6.375 Spring 2016 Final Project

Final Report

Group 2
David E. Williams and Lucas L. Sanches

1. Introduction

From the birth of popular music, technology shaped its creation. This effect has touched every aspect of the industry, from the business model to the music itself. One particular interest is how technology has changed the style and sound of music. For example, the electric guitar forever changed music by ushering in rock, while computers caused an equally large shift by enabling modern electronic music, such as techno. The synthesizer was another influential device that changed rock music as well as helped to create electronic music.

Synthesizers were the first purely electrical instruments. Although electric guitars preceded synthesizers, synthesizers produce purely electronic signals while electric guitars convert mechanical signals (the vibration of wires) into voltages that are then processed to produce sound. Due to this mechanical input, mechanical instruments produce distorted waveforms that give the instruments their distinctive sound. Similarly, the unique sound of a synthesizer is due to the perfect waveforms generated by these instruments. This perfect signal has a sound that is nothing like a conventional instrument and has helped to create as well as heavily influenced electronic music. Influential bands such as Kraftwerk and Brian Eno used analog synthesizers prominently in their seminal work.

The first synthesizers were analog. In the eighties however, analog synthesizers were largely replaced by digital synthesizers due to the extreme flexibility and cheapness of digital systems. In addition, digital systems can produce much more complex waveforms because they can easily generate non-integer harmonics and use memory to arbitrarily delay signals

2. Project Objective

It is with this history in mind that this project presents an implementation of a Digital Synthesizer on an FPGA. A digital synthesizer (synth) is an electronic system that uses DSP blocks to generate a wide range of sounds to produce music. A modular synth is a collection of modules that can be connected (patched) in almost limitless configurations to produce unique and interesting sounds. These modules include (but are not limited to) oscillators, arbitrary waveform generators, summers, filters, mixers, envelope generators, distorters, and delays. The goal of these blocks is to generate a stream of audio data based on an input from a keyboard. By making the system programmable and modular, the harmonic content of the waveform can be controlled to generate tones with arbitrary textures and timber. In addition, time

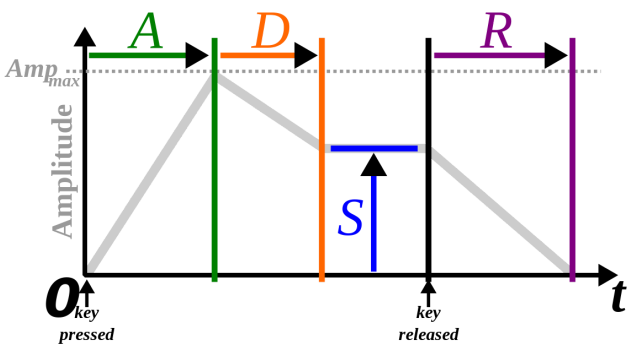
domain modifications can be used to create effects such as vibrato, chorus, tremolo, and reverb.

The goal of this project was to develop a reconfigurable modular digital synthesizer on an FPGA with an Envelope Generator, a Signal Generator, and at least three effects blocks. The advantage of placing a synth on an FPGA is that it enables a large number of DSP calculations to be performed in real time at a much lower cost and power consumption compared to using a processor. The synthesizer takes in information on what notes to play and for how long and generates 16-bit audio sample data at 44,100 samples per second. By ensuring that the system throughput is at least 44,100 samples per second, the synthesizer is able to opperate in real time as an instrument. It would be relatively simple to interface this setup with an ADC and a keyboard control circuit to create a complete hardware synth. This physical construction was outside the scope of the project.

3. Background

Our proposed design leveraged the advantages of digital processing by using additive synthesis to create arbitrary waveforms. The tonality of an instrument, the reason that guitars and violins sound different, is determined by the harmonic content of the note that it plays. In additive synthesis, a bank of oscillators is used to add sinusoids of different frequencies and amplitudes. By carefully selecting the parameters of these harmonics, it is possible to generate arbitrary waveforms such as saw-tooth or square waves. In addition, interesting effects such as beat frequencies and chords can be achieved by using harmonics at non-integer multiples of the note frequency.

