The Iby and Aladar Fleischman Faculty of Engineering
Tel Aviv University

FPGA Design and Implementation of Electric Guitar Audio Effects

Project Report

By:

Vladi Litmanovich

Adi Mikler

Advisor: Jacob Fainguelernt

Project Carried Out at Tel Aviv University

Video is available at: https://youtu.be/ANxHyCAYGp0

Abstract

This is our senior-year Electrical Engineering project at Tel-Aviv University.

The goal of the project was to design and implement audio-processing algorithms on an FPGA device, to create a platform for electric guitar audio effects. This is known in the music industry as a "Multi-effects pedal".

As opposed to the commercial devices which mostly rely on DSP and software cores, we wanted to migrate the entire system into an FPGA logic, and test the benefits it provides to this kind of system.

The project was carried out on the Zedboard - a development board for the Xilinx Zynq-7000 all programmable SoC (System-on-Chip), which comprises both a Dual ARM-A9 processing system (PS), and an FPGA programmable logic. The combination of the FPGA fabric, ARM PS, and various physical interfaces on the board itself, allowed us to create a full Multi-effect system, providing both audio-processing and user interface, on a single board.

The effects we created were: Distortion, Octavelo (an experimental effect which combines Octaver and Tremolo), Tremolo, and Delay. Each effect block has several internal settings from which the user can choose, and it is also possible to concatenate several effects in series, to produce interesting combinations.

The results were satisfying and very engaging for us and for other guitar players who tried our system, as some of the effects are well known, and some provide new and unique sound qualities.

The real-time performance of the system was also satisfactory, with a maximal latency of around 1ms, which is better than some leading commercial pedals which state their latency to be "several milliseconds".

Table of Contents

1. Introduction

1.1 A Brief History

Guitar effects have been around for almost as long as the electric guitar itself. The first standalone effect unit (called "stompbox", or "pedal") was built in 1948, and it was a Tremolo pedal. Since then countless effects were designed, created, re-created, modified and improved, taking the music industry by storm, creating a fertile ground for the development of different music genres, including the good old Rock 'n' Roll.

These stompboxes were (and many still are) analog – they consisted of discrete analog components, first vacuum-tubes and later transistors, and could create high quality audio effects, some of which we try to emulate to this very day by digital means.

In the 1980's the first digitized units started to appear, taking advantage of integrated circuits (IC's) and combining several effects units within a single unit. As digital technology increased in quality in the 1990's, better tools allowed more freedom and ease for engineers to experiment with different audio effects, eventually creating the all-in-one software based Multi-effect units which we can see today.

1.2 Today's Market

Today's Multi-effect units resemble computers rather than stompboxes, with powerful software-driven DSP cores, allowing the creation of audio effects which were impossible 50 years ago.

Some of the effects, such as Delay and various Modulation effects which are mathematically speaking- linear, immediately became more robust and versatile when implemented in the digital domain. Non-linear effects such as Overdrive and Distortion, required a little bit more time for digital modeling technology to improve to resemble their analog counterparts, but today we can find commercial high quality digital effects which will fool even the most sensitive of ears.

Generally speaking, digital effects can be divided into 3 categories:

1. Software-only solutions such as Garage-Band, Bias-FX, or Guitar Rig, which can run on a computer or a mobile device (usually requiring external audio interface to connect the guitar to the computer) where the audio processing itself is done on the device's CPU.
2. Built-in effect units inside a guitar amplifier - almost all contemporary amplifiers are equipped with some degree of digital effects.
3. Physical Multi-effects units (also called pedals) from vendors such as Zoom, Boss and DigiTech, with a guitar input and amplifier output, a local DSP core, and a physical UI (knobs and stomp-buttons) for live performances and easy on-the-fly control.

These solutions are based on a chip with a hard architecture that cannot be changed (be it a CPU, DSP, or a simple microcontroller) but can be programmed to implement various effects.

Lately, a fourth category is starting to emerge into the commercial world of digital guitar effects, and it's the FPGA-based signal processing.

1.3 Why FPGA

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks connected via programmable interconnects. Straight from the manufacturer, they come with no hardcoded architecture, and resemble a blank "field of logical gates" (hence the name) which can be reprogrammed according to the desired application. Although one-time programmable FPGAs are available, the dominant types are those which can be reprogrammed as the design evolves.

When programmed, connections are formed between the various logic blocks, which can implement every logical function, and even implement a CPU or DSP in the FPGA fabric (since they are all essentially made from the same logic gates).

An FPGA design has an inherent parallel nature. Since it is a direct implementation of a digital circuit, the signal can propagate through the entire system like in a circuit, allowing many audio channels to be processed together, improving real-time performances.

Modern FPGAs provide a cost-effective and flexible alternative to other application-specific products, allowing engineers to implement system-on-chip designs that eliminate the need for separate components to perform audio processing tasks, thereby reducing costs, particularly for multi-channel audio applications.

1.4 Project Goals

In this project, we wanted to experiment with the concept of implementing audio-processing algorithms on an FPGA, instead of the common CPU/DSP-based platforms.

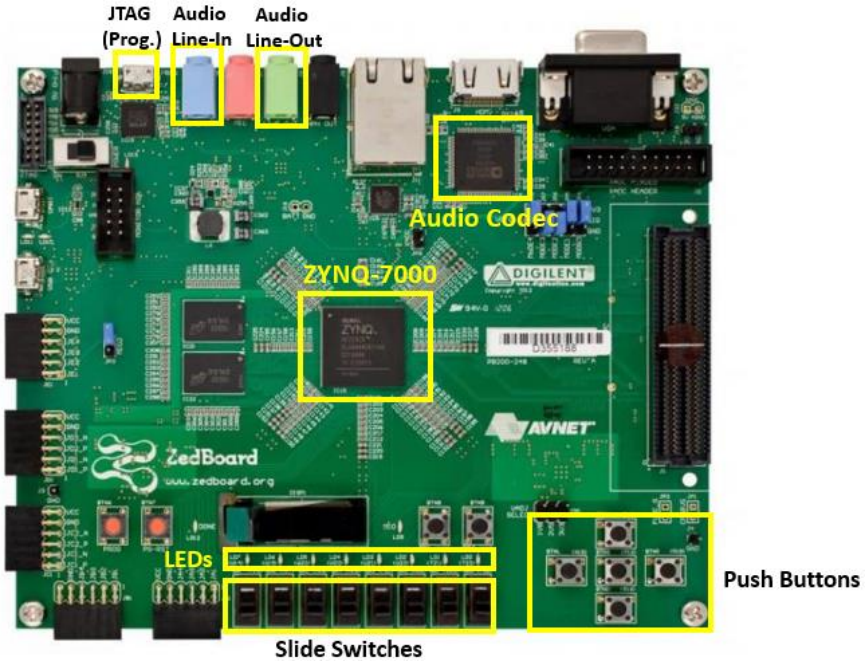
This approach is not wide spread in the guitar-effects industry as of 2017 - we found only one commercial vendor (Antelope Audio) which integrated an FPGA logic into their audio effects platform, as well as some other projects done by private individuals around the world.

Our main goals are:

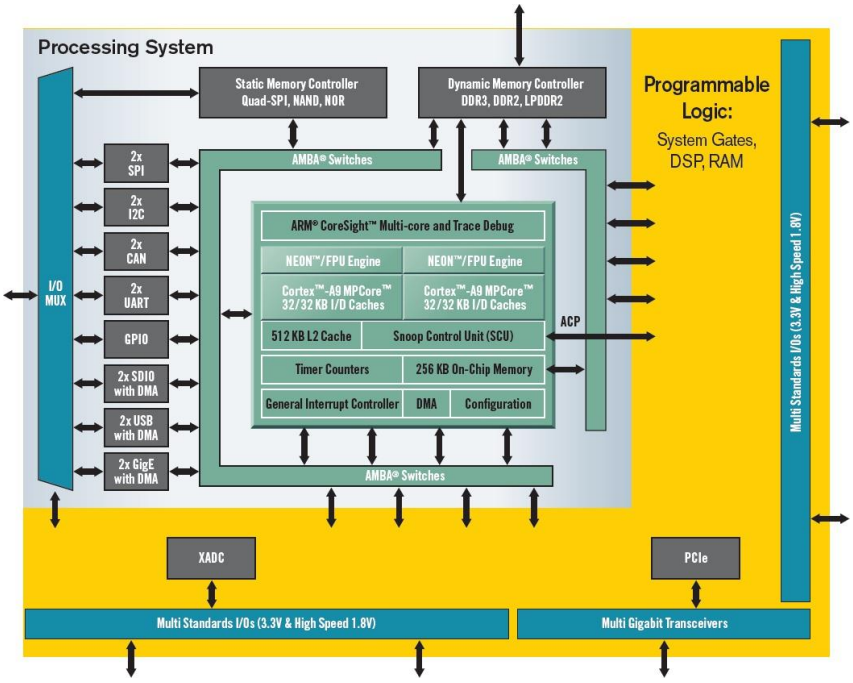
- Create an FPGA audio-processing platform and implement effects for the electric guitar, with a design resembling that of commercial physical Multi-effect pedals.
- Obtain knowledge and experience in VHDL and FPGA development in general, as this field grows in popularity in recent years.
- Put our engineering skills and effort to create something practical which we will enjoy using.

1.5 Platform and Tools

We used the Zedboard Zynq-7000 ARM/FPGA SoC Development Board - a development board for the Xilinx Zynq-7000 all programmable SoC (System-on-Chip), which comprises an ARM processing system (PS) and an FPGA programmable logic (PL). The Zedboard is equipped with a high-quality Audio-Codec, 3.5mm audio jacks and an abundance of buttons, switches, and LEDs, which serves all the needs of our project, in a single board.



The Zedboard – in yellow are the components we utilized – the Zynq-7000 and the audio codec as the heart of the system, Line-in and Line-out connections for the guitar and amplifier respectively, JTAG USB port for programming, and switches, buttons and LEDs for user interface.



The internal architecture of the Zynq-7000. In the middle is the ARM processor, interconnected with FPGA PL (orange), and surrounded by various I/O interfaces

Software-

- The main working environment we used was Vivado 2016.2 – in which we created various IP (Intellectual Property) blocks for the audio signal path, effects, user interface and PS-PL connectivity – all done manually in VHDL.
- Since the embedded ARM processor is also a part of the design, we used Xilinx SDK 2016.2 (programmed in C), to read and write audio samples to/from the PL via AXI functions, and initialize the on-board Audio Codec.
- Prior to VHDL implementation, we used MATLAB to run simulations and test our design.

Hardware-

Apart from the Zedboard, electric guitar and amplifier, we also used an Agilent InfiniiVision-3034A digital oscilloscope, and an Agilent 33120A signal generator for exact measurements and performance validation.

2. Project Overview

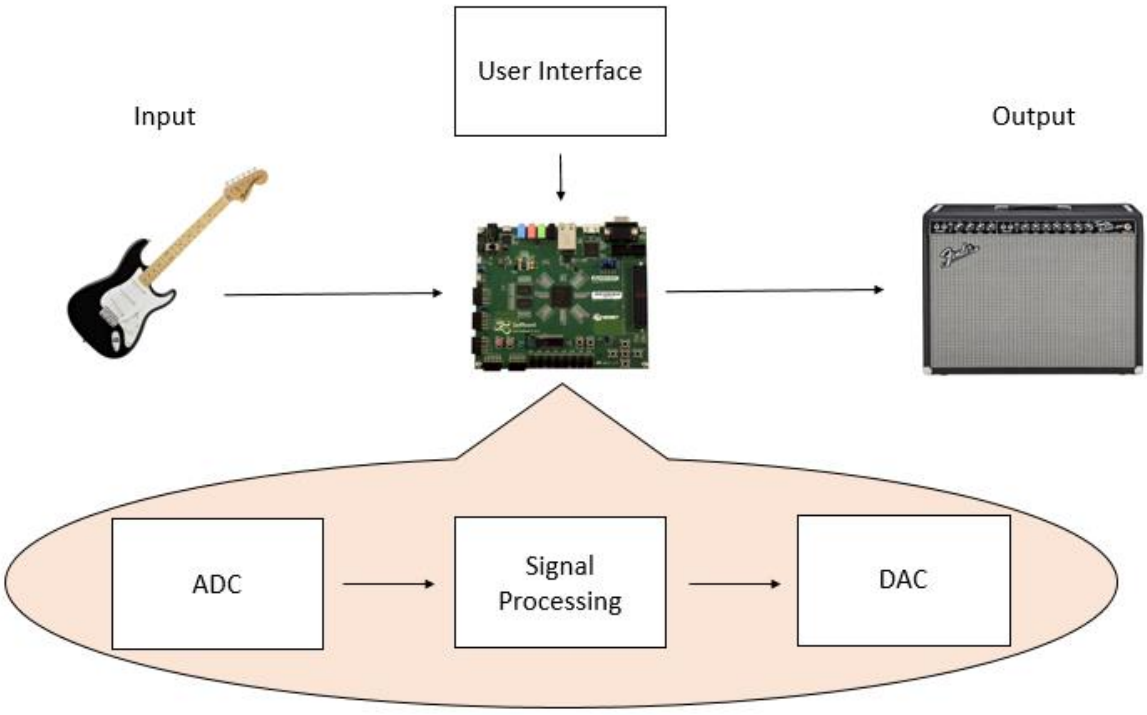
2.1 General

Since the goal of the project was to test the ability of FPGA (PL) as the platform for effects implementation, the embedded ARM processor (PS), was used for the general infrastructure and support.

Although it's possible to create an identical system without using the PS at all, the Zynq comes with many convenient C functions which allowed us to easily establish I2C communication with the Audio Codec and configure its various registers. Moreover, the PS provides an easy control over AXI peripherals, which will be discussed later in this paper.

This combination of PS and PL allowed us to invest most of our time on the core VHDL design of the effects and other supporting IPs, instead of worrying about the need to implement complex protocols such as I2C, I2S, and AXI from scratch.

The general (and simplified) block diagram of the system looks like this:



The guitar is connected to the Line-in 3.5mm jack (blue) and Line-out jack (green) is connected to the amplifier. Since the amplifier works in mono, we used only the left channel for input, output, and processing.

The audio processing is done entirely in the PL, as well as the generation of various clocks and user interface implementation.

2.2 The Effects

We created 4 effect blocks in total. An effect is not a single-trick horse, and can produce different sounds depending on its parameters (gain, time, frequency etc.).

