COMMUNICATION MODEL DESIGN & SIMULATION: DEMO #2

ELECTRICAL DESIGN STUDIO II

TEAM 12 - Aaron Loh, Fiona Luo, Peter Kim

Index

Objectives, Requirements and Constraints

The goal of this project is to design and develop a communication system that aligns with the specified performance criteria detailed in scenario F. Scenario F necessitates a broader audio bandwidth of 20kHz, thereby demanding a higher data transmission rate. Also, even though the channel noises are to be designed with 21dB and 9dB across all scenarios, the broader bandwidth introduces a higher potential for bit errors due to its wider frequency range, which is reflected in the less stringent bit error rate (BER) requirement of 10^-3. Moreover, the system must adhere to the delay constraint of 25ms, posing limitations on the selection of error detection, error correction schemes, and quantization processes.

Our approach to addressing these challenges involves a thorough review of existing literature and hands-on validation to select appropriate subsystem blocks and optimize their parameters.

The resulting communication system design successfully meets all specified performance metrics, except for the BER, which has been measured to exceed by 1.678%.

Table 6: Performance Scenarios

Scenario (Teams)	High Priority				Low Priority
	Audio BW (kHz)	Target BER	Spectral Mask (kHz)	Target Channel	Delay (ms)
A	4	10^{-5}	100	δ	25
B	8	10^{-4}	100	γ	25
C	20	10^{-3}	150	β	35
D	4	10^{-5}	150	δ	15
E	8	10^{-4}	200	γ	15
F	20	10^{-3}	200	β	25

Figure 1: Distributed Performance Scenarios

Communication System Design
System Overview

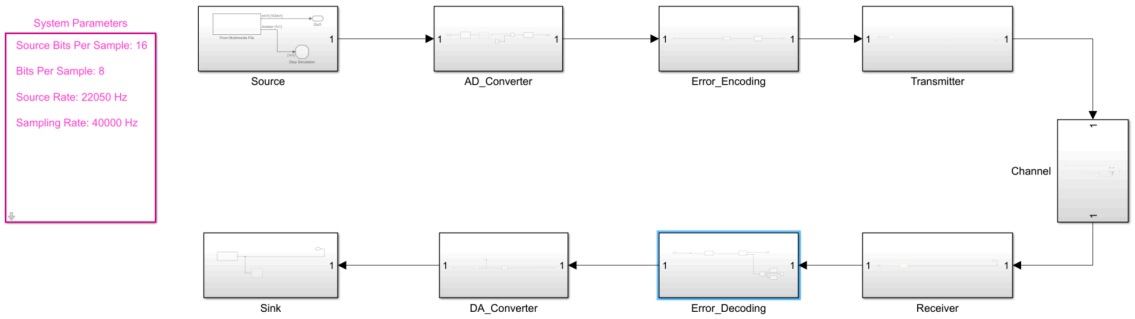


Figure 2: Complete System Block Diagram

Both the simulink and the FPGA modules were implemented based on the priorities given in figure 2. Given that audio bandwidth was the highest priority, we established the targeted sample rate at the Nyquist sampling rate of our audio bandwidth. Also, a quantization rate was chosen based on qualitative assessments of source audio quality, with the expectation that this parameter would be refined as the system development progressed.

To address the BER requirement, we adopted convolutional error encoding early in the design phase to mitigate the potential bit error rate increase due to the broader audio bandwidth. Following this, medium priority considerations, such as spectral mask and target channel requirements, were refined by adjusting parameters within the transmitter and receiver blocks.

Scenario (Teams)	High Priority				Low Priority
	Audio BW (kHz)	Target BER	Spectral Mask (kHz)	Target Channel	Delay (ms)
F	20	10^{-3}	200	β	25

Criterion	Message Transmission (kbits/s)	Transmission Reliability (bit error rate)	Channel Bandwidth (MHz)	Processing Delay (ms)
Simulink Performance	320	10^-5	0.2	27
FPGA Performance	320	10.6^-3	4.8	0.1

Figure 3: Complete System Performance Chart

The FPGA implementation directly translates from Simulink parameters, aiming for simulated results closely mirroring the FPGA solution. However, the FPGA's performance exhibited a higher BER and significantly shorter delay. This discrepancy can be attributed to substituting the FIR filter with the Zero-Order Hold block. And the BER difference arises from the reduced constraint length and traceback depth in the FPGA implementation. Also, the increase in channel bandwidth is primarily caused by the parallelized Verilog modules.

Subsystem Design

Source/Sink

Simulink Model

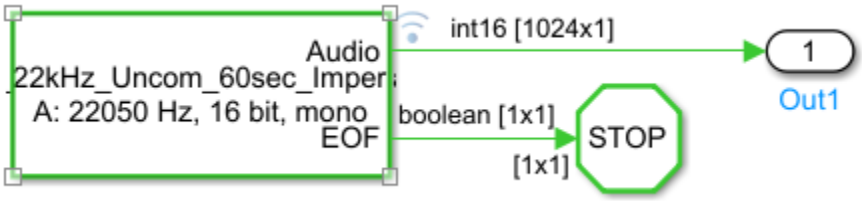


Figure 4: Source Subsystem

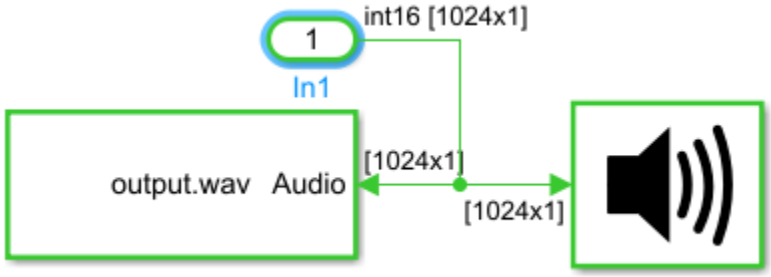


Figure 5: Sink Subsystem

For the source, we used the "From Multimedia File" block in Simulink. This block enabled us to play various .wav files at different bit widths and sample rates, which allowed us to evaluate system performance across different inputs. The block was also configured to send an end-of-file signal upon completion of the source file, which we used to terminate the simulation via a stop block.

For the sink subsystem, we employed the "To Multimedia File" block to write our results to an external .wav file, allowing for comparison in other audio analysis programs. Additionally, the audio-to-device writer block was used to enable real-time auditory output of the results.

Simulink Block	**Parameters**
From Multimedia File	Read Range: [1 inf] Samples per frame: 1024
To Multimedia File	File type: .wav format Audio Compression: none Audio data type: determine from input data

FPGA Implementation

While our simulink model had the source and sink blocks interacting directly with the ADC and DAC converters, our FPGA implemented the Wolfson WM8731 Audio Codec to interact with the 3.5mm audio jack connections on the DE1-Soc board. As a result, instead of our source and sink blocks reading or writing to a file, the FPGA equivalent blocks were the input and output devices connected to the board.

For our source block, we mainly used the microphone as an input to the DE1-SOC board as it was a quick and efficient way to test whether the output channel was working and reasonably reproduced the audio that we were inputting in real time. However, for more advanced testing, we utilized a 3.5mm to 3.5mm audio jack cable that allowed us to connect the audio output of a cell phone to the input jack of the DE1 board, Allowing us to play different audio files.

For our sink block, we used the provided stereo speakers as it was able to accurately output the audio results of our system in real time. Additionally, the volume rocker was useful as it allowed us to adjust the output to hear more minute details in the audio output that may not have been obvious at lower volume levels.

A/D and D/A Converters
Simulink Model

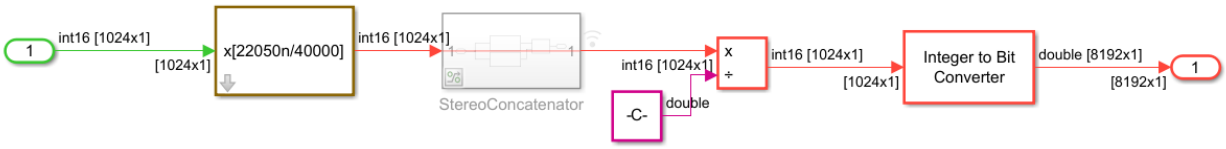


Figure 6: A/D Converter Subsystem

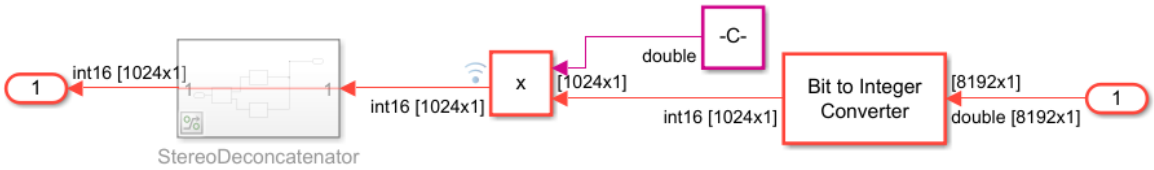


Figure 7: D/A Converter System

For the analog to digital converter, we used the FIR rate conversion block to downsample or upsample the source to our desired sampling rate of 40 kHz. This was chosen based on the Nyquist sampling rate of our assigned audio bandwidth of 20 kHz and was left unchanged throughout the design process due to the high priority nature of the audio bandwidth requirement. A division block was then used to scale the analog input to our chosen quantization bit width before it was passed through the integer to bit conversion block. The quantization bit width chosen was 16 bits as it preserved most of the original audio quality from the source while still resulting in an acceptable bit rate that, in combination with our other chosen parameters, satisfied most of our design requirements.

In the digital to analog converter block, a multiplication block was used to rescale the audio back to the original bit width of the source. As the integer to bit converter did not support multiple channels, when playing stereo sound an optional subsystem in the analog to digital block uses matrix operation blocks to concatenate the two channels into a single column matrix before passing it through the system. In the digital to analog converter, another subsystem uses similar matrix operation blocks to separate the concatenated matrix back into two channels. The simulink block diagrams for the matrix concatenator and separator can be found in the Appendices A and B respectively.

Simulink Block	**Parameters**
FIR Rate Conversion	Interpolation Factor: Desired Sampling Rate Decimation Factor: Source File Sampling Rate Allow Multirate Processing: Yes
Division Block	Dividend: Analog Integer from Source File Divisor: 2^(Source bit depth - Quantization bit width)
Integer to Bits Converter	Number of Bits Per Integer: Quantization bit width Treat input values as: signed Output Bit Order: MSB first Output data type: double
Bits to Integer	Number of Bits Per Integer: Quantization bit width

	Input Bit Order: MSB first Treat results as: signed Output data type: inherit via internal rule
Multiplication block	Input 1: integer result after bit to integer conversion Input 2: 2^(Source bit depth - Quantization bit width)

FPGA Implementation

As our simulink model had a higher system delay than we would like, we decided to reduce the quantization from 16 bits to 8 bits in order to decrease the number of clock cycles required to send bits serially from one module to another. Also, it was qualitatively determined that using 8 bits instead of 16 bits led to minimal losses in audio quality. Another difference between our simulink and FPGA implementation was that rather than using the FIR rate change block, we decided to use a zero order hold method. This was done as the FIR rate change block relied on a conversion factor between the original and desired sampling rates to down sample the signal and could be prone to disruptions caused by hardware delays in the Wolfson WM8731 Audio Codec.
To implement the ADC converter on the FPGA, we first connected only the 8 most significant bits of the 24 bit wide audio bus in order to achieve our quantization level. From here the zero order hold was implemented using a flip flop circuit with the fast clock for the clock input and the slow clock for the enable signal with additional logic to only stay high for one cycle of the fast clock. Our ADC block also controlled the read ready/enable signals for the audio codec.

