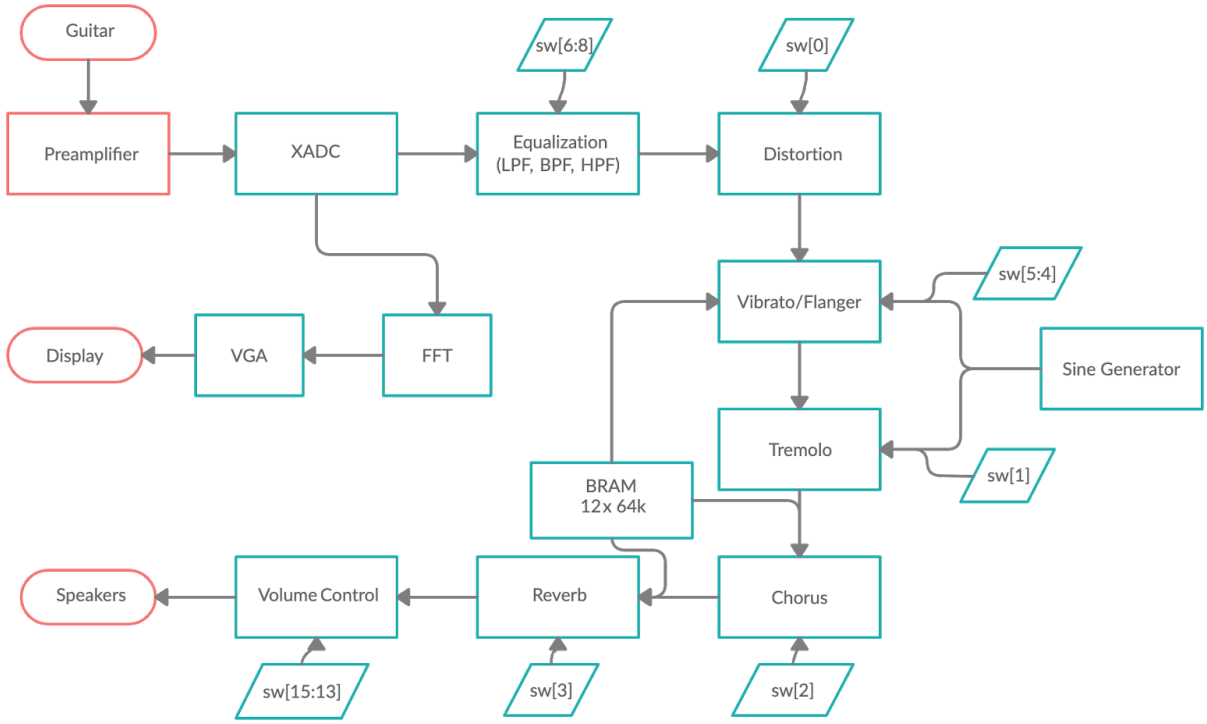
6.111 Final Project Report: FPGAmp20

Dongjoon Lee
Allan Garcia-Zych

Project Overview

The goal of our project was to take in an analog signal from an electric guitar, process it onboard the FPGA with a variety of effects, toggled by switches, and output audio to speakers. To accomplish this goal, we had two major components. First, we needed a preamp circuit which would take the electric guitar signal with approximately 100mV RMS and amplify the signal so the ADC on the FPGA could read it. Second, we needed to implement a variety of effects that could be toggled on and off to process the sound signal.

Block Diagram



Pre-Amp Circuit

The pre-amp circuit was provided by Joe Steinmeyer and is essentially an op-amp based amplifier. The circuit is powered by the 3.3V power supply from the FPGA and the output voltage can be tweaked by the 10kΩ potentiometer at the output. The input signal strength depends on the specific guitar as well as onboard volume and pickup controls, so the potentiometer in the pre-amp circuit had to be tuned to the particular guitar and settings used. If the output voltage is too high for the FPGA ADC, then the ADC clips information is lost. If the output voltage is too low,

the guitar signal becomes inaudible.

