Final Report
FPGA-capella: A Real-Time Audio FX Unit

Cosma Kufa, Justin Xiao

December 9, 2015

1 Introduction

In this report, we will present a summary of our project, FPGA-capella. Musicians often find that, during live performance, it is often desirable to apply audio effects to the incoming sound. Usually, this would require expensive and bulky outboard analog gear. However, with FPGA-capella, there becomes a simple, digital implementation that allows for flexibility for the user, as well as an appealing visual interface. The user can layer 3 effects at a time, choosing the order as well as any parameters that the effect would require.
In the paper below, we will give an overview of the project. We will then detail each module's description as well as how they were implemented. Finally, we will discuss the process of making each of the modules and bringing the project together.

2 Overview

FPGA-capella can be represented by three major modules: the FX module, the FX controller module, and the visualizer module, as shown in Figure 1 below. The FX module is responsible for taking in audio and applying effects like delay and distortion to the audio. The FX controller module allows the user to determine the ordering of the effects, as well as control parameters of the FX, while the visualizer module is responsible for displaying the FX controls and a visualization of the audio. The FX module takes in audio data from the AC97 and outputs audio data with effects back to the AC97. It also takes input from the FX controller module and outputs to the visualizer module. The FX controller module takes input in the former of user selected parameters, and outputs to the FX module and visualizer module. The visualizer takes input from the FX and FX controller modules. Within the FX module are many modules that correspond to each of the different FX, while the controller module has three submodules that correspond to each of the three effects that are applied, and how to apply them. the visualizer module is responsible of relaying the information to the screen. it is responsible for getting information for each pixel and determining what color it should be and relaying that information to the VGA output. Additionally within the visualizer, there are submodules for the different images used, which each have of them have their own specified ROMs holding information about the images. While we were unable to integrate the modules such that one could control the FX from the visual interface, we were able to visualize the audio as well as visually depict the FX controller. In the following section, we will detail each of the modules and their implementations.

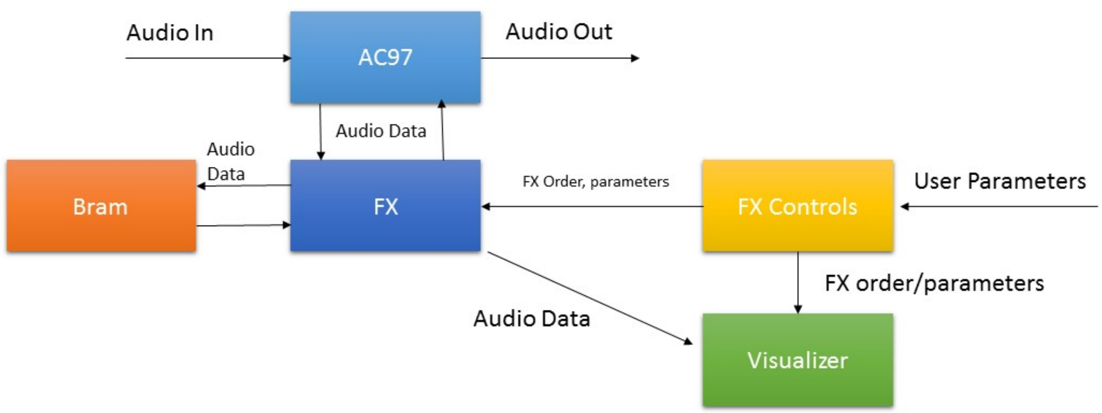

Figure 1: High-level Block Diagram

3 Modules

3.1 Audio

Justin Xiao

In this section I will detail the audio component of the project, including the audio framework of the project as well as the FX module and its submodules. For the project, we used the general audio lab framework provided in Lab 5. One of the first things to consider was whether to use 8-bit or 16-bit audio. To do this, it is a simple matter of changing a few variables from 8 to 16 bits in the lab framework. There was no noticeable difference, so we stayed with 8-bit audio to maximize memory capabilities. This lack of difference is expected; the biggest difference between 8 and 16-bit audio is dynamic range, which we decided was not as important as having enough memory to store the audio. Next, to allow for stereo audio, we configure the 8-bit audio to two 8-bit bytes stored in a 16 bit array, which is then interpreted by the AC97 as stereo audio, with each of the bytes corresponding to a left and right speaker.

For the FX modules, we had initially planned to implement delay, echo, filtering, distortion, panning, looping, pitch shifting, octavising, chorus, and reverberation. However, since pitch shifting, octavising and chorus involve fast Fourier Transforming (FFT) the audio, processing the audio in frequency space, and then inverse fast Fourier Transforming to get back to the time domain, it was determined that these effects would not be reasonable to implement, since the IFFT takes too much memory and is a difficult computation that has not been achieved in previous projects. Instead, we decided to implement delay, echo, reverb, panning, tremolo, distortion, filtering, and looping. Below, I will detail each of these effects and how they were implemented.

Delay

Delay is a commonly used effect to give a sense of space, whereby an audio signal is processed such that a time-delayed, often times attenuated, copy of the original signal is added to the original signal. A block diagram depicting delay can be found in Figure 2 below.