To implement the integer to bit converter simulink block, we used a serializer circuit that would update its input on the rising edge of every slow clock and once updated, would serially output one of the bus bits on every rising edge of the fast clock. Figures [8] and [9] show the HDL block diagrams for these modules:

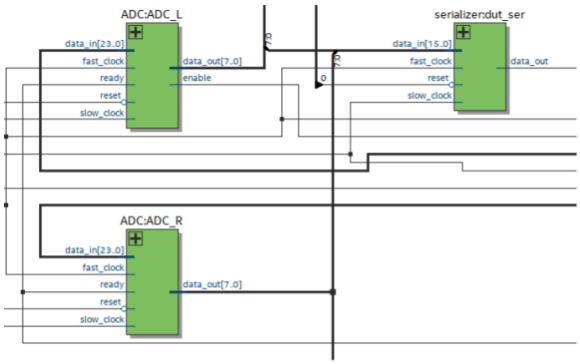


Figure 8: ADC and serializer block diagram

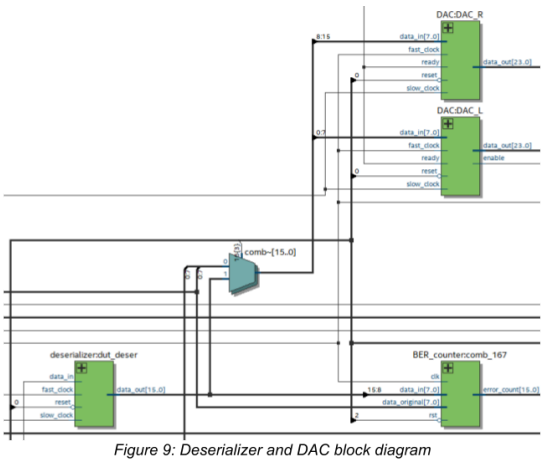


Figure 9: Deserializer and DAC block diagram

Module	**Parameters**
ADC Converter	Input: 24 bus Output: Quantization bit width bus Output rate: 40 kHz Quantization: 8 bits
DAC	Input: Quantization bit width bus

Converter	Output: 24 bit bus Output rate: 40 kHz Quantization: 8 bits
Serializer	Input: Quantization bit bus Output: 1 serial bit Fast clock: 50 MHz Slow clock: 40 kHz Quantization: 8 bits
Deserializer	Input: 1 serial bit Output: Quantization bit width bus Fast clock: 5 MHz Slow clock: 40 kHz Quantization: 8 bits

Error Correction
Simulink Model

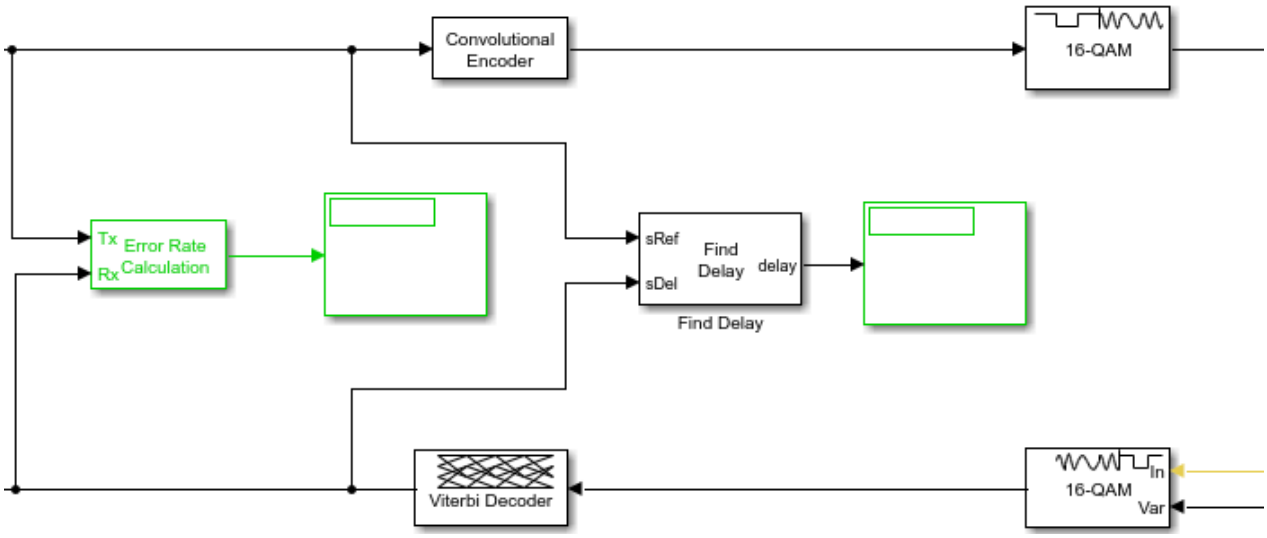


Figure 10: Encoder/Decoder & Modulation/Demodulation System

The system incorporates convolutional encoding and Viterbi decoding, based on reviews of related literature, such as "Convolution Coding and Applications: A Performance Analysis under AWGN Channel"[1] and "Comparative Performance Analysis of Block and Convolution Codes"[2]. These studies consistently underscore the efficacy of convolutional codes in error detection and correction, particularly suited for high-demand real-time applications such as audio channels.

The convolutional encoder operates at a ½ information rate with a constraint length of 7, chosen to optimize both computational efficiency and error correction capability. While convolutional encoding is susceptible to burst errors, this challenge can potentially be mitigated through interleaving, a consideration for future updates in our system design. Moreover, the Viterbi decoder is designed with a traceback depth of 48, a parameter chosen based on recommendations provided by MathWorks[5].

For modulation, the system employs 16-QAM due to its high data rate capability, crucial for efficient transmission of audio signals. However, its sensitivity to noise, due to smaller symbol distances and the complexity involved in recovering phase and amplitude, presents potentially increased BER and additional design considerations.

In evaluating QAM techniques, we conducted an analysis encompassing hard, soft, and unquantized decision-making methods (see Appendix C), alongside literature such as "Comparison Between Viterbi Algorithm Soft And Hard Decision Decoding"[6]. Our findings as well as the research indicate that unquantized decision-making exhibits superior performance; however, its applicability in real-world scenarios is constrained. As a result, we opted for soft decoding as it achieves a balanced compromise between performance and practical feasibility in our system design.

Simulink Block	**Parameters**
QAM Modulator	M-ary Number: 16 Input Type: Bit

	Constellation Ordering: binary Normalizing Method: Min. distance between symbols Minimum Distance: 2 Phase Offset (rad): 0
QAM Demodulator	M-ary Number: 16 Output Type: Bit Decision Type: Log-likelihood ratio Noise variance source: Port Constellation Ordering: Binary Normalizing Method: Min. distance between symbols Minimum Distance: 2 Phase Offset (rad): 0
Convolutional Encoder	Trellis Structure: poly2trellis(7, [171 133]) Operation Mode: Continuous
Viterbi Decoder	Trellis Structure: poly2trellis(7, [171 133]) Decision Type: Unquantized Traceback Depth: 48 Operation Mode: Continuous

FPGA Implementation

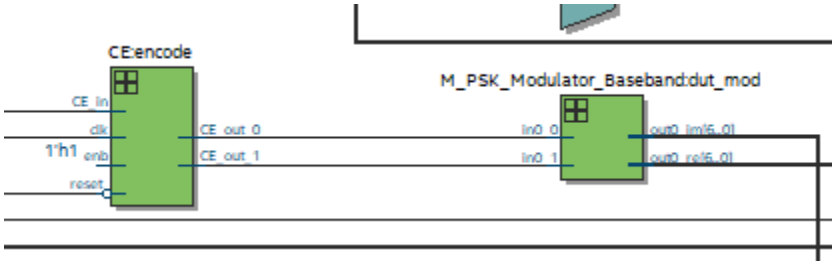


Figure 11: Encoder/Decoder Block Diagram

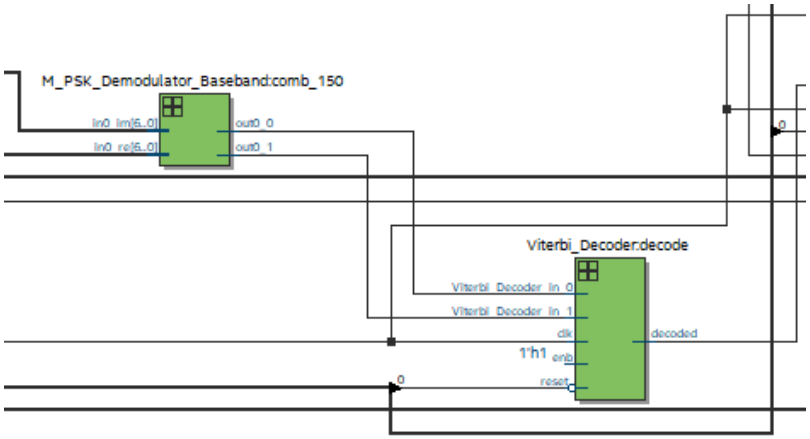


Figure 12: Modulation/Demodulation System

The convolutional encoder in our FPGA operates at a ½ code rate with a constraint length reduced to 3, a change from the previous length of 7. This adjustment aimed to minimize system delay and maintain a satisfactory BER. In addition, the decision on the generator polynomial was guided by insights from "Goodness Analysis of Generator Polynomial for Convolution Code with Varying Constraint Length"[3], which identified the [7 5] polynomial combination as the most effective among 12 alternatives considered for our specific design. The encoder was implemented using shift registers, as shown in Appendix D.

The Viterbi decoder is configured with a traceback depth of 16, modified from 48, based on guidelines provided in "A truncation depth rule of thumb for convolutional codes"[4]. Initially, adhering to the 5 multiple rule suggested a depth of 15, but further optimization using two's complement representation led us to adjust it to 16 for enhanced FPGA performance. The implementation on the FPGA made use of Matlab's HDL coder, with configuration of input and output ports tailored to our design. The generated module was integrated to our system without having to make any modification.

The modulation scheme has also been revised from 16-QAM to QPSK. This change was driven by considerations related to the reduced constraint length in the encoder, where higher-order modulations posed a risk of exceeding BER thresholds specified in our design requirements (see Appendix D). Our FPGA implementation mapped the 4 possible states of 2-bit inputs to a 6-bit bus representing ±1, with a lookup table incorporated for signal demodulation. Additionally, our demodulation module included overflow detection mechanisms to manage extreme noise levels from the channel, where positive overflow was assigned to the maximum of the signed 6-bit bus, while negative overflow was assigned to its minimum value.

FPGA Block	**Parameters**
QPSK Modulator	Input Type: Bit Constellation Ordering: binary Phase Offset (rad): pi/4
QPSK Demodulator	Output Type: Bit Decision Type: Hard Decision Constellation Ordering: Binary Phase Offset (rad): pi/4
Convolutional Encoder	Trellis Structure: poly2trellis(3, [7 5]) Operation Mode: Continuous
Viterbi Decoder	Trellis Structure: poly2trellis(3, [7 5]) Decision Type: Hard Decision Traceback Depth: 16 Operation Mode: Continuous

Transmitter/Receiver
Simulink Model

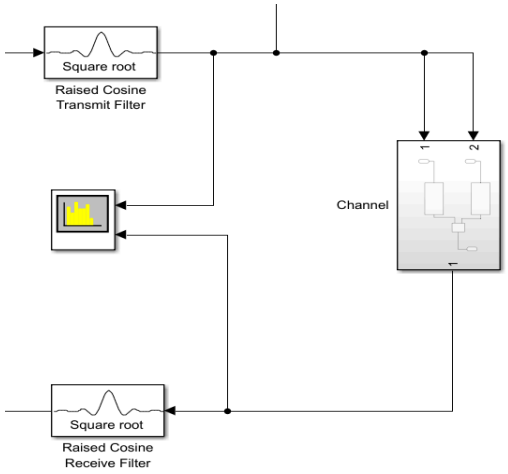


Figure 13: Transmitter/Receiver System

In this subsystem, we opted for the "raised cosine transmit filter" block for transmission, paired with a square-root raised cosine filter as a matched filter at the receiver. This configuration achieves a raised-cosine spectral shape that meets the Nyquist criterion, effectively mitigating inter-symbol interference (ISI)[7]. Furthermore, it allows for controlled bandwidth expansion by modifying the roll-off factor. However, it's worth noting that the square-root raised cosine filter introduces some delay due to its inherent filtering process[8].

An alternative we considered was employing rectangular filters and matched filters, which would offer lower computational complexity. However, this approach would not meet the requirement for a 200kHz spectral mask due to the inability to control bandwidth. Given that prioritizing delay time was less critical in our decision-making process, we concluded that using square-root raised cosine filters would be the optimal choice for our system.

