ELEC 391
Final Project Report
Team 7

Saksham Mahajan
Bobby Smith
Harnoor Saigal

Table of Contents

Objectives, Requirements & Constraints

In this project, our team's primary objective was to design and implement a reliable digital communication system for audio and data communication. The design, however, was bound to certain conditions, requirements, and constraints that we had to keep in mind while making the whole system. Our team was given scenario A for designing and implementing the overall project. The system was developed and tested using MATLAB and Simulink tools, and was prototyped on the Altera DE1-SoC FPGA board. In this report, we would lay out our design details, implementation, explanation, and some of the major design decisions that we took. This report's major emphasis would be on our Simulink design and FPGA implementation. Our design focused on achieving high fidelity and low bit error rate (BER) while operating within the specified bandwidth and processing delay constraints. All our subsystems and the overall design are functioning correctly and completely in both simulink and on the FPGA.

Our system was required to adhere to the following requirements and constraints:
1. Bandwidth Efficiency: The system should support a 4 kHz bandwidth for the audio signal.
2. Bit Error Rate (BER): Achieving a BER of (10^-5), ensuring minimal transmission errors.
3. Low Processing Delay: Maintaining an end-to-end delay from source to sink to be 25ms
4. Spectral Mask: Operating within a 100 kHz spectral mask to avoid interference.
5. Transmit Power: Maintaining the average transmit power to be maintained at 1 W.
6. Channel: Modeling the communication channel using the Gilbert- Fading model, simulating the transition between good and bad states based on the provided probabilities.
7. Modulation Scheme: Using Quadrature Phase Shift Keying (QPSK) or higher modulation schemes to efficiently map bits to symbols.
8. FPGA Implementation: Implementing the system on the Altera DE1-SoC FPGA board, utilizing its Wolfson WM8731 Audio CODEC for A/D and D/A conversion.

Some of the constraints that were involved included: hardware limitations of DE1-SoC board while implementing the WAV file source, the system's robustness against variations in channel conditions as modeled by the Gilbert-fading model, handling of additive white Gaussian noise (AWGN) in both good and bad channel states, and ensuring high fidelity from the source to the sink while minimizing degradation during transmission.

One of the most significant design challenges is balancing the trade-offs between achieving a low bit error rate, maintaining low processing delay, and operating within the specified spectral mask and constraints. Also, our team greatly struggled with implementing the Transmitter/Receiver in Verilog before it started working correctly. The initial Verilog implementation of the finite impulse response (FIR) filter that is used in the transmitter and receiver proved to be extremely difficult to showcase. Samples were moving through the buffers but there appeared to be errors occurring when calculating the taps (product of the coefficient with time delayed input samples). To simplify the calculations we opted to use a multiplication

module that could take in signed numbers and calculate the taps. This simplified the debugging process greatly and we were then able to see waveforms properly being formed from the filter. The receiver also created some discrepancies as the coefficients generated from the MATLAB receiver were the same as those for the transmitter. However, they were not working for the filter. After realizing that these had to be divided by the decimation factor in the receiver before doing the convolution, the transmitter/receiver pair started working as intended.

Audio BW(kHz)	Target BER	Spectral Mask (kHz)	Target Channel	Delay (ms)
4	10^{-5}	100	δ	25

Table 1: System Requirements for Team A scenario

Communication System Design

I. System Overview

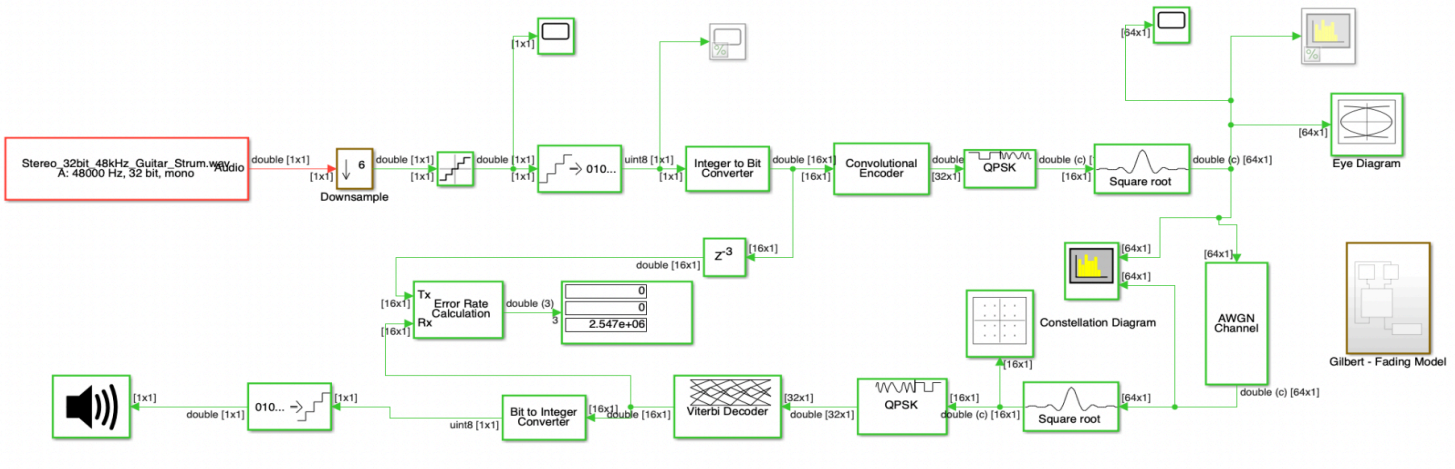


Figure 1: Simulink Working System

Simulink: The system involves an audio input file of 32 bits, 48 kHz. The ADC subsystem down samples and quantizes the signal which is input to the convolutional encoder that doubles the number of bits and is modulated using QPSK. The symbols are shaped and normalized through the raised cosine filter and go through the AWGN channel. Using the Gilbert fading model, the channel switches between good and bad using δ target probabilities through a state flow chart. The receiver raised cosine filter converts the waveform to symbols that get demodulated, and the Viterbi decoder performs error correction and produces binary outputs. The DAC subsystem finally produces integer values that are played on the speaker. Trade-offs and Performance criteria:

- The average transmit power is constrained to 1 W. QPSK modulation and raised cosine filtering help maintain efficient power usage while ensuring signal integrity.
- Occupied bandwidth is contained within the spectral mask using QPSK modulation and raised cosine filtering. The rolloff factor of 0.5 provides a balance between bandwidth efficiency and ISI suppression.
- The data rate is influenced by the sampling rate and modulation scheme. QPSK transmits 2 bits per symbol, and the system's throughput is calculated based on the effective bit rate after encoding and modulation.
- The target bit error rate (BER) is 10^{-5}. The convolutional encoder and Viterbi decoder combination along with QPSK that has low error probability is designed to achieve this error performance under the given channel conditions.

FPGA:

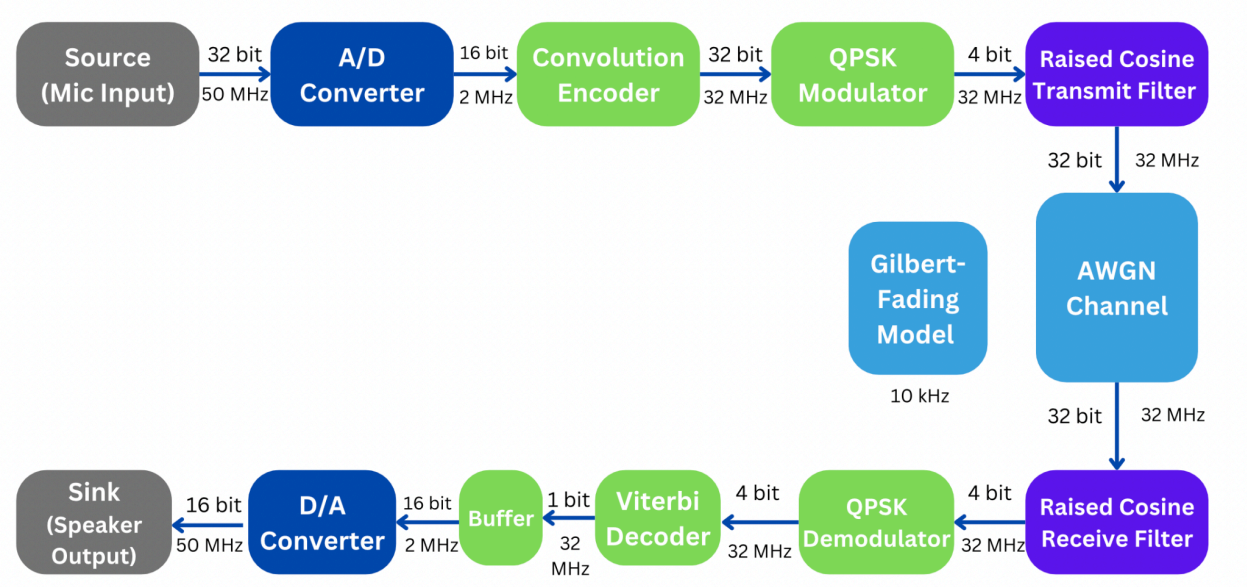


Figure 2: Digital System block diagram

The FPGA implementation involves using 2 clocks from PLL IP catalog that are 2 MHz and 32 MHz. The audio codec operates on a 50 MHz clock and we use the read and write control signals to interface the downsampler to process the 16 bit data at 2 MHz. Before the next input arrives from the downsampler, The encoder processes each bit of the 16 bit data 16 times faster at 32 MHz clock and sends 32 bits at once as output. These clocks are used to align inputs and outputs from these modules. The QPSK module also uses the 32 MHz clock and maps 2 bits to a symbol and outputs every clock cycle. The Transmitter shifts the 2 bit input to size it to 32 bits and processes the data at 32 MHz clock. The receiver converts the waveform back to symbols of 2 bits each of I and Q. The demodulator takes 4 bits and based on 4 possible combinations demodulates the symbols to 2 audio bits which are sized to 4 bits. The Viterbi module takes 4 bits and it's extraordinarily strong trellis structure and other features provided by the IP catalog accounts for errors induced by missing combinations from the modulation and demodulation blocks which were caused because we giving a default value to the symbols that were not perfectly aligned in the constellation. Lastly, the buffer uses a counter to shift the coming inputs from the Viterbi decoder in a 16 bit register and outputs after 16 clock cycles. This approach synchronizes all our subsystems with the use of PLL and the diagram below provides a visual representation of how all the signals are synchronized.

