

# **ELEC 391 Product Report**

Team 6

Yuqian Song, Chloe Sun, Pengyu Ji

## Table of Contents

<b>Objectives, Requirement &amp; Constraints</b>	3
<b>System Overview</b>	4
Source / Sink	5
A/D and D/A converters	7
Error Correction Encoder/ Decoder	12
Modulator / Demodulator	14
Transmitter / Receiver	16
Channel	19
<b>Verification</b>	23
Source / Sink	23
A/D and D/A converters	23
Error Correction Encoder / Decoder.	24
Channel	28
<b>Appendix A: Deliverables</b>	30
<b>Appendix B</b>	30
<b>Team Contributions</b>	32

## Objectives, Requirement & Constraints

The main goal of this project is to design a reliable digital communication system that can transmit audio and data over a wireless channel. The first phase of the project is to develop the system using Simulink by choosing suitable Simulink blocks and tuning the block parameters. After the Simulink model is finalized, we need to implement every block in hardware using HDL. Finally, we compile our design files and download them into the FPGA.

The following table is our design requirement:

Audio BW (kHz)	Target BER	Spectral Mask (kHz)	Target Channel	Delay (ms)
20	1e-3	200	B	25

During the design process, we encountered several significant challenges. For example, the different blocks we used operate at varying frequencies, which necessitated adjustments to the clock settings to ensure proper synchronization. In our FPGA implementation, to ensure our bandwidth requirement of 20 kHz is met, we implemented a clock divider to generate a 40 kHz clock frequency for the input data width. The channel frequency in the scenario is 1 MHz, so we used a PLL IP block to generate a 1 MHz clock frequency for the channel. The quantization level is adjusted by the data shifter module. We selected a 16-bit input width to balance audio quality with the requirements of the subsequent Hamming encoder process (21:16).

Moreover, when it came to noise generation, we initially considered direct random number generation using “\$random”. However, this approach proved infeasible on the FPGA, leading us to implement Linear Feedback Shift Registers (LFSR) instead. This method allowed us to generate random numbers efficiently within the FPGA constraints.

Additionally, our goal was to generate Gaussian noise for realistic channel simulations. The direct implementation of cosine and sine functions in our HDL was not synthesizable, so we utilized the Central Limit Theorem to approximate Gaussian noise. This approach is to use CLT to sum uniform random variables to achieve a distribution that closely mimicked Gaussian noise, ensuring effective and accurate noise simulation.

Through these solutions, we were able to address the challenges and meet the design objectives of our digital communication system.

## System overview

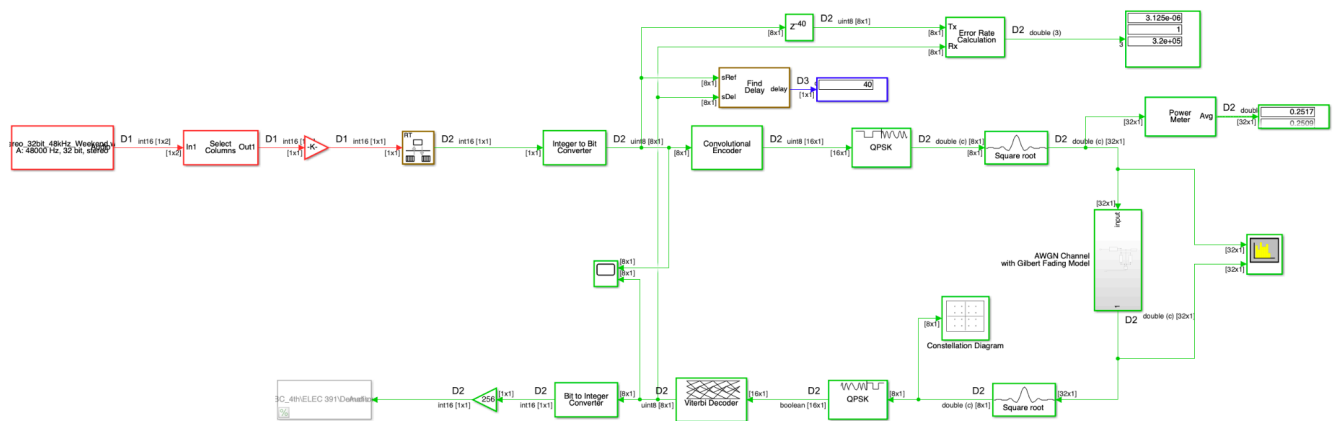


Figure 1: Simulink model of the system

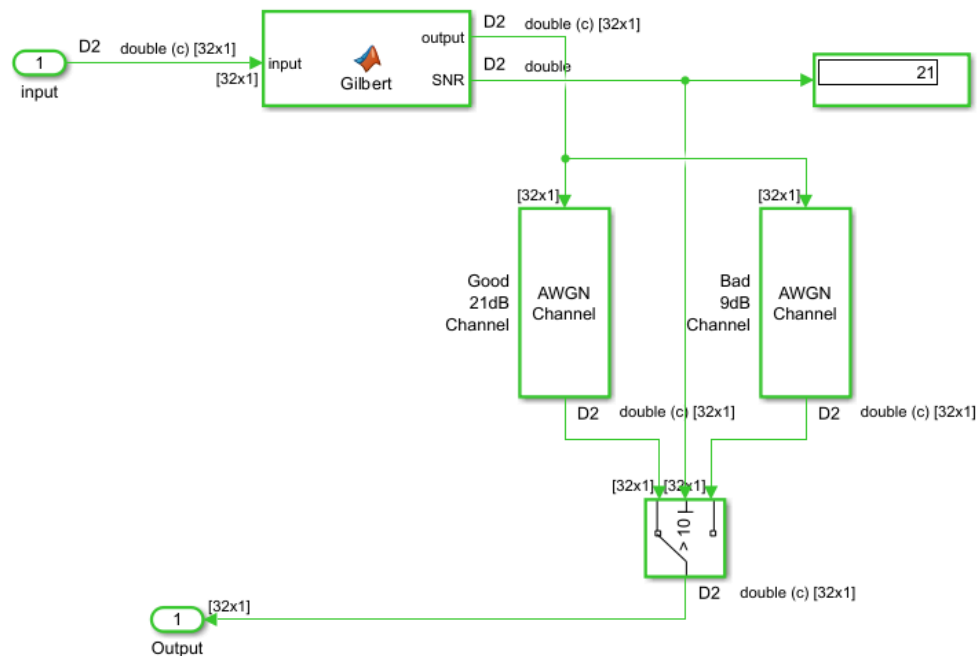


Figure 2: subsystem for AWGN channel with Gilbert Fading Model

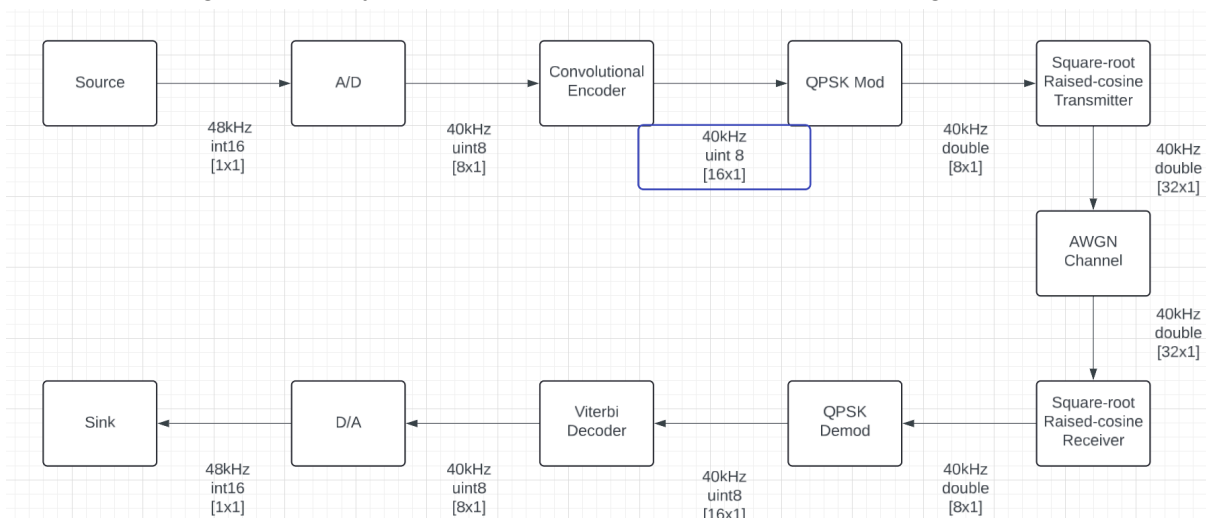


Figure 3: system block diagram

Criterion	Simulink Perf.	FPGA Perf.
Message Transmission (bit rate)	320 kbps	640 kbps
Transmission Reliability (bit error probability)	3.1e-6	
Processing Delay	1 ms	29 ns
Channel Bandwidth	210 kHz	660 kHz

Our digital communication system is designed to achieve reliable audio transmission by balancing processing delay, channel bandwidth, transmission bit rate, and bit error rate to meet specific project requirements. To meet the 200 kHz channel bandwidth requirement, we select a roll-off factor ( $\beta$ ) of 0.5 for the raised cosine filter. This low roll-off factor minimizes bandwidth usage, but it also causes higher ripples in the time domain, necessitating a longer filter span to capture all necessary samples and waveform information accurately. Encoder/decoder and modulator/demodulator are the two blocks that can directly influence the bit error rate. To achieve the ideal bit error rate, we choose the QPSK modulator/demodulator, as it gives the lowest bit error rate than the other modulation schemes like 8PSK or 16QAM. In terms of encoder/decoder, we initially tried the Hamming and BCH encoder/decoder, but the smallest bit error rate we can achieve is 0.14, and the output sound file still sounds noisy. As a result, we choose the combination of Convolutional encoder and Viterbi decoder, which gives us a bit error rate close to zero. Due to time constraints, we could not implement a circuit to calculate the bit error rate of our FPGA implementation.

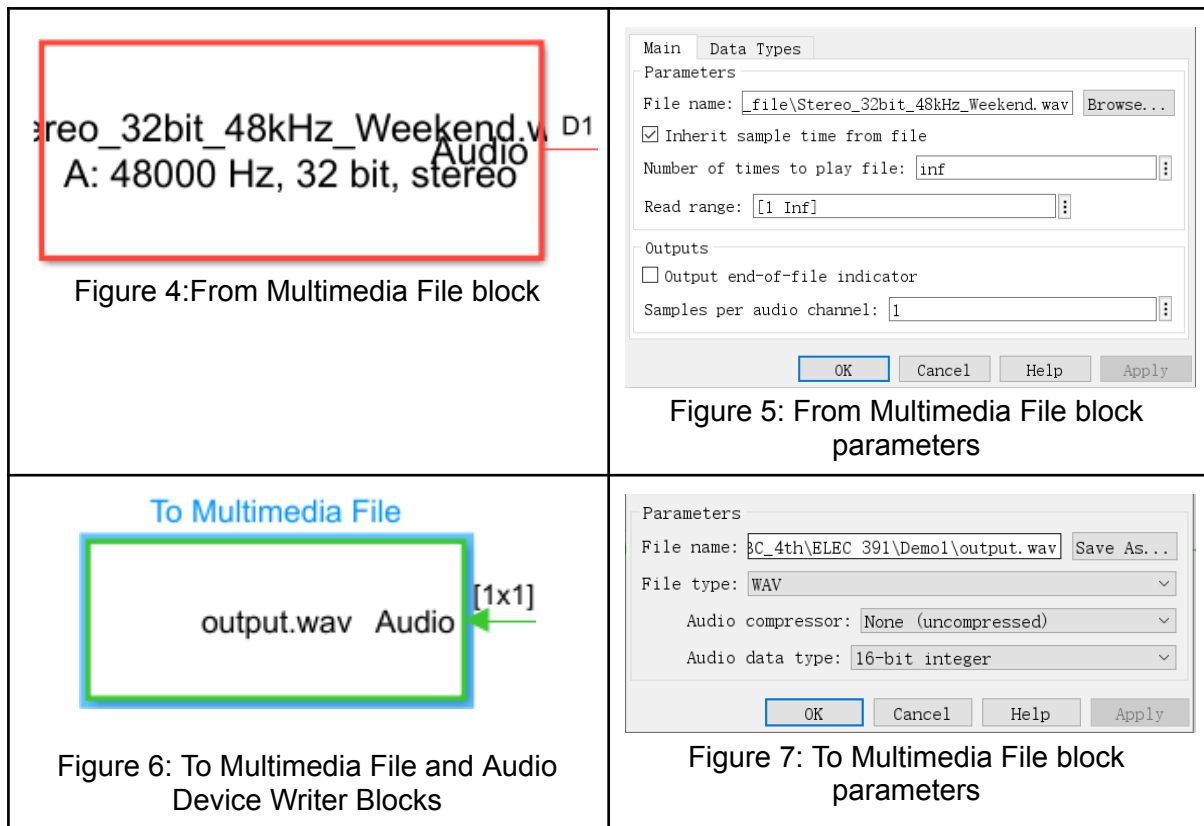
## Subsystem design

### Source / Sink

We used the "From Multimedia File" block to import our audio sample, selecting a 32-bit, 48 kHz sampling rate stereo audio file. We set the output data type to a 16-bit integer and configured the samples per audio channel to 1. Consequently, the source output is a 1x2 matrix.

