Selective Listening

By

Bryce Tharp

John Hammond

Wei Yang Ang

Final Report for ECE 445, Senior Design, Spring 2021

TA: Evan Widloski

4^{th} May 2021

Project No. 15

Abstract

This ECE 445 project realizes a flexible, real-time audio filtering and re-mixing platform for general-purpose hearing aid research. The processing unit used was the Altera DE10-Nano FPGA board that enumerates efficient FIR filters. A custom audio codec was designed as a daughter-card for the FPGA and includes a programmable ADC, analog-controlled DAC, custom headers for mounting onto the FPGA and for processing 16-channel stereo audio. Multiple microphone array PCBs were produced to create a complete array of fourteen digital MEMS microphones. These microphones were integral to collecting enough audio information to generate unique filter coefficients. A slider board was designed to give the user control over which FIR filter to use, and demonstrated how they can reload coefficients in real-time.

Our demonstration showed how users can switch between different FIR filters almost instantly, and showcased a 16 count microphone array comprised of digital and analog microphones. FIR coefficients were generated in software via the ARM core of the DE10-Nano and loaded into the FIR filter banks located in the FPGA fabric.

Contents

1 Introduction

1.1 Problem and Solution Overview

Current hearing aid research is often hampered by the inability to quickly develop hardware test platforms. These platforms must be fast enough to process audio signals in real-time, and efficient enough to perform expensive mathematical operations. Additionally, modern day hearing aid technology is often proprietary deeming it inaccessible to researchers worldwide. Our ECE 445 project solves these problems by providing a flexible microphone array processing platform for real-time filtering and audio remixing. By using our device, researchers can run their own audio processing algorithms, generate custom FIR coefficients and reload them into the Augmented Listening platform in real-time.

The platform we have constructed is an extension of the work done by the Augmented Listening Laboratory under Ryan Corey [4]. Our project will be published as an open-source release for researchers to leverage in order to expedite the hardware design for realizable signal processing devices.

What follows is a discussion of the primary objectives, detailed design procedure, verification and results of the Augmented Listening Platform.

1.2 Background

Hearing aid devices have long been shrouded in proprietary technology. The Augmented Listening Laboratory is creating an open source listening platform to improve hearing aid technology. Standard hearing aid devices typically contain only two microphones close to the ears. Hearing aid performance can be improved by adding multiple microphones surrounding the headpiece in the relevant space near the target source. Our demonstration will show off the power of adding many microphones by using a directional listening technique called "beamforming".

Beamforming is a signal processing technique used to extract specific sounds from sources in a room. In our demonstration we attempted to show in practice how to calibrate and beamform towards multiple sources with user input. Although that goal was not fully accomplished due to a few hardware bugs, this operation is completely possible and effective within the capabilities of the Augmented Listening Platform.

1.3 High-Level Requirements

- The listener can distinctly hear each unique source one at a time in our demo using a mixing board, and independently adjust the level of several real-life sound sources.
- The total delay through our system should be no more than 10ms to avoid disorientation.
- The binaural headphones must preserve spatial awareness and directionality of all sources.

1.4 Visual Aid

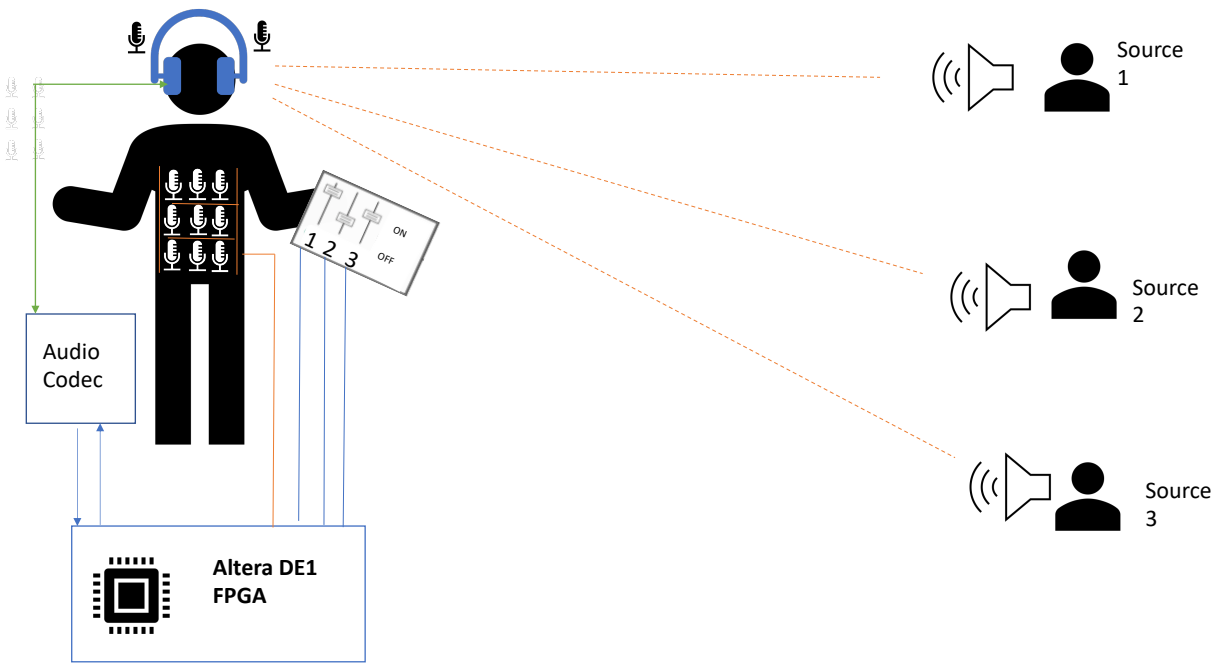

Figure 1: User diagram

2 Design

The Augmented Listening hardware platform is divided into four main subsystems. The FPGA audio processing block, the custom audio codec and the "mixing" volume control board are a complete unit. The microphone array and binaural headphones nested within the "Listener" design block are affixed to the user, or in our demonstration we used a mannequin. Figure 2 shows the block diagram of all hardware components in the system.

For detailed images of the schematics and PCBs involved in this design, turn your attention towards the Appendix at the tail end of this paper. The schematics and layouts are located in Appendices C, D and E.

What follows is a detailed breakdown of each subsystem.

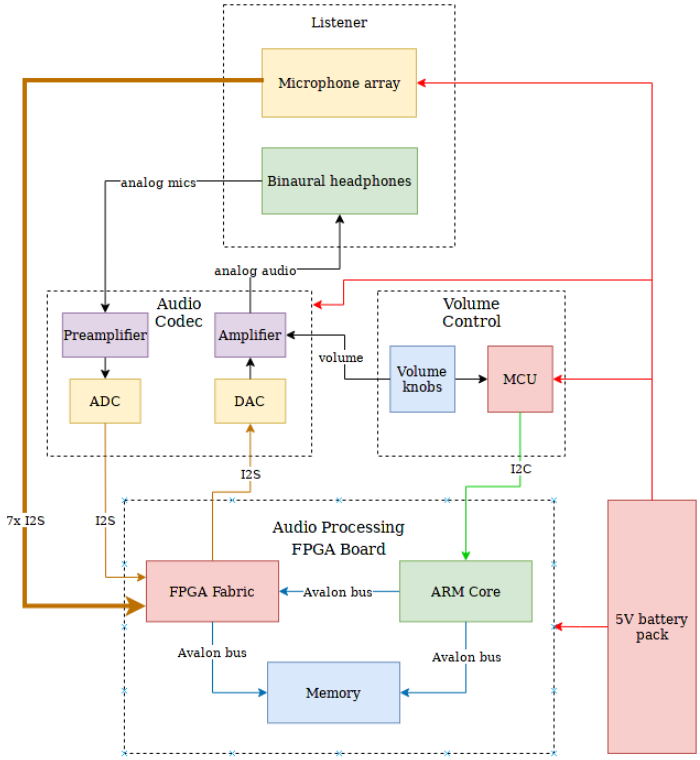

Figure 2: Block diagram of full listening platform

2.1 Listener

The Listener design block represents the wearable audio equipment that the user adorns while interacting with the platform. The two components are the binaural headphones and the digital microphone array. Binaural headphones are used in order to preserve spatial awareness - these headphones give the user audio perspective relative to other audio sources. They are a substitution for measuring the "Head-Related Transfer Function" which would be necessary in their absence. A microphone array is worn by the user to capture the audio in the system and provide clean audio signals to the software algorithm computing the unique filter coefficients for a beamforming application.

