Acoustic B-FSK Link Design

Summer 2022 Final Report

Prepared by: Mohammedali Khalaf

Table of Contents

Table of Contents

Table of Figures

1. Project Objectives

The main objective of this project was to model and program an Acoustic BFSK transmitter system with various improvements on an FPGA and test it underwater to analyze the results. In doing so, data can be provided to analyze the effects of different improvements to the system, such as turbo coding and frequency hopping. The project can generally be split into two main components, modelling in python, and implementation in VHDL.

The purpose of modelling the system is to analyze the behaviour of different components before implementing the system to gain a thorough understanding of the system architecture. Furthermore, with the use of the matplotlib package, it is also possible to examine the signal and its various stages under time domain and frequency domain plots. Lastly, the model allows for the analysis of bit error rates under various conditions which helps predict the quality of the system.

After modelling the system, implementation of the transmitter chain was to be done on an FPGA board in VHDL. The use of an FPGA allows for easy switching between different versions of the transmitter chain, allowing for the collection of more data. To take advantage of this characteristic of FPGA's, three different transmitter chains were to be designed, each having their own combinations of different improvements. By creating three different transmitters, it is possible to analyze and compare the effects of the different improvements on the bit error rate.

2. Modelling Methodology

The first step in developing an underwater communication system is creating a model to observe how the system will behave. In the case of this project, modelling involved a theoretical component, and a software component. Generally, the software component could be considered the end goal of the modelling portion of the project, however, to develop a proper script to simulate the communication system, the communication theory of what is to be implemented must be understood first.

The scope of communication theory in terms of this project encompasses basic theory on what BFSK communication means, and why improvements such as turbo coding and frequency hopping are useful, as well as how they work. In this chapter, the first topic of discussion regards the necessary communication theory. The second topic of discussion will be the parameters that the system is confined to with regards to the topics discussed in the communication theory section. After the first two topics of discussion, there will be an adequate understanding of what is to be implemented in python, allowing for the third topic of discussion, the python model script.

2.1. Communication Theory

The physical layer of the transmitter is set to be compliant with the JANUS protocol. The JANUS protocol is a description of the physical layer of the communication link. The main notable features of the JANUS protocol are BFSK modulation, 13 frequency hopping tones, and turbo coding. To understand the model to be implemented, the concepts of BFSK communication, frequency hopping, and turbo coding are to be understood.

2.1.1. BFSK Communication

BFSK stands for “Binary Frequency Shift Keying”. The main premise of BFSK is that one selected frequency represents a binary 0 and another represents a binary 1. This is the simplest form of frequency shift keying. The figure below illustrates a BFSK signal in the time domain.

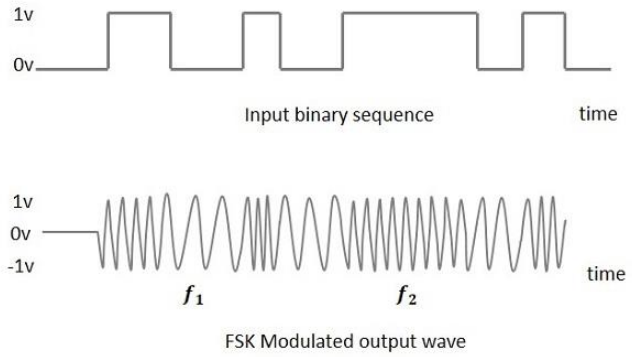

Figure 1: BFSK Communication Time Domain Representation (TutorialsPoint)

2.1.2. Frequency Hopping

Frequency hopping is a method used to counter the effects of multipath. Multipath is when a signal sent from a source arrives at its destination at different times due to interactions with the boundaries of the media set the signal is sent in. The image below is a visual representation of multipath.

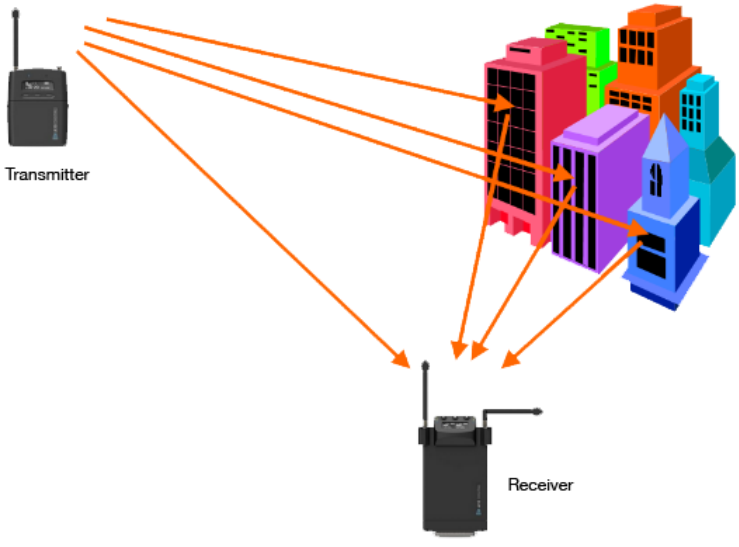

Figure 2: Visual Representation of Multipath (Sound Devices)

The image above shows the transmitted signal arriving at the receiver at four different times due to the extra signals reflecting off other surfaces. This is a problem for BFSK systems because if a binary 0

frequency is sent, and then a binary 1 frequency is sent, then what happens is that extra copies of the binary 0 frequency signal can arrive when the system intends to communicate a binary 1. Essentially, multipath can corrupt the signal. Frequency hopping can counter this by defining a finite number of sub carriers to perform BFSK modulation on.

The way that frequency hopping works is by switching through a finite number of subcarriers sequentially and performing BFSK modulation on the subcarriers. For example, consider the following parameters

- Bandwidth between binary 0 and binary 1 of 10kHz
- Two frequency hopping subcarriers 10kHz and 30kHz

Based on this system, the carrier will switch between 10kHz and 30kHz for each symbol. Furthermore, based on the bandwidth between the 0 and 1 frequencies, if the carrier is at 10kHz then 5kHz represents false, and 15 kHz represents true, and if the carrier is at 30kHZ, then 25 kHz represents false, and 35kHz represents true. By switching from subcarrier to subcarrier, multipath can be negated as the receiver will be looking within a band centred on a different subcarrier than the past. For example, considering the same frequency hopping parameters described in the last example, if we send a bit 0 over the 10 kHz subcarrier (5kHz), and a bit 1 over the 30kHz subcarrier (35kHZ), then even if a copy of the 0 were to arrive with the one, it will not matter as the receiver would be looking between 25kHz and 35kHz, essentially ignoring the incoming 5 kHz signal. In the case of the JANUS protocol, 13 frequency hopping subcarriers are used.

