ALTERAudio

A real time audio manipulation device

Authors: Roshan Nair, Nicholas Paiva, Nicholas Saizan
Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—ALTERAudio allows musicians to add complex effects to live music using existing MIDI controllers. Current competition only supports limited effects with single axes of control. Our system is capable of layering effects and controlling them as they play. ALTERAudio brings new depth to live music.**

***Index Terms*—Altera, Audio, MIDI, Real-time, FPGA, DSP**

I. Introduction

IN today's music industry, artists are trying to find ways to distinguish themselves from their peers. With guitar pedals, and other, similar equipment having been around for decades, their uses seem to have been fully explored. In order for artists to use multiple effects they would need many of these guitar pedal like devices, which is not only clunky but also difficult to manage. But with ALTERAudio we aim to challenge these assumptions but giving artists the ability to manipulate audio in a new and exciting way. By utilizing the current MIDI equipment that all artists already have and are familiar with we aim to give them greater control over their live effects while also giving them the ability to use more effects at any given time. The best part is that ALTERAudio's effects list can easily be expanded with future updates to satisfy any artist's creative wishes.

II. Design Requirements

There are a few core requirements for this project. They come from the use case for our project as well as the capabilities of the competition. The FPGA we have chosen for this project is the Altera Cyclone V, on the Terasic DE0-CV board. The smaller footprint of this board and the capable FPGA are conducive to the following requirements.

First, we want realtime playback coming from the device. The delay through the device should not be perceptible if we want to be able to use it in a live setting. Because the minimum perceptible delay to the human ear is about 30ms, we would like the total delay through our device to be less than that. This will be measured quantitatively by using an oscilloscope to measure the onboard test points.

Secondly, the quality of the sound is important. If the output of our device is noisy or distorted, it will be difficult to play. This requirement can be enumerated both qualitatively and quantitatively. Qualitatively, we want audio to sound the same playing directly from an output device as it sounds coming from the ALTERAudio device when no effects are enabled. Any obvious differences will be perceptible by the human ear. Quantitatively, we want a flat frequency response (less than 1dB of variation across the audible range) and low noise on the input (less than 5mV). The first requirement comes from a standard in the audio industry, while the second comes from the fact that 5mV represents a few LSBs of our ADC input. We will be using a 16 bit, 48kHz, dual channel representation to encode the sound.

Third, it is necessary to support multiple effects through the pipeline to truly compete with existing solutions. The requirements and validation for this will be specific to the effect, which is explained in more detail below. We implemented the following effects:

1. Panning
2. Bit Crushing
3. Tremolo
4. Sine Amplitude Modulation
5. Delay
6. Moving Average Filter
7. Echoing
8. Chorus
9. Pitch Shifting
10. Frequency Filtering

Some of the effects follow from others. For example, Amplitude Modulation uses similar code to Tremolo. The effects listed above are roughly ordered by difficulty of implementation.

Fourth, the control of our device should be intuitive and capable of variation. The intuitiveness of our design is hard to measure quantitatively. Qualitatively, we can test this by having someone not familiar with the project play with the device. To make the operation intuitive, we emulate the way a piano is controlled. This will allow for a lot of variation in the effects as well. The device will be capable of utilizing both key velocity and pressure (Aftertouch) measurements.

Fifth, we want the packaging of the device to be neat and robust. We will accomplish this by assembling a custom PCB that will slot into our FPGA.

Finally, we want to have a frame rate to the onboard display of greater than 24FPS. 24 FPS is the standard frame rate of the movie industry, and will result in a smooth picture. We use the display to show the transformed audio signal in real time. Some of the effects, such as panning and tremolo, are very easy to see on the display. Pitch shifting and frequency filtering are not very obvious.

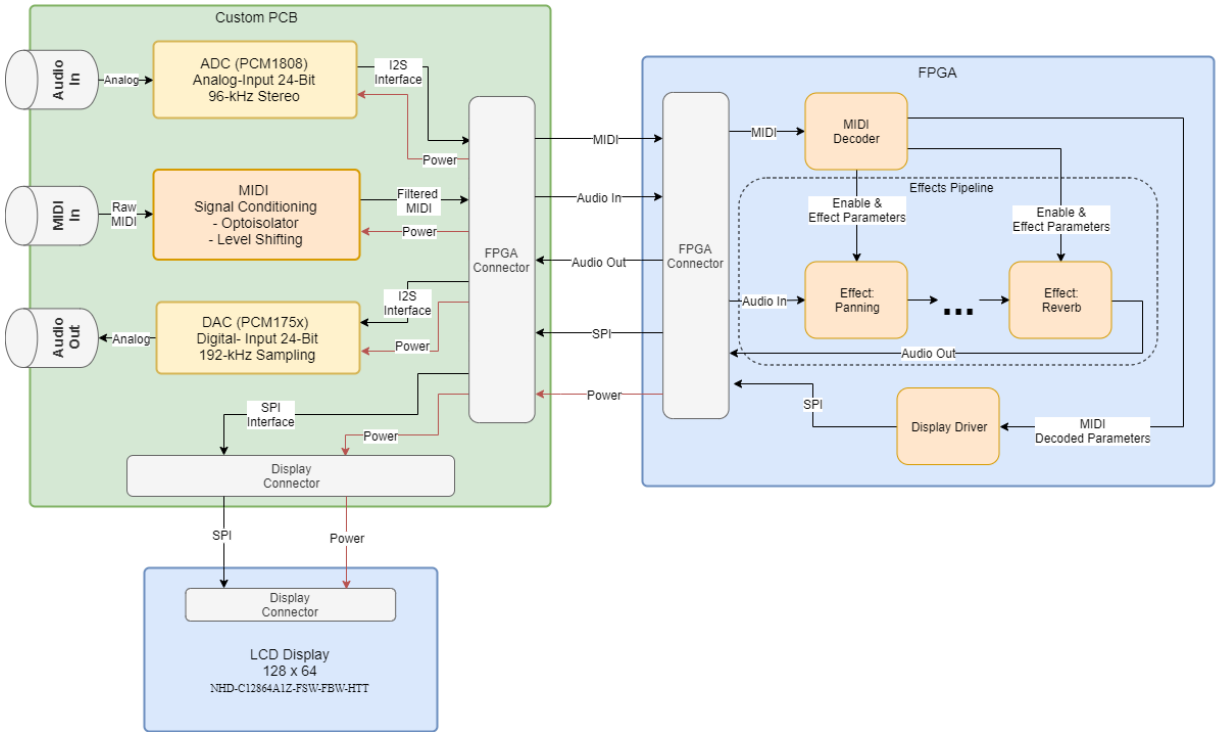


Fig. 1. Block Diagram

III. Architecture and/or Principle of Operation

The overall architecture of our DSP is a pipeline. Data is translated from the audio input into digital form by the ADC, and then passed to the first effect in the pipeline as a 16 bit two's complement value. The first effect then does its operations and transfers control to the next part of the pipeline and so on. The last effect outputs data that is converted back into an audio signal by the DAC. The MIDI controller is used to control the individual effects, and its output is interpreted by a MIDI decoder.

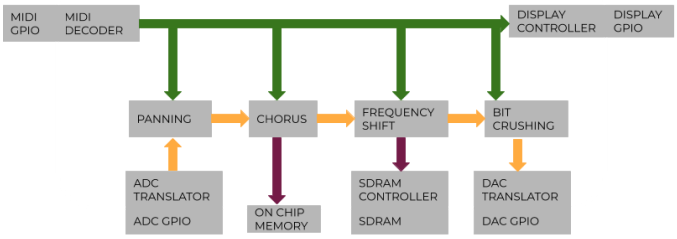


Fig. 2. Pipeline Diagram with half the effects

Each of these DSP blocks has valid/ready protocols for the input and output. Though the system deals with real-time audio, the sample rate is 48.8 kHz which is three orders of magnitude slower than the clock rate of the RTL (~50Mhz). Additionally the SDRAM and all memory handling logic is clocked at 100 Mhz. There are thus a large number of clock cycles between successive audio frame samples. This leads to the standard of the valid/ready protocol to transmit data.

