end //rst
  end //always_ff

Endmodule

module DAC_stuff(input clk_in, input rst_in, input [7:0] level_in, output logic pwm_out);

//PWM generator for audio generation

  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

module FM_demod_stage_1(
    input clk,
    input rst,
    //ouput from RF mixer
    input signed [23:0] IF_in,
    //driven by ADC_data_valid
    input IF_data_valid,
    //if 1, use wideband FM, if 0, use narrowband FM
    input FM_BP_width,
    //derivative of band passed FM signal
    output logic signed [33:0] FM_derivative_out,
    output logic FM_data_valid
  );

  parameter N = 3;
  logic signed [17:0] a1 [(N-2):0]; //N-1 feedback coeffs [a(N-1)...a1], unpacked array
  logic signed [17:0] a2 [(N-2):0]; //N-1 feedback coeffs [a(N-1)...a1], unpacked array
  logic signed [17:0] a3 [(N-2):0]; //N-1 feedback coeffs [a(N-1)...a1], unpacked array
  logic signed [17:0] a4 [(N-2):0]; //N-1 feedback coeffs [a(N-1)...a1], unpacked array

  //select NFM or WFM
  always_comb begin
    if (FM_BP_width) begin
        //FM (Wideband) Bandpass Filter
        a1 [(N-2):0] = '{18'sd61600,-18'sd119464}; //a coeff MATLAB:
        a2 [(N-2):0] = '{18'sd62943,-18'sd125245}; //a coeff MATLAB:
        a3 [(N-2):0] = '{18'sd57240,-18'sd117002}; //a coeff MATLAB:
        a4 [(N-2):0] = '{18'sd58551,-18'sd120066}; //a coeff MATLAB:
    end else begin
        //FM (narrowband) bandpass filter
        a1 [(N-2):0] = '{18'sd64789,-18'sd124533}; //a coeff MATLAB:
        a2 [(N-2):0] = '{18'sd64853,-18'sd125569}; //a coeff MATLAB:
        a3 [(N-2):0] = '{18'sd63792,-18'sd123882}; //a coeff MATLAB:
        a4 [(N-2):0] = '{18'sd63855,-18'sd124342}; //a coeff MATLAB:
    end //if narrow or wide
  end //always_comb

  //section 1 ------------------------------------------
  //initialize coeffs
  logic signed [17:0] b1 [(N-1):0]; //N b feedforward coeffs [b(N-1)...b0), unpacked array

  assign b1 [(N-1):0] = '{-18'sd65536,18'sd0,18'sd65536}; //b coeff MATLAB: [1,0,-1]

  logic signed [33:0] filt_sec_1_out;
  logic sec_1_ready;

  AM_BP_Filter #(.N(N)) FM_BP_sec_1 (.clk_in(clk),.rst(rst),.b(b1),.a(a1),

.sample_ready(IF_data_valid),.sample(IF_in),.filt_out(filt_sec_1_out),.filt_valid(sec_1_ready));

  //section 2 ------------------------------------------
  logic signed [23:0] filt_sec_2_in;
  //divide output from filter section 1 by 2^5 to fit 24 bit input parameter
```

```
    assign filt_sec_2_in = (filt_sec_1_out>>>5);

    //initialize coeffs
    logic signed [17:0] b2 [(N-1):0]; //N b feedforward coeffs [b(N-1)...b0), unpacked array

    assign b2 [(N-1):0] = '{-18'sd65536,18'sd0,18'sd65536}; //b coeff MATLAB: [1,0,-1]

    logic signed [33:0] filt_sec_2_out;
    logic sec_2_ready;

    AM_BP_Filter #(.N(N)) FM_BP_sec_2 (.clk_in(clk),.rst(rst),.b(b2),.a(a2),

.sample_ready(sec_1_ready),.sample(filt_sec_2_in),.filt_out(filt_sec_2_out),.filt_valid(sec_2_rea
dy));

    //section 3 -----------------------------------------
    logic signed [23:0] filt_sec_3_in;
    //divide output from filter section 1 by 2^5 to fit 24 bit input parameter
    assign filt_sec_3_in = (filt_sec_2_out>>>5);

    //initialize coeffs
    logic signed [17:0] b3 [(N-1):0]; //N b feedforward coeffs [b(N-1)...b0), unpacked array

    assign b3 [(N-1):0] = '{-18'sd65536,18'sd0,18'sd65536}; //b coeff MATLAB: [1,0,-1]

    logic signed [33:0] filt_sec_3_out;
    logic sec_3_ready;

    AM_BP_Filter #(.N(N)) FM_BP_sec_3 (.clk_in(clk),.rst(rst),.b(b3),.a(a3),

.sample_ready(sec_2_ready),.sample(filt_sec_3_in),.filt_out(filt_sec_3_out),.filt_valid(sec_3_rea
dy));


     //section 4 -----------------------------------------
    logic signed [23:0] filt_sec_4_in;
    //divide output from filter section 1 by 2^5 to fit 24 bit input parameter
    assign filt_sec_4_in = (filt_sec_3_out>>>5);

    //initialize coeffs
    logic signed [17:0] b4 [(N-1):0]; //N b feedforward coeffs [b(N-1)...b0), unpacked array

    assign b4 [(N-1):0] = '{-18'sd65536,18'sd0,18'sd65536}; //b coeff MATLAB: [1,0,-1]
    logic signed [33:0] filt_sec_4_out;
    logic sec_4_ready;

    //triggers on ADC_sample_valid
    AM_BP_Filter #(.N(N)) FM_BP_sec_4 (.clk_in(clk),.rst(rst),.b(b4),.a(a4),

.sample_ready(sec_3_ready),.sample(filt_sec_4_in),.filt_out(filt_sec_4_out),.filt_valid(sec_4_rea
dy));



    //FM signal derivative with respect to time
    //stores previous value from last section in FM bandpass
    logic signed [33:0] past_filt_4_out;
    logic signed [33:0] past_past_filt_4_out;
    logic signed [33:0] past_FM_derivative;
    logic signed [33:0] FM_derivative;

    //Calculates derivative of band passed FM signal to pass into peak detection for smoothing
    //operates at constant sampling freq therefore d(FM_signal)/d(t) = current_val - past_val / t
    //and dividing by time doesn't matter so can ignore (constant)
    always_ff @(posedge clk) begin
        if (rst) begin
            past_filt_4_out <= 0;
            past_past_filt_4_out <=0;
            past_FM_derivative <= 0;
            FM_derivative <= 0;
            FM_derivative_out <=0;
            FM_data_valid <= 0;
        end else begin
            if (sec_4_ready) begin
                past_past_filt_4_out <= past_filt_4_out;
                past_filt_4_out <= filt_sec_4_out;
                //calc derivative and past derivative
                past_FM_derivative <= past_filt_4_out - past_past_filt_4_out;
                FM_derivative <= filt_sec_4_out - past_filt_4_out;
                //current derivative out averaged over past 2
                FM_derivative_out <= (FM_derivative >>> 2) + (past_FM_derivative >>> 2);
                //trigger data valid. Will occur at 10 MHz
                FM_data_valid <= 1;
            end else begin
                //waiting for data so data_valid = 0
            FM_data_valid <= 0;
          end
        end //rst else
    end //always_ff
Endmodule

module Local_Oscillator(
        input clk_in,
        input rst,
        input [17:0] center_freq_div_20,
        output logic [11:0] LO_out
    );

    //Intermediate freq of radio / 20. typical AM receiver uses 455 kHz
    parameter IF_FREQ_div_20 = 18'd22_750;

    //desired frequency of LO output signal / 20
    logic [17:0] LO_freq_div_20;
    //center_freq should be 18 bit value for approx 20 Hz resolution up to 5 MHz
    assign LO_freq_div_20 = center_freq_div_20 - IF_FREQ_div_20;

    //Parameter to convert desired frequency to phase increments.
    //A = round( (2^32 - 1)/f_clk * 2^8 )
    parameter A = 13744;
    logic [31:0] phase_inc;
    assign phase_inc = (LO_freq_div_20 * A) >> 4;

    logic [31:0] phase;
    //phase[31:24] = 8 bits range for sine wave look up table in memory
    //returns 12 bit value from memory intialized with sine wave [0,4095]
    sine_wave LO_sine_wave (.clka(clk_in), .addra(phase[31:24]), .douta(LO_out));

    //increment phase on rising edge of 100 MHz clk
    always_ff @(posedge clk_in)begin
        if (rst)begin
            phase <= 32'b0;
        end else begin
            phase <= phase + phase_inc;
        end
    end
Endmodule

module Mixer(
        input [11:0] in_a,
        input [11:0] in_b,

        output [23:0] p_out
    );

    assign p_out[23:0] = in_a[11:0] * in_b[11:0];

Endmodule

module Peak_detect_hold(
        input clk,
        input rst,

        input sample_ready,
        input signed [33:0] sample_in,

        output logic [34:0] peak_value,
        output logic sample_ready_out
    );

    logic signed [33:0] past_val [7:0]; //store past 8 values
    logic deriv [7:0]; //store past 8 derivatives
    logic [2:0] index; //store index in past_val

    //calculates sum of derivatives
    logic [3:0] deriv_sum;
    assign deriv_sum = deriv[7]+deriv[6]+deriv[5]+deriv[4]+deriv[3]+deriv[2]+deriv[1]+deriv[0];