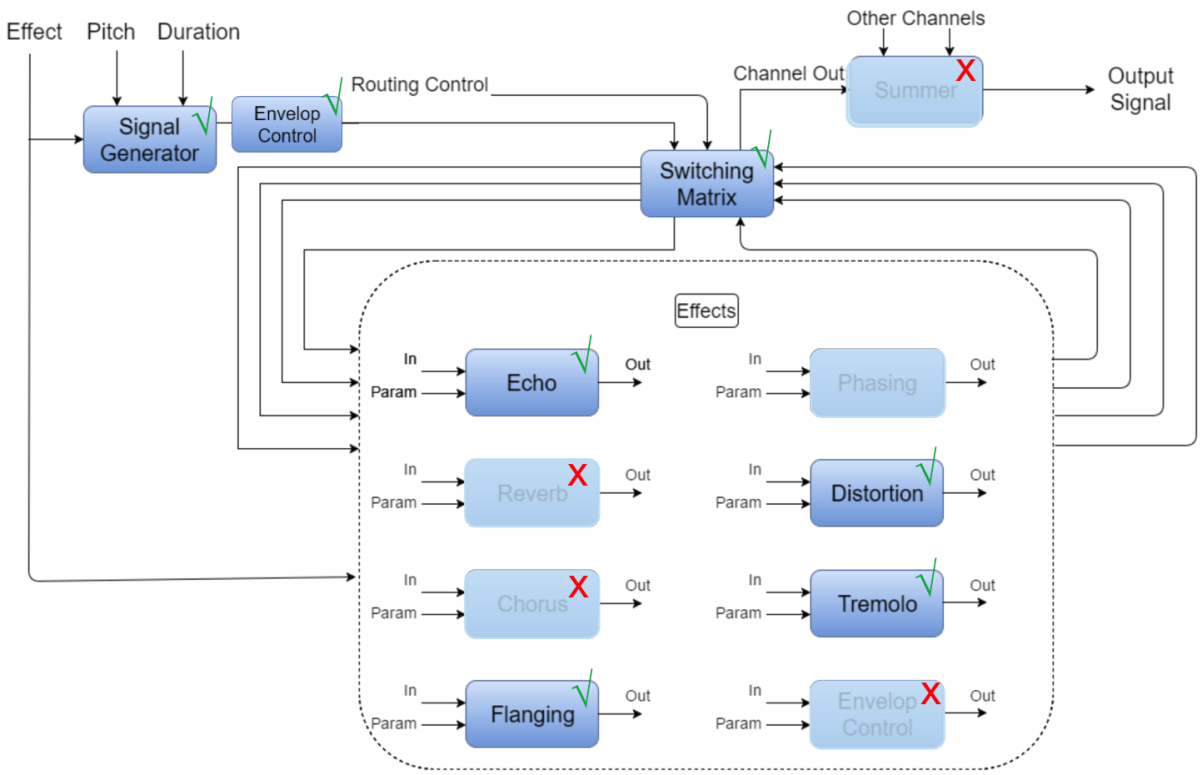
Yet, harmonics are not the only parameters that describe an instrument's sound. Another important aspect is how the instrument's volume changes over time. Therefore, an envelope generator is needed to mimic the distinct percussive sound of a piano, the plucking sound of a guitar, or the gentle rise and fall of a violin. To accomplish this programmability, the envelope generator creates an Attack-Decay-Sustain-Release waveform (ADSR) shown below.



The signal quickly rises to a peak amplitude during the attach phase before dropping to an amplitude that is held for the duration of the note. When the note is released, it slowly decays back to zero. Almost any instrument can be mimicked by making the attack, decay, and release times, in addition to the sustain level, programmable.

While the signal and envelope generator can be used to mimic an instrument, the power of digital circuits is fully maximized by including several special effects blocks to create sounds that are more dynamic. The presented design contains five such effects. The Echo block repeats the inputted signal after a fixed time delay to create and "echo". The Flanging module repeats the inputted signal after a time delay that itself changes over time, creating an interesting sweeping comb-filter effect. Tremolo creates a rapid variation in signal amplitude while distortion distorts the waveform to create additional amplitude dependent harmonics.

4. High-Level Design



The diagram above depicts the high-level diagram of the implemented system. The Signal Generator, Switching Matrix and Envelop Control blocks were considered the most critical ones. The Effects showed were initial ideas before the implementation. The final design had four of them: Echo, Flanging, Distortion, and Tremolo. The location of the Envelope generator was moved to simplify architecture design.