We chose a 32-bit audio file instead of a 16-bit one due to the substantial benefits it offers. The 32-bit format provides an extremely wide dynamic range, allowing the capture of both very quiet and very loud sounds without distortion. This format is particularly advantageous in audio processing and editing because it can handle signals with very low and very high amplitudes without distortion. Although this choice results in larger storage requirements and longer processing times, we believe the trade-off is justified to achieve higher quality audio.

### Simulink:



## FPGA Implementation:

### 1. Block diagram

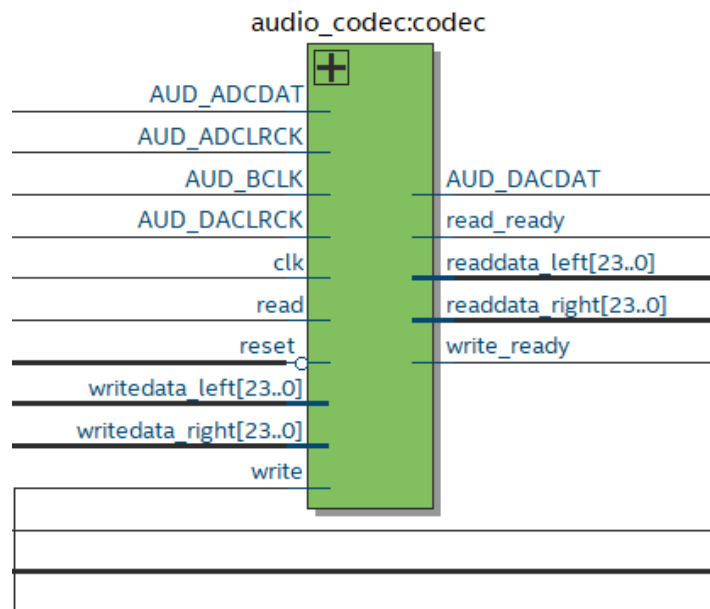


Figure 8: Audio\_codec block diagram

### 2. Design Justification

To thoroughly evaluate our system's performance, we conducted tests using a microphone and speaker, which provided clear and noticeable results. These tests validated our system's

ability to handle real audio signals effectively, demonstrating the practical application and robustness of our design.

### 3. Subsystem components and custom blocks

The speaker and microphone that we used were the subsystem components we had.

#### A/D and D/A converters

In this project, we use a Variable Selector block to handle the output from the From Multimedia File block, which provides a 2-channel audio matrix.

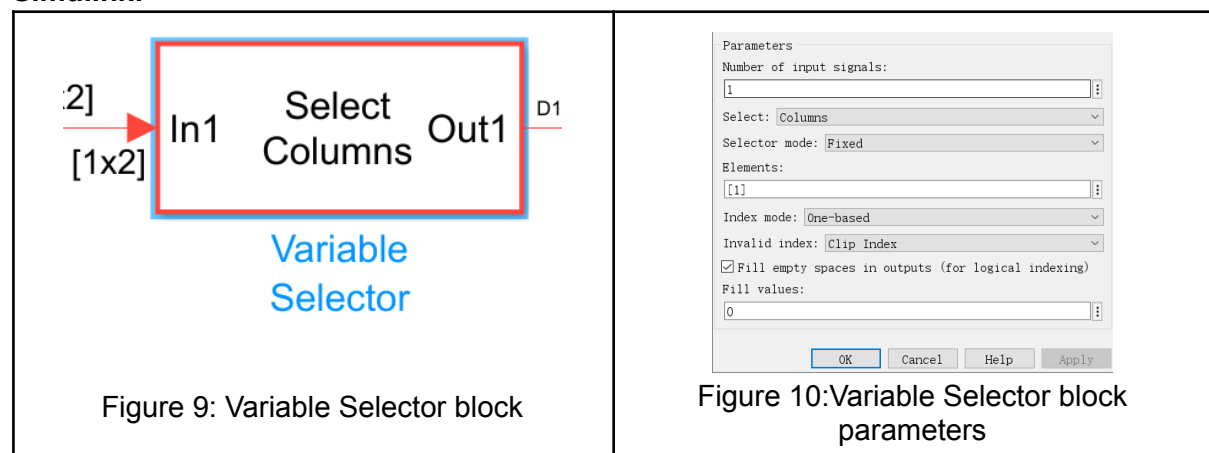
Initially, the Multiply block decreases the quantization level of the audio data to meet the specific scenario requirements. Reducing the quantization level helps to lower the bandwidth and bit rate/symbol rate during transmission through the channel.

Following this, a Rate Transition block down-samples the audio signal from 48 kHz to 40 kHz. This step ensures that the audio bandwidth in the frequency domain meets the required 20 kHz.

The bitstream is reconverted into a 16-bit integer by the Bit to Integer Converter block. To counteract the increased noise power caused by the lower quantization level and to enhance audio quality, we use another Multiply block to restore the quantization level to its original state.

Finally, This process effectively balances the need for reduced bandwidth and bit rate during transmission while maintaining high audio quality through proper quantization level adjustments and sampling rate conversion.

#### Simulink:



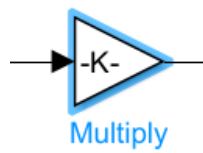


Figure 11: Multiply bolck

Figure 12: Multiply block parameters

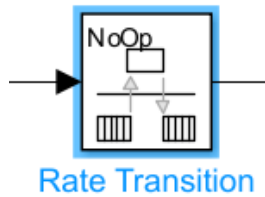


Figure 13: Rate Transaction block

Figure 14: Rate Transaction block parameters

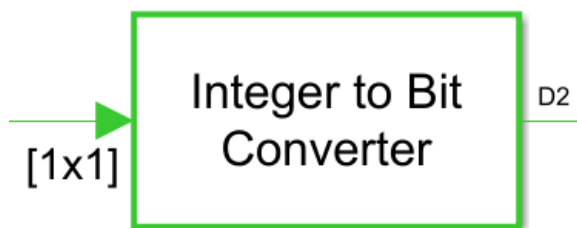


Figure 15: Integer to Bit Converter

Figure 16: Integer to Bit Converter parameters

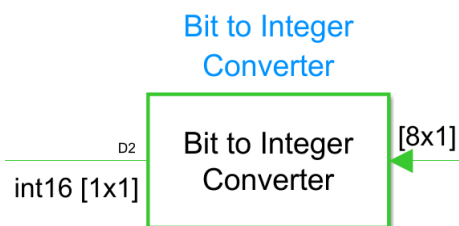


Figure 17: Bit to Integer Converter

Figure 18: Bit to Integer Converter parameters

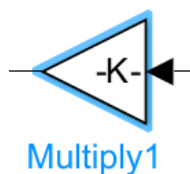


Figure 19: Multiply block

Figure 20: Multiply block parameters

## FPGA Implementation:

### 1. Block diagram



data\_shifter\_right:sampl

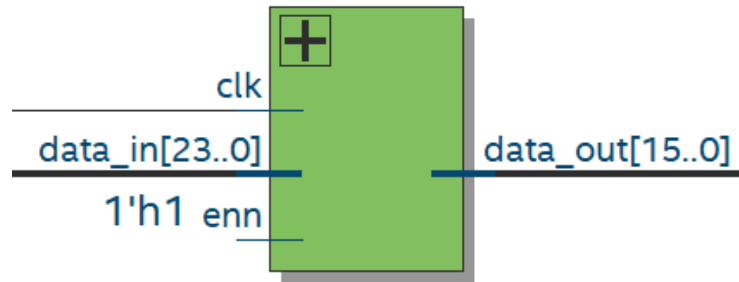


Figure 21: Data\_shifter\_right Block Diagram(left channel)

data\_shifter\_right:sampr

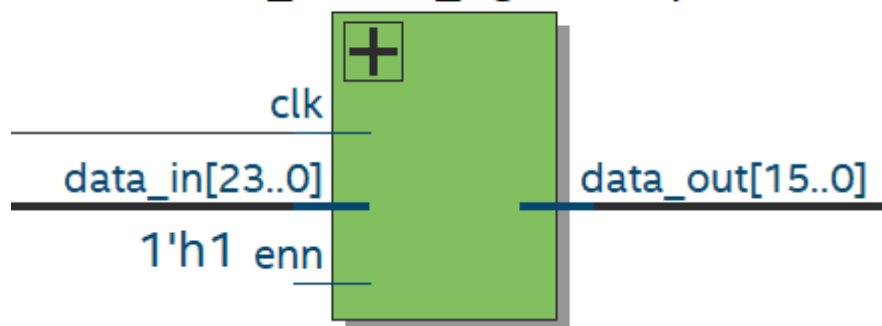


Figure 22: Data\_shifter\_right Block Diagram(right channel)