    //calculate sum of values to determine current sign
    logic signed [40:0] val_sum;
    assign val_sum = past_val[7]+past_val[6]+past_val[5]+past_val[4]+past_val[3]+
                    past_val[2]+past_val[1]+past_val[0];

    logic signed [33:0] extrema;

    always_ff @(posedge clk) begin
        if (rst) begin
            deriv <= '{0,0,0,0,0,0,0,0};
            past_val <= '{0,0,0,0,0,0,0,0};
            index <= 0;
        end else begin
            if (sample_ready) begin
            //store current sample at index in array
            past_val[index] <= sample_in;
            //deriv = 1 if positive, 0 if negative
            if (index >= 1) begin
                deriv[index] <= ( (sample_in - past_val[index-1]) >= 0 );
            end else begin
                deriv[index] <= ( (sample_in - past_val[7]) >= 0 );
            end
            //increment index. Will overflow once reaches 7
            index <= index + 1;
        end //sample_ready
    end //rst
end //always_ff

logic [3:0] s_index;

always_ff @(posedge clk) begin
    if (rst) begin
        s_index <= 0;
        extrema <= 0;
        peak_value <= 0;
        sample_ready_out <= 0;
    end else begin
       //peak detected if 5 positives & 5 negative derivatives
        //find and output max or min value, stop if searched all index
        if (deriv_sum == 5 && s_index < 8 ) begin
            //if > 0, search for peak (max)
            if (val_sum >= 0) begin
                //if value at index is greater than previous, set extrema to that. otherwise set
                //equal to current extrema value
                extrema <= ( past_val[s_index] > extrema ) ? past_val[s_index] : extrema;
            end else begin  //if < 0, look for trough (min). Also multiply value by -1
                extrema <= ( past_val[s_index] < extrema ) ? past_val[s_index] : extrema;
            end
            //increment search index
            s_index <= s_index + 1;
        end else begin
            //reset search index
            s_index <= 0;
            //reset extrema to 0
            extrema <= 0;
            //reset extrema to past_vals[0] if in positive region, otherwise -past_vals[0]
//              if (val_sum >= 0) extrema <= past_val[s_index - 1];
//              else extrema <= (-'sd1)*past_val[s_index - 1];
            end

            //peak_Value only re-assigned once finishing searching all values, then holds
            //until after next search. Also converts from signed to unsigned variable
            if (s_index == 8) begin
                peak_value <= (extrema >= 0) ? extrema : -'sd1*extrema;
                //assert sample for 1 clk cycle
                sample_ready_out <= 1;
            end else sample_ready_out <= 0;

        end //!rst

    end //always_ff

Endmodule

module trigger_buffer (
  input clock_in,
  input reset_in,
  input [11:0] signal_in,
  input [11:0] trigger_height,
  input [10:0] hcount_in,
  input [9:0]  vcount_in,
  input [11:0] period,
  output logic [11:0] signal_out
  );

  //States for our state machine
  parameter RESET = 5'b00000;
  parameter BRAM1_WAIT_FOR_TRIGGER = 5'b00001;
  parameter BRAM1_VERIFY_TRIGGER = 5'b00011;
  parameter BRAM1_WAIT_FOR_FILL = 5'b00111;
  parameter BRAM1_WAIT_FOR_FRAME = 5'b01111;
  parameter BRAM2_WAIT_FOR_TRIGGER = 5'b11111;
  parameter BRAM2_VERIFY_TRIGGER = 5'b11101;
  parameter BRAM2_WAIT_FOR_FILL = 5'b11001;
  parameter BRAM2_WAIT_FOR_FRAME = 5'b10001;

  //state variable
  logic [4:0] state;

  //Important wires for interfacing with our first bram
```

```
logic [9:0] frame1_addr;
logic [11:0] data_to_frame1;
logic [11:0] data_from_frame1;
logic write_frame1;

//Creating the first bram with the correct wires.
frame_bram frame1(.addra(frame1_addr), .clka(clock_in),
   .dina(data_to_frame1), .douta(data_from_frame1), .ena(1),
   .wea(write_frame1));

//Important wires for interfacing with our second bram
logic [9:0] frame2_addr;
logic [11:0] data_to_frame2;
logic [11:0] data_from_frame2;
logic write_frame2;

//Creating the second bram with the correct wires.
frame_bram frame2(.addra(frame2_addr), .clka(clock_in),
   .dina(data_to_frame2), .douta(data_from_frame2), .ena(1),
   .wea(write_frame2));

//Registers for out past values. Used for triggering.
logic [11:0] past_signal;
logic [11:0] past_signal2;
logic [11:0] past_signal3;
logic [11:0] past_signal4;
logic [11:0] past_signal5;
logic [11:0] past_signal6;
logic [11:0] past_signal7;
logic [11:0] past_signal8;
logic [11:0] past_signal9;
logic [11:0] past_signal10;
logic [11:0] past_signal11;
logic [11:0] past_signal12;
logic [11:0] past_signal13;
logic [11:0] past_signal14;
logic [11:0] past_signal15;
logic [11:0] past_signal16;
logic [11:0] past_signal17;
logic [11:0] past_signal18;
logic [11:0] past_signal19;

//Counter to make sure we don't store too much data.
```

```
logic [11:0] counter;

//Counter to make sure we stay high for a certain
//amount of time.
logic [11:0] trigger_counter;

//Parameter to set how long you have to stay above trigger.
parameter NEEDED_HIGH = 'd3;

always_ff @(posedge clock_in) begin
  case(state)
    RESET: begin
      //Reset all variables
      frame1_addr <= 0;
      frame2_addr <= 0;
      write_frame1 <= 0;
      write_frame2 <= 0;
      data_to_frame1 <= 0;
      data_to_frame2 <= 0;
      counter <= 0;
      signal_out <= 0;
      //Past values reset to trigger so that
      //they don't do a false positive.
      past_signal19 <= trigger_height;
      past_signal18 <= trigger_height;
      past_signal17 <= trigger_height;
      past_signal16 <= trigger_height;
      past_signal15 <= trigger_height;
      past_signal14 <= trigger_height;
      past_signal13 <= trigger_height;
      past_signal12 <= trigger_height;
      past_signal11 <= trigger_height;
      past_signal10 <= trigger_height;
      past_signal9 <= trigger_height;
      past_signal8 <= trigger_height;
      past_signal7 <= trigger_height;
      past_signal6 <= trigger_height;
      past_signal5 <= trigger_height;
      past_signal4 <= trigger_height;
      past_signal3 <= trigger_height;
      past_signal2 <= trigger_height;
      past_signal <= trigger_height;
      if (!reset_in) begin
    state <= BRAM1_WAIT_FOR_TRIGGER;
  end
end
//In this state we wait until the signal
//goes above trigger
BRAM1_WAIT_FOR_TRIGGER: begin
  //We set the signal out based on the output from
  //bram 2. That is because bram 2 current holds the
  //captured signal from the last run and we want to
  //keep displaying that until we get a new sample.
  frame2_addr <= (hcount_in - 'd101);
  signal_out <= data_from_frame2;
  past_signal19 <= past_signal18;
  past_signal18 <= past_signal17;
  past_signal17 <= past_signal16;
  past_signal16 <= past_signal15;
  past_signal15 <= past_signal14;
  past_signal14 <= past_signal13;
  past_signal13 <= past_signal12;
  past_signal12 <= past_signal11;
  past_signal11 <= past_signal10;
  past_signal10 <= past_signal9;
  past_signal9 <= past_signal8;
  past_signal8 <= past_signal7;
  past_signal7 <= past_signal6;
  past_signal6 <= past_signal5;
  past_signal5 <= past_signal4;
  past_signal4 <= past_signal3;
  past_signal3 <= past_signal2;
  past_signal2 <= past_signal;
  past_signal <= signal_in;
  trigger_counter <= 0;
  //All the past signals need to be below
  //trigger while our current value needs
  //to be at or above trigger level. There
  //is an aditional requirement that the jump
  //above the trigger cannot be too high of a gap.
  //That makes it so that noise cannot set off the
  //trigger.
  if((past_signal < trigger_height) &&
    (past_signal2 < trigger_height) &&
    (past_signal3 < trigger_height) &&
    (past_signal4 < trigger_height) &&
      (past_signal5 < trigger_height) &&
      (past_signal6 < trigger_height) &&
      (past_signal7 < trigger_height) &&
      (past_signal8 < trigger_height) &&
      (past_signal9 < trigger_height) &&
      (past_signal10 < trigger_height) &&
      (past_signal11 < trigger_height) &&
      (past_signal12 < trigger_height) &&
      (past_signal13 < trigger_height) &&
      (past_signal14 < trigger_height) &&
      (past_signal15 < trigger_height) &&
      (past_signal16 < trigger_height) &&
      (past_signal17 < trigger_height) &&
      (past_signal18 < trigger_height) &&
      (past_signal19 < trigger_height) &&
      ((signal_in - past_signal) < 'd1000) &&
      (signal_in >= trigger_height)) begin
       //Set the address to 0 for capturing
       frame1_addr <= 0;
       //Set the write high
       write_frame1 <= 1;
       //Set the correct data in.
       data_to_frame1 <= signal_in;
       //Change state
       state <= BRAM1_VERIFY_TRIGGER;
       //Set counters.
       counter <= 0;
       trigger_counter <= 0;
   end
end
//Here we wait to see if the trigger actually
//holds.
BRAM1_VERIFY_TRIGGER: begin
   //If the signal goes below trigger it doesn't hold.
   //All the wires are reset and go back to waiting for
   //the trigger.
   if(signal_in < trigger_height) begin
      state <= BRAM1_WAIT_FOR_TRIGGER;
      counter <= 0;
      trigger_counter <= 0;
      frame2_addr <= (hcount_in - 'd101);
      signal_out <= data_from_frame2;
      frame1_addr <= 0;
```

