FPGA Digital Music Synthesizer

A Major Qualifying Project Report

Submitted to the Faculty of Worcester Polytechnic Institute

In partial fulfillment of the requirements for the

Degree of Bachelor of Science in Electrical & Computer Engineering

By:

Evan Briggs

Sidney Veilleux

April 16, 2015

Advisor:

Professor R. James Duckworth

Acknowledgements

Many thanks to Professor Duckworth for his invaluable guidance throughout the project.

We appreciate Professor Cyganski's generosity in granting us laboratory space for project development.

We would also like to thank Professor Manzo for allowing us to use the resources of the digital music production lab.

Table of Acronyms

ADSR	Attack, Decay, Sustain, Release
AXI	Advanced eXtensible Interface
DDS	Direct Digital Synthesis
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
IIR	Infinite Impulse Response
LFO	Low Frequency Oscillator
LFSR	Linear Feedback Shift Register
MIDI	Musical Instrument Digital Interface
MQP	Major Qualifying Project
NCO	Numerically Controlled Oscillator
PL	Programmable Logic
PS	Programmable Software
SoC	System-on-Chip
WPI	Worcester Polytechnic Institute

Abstract

This project comprised the development of a digital music synthesizer capable of performing subtractive synthesis with several stages of audio effects processing. The system was implemented on the Zedboard, a development board with a Xilinx Zynq System-on-Chip (SoC), a multifunctional device which features a dual-core ARM microprocessor and Field-Programmable Gate Array (FPGA) logic. The completed synthesizer was capable of producing a wide variety of musical tones, many aspects of which were completely controllable by the user. Overall system control was provided by real-time embedded software running on the ARM microprocessor. Waveform generation and synthesizer effects modules were designed in Verilog and implemented in custom logic using fixed-point digital signal processing techniques.

Executive Summary

This project encompassed the design and realization of a digital music subtractive synthesizer with various audio processing effects implemented on Field-Programmable Gate Array (FPGA) hardware. This was accomplished using a Xilinx Zedboard, powered by the Zynq 7000-series System-On-Chip (SoC). The Zynq chip features FPGA logic similar in design to the Artix-7, which provides custom Programmable Logic (PL), as well as an ARM Cortex-A9 dual-core embedded processor capable of executing Programmable Software (PS). Input to the system is provided by a standard USB Musical Input Digital Interface (MIDI) device, which is connected to a computer running Windows. The computer acts as a mediator for input data sent to the Zedboard, as well as supplying a Graphical User Interface (GUI) for control of overall system operation. System output is produced by an audio codec featured on the Zedboard, in the form of a high resolution 24-bit signal sampled at 96 kHz.

Software running in real-time on the Zynq chip's Cortex-A9 microprocessor was utilized to provide overall control of the system. This software handles user input sent by the computer in the form of USB Universal Asynchronous Receiver/Transmitter (UART) serial data, which is received and parsed by the Cortex-A9. Controller input, which is formatted according to MIDI protocol standards, is interpreted and sent to the respective PL effects modules across the Zynq chip via the Advanced eXtensible Interface (AXI) interconnect. This interconnect was also utilized for receiving processed samples from the PL component of the chip. Samples are stored in memory in two 192,000 element arrays for implementation of delay and echo effects, capable of operating with a depth of up to two seconds. The Cortex-A9's Floating-Point Processing Unit (FPU) was utilized to scale the amplitude of outgoing signals according to the velocity at which the current note was pressed, and to perform precise multiplications for the exponential decay of the echo effect. Audio codec initialization and control is conducted in software as well, with output sample rate controlled by a timer-generated interrupt.

Waveform generation and effects processing are performed within the Zynq chip's FPGA logic. The system implements Direct Digital Synthesis (DDS) to produce a ten-octave musical range of 24-bit base waveforms comprising of sine, sawtooth, square, and triangle waves. This

allows for musical notes at 121 discrete fundamental frequencies spanning 8 kHz to be played. Waveform samples are produced at the signal generation stage at a rate of 96 kHz. These harmonically-rich signals undergo four sequential stages of variable high- and low-pass filtering to remove components of the waveform, providing an essential requirement of user-controllable subtractive synthesis.

Synthesizer effects modules designed in Verilog are utilized to alter the signal, producing a wide range of user-defined voices. The use of FPGA hardware allows these modules to perform complex processing in real-time using fixed-point arithmetic. An Attack-Decay-Sustain-Release (ADSR) amplitude envelope is used to modulate the level of the signal over time, allowing the system to model the physical amplitude response of a wide range of musical instruments. Several distortive effects were implemented with the purpose of deconstructing and obscuring the waveform, to emulate the imperfections of older music production technology. A Frequency Modulation (FM) module was designed to create a musical Vibrato effect. Lastly, Low Frequency Oscillator (LFO) modules were implemented to produce automated effect parameter control, allowing for dynamic changes to synthesizer effects using basic waveforms. All effect control signals are routed through the LFO modules, which grants the user the ability to turn any static effect parameter into one which varies over time at a specified rate and depth.

Extensive system testing was performed at each stage of the design process. Many effects were simulated in MATLAB to confirm theoretical functionality before being designed in Verilog. Hardware module verification was performed using Verilog test benches to confirm logic implementations and post-synthesis functionality. Finally, the top-level system was tested via oscilloscope measurements of both analog output signals from the codec and digital internal signals, which were routed to Zedboard package pins for debugging. Software functionality on the Cortex-A9 was verified using tools included in the Xilinx Software Development Kit. The completed synthesizer met all requirements defined at the inception of the project. Overall, the team was successful in producing a stable and fully-functional subtractive synthesizer.

Table of Contents

Acknowledgements ii
Table of Acronyms iii
Abstract iv
Executive Summary v
Table of Contents vii
Table of Figures x
1 Introduction 1
2 Background 6
2.1 Origins of Synthesis 6
2.2 Synthesis Techniques 9
2.3 Synthesizer Effects 10
2.4 MIDI Protocol 10
2.5 AKAI USB MIDI Controller 12
2.6 Zedboard Development Board 12
2.7 Prior Art 14
3 System Design 16
3.1 Overall System Block Diagram 16
3.2 Zynq SoC and Architecture 17
3.3 AXI Peripheral Interconnect 17
3.4 Software on the ARM Cortex-A9 18
3.5 IP Module Generation for FPGA Processing 19
3.6 MIDI Data Transmission to FPGA 20
3.7 Clock Domains 22
3.8 Waveform Generation 23
3.8.1 Sine Wave 25
3.8.2 Sawtooth Wave 26
3.8.3 Square Wave 27

3.8.4 Triangle Wave 28
3.8.5 Complex Waveform Generation 28
3.9 Synthesizer Effects 30
3.10 Audio Codec for System Output 30
4 Synthesizer Effects Implementation 32
4.1 ADSR Amplitude Envelope Generator 32
4.2 Filtering Stages 36
4.3 Distortive Effects 41
4.3.1 Amplitude Clipping and Overdrive 41
4.3.2 Waveform Resolution Reduction 43
4.3.3 Noise Generation 44
4.4 Delay Effects 46
4.5 Low-Frequency Oscillator and MIDI Control 49
4.6 Vibrato 51
5 Testing and Results 54
5.1 Hardware Testing 54
5.1.1 Verilog Test Benches 55
5.1.2 Resource Utilization 56
5.2 Software Testing 56
5.2.1 Debugging Techniques 57
5.3 System Testing 57
5.3.1 Waveform Generation 58
5.3.2 ADSR Envelope 60
5.3.3 Filter Stages 61
5.3.4 Compression and Overdrive 63
5.3.5 Waveform Resolution Reduction 65
5.3.6 Delay and Echo 66
5.3.7 Vibrato 67
5.3.8 LFO 68

5.4 Signal-to-Noise Ratio 70
5.5 Real-Time Capability 70
6 Conclusions 74
6.1 Future Work 76
7 References 78
Appendix A: Overall Block Diagram 81
Appendix B: Development Code 82
MATLAB Scripts 82
Direct Digital Synthesis Waveform Modules Generation 82
FIR Filter Module Generation 90
IIR Filter Module Generation 94
Verilog Code 99
ADSR Envelope Module 99
LFSR Noise Effect Module 102
Compression Effect Module 106
Embedded Software Excerpts 108
Output Sample Processing Interrupt Service Routine 108
Controller Value Update Loop 109

Table of Figures

Figure 1-1: Zedboard Development Board 2
Figure 1-2: Zedboard Architecture 3
Figure 1-3: System Operation Overview 4
Figure 2-1: Musical Telegraph Patent Diagram 6
Figure 2-2: Sequential Circuits Prophet-5 [2] 8
Figure 2-3: Analog Subtractive Synthesizer 9
Figure 2-4: MIDI Data Bytes 11
Figure 2-5: AKAI USB MIDI Controller 12
Figure 2-6: Overview of Zynq Architecture 13
Figure 2-7: FPGA Cyclone Synthesizer Block Diagram [15] 15
Figure 3-1: Overall System Block Diagram 16
Figure 3-2: Zynq Software-Hardware Interface 17
Figure 3-3: AXI Interconnect Signals 18
Figure 3-4: Control Software Flowchart 19
Figure 3-5: GUI for MIDI Data Program 21
Figure 3-6: MIDI Velocity to Amplitude Scale Mapping 22
Figure 3-7: DDS Implementation Block Diagram 24
Figure 3-8: Sine Wave Sample Table 25
Figure 3-9: Sawtooth Wave Sample Table 26
Figure 3-10: Square Wave Sample Table 27
Figure 3-11: Triangle Wave Sample Table 28
Figure 3-12: Sine-Saw Combination in MATLAB 29
Figure 3-13: Synthesizer Effects 30
Figure 3-14: Audio Codec Implementation 31
Figure 4-1: Synthesizer Effects Order 32
Figure 4-2: ADSR Envelope Diagram 33
Figure 4-3: ADSR Top-Level Implementation 34

Figure 4-4: ADSR Module State Transition Diagram 35
Figure 4-5 - ADSR Amplitude Envelope Model (A,D,R = .2 seconds, Sustain level = .5) 36
Figure 4-6: MIDI to Frequency Cutoff Mapping 38
Figure 4-7: FIR Filter Block Diagram 39
Figure 4-8: Sawtooth Waveform with and Without FIR Low-pass Filter 40
Figure 4-9: FIR/IIR Comparison: Sawtooth 41
Figure 4-10: Clipping Effect 42
Figure 4-11: Bitcrusher Effect 44
Figure 4-12: LFSR Noise Generator Frequency Response 45
Figure 4-13: LFSR Block Diagram 46
Figure 4-14: Echo Simulation in MATLAB 48
Figure 4-15: LFO Applied to Signal Amplitude 50
Figure 4-16: LFO Effect Control 51
Figure 4-17: Vibrato Effect 52
Figure 4-18: Vibrato System Level Implementation 53
Figure 5-1: Verilog for Module Testing 54
Figure 5-2: Timing Simulation of FIR Filter 55
Figure 5-3: Verification of Hardware Input via Debugging Tool 57
Figure 5-4: Sine DDS Output 58
Figure 5-5: Sawtooth DDS Output 59
Figure 5-6: Square DDS Output 59
Figure 5-7: Triangle DDS Output 60
Figure 5-8: ADSR Scope Capture 61
Figure 5-9 a-c: FIR and IIR FIlters, Sawtooth 62
Figure 5-10: FIR Filter Comparison (MATLAB to FPGA) 63
Figure 5-11 a-c: Compression and Overdrive 65
Figure 5-12: Bitcrusher Effect Test 66
Figure 5-13: Echo of Signal over time 67
Figure 5-14: Square Signal with Vibrato Effect 68

Figure 5-15: Sinusoidal Compression LFO Effect 69
Figure 5-16: LFO Applied to FIR High-Pass.. 69
Figure 5-17: Effects Processing Throughput 72
Figure 5-18: System Throughput Latency 73
Figure 6-1: FPGA Digital Synthesizer System 75

1 Introduction

Electronic synthesizers have been experimented with for the better part of the last century, but have more recently become popular with the introduction of modern digital music production and universal MIDI standards. Synthesizers are capable of making many unique sounds that are all completely controlled by the user of the instrument. Through the use of tonal oscillators and effects modules implementing signal processing techniques, the number of possible sounds that can be created is seemingly infinite. Due to this high level of tonal customization available on a singular instrument, synthesizers have become a quintessential part of production across many genres of music in the modern era. Early electronic instruments were developed using analog circuits, while modern synthesizers make use of digital systems such as embedded processors and integrated circuits. Given the increased processing capabilities of Field-Programmable Gate Arrays (FPGAs) over the past few decades, the design of a fully-functional synthesizer using this form of programmable logic has become possible.

The goal of this project was to develop a subtractive synthesizer using the Zynq System-on-Chip (SoC) on a Xilinx Zedboard Development Board to read in musician input, perform signal generation and effect processing, and output an analog waveform to be amplified and played through speakers. We designed the system to be capable of producing high-resolution waveforms and executing real-time effects for audio signal output. The Zedboard Development Board is pictured in Figure 1-1, with the Zynq SoC visible in the center of the board.

Figure 1-1: Zedboard Development Board

Knowledge of several engineering disciplines is necessary in order to create a digital synthesizer that meets these requirements. Experience with digital design of programmable logic for FPGAs is essential for the creation of hardware capable of performing high-speed data manipulation. A working knowledge of signal processing techniques such as filtering is also required, as well as the ability to apply this knowledge to the paradigm of digital logic. Fixed-point data is often used for performing precise numerical computations in digital logic, therefore choosing appropriate word sizes and radix positions is crucial for proper system functionality. Knowledge of waveform generation methods such as Direct Digital Synthesis (DDS) is required to generate high-resolution signals in real-time at discrete tonal frequencies for musical applications.

Many audio processing techniques are implemented on modern synthesizers to alter the sound of the instrument. This project was developed with the intent to study, design, and implement a wide variety of effects. These provide the user with a multitude of controllable modules to utilize during performances, allowing for dynamic transformations of the instrument's sound. Implementing these effects in hardware allows for the instantiation of many different modules running simultaneously to perform complex real-time signal processing. Many FPGA synthesizer projects have previously been developed, but none made

use of a System-on-Chip comprising of both a hardcore microprocessor and FPGA logic. The Zedboard features a number of peripherals connected to the Zynq SoC, making it an excellent candidate for synthesizer production. A top-level diagram of the board's architecture can be seen in Figure 1-2.

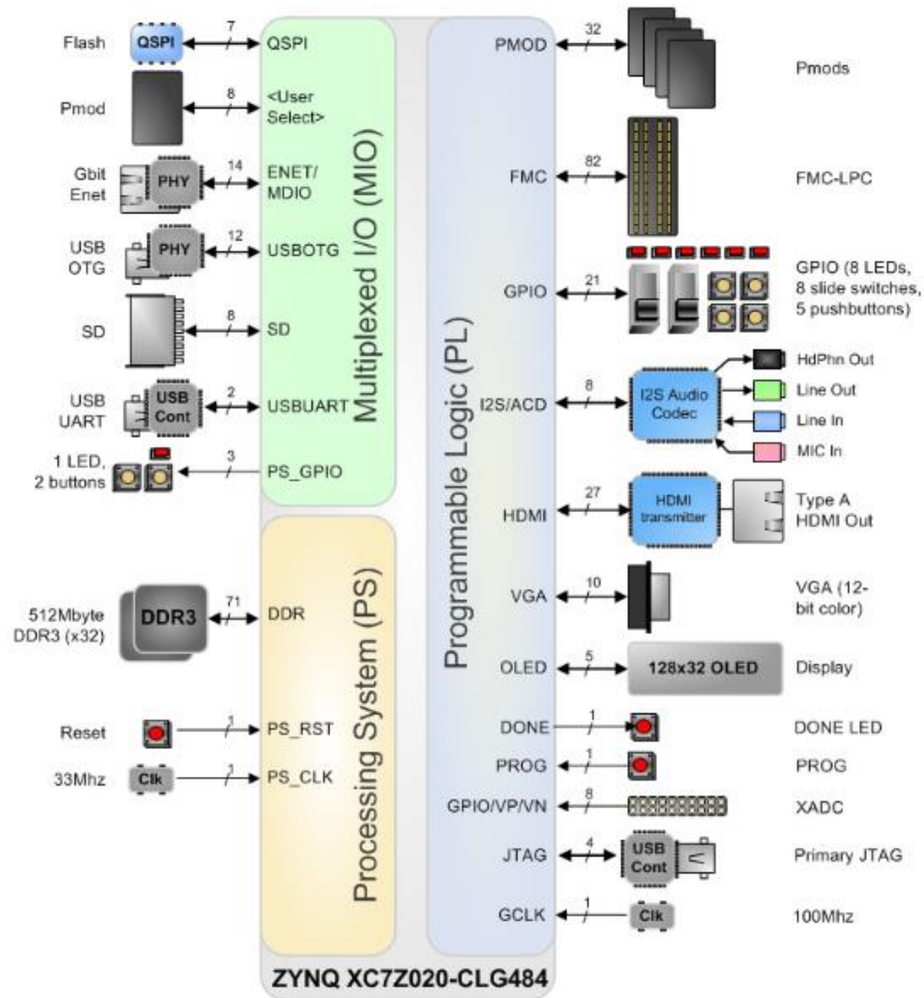


Figure 1-2: Zedboard Architecture

As can be seen in Figure 1-2, the Zedboard features a great number of peripherals for a wide range of applications. We chose this development board for system development because of the unique architecture of the Zynq chip, which offers an array of programmable logic (shown as PL in the figure) connected to a hardcore microprocessor processing system. This allows for the division of system tasks between these two components, in order to achieve the

best possible functionality and quickest overall throughput. Zedboard peripherals used in this project include the I2S Audio Codec, USB UART bridge, and slider switches.

The synthesizer system we designed is composed of five major parts, displayed graphically in Figure 1-3. Musician input is captured by a digital keyboard interface and sent to the Zedboard in the input component of the system. The system control block, implemented on the Zynq chip's Dual-Core ARM Cortex-A9 microprocessor, interprets input and provides control over every component of the system. The waveform generation module, implemented within programmable logic, produces base periodic waveforms. These waveforms feed into the effects processing subsystem (also in custom logic) for signal processing and customizable tone production. The final synthesized signal is converted into an analog waveform at the system output stage to be amplified and played through speakers.

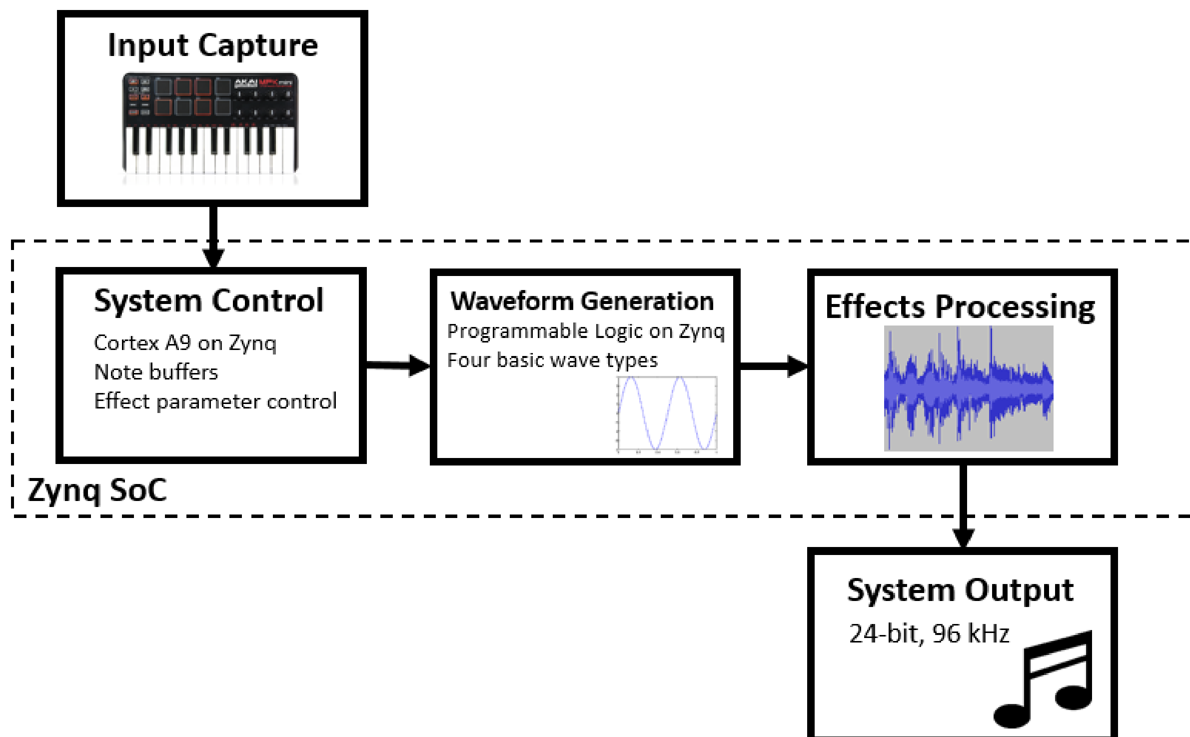


Figure 1-3: System Operation Overview

While a wide range of synthesizer effects exist, this project sought to implement modules usually found in the typical modern synthesizer. This includes several stages of

filtering, which remove components of harmonically-rich waveforms to perform subtractive synthesis. Amplitude Modulation envelopes are also essential to any synthesizer, allowing the device to model the physical response of a specific instrument. Distortive effects, which act to deconstruct or obscure an audio signal, are often used to model the imperfections present in early analog amplifiers and synthesizers. Many synthesizers also make use of auxiliary oscillators operating at low frequencies (0-20Hz), which are applied to the controlling input of various effects modules. This allows for dynamic changes to virtually any effect implemented on the device. These Low Frequency Oscillators (LFOs) are also used to apply frequency modulation to a signal, resulting in a vibrato effect.

Synthesizers are an excellent tool for musicians who wish to create their own unique sounds for recording songs or performing on stage. Developments in synthesizer technology allow these artists to be expressive and show their creativity through the use of waveform manipulation, made possible by the abundance of effects available to them. One benefit of creating such a system was the opportunity for us to learn how each of these subsystems function, providing a tangible application of digital signal processing. As engineers with backgrounds in music as well as in signal processing, we were able to research and put into practice the mathematical theory behind our musical training. This helped to conceptualize the engineering required for the development of a complex musical instrument.

The outline of this report is as follows. The following chapter provides an overview of the origins of electronic music production and the progression of synthesizer technology, leading up to the modern era. This chapter also includes a description of current industry standards and an overview of the Zedboard Development Board. Chapter Three describes the methods of design and implementation that were utilized in the development of our overall system design, including system control and waveform generation. Chapter Four outlines in detail each synthesizer effect, including theory of operation and methods of module development. The fifth chapter describes methods and results of system testing and verification, and the sixth chapter provides conclusions drawn from the completion of the project.

2 Background

A synthesizer is an electronic musical instrument capable of producing tonal signals that can be played audibly for the creation of music. The complexity of this instrument has grown rapidly over the past century, as many advancements were made in the fields of electrical engineering and signal processing during this time. Synthesizers have become increasingly popular in recent years, due to an increase in their commercial availability as well as their wide breadth of applications across the music industry**.**

2.1 Origins of Synthesis

In the late 1800s, the first electric musical instrument was developed by Elisha Gray using electromagnetic circuits to create single-note oscillators [1]. These oscillators were capable of producing simple sinusoidal tones from vibrations at the circuit's resonant frequency. A patent diagram for this device, called the Musical Telegraph, can be found in Figure 2-1, showing individual oscillators for each musical note.

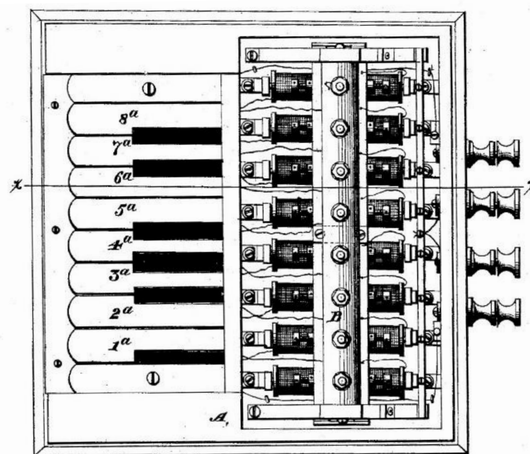


Figure 2-1: Musical Telegraph Patent Diagram

This concept was further developed into what is commonly referred to as a Voltage Controlled Oscillator (VCO), in which the frequency of oscillation is proportional to the circuit's input voltage. VCOs allowed for a more modular and compact design in which one oscillator could produce a wide range of tonal frequencies, which lead to the development of portable analog synthesizers. Analog filters, vacuum-tube amplifiers, and other circuits were used to shape and scale synthesized waveforms, changing the musical timbre of the electronic instrument. As electrical technology advanced over the course of the 20th century, new designs for more complex and commercially-available synthesizers became prevalent world-wide. This expansion grew exponentially in the early 1970's as the synthesizer became a common instrument in popular recorded music. The fully-analog synthesizers of this era made use of continuous time signal processing techniques, with effects subsystems connected in series using patch cables to create modular designs.

In early synthesizers, each effect was designed as a standalone analog module that could be connected to the system sequentially in arbitrary order. This resulted in large, expensive systems that had to be purchased piece by piece. With advancements in the field of digital systems and microprocessors, smaller and less expensive systems were able to be produced. Different methods of synthesis were used to create a wide variety of sounds, to achieve specific timbres for production across many genres of music.

By the end of the 1970s, the first microprocessor-controlled digital synthesizers had become commercial available. These systems sought to recreate the signature sounds of analog synthesizers, with much more compact designs. Tonal generation and effects modules were implemented in embedded software, allowing for more complex signal processing to be performed. This allowed for a wider range of effects to be developed on a single system, implementing the latest developments in the field of digital signal processing. One such device which became quite popular in recorded music of that era, The Prophet-5, is pictured in Figure 2-2 [2]. This system featured analog and digital components, utilizing a microprocessor to for patch memory and effects control. Early digital synthesizer systems were limited by the speed and memory available to microprocessors, which has grown logarithmically over the past 30 years [3].

Figure 2-2: Sequential Circuits Prophet-5 [2]

Modern digital synthesizers make use of high-speed microprocessors to accomplish a multitude of real-time effects processing techniques. The decreased cost of custom silicon production has allowed for the development of many Application-Specific Integrated Circuit (ASIC) powered synthesizers as well. As the name suggests, ASIC designs are custom-tailored to fit a specific application, allowing for the implementation of hardware accelerators for Fast Fourier Transforms and concurrent processing of multiple signals at rates unachievable in software. When considering designs not outrageous in complexity, FPGAs have many of the same capabilities of an ASIC, at reduced speed and much lower cost [4]. Given the schedule limitations of the project, this functionality makes FPGAs an excellent medium for custom synthesizer design.

Through the use of different effects modules, analog or digital, musicians are able to create a wide range of dynamic and interesting sounds to match their style of music. Effects that feature user-controlled parameters allow for many different possible tones that can be used across many genres of music. These effect modules introduce different types of signal processing techniques to create additional sounds that add harmonic complexity to the original signal. Some modules introduce noise to add distortive frequencies to the waveform, while others remove parts of the signal using filtering techniques. Modern synthesizers can feature

hundreds of effects subsystems, allowing for a seemingly infinite number of attainable musical timbres.

2.2 Synthesis Techniques

There are many different forms of synthesis that have been developed for musical applications in both analog and digital synthesizers. The origins of waveform synthesis stem from Additive Synthesis [5]. This method allows the user to add specific harmonics of a base frequency into the desired signal, through the use of multiple oscillators. The Hammond Organ is one of the earliest examples of an additive synthesizer, which implemented analog sinusoidal additive synthesis. With the development of the VCO and Voltage Controlled Filters (VCF), a new synthesis method was developed, known as Subtractive Synthesis [6]. This allows the user to start with a more complex waveform (such as a sawtooth or square signal) and filter off undesired harmonics, resulting in a wide variety of possible sounds. A simple analog subtractive synthesizer is described graphically by the diagram in Figure 2-3. This system also makes use of an Amplitude Modulation Envelope to allow for dynamic control of the gain of the output amplifier.

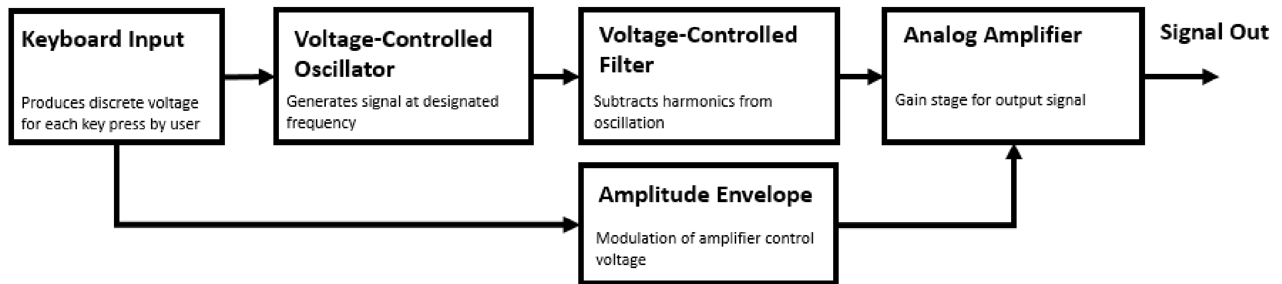


Figure 2-3: Analog Subtractive Synthesizer

Other synthesis methods, such as Frequency Modulation Synthesis, were implemented into synthesizers to create even more possibilities of timbres and sounds [7]. This technique involves modulating the frequency of a base waveform with multiple carrier frequencies in several stages, allowing for high orders of frequency modulation to be performed. This results in the creation of a wide range of different complex tones.

A common replacement for VCOs in the realm of digital synthesizers is the use of Direct Digital Synthesis (DDS). This method of waveform generation makes use of high-resolution tables containing one period of each base waveform, which are stepped through at discrete rates defined by a Numerically Controlled Oscillator (NCO). Stepping through a wavetable at variable rates allows for notes of many frequencies to be produced, essentially replacing the sinusoidal VCO with any desired periodic waveform. The VCF implemented in analog synthesizers is generally replaced by digital filtering techniques. Through the use of DDS and digital filters, the method of subtractive synthesis can be accomplished on a digital system.

2.3 Synthesizer Effects

Countless synthesizer effects have been developed since the origin of electronic instruments. The focus of this project was on the implementation of some of the most common effects, considered essential to any modern synthesizer. Amplitude modulation envelopes, which allow an electronic system to model the physical response of an acoustic instrument, introduce smooth transitions between waveform amplitudes [8]. The Vibrato effect creates an oscillating modulation of a waveform's frequency, simulating the vibrato created by an opera singer [9]. Distortive effects act to reduce the quality of a waveform, to simulate the effects of older analog synthesizers. Noise generation adds broad-spectrum static to the signal, simulating the cracking sound produced by cathode-ray tube amplifiers. Amplitude clipping and pre-amplification effects imitate analog amplifier systems operating beyond their maximum output voltages, resulting in an overdrive effect [10]. The Bitcrusher effect reduces the resolution of a digital waveform, creating grainy sounds associated with early digital systems of inferior sound quality [11].

2.4 MIDI Protocol

The standard protocol for communication between modern digital music devices and computers is Musical Instrument Digital Interface (MIDI). Data transmitted using this protocol is composed of three-byte messages, sent asynchronously over a serial communication port as 24-bits of data. Bytes whose MSB are one correspond to control status signals, while bytes beginning in zero represent parameter values. The following seven bits represent unsigned

values between 0 and 127. In a three-byte message, the first byte is a status signal which alerts the system to the type of input that is being changed. For the purposes of this project, this includes note on/off messages and controller changes. The second byte is a parameter relevant to the type of message being sent. For a note message, this byte represents the value of the MIDI note being played. For a controller change, the second byte corresponds to the number assigned to the particular controller being modified. The final byte of data is a second parameter, which corresponds to either the velocity at which a note was pressed, or the new value of the controller being modified. The selected input device for this project, an AKAI MPK Mini controller, connects to a host via USB, and sends MIDI protocol input for interpretation by the system. [12] Figure 2-4 shows the MIDI data bytes described above in order or transmission.

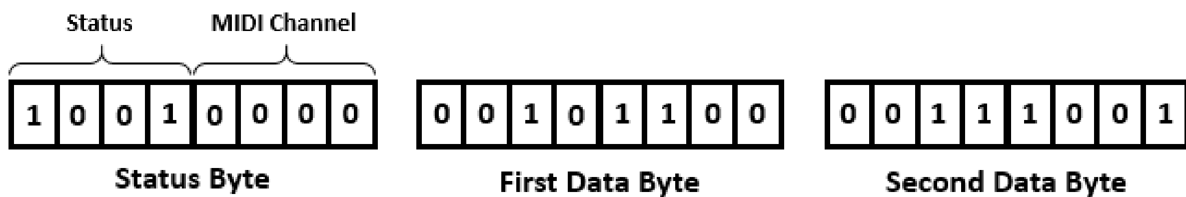


Figure 2-4: MIDI Data Bytes

MIDI parameters utilize the lower seven bits of the data bytes to effectively achieve 128 different parameters per byte. A MIDI device is assigned one MIDI channel to operate on capable of sending and receiving data. For MIDI note data, each integer of the first data byte represents a pitch in the chromatic scale in chromatic order, where MIDI note 60 is representative of Middle C (C4). With this format, increasing a note value by 12 will produce the octave of the note, as there are 12 notes in the chromatic scale. For each pressed note, the velocity (how hard the note was pressed) value is sent over in the second data byte, allowing a synthesizer to alter the volume of the pressed note. For controller data, the format is similar as it allows for 128 different controllers to operate per channel. This allows for control over a multitude of parameters within synthesizer systems.

2.5 AKAI USB MIDI Controller

The AKAI is a 25-key two-octave USB device capable of producing MIDI input data to music systems. Through the use of octave control buttons, the device is able to send note input for MIDI notes 0 to 120, and features 8 controller knobs for the parameterization of effects. MIDI note input from the AKAI includes velocity sensitivity, allowing sophisticated systems to take into account the force at which each note is played. Other MIDI input devices feature more keys and additional control knobs, at much greater monetary cost. This device, which was chosen for our system due to its low cost and USB connectivity is pictured in Figure 2-5.