audio\_codec:codec

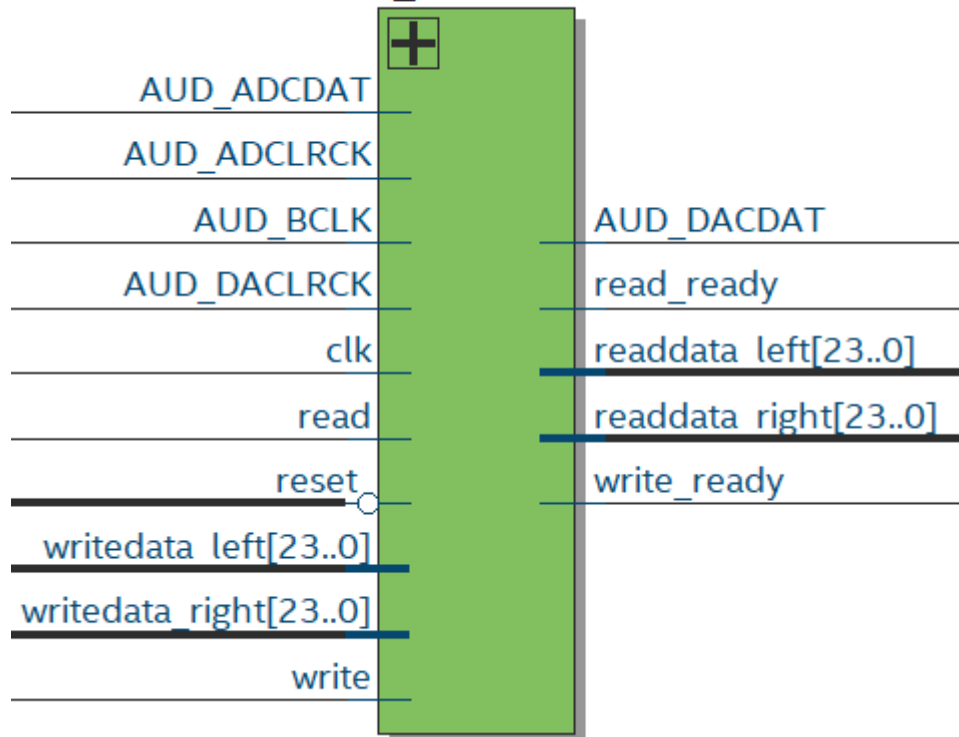


Figure 23:Audio\_codec Block Diagram

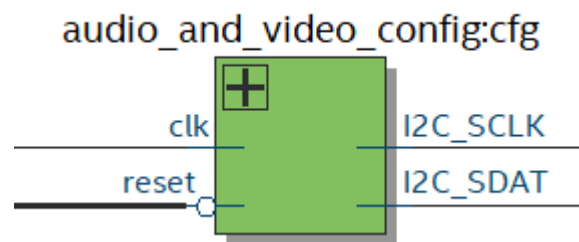


Figure 24:Audio\_and\_video\_config Block Diagram

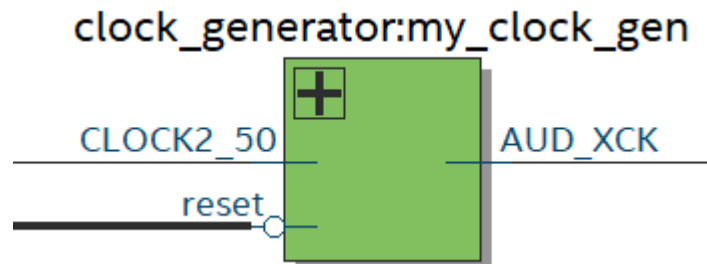


Figure 25:Clock\_generator Block Diagram

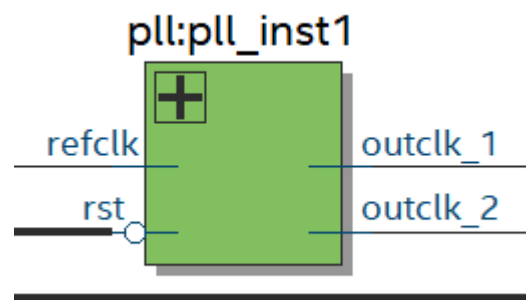


Figure 26:PLL Block Diagram

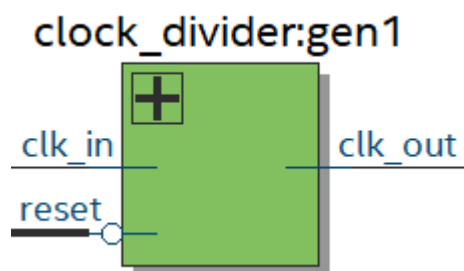


Figure 27:Clock\_divider Block Diagram

## 2. Design Justification

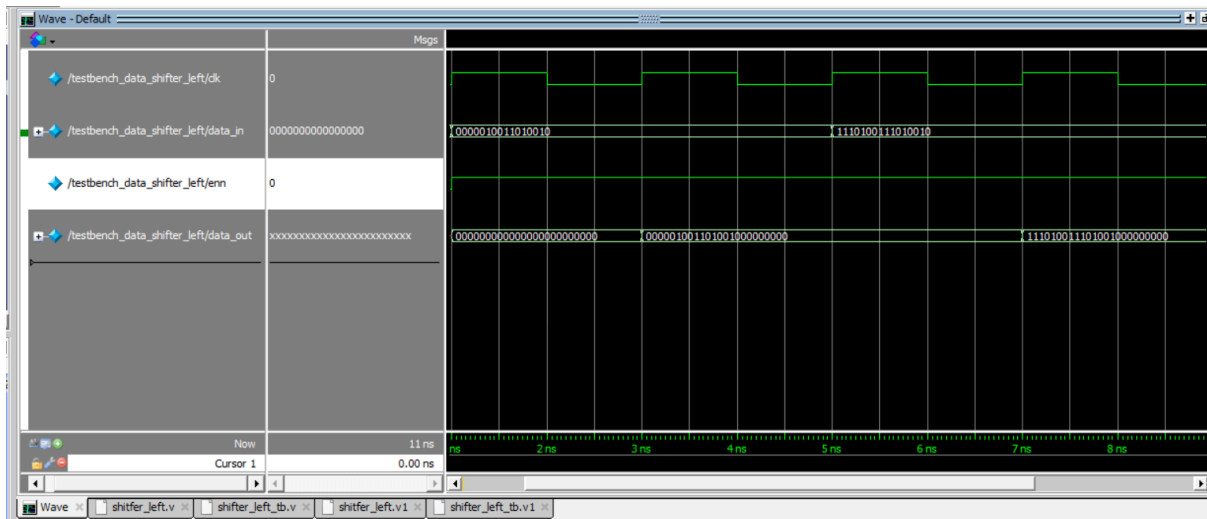


Figure 28: Testbench of Left Data Shifter

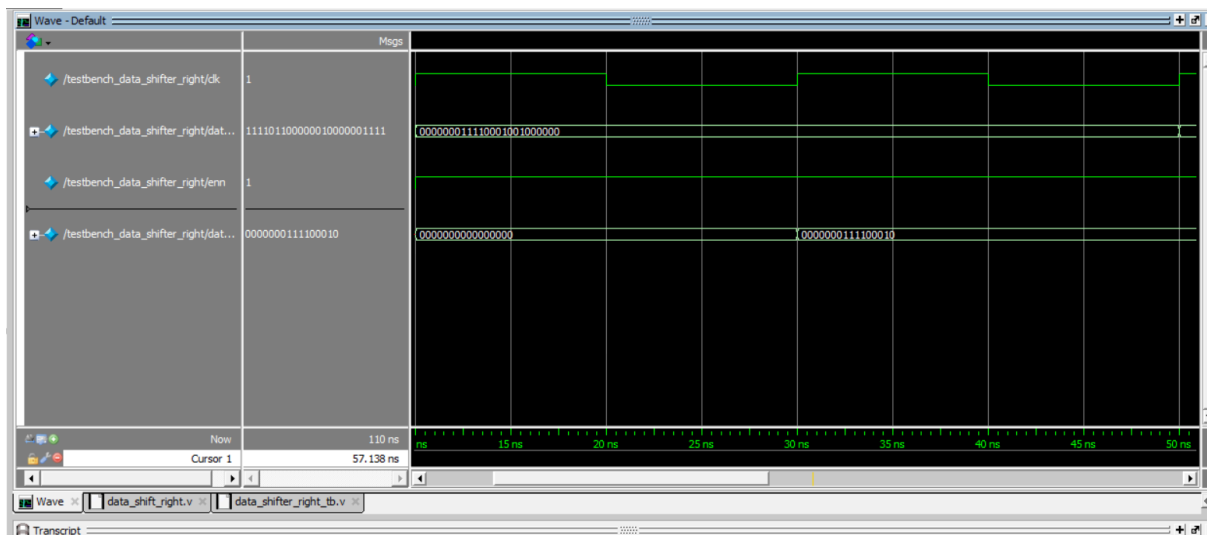


Figure 29: Testbench of Right Data Shifter

As the testbench shows that the data\_shifter\_left module shifts a 16 bit data to a 24 bit data. The data\_shift\_right module shifts a 24 bit data to a 16 bit data.

### 3. Subsystem components and custom blocks

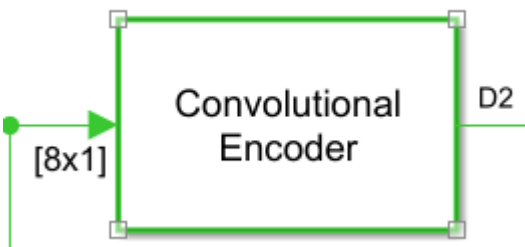
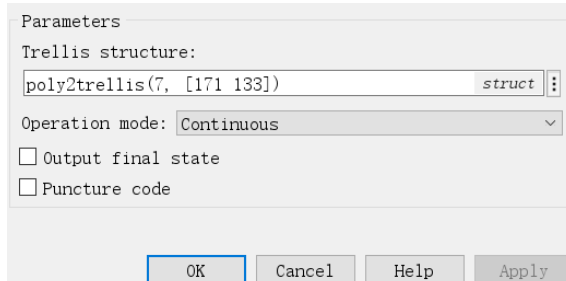
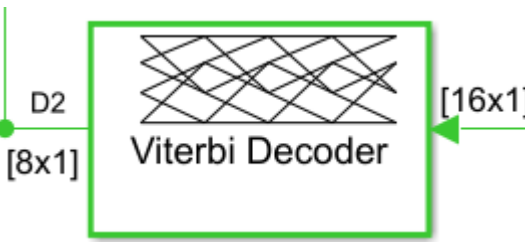
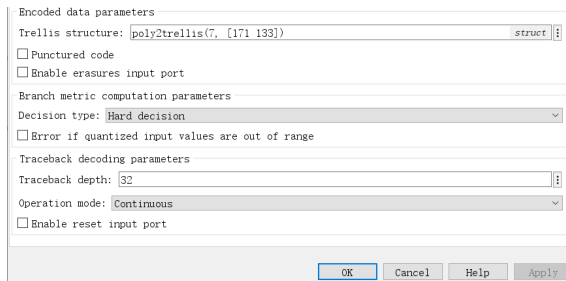
In our FPGA implementation of the AD/DA block, the default input bit depth is 24 bits. To accommodate this, we have implemented a right bit-shifting module that converts the 24-bit input data into a 16-bit signed integer. This reduction in bit depth is necessary for processing within certain blocks of our BCH encoder/decoder. Subsequently, we use a left bit-shifting module to convert the 16-bit data back to its original 24-bit format, ensuring compatibility with the audio\_codec and enabling proper output to the speaker.

To achieve the sampling frequency required by our scenario, we need to generate a 40 kHz clock cycle. However, the lowest frequency the PLL IP core can generate is 0.6 MHz. To address this, we designed a clock divider module that toggles the clock signal every 15th count. This effectively divides the 0.6 MHz clock down to the desired 40 kHz frequency.

These steps ensure that the data is accurately processed and converted, maintaining audio quality while meeting the required sampling rates for effective signal transmission and output.

## Error Correction Encoder/ Decoder

### Simulink:

 <p>Figure 30: Convolutional Encoder</p>	 <p>Figure 31: Convolutional Encoder block parameters</p>
 <p>Figure 32: Simulink Viterbi Decoder block</p>	 <p>Figure 33: Viterbi Decoder block parameters</p>

The convolution encoder applies a polynomial-based encoding scheme to the input bit stream, introducing redundancy by outputting multiple bits for each input bit according to the specified generator polynomials. This redundancy helps in detecting and correcting errors in the transmitted data. At the receiver end, the convolution decoder uses Viterbi algorithm to decode the received bit stream by identifying the most likely original data sequence, considering the redundancy introduced by the encoder. This process significantly enhances the reliability of data transmission in communication systems by mitigating the impact of noise and other transmission impairments.

