

Rochester Institute of Technology
RIT Scholar Works

[Theses](#)

5-2020

Design of DSP Guitar Effects with FPGA Implementation

Connor William Blasie
cwb8984@rit.edu

Follow this and additional works at: <https://scholarworks.rit.edu/theses>

Recommended Citation

Blasie, Connor William, "Design of DSP Guitar Effects with FPGA Implementation" (2020). Thesis.
Rochester Institute of Technology. Accessed from

This Master's Project is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

DESIGN OF DSP GUITAR PEDAL EFFECTS WITH FPGA IMPLEMENTATION

by
CONNOR WILLIAM BLASIE

GRADUATE PAPER

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

Approved by:

Mr. Mark A. Indovina, Senior Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
MAY 2020

I would like to dedicate this paper to my family, friends, and all of those I have had the joy of meeting through this journey at Rochester Institute of Technology. Each relationship has sculpted an experience culminating in this work that would not have been possible in any other aspect.

Abstract

The face of music has been drastically evolving over the past century starting with the advent of the electric guitar. The emergence of digital signal processing in guitar audio applications has constantly been driven by the ability to diversify tonalities from experimenting with instrument materials, pick-ups, amplifiers, and effects. This paper demonstrates an alternative approach to guitar effects by using FPGA implementation, in lieu of DSP cores or analog components, to perform digital signal processing intended for alteration of guitar audio signals. The audio processing of the multiple effects were developed on a Zedboard incorporated with a Xilinx Zynq®-7000 SOC which utilizes programmable logic fabric as well as dual ARM-A9 processors. The different guitar effects were derived from custom IP blocks written in a mixtures of Verilog and VHDL in the programmable logic (PL) are controlled with rotary potentiometers and switches on custom guitar effect pedals that connect to the processor system (PS) via an I2C bus. The overall design serves to offer an alternative solution to traditional guitar pedals which perform the signal processing with lower latency while eliminating numerous patch cables, power cables, and overall costly pedals.

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Connor William Blasie

May, 2020

Acknowledgements

I would like to thank my advisor Mark Indovina for his phenomenal support through this amazing academic adventure and setting me up for a lifetime of learning. I surely would not be in the same exuberant mindset without Mark's guidance, advice, and ample desire to encourage this work as much as I have dreamt about. In my mind, Mark stands as a benchmark that all teachers should strive to achieve; a balance of humor, humility, and a desire to bring others to the precipice of knowledge.

The RIT Baja SAE team is a critical component to my success of incorporating real world problem solving with aspects learned in the classroom. I have had the privilege of working with many fine students, many of whom I share many sleepless night and hilarious stories with.

Lastly, I want to thank my family as well as my fiancee Samantha for being supportive through this whole ordeal. They have never hesitated to encourage and bring out the best in me and give their unwavering love.

Contents

Contents	v
List of Figures	vii
List of Tables	x
Listings	xi
1 Introduction	1
1.1 Intentions of this Project	1
1.2 Previous Work	2
1.3 Organization	3
2 Synopsis of guitars and pedal effects	4
2.1 Electric Guitars	4
2.2 Rise of Digital Signal Processing for guitars	5
3 SoC and Embedded Systems Architecture	7
3.1 Hardware Overview	7
3.2 AXI Interconnect for PL-PS Connections	9
3.3 Zynq Processing System (PS)	12
3.3.1 Core 0: Audio Codec	12
3.3.2 Core 1: MSP430 Guitar Pedals	14
3.4 Programmable Logic (PL)	18
4 Distortion Effects	22
4.1 Gain Effect	22
4.2 Overdrive Effect	25
4.3 Distortion Effect	29
5 Time Varying Effects	34
5.1 Delay Effect	34

5.2	Chorus Effect	38
5.3	Tremolo Effect	41
6	Attenuation Effects	46
6.1	Noise Gate Effect	46
6.2	Equalizer Effect	51
7	Verification and Performance	58
7.1	Hardware Verification	58
7.2	Equalizer	59
7.3	Gain	63
7.4	Overdrive	65
7.5	Distortion	68
7.6	Chorus	70
7.7	Tremolo	73
7.8	Delay	75
7.9	Noise Gate	79
8	Conclusion	82
8.1	Conclusion	82
8.2	Future work	83
References		85
I	Fixed Point Mathematics	I-1
II	HDL Source Code	II-4
III	Hardware Pinout	III-244
IV	MSP430 C Code	IV-249
V	MATLAB Equalizer Filter Generation	V-266
VI	Waveform Signals	VI-271

List of Figures

2.1	Timeline of the acoustic guitar [1]	5
3.1	Zedboard Block Diagram [2]	8
3.2	Zynq 7020 Block diagram	9
3.3	Audio Processing AXI protocol IP	11
3.4	Audio codec chip diagram [3]	13
3.5	Zynq Core 0 Flowchart	14
3.6	MSP430 Flowchart	15
3.7	MSP430 Graphical Pinout [4]	16
3.8	Zynq Core 1 Flowchart	17
3.9	Guitar Pedal Enclosure	17
3.10	PL main block diagram	18
3.11	cwb_PL_processing hierarchy block diagram	19
3.12	cwb_DSP_effects block diagram	20
3.13	Guitar Effects signal chain	21
3.14	I2C data algorithm parameter control loop	21
4.1	Gain effect block diagram	23
4.2	Gain effect algorithm	24
4.3	Gain effect with high gain input 0xF0 corresponding to a gain of 15	24
4.4	Gain effect with low gain input 0x10 corresponding to a gain of 1	25
4.5	Example of hard and soft clipping for an audio signal [5]	26
4.6	Hard clipping block diagram	26
4.7	Overdrive hard clipping algorithm	28
4.8	Overdrive test bench results	28
4.9	Quantization Distortion Effect in MATLAB simulation - Small Quantization Regions	30
4.10	Quantization Distortion Effect in MATLAB simulation - Large Quantization Regions	31
4.11	Distortion quantization audio range bin configuration: Zero based large bins (left) Level based large bins (center) Level based small bins (right)	32

4.12	Distortion effect quantization process: Positive value quantization (top) and negative value quantization (bottom)	33
4.13	Distortion test bench	33
5.1	Delay Block Diagram	35
5.2	Delay effect algorithm	37
5.3	Delay effect simulation	38
5.4	Chorus effect block diagram	39
5.5	Nearest Neighbor Interpolation L=8 Reconstruction Algorithm	40
5.6	Chorus Effect Simulation	41
5.7	Example of tremolo AM [6]	42
5.8	Tremolo block diagram	42
5.9	Tremolo Wavesheaping value in MATLAB simulations: 255	43
5.10	Tremolo Wavesheaping value in MATLAB simulations: 127	44
5.11	Tremolo Wavesheaping value in MATLAB simulations: 1	44
5.12	Tremolo test bench simulation	45
6.1	Noise gate response in dB [7]	47
6.2	Noise gate timing parameters [8]	47
6.3	Noise gate block diagram [9]	48
6.4	Noise gate test bench simulation	51
6.5	Equalizer Block Diagram	52
6.6	Low-pass bass filter response	54
6.7	Band-pass Mids filter response	55
6.8	High-pass Treble filter response	55
6.9	Equalizer filter response	56
6.10	Equalizer Algorithm HDL Simulation	57
7.1	Guitar Effect Platform Latency	59
7.2	Equalizer Spectrum - LFM_long - effect off	60
7.3	Equalizer Test One - Bass Response	61
7.4	Equalizer Test Two - Mids Response	62
7.5	Equalizer Test Three - Treble Response	62
7.6	Equalizer Test Four - Maximum value for each input parameters	63
7.7	Gain Spectrum - Sinusoid - effect off	64
7.8	Gain Test One - Attenuation	64
7.9	Gain Test Two - Boost	65
7.10	Overdrive Test One - Time Domain	66
7.11	Overdrive Test One - Frequency Domain	67
7.12	Overdrive Test Two - Time Domain	67
7.13	Overdrive Test Two - Frequency Domain	68
7.14	Distortion Reference - Overdrive Higher order Harmonics	69

7.15	Distortion Test One - Small Quantization Bins	70
7.16	Distortion Test Two - Large Quantization Bins	71
7.17	Chorus Test One - Fast LFO and small delay	72
7.18	Chorus Test Two - Slow LFO and large delay	72
7.19	Tremolo Test 1 - Fast Modulation with Triangular Wave	74
7.20	Tremolo Test 2 - Slow Modulation with Triangular Wave	74
7.21	Tremolo Test 3 - Slow Modulation with Modified Triangular/Brick Wave	75
7.22	Delay Test One - Short FIR Delay	76
7.23	Delay Test Two - Long FIR Delay	77
7.24	Delay Test Three - Short IIR Delay	78
7.25	Delay Test Four - Long IIR Delay	78
7.26	Noise Gate Test One - Opening Gate	80
7.27	Noise Gate Test Two - Closing Gate	81
8.1	Guitar Effect Pedals (Front)	83
8.2	Guitar Effect Pedals (Rear)	84
VI.1	LFM_cwb Waveform	VI-272
VI.2	LFM_long Waveform	VI-273
VI.3	Sinusoid Waveform	VI-274
VI.4	Pulsed Sinusoid Waveform	VI-275

List of Tables

3.1	I2C_ADC_data_transfer memory allocation	12
6.1	Noise gate fixed constant parameter values	50
6.2	Noise Gate calculated constant parameter values	50
6.3	Filter Specifications for EQ Filter	54
I.1	Floating point to fixed point examples	I-2
VI.1	LFM_cwb Waveform parameters	VI-272
VI.2	LFM_long Waveform parameters	VI-273
VI.3	Sinusoid Waveform parameters	VI-274
VI.4	Pulsed Sinusoid Waveform parameters	VI-275

Listings

II.1	Gain Algorithm Verilog File	II-4
II.2	Gain Algorithm Verilog Test Bench File	II-10
II.3	Overdrive Algorithm Verilog File	II-13
II.4	Overdrive Algorithm Verilog Test Bench File	II-19
II.5	Distortion Algorithm Verilog File	II-24
II.6	Distortion Algorithm Verilog Test Bench File	II-49
II.7	Delay Algorithm Verilog File	II-55
II.8	Delay Algorithm Verilog Test Bench File	II-64
II.9	Chorus Algorithm Verilog File	II-69
II.10	Chorus Algorithm Verilog Test Bench File	II-177
II.11	Tremolo Algorithm Verilog File	II-182
II.12	Tremolo Algorithm Verilog Test Bench File	II-194
II.13	Noise Gate Algorithm Verilog File	II-200
II.14	Noise Gate Algorithm Verilog Test Bench File	II-213
II.15	Equalizer Algorithm Verilog File	II-219
II.16	Equalizer Algorithm Verilog Test Bench File	II-227
III.1	Xilinx Zynq 7020 XDC Pinout	III-246
IV.1	MSP430 I2C Slave Interrupt C File	IV-249
V.1	Lowpass EQ Bass Filter	V-266
V.2	Bandpass EQ Mids Filter	V-267
V.3	Highpass EQ Treble Filter	V-269
V.4	EQ filter overlay	V-270

Chapter 1

Introduction

1.1 Intentions of this Project

This project serves to discuss and build upon numerous aspects of electrical engineering related to audio signal processing by building a platform to perform various guitar effects digitally with FPGA implementation. The following aspects of the project can be broken into the following general skills and focus areas:

- Embedded programming of microcontrollers / processors / peripheries
- HDL programming of FPGA fabric
- Serial communication protocols
- AXI Interconnects (FPGA to processor communication)
- Algorithm development

1.2 Previous Work

The main inspiration for this work comes from a senior design project from the University of Tel Aviv [10]. The group developed an audio signal processing platform on the Zedboard evaluation board. The evaluation board is used to sample the incoming audio signal with a codec (coder-decoder), passes the audio from the processor to the FPGA fabric, apply digital processing algorithms in the FPGA, passes the modified signal from the fabric to the processor, routes the signal from the processor back to the codec, and finally passes the data to the output jack. The group created the platform with the intent of having a user interface of buttons and switches on the Zedboard that allowed the guitar player to chose between specific preset effects. The effects included:

- Overdrive/Distortion
- Delay
- Tremolo
- Octaver - an experimental mixture of tremolo and octave

The project was successful in completing the audio processing loop described above and demonstrating the four aforementioned guitar effects. However, there were two main design features native to guitar pedals that were not included in the project. The first is the tunability that is common with a guitar effect pedal where a user can alter the effect with potentiometers and switches rather than having only preset effects to chose from. The second is that the Tel Aviv user interface consisted of buttons and switches on the Zedboard where traditional guitar pedals are foot controlled. These two design limitations were noted in their paper as a limitation in time did not allow for the further development of the additional hardware and software. This paper seeks to build upon these two design limitations by creating tunable algorithms

and a user interface that is foot controlled, more traditional to a guitar pedal that is not mechanically connected to the evaluation board. The user interface will consist of a circuit board that takes user inputs in the form of switches and potentiometers. The tunable guitar effect algorithms will take in one to four user inputs from the pedal user interface and change the audio algorithm based upon the input parameters. These proposed modifications will involve developing new algorithms as the previous algorithms do not support the tunability function as well as developing additional algorithms for implementing more effects. This will also involve developing additional software for reading the user inputs on the foot pedals as well as communicating the data into the FPGA for tuning the algorithms.

1.3 Organization

Chapter 2 will discuss the history of guitar effects and how the rise of digital signal processing (DSP) has come to be an accepted standard in guitar effect pedals. Chapter 3 will outline the architecture of the Xilinx Zynq 7020 system on a chip (SoC) being used to implement the DSP as well as the Texas Instruments MSP430FR2355 which takes the user inputs from the guitar pedal effects and pushes the values to the SoC to control the DSP algorithms. Chapter 4 will cover the DSP algorithms that pertain to overdrive and distortion. Chapter 5 will cover delay, chorus, and tremolo effects which have a time varying component. Chapter 6 will cover equalizer and noise gate effects which apply attenuation. Chapter 7 will discuss effect hardware verification as well as system performance. Chapter 8 will discuss improvements that could be made to the project and overall conclusions.

Chapter 2

Synopsis of guitars and pedal effects

2.1 Electric Guitars

The rise of the electric guitar comes a century following the advent of the first acoustic guitar. C.F. Martin of Martin Guitars is credited with creating some of the first modern acoustic guitars in 1833. The first acoustic guitar designs take influence from C.F. Martin's previous design of mandolins but is targeted more towards American interests [11]. The design of the acoustic guitar changed drastically over the next century as size and shape was adapted to improve performance and overall volume as seen in Figure 2.1. At the end of the 19th century, musicians started to take note that musical venue size continued to grow requiring louder instruments; however, the acoustic guitar alone could not meet this demand [12]. A new instrument would have to be created in order to meet the needs of musicians looking to broadcast their music at louder levels.

The first modern electric guitar came about in the late 1920s leveraging the audio amplification of steel strings which was found in the late 19th century. Alan Rickenbacker created a guitar pick up using electromagnetics and steel strings for a lap-steel electric gui-



Figure 2.1: Timeline of the acoustic guitar [1]

tar. Many designers created solid body guitars through the 1930s to the 1940s, most notably The Slinger company producing Spanish style solid body electric guitars [13, 14]. Leo Fender created the first commercially successful solid body guitar first known as the Esquire in 1949 which was later renamed the iconic Fender Telecaster [15].

The sound of the guitar, known as the voicing, is quite specific to the wood and the pickup used in guitar. Many individuals sought to create their own guitar voicing, often in the form of guitar amp and speaker selection with select distortion. The first deliberate signal distortion used in music came from Johnny Burnette's Rock n Roll Trio in 1959 using a Fuzz guitar pedal made by Mike Piera [16]. Once distortion based effects started coming to market, many more commercial analog based effects started to be created like analog delay, tremolo, and reverb.

2.2 Rise of Digital Signal Processing for guitars

The advent of digital signal processing revolutionized the audio industry. Pedal effects were able to change the voicing of the electric guitar but not always as consistently as analog pedals could vary drastically due to the use of component tolerance. Digital pedals offered the ability to have more consistent voicing and tone since all processing is done in the digital domain rather than the analog domain [17, 18] . Digital guitar pedals start by sampling the incoming guitar signal with an analog to digital converter (ADC) and quantizing the signal

into audio range bins, common ranges include 16 and 24 bit levels. The digital signal processing algorithm that mimics the analog pedal effect is performed on the quantized signal. The digital signal is then converted back to the analog domain using a digital to analog converter (DAC) [19]. Many common analog pedals found themselves being replaced by smaller, cheaper, and more power efficient digital alternatives at the sacrifice of no longer having a purely analog signal chain which some guitar purists seek. Analog tube amplifier and BJT based overdrive and distortion circuits were soon replaced by symmetric and asymmetric clipping digital algorithms. Analog delay based effects using bucket-brigade devices, effectively cascaded capacitors, were replaced by digital memory. Analog reverb pedals that utilized a spring were replaced by cascaded all-pass filters[20]. As microcontrollers, microprocessors, and digital signal processors became cheaper to manufacture and easier to program, more guitar companies started to manufacture digital based guitar pedals [21].

Chapter 3

SoC and Embedded Systems Architecture

3.1 Hardware Overview

The Xilinx Zynq 70XX series is a System On a Chip (SoC) that combines the benefits of up to two ARM Cortex-A9 processors and a 28nm Xilinx Artix-7 FPGA into a single chip [22]. This combination yields a chip that is able to perform hardware processing on an FPGA with the embedded needs of a MCU/MPU for periphery interface. This chip will be utilized on the Digilent Zedboard evaluation board which has a Xilinx Zynq 7020 (XC7Z020-CLG484-1). This evaluation module also contains an Analog Devices 24 bit audio codec (ADAU1761) which is routed into the processor side of the SoC. The codec audio input and output are routed off the board using 3.5mm jacks which will be used to taking in the guitar signal and outputting the modified signal. LEDs are also used through the design for status/ debugging to show the user when the evaluation board recognizes that a guitar effect is active. The block diagram of all peripheries contained on the Zedboard evaluation board is found in Figure 3.1.

The audio processing is desired to be done in the reconfigurable hardware of the FPGA, therefore the audio data will need to be moved from the processor into the fabric via an AXI

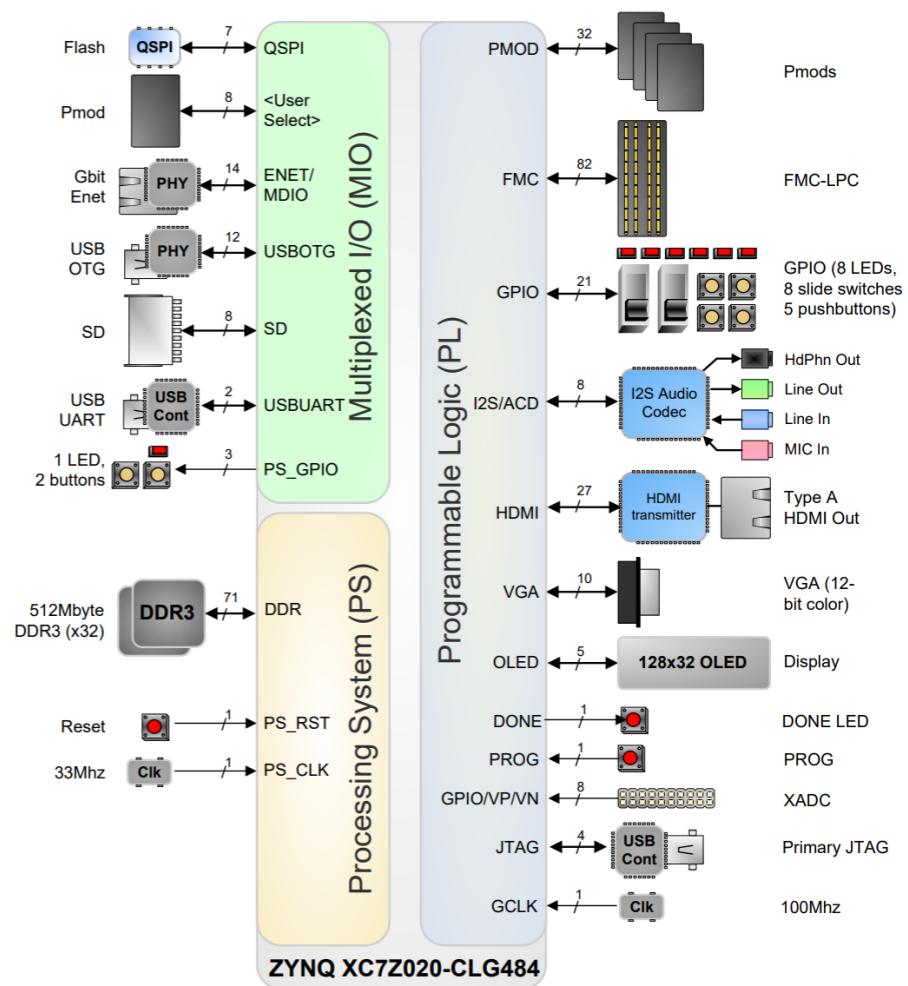


Figure 1 – ZedBoard Block Diagram

Figure 3.1: Zedboard Block Diagram [2]

bus. The codec communicates with one processor core of the SoC over I2S. An I2C channel will be used to initialize the codec. The MSP430s will emulate a guitar pedal and convert user inputs to the PS over I2C communication which will be used to alter the guitar effect algorithms. These inputs will be passed from the second processor core to the FPGA fabric using an AXI bus. The communication buses used to pass data back and forth between the aforementioned blocks can be found in Figure 3.2.

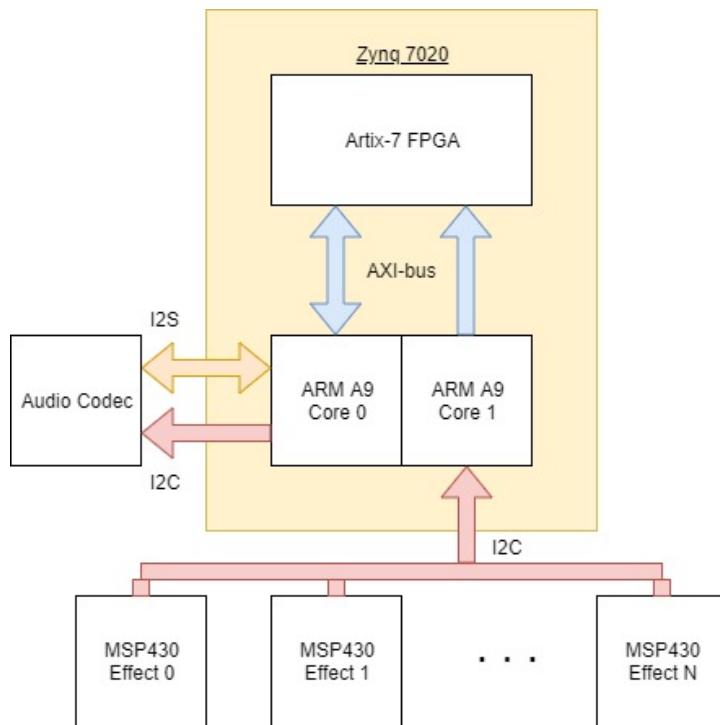


Figure 3.2: Zynq 7020 Block diagram

3.2 AXI Interconnect for PL-PS Connections

An advantage for using a SoC for this project is that the FPGA fabric in the PL and the ARM cores in the PS allows for complex design of systems with minimal delay while passing data between the PS-PL. This results in less board space wasted in PCB layout as well as demand-

ing less passive components and voltage regulators compared to if the PL and PS ICs were independent. In order to pass data between the PS and PL a standard communication protocol is utilized. Xilinx SoCs takes advantage of Advanced eXtensible Interface (AXI) in order to communicate between the PS and the PL [23]. This protocol is a Master-Slave style handshaking that is able to write to and read from preallocated memory portions of the PL and PS. This means that when data is stored in one space in memory of the PS, it can be transferred over an AXI bus into a bitstream, register, or block ram of the PL. Conversely, it can also mean that a bitstream, resister or block ram set of data in the PL can be pushed into the PS. Various free IP exists in the Xilinx developmental environment that utilizes the AXI protocol for PS-PL interaction.

This project takes use of the AXI protocol in three places: guitar audio in, algorithm parameters, and processed audio out. The guitar audio in uses AXI to move the incoming guitar audio sampled by the codec into the PL to be processed. The algorithm parameters come from the guitar pedal potentiometers on the MSP430 and are pushed from the PS into the PL to modify the audio signal processing algorithms. The processed audio out uses the AXI protocol to take the processed audio from the PL and move the data in the PS so the data can be sent back to the audio codec over the I2S interface. As part of the Zynq Book and the work done at the university of Tel Aviv, the guitar audio in and proceeded audio out AXI buses were already created, made open source, and well documented so they were used in this project[10, 24]. The algorithm parameter AXI was developed for this project specifically and follows a similar implementation to the guitar audio in AXI bus previously mentioned. The AXI busses are called in the block diagram tool in Vivado to be instantiated.

The work performed by [10, 24] was hierarchically grouped in the PL to demonstrate the work was being leverage by this project to speed up development time and not take credit for work previously performed as can be seen in Figure 3.3.

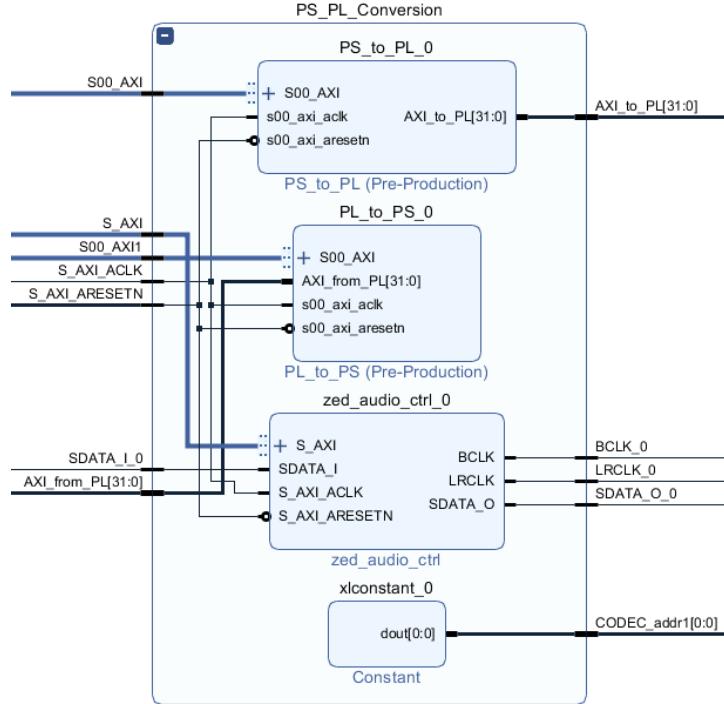


Figure 3.3: Audio Processing AXI protocol IP

The algorithm parameter AXI was developed and packaged as IP and implemented in the PL in a similar manner to the previous AXI bus but was instead placed into the PL processing hierarchy with the DSP elements in order to clean up the block diagram. The algorithm parameter AXI was listed as the “I2C_ADC_data_transfer_x”. Each guitar pedal potentiometer value is routed into the PS and then routed into the PL by the AXI bus. The address of the guitar pedal as well as the ADC values are pushed into the PL along with a data ready line to the processing blocks can update algorithm parameters. The AXI bus uses 32 bit wide register values in order to transfer the data. The memory allocation required by the AXI is minimal in the case of moving the algorithm parameters into the PL where X’s in the memory table serve as don’t cares and checks serve as utilized.

Table 3.1: I2C_ADC_data_transfer memory allocation

Reg Name	31:24	23:16	16:8	7:1	0	Base Offset
ADDR_TO_PL	X	X	X	✓	✓	0x0000000000
ADC1_TO_PL	X	X	X	✓	✓	0x0100000000
ADC2_TO_PL	X	X	X	✓	✓	0x0200000000
ADC3_TO_PL	X	X	X	✓	✓	0x0300000000
ADC4_TO_PL	X	X	X	✓	✓	0x0400000000
DataMoveEnable	X	X	X	X	✓	0x0500000000

3.3 Zynq Processing System (PS)

This design uses a multi-core processor with Core 0 dealing exclusively with the audio codec and sampling the incoming audio signal as well as outputting the processed signal and Core 1 dealing exclusively with the MSP430 I2C communication bus for gathering algorithm parameter data. While the complexity of the processor portion of the design is overall quite small and could easily be built on a single processor core, a dual core design approach was taken in order to learn more about the architecture of the Zynq. This design choice also minimized time spent developing timing/interrupt state machines for switching between the different communication buses as each core ran hardware independent of the other. The dual core approach also allowed the flexibility to run specific core tasks independently for debugging purposes early on in the design cycle.

3.3.1 Core 0: Audio Codec

The audio codec used on the Digilent Zedboard is a 24 bit stereo Analog Devices ADAU1761 shown in Figure 3.4. The codec contains onboard DSP cores running Analog Devices SigmaDSP support as well as A/D and D/A filtering. However, the codec will not be used to perform any audio processing as that will all be computed in the FPGA. Therefore the codec

is only being used to sample the incoming mono-channel guitar signal on the ADC and then output the modified audio from the guitar effect chain from the DAC. The codec is set up to record in stereo 48kHz sampling. A guitar generates a mono-channel signal therefore the audio signal is transferred into the codec on only the left channel on the input line. Likewise, the output signal from the guitar effects chain is mono-channel therefore the DAC is converting in a stereo fashion but a signal will only appear on the left channel as is convention for running a single channel through a stereo connection.

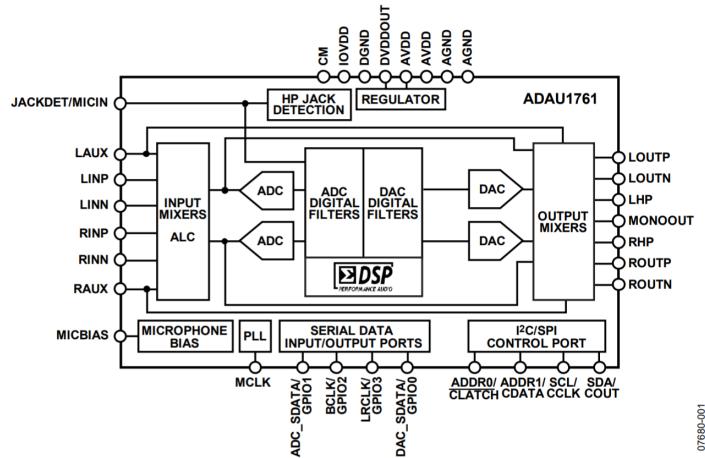


Figure 3.4: Audio codec chip diagram [3]

As mentioned before, this project builds upon the work of [10] who leverages the work performed by [24] in the Zynq Book. The majority of the work performed in setting up the audio codec goes to the two aforementioned groups as they developed the I2C interface to set up the codec, the I2S communication with the PS of the Zynq, and the AXI interfaces to pass the audio data between the PS and the PL with the flowchart shown in Figure 3.5. The only modification to the open source code was applying a gain to the output signal at the codec through an I2C configuration register while the I2S communication protocol and AXI interfaces went unmodified.

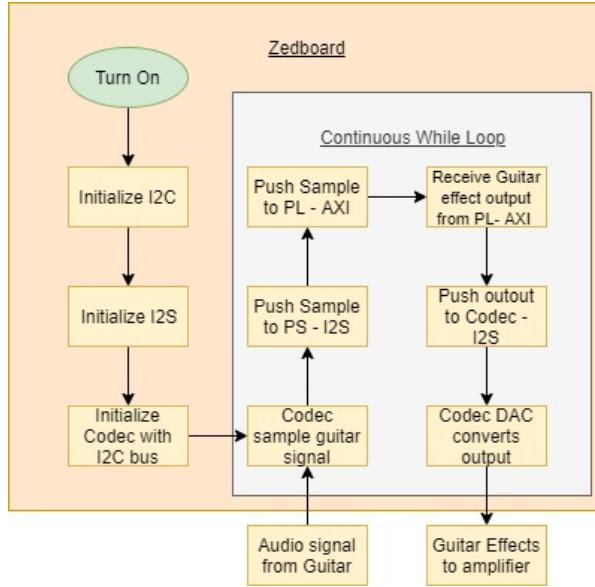


Figure 3.5: Zynq Core 0 Flowchart

3.3.2 Core 1: MSP430 Guitar Pedals

The MSP430 is a Texas Instruments microcontroller that is used as a interface between the user and the algorithm parameters that are tunable in the audio signal processing algorithms. In lieu of designing a custom PCB for the physical guitar pedals, an MSP430 evaluation board was used to save time of development and cost. Each MSP430 evaluation board (MSP-EXP430FR2355) contains a MSP430FR2355 MCU running a core clock of 24MHz, 32KB of ferroelectric RAM, and 32KB on chip memory [4]. All of the code utilized by the MSP430 was written in C using the Texas Instruments Code Composer Studio 8.3.0. The MCU is broken out on the evaluation board to 44 GPIO. Each evaluation board is given a unique I2C address which corresponds to a specific guitar effect algorithm in the FPGA. The MSP430 is set up as a slave in I2C communication and records values from an ON/OFF switch and one to four potentiometers. An internal 12 bit ADC in the MSP430 converts the analog voltages coming from the wiper of the potentiometers to a 12 bit digital value. The lowest 4 bits of the data are truncated to make the resulting digital value 8 bits for easier communication and data

storage. While this cuts down the resolution of the ADC, the trade off can be accepted as this still give 256 options within the 270 degrees of rotation that the potentiometers can spin, thus approximately each degree of the potentiometer corresponds to a change in the input signal. Shortening the data to eight bits also allows the data to be more easily communicated over I2C as the I2C standard data protocol sends 8 bits of data and one stop bit per each cycle.

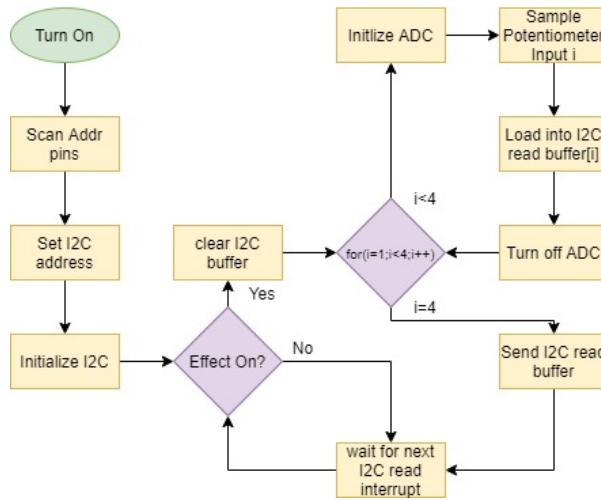


Figure 3.6: MSP430 Flowchart

Figure 3.6 serves to show the operating sequence from power up to communication with the ZedBoard. When the MSP430 powers on, four GPIO pins shown in Figure 3.7, are used to set the I2C address of the specific board with jumpers. If an address GPIO pin is pulled up with a jumper then a corresponding bit position is added as an offset to the I2C address base which is 0x40 (hex). Concatenating the adders pins into a 4 bit value [P6.3, P6.2, P6.1, P6.0] and adding the hex value from this process to the base address gives an address range of 0x40 to 0x4F for sixteen pedal options.

After the I2C address is set with the GPIO headers, the I2C hardware within the MSP430 is initialized. The I2C is set in interrupt driven mode. After each I2C read command has been issued to the MSP430 and completed, the read buffer is updated. If the effect is turned on

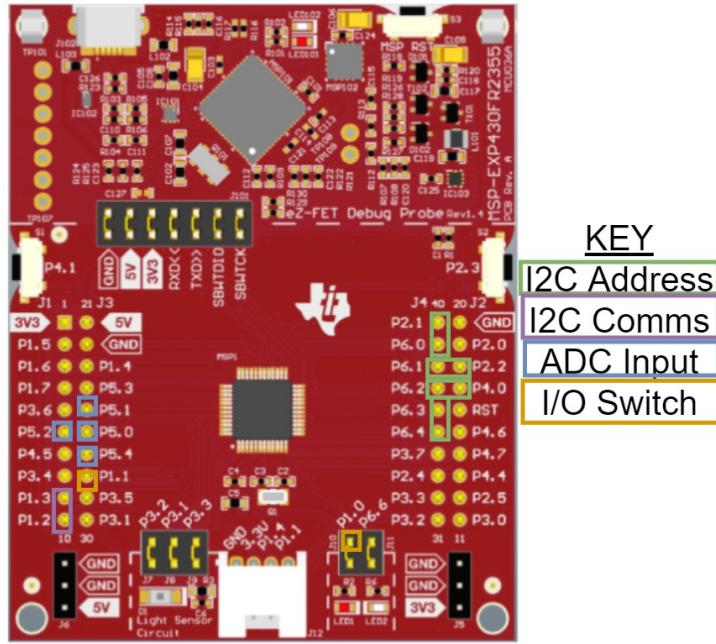


Figure 3.7: MSP430 Graphical Pinout [4]

by means of the SPST ON/OFF switch, each ADC input corresponding to a potentiometer is sampled. Each potentiometer value is updated in the I2C read buffer and sent on the next I2C read command. If the pedal is turned off, the read buffer is sent as all zeros. By sending all zeros for an off state, the pedal status is able to be encoded and later read in the PL as an off state instead of sending an additional data byte signifying an active state. The incoming I2C data is parsed out in the PS as shown in Figure 3.8 and written to the AXI bus that pushes the data to the PL to alter the audio effect algorithms.

The MSP430 is housed in a custom enclosure designed specifically to fit the mounting holes in the MSP430 evaluation board. The enclosure, shown in Figure 3.9, was designed in Solidworks 2019 and 3D printed for rapid prototyping. Based upon parameters needed for the audio processing algorithms, it was decided early on that the most parameters a guitar pedal would need would be three. Therefore the enclosure was designed to have a SPST switch with LED indicator for turning on the guitar pedal and three potentiometers to modify the algorithm.

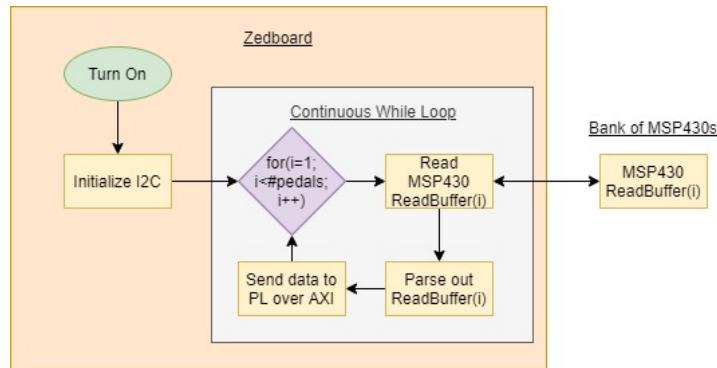


Figure 3.8: Zynq Core 1 Flowchart



Figure 3.9: Guitar Pedal Enclosure

3.4 Programmable Logic (PL)

The main design artifacts of this project reside within the PL of the Zynq SoC. Vivado 2018.2 was used to develop the IP and block diagram that the project leverages. The hierarchical approach demonstrated in Figure 3.10 was used for designing the block diagram to group similar objects.

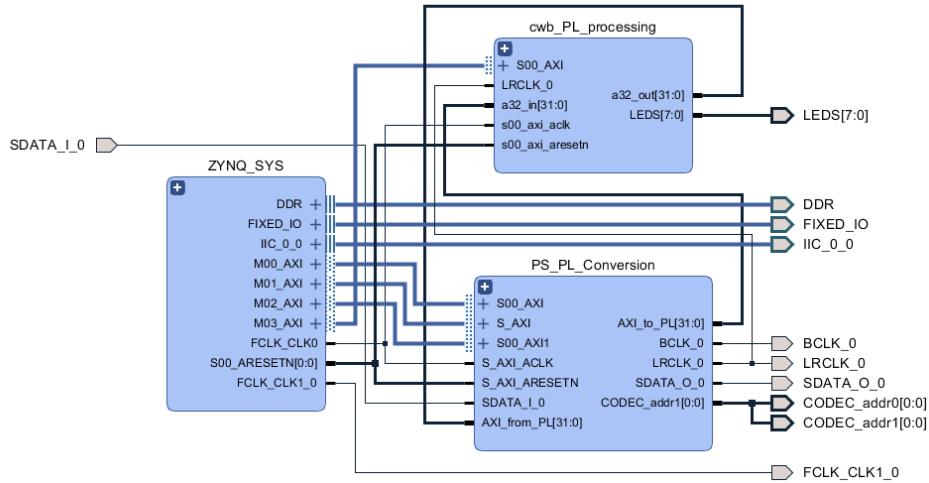


Figure 3.10: PL main block diagram

The ZYNQ SYS references the Zynq processor system which is used to set up the core and reset of the FPGA on the PL. The PS_PL_Conversion houses the IP mentioned in the previous sections that was provided as open source code to move input and output audio data. The last IP block, seen in Figure 3.11, houses the largest portion of code which is used to perform the audio signal processing: cwb_PL_processing.

The I2C_ADC_data_transfer AXI bus pushes the guitar pedal potentiometer data into the PL from the PS. The AXI data bus comes in 32 bit depth as well as 64 bit depth, however this project leverages the 32 bit depth version. This leads to additional and unneeded zero padding on all of the data coming into the FPGA and therefore can be removed prior to processing in

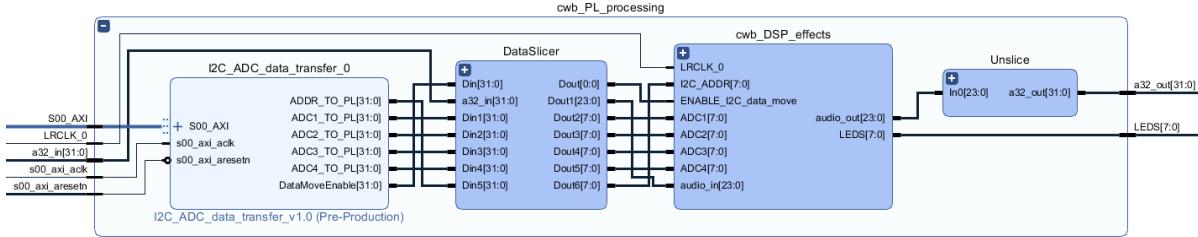
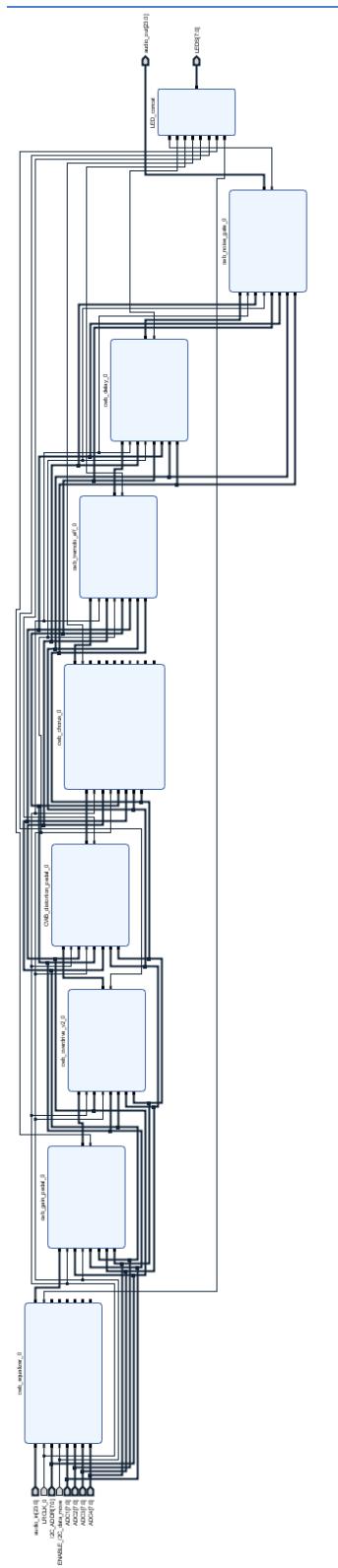


Figure 3.11: cwb_PL_processing hierarchy block diagram

order to improve efficiency and reduce unneeded resources. The zero padding on the data is removed using integrated IP called slicers in the DataSlicer block. The ADC data was sliced to the checked values as mentioned in Table 3.1 and the incoming audio signal from the guitar was sliced to the lowest 24 bits as an 8 bit left zero pad was utilized to send the data over the AXI bus. The audio signal is then processed according to the enabled algorithms in the cwb_DSP_effects block shown in Figure 3.12.

The last stage of the signal processing is to place a left zero pad of 8 bits on the processed audio signal so it can be sent back over the AXI bus into the PS and then back to the audio codec. Diving further into the cwb_DSP_effects block reveals the algorithms and their corresponding IP that are being implemented for the project. The algorithms that are being used in this project include amplitude vary, time varying and non-linear signal processing effects. A chapter is dedicated to each of these types of algorithm and each guitar effect implemented is explored in further detail. A standard approach to guitar pedal effect layout represented by Figure 3.13 was utilized in order to minimize unwanted signal chain distortion. This ordering can easily be modified in the FPGA fabric and resynthesized at any point in order to experiment with different tonalities.

Within each of the guitar effects, there is a control block block that takes in the parsed I2C data to update the algorithm parameters shown in Figure 3.14. The system works as there is I2C data being stream into all the guitar effects. The control loop processes if the streamed

Figure 3.12: `cwb_DSP_effects` block diagram

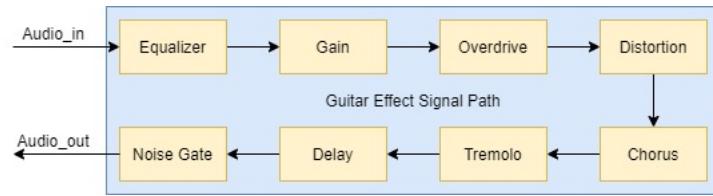


Figure 3.13: Guitar Effects signal chain

parameters apply to that specific algorithm. When the data is valid and the address of the guitar pedal matches the address of the algorithm block, register values within the algorithm block are updated and these parameters are referenced in the tuning parameters within the algorithm.

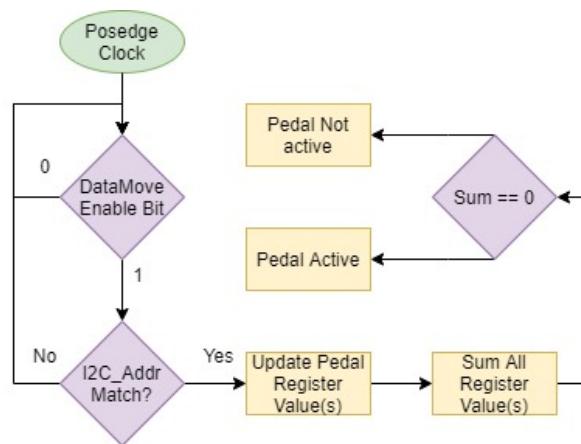


Figure 3.14: I2C data algorithm parameter control loop

Chapter 4

Distortion Effects

4.1 Gain Effect

Gain is an effect that is common in rock, metal and country where the guitar signal is desired to be much larger going into the amplifier. The driven signal can cause the amplifier circuit in the guitar amplifier to become saturated causing desirable distortion of the signal. Gain can be found incorporated in a guitar amplifier or found as a stand alone guitar pedal. In the scope of this project, a gain pedal algorithm was developed that took the input signal and amplified the signal. This effect differs from that of a standard pre-amp pedal due to there being no tone or equalization options as these option will later be incorporated into another guitar pedal for this project. The only input into the effect is a gain parameter that is controlled to a potentiometer on the guitar pedal as shown in Figure 4.1.

The gain term serves to boost a weak signal coming from the guitar or boost a guitar with lower pick-up output like a vintage Fender with a signal coil set up. The amount of gain that

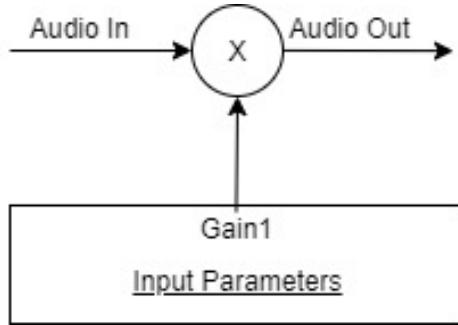


Figure 4.1: Gain effect block diagram

is used to increase the input signal is related to the output as:

$$y[n] = x[n] * Gain1 \quad (4.1)$$

The gain that is being implemented through multiplication of the incoming signal by a fixed value across the entire input range. The parameter is set as a fixed-point Q4.4 giving it a least significant bit resolution of 1/16 and a total range of 0 to 15.9375 which corresponds to a input of 0x00 to 0xFF. The gain algorithm in Figure 4.2 is simple to implement in fixed point arithmetic.

A simple test bench was generated to show that the input signal coming into the gain block was increased by the proper parameter amount depending on the gain input. Two different cases were demonstrated with the same input signal and varying the gain parameter from a low input to a high input. The output of the gain algorithm was fixed in the simulation to show the output in the range of -45000 to 450000. In the first test case shown in Figure 4.3, a gain of 15 was applied to the input signal with a peak to peak amplitude of 30000 which resulted in an output signal with a peak to peak amplitude of 450000. The last test case in Figure 4.4 featured the same input signal with a gain value of 1 which confirmed that a signal would pass through the gain block without any modification.

The high gain and low gain simulation result show that the gain effect block operates as

```

81  //-----
82  //-----  

83  //-----  

84  //-----  

85  //-----  

86  //-----  

87  //-----  

88  //-----  

89  //-----  

90  //-----  

91  //-----  

92  //-----  

93  //-----  

94  //-----  

95  //-----  

96  //-----  

97  //-----  

98  //-----  

99  //-----  

100 //-----  

101 //-----  

102 //-----  

103 //-----  

104 //-----  

105 //-----  

106 //-----  

107 //-----  

108 endmodule

```

Figure 4.2: Gain effect algorithm

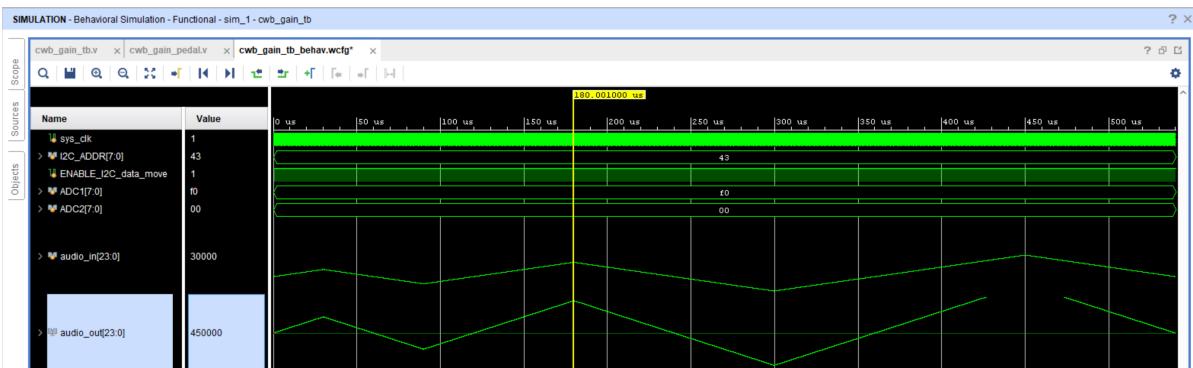


Figure 4.3: Gain effect with high gain input 0xF0 corresponding to a gain of 15

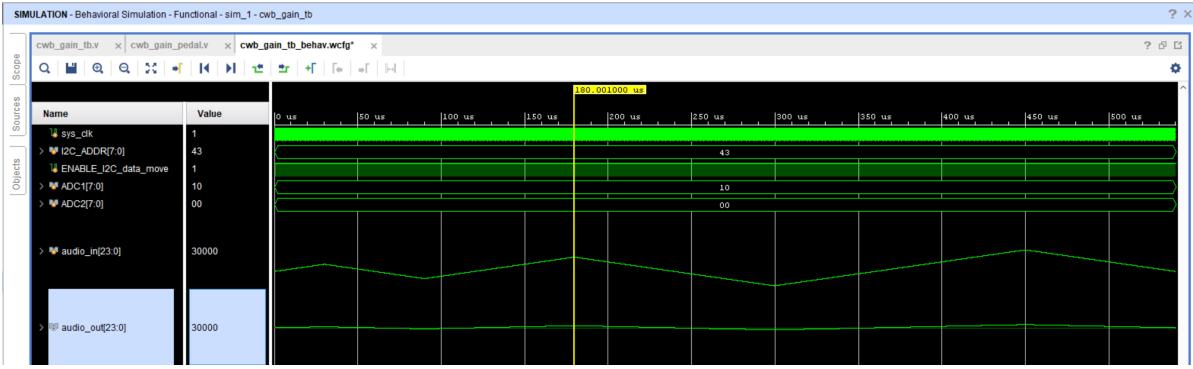


Figure 4.4: Gain effect with low gain input 0x10 corresponding to a gain of 1

intended. After experimental testing, it was found that the gain pedal with a Q4.4 format over-saturated the amplifier too much so the gain block was dialed back to a Q2.6 format.

4.2 Overdrive Effect

The overdrive effect as mentioned earlier is an effect of a signal being driven over a given threshold whether that be the headroom of tube amps or an electronic limiter. When a signal is driven beyond this threshold, two common clipping instances can occur: hard and soft clipping [25]. Soft clipping occurs when the input signal is greater than a specified threshold and the resulting signal which is greater than the threshold is compressed at the top end such that the peaks above the threshold are lowered to below the threshold. Hard clipping utilizes a different approach in that any input values that exceed the threshold limit cause the output of the signal to be equal to the threshold limit value. Hard clipping results in more distinct cutoffs than soft clipping and therefore adds more harmonics to the output signal resulting in a darker and harsher sound [26]. The visual difference between hard clipping and soft clipping is represented in Figure 4.5

In this project, a hard clipping overdrive pedal is featured due to the simplicity and minimal utilization of resources on the FPGA where a soft clipping algorithm would have used more

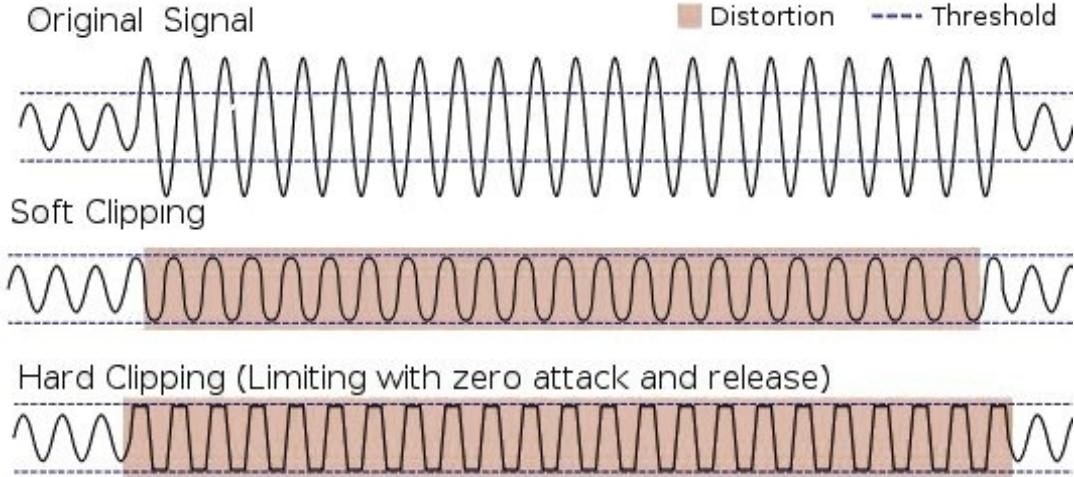


Figure 4.5: Example of hard and soft clipping for an audio signal [5]

DSP resources to account for the gain smoothing when the input signal crosses the threshold limit. The hard clipping circuit shown in Figure 4.6 will take in the audio signal and apply some amount of gain to the signal. A comparator is used to compare the threshold limit to the output of the boosted signal. If the boosted signal is still below the threshold, the comparator will output the boosted audio signal as the audio output. If the boosted audio signal is above the threshold limit, the comparator will output the value of the threshold limit as the audio output. This will cause a hard distortion sound due as well as add in odd harmonics relative to the main frequency components of the input signal. [27]. The block diagram describing this process can be found below.

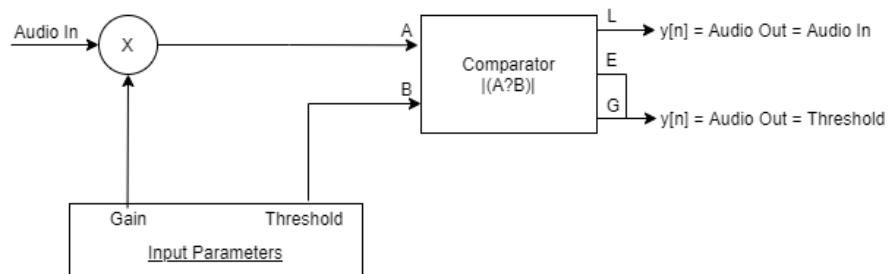


Figure 4.6: Hard clipping block diagram

For a positive signed input $x[n]$ into the overdrive effect, the output is:

$$y[n] = \begin{cases} \text{Threshold} & x[n] * \text{Gain} \geq \text{Threshold} \\ x[n] * \text{Gain} & x[n] * \text{Gain} < \text{Threshold} \end{cases} \quad (4.2)$$

For a negative signed input $x[n]$ into the overdrive effect, the output is:

$$y[n] = \begin{cases} x[n] * \text{Gain} & x[n] * \text{Gain} > -\text{Threshold} \\ -\text{Threshold} & x[n] * \text{Gain} \leq -\text{Threshold} \end{cases} \quad (4.3)$$

The algorithm which is implemented in order to create the overdrive effect, shown in Figure 4.7, is no more than an 'if-else' statement used to implement the hard clipping portion of the circuit.

The overdrive effect was written in Verilog and an associated test bench was written to validate the correct output of the hard clipping overdrive algorithm. The simulation varied only the gain parameter as the gain circuit was already verified isolating any resulting error to the threshold parameter. The overdrive threshold was held constant as a gain of 8 was applied in the first half of the simulation and a gain of 1 was applied in the latter half.

The high gain and low gain simulation result shown in Figure 4.8 verifies that the overdrive effect block operates as intended. When the gain was set higher in the first instance, the output waveform was noticeably clipped whereas the lower gain test shows that the boosted signal amplitude less than the threshold was passed without clipping. This verified the desired performance of the overdrive circuit. After experimental testing it was found that similar to the gain pedal, the Q4.4 gain stage was too much for the amplifier to handle and the gain parameter was adjusted to a Q2.6 format as well.

```

100 //-----
101 always @ (posedge sys_clk) begin //Control the output audio and modification
102     OverdriveLED = PedalON;
103
104 //Let ADC2reg = Gain2 with zero padding
105 //GAIN = audio_in*ADC1reg; //Q24.0*Q4.4 = Q28.4
106 //Thresh ~ ADC2
107     THRESH = {4'h0,ADC2reg, 12'h0};//Use ADC drive to determine Threshold for cutoff
108
109
110 if(PedalON==1)begin
111     if(audio_in[23] == 1'b0) begin //Positive gain
112         Gain1 = audio_in*ADC1reg; //Q24.0*Q2.6 = Q26.6
113         GAIN = Gain1[29:6];
114         if(GAIN>THRESH)begin
115             audio_out = THRESH;
116         end
117         else begin
118             audio_out = GAIN;
119         end
120     end
121     if(audio_in[23] == 1'b1) begin //Negative gain
122         tempAudioin = -audio_in;
123         Gain1 = tempAudioin*ADC1reg;
124         tempAudioout = Gain1[29:6];
125         GAIN = -tempAudioout;
126         if(tempAudioout>THRESH)begin
127             audio_out = -THRESH;
128         end
129         else begin
130             audio_out = GAIN;
131         end
132     end
133 end

```

Figure 4.7: Overdrive hard clipping algorithm

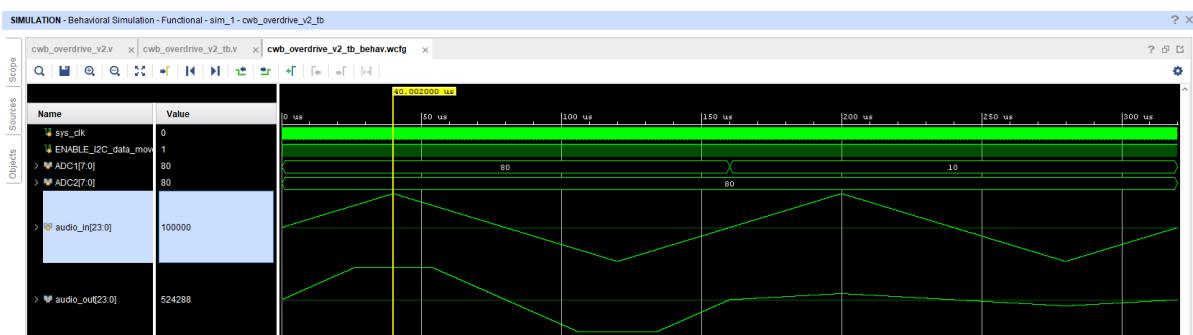


Figure 4.8: Overdrive test bench results

4.3 Distortion Effect

Distortion functions are similar to the overdrive in that the resulting signal is modified with respect to some given amplitude threshold. However, the distortion acts over a wider range of thresholds whereas overdrive is often a singular threshold. There are many different forms of distortion effects that are non-linear targeted for guitar such as cubic distortion, arc-tan distortion and absolute value. These algorithms lend themselves well to floating point arithmetic but are impractical to implement on an FPGA due to resource utilization as well as the restricting input ranges to values from -1 to 1 in most cases [28, 29].

$$y = \frac{x}{1 + |x|} \quad (4.4)$$

$$y = \begin{cases} -\frac{2}{3} & x \leq -1 \\ x + \frac{x^3}{3} & -1 < x < 1 \\ \frac{2}{3} & x \geq 1 \end{cases} \quad (4.5)$$

$$y = \tan^{-1}(x) \quad (4.6)$$

Another common method to create a digital distortion effect for electric guitar is to use a large lookup table and compare the incoming audio signal to the look up table. The audio signal will inevitably fall in between two different audio range bins. The signal will then be quantized to either of the two closest range bins depending on the encoding scheme. This method is well suited to FPGAs that have large look up table capabilities but being able to be adjusted is not always a parameter that can be tuned to various ranges like an analog distortion pedal [30]. In order to accommodate the tunability requirement associated with this project,

a scalable quantization strategy was utilized. The algorithm seeks to have two main tunable parameters: level and distortion.