2.1.1 Binarual Headphones

The binaural headphones are worn by the listener in order to preserve spatial cues. In this way, the listener is able to naturally recognize noise regardless of their orientation in the sound field. The headphones used contain two analog microphones that are then converted digitally to be processed with the microphone array digital microphones. By positioning these analog microphones close to the ears, the listener can negotiate the location of audio sources in the room without additional measurements.

2.1.2 Microphone Array

The main microphone array consists of a static grid of 14 small MEMS [1] microphones. These MEMS microphones have a built in ADC so they connect directly to the FPGA over I^2S. We chose 14 microphones

here to keep a total array size of 16, including the binaural analog microphones.

Each of our MEMS microphones comes on a small breakout board. We chose to use these boards instead of directly soldering to the MEMS microphones because they are very fragile, and a challenging package to work with. However, these breakout boards themselves are not sufficient to create a full array. We constructed a larger board to affix each breakout board to, and daisy chained these larger boards together to form the complete array. A 16-pin ribbon cable connected each of these larger PCBs together. The schematic and PCB of this subsystem can be referenced in Appendix C.

Signal integrity and clock skew were our final considerations when defining array specifications. Each microphone must be on the same sample clock and bit clock to ensure samples are lined up correctly in time. By virtue of our daisy chain design, each microphone shared the same conductor for both of these clock signals, which limited our skew to that caused by the wires themselves. Our longest cable ran from the end of the microphone array daisy chain back to the FPGA, which was no more than 14 feet with the cables we chose. We did not experience any signal integrity issues, and the microphone array as a whole functioned as expected.

2.2 Audio Codec Board

The Audio Codec board was designed because the DE10-Nano (the FPGA board we used) did not have an on-board codec. The Audio Codec board interfaces directly with the DE10-Nano's GPIO pins - a 40-pin female header connects the two together. A programmable ADC, input and output amplifiers as well as an analog DAC were used to have complete control over the audio format and sound.

The complete schematic and PCB layout for the Audio Codec board can be found at Appendix D.

The following sections describe how the input audio is handled, and then describe the output sequence back to the listener.

2.2.1 Input Stage

The input stage is comprised of three main components: the input microphone jack, input amplifier and the programmable ADC. The input microphone jack accepted the 3.5mm analog microphone jack and produced the the left and right analog signals. The input amplifier was a dual op-amp that could boost both signals to line level. Unfortunately, the input amplifier burned out after it was soldered. Luckily, we had a programmable ADC on hand.

The ADC IC we used from Texas Instruments was a PCM1863DBT [7]. A standard industry ADC with programmable registers to configure gain, boosted input amplifiers and I^2C control. Via the FPGA hard processor our team was able to program the ADC registers to boost the input signal to line level, and configure the audio format to an I^2S output.

We tested the sound coming from the ADC by using the the analog microphones. Once the output sound was clear and showed an I^2S data line and clock signals on the oscilloscope, we were confident in our ADC implementation and in our decision to bypass the input pre-amplifier.

2.2.2 Output Stage

The output stage of the Audio Codec board is comprised of the DAC and the output amplifier. The DAC chosen for the codec was a PCM5100APW series IC [8] from Texas Instruments. It was completely analog controlled which lent itself to a simple PCB layout. However, we ran into issues with the chip and found it difficult to debug due to an inability to garner any information from information registers. Thus, we pivoted to using a second daughter board on-hand to complete the codec.

The PIC32 audio codec board [11] we had was a board that broke out the AK4642EN codec from MicroChip. We used purely just the DAC portion of this chip and configured programmable parameters such as output gain and volume level via I^2C. Once programmed, the custom codec we designed, and the connected codec IC worked nicely together. The analog microphones from the binaural headphones sounded clear coming out the DAC through the "Line-Out" jack.

Even though many modifications to the custom codec PCB were made, the end result was successful. The board was the bridge between the FPGA and the rest of the system and fed the augmented audio back to the user of the platform.

2.3 Volume Control

The Volume Control subsystem was designed as a PCB with potentiometer sliders to send an analog signal to the FPGA to augment the volume of several real-time sources. An AVR128 microcontroller IC [10] was responsible for sending the sampled voltage level via I^2C to the FPGA. However, there were hiccups in the integration of this block and many of the features designed in theory never made it to the final demo. Nonetheless, the team provided a simple user-interface to showcase the real-time filtering operation with the current architecture in place.

2.3.1 Theoretical Design

This module was designed as the third PCB in the Augmented Listening hardware platform. It contained four potentiometer sliders - the first acting as a variable resistor to control the master volume. This was done by placing its resistance across the output amplifier on the Audio Codec board. The other three slide potentiometers were to be used to send independent volume adjustments across the three sources relative to the level of the master volume. The PCB included the AVR128 to sample the three source slider values and send their relative position to the FPGA. An on-board 16 pin header (identical to the headers on the Audio Codec board and Microphone Array board) was connected to send the relevant signals back to the FPGA.

To see the schematic and associated PCB reference Appendix E.

2.3.2 Realized Implementation

However, the full implementation was not included in the final demonstration. There were problems using the AVR128's I^2C interface with the HPS on the FPGA. The FPGA's hard processor could not recognize the supposedly active I^2C bus on the AVR128. We reason that the overall pull-up resistance on the bus was not high enough to pull up the necessary signals for I^2C communication, or the code written for the AVR was faulty.

Nonetheless, user input was a necessary feature and the Ardunio UNO [2] supplied an I^2C bus with internal

pull-up resistors. The UNO controlled one of the potentiometer sliders, and the slider was sampled by the FPGA's ADC. This digital signal was sent to a software script on the HPS which selected a set of FIR coefficients to load into the filters. Thus, user input was factored into the demonstration to showcase real-time filter reloading.

2.4 FPGA Audio Processing Platform

The main brains of our array processing platform is an Altera DE10-Nano FPGA development board. This board features a Cyclone V SoC, which has both an FPGA and ARM hard processor built in. We chose to use this platform as it has the flexibility to support highly parallelized, low latency filtering, as well as embedded linux to control our many hardware features from a shell.

2.4.1 DC Offset Removal

When dealing with fixed point numbers in our filter architecture, the samples are 16-bit signed integer numbers. These numbers have a relatively limited dynamic range, so we need to ensure that we make the best use of it. Unfortunately, the samples directly from our digital MEMS microphones have a nasty DC offset. The gain in hardware was severely limited, and the 16-bit numbers were easy to clip. To combat these issues, we wrote a small module to remove the DC offset from each microphone. Every MEMS mic we tested had a slightly different offset, so we decided to make our DC offset logic dynamic. There is a two second calibration period, set by software, during which this module records each sample coming in and averages them together. The resulting average value is very close to the DC offset of the mic. We can then simply subtract this number from the samples coming in, and the subsequent output sample will be centered around zero.

