Sound Localisation - Final Report

Motivation and Introduction 2

Comparison to MVP and Checklist 3

Proposed Checklist 3

Completed Tasks 3

General Block Diagram and Schematics 5

Module Specification 7

i2s interface - Joseph 7

bclk/ws generator - Joseph 8

IIR integrator - Joseph 8

Footstep detector - Joseph 9

Cross correlation calculator - Joseph 10

Argmax - Joseph 13

Location Wrapper - John 14

Location Predictor - John 16

Square Sub - John 17

B Square - John 18

K Numerator Calculator - John 18

Inverse Cosine - John 18

Final Calculation Step - John 18

Ambiguity solver - Joseph 19

Display - John 19

Footstep Direction Calculator - Joseph 20

Footsteps - John 21

Picture Blob - John 21

Challenges 22

Git, Version Control and Collaboration - John 22

Signal processing - Joseph 22

Math - John 25

Display - John 26

Advice, Future Work and General Reflection 27

Advice 27

Future Work 27

Reflection - Joseph 28

Reflection - John 28

Motivation and Introduction

The inspiration for this project started with our interest in signal processing, particularly from previously taken classes at MIT. As soon as we asserted that common interest, we knew that we wanted our project to be oriented around light or sound in some manner. Initial ideas included building a laser measurement system or attempting to measure the speed of light; however, our focus shifted to sound when, during one class, we were introduced to the idea of localising sound using an array of microphones. We enjoyed hearing about this idea and wanted to put our own twist on it.

After some back and forth, we landed on the idea of replicating the Marauder's Map from Harry Potter. The Marauder's Map is a magical document which shows the location of individuals by plotting their footsteps on a map. We wanted to do the same, but instead of using magic to find where the individual is, we would use audio, signal processing and math (which, at times, can seem like magic!).

The Marauder's Map from the Harry Potter Film Franchise

Research was then conducted on how this could be performed with as few microphones as possible. Many of the papers we read were written with a focus on Structural Health Monitoring (SHM), which includes identifying when and where cracks appear in objects by listening to the sound waves they produce. This process is known as Acoustic Source Localisation and can become very complex when dealing with anisotropic materials. Thankfully, we were just working with air, so the math became a lot more simple and we deemed it a feasible task to perform on the FPGA. Furthermore, we learnt that this calculation only needed 4 microphones, which was reasonable for the project.

And so that's the story of how the FPGA Marauder's Map using Sound Localisation was born. What followed was an incredible journey of learning, pain and, eventually, satisfaction, all of which is documented in the rest of this report. I hope you enjoy reading this as much as we enjoyed making the project.

Comparison to MVP and Checklist

Proposed Checklist

The proposed final checklist given before thanksgiving was as follows:

Math:

Minimum: Given cross-correlations, calculates a possible source location

Commitment: Resolves ambiguity using 4th microphone

Reach: None

Display:

Minimum: Displays footsteps on screen using simple shapes (eg. square)

Commitment: Footsteps fade over time; Use image of footstep

Reach: Given previous footstep, angles the next footstep to show 'direction' of travel

Signals:

Minimum: Simple constant threshold for peak detection

Commitment: Variable threshold based on noise, anti-aliasing

Reach: Fancier techniques like noise removal with a Wiener filter, handling quantization errors, a filter to make footsteps narrower in time. Maybe a threshold that takes more factors into account like signal energy. The goal is to be able to pick up as quiet of a footstep as possible in the noisiest environment possible.

Completed Tasks

Math:

The math portion of the project was completed as specified in the checklist. That is, given an input of 2 out of the 3 time difference of arrivals, the module would compute 2 potential points from which the footstep could have originated. To resolve the ambiguity between these two points, the calculation was performed again, but using a different subset of the 3 time difference of arrival values. This second calculation yet again computes 2 potential points, one of which, conditional on accurate inputs, is the same as one of the earlier points.

Those four points are fed into the ambiguity solver, which, to begin with, would find the two matching points and return one of them. However, it was often the case that the input was not as accurate as the module required and there would be no alignment of any points with each other, so a heuristic was implemented to compare the 4 potential points with the previously

detected sound. If none of the 4 points were within a reasonable distance to another point, then the closest point to the previous sound would be deemed as correct and sent to the display.

Display:

The display set the screen to show, at all times, the locations of the four microphones relative to the FPGA, which was represented as a white square in the middle of the screen. To aid in setup and debugging, the four microphones on the screen are coloured red, green, blue and purple to distinguish the orientation of the system. Additionally, the border and crosshair default display from Lab 3 was used as a starting point for this module, but was left in as it was deemed helpful.

One part of the commitment was to have the feet fade over time as they display on screen. This feature was implemented, but was deemed unnecessary, unhelpful and annoying after extensive use. To replace this feature, an upper limit on the total number of footsteps displayed on the screen was imposed; as more footsteps were detected, the oldest would disappear.

The second part of the commitment and the reach goal was to use an image of a footstep and angle it to represent the direction of travel. Not only was this task implemented but so was the ability to predict left and right feet based on standard walking patterns.

Signals:

The signals end definitely satisfies the minimum of having a fixed threshold to detect footsteps and then put them through a cross correlation. The filters listed in the proposed checklist didn’t end up being the most relevant ones to getting good results, so I opted to go for others. In the end, I made a filter to center the audio signal at 0 and a filter to detect rising edges in the signal energy instead of the standard detection of a large peak. Those combined with rejecting bad answers gave better answers in our system than I think doing fancier methods would have, given that we can’t characterize our noise very well, the microphones aren’t fantastic, and we were running out of time on the project near the end.

The new method gives much better time precision, but the noise rejection is currently pretty bad. I thought this was a necessary tradeoff since the time precision is so much better now.

