ELEC 391 Project Report

Reliable Transfer of Wireless Information

Group 8

Yifeng Liu (28177384):

A / D and D / A Conversion, AWGN Channel

Lynn Kelly (14711071):

Error Encoding / Decoding, Modulation / Demodulation

Sidharth Sudhir (11255635):

Transmitter / Receiver, Gilbert Fading Model

Table of Contents

Objectives, Requirements & Constraints

The objective of our team was to implement the reliable transfer of wireless information over a noisy channel that meets the following communication scenario specifications:

Message Transmission	Reliable Transmissio n	Processing Delay (mic to speaker)	Simulation tool	Prototype
8 kHz bandwidth	Bit error rate of 10^{-4}	25 ms	Matlab / Simulink / ModelSim	Altera DE1-SoC FPGA board

Average transmit power	Channel bandwidth	Audio source	Audio sink
1 W	100 kHz spectral mask	• Audio file in a WAV format with 32 bits per sample • DE1 microphone input • Duration 5 sec – 10 sec	• Audio file in WAV format based on team requirements • DE1 speaker output

Channel Sample Frequency	Channel quantization / dimension	Channel amplitude range	Channel quantization interval	Gilbert channel model Name
1 MHz	16 bits	±4	2^{-13}	γ

We were fully able to meet all of the required specifications in our Simulink model, and for the specifications that we were able to measure on the FPGA implementation, we met those specifications as well. For the Simulink modeling, the most significant challenge was reducing the channel bandwidth usage down to the required 100kHz. We tried to modify various parts of our design to reduce the bandwidth including changing our encoder from a ½ rate code to a ⅔ code, and increasing the bits per symbol for modulation, however both of these changes caused our BER to increase above our required specification. We eventually decided to modify our ADC quantization and sampling rate to reduce the initial bit rate, which consequently reduced the symbol rate. This brought our bandwidth usage down to 100kHz. During the implementation phase we faced several significant challenges including how to read audio input from a file and how to output an audio signal to a file. The implementation of the decoder also proved to be more difficult than expected. We had initially intended to use the decoder from the IP library, but implementing this was more complicated than expected so we had to find an alternative solution. Due to resource constraints of the Altera DE1-SoC FPGA board, we also found that it would be difficult to send our signal in frames as originally intended so we had to design a serializer and deserializer to serialize our data. Lastly, one of the greatest challenges of the project was the FPGA implementation of the receiver and transmitter as this required a complex code and a significant amount of debugging and timing analysis.

System Design

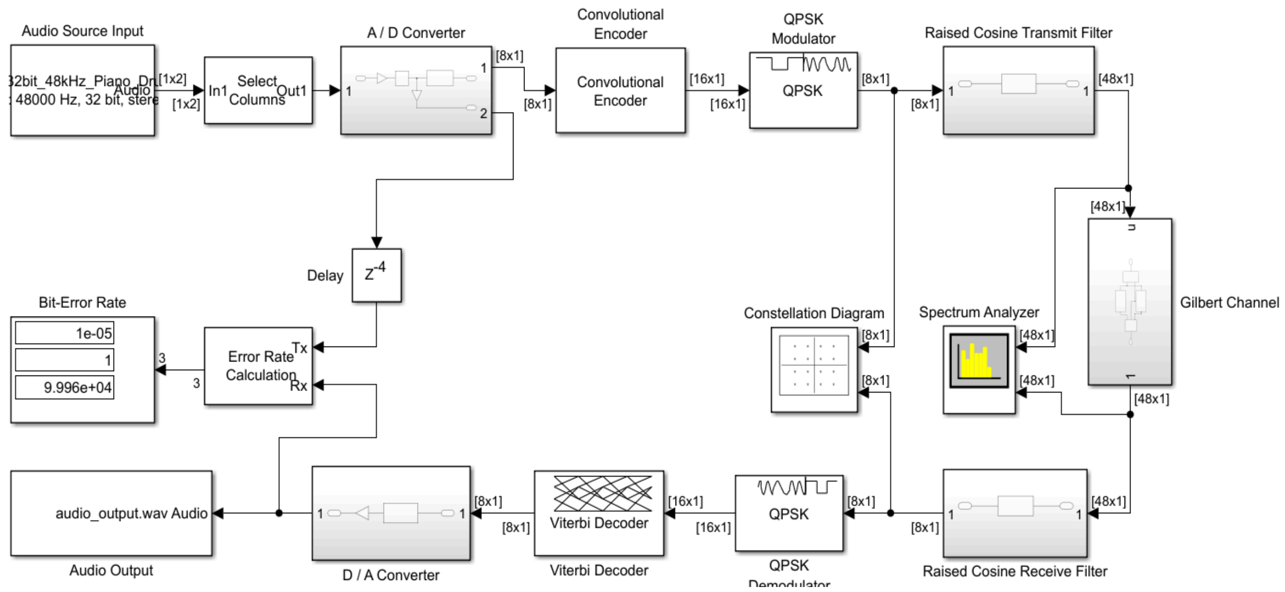


Figure 1: System block diagram

Our system consists of the subsystem blocks shown in the figure above. Our system is able to receive as input either a 32-bit WAV file to produce a 16-bit output WAV file in the Simulink model. For the FPGA implementation, our system is able to either use the 16 bits .mif file stored in memory as an input and store it to another mif file as output, or it is able to receive 24-bit microphone input and output audio through attached speakers. The ADC subsystem converts the signal from analog to digital. The Nyquist rate was used for the 8kHz message bandwidth to determine the minimum sampling rate, which is 16kHz (double the message bandwidth). Using this constraint and the requirement to keep the channel bandwidth within 100kHz, the source was quantized to 8-bit frames at a sampling rate of 20kHz. On the output side, the DAC block reverses this process to convert the digital signal back to an analog signal of the same form as the input signal. The next subsystem pair consists of a convolutional encoder and Viterbi decoder for error correction. The convolutional code chosen has a ½ code rate with robust error correction abilities for a moderately noisy channel. For the modulation / demodulation subsystem, QPSK was chosen because it has a low theoretical BER. The symbol rate of our system was calculated to be 160kHz, which sets the minimum channel bandwidth usage at 80kHz. The transmitter/ receiver subsystem uses a raised cosine filter for pulse shaping to transmit and receive the signal. MATLAB-derived raised cosine coefficients were generated to keep the transmission power at 1W, and a roll-off factor of 0.25 was used which causes our channel bandwidth usage to be around 100kHz as seen in our Simulink model. The channel uses the γ Gilbert channel model to model a noisy AWGN channel. The block was designed to switch between a good and bad channel using Markov chain probabilities of state switching.

The results of our system simulation and FPGA results compared to our specifications are shown in the following table:

Project Conditions	Specifications	Simulation Results	FPGA Results
Message Transmission	8 kHz bandwidth	8 kHz bandwidth	8 kHz bandwidth
Reliable Transmission	BER of 10^{-4}	BER of 0 to 5×10^{-5}	Unknown*
Processing Delay	25 ms	0.20 ms	0.29 ms
Average Transmit Power	1W	1W	1W
Channel Bandwidth	100 kHz	100 kHz	100 kHz

**Since our AWGN channel required FPGA memory blocks for implementation, we were unable to incorporate the AWGN channel into the testbench to test the BER before the project deadline. We did implement a switch in the FPGA to switch between the Gilbert fading model channel and a noise-free channel and we were unable to hear a difference between the two states so we suspect our BER remains very low.*

Subsystem Design

1 Source / Sink

Block Diagram (Simulink):

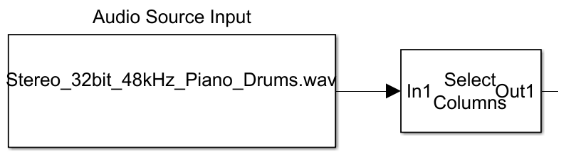


Figure 2: Block diagram for the source

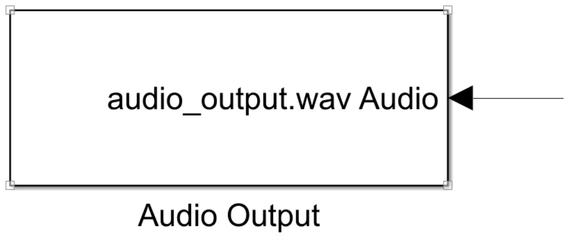


Figure 3: Block diagram for the sink

Description and justification (Simulink):

The *From Multimedia File* is chosen as the source of our system as shown in figure. The output type is set to int16 with 1 sample per channel. A 32-bit stereo .WAV file is chosen as the source. Due to the stereo audio of the 32-bit source, the variable selector block is added to select one channel out of the source. For the sink, the *To Multimedia File* is implemented with the output data type of 16-bit integer to get better clarity.

Simulink Blocks & Parameters:

Simulink Block	Parameters Used
From Multimedia File	**Number of times to play file**: 1 **Read range**: [1 inf] **Samples per audio channel**: 1
Variable Selector	**Select:** Columns **Selector mode:** Fixed **Elements:** [1]
Data Type Converter	**Output data type:** double
To Multimedia File	**File type:** WAV **Audio data type:** 16-bit integer

Block diagram (FPGA):

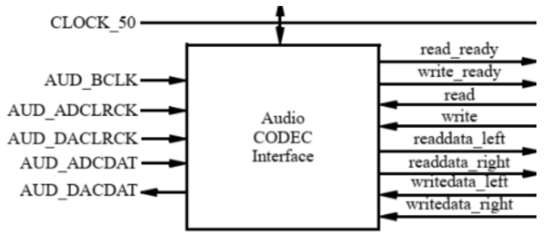


Figure 4: Block diagram of Audio Codec

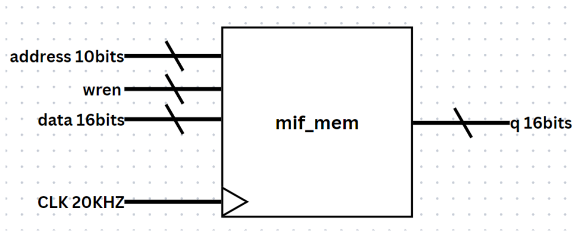


Figure 5: Block diagram of RAM

Description and Justification (FPGA):

The microphone is implemented as one type of input. It comes with the audio codec with 24 bits data input. The other type of input is memory input. The wav file is preloaded into the RAM block as the mif file form.

Subsystem Components (FPGA):

Blocks	**Parameters**	
audio_codec	**Inputs**	**Outputs**

	Input clk;
	Input reset;
	Input read;
	Input write;
	Input [23:0] writedata_left;
	Input [23:0] writedata_right;
	Input AUD_ADCDAT;
	Input AUD_BCLK;
	Input AUD_ADCLRCK;
	Input AUD_DACLRCK;
		output read_ready, write_ready;
		output [23:0] readdata_left;
		output [23:0] readdata_right;
		output AUD_DACDAT;

mono_mem	**Inputs**	**Outputs**

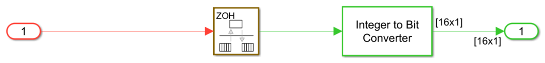
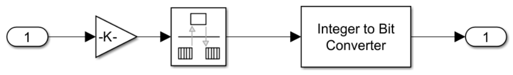
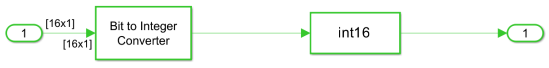
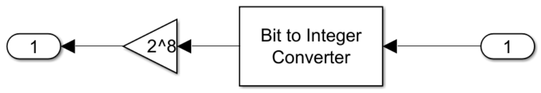
	Input [9:0] address;
	Input clock;
	Input [15:0] data;
	Input wren;
		output [15:0] q;

save_mem	**Inputs**	**Outputs**

	Input [9:0] address;
	Input clock;
	Input [15:0] data;
	Input wren;
		output [15:0] q;

2 A/D and D/A Converters

Block Diagram (Simulink):

 1 ZOH Integer to Bit Converter [16x1] [16x1] 1 *Figure 6: Old ADC module with 16 bits transformation*	 *Figure 7: New ADC module with 8 bits transformation*
 1 [16x1] [16x1] Bit to Integer Converter int16 1 *Figure 8: Old DAC module with 16 bits transformation*	 *Figure 9: New DAC module with 8 bits transformation*

Description and justification (Simulink):

For the A/D converter, the *Rate Transition* and *Integer to Bit Converter* are used to convert the signal from analog to digital. According to the Nyquist rate, the sampling rate should be at least two times the bandwidth. The target message bandwidth for our system is 8kHz. Therefore, the Output port sample time is required to be set to at least 1/160000 to avoid aliasing. The old system is set with a sampling time of 1/16000, and the *Integer to Bit Converter* is set to 16 bits per integer to produce a 16-bit quantization per sample. However, in the subsequent verification of the frequency spectrum, the resulting symbol rate of 160kHz resulted in a channel bandwidth of over 160kHz, which exceeded our requirement of a 100kHz spectral mask. Therefore, the decision was made to adjust the Output port sample time to 1/20000, and change the quantization per sample to 8 bits. A *multiply* is added to avoid the overflow of int16 input. The gain is set to 1/256 which is 2^8.