The pipeline behaves in the following matter. A Note struct containing the key, duration, and amplitude of the note to be played is passed to the signal generator. The signal generator passes the generated samples to the envelope generator in addition to how many samples have passed since the note began and how many samples are in the note. The envelope generator uses this time information to generate the appropriate envelope and multiplies the sample data by the envelope amplitude. The output samples of the envelope generator are the routed through an arbitrary chain of effects including Echo, Flanging, Tremolo, and Distortion. In order to maximize the number of sounds that can be generated, the signal chain is reconfigurable without requiring the FPGA to be reprogramed. In addition, any of the effects can be bypassed. The output of the switching network is then returned to the test bench.

Most of the parameters in the system are reconfigurable over Sce-Mi without requiring the FPGA to be reprogramed Because it does not make sense to reroute the signal chain while playing a note, reconfiguration can only occur when all of the blocks and FIFOs are “finished” and do not contain any sample data. If all of the blocks are finished and configuration data is in the configuration input FIFO, the synth enters configuration mode and all of the parameters are altered. Once configuration is complete, the synthesizer will accept note inputs again. Configured and Finished signals are passed to the test bench in order to synchronize the software and hardware.

Configuration data is passed to the synth in a single struct. The top-level module then splits off the configuration data for each individual module (also stored as structs) and passes them to the appropriate modules. This scheme only requires a single Sce-Mi port, reducing interface complexity. However, because a very large amount of data is needed to program the harmonics in the signal generator, the Sce-Mi port is used multiple times so that information on only one harmonic is passed at a time.

In this scheme, the software test bench can dynamically reconfigure the synthesizer and then pass in a series of notes to generate a song. A large number of harmonics in addition to a song, “Karma Police” by Radiohead are hard-coded into the test-bench. The output of the synth is then stored in a pcm file that can easily be converted into a wav file and listened to.

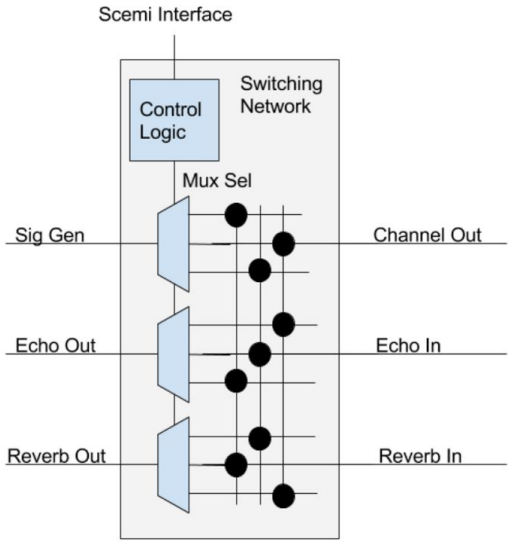
5. Test Plan

To facilitate testing and implementation, we utilized the testing framework developed for the Audio Pipeline laboratories. Inputs to the system were signals signifying what waveform to generate and for how long. The synth then generates a PCM file composed of the appropriate audio signal. Audacity was used to visual inspect the waveforms to ensure that both the shape (sin vs. square vs. saw-tooth) and envelope were correct.

Individual modules were tested with input and expected output PCM files created using MATLAB (or results found for other software implementations). Because the switching matrix is a pipeline, display statements were used to ensure that the effects fired in the proper order.