As a result of this structure, each of these DSP blocks have the ability to stall in order to access memory or perform multiple reads and writes for a single audio sample. Each of these DSP blocks additionally have a set number of registers that can take in different input values which are then used to control the effects. This is a simple read into a set of registers provided by the decoder.

Some of the DSP require the use of stored audio samples in order to perform the effect. Memory reads and writes are available in two forms: block RAM and SDRAM. The block RAM is limited in size and is therefore used in Chorus, Pitch Shifting, and Frequency Filtering modules since it comparatively has a smaller memory footprint than the other effects. For the blocks such as the Echoing Effect a large number of samples are needed in order to store seconds of audio samples at the rate of 48.8 kHz. These blocks instead communicate with the SDRAM chip which provides a larger set of working memory.

A. Filtering Effect

The goal of the filtering effect is to implement an arbitrary digital filter. With this capability it is possible to implement an equalizer and other interesting effects. We considered using a Short Time Fourier Transform (STFT) to implement frequency filtering, but the inherent delay and complexity of this method was intractable. Instead we opted to implement a Finite Impulse Response (FIR) filter block. This is a time domain solution with only minimal pipeline delay. To properly implement the FIR filter we use a precompiled impulse response and fixed-point arithmetic. We decided to

use four bands for our final device. The middle bands isolate the vocals of the song quite well.

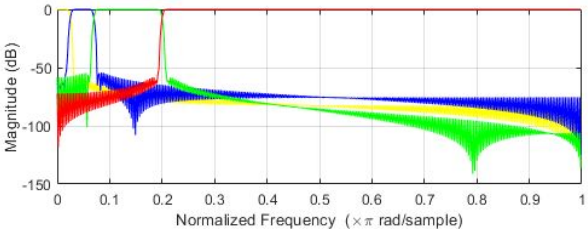


Fig. 3. Chosen Filter Bands

The filters were generated using MATLAB's FIR tools. They are then translated into an impulse response that can be convoluted with the signal in the time domain. The impulse response is converted from floating point arithmetic to a form that can be used for fixed point arithmetic by multiplying it by (2^(n-1)), where n is the desired word size of the filter data. The data is exported from MATLAB and then processed through a Python script into the Intel Hex file format so that it can be loaded into ROM on the FPGA.

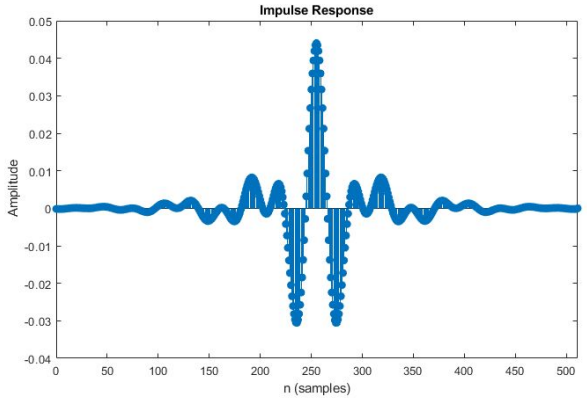


Fig. 4. Impulse Response for the blue band in Fig. 3

To generate the final audio for this effect, it is a simple matter of convolution with the audio data stream. The impulse response contains 512 points, and a two-port RAM/ROM solution is used to lower the amount of time required to calculate each data point.

B. Pitch Shifting Effect

The pitch shifting effect is used to shift the frequency content of the signal and make it sound higher or lower pitched. The method we use to implement pitch shifting is based on the implementation described in [1].

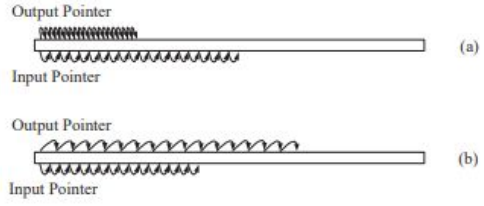


Fig. 5. Resampling Process, [1]

There are two pointers that move around a single ring buffer. The first pointer writes data at the sample rate, while the second pointer moves faster or slower depending on the desired shifting. Interpolation is used to resample the signal. If the output pointer gets too close or far from the input pointer, it jumps to a new place in the buffer. The goal is to have the output pointer never cross the input pointer.

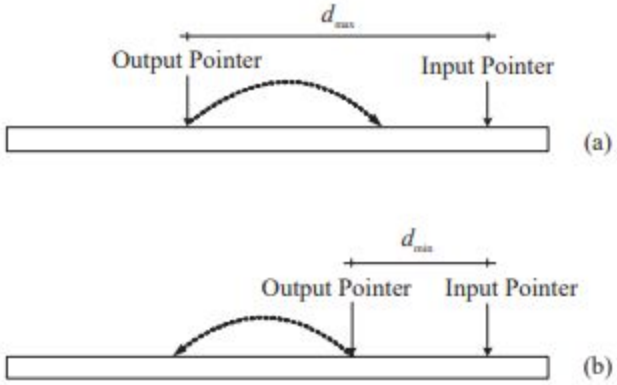


Fig. 6. Pointer Jumping, [1]

Our implementation differs from [1] in a few key ways. Firstly, we use linear interpolation instead of sinc interpolation for ease of computation. Secondly, we do not search the audio stream for the best jumping point. Thirdly, we use our filtering block to only shift the mid band frequencies. This is because our simpler implementation does not have the same quality, and lower frequency signals tend to create discontinuities when the output pointer jumps. Filtering the signal results in better performance of our implementation. Overall, our pitch shifting implementation works quite well for how simple it is.

C. Moving Average Effect

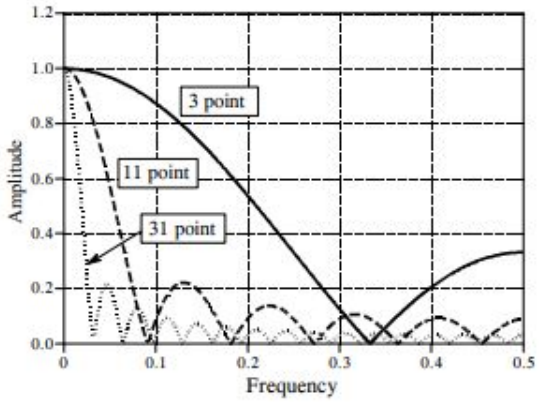


Fig. 7. Frequency Response for a Moving Average Filter [15]

The moving average effect is a special case of an FIR filter. This effect is implemented in a more efficient manner in the FPGA hardware. Whereas the FIR filtering block performs 512 multiplications and additions per sample, the moving average filter does two additions per sample. This results in a highly efficient smoothing filter. This effect is similar to a low-pass filter, but does not have the same frequency response

and bandwidth. This effect was implemented before the FIR filter block, and we decided to keep it in because its sound has a different character than the FIR lowpass filter.

D. Bit Crushing Effect

The goal of this audio effect is to uniformly provide constant noise to an audio signal such that the precision of the signal decreases, emulating an older audio device. These systems however worked natively in these low resolution audio samples which gave them their unique audio. In our 16-bit implementation this can be achieved by zeroing out the least significant bits in the audio sample (and gates with 0). In order to make this effect work better with lower volume signals, the bit crushing module tracks the envelope of the audio signal and crushes the top few bits of its magnitude. We also have a similar effect that clips off the top bits of the signal.

E. Panning Effect

Panning is when one channel of audio is at a different magnitude than the other channel of audio. Our system is stereo which results in each frame containing a left and right channel. Increasing the amplitude of one of these audio channels while decreasing the other channel will achieve this panning effect. Even at high amplitudes panning should not lose significant data resulting in clipping. Additionally this module needs to have the option of auto adjusting itself to give the illusion of 'moving' audio between the two channels.

F. Chorus Effect

The chorus effect is achieved by mixing samples with future audio samples. However, these stored samples have a delay, frequency shift, and amplitude changes. The delay is very small as it is not meant to sound like an echo. Instead, it should make a single sound be perceived as multiple sounds in unison. This has a relatively small memory footprint because it only needs a few stored samples of audio. As a result the required size to fit these samples can be placed in block RAM itself. Multipliers will be used to implement the necessary frequency shifts and amplitude changes.