```
    write_frame1 <= 0;
    write_frame2 <= 0;
    data_to_frame1 <= 0;
    data_to_frame2 <= 0;
    past_signal19 <= trigger_height;
    past_signal18 <= trigger_height;
    past_signal17 <= trigger_height;
    past_signal16 <= trigger_height;
    past_signal15 <= trigger_height;
    past_signal14 <= trigger_height;
    past_signal13 <= trigger_height;
    past_signal12 <= trigger_height;
    past_signal11 <= trigger_height;
    past_signal10 <= trigger_height;
    past_signal9 <= trigger_height;
    past_signal8 <= trigger_height;
    past_signal7 <= trigger_height;
    past_signal6 <= trigger_height;
    past_signal5 <= trigger_height;
    past_signal4 <= trigger_height;
    past_signal3 <= trigger_height;
    past_signal2 <= trigger_height;
    past_signal <= trigger_height;
  end else begin
    //Keep showing the old bram framw while
    //we verify
    frame2_addr <= (hcount_in - 'd101);
    signal_out <= data_from_frame2;
    //Write current signal while we verify
    data_to_frame1 <= signal_in;
    //If the count is above the needed high,
    //we can go to the next state where we wait
    //until the bram fills.
    if(trigger_counter >= NEEDED_HIGH) begin
      state<= BRAM1_WAIT_FOR_FILL;
    end else begin
      //Else we count up.
      trigger_counter <= trigger_counter + 1;
    end
    //Counter in place to make sure that we wait
    //a certain period to put samples into the bram.
    //This lets us see signals of varying frequencies.
    if(counter == period) begin
```

```
      frame1_addr <= frame1_addr + 1;
      counter <= 0;
    end else begin
      counter <= counter + 1;
    end
  end
end
//In this state we wait for the bram to capture all
//of the data it needs.
BRAM1_WAIT_FOR_FILL: begin
  //Keep displaying from second bram while we fill
  frame2_addr <= (hcount_in - 'd101);
  signal_out <= data_from_frame2;
  //Keep writing signal to bram 1
  data_to_frame1 <= signal_in;
  //If the addr becomes big enough we can move to
  //next state.
  if(frame1_addr == 'd821) begin
    state<= BRAM1_WAIT_FOR_FRAME;
    write_frame1 <= 0;
  end else begin
    //Have the counter to make sure we are sampling
    //at the user given period.
    if(counter == period) begin
      frame1_addr <= frame1_addr + 1;
      counter <= 0;
    end else begin
      counter <= counter + 1;
    end
  end
end
//In this state we wait for the fram to end before we switch
//the brams.
BRAM1_WAIT_FOR_FRAME: begin
  frame2_addr <= (hcount_in - 'd101);
  signal_out <= data_from_frame2;
  //If vcount gets to the last line, we switch the state
  if((vcount_in == 'd767)) begin
    state<= BRAM2_WAIT_FOR_TRIGGER;
  end
end
//The state machine from before is the exact same as above,
//it just switches the role of bram1 and bram2. We would be
```

```
//displaying from bram1 while bram2 is being written to.
BRAM2_WAIT_FOR_TRIGGER: begin
   frame1_addr <= (hcount_in - 'd101);
   signal_out <= data_from_frame1;
   past_signal19 <= past_signal18;
   past_signal18 <= past_signal17;
   past_signal17 <= past_signal16;
   past_signal16 <= past_signal15;
   past_signal15 <= past_signal14;
   past_signal14 <= past_signal13;
   past_signal13 <= past_signal12;
   past_signal12 <= past_signal11;
   past_signal11 <= past_signal10;
   past_signal10 <= past_signal9;
   past_signal9 <= past_signal8;
   past_signal8 <= past_signal7;
   past_signal7 <= past_signal6;
   past_signal6 <= past_signal5;
   past_signal5 <= past_signal4;
   past_signal4 <= past_signal3;
   past_signal3 <= past_signal2;
   past_signal2 <= past_signal;
   past_signal <= signal_in;
   if((past_signal < trigger_height) &&
      (past_signal2 < trigger_height) &&
      (past_signal3 < trigger_height) &&
      (past_signal4 < trigger_height) &&
      (past_signal5 < trigger_height) &&
      (past_signal6 < trigger_height) &&
      (past_signal7 < trigger_height) &&
      (past_signal8 < trigger_height) &&
      (past_signal9 < trigger_height) &&
      (past_signal10 < trigger_height) &&
      (past_signal11 < trigger_height) &&
      (past_signal12 < trigger_height) &&
      (past_signal13 < trigger_height) &&
      (past_signal14 < trigger_height) &&
      (past_signal15 < trigger_height) &&
      (past_signal16 < trigger_height) &&
      (past_signal17 < trigger_height) &&
      (past_signal18 < trigger_height) &&
      (past_signal19 < trigger_height) &&
      ((signal_in - past_signal) < 'd1000) &&
      (signal_in >= trigger_height)) begin
       frame2_addr <= 0;
       write_frame2 <= 1;
       data_to_frame2 <= signal_in;
       state <= BRAM2_VERIFY_TRIGGER;
       counter <= 0;
    end
  end
  BRAM2_VERIFY_TRIGGER: begin
    if(signal_in < trigger_height) begin
      state <= BRAM2_WAIT_FOR_TRIGGER;
      counter <= 0;
      trigger_counter <= 0;
      frame1_addr <= (hcount_in - 'd101);
      signal_out <= data_from_frame2;
      frame2_addr <= 0;
      write_frame1 <= 0;
      write_frame2 <= 0;
      data_to_frame1 <= 0;
      data_to_frame2 <= 0;
      past_signal19 <= trigger_height;
      past_signal18 <= trigger_height;
      past_signal17 <= trigger_height;
      past_signal16 <= trigger_height;
      past_signal15 <= trigger_height;
      past_signal14 <= trigger_height;
      past_signal13 <= trigger_height;
      past_signal12 <= trigger_height;
      past_signal11 <= trigger_height;
      past_signal10 <= trigger_height;
      past_signal9 <= trigger_height;
      past_signal8 <= trigger_height;
      past_signal7 <= trigger_height;
      past_signal6 <= trigger_height;
      past_signal5 <= trigger_height;
      past_signal4 <= trigger_height;
      past_signal3 <= trigger_height;
      past_signal2 <= trigger_height;
      past_signal <= trigger_height;
    end else begin
      frame1_addr <= (hcount_in - 'd101);
      signal_out <= data_from_frame1;
      data_to_frame2 <= signal_in;
```

```
          if(trigger_counter >= NEEDED_HIGH) begin
            state<= BRAM2_WAIT_FOR_FILL;
          end else begin
            trigger_counter <= trigger_counter + 1;
          end
          if(counter == period) begin
            frame2_addr <= frame2_addr + 1;
            counter <= 0;
          end else begin
            counter <= counter + 1;
          end
        end
      end
      BRAM2_WAIT_FOR_FILL: begin
        frame1_addr <= (hcount_in - 'd101);
        signal_out <= data_from_frame1;
        data_to_frame2 <= signal_in;
        if(frame2_addr == 'd821) begin
          state<= BRAM2_WAIT_FOR_FRAME;
          write_frame2 <= 0;
        end else begin
          if(counter == period) begin
            frame2_addr <= frame2_addr + 1;
            counter <= 0;
          end else begin
            counter <= counter + 1;
          end
        end
      end
      BRAM2_WAIT_FOR_FRAME: begin
        frame1_addr <= (hcount_in - 'd100);
        signal_out <= data_from_frame1;
        if((vcount_in == 'd767)) begin
          state<= BRAM1_WAIT_FOR_TRIGGER;
        end
      end
      default: state <= RESET;
    endcase
  end
Endmodule

module function_pixel_logic (
  input vclock_in,       // 65MHz clock
input reset_in,         // 1 to initialize module
input [10:0] hcount_in, // horizontal index of current pixel (0..1023)
input [9:0]  vcount_in, // vertical index of current pixel (0..767)
input [11:0] height_adjust,
input [11:0] signal_in,
input [11:0] trigger_height,
input is_audio,
output [11:0] pixel_out  // pong game's pixel  // r=11:8, g=7:4, b=3:0
);

//The pixel values for our function,
//centerline, and triggerline
logic [11:0]pixel_out_centerline;
logic [11:0]pixel_out_triggerline;
logic [11:0]pixel_out_function;

//Raw signe values
logic signed[23:0] raw_signal_height;
logic signed[23:0] raw_trigger_height;
//Scaled down to 12 bits
logic signed[11:0] scaled_signal_height;
logic signed[11:0] scaled_trigger_height;

//Used for the special audio version
logic signed[11:0] signed_signal;
assign signed_signal = signal_in;
logic signed[11:0] signed_height_adjust;
assign signed_height_adjust = height_adjust;

//parameters for possible offsets
parameter OFFSET_25 = 'sd2048;
parameter OFFSET_20 = 'sd1638;
parameter OFFSET_15 = 'sd1228;
parameter OFFSET_AUDIO = 'sd0;

always_comb begin
  //If it is audio we want to use a different offset
  //and different variables.
  if(is_audio) begin
    //multiply to get the signal up.
    raw_signal_height = (signed_signal - OFFSET_AUDIO) * signed_height_adjust;
    //Divide to normalize the signal.
    scaled_signal_height = raw_signal_height >>> 11;
```