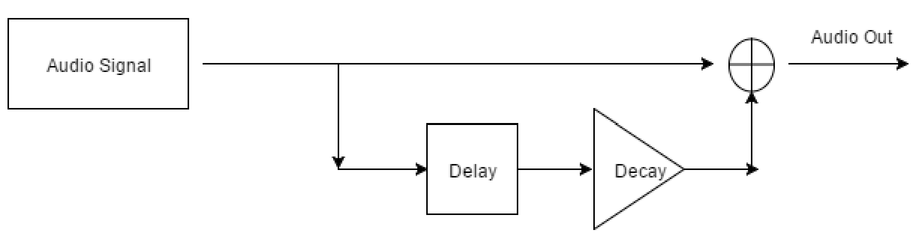

Figure 2: Delay Block Diagram

Implementing this was rather challenging for me, conceptually. My first idea was to simply write to BRAM memory and then read when necessary. In my naïve implementation, though, we find that this isn't possible since we can't read and write at different addresses of the same BRAM at the same time. I next thought to store every single sample in a temporary buffer state, but this soon proved to be a very unreasonable approach, as even one second of recording would contain millions of samples. To do the final implementation, I went back to the BRAM concept; first we go to the delayed sample in memory by subtracting the delay address from the address, and then buffering to prevent glitches as well as multiply by the decay. We then take that delayed sample and go back to the current address incremented by 1, set that to be the memory into BRAM, and add that to the output.

One potential problem is overflow. For example, if an incoming eight-bit audio signal has another 8-bit delayed signal added to it, the result may need nine bits to represent, resulting in a nonsensical output. To compensate, we must normalize the resulting audio signal such that the result remains eight bits. This can be achieved by setting a 9-bit reg to the sum of the two samples and then taking the lower 8 bits. This technique results in a normalized sum.

The delay module takes in two parameters from the user: delay time and decay, which are both 10 bit numbers. The delay time determines how much time elapses between the initial sound and the playback of the copied sound. A longer time gives a larger sense of space. Decay determines how much attenuation there is between the original sound and its copy. With more decay, the copied sound becomes softer relative to the original sound.

Echo

Similar to delay, echo adds time-shifted copies of the original signal, except with a feedback loop such that multiple, decaying delays are played constantly after the original signal, much like a real echo. A block diagram of echo can be found in Figure 3 below.

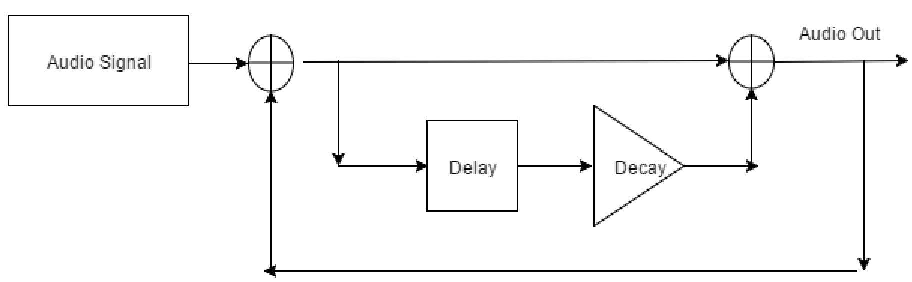

Figure 3: Echo Block Diagram

Implementing this was almost identical as to the case of delay, with the crucial difference being that instead of just sending the audio input to memory, we also send the retrieved memory to memory.

Like delay, echo takes in two parameters, delay time and decay, which are both 10-bit numbers, as in delay. Delay time now determines the time between each copy of the original sound, such that they are equally spaced from each other and the original. Similarly, decay now determines how much attenuation there is between each copy of the sound.

Filtering

Much like in Lab 5, filtering works by attenuating frequencies of some signal. In the lab we implemented a low pass filter to block out high frequency noise, but filtering is often applied for artistic reasons, such as using the low pass filter for a muffled, behind-closed-doors sound, or using a high-pass filter for a bright, airy and somewhat empty sound.

In our project, we implemented filtering very similarly to in lab 5. The user has the choice of three filters: a low-pass filter with cutoff frequency at 800 Hz, a bandpass filter with a center frequency at 1500 Hz, and a high-pass filter with cutoff at 1500 Hz. To make the filters, I used the fir1 function in MATLAB, with sampling frequency of 48000 Hz and using a 63 tap FIR filter. I then scaled the coefficients for the low-pass and bandpass filters by 2048, while scaling the high-pass filter by 1024, and then rounding to the nearest integer. Using the filter module developed in lab 5, we can then filter the sounds using any of these filters. The MATLAB filter frequency response is shown in Figure 4 below.