The choice of parameters for the root-raised cosine transmit filter is critical to optimizing the performance of our system. By observing the spectrum analyzer in Simulink, we chose a roll-off coefficient of 0.8 to ensure that our transmit spectrum is below the 200kHz spectral mask boundaries. The filter span of 8 symbols ensures that the filter has sufficient length to capture the necessary pulse shaping characteristics and reduce ISI. By sampling 8 outputs per symbol, we achieve the required sample rate, which is approximately 1MHz. Performing single-rate processing avoids having multiple sampling rates throughout the system.

To implement this on FPGA, we made adjustments to accommodate the limited DSP blocks available. We reduced the filter span from 8 symbols to 4 symbols and decreased the sampling rate from 8 samples per symbol to 5 samples per symbol. These parameter changes effectively reduced the number of filter taps, resulting in significant memory savings.

Simulink Block	**Parameters**
Raised Cosine Transmit Filter	Filter Shape: Square root Roll-off factor: 0.8 Filter span in symbols: 4 Output samples per symbol: 5

	Linear amplitude filter gain: 1 Input processing: Columns as channels (frame based) Rate options: Enforce single-rate processing
Raised Cosine Receive Filter	Filter Shape: Square root Roll-off factor: 0.8 Filter span in symbols: 4 Output samples per symbol: 5 Decimation factor: 5 Decimation offset: 0 Linear amplitude filter gain: 1 Input processing: Columns as channels (frame based) Rate options: Enforce single-rate processing

FPGA Implementation

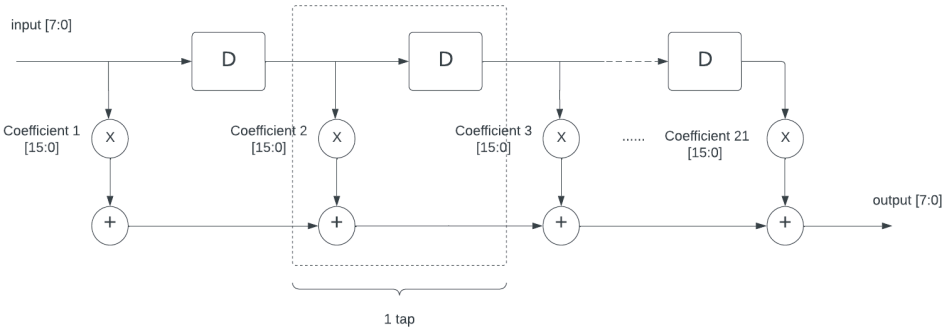


Figure 14: Transmitter/Receiver Block Diagram

As depicted in the figure above, implementing square-root raised-cosine (SRRC) filters on the FPGA closely resembles the process of implementing an FIR filter. Initially, the filter coefficients were exported from the Simulink model. These coefficients, originally in floating-point format, were converted to fixed-point numbers to facilitate computations on the FPGA.

To implement the filter, we considered two possible algorithms: Multiply-Accumulate (MAC) and Distributed Arithmetic (DA). We chose the Multiply-Accumulate (MAC) approach for our Square Root Raised Cosine (SRRC) filter due to its superior performance and flexibility[8]. MAC operations handle high sample rates efficiently, making them ideal for high-speed digital signal processing applications like our SRRC filter.

Resource utilization also played a critical role in our decision. While Distributed Arithmetic (DA) can be resource-efficient, it becomes less so with increased filter complexity[9]. Our initial 33-tap filter design was too resource-intensive, necessitating a reduction to 21 taps. The MAC approach uses dedicated multipliers and accumulators more effectively, balancing resource use to meet performance requirements without exceeding hardware limits.

Finally, the MAC design is more scalable and easier to implement than DA. It supports future upgrades and modifications with minimal reengineering. Overall, the MAC approach provides the necessary efficiency, flexibility, and scalability for our communication system.

Channel
Simulink Model

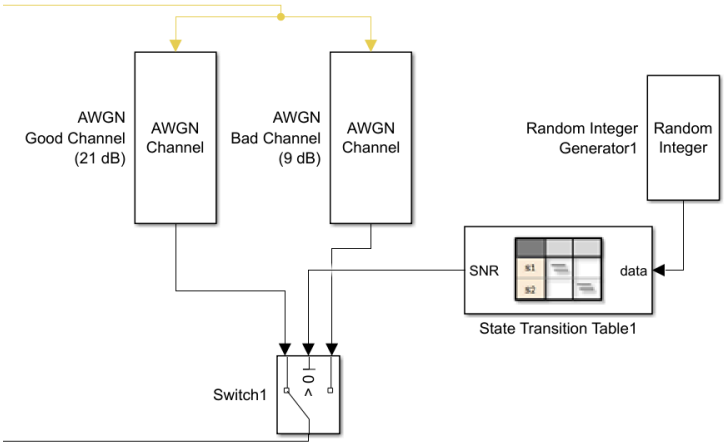


Figure 15: Gilbert Fading Channel System

The Gilbert Channel model simulates bursty errors by transitioning between Good and Bad states. To model the Gilbert Fading Channel, we combined a state transition table and a random integer generator block to determine the state transitions. First, the random number generator generates an integer between 0 and 99, and it is used to ensure that the transitions between "Good" and "Bad" states occur according to fixed probabilities. By inputting the integer into the “state transition table” block, the block generates the current state. The output of the state transition table is connected to a switch, which determines if the AWGN channel is at the good state (SNR is 21 dB) or the bad state (SNR is 9 dB).

Simulink Block	**Parameters**
Random Integer Generator	Set size: 100 Sample time: 0.0001 Samples per frame: 1
State Transition Table	Output: SNR(good) = 1, SNR(bad) = -1
Switch	Threshold: input > 0
AWGN channel (Good)	SNR: 21 Input power: 1
AWGN channel (Bad)	SNR: 9 Input power: 1

FPGA Implementation

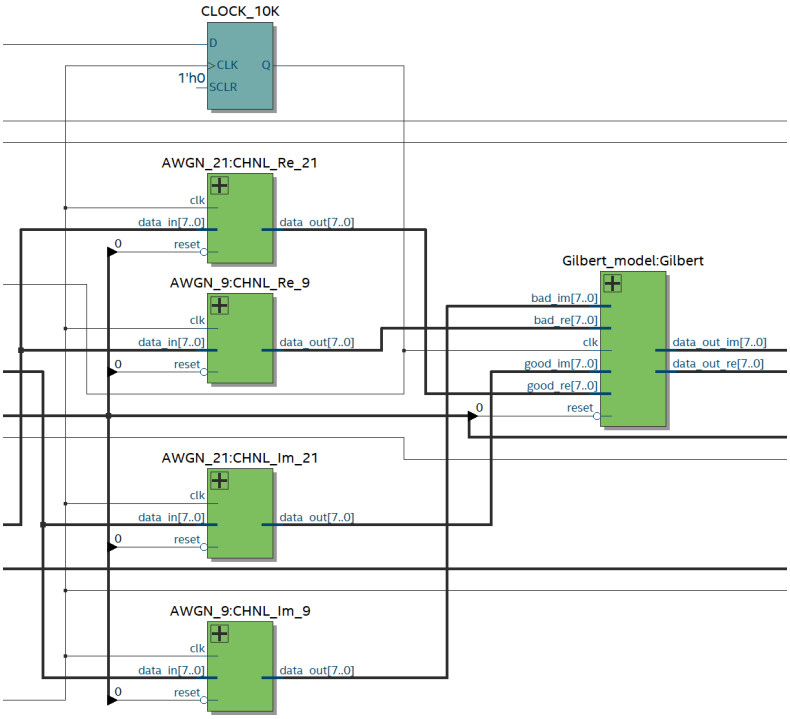


Figure 16: Channel Block Diagram

The channel subsystem consists of two parts: a state-transition machine corresponding to the Gilbert Model, and the AWGN noise generator.

The state-transition machine was implemented corresponding to the Simulink model. First, we used the Linear Feedback Shift Register (LFSR) to randomly generate an 8-bit number. Then, we compute threshold values based on the given probabilities, and then compare the random number with the threshold value to control the state transition.

For the AWGN channel hardware implementation, we had considered generating the values in real time using the Box-Muller algorithm and Chebyshev approximations. However as these implementations already required storing coefficients in memory, we decided that the better option was to generate the gaussian noise in advance and store them in the memory blocks on the FPGA. This implementation is beneficial in that it allows us to visualize the values before loading into memory and gives us a way to compare and verify whether our hardware implementation is working as expected.

To implement this on the FPGA, we used a matlab script to generate gaussian noise values and output it in an MIF format that could be stored in M10K blocks on the DE1 board. The complete Matlab script can be found in Appendix G. We then used a LFSR circuit connected to the address port to randomly access values in memory and had a top level module to add the gaussian noise (output port of memory) to our input data before passing it through the channel.

Verification

Source/Sink

Simulink Model

For the source and sink verification, an AWGN channel was added between the two subsystems with a spectrum analyzer connected to both sides. The resulting system is shown below:

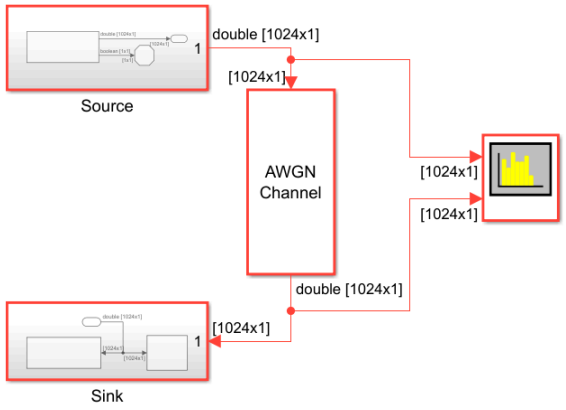


Figure 17: Verification System for Source/Sink

The frequency spectrum for SNRs of 9dB and 100dB are shown in figures 18 and 19 respectively:

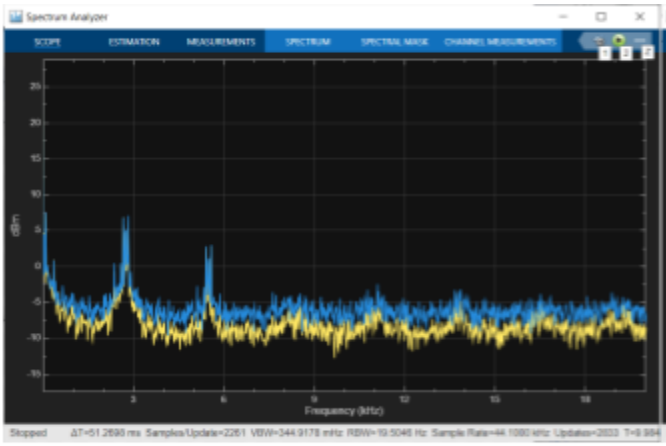


Figure 18: Frequency Spectrum of Source and Sink at SNR of 9 dB

Figure 19: Frequency Spectrum of Source and Sink at SNR of 100 dB

As expected in figure 18, we can see that the frequency spectrum changed after passing through the AWGN noise channel while in figure 19, at a higher SNR of 100 dB, both frequency spectrums match perfectly and are right on top of each other.

FPGA Implementation

To verify that our source and sink were capable of sending and receiving sound, we reused our module from the FPGA starter assignment and verified that the source block, which was either the wired microphone or cellphone, was able to input an audio signal and that our speaker, which was our sink block, was able to receive and play the audio outputted to it. In the unlikely event that both source and sink blocks may not have been functioning, we had alternative sources such as a laptop with an audio jack, and alternative sinks such as earbuds to isolate and determine the non-functioning component.