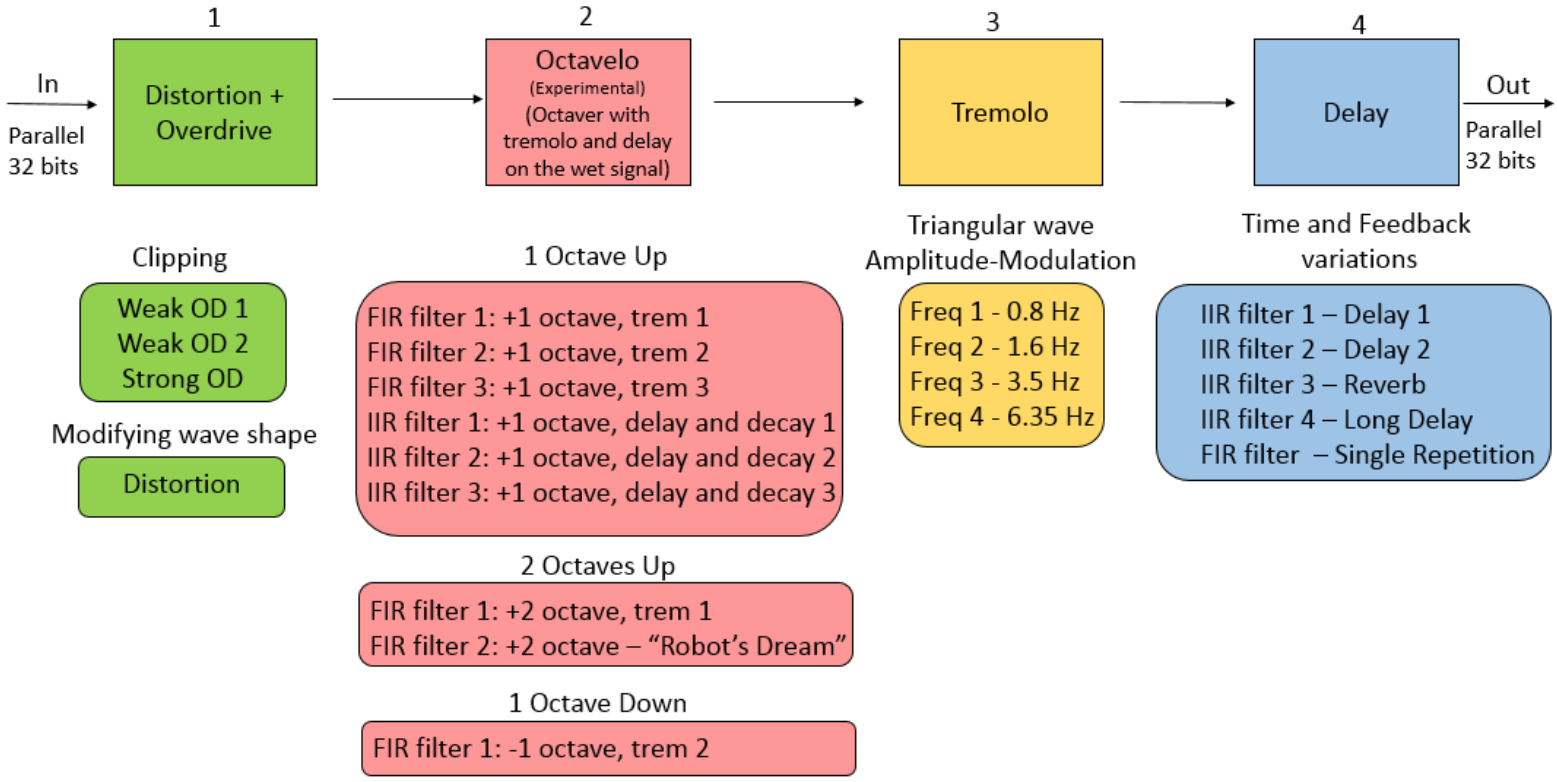
Our work method for creating each effect was:

1. Research the theory and implementation method of the effect.
2. Create a Matlab simulation to test if it performs as expected.
3. Implement, simulate, and synthesize (for debugging) the effect in Vivado.
4. Package the effect as a stand-alone IP block.
5. Import the IP to the Main project, synthesize, and test it with the Zedboard.

Different pieces of music require different effect configurations, even if they all reside under one general name. For example, "Delay" doesn't say anything except for the fact that the sound is delayed somehow at the output. It's different qualities such as time variation, feedback (the amount of repetitions) and the dry/wet signal mix ratio, define how we perceive the effect.

Therefore, we created several internal configurations inside each of the effect blocks (controlled by the user with the on-board switches and buttons) so the user can fiddle around and find the best match for each scenario.

Below is the effects-chain and the internal variations of each effect:



The above diagram shows the effects and their order of appearance in the serial effects-chain. The idea of creating this kind of chain is the standard way to do things is music - concatenate several effects, let the signal flow through all of them, and enjoy the combination.

There is no single "right" way to concatenate effects, and each guitar player can experiment and find his own preferred tone. However, there are some best practices which proved themselves over the years, and some guidelines that musicians should follow.

Our placement follows these guidelines:

- First comes Distortion/Overdrive, to engage the non-linear processing on the raw signal and create the basis which other effects will build upon.
- Next come pitch-shifting effects, such as the Octaver, to change the pitch of the signal and add some harmonics.
- Then there are volume-altering effects, such as the Tremolo, to force the amplitude-modulation uniformly over the signal. This is the last stop before time-varying effects, and for the tremolo to be effective, no other modulation should follow.
- Last are time-varying effects, such as Delay or Reverb, which only shift the signal in time, after it was already processed by the entire chain.

2.3 Prominent Protocols

Throughout the design, we utilized 3 well-known communication protocols:

- AXI – for PS-PL interconnection
- I^2C – to control the various registers of the on-board Audio-Codec
- I^2S – to send digital audio to/from the Audio-Codec

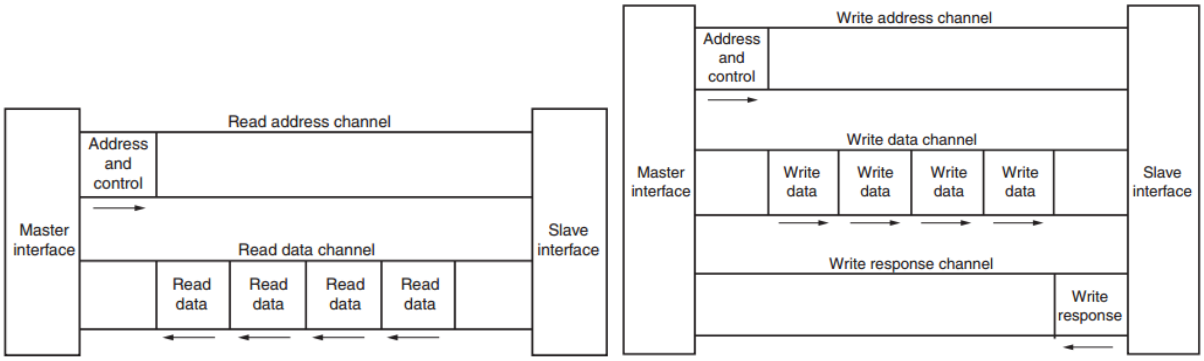
AXI - (Advanced eXtensible Interface):

AXI is part of ARM AMBA (Advanced Microcontroller Bus Architecture), a family of micro controller buses first introduced in 1996. Many IP (Intellectual Property) providers support the AXI protocol, and a robust collection of third-party AXI tools is available, making it a standardized and convenient protocol for IP implementation.

There are three types of AXI4 interfaces:

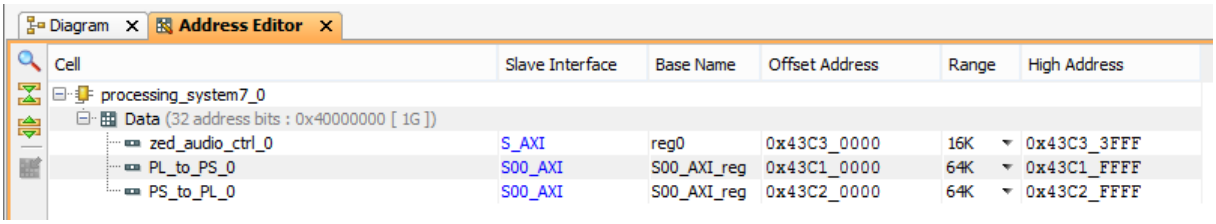
- AXI4—for high-performance memory-mapped requirements.
- AXI4-Lite—for simple, low-throughput memory-mapped communication. It has a small logic footprint and is a simple interface to work with both in design and usage.
- AXI4-Stream—for high-speed streaming data.

We used the AXI4-Lite as it is simple and sufficient for our design. Each AXI IP has four 32-bit registers which can be written/read from the PS. The AXI protocol works in a master-slave mode, representing IP cores that exchange information with each other.



AXI Read and Write channels

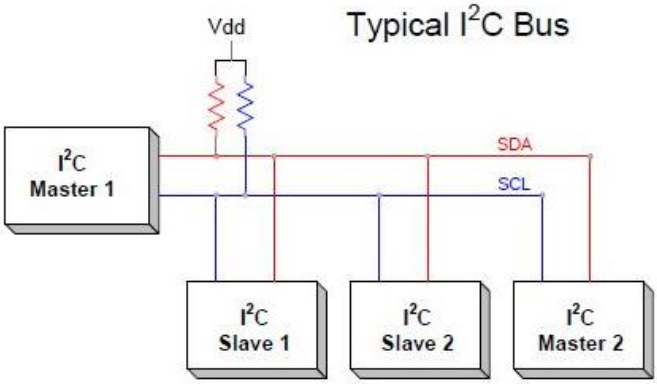
In our implementation, we use AXI to send audio samples between the PS and PL. Below is a screenshot from the project, showing the 3 AXI IP blocks we used, with the address range for each.



AXI IPs with their address range. The built-in functions in the ARM BSP (board support package) refer to these addresses for read/write operations.

I^2C - (Inter-integrated Circuit, also written as I2C):

The Inter-Integrated Circuit (I2C) Protocol is a protocol intended to allow multiple "slave" digital integrated circuits to communicate with one or more "master" chips. I2C uses two bidirectional lines, Serial Data Line (SDA) and Serial Clock Line (SCL), and on the Zedboard they are pulled up with $2k\Omega$ resistors and a voltage of +3.3 V. We used I2C to communicate with the Audio-Codec.



Typical I2C Bus

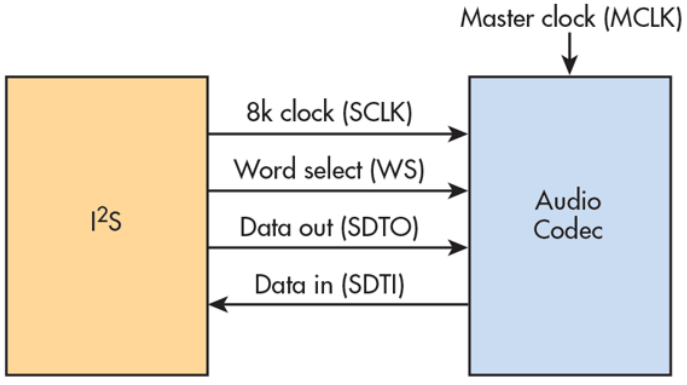
I^2S - (Inter-IC Sound, also written as I2S):

I2S (Inter-IC Sound), is an electrical serial bus interface standard used for connecting digital audio devices together. It is used to communicate PCM audio data between integrated circuits in an electronic device. The I2S bus separates clock and serial data signals, resulting in a lower jitter than is typical of communications systems that recover the clock from the data stream.

The bus consists of:

1. Bit clock line – officially called "continuous serial clock (SCK)" and labeled "bit clock" or BCLK.
2. Word clock line – officially called "word select (WS)" and labeled "left-right clock" or LRCLK.
3. Two multiplexed serial data lines – labeled SDATA_I and SDATA_O.

Exact clock rates are described in the Audio-Codec section later in this paper.

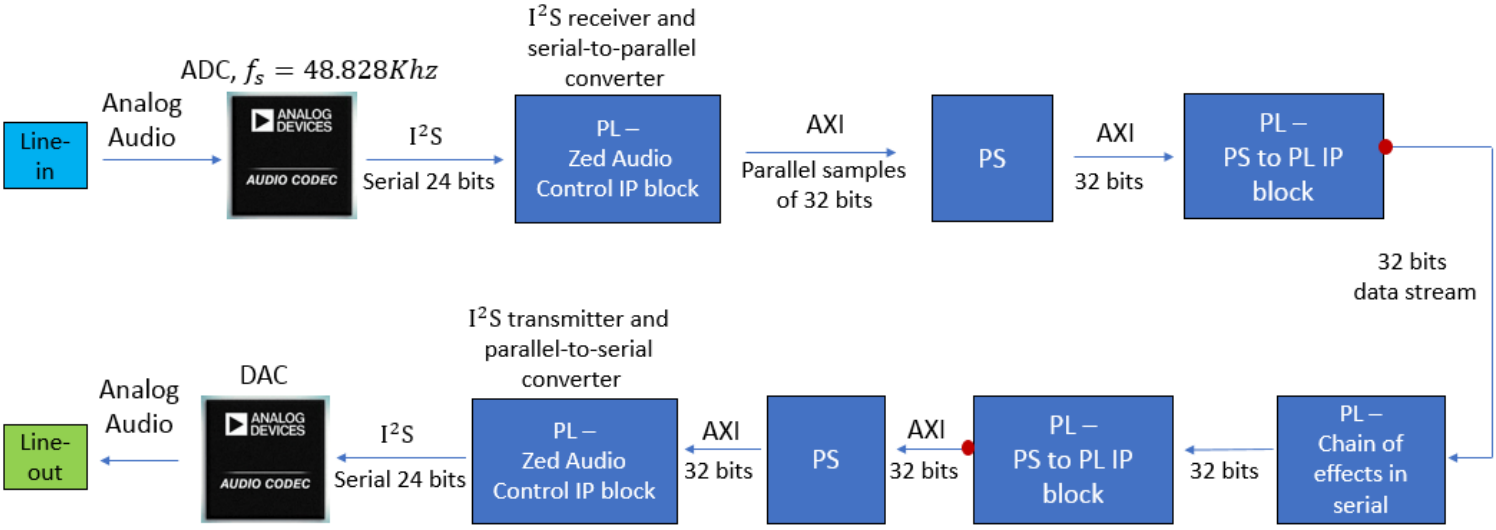


Typical I2S Bus connection between the codec and master chip

2.4 Signal Path

The following is a general description of the audio and control paths, and each block will be discussed in depth in the next sections.

2.4.1 Audio Signal Path



The guitar we used is a Squire Stratocaster with an output of 20-60mV, going into the Line-in 3.5mm jack. The output of the system was measured at 100-400mV, depending on the effect.

The analog audio is processed by the A/D on the Audio-Codec at a sampling rate of 48.828Khz and 24 bits, outputting a serial stream of bits in the I2S protocol.

The serial data enters the Zed_Audio_Control block, which creates parallel 32-bit samples (since the AXI peripherals in Vivado were designed for 32 bits), where the data is stored in the 24 LSB, padded from the left by 8 zeros, to complement the required 32-bit word. This IP also generates the LRCLK and BCLK required for the I2S protocol.

The 32-bit samples are outputted via AXI to the PS.

The PS reads the sample via AXI, and writes is back (also via AXI), to the PS_to_PL block, which then outputs the 32-bit audio sample into the PL fabric, for processing.

The reason we needed to pass the samples through the PS (and also the reason we needed to create the PS_to_PL and PL_to_PS blocks) is that the Zed_Audio_Control IP was taken from the Zynq-Book tutorial, where it was implemented in such a way that it outputs the samples to the PS rather than directly to the PL, and we needed to re-route the stream into the PL fabric for the audio processing.

So, the output of the PS_to_PL block is essentially the entry point of the 32-bit audio sample into the PL fabric (marked by the first red dot).