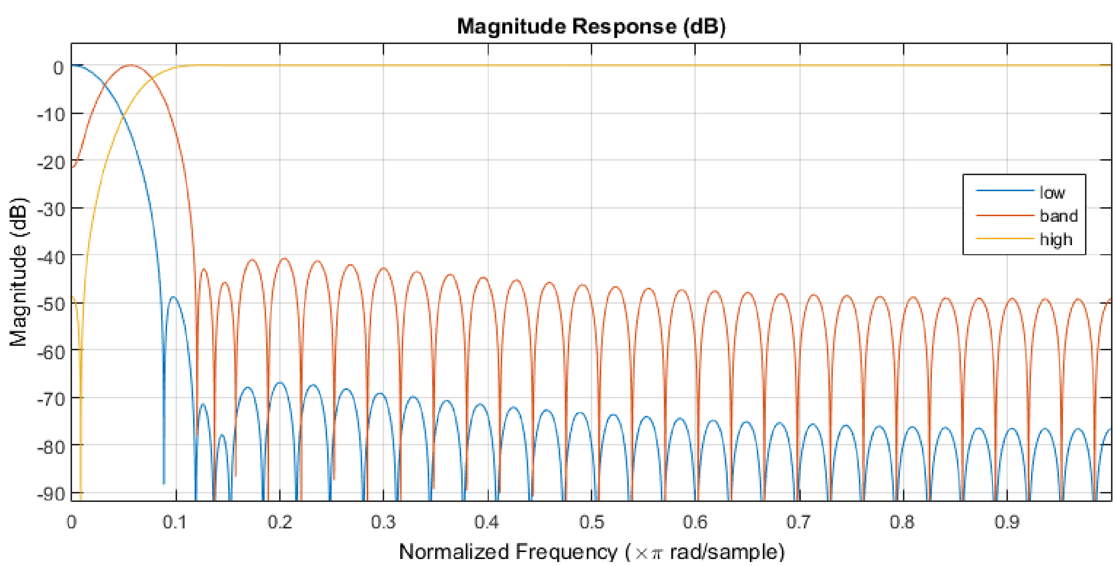

Figure 4: Filter Frequency Response

The filtering module takes as user input one parameter, which is a 2-bit number that determines which filter to use. The user then has creative control over how their signal should be filtered.

Distortion

Distortion is an effect commonly used in guitar music, such as the canonical crunchy electric guitar sound. While there are many methods of achieving this effect, we decided the most reliable method would be bitcrushing, where the incoming audio is reduced in bit depth, thus quantizing the audio signal more. This has the effect of producing a distorted sound. A waveform example of bitcrushing is shown in Figure 5 below.

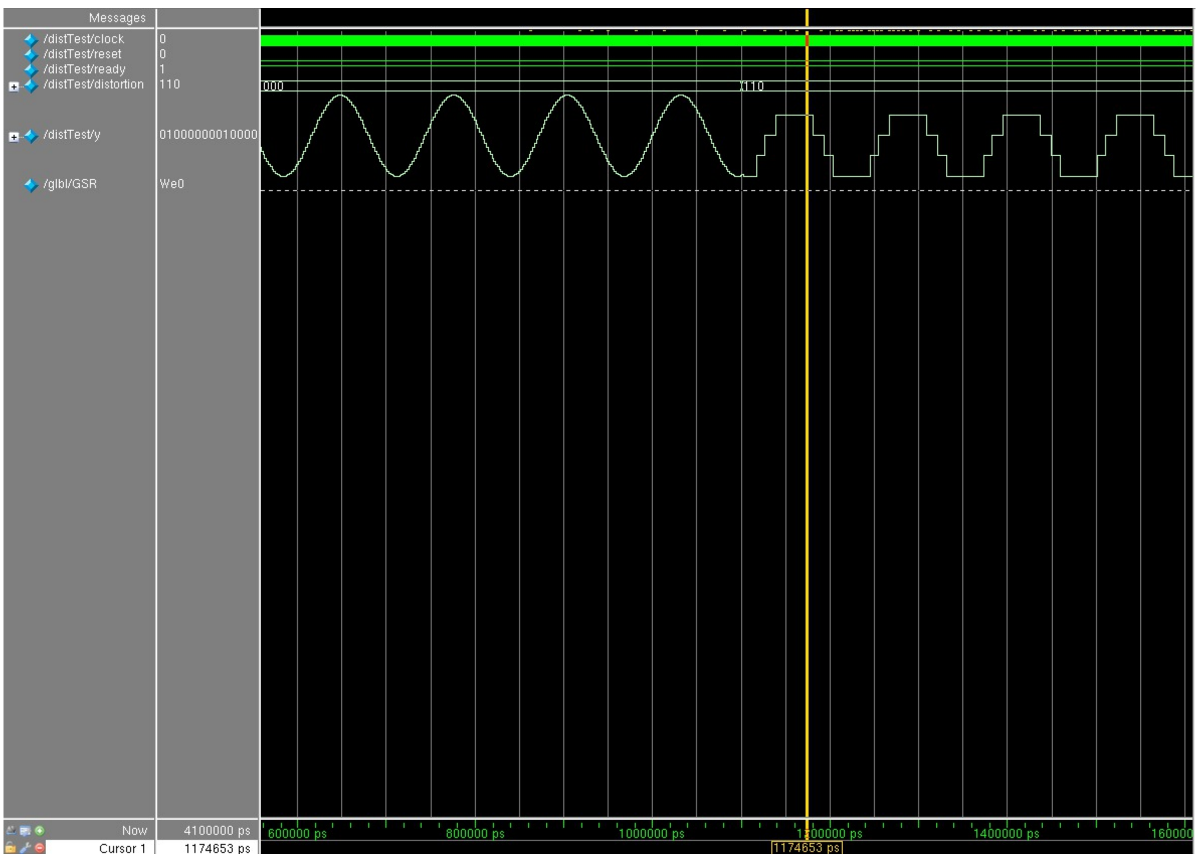

Figure 5: Comparison of Undistorted and Distorted Sine Wave

Implementing distortion was relatively simple, whereby the incoming 8-bit audio has its lower order bits set to zero, effectively reducing the bit depth of the signal and thus introducing the distortion.

The distortion module takes in one parameter from the user, a 7 bit number that determines how much distortion there is. The number essentially determines just how many low order bits to set to zero.