A/D and D/A Converter

Simulink Model

For the A/D and D/A verification, an AWGN channel was placed in between the two subsystems and an error rate calculation block was attached to either side of the channel to measure the bit error rate. The resulting system is shown below:

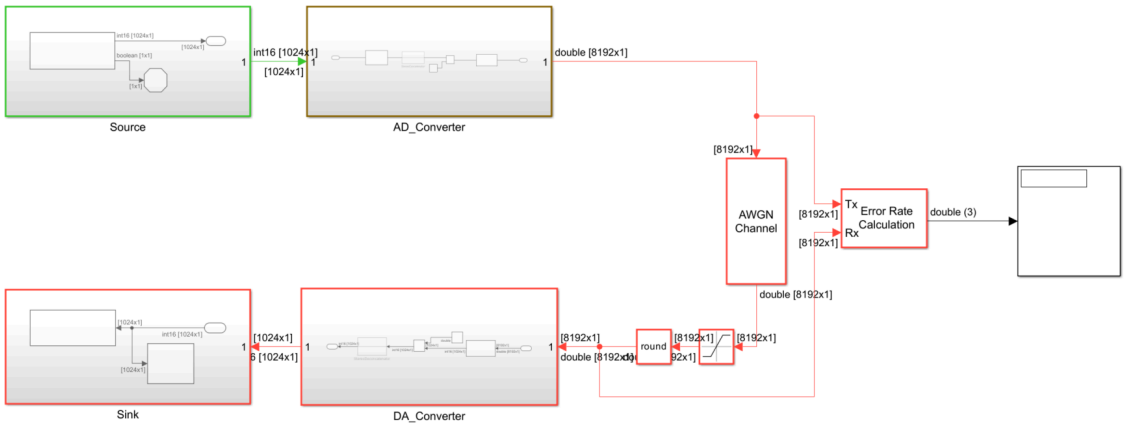


Figure 20: Verification system for AD and DA converter

Because the DA converter expects a binary input while the AWGN adds a double value to the signal, the output is sent through a saturation block to cut off the output between zero and one and then rounded with the round block. The resulting bit error rates (BER) when the channel has an SNR of 9 dB and 21 dB are shown in the table below:

SNR (dB)	Bits Sent	Bits with Error	BER
9	1.92×10^7	1.52×10^6	0.0794
21	1.92×10^7	0	0.000

As expected, the channel with an SNR of 9 dB has some bit errors introduced, however at a SNR of 21 dB, there are zero bit errors while passing through the channel.

FPGA Implementation

For verification of our A/D converter, we used a testbench in modelsim to verify that the data out is only being updated at the rising edge of our slow clock (where the slow clock was our chosen sampling rate) and that the output was our chosen quantization of 8 bits. The results can be seen in the figure below:

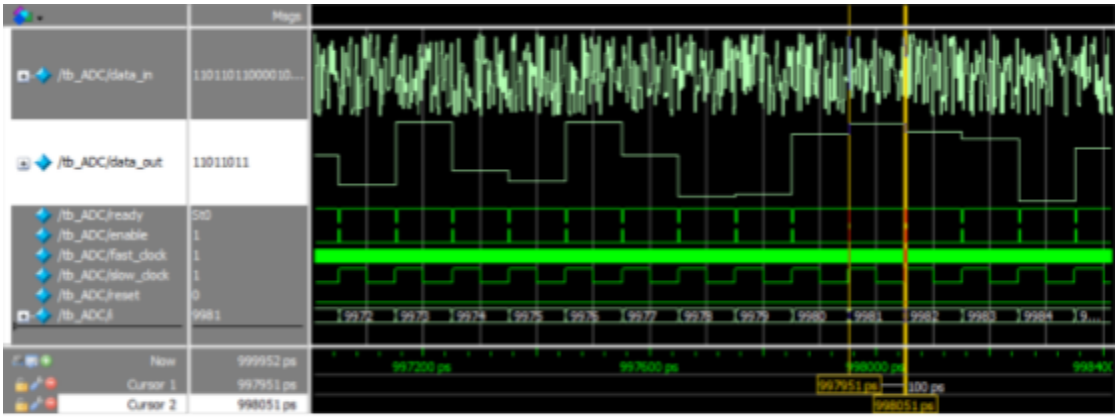


Figure 21: ADC Verification Waveform for sampling rate

From the above testbench, we can verify that while the data_in changes at a significantly higher rate, our output only changes on the rising edge of slow_clock, ensuring that when we connect a 40kHz clock, our sampling rate will be fixed. A second testbench verifies that the input to the A/D converter is accurately reproduced at a lower quantization level. In the waveform below, the 8 bit output is scaled up in the modelsim window allowing use to verify our results against the original 24 bit input:

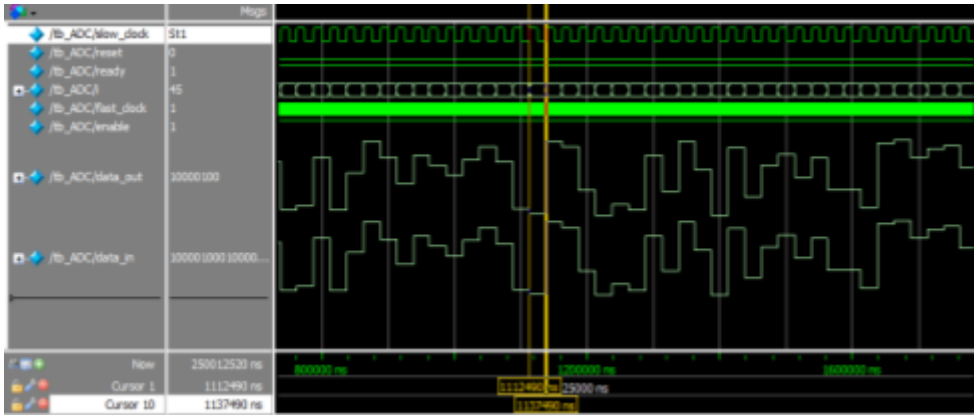


Figure 22: ADC data verification waveform

Similarly, for the D/A converter we used a testbench in modelsim to verify that the input data is scaled appropriately and outputs at the original 24 bit width of the audio codec:

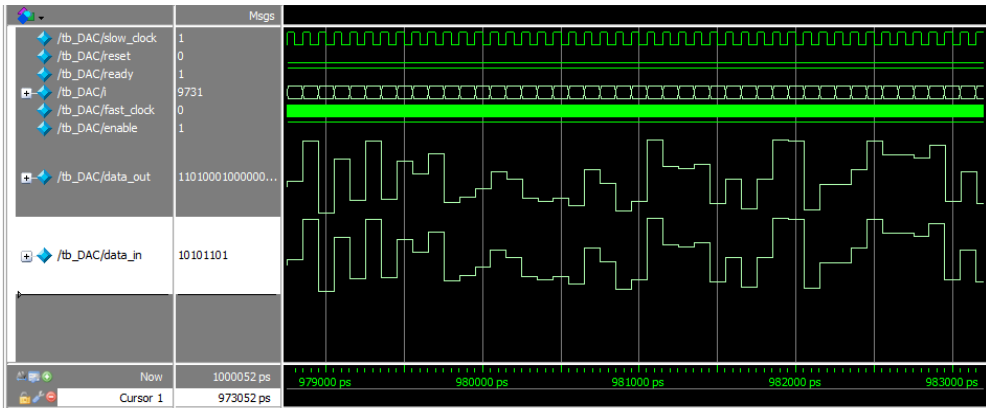


Figure 23: DAC data verification waveform

Error Correction

Simulink Model

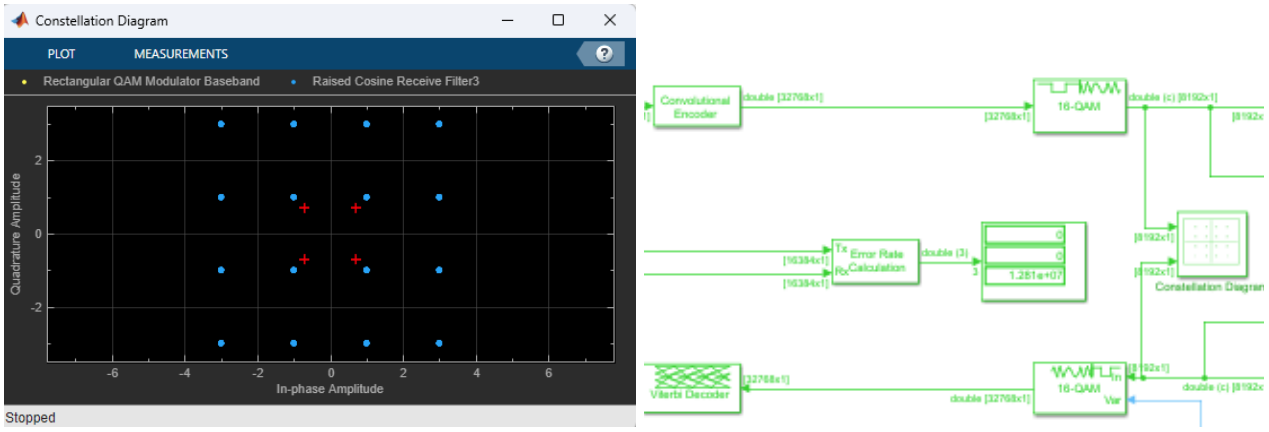


Figure 24: Constellation Diagram and Bit Error Rate for 100dB

There are no errors detected given 100dB and 9dB channel noise. The constellation diagram for 16 QAM modulation, therefore is expected to show the 16 symbols without any variance between the channels. The simulated diagram accurately depicts such a case.

As the error rate increases, the constellation diagram shows variance within its diagram as expected. For example, given that the channel noises are 2dB and 9dB:

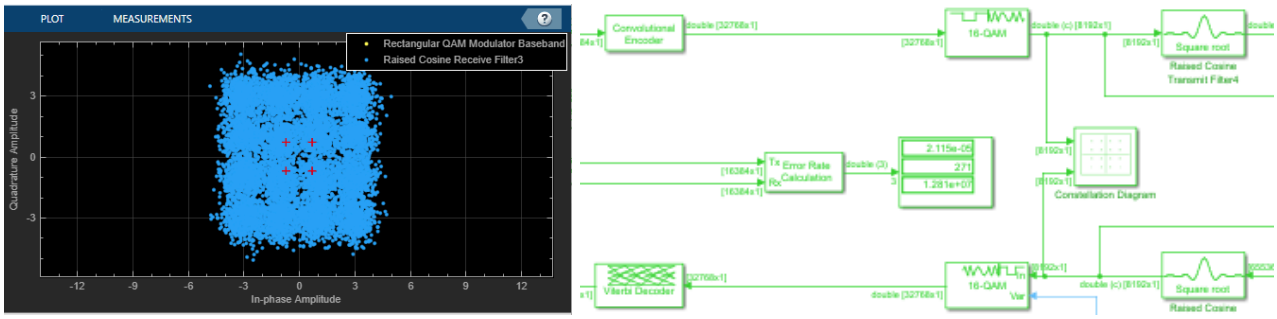


Figure 25: Constellation Diagram and Bit Error Rate for 2dB and 9dB

As the error rate increases, we can observe both increased variances in the constellation diagram and higher BER.

FPGA Implementation

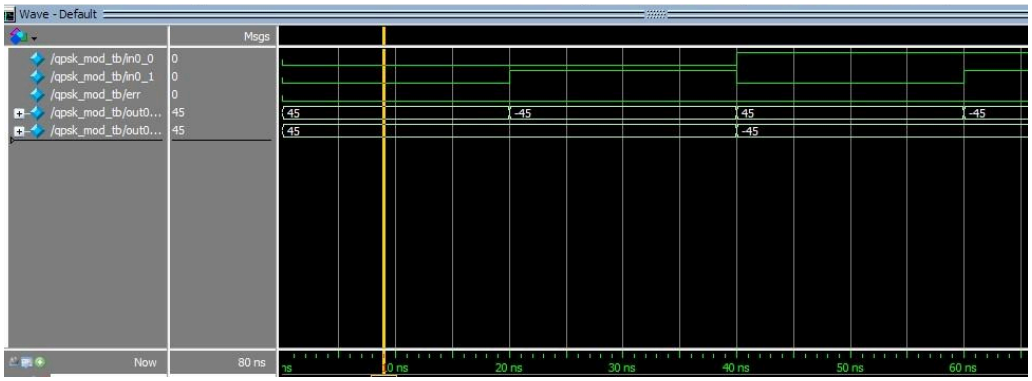


Figure 26: Modulation Data Verification Waveform

The proposed FPGA implementation of QPSK modulation represents each symbol in the constellation to a specific 6-bit binary bus. The symbol for 1 is encoded as 0101101 and -1 as 1010011, which translate to binary values 45 and -45 respectively.