6. Microarchitectural Description

6.1 Switching Matrix

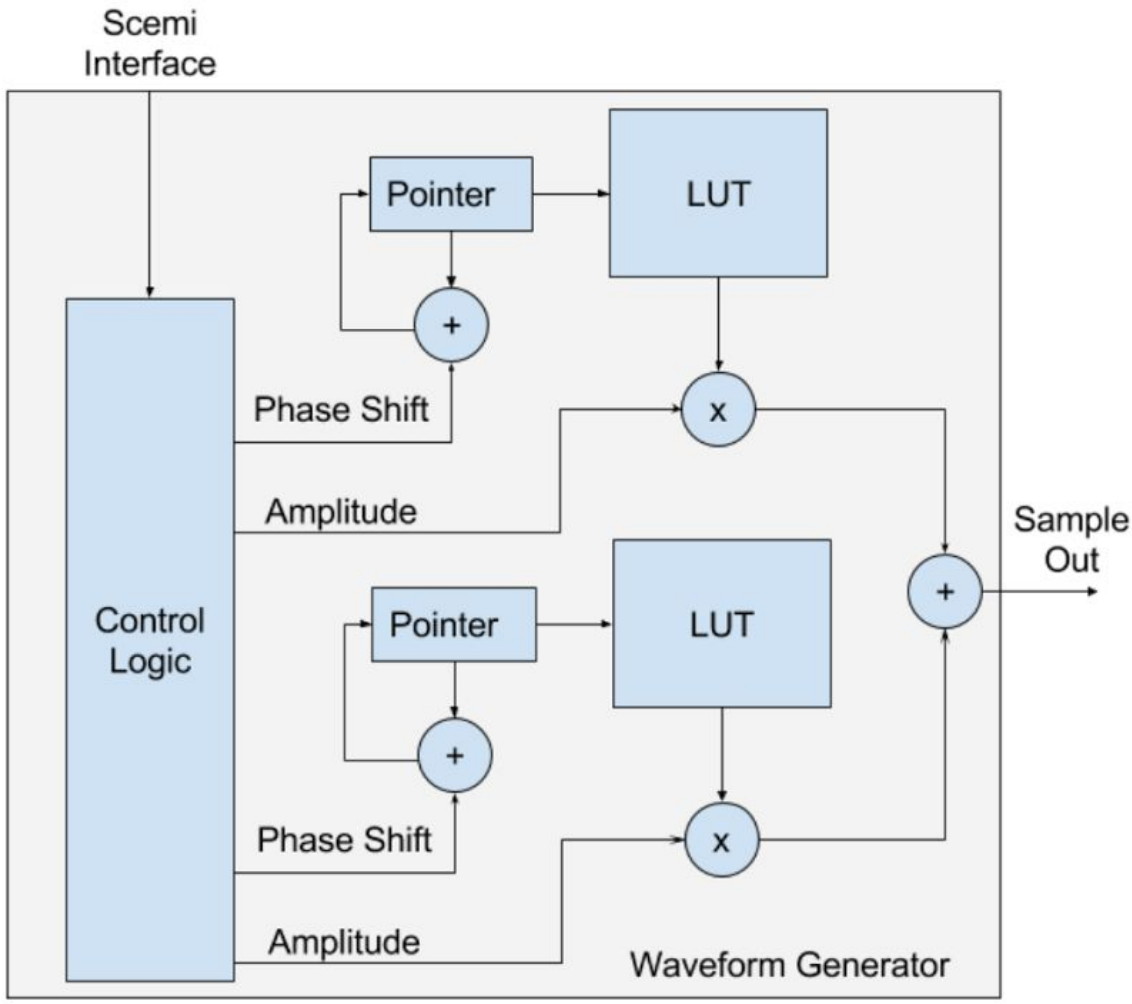


The switching matrix is comprised of a series of multiplexers that redirect the output of each block to the next block. These multiplexers are controlled by a control logic unit that decodes the block order from a Sce-Mi interface and routes the signals accordingly. The calculated selection bits are stored in registers until the system is reconfigured.

For the switching matrix Sce-Mi interface, an enum is used to assign each block a name/number. Because there are a fixed number of blocks, every reconfiguration order can be represented by a list of these numbers. To account for configurations where not every block is used, an additional enum value, NONE, is used to indicate that section of the pipeline should be bypassed. Thus, the Sce-Mi interface to the switching matrix consists of a vector of integers defining the order.

In order for the modules to be completely reconfigurable, the input and output interfaces must be consistent. Therefore, the interface to each module is a 16 bit fixed-point sample. This abstracts away the internal format necessary for the effects computation (such as a complexMP for pitch shifting in Laboratory 4). This interface trades off performance and area for ease of implementation and configurability as multiple blocks that process data in the frequency domain will require separate FFTs and IFFTs. FIFO's are used at the input of each block to aid in pipelining.

6.2 Signal Generator



The signal generator consists of a bank of “oscillators” who amplitudes and frequencies can be adjusted. The oscillator implementation is a BRAM LUT that stores sine wave data calculated at compile time. A pointer indicates what position along the waveform should be read for a sample at a given moment in time. Different frequencies can be generated by incrementing the counter the proper amount for each sample. A second LUT is used to map input key numbers (for example, middle C = 60) to the phase shifted needed to generate the appropriate frequency. The amplitudes read from the lookup table are then multiplied by an amplitude factor and added to the output sample. Up to 100 harmonics can be generated by summing the output of 100 oscillators. The appropriate duration of the signal is generated by using a counter to generate the appropriate number of samples.