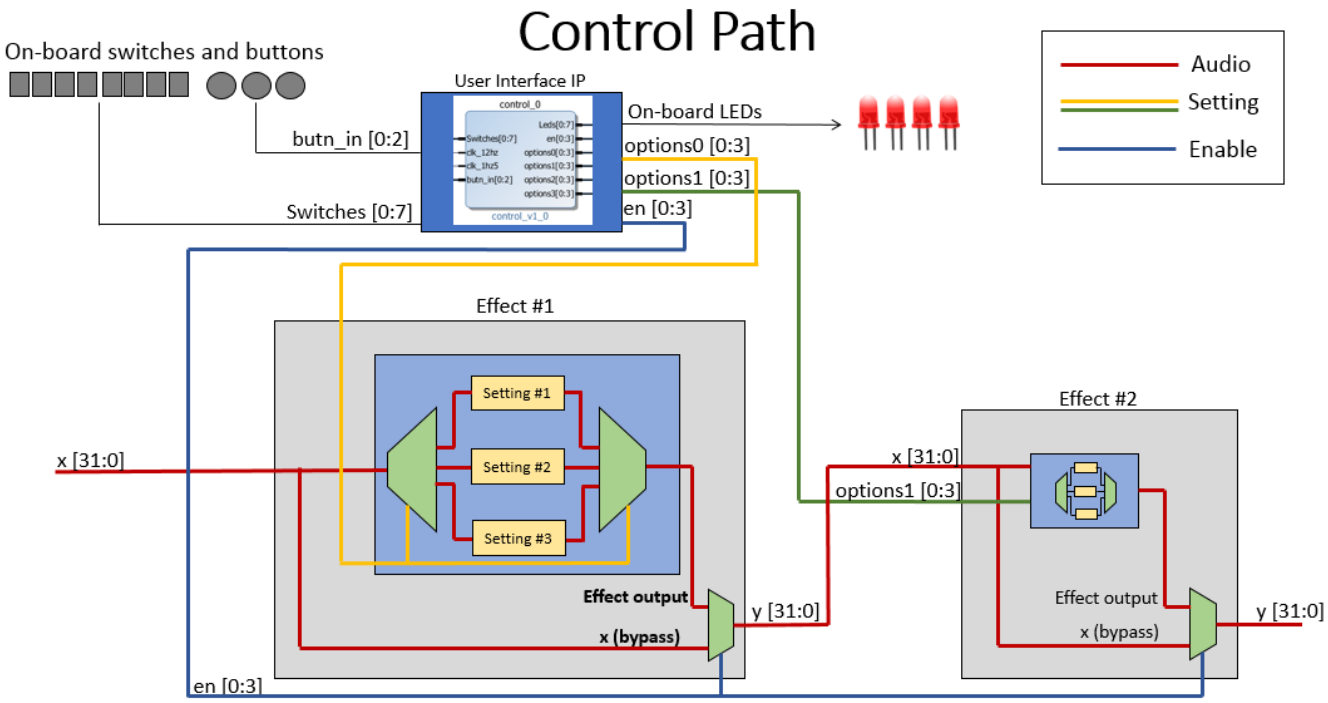
The sample travels through the chain of effects, altered by one or more of the effects (depending on the control parameters, which are described in the next section), and then it enters the PL_to_PS block, which sends the sample via AXI back to PS.

The same thing that happened in the beginning happens here in reverse - the sample goes from the PS to the Zed_Audio_Control block where it's converted to serial, then via I2S to the D/A in the codec, and out via the Line-out 3.5mm jack, to the guitar amplifier.

2.4.2 Control Signal Path

Below is a logical description of the control sub-system, demonstrated for 2 effects in series. A central IP block acts as a hub for all control inputs (buttons and switches) and outputs (switches and LEDs). The outputs are distributed among the effect blocks in the manner described below.

In this design, we tried to emulate the physical UI of a Multi-effects pedal.



Enable:

At the output of each effect block there is a 2x1 MUX which receives the 'en' signal, so the user can choose whether to activate the effect, or bypass it.

This signal is connected to the first 4 slide-switches on the Zedboard (when counting from left to right), and the user can choose which effect to activate by raising/lowering the relevant switch. The effects have a predefined order of appearance, which conforms to the musical theory of effect-placement (as described in a previous section). In this configuration, there can be up to 4 effects in the system.

Although all the effects share the same 'enable' bus, each enable signal is unique, so they are completely independent, but still can be combined in series if two or more are enabled – this is exactly the logic in the commercial Multi-effect pedals we wish to emulate.

Options:

Once selected, each effect has several different internal settings to choose from (for example various tremolo rates, or time delays). The 'options' control line sends a command to activate one of the internal settings, creating the effect. Each effect receives its own individual options line, so the user can change the setting for one effect, without affecting another.

This signal is connected to the other 4 slide-switches, allowing up to 16 internal settings for each effect.

Buttons and LEDs:

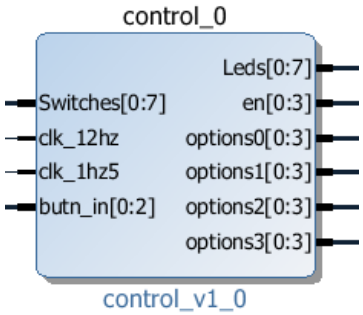
There kind of a "menu" logic to the buttons and LEDs system. At every moment only one LED is lit, which is one of the first 4 LEDs (corresponding to the 4 possible effects to choose from). The user can use the left and right push-buttons to "scroll" between effects, which also lights up the relevant LED.

This scrolling doesn't change any configuration for the currently working effects, and it is equivalent to a simple menu navigation. To select an effect, the user presses the central push-button, and then he "enters" the effect (at this point the left and right buttons are disabled) and he can choose one of the internal settings with the 4 'option' slide-switches, without affecting the other blocks.

To "exit" the effect, the user presses the central push-button again (the setting he chose is saved) and then he is back at the menu and can scroll to the next effect.

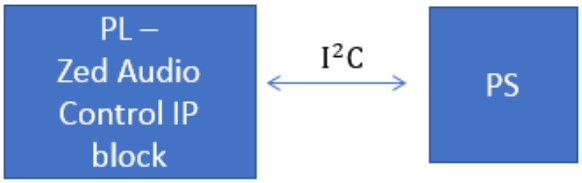
We found this UI to be intuitive and easy to understand, since it conforms with other UIs in various products we use.

Below is the UI IP block. Each of the 'options' connects to a different effect block:



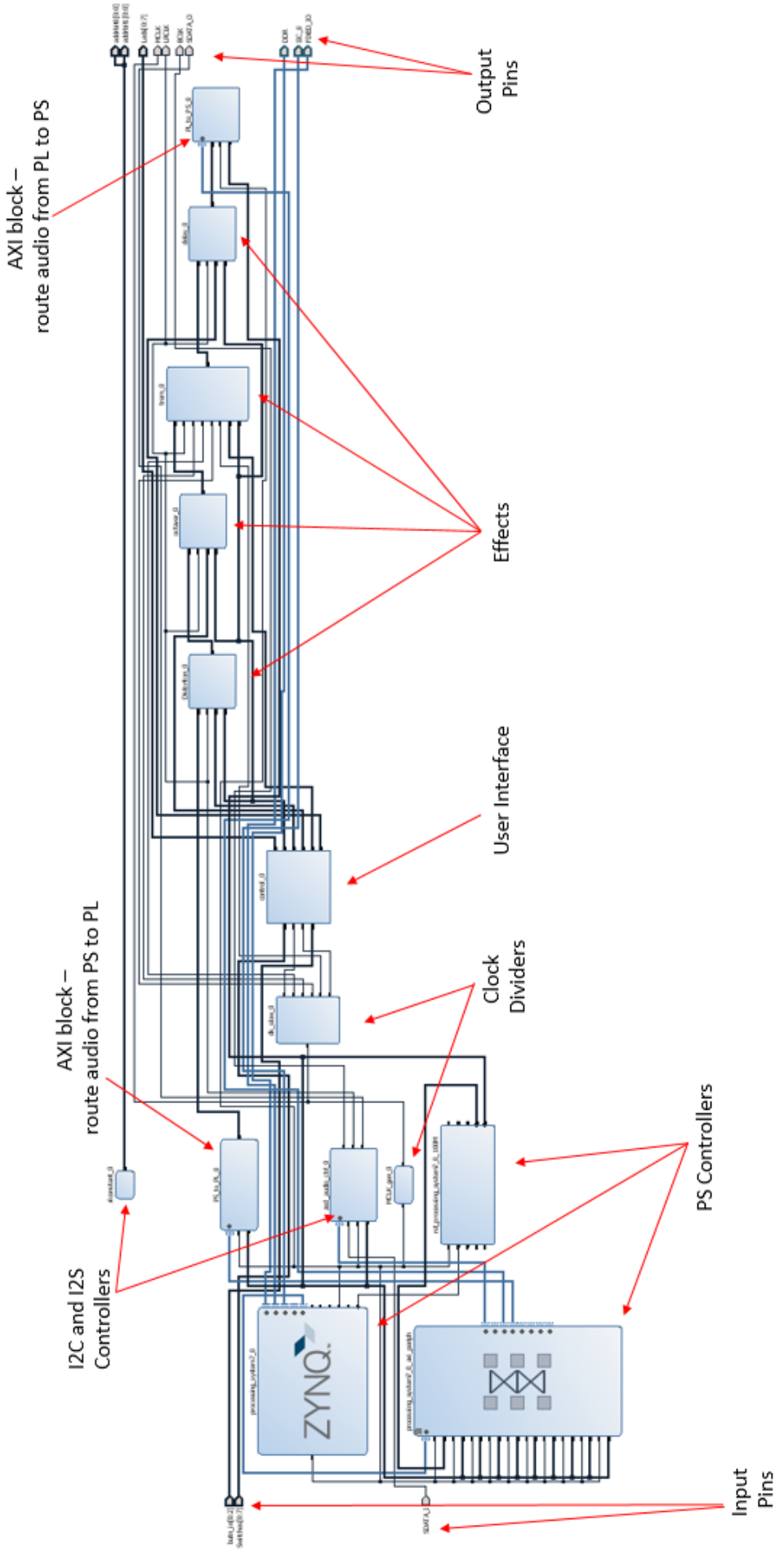
I2C:

Another control path is the I2C communication with the Audio Codec.



The I2C protocol is implemented in the PS, and the C functions came as a part of the Zed_Audio_Control IP from the Zynq-book tutorial.

2.5 Full Vivado Block Diagram



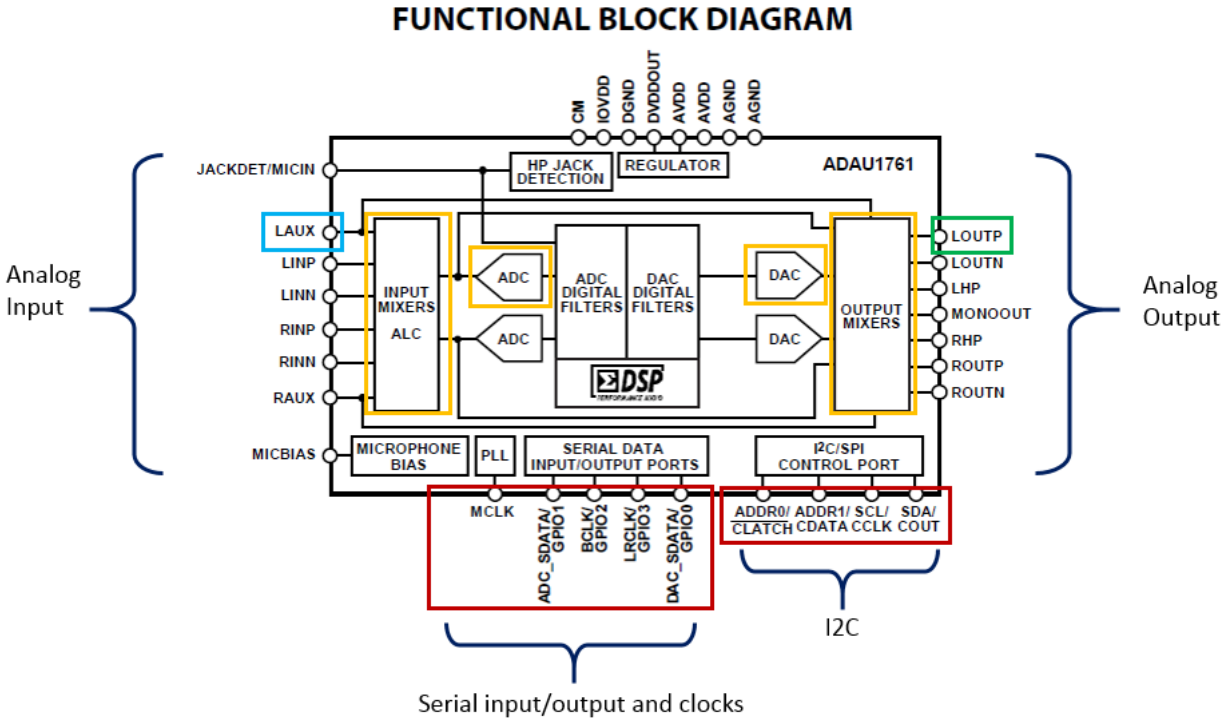
3. Building Blocks

3.1 Analog-Devices ADAU1761 Audio Codec

The Zedboard is equipped with an on-board Audio Codec, the ADAU1761 by Analog Devices. It supports sampling rates from 8-96 Khz and 24-bit depth stereo audio ADC and DAC, with SNR > 98dB.

In our project, we found it sufficient to work at around 48Khz. It is more than twice the maximal human hearing limit of 20Khz, fitting the Nyquist–Shannon sampling theorem, and it is even better than CD quality, which is 44.1Khz.

The Audio Codec is also the A/D and D/A, so a proper configuration is vital for good audio quality. Below is a block diagram taken from the user manual, with some additional annotations:



As can be seen in the block diagram above, the codec has various analog inputs and outputs, from which we use only the left channel of the physical Line-in port, and left channel of the physical Line-out port, since the guitar amplifier has a single channel connection. Usually when a 3.5mm connector is used as mono instead of stereo, the wiring is such that the left channel is the one utilized.

In the middle, there are some input and output mixers (which are described in detail in the next section), and separate left/right ADC and DAC. There is also a built-in DSP in the codec, but we didn't use it, as we wanted the audio processing to be done entirely by the FPGA.

At the bottom, there are 3 clock inputs - MCLK, LRCLK and BCLK, which are used by I2S for digital audio communication, and two serial ports – ADC_SDATA which is the output of the codec and the input of our system, and DAC_SDATA which is the input on the other end of the codec and the output of our system.

In addition, there are 4 ports for I2C – SCL and SDA which are the standard I2C lines, and ADDR0 + ADDR1 which represent 2 out of the 7 bits of the codec's I2C address.