```
      //same process
      raw_trigger_height = (trigger_height - OFFSET_AUDIO) * signed_height_adjust;
      scaled_trigger_height = raw_trigger_height >>> 11;
    end else begin
      //same process as above but different offset and using
      //original signal and trigger in.
      raw_signal_height = (signal_in - OFFSET_25) * height_adjust;
      scaled_signal_height = raw_signal_height >>> 11;

      raw_trigger_height = (trigger_height - OFFSET_25) * height_adjust;
      scaled_trigger_height = raw_trigger_height >>> 11;
    end
  end

  always_ff @(posedge vclock_in) begin
    //If the hcount is within a specific range, we might
    //have to display our function
    if((hcount_in>104) & (hcount_in<919)) begin
      //Using three pixel height, if our vcount matches the signal height we make
      //the function pixel white.
      if((vcount_in == (12'sd384 - scaled_signal_height))
        || (vcount_in == (12'sd384 - scaled_signal_height + 12'sd1))
        || (vcount_in == (12'sd384 - scaled_signal_height + 12'sd2))) begin
         pixel_out_function <= 12'hfff;
      end else begin
         pixel_out_function <= 12'h0;
      end
    end else begin
      pixel_out_function <= 12'h0;
    end
  end
  //Centerline is constant
  assign pixel_out_centerline = {12{((vcount_in==384) & (hcount_in>100) & (hcount_in<923))}};
  //Triggerline is like centerline but varies depending on the
  //actual trigger out
  assign pixel_out_triggerline = {4{((vcount_in==(12'sd384 - scaled_trigger_height)) &
(hcount_in>100) & (hcount_in<923))}};

  //The pixel we send out is a combination of all of the pixel values.
  assign pixel_out = pixel_out_centerline + pixel_out_function + pixel_out_triggerline;
Endmodule
```

```
module histogram(
   input logic clk,
   input logic rst,
   input logic [10:0] hcount,
   input logic [9:0] vcount,
   input logic blank,
   input logic [1:0] range_in,
   output logic [9:0] vaddr,
   input logic [15:0] vdata,
   input logic [18:0] freq,
   input logic is_if,
   input [9:0] zoom_offset_in,
   input is_waterfall,
   input vsync_pulse,
   output logic [12:0] pixel
   );

   //Range and zoom offset wires
   logic[1:0] range;
   logic[9:0] zoom_offset;

   // 1 bin per pixel, with the selected range
   assign vaddr = (hcount[9:0] >> range) + zoom_offset;

   logic [9:0] hheight; // Height of histogram bar
   assign hheight = vdata >> 4;
   logic [9:0] vheight; // The height of pixel above bottom of screen
   assign vheight = 10'd500 - vcount;
   logic blank1; // blank pipelined 1

   //Pixel wires for different parts of the FFT display
   logic[12:0] pixel_histogram;
   logic[12:0] pixel_waterfall;
   logic[12:0] pixel_marker;

   //Set registers to default values for current frequency
   //and the current position of the center frequency
   //marker.
   reg [22:0] current_freq = 22'd600_000;
   reg [16:0]  current_position = 10'd246;

   //Wires for logic in the zooming.
   logic[21:0] max_frequency;
```

```
logic[21:0] min_frequency_window;
logic[21:0] max_frequency_window;
logic[16:0] display_position;
logic[11:0] spacer;
logic[16:0] if_position;
always_comb begin
   //The max frequency of a zoom is the original max
   //shifted by the range.
   max_frequency = 'd2500000 >> range;
   //The position of the center frequency marker will be shifted
   //by the zoom offset and than shifted to match the zoom
   display_position = ((current_position - zoom_offset) << range);
   //The spacer is a constant value, it is the frequency difference
   //between pixels at no zoom
   spacer = 'd2500000 >> 10;
   //This is the line for the intermediate frequency. It is constant.
   if_position = ('d187 - zoom_offset) << range;
   //The minimum frequency for our window will by the zoom offset multiplied
   //by the spacer
   min_frequency_window = zoom_offset * spacer;
   //The max frequency of the window will the minimum freqency plus the max
   //frequency of the zoom.
   max_frequency_window = min_frequency_window + max_frequency;

   //If we are trying to display the waterfall we want a certain zoom and
   //position, else we do waht the user has input.
   if(is_waterfall) begin
      range <= 'd11;
      zoom_offset <= current_position - 'd60;
   end else begin
      range <= range_in;
      zoom_offset <= zoom_offset_in;
   end
end

always_comb begin
  //The pixel of the fft is based on the current vcount (vheight) and
  //the height of the hisogram.
  pixel_histogram = (vheight < hheight) ? 12'hfff : 12'b0;

  //We should display the frequency center market if we are around the
  //correct horizontal area corresponding to that frequency. We also have to
  //make sure the freqency is in range, as the display_position can overflow
```

```
      if((hcount == display_position || hcount == (display_position + 1) || hcount ==
(display_position - 1))
          && (current_freq < max_frequency_window) && (current_freq > min_frequency_window))
begin
         //Sets the height of the market.
         if((vcount < 'd520) && (vcount > 'd505)) begin
           pixel_marker = 12'hd22;
         end else begin
           pixel_marker = 0;
         end
      end else begin
         //Same for the centerfrequency marker but for the IF frequency.
         //The color for this marker in mint.
         if(is_if) begin
           if((hcount == if_position || hcount == (if_position + 1) || hcount == (if_position - 1))
             && ('d455_000 < max_frequency_window) && ('d455_000 > min_frequency_window))
begin
             if((vcount < 'd520) && (vcount > 'd505)) begin
               pixel_marker = 12'h0fb;
             end else begin
               pixel_marker = 0;
             end
           end else begin
             pixel_marker = 0;
           end
         end else begin
           pixel_marker = 0;
         end
      end
    end

    //State machine for the waterfall displays.
    //The waterfall display currently does not
    //work as expected.
    parameter RESET = 2'b00;
    parameter ADDING = 2'b01;
    parameter TRANSITION = 2'b11;
    logic [1:0]state;

    //The concept for the waterfall is that it would be 8 sections
    //lengthwise and 7 seconts heightwise. Here we have a value
    //for each of those.
    logic[9:0] bin1_section1;
```

```
logic[9:0] bin2_section1;
logic[9:0] bin3_section1;
logic[9:0] bin4_section1;
logic[9:0] bin5_section1;
logic[9:0] bin6_section1;
logic[9:0] bin7_section1;
logic[9:0] bin8_section1;

logic[9:0] bin1_section2;
logic[9:0] bin2_section2;
logic[9:0] bin3_section2;
logic[9:0] bin4_section2;
logic[9:0] bin5_section2;
logic[9:0] bin6_section2;
logic[9:0] bin7_section2;
logic[9:0] bin8_section2;

logic[9:0] bin1_section3;
logic[9:0] bin2_section3;
logic[9:0] bin3_section3;
logic[9:0] bin4_section3;
logic[9:0] bin5_section3;
logic[9:0] bin6_section3;
logic[9:0] bin7_section3;
logic[9:0] bin8_section3;

logic[9:0] bin1_section4;
logic[9:0] bin2_section4;
logic[9:0] bin3_section4;
logic[9:0] bin4_section4;
logic[9:0] bin5_section4;
logic[9:0] bin6_section4;
logic[9:0] bin7_section4;
logic[9:0] bin8_section4;

logic[9:0] bin1_section5;
logic[9:0] bin2_section5;
logic[9:0] bin3_section5;
logic[9:0] bin4_section5;
logic[9:0] bin5_section5;
logic[9:0] bin6_section5;
logic[9:0] bin7_section5;
logic[9:0] bin8_section5;
```

```
logic[9:0] bin1_section6;
logic[9:0] bin2_section6;
logic[9:0] bin3_section6;
logic[9:0] bin4_section6;
logic[9:0] bin5_section6;
logic[9:0] bin6_section6;
logic[9:0] bin7_section6;
logic[9:0] bin8_section6;

logic[9:0] bin1_section7;
logic[9:0] bin2_section7;
logic[9:0] bin3_section7;
logic[9:0] bin4_section7;
logic[9:0] bin5_section7;
logic[9:0] bin6_section7;
logic[9:0] bin7_section7;
logic[9:0] bin8_section7;

//We had accumulators that take the average of the
//areas, which we will divide to get the average
//value
logic[21:0] bin1_accumulator;
logic[21:0] bin2_accumulator;
logic[21:0] bin3_accumulator;
logic[21:0] bin4_accumulator;
logic[21:0] bin5_accumulator;
logic[21:0] bin6_accumulator;
logic[21:0] bin7_accumulator;
logic[21:0] bin8_accumulator;

logic[5:0] counter;
always @(posedge clk) begin
  //Only do this if waterfall is active
  if(is_waterfall) begin
    case(state)
      RESET: begin
        //Reset everything
        bin1_section1 <= 0;
        bin2_section1 <= 0;
        bin3_section1 <= 0;
        bin4_section1 <= 0;
        bin5_section1 <= 0;
```