The level parameter sets the upper limit on the input signal similar to that of the hard clipping overdrive circuit. This results in any signal that is greater than or equal to the level is quantized to the value of the level input. The distortion parameter works in conjunction with the audio range bins to determine the spacing between the range bins. The algorithm uses 40 positive and 40 negative audio range bins for quantization. Through empirical testing it was found that 80 audio range bins provided sufficient distortion without sacrificing too much tone. The algorithm was first developed in MATLAB using a summation of sine waves at different frequencies for the input signal. Distortion range bins were calculated relative to the maximum threshold parameter and the distortion parameter.

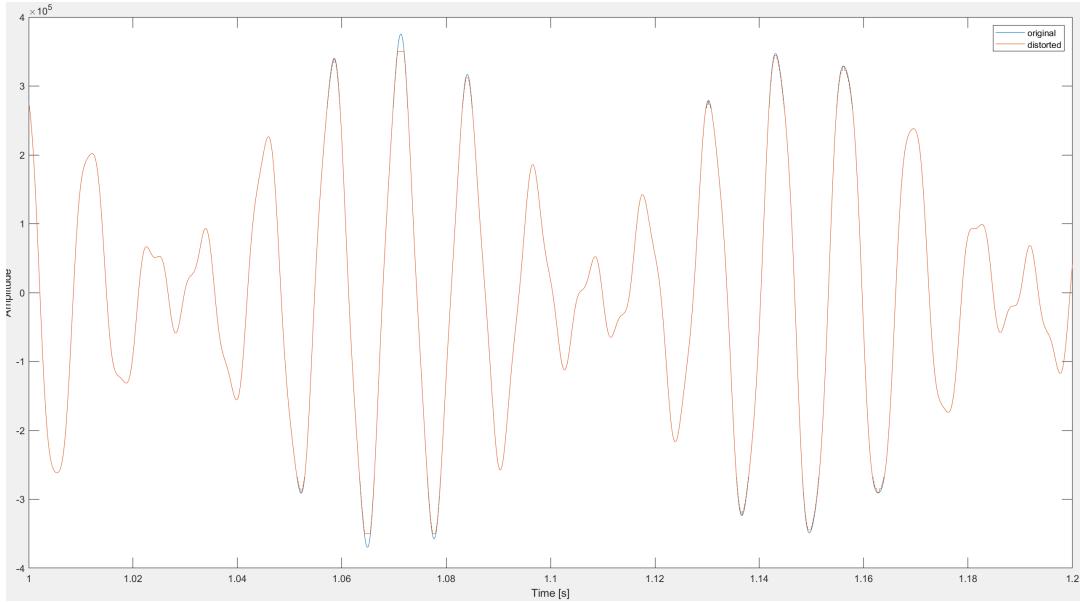


Figure 4.9: Quantization Distortion Effect in MATLAB simulation - Small Quantization Regions

In the case where little distortion is applied, the range bin spacing is relatively small in Figure 4.9 when compared to the large distortion parameter that was used in Figure 4.10 where

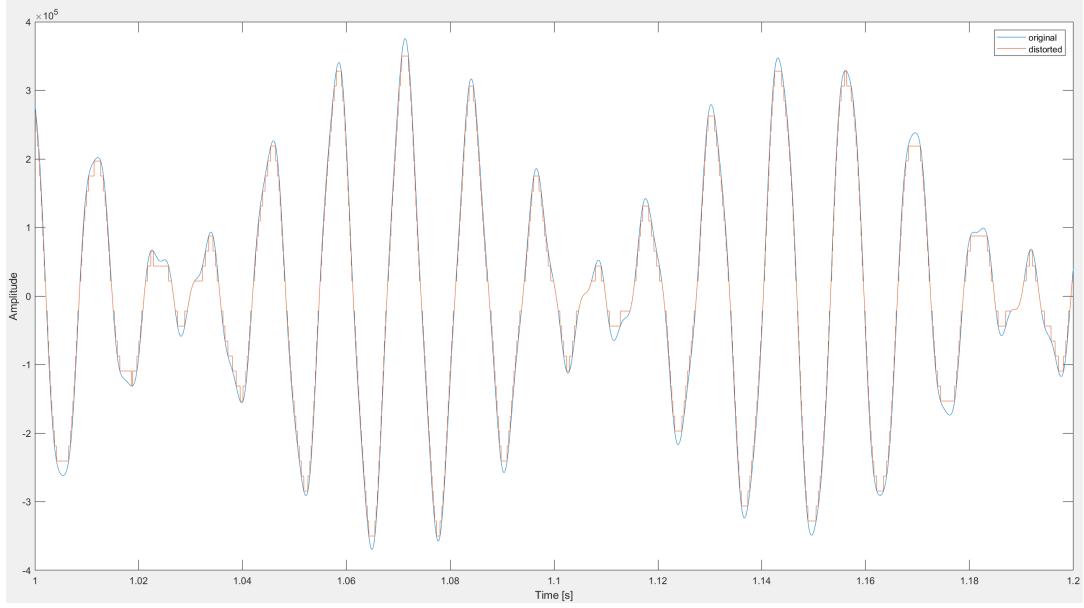


Figure 4.10: Quantization Distortion Effect in MATALB simulation - Large Quantization Regions

the audio range bin spacing is much larger. The audio range bins are calculated and expanded from the value of the level value instead of from zero. This effect can be demonstrated by the MATLAB simulations as well noting that in the case of small distortion values, distortion of the signal occurs greater than $|2.75 * 10^5|$. In the case of the large distortion value, it can be seen that the range bin spacing is much greater and there is more quantization occurring for values greater than $|.25 * 10^5|$. This is due to a combination of setting the quantizing bin priority from the level value instead of zero as well as allowing the range bins to change spacing relative to the level as seen in Figure 4.11.

When porting the MATLAB code to Verilog the scaling parameter that calculated the audio range bin spacing had to be converted from a floating point division to a fixed point multiplication. This was achieved by first using the two input parameters that controlled the level and distortion spacing parameter. The two parameters are 8 bit Q8.0. Combining this audio range bin calculation paired with the 80 total range bins offered the most audibly pleasing tone. A

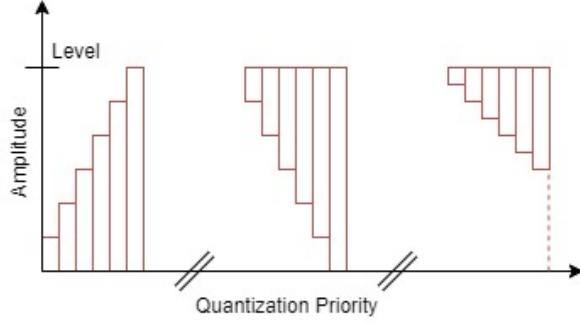


Figure 4.11: Distortion quantization audio range bin configuration: Zero based large bins (left) Level based large bins (center) Level based small bins (right)

portion of the positive and negative comparator style distortion algorithm is represented in Figure 4.12

A test bench was created to ensure proper operation of the distortion algorithm. Three test cases were verified with the test bench by varying the threshold and distortion coefficient inputs. The first case has a high threshold and a small distortion coefficient. The second case has a low threshold and a small distortion coefficient. The third case has a low threshold and a large distortion coefficient.

In the first case of Figure 4.13 it can be seen that the high threshold limit coupled with the small distortion ranges creates little if any distortion. When the distortion input was held constant and the level threshold was lowered in the second case, it can easily be seen that the input signal is clipped and other small amounts of quantization take place in the local area around the clipping. In the last case, large amounts of distortion can be seen in the clipped region as well as in the previously clean signal in case two as the distortion coefficient was increased and the level threshold was held constant. The three cases together serve to demonstrate that the distortion algorithm is properly functioning as designed.

```

177     if(PedalON == 1) begin //Start pedal ON
178     //DistortionLED = PedalON;
179     //-----Positive-----
180     if(audio_in[23] == 0) begin
181         if (audio_in >= level) begin
182             audio_out = level;
183         end
184         else if (audio_in >= DIST1 && audio_in < level) begin
185             audio_out = DIST1;
186         end
187         else if (audio_in >= DIST2 && audio_in < DIST1) begin
188             audio_out = DIST2;
189         end
190         else if (audio_in>=DIST3 && audio_in<DIST2) begin
191             audio_out = DIST3;
192         end
193         else if (audio_in>=DIST4 && audio_in<DIST3) begin
194             audio_out = DIST4;
195         end
196     //-----Negative-----
197     else if (audio_in[23] == 1) begin
198         if (-audio_in >= level) begin
199             audio_out = -level;
200         end
201         else if (-audio_in >= DIST1 && -audio_in < level) begin
202             audio_out = -DIST1;
203         end
204         else if (-audio_in>=DIST2 && -audio_in<DIST1) begin
205             audio_out = -DIST2;
206         end
207         else if (-audio_in>=DIST3 && -audio_in<DIST2) begin
208             audio_out = -DIST3;
209         end
210         else if (-audio_in>=DIST4 && -audio_in<DIST3) begin
211             audio_out = -DIST4;
212         end
213     end
214     ...
215     ...
216     ...
217     ...
218     ...
219     ...
220     ...
221     ...
222     ...
223     ...
224     ...
225     ...
226     ...
227     ...
228     ...
229     ...
230     ...
231     ...
232     ...
233     ...
234     ...
235     ...
236     ...
237     ...
238     ...
239     ...
240     ...
241     ...
242     ...
243     ...
244     ...
245     ...
246     ...
247     ...
248     ...
249     ...
250     ...
251     ...
252     ...
253     ...
254     ...
255     ...
256     ...
257     ...
258     ...
259     ...
260     ...
261     ...
262     ...
263     ...
264     ...
265     ...
266     ...
267     ...
268     ...
269     ...
270     ...
271     ...
272     ...
273     ...
274     ...
275     ...
276     ...
277     ...
278     ...
279     ...
280     ...
281     ...

```

Figure 4.12: Distortion effect quantization process: Positive value quantization (top) and negative value quantization (bottom)

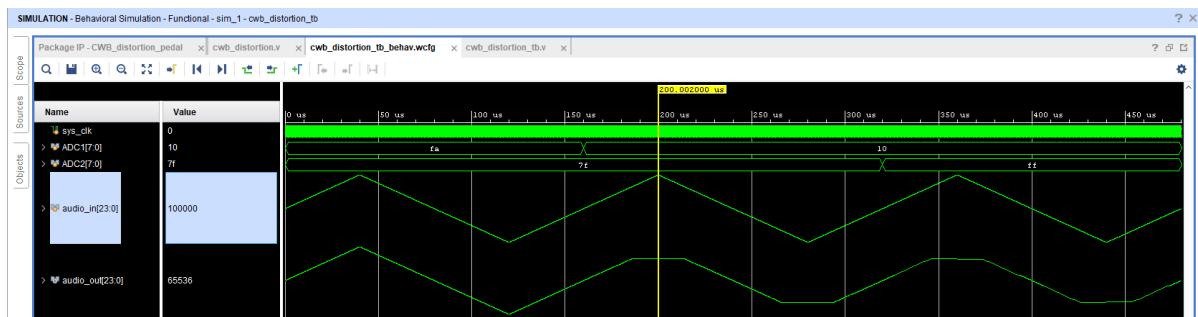


Figure 4.13: Distortion test bench

Chapter 5

Time Varying Effects

5.1 Delay Effect

Delay is a time based effect that buffers an incoming signal and after a fixed amount of time, passes the circularly buffered signal to the output. This effect can give the appearance of multiple notes being played and add complexity to a song when in actuality the added notes are delayed repetitions of the input signal. There are two main options in terms of delay effects for guitar: infinite repetition or a finite amount of repetitions as can be seen in Figure 5.1. The infinite repetition consists of using an infinite impulse response (IIR) style design which circularly buffers a sum of delayed variants. The finite repetition consists of using a corresponding number of delay banks in a finite impulse response (FIR) implementation. In this project, a single repetition FIR delay effect is used as well as an IIR delay effect.

The effect utilizes three parameters that modify the delay algorithm: Dry/Wet, Delay, and IIR/FIR. The Dry/Wet is used to control the gain of the data being circularly buffered where G takes values from 0 to .99 to ensure stability. The delay parameter is used to index the maximum indexing value allowed in the circular buffer where M represents the number of

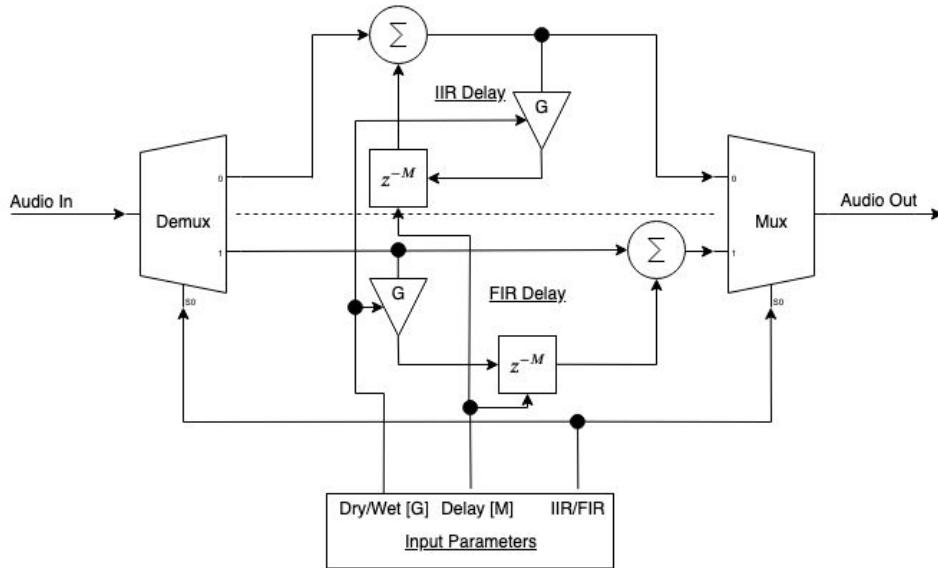


Figure 5.1: Delay Block Diagram

delay units. The IIR/FIR parameter is used to switch between the IIR and FIR guitar effect.

The circuit output equation can be mathematically represented as:

$$y[n] = \begin{cases} x[n] + G * x[n - M] & IIR/FIR = 0 \\ x[n] + G * y[n - M] & IIR/FIR = 1 \end{cases} \quad (5.1)$$

This effect can also be represented more commonly in terms of a difference equations as well as a Z transform which is evident from the block diagram where the FIR delay effect is represented as:

$$Y(z) = X(z) + G * z^{-M} * X(z) \quad (5.2)$$

$$Y(z) = X(z) * (1 + G * z^{-M}) \quad (5.3)$$

$$H(z) = (1 + G * z^{-M}) \quad (5.4)$$

and the IIR delay effect is represented as:

$$Y(z) - G * z^{-M} * Y(z) = X(z) \quad (5.5)$$

$$Y(z) * (1 - G * z^{-1}) = X(z) \quad (5.6)$$

$$H(z) = \frac{1}{(1 - G * z^{-1})} \quad (5.7)$$

In order to buffer the data, a dual port block RAM (BRAM) IP element was used. Using the Vivado IP tool, the BRAM was created as opposed to using inference or instantiation. The BRAM was designed with 24 bit width and 16 bit depth in order to account for circularly buffering the input audio signal for up to 65356 samples which at the clock rate of approximately 50kHz would allow for more than 1.3 seconds of total delay capability. The algorithm was then written calling the BRAM generated IP. The indexing signals were created using a simple counter scheme and offset using between the read and write pointers on initialization. If the pedal is active the pedal will actively read out the BRAM data and output the modified audio signal. When the pedal is turned off, a value of zero will be written into the BRAM in order to eliminate previous data. This will eliminate the chance of turning on the delay effect with previous data coming through even after being turned off. The implementation of the BRAM and delay algorithm are demonstrated in Figure 5.2.

The algorithm was simulated with a test bench for the FIR and the IIR input cases to demonstrate performance capabilities. Figure 5.3 demonstrates in the first half of the simula-

```

132 //Start calling the delay into BRAM
133 if(PedalONDelay == 0) begin//Device OFF
134     Din = 24'b000000000000000000000000;
135     audio_out = audio_in;
136 end
137 else if(PedalONDelay == 1) begin
138     //read value from BRAM
139     tempRAM = Dout;
140     //BRAM_OUT = tempRAM; //DEBUG
141     audio_out = audio_in + tempRAM;
142
143 //Output statges
144 //FIR implementation -> only repeats one note and is working
145 if(isFIR == 1) begin
146     if(audio_in[23] == 0) begin //positive
147         temp_audio_in = audio_in;
148         temp_Din = temp_audio_in*ADC2reg;//Q24.0*Q0.8= Q24.8
149         Din = (temp_Din[31:8]);
150     end
151     else begin // negative
152         temp_audio_in = -audio_in;
153         temp_Din = temp_audio_in*ADC2reg;//Q24.0*Q0.8= Q24.8
154         Din = -temp_Din[31:8];
155     end
156 end
157
158 //IIR implementation -> continual feedback
159 else begin
160     Din_accum = audio_in+tempRAM;
161     if(Din_accum[23] == 0) begin //positive
162         temp_audio_in = Din_accum;
163         temp_Din = temp_audio_in*ADC2reg;//Q24.0*Q0.8= Q24.8
164         Din = temp_Din[31:8];
165     end
166     else begin // negative
167         temp_audio_in = -Din_accum;
168         temp_Din = temp_audio_in*ADC2reg;//Q24.0*Q0.8= Q24.8
169         Din = -temp_Din[31:8];
170     end
171 end//End IIR
172 end //end pedal on case

```

Figure 5.2: Delay effect algorithm

tion window the FIR delay and the second half of the simulation window the IIR delay.

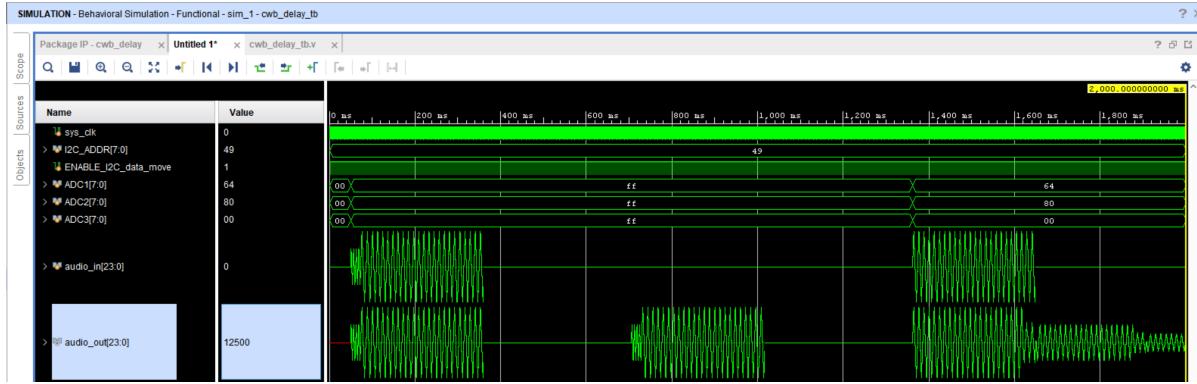


Figure 5.3: Delay effect simulation

5.2 Chorus Effect

Chorus is similar to the delay effect in that it takes input data and buffers an amount of time before adding time delayed version of the input signal to the output channel. The delay effect implements a fixed amount of delay time whereas the chorus effect varies the amount of time delay applied to the input signal [31]. This can be easily viewed in Figure 5.4 as a delay pedal will have a fixed delay unit of z^{-M} where the chorus effect with have a varying delay unit of $z^{-(M+R)}$ where R is a range of values. Typical values for $(M + R)$ fall in the range of 10 to 25 milliseconds and will give the effect of multiple instruments playing in near unison rate however off just slightly enough to be audible; thus, leading to an output that sounds like a chorus of instruments [32]. There are two major parameters used in controlling the chorus effect: the rate and the depth. The rate is the R parameter aforementioned and will set the upper bound of the low-frequency oscillator (LO) used for referencing the delayed output. The depth parameter G will control the attenuation of the delayed signal going into the block ram.

The LO used in order to read varying values in the circular buffer can often be designed

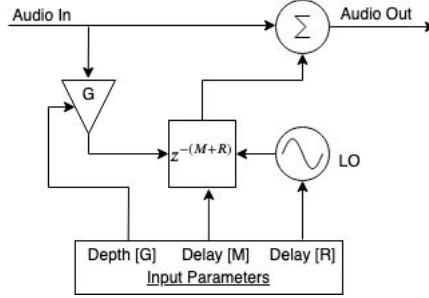


Figure 5.4: Chorus effect block diagram

as a triangular wave or a sine wave, for this project a triangular wave was easily developed using a count up/down architecture [33]. Further design consideration was taken in order to properly read out the values from memory without introducing too much unwanted distortion. Elegant methods to provide seamless transition between sample values range from developing a poly-phase filter or a fractional delay line leveraging Farrow polynomial filter structures [34]. These designs, while elegant, are relatively more complex to implement and can quickly use up resources much faster than a simple approach of nearest neighbor interpolation shown in Figure 5.5 which can be resource minimal and quick to implement [35]. Empirical testing found that an interpolation by a factor of $L=8$ to reconstruct a value between two signals provided ample fidelity in the desired operating range of the chorus pedal. At higher frequencies, an interpolation factor of 16 could be used for fractional sample reconstruction pending less resource utilization. In order to implement the nearest neighbor interpolation scheme, a tapped delay line of 128 samples were fed into an interpolation stage with a factor of 8. The LO works in an up/down counter mode and the value of the counter determines the position within the pseudo-fractional delay line where to take the data.

The interpolation by a factor of 8 on a delay line of 128 points gate a total of 1024 values for the LFO to operate through in order to give the chorus voicing as many options as possible extending into the operating ranges of flangers and echos without any feedback components.

```

451
452     always @(posedge sys_clk) begin
453         case(interp_index)
454             16'h0000:   Interpolation_Accum <= (DelayRead0>>>3)*8;
455             16'h0001:   Interpolation_Accum <= (DelayRead0>>>3)*7 + (DelayRead1>>>3)*1;
456             16'h0002:   Interpolation_Accum <= (DelayRead0>>>3)*6 + (DelayRead1>>>3)*2;
457             16'h0003:   Interpolation_Accum <= (DelayRead0>>>3)*5 + (DelayRead1>>>3)*3;
458             16'h0004:   Interpolation_Accum <= (DelayRead0>>>3)*4 + (DelayRead1>>>3)*4;
459             16'h0005:   Interpolation_Accum <= (DelayRead0>>>3)*3 + (DelayRead1>>>3)*5;
460             16'h0006:   Interpolation_Accum <= (DelayRead0>>>3)*2 + (DelayRead1>>>3)*6;
461             16'h0007:   Interpolation_Accum <= (DelayRead0>>>3)*1 + (DelayRead1>>>3)*7;
462             16'h0008:   Interpolation_Accum <= (DelayRead1>>>3)*8 + (DelayRead2>>>3)*0;

```

Figure 5.5: Nearest Neighbor Interpolation L=8 Reconstruction Algorithm

As it would be tedious to continually write this algorithm out by hand and HDL for loops are messy for implementation, a scripting format was taken to generate the vectors presented. The formula takes in the interpolation factor and the number of delay taps desired and creates the interpolation vectors above which reduce time needed to create the vectors by hand. This makes scaling up the tapped delay line or the interpolation factor for reconstruction trivial. The only limitation one the scripting method is that it does not approximate the resource utilization of the FPGA therefore it can be easy to design the simple nearest neighbor interpolation but it may use too many resources to fit in the FPGA with the other effects. After designing the chorus effect with the aforementioned 128 taps and interpolation factor of L=8, the overall system resource utilization was higher than that of a Farrow or poly-phase filter but the development time was much quicker. The chorus pedal was simulated for long and short delays as well as long and short LFO frequencies as shown in Figure 5.6. In the first half of the simulation window, a short delay with a relatively quick LFO frequency was used and a longer delay with a slower LFO frequency and some attenuation in the second half of the simulation window. The periodic repetition in the first half of the simulation shows the LFO delay properly activating.

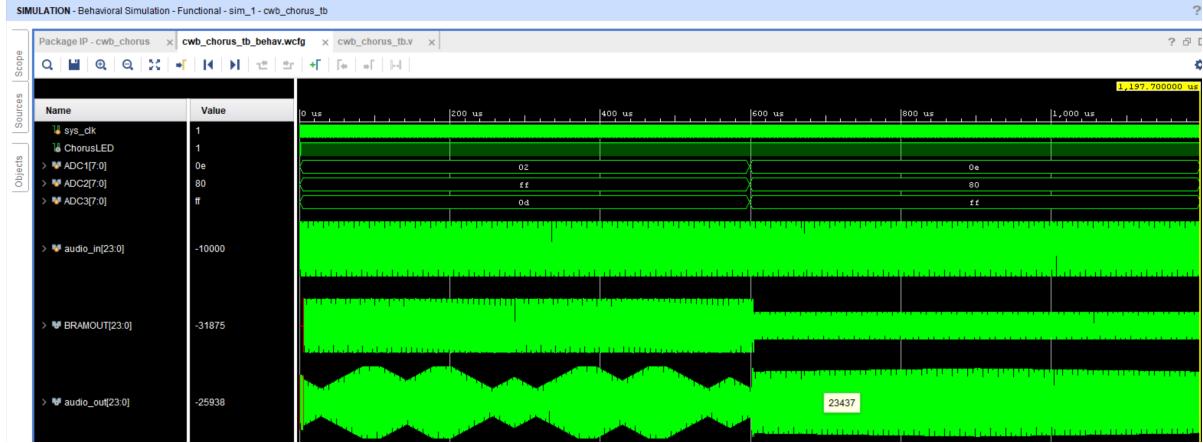


Figure 5.6: Chorus Effect Simulation

5.3 Tremolo Effect

Tremolo is a modulation effect that utilizes amplitude modulation (AM), visualized in Figure 5.7, to modulate the incoming audio signal. In communications systems, AM modulation utilizes a high carrier frequency for modulation to move the baseband signal to a bandpass signal. In the case of the tremolo, the modulating signal frequency is less than the input audio signal frequency, often in a range of 1Hz to 50 Hz.

The effect has two main parameters: tremolo frequency and waveshape as seen in Figure 5.8. The tremolo frequency is responsible for setting the clock rate that will be used to create the modulating signal. The clock generated from the tremolo frequency parameter drives an up-down style counter that creates a sawtooth waveform. The sawtooth waveform can be modified according to the waveshape input parameter. The shaping parameter can leave the sawtooth waveform shape unmodified or modify the sawtooth signal's shape towards that of a rectangular pulse.

The waveshape was first developed in MATLAB by creating a sawtooth waveform shape that would be created from an up-down counter [27]. This sawtooth waveform is easy to implement in hardware as it is the result from an incremental counter/timer within hardware

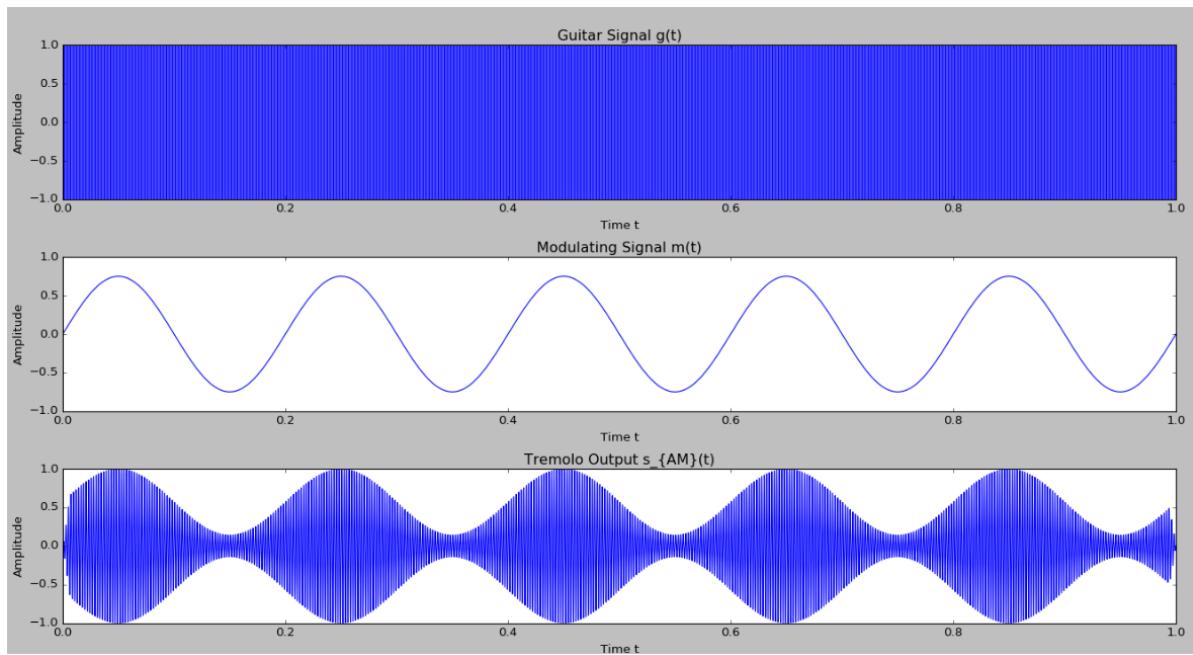


Figure 5.7: Example of tremolo AM [6]

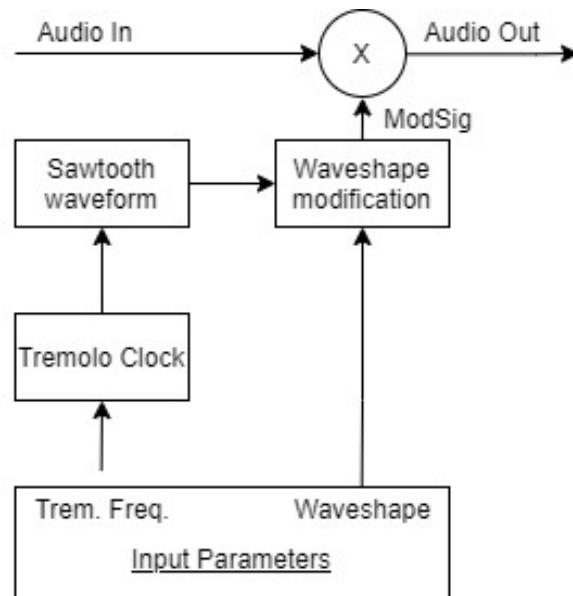


Figure 5.8: Tremolo block diagram

and requires minimal additional resources. A sinusoidal waveform could have been created but would have used many more resources than the sawtooth waveform selected. The up-down counter limit was set to 255 to make calculations simple using a Q8.0. The waveshaping process takes in the raw sawtooth waveform as an input and utilizes a comparator to assess upper and lower threshold limits.

$$x_{\text{waveshaping}}[n] = \begin{cases} \text{CounterLimit} & x_{\text{sawtooth}}[n] > \frac{\text{WaveShape} + \text{CounterLimit}}{2} \\ 1 & x_{\text{sawtooth}}[n] < \frac{\text{WaveShape} + \text{CounterLimit}}{2} \\ x_{\text{sawtooth}}[n] & \text{otherwise} \end{cases} \quad (5.8)$$

The waveshape parameter can take values from 0 to 255 which results in a square wave to an unmodified triangular wave. Various wave shape examples are demonstrated from MATLAB simulations for the waveshape parameter set to 255, 127, and 1 which are demonstrated in Figure 5.9, Figure 5.10, and Figure 5.11 respectively.

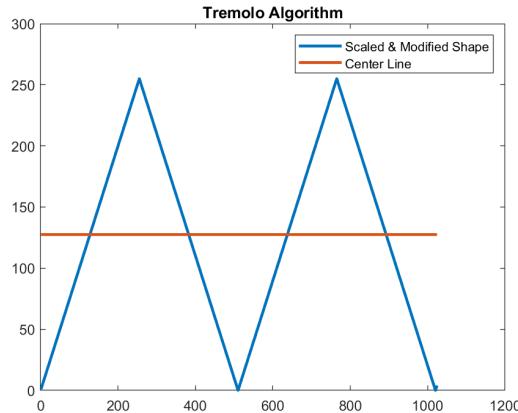


Figure 5.9: Tremolo Waveshaping value in MATLAB simulations: 255

The waveshaping parameter was then multiplied to the input signal resulting in a Q32.0 value. The highest 24 were passed onto the next algorithm in the chain. A test bench as

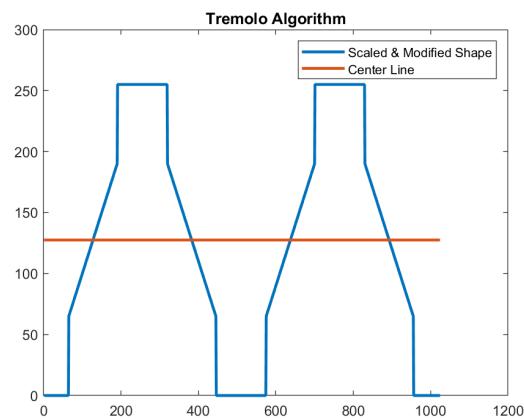


Figure 5.10: Tremolo Waveshaping value in MATLAB simulations: 127

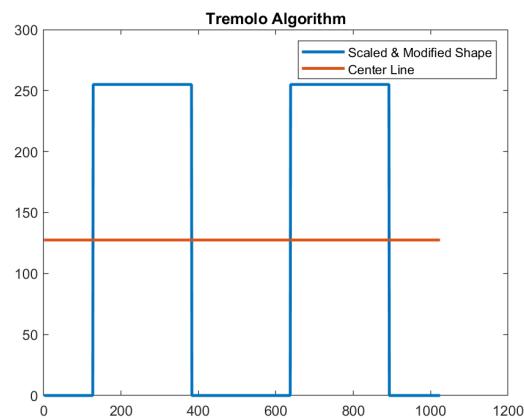


Figure 5.11: Tremolo Waveshaping value in MATLAB simulations: 1

seen in Figure 5.12 was created to test the tremolo effect with the first half of the simulation window demonstrating a triangular waveform modulation whereas the latter half of the simulation window shows the input signal being modulated by a waveform similar to the MATLAB demonstrated waveform for an input value of 127.

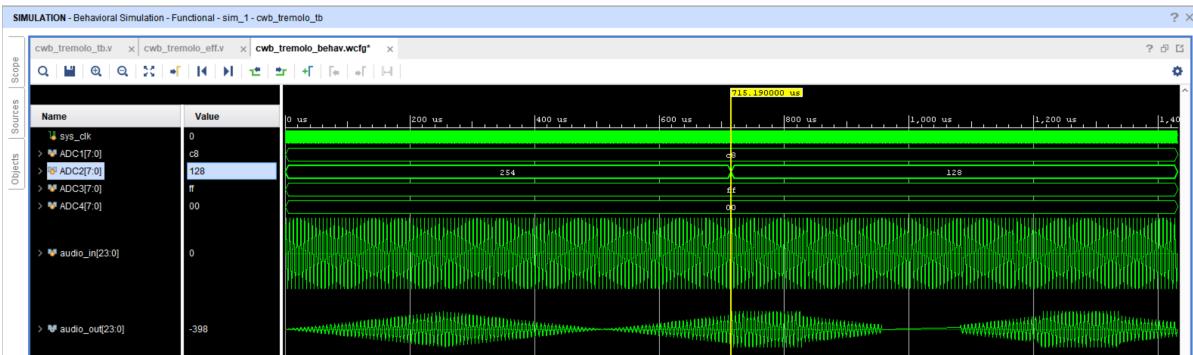


Figure 5.12: Tremolo test bench simulation

Chapter 6

Attenuation Effects

6.1 Noise Gate Effect

In a high gain audio chain, small noise can be noticeably amplified by the time it reaches the output stage. A noise gate serves to minimize the noise that passes through the effects chain by utilizing dynamic range compression for signals below a certain threshold and allows signals greater than the threshold to pass through as seen in Figure 6.1. When audio is able to pass through the effect with no alteration the gate is said to be open. Conversely, when the audio is attenuated the gate is said to be closed. The gate cannot close immediately (in only one clock cycle) as an instantaneous drop off or rise in audio level will result in unwanted harmonics. Therefore a scheme involving some delay aspects must be utilized in order to minimize these harmonics from changing amplitudes instantaneously.

One delay scheme which is useful in achieving the minimal instantaneous changes is to use what are referred to as attack, release, and hold times demonstrated in Figure 6.2. The attack time T_A is responsible for allowing the gate to open in a specific amount of time after a signal has been above the specified threshold. The hold time T_H is responsible for keeping the gate

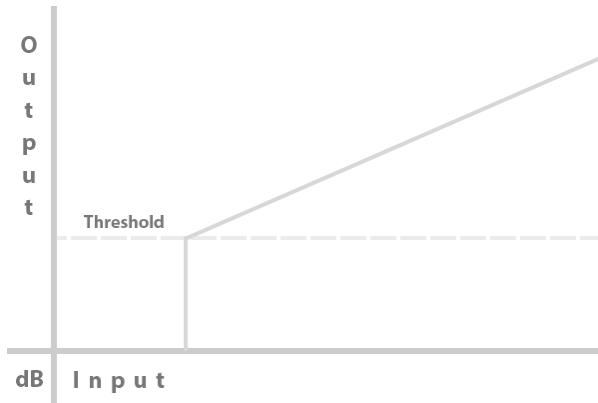


Figure 6.1: Noise gate response in dB [7]

open after the specified signal drops below the threshold. The release time T_R is responsible for closing the gate in a specific amount of time following the hold time.

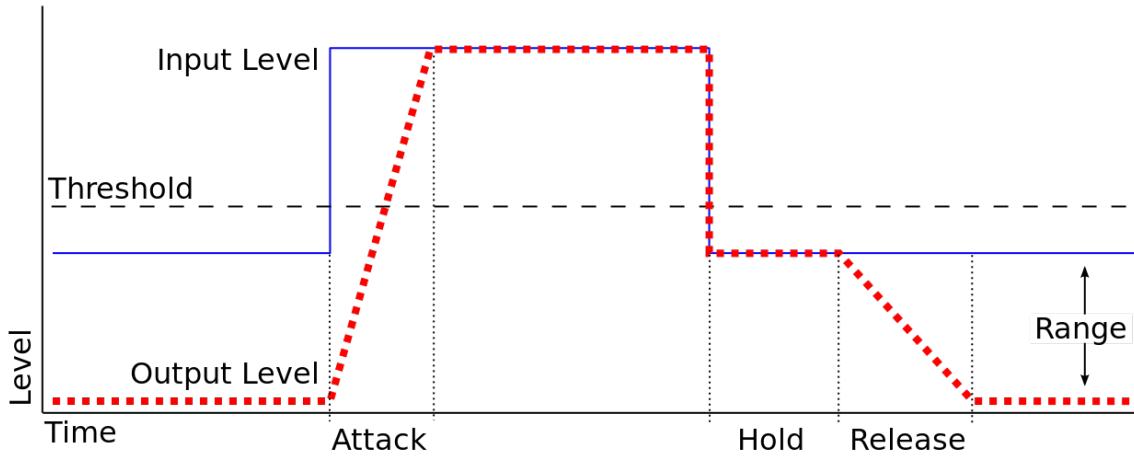


Figure 6.2: Noise gate timing parameters [8]

These parameters are summed up to control the noise gate and determine the rejection of unwanted audio levels. Giannoulis et al. presented a simple methodology for designing Dynamic range compressors [36]. MathWorks later used [36] as a reference in designing the MATLAB ‘noiseGate’ function [9]. The functional block diagram in Figure 6.3 was the basis for the design of this noise gate as it met all of the tuning parameter requirements however

it was developed in floating point arithmetic as well as using dB notation for the mathematical operations. In order to keep complexity and resource utilization down, the algorithm was adjusted to use fixed point notation on an FPGA for mathematical operations as well as pre-calculate certain values for the attack, release, and hold times as natural and base 10 logarithmic calculations on the FPGA would have been more costly in complexity [37].

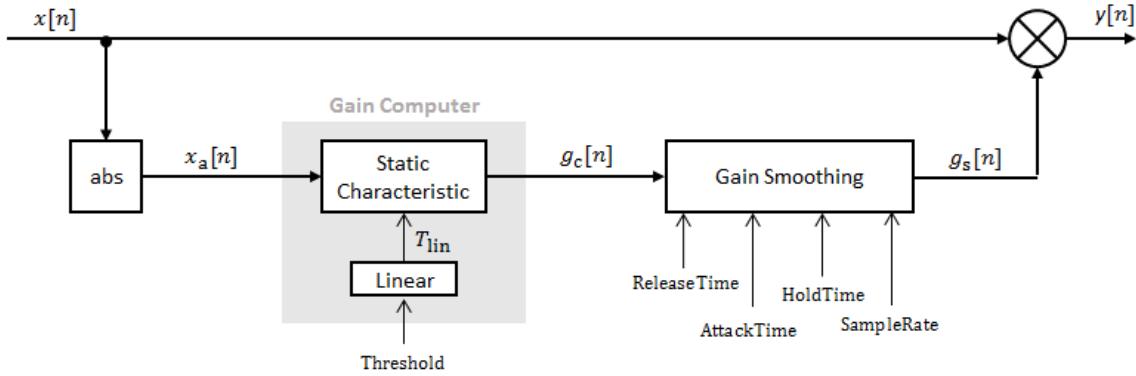


Figure 6.3: Noise gate block diagram [9]

Utilizing the block diagram for the noise gate presented, the following calculations are drawn as well as making note the fixed point notation utilized for design. The input audio signal, $x[n]$, is Q24.0 signed. The signal is then branched off and the magnitude of the signal is taken. Taking the magnitude of the purely real signal is equivalent to the absolute value giving the value at the $x_a[n]$ node as:

$$x_a[n] = |x[n]| \quad (6.1)$$

The magnitude of the signal is then passed into the Gain Computer where the magnitude is checked by a comparator. If the threshold is greater than the input to the gain computer then

the output, $g_c[n]$, is zero. Otherwise the output is one. This leads to a Q1.0:

$$g_c[n] = g_c(x_a[n]) = \begin{cases} 0 & x_a[n] < T_{lin} \\ 1 & x_a[n] \geq T_{lin} \end{cases} \quad (6.2)$$

Where T_{lin} is equal to the threshold set by the user input. The block diagram notation indicates that the threshold is originally in dB however for simplicity it is assumed that the threshold input can already be linear. Once the output of $g_c[n]$ is achieved it can be passed into the gain smoothing computer. Further care is taken in the conversion of the gain computer to fixed point as $g_s[n]$ Q1.23 takes:

$$g_s[n] = g_s(g_c[n]) = \begin{cases} \alpha_A * g_s[n-1] + (1 - \alpha_A) * g_c[n] & (C_A > T_H) \&& (g_c[n] \leq g_s[n-1]) \\ \alpha_R * g_s[n-1] + (1 - \alpha_R) * g_c[n] & (C_R > T_H) \&& (g_c[n] > g_s[n-1]) \\ g_s[n-1] & C_A \leq T_H \\ g_s[n-1] & C_R \leq T_H \end{cases} \quad (6.3)$$

Where C_R and C_A are the counters associated with the attack and release interval. The constants α_A and α_R are a function of the 90% rise and fall times needed for smooth transition needed minimize unwanted distortion according to:

$$\begin{aligned} \alpha_A &= e^{-\ln(9)/(F_S * T_A)} \\ \alpha_R &= e^{-\ln(9)/(F_S * T_R)} \end{aligned} \quad (6.4)$$

Where F_S is the associated sampling frequency with the audio 48100 Hz. The calculations to achieve the α parameters was obtained outside of the FPGA as rise, fall, and hold times are not easily calculated on the FPGA. Therefore the calculations were performed off chip. The

values were also converted to a known fixed point form to make more efficient calculations of the FPGA possible.

Table 6.1: Noise gate fixed constant parameter values

Parameter	T_H [ms]	T_R [ms]	T_A [ms]	F_S [Hz]
Value	.8	100	1	50,000

Table 6.2: Noise Gate calculated constant parameter values

Parameter	α_A	$(1 - \alpha_A)$
Decimal Value	0.957007078	0.042992922
Fixed Point Q1.23	0.1111010011111100110101	0.00001011000000011001010
Parameter	α_R	$(1 - \alpha_R)$
Decimal Value	0.999560652	0.000439348
Fixed Point Q1.23	0.1111111111000110011010	0.000000000000111001100101

To confirm the fixed point notation and calculations the derivation is shown below to ensure that the proper value ranges were selected in order to keep a valid signal. Taking the first line in the $g_s[n]$ equation and writing the fixed point notation for the terms yields:

$$g_s[n] = (Q1.23) * (Q1.23) + (Q1.23) * (Q1.0) \quad (6.5)$$

$$g_s[n] = (Q2.47) + (Q2.23) \quad (6.6)$$

Knowing that the resultants cannot be a two bit binary answer, we can shift the decimal point to the left by one bit yielding:

$$g_s[n] = (Q1.47) + (Q1.24) \quad (6.7)$$

In order to add the two values the decimal point must be aligned and therefore a zero

padding size 23 is added to the latter term:

$$g_s[n] = (Q1.47) + (Q1.47) \quad (6.8)$$

This verifies that the correct notation can be used in the algorithm and the assumed Q fixed point notation was properly placed. Once $g_s[n]$ is calculated, it is multiplied to the original input signal $x[n]$ yielding the output $y[n]$.

$$y[n] = x[n] * g_s[n] \quad (6.9)$$

This signal chain was simulated via a test bench with a monotone signal with increasing amplitude to test the noise compression over different ranges. The simulation in Figure 6.4 verifies that when the incoming audio signal is below the threshold for a set hold time, the signal is attenuated in a roll off fashion as well as when the signal recovers and is greater than the threshold.

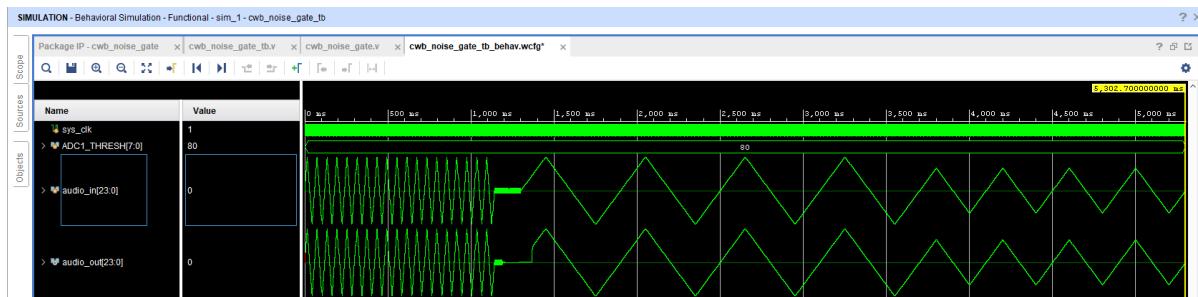


Figure 6.4: Noise gate test bench simulation

6.2 Equalizer Effect

Guitar effects can often add unwanted signal imaging into the desired signal. These signals fall into three broad categories depending on their frequency spectrum. A guitar that is tuned to

standard E tuning at A440 will have a dynamic audio range from 80Hz to 1200Hz excluding natural harmonics past 1200Hz. This broad range of frequencies is broken down into three main sub regions: low, medium, and high frequencies. The standard music terminology calls low frequencies bass, medium frequencies are called mids, and high frequencies are called treble [38]. Within each of these three sub categories, musicians can break down the ranges even further. This project scope only centers around equalizing bass, mids, and treble as demonstrated in Figure 6.5. In some cases, a bright and crisp (mids and treble) sounding signal can become dark and muddled (bass) after passing through distorting effects. In cases like this it is desirable to equalize the incoming signal before the distortion effect and drop amplitude of the frequencies in the bass region so when the distortion effect is applied, the overall frequency response is much flatter. Conversely, the EQ could also be used to boost the mids and the treble in order to account for the increase in the bass response [39, 40]. Equalizers are also used to account for issues when playing in certain environments and rooms or even to make the voicing of the guitar sound different. An example of the latter would be to make a humbucker pick up sound like a single coil pick up by dropping the bass and the treble while boosting the mids.

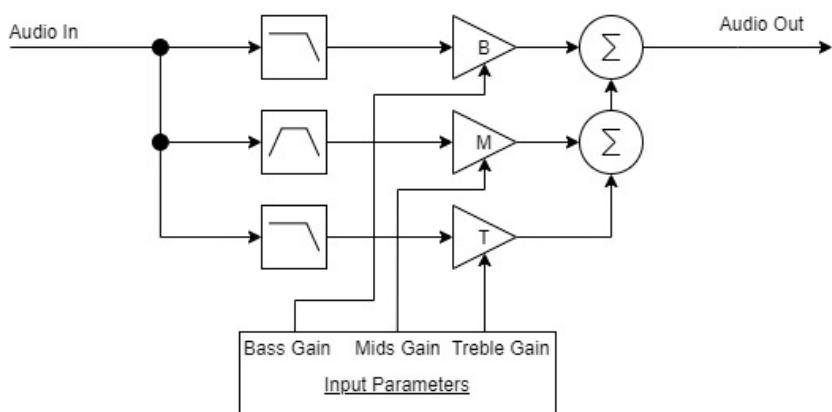


Figure 6.5: Equalizer Block Diagram

There is no definitive acceptance for the range of values associated with bass, mids, and

treble; however most sources give a range of values. A general guide line for most amp manufacturers has a bass response from 20Hz-300Hz, a mids response is typically centered around 600-700Hz, and a treble response contains 1200Hz and up. Different guitar tuning and tonalities of the instrument will give varying frequency responses so the design of this effect sought to stay close to these design criterion. This effect will use the three bands of frequencies and generate the corresponding filters with the three user inputs corresponding to the gain applied to the outputs from the filter banks. The gain parameters will have a Q0.8 format and then the resulting accumulations will be summed together and result in the output of the algorithm. MATLAB DSP tool box was used to create the necessary digital filters as well as evaluate their performance between fixed point implementation and floating point. MATLAB filterBuilder UI was then used to create the desired filter and create the HDL code to implement the necessary filtering. The Xilinx Zynq 7020 has 220 DSP slices which limits the overall filter sizes in addition to the performance of the filter roll off. The goal to design each of the three filters was to keep the DSP slice count under 50 units per filter which would still leave more than enough DSP slice resources for the other effects being implemented with less rigorous filtering requirements. The filter taps will all use a Q12.0 fixed point format and a symmetric FIR filter topology in order to minimize resource usage.

The low-pass filter for bass response shown in Figure 6.6 was designed to pass frequencies with a 1db pass band ripple and a 20dB attenuation in the stop band at 600Hz. This filter resulted in a total DSP slice utilization of 47 units and a 92 tap FIR filter. The band-pass mids filter was designed and shown in Figure 6.7 to have a 20dB attenuation at DC and 1600Hz with a .1dB ripple in the pass band from 600Hz to 800Hz. This filter also utilized a symmetric FIR topology and had a DSP slice utilization of 58 units and a total of 127 taps. Lastly, the high-pass treble filter was designed and demonstrated in Figure 6.8 to have a 20dB attenuation in the stop band up to 800Hz and a .1dB pass band ripple above 1600Hz. The filter utilizes a

direct for symmetric FIR filter with a DSP slice utilization of 45 units and a total of 90 taps. All of the filter frequency responses are presented for their ideal float point representation and their quantized response. All critical frequencies selected for the equalizer are grouped together in Table 6.3

Table 6.3: Filter Specifications for EQ Filter

Filter	Type	$F_{pass1}[\text{Hz}]$	$F_{pass2}[\text{Hz}]$	$F_{stop1}[\text{Hz}]$	$F_{stop2}[\text{Hz}]$	Pass Band Ripple [dB]	Stop Band Attenuation [dB]
Bass	LPF	0	N/A	600	N/A	1	20
Mids	BPF	600	800	DC	1600	.1	20
Treble	HPF	1600	N/A	800	N/A	.1	20

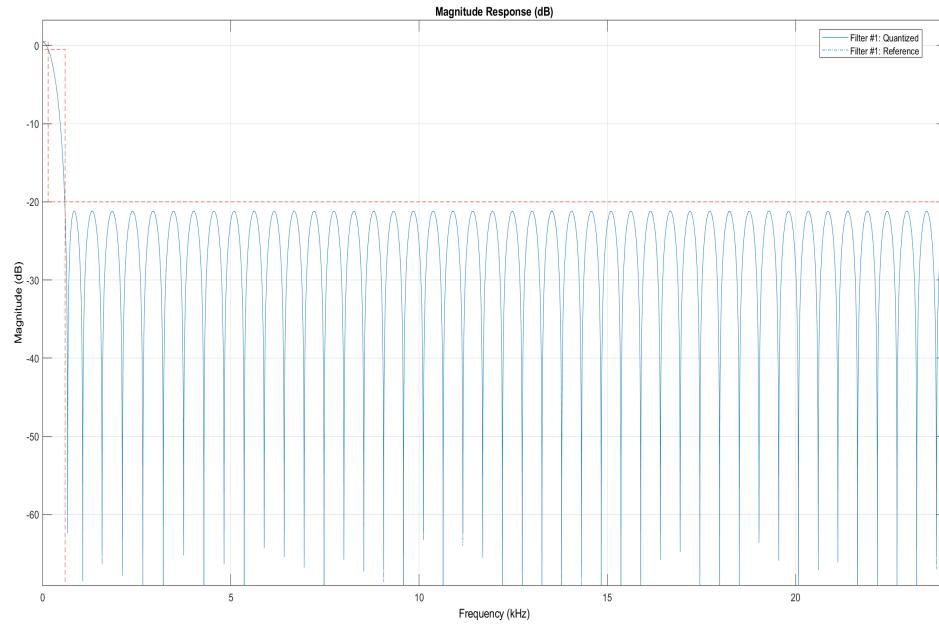


Figure 6.6: Low-pass bass filter response

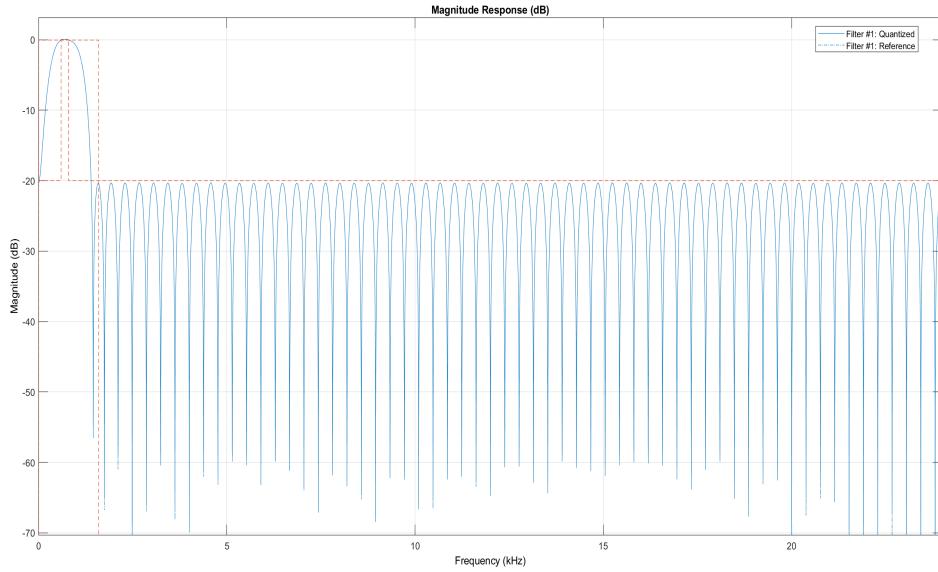


Figure 6.7: Band-pass Mids filter response

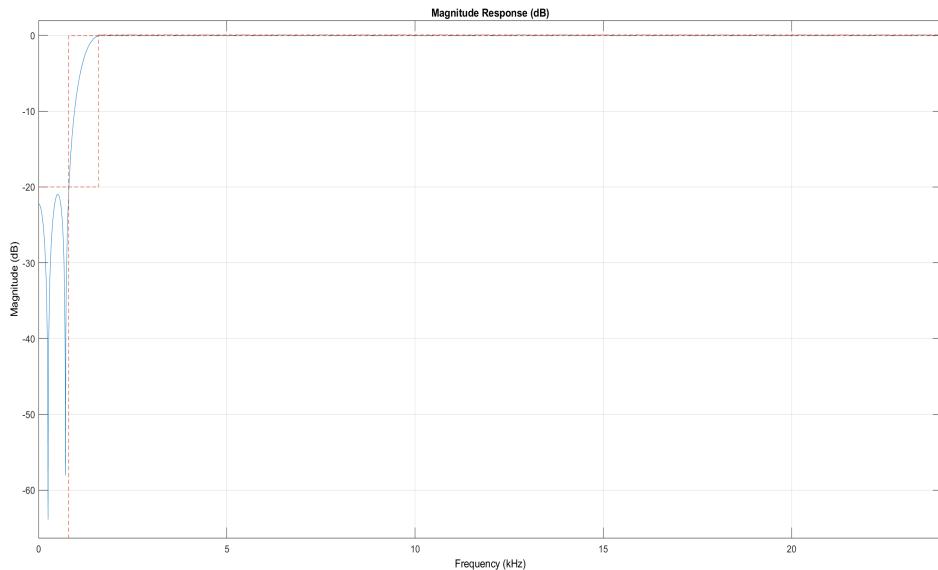


Figure 6.8: High-pass Treble filter response

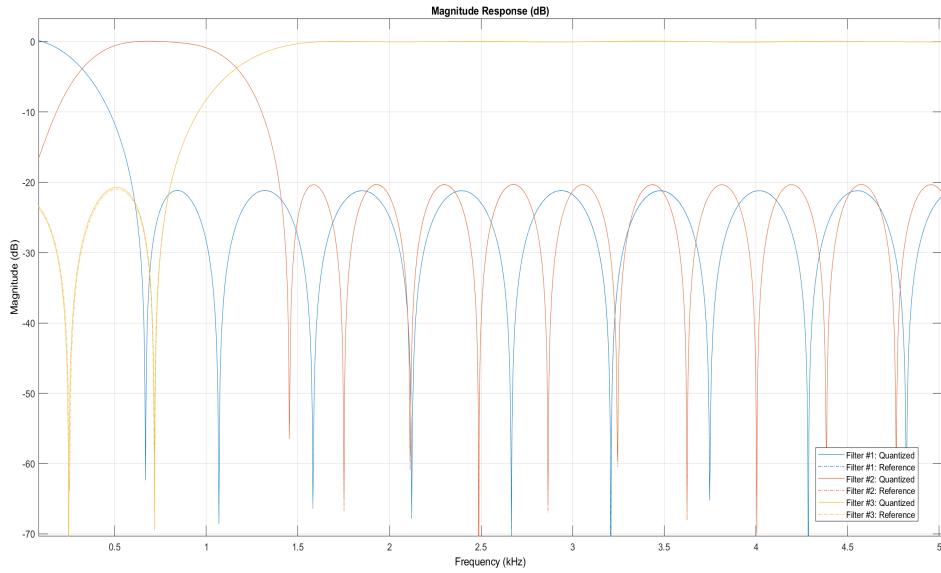


Figure 6.9: Equalizer filter response

When the three filters are plotted together using the MATLAB fvtool command and shown in Figure 6.9, it can be seen that the filters create a fairly flat frequency response in the operating frequency of the guitar. The HDL generated files from the MATLAB filter design were then called in the equalizing algorithm. Various test cases were used to verify the filters performance using a generalized input case of low, medium and high frequencies. The simulation in Figure 6.10 verifies that the filters are able to attenuate frequencies out of their respective pass bands as well as pass the frequencies in the desired pass bands. The simulation output shows three test cases: only bass, only mids, and only treble.

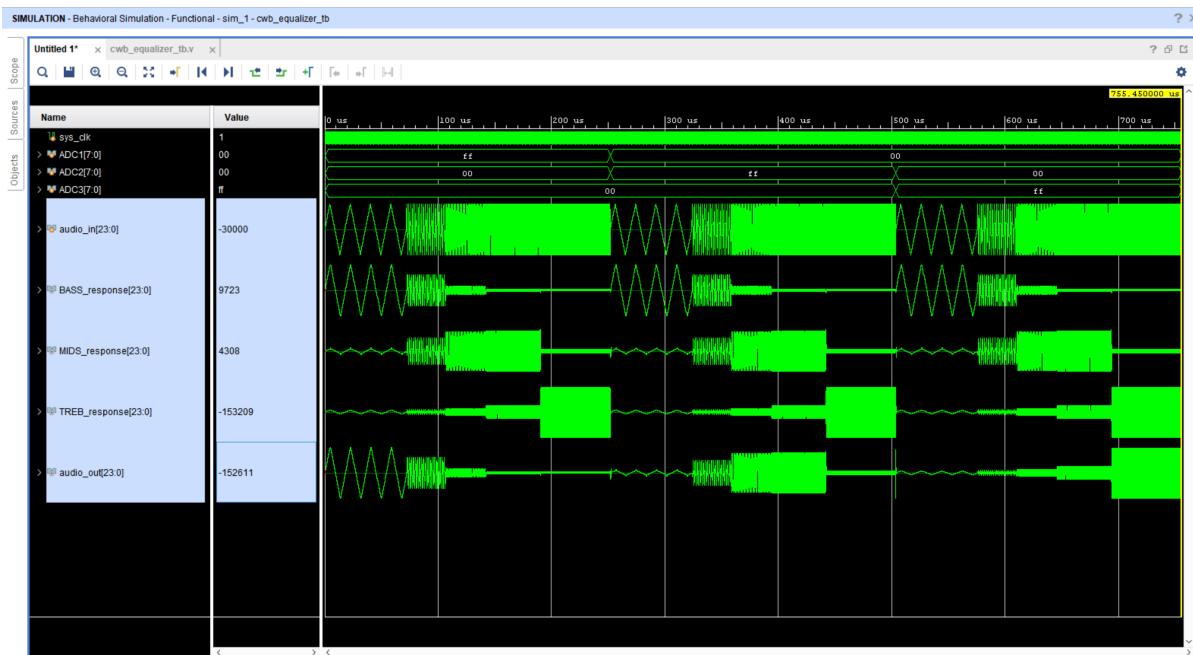


Figure 6.10: Equalizer Algorithm HDL Simulation

Chapter 7

Verification and Performance

7.1 Hardware Verification

Each guitar effect was tested using external hardware to verify proper performance of the design. Since low signal speeds are used for this design, an Analog Discovery 2 module (AD2) was used which incorporates a two channel oscilloscope, waveform generator, and spectrum analyzer all in a small form factor package. The AD2 is used to generate various signals with the function generator which are monitored on channel one of the oscilloscope and fed into the line in port on the Zynq evaluation board. The output of the audio processing chain is fed out through the line out port on the Zynq evaluation board and this output is monitored with channel two on the AD2. The first step in verifying the performance criteria of real-time signal processing it to demonstrate that the input signal can pass through the signal chain and make it to the output within 10ms; the latency of human hearing. Each signal created for the simulation is documented in Appendix VI. The LFM_cwb_signal was used to test the input latency and it was found that with no guitar effect algorithms on, the system latency of the platform was 1.1441ms in Figure 7.1. Throughout the hardware validation presented in this

chapter, the input signal is represented by channel one with an orange trace. The output signal is represented on channel blue represented by the blue trace.