2.1.3. Turbo Coding

Turbo coding is an error correction algorithm that uses convolutional coding to add redundancy to the code. Essentially, if you feed a turbo encoder binary data at the rate of F_s then it will output encoded bits at the rate of $3F_s$. Furthermore, a single bit at the input must be read through 3 consecutive bits at the output. This way, when sending a bit into the encoder, it will be output 3 times in an encoded manner so that if one of the 3 bits are wrong, the other two will correct the interpretation. It should be noted that in the case of this project, the turbo coding uses two encoded bits instead of three.

2.2. System Parameters

Before discussing the python model, it is important to first be familiar with the parameters that will dictate the nature of the python model and VHDL implementation. Firstly, the acoustic source only operates at a carrier frequency of 27.5kHz and a bandwidth of 5kHz. Essentially, the system is confined to producing signals with frequencies between 25kHz and 30 kHz. For the base level transmitter, and for the python model, the two binary baseband frequencies will be 2.5kHz for binary 1 and -2.5kHz for binary 0. The baseband signal will then be upconverted by 27.5kHz, meaning that at passband, 25kHz will represent binary 0 and 30kHz will represent binary 1.

In terms of the upgraded transmitter chains, considering that frequency hopping will be featured, the frequency parameters will look different. In terms of the baseband stage of the signal, rather than representing binary 1 and 0 with ±2.5kHz, the two binary symbols will be represented with ±96Hz. This will allow for the frequency switching of BFSK to use less bandwidth. After that, when implementing the frequency hopping tones, the 13 frequency hopping tones will be spread symmetrically between ±2304Hz. Consequently, the bandwidth between each tone will be 384Hz. After considering frequency hopping, and the baseband BFSK frequencies, before upconversion to pass band, the system will output

frequencies between 2400Hz and -2400Hz. After upconversion, the signals will be between 29.9kHz and 25.1kHz.

Other than frequency parameters, the only other parameters that should be considered are timing parameters. For the first of the three upgraded configurations, frequency hopping will be the only implemented upgrade. In the case of this configuration, each bit will last 20 milliseconds. For the case of the second upgraded configuration, spread spectrum will be featured. Since each bit will be spread across the 13 subcarriers one at a time, the symbol period will be 13 times longer. Consequently, the symbol period will be 260 milliseconds. The last configuration to be considered will feature turbo coding and frequency hopping. The turbo coding that will be implemented will have the rate at which bits are communicated. As a result, the third symbol period will be 40 milliseconds. In the case of all three configurations, there will be a total of 1024 bits per frame, each spanning a timeframe of 1024 times the symbol period.

2.3. Python Model

To understand what is to be implemented in VHDL, the desired system had to be modelled in python first. The desired outcomes of the model are as follows

- Create a pseudo-random (PN) bit sequence
- Create and plot a baseband FSK signal that communicates the PN sequence
- Upconvert the baseband FSK signal and observe the plotted results
- Create and add noise to the FSK signal
- Implement matched filters to analyze the bit error rates of the signal as a function of signal to noise ratio

Since all the other outcomes are necessary steps to fulfill the last one, it was possible to achieve all of these by simply aiming towards the last outcome and creating plots along the way that would allow for the analysis of the other preceding outcomes.

Before the python script was written, a data flowchart was necessary to set a proper path towards the end goal. The figure below displays the data flowchart used to describe the python script used.

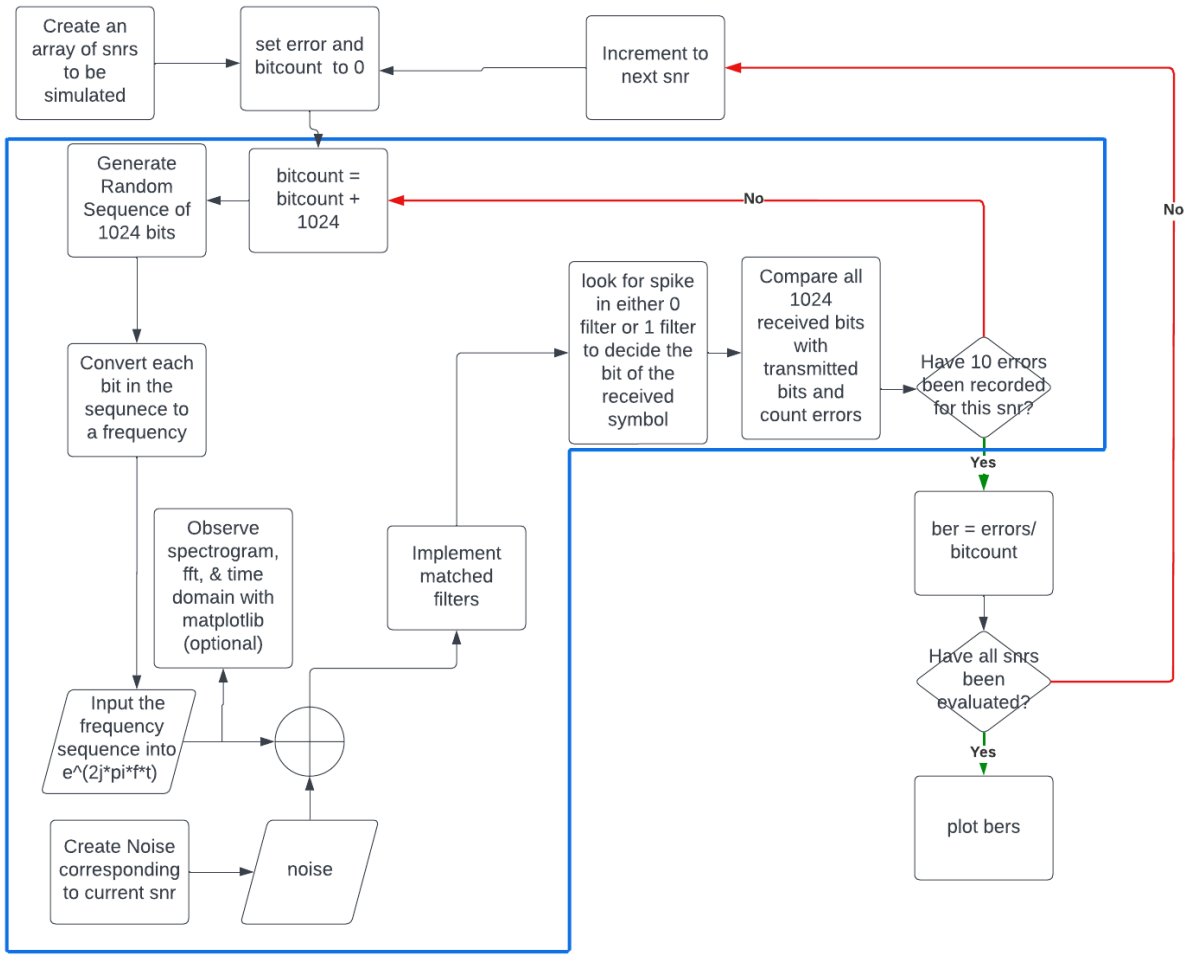

Figure 3: Data Flowchart of the Python Model

The script described in the flowchart goes as follows,

1. Define a finite number of signals to noise ratios to be evaluated
2. Create an error counter, and another counter that tracks the number of bits transmitted for a given signal to noise ratio and initialize them to 0.
3. Process a frame of data which in the case of the implemented script is 1024 bits and add 1024 to the transmitted bit counter. Within the processed frame, the noise added to the signal is a function of the current signal to noise ratio being evaluated.
4. Count the number of errors and add them to your error counter.
5. If the error counter has reached 10, move on to step 6, otherwise, repeat steps 3 and 4
6. Calculate the bit error rate for the current signal to noise ratio with the formula: $BER = (error\ count)/(transmitted\ bit\ count)$
7. Repeat steps 2-6 (everything in the blue box) for the next defined signal to noise ratio until all defined signal to noise ratios have been evaluated
8. Take all calculated bit error rates and plot them on a logarithmic scale to be viewed