2.4.2 Downsampling

Our platform natively runs at a 48kHz sampling rate, which is great as it gives a very high fidelity recording of the mic audio. However, running as such a fast sample rate limits the maximum length of our FIR filters. To achieve better performance with longer filters, we needed our system to run at a lower sampling rate of 16kHz. Our clocks are generated directly from the FPGAs hardware PLLs, and it is easy enough to change the sample rate by just changing a number in software. However, our MEMS mics do not have the ability to run down to 16kHz; the minimum sample rate is 32kHz.

We decided to implement a downsampling module to get around this limitation of the MEMS mics. Downsampling from 48kHz to 16kHz is simple as its an integer factor, and very little data is lost in the speech frequency range. The first part of our downsampling module is a low pass filter. This filter is built very similarly to our large FIR filters in the filter bank, but it is only 127 taps long. This filter attenuates all information over 8kHz to prevent aliasing when it is resampled to 16kHz. After this filtering operation is complete, every third sample is output from the module, creating a clean 16kHz audio stream.

2.4.3 Floating Point vs Fixed Point

Number systems are a very important consideration when working with embedded signal processing applications. The larger a number representation, the more memory it takes, and the more complex and arithmetic operations are. From a user perspective, floating point numbers are far easier to deal with. When designing fixed point filters, the user needs to be careful not to overflow or underflow the filter, and must scale all

coefficients carefully. We decided that the extra design complexity of using floating point numbers would be worth it in the long run for the end users of this platform.

2.4.4 Floating/Fixed Number Conversions

The first challenge of implementing floating point filtering was converting the 16-bit samples from the mics to their equivalent IEEE-754 representation. We could've skipped this step and created multiply/add units that used both fixed and floating numbers, but it was simpler to just do all arithmetic in the filters using floating point numbers. Thankfully, we were able to take a bit of a shortcut when creating these modules. The fixed point numbers we deal with in this system are formatted as `fx16.0`, meaning that there are 16 magnitude bits and zero fractional bits. This is to say that our fixed point numbers are all guaranteed to be a whole number between $(-32768, 32767)$. As such, we could design our fixed/floating converters to simply truncate the decimal off the floating point numbers. This is not the most complete or correct implementation, but it greatly reduces the conversion complexity and works well for our application.

2.4.5 Area and Time Constraints

Our filtering architecture is subject to strict time and area constraints. First, each filter needs to perform its full convolution within one period of the sample clock. In other words, the filter needs to compute its output for the current input sample before a new sample arrives. We were not interested in doing any sort of pipelining on these filters, so this is a hard restriction. Next, the FPGA only has 112 multipliers and 5.6 MBits of block ram [9]. Typically FIR filtering is done on an FPGA with a string of multiply/add blocks, but we would not be able to build a filter in this fashion longer than 112 taps. This architecture becomes even less appealing when we considered that we wanted the ability to filter 16 channels simultaneously. To get around this constraint, we chose to utilize a FIR filter architecture that uses a single multiply/add unit to perform the entire convolution.

2.4.6 Modified ZIPCPU Slowfilters

We were able to find an open source low are FIR filter online, created as part of the ZIPCPU project [5]. The ZIPCPU is a soft processor made by Dan Gisselquist. His filter utilizes a single multiplier and adder, and has some support for dynamically reloading filter coefficients. The slowfilter can handle `fxX.0` numbers of various widths. This made a great starting point for our FIR bank design.

The first modification was naturally to implement floating point arithmetic instead of the existing fixed point operations. Unfortunately, it is challenging to find existing modules that are both single cycle and readable. We were able to find reasonable multiply/add modules, both written by Sheetal Swaroop Burada [3]. They are both purely combinational, and function pretty well. Though we will cover an issue we ran into with the floating point multiplier later, and speculate on some issues that we couldn't quite track.

The second modification we needed to add was some supporting logic to actually reload the coefficients. Interfacing with the Avalon bus directly from the HPS seemed inconvenient, so we chose to have a module that would load taps into a small memory, then a state machine that would send those taps one at a time to the filter's tap input. Writing taps directly from the HPS is not exactly reliable because the Linux kernel scheduler does not guarantee that it will execute every instruction in real time.

2.4.7 Floating Point Filter Bug

We encountered a couple of bugs throughout our work with these modified floating filters, some of which we could resolve and some of which we did not have time to fix. First, we encountered some impulsive noises in the output when the sample stream would cross the zero point. This occurred with any filters we tested, so we could confidently say that the cause was in our floating point math. The floating point multiplier we used would not properly detect when a product should be zero, instead outputting a very large product. This caused the impulsive noise we were seeing. We resolved this by checking for zeros in the input multiplicands rather than checking if the output should be zero after the numbers had been normalized already.

The other bug we encountered was similar in nature but much harder to trace. When we tested our filtering architecture with the filter sets to do beamforming, we observed an arbitraty impulsive noise at the output. Figure 3 shows this issue in a recording we took. This issue is likely due to another bug in the floating point unit, as the nature of the noise is similar to that we saw from the zero crossings before. However, it is much harder to trace because it does not appear at a specific value, and therefore we can't just pick out a point in time to look at on SignalTap. We set up a thorough software testbench to test this multiplier with every possible input sample, and every product matched with the C++ floating point math. Therefore, we determined this is likely an issue with timing constraints through our multiplier on the board. Pipelining the floating point multiplier would likely solve this issue, and that is an important next step for this platform.

2.4.8 Avalon Interface

With all of our hardware setup, we need the ability to interface with many registers directly from the Linux system. We chose to use the Intel Avalon bus to do most of this communication. Thankfully, setting up an avalon slave interface is relatively sample from the hardware side. The Cyclone V HPS has an AXI master hardware peripheral, which is memory mapped to the CPU. We can then use an Intel IP Core to interface between AXI and Avalon. We will not go into the details about virtual memory and translating between virtual and physical addresses in Linux. At a high level, reads and writes can be performed over the Avalon bus simply by dereferencing the virtual memory address.

2.4.9 Hardware I^2C Peripheral

Along with the Avalon bus, we utilize the I^2C bus to communicate with some peripherals off the board. The Cyclone V HPS features a hardware peripheral for I^2C, much like the AXI bus. However, the I^2C peripheral is simpler for us to use. We built the embedded Linux kernel with support for the peripheral, which allows us to use the `i2cset` and `i2cget` programs to read and write bytes over the I^2C bus. In this case the kernel driver interfaces with the I^2C peripheral, which saved us time over writing our own driver.

2.4.10 Coefficient Reloading Code

There are two major components to our coefficient reloading program. First, the program needs to read the floating point taps from a .csv file to be loaded into the filters. We did this by making use of a small C++ library that parses CSV files. Once the coefficients are loaded into float vectors, we can then send those coefficients over the Avalon bus to the FIR system's coefficient loading logic. This is done relatively easily, with the one caveat being sending those floating point numbers as their IEEE-754 floating point representations. In C++, when we write a float to the Avalon interface, it isn't necessarily send in its binary representation. We needed to dereference the floating point memory locations as integers to send over the

Avalon bus.

2.4.11 FPGA Conclusion

These modules and software come together to create a very flexible and robust platform capable of coping with the imperfections of real world use. The end user has the ability to use microphones that aren't perfectly centered about zero, use multiple sample rates, use very long filters, reconfigure the number of input channels, and much more. Though we have made significant progress on this platform, there are still a few small bugs to iron out before we feel comfortable with an open source release.