General Block Diagram and Schematics

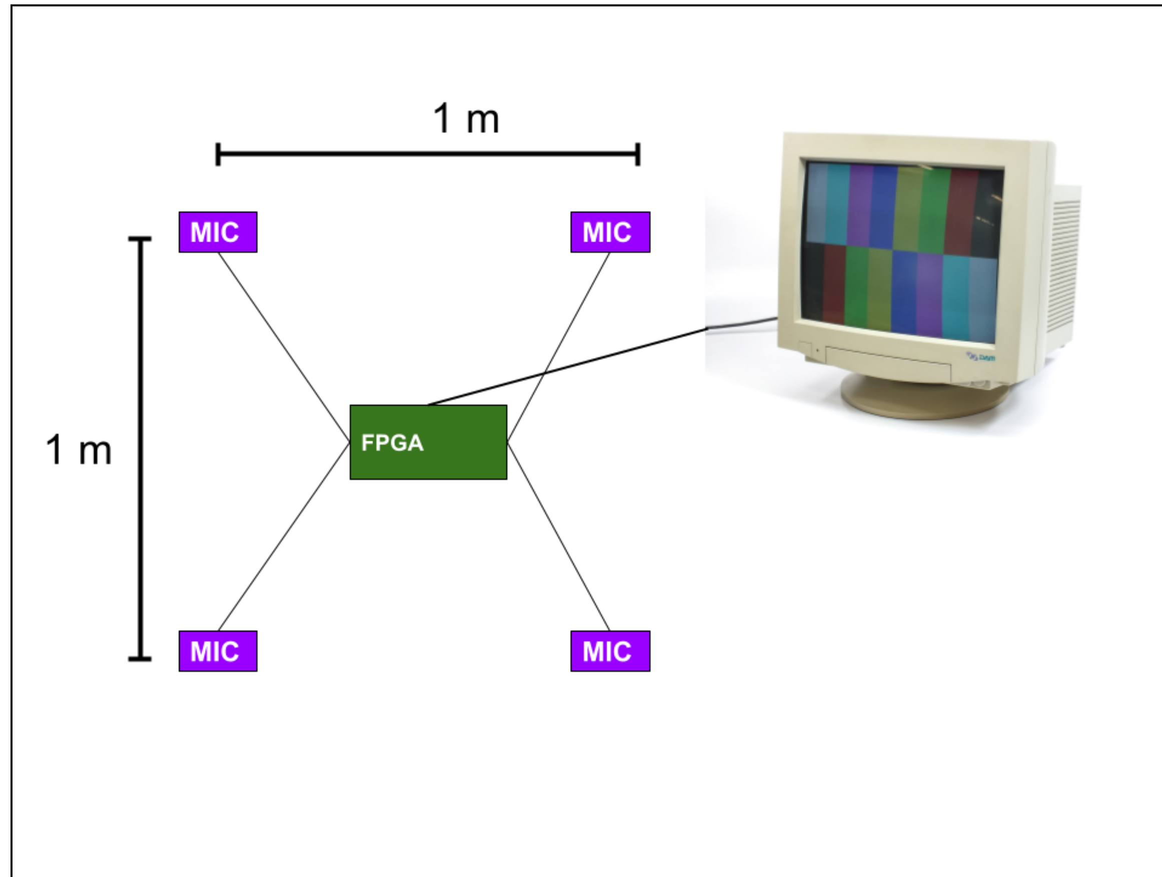
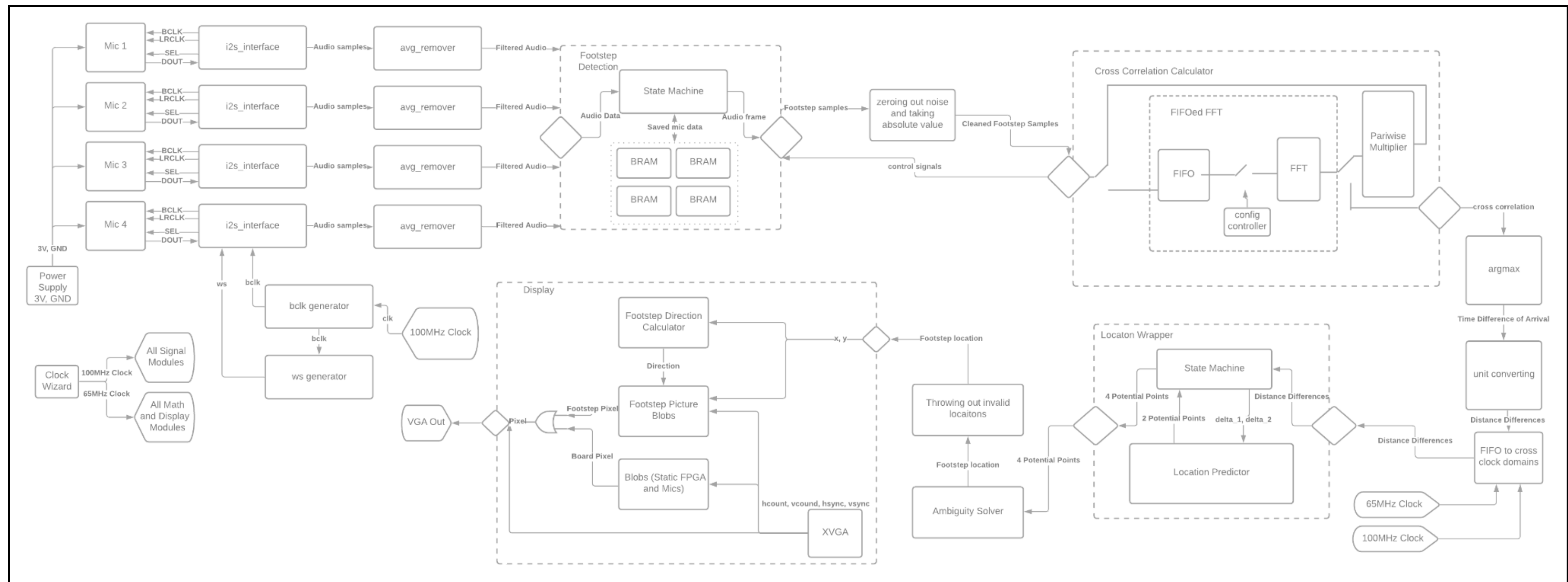


Diagram of the physical setup

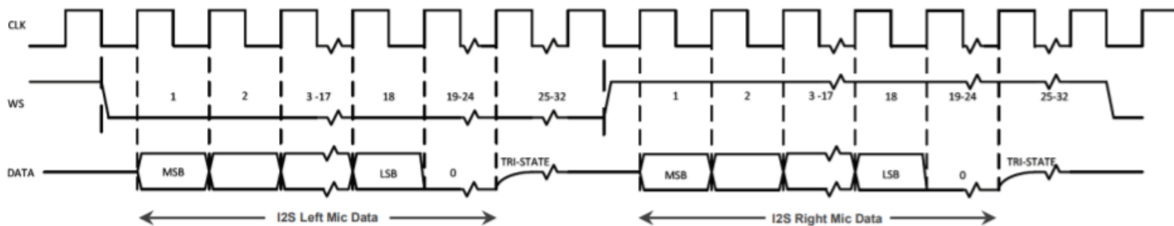


FPGA Block Diagram

Module Specification

i2s interface - Joseph

We used 4 SPH0645LM4H-B mics on their Adafruit evaluation board. They use their own version of i2s specified with this timing diagram:

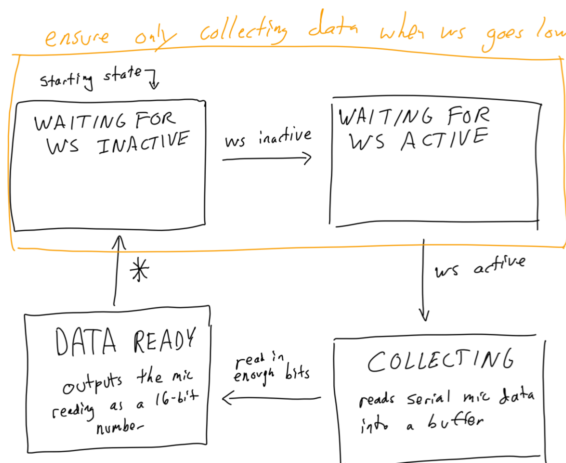


i2s Timing Diagram

The difference is that normal i2s does everything here that is on a rising edge on a falling edge and vice versa. This means we had to write a custom i2s interface to talk to these microphones. We avoided stereo issues by setting all the mics' select pins to low so they all act as left mics.

We chose these mics since they claim to be omnidirectional (that claim was dubious in practice), have a nice evaluation board, and aren't crazy expensive. We wanted to use mics with a digital instead of analog interface to avoid issues of picking up noise on our long wires from the mics to the FPGA.

Since the mic asserts data on the rising edge of the clock and there can be some delay, we read on the falling edge of the clock to ensure that we get the value that was actually asserted on the previous rising edge.



i2s Interface State Machine

bclk/ws generator - Joseph

The i2s interface requires clocks at specific frequencies: a bit clock (bclk) at 2-4 MHz and a word select (ws) clock at the bclk frequency divided by 64. Bclk synchronizes the serial communication between the FPGA and mic while ws tells the mic which stereo channel to send. For simplicity, we set all our mics to be left mics and didn't use the stereo features of i2s.