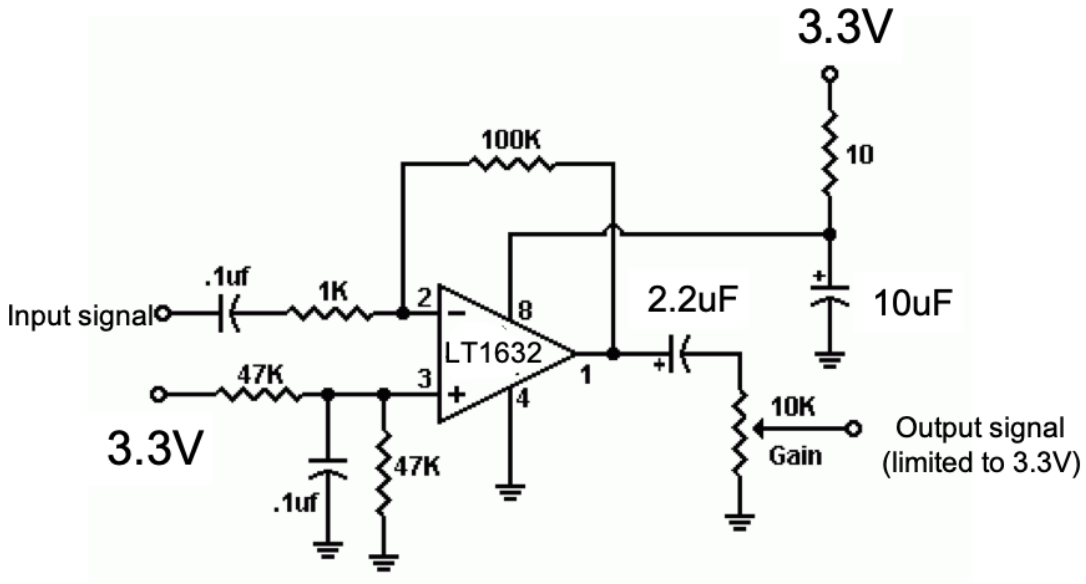


Figure: The Pre-Amp Circuit

Multi-Effects

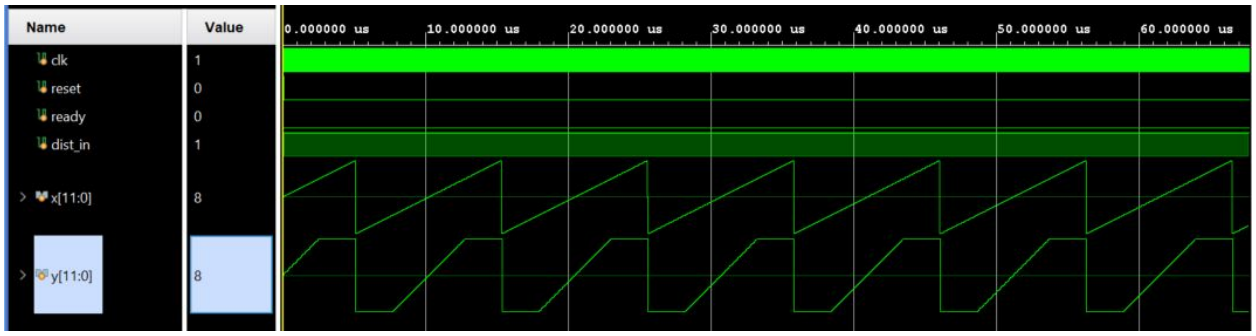
As mentioned in the project overview, one of our main goals for the project was to implement a variety of popular guitar effects on our FPGA-based amplifier. Typically, gaining access to these kinds of guitar effects requires either multiple standalone guitar pedal purchases, or an expensive software suite to be used with an audio interface to a personal computer. Our aim was to demonstrate that these effects could be implemented together on our FPGA without additional equipment, as well as to demystify how they worked.

An important overarching factor in our design was the desire to build a system without any perceptible latency for the user. This meant trying to fit as much of our processing as possible between samplings of the guitar signal. The speed of the Nexys FPGA's clock was helpful in this effort, since even while sampling at 48kHz, we still had a few hundred clock cycles to work with. However, we could not time all of our modules with just the one ready signal as in lab 5a, since that would cap our effects per cycle at one. Instead, we ensured that every effect module not only received a `ready_in` signal, but also provided a `ready_out` signal which could be connected to the next module in the chain.