3 Verification

3.1 FPGA

The FPGA is the most challenging component of our design to verify due to the black box nature of FPGA development. It is difficult to observe the performance of the platform because we are limited to using SignalTap to view some internal signals or listening to the output audio. Despite these challenges we were still able to meet most verification items for our FPGA subsystem.

Our first verification item was to confirm the source to listener latency through our system was less than 10ms. 10ms is a common figure cited as the minimum perceptible latency, though it varies from person to person. Regardless, we were confident that our system would have an extremely low latency, so we chose to try to exceed this 10ms figure. Our testing setup for this was relatively simple. We used an audio interface that can record a pair of microphones, and had a beat ready to play while recordings were taken. One of the microphone inputs was an analog microphone, and the other input was connected to the headphone jack on our codec board. The idea was that we could play the reference audio on the speaker, and compare the difference between the time the sound arrives from the single analog mic against the audio going through our filter system and coming out of the DAC. Figure 7 shows the results of this test in audacity. You can see that the time difference between the analog mic on channel 1 and our platform on channel 2 is only about 3ms. This is totally imperceptible and exceeds the 10ms figure we wanted to be under. Low latency is one of the biggest strengths of this platform over other solutions.

For the second verification item, we wanted to be able to measure the performance of the geometric beam former. Unfortunately, our final design could not actually perform real time beamforming due to the filter limitations discussed in section 2.4.7. Therefore, we could not verify any specific attenuation figures.

The third verification item is related to calculating beam former coefficients in real time to reload on the platform. Originally we were going to be calculating the coefficients to perform the beamforming, but we changed our final demo to switch between two filters instead. The runtime is very comparable as the actual operation being done in software is almost the exact same. In our real time demo the filter adjustment time is almost imperceptible, so we are comfortable saying we meet this requirement.

Next, we wanted to ensure that the transitions between filter coefficients does not create a loud or annoying popping sound. This can happen because different filters will have a different total energy, and that change in output amplitude can be large enough to create a pop in the output audio stream. We could easily check this by recording the output audio while changing the filter coefficients. The results of this test are shown in Figure 6. As you can see, there is a small noise when the filters change, but it is neither loud nor distracting from the actual audio. We consider this requirement met.

Finally, we wanted the ability for our platform to calculate filter coefficients and reload them without user input. Our scripts and programs on Linux are written in such a way to support this kind of operation. We have scripts that sample user input, modify filters, and load those filter coefficients onto the board. It is certainly possible to modify our existing scripts to do more complex filter calculations using a python script. We are satisfied with the flexibility of our assortment of software features, and computing/loading filter coefficients automatically is well supported.

3.2 Volume Control

The volume control subsystem was a bit different from our initial plan, but we were still able to demonstrate the performance. Using a single slider was sufficient to test our verification items.

Our first verification item was related to the sampling resolution of our sliders. We wanted to have at least 20 levels on the sliders to give the users a relatively smooth volume adjustment control. The arduino samples the sliders at an 8-bit resolution, and these samples are sent over I^2C to the HPS. This gives the user 256 levels of control on each slider. From the output of our real time filtering demo script, you can see the slider takes values between `0x00 - 0xFF`. This level of control exceeded our requirement.

The other requirement for our slider system was related to the sampling rate of the slider levels. We wanted the HPS to sample the sliders at a rate of 1000 samples/second. To verify this, we read 1000 samples from the arduino over the I^2C bus. This is a good approximation of the transfer speed we would get while doing an actual demo with user input. Figure 4 shows that we were able to read 1000 byes in under one second.

3.3 Audio Codec

The audio codec underwent significant changes from our intial design, so some of the verification items are not applicable to the current implementation. However, we can still observe the safety of the output level through headphones.

4 Cost

The total cost of the whole project amounted to $18,194.94, consisting or $194.94 in parts cost and $18,000 in labor costs. The breakdown of each costs is broken down in detail below.

4.1 Parts

The parts list is listed below in table 2. Both the retail and actual costs are shown. The difference between the actual and retail costs is the retail cost contains free parts from the ECE 445 or Ryan Corey's lab.

The total retail and actual cost accumulated to $273.12 and $194.94 respectively.

Table 1: Parts Costs

Item	Part Number	Quantity	Retail Cost ($)	Actual Cost ($)
FPGA	DE10-Nano	1	135.00	135.00
Ribbon cables	485-4170	6	1.62	1.62
16-pin headers	649-71918-116LF	12	17.52	17.52
Potentiometer	858-PS6020MC1BR10K	4	9.8	9.8
ATmega328P*	SparkFun RedBoard	1	20.32	0
PCB		7	5.00	5.00
Amplifiers	RC4580	6	2.58	2.58
DAC	DAC8551	2	12.26	12.26
ADC	PCM1863DBT	2	11.16	11.16
Resistors*	Variety Pack	1	12.93	0
Capacitors*	Variety Pack	1	7.95	0
Jumper Wires*	Variety Pack	1	6.98	0
Binaural Headphones		1	30	0
Total			273.12	194.94

Items with asterisks were available at no cost from the ECE 445 or Ryan Corey's laboratory.

4.2 Labor

Our labor costs are estimated to be $40 per hour, 20 hours per week, 10 weeks per semester for three people. We accomplished 75% of our design this semester:

$$\$40 \cdot 20 \cdot 3 \cdot 10 \cdot 0.75 = \$18,000 \tag{1}$$

5 Conclusion

This ECE 445 project made a significant impact on the future of the Augmented Listening Lab at UIUC. The hardware platform constructed over the course of this semester allows researchers here at Illinois and other universities to run audio experiments faster and with more precision. Very few existing open-source audio hardware platforms can provide flexible filtering capabilities across 16 or more channels of analog and digital microphones.

Although there is more work to be done, this project demonstrates that this hardware platform can be beneficial to audio and hearing aid researchers for years to come upon its open-source release.

5.1 Accomplishments

Significant accomplishments were made this semester. Using modified slowfilters the system can filter 16 channel audio in real-time. Additionally, in software the FPGA can reload coefficients seamlessly - augmenting the audio in real-time. By leveraging the ARM processor with the flexible FPGA fabric, the heart of the platform is optimized for speed. Further verification indicates the total latency in the platform is 3 ms - an astronomically low number for audio processing. This in and of itself is a big achievement.

5.2 Ethical considerations

As a team, we fully understand the importance of upholding the IEEE Code of Ethics and we take it very seriously [6]. We strive to develop and maintain our platform as sustainably as possible, and to protect our users privacy in accordance with IEEE Code of Ethics #1. Our project, beyond ECE 445, will be released as an open source platform for hearing aid researchers and manufacturers to take advantage of to improve accessibility of hearing aid technology to benefit all communities. No user's audio data will be collected on our system in accordance with IEEE Code of Ethics #5 and #6. We will take every precaution to uphold the IEEE Code of Ethics as we develop and release our platform to the public to benefit people who struggle with hearing impairments. The users of our platform will face no discrimination of any form in accordance with IEEE Code of Ethics #7.

Our mission as a team is to make the most accessible and safe open-source hearing aid technology to researchers and hobbyists worldwide. We strongly believe that our system can contribute significant welfare to the public in a variety of life-changing applications people use everyday.

5.3 Future work

There remains work to be done despite the accomplishments attained over the last 10 weeks. The custom Audio Codec PCB has several revisions to undergo. Namely adding I2C pull-up resistors across the bus spanning multiple devices, using a complete codec IC instead of breaking up the ADC and DAC and improve how clocks are shared among the ICs and FPGA by optimizing the routing.

The noisy audio that results from some combinations of coefficients generated by Ryan Corey's beamformer algorithm also must be fixed. Pipelining the multiplier (the longest combinational critical path in the logic) seems to be a potential solution. Finally, fully paramterizing our hardware modules will increase the flexibility and configurability of the platform for the end user. As we are targeting an open source release,

we believe it is important that the hardware can be reconfigured by changing a single parameter.