Due to the reduced bit rate output from the ADC, the symbol rate is reduced, which is shown in a lower bandwidth usage of 100kHz which matches our requirements. However, this adjustment directly results in our transformed signal containing less information which reduces the fidelity of our output audio signal.

To convert it back to the analog signal, *Bit to Integer Converter* and *Data Type Converter* are used to convert the signal from digital to analog. The number of bits per integer is also adjusted to 8 to match the setting in the A/D converter instead of 16 bits per integer in the previous module. The *multiply* is added to restore data length to 16 bits. The *Data Type Converter* is implemented to ensure the correctness of the data type.

Simulink Blocks & Parameters:

Simulink Block	Parameters Used
Rate Transitions	**Initial conditions:** 0 **Output port sample time options:** Specify **Output port sample time:** 1/20000
Integer to Bit Converter	**Number of bits per integer(M):** 8 **Treat input values as:** Signed **Output bit order:** MSB first **Output data type:** Inherit via internal rule
Bit to Integer Converter	**Number of bits per integer(M):** 8 **Input bit order:** MSB first **After bit packing, treat resulting integer values as:** Signed **Output data type:** Inherit via internal rule
Multiply (ADC)	**Gain:** 1/256 **Multiplication:** Element-wise (K*u)
Multiply (DAC)	**Gain:** 256 **Multiplication:** Element-wise (K*u)
Data Type Converter	**Output data type:** int16

Block diagram (FPGA):

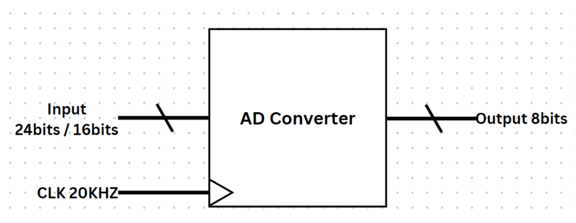


Figure 10: Block diagram of AD converter

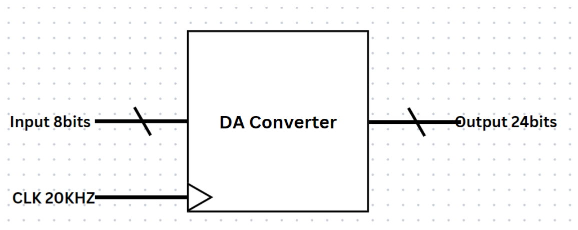


Figure 11: Block diagram of DA converter

Description and Justification (FPGA):

For the mic input, due to the 24-bit input of Audio Codec, we have shifted from 24 bits to 8 bits to match up the channel input. A 20KHz clock is used to implement the sample frequency of 20KHz.
The other input source is the .mif file which has a 16 bit input source. Therefore, we have shifted from 16 bits to 8 bits to match the design requirement. A 20 kHz clock is also implemented.

Subsystem Component (FPGA):

Blocks	Parameters	
data_shifter_sampling	**Input**	**Output**

	input wire clk;
	input wire signed [23:0] data_in;
	input enn;
		output reg [7:0] data_out;
		output reg ser_en;

data_resampling	**Input**	**Output**

	input wire clk;
	input wire signed [7:0] data_in;
	input enn;
		output reg [23:0] data_out;
		output reg ser_en;

data_shifter_sampling_wav	**Input**	**Output**

	input wire clk;
	input wire signed [15:0] data_in;
	input enn;
		output reg [7:0] data_out;
		output reg ser_en;

data_resampling_wav	**Input**	**Output**

	input wire clk;
	input wire signed [7:0] data_in;
	input enn;
		output reg [15:0] data_out;
		output reg ser_en;

3 Error Correction Encoder / Decoder

Block Diagram (Simulink):

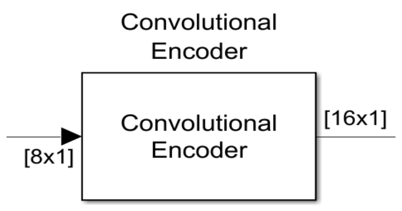


Figure 12: Block Diagram Convolutional Encoder

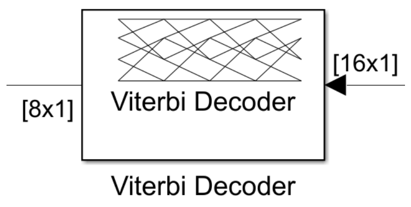


Figure 13: Block Diagram Viterbi Decoder

Description and justification (Simulink):

The error correction encoder and decoder are an integral part of our system. We decided to use a convolutional error correction encoder with a Viterbi decoder due to its robust error correction abilities and flexibility in being able to choose from a variety of constraint lengths, traceback depths, and code rates. During our initial design, we had chosen to use a ½ rate (7, [171, 133]) convolutional encoder with a Viterbi decoder with traceback depth of 32. Using this encoder/decoder combination, we were consistently getting a BER of 0 in our Simulink model even with a constant channel SNR of 9. However, upon implementation on the FPGA, we found that with a constraint length of 7 and traceback of 32 for the Viterbi decoder, a large amount of FPGA resources were required due to its complexity, which we worried could cause resource usage issues in the compilation of our entire system. Therefore, we decided to scale back the constraint length and traceback depth to a ½ rate (4, [12, 15]) convolutional encoder with a traceback depth of 16. This reduced the amount of resources needed for the Viterbi decoder as well as reduced the overall delay of the system from 32 cycles to 16 cycles. The main tradeoff of this decision is that the error correction abilities of a lower constraint convolutional code with lower traceback depth is reduced. Therefore, especially given the presence of burst errors in the Gilbert fading model, our Simulink model sometimes had a BER >0. However the BER still remained very low at less than $5\text{x}10^{-5}$, which remains under our required specification of a BER of less than 10^{-4}. Additionally, the error correcting abilities of many ½ rate convolutional codes in combination with QPSK modulation, which was used in our system, have been shown to result in a low theoretical BER in moderately noisy AWGN channels as shown in *Appendix B. Figure 1* [2]. Although the chosen (4, [12 15]) convolutional code is not included in this graph, given the pattern of curves for slightly higher constraint length codes it is likely that a constraint length of 4 follows a similar pattern to the above curves and results in a similarly low BER for a SNR of 9 or greater.

Simulink Blocks & Parameters:

Simulink Block	Parameters Used
Convolutional Encoder	**Trellis Structure:** poly2trellis(4, [12 15]) **Operation mode:** Continuous
Viterbi Decoder	**Trellis Structure:** poly2trellis(4, [12 15]) **Decision type:** Hard decision **Traceback Depth:** 16 **Operation mode:** Continuous **State metric word length:** 16 **Output data type:** Inherit via internal rule

Block diagram (FPGA):

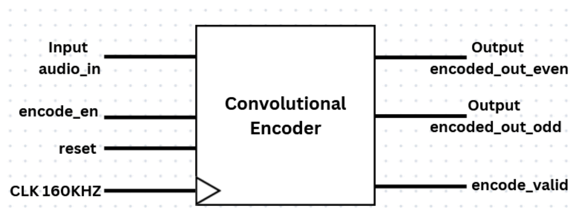


Figure 14: Block Diagram Convolutional Encoder

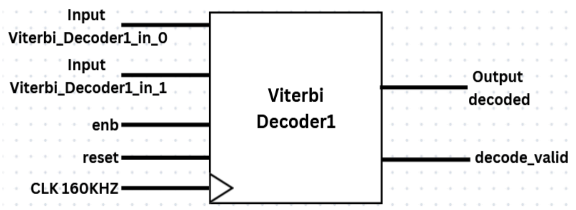


Figure 15: Block Diagram Viterbi Decoder

Description and Justification (FPGA):

Modules for a (4, [12 15]) convolutional encoder and Viterbi Decoder with traceback depth of 16 were created in Verilog and SystemVerilog. Although the Simulink model used frames of 8 bits, we found that a large amount of FPGA resources were required to generate multiple instantiations of each subsystem module for parallel processing. This significantly slowed our compilation time and we were worried we would not have enough resources to include all parts of our system so we decided to serialize the data for all subsystems from the encoder / decoder onwards. A serializer, deserializer and delay module to control the timing of deserialization was created so each module thereafter took inputs and outputs of single bits. The convolutional encoder uses shift registers to sequentially shift the input bits through 3 registers, and the incoming bit and the bit values of the output wires of the registers are mod2 summed based on the generator polynomials 12_octal and 15_octal to produce the two output bits. The output bits were separated into an odd-position bit stream and even-position bit-stream. The output bits were separated since they can be inputted into the modulator and processed separately into the real and imaginary parts of the symbol. The Viterbi decoder proved more difficult than anticipated to implement. Initially we had intended to use the Viterbi decoder in the Quartus IP library, however determining how to configure and use the enable signals was significantly more difficult than anticipated. After several failed attempts to use the Viterbi IP block, we decided to instead use a Viterbi decoder generated from the HDL Coder using our Simulink model. As mentioned, we had originally generated a (7, [171 133]) Viterbi Decoder, however the complexity of the generated files used up a lot of FPGA resources, so we scaled down to generating the (4, [12 15]) decoder with traceback depth of 16 from the HDL Coder. The HDL Coder generated code did add an additional 13 cycles of delay above the 16 cycles specified in our Simulink model, which is suspected to be due to additional processing for practical implementation. However our overall processing delay remained significantly under our maximum allowable delay.

Subsystem Component (FPGA):

Blocks	**Parameters**	
convolutional_encoder	**Inputs**	**Outputs**

	input clk;
	input reset;
		output encoded_out_odd;
		output encoded_out_even;

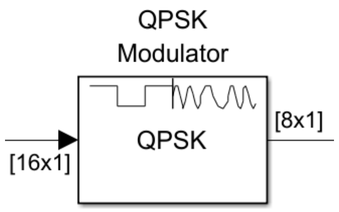
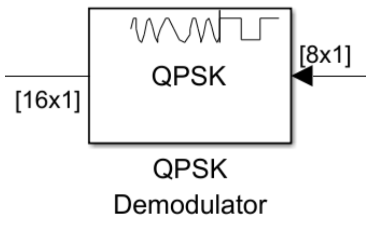
	input encode_en;
	input audio_in;
		output encode_valid;

Viterbi_Decoder1 **note: there are multiple submodules including Traceback modules, ACS modules and BranchMetric. A full list is included in Appendix A*	**Inputs**	**Outputs**

	input clk;
	input reset;
	input enb;
	input Viterbi_Decoder1_in_0;
	input Viterbi_Decoder1_in_1;
		output decoded;
		output reg decode_valid;

4 Modulation / Demodulation

Block Diagram (Simulink):

 *Figure 16: Block Diagram for Modulator*	 *Figure 17: Block Diagram for Demodulator*

Description and justification (Simulink):

QPSK modulation and demodulation was chosen for our system because it has a low theoretical BER. In particular it has the same BER as that of BPSK when Gray coding is used with a phase offset of pi/4 since this can be implemented as two BPSK signals orthogonally oriented to one another [3]. The trade-off of using QPSK is that it has a relatively low bit to symbol ratio of 2 bits/symbol, so the symbol rate is higher than that of a higher order modulation scheme. Thus the bandwidth usage is greater than that of a higher order modulation scheme. However, QPSK is still sufficient to offset the ½ rate of the error correction code, which causes the system to have a symbol rate that is equal to the sampling rate set by the ADC. When we were attempting to reduce our channel bandwidth usage to meet our spectral mask requirements of 100kHz, we tried higher order modulation schemes such as 8-PSK and 16-PSK, however these notably increased our BER to a level that was close or higher than our BER limit of 10^{-4}, and so we decided to continue with QPSK given it's low theoretical BER and opted to reduce the quantization and sampling frequency at the ADC instead.

