Selective Listening

ECE 445 Design Document - Spring 2021

John Hammond, Bryce Tharp and Wei Yang Ang

TA: Evan Widloski

Contents

1 Introduction

1.1 Problem and Solution Overview

Imagine you are in a noisy environment (a restaurant, concert venue, etc) and you want to listen to one person or group in particular. For individuals who struggle with hearing loss this can be a challenging situation, even with modern hearing aids. Most hearing aid improvements are developed by corporations that are shrouded in proprietary technology and are inaccessible to researchers working on this complex issue. In addition, hearing aid research is particularly difficult because there are not readily available hardware platforms that are fast enough or suitable for real-time experiments. Our solution is to construct a comprehensive, open-source hardware platform for general-purpose research use.

The listening platform we are designing in ECE 445 is a culmination of a large body of research and development done by Ryan Corey and members of the Augmented Listening Laboratory (Bryce Tharp and John Hammond are members on the team). In the context of this course, we will build the supporting hardware systems necessary to create a flexible listening platform and showcase a solution to the Cocktail Party Problem - "the task of hearing a sound of interest in a noisy auditory environment" [1]. We will use microphone arrays composed of over a dozen microphones, a powerful audio processing unit powered by an FPGA (Field Programmable Gate Array) and an audio mixing board to tune the level of each source present in the system.

The solution we are developing for ECE 445 to solve the cocktail party problem is particularly unique. No other listening device on the market today comes with a real life mixing board to augment individual sounds in real time. We have a suite of programmable logic with proven fast and accurate FIR filters and we are leveraging a powerful DSP algorithm developed by Ryan Corey [2] to improve listening technology for years upon the open source release of the Augmented Listening Platform.

1.2 Visual Aid

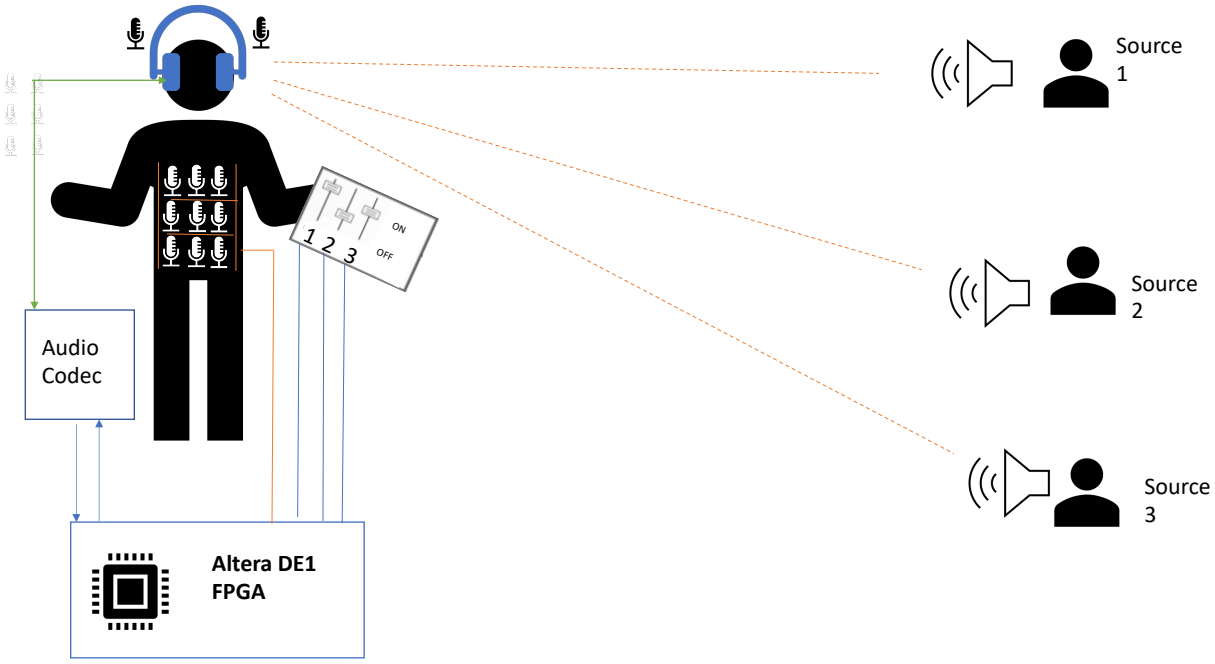

Figure 1: User diagram

1: The binaural headphones will be worn by the listener in order to preserve spatial cues. In this way, the listener will be able to naturally recognize noise regardless of their orientation in the sound field.

2: The microphone array will consist of 14 digital MEMS microphones [2] directly connected to the FPGA GPIO pins. These microphones will help the beamformer reduce noise more effectively.

3: The FPGA will be portable audio processing unit capable of filtering audio and providing unique coefficients to construct the desired beamformer.

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.

2 Design

2.1 Block Diagram

Figure 2 shows a block diagram of our entire system. The main components to look at here are the microphone array and the audio processing subsystem. Samples are fed from the microphones, through the FIR banks in the FPGA, and back to the headphones for the user to listen to. This basic platform has many uses, but for the sake of this class we will be showing the selective listening demo.