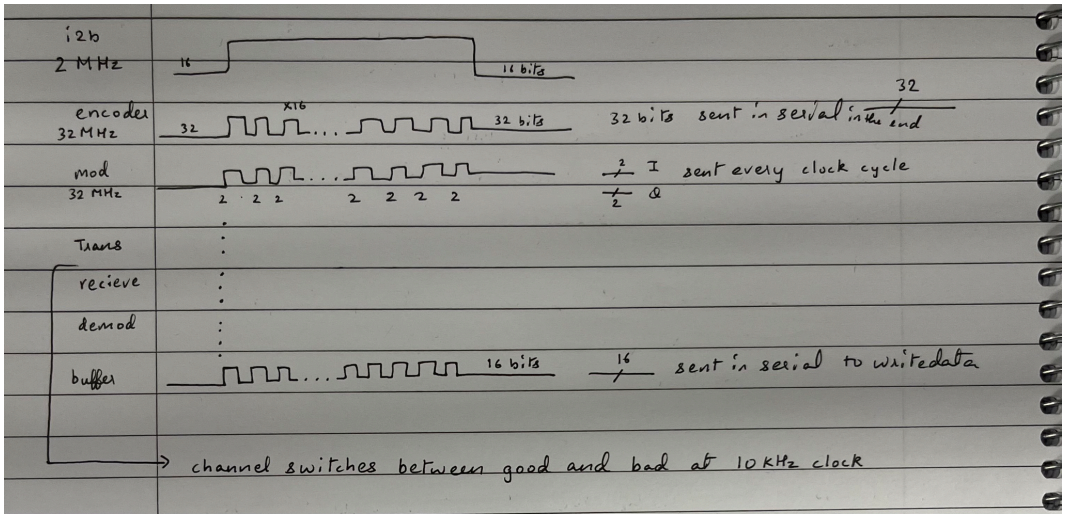


Figure 3: Clock synchronization representation

II. Subsystem Design

Source/Sink

Simulink Model: For implementing our system, we had to choose our source/sink. We had the option of choosing from a WAV file input or microphone/speaker set. For the Simulink implementation of the communication model, we have decided to choose the WAV file as the input and the laptop's default speakers as the output.

There are several reasons for this decision:
1. It would offer a consistent testing environment which can help us in meeting the BER specification and would also help in testing and debugging.
2. It would allow for more precise measurement and documentation
3. It would comparatively be easier to implement.

However, this decision comes with some trade-offs. Not using a microphone as an input source can make the system to be considered a bit unreliable since we would not be testing it on real-time audio, but the advantages of using a WAV file overweighs the advantages and complexities of using the microphone/speaker set, hence our decision became a valid trade-off for the simulink design.

The simulink blocks and the parameters used for source and sink are summarized below:

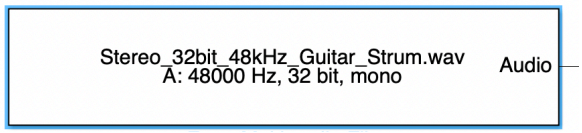
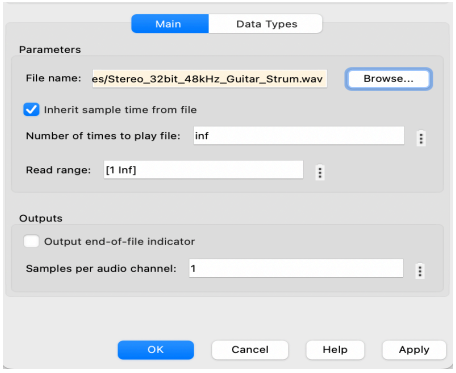
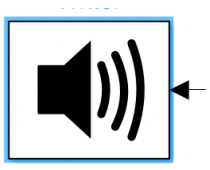
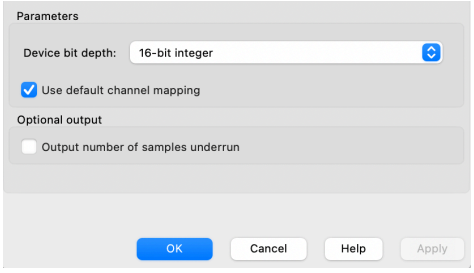
Blocks	Parameters
 From Multimedia File	
 Audio Device Writer	

Table 2: Simulink blocks and respective parameters used for source/sink

FPGA Implementation: The source/sink utilizes the Wolfson WM8731 Audio CODEC interface for interaction. The microphone converts sound waves into analog electrical signals, which are then downsampled and converted to digital signals by the Audio CODEC. These digital signals travel through the system and are transformed back to analog signals by the CODEC, which are then fed into the speaker to produce sound. The microphone and speaker are directly connected to the FPGA, ensuring seamless audio processing.

For the FPGA implementation, we aimed to overcome the design trade-off discussed earlier by incorporating both microphone and WAV file inputs into the FPGA design. We designed a mux with a switch to choose between the microphone and file inputs. We wrote behavioral Verilog code for the mux and switch, and instantiated the module alongside the Audio_ROM in the DE1-SoC for playing memory initialization files.

However, using Memory Initialisation Files to test, was taking a longer time as compared to the microphone input. Not only this, but having a microphone input allowed us to test the entire system in real-time audio conditions, to ensure more reliability. Providing immediate feedback, easy to use interface, straightforward debugging were some more advantages associated with using the microphone/speaker option. For testing purposes we used the Amazon Basics, microphone/speaker set provided to us, with the mic connected to the Pink coloured line, and speaker connected to the green line out.

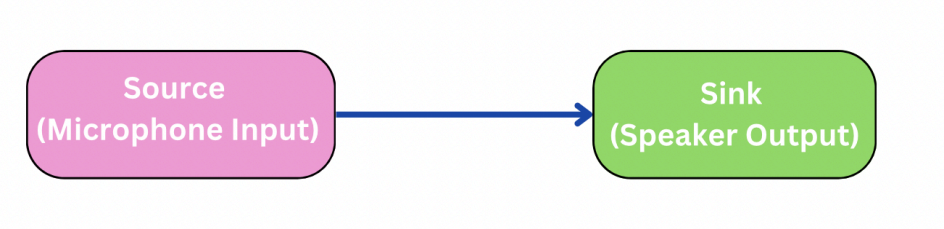
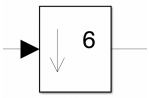
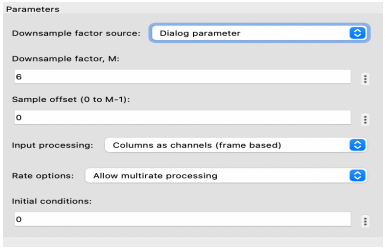
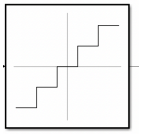
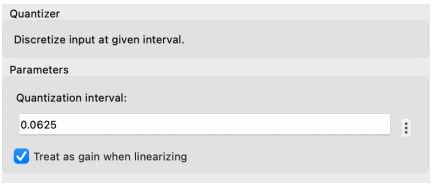


Figure 4: Brief Block Diagram for overall source/sink implementation

A/D and D/A Conversion

Simulink Model: For our communication design, we took the 32bit and 48kHz WAV file as our primary input source. After taking the input file, we had to downsample the file to fit within the given requirements. Our scenario asked us to have an audio bandwidth of 4kHz. According to Nyquist's theorem, we decided to take our sampling rate to be at least 8kHz, which is twice the bandwidth provided to us.. We downsampled the input from 48kHz to 8kHz by making the downsampling factor = 6 in the downsample simulink block, since (48000/6=8000). Next, we used a quantizer block that takes in continuous (analog) input signals and then maps these continuous values to a finite set of discrete levels. We set the quantization interval using the formula: Quantization interval = Signal Range/ $2^{\text{bit depth}}$, assuming a signal range of +/-1, we deduced the quantization interval to be equal to 1/ (2^4) = 0.0625, for 32 bit input. In the next step, we used an uniform encoder to encode these quantized values in binary form. In our design, the output of the encoder is in unsigned integer format. Finally, we used an integer to bit converter block, which uses 16 bits per integer (for better audio resolution) to make the whole subsystem coherent.

The simulink blocks and the parameters are:

Blocks	Parameters
 Downsample Block	
 Quantizer Block	

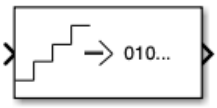
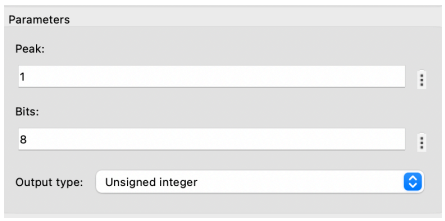
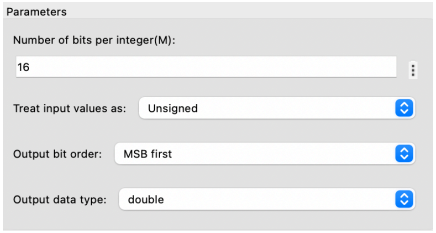
 Uniform Encoder	
 Integer to Bit Converter Block	

Table 3: Simulink blocks and respective parameters used for A/D conversion

To conclude this subsystem, we designed a D/A conversion system that includes a bit to integer converter, followed by a uniform decoder. The decoder converts the binary bitstream to the original quantized analog signal.

The simulink blocks and the parameters used for D/A converters are summarized below:

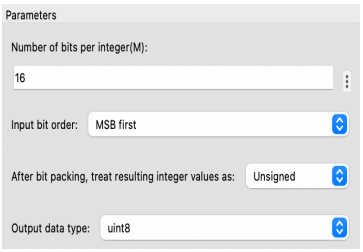
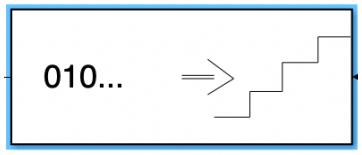
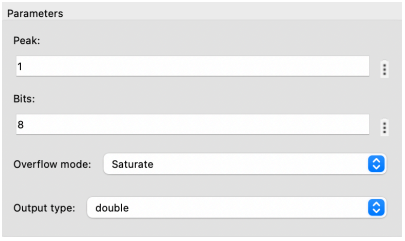
Blocks	Parameters
 Bit to Integer Converter Block	
 Uniform Decoder Block	

Table 4. Simulink blocks and respective parameters used for D/A conversion

The overall subsystem implementation including the source/sink and the A/D and D/A conversion:

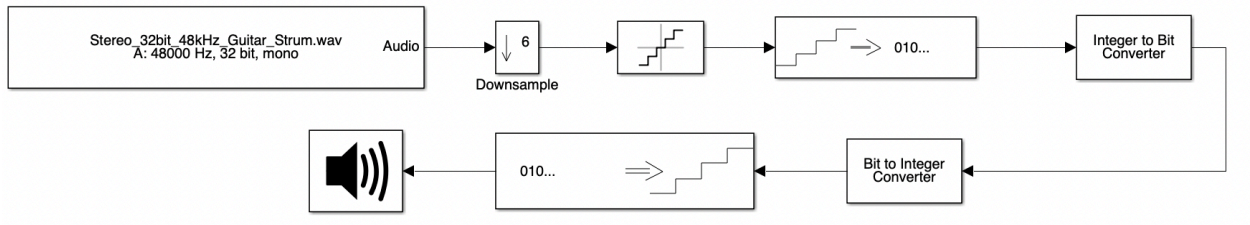


Figure 5: Simulink subsystem for source/sink and A/D and D/A conversion

FPGA Implementation: For the FPGA component of the A/D and D/A conversion, we heavily relied on the Audio CODEC to convert the analog signals from the microphone to digital signals, and then convert the digital signals back to analog signals. However, two critical parameters had to be considered while using the CODEC for ADC and DAC: sampling frequency and quantization.