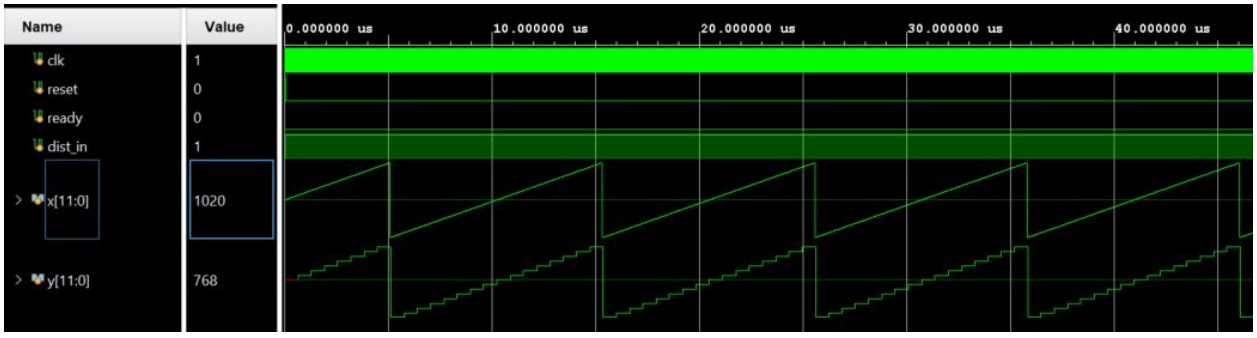
Distortion (Dongjoon)

Distortion has many types of implementations. In our case, we tried two different methods-- hard clipping and bit crunching. Hard clipping takes the input signal and multiplies the amplitude by a certain gain until the signal clips, at which point the amplitude clips at the maximum value. Bit crunching works by reducing the fineness of the quantization which results in an audio signal with a set level of discrete amplitudes. Between the two implementations, the bit crunching worked slightly better. Given a 12 bit audio signal, we right shifted 8 bits then left shifted 8 bits, which threw away the 8 least significant bits. This also had the side effect of becoming a noise gate because all of the ambient noise from the circuitry and the guitar would have a signal strength below the threshold to hit the first quantization level, and would instead round down to zero. Rounding the signal down has an effect of decreasing the signal strength or the volume of the output sound. This was good for our case because bit crunch distortion has a relatively rough edge to its sonic characteristics and turning down the volume this way made the effect sound more pleasing.

The first waveform is hard clipping. The gain is set to two and if the multiplication overflows, then we simply set the output to the maximum or the minimum.



The below waveform shows bit crunching. The distinct quantized levels are visible in the output signal and produce a distorted sound.



Tremolo(Allan)

Tremolo is one of the most commonly used guitar effects, frequently built into consumer grade amplifiers. The functioning of a tremolo system is relatively easy to understand. The guitar signal's amplitude is modulated by a low frequency oscillator, which has the effect of periodically reducing and increasing the volume of the output. In our case, we attempted first to modulate the guitar signal by a triangle, or sawtooth wave, which we represented as an incrementing counter which would periodically overflow and restart counting. Choosing the frequency of the volume modulation was then as simple as selecting an appropriately sized counter, so that it would overflow at a pleasant frequency. However, we suspected that the jumps from maximum value to minimum value were too harsh for our speaker output, and as expected, this resulted in a harsh clicking noise at the end of every crescendo.

This motivated a switch to a sine wave modulation, based off of the sine-wave generator used in lab 5a. It uses a lookup table and a step trigger to generate values from a minimum near zero, to a maximum of 255, much more gradually than the triangle wave. Naively multiplying our input by numbers greater than 1 would of course have led to undesirable signal distortion, so it is important to note that we normalized our modulated signal by right shifting the input after multiplication. Although initially we allowed the sine wave to reach zero, we discovered that this also led to a harsh clicking sound whenever the input signal was completely nullified. To prevent this, we simply modified the lookup table to never reach zero, instead changing slope at a minimum value of 1. Our tremolo circuit can be activated by the second switch in the array of switches on the Nexys board.