Panning

Panning of a stereo signal involves sending differing amounts of an audio signal to the left and right speakers. This is often useful for spatialization of the sound, allowing for the listener to perceive sounds coming from different directions. A comparison of output for left and right channels of a stereo sine wave is shown in Figure 6 below.

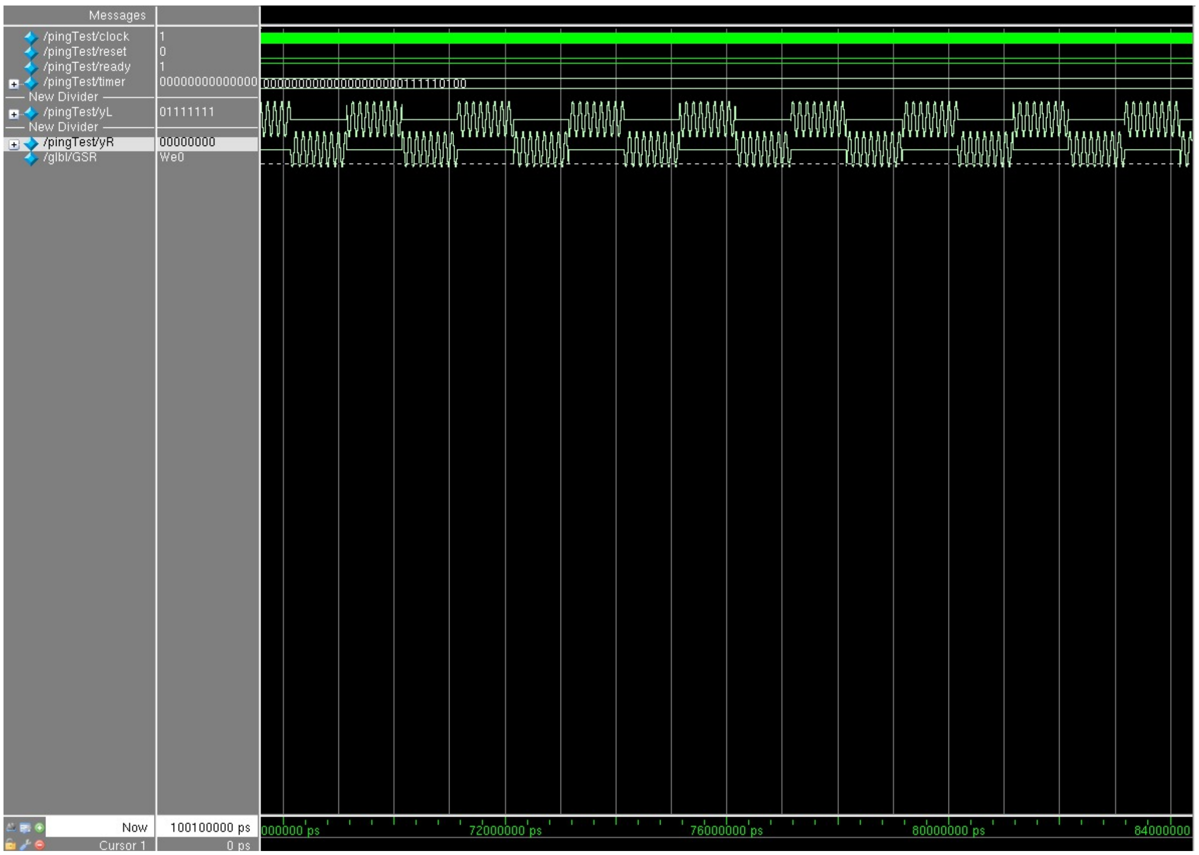

Figure 6: Comparison Left and Right Channels of Sine Wave in Ping Pong Pan

In our project, we use the concept of panning to create an interesting pan effect known as ping-pong panning. Ping-pong panning is when the audio signal bounces between the left and right speakers periodically. For example, if we set our period to be one second, then every second the sound switches from being played in the left ear to the right ear and so on. This forms an interesting spatialization, and makes for a cool audio effect.

To implement this, I send a 16 bit version of the 8 bit audio, 8 bits per channel, to the AC97 for stereo. Given a period, on every half period, one set of 8 bits is set to zero while the other 8 bits remain, turning off the corresponding channel. Thus, in our one-second period example, given the array [Left, Right], for every second, this becomes [0, Right] and [Left, 0] as indicated.

The panning module takes as input one parameter, which is a 29-bit number that corresponds to the period of the ping-pong pan. As the parameter increases in magnitude, so does the period.

Tremolo

Tremolo is another audio effect often found with electric guitars, where the input signal is rapidly varied in volume with some period. This creates an interesting shuddering effect, where the sound quickly goes from sounding to not sounding. Figure 7 shows a simulation of tremolo on a sine wave.

Implementation of tremolo was relatively simple, using the same concept as from panning. Given some period, on every half cycle, the audio output of the tremolo module is set either to the incoming audio or to zero, which corresponds to silence.

The tremolo module takes in one parameter as input, which, like the panning module, is a 29-bit number that corresponds to the tremolo period.

Looping

Looping is simply the recording and layering sounds; the performer records a sound and the recording will play on loop, similar to lab 5. However, once a new sound is recorded, that sound is played simultaneously with the old sound and looped. This allows for a layering of ideas and increased musical complexity for a solo performer.