9. (OPTIONAL) During step 3, create FFT plots, spectrograms, and time domain representations of the generated data to be viewed and analyzed

This model allowed for the creation of various plots in the process of creating a bit error rate plot, such as spectrograms and time-domain representations for both the baseband and passband signals. In the next section, the various plots that were extracted from this model will be discussed. It should be noted that this python script does not include frequency hopping, turbo coding, or the effects of multipath considered.

3. Modelling Results

Through modelling the system in python, it was made possible to see how the signal will behave through different parts of the transmission process. Generally, the main stages that are desirable to look at are at baseband, passband, and the bit error rate at the end of the receiver. These are the main parts to be discussed in this chapter.

3.1. Baseband Modelling Results

It should be noted that it is relatively difficult to observe a baseband FSK signal in the time-domain and be able to point out where the frequency changes, therefore, it is most effective to make use of a spectrogram to be able to observe how the frequency changes over time. The figure below is a spectrogram of the BFSK signal communicating 1024 pseudo-random bits.

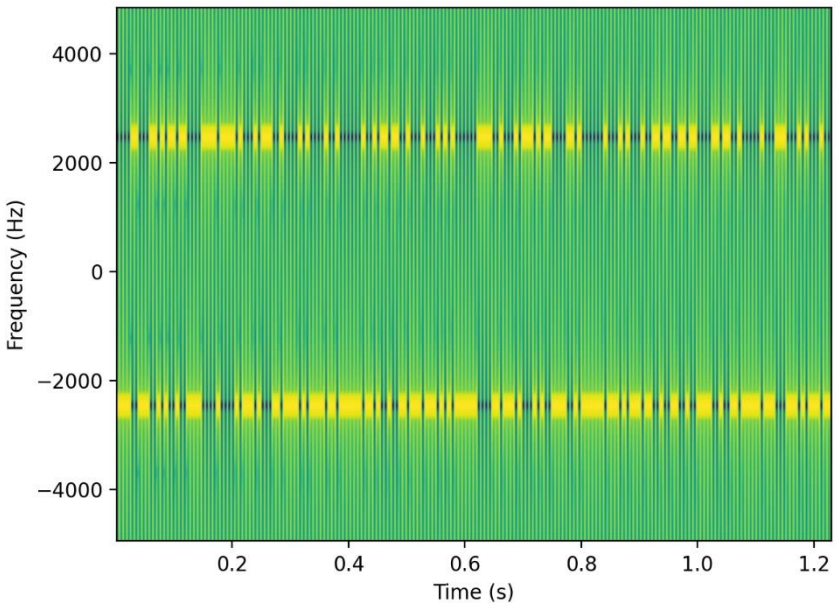

Figure 4: Baseband Spectrogram from the Python Model

As can be seen in the spectrogram, when the python model is programmed to behave the same as the transmitter that was to be built, successful switching between frequencies is achieved. Furthermore, the spectrogram demonstrates that the proper baseband frequencies of ±2.5kHz are also achieved. The

results of the baseband component of this model confirm that in the baseband portion of the transmitter, everything is properly functioning.

3.2. Passband Modelling Results

In terms of the plotting the passband component of the model, it should be noted that similar to how it would be difficult to observe frequency changes between positive and negative 2500Hz, the difference between 25 kHz and 30kHz is likewise. For that reason, a spectrogram is once again in order. The figure below illustrates the spectrogram of the FSK signal in the python model after upconversion.

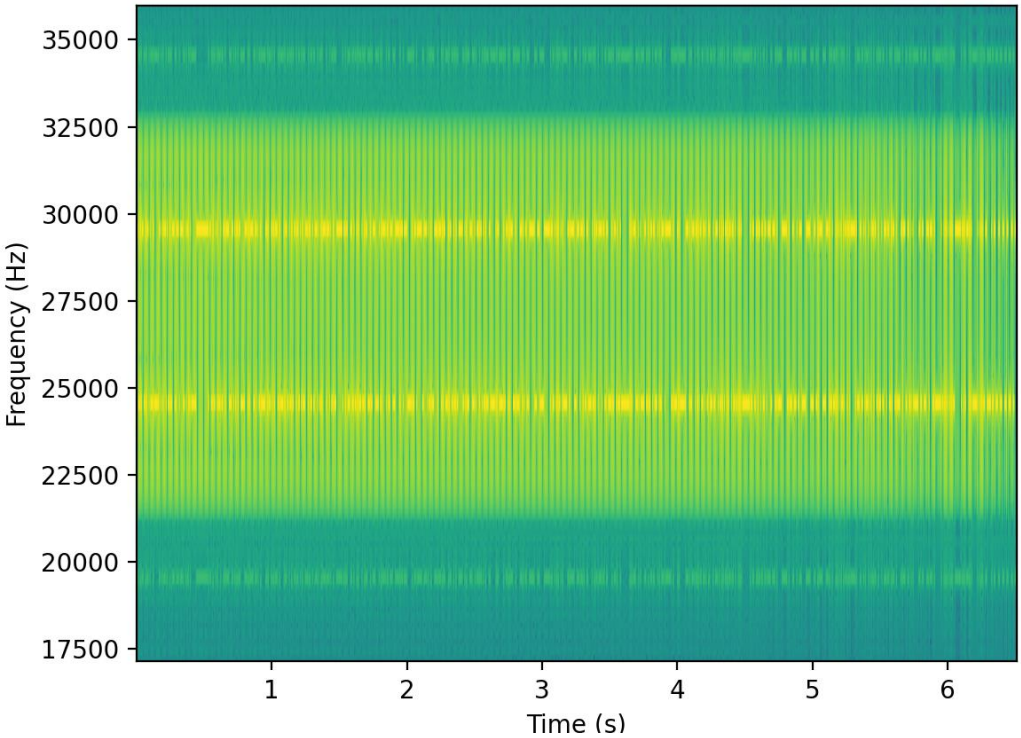

Figure 5: Passband Spectrogram from the Python Model

As can be seen in the figure above, the same signal is successfully upconverted to a carrier of 27 kHz. This confirms that the upconversion component of the transmitter circuit will work as intended.