As explained earlier, our downsampling factor was 6. The audio CODEC does not downsample the input, so we had to write a separate module for downsampling, called downsampler. Downsampling involves taking every nth sample of the input signal. In our case, we took every 6th sample to reduce the sampling frequency from 48kHz to 8kHz by setting the counter value to 5. Please refer to figure 24 in the verification section for the code of downsampler.

To achieve the desired quantization interval, as explained in the simulink section above, we needed to modify the parameters in the CODEC code. Our system uses 16-bit values, while the CODEC operates on 24-bit values. Therefore, we changed the parameters from 24 to 16 bits and adjusted the corresponding register values from 5'd23 to 4'd15.

Parameter	Old Value	New Value
AUDIO_DATA_WIDTH	24	16
BIT_COUNTER_INIT	5’d23	4’d15

Table 5. Simulink blocks and respective parameters used for D/A conversion

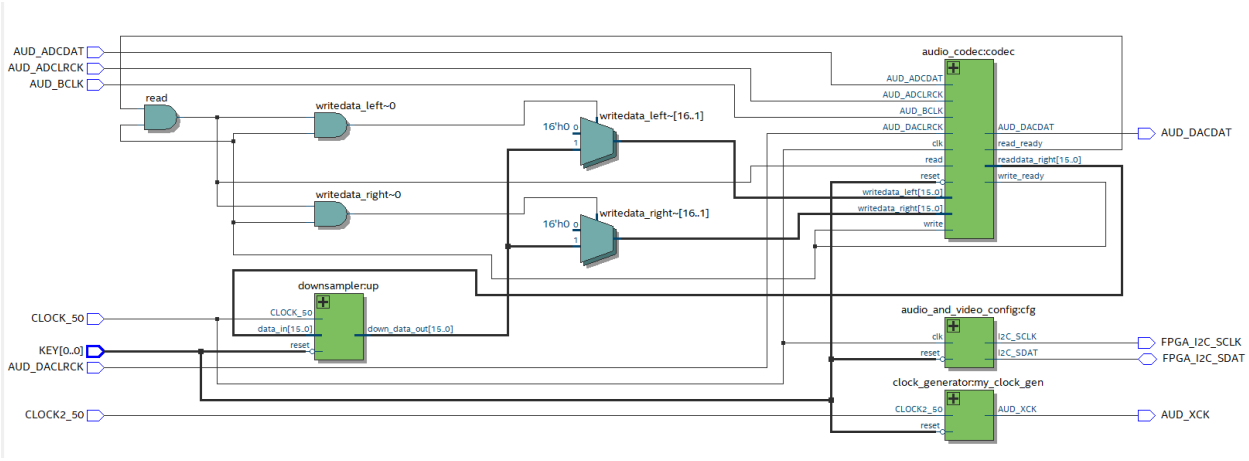


Figure 6: Block Diagram for overall ADC and DAC subsystem, including the downsampler

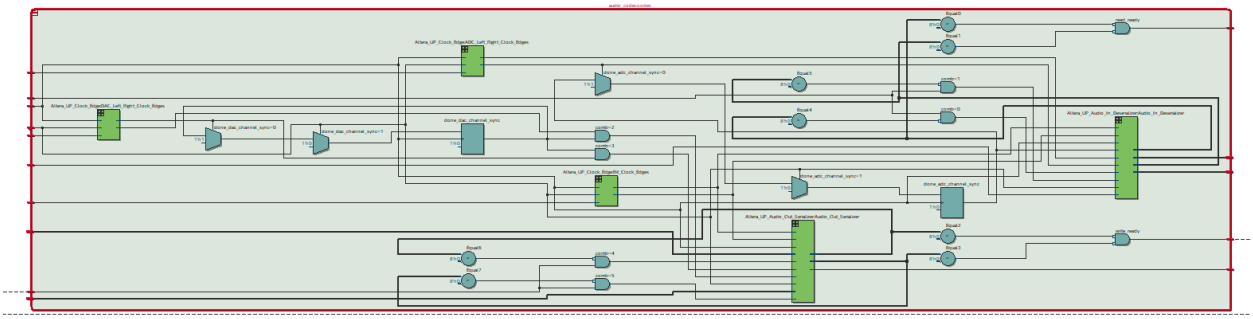


Figure 7: Block Diagram for the Audio CODEC

Error Correction Encoder / Decoder

Simulink model: The parameters for the subsystem blocks are shown in the table below. The convolutional encoder is configured with a constraint length of 7 and standard polynomials [171, 133] in octal notation. This setup is chosen for its balance between error correction capability and computational complexity. Continuous mode is used to maintain the state of the encoder across input frames, ensuring consistent error correction. The Viterbi decoder is matched to the convolutional encoder's trellis structure. Hard decision decoding is chosen to simplify the decoding process by making binary decisions on received bits. A traceback depth of 32 is selected to balance decoding accuracy and complexity. Continuous mode ensures that the decoder processes the input stream consistently for achieving a BER of 10^{-5}.

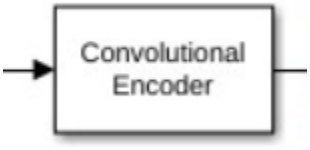
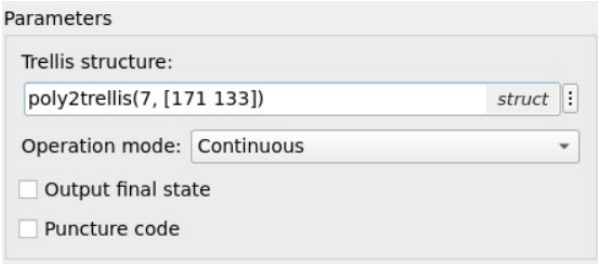
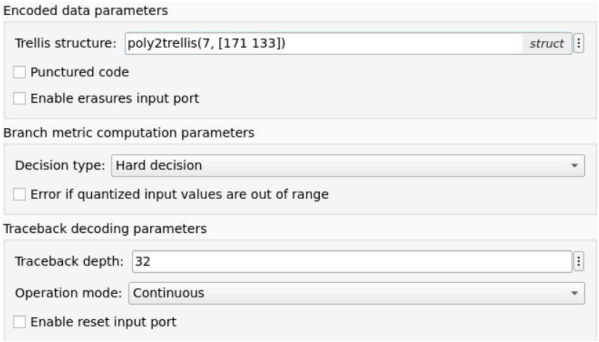
 Convolution Encoder Block	
 Viterbi Decoder Block	

Table 6: Simulink blocks for error correction

FPGA implementation: The convolutional encoder uses the same parameter configuration from the simulink block. The block diagram below shows the inputs and outputs.

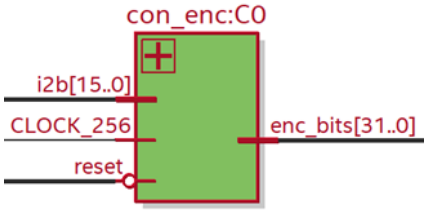


Figure 8: Block diagram for Convolutional encoder

It takes a 16-bit input from the ADC subsystem and processes each bit and outputs 2 bits. Therefore, it operates on a clock that is 16 times faster than that the clock that the down-sampler operates on because when new input will arrive from i2b then it would have processed each bit for 16 times and send 32-bit encoded output in serial. A more detailed RTL block diagram is shown below.

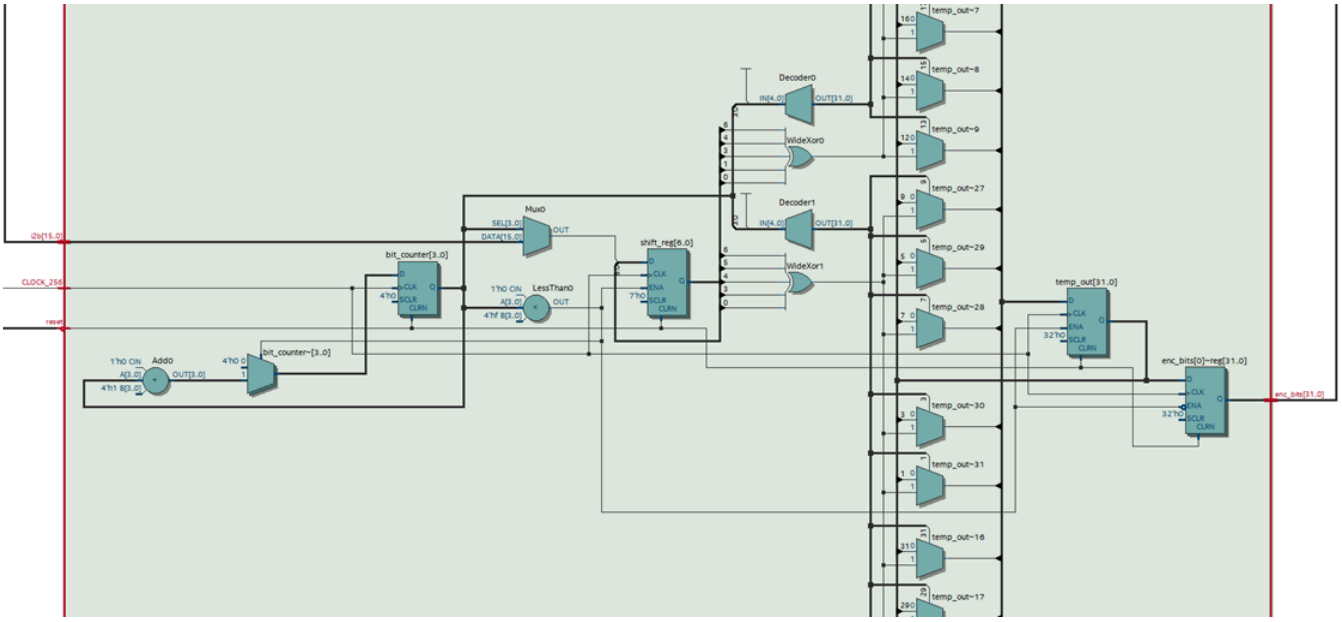


Figure 9. RTL block diagram of convolutional encoder

The encoding involves using a shift register to hold the current input bit. The output bits 1 and 2 are calculated by doing bit wise AND with the generator polynomials and then XOR reduction to 1 bit. The output bits are stored in a temp_out register and a counter is used to keep adding 2 bits to the temp_out register every clock cycle. After 16 clock cycles, the 32 bit enc_bits is given the value of temp_out and is the output from the module.