Figure 2-5: AKAI USB MIDI Controller

2.6 Zedboard Development Board

The Zedboard is a development board produced by Xilinx featuring the Zynq-7000 series SoC, an ADAU1761 24-bit audio codec, and numerous peripherals. The device also features eight binary slider switches, LEDs, five push-button switches, and USB UART (Universal Asynchronous Receiver/Transmitter) connectivity, among other peripherals not used in this project. The Zynq-7000 contains a 667 MHz dual-core Arm Cortex-A9 embedded microprocessor, as well as an Artix-7-equivalent FPGA with 53,200 LUTs (Lookup Tables) and 106,400 Flip-Flops for the creation of complex digital logic. Lookup tables allow for the creation

of any perceivable array of combinational logic, using truth tables to recreate boolean logic gates [13]. The Cortex-A9 processor is capable of running XiLinux, a Xilinx distribution of the Linux Operating System. Communication on the Zynq between the Cortex-A9 and the FPGA is accomplished using the Advanced eXtensible Interface (AXI) Interconnect, a 100MHz bus, which acts as a medium for the control of many system peripherals as well. The ADAU1761 Audio Codec is capable of receiving and transmitting a 24-bit audio signal at sampling rates of up to 96 kHz, via four onboard 3.5mm two-channel audio ports. Figure 2-6 shows a diagram of the overall Zynq architecture and some of the peripherals available on the chip.

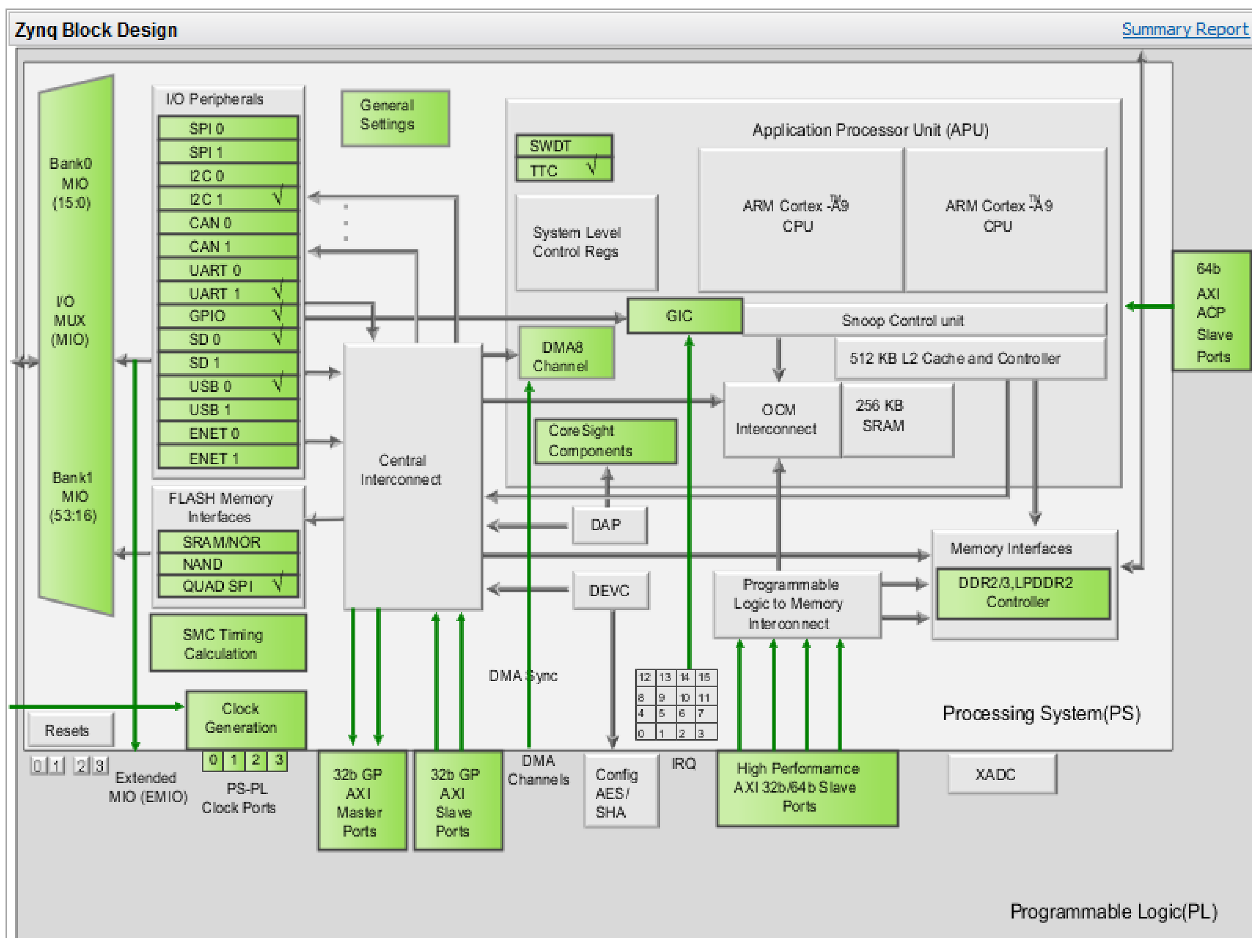


Figure 2-6: Overview of Zynq Architecture

This board was chosen by the team for synthesizer development after much consideration. The team required a board that would be able to accomplish the goals laid out for system design. The Zynq chip boasts a great deal of processing power in both hardware and software, which exceeded the requirements for signal generation and effects processing. The AXI Interconnect allows for high-speed transmission of an arbitrary number of signals between the Cortex-A9 and the Programmable Logic, meeting the requirement of sufficient communication of control signals from software to hardware. The ADAU1761 is capable of producing high quality waveforms, meeting the requirement for system output, with the added bonus of audio jacks for convenient connection of the synthesizer to speakers.

2.7 Prior Art

Several FPGA-based synthesizer projects have been developed by engineers in recent years. Due to the ever-increasing functionality of FPGAs per unit cost, it has become much more feasible to create complex synthesizers which produce high-quality sounds. One such synthesizer, the “FPGA-Synth” was produced on a Xilinx Spartan 3E and featured two NCOs for two simultaneous voices, and a simple filtering stage to perform subtractive synthesis [14]. Another synthesizer, called “FPGA Cyclone Synthesizer”, made use of an Altera C5 FPGA and featured several effects implemented in hardware, including several filtering stages as well as a Vibrato effect [15]. This device, described graphically in Figure 2-7, made use of duplicated hardware modules to produce 16 independent synthesizer voices to implement polyphony.

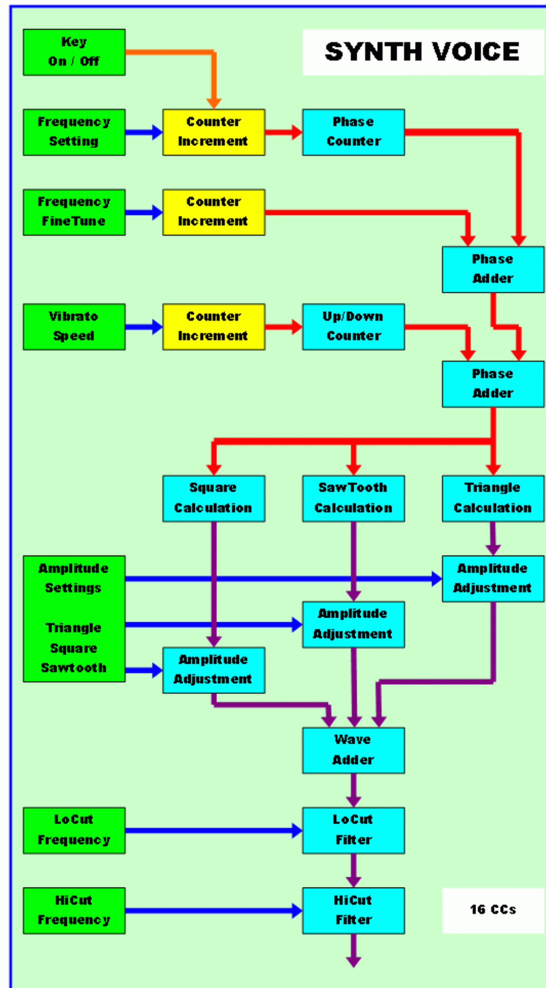


Figure 2-7: FPGA Cyclone Synthesizer Block Diagram [15]

None of the digital synthesizer projects we researched were completed on systems that contained both an embedded microprocessor and programmable logic. Many of the effects chosen for implemented in our system were found in use on these developed projects, but none of the synthesizers utilized as many effects processing stages as our proposed design, likely due to the system limitations of older FPGAs and the lack of a hard-core microprocessor.

The next chapter provides a complete description of the theory and methods of our synthesizer design, as well as system implementation on the Zedboard. This includes overall system throughput, embedded software development, and waveform synthesis module design.

3 System Design

This chapter describes the theory and methods of the overall synthesizer system design and its implementation on the Zedboard Development Board. System control and signal generation methods are discussed, as well as overall synthesizer throughput. This encompasses system input, digital waveform synthesis, and analog signal output.

3.1 Overall System Block Diagram

The system layout is defined graphically by the diagram in Figure 3-1. From a top-level throughput standpoint, the synthesizer operates as follows. MIDI protocol input is produced by the USB input controller, which is connected to a computer running a GUI to multiplex controls. Modified input data is sent serially to the Zedboard and captured by the microprocessor on the Zynq chip, which acts as an overall system control block. Within real-time software, updates to control signals are sent across the Zynq chip to FPGA hardware. In programmable logic, waveform generation and several stages of effects processing take place, with new samples sent back to the software at a rate of 96 kHz. On the microprocessor, a final stage of processing is implemented before the sample is sent to audio codec for system output.

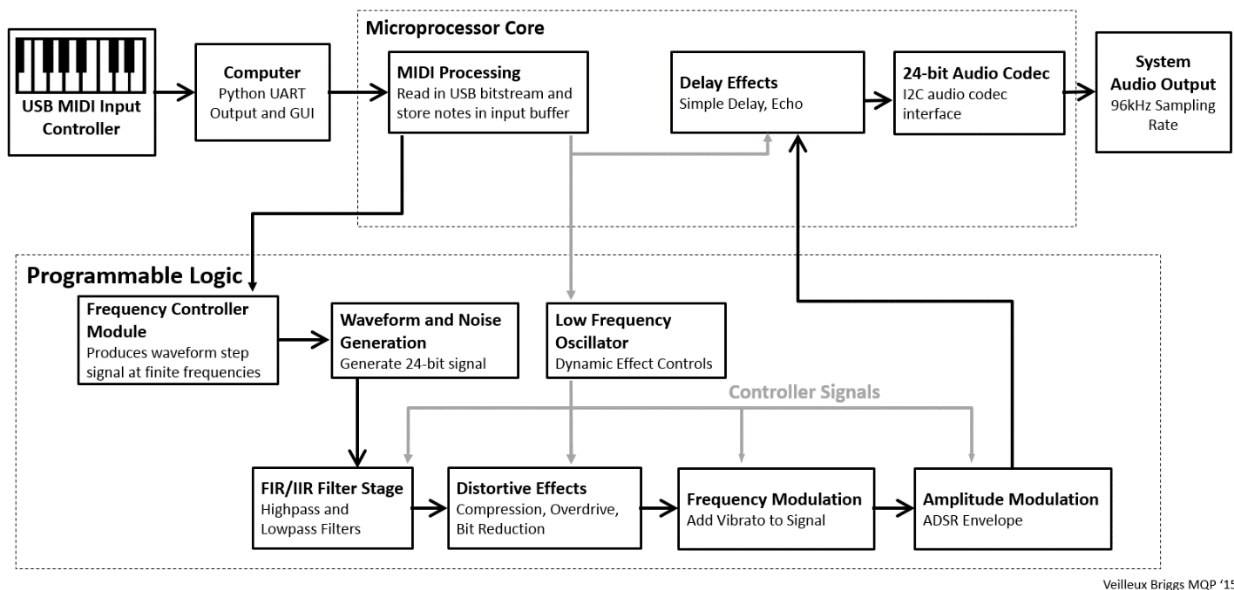


Figure 3-1: Overall System Block Diagram

3.2 Zynq SoC and Architecture

The Zynq SoC was chosen for this project due to the inherent design flexibility that comes with having access to both a microprocessor core and programmable logic on one chip. This allows for high-speed signal processing to be performed in hardware on the FPGA, while memory-intensive tasks (such as delay effect generation) are accomplished in software. The microprocessor component of the chip is used for overall system control, while the generation of waveforms and synthesizer effects are implemented in hardware modules on the FPGA. A top-level view of these two components and the interface between them can be seen in Figure 3-2.

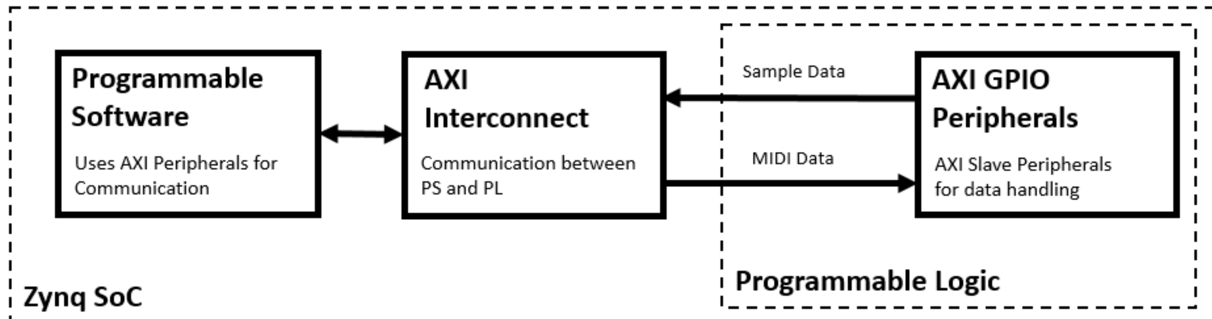


Figure 3-2: Zynq Software-Hardware Interface

3.3 AXI Peripheral Interconnect

The Advanced eXtensible Interface (AXI) Interconnect is used to connect the Zynq SoC with each of the peripherals on the ZedBoard. The AXI Interconnect also facilitates high-speed communication between the two main components of the SoC, the Cortex-A9 and FPGA logic, allowing for the seamless integration of a multifaceted system. This interconnect is utilized for sending MIDI-format note and controller values from the Programmable Software (PS) to FPGA hardware, as well as transmitting 24-bit synthesized waveform samples from the Programmable Logic (PL) to the PS. This is accomplished through the use of a number of AXI General Purpose Input/Output (GPIO) modules, each of which allow for two unidirectional 32-bit channels of data to be transmitted over this 100MHz bus. GPIO value reads and writes are

controlled in software on the Cortex-A9. A graphical representation of the signals transmitted over this interconnect can be observed in Figure 3-3.

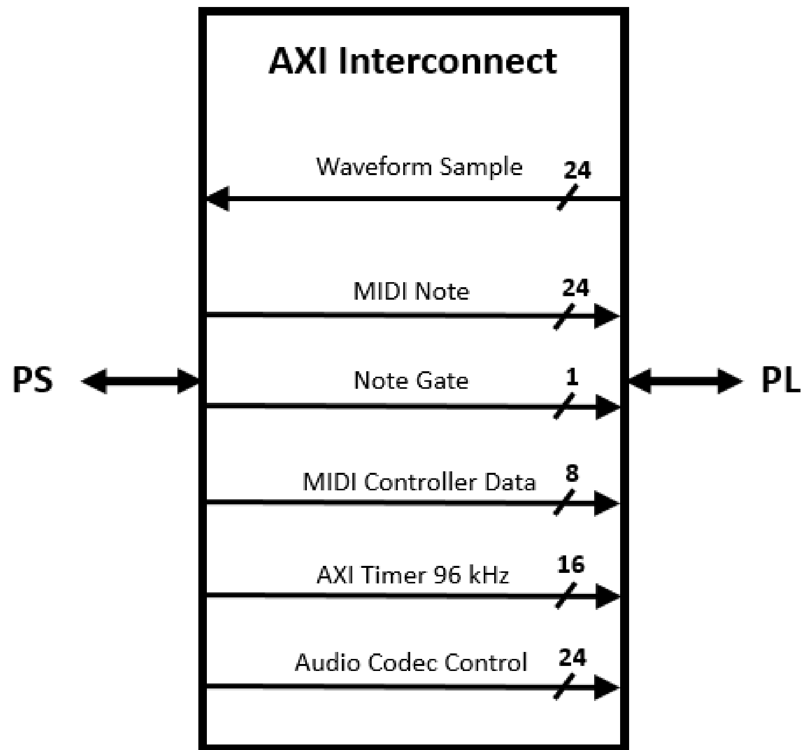


Figure 3-3: AXI Interconnect Signals

3.4 Software on the ARM Cortex-A9

The Cortex-A9 has three main functions in its role as overall synthesizer system controller. The first function is to handle audio codec initialization and create a software interface for transmission of output samples to the codec. Secondly, the PS handles UART data from the host computer containing the MIDI input data from the AKAI Mini. Controller changes and note press events are parsed, with updates to these values stored in buffers in microprocessor data memory. The routine also checks for changes in the controller values and formats the values for transmission to the FPGA over AXI. The third main responsibility of Programmable Software is the implementation of delay effect functions. This is accomplished by storing and manipulating values as each new sample is read into the microprocessor, prior to codec output. The FPU on the Cortex-A9 is utilized to scale output amplitude according to the

velocity at which the user pressed the current note with the precision of floating-point multiplication. Figure 3-4 provides a flowchart describing the three main tasks performed in software.

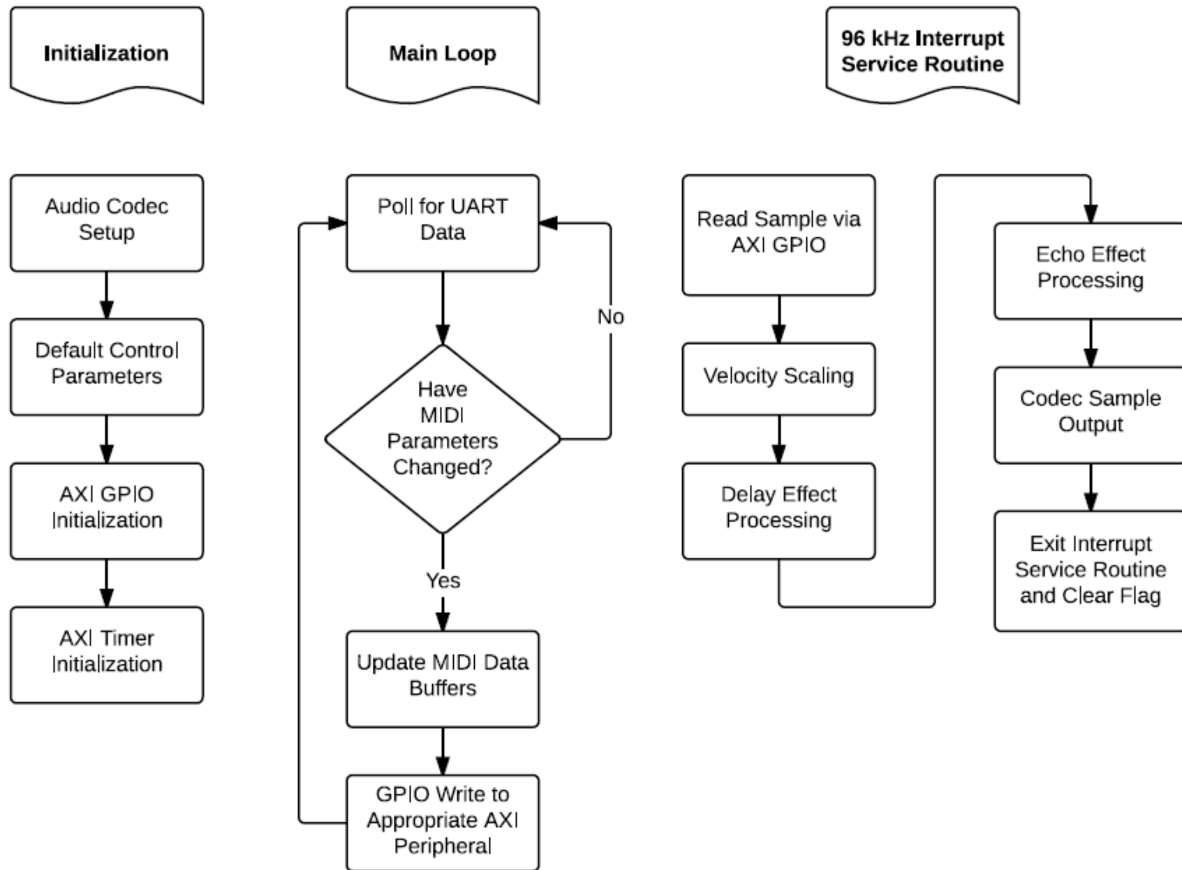


Figure 3-4: Control Software Flowchart

3.5 IP Module Generation for FPGA Processing

Each stage of effects processing was designed as a standalone module using the Verilog HDL. Xilinx Vivado Design Suite, which is used to create the top-level hardware design of the system, allows for easy integration of lower-level system parts using a GUI for connecting IP modules in a system block diagram. In order to integrate effects processing modules into the design, every module was created as a standalone project, which was published as a Xilinx IP block and imported into the top-level system design. In this manner, modules could be easily interchanged and version-controlled as the project progressed. This also facilitated module

testing, as each element of the system was a standalone digital circuit that could be tested individually or disconnected and bypassed in the top-level design.

3.6 MIDI Data Transmission to FPGA

MIDI data transmission is handled by a host computer running Windows. The MIDI controller is connected to the computer so data can be received as new notes are being pressed. A Python script was developed to handle reading data from the MIDI controller and sending it to the Zedboard via UART serial connection. The PyGame MIDI library was used to poll serial data from the MIDI Controller. As the three-byte sets of data all read in, the script sends each byte separately over to the Zedboard.

The Akai Mini only features eight controller knobs for MIDI controller data transmission to the synthesizer. Data routed through the Python-powered serial communication script was altered before transmission to the synthesizer system to multiplex these controllers. Using this method, the current controller number can be changed for any given controller knob. We accomplished this through the development of a GUI that allows the user to select different groups of functions to be controlled by the limited USB input device. As new effects were added to the system, this interface was expanded to allow for complete customization of effects controls.

Controllers are labeled individually to show the effect that each knob is currently controlling. The GUI also allows for selection of the base waveforms for LFO signals. This user interface is run independently of the MIDI serial data transmission script, through the use of multithreading application libraries built into Python. Figure 3-5 shows a screen image of the GUI we developed for the project.

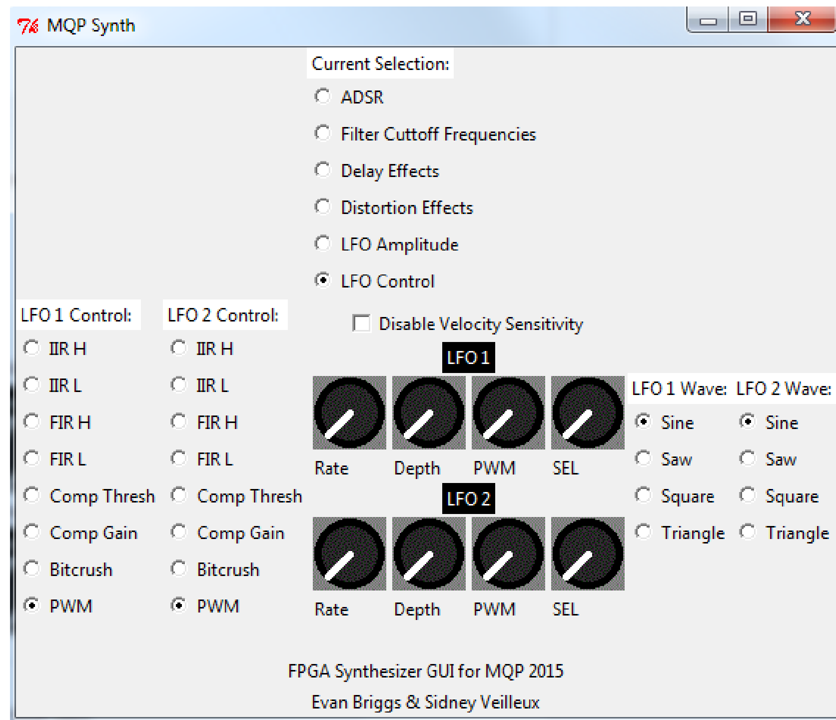


Figure 3-5: GUI for MIDI Data Program

For every new note that is pressed, the MIDI velocity parameter reflects how hard the key was struck. This gives the system a parameter that allows for the scaling of the output waveform to model the volume response of a keyed instrument when it is being played. The 7-bit velocity parameter is mapped to a floating point value that is then used to scale audio samples before sending them to the codec. We chose to use an exponential mapping of scale values between 0 and 1 to better represent the nonlinear relationship between force applied and output amplitude typical of pianos and other keyed instruments. This mapping scheme provides the user with more precision when applying softer key presses. This relationship is defined mathematically in Equation 3-1, in which x represents the MIDI velocity and α is a scalar used to modify the sharpness of the exponential curve.

$$y = \frac{e^{\left(\frac{x * \propto}{127}\right)} - 1}{e^{\propto} - 1} \qquad (3\text{-}1)$$

For our system, α was set to 2.5, a value which meets our requirement of 75% of the velocity values below the half-amplitude mapped value of 0.5. Figure 3-6 graphically depicts the relationship between note velocity and output amplitude on the synthesizer.

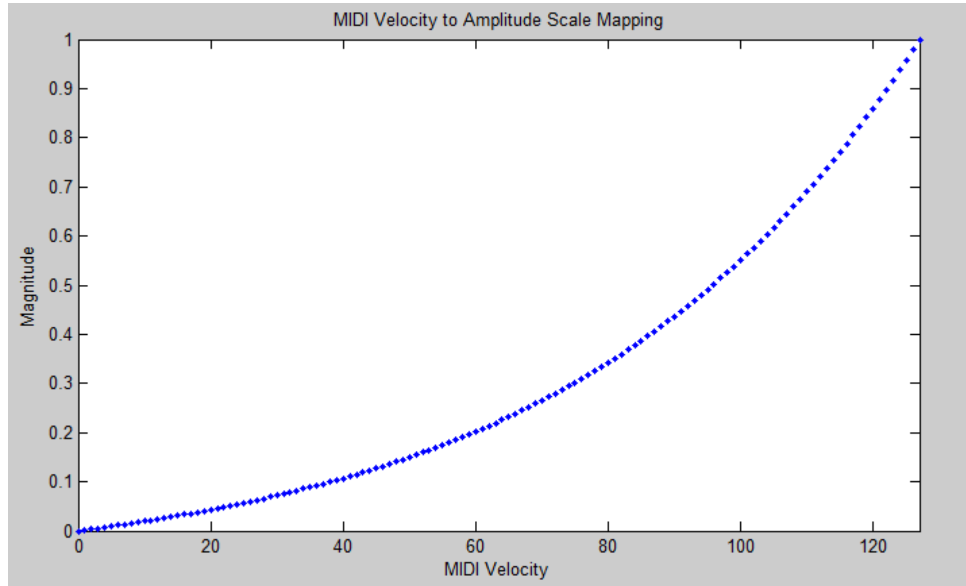


Figure 3-6: MIDI Velocity to Amplitude Scale Mapping

3.7 Clock Domains

The system made use of two clock domains for timing of programmable logic state elements. A 100 MHz clock was used as the main clock domain for all effects processing and much of the waveform generation hardware. This clock speed was chosen due to its frequency being sufficiently high to allow for all effects processing to be performed serially within waveform samples, thus meeting the development goals of a real-time system. This clock frequency also simplifies overall timing of the design, as the AXI interconnect operates on the same 100 MHz clock domain as well. A 200 MHz clock was derived from the same Phase-Locked Loop (PLL) module located on the Cortex-A9 microprocessor. This clock was used to allow for increased precision of wavetable stepping and signal generation, with successive samples synchronized to the 100 MHz clock for synchronous operation with the rest of the system.

A 96 kHz interrupt signal is generated by an AXI timer peripheral, which is used to signal the programmable software ISR to output the next sample to the codec. This interrupt is also

used in hardware to signal the waveform generation block, requesting the next sample for output. The signal then propagates through each effects processing block, allowing each module to alert the next effect that the current sample output is ready for sequential processing.

3.8 Waveform Generation

The generation of base waveforms is performed entirely in hardware on the programmable logic component of the Zynq. MIDI note data is transmitted to hardware from the system control block when a new note is pressed by the user. Each MIDI note corresponds to a discrete frequency from 8 Hz (note zero) to 12.5 kHz (note 127). The Akai MIDI controller chosen for system input is capable of producing MIDI note values from 0 to 120 (8.37 kHz). Equation 3-2 defines the conversion from MIDI note value to its respective frequency in Hz [16].

$$f_{note} = \frac{440}{32} * 2^{\frac{x-9}{12}} \qquad (3\text{-}2)$$

Using DDS for waveform generation, wavetables containing predetermined values for one period of each base waveform were created using MATLAB and implemented with LUTs on programmable logic. For this project, wavetables of 4096 samples were created to achieve high resolution waveforms, especially at low frequencies. The waveform generation module utilizes a 200MHz clock from the Cortex-A9 Phase-Locked Loop (PLL) to achieve accurate clock divisions. Slower clocks at discrete frequencies are derived within the module to achieve the correct wavetable step rate for MIDI notes. This allows the system to step through the wavetable at variable rates, producing the correct frequency response of the current input note. Equation 3-3 outlines how many clock cycles the waveform generator should wait before stepping to the next sample for a given frequency.

$$divisions = round\left(\frac{F_{clk}}{N_{samples} * F_{Desired}}\right) \qquad (3\text{-}3)$$

The frequencies of the waveforms generated by the DDS module are defined by an 8-bit MIDI note, which is routed to the NCO clock division module. This module makes use of the logarithmic property of music notes to produce only twelve discrete clock divisions while still

producing all 128 MIDI notes. While the chromatic scale contains twelve notes, each subsequent set of twelve (octave) is double the frequency of the previous set. The DDS module features an input for the step size of the waveform being generated. Keeping the clock division constant and doubling the size of the incremental wavetable step doubles the frequency of the output waveform. By doubling this step size for each musical octave, the system is able to produce notes of any MIDI frequency while preserving the accuracy of notes at the higher range of the instrument. Using Equation 3-3 to divide a 200MHz clock for the production of notes in higher MIDI octaves (above note 84) would have resulted in an increased frequency error due to the rounding of dividends associated with dividing a clock by a whole number. This issue is avoided by dividing the 200MHz input clock to very low frequencies (notes 0 through 11) and using the step size method. Figure 3-7 shows a block diagram of the hardware modules used to implement DDS using this technique.

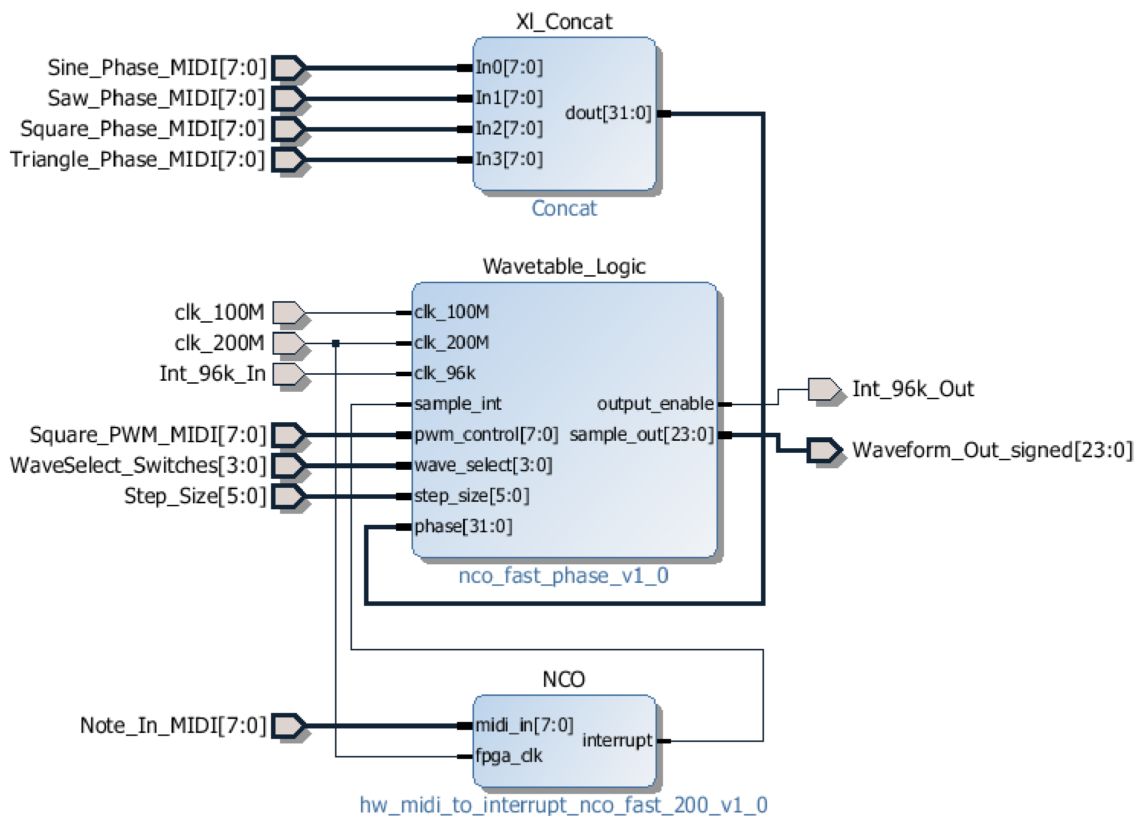


Figure 3-7: DDS Implementation Block Diagram

Four different base signals can be generated by the DDS module. These include sinusoid, sawtooth, square, and triangle waveforms. Multiple signals can be selected simultaneously, and the module produces a signal containing each of the base waveforms selected by the user, combined in a weighted addition to prevent value overflow. This allows for any combination of the four base waveforms to be produced, resulting in 15 different synthesizer signals able to be produced before any filtering or effects processing is applied.