IIR integrator - Joseph

These microphones had an issue where values were centered at around -2000 instead of 0, so we made a simple averaging filter with this formula:

$$y[n] = \alpha x[n] + (1 - \alpha)x[y - 1]$$

We chose this filter for two reasons:

1. If we set alpha to be very small, the value will be very stable which would be harder to maintain with an FIR filter
2. Very little memory is required so implementation in verilog doesn't require any BRAM IP and is generally straightforward

Once we have the average, we can subtract it from the signal to get it centered at zero.

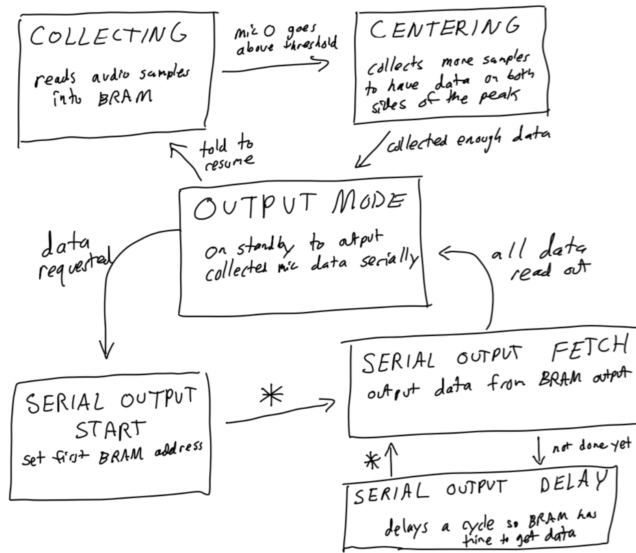
Footstep detector - Joseph

This module's job is to run peak detection to find a footstep, wait for some time to make sure every mic's peak is stored, and then be able to output all the data to the next stage in the pipeline.

The peak detection works with a simple thresholder on mic 0 controlled by a threshold set with the VIO IP. The shift duration is also set by the VIO. Audio samples are stored in a circular buffer in BRAM.

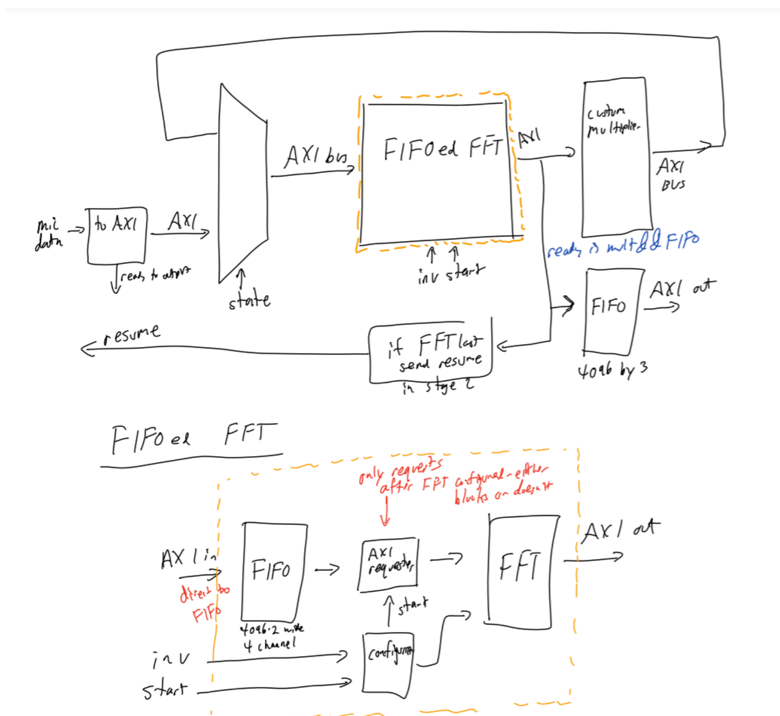
This module also has some counters that apply artificial delays between picking up individual footsteps since sometimes the echoes of one footstep would be picked up and seen as a second footstep.

In a final improvement discussed more in-depth in the signal processing section, thresholds on all mics are recorded. If there is too long of a delay between some, but not all mics seeing a peak, the peak detection will reset and search for a new footstep.



Footstep Detector State Machine

Cross correlation calculator - Joseph



Cross-correlation Calculator State Machine

This module moves the signal into the frequency domain to calculate the cross-correlation between each mic 0 and every other mic, yielding 3 cross-correlations. Some initial tests made us concerned that we wouldn't be able to fit the desired 7 FFT IPs on the FPGA so we decided to build a state machine around a 4 channel FFT. In hindsight it doesn't seem that having 7 FFTs would have been an issue, but this was still a good exercise in state machines.

This uses two state machines. The main state machine handles the high-level operations where it calculates the cross-correlation in two stages. First it gets the FFT of each of the 4 mics by piping the mic data into the FFT. Then that goes through the multiplier module which does the cross-correlation in the frequency domain using this identity about the Fourier Transform of the cross-correlation:

$$(x * \overleftarrow{y}) \leftrightarrow X(-j\omega)Y(j\omega)$$

The next stage puts the multiplier output into the FFT module to do an inverse FFT to move the signal back into the time domain. The data is then output to the next module.

The other state machine is a wrapper around the FFT that abstracts away some of the annoying quirks of the FFT IP and adds a FIFO to the front of it. Having a smaller wrapper for the FFT independent of the rest of the system meant that we were able to isolate debugging of the FFT IP from the rest of the system.

The FFT IP has a few quirks that make this necessary. First, to be able to do both forward and inverse transforms one needs to work with the FFT's config input. There are a lot of extra options in the FFT IP and the config interface is just a single giant data bus that you are supposed to put data into with a specific format laid out in the datasheet.

The main source of confusion here was that we had disabled all the options for what can be configured except for the part saying whether it is a forward or inverse transform. However, the config data input was still about a hundred bits long despite the forward/inverse transform option only taking 4 bits (1 bit per channel). The documentation seemed to suggest that you can put dummy data in the other sections to just offset the forward/inverse bits and then the others will be optimized away during synthesis:

TDATA Format

The configuration fields are packed into the `s_axis_config_tdata` vector in the following order (starting from the LSB):

1. (optional) NFFT plus padding
2. (optional) CP_LEN plus padding
3. FWD/INV
4. (optional) SCALE_SCH

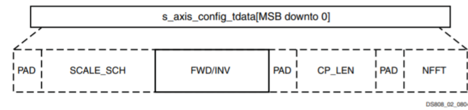


Figure 3-2: **Configuration Channel TDATA (s_axis_config_tdata) Format**

Optional fields are shown as dotted. Note that the bus width of `s_axis_config_tdata` might exceed the width necessary to accommodate all fields, including `SCALE_SCH`, even when `SCALE_SCH` field is padded to ensure the width is a multiple of 8 bits. The additional bits are unused, and therefore they will be optimized away during synthesis.

Offending part of the datasheet

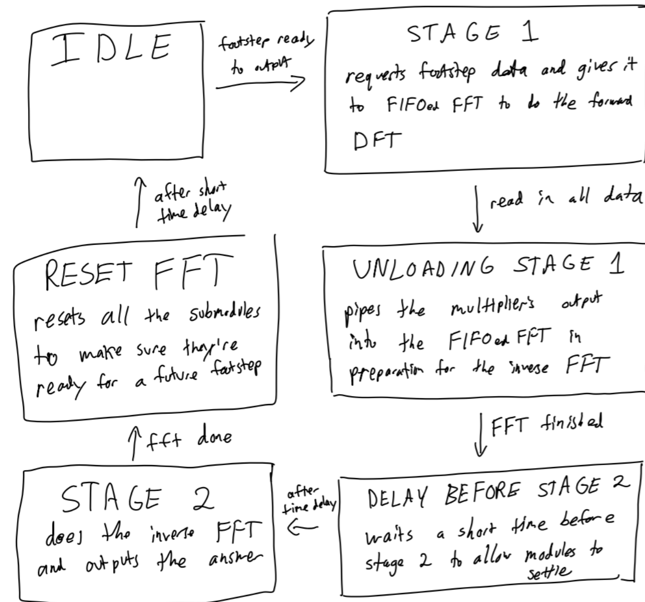
However, it turns out that if a field is optional and not configurable it actually just gets completely ignored, causing the forward/inverse bits to just simply be the 4 LSBs. This took a lot of trial and error to figure out and get working.