I2C control:

The codec itself has an I2C slave address:

Table 21. ADAU1761 I²C Address and Read/Write Byte Format

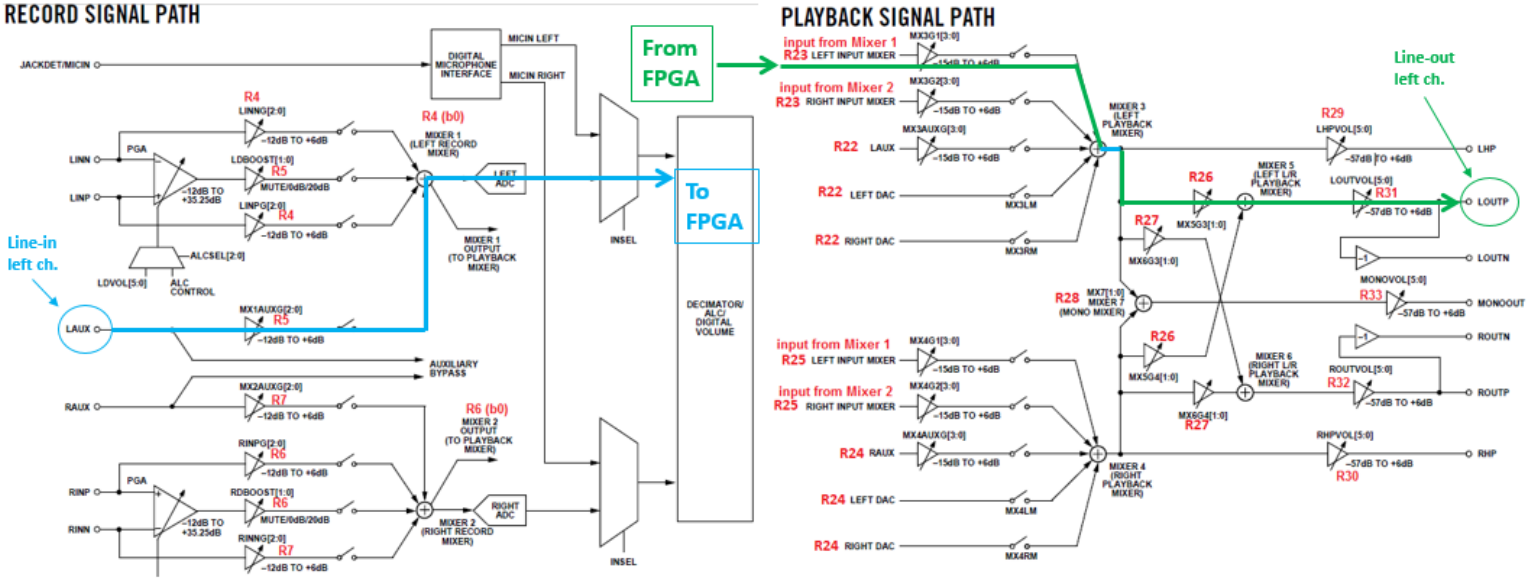
Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
0	1	1	1	0	ADDR1	ADDR0	R/$\overline{W}$

This address gives us two degrees of freedom in the form of ADDR0 and ADDR1 which are configurable, and we set them both to 0.

Mixers and signal path:

The codec has 67 programmable 8-bit registers (R0-R66) each control a certain aspect of the codec, and by writing to these registers it is possible to change the internal configuration of the codec.

The data sent to the registers consists of 3 bytes. The first two bytes are the address of the register between 0x4000 - 0x40FA, and the last byte is the value to be written.



The above diagram is taken from the user manual, with some additional annotations. We mapped the important registers which control the various mixers, so it will be easier to navigate through the schematic, and the path we took through all the relevant mixers (and registers) is also marked.

Clocking:

As mentioned before, the I2S protocol requires the codec to work with 2 different clocks (BCLK and LRCLK), and the codec also requires a Master-clock input (MCLK).

Clocks for the converters and the serial ports are derived from the core clock. The core clock can be derived directly from MCLK or it can be generated by the an internal PLL.

The PLL can be bypassed or used, resulting in two different approaches to clock management – we had some troubles locking the PLL which resulted in a lot of jitter and noises in the audio path, so we eventually decided to bypass the PLL entirely and feed a 50Mhz MCLK directly from the FPGA, which the codec uses to generate its master clock.

The master clock formula is also register-based (with 4 predefined values) and we chose the highest multiplier of $MCLK = 1024 \times f_s$ which results in: $f_s = 50Mhz/1024 = 48.828Khz$.

From the timing diagram in the user manual, it is possible to infer the required rates of BCLK and LRCLK:

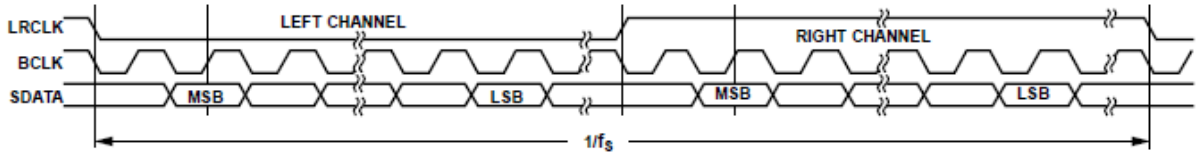


Figure 58. I²S Mode—16 Bits to 24 Bits per Channel

The bit clock pulses once for each discrete bit of data on the data lines. The number of BCLK cycles per audio frame is also configurable, and it can be 32 or 64. We set it to 64, resulting in two 32-bit windows (for left and right channels), where the actual data is in 24 of the 32 bits, and the rest is zero-padding.

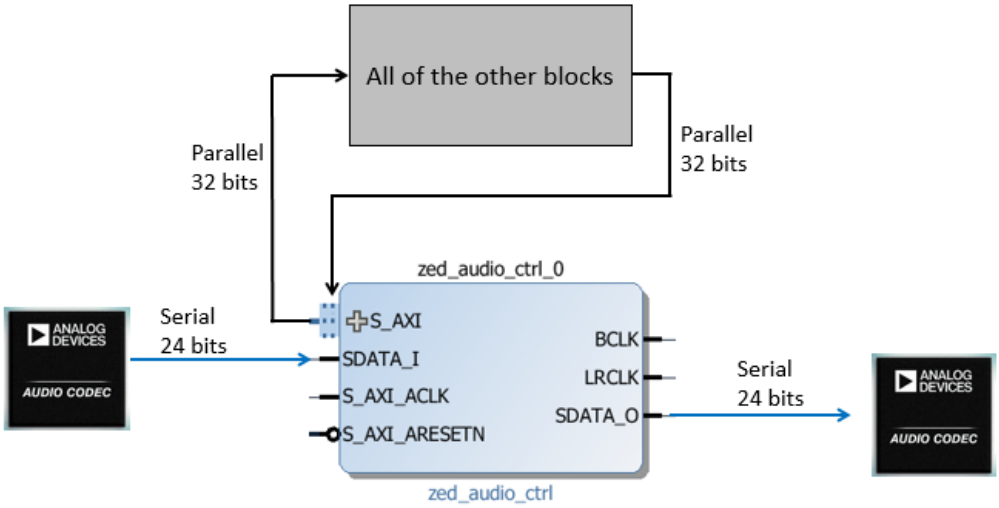
Since LRCLK defines the audio frame, this will be our sampling rate, so - LRCLK = 48.828Khz.

Now we need to fit 64 BCLK cycles into one LRCLK cycle, so – BCLK = 64 x LRCLK = 3.125Mhz.

3.2 Zed Audio Control

This IP (both HDL and C functions) was part of the code resources provided by the "The Zynq-Book Tutorials" eBook, in chapter 5 – "Adventures with IP Integrator". The original code was more extensive, and we minimized it to the essentials.

This block is both the entry and the exit point, and it communicates with the Codec.



This is an AXI block, and the AXI inputs connect to the main AXI controller. This block is the first to receive the serial sampled data from the Codec, on the SDATA_I port. The data flows by using the I2S protocol, and inside it's converted to parallel (32 bits, because of the AXI) and sent back via AXI to the PS.

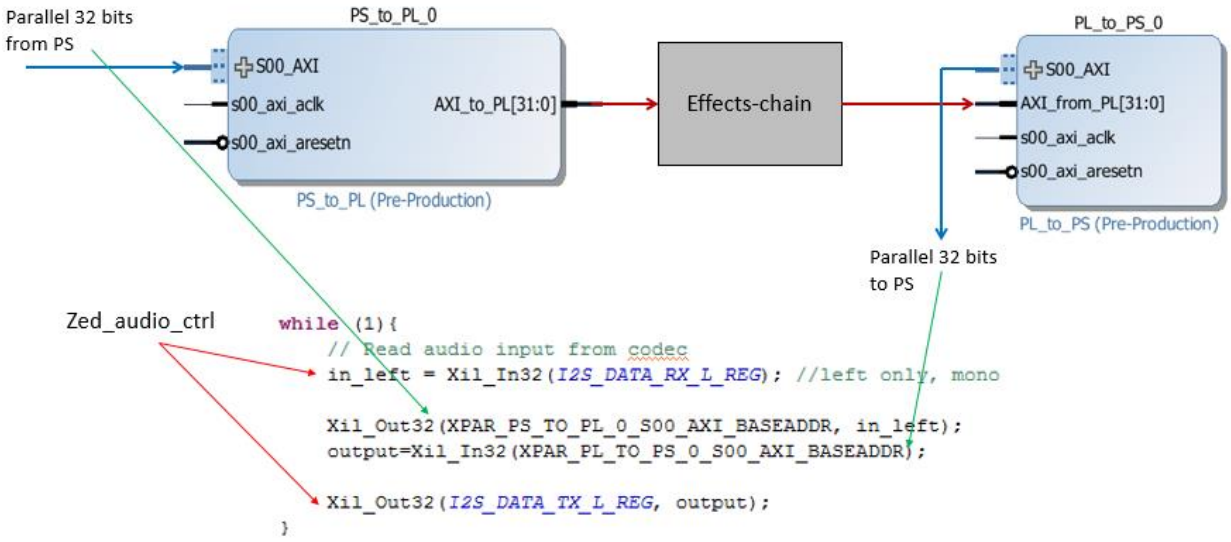
After all the processing is done, the signal returns to this block, to be converted to serial again, and it flows back to the codec on the SDATA_O port.

This block also generates the I2S BCLK and LRCK clocks.

3.3 PS-to-PL and PL-to-PS

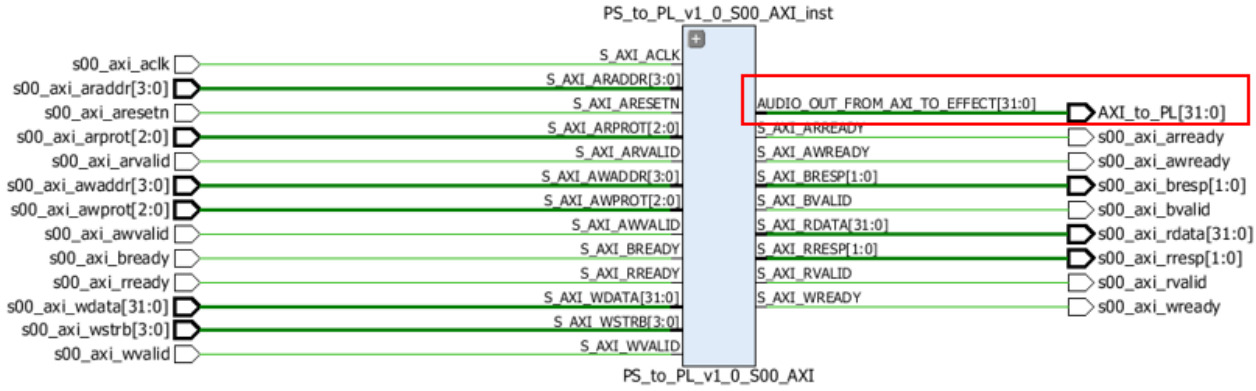
The PS_to_PL block is an AXI IP we created, which takes the audio from the PS and routes it in to the PL fabric. After the audio is processed, it returns to the PL_to_PS block, and then sent to the PS.

Below is a C code snippet from the process with the infinite audio-loop which drives the system.



PS to PL –

Each AXI4-Lite block has 4 registers, and we're using the first register, *slv_reg0* to write the data into. We took the basic AXI template and routed *slv_reg0* to the AXI_to_PL output. This way, the C AXI function writes to the register, and we route it into the PL.

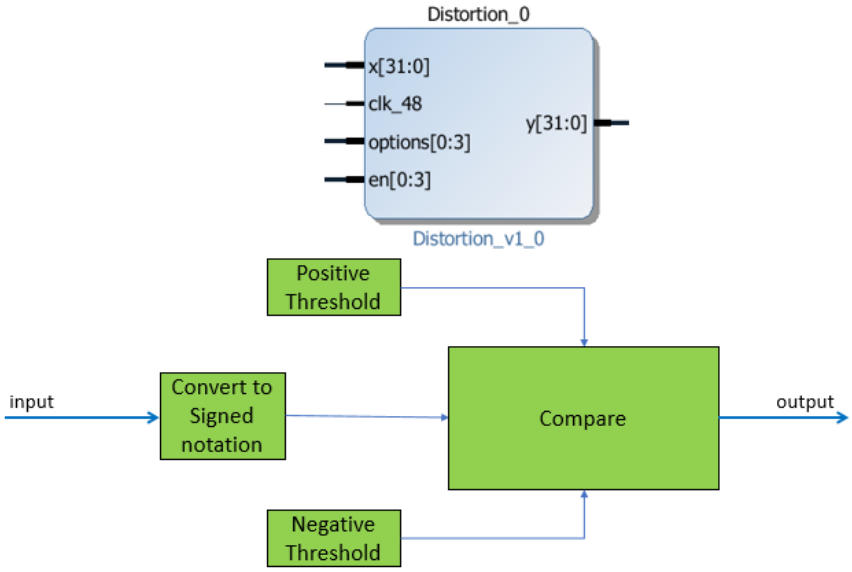


PL to PS –

This required a little bit of digging in the AXI protocol. We modified the code so that it will contain a permanent write-address of "0000", which is *slv_reg0*, and it will always write the data coming into the AXI_from_PL port, to that register. Since we haven't modified any of the internal logic of the block, the AXI protocol is still functioning, and we can read the data from this register with the C function above.

3.4 Distortion and Overdrive

Both Distortion and Overdrive (OD) belong to the family of non-linear effects. Usually people refer to one but mean another, since they are closely related, and sometimes is it hard to tell the difference. Distortion describes any process which add harmonics or intermodulations (or both) to the original sound. The human ear hears it as a distorted sound because of the added frequencies, where higher harmonics (7^{th} order and above) tend to sound harsher. The goal of the effect then, is to enrich the original signal by adding more spectral content, in a way which is pleasant to the ear.

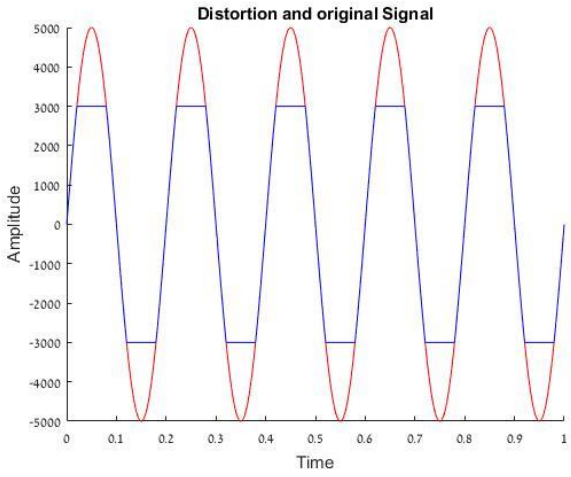


Overdrive -

Overdrive is a type of distortion, in which an amplifier is saturated (overdrived) because of high-gain input (since both transistors and valves has a saturation point beyond a certain input gain).

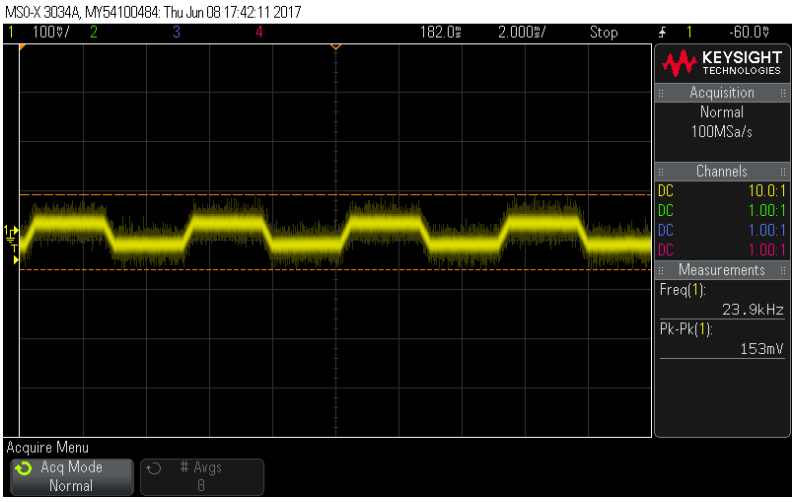
Digital OD wishes to emulate this process, by artificially creating a saturated wave-shape, by clipping the input signal above a certain threshold in the positive peaks, and below a certain threshold in the negative peaks. The level of clipping affects how distorted the sound is.