References

[1] Adafruit. "ADAFRUIT SILICON MEMS MICROPHONE BREAKOUT". In: (2016). URL: https://www.adafruit.com/product/2716.

[2] Arduino. "Ardunio UNO". In: (2021). URL: https://www.arduino.cc/en/Main/arduinoBoardUno>.

[3] Sheetal Burada. "Floating-Point-ALU-in-Verilog/Multiplication at master · CruzeBurada/Floating-Point-ALU-in-Verilog · GitHub". In: (2021). URL: https://github.com/CruzeBurada/Floating-Point-ALU-in-Verilog.

[4] Ryan Corey. "Microphone Array Processing for Augmented Listening". In: (2019). URL: https://www.ideals.illinois.edu/handle/2142/106227.

[5] Dan Gisselquist. "A better filter implementation for slower signals". In: (2021). URL: https://zipcpu.com/dsp/2017/12/30/slowfil.html.

[6] IEEE. "IEEE Code of Ethics". In: (2021). URL: https://www.ieee.org/about/corporate/governance/p7-8.html.

[7] Texas Instruments. "PCM186x 4-Channel or 2-Channel, 192-kHz, Audio ADCs". In: (2018). URL: https://www.ti.com/lit/ds/symlink/pcm1863.pdf?ts=1620178056188&ref_url=https253A252F252Fwww.google.com252F.

[8] Texas Instruments. "PCM510xA2.1VRMS, 112/106/100dBAudioStereoDAC with PLL and 32-bit, 384 kHz PCM Interface". In: (2015). URL: https://www.ti.com/lit/ds/symlink/pcm5100a.pdf?HQS=dis-mous-null-mousermode-dsf-pf-null-wwe&ts=1620249276753.

[9] Intel. "Cyclone V Device Overview". In: (2021). URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51001.pdf.

[10] MicroChip. "AVR128DA28T-I/SS". In: (2020). URL: https://www.mouser.com/ProductDetail/Microchip-Technology/AVR128DA28T-I-SS/?qs=O1C7AqGiEDnHgnKTQ6SQRADD.

[11] MicroChip. "PIC32 Audio Codec Daughter Card - AK4642EN". In: (2014). URL: https://www.microchip.com/DevelopmentTools/ProductDetails/ac320100#additional-summary.

Appendix A Requirement and Verification Tables

A.0.1 Microphone Array

Requirements	Verification	Verification status (Y or N)
All microphones are synchronized to the same bit/sample clock within a single cycle	1. Connect oscilloscope to 4 microphones' clock pins 2. Verify that clock skew is within one period	Yes?

A.0.2 Audio Codec

Requirements	Verification	Verification status (Y or N)
Input pre-amplifier amplifies microphones to a line level between 0.75Vpp and 1.0Vpp.	This can be verified using an oscilloscope. 1. Play loud tone on speaker near microphone to represent maximum input volume 2. Use averaged oscilloscope measured output to verify level is between 0.75Vpp and 1.0Vpp	Yes?
Output volume level does not exceed 85dB SPL	85dB SPL is the threshold for hearing damage. 1. Play loud tone through input microphones with no filtering 2. Place earbuds in headphone SPL measurement dummy ear canals 3. Adjust output volume slider to maximum 4. Ensure output does not exceed 85dB on dummy sensor output	Yes?
Output signal noise floor does not exceed 10dB SPL	1. Disconnect microphones from board input 2. Place earbuds in headphone SPL measurement dummy ear canals 3. Measure SPL from dummy sensors output	Yes?

A.0.3 FPGA

Requirements	Verification	Verification Met

Source to listener latency must be less than 10ms	1. Audio source is connected to a speaker and oscilloscope 2. Audio output is also connected to the oscilloscope 3. Observe latency between audio captured and processed through the system and the reference signal	Y: figure REF shows 3ms latency through the system. Section REF speaks more about latency.
Filters should be able to achieve -6dB attenuation on suppressed sources	1. Record three sounds being played at 900Hz, 1000Hz, 1100Hz 2. Plot energy spectrum 3. Ensure that suppressed sources are at least 6dB down from peak of main source 4. Repeat test with each of the frequencies being attenuated	N: We could not perform geometric beamforming tests due to REF SECTION
50ms time to calculate filter coefficients	1. Set up selective listening demo 2. Run Python script with unix `time` command. This will measure time elapsed	Y: Filters can be modified in real time
No pops when reloading filter banks	1. Take sound recording 2. Observe in audacity and ensure waveform maintains amplitude through filter transitions	Y: Figure REF shows only a small noise through filter transitions
Ability to use Python to compute filter coefficients	1. Python code should be able to get samples from SD card 2. Python code will export filter coefficients to a .csv file 3. Bash script will glue together the Python/C++	Y: Real time filter loading from .csv performs well

A.0.4 Volume Control

Requirements	Verification	Verification Met
Volume sliders should have at least 20 levels for fine control	1. Write small shell script to display slider level over `ssh` 2. Move slider and ensure that volume increments in quantities of less than 5%	Y: See figure REF for script output and explanation.
I^2C interface is capable of sending volume levels 1000 times per second	1. Read 1000 bytes from arduino over I^2C using `i2ctransfer` 2. Use unix `time` to ensure the time to read is less than 1 second	Y: Transfer speed exceeds 1000 transactions/sec. See Figure REF

Appendix B Figures

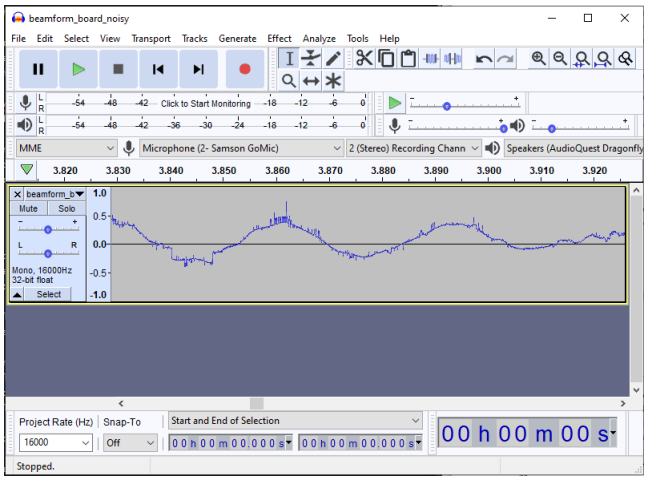

Figure 3: Impulsive noise in beamformer output

```
root@cyclone5:~/Code# time i2ctransfer 1 r1000@0x13 > /dev/null
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will send the following messages to device file /dev/i2c-1:
msg 0: addr 0x13, read, len 1000
Continue? [y/N] y

real    0m0.989s
user    0m0.006s
sys     0m0.000s
root@cyclone5:~/Code#
```