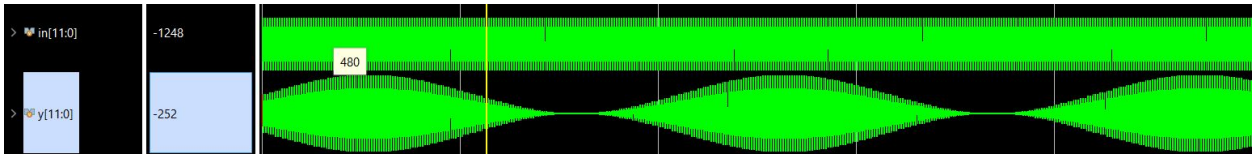


Figure: An Audio Signal Modulated by a Sine Wave Tremolo

Chorus(Allan)

The chorus guitar effect is meant to give the impression that the listener is hearing not just one, but multiple different electric guitars and players, all playing the same notes. Of course, no two humans can be perfectly in sync, nor can they all be located exactly the same distance from the listener. This results in the listener hearing not just one unified signal, but multiple copies, each delayed by a very small amount from the original. This can be achieved on the FPGA by utilizing the onboard BRAM, and address pointers trailing the write address for the current sample. The amount of BRAM necessary is relatively small, since the delays involved in a realistic chorus sound are on the order of milliseconds, rather than longer delay based effects such as reverb. Since none of the delayed copies of the signal should be quieter than the others, it becomes important to normalize the combined signal. This is achieved by accumulating the 12 bit samples

into a 14 bit variable, and right shifting this value by two just before sending the output to the next stage of the signal chain. Our reverb uses 4 small delays, and two BRAMs, which allows us to perform all the necessary computations within 5 clock cycles: one to record, a second to prepare to read the first two delayed samples, a third to collect the first two and prepare to read the second two, a fourth to collect these, and a fifth to prepare the signal for output and return to the first state. The number of cycles could be reduced by adding more BRAMs, and conversely, the amount of memory used could be reduced by increasing the processing time and using fewer BRAMs for the same number of delays. A more realistic chorus could perhaps be implemented by finding a way to generate random delay times in the neighborhood of a few milliseconds, to mimic the way in which real musicians are never consistent in their variations from each other. Our chorus is controlled by the third switch in our FPGA's array.

Reverb(Allan)

Reverb is perhaps the most essential guitar effect, even if its impact on the sound of the guitar is relatively subtle in most use cases. It seeks to replicate the acoustic qualities of concert halls or practice rooms, where the sound of the music played is reflected around the space, subtly echoing the original signal. Our implementation is similar to that of lab 5a's audio recorder in that it achieves this effect by storing incoming samples into BRAM, and adding previous samples, reduced in volume, at strategically placed delays, to the original signal. However, our implementation differs in that for a live guitar performance, the reverberations need to be added to the signal in real time, rather than after recording has been completed. This requires a finite state machine capable of first writing a sample to memory, then reading multiple delayed samples and adding them to the original signal, all before the next sample is fed into our system. Our delays are implemented with trailing pointers, rather than FIR taps, since our delays are relatively far from the head of the BRAM, which would necessitate a very long FIR filter, which would for the majority of its computation be performing unnecessary calculations. We use two BRAMs and output 6 delayed samples in addition to the dry signal, to produce a realistic sounding reverb in relatively few clock cycles. As in chorus, it would be possible to change our implementation by trading time performance with space performance, depending on the particular limitations of the hardware. Reverb is controlled by the fourth switch in the FPGA's array.

Vibrato/Flanger(Allan)

The final delay based effect combines aspects of the approaches of delay with part of the tremolo implementation to reproduce the vibrato effect on our guitar's input signal. Vibrato, often mistaken for its cousin tremolo, is the time based pitch variation of notes played on a stringed instrument. Skilled players are capable of subtly modulating above and below their target pitch on their instruments, which provides additional expressive qualities to their playing. To achieve this on our FPGA using less artistic quality, we again turned to delay, except this time, a time varying delay which was controlled by the same kind of low frequency oscillator as our tremolo.