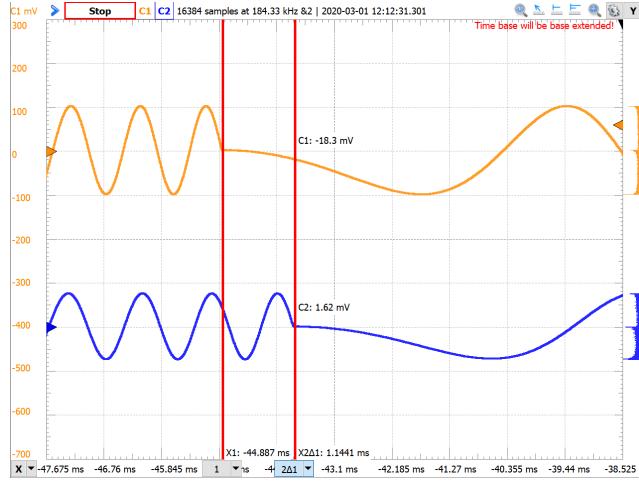


Figure 7.1: Guitar Effect Platform Latency

This result verifies that the test platform is capable of operating within the terminology of real-time. Compared to the worked performed by [10], adding four additional guitar effect algorithms as well as additional AXI bus latency only increased the latency by 481 microseconds which is an acceptable delay.

7.2 Equalizer

The equalizer effect utilizes the LFM_long waveform to create multiple harmonics in a fairly flat manner. This waveform was leveraged to test the performance of the equalizer filter to verify the attenuation regions associated with each input case. The base line condition of the LFM_long input and output response were measured in Figure 7.2 to find that the output signal is roughly 2.5dB down at the output across all frequency ranges prior to the equalizer effect being activated. Verification with the guitar effect consisted of four test cases where the tunable input parameters take the following values.

1. Bass Input (0xFF), Mids Input (0x00), Treble Input (0x00)
2. Base Input (0x00), Mids Input (0xFF), Treble Input (0x00)
3. Base Input (0x00), Mids Input (0x00), Treble Input (0xFF)
4. Base Input (0xFF), Mids Input (0xFF), Treble Input (0xFF)

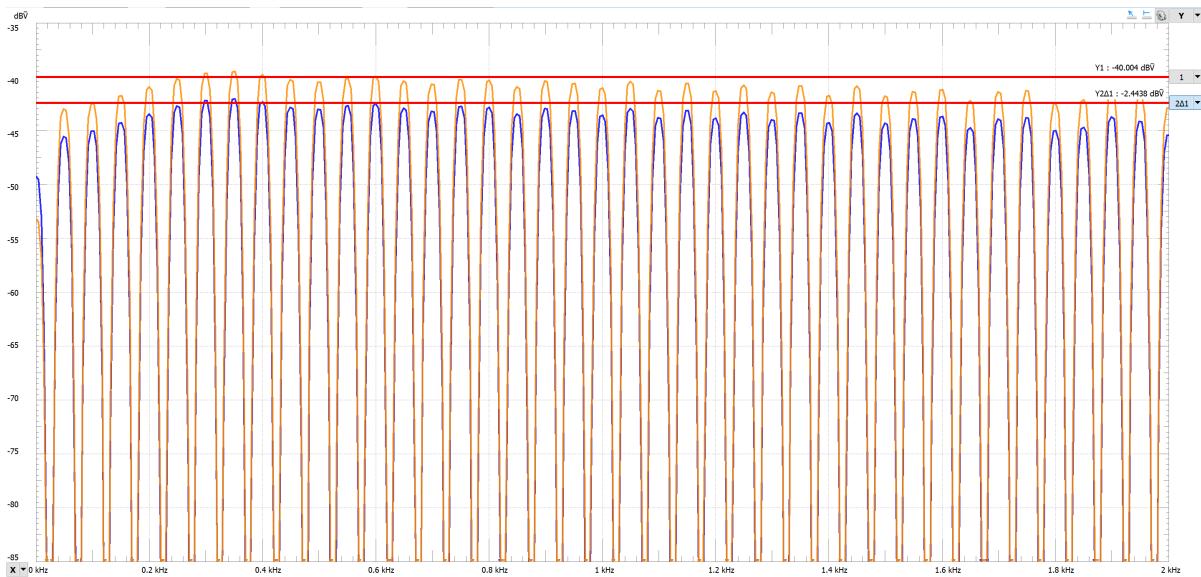


Figure 7.2: Equalizer Spectrum - LFM_long - effect off

The first test in Figure 7.3 demonstrates the equalizer pedal for all of the input parameters set to zero other than the bass input which was maximized. The response of the system was measured at 600 Hz which was designed to be 20dB down. In the hardware testing it was found that the bass response exceeded the design specification by attenuating the signal by 22.6dB at 600Hz.

The second test in Figure 7.4 demonstrates the equalizer pedal for all of the input parameters set to zero other than the mids input which was maximized. The filter was originally designed to have an attenuation of 20dB at 1600Hz and around DC. It was found during the

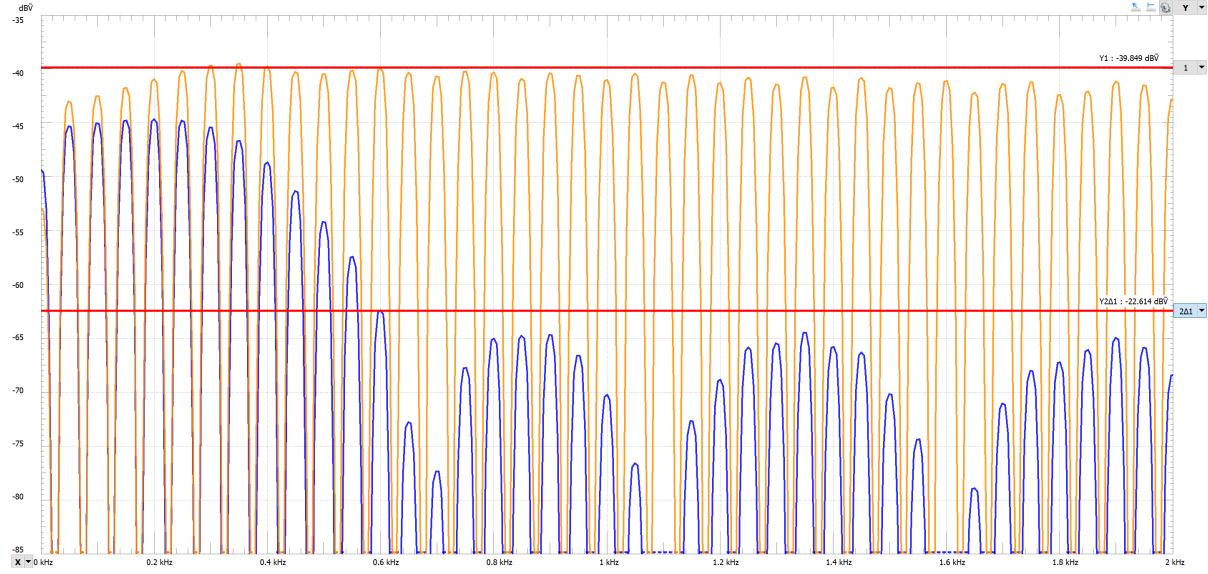


Figure 7.3: Equalizer Test One - Bass Response

hardware simulation that the mids response met the 1600 Hz design requirement and also met the design requirement at 50Hz. At frequencies lower than 50Hz, the filter actually amplified the frequencies however this does not affect the overall effect as a guitar in standard tuning does not have any frequency components below 80 Hz. This means that the mids filtering meets the overall design specifications in the playable range of the guitar frequency band.

The third test in Figure 7.5 demonstrates the equalizer pedal for all of the input parameters set to zero other than the treble input which was maximized. The filter was designed to attenuate 20dB at frequencies lower than 800 Hz. In hardware implementation, the treble filter was able to meet and exceed the design specification by being 25.5 dB down at 800 Hz.

The fourth and final test in Figure 7.6 for the equalizer effect was to maximize all of the equalizer inputs and to see what the overall response of the entire system would be. The system was originally designed to have an attenuation around 1200 Hz through designing the overlapping stop bands of the the mids and treble parameters. This would help to roll off higher frequency harmonics while playing in the low frequencies and attenuate high frequency

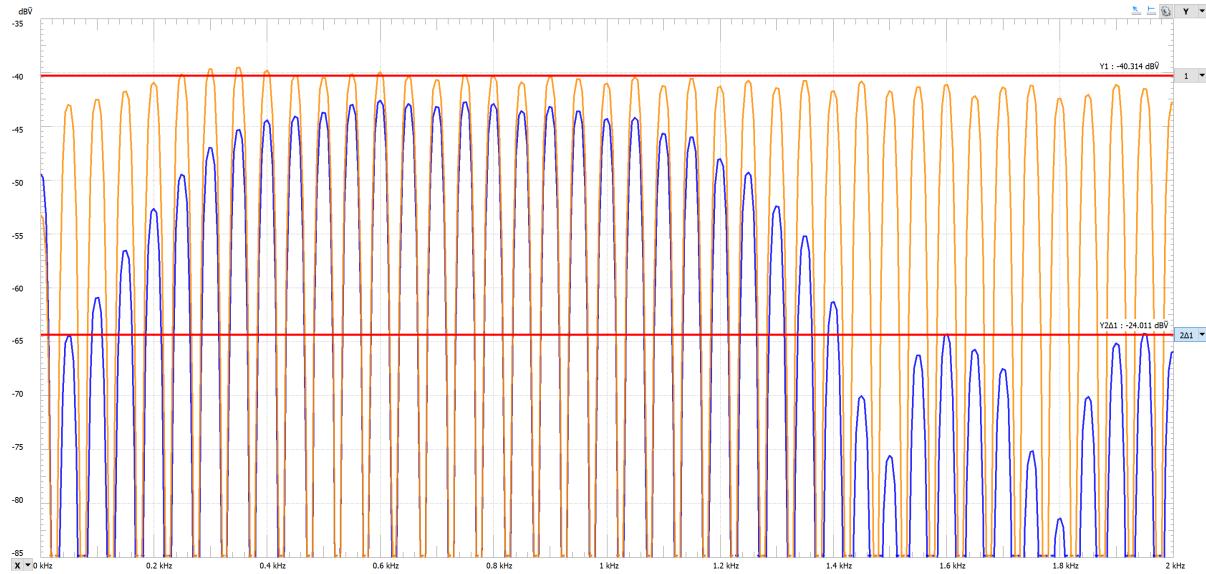


Figure 7.4: Equalizer Test Two - Mids Response

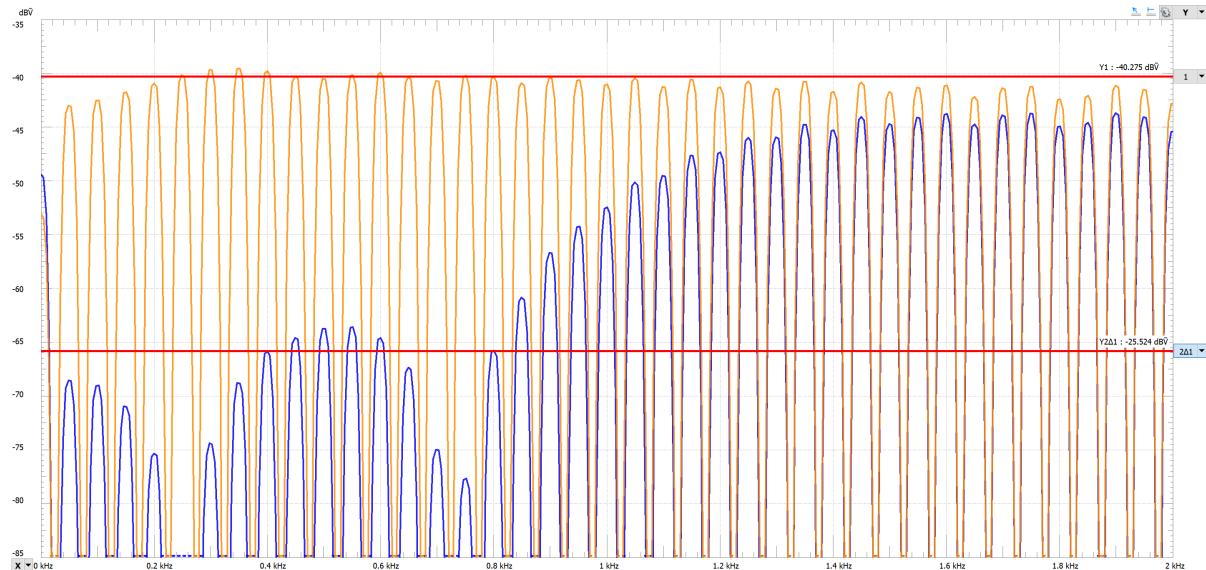


Figure 7.5: Equalizer Test Three - Treble Response

notes created from playing high frets on the high e string. Hardware testing validated that the response of the equalizer with all parameters maximized resulted in a 14dB attenuation around 1200 Hz.

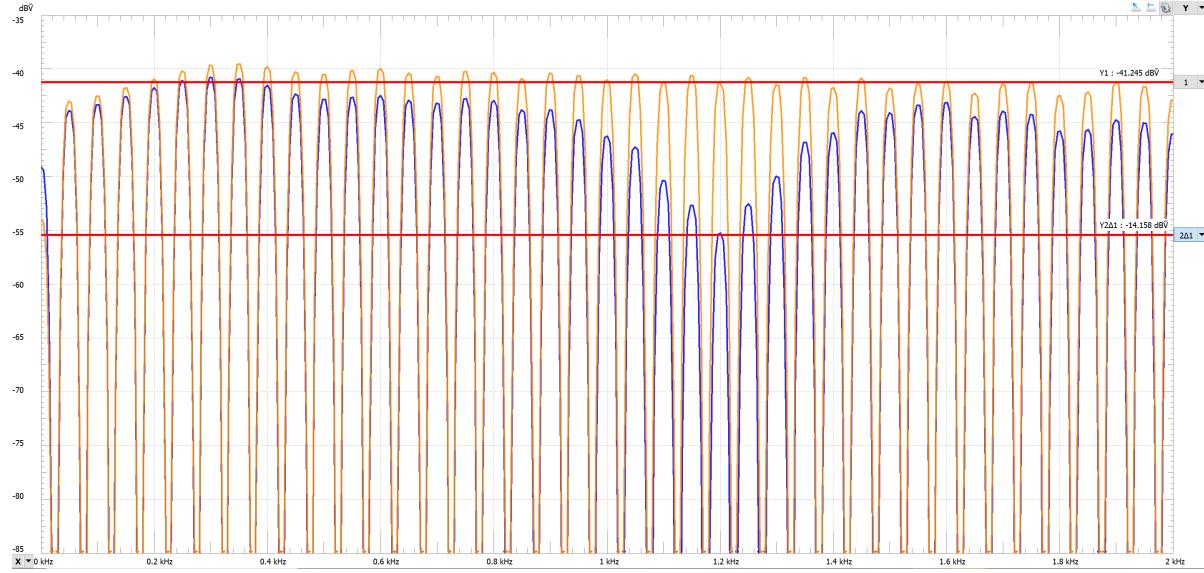


Figure 7.6: Equalizer Test Four - Maximum value for each input parameters

7.3 Gain

The gain pedal was tested using a sinusoidal function. The effect was verified for one test case where the gain pedal was used attenuate the signal and another to verify the maximum boost level achieved by the algorithm. It is noted with the effect off in Figure 7.7 that the output signal is 3dB down at the target frequency of 100Hz.

In test one, the gain parameter was minimized to be as near zero as possible. This hardware test resulted in demonstrating the gain block was able to achieve an overall attenuation of 38 dB when in the lowest operable position.

In the second test in Figure 7.9, the gain parameter was increased to the maximum operat-

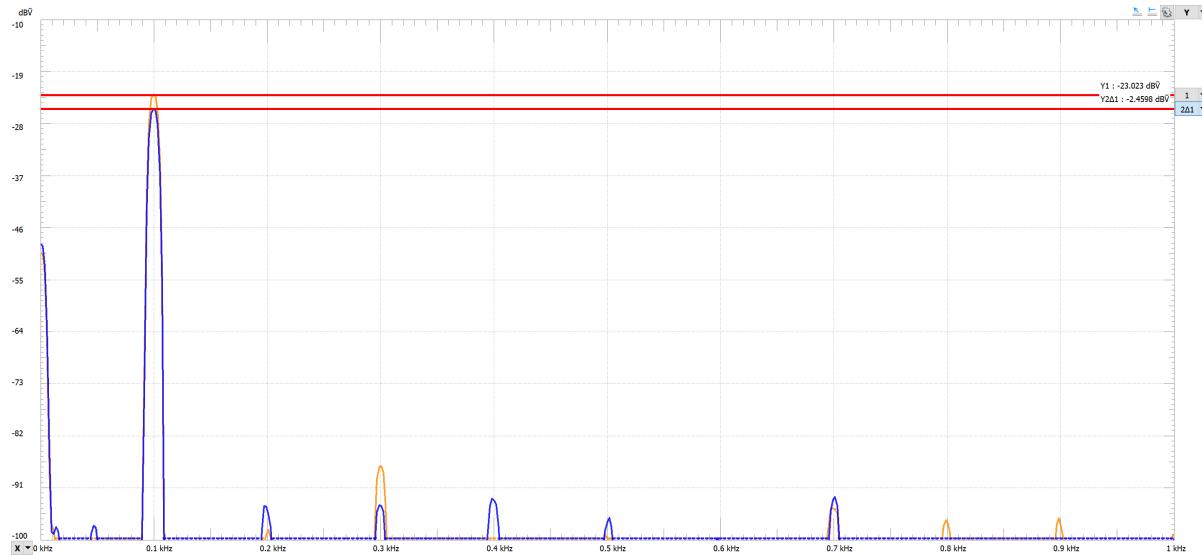


Figure 7.7: Gain Spectrum - Sinusoid - effect off



Figure 7.8: Gain Test One - Attenuation

ing position. The gain block was designed to have a gain of up to four times the input signal, or an overall gain of 12 dB. In the hardware simulation it was found that the total delta from the input signal to the output signal was a 9 dB increase relative to the input signal. Accounting for the 3dB attenuation of the circuit, the total overall gain of the effect is 12 dB which is the design specification. This confirms that the gain blocks meets all of the design specifications.

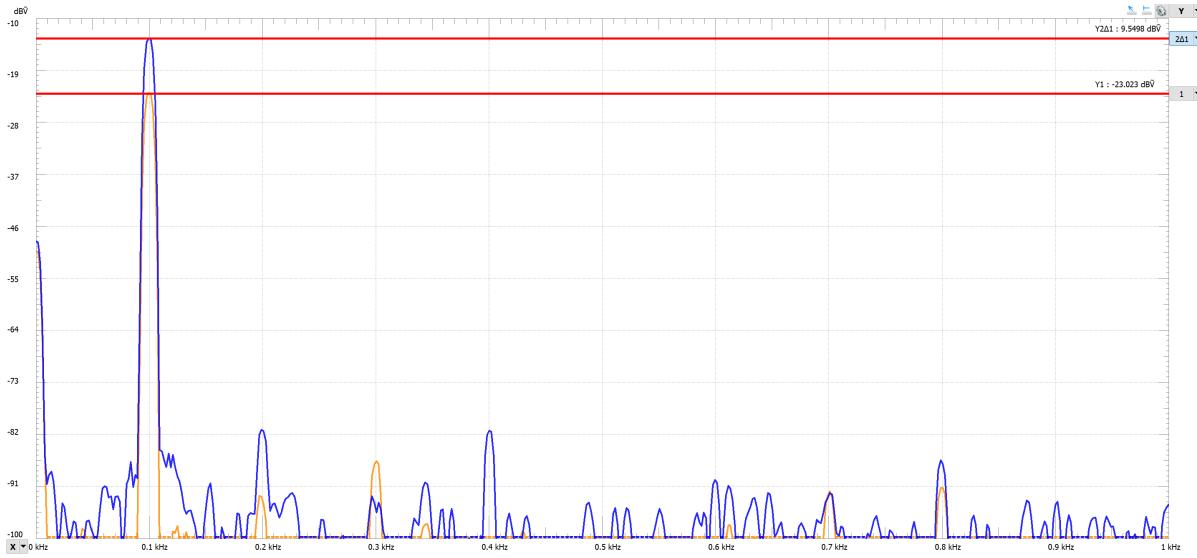


Figure 7.9: Gain Test Two - Boost

7.4 Overdrive

The overdrive pedal was tested using a sinusoidal function with an amplitude of 200mV peak to peak at 100 Hz. The overdrive circuit is designed to produce hard clipping or limiting when the input signal exceeds a certain threshold limit. Two test cases are examined for where the input signal is just starting to become clipped and introduce in odd harmonics. The second test case involves boosting the signal with gain and lowering the threshold to induce more clipping which would result in higher amplitude odd harmonics. In order to demonstrate the

capabilities of the pedal. The gain stage in the overdrive pedal was used in order to have the input signal and the output signal have the same power at 100 Hz; around -23 dBV.

1. Gain Input (0x56), Threshold Input (0x91)

2. Gain Input (0xFF), Threshold Input (0x82)

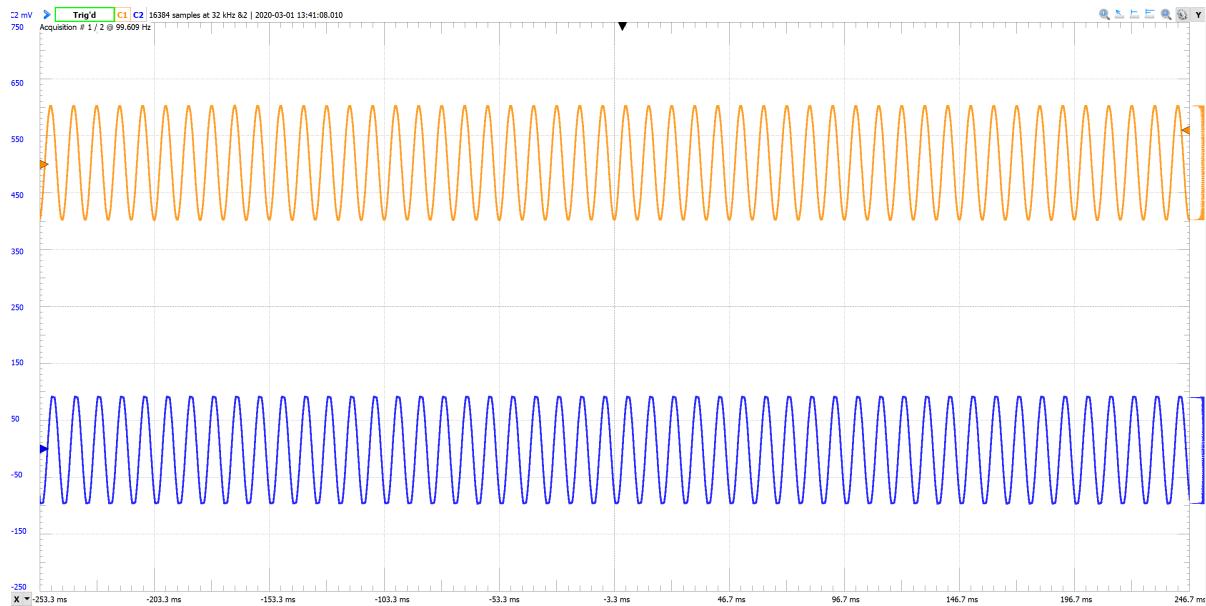


Figure 7.10: Overdrive Test One - Time Domain

In the first test case, it can be seen in Figure 7.10 that the overdrive pedal with minimal clipping in the time domain, adds additional odd harmonics in the frequency domain in comparison to the baseline sinusoidal signal which is shown in Figure 7.11. The power of the odd harmonics increase around 27 dB from the baseline testing conditions which introduced a nice hum to the overall sound of the guitar.

In the second test case with much more distinct clipping in the time domain seen in Figure 7.12 that the odd harmonics in the frequency domain are boosted even more so than the minimal clipping case presented in the first test condition as shown in Figure 7.13. The odd harmonics are increased more than 40 dB than the input signal with each odd harmonic being

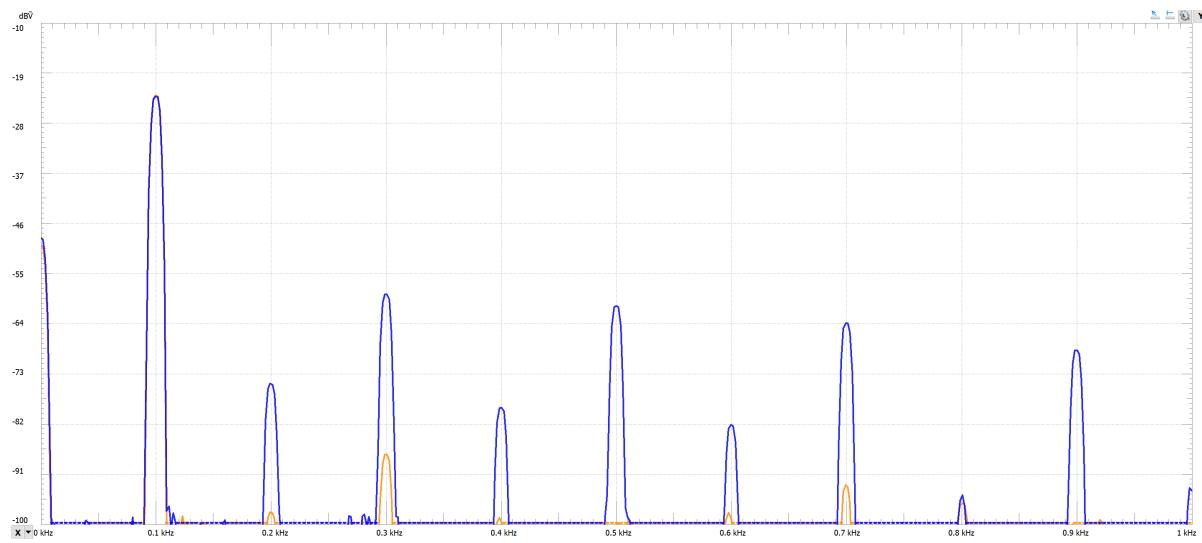


Figure 7.11: Overdrive Test One - Frequency Domain

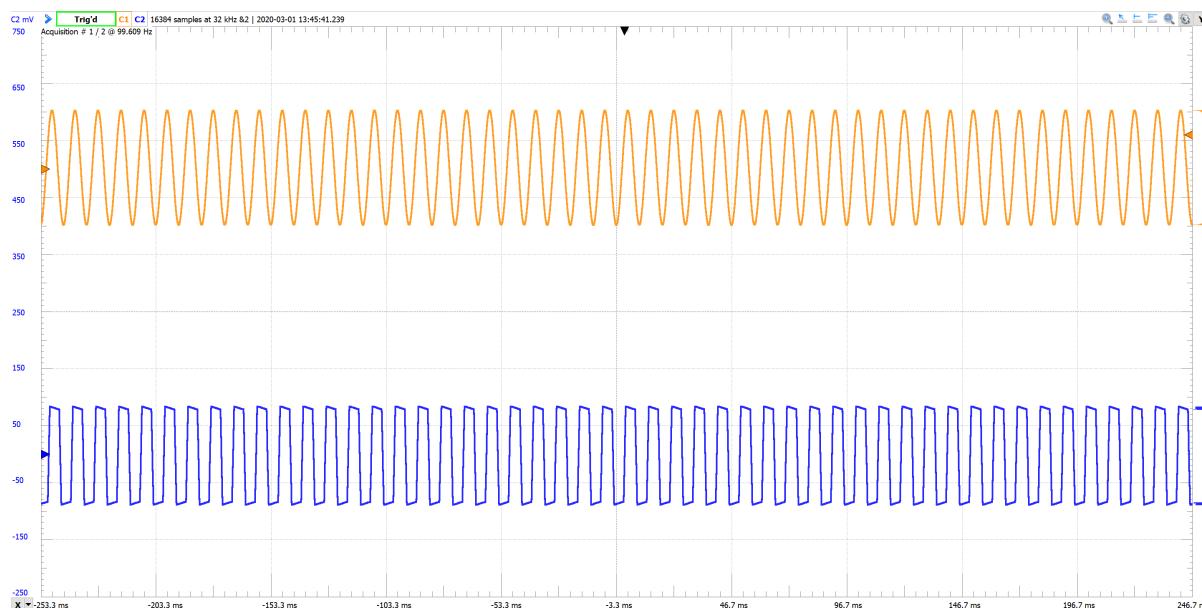


Figure 7.12: Overdrive Test Two - Time Domain

5-9 dB down from the previous odd harmonics as the order increases. This test case provided a guitar with a very thick sound great for heavy metal and hard rock.

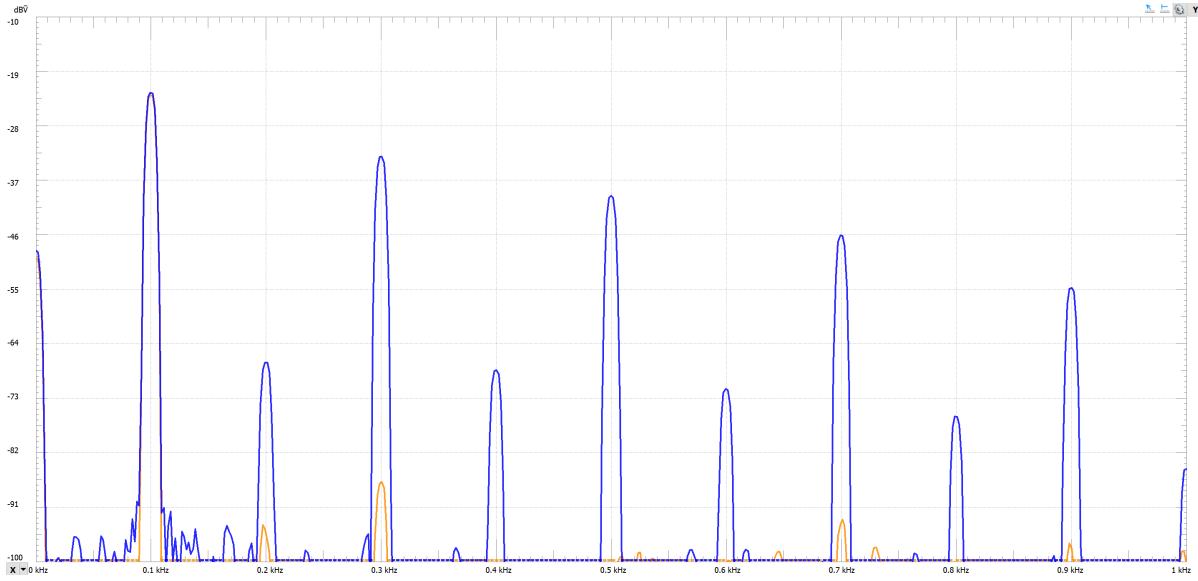


Figure 7.13: Overdrive Test Two - Frequency Domain

7.5 Distortion

The sinusoid waveform amplitude was increased to 250mV for the distortion test cases. Test one demonstrates the level clipping circuit with small quantizing distortion bins where test two demonstrates the same level clipping parameter with much greater quantizing distortion bins.

1. Level Input (0x62), Distortion Input (0x03)
2. Level Input (0x64), Distortion Input (0xFF)

Higher order harmonics added by a hard clipping circuit may not always be as desirable and sound a bit too harsh. This can be seen in Figure 7.14 where an overdrive effect has the exact clipping and amplitude parameters of the distortion effect. The overdrive circuit will exhibit

strong harmonics well in excess of the 11th order. Quantizing the signal at the top end in the distortion effect smooths out the clipping circuit at the top end. For a smaller range bin quantization, it can be seen that the higher order harmonics are removed where the larger quantization bins provide some of the higher order harmonics to pass; however, the amplitude of these higher order harmonics are significantly reduced compared to the overdrive circuit. To make a direct comparison to the overdrive pedal, the input signal and the output signal power were matched at 100 Hz to better directly compare the harmonic performance.

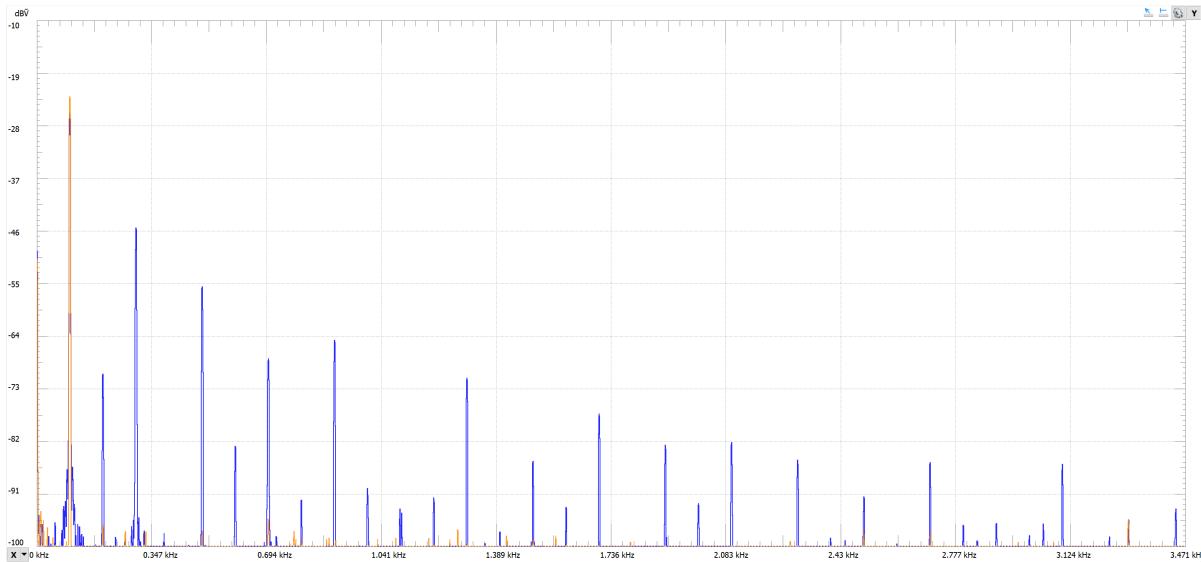


Figure 7.14: Distortion Reference - Overdrive Higher order Harmonics

In the first test case shown in Figure 7.15, the distortion pedal with small quantization bins can be seen to not introduce many harmonics above 2200Hz whereas the overdrive pedal with hard clipping still has fairly prominent odd harmonics extending well past 3300 Hz. The smoothing of the input signal with small range bins at the top end of the distortion pedal helps to remove these higher order harmonics. It can also be seen in the lower order harmonics that the third order harmonic of the distortion pedal for small quantization bins is lower than the overdrive pedal by 6dB.

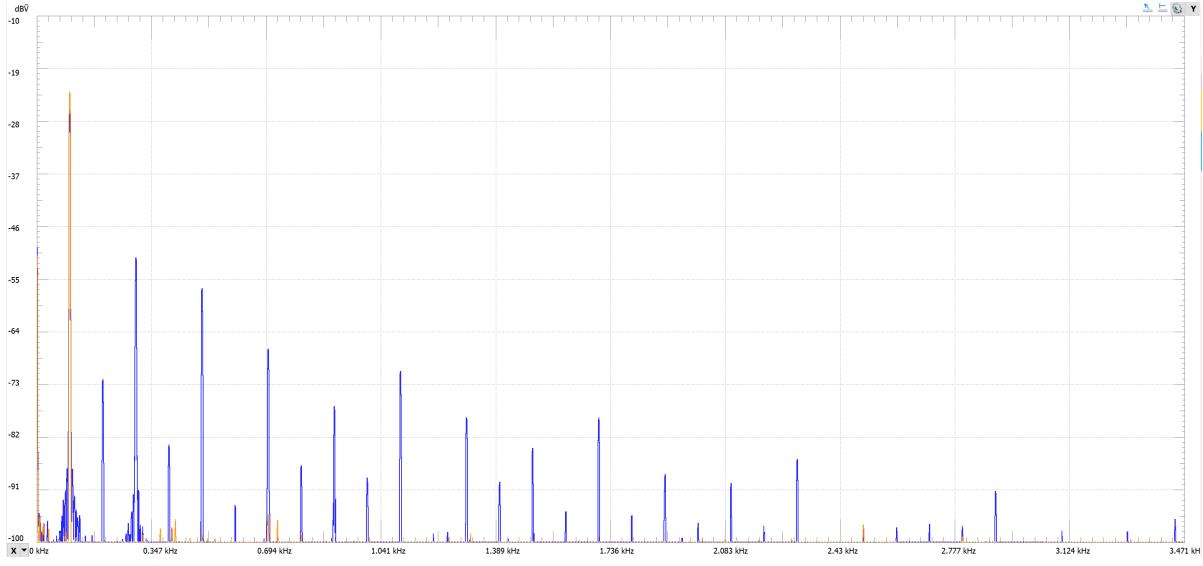


Figure 7.15: Distortion Test One - Small Quantization Bins

In the second test case with the distortion pedal, the quantization bins were increased to their maximum size. The higher order odd harmonics seen in Figure 7.16 remained much lower than the overdrive pedal above 2000 Hz but higher than the distortion pedal with small quantization bins. It should also be noted that with the exception of the third harmonic being higher in the large quantization test, the distortion pedal has closely matched odd order harmonics from the fifth harmonics up to around 2000 Hz. The change in the quantization bins could also be audibly heard while playing the guitar to the point where the larger quantization bins gave a warmer sounding distortion over the small range bins. The small range bins sounded most like the hard clipping overdrive circuit but sounded a bit brighter from having less of the higher order odd harmonics.

7.6 Chorus

The chorus effect was tested with a triangular waveform of 100Hz with an amplitude of 100mV. The first test demonstrates a short time delay and a relatively fast LFO frequency. The second

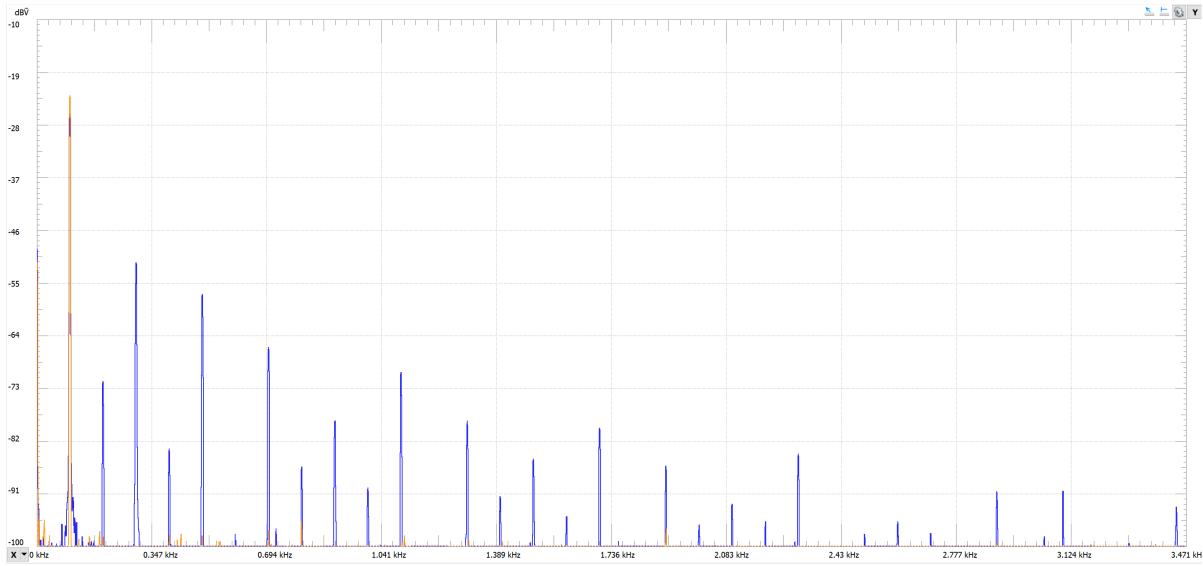


Figure 7.16: Distortion Test Two - Large Quantization Bins

test demonstrates a much slower LFO frequency with a larger delay parameter. The values used in the HDL simulation portion of the chorus design was used again in the hardware testing.

1. Delay Input (0x04), Wet Input (0xFF), LFO Input (0x0E)
2. Delay Input (0x0E), Wet Input (0x84), LFO Input (0xFF)

In the first test case it can be seen in Figure 7.17 that there is a minimal delay and the LFO is oscillating at a fairly quick rate. This result in the output signal fluctuating with the pitch shifting property of the chorus very audible. At very high LFO frequency, it was found that the interpolation factor of 8 could result in a bit or distortion at times if the delay was not adequate.

The second test case with the longer delay and slow LFO frequency create a slowly modulating chorus effect that is almost indiscernible audibly other than a small warble voicing as seen in Figure 7.18. This effect gives more of a swelling effect of rising and falling volume overtime as the interpolated signal is much slower to update than the first test case. When the wet dry mix was increased in the second test case the warble was much more noticeable and the swelling and falling of the volume produced a suitable effect for chord strumming.

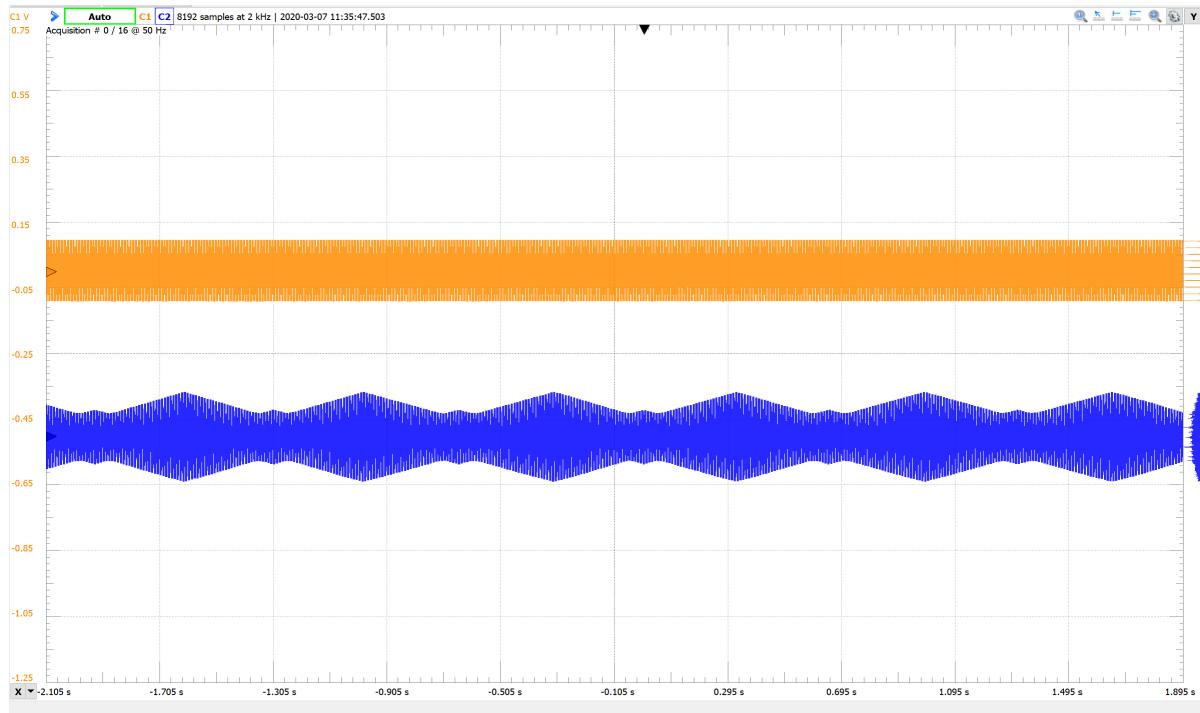


Figure 7.17: Chorus Test One - Fast LFO and small delay

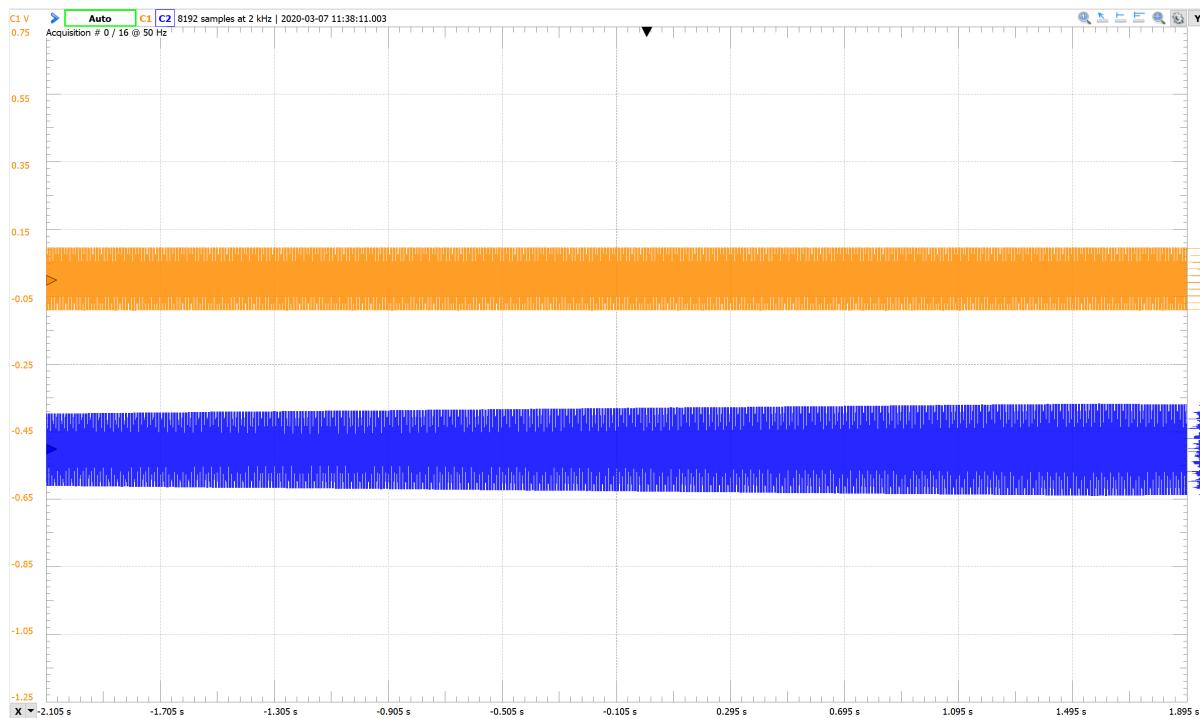


Figure 7.18: Chorus Test Two - Slow LFO and large delay

Due to resource utilization, an interpolation factor of 12 or 16 was not tested but could result in better performance for the very fast modulation of the LFO. However, for the given test cases, the audio performance of the chorus was on par with that of off the shelf modules while producing a wider range of voicing.

7.7 Tremolo

A sine wave of 100Hz with an amplitude of 100mV was used to test the tremolo circuit. Three test cases were run to demonstrate the wave shape control as well as the triangular wave modulation period.

1. Timing Input (0x07), Shape Input (0xFF)
2. Timing Input (0x40), Shape Input (0xFF)
3. Timing Input (0x40), Shape Input (0xB7)

The first test case in Figure 7.19 shows the tremolo effect for a very short period of the modulating triangular wave. This effect results in the input signal oscillating for a few periods in between the modulating triangular wave. The effect overall sounds quite interesting as the numerous zero crossings results in an output signal that does not stay on for a long period of time but resembles a kill switch effect which rapidly turns on and off the guitar output signal.

For the second test case in Figure 7.20, the period of the modulating triangular wave was increased to allow for more oscillations of the input signal to fit within the envelope of the triangular wave. This combination of parameters leads to a slowly modulating signal that raises and lowers the volume with zero crossings completely muting the output signal.

The third test case in Figure 7.21 demonstrates the ability of the wave shaping parameter where the modulating triangle wave is clipped and normalized in order to keep the depth

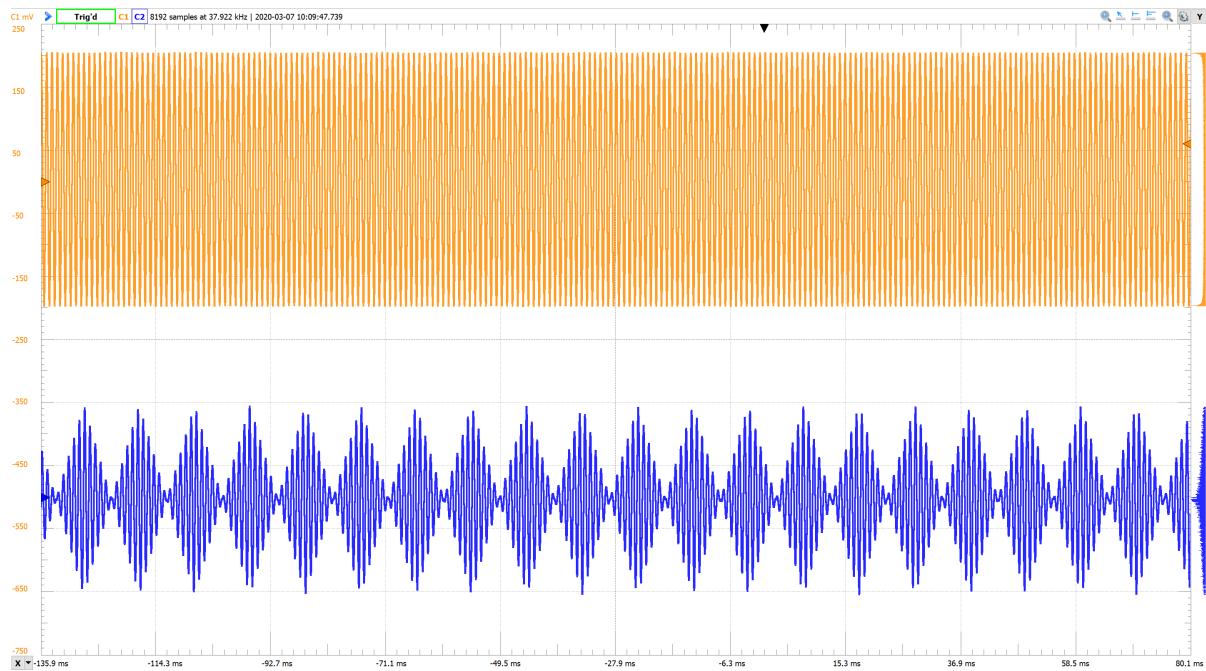


Figure 7.19: Tremolo Test 1 - Fast Modulation with Triangular Wave

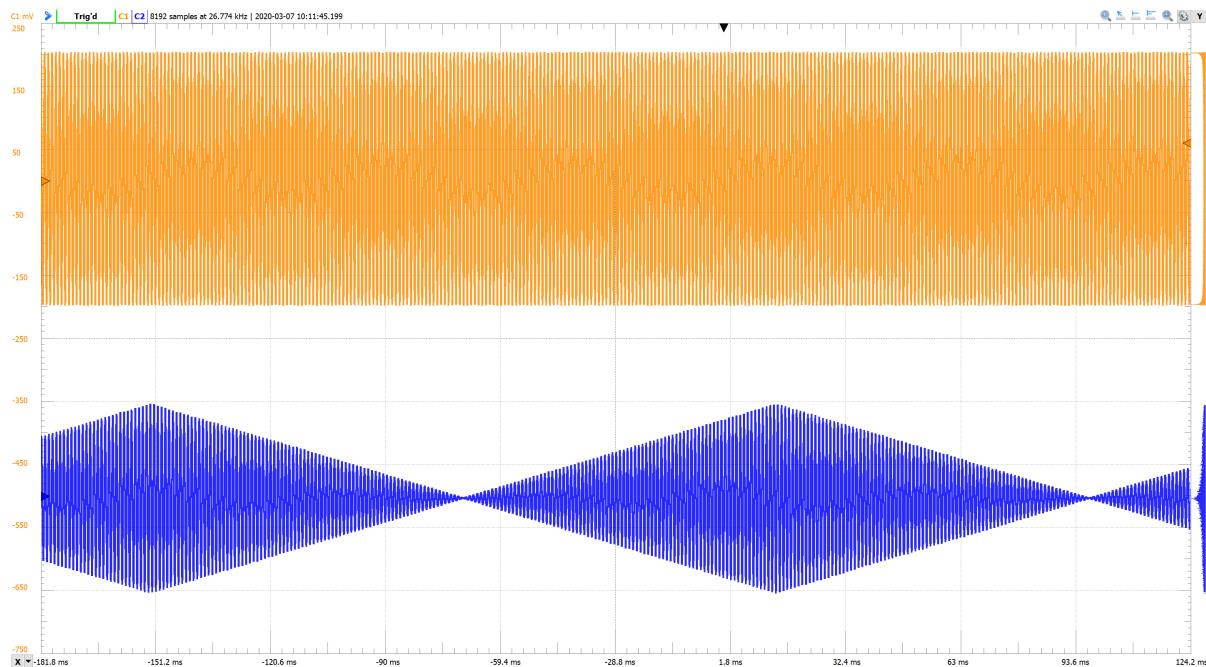


Figure 7.20: Tremolo Test 2 - Slow Modulation with Triangular Wave

constant. This effect causes the incoming signal to be unmodified in a specific region where the modulating signal is flat but quick volume swings when in the changing regions of the waveform. This makes the effect almost like a depth control but the overall zero crossings still yields a muting effect. As the steepness of the modulating signal increases and the constant plateau increases, a bit of distortion is added to the output signal which creates an interesting tonal property of something between a distortion pedal combined with a vibrato kill switch effect which is quite unique.

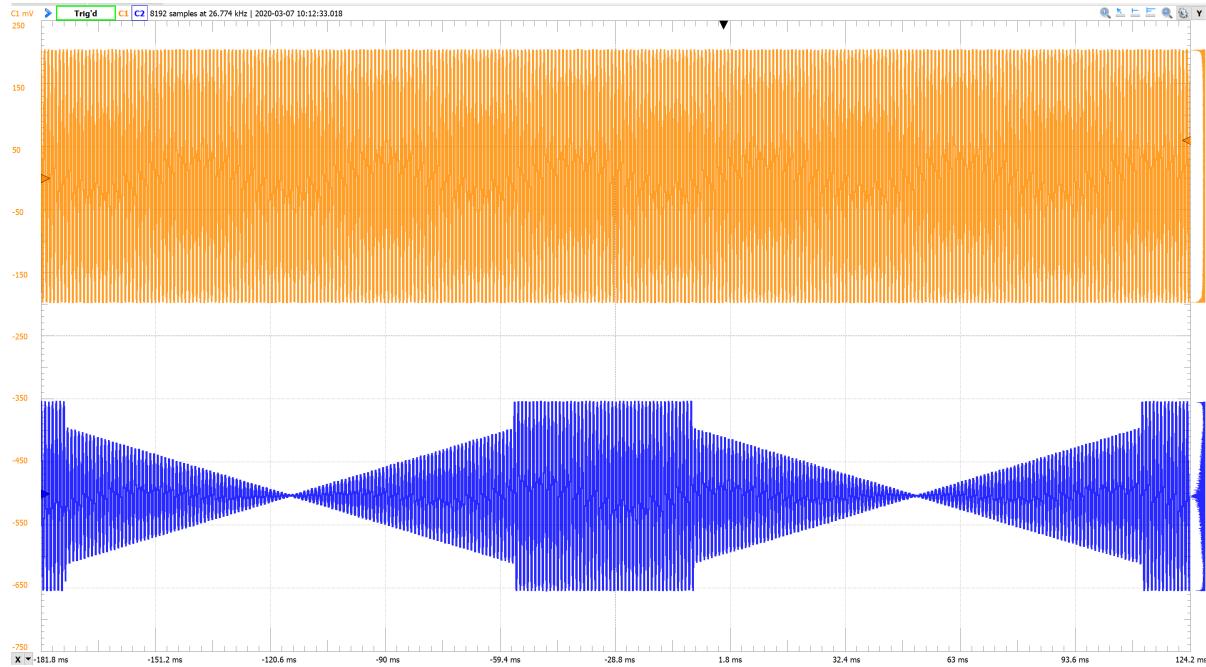


Figure 7.21: Tremolo Test 3 - Slow Modulation with Modified Triangular/Brick Wave

7.8 Delay

The delay algorithm was tested with the pulsed sinusoid signal. The delay algorithm was tested in four cases varying the overall delay, attenuated wet signal, and the IR filter mode.

1. Delay Input (0x01), Wet Attenuation Input (0x5F), FIR/IIR Input (0xFF)

2. Delay Input (0x3A), Wet Attenuation Input (0xC8), FIR/IIR Input (0xFF)
3. Delay Input (0x08), Wet Attenuation Input (0x7F), FIR/IIR Input (0x00)
4. Delay Input (0x08), Wet Attenuation Input (0xDE), FIR/IIR Input (0x00)

The first delay test case demonstrates a FIR delay with a relatively short delay period in Figure 7.22. The FIR mode gives a single repetition of the incoming signal in addition to the incoming signal. This replication is stored in the circular buffer and attenuated before being read out. The corresponding delay of the test case one demonstrates a delay of around 20 ms or the original signal which is audible to most individuals.

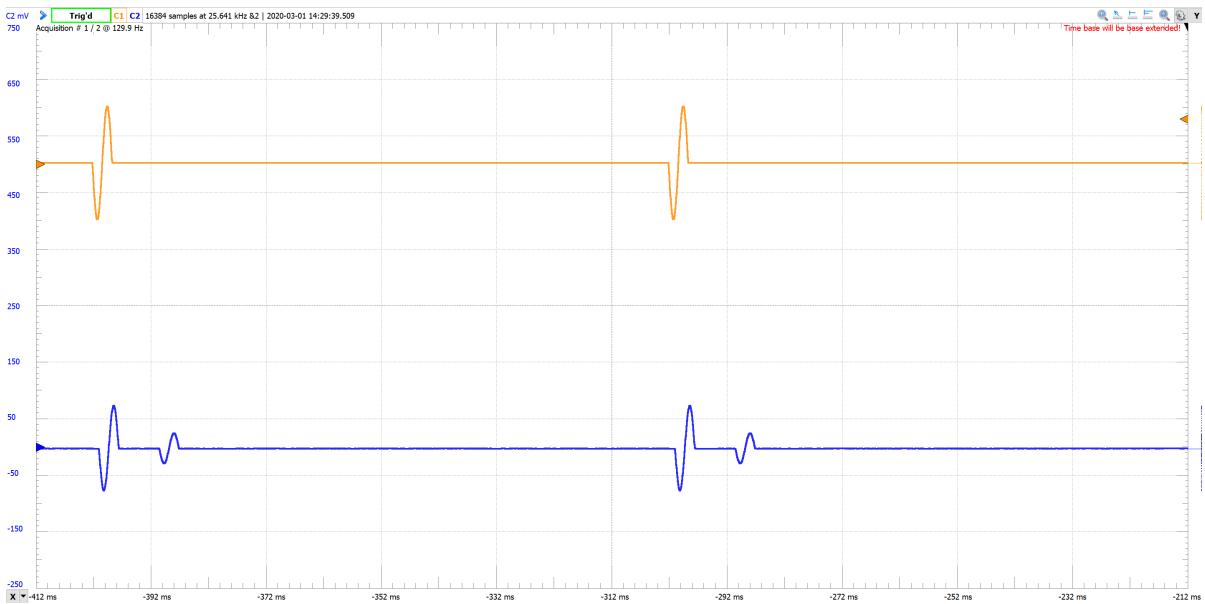


Figure 7.22: Delay Test One - Short FIR Delay

The FIR test case two increased the overall delay of the input signal in the circular buffer to over 100 ms with less attenuation on the buffered signal as seen in Figure 7.23. The delay pedal has a much larger buffer and has been tested to over one second delay in duration but the oscilloscope capture software did not perform well at low sampling rates and the collected data was not visually pleasing even though empirical testing yielded satisfactory results.



Figure 7.23: Delay Test Two - Long FIR Delay

Test case three in Figure 7.24 changes the delay type to IIR where a portion of the delayed signal and the input signal are added into the circular buffer with some attenuation. The amount of attenuation controls the amount of repetitions coming from the pedal effect. For the third test case the overall delay and attenuation were kept fairly conservative in order to demonstrate that the IIR feature is producing more replications than that of the FIR delay.

The last test case for the delay pedal shown in Figure 7.25 involved demonstrating stacking the delayed IIR repetitions directly back to back and finding the right amount of attenuation to halt the signal before the next input signal peak. In order to accomplish this, the delay time was kept constant and the attenuation of the effect was raised in order to allow more repetitions of the signal. The period of the input signal was also increased from 500ms to 1000 ms which helped to better visualize the multiple repetitions of the delayed signal and the IIR effect.



Figure 7.24: Delay Test Three - Short IIR Delay

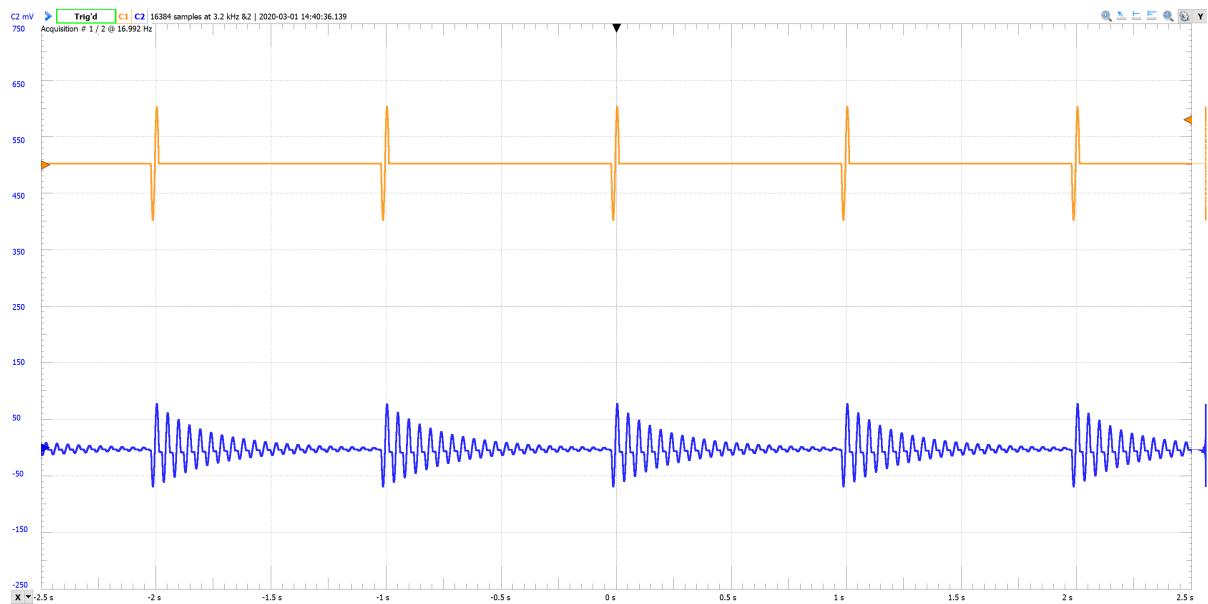


Figure 7.25: Delay Test Four - Long IIR Delay

7.9 Noise Gate

The noise gate effect was tested in hardware to verify a signal would open the gate when it exceeds a certain threshold. Additionally, the gate should close if the input signal is below the threshold for a specified period of time. The test performed on the noise gate effect tested both the opening and closing of the noise gate for a 200Hz sine wave. To open the gate the sine wave amplitude was increased from 5mV to 100mV. To test the closing of the noise gate the sine wave was decreased from an amplitude of 100mV to 5mV.