Figure 4: Speed of a 1KByte I^2C transfer

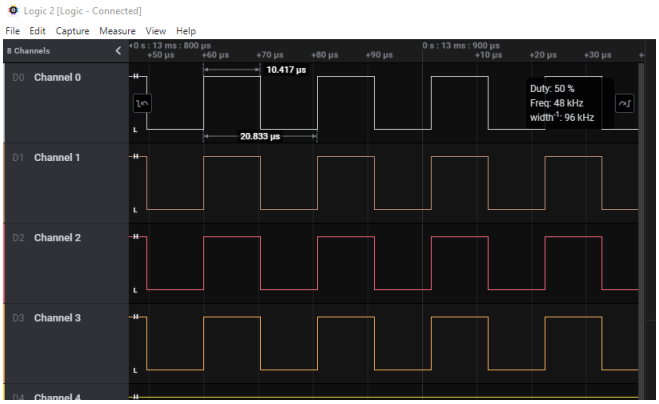

Figure 5: Logic analyzer trace showing four MEMS mics with synced clocks

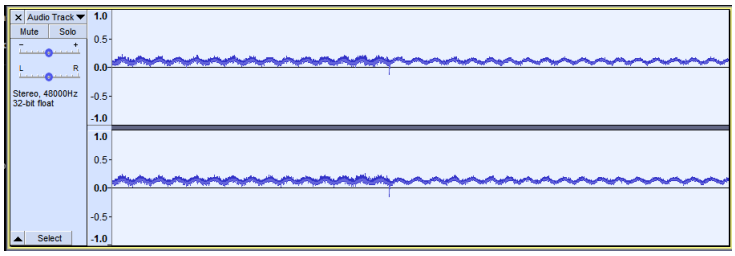

Figure 6: Audio showing filter transitions

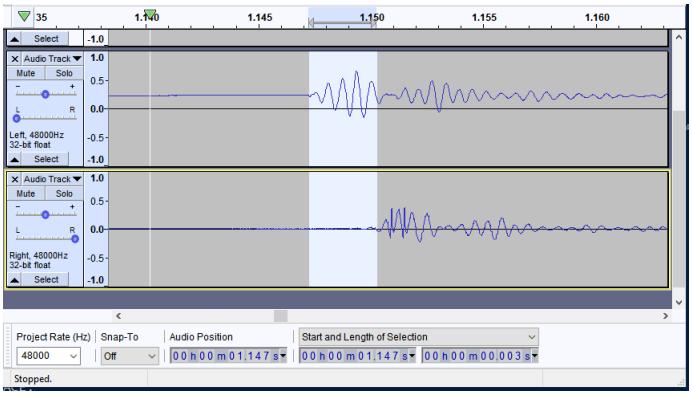

Figure 7: Audio signal showing latency of 3ms through our processing platform

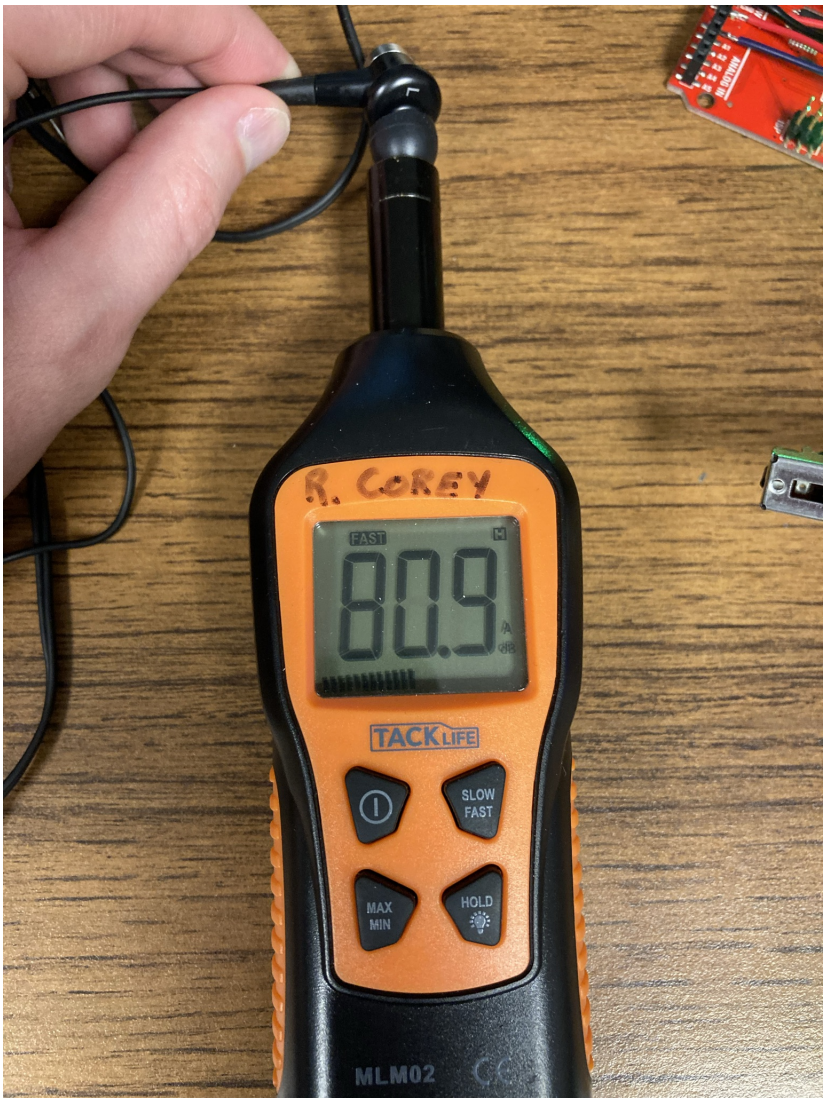