Also getting the scale of the transform was very difficult since the volume of a footstep, clap, snap, or stomp can vary wildly. Sometimes a peak is small, very tall but also thin, or very tall and wide. These variations can cause the magnitude of the FFT to go all over the place, often overflowing and giving invalid data. A lot of time was spent trying to find scale values that make sense using the Scale Schedule configuration option, but eventually we discovered the Block Floating Point scaling option that automatically scales the signal and reports the location of the decimal place at the end. For our purposes, the decimal point location doesn't matter since we only want the maximum so we were able to just ignore that part of the data.

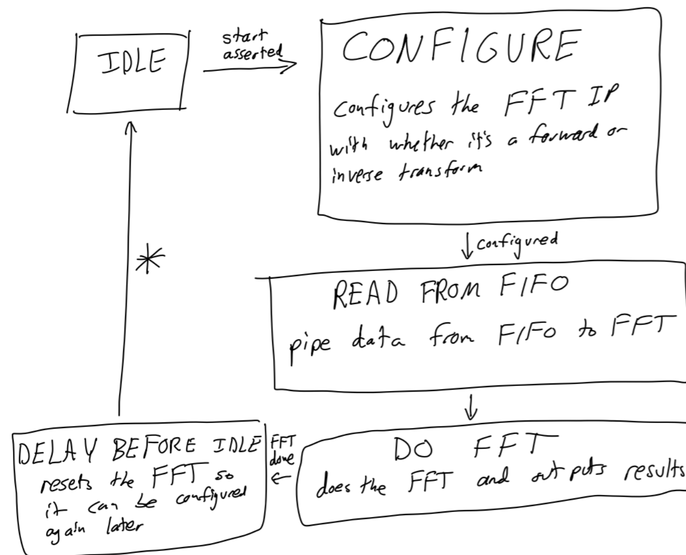
The way the FFT module is configured is also fairly annoying. The datasheet doesn't specify when you are allowed to configure the FFT. We had assumed that you could just configure between transforms but it turns out the entire FFT has to be reset before it can be reconfigured.

The pairwise multiplication module that does the cross-correlation math in the frequency domain is relatively simple. It takes in the frequency domain mic signals and puts them through a 3 stage pipeline to multiply the complex numbers. There were some timing violations when we tried fitting too much combinational logic in a single stage, but eventually it worked.

To make it simpler, we wrote it as a module that multiplies a pair of signals and then made 3 copies of that. This avoided copy/pasting code which is very error-prone.



Main State Machine

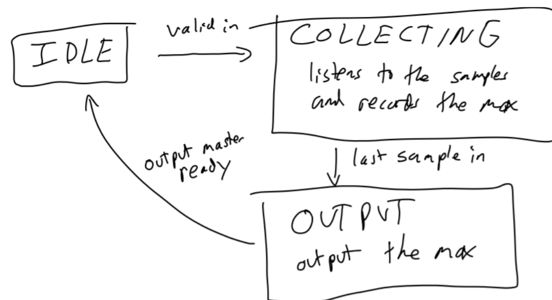


FFT Wrapper State Machine

Argmax - Joseph

This module takes in the cross-correlation and finds the time shift where it is maximized. This was written by making a module that takes the argmax of a single signal and making 3 of them.

This module was fairly straightforward except with the one issue that the math module wants signed time shifts while the cross correlation FFT comes in naturally indexed from 0 to 4095. Since our frames are a power of 2 long, we were able to use a signed time index and use the fact that it naturally overflows to get negative time shifts.



Single Signal Argmax State Machine

Location Wrapper - John

The location wrapper module performs all the math required to calculate the location of the source sound. The main theory for this sound localisation comes from a paper titled [“Acoustic-Emission Source Location in Two Dimensions” (Tobias, A. 1976)](). To calculate the source of the sound, the time difference of arrival between each mic needs to be converted to a distance difference using the speed of sound (this was performed in the the previous module). Then, those values are fed through a series of math steps:

Firstly, the radius from the designated microphone 0 is calculated using the following equation:

$$r = \frac{A_1}{2(x_1 \cos\theta + y_1 \sin\theta + \delta_1)} = \frac{A_2}{2(x_2 \cos\theta + y_2 \sin\theta + \delta_2)}$$

where

$$A_1 = x_1^2 + y_1^2 - \delta_1^2$$

and

$$A_2 = x_2^2 + y_2^2 - \delta_2^2$$

Where $(x_1, y_1), (x_2, y_2)$ is the location of mic 1 and 2 with mic 0 being at $(0, 0)$, δ_1 and δ_2 are the source sound distance differences for mic 1 and 2 respectively to mic 0, and θ is the angle from the x axis to the source sound.

θ is calculated as follows:

$$K = \frac{(A_2\delta_1 - A_1\delta_2)}{B}$$

$$B = [(A_1x_2 - A_2x_1)^2 + (A_1y_2 - A_2y_1)^2]^{½}$$

$$\beta = \cos^{-1} |\mathbf{K}|$$

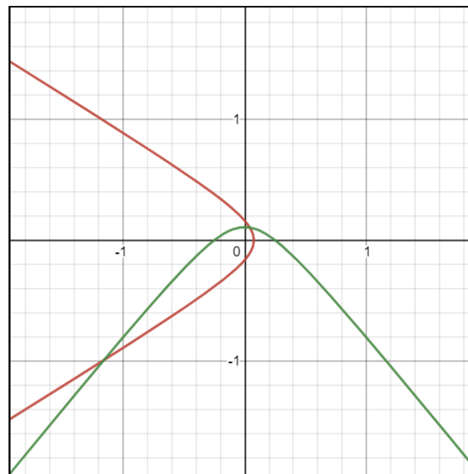
$$\tan \phi = \frac{(A_1y_2 - A_2y_1)}{(A_1x_2 - A_2x_1)}$$

$$\cos(\theta - \phi) = K$$

$$\theta = (\alpha + \beta) + 2m\pi \text{ or } (\alpha - \beta) + 2m\pi, \text{ for } m = 0, \pm 1 \pm 2$$

As θ can take one of two values, we also have two potential points for the source sound (x, y) .

Another way to think about this is that the Time Difference of Arrival defines a locus of points for each microphone, which is a hyperbola. These two potential points are the intersections of these hyperbolas. The math above is merely the algebraic method to solve for these points.