1. Thresh Input (0x255) - gate opening
2. Thresh Input (0x255) - gate closing

For the first test case in Figure 7.26, the input signal is elevated from 5mV to 100 mV within the waveform generating tool. The noise gate is fairly quick to open once the input signal crosses the input threshold and the attack of the signal comes to full volume over the course of 60 ms which removes any popping or distortion from the output signal rising too fast. Through empirical testing it was found that a quick attack time was more adventurous and that sliding fingers around on the strings or accidental picking of the strings did not open the gate. Noise from high gain stages also did not trip the gate open in a false opening which verified the design goal outlined in previous sections.

The second test for the noise gate in Figure 7.27 consisted of having the 100mV signal amplitude change to 5mV to simulate a guitar signal that would fall below a given threshold but not be completely zero. The noise gate was able to meet closure withing a few milliseconds and there was minimum ripple on the output signal. Through further empirical testing it was found that a long hold time with a small threshold made closure very difficult to achieve and thus a short hold time was utilized on order to meet realistic guitar playing needs. This ensured

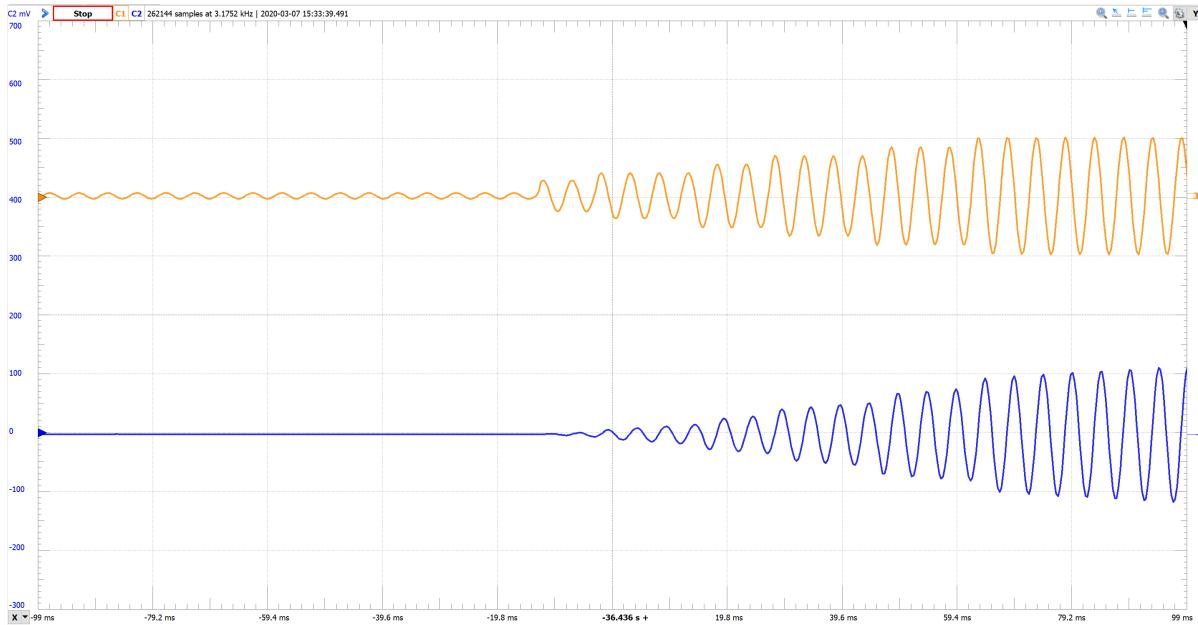


Figure 7.26: Noise Gate Test One - Opening Gate

that after the player mutes the strings of the volume decays past the threshold, the output to the amp is near muted with no popping or distortion.

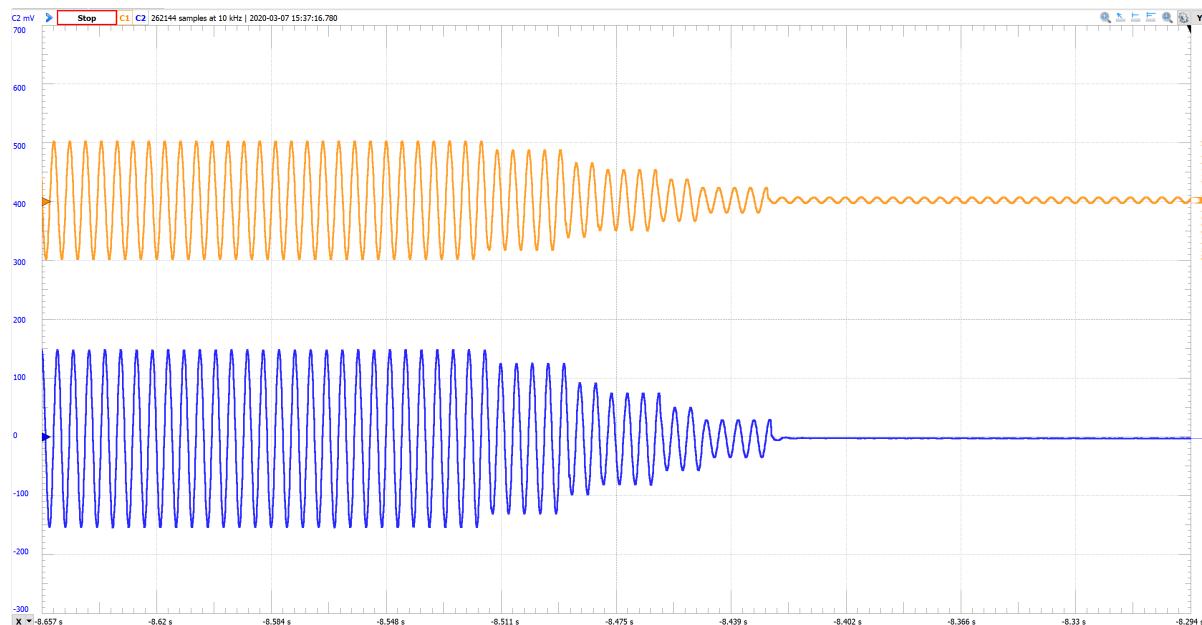


Figure 7.27: Noise Gate Test Two - Closing Gate

Chapter 8

Conclusion

8.1 Conclusion

This project sought to deliver a robust platform to perform digital signal processing on an FPGA for guitar effect pedals. Mechanical housings were designed to secure and protect the TI MSP430 evaluation boards which took user inputs in the form of potentiometer data. The potentiometer data was sampled using ADCs on the TI MSP430 evaluation board and communicated to the Zynq SoC PS over an I2C bus. At the same time, an ADI codec sampled incoming audio signals and communicated the 48kHz sampled data to the Zynq SoC PS. The data in the PS was pushed into the PL over AXI busses. The sampled potentiometer data was routed to each signal processing algorithm to modify the respective effects. The sampled audio passes serially through the guitar effects until modified by an active algorithm. The final modified audio output signal is then routed from the PL to the PS via another AXI bus. Once in the PS, the modified audio signal is routed back to the ADI codec through I2S. This project successfully developed eight guitar effect pedal algorithms in hardware and demonstrated the feasibility of putting numerous guitar effects into an entry level FPGA while using minimal

additional hardware. These pedals also perform functionally similar to that of stand alone guitar pedals lending themselves to be foot operated as opposed to limited audio options and hand operation seen in previous iterations[10].



Figure 8.1: Guitar Effect Pedals (Front)

8.2 Future work

Over 20 months of working in MATLAB, Vivado, XSDK, and TI CCS IDEs where coding in C, Verilog, VHDL, and MATLAB as well as creating mechanical housings in Solidworks, this project has grown much larger than initially anticipated for one person to tackle. That being said, a lot has been learned about FPGA design, SoC implementation, digital signal processing, algorithm design, and embedded systems. The design of the guitar effects platform was able to successfully integrate tunable algorithms with a traditional foot pedal interface. This project

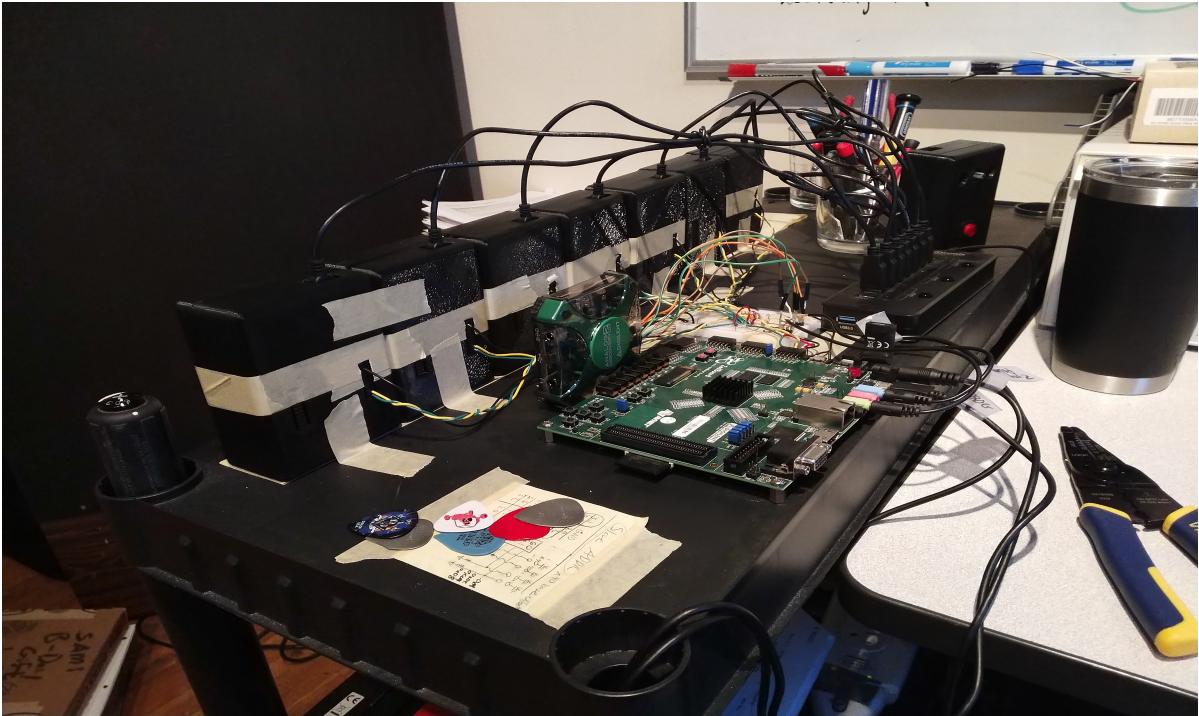


Figure 8.2: Guitar Effect Pedals (Rear)

would serve as a great starting platform for any student wishing to experiment with audio signal processing. Additional algorithms like Flanger, Phaser, Wah-Wah, Wammy, and so many more effects could be designed in a bare metal sense or designed in Simulink and export the algorithm via HDL Coder. Additional effort could be put into designing custom PCBs for the guitar pedals as well as the main SoC board in order to shrink size making it possible to have a fully integrated solution like the Line 6 Helix pedal board. Furthermore, additional cabinet emulation effects could be added into the processing chain to give different amplifier sounds like are found in high end DAWs. There are countless examples of directions to take this project and it offers a chance for beginner or experienced signal processing engineers to start creating their own guitar effects allowing for more individualized tone.

References

- [1] Scott Marquart. Guitar strings order: How the guitar is tuned and why. Website, August 2016.
- [2] Avnet. *Zedboard Hardware Users Manual*. Avnet, 2.2 edition, January 2014.
- [3] AnalogDevices. *ADAU1716 Datasheet*. Analog Devices, One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A, e edition, October 2018.
- [4] Texas Instruments. *MSP430FR2355 LaunchPad Development Kit - SLAU680*. Texas Instruments, May 2018.
- [5] YATRITRIVEDI. How do guitar distortion and overdrive work? Web Article, September 2016.
- [6] Mimmitronics. Talk theory to me: Tremolo effect. Website, January 2017.
- [7] TeachMeAudio. Noise gate. Website, November 2016.
- [8] Iain Fergusson. Noise gate diagram. Website, June 2008.
- [9] MathWorks. noisegate. WEbsite, 2016.
- [10] Vladi Litmanovich and Adi Mikler. Fpga design and implementation of electric guitar audio effects. Project report, Tel Aviv University, 2015.

- [11] MartinGuitarCompany. The timeline of martin guitar: 1796-1873. Electronics, 2019. doi:<https://www.martinguitar.com/about/martin-story-martin-timeline/the-timeline-of-martin-guitar-1796-1873/>.
- [12] Yamaha. The origins of the acoustic guitar. Website, 2019. doi:https://www.yamaha.com/en/musical_instrument_guide/acoustic_guitar/structure/.
- [13] Marcia Daft and Charlie McGovern. The invention of the electric guitar. Website, April 2014. doi:<https://invention.si.edu/node/791/p/413-credits>.
- [14] Richard French. *Engineering the Guitar Theory and Practice*. Springer, 2009.
- [15] Timothy Berg. Electric guitar. Website, December 2019.
- [16] NPR. 'voices within the music': A brief history of guitar effects. Website, December 2014.
- [17] M. Malko, J. Kovacevic, R. Peckai-Kovac, M. Gajic, and M. Jovanovic. Implementation of digital audio effects for electric guitar on dsp platform. In *2011 19thTelecommunications Forum (TELFOR) Proceedings of Papers*, pages 1099–1102, Nov 2011. doi:[10.1109/TELFOR.2011.6143741](https://doi.org/10.1109/TELFOR.2011.6143741).
- [18] K. Dempwolf, M. Holters, S. MÄ¶ller, and U. ZÄ¶lzer. Geb1 - a robust dsp platform for audio and guitar signal processing in education. In *4th European Education and Research Conference (EDERC 2010)*, pages 11–14, Dec 2010.
- [19] Denton Dailey. *Electronics for Guitarists*. Springer, 2013. doi:<https://doi.org/10.1007/978-1-4614-4087-1>.
- [20] Brian Tarquin. *Stomp on This!* Cengage Learning, 2014.

- [21] Sujit Rokka Chhetri, Bikash Poudel, Sandesh Ghimire, Shaswot Shresthamali, and Dinesh Sharma. Implementation of audio effect generator in fpga. *Nepal Journal of Science and Technology*, 15, 02 2015. doi:10.3126/njst.v15i1.12022.
- [22] Xilinx. *XMP097*. Xilinx, 1.3.2 edition, 2014.
- [23] Xilinx. *AXI Reference Guide*. Xilinx, 13.1 edition, March 2011.
- [24] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, Glasgow, GBR, 2014.
- [25] M. J. Caputa. Developing real-time digital audio effects for electric guitar in an introductory digital signal processing class. *IEEE Transactions on Education*, 41(4):10 pp.–, Nov 1998. doi:10.1109/13.728274.
- [26] S. Palakvangsa, D. Sookcharoenphol, and P. Sooraksa. New adjustable non-linear characteristic approximated by bernstein polynomial for guitar effect. In *2017 21st International Computer Science and Engineering Conference (ICSEC)*, pages 1–5, Nov 2017. doi:10.1109/ICSEC.2017.8443780.
- [27] Dave Marshall. Digital audio effects. Lecture, Cardiff University, January 2011.
- [28] Woody Herman. Dspfuzz a guitar distortion pedal using the stanford dsp sheild. Project report, Stanford University-Center For Computer Research In Music And Acoustics, 2015.
- [29] David Yeh. *DIGITAL IMPLEMENTATION OF MUSICAL DISTORTION CIRCUITS BY ANALYSIS AND SIMULATION*. PhD thesis, STANFORD UNIVERSITY, June 2009.

- [30] E. Zeki and T. Ciloglu. Enhanced modelling of guitar distortion. In *2015 23nd Signal Processing and Communications Applications Conference (SIU)*, pages 1473–1476, May 2015. doi:10.1109/SIU.2015.7130122.
- [31] P. Anghelescu, S. Angelescu, and S. Ionita. Real-time audio effects with dsp algorithms and directsound. In *Proceedings of the 2014 6th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 35–40, Oct 2014. doi:10.1109/ECAI.2014.7090207.
- [32] Julius.O. Smith. *Physical Audio Signal Processing, online book, 2010 edition.* W3K Publishing, 2010.
- [33] Mihai Micea, M. Stratulat, Dan Ardelean, and Daniel Aioanei. Implementing professional audio effects with dssps. *Scientific Bulletin of the UPT: Transactions on Automatic Control and Computer Science*, 46:55–60, 04 2001.
- [34] Gerard Amposta Boquera. Implement a chorus effect in a real-time embedded system. Master’s thesis, Escola Tecnica Superior dEnginyeria de Telecomunicacio de Barcelona, 2016.
- [35] Richard Dudas. "warm chorus": Re-thinking the chorus effect using an orchestral section model. *ICMC 2012: Non-Cochlear Sound - Proceedings of the International Computer Music Conference 2012*, pages 84–87, 01 2012.
- [36] Dimitrios Giannoulis, Michael Massberg, and Joshua Reiss. Digital dynamic range compressor design a tutorial and analysis. *J. Audio Eng. Soc*, 60(6):399–408, June 2012.
- [37] Salvador Tropea. Fpga implementation of base-n logarithm. Technical report, Electronica e Informatica Instituto Nacional de Tecnologia Industrial, 2007.

- [38] Julius O. Smith. *Introduction to Digital Filters with Audio Applications*. W3K Publishing, <http://www.w3k.org/books/>, 2007.
- [39] Julius O. Smith. *Spectral Audio Signal Processing*. <http://ccrma.stanford.edu/~jos/sasp/>, 2011. online book, 2011 edition.
- [40] Julius O. Smith. *Mathematics of the Discrete Fourier Transform (DFT)*. W3K Publishing, <http://www.w3k.org/books/>, 2007.
- [41] Wayne T. Padgett and David V. Anderson. *Fixed-Point Signal Processing*. Morgan and Claypool Publishers, 1st edition, 2009.

Appendix I

Fixed Point Mathematics

Floating point mathematics make audio processing algorithms easy and straight forward to implement in practice with tools like MATLAB, C, etc. There are IP cores that allow FPGAs to perform floating point mathematics but these cores often result in high chip utilization. To avoid high resource allocation, fixed point mathematics are more commonly used on FPGA where the quantization trade offs can be allowed. Therefore, fixed point notation can provide better resource usage and faster performance at the cost of up-front calculations and possible resolution trade offs. Padgett and Anderson provide an in depth text that covers fixed point notation and working with fixed point data for signal processing applications. The portions of text below are a summary from Chapter 4[41] to provide readers with an understanding of fixed point notation and what modifications can be made to convert floating point algorithms to fixed point algorithms.

Floating point notation refers to the decimal placement in a calculation. If the equation $y = 2.5 * 2$ was performed in floating point math the answer would be $y=5$. Floating point mathematics keeps placement of the decimal and automatically places the decimal in the final output of the answer. If we were to perform the same calculation in fixed point mathematics

Table I.1: Floating point to fixed point examples

Floating Point	2^i	Binary	Q	Word length
1.5	$2^0 * 2^{-1}$	1.1	Q1.1	2
2.125	$2^0 * 2^{-3}$	10.001	Q2.3	5
0.0625	2^{-4}	0.0001	Q1.4	5
19.125	$2^4 * 2^1 * 2^0 * 2^{-3}$	10011.001	Q5.3	8

we would be lost as the decimal place needs to first be specified. The decimal placement is specified in fixed point format using Q(I.D) notation where I is the whole number and D is the decimal. The integer and the decimal are expressed in binary notation. A shorthand is to drop the parentheses and represent the notation as QI.D. A few examples to show float point notation and the conversion to fixed point notation:

When performing arithmetic with fixed point numbers it is critical to keep track of the decimal throughout the calculation in order to obtain the correct answer. Given the word length and Q notation of number i as M_i and $Q_iI.D$ respectively along with a number j or corresponding word length and Q notation M_j and $Q_jI.D$, the resulting operation gives word length and Q notation M_k and $Q_kI.D$.

Fixed point multiplication

$$M_k = M_i + M_j \quad (\text{I.1})$$

$$Q_kI_{i+j}.D_{i+j} = Q_iI.D + Q_jI.D \quad (\text{I.2})$$

An example would be:

$$0.5 * .25_{10} = .125$$

$$0.1_2 * 0.01_2 = 00.001_2$$

$$Q1.1 * Q1.2 = Q2.3$$

Noting the position of the decimal allows for resizing of the length or Q format.

$$.125_{10} = 00.001_1$$

Fixed point addition

$$M_k = \max(M_i, M_j) + 1 \quad (\text{I.3})$$

$$Q_k I_{i+j}.D_{i+j} = Q_i I.D + Q_j I.D \quad (\text{I.4})$$

An example would be:

$$.5 + .25 = .75$$

Addition in fixed point needs to be decimal aligned therefor the first term is zero padded by one bit.

$$0.1_2 + 0.01_2 = 0.10_2 + 0.01_2 = 0.11_2$$

$$.75_{10} = 0.11_2$$

Appendix II

HDL Source Code

The code taken for the appendix comes from the ip_repo folder at the same level as the Lyx project in the repository.

Gain Pedal Code

```
1 module cwb_gain_pedal(
2     input signed [23:0] audio_in ,
3     input sys_clk ,
4     input [7:0] I2C_ADDR ,
5     input ENABLE_I2C_data_move ,
6     input [7:0] ADC1_Gain ,
7     input [7:0] ADC2, // NotUsed
8     input [7:0] ADC3, // Not Used
9     input [7:0] ADC4, // Not Used
10    output reg signed [23:0] audio_out ,
```

```
11      output reg GainLED
12      // output reg [23:0] THRESH_TEST_pos,
13      // output reg [23:0] THRESH_TEST_neg
14      );
15
16  /*
17  24 bit audio comes into the block - active:
18      Drive controls gain/boost of OG signal
19      Level Control controls cut off volume
20      Output signal
21 */
22
23 // Control Registers
24 reg PedalON = 0;
25
26 reg [7:0] ADC1reg = 0;
27 reg [7:0] ADC2reg = 0;
28 reg [7:0] ADC3reg = 0;
29 reg [7:0] ADC4reg = 0;
30
31 reg [7:0] ADC1regtemp = 0;
32 reg [7:0] ADC2regtemp = 0;
33 reg [7:0] ADC3regtemp = 0;
34 reg [7:0] ADC4regtemp = 0;
35
```

```
36     reg signed [23:0] GAIN = 0;
37     reg signed [31:0] Gain1 = 0;
38
39     reg signed [23:0] tempAudioin = 0;
40     reg signed [23:0] tempAudioout = 0;
41
42
43
44 // End Control Registers
45 //
```

```
46
47
48 //
```

```
49
50     always @ (posedge sys_clk) begin
51         if(ENABLE_I2C_data_move == 1) begin // values are
52             stable and not changing so we can store them
53             if(I2C_ADDR == 8'h43) begin // correct address to
54                 store values
```

```
54      if(ADC1_Gain != 0) begin // this will be used
55          PedalON = 1;
56      end else begin
57          PedalON = 0;
58      end
59      // Set the values of the actual regs to have
60      // data be displayed to other modules
60      ADC1reg = ADC1_Gain;
61      // ADC2reg = ADC2_Level;
62      // ADC3reg = ADC3;
63      // ADC4reg = ADC4;
64      // set temp reg values for when not addr or not
65      // valid data move
65      ADC1regtemp = ADC1reg;
66      ADC2regtemp = ADC2reg;
67      // ADC3regtemp = ADC3reg;
68      // ADC4regtemp = ADC4reg;
69      end else begin
70          ADC1reg = ADC1reg;
71          // ADC2reg = ADC2reg;
72          // ADC3reg = ADC3reg;
73          // ADC4reg = ADC4reg;
74      end
75      end else begin
```

```
76          //nop ?
77      end
78  end
79 //end control always block - put this in each pedal effect
     module
80
81 //
```

```
82      always @ (posedge sys_clk) begin //Control the output
           audio and modification
83      GainLED = PedalON;
84
85 //Let ADC2reg = Gain2 with zero padding
86 //Gain1 = audio_in*ADC1reg; //Q24.0*Q4.4 = Q28.4
87
88 if (PedalON==1)begin
89     if (audio_in[23] == 1'b0) begin //Positive gain
90         Gain1 = audio_in*ADC1reg; //Q24.0*Q2.6 = Q26.6
91         audio_out = Gain1[29:6];
92     end
93     else if (audio_in[23] == 1'b1) begin //Negative gain
94         tempAudioin = -audio_in;
95         Gain1 = tempAudioin*ADC1reg;
96         tempAudioout = Gain1[29:6];
```

```
97         audio_out = -tempAudioout;  
98     end  
99     else begin // possible zero error  
100        audio_out = 24'h000000;  
101    end  
102  end  
103 else begin // pedal off  
104    audio_out = audio_in;  
105  end  
106 end // end always  
107  
108 endmodule
```

Listing II.1: Gain Algorithm Verilog File

Gain Pedal Test Bench Code

```
1 module cwb_gain_tb ;
2     reg signed [23:0] audio_in ;
3     reg sys_clk ;
4     reg [7:0] I2C_ADDR ;
5     reg ENABLE_I2C_data_move ;
6     reg [7:0] ADC1_Gain ;
7     reg [7:0] ADC2; // NotUsed
8     reg [7:0] ADC3; // Not Used
9     reg [7:0] ADC4; // Not Used
10    wire signed [23:0] audio_out ;
11    wire GainLED ;
12    integer i ;
13
14    cwb_gain_pedal DUT(.audio_in(audio_in) ,
15                      .sys_clk(sys_clk) ,
16                      .I2C_ADDR(I2C_ADDR) ,
17                      .ENABLE_I2C_data_move(
18                           ENABLE_I2C_data_move) ,
19                           .ADC1_Gain(ADC1_Gain) ,
20                           .ADC2(ADC2) ,
21                           .ADC3(ADC3) ,
22                           .ADC4(ADC4) ,
23                           .audio_out(audio_out) ,
```

```
23          . GainLED( GainLED ) ,  
24          . temp_GAIN( temp_GAIN ) );  
25  
26  
27 // Define Clock Interval  
28 always #1 sys_clk = ~sys_clk;  
29  
30 initial  
31 begin  
32     sys_clk <= 0;  
33     I2C_ADDR <= 8'h43;  
34     ENABLE_I2C_data_move <= 1;  
35     ADC1_Gain <= 8'hF0; //0-255 Range  
36     audio_in <= 0;  
37  
38 // Inactive  
39     ADC2 <=0;  
40     ADC3 <=0;  
41     ADC4<=0;  
42  
43 // Give some time  
44 for ( i=1;i<200000;i=i+5) begin  
45     #2 audio_in <= i ;  
46 end  
47
```

```
48
49     for ( i=200000;i>-200000;i=i-5) begin
50         #2 audio_in <= i ;
51     end
52
53     for ( i=-200000;i<0;i=i+5) begin
54         #2 audio_in <= i ;
55     end
56
57
58     #10 $finish ;
59
60 endmodule
```

Listing II.2: Gain Algorithm Verilog Test Bench File

Overdrive Pedal Code

```
1 module cwb_overdrive_eff(
2     input signed [23:0] audio_in ,
3     input sys_clk ,
4     input [7:0] I2C_ADDR ,
5     input ENABLE_I2C_data_move ,
6     input [7:0] ADC1, // Gain
7     input [7:0] ADC2, // Thresh
8     input [7:0] ADC3, // Not Used
9     input [7:0] ADC4, // Not Used
10    output reg signed [23:0] audio_out ,
11    output reg OverdriveLED
12 // output reg [23:0] THRESH_TEST_pos ,
13 // output reg [23:0] THRESH_TEST_neg
14 );
15
16 /*
17 24 bit audio comes into the block - active:
18 Drive controls gain/boost of OG signal
19 Level Control controls cut off volume
20 Output signal
21 */
22
23 // Control Registers
```

```
24 reg PedalON = 0;  
25  
26 reg [7:0] ADC1reg = 0;  
27 reg [7:0] ADC2reg = 0;  
28 reg [7:0] ADC3reg = 0;  
29 reg [7:0] ADC4reg = 0;  
30  
31 reg [7:0] ADC1regtemp = 0;  
32 reg [7:0] ADC2regtemp = 0;  
33 reg [7:0] ADC3regtemp = 0;  
34 reg [7:0] ADC4regtemp = 0;  
35  
36 reg [23:0] GAIN = 0;  
37 reg [31:0] Gain1 = 0;  
38 reg [23:0] THRESH = 0;  
39  
40 //End Control Registers  
41 //
```

```
42  
43  
44 //
```

```
45
46 always @ (posedge sys_clk) begin
47     if(ENABLE_I2C_data_move == 1) begin // values are stable
48         and not changing so we can store them
49         if(I2C_ADDR == 8'h47) begin // correct address to store
50             values
51             if(ADC1+ADC2 != 0) begin // this will be used later
52                 as a audio bypass
53                 PedalON = 1;
54             end else begin
55                 PedalON = 0;
56             end
57             // Set the values of the actual regs to have data
58             // be displayed to other modules
59             ADC1reg = ADC1;
60             ADC2reg = ADC2;
61             //ADC3reg = ADC3;
62             //ADC4reg = ADC4;
63             // set temp reg values for when not addr or not
64             valid data move
65             ADC1regtemp = ADC1reg;
66             ADC2regtemp = ADC2reg;
67             //ADC3regtemp = ADC3reg;
68             //ADC4regtemp = ADC4reg;
69         end else begin
```

```
65      ADC1reg = ADC1reg;
66      //ADC2reg = ADC2reg;
67      //ADC3reg = ADC3reg;
68      //ADC4reg = ADC4reg;
69      end
70  end else begin
71      //nop ?
72  end
73 end
74 //end control always block - put this in each pedal effect
    module
75
76 //
```

```
77 always @ (posedge sys_clk) begin //Control the output audio
    and modification
78 OverdriveLED = PedalON;
79
80 //Let ADC2reg = Gain2 with zero padding
81 //GAIN = audio_in*ADC1reg; //Q24.0*Q4.4 = Q28.4
82 //Thresh ~ ADC2
83 THRESH = {4'h0,ADC2reg, 12'h0}; //Use ADC drive to
    determine Threshold for cutoff
84
```

```
85
86 if (PedalON==1) begin
87     if (audio_in[23] == 1'b0) begin // Positive gain
88         Gain1 = audio_in*ADC1reg; //Q24.0*Q4.4 = Q28.4
89         GAIN = Gain1[27:4];
90         if (GAIN>THRESH) begin
91             audio_out = THRESH;
92         end
93         else begin
94             audio_out = GAIN;
95         end
96     end
97     if (audio_in[23] == 1'b1) begin // Negative gain
98         Gain1 = -audio_in*ADC1reg;
99         GAIN = Gain1[27:4];
100        if (GAIN>THRESH) begin
101            audio_out = -THRESH;
102        end
103        else begin
104            audio_out = -GAIN;
105        end
106    end
107 end
108 else begin // pedal off
109     audio_out = audio_in;
```

```
110 end
111 end // end always
112
113 endmodule
```

Listing II.3: Overdrive Algorithm Verilog File

Overdrive Pedal Test Bench Code

```
21
22
23 module cwb_overdrive_v2_tb;
24 reg signed [23:0] audio_in;
25 reg sys_clk;
26 reg [7:0] I2C_ADDR;
27 reg ENABLE_I2C_data_move;
28 reg [7:0] ADC1; // Gain
29 reg [7:0] ADC2; // Thresh
30 reg [7:0] ADC3; // Not Used
31 reg [7:0] ADC4; // Not Used
32 wire signed [23:0] audio_out;
33 wire OverdriveLED;
34 integer i;
35
36 cwb_overdrive_v2 DUT(.audio_in(audio_in),
37 .sys_clk(sys_clk),
38 .I2C_ADDR(I2C_ADDR),
39 .ENABLE_I2C_data_move(ENABLE_I2C_data_move)
40 ,
41 .ADC1(ADC1),
42 .ADC2(ADC2),
43 .ADC3(ADC3),
44 .ADC4(ADC4),
```

```
44          . audio_out(audio_out) ,  
45          . OverdriveLED(OverdriveLED));  
46  
47  
48 // Define Clock Interval  
49 always #1 sys_clk = ~sys_clk;  
50  
51 initial  
52 begin  
53 sys_clk <= 0;  
54 I2C_ADDR <= 8'h47;  
55 ENABLE_I2C_data_move <= 1;  
56 audio_in <= 0;  
57 ADC1 <= 128; //0-255 Range  
58 ADC2 <=128;    //0-255 Range  
59 // Inactive  
60 ADC3 <=0;  
61 ADC4<=0;  
62  
63 // Give some time  
64 for (i=1;i<100000;i=i+5) begin  
65     #2 audio_in <= i;  
66 end  
67  
68 for (i=100000;i>-100000;i=i-5) begin
```

```
69          #2 audio_in <= i;
70      end
71
72      for (i=-100000;i<0;i=i+5) begin
73          #2 audio_in <= i;
74      end
75
76 // Test 2-----
77      ADC1 <= 16; //0-255 Range
78      ADC2 <=64;    //0-255 Range
79
80      for (i=1;i<100000;i=i+5) begin
81          #2 audio_in <= i;
82      end
83
84      for (i=100000;i>-100000;i=i-5) begin
85          #2 audio_in <= i;
86      end
87
88      for (i=-100000;i<0;i=i+5) begin
89          #2 audio_in <= i;
90      end
91
92      #10 $finish;
93  end
```

94 **endmodule**

Listing II.4: Overdrive Algorithm Verilog Test Bench File

Distortion Pedal Code

```
1 // module CWB_distortion_pedal (audio_in , audio_out , sys_clk ,
2 //      enable , distortion , level_in );
3 //      input signed [23:0] audio_in ;
4 //      output reg signed [23:0] audio_out ;
5 //      input sys_clk ;
6 //      // input controls for the distortion effect
7 //      input [7:0] enable ;
8 //      input [11:0] distortion ; // values are fractional from 0
//          to 1 – most likely will be a 12 bit ADC
9 //      input [11:0] level_in ; // values range from 4096 to 2^24
//          – may change this range later to be a bit more reasonable
10
11 module CWB_distortion_pedal(
12     input signed [23:0] audio_in ,
13     input sys_clk ,
14     input [7:0] I2C_ADDR ,
15     input ENABLE_I2C_data_move ,
16     input [7:0] ADC1 , // Level
17     input [7:0] ADC2 , // Distortion
18     input [7:0] ADC3 , // Not Used
19     input [7:0] ADC4 , // Not Used
20     output reg signed [23:0] audio_out ,
```

```
21 output reg DistortionLED
22 );
23 //output reg [23:0] testLevel ,
24 //output reg [23:0] testDIST ,
25 //output reg [23:0] testLevelN
26 //Working Case
27 //reg [30:0] tempDIST1 ;
28
29 // Control Registers
30 reg PedalON = 0;
31
32 reg [7:0] ADC1reg = 0;
33 reg [7:0] ADC2reg = 0;
34 reg [7:0] ADC3reg = 0;
35 //reg [7:0] ADC4reg = 0;
36
37 reg [7:0] ADC1regtemp = 0;
38 reg [7:0] ADC2regtemp = 0;
39 reg [7:0] ADC3regtemp = 0;
40 //reg [7:0] ADC4regtemp = 0;
41
42 //Distortion Levels for shaping
43 reg [23:0] DIST1 ;
44 reg [23:0] DIST2 ;
45 reg [23:0] DIST3 ;
```

```
46      reg [23:0] DIST4;
47      reg [23:0] DIST5;
48      reg [23:0] DIST6;
49      reg [23:0] DIST7;
50      reg [23:0] DIST8;
51      reg [23:0] DIST9;
52      reg [23:0] DIST10;
53      reg [23:0] DIST11;
54      reg [23:0] DIST12;
55      reg [23:0] DIST13;
56      reg [23:0] DIST14;
57      reg [23:0] DIST15;
58      reg [23:0] DIST16;
59      reg [23:0] DIST17;
60      reg [23:0] DIST18;
61      reg [23:0] DIST19;
62      reg [23:0] DIST20;
63      reg [23:0] DIST21;
64      reg [23:0] DIST22;
65      reg [23:0] DIST23;
66      reg [23:0] DIST24;
67      reg [23:0] DIST25;
68      reg [23:0] DIST26;
69      reg [23:0] DIST27;
70      reg [23:0] DIST28;
```



```

92      if(ENABLE_I2C_data_move == 1) begin // values are
93          stable and not changing so we can store them
94
95      if(I2C_ADDR == 8'h46) begin // correct address to
96          store values
97
98      if(ADC1+ADC2+ADC3 != 0) begin // this will be
99          used later as a audio bypass
100
101         PedalON = 1;
102
103     end else begin
104
105         PedalON = 0;
106
107     end
108
109 // Set the values of the actual regs to have
110 // data be displayed to other modules
111
112     ADC1reg = ADC1;
113
114     ADC2reg = ADC2;
115
116     ADC3reg = ADC3;
117
118 // ADC4reg = ADC4;
119
120
121 // set temp reg values for when not addr or not
122 // valid data move
123
124     ADC1regtemp = ADC1reg;
125
126     ADC2regtemp = ADC2reg;
127
128     ADC3regtemp = ADC3reg;
129
130 // ADC4regtemp = ADC4reg;
131
132 end else begin

```

```
112          ADC1reg = ADC1reg;
113          ADC2reg = ADC2reg;
114          ADC3reg = ADC3reg;
115          //ADC4reg = ADC4reg;
116      end
117  end else begin
118      //nop ?
119  end
120 end //end always block
121 //--Distortion
```

```
122
123 //always @ (posedge sys_clk) begin
124 //    // This is the main block which controls the distortion
125 //    levels
126 //    // and then how it affects the final output. The blocks
127 //    are set
128 //    // up for 48 steps in the plus and minus range. If the
129 //    number
```

```
127 //    // of blocks are changed, the step count needs to be
128 //    // changed
129 //    // always @ (posedge sys_clk) begin //ALWAYS2
130 //        // This allows the DIST to change - may want to
131 //        add another
```

```

130 //           // DIST[23:0] = (level[11:0]*.02083333333)*distortion
131 //           [11:0]; // where 1/48=steps
132 //           // Must do in steps to make sure there is no overflow
133 //           // 1/48 = .20833333333
134 //           // @Q12 -> 1/48 = FixedPoint
135 //           // DIST[23:0] = distortion[11:0] * 12'b000001010101;
136 //           // Only need lowest 12:0 to make next frac
137 //           // remove fixed point to get back to float
138 //           // DIST[23:0] = DIST[23:0] >> 8; // Shift right by 12
139 //           as that is the Q value of 12 for next frac mult
140 //           // Time to multiply back in the level component
141 //           // DIST[32:0] = DIST[23:0] * level[19:8];// will be at
142 //           // Q48.12 now
143 //           // DIST[32:0] = DIST[32:0] >> 8; // This is the final
144 //           // value equivalent to Q12.0
145
146 //           tempDIST1 = ADC2reg;
147 //           tempDIST2 = tempDIST1 >> 4;
148 //           tempDIST3 = tempDIST2 * level[15:4];
149 //           DIST = tempDIST3 >> 5;
150 //           // tempDIST3 = ADC2reg * level[11:4];
151
152 //           // testDIST = DIST;

```

```
149 //      end //end distortion calculation block
150
151 //      always @ (posedge sys_clk) begin //ALWAYS2
152
153 //          testLevel = level;
154 //          testLevelN = 1-level;
155 //      end
156
157     always @ (posedge sys_clk) begin //ALWAYS2
158
159     level[23:0] <= {4'b0000, ADC1reg, 12'b000000000000};
160
161     DIST <= ADC2reg<<4;
162
163     DIST1<=level-1*DIST;
164     DIST2<=level-2*DIST;
165     DIST3<=level-3*DIST;
166     DIST4<=level-4*DIST;
167     DIST5<=level-5*DIST;
168     DIST6<=level-6*DIST;
169     DIST7<=level-7*DIST;
170     DIST8<=level-8*DIST;
171     DIST9<=level-9*DIST;
172     DIST10<=level-10*DIST;
173     DIST11<=level-11*DIST;
```

```
174     DIST12<=level -12*DIST;  
175     DIST13<=level -13*DIST;  
176     DIST14<=level -14*DIST;  
177     DIST15<=level -15*DIST;  
178     DIST16<=level -16*DIST;  
179     DIST17<=level -17*DIST;  
180     DIST18<=level -18*DIST;  
181     DIST19<=level -19*DIST;  
182     DIST20<=level -20*DIST;  
183     DIST21<=level -21*DIST;  
184     DIST22<=level -22*DIST;  
185     DIST23<=level -23*DIST;  
186     DIST24<=level -24*DIST;  
187     DIST25<=level -25*DIST;  
188     DIST26<=level -26*DIST;  
189     DIST27<=level -27*DIST;  
190     DIST28<=level -28*DIST;  
191     DIST29<=level -29*DIST;  
192     DIST30<=level -30*DIST;  
193     DIST31<=level -31*DIST;  
194     DIST32<=level -32*DIST;  
195     DIST33<=level -33*DIST;  
196     DIST34<=level -34*DIST;  
197     DIST35<=level -35*DIST;  
198     DIST36<=level -36*DIST;
```

```

199     DIST37<=level -37*DIST;
200     DIST38<=level -38*DIST;
201     DIST39<=level -39*DIST;
202 end
203
204 always @(posedge sys_clk) begin
205
206     if(PedalON == 1) begin // Start pedal ON
207         // DistortionLED = PedalON;
208         // -Positive
209
210         if(audio_in[23] == 0) begin
211             if(audio_in >= level) begin
212                 audio_out <= level;
213             end
214             else if(audio_in >= DIST1 && audio_in <
215                     level) begin
216                 audio_out <= DIST1;
217             end
218             else if(audio_in >= DIST2 && audio_in <
219                     DIST1) begin
220                 audio_out <= DIST2;
221             end
222         end
223     end
224 end

```

```
219      else if (audio_in>=DIST3 && audio_in<DIST2
220          ) begin
221              audio_out <= DIST3;
222          end
223      else if (audio_in>=DIST4 && audio_in<DIST3
224          ) begin
225              audio_out <= DIST4;
226          end
227      else if (audio_in>=DIST5 && audio_in<DIST4
228          ) begin
229              audio_out <= DIST5;
230          end
231      else if (audio_in>=DIST6 && audio_in<DIST5
232          ) begin
233              audio_out <= DIST6;
234          end
235      else if (audio_in>=DIST7 && audio_in<DIST6
236          ) begin
```

```
237     else if (audio_in>=DIST9 && audio_in<DIST8
238         ) begin
239             audio_out <= DIST9;
240         end
241     else if (audio_in>=DIST10 && audio_in<
242             DIST9) begin
243                 audio_out <= DIST10;
244             end
245     else if (audio_in>=DIST11 && audio_in<
246             DIST10) begin
247                 audio_out <= DIST11;
248             end
249     else if (audio_in>=DIST12 && audio_in<
250             DIST11) begin
251                 audio_out <= DIST12;
252             end
253     else if (audio_in>=DIST13 && audio_in<
254             DIST12) begin
255                 audio_out <= DIST13;
256             end
257     else if (audio_in>=DIST14 && audio_in<
258             DIST13) begin
259                 audio_out <= DIST14;
260             end
261         end
262     end
263 end
```

```
255      else if (audio_in>=DIST15 && audio_in<
256          DIST14) begin
257              audio_out <= DIST15;
258      end
259      else if (audio_in>=DIST16 && audio_in<
260          DIST15) begin
261              audio_out <= DIST16;
262      end
263      else if (audio_in>=DIST17 && audio_in<
264          DIST16) begin
265              audio_out <= DIST17;
266      end
267      else if (audio_in>=DIST18 && audio_in<
268          DIST17) begin
269              audio_out <= DIST18;
270      end
271      else if (audio_in>=DIST19 && audio_in<
272          DIST18) begin
273              audio_out <= DIST19;
274      end
275      else if (audio_in>=DIST20 && audio_in<
276          DIST19) begin
277              audio_out <= DIST20;
278      end
279  end
```

```
273     else if (audio_in>=DIST21 && audio_in<
274             DIST20) begin
275             audio_out <= DIST21;
276         end
277     else if (audio_in>=DIST22 && audio_in<
278             DIST21) begin
279             audio_out <= DIST22;
280         end
281     else if (audio_in>=DIST23 && audio_in<
282             DIST22) begin
283             audio_out <= DIST23;
284         end
285     else if (audio_in>=DIST24 && audio_in<
286             DIST23) begin
287             audio_out <= DIST24;
288         end
289     else if (audio_in>=DIST25 && audio_in<
290             DIST24) begin
291             audio_out <= DIST25;
292         end
293     else if (audio_in>=DIST26 && audio_in<
294             DIST25) begin
295             audio_out <= DIST26;
296         end
297     end
298 end
```

```
291      else if (audio_in>=DIST27 && audio_in<
292          DIST26) begin
293          audio_out <= DIST27;
294      end
295      else if (audio_in>=DIST28 && audio_in<
296          DIST27) begin
297          audio_out <= DIST28;
298      end
299      else if (audio_in>=DIST29 && audio_in<
300          DIST28) begin
301          audio_out <= DIST29;
302      end
303      else if (audio_in>=DIST30 && audio_in<
304          DIST29) begin
305          audio_out <= DIST30;
306      end
307      else if (audio_in>=DIST31 && audio_in<
308          DIST30) begin
309          audio_out <= DIST31;
310      end
311      else if (audio_in>=DIST32 && audio_in<
312          DIST31) begin
313          audio_out <= DIST32;
314      end
315  end
```

```
309      else if (audio_in>=DIST33 && audio_in<
310          DIST32) begin
311          audio_out <= DIST33;
312      end
313      else if (audio_in>=DIST34 && audio_in<
314          DIST33) begin
315          audio_out <= DIST34;
316      end
317      else if (audio_in>=DIST35 && audio_in<
318          DIST34) begin
319          audio_out <= DIST35;
320      end
321      else if (audio_in>=DIST36 && audio_in<
322          DIST35) begin
323          audio_out <= DIST36;
324      end
325      else if (audio_in>=DIST37 && audio_in<
326          DIST36) begin
327          audio_out <= DIST37;
328      end
329      else if (audio_in>=DIST38 && audio_in<
330          DIST37) begin
331          audio_out <= DIST38;
332      end
```

```

327         else if (audio_in >= DIST39 && audio_in <
328             DIST38) begin
329             audio_out <= DIST39;
330         end
331         // Catch case
332         else begin // if (audio_in < DIST24) begin
333             audio_out <= audio_in; // Plus one for
334             debugging but not noticeable in
335             audio out
336         end
337
338
339 // -Negative

```

```

340     else if (audio_in[23] == 1) begin
341
342     // NOTE - while the equality looks fucked... the
343     //           equality is bitwise which does not actually
344     //           account
345
346     //           for the signage... only the binary value
347     //           - dont let this trip you up at 3am idiot!

```

```
344         if (-audio_in >= level) begin
345             audio_out <= -level;
346         end
347         else if (-audio_in >= DIST1 && -audio_in <
348             level) begin
349             audio_out <= -DIST1;
350         end
350         else if (-audio_in>=DIST2 && -audio_in<
351             DIST1) begin
351             audio_out <= -DIST2;
352         end
353         else if (-audio_in>=DIST3 && -audio_in<
354             DIST2) begin
354             audio_out <= -DIST3;
355         end
356         else if (-audio_in>=DIST4 && -audio_in<
357             DIST3) begin
357             audio_out <= -DIST4;
358         end
359         else if (-audio_in>=DIST5 && -audio_in<
360             DIST4) begin
360             audio_out <= -DIST5;
361         end
362         else if (-audio_in>=DIST6 && -audio_in<
362             DIST5) begin
```

```
363           audio_out <= -DIST6;
364       end
365   else if (-audio_in>=DIST7 && -audio_in<
366           DIST6) begin
367       audio_out <= -DIST7;
368   end
369   else if (-audio_in>=DIST8 && -audio_in<
370           DIST7) begin
371       audio_out <= -DIST8;
372   end
373   else if (-audio_in>=DIST9 && -audio_in<
374           DIST8) begin
375       audio_out <= -DIST9;
376   end
377   else if (-audio_in>=DIST10 && -audio_in<
378           DIST9) begin
379       audio_out <= -DIST10;
380   end
381   else if (-audio_in>=DIST11 && -audio_in<
382           DIST10) begin
383       audio_out <= -DIST11;
384   end
385   else if (-audio_in>=DIST12 && -audio_in<
386           DIST11) begin
387       audio_out <= -DIST12;
```

```
382         end
383     else if (-audio_in>=DIST13 && -audio_in<
384             DIST12) begin
385         audio_out <= -DIST13;
386     end
387     else if (-audio_in>=DIST14 && -audio_in<
388             DIST13) begin
389         audio_out <= -DIST14;
390     end
391     else if (-audio_in>=DIST15 && -audio_in<
392             DIST14) begin
393         audio_out <= -DIST15;
394     end
395     else if (-audio_in>=DIST16 && -audio_in<
396             DIST15) begin
397         audio_out <= -DIST16;
398     end
399     else if (-audio_in>=DIST17 && -audio_in<
400             DIST16) begin
```

```
401      else if (-audio_in>=DIST19 && -audio_in<
402          DIST18) begin
403              audio_out  <=  -DIST19;
404          end
405      else if (-audio_in>=DIST20 && -audio_in<
406          DIST19) begin
407              audio_out  <=  -DIST20;
408          end
409      else if (-audio_in>=DIST21 && -audio_in<
410          DIST20) begin
411              audio_out  <=  -DIST21;
412          end
413      else if (-audio_in>=DIST22 && -audio_in<
414          DIST21) begin
415              audio_out  <=  -DIST22;
416          end
417      else if (-audio_in>=DIST23 && -audio_in<
418          DIST22) begin
419              audio_out  <=  -DIST23;
420          end
421      else if (-audio_in>=DIST24 && -audio_in<
422          DIST23) begin
423              audio_out  <=  -DIST24;
424          end
425      end
426  end
```

```
419      else if (-audio_in>=DIST25 && -audio_in<
420          DIST24) begin
421              audio_out  <=  -DIST25;
422          end
423      else if (-audio_in>=DIST26 && -audio_in<
424          DIST25) begin
425              audio_out  <=  -DIST26;
426          end
427      else if (-audio_in>=DIST27 && -audio_in<
428          DIST26) begin
429              audio_out  <=  -DIST27;
430          end
431      else if (-audio_in>=DIST28 && -audio_in<
432          DIST27) begin
433              audio_out  <=  -DIST28;
434          end
435      else if (-audio_in>=DIST29 && -audio_in<
436          DIST28) begin
```

```
437      else if (-audio_in>=DIST31 && -audio_in<
438          DIST30) begin
439              audio_out  <=  -DIST31;
440          end
441      else if (-audio_in>=DIST32 && -audio_in<
442          DIST31) begin
443              audio_out  <=  -DIST32;
444          end
445      else if (-audio_in>=DIST33 && -audio_in<
446          DIST32) begin
447              audio_out  <=  -DIST33;
448          end
449      else if (-audio_in>=DIST34 && -audio_in<
450          DIST33) begin
451              audio_out  <=  -DIST34;
452          end
453      else if (-audio_in>=DIST35 && -audio_in<
454          DIST34) begin
```

```
455           else if (-audio_in>=DIST37 && -audio_in<
456                           DIST36) begin
457               audio_out  <=  -DIST37;
458           end
459           else if (-audio_in>=DIST38 && -audio_in<
460                           DIST37) begin
461               audio_out  <=  -DIST38;
462           end
463           else if (-audio_in>=DIST39 && -audio_in<
464                           DIST38) begin
465               audio_out  <=  -DIST39;
466           end
467           // Catch case
468           else begin
469               audio_out <= audio_in; // Plus one for
470                               debugging but not noticeable in
471                               audio out
472           end
473           // else
474           audio_out = audio_in; // test value to see
475           why its not entering loop
476       end // End pedal ON
```

```
474
475      // Pedal is in off state
476      else begin
477          audio_out <= audio_in; // test value to see why its
                           not entering loop
478      end
479      DistortionLED <= PedalON;
480  end //ALWAYS2
481 endmodule
```

Listing II.5: Distortion Algorithm Verilog File

Distortion Pedal Test Bench Code

```
1 // 'timescale 1ps / 1ps
2
3 // module cwb_distortion_tb;
4 // reg [23:0] audio_in_tb;
5 // wire [23:0] audio_out_tb;
6 // reg sys_clk_tb;
7 // reg [7:0] enable_tb;
8 // reg [11:0] distortion_tb;
9 // reg [11:0] level_tb;
10
11 module cwb_distortion_tb;
12 //--new module
13 reg signed [23:0] audio_in;
14 reg sys_clk;
15 reg [7:0] I2C_ADDR;
16 reg ENABLE_I2C_data_move;
17 reg [7:0] ADC1; // Level
18 reg [7:0] ADC2; // Distortion
19 reg [7:0] ADC3; // Not Used
20 reg [7:0] ADC4; // Not Used
21 wire signed [23:0] audio_out;
22 wire DistortionLED;
23 integer i;
```

```
24
25 CWB_distortion_pedal DUT(. audio_in(audio_in) ,
26           . sys_clk(sys_clk) ,
27           . I2C_ADDR(I2C_ADDR) ,
28           . ENABLE_I2C_data_move(ENABLE_I2C_data_move)
29           ,
30           . ADC1(ADC1) ,
31           . ADC2(ADC2) ,
32           . ADC3(ADC3) ,
33           . ADC4(ADC4) ,
34           . audio_out(audio_out) ,
35           . DistortionLED(DistortionLED));
36
37 // Define Clock Interval
38
39 initial
40 begin
41   sys_clk <= 0;
42   I2C_ADDR <= 8'h46;
43   ENABLE_I2C_data_move <= 1;
44   audio_in <= 0;
45   ADC1 <= 250; // 0-255 Range
46   ADC2 <= 127; // 0-255 Range
47   // Inactive
```

```
48     ADC3 <=0;
49     ADC4<=0;
50
51 // Give some time
52 for( i=1;i<100000;i=i+10) begin
53     #2 audio_in <= i;
54 end
55
56 for( i=100000;i>-100000;i=i-10) begin
57     #2 audio_in <= i;
58 end
59
60 for( i=-100000;i<0;i=i+10) begin
61     #2 audio_in <= i;
62 end
63
64 // Test 2-----
```

```
65     ADC1 <= 16; //0-255 Range
66     ADC2 <=127; //0-255 Range
67
68 for( i=1;i<100000;i=i+10) begin
69     #2 audio_in <= i;
70 end
71
72 for( i=100000;i>-100000;i=i-10) begin
```

```
73      #2 audio_in <= i;
74  end
75
76  for (i=-100000;i<0;i=i+10) begin
77      #2 audio_in <= i;
78  end
79
80 // Test 3-----
81      ADC1 <= 16; // 0-255 Range
82      ADC2 <=255; // 0-255 Range
83
84  for (i=1;i<100000;i=i+10) begin
85      #2 audio_in <= i;
86  end
87
88  for (i=100000;i>-100000;i=i-10) begin
89      #2 audio_in <= i;
90  end
91
92  for (i=-100000;i<0;i=i+10) begin
93      #2 audio_in <= i;
94  end
95
96 // Test 4-----
97      ADC1 <= 64; // 0-255 Range
```

```
98     ADC2 <=16;      // 0-255 Range
99
100    for (i=1;i<300000;i=i+100) begin
101        #2 audio_in <= i ;
102    end
103
104    for (i=300000;i>-300000;i=i-100) begin
105        #2 audio_in <= i ;
106    end
107
108    for (i=-300000;i<0;i=i+100) begin
109        #2 audio_in <= i ;
110    end
111
112 // Test 5-----
113     ADC1 <= 64; // 0-255 Range
114     ADC2 <= 64; // 0-255 Range
115
116    for (i=1;i<300000;i=i+10) begin
117        #2 audio_in <= i ;
118    end
119
120    for (i=300000;i>-300000;i=i-10) begin
121        #2 audio_in <= i ;
122    end
```

```
123
124      for ( i=-300000;i<0; i=i+10) begin
125          #2 audio_in <= i ;
126      end
127
128      #10 $finish ;
129
130  end
131 endmodule
```

Listing II.6: Distortion Algorithm Verilog Test Bench File

Delay Pedal Code

```
1 module cwb_delay
2     input signed [23:0] audio_in ,
3     input sys_clk ,
4     input [7:0] I2C_ADDR ,
5     input ENABLE_I2C_data_move ,
6     input [7:0] ADC1, // Time Step
7     input [7:0] ADC2, // Dry / Wet Mix
8     input [7:0] ADC3, // FIR / NOTFIR (1/0)
9     input [7:0] ADC4, // Not Used
10    output reg signed [23:0] audio_out ,
11    output reg DelayLED);
12 //     output reg [15:0] ADDR1out,
13 //     output reg [15:0] ADDR2out,
14 //     output reg [15:0] Max_DelayOUT,
15 //     output reg [23:0] BRAM_OUT);
16
17 parameter Samples = 65536;
18 parameter Dwidth = 16;
19
20 // Control Registers
21 reg PedalONDelay = 0;
22
23 reg [7:0] ADC1reg = 0;
```

```
24      reg [7:0] ADC2reg = 0;
25      reg [7:0] ADC3reg = 0;
26 // reg [7:0] ADC4reg = 0;
27
28      reg [7:0] ADC1regtemp = 0;
29      reg [7:0] ADC2regtemp = 0;
30      reg [7:0] ADC3regtemp = 0;
31 // reg [7:0] ADC4regtemp = 0;
32
33 //RAM specific Registers
34 reg signed [23:0] tempRAM = 0;
35 wire WEN = 1;
36 reg [Dwidth-1:0] Addr1 = 0;
37 reg [Dwidth-1:0] Addr2 = 15'b0000000000000001;
38 wire [Dwidth-1:0] Addr1_IN;
39 wire [Dwidth-1:0] Addr2_IN;
40
41 reg signed [23:0] Din;
42 wire signed [23:0] Dout;
43 reg [Dwidth-1:0] Max_Delay; // = Samples-1;
44 reg [Dwidth-1:0] i = 0;
45
46 // Buffer variables
47 reg signed [31:0] temp_Din;
48 reg signed [23:0] Din_accum;
```

```
49     reg signed [23:0] temp_audio_in ;
50     reg signed [23:0] Dout_clocked ;
51
52     reg isFIR ;
53 //Dont think I need this cause not using presets - rather
      POTs
54
55 // Call BRAM to be used in this block
56 // The following must be inserted into your Verilog file for
      this
57 // core to be instantiated. Change the instance name and port
      connections
58 // (in parentheses) to your own signal names.
59 blk_mem_gen_0 DelayBRAM (
60     .clka(sys_clk),      // input wire clka
61     .wea(WEN),          // input wire [0 : 0] wea
62     .addr(a),           // input wire [14 : 0] addr
63     .dina(Din),         // input wire [23 : 0] dina
64     .clkb(sys_clk),      // input wire clk
65     .addrb(Addr2_IN),    // input wire [14 : 0] addrb
66     .doutb(Dout)        // output wire [23 : 0] doutb
67 );
68 //
```

```
69 // Control Loop Always block to store ADC data
70     always @ (posedge sys_clk) begin
71         if(ENABLE_I2C_data_move == 1) begin // values are
72             stable and not changing so we can store them
73             if(I2C_ADDR == 8'h49) begin // correct address to
74                 store values
75             if(ADC1+ADC2+ADC3 != 0) begin // this will be
76                 used later as a audio bypass
77                 PedalONDelay = 1;
78             end else begin
79                 PedalONDelay = 0;
80             end
81             // Set the values of the actual regs to have
82             // data be displayed to other modules
83             ADC1reg = ADC1;
84             ADC2reg = ADC2;
85             ADC3reg = ADC3;
86             // ADC4reg = ADC4;
87
88             // set temp reg values for when not addr or not
89             // valid data move
90             ADC1regtemp = ADC1reg;
91             ADC2regtemp = ADC2reg;
92             ADC3regtemp = ADC3reg;
93             // ADC4regtemp = ADC4reg;
```

```
89         end else begin
90             ADC1reg = ADC1reg;
91             ADC2reg = ADC2reg;
92             ADC3reg = ADC3reg;
93             // ADC4reg = ADC4reg;
94         end
95     end else begin
96         // nop ?
97     end
98     // end // end always
99
100    //--Main Process
```

```

101    // always @(posedge sys_clk) begin // Address indexing
102        Max_Delay = {ADC1reg,8'b11111110};
103        isFIR = ADC3reg[7]; // 1 = fir , 0 = IIR
104        DelayLED = PedalONDelay; // Set LED value
105        //Max_DelayOUT = Max_Delay; // Debug
106    // end
107
108    // always @(posedge sys_clk) begin // Address indexing
109        if (Addr1 == Max_Delay-2) begin
110            Addr1 = 15'b0000000000000000;
111        end

```

```
112         else begin
113             Addr1 = Addr1 + 1;
114         end
115         // ADDR1out = Addr1; // Debug
116     // end
117
118     // always @(posedge sys_clk) begin // Address indexing
119     // if (Addr2 == Max_Delay - 2 && PedalONDelay == 1)
120         begin
121             if (Addr2 == Max_Delay - 2) begin
122                 Addr2 = 15'b0000000000000000;
123             end
124             else begin
125                 Addr2 = Addr2 + 1;
126             end
127             // ADDR2out = Addr2;
128             // CollisionRW <= (Addr2 == Addr1);
129         end
130
131     // always @(posedge sys_clk) begin // Main audio processsing
132         block
133             // Start calling the delay into BRAM
134             if (PedalONDelay == 0) begin // Device OFF
135                 Din = 24'b000000000000000000000000;
```

```
135         audio_out = audio_in;  
136     end  
137     else if(PedalONDelay == 1) begin  
138         // read value from BRAM  
139         tempRAM = Dout;  
140         //BRAM_OUT = tempRAM; //DEBUG  
141         audio_out = audio_in + tempRAM;  
142  
143         // Output stages  
144         //FIR implementation -> only repeats one note  
145         and is working  
146         if(isFIR == 1) begin  
147             if(audio_in[23] == 0) begin // positive  
148                 temp_audio_in = audio_in;  
149                 temp_Din = temp_audio_in*ADC2reg; //Q24  
150                 .0*Q0.8= Q24.8  
151                 Din = (temp_Din[31:8]);  
152             end  
153             else begin // negative  
154                 temp_audio_in = -audio_in;  
155                 temp_Din = temp_audio_in*ADC2reg; //Q24  
156                 .0*Q0.8= Q24.8  
157                 Din = -temp_Din[31:8];  
158             end  
159         end
```

```
157
158          // IIR implementation -> continual feedback
159      else begin
160          Din_accum = audio_in+tempRAM;
161          if(Din_accum[23] == 0) begin // positive
162              temp_audio_in = Din_accum;
163              temp_Din = temp_audio_in*ADC2reg; //Q24
164                  .0*Q0.8= Q24.8
165          end
166          else begin // negative
167              temp_audio_in = -Din_accum;
168              temp_Din = temp_audio_in*ADC2reg; //Q24
169                  .0*Q0.8= Q24.8
170          end
171      end // End IIR
172  end // end pedal on case
173 end
174
175 assign Addr1_IN = Addr1;
176 assign Addr2_IN = Addr2;
177
178 endmodule
```