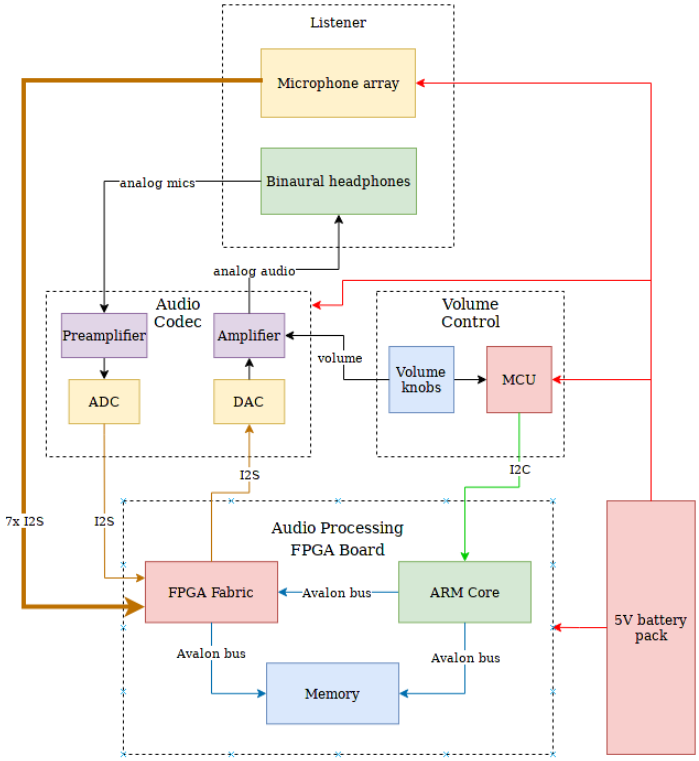

Figure 2: Block diagram of full listening platform

2.2 Subsystems

2.2.1 Listener

The listener is the user of our wearable microphone array and audio processing system. They will wear a pair of binaural headphones and affix a microphone array to their chest.

2.2.2 Binaural Headset

A binaural headset is simply a pair of headphones with a microphone positioned at each ear. We will be using a readily available pair of binaural earbuds in our design. The binaural headphones will serve two purposes. The user will listen to the processed audio through the earbuds, and the microphones at each ear will serve as a part of the complete microphone array. Most binaural headsets have analog microphones at the ears, and we will also use a pair with analog mics.

2.2.3 Microphone Array

The main microphone array will be a static grid of 14 small MEMS microphones. These MEMS microphones have a built in ADC so they will connect directly to the FPGA over I^2S. We chose 14 microphones here to keep a total array size of 16, including the binaural 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 will be building a larger board to affix each breakout board to, and we will daisy chain these larger boards together to form the complete array. A 16-pin ribbon cable will connect each of these larger PCBs together, but the absolute minimum size is 11 conductors. We chose 16 conductors because cables are readily available.

The geometry of this array must be carefully designed for our beam former and other future applications to work well. The shape of the array is not so important as long as the mics are in fixed positions relative to one another. Varying these positions significantly increases the complexity of beam forming and we will not look to solve that problem with this demo. Inter microphone spacing is the most important specification for the array.

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 will share the same conductor for both of these clock signals, which will limit our skew to that caused by the wires themselves. Our longest cable run will be from the end of the microphone array daisy chain back to the FPGA, which will be no greater than 14 feet with the cables we chose. We do not believe there will be a significant signal degradation over this short distance at the modest frequencies audio signals use.

Requirements	Verification
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

2.2.4 Audio Codec

Our audio codec will be made up of four main components. First, we will have a pre-amplifier to bring the binaural microphone outputs to line level. Then, an ADC is needed to sample these microphones for processing on the FPGA. Aditionally, a DAC will be needed to convert the filtered audio back to analog to play on the headphones. Finally, there will be an amplifier at the output of the DAC chip to allow the user to change the listening level.

A pre-amplifier circuit is needed to bias the two binaural electret microphones and amplify their output voltage levels to line level. Most off the shelf ADC ICs do not have a suitable amplifier circuit built in, and require an external circuit to amplify mic level to line level. The pre-amplifier is made up mostly of an op amp and supporting components. Figure 3 shows our premilinary schematic. We have not selected resistor values for our op amp circuit (figure 3) because we have not yet measured the output level of the microphones. Op amps are capable of a wide range of amplification levels so we are not concerned about getting one that won't fit our needs.

The audio codec system will also feature a pair of ADCs to complete our input stage. These are necessary to sample our analog mics to create an I^2S stream to the FPGA. These ADCs will operate at our -10dBV line level input, which comes from the pre-amplifier circuit. We have been careful to choose a chip that complies with the 16-bit/48kHz I^2S sampling our FPGA uses.

The first component of our codec's output stage is a pair of DAC chips. These chips take the 16-bit/48kHz I^2S processed audio stream from the FPGA and convert it back to analog to be played on headphones for the listener. They will output at a nominal line level of -10dBV.

Finally, our output amplifier stage will function primarily to change the output volume. We chose to utilize the same op amp IC as the pre-amplifier for simplicity. This amplifier will use a simple two resistor amplifier circuit, but with the ability for the user to change the output level. This amplifier will be similar to the circuit in figure 3, but a potentiomenter will be inserted in the design for volume control. We also have not picked exact resistor values here, and we will make those decisions via experimentation on our first prototype.

Requirements	Verification
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
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 measurement dummy ear canals 3. Adjust output volume slider to maximum 4. Ensure output does not exceed 85dB
Output signal noise floor does not exceed 10bB SPL	1. Disconnect microphones from board input 2. Place earbuds in headphone measurement dummy ear canals 3. Measure SPL

The pre-amplifier design we have chosen is based off of the standard non-inverting amplifier design. The integrated chip in our pre-amplifier schematic is the proven and robust TI RC4580. It is a dual operational amplifier chip which will serve our need for two channels (right and left). As stated earlier, we still need to measure the output voltage of the analog headphones, but once we do we can calculate the operational amplifier gain needed to hit -10dBV.

Gain in a simple operational amplifier is calculated by:

$$A_v = \frac{V_{out}}{V_{in}}$$

Our design needs to output a voltage around line level (say 1V to simplify the problem), if we assume the input voltage hovers around 100 mV, then:

$$A_v = \frac{1V}{0.1V} = 10V/V$$

We can calculate the dB gain:

$$G = 20\log(10) = 20dB$$

As a result of this calculation, we can tune our resistor values to ensure we build a pre-amplifier circuit with 20 dB gain. In a non-inverting amplifier a good guideline for computing resistor values is:

$$A_v = 1 + \frac{R_1}{R_2}$$

The two outputs of the pre-amplifier shown below will connect to the $V_{in}R$ and $V_{in}L$ of the ADC IC shown in Figure 5.