The signal is compered against 2 thresholds – a positive and negative, and if the value exceeds one of them, it is clipped to the threshold's value. Otherwise the signal is passed unaffected. Below is a Matlab simulation of the Overdrive:



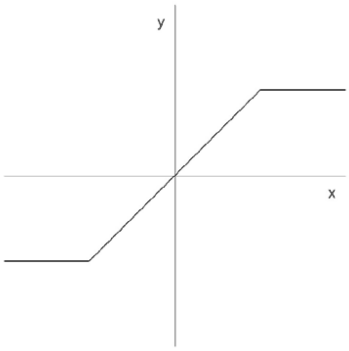
Clipping simulation in Matlab. Red is the original wave, and blue is the clipped output.

Below is a simple sine wave which passes through the effect, measured with a scope:



Clipping in real-life

In the frequency domain, the outcome is also expected. Since the clipping function is an odd function, when applying it to the signal we expect only odd harmonics to appear in the spectrum.



The clipping function

To reinforce the idea using some DSP math – we know that the Fourier series is defined as:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} [\, a_n \cos(nx) + b_n \sin(nx) \,]$$

With the coefficients:

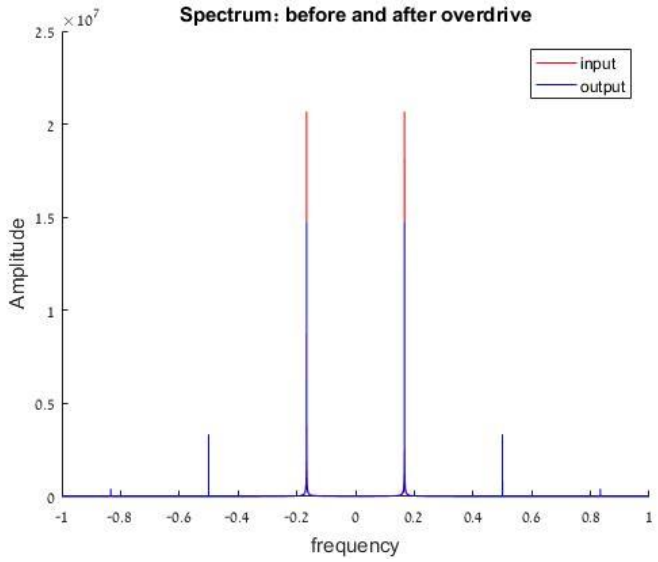
$$a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x)\, dx$$

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(n x)\, dx$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(n x)\, dx$$

So, if $f(x)$ is an odd function (like our clipping function), all a_n coefficients will equal zero (since we're integrating an odd function over symmetrical limits), and when calculating b_n we are left only with the odd coefficients, and the odd harmonics.

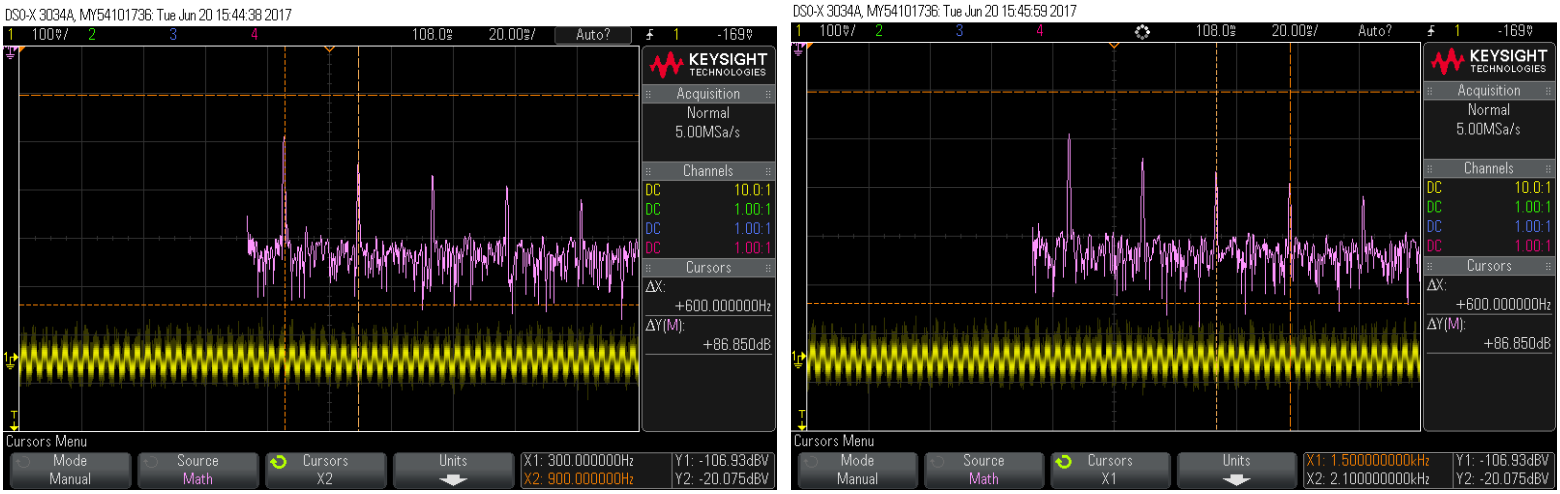
In a Matlab simulation, the odd harmonics are very clear:



OD spectrum. The frequency scale is normalized to [-1,1].

Although the frequency scale is normalized for the simulation, the relations between the frequencies is clear – the fundamental frequency is at around 0.18 in the plot, and it is the strongest. Its third harmonic at 0.54 also shows, and a little spike is also visible at 0.9, which is the 5th harmonic.

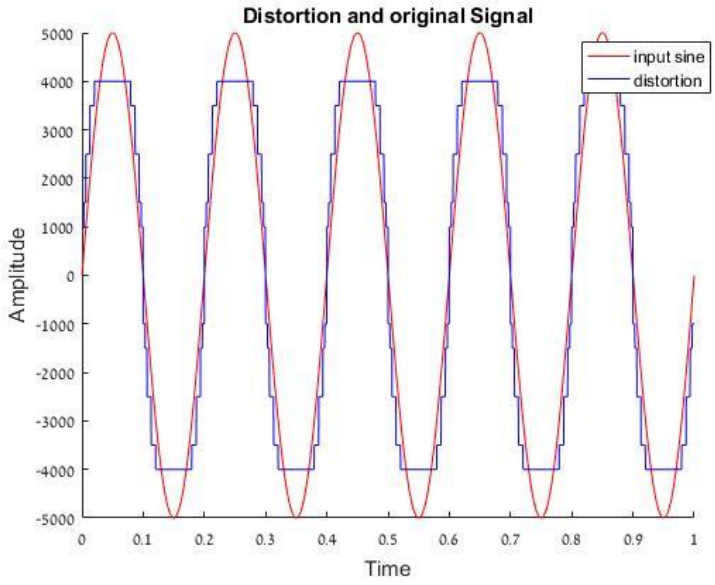
In the scope, we see the same effect, and it's even more prominent. The fundamental frequency is 300Hz, and its odd harmonics at 900Hz, 1500Hz, 2100Hz and 2700Hz (3rd, 5th, 7th, 9th) are showing:



OD spectrum, fundamental frequency of 300Hz, and its odd harmonics.

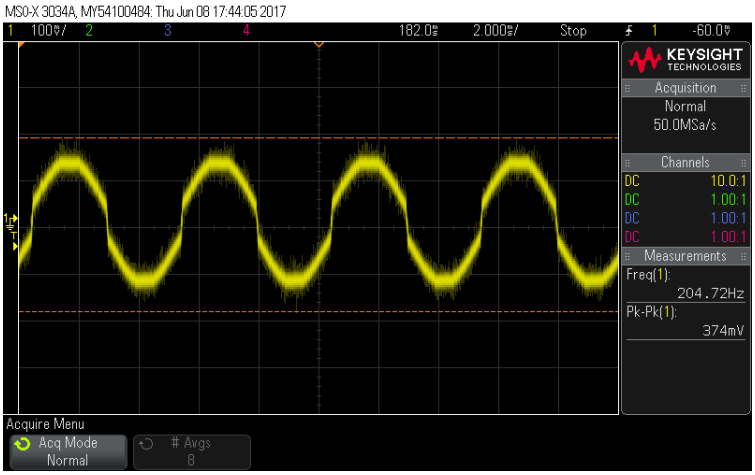
Distortion -

As opposed to OD, we implemented Distortion as a more complex wave-shaping, rather than clipping at a single threshold. We measured the output range of the Codec, divided the entire range into roughly 80 different sub-sections, and compared each one with a different threshold. This results in a harsher sound than the OD. Below is a Matlab simulation (with fewer threshold than the VHDL implementation, but the idea is clear):



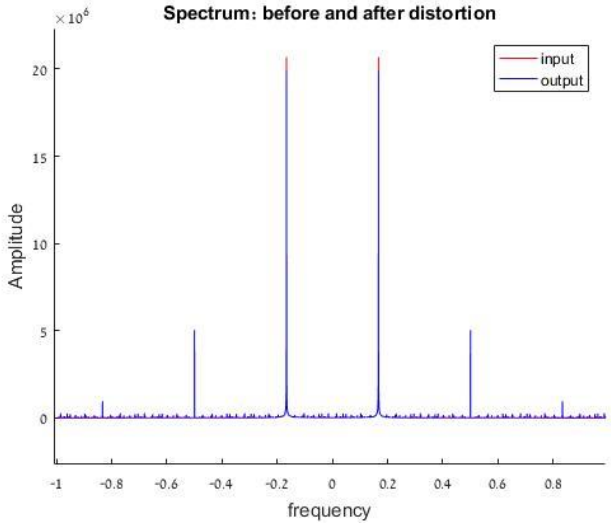
Distortion simulation in Matlab. Red is the original wave, and blue is the distorted output.

Below is a simple sine wave which passes through the effect, measured with a scope. The wave-shaping is more aggressive, and it alters the entire wave, not only the peaks:



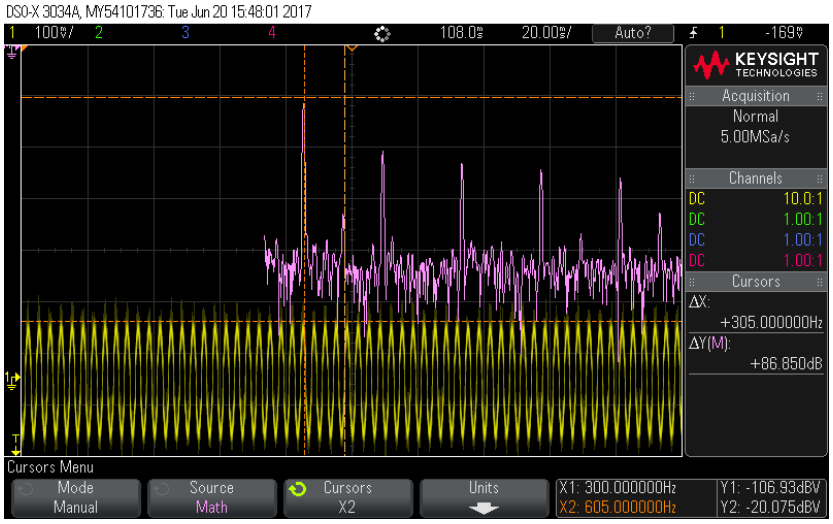
Distortion in real-life

In the frequency domain, the results are similar to the OD, with louder harmonics:



Distortion spectrum. The frequency scale is normalized to [-1,1].

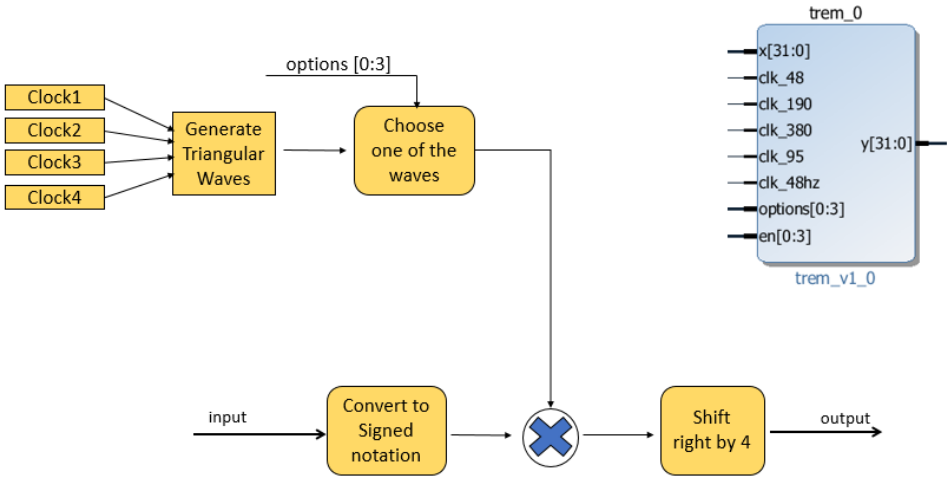
In the scope, we can see a richer harmonic content, with some visible even-harmonics as well, since the distorting-function is no longer simply odd or even:



Distortion spectrum. Strong odd harmonics, and some even-harmonics.

3.5 Tremolo

Tremolo is basically Amplitude Modulation (AM). The effect is generated by multiplying the audio with a modulating signal, which changes the envelope of the modulated signal. In VHDL, we created several triangular waves, and multiplied them by the input signal. The result is a triangular envelope on the input signal.



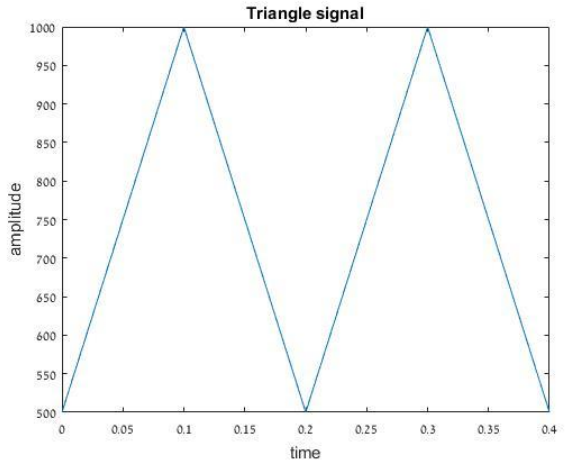
There are 4 input clocks, which generate 4 different triangular waves. The problem with multiplication is that if the output is too large, it will saturate the DAC, and the audio will be distorted. We found that multiplying by up to 30, and then normalizing the output by 16 (shift right by 4) yields a good result, and the output is not saturated.

The triangle wave is implemented by a counter, which counts to a certain value, and then back down. Setting this value to 30, a full period means 60 clock cycles. So, by calculating how many periods are required from the triangle (the tremolo's frequency), it is easy to determine which clock is required to produce this frequency.