I expanded on lab 5 in order to implement looping. In lab 5, we recorded a segment of audio, and then upon playback it would loop that segment on end. Once we recorded a new sample, however, the previous one is overwritten. In my implementation, however, instead of overwriting the previous sample, whatever is recorded is now added on top of the previous sample, as the sample is playing such that during recording the performer hears both what they put into the mic as well as what comes out from BRAM. Our implementation of looping also comes with a reset; if the user no longer wants whatever audio is stored in BRAM, pressing the reset button will loop through the BRAM addresses and set all of the memory at those addresses to zero.

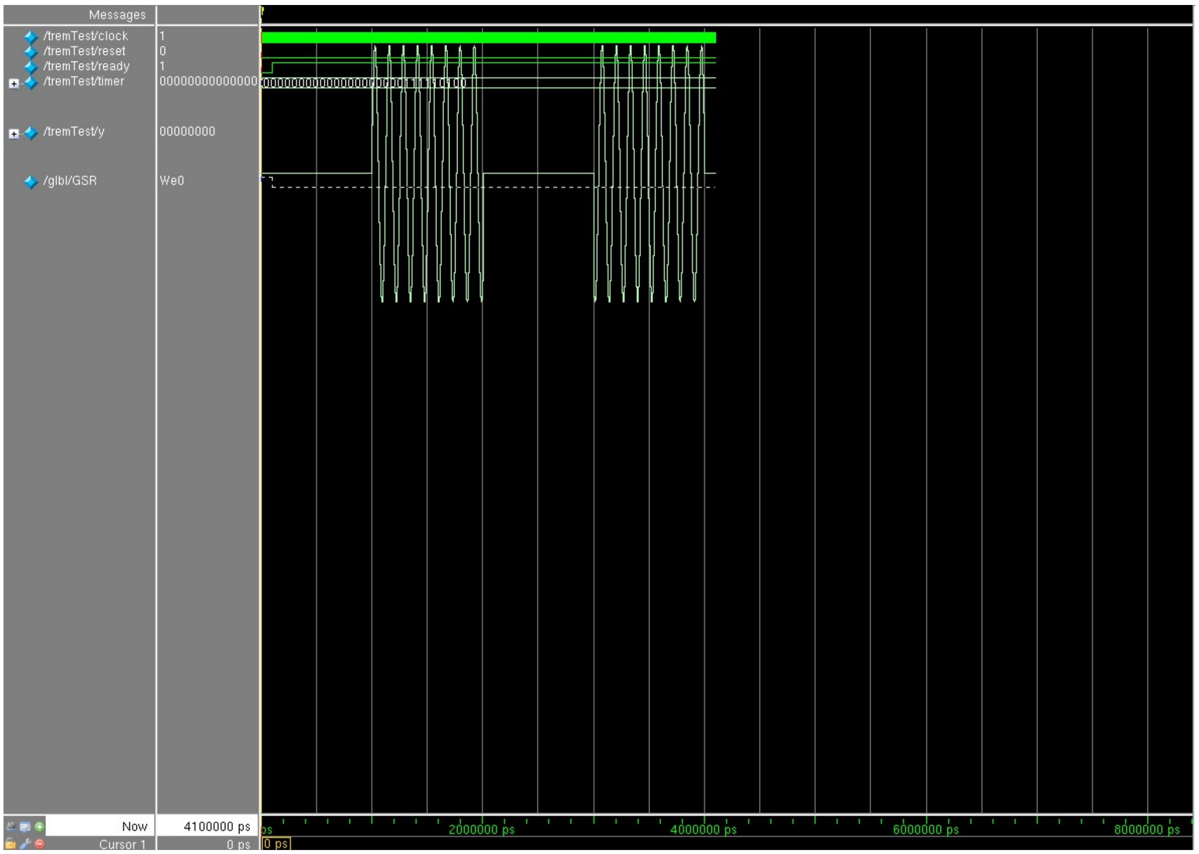

Figure 7: Tremolo on Sine Wave

Now, we run into the same issue here as we did in delay, except even worse. The issue again arises as to how we could continuously add sound samples without quickly running into overflow problems. In order to get around this, we apply the same concept of normalizing the audio, such that we halve both the looping audio as well as the incoming audio and add the two together in a fashion identical to as in decay.

Reverberation

One of our project's reach goals was to implement reverberation, which is rather computationally expensive in most algorithmic implementations. Reverb gives the effect of space, since in a physical room a sound reflects off of the walls multiple time, creating a persistence of sound after the sound is produced. At a basic level, this is implemented by adding multiple short delays of decaying strength; however, it is difficult to algorithmically implement.

To get around the fact that most reverbs are hard to implement, I decided that a simple yet effective workaround was that of adding together multiple echoes, so within the module I added together two echoes of different decays and delays that I optimized for the most reverb-y sound. Although it sounds more like echo than reverb for sounds with short attacks and decays, such as snaps and claps, for more smooth sounds such as singing, it sounds very much like a reverb. While more echoes would allow for more reverb-like sound, it became impossible to do because multiple calls to the echo module creates more instances of BRAM and thus takes too much BRAM that the labkit could not provide.

3.2 FX Controller

Justin Xiao

Here I will discuss how the audio FX controller module works. The first thing I wanted to implement was effect control via the labkit switches. I configured switches 1 and 2 to act as an FX order selector; if they are 00 then we are changing FX1, and so on. Switches 3-5 determine which effect to use; for example, if the switches are set to 001 then distortion is being used. Reprogramming the effects is done with Button 0; once we have configured the switches to change the effect order to the effect we want, we press the reprogram button and the effect is updated. This is done in a high-order FX parameter module, which take as input a reprogram bit, the 2-bit selector, and the 3-bit switch value, and then outputs the three 3-bit numbers corresponding to each effect for effect one, effect two, and effect three.