Listing II.7: Delay Algorithm Verilog File

Delay Pedal Test Bench Code

```
1  `timescale 1us/1ns
2
3  module cwb_delay_tb;
4  reg signed [23:0] audio_in;
5  reg sys_clk;
6  reg [7:0] I2C_ADDR;
7  reg ENABLE_I2C_data_move;
8  reg [7:0] ADC1; // Delay time
9  reg [7:0] ADC2; // Depth
10 reg [7:0] ADC3; // isFIR
11 reg [7:0] ADC4; // Not Used
12 wire signed [23:0] audio_out;
13 wire DelayLED;
14 wire [14:0] ADDR1out;
15 wire [14:0] ADDR2out;
16 wire [14:0]Max_DelayOUT;
17 integer i;
18 integer j;
19 integer Iter = 5;
20 integer Iter2 = 25;
21 wire signed [23:0] BRAM_OUT;
22
23 cwb_delay dut(.audio_in(audio_in),
```

```
24          .sys_clk(sys_clk) ,
25          .I2C_ADDR(I2C_ADDR) ,
26          .ENABLE_I2C_data_move(ENABLE_I2C_data_move) ,
27          .ADC1(ADC1) ,
28          .ADC2(ADC2) ,
29          .ADC3(ADC3) ,
30          .ADC4(ADC4) ,
31          .audio_out(audio_out) ,
32          .DelayLED(DelayLED)) ;
33 //          .ADDR1out(ADDR1out) ,
34 //          .ADDR2out(ADDR2out) ,
35 //          .Max_DelayOUT(Max_DelayOUT) ,
36 //          .BRAM_OUT(BRAM_OUT)) ;
37
38 // Define Clock Interval
39 always #5 sys_clk = ~sys_clk ;
40
41 initial
42 begin
43     sys_clk <= 0;
44     I2C_ADDR <= 8'h49;
45     ENABLE_I2C_data_move <= 1;
46     audio_in <= 0;
47
48     ADC1 <= 0; // 0-255 Range - Time
```

```
49     ADC2 <=0; // 0-255 Range - Dry/WetMix
50     ADC3 <=0; // 0-255 Range - FIR(1)/IIR(0)
51 // Inactive
52
53 #50000      ADC1 <= 255; // 0-255 Range - Time
54 ADC2 <=255; // 0-255 Range - Dry/WetMix
55 ADC3 <=255; // 0-255 Range - FIR(1)/IIR(0)
56
57
58 for(j=1;j<Iter ;j=j+1) begin
59     // Give some time
60     for(i=1;i<150000;i=i+1000) begin
61         #10 audio_in <= i ;
62     end
63
64     for(i=150000;i>-150000;i=i-1000) begin
65         #10 audio_in <= i ;
66     end
67     for(i=-150000;i<1;i=i+1000) begin
68         #10 audio_in <= i ;
69     end
70 end
71
72 for(j=1;j<Iter2 ;j=j+1) begin
73     // Give some time
```

```
74      for ( i=1;i<300000;i=i+1000) begin
75          #10 audio_in <= i ;
76      end
77
78      for ( i=300000;i>-300000;i=i-1000) begin
79          #10 audio_in <= i ;
80      end
81      for ( i=-300000;i<1;i=i+1000) begin
82          #10 audio_in <= i ;
83      end
84  end
85
86 #1000000
87 ADC1 <= 100; //0-255 Range - Time
88 ADC2 <=128; //0-255 Range - Dry/WetMix
89 ADC3 <=0; //0-255 Range - FIR(1)/IIR(0)
90
91
92      for ( j=1;j<Iter2 ;j=j+1) begin
93          // Give some time
94          for ( i=1;i<300000;i=i+1000) begin
95              #10 audio_in <= i ;
96          end
97
98      for ( i=300000;i>-300000;i=i-1000) begin
```

```
99          #10 audio_in <= i;  
100         end  
101         for (i=-300000;i<1;i=i+1000) begin  
102             #10 audio_in <= i;  
103         end  
104     end  
105  
106     #500000 $finish;  
107  
108 end  
109 endmodule
```

Listing II.8: Delay Algorithm Verilog Test Bench File

Chorus Pedal Code

```

1 module cwb_chorus(
2     input signed [23:0] audio_in ,
3     input sys_clk ,
4     input [7:0] I2C_ADDR ,
5     input ENABLE_I2C_data_move ,
6     input [7:0] ADC1, // Max Delay
7     input [7:0] ADC2, // Dry / Wet Mix
8     input [7:0] ADC3, // LO depth
9     input [7:0] ADC4, // Not Used
10    output reg signed [23:0] audio_out ,
11    output reg ChorusLED ,
12    output reg [15:0] Addr1_debug ,
13    output reg [15:0] Addr2_debug ,
14    output reg signed [16:0] k_debug ,
15    output reg signed [23:0] Chorus1 ,
16    output reg signed [23:0] BRAMOUT,
17    output reg ConflictingWR ,
18    output reg DIR_DEBUG,
19    output reg [15:0] MAXDELAY_DEBUG,
20    output reg [15:0] Index_DEBUG );
21
22 parameter Dwidth = 16;
23 parameter LO_depth = 5; //How depth the LO extends

```

```
24
25      // Control Registers
26      reg PedalON = 0;
27
28      reg [7:0] ADC1reg = 0;
29      reg [7:0] ADC2reg = 0;
30      reg [7:0] ADC3reg = 0;
31      // reg [7:0] ADC4reg = 0;
32
33      reg [7:0] ADC1regtemp = 0;
34      reg [7:0] ADC2regtemp = 0;
35      reg [7:0] ADC3regtemp = 0;
36      // reg [7:0] ADC4regtemp = 0;
37
38      //RAM specific Registers
39      reg signed [23:0] tempRAM = 0;
40      reg WEN = 1;
41      reg [Dwidth-1:0] Addr1 = 16'b0000000000000000;
42      reg [Dwidth-1:0] Addr2 = 16'b0000000000000000;
43      // Offset by the number of interpolation memory
44      reg signed [23:0] Din;
45      wire signed [23:0] Dout;
46      reg [31:0] temp_Din;
47      reg signed [23:0] temp_audio_in;
48
```

```
49
50    //LO and Chorus Specific
51    reg [Dwidth-1:0] Max_Delay; // = Samples -1;
52    reg [Dwidth-1:0] LO_freq;
53    reg [Dwidth-1:0] LO_counter = 0;
54    reg LO_change = 0;
55
56    reg [Dwidth-1:0] i = 16'b0000000000000001; //used for
57        ADDR2 memory pointing
58    reg [Dwidth-1:0] k = 16'b0000000000000000; //500 minimum
59        //Used for chorus LO index
60    reg signed [7:0] LO_depth_counter = 0;
61
62    //Registers used to implement Linear Interpolation -
63        Interpolation by factor of 2
64    reg LO_dir = 1; //1 for up and 0 for down
65    reg [15:0] interp_index = 16'h0000;
66
67    reg signed [23:0] DelayRead0;
68    reg signed [23:0] DelayRead1;
69    reg signed [23:0] DelayRead2;
70    reg signed [23:0] DelayRead3;
71    reg signed [23:0] DelayRead4;
72    reg signed [23:0] DelayRead5;
73    reg signed [23:0] DelayRead6;
```

```
71     reg signed [23:0] DelayRead7;
72     reg signed [23:0] DelayRead8;
73     reg signed [23:0] DelayRead9;
74     reg signed [23:0] DelayRead10;
75     reg signed [23:0] DelayRead11;
76     reg signed [23:0] DelayRead12;
77     reg signed [23:0] DelayRead13;
78     reg signed [23:0] DelayRead14;
79     reg signed [23:0] DelayRead15;
80     reg signed [23:0] DelayRead16;
81     reg signed [23:0] DelayRead17;
82     reg signed [23:0] DelayRead18;
83     reg signed [23:0] DelayRead19;
84     reg signed [23:0] DelayRead20;
85     reg signed [23:0] DelayRead21;
86     reg signed [23:0] DelayRead22;
87     reg signed [23:0] DelayRead23;
88     reg signed [23:0] DelayRead24;
89     reg signed [23:0] DelayRead25;
90     reg signed [23:0] DelayRead26;
91     reg signed [23:0] DelayRead27;
92     reg signed [23:0] DelayRead28;
93     reg signed [23:0] DelayRead29;
94     reg signed [23:0] DelayRead30;
95     reg signed [23:0] DelayRead31;
```

```
96      reg signed [23:0] DelayRead32;
97      reg signed [23:0] DelayRead33;
98      reg signed [23:0] DelayRead34;
99      reg signed [23:0] DelayRead35;
100     reg signed [23:0] DelayRead36;
101     reg signed [23:0] DelayRead37;
102     reg signed [23:0] DelayRead38;
103     reg signed [23:0] DelayRead39;
104     reg signed [23:0] DelayRead40;
105     reg signed [23:0] DelayRead41;
106     reg signed [23:0] DelayRead42;
107     reg signed [23:0] DelayRead43;
108     reg signed [23:0] DelayRead44;
109     reg signed [23:0] DelayRead45;
110     reg signed [23:0] DelayRead46;
111     reg signed [23:0] DelayRead47;
112     reg signed [23:0] DelayRead48;
113     reg signed [23:0] DelayRead49;
114     reg signed [23:0] DelayRead50;
115     reg signed [23:0] DelayRead51;
116     reg signed [23:0] DelayRead52;
117     reg signed [23:0] DelayRead53;
118     reg signed [23:0] DelayRead54;
119     reg signed [23:0] DelayRead55;
120     reg signed [23:0] DelayRead56;
```

```
121     reg signed [23:0] DelayRead57;
122     reg signed [23:0] DelayRead58;
123     reg signed [23:0] DelayRead59;
124     reg signed [23:0] DelayRead60;
125     reg signed [23:0] DelayRead61;
126     reg signed [23:0] DelayRead62;
127     reg signed [23:0] DelayRead63;
128     reg signed [23:0] DelayRead64;
129     reg signed [23:0] DelayRead65;
130     reg signed [23:0] DelayRead66;
131     reg signed [23:0] DelayRead67;
132     reg signed [23:0] DelayRead68;
133     reg signed [23:0] DelayRead69;
134     reg signed [23:0] DelayRead70;
135     reg signed [23:0] DelayRead71;
136     reg signed [23:0] DelayRead72;
137     reg signed [23:0] DelayRead73;
138     reg signed [23:0] DelayRead74;
139     reg signed [23:0] DelayRead75;
140     reg signed [23:0] DelayRead76;
141     reg signed [23:0] DelayRead77;
142     reg signed [23:0] DelayRead78;
143     reg signed [23:0] DelayRead79;
144     reg signed [23:0] DelayRead80;
145     reg signed [23:0] DelayRead81;
```

```
146     reg signed [23:0] DelayRead82;
147     reg signed [23:0] DelayRead83;
148     reg signed [23:0] DelayRead84;
149     reg signed [23:0] DelayRead85;
150     reg signed [23:0] DelayRead86;
151     reg signed [23:0] DelayRead87;
152     reg signed [23:0] DelayRead88;
153     reg signed [23:0] DelayRead89;
154     reg signed [23:0] DelayRead90;
155     reg signed [23:0] DelayRead91;
156     reg signed [23:0] DelayRead92;
157     reg signed [23:0] DelayRead93;
158     reg signed [23:0] DelayRead94;
159     reg signed [23:0] DelayRead95;
160     reg signed [23:0] DelayRead96;
161     reg signed [23:0] DelayRead97;
162     reg signed [23:0] DelayRead98;
163     reg signed [23:0] DelayRead99;
164     reg signed [23:0] DelayRead100;
165     reg signed [23:0] DelayRead101;
166     reg signed [23:0] DelayRead102;
167     reg signed [23:0] DelayRead103;
168     reg signed [23:0] DelayRead104;
169     reg signed [23:0] DelayRead105;
170     reg signed [23:0] DelayRead106;
```

```
171     reg signed [23:0] DelayRead107;
172     reg signed [23:0] DelayRead108;
173     reg signed [23:0] DelayRead109;
174     reg signed [23:0] DelayRead110;
175     reg signed [23:0] DelayRead111;
176     reg signed [23:0] DelayRead112;
177     reg signed [23:0] DelayRead113;
178     reg signed [23:0] DelayRead114;
179     reg signed [23:0] DelayRead115;
180     reg signed [23:0] DelayRead116;
181     reg signed [23:0] DelayRead117;
182     reg signed [23:0] DelayRead118;
183     reg signed [23:0] DelayRead119;
184     reg signed [23:0] DelayRead120;
185     reg signed [23:0] DelayRead121;
186     reg signed [23:0] DelayRead122;
187     reg signed [23:0] DelayRead123;
188     reg signed [23:0] DelayRead124;
189     reg signed [23:0] DelayRead125;
190     reg signed [23:0] DelayRead126;
191     reg signed [23:0] DelayRead127;
192     reg signed [23:0] DelayRead128;
193
194
195
```

```
196     reg signed [23:0] Interpolation_Accum;
197
198
199 // Call BRAM to be used in this block
200 // The following must be inserted into your Verilog file for
201 // this
202 // core to be instantiated. Change the instance name and port
203 // connections
204 // (in parentheses) to your own signal names.
205
206 chorus_ram ChorusBRAM (
207     .clka(sys_clk),      // input wire clka
208     .wea(WEN),          // input wire [0 : 0] wea
209     .addr(a),           // input wire [15 : 0] addr
210     .dina(Din),         // input wire [23 : 0] dina
211     .clkb(sys_clk),      // input wire clk
212     .addrb(Addr2),       // input wire [15 : 0] addrb
213     .doutb(Dout)        // output wire [23 : 0] doutb
214 );
215
216 ///////////////////////////////////////////////////////////////////
217
218
219 // Control Loop Always block to store ADC data
220
221 always @ (posedge sys_clk) begin
222     if (ENABLE_I2C_data_move == 1) begin // values are
223         stable and not changing so we can store them
224     end
225 end
```

```
216      if(I2C_ADDR == 8'h47) begin // correct address to
217          store values
218
219          if(ADC1+ADC2+ADC3 != 0) begin // this will be
220              used later as a audio bypass
221
222          PedalON = 1;
223
224          end else begin
225
226          PedalON = 0;
227
228          end
229
230          // Set the values of the actual regs to have
231
232          data be displayed to other modules
233
234          ADC1reg = ADC1;
235
236          ADC2reg = ADC2;
237
238          ADC3reg = ADC3;
239
240          // ADC4reg = ADC4;
241
242
243          // set temp reg values for when not addr or not
244
245          valid data move
246
247          ADC1regtemp = ADC1reg;
248
249          ADC2regtemp = ADC2reg;
250
251          ADC3regtemp = ADC3reg;
252
253          // ADC4regtemp = ADC4reg;
254
255          end else begin
256
257          ADC1reg = ADC1reg;
258
259          ADC2reg = ADC2reg;
260
261          ADC3reg = ADC3reg;
```

```
237          // ADC4reg = ADC4reg;
238      end
239  end else begin
240      // nop ?
241  end
242 end // end always
243
244 // Create LO Clock
```

```
245     always @ (posedge sys_clk) begin
246         LO_freq    <= {8'b00000000,ADC3reg};
247
248         if (LO_counter < LO_freq) begin
249             LO_counter <= LO_counter +1;
250             LO_change <= 0;
251         end
252         else begin // greater than or equal
253             LO_counter <= 0;
254             LO_change <= 1;
255         end
256
257         // Let LO_depth_counter count up to depth and then down
258         for neg depth.
```

```
258     if(LO_change == 1 && LO_dir == 1 && interp_index <
259         1023) begin //count up
260         interp_index <= interp_index + 1;
261     end
262     else if(LO_change == 1 && LO_dir == 1 && interp_index
263             == 1023) begin //change direction and count down
264             next
265             interp_index <= interp_index - 1;
266             LO_dir <= ~LO_dir;
267         end
268     else if(LO_change == 1 && LO_dir == 0 && interp_index
269             > 0) begin //count down
270             interp_index <= interp_index - 1;
271         end
272     else if(LO_change == 1 && LO_dir == 0 && interp_index
273             == 0) begin //change direction and count up next
274             interp_index <= interp_index + 1;
275             LO_dir <= ~LO_dir;
276         end
277
278     DIR_DEBUG <= LO_dir; //DEBUG
279     Index_DEBUG <= interp_index; //DEBUG
280
281
282 end //end always
```

```
277 //--Index Creation
-----
278     always @ (posedge sys_clk) begin // Main audio processsing
279         block
280             // Chorus between 10ms and 25ms Delay range
281             // 500 + concatenated value
282             Max_Delay <= 16'b0000000111110100 + {6'b000000,
283             ADC1reg,2'b11};
284
285             // Set the write address and read address index
286             if (Addr1 == Max_Delay -2) begin
287                 Addr1 <= 16'b0000000000000000;
288             end
289             else begin
290                 Addr1 <= Addr1 + 1;
291             end
292
293             // Address for Addr2
294             if (i == Max_Delay -2) begin
295                 i <= 16'b0000000000000000;
296             end
297             else begin
298                 i <= i + 1;
299             end
```

```
298
299         // Set the read address index
300         if(Addr2 >= (Max_Delay - 2)) begin
301             Addr2 <= 16'b0000000000000000;
302         end
303         else begin
304             Addr2 <= i + 1;
305         end
306
307
308         MAXDELAY_DEBUG <= Max_Delay - 2;
309         k_debug <= LO_depth_counter; //DEBUG
310         Addr1_debug <= Addr1; // Debug
311         Addr2_debug <= Addr2; // Debug
312         ConflictingWR <= (Addr1 == Addr2); // Debug
313     end // end index process
314
315
316 //--Interpolation Process
```

```
317     always @(posedge sys_clk) begin
318         // Assign the delayed values from the BRAM so we can do
319         interpolation
```

```
319     DelayRead0 <= Dout; //From the BRAM -This is the intial
                     delayed signal
320 DelayRead1 <= DelayRead0;
321     DelayRead2 <= DelayRead1;
322     DelayRead3 <= DelayRead2;
323     DelayRead4 <= DelayRead3;
324     DelayRead5 <= DelayRead4;
325     DelayRead6 <= DelayRead5;
326     DelayRead7 <= DelayRead6;
327     DelayRead8 <= DelayRead7;
328     DelayRead9 <= DelayRead8;
329     DelayRead10 <= DelayRead9;
330     DelayRead11 <= DelayRead10;
331     DelayRead12 <= DelayRead11;
332     DelayRead13 <= DelayRead12;
333     DelayRead14 <= DelayRead13;
334     DelayRead15 <= DelayRead14;
335     DelayRead16 <= DelayRead15;
336     DelayRead17 <= DelayRead16;
337     DelayRead18 <= DelayRead17;
338     DelayRead19 <= DelayRead18;
339     DelayRead20 <= DelayRead19;
340     DelayRead21 <= DelayRead20;
341     DelayRead22 <= DelayRead21;
342     DelayRead23 <= DelayRead22;
```

```
343     DelayRead24 <= DelayRead23 ;
344     DelayRead25 <= DelayRead24 ;
345     DelayRead26 <= DelayRead25 ;
346     DelayRead27 <= DelayRead26 ;
347     DelayRead28 <= DelayRead27 ;
348     DelayRead29 <= DelayRead28 ;
349     DelayRead30 <= DelayRead29 ;
350     DelayRead31 <= DelayRead30 ;
351     DelayRead32 <= DelayRead31 ;
352     DelayRead33 <= DelayRead32 ;
353     DelayRead34 <= DelayRead33 ;
354     DelayRead35 <= DelayRead34 ;
355     DelayRead36 <= DelayRead35 ;
356     DelayRead37 <= DelayRead36 ;
357     DelayRead38 <= DelayRead37 ;
358     DelayRead39 <= DelayRead38 ;
359     DelayRead40 <= DelayRead39 ;
360     DelayRead41 <= DelayRead40 ;
361     DelayRead42 <= DelayRead41 ;
362     DelayRead43 <= DelayRead42 ;
363     DelayRead44 <= DelayRead43 ;
364     DelayRead45 <= DelayRead44 ;
365     DelayRead46 <= DelayRead45 ;
366     DelayRead47 <= DelayRead46 ;
367     DelayRead48 <= DelayRead47 ;
```

```
368     DelayRead49 <= DelayRead48 ;
369     DelayRead50 <= DelayRead49 ;
370     DelayRead51 <= DelayRead50 ;
371     DelayRead52 <= DelayRead51 ;
372     DelayRead53 <= DelayRead52 ;
373     DelayRead54 <= DelayRead53 ;
374     DelayRead55 <= DelayRead54 ;
375     DelayRead56 <= DelayRead55 ;
376     DelayRead57 <= DelayRead56 ;
377     DelayRead58 <= DelayRead57 ;
378     DelayRead59 <= DelayRead58 ;
379     DelayRead60 <= DelayRead59 ;
380     DelayRead61 <= DelayRead60 ;
381     DelayRead62 <= DelayRead61 ;
382     DelayRead63 <= DelayRead62 ;
383     DelayRead64 <= DelayRead63 ;
384     DelayRead65 <= DelayRead64 ;
385     DelayRead66 <= DelayRead65 ;
386     DelayRead67 <= DelayRead66 ;
387     DelayRead68 <= DelayRead67 ;
388     DelayRead69 <= DelayRead68 ;
389     DelayRead70 <= DelayRead69 ;
390     DelayRead71 <= DelayRead70 ;
391     DelayRead72 <= DelayRead71 ;
392     DelayRead73 <= DelayRead72 ;
```

```
393     DelayRead74 <= DelayRead73 ;
394     DelayRead75 <= DelayRead74 ;
395     DelayRead76 <= DelayRead75 ;
396     DelayRead77 <= DelayRead76 ;
397     DelayRead78 <= DelayRead77 ;
398     DelayRead79 <= DelayRead78 ;
399     DelayRead80 <= DelayRead79 ;
400     DelayRead81 <= DelayRead80 ;
401     DelayRead82 <= DelayRead81 ;
402     DelayRead83 <= DelayRead82 ;
403     DelayRead84 <= DelayRead83 ;
404     DelayRead85 <= DelayRead84 ;
405     DelayRead86 <= DelayRead85 ;
406     DelayRead87 <= DelayRead86 ;
407     DelayRead88 <= DelayRead87 ;
408     DelayRead89 <= DelayRead88 ;
409     DelayRead90 <= DelayRead89 ;
410     DelayRead91 <= DelayRead90 ;
411     DelayRead92 <= DelayRead91 ;
412     DelayRead93 <= DelayRead92 ;
413     DelayRead94 <= DelayRead93 ;
414     DelayRead95 <= DelayRead94 ;
415     DelayRead96 <= DelayRead95 ;
416     DelayRead97 <= DelayRead96 ;
417     DelayRead98 <= DelayRead97 ;
```

```
418     DelayRead99 <= DelayRead98 ;
419     DelayRead100 <= DelayRead99 ;
420     DelayRead101 <= DelayRead100 ;
421     DelayRead102 <= DelayRead101 ;
422     DelayRead103 <= DelayRead102 ;
423     DelayRead104 <= DelayRead103 ;
424     DelayRead105 <= DelayRead104 ;
425     DelayRead106 <= DelayRead105 ;
426     DelayRead107 <= DelayRead106 ;
427     DelayRead108 <= DelayRead107 ;
428     DelayRead109 <= DelayRead108 ;
429     DelayRead110 <= DelayRead109 ;
430     DelayRead111 <= DelayRead110 ;
431     DelayRead112 <= DelayRead111 ;
432     DelayRead113 <= DelayRead112 ;
433     DelayRead114 <= DelayRead113 ;
434     DelayRead115 <= DelayRead114 ;
435     DelayRead116 <= DelayRead115 ;
436     DelayRead117 <= DelayRead116 ;
437     DelayRead118 <= DelayRead117 ;
438     DelayRead119 <= DelayRead118 ;
439     DelayRead120 <= DelayRead119 ;
440     DelayRead121 <= DelayRead120 ;
441     DelayRead122 <= DelayRead121 ;
442     DelayRead123 <= DelayRead122 ;
```

```
443     DelayRead124 <= DelayRead123 ;
444     DelayRead125 <= DelayRead124 ;
445     DelayRead126 <= DelayRead125 ;
446     DelayRead127 <= DelayRead126 ;
447     DelayRead128 <= DelayRead127 ;
448
449 end // end always
450
451
452 always @(posedge sys_clk) begin
453     case(interp_index)
454         16'h0000:     Interpolation_Accum <= (DelayRead0
455                         >>>3)*8;
456         16'h0001:     Interpolation_Accum <= (DelayRead0
457                         >>>3)*7 + (DelayRead1>>>3)*1;
458         16'h0002:     Interpolation_Accum <= (DelayRead0
459                         >>>3)*6 + (DelayRead1>>>3)*2;
460         16'h0003:     Interpolation_Accum <= (DelayRead0
461                         >>>3)*5 + (DelayRead1>>>3)*3;
462         16'h0004:     Interpolation_Accum <= (DelayRead0
463                         >>>3)*4 + (DelayRead1>>>3)*4;
464         16'h0005:     Interpolation_Accum <= (DelayRead0
465                         >>>3)*3 + (DelayRead1>>>3)*5;
466         16'h0006:     Interpolation_Accum <= (DelayRead0
467                         >>>3)*2 + (DelayRead1>>>3)*6;
```

```
461      16'h0007:     Interpolation_Accum <= (DelayRead0
                  >>>3)*1 + (DelayRead1>>>3)*7;
462      16'h0008:     Interpolation_Accum <= (DelayRead1
                  >>>3)*8 + (DelayRead2>>>3)*0;
463      16'h0009:     Interpolation_Accum <= (DelayRead1
                  >>>3)*7 + (DelayRead2>>>3)*1;
464      16'h000A:     Interpolation_Accum <= (DelayRead1
                  >>>3)*6 + (DelayRead2>>>3)*2;
465      16'h000B:     Interpolation_Accum <= (DelayRead1
                  >>>3)*5 + (DelayRead2>>>3)*3;
466      16'h000C:     Interpolation_Accum <= (DelayRead1
                  >>>3)*4 + (DelayRead2>>>3)*4;
467      16'h000D:     Interpolation_Accum <= (DelayRead1
                  >>>3)*3 + (DelayRead2>>>3)*5;
468      16'h000E:     Interpolation_Accum <= (DelayRead1
                  >>>3)*2 + (DelayRead2>>>3)*6;
469      16'h000F:     Interpolation_Accum <= (DelayRead1
                  >>>3)*1 + (DelayRead2>>>3)*7;
470      16'h0010:     Interpolation_Accum <= (DelayRead2
                  >>>3)*8 + (DelayRead3>>>3)*0;
471      16'h0011:     Interpolation_Accum <= (DelayRead2
                  >>>3)*7 + (DelayRead3>>>3)*1;
472      16'h0012:     Interpolation_Accum <= (DelayRead2
                  >>>3)*6 + (DelayRead3>>>3)*2;
```

```
473      16'h0013 :     Interpolation_Accum <= (DelayRead2
               >>>3)*5 + (DelayRead3>>>3)*3;
474      16'h0014 :     Interpolation_Accum <= (DelayRead2
               >>>3)*4 + (DelayRead3>>>3)*4;
475      16'h0015 :     Interpolation_Accum <= (DelayRead2
               >>>3)*3 + (DelayRead3>>>3)*5;
476      16'h0016 :     Interpolation_Accum <= (DelayRead2
               >>>3)*2 + (DelayRead3>>>3)*6;
477      16'h0017 :     Interpolation_Accum <= (DelayRead2
               >>>3)*1 + (DelayRead3>>>3)*7;
478      16'h0018 :     Interpolation_Accum <= (DelayRead3
               >>>3)*8 + (DelayRead4>>>3)*0;
479      16'h0019 :     Interpolation_Accum <= (DelayRead3
               >>>3)*7 + (DelayRead4>>>3)*1;
480      16'h001A :     Interpolation_Accum <= (DelayRead3
               >>>3)*6 + (DelayRead4>>>3)*2;
481      16'h001B :     Interpolation_Accum <= (DelayRead3
               >>>3)*5 + (DelayRead4>>>3)*3;
482      16'h001C :     Interpolation_Accum <= (DelayRead3
               >>>3)*4 + (DelayRead4>>>3)*4;
483      16'h001D :     Interpolation_Accum <= (DelayRead3
               >>>3)*3 + (DelayRead4>>>3)*5;
484      16'h001E :     Interpolation_Accum <= (DelayRead3
               >>>3)*2 + (DelayRead4>>>3)*6;
```

```
485      16'h001F:    Interpolation_Accum <= (DelayRead3
               >>>3)*1 + (DelayRead4>>>3)*7;
486      16'h0020:    Interpolation_Accum <= (DelayRead4
               >>>3)*8 + (DelayRead5>>>3)*0;
487      16'h0021:    Interpolation_Accum <= (DelayRead4
               >>>3)*7 + (DelayRead5>>>3)*1;
488      16'h0022:    Interpolation_Accum <= (DelayRead4
               >>>3)*6 + (DelayRead5>>>3)*2;
489      16'h0023:    Interpolation_Accum <= (DelayRead4
               >>>3)*5 + (DelayRead5>>>3)*3;
490      16'h0024:    Interpolation_Accum <= (DelayRead4
               >>>3)*4 + (DelayRead5>>>3)*4;
491      16'h0025:    Interpolation_Accum <= (DelayRead4
               >>>3)*3 + (DelayRead5>>>3)*5;
492      16'h0026:    Interpolation_Accum <= (DelayRead4
               >>>3)*2 + (DelayRead5>>>3)*6;
493      16'h0027:    Interpolation_Accum <= (DelayRead4
               >>>3)*1 + (DelayRead5>>>3)*7;
494      16'h0028:    Interpolation_Accum <= (DelayRead5
               >>>3)*8 + (DelayRead6>>>3)*0;
495      16'h0029:    Interpolation_Accum <= (DelayRead5
               >>>3)*7 + (DelayRead6>>>3)*1;
496      16'h002A:    Interpolation_Accum <= (DelayRead5
               >>>3)*6 + (DelayRead6>>>3)*2;
```

```
497      16'h002B :     Interpolation_Accum <= (DelayRead5
               >>>3)*5 + (DelayRead6>>>3)*3;
498      16'h002C :     Interpolation_Accum <= (DelayRead5
               >>>3)*4 + (DelayRead6>>>3)*4;
499      16'h002D :     Interpolation_Accum <= (DelayRead5
               >>>3)*3 + (DelayRead6>>>3)*5;
500      16'h002E :     Interpolation_Accum <= (DelayRead5
               >>>3)*2 + (DelayRead6>>>3)*6;
501      16'h002F :     Interpolation_Accum <= (DelayRead5
               >>>3)*1 + (DelayRead6>>>3)*7;
502      16'h0030 :     Interpolation_Accum <= (DelayRead6
               >>>3)*8 + (DelayRead7>>>3)*0;
503      16'h0031 :     Interpolation_Accum <= (DelayRead6
               >>>3)*7 + (DelayRead7>>>3)*1;
504      16'h0032 :     Interpolation_Accum <= (DelayRead6
               >>>3)*6 + (DelayRead7>>>3)*2;
505      16'h0033 :     Interpolation_Accum <= (DelayRead6
               >>>3)*5 + (DelayRead7>>>3)*3;
506      16'h0034 :     Interpolation_Accum <= (DelayRead6
               >>>3)*4 + (DelayRead7>>>3)*4;
507      16'h0035 :     Interpolation_Accum <= (DelayRead6
               >>>3)*3 + (DelayRead7>>>3)*5;
508      16'h0036 :     Interpolation_Accum <= (DelayRead6
               >>>3)*2 + (DelayRead7>>>3)*6;
```

```
509      16'h0037:    Interpolation_Accum <= (DelayRead6
               >>>3)*1 + (DelayRead7>>>3)*7;
510      16'h0038:    Interpolation_Accum <= (DelayRead7
               >>>3)*8 + (DelayRead8>>>3)*0;
511      16'h0039:    Interpolation_Accum <= (DelayRead7
               >>>3)*7 + (DelayRead8>>>3)*1;
512      16'h003A:    Interpolation_Accum <= (DelayRead7
               >>>3)*6 + (DelayRead8>>>3)*2;
513      16'h003B:    Interpolation_Accum <= (DelayRead7
               >>>3)*5 + (DelayRead8>>>3)*3;
514      16'h003C:    Interpolation_Accum <= (DelayRead7
               >>>3)*4 + (DelayRead8>>>3)*4;
515      16'h003D:    Interpolation_Accum <= (DelayRead7
               >>>3)*3 + (DelayRead8>>>3)*5;
516      16'h003E:    Interpolation_Accum <= (DelayRead7
               >>>3)*2 + (DelayRead8>>>3)*6;
517      16'h003F:    Interpolation_Accum <= (DelayRead7
               >>>3)*1 + (DelayRead8>>>3)*7;
518      16'h0040:    Interpolation_Accum <= (DelayRead8
               >>>3)*8 + (DelayRead9>>>3)*0;
519      16'h0041:    Interpolation_Accum <= (DelayRead8
               >>>3)*7 + (DelayRead9>>>3)*1;
520      16'h0042:    Interpolation_Accum <= (DelayRead8
               >>>3)*6 + (DelayRead9>>>3)*2;
```

```
521      16'h0043 :     Interpolation_Accum <= (DelayRead8
           >>>3)*5 + (DelayRead9>>>3)*3;
522      16'h0044 :     Interpolation_Accum <= (DelayRead8
           >>>3)*4 + (DelayRead9>>>3)*4;
523      16'h0045 :     Interpolation_Accum <= (DelayRead8
           >>>3)*3 + (DelayRead9>>>3)*5;
524      16'h0046 :     Interpolation_Accum <= (DelayRead8
           >>>3)*2 + (DelayRead9>>>3)*6;
525      16'h0047 :     Interpolation_Accum <= (DelayRead8
           >>>3)*1 + (DelayRead9>>>3)*7;
526      16'h0048 :     Interpolation_Accum <= (DelayRead9
           >>>3)*8 + (DelayRead10>>>3)*0;
527      16'h0049 :     Interpolation_Accum <= (DelayRead9
           >>>3)*7 + (DelayRead10>>>3)*1;
528      16'h004A :     Interpolation_Accum <= (DelayRead9
           >>>3)*6 + (DelayRead10>>>3)*2;
529      16'h004B :     Interpolation_Accum <= (DelayRead9
           >>>3)*5 + (DelayRead10>>>3)*3;
530      16'h004C :     Interpolation_Accum <= (DelayRead9
           >>>3)*4 + (DelayRead10>>>3)*4;
531      16'h004D :     Interpolation_Accum <= (DelayRead9
           >>>3)*3 + (DelayRead10>>>3)*5;
532      16'h004E :     Interpolation_Accum <= (DelayRead9
           >>>3)*2 + (DelayRead10>>>3)*6;
```

```
533      16'h004F:    Interpolation_Accum <= (DelayRead9
               >>>3)*1 + (DelayRead10>>>3)*7;
534      16'h0050:    Interpolation_Accum <= (DelayRead10
               >>>3)*8 + (DelayRead11>>>3)*0;
535      16'h0051:    Interpolation_Accum <= (DelayRead10
               >>>3)*7 + (DelayRead11>>>3)*1;
536      16'h0052:    Interpolation_Accum <= (DelayRead10
               >>>3)*6 + (DelayRead11>>>3)*2;
537      16'h0053:    Interpolation_Accum <= (DelayRead10
               >>>3)*5 + (DelayRead11>>>3)*3;
538      16'h0054:    Interpolation_Accum <= (DelayRead10
               >>>3)*4 + (DelayRead11>>>3)*4;
539      16'h0055:    Interpolation_Accum <= (DelayRead10
               >>>3)*3 + (DelayRead11>>>3)*5;
540      16'h0056:    Interpolation_Accum <= (DelayRead10
               >>>3)*2 + (DelayRead11>>>3)*6;
541      16'h0057:    Interpolation_Accum <= (DelayRead10
               >>>3)*1 + (DelayRead11>>>3)*7;
542      16'h0058:    Interpolation_Accum <= (DelayRead11
               >>>3)*8 + (DelayRead12>>>3)*0;
543      16'h0059:    Interpolation_Accum <= (DelayRead11
               >>>3)*7 + (DelayRead12>>>3)*1;
544      16'h005A:    Interpolation_Accum <= (DelayRead11
               >>>3)*6 + (DelayRead12>>>3)*2;
```

```
545      16'h005B :     Interpolation_Accum <= (DelayRead11
               >>>3)*5 + (DelayRead12>>>3)*3;
546      16'h005C :     Interpolation_Accum <= (DelayRead11
               >>>3)*4 + (DelayRead12>>>3)*4;
547      16'h005D :     Interpolation_Accum <= (DelayRead11
               >>>3)*3 + (DelayRead12>>>3)*5;
548      16'h005E :     Interpolation_Accum <= (DelayRead11
               >>>3)*2 + (DelayRead12>>>3)*6;
549      16'h005F :     Interpolation_Accum <= (DelayRead11
               >>>3)*1 + (DelayRead12>>>3)*7;
550      16'h0060 :     Interpolation_Accum <= (DelayRead12
               >>>3)*8 + (DelayRead13>>>3)*0;
551      16'h0061 :     Interpolation_Accum <= (DelayRead12
               >>>3)*7 + (DelayRead13>>>3)*1;
552      16'h0062 :     Interpolation_Accum <= (DelayRead12
               >>>3)*6 + (DelayRead13>>>3)*2;
553      16'h0063 :     Interpolation_Accum <= (DelayRead12
               >>>3)*5 + (DelayRead13>>>3)*3;
554      16'h0064 :     Interpolation_Accum <= (DelayRead12
               >>>3)*4 + (DelayRead13>>>3)*4;
555      16'h0065 :     Interpolation_Accum <= (DelayRead12
               >>>3)*3 + (DelayRead13>>>3)*5;
556      16'h0066 :     Interpolation_Accum <= (DelayRead12
               >>>3)*2 + (DelayRead13>>>3)*6;
```

```
557      16'h0067:     Interpolation_Accum <= (DelayRead12
                  >>>3)*1 + (DelayRead13>>>3)*7;
558      16'h0068:     Interpolation_Accum <= (DelayRead13
                  >>>3)*8 + (DelayRead14>>>3)*0;
559      16'h0069:     Interpolation_Accum <= (DelayRead13
                  >>>3)*7 + (DelayRead14>>>3)*1;
560      16'h006A:     Interpolation_Accum <= (DelayRead13
                  >>>3)*6 + (DelayRead14>>>3)*2;
561      16'h006B:     Interpolation_Accum <= (DelayRead13
                  >>>3)*5 + (DelayRead14>>>3)*3;
562      16'h006C:     Interpolation_Accum <= (DelayRead13
                  >>>3)*4 + (DelayRead14>>>3)*4;
563      16'h006D:     Interpolation_Accum <= (DelayRead13
                  >>>3)*3 + (DelayRead14>>>3)*5;
564      16'h006E:     Interpolation_Accum <= (DelayRead13
                  >>>3)*2 + (DelayRead14>>>3)*6;
565      16'h006F:     Interpolation_Accum <= (DelayRead13
                  >>>3)*1 + (DelayRead14>>>3)*7;
566      16'h0070:     Interpolation_Accum <= (DelayRead14
                  >>>3)*8 + (DelayRead15>>>3)*0;
567      16'h0071:     Interpolation_Accum <= (DelayRead14
                  >>>3)*7 + (DelayRead15>>>3)*1;
568      16'h0072:     Interpolation_Accum <= (DelayRead14
                  >>>3)*6 + (DelayRead15>>>3)*2;
```

```
569      16'h0073:     Interpolation_Accum <= (DelayRead14  
           >>>3)*5 + (DelayRead15>>>3)*3;  
570      16'h0074:     Interpolation_Accum <= (DelayRead14  
           >>>3)*4 + (DelayRead15>>>3)*4;  
571      16'h0075:     Interpolation_Accum <= (DelayRead14  
           >>>3)*3 + (DelayRead15>>>3)*5;  
572      16'h0076:     Interpolation_Accum <= (DelayRead14  
           >>>3)*2 + (DelayRead15>>>3)*6;  
573      16'h0077:     Interpolation_Accum <= (DelayRead14  
           >>>3)*1 + (DelayRead15>>>3)*7;  
574      16'h0078:     Interpolation_Accum <= (DelayRead15  
           >>>3)*8 + (DelayRead16>>>3)*0;  
575      16'h0079:     Interpolation_Accum <= (DelayRead15  
           >>>3)*7 + (DelayRead16>>>3)*1;  
576      16'h007A:     Interpolation_Accum <= (DelayRead15  
           >>>3)*6 + (DelayRead16>>>3)*2;  
577      16'h007B:     Interpolation_Accum <= (DelayRead15  
           >>>3)*5 + (DelayRead16>>>3)*3;  
578      16'h007C:     Interpolation_Accum <= (DelayRead15  
           >>>3)*4 + (DelayRead16>>>3)*4;  
579      16'h007D:     Interpolation_Accum <= (DelayRead15  
           >>>3)*3 + (DelayRead16>>>3)*5;  
580      16'h007E:     Interpolation_Accum <= (DelayRead15  
           >>>3)*2 + (DelayRead16>>>3)*6;
```

```
581      16'h007F:    Interpolation_Accum <= (DelayRead15
                  >>>3)*1 + (DelayRead16>>>3)*7;
582      16'h0080:    Interpolation_Accum <= (DelayRead16
                  >>>3)*8 + (DelayRead17>>>3)*0;
583      16'h0081:    Interpolation_Accum <= (DelayRead16
                  >>>3)*7 + (DelayRead17>>>3)*1;
584      16'h0082:    Interpolation_Accum <= (DelayRead16
                  >>>3)*6 + (DelayRead17>>>3)*2;
585      16'h0083:    Interpolation_Accum <= (DelayRead16
                  >>>3)*5 + (DelayRead17>>>3)*3;
586      16'h0084:    Interpolation_Accum <= (DelayRead16
                  >>>3)*4 + (DelayRead17>>>3)*4;
587      16'h0085:    Interpolation_Accum <= (DelayRead16
                  >>>3)*3 + (DelayRead17>>>3)*5;
588      16'h0086:    Interpolation_Accum <= (DelayRead16
                  >>>3)*2 + (DelayRead17>>>3)*6;
589      16'h0087:    Interpolation_Accum <= (DelayRead16
                  >>>3)*1 + (DelayRead17>>>3)*7;
590      16'h0088:    Interpolation_Accum <= (DelayRead17
                  >>>3)*8 + (DelayRead18>>>3)*0;
591      16'h0089:    Interpolation_Accum <= (DelayRead17
                  >>>3)*7 + (DelayRead18>>>3)*1;
592      16'h008A:    Interpolation_Accum <= (DelayRead17
                  >>>3)*6 + (DelayRead18>>>3)*2;
```

```
593      16'h008B :     Interpolation_Accum <= (DelayRead17
594                      >>>3)*5 + (DelayRead18>>>3)*3;
594      16'h008C :     Interpolation_Accum <= (DelayRead17
595                      >>>3)*4 + (DelayRead18>>>3)*4;
595      16'h008D :     Interpolation_Accum <= (DelayRead17
596                      >>>3)*3 + (DelayRead18>>>3)*5;
596      16'h008E :     Interpolation_Accum <= (DelayRead17
597                      >>>3)*2 + (DelayRead18>>>3)*6;
597      16'h008F :     Interpolation_Accum <= (DelayRead17
598                      >>>3)*1 + (DelayRead18>>>3)*7;
598      16'h0090 :     Interpolation_Accum <= (DelayRead18
599                      >>>3)*8 + (DelayRead19>>>3)*0;
599      16'h0091 :     Interpolation_Accum <= (DelayRead18
600                      >>>3)*7 + (DelayRead19>>>3)*1;
600      16'h0092 :     Interpolation_Accum <= (DelayRead18
601                      >>>3)*6 + (DelayRead19>>>3)*2;
601      16'h0093 :     Interpolation_Accum <= (DelayRead18
602                      >>>3)*5 + (DelayRead19>>>3)*3;
602      16'h0094 :     Interpolation_Accum <= (DelayRead18
603                      >>>3)*4 + (DelayRead19>>>3)*4;
603      16'h0095 :     Interpolation_Accum <= (DelayRead18
604                      >>>3)*3 + (DelayRead19>>>3)*5;
604      16'h0096 :     Interpolation_Accum <= (DelayRead18
605                      >>>3)*2 + (DelayRead19>>>3)*6;
```

```
605      16'h0097:    Interpolation_Accum <= (DelayRead18
               >>>3)*1 + (DelayRead19>>>3)*7;
606      16'h0098:    Interpolation_Accum <= (DelayRead19
               >>>3)*8 + (DelayRead20>>>3)*0;
607      16'h0099:    Interpolation_Accum <= (DelayRead19
               >>>3)*7 + (DelayRead20>>>3)*1;
608      16'h009A:    Interpolation_Accum <= (DelayRead19
               >>>3)*6 + (DelayRead20>>>3)*2;
609      16'h009B:    Interpolation_Accum <= (DelayRead19
               >>>3)*5 + (DelayRead20>>>3)*3;
610      16'h009C:    Interpolation_Accum <= (DelayRead19
               >>>3)*4 + (DelayRead20>>>3)*4;
611      16'h009D:    Interpolation_Accum <= (DelayRead19
               >>>3)*3 + (DelayRead20>>>3)*5;
612      16'h009E:    Interpolation_Accum <= (DelayRead19
               >>>3)*2 + (DelayRead20>>>3)*6;
613      16'h009F:    Interpolation_Accum <= (DelayRead19
               >>>3)*1 + (DelayRead20>>>3)*7;
614      16'h00A0:    Interpolation_Accum <= (DelayRead20
               >>>3)*8 + (DelayRead21>>>3)*0;
615      16'h00A1:    Interpolation_Accum <= (DelayRead20
               >>>3)*7 + (DelayRead21>>>3)*1;
616      16'h00A2:    Interpolation_Accum <= (DelayRead20
               >>>3)*6 + (DelayRead21>>>3)*2;
```

```
617      16'h00A3:    Interpolation_Accum <= (DelayRead20
              >>>3)*5 + (DelayRead21>>>3)*3;
618      16'h00A4:    Interpolation_Accum <= (DelayRead20
              >>>3)*4 + (DelayRead21>>>3)*4;
619      16'h00A5:    Interpolation_Accum <= (DelayRead20
              >>>3)*3 + (DelayRead21>>>3)*5;
620      16'h00A6:    Interpolation_Accum <= (DelayRead20
              >>>3)*2 + (DelayRead21>>>3)*6;
621      16'h00A7:    Interpolation_Accum <= (DelayRead20
              >>>3)*1 + (DelayRead21>>>3)*7;
622      16'h00A8:    Interpolation_Accum <= (DelayRead21
              >>>3)*8 + (DelayRead22>>>3)*0;
623      16'h00A9:    Interpolation_Accum <= (DelayRead21
              >>>3)*7 + (DelayRead22>>>3)*1;
624      16'h00AA:    Interpolation_Accum <= (DelayRead21
              >>>3)*6 + (DelayRead22>>>3)*2;
625      16'h00AB:    Interpolation_Accum <= (DelayRead21
              >>>3)*5 + (DelayRead22>>>3)*3;
626      16'h00AC:    Interpolation_Accum <= (DelayRead21
              >>>3)*4 + (DelayRead22>>>3)*4;
627      16'h00AD:    Interpolation_Accum <= (DelayRead21
              >>>3)*3 + (DelayRead22>>>3)*5;
628      16'h00AE:    Interpolation_Accum <= (DelayRead21
              >>>3)*2 + (DelayRead22>>>3)*6;
```

```
629      16'h00AF:    Interpolation_Accum <= (DelayRead21
               >>>3)*1 + (DelayRead22>>>3)*7;
630      16'h00B0:    Interpolation_Accum <= (DelayRead22
               >>>3)*8 + (DelayRead23>>>3)*0;
631      16'h00B1:    Interpolation_Accum <= (DelayRead22
               >>>3)*7 + (DelayRead23>>>3)*1;
632      16'h00B2:    Interpolation_Accum <= (DelayRead22
               >>>3)*6 + (DelayRead23>>>3)*2;
633      16'h00B3:    Interpolation_Accum <= (DelayRead22
               >>>3)*5 + (DelayRead23>>>3)*3;
634      16'h00B4:    Interpolation_Accum <= (DelayRead22
               >>>3)*4 + (DelayRead23>>>3)*4;
635      16'h00B5:    Interpolation_Accum <= (DelayRead22
               >>>3)*3 + (DelayRead23>>>3)*5;
636      16'h00B6:    Interpolation_Accum <= (DelayRead22
               >>>3)*2 + (DelayRead23>>>3)*6;
637      16'h00B7:    Interpolation_Accum <= (DelayRead22
               >>>3)*1 + (DelayRead23>>>3)*7;
638      16'h00B8:    Interpolation_Accum <= (DelayRead23
               >>>3)*8 + (DelayRead24>>>3)*0;
639      16'h00B9:    Interpolation_Accum <= (DelayRead23
               >>>3)*7 + (DelayRead24>>>3)*1;
640      16'h00BA:    Interpolation_Accum <= (DelayRead23
               >>>3)*6 + (DelayRead24>>>3)*2;
```

```
641      16'h00BB:    Interpolation_Accum <= (DelayRead23
              >>>3)*5 + (DelayRead24>>>3)*3;
642      16'h00BC:    Interpolation_Accum <= (DelayRead23
              >>>3)*4 + (DelayRead24>>>3)*4;
643      16'h00BD:    Interpolation_Accum <= (DelayRead23
              >>>3)*3 + (DelayRead24>>>3)*5;
644      16'h00BE:    Interpolation_Accum <= (DelayRead23
              >>>3)*2 + (DelayRead24>>>3)*6;
645      16'h00BF:    Interpolation_Accum <= (DelayRead23
              >>>3)*1 + (DelayRead24>>>3)*7;
646      16'h00C0:    Interpolation_Accum <= (DelayRead24
              >>>3)*8 + (DelayRead25>>>3)*0;
647      16'h00C1:    Interpolation_Accum <= (DelayRead24
              >>>3)*7 + (DelayRead25>>>3)*1;
648      16'h00C2:    Interpolation_Accum <= (DelayRead24
              >>>3)*6 + (DelayRead25>>>3)*2;
649      16'h00C3:    Interpolation_Accum <= (DelayRead24
              >>>3)*5 + (DelayRead25>>>3)*3;
650      16'h00C4:    Interpolation_Accum <= (DelayRead24
              >>>3)*4 + (DelayRead25>>>3)*4;
651      16'h00C5:    Interpolation_Accum <= (DelayRead24
              >>>3)*3 + (DelayRead25>>>3)*5;
652      16'h00C6:    Interpolation_Accum <= (DelayRead24
              >>>3)*2 + (DelayRead25>>>3)*6;
```

```
653      16'h00C7:    Interpolation_Accum <= (DelayRead24
              >>>3)*1 + (DelayRead25>>>3)*7;
654      16'h00C8:    Interpolation_Accum <= (DelayRead25
              >>>3)*8 + (DelayRead26>>>3)*0;
655      16'h00C9:    Interpolation_Accum <= (DelayRead25
              >>>3)*7 + (DelayRead26>>>3)*1;
656      16'h00CA:    Interpolation_Accum <= (DelayRead25
              >>>3)*6 + (DelayRead26>>>3)*2;
657      16'h00CB:    Interpolation_Accum <= (DelayRead25
              >>>3)*5 + (DelayRead26>>>3)*3;
658      16'h00CC:    Interpolation_Accum <= (DelayRead25
              >>>3)*4 + (DelayRead26>>>3)*4;
659      16'h00CD:    Interpolation_Accum <= (DelayRead25
              >>>3)*3 + (DelayRead26>>>3)*5;
660      16'h00CE:    Interpolation_Accum <= (DelayRead25
              >>>3)*2 + (DelayRead26>>>3)*6;
661      16'h00CF:    Interpolation_Accum <= (DelayRead25
              >>>3)*1 + (DelayRead26>>>3)*7;
662      16'h00D0:    Interpolation_Accum <= (DelayRead26
              >>>3)*8 + (DelayRead27>>>3)*0;
663      16'h00D1:    Interpolation_Accum <= (DelayRead26
              >>>3)*7 + (DelayRead27>>>3)*1;
664      16'h00D2:    Interpolation_Accum <= (DelayRead26
              >>>3)*6 + (DelayRead27>>>3)*2;
```

```
665      16'h00D3:    Interpolation_Accum <= (DelayRead26
              >>>3)*5 + (DelayRead27>>>3)*3;
666      16'h00D4:    Interpolation_Accum <= (DelayRead26
              >>>3)*4 + (DelayRead27>>>3)*4;
667      16'h00D5:    Interpolation_Accum <= (DelayRead26
              >>>3)*3 + (DelayRead27>>>3)*5;
668      16'h00D6:    Interpolation_Accum <= (DelayRead26
              >>>3)*2 + (DelayRead27>>>3)*6;
669      16'h00D7:    Interpolation_Accum <= (DelayRead26
              >>>3)*1 + (DelayRead27>>>3)*7;
670      16'h00D8:    Interpolation_Accum <= (DelayRead27
              >>>3)*8 + (DelayRead28>>>3)*0;
671      16'h00D9:    Interpolation_Accum <= (DelayRead27
              >>>3)*7 + (DelayRead28>>>3)*1;
672      16'h00DA:    Interpolation_Accum <= (DelayRead27
              >>>3)*6 + (DelayRead28>>>3)*2;
673      16'h00DB:    Interpolation_Accum <= (DelayRead27
              >>>3)*5 + (DelayRead28>>>3)*3;
674      16'h00DC:    Interpolation_Accum <= (DelayRead27
              >>>3)*4 + (DelayRead28>>>3)*4;
675      16'h00DD:    Interpolation_Accum <= (DelayRead27
              >>>3)*3 + (DelayRead28>>>3)*5;
676      16'h00DE:    Interpolation_Accum <= (DelayRead27
              >>>3)*2 + (DelayRead28>>>3)*6;
```

```

677      16'h00DF:    Interpolation_Accum <= (DelayRead27
              >>>3)*1 + (DelayRead28>>>3)*7;
678      16'h00E0:    Interpolation_Accum <= (DelayRead28
              >>>3)*8 + (DelayRead29>>>3)*0;
679      16'h00E1:    Interpolation_Accum <= (DelayRead28
              >>>3)*7 + (DelayRead29>>>3)*1;
680      16'h00E2:    Interpolation_Accum <= (DelayRead28
              >>>3)*6 + (DelayRead29>>>3)*2;
681      16'h00E3:    Interpolation_Accum <= (DelayRead28
              >>>3)*5 + (DelayRead29>>>3)*3;
682      16'h00E4:    Interpolation_Accum <= (DelayRead28
              >>>3)*4 + (DelayRead29>>>3)*4;
683      16'h00E5:    Interpolation_Accum <= (DelayRead28
              >>>3)*3 + (DelayRead29>>>3)*5;
684      16'h00E6:    Interpolation_Accum <= (DelayRead28
              >>>3)*2 + (DelayRead29>>>3)*6;
685      16'h00E7:    Interpolation_Accum <= (DelayRead28
              >>>3)*1 + (DelayRead29>>>3)*7;
686      16'h00E8:    Interpolation_Accum <= (DelayRead29
              >>>3)*8 + (DelayRead30>>>3)*0;
687      16'h00E9:    Interpolation_Accum <= (DelayRead29
              >>>3)*7 + (DelayRead30>>>3)*1;
688      16'h00EA:    Interpolation_Accum <= (DelayRead29
              >>>3)*6 + (DelayRead30>>>3)*2;

```

```
689      16'h00EB :     Interpolation_Accum <= (DelayRead29
               >>>3)*5 + (DelayRead30>>>3)*3;
690      16'h00EC :     Interpolation_Accum <= (DelayRead29
               >>>3)*4 + (DelayRead30>>>3)*4;
691      16'h00ED :     Interpolation_Accum <= (DelayRead29
               >>>3)*3 + (DelayRead30>>>3)*5;
692      16'h00EE :     Interpolation_Accum <= (DelayRead29
               >>>3)*2 + (DelayRead30>>>3)*6;
693      16'h00EF :     Interpolation_Accum <= (DelayRead29
               >>>3)*1 + (DelayRead30>>>3)*7;
694      16'h00F0 :     Interpolation_Accum <= (DelayRead30
               >>>3)*8 + (DelayRead31>>>3)*0;
695      16'h00F1 :     Interpolation_Accum <= (DelayRead30
               >>>3)*7 + (DelayRead31>>>3)*1;
696      16'h00F2 :     Interpolation_Accum <= (DelayRead30
               >>>3)*6 + (DelayRead31>>>3)*2;
697      16'h00F3 :     Interpolation_Accum <= (DelayRead30
               >>>3)*5 + (DelayRead31>>>3)*3;
698      16'h00F4 :     Interpolation_Accum <= (DelayRead30
               >>>3)*4 + (DelayRead31>>>3)*4;
699      16'h00F5 :     Interpolation_Accum <= (DelayRead30
               >>>3)*3 + (DelayRead31>>>3)*5;
700      16'h00F6 :     Interpolation_Accum <= (DelayRead30
               >>>3)*2 + (DelayRead31>>>3)*6;
```

```
701      16'h00F7 :     Interpolation_Accum <= (DelayRead30
               >>>3)*1 + (DelayRead31>>>3)*7;
702      16'h00F8 :     Interpolation_Accum <= (DelayRead31
               >>>3)*8 + (DelayRead32>>>3)*0;
703      16'h00F9 :     Interpolation_Accum <= (DelayRead31
               >>>3)*7 + (DelayRead32>>>3)*1;
704      16'h00FA :     Interpolation_Accum <= (DelayRead31
               >>>3)*6 + (DelayRead32>>>3)*2;
705      16'h00FB :     Interpolation_Accum <= (DelayRead31
               >>>3)*5 + (DelayRead32>>>3)*3;
706      16'h00FC :     Interpolation_Accum <= (DelayRead31
               >>>3)*4 + (DelayRead32>>>3)*4;
707      16'h00FD :     Interpolation_Accum <= (DelayRead31
               >>>3)*3 + (DelayRead32>>>3)*5;
708      16'h00FE :     Interpolation_Accum <= (DelayRead31
               >>>3)*2 + (DelayRead32>>>3)*6;
709      16'h00FF :     Interpolation_Accum <= (DelayRead31
               >>>3)*1 + (DelayRead32>>>3)*7;
710      16'h0100 :     Interpolation_Accum <= (DelayRead32
               >>>3)*8 + (DelayRead33>>>3)*0;
711      16'h0101 :     Interpolation_Accum <= (DelayRead32
               >>>3)*7 + (DelayRead33>>>3)*1;
712      16'h0102 :     Interpolation_Accum <= (DelayRead32
               >>>3)*6 + (DelayRead33>>>3)*2;
```

```
713      16'h0103:    Interpolation_Accum <= (DelayRead32
               >>>3)*5 + (DelayRead33>>>3)*3;
714      16'h0104:    Interpolation_Accum <= (DelayRead32
               >>>3)*4 + (DelayRead33>>>3)*4;
715      16'h0105:    Interpolation_Accum <= (DelayRead32
               >>>3)*3 + (DelayRead33>>>3)*5;
716      16'h0106:    Interpolation_Accum <= (DelayRead32
               >>>3)*2 + (DelayRead33>>>3)*6;
717      16'h0107:    Interpolation_Accum <= (DelayRead32
               >>>3)*1 + (DelayRead33>>>3)*7;
718      16'h0108:    Interpolation_Accum <= (DelayRead33
               >>>3)*8 + (DelayRead34>>>3)*0;
719      16'h0109:    Interpolation_Accum <= (DelayRead33
               >>>3)*7 + (DelayRead34>>>3)*1;
720      16'h010A:    Interpolation_Accum <= (DelayRead33
               >>>3)*6 + (DelayRead34>>>3)*2;
721      16'h010B:    Interpolation_Accum <= (DelayRead33
               >>>3)*5 + (DelayRead34>>>3)*3;
722      16'h010C:    Interpolation_Accum <= (DelayRead33
               >>>3)*4 + (DelayRead34>>>3)*4;
723      16'h010D:    Interpolation_Accum <= (DelayRead33
               >>>3)*3 + (DelayRead34>>>3)*5;
724      16'h010E:    Interpolation_Accum <= (DelayRead33
               >>>3)*2 + (DelayRead34>>>3)*6;
```

```
725      16'h010F:    Interpolation_Accum <= (DelayRead33
              >>>3)*1 + (DelayRead34>>>3)*7;
726      16'h0110:    Interpolation_Accum <= (DelayRead34
              >>>3)*8 + (DelayRead35>>>3)*0;
727      16'h0111:    Interpolation_Accum <= (DelayRead34
              >>>3)*7 + (DelayRead35>>>3)*1;
728      16'h0112:    Interpolation_Accum <= (DelayRead34
              >>>3)*6 + (DelayRead35>>>3)*2;
729      16'h0113:    Interpolation_Accum <= (DelayRead34
              >>>3)*5 + (DelayRead35>>>3)*3;
730      16'h0114:    Interpolation_Accum <= (DelayRead34
              >>>3)*4 + (DelayRead35>>>3)*4;
731      16'h0115:    Interpolation_Accum <= (DelayRead34
              >>>3)*3 + (DelayRead35>>>3)*5;
732      16'h0116:    Interpolation_Accum <= (DelayRead34
              >>>3)*2 + (DelayRead35>>>3)*6;
733      16'h0117:    Interpolation_Accum <= (DelayRead34
              >>>3)*1 + (DelayRead35>>>3)*7;
734      16'h0118:    Interpolation_Accum <= (DelayRead35
              >>>3)*8 + (DelayRead36>>>3)*0;
735      16'h0119:    Interpolation_Accum <= (DelayRead35
              >>>3)*7 + (DelayRead36>>>3)*1;
736      16'h011A:    Interpolation_Accum <= (DelayRead35
              >>>3)*6 + (DelayRead36>>>3)*2;
```

```

737      16'h011B :     Interpolation_Accum <= (DelayRead35
                  >>>3)*5 + (DelayRead36>>>3)*3;
738      16'h011C :     Interpolation_Accum <= (DelayRead35
                  >>>3)*4 + (DelayRead36>>>3)*4;
739      16'h011D :     Interpolation_Accum <= (DelayRead35
                  >>>3)*3 + (DelayRead36>>>3)*5;
740      16'h011E :     Interpolation_Accum <= (DelayRead35
                  >>>3)*2 + (DelayRead36>>>3)*6;
741      16'h011F :     Interpolation_Accum <= (DelayRead35
                  >>>3)*1 + (DelayRead36>>>3)*7;
742      16'h0120 :     Interpolation_Accum <= (DelayRead36
                  >>>3)*8 + (DelayRead37>>>3)*0;
743      16'h0121 :     Interpolation_Accum <= (DelayRead36
                  >>>3)*7 + (DelayRead37>>>3)*1;
744      16'h0122 :     Interpolation_Accum <= (DelayRead36
                  >>>3)*6 + (DelayRead37>>>3)*2;
745      16'h0123 :     Interpolation_Accum <= (DelayRead36
                  >>>3)*5 + (DelayRead37>>>3)*3;
746      16'h0124 :     Interpolation_Accum <= (DelayRead36
                  >>>3)*4 + (DelayRead37>>>3)*4;
747      16'h0125 :     Interpolation_Accum <= (DelayRead36
                  >>>3)*3 + (DelayRead37>>>3)*5;
748      16'h0126 :     Interpolation_Accum <= (DelayRead36
                  >>>3)*2 + (DelayRead37>>>3)*6;