```
bin6_section1 <= 0;
bin7_section1 <= 0;
bin8_section1 <= 0;

bin1_section2 <= 0;
bin2_section2 <= 0;
bin3_section2 <= 0;
bin4_section2 <= 0;
bin5_section2 <= 0;
bin6_section2 <= 0;
bin7_section2 <= 0;
bin8_section2 <= 0;

bin1_section3 <= 0;
bin2_section3 <= 0;
bin3_section3 <= 0;
bin4_section3 <= 0;
bin5_section3 <= 0;
bin6_section3 <= 0;
bin7_section3 <= 0;
bin8_section3 <= 0;

bin1_section4 <= 0;
bin2_section4 <= 0;
bin3_section4 <= 0;
bin4_section4 <= 0;
bin5_section4 <= 0;
bin6_section4 <= 0;
bin7_section4 <= 0;
bin8_section4 <= 0;

bin1_section5 <= 0;
bin2_section5 <= 0;
bin3_section5 <= 0;
bin4_section5 <= 0;
bin5_section5 <= 0;
bin6_section5 <= 0;
bin7_section5 <= 0;
bin8_section5 <= 0;

bin1_section6 <= 0;
bin2_section6 <= 0;
bin3_section6 <= 0;
```

```
    bin4_section6 <= 0;
    bin5_section6 <= 0;
    bin6_section6 <= 0;
    bin7_section6 <= 0;
    bin8_section6 <= 0;

    bin1_section7 <= 0;
    bin2_section7 <= 0;
    bin3_section7 <= 0;
    bin4_section7 <= 0;
    bin5_section7 <= 0;
    bin6_section7 <= 0;
    bin7_section7 <= 0;
    bin8_section7 <= 0;

    bin1_accumulator <= 0;
    bin2_accumulator <= 0;
    bin3_accumulator <= 0;
    bin4_accumulator <= 0;
    bin5_accumulator <= 0;
    bin6_accumulator <= 0;
    bin7_accumulator <= 0;
    bin8_accumulator <= 0;

    counter <= 0;

    state <= ADDING;
  end
  ADDING: begin
      //For one line we add up all the hheights of a certain area.
      //This total will be divided to get the average.
      if(vcount == 'd521) begin
        if((hcount >= 0) && (hcount <= 'd127)) begin
          bin1_accumulator <= bin1_accumulator + hheight;
        end else if ((hcount >= 'd128) && (hcount <= 'd255)) begin
          bin2_accumulator <= bin2_accumulator + hheight;
        end else if ((hcount >= 'd256) && (hcount <= 'd383)) begin
          bin3_accumulator <= bin3_accumulator + hheight;
        end else if ((hcount >= 'd384) && (hcount <= 'd511)) begin
          bin4_accumulator <= bin4_accumulator + hheight;
        end else if ((hcount >= 'd512) && (hcount <= 'd639)) begin
          bin5_accumulator <= bin5_accumulator + hheight;
        end else if ((hcount >= 'd640) && (hcount <= 'd767)) begin
```

```
            bin6_accumulator <= bin6_accumulator + hheight;
          end else if ((hcount >= 'd768) && (hcount <= 'd895)) begin
            bin7_accumulator <= bin7_accumulator + hheight;
          end else if ((hcount >= 'd896) && (hcount <= 'd1023)) begin
            bin8_accumulator <= bin8_accumulator + hheight;
          end
      end
      //On next like we transistion
      if(vcount == 'd522) begin
        state <= TRANSITION;
      end
    end
    TRANSITION: begin
      //For transition the most recent section is the division
      //of the accumulator and the rest are cascaded down.
      bin1_section1 <= bin1_accumulator >> 7;
      bin2_section1 <= bin2_accumulator >> 7;
      bin3_section1 <= bin3_accumulator >> 7;
      bin4_section1 <= bin4_accumulator >> 7;
      bin5_section1 <= bin5_accumulator >> 7;
      bin6_section1 <= bin6_accumulator >> 7;
      bin7_section1 <= bin7_accumulator >> 7;
      bin8_section1 <= bin8_accumulator >> 7;

      bin1_section2 <= bin1_section1;
      bin2_section2 <= bin2_section1;
      bin3_section2 <= bin3_section1;
      bin4_section2 <= bin4_section1;
      bin5_section2 <= bin5_section1;
      bin6_section2 <= bin6_section1;
      bin7_section2 <= bin7_section1;
      bin8_section2 <= bin8_section1;

      bin1_section3 <= bin1_section2;
      bin2_section3 <= bin2_section2;
      bin3_section3 <= bin3_section2;
      bin4_section3 <= bin4_section2;
      bin5_section3 <= bin5_section2;
      bin6_section3 <= bin6_section2;
      bin7_section3 <= bin7_section2;
      bin8_section3 <= bin8_section2;

      bin1_section4 <= bin1_section3;
```

```
bin2_section4 <= bin2_section3;
bin3_section4 <= bin3_section3;
bin4_section4 <= bin4_section3;
bin5_section4 <= bin5_section3;
bin6_section4 <= bin6_section3;
bin7_section4 <= bin7_section3;
bin8_section4 <= bin8_section3;

bin1_section5 <= bin1_section4;
bin2_section5 <= bin2_section4;
bin3_section5 <= bin3_section4;
bin4_section5 <= bin4_section4;
bin5_section5 <= bin5_section4;
bin6_section5 <= bin6_section4;
bin7_section5 <= bin7_section4;
bin8_section5 <= bin8_section4;

bin1_section6 <= bin1_section5;
bin2_section6 <= bin2_section5;
bin3_section6 <= bin3_section5;
bin4_section6 <= bin4_section5;
bin5_section6 <= bin5_section5;
bin6_section6 <= bin6_section5;
bin7_section6 <= bin7_section5;
bin8_section6 <= bin8_section5;

bin1_section7 <= bin1_section6;
bin2_section7 <= bin2_section6;
bin3_section7 <= bin3_section6;
bin4_section7 <= bin4_section6;
bin5_section7 <= bin5_section6;
bin6_section7 <= bin6_section6;
bin7_section7 <= bin7_section6;
bin8_section7 <= bin8_section6;

bin1_accumulator <= 0;
bin2_accumulator <= 0;
bin3_accumulator <= 0;
bin4_accumulator <= 0;
bin5_accumulator <= 0;
bin6_accumulator <= 0;
bin7_accumulator <= 0;
bin8_accumulator <= 0;
```

```
    counter <= 0;
    state <= ADDING;
  end
  default: state <= RESET;
endcase
//For display, we have to determine what area the waterfall is in and
//have the correct bin and section determine the pixel color. In this
//scheme red colors are supposed to be high magnitude while blue colors
//are low magnitude.
if((vcount >= 'd530) && (vcount <= 'd549)) begin
  if((hcount >= 0) && (hcount <= 'd127)) begin
    pixel_waterfall <= {{hheight[6:2]}, {4'd0}, {4'hF - hheight[6:2]}};
  end else if ((hcount >= 'd128) && (hcount <= 'd255)) begin
    pixel_waterfall <= {{hheight[6:2]}, {4'd0}, {4'hF - hheight[6:2]}};
  end else if ((hcount >= 'd256) && (hcount <= 'd383)) begin
    pixel_waterfall <= {{hheight[6:2]}, {4'd0}, {4'hF - hheight[6:2]}};
  end else if ((hcount >= 'd384) && (hcount <= 'd511)) begin
    pixel_waterfall <= {{hheight[6:2]}, {4'd0}, {4'hF - hheight[5:1]}};
  end else if ((hcount >= 'd512) && (hcount <= 'd639)) begin
    pixel_waterfall <= {{hheight[6:2]}, {4'd0}, {4'hF - hheight[6:2]}};
  end else if ((hcount >= 'd640) && (hcount <= 'd767)) begin
    pixel_waterfall <= {{hheight[6:2]}, {4'd0}, {4'hF - hheight[6:2]}};
  end else if ((hcount >= 'd768) && (hcount <= 'd895)) begin
    pixel_waterfall <= {{hheight[6:2]}, {4'd0}, {4'hF - hheight[6:2]}};
  end else if ((hcount >= 'd896) && (hcount <= 'd1023)) begin
    pixel_waterfall <= {{hheight[6:2]}, {4'd0}, {4'hF - hheight[6:2]}};
  end
end else if((vcount >= 'd550) && (vcount <= 'd569)) begin
  if((hcount >= 0) && (hcount <= 'd127)) begin
    pixel_waterfall <= {{bin1_section2[6:2]}, {4'd0}, {4'hF - bin1_section2[6:2]}};
  end else if ((hcount >= 'd128) && (hcount <= 'd255)) begin
    pixel_waterfall <= {{bin2_section2[6:2]}, {4'd0}, {4'hF - bin2_section2[6:2]}};
  end else if ((hcount >= 'd256) && (hcount <= 'd383)) begin
    pixel_waterfall <= {{bin3_section2[6:2]}, {4'd0}, {4'hF - bin3_section2[6:2]}};
  end else if ((hcount >= 'd384) && (hcount <= 'd511)) begin
    pixel_waterfall <= {{bin4_section2[6:2]}, {4'd0}, {4'hF - bin4_section2[6:2]}};
  end else if ((hcount >= 'd512) && (hcount <= 'd639)) begin
    pixel_waterfall <= {{bin5_section2[6:2]}, {4'd0}, {4'hF - bin5_section2[6:2]}};
  end else if ((hcount >= 'd640) && (hcount <= 'd767)) begin
    pixel_waterfall <= {{bin6_section2[6:2]}, {4'd0}, {4'hF - bin6_section2[6:2]}};
  end else if ((hcount >= 'd768) && (hcount <= 'd895)) begin
    pixel_waterfall <= {{bin7_section2[6:2]}, {4'd0}, {4'hF - bin7_section2[6:2]}};
    end else if ((hcount >= 'd896) && (hcount <= 'd1023)) begin
      pixel_waterfall <= {{bin8_section2[6:2]}, {4'd0}, {4'hF - bin8_section2[6:2]}};
    end
  end else if((vcount >= 'd570) && (vcount <= 'd589)) begin
    if((hcount >= 0) && (hcount <= 'd127)) begin
      pixel_waterfall <= {{bin1_section3[6:2]}, {4'd0}, {4'hF - bin1_section3[6:2]}};
    end else if ((hcount >= 'd128) && (hcount <= 'd255)) begin
      pixel_waterfall <= {{bin2_section3[6:2]}, {4'd0}, {4'hF - bin2_section3[6:2]}};
    end else if ((hcount >= 'd256) && (hcount <= 'd383)) begin
      pixel_waterfall <= {{bin3_section3[6:2]}, {4'd0}, {4'hF - bin3_section3[6:2]}};
    end else if ((hcount >= 'd384) && (hcount <= 'd511)) begin
      pixel_waterfall <= {{bin4_section3[6:2]}, {4'd0}, {4'hF - bin4_section3[6:2]}};
    end else if ((hcount >= 'd512) && (hcount <= 'd639)) begin
      pixel_waterfall <= {{bin5_section3[6:2]}, {4'd0}, {4'hF - bin5_section3[6:2]}};
    end else if ((hcount >= 'd640) && (hcount <= 'd767)) begin
      pixel_waterfall <= {{bin6_section3[6:2]}, {4'd0}, {4'hF - bin6_section3[6:2]}};
    end else if ((hcount >= 'd768) && (hcount <= 'd895)) begin
      pixel_waterfall <= {{bin7_section3[6:2]}, {4'd0}, {4'hF - bin7_section3[6:2]}};
    end else if ((hcount >= 'd896) && (hcount <= 'd1023)) begin
      pixel_waterfall <= {{bin8_section3[6:2]}, {4'd0}, {4'hF - bin8_section3[6:2]}};
    end
  end else if((vcount >= 'd590) && (vcount <= 'd609)) begin
    if((hcount >= 0) && (hcount <= 'd127)) begin
      pixel_waterfall <= {{bin1_section4[6:2]}, {4'd0}, {4'hF - bin1_section4[6:2]}};
    end else if ((hcount >= 'd128) && (hcount <= 'd255)) begin
      pixel_waterfall <= {{bin2_section4[6:2]}, {4'd0}, {4'hF - bin2_section4[6:2]}};
    end else if ((hcount >= 'd256) && (hcount <= 'd383)) begin
      pixel_waterfall <= {{bin3_section4[6:2]}, {4'd0}, {4'hF - bin3_section4[6:2]}};
    end else if ((hcount >= 'd384) && (hcount <= 'd511)) begin
      pixel_waterfall <= {{bin4_section4[6:2]}, {4'd0}, {4'hF - bin4_section4[6:2]}};
    end else if ((hcount >= 'd512) && (hcount <= 'd639)) begin
      pixel_waterfall <= {{bin5_section4[6:2]}, {4'd0}, {4'hF - bin5_section4[6:2]}};
    end else if ((hcount >= 'd640) && (hcount <= 'd767)) begin
      pixel_waterfall <= {{bin6_section4[6:2]}, {4'd0}, {4'hF - bin6_section4[6:2]}};
    end else if ((hcount >= 'd768) && (hcount <= 'd895)) begin
      pixel_waterfall <= {{bin7_section4[6:2]}, {4'd0}, {4'hF - bin7_section4[6:2]}};
    end else if ((hcount >= 'd896) && (hcount <= 'd1023)) begin
      pixel_waterfall <= {{bin8_section4[6:2]}, {4'd0}, {4'hF - bin8_section4[6:2]}};
    end
  end else if((vcount >= 'd610) && (vcount <= 'd629)) begin
    if((hcount >= 0) && (hcount <= 'd127)) begin
      pixel_waterfall <= {{bin1_section5[6:2]}, {4'd0}, {4'hF - bin1_section5[6:2]}};
    end else if ((hcount >= 'd128) && (hcount <= 'd255)) begin
```