This generates the perception of a pitch shift in much the same way as when an ambulance drives by with its siren blaring. The input signal is stretched along the time axis, as if the player's location was oscillating towards and from the listener. When the signal is delayed by a longer time, the pitch shifts more, and when it is delayed by less time, it shifts less. All that is left is then to choose an appropriate range for the oscillations, as well as an appropriate frequency. In our case, we used a similar frequency as for the tremolo, in the neighborhood of 2-5 Hz, and a range of delay between 1 and 120 samples. The vibrato sound is just the delayed samples, and can then be toggled by the fifth switch on the FPGA's array.

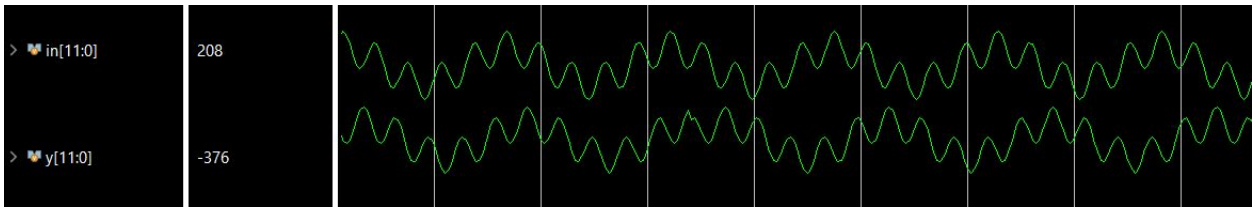
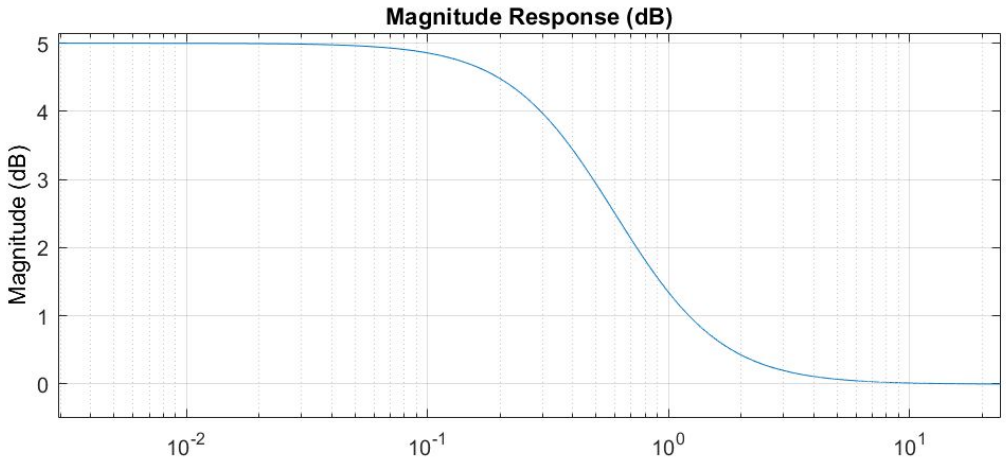