The Viterbi decoder was used from the IP catalog and it also operates at the 32 MHz clock. The block diagram below shows the inputs and outputs.

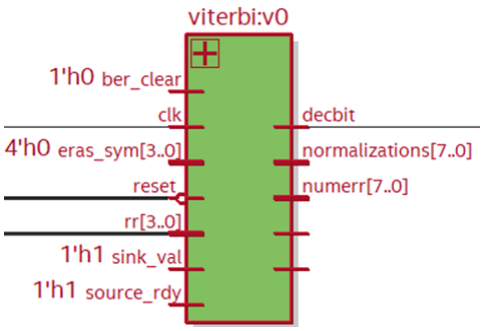


Figure 10: Block diagram of Viterbi decoder

It takes 4 bit input from the rr signal and outputs a decoded 1 bit decbit signal. It also outputs the number errors counted. The IP catalog below shows the parameters selected for the Viterbi decoder. These are matched to the parameters from the simulink model.

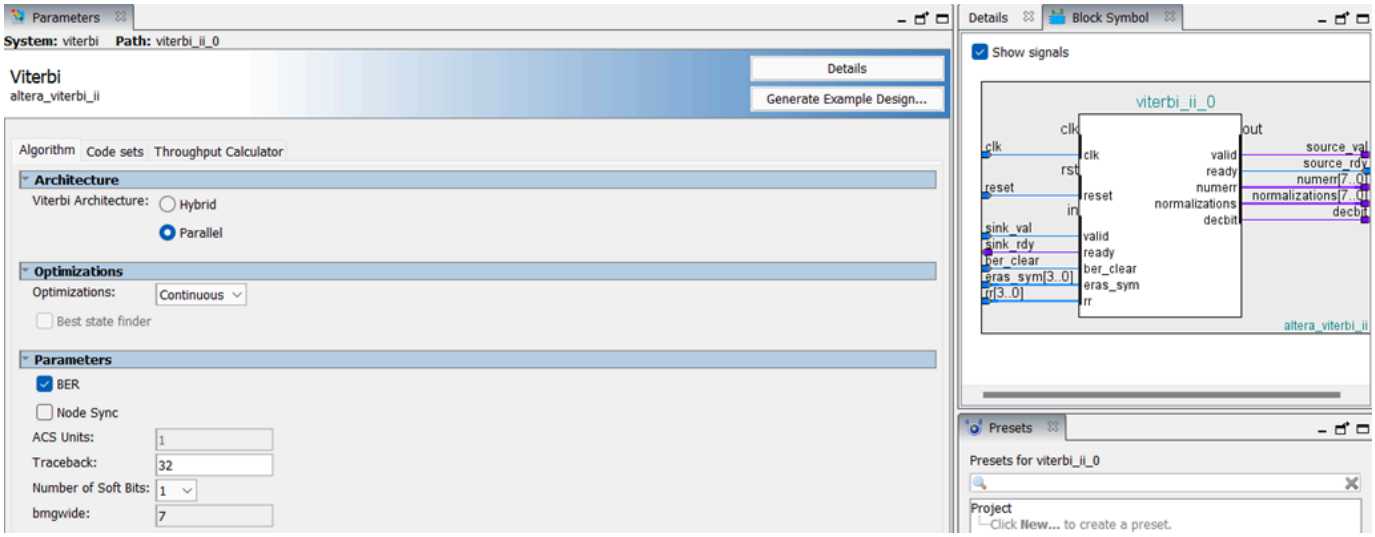


Figure 11: IP catalog for Viterbi decoder

Additionally, soft bits of 1 mean hard decision decoding and the architecture is parallel for faster decoding to reduce computational delays. The RTL diagram below shows a detailed implementation.

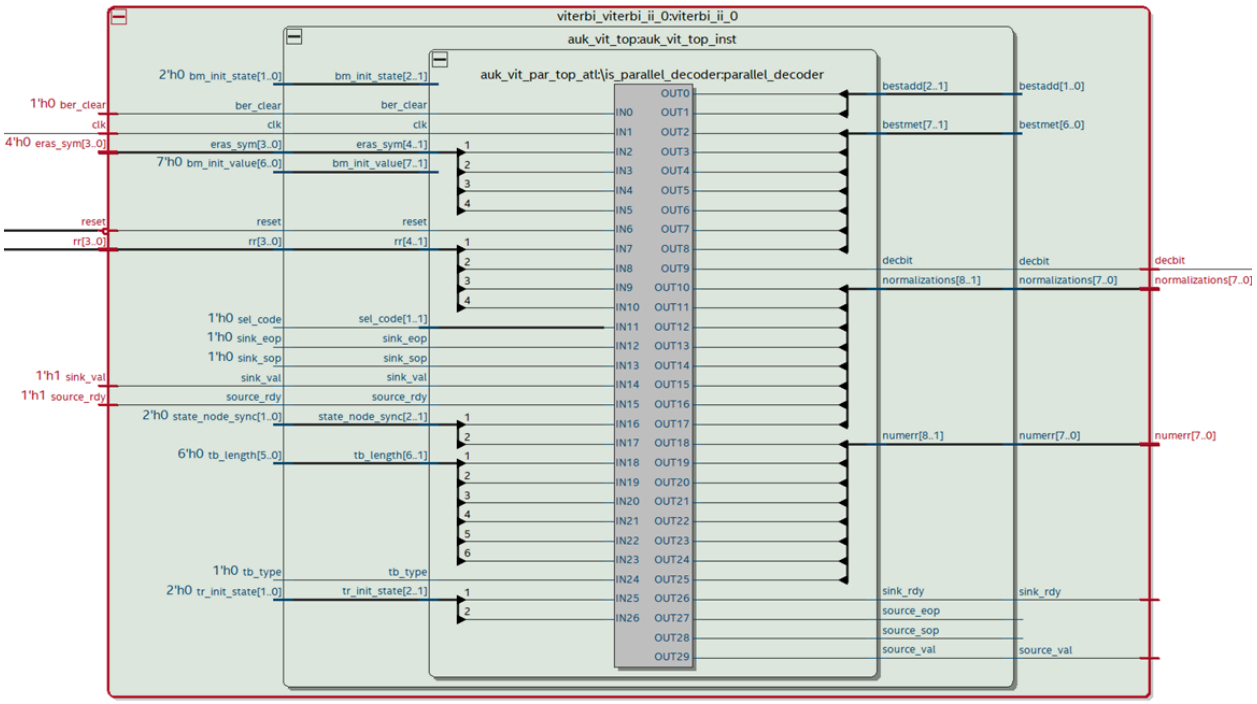


Figure 12: RTL block diagram for Viterbi decoder

The IP core guide gave all the information on the signals involved and which files have the template to instantiate the Viterbi module and the black box module. The input signal rr is the input which is connected to the demod_bits port. The signal numrr counts the number of errors occurring and is outputted by the module. From reading the IP core guide with limited description of the use of various control signals, I assume sink_val, source are always valid. Also that there is no BER clear signal and no erasure symbols. Also, since the Viterbi module outputs 1 bit every clock cycle, I use a buffer as shown in the block diagram below to output 16 bits to the DAC subsystem.

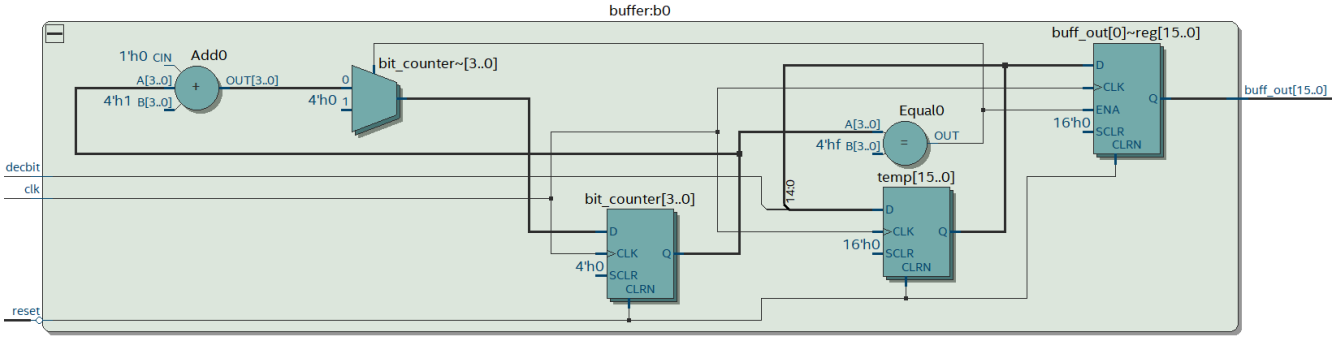


Figure 13: RTL block diagram for buffer module

It uses a counter to update a 16 bit register every clock cycle till the counter reaches 15 and then outputs the buffer to the DAC subsystem.

Modulator/ Demodulator

Simulink model: The parameters for the subsystem blocks are shown in the table below. QPSK (Quadrature Phase Shift Keying) is used for its efficiency in bandwidth and power. It has the lowest error probability compared to other higher modulation techniques so it would reliably guarantee BER of 10^{-5}. Gray coding is chosen for constellation ordering to minimize bit errors, as adjacent symbols differ by only one bit. A phase offset of pi/4 is selected to prevent phase ambiguity and facilitate differential decoding. The QPSK demodulator parameters are configured to match the modulator settings. Hard decisions are used to simplify the decision-making process during demodulation. Matching the constellation ordering and phase offset ensures proper demodulation of the received signal.

QPSK Block

QPSK Block

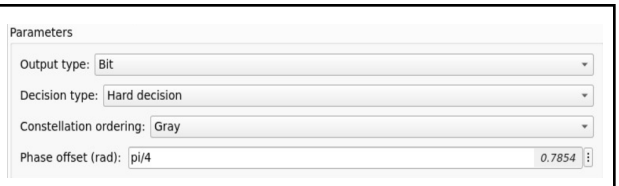


Table 7: Simulink blocks for modulation and demodulation

FPGA implementation: The QPSK modulator operates on the 32 MHz clock and the block diagram below shows the inputs and outputs.