Following the QPSK modulation scheme, the expected outputs for the bit combinations [0 0], [0 1], [1 0], and [1 1] are mapped to the constellation points [1 1], [-1 1], [1 -1], and [-1 -1]. These mappings then directly correlate to the coordinates [45 45], [-45 45], [45 -45], and [-45 -45], as illustrated in the waveform provided above.

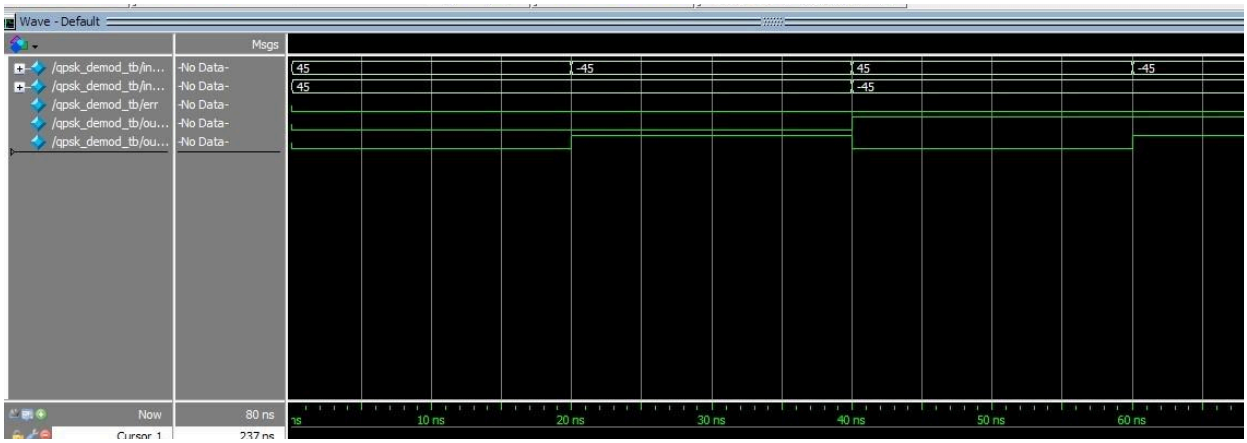


Figure 27: De-Modulation Data Verification Waveform

Conversely, the demodulator interprets the received constellation points [45 45], [-45 45], [45 -45], and [-45 -45], translating them back into their corresponding bit combinations: [0 0], [0 1], [1 0], and [1 1], as depicted in the waveform above.

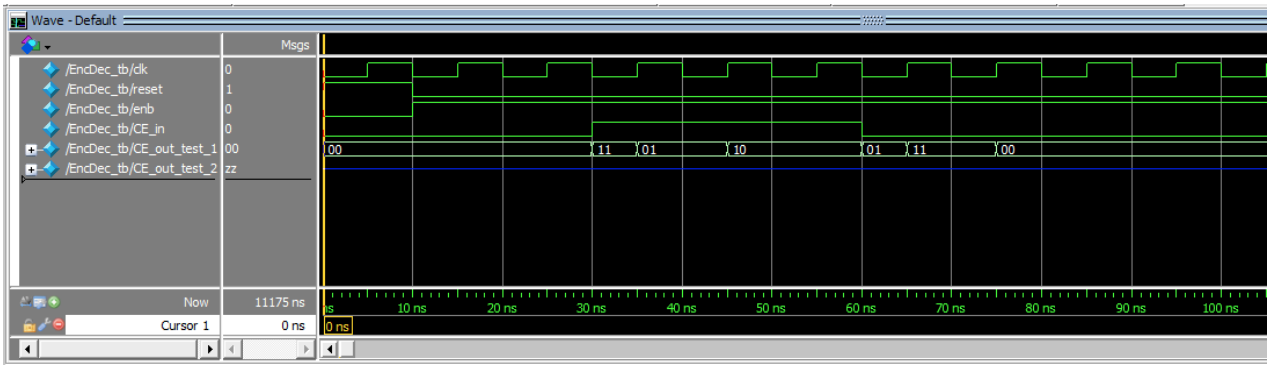


Figure 28: Convolutional Encoding Verification Waveform 1

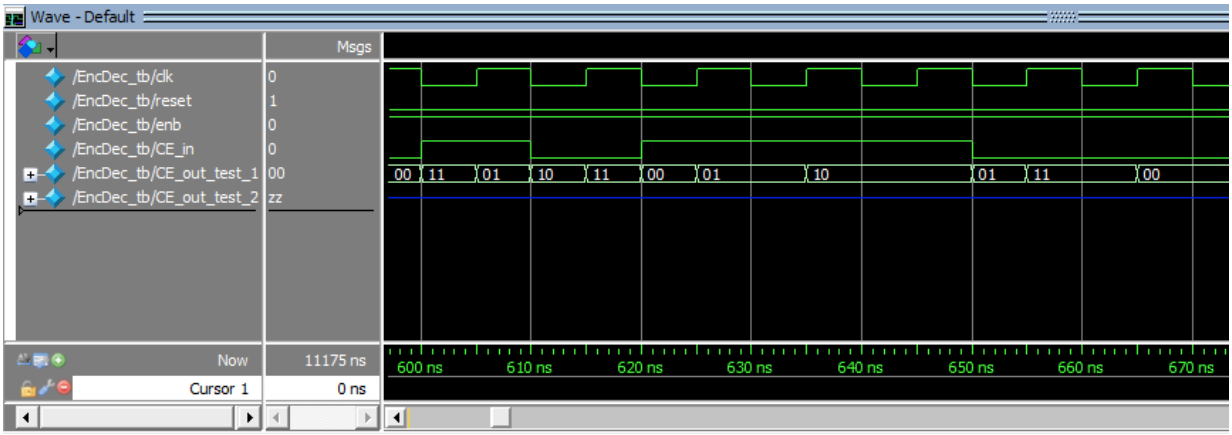


Figure 29: Convolutional Encoding Verification Waveform 2

The FPGA implementation of convolutional encoding employs shift registers with a constraint length of 3 and a code rate of 1/2, determined by the generator polynomial [7 5]. The design aims to mimic the behavior of a 2-bit state machine, as described in Appendix D.

In the provided waveform, the initial input sequence is 011100. According to the convolutional encoding process specified by the generator polynomial [7 5], the expected output sequence should be 00 11 01 10 01 11. Similarly, for the input sequence 101110, the expected output sequence is 11 10 00 01 10 01, based on the same encoding scheme, which can be observed above.

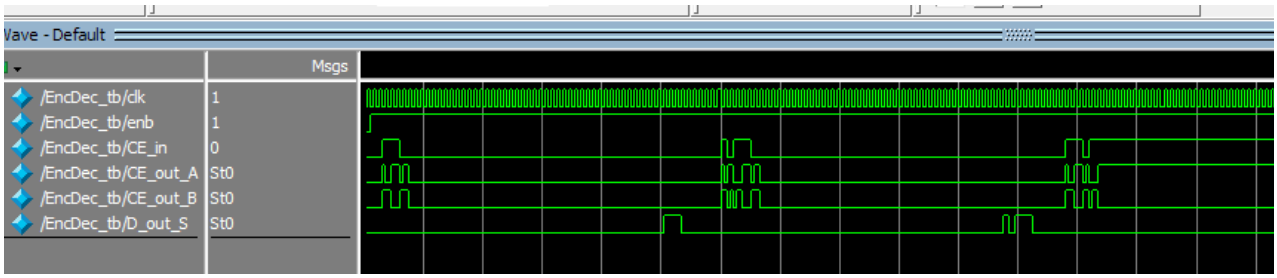


Figure 30: Error Correction Verification Waveform

To validate the functionality of the Viterbi decoder, serialized bits were inputted into an encoder directly connected to the decoder, enabling exhaustive testing. As indicated by the waveform above, the input signals are recovered after a delay, which aligns with expectations. This delay occurs because the Viterbi decoder's traceback depth necessitates the accumulation of an error matrix to a certain extent before the correct decoded output can be determined and produced.

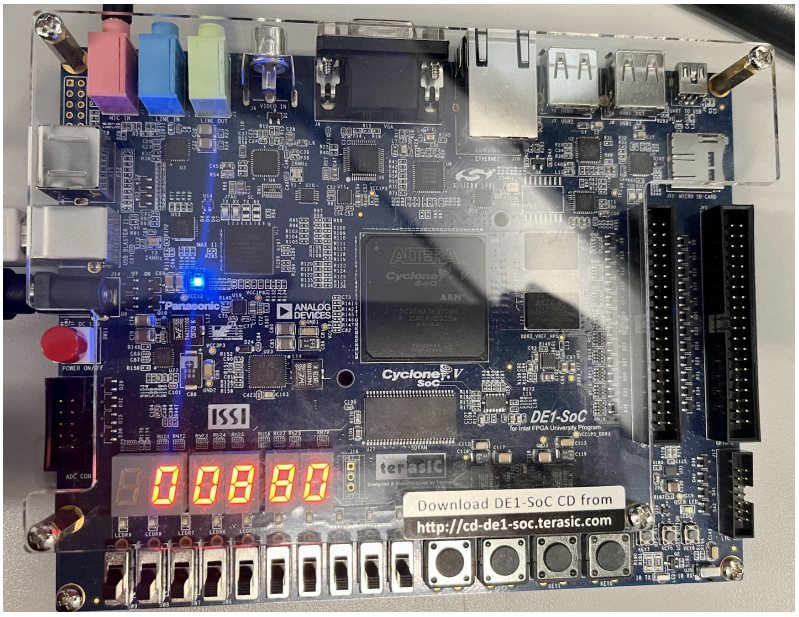


Figure 31: BER Status

The Bit Error Rate (BER) was calculated using a counter that accounts for the total system delay to ensure accurate data comparison. As depicted in the figure, the measured BER is 880. To manually determine the BER for this instance, we used the formula (65535 samples * 8 bits per sample) divided by the output BER, resulting in a BER value of approximately $1.678_{E}-3$ which exceeds the required BER by a small margin.

Transmitter / Receiver

Simulink Model

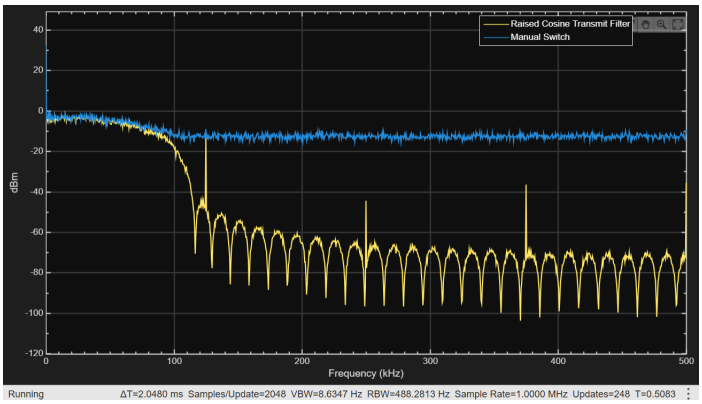


Figure 32: Frequency domain signal for 9dB SNR

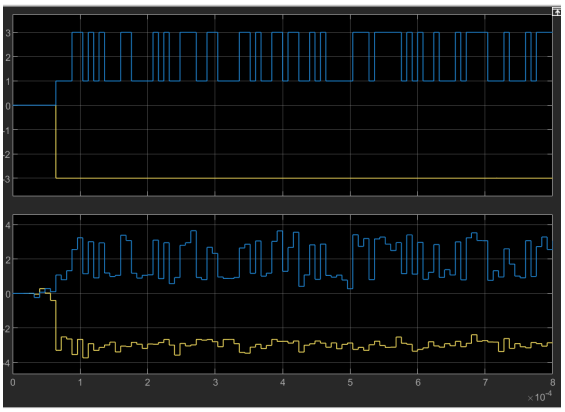


Figure 33: Time domain signal for 9dB SNR

Comparing the frequency domain signals from the output of the transmitter to the input of the receiver for 9dB SNR, we can observe that our spectrum fits under the 200kHz spectral mask. Meanwhile, in the time domain, the signal after the channel is noisy as expected.