### FPGA Implementation:

#### 1. Block diagram

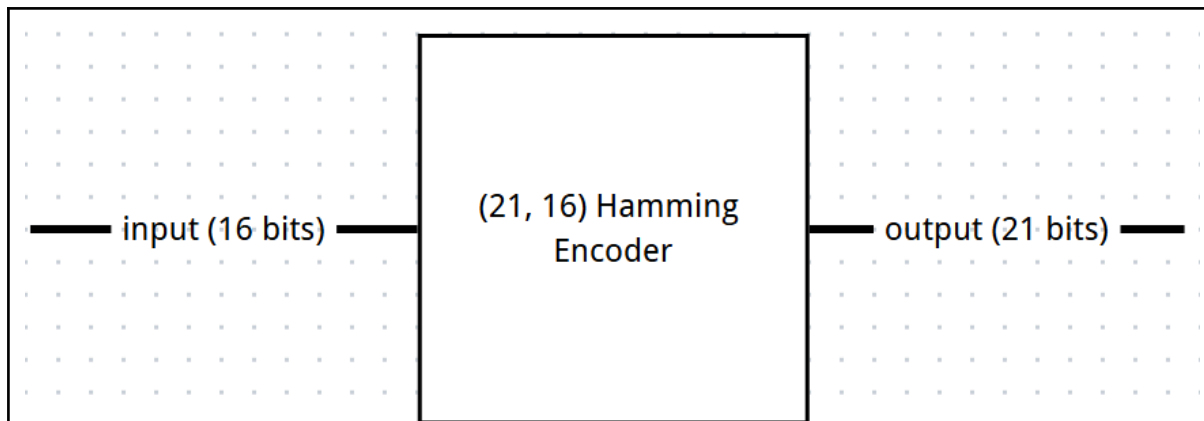


Figure 34: Block diagram of the encoder

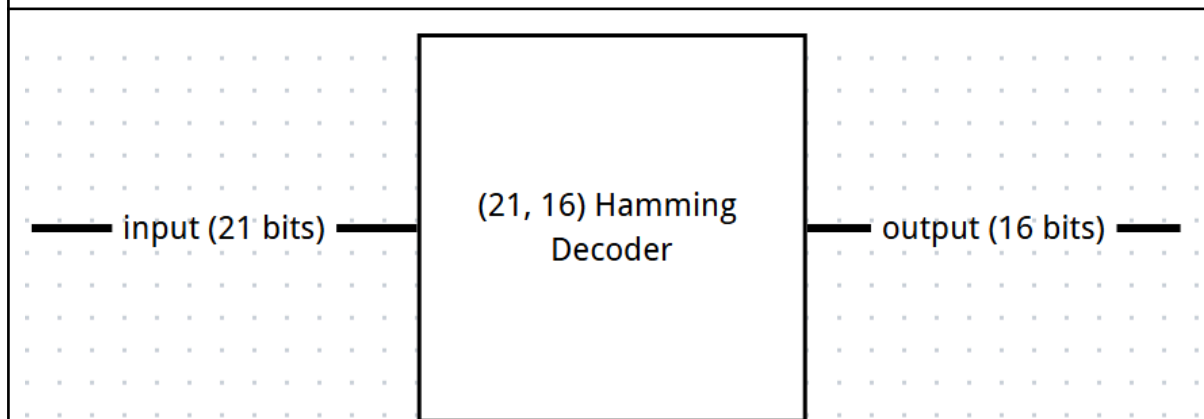


Figure 35: Block diagram of the decoder

## 2. Design Justification

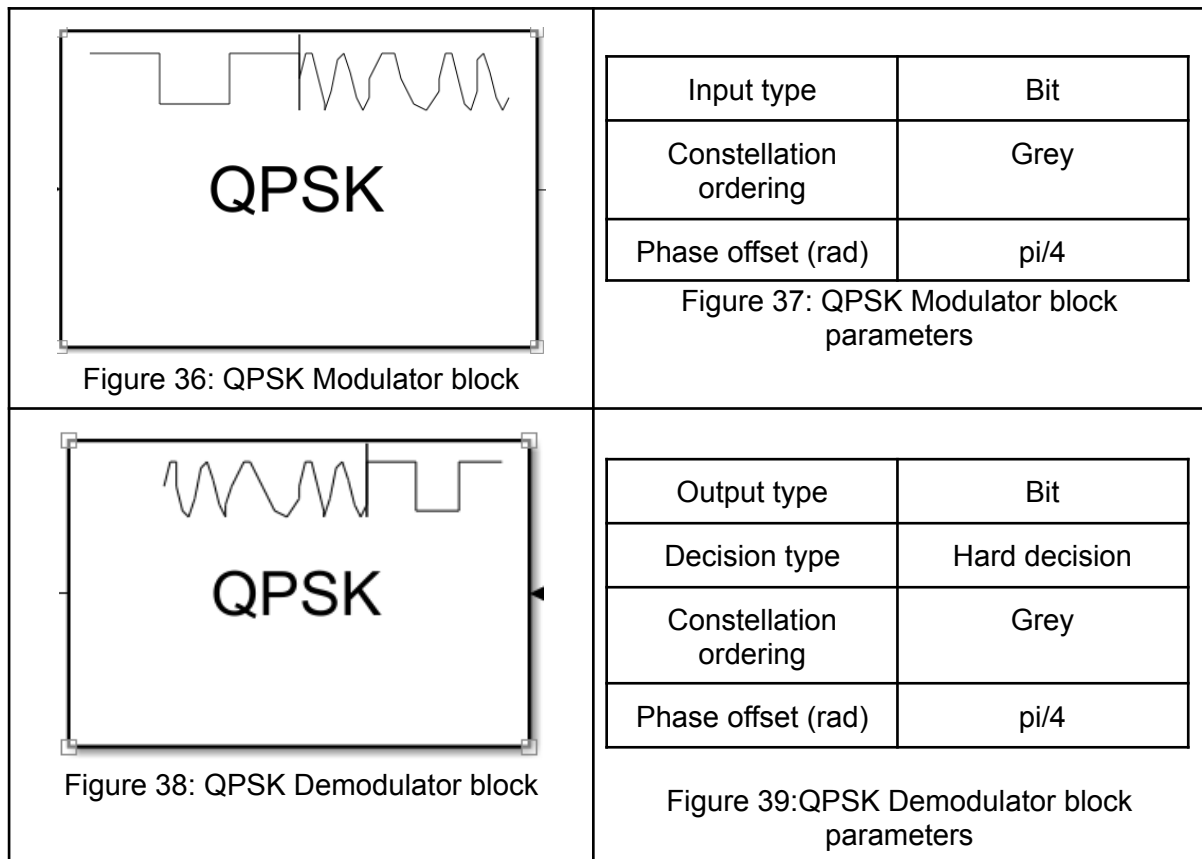
We tried but failed to implement the convolutional encoder and the Viterbi decoder in systemverilog. As a result, we decided to implement the (21, 16) Hamming encoder and decoder. We chose the Hamming encoder/decoder because it is straightforward to implement with a lot of documentation online. Also, the output of our A/D converter is 16 bits, so it's more suitable to choose the message length of 16 bits.

## 3. Subsystem components and custom blocks

In our FPGA implementation, we did not utilize any pre-defined subsystem components or custom blocks. Instead, we designed and implemented all necessary modules from scratch to meet the specific requirements of our project.

## Modulator / Demodulator

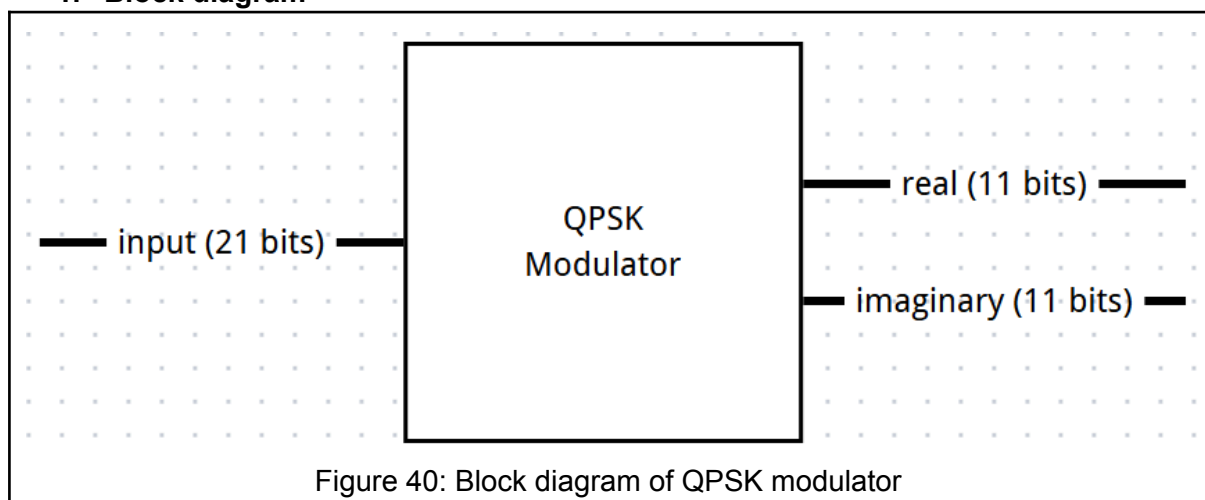
Simulink:

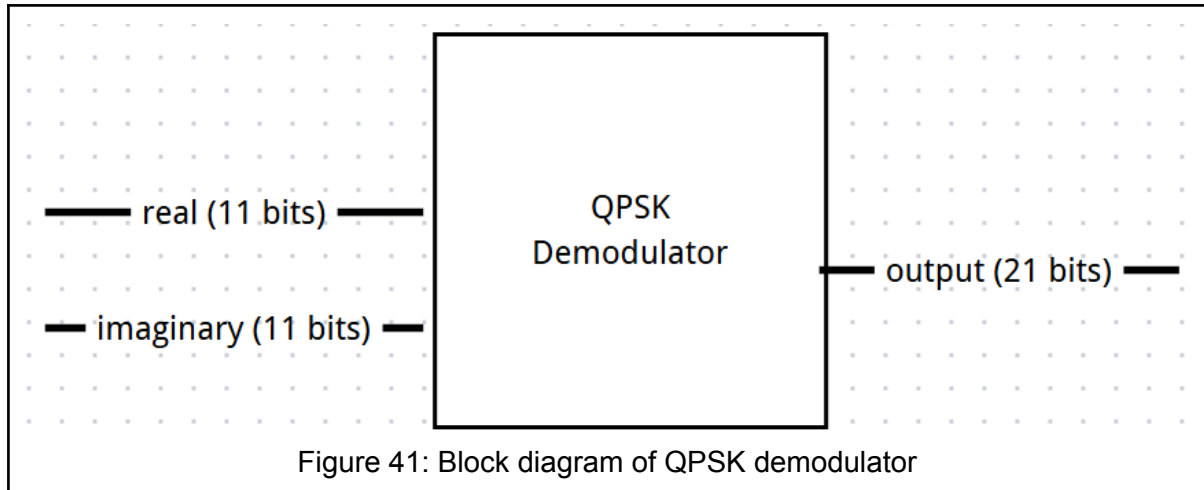


The modulator transforms bits into real and complex symbols that make up the signal constellation; the demodulator maps the received symbols (with noise) to the most probable constellation symbol. The project document asked us to use QPSK or higher modulation. We tested QPSK, 8PSK, and 16PSK in our system. We found that using QPSK yields the smallest bit error rate, which is also very close to the desired 0.001. In comparison, 8PSK and 16PSK yields a BER of approximately 0.01 and 0.02, respectively.

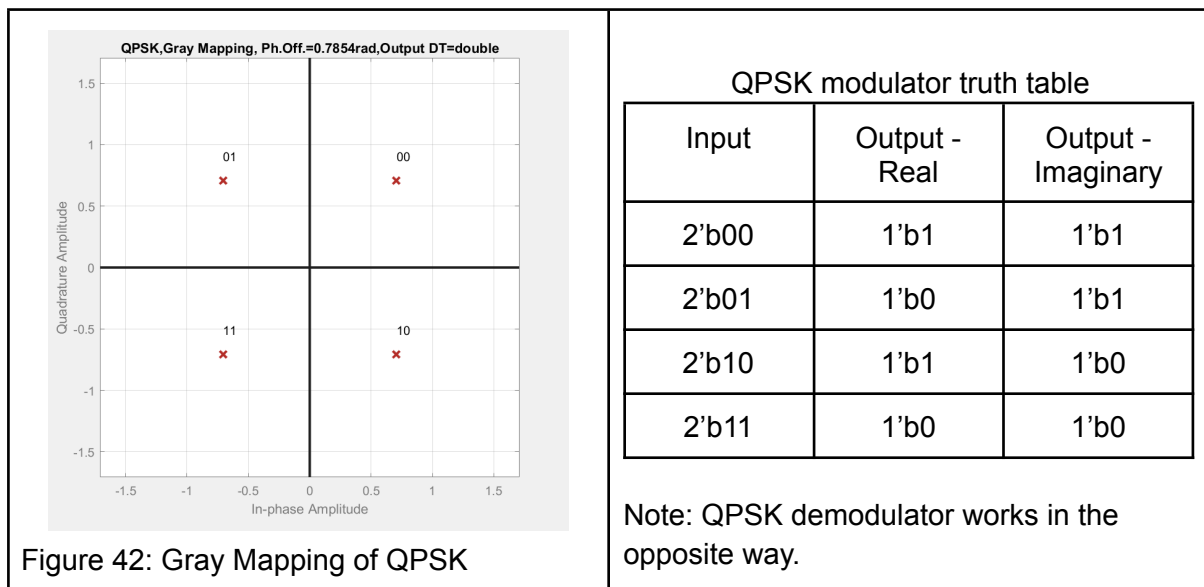
## **FPGA Implementation:**

### **1. Block diagram**





## 2. Design Justification



In our FPGA implementation, we use 1 to represent positive output value and 0 to represent negative output value. We originally implemented a sequential logic, in which the modulator outputs a single bit of real value and imaginary value in every clock cycle. However, we faced problems because it takes 11 cycles for a 22-bit input to be processed, but the encoder output updates every clock cycle. As a result, we choose to implement a purely combinational logic, in which the modulator has two 11-bit outputs, one for real value, and the other for imaginary value. The QPSK demodulator operates in the opposite way, which takes two bits (one real and one imaginary) and maps them to the corresponding 2-bit binary representation. Since the QPSK modulator needs an even number of bits as the input, we append a 1-bit 0 at the index 0 of the 21-bit input to make it 22 bits.

## 3. Subsystem components and custom blocks

The QPSK modulator consists of two different blocks: one that receives the 21 bit input data and zero pad by 1 bit, the other that converts the input data into complex plane representation of the data. The demodulator does this in reverse order.

## Transmitter / Receiver

For both the transmitter and receiver, we opted for square root raised cosine filter blocks. These filters proved to be efficient and convenient, as we could easily adjust their bandwidth to match our channel by modifying the roll-off factor and filter span.