Because the LUT is read only, every oscillator in the system uses the same LUT. The size of the lookup table is set by the sample rate and the minimum

frequency. This is because the minimum frequency's pointer will increment the least amount. Because we used the typical audio sample rate of 44,100 Hz with a sample size of 2 bytes, a 1MB LUT (2^16 samples) is used to generate frequencies as low as 20Hz. If necessary, the size of the LUT and/or the sample rate could be reduced to save area or improve the minimum low frequency signal.

To support this arbitrary harmonic generation, a Sce-Mi interface is necessary. It is important to note that while the fundamental frequency of the note to be generated is specified in real time, the structure of the harmonic content is configured similar to the effects and network configuration. Therefore, this process does not occur in real time. Thus, harmonics can be added through multiple calls through the Sce-Mi interface. The number of harmonics that can be generated is limited by the performance of the folded structure and/or the number of parallel generators. Due to BRAM access limitation, the proposed design used a folded structure to save area. The data structure used to store harmonic data is a struct indicating the amplitude, location, and the offset of the harmonic to be added. In order to support non-integer harmonics, the location of the harmonic is generated from the fundamental through a linear function. Therefore, a multiplication factor and offset should be specified for each harmonic. In addition, an enable bit is used to disable harmonics should 100 not be necessary.

6.3 Managing System Reconfiguration

All of the modules have a default state that can be reprogramed. Because it does not make sense to reconfigure the pipeline while signals are still being processed, each module indicates when they are finished. Once all of the modules are finished performing their calculations, the pipeline is reconfigured. In addition, no modules are able to begin calculations until the system is completely configured. Therefore, it is necessary for the entire system to have a state controller that dictates whether calculations can begin. This is easily accomplished through a configuration control sub interface in the top-level block and in each module.

6.4 Effect Block Microarchitectures

The effects modules, each one, execute an effect, modifying the original signal. Each effect was implemented as a different module. As some effects are very

similar in their operation, several effect modules will use the same structure. For instance, Echo and Flanging use a Delay configuration and implementation.

The effects blocks receive integer values as parameters, to execute the specific operations they are supposed to. For the case of Echo, for instance, it receives a 16-bit integer value, representing the amount of delay in milliseconds. Therefore, those parameters produce a queue of registers, which are added back to produce the output signal, with another parameter: the intensity of delay, a fixed-point value that represents percentage of feedback.

In a similar way, Tremolo receives the amount of time that the effect should be performed, as well as its intensity. In the case of Flanging, it receives only a fixed-point value that represents percentage of feedback, to determine the intensity of this effect.

The Distortion block loads a lookup table file to the BRAM, containing the distorted values for all possible entries, calculated by inverse tangent. The table is one to one mapped, so for each input sample, the effect produces an output sample, after retrieving from memory.

Effects that needed to modify the signal in frequency domain weren't implemented. To accomplish that, an FFT module would be needed to transform the signal from the time domain to frequency domain. Then, after the modifying changes, an IFFT would be needed to transform the signal back to time domain. This would use too much space, so none of the effects is performed in frequency domain.
Then, using manipulation of signal amplitude, the original signal can be combined with the chosen envelope, producing the desired output.

The minimum number of effect blocks for the first version of the system was three. We considered this number could provide good combinations of results and a proof of concept for our design. In the end, we designed and implemented four effect modules for the system.

7. Implementation Evaluation

We started the implementation dividing the system in two parts: the signal generator, the switching matrix, and the effect blocks.

For the effect blocks testing, the original Audio Pipeline structure was used. The first effect block implemented was the Echo. It saves samples and adds them later to other ones. The amount of samples stored to be added later depends on the

delay. The parameter passed to this module is used to establish how much samples should be delayed and added later. This addition is represented by a percentage and is passed as parameter.

The second block implemented was the Flanging. Very similar to the first one, it was tested and performed well with a small issue: discontinuities generated by the variation of the delay cause clipping sounds in the output. As discovered later, that noise was added in the very beginning and amplified if the intensity of effect was set as high value. The signal-noise relation in this case is not quite good. Therefore, as tested and realized, to reduce the small, the intensity of the effect should not be too high.

Then, Distortion was implemented. Firstly, we tried to implement mathematical functions to produce a distorted output. All the calculation tried in hardware didn't show exciting results. Therefore, we decided to use a pre-loaded file as lookup table for this module. Using the inverted tangent function, the input sample produces an output one, found in the table. This method showed good results. The file can be changed in a way that the intensity of distortion changes. The problem with that is: every time we want a new configuration for this effect, we need to reconfigure it, loading the different file to the BRAM.