3.8.1 Sine Wave

The sine wave is the most fundamental of the waveforms, as it is made up of only one harmonic component in the frequency spectrum. Due to this harmonic singularity, the timbre of this waveform is very hollow compared to other more complex signals [17]. Sinusoids are particularly useful in creating sub-octave oscillators for a more bass-driven sound. Other complex waveforms can be recreated through the combination of sine waves at different frequencies and amplitudes to recreate the frequency composition of a tone. Figure 3-8 shows the 4096-sample wavetable of the sine wave used in the DDS block, with a maximum amplitude of 4.19e5, or 2^23.

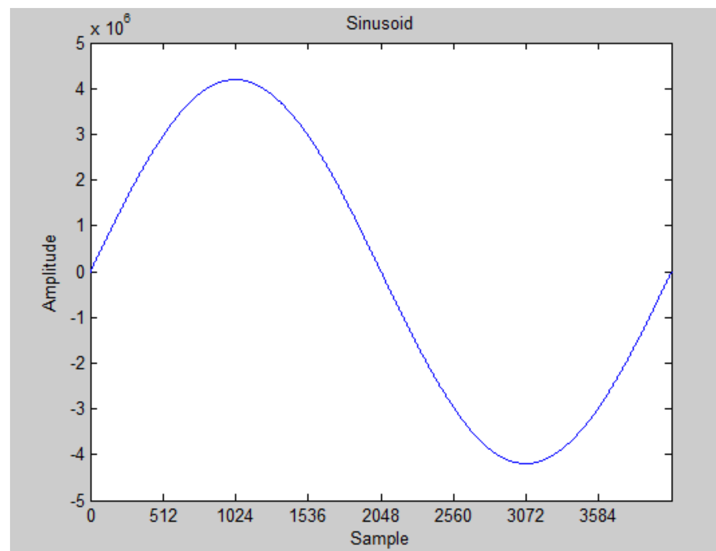


Figure 3-8: Sine Wave Sample Table

3.8.2 Sawtooth Wave

The Sawtooth is a common waveform featured in many synthesizers. This signal produces a much richer tone than others, as it is made up of many sinusoids across the frequency spectrum. In the time domain, a sawtooth is simply a repeating ramp function, producing sharp transitions from the maximum to minimum peak once per period. This sharp transition creates the "blades" that give the waveform its name. The signal is composed of both even and odd harmonics of the fundamental frequency, giving it a 'full' sound. The MATLAB plot for this waveform can be found in Figure 3-9. The harsh sound generated by the sawtooth is similar to that of a trumpet, an instrument which also produces many harmonics of the fundamental frequency being played. A mathematical representation of the Fourier Series summation of each harmonic sinusoidal component that defines a sawtooth waveform's distribution across the frequency spectrum can be seen in Equation 3-4 [18].

$$x(t) = \frac{A}{2} - \frac{A}{\pi}\sum_{i=1}^{\infty}\frac{sin(2\pi f i t)}{i} \quad (3\text{-}4)$$

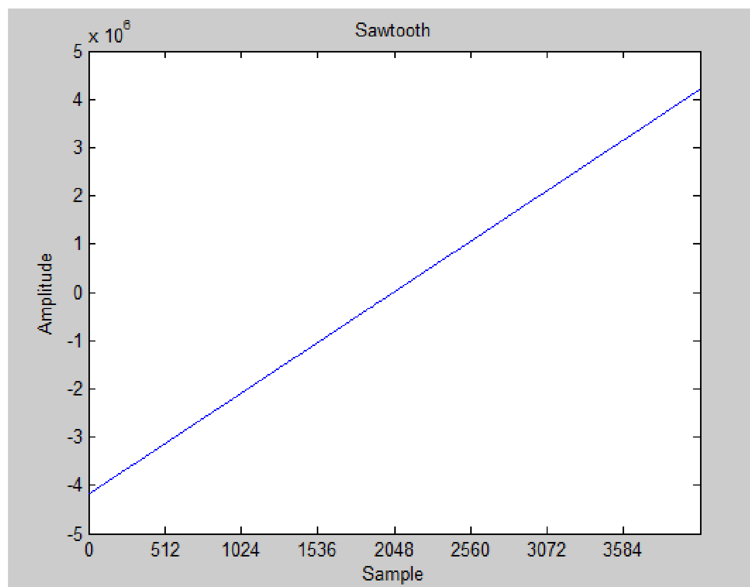


Figure 3-9: Sawtooth Wave Sample Table

3.8.3 Square Wave

The square wave is an uncomplicated waveform in the time domain, as it simply switches between its maximum and minimum peaks with near infinite slope. In the frequency domain, the signal is much more complex, containing only odd harmonics of the fundamental frequency, with the subsequent higher frequencies of decreasing amplitude. This creates a sound similar to the sawtooth waveform, but is much hollower in timbre. This makes the square wave an excellent candidate for mimicking wind instruments using subtractive synthesis, as these instruments produce a similar hollow tone. The square wave was implemented with a modular duty-cycle, which allows for different timbres to be generated from the same type of waveform. The wavetable for one period of the square wave (at 50% duty cycle) can be found in Figure 3-10. Equation 3-5 provides the Fourier Series representation of a square wave, containing the summation of sinusoidal terms that make up each of its harmonics [19]. The process of subtractive synthesis acts to attenuate components at the beginning and/or end of this summation.

$$x(t) = \frac{4}{\pi}\sum_{i=1}^{\infty}\frac{sin(2\pi f t(2i-1))}{2i-1} \tag{3-5}$$

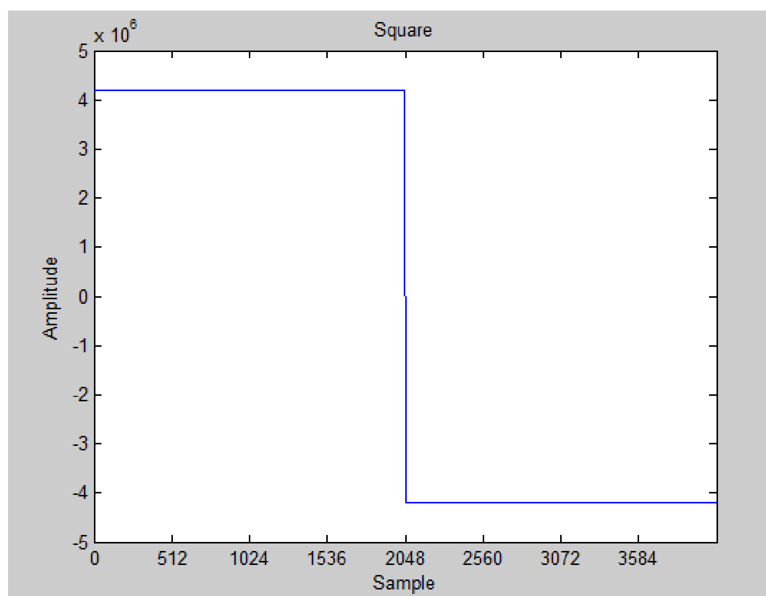


Figure 3-10: Square Wave Sample Table

3.8.4 Triangle Wave

The final waveform our system is capable of generating is the triangle wave. This signal is similar to the sawtooth in the time domain, consisting of two ramp functions of inverted slopes. In the frequency domain, it is made up of odd harmonics, with a much steeper roll-off in amplitude of high harmonics than the square wave. Additionally, every 4th harmonic of the triangle wave is 180 degrees out of phase with the other present sinusoids. This produces a sound similar to the square wave, but much richer in timbre. The wavetable for the triangle wave can be seen in Figure 3-11. The Fourier Series representation containing the sinusoidal terms that make up each of the harmonic components of a triangle wave can be seen in Equation 3-6 [20].

$$x(t) = \frac{8}{\pi^2} \sum_{i=1}^{\infty} (-1)^i \frac{sin(2\pi f t(2i+1))}{2i+1} \qquad (3\text{-}6)$$

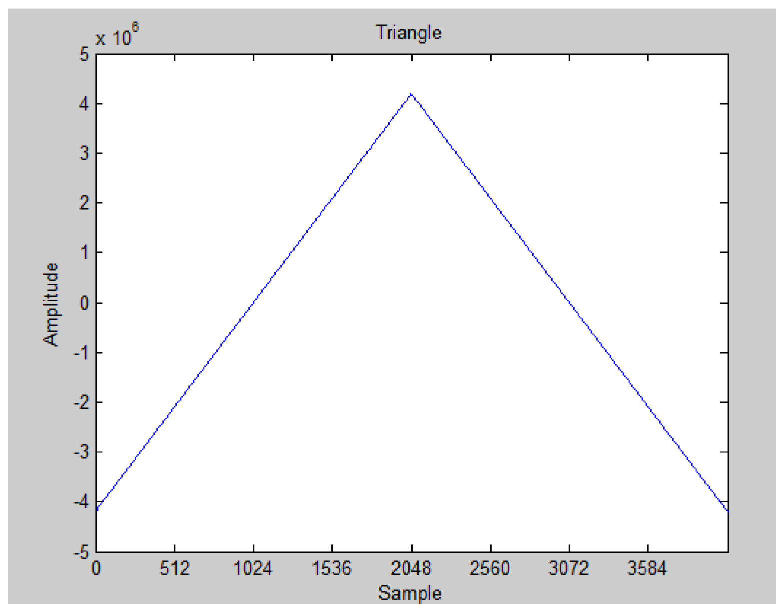


Figure 3-11: Triangle Wave Sample Table

3.8.5 Complex Waveform Generation

Other input parameters used by the DDS block include phase shift and pulse-width modulation (PWM) controls. The phase shift control is in the form of four MIDI controller values, each of which controls the independent phase angle of the four base waveforms. This allows for many different waveform shapes to be created by the user, simply by shifting the

base signals in and out of phase. This is implemented in the DDS module by scaling the MIDI inputs by a factor of 32 and adding them to the current wavetable index for their respective base signals. In this manner, the phase shift is linearly mapped to nearly the entire 4096-sample period of the waveform.

The user is able to select any or all of the four base waveforms to be active concurrently. This, when combined with independent phase-shift parameterization, allows for the production of a wide range of complex base waveforms. As an example, Figure 3-12 shows a 440Hz sinusoid, shifted 90 degrees in phase, and a sawtooth at the same frequency with no phase shift. The resulting combination of these two signals, called a sine-saw, is visible in the lower plot of the figure. This complex waveform has a musical timbre that is different from the original signals.

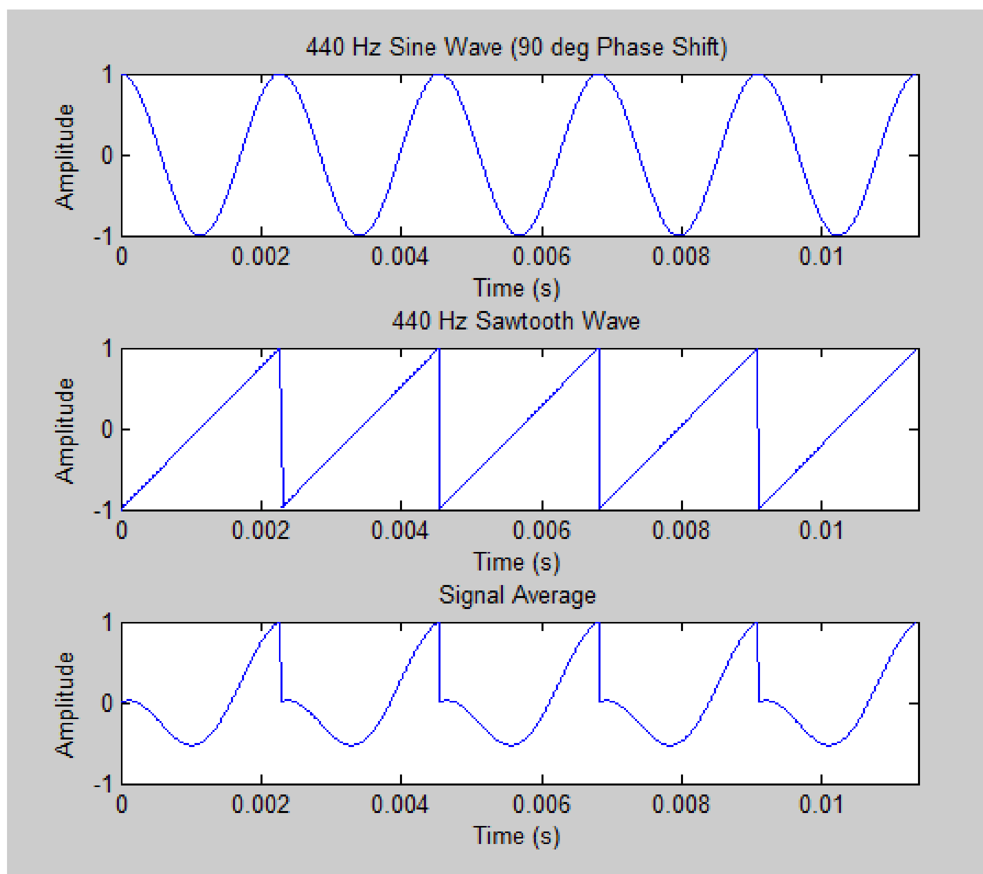


Figure 3-12: Sine-Saw Combination in MATLAB

The PWM input allows the user to control the duty cycle of the square wave. This alters the shape of the waveform, effectively changing its tone. Within the DDS module, the PWM input is used to simply change the wavetable index at which the signal switches from high to low using combinational logic. Similar to the phase shift implementation, the value is scaled by a factor of 32 to allow for a fully-customizable duty cycle from 0% to 100%.

3.9 Synthesizer Effects

The signal generated by the combination of base waveforms is routed through four stages of filtering in order to perform subtractive synthesis. The signal also passes through several synthesizer effects stages connected in series before being sent to the audio codec for system output. These effects include a frequency modulation stage, which creates musical vibrato, and various distortive effects to introduce additional harmonics and alter the waveform. An amplitude modulation envelope scales the volume of the signal to model the amplitude response of various physical instruments. Finally, delay and echo effects are implemented in software as the last stage before signal output. Synthesizer effects implemented on the system are discussed in more detail in Chapter Four. A top-level diagram of these effects and their order can be seen in Figure 3-13.

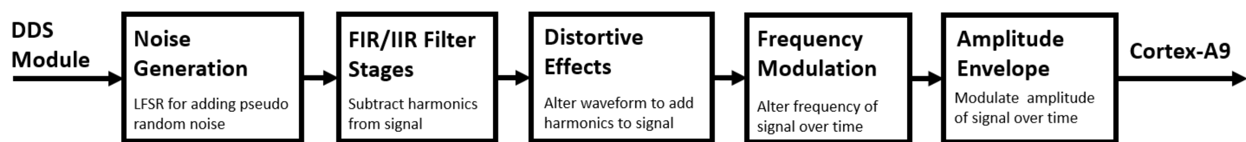


Figure 3-13: Synthesizer Effects

3.10 Audio Codec for System Output

The ADAU 1761 supplied on the Zedboard is capable of using either 16 or 24-bit data at standard data rates of 44.1 kHz, 48 kHz, and 96 kHz. For this project, the codec was initialized to use 24-bit data at a rate of 96 kHz to achieve the highest possible resolution on the output of the system. The codec receives configuration signals via I2C protocol and features a separate I2S connection to handle the transmission of audio signal data to and from the processor.

Libraries provided by The Zynq Book Tutorial were used to aid in configuration of the registers and initialization of the PLL on the ADAU1761 [21]. An AXI peripheral controlled by software was utilized to allow for Integrated Interchip Sound (I2S) serial data transmission to the codec on the Zedboard. The I2S AXI Peripheral resides in the programmable logic and receives sample data from the Cortex-A9 at a rate of 96 kHz. This module sends the 24-bit audio sample serially to the DAC of the audio codec for analog output. Figure 3-14 provides a block diagram of the audio codec data transmission between the Zynq components and the codec.

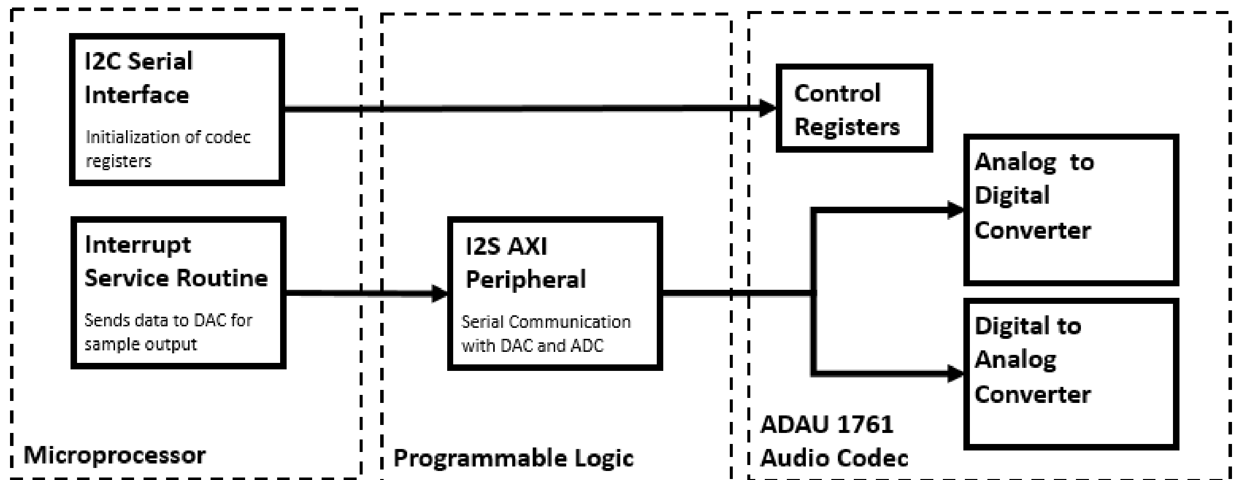


Figure 3-14: Audio Codec Implementation

The following chapter provides a detailed description of synthesizer effects implemented on our system. This includes the operational theory behind each effects processing stage, as well as programmable logic implementation of each synthesizer component module.

4 Synthesizer Effects Implementation

This section describes each of the synthesizer effects implemented on the system. These effects are arranged serially, with the first effect connected to a 96 kHz pulse signal that signifies a new sample has been generated by the DDS block and is ready for processing. The pulse propagates sequentially through each module, such that each effects block can alert the next module that its current output sample is ready for processing. This ensures that effects processing occurs in order, with the most recently-generated sample propagating through the entire system. Figure 4-1 graphically depicts the order of operations of each effect module implemented on the system.

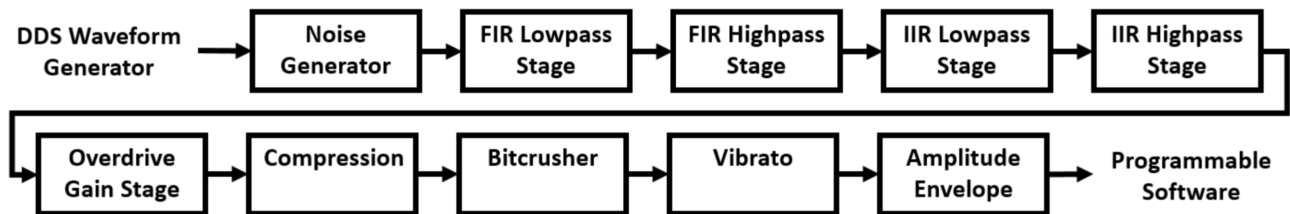


Figure 4-1: Synthesizer Effects Order

4.1 ADSR Amplitude Envelope Generator

The ADSR (Attack-Decay-Sustain-Release) Amplitude Modulation Envelope is an essential component of any modern synthesizer. This time-based envelope is used to alter the amplitude of a signal over the duration at which a selected note is being held down. This is used in practice to model the physical response of an instrument, and its change in volume over time. For example, the pluck of a guitar string can be represented by a sharp increase in amplitude followed by a long, drawn out attenuation as the vibrating string loses energy. An ADSR envelope is comprised of four stages which give it its name, outlined in Figure 4-2. Attack, the first stage, begins the moment a note is pressed. During this phase, the amplitude of the signal rises from its idle state at zero to a maximum value of one. Once the maximum value is reached, the envelope enters the decay stage, during which the amplitude falls to a user-defined level, called sustain. The envelope remains in the sustain stage with constant amplitude until the user stops playing the note. At this point, the release stage begins, during which the

amplitude decays back to zero. The shape of the envelope is chosen by four parameters, three of which control the rates of ascent and descent for attack, decay, and release stages. The fourth parameter allows for control of the constant sustain level.

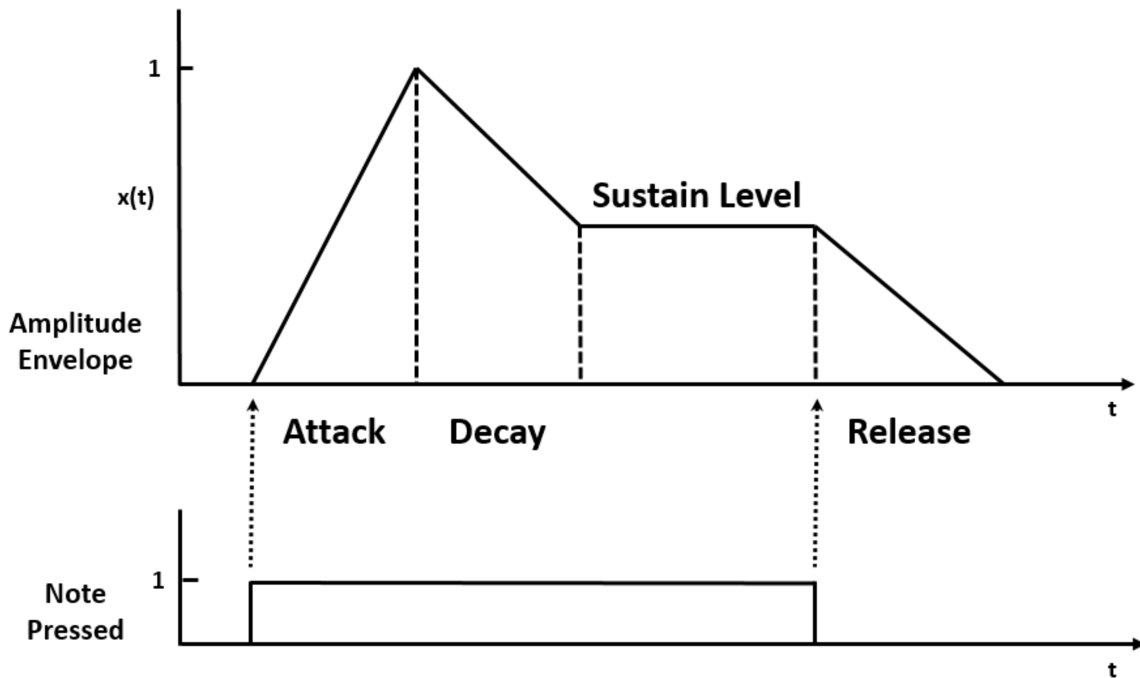


Figure 4-2: ADSR Envelope Diagram

A properly implemented ADSR envelope allows for the development of a synthesizer that is able to model a wide variety of musical instruments. For example, an envelope used to model the physical response of a piano would have a very short attack stage, and a slightly longer decay. This would be followed by a relatively high sustain as the note is being held down, and a very short release, due to damping felt quickly absorbing vibrations in the instrument's strings.

The ADSR envelope generator was designed as a standalone Verilog module which was published into a Xilinx IP block to be incorporated into the top-level design. All processing for this effect is performed using fixed-point multiplication on the programmable logic hardware. The envelope module generates a 16-bit signal in Q-15 fixed-point number format. This unsigned value is used to represent a scale factor between zero and one. The value is then multiplied by the synthesized waveform signal using a Xilinx Multiplier Block, effectively

modulating its amplitude over time. Figure 4-3 shows the top-level design of this effect module, including MIDI control logic inputs and integration with the Multiplier Block to produce an output signal for processing by the next stage of effects.

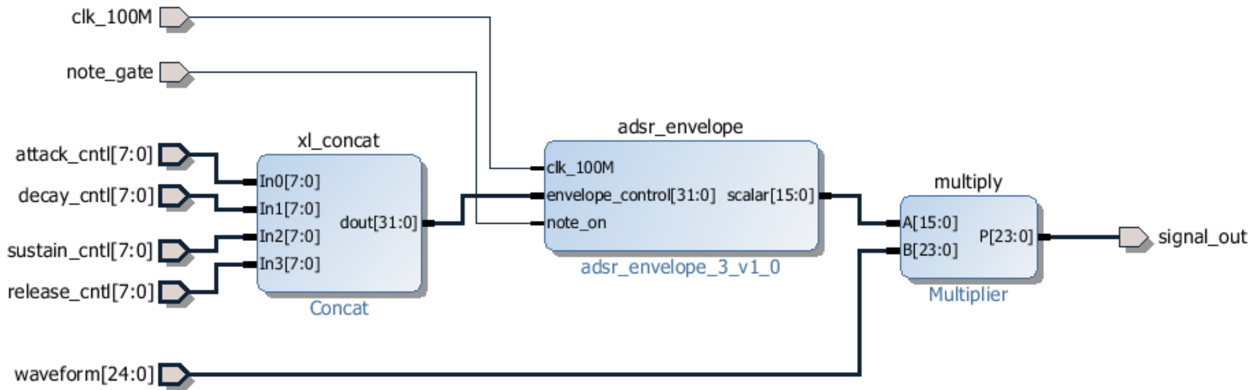


Figure 4-3: ADSR Top-Level Implementation

The module was designed as a finite state machine with each envelope stage as well as the idle stage corresponding to a different state. During each of the four active states, the scale factor is updated with 1ms resolution. This allows for smooth changes in signal amplitude while still maintaining a 16-bit representation of the scalar. State logic is controlled by the *note_on* signal, which is asserted high whenever a note is being pressed. A graphical representation of this logic can be seen in the state transition diagram in Figure 4-4.

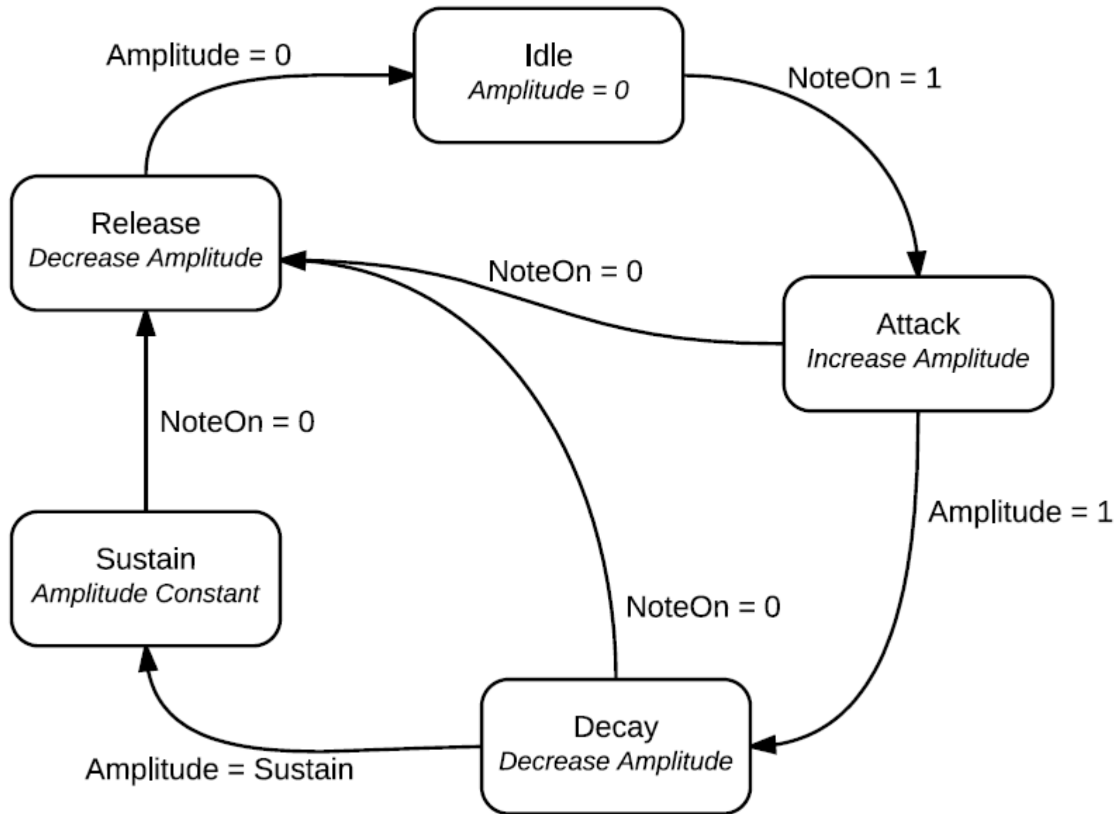


Figure 4-4: ADSR Module State Transition Diagram

Four different MIDI controller signals are mapped to control these four parameters in hardware, which can be selected under the ADSR menu in the GUI. The ADSR IP module handles data parsing of the four individual bytes that make up the ADSR parameter data. Each of the parameters that control the four stages can be modified independently, allowing for the user to program a specific amplitude response. This allows for a wide range of possible envelope shapes, which can be used to model the physical response of a variety of instruments. Figure 4-5 shows an example of the ADSR envelope, simulated in MATLAB. In this example, the note is first pressed at time t=0, and released at t=0.8 seconds. Accordingly, the amplitude of the output signal is modulated over the course of the four envelope stages, separated by dashed lines in the center graph.

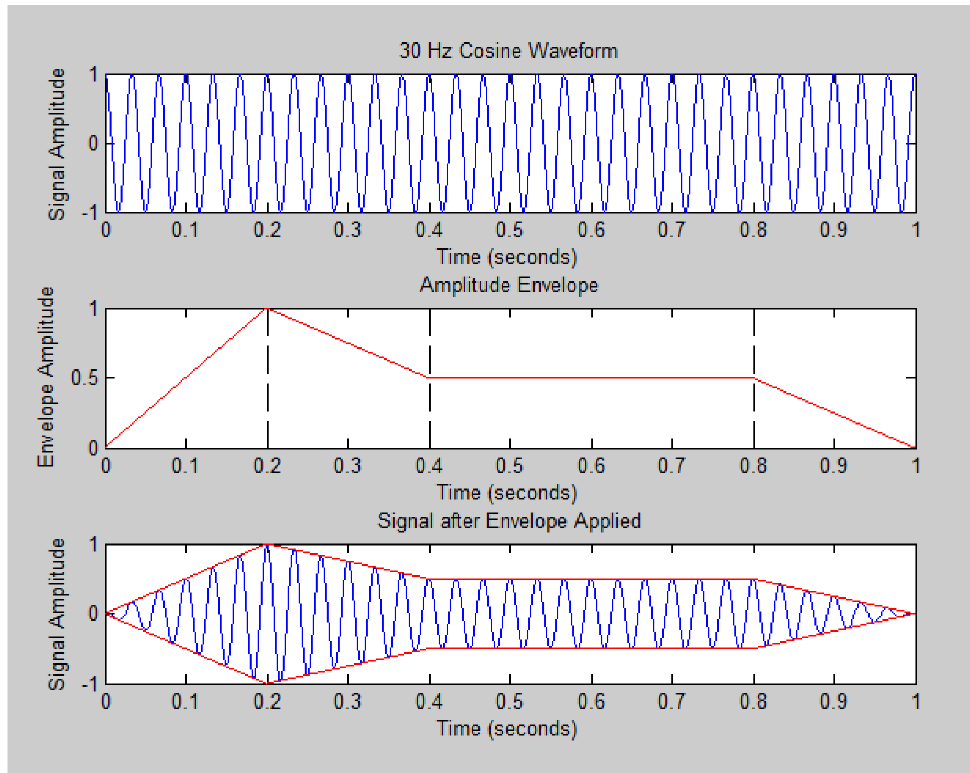


Figure 4-5 - ADSR Amplitude Envelope Model (A,D,R = .2 seconds, Sustain level = .5)

4.2 Filtering Stages

In order to perform subtractive synthesis, it is necessary to include filter stages in the synthesizer design to attenuate certain portions of the frequency spectrum, thus subtracting harmonic components of the base waveforms. When applied to a simple sinusoid, filters merely reduce the amplitude of the entire waveform, as the signal only exists at one discrete location on the frequency spectrum. However, filters can produce much more interesting effects when applied to complex signals. This is true for sawtooth, square, and triangle waveforms, as these signals are made up of many harmonics of their fundamental frequency, occupying a much larger bandwidth. When applied to these complex waveforms, filter stages act to attenuate each of the harmonics differently, which can dramatically alter the shape of the waveform.

We were able to develop a Finite Impulse Response (FIR) filter module as well as an Infinite Impulse Response (IIR) module. Each module was duplicated to create both Low-pass and High-pass filters, with variable discrete cutoff frequencies between 0 and 48 kHz. These filters were implemented in serial, resulting in the base waveforms passing through a total of

four filtering stages before continuing on to other effects. The filters were designed using the Filter Design and Analysis tool in MATLAB, and MATLAB scripts were written to automate the production of Verilog files describing their hardware implementation. These filters made use of Xilinx Multiply-Accumulate blocks, which supplied the fixed-point multiplication and addition hardware necessary for signal filtering in digital logic.