Simulink Blocks & Parameters:

Simulink Block	Parameters Used
QPSK Modulator Baseband	**Input type:** Bit **Constellation ordering:** Gray **Phase offset (rad):** pi/4 **Output data type:** double
QPSK Demodulator Baseband	**Output type:** Bit **Decision type:** Hard decision **Constellation ordering:** Gray **Phase offset (rad):** pi/4 **Output data type:** Inherit via internal rule **Derotate factor:** Same word length as input **Rounding Mode:** Nearest **Overflow mode:** Saturate

Block diagram (FPGA):

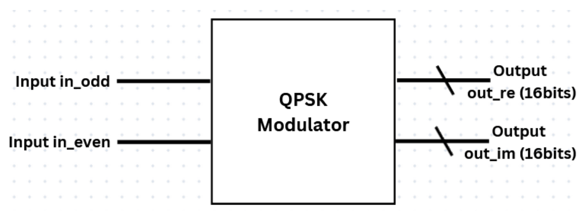


Figure 18: Block Diagram of Modulator

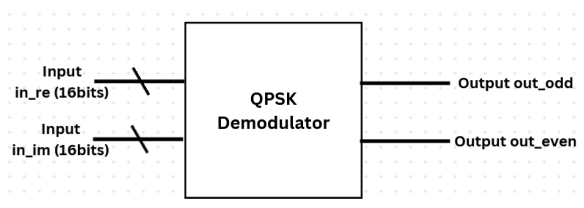


Figure 19: Block Diagram of Demodulator

Description and Justification (FPGA):

The QPSK modulator and demodulator modules took inspiration from the HDL Coder generated code's logic of using look-up tables to map 2-bit inputs to a predefined symbol. The inputs of the modulator are 2 bits, one from the odd-bit stream and one from the even-bit stream. Using a lookup table, the combination of these two bits are mapped to one of four Grey coded QPSK symbols at angles of pi/4, 3pi/4, 5pi/4 and 7pi/4 on the unit circle. The real and imaginary magnitudes of these symbols are both sqrt(2)/2, which is represented by Q2.13 fixed signed 16-bit representation. This data type was chosen to match the expected data type for the signal through the channel. The modulator outputs two 16-bit outputs: the representation of the real part of the symbol, and the imaginary part of the symbol. The demodulator reverses this process and takes in two inputs: the 16-bit real and imaginary parts of the symbol. Given the values are expected to have shifted due to channel transmission and noise, these values must be mapped back to the most likely symbol. This is done by comparing each input as being less than, greater than, or equal to zero and combining the real and imaginary parts to determine the most likely symbol in a lookup table. The symbol is then used to recover the appropriate bit-pair, which are the two output bits.

Subsystem Component (FPGA):

Blocks	**Parameters**	
qpsk_modulator	**Input**	**Output**

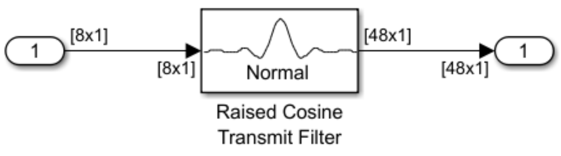
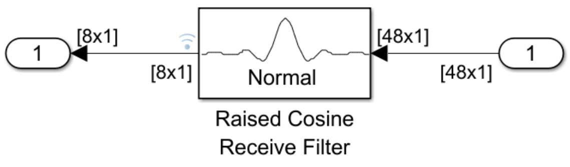
	input logic in_odd;
	input logic in_even;
		output signed [15:0] out_re;
		output signed [15:0] out_im;

qpsk_demodulator	**Input**	**Output**

	input signed [15:0] in_re;
	input signed [15:0] in_im;
		output out_odd;
		output out_even;

5 *Transmitter / Receiver*

Block Diagram (Simulink):

 1 [8x1] [8x1] Normal [48x1] [48x1] 1 Raised Cosine Transmit Filter *Figure 20 : Block Diagram for Transmitter*	 1 [8x1] [8x1] Normal [48x1] [48x1] 1 Raised Cosine Receive Filter *Figure 21: Block Diagram for Receiver*

Description and justification (Simulink):

The Raised Cosine Filter was chosen for our transmitter and its matched version was chosen for our receiver. The parameters were chosen so as to balance efficient spectral usage and signal clarity. The filter's roll-off factor of 0.25 ensures minimal inter-symbol interference (ISI) by controlling the bandwidth and smoothing the transitions between symbols. The filter span of 16 symbols aligns with the 16-bit data chunks, simplifying delay calculations. Additionally, using 6 samples per symbol enhances the accuracy of the signal representation, leading to better performance in subsequent processing stages and more accurate signal reconstruction, furthermore leading us to a sampling frequency of 960kHz. The matched filter on the receiver end optimizes the signal-to-noise ratio (SNR) and further minimizes ISI. Implementing these filters on an FPGA using Finite Impulse Response (FIR) filter approximations leverages the FPGA's capabilities and resources. The filter coefficients, which are designed to match the transmitter filter, ensure that the FPGA processes the signal accurately, maintaining the benefits of the Raised Cosine filtering in real-time applications.

Simulink Blocks & Parameters:

Simulink Block	Parameters Used
Raised Cosine Transmit Filter	**Filter shape:** Normal **Rolloff factor:** 0.25 **Filter span in symbols:** 16 **Output samples per symbol:** 6 **Linear amplitude filter gain:** 1 **Input processing:** Columns as channels (frame based) **Rate options:** Enforce single-rate processing **Export filter coefficients to workspace:** Checked **Coefficient variable name:** rcTxFilt
Raised Cosine Receive Filter	**Filter shape:** Normal **Rolloff factor**: 0.25 **Filter span in symbols:** 16 **Input samples per symbol:** 6 **Decimation factor:** 6 **Decimation offset:** 0 **Linear amplitude filter gain:** 1 **Input processing:** Columns as channels (frame based) **Rate options:** Enforce single-rate processing **Export filter coefficients to workspace:** Checked **Coefficient variable name:** rcRxFilt

Block diagram (FPGA):

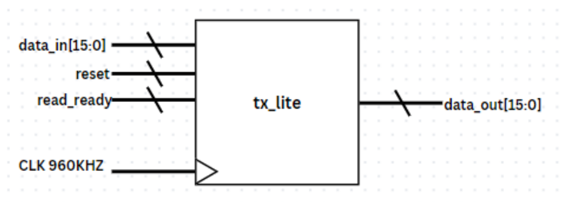


Figure 22: Block Diagram of Transmitter

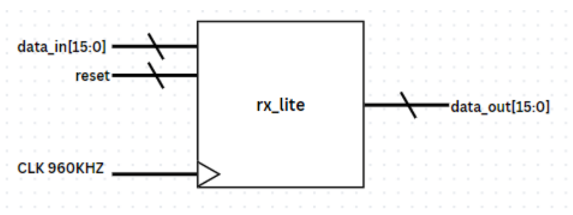


Figure 23: Block Diagram of Receiver

Description and Justification (FPGA):

The decision to implement raised cosine filters at both the transmitter and receiver was driven by the favorable results obtained from MATLAB simulations. These simulations met our specification of a 100kHz bandwidth by adjusting the roll-off factor of the filter. We opted for the conventional method of implementing these filters using FIR approximations over utilizing Quartus IP library blocks. This choice was made due to several challenges associated with the IP library. The IP tools obscure the design by black-boxing the components, making it difficult to understand and control the signals of the transmitter and receiver. Additionally, creating effective testbenches was problematic due to the opaque nature of the IP blocks, which led to significant delays. The IP catalog's approach also introduced unnecessary complexity and increased resource consumption by synthesizing numerous VHDL files outside the primary Verilog design.

By choosing FIR filter implementation, we gained greater control and understanding of the system. This approach allowed for a clearer comprehension of control signals, making testing and validation more straightforward. The filter coefficients were derived from MATLAB and scaled appropriately to meet channel quantization requirements by multiplying with 2**(13). Despite the higher resource usage, the FIR-based design fits within the FPGA's constraints, considering the heavy utilization of DSP blocks and other FPGA resources by other parts of the system. At the receiver end, the filter coefficients were scaled the same as the transmitter but they were additionally scaled down by the decimation factor to ensure the output amplitude matched expectations, thus avoiding distortions. Furthermore, the receiver end has a downsampler module which take a sample every 6 clock cycles thus matching the decimation factor needed for the filter.

Overall, the decision to use raised cosine FIR filters resulted in a more comprehensible and controllable design process, facilitating effective testing and integration within the FPGA while meeting the system's performance specifications.

Subsystem Component (FPGA):

Blocks	Parameters	
tx_lite	**Input**	**Output**

	input clk;
	input reset;
	input read_ready;
	input [15:0] data_in;
		output reg [16:0] data_out;

rx_lite	**Input**	**Output**

	input clk;
	input reset;
		output reg [16:0] data_out;

	input read_ready;
	input [15:0] data_in;	

multiply	**Input**	**Output**

	input signed [15:0] in1;
	input signed [15:0] in2;
		output reg signed [15:0] out;

decimation	**Input**	**Output**

	input wire [15:0] in;
	input wire clk;
	input wire reset;
		output reg [15:0] out;

6 Channel

Block Diagram (Simulink):

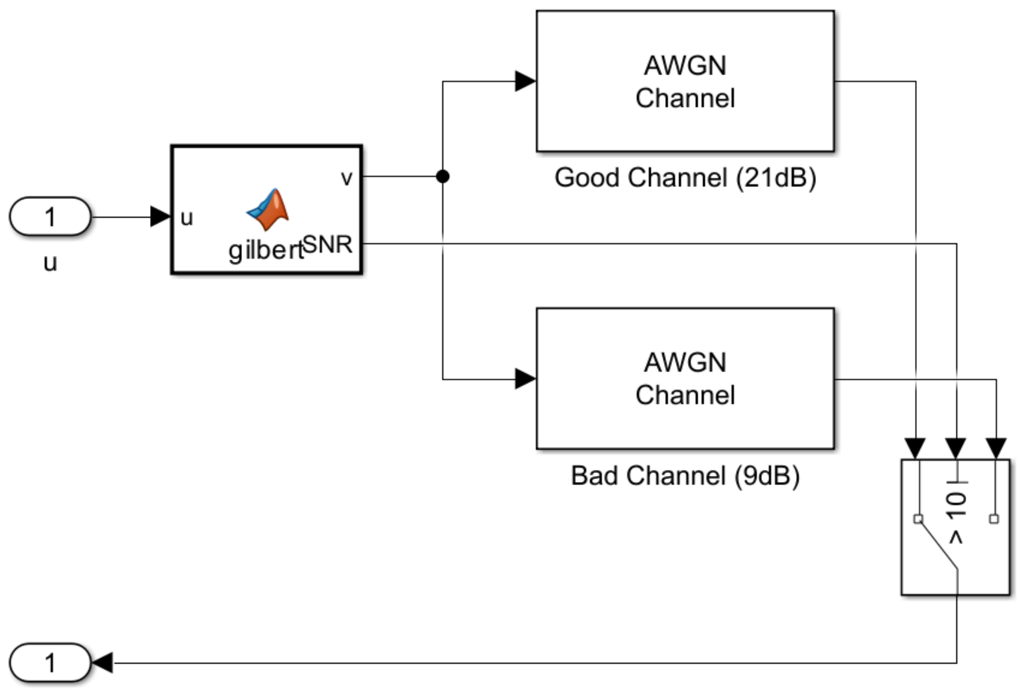


Figure 24: Block Diagram for Channel

Description and justification (Simulink):

The channel model follows a Gilbert model to simulate the effects of fading and varying channel conditions in a communication system. A MATLAB function block ('gilbert') determines the state of the channel, routing the input signal to two Additive White Gaussian

Noise (AWGN) channels (Good and Bad Channel). The "Good Channel" has a high signal-to-noise ratio (SNR) of 21 dB, while the "Bad Channel" has a lower SNR of 9 dB. The switch block uses the output of the gilbert function to direct the output signal coming out of either the good or bad channel based on the probabilistic state of the Gilbert model of channel γ. The input signal power is set to 1 watt (W) to maintain consistent power levels.