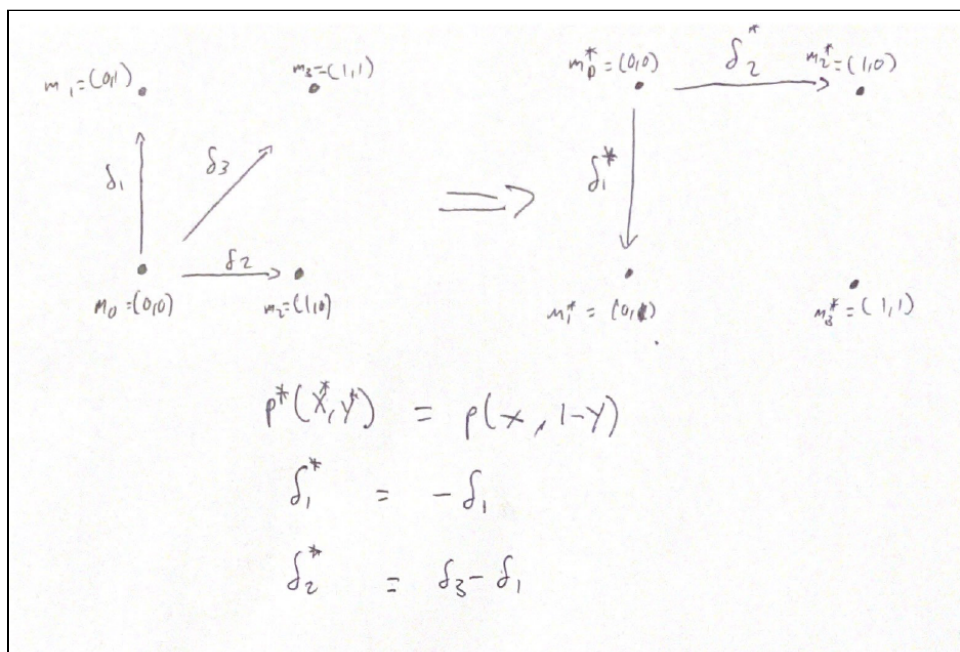
Two intersecting hyperbolas

This ambiguity can be resolved by using the time difference arrival between mic 3 and mic 0 in multiple different ways. The most obvious of which is to calculate the theoretical time difference of arrivals from each potential source to mic 3 and mic 0 and compare those values to the true time difference of arrival. However, this method uses more trigonometric functions and requires

another division module, which both uses more resources and introduces more possibility for error (additionally, given the problems we had with our microphones, it is likely that this result would have given us inaccurate results). Another possible method is to do the above calculation again, but change the coordinate system and choose a different subset of the microphones so that the time difference of arrival between mic 3 and mic 0 can be used. The second calculation would provide another set of 2 potential points, one of which being exactly the same as one of the points from the previous calculation. The points which overlap would therefore be the chosen predicted source location.

The point $(0, 0)$ is defined to be the location of mic 0. For simplicity and to reduce the burden of calculations, we decided to place the other microphones in a square pattern so that microphones 1, 2 and 3 can be located in coordinates $(0, 1)$, $(1, 0)$ and $(1, 1)$ respectively. The zeros allow for many operations to be ignored or simplified, reducing the complexity of the module.

So, the first calculation would use microphones 0, 1 and 2 in the coordinate system defined above. However, in order to perform the calculation again but with mic 3, we would need to redefine the coordinate system so that we have a new set of microphones at $(0, 0)$, $(0, 1)$ and $(1, 0)$. As well as transforming the coordinates, we would have to transform the Time Difference of Arrivals to make them all in relation to the new mic at $(0, 0)$. To maintain simplicity, the transformation was performed like so:



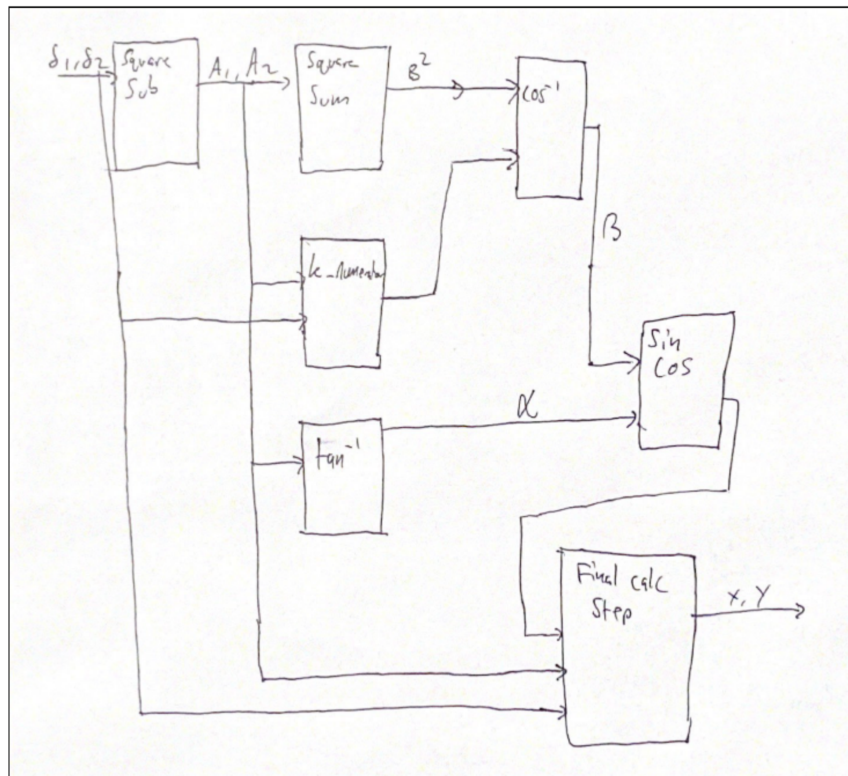
Coordinate transformation for ambiguity detection

In essence, mic 1 is converted to the origin and the Time Difference of Arrival is recalculated using the previous Time Difference of Arrival values. This allows the Location Predictor module to be reused for both calculations, with the inputs and outputs needing to be transformed as above for the second pass.

To keep track of which pass is currently being worked on, the module maintains a state of 3 possible states: ready, calculating first pass and calculating second pass. Only when the second pass is complete will the module assert valid and pass the predicted points to the next module.

Location Predictor - John

The Location Predictor module performs the actual math above as described by this block diagram:



Overall math Block Diagram

The rationale for only calculating the numerator of K will become clear below in the discussion of the arccos module.

This module also contains a state machine within it in order to calculate the two ambiguous points. This state machine is wrapped around the sincos and final calculation step as that is where the ambiguity appears. This state machine functions essentially the same as the state machine in the Location Wrapper function, having 3 states: ready, calculating first pass and calculating second pass.

Square Sub - John

This module calculates the Avalues in the math above and stores them to registers. Due to the arrangement of the microphone coordinates, this calculation simplifies to:

$$A = 1 - \quad ^2$$

To find alpha, the inverse tangent of the ratio of these two Avalues must be computed. However, the CORDIC IP, which is used to implement the inverse tangent, accepts as input the numerator and denominator, NOT their ratio. This proved incredibly useful, as it meant that we did not have to implement another division operation before passing these values onto the inverse tangent function.

B Square - John

This module takes the output of the previous module and calculates the square of B. Because of the way in which the inverse cosine was computed, it is actually unnecessary to take the square root to find the value of B. Again, the choice of microphone location simplifies this math to:

$$B^2 = A_1^{\ 2} + A_2^{\ 2}$$

K Numerator Calculator - John

This module uses both the input Time Difference of Arrival as well as the previously calculated A values. Again, due to the way the inverse cosine is computed, we only need to calculate the numerator, and do not need to divide by B.

Inverse Cosine - John

This was one of the more complex but fun modules to implement. The CORDIC IP does not contain an inverse cosine, instead, trigonometric identities would have to be used to convert this operation into an inverse tangent. The steps to do so were as follows:

$$\cos^{-1}(K) = \cos^{-1}(\frac{K_{num}}{B})$$

$$= \tan^{-1}\left(\frac{\sqrt{1 - \frac{K_{num}^2}{B^2}}}{\frac{K_{num}}{B}}\right)$$

$$= \tan^{-1}(\frac{\sqrt{B\text{^}2 - K_{num}^2}}{K_{num}})$$

So, this module takes in the numerator and B^2, takes their difference, then takes the square root and passes that into the inverse tangent CORDIC IP. However, There is a slight difference when it comes to the range of results, so, when the numerator is positive the answer must be subtracted by Pi.