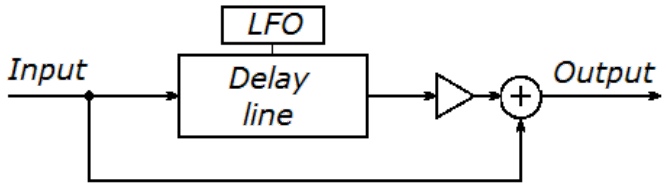


Fig. 8. Chorus effect achieved with delaying a sample, modulating the amplitude slightly with a LFO (low frequency oscillator)

G. Delay

The delay effect is, from a signal processing perspective, relatively simple. Audio comes in at one end, and comes out at the other at a later time. From an implementation perspective, it is more difficult. A relatively large amount of memory is required to implement a noticeable delay. To implement delay on our device we used the onboard SDRAM.

H. Echoing Effect

Echoing is achieved by storing a large number of successive samples and injecting them into the audio stream after a certain delay. The delay target is from 0.25 to 2 seconds. With the specifications of the system this results in a max of 2116800 bits (seconds * sample rate * number of channels * bit precision) stored for each audio frame. Since this will not fit into the block RAM this DSP module instead will need to maintain a circular buffer of samples in DRAM where the newest sample will replace the oldest. Then on every new audio frame, this will add the current sample with the stored sample.

I. Amplitude Modulation and Tremolo

This operation achieves a tremolo effect in the outputted audio stream. The volume or amplitude of the signal changes according to a certain frequency which results in a periodic vibrating sound. It is achieved by modulating the amplitude of the audio stream by a certain frequency. Within a given period, each sample of the audio stream is scaled according to a reference sinusoid. This block utilizes the on chip multipliers of the FPGA to multiply each sample by a different scaled value on each sample within the sinusoid period.

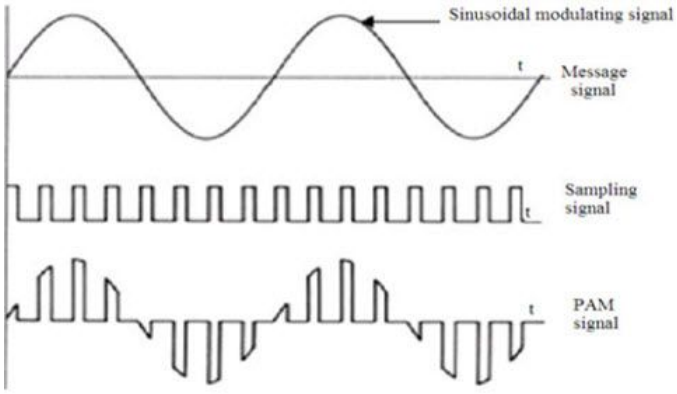


Fig. 9. Amplitude modulation with a pulse train being used as the sample signal. [14]

IV. Testing and Verification

Because we have so many different effects, testing is very important. If any of the effects malfunction, they can cause other effects to malfunction as well. To combat this, we've developed a rigorous testing plan that involves simulation and data collection.

A. Audio Interface

The audio interface itself is easy to test. Trivially, we can use the interface as a pass through and perform a manual, auditory inspection. We can also do more rigorous qualitative tests to evaluate the "flatness" of the frequency response and the noise on the input. To measure the frequency response we can input sine waves generated by a function generator, and measure the amplitude of the sine waves on the output. To

measure the noise in the circuit we can watch the input jitter when there is no audio connected.

We measured the frequency response of our audio passthrough, and found that it is satisfiably flat. The gain is quite flat in the most important audio range of 1-10kHz, and the peak in the higher frequencies does not degrade audio quality. When listening to the puretones we were measuring with, we could barely hear sounds above 14kHz.

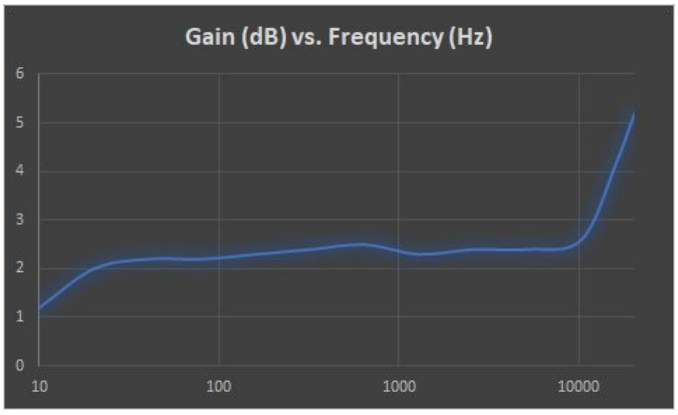


Fig. 10. Frequency Response of our Audio Passthrough

B. MIDI Decoder

The MIDI interface is very well documented online. Using all of this documentation we can develop an FSM-D which we expect to decode the incoming MIDI data properly. To verify that we are receiving pitch parameter correctly, we can display which key is read by the MIDI decoder on the onboard hex displays on the FPGA. Testing for key pressed, key released, velocity, and aftertouch can be testing similarly. Because velocity and aftertouch will be dependant on how hard we are pressing, it will also be worth our time to verify the decoding is working in a testbench before we test in synthesized hardware.

We've thoroughly tested the MIDI decoder by comparing the decoded data to hand-decoded samples on the oscilloscope. Additionally, we have tested it by using it to control the other effects and ensuring that it controls them properly. From our testing, we have determined that we have upwards of 90% accuracy on our MIDI decoder module.

C. Filtering Blocks

To test our filtering blocks, we mostly completed qualitative testing. The filters are effective enough that this sufficed. We did not collect any data for the FIR filter blocks, but took an informal frequency response by listening for cutoff frequencies with our ears. The cutoff frequencies matched our expectations, and the rolloff is very sharp. We also tested our FIR filter blocks by playing music and isolating the high, mid, and low ranges of the track. From our qualitative testing, the FIR filter blocks work phenomenally.

We tested the moving average block more formally. We tested this filter with a pure square wave on the input. When the moving average is applied to this input it produces a triangle wave of a similar frequency, as expected. When we measured the FFT of these different waves, the higher frequency harmonics of the original wave are reduced significantly while the lower frequency harmonics are preserved. This is exactly as desired. We also tested the moving average filter qualitatively, and it clearly attenuates higher frequency components.

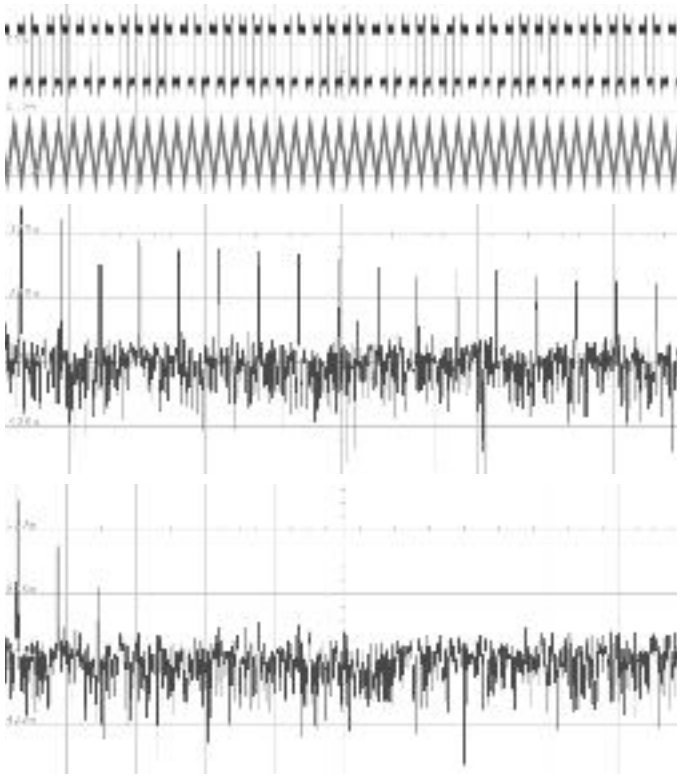


Fig. 11. Moving Average Filter Waveforms (Top: Square wave input and averaged output. Bottom: Input spectra vs. Output spectra)

D. Pitch Shifting

Pitch shifting is another effect that is easy to test qualitatively. We tried audio from various artists and pitch shifted the tracks both higher and lower. The effect of the pitch shifting is quite obvious to the ear. This effect also shows up quite well on a spectrograph. We recorded audio while switching between high and low pitch shifts, and then plotted a spectrograph of that recording. When downshifting is enabled, there is much more energy in the lower frequencies. When downshifting is enabled, we can see more energy in the higher frequencies. This is exactly as desired.