```

```
749      16'h0127:    Interpolation_Accum <= (DelayRead36
               >>>3)*1 + (DelayRead37>>>3)*7;
750      16'h0128:    Interpolation_Accum <= (DelayRead37
               >>>3)*8 + (DelayRead38>>>3)*0;
751      16'h0129:    Interpolation_Accum <= (DelayRead37
               >>>3)*7 + (DelayRead38>>>3)*1;
752      16'h012A:    Interpolation_Accum <= (DelayRead37
               >>>3)*6 + (DelayRead38>>>3)*2;
753      16'h012B:    Interpolation_Accum <= (DelayRead37
               >>>3)*5 + (DelayRead38>>>3)*3;
754      16'h012C:    Interpolation_Accum <= (DelayRead37
               >>>3)*4 + (DelayRead38>>>3)*4;
755      16'h012D:    Interpolation_Accum <= (DelayRead37
               >>>3)*3 + (DelayRead38>>>3)*5;
756      16'h012E:    Interpolation_Accum <= (DelayRead37
               >>>3)*2 + (DelayRead38>>>3)*6;
757      16'h012F:    Interpolation_Accum <= (DelayRead37
               >>>3)*1 + (DelayRead38>>>3)*7;
758      16'h0130:    Interpolation_Accum <= (DelayRead38
               >>>3)*8 + (DelayRead39>>>3)*0;
759      16'h0131:    Interpolation_Accum <= (DelayRead38
               >>>3)*7 + (DelayRead39>>>3)*1;
760      16'h0132:    Interpolation_Accum <= (DelayRead38
               >>>3)*6 + (DelayRead39>>>3)*2;
```

```
761      16'h0133:    Interpolation_Accum <= (DelayRead38
               >>>3)*5 + (DelayRead39>>>3)*3;
762      16'h0134:    Interpolation_Accum <= (DelayRead38
               >>>3)*4 + (DelayRead39>>>3)*4;
763      16'h0135:    Interpolation_Accum <= (DelayRead38
               >>>3)*3 + (DelayRead39>>>3)*5;
764      16'h0136:    Interpolation_Accum <= (DelayRead38
               >>>3)*2 + (DelayRead39>>>3)*6;
765      16'h0137:    Interpolation_Accum <= (DelayRead38
               >>>3)*1 + (DelayRead39>>>3)*7;
766      16'h0138:    Interpolation_Accum <= (DelayRead39
               >>>3)*8 + (DelayRead40>>>3)*0;
767      16'h0139:    Interpolation_Accum <= (DelayRead39
               >>>3)*7 + (DelayRead40>>>3)*1;
768      16'h013A:    Interpolation_Accum <= (DelayRead39
               >>>3)*6 + (DelayRead40>>>3)*2;
769      16'h013B:    Interpolation_Accum <= (DelayRead39
               >>>3)*5 + (DelayRead40>>>3)*3;
770      16'h013C:    Interpolation_Accum <= (DelayRead39
               >>>3)*4 + (DelayRead40>>>3)*4;
771      16'h013D:    Interpolation_Accum <= (DelayRead39
               >>>3)*3 + (DelayRead40>>>3)*5;
772      16'h013E:    Interpolation_Accum <= (DelayRead39
               >>>3)*2 + (DelayRead40>>>3)*6;
```

```
773      16'h013F:    Interpolation_Accum <= (DelayRead39
               >>>3)*1 + (DelayRead40>>>3)*7;
774      16'h0140:    Interpolation_Accum <= (DelayRead40
               >>>3)*8 + (DelayRead41>>>3)*0;
775      16'h0141:    Interpolation_Accum <= (DelayRead40
               >>>3)*7 + (DelayRead41>>>3)*1;
776      16'h0142:    Interpolation_Accum <= (DelayRead40
               >>>3)*6 + (DelayRead41>>>3)*2;
777      16'h0143:    Interpolation_Accum <= (DelayRead40
               >>>3)*5 + (DelayRead41>>>3)*3;
778      16'h0144:    Interpolation_Accum <= (DelayRead40
               >>>3)*4 + (DelayRead41>>>3)*4;
779      16'h0145:    Interpolation_Accum <= (DelayRead40
               >>>3)*3 + (DelayRead41>>>3)*5;
780      16'h0146:    Interpolation_Accum <= (DelayRead40
               >>>3)*2 + (DelayRead41>>>3)*6;
781      16'h0147:    Interpolation_Accum <= (DelayRead40
               >>>3)*1 + (DelayRead41>>>3)*7;
782      16'h0148:    Interpolation_Accum <= (DelayRead41
               >>>3)*8 + (DelayRead42>>>3)*0;
783      16'h0149:    Interpolation_Accum <= (DelayRead41
               >>>3)*7 + (DelayRead42>>>3)*1;
784      16'h014A:    Interpolation_Accum <= (DelayRead41
               >>>3)*6 + (DelayRead42>>>3)*2;
```

```
785      16'h014B :     Interpolation_Accum <= (DelayRead41  
           >>>3)*5 + (DelayRead42>>>3)*3;  
786      16'h014C :     Interpolation_Accum <= (DelayRead41  
           >>>3)*4 + (DelayRead42>>>3)*4;  
787      16'h014D :     Interpolation_Accum <= (DelayRead41  
           >>>3)*3 + (DelayRead42>>>3)*5;  
788      16'h014E :     Interpolation_Accum <= (DelayRead41  
           >>>3)*2 + (DelayRead42>>>3)*6;  
789      16'h014F :     Interpolation_Accum <= (DelayRead41  
           >>>3)*1 + (DelayRead42>>>3)*7;  
790      16'h0150 :     Interpolation_Accum <= (DelayRead42  
           >>>3)*8 + (DelayRead43>>>3)*0;  
791      16'h0151 :     Interpolation_Accum <= (DelayRead42  
           >>>3)*7 + (DelayRead43>>>3)*1;  
792      16'h0152 :     Interpolation_Accum <= (DelayRead42  
           >>>3)*6 + (DelayRead43>>>3)*2;  
793      16'h0153 :     Interpolation_Accum <= (DelayRead42  
           >>>3)*5 + (DelayRead43>>>3)*3;  
794      16'h0154 :     Interpolation_Accum <= (DelayRead42  
           >>>3)*4 + (DelayRead43>>>3)*4;  
795      16'h0155 :     Interpolation_Accum <= (DelayRead42  
           >>>3)*3 + (DelayRead43>>>3)*5;  
796      16'h0156 :     Interpolation_Accum <= (DelayRead42  
           >>>3)*2 + (DelayRead43>>>3)*6;
```

```
797      16'h0157:    Interpolation_Accum <= (DelayRead42
               >>>3)*1 + (DelayRead43>>>3)*7;
798      16'h0158:    Interpolation_Accum <= (DelayRead43
               >>>3)*8 + (DelayRead44>>>3)*0;
799      16'h0159:    Interpolation_Accum <= (DelayRead43
               >>>3)*7 + (DelayRead44>>>3)*1;
800      16'h015A:    Interpolation_Accum <= (DelayRead43
               >>>3)*6 + (DelayRead44>>>3)*2;
801      16'h015B:    Interpolation_Accum <= (DelayRead43
               >>>3)*5 + (DelayRead44>>>3)*3;
802      16'h015C:    Interpolation_Accum <= (DelayRead43
               >>>3)*4 + (DelayRead44>>>3)*4;
803      16'h015D:    Interpolation_Accum <= (DelayRead43
               >>>3)*3 + (DelayRead44>>>3)*5;
804      16'h015E:    Interpolation_Accum <= (DelayRead43
               >>>3)*2 + (DelayRead44>>>3)*6;
805      16'h015F:    Interpolation_Accum <= (DelayRead43
               >>>3)*1 + (DelayRead44>>>3)*7;
806      16'h0160:    Interpolation_Accum <= (DelayRead44
               >>>3)*8 + (DelayRead45>>>3)*0;
807      16'h0161:    Interpolation_Accum <= (DelayRead44
               >>>3)*7 + (DelayRead45>>>3)*1;
808      16'h0162:    Interpolation_Accum <= (DelayRead44
               >>>3)*6 + (DelayRead45>>>3)*2;
```

```
809      16'h0163 :     Interpolation_Accum <= (DelayRead44
               >>>3)*5 + (DelayRead45>>>3)*3;
810      16'h0164 :     Interpolation_Accum <= (DelayRead44
               >>>3)*4 + (DelayRead45>>>3)*4;
811      16'h0165 :     Interpolation_Accum <= (DelayRead44
               >>>3)*3 + (DelayRead45>>>3)*5;
812      16'h0166 :     Interpolation_Accum <= (DelayRead44
               >>>3)*2 + (DelayRead45>>>3)*6;
813      16'h0167 :     Interpolation_Accum <= (DelayRead44
               >>>3)*1 + (DelayRead45>>>3)*7;
814      16'h0168 :     Interpolation_Accum <= (DelayRead45
               >>>3)*8 + (DelayRead46>>>3)*0;
815      16'h0169 :     Interpolation_Accum <= (DelayRead45
               >>>3)*7 + (DelayRead46>>>3)*1;
816      16'h016A :     Interpolation_Accum <= (DelayRead45
               >>>3)*6 + (DelayRead46>>>3)*2;
817      16'h016B :     Interpolation_Accum <= (DelayRead45
               >>>3)*5 + (DelayRead46>>>3)*3;
818      16'h016C :     Interpolation_Accum <= (DelayRead45
               >>>3)*4 + (DelayRead46>>>3)*4;
819      16'h016D :     Interpolation_Accum <= (DelayRead45
               >>>3)*3 + (DelayRead46>>>3)*5;
820      16'h016E :     Interpolation_Accum <= (DelayRead45
               >>>3)*2 + (DelayRead46>>>3)*6;
```

```
821      16'h016F:    Interpolation_Accum <= (DelayRead45
               >>>3)*1 + (DelayRead46>>>3)*7;
822      16'h0170:    Interpolation_Accum <= (DelayRead46
               >>>3)*8 + (DelayRead47>>>3)*0;
823      16'h0171:    Interpolation_Accum <= (DelayRead46
               >>>3)*7 + (DelayRead47>>>3)*1;
824      16'h0172:    Interpolation_Accum <= (DelayRead46
               >>>3)*6 + (DelayRead47>>>3)*2;
825      16'h0173:    Interpolation_Accum <= (DelayRead46
               >>>3)*5 + (DelayRead47>>>3)*3;
826      16'h0174:    Interpolation_Accum <= (DelayRead46
               >>>3)*4 + (DelayRead47>>>3)*4;
827      16'h0175:    Interpolation_Accum <= (DelayRead46
               >>>3)*3 + (DelayRead47>>>3)*5;
828      16'h0176:    Interpolation_Accum <= (DelayRead46
               >>>3)*2 + (DelayRead47>>>3)*6;
829      16'h0177:    Interpolation_Accum <= (DelayRead46
               >>>3)*1 + (DelayRead47>>>3)*7;
830      16'h0178:    Interpolation_Accum <= (DelayRead47
               >>>3)*8 + (DelayRead48>>>3)*0;
831      16'h0179:    Interpolation_Accum <= (DelayRead47
               >>>3)*7 + (DelayRead48>>>3)*1;
832      16'h017A:    Interpolation_Accum <= (DelayRead47
               >>>3)*6 + (DelayRead48>>>3)*2;
```

```
833      16'h017B :     Interpolation_Accum <= (DelayRead47
               >>>3)*5 + (DelayRead48>>>3)*3;
834      16'h017C :     Interpolation_Accum <= (DelayRead47
               >>>3)*4 + (DelayRead48>>>3)*4;
835      16'h017D :     Interpolation_Accum <= (DelayRead47
               >>>3)*3 + (DelayRead48>>>3)*5;
836      16'h017E :     Interpolation_Accum <= (DelayRead47
               >>>3)*2 + (DelayRead48>>>3)*6;
837      16'h017F :     Interpolation_Accum <= (DelayRead47
               >>>3)*1 + (DelayRead48>>>3)*7;
838      16'h0180 :     Interpolation_Accum <= (DelayRead48
               >>>3)*8 + (DelayRead49>>>3)*0;
839      16'h0181 :     Interpolation_Accum <= (DelayRead48
               >>>3)*7 + (DelayRead49>>>3)*1;
840      16'h0182 :     Interpolation_Accum <= (DelayRead48
               >>>3)*6 + (DelayRead49>>>3)*2;
841      16'h0183 :     Interpolation_Accum <= (DelayRead48
               >>>3)*5 + (DelayRead49>>>3)*3;
842      16'h0184 :     Interpolation_Accum <= (DelayRead48
               >>>3)*4 + (DelayRead49>>>3)*4;
843      16'h0185 :     Interpolation_Accum <= (DelayRead48
               >>>3)*3 + (DelayRead49>>>3)*5;
844      16'h0186 :     Interpolation_Accum <= (DelayRead48
               >>>3)*2 + (DelayRead49>>>3)*6;
```

```
845      16'h0187:    Interpolation_Accum <= (DelayRead48
               >>>3)*1 + (DelayRead49>>>3)*7;
846      16'h0188:    Interpolation_Accum <= (DelayRead49
               >>>3)*8 + (DelayRead50>>>3)*0;
847      16'h0189:    Interpolation_Accum <= (DelayRead49
               >>>3)*7 + (DelayRead50>>>3)*1;
848      16'h018A:    Interpolation_Accum <= (DelayRead49
               >>>3)*6 + (DelayRead50>>>3)*2;
849      16'h018B:    Interpolation_Accum <= (DelayRead49
               >>>3)*5 + (DelayRead50>>>3)*3;
850      16'h018C:    Interpolation_Accum <= (DelayRead49
               >>>3)*4 + (DelayRead50>>>3)*4;
851      16'h018D:    Interpolation_Accum <= (DelayRead49
               >>>3)*3 + (DelayRead50>>>3)*5;
852      16'h018E:    Interpolation_Accum <= (DelayRead49
               >>>3)*2 + (DelayRead50>>>3)*6;
853      16'h018F:    Interpolation_Accum <= (DelayRead49
               >>>3)*1 + (DelayRead50>>>3)*7;
854      16'h0190:    Interpolation_Accum <= (DelayRead50
               >>>3)*8 + (DelayRead51>>>3)*0;
855      16'h0191:    Interpolation_Accum <= (DelayRead50
               >>>3)*7 + (DelayRead51>>>3)*1;
856      16'h0192:    Interpolation_Accum <= (DelayRead50
               >>>3)*6 + (DelayRead51>>>3)*2;
```

```
857      16'h0193 :     Interpolation_Accum <= (DelayRead50
               >>>3)*5 + (DelayRead51>>>3)*3;
858      16'h0194 :     Interpolation_Accum <= (DelayRead50
               >>>3)*4 + (DelayRead51>>>3)*4;
859      16'h0195 :     Interpolation_Accum <= (DelayRead50
               >>>3)*3 + (DelayRead51>>>3)*5;
860      16'h0196 :     Interpolation_Accum <= (DelayRead50
               >>>3)*2 + (DelayRead51>>>3)*6;
861      16'h0197 :     Interpolation_Accum <= (DelayRead50
               >>>3)*1 + (DelayRead51>>>3)*7;
862      16'h0198 :     Interpolation_Accum <= (DelayRead51
               >>>3)*8 + (DelayRead52>>>3)*0;
863      16'h0199 :     Interpolation_Accum <= (DelayRead51
               >>>3)*7 + (DelayRead52>>>3)*1;
864      16'h019A :     Interpolation_Accum <= (DelayRead51
               >>>3)*6 + (DelayRead52>>>3)*2;
865      16'h019B :     Interpolation_Accum <= (DelayRead51
               >>>3)*5 + (DelayRead52>>>3)*3;
866      16'h019C :     Interpolation_Accum <= (DelayRead51
               >>>3)*4 + (DelayRead52>>>3)*4;
867      16'h019D :     Interpolation_Accum <= (DelayRead51
               >>>3)*3 + (DelayRead52>>>3)*5;
868      16'h019E :     Interpolation_Accum <= (DelayRead51
               >>>3)*2 + (DelayRead52>>>3)*6;
```

```
869      16'h019F:    Interpolation_Accum <= (DelayRead51  
870          >>>3)*1 + (DelayRead52>>>3)*7;  
870      16'h01A0:    Interpolation_Accum <= (DelayRead52  
871          >>>3)*8 + (DelayRead53>>>3)*0;  
871      16'h01A1:    Interpolation_Accum <= (DelayRead52  
872          >>>3)*7 + (DelayRead53>>>3)*1;  
872      16'h01A2:    Interpolation_Accum <= (DelayRead52  
873          >>>3)*6 + (DelayRead53>>>3)*2;  
873      16'h01A3:    Interpolation_Accum <= (DelayRead52  
874          >>>3)*5 + (DelayRead53>>>3)*3;  
874      16'h01A4:    Interpolation_Accum <= (DelayRead52  
875          >>>3)*4 + (DelayRead53>>>3)*4;  
875      16'h01A5:    Interpolation_Accum <= (DelayRead52  
876          >>>3)*3 + (DelayRead53>>>3)*5;  
876      16'h01A6:    Interpolation_Accum <= (DelayRead52  
877          >>>3)*2 + (DelayRead53>>>3)*6;  
877      16'h01A7:    Interpolation_Accum <= (DelayRead52  
878          >>>3)*1 + (DelayRead53>>>3)*7;  
878      16'h01A8:    Interpolation_Accum <= (DelayRead53  
879          >>>3)*8 + (DelayRead54>>>3)*0;  
879      16'h01A9:    Interpolation_Accum <= (DelayRead53  
880          >>>3)*7 + (DelayRead54>>>3)*1;  
880      16'h01AA:    Interpolation_Accum <= (DelayRead53  
880          >>>3)*6 + (DelayRead54>>>3)*2;
```

```
881      16'h01AB:    Interpolation_Accum <= (DelayRead53
882          >>>3)*5 + (DelayRead54>>>3)*3;
883      16'h01AC:    Interpolation_Accum <= (DelayRead53
884          >>>3)*4 + (DelayRead54>>>3)*4;
885      16'h01AD:    Interpolation_Accum <= (DelayRead53
886          >>>3)*3 + (DelayRead54>>>3)*5;
887      16'h01AE:    Interpolation_Accum <= (DelayRead53
888          >>>3)*2 + (DelayRead54>>>3)*6;
889      16'h01AF:    Interpolation_Accum <= (DelayRead53
890          >>>3)*1 + (DelayRead54>>>3)*7;
891      16'h01B0:    Interpolation_Accum <= (DelayRead54
892          >>>3)*8 + (DelayRead55>>>3)*0;
893      16'h01B1:    Interpolation_Accum <= (DelayRead54
894          >>>3)*7 + (DelayRead55>>>3)*1;
895      16'h01B2:    Interpolation_Accum <= (DelayRead54
896          >>>3)*6 + (DelayRead55>>>3)*2;
897      16'h01B3:    Interpolation_Accum <= (DelayRead54
898          >>>3)*5 + (DelayRead55>>>3)*3;
899      16'h01B4:    Interpolation_Accum <= (DelayRead54
900          >>>3)*4 + (DelayRead55>>>3)*4;
901      16'h01B5:    Interpolation_Accum <= (DelayRead54
902          >>>3)*3 + (DelayRead55>>>3)*5;
903      16'h01B6:    Interpolation_Accum <= (DelayRead54
904          >>>3)*2 + (DelayRead55>>>3)*6;
```

```
893      16'h01B7:    Interpolation_Accum <= (DelayRead54
               >>>3)*1 + (DelayRead55>>>3)*7;
894      16'h01B8:    Interpolation_Accum <= (DelayRead55
               >>>3)*8 + (DelayRead56>>>3)*0;
895      16'h01B9:    Interpolation_Accum <= (DelayRead55
               >>>3)*7 + (DelayRead56>>>3)*1;
896      16'h01BA:    Interpolation_Accum <= (DelayRead55
               >>>3)*6 + (DelayRead56>>>3)*2;
897      16'h01BB:    Interpolation_Accum <= (DelayRead55
               >>>3)*5 + (DelayRead56>>>3)*3;
898      16'h01BC:    Interpolation_Accum <= (DelayRead55
               >>>3)*4 + (DelayRead56>>>3)*4;
899      16'h01BD:    Interpolation_Accum <= (DelayRead55
               >>>3)*3 + (DelayRead56>>>3)*5;
900      16'h01BE:    Interpolation_Accum <= (DelayRead55
               >>>3)*2 + (DelayRead56>>>3)*6;
901      16'h01BF:    Interpolation_Accum <= (DelayRead55
               >>>3)*1 + (DelayRead56>>>3)*7;
902      16'h01C0:    Interpolation_Accum <= (DelayRead56
               >>>3)*8 + (DelayRead57>>>3)*0;
903      16'h01C1:    Interpolation_Accum <= (DelayRead56
               >>>3)*7 + (DelayRead57>>>3)*1;
904      16'h01C2:    Interpolation_Accum <= (DelayRead56
               >>>3)*6 + (DelayRead57>>>3)*2;
```

```
905      16'h01C3:    Interpolation_Accum <= (DelayRead56  
906          >>>3)*5 + (DelayRead57>>>3)*3;  
907      16'h01C4:    Interpolation_Accum <= (DelayRead56  
908          >>>3)*4 + (DelayRead57>>>3)*4;  
909      16'h01C5:    Interpolation_Accum <= (DelayRead56  
910          >>>3)*3 + (DelayRead57>>>3)*5;  
911      16'h01C6:    Interpolation_Accum <= (DelayRead56  
912          >>>3)*2 + (DelayRead57>>>3)*6;  
913      16'h01C7:    Interpolation_Accum <= (DelayRead56  
914          >>>3)*1 + (DelayRead57>>>3)*7;  
915      16'h01C8:    Interpolation_Accum <= (DelayRead57  
916          >>>3)*8 + (DelayRead58>>>3)*0;  
917      16'h01C9:    Interpolation_Accum <= (DelayRead57  
918          >>>3)*7 + (DelayRead58>>>3)*1;  
919      16'h01CA:    Interpolation_Accum <= (DelayRead57  
920          >>>3)*6 + (DelayRead58>>>3)*2;  
921      16'h01CB:    Interpolation_Accum <= (DelayRead57  
922          >>>3)*5 + (DelayRead58>>>3)*3;  
923      16'h01CC:    Interpolation_Accum <= (DelayRead57  
924          >>>3)*4 + (DelayRead58>>>3)*4;  
925      16'h01CD:    Interpolation_Accum <= (DelayRead57  
926          >>>3)*3 + (DelayRead58>>>3)*5;  
927      16'h01CE:    Interpolation_Accum <= (DelayRead57  
928          >>>3)*2 + (DelayRead58>>>3)*6;
```

```
917      16'h01CF:    Interpolation_Accum <= (DelayRead57
               >>>3)*1 + (DelayRead58>>>3)*7;
918      16'h01D0:    Interpolation_Accum <= (DelayRead58
               >>>3)*8 + (DelayRead59>>>3)*0;
919      16'h01D1:    Interpolation_Accum <= (DelayRead58
               >>>3)*7 + (DelayRead59>>>3)*1;
920      16'h01D2:    Interpolation_Accum <= (DelayRead58
               >>>3)*6 + (DelayRead59>>>3)*2;
921      16'h01D3:    Interpolation_Accum <= (DelayRead58
               >>>3)*5 + (DelayRead59>>>3)*3;
922      16'h01D4:    Interpolation_Accum <= (DelayRead58
               >>>3)*4 + (DelayRead59>>>3)*4;
923      16'h01D5:    Interpolation_Accum <= (DelayRead58
               >>>3)*3 + (DelayRead59>>>3)*5;
924      16'h01D6:    Interpolation_Accum <= (DelayRead58
               >>>3)*2 + (DelayRead59>>>3)*6;
925      16'h01D7:    Interpolation_Accum <= (DelayRead58
               >>>3)*1 + (DelayRead59>>>3)*7;
926      16'h01D8:    Interpolation_Accum <= (DelayRead59
               >>>3)*8 + (DelayRead60>>>3)*0;
927      16'h01D9:    Interpolation_Accum <= (DelayRead59
               >>>3)*7 + (DelayRead60>>>3)*1;
928      16'h01DA:    Interpolation_Accum <= (DelayRead59
               >>>3)*6 + (DelayRead60>>>3)*2;
```

```
929      16'h01DB:     Interpolation_Accum <= (DelayRead59
               >>>3)*5 + (DelayRead60>>>3)*3;
930      16'h01DC:     Interpolation_Accum <= (DelayRead59
               >>>3)*4 + (DelayRead60>>>3)*4;
931      16'h01DD:     Interpolation_Accum <= (DelayRead59
               >>>3)*3 + (DelayRead60>>>3)*5;
932      16'h01DE:     Interpolation_Accum <= (DelayRead59
               >>>3)*2 + (DelayRead60>>>3)*6;
933      16'h01DF:     Interpolation_Accum <= (DelayRead59
               >>>3)*1 + (DelayRead60>>>3)*7;
934      16'h01E0:     Interpolation_Accum <= (DelayRead60
               >>>3)*8 + (DelayRead61>>>3)*0;
935      16'h01E1:     Interpolation_Accum <= (DelayRead60
               >>>3)*7 + (DelayRead61>>>3)*1;
936      16'h01E2:     Interpolation_Accum <= (DelayRead60
               >>>3)*6 + (DelayRead61>>>3)*2;
937      16'h01E3:     Interpolation_Accum <= (DelayRead60
               >>>3)*5 + (DelayRead61>>>3)*3;
938      16'h01E4:     Interpolation_Accum <= (DelayRead60
               >>>3)*4 + (DelayRead61>>>3)*4;
939      16'h01E5:     Interpolation_Accum <= (DelayRead60
               >>>3)*3 + (DelayRead61>>>3)*5;
940      16'h01E6:     Interpolation_Accum <= (DelayRead60
               >>>3)*2 + (DelayRead61>>>3)*6;
```

```
941      16'h01E7:    Interpolation_Accum <= (DelayRead60
942                      >>>3)*1 + (DelayRead61>>>3)*7;
943      16'h01E8:    Interpolation_Accum <= (DelayRead61
944                      >>>3)*8 + (DelayRead62>>>3)*0;
945      16'h01E9:    Interpolation_Accum <= (DelayRead61
946                      >>>3)*7 + (DelayRead62>>>3)*1;
947      16'h01EA:    Interpolation_Accum <= (DelayRead61
948                      >>>3)*6 + (DelayRead62>>>3)*2;
949      16'h01EB:    Interpolation_Accum <= (DelayRead61
950                      >>>3)*5 + (DelayRead62>>>3)*3;
951      16'h01EC:    Interpolation_Accum <= (DelayRead61
952                      >>>3)*4 + (DelayRead62>>>3)*4;
953      16'h01ED:    Interpolation_Accum <= (DelayRead61
954                      >>>3)*3 + (DelayRead62>>>3)*5;
955      16'h01EE:    Interpolation_Accum <= (DelayRead61
956                      >>>3)*2 + (DelayRead62>>>3)*6;
957      16'h01EF:    Interpolation_Accum <= (DelayRead61
958                      >>>3)*1 + (DelayRead62>>>3)*7;
959      16'h01F0:    Interpolation_Accum <= (DelayRead62
960                      >>>3)*8 + (DelayRead63>>>3)*0;
961      16'h01F1:    Interpolation_Accum <= (DelayRead62
962                      >>>3)*7 + (DelayRead63>>>3)*1;
963      16'h01F2:    Interpolation_Accum <= (DelayRead62
964                      >>>3)*6 + (DelayRead63>>>3)*2;
```

```
953      16'h01F3 :     Interpolation_Accum <= (DelayRead62
               >>>3)*5 + (DelayRead63>>>3)*3;
954      16'h01F4 :     Interpolation_Accum <= (DelayRead62
               >>>3)*4 + (DelayRead63>>>3)*4;
955      16'h01F5 :     Interpolation_Accum <= (DelayRead62
               >>>3)*3 + (DelayRead63>>>3)*5;
956      16'h01F6 :     Interpolation_Accum <= (DelayRead62
               >>>3)*2 + (DelayRead63>>>3)*6;
957      16'h01F7 :     Interpolation_Accum <= (DelayRead62
               >>>3)*1 + (DelayRead63>>>3)*7;
958      16'h01F8 :     Interpolation_Accum <= (DelayRead63
               >>>3)*8 + (DelayRead64>>>3)*0;
959      16'h01F9 :     Interpolation_Accum <= (DelayRead63
               >>>3)*7 + (DelayRead64>>>3)*1;
960      16'h01FA :     Interpolation_Accum <= (DelayRead63
               >>>3)*6 + (DelayRead64>>>3)*2;
961      16'h01FB :     Interpolation_Accum <= (DelayRead63
               >>>3)*5 + (DelayRead64>>>3)*3;
962      16'h01FC :     Interpolation_Accum <= (DelayRead63
               >>>3)*4 + (DelayRead64>>>3)*4;
963      16'h01FD :     Interpolation_Accum <= (DelayRead63
               >>>3)*3 + (DelayRead64>>>3)*5;
964      16'h01FE :     Interpolation_Accum <= (DelayRead63
               >>>3)*2 + (DelayRead64>>>3)*6;
```

```

965      16'h01FF:    Interpolation_Accum <= (DelayRead63
              >>>3)*1 + (DelayRead64>>>3)*7;
966      16'h0200:    Interpolation_Accum <= (DelayRead64
              >>>3)*8 + (DelayRead65>>>3)*0;
967      16'h0201:    Interpolation_Accum <= (DelayRead64
              >>>3)*7 + (DelayRead65>>>3)*1;
968      16'h0202:    Interpolation_Accum <= (DelayRead64
              >>>3)*6 + (DelayRead65>>>3)*2;
969      16'h0203:    Interpolation_Accum <= (DelayRead64
              >>>3)*5 + (DelayRead65>>>3)*3;
970      16'h0204:    Interpolation_Accum <= (DelayRead64
              >>>3)*4 + (DelayRead65>>>3)*4;
971      16'h0205:    Interpolation_Accum <= (DelayRead64
              >>>3)*3 + (DelayRead65>>>3)*5;
972      16'h0206:    Interpolation_Accum <= (DelayRead64
              >>>3)*2 + (DelayRead65>>>3)*6;
973      16'h0207:    Interpolation_Accum <= (DelayRead64
              >>>3)*1 + (DelayRead65>>>3)*7;
974      16'h0208:    Interpolation_Accum <= (DelayRead65
              >>>3)*8 + (DelayRead66>>>3)*0;
975      16'h0209:    Interpolation_Accum <= (DelayRead65
              >>>3)*7 + (DelayRead66>>>3)*1;
976      16'h020A:    Interpolation_Accum <= (DelayRead65
              >>>3)*6 + (DelayRead66>>>3)*2;

```

```
977      16'h020B :     Interpolation_Accum <= (DelayRead65
               >>>3)*5 + (DelayRead66>>>3)*3;
978      16'h020C :     Interpolation_Accum <= (DelayRead65
               >>>3)*4 + (DelayRead66>>>3)*4;
979      16'h020D :     Interpolation_Accum <= (DelayRead65
               >>>3)*3 + (DelayRead66>>>3)*5;
980      16'h020E :     Interpolation_Accum <= (DelayRead65
               >>>3)*2 + (DelayRead66>>>3)*6;
981      16'h020F :     Interpolation_Accum <= (DelayRead65
               >>>3)*1 + (DelayRead66>>>3)*7;
982      16'h0210 :     Interpolation_Accum <= (DelayRead66
               >>>3)*8 + (DelayRead67>>>3)*0;
983      16'h0211 :     Interpolation_Accum <= (DelayRead66
               >>>3)*7 + (DelayRead67>>>3)*1;
984      16'h0212 :     Interpolation_Accum <= (DelayRead66
               >>>3)*6 + (DelayRead67>>>3)*2;
985      16'h0213 :     Interpolation_Accum <= (DelayRead66
               >>>3)*5 + (DelayRead67>>>3)*3;
986      16'h0214 :     Interpolation_Accum <= (DelayRead66
               >>>3)*4 + (DelayRead67>>>3)*4;
987      16'h0215 :     Interpolation_Accum <= (DelayRead66
               >>>3)*3 + (DelayRead67>>>3)*5;
988      16'h0216 :     Interpolation_Accum <= (DelayRead66
               >>>3)*2 + (DelayRead67>>>3)*6;
```

```
989      16'h0217:    Interpolation_Accum <= (DelayRead66
               >>>3)*1 + (DelayRead67>>>3)*7;
990      16'h0218:    Interpolation_Accum <= (DelayRead67
               >>>3)*8 + (DelayRead68>>>3)*0;
991      16'h0219:    Interpolation_Accum <= (DelayRead67
               >>>3)*7 + (DelayRead68>>>3)*1;
992      16'h021A:    Interpolation_Accum <= (DelayRead67
               >>>3)*6 + (DelayRead68>>>3)*2;
993      16'h021B:    Interpolation_Accum <= (DelayRead67
               >>>3)*5 + (DelayRead68>>>3)*3;
994      16'h021C:    Interpolation_Accum <= (DelayRead67
               >>>3)*4 + (DelayRead68>>>3)*4;
995      16'h021D:    Interpolation_Accum <= (DelayRead67
               >>>3)*3 + (DelayRead68>>>3)*5;
996      16'h021E:    Interpolation_Accum <= (DelayRead67
               >>>3)*2 + (DelayRead68>>>3)*6;
997      16'h021F:    Interpolation_Accum <= (DelayRead67
               >>>3)*1 + (DelayRead68>>>3)*7;
998      16'h0220:    Interpolation_Accum <= (DelayRead68
               >>>3)*8 + (DelayRead69>>>3)*0;
999      16'h0221:    Interpolation_Accum <= (DelayRead68
               >>>3)*7 + (DelayRead69>>>3)*1;
1000     16'h0222:   Interpolation_Accum <= (DelayRead68
               >>>3)*6 + (DelayRead69>>>3)*2;
```

```
1001      16'h0223 :     Interpolation_Accum <= (DelayRead68
               >>>3)*5 + (DelayRead69>>>3)*3;
1002      16'h0224 :     Interpolation_Accum <= (DelayRead68
               >>>3)*4 + (DelayRead69>>>3)*4;
1003      16'h0225 :     Interpolation_Accum <= (DelayRead68
               >>>3)*3 + (DelayRead69>>>3)*5;
1004      16'h0226 :     Interpolation_Accum <= (DelayRead68
               >>>3)*2 + (DelayRead69>>>3)*6;
1005      16'h0227 :     Interpolation_Accum <= (DelayRead68
               >>>3)*1 + (DelayRead69>>>3)*7;
1006      16'h0228 :     Interpolation_Accum <= (DelayRead69
               >>>3)*8 + (DelayRead70>>>3)*0;
1007      16'h0229 :     Interpolation_Accum <= (DelayRead69
               >>>3)*7 + (DelayRead70>>>3)*1;
1008      16'h022A :     Interpolation_Accum <= (DelayRead69
               >>>3)*6 + (DelayRead70>>>3)*2;
1009      16'h022B :     Interpolation_Accum <= (DelayRead69
               >>>3)*5 + (DelayRead70>>>3)*3;
1010      16'h022C :     Interpolation_Accum <= (DelayRead69
               >>>3)*4 + (DelayRead70>>>3)*4;
1011      16'h022D :     Interpolation_Accum <= (DelayRead69
               >>>3)*3 + (DelayRead70>>>3)*5;
1012      16'h022E :     Interpolation_Accum <= (DelayRead69
               >>>3)*2 + (DelayRead70>>>3)*6;
```

```
1013      16'h022F:    Interpolation_Accum <= (DelayRead69
               >>>3)*1 + (DelayRead70>>>3)*7;
1014      16'h0230:    Interpolation_Accum <= (DelayRead70
               >>>3)*8 + (DelayRead71>>>3)*0;
1015      16'h0231:    Interpolation_Accum <= (DelayRead70
               >>>3)*7 + (DelayRead71>>>3)*1;
1016      16'h0232:    Interpolation_Accum <= (DelayRead70
               >>>3)*6 + (DelayRead71>>>3)*2;
1017      16'h0233:    Interpolation_Accum <= (DelayRead70
               >>>3)*5 + (DelayRead71>>>3)*3;
1018      16'h0234:    Interpolation_Accum <= (DelayRead70
               >>>3)*4 + (DelayRead71>>>3)*4;
1019      16'h0235:    Interpolation_Accum <= (DelayRead70
               >>>3)*3 + (DelayRead71>>>3)*5;
1020      16'h0236:    Interpolation_Accum <= (DelayRead70
               >>>3)*2 + (DelayRead71>>>3)*6;
1021      16'h0237:    Interpolation_Accum <= (DelayRead70
               >>>3)*1 + (DelayRead71>>>3)*7;
1022      16'h0238:    Interpolation_Accum <= (DelayRead71
               >>>3)*8 + (DelayRead72>>>3)*0;
1023      16'h0239:    Interpolation_Accum <= (DelayRead71
               >>>3)*7 + (DelayRead72>>>3)*1;
1024      16'h023A:    Interpolation_Accum <= (DelayRead71
               >>>3)*6 + (DelayRead72>>>3)*2;
```

```
1025      16'h023B :     Interpolation_Accum <= (DelayRead71  
           >>>3)*5 + (DelayRead72>>>3)*3;  
1026      16'h023C :     Interpolation_Accum <= (DelayRead71  
           >>>3)*4 + (DelayRead72>>>3)*4;  
1027      16'h023D :     Interpolation_Accum <= (DelayRead71  
           >>>3)*3 + (DelayRead72>>>3)*5;  
1028      16'h023E :     Interpolation_Accum <= (DelayRead71  
           >>>3)*2 + (DelayRead72>>>3)*6;  
1029      16'h023F :     Interpolation_Accum <= (DelayRead71  
           >>>3)*1 + (DelayRead72>>>3)*7;  
1030      16'h0240 :     Interpolation_Accum <= (DelayRead72  
           >>>3)*8 + (DelayRead73>>>3)*0;  
1031      16'h0241 :     Interpolation_Accum <= (DelayRead72  
           >>>3)*7 + (DelayRead73>>>3)*1;  
1032      16'h0242 :     Interpolation_Accum <= (DelayRead72  
           >>>3)*6 + (DelayRead73>>>3)*2;  
1033      16'h0243 :     Interpolation_Accum <= (DelayRead72  
           >>>3)*5 + (DelayRead73>>>3)*3;  
1034      16'h0244 :     Interpolation_Accum <= (DelayRead72  
           >>>3)*4 + (DelayRead73>>>3)*4;  
1035      16'h0245 :     Interpolation_Accum <= (DelayRead72  
           >>>3)*3 + (DelayRead73>>>3)*5;  
1036      16'h0246 :     Interpolation_Accum <= (DelayRead72  
           >>>3)*2 + (DelayRead73>>>3)*6;
```

```

1037      16'h0247:    Interpolation_Accum <= (DelayRead72
                  >>>3)*1 + (DelayRead73>>>3)*7;
1038      16'h0248:    Interpolation_Accum <= (DelayRead73
                  >>>3)*8 + (DelayRead74>>>3)*0;
1039      16'h0249:    Interpolation_Accum <= (DelayRead73
                  >>>3)*7 + (DelayRead74>>>3)*1;
1040      16'h024A:    Interpolation_Accum <= (DelayRead73
                  >>>3)*6 + (DelayRead74>>>3)*2;
1041      16'h024B:    Interpolation_Accum <= (DelayRead73
                  >>>3)*5 + (DelayRead74>>>3)*3;
1042      16'h024C:    Interpolation_Accum <= (DelayRead73
                  >>>3)*4 + (DelayRead74>>>3)*4;
1043      16'h024D:    Interpolation_Accum <= (DelayRead73
                  >>>3)*3 + (DelayRead74>>>3)*5;
1044      16'h024E:    Interpolation_Accum <= (DelayRead73
                  >>>3)*2 + (DelayRead74>>>3)*6;
1045      16'h024F:    Interpolation_Accum <= (DelayRead73
                  >>>3)*1 + (DelayRead74>>>3)*7;
1046      16'h0250:    Interpolation_Accum <= (DelayRead74
                  >>>3)*8 + (DelayRead75>>>3)*0;
1047      16'h0251:    Interpolation_Accum <= (DelayRead74
                  >>>3)*7 + (DelayRead75>>>3)*1;
1048      16'h0252:    Interpolation_Accum <= (DelayRead74
                  >>>3)*6 + (DelayRead75>>>3)*2;

```

```
1049      16'h0253:    Interpolation_Accum <= (DelayRead74  
           >>>3)*5 + (DelayRead75>>>3)*3;  
1050      16'h0254:    Interpolation_Accum <= (DelayRead74  
           >>>3)*4 + (DelayRead75>>>3)*4;  
1051      16'h0255:    Interpolation_Accum <= (DelayRead74  
           >>>3)*3 + (DelayRead75>>>3)*5;  
1052      16'h0256:    Interpolation_Accum <= (DelayRead74  
           >>>3)*2 + (DelayRead75>>>3)*6;  
1053      16'h0257:    Interpolation_Accum <= (DelayRead74  
           >>>3)*1 + (DelayRead75>>>3)*7;  
1054      16'h0258:    Interpolation_Accum <= (DelayRead75  
           >>>3)*8 + (DelayRead76>>>3)*0;  
1055      16'h0259:    Interpolation_Accum <= (DelayRead75  
           >>>3)*7 + (DelayRead76>>>3)*1;  
1056      16'h025A:    Interpolation_Accum <= (DelayRead75  
           >>>3)*6 + (DelayRead76>>>3)*2;  
1057      16'h025B:    Interpolation_Accum <= (DelayRead75  
           >>>3)*5 + (DelayRead76>>>3)*3;  
1058      16'h025C:    Interpolation_Accum <= (DelayRead75  
           >>>3)*4 + (DelayRead76>>>3)*4;  
1059      16'h025D:    Interpolation_Accum <= (DelayRead75  
           >>>3)*3 + (DelayRead76>>>3)*5;  
1060      16'h025E:    Interpolation_Accum <= (DelayRead75  
           >>>3)*2 + (DelayRead76>>>3)*6;
```

```
1061      16'h025F:    Interpolation_Accum <= (DelayRead75
                  >>>3)*1 + (DelayRead76>>>3)*7;
1062      16'h0260:    Interpolation_Accum <= (DelayRead76
                  >>>3)*8 + (DelayRead77>>>3)*0;
1063      16'h0261:    Interpolation_Accum <= (DelayRead76
                  >>>3)*7 + (DelayRead77>>>3)*1;
1064      16'h0262:    Interpolation_Accum <= (DelayRead76
                  >>>3)*6 + (DelayRead77>>>3)*2;
1065      16'h0263:    Interpolation_Accum <= (DelayRead76
                  >>>3)*5 + (DelayRead77>>>3)*3;
1066      16'h0264:    Interpolation_Accum <= (DelayRead76
                  >>>3)*4 + (DelayRead77>>>3)*4;
1067      16'h0265:    Interpolation_Accum <= (DelayRead76
                  >>>3)*3 + (DelayRead77>>>3)*5;
1068      16'h0266:    Interpolation_Accum <= (DelayRead76
                  >>>3)*2 + (DelayRead77>>>3)*6;
1069      16'h0267:    Interpolation_Accum <= (DelayRead76
                  >>>3)*1 + (DelayRead77>>>3)*7;
1070      16'h0268:    Interpolation_Accum <= (DelayRead77
                  >>>3)*8 + (DelayRead78>>>3)*0;
1071      16'h0269:    Interpolation_Accum <= (DelayRead77
                  >>>3)*7 + (DelayRead78>>>3)*1;
1072      16'h026A:    Interpolation_Accum <= (DelayRead77
                  >>>3)*6 + (DelayRead78>>>3)*2;
```

```
1073      16'h026B :     Interpolation_Accum <= (DelayRead77
               >>>3)*5 + (DelayRead78>>>3)*3;
1074      16'h026C :     Interpolation_Accum <= (DelayRead77
               >>>3)*4 + (DelayRead78>>>3)*4;
1075      16'h026D :     Interpolation_Accum <= (DelayRead77
               >>>3)*3 + (DelayRead78>>>3)*5;
1076      16'h026E :     Interpolation_Accum <= (DelayRead77
               >>>3)*2 + (DelayRead78>>>3)*6;
1077      16'h026F :     Interpolation_Accum <= (DelayRead77
               >>>3)*1 + (DelayRead78>>>3)*7;
1078      16'h0270 :     Interpolation_Accum <= (DelayRead78
               >>>3)*8 + (DelayRead79>>>3)*0;
1079      16'h0271 :     Interpolation_Accum <= (DelayRead78
               >>>3)*7 + (DelayRead79>>>3)*1;
1080      16'h0272 :     Interpolation_Accum <= (DelayRead78
               >>>3)*6 + (DelayRead79>>>3)*2;
1081      16'h0273 :     Interpolation_Accum <= (DelayRead78
               >>>3)*5 + (DelayRead79>>>3)*3;
1082      16'h0274 :     Interpolation_Accum <= (DelayRead78
               >>>3)*4 + (DelayRead79>>>3)*4;
1083      16'h0275 :     Interpolation_Accum <= (DelayRead78
               >>>3)*3 + (DelayRead79>>>3)*5;
1084      16'h0276 :     Interpolation_Accum <= (DelayRead78
               >>>3)*2 + (DelayRead79>>>3)*6;
```

```
1085      16'h0277:    Interpolation_Accum <= (DelayRead78
               >>>3)*1 + (DelayRead79>>>3)*7;
1086      16'h0278:    Interpolation_Accum <= (DelayRead79
               >>>3)*8 + (DelayRead80>>>3)*0;
1087      16'h0279:    Interpolation_Accum <= (DelayRead79
               >>>3)*7 + (DelayRead80>>>3)*1;
1088      16'h027A:    Interpolation_Accum <= (DelayRead79
               >>>3)*6 + (DelayRead80>>>3)*2;
1089      16'h027B:    Interpolation_Accum <= (DelayRead79
               >>>3)*5 + (DelayRead80>>>3)*3;
1090      16'h027C:    Interpolation_Accum <= (DelayRead79
               >>>3)*4 + (DelayRead80>>>3)*4;
1091      16'h027D:    Interpolation_Accum <= (DelayRead79
               >>>3)*3 + (DelayRead80>>>3)*5;
1092      16'h027E:    Interpolation_Accum <= (DelayRead79
               >>>3)*2 + (DelayRead80>>>3)*6;
1093      16'h027F:    Interpolation_Accum <= (DelayRead79
               >>>3)*1 + (DelayRead80>>>3)*7;
1094      16'h0280:    Interpolation_Accum <= (DelayRead80
               >>>3)*8 + (DelayRead81>>>3)*0;
1095      16'h0281:    Interpolation_Accum <= (DelayRead80
               >>>3)*7 + (DelayRead81>>>3)*1;
1096      16'h0282:    Interpolation_Accum <= (DelayRead80
               >>>3)*6 + (DelayRead81>>>3)*2;
```

```
1097      16'h0283:    Interpolation_Accum <= (DelayRead80
               >>>3)*5 + (DelayRead81>>>3)*3;
1098      16'h0284:    Interpolation_Accum <= (DelayRead80
               >>>3)*4 + (DelayRead81>>>3)*4;
1099      16'h0285:    Interpolation_Accum <= (DelayRead80
               >>>3)*3 + (DelayRead81>>>3)*5;
1100      16'h0286:    Interpolation_Accum <= (DelayRead80
               >>>3)*2 + (DelayRead81>>>3)*6;
1101      16'h0287:    Interpolation_Accum <= (DelayRead80
               >>>3)*1 + (DelayRead81>>>3)*7;
1102      16'h0288:    Interpolation_Accum <= (DelayRead81
               >>>3)*8 + (DelayRead82>>>3)*0;
1103      16'h0289:    Interpolation_Accum <= (DelayRead81
               >>>3)*7 + (DelayRead82>>>3)*1;
1104      16'h028A:    Interpolation_Accum <= (DelayRead81
               >>>3)*6 + (DelayRead82>>>3)*2;
1105      16'h028B:    Interpolation_Accum <= (DelayRead81
               >>>3)*5 + (DelayRead82>>>3)*3;
1106      16'h028C:    Interpolation_Accum <= (DelayRead81
               >>>3)*4 + (DelayRead82>>>3)*4;
1107      16'h028D:    Interpolation_Accum <= (DelayRead81
               >>>3)*3 + (DelayRead82>>>3)*5;
1108      16'h028E:    Interpolation_Accum <= (DelayRead81
               >>>3)*2 + (DelayRead82>>>3)*6;
```

```
1109      16'h028F:    Interpolation_Accum <= (DelayRead81  
1110          >>>3)*1 + (DelayRead82>>>3)*7;  
1111      16'h0290:    Interpolation_Accum <= (DelayRead82  
1112          >>>3)*8 + (DelayRead83>>>3)*0;  
1113      16'h0291:    Interpolation_Accum <= (DelayRead82  
1114          >>>3)*7 + (DelayRead83>>>3)*1;  
1115      16'h0292:    Interpolation_Accum <= (DelayRead82  
1116          >>>3)*6 + (DelayRead83>>>3)*2;  
1117      16'h0293:    Interpolation_Accum <= (DelayRead82  
1118          >>>3)*5 + (DelayRead83>>>3)*3;  
1119      16'h0294:    Interpolation_Accum <= (DelayRead82  
1120          >>>3)*4 + (DelayRead83>>>3)*4;  
1121      16'h0295:    Interpolation_Accum <= (DelayRead82  
1122          >>>3)*3 + (DelayRead83>>>3)*5;  
1123      16'h0296:    Interpolation_Accum <= (DelayRead82  
1124          >>>3)*2 + (DelayRead83>>>3)*6;  
1125      16'h0297:    Interpolation_Accum <= (DelayRead82  
1126          >>>3)*1 + (DelayRead83>>>3)*7;  
1127      16'h0298:    Interpolation_Accum <= (DelayRead83  
1128          >>>3)*8 + (DelayRead84>>>3)*0;  
1129      16'h0299:    Interpolation_Accum <= (DelayRead83  
1130          >>>3)*7 + (DelayRead84>>>3)*1;  
1131      16'h029A:    Interpolation_Accum <= (DelayRead83  
1132          >>>3)*6 + (DelayRead84>>>3)*2;
```

```
1121      16'h029B :     Interpolation_Accum <= (DelayRead83
               >>>3)*5 + (DelayRead84>>>3)*3;
1122      16'h029C :     Interpolation_Accum <= (DelayRead83
               >>>3)*4 + (DelayRead84>>>3)*4;
1123      16'h029D :     Interpolation_Accum <= (DelayRead83
               >>>3)*3 + (DelayRead84>>>3)*5;
1124      16'h029E :     Interpolation_Accum <= (DelayRead83
               >>>3)*2 + (DelayRead84>>>3)*6;
1125      16'h029F :     Interpolation_Accum <= (DelayRead83
               >>>3)*1 + (DelayRead84>>>3)*7;
1126      16'h02A0 :     Interpolation_Accum <= (DelayRead84
               >>>3)*8 + (DelayRead85>>>3)*0;
1127      16'h02A1 :     Interpolation_Accum <= (DelayRead84
               >>>3)*7 + (DelayRead85>>>3)*1;
1128      16'h02A2 :     Interpolation_Accum <= (DelayRead84
               >>>3)*6 + (DelayRead85>>>3)*2;
1129      16'h02A3 :     Interpolation_Accum <= (DelayRead84
               >>>3)*5 + (DelayRead85>>>3)*3;
1130      16'h02A4 :     Interpolation_Accum <= (DelayRead84
               >>>3)*4 + (DelayRead85>>>3)*4;
1131      16'h02A5 :     Interpolation_Accum <= (DelayRead84
               >>>3)*3 + (DelayRead85>>>3)*5;
1132      16'h02A6 :     Interpolation_Accum <= (DelayRead84
               >>>3)*2 + (DelayRead85>>>3)*6;
```

```
1133      16'h02A7:    Interpolation_Accum <= (DelayRead84
               >>>3)*1 + (DelayRead85>>>3)*7;
1134      16'h02A8:    Interpolation_Accum <= (DelayRead85
               >>>3)*8 + (DelayRead86>>>3)*0;
1135      16'h02A9:    Interpolation_Accum <= (DelayRead85
               >>>3)*7 + (DelayRead86>>>3)*1;
1136      16'h02AA:    Interpolation_Accum <= (DelayRead85
               >>>3)*6 + (DelayRead86>>>3)*2;
1137      16'h02AB:    Interpolation_Accum <= (DelayRead85
               >>>3)*5 + (DelayRead86>>>3)*3;
1138      16'h02AC:    Interpolation_Accum <= (DelayRead85
               >>>3)*4 + (DelayRead86>>>3)*4;
1139      16'h02AD:    Interpolation_Accum <= (DelayRead85
               >>>3)*3 + (DelayRead86>>>3)*5;
1140      16'h02AE:    Interpolation_Accum <= (DelayRead85
               >>>3)*2 + (DelayRead86>>>3)*6;
1141      16'h02AF:    Interpolation_Accum <= (DelayRead85
               >>>3)*1 + (DelayRead86>>>3)*7;
1142      16'h02B0:    Interpolation_Accum <= (DelayRead86
               >>>3)*8 + (DelayRead87>>>3)*0;
1143      16'h02B1:    Interpolation_Accum <= (DelayRead86
               >>>3)*7 + (DelayRead87>>>3)*1;
1144      16'h02B2:    Interpolation_Accum <= (DelayRead86
               >>>3)*6 + (DelayRead87>>>3)*2;
```

```
1145      16'h02B3:    Interpolation_Accum <= (DelayRead86
               >>>3)*5 + (DelayRead87>>>3)*3;
1146      16'h02B4:    Interpolation_Accum <= (DelayRead86
               >>>3)*4 + (DelayRead87>>>3)*4;
1147      16'h02B5:    Interpolation_Accum <= (DelayRead86
               >>>3)*3 + (DelayRead87>>>3)*5;
1148      16'h02B6:    Interpolation_Accum <= (DelayRead86
               >>>3)*2 + (DelayRead87>>>3)*6;
1149      16'h02B7:    Interpolation_Accum <= (DelayRead86
               >>>3)*1 + (DelayRead87>>>3)*7;
1150      16'h02B8:    Interpolation_Accum <= (DelayRead87
               >>>3)*8 + (DelayRead88>>>3)*0;
1151      16'h02B9:    Interpolation_Accum <= (DelayRead87
               >>>3)*7 + (DelayRead88>>>3)*1;
1152      16'h02BA:    Interpolation_Accum <= (DelayRead87
               >>>3)*6 + (DelayRead88>>>3)*2;
1153      16'h02BB:    Interpolation_Accum <= (DelayRead87
               >>>3)*5 + (DelayRead88>>>3)*3;
1154      16'h02BC:    Interpolation_Accum <= (DelayRead87
               >>>3)*4 + (DelayRead88>>>3)*4;
1155      16'h02BD:    Interpolation_Accum <= (DelayRead87
               >>>3)*3 + (DelayRead88>>>3)*5;
1156      16'h02BE:    Interpolation_Accum <= (DelayRead87
               >>>3)*2 + (DelayRead88>>>3)*6;
```

```
1157      16'h02BF:    Interpolation_Accum <= (DelayRead87
               >>>3)*1 + (DelayRead88>>>3)*7;
1158      16'h02C0:    Interpolation_Accum <= (DelayRead88
               >>>3)*8 + (DelayRead89>>>3)*0;
1159      16'h02C1:    Interpolation_Accum <= (DelayRead88
               >>>3)*7 + (DelayRead89>>>3)*1;
1160      16'h02C2:    Interpolation_Accum <= (DelayRead88
               >>>3)*6 + (DelayRead89>>>3)*2;
1161      16'h02C3:    Interpolation_Accum <= (DelayRead88
               >>>3)*5 + (DelayRead89>>>3)*3;
1162      16'h02C4:    Interpolation_Accum <= (DelayRead88
               >>>3)*4 + (DelayRead89>>>3)*4;
1163      16'h02C5:    Interpolation_Accum <= (DelayRead88
               >>>3)*3 + (DelayRead89>>>3)*5;
1164      16'h02C6:    Interpolation_Accum <= (DelayRead88
               >>>3)*2 + (DelayRead89>>>3)*6;
1165      16'h02C7:    Interpolation_Accum <= (DelayRead88
               >>>3)*1 + (DelayRead89>>>3)*7;
1166      16'h02C8:    Interpolation_Accum <= (DelayRead89
               >>>3)*8 + (DelayRead90>>>3)*0;
1167      16'h02C9:    Interpolation_Accum <= (DelayRead89
               >>>3)*7 + (DelayRead90>>>3)*1;
1168      16'h02CA:    Interpolation_Accum <= (DelayRead89
               >>>3)*6 + (DelayRead90>>>3)*2;
```

```
1169      16'h02CB:     Interpolation_Accum <= (DelayRead89
               >>>3)*5 + (DelayRead90>>>3)*3;
1170      16'h02CC:     Interpolation_Accum <= (DelayRead89
               >>>3)*4 + (DelayRead90>>>3)*4;
1171      16'h02CD:     Interpolation_Accum <= (DelayRead89
               >>>3)*3 + (DelayRead90>>>3)*5;
1172      16'h02CE:     Interpolation_Accum <= (DelayRead89
               >>>3)*2 + (DelayRead90>>>3)*6;
1173      16'h02CF:     Interpolation_Accum <= (DelayRead89
               >>>3)*1 + (DelayRead90>>>3)*7;
1174      16'h02D0:     Interpolation_Accum <= (DelayRead90
               >>>3)*8 + (DelayRead91>>>3)*0;
1175      16'h02D1:     Interpolation_Accum <= (DelayRead90
               >>>3)*7 + (DelayRead91>>>3)*1;
1176      16'h02D2:     Interpolation_Accum <= (DelayRead90
               >>>3)*6 + (DelayRead91>>>3)*2;
1177      16'h02D3:     Interpolation_Accum <= (DelayRead90
               >>>3)*5 + (DelayRead91>>>3)*3;
1178      16'h02D4:     Interpolation_Accum <= (DelayRead90
               >>>3)*4 + (DelayRead91>>>3)*4;
1179      16'h02D5:     Interpolation_Accum <= (DelayRead90
               >>>3)*3 + (DelayRead91>>>3)*5;
1180      16'h02D6:     Interpolation_Accum <= (DelayRead90
               >>>3)*2 + (DelayRead91>>>3)*6;
```

```
1181      16'h02D7:    Interpolation_Accum <= (DelayRead90
                  >>>3)*1 + (DelayRead91>>>3)*7;
1182      16'h02D8:    Interpolation_Accum <= (DelayRead91
                  >>>3)*8 + (DelayRead92>>>3)*0;
1183      16'h02D9:    Interpolation_Accum <= (DelayRead91
                  >>>3)*7 + (DelayRead92>>>3)*1;
1184      16'h02DA:    Interpolation_Accum <= (DelayRead91
                  >>>3)*6 + (DelayRead92>>>3)*2;
1185      16'h02DB:    Interpolation_Accum <= (DelayRead91
                  >>>3)*5 + (DelayRead92>>>3)*3;
1186      16'h02DC:    Interpolation_Accum <= (DelayRead91
                  >>>3)*4 + (DelayRead92>>>3)*4;
1187      16'h02DD:    Interpolation_Accum <= (DelayRead91
                  >>>3)*3 + (DelayRead92>>>3)*5;
1188      16'h02DE:    Interpolation_Accum <= (DelayRead91
                  >>>3)*2 + (DelayRead92>>>3)*6;
1189      16'h02DF:    Interpolation_Accum <= (DelayRead91
                  >>>3)*1 + (DelayRead92>>>3)*7;
1190      16'h02E0:    Interpolation_Accum <= (DelayRead92
                  >>>3)*8 + (DelayRead93>>>3)*0;
1191      16'h02E1:    Interpolation_Accum <= (DelayRead92
                  >>>3)*7 + (DelayRead93>>>3)*1;
1192      16'h02E2:    Interpolation_Accum <= (DelayRead92
                  >>>3)*6 + (DelayRead93>>>3)*2;
```

```
1193      16'h02E3 :     Interpolation_Accum <= (DelayRead92
               >>>3)*5 + (DelayRead93>>>3)*3;
1194      16'h02E4 :     Interpolation_Accum <= (DelayRead92
               >>>3)*4 + (DelayRead93>>>3)*4;
1195      16'h02E5 :     Interpolation_Accum <= (DelayRead92
               >>>3)*3 + (DelayRead93>>>3)*5;
1196      16'h02E6 :     Interpolation_Accum <= (DelayRead92
               >>>3)*2 + (DelayRead93>>>3)*6;
1197      16'h02E7 :     Interpolation_Accum <= (DelayRead92
               >>>3)*1 + (DelayRead93>>>3)*7;
1198      16'h02E8 :     Interpolation_Accum <= (DelayRead93
               >>>3)*8 + (DelayRead94>>>3)*0;
1199      16'h02E9 :     Interpolation_Accum <= (DelayRead93
               >>>3)*7 + (DelayRead94>>>3)*1;
1200      16'h02EA :     Interpolation_Accum <= (DelayRead93
               >>>3)*6 + (DelayRead94>>>3)*2;
1201      16'h02EB :     Interpolation_Accum <= (DelayRead93
               >>>3)*5 + (DelayRead94>>>3)*3;
1202      16'h02EC :     Interpolation_Accum <= (DelayRead93
               >>>3)*4 + (DelayRead94>>>3)*4;
1203      16'h02ED :     Interpolation_Accum <= (DelayRead93
               >>>3)*3 + (DelayRead94>>>3)*5;
1204      16'h02EE :     Interpolation_Accum <= (DelayRead93
               >>>3)*2 + (DelayRead94>>>3)*6;
```

```

1205      16'h02EF:    Interpolation_Accum <= (DelayRead93
              >>>3)*1 + (DelayRead94>>>3)*7;
1206      16'h02F0:    Interpolation_Accum <= (DelayRead94
              >>>3)*8 + (DelayRead95>>>3)*0;
1207      16'h02F1:    Interpolation_Accum <= (DelayRead94
              >>>3)*7 + (DelayRead95>>>3)*1;
1208      16'h02F2:    Interpolation_Accum <= (DelayRead94
              >>>3)*6 + (DelayRead95>>>3)*2;
1209      16'h02F3:    Interpolation_Accum <= (DelayRead94
              >>>3)*5 + (DelayRead95>>>3)*3;
1210      16'h02F4:    Interpolation_Accum <= (DelayRead94
              >>>3)*4 + (DelayRead95>>>3)*4;
1211      16'h02F5:    Interpolation_Accum <= (DelayRead94
              >>>3)*3 + (DelayRead95>>>3)*5;
1212      16'h02F6:    Interpolation_Accum <= (DelayRead94
              >>>3)*2 + (DelayRead95>>>3)*6;
1213      16'h02F7:    Interpolation_Accum <= (DelayRead94
              >>>3)*1 + (DelayRead95>>>3)*7;
1214      16'h02F8:    Interpolation_Accum <= (DelayRead95
              >>>3)*8 + (DelayRead96>>>3)*0;
1215      16'h02F9:    Interpolation_Accum <= (DelayRead95
              >>>3)*7 + (DelayRead96>>>3)*1;
1216      16'h02FA:    Interpolation_Accum <= (DelayRead95
              >>>3)*6 + (DelayRead96>>>3)*2;