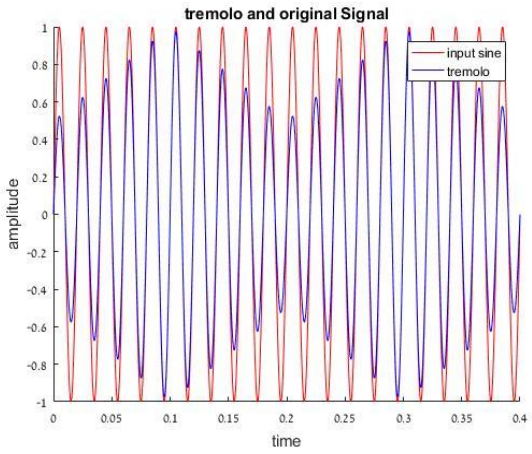
After some experimentation, we found several clock values which create triangle waves in 4 different frequencies, and sounded very good:

Clock rate (Hz)	Tremolo frequency (Hz)
48	0.8
95	1.6
190	3.2
380	6.35

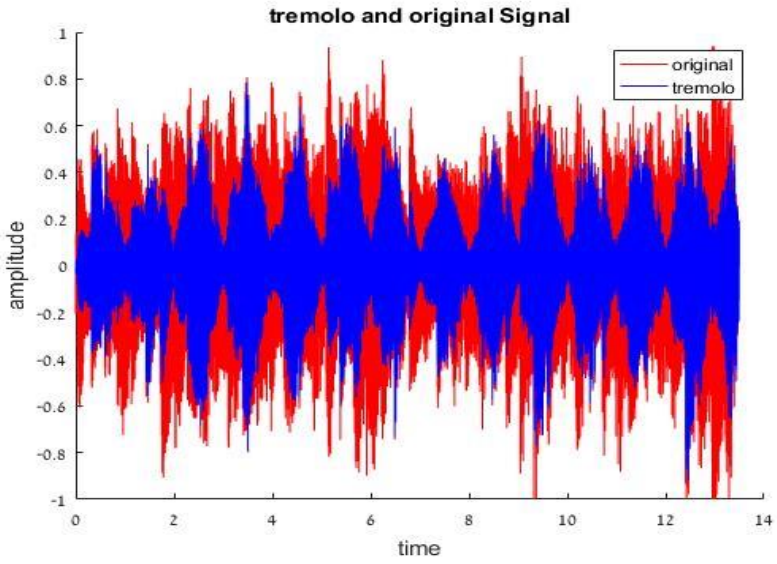
Matlab simulations:



Triangle wave

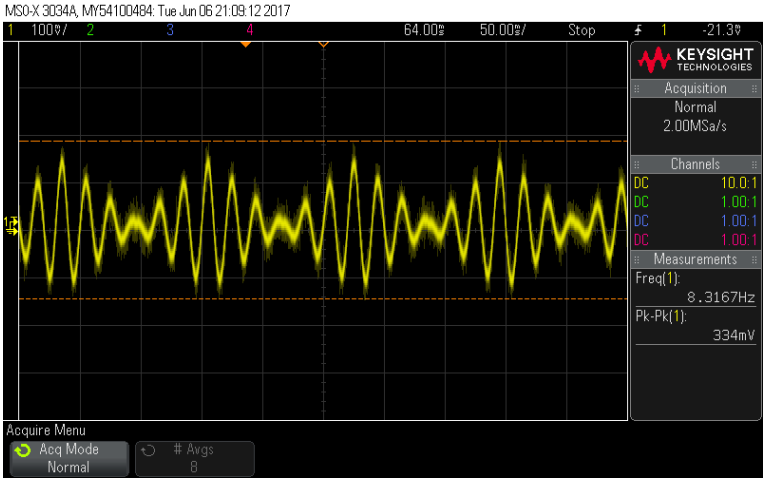


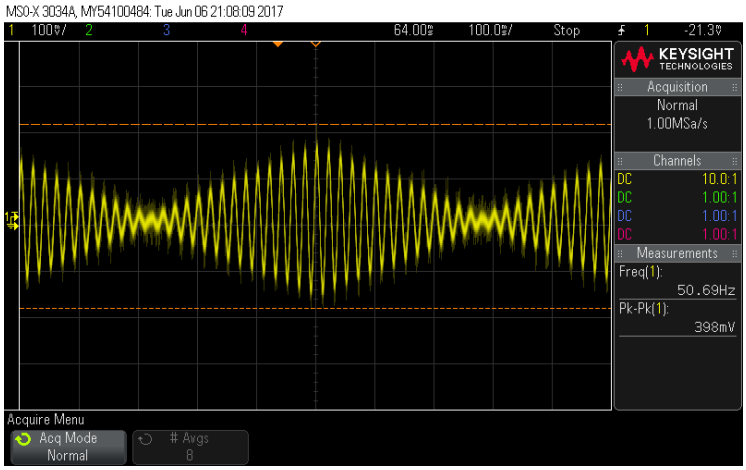
Modulated sine wave

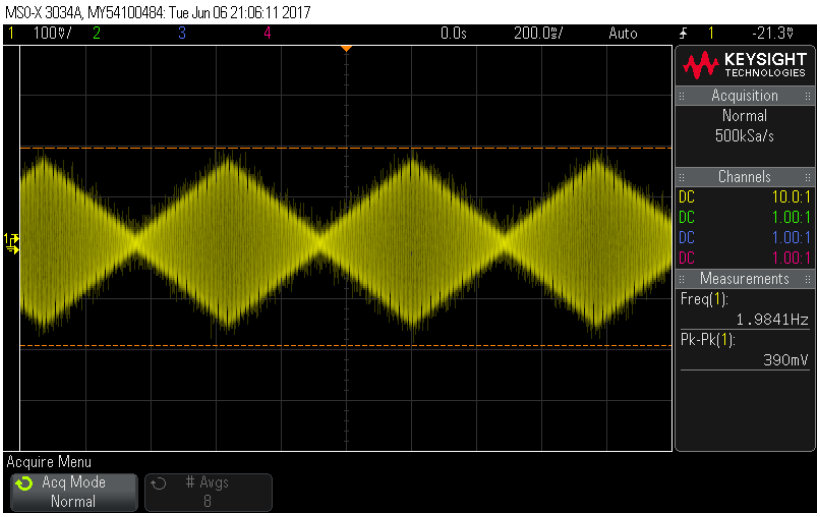


Modulated audio recording

The effect is clearly visible in the simulation – the signal's envelope has received the shape of the triangle wave. In the scope measurement, the result is the same:





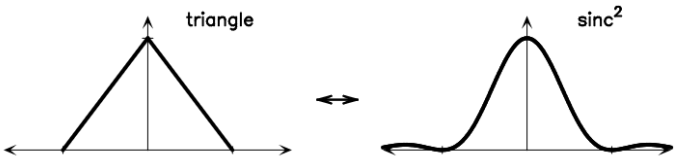


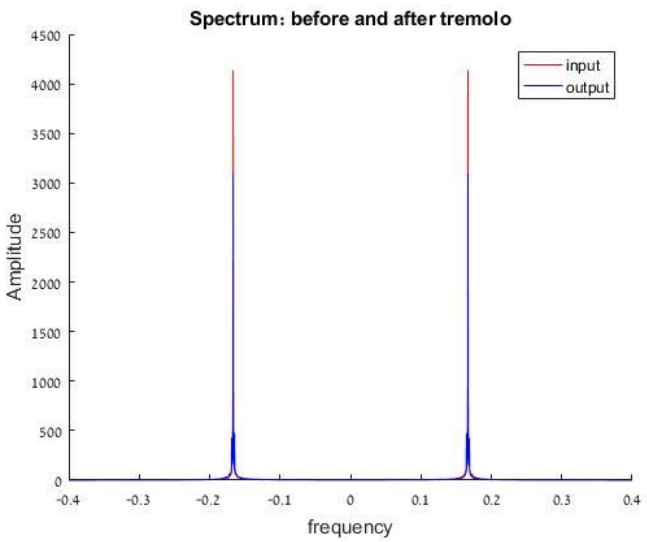
Tremolo with various frequencies. The frequency of the original signal is not affected, but only its amplitude – the envelope of the signal is transformed into a triangle.

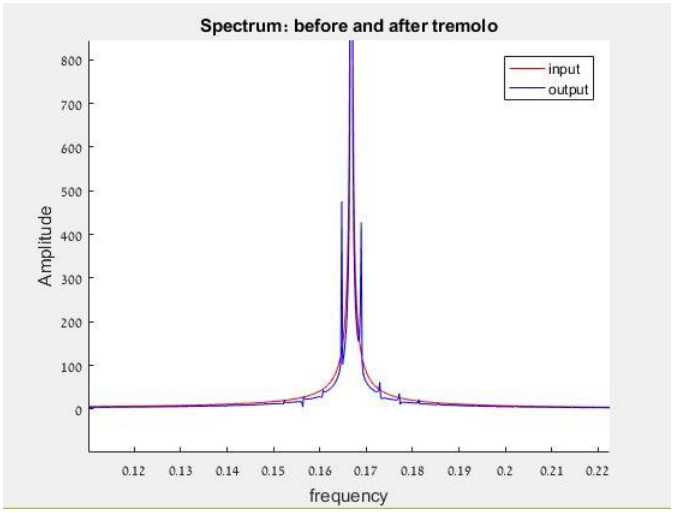
In the frequency domain, there's an interesting outcome. Since we are multiplying by a triangle wave in the time domain, it is equivalent to convolution in the frequency domain.

A tringle wave in the time domain is created by convoluting two square pulses – each square is a Sinc function in the frequency domain, so the triangle will be $Sinc^2(f)$ in the frequency domain.

Now, we are convoluting a delta function (sine wave in the time domain) with the $Sinc^2(f)$, which is akin to moving the Sinc to the delta's frequency.

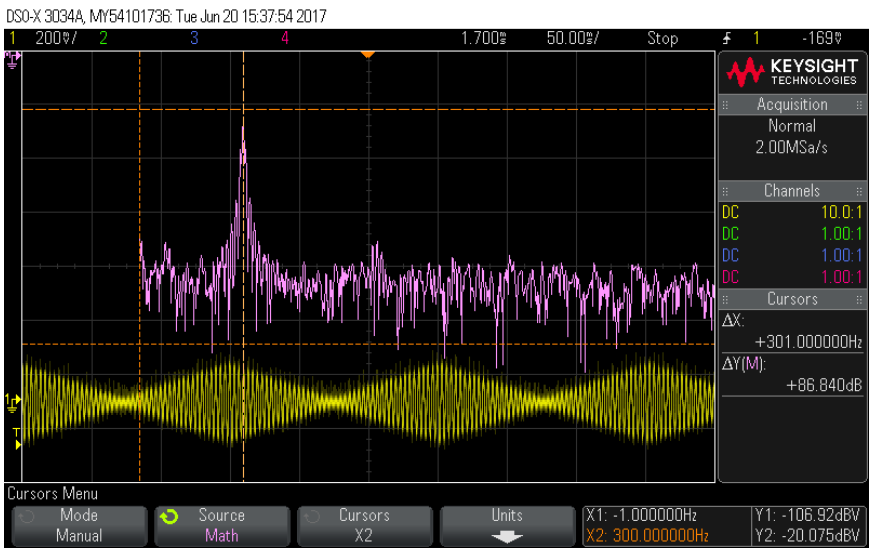






Tremolo spectrum - when zooming in (right) it is possible to see the original wave shape. This is how Matlab interpreted the triangle wave.

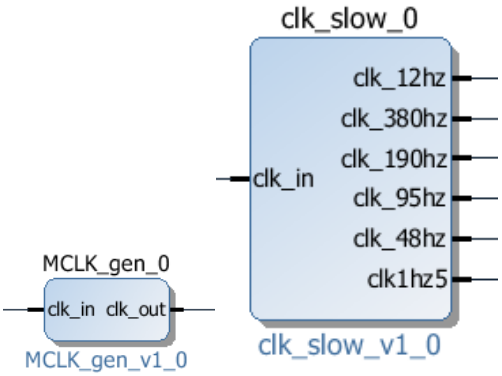
In a real measurement, we see the same result. The audio signal was a 300Hz sine wave, thus the spectrum is around that frequency, with some bandwidth which resembles $Sinc^2(f)$.



Tremolo in real life

3.6 Clock Dividers

The Master clock in our project is a 100Mhz clock, which we divided into various clocks for our use. The MCLK_gen creates the 50Mhz MCLK for the codec, and the clk_slow module further divides the 50Mhz clock into slower clocks.



The division is done by powers of 2, with a 26-bits counter which counts at 50Mhz, and we can generate various clocks by accessing different places in the counter.

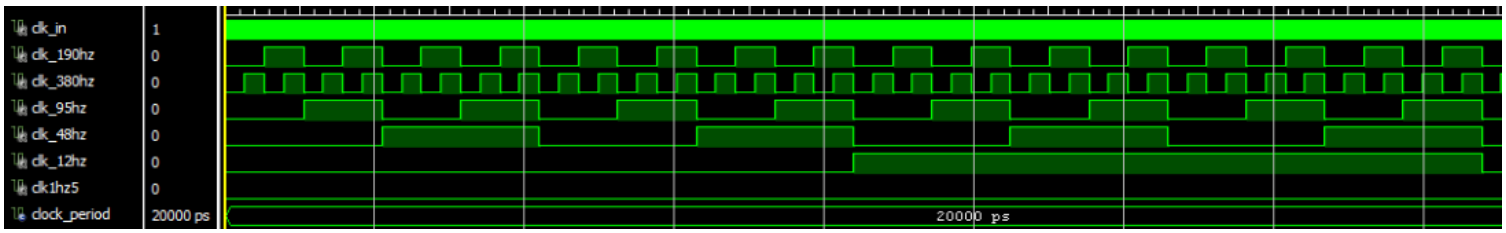
```
signal clk_cntr : std_logic_vector(25 downto 0) := (others => '0');

--****** for tremolo  **********
clk_380hz <= clk_cntr(16);
clk_190hz <= clk_cntr(17);
clk_95hz <= clk_cntr(18);
clk_48hz <= clk_cntr(19);
--*****************************
```

For example:

$$\frac{50Mhz}{2^{17}} = 381Hz$$

So, to create a 381Hz clock, we need to read the 16th place in the counter (since it starts from 0).

Below is a Vivado simulation of the various clocks. The clk_in pin is at 50Mhz, so it's not visible in this resolution:



3.7 Delay

Delay is a time-varying effect, which stores incoming samples and then adds them back to the original signal, to create overlaying repetitions.

In this block, we implemented both FIR and IIR filters, since each approach creates a different sounding effect. For both filters we set g=0.5, and by changing M, we control the delay time.

FIR (Finite Impulse Response) –

We implemented a single-tap FIR filter, which creates a single repetition with a delay of M samples. The output depends only on the current sample and the once-delayed sample. Each sample is multiplied by g=0.5.

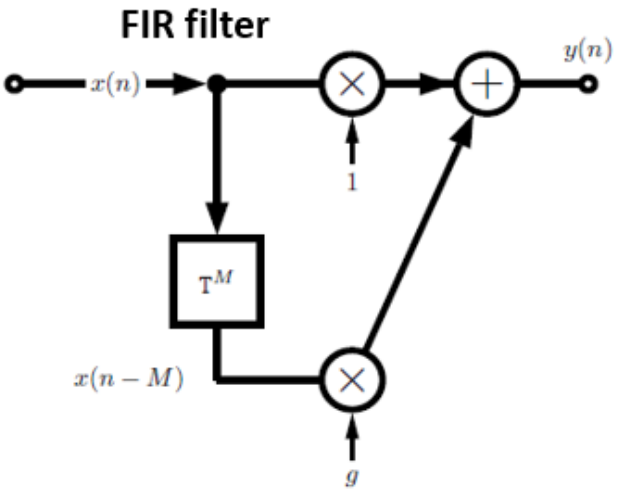
The FIR filter that we used has the following difference equation:

$$y(n) = x(n) + g \cdot x(n - M)$$

The effect sums the current sample x(n) and another delayed sample, x(n-M).