3.3. Bit Error Rate Results

The final measure that was to be observed with the python model was the bit error rate as a function of signal to noise ratio. The figure below shows bit error rate plotted against signal to noise ratio on a logarithmic scale.

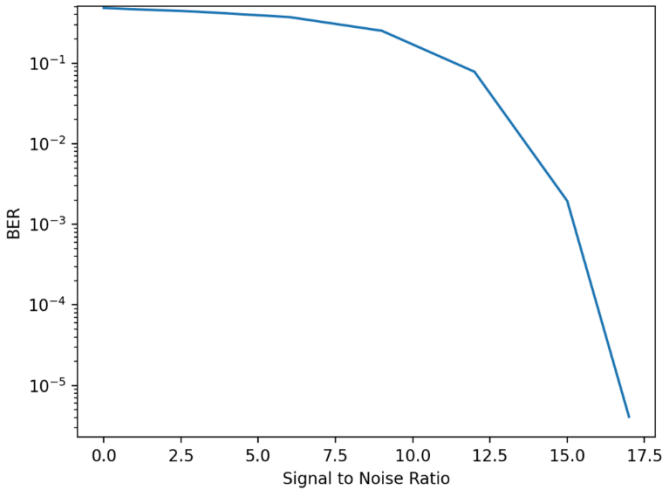

Figure 6: Bit Error Rates of a Basic BFSK Transmitter from the Python Model

From the bit error rate curve, the chances for error steeply drop at a signal to noise ratio of 12. It should be noted that this python model assumes no multipath. Also, due to extremely low bit error rates beyond a signal to noise ratio of 17, it is extremely difficult to plot further than 17 as processing times can take hours. Given that multipath is not considered in this model, and that it is expected to be very prevalent during testing, it is easy for one to assume that creating a bit error rate plot is trivial, however this is a false assumption. By creating this bit error rate plot and observing the expected results which would be a waterfall curve and extremely low error rates, it can be confirmed that the transmitter system will work as intended. From this, it is easy to move on to the VHDL implementation knowing that the system to be implemented will work.

4. Implementation Methodology

The VHDL implementation could be split into four sections. The first section is the implementation of a base level chain without any improvements. Creating this was necessary to understand how the basics of the later upgraded versions are crafted within the FPGA. The other three sections are the individual upgraded configurations. The first topic of discussion in this chapter will be the first section. Afterwards, the three sections that cover the upgraded versions will be discussed in the later parts of this chapter.

4.1. Base Level Implementation

When designing the base level transmitter, it is best organized when different functions are split into different components. The data within the transmitter follows a rather simple path through the transmitter chain. To properly understand the path, each component of the transmitter chain should be discussed independently.

4.1.1. PN Generator

At the start of the transmitter chain is a PN generator. The PN generator generates a pseudo-random stream of bits. The rate of bit generation is the same clock frequency it is fed. There are multiple different types of PN generators to generate different types of random data, however, the one in specific that was used for this project is called a Gold Sequence, or Gold Code. The Gold code is a pseudo random sequence that can be used by those who want to have multiple callers on the same band is the data it generates in different streams are statistically orthogonal. A multiple input/multiple output system (MIMO) was not within the scope of this project, so this multiple caller feature was not yet use. The reason then, that the Gold Sequence was selected, was so that if a MIMO system were to be considered in future works, the framework for the random bit generation is already set in place. In summary, the purpose of the PN generator in this project is to generate a random sequence of bits that can be sent through the transmitter, and that can be generated on another laptop so that the stream of bits sent is known, allowing for post processing. The random bits that are output from the PN Sequence are fed to a bit to phase converter.

4.1.2. DDS and Bit-to-Phase Conversion

The next component of the transmitter chain that should be discussed is the DDS. The DDS (Digital Direct Synthesizer) is an IP Core within Vivado that is used for generating complex signals. How the DDS works is that it is fed a clock and a phase (Θ). With each clock cycle, it will increment through it's internal LUT that is connected to the output. As the DDS cycles through it's internal LUT, the output will take the form of a sinusoidal wave. The number of bits at the output that are used to represent the complex waveform can be selected by the user. The speed at which the DDS cycles through its LUT is controlled by the phase. As a result, the output frequency is a function of, the clock and the phase. The exact formula to calculate the output frequency is:

$$f = \frac{fclk * \theta}{2\text{^}B(\theta)}$$

Where "fclk" is the input clock frequency, Θ is the input phase, and "B(Θ)" is the input phase bit width. It should be noted that if a 16-bit output is requested, the IP core will output 32 bits. The reason is that the first 16 bits represent a cosine wave of the generated frequency, and the second half of the bits represents a sine version. Both are necessary for later upconversion so both will be use.

By using the DDS, the desired baseband FSK signal can be generated. In terms of the base level transmitter chain, the two frequencies that are desired are positive and negative 2.5kHz. The switching between these two frequencies can be achieved by switching between the two different phases that will output these desired frequencies. To have the DDS output different frequencies as a function of the random bits that are output from the PN generator, bit-to-phase conversion must be done first. Essentially, between the DDS block, and the PN generator block, the bit-to-phase conversion block was placed between. This block functions as a simple multiplexer that outputs 1 phase for a binary 1, and the negative version of that phase for a binary 0. Generally, this is easily achieved with a simple "case" statement in VHDL.

To summarize the functions of the components in this section, the random bits from the PN generator are sent through a bit to phase converter that outputs a phase corresponding to the binary value it is fed. The phase from the converter is then sent to the DDS to control the output digital signals

frequency. In the case of this projects base level transmitter, the DDS is fed a 100MHz clock and outputs a 2500Hz digital complex waveform.

4.1.3. Upconversion

To be able to send the signal into the real world, it must be upconverted from passband to baseband. Generally, the objective of this component is to increase the frequency by 27.5kHz. In other words, the 2500Hz will be come 30kHz and the -2500Hz becomes 25 kHz.

To upconvert to the proper passband frequencies, first, a carrier signal must be generated. This is done with another DDS that is generated in fixed frequency mode. Fixed frequency mode is an option available when generating the IP core in Vivado which allows the user to tell the compiler that the frequency is not intended to oscillate. The consequence of this mode is that there will no longer be phase input port, only an input clock port, and the user will have to type into the compiler what exact frequency they want. In the case of the carrier, a frequency of 27.5kHz is desired so that is how it was configured. Like the previous DDS, the first half of the output bits represent cosine, while the second half represent sine.

To perform upconversion using the baseband waveform, and the passband waveform, the following is done,

1. Take the first half bits of the 2 waveforms and multiply them together. This is the even or cosine stream.
2. Take the second half bits of the 2 waveforms and multiply them together. This is the odd or sine stream.
3. Take the two streams and add them together.

After completing this process, a passband signal is achieved. The figure below shows a Vivado simulation of this process.