An FIR filter is implemented using a weighted sum of its past input values. The implemented FIR modules made use of a 65-point weighted sum, which is defined mathematically as a 64th-order filter. Use of a higher order filter results in a sharper pass- to stop-band transition at the expense of longer computation time, due to the increased number of required Multiply-Accumulates (MACs). One important characteristic of FIR filters is that their phase response across the frequency spectrum is generally linear in slope. Due to this effect, the input signal is filtered in amplitude without introducing any of the distortion brought about by nonlinear phase shifting. Equation 4-1 shows the mathematical definition of the Direct Form Implementation for an Nth-order FIR filter, in which the vector b represents the filter coefficients, and the vector x represents the previous N inputs to the system. This implementation served as the basis for this module's logic design.

$$y[n] = \sum_{i=0}^{N} b_i * x[n-i] \qquad (4\text{-}1)$$

An IIR filter is similar in implementation to the FIR filter, with the addition n of a weighted sum of previous system output values. This feedback component results in much sharper frequency transitions at lower order filters, which can greatly save computation time in comparison to FIR filters. Due to this property of the IIR filter, the modules were implemented as 4th order filters. A common side-effect of the IIR filter is a nonlinear phase response in the frequency domain. This results in a filtered signal containing some frequencies shifted in phase. This is often undesired in practice, as it merely adds distortion to the waveform. However, in the application of digital music synthesis, phase distortion can add extra flair to the sound being created by the user, and is considered a useful effect. Equation 4-2 shows the mathematical definition of the Direct Form implementation of an IIR filter.

$$y[n] = \sum_{i=0}^{N} b_i * x[n-i] + \sum_{j=1}^{N} a_j * y[n-j] \qquad (4\text{-}2)$$

The cutoff frequencies for the implemented filters were chosen taking into account the limitations of MIDI parameter data. With 128 possible values of a MIDI controller, an equal number of frequencies had to be mapped to these values, across 32 kHz of bandwidth. Figure 4-6 depicts the mapping of the MIDI values to their corresponding cutoff frequencies. The frequencies were mapped using Equation 3-1, scaling the maximum possible cutoff to 32 kHz. This logarithmic mapping allows for higher precision cutoff frequencies in the lower range of the frequency spectrum, since all 128 note frequencies as well as the range of human hearing lie below 20 kHz, which is less than half of the system's Nyquist rate. A MIDI value of 0 corresponds to a signal pass-through, in which no filter is applied to the signal to allow for bypassing of any or all filter stages.

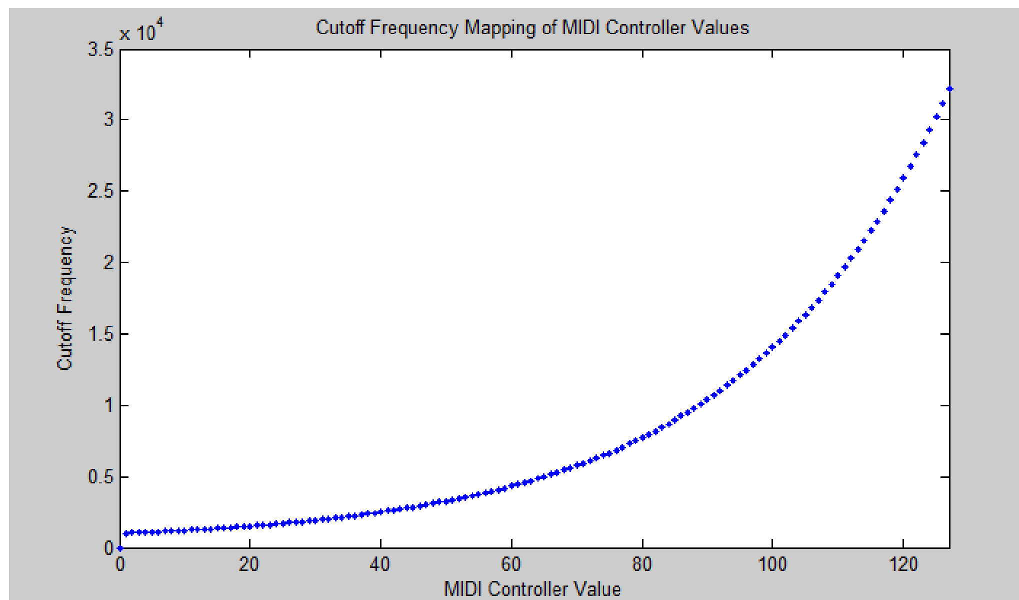


Figure 4-6: MIDI to Frequency Cutoff Mapping

For each of the four filter modules, coefficients for the desired cutoff frequencies were generated in MATLAB to achieve double precision floating point values. These values were then quantized and set to a corresponding fixed point number for hardware fixed point processing. The FIR filter coefficients were implemented at 16 bit Q-15 fixed point integers. The IIR coefficients were implemented as signed 34 bit 3.Q-30 values. The higher bit bus was required for IIR to maintain precision of low valued fractional coefficients, thus avoiding filter instability.

Verilog sub-modules were designed to read MIDI controller data representing the desired filter cutoff frequency and select the appropriate set of coefficients. These modules were produced by a MATLAB script, written to calculate sets of coefficients for each of the four filters. For the IIR filters, additional functionality was added to allow for the clearing of registers containing the previous inputs and outputs from the filter block. This is actuated by the application of a synchronous reset signal whenever the filter cutoff frequency is changed. The purpose of this is to prevent filter instability resulting from changing filter coefficients part-way through sample processing. If the switching of coefficients causes an unstable response, the error could then persist indefinitely, due to the infinite response characteristic defined in name by the IIR filter. Figure 4-7 shows the system implementation of an IIR filter with the corresponding inputs and outputs of the filter stage.

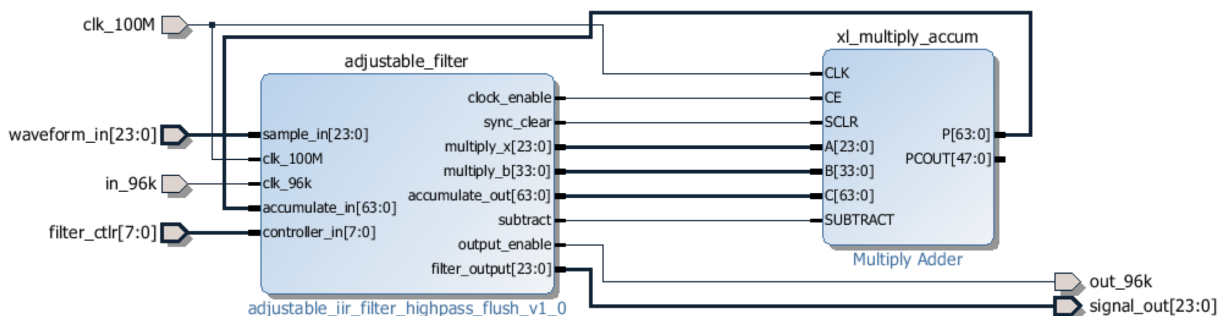


Figure 4-7: FIR Filter Block Diagram

Figure 4-8 displays the result of applying a 64th order Low-pass FIR filter module with a cutoff frequency of 10 kHz to a sawtooth waveform, generated and simulated within MATLAB. The unfiltered waveform was plotted in the frequency spectrum as well, to demonstrate the frequency response of the filter. The same sawtooth waveform, with the FIR Low-pass filter applied to the signal, can be seen in green, with the high-frequency harmonics greatly attenuated. This creates a smoother timbre of sound at the output of the system, reducing the sharpness of the blades of the sawtooth.

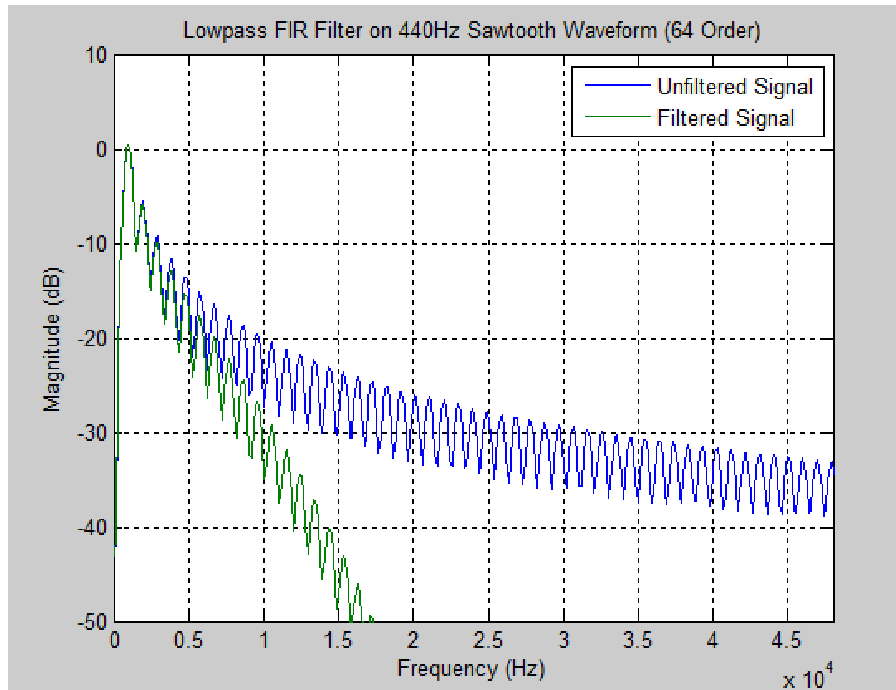


Figure 4-8: Sawtooth Waveform with and Without FIR Low-pass Filter

As a comparison of the different effects produced by FIR and IIR filters, a sawtooth waveform was generated in MATLAB and passed through two independent filter stages. Each filter was a low-pass, designed at a cutoff frequency of 1500 Hz. An unfiltered sawtooth at 440 Hz can be seen in the upper plot of Figure 4-9. The center plot shows the result of passing the waveform through the 64-order FIR filter stage. The sharp peaks of the signal have become more round, as the high harmonics of the waveform are filtered off. The lower plot in the figure shows the result of applying the 4-order IIR filter, in which nonlinear phase shifts across the frequency spectrum has caused the waveform to begin to lose its signature shape.

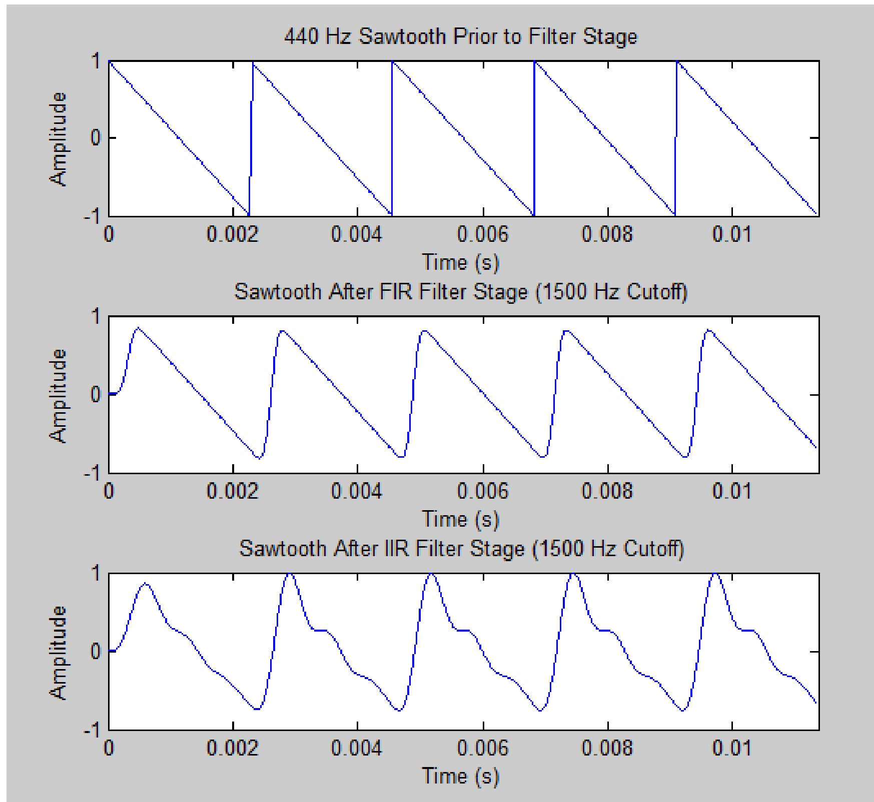


Figure 4-9: FIR/IIR Comparison: Sawtooth

4.3 Distortive Effects

Several effects that were developed and implemented on the synthesizer fall into the classification of Distortive Effects. This type of effect is categorized by its reduction of the quality of a waveform, through either the removal of components of the signal or the addition of other waveforms. These modifications to the waveform alter the harmonics present in the signal, effectively changing the tone of the note being played. Distortive effects implemented on our system include amplitude clipping, overdrive, bit reduction, and noise addition.

4.3.1 Amplitude Clipping and Overdrive

The first implemented distortive effect is amplitude clipping, or Compression, in which the top and bottom peaks of a waveform are capped at a maximum absolute value, resulting in the removal of any components of the waveform above the desired amplitude level. An ideal system is designed such that the amplitude of a waveform is never able to exceed the full-scale range of its output. However, any time a waveform is routed through an amplification stage,

the risk of signal clipping becomes a possibility, as the waveform is increased in amplitude to above the maximum output of the system. The Compression effect acts to emulate this phenomenon in a controlled environment. Figure 4-10 shows a simulation of the clipping effect in the time domain using MATLAB generated sine wave with a clipping level of half of the maximum amplitude.

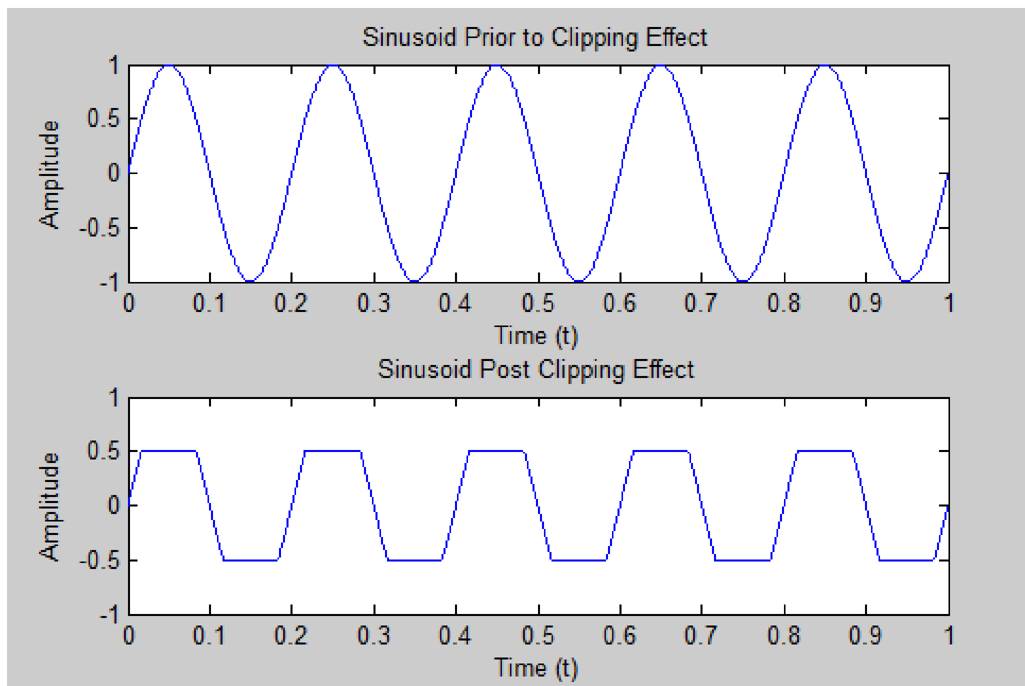


Figure 4-10: Clipping Effect

In order to model clipping in a digital system, an adjustable clipping effect module was designed. This was accomplished using a programmable logic module which simply prevented the amplitude of the input signal from surpassing the current clip threshold using combinational logic. User input is applied to control the clipping threshold, which greatly distorts the waveform when lowered. This was implemented using a linear mapping of the MIDI controller input to the level of clipping of the signal. If the signal amplitude rises above the defined threshold or below the negation of this threshold, the output of the module for that sample is simply equal to the threshold level (or its negation).

The clipping effect was modified to include a pre-amplification (signal gain) stage before amplitude clipping was performed. This increases the amplitude of the waveform, resulting in a larger portion of the signal being clipped. This type of distortive effect, which makes use of both stages, is referred to as Overdrive, and models the response of an analog guitar amplifier circuit turned past its recommended level of operation, resulting in a distortive clipping at the positive and negative power rails of the circuit. This was also implemented using linear mapping of the MIDI controller value to a 16 bit Q-15 number, which was routed into a Xilinx Multiplier Block to effectively scale the signal. The maximum effective gain was defined as 10 times the amplitude of the original signal, which allows for clipping to occur even when the compression threshold is at its maximum level.

4.3.2 Waveform Resolution Reduction

Another distortive effect is waveform resolution reduction, commonly referred to as Bitcrushing. This type of effect is implemented by simply removing the lower bits of a waveform and replacing them with zeros, thus reducing the bitwise resolution of the signal. As samples are quantized to increasingly fewer discrete values, the resulting tone becomes simpler, similar to early computer-generated sound effects. The Bitcrusher effect essentially emulates a system of lower bit-resolution, allowing a user to produce timbres associated with early digital synthesizers.

Figure 4-11 displays graphically the effects of reducing the resolution of a 24-bit waveform. The upper plot shows a 440 Hz pure sinusoid at 24 bits of resolution. The center graph is the result of applying the Bitcrusher effect in MATLAB with reduction to 3 bits of resolution (and one sign bit). Increasing the level of reduction to two bits results in the entire range of the sinusoid quantized to eight discrete values, resulting in a much hollower sound.

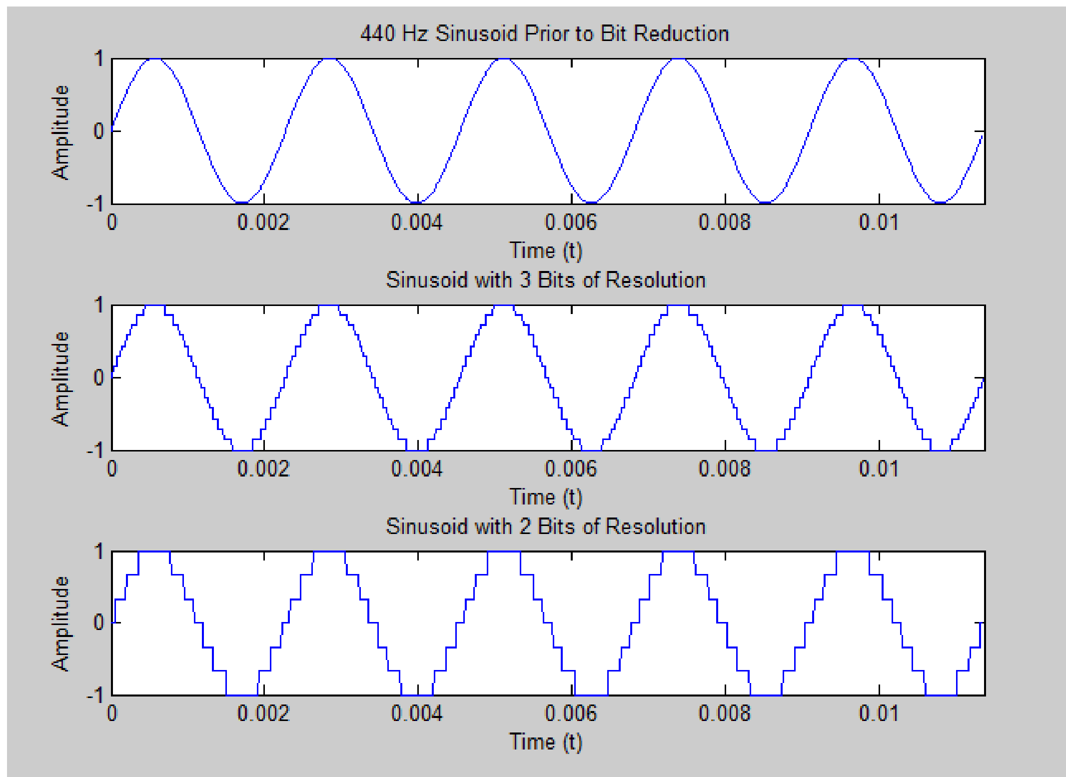


Figure 4-11: Bitcrusher Effect

The waveform resolution reduction module was developed in Verilog for implementation in programmable logic. Input to the system allows for a user-controlled level of waveform reduction, from full-scale resolution down to just the upper two most significant bits of the signal (for up to 21 bits of reduction). This module implements linear mapping of the MIDI controller input value to the number of bits to be truncated. The lower "crushed" bits in each sample are essentially bitwise ANDed with zero to remove the appropriate portion of data from the signal.

4.3.3 Noise Generation

The final distortive effect implemented in the system was a noise generator. This effect models the added noise that is often observed at the outputs of older analog synthesizers and amplifiers. White noise was often present in these systems due to the use of vacuum tubes, which often degrade over time, allowing for an increased thermal flow of electrons. Many modern synthesizers produce very little uncontrollable noise, as semiconductor transistors are much more effective for the control of electric current. In order to introduce this noise

component back into modern systems, a standalone noise production module must be designed.

A linear-feedback shift register (LFSR) is used to produce a pseudorandom 24-bit signal, the amplitude of which is user-controllable through scalar multiplication [22]. Figure 4-12 shows the frequency response of the noise generator, as the quality of pure noise is defined by how evenly distributed its power magnitude appears across all relevant frequency bands. This LFSR uses three XOR feedback taps and three XOR feed-forward taps to induce an even distribution of noise across the frequency domain. An LFSR with only feedback taps generally features a less even distribution of frequencies present in the noise generator signal, while using both feedback and feed-forward components increases the overall randomness of the signal.

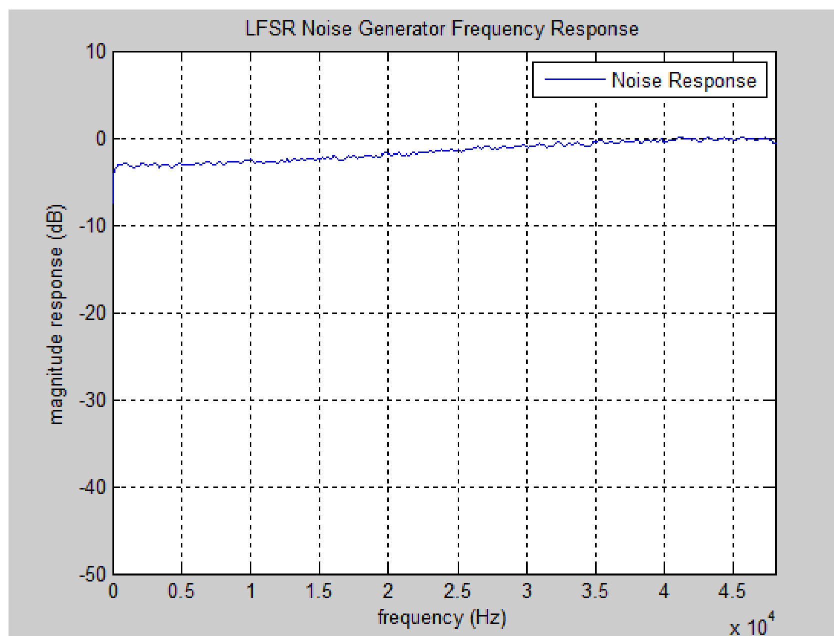


Figure 4-12: LFSR Noise Generator Frequency Response

The LFSR noise signal is scaled through the use of fixed point multiplication. A MIDI controller is routed through a linear mapping module to scale the 8-bit controller value up to a 16-bit unsigned integer. This scalar and the noise signal are used as inputs to a multiplication block to produce a scaled version of the noise signal. This allows the user to control how much

noise is to be applied to the synthesizer signal. The scaled noise is then fed back into the IP module to perform signal addition with the base synthesizer waveform. Figure 4-13 shows the block diagram of the noise effect as it is implemented on our system and connected to a Multiplier Block.

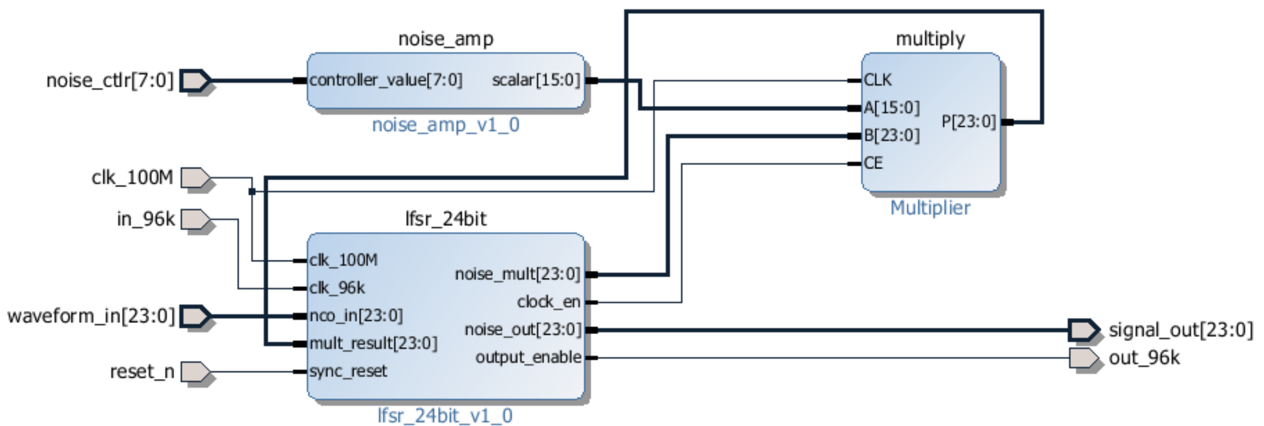


Figure 4-13: LFSR Block Diagram

4.4 Delay Effects

Several common synthesizer effects make use of delay modules, which store previous values of a signal for a variable length of time, usually for up to several seconds. This allows for simple delay between user input and system output, as well as more complex effects that create echoes and reverberations. Echo effects make use of a delay module to store previous echo values, which are combined with the current sample in weighted addition. This signal mixing is typically adjustable, allowing for user-defined echo amplitude and rate of decay. These variable parameters result in a customizable echo effect that can be used to model echo chambers of arbitrary size and level of noise dampening.

Two types of delay effects were implemented on the synthesizer. Unlike many of the other effects, these were implemented in software on the Cortex-A9 microprocessor.

Implementation of a delay module requires storing many previous samples in order to recall them later to be added back into the outgoing signal. For these implementations, two seconds (192,000 samples) of previous outputs were required to be stored for each effect. This is possible to accomplish in hardware with a Direct Memory Access (DMA) block to a large source of memory, but is more efficient to implement in software, as samples pass through the microprocessor before codec output. The Cortex-A9 has access to 512MB of Random Access Memory (RAM), allowing for large arrays of samples to be easily stored for quick access. Between each codec output interrupt, which occurs at a rate of 96kHz, there is sufficient time to process the delay effects as well as the MIDI controller data coming into the system.

The first effect is a basic delay function, implemented using a 32x192000-bit circular array. Each new incoming sample from the FPGA hardware is stored, and the array index is updated to the newest position. The effect simply delays the signal by a certain number of samples, defined by 8-bit user input. The user is then able to select how much of the delayed signal is present in the outgoing signal by changing the 'Dry/Wet' input. This allows for a weighted addition of the two samples with a full range of selection, in which the user can feed the original signal (100% Dry), the delayed signal (100% Wet), or an arbitrary combination of the two.

The second delay-based effect is Echo, with acts to emulate the physical phenomenon of an echo. A second array that is equal in size to the original delay effect array is utilized to store previous echo output values, resulting in decaying signal feedback. This feedback effect creates a digital echo, similar to the physical response produced in an echo chamber. Equations 4-3 and 4-4 mathematically define the theory of operation for echo effect as it implemented in the system. The current output sample, denoted at $y[n]$, is dependent upon the input sample $x[n]$ and the echo sample $e[m]$ stored from previous echo iterations. The amount that the echo sample influences the output is the feedback coefficient α. The new echo sample to be stored is equal to the output sample multiplied by the decay constant γ. These two mapped parameters allow for numerous combinations to alter the echo response of the system.

$$y[n] = x[n] + e[m] * \propto \qquad (4\text{-}3)$$

$$e[m] = y[n] * \gamma \qquad (4\text{-}4)$$

This echo effect implementation was simulated in MATLAB prior to system development in software. A digital echo with the decay constant set to 0.5 was applied to a short pulse of a 440 Hz sine wave. The resulting series of decaying pulses can be seen in Figure 4-14, in which the amplitude of each echoed pulse is half that of the previous.

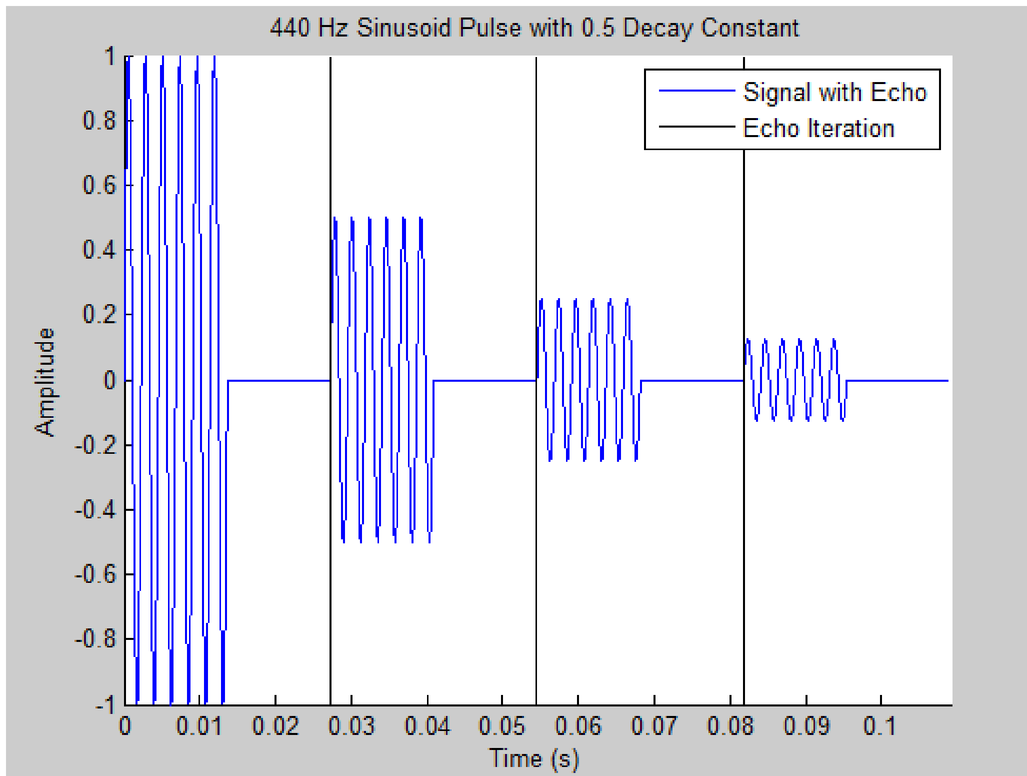


Figure 4-14: Echo Simulation in MATLAB

The user is able to select the feedback ratio of the previously delayed samples, as well as the rate of decay of the echo, using 8-bit MIDI controls. This allows for a variation of how quickly the previous samples will fade away and the length of time between a noise and its corresponding echo, which corresponds to being able to change both the size and dampening constant of an echo chamber. The decay and feedback MIDI controllers are linearly mapped to floating point values between 0 and 1. The Delay Times are linearly mapped to an amount of samples that that allows for up to two seconds of delay buffer storage. This module makes use

of the Cortex-A9's FPU, performing floating-point multiplication, which provides more precise results than would the same calculations in fixed-point arithmetic.

4.5 Low-Frequency Oscillator and MIDI Control

A Low Frequency Oscillator (LFO) module was created to allow the user to modulate effect parameters continuously, without having to physically turn the controller knobs. This type of module is common to many analog synthesizers, resulting in dynamic changes in effects. In early synthesizers, the user was required to physically plug an LFO module patch into the effect to be controlled each time a new effect was to be made dynamic. This system features a hardware module which applies the same selection process via digital multiplexing. The LFO module allows the user to select the rate (frequency), the depth (amplitude) and the pulse width (for square wave) of the oscillator. The user is also capable of choosing from different waveforms to modulate the effects (sine, square, saw, and triangle), providing different patterns of parameter manipulation. LFO rate can be controlled within the range of 0-20 Hz.

The controls for the LFO can be found in the GUI. Two independent LFO modules are implemented into the system to allow the user to modulate two different effects within the system. The modules are connect in series to allow for the possibility of both LFOs controlling the same effect. The user can select one of the following parameters to control:

- IIR High-Pass Filter Cutoff Frequency
- IIR Low-Pass Filter Cutoff Frequency
- FIR High-Pass Filter Cutoff Frequency
- FIR Low-Pass Filter Cutoff Frequency
- Compression Amplitude Clipping Threshold
- Compression Pre-Amp Gain Level
- Bitcrusher Number of LSBs to nullify
- Pulse Width of Signal Square Wave

Figure 4-15 shows the simulation of an LFO effect controller, with a 40Hz sinusoidal oscillation applied to a compression effect threshold level. This oscillation is applied to a 440 Hz sine wave, resulting in periodic compression of the signal.