The transfer function:

$$H(z) = 1 + g \cdot z^{-M}$$



IIR (Infinite Impulse Response) –

This effect consists of an infinite series of decaying delayed samples.

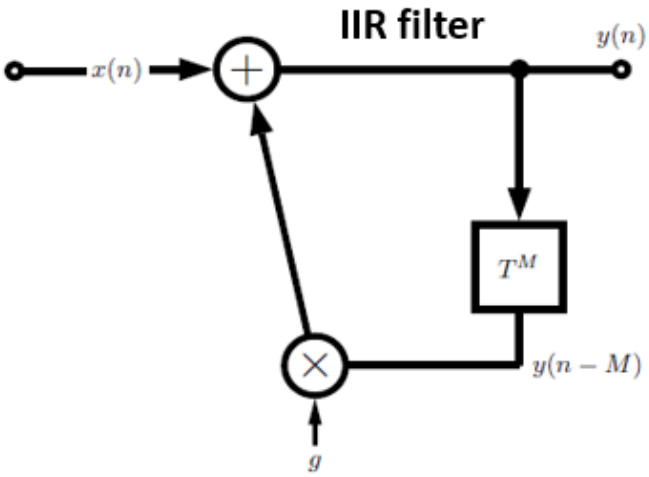
The effect sums the current sample x(n) and all the samples before it. The delayed sample is multiplied by a factor of g=0.5, to attenuate the older samples. In this filter, the attenuation is important, to keep the output which is created by infinite summations, in bound.

The IIR filter that we used has the following difference equation:

$$y(n) = x(n) + g \cdot y(n - M)$$

And transfer function:

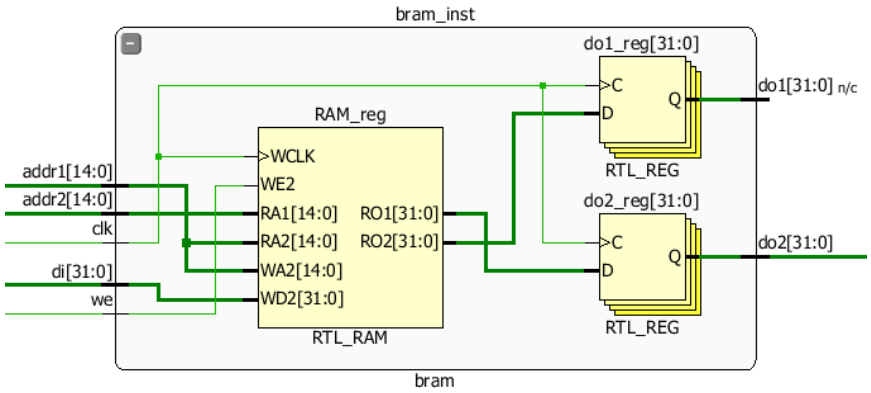
$$H(z) = \frac{1}{1 - g \cdot z^{-M}}$$



BRAM –

The Zynq-7000 contains dedicated RAM blocks in the FPGA fabric, which are called Block-RAMs (BRAM). Each BRAM is 36kb in size, and can be used as single port or dual port RAM interface, depending on the implementation. In order to store the delayed samples, we created several BRAM instances, with a size to 20,000 samples (which yields a maximum delay of approx. 400ms, at 48Khz sampling rate).

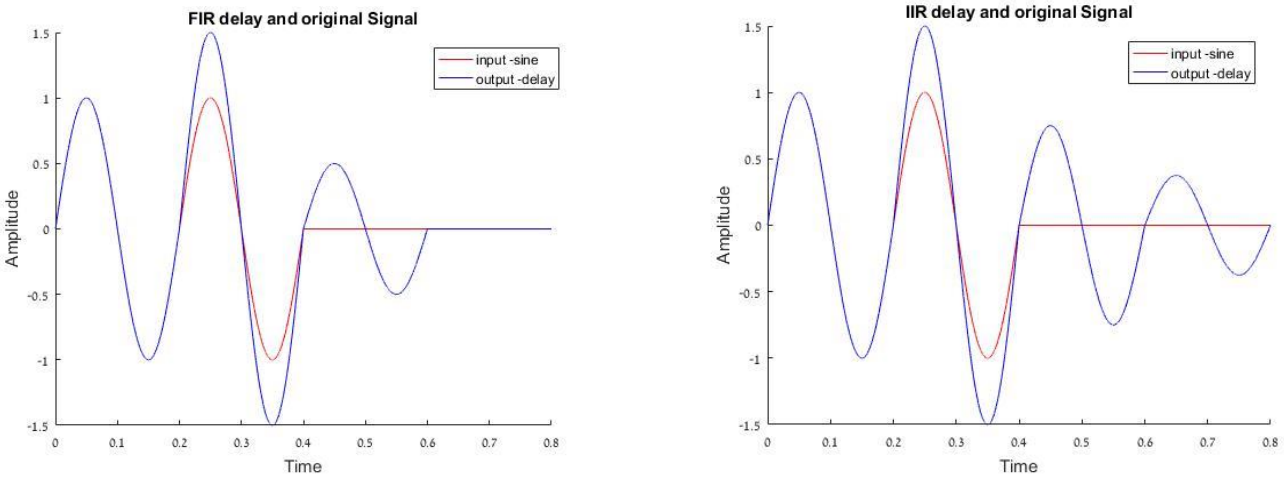
We created a dual-port RAM, with a separate read and write pointers, where one of them is used to constantly fill the RAM array, and the other one to read the samples.



By writing the code with a certain template (called XST), the Vivado synthesis tool can infer the use of BRAM, thus using the space in the FPGA more efficiently. Without using this template, we found that several 20,000-sample arrays cause the FPGA to use 98% of its LUT capacity, instead of using BRAM blocks. After following the XST template, a dedicated block of a RAM_reg type was created.

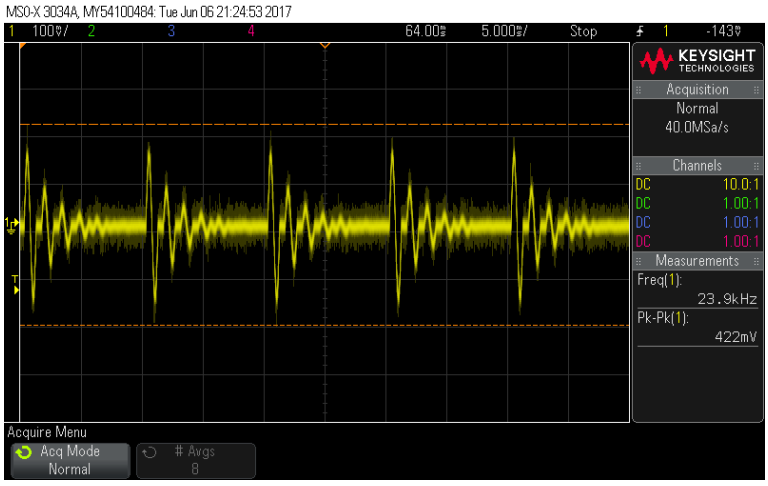
We implemented a BRAM instance in the Delay and Octavelo blocks, with 20,000 samples of 32-bits each, and used 46 out of the total 140 available BRAMs on the Zynq.

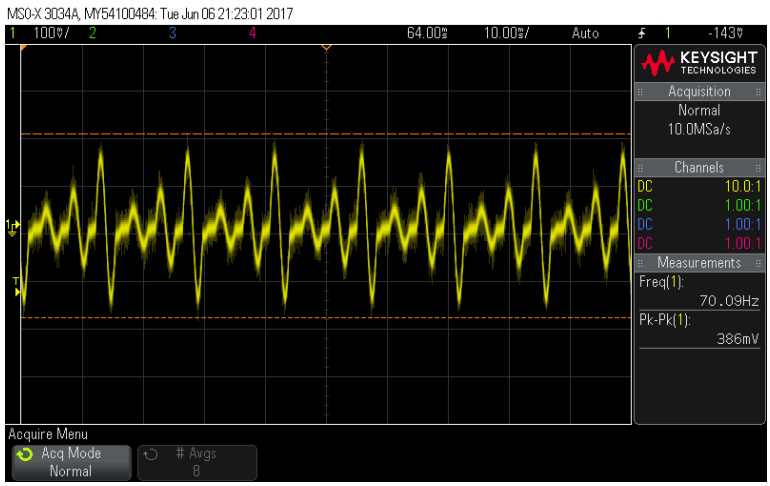
Below are time domain Matlab simulations. The input wave (red) is a single pulse, with a trailing edge of zeros. The FIR makes a single repetition and stops, while the IIR continues to add more samples.



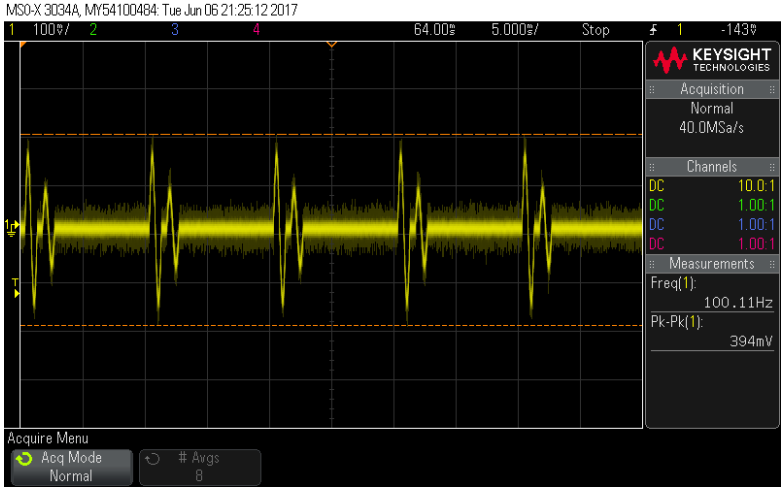
FIR and IIR. Before the delay kicks in, the waves overlap.

Scope measurement:





IIR at different time delays – infinite decaying samples. g=0.5.



FIR – a single repetition. g=0.5

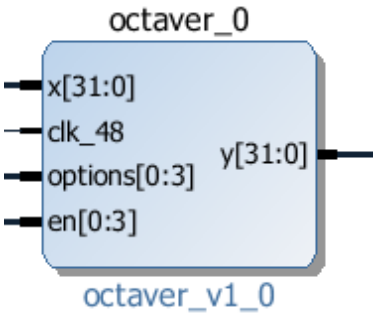
3.8 Octavelo

This is an experimental effect. We decided to test some interesting combinations to see what is possible to achieve with the entire digital domain and our imagination.

Octavelo is a combination of Octaver and Tremolo – the dry signal is added to the processed signal (wet), which is raised or lowered by 1 or 2 octaves (by up-sampling or down-sampling). In addition, we added some variations which create a set of decaying samples of the octave-up (based on IIR or FIR) which produced some very interesting effects.

The basis for this effect were the FIR and IIR filters and the BRAM, which are described in the Delay chapter.

The up-sampling and down-sampling was achieved by varying the speed ratio between the read and write pointers in the BRAM.



Up-Octave (Down-Sampling)

In music, an octave is the interval between one pitch and another with half (down-octave) or double (up-octave) its frequency. For example, the high E string on a guitar has a frequency of approx. 330Hz, and the 12th fret on this string, which is an octave-up, is 660Hz.

Up-Octave is akin to downsampling in DSP, where we are diluting the number of samples and reading every second sample. This can be seen in the following Vivado simulation:

Addr1 is the write address, which works at the sampling rate, and addr2 is the read address. When reading every second sample, we essentially use only half of the samples, which in the frequency domain is equivalent to expanding the frequency axis (multiplying it by 2). We hear it as doubling the frequency – hence the up-octave.

Since in the process of storing and recalling samples we use FIR and IIR filters, a tremolo and delay effects are also achieved. The tremolo happens because of the rapid reiteration of notes (which is a different tremolo kind than the dedicated effect block we created, but it is also called 'tremolo').

The math behind the process:

If the original sampling time was T, then after diluting the number of samples we are enlarging the time between samples by a factor of M, so T'=MT.

The Fourier transform of the signal (with replications) is:

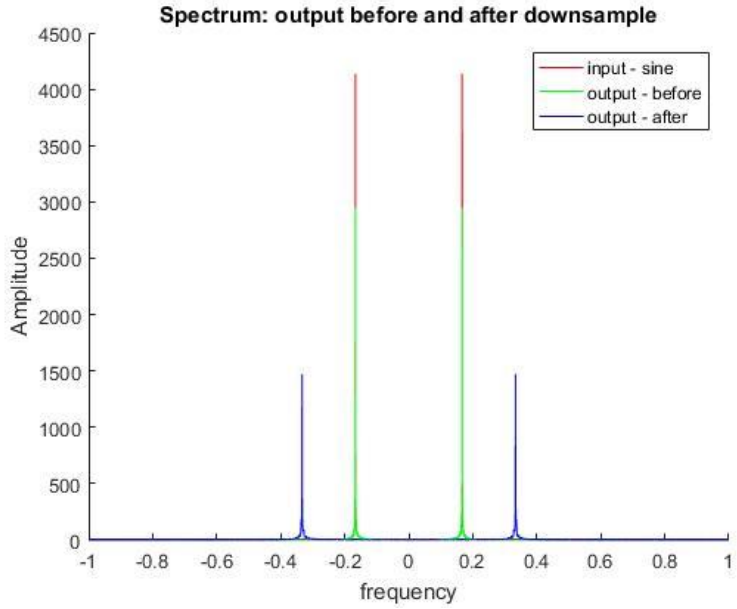
$$X(e^{j\omega}) = \frac{1}{T}\sum_{k=-\infty}^{\infty} X_c(j(\frac{\omega}{T} - \frac{2\pi k}{T}))$$

If we substitute T with MT we get:

$$X_d(e^{j\omega}) = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{T}\sum_{k=-\infty}^{\infty}X_c\left(j\left(\frac{\omega}{MT}-\frac{2\pi k}{MT}-\frac{2\pi i}{MT}\right)\right)$$
$$= \frac{1}{M}\sum_{i=0}^{M-1}X\left(e^{j\left(\frac{\omega}{M}-\frac{2\pi i}{M}\right)}\right)$$

In this formula, we can see that the frequency (ω) is divided by a factor of M, which causes the frequency axis to *expand* by M. In our case, M=2.