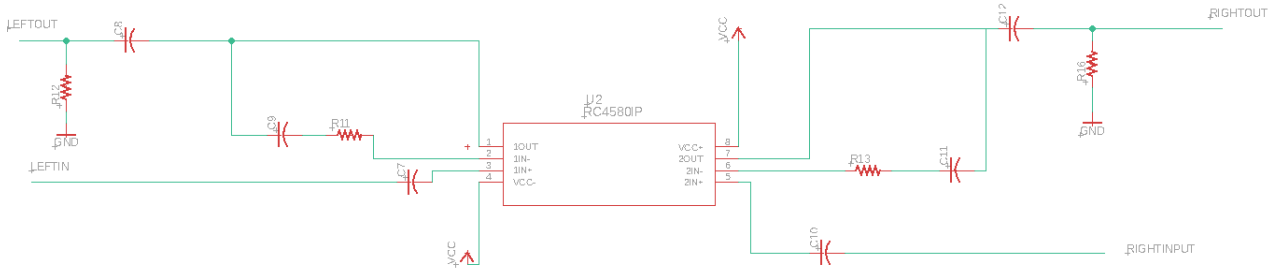

Figure 3: General Purpose Pre-Amplifier Design

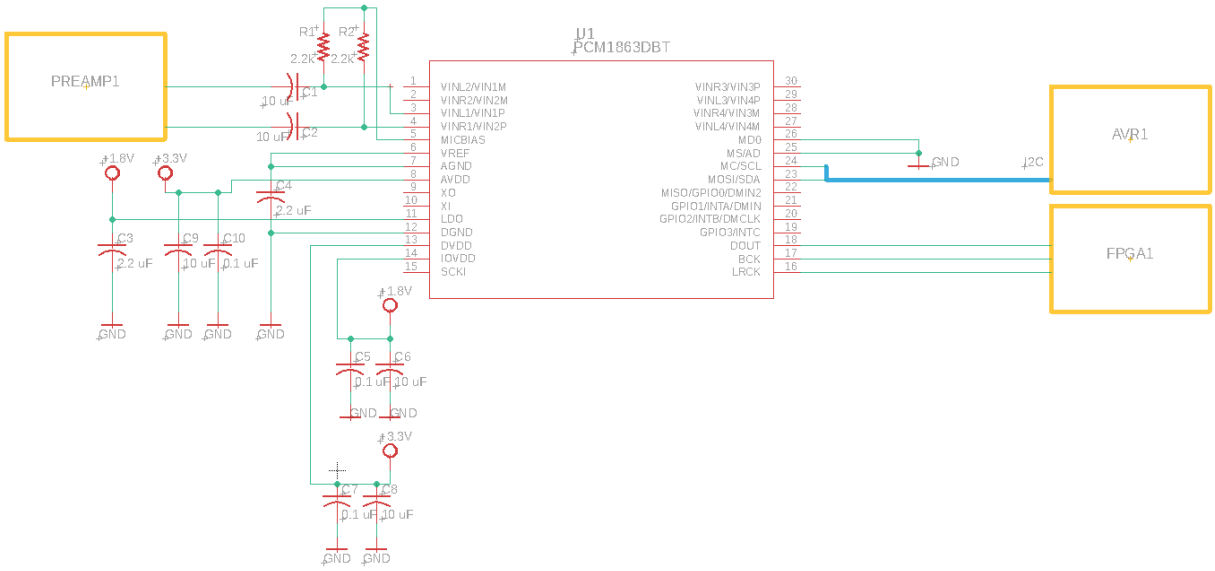

Figure 4: ADC Schematic

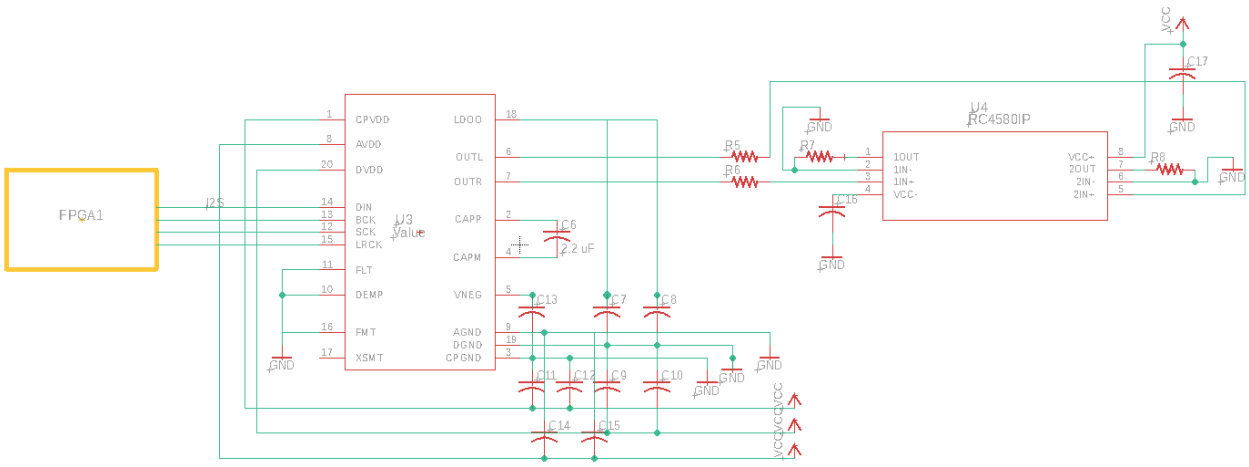

Figure 5: DAC Schematic

2.2.5 Volume Control

Our platform will feature a volume control interface for the user. In this specific beam forming demo, there will be four sliders: one will control the master output volume level, and the other three will be used to adjust the volume level of each individual source. Each slider is going to be sampled by a small AVR microcontroller, and the resulting levels will be sent to the FPGA via I^2C to factor into our beam forming algorithm.