Simulink Blocks & Parameters:

Simulink Block	Parameters Used
MATLAB Function	**Coding:** Appendix A 1
AWGN Channel (GOOD)	**Mode:** Signal to noise ratio (SNR) **SNR(dB):** 21 **Input signal power:** 1 **Random number source:** Global stream **Simulate using:** Code generation
AWGN Channel (BAD)	**Mode:** Signal to noise ratio (SNR) **SNR(dB):** 9 **Input signal power:** 1 **Random number source:** Global stream **Simulate using:** Code generation
Switch	**Criteria for passing first input:** u2>Threshold **Threshold:** 10

Block diagram (FPGA):

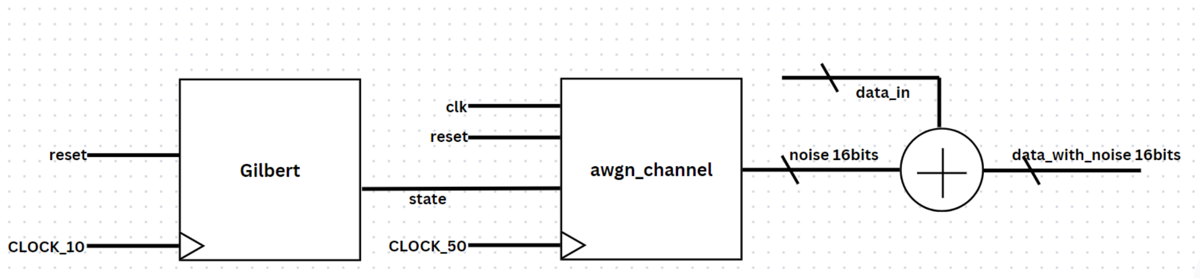


Figure 25: Block diagram for channel

Description and Justification (FPGA):

For the gilbert fading model, it is used to generate an enable signal to control the good and bad channel. We are not able to directly generate a random number. Therefore, we have used a LFSR block(random_number_generator) for random numbers which is not totally random. We have used the pre-generated lookup table for AWGN channel.

Subsystem Component (FPGA):

Blocks	Parameters	
awgn_channel	**Input**	**Output**

	input wire clk;
	input wire reset;
	input wire state;
		output reg [15:0] signal_out;

Gilbert	**Input**	**Output**

	parameter P_GB = 50;
	parameter P_BG = 200;
	input logic clk;
	input logic reset;
		output logic channel_state;

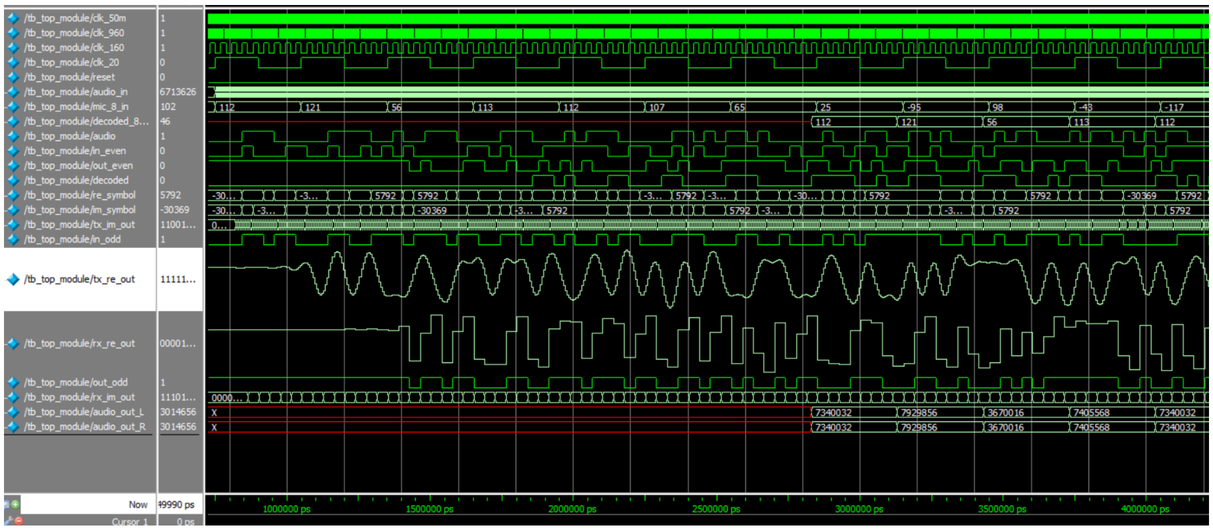
random_number_generator_8	**Input**	**Output**

	input logic clk;
	Input logic reset;
		output reg [7:0] rnd_number;

Verification

System Performance

FPGA:



```
[PASS]: left output value is 010010010000000000000000 (expected 010010010000000000000000)

[PASS]: left output value is 000111110000000000000000 (expected 000111110000000000000000)

[PASS]: left output value is 110000100000000000000000 (expected 110000100000000000000000)

# ========= TEST SUMMARY =========
#      TEST COUNT: 192
#   -      PASSED: 192
#   -      FAILED: 0
#   - PERCENT BER: 0
# ================================
```

Figure 26: System testbench and BER calculations

We designed a testbench to simulate the entire system from input to output excluding the channel, which could not be simulated because it requires a fpga memory block (thus the above waveform is a simulation of our system in a noise free channel). The waveforms above show the relevant signals from each major subsystem including the analog waveform of the receiver and transmitter. The test summary indicates that the inputs correctly match the outputs. Since our system and testbench were designed to mimic the Simulink model, these results show that our FPGA implementation closely matches our Simulink model and is able to reproduce the results using each of the system components modeled in the Simulink model and transformed into FPGA implementation. There are some minor differences between our FPGA implementation and the Simulink model which was seen with this testbench. The biggest difference is that the delay of our implemented FPGA design has an additional 13 cycles of delay compared to the Simulink model. This is due to the HDL generated Viterbi block. We also serialized our data for our FPGA implementation, whereas the Simulink model uses frames.

1 Source and Sink

Simulink:

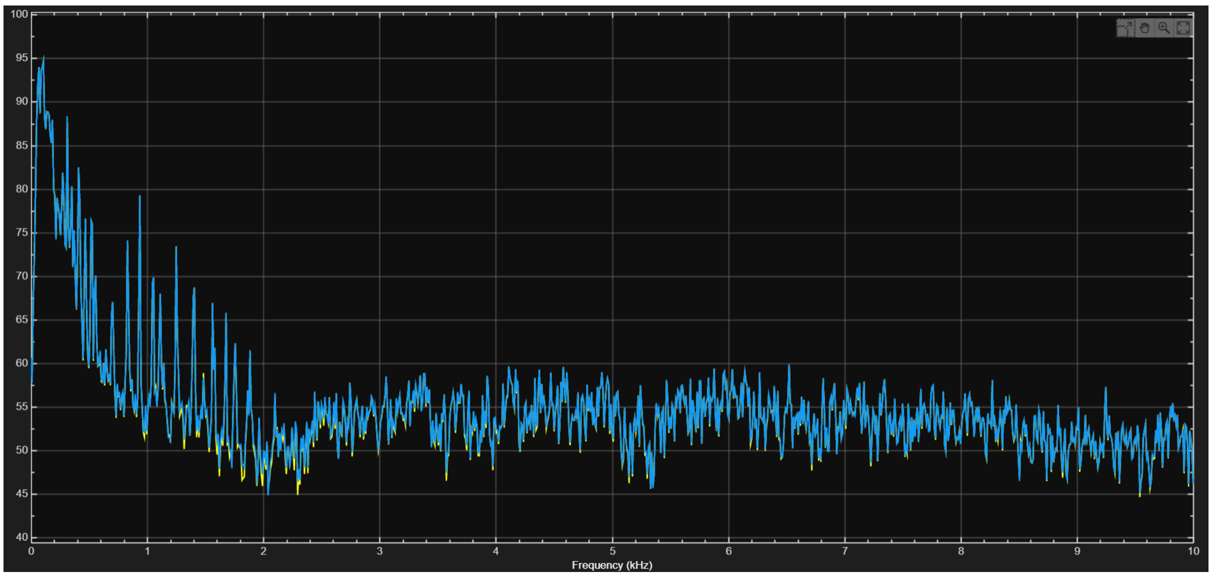


Figure 27: Frequency spectrum comparing Source (yellow) and Sink (blue) at 100 dB SNR

As the Figure shows above, the source frequency spectrum is shown in yellow and the sink frequency spectrum is shown in blue. In analyzing the frequency spectrum of the source and sink waveforms, it is evident that both spectra are nearly identical when observed under conditions of a high signal-to-noise ratio.

FPGA:

Input:

For the microphone input, the audio codec module from the FPGA starter has been implemented to capture the audio and transfer it in digital with a stereo channel of 24 bits.

For the Wav file input, it was converted to mif file to store into the memory. It is stored in 16 bits and 1024 words.

Output:

For the speaker output, the signal is transferred to the audio codec as 24 bits in the stereo channel. As expected, the sound output from the speaker matches the input microphone.

For the wav file, the signal is stored to another memory block whose number of words and bits matched with the input memory. Then, we export the memory and convert it back to a wav file. The resulting wav sounds very similar to the original wav.

2 A/D and D/A Converters

Simulink:

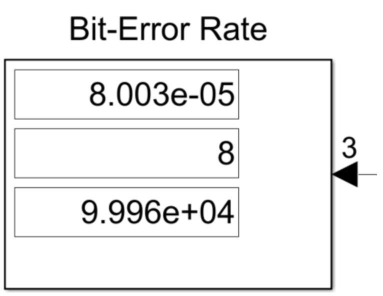


Figure 28: Converter BER at SNR = 9dB

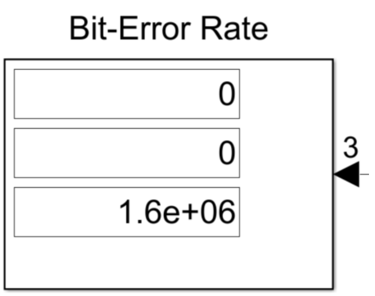


Figure 29: Converter BER at SNR =100dB

Our system seems to perform well through a channel with a SNR of both 9dB and 100dB with respect to bit error rate (BER). The BER for SNR of 100dB was 0 as expected. The BER when SNR = 9dB did have an increase in error, which is expected given the SNR is much lower, however the BER was still under our project's requirements of BER < 10^{-4}. Our encoder in combination with QPSK modulation seems to be performing well.

FPGA:

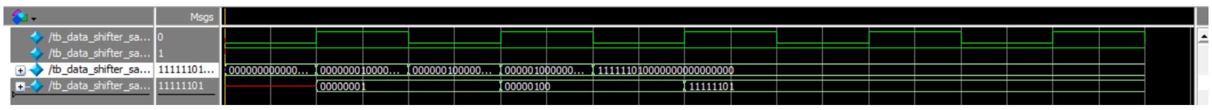


Figure 30: shift sampling (ADC)

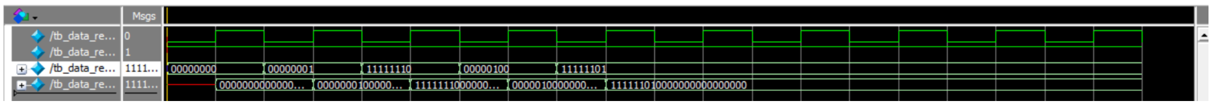
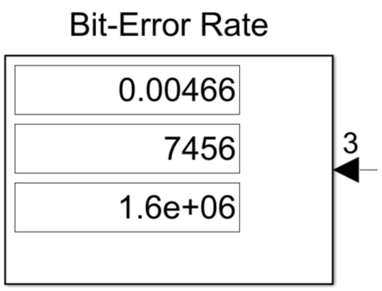
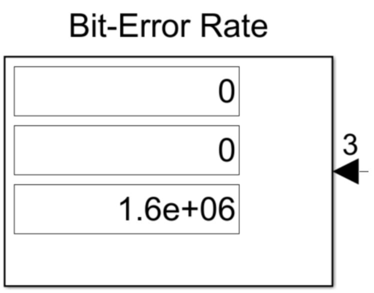
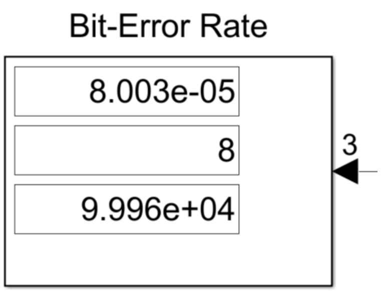
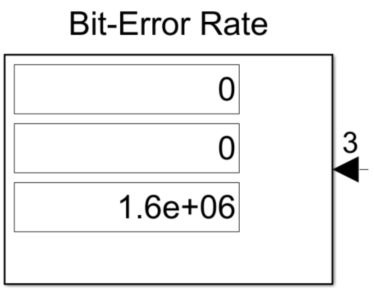


Figure 31: resampling (DAC)

The ADC and DAC blocks are working as expected. For the ADC, it shrinks from 24 bits to 8 bits and passes the output to the encoder module, and is triggered at the positive edge of the clock. The clock is implemented at 20 kHz. For the DAC, it expands the 8-bit input from the Viterbi decoder to 24 bits to pass through the Audio Codec. Therefore, they are satisfied with the quantization level and sampling rate requirements.