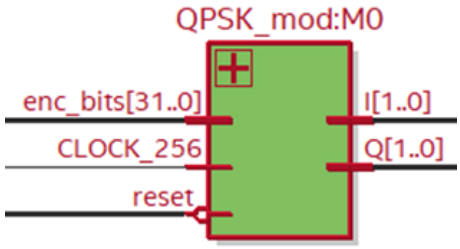


Figure 14: Block diagram for QPSK modulator

The QPSK modulator uses the same parameters from the Simulink block except the phase offset. For the FPGA implementation the phase offset was chosen as pi/2 to prevent phase ambiguity but more importantly not needing to use floating point values and use integers. The table below shows the symbol mapping for the distinct phase shifts.

Symbol Mapping	**Phase shift (rad)**
00	0
01	Pi/2
11	Pi
10	3pi/2

Table 8: Table for symbol mapping

The modulator takes an input of 32 bits from the encoder module and processes 2 bits every clock cycle at 32 MHz, the same clock frequency as the convolutional encoder. Based on the 2 bits selected I and Q symbols are outputted. I represents the real bits and is the In-phase component. Q represents the imaginary bits and is the Quadrature-phase component. The table below shows the combinations of values that I and Q can be based on the symbol received.

Symbol	I	Q	Reason
00	01	00	cos(0) = 1 and sin(0) = 0
01	00	01	cos(pi/2) = 0 and sin(pi/2) = 1
11	11	00	cos(pi) = -1 and sin(pi) = 0
10	00	11	cos(3pi/2) = 0 and sin(3pi/2) = -1

Table 9: Combinations selected for I and Q

There should be 16 combinations for {I,Q} because the point can be anywhere in the constellation. However, we can just output {00,00} by default because the Viterbi's trellis structure is 7 which makes the decoding very robust. It happens to successfully correct these induced errors. So we can keep our modulation simple. The RTL diagram below shows detailed working of the modulator.

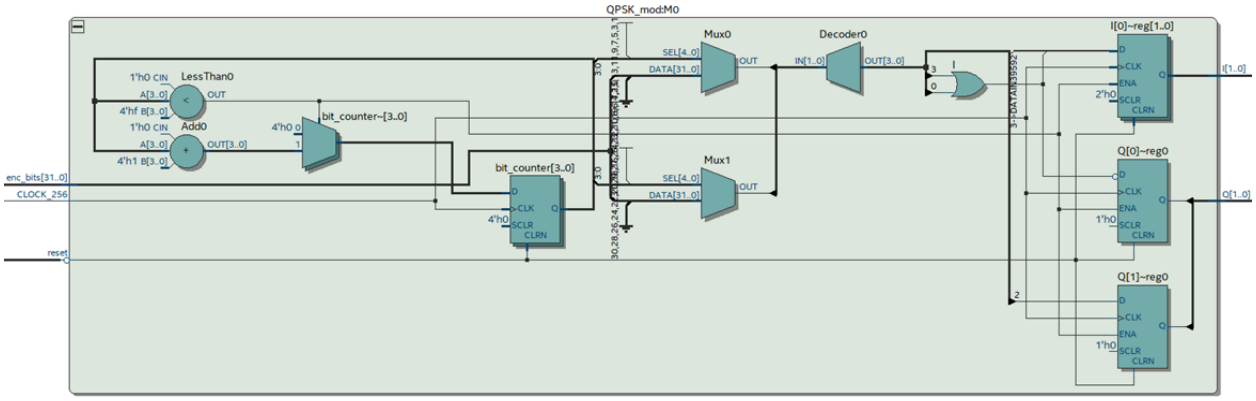


Figure 15: RTL block diagram for QPSK modulator

The block diagram below shows the input and output signals of the demodulator module.

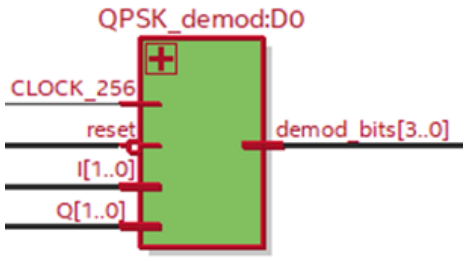


Figure 16: Block diagram for QPSK demodulator

At 32 MHz clock, for every clock cycle the receiver module sends I and Q as inputs with noise added from the channel when they are transmitted. The module processes 4 bits from I and Q in concatenation and maps the symbols to audio bits. The output demod_bits is sized to 4 because the Viterbi decoder takes 4 bits as input at once. The table below shows the symbol mapping.

I	Q	Demodulated bits
01	00	0000
00	01	0001
11	00	0011
00	11	0010

Table 10: Demodulation combinations

The RTL diagram below shows the detailed implementation.

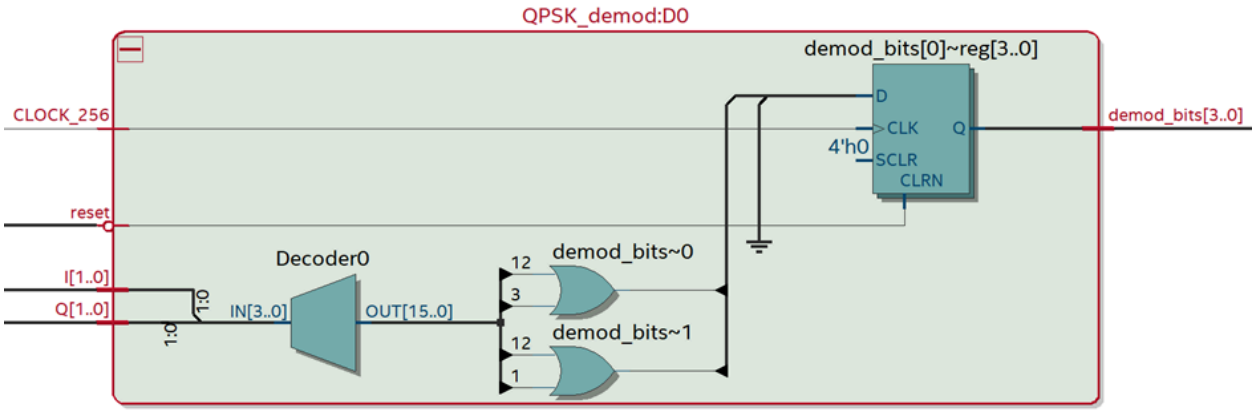


Figure 17: RTL block diagram for QPSK demodulator

The overall subsystem implementation in Simulink is shown below.

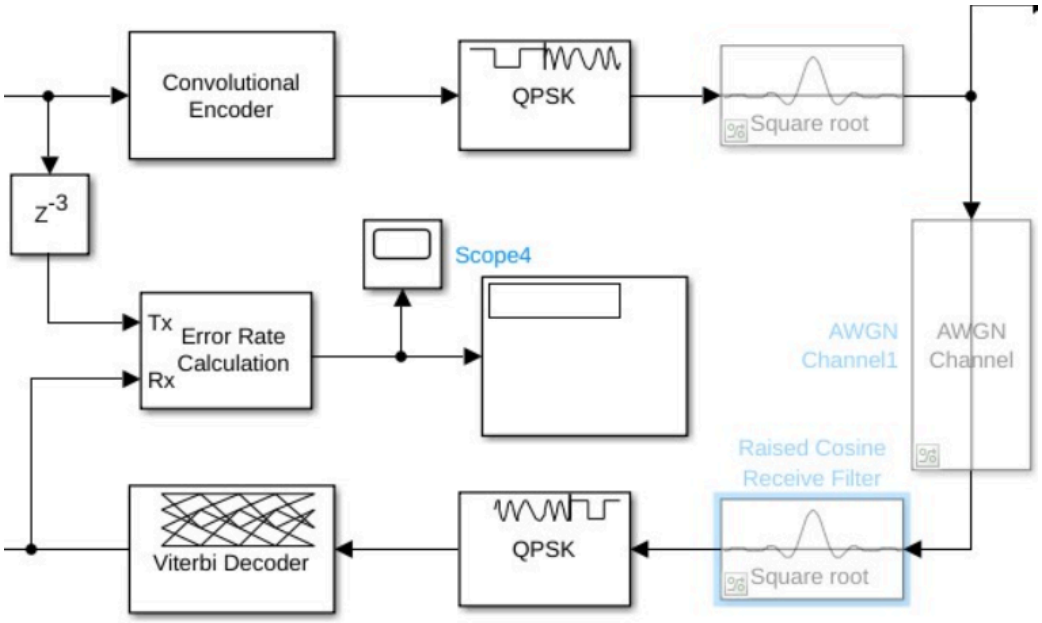


Figure 18: Simulink blocks for error correction and modulation/demodulation

Transmitter / Receiver

Simulink Model: For the Simulink model of the Simulink blocks used include the raised cosine transmit filter/receiver. For our group especially, efficient use of the frequency spectrum was one of the main concerns, this necessitated a transmission pulse that was more efficient in use of the frequency spectrum than the square wave. The raised cosine filter works better as it has a narrow bandwidth and obeys the Nyquist criterion for zero inter-symbol interference with zeroes at multiples of the signal period. Using this wave helps ensure a good BER and that the transmission pulse comes in under the strict 100kHz spectral mask restriction with the wave diminishing at around 74kHz. The square root applied to the signal also takes care of the energy normalization. The transmitter had a low roll-off factor of 0.3 for the purpose of staying below the spectral mask. Output samples per symbol was kept at 4 which was satisfactory for reducing aliasing in our case. The filter span was set to 16 as a longer filter span yields a narrower bandwidth for the transmitted signal since the spectral content is concentrated over a longer duration. Parameter selection for the receiver mimicked that of the transmitter as a matched filter design is ideal for optimizing the SNR of the received waveform with AWGN. The decimation factor is set to 4 to reverse the upsampling done by the transmitter and maintain synchronization within the simulation.

The simulink blocks and parameters used for the Transmitter/Receiver are described below.