The volume sliders were very straightforward with regards to design. Sliders like these are more specifically called fader potentiometers. We will place these potentiometers between the V_{ref} rail and ground, and connect the wiper pins to the A/D pins on the AVR. Our only design considerations for these sliders is the travel distance. A longer slider will give the user a greater resolution in volume levels.

The microcontroller for this subsystem also has few requirements. We chose to implement a microcontroller to sample the sliders to help take some of the design complexity away from the FPGA board. The DE-10 Nano board does feature A/D inputs that we could use, but writing code to use those would increase the size of our code base for the main platform, which is already large and difficult to manage. A quick program on the AVR micro will suffice to bridge the interface between the FPGA board and the sliders. The sliders will be sampled at a frequency of 1kHz to keep a responsive feel for the user.

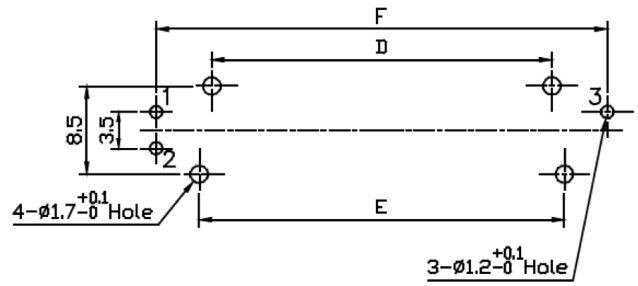

Figure 6: Physical Diagram of Potentiometer Sliders [3]

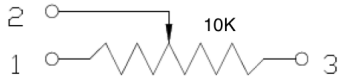

Figure 7: Simple electrical circuit for Potentiometer Sliders

The Mixing Board will be soldered onto a PCB along with the Audio Codec components. The potentiometer sliders output (pin 2 on figure 7) will directly plug into the AVR digital pins. Those signals will be sent via I^2C to the FPGA GPIO pins.

Requirements	Verification
Volume sliders should have at least 20 levels for fine control	1. Write small shell script to display volume level over `ssh` 2. Move slider and ensure that volume increments in quantities of less than 5%
I^2C interface is capable of sending volume levels 1000 times per second	1. Prepare a 16kB binary file 2. Stream it over I^2C from a shell 3. Use unix `time` to ensure the time to send is less than 1 second

2.2.6 Audio Processing Unit

The audio processing unit is the main computer of the Augmented Listening Platform, a Terasic DE-10 Nano board. The primary component is the Altera Cyclone V SoC Hard Processor System (HPS). The Cyclone V series of chips have both an FPGA built in as well as a dual core ARM Cortex-A9 processor. This setup is significantly more powerful than modern hearing aids, and allows users to prototype complex algorithms. The ARM processor is used to interface with on board peripherals and run our DSP code. On the other hand, the FPGA is used to accelerate FIR filtering, interface with I^2S peripherals, and record audio. Much of the FPGA codebase is taken from the Augmented Listening Project, though we will be expanding the functionality and polishing it for an open source release.

Like we mentioned above, the FPGA in our design serves a few purposes. The most important is the FIR filtering acceleration. One of the main goals of this platform is to be high performance and low latency, two areas in which an FPGA/HPS architecture is usually beneficial. More specifically, we want the ability to do time domain filtering with very sharp FIR filters (greater than 512 taps). Software only applications will usually utilize an FFT/IFFT to do processing in the frequency domain because it is faster on a general processor. However, with our time domain processing and dedicated FIR filter banks, we should be able to achieve source to listener latency of around 5 ms. Figure 8 shows the signal flow from source to listener through the FIR bank, as well as a calculation for overall latency. The filter outputs can either be recorded individually or summed together.

John spent a significant amount of time designing the filter architecture working as a part of the Augmented Listening Lab. This design requires sharp filters to support as many applications as possible, including our planned selective listening demo. However, the FPGA has a finite amount of logic units (ALMs) and hardware multipliers. Specifically, our Cyclone V chip only has 224 hardware 18x18 bit multipliers. Clearly this small number of multipliers would not be sufficient for a large number of basic FIR filters. Instead, we have decided to leverage an open source low area FIR filter written by Gary Gisselquist. This filter has two memories, one for taps and one for the samples "shifting through." A state machine iterates over these two memories and uses a single multiply/accumulate unit to perform the convolution. This approach is slow, but it is sufficiently fast for audio processing and only uses one multiplier regardless of the length of our filter.

Another advantage of having such a low area filter is the ability to use full IEEE-754 floating point math for all filtering. Our samples are 16-bit from the MEMS microphones and ADCs, so it would be possible to just do 16-bit FIR filtering. That is how the platform functioned in the past. However, researchers will greatly appreciate the convenience of using floating point taps from an end user perspective. Fixed point systems are comparatively limited in dynamic range, and floating point numbers are easier to understand. The Cyclone V is still able to create 112 32-bit multipliers so we are not concerned about the area tradeoff of this number system. John wrote a pair of SystemVerilog modules at both the input and output of the system to convert between 16-bit fixed point and 32-bit IEEE-754 floating point numbers.

Aside from the actual filters, the FPGA has logic in place to dynamically update the filters from the ARM HPS. A program running on the linux system can interact with the memory mapped Avalon interface to write tap coefficients to the filters. We will speak more about this process in the following paragraph, and also discuss more in depth some of the challenges of this approach.

The ARM HPS is the other half of the audio processing unit. While the FPGA is mainly for FIR acceleration, the ARM CPU is interfacing with outside devices and controlling the FPGA logic. Our ARM HPS is running linux, specifically a fork of Yocto linux called *rsYocto*. The linux system is responsible for networking, memory and storage management, and providing a platform for our C and python code to interface with the FPGA hardware. Section 2.3 describes how our python script will actually solve for the beam former, and figure 9 shows how some of our C code wraps around the python script.