Figure 8: Sound pressure level reading showing output level does not exceed 85dB

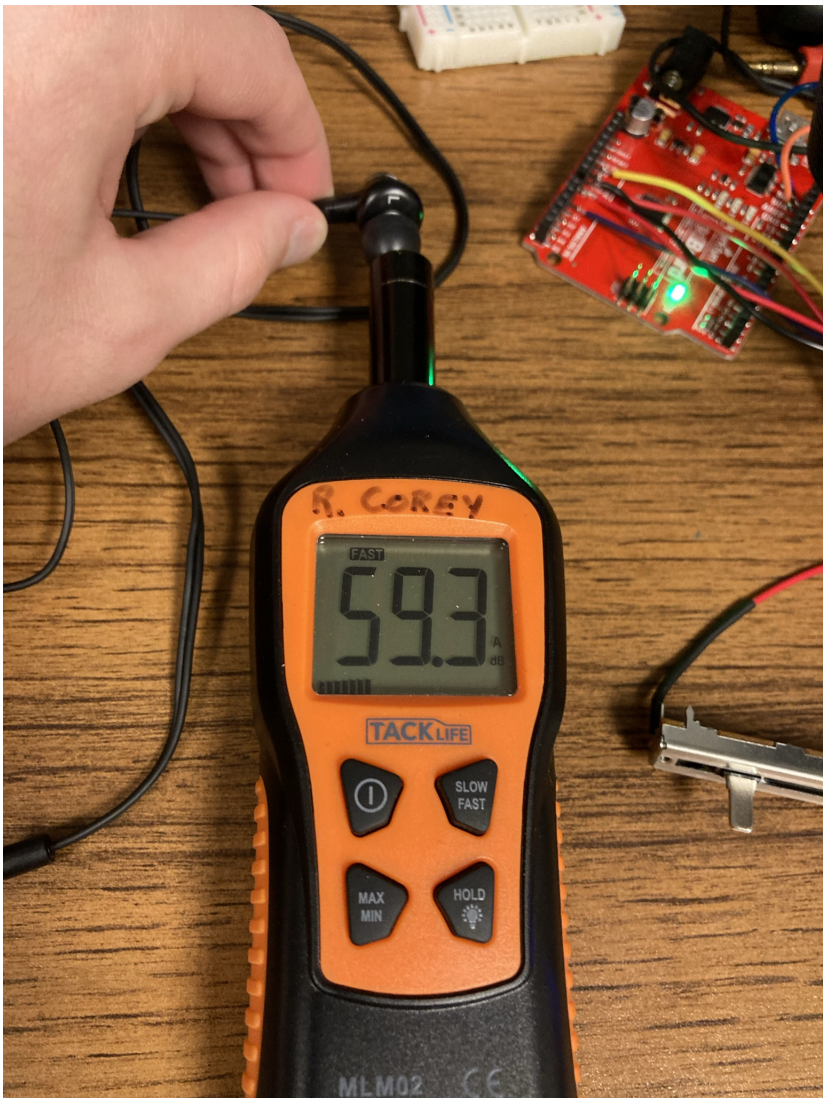

Figure 9: Sound pressure level reading showing output noise floor

Appendix C Microphone Array Schematic and PCB

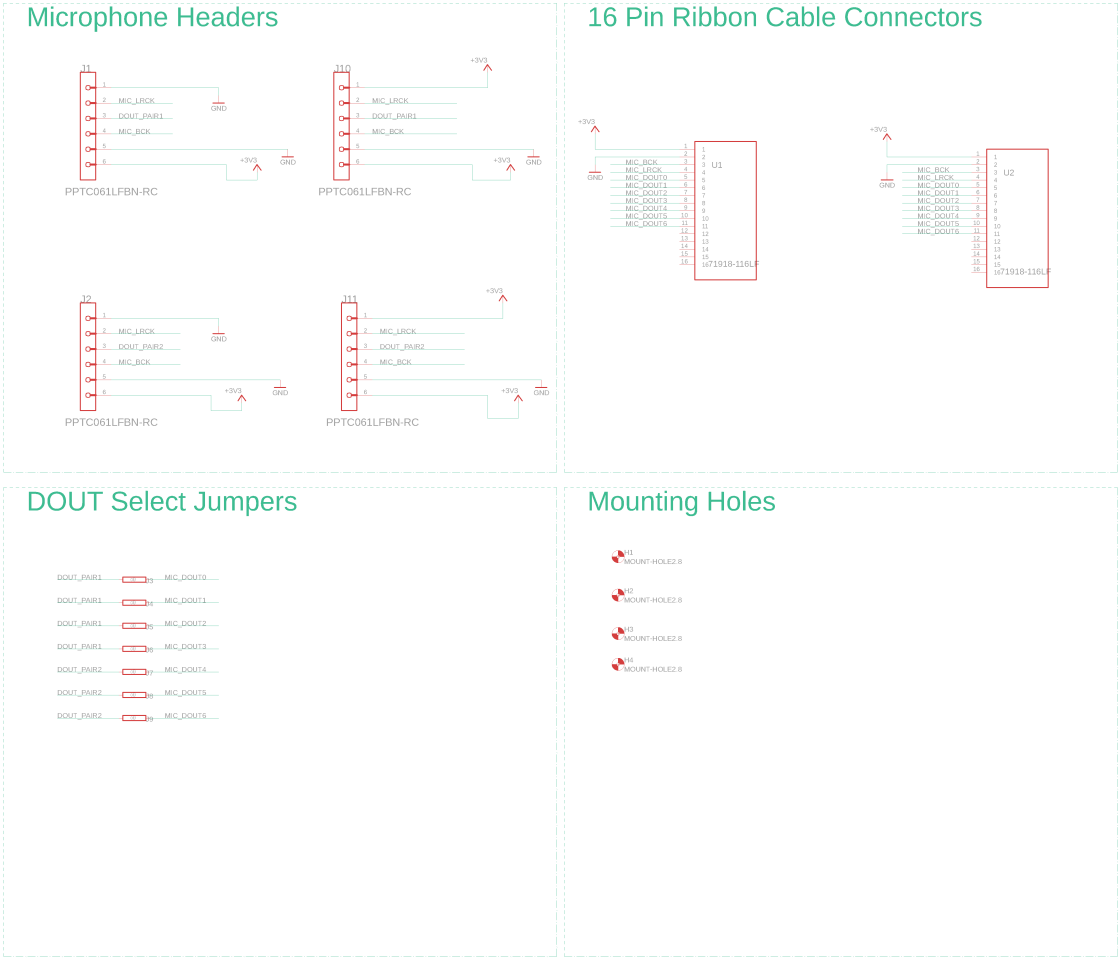

Figure 10: Schematic of Microphone Array

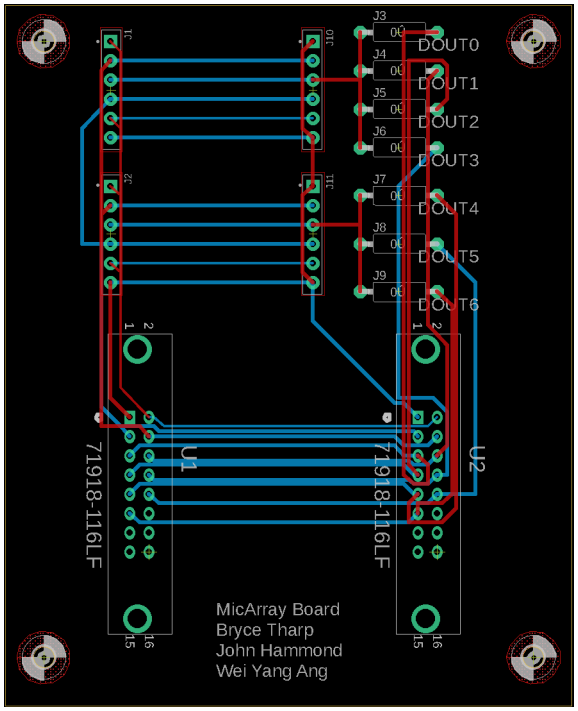

Figure 11: Microphone Array PCB Design