3 Error Correction Encoder/Decoder

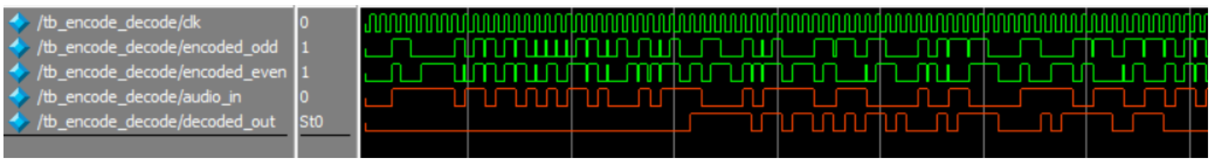
Simulink:

 Bit-Error Rate 0.00466 7456 1.6e+06 3 *Figure 32: BER at 9dB without encoder/ decoder*	 Bit-Error Rate 0 0 1.6e+06 3 *Figure 33: BER at 100dB without encoder/ decoder*
 Bit-Error Rate 8.003e-05 8 9.996e+04 3 *Figure 34: BER at 9dB with encoder/decoder*	 Bit-Error Rate 0 0 1.6e+06 3 *Figure 35: BER at 100dB with encoder/ decoder*

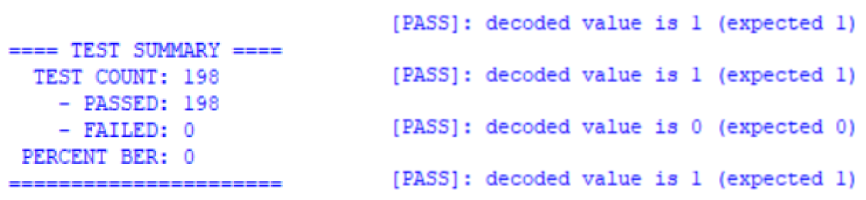
As expected, the BER with the error correction encoder and decoder implemented is significantly lower than without the encoder and decoder enabled when the SNR is 9dB. It also meets expectations that the BER is higher at 9dB compared to 100dB since a SNR of 100dB has minimal noise.

FPGA:

Because our method of implementing the AWGN channel in the fpga used the fpga memory blocks, we were not able to simulate our channel in a testbench. Therefore, we were unable to test the BER of the encoder and decoder on our channel model using a testbench. However, we can show that our encoder and decoder work perfectly to encode the data and then decode the data in a noise-free channel.



```
/tb_encode_decode/clk           0
/tb_encode_decode/encoded_odd   1
/tb_encode_decode/encoded_even  1
/tb_encode_decode/audio_in      0
/tb_encode_decode/decoded_out   St0
```



```
==== TEST SUMMARY ====
  TEST COUNT: 198
    - PASSED: 198
    - FAILED: 0
 PERCENT BER: 0
======================
```

```
[PASS]: decoded value is 1 (expected 1)

[PASS]: decoded value is 1 (expected 1)

[PASS]: decoded value is 0 (expected 0)

[PASS]: decoded value is 1 (expected 1)
```

Figure 36: Encoder and Decoder waveform showing correct encoding and decoding through a noise-less channel

As shown in the waveform from a noise-free channel, the red signals are replicas of one another, but the lower decoded output signal has a delay, which is expected due to the Viterbi decoder. The two waveforms above in green are the transformed encoded signal from audio_in. As shown in the test summary, the percent BER for a noise-less channel using the encoder and decoder is 0% as expected, which shows that the encoder and decoder work together. We did implement a switch on our fpga to turn on and off the noisy channel so we toggled between a noise-free channel and a noisy channel. Although we could not verify a specific BER, we did not notice any difference in the sound quality when we toggled between a noise-free channel and our noisy channel in our FPGA implementation so we suspect that the BER is relatively low since error was not noticeably detectable by the ear in our system.

Of note as well is that our symbol rate has increased with encoding by a factor of 2 since with one input bit (audio_in), we get two output bits (encoded_even and encoded_odd) due to the code rate being ½. Thus the symbol rate has now doubled since there are twice as many bits on output as on input.

4 Modulation/Demodulation

Simulink:

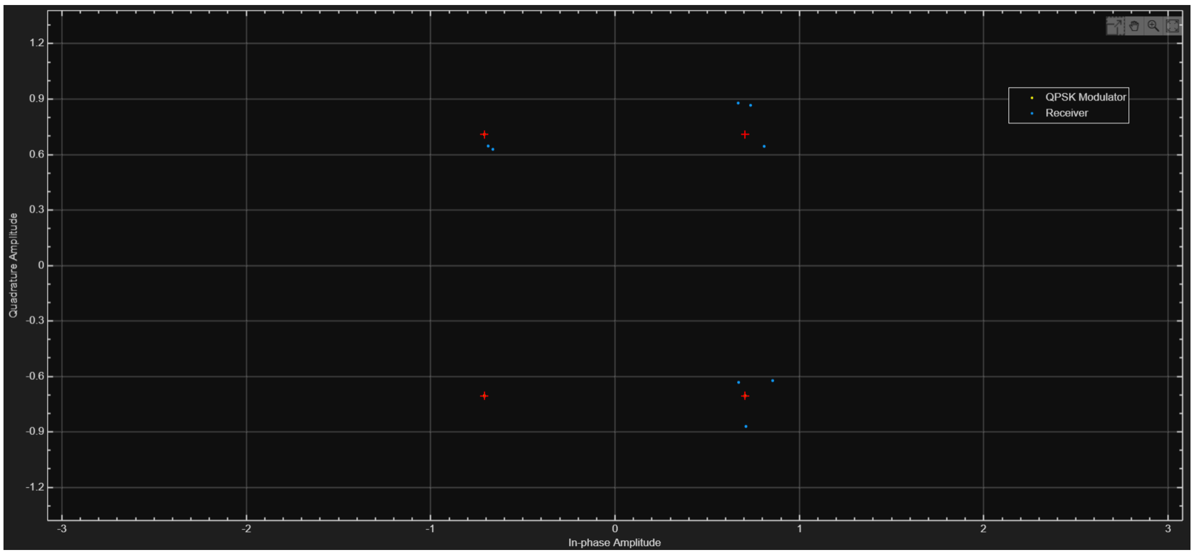


Figure 37: Constellation scatter plot comparing modulator output & demodulator input for 100dB SNR

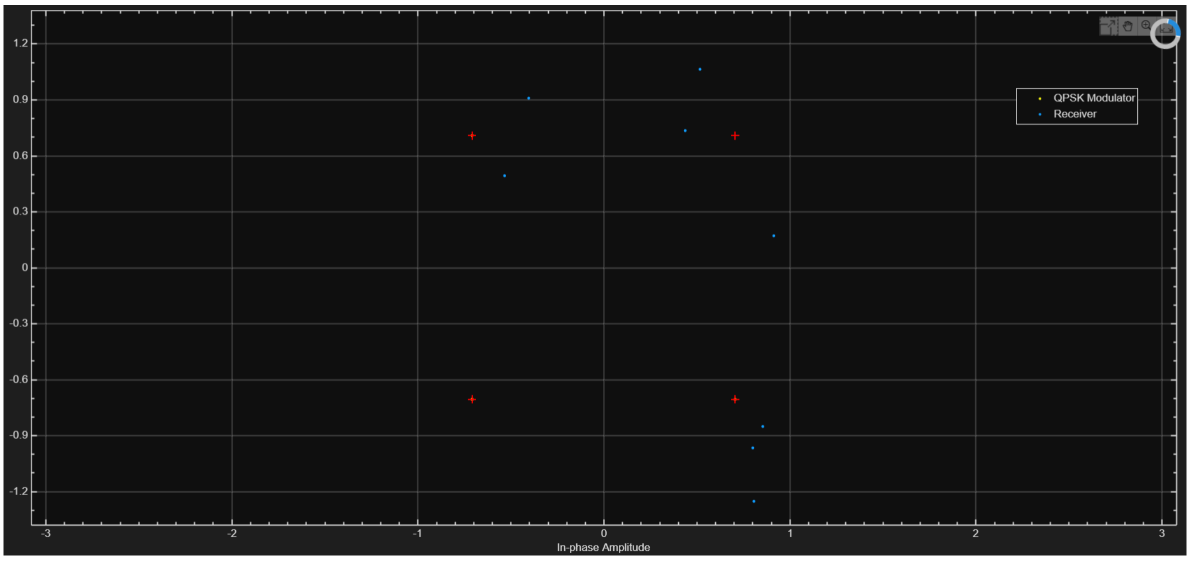
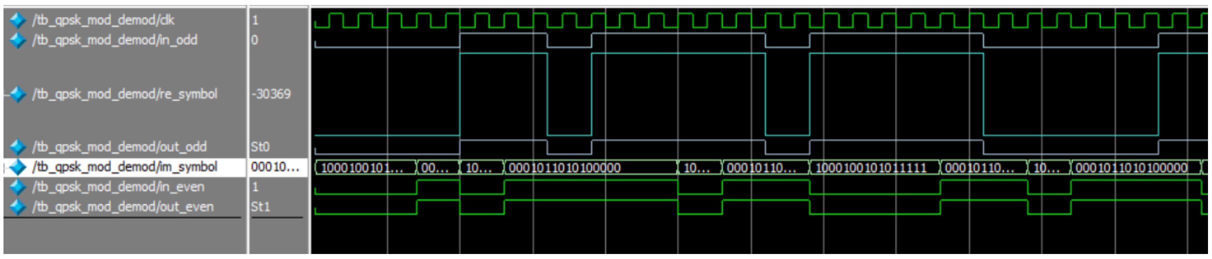


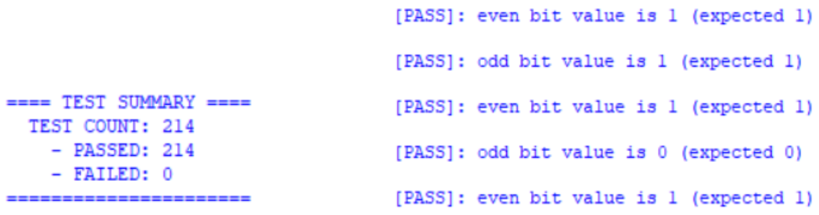
Figure 38: Constellation scatter plot comparing modulator output and demodulator input for 9dB SNR

As expected, there is greater scatter and spread of the points around the symbols for a SNR of 9dB than 100dB. The points are very close to the symbols for 100dB, which is consistent with a very low noise channel, but they are more spread out and further from the expected values of the symbols for 9dB because the noise would add to the signal's symbols and move them further from their expected values.

Of note, the modulator output points for both graphs cannot easily be seen on the scatter plot because they all lie on the red '+' indicating the location of the symbols. This is expected since all the bits should have been mapped to symbols at these locations by the modulator.

FPGA:


/tb_qpsk_mod_demod/clk
1
/tb_qpsk_mod_demod/in_odd
0
/tb_qpsk_mod_demod/re_symbol
-30369
/tb_qpsk_mod_demod/out_odd
St0
/tb_qpsk_mod_demod/im_symbol
00010...
/tb_qpsk_mod_demod/in_even
1
/tb_qpsk_mod_demod/out_even
St1
1000100101...
00...
10...
0001011010100000
10...
00010110...
1000100101011111
00010110...
10...
0001011010100000



```
==== TEST SUMMARY ====
  TEST COUNT: 214
    - PASSED: 214
    - FAILED: 0
======================
```

```
[PASS]: even bit value is 1 (expected 1)

[PASS]: odd bit value is 1 (expected 1)

[PASS]: even bit value is 1 (expected 1)

[PASS]: odd bit value is 0 (expected 0)

[PASS]: even bit value is 1 (expected 1)
```

Figure 39: Modulator and Demodulator Inputs and Outputs in ModelSim and BER tests

The waveform above shows how the input of the modulator (top waveform in purple) is transformed to its symbol representation (waveform in blue), and then is demodulated back into the correct bit form on the output of the demodulator (purple waveform below the blue waveform). This can also be seen in the green waveforms below which show the input and output of the even-bits, and the binary representation of the imaginary part of the symbol. The inputs and outputs match perfectly, and the binary representation of the symbol is either the positive or negative Q2.13 signed fixed point representation of sqrt(2)/2 as expected. The Test summary and values below show the comparison done by the testbench, which shows the BER would be 0 since there are no failed tests comparing input and output.

The modulator correctly adjusts the symbol rate because even though there are two outputs for two input bits, the two outputs are each a half of the complex symbol (one is the real part and the other is the imaginary part) so they constitute a single symbol. Therefore, the symbol rate has been halved as expected with QPSK.