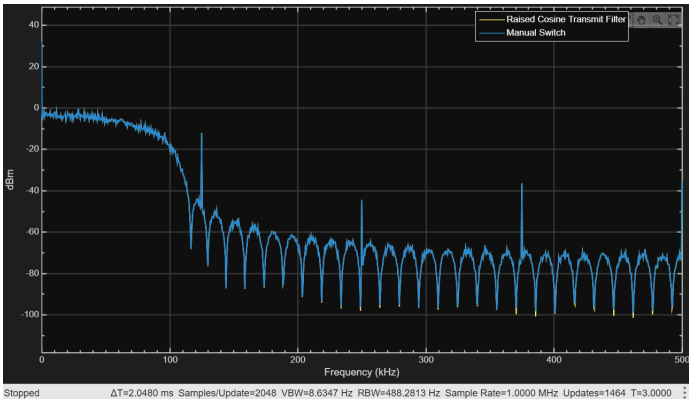


Figure 34: Frequency domain signal for 100dB SNR

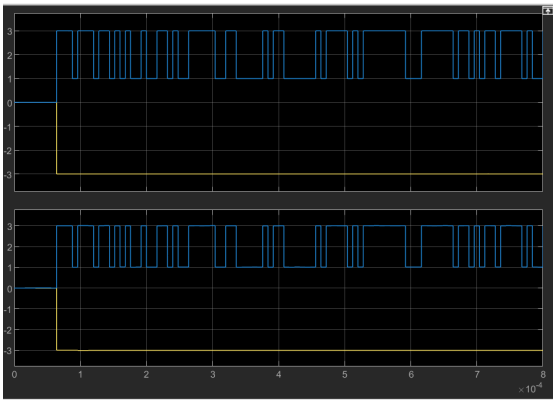


Figure 35: Time domain signal for 100dB SNR

For 100dB SNR, the frequency and time domain signals are both identical at the output of the transmitter and the input of the receiver. These plots meet our expectations, since 100dB SNR has much smaller noise than 9dB SNR. The average transmitted signal energy is 1 Watt as expected.

FPGA Implementation

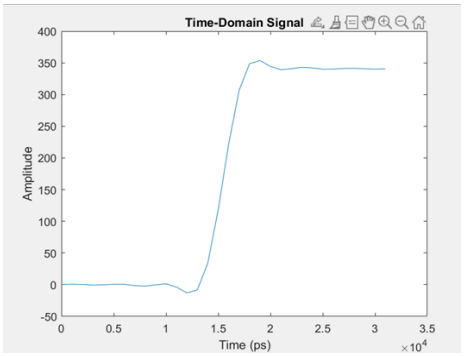


Figure 36: Time domain transmitted signal

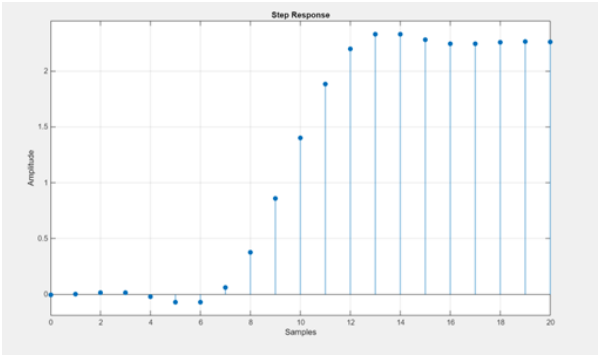
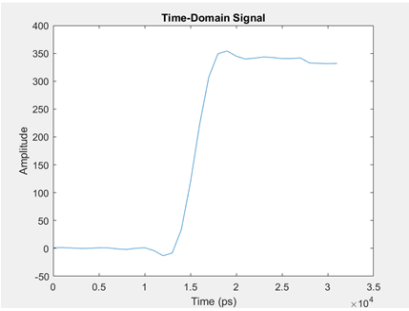
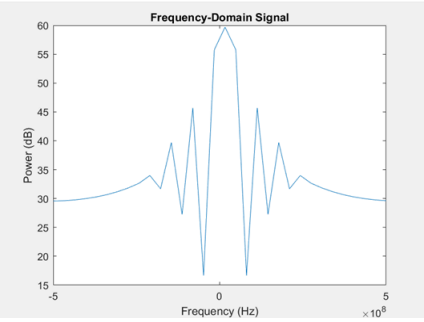


Figure 37: Step response of the SRRC transmit filter





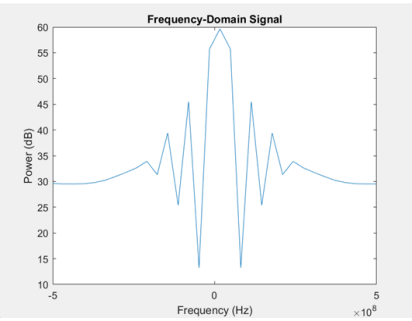


Figure 38: Time domain received signal *Figure 39: Frequency domain transmitted signal* *Figure 40: Frequency domain received signal*

To verify our transmitter and receiver design, we collected data from testbench simulation with a step input. Then, we plotted the time domain signals at the transmitter and the receiver side in MATLAB. The frequency domain signals can then be computed by converting the time domain signals. The MATLAB code for plotting the signals is included in Appendix E.

In the time domain, the transmitter output signal is the same as expected, compared to the step response generated in the Simulink model. The received signal, which has noise added to the transmitted signal, is not as smooth as the transmitted signal. Meanwhile, when we compare the frequency domain signals, the difference is marginal, due to the 21 dB channel noise does not alter the transmitted signal a lot. The received signal in the frequency domain should have a broader spectrum, caused by the noise and distortions introduced by the channel.

Channel

Simulink Model

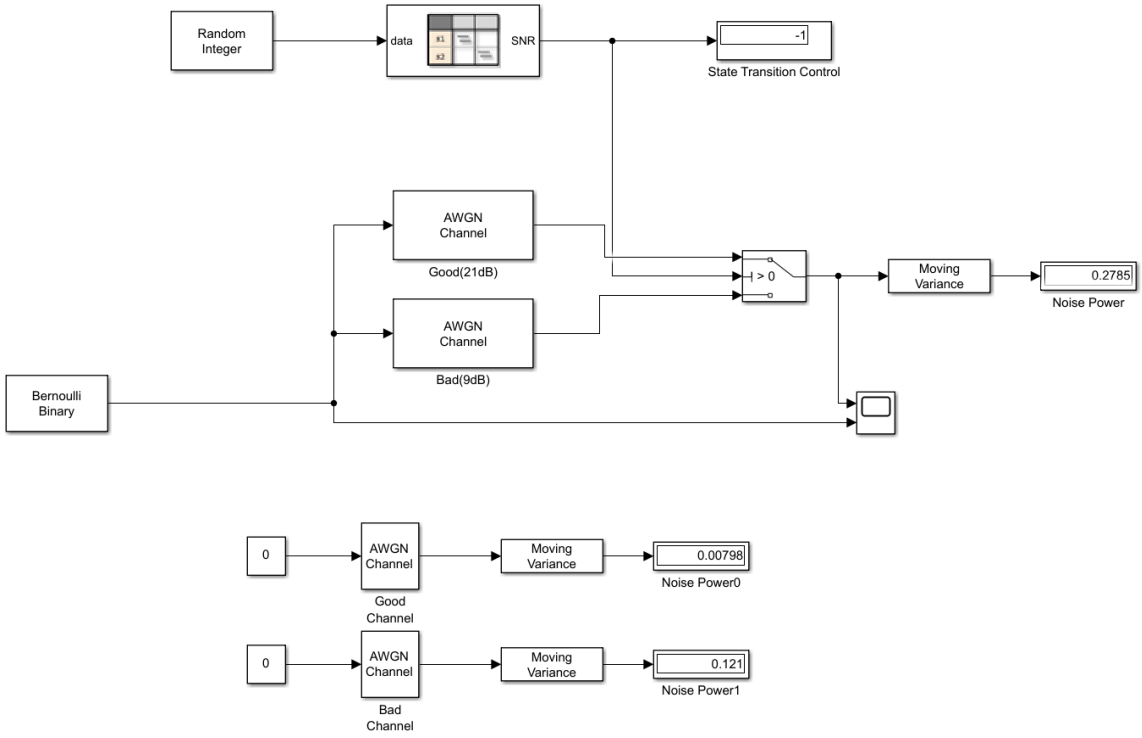


Figure 41: Verification system for Gilbert Channel

Using the formula $\sigma^2 = \frac{P_{\text{signal}}}{\text{SNR (linear)}}$, we can calculate the variance of the AWGN channels. By multiplying each variance with the expected probability, we can estimate the variance for the Gilbert Channel. From the simulation in figure 20, the variance of the Gilbert channel is 0.27, as expected.

Meanwhile, the average number of transitions in the Good/Bad states can be calculated using the given transition probabilities (P_GB=0.03, P_BG=0.25), as shown in Appendix H.

FPGA Implementation

```
# At time              999860, good_count =       23385, bad_count =        1611
# At time              999900, good_count =       23386, bad_count =        1611
# At time              999940, good_count =       23387, bad_count =        1611
# At time              999980, good_count =       23388, bad_count =        1611
# At time             1000020, good_count =       23389, bad_count =        1611
# At time             1000060, good_count =       23390, bad_count =        1611
# ** Note: $stop    : C:/UBC/ELEC391/ProjectFPGA/tb_Gilbert.v(30)
#    Time: 1000100 ps  Iteration: 0  Instance: /tb_Gilbert
# Break in Module tb_Gilbert at C:/UBC/ELEC391/ProjectFPGA/tb_Gilbert.v line 30
```

Figure 42: Gilbert Model Testbench Output

To verify that the Gilbert model was implemented correctly on FPGA, we designed a testbench counting the number of each state in a given timeframe (see Appendix F). The probability of staying in the Good state is 0.93, and 0.07 for the bad state. These values are close to our expectations, which are the computed steady-state probabilities.

To verify that our AWGN channels had the correct distribution, we used a testbench that set the channel input to zero and then recorded the output noise coming from the channel. The recorded data was exported as a text file that could then be imported to Matlab. Using the histogram function, we could visually compare the distributions of the generated values and channel noise to verify our design. The results are shown in figures [43] and [44] below:

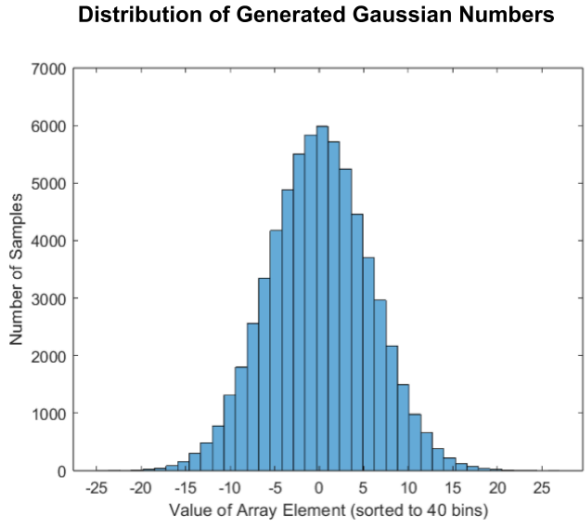


Figure 43: Distribution of Generated Values from Matlab Script

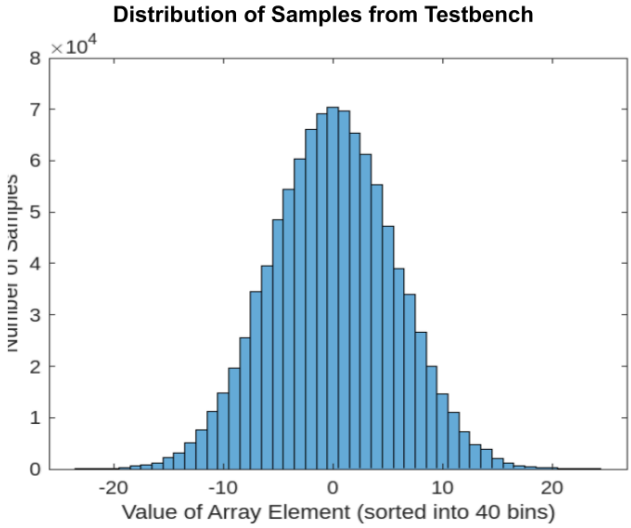


Figure 44: Distribution of Samples from Modelsim Testbench

As shown from the figures above, the two distributions match verifying that our FPGA implementation is working as expected.