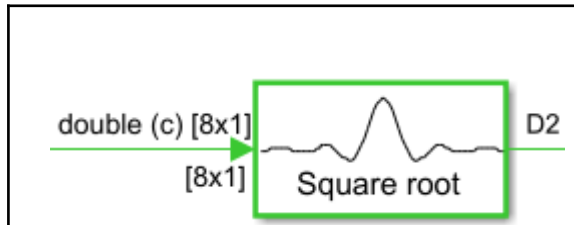


Figure 43: raised cosine transmitter block

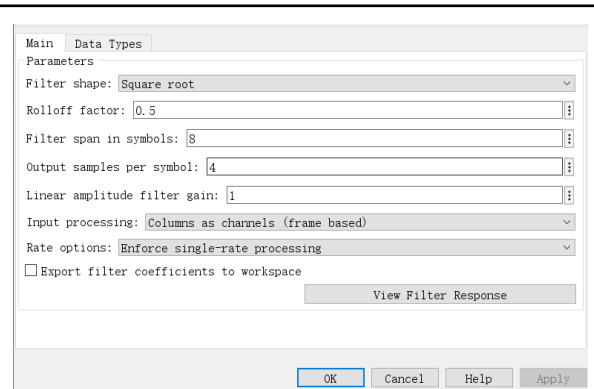


Figure 44: transmitter parameters

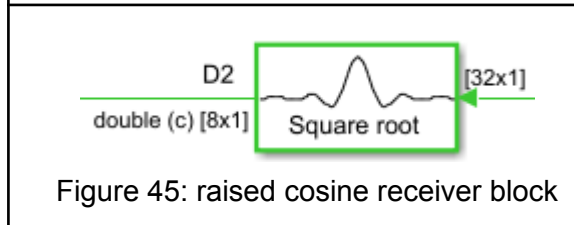


Figure 45: raised cosine receiver block

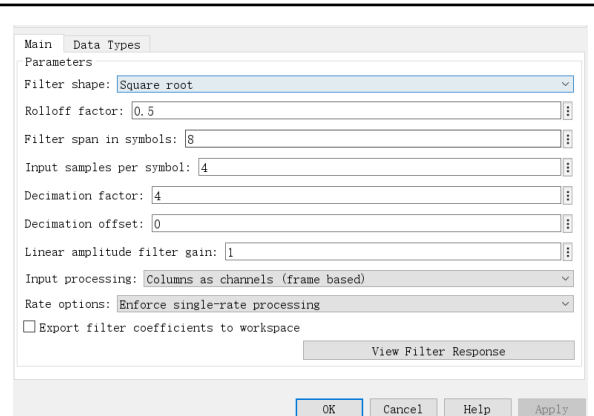


Figure 46: receiver parameter

## FPGA Implementation:

### 1. Block diagram

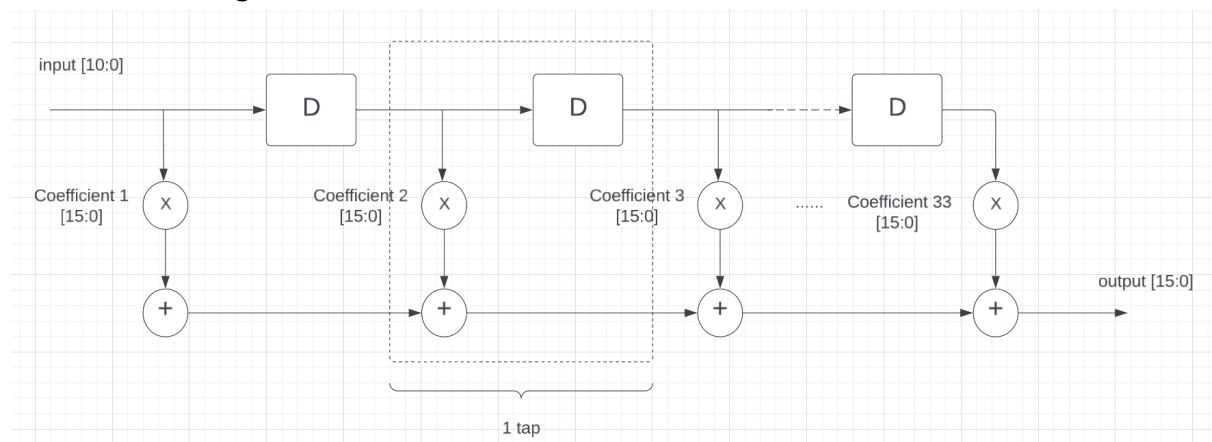


Figure 47: Detailed block diagram for transmitter



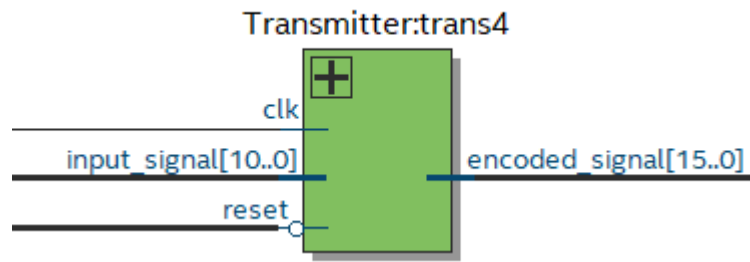


Figure 48:Transmitter Block Diagram

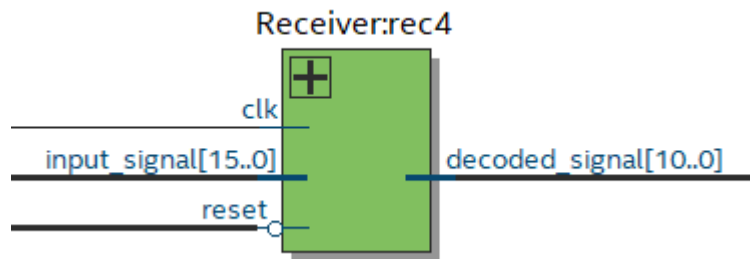


Figure 49: Receiver Block Diagram

## 2. Design Justification

Coefficients
-0.00505281304553084
-0.00191087927654681
0.00535931755284406
0.0082264616762359
0.00151584391365925
-0.0082264616762359
-0.00750304457398167
0.00773451135745139
0.0212218147912295
0.00773451135745138
-0.0375152228699084
-0.0784256013134489
-0.0530545369780738
0.078425601313449
0.289331991458557
0.487274215519437
0.56834083728333
0.487274215519437
0.289331991458557
0.078425601313449
-0.0530545369780738
-0.0784256013134489
-0.0375152228699084
0.00773451135745138
0.0212218147912295
0.00773451135745139
-0.00750304457398167
-0.0082264616762359
0.00151584391365925
0.0082264616762359
0.00535931755284406
-0.00191087927654681
-0.00505281304553084

## Coefficient Function Plot

Y Coefficient Value , X Index

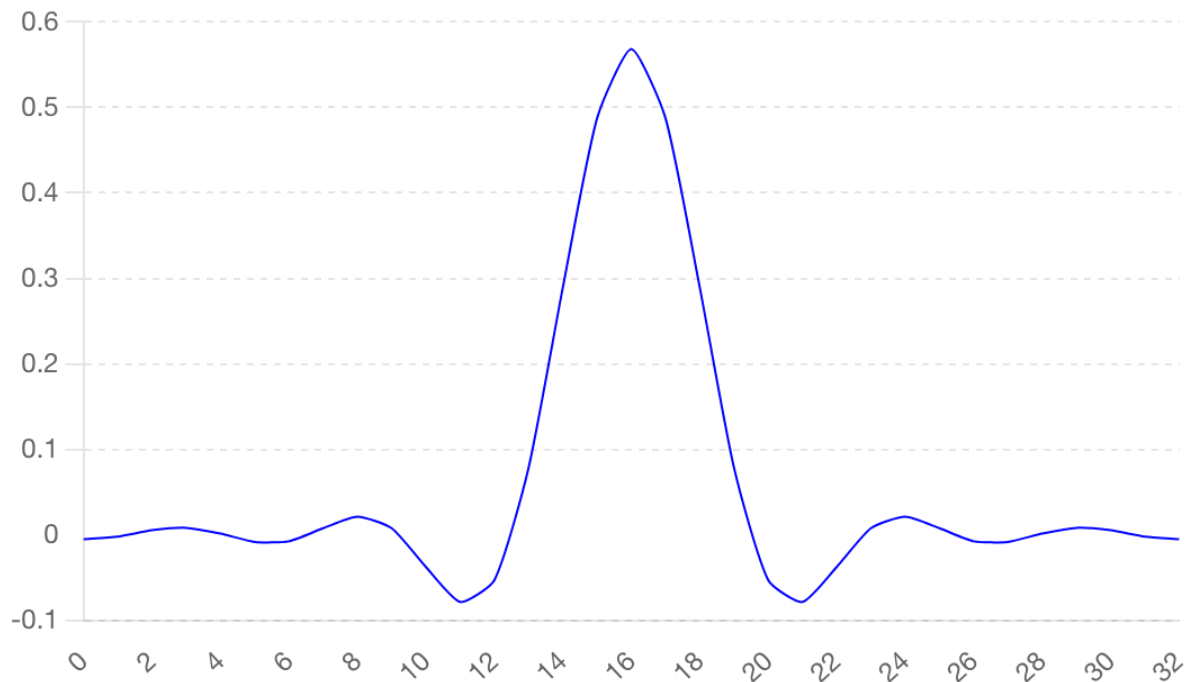


Figure 50: coefficient graph of the raised cosine transmitter

After we made our simulink perfect, we exported our coefficient directly from Matlab.

### 3. Subsystem components and custom blocks

Component	Parameter	Description
Square Root Raised Cosine Filter	Taps: 33, Coefficients: Pre-defined scaled by 32768	Performs matched filtering for signal processing
Delay Line	33-entry register array	Stores intermediate input samples for convolution
Intermediate Result (temp)	48-bit register	Holds convolution result to prevent overflow
Output Signal	16-bit signed register	Scaled output of the convolution, representing the filtered signal

## Channel

We chose these modules and their configuration for the following reasons: The input module receives the signal, which is then processed through the Gilbert model (MATLAB function) to

simulate the channel's effects. This model is effective for simulating burst errors in communication channels and allows us to configure the Signal-to-Noise Ratio (SNR). We have two Additive White Gaussian Noise (AWGN) channels with different SNRs—one with a good SNR of 21dB and another with a Bad SNR of 9dB—to analyze system performance under varying state. The switch module selects the output from one of the AWGN channels based on a condition (greater than 10). Finally, the output module collects the processed signal for further analysis or processing. This configuration allows us to effectively simulate and analyze the communication system's performance under different channel conditions and noise levels.

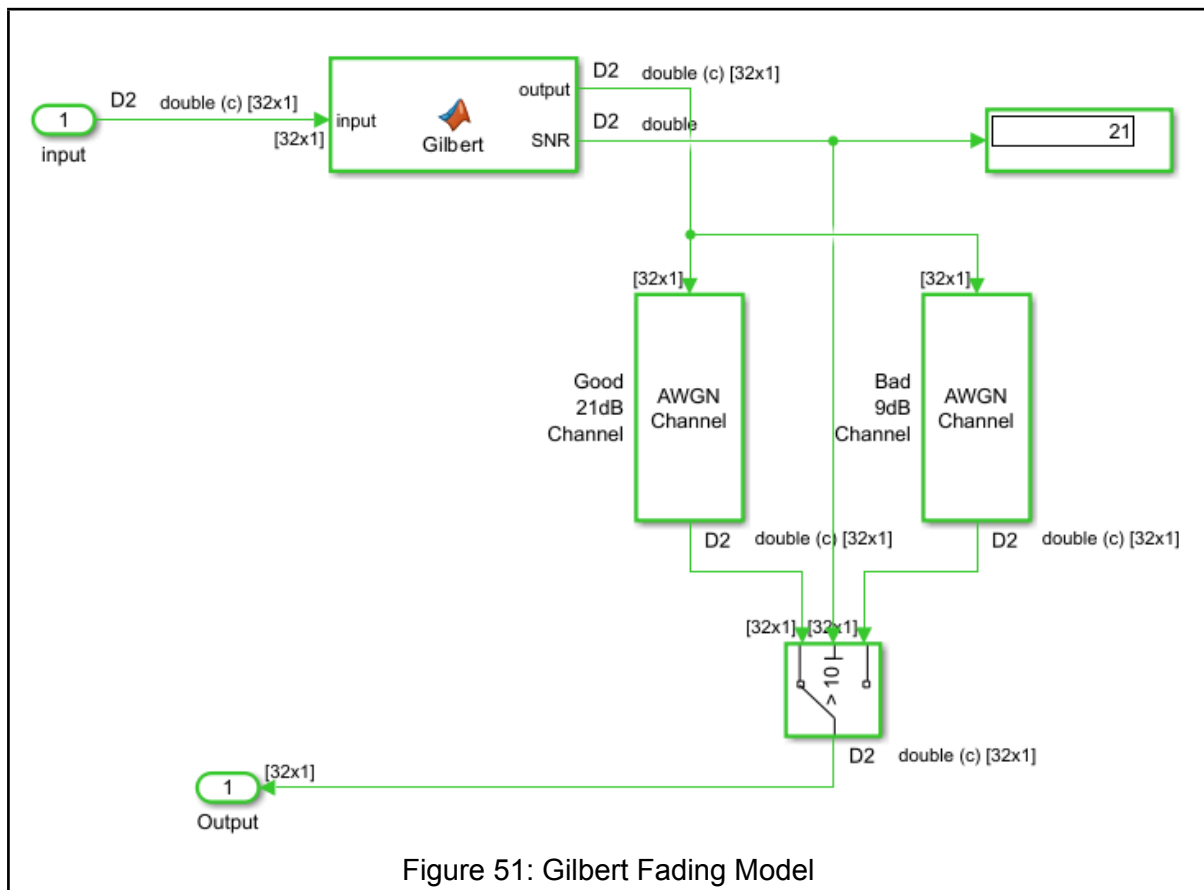


Figure 51: Gilbert Fading Model

Block Parameters: Good 21dB Channel

AWGN Channel  
Add white Gaussian noise to the input signal  
[Source code](#)

Parameters

Mode: Signal to noise ratio (SNR)