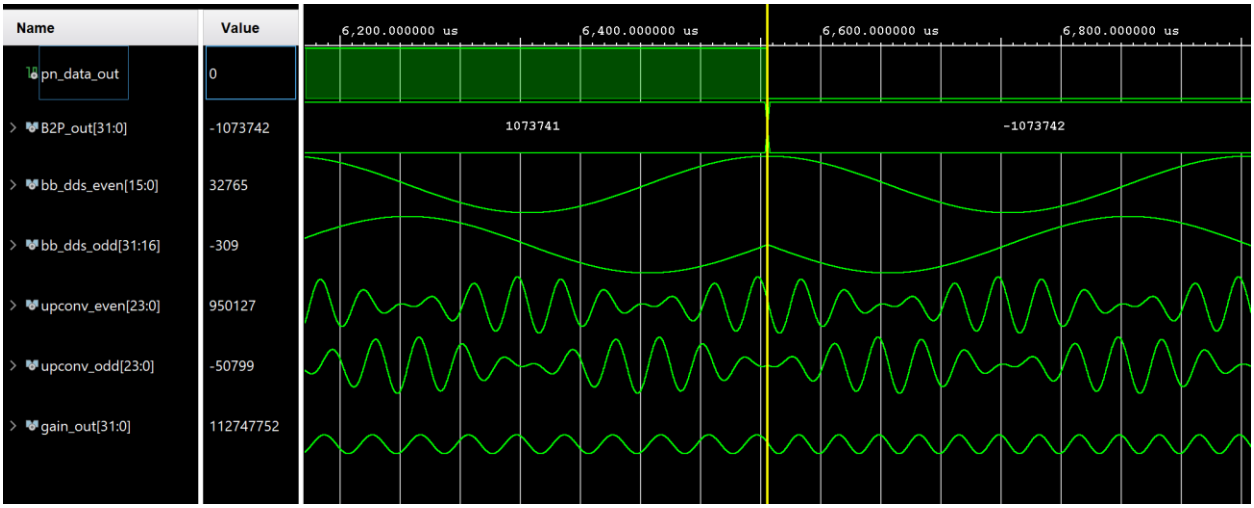

Figure 7: Wave Modulation Simulations in Vivado

The figure above displays the wave form at different stages at the time of a frequency switch. The top two sine waves are the even and odd streams of the baseband signal. In the even stream we see that switching to a negative frequency doesn't change the look of the waveform. This is compliant with

the behaviour of cosine waves when they are configured with a negative polarity. In the odd wave, a break can be seen at the frequency switch. This is again compliant with how sine waves behave. The two waves in the middle are the even and odd streams after being multiplied with the even and odd passband streams respectively. The wave at the bottom is the result of adding the even and odd stream together. By properly demonstrating in the Vivado simulation that the current waves are being generated, it can be said with confidence that the system up until this point behaves as intended. The multiplications and additions of the upconversion process are performed at 10MHz and the passband DDS runs at 100MHz. The final step within VHDL is to take the now upconverted signal and convert it to an analog version of itself.

4.1.4. Digital to Analog Conversion

One of the methods of digital to analog conversion is sigma delta modulation, otherwise known as pulse density modulation. The premise of sigma delta modulation is that upon being fed a digital signal, it outputs a pulse density that corresponds to amplitude. For high amplitudes, the pulse density modulatory (pdm) outputs a low frequency high duty cycle clock. For low amplitudes it will do the same but with a low duty cycle. For middle values the pdm outputs a high frequency clock. The pdm signal can then be turned into an analog sine wave by being fed into a low pass filter which averages out the values. The figure below shows a pdm waveform next to its corresponding sine waveform as simulated in Vivado.

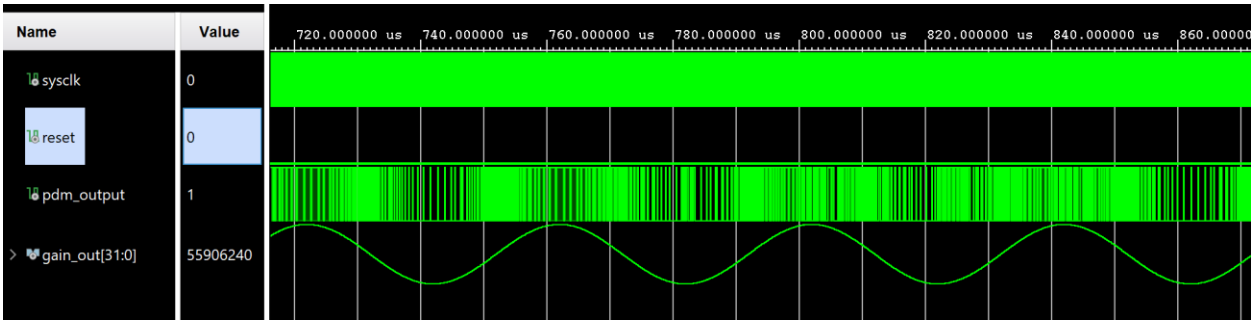

Figure 8: Digital to Analog Simulation in Vivado

One of the necessary steps in performing proper pulse density modulation is a multiplication of the signal prior to entering the pulse density modulator. Essentially, the signal is multiplied by an integer percentage to ensure that it does not become over saturated in the pulse density modulator, which causes imaging issues. In the case of this project, the integer percentage that provided the clearest results was 66 percent. Going higher would cause imaging issues and going lower would cause noise within the circuit to become more prevalent as the amplitude would be rather low. The pulse density modulator runs at a clock frequency of 10MHz. At the output of the pdm is the final signal that is desired to be output from the FPGA and so it is routed to a high speed PMOD port on the board. After being def through the low pass filter, the signal will be ready to go through a power amplifier and then be fed into the acoustic source that will transmit the underwater acoustic waves.

4.2. System Configuration 1: Frequency Hopping

Once the base level transmitter was completed, the upgraded configuration to be developed featured frequency hopping. This version of the chain was almost identical to the previous chain in components used and the purposes of each component. Only a few tweaks were necessary.

The main difference is at the bit-to-phase converter. Rather than having two frequencies for the two binary symbols, the 5 kHz bandwidth had to support 13 subcarriers. The first change is that rather than having 0 and 1 at the input translate to a ±2500Hz phase, the multiplexer was reconfigured to translate to ±96Hz. Also, another algorithm had to get fitted within the converter to cycle through the 13 subcarriers. What this would mean for the converter is that it would now require a clock. The final step within the converter was to take the subcarriers and add them to the ±96Hz signal

After all the tweaks, the updated functionality of the bit-to-phase converter is to cycle through 13 subcarriers and output a phase that corresponds to either 96Hz greater or lower than the current subcarrier, depending on whether the converter reads a 1 or a 0 at the input.

The only other tweak was configuring the system to pause for one second after each 1024-bit frame. Since the 1024-bit sequence is known and intended to be analyzed during post processing, the one second guard interval makes it easy to prevent post processing two frames at once, as there is a clear visual border in the readings.