Appendix D Audio Codec Schematic and PCB

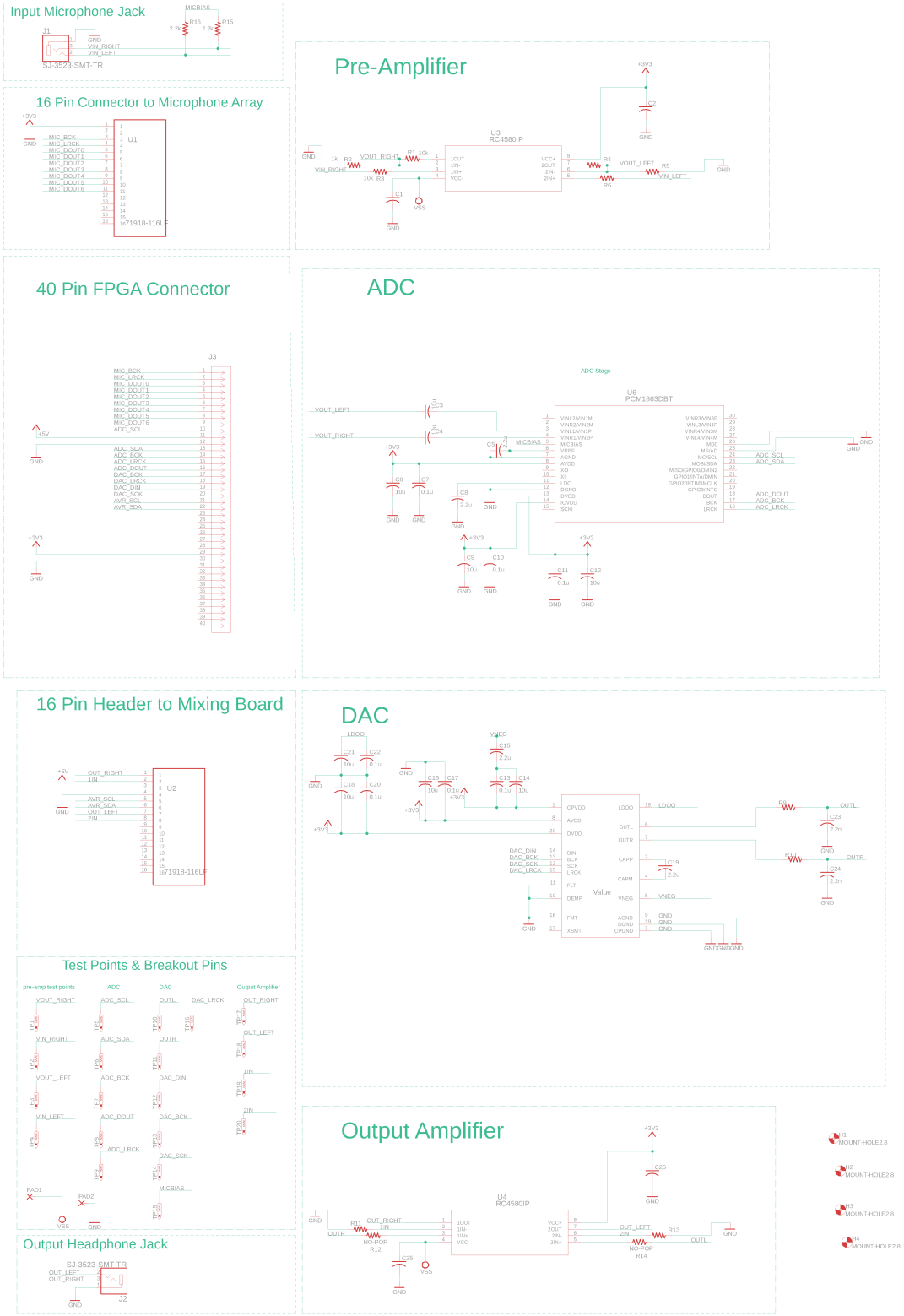

Figure 12: Schematic of Audio Codec

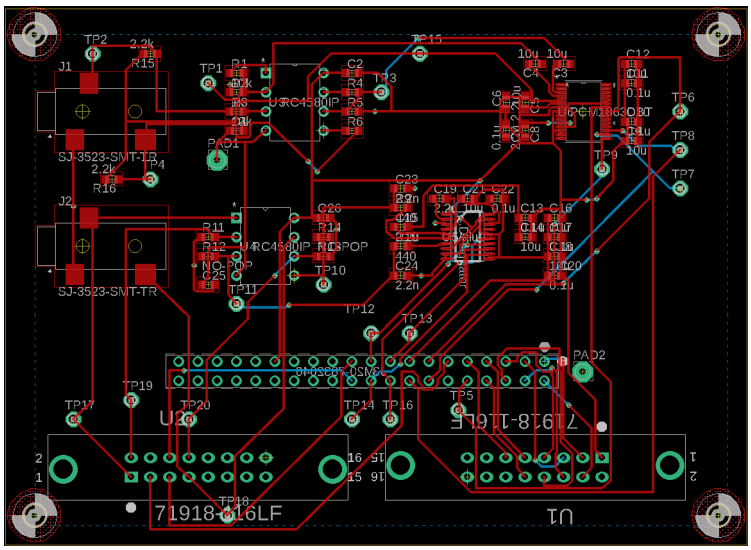

Figure 13: Audio Codec PCB Design

Appendix E Volume Control Schematic and PCB

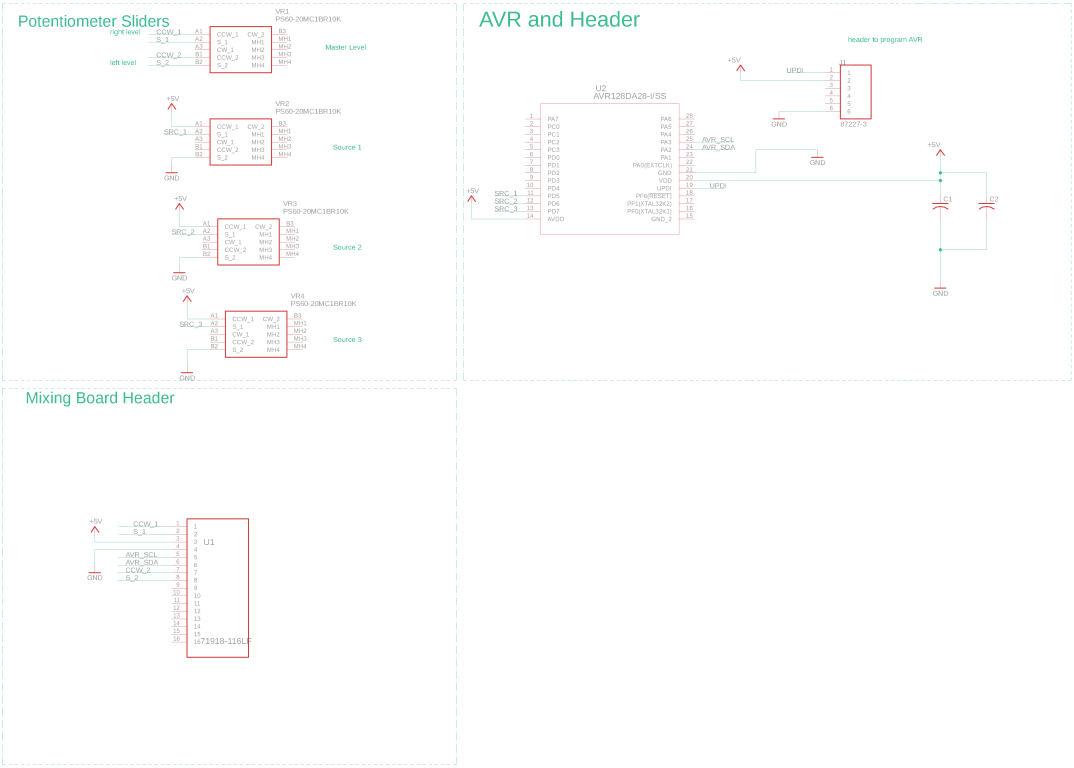

Figure 14: Schematic of Volume Control Board

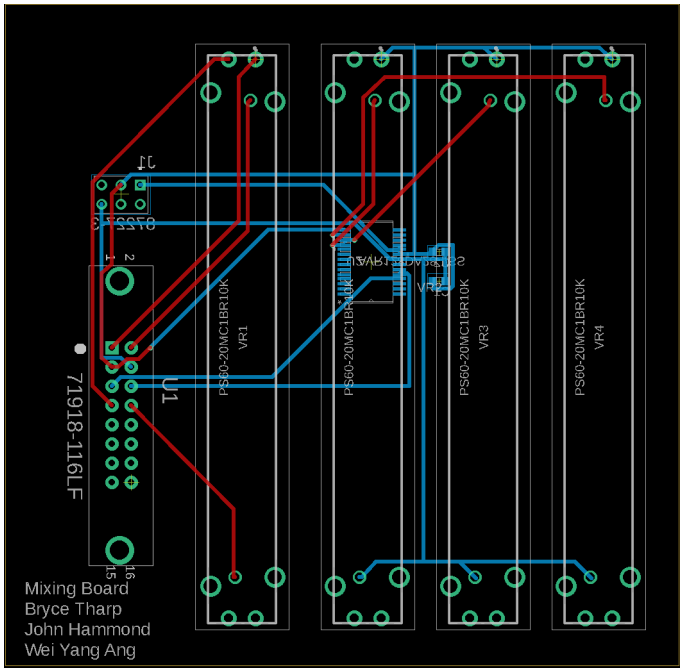

Figure 15: Volume Control Board PCB Design