5 Transmitter/Receiver

Simulink:

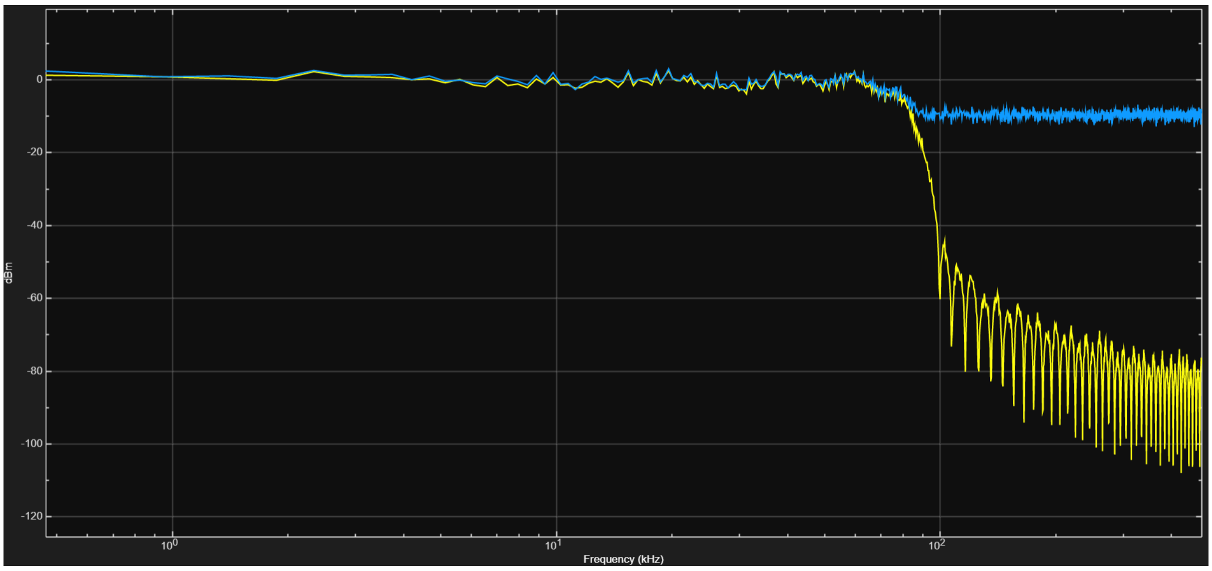


Figure 40: Frequency domain signals from the output of the transmitter to the input of the receiver at 9dB

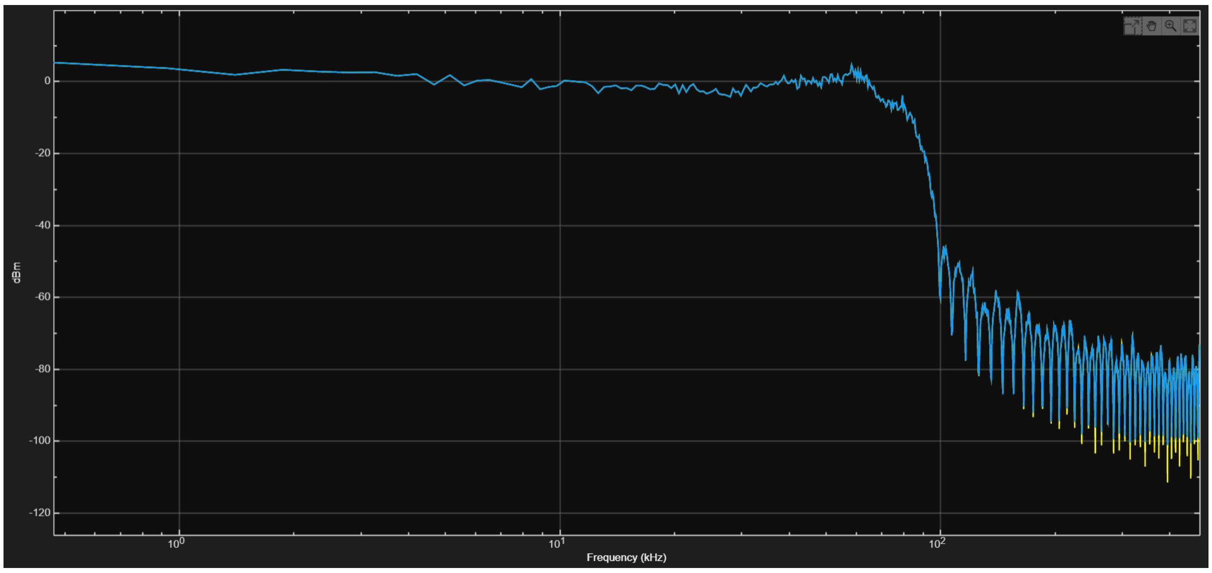


Figure 41: Frequency domain signals from the output of the transmitter to the input of the receiver at 100dB

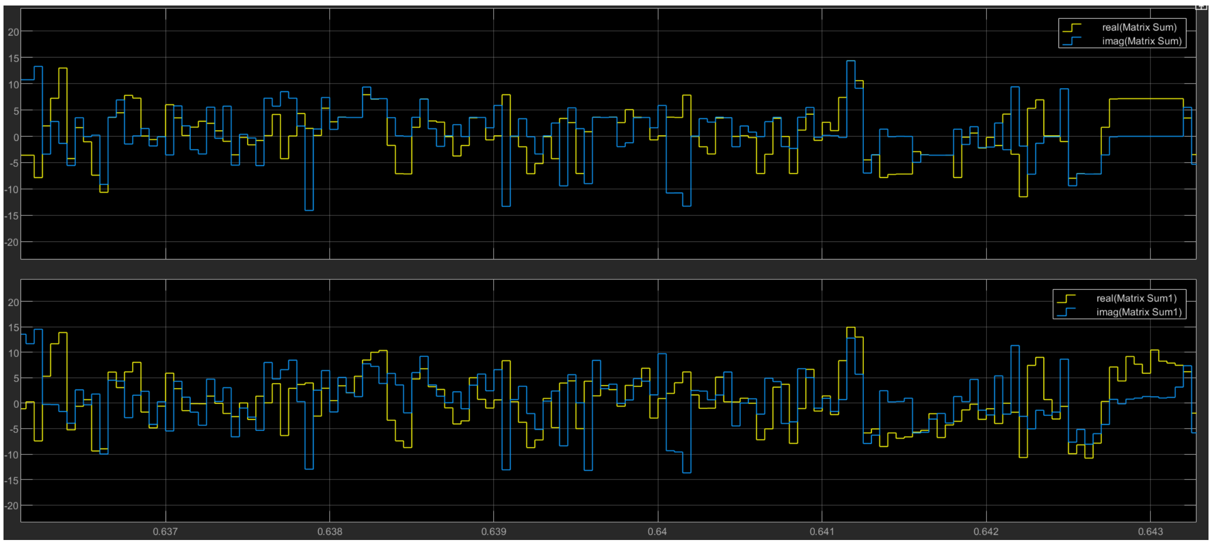


Figure 42: Time domain signals from the output of the transmitter to the input of the receiver at 9dB

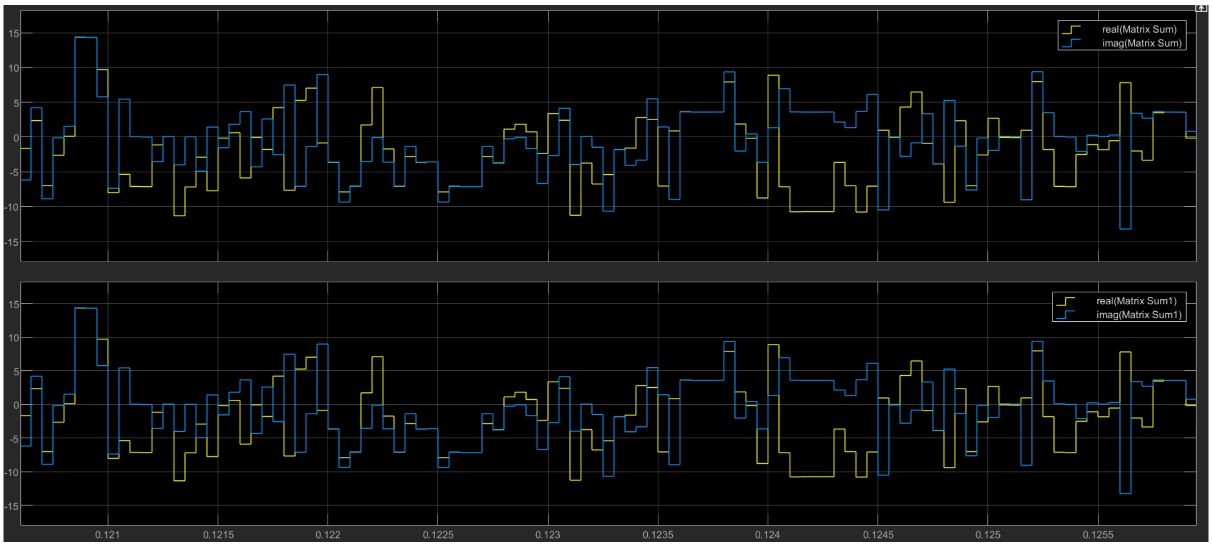
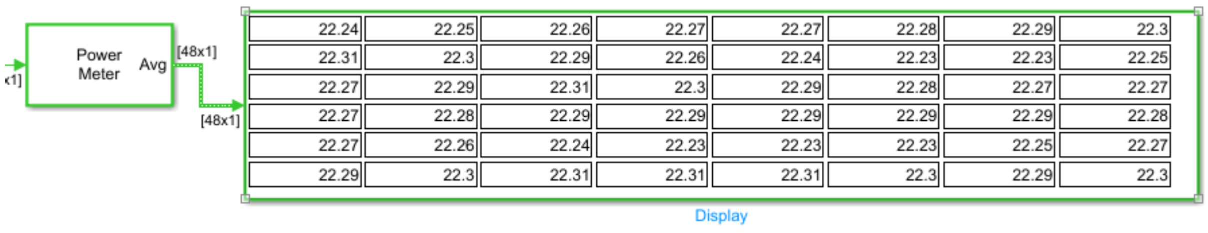


Figure 43: Time domain signals from the output of the transmitter to the input of the receiver at 100dB

The two graphs compare the frequency domain signals from the output of the transmitter to the input of the receiver at SNR levels of 9 dB and 100 dB respectively. In the first graph, the transmitted waveform (yellow) shows significant attenuation and distortion due to the low SNR condition. The received waveform (blue) is heavily affected by noise, resulting in high fluctuations and reduced amplitude, indicating a degraded signal quality. In contrast, the second graph shows that the transmitted signal maintains its amplitude and consistency across the frequency spectrum with minimal distortion. The received waveform closely matches the transmitted signal, indicating maximum signal integrity with minimal noise interference.



22.24	22.25	22.26	22.27	22.27	22.28	22.29	22.3
22.31	22.3	22.29	22.26	22.24	22.23	22.23	22.25
22.27	22.29	22.31	22.3	22.29	22.28	22.27	22.27
22.27	22.28	22.29	22.29	22.29	22.29	22.29	22.28
22.27	22.26	22.24	22.23	22.23	22.23	22.25	22.27
22.29	22.3	22.31	22.31	22.31	22.3	22.29	22.3

We use the following configuration above to calculate the total energy over a given time frame. In the above example we send the transmitted waveform into the power meter. The sum of elements block sums the 64 elements per frame and feeds it to the integrator giving us the total energy. As we can see the total energy over a duration of 5 seconds gives us 80.02 J of energy.

FPGA:

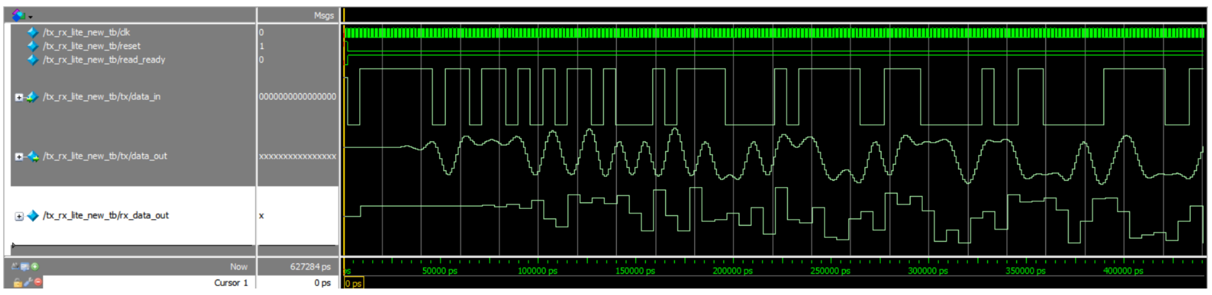


Figure 44: Testbench waveforms of the transmitter and receiver modules

The testbench devised to test the transmitter and receiver generates random expected QPSK values at intervals of 160kHz. This QPSK input is passed into the transmitter and then the waveform produced by the transformer is passed into the receiver for signal reconstruction. The analog waveform ‘tx/data_in’ is the waveform of randomized QPSK values that the transmitter can expect from the modulator. The waveform ‘tx/data_out’ shapes the symbols into cosine pulses and introduces a delay, which makes sense since the transmitter introduces a delay which is proportional to the span of the filter. The waveform ‘rx_data_out’ tries to recover the QPSK symbols from the transmitted cosine pulses, and it accurately does so with a rougher waveform since the receiver includes a downsampler. Once again, the filter span in the receiver introduces more delay to the data transmission.