4.3. System Configuration 2: Spread Spectrum

The second system configuration to be developed would feature a spread spectrum. For the spread spectrum configuration, everything from the previous frequency hopping configuration was copied over initially to a new Vivado project. The only necessary tweak was to have each bit in the PN generator span 13 times the timeframe of the previous version so that each bit would be sent through the entire spectrum of the 13 subcarriers.

4.4. System Configuration 3: Turbo Coding and Frequency Hopping

The third and final system configuration was to be a copy of the first frequency hopping configuration, but that also included turbo coding. Like the second configuration, everything from the first configuration was copied over to the third to begin with. After copying the first project, the convolutional encoder IP core from Vivado was implemented into the system between the PN generator and the bit-to-phase converter. The frequency of bits being fed into the converter is still 50Hz, just like the first, however, the actual PN generator runs at 25Hz and is fed into the turbo coder which outputs encoded bits at 50Hz. Other than implementing a convolutional encoder, no other alterations to the system were required.

5. Implementation Results

After configuring the three upgraded systems in VHDL, the system was tested in the Dalhousie Aquatron. The transmitter circuit was set up on a table next to the pool where the only component to go under water was the acoustic source. On the opposite end of the pool, a receiver was set underwater. The photograph below displays the transmitter circuit without the power amplifier, which includes the FPGA and lowpass filter printed on a PCB board.

The figure below shows the same circuit with the power amplifier attached on top. Before testing the system in the aquatron, it was first necessary to observe the signal on an oscilloscope in the lab. This

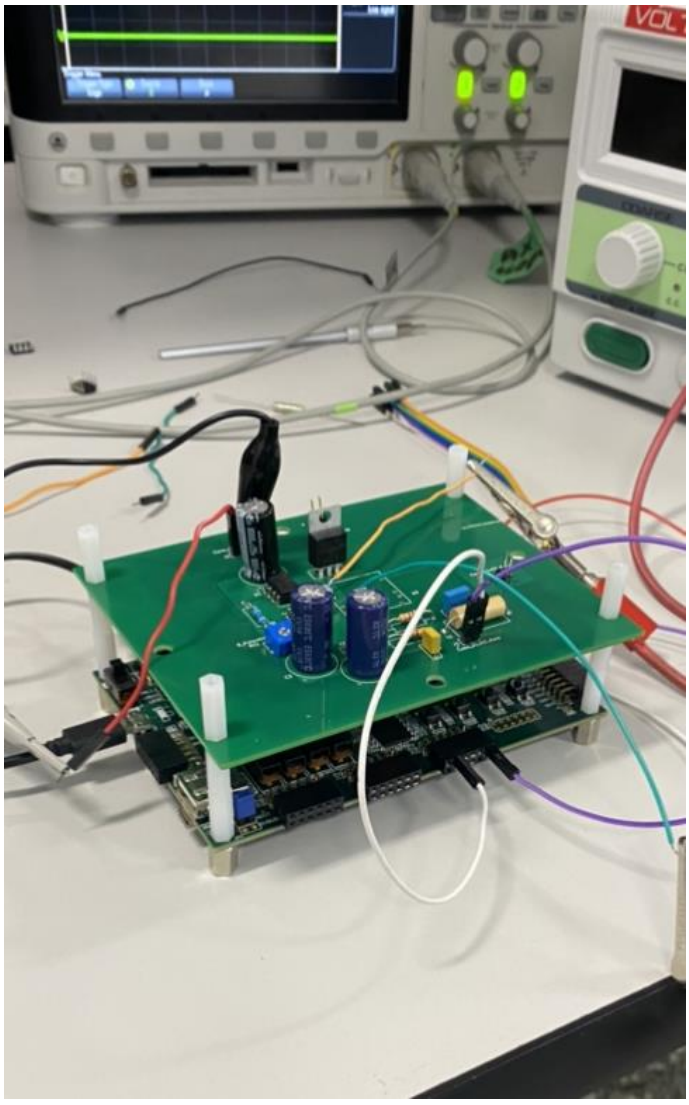

Figure 9: Implemented Transmitter Circuit

was to ensure the signal was outputting the proper frequencies. The figure below displays a snapshot of the oscilloscope readings for one of the systems.

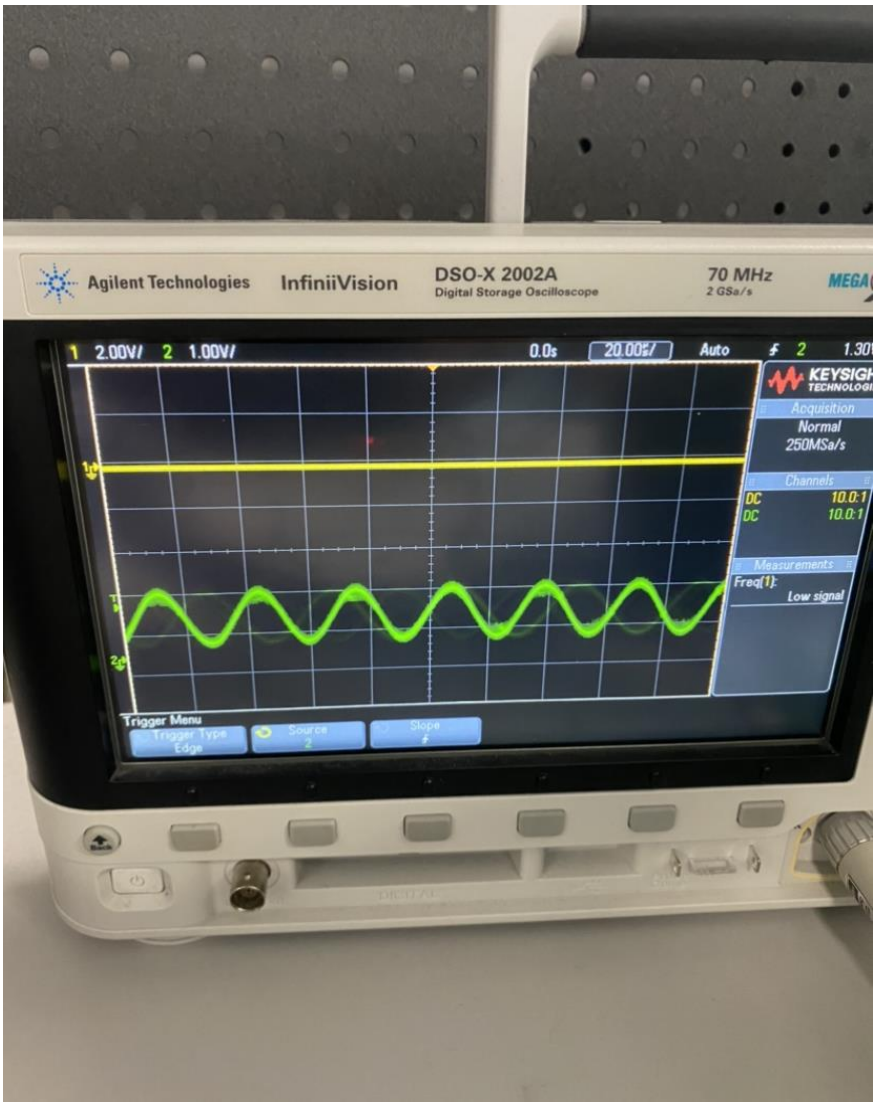

Figure 10: Filtered Signal without Power Amplification

As can be seen, at the output of the low pass filter is a clean sine wave with minimal noise. One of the hinderances however, is the low amplitude of the signal. As mentioned earlier when discussing sigma delta modulation, the signal had to be multiplied by an integer percentage that would lower the amplitude to prevent oversaturation. As a result, the amplitude does not use the whole 3.3V scale. Instead, the wave oscillates between approximately 0.9V and 2.1V. While this is not entirely ideal, it is acceptable as it will be visible enough for the receiver.