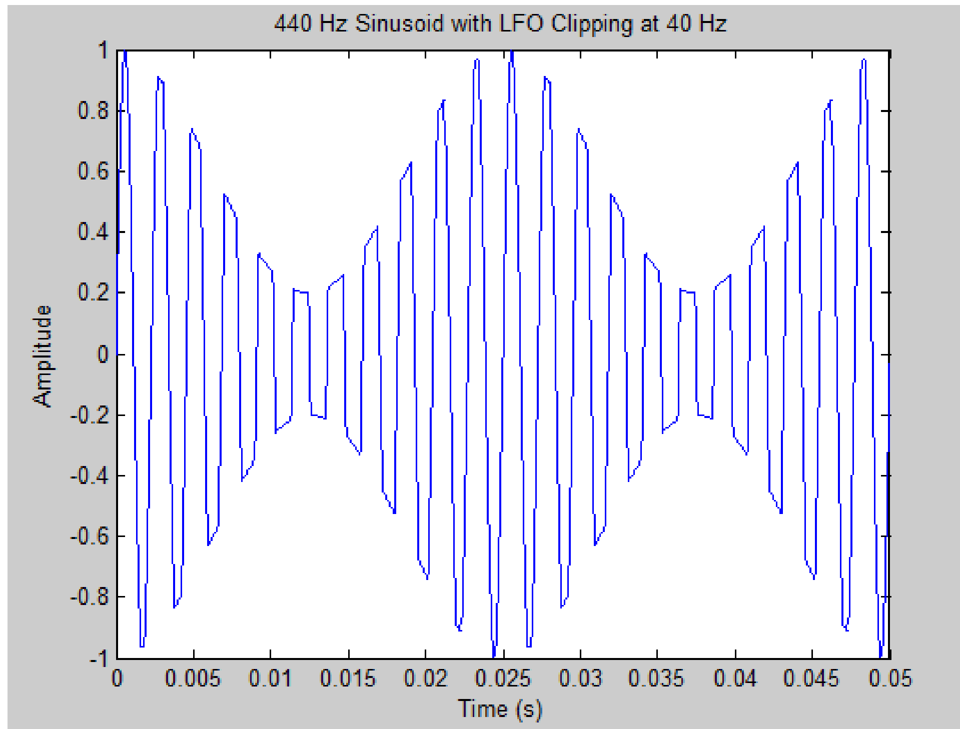


Figure 4-15: LFO Applied to Signal Amplitude

The LFO is constructed using a DDS oscillator similar to the one used in base waveform generation. This features the four different 24-bit resolution waveforms generated using 4096-sample periods. The module utilizes 8-bit MIDI data for frequency input, but instead of mapping the MIDI values to note frequencies, the values are linearly mapped frequencies between 0 and 20Hz. The user can define the rate of the LFO using controllers multiplexed by the GUI. The 24-bit waveform is then scaled down to 8-bits of resolution (1 sign bit and 7 data bits) using a bitwise slice of the upper 8-bits of sample data. This converts the LFO waveform into a MIDI-standard format oscillation to be used alongside data from the MIDI controllers for effects parameterization.

The 8-bit waveform is then sent through a multiplication stage for signal scaling, allowing the user to change the amplitude of the oscillation, thus altering the depth of the modulation. A second module was created to receive an 8-bit MIDI signal representing LFO depth, and linearly map this data to a 16-bit scalar value between zero and one. This is multiplied by the 8-bit LFO waveform using fixed-point multiplication. The output of the multiplication is a scaled version of the LFO waveform. This scaled signal is then utilized by a second module, implemented to

continuously add the waveform to the user-selected effect control parameter. This results in a continuously oscillating MIDI parameter, fully controllable by the user. Figure 4-16 graphically depicts the operation of the LFO effects control module, simplified to the application of one effect.

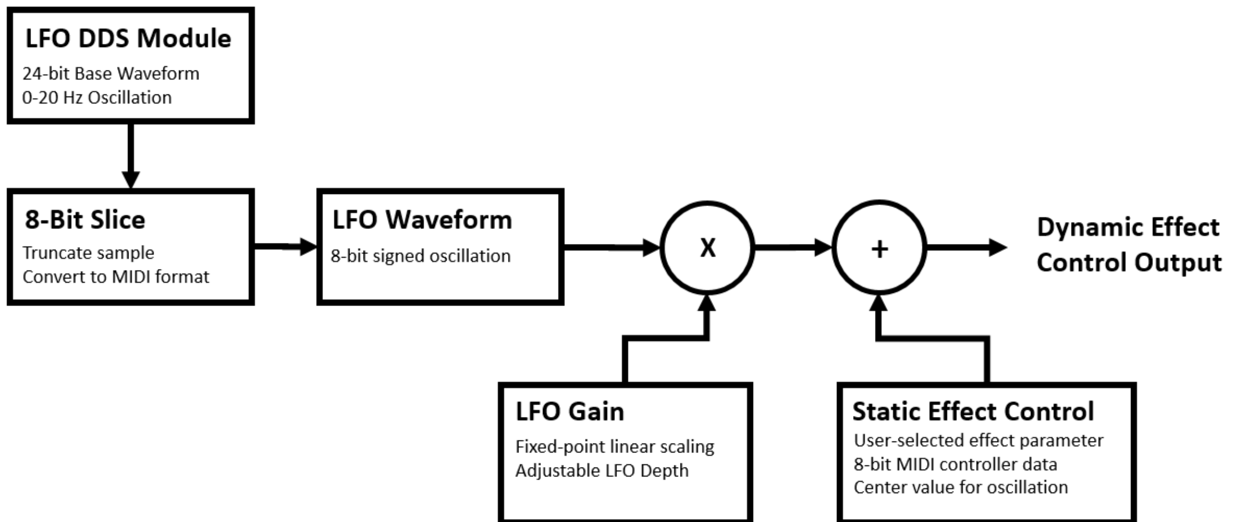


Figure 4-16: LFO Effect Control

4.6 Vibrato

A common synthesizer element that affects the frequency of a waveform is Vibrato. This effect makes use of an LFO to perform phase-driven Frequency Modulation (FM) on a signal, causing the fundamental period of the signal to expand and contract repeatedly. The modulation can be performed using an LFO signal to control a variable delay applied to the generated note waveform. Delay modulation in discrete time is a form of Phase Modulation (PM), defined mathematically in Equation 4-5.

$$y[t] = sin\left(2\pi f\left(t + x_{LFO}[t]\right)\right) \tag{4-5}$$

Applying a variable phase delay to a waveform is one of the simplest methods of achieving FM. In order to calculate the changes in the fundamental waveform frequency f, Equation 4-5 is manipulated using simple algebra to achieve the sinusoid defined in Equation 4-

6, which defines the dynamic frequency of the modulated waveform as the sum of f and the delay function divided by time.

$$y[t] = sin\left(2\pi t\left(f + \frac{x_{LFO}[t]}{t}\right)\right) \tag{4-6}$$

Figure 4-17 shows a simulation of the Vibrato effect, with a 20Hz sinusoidal oscillation controlling the shift in the effect's delay index value. The effect is applied to a 440 Hz sinusoid, which can be seen unchanged in the upper plot. The lower plot shows the result of applying the Vibrato effect to the waveform, in which the signal period can be clearly seen increasing, as the delay index increases.

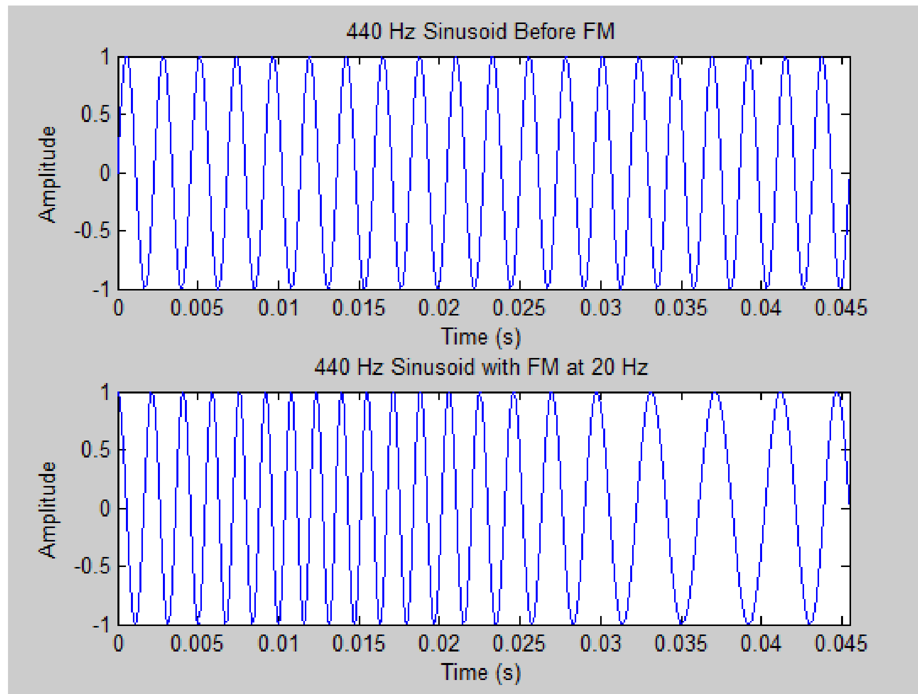


Figure 4-17: Vibrato Effect

The Vibrato effect is accomplished on FPGA hardware by introducing delay elements to store 42ms (4096 samples) of previous waveform samples, and modulating an index determining which delay element to send as the current output. An LFO output is routed to the 8-bit delay input to this module. While the amount of delay is increasing, the period of the waveform is increased, causing the signal frequency to decrease. While the delay factor

decreases, the opposite occurs, and frequency increases. Figure 4-18 shows the block diagram of the vibrato effect as it is implemented in the system.

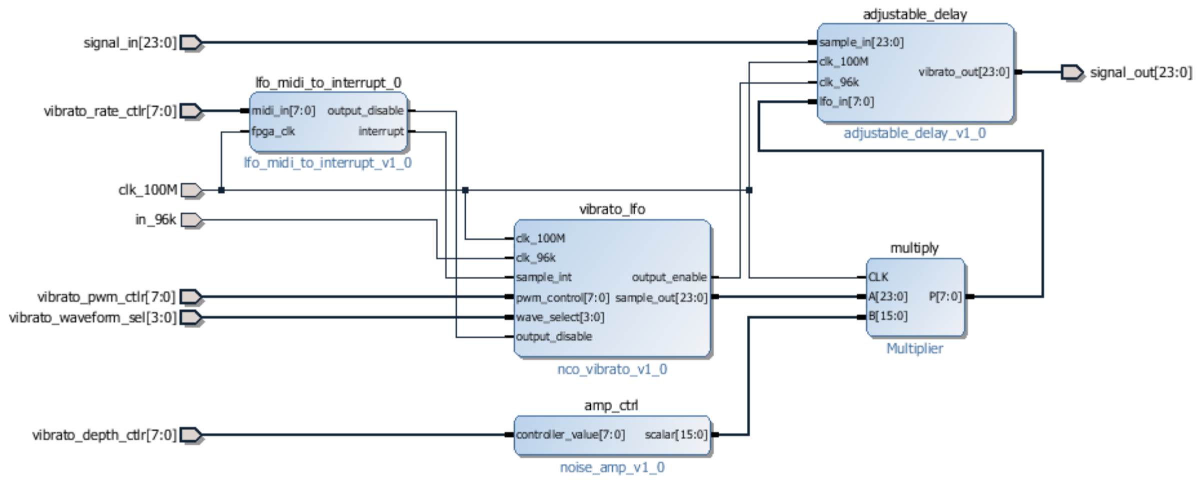


Figure 4-18: Vibrato System Level Implementation

This chapter provided a complete description of all effects modules implemented on our synthesizer system, and their integration within our top-level design. The following chapter contains a summary of methods and results of module and system testing.

5 Testing and Results

No design is sufficiently ready to be put into practice until it has undergone extensive testing. The synthesizer system was tested using several methods at every stage of the design process. System hardware was tested using computer-generated test benches, as well as using oscilloscope readings of internal signals and analog outputs. Software was tested using debugging tools built-in to the Xilinx SDK. This chapter outlines the methods used and results obtained from system testing.

5.1 Hardware Testing

At each stage in the development process of this project, new modules were tested as standalone units before they were integrated into the top-level system design. This was accomplished through the use of Verilog test benches, which allow an engineer to instantiate a designed hardware module under very specific input conditions. Using Verilog for Simulation, an engineer can model changes in input to a system and verify the circuit's behavior by observing its output. The use of test benches allowed the team to not only test the functionality of a digital circuit, but also to aid the process of debugging nonfunctional circuits. Verilog testing comprises writing a test fixture and instatiating a module as the Unit Under Test (UUT), which can be seen in Figure 5-1.

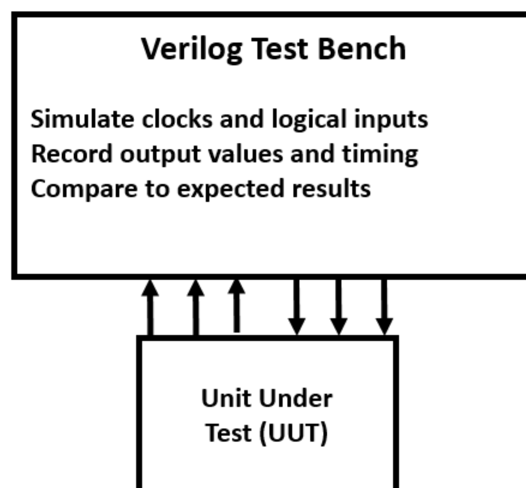


Figure 5-1: Verilog for Module Testing

5.1.1 Verilog Test Benches

After the hardware description of each module was completed, a test bench was designed for verifying that synthesized hardware matched the intended functionality of the component. Simulations were conducted within Vivado Design Suite running XSim. In some cases of debugging, it proved useful to conduct behavioral simulation as well as post-synthesis simulation, to determine if the cause of a functional flaw was inherent of the design or simply a result of implicit synthesis. For example, a module which passes all tests in behavioral simulation but fails post-synthesis is generally the cause of synthesizable Verilog code that could be written more explicitly, such that the synthesis tool interprets the correct netlist.

In order to fully test a hardware module, one must simulate all possible combinations of inputs. Test benches created in this project sought to produce as many input combinations as possible, while still meeting the scheduled goals of development. For this reason, much of the testing was focused on simulating normal expected system operations rather than extreme cases of input corruption or system error. Figure 5-2 shows simulation data for the testing of a fixed-coefficient FIR filter that was used to confirm proper timing synchronization with a Xilinx Multiply Accumulate block. This test bench simulates an input sample applied to a simple second-order filter, testing the circuit's ability to properly time multiplies using the Xilinx IP, and to confirm the validity of accumulation data.

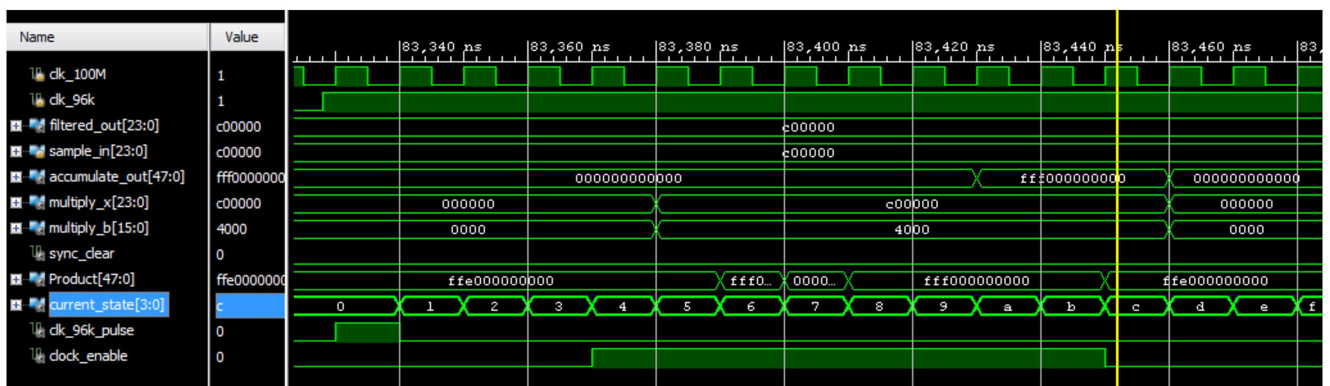


Figure 5-2: Timing Simulation of FIR Filter

Hardware testing was a critical component of Term C project development. During this term, a rigid schedule of effects module completion was laid out, with the intent of implementing several hardware-driven synthesizer effects per week. In order to confirm that the team had accomplished the goals of a given week, extensive testing was performed on each newly-developed module. This allowed the team to confidently continue with project development, knowing that each effect had been completed and was functioning properly.

5.1.2 Resource Utilization

The Zynq features an Artix-7-equivalent FPGA with 53,200 LUTs and 106,400 Flip-Flops for the creation of complex digital logic. As the complexity of the project grew, so did the importance of not exceeding the resources available to us for further development. At the conclusion of project development, the system occupied less than 30% of all Look-up Tables and 11% of Flip Flop state memory devices. This exemplifies the logical efficiency implemented in the design of a complex system. The vast remainder of unused logic cells allow for much expansion to be made in future project development, including additional synthesizer effects. Detailed resource utilization can be found in Table 5-1.

Resource	Utilization	Available	Utilization %
Flip Flops	11830	106400	11.12
Look-Up Tables	14611	53200	27.46
Memory LUTs	1588	17400	9.13
I/O	32	200	16.00
BRAM	31.50	140	22.50
DSP48 Slices	16	220	7.27

Table 5-1: System Resource Utilization

5.2 Software Testing

The functionality of control software is not only dependent on the team’s coding ability, but also on the ability to find and fix problems present at any time during code development. Testing of software is also necessary to ensure that the interface to FPGA hardware is functional as well. With the software providing overall system control, it is crucial to confirm the accuracy and reliability of every aspect of this component of the project. With the use of

debugging tools present in Xilinx SDK, code stability was tested extensively to ensure proper execution as the system developed in complexity.

5.2.1 Debugging Techniques

The software debugging tools included in Xilinx SDK allow for verification of inputs to the system as well as bitwise manipulations of data. The ability to pause the execution of the system and check the state of software and memory at any given moment is quintessential for this process. For example, the debugging tool was utilized to confirm functionality of the UART serial input from the host computer. This ensured that the MIDI data bytes were coming in and correctly being parsed, even with many sequential inputs. The debugger was also used to check the sample output from FPGA hardware to verify that the signed 24-bit integer was received by the system from the AXI peripheral in the correct format and interpreted properly.

Debugging techniques were used to verify the functionality of the delay and echo functions, to ensure sample data was correctly loaded into the respective arrays. The SDK debug tool was utilized at many development steps through the course of the project, and proved to be an essential part of embedded software testing. Figure 5-3 provides an example of the debug tool in use, with the Expressions window utilized to confirm the functionality of GPIO readings of new samples produced in hardware.

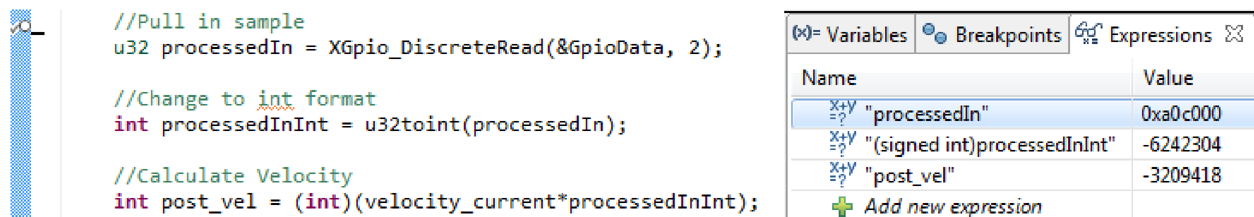


Figure 5-3: Verification of Hardware Input via Debugging Tool

5.3 System Testing

The oscilloscope used for testing system audio output was an Agilent Infiniium 54833D, capable of sampling at a rate of 4 GSa/s. This sampling rate allows for fine details of the analog output waveform to be captured and observed in high resolution. The oscilloscope is also

capable of performing a Fast Fourier Transform (FFT) in software for spectral analysis of the signal. This was utilized for testing the functionality of filter modules, for verification of signal attenuation across the frequency spectrum. Digital oscilloscope inputs were also utilized in the verification of several internal logic signals, including the clock divider module for DDS synthesis. This was used to verify that the waveform generation block was provided step signals at precise frequencies, matching their corresponding note frequencies. Supplemental to oscilloscope testing, speakers were used to listen to system output and confirm audibly the results each synthesizer effect. System output sounds were compared to those produced by software synthesizers emulated on the laboratory computer.

5.3.1 Waveform Generation

Functionality of the DDS waveform generation stage was verified for each of the four base waveforms. This ensured that each of the waveforms were being generated at the correct frequency, and no instability or waveform abnormalities were present. The following figures show oscilloscope captures of the four waveform types at the same note frequency 261.63 Hz, or MIDI note 60 (C4 on a musical scale). Figure 5-4 depicts a pure sinusoid at this frequency.

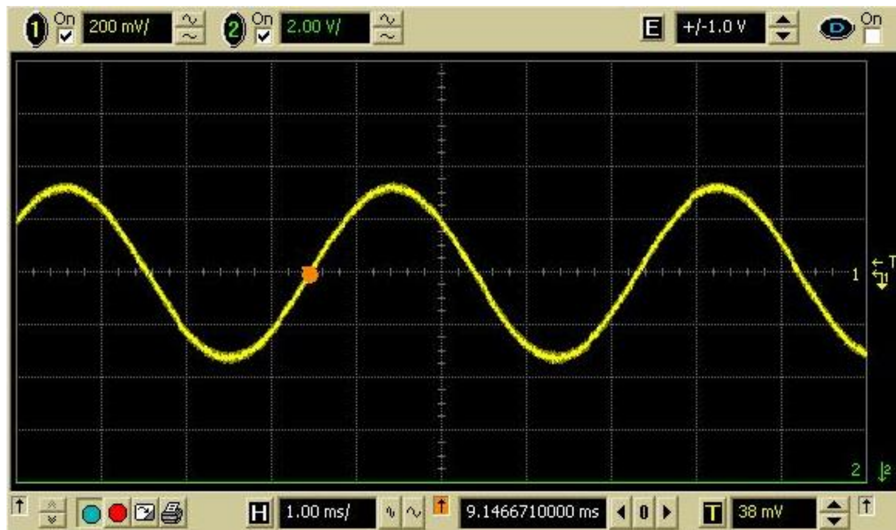


Figure 5-4: Sine DDS Output

Figure 5-5 shows a sawtooth wave at MIDI note 60. The sharpness of the waveform's saw blades can be observed once per period, indicating near-infinitely-sloped transitions from the signal's minimum value to its maximum.

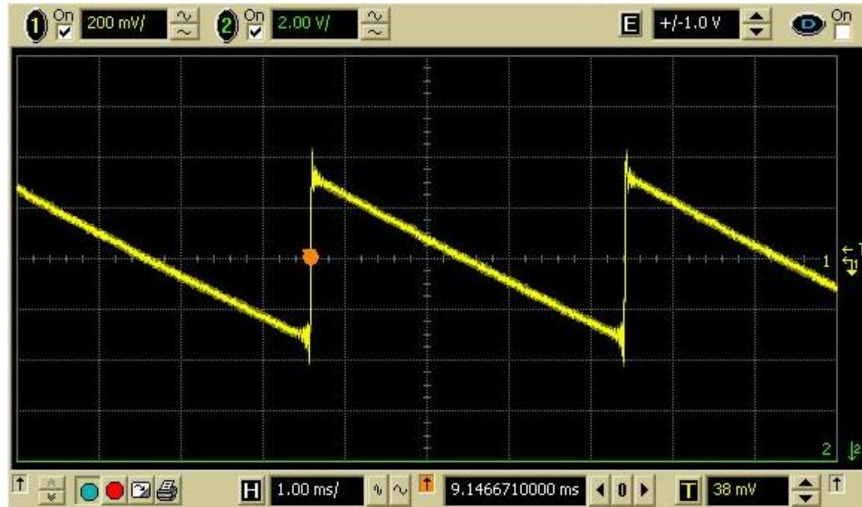


Figure 5-5: Sawtooth DDS Output

Figure 5-6 depicts a square waveform played at MIDI note 60. In this figure, the duty cycle of the waveform is set to just under 50%, with the lower portion of the signal slightly longer in duration than the upper part.

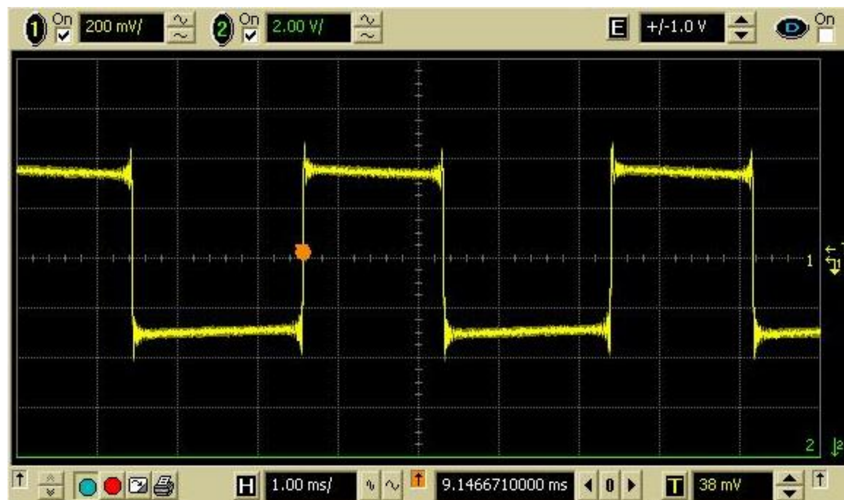


Figure 5-6: Square DDS Output

Figure 5-7 contains an oscillogram of a triangle waveform, at the same frequency as the above plots. Each of the four waveforms feature the same minimum and maximum values, corresponding to identical waveform amplitudes.

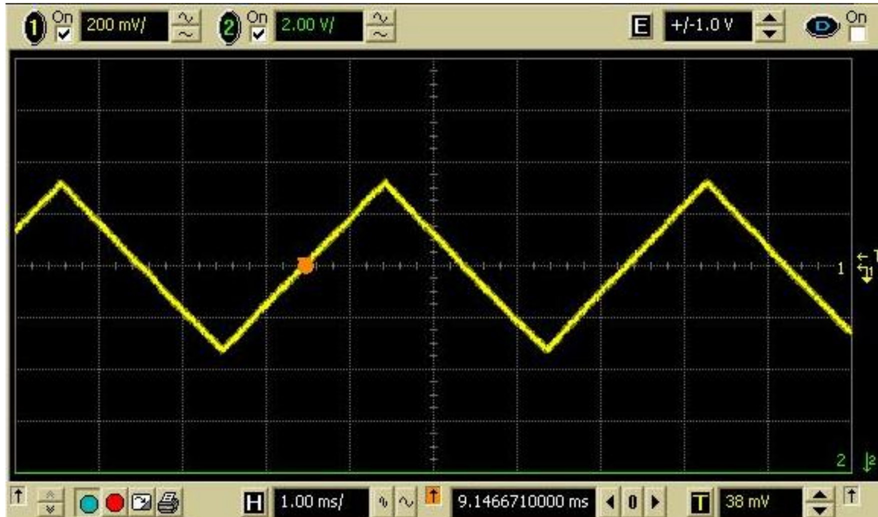


Figure 5-7: Triangle DDS Output

5.3.2 ADSR Envelope

The amplitude modulation envelope was tested on the oscilloscope as well. Using a prolonged scope timescale, many samples of a synthesized waveform were captured. The ADSR parameters were set to mid-range values, to exaggerate each stage of the amplitude envelope. Figure 5-8 shows an annotated oscillogram generated by the envelope when applied to a sine wave. This verifies the operation of the ADSR module and can be further exaggerated using higher valued parameters.

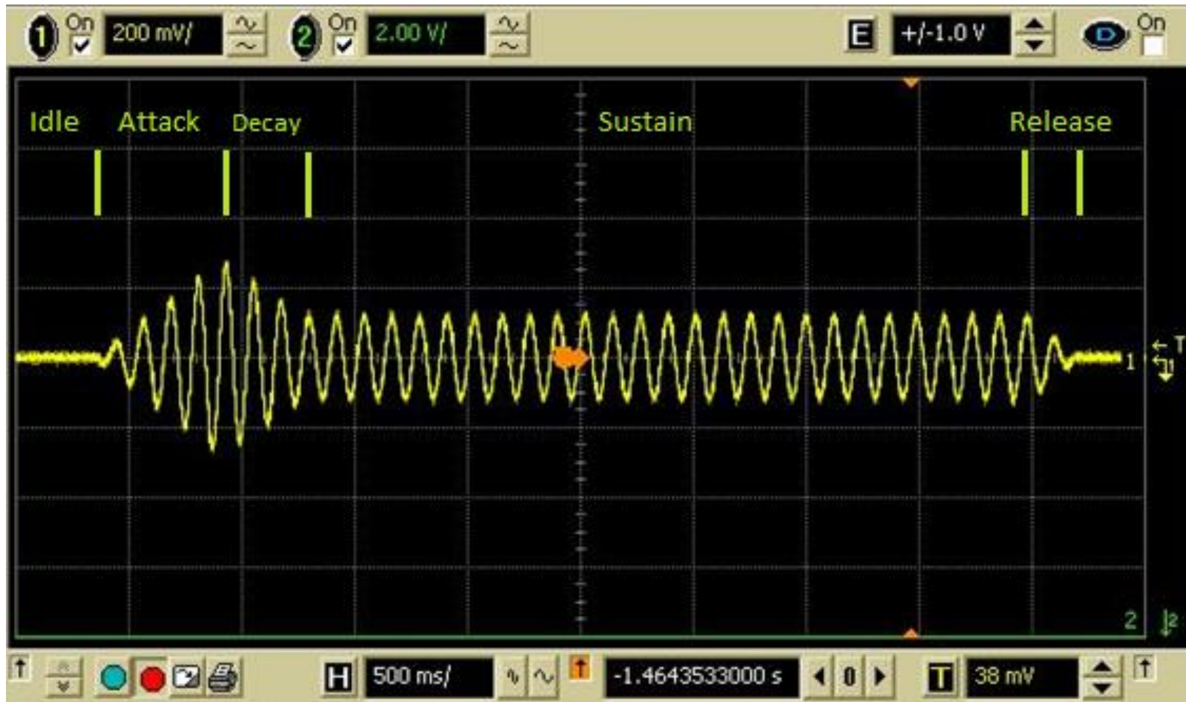


Figure 5-8: ADSR Scope Capture

5.3.3 Filter Stages

The four filter modules were tested extensively using three methods of verification. First, the cutoff frequencies of each filter were modulated across their full-scale ranges and confirmed audibly to be both stable and capable of producing their intended synthesizer sounds. Next, the filters were tested using computer capture of waveforms and MATLAB processing for spectral analysis of the signal before and after filtering. Finally, both FIR and IIR type filter stages were verified to be functional using the oscilloscope in the time domain, where the different properties of each filtering method could be observed.

Figure 5-9 shows an oscillogram of a sawtooth wave, with various filter stages applied. Figure 5-9-a features a pure sawtooth waveform, in which the unmodified sharpness of the saw blades can be clearly observed. An FIR Low-pass filter was applied to the same sawtooth signal with the cutoff frequency set to 4.076 kHz. This filter attenuates the high harmonic components of the waveform, resulting in a reduction in the sharpness of the signal. In the oscillogram of Figure 5-9-b, the blades appear duller, with smoother transitions from the minimum to

maximum signal values. An IIR filter set to the same cutoff frequency was applied to the same sawtooth wave. The high harmonics are attenuated at a similar level to that of the FIR filter, but the waveform is much more visibly distorted. This exemplifies the non-linear phase response effect observed with IIR filters. This nonlinearity causes harmonic components of the waveform to be shifted at different phases, causing to sawtooth to begin to lose its signature shape, as can be seen in Figure 5-9-c. Increasing the cutoff frequency further results in more reduction of the sawtooth peaks and further phase distortion to harmonic components of the signal.

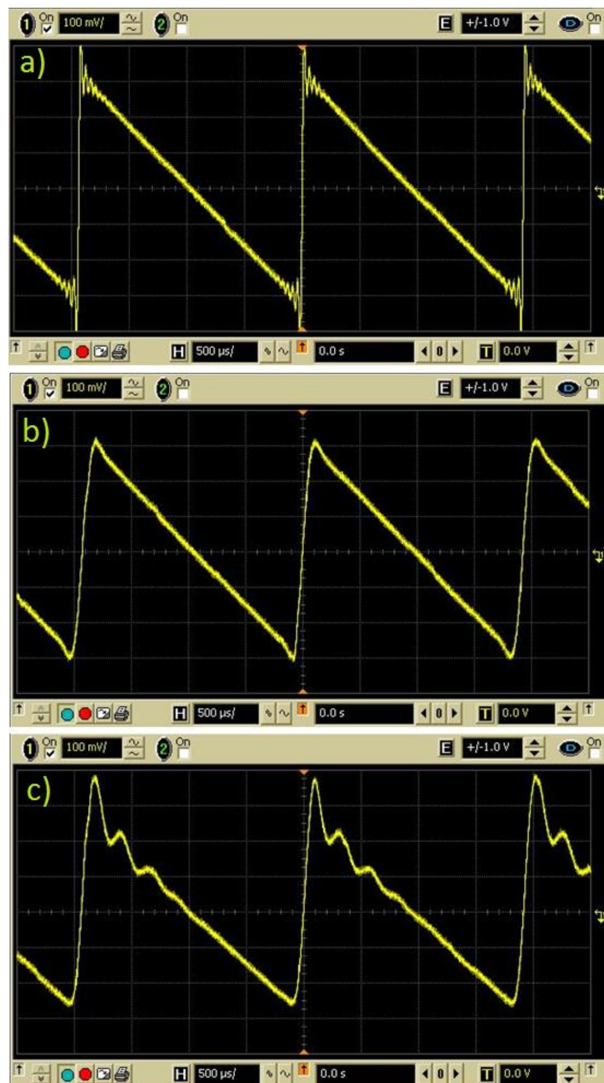


Figure 5-9 a-c: FIR and IIR FIlters, Sawtooth

Figure 5-10 shows a comparison between the FPGA filter implementation and a similar 64th order FIR filter generated in MATLAB using FDA tool. The cutoff frequencies of both filters

were set to 2.7 kHz and are nearly identical, except the FPGA implementation uses 16-bit fixed-point coefficients while MATLAB implementation utilizes more precise 64-bit floating-point values. Error between the two filter implementations can be seen at the higher end of the frequency spectrum, where the FPGA filter is only able to attenuate the signal by -55dB, while the MATLAB implementation is able to reduce the harmonic amplitudes even further. This level of attenuation is acceptable for this application, as -55dB corresponds to multiplying these harmonics by a factor of 3e-6, reducing signal harmonics to fractional amplitudes that are lost due to the use of a fixed-point waveform.