```
            pixel_waterfall <= {{bin2_section5[6:2]}, {4'd0}, {4'hF - bin2_section5[6:2]}};
          end else if ((hcount >= 'd256) && (hcount <= 'd383)) begin
            pixel_waterfall <= {{bin3_section5[6:2]}, {4'd0}, {4'hF - bin3_section5[6:2]}};
          end else if ((hcount >= 'd384) && (hcount <= 'd511)) begin
            pixel_waterfall <= {{bin4_section5[6:2]}, {4'd0}, {4'hF - bin4_section5[6:2]}};
          end else if ((hcount >= 'd512) && (hcount <= 'd639)) begin
            pixel_waterfall <= {{bin5_section5[6:2]}, {4'd0}, {4'hF - bin5_section5[6:2]}};
          end else if ((hcount >= 'd640) && (hcount <= 'd767)) begin
            pixel_waterfall <= {{bin6_section5[6:2]}, {4'd0}, {4'hF - bin6_section5[6:2]}};
          end else if ((hcount >= 'd768) && (hcount <= 'd895)) begin
            pixel_waterfall <= {{bin7_section5[6:2]}, {4'd0}, {4'hF - bin7_section5[6:2]}};
          end else if ((hcount >= 'd896) && (hcount <= 'd1023)) begin
            pixel_waterfall <= {{bin8_section5[6:2]}, {4'd0}, {4'hF - bin8_section5[6:2]}};
          end
        end else begin
          pixel_waterfall <= 0;
        end
      end else begin
        pixel_waterfall <= 0;
      end
    end

    always @(posedge clk) begin
       //The pixel we send out is the combination of all the pixels
       //we are written so far.
       pixel <= pixel_histogram + pixel_marker + pixel_waterfall;
       //If we are reset we set the frequency back to its original
       //spot
       if(rst) begin
         current_freq <= 22'd600_000;
         current_position <= 10'd246;
       //Else we check if the frequency is out of range. We make sure it isn't too far
       //off, as if you subtract this way a shift in the opposite direction would overflow.
       end else if(((20*freq - current_freq) >= spacer) && ((20*freq - current_freq) < 'd100_000))
  begin
         //Update the position and the current frequency for a signle position. If
         //multiple steps are needed it will take multiple clock cycles.
         current_position <= current_position + 1;
         current_freq = current_freq + spacer;
       end else begin
         //Same as above for the opposite direction.
         if(((current_freq - 20*freq) >= spacer) && ((current_freq - 20*freq) < 'd100_000)) begin
           current_position <= current_position - 1;
```

```
            current_freq = current_freq - spacer;
          end
        end
    end
Endmodule

module control_height(
    input clk,
    input up,
    input down,
    input reset,
    input sw,
    input is_audio,
    input is_fft,
    output logic [11:0] height_out
    );

     //States for the state machines
    parameter RESET = 3'b000;
    parameter IDLE = 3'b001;
    parameter UP_INCREMENT = 3'b011;
    parameter DOWN_INCREMENT = 3'b111;
    parameter WAIT_NORMAL = 3'b101;

    //Two seperate state machines with two different
    //outputs. We do this so that the values don't
    //interfere with each other.
    logic [2:0] state_signal;
    logic [2:0] state_audio;

    logic [11:0] height_out_signal;
    logic [11:0] height_out_audio;


    always_ff @(posedge clk) begin
      //If we aren't displaying audio, we are
      //only adjusting the height of the raw
      //ADC signal.
      if(!is_audio) begin
          case(state_signal)
              RESET: begin
                //Set important variables to 0
                height_out_signal <= 'd100;
```

```
  state_signal <= IDLE;
end
IDLE: begin
  //If reset, then go back to reset state
  if(reset) begin
     state_signal <= RESET;
  end else begin
    //If up goes low we are turning
    //clockwise and need to increment up.
    //Check to make sure fft mode isn't
    //on.
    if(!up && sw && !is_fft) begin
      state_signal <= UP_INCREMENT;
    end else begin
      //If down goes low we are turning
      //counter-clockwise and need to
      //increment down. Check to make
      //sure fft mode isn't on.
      if(!down && sw && !is_fft) begin
        state_signal <= DOWN_INCREMENT;
      end
    end
  end
end
UP_INCREMENT: begin
  if(reset) begin
    state_signal <= RESET;
  end else begin
    //Make sure we aren't above our highest possible height.
    //raw ADC signal has less height since the signal is stronger.
    if(height_out_signal <= 'd380) begin
      height_out_signal <= height_out + 'd10;
    end
    state_signal <= WAIT_NORMAL;
  end
end
DOWN_INCREMENT: begin
  if(reset) begin
    state_signal <= RESET;
  end else begin
    //Make sure we dont have a 0 height
    if(height_out_signal >= 'd15) begin
      height_out_signal <= height_out - 'd10;
```

```
        end
        state_signal <= WAIT_NORMAL;
      end
    end
    WAIT_NORMAL: begin
      if(reset) begin
        state_signal <= RESET;
      end else begin
        //Wait until our signals both go high
        //signifying we are idle again.
        if(up & down & sw && !is_fft) begin
          state_signal <= IDLE;
        end
      end
    end
    default: state_signal <= RESET;
  endcase
//State machine for displaying audio. Same as above.
end else begin
  case(state_audio)
    RESET: begin
      height_out_audio <= 'd300;
      state_audio <= IDLE;
    end
    IDLE: begin
      if(reset) begin
         state_audio <= RESET;
      end else begin
        if(!up && sw && !is_fft) begin
          state_audio <= UP_INCREMENT;
        end else begin
          if(!down && sw && !is_fft) begin
            state_audio <= DOWN_INCREMENT;
         end
       end
      end
    end
    UP_INCREMENT: begin
      if(reset) begin
        state_audio <= RESET;
      end else begin
        //Has a higher cap and more movement per turn.
        if(height_out_audio <= 'd500) begin
```

```
                height_out_audio <= height_out + 'd20;
              end
              state_audio <= WAIT_NORMAL;
            end
          end
          DOWN_INCREMENT: begin
            if(reset) begin
              state_audio <= RESET;
            end else begin
              if(height_out_audio >= 3) begin
                height_out_audio <= height_out - 'd20;
              end
              state_audio <= WAIT_NORMAL;
            end
          end
          WAIT_NORMAL: begin
            if(reset) begin
              state_audio <= RESET;
            end else begin
              if(up & down & sw && !is_fft) begin
                state_audio <= IDLE;
              end
            end
          end
          default: state_audio <= RESET;
        endcase
      end
    end