Final Calculation Step - John

This final math module takes as input the $_1, A_1$ and the sine and cosine of θ to predict the final location. It does this first by using a divider module to calculate the radius, then multiplies this radius with sine and cosine θ to get the y and x values respectively.

One of the trickier parts of this module is matching the precision of the various input values, as they are all represented as different values.

Ambiguity solver - Joseph

This module takes in the 4 guesses produced by the math stage and sees which two are closest together. These correspond to where the two solutions agree and should then be the correct answer.

To find the closest two, we needed to find the minimum value within a list of all the distances. We only consider the distances between points from different math solves, so we have 4 pairs of values to compare.

We realized that for our short list it could be done very fast using the parallel nature of an FPGA. First we calculate the relative size of each difference and store them in the first pipeline stage. There are 6 pairs of distances, so we have 6 comparisons in this stage. In the second stage, we have all the comparisons precomputed so we can run a little tournament where we compare the first two and last two and then the winner of each of those rounds. This ends up being a function that maps 6 bits to a 2 bit index of the mic, which can be computed using only 2 LUT6es.

In theory that would be the end of it, but it turned out that the noisy location input to the math module meant that sometimes the answers just would not agree well. When the closest footsteps aren't within some specified distance bound, we had it output whichever guess is closest to the previous footstep. This relies on the idea that if a person is walking around then their feet are generally in the same area so the best guess would also be fairly close to the previous footstep.

Display - John

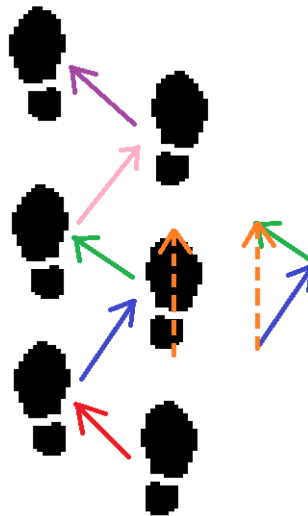
The display module is largely built on the module we built in lab 3, and reuses the XVGA module provided to us. The purpose of this module is to display the location of the previous 5 footsteps along with the location of the microphones and the FPGA. The major change to this module from lab 3 is that the `vcount` variable has been negated in order to have a coordinate system which increases in x and y as you move right and up respectively.

The microphones and FPGA are displayed using the blob module without any modifications. The FPGA is in the middle of the screen, so a small change to the coordinate system is that mic 0 is no longer at the point (0,0).

The locations of the previous 5 footsteps are held in an array of registers, with the oldest footstep getting kicked out as each new footstep is detected. This array of footsteps is passed on to the Footstep Direction Calculator and the Footsteps module to calculate the color of the pixel to be displayed at the current `vcount` and `hcount`. Earlier in the project, we also maintained an 'intensity' array, to give the feet a fade effect over time. However, this feature was removed.

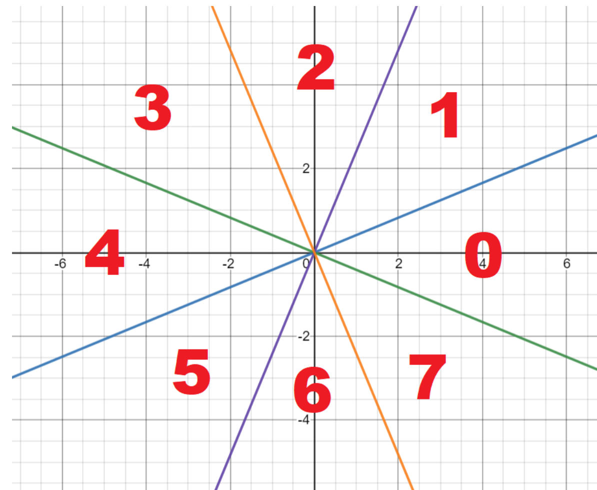
Footstep Direction Calculator - Joseph

We only get data about the location of a footstep, so it wasn't obvious how to get the footstep's direction from that. We took advantage of the way people walk. We can get the direction of a footstep by taking the average of the displacements from the previous and next footsteps:



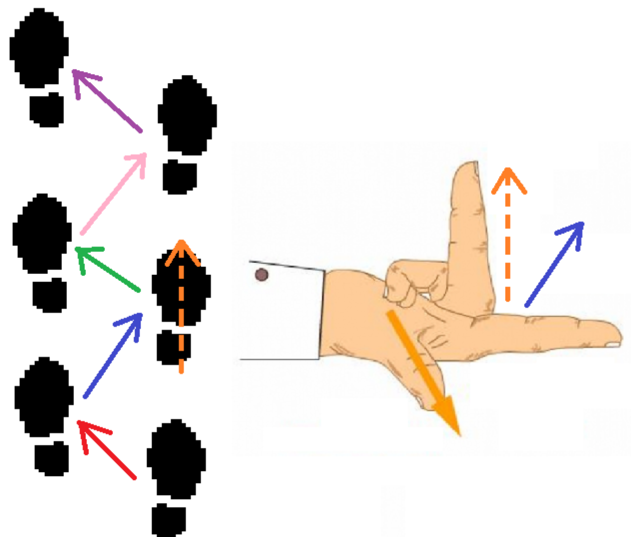
Averaging footstep displacements

Once we have the direction, we faced another problem of discretizing it. The first idea was to find the angle using trig and then check within thresholds, but that would have been a pain with the IP involved. Instead, we decided to bin it by comparing to the lines separating the different areas of the coordinate plane. Using fairly simple math, we can see if we are above or below each of the 4 lines. Each bin has a unique combination of being above and below lines, so we can find where the footstep is.



The numbering of the footstep direction bins

Then we faced the problem of how to tell a right foot from a left foot. We were able to find that by taking the cross product of a footstep's displacement with its direction. A right foot will have a positive cross product in the k hat direction and a left foot will have a negative one.



Determining footstep parity

We calculate the direction of the newest footstep and set the others to be alternating between right and left behind it.

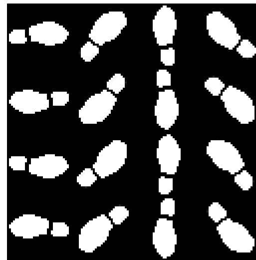
The module takes the past 5 footstep locations as input and outputs the 5 bin indexes for each foot sprite. The top bit says whether it is right or left and the bottom 3 bits say which of the 8 directions it is.

Footsteps - John

The footsteps module takes as input an array of the previous 5 footstep locations and their orientation and returns the pixel color for the current `vcount` and `hcount`. This module is fully parameterised, and can be adjusted to display a different number of footsteps. To do this, the module instantiates a picture blob for each footstep.

Picture Blob - John

The picture blob module uses the old picture blob module from Lab 3 as a template and is used for plotting the footsteps on the screen at the correct location with the correct orientation. The image used is a 32x32 black and white foot, reflected and rotated in 16 different orientations to represent 8 directions for each foot.



The 16 footstep possibilities

As this image is only black and white, we were able to do away with a colormap and compress the size of each pixel to just one pixel, for a total of 32*32*16 = 16384 bits. This allowed us to load all possible orientations of the foot onto the FPGA without worrying about size constraints. The COE file used was simply a combination of every individual COE. Having the COE in this format was extremely advantageous as we could split the bits of the ROM address into distinct parts with real meaning like so:

```
[13:0] address = {foot parity, foot index, column, row}
```

This was very useful, and made the integration of the ROM into the display easier than expected.