Team Contribution

Aaron Loh

For this project I was responsible for implementing the A/D converter, D/A converter, as well as the AWGN channel. This meant that I was responsible for researching the appropriate blocks to be used in our simulink model, understanding how the different parameters affected the performance of each block, and also determining how the subsystem that I was responsible for would interact with and affect the performance of other blocks. Similarly, for the FPGA implementation side I was responsible for designing modules that would match the specifications of our simulink model, and coordinating the type of data and timing of transmissions to other blocks. Finally, I also worked on all the written sections for my respective subsystems and also wrote a majority of the system overview section used in reports 1 and 2. Overall, I successfully completed both the simulink model as well as my FPGA implementations ahead of schedule, allowing for extra bandwidth to assist wherever needed.

Outside of contributions that I made to my own section, I have also assisted with and helped to do some detailed design for other sections as needed and assisted with system integration to ensure that our system was fully functional before our project deadlines arrived. Additionally, in terms of non-technical contributions I would say that I have played a role in ensuring that we stayed coordinated for administrative tasks such as meetings, and that we were aware of deadlines as they arrived.

Team Effectiveness

Overall, I would say that we were fairly effective as a team with minimal disagreements or conflicts throughout the term. Over the past 6 weeks, we fostered a casual, but supportive environment where no one was afraid to ask questions or ask for help where needed. One thing that I believe solidified our collaborative environment, was a mutual understanding that although we were all assigned a given part, the team as a whole was building the system and one member's responsibilities did not end at their own subsystem. I strongly believe that this understanding was what allowed us to not just work as three individuals with a common goal, but as a fully functioning team with different members going above and beyond the call as the situation required.

While I would say that we were fairly effective as a team, there were some times where we did face challenges. One example of this was when we were first attempting to integrate our simulink model together. Although we had done our best to try to coordinate amongst each other on how blocks would be connected and what data they should be expecting, we did still encounter challenges when integrating, most likely caused by some lack of communication or unclear goals on who was responsible for what. However I believe that this was something that we improved upon throughout the term as we became more attuned to each other's workflows and ways of thinking. Another challenge that we faced as a team was documentation and version control. While there was some effort near the beginning to ensure that our files were organized in standard locations, this system unfortunately fell apart throughout the term. As these expectations were not clearly defined, variations in documentation due to the different way people think began to occur until they digressed too far to be recognizable to each other. This led to some issues throughout the term where incomplete or outdated files were being used by a team member leading to confusion and ultimately lost time. Given the opportunity to revisit this issue, I believe that I would have pushed stronger for a clearly defined documentation standard and possibly a more strict form of version control such as through Github.

Other Comments

None.

Fiona Luo

I was responsible for the transmitter and receiver subsystem blocks, as well as the Gilbert model in the channel. For the Simulink model, I researched on the implementation of state transition in Simulink, and implemented the channel subsystem. I also worked on testing the transmitter and receiver parameters with the spectrum analyzer, and designed the transmitter and receiver blocks. For the FPGA implementation, I have implemented the Gilbert model, and researched about random number generation methods on the FPGA. I have altered the parameters of the transmitter and receiver, and implemented the design in Verilog. Meanwhile, I contributed to checking the project requirements and validating that the channel, transmitter and receivers are integrated to the top level module according to the project constraints.

Except for the technical contribution, I contributed to the report documentation as well as the presentation explaining the subsystem I designed. I worked on the design decision of choosing the SRRC filter as our transmitter and receiver for the first Demo, and the system overview and transmitter/receiver verification for the second Demo.

Team Effectiveness

Our team has shown significant effectiveness, particularly due to our weekly scrum meetings, which have ensured clear and regular communication. These meetings have kept everyone informed about project progress and challenges, allowing us to promptly address issues and adjust our strategies. We set up an extra online meeting time for getting project progress updates after Demo 1, and it helped us in planning the project progress in advance so that we complete our tasks on time. Overall, our team collaborated effectively throughout the term.

Despite our strengths, there are still areas for improvement. Increasing the frequency of informal check-ins between formal meetings could help resolve issues even earlier. Additionally, implementing a shared project management tool would enhance our ability to monitor task completion and dependencies in real-time, providing greater transparency and proactive problem-solving. For instance, we used shared google drive to update changes in our Simulink model and Verilog code. However, in terms of code sharing, this method caused some problems since we were manually uploading files and we sometimes uploaded the wrong version. To improve code sharing and avoid version control issues, we should transition from manual file uploads on shared Google Drive to a version control system like Git.

Other Comments

None.

Peter Kim

I was responsible for the error correction and modulation subsystem block. Simulink and FPGA implementations were completed ahead of the project timeline, and the progress was communicated in a timely manner. I have also provided assistance in configuring the simulink models and integrating the sub-systems to meet the design requirements. Further contribution was made in designing and debugging the filter module and system integration on the FPGA.

In addition to the technical responsibilities, literature reviews were conducted to validate our design decisions, and tasks such as report formatting and file management were undertaken to improve the overall quality of our project deliverables.

Team Effectiveness

Initially, we encountered communication gaps and challenges with file version control, significantly impacting our efficiency and collaboration, which in turn led to setbacks and delays in meeting project milestones during the early stages.

Recognizing these hurdles had us take decisive action. We implemented more structured meeting schedules and enhanced our version control systems to minimize confusion and improve coordination among team members.

As we progressed, our capacity to handle conflicts also grew. Rather than seeing disagreements as obstacles, we embraced them as opportunities to identify a better design. Addressing conflicts transparently and respectfully streamlined decision-making processes, ensuring steady momentum towards achieving our project goals.

One of the initial improvements that stands out is the early adoption of a robust version control system for the project. This measure would have ensured timely access to accurate files and reduced the need for extensive communication regarding version discrepancies. Furthermore, maintaining more frequent progress updates from team members would have fostered continuous synchronization among team members, thereby enabling proactive support and mitigating last-minute rushes.

Other Comments

None.

Reference

[1] Convolution coding and applications: A performance analysis under AWGN channel. (2015, November 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/7507304

[2] Pandey, M., & Pandey, V. K. (2015, June 6). Comparative Performance Analysis of Block and Convolution Codes. https://www.ijcaonline.org/archives/volume119/number24/21388-4398/

[3] Brar, J. K., Bansal, R. K., & Bansal, S. (2016). Goodness analysis of generator polynomial for convolution code with varying constraint length. https://www.ijarcce.com/upload/2016/november-16/IJARCCE%2074.pdf

[4] *A truncation depth rule of thumb for convolutional codes*. (2008, January 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/4601052

[5] Decode convolutionally encoded data using Viterbi algorithm - Simulink. (2024, June 20). https://www.mathworks.com/help/comm/ref/viterbidecoder.html

[6] Meliani, H., & Guellal, A. (2006). COMPARISON BETWEEN VITERBI ALGORITHM SOFT AND HARD DECISION DECODING. *ResearchGate*. https://www.researchgate.net/publication/228758688_COMPARISON_BETWEEN_VITERBI_ALGORITHM_SOFT_AND_HARD_DECISION_DECODING

[7] Matched Filter Design for RRC Spectrally Shaped Nyquist-WDM Systems. (2013, December 1). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/document/6637045

[8] Hermanowicz, E., & Rojewski, M. (2014b). Square Root Raised Cosine Fractionally Delaying Nyquist Filter - Design and Performance Evaluation. *International Journal of Electronics and Telecommunications*, *60*(3), 247–252. https://doi.org/10.2478/eletel-2014-0031

[9] *Implementing Root Raised Cosine (RRC) filter for WCDMA using Xilinx*. (2011, April 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/5959095

[10] Singh, M., & Chandigarh, N. (2017). *FPGA based rrc filter using distributed arithmetic Algorithm*. https://www.semanticscholar.org/paper/FPGA-based-rrc-filter-using-distributed-arithmetic-Singh-Chandigarh/4957df6e6f82225624dd6f354f4c1ae82b1a4edf

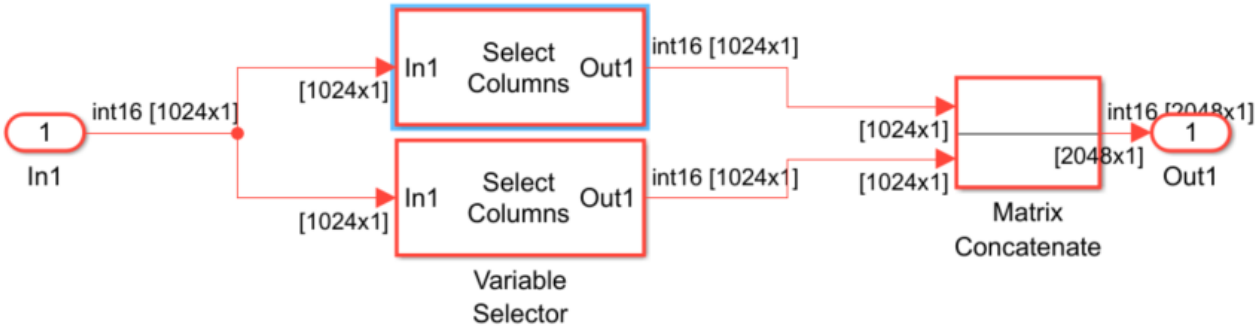
[11] *Decode convolutionally encoded data using Viterbi algorithm - Simulink*. (n.d.). https://www.mathworks.com/help/wireless-hdl/ref/viterbidecoder.html

[12] Langton, C. (1999). Introduction to Coding and decoding with Convolutional Codes. In *https://complextoreal.com*. SIGNAL PROCESSING & SIMULATION NEWSLETTER. Retrieved June 19, 2024, from https://complextoreal.com/wp-content/uploads/2018/09/convolutional-codes.pdf

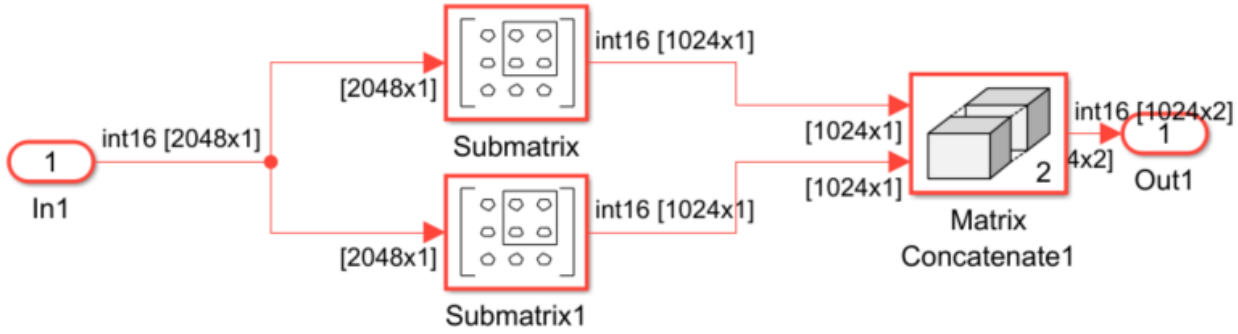
[13] Bawane, S., & Gohokar, V. V. (2014). SIMULATION OF CONVOLUTIONAL ENCODER. In *https://ijret.org*. International Journal of Research in Engineering and Technology. Retrieved June 19, 2024, from https://ijret.org/volumes/2014v03/i03/IJRET20140303104.pdf

[14] Tuominen, J., & Plosila, J. (n.d.). Asynchronous Viterbi Decoder in Action Systems. *ResearchGate*. https://www.researchgate.net/publication/31595950_Asynchronous_Viterbi_Decoder_in_Action_Systems