  //Assign the actual height out based upon which state machine is being used.
  assign height_out = is_audio ? height_out_audio : height_out_signal;
Endmodule

module control_trigger_height(
  input clk,
  input up,
  input down,
  input reset,
  input sw,
  input is_audio,
  input is_fft,
  output logic [11:0] height_out
  );
```

```
//states for our state machine
parameter RESET = 3'b000;
parameter IDLE = 3'b001;
parameter UP_INCREMENT = 3'b011;
parameter DOWN_INCREMENT = 3'b111;
parameter WAIT_NORMAL = 3'b101;

//Two seperate state machines with two different
//outputs. We do this so that the values don't
//interfere with each other.
logic [2:0] state_signal;
logic [2:0] state_audio;

logic [11:0] height_out_signal;
logic [11:0] height_out_audio;


always_ff @(posedge clk) begin
  //If we aren't displaying audio, we are
  //only adjusting the trigger of the raw
  //ADC signal.
  if(!is_audio) begin
     case(state_signal)
        RESET: begin
          //Set important variables to 0
          height_out_signal <= 'd1500;
          state_signal <= IDLE;
        end
        IDLE: begin
          if(reset) begin
             state_signal <= RESET;
          end else begin
            //If up goes low we are turning
            //clockwise and need to increment up.
            //Check to make sure fft mode isn't
            //on.
            if(!up && !sw && !is_fft) begin
              state_signal <= UP_INCREMENT;
            end else begin
              //If down goes low we are turning
              //counter-clockwise and need to
              //increment down. Check to make
          //sure fft mode isn't on.
          if(!down && !sw && !is_fft) begin
             state_signal <= DOWN_INCREMENT;
         end
      end
    end
  end
  UP_INCREMENT: begin
    if(reset) begin
      state_signal <= RESET;
    end else begin
      //Make sure we aren't above our highest possible height.
      //raw ADC signal has less height since the signal is stronger.
      if(height_out_signal <= 'd3950) begin
        height_out_signal <= height_out + 'd50;
      end
      state_signal <= WAIT_NORMAL;
    end
  end
  DOWN_INCREMENT: begin
    if(reset) begin
      state_signal <= RESET;
    end else begin
       //Make sure we dont have a 0 height
      if(height_out_signal >= 55) begin
        height_out_signal <= height_out - 'd50;
      end
      state_signal <= WAIT_NORMAL;
    end
  end
  WAIT_NORMAL: begin
    if(reset) begin
      state_signal <= RESET;
    end else begin
      //Wait until our signals both go high
      //signifying we are idle again.
      if(up & down & !sw && !is_fft) begin
        state_signal <= IDLE;
      end
    end
  end
  default: state_signal <= RESET;
endcase
```

```
//State machine for displaying audio. Same as above.
end else begin
   case(state_audio)
      RESET: begin
        height_out_audio <= 'd53;
        state_audio <= IDLE;
      end
      IDLE: begin
        if(reset) begin
            state_audio <= RESET;
        end else begin
          if(!up && !sw && !is_fft) begin
            state_audio <= UP_INCREMENT;
          end else begin
            if(!down && !sw && !is_fft) begin
              state_audio <= DOWN_INCREMENT;
             end
           end
        end
      end
      UP_INCREMENT: begin
        if(reset) begin
          state_audio <= RESET;
        end else begin
          if(height_out_audio <= 'd3950) begin
            height_out_audio <= height_out + 'd20;
          end
          state_audio <= WAIT_NORMAL;
        end
      end
      DOWN_INCREMENT: begin
        if(reset) begin
          state_audio <= RESET;
        end else begin
          if(height_out_audio >= 53) begin
            height_out_audio <= height_out - 'd20;
          end
          state_audio <= WAIT_NORMAL;
        end
      end
      WAIT_NORMAL: begin
        if(reset) begin
          state_audio <= RESET;
```

```
                end else begin
                    if(up & down & !sw && !is_fft) begin
                        state_audio <= IDLE;
                    end
                end
            end
            default: state_audio <= RESET;
        endcase
    end
end

//Assign the actual height out based upon which state machine is being used.
assign height_out = is_audio ? height_out_audio : height_out_signal;
endmodule

module control_period(
    input clk,
    input right,
    input left,
    input reset,
    input is_fast,
    input is_fft,
    output logic [11:0] period_out
    );

    //States for the state machine
    parameter RESET = 3'b000;
    parameter IDLE = 3'b001;
    parameter RIGHT_INCREMENT = 3'b011;
    parameter LEFT_INCREMENT = 3'b111;
    parameter WAIT_NORMAL = 3'b101;

    //Registers to hold the state
    logic [2:0] state;

    always_ff @(posedge clk) begin
        case(state)
            //Reset the correct variables
            RESET: begin
                period_out <= 'd0;
                state <= IDLE;
            end
            IDLE: begin
```

```
   //If reset, then go back to reset state
   if(reset) begin
       state <= RESET;
   end else begin
      //If right goes low we are turning
      //clockwise and need to increment up.
      //Check to make sure that fft mode isn't
      //enabled.
      if(!right && !is_fft) begin
         state <= RIGHT_INCREMENT;
      end else begin
         //If left goes low we are turning
         //counter-clockwise and need to
         //increment down. Check to make sure
         //that fft mode isn't enabled
         if(!left && !is_fft) begin
            state <= LEFT_INCREMENT;
        end
     end
   end
end
RIGHT_INCREMENT: begin
   if(reset) begin
      state <= RESET;
   end else begin
      //If in fast mode, we will change our center frequency faster
      //per increment.
      if(!is_fast) begin
         //Make sure our period isn't too high
         if(period_out <= 'd3950) begin
            period_out <= period_out + 1;
         end
      end else begin
         //Make sure our period isn't too high
         if(period_out <= 'd3930) begin
            period_out <= period_out + 'd20;
         end
      end
      state <= WAIT_NORMAL;
   end
end
LEFT_INCREMENT: begin
   if(reset) begin
```

```
            state <= RESET;
          end else begin
            //If in fast mode, we will change our center frequency faster
            //per increment.
            if(!is_fast) begin
              //Make sure we don't get a negative period.
              if(period_out >= 1) begin
                period_out <= period_out - 1;
              end
            end else begin
              //Make sure we don't get a negative period.
              if(period_out >= 20) begin
                period_out <= period_out - 'd20;
              end
            end
            state <= WAIT_NORMAL;
          end
        end
        WAIT_NORMAL: begin
          if(reset) begin
            state <= RESET;
          end else begin
            //Wait until our signals both go high
            //signifying we are idle again.
            if(left & right && !is_fft) begin
              state <= IDLE;
            end
          end
        end
        default: state <= RESET;
      endcase
    end
Endmodule

module control_center_frequency(
  input clk,
  input right,
  input left,
  input reset,
  input is_fast,
  output logic [17:0] center_frequency_out
  );
```

```
//States for the state machine
parameter RESET = 3'b000;
parameter IDLE = 3'b001;
parameter RIGHT_INCREMENT = 3'b011;
parameter LEFT_INCREMENT = 3'b111;
parameter WAIT_NORMAL = 3'b101;

//Registers to hold the state
logic [2:0] state;

always_ff @(posedge clk) begin
  case(state)
     //Reset the center frequency on reset.
     RESET: begin
       center_frequency_out <= 18'd30_000;
       state <= IDLE;
     end
     IDLE: begin
       //If reset, then go back to reset state
       if(reset) begin
          state <= RESET;
       end else begin
         //If right goes low we are turning
         //clockwise and need to increment up,
         if(!right) begin
           state <= RIGHT_INCREMENT;
         end else begin
           //If left goes low we are turning
           //counter-clockwise and need to
           //increment down.
           if(!left) begin
             state <= LEFT_INCREMENT;
           end
         end
       end
     end
     RIGHT_INCREMENT: begin
       if(reset) begin
         state <= RESET;
       end else begin
         //If in fast mode, we will change our center frequency faster
         //per increment.
         if(is_fast) begin
```

```
        //Make sure we aren't above our highest possible frequency
        if(center_frequency_out <= 18'd249_699) begin
          center_frequency_out <= center_frequency_out + 'd300;
        end
      end else begin
        //Make sure we aren't above our highest possible frequency
        if(center_frequency_out <= 18'd249_999) begin
          center_frequency_out <= center_frequency_out + 'd25;
        end
      end
      state <= WAIT_NORMAL;
    end
  end
  LEFT_INCREMENT: begin
    if(reset) begin
      state <= RESET;
    end else begin
      //If in fast mode, we will change our center frequency faster
      //per increment.
      if(is_fast) begin
        //Make sure we aren't below our lowest possible frequency
        if(center_frequency_out >= 'd25_300) begin
          center_frequency_out <= center_frequency_out - 'd300;
        end
      end else begin
        //Make sure we aren't below our lowest possible frequency
        if(center_frequency_out >= 'd25_001) begin
          center_frequency_out <= center_frequency_out - 'd25;
        end
      end
      state <= WAIT_NORMAL;
    end
  end
  WAIT_NORMAL: begin
    if(reset) begin
      state <= RESET;
    end else begin
      //Wait until our signals both go high
      //signifying we are idle again.
      if(left & right) begin
        state <= IDLE;
      end
    end
```

```
        end
        default: state <= RESET;
     endcase
  end
Endmodule

module control_volume(
  input clk,
  input up,
  input down,
  input reset,
  input signed [7:0] DAC_in,
  output logic [4:0] volume_out
  );

  //States for the state machine
  parameter RESET = 3'b000;
  parameter IDLE = 3'b001;
  parameter UP_INCREMENT = 3'b011;
  parameter DOWN_INCREMENT = 3'b111;
  parameter WAIT_NORMAL = 3'b101;