Since we decided to layer up to three effects on top of each other, I wrote three different modules to correspond to each of the three effects to use. Within these modules, I felt that the easiest way to go about it was to instantiate all of the FX submodules within the controller module, since it is relatively difficult to conditionally instantiate each module, and have them correspond to a 3-bit number. As input, each controller module took in all user parameters for every effect, allowing it to instantiate each module in the way the user wanted. The controller module also took as input the three bit number that told the controller which of the effects output to use.

One problem I ran into was that, by instantiating every effect three times, I quickly run out of BRAM space, not allowing for the labkit to be programmed. To get around this, I instantiate each of the BRAM-taking modules once in one of the controller modules; in the first controller module, I allow the user to choose whether or not to loop, without allowing for reverb or echo. In the second module I allow the user to choose one of reverb or echo, but not looping. I had to get rid of delay as an effect, since it took too much BRAM and was not as useful as echo or reverb. Another problem I ran into was that, most of the effects were built so that they only had mono output, so panning in controller one or controller two would give an effect more like tremolo, with a mono output.

Thus, the user can only apply the pingpong panning effect in controller three. A finalized list of the effects and the 3-bit number they corresponded to can be found in the table below.

Table 1: Effects in Each Controller

3-bit Input	FX1	FX2	FX3
000	None	None	None
001	Distortion	Distortion	Distortion
010	None	None	PingPong
011	Tremolo	Tremolo	Tremolo
101	None	Echo	None
110	Filter	Filter	Filter
111	Looping	Reverb	None

These controller modules, along with the FX parameter module, were instantiated in the top-level document such that the switch info was fed into the FX parameter module, which tells each of the controller modules which effect to use. The audio into the AC97 chip is fed into controller 1, which outputs into controller 2, which then outputs into controller 3. Finally, controller 3 outputs its audio back to the AC97, which plays the audio.

3.3 Visuals

Cosma Kufa

In this section I will detail the visual component of the project, including the visualizer module, the backbone of project, the individual submodules, and the logic that went into bringing out some of the effects.

Visualizer

This is the main module responsible for the framework of the display. within this modules instances for the different images, which are the main title, frequency title, analog title, analog meter, the table of effects, the table specifying the effect, and the three selector blobs specifying the effect, are instantiated, with their parameters specifying, which location they will be located. Additionally within the module there is logic for determining the location for the three effect selectors. The module takes in information from the FX controller which specifies, what the selected effects are for each effect slot. Using a different case statement for each selected slot we update the location of the selector and visibility of the selector dependent on whether an effect has been chosen and if so, what pixel locations should it be placed to convey the information. Additionally another main input for this module is it takes in the magnitude of the audio, which will discuss later in the analog module.

Analog

This module is designed to reveal the different colors of the analog meter. Within the module, it has BRAMs specifying pixels and a color table to specify what the color should be. Additionally, unlike the blob module and other image module, this module takes in an input called reveal, which is just the magnitude of the height. Within this module we proportionate the magnitude of the audio to the height of the analog meter. Thus, if the audio is the loudest, the full image of the analog meter is revealed, if the magnitude of the audio is 0, the image is not shown, and if the magnitude is in between a proportional piece of the analog meter is revealed.

Image modules : Frequency Title, Analog Title, Effects Row, Effects Slot Column

These modules are all similar because they all do the same thing, which is display the image specified by the module name. However, they each have their own encoding for each pixel and encoding for their color table. Additionally, because I designed for the text to be green and have the background be black. When creating the BRAMs for these title images, I saved memory as well as I did not need to BRAMs for the red and blue component as I was able to just hard code them to be 0 at all times.

Process Audio

This module was responsible for doing the calculations for the Fast Fourier Transform of the data and creating the histogram plot that was displayed on the screen. One of the benefits of this module was that it had a scaling factor, which would allow you manipulate the data and proportionate it, which was very useful in this project as we had limited the FFT to a specific space. however, even with scaling this was not enough and the FFT would have pixel representation in other places on the screen.Thus, in addition to the scaling factor, I placed a cut off height for the magnitude of the frequencies, to the height of the designated pixel space for the spectrum.

4 Visual Design Implementation Process

Cosma Kufa

For the project, I used the general visual platform from lab3 as a way to display the sprites and images that we wanted to display. Additionally I made use of the open source resources for the Fast Fourier Transform, to display the FFT of our audio and the MATLAB script for converting bmp file images into COE files that could be used by Verilog in order to display the images.

The first task at hand was to consider what our constraints were and how we wanted our display to look. However, although we wanted the display for the visualizer to be visually pleasing, at the same time we were limited by our memory resources. When converting an image file into a COE file, there are four files needed to be created, the COE file with an encoding for each pixel color, a COE file for the decoding of the red component, the green component, and the blue component. Thus, every image would require four BRAM sources to hold the information necessary to display the image. The memory space for COE file, specifying the encoding for each pixel, was almost static, so it's size could not be significantly changed. However depending on the variety of colors in your image the COE files for the color components could be reduced. To put this into perspective there are over 16 million colors that a display can show and each color is represented by 24 bits, so over 100 MegaBytes of memory would be required and our labkit would not be able handle such a large volume, so a limited range of colors was required, so for our project we limited our color choices to 256, which still was a large enough spectrum that the quality of the image was not destroyed. Additionally, when it came to text images, we chose to limit text to a single color, and chose the color green, as that was the color that would be used to display the frequency spectrum.