After the address is calculated it is simply looked up in the ROM and a pixel is either drawn white or nothing is drawn.

Challenges

Git, Version Control and Collaboration - John

We were both initially very adamant towards version controlling the entire project as we were aware of the incredible benefits that git provides. The initial move to git went smoothly and everything appeared to work for the beginning part of the project. However, as the project became larger and more complex, we started to run into various issues that would appear seemingly at random. These issues would range from IP suddenly throwing errors in simulation to bitstreams failing to write to the FPGA, but they would all be strangely fixed by simply deleting and recreating IP or files. These issues started to amount to a non-trivial amount of time spent on debugging and took valuable hours away from actual work on the project. So, it was eventually decided to move everything out of git and to simply have our own sides of the projects separate for the time being.

This, consequently, made collaboration a lot more difficult. Our system of version control ended up devolving to an email chain of modules being sent back and forth. This worked, but was highly inconvenient, especially when it came to the end of the project when many edits were being made across several different modules in rapid succession.

Signal processing - Joseph

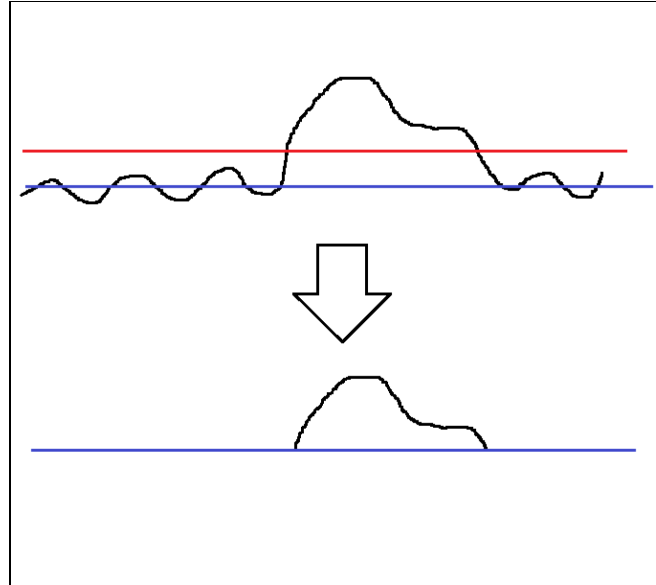
The signals end is the current main source of error in the project since the theory of sound localization isn't working well enough in our real-world system.

For the first pass, we just implemented the canonical solution: putting the signals through a cross-correlation and then finding the argmax to get the TDOA. This method should have worked, but the signal was not clean enough for it to work consistently.

The first clean-up job was to fix the mic's offset. Our mic data is centered around -2000 instead of 0, so we made an IIR integrator to find the average and then subtract it out to get a signal centered at 0.

Another issue is that the signs can vary - some footsteps are heard as a big positive spike and others are heard as big negative spikes, so we took the absolute value of all the signals.

When trying the system in different rooms, we found background noise could cause a lot of problems and each room would have its own noise. Instead of messing around with spending lots of time characterizing the noise and bandpassing, we deleted everything with an absolute value below the threshold and then shifted the signal to have its zero at the threshold value. This solution was somewhat hacky but seriously improved the signal quality since periodic noise was often overpowering our signal since cross-correlations love long sine waves that they can match up.



Zeroing out the noise. Red is the threshold and blue is 0

There was also an interesting case where our mics were not ideal and would just miss some footsteps. This led to displaying some footsteps with clearly bad values like a time difference of exactly 0 (it’s possible but *very* unlikely), or time differences that indicate distances more than the longest distance between mics. We were able to get rid of a lot of invalid footsteps by throwing those out.

We also went through a few iterations with the direction of the microphones. They’re advertised as omnidirectional mics, but at least according to Adafruit’s instructions, it seems like they work much better in the direction of their hole. We tried putting them on stilts to point them at the ground where we may be able to get a better signal. That gave better results, but it was unclear if the sound quality was better near the floor or if the breadboard was just absorbing ambient noise in the room (or neither!). Eventually we remembered that the speed of sound is different through different materials, so putting the mics on stilts in the air could mean that if sound wants to get to the microphone, it has to go through the air where we know its speed. Joe also suggested that our solid-core wire could also be transmitting sound, but we didn’t have time to test that possibility extensively.

These previous problems were solvable by various means, but our approach was fundamentally flawed when it came to two other issues: echo and distortion.

In all the rooms we tried, we had significant amounts of echo. This would cause one peak to turn into multiple peaks spread out in time. The cross-correlation would try to match up all the peaks at once and give a wrong answer. We could have fixed this in a few ways, such as detecting which peak had the largest magnitude in the time domain and deleting the others, but that would require another module that does all that processing. We didn’t have the time left to try that and we had no guarantees that would actually work (in hindsight I don’t really think it would have solved it). We could have also multiplied with the inverse of the frequency response

of the room, but that changes at each point in the room and would be a mess to calculate. We could have also multiplied with the inverse of the frequency response of a footstep to turn them into deltas which would be much easier to filter, but characterizing footsteps to that precision would have been too complicated.

There was also another big issue where a couple mics would hear a peak clearly but then another would get a distorted version that just looks like higher amplitude noise. The cross-correlation relies on having a clear peak that looks similar across all mics, so when the footstep devolves into fuzz we just can't do anything intelligent with that.

In a last-ditch effort to get the system working reliably, we thought about the structure of a footstep. When we recorded footsteps on a phone they had a beautiful peak, but on these mics they just aren't as clean. On our mics, footsteps have a bit of fuzz before their peak, meaning that a microphone is being affected by the footstep before it gets the important part of the signal. In theory, if information is only being conveyed by sound, information about the existence of a footstep should only be able to get to the microphone at the speed of sound. So, we wanted to implement a filter that looks for a rising edge in signal energy, corresponding to the first time the microphone is affected by the footstep.

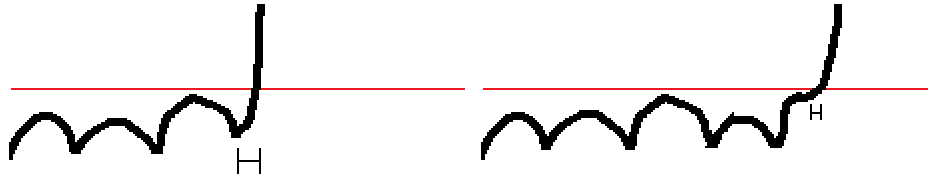
We did this by taking the absolute value of the signal to avoid sign issues and then waiting for it to pass a threshold set just barely above the amplitude of the ambient noise. Once it goes above that threshold, it means that the microphone is being affected by the footstep and we can zero out every sample other than the first one above the threshold. This creates a signal that's all zero except for a delta at the moment the microphone even smelled a footstep. This approach gave us much better time precision and solved the echo and distortion problems at the cost of being extremely sensitive.

This solved the echo problem because echoes always happen after the initial fuzz, so if everything afterwards gets deleted so do the echoes.

This solved the distortion problem because we don't care that it doesn't have a peak in it - it still has the rise in signal energy.

There were, of course, still other issues to deal with. Zeroing out all the other samples means that if only one mic has heard something above ambient noise then it goes into a mode of outputting zeroes and can't hear the next footstep. This was solvable with a counter that periodically turns the mics back on when a footstep hasn't been detected in a while.

This approach also has issues in that it depends on the phase of the noise. If the noise is at the bottom of its period, then the signal could take a few samples to ramp up to the threshold. If the noise is at the top of its period, then the signal will go above the threshold immediately, as shown in these figures:



Noise being in different phases causing time delays

This time difference seems to max out at around 5 samples, which is about 14 inches off. This could potentially be solved by thresholding the derivative of the signal instead, but we don't have enough time left to make the module for that.