SNR (dB): 21

Input signal power, referenced to 1 ohm: 1

Randomization

Random number source: Global stream

Simulate using: Code generation

OK Cancel Help Apply

Figure 52: Good 21dB channel parameters

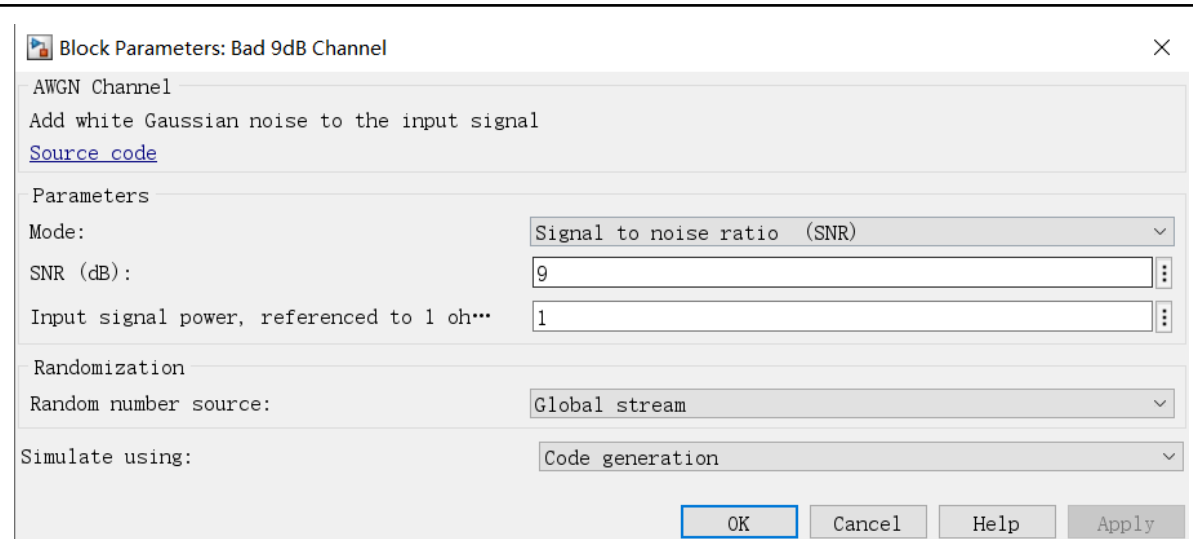


Figure 53: Bad 9dB channel parameters

Note: For the state machine codes in Matlab function module, see Appendix

## FPGA Implementation:

### 1. Block diagram

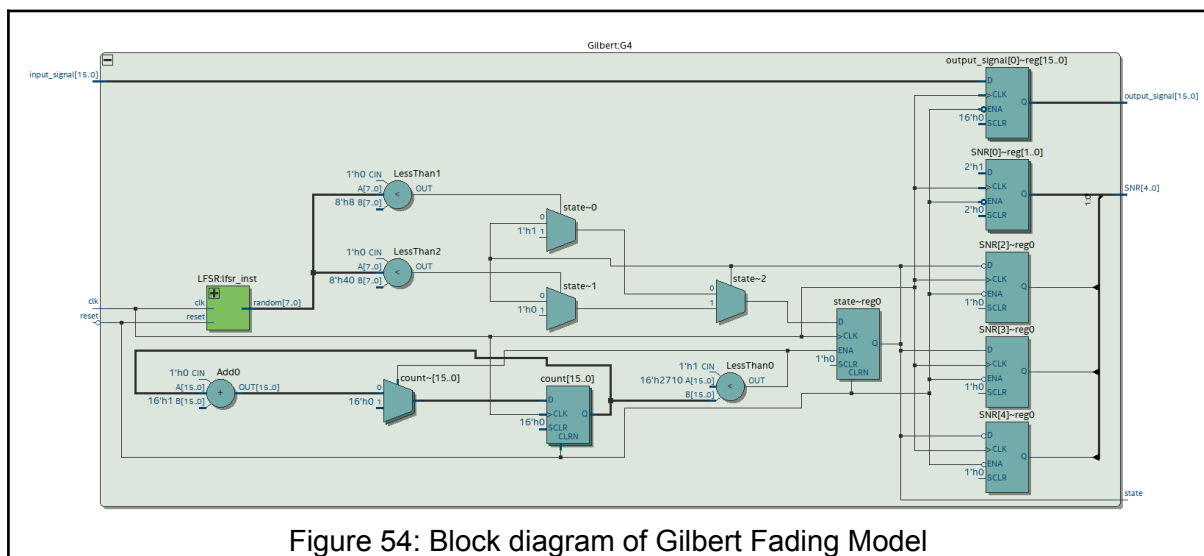


Figure 54: Block diagram of Gilbert Fading Model

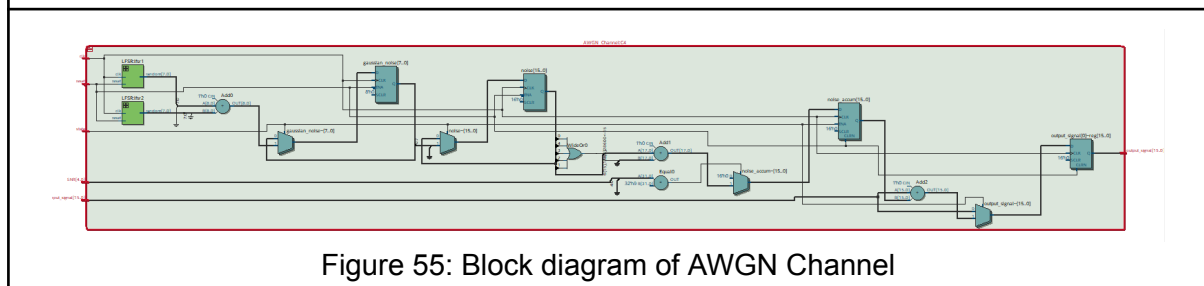


Figure 55: Block diagram of AWGN Channel

### 2. Design Justification

We used LFSR for the noise generation as it is easy to implement it in the FPGA. The LFSR can generate random numbers.

The direct implementation of cosine and sine functions was not suitable for FPGA, so we utilized the Central Limit Theorem to approximate Gaussian noise. This approach involves summing multiple uniform random variables to achieve a distribution that closely mimicked Gaussian noise, ensuring effective and accurate noise simulation.

### 3. Subsystem components and custom blocks

component	parameter	description
AWGN Channel	Input Signal: 16-bit signed, SNR: 5-bit	Adds Gaussian noise to the input signal based on SNR and state
LFSR Modules	8-bit random number generation	Generates random numbers for noise approximation
Gaussian Noise Approximation	Averaged LFSR outputs	Approximates Gaussian noise
Gilbert Channel	State Transition Probabilities, SNR	Simulates burst error channel with state transitions
State Transition Logic	P_GB, P_BG, Ts	Manages state transitions and sets SNR based on state
LFSR Custom Block	Polynomial: $\text{lfsr}[7] \wedge \text{lfsr}[5] \wedge \text{lfsr}[4] \wedge \text{lfsr}[3]$	Generates random numbers for state transitions

## Verification

### Source / Sink

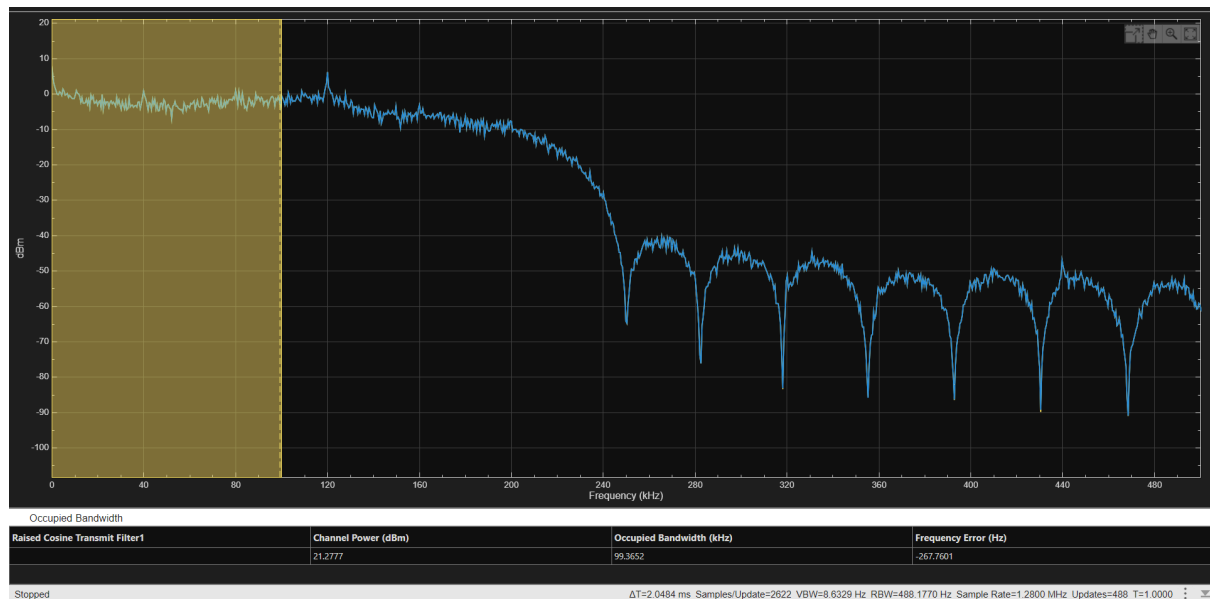
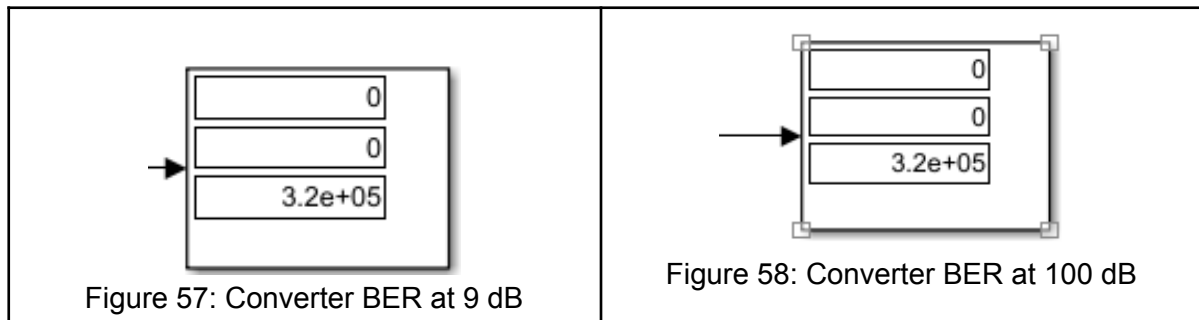


Figure 56: Frequency spectrum of Source (yellow) and Sink (blue) with 100 dB SNR

In the figure above, the frequency spectrum of the source is shown in yellow, while the sink is displayed in blue. At a high signal-to-noise ratio (100 SNR), we could barely see the source spectrum, as they are almost identical. This demonstrates the effectiveness of maintaining a high SNR in preserving the integrity of the transmitted signal.

### A/D and D/A converters

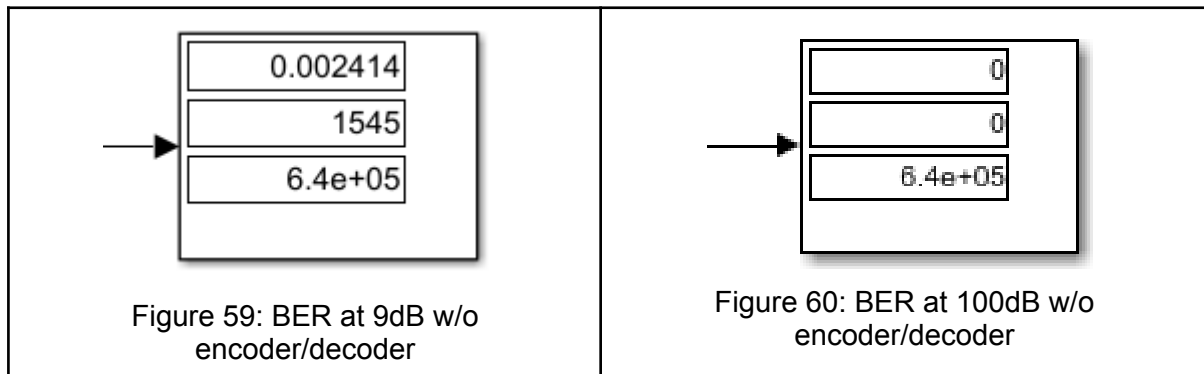
#### Simulink Verification:



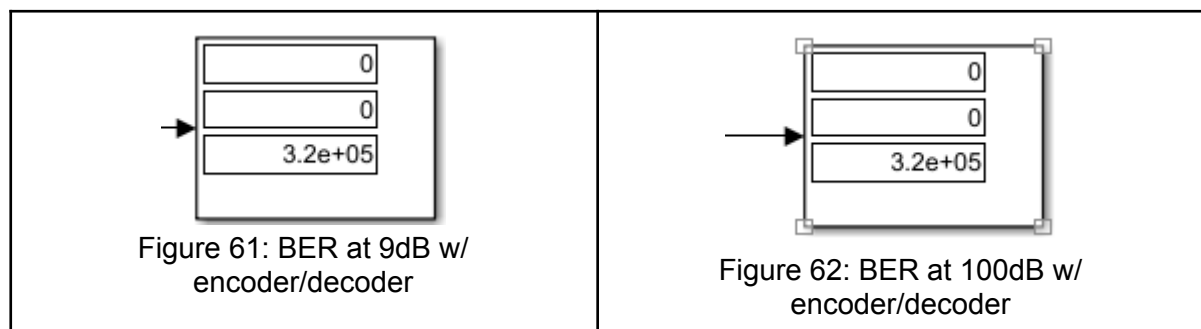
### Error Correction Encoder / Decoder.

#### Simulink Verification:

Without Encoder/Decoder:



With Encoder/Decoder:



From the 4 figures above, the only time the bit error rate is not zero is when SNR is 9dB without encoder and decoder. As a result, the Convolutional encoder and Viterbi decoder in our system are very effective in error correction.

### FPGA Verification:

To test whether the encoder and decoder is working properly, we create a top module that connects the encoder and decoder together. In the test bench for this top module, we send several 16-bit inputs to the encoder and check whether the outputs of the decoder are the same as the corresponding inputs.

Here is the Modelsim waveform we observed when doing this test. We can see that the inputs of the encoder and the outputs of the decoder are exactly the same.