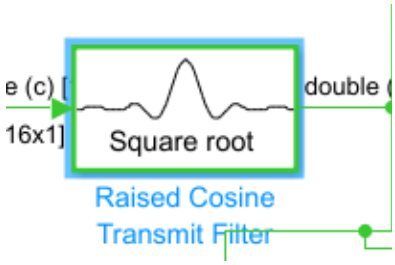
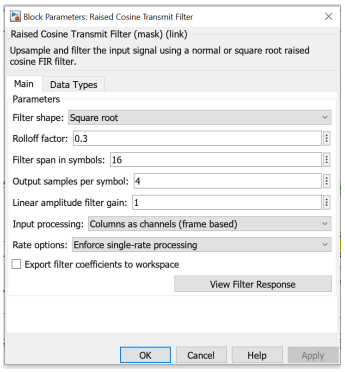
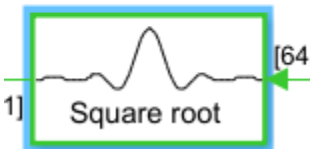
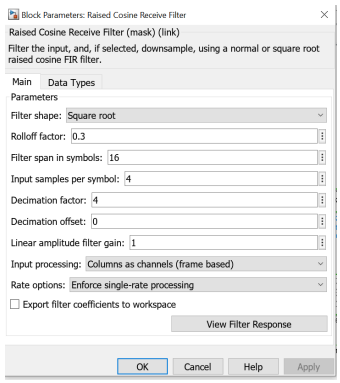
Blocks	Parameters
	
	

Table 11: Transmitter/Receiver Simulink Parameters

FPGA Implementation:

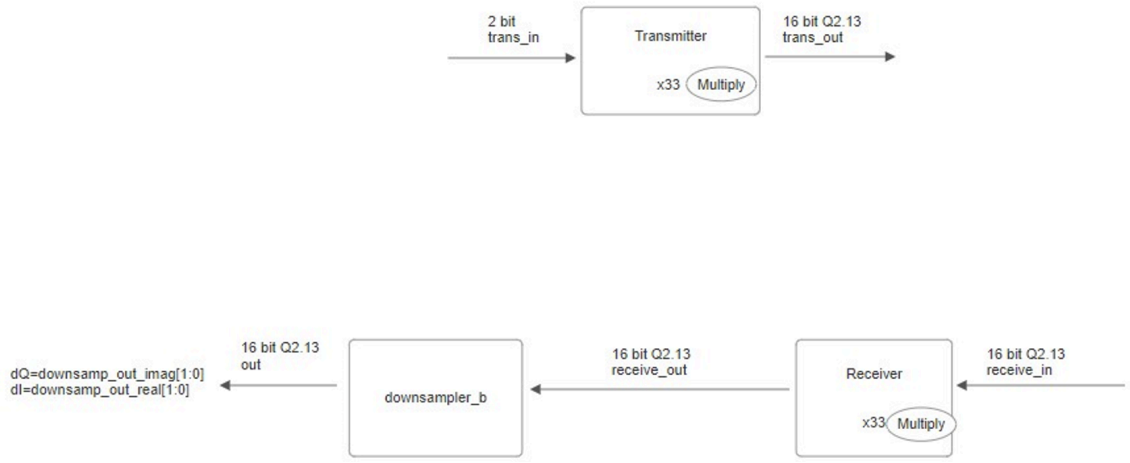


Figure 19: Block diagram of the Transmitter/Receiver FPGA Implementation

For the FPGA implementation, the transmitter and receiver were both implemented as FIR filters, using shift registers to hold and move down the input samples and the filter coefficients being held in stationary registers. Each clock cycle the inputs are shifted down and the taps are calculated by multiplying all the shifted input samples by the coefficients. This is simplified through the use of a multiply module that is instantiated in both the receiver and transmitter. The multiplied results are then accumulated to form the output. The main difference between the two is that the receiver coefficients are the same as the transmitter but divided by the decimation factor of 4 to properly perform the convolution. The coefficients were also exported with the square root option selected in the block to ensure that they reflect that aspect of the waveform and ensure energy normalization.

For the interface between the system and other systems, the transmitter inputs 2 bits for the imaginary and real signals, it then concatenates 14 zeroes onto the front of this then left shifts it by 13 to convert to Q2.13 format, this ensures meeting the channel constraints and makes it impossible for the amplitude to exceed plus or minus 4. Similarly, after downsampling, the outputs from the receiver are converted to two bits again by extracting bits [1:0] from the output signal.

The downsampler_b module is very simple and just outputs the input when a counter that reaches the decimation factor of 4 has been reached, this reverses the effect of the interpolation from the transmitter module.

For the FPGA implementation, the filter span in symbols was reduced from 16 to 8, this was done to reduce the filter coefficients to 33, which are the product of filter span and output samples per symbol. The main reason for this was to reduce design complexity, as the coefficients have to be converted to Q2.13 format and then copied in as a lookup table. Doing this also reduces the delay in the system and helps to hit our target of 25ms. Additionally, given how robust the Viterbi decoder had shown to be in our simulation with a bit error rate of zero, it made sense to make the tradeoff of possibly increasing the bit error rate by shortening the filter

span. Some of the additional accuracy from this very long filter span was being wasted and we were not in danger of passing our target BER of 10^{-5}. This simplification facilitated an easier time implementing shift registers for the input samples and the registers for the coefficients to perform the convolution.

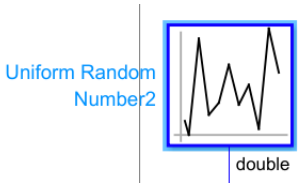
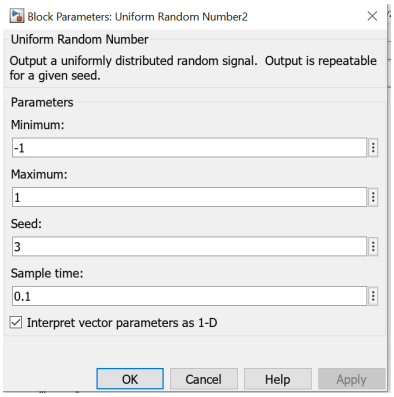
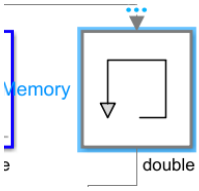
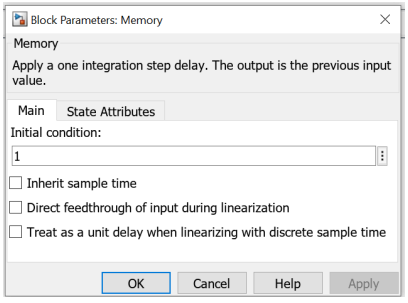
Component	Parameters
transmitter7	trans_in, trans_out, clk, reset, read_ready
receiver	receive_in, receive_out, clk, reset
multiply	in1, in2, out
downsampler_b	In, out, clk, reset

Table 12: Subsystem components for the Transmitter/Receiver and their parameters

Channel

Simulink Model: Simulink blocks used include the AWGN channel, and in the Gilbert-Fading subsystem a Matlab function block, uniform number generator block, memory block, and display block. The AWGN channel is operated using the Gilbert-Fading model. The model uses a uniform random number between 1 and -1 which is updated 10 times every second and fed into the matlab function block which handles the state transition logic using case statements and updates the snr variable in the workspace that is used by the AWGN block. The state is output from this block and held in a memory block which allows for the function to remember its state. The function uses the case statement and uniform random numbers to figure out when to transition between the good and bad states given the probabilities in the design specifications. For the delta channel, the transition probabilities are defined as: $p_{gb} = 0.07$ and $p_{bg} = 0.15$.

The simulink blocks and parameters used for the channel are described below.

Blocks	Parameters
	
	

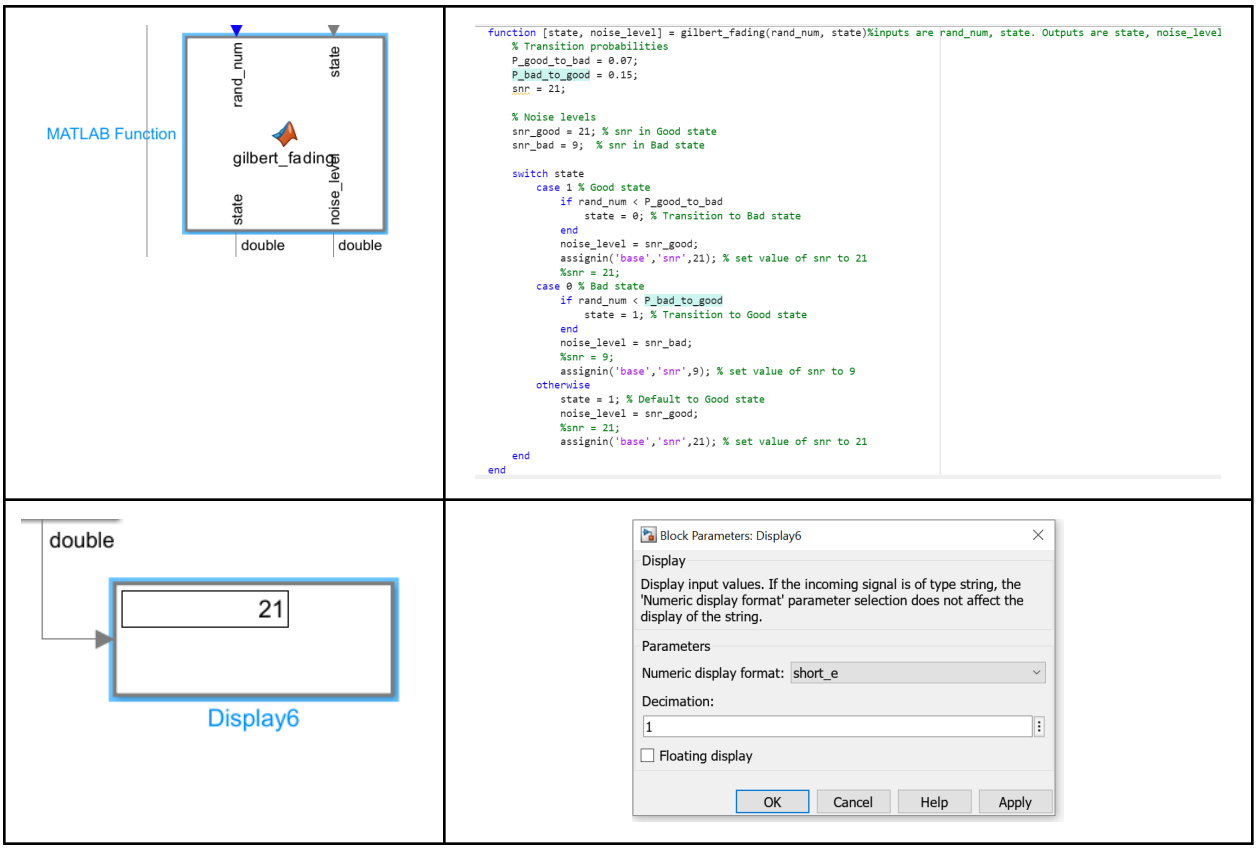


Table 13: Channel simulink blocks used and their parameters
*Please find the MATLAB Function Code in the Appendix

FPGA Implementation:

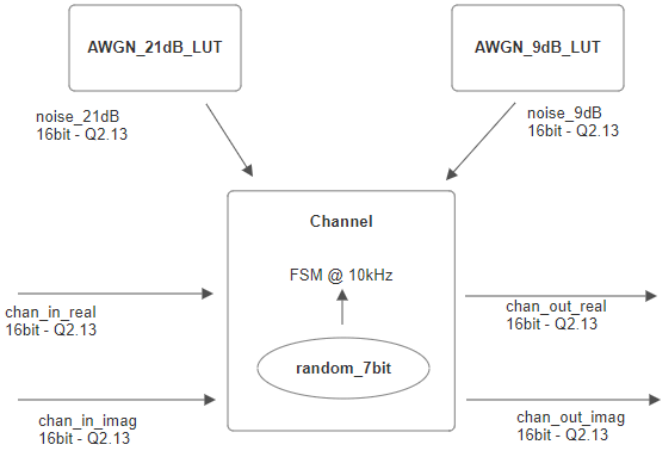


Figure 20: Block diagram of FPGA implementation of the AWGN Gilbert Channel.