A big challenge of this hybrid ARM/FPGA design is dealing with virtual memory. The ARM Cortex-A9, like most high performance modern processors, uses virtual memory through paging. This is great from a system security perspective and efficiency in memory allocation, but it presents challenges when we have one device (HPS) using virtual memory addresses and another (FPGA) using physical memory addresses. Thankfully, linux allows us to use `mmap()` to translate between virtual and physical addresses, and `/dev/mem` to touch memory directly.

The software running on the HPS is generally C/C++, python, and bash. We need to split our code base like this because python is not usable for doing low level memory operations (refer to the last paragraph). On the other hand, C/C++ is not great at doing high level signal processing and linear algebra math. Although projects like OpenBLAS exist they do not match the usability of something like python or `MATLAB`. Therefore, we decided to use C/C++ programs for memory access, python scripts for doing the actual signal processing work, and bash to glue everything together. A future goal of the project is to consolidate everything into either C/C++ or python but that has proven to be very difficult.

Requirements	Verification
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
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
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
No pops when reloading filter banks	1. Take sound recording 2. Observe in audacity and ensure waveform maintains amplitude 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++

2.3 DSP Code

The signal processing algorithm that we will be programming comes from Ryan Corey's PhD Thesis [2]. What follows is a mathematical overview of the algorithm.

The algorithm that is running on the ARM core of the Audio Processing Unit is designed to determine a unique set of coefficients to be loaded into the FIR filter banks to perform a signal processing operation called "beamforming". Beamforming is a speech audio process that can extract unique sounds in a noisy room. If we want to listen to a single source or multiple specific sources, we have to calculate the FIR filter coefficients for each source. The problem can be described by the following:

Let's say there exists N sound sources and M microphones, and additive noise in the system. In addition, we can also assume the sources and microphones are static so that the system is Linear-Time Invariant (LTI). If the system is LTI then we can describe it using convolution:

$$X_m = \sum_{n=1}^{N}(a_{m,n} * S_n)(t) + Z_m(t), \text{ for m = 1,...,M}$$

In the case of our beamformer, it is also an LTI system. Each signal has its own FIR filter, and since the system is linear we can sum the outputs of each filter:

$$y(t) = \sum_{m=1}^{M}(w_m * x_m)(t)$$

However, since we need two signals (for the left and right ears) we need two beamformers composed of multiple different filters per signal.

$$y_1(t) = \sum_{m=1}^{M}(w_{m1} * x_{m1})(t)$$
$$y_2(t) = \sum_{m=1}^{M}(w_{m2} * x_{m2})(t)$$

We can think about this problem in the frequency domain and create an "Acoustic Model". The system can be described in terms of matrix multiplication.

Acoustic Model:

$$\vec{X}(\Omega) = A(\Omega)\vec{S}(\Omega) + \vec{Z}(\Omega)$$

All of the above terms can be described as matrices (for instance $\vec{A}(\Omega)$ can be represented as an M x N matrix and $\vec{X}(\Omega)$ can be described as an M x 1 matrix). In the same way we can represent the beamformer:

$$(\Omega) = W(\Omega)\vec{X}(\Omega)$$

Let's first consider the simplest case: we want to separate all N sources such that M = N and no noise exists. We simply invert the channel matrix (A) to retrieve the sources back:

$$\begin{aligned} \mathrm{W}(\Omega) &= A^{-1}(\Omega) \\ &= \mathrm{A}^{-1}(\Omega)\vec{X}(\Omega) \\ &= \mathrm{A}^{-1}(\Omega)A(\Omega)\vec{S}(\Omega) \\ &= \vec{S}(\Omega) \end{aligned}$$

Next, let's add microphones such that M > N and add noise to the system. The "Power Spectral Density" of the noise must be considered. This density can be described as a co-variance matrix for random noise signals. One type of beamformer we will use to mathematically describe the system is a "Linearly Constrianed Minimum Variance" (LCMV) beamformer (however it should be noted that we can interchange this beamformer with others - the theory is very similar). The equation to describe such a beamformer is:

$$\mathrm{W}(\Omega) = [A^H(\Omega)\ \textstyle\sum_Z^{-1}(\Omega)\ \mathrm{A}(l)]^{-1}A^H(\Omega)\textstyle\sum_Z^{-1}(\Omega)$$

The LCMV is a "left inverse" of A meaning that:

$$\mathrm{W}(\Omega)A(\Omega)\vec{S} = [A^H(\Omega)\ \textstyle\sum_Z^{-1}(\Omega)\ \mathrm{A}(\Omega)]^{-1}A^H(\Omega)\textstyle\sum_Z^{-1}(\Omega) =$$

$$\vec{S}$$

One advantage of using the LCMV is that it minimizes noise power (remember the "Power Spectral Density" we included in the M > N case with additive noise). This beamformer blocks the noise power without affecting the channel noise.

In our demonstration we will have multiple sources, which means we will require multiple beamformers. However, let's consider a single source beamformer. A is the channel matrix and is commonly considered as the Acoustic Transfer Function.

$$\vec{X}(\Omega) = \vec{A}(\Omega)S(\Omega) + \vec{Z}(\Omega)$$

The single target LCMV beamformer is commonly known as the "Minimmum Variance Distortionless Response" (MVDR) beamformer:

$$\mathrm{W}(\Omega) = \frac{(\vec{A}^H(\Omega)\sum_Z^{-1}(\Omega))}{(\vec{A}^H(\Omega)\sum_Z^{-1}(\Omega)\vec{A}(\Omega))}$$

Now, let's scale this MVDR beamformer equation by the Acoustic Transfer Function of one of the microphone sources:

$$W(\Omega) = A_1(\Omega)\frac{(\vec{A}^H(\Omega)\sum_Z^{-1}(\Omega))}{(\vec{A}^H(\Omega)\sum_Z^{-1}(\Omega)\vec{A}(\Omega))}$$

The advantage of using this source microphone is that the filter will work regardless of how we scale the transfer function vector (by multiplying $A_1(\Omega)$ by any scalar). In other words, the beamformer output is estimating the sound that would have been recorded by the microphone if no noise was present. Regardless of the beamformer we choose, the relative transfer function that is substituted into the beamformer formulas will yield the same result as scaling the output to match the reference microphone.