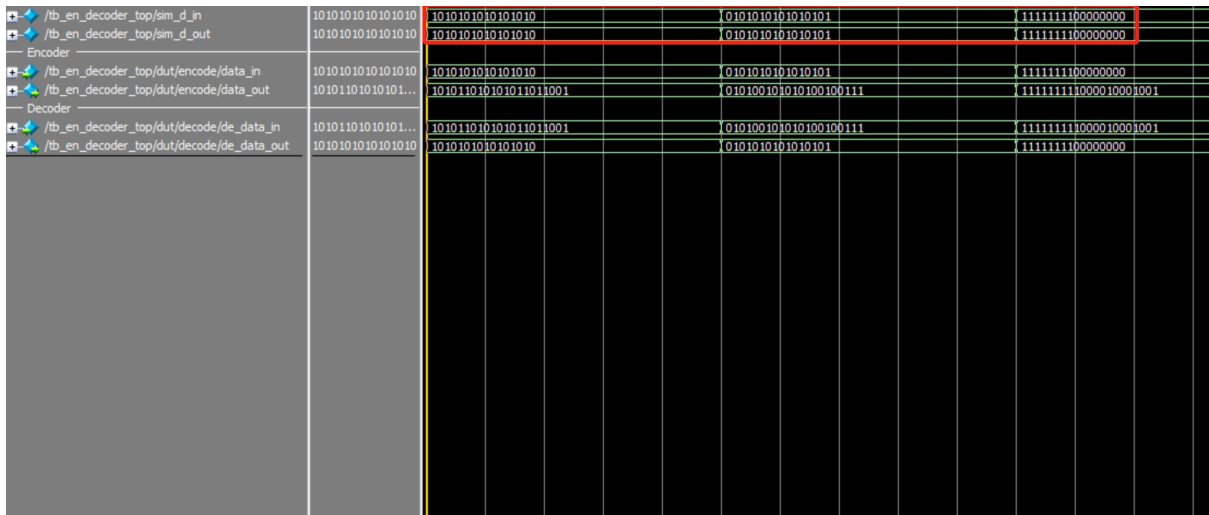


Figure 63: Modelsim Waveform of encoder/decoder testing

Moreover, we instantiate the encoder and decoder in the top level module of our project (modified from the part1 of the FPGA starter assignment). With only the encoder and decoder connected in the system, when speaking to the microphone, we can hear the speaker output clearly. As a result, the encoder and decoder are working properly.

## Modulator / Demodulator

### Simulink Verification:

At 9 dB SNR:

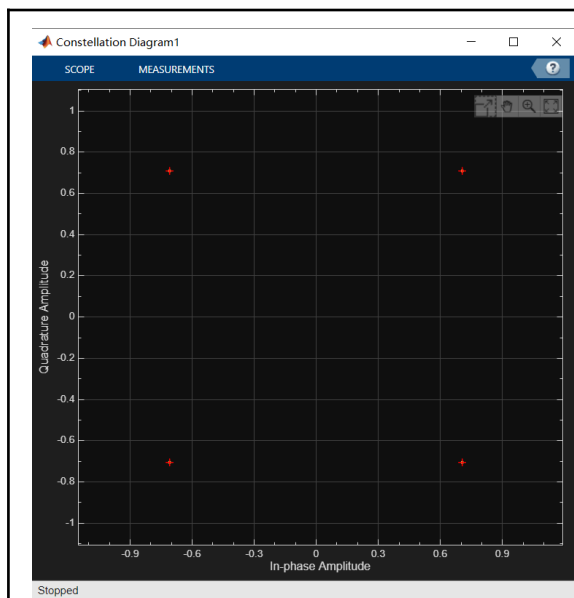


Figure 64: constellation diagram at the output of modulator at 9dB SNR.

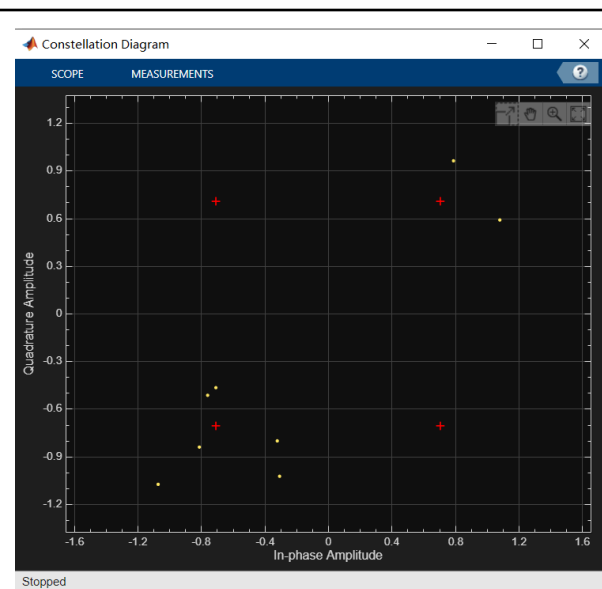


Figure 65: constellation diagram at the input of demodulator at 9dB SNR.

At 100 dB SNR:



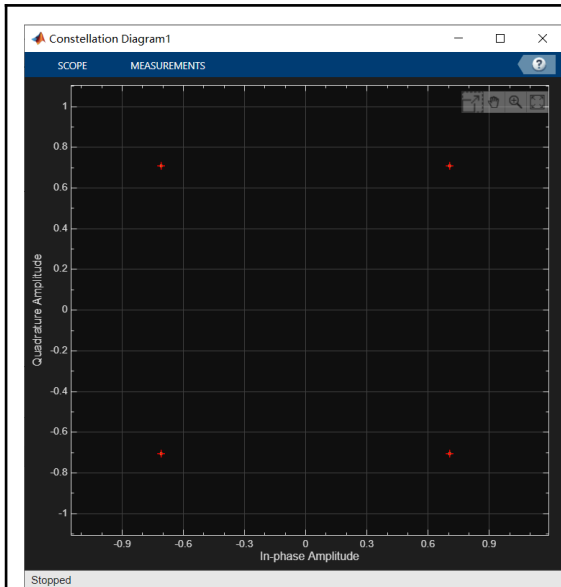


Figure 66: constellation diagram at the output of modulator at 100dB SNR.

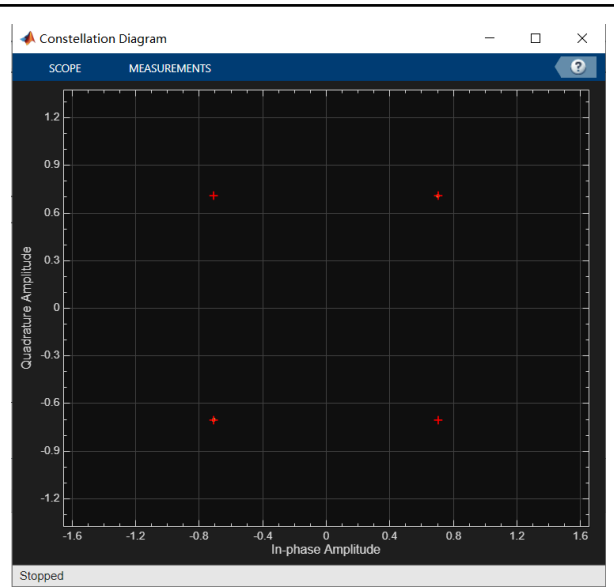


Figure 67: constellation diagram at the input of demodulator at 100dB SNR.

From the above figures, we can clearly see that after the noise is added in the AWGN channel, the mapped symbols show some deviations from the original four points. Also, the scatter is more organized and closer to the four points when the SNR is 100dB. This is expected because the BER when SNR is 100 dB is smaller.

### FPGA Verification:

To test whether the modulator and demodulator are working properly, we first created a top level module that connects the modulator and demodulator together. Then, in the testbench of this top level module, we send several 21-bit inputs and check whether the outputs of the demodulator are the same as the corresponding inputs.

Here is the Modelsim waveform we observed when doing this test. We can see that the inputs of the modulator and the outputs of the demodulator are exactly the same.

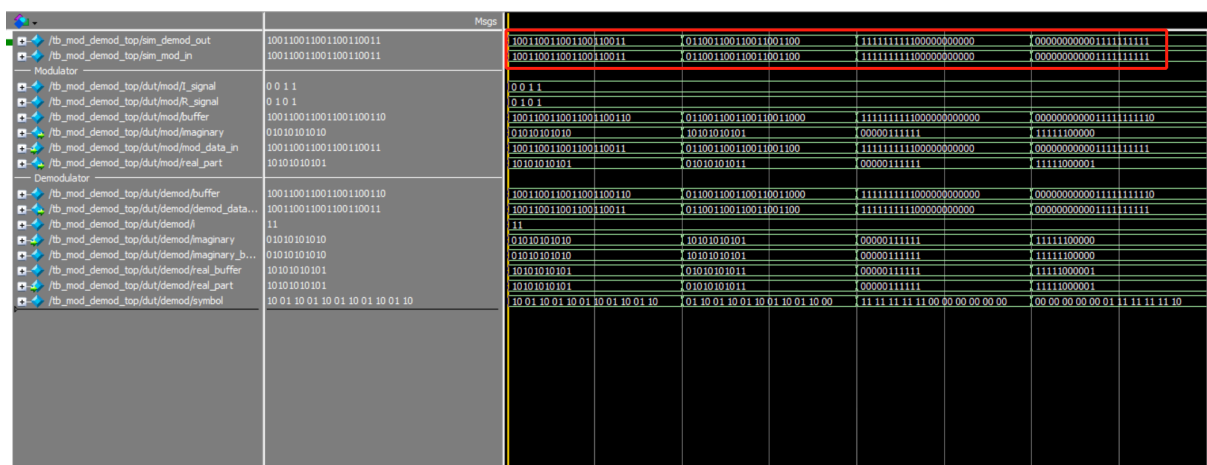


Figure 68: Modelsim waveform of modulator/demodulator testing

Moreover, we instantiated the modulator and demodulator in our top level module. Then, we connect them with the previously verified encoder and decoder to form a subsystem. When we download the system to FPGA and speak to the microphone, we can hear our voice coming out of the speaker clearly, indicating that this subsystem with modulator and demodulator is working properly.

Transmitter / Receiver

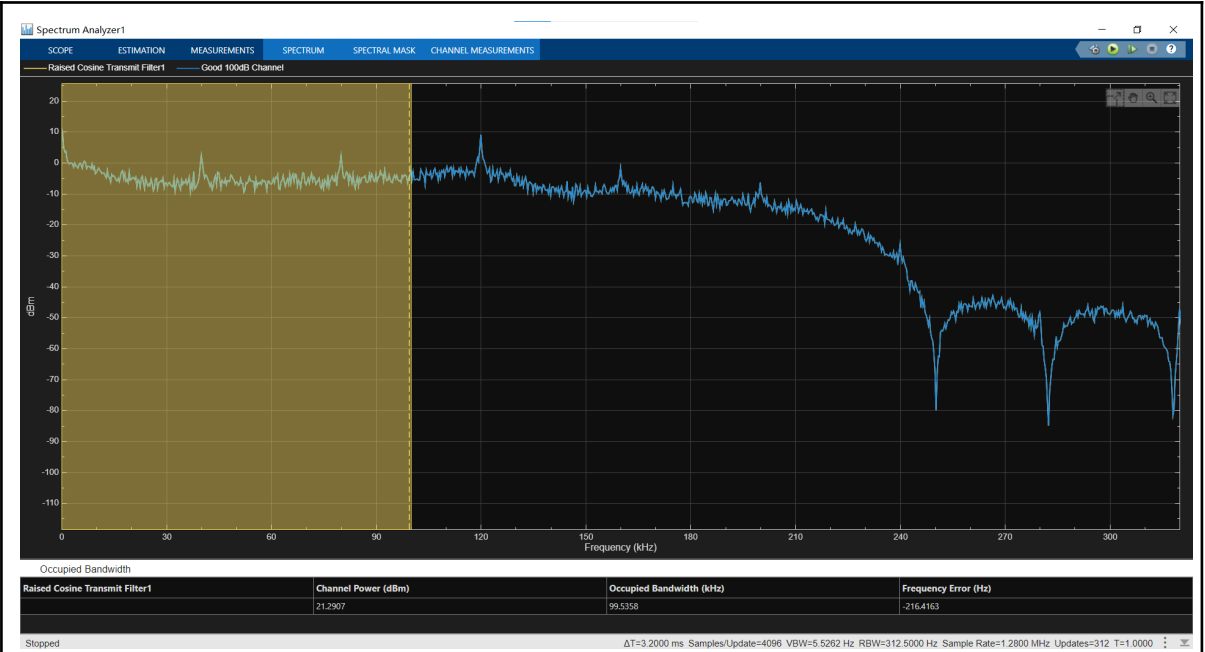
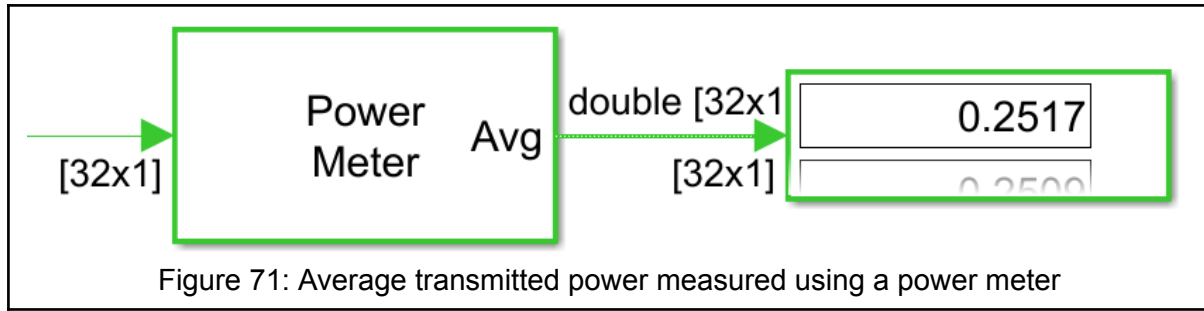