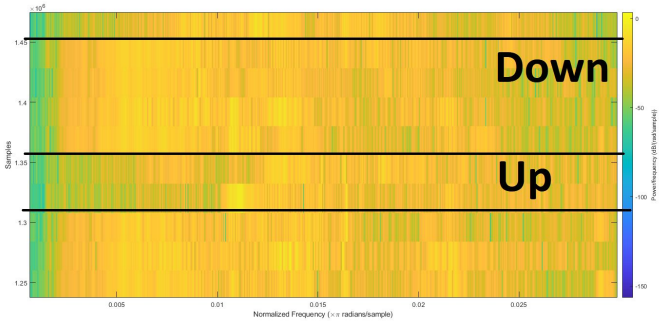


Fig. 12. Pitch Shifting Frequency Spectra

E. *Bit Crushing*

Bit crushing is easily verifiable using an automated testbench which inputs various digital audio samples with bit crushing enabled. The output can be checked to ensure that only the lower bits of the signal were removed and that the remaining bit information is left intact. We have also verified bit crushing by capturing it on an oscilloscope. The bit crushed wave forms, on the left, are indeed blockier than the uncrushed waveforms on the right.

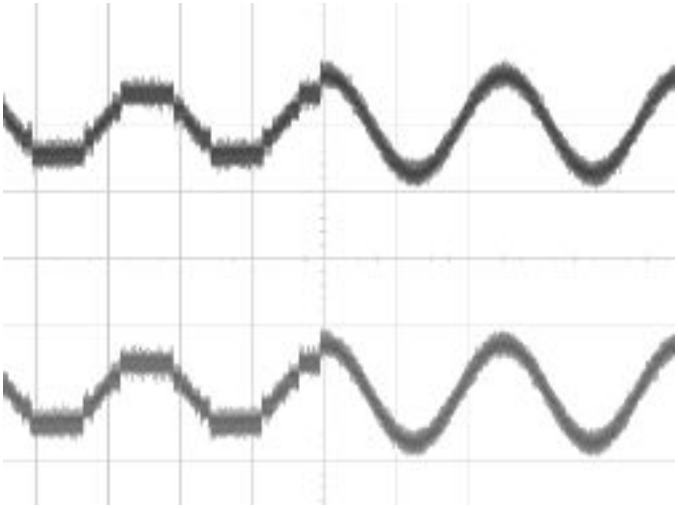


Fig. 13. Bit Crushing Effect Waveform (Left: Crushed, Right: Original)

F. *Panning Effect*

Similarly to bit crushing, panning can be verified digitally in our testbench. To do this we will send in audio data and a panning parameter at random. Then on the output we will record the signal being output on both the left and right audio channels. Simultaneously we can replicate the multiplication of audio in and panning parameter to obtain a reference output value. By comparing the reference to the actual we can ensure our module is behaving correctly.

Waveforms of our panning effect in use are pictured below. To capture this picture we passed in a pure sinusoidal tone of equal magnitude to both channels. The panning turns on in roughly the middle of the capture. The increase and decreases in signal magnitude are clearly visible in this picture.

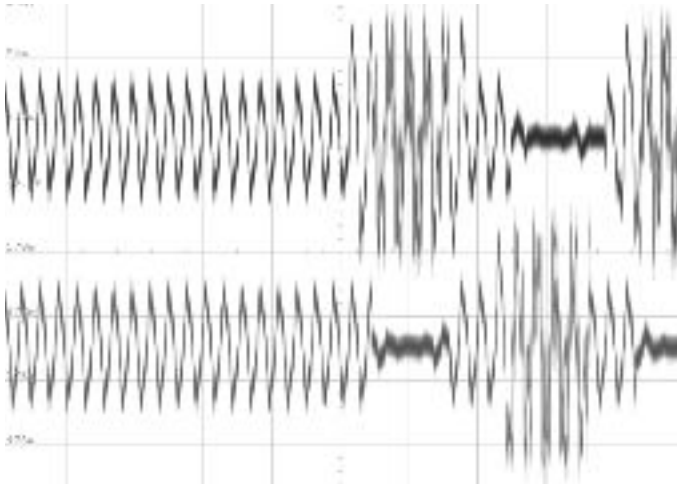


Fig. 14. Panning Effect Waveforms

G. *Chorus Effect*

Testing the chorus effect will involve carefully choosing a testing input which enabled us to get a sense of how the system is working. By inputting a signal similar to an impulse response we can view the output waveform to confirm that there are indeed multiple ‘voices’ each with a different associated delay as compared to the original input ‘voice’.

H. *Delay / Echoing Effects*

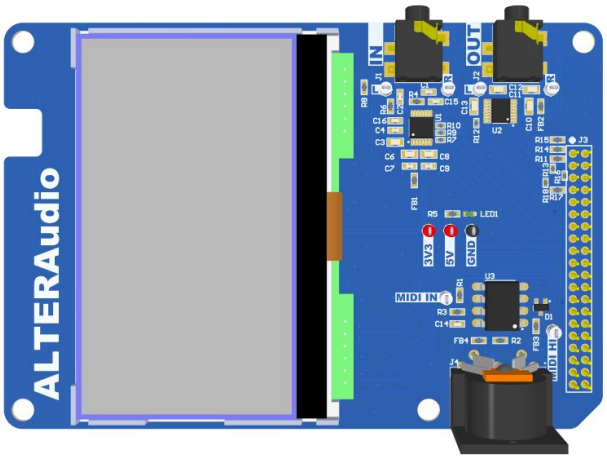
We tested the delay and echoing Effect in various ways. The first testing method we used is subjective testing. We played various non-periodic songs, and listened to the results. Delayed audio comes in at the prescribed time, overlayed with the original audio, for a mind-bending effect.

We also tested delay and echo on the oscilloscope by playing bursts of sound and watching for their replication later in time. We verified that the echoes come in at the correct time. Pictured below is a two second delay.

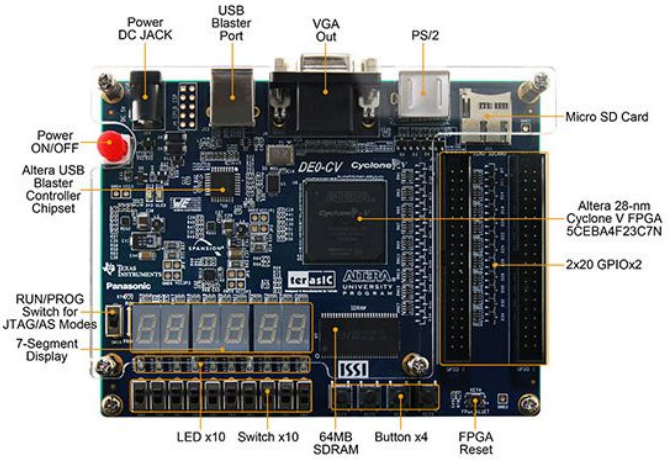
Fig. 15. Delay Effect Waveforms

V. System Implementation and Trade Study

Our system is composed of two main components. The custom PCB and the DE0-CV FPGA board. The custom PCB houses all of the circuitry for the audio system, the MIDI interface, and the display. The PCB has various test points to make the validation process easier. To make connection to the FPGA simple and robust, the PCB is designed to replace the acrylic cover that comes with the dev board. Designing a PCB took more effort than a breadboard circuit, but it will be cleaner and easier to debug. The FPGA implements the custom DSP that enables our effects. More detailed discussions follow below.



(a)



(b)

Fig. 16. System picture. (a) Main PCB. (b) DE0-CV FPGA.