The FPGA implementation of the AWGN generator was significantly simpler than that of the transmitter and receiver. Two lookup tables of 50 values were generated by running a constant block with value of 0 through a simulink AWGN block and exporting it to the workplace. These could then be converted to Q2.13 format and stored in lookup table modules, which could be indexed through each clock cycle to select the next value to add to the signal.

The Gilbert fading model is a basic state machine implemented directly in the channel module, this is controlled by a linear feedback shift register which generates random 7 bit numbers which are then fed into conditional statements to control the switching of the states. The good state results in adding noise from the 21dB lookup table to the input signals, and the bad state does the same for the 9dB lookup table.

The justification for this design in terms of our scenario follows, the decision to go with Q2.13 as my representation allowed me to meet the 16 bit requirement for channel quantization and also made it impossible to go outside of the +-4 range for channel amplitude. Some drawbacks of this were increased complexity in making sure my numbers (such as symbols, coefficients, noise lookup table values) were all being converted perfectly.

The choice to go with a lookup table instead of HDL generated was based on simplicity, numbers were easy to generate by hooking up a constant 0 signal in simulink to an AWGN generator then exporting results from 9 and 21dB, then converting to Q2.13 and storing as signed decimals in Verilog. A drawback of this is being limited to a finite number of samples, 50, reducing randomness.

The reasoning behind the 7 bit linear shift feedback register to generate random numbers is it made it easy to translate the probabilities to Verilog (ie. max random integer is 127, P(rand<19) = 0.15). It was relatively easy to validate as numbers could be exported and graphed in excel with minimal effort. The tradeoff here is linear shift feedback registers are only pseudo random so you do get some clustering around ranges of values before it moves into another range.

Component	Parameters
AWGN_21dB_LUT	input wire clk, output noise_21dB
AWGN_9dB_LUT	input wire clk, output noise_9dB
Channel	clk, CLOCK_50, real_channel_in, imag_channel_in, real_channel_out, imag_channel_out, reset
random_7bit	input clk, input rst, output random_num

Table 14: Channel subsystem components and their parameters.

Verification

I. System Performance

Criterion	Simulink Perf.	FPGA perf.
Message Transmission (bit rate)	~4kHz	2.60kHz
Transmission Reliability (bit error probability)	0	~0
Processing Delay	129 ms	14.94ms
Channel Bandwidth	74.25kHz	88.54kHz

Table 15: Performance Scenario Table

Message Transmission (bit rate): For the FPGA performance, we calculated the bit rate by following certain steps. We know as our channel was working at a 1MHz frequency, and since our samples per symbol from the transmitter were 8, we divided 1MHz by 8 to get 125kHz. Then, we further divided this value by 8, because our modulator serialized the 32 bits into 4-bit output streams, giving again a factor of 8. We hence got the value of 15.625kHz. As we are downsampling the overall input by a factor of 6, we divided this value again by 6 to get the bit rate equal to 2.6kHz we get the value of 2.60kHz

Transmission Reliability (bit error probability): For the FPGA performance we based it on the output from the Speakers which was almost the same when we switched between noise and no-noise using a KEY. It wasn't possible to simulate the entire system on modelim to compare the output of the decoder with the input of the encoder because the Viterbi module was a black box which can't be simulated in modelsim.

Processing Delay: For the FPGA performance, we added the setup time to the combinational logic delay to get this.

Channel Bandwidth: For the FPGA performance we took the number of clock cycles it took to transmit a complete wave, then divided the channel frequency by that.

II. Source/Sink

Simulink: For verifying the source/sink subsystem, we used the spectrum analyser with two input ports. The first input port takes the downsampled input of the uniform encoder and the other input is the output of the uniform decoder. The SNR is set to 100dB for the sake of testing, and with such high SNR, the noise should be negligible and the two signals should fairly overlap each other. After running the system, we see the following in the spectrum analyser:

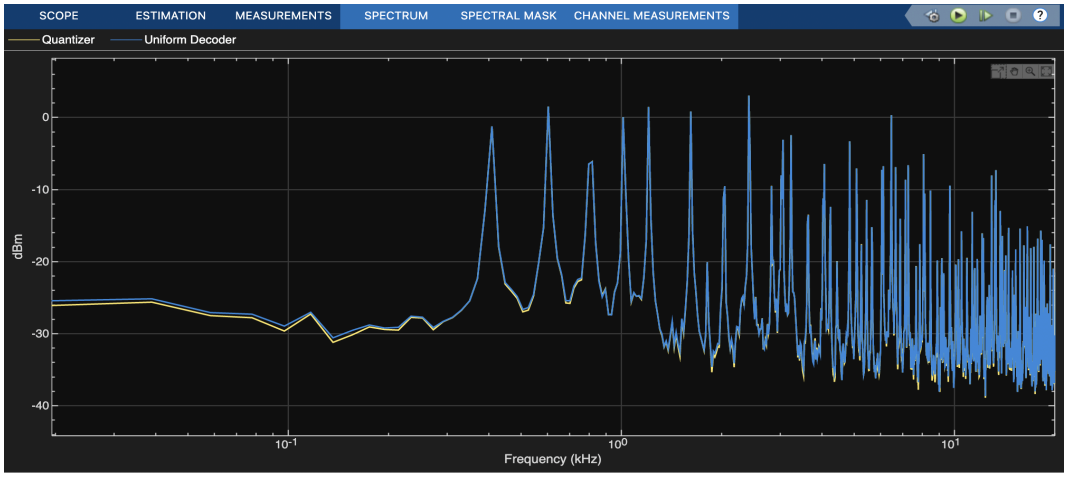


Figure 21: Spectrum analyser verifying the **source/sink subsystem** with SNR = 100dB

FPGA: For FPGA implementation, the evidence that we can provide is through practically demonstrating that our system works, which we already did during the Demo 2 presentation held on 20th June, 2024. Another evidence that we can provide is a video of the working system, that can be found by accessing the link to the google drive below:

Working Source Sink Demonstration.MOV

III. A/D and D/A converters

Simulink: For verifying the A/D and D/A conversion subsystem, we calculate the Bit Error Rate at the output of A/D and the input of the D/A converters, for just the subsystem as shown in figure 5. We can expect the BER to be less when the SNR is 100dB since the noise is negligible and the BER is more when SNR is 9dB.

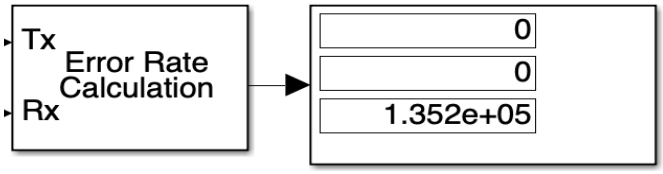


Figure 22: BER for SNR = 100dB

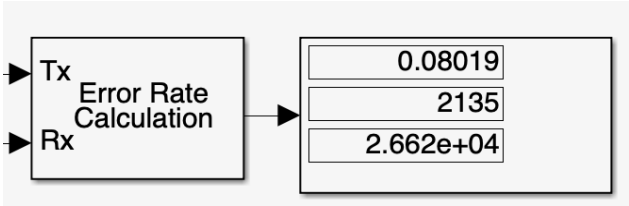


Figure 23: BER for SNR = 9dB

FPGA: For verifying that our ADC and DAC were correctly outputting the input with the expected sampling frequency of 8kHz, we had already created a downsampler block. We tested the downsampler block using a testbench and simulated it in ModelSim. We plugged-in decimal values from 100 to 2000, with an increment of 100 each, and we had made the design in such a way that the output would always take the 6th sample (as soon as the counter would hit 5)as the input, thereby taking only 600, 1200 and 1800 as the inputs as shown in the waveform below:

```
module downsampler(
input CLOCK_50,
input wire reset,
input wire [15:0] data_in,
output reg [15:0] down_data_out
);

 reg [2:0] count;

always@(posedge CLOCK_50 or posedge reset) begin
if(reset) begin
count<=3'b0;
down_data_out<=16'b0;

end else if (count == 3'b101) begin
down_data_out <= data_in;
count <=3'b0;

end else begin
count <= count + 3'b1;
down_data_out <= down_data_out;
end
end
endmodule
```

Figure 24: Downsampler module

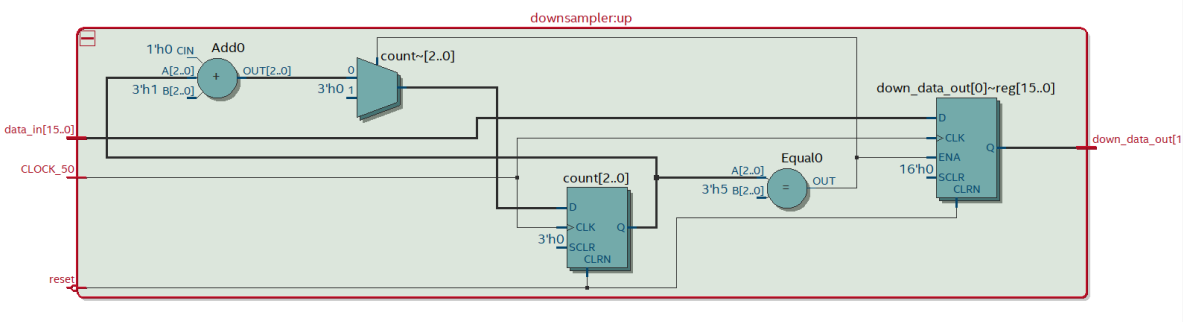


Figure 25: Block Diagram for the downsampler module

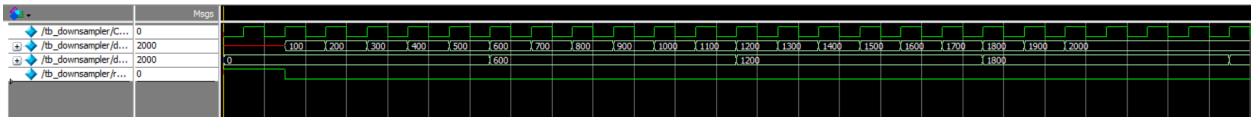


Figure 26: ModelSim waveform of the downsampler module, simulated using its testbench **(without annotations)**