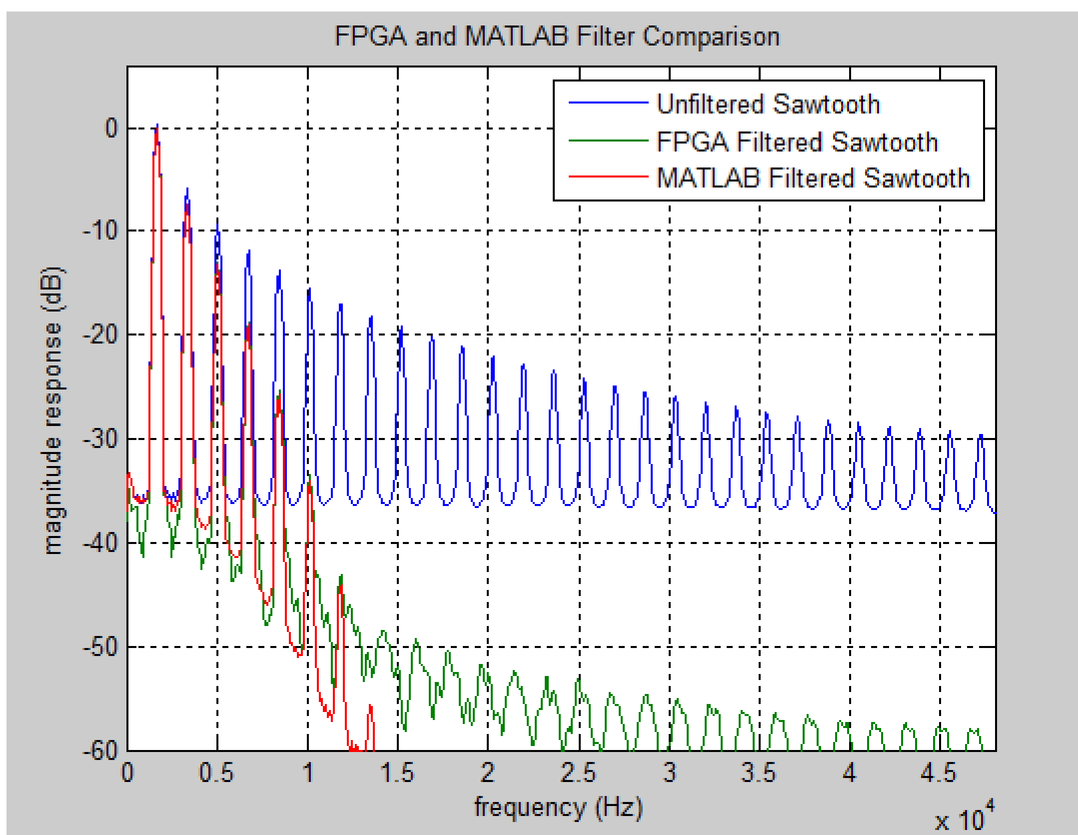


Figure 5-10: FIR Filter Comparison (MATLAB to FPGA)

5.3.4 Compression and Overdrive

The compression and gain modules were tested in the time domain using the oscilloscope. The amplitude clipping effect is very well-defined on oscilloscope output and can easily be seen observed using sine wave for testing purposes. Increasing the clip level visibly reduces the overall waveform amplitude, replacing the smooth sinusoid with flat peaks. When

the gain factor is introduced to produce Overdrive, the sinusoid increased in amplitude, causing it to become even less smooth. With sufficient gain applied to the signal, the waveform becomes nearly a square, which is known to be much richer in harmonics. Figure 5-11 shows oscillograms of several stages of clipping and Overdrive. Figure 5-11-a depicts a pure sinusoid, with no distortive effects applied.

The Compression effect was applied to the sine wave with the threshold level set to 62, corresponding to 48% of the original full-scale amplitude range. The round peaks of the waveform were capped, creating a signal that appears trapezoidal, as can be seen in Figure 5-11-b. The waveform peak-to-peak voltage was reduced from 6.4 to 3.19, corresponding to a 48% reduction. With the clipping threshold held constant, the pre-amplification gain was increased to near its maximum value, resulting in the nearly-square waveform observed in Figure 5-11-c.

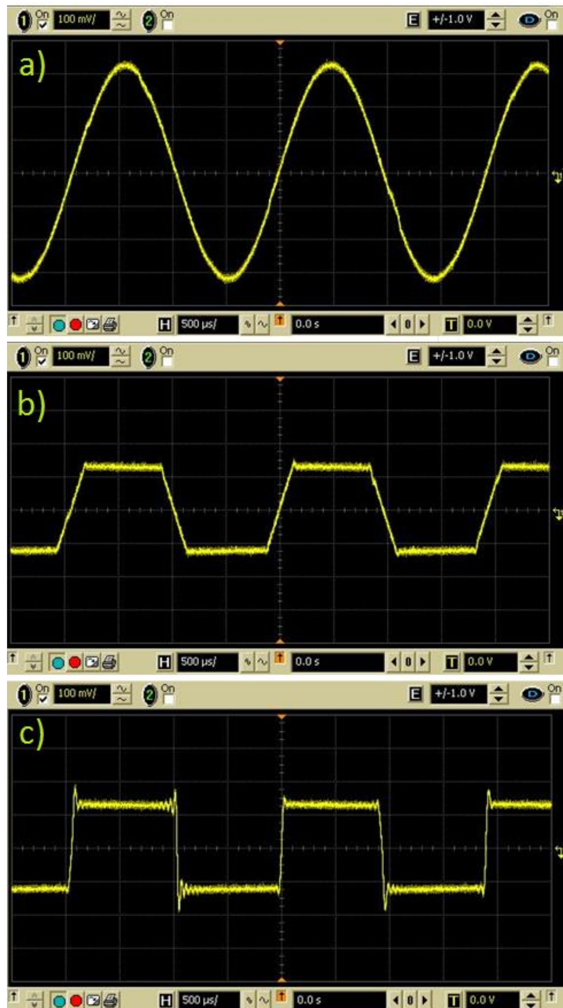


Figure 5-11 a-c: Compression and Overdrive

5.3.5 Waveform Resolution Reduction

The Bitcrusher effect was tested at various levels of waveform resolution reduction to confirm module functionality. Figure 5-12 shows an oscillogram of a 440 Hz sine wave with two bits of waveform resolution. In this test, the original 24-bit signal now occupies four discrete amplitude levels, resulting in a great reduction in tonal quality.

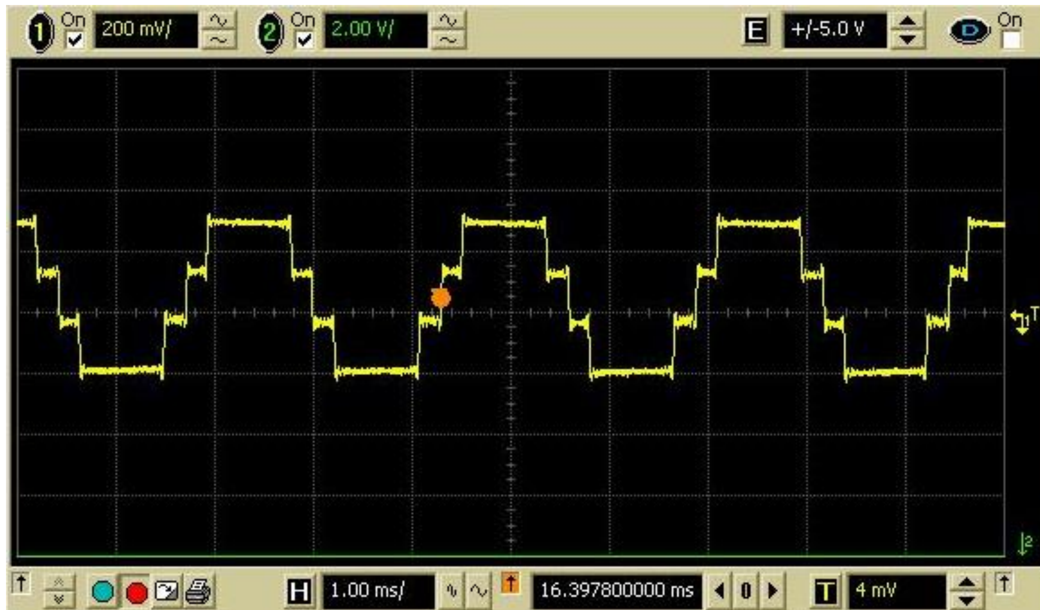


Figure 5-12: Bitcrusher Effect Test

5.3.6 Delay and Echo

The delay-based effects were verified through the use of the desktop speakers, as the delayed signal could easily be verified in its repetition over time. They were also verified on the oscilloscope, where the delayed signal can be seen following the original signal over time. An oscillogram of the echo effect in practice over the course five seconds can be seen in Figure 5-13. In the figure, a sawtooth note of short duration is played once (at t=500ms), and is echoed with a low decay rate, allowing for multiple echoes to be seen decaying for several seconds after the note has been released.

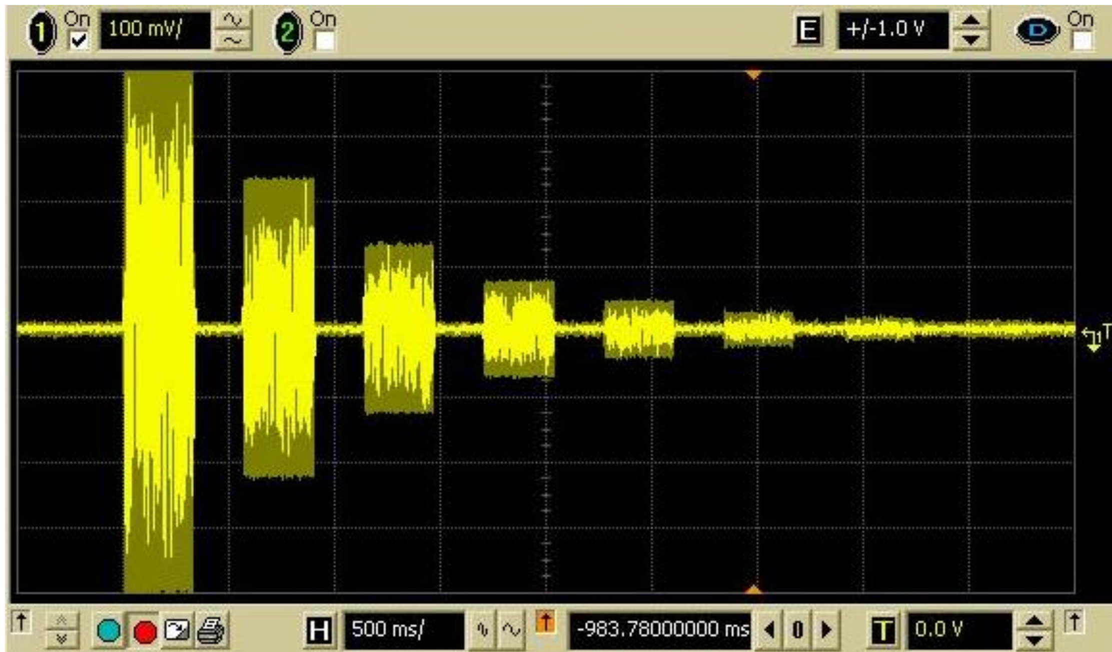


Figure 5-13: Echo of Signal over time

5.3.7 Vibrato

The vibrato effect modulates the frequency of the signal at a user defined rate and depth. The functionality of this effect was confirmed audibly as well as visually on the oscilloscope. Using a maximum depth parameter to exaggerate the frequency change of the signal, many different frequencies can be observed on the oscilloscope for a single input note. Figure 5-14 shows a scope capture of the vibrato effect on a static note, with a square wave selected. A rate of modulation was chosen such that changes in the period of the waveform can be observed, thus indicating changes in frequency. From left to right, the periods of the square waves can be observed to be increasing, as the sinusoidal LFO controlling the Vibrato module reaches a peak, and the resulting square wave becomes increasingly lower in frequency.

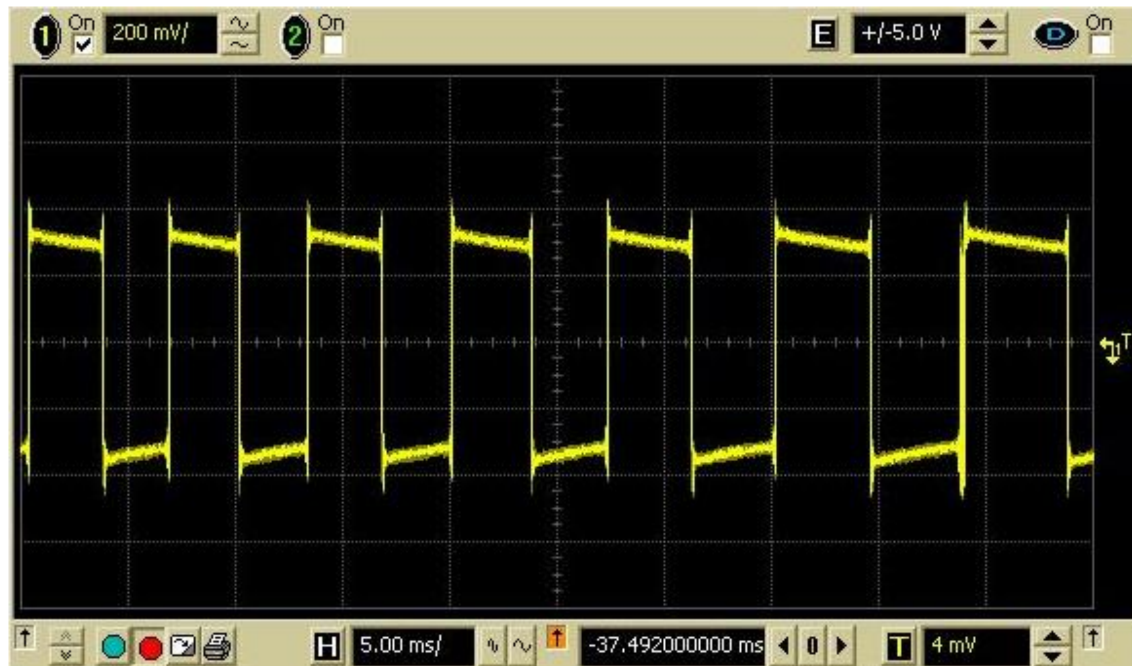


Figure 5-14: Square Signal with Vibrato Effect

5.3.8 LFO

The Low Frequency Oscillator effect control modules allow for dynamic changes in every effect parameter on the system. Extensive testing was performed to ensure that these modules were successful in their ability to modulate each effect parameter. This is best demonstrated using an LFO to change the clipping threshold of the compression effect. An LFO sinusoid was used to modulate the compression threshold, resulting in the oscillogram shown in Figure 5-15**.** The dynamic changes to this effect parameter can be clearly observed, resulting in a sinusoidal change in signal amplitude over time.

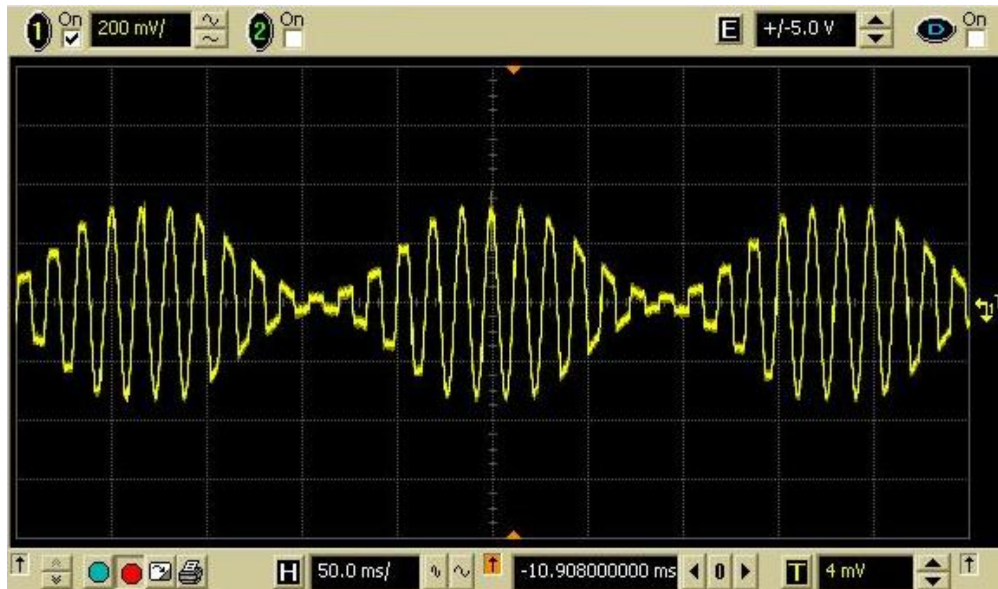


Figure 5-15: Sinusoidal Compression LFO Effect

An LFO effect control module was then applied to the controlling input of the FIR high-pass filter module. This caused the cutoff frequency of this filter to oscillate, resulting in the waveform shown in the oscilloscope capture of Figure 5-16. As the cutoff frequency increases, the low frequency harmonics that make up the fundamental shape of the sawtooth can be seen attenuating until the center of the plot, when the cutoff frequency begins to decrease and the sawtooth regains its signature shape.

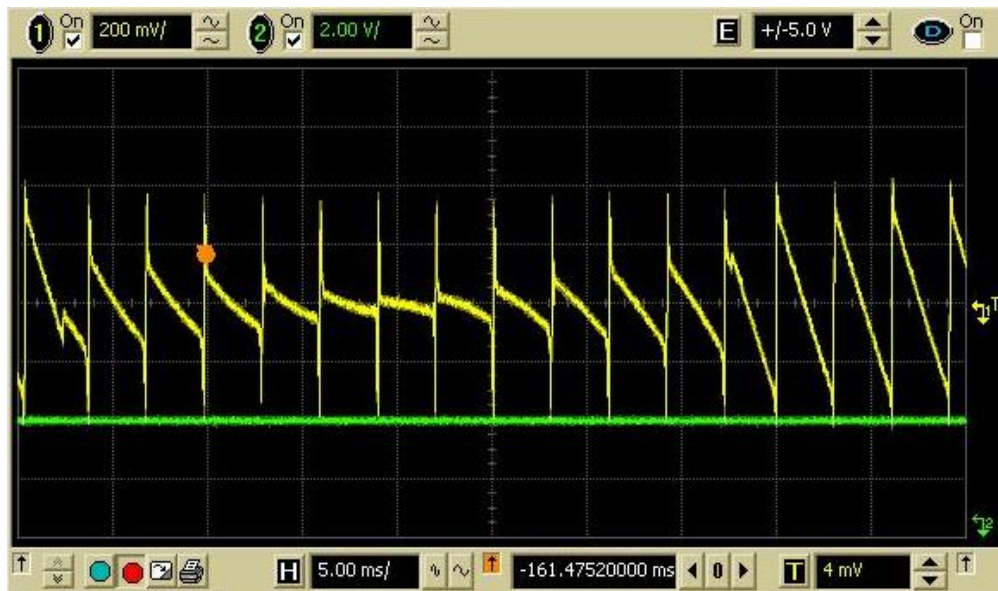


Figure 5-16: LFO Applied to FIR High-Pass

5.4 Signal-to-Noise Ratio

The analog signal produced by the codec is susceptible to noise, as it is transmitted through an auxiliary audio cable to the speakers. This noise is undesired and can have a negative effect on the signal being generated. Using the root mean square (RMS) voltages of the noise and the signal, it is possible to calculate the signal to noise ratio (SNR). The measured RMS voltage for the noise was 6.3mV and the best case signal RMS voltage was measured to be 229mV. Using Equation 5-1, the SNR was calculated to be 1321.26. Equation 5-2 is the conversion from SNR to SNR dB, which is represented logarithmically using decibels. Using this conversion, the SNR_{dB} of the final signal was determined to be 31.21dB.

$$SNR = \left(\frac{RMS_{signal}}{RMS_{noise}}\right)^2 \tag{5-1}$$

$$SNR_{dB} = 10log_{10}(SNR) \tag{5-2}$$

5.5 Real-Time Capability

A digital instrument cannot be deemed useable if there is any delay in overall throughout, as this can be frustrating to the musician. The user should not need to compensate his actions at the cost of the hardware. A digital system not running in real-time is prone to error, and can result in instability and missed output samples, reducing the quality of output. The synthesizer's input-to-output delay is so minute that it is undetectable by a human user. However, the overall real-time performance was examined to determine the throughput capabilities of both the hardware and software. There are limitations to the real-time ability of the system, as input data is polled via UART at a rate of 115200 baud. UART protocol utilizes one start and two stop bits in addition to the 8-bit MIDI message data, resulting in 11 bits per data byte transmission**.** Three bytes of MIDI data are sent to the system for every key press, resulting in a 286µs delay at this transmission rate. This allows for over 3000 notes to be transmitted per second, which is sufficient for any level of user playing music at any humanly – possible tempo.

The second concern when determining real-time operation is ensuring that both the hardware and software are able to complete their tasks in the allotted time between each 96

kHz interrupt. This is crucial, as processing deadlines must be met in order for the system output to represent the current sample in time.

Effects processing logic, operating on a 100MHz clock, has a deadline of 1041 clock cycles (10us) to complete all signal processing in time for concurrent sample output. Some modules are able to complete processing in as few as 2 clock cycles, while more computationally complex effects can take hundreds of cycles. Filtering effects take many clock cycles due to the sequentially implemented filter coefficient multiply-accumulation stages. Table 5-2 shows the number of clock cycles for each effect implemented on the programmable logic, and the total calculated computation time for effects processing on each waveform sample.

Effect Module	100 MHz Clock Cycles	Time (ns)
ADSR	3	30
LFSR	7	70
FIR Filter x2 (64-order)	266 + 266	5320
IIR Filter x2 (4-order)	86 + 86	1760
Gain	4	40
Compression	2	20
Bitcrusher	2	20
Vibrato	2	20
Total Effects Throughput	**728**	**7280**

Table 5-2 - Clock Cycle Count of Hardware Effects

The effect processing throughput was confirmed in practice on the completed system by determining the time taken for a sample to pass through each of the effects modules. Using an oscilloscope, the time difference between new waveform sample generation (determined by the assertion of the output_enable signal on the DDS block) and the assertion of the output_enable signal on the final effects stage. With all effects active, this time was measured to be 7205 ns, as can be seen in the oscillogram of Figure 5-17.

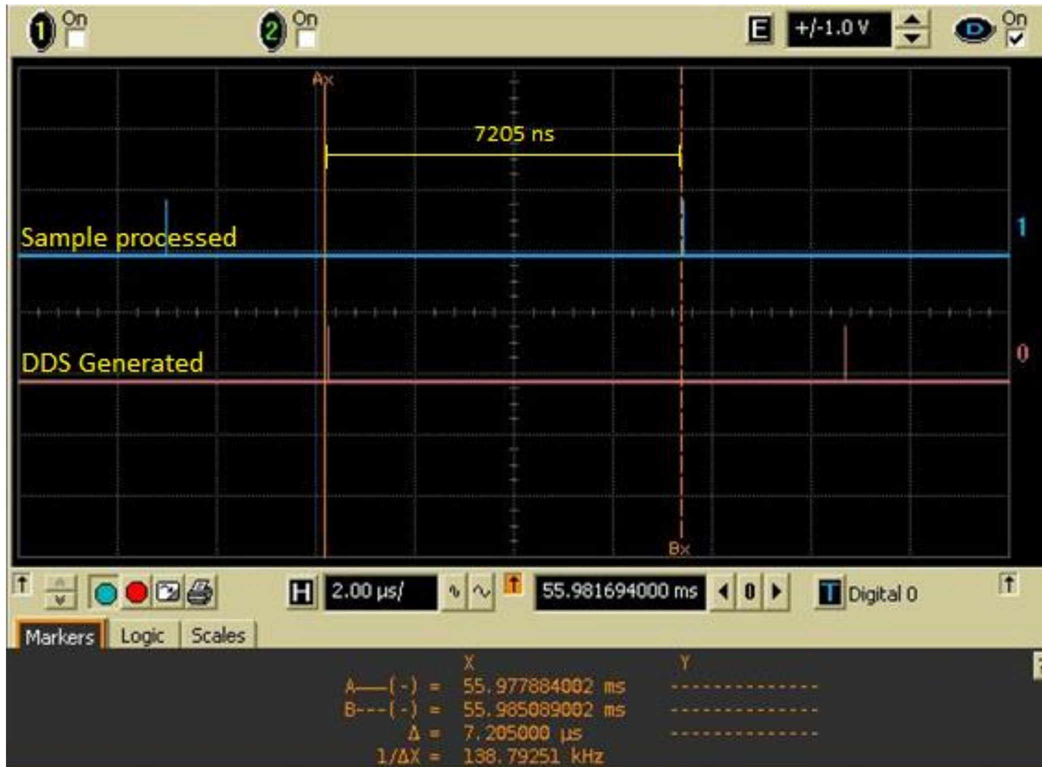


Figure 5-17: Effects Processing Throughput

Overall system throughput was determined by measuring the difference in time between the note_on pulse, which generated when a user presses a key, and the moment the first sample of a waveform appears at the system's analog output. This encompasses the time required to generate a waveform sample, complete effects processing, transmit sample over AXI, perform software effects processing, and send the sample to the codec for system output. This delay was measured to be 3.309 ms, as can be seen in the oscillogram of Figure 5-18, in which the green signal represents note_on and the yellow waveform is the synthesizer output.

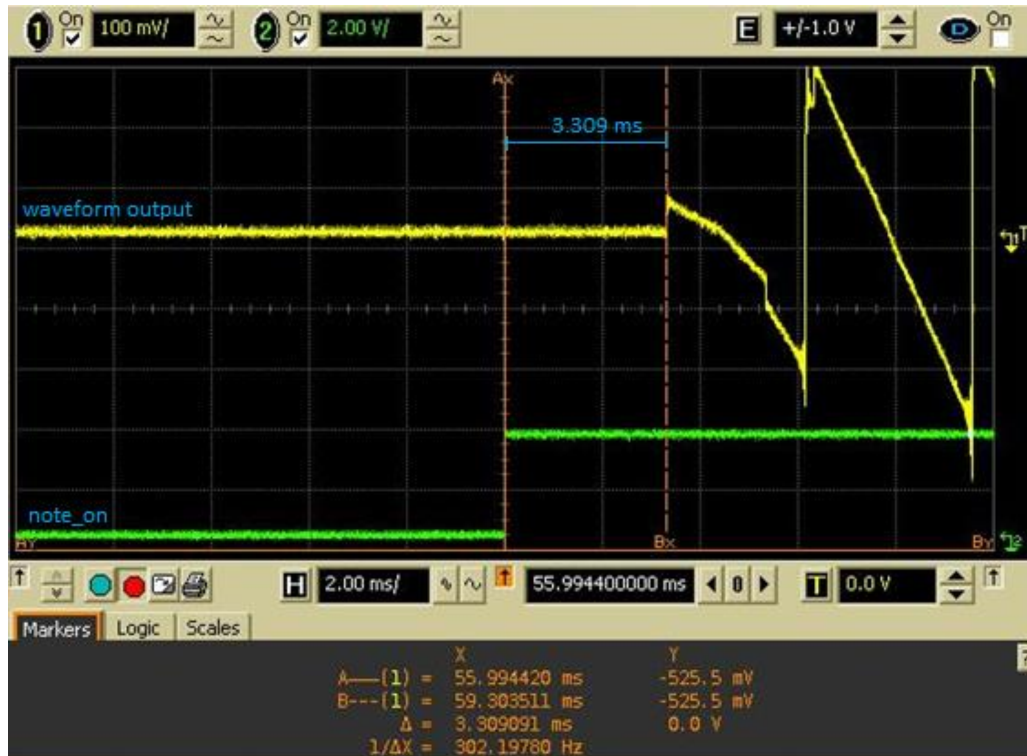


Figure 5-18: System Throughput Latency

An ideal electronic instrument has a throughout delay near zero seconds, to appropriately model the response of an acoustic instrument. The measured system delay was compared to the latency response of a similar keyed instrument, a Grand Piano. This acoustic instrument contains several moving mechanical parts that introduce a 30ms delay between key press and string vibration [23]. Thus, a latency of 3.309 ms is more than acceptable for the development of a musical instrument.

This chapter provided a summary of low-level testing of hardware modules developed for our system, as well as top-level synthesizer functionality testing. This included system throughput testing, from USB MIDI input to waveform output. The final chapter contains a summary of project accomplishments and a discussion of future expansion to the system.

6 Conclusions

This project consisted of the design and implementation of a digital music synthesizer on a Xilinx Zynq SoC, capable of performing subtracting synthesis and producing a multitude of effects. The finalized system met all design goals, which are outlined below in Table 6-1.

Goal	Implementation
Input delivered to system for user control	MIDI controller, Python-powered GUI on host computer
Complete real-time system control	ARM Cortex-A9 executes software for top-level control
High quality waveform generation	DDS using 4096-sample 24-bit wavetables on PL
Multiple effects processing stages	Real-time DSP techniques implemented on PL
Real-time system operation	96 kHz signal output powered by ADAU Audio codec

Table 6-1: Project Goals

The completed system offered the high-speed control throughput of a modern digital synthesizer, successfully meeting all requirements laid out at the beginning of the project. Project development on the Zynq SoC allowed for the design of a system containing both embedded software and programmable logic components. The use of embedded software allowed for execution of memory-intensive tasks and implementation of control logic. Performing signal generation and processing in programmable logic allowed for the creation of many separate effects modules, carried out independently from software for real-time operation. This project sought to implement many synthesizer effects featured on modern systems. This goal was accomplished, though the implemented modules represent only a subset of the vast multitude of well-known synthesizer effects. The completed system can be seen pictured in Figure 6-1.

Figure 6-1: FPGA Digital Synthesizer System

The act of meeting the above requirements was not without its challenges. The original design for input implementation featured the USB MIDI device connected directly to the Zedboard, with embedded software on the Zynq chip configuring the device as its USB host. This proved to be a rather arduous task, requiring the design of bare-metal device drivers and the study of USB protocol. The Zedboard is designed to act as a USB host through the use of XiLinux running on the Cortex-A9, which would require the entire system control block to be implemented in Linux instead of using a bare-metal embedded system. In order to avoid the overhead of running an operating system and to simplify the embedded C design component of the project, USB host functionality was outsourced to the computer. This implementation was later expanded to also feature the GUI for enhanced system control.

Another development challenge was encountered during design of the filter modules. Rather than calculate each set of filter coefficients in MATLAB and store them permanently in combinational logic, the team had originally desired to calculate these constants in real-time. Fixed-point filter design using windowing techniques was simulated in MATLAB to determine if stable, high-quality filters could be produced. Within the system, two MIDI input parameters would set the upper and lower bounds of a rectangular window, upon which an Inverse Fast Fourier Transform (IFFT) would be performed to generate the impulse response of an FIR filter. The resulting filters did not produce the same sharp pass- to stop-band transitions observed from filters developed using MATLAB's FDA tool. Due to the high level of importance of this

filter characteristic, the modules were designed using FDA tool-generated coefficients, which were stored in combinational logic on the FPGA. This method proved to be a simple solution to the problem, considering an 8-bit MIDI control parameter only allows for 128 discrete input values, limiting the number of possible filter cutoff frequencies.

Timing closure across multiple clock domains also proved to be a challenge. The original system design featured every programmable logic module operating on one 100 MHz clock. In practice, it was found that using a 100 MHz clock to operate the NCO clock divider module for DDS did not provide sufficiently accurate frequencies for waveform generation. Using the PLL to derive a second clock at 200 MHz allowed for this module to generate waveform step interrupts with twice the precision, at the cost of timing synchronization. The timing issue was solved by using the 100 MHz clock and several state elements to create a synchronizer circuit, which reduced the risk of metastability occurring at the clock domain crossing.

This project proved to be an excellent learning experience for the team. The development of a complex digital music synthesizer required independent study of many topics not presented in undergraduate coursework at this university. Over the course of the first term of the project, the team learned much about Zynq SoC architecture and development using Vivado Design Suite. Real-time two-way communication between logic hardware and embedded software on this chip was necessary for system operation. The team was able to apply real-time fixed-point processing techniques learned in signals courses to digital logic, a paradigm in which even the smallest of timing mistakes can cause a system to be unstable or even inoperable. As musicians, the team was familiar with the basic concepts of many of the synthesizer effects implemented, but learning the mathematics and theory behind each of the effects was essential for system development. This allowed the team to apply concepts learned as engineering undergraduates to a field in which we were personally interested, bridging the gap between computer engineering and music.

6.1 Future Work

For future development on this project or MQPs of a similar nature, several additional expansions are possible. One optional system component that was not completed due to time

limitations was polyphony, in which multiple notes can be played simultaneously. Many synthesizers are monophonic, but the addition of polyphony allows the user to play chords for increased musical functionality. The development of this component would require additional control logic, the duplication of several hardware modules, and hardware multipliers for the scaled addition of each individual synthesizer voice.

Another possible addition to this project is the implementation of a system capable of performing effects processing on external signals. The ADAU1761 codec features an analog input, capable of capturing high-quality 24-bit samples at 96 kHz. This could be utilized to allow for a user to play an acoustic instrument or sing into a microphone, with audio effects processing performed on the analog signal. A synthesizer that features a microphone input for this purpose is called a Vocoder, and generally includes additional effects tailored to voice processing, such as pitch shifting.

This MQP resulted in the development of a small subset of the vast number of synthesizer effects that exist in modern systems. Due to the schedule limitations of a three-term project, the project implemented some of the most common modules considered essential to any synthesizer. Future work could focus on the development of more of these effects, which would be limited by the amount of programmable logic available on the Zynq chip. Some effects that would be possible to add to the system include flanger, chorus, and phasor effects. These are well-known and are desirable features found in many synthesizers.

7 References

[1] "The 'Musical Telegraph'," [Online]. Available: http://120years.net/the-musical-telegraphelisha-greyusa1876/.

[2] Wikpedia, "Wikipedia, the Free Encyclopedia," 2015. [Online]. Available: http://en.wikipedia.org/wiki/Sequential_Circuits_Prophet-5.

[3] V. Beal, "Moore's Law," [Online]. Available: http://www.webopedia.com/TERM/M/Moores_Law.html.

[4] Xilinx Inc., "FPGA vs ASIC," 2015. [Online]. Available: http://www.xilinx.com/fpga/asic.htm.

[5] G. Reid, "Synthesizing Tonewheel Organs," November 2003. [Online]. Available: http://www.soundonsound.com/sos/nov03/articles/synthsecrets.htm.

[6] M. Ottewill, "Subtractive Synthesis," [Online]. Available: www.planetoftunes.com/synthesis/subtractive-synthesis.htm. [Accessed 13 April 2015].