A. *Audio Subsystem*

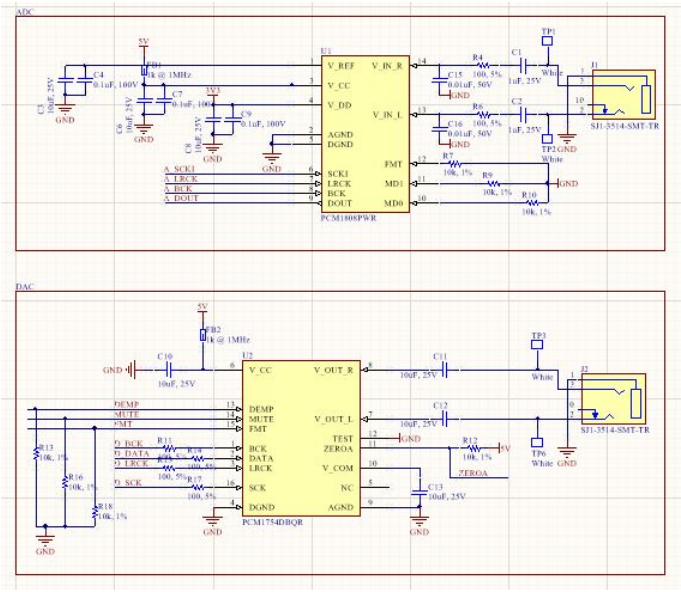


Fig. 17. Circuit diagram for the ADC, DAC, and audio connectors

For the audio in/out of the project, we decided to use some specialized application ADC/DACs from TI. The chips we used for the ADC and DAC are the PCM1808PWR and PCM1754DBQR respectively. We chose these chips over others because they are cheap, well-packaged, and specifically meant for audio. They both support dual channel 24 bit audio at high sampling frequencies, but we plan on only using 16 bits at about 48 kHz. AC decoupling capacitors block any DC biases from travelling through the audio lines. The chips connect to the FPGA through an I^{2}S interface. This interface was chosen because it is the interface dictated by the chosen ADC and DAC.

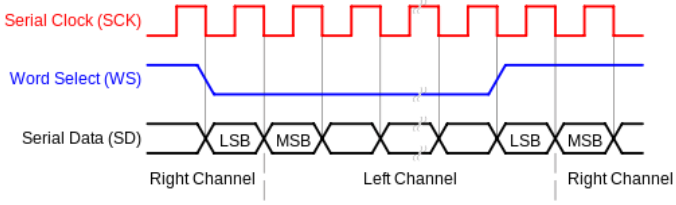


Fig. 18. I^{2}S Timing Waveforms [9]

I^{2}S is a simple 3 wire interface specifically meant for sound [5]. Despite the similarity in name, I^{2}S is more similar to SPI than I^{2}C. I^{2}S has minimal overhead, which is perfect for our application. Data is sampled on the rising edge of the clock, and the word select line corresponds to the different channels of audio.

B. *MIDI Interface*

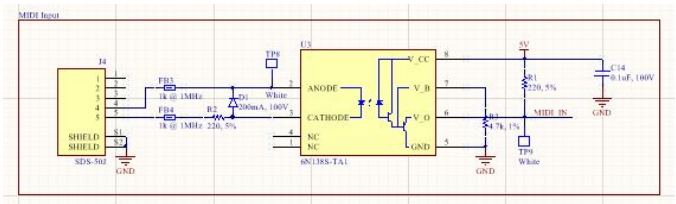


Fig. 19. MIDI connector and optoisolator

For the MIDI Interface, we are using a standard optoisolator circuit to isolate the FPGA from the controller. This isolation is important to protect against ground loops and DC biases. A diode on the LED side of the optoisolator protects the circuit against reverse polarity connections. The MIDI protocol is current loop based. After the input signal passes through the optoisolator, the output is basically UART. Commands are sent as three bytes. The first byte indicates the command, and the next two are data related to the command. We chose MIDI over other protocols (such as USB-MIDI, Bluetooth, etc.) because it is simple, ubiquitous, and robust.

C. *Display*

Fig. 20. NHD-C12864A1Z-FSW-FBW-HTT [6]

The display we've chosen is from Newhaven Display. It is a 128x64 pixel monochromatic screen. It has a simple SPI interface for control. We use it to display real-time audio information for each channel. It ended up working quite well, although the refresh rate of the display leaves a bit to be desired. If we attempt to update the display too quickly it ends up looking blurred.

D. *Enclosure/Packaging*

Since ALTERAudio is intended to be a simple and convenient solution for artists to replace their many audio manipulation devices it's important that we package it in a compact and desirable manner. This is why we've designed our PCB to interface nicely with the FPGA and the display module. As you can see in Figure 4, the mounting holes of the PCB are aligned perfectly with the mounting holes of the FPGAs cover. And the rightmost GPIO box header is aligned with the male pins on the right side of the PCB. This is so that the PCB can mount snugly on the FPGA. Furthermore, the display is seated nicely on top of the whole package. Since the PCB receives power directly from the FPGA, this means we only need 1 power cable for the whole system. Connectors for MIDI and Aux In/Out are placed conveniently around the board as well.

VI. Project Management

A. *Schedule*

NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE
1	**Project Conception and Initiation**					
1.1	BOM (Component Selection)	Nick	9/16/19	9/20/19	4	100%
1.2	Basic FPGA IO and SDRAM Tests	Roshan	9/16/19	9/19/19	3	100%
1.3	PCB Schematics	Nick P.	9/18/19	9/23/19	5	100%
1.4	PCB Layout	Nick P.	9/18/19	9/23/19	5	100%
1.5	MIDI Protocol Research	Nick S.	9/19/19	9/23/19	4	100%
1.6	Panning Effect Research	Roshan	9/19/19	9/23/19	4	100%
1.7	Chorus Effect Research	Nick S.	9/23/19	9/26/19	3	100%
1.8	Frequency Filtering Effect Research	Nick P.	9/23/19	9/26/19	3	100%
1.9	Bit Crushing Effect Research	Roshan	9/23/19	9/26/19	3	100%
2	**Project Integration**					
2.1	Audio Jack/ADC to PCB	Nick P.	9/27/19	10/3/19	6	100%
2.2	Speaker/DAC to PCB	Nick S.	9/27/19	10/3/19	6	100%
2.3	FPGA IO to PCB	Roshan	9/27/19	10/3/19	6	100%
2.4	MIDI to PCB	Nick S.	10/4/19	10/8/19	4	100%
2.5	FPGA Bypass Implementation	Roshan	10/4/19	10/7/19	3	100%
2.6	MIDI Decoder Implementation	Nick S.	10/8/19	10/14/19	6	100%
2.7	Audio Stream Integration	All	10/8/19	10/17/19	9	100%
3	**Base Effect Implementation**					
3.1	Panning Effect Implementation	Roshan	10/14/19	10/18/19	4	100%
3.2	Chorus Effect Implementation	Nick S.	10/14/19	10/18/19	4	90%
3.3	Frequency Filtering Effect Implementation	Nick P.	10/14/19	10/18/19	4	100%
3.4	Bit Crushing Effect Implementation	Roshan	10/21/19	10/23/19	2	100%
4	**Core Effect Implementation**					
4.1	Amplitude Modulation Effect Research/Implementation	Nick S.	10/21/19	10/30/19	9	100%
4.2	Echoing Effect Research/Implementation	Roshan	10/21/19	10/30/19	9	90%
4.3	Display Implementation	Nick P.	10/21/19	10/30/19	9	100%
4.4	Reverb Effect Research/Implementation	Roshan	10/31/19	11/14/19	14	80%
4.5	Pitch Adjustment Effect Research/Implementation	Nick P.	10/31/19	11/14/19	14	100%
4.6	Autotune Adjustment Effect Research/Implementation	Nick P.	10/31/19	11/14/19	14	50%
4.7	Vocoder Adjustment Effect Research/Implementation	Nick S.	10/31/19	11/14/19	14	30%
5	**Extra**					
5.1	System Wiring/Construction	All	11/14/19	11/20/19	6	100%
5.2	Slack	All	11/21/19	11/29/19	8	100%

Fig. 21. Project Gantt Chart

B. Team Member Responsibilities

Each of the separate subsystems were split among each team member such that it aligned with their area of concentration and strength.