In the system we are building, there are fourteen other microphones located on the listener's body. These extra microphones estimate the unwanted noise at each ear and cancel it out without changing the source the listener wants to hear.

$$\mathrm{X}_1(\Omega) = A_1(\Omega)S(\Omega) + Z(\Omega).$$

$$\begin{aligned}\mathrm{Y}_1(\Omega) &= X_1(\Omega) - Z(\Omega) \\ &\approx A_1(\Omega)S(\Omega)\end{aligned}$$

The beamformer we are building will be a "multiple-output" beamformer. Each source must be scaled by its own transfer function in each ear:

$$\mathrm{W}_{left}(\Omega) = A = \begin{bmatrix} A_{1,1}(\Omega) & & 0 \\ & \ddots & \\ 0 & & A_{1,N}(\Omega) \end{bmatrix} [\mathrm{A}^H(\Omega) \sum_Z^{-1}(\Omega) A(\Omega)]^{-1} A^H(\Omega) \sum_Z^{-1}(\Omega)$$

$$\mathrm{W}_{right}(\Omega) = A = \begin{bmatrix} A_{2,1}(\Omega) & & 0 \\ & \ddots & \\ 0 & & A_{2,N}(\Omega) \end{bmatrix} [\mathrm{A}^H(\Omega) \sum_Z^{-1}(\Omega) A(\Omega)]^{-1} A^H(\Omega) \sum_Z^{-1}(\Omega)$$

In addition to the binaural beamforming, we are building a binarual "remixing" system. This means the left and right outputs should be identically scaled combinations of the source components:

$$\mathrm{Y}_1(\Omega) = \sum_{n=1}^{N} G_n A_{1,n}(\Omega) S_n(\Omega)$$

$$\mathrm{Y}_2(\Omega) = \sum_{n=1}^{N} G_n A_{2,n}(\Omega) S_n(\Omega)$$

These mixing levels can be built into the binaural beamformer:

$$\mathrm{W}(\Omega) = \begin{bmatrix} G_1 A_{1,1}(\Omega) & \dots & G_N A_{1,N}(\Omega) \\ G_1 A_{2,1}(\Omega) & \dots & G_N A_{2,N}(\Omega) \end{bmatrix} [\mathrm{A}^H(\Omega) \sum_Z^{-1}(\Omega) A(\Omega)]^{-1} A^H(\Omega) \sum_Z^{-1}(\Omega)$$

On the FPGA we have constructed FIR filters that can be reloaded with new coefficients. Using this logic we will beamform each source separately, then scale and add up the outputs.

In order to estimate the Acoustic Transfer Function $A(\Omega)$ we will play a known pilot signal at each source location. This Transfer Function is the Fourier Transform of cross-correlation:

$$\vec{A}(\Omega) = \frac{\vec{X}(\Omega) S^*(\Omega)}{S(\Omega) S^*(\Omega)}$$

With that transfer function in hand we can play a pilot signal at each source and use the beamforming algorithm described to calculate the coefficients to load into the FIR filter banks. The software algorithm will be written in Python.

Figure 8 shows the forward model of signals propagating through the Audio Listening Platform. There are sixteen total microphones (two analog and fourteen digital) that listen to audio from the sources present. Then the analog stream is boosted to line level and converted to a digital format for processing. The FPGA logic contains the FIR filters that will process the coefficients computed from software on the ARM core. Finally, the signals will be augmented by the beamformers and converted back to analog signals for the listener wearing the binaural headphones.

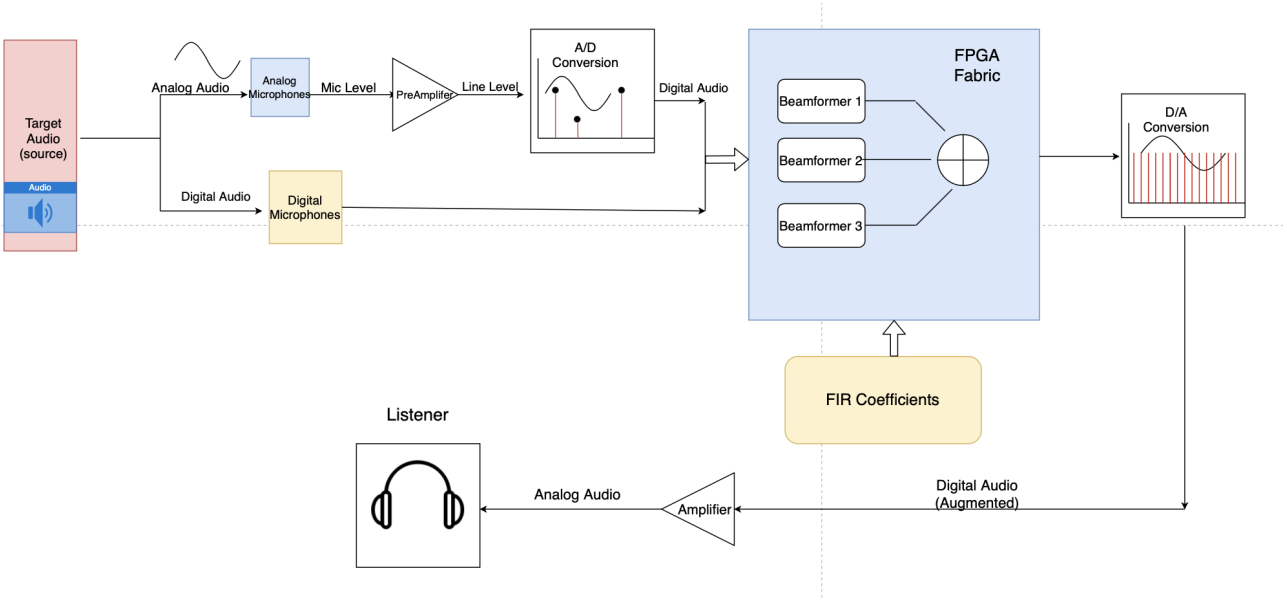

Figure 8: Forward Model of signals propagating through the Augmented Listening Platform