Figure: The Effect of Vibrato on a Dry Signal

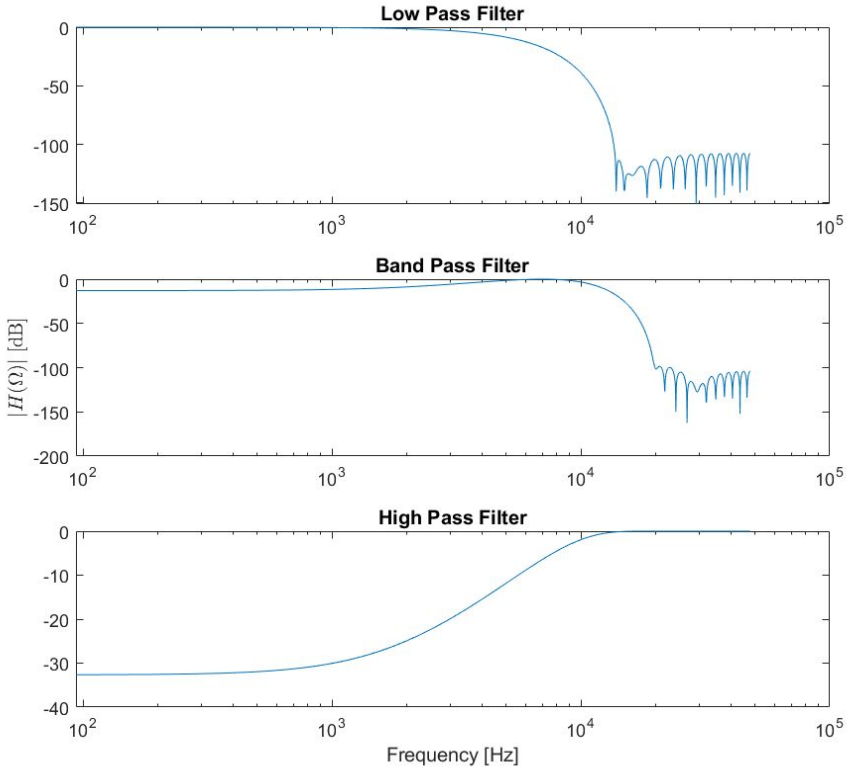
If the signal output by the vibrato circuit is combined with the original signal, the output is effectively an entirely new effect, known as flanger. Flanger combines a time varying delayed signal with the original, and the interference patterns provide a unique resonating quality to the sound of the electric guitar. Rather than implementing this as an entirely new module, we connected another user input switch to the vibrato module, which determines whether or not the dry signal is mixed in. Although this means that you cannot have both vibrato and flanger running simultaneously, it saves significant space on the FPGA. Our flanger is controlled by the sixth switch on the FPGA's array.

Equalization (Dongjoon)

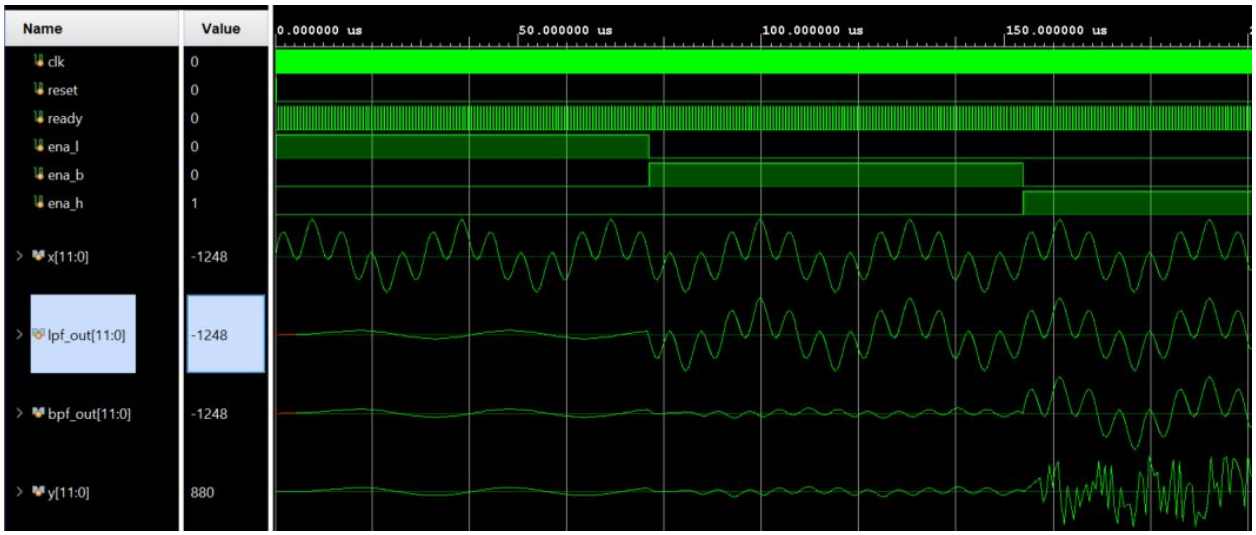
In traditional music signal processing, equalization is done using shelving and peak filters that boost a specific range of frequencies. Our approach for equalization began with using MATLAB for designing these filters. We worked on a variety of filter implementations and compared the tradeoff between performance and feasibility of implementation. The shelving and peak filters were relatively easy to design in MATLAB and had great predicted performance on the Bode plots, but a lot of the actual filter implementation and math was black-boxed within the audio processing toolbox and we had access to a few coefficients that defined the IIR.