Attached below is the graph of the time domain versus the frequency domain signal of the transmitter output:

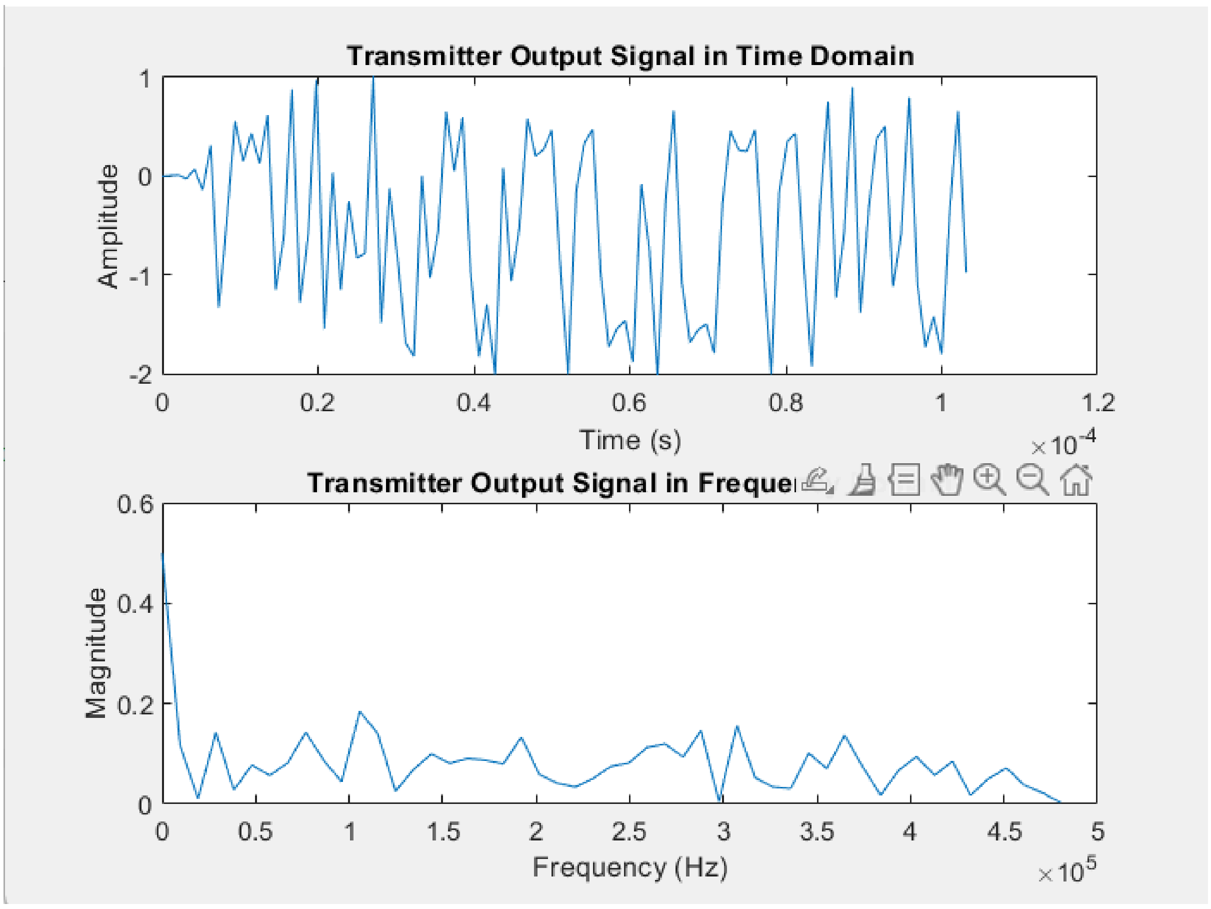


Figure 45: Time Domain vs Frequency Domain of the Transmitter

From the frequency domain graph, we see that we meet the 100kHz spectral mask constraints. This was pulled using MATLAB. Using ModelSim, we pulled the values of the transmitter output from a generated random signal. We do the necessary scaling to match the quantization of the bits and then take the mean of the square of the signal. On running the matlab script, we see that the average energy is 1.0081, which is close to one.

Since the channel uses memory blocks from the FPGA, we could not get a testbench with the channel integrated to be working because of which we could not pull data from the output of the channel to calculate power.

6 Channel

Simulink:

The Variance for the AWGN Channel is calculated as follows:

$$10^{-SNR/10} = 10^{-9/10} = 0.126$$

$$10^{-SNR/10} = 10^{-21/10} = 0.00794$$

With solving the following equations, we are able to get that at steady state, the average amount of time the channel is in the good state is 80%, and the average time it is in the bad state is 20%:

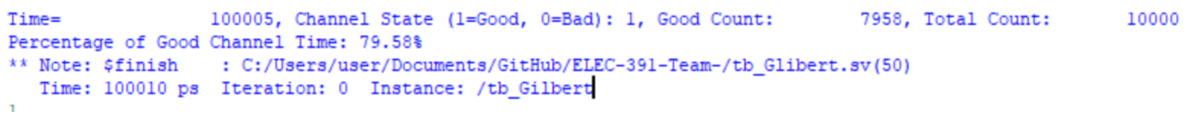
x: fraction of time the channel is in the good state at steady state

y: fraction of time the channel is in the bad state at steady state

x + y = 1

$$\begin{bmatrix} 0.95 & 0.8 \\ 0.05 & 0.2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

FPGA:



```
Time=              100005, Channel State (1=Good, 0=Bad): 1, Good Count:       7958, Total Count:      10000
Percentage of Good Channel Time: 79.58%
** Note: $finish    : C:/Users/user/Documents/GitHub/ELEC-391-Team-/tb_Glibert.sv(50)
   Time: 100010 ps  Iteration: 0  Instance: /tb_Gilbert
```

Figure 46: Gilbert fading model testing result

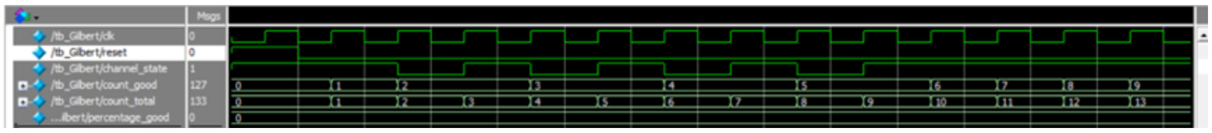

Msgs
/tb_Gilbert/clk 0
/tb_Gilbert/reset 0
/tb_Gilbert/channel_state 1
/tb_Gilbert/count_good 127
/tb_Gilbert/count_total 133
...ilbert/percentage_good 0

Figure 47: Waveform shows the changing in enable signal

For the Gilbert module, the testbench is implemented to test the steady state of the Gilbert fading module. Our Gilbert module is set to output an enable signal which is high for good channels and low for bad channels. Our testbench counts the number of goodstates and finds its ratio to all states. The result shows a rate of 80% which is matched up with the average percentage of good state calculated simulink modules. As well, as we can capture in the waveform, the enable signal does change as expected.

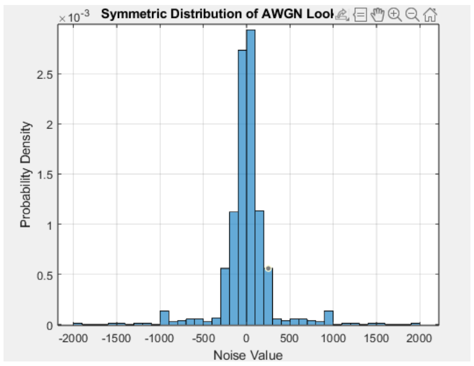


Figure 48: Distribution in 9dB

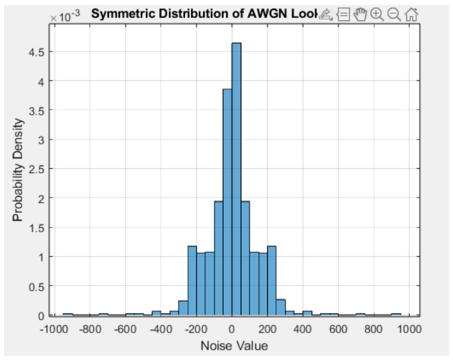


Figure 49: Distribution in 21dB

For the AWGN channel, due to the implementation of mif file, there is no direct method to directly test its distribution through a testbench. We used the matlab to show the distribution of the lookup table.As figure shown, it is normally distributed and standard deviation is wider for 21dB noise as expected.

References

[1] Information on convolutional codes from:

https://en.wikipedia.org/wiki/Convolutional_code

[2] Graph for BER for Convolutional codes taken from Wikimedia Commons:

https://commons.wikimedia.org/wiki/File:Lenss.png

3] Information on the implementation and BER of QPSK found from:

https://www.gaussianwaves.com/2010/10/qpsk-modulation-and-demodulation-2/

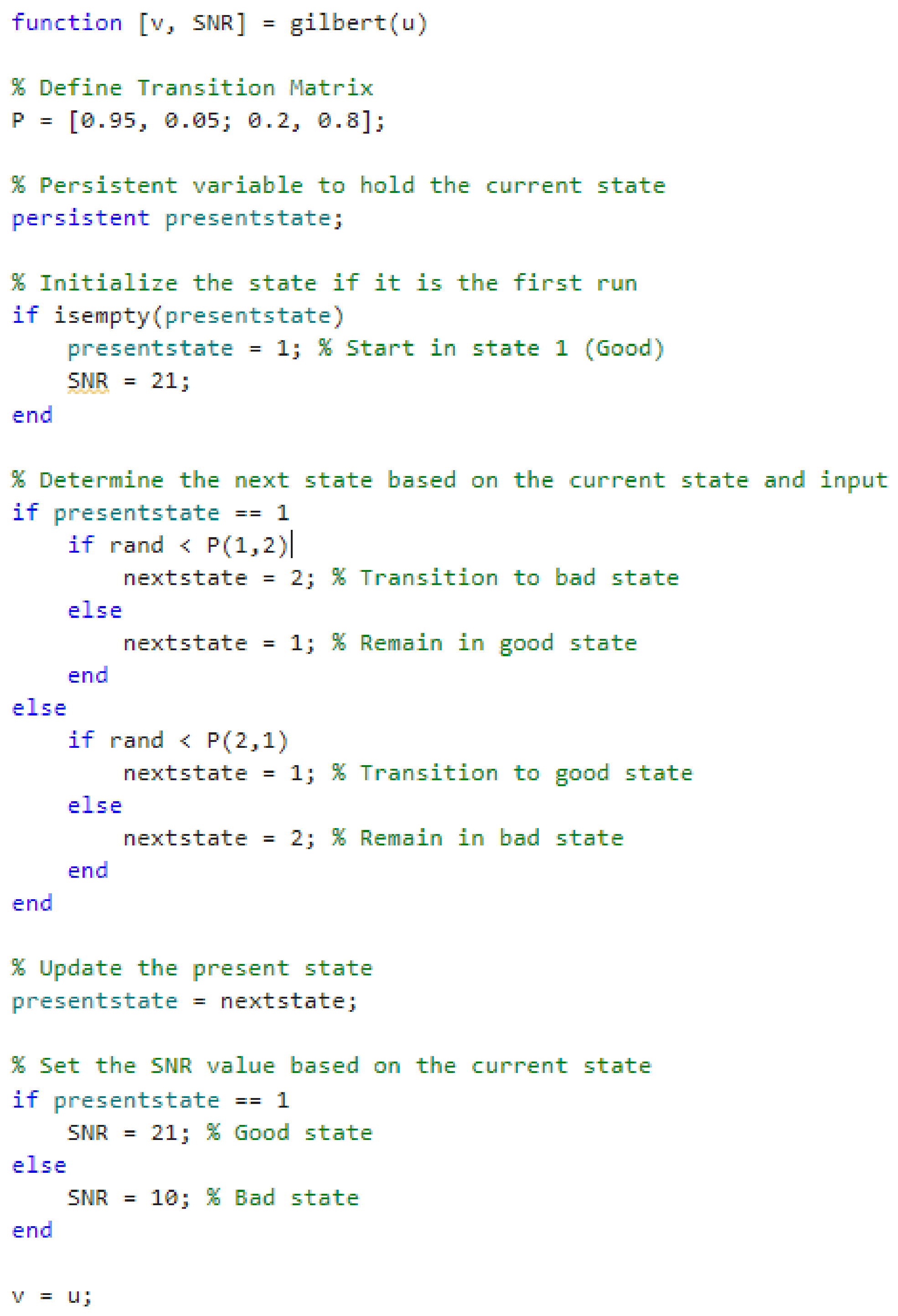
Appendix A: Deliverables

Deliverable	File Name	Notes (optional)
Product Document	ELEC391_FinalReport_Team8.pdf	
Product Presentation	ELEC 391 Final Presentation.pdf	
Simulink system file	Team8_SimulinkModel_FINAL.slx Stereo_32bit_48kHz_Piano_Drums.wav	
HDL source code, including testbenches and readme files	**Top Module** Top_module.sv	
	ADC / DAC ADC_DAC_description.txt Test Strategy&waveform_ADC_DAC.docx data_shifter_sampling.v tb_data_resampling.v tb_data_shifter_sampling.v	
	Encoder / Decoder encoder_decoder_design_and_test_strategy .docx convolutional_encoder.v Viterbi_Decoder1.v ACS.v ACSEngine.v ACSRenorm.v ACSUnit.v BranchMetric.v Traceback.v TracebackUnit.v tb_encode_decode.sv	
	Modulation / Demodulation Modulator_demodulator_design_and_test_ strategy.docx qpsk_modulator.sv qpsk_demodulator.v tb_qpsk_mod_demod.sv	
	Transmitter / Receiver tx_rx_module_description.pdf tx_lite.sv rx_lite.v multiply.v decimation.sv tx_rx_tb_new.sv	

	<u>Channel</u> AWGN_description.txt Gilbert_description.txt awgn_channel.sv tb_awgn_channel.sv Gilbert.sv tb_Gilbert.sv AWGN_LUT_9dB.mif AWGN_LUT_21dB.mif	
	<u>Serialization & Top Module Testing</u> top_level_system_test_strategy.docx tb_top_module.sv top_module_test.sv serializer.sv deserializer.sv delay.sv	Created a test version of top_ module without the audio codec variables for testing with the system testbench.

Appendix B.

- The Gilbert fading model code is generated by ChatGPT



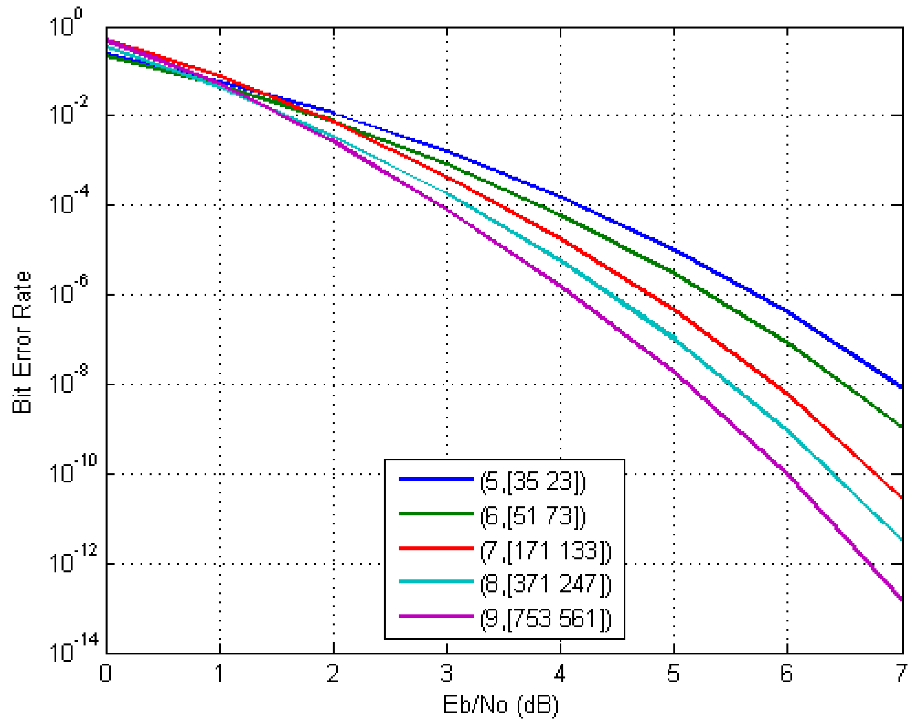
```
function [v, SNR] = gilbert(u)

% Define Transition Matrix
P = [0.95, 0.05; 0.2, 0.8];

% Persistent variable to hold the current state
persistent presentstate;

% Initialize the state if it is the first run
if isempty(presentstate)
    presentstate = 1; % Start in state 1 (Good)
    SNR = 21;
end

% Determine the next state based on the current state and input
if presentstate == 1
    if rand < P(1,2)
        nextstate = 2; % Transition to bad state
    else
        nextstate = 1; % Remain in good state
    end
else
    if rand < P(2,1)
        nextstate = 1; % Transition to good state
    else
        nextstate = 2; % Remain in bad state
    end
end

% Update the present state
presentstate = nextstate;

% Set the SNR value based on the current state
if presentstate == 1
    SNR = 21; % Good state
else
    SNR = 10; % Bad state
end

v = u;
```

-

The Viterbi_Decoder1 module and submodules were generated using HDL Coder. A screenshot of the code is not provided since all the files are included in the HDL zip folder, and include commentary in the code indicating they were generated by the HDL Coder.

Appendix C.

Appendix C. Figure 1: Theoretical bit error rate curves for QPSK modulated signals encoded with convolutional codes of various constraint lengths through an AWGN channel.



Appendix D.

- This is the script used to generate the average energy consumed by the transmitted signal and the input of the receiver. Furthermore, it plots the time domain signal vs the frequency domain signal

```
fileID = fopen('tx_output_signal.txt', 'r');
tx_hex_data = textscan(fileID, '%s');
fclose(fileID);
tx_output_signal = hex2dec(tx_hex_data{1});
tx_output_signal(tx_output_signal >= 2^15) = tx_output_signal(tx_output_signal >=
2^15) - 2^16;
% Apply quantization scaling
tx_output_signal = tx_output_signal * 2^-13;
% Calculate the time vector assuming the sampling period Ts
Ts = 1/960e3; % assuming 960 kHz sampling frequency
t = (0:length(tx_output_signal)-1) * Ts;
% Plot time domain signals
figure;
subplot(2, 1, 1);
plot(t, tx_output_signal);
title('Transmitter Output Signal in Time Domain');
xlabel('Time (s)');
ylabel('Amplitude');
% Calculate and plot frequency domain signals
n = length(tx_output_signal);
f = (0:n-1)*(1/(n*Ts));
Tx_fft = abs(fft(tx_output_signal)/n);
Tx_fft = Tx_fft(1:n/2+1);
subplot(2, 1, 2);
plot(f(1:n/2+1), Tx_fft);
title('Transmitter Output Signal in Frequency Domain');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
% Calculate the average signal energy
signal_energy = mean(tx_output_signal.^2);
disp(['Average Signal Energy: ', num2str(signal_energy)]);
% Ensure average signal energy is one
if abs(signal_energy - 1) < 0.01
   disp('Average signal energy is approximately one.');
else
   disp('Average signal energy is not one. Adjustments might be needed.');
end
```

Team Contributions

Yifeng Liu:

Individual Contribution:
In the distribution of tasks and cooperation within our group, I played a significant role in the design and implementation of the ADC module, DAC module, and the AWGN channel. In addition to these individual contributions, I collaborated with Sidharth on the development of the Gilbert fading module.

One of the primary challenges I encountered was related to the generation of random numbers in hardware. Initially, we planned to use a random number function for state transitions in the Gilbert fading module. However, the direct implementation of this function was not feasible in hardware. To address this, we substituted the random number function with a Linear Feedback Shift Register (LFSR), which is more suitable for hardware applications.

Another significant challenge was related to memory implementation. At the outset, I struggled with understanding how to properly read from and write to memory, which made it difficult to write an effective testbench for testing purposes. To overcome this, I decided to implement the memory directly on the FPGA board. Through several iterations and trials, I was able to achieve a functional implementation.

Team Effectiveness:
Our team has exhibited excellent chemistry, and everyone has taken their individual roles seriously. Despite encountering issues with the transmitter and receiver components during the integration of the entire system, we managed to complete the task on time. I feel honored to be part of this team.

However, there is room for improvement in our communication. Certain misunderstandings occurred between us, leading to the issues with the transmitter and receiver. In the future, we should be more proactive in communicating about the modules relevant to our respective roles. This will help us to prevent similar issues and ensure smoother collaboration.

Other Comments:
I believe we should have a better understanding of how the entire system works, rather than just focusing on our individual parts. Gaining a deeper insight into the overall system will enable us to comprehend our respective blocks more accurately. This will make future system integration easier.

Lynn Kelly:

Individual Contribution:
My role on the team was as project lead for the encoder/decoder subsystem and the modulator/demodulator subsystem. I researched and made the design decisions for choosing and implementing the error correction encoder and decoder and the modulation and demodulation scheme. I modeled the blocks on Simulink and chose the parameters to optimize the system. I also wrote HDL code and found methods to generate HDL code for these subsystems to implement them into the FPGA. In addition to these subsystems, I also designed and wrote the serializer, deserializer and delay modules for the FPGA implementation. I designed and wrote testbenches for all of my systems and the serializer and delay modules, and I also designed and wrote the top-level system testbench, which was a testbench that incorporated all of the subsystems and allowed us to visualize all of the signals between subsystems to determine delay and synchronization, and was essential in debugging integration issues.
I participated with the rest of the team on integrating the subsystems for both Simulink and FPGA, and I assisted my teammates to the best of my ability if they came across problems. As a team we regularly reviewed the systems together and had group discussions on design decisions and any changes that needed to be made.

Team Effectiveness:
I believe we worked well together as a team and we all brought different strengths to the team. Everyone worked very hard and consistently communicated and participated throughout the project. Although there was a lot of pressure on us near the end as we were approaching the deadline and we were worried we were not going to get the project finished, we still managed to work together up to the end and in doing that we were able to complete the full implementation and meet our objectives and deadlines.

Although I do think we worked very well together, we do still have some room for improvement in certain areas. I think as a team we could work at communicating better and more clearly to make sure we all understand what everyone is working on and what problems we are all facing on our parts. As well, I think we could do a better job of trying to organize ourselves earlier on before deadlines to determine what our priorities are to get a better workflow.

Other Comments:
With regards to how I ensured my tasks would integrate with my group, I tried to make design decisions that were relatively flexible in case others in the group needed me to change things about my design. As well, I tried to understand the other's systems and their design decisions so that I could optimize the integration between our systems as best as possible.
One of the most significant hurdles I had to overcome was trying to learn and understand the theory of the communication system as a whole and each subsystem within it. I spent a lot of time trying to learn about the system as I found it was essential knowledge to be able to complete the project and meet our specifications.

Sidharth Sudhir:

Individual Contribution:
I played a role in implementing the transmitter and receiver modules of the system, and collaborated with Yifeng to figure out the Gilbert Fading Model. While we managed to get the transmitter and receiver working towards the end, I came across a lot of roadblocks during the implementation of the transmitter and receiver because of which our system integration was delayed. This is primarily because of the fact that the FIR module in the IP Library had a lot of black box logic within it which is why I took a lot of time to understand the operation of the filter. Furthermore, the IP modules introduced a lot of external files in Quartus which had to be taken into consideration during simulation. This resulted in a lot of errors when running the testbenches in modelsim due to new libraries needing to be compiled. While I made significant progress in figuring out the IP module, our team was under the fear of not finishing integration on time due to which I resorted to a much more conventional implementation of the FIR filter, which worked by the end of it.

Team Effectiveness:
Our team was dedicated to finishing all the tasks properly and on time, despite facing multiple issues with the transmitter and receiver during system integration. In the future, it would be best for all the team members to be aware of the functionality of different components in the system so that different perspectives can be taken when facing issues.

Other Comments:
N/A