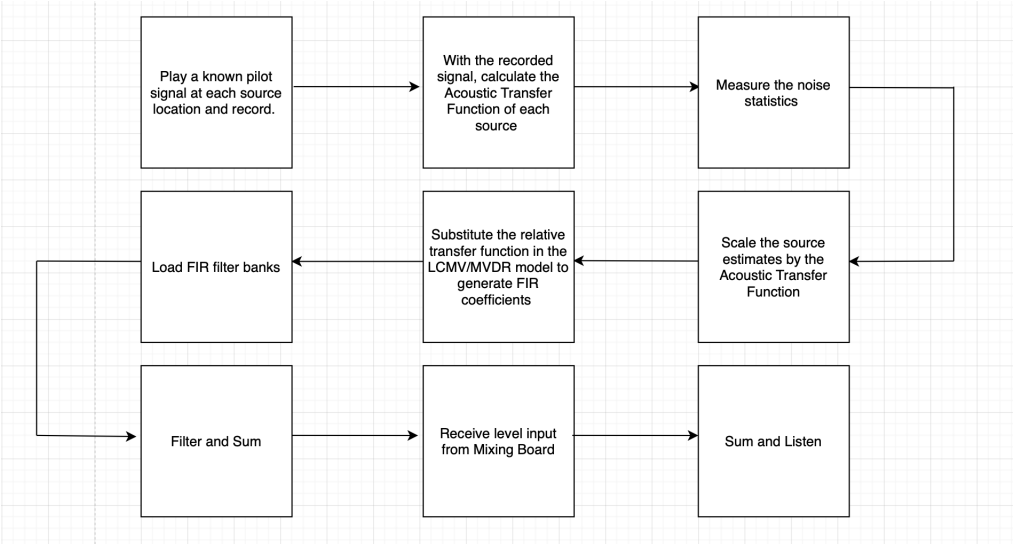

Figure 9: High level overview of the ARM beamforming program

2.4 Tolerance Analysis - Latency

One overarching goal of our project is to maintain a short source to listener latency. Research suggests that audio latency greater than 10ms will be noticeable by a listener and can be disorienting. To stay below this figure, we have made decisions at every step in our design process to minimize latency. In most signal processing systems, the largest source of latency is processing the audio. This can come in the form of FIR filters, IIR filters, or FFT/IFFT and freqency domain processing. Below we will show some of the steps in processing audio, as well as a breakdown of how we minimized latency.

Our platform can be represented as a linear system, as shown below. Each element in this system has some delay associated L. $Y_n(t)$ is the output of each filter, $H_n(t)$ is the impulse response of each filter, and $X_n(t)$ is the signal coming from microphone n.

$$\begin{aligned} Y_1(t) &= X_1(t) * H_1(t) \\ Y_2(t) &= X_2(t) * H_2(t) \\ &\vdots \\ Y_n(t) &= X_n(t) * H_n(t) \end{aligned}$$

Notice that we have chosen to do time domain convolution to implement our FIR filtering rather than frequency domain multiplication. Using an FPGA affords us this luxury. The filtering latency from our FIR banks is directly proportional to the number of taps in our filters. Below, N is the number of taps, $H[n]$ is the discrete-time impulse response, F_s is the audio sample rate, $X[n]$ is the input signal, and $Y[n]$ is the output signal.

$$L = (N-1)/2 \cdot (1/F_s) \tag{1}$$

This latency L is referred to as the *group delay* of the filter. Keep in mind this only holds true with a linear phase FIR filter, which has symmetrical coefficients. Applying equation 1 nets us a group delay of:

$$L = (512-1)/2 \cdot (1/F_s) = 5.3229\,\text{ms}$$

This latency figure is well within the perceptible 10ms. We are able to generalize the latency of any arbitrary filter because our design is strictly using linear phase filters. Linear phase filters are always symmetric about the middle of the filter, so we consider an output sample is valid once its input sample reaches the center of the filter.

Unfortunately, this group delay is an unavoidable consequence of using long, linear phase filters at relatively slow sample rates. Our FPGA implementation

is more than fast enough to do time domain filtering at 48kHz or faster. Below is a calculation showing the time to perform a single convolution. N is the number of taps, F_{clk} is the FPGA system clock, and M is the number of multiply/adds.

$$T = (1/F_{clk}) \cdot (N/M) = 10.240\,\mu s$$

This time is more than fast enough to complete a 512-length convolution within one clock cycle at 48kHz. We chose to use a single multiply/add unit to minimize area, however we could add additional units to significantly increase speed at the cost of area. Using an FPGA gives us the ability to do filtering now and adapt to even more sophisticated filters in the future.

3 Cost and Scheduele

3.1 Cost and Analysis

Our fixed development costs are estimated to be $40 per hour, 15 hours per week for three people. We plan to finish 70% of our design this semester:

$$2 \cdot \$40 \cdot 15 \cdot 16/0.7 = \$13,440 \tag{2}$$

Our parts cost is estimated to be $307.75 for the whole system, hence our total cost is $13,747.75.