Once I knew how to create images and the COE files The next task at hand was implementing the visual design. When it came to implementing the visual design, I ended up having to go back to the drawing board multiple times. The First design is depicted in figure 8. In the first design, the design was more favorable for a screen that was taller and thinner. However, when it came to implementation time, I came to the realization that the screen would be wider and shorter. Due to this different dimensions when implementing the design the visuals for the spectrum and analog meter would look squished. Thus, I went back to the drawing board to design another layout. The second layout was better suited for the screen as it took into account the dimensions of the screen, and utilized the wider space, as can be seen in Figure 9.

After deciding on our final design, I decided to implement it. when implementing, there were quite a bit of my expected implementations that were flawed and did not work. for instance I had hoped to make a grid-ed image for the effects selection visualizer. however, when producing the image COE file. I realized this would be a problem as the memory for the COE files were too big even with the limited color range. Thus, I had to redesign that image. Thus i decided to split it into two images. one column image with the definition for each row and one row image with definition for each column. This redesign helped reduce the memory. Additionally I came to the conclusion the grid lines weren't necessary and the selector blobs were the right size for a user to distinguish which row and column it was located in.

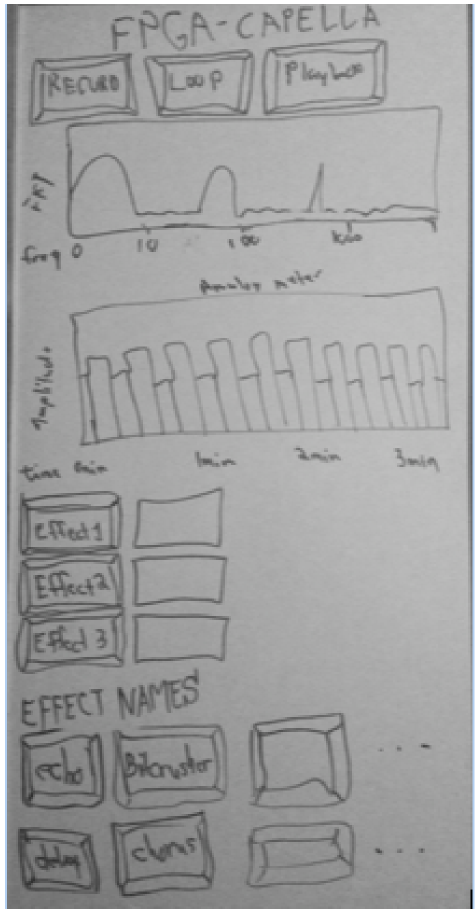

Figure 8: Visual Design 1

Also another changed implementation was the analog meter. Originally the idea was to have a shifting analog meter which would have the most recent amplitude shown on the left most side and then keep shifting over to the right of the time. However, again I came to the conclusion that this would be not plausible as it would require a lot of memory to keep track of the amplitude at each height. Thus, I created an analog meter, which would display the magnitude of the audio.

Last, but not the least. I was faced with the challenge of incorporating the FFT code with the code for the interface I had created. The Challenge was that I needed to figure out the necessary files from one project file and transfer them over. Because the two modules were in two different project files I had check how each project file interacted with the labkit. It was easy for some