At last, Tremolo was implemented. With variation in amplitude, we could achieve a good example of this effect. The duration of variation is set by a parameter, in the same way that Echo does. The other parameter sets the intensity of the effect, by determining the amount of amplitude variation. Again, if set to a too high value, as the signal-noise relation is not excellent, some clipping sounds can be heard in the output.

In that way, we programmed all the modules for the system. We improved modules and fixed problems that we were finding, as we advanced in the project. The signal generator, envelope generator, and effects blocks were, finally, integrated into a single synthesizer unit. In addition, the order of the modules in the pipeline has been made reprogrammable.

The most difficult part of the implementation was none of the system modules as showed above, but the Sce-Mi interfaces and the BRAM. Initially all of the harmonic configuration data was sent in a single struct in a single Sce-Mi call. The resulting Sce-Mi message was 9971 bits long. Sce-Mi was unable to handle messages of this length so packets were being dropped and communication with the FPGA did not work. This issue was resolved by adjusting the configuration

architecture to only send one harmonic at a time and programing the harmonics through 100 sequential calls. The BRAM produced two challenging problems, the first of which was its inability to locate the initialization file through a relative path. Thus, the LUT was initialized to zero and no data was generated. Second, the initial design called for a LUT with a 19-bit address (for more accuracy). Unfortunately, the BRAM on the FPGA was only able to address up to 16 bits. The resulting waveforms were repeated triangles (the first 10% of a sin wave). This problem was resolved by reducing the size of the BRAM. These issues made the final steps of the project (porting to the FPGA) hard to complete. Although it was difficult and time consuming, the system could be totally operated using the FPGA, at last.

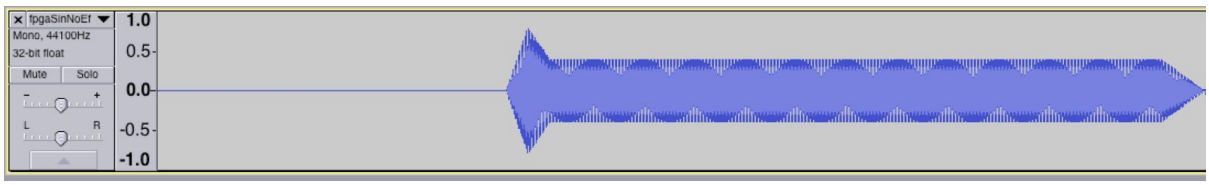
The whole system is composed of 2,608 lines of code, 1,370 lines of BSV and 1,238 lines of C++ in addition to uncounted python code to generate the appropriate BRAM initialization data and to generate c++ code to play “Karma Police”.

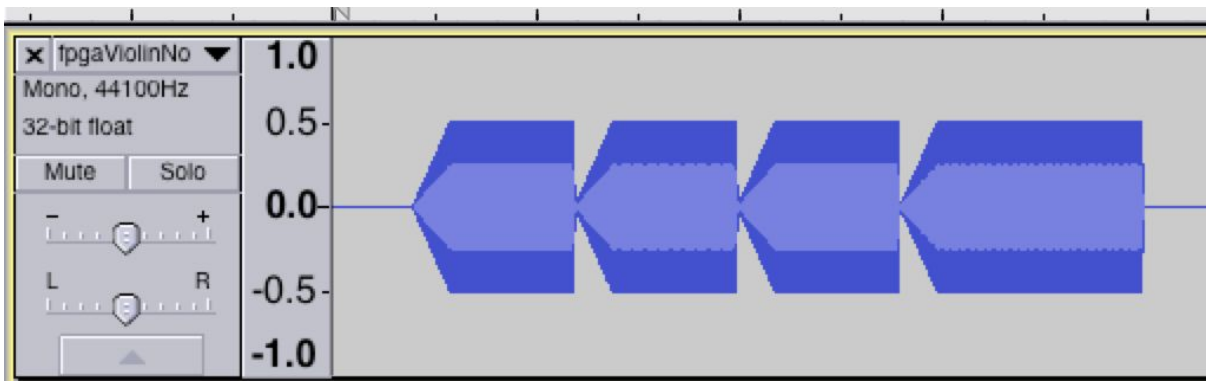
File	Lines of Code
Synthesizer (Top)	292
Synthesizer Types (Type Declarations)	228
Signal Generator	236
Envelope Generator	171
Flange	114
Echo	98
Tremolo	98
Distortion	69
SceMi Layer	89
Test Bench	1238
Total	2,608