Nicholas Paiva's primary responsibility is in the design of the PCB. This is split among three parts which include component selection, schematic design, and finally layout design. Nicholas is also responsible for the screen interface and protocol RTL as well as the research/design of the Frequency Filtering, Pitch Adjustment, and Averaging Effects.

Nick Saizan's responsibility lies in primarily the MIDI interface systems which talk to the MIDI ports as well as all the DSP blocks in our pipeline. He is in charge of ensuring that the various user inputs such as pad velocity and key aftertouch are correctly transmitted to all the DSP blocks which is crucial to configuring them. Additionally he also shares responsibility managing the on chip blocks in the FPGA such as block ram and multipliers. For effects he is responsible for Chorus, and Tremolo/Amplitude Modulation.

Roshan Nair's responsibility resides the organization of the DSP pipeline, the verification testbench, and SDRAM management. The overall valid/ready protocol determines how each of the DSP blocks talk to each other in the pipeline. Additionally he is in charge of providing a preliminary way of testing these effects through simulation. For effects he is responsible for Panning, 8 bit, and Echoing.

All members share responsibility for PCB to FPGA integration through the GPIO interfaces.

C. BOM

See below for the full BOM. As of the current timeline in this project, all components have the intention of being used with 1-3 extras for replacement in case the original components fail throughout testing. The PCB specific BOM is on the following page. Overall we were under our budget by about $300.

Index	Item	Cost	Quantity
1	Keyboard	$179.00	1
2	MIDI Cable	$5.99	2
3	FPGA	$150.00	1
4	PCB	$100.00	1

Fig. 22. Overall BOM

Quantity	Digi-Key part Number	Unit Cost	Total
7	587-2985-1-ND	$ 0.16	$ 1.56
2	CP1-3514SJCT-ND	$ 1.30	$ 2.60
1	CP-2350-ND	$ 1.86	$ 1.86
1	311-0.0GRCT-ND	$ 0.10	$ 0.10
1	311-4.70KHRCT-ND	$ 0.10	$ 0.10
7	311-10.0KLRCT-ND	$ 0.02	$ 0.15
1	296-26307-1-ND	$ 2.14	$ 2.14
1	296-26302-1-ND	$ 2.38	$ 2.38
1	NHD-C12864A1Z-FSW-FBW-HTT-ND	$ 22.69	$ 22.69
4	445-8672-1-ND	$ 0.10	$ 0.40
1	MMBD1201CT-ND	$ 0.21	$ 0.21
1	160-1446-1-ND	$ 0.28	$ 0.28
3	RHM220DCT-ND	$ 0.10	$ 0.30
6	RHM100DCT-ND	$ 0.10	$ 0.60
1	SAM9315-ND	$ 4.78	$ 4.78
4	1276-6807-1-ND	$ 0.15	$ 0.60
3	1276-1102-1-ND	$ 0.10	$ 0.30
2	445-5662-1-ND	$ 0.10	$ 0.20
6	36-5002-ND	$ 0.35	$ 2.10
1	36-5001-ND	$ 0.35	$ 0.35
2	36-5000-ND	$ 0.35	$ 0.70
1	160-1798-1-ND	$ 0.89	$ 0.89

Fig. 23. PCB BOM

D. Tools

a) PCB design - Altium Designer
b) FPGA simulation - VCS
c) FPGA synthesis - Quartus
d) Test data generation - Python

E. Risk Management

The primary risks throughout our project come in the form of component failure, PCB turnaround time, integration debugging, and RTL debugging. Component failure can occur if any of the shipped components are faulty or stop working during testing. To counteract this we ordered replacements alongside the originals for backup. PCB turnaround also is another risk factor as though it is estimated to be around a week, it could take longer due to external factors and as such we made sure that other tasks not directly dependent (such as rtl design and effect research) were scheduled during this turnaround time period. Extra slack time was placed in our schedule for integration debugging as any errors in our communication protocols will bring the entire system down. In anticipation of RTL bugs in our DSP blocks a rigorous testbench was developed beforehand so that preliminary simulation can catch the majority of the initial bugs rather than finding them in the synthesized hardware.

VII. Summary

Overall, our project went quite well. We were able to meet most of the design specifications and implement most of the effects we sought out to do. We are especially happy with the user interface of our project because it is quite intuitive and easy to use. The effects that we implemented are interesting and fun to play with, and they can be combined and layered in various ways to make new effects.

One of the limits of our system is the fact that there is only one audio input port. With a second audio input port there are more interesting effects we could do with combining the two tracks.

Another limit of our system is the lack of a Fourier Transform block. This did not seem difficult at first, but the more we read about it the more obvious it became that implementing the Fourier Transform in hardware is a monumental task. We were able to get around this problem with some clever time domain implementations, but it would have been nice to have this capability on our device.

Finally, we did not have time to implement every effect we wanted to. Some effects, such as the vocoder, were more complicated than we thought they would be. Others, such as autotuning, did not make much sense for our platform in its current state. Without a second audio input, what do you autotune to? These are design considerations that were not obvious until it was too late to adjust the architecture of our system to accommodate them.

A. Future Work

We do not plan on working on this system beyond this semester. This is primarily because we will not be able to keep two of the most integral parts of our project: the FPGA and the keyboard. We could potentially purchase them later on our own, but there does not seem to be much reason for that. If we did keep working on this system, we would add a second audio port to accommodate a microphone. We would also add more effects.

B. Lessons Learned

We learned quite a few lessons about project management and design throughout this semester.

First, audio is not trivial. There are a lot of considerations about audio that are not obvious at first that can greatly affect a system. If the magnitude of the signal is not handled properly, effects might have different results when a song is played at different volumes.

Second, always understand the format of your data before you use it. We were not initially aware that our audio data from the ADC is in two's complement format. This is actually never mentioned anywhere in the ADC datasheet. We assumed it was unsigned, and this resulted in a few odd problems until it was resolved.

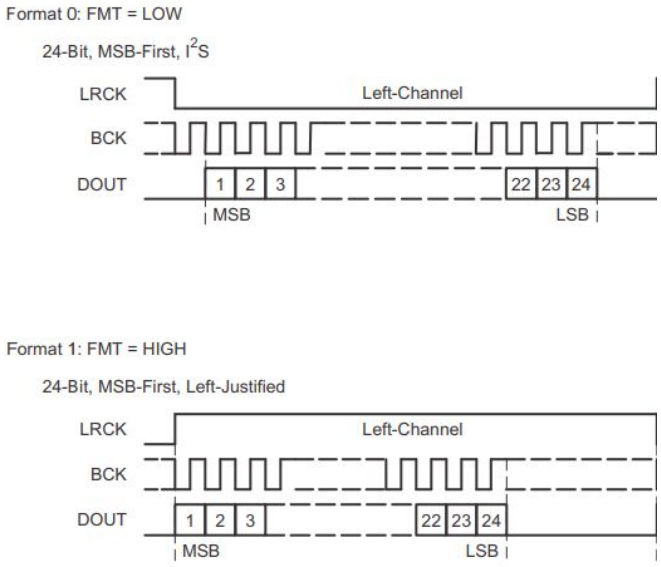


Fig. 24. The troublesome datasheet section

Third, always read datasheets carefully. We did not read the datasheet for our ADC carefully enough, and as a result did not realize that the first bit on the DOUT line for the I2C protocol is a junk bit. We ended up with this junk bit as the most significant bit of our audio data.

Fourth, always run Design Rule Checks (DRC). If you rush through changes to your design and forget to run DRC, you will probably end up with obvious and avoidable mistakes. The best way to always remember to run your DRC is with a checklist you go through before ordering.

Fifth, don't underestimate the difficulty of using IP blocks. We spent a whole day nonstop trying to get the FFT IP block in Quartus to work to no avail. Make sure to budget enough time for setting them up, and don't depend on them working. Always have a backup plan.

Sixth, don't spend too much time trying to solve any one problem. This ties into the last point. If we hadn't spent so long on trying to get the FFT to work, perhaps we could have implemented more effects. Wasting time hitting a wall makes the project suffer.

Finally, don't underestimate how much time integration takes. We spent the first half of the semester designing our board and writing drivers, and didn't get to the interesting part of writing effects until much later than we planned. Integration is often overlooked, but incredibly important.