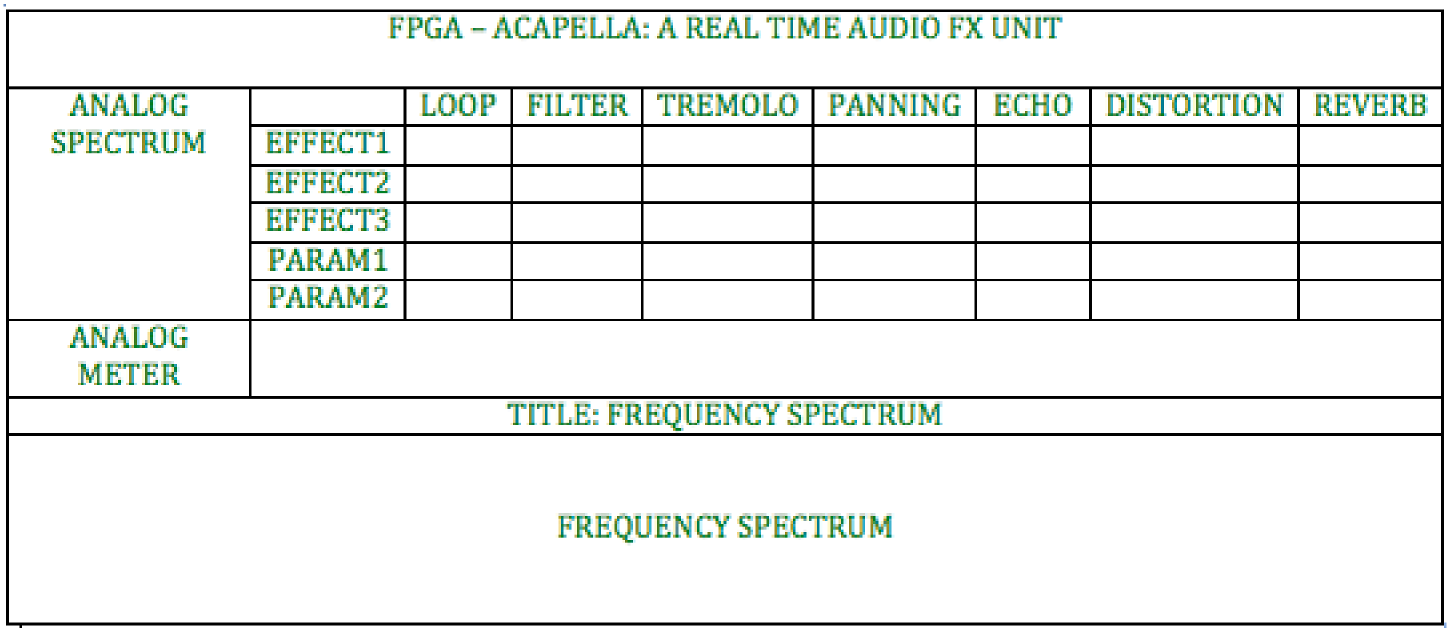

Figure 9: Visual Design 2

components because some of the components were mutually exclusive, however the one that proved tricky was with VGA display. at first it was unclear how i would combine the two pixel data, but I finally came to the conclusion to treat them like the blob and image modules. because I had partitioned the space within the layout I knew that if i used an or operation on the two pixel data's to form a single the FFT would show up in it's designated face and would not interfere with the rest of the layout.

However, when it came to implementation this was not true and the FFT spectrum would affect the pixels of the interface, which is why the cutoff for the FFT data was implemented in the Process Audio module.

5 Integration

To integrate the audio and visual components of the project, we used two different labkits for both audio and visuals respectively, since both took up too much BRAM on their own. In order to feed the audio from the FX labkit to the visuals labkit, we simply connected an audio aux cable from the speaker output of the audio labkit to the microphone input of the visuals labkit, which has speakers/headphone output.

However, this does not allow for the visuals module to see which FX are being used and display them. To do this, the user1 pin outs on the audio labkit

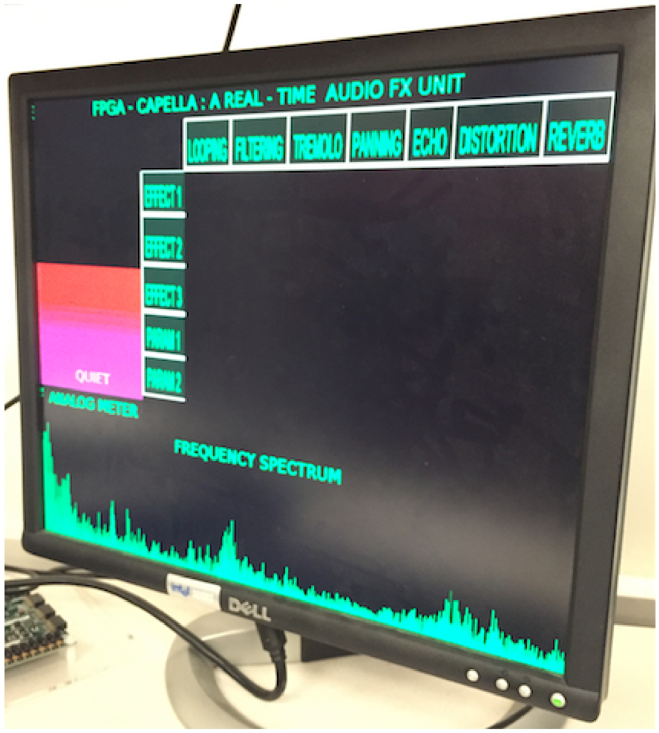

Figure 10: Final Visual Design

are configured to correspond to each of the 3-bit numbers corresponding to what effect is being used, such that user1[2:0] corresponds to FX1, user1[5:3] to FX2, and user1[8:6] to FX3. These are then fed to the user1 pin in on the visuals labkit, allowing for the visuals labkit to process and display which effects are being used.

6 Expansion

One of our goals was to make the visual interface user interactive, such that, using a mouse, the musician could drag and drop effects and click on different parameters. However, we were not able to get the mouse working in time for this to happen.

One of the expansions to our project would be the ability to record sounds and apply effects to the recorded sound. While not terribly difficult, we decided not to implement this; rather we focused our time on live sound, since that is the audience this product caters to. However, in the future, we would like to implement recording ability, such that audio is stored into ZBT memory (for longer audio samples), and then applying effects to those recordings being played back, rather than to live input. We could than rewind, fast-forward, slow down, and apply other time-based effects to the sound. Visually, we could represent the time location with a playback time bar, which fills up as more time passes.

7 Conclusion

We have, in conclusion, developed an audio effects unit that allows musicians to customize their sound with interesting audio effects, as well as visualize their sounds in an interesting fashion. Overall, the project was a success. We were able to apply every effect we had set out to implement (sans pitch-based effects, which would be nigh impossible), along with an interesting visual interface that displays information about the audio as well as a table of which effects are being used. Creating this project was both educational and interesting; going through the design process and the building process, as well as presenting on the project along the way, gave an opportunity to taste the real world of engineering. In future work, we would like to expand the project to include audio recording capabilities, as well as direct interaction between the user and visual interface.