[7] Burk, Polansky, Repetto, Roberts and Rockmore, "FM Synthesis," Columbia University, [Online]. Available: http://music.columbia.edu/cmc/musicandcomputers/chapter4/04_07.php. [Accessed 19 November 2014].

[8] Wikipedia, "Synthesizer, ADSR," [Online]. Available: http://en.wikipedia.org/wiki/Synthesizer#ADSR_envelope. [Accessed 12 April 2015].

[9] Sound on Sound, "Synth Secrets: Modulation," February 2000. [Online]. Available: http://www.soundonsound.com/sos/feb00/articles/synthsecrets.htm.

[10] G. Reid, "Synthesizer Secrets," Sound on Sound, January 2004. [Online]. Available: http://www.soundonsound.com/sos/jan04/articles/synthsecrets.htm.

[11] Test Tone, "What is Bitcrusher Effect," [Online]. Available: http://testtone.com/fundamentals/what-bitcrusher-effect.

[12] MIDI Manufacturers Association, "MIDI Messages," [Online]. Available: www.midi.org/techspecs/midimessages.php.

[13] All About Circuits, "Look-up Tables," All About Circuits, 2015. [Online]. Available: http://www.allaboutcircuits.com/vol_4/chpt_16/2.html.

[14] N. Wooster, "FPGA-Synth," August 2010. [Online]. Available: http://woosteraudio.com/fpga-synth.html.

[15] R. Sassinger, "Cyclone 5 Music Synthesizer," [Online]. Available: http://fpgasynth.beepworld.de/cyclone.htm.

[16] J. Wolfe, "Note names, MIDI numbers and frequencies," June 2005. [Online]. Available: http://newt.phys.unsw.edu.au/jw/notes.html.

[17] Burk, Polanski, Repetto, Roberts and Rockmore, "Timbre," Columbia University, [Online]. Available: http://music.columbia.edu/cmc/musicandcomputers/chapter1/01_04.php.

[18] E. W. Weisstein, "Fourier Series- Sawtooth Wave," 2 April 2015. [Online]. Available: http://mathworld.wolfram.com/FourierSeriesSawtoothWave.html.

[19] E. W. Weisstein, "Fourier Series- Square Wave," 2 April 2015. [Online]. Available: http://mathworld.wolfram.com/FourierSeriesSquareWave.html.

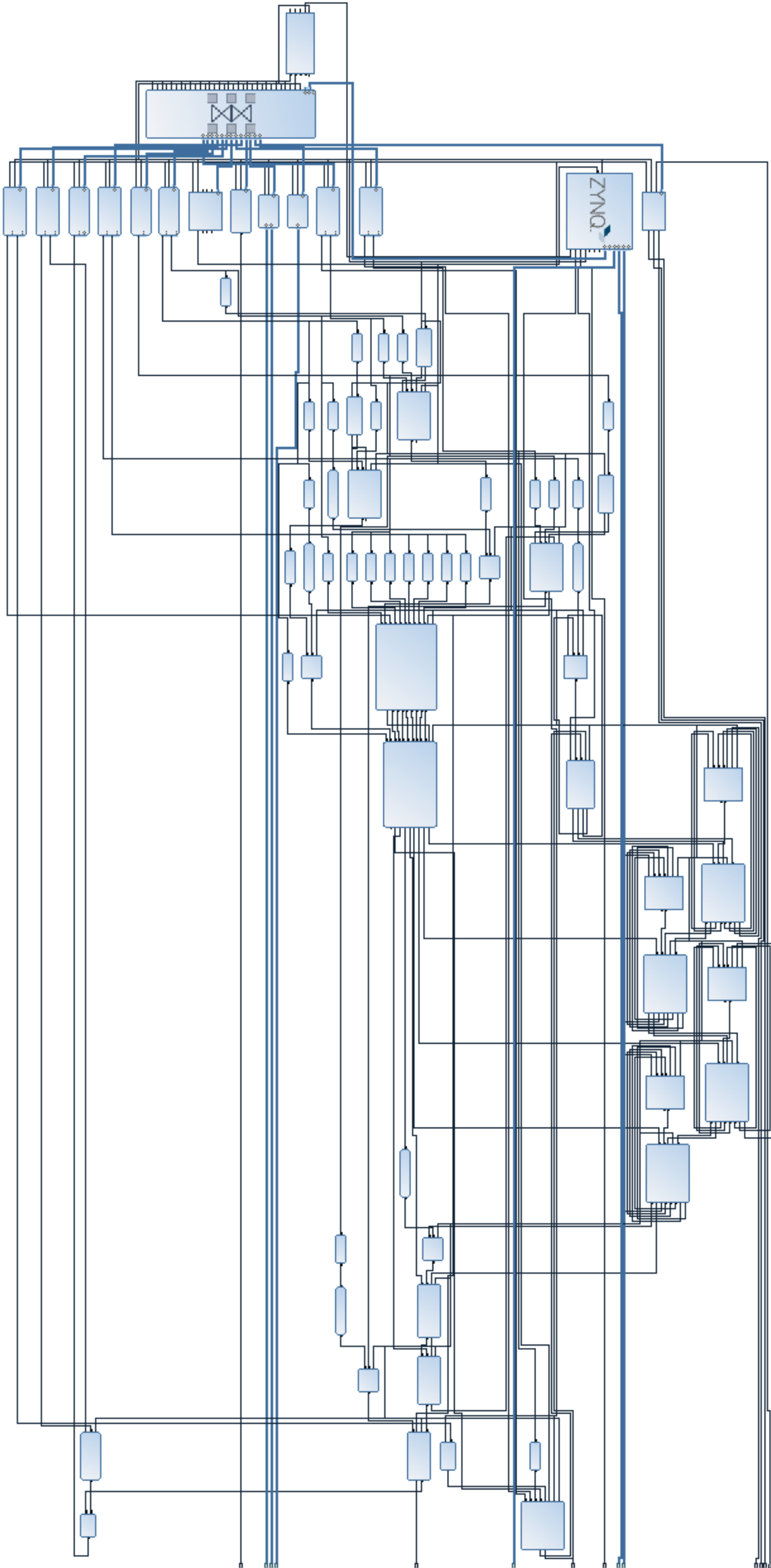
[20] E. W. Weisstein, "Fourier Series-Triangle Wave," 2 April 2015. [Online]. Available: http://mathworld.wolfram.com/FourierSeriesTriangleWave.html.

[21] L. H. Crocket, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, "The Zynq Book," July 2014. [Online]. Available: http://mediagoblin.tami.org.il/mgoblin_media/media_entries/660/The_Zynq_Book_ebook.pdf.

[22] C. Stroud, "Linear Feedback Shift Registers," October 2004. [Online]. Available: http://www.eng.auburn.edu/~strouce/class/elec6250/LFSRs.pdf.

[23] A. Askenfelt and E. Jansson, "Royal Swedish Academy of Music," [Online]. Available: http://www.speech.kth.se/music/5_lectures/askenflt/measure.html.

Appendix A: Xilinx Vivado Block Diagram



Appendix B: Development Code

MATLAB Scripts

Direct Digital Synthesis Waveform Modules Generation

```
%Sidney Veilleux
%Evan Briggs
%MQP Digital Synthesizer

function hardware_nco_fast_phase(clk,samples)

%This function creates q-sample wavetables for synthesis



close all
t=0:1:(samples-1);
sample_bits = ceil(log(samples)/log(2));
sample_msb = sample_bits -1;
%% Sine Wave

x(1,:)=sin(2*pi*t./samples);
x(1,:)=round(x.*2^22);

%% Sawtooth Wave

x(2,:) = t*2^(23-log2(samples))-2^22;  %% 15 for 256 samples, 13 for 1024

%% Square Wave

x(3,1:(samples/2)) = 2^22;
x(3,(samples/2+1):samples) = -(2^22);

%% Triangle Wave

x(4,1:samples/2) = t(1:samples/2)*(2^(23-(log2(samples)))-1);    %% 15 for
256 samples, 13 for 1024
x(4,(samples/2+1):samples) = (2^22-t(1:(samples/2))*(2^(23-(log2(samples)))-
1));   %% 15 for 256 samples, 13 for 1024
x(4,:) = 2*x(4,:) - 2^22;


x = round(x/4);     %scale down by four
%% Plots
   figure;
   plot(t,x(1,:),'.');
   title('Sinusoid');

   figure;
   plot(t,x(2,:),'.');
   title('Sawtooth');
```

```
    figure;
    plot(t,x(3,:),'.');
    title('Square');

    figure;
    plot(t,x(4,:),'.');
    title('Triangle');

%% Counter Module
sample_bits = ceil(log(samples)/log(2));
nco_sine = fopen('hardware_nco_sine_fast_phase.v','w');
fprintf(nco_sine, '`timescale 1ns / 1ps\n\n');
fprintf(nco_sine, '//MATLAB Generated Verilog\n');
fprintf(nco_sine, '//Sidney Veilleux and Evan Briggs\n');
fprintf(nco_sine, '//Worcester Polytechnic Institute 2015\n\n\n');


fprintf(nco_sine, 'module hardware_nco_sine_fast_phase(\n');
fprintf(nco_sine, '    input clk_100M,\n');
fprintf(nco_sine, '    input clk_200M,\n');
fprintf(nco_sine, '    input clk_96k,\n');
fprintf(nco_sine, '    input sample_interrupt,\n');
fprintf(nco_sine, '    input [3:0] wave_select,\n');
fprintf(nco_sine, '    input [5:0] step_size,\n'); %up to 16 step size
fprintf(nco_sine, '    input [7:0] phase,\n');
fprintf(nco_sine, '    output output_enable,\n');
fprintf(nco_sine, '    output[23:0] sample\n);\n\n');

fprintf(nco_sine, 'reg signed[23:0] sine_sample;\n');
fprintf(nco_sine, 'reg[23:0] syncronous_sample;\n');
fprintf(nco_sine, 'wire[23:0] unscaled_sample;\n');
fprintf(nco_sine, 'wire[2:0] active_waveforms;\n');
fprintf(nco_sine, 'reg [%i:0] sample_count;\n',sample_msb);
fprintf(nco_sine, 'wire [%i:0] sample_count_phased;\n\n',sample_msb);
fprintf(nco_sine, 'assign sample_count_phased  = sample_count + {phase[6:0],
%i''b0};\n',(sample_msb-6));

fprintf(nco_sine, '\n//Create 1ns pulse from 96kHz clock\n');
fprintf(nco_sine, 'reg clk_96k_q;\n');
fprintf(nco_sine, 'reg clk_96k_qq;\n');
fprintf(nco_sine, 'wire clk_96_pulse;\n');
fprintf(nco_sine, 'always @(posedge clk_100M)\n');
fprintf(nco_sine, '        begin\n');
fprintf(nco_sine, '            clk_96k_q <= clk_96k;\n');
fprintf(nco_sine, '            clk_96k_qq <= clk_96k_q;\n');
fprintf(nco_sine, '        end\n');
fprintf(nco_sine, 'assign clk_96_pulse = (clk_96k_q && ~(clk_96k_qq));\n\n');

fprintf(nco_sine, '\n//Create 1ns pulse from interrupt signal\n');
fprintf(nco_sine, 'reg interrupt_q;\n');
fprintf(nco_sine, 'reg interrupt_qq;\n');
fprintf(nco_sine, 'wire interrupt_pulse;\n');
fprintf(nco_sine, 'always @(posedge clk_100M)\n');
fprintf(nco_sine, '        begin\n');
fprintf(nco_sine, '            interrupt_q <= sample_interrupt;\n');
fprintf(nco_sine, '            interrupt_qq <= interrupt_q;\n');
fprintf(nco_sine, '        end\n');
```

```
fprintf(nco_sine, 'assign interrupt_pulse = (interrupt_q &&
~(interrupt_qq));\n\n');

fprintf(nco_sine, 'always @(posedge clk_100M)\n');
fprintf(nco_sine, '    if(interrupt_pulse)\n');
fprintf(nco_sine, '        sample_count = sample_count + step_size;\n');
fprintf(nco_sine, '    else\n');
fprintf(nco_sine, '        sample_count = sample_count;\n\n');


fprintf(nco_sine, 'always @(posedge clk_200M)\n');
fprintf(nco_sine, '    case(sample_count_phased)\n');
for k=1:1:samples
fprintf(nco_sine, '        %i:    sine_sample = %i;\n',(k-1),x(1,k));
end
fprintf(nco_sine, '        default:    sine_sample = 24''b0;\n');
fprintf(nco_sine, '    endcase\n\n');



fprintf(nco_sine, 'always @(posedge clk_100M)\n');
fprintf(nco_sine, '    if(clk_96_pulse)\n');
fprintf(nco_sine, '        syncronous_sample = sine_sample;\n');
fprintf(nco_sine, '    else\n');
fprintf(nco_sine, '        syncronous_sample = syncronous_sample;\n\n');

fprintf(nco_sine, 'assign unscaled_sample = (wave_select[0]) ?
syncronous_sample : 24''b0;\n');

fprintf(nco_sine, 'assign active_waveforms = wave_select[0] + wave_select[1]
+ wave_select[2] + wave_select[3];\n');
fprintf(nco_sine, 'assign sample = (active_waveforms == 1) ?
{unscaled_sample[23], unscaled_sample[20:0], 2''b0} : \n');
fprintf(nco_sine, '(active_waveforms == 2) ? {unscaled_sample[23:22],
unscaled_sample[20:0], 1''b0} : \n');
fprintf(nco_sine, '(active_waveforms == 3) ? {unscaled_sample[23:22],
unscaled_sample[20:0], 1''b0} : \n');
fprintf(nco_sine, '(active_waveforms == 4) ? unscaled_sample : \n');
fprintf(nco_sine, '24''b0;\n\n');

fprintf(nco_sine, 'assign output_enable = clk_96_pulse;\n');
fprintf(nco_sine, 'endmodule\n');


fclose(nco_sine);

%% Sawtooth
nco_saw = fopen('hardware_nco_saw_fast_phase.v','w');
fprintf(nco_saw, '`timescale 1ns / 1ps\n\n');
fprintf(nco_saw, '//MATLAB Generated Verilog\n');
fprintf(nco_saw, '//Sidney Veilleux and Evan Briggs\n');
fprintf(nco_saw, '//Worcester Polytechnic Institute 2015\n\n\n');


fprintf(nco_saw, 'module hardware_nco_saw_fast_phase(\n');
fprintf(nco_saw, '    input clk_100M,\n');
fprintf(nco_saw, '    input clk_200M,\n');
```

```
fprintf(nco_saw, '    input clk_96k,\n');
fprintf(nco_saw, '    input sample_interrupt,\n');
fprintf(nco_saw, '    input [3:0] wave_select,\n');
fprintf(nco_saw, '    input [5:0] step_size,\n'); %up to 16 step size
fprintf(nco_saw, '    input [7:0] phase,\n');
fprintf(nco_saw, '    output output_enable,\n');
fprintf(nco_saw, '    output[23:0] sample\n);\n\n');

fprintf(nco_saw, 'reg signed[23:0] saw_sample;\n');
fprintf(nco_saw, 'reg[23:0] syncronous_sample;\n');
fprintf(nco_saw, 'wire[23:0] unscaled_sample;\n');
fprintf(nco_saw, 'wire[2:0] active_waveforms;\n');
fprintf(nco_saw, 'reg [%i:0] sample_count;\n',sample_msb);
fprintf(nco_saw, 'wire [%i:0] sample_count_phased;\n\n',sample_msb);
fprintf(nco_saw, 'assign sample_count_phased  = sample_count + {phase[6:0],
%i''b0};\n',(sample_msb-6));

fprintf(nco_saw, '\n//Create 1ns pulse from 96kHz clock\n');
fprintf(nco_saw, 'reg clk_96k_q;\n');
fprintf(nco_saw, 'reg clk_96k_qq;\n');
fprintf(nco_saw, 'wire clk_96_pulse;\n');
fprintf(nco_saw, 'always @(posedge clk_100M)\n');
fprintf(nco_saw, '        begin\n');
fprintf(nco_saw, '            clk_96k_q <= clk_96k;\n');
fprintf(nco_saw, '            clk_96k_qq <= clk_96k_q;\n');
fprintf(nco_saw, '        end\n');
fprintf(nco_saw, 'assign clk_96_pulse = (clk_96k_q && ~(clk_96k_qq));\n\n');

fprintf(nco_saw, '\n//Create 1ns pulse from interrupt signal\n');
fprintf(nco_saw, 'reg interrupt_q;\n');
fprintf(nco_saw, 'reg interrupt_qq;\n');
fprintf(nco_saw, 'wire interrupt_pulse;\n');
fprintf(nco_saw, 'always @(posedge clk_100M)\n');
fprintf(nco_saw, '        begin\n');
fprintf(nco_saw, '            interrupt_q <= sample_interrupt;\n');
fprintf(nco_saw, '            interrupt_qq <= interrupt_q;\n');
fprintf(nco_saw, '        end\n');
fprintf(nco_saw, 'assign interrupt_pulse = (interrupt_q &&
~(interrupt_qq));\n\n');

fprintf(nco_saw, 'always @(posedge clk_100M)\n');
fprintf(nco_saw, '    if(interrupt_pulse)\n');
fprintf(nco_saw, '        sample_count = sample_count + step_size;\n');
fprintf(nco_saw, '    else\n');
fprintf(nco_saw, '        sample_count = sample_count;\n\n');


fprintf(nco_saw, 'always @(posedge clk_200M)\n');
fprintf(nco_saw, '    case(sample_count_phased)\n');
for k=1:1:samples
fprintf(nco_saw, '        %i:    saw_sample = %i;\n',(k-1),x(2,k));
end
fprintf(nco_saw, '        default:    saw_sample = 24''b0;\n');
fprintf(nco_saw, '    endcase\n\n');
```

```
fprintf(nco_saw, 'always @(posedge clk_100M)\n');
fprintf(nco_saw, '    if(clk_96_pulse)\n');
fprintf(nco_saw, '        syncronous_sample = saw_sample;\n');
fprintf(nco_saw, '    else\n');
fprintf(nco_saw, '        syncronous_sample = syncronous_sample;\n\n');

fprintf(nco_saw, 'assign unscaled_sample = (wave_select[1]) ?
syncronous_sample : 24''b0;\n');

fprintf(nco_saw, 'assign active_waveforms = wave_select[0] + wave_select[1] +
wave_select[2] + wave_select[3];\n');
fprintf(nco_saw, 'assign sample = (active_waveforms == 1) ?
{unscaled_sample[23], unscaled_sample[20:0], 2''b0} : \n');
fprintf(nco_saw, '(active_waveforms == 2) ? {unscaled_sample[23:22],
unscaled_sample[20:0], 1''b0} : \n');
fprintf(nco_saw, '(active_waveforms == 3) ? {unscaled_sample[23:22],
unscaled_sample[20:0], 1''b0} : \n');
fprintf(nco_saw, '(active_waveforms == 4) ? unscaled_sample : \n');
fprintf(nco_saw, '24''b0;\n\n');

fprintf(nco_saw, 'assign output_enable = clk_96_pulse;\n');
fprintf(nco_saw, 'endmodule\n');


fclose(nco_saw);

%% Square
nco_square = fopen('hardware_nco_square_fast_phase.v','w');
fprintf(nco_square, '`timescale 1ns / 1ps\n\n');
fprintf(nco_square, '//MATLAB Generated Verilog\n');
fprintf(nco_square, '//Sidney Veilleux and Evan Briggs\n');
fprintf(nco_square, '//Worcester Polytechnic Institute 2015\n\n\n');


fprintf(nco_square, 'module hardware_nco_square_fast_phase(\n');
fprintf(nco_square, '    input clk_100M,\n');
fprintf(nco_square, '    input clk_200M,\n');
fprintf(nco_square, '    input clk_96k,\n');
fprintf(nco_square, '    input [7:0] pwm_control,\n');
fprintf(nco_square, '    input sample_interrupt,\n');
fprintf(nco_square, '    input [3:0] wave_select,\n');
fprintf(nco_square, '    input [5:0] step_size,\n'); %up to 16 step size
fprintf(nco_square, '    input [7:0] phase,\n');
fprintf(nco_square, '    output output_enable,\n');
fprintf(nco_square, '    output[23:0] sample\n);\n\n');

fprintf(nco_square, 'reg signed[23:0] square_sample;\n');
fprintf(nco_square, 'reg[23:0] syncronous_sample;\n');
fprintf(nco_square, 'wire[23:0] unscaled_sample;\n');
fprintf(nco_square, 'wire[2:0] active_waveforms;\n');
fprintf(nco_square, 'reg [%i:0] sample_count;\n',sample_msb);
fprintf(nco_square, 'wire [%i:0] sample_count_phased;\n\n',sample_msb);
fprintf(nco_square, 'assign sample_count_phased  = sample_count +
{phase[6:0], %i''b0};\n',(sample_msb-6));


fprintf(nco_square, '\n//Create 1ns pulse from 96kHz clock\n');
```

```
fprintf(nco_square, 'reg clk_96k_q;\n');
fprintf(nco_square, 'reg clk_96k_qq;\n');
fprintf(nco_square, 'wire clk_96_pulse;\n');
fprintf(nco_square, 'always @(posedge clk_100M)\n');
fprintf(nco_square, '        begin\n');
fprintf(nco_square, '            clk_96k_q <= clk_96k;\n');
fprintf(nco_square, '            clk_96k_qq <= clk_96k_q;\n');
fprintf(nco_square, '        end\n');
fprintf(nco_square, 'assign clk_96_pulse = (clk_96k_q &&
~(clk_96k_qq));\n\n');

fprintf(nco_square, '\n//Create 1ns pulse from interrupt signal\n');
fprintf(nco_square, 'reg interrupt_q;\n');
fprintf(nco_square, 'reg interrupt_qq;\n');
fprintf(nco_square, 'wire interrupt_pulse;\n');
fprintf(nco_square, 'always @(posedge clk_100M)\n');
fprintf(nco_square, '        begin\n');
fprintf(nco_square, '            interrupt_q <= sample_interrupt;\n');
fprintf(nco_square, '            interrupt_qq <= interrupt_q;\n');
fprintf(nco_square, '        end\n');
fprintf(nco_square, 'assign interrupt_pulse = (interrupt_q &&
~(interrupt_qq));\n\n');

fprintf(nco_square, 'always @(posedge clk_100M)\n');
fprintf(nco_square, '    if(interrupt_pulse)\n');
fprintf(nco_square, '        sample_count = sample_count + step_size;\n');
fprintf(nco_square, '    else\n');
fprintf(nco_square, '        sample_count = sample_count;\n\n');


fprintf(nco_square, 'always @(posedge clk_200M)\n');
fprintf(nco_square, '    if(sample_count[%i:5] < pwm_control)\n',sample_msb);
fprintf(nco_square, '        square_sample = 4194304;\n');
fprintf(nco_square, '    else    \n');
fprintf(nco_square, '        square_sample = -4194304;\n\n');



fprintf(nco_square, 'always @(posedge clk_100M)\n');
fprintf(nco_square, '    if(clk_96_pulse)\n');
fprintf(nco_square, '        syncronous_sample = square_sample;\n');
fprintf(nco_square, '    else\n');
fprintf(nco_square, '        syncronous_sample = syncronous_sample;\n\n');

fprintf(nco_square, 'assign unscaled_sample = (wave_select[2]) ?
syncronous_sample : 24''b0;\n');

fprintf(nco_square, 'assign active_waveforms = wave_select[0] +
wave_select[1] + wave_select[2] + wave_select[3];\n');
fprintf(nco_square, 'assign sample = (active_waveforms == 1) ?
{unscaled_sample[23], unscaled_sample[20:0], 2''b0} : \n');
fprintf(nco_square, '(active_waveforms == 2) ? {unscaled_sample[23:22],
unscaled_sample[20:0], 1''b0} : \n');
fprintf(nco_square, '(active_waveforms == 3) ? {unscaled_sample[23:22],
unscaled_sample[20:0], 1''b0} : \n');
fprintf(nco_square, '(active_waveforms == 4) ? unscaled_sample : \n');
fprintf(nco_square, '24''b0;\n\n');
```

```
fprintf(nco_square, 'assign output_enable = clk_96_pulse;\n');
fprintf(nco_square, 'endmodule\n');


fclose(nco_square);



%% Triangle

nco_triangle = fopen('hardware_nco_triangle_fast_phase.v','w');
fprintf(nco_triangle, '`timescale 1ns / 1ps\n\n');
fprintf(nco_triangle, '//MATLAB Generated Verilog\n');
fprintf(nco_triangle, '//Sidney Veilleux and Evan Briggs\n');
fprintf(nco_triangle, '//Worcester Polytechnic Institute 2015\n\n\n');


fprintf(nco_triangle, 'module hardware_nco_triangle_fast_phase(\n');
fprintf(nco_triangle, '    input clk_100M,\n');
fprintf(nco_triangle, '    input clk_200M,\n');
fprintf(nco_triangle, '    input clk_96k,\n');
fprintf(nco_triangle, '    input sample_interrupt,\n');
fprintf(nco_triangle, '    input [3:0] wave_select,\n');
fprintf(nco_triangle, '    input [5:0] step_size,\n'); %up to 16 step size
fprintf(nco_triangle, '    input [7:0] phase,\n');
fprintf(nco_triangle, '    output output_enable,\n');
fprintf(nco_triangle, '    output[23:0] sample\n);\n\n');

fprintf(nco_triangle, 'reg signed[23:0] triangle_sample;\n');
fprintf(nco_triangle, 'reg[23:0] syncronous_sample;\n');
fprintf(nco_triangle, 'wire[23:0] unscaled_sample;\n');
fprintf(nco_triangle, 'wire[2:0] active_waveforms;\n');
fprintf(nco_triangle, 'reg [%i:0] sample_count;\n',sample_msb);
fprintf(nco_triangle, 'wire [%i:0] sample_count_phased;\n\n',sample_msb);
fprintf(nco_triangle, 'assign sample_count_phased  = sample_count +
{phase[6:0], %i''b0};\n',(sample_msb-6));


fprintf(nco_triangle, '\n//Create 1ns pulse from 96kHz clock\n');
fprintf(nco_triangle, 'reg clk_96k_q;\n');
fprintf(nco_triangle, 'reg clk_96k_qq;\n');
fprintf(nco_triangle, 'wire clk_96_pulse;\n');
fprintf(nco_triangle, 'always @(posedge clk_100M)\n');
fprintf(nco_triangle, '        begin\n');
fprintf(nco_triangle, '            clk_96k_q <= clk_96k;\n');
fprintf(nco_triangle, '            clk_96k_qq <= clk_96k_q;\n');
fprintf(nco_triangle, '        end\n');
fprintf(nco_triangle, 'assign clk_96_pulse = (clk_96k_q &&
~(clk_96k_qq));\n\n');

fprintf(nco_triangle, '\n//Create 1ns pulse from interrupt signal\n');
fprintf(nco_triangle, 'reg interrupt_q;\n');
fprintf(nco_triangle, 'reg interrupt_qq;\n');
fprintf(nco_triangle, 'wire interrupt_pulse;\n');
fprintf(nco_triangle, 'always @(posedge clk_100M)\n');
fprintf(nco_triangle, '        begin\n');
```

```
fprintf(nco_triangle, '            interrupt_q <= sample_interrupt;\n');
fprintf(nco_triangle, '            interrupt_qq <= interrupt_q;\n');
fprintf(nco_triangle, '        end\n');
fprintf(nco_triangle, 'assign interrupt_pulse = (interrupt_q &&
~(interrupt_qq));\n\n');

fprintf(nco_triangle, 'always @(posedge clk_100M)\n');
fprintf(nco_triangle, '    if(interrupt_pulse)\n');
fprintf(nco_triangle, '        sample_count = sample_count + step_size;\n');
fprintf(nco_triangle, '    else\n');
fprintf(nco_triangle, '        sample_count = sample_count;\n\n');


fprintf(nco_triangle, 'always @(posedge clk_200M)\n');
fprintf(nco_triangle, '    case(sample_count_phased)\n');
for k=1:1:samples
fprintf(nco_triangle, '        %i:    triangle_sample = %i;\n',(k-1),x(4,k));
end
fprintf(nco_triangle, '        default:    triangle_sample = 24''b0;\n');
fprintf(nco_triangle, '    endcase\n\n');



fprintf(nco_triangle, 'always @(posedge clk_100M)\n');
fprintf(nco_triangle, '    if(clk_96_pulse)\n');
fprintf(nco_triangle, '        syncronous_sample = triangle_sample;\n');
fprintf(nco_triangle, '    else\n');
fprintf(nco_triangle, '        syncronous_sample = syncronous_sample;\n\n');

fprintf(nco_triangle, 'assign unscaled_sample = (wave_select[3]) ?
syncronous_sample : 24''b0;\n');
fprintf(nco_triangle, 'assign active_waveforms = wave_select[0] +
wave_select[1] + wave_select[2] + wave_select[3];\n');
fprintf(nco_triangle, 'assign sample = (active_waveforms == 1) ?
{unscaled_sample[23], unscaled_sample[20:0], 2''b0} : \n');
fprintf(nco_triangle, '(active_waveforms == 2) ? {unscaled_sample[23:22],
unscaled_sample[20:0], 1''b0} : \n');
fprintf(nco_triangle, '(active_waveforms == 3) ? {unscaled_sample[23:22],
unscaled_sample[20:0], 1''b0} : \n');
fprintf(nco_triangle, '(active_waveforms == 4) ? unscaled_sample : \n');
fprintf(nco_triangle, '24''b0;\n\n');

fprintf(nco_triangle, 'assign output_enable = clk_96_pulse;\n');
fprintf(nco_triangle, 'endmodule\n');


fclose(nco_triangle);
```

FIR Filter Module Generation

```
%Sidney Veilleux
%Evan Briggs
%FPGA Synthesizer MQP

function VerilogFirFilter3()

fileID = fopen('verilog_fir_filter3.v','w');

M=64;
state_size = ceil(log(M+1)/log(2));
state_bits = state_size + 1;
msb = state_size + 2;
max_state = ((M+2)*4)+ 3;

fprintf(fileID, '`timescale 1ns / 1ps\n\n');
fprintf(fileID, '//MATLAB Generated Verilog\n');
fprintf(fileID, '//Sidney Veilleux and Evan Briggs\n');
fprintf(fileID, '//Worcester Polytechnic Institute 2015\n\n\n');

%% Inputs/Outputs
fprintf(fileID, 'module adjustable_fir_filter(\n');
fprintf(fileID, '    input [23:0] sample_in,\n');
fprintf(fileID, '    input clk_100M,\n');
fprintf(fileID, '    input clk_96k,\n');
fprintf(fileID, '    input[47:0] accumulate_in,\n');
fprintf(fileID, '    input[7:0] controller_in,\n');
fprintf(fileID, '    output clock_enable,\n');
fprintf(fileID, '    output sync_clear,\n');
fprintf(fileID, '    output reg[23:0] multiply_x,\n');
fprintf(fileID, '    output reg[15:0] multiply_b,\n');
fprintf(fileID, '    output reg[47:0] accumulate_out,\n');
fprintf(fileID, '    output subtract,\n');
fprintf(fileID, '    output output_enable,\n');
fprintf(fileID, '    output reg[23:0] filtered_out\n);\n\n');

fprintf(fileID, 'reg[47:0] accumulate_in_temp;\n');
fprintf(fileID, 'assign sync_clear = 1''b0;\n\n');

%% Clock Divisions
fprintf(fileID, '//Create 1ns pulse from 96kHz clock\n');
fprintf(fileID, 'reg clk_96k_q;\n');
fprintf(fileID, 'reg clk_96k_qq;\n');
fprintf(fileID, 'wire clk_96_pulse;\n');
fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '        begin\n');
fprintf(fileID, '            clk_96k_q <= clk_96k;\n');
fprintf(fileID, '            clk_96k_qq <= clk_96k_q;\n');
fprintf(fileID, '        end\n');
fprintf(fileID, 'assign clk_96_pulse = (clk_96k_q && ~(clk_96k_qq));\n\n');

fprintf(fileID, 'assign subtract = 1''b0;\n');
fprintf(fileID, 'assign clock_enable = (current_state[%i:2] == %i''b0) ?
1''b0 : (current_state[%i:2] <= %i) ? 1''b1 : 1''b0;\n',msb, state_bits, msb,
M+1);
```

```
fprintf(fileID, '//%i Order Fir Filter\n',M);

%% Coefs
fprintf(fileID, '//Coefficient Registers\n');
for k=0:1:M
    fprintf(fileID, 'wire signed [15:0] coef_%i;\n',k);
end
fprintf(fileID, 'coef_switch coef_switch_i(\n');
fprintf(fileID, '    .controller_value(controller_in),\n');
for k=0:1:M-1
    fprintf(fileID, '    .coef_%i(coef_%i),\n',k, k);
end
    fprintf(fileID, '    .coef_%i(coef_%i));\n\n',M,M);

%% Previous Input
fprintf(fileID, '//Previous Input Registers\n');
for k=1:1:M
    fprintf(fileID, 'reg[23:0] x_delayed_%i;\n',k);
end
fprintf(fileID, '\n\n');

%% Next State Logic
fprintf(fileID, 'reg[%i:0] current_state;\n', msb);
fprintf(fileID, 'wire[%i:0] next_state;\n', msb);

fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '    current_state = next_state;\n\n');


fprintf(fileID, '//Next state logic \n');

fprintf(fileID, 'assign next_state = ((current_state == 0) && (clk_96_pulse
== 1''b1))? 1:\n');
fprintf(fileID, '    ((current_state == 0) && (clk_96_pulse == 1''b0)) ?
0:\n');
fprintf(fileID, '    (current_state < %i) ? (current_state + 1''b1):\n',
max_state);
fprintf(fileID, '    0;\n\n');


%% Shift Delay
fprintf(fileID, '//Shift delayed inputs with every new sample \n');

fprintf(fileID, 'always @ (posedge clk_100M)\n');
fprintf(fileID, '    if(current_state == %i)\n', max_state);
fprintf(fileID, '        begin\n');
fprintf(fileID, '            x_delayed_1 <= sample_in;\n');
for k=2:1:M
    fprintf(fileID, '            x_delayed_%i <= x_delayed_%i;\n', k, k-1);
end
fprintf(fileID, '        end\n\n');

%% Multiply
fprintf(fileID, '//Multiply Logic\n');

fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '    if(current_state[%i:2] == 0)\n', msb);
```

```
fprintf(fileID, '        begin\n');
fprintf(fileID, '            multiply_x = 24''b0;\n');
fprintf(fileID, '            multiply_b = 16''b0;\n');
fprintf(fileID, '        end\n');
fprintf(fileID, '    else if(current_state[%i:2] == 1)\n', msb);
fprintf(fileID, '        begin\n');
fprintf(fileID, '            multiply_x = sample_in;\n');
fprintf(fileID, '            multiply_b = coef_0;\n');
fprintf(fileID, '        end\n');
for k=2:1:M+1
    fprintf(fileID, '    else if(current_state[%i:2] == %i)\n', msb, k);
    fprintf(fileID, '        begin\n');
    fprintf(fileID, '            multiply_x = x_delayed_%i;\n', k-1);
    fprintf(fileID, '            multiply_b = coef_%i;\n', k-1);
    fprintf(fileID, '        end\n\n');
end
fprintf(fileID, '    else\n');
fprintf(fileID, '        begin\n');
fprintf(fileID, '            multiply_x = 24''b0;\n');
fprintf(fileID, '            multiply_b = 16''b0;\n');
fprintf(fileID, '        end\n\n');

%% Accumulate
fprintf(fileID, '//Accumulate Logic \n');
fprintf(fileID, 'always @ (posedge clk_100M)\n');
fprintf(fileID, '    if(current_state <= 4)\n');
fprintf(fileID, '        accumulate_in_temp = 48''b0;\n');
fprintf(fileID, '    else if (current_state[1:0] == 2''b00)\n');
fprintf(fileID, '        accumulate_in_temp = accumulate_in;\n');
fprintf(fileID, '    else\n');
fprintf(fileID, '        accumulate_in_temp = accumulate_in_temp;\n\n');

fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '    if(current_state == 0)\n');
fprintf(fileID, '        accumulate_out = 48''b0;\n');
fprintf(fileID, '    else if((current_state[%i:2] != %i) &&
(current_state[1:0] == 2''b01))\n',msb, M+2);
fprintf(fileID, '        accumulate_out = accumulate_in_temp;\n');
fprintf(fileID, '    else if(current_state[%i:2] != %i)\n', msb, M+2);
fprintf(fileID, '        accumulate_out = accumulate_out;\n');
fprintf(fileID, '    else\n');
fprintf(fileID, '        accumulate_out = 48''b0;\n\n');

%% Output
fprintf(fileID, '//Output Logic \n');
fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '    if(current_state == %i)\n', max_state-3);
fprintf(fileID, '        filtered_out = {accumulate_in[47],
accumulate_in[37:15]};\n');
fprintf(fileID, '    else\n');
fprintf(fileID, '        filtered_out = filtered_out;\n\n');

fprintf(fileID, 'assign output_enable = (current_state == %i) ? 1''b1 :
1''b0;\n\n',max_state-3);

fprintf(fileID, 'endmodule\n');
```

```
fclose(fileID);

end
```