References

[1] A. Haghparast, "Real-Time Pitch-Shifting of Musical Signals By a Time-Varying Factor", 2007. Available: Semantic Scholar, https://www.semanticscholar.org/paper/REAL-TIME-PITCH-SHIFTING-OF-MUSICAL-SIGNALS-BY-A-Haghparast-Penttinen/87fcd0d6e5ae2eefc208f054e871119af6e79fa4

[2] G. Slade, "The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation," 2013. Available: MIT, http://web.mit.edu/6.111/www/f2017/handouts/FFTtutorial121102.pdf

[3] A. Saeed, M. Elbably, G. Abdelfadeel, and M. I. Eladawy, "Efficient FPGA implementation of FFT/IFFT Processor," 2009. Available: Semantic Scholar, https://pdfs.semanticscholar.org/0726/8155508a66c45e061462beacb7d81a4e1d69.pdf

[4] "Window Function," Wikipedia.org: https://en.wikipedia.org/wiki/Window_function

[5] "I2S", Wikipedia.org: https://en.wikipedia.org/wiki/I%C2%B2S

[6] "Newhaven Display," Digikey.com: https://www.digikey.com/product-detail/en/newhaven-display-intl/NHD-C12864A1Z-FSW-FBW-HTT/NHD-C12864A1Z-FSW-FBW-HTT-ND/3767469

[7] D. Vandenneucker, "MIDI Tutorial for Programmers," 2012. Available: https://www.cs.cmu.edu/~music/cmsip/readings/MIDI%20tutorial%20for%20programmers.html

[8] D. Brink, "David's MIDI Spec," 1995. Available: cs.cmu.edu, https://www.cs.cmu.edu/~music/cmsip/readings/davids-midi-spec.htm

[9] By wdwd - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=16579640

[10] "Chorus Guitar Effects," Available: https://www.hobby-hour.com/guitar/chorus_effects.php

[11] Digilent, "Waveform Tools...The Spectrum Analyzer," 2016. Available: https://blog.digilentinc.com/waveforms-tools-with-the-ad2-and-eeboard-the-spectrum-analyzer/

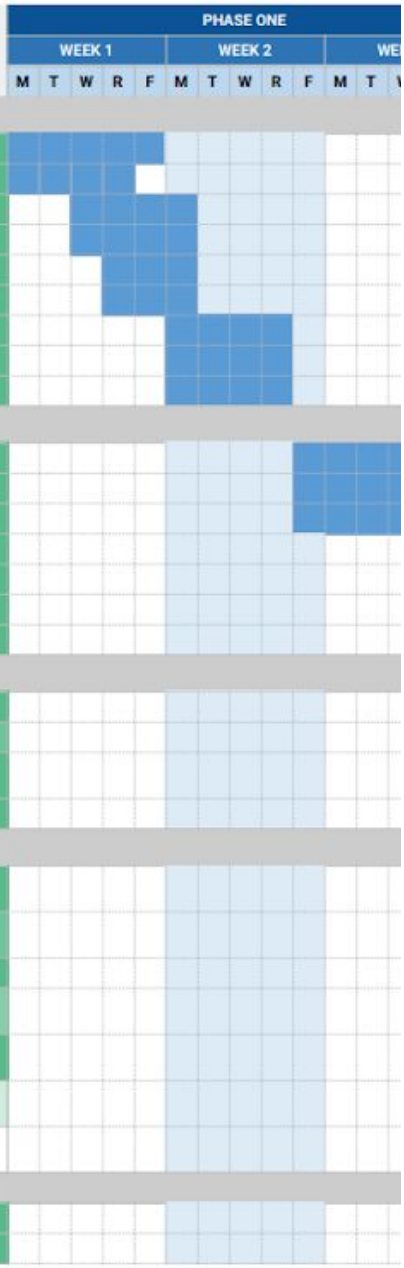
[12] S. Carroll, G. Mirza, J. Talmage, "PitchShifter," 2017. Available: https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/s2017/jmt329_swc63_gzm3/jmt329_swc63_gzm3/PitchShifter/index.html

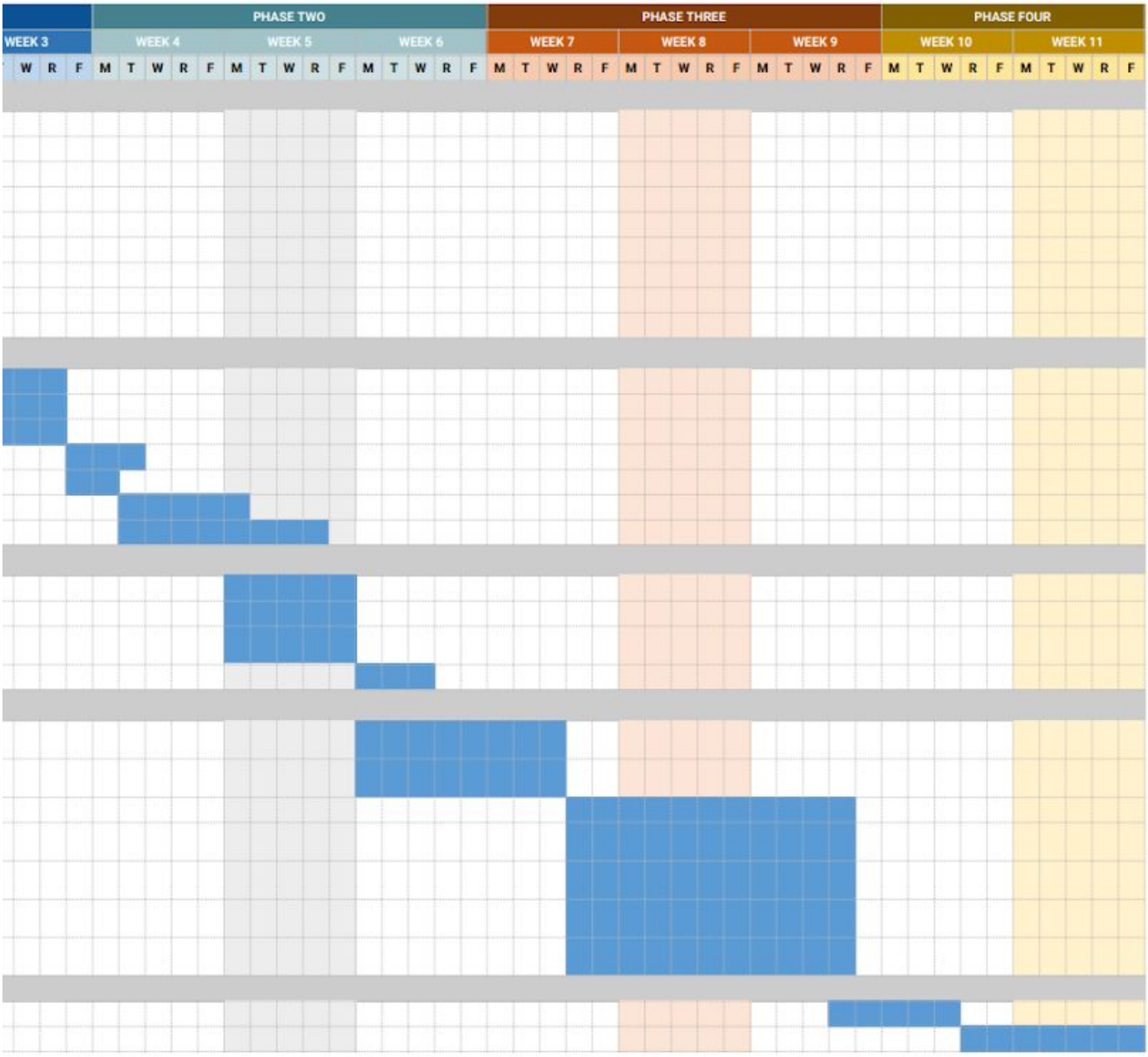
[13] "Synchronize Sports Broadcasts with Television," 2016. Available: https://www.fountainware.com/Products/AudioDelay/index.htm