Part	Number #	Cost Per Piece	Quantity	Total Cost
AVR128	579-AVR128DA28T-I/SS	$1.54	4	$6.16
FPGA	DE10-Nano	$135.00	1	$135.00
Ribbon cables	485-4170	$0.27	20	$5.36
16-pin headers	649-71918-116LF	$1.46	32	$46.72
potentiometer	858-PS6020MC1BR10K	$2.27	8	$18.16
PCB		$4.00	1	$4.00
Amplifiers	RC4580	$0.43	6	$2.58
DAC	DAC8551	$6.13	4	$24.52
ADC	PCM1863DBT	$5.58	4	$22.32
Resistors	Variety Pack	$12.93	1	$12.93
Binaural Headphones		$30.00	1	$30.00
Total				$307.75

Figure 10: Parts List

3.2 Schedule

Week	Bryce	John	Wei Yang
8-Mar	Primary PCB Design	Microphone Array Prototyping	Work on Bash Script
15-Mar	Complete PCB Layout Codec Board	Expanding FPGA Functionality	Coding the Filter Coefficients
22-Mar	PCB Validation and Soldering	Expanding FPGA Functionality	Coding the Filter Coefficients
29-Mar	Make Changes for PCB / Full Validation and Soldering	Linux Device Tree, Software Validation	Software for the AVR / Codec
5-Apr	Integration of all Components	Integration of all Components	Integration of all Components
12-Apr	Testing of Device	Testing of Device	Testing of Device
19-Apr	Final Changes	Final Changes	Final Changes
26-Apr	Working on Final Paper	Working on Final Paper	Working on Final Paper
3-May	Finishing Final Paper	Finishing Final Paper	Finishing Final Paper

Figure 11: Detailed Schedule

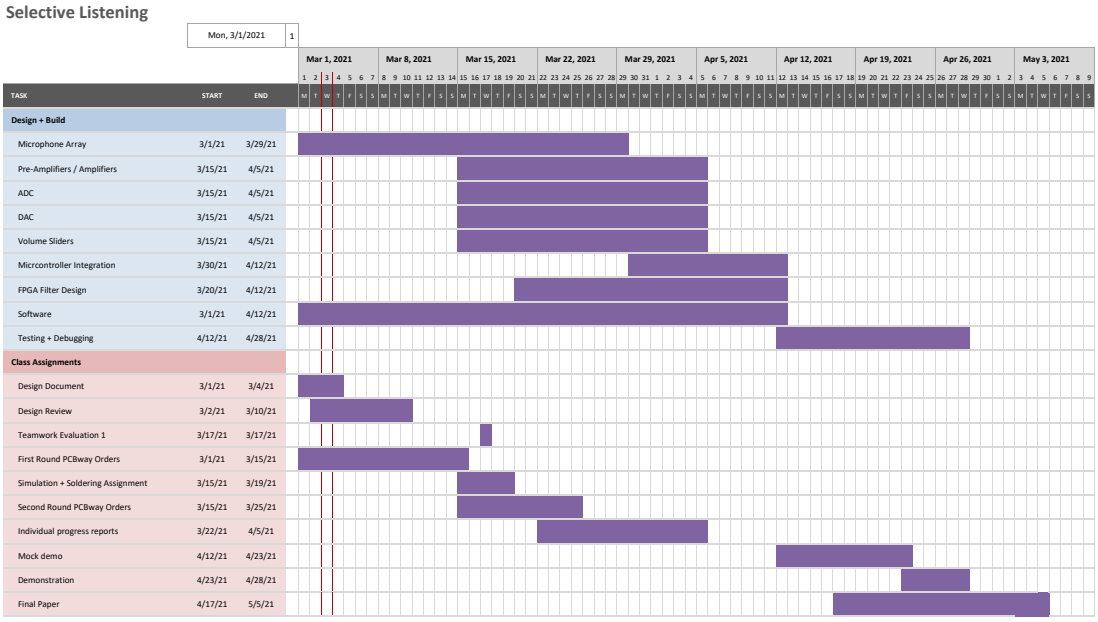

Figure 12: Gantt chart for schedule

4 Discussion of Ethics and Safety

Our project could present multiple safety hazards. One such hazard is our power system. We are using a rechargeable battery pack that could cause a fire hazard if not cared for properly. If the power system experiences any physical damage there poses a risk of short circuiting the device, or overcharging the device [4]. If the device is short circuited or overcharged it could overheat and damage the board or even pose a fire hazard. Before plugging our battery pack into our system (i.e. the FPGA) we will test the battery pack to ensure it doesn't exceed 6V. Another risk that could pose a safety hazard to users of our platform is hearing damage. The users of our device will be wearing binaural headphones, with microphones located close to the ear canal. We will ensure that listeners using our product will not experience noises exceeding 85 dB [5]. One key activity that contributes to hearing loss is listening to sounds through headphones that are too loud [6]. Our team will take measures to eliminate the risk of hearing damage while using our platform.

As a team, we fully understand the importance of upholding the IEEE Code of Ethics and we take it very seriously [7]. We strive to develop our platform to be constructed sustainably, 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 Citations

1. "Cocktail Party Effect." Wikipedia, Wikimedia Foundation, 2 Mar. 2021, en.wikipedia.org/wiki/Cocktailpartyeffect.

2. Corey, Ryan Michael. "Microphone Array Processing for Augmented Listening." University of Illinois at Urbana-Champaign, 2019.

3. "PS60-20MC1BR10K BI Technologies / TT Electronics: Mouser." Mouser Electronics, www.mouser.com/ProductDetail/BI-Technologies-TT-Electronics/PS60-20MC1BR10K/?qs=sGAEpiMZZMukHu252BjC5l7YeT2c9CjZIR0dvTECv0j98Y3D.

4. Back, Jerry. "LITHIUM BATTERY SAFETY." EHS, www.ehs.washington.edu/system/files/resources/.

5. Rai, Amrit, and Naomi Dainty. "6 Simple Ways To Check If Your Headphones Are Too Loud." Deafblind UK, 18 Aug. 2020, deafblind.org.uk/6-simple-ways-to-check-if-your-headphones-are-too-loud/.

6. "Do Headphones Increase Your Risk of Hearing Loss? Facts You Should Know." Oklahoma Hearing Center, 17 Feb. 2021, okhc.org/headphones-hearing-loss/.

7. "IEEE Code of Ethics." IEEE, www.ieee.org/about/corporate/governance/p7-8.html.