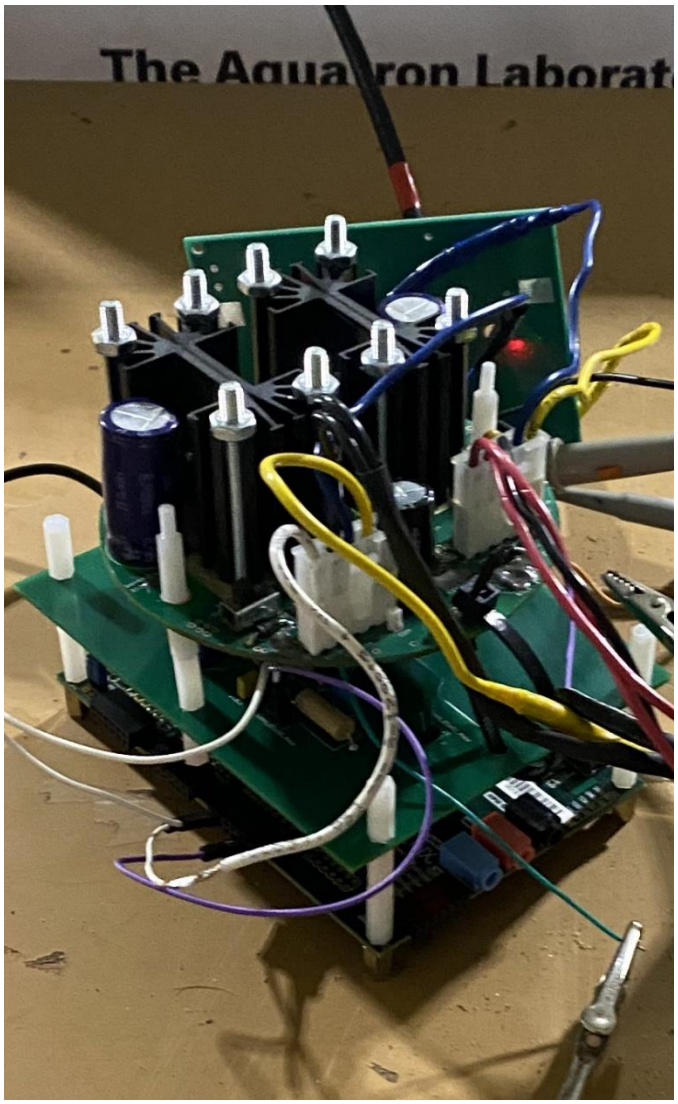

Figure 11: Implemented Transmitter Circuit with Power Amplifier

Upon running the signal through the system and into the water as an acoustic signal, a laptop connected to the receiver would stream a spectrogram representation of the readings in real time. The spectrogram displayed that proper frequency hopping was taking place with all three systems. It was also able to display the one second guard intervals that were used to separate each 1024-bit frame. The figure below displays a snapshot of the spectrogram that was streamed in real time.