  //Registers to hold the state
  logic [2:0] state;

  //Variable to tell if we have reached our maximum volume.
  logic is_max;

  always_ff @(posedge clk) begin
     case(state)
        RESET: begin
           //Set important variables to 0
           is_max <= 0;
           volume_out <= 'd0;
           state <= IDLE;
        end
        IDLE: begin
           //If our DAC value is really high, we
           //have gone above our maximum value.
           if(DAC_in > 'sd64) begin
              is_max <= 1;
           end
           //If reset, then go back to reset state
  if(reset) begin
     state <= RESET;
  end else begin
    //If up goes low we are turning
    //clockwise and need to increment up,
    if(!up) begin
      state <= UP_INCREMENT;
    end else begin
      //If down goes low we are turning
      //counter-clockwise and need to
      //increment down.
      if(!down) begin
        state <= DOWN_INCREMENT;
      end
    end
  end
end
UP_INCREMENT: begin
  if(reset) begin
    state <= RESET;
  end else begin
    //Make sure we aren't above our highest possible bit shift
    if(volume_out < 'd26 && !is_max) begin
      volume_out <= volume_out + 1;
    end
    state <= WAIT_NORMAL;
  end
end
DOWN_INCREMENT: begin
  if(reset) begin
    state <= RESET;
  end else begin
    //Make sure we don't do a negative bit shift
    if(volume_out > 'd0) begin
      volume_out <= volume_out - 1;
    end
    is_max <= 0;
    state <= WAIT_NORMAL;
  end
end
WAIT_NORMAL: begin
  if(reset) begin
    state <= RESET;
```

```
            end else begin
              //Wait until our signals both go high
              //signifying we are idle again.
              if(up & down) begin
                state <= IDLE;
              end
            end
          end
          default: state <= RESET;
      endcase
    end
Endmodule

module control_zoom_magnitude(
    input clk,
    input up,
    input down,
    input reset,
    input is_fft,
    input is_waterfall,
    output logic [1:0] zoom_out
    );

    //States for the state machine
    parameter RESET = 3'b000;
    parameter IDLE = 3'b001;
    parameter UP_INCREMENT = 3'b011;
    parameter DOWN_INCREMENT = 3'b111;
    parameter WAIT_NORMAL = 3'b101;

    //Registers to hold the state
    logic [2:0] state;

    always_ff @(posedge clk) begin
      case(state)
          RESET: begin
            //Reset the correct variables
            zoom_out <= 'd0;
            state <= IDLE;
          end
          IDLE: begin
            //If reset, then go back to reset state
            if(reset) begin
```

```
     state <= RESET;
  end else begin
    //If up goes low we are turning
    //clockwise and need to increment up.
    //Check to make sure that fft mode is
    //enabled and waterfall isn't.
    if(!up & is_fft & !is_waterfall) begin
      state <= UP_INCREMENT;
    end else begin
      //If down goes low we are turning
      //counter-clockwise and need to
      //increment down. Check to make sure
      //that fft mode is enabled and
      //waterfall isn't
      if(!down & is_fft & !is_waterfall) begin
        state <= DOWN_INCREMENT;
     end
   end
  end
end
UP_INCREMENT: begin
  if(reset) begin
    state <= RESET;
  end else begin
    //Make sure our zoom isn't at it's max
    if(zoom_out < 'd3) begin
      zoom_out <= zoom_out + 1;
    end
    state <= WAIT_NORMAL;
  end
end
DOWN_INCREMENT: begin
  if(reset) begin
    state <= RESET;
  end else begin
    //Make sure zoom doen'st overflow
    if(zoom_out > 'd0) begin
      zoom_out <= zoom_out - 1;
    end
    state <= WAIT_NORMAL;
  end
end
WAIT_NORMAL: begin
```

```
            if(reset) begin
              state <= RESET;
            end else begin
              //Wait until our signals both go high
              //signifying we are idle again.
              if(up & down & is_fft & !is_waterfall) begin
                state <= IDLE;
              end
            end
          end
          default: state <= RESET;
      endcase
    end
Endmodule

module control_zoom_window(
    input clk,
    input up,
    input down,
    input reset,
    input is_fft,
    input is_waterfall,
    input [1:0] zoom_magnitude,
    output logic [9:0] zoom_pos_out
    );

    //States for the state machine
    parameter RESET = 3'b000;
    parameter IDLE = 3'b001;
    parameter UP_INCREMENT = 3'b011;
    parameter DOWN_INCREMENT = 3'b111;
    parameter WAIT_NORMAL = 3'b101;

    //Three seperate state machines all for the
    //different zoom levels.
    logic [2:0] state_01;
    logic [9:0] zoom_out_01;

    logic [2:0] state_10;
    logic [9:0] zoom_out_10;

    logic [2:0] state_11;
    logic [9:0] zoom_out_11;
```

```
always_ff @(posedge clk) begin
  case(zoom_magnitude)
    //No zoom means we can't alter the offset
    2'b00: zoom_pos_out <= 0;
    2'b01: begin
      case(state_01)
       //Reset Variables
       RESET: begin
         zoom_out_01 <= 'd0  ;
         state_01 <= IDLE;
       end
       IDLE: begin
         //If reset, then go back to reset state
         if(reset) begin
            state_01 <= RESET;
         end else begin
           //If up goes low we are turning
           //clockwise and need to increment up.
           //Check to make sure that fft mode is
           //enabled and waterfall isn't.
           if(!up & is_fft & !is_waterfall) begin
             state_01 <= UP_INCREMENT;
           end else begin
             //If down goes low we are turning
             //counter-clockwise and need to
             //increment down. Check to make sure
             //that fft mode is enabled and
             //waterfall isn't
             if(!down & is_fft & !is_waterfall) begin
               state_01 <= DOWN_INCREMENT;
            end
          end
         end
       end
       UP_INCREMENT: begin
         if(reset) begin
           state_01 <= RESET;
         end else begin
            //Make sure we don't past our maximum.
           if(zoom_out_01 < 'd503) begin
             zoom_out_01 <= zoom_out_01 + 'd10;
            end
            state_01 <= WAIT_NORMAL;
          end
        end
        DOWN_INCREMENT: begin
          if(reset) begin
            state_01 <= RESET;
          end else begin
             //Make sure we don't get a negative offset.
            if(zoom_out_01 > 'd9) begin
              zoom_out_01 <= zoom_out_01 - 'd10;
            end
            state_01 <= WAIT_NORMAL;
          end
        end
        WAIT_NORMAL: begin
          if(reset) begin
            state_01 <= RESET;
          end else begin
            //Wait until our signals both go high
            //signifying we are idle again.
            if(up & down & is_fft & !is_waterfall) begin
              state_01 <= IDLE;
            end
          end
        end
        default: state_01 <= RESET;
    endcase
    //Set the position out to the current scale.
    zoom_pos_out <= zoom_out_01;
  end
  //Same as above with different limits
  2'b10: begin
    case(state_10)
        RESET: begin
          zoom_out_10 <= 'd0;
          state_10 <= IDLE;
        end
        IDLE: begin
          if(reset) begin
             state_10 <= RESET;
          end else begin
            if(!up & is_fft & !is_waterfall) begin
            state_10 <= UP_INCREMENT;
          end else begin
            if(!down & is_fft & !is_waterfall) begin
              state_10 <= DOWN_INCREMENT;
           end
         end
        end
      end
      UP_INCREMENT: begin
        if(reset) begin
          state_10 <= RESET;
        end else begin
          if(zoom_out_10 < 'd779) begin
            zoom_out_10 <= zoom_out_10 + 'd20;
          end
          state_10 <= WAIT_NORMAL;
        end
      end
      DOWN_INCREMENT: begin
        if(reset) begin
          state_10 <= RESET;
        end else begin
          if(zoom_out_10 > 'd19) begin
            zoom_out_10 <= zoom_out_10 - 'd20;
          end
          state_10 <= WAIT_NORMAL;
        end
      end
      WAIT_NORMAL: begin
        if(reset) begin
          state_10 <= RESET;
        end else begin
          if(up & down & is_fft & !is_waterfall) begin
            state_10 <= IDLE;
          end
        end
      end
      default: state_10 <= RESET;
    endcase
    zoom_pos_out <= zoom_out_10;
  end
  //Same as above with different limits.
  2'b11: begin
```

```
case(state_11)
  RESET: begin
    zoom_out_11 <= 'd0  ;
    state_11 <= IDLE;
  end
  IDLE: begin
    if(reset) begin
       state_11 <= RESET;
    end else begin
      if(!up & is_fft & !is_waterfall) begin
        state_11 <= UP_INCREMENT;
      end else begin
        if(!down & is_fft & !is_waterfall) begin
          state_11 <= DOWN_INCREMENT;
         end
       end
    end
  end
  UP_INCREMENT: begin
    if(reset) begin
      state_11 <= RESET;
    end else begin
      if(zoom_out_11 < 'd867) begin
        zoom_out_11 <= zoom_out_11 + 'd30;
      end
      state_11 <= WAIT_NORMAL;
    end
  end
  DOWN_INCREMENT: begin
    if(reset) begin
      state_11 <= RESET;
    end else begin
      if(zoom_out_11 > 'd29) begin
        zoom_out_11 <= zoom_out_11 - 'd30;
      end
      state_11 <= WAIT_NORMAL;
    end
  end
  WAIT_NORMAL: begin
    if(reset) begin
      state_11 <= RESET;
    end else begin
      if(up & down & is_fft & !is_waterfall) begin
```

```
                state_11 <= IDLE;
              end
            end
          end
          default: state_11 <= RESET;
        endcase
        zoom_pos_out <= zoom_out_11;
      end
    endcase
  end
endmodule
```