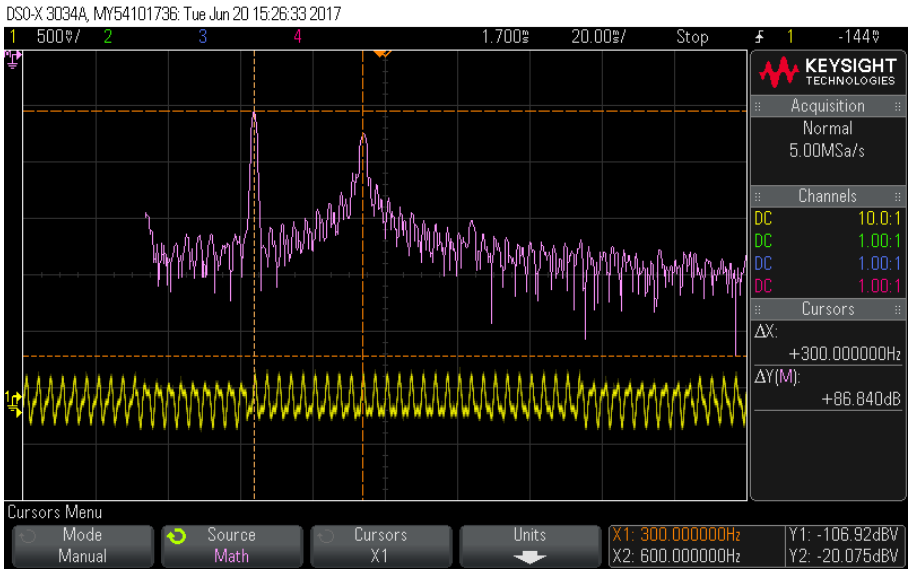
Matlab simulation:



Down-sampling: Red – input, Green – regular IIR output, Blue – IIR + 1 Octave-Up

In the Matlab simulation, the frequency has doubled, which is the Octaver effect.

Scope measurement:



The fundamental frequency at 300Hz and the up-octave at 600Hz are visible

Down-Octave (Up-Sampling)

Down-octave is similar to up-octave, only we're slowing down the speed of the read-pointer (addr2) instead of speeding it up.

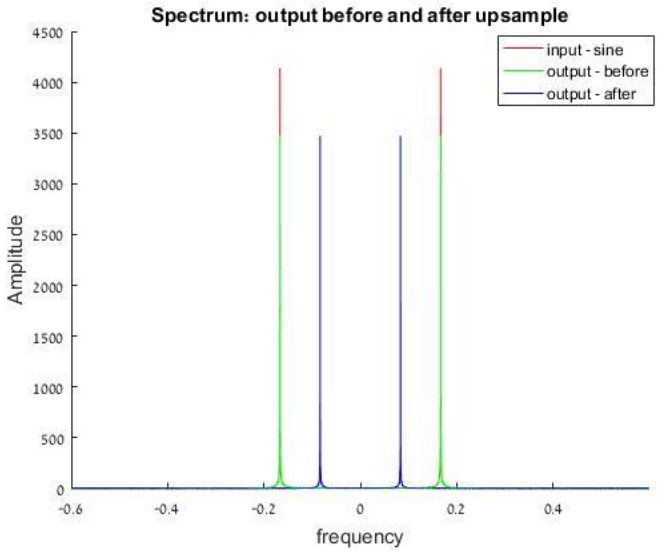
Each sample of addr1 is read twice by addr2– this is akin to up-sampling in DSP.

The math has the same principals as downsampling, only now we multiply the exponent by a factor L (L=2 in our case) - $e^{j\omega L}$ instead of dividing it. This process is also called *interpolation*, in which a new vector of samples is produced by separating each 2 samples in the original vector with zeros, and then smoothing out the output with a LPF.

In our case, the doubling of samples with the read pointer achieves a similar effect to interpolation.

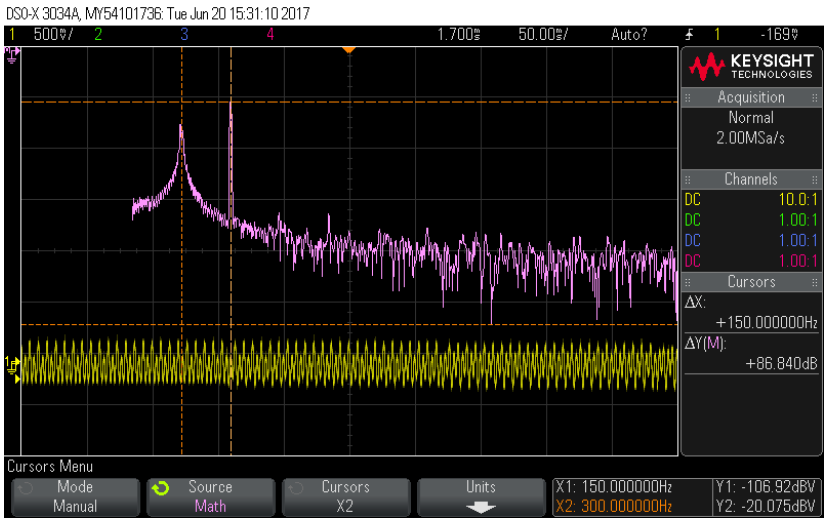
In the frequency domain, this produces a contraction of the frequency axis, and we hear it as a down-octave effect.

Below is a Matlab simulation. This time the blue plot has half the frequency of the green:



Up-sampling: Red – input, Green – regular IIR output, Blue – IIR and 1 Octave-Down

Scope measurement:



The fundamental frequency at 300Hz and the down-octave at 150Hz are visible

4. Summary and Conclusions

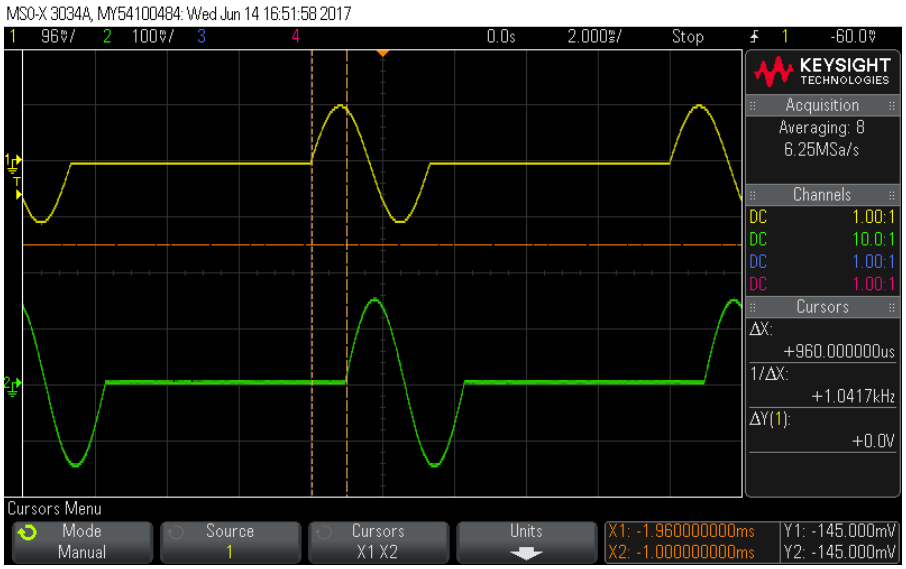
4.1 Real-Time Performance

The FPGA is renowned for its parallel-processing and low latency, and our measurements reflect these properties as well.

We generated a short burst in the signal generator, and measured the delay between the signals at the input and output of our system. The yellow signal is a direct feed from the generator, and the green is the output of the board.

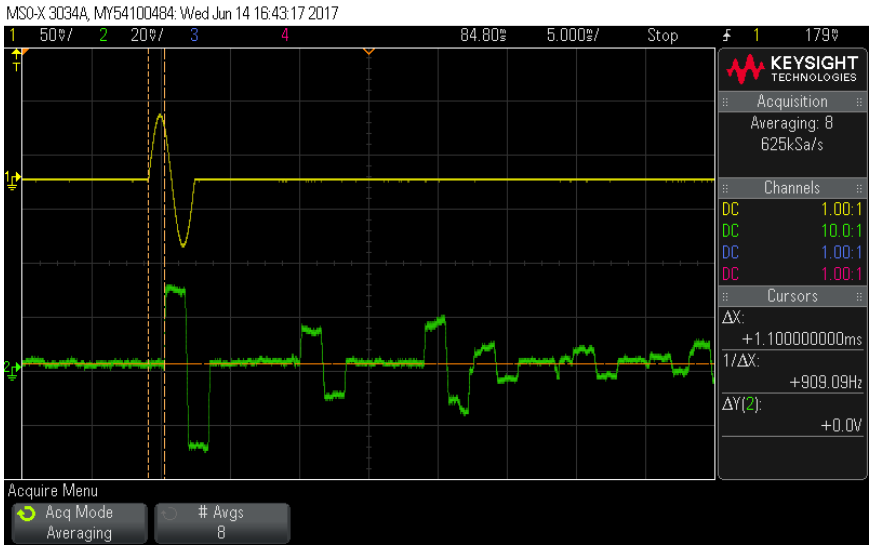
To estimate the overall latency, we took two measurements:

1. In bypass mode –no effects are activated, and the signal is affected only by the A/D, D/A, and the supporting IP blocks.



Latency in Bypass mode – no effects are activated

2. In full capacity – all the effects are activated in series:



Latency in full capacity – all effects are activated

In bypass mode, the overall latency was around 960us, and at full capacity, only 140us were added – to a total of 1.1ms latency.

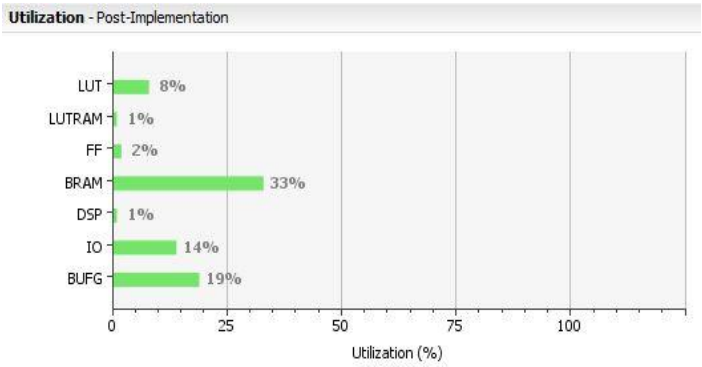
The real-time performance of the system was satisfactory, since 1.1ms is even better than some leading commercial pedals which state their latency to be "several milliseconds".

Since most (if not all) of commercial pedals use a variation of DSP and software processing, this shows that FPGA is indeed superior in terms of real time results.

4.2 FPGA Utilization

The capacity of the Zynq-7000 is very high, and we haven't use much of it. Most of the space went to the BRAMs, where the samples are stored for the Delay and Octavelo effects.

Below is the post-implementation utilization report for the project, taken from Vivado:



Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
LUT	4465	53200	8.39
LUTRAM	68	17400	0.39
FF	1911	106400	1.80
BRAM	46	140	32.86
DSP	2	220	0.91
IO	28	200	14.00
BUFG	6	32	18.75

Vivado utilization reports – graphic view and table view

4.3 Summary

Every aspect of this project required a very steep learning curve on our part. As one of our goals was to enter the world of FPGA development without any prior knowledge or experience, some extensive self-education was in order. Luckily, in our day and age, the internet provides more than enough resources on every possible topic, and Xilinx itself has an abundance of user manuals, workshops, and tutorials (in a sometimes-overwhelming quantity), to guide us.

Before diving into any relevant work, we had to first learn VHDL, understand the work relations between the PS and PL, the workflow and structure of Vivado and Xilinx SDK, and how to use the tools they provide. We learned about the AXI protocol, about creating our own IPs, and gradually became more comfortable with the tools at our disposal.

We understood the internal workings of the on-board Audio-Codec with its many registers and clocking scheme, and found a configuration which suits us best.

On the theoretical side, we learned about DAFX (Digital Audio Effects), and found a vast new world with a large community of effect-enthusiasts, and countless academic papers on every imaginable topic – from various algorithms, to implementation methods (which are in a constant state of development and improvement), physical modeling and more.

Due to time constraints (in the real-world, not on the FPGA), the steep learning curve and a lot of technical difficulties on the way, we weren't able to venture very far into this interesting world, but we hope to continue the development under more relaxed circumstances in the future.

Overall, the results were satisfying and very engaging for us and for other guitar players who tried our system, as some of the effects are well known, and some provide new and unique sound qualities. Although the learning curve for the tools we used was quite steep, we think that the effort was worth it, since we acquainted ourselves with FPGA development and DAFX theory, and created a working product combining both our knowledge and passion!

4.4 Future Work

In this project, we created a platform which can serve as a starting point for future development. The project can be endlessly expanded – from experimenting with new effects, to improving the existing algorithms and creating new ones (e.g. implementing FFT, or numeric-approximations for non-linear effects processing). The Zedboard itself can be upgraded with various PMODS (such as the rotary-encoder) for a more authentic user-experience, and the XADC can be used to incorporate external analog signals into the project (e.g. use an actual pedal for control).

5. References

[1] Udo Zolzer (Editor), 2011. DAFX Digital Audio Effects, 2nd Edition. John Wiley & Sons.

[2] Bryan Mealy, Fabrizio Tappero, 2016. Free Range VHDL. [Online]. Available: http://freerangefactory.org/

[3] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz and Robert W. Stewart. The Zynq-Book, 2014. [Online]. Available: http://www.zynqbook.com/

[4] David Te-Mao Yeh, Digital implementation of musical distortion circuits by analysis and simulation, 2009. [Online]. Available: https://ccrma.stanford.edu/~dtyeh/papers/DavidYehThesissinglesided.pdf

[5] Cardiff University. Digital Audio Effects. [Online]. Available: http://users.cs.cf.ac.uk/Dave.Marshall/CM0268/PDF/10_CM0268_Audio_FX.pdf

[6] John Michael Espinosa Durán, Pedro P. Liévano Torres, Claudia P. Rentería Mejía, Jaime Velasco Medina, 2014. Design of A Programmable Microsystem for Digital Audio Effects Using FPGAs. [Online]. Available: http://www.scielo.org.co/scielo.php?pid=S1794-12372014000200011&script=sci_arttext&tlng=pt

[7] Markus Pfaff, David Malzner, Johannes Seifert, Johannes Traxler, Horst Weber, Gerhard Wiendl, (DAFX-2007). Implementing Digital Audio Effects Using a Hardware/Software Co-Design Approach. [Online]. Available: http://dafx.labri.fr/main/papers/p125.pdf

[8] Sujit Rokka Chhetri, Bikash Poudel, Sandesh Ghimire, Shaswot Shresthamali and Dinesh Kumar Sharma, 2015. Implementation of Audio Effect Generator in FPGA. [Online]. Available: https://www.researchgate.net/publication/272386389_Implementation_of_Audio_Effect_Generator_in_FPGA

[9] Julius O. Smith III, Center for Computer Research in Music and Acoustics (CCRMA), 2007. Introduction to Digital Filters. [Online]. Available: https://ccrma.stanford.edu/~jos/filters/

[10] Julius O. Smith III, Center for Computer Research in Music and Acoustics (CCRMA), 2010. Physical Audio Signal Processing. [Online]. Available: https://ccrma.stanford.edu/~jos/pasp/

[11] Julius O. Smith III, Center for Computer Research in Music and Acoustics (CCRMA), 2011. Spectral Audio Signal Processing. [Online]. Available: https://ccrma.stanford.edu/~jos/sasp/

[12] Weijun Zhang, 2001. VHDL Tutorial: Learn by Example. [Online]. Available: http://esd.cs.ucr.edu/labs/tutorial/

[13] Avnet, 2012. Zedboard Hardware User's Guide. [Online]. Available: http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[14] Xilinx Inc, 2016. Zynq-7000 AP SoC Technical Reference Manual. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[15] Xilinx Inc. Vivado-Based Workshops. [Online]. Available: https://www.xilinx.com/support/university/vivado/vivado-workshops.html

[16] Analog Devices, 2010. ADAU1761 Audio Codec Data Sheet. [Online]. Available: http://www.analog.com/media/en/technical-documentation/data-sheets/ADAU1761.pdf

[17] Term Paper: Format of Citations and References. [Online]. Available: http://nob.cs.ucdavis.edu/classes/ecs015-2007-02/paper/citations.html