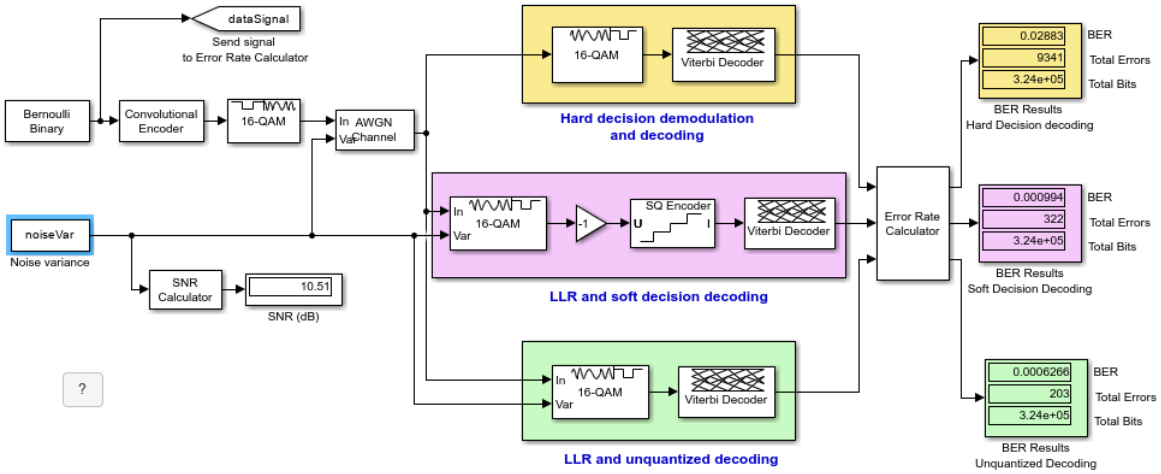
Appendix [A]



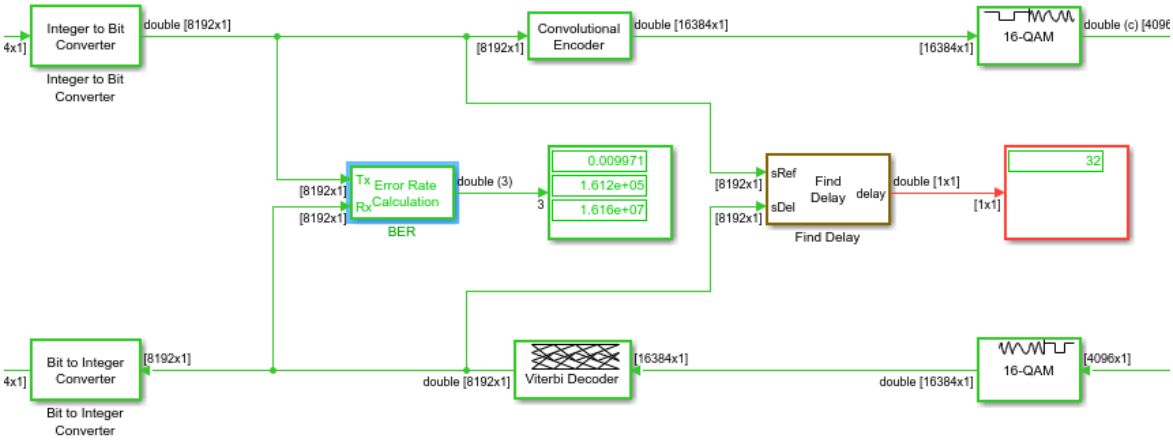
Appendix [B]

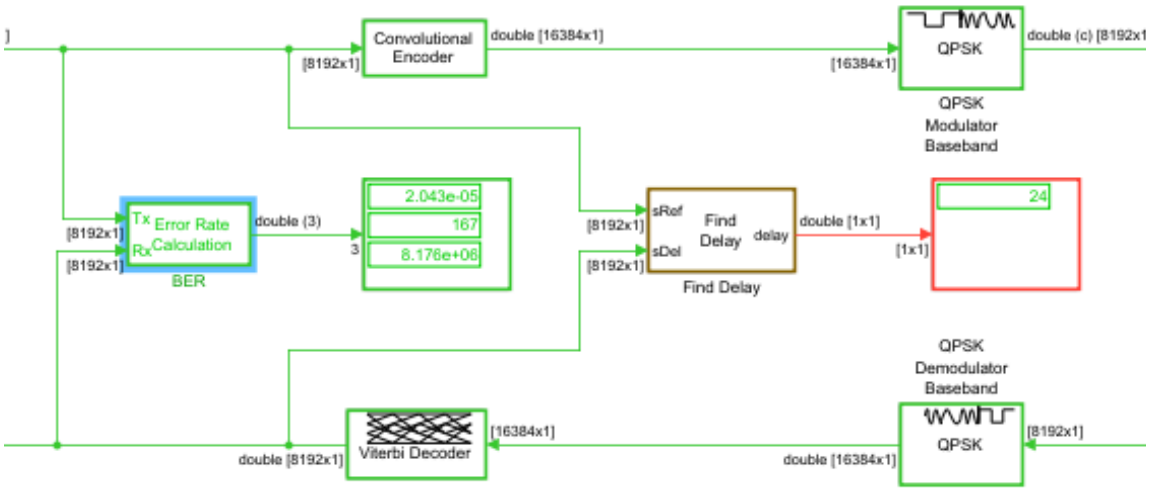


Appendix [C]



Appendix [D]


Integer to Bit Converter
Integer to Bit Converter
double [8192x1]
Convolutional Encoder
double [16384x1]
16-QAM
double (c) [4096
Tx Error Rate Calculation Rx
BER
double (3)
0.009971
1.612e+07
1.616e+07
sRef
sDel
Find Delay
delay
Find Delay
double [1x1]
32
Bit to Integer Converter
Bit to Integer Converter
Viterbi Decoder
16-QAM
[8192x1]
[16384x1]
[4096x1]
[1x1]


Convolutional Encoder
double [16384x1]
QPSK
QPSK Modulator Baseband
double (c) [8192x1
Tx Error Rate Calculation Rx
BER
double (3)
2.043e-05
167
8.176e+06
sRef
sDel
Find Delay
delay
Find Delay
double [1x1]
24
QPSK Demodulator Baseband
Viterbi Decoder
QPSK
[8192x1]
[16384x1]
[1x1]

Appendix [E]

```
% Define the extracted decimal values
decimal_values = [

];

% Generate a time vector (assuming equal time steps)
N = length(decimal_values);
time = (0:N-1) * 1e3; % Example: time in ps (adjust as needed)

% Plot Time-Domain Signal
figure;
plot(time, decimal_values);
title('Time-Domain Signal');
xlabel('Time (ps)');
ylabel('Amplitude');

% Frequency-Domain Analysis
Ts = (time(2) - time(1)) * 1e-12; % Convert time step to seconds
Fs = 1 / Ts; % Sampling frequency in Hz
fft_output = fft(decimal_values);

% Frequency vector
freq = linspace(-Fs/2, Fs/2, N);

% Power Spectrum
fft_output_shifted = fftshift(fft_output); % Shift zero-frequency component to center
power_spectrum = (abs(fft_output_shifted).^2) / N;

% Plot Frequency-Domain Signal
figure;
plot(freq, 10*log10(power_spectrum));
title('Frequency-Domain Signal');
xlabel('Frequency (Hz)');
ylabel('Power (dB)');
```

Appendix [F]

```
always @(*) begin
    case(cur_state)
        GOOD: begin
            if (p < p_GB) begin
                next_state = BAD; // Transition to bad channel
            end else begin
                next_state = GOOD; // Stay in good otherwise
            end
        end
        BAD: begin
            if (p < p_BG) begin
                next_state = GOOD; // Transition back to good channel
            end else begin
                next_state = BAD; // Stay in bad otherwise
            end
        end
    endcase
end
```

Appendix [G]

```
%Intialize answer variable to no for looping
answer = "No";
while answer~="Yes"
   %variable to determine the depth, width, and scaling of data
   iterations = 2^16;
   denomination = 2^4;
   SNR = 9;

   %generate gaussian values and scale
   matrix = randn([1 iterations]);

   variance = 10^-(SNR/10);

   matrix = matrix.*sqrt(variance);

   matrix_rounded = round(matrix.*(denomination));

   figure(1);
   h = histogram(matrix,100);
   xlabel("Value of Array Element (sorted to 100 bins)");
   ylabel("Number of Elements in Array");
   title("Distribution for Generated Gaussian Values");

   histogram_array = h.Values;

   %loop to check histogram bins and reject if curve does not stritcly
increase/decrease (randn
   %function did not distribute numbers evenly enough)
   prev_value = 0;
   ascending = 1;
   rejection_flag = 0;
   tolerance = 5;
   for n = 1:99
       if(histogram_array(1,n)+tolerance < prev_value && ascending == 1)
           ascending = 0;
       end
       if(histogram_array(1,n)-tolerance > prev_value && ascending == 0)
           rejection_flag = 1;
       end
       prev_value = histogram_array(1,n);
   end
   if(max(matrix)>(64/denomination) || min(matrix)<(-64/denomination))
       rejection_flag = 1;
   end

   %check whether user is accepts the curve, if not regenerate
   if(rejection_flag~=1)
       answer = questdlg('proceed with file write?','Yes','No');
```

```
        if(answer ~= "Yes" && answer ~= "No")
            return
        end
    end
end
    %{
    %scatter plot to show distribution from constellation point
    figure(2);
    clf(2);
    hold on;
    for m = 1:10000
        x = denomination+matrix(1,randi(iterations))*denomination;
        y = denomination+matrix(1,randi(iterations))*denomination;
        plot(x,y,'.');
    end
    xline(denomination,'--', 'LineWidth', 1, 'Color', "black");
    yline(denomination,'--', 'LineWidth', 1, 'Color', "black");
    plot(denomination, denomination, '.', 'Markersize',20, 'Color', "black");
    xlim tight;
    ylim tight;
    %}

%writing to file
filename = append("Gaussian_",num2str(SNR),"db_Values.txt");
myfile = fopen(filename,'w');
fprintf(myfile, "WIDTH = 8;");
fclose(myfile);
myfile = fopen(filename,'a+');
fprintf(myfile, "\nDEPTH=%d;\n\nADDRESS_RADIX = HEX;\nDATA_RADIX = HEX;\n\nCONTENT
BEGIN\n", iterations);
for m = 1:iterations-1
    fprintf(myfile, "%s : %s;\n", dec2hex(m,3), dec2hex(matrix_rounded(1,m),2));
end
fprintf(myfile, "END;");
fclose(myfile);
```

Appendix [H]

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.
From G*P_GB=B*P_BG and G+B=1, we can solve the equations to get G = 0.89, B = 0.11
Average transitions from Good to Bad=G*P_GB=0.89*0.03≈0.0268
Average transitions from Bad to Good=B*P_BG=0.11*0.25≈0.0267

Appendix [I]

Deliverable	**File Name**
Product Document	Demo 2_Project Report.pdf
Product Presentation	Demo 2 - Communication Model Design & Simulation -1-5.pdf
Simulink system file	Modelnew-1-5.slx
HDL source code, including testbenches and *readme* files	part1.v Buffers.v Converters.v tb_ADC.v tb_buffer.v tb_DAC.v tb_debuffer.v AWGN_Channel.v Gilbert_model.v LFSR.v S_2.v S_9.v S.21.v SRRC_filter.v tap.v tb_TR.v tb_transmitter.v EncDec_tb.v Traceback.v TracebackUnit.v Viterbi_Decoder.v ASC.v ACSEngine.v ACSRenorm.v ACSUnit.v BranchMetric.v CE.v M_PSK_Modulator_Baseband.v M_PSK_Modulator_Baseband_tb.v M_PSK_Demodulator_Baseband.v M_PSK_Demodulator_Baseband_tb.v BER_Counter.v n_bit_ff.v seven_seg_controller.v sevenseg.v Altera_UP_Audio_Bit_Counter.v Altera_UP_Audio_In_Deserializer.v Altera_UP_Audio_Out_Serializer.v

	Altera_UP_Clock_Edge.v Altera_UP_I2C.v Altera_UP_I2C_AV_Auto_Initialize.v Alterta_UP_I2C_DC_Auto_Initialize.v Altera_UP_I2C_LCM_Auto_Initalize.v Altera_UP_SYNC_FIFO.v Audio_and_video_config.v audio_codec.v