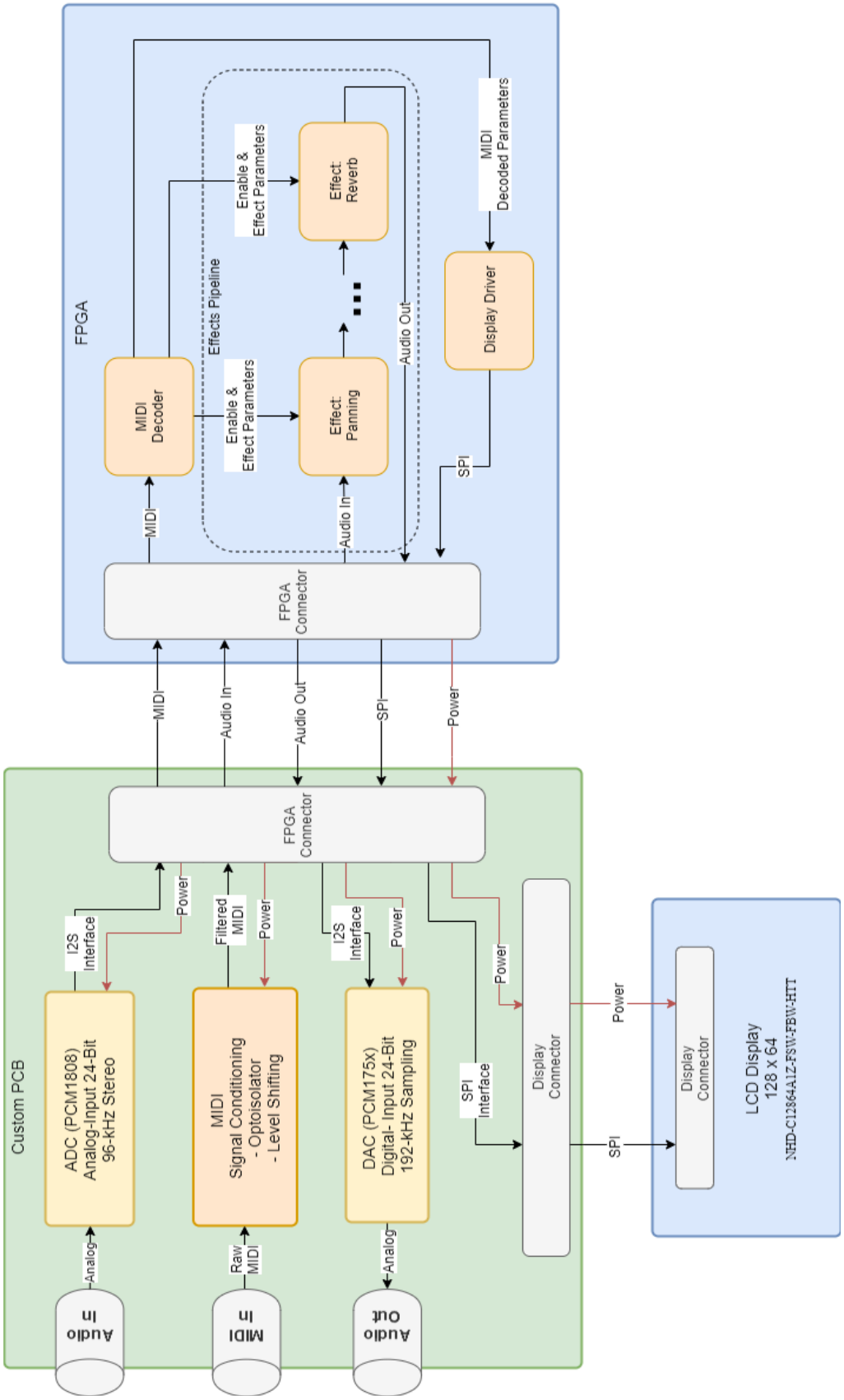
[14] ELPROCUS, "Pulse Amplitude Modulation," Available: https://www.elprocus.com/pulse-amplitude-modulation/

[15] Analog.com, "Moving Average Filters," Available: https://www.analog.com/media/en/technical-documentation/dsp-book/dsp_book_Ch15.pdf

NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE
1	**Project Conception and Initiation**					
1.1	BOM (Component Selection)	Nick	9/16/19	9/20/19	4	100%
1.2	Basic FPGA IO and SDRAM Tests	Roshan	9/16/19	9/19/19	3	100%
1.3	PCB Schematics	Nick P.	9/18/19	9/23/19	5	100%
1.4	PCB Layout	Nick P.	9/18/19	9/23/19	5	100%
1.5	MIDI Protocol Research	Nick S.	9/19/19	9/23/19	4	100%
1.6	Panning Effect Research	Roshan	9/19/19	9/23/19	4	100%
1.7	Chorus Effect Research	Nick S.	9/23/19	9/26/19	3	100%
1.8	Frequency Filtering Effect Research	Nick P.	9/23/19	9/26/19	3	100%
1.9	Bit Crushing Effect Resarch	Roshan	9/23/19	9/26/19	3	100%
2	**Project Integration**					
2.1	Audio Jack/ADC to PCB	Nick P.	9/27/19	10/3/19	6	100%
2.2	Speaker/DAC to PCB	Nick S.	9/27/19	10/3/19	6	100%
2.3	FPGA IO to PCB	Roshan	9/27/19	10/3/19	6	100%
2.4	MIDI to PCB	Nick S.	10/4/19	10/8/19	4	100%
2.5	FPGA Bypass Implementation	Roshan	10/4/19	10/7/19	3	100%
2.6	MIDI Decoder Implementation	Nick S.	10/8/19	10/14/19	6	100%
2.7	Audio Stream Integration	All	10/8/19	10/17/19	9	100%
3	**Base Effect Implementation**					
3.1	Panning Effect Implementation	Roshan	10/14/19	10/18/19	4	100%
3.2	Chorus Effect Implementation	Nick S.	10/14/19	10/18/19	4	90%
3.3	Frequency Filtering Effect Implementation	Nick P.	10/14/19	10/18/19	4	100%
3.4	Bit Crushing Effect Implementation	Roshan	10/21/19	10/23/19	2	100%
4	**Core Effect Implementation**					
4.1	Amplitude Modulation Effect Research/Implementation	Nick S.	10/21/19	10/30/19	9	100%
4.2	Echoing Effect Research/Implementation	Roshan	10/21/19	10/30/19	9	90%
4.3	Display Implementation	Nick P.	10/21/19	10/30/19	9	100%
4.4	Reverb Effect Research/Implementation	Roshan	10/31/19	11/14/19	14	80%
4.5	Pitch Adjustment Effect Research/Implementation	Nick P.	10/31/19	11/14/19	14	100%
4.6	Autotune Adjustment Effect Research/Implementation	Nick P.	10/31/19	11/14/19	14	50%
4.7	Vocoder Adjustment Effect Research/Implementation	Nick S.	10/31/19	11/14/19	14	30%
5	**Extra**					
5.1	System Wiring/Construction	All	11/14/19	11/20/19	6	100%
5.2	Slack	All	11/21/19	11/29/19	8	100%




PHASE TWO
PHASE THREE
PHASE FOUR
WEEK 3
WEEK 4
WEEK 5
WEEK 6
WEEK 7
WEEK 8
WEEK 9
WEEK 10
WEEK 11
W R F M T W R F M T W R F M T W R F M T W R F M T W R F M T W R F M T W R F M T W R F
M T W R F M T W R F M T W R F M T W R F M T W R F M T W R F M T W R F M T W R F


FPGA
MIDI Decoder
Enable & Effect Parameters
Effects Pipeline
Effect: Panning
Effect: Reverb
Audio Out
MIDI Decoded Parameters
Display Driver
SPI
MIDI
Audio In
FPGA Connector
Custom PCB
I2S Interface
Power
Filtered MIDI
ADC (PCM1808) Analog-Input 24-Bit 96-kHz Stereo
MIDI Signal Conditioning - Optoisolator - Level Shifting
DAC (PCM175x) Digital- Input 24-Bit 192-kHz Sampling
SPI Interface
Display Connector
LCD Display 128 x 64
NHD-C12864A1Z-FSW-FBW-HTT
Audio In
MIDI In
Audio Out
Analog
Raw MIDI