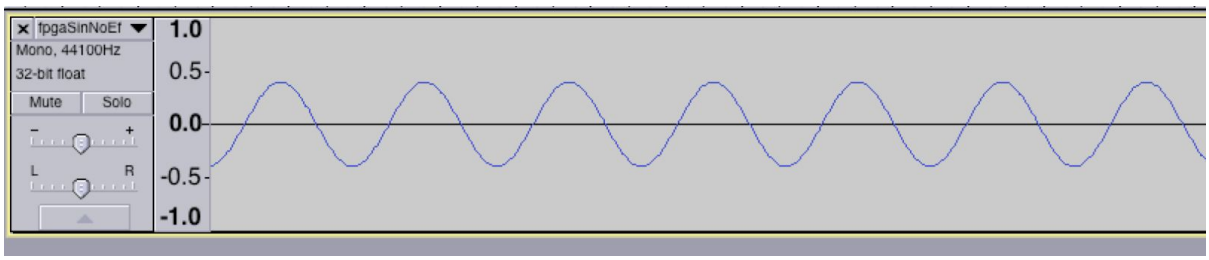
The described system is large. It utilizes 29.8% of the Slice LUTs, 5.5% of the Slice Registers, 2.2% of the F7 Muxes, 1.9% of the F8 Muxes, 6.7% of the BRAM, 28.9% of the DSP blocks. The bulk of this space was used by the BRAM for the Signal Generator (27,328 LUTs), the register file for the delay (27,611 LUTs), and the Sce-Mi interface (13,140).

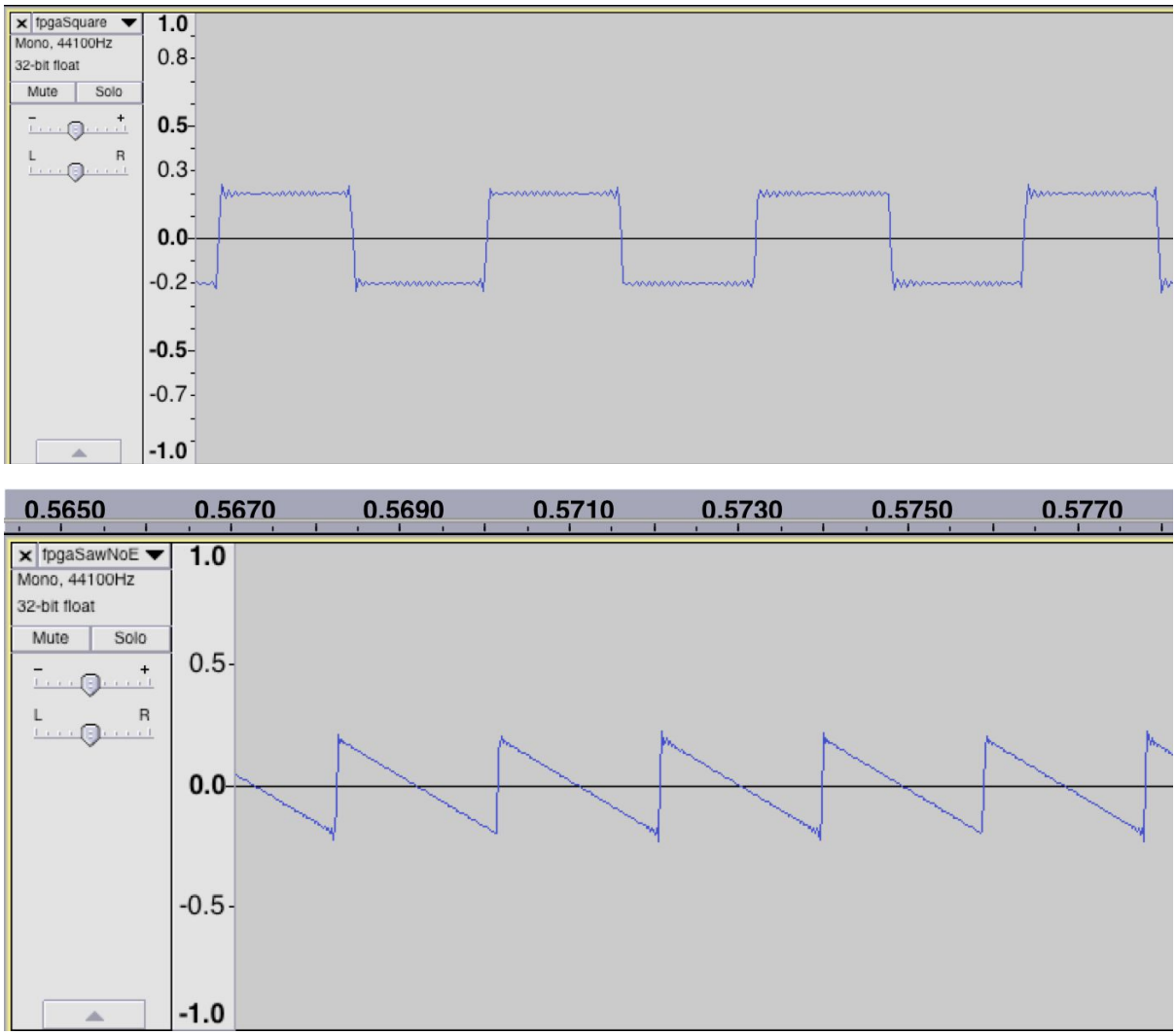
The critical path of our system is within the echo block. Data needs to be read from the regfile, processed, and stored back into the regfile. The path has a delay of 17.323 ns, 98.5% of which is routing delay.