There was also a big issue with the nature of the interaction of the math module with noise. When given accurate TDOAs, the math module can give out the exact answer. When confronted with noisy measurements, the hyperbolas are in the wrong places and the ambiguity detector just gets 4 points spread around, without any particularly close to one another. We put a bandaid on this by saying that if the two solves didn't agree enough then we would just choose the footstep closest to the old one, but that is not as robust of a solution as properly getting a less noisy TDOA.

At this point, when we have wrong answers they often have structure to them where they are on the other side of the screen, corresponding to the other solution of the two hyperbola intersections. We were able to further improve results by only considering footsteps in the bottom part of the screen and only walking there.

The signals end is currently very imperfect, but it's much better than before.

Math - John

One of the biggest challenges for the math modules in the FPGA was working with fractional and signed values. The way these values are represented is by using two sets of bits, one to represent the integer and signed portion and the other to represent the fractional portion. Regular, integer values in SystemVerilog are instatianted in the form of `logic [n-1:0]`, however, it is possible to instantiate values using the following syntax: `logic [m-1:-k]`. I used this method to keep track of how many fractional bits a value had, allowing me to better calculate how the precision changed over time and what operations I needed to perform to maintain proper bit width.

This became an issue when I, incorrectly, thought that the maximum Distance Difference was restricted to (-1,1) as the distance from the two adjacent microphones to Mic 0 is 1 meter. So I decided to use a 16-bit, fixed-point, signed number in the form of `logic [0:-15]`, which satisfies these bounds exactly. While these bounds are correct for the Location Calculator module, which only looks at 3 of the 4 points, it is not true for the input to the Location Wrapper module, which needs the Distance Difference between Mic 0 and Mic 3, which could have a

magnitude up to the square root of 2. Thankfully, after the change of coordinate computation I was able to get the new Distance Differences within the original bounds, so I didn't have to make many changes to the Location Predictor module, but had to ensure that the Location Wrapper was passing through the correctly sized value.

Additionally, when using test benches to evaluate the math modules, the simulation would perform as expected. The process for performing these tests was as follows: pick an arbitrary point from which the sound would originate from, use Python to calculate the Time Difference of Arrival for each mic, pass those values into Vivado and compare the output with the initial point. As Python calculated the Time Difference of Arrival, it was accurate for every microphone, and my module was able to flawlessly predict where the sound originated from. However, in real life, the Time Difference of Arrival was not accurate for every mic for many reasons. Most of these reasons are due to the signal problems described above, but another, non-signal-related reason is the existence of the 3rd dimension. The algorithm used by the math module assumes that the sound originated from within the plane of the microphones, but this is a false assumption. As the source of the sound extends below or above the plane of the microphones, the Time Difference of Arrival between two microphones shrinks. This causes two problems. Firstly, when performing the first pass of the Location Predictor module, the result can simply be inaccurate. The degree of error is acceptable for small heights but can become unusable for some heights. Secondly, when comparing the first to the second pass of the Location Predictor, if there is a significant z-component to the source, then there will be discrepancies between the predictions. Couple this with the previous problems with the signals and it becomes incredibly difficult to generate an accurate prediction if the sound is not created on the plane of the microphones.

Display - John

The display was, thankfully, one of the easier modules to implement. The biggest challenge was the change of coordinate system from meters to pixels and flipping the y axis to go from bottom to top, which caused small bugs such as the footsteps being displayed backwards. However, even these issues paled in comparison to those previously discussed. We also anticipated a lot of issues with displaying the different directions of the foot, as there were 16 possible orientations, but none arrived.

One of the main reasons for the lack of problems with the display was that we had available to us the code from the previously completed Lab 3, which provided a solid framework from which we could build and adapt.

Advice, Future Work and General Reflection

Advice

One of the most useful tools for debugging was the VIO IP. This IP allowed us to set a chosen register to any value from the Hardware Manager in Vivado (and also technically was able to read values like an ILA, but that functionality was simply inferior to the ILA IP). This opened up the possibility for much faster testing of modules and integration. The VIO IP was also used for online calibration (in the footstep detector module), reducing the time spent compiling fresh bitstreams.

The availability of this feature significantly reduced the development time of our project and allowed us to have more accurate calibration of the microphones. We highly recommend to anyone approaching any FPGA project to make extensive use of this IP for testing on the board.

The onboard switches and leds are also very useful. The leds can display important numbers in binary, such as a state or key value that you want to look at continuously instead of using a trigger on an ILA.

Also AXI is your friend - using it for your modules means you don't have to think about an interface and accidentally make a buggy one. You can also hook it up to anything fairly directly, such as a FIFO if you ever surprisingly need one in between modules. It also makes collaboration much easier since you know what to expect from somebody else's modules.

When trying to solve signals problems in this project, we kept having much better success with very heavy-handed hacky methods than nuanced ones. On the time scale of this project and the slow pace of FPGA development, there isn't very much time to come up with generalized, pretty signal processing. Sometimes just setting a bunch of stuff to zero can get pretty good results and save time.

Future Work

As the main source of error is the Time Difference of Arrival calculation, this would be the most fruitful avenue to pursue for more accurate footstep localization. If we had more time to iterate on the signals side, we would have liked to use better microphones, write a more intelligent filter for finding the rising edge of signal energy (perhaps looking at the rising edge of its derivative), and use better filters to remove noise (perhaps a Wiener filter). Our wires are also potentially causing issues since copper carries sound much faster than air so we could potentially clean up our data by using braided instead of solid core wire. Having more microphones and averaging answers from different sets could also potentially improve signal quality.

The display is another area which could be improved in future work. While the focus of the project was the signals and math, having an aesthetically pleasing display which mimics Harry Potter's Marauder's Map more closely would be a nice addition to the project.

Reflection - Joseph

I seriously enjoyed working on this project and feel much more mature in my FPGA skills. At the beginning, I was somewhat intimidated by all the steps and the never ending proofs of Murphy's Law, but by the end writing code much more confidently. I feel fairly proud of the size of the codebase and how many moving parts had to come together.

For a long time I've wanted to try working with firmware stuff so I really got a lot out of interfacing directly with the i2s microphone.

This was also my first open-ended signal processing project - I've taken 6.003 and 6.011 and I was very excited to finally apply what I learned in those courses to a real problem. It was very cool for me to actually reason about signals in an engineering setting when I've only really seen the theory so far.

This project also gave me a much deeper appreciation for what a computer actually does. When integer precision, IP for more complicated functions, and timing constraints start playing a role, the fact that a computer is a physical thing on a chip doing math actually sinks in.

Reflection - John

Overall, I thoroughly enjoyed this project and found it very rewarding. There was a lot of time sunk into various issues; but, with each bug ironed out I felt that I was becoming a better FPGA Engineer, and would not classify any of that time as "lost" or "wasted" (besides the git stuff that was extremely weird). While the accuracy of the final product was not quite what we were looking for, the fact that we were able to track some footsteps was incredibly satisfying. It's one thing to read swaths of papers and wikipedia on the theory, but to see it working from microphone to screen did feel like 'magic'.

Another aspect I enjoyed about this project was coming up with clever ways to perform fixed point math. Due to limitations in the IP and binary math, it was the case that either not all operations were available to me or I would not want to perform some operations. Being able to scribble some equations on a piece of paper for a few hours to really think about how to move the math onto the FPGA was very fun and intellectually stimulating. This class has inspired me to pursue a career in FPGAs and I am looking forward to being able to apply the skills learnt in this project to both future academic and professional work.