Figure 69: frequency domain transmitter/receiver signal at 100dB

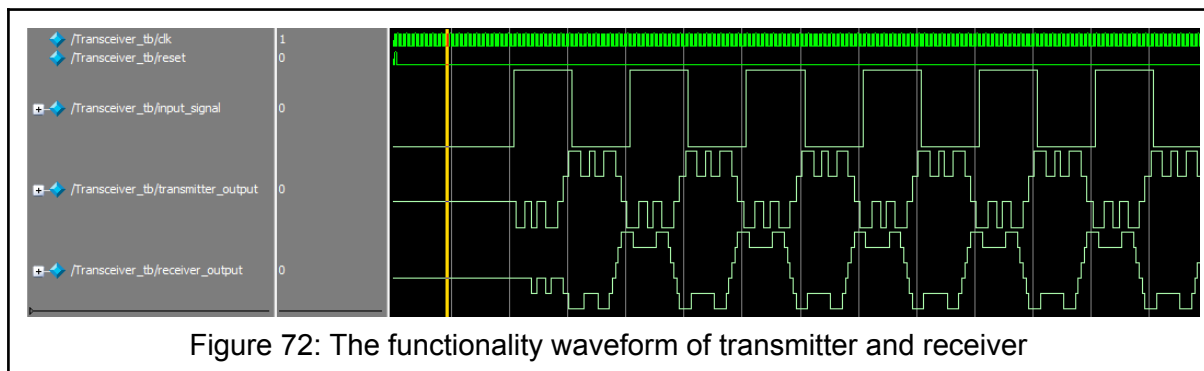


Figure 70: frequency domain transmitter/receiver signal at 9dB



From the spectrum analyzer plots above, we can see that the channel bandwidth is approximately 200 kHz. The transmitter power, measured by power meter, is 0.2517W.

### FPGA Verification:



During the testing phase, when a square wave was used as the input signal, the system was able to maintain the integrity of the signal throughout the transmission process. After passing through the transmitter and receiver modules, the output signal closely resembled the original square wave input. This demonstrates the effectiveness of our design in preserving signal characteristics.

### Channel

The variance of the AWGN channel can be calculated below:

$$10^{(-SNR/10)} = 10^{(-9/10)} = 0.12589254117$$

$$10^{(-SNR/10)} = 10^{(-21/10)} = 0.00794328234$$

The average number of transitions in the Good/Bad states can be calculated below:

Transition probability from Good to Bad,  $P_{GB}=0.03$

Transition probability from Bad to Good,  $P_{BG}=0.25$

Let G be the steady-state probability of being in the Good state.

Let B be the steady-state probability of being in the Bad state.

The steady-state probabilities can be calculated using the balance equations:

$$G \cdot P_{GB} = B \cdot P_{BG}$$

$$G + B = 1$$

solving the equation, we can get  $G = 0.893$ ,  $B = 0.107$

$$\text{Average transitions from Good to Bad} = G \cdot P_{GB} = 0.893 \cdot 0.03 \approx 0.02679$$

$$\text{Average transitions from Bad to Good} = B \cdot P_{BG} = 0.107 \cdot 0.25 \approx 0.02675$$

## FPGA Verification:

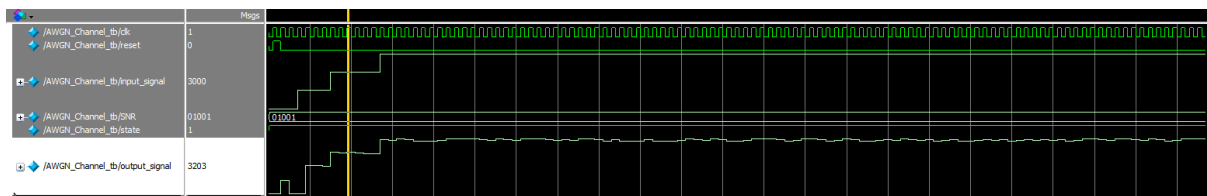


Figure 73: The waveform of AWGN channel

The figure illustrates the waveform output of the AWGN channel. The clock signal (clk) synchronizes operations, while the reset signal initializes the system. The input\_signal, set to 3000, represents the transmitted data. The SNR value (01001) controls the noise level added to the signal. The state signal indicates the channel condition, with 0 for "Good" and 1 for "Bad." The output\_signal, shown as 3203, demonstrates the input signal affected by the AWGN, reflecting the added noise based on the current SNR and state.

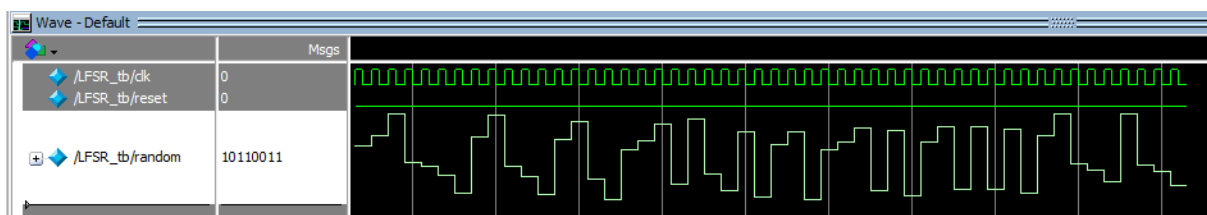


Figure 74: LFSR's functionality to generate random numbers

The figure demonstrates the functionality of the Linear Feedback Shift Register (LFSR) in generating random numbers. The clk signal represents the clock input, synchronizing the operations of the LFSR. The reset signal initializes the LFSR, setting it to a known state. The random signal shows the generated 8-bit random number, which changes with each clock cycle. As seen in the waveform, the LFSR produces a sequence of pseudo-random numbers, effectively providing the randomness required for various simulation and noise generation tasks within the FPGA design.

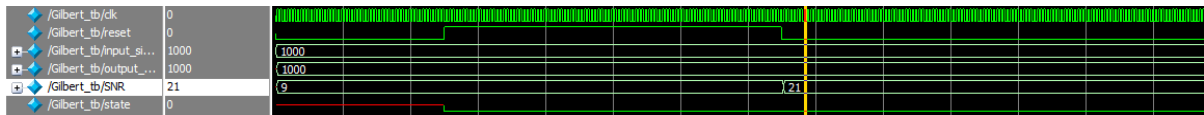


Figure 75: Gilbert fading model. Transit from 9dB to 21dB

The figure illustrates the Gilbert fading model transitioning from 9dB to 21dB. The clk signal represents the clock input, ensuring synchronized operations. The reset signal initializes the system. The input\_signal is set to 1000, representing the signal entering the channel. The output\_signal shows the same value, indicating no initial noise or distortion. The SNR signal transitions from 9dB to 21dB, controlled by the state signal. This change in SNR reflects the transition from a bad state (higher noise, 9dB) to a good state (lower noise, 21dB), demonstrating the model's ability to simulate varying channel conditions effectively.

## Appendix A: Deliverables

Deliverable	File Name	Note (optional)
Product document	ELEC_391_Product_Report	
Product presentation	ELEC 391 Demo 2	Presentation slides for Demo2
Simulink system file	Demo2_model	Modified simulink model based on the feedback received from Demo1.
HDL source code, including testbenches and readme files.	Altera_ ... files bch_encoder bch_decoder qpsk_modulator qpsk_demodulator Transmitter Receiver AWGN_Channel Gilbert LFSR ....	.v or .sv files for the FPGA implementation

## Appendix B

MATLAB code for Gilbert Fading Model:

```
function [output, SNR] = Gilbert(input)

% Scenario F Gilbert Model Parameters

P_GB = 0.03; % Increased transition probability from Good to Bad
P_BG = 0.25; % Decreased transition probability from Bad to Good
f_tr = 10e3; % State transition frequency in Hz
Ts = 1 / f_tr; % Sampling time

persistent state count;

if isempty(state)

    state = 'G'; % Initial state is Good
    count = 0;

end

% Update count
count = count + 1;

% Gilbert model logic with state transition frequency
if count >= Ts

    count = 0; % Reset count after each time step

    if strcmp(state, 'G')

        if rand < P_GB

            state = 'B';

        end

    else

        if rand < P_BG

            state = 'G';

        end

    end

end

% Determine current state
```

```
if strcmp(state, 'G')  
    SNR = 21;  
  
else  
    SNR = 9;  
  
end  
  
% Output the unchanged input signal  
  
output = input;
```

## Team Contributions

### Chloe Sun

#### Individual Contribution:

I was responsible for the design and implementation of the Transmitter and Receiver modules, the Gilbert Fading Model, and the AWGN Channel. I designed the transmitter to effectively modulate the input signal, ensuring minimal signal distortion and maximum data integrity. Additionally, I developed the receiver to accurately demodulate the incoming signal, leveraging matched filtering techniques. I also implemented the Gilbert Fading Model to simulate burst error conditions and the AWGN Channel to introduce Gaussian noise based on SNR levels. My work involved extensive simulation and validation to ensure the robustness of these modules.

#### Team Effectiveness:

From my perspective, our team was highly effective in dividing the workload and ensuring each member focused on their strengths. The collaborative approach allowed for regular feedback and integration sessions, which significantly enhanced the quality and coherence of our final product. However, there were occasional delays in communication that could have been improved with more frequent team meetings and check-ins.

**Other Comments:**

I ensured that my tasks were well-integrated with the group's overall system design by maintaining open communication channels and regularly sharing progress updates. One of the main challenges I faced was accurately simulating noise and fading conditions, which required thorough research and iterative testing. To overcome these hurdles, I consulted relevant literature and used simulation tools extensively to refine the models.

However, there were some practical issues related to the project's workflow. Each team member completed their parts at different times, and we hadn't set a common deadline for everyone. As a result, after finishing my tasks, I often had to wait for other team members to complete theirs. This waiting period was somewhat unproductive as I wasn't sure how to contribute further until their parts were done. In retrospect, establishing clear milestones and synchronized deadlines could have mitigated this issue and ensured a more efficient workflow.

Despite these challenges, through this project, I gained a deeper understanding of FPGA-based communication systems and improved my skills in digital signal processing. The hands-on experience with noise simulation and integration of various modules provided invaluable insights into practical implementation challenges and solutions.

**Pengyu Ji**

**Individual Contribution:**

I designed, implemented and tested the encoder/decoder and modulator/demodulator; I helped Yuqian to design the top level module and debug the whole system; I wrote the sections about my designs in the product report.

**Team Effectiveness:**



Our team worked well together, leveraging each other's strengths to tackle complex problems. While we managed to meet deadlines effectively, there were moments when overlapping schedules caused delays. Regular scrum meetings and shared documentation tools helped mitigate these issues and ensured smooth integration of different subsystems.

### **Other Comments:**

I collaborated closely with my teammates to ensure seamless integration of the error correction and modulation modules with the rest of the system. As the only member who had taken the CPEN311 course, I provided significant assistance in using ModelSim and Quartus software, helping my teammates navigate these tools effectively. The main challenges I encountered were in optimizing the modulation techniques to balance between data rate and noise resilience, which I overcame through detailed simulations and iterative refinement. Additionally, I assisted in debugging and fine-tuning the FPGA implementation, ensuring that our designs met the project requirements. This hands-on experience enhanced my understanding of error correction techniques and modulation strategies, and I appreciated the opportunity to apply theoretical knowledge to practical scenarios.

### **Yuqian Song**

#### **Individual Contribution:**

I was responsible for the A/D and D/A conversion modules and the overall integration of all subsystems. I designed the analog-to-digital and digital-to-analog converters to accurately translate between analog signals and digital data, ensuring minimal signal loss. Additionally, I played a pivotal role in integrating the various subsystems, ensuring that all components worked harmoniously to achieve the project's objectives.

Furthermore, I rebuilt the entire Simulink model based on the feedback from Demo 1, enabling the system to function effectively in our scenario and laying a strong foundation for

our team's FPGA design and validation. The coefficients for the transmitter data were generated from the Simulink model that I developed.

For the FPGA implementation, I introduced the PLL IP core and clock divider, which generated the required clock frequency, serving as the sampling rate for both the input data and the channel in our FPGA system.

### **Team Effectiveness:**

Our team worked well together, and we were good at combining different parts of the project. We divided tasks based on each person's skills, which helped us work efficiently. However, if we had planned better and defined our roles more clearly from the start, we could have avoided some repeated efforts and made the process smoother.

### **Other Comments:**

I ensured that my conversion modules were compatible with the overall system by conducting rigorous tests and adjustments. The main challenges I faced were related to signal synchronization and data integrity during the integration phase, which I addressed through detailed debugging and iterative testing. My experience on this project deepened my understanding of system integration and the critical role of A/D and D/A conversion in communication systems. I valued the collaborative spirit of our team and the practical application of theoretical knowledge.