In the end, we decided to implement three filters - a low pass, bandpass, and a high pass filter using FIR and convolution and toggle each on and off separately. With the actual filter design, we settled on using the MATLAB fir1() function with a Chebyshev window. Both the low and band pass filters had a certain high frequency pitched noise. Conversely, our goal with the high pass filter was to boost the harmonics of the notes and attenuate the fundamental frequency, but instead, we heard much more of the noise than the harmonics. The most likely explanation is a numerical error in carrying the outputs of the filters.



Shown below is a testbench of all three filters. The first waveform is the analog input and the waveforms below are the output of the low pass, bandpass, and high pass respectively. The enable signals for each filter are also shown and the difference between low pass bandpass and high pass are visible.



Reach Goals (Dongjoon/Allan)

We wanted to implement a variety of effects in the frequency domain such as pitch shift and harmonization, but weren't able to complete the implementation in time. We had the FFT implemented, taking 1024 samples of the audio downsampled from 48kHz to 12kHz. In selecting the parameters for the FFT, there were three main considerations. First, we wanted to be able to take the FFT and IFFT quickly enough that we would be able to keep up with the musician with minimal latency when doing FFT based effects. Second, we wanted to have enough frequency bins at the low end since the lowest note of the guitar is an E2 at 82.5Hz. Third, we wanted to minimize the necessary BRAM usage to be as efficient as possible.

The dominating constraint was the low frequency bin resolution because we absolutely needed more information in the lower end. When looking at the FFT of the default values (48kHz and 1024 samples), we found that the higher frequencies had almost no energy and all the harmonics of the notes died below the noise past approximately the 6th harmonic. Therefore, we only needed approximately 6 times the highest fundamental frequency of the guitar which would be approximately 6 * 1200Hz = 7200Hz. Therefore, we downsampled from 48kHz to 12kHz which is barely below the Nyquist rate for being able to recognize the highest harmonic of the highest note. Our current implementation takes 1024 samples which would mean the frequency bins have a width of approximately 10Hz. Ideally, we would want slightly more fineness, but we initially

wanted to just get the FFT and IFFT working and producing sound and hoped to tackle the increased frequency resolution in the future.

We struggled with getting audio out of the IFFT. The FFT module works in the inverse direction with a single bit parameter, and we had the module set up for the FFT in both directions, but it was a challenge to line up the outgoing samples from the FFT since the data would come in bursts.

Conclusion and Lessons Learned

Over the course of the past few weeks, we were able to use our FPGA kits to build a working guitar amplifier and multi effects unit, featuring some of the most popular electric guitar effects. Our experience building FPGamp 20 was rewarding, though of course there are aspects of our work on the project that we feel that we could have done better. First, we were impressed by the quality of sound that we were able to achieve with minimal equipment beyond the basic FPGA kit that we received. Additionally, we found it interesting to learn how a variety of popular guitar effects work, and surprised by how relatively simple they are conceptually. It was also beneficial for the two of us to be working from the same location, rather than virtually, as many other project groups did.

One thing we realized pretty late through the project was that although 48kHz sampling rate is standard for recording music, it wasn't absolutely necessary in the case of electric guitar. Beyond approximately 7kHz, we only saw noise and the upper harmonics died out pretty quickly. It could have been better to instead sample at 12kHz instead and have more time to process the signal between each sample. The FFT module did downsample to 12kHz, but it wouldn't have hurt to downsample every part of the audio signal chain. We also could have used 16 bit audio rather than 12 bit. We stuck with 12 bits in case we have timing issues with computation, but having more quantization could have helped the sound and reduce the distortion.

In all, we were satisfied with our project and the quality of the output sounds and effects. Although better planning and foresight could have helped us accomplish more, we enjoyed the progress we made over the last month in being able to implement a lot of cool effects and testing out what worked and didn't work in terms of processing the audio signal.

Link to Code and Demo

GitHub Code
YouTube Demo