```

```
1217      16'h02FB :     Interpolation_Accum <= (DelayRead95
               >>>3)*5 + (DelayRead96>>>3)*3;
1218      16'h02FC :     Interpolation_Accum <= (DelayRead95
               >>>3)*4 + (DelayRead96>>>3)*4;
1219      16'h02FD :     Interpolation_Accum <= (DelayRead95
               >>>3)*3 + (DelayRead96>>>3)*5;
1220      16'h02FE :     Interpolation_Accum <= (DelayRead95
               >>>3)*2 + (DelayRead96>>>3)*6;
1221      16'h02FF :     Interpolation_Accum <= (DelayRead95
               >>>3)*1 + (DelayRead96>>>3)*7;
1222      16'h0300 :     Interpolation_Accum <= (DelayRead96
               >>>3)*8 + (DelayRead97>>>3)*0;
1223      16'h0301 :     Interpolation_Accum <= (DelayRead96
               >>>3)*7 + (DelayRead97>>>3)*1;
1224      16'h0302 :     Interpolation_Accum <= (DelayRead96
               >>>3)*6 + (DelayRead97>>>3)*2;
1225      16'h0303 :     Interpolation_Accum <= (DelayRead96
               >>>3)*5 + (DelayRead97>>>3)*3;
1226      16'h0304 :     Interpolation_Accum <= (DelayRead96
               >>>3)*4 + (DelayRead97>>>3)*4;
1227      16'h0305 :     Interpolation_Accum <= (DelayRead96
               >>>3)*3 + (DelayRead97>>>3)*5;
1228      16'h0306 :     Interpolation_Accum <= (DelayRead96
               >>>3)*2 + (DelayRead97>>>3)*6;
```

```
1229      16'h0307:    Interpolation_Accum <= (DelayRead96
               >>>3)*1 + (DelayRead97>>>3)*7;
1230      16'h0308:    Interpolation_Accum <= (DelayRead97
               >>>3)*8 + (DelayRead98>>>3)*0;
1231      16'h0309:    Interpolation_Accum <= (DelayRead97
               >>>3)*7 + (DelayRead98>>>3)*1;
1232      16'h030A:    Interpolation_Accum <= (DelayRead97
               >>>3)*6 + (DelayRead98>>>3)*2;
1233      16'h030B:    Interpolation_Accum <= (DelayRead97
               >>>3)*5 + (DelayRead98>>>3)*3;
1234      16'h030C:    Interpolation_Accum <= (DelayRead97
               >>>3)*4 + (DelayRead98>>>3)*4;
1235      16'h030D:    Interpolation_Accum <= (DelayRead97
               >>>3)*3 + (DelayRead98>>>3)*5;
1236      16'h030E:    Interpolation_Accum <= (DelayRead97
               >>>3)*2 + (DelayRead98>>>3)*6;
1237      16'h030F:    Interpolation_Accum <= (DelayRead97
               >>>3)*1 + (DelayRead98>>>3)*7;
1238      16'h0310:    Interpolation_Accum <= (DelayRead98
               >>>3)*8 + (DelayRead99>>>3)*0;
1239      16'h0311:    Interpolation_Accum <= (DelayRead98
               >>>3)*7 + (DelayRead99>>>3)*1;
1240      16'h0312:    Interpolation_Accum <= (DelayRead98
               >>>3)*6 + (DelayRead99>>>3)*2;
```

```
1241      16'h0313 :     Interpolation_Accum <= (DelayRead98
               >>>3)*5 + (DelayRead99>>>3)*3;
1242      16'h0314 :     Interpolation_Accum <= (DelayRead98
               >>>3)*4 + (DelayRead99>>>3)*4;
1243      16'h0315 :     Interpolation_Accum <= (DelayRead98
               >>>3)*3 + (DelayRead99>>>3)*5;
1244      16'h0316 :     Interpolation_Accum <= (DelayRead98
               >>>3)*2 + (DelayRead99>>>3)*6;
1245      16'h0317 :     Interpolation_Accum <= (DelayRead98
               >>>3)*1 + (DelayRead99>>>3)*7;
1246      16'h0318 :     Interpolation_Accum <= (DelayRead99
               >>>3)*8 + (DelayRead100>>>3)*0;
1247      16'h0319 :     Interpolation_Accum <= (DelayRead99
               >>>3)*7 + (DelayRead100>>>3)*1;
1248      16'h031A :     Interpolation_Accum <= (DelayRead99
               >>>3)*6 + (DelayRead100>>>3)*2;
1249      16'h031B :     Interpolation_Accum <= (DelayRead99
               >>>3)*5 + (DelayRead100>>>3)*3;
1250      16'h031C :     Interpolation_Accum <= (DelayRead99
               >>>3)*4 + (DelayRead100>>>3)*4;
1251      16'h031D :     Interpolation_Accum <= (DelayRead99
               >>>3)*3 + (DelayRead100>>>3)*5;
1252      16'h031E :     Interpolation_Accum <= (DelayRead99
               >>>3)*2 + (DelayRead100>>>3)*6;
```

```
1253      16'h031F:    Interpolation_Accum <= (DelayRead99
                  >>>3)*1 + (DelayRead100>>>3)*7;
1254      16'h0320:    Interpolation_Accum <= (DelayRead100
                  >>>3)*8 + (DelayRead101>>>3)*0;
1255      16'h0321:    Interpolation_Accum <= (DelayRead100
                  >>>3)*7 + (DelayRead101>>>3)*1;
1256      16'h0322:    Interpolation_Accum <= (DelayRead100
                  >>>3)*6 + (DelayRead101>>>3)*2;
1257      16'h0323:    Interpolation_Accum <= (DelayRead100
                  >>>3)*5 + (DelayRead101>>>3)*3;
1258      16'h0324:    Interpolation_Accum <= (DelayRead100
                  >>>3)*4 + (DelayRead101>>>3)*4;
1259      16'h0325:    Interpolation_Accum <= (DelayRead100
                  >>>3)*3 + (DelayRead101>>>3)*5;
1260      16'h0326:    Interpolation_Accum <= (DelayRead100
                  >>>3)*2 + (DelayRead101>>>3)*6;
1261      16'h0327:    Interpolation_Accum <= (DelayRead100
                  >>>3)*1 + (DelayRead101>>>3)*7;
1262      16'h0328:    Interpolation_Accum <= (DelayRead101
                  >>>3)*8 + (DelayRead102>>>3)*0;
1263      16'h0329:    Interpolation_Accum <= (DelayRead101
                  >>>3)*7 + (DelayRead102>>>3)*1;
1264      16'h032A:    Interpolation_Accum <= (DelayRead101
                  >>>3)*6 + (DelayRead102>>>3)*2;
```

```
1265      16'h032B :     Interpolation_Accum <= (DelayRead101
               >>>3)*5 + (DelayRead102>>>3)*3;
1266      16'h032C :     Interpolation_Accum <= (DelayRead101
               >>>3)*4 + (DelayRead102>>>3)*4;
1267      16'h032D :     Interpolation_Accum <= (DelayRead101
               >>>3)*3 + (DelayRead102>>>3)*5;
1268      16'h032E :     Interpolation_Accum <= (DelayRead101
               >>>3)*2 + (DelayRead102>>>3)*6;
1269      16'h032F :     Interpolation_Accum <= (DelayRead101
               >>>3)*1 + (DelayRead102>>>3)*7;
1270      16'h0330 :     Interpolation_Accum <= (DelayRead102
               >>>3)*8 + (DelayRead103>>>3)*0;
1271      16'h0331 :     Interpolation_Accum <= (DelayRead102
               >>>3)*7 + (DelayRead103>>>3)*1;
1272      16'h0332 :     Interpolation_Accum <= (DelayRead102
               >>>3)*6 + (DelayRead103>>>3)*2;
1273      16'h0333 :     Interpolation_Accum <= (DelayRead102
               >>>3)*5 + (DelayRead103>>>3)*3;
1274      16'h0334 :     Interpolation_Accum <= (DelayRead102
               >>>3)*4 + (DelayRead103>>>3)*4;
1275      16'h0335 :     Interpolation_Accum <= (DelayRead102
               >>>3)*3 + (DelayRead103>>>3)*5;
1276      16'h0336 :     Interpolation_Accum <= (DelayRead102
               >>>3)*2 + (DelayRead103>>>3)*6;
```

```
1277      16'h0337:    Interpolation_Accum <= (DelayRead102
               >>>3)*1 + (DelayRead103>>>3)*7;
1278      16'h0338:    Interpolation_Accum <= (DelayRead103
               >>>3)*8 + (DelayRead104>>>3)*0;
1279      16'h0339:    Interpolation_Accum <= (DelayRead103
               >>>3)*7 + (DelayRead104>>>3)*1;
1280      16'h033A:    Interpolation_Accum <= (DelayRead103
               >>>3)*6 + (DelayRead104>>>3)*2;
1281      16'h033B:    Interpolation_Accum <= (DelayRead103
               >>>3)*5 + (DelayRead104>>>3)*3;
1282      16'h033C:    Interpolation_Accum <= (DelayRead103
               >>>3)*4 + (DelayRead104>>>3)*4;
1283      16'h033D:    Interpolation_Accum <= (DelayRead103
               >>>3)*3 + (DelayRead104>>>3)*5;
1284      16'h033E:    Interpolation_Accum <= (DelayRead103
               >>>3)*2 + (DelayRead104>>>3)*6;
1285      16'h033F:    Interpolation_Accum <= (DelayRead103
               >>>3)*1 + (DelayRead104>>>3)*7;
1286      16'h0340:    Interpolation_Accum <= (DelayRead104
               >>>3)*8 + (DelayRead105>>>3)*0;
1287      16'h0341:    Interpolation_Accum <= (DelayRead104
               >>>3)*7 + (DelayRead105>>>3)*1;
1288      16'h0342:    Interpolation_Accum <= (DelayRead104
               >>>3)*6 + (DelayRead105>>>3)*2;
```

```
1289      16'h0343 :     Interpolation_Accum <= (DelayRead104
              >>>3)*5 + (DelayRead105>>>3)*3;
1290      16'h0344 :     Interpolation_Accum <= (DelayRead104
              >>>3)*4 + (DelayRead105>>>3)*4;
1291      16'h0345 :     Interpolation_Accum <= (DelayRead104
              >>>3)*3 + (DelayRead105>>>3)*5;
1292      16'h0346 :     Interpolation_Accum <= (DelayRead104
              >>>3)*2 + (DelayRead105>>>3)*6;
1293      16'h0347 :     Interpolation_Accum <= (DelayRead104
              >>>3)*1 + (DelayRead105>>>3)*7;
1294      16'h0348 :     Interpolation_Accum <= (DelayRead105
              >>>3)*8 + (DelayRead106>>>3)*0;
1295      16'h0349 :     Interpolation_Accum <= (DelayRead105
              >>>3)*7 + (DelayRead106>>>3)*1;
1296      16'h034A :     Interpolation_Accum <= (DelayRead105
              >>>3)*6 + (DelayRead106>>>3)*2;
1297      16'h034B :     Interpolation_Accum <= (DelayRead105
              >>>3)*5 + (DelayRead106>>>3)*3;
1298      16'h034C :     Interpolation_Accum <= (DelayRead105
              >>>3)*4 + (DelayRead106>>>3)*4;
1299      16'h034D :     Interpolation_Accum <= (DelayRead105
              >>>3)*3 + (DelayRead106>>>3)*5;
1300      16'h034E :     Interpolation_Accum <= (DelayRead105
              >>>3)*2 + (DelayRead106>>>3)*6;
```

```
1301      16'h034F:    Interpolation_Accum <= (DelayRead105
                  >>>3)*1 + (DelayRead106>>>3)*7;
1302      16'h0350:    Interpolation_Accum <= (DelayRead106
                  >>>3)*8 + (DelayRead107>>>3)*0;
1303      16'h0351:    Interpolation_Accum <= (DelayRead106
                  >>>3)*7 + (DelayRead107>>>3)*1;
1304      16'h0352:    Interpolation_Accum <= (DelayRead106
                  >>>3)*6 + (DelayRead107>>>3)*2;
1305      16'h0353:    Interpolation_Accum <= (DelayRead106
                  >>>3)*5 + (DelayRead107>>>3)*3;
1306      16'h0354:    Interpolation_Accum <= (DelayRead106
                  >>>3)*4 + (DelayRead107>>>3)*4;
1307      16'h0355:    Interpolation_Accum <= (DelayRead106
                  >>>3)*3 + (DelayRead107>>>3)*5;
1308      16'h0356:    Interpolation_Accum <= (DelayRead106
                  >>>3)*2 + (DelayRead107>>>3)*6;
1309      16'h0357:    Interpolation_Accum <= (DelayRead106
                  >>>3)*1 + (DelayRead107>>>3)*7;
1310      16'h0358:    Interpolation_Accum <= (DelayRead107
                  >>>3)*8 + (DelayRead108>>>3)*0;
1311      16'h0359:    Interpolation_Accum <= (DelayRead107
                  >>>3)*7 + (DelayRead108>>>3)*1;
1312      16'h035A:    Interpolation_Accum <= (DelayRead107
                  >>>3)*6 + (DelayRead108>>>3)*2;
```

```

1313      16'h035B :     Interpolation_Accum <= (DelayRead107
                  >>>3)*5 + (DelayRead108>>>3)*3;
1314      16'h035C :     Interpolation_Accum <= (DelayRead107
                  >>>3)*4 + (DelayRead108>>>3)*4;
1315      16'h035D :     Interpolation_Accum <= (DelayRead107
                  >>>3)*3 + (DelayRead108>>>3)*5;
1316      16'h035E :     Interpolation_Accum <= (DelayRead107
                  >>>3)*2 + (DelayRead108>>>3)*6;
1317      16'h035F :     Interpolation_Accum <= (DelayRead107
                  >>>3)*1 + (DelayRead108>>>3)*7;
1318      16'h0360 :     Interpolation_Accum <= (DelayRead108
                  >>>3)*8 + (DelayRead109>>>3)*0;
1319      16'h0361 :     Interpolation_Accum <= (DelayRead108
                  >>>3)*7 + (DelayRead109>>>3)*1;
1320      16'h0362 :     Interpolation_Accum <= (DelayRead108
                  >>>3)*6 + (DelayRead109>>>3)*2;
1321      16'h0363 :     Interpolation_Accum <= (DelayRead108
                  >>>3)*5 + (DelayRead109>>>3)*3;
1322      16'h0364 :     Interpolation_Accum <= (DelayRead108
                  >>>3)*4 + (DelayRead109>>>3)*4;
1323      16'h0365 :     Interpolation_Accum <= (DelayRead108
                  >>>3)*3 + (DelayRead109>>>3)*5;
1324      16'h0366 :     Interpolation_Accum <= (DelayRead108
                  >>>3)*2 + (DelayRead109>>>3)*6;

```

```
1325      16'h0367:    Interpolation_Accum <= (DelayRead108
               >>>3)*1 + (DelayRead109>>>3)*7;
1326      16'h0368:    Interpolation_Accum <= (DelayRead109
               >>>3)*8 + (DelayRead110>>>3)*0;
1327      16'h0369:    Interpolation_Accum <= (DelayRead109
               >>>3)*7 + (DelayRead110>>>3)*1;
1328      16'h036A:    Interpolation_Accum <= (DelayRead109
               >>>3)*6 + (DelayRead110>>>3)*2;
1329      16'h036B:    Interpolation_Accum <= (DelayRead109
               >>>3)*5 + (DelayRead110>>>3)*3;
1330      16'h036C:    Interpolation_Accum <= (DelayRead109
               >>>3)*4 + (DelayRead110>>>3)*4;
1331      16'h036D:    Interpolation_Accum <= (DelayRead109
               >>>3)*3 + (DelayRead110>>>3)*5;
1332      16'h036E:    Interpolation_Accum <= (DelayRead109
               >>>3)*2 + (DelayRead110>>>3)*6;
1333      16'h036F:    Interpolation_Accum <= (DelayRead109
               >>>3)*1 + (DelayRead110>>>3)*7;
1334      16'h0370:    Interpolation_Accum <= (DelayRead110
               >>>3)*8 + (DelayRead111>>>3)*0;
1335      16'h0371:    Interpolation_Accum <= (DelayRead110
               >>>3)*7 + (DelayRead111>>>3)*1;
1336      16'h0372:    Interpolation_Accum <= (DelayRead110
               >>>3)*6 + (DelayRead111>>>3)*2;
```

```

1337      16'h0373:    Interpolation_Accum <= (DelayRead110
                  >>>3)*5 + (DelayRead111>>>3)*3;
1338      16'h0374:    Interpolation_Accum <= (DelayRead110
                  >>>3)*4 + (DelayRead111>>>3)*4;
1339      16'h0375:    Interpolation_Accum <= (DelayRead110
                  >>>3)*3 + (DelayRead111>>>3)*5;
1340      16'h0376:    Interpolation_Accum <= (DelayRead110
                  >>>3)*2 + (DelayRead111>>>3)*6;
1341      16'h0377:    Interpolation_Accum <= (DelayRead110
                  >>>3)*1 + (DelayRead111>>>3)*7;
1342      16'h0378:    Interpolation_Accum <= (DelayRead111
                  >>>3)*8 + (DelayRead112>>>3)*0;
1343      16'h0379:    Interpolation_Accum <= (DelayRead111
                  >>>3)*7 + (DelayRead112>>>3)*1;
1344      16'h037A:    Interpolation_Accum <= (DelayRead111
                  >>>3)*6 + (DelayRead112>>>3)*2;
1345      16'h037B:    Interpolation_Accum <= (DelayRead111
                  >>>3)*5 + (DelayRead112>>>3)*3;
1346      16'h037C:    Interpolation_Accum <= (DelayRead111
                  >>>3)*4 + (DelayRead112>>>3)*4;
1347      16'h037D:    Interpolation_Accum <= (DelayRead111
                  >>>3)*3 + (DelayRead112>>>3)*5;
1348      16'h037E:    Interpolation_Accum <= (DelayRead111
                  >>>3)*2 + (DelayRead112>>>3)*6;

```

```
1349      16'h037F:    Interpolation_Accum <= (DelayRead111
                  >>>3)*1 + (DelayRead112>>>3)*7;
1350      16'h0380:    Interpolation_Accum <= (DelayRead112
                  >>>3)*8 + (DelayRead113>>>3)*0;
1351      16'h0381:    Interpolation_Accum <= (DelayRead112
                  >>>3)*7 + (DelayRead113>>>3)*1;
1352      16'h0382:    Interpolation_Accum <= (DelayRead112
                  >>>3)*6 + (DelayRead113>>>3)*2;
1353      16'h0383:    Interpolation_Accum <= (DelayRead112
                  >>>3)*5 + (DelayRead113>>>3)*3;
1354      16'h0384:    Interpolation_Accum <= (DelayRead112
                  >>>3)*4 + (DelayRead113>>>3)*4;
1355      16'h0385:    Interpolation_Accum <= (DelayRead112
                  >>>3)*3 + (DelayRead113>>>3)*5;
1356      16'h0386:    Interpolation_Accum <= (DelayRead112
                  >>>3)*2 + (DelayRead113>>>3)*6;
1357      16'h0387:    Interpolation_Accum <= (DelayRead112
                  >>>3)*1 + (DelayRead113>>>3)*7;
1358      16'h0388:    Interpolation_Accum <= (DelayRead113
                  >>>3)*8 + (DelayRead114>>>3)*0;
1359      16'h0389:    Interpolation_Accum <= (DelayRead113
                  >>>3)*7 + (DelayRead114>>>3)*1;
1360      16'h038A:    Interpolation_Accum <= (DelayRead113
                  >>>3)*6 + (DelayRead114>>>3)*2;
```

```
1361      16'h038B :     Interpolation_Accum <= (DelayRead113
               >>>3)*5 + (DelayRead114>>>3)*3;
1362      16'h038C :     Interpolation_Accum <= (DelayRead113
               >>>3)*4 + (DelayRead114>>>3)*4;
1363      16'h038D :     Interpolation_Accum <= (DelayRead113
               >>>3)*3 + (DelayRead114>>>3)*5;
1364      16'h038E :     Interpolation_Accum <= (DelayRead113
               >>>3)*2 + (DelayRead114>>>3)*6;
1365      16'h038F :     Interpolation_Accum <= (DelayRead113
               >>>3)*1 + (DelayRead114>>>3)*7;
1366      16'h0390 :     Interpolation_Accum <= (DelayRead114
               >>>3)*8 + (DelayRead115>>>3)*0;
1367      16'h0391 :     Interpolation_Accum <= (DelayRead114
               >>>3)*7 + (DelayRead115>>>3)*1;
1368      16'h0392 :     Interpolation_Accum <= (DelayRead114
               >>>3)*6 + (DelayRead115>>>3)*2;
1369      16'h0393 :     Interpolation_Accum <= (DelayRead114
               >>>3)*5 + (DelayRead115>>>3)*3;
1370      16'h0394 :     Interpolation_Accum <= (DelayRead114
               >>>3)*4 + (DelayRead115>>>3)*4;
1371      16'h0395 :     Interpolation_Accum <= (DelayRead114
               >>>3)*3 + (DelayRead115>>>3)*5;
1372      16'h0396 :     Interpolation_Accum <= (DelayRead114
               >>>3)*2 + (DelayRead115>>>3)*6;
```

```
1373      16'h0397:    Interpolation_Accum <= (DelayRead114
                  >>>3)*1 + (DelayRead115>>>3)*7;
1374      16'h0398:    Interpolation_Accum <= (DelayRead115
                  >>>3)*8 + (DelayRead116>>>3)*0;
1375      16'h0399:    Interpolation_Accum <= (DelayRead115
                  >>>3)*7 + (DelayRead116>>>3)*1;
1376      16'h039A:    Interpolation_Accum <= (DelayRead115
                  >>>3)*6 + (DelayRead116>>>3)*2;
1377      16'h039B:    Interpolation_Accum <= (DelayRead115
                  >>>3)*5 + (DelayRead116>>>3)*3;
1378      16'h039C:    Interpolation_Accum <= (DelayRead115
                  >>>3)*4 + (DelayRead116>>>3)*4;
1379      16'h039D:    Interpolation_Accum <= (DelayRead115
                  >>>3)*3 + (DelayRead116>>>3)*5;
1380      16'h039E:    Interpolation_Accum <= (DelayRead115
                  >>>3)*2 + (DelayRead116>>>3)*6;
1381      16'h039F:    Interpolation_Accum <= (DelayRead115
                  >>>3)*1 + (DelayRead116>>>3)*7;
1382      16'h03A0:    Interpolation_Accum <= (DelayRead116
                  >>>3)*8 + (DelayRead117>>>3)*0;
1383      16'h03A1:    Interpolation_Accum <= (DelayRead116
                  >>>3)*7 + (DelayRead117>>>3)*1;
1384      16'h03A2:    Interpolation_Accum <= (DelayRead116
                  >>>3)*6 + (DelayRead117>>>3)*2;
```

```

1385      16'h03A3:    Interpolation_Accum <= (DelayRead116
                  >>>3)*5 + (DelayRead117>>>3)*3;
1386      16'h03A4:    Interpolation_Accum <= (DelayRead116
                  >>>3)*4 + (DelayRead117>>>3)*4;
1387      16'h03A5:    Interpolation_Accum <= (DelayRead116
                  >>>3)*3 + (DelayRead117>>>3)*5;
1388      16'h03A6:    Interpolation_Accum <= (DelayRead116
                  >>>3)*2 + (DelayRead117>>>3)*6;
1389      16'h03A7:    Interpolation_Accum <= (DelayRead116
                  >>>3)*1 + (DelayRead117>>>3)*7;
1390      16'h03A8:    Interpolation_Accum <= (DelayRead117
                  >>>3)*8 + (DelayRead118>>>3)*0;
1391      16'h03A9:    Interpolation_Accum <= (DelayRead117
                  >>>3)*7 + (DelayRead118>>>3)*1;
1392      16'h03AA:    Interpolation_Accum <= (DelayRead117
                  >>>3)*6 + (DelayRead118>>>3)*2;
1393      16'h03AB:    Interpolation_Accum <= (DelayRead117
                  >>>3)*5 + (DelayRead118>>>3)*3;
1394      16'h03AC:    Interpolation_Accum <= (DelayRead117
                  >>>3)*4 + (DelayRead118>>>3)*4;
1395      16'h03AD:    Interpolation_Accum <= (DelayRead117
                  >>>3)*3 + (DelayRead118>>>3)*5;
1396      16'h03AE:    Interpolation_Accum <= (DelayRead117
                  >>>3)*2 + (DelayRead118>>>3)*6;

```

```

1397      16'h03AF:    Interpolation_Accum <= (DelayRead117
                  >>>3)*1 + (DelayRead118>>>3)*7;
1398      16'h03B0:    Interpolation_Accum <= (DelayRead118
                  >>>3)*8 + (DelayRead119>>>3)*0;
1399      16'h03B1:    Interpolation_Accum <= (DelayRead118
                  >>>3)*7 + (DelayRead119>>>3)*1;
1400      16'h03B2:    Interpolation_Accum <= (DelayRead118
                  >>>3)*6 + (DelayRead119>>>3)*2;
1401      16'h03B3:    Interpolation_Accum <= (DelayRead118
                  >>>3)*5 + (DelayRead119>>>3)*3;
1402      16'h03B4:    Interpolation_Accum <= (DelayRead118
                  >>>3)*4 + (DelayRead119>>>3)*4;
1403      16'h03B5:    Interpolation_Accum <= (DelayRead118
                  >>>3)*3 + (DelayRead119>>>3)*5;
1404      16'h03B6:    Interpolation_Accum <= (DelayRead118
                  >>>3)*2 + (DelayRead119>>>3)*6;
1405      16'h03B7:    Interpolation_Accum <= (DelayRead118
                  >>>3)*1 + (DelayRead119>>>3)*7;
1406      16'h03B8:    Interpolation_Accum <= (DelayRead119
                  >>>3)*8 + (DelayRead120>>>3)*0;
1407      16'h03B9:    Interpolation_Accum <= (DelayRead119
                  >>>3)*7 + (DelayRead120>>>3)*1;
1408      16'h03BA:    Interpolation_Accum <= (DelayRead119
                  >>>3)*6 + (DelayRead120>>>3)*2;

```

```

1409      16'h03BB:    Interpolation_Accum <= (DelayRead119
                  >>>3)*5 + (DelayRead120>>>3)*3;
1410      16'h03BC:    Interpolation_Accum <= (DelayRead119
                  >>>3)*4 + (DelayRead120>>>3)*4;
1411      16'h03BD:    Interpolation_Accum <= (DelayRead119
                  >>>3)*3 + (DelayRead120>>>3)*5;
1412      16'h03BE:    Interpolation_Accum <= (DelayRead119
                  >>>3)*2 + (DelayRead120>>>3)*6;
1413      16'h03BF:    Interpolation_Accum <= (DelayRead119
                  >>>3)*1 + (DelayRead120>>>3)*7;
1414      16'h03C0:    Interpolation_Accum <= (DelayRead120
                  >>>3)*8 + (DelayRead121>>>3)*0;
1415      16'h03C1:    Interpolation_Accum <= (DelayRead120
                  >>>3)*7 + (DelayRead121>>>3)*1;
1416      16'h03C2:    Interpolation_Accum <= (DelayRead120
                  >>>3)*6 + (DelayRead121>>>3)*2;
1417      16'h03C3:    Interpolation_Accum <= (DelayRead120
                  >>>3)*5 + (DelayRead121>>>3)*3;
1418      16'h03C4:    Interpolation_Accum <= (DelayRead120
                  >>>3)*4 + (DelayRead121>>>3)*4;
1419      16'h03C5:    Interpolation_Accum <= (DelayRead120
                  >>>3)*3 + (DelayRead121>>>3)*5;
1420      16'h03C6:    Interpolation_Accum <= (DelayRead120
                  >>>3)*2 + (DelayRead121>>>3)*6;

```

```

1421      16'h03C7:    Interpolation_Accum <= (DelayRead120
                  >>>3)*1 + (DelayRead121>>>3)*7;
1422      16'h03C8:    Interpolation_Accum <= (DelayRead121
                  >>>3)*8 + (DelayRead122>>>3)*0;
1423      16'h03C9:    Interpolation_Accum <= (DelayRead121
                  >>>3)*7 + (DelayRead122>>>3)*1;
1424      16'h03CA:    Interpolation_Accum <= (DelayRead121
                  >>>3)*6 + (DelayRead122>>>3)*2;
1425      16'h03CB:    Interpolation_Accum <= (DelayRead121
                  >>>3)*5 + (DelayRead122>>>3)*3;
1426      16'h03CC:    Interpolation_Accum <= (DelayRead121
                  >>>3)*4 + (DelayRead122>>>3)*4;
1427      16'h03CD:    Interpolation_Accum <= (DelayRead121
                  >>>3)*3 + (DelayRead122>>>3)*5;
1428      16'h03CE:    Interpolation_Accum <= (DelayRead121
                  >>>3)*2 + (DelayRead122>>>3)*6;
1429      16'h03CF:    Interpolation_Accum <= (DelayRead121
                  >>>3)*1 + (DelayRead122>>>3)*7;
1430      16'h03D0:    Interpolation_Accum <= (DelayRead122
                  >>>3)*8 + (DelayRead123>>>3)*0;
1431      16'h03D1:    Interpolation_Accum <= (DelayRead122
                  >>>3)*7 + (DelayRead123>>>3)*1;
1432      16'h03D2:    Interpolation_Accum <= (DelayRead122
                  >>>3)*6 + (DelayRead123>>>3)*2;

```

```

1433      16'h03D3:    Interpolation_Accum <= (DelayRead122
                  >>>3)*5 + (DelayRead123>>>3)*3;
1434      16'h03D4:    Interpolation_Accum <= (DelayRead122
                  >>>3)*4 + (DelayRead123>>>3)*4;
1435      16'h03D5:    Interpolation_Accum <= (DelayRead122
                  >>>3)*3 + (DelayRead123>>>3)*5;
1436      16'h03D6:    Interpolation_Accum <= (DelayRead122
                  >>>3)*2 + (DelayRead123>>>3)*6;
1437      16'h03D7:    Interpolation_Accum <= (DelayRead122
                  >>>3)*1 + (DelayRead123>>>3)*7;
1438      16'h03D8:    Interpolation_Accum <= (DelayRead123
                  >>>3)*8 + (DelayRead124>>>3)*0;
1439      16'h03D9:    Interpolation_Accum <= (DelayRead123
                  >>>3)*7 + (DelayRead124>>>3)*1;
1440      16'h03DA:    Interpolation_Accum <= (DelayRead123
                  >>>3)*6 + (DelayRead124>>>3)*2;
1441      16'h03DB:    Interpolation_Accum <= (DelayRead123
                  >>>3)*5 + (DelayRead124>>>3)*3;
1442      16'h03DC:    Interpolation_Accum <= (DelayRead123
                  >>>3)*4 + (DelayRead124>>>3)*4;
1443      16'h03DD:    Interpolation_Accum <= (DelayRead123
                  >>>3)*3 + (DelayRead124>>>3)*5;
1444      16'h03DE:    Interpolation_Accum <= (DelayRead123
                  >>>3)*2 + (DelayRead124>>>3)*6;

```

```

1445      16'h03DF:    Interpolation_Accum <= (DelayRead123
                  >>>3)*1 + (DelayRead124>>>3)*7;
1446      16'h03E0:    Interpolation_Accum <= (DelayRead124
                  >>>3)*8 + (DelayRead125>>>3)*0;
1447      16'h03E1:    Interpolation_Accum <= (DelayRead124
                  >>>3)*7 + (DelayRead125>>>3)*1;
1448      16'h03E2:    Interpolation_Accum <= (DelayRead124
                  >>>3)*6 + (DelayRead125>>>3)*2;
1449      16'h03E3:    Interpolation_Accum <= (DelayRead124
                  >>>3)*5 + (DelayRead125>>>3)*3;
1450      16'h03E4:    Interpolation_Accum <= (DelayRead124
                  >>>3)*4 + (DelayRead125>>>3)*4;
1451      16'h03E5:    Interpolation_Accum <= (DelayRead124
                  >>>3)*3 + (DelayRead125>>>3)*5;
1452      16'h03E6:    Interpolation_Accum <= (DelayRead124
                  >>>3)*2 + (DelayRead125>>>3)*6;
1453      16'h03E7:    Interpolation_Accum <= (DelayRead124
                  >>>3)*1 + (DelayRead125>>>3)*7;
1454      16'h03E8:    Interpolation_Accum <= (DelayRead125
                  >>>3)*8 + (DelayRead126>>>3)*0;
1455      16'h03E9:    Interpolation_Accum <= (DelayRead125
                  >>>3)*7 + (DelayRead126>>>3)*1;
1456      16'h03EA:    Interpolation_Accum <= (DelayRead125
                  >>>3)*6 + (DelayRead126>>>3)*2;

```

```

1457      16'h03EB :     Interpolation_Accum <= (DelayRead125
                  >>>3)*5 + (DelayRead126>>>3)*3;
1458      16'h03EC :     Interpolation_Accum <= (DelayRead125
                  >>>3)*4 + (DelayRead126>>>3)*4;
1459      16'h03ED :     Interpolation_Accum <= (DelayRead125
                  >>>3)*3 + (DelayRead126>>>3)*5;
1460      16'h03EE :     Interpolation_Accum <= (DelayRead125
                  >>>3)*2 + (DelayRead126>>>3)*6;
1461      16'h03EF :     Interpolation_Accum <= (DelayRead125
                  >>>3)*1 + (DelayRead126>>>3)*7;
1462      16'h03F0 :     Interpolation_Accum <= (DelayRead126
                  >>>3)*8 + (DelayRead127>>>3)*0;
1463      16'h03F1 :     Interpolation_Accum <= (DelayRead126
                  >>>3)*7 + (DelayRead127>>>3)*1;
1464      16'h03F2 :     Interpolation_Accum <= (DelayRead126
                  >>>3)*6 + (DelayRead127>>>3)*2;
1465      16'h03F3 :     Interpolation_Accum <= (DelayRead126
                  >>>3)*5 + (DelayRead127>>>3)*3;
1466      16'h03F4 :     Interpolation_Accum <= (DelayRead126
                  >>>3)*4 + (DelayRead127>>>3)*4;
1467      16'h03F5 :     Interpolation_Accum <= (DelayRead126
                  >>>3)*3 + (DelayRead127>>>3)*5;
1468      16'h03F6 :     Interpolation_Accum <= (DelayRead126
                  >>>3)*2 + (DelayRead127>>>3)*6;

```

```

1469      16'h03F7 :     Interpolation_Accum <= (DelayRead126
1470                  >>>3)*1 + (DelayRead127>>>3)*7;
1470      16'h03F8 :     Interpolation_Accum <= (DelayRead127
1471                  >>>3)*8 + (DelayRead128>>>3)*0;
1471      16'h03F9 :     Interpolation_Accum <= (DelayRead127
1472                  >>>3)*7 + (DelayRead128>>>3)*1;
1472      16'h03FA :     Interpolation_Accum <= (DelayRead127
1473                  >>>3)*6 + (DelayRead128>>>3)*2;
1473      16'h03FB :     Interpolation_Accum <= (DelayRead127
1474                  >>>3)*5 + (DelayRead128>>>3)*3;
1474      16'h03FC :     Interpolation_Accum <= (DelayRead127
1475                  >>>3)*4 + (DelayRead128>>>3)*4;
1475      16'h03FD :     Interpolation_Accum <= (DelayRead127
1476                  >>>3)*3 + (DelayRead128>>>3)*5;
1476      16'h03FE :     Interpolation_Accum <= (DelayRead127
1477                  >>>3)*2 + (DelayRead128>>>3)*6;
1477      16'h03FF :     Interpolation_Accum <= (DelayRead127
1478                  >>>3)*1 + (DelayRead128>>>3)*7;
1478
1479      default : Interpolation_Accum <= DelayRead0 ;
1480      endcase
1481  end
1482 //--Main Process

```

```
1483      always @(posedge sys_clk) begin // implementation
1484          ChorusLED = PedalON; // Set LED value
1485
1486          // Start calling the delay into BRAM
1487          if(PedalON == 0) begin // Device OFF
1488              Din = 24'b00000000000000000000000000;
1489              audio_out = audio_in;
1490          end
1491          else if(PedalON == 1) begin
1492
1493              tempRAM = Interpolation_Accum; // Output from
1494                  Interpolation Stage
1495
1496              BRAMOUT = tempRAM; // DEbug
1497
1498              if(audio_in[23] == 0) begin // positive
1499                  temp_Din = audio_in*ADC2reg; // Q24.0*Q0.8=
1500                      Q24.8
1501
1502                  Din = temp_Din[31:8];
1503
1504                  // Debugging Pitch Shifting
1505                  // Din = audio_in;
1506
1507              end
1508              else begin // negative
1509                  temp_audio_in = -audio_in;
```

```
1506          temp_Din = temp_audio_in*ADC2reg; //Q24.0*
1507          Q0.8= Q24.8
1508
1509          Din = -temp_Din[31:8];
1510
1511          // Debugging
1512          // Din = audio_in;
1513
1514          end
1515
1516          // What I think the output should be
1517          audio_out = audio_in + (tempRAM>>>1);
1518
1519          // Debugging just to see what the pitch
1520          // shifting output sounds like
1521
1522          // audio_out = tempRAM;
1523
1524          end // end pedal on case
1525
1526          end
1527
1528      endmodule
1529
1530
1531
1532  /* References
1533  http://www.donreiman.com/Chorus/Chorus.htm
1534  http://users.cs.cf.ac.uk/Dave.Marshall/CM0268/PDF/10
1535          _CM0268_Audio_FX.pdf
1536
1537  https://ccrma.stanford.edu/~orchi/Documents/DAFx.pdf
1538
1539  https://upcommons.upc.edu/bitstream/handle/2117/98219/pfc_doc.
1540          pdf?sequence=1&isAllowed=y
```

```
1527 http://richarddudas.com/pdfs/dudas_icmc2012.pdf  
1528 https://www.ti.com/lit/an/spraaa5/spraaa5.pdf  
1529 */
```

Listing II.9: Chorus Algorithm Verilog File

Chorus Pedal Test Bench Code

```
1 module cwb_chorus_tb;
2 reg signed [23:0] audio_in;
3 reg sys_clk;
4 reg [7:0] I2C_ADDR;
5 reg ENABLE_I2C_data_move;
6 reg [7:0] ADC1; // Time Step
7 reg [7:0] ADC2; // Dry/Wet Mix
8 reg [7:0] ADC3; //FIR/NOTFIR (1/0)
9 reg [7:0] ADC4; //Not Used
10 wire signed [23:0] audio_out;
11 wire ChorusLED;
12 wire [15:0] Addr1_debug;
13 wire [15:0] Addr2_debug;
14 wire [15:0] k_debug;
15 integer i;
16 integer j;
17 integer Iter = 1;
18 integer Iter2 = 500;
19 integer Iter3 = 0;
20 wire signed [23:0] Chorus1;
21 wire signed [23:0] BRAMOUT;
22 wire ConflictingWR;
23 wire DIR_DEBUG;
```

```
24 wire [15:0] MAXDELAY_DEBUG;
25 wire [15:0] Index_DEBUG;
26
27 cwb_chorus dut(.audio_in(audio_in),
28                 .sys_clk(sys_clk),
29                 .I2C_ADDR(I2C_ADDR),
30                 .ENABLE_I2C_data_move(ENABLE_I2C_data_move),
31                 .ADC1(ADC1),
32                 .ADC2(ADC2),
33                 .ADC3(ADC3),
34                 .ADC4(ADC4),
35                 .audio_out(audio_out),
36                 .ChorusLED(ChorusLED),
37                 .Addr1_debug(Addr1_debug),
38                 .Addr2_debug(Addr2_debug),
39                 .k_debug(k_debug),
40                 .Chorus1(Chorus1),
41                 .BRAMOUT(BRAMOUT),
42                 .ConflictingWR(ConflictingWR),
43                 .DIR_DEBUG(DIR_DEBUG),
44                 .MAXDELAY_DEBUG(MAXDELAY_DEBUG),
45                 .Index_DEBUG(Index_DEBUG));
46 // Define Clock Interval
47 always #5 sys_clk = ~sys_clk;
48
```

```
49      initial
50          begin
51              sys_clk <= 0;
52              I2C_ADDR <= 8'h47;
53              ENABLE_I2C_data_move <= 1;
54              audio_in <= 0;
55
56              ADC1 <= 2; // 0-255 Range - Max delay
57              ADC2 <= 255; // 0-255 Range - Dry/WetMix
58              ADC3 <= 13; // 0-255 Range - LO freq
59          // Inactive
60          ADC4<=0;
61
62
63      for (j=1;j<Iter2 ;j=j+1) begin
64          // Give some time
65          for (i=0;i<300000;i=i+10000) begin
66              #10 audio_in <= i ;
67          end
68
69          for (i=300000;i>-300000;i=i-10000) begin
70              #10 audio_in <= i ;
71          end
72          for (i=-300000;i<0;i=i+10000) begin
73              #10 audio_in <= i ;
```

```
74         end
75     end
76
77
78 //New set up testing
79 ADC1 <= 14; //0-255 Range - Max delay
80 ADC2 <=128; //0-255 Range - Dry/WetMix
81 ADC3 <=255; //0-255 Range - LO freq
82 //Inactive
83 ADC4<=0;
84
85 for (j=1;j<Iter2 ;j=j+1) begin
86     // Give some time
87     for (i=0;i<300000;i=i+10000) begin
88         #10 audio_in <= i ;
89     end
90
91     for (i=300000;i>-300000;i=i-10000) begin
92         #10 audio_in <= i ;
93     end
94     for (i=-300000;i<0;i=i+10000) begin
95         #10 audio_in <= i ;
96     end
97 end
98
```

```
99          #100 $finish ;  
100  
101      end  
102      endmodule
```

Listing II.10: Chorus Algorithm Verilog Test Bench File

Tremolo Pedal Code

```

1 module cwb_tremolo_eff(
2   input signed [23:0] audio_in ,
3   input sys_clk ,
4   input [7:0] I2C_ADDR ,
5   input ENABLE_I2C_data_move ,
6   input [7:0] ADC1, // Rate -> how fast the trem function is
7   going
8   input [7:0] ADC2, // Shape -> Triangular to rectangular
9   input [7:0] ADC3, // Depth -> How much the amplitude of the
10  trem is mixed into the og signal
11  input [7:0] ADC4, // Not Used
12  output reg signed [23:0] audio_out ,
13  output reg TremoloLED
14  //output reg [7:0] temp_MODULATOR,
15  //output reg [7:0] temp_tremCounter ,
16  //output reg temp_tremCLK ,
17  //output reg [32:0] output1 ,
18  //output reg [63:0] output2 ,
19  //output reg signed [7:0] output3
20 );
21 /*

```

```
22 24 bit audio comes into the block - active:  
23 Drive controls gain/boost of OG signal  
24 Level Control controls cut off volume  
25 Output signal  
26 */  
27 // Control Registers  
28 reg PedalON = 0;  
29  
30 reg [7:0] ADC1reg = 0;  
31 reg [7:0] ADC2reg = 0;  
32 reg [7:0] ADC3reg = 0;  
33 reg [7:0] ADC4reg = 0;  
34  
35 reg [7:0] ADC1regtemp = 0;  
36 reg [7:0] ADC2regtemp = 0;  
37 reg [7:0] ADC3regtemp = 0;  
38 reg [7:0] ADC4regtemp = 0;  
39  
40 // Block Specific Registers  
41 reg [23:0] counter = 0;  
42 reg [23:0] tremCLK = 0;  
43 reg [23:0] tremLim = 0; // Values set by POTS  
44 reg signed [7:0] tremCounter =0;  
45 reg TremDIR = 1;  
46 reg signed [7:0] ModSig = 0;
```

```
47 reg signed [16:0] outTemp = 0;  
48 reg signed [31:0] outTemp1 = 0;  
49 reg signed [63:0] outTemp2 = 0;  
50 reg signed [7:0] outTemp3 = 0;  
51 reg signed [7:0] TremThresh;  
52  
53  
54 //End Control Registers  
55 //
```

```
56 //
```

```
57 //
```

```
58 //
```

```
59 //
```

```
60 //
```

```
61      always @ (posedge sys_clk) begin
62          if(ENABLE_I2C_data_move == 1) begin // values are
63              stable and not changing so we can store them
64          if(I2C_ADDR == 8'h48) begin // correct address to
65              store values
66              if(ADC1+ADC2+ADC3 != 0) begin // this will be
67                  used later as a audio bypass
68                  PedalON = 1;
69              end else begin
70                  PedalON = 0;
71              end
72              // Set the values of the actual regs to have
73              // data be displayed to other modules
74              ADC1reg = ADC1;
75              ADC2reg = ADC2;
76              ADC3reg = ADC3;
77              // ADC4reg = ADC4;
78              // set temp reg values for when not addr or not
79              // valid data move
80              ADC1regtemp = ADC1reg;
81              ADC2regtemp = ADC2reg;
82              ADC3regtemp = ADC3reg;
83              // ADC4regtemp = ADC4reg;
```

```
80          end else begin
81              ADC1reg = ADC1reg;
82              ADC2reg = ADC2reg;
83              ADC3reg = ADC3reg;
84              // ADC4reg = ADC4reg;
85          end
86      end else begin
87          // nop ?
88      end
89  end // end control always block - put this in each pedal
       effect module
90 //
```

```
91 //
```

```
92 //
```

```
93 //
```

```
94 //
```

```
95      // Generate the clocks used for Trem
96      always @(posedge sys_clk) begin
97          counter = counter + 1;
98          tremLim = ADC1reg>>2;
99
100         // Set up a catch condition if both POTS are low, set
101         // some lowest Trem Limit acceptable
102         if(tremLim < 8'b00000001) begin
103             tremLim = 1;
104         end
105
106         if(counter > tremLim) begin // counter goes over the
107             // thresh so change clock and reset counter
108             counter = 1;
109             tremCLK = !tremCLK; // make opposite of original
110             // signal - we will use this as an enable on the
111             // trem CLK
112         end
113
114         // temp_tremCLK = tremCLK;
115     end
116
117 //
```

```
113 //  
114 //  
115 //  
116 //  
117 //  
118 // tremCounter controlling ->count up to 128 and then back  
      down to 0 and repeat -> dynamic updown counter  
119 always @(posedge tremCLK) begin  
120     // Change couter  
121     if(PedalON == 1) begin  
122         if(TremDIR == 1) begin  
123             tremCounter = tremCounter + 1;  
124         end  
125         else if(TremDIR == 0) begin  
126             tremCounter = tremCounter - 1;
```

```
127         end  
128  
129         // Change Direction  
130         if (tremCounter == 127 || tremCounter == -127)  
131             begin  
132                 TremDIR = !TremDIR;  
133             end  
134             // temp_tremCounter = tremCounter;  
135         end  
136     end  
137     //
```

```
138     //
```

```
139     //
```

```
140     //
```

```
141     //
```

```

142 // Tremeolo main processing block
143 always @(posedge sys_clk) begin
144     // Creation of the modulatin signal
145     TremThresh = (ADC2reg>>1);
146         //Debug to make sure the tremolo varies with
147         // input time
148
149     TremoloLED = PedalON;
150
151     if (PedalON == 1) begin
152         if (tremCounter > TremThresh && tremCounter[7] ==
153             0) begin
154             ModSig = 8'b01111111;
155         end
156         else if (-tremCounter > TremThresh && tremCounter
157             [7] == 1) begin
158             ModSig = 8'b10000001;
159         end
160         else begin
161             ModSig = tremCounter;
162         end
163
164         //temp_tremCounter = tremCounter;
165         //temp_MODULATOR = ModSig; //DEBUGGING
166
167 
```

```
163 // This is the modulated input
164 outTemp = (ModSig)*ADC3reg; // Multiply by 16 bits
    with 8 fractional bits
165 //Q8.0*Q8.0 = Q16.0 -> in case of overflow allow
    extra bit
166
167 outTemp3 = outTemp[16:9];
168 // output3 = outTemp3;
169
170
171 if (audio_in[23] == 0 && ModSig[7] == 0) begin // positive value
172     outTemp1= audio_in*ModSig; // mix dry (original)
        with wet signal (Amplitude modulated
        signal)
173     audio_out = outTemp1[31:8]; //Q24.0*Q8.0 = Q32
        .0
174 end
175 else if (audio_in[23] == 0 && ModSig[7] == 1) begin
        // positive audio in neg mod signal
176     outTemp1= audio_in*(-ModSig); // mix dry (
        original) with wet signal (Amplitude
        modulated signal)
177     audio_out = outTemp1[31:8]; //Q24.0*Q8.0 = Q32
        .0
```

```
178         end
179     else if (audio_in[23] == 1 && ModSig[7] == 0) begin
180         // positive audio in neg mod signal
181         outTemp1= -audio_in*(ModSig); // mix dry (
182             original) with wet signal (Amplitude
183             modulated signal)
184         audio_out = -outTemp1[31:8]; // Q24.0*Q8.0 =
185             Q32.0
186         end
187     else begin
188         // outTemp1= -audio_in*outTemp3;// mix dry (
189             original) with wet signal (Amplitude modulated signal)
190         // audio_out = -outTemp1[31:8]; // mix dry (
191             original) with wet signal (Amplitude modulated signal)
192         outTemp1= -audio_in*(-ModSig); // mix dry (
193             original) with wet signal (Amplitude
194             modulated signal)
195         audio_out = -outTemp1[31:8]; // Q24.0*Q8.0 =
196             Q32.0
197         end
198     end
199 else begin // Pedal Off
200     audio_out = audio_in;
201 end
202
```

```
194      // TremoloLED = PedalON;  
195  end // End main tremolo processing block  
196 endmodule // end the module of the Tremolo file
```

Listing II.11: Tremolo Algorithm Verilog File

Tremolo Pedal Test Bench Code

```
1  `timescale 1ns / 1ns
2
3 // module cwb_distortion_tb;
4 // reg [23:0] audio_in_tb;
5 // wire [23:0] audio_out_tb;
6 // reg sys_clk_tb;
7 // reg [7:0] enable_tb;
8 // reg [11:0] distortion_tb;
9 // reg [11:0] level_tb;
10
11 module cwb_tremolo_tb;
12 reg [23:0] audio_in;
13 reg sys_clk;
14 reg [7:0] I2C_ADDR;
15 reg ENABLE_I2C_data_move;
16 reg [7:0] ADC1; // Rate
17 reg [7:0] ADC2; // Shape
18 reg [7:0] ADC3; // Depth
19 reg [7:0] ADC4; // Not Used
20 wire [23:0] audio_out;
21 wire TremoloLED;
22 wire [7:0] temp_MODULATOR;
23 wire [7:0] temp_tremCounter;
```

```
24 wire [32:0] output1;
25 wire [63:0] output2;
26 wire [7:0] output3;
27 //TEST VECTORS
28 // wire [23:0] OverPosThresh_tb;
29 // wire [23:0] OverNegThresh_tb;
30 // wire [23:0] InPlusDrive_tb;
31 // wire [23:0] InMinusDrive_tb;
32 integer i;
33 integer j;
34 integer Iter = 120;
35
36 //CWB_distortion_pedal (audio_in , audio_out ,sys_clk ,enable ,
37 // distortion ,level );
37 cwb_tremolo_eff dut (.audio_in(audio_in) ,
38 .sys_clk(sys_clk) ,
39 .I2C_ADDR(I2C_ADDR) ,
40 .ENABLE_I2C_data_move(
41 .ENABLE_I2C_data_move) ,
42 .ADC1(ADC1) ,
43 .ADC2(ADC2) ,
44 .ADC3(ADC3) ,
45 .ADC4(ADC4) ,
46 .audio_out(audio_out) ,
. TremoloLED(TremoloLED));
```

```
47 // .temp_MODULATOR(temp_MODULATOR) ,  
48 // .temp_tremCounter(temp_tremCounter  
),  
49 // .output1(output1) ,  
50 // .output2(output2) ,  
51 // .output3(output3));  
52 // OverPosThresh_tb ,  
53 // OverNegThresh_tb ,  
54 // InPlusDrive_tb ,  
55 // InMinusDrive_tb );  
56  
57 // Define Clock Interval  
58 always #5 sys_clk = ~sys_clk ;  
59  
60 initial  
61 begin  
62 sys_clk <= 0;  
63 I2C_ADDR <= 8'h48 ;  
64 ENABLE_I2C_data_move <= 1;  
65 audio_in <= 1;  
66  
67 ADC1 <= 200; // 0-255 Range - Rate  
68 ADC2 <= 254; // 0-255 Range - Shape  
69 ADC3 <= 255; // 0-255 Range - Dry/WetMix  
70 // Inactive
```

```
71      ADC4<=0;  
72  
73      for(j=1;j<Iter;j=j+1) begin  
74          // Give some time  
75          for(i=1;i<150000;i=i+1000) begin  
76              #10 audio_in <= i;  
77          end  
78  
79          for(i=150000;i>-150000;i=i-1000) begin  
80              #10 audio_in <= i;  
81          end  
82          for(i=-150000;i<1;i=i+1000) begin  
83              #10 audio_in <= i;  
84          end  
85      end  
86  
87 ADC3 <=255; // 0-255 Range - Dry/WetMix  
88 ADC2 <=128; // 0-255 Range - Shape  
89  
90      for(j=1;j<Iter;j=j+1) begin  
91          // Give some time  
92          for(i=1;i<150000;i=i+1000) begin  
93              #10 audio_in <= i;  
94          end  
95
```

```
96      for ( i=150000;i >-150000;i=i -1000) begin
97          #10 audio_in <= i ;
98      end
99      for ( i=-150000;i <1;i=i +1000) begin
100          #10 audio_in <= i ;
101      end
102  end
103
104 ADC3 <=127; // 0-255 Range - Dry/WetMix
105 ADC2 <=254; // 0-255 Range - Shape
106 ADC1 <=254;
107
108      for ( j=1;j<Iter ;j=j +1) begin
109          // Give some time
110          for ( i=1;i <150000;i=i +1000) begin
111              #10 audio_in <= i ;
112          end
113
114          for ( i=150000;i >-150000;i=i -1000) begin
115              #10 audio_in <= i ;
116          end
117          for ( i=-150000;i <1;i=i +1000) begin
118              #10 audio_in <= i ;
119          end
120      end
```

```
121
122 ADC3 <=127; // 0-255 Range - Dry/WetMix
123 ADC2 <=127; // 0-255 Range - Shape
124     for(j=1;j<Iter;j=j+1) begin
125         // Give some time
126         for(i=1;i<150000;i=i+1000) begin
127             #10 audio_in <= i;
128         end
129
130         for(i=150000;i>-150000;i=i-1000) begin
131             #10 audio_in <= i;
132         end
133         for(i=-150000;i<1;i=i+1000) begin
134             #10 audio_in <= i;
135         end
136     end
137     #10 $finish;
138
139 end
140 endmodule
```

Listing II.12: Tremolo Algorithm Verilog Test Bench File

Noise Gate Pedal Code

```
1 module cwb_noise_gate(
2     input signed [23:0] audio_in ,
3     input sys_clk ,
4     input [7:0] I2C_ADDR,
5     input ENABLE_I2C_data_move ,
6     input [7:0] ADC1_THRESH,
7     input [7:0] ADC2,
8     input [7:0] ADC3, // Not Used
9     input [7:0] ADC4, // Not Used
10    output reg signed [23:0] audio_out ,
11    output reg NoiseGateLED
12 //    output reg temp_gcn ,
13 //    output reg [23:0] temp_gsn1 ,
14 //    output reg [23:0] temp_gsn_previous ,
15 //    output reg [47:0] temp_DynamicMultResult ,
16 //    output reg [23:0] temp_THRESHOLD ,
17 //    output reg [23:0] temp_Ac ,
18 //    output reg [23:0] temp_Rc ,
19 //    output reg [2:0] temp_gsn_state
20 );
21
22
23 // Control Registers
```

```
24      reg PedalON = 0;
25
26      reg [7:0] ADC1reg = 0;
27      reg [7:0] ADC2reg = 0;
28      reg [7:0] ADC3reg = 0;
29      reg [7:0] ADC4reg = 0;
30
31      reg [7:0] ADC1regtemp = 0;
32      reg [7:0] ADC2regtemp = 0;
33      reg [7:0] ADC3regtemp = 0;
34      reg [7:0] ADC4regtemp = 0;
35
36      reg [23:0] THRESH = 0;
37      reg [23:0] HoldTime = 40; // 250 ~ 5ms
38
39 // Intermediate Signals
40      reg [23:0] mag_input = 0;
41      reg gcn = 0;
42      reg [23:0] gsn = 24'h800000; // initilize to not throw any
43                                errors
44      reg [23:0] gsn_previous = 24'h800000; // initilize ont to
45                                throw any errors
46      reg [61:0] DynamicMultResult = 0;
```

```
47      reg [61:0] DynamicMultResult_2 = 0;
48      reg [23:0] AttackCounter = 0;
49      reg [23:0] ReleaseCounter = 0;
50
51      reg [23:0] a_A = 24'b01111010011111100110101; //Q1.23 for
52          .001s
53
54      reg [23:0] one_minus_a_A = 24'b00000101100000011001010;
55          //Q1.23 for .001s
56
57 //      reg [23:0] a_A = 24'b0111110110000101000100; //Q1.23
58     for .005s
59
60 //      reg [23:0] one_minus_a_A = 24'b000000000001110011001001;
61     //Q1.23 for .005s
62
63 //      reg [23:0] a_A = 24'b01111111000110011010; //Q1.23
64     for .05s
65
66 //      reg [23:0] one_minus_a_A = 24'b000000000001110011001001;
67     //Q1.23 for .05s
68
69 //      reg [23:0] a_A = 24'b01111111011000001010001; //Q1.23
70     for .01s
71
72 //      reg [23:0] one_minus_a_A = 24'b00000000100011110101110;
```

```
63      // reg [23:0] a_A = 24'b0111111111000110011010; //Q1.23
          for .1s
64      // reg [23:0] one_minus_a_A = 24'b00000000000011001100101;
          //Q1.23 for .1s
65
66
67
68
69
70      reg [23:0] a_R = 24'b0111111111000110011010; //Q1.23 for
          .1s
71      reg [23:0] one_minus_a_R = 24'b00000000000011001100101;
          //Q1.23 for .1s
72
73 //      reg [23:0] a_R = 24'b011111110110000010100; //Q1.23
          for .02s
74 //      reg [23:0] one_minus_a_R = 24'b00000001000111010111011;
          //Q1.23 for .02s
75
76 //      reg [23:0] a_R = 24'b0111110110000101000100; //Q1.23
          for .005s
77 //      reg [23:0] one_minus_a_R = 24'b00000000000110011001001;
          //Q1.23 for .005s
78
79
```

```
80      // testing 12/23/2019
81      reg isNeg = 0; // if input is negative val == 1
82      reg [47:0] gamma1 = 0;
83      reg [47:0] gamma2 = 0;
84      reg [47:0] beta1 = 0;
85      reg [47:0] beta2 = 0;
86      reg [47:0] gsn_accumulate1 = 0;
87      reg [47:0] gsn_accumulate2 = 0;
88
89      reg [47:0] yn_accumulated = 0;
90
91
92      //End Control Registers
93 //
```

```
94
95
96 //
```

```
97
98      always @ (posedge sys_clk) begin
99          if (ENABLE_I2C_data_move == 1) begin // values are
                                         stable and not changing so we can store them
```

```
100
101    if (I2C_ADDR == 8'h4B) begin // correct address to
102        store values
103
104    if (ADC1_THRESH != 0) begin // this will be used
105        later as a audio bypass
106
107    PedalON = 1;
108
109    end else begin
110
111    PedalON = 0;
112
113    end
114
115    // Set the values of the actual regs to have
116    // data be displayed to other modules
117
118    ADC1reg = ADC1_THRESH;
119
120    // ADC2reg = ADC2_Level;
121
122    // ADC3reg = ADC3;
123
124    // ADC4reg = ADC4;
125
126    // set temp reg values for when not addr or not
127    // valid data move
128
129    ADC1regtemp = ADC1reg;
130
131    // ADC2regtemp = ADC2reg;
132
133    // ADC3regtemp = ADC3reg;
134
135    // ADC4regtemp = ADC4reg;
136
137    end else begin
138
139    ADC1reg = ADC1reg;
140
141    // ADC2reg = ADC2reg;
142
143    // ADC3reg = ADC3reg;
```

```
121          // ADC4reg = ADC4reg;  
122      end  
123  end else begin  
124      // nop ?  
125  end  
126 end  
127 //end control always block - put this in each pedal effect  
     module  
128  
129 //
```

```
130     always @ (posedge sys_clk) begin // Control the output  
           audio and modification  
131     NoiseGateLED = PedalON;  
132     THRESH[13:6] = ADC1reg; // -> Will mostlikely need to shift  
           this value around  
133  
134 // temp_THRESH = THRESH; // DEBUG  
135  
136     if(PedalON == 1) begin // Algorithm active  
137         // 1. Take the magnitude of the input signal  
138         if(audio_in[23] == 1'b1) begin  
139             mag_input = -audio_in;  
140             isNeg = 1;
```

```
141         end
142     else begin
143         mag_input = audio_in;
144         isNeg = 0;
145     end
146
147 // 2. Pass through the gain computer. Compared to
148 // threshold set on POT ->NEEDS TO BE TUNED
149
150     if( mag_input < THRESH) begin // below thresh
151         gcn = 0;
152         AttackCounter = AttackCounter + 1;
153         ReleaseCounter = 0;
154     end
155     else begin // above thresh
156         gcn = 1;
157         AttackCounter = 0;
158         ReleaseCounter = ReleaseCounter + 1;
159     end
160
161     temp_gcn = gcn; //DEBUG
162     temp_Ac = AttackCounter; //DEBUG
163     temp_Rc = ReleaseCounter; //DEBUG
```

```
164 // 3. Compute the gain smoothing relative to attack
     , release and hold
165 gsn_previous = gsn; // this will be the
                      previous gsn signal used in calculations
166 /*with reference to the gain smoothing we know
   that gsn can be one but with our code it
   mightnever be that way
167 so we can take some creative liberties and
   say that if g_s[n-1] = Q1.23 =
   0.111111111XXXXXX (10 ones)
168 then we can approximate g_s[n] as one
   otherwise it will be less than 'one and
   therefore be zero
169 */
170
171 // with refference to 12/23 notes calc g_s[n]
     for eqn 1 and 2
172 // Could make more efficient by nesting calcs
     in conditionals
173 gamma1 = (a_A * gsn_previous)<<1; // Q2.46 =
   Q1.23*Q1.23 -> really Q1.47 so shift left
   by 1
174 gamma2 = (a_R * gsn_previous)<<1; // Q2.46 =
   Q1.23*Q1.23 -> really Q1.47 so shift left
   by 1
```

```

175
176     beta1 = {(one_minus_a_A*gcn)<<1,23'b0}; //Q2
           .23=Q1.23*Q1.0 -> really want Q1.24 so
           shift left 1 -> align zeros for addition by
           LSbit zero pad by 23 to get Q1.47
177     beta2 = {(one_minus_a_R*gcn)<<1,23'b0}; //Q2
           .23=Q1.23*Q1.0 -> really want Q1.24 so
           shift left 1 -> align zeros for addition by
           LSbit zero pad by 23 to get Q1.47
178
179     gsn_accumulate1 = gamma1 + beta1;
180     gsn_accumulate2 = gamma2 + beta2;
181
182     //now we can get into the four cases and we
           will assign a default case of outputting 1
           as a gain (we will assume this default
           state is an error state
183     if(AttackCounter > HoldTime && {gcn,23'b0} <=
           gsn_previous) begin
184         //
185         gsn = gsn_accumulate1[47:24];
186     //           temp_gsn_state = 1;
187     end
188     else if(ReleaseCounter > HoldTime && {gcn,23'
           b0} > gsn_previous) begin