IIR Filter Module Generation

```
%Sidney Veilleux
%Evan Briggs
%FPGA Synthesizer MQP

function VerilogIIRFilter_flush()

fileID = fopen('verilog_iir_filter_flush.v','w');
% 4th order IIR
% 5 b's, 4 a's
numB = 5;
numA = 4;
M= numA + numB;
state_size = ceil(log(M+1)/log(2));
state_bits = state_size + 1;
msb = state_size + 3;
max_state = ((M+1)*8)+ 7;   % ?????? M+2????????????

fprintf(fileID, '`timescale 1ns / 1ps\n\n');
fprintf(fileID, '//MATLAB Generated Verilog\n');
fprintf(fileID, '//Sidney Veilleux and Evan Briggs\n');
fprintf(fileID, '//Worcester Polytechnic Institute 2015\n\n\n');

%% Inputs/Outputs
fprintf(fileID, 'module adjustable_iir_filter(\n');
fprintf(fileID, '    input [23:0] sample_in,\n');
fprintf(fileID, '    input clk_100M,\n');
fprintf(fileID, '    input clk_96k,\n');
fprintf(fileID, '    input[63:0] accumulate_in,\n');
fprintf(fileID, '    input[7:0] controller_in,\n');
fprintf(fileID, '    output clock_enable,\n');
fprintf(fileID, '    output sync_clear,\n');
fprintf(fileID, '    output reg[23:0] multiply_x,\n');
fprintf(fileID, '    output reg[33:0] multiply_b,\n');
fprintf(fileID, '    output reg[63:0] accumulate_out,\n');
fprintf(fileID, '    output subtract,\n');
fprintf(fileID, '    output output_enable,\n');
fprintf(fileID, '    output[23:0] filter_output\n);\n\n');

fprintf(fileID, 'reg[63:0] accumulate_in_temp;\n');
fprintf(fileID, 'reg[7:0] controller_current;\n');
fprintf(fileID, 'reg reset;\n');

fprintf(fileID, 'reg[23:0] filtered_out;\n');

fprintf(fileID, 'assign sync_clear = 1''b0;\n\n');

%%Reset Flush Logic

fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '    if(controller_current != controller_in)\n');
fprintf(fileID, '        begin\n');
fprintf(fileID, '            controller_current <= controller_in;\n');
fprintf(fileID, '            reset <= 1''b1;\n');
fprintf(fileID, '        end\n');
fprintf(fileID, '    else\n');
```

```
fprintf(fileID, '        begin\n');
fprintf(fileID, '            controller_current <= controller_current;\n');
fprintf(fileID, '            reset <= 1''b0;\n');
fprintf(fileID, '        end\n');

%% Clock Divisions
fprintf(fileID, '//Create 1ns pulse from 96kHz clock\n');
fprintf(fileID, 'reg clk_96k_q;\n');
fprintf(fileID, 'reg clk_96k_qq;\n');
fprintf(fileID, 'wire clk_96_pulse;\n');
fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '        begin\n');
fprintf(fileID, '            clk_96k_q <= clk_96k;\n');
fprintf(fileID, '            clk_96k_qq <= clk_96k_q;\n');
fprintf(fileID, '        end\n');
fprintf(fileID, 'assign clk_96_pulse = (clk_96k_q && ~(clk_96k_qq));\n\n');

fprintf(fileID, 'assign subtract = ((current_state[%i:3] >=
%i)&&(current_state[%i:3] <= %i)) ? 1''b1 : 1''b0;\n', msb, 1, msb, numA);
fprintf(fileID, 'assign clock_enable = (current_state[%i:3] == %i''b0) ?
1''b0 : (current_state[%i:3] <= %i) ? 1''b1 : 1''b0;\n',msb, state_bits, msb,
M+1);

fprintf(fileID, '//%i th Order IIR Filter\n',numB);

%% Coefs
fprintf(fileID, '//Coefficient Registers\n');
for k=0:1:numB-1
    fprintf(fileID, 'wire signed [33:0] coef_b_%i;\n',k);
end
for k=1:1:numA
    fprintf(fileID, 'wire signed [33:0] coef_a_%i;\n',k);
end
fprintf(fileID, 'coef_switch coef_switch_i(\n');
fprintf(fileID, '    .controller_value(controller_in),\n');
for k=0:1:numB-1
    fprintf(fileID, '    .coef_b_%i(coef_b_%i),\n',k, k);
end
for k = 1:1:numA-1
    fprintf(fileID, '    .coef_a_%i(coef_a_%i),\n',k, k);
end

fprintf(fileID, '    .coef_a_%i(coef_a_%i));\n\n',numA,numA);

%% Previous Input/Output
fprintf(fileID, '//Previous Input Registers\n');
for k=1:1:numB
    fprintf(fileID, 'reg[23:0] x_delayed_%i;\n',k);
end
fprintf(fileID, '\n');

fprintf(fileID, '//Previous Output Registers\n');
for k=1:1:numA
    fprintf(fileID, 'reg[23:0] y_delayed_%i;\n',k);
end
fprintf(fileID, '\n\n');
```

```
%% Next State Logic
fprintf(fileID, 'reg[%i:0] current_state;\n', msb);
fprintf(fileID, 'wire[%i:0] next_state;\n', msb);

fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '    current_state = next_state;\n\n');


fprintf(fileID, '//Next state logic \n');

fprintf(fileID, 'assign next_state = ((current_state == 0) && (clk_96_pulse
== 1''b1))? 1:\n');
fprintf(fileID, '    ((current_state == 0) && (clk_96_pulse == 1''b0)) ?
0:\n');
fprintf(fileID, '    (current_state < %i) ? (current_state + 1''b1):\n',
max_state);
fprintf(fileID, '    0;\n\n');


%% Shift Delay
fprintf(fileID, '//Shift delayed inputs with every new sample \n');

fprintf(fileID, 'always @ (posedge clk_100M)\n');
fprintf(fileID, '    if(reset)');
fprintf(fileID, '        begin\n');
fprintf(fileID, '            x_delayed_1 <= 24''b0;\n');
for k=2:1:numB
    fprintf(fileID, '            x_delayed_%i <= 24''b0;\n', k);
end
fprintf(fileID, '        end\n\n');

fprintf(fileID, '    else if((current_state == %i) && (controller_in
!=0))\n', max_state);
fprintf(fileID, '        begin\n');
fprintf(fileID, '            x_delayed_1 <= sample_in;\n');
for k=2:1:numB
    fprintf(fileID, '            x_delayed_%i <= x_delayed_%i;\n', k, k-1);
end
fprintf(fileID, '        end\n\n');

fprintf(fileID, '    else if(controller_in == 0)\n');
fprintf(fileID, '        begin\n');
fprintf(fileID, '            x_delayed_1 <= 24''b0;\n');
for k=2:1:numB
    fprintf(fileID, '            x_delayed_%i <= 24''b0;\n', k);
end
fprintf(fileID, '        end\n\n');

fprintf(fileID, '    else\n');
fprintf(fileID, '        begin\n');
for k=1:1:numB
    fprintf(fileID, '            x_delayed_%i <= x_delayed_%i;\n',k,k);
end
fprintf(fileID, '        end\n\n');


fprintf(fileID, 'always @ (posedge clk_100M)\n');
```

```
fprintf(fileID, '    if(reset)');
fprintf(fileID, '        begin\n');
fprintf(fileID, '            y_delayed_1 <= 24''b0;\n');
for k=2:1:numA
    fprintf(fileID, '            y_delayed_%i <= 24''b0;\n', k);
end
fprintf(fileID, '        end\n\n');
fprintf(fileID, '    else if((current_state == %i) && (controller_in !=
0))\n', max_state);
fprintf(fileID, '        begin\n');
fprintf(fileID, '            y_delayed_1 <= filtered_out;\n');
for k=2:1:numA
    fprintf(fileID, '            y_delayed_%i <= y_delayed_%i;\n', k, k-1);
end
fprintf(fileID, '        end\n\n');

fprintf(fileID, '    else if(controller_in == 0)\n');
fprintf(fileID, '        begin\n');
fprintf(fileID, '            y_delayed_1 <= 24''b0;\n');
for k=2:1:numA
    fprintf(fileID, '            y_delayed_%i <= 24''b0;\n', k);
end
fprintf(fileID, '        end\n\n');

fprintf(fileID, '    else\n');
fprintf(fileID, '        begin\n');
for k=1:1:numA
    fprintf(fileID, '            y_delayed_%i <= y_delayed_%i;\n',k,k);
end
fprintf(fileID, '        end\n\n');


%% Multiply
fprintf(fileID, '//Multiply Logic\n');

fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '    if(current_state[%i:3] == 0)\n', msb);
fprintf(fileID, '        begin\n');
fprintf(fileID, '            multiply_x = 24''b0;\n');
fprintf(fileID, '            multiply_b = 34''b0;\n');
fprintf(fileID, '        end\n');

for k=1:1:(numA)
    fprintf(fileID, '    else if(current_state[%i:3] == %i)\n', msb, k);
    fprintf(fileID, '        begin\n');
    fprintf(fileID, '            multiply_x = y_delayed_%i;\n', k);
    fprintf(fileID, '            multiply_b = coef_a_%i;\n', k);
    fprintf(fileID, '        end\n\n');
end

fprintf(fileID, '    else if(current_state[%i:3] == %i)\n', msb,numA+1);
fprintf(fileID, '        begin\n');
fprintf(fileID, '            multiply_x = sample_in;\n');
fprintf(fileID, '            multiply_b = coef_b_0;\n');
fprintf(fileID, '        end\n');

for k=(numA+2):1:(numB+numA)
```

```
    fprintf(fileID, '    else if(current_state[%i:3] == %i)\n', msb, k);
    fprintf(fileID, '        begin\n');
    fprintf(fileID, '            multiply_x = x_delayed_%i;\n', k-numA-1);
    fprintf(fileID, '            multiply_b = coef_b_%i;\n', k-numA-1);
    fprintf(fileID, '        end\n\n');
end

fprintf(fileID, '    else\n');
fprintf(fileID, '        begin\n');
fprintf(fileID, '            multiply_x = 24''b0;\n');
fprintf(fileID, '            multiply_b = 34''b0;\n');
fprintf(fileID, '        end\n\n');

%% Accumulate
fprintf(fileID, '//Accumulate Logic \n');
fprintf(fileID, 'always @ (posedge clk_100M)\n');
fprintf(fileID, '    if(current_state <= 8)\n');
fprintf(fileID, '        accumulate_in_temp = 64''b0;\n');
fprintf(fileID, '    else if (current_state[2:0] == 3''b000)\n');
fprintf(fileID, '        accumulate_in_temp = accumulate_in;\n');
fprintf(fileID, '    else\n');
fprintf(fileID, '        accumulate_in_temp = accumulate_in_temp;\n\n');

fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '    if(current_state == 0)\n');
fprintf(fileID, '        accumulate_out = 64''b0;\n');
fprintf(fileID, '    else if((current_state[%i:3] != %i) &&
(current_state[2:0] == 3''b100))\n',msb, M+1);
fprintf(fileID, '        accumulate_out = accumulate_in_temp;\n');
fprintf(fileID, '    else if(current_state[%i:3] != %i)\n', msb, M+1);
fprintf(fileID, '        accumulate_out = accumulate_out;\n');
fprintf(fileID, '    else\n');
fprintf(fileID, '        accumulate_out = 64''b0;\n\n');

%% Output
fprintf(fileID, '//Output Logic \n');
fprintf(fileID, 'always @(posedge clk_100M)\n');
fprintf(fileID, '    if(current_state == %i)\n', max_state-7);
fprintf(fileID, '        filtered_out = {accumulate_in[63],
accumulate_in[52:30]};\n');
fprintf(fileID, '    else\n');
fprintf(fileID, '        filtered_out = filtered_out;\n\n');

fprintf(fileID, 'assign output_enable = (current_state == %i) ? 1''b1 :
1''b0;\n\n',max_state-2);
fprintf(fileID, 'assign filter_output = (reset) ? sample_in :
filtered_out;\n');

fprintf(fileID, 'endmodule\n');

fclose(fileID);


end
```

Verilog Code

ADSR Envelope Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: WPI
// Engineer: Evan Briggs and Sidney Veilleux
//
// Create Date: 01/17/2015 03:36:17 PM
// Design Name:
// Module Name: adsr_envelope
// Project Name:
// Target Devices:
// Tool Versions:
// Description: ADSR Amplitude Envelope
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module adsr_envelope_3(
    input clk_100M,
    input [31:0] envelope_control,
    input note_on,
    output reg[15:0] scalar
    );

    //split inputs
    wire[7:0] attack_control;
    wire[7:0] decay_control;
    wire[7:0] sustain_control;
    wire[7:0] release_control;

    assign attack_control  = (127 - envelope_control[31:24]);
    assign decay_control   = (127 - envelope_control[23:16]);
    assign sustain_control = envelope_control[15:8];
    assign release_control = (127 - envelope_control[7:0]);

    //Clocking and state registers
    wire pulse_1k;
    reg[2:0] envelope_state;
    reg[2:0] next_state;

    //Define Each State for Amplitude Envelope Stages
    parameter idle_param = 3'b000;
    parameter attack_param = 3'b001;
    parameter decay_param = 3'b010;
    parameter sustain_param = 3'b011;
    parameter release_param = 3'b100;
    //Scalars for state machine
    reg signed [16:0] scalar_temp;

    //Scalars for rates and sustain level
    wire[15:0] attack_scalar;
    wire[15:0] decay_scalar;
```

```
    wire[15:0] sustain_scalar;
    wire[15:0] release_scalar;

    assign sustain_scalar = {sustain_control, 8'b0};

    //Exponentially Map Controller Values to Scalars
    controller_map attack_cntrl(.ctlr_in(attack_control), .ctlr_out(attack_scalar));
    controller_map decay_cntrl(.ctlr_in(decay_control), .ctlr_out(decay_scalar));
    controller_map release_cntrl(.ctlr_in(release_control),
.ctlr_out(release_scalar));

    //Create 1k Clock Signal for counting milliseconds in envelope stages
   clk_div_100M_1k div_clk(.clk_100M(clk_100M), .clk_1k(pulse_1k));

    always @(posedge clk_100M)
        if(pulse_1k)
            case(envelope_state)
                idle_param:
                    if(note_on)
                        next_state <= attack_param;
                    else
                        next_state <= idle_param;
                attack_param:
                    if(scalar_temp >= 16'h7FFF)
                        next_state <= decay_param;
                    else if(note_on == 0)
                        next_state <= release_param;
                    else
                        next_state <= attack_param;
                decay_param:
                    if(scalar_temp <= sustain_scalar)
                        next_state = sustain_param;
                    else if(note_on == 0)
                        next_state <= release_param;
                    else
                        next_state <= decay_param;
                sustain_param:
                    if(note_on)
                        next_state <= sustain_param;
                    else
                        next_state <= release_param;
                release_param:
                    if(scalar_temp <= 0)
                        next_state = idle_param;
                    else if(note_on)
                        next_state <= attack_param;
                    else
                        next_state <= release_param;
              endcase
           else
                next_state <= next_state;

       always @(posedge clk_100M)
            if(pulse_1k)
                envelope_state <= next_state;
            else
               envelope_state <= envelope_state;

        always @(posedge clk_100M)
            if(pulse_1k)
                case(envelope_state)
                    idle_param:
                        begin
```

```
                    scalar_temp = 16'h0000;
                    scalar = 16'h0000;
                end
            attack_param:
                begin
                    scalar_temp = scalar_temp + attack_scalar;
                    if(scalar_temp > 16'h7FFF)
                        begin
                        scalar_temp = 16'h7FFF;
                        scalar = scalar_temp[15:0];
                        end
                    else
                        scalar = scalar_temp[15:0];
                end
            decay_param:
                begin
                    scalar_temp = scalar_temp - decay_scalar;
                    if (scalar_temp <= sustain_scalar)
                        begin
                        scalar_temp = sustain_scalar;
                        scalar = scalar_temp[15:0];
                        end
                    else if (scalar_temp <= 0)
                        begin
                        scalar_temp = 16'h0000;
                        scalar = scalar_temp[15:0];
                        end
                    else
                        scalar = scalar_temp[15:0];
                end
            sustain_param:
                begin
                    scalar = scalar_temp; //sustain_scalar;
                 end
            release_param:
                begin
                    scalar_temp = scalar_temp - release_scalar;
                    if (scalar_temp <= 0)
                        begin
                        scalar_temp = 16'h0000;
                        scalar = scalar_temp[15:0];
                        end
                    else
                        scalar = scalar_temp[15:0];
                end
          endcase
     else
        begin
            scalar <= scalar;
            scalar_temp <= scalar_temp;
        end
endmodule
```

LFSR Noise Effect Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Worcester Polytechnic Institute MQP
// Engineer: Sidney Veilleux and Evan Briggs
//
// Create Date: 02/18/2015 01:37:33 PM
// Design Name: RJD MQP 2015
// Module Name: lfsr_24bit
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module lfsr_24bit(
    input clk_100M,
    input clk_96k,
    input signed [23:0] nco_in,
    input signed [23:0] mult_result,
    input sync_reset,
    output reg [23:0] noise_mult,
    output clock_en,
    output reg [23:0] noise_out,
    output output_enable
    );

    //State machine consts
    parameter IDLE = 3'b000;
    parameter SHIFT = 3'b001;
    parameter MULT = 3'b010;
    parameter ADD = 3'b011;
    parameter OUT = 3'b100;

    reg [23:0] total;

    //Create 1ns pulse from 96kHz clock
    reg clk_96k_q;
    reg clk_96k_qq;
    wire clk_96_pulse;
    always @(posedge clk_100M)
            begin
                clk_96k_q <= clk_96k;
                clk_96k_qq <= clk_96k_q;
            end
    assign clk_96_pulse = (clk_96k_q && ~(clk_96k_qq));

    assign clock_en = (current_state == MULT) ? 1'b1 : 1'b0;

    assign output_enable = (current_state == OUT) ? 1'b1 : 1'b0;

    always @(posedge clk_100M)
        if(current_state == OUT)
            noise_out = total;
```

```
        else
            noise_out = noise_out;

    always @ (posedge clk_100M)
    if(current_state == MULT)
        noise_mult[23:0] = noise[23:0];
    else
        noise_mult = 24'b0;




    always @ (posedge clk_100M)
    if(current_state == ADD)
        total = mult_result + nco_in;
    else
        total = total;

//    assign total = (current_state == ADD) ? (mult_result + nco_in) : total;

    //State registers
    reg [2:0] current_state;
    wire [2:0] next_state;
    reg [1:0] mult_delay_count;

    //Next state logic
    always @ (posedge clk_100M)
        current_state = next_state;

    assign next_state = ((current_state == IDLE) && (clk_96_pulse == 1'b1)) ? SHIFT :
                        ((current_state == IDLE) && (clk_96_pulse == 1'b0)) ? IDLE  :
                        (current_state == SHIFT) ? MULT :
                        ((current_state == MULT) && (mult_delay_count == 2'b11)) ? ADD
:
                        ((current_state == MULT) && (mult_delay_count != 2'b11)) ?
MULT :
                        (current_state == ADD) ? OUT :
                        (current_state == OUT) ? IDLE :
                        IDLE;
 //   always @ (posedge clk_100M)
  //      case(current_state)
   //         IDLE: if(clk_96_pulse)
    //                next_state = SHIFT;
     //             else
     //               next_state = IDLE;
     //       SHIFT: next_state = MULT;
     //       MULT: if(mult_delay_count == 2'b11)
    //                next_state = ADD;
    //              else
   //                 next_state = MULT;
   //         ADD: next_state = OUT;
   //         OUT: next_state = IDLE;
   //         default: next_state = IDLE;
   //     endcase

    //Count clock cycles for multiply stage
    always @ (posedge clk_100M)
    if(current_state == MULT)
        mult_delay_count = mult_delay_count + 1'b1;
    else
        mult_delay_count = 2'b00;

    reg [23:0] noise;
```

```
    always @ (posedge clk_100M)
    if(~sync_reset)
        noise = 24'b101010111010101010101010;
    else if(current_state == SHIFT)
        begin
        noise[0] <= noise[19]^noise[13]^noise[3];
        noise[1] <= noise[0];
        noise[2] <= noise[1]^noise[0];
        noise[3] <= noise[2];
        noise[4] <= noise[3]^noise[0];
        noise[5] <= noise[4];
        noise[6] <= noise[5];
        noise[7] <= noise[6]^noise[0];
        noise[8] <= noise[7];
        noise[9] <= noise[8];
        noise[10] <= noise[9]^noise[0];
        noise[11] <= noise[10];
        noise[12] <= noise[11]^noise[0];
        noise[13] <= noise[12];
        noise[14] <= noise[13];
        noise[15] <= noise[14];
        noise[16] <= noise[15]^noise[0];
        noise[17] <= noise[16];
        noise[18] <= noise[17]^noise[0];
        noise[19] <= noise[18];
        noise[20] <= noise[19];
        noise[21] <= noise[20]^noise[0];
        noise[22] <= noise[21];
        noise[23] <= noise[22];
        end
     else
        noise = noise;

//    always @ (posedge clk_100M)
//    if(~sync_reset)
//        noise = 24'b101010111010101010101010;
//    else if(current_state == SHIFT)
//        begin
//        noise[0] <= noise[19]^noise[13]^noise[3];
//        noise[1] <= noise[0];
//        noise[2] <= noise[1];
//        noise[3] <= noise[2];
//        noise[4] <= noise[3];
//        noise[5] <= noise[4];
//        noise[6] <= noise[5];
//        noise[7] <= noise[6];
//        noise[8] <= noise[7];
//        noise[9] <= noise[8];
//        noise[10] <= noise[9];
//        noise[11] <= noise[10];
//        noise[12] <= noise[11];
//        noise[13] <= noise[12];
//        noise[14] <= noise[13];
//        noise[15] <= noise[14];
//        noise[16] <= noise[15];
//        noise[17] <= noise[16];
//        noise[18] <= noise[17];
//        noise[19] <= noise[18];
//        noise[20] <= noise[19];
//        noise[21] <= noise[20];
//        noise[22] <= noise[21];
//        noise[23] <= noise[22];
```

```
//        end
//     else
//        noise[23:0] = noise[23:0];


endmodule
```

Compression Effect Module

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Worcester Polytechnic Institute
// Engineer: Evan Briggs and Sidney Veilleux
//
// Create Date: 02/17/2015 01:22:41 PM
// Design Name: RJD MQP 2015
// Module Name: waveform_compression
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module waveform_compression(
    input clk_100M,
    input clk_96k,
    input signed [27:0] sample_in,
    input [7:0] effect_control,
    output output_enable,
    output signed [23:0] sample_out
    );


parameter IDLE = 2'b00;
parameter CLIP_HIGH = 2'b01;
parameter CLIP_LOW = 2'b10;

reg[1:0] current_state;
reg[1:0] next_state;

wire signed [23:0] threshold_high;
wire signed [23:0] threshold_low;

assign threshold_high = {1'b0, effect_control[6:0], 16'h0000};
assign threshold_low = {1'b1, ~effect_control[6:0], 16'hFFFF};


 //Create 96k pulse synchronized to clk_100M
reg clk_96k_q;
reg clk_96k_qq;
wire clk_96_pulse;
always @(posedge clk_100M)
        begin
            clk_96k_q <= clk_96k;
            clk_96k_qq <= clk_96k_q;
        end
assign clk_96_pulse = (clk_96k_q && ~(clk_96k_qq));

assign output_enable = clk_96_pulse;


always @(posedge clk_100M)
```

```
    current_state = next_state;


always @(posedge clk_100M)
    if(clk_96_pulse)
        next_state = (sample_in > threshold_high) ? CLIP_HIGH :
                     (sample_in < threshold_low)  ? CLIP_LOW  :
                     IDLE;
    else
        next_state = current_state;

assign sample_out = (current_state == CLIP_HIGH) ? threshold_high :
                     (current_state == CLIP_LOW)  ? threshold_low  :
                      {sample_in[27],sample_in[22:0]};

endmodule
```

Embedded Software Excerpts

Output Sample Processing Interrupt Service Routine

```
void Timer_InterruptHandler(void *data, u8 TmrCtrNumber) {

	//Pull in sample

	u32 processedIn = XGpio_DiscreteRead(&GpioData, 2);

	//Change to int format
	int processedInInt = u32toint(processedIn);

	//Calculate Velocity
	int post_vel = (int)(velocity_current*processedInInt);

	//Calculate Delay Function
	delay_store = post_vel;
	int post_delay = (delayedSamples[delayedSamplesIndex]*delayWet) + 
(post_vel*delayDry);

//	//Calculate Echo
//	int post_echo = (post_delay) + (echoSamples[echoIndex]*echoWet); //echo dry
//	echo_store = post_echo*echoDecay;

	//Calculate Echo
	int post_echo = (post_delay) + (echoSamples[echoIndex]*echoWet); //echo dry
	echo_store = post_echo*echoDecay;

	//Clip if above Max
	if(echo_store  > MAXAMPLITUDE)
		echo_store = MAXAMPLITUDE;
	else if(echo_store < MINAMPLITUDE)
		echo_store = MINAMPLITUDE;

	//Change back to 24bit u32
	u32 final_24b = inttou32(post_echo);

	//Output sample to codec
	Xil_Out32(I2S_DATA_TX_L_REG, final_24b);
	Xil_Out32(I2S_DATA_TX_R_REG, final_24b);

	InterruptFlag = 1;
	delayFlag = 1;

}
```

Controller Value Update Loop

```
        //Status In
        do{

                adsr_out = 
((controller_buffer[ATTACK]<<24)|(controller_buffer[DECAY]<<16)|(controller_buffer[SU
STAIN]<<8)|(controller_buffer[RELEASE]));
                filters_out = 
((controller_buffer[FIRHP]<<24)|(controller_buffer[FIRLP]<<16)|(controller_buffer[IIR
HP]<<8)|(controller_buffer[IIRLP]));
                delay_values = 
((controller_buffer[DELAYDEPTH]<<8)|controller_buffer[DELAYWET]);
                echo_depth = controller_buffer[ECHODEPTH];
                echo_values = 
((controller_buffer[ECHODEPTH]<<16)|(controller_buffer[ECHODECAY]<<8)|controller_buff
er[ECHOWET]);
                pwm_value = controller_buffer[PWM];
                compression_value = 
((controller_buffer[REDUX]<<24)|(controller_buffer[NOISEAMP]<<16)|(controller_buffer[
COMPGAIN]<<8)|controller_buffer[COMPCLIP]);
                lfo_values = 
((controller_buffer[LFOAMPPWM]<<16)|(controller_buffer[LFOAMPDEPTH]<<8)|controller_bu
ffer[LFOAMPRATE]);
                lfo1_values = 
((lfo1<<24)|(controller_buffer[LFO1PWM]<<16)|(controller_buffer[LFO1DEPTH]<<8)|contro
ller_buffer[LFO1RATE]);
                lfo2_values = 
((lfo2<<24)|(controller_buffer[LFO2PWM]<<16)|(controller_buffer[LFO2DEPTH]<<8)|contro
ller_buffer[LFO2RATE]);
                lfoWaveSel = (((u32)lfo2Wave<<4)|(u32)lfo1Wave);
                wavephase = 
((controller_buffer[TRIPHASE]<<24)|(controller_buffer[SQUAREPHASE]<<16)|(controller_b
uffer[SAWPHASE]<<8)|controller_buffer[SINEPHASE]);

                //Delay Buffer
                if(delayFlag == 1){
                        intMasterDisable();

                        //Update current Index
                        currentSamplesIndex++;
                        if(currentSamplesIndex >= 192000)
                                currentSamplesIndex = currentSamplesIndex - 192000;

                        //Update current Index of Delay
                        delayedSamplesIndex =  currentSamplesIndex - delay;
                        if(delayedSamplesIndex < 0)
                                delayedSamplesIndex = 192000 + delayedSamplesIndex;

                        //Store sample in delay buffer
                        delayedSamples[currentSamplesIndex] = delay_store;
                        echoSamples[echoIndex] = echo_store;

                        //Update current Echo Index
                        echoIndex++;
```

```
            if(echoIndex >= maxEchoIndex)
                echoIndex = 0;

            delayFlag = 0;

            intMasterEnable();
        }
        if(wavephase != wavephase_prev){
            XGpio_DiscreteWrite(&GpioLfoWaveSel,2,wavephase);
            wavephase_prev = wavephase;
        }
        if(lfoWaveSel != lfoWaveSel_prev){
            XGpio_DiscreteWrite(&GpioLfoWaveSel,1,lfoWaveSel);
            lfoWaveSel_prev = lfoWaveSel;
        }
        else if(lfo1_values != lfo1_values_prev){
            XGpio_DiscreteWrite(&GpioLfo1,1,lfo1_values);
            lfo1_values_prev = lfo1_values;
        }
        else if(lfo2_values != lfo2_values_prev){
            XGpio_DiscreteWrite(&GpioLfo1,2,lfo2_values);
            lfo2_values_prev = lfo2_values;
        }
        else if(compression_value != compression_value_prev){
            XGpio_DiscreteWrite(&GpioFilter,2,(u32)compression_value);
            compression_value_prev = compression_value;
        }
        else if(lfo_values != lfo_values_prev){
            XGpio_DiscreteWrite(&GpioLfo,1,lfo_values);
            lfo_values_prev = lfo_values;
        }
        else if(pwm_value != pwm_value_prev){
            XGpio_DiscreteWrite(&Gpio2,2,controller_buffer[PWM]);
            pwm_value_prev = pwm_value;
        }
        else if(delay_values != delay_values_prev){
            intMasterDisable();
            delayWet = wetDry[controller_buffer[DELAYWET]];
            delayDry = wetDry[127 -controller_buffer[DELAYWET]];
            delay = delayDepth[controller_buffer[DELAYDEPTH]];
            intMasterEnable();
            delay_values_prev = delay_values;
        }
        else if(echo_values != echo_values_prev){
            intMasterDisable();
            echoWet = wetDry[controller_buffer[ECHOWET]];
            echoDry = wetDry[127 -controller_buffer[ECHOWET]];
            if(echo_depth > echo_depth_prev){
                for(i=delayDepth[echo_depth_prev];
i<delayDepth[echo_depth]; i++){
                    echoSamples[i] = 0;
                }
            }
            maxEchoIndex = delayDepth[controller_buffer[ECHODEPTH]];
            echoDecay = wetDry[controller_buffer[ECHODECAY]];
```

```
			intMasterEnable();
			echo_depth_prev = echo_depth;
			echo_values_prev = echo_values;

		}
		else if(adsr_out != adsr_out_prev){
			XGpio_DiscreteWrite(&GpioNote, 2, (u32)adsr_out);
			adsr_out_prev = adsr_out;
		}
		else if(filters_out != filters_out_prev){
			XGpio_DiscreteWrite(&GpioFilter, 1,(u32)filters_out);
			filters_out_prev = filters_out;
		}
		else if(switch_in!=XGpio_DiscreteRead(&Gpio, SWITCH_CHANNEL)){
			switch_in = XGpio_DiscreteRead(&Gpio, SWITCH_CHANNEL);
			Xil_Out32(LED_BASE, switch_in);
			XGpio_DiscreteWrite(&GpioWave,1,switch_in);
		}
	}while (!XUartPs_IsReceiveData(UART_BASEADDR));
```