The final synthesizer works beautifully. We were able to accomplish all of the initial project goals including arbitrary waveform generation in real time. Because the system is pipelined, the throughput is fixed by the throughput of the slowest block. Because of its folded structure and the need for 100 harmonics, the signal generator is able to produce a sample after ~ 100 clock cycles (it takes one cycle to load a note but this is amortized over the length of the note). Because the system is clocked at 55.6 MHz the throughput of the system is 556k Samples per Second, well above the 44,100 samples per second needed to operate in real time. Pictured below are two envelope shapes in addition to a sin wave, a square wave, and a saw-tooth waveform formed by the signal generator.









8. Design Exploration

The size of the design appears to prevent it from incorporating multiple channels (at most 3 channels might fit). However, this is not the case. The three largest components, the sin LUT, the echo delay, and the Sce-Mi interface are substantially larger than the rest of the system. Fortunately, all of these can be easily addressed. The size of the sin LUT cannot be reduced arbitrarily without creating distortion because a smaller LUT causes errors in the regions where the slope of the sin wave is largest and thus the amplitude changes the most between “samples”. However, it is likely possible to reduce its size a decent amount without much problem because the sin wave is only accurate to 16 bits anyways. In addition, this LUT can be reused by every channel in the system and thus does not need to be replicated. While this would cost throughput due to the limited BRAM access, our present design has more then enough throughput.

In addition, the size of the echo delay could be greatly reduced (and the critical path improved) by using a BRAM for sample storage instead of a regfile. The inputs and outputs of BRAMs are pipelined and optimized for large data storage so the critical path delay would be greatly reduced and the size would shrink. In addition, echoes with second long delays are not particularly interesting as an audio effect. Therefore the number of samples stored could be reduced to 2^15 and cut the size in half.

Finally, the size of the Sce-Mi interface could be greatly reduced by continuing to reduce the size of the messages being passed and using multiple calls. A clever method would be to use a struct with one variable indicating the parameter to change and another containing the new data value. This would create a very lightweight, expandable, Sce-Mi interface.

With these changes, the size of the synth could probably be decreased by a factor of three or more. The decreased critical path could enable increased folding to reduce size even further.

9. Conclusion

The signal generator can produce waveforms with up to 100 reprogrammable harmonics. For the effects blocks, instead of the first goal of three, we implemented four: Echo, Flanging, Tremolo and Distortion. The Flanging and the Tremolo blocks, because of non-ideal signal-noise relation, have clipping sounds on the output when a too high value of feedback is used. However, considering the general purpose of these modules, they all work as expected. Thus, the system was able to completely and correctly operate as proposed. It was successful implemented on the FPGA.

This project could show the possibility of a digital synthesizer implemented in hardware, embedded in an FPGA. Although it was more complex than expected (most because of interface issues using Sce-Mi and BRAM), it can be said that the main goal set in the beginning of the project was accomplished.

This proof-of-concept example of complex hardware system, therefore, shows the power and advantages of using an FPGA to implement solutions to problems from several fields of study.