```

```
189          //  
190          gsn = gsn_accumulate2[47:24];  
191      //  
192      end  
193      else if(AttackCounter <= HoldTime) begin  
194          //  
195          gsn = gsn_previous;  
196      //  
197      temp_gsn_state = 3;  
198      end  
199      else if(ReleaseCounter <= HoldTime) begin  
200          //  
201          gsn = gsn_previous;  
202      //  
203      temp_gsn_state = 4;  
204      end  
205      else begin // default cases thus in error  
206          //  
207          gsn = 24'h000000;  
208      end  
209      //  
210      temp_gsn1 = gsn;  
211      //  
212      temp_gsn_previous = gsn_previous;  
213  
214      // 4. Final output mixing of values
```

```
213          // yn = xn*gsn -> Q24.0 * Q1.23 = Q25.23 -> we
214          // will shift left by one (Q24.24) and then
215          // take only the 24 MSbits
216
217          //bc multiplying by neg is bad in Q point , we
218          // will mult the mag of input adn account for
219          // sign later
220
221          yn_accumulated = (mag_input * gsn) << 1;
222
223
224          //temp_DynamicMultResult = yn_accumulated; // DEBUG
225
226          // therefore
227          if(isNeg == 1) begin // negative input
228              audio_out = -yn_accumulated[47:24];
229          end
230
231          else begin // positive input
232              audio_out = yn_accumulated[47:24];
233          end
234
235          end //End if algo on
236
237          else begin
238              audio_out = audio_in;
239
240              gcn = 0;
241
242              gsn = 24'h800000; // initilize to not perm
243
244              mute
245
246              gsn_previous = 24'h800000; // initilize to
247
248              not perm mute
```

```
231      end // End if algo on  
232  end  
233 endmodule
```

Listing II.13: Noise Gate Algorithm Verilog File

Noise Gate Pedal Test Bench Code

```
1 module cwb_noise_gate_tb;
2     reg signed [23:0] audio_in;
3     reg sys_clk;
4     reg [7:0] I2C_ADDR;
5     reg ENABLE_I2C_data_move;
6     reg [7:0] ADC1_THRESH;
7     reg [7:0] ADC2; // Not Used
8     reg [7:0] ADC3; // Not Used
9     reg [7:0] ADC4; // Not Used
10    wire signed [23:0] audio_out;
11    wire NoiseGateLED;
12 //    wire temp_gcn;
13 //    wire [23:0] temp_gsn1;
14 //    wire [23:0] temp_gsn_previous;
15 //    wire [47:0] temp_DynamicMultResult;
16 //    wire [23:0] temp_THRESH;
17 //    wire [23:0] temp_Ac;
18 //    wire [23:0] temp_Rc;
19 //    wire [2:0] temp_gsn_state;
20    //For loop indexing
21 integer i;
22     integer j;
23     integer Iter = 10;
```

```
24      integer Iter2 = 10;
25
26 cwb_noise_gate DUT(.audio_in(audio_in),
27                      .sys_clk(sys_clk),
28                      .I2C_ADDR(I2C_ADDR),
29                      .ENABLE_I2C_data_move(ENABLE_I2C_data_move)
30
31                      ,
32                      .ADC1_THRESH(ADC1_THRESH),
33                      .ADC2(ADC2), // Inactive
34                      .ADC3(ADC3), // Inactive
35                      .ADC4(ADC4), // Inactive
36                      .audio_out(audio_out),
37                      .NoiseGateLED(NoiseGateLED));
38
39 //                      .temp_gcn(temp_gcn),
40 //                      .temp_gsn1(temp_gsn1),
41 //                      .temp_gsn_previous(temp_gsn_previous),
42 //                      .temp_DynamicMultResult(
43 //                        temp_DynamicMultResult),
44 //                      .temp_THRESH(temp_THRESH),
45 //                      .temp_Ac(temp_Ac),
46 //                      .temp_Rc(temp_Rc),
47 //                      .temp_gsn_state(temp_gsn_state));
48
49 // Define Clock Interval
50
51 always #1 sys_clk = ~sys_clk;
```

```
47
48 initial
49 begin
50     sys_clk <= 0;
51     I2C_ADDR <= 8'h4B;
52     ENABLE_I2C_data_move <= 1;
53     ADC1_THRESH <= 255; // 0-255 Range
54     audio_in <= 0;
55
56     // Inactive
57     ADC2 <=0;
58     ADC3 <=0;
59     ADC4<=0;
60
61
62 //      for(j=1;j<Iter ;j=j+1) begin
63 //          // Give some time
64 //          for(i=1;i<150000;i=i+100) begin
65 //              #10 audio_in <= i ;
66 //          end
67
68 //          for(i=150000;i>-150000;i=i-100) begin
69 //              #10 audio_in <= i ;
70 //          end
71 //          for(i=-150000;i<1;i=i+100) begin
```

```
72 // #10 audio_in <= i ;
73 // end
74 // end
75
76 // Change
77 //ADC1_THRESH <= 128; //0-255 Range
78 for(j=1;j<Iter;j=j+1) begin
79     // Give some time
80     for(i=1;i<15000;i=i+1000) begin
81         #10 audio_in <= i;
82     end
83
84     for(i=15000;i>-15000;i=i-1000) begin
85         #10 audio_in <= i;
86     end
87     for(i=-15000;i<1;i=i+1000) begin
88         #10 audio_in <= i;
89     end
90 end
91
92
93
94 // Change
95 //ADC1_THRESH <= 64; //0-255 Range
96 for(j=1;j<Iter;j=j+1) begin
```

```
97      // Give some time
98      for ( i=1;i<150000;i=i+1000) begin
99          #10 audio_in <= i ;
100     end
101
102     for ( i=150000;i>-150000;i=i-1000) begin
103         #10 audio_in <= i ;
104     end
105     for ( i=-150000;i<1;i=i+1000) begin
106         #10 audio_in <= i ;
107     end
108 end
109
110 // // Change
111 //      ADC1_THRESH <= 16; //0-255 Range
112
113
114     for ( j=1;j<Iter2 ;j=j+1) begin
115         // Give some time
116         for ( i=1;i<100000;i=i+100) begin
117             #10 audio_in <= i ;
118         end
119
120         for ( i=100000;i>-100000;i=i-100) begin
121             #10 audio_in <= i ;
```

```
122      end
123      for ( i=-100000;i<1;i=i+100) begin
124          #10 audio_in <= i ;
125      end
126      end
127
128      #10 $finish ;
129
130 end
131 endmodule
```

Listing II.14: Noise Gate Algorithm Verilog Test Bench File

Compressor Pedal Code

```
1 module cwb_equalizer(
2     input signed [23:0] audio_in ,
3     input sys_clk ,
4     input [7:0] I2C_ADDR ,
5     input ENABLE_I2C_data_move ,
6     input [7:0] ADC1, // Time Step
7     input [7:0] ADC2, // Dry / Wet Mix
8     input [7:0] ADC3, // FIR / NOTFIR (1/0)
9     input [7:0] ADC4, // Not Used
10    output reg signed [23:0] audio_out ,
11    output reg EQLed ,
12    output reg signed [23:0] BASS_response ,
13    output reg signed [23:0] MIDS_response ,
14    output reg signed [23:0] TREB_response ,
15    output reg signed [8+23:0] BASS_EQ_ACCUM_DEBUG,
16    output reg signed [8+23:0] MID_EQ_ACCUM_DEBUG,
17    output reg signed [8+23:0] TREBLE_EQ_ACCUM_DEBUG) ;
18
19
20 // Control Registers
21 reg [7:0] ADC1reg = 0;
22 reg [7:0] ADC2reg = 0;
23 reg [7:0] ADC3reg = 0;
```

```
24
25 reg [8:0] ADC1reg_signed = 0;
26 reg [8:0] ADC2reg_signed = 0;
27 reg [8:0] ADC3reg_signed = 0;
28 // reg [7:0] ADC4reg = 0;
29
30 reg [7:0] ADC1regtemp = 0;
31 reg [7:0] ADC2regtemp = 0;
32 reg [7:0] ADC3regtemp = 0;
33 // reg [7:0] ADC4regtemp = 0;
34
35 reg PedalON = 0;
36
37 // Filter Bank Output
38 wire signed [39:0] BASS_EQ_OUT;
39 wire signed [39:0] MID_EQ_OUT;
40 wire signed [36:0] TREBLE_EQ_OUT;
41 reg signed [39:0] BASS_EQ_OUTreg;
42 reg signed [39:0] MID_EQ_OUTreg;
43 reg signed [36:0] TREBLE_EQ_OUTreg;
44
45 reg signed [23:0] tempBASS;
46 reg signed [23:0] tempMIDS;
47 reg signed [23:0] tempTREB;
```

```
49
50 //ADC mult Accum regs
51 reg signed [9+24:0] BASS_EQ_ACCUM;
52 reg signed [9+24:0] MID_EQ_ACCUM;
53 reg signed [9+24:0] TREBLE_ACCUM;
54
55
56 // Call the BASS EQ module
57 // input clk ;
58 // input clk_enable ;
59 // input reset ;
60 // input signed [23:0] filter_in ; // sfix24
61 // output signed [39:0] filter_out ; // sfix40_En15
62 LPF_v4 BASS_EQ (.clk(sys_clk) ,
63 .clk_enable(1'b1) ,
64 .reset(1'b0) ,
65 .filter_in(audio_in) ,
66 .filter_out(BASS_EQ_OUT)) ;
67
68 // // Call the MIDS EQ module
69 // input clk ;
70 // input clk_enable ;
71 // input reset ;
72 // input signed [23:0] filter_in ; // sfix24
73 // output signed [39:0] filter_out ; // sfix40_En15
```

```

74 BPF_v1 MIDS_EQ (.clk(sys_clk),
75 .clk_enable(1'b1),
76 .reset(1'b0),
77 .filter_in(audio_in),
78 .filter_out(MID_EQ_OUT));
79 // Call the TREBLE EQ module
80 //input clk;
81 //input clk_enable;
82 //input reset;
83 //input signed [23:0] filter_in; //sfixed24
84 //output signed [36:0] filter_out; //sfixed37_En11
85 HPF_v1 TREB_EQ (.clk(sys_clk),
86 .clk_enable(1'b1),
87 .reset(1'b0),
88 .filter_in(audio_in),
89 .filter_out(TREBLE_EQ_OUT));
90 //
91 // Control Loop Always block to store ADC data
92 always @ (posedge sys_clk) begin
93 if (ENABLE_I2C_data_move == 1) begin //values are stable
94 and not changing so we can store them
95 if (I2C_ADDR == 8'h44) begin //correct address to
96 store values

```

```
95      if(ADC1+ADC2+ADC3 != 0) begin // this will be
96          used later as a audio bypass
97      end else begin
98          PedalON = 1;
99      end
100     // Set the values of the actual regs to have data
101    be displayed to other modules
102    ADC1reg = ADC1;
103    ADC2reg = ADC2;
104    ADC3reg = ADC3;
105    // ADC4reg = ADC4;
106    // set temp reg values for when not addr or not
107    valid data move
108    ADC1regtemp = ADC1reg;
109    ADC2regtemp = ADC2reg;
110    ADC3regtemp = ADC3reg;
111    // ADC4regtemp = ADC4reg;
112    end else begin
113        ADC1reg = ADC1reg;
114        ADC2reg = ADC2reg;
115        ADC3reg = ADC3reg;
116        // ADC4reg = ADC4reg;
117    end
```

```
117           end else begin  
118                   // nop ?  
119           end  
120       end // end always  
121  
122 //--Main_Process
```

```

123 always @(posedge sys_clk) begin
124 ADC1reg_signed = {1'b0,ADC1reg};
125 ADC2reg_signed = {1'b0,ADC2reg};
126 ADC3reg_signed = {1'b0,ADC3reg};
127
128 BASS_EQ_OUTreg = BASS_EQ_OUT[39:15];
129 MID_EQ_OUTreg = MID_EQ_OUT[39:15];
130 TREBLE_EQ_OUTreg = TREBLE_EQ_OUT[36:11];
131
132 BASS_response = BASS_EQ_OUT[39:15]; //DEBUG
133 MIDS_response = MID_EQ_OUT[39:15]; //DEBUG
134 TREB_response = TREBLE_EQ_OUT[36:11]; //DEBUG
135
136
137 if(BASS_EQ_OUTreg[23] == 0) begin // positive
138     BASS_EQ_ACCUM = BASS_EQ_OUTreg*ADC1reg; //Q24.0*Q0.8 =

```

```
139      end
140      else begin // negative
141          tempBASS = -BASS_EQ_OUTreg;
142          BASS_EQ_ACCUM = -(tempBASS*ADC1reg); //Q24.0*Q0.8 =
143              24.8
144
145
146
147      if (MID_EQ_OUTreg[23] == 0) begin // positive
148          MID_EQ_ACCUM = MID_EQ_OUTreg*ADC2reg; //Q24.0*Q0.8 =
149              24.8
150
151      end
152      else begin // negative
153          tempMIDS = -MID_EQ_OUTreg;
154          MID_EQ_ACCUM = -(tempMIDS*ADC2reg); //Q24.0*Q0.8 =
155              24.8
156
157      if (TREBLE_EQ_OUTreg[23] == 0) begin // positive
158          TREBLE_ACCUM = TREBLE_EQ_OUTreg*ADC3reg; //Q24.0*Q0.8
159              = 24.8
160
161      end
```

```
160     else begin // negative
161         tempTREB = -TREBLE_EQ_OUTreg;
162         TREBLE_ACCUM = -(tempTREB*ADC3reg); // Q24.0*Q0.8 =
163             24.8
164     end
165
166     BASS_EQ_ACCUM_DEBUG = BASS_EQ_ACCUM[31:8];
167     MID_EQ_ACCUM_DEBUG = MID_EQ_ACCUM[31:8];
168     TREBLE_ACCUM_DEBUG = TREBLE_ACCUM[31:8];
169
170
171     EQLed = PedalON;
172     if (PedalON==1) begin // Pedal ON
173         audio_out = BASS_EQ_ACCUM[31:8] + MID_EQ_ACCUM[31:8] +
174             TREBLE_ACCUM[31:8];
175     end
176     else begin // Pedal Off
177         audio_out = audio_in;
178     end
179 end
180 endmodule
```

Listing II.15: Equalizer Algorithm Verilog File

Compressor Pedal Test Bench Code

```
1 module cwb_equalizer_tb;
2 reg signed [23:0] audio_in;
3 reg sys_clk;
4 reg [7:0] I2C_ADDR;
5 reg ENABLE_I2C_data_move;
6 reg [7:0] ADC1; //Delay time
7 reg [7:0] ADC2; //Depth
8 reg [7:0] ADC3; //isFIR
9 reg [7:0] ADC4; //Not Used
10 wire signed [23:0] audio_out;
11 wire EQLed;
12
13 wire signed [23:0] BASS_response;
14 wire signed [23:0] MIDS_response;
15 wire signed [23:0] TREB_response;
16
17 wire signed [8+23:0] BASS_EQ_ACCUM_DEBUG;
18 wire signed [8+23:0] MID_EQ_ACCUM_DEBUG;
19 wire signed [8+23:0] TREBLE_ACCUM_DEBUG;
20
21
22 integer i;
23 integer j;
```

```
24 integer a;
25 integer Iter = 5;
26 integer Iter2 = 10;
27 integer Ylim = 450000;
28
29 cwb_equalizer dut(.audio_in(audio_in),
30                     .sys_clk(sys_clk),
31                     .I2C_ADDR(I2C_ADDR),
32                     .ENABLE_I2C_data_move(ENABLE_I2C_data_move),
33                     .ADC1(ADC1),
34                     .ADC2(ADC2),
35                     .ADC3(ADC3),
36                     .ADC4(ADC4),
37                     .audio_out(audio_out),
38                     .EQLed(EQLed),
39                     .BASS_response(BASS_response),
40                     .MIDS_response(MIDS_response),
41                     .TREB_response(TREB_response),
42                     .BASS_EQ_ACCUM_DEBUG(BASS_EQ_ACCUM_DEBUG),
43                     .MID_EQ_ACCUM_DEBUG(MID_EQ_ACCUM_DEBUG),
44                     .TREBLE_ACCUM_DEBUG(TREBLE_ACCUM_DEBUG));
45
46
47
48 // Define Clock Interval
```

```
49  always #5 sys_clk = ~sys_clk ;
50
51  initial
52      begin
53          sys_clk <= 0;
54          I2C_ADDR <= 8'h44 ;
55          ENABLE_I2C_data_move <= 1;
56          audio_in <= 0;
57
58          ADC1 <= 0; //0-255 Range - bass
59          ADC2 <=0; //0-255 Range - mids
60          ADC3 <=0; //0-255 Range - treble
61          //Inactive
62
63 //Low Freq
64      for(j=1;j<Iter2/2;j=j+1) begin
65          // Give some time
66          for(i=1;i<Ylim;i=i+1000) begin
67              #10 audio_in <= i ;
68      end
69
70      for(i=Ylim;i>-Ylim;i=i-1000) begin
71          #10 audio_in <= i ;
72      end
73      for(i=-Ylim;i<1;i=i+1000) begin
```

```
74          #10 audio_in <= i;  
75      end  
76  end  
77  
78  
79  ADC1 <= 128; // 0-255 Range - bass  
80  ADC2 <=0; // 0-255 Range - mids  
81  ADC3 <=0; // 0-255 Range - treble  
82  
83 //Low Freq  
84  for(j=1;j<Iter2/2;j=j+1) begin  
85      // Give some time  
86      for(i=1;i<Ylim;i=i+1000) begin  
87          #10 audio_in <= i;  
88      end  
89  
90      for(i=Ylim;i>-Ylim;i=i-1000) begin  
91          #10 audio_in <= i;  
92      end  
93      for(i=-Ylim;i<1;i=i+1000) begin  
94          #10 audio_in <= i;  
95      end  
96  end  
97  
98
```

```
99
100 //Low Mids
101   for(j=1;j<Iter2 *2;j=j+1) begin
102     // Give some time
103     for(i=1;i<Ylim;i=i+10000) begin
104       #10 audio_in <= i;
105     end
106
107     for(i=Ylim;i>-Ylim;i=i-10000) begin
108       #10 audio_in <= i;
109     end
110     for(i=-Ylim;i<1;i=i+10000) begin
111       #10 audio_in <= i;
112     end
113   end
114
115 //Med Mids
116   for(j=1;j<Iter2 *4;j=j+1) begin
117     // Give some time
118     for(i=1;i<Ylim;i=i+20000) begin
119       #10 audio_in <= i;
120     end
121
122     for(i=Ylim;i>-Ylim;i=i-20000) begin
123       #10 audio_in <= i;
```

```
124      end
125      for ( i=-Ylim ; i < 1 ; i = i + 20000 ) begin
126          #10 audio_in <= i ;
127      end
128      end
129
130 // Treble
131      for ( j = 1 ; j < Iter2 * 8 ; j = j + 1 ) begin
132          // Give some time
133          for ( i = 1 ; i < Ylim ; i = i + 30000 ) begin
134              #10 audio_in <= i ;
135          end
136
137          for ( i = Ylim ; i > -Ylim ; i = i - 30000 ) begin
138              #10 audio_in <= i ;
139          end
140          for ( i = -Ylim ; i < 1 ; i = i + 30000 ) begin
141              #10 audio_in <= i ;
142          end
143      end
144
145 // High Treble
146      for ( j = 1 ; j < Iter2 * 20 ; j = j + 1 ) begin
147          // Give some time
148          for ( i = 1 ; i < Ylim ; i = i + 60000 ) begin
```

```
149          #10 audio_in <= i;
150      end
151
152      for ( i=Ylim ; i>-Ylim ; i=i -60000) begin
153          #10 audio_in <= i;
154      end
155      for ( i=-Ylim ; i<1 ; i=i +60000) begin
156          #10 audio_in <= i;
157      end
158  end
159
160
161
162
163
164      ADC1 <= 0; //0-255 Range - bass
165      ADC2 <=255; //0-255 Range - mids
166      ADC3 <=0; //0-255 Range - treble
167 //Low Freq
168      for ( j=1;j<Iter2 /2 ; j=j +1) begin
169          // Give some time
170          for ( i=1;i<Ylim ; i=i +1000) begin
171              #10 audio_in <= i;
172          end
173
```

```
174     for ( i=Ylim ; i>-Ylim ; i=i -1000) begin
175         #10 audio_in <= i ;
176     end
177     for ( i=-Ylim ; i<1 ; i=i +1000) begin
178         #10 audio_in <= i ;
179     end
180 end
181
182 //Low Mids
183 for (j=1;j<Iter2 *2;j=j+1) begin
184     // Give some time
185     for ( i=1;i<Ylim ; i=i +10000) begin
186         #10 audio_in <= i ;
187     end
188
189     for ( i=Ylim ; i>-Ylim ; i=i -10000) begin
190         #10 audio_in <= i ;
191     end
192     for ( i=-Ylim ; i<1 ; i=i +10000) begin
193         #10 audio_in <= i ;
194     end
195 end
196
197 //Med Mids
198 for (j=1;j<Iter2 *4;j=j+1) begin
```

```
199      // Give some time
200      for ( i=1;i<Ylim ; i=i+20000) begin
201          #10 audio_in <= i ;
202      end
203
204      for ( i=Ylim ; i>-Ylim ; i=i-20000) begin
205          #10 audio_in <= i ;
206      end
207      for ( i=-Ylim ; i<1; i=i+20000) begin
208          #10 audio_in <= i ;
209      end
210  end
211
212 // Treble
213      for ( j=1;j<Iter2 *8;j=j+1) begin
214          // Give some time
215          for ( i=1;i<Ylim ; i=i+30000) begin
216              #10 audio_in <= i ;
217          end
218
219          for ( i=Ylim ; i>-Ylim ; i=i-30000) begin
220              #10 audio_in <= i ;
221          end
222          for ( i=-Ylim ; i<1; i=i+30000) begin
223              #10 audio_in <= i ;
```

```
224         end
225     end
226
227 // High Treble
228     for (j=1;j<Iter2 *20;j=j+1) begin
229         // Give some time
230         for (i=1;i<Ylim ; i=i+60000) begin
231             #10 audio_in <= i;
232         end
233
234         for (i=Ylim ; i>-Ylim ; i=i -60000) begin
235             #10 audio_in <= i;
236         end
237         for (i=-Ylim ; i <1; i=i +60000) begin
238             #10 audio_in <= i;
239         end
240     end
241
242
243
244
245
246
247     ADC1 <= 0; //0-255 Range - bass
248     ADC2 <=0; //0-255 Range - mids
```

```
249     ADC3 <=255; // 0-255 Range - treble
250     //Low Freq
251     for(j=1;j<Iter2/2;j=j+1) begin
252         // Give some time
253         for(i=1;i<Ylim;i=i+1000) begin
254             #10 audio_in <= i;
255         end
256
257         for(i=Ylim;i>-Ylim;i=i-1000) begin
258             #10 audio_in <= i;
259         end
260         for(i=-Ylim;i<1;i=i+1000) begin
261             #10 audio_in <= i;
262         end
263     end
264
265     //Low Mids
266     for(j=1;j<Iter2*2;j=j+1) begin
267         // Give some time
268         for(i=1;i<Ylim;i=i+10000) begin
269             #10 audio_in <= i;
270         end
271
272         for(i=Ylim;i>-Ylim;i=i-10000) begin
273             #10 audio_in <= i;
```

```
274         end  
275         for ( i=-Ylim ; i < 1 ; i = i + 10000 ) begin  
276             #10 audio_in <= i ;  
277         end  
278         end  
279  
280 //Med Mids  
281         for ( j = 1 ; j < Iter2 * 4 ; j = j + 1 ) begin  
282             // Give some time  
283             for ( i = 1 ; i < Ylim ; i = i + 20000 ) begin  
284                 #10 audio_in <= i ;  
285             end  
286  
287             for ( i = Ylim ; i > -Ylim ; i = i - 20000 ) begin  
288                 #10 audio_in <= i ;  
289             end  
290             for ( i = -Ylim ; i < 1 ; i = i + 20000 ) begin  
291                 #10 audio_in <= i ;  
292             end  
293         end  
294  
295 // Treble  
296         for ( j = 1 ; j < Iter2 * 8 ; j = j + 1 ) begin  
297             // Give some time  
298             for ( i = 1 ; i < Ylim ; i = i + 30000 ) begin
```

```
299          #10 audio_in <= i ;  
300      end  
301  
302      for( i=Ylim ; i>-Ylim ; i=i-30000) begin  
303          #10 audio_in <= i ;  
304      end  
305      for( i=-Ylim ; i<1 ; i=i+30000) begin  
306          #10 audio_in <= i ;  
307      end  
308  end  
309  
310 // High Treble  
311  for( j=1 ; j<Iter2 *20 ; j=j+1) begin  
312      // Give some time  
313      for( i=1 ; i<Ylim ; i=i+60000) begin  
314          #10 audio_in <= i ;  
315      end  
316  
317      for( i=Ylim ; i>-Ylim ; i=i-60000) begin  
318          #10 audio_in <= i ;  
319      end  
320      for( i=-Ylim ; i<1 ; i=i+60000) begin  
321          #10 audio_in <= i ;  
322      end  
323  end
```

```
324
325
326     ADC1 <= 128; // 0-255 Range - bass
327     ADC2 <= 128; // 0-255 Range - mids
328     ADC3 <= 128; // 0-255 Range - treble
329     //Low Freq
330     for(j=1;j<Iter2/2;j=j+1) begin
331         // Give some time
332         for(i=1;i<Ylim;i=i+1000) begin
333             #10 audio_in <= i;
334         end
335
336         for(i=Ylim;i>-Ylim;i=i-1000) begin
337             #10 audio_in <= i;
338         end
339         for(i=-Ylim;i<1;i=i+1000) begin
340             #10 audio_in <= i;
341         end
342     end
343
344     //Low Mids
345     for(j=1;j<Iter2*2;j=j+1) begin
346         // Give some time
347         for(i=1;i<Ylim;i=i+10000) begin
348             #10 audio_in <= i;
```

```
349         end

350

351     for( i=Ylim ; i>-Ylim ; i=i-10000) begin
352         #10 audio_in <= i ;
353     end
354     for( i=-Ylim ; i<1; i=i+10000) begin
355         #10 audio_in <= i ;
356     end
357 end

358

359 //Med Mids
360 for( j=1;j<Iter2 *4;j=j+1) begin
361     // Give some time
362     for( i=1;i<Ylim ; i=i+20000) begin
363         #10 audio_in <= i ;
364     end
365
366     for( i=Ylim ; i>-Ylim ; i=i-20000) begin
367         #10 audio_in <= i ;
368     end
369     for( i=-Ylim ; i<1; i=i+20000) begin
370         #10 audio_in <= i ;
371     end
372 end
373
```

```
374      // Treble
375      for (j=1;j<Iter2 *8;j=j+1) begin
376          // Give some time
377          for (i=1;i<Ylim ; i=i+30000) begin
378              #10 audio_in <= i ;
379          end
380
381          for (i=Ylim ; i>-Ylim ; i=i-30000) begin
382              #10 audio_in <= i ;
383          end
384          for (i=-Ylim ; i<1 ; i=i+30000) begin
385              #10 audio_in <= i ;
386          end
387      end
388
389      // High Treble
390      for (j=1;j<Iter2 *20;j=j+1) begin
391          // Give some time
392          for (i=1;i<Ylim ; i=i+60000) begin
393              #10 audio_in <= i ;
394          end
395
396          for (i=Ylim ; i>-Ylim ; i=i-60000) begin
397              #10 audio_in <= i ;
398          end
```

```
399      for( i=-Ylim ; i < 1 ; i=i+60000) begin
400          #10 audio_in <= i ;
401      end
402      end
403      #50 $finish ;
404  end
405 endmodule
```

Listing II.16: Equalizer Algorithm Verilog Test Bench File

Appendix III

Hardware Pinout

TI MSP430 Pinout

MSP430 Pin	Purpose	PU/PD
P6.0	I2C Address bit 0	-
P6.1	I2C Address bit 1	-
P6.2	I2C Address bit 2	-
P6.3	I2C Address bit 3	-
P2.1	I2C Address bit 0 PU	PU
P2.2	I2C Address bit 0 PU	PU
P4.0	I2C Address bit 0 PU	PU
P6.4	I2C Address bit 0 PU	PU
P1.0	On/Off Switch LED	PD
P1.1	On/Off Switch	PD
P5.0	ADCINCH8-Pot0	-
P5.1	ADCINCH9-Pot1	-
P5.2	ADCINCH10-Pot2	-
P5.3	ADCINCH11-Pot3	-
P3.1	Pot0 3V3	PU

Xilinx Zynq 7020 Pinout

```
1 #Timing Constraints
2 #Pinout constraints
3 set_property PACKAGE_PIN AB1 [get_ports {CODEC_addr0[0]}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {CODEC_addr0[0]}]
5 set_property PACKAGE_PIN Y5 [get_ports {CODEC_addr1[0]}]
6 set_property IOSTANDARD LVCMOS33 [get_ports {CODEC_addr1[0]}]
7 set_property PACKAGE_PIN AB2 [get_ports FCLK_CLK1_0]
8 set_property IOSTANDARD LVCMOS33 [get_ports FCLK_CLK1_0]
9 set_property PACKAGE_PIN AA6 [get_ports BCLK_0]
10 set_property IOSTANDARD LVCMOS33 [get_ports BCLK_0]
11 set_property PACKAGE_PIN Y6 [get_ports LRCLK_0]
12 set_property IOSTANDARD LVCMOS33 [get_ports LRCLK_0]
13 set_property PACKAGE_PIN Y8 [get_ports SDATA_O_0]
14 set_property IOSTANDARD LVCMOS33 [get_ports SDATA_O_0]
15 set_property PACKAGE_PIN AA7 [get_ports SDATA_I_0]
16 set_property IOSTANDARD LVCMOS33 [get_ports SDATA_I_0]
17 set_property PACKAGE_PIN AB5 [get_ports IIC_0_0_sda_io]
18 set_property IOSTANDARD LVCMOS33 [get_ports IIC_0_0_sda_io]
19 set_property PACKAGE_PIN AB4 [get_ports IIC_0_0_scl_io]
20 set_property IOSTANDARD LVCMOS33 [get_ports IIC_0_0_scl_io]
21 #set_property PACKAGE_PIN F22 [get_ports {Switches[0]}]
22 #set_property IOSTANDARD LVCMOS25 [get_ports {Switches[0]}]
23 #set_property PACKAGE_PIN G22 [get_ports {Switches[1]}]
```

```
24 #set_property IOSTANDARD LVCMOS25 [get_ports {Switches[1]}]
25 #set_property PACKAGE_PIN H22 [get_ports {Switches[2]}]
26 #set_property IOSTANDARD LVCMOS25 [get_ports {Switches[2]}]
27 #set_property PACKAGE_PIN F21 [get_ports {Switches[3]}]
28 #set_property IOSTANDARD LVCMOS25 [get_ports {Switches[3]}]
29 #set_property PACKAGE_PIN H19 [get_ports {Switches[4]}]
30 #set_property IOSTANDARD LVCMOS25 [get_ports {Switches[4]}]
31 #set_property PACKAGE_PIN H18 [get_ports {Switches[5]}]
32 #set_property IOSTANDARD LVCMOS25 [get_ports {Switches[5]}]
33 #set_property PACKAGE_PIN H17 [get_ports {Switches[6]}]
34 #set_property IOSTANDARD LVCMOS25 [get_ports {Switches[6]}]
35 #set_property PACKAGE_PIN M15 [get_ports {Switches[7]}]
36 #set_property IOSTANDARD LVCMOS25 [get_ports {Switches[7]}]
37 set_property PACKAGE_PIN T22 [get_ports {LEDS[0]}]
38 set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[0]}]
39 set_property PACKAGE_PIN T21 [get_ports {LEDS[1]}]
40 set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[1]}]
41 set_property PACKAGE_PIN U22 [get_ports {LEDS[2]}]
42 set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[2]}]
43 set_property PACKAGE_PIN U21 [get_ports {LEDS[3]}]
44 set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[3]}]
45 set_property PACKAGE_PIN V22 [get_ports {LEDS[4]}]
46 set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[4]}]
47 set_property PACKAGE_PIN W22 [get_ports {LEDS[5]}]
48 set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[5]}]
```

```
49 set_property PACKAGE_PIN U19 [get_ports {LEDS[6]}]
50 set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[6]}]
51 set_property PACKAGE_PIN U14 [get_ports {LEDS[7]}]
52 set_property IOSTANDARD LVCMOS33 [get_ports {LEDS[7]}]
```

Listing III.1: Xilinx Zynq 7020 XDC Pinout

Appendix IV

MSP430 C Code

```
1 /* --COPYRIGHT--,BSD_EX
2 *
3 */
4 #include <msp430.h>
5 #include <msp430fr2355.h>
6 #include <INTRINSICS.H>
7
8 // Declaration of Variables
9 unsigned int IOswitch = 0;           //On off switch that enables
                                         pedal data collection
10 unsigned int checkInputs = 0;
11 unsigned int DATA[4] = {0,0,0,0};      //Array that holds data
                                         collected from ADC conversions
12 unsigned int DATATX[4] = {0,0,0,0};     //Array containing
                                         enable bit and data collected from the ADC
```

```
13 unsigned int TXDataIndex = 0;
14 unsigned int ADCchannel[4] = {ADCINCH_8, ADCINCH_9, ADCINCH_10
   , ADCINCH_11 };
15 unsigned int DATAindex = 0; // Index through the data array
16 unsigned int ADC_SW_FLAG = 0; // Used to let us know when
   sequential ADC channel sampling is completed one round
17 unsigned const BASE_I2C_ADDR = 0x40;
18 unsigned int OFFSET_I2C_ADDR = 0x00;
19 unsigned int I2C_communication_Not_Happen = 1;
20
21 // Function Definitions
22 void IsPedalEnabled(void);
23 void ADCcollect(void);
24 void AcquireData(void);
25
26 int main(void)
27 {
28     WDTCTL = WDTPW + WDTHOLD;
29
30     // Configure GPIO for I2C
31     P1SEL0 |= BIT2 | BIT3; // I2C pins
32
33     //
```

```
34 // Set up the port to read the ON OFF button and
35 // Toggle onboard LED which will later be led in button
36 P1DIR = 0xF1;
37 //P1DIR |= BIT7 | BIT6 | BIT5| BIT4 | BIT1; // Bitwise or -
38 // PINS P1.1-3 are inputs and the rest are outputs
39 //P1DIR = BIT7 | BIT6 | BIT5| BIT4 | BIT1;
40 P1OUT = 0x0000; // Bit wise AND with 0 - all OUTPUT LOW
41 P1REN = BIT1; // Pull down P1.1 line
42
43
44
45
46 // Initilize and set up the ADC for analog voltages - dir
47 // dont cares
48 //P5DIR |= 0xFF; // Bitwise or - set all pins to inputs
49 //P5OUT &= ~0xFF; // Bit wise AND with 0 - all OUTPUT LOW
50 P5SEL0 = 1; // This may or may not set up the pin in analog
51 // mode
52 P5SEL1 = 1; // This may or may not set up the pin in analog
53 // mode
54 // Double check the above with carlos
```

```
54 // Initilize the pins that will be used to set the I2C
      offset address value
55 //P2.1 ,P2.2 ,P4.0 ,P6.4 pull up high
56 //P6.0-P6.3 will be set to read valuesas inputs pulled low
57 //
*****
```



```
58 P6DIR = BIT4; // Bitwise or - set lowest pins to inputs
      with P6.4 as output
59 P6OUT = BIT4; // Bit wise AND with 0 - all OUTPUT LOW
      except P6.4
60 P6REN = BIT4; // Pull up P1.1 line
61
62 P4DIR = BIT0; // Bitwise or - set lowest pins to inputs
      with P4.0 as output
63 P4OUT = BIT0; // Bit wise AND with 0 - all OUTPUT LOW
      except P4.0
64 P4REN = BIT0; // Pull up P4.0 line
65
66 P2DIR = BIT2|BIT1; // Bitwise or - set lowest pins to
      inputs with P2.1 and 2.2 as output
67 P2OUT = BIT2|BIT1; // Bit wise AND with 0 - all OUTPUT LOW
      except P2.1 and 2.2
68 P2REN = BIT2|BIT1; // Pull up P2.1 and 2.2 line
69
```

```
70 // Disable the GPIO power-on default high-impedance mode to
    activate
71 // previously configured port settings
72 PM5CTL0 &= ~LOCKLPM5;
73
74
75 __bis_SR_register(GIE);      // enable interrupts
76
77
78 // Get the initial conditions of the potentiometers from the
    pedals - most likely will be off on power up
79 AcquireData(); // -CWB
80
81
82 /*Get the address of the I2C device by using jumpers on the
    right pin bank
83 * Offset address set by the values P6.3 ,P6.2 ,P6.1 ,P6.0
84 * The above pins are pulled low and default to 4'b0000
85 * P6.3 pulled up if shunted to P6.4 - 4'b1000
86 * P6.2 pulled up if shunted to P4.0 - 4'b0100
87 * P6.1 pulled up if shunted to P2.2 - 4'b0010
88 * P6.0 pulled up if shunted to P2.1 - 4'b0001
89 */
90
91 // Get the Base_offset value
```

```
92     OFFSET_I2C_ADDR = P6IN & 0x0F;  
93  
94 // Configure USCI_B0 for I2C mode  
-----  
95 UCB0CTLW0 = UCSWRST;                      // Software reset  
    enabled  
96 UCB0CTLW0 |= UCMODE_3 | UCSYNC;           // I2C mode, sync  
    mode  
97 UCB0I2COA0 = (BASE_I2C_ADDR+OFFSET_I2C_ADDR) | UCOAEN;  
    // own address is 0x48 + enable  
98 //UCB0I2COA0 = 0x48 | UCOAEN;              // own address  
    is 0x48 + enable  
99 UCB0CTLW0 &= ~UCSWRST;                   // clear reset  
    register  
100 UCB0IE |= UCTXIE0 | UCSTPIE;            // I2C transmit,  
    stop interrupt enable  
101  
102  
103  
104 //  
    ////////////////////////////////  
105  
106 // Wait for I2C communication
```

```

107 //Once communication finished
108         // Collect ADC values to update
109     while(1){ //Continue this process forever
110         while(I2C_communication_Not_Happen){
111             // wait for I2C communication to happen
112         }
113         // Disable I2C ints
114         UCB0IE = 0x00;           // I2C transmit ,stop
115                     interrupt DISABLE
116
117         // Update ADC values
118         AcquireData();          // Update data from ADCs
119
120         //Re enable I2C and look for comms
121         UCB0IE |= UCTXIE0 | UCSTPIE; // I2C transmit ,stop
122                     interrupt ENABLE
123         I2C_communication_Not_Happen = 1;
124
125     }
126 //
```

```
127 //  
128  
129 void AcquireData(void){  
130     //Check to see if the Enable Pin is high or low  
131     //Need to fix enablePIN bc checking P1.0 but also then  
132         writing out? BAD  
133  
134     //IOswitch = 1; //Uncheck this for debugging the switch  
135         read capability  
136     if (IOswitch == 1){  
137         ADCcollect();  
138         //Write data to single vector after all data is  
139             collected to prevent interrupts from getting  
140                 partial data  
141             DATATX[0] = DATA[0] >> 4;  
142             DATATX[1] = DATA[1] >> 4;  
143             DATATX[2] = DATA[2] >> 4;  
144             DATATX[3] = DATA[3] >> 4;  
145         }  
146     else {  
147         DATATX[0] = 0;
```

```
146         DATATX[ 1 ] = 0;  
147         DATATX[ 2 ] = 0;  
148         DATATX[ 3 ] = 0;  
149         DATATX[ 4 ] = 0;  
150  
151         //DATATX = { 0 ,0 ,0 ,0 };  
152     }  
153 }  
154 //  
//////////////////////////////////////////////////////////////////  
  
155 void IsPedalEnabled( void ){  
156     //Check to see if the button was pushed  
157     //Assuming that the design uses a latching switch  
158     //Pull down is enabled on pin to keep from floating so  
     //will  
159     //be low unless otherwise triggered  
160  
161     //For digital IO bit of Port.Pin is 0 if low and 1 if high  
162     //Scan Port 1.1 to see if input is high  
163  
164     //      P1DIR |= 0x01; //Bitwise or - set all pins to inputs  
     //      except P1.0 -output  
165     //      P1OUT &= ~0xFF;// Bit wise AND with 0 - all OUTPUT  
     //      LOW
```



```
186 //ADCSHT =1011 -768 CLKcycles ; ADCMSC = 0 -single
     smaples
187 //ADCON = 1 - ADC is ON ; ADCENC = 0 - dont start
     conversions
188 //ADCSC 0 no sample and conversion start
189 //
     /////////////////////////////////
190 ADCCTL0 = (ADCSHT_11 | ADCON_1);
191
192 // Set up the ADCCTL1 Register
     ///////////////////////////////
193 //ADCSHS = 0-ADCSC bit;ADCSHP=1-SAMPCON from timer;
194 //ADCISSH=0-noninverted;ADCDIV=111-divide by 8;
195 //ADCSSEL=0-MODCLK;ADCCONSEQ=0-single sample;ADCBUSY=0
196 //
     ///////////////////////////////
197 ADCCTL1 = (ADCSHP_1 | ADCDIV_7);
198
199 // Set up the ADCCTL2 Register
     ///////////////////////////////
200 //ADCPDIV = 0 -predivide; ADCRES = 10 - 12 bits
201 //ADCDF = 0 -binary unsigned; ADCSR = 0 -200ksps
```

```
202      //  
203      ///////////////////////////////////////////////////////////////////  
204  
205      ADCCTL2 = (ADCRES_2);  
206      // Set up the ADCMCTL0 Register  
207      ///////////////////////////////////////////////////////////////////  
208      //ADCSREF = 000 - Vcc and Vss; ADCINCH = 1011 to 1000  
209      // -Channel inputs we need to test  
210      ///////////////////////////////////////////////////////////////////  
211      ADCMCTL0 = (ADCchannel[DATAindex]);  
212  
213      // Start the process of collecting the ADC channel  
214      // inputs  
215      // __bis_SR_register(GIE);      // enable interrupts  
216  
217      ADCIE = 0x01; // Enable Interrupts for when the  
218      // ADC conversion is implemented  
219  
220      ADC_SW_FLAG =0;          // Clear ADC SW FLAG  
221  
222      ADCCTL0 |= (ADCENC_1 | ADCSC_1); // bis.w #(  
223      // ENC + ADC10SC) , &ADC10CTL0 // Start a
```



```
235 // Interrupt Table
236 #pragma vector = ADC_VECTOR
237 __interrupt void ADC12_ISR(void) {
238     // ".int29"
239     // ADC12_ISR CODE HERE
240     ADCCTL0 &= ~ADCIFG0;      // Remove the SW Flag from the
241     registers
242     DATA[DATAindex] = 0;        // Clear any old values in the
243     DATA
244     DATA[DATAindex] = ADCMEM0;   // Move value from ADC
245     conversion to data
246     DATAindex++;
247     ADC_SW_FLAG = 1;           // Set a flag to show ADC
248     completed
249     /////////////////////////////////
250 // TI interrupt vector for the I2C
251 #if defined(__TI_COMPILER_VERSION__) || defined(
252     __IAR_SYSTEMS_ICC__)
253 #pragma vector = USCI_B0_VECTOR
```

```
253 __interrupt void USCIB0_ISR(void)
254 #elif defined(__GNUC__)
255 void __attribute__((interrupt(USCI_B0_VECTOR))) USCIB0_ISR (
256     void)
257 #else
258 #error Compiler not supported!
259 {
260     switch (__even_in_range(UCB0IV, USCI_I2C_UCBIT9IFG))
261     {
262         case USCI_NONE: break; // Vector 0: No
263             interrupts
264         case USCI_I2C_UCALIFG: break; // Vector 2: ALIFG
265         case USCI_I2C_UCNACKIFG: break; // Vector 4:
266             NACKIFG
267         ///////////////////////////////////////////////////////////////////
268         case USCI_I2C_UCSTTIFG:
269             break; // Vector 6: STTIFG
270         ///////////////////////////////////////////////////////////////////
271         case USCI_I2C_UCSTPIFG: // Vector 8:
272             STPIFG
273     }
```

```
270     UCB0IFG &= ~UCSTPIFG; // Clear stop  
271     condition int flag  
272     TXDataIndex++; // Increase index  
273     if (TXDataIndex % 4 == 0) { // All ADC values  
274         (4) have been sent  
275         TXDataIndex=0; // Reset TX index  
276         I2C_communication_Non_Happen = 0;  
277     }  
278     break;  
279 ///////////////////////////////////////////////////////////////////  
280  
281     case USCI_I2C_UCRXIFG3: break; // Vector 10:  
282         RXIFG3  
283     case USCI_I2C_UCTXIFG3: break; // Vector 14:  
284         TXIFG3  
285     case USCI_I2C_UCRXIFG2: break; // Vector 16:  
286         RXIFG2  
287     case USCI_I2C_UCTXIFG2: break; // Vector 18:  
288         TXIFG2  
289     case USCI_I2C_UCRXIFG1: break; // Vector 20:  
290         RXIFG1  
291     case USCI_I2C_UCTXIFG1: break; // Vector 22:  
292         TXIFG1
```

```
285     case USCI_I2C_UCRXIFG0: break;           // Vector 24:  
286     //  
287     ///////////////////////////////////////////////////////////////////  
288  
289     case USCI_I2C_UCTXIFG0:  
290         UCB0TXBUF = DATATX[ TXDataIndex ];  
291         break;                           // Vector 26:  
292         TXIFG0  
293     //  
294     ///////////////////////////////////////////////////////////////////  
295  
296     case USCI_I2C_UCBCNTIFG: break;           // Vector 28:  
297     BCNTIFG  
298     case USCI_I2C_UCCLTOIFG: break;           // Vector 30:  
299         clock low timeout  
300     case USCI_I2C_UCBIT9IFG: break;           // Vector 32: 9th  
301         bit  
302     default: break;  
303 }  
304 }
```

Listing IV.1: MSP430 I2C Slave Interrupt C File

Appendix V

MATLAB Equalizer Filter Generation

The MATLAB filterBuilder UI was used in order to create the low-pass, band-pass, and high-pass filters needed to implement the equalizer effect. This appendix documents the UI window output constrains as well as the script files to generate the filter which can then be exported to HDL.

```
1 function Hd = LPF_func_return_v4
2 %GETFILTER Returns a discrete-time filter object.
3
4 % MATLAB Code
5 % Generated by MATLAB(R) 9.7 and DSP System Toolbox 9.9.
6 % Generated on: 22-Feb-2020 12:04:35
7
8 Fpass = 150;      % Passband Frequency
9 Fstop = 600;      % Stopband Frequency
10 Apass = 1;        % Passband Ripple (dB)
11 Astop = 20;       % Stopband Attenuation (dB)
```

```

12 Fs      = 48000; % Sampling Frequency
13
14 h = fdesign.lowpass( 'fp ,fst ,ap ,ast' , Fpass , Fstop , Apass ,
15           Astop , Fs );
16 Hd = design(h, 'equiripple' , ...
17           'FilterStructure' , 'dfsymfir' , ...
18           'MinOrder' , 'any' , ...
19           'StopbandShape' , 'flat' );
20
21 set(Hd, 'Arithmetic' , 'fixed' , ...
22       'InputWordLength' , 24 , ...
23       'InputFracLength' , 0 , ...
24       'CoeffWordLength' , 12 , ...
25       'CoeffAutoScale' , true , ...
26       'FilterInternals' , 'Fullprecision' );

```

Listing V.1: Lowpass EQ Bass Filter

```

1 function Hd = BPF_func_return_v1
2 %GETFILTER Returns a discrete-time filter object.
3
4 % MATLAB Code
5 % Generated by MATLAB(R) 9.7 and DSP System Toolbox 9.9.
6 % Generated on: 22-Feb-2020 13:30:40
7

```

```
8 Fstop1 = 1;           % First Stopband Frequency
9 Fpass1 = 600;          % First Passband Frequency
10 Fpass2 = 800;          % Second Passband Frequency
11 Fstop2 = 1600;         % Second Stopband Frequency
12 Astop1 = 20;           % First Stopband Attenuation (dB)
13 Apass = 0.1;           % Passband Ripple (dB)
14 Astop2 = 20;           % Second Stopband Attenuation (dB)
15 Fs      = 48000;        % Sampling Frequency
16
17 h = fdesign.bandpass('fst1',fp1,fp2,fst2,ast1,ap,ast2', Fstop1,
18 Fpass1, ...
19 Fpass2, Fstop2, Astop1, Apass, Astop2, Fs);
20
21 Hd = design(h, 'equiripple', ...
22 'FilterStructure', 'dfsymfir', ...
23 'MinOrder', 'any');
24
25 set(Hd, 'Arithmetic', 'fixed', ...
26 'InputWordLength', 24, ...
27 'InputFracLength', 0, ...
28 'CoeffWordLength', 12, ...
29 'CoeffAutoScale', true, ...
30 'FilterInternals', 'Fullprecision');
```

Listing V.2: Bandpass EQ Mids Filter

```
1 function Hd = HPF_func_return_v1
2 %GETFILTER Returns a discrete-time filter object.
3
4 % MATLAB Code
5 % Generated by MATLAB(R) 9.7 and DSP System Toolbox 9.9.
6 % Generated on: 22-Feb-2020 12:37:28
7
8 Fstop = 800;      % Stopband Frequency
9 Fpass = 1600;     % Passband Frequency
10 Astop = 20;       % Stopband Attenuation (dB)
11 Apass = 0.1;      % Passband Ripple (dB)
12 Fs      = 48000;   % Sampling Frequency
13
14 h = fdesign.highpass('fst',fp,ast,ap, Fstop, Fpass, Astop,
15 Apass, Fs);
16
17 Hd = design(h, 'equiripple', ...
18     'FilterStructure', 'dfsymfir', ...
19     'MinOrder', 'any', ...
20     'StopbandShape', 'flat');
21
22 set(Hd, 'Arithmetic', 'fixed', ...
23     'InputWordLength', 24, ...
24     'InputFracLength', 0, ...
```

```
24      'CoeffWordLength' , 12 , ...
25      'CoeffAutoScale' , true , ...
26      'FilterInternals' , 'Fullprecision' );
```

Listing V.3: Highpass EQ Treble Filter

```
1 %3 Band EQ design
2 %C Blasie
3 %22 Feb 2020
4 %
5 %Use MATLAB FIR HDL filterBuilder to make FIR filters for a 3
   badn EQ
6 %guitar effect
7
8 %Call in the filters
9 BASS = LPF_func_return_v4 ;
10 MIDS = BPF_func_return_v1 ;
11 TREB =HPF_func_return_v1 ;
12
13 fvtool(BASS,MIDS,TREB) ;
```

Listing V.4: EQ filter overlay

Appendix VI

Waveform Signals

Chapter 7 of this project deals with the simulation of the effects on hardware using the AD2. Each of the waveforms used for simulation are presented below at the algorithm level as well as the parameters needed to recreate the waveform in another environment. The WaveForms software used to control the AD2 allows users to create custom waveforms with math equations. This option involves describing a mathematical input function over a number of samples in a range from zero to one that repeats every period. Each waveform is documented in the mathematical formula used to generate the input signal, the number of samples, period, the frequency, and amplitude of the input signal.

LFM_cwb

This signal was used in initial pass through testing to verify that the input signal to the system and output from the signal processing algorithms was of minimal latency.

$$y(x) = \sin(20 * \pi * x^2) \quad (\text{VI.1})$$

Table VI.1: LFM_cwb Waveform parameters

Parameter	Samples	Period [ms]	Freq [Hz]	Amplitude [mV]
Value	16384	20	50	100

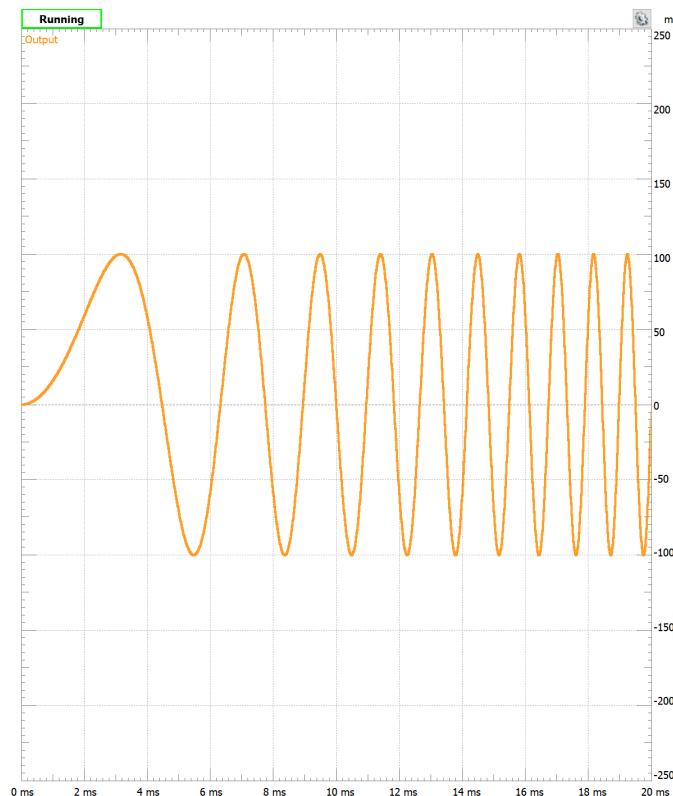


Figure VI.1: LFM_cwb Waveform

Table VI.2: LFM_long Waveform parameters

Parameter	Samples	Period [ms]	Freq [Hz]	Amplitude [mV]
Value	6535	20	50	100

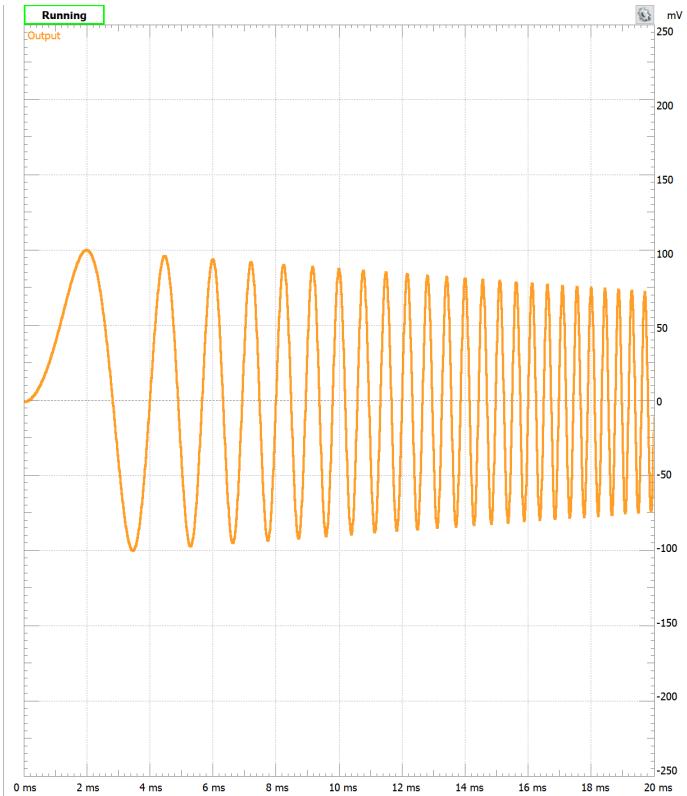


Figure VI.2: LFM_long Waveform

LFM_Long

This signal was created to have a relatively flat harmonic spectrum in order to test the equalizer effect. The LFM_Long signal was created to have a decay associated with the LFM_cwb signal in order to create a flatter spectrum response.

$$y(x) = (1 - .3x) * \sin(50 * \pi * x^2) \quad (\text{VI.2})$$

Table VI.3: Sinusoid Waveform parameters

Parameter	Samples	Period [ms]	Freq [Hz]	Amplitude [mV]
Value	4096	10	100	100

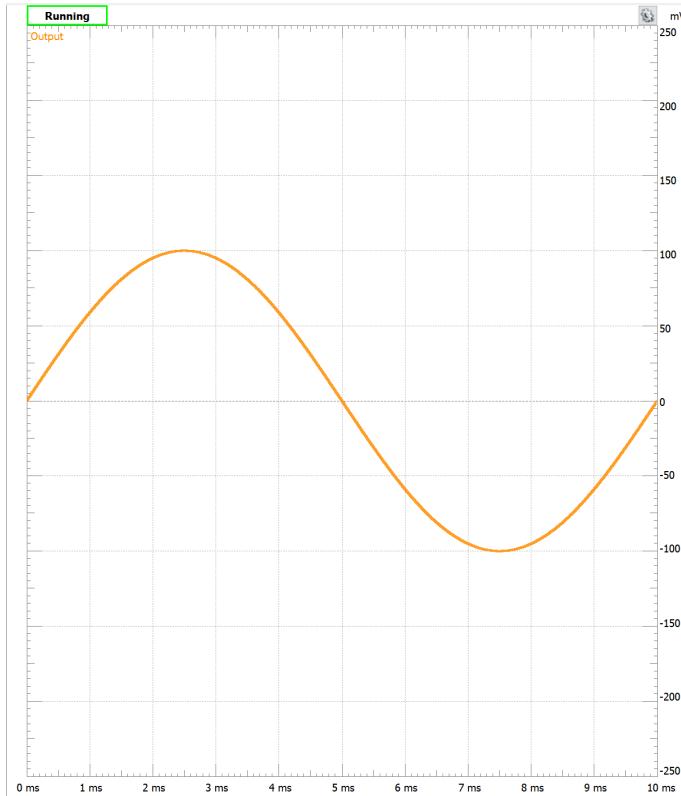


Figure VI.3: Sinusoid Waveform

Sinusoid

The sinusoid signal was used in order to verify the performance of the gain, overdrive, distortion algorithm.

$$y(x) = \sin(2 * \pi * x) \quad (\text{VI.3})$$

Table VI.4: Pulsed Sinusoid Waveform parameters

Parameter	Samples	Period [ms]	Freq [Hz]	Amplitude [mV]
Value	29999	100	10	100

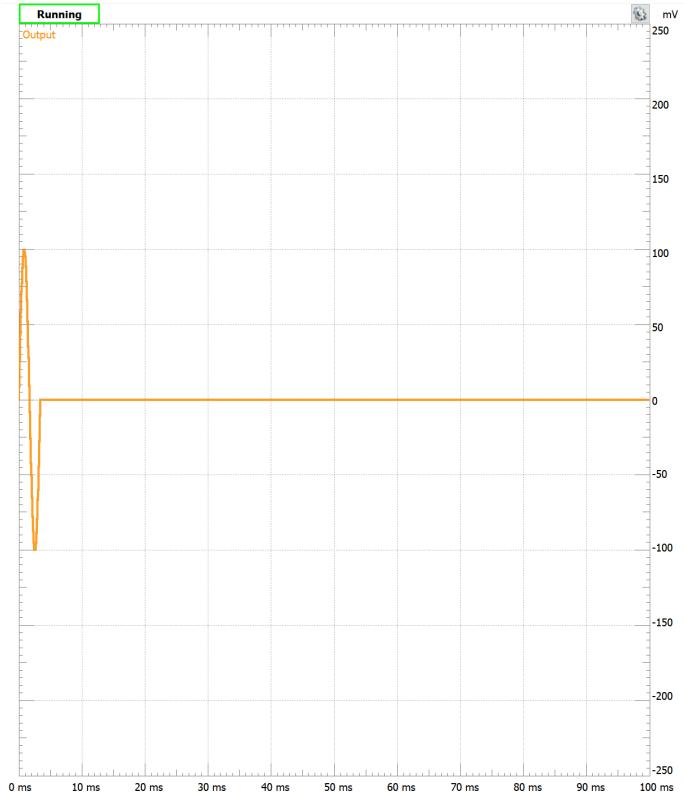


Figure VI.4: Pulsed Sinusoid Waveform

Pulsed Sinusoid

The pulsed sinusoid signal was used in order to verify the performance of the delay algorithm. This pulse was first created by using the equation of the sinusoid signal for 1028 samples. The signal was then zero padded until the total signal length became 29999 samples.