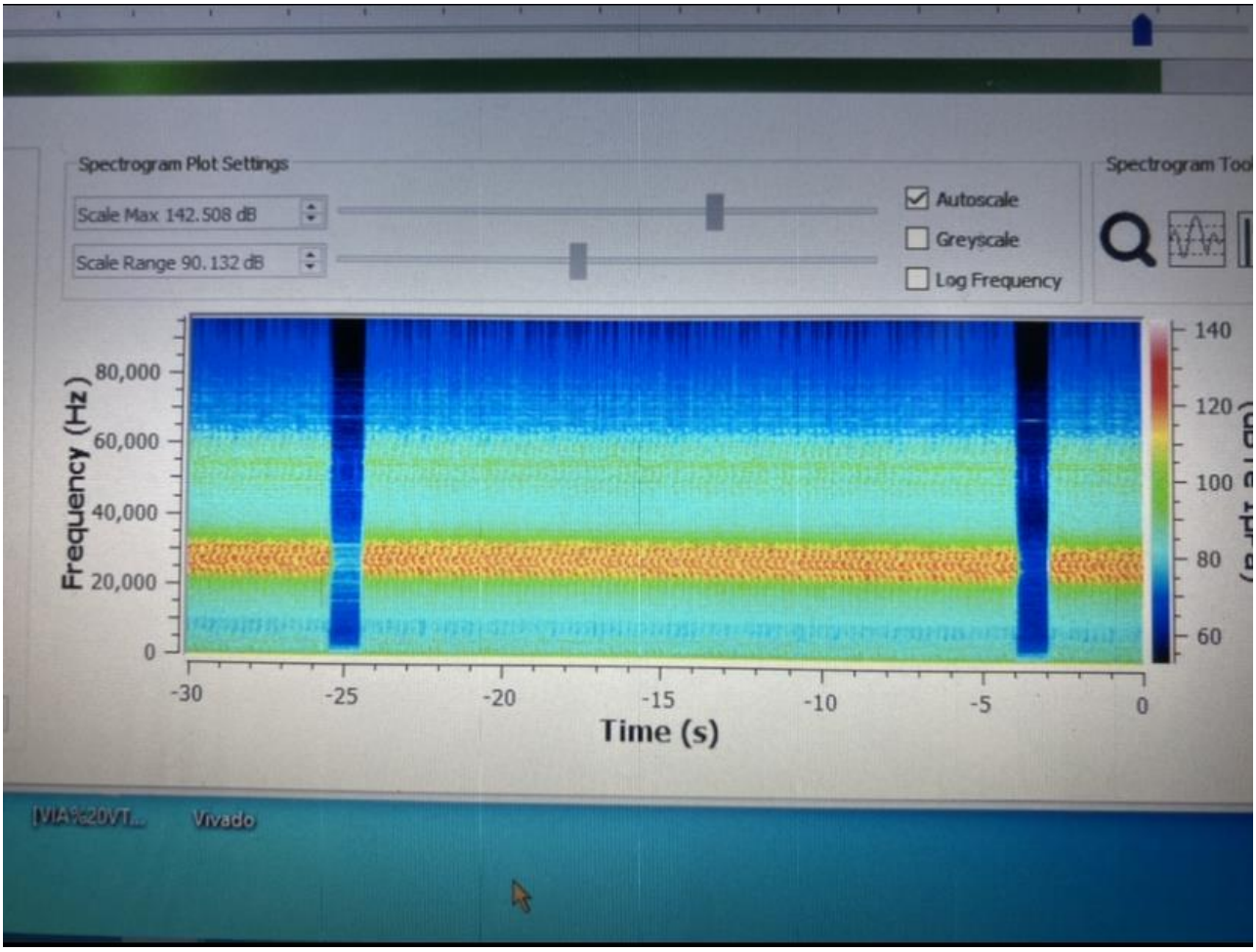

Figure 12: High Amplitude Signal Spectrogram

There is a presence of a second harmonic as can be seen in the spectrogram. This is due to the testing being done in the Aquatron, rather than in ocean. Essentially, the equipment was designed for communication of distances up to 1km. as a result, the system is expecting the signal to be attenuated in amplitude. Due to the small size of the Aquatron, the signal did not have a lot of distance travelled so it had not attenuated as much as the receiver would have expected, which had cause clipping in the received signal. As a result of the clipping, a second harmonic can be seen in the spectrogram. It should be noted that the harmonics are not expected to have a large effect on the quality of the readings, or the bit error rate, however, they are worth noting. One of the methods to mitigate the presence of the harmonic was to manually adjust the amplitude to a smaller magnitude to prevent clipping. Initially, at the output of the power amplifier was approximately a 12V amplitude signal. Upon being adjusted to 3V, the harmonic was significantly decreased, however, due to a smaller amplitude, noise became more present in the readings. In the end, a middle ground of 6V was settled on. The figure below displays the spectrogram of the 3V amplitude signal.

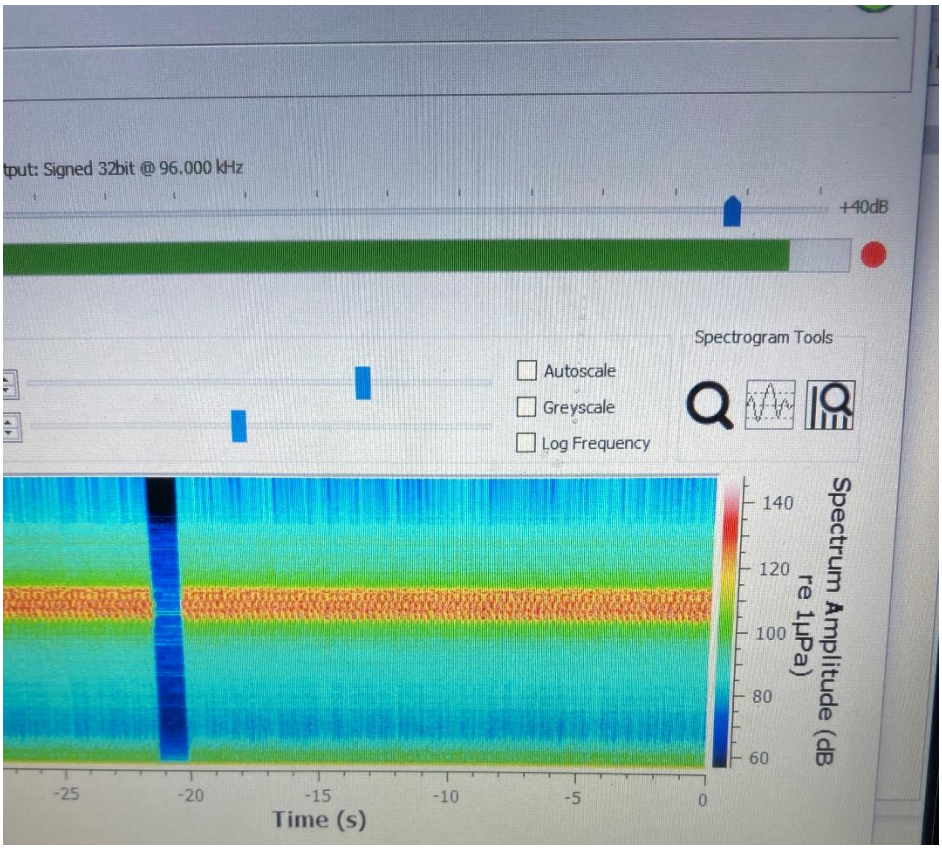

Figure 13: Low Amplitude Signal Spectrogram

As can be seen with the figure above, at 3V there is no longer a significant harmonic. The data that was read by the receiver was exported to separate wave files for later post-processing to assess the bit error rate. In terms of what can be seen with the oscilloscope and spectrogram, the system was functionally successful.

6. Conclusion

At the start of the project, a simple block design of the transmitter chain was proposed. It was desired to model and implement the transmitter chain to both analyze the quality of the system and prepare it for further improvements. Upon modelling the system mathematically in python, it was seen that the system is expected to function properly upon implementation. Upon implementing the system on an FPGA and testing it in an underwater environment, it was seen that proper frequency hopping and frequency switching occurred. In terms of future work and improvements, the system will need to eventually be fully submerged underwater. To be able to enclose the system, efficient and small batteries will need to be researched for their potential to fit into the system. Furthermore, the system will need to use the ZYNQ7 processing system to feature programmability. Moving forward, the system is ready to become a capstone project.

7. Recommendations for Future Work

For the system to be worked on in the future, the students that will take on the capstone project will need to learn about the FPGA environment themselves which in and of itself is not a trivial task. For that

reason, it would be smart to briefly talk about recommendations for learning how to use Vivado and VHDL. First and foremost, it is important to recognizer that FPGA software and hardware are both very well documented, however, unlike other fields of study, the documentation is not exactly set in a linear path which might make it difficult to know where to look or go next. Generally, when learning how to program an FPGA, the following order is recommended:

1. Learn basic VHDL syntax only to the extent of being able to implement a few logic gates.
2. Learn how to write a very basic testbench file to simulate the VHDL code
3. Learn how to use the simulation tool in Vivado to analyze your system using the written testbench
4. Learn how to set constraints and route entities in the code to physical switches and LEDS
5. Learn how to generate a bitstream and upload it to the board for physical testing
6. Go back and learn the rest of VHDL syntax to approach more complex designs. Emphasis is on how to implement "components" in VHDL. This is an extremely important concept to know.
7. Learn how to implement Vivado IP cores using "component instantiation"

By following this path, one can ensure that they will not waste time learning something they do not need to analyze. Due to the scope of this project, use of the AXI protocol was not necessary, nor was the use of the Zynq7 processing system that was featured on the FPGA board. Generally, I would say that these subjects should come after everything that is listed above.

8. Sources Cited

TutorialsPoint. "Frequency Shift Keying".
https://www.tutorialspoint.com/digital_communication/digital_communication_frequency_shift_keying.htm

Sound Devices. "What is multipath interference?".

https://www.sounddevices.com/multipath-interference-and-diversity-switching/