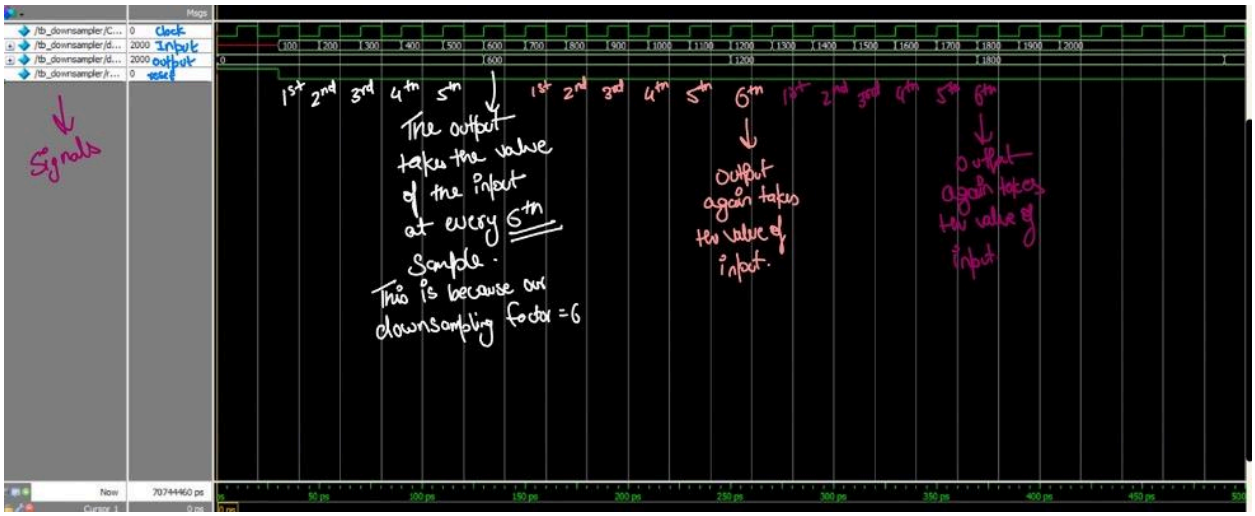


Figure 27: ModelSim waveform of the downsampler module, simulated using its testbench **(with annotations)**

For the quantization, as mentioned in the previous sections, we changed the parameters of the Audio CODEC for the system to operate at 16-bit, and also to reduce latency and resource utilization.

IV. Error correction encoder / Decoder

Simulink: For verifying the error correction subsystem we calculate BER based on the input of the encoder and output of the decoder for 9 dB and 100 dB SNR. The following figure shows the results. BER is 0 for both SNRs so only 1 image is shown for reference.

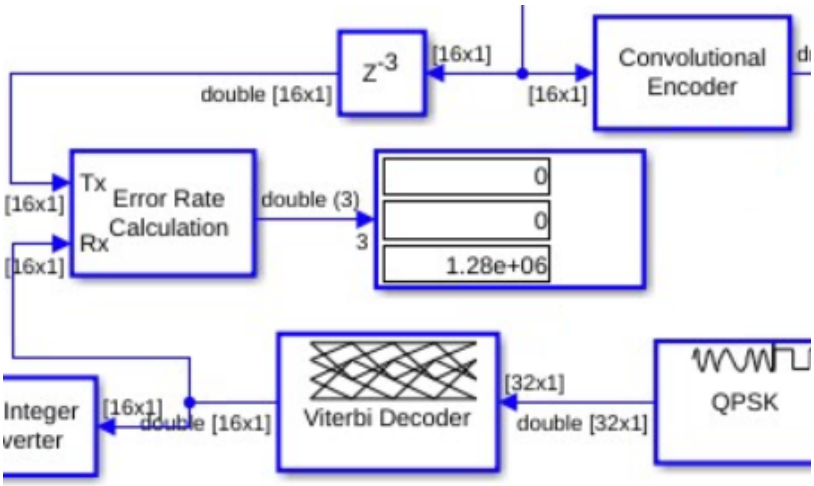


Figure 28: BER at 9 dB and 100 dB SNR

FPGA: For the FPGA implementation since BER couldn't be calculated from simulation in modelsim because Viterbi module was implemented from the IP catalog, a testbench was made to simulate the correctness of the convolutional encoder. The simulation below shows the encoding process. We can see the counter incrementing and updating temp_out when reset was turned low. After 16 cycles enc_bit is updated and we can see the correct output value of what we expected for the given input.

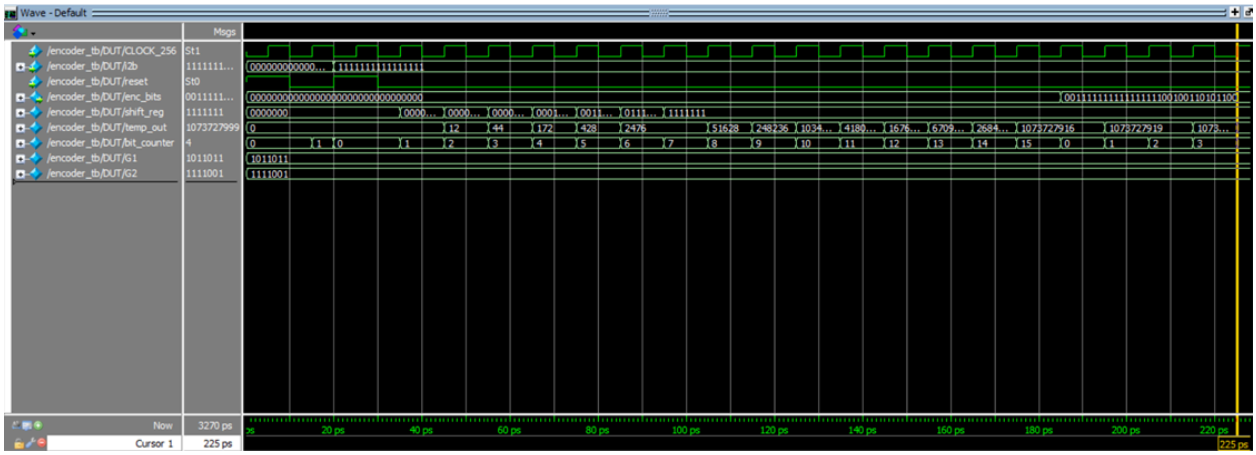


Figure 29: modelsim waveform of the convolutional encoder

The strategy was to calculate the expected value assuming a 16 bit input for a few tests and then use $random function in the testbench to test lots of different 16 bit inputs. Below is an example of the method used to calculate the expected output value.

Expected value calculation:

If i2b (input to our module) = 16'b0000111100001111

Parameters: K = 7, G1 = 7'b1011011 (171 in Octal), G2 = 7'b1111001 (133 in Octal)

Initial state:

- shift_reg = 7'b0000000
- temp_out = 32'b00000000000000000000000000000000

Iteration 1:

- shift_reg = {6'b000000, 1'b1} = 7'b0000001
- Output bit 1 (temp_out[0]) = ^(shift_reg & G1) = ^(7'b0000001 & 7'b1011011) = ^(7'b0000001) = 1
- Output bit 2 (temp_out[1]) = ^(shift_reg & G2) = ^(7'b0000001 & 7'b1111001) = ^(7'b0000001) = 1

Iteration 2:

- shift_reg = {5'b00000, 1'b1, 1'b1} = 7'b0000011
- Output bit 1 (temp_out[2]) = ^(shift_reg & G1) = ^(7'b0000011 & 7'b1011011) = ^(7'b0000011) = 0
- Output bit 2 (temp_out[3]) = ^(shift_reg & G2) = ^(7'b0000011 & 7'b1111001) = ^(7'b0000011) = 0

Summary of output bits for each iteration:

Iterations	Temp_out	value
1	temp_out[1:0]	2’b11
2	temp_out[3:2]	2’b00
3	temp_out[5:4]	2’b11
4	temp_out[7:6]	2’b10
5	temp_out[9:8]	2’b00
6	temp_out[11:10]	2’b00
7	temp_out[13:12]	2’b00
8	temp_out[15:14]	2’b00
9	temp_out[17:16]	2’b11
10	temp_out[19:18]	2’b00
11	temp_out[21:20]	2’b11
12	temp_out[23:22]	2’b10
13	temp_out[25:24]	2’b00
14	temp_out[27:26]	2’b00
15	temp_out[29:28]	2’b00
16	temp_out[31:30]	2’b00

Table 16: Output bits

Final output enc_bits = 32'b10101100101011001010110010101100. Which matches with the results seen on the simulation.

V. Modulator / Demodulator

Simulink: For verifying the modulator and demodulator subsystem we visualize the constellation scatter plot for modulator output and demodulator input at 9 dB and 100 dB SNR. The figures below show modulator constellation scatter plots.

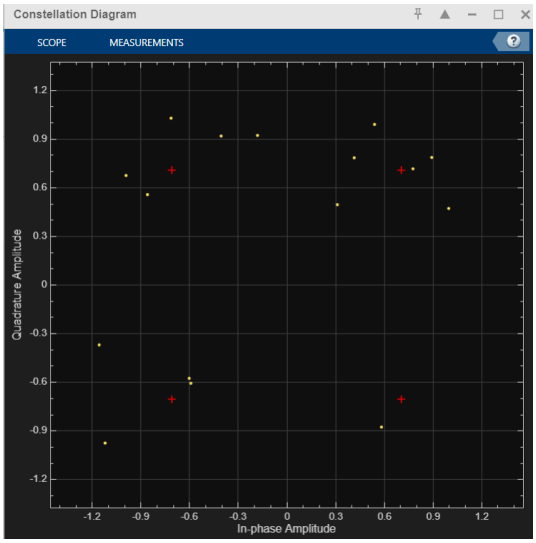


Figure 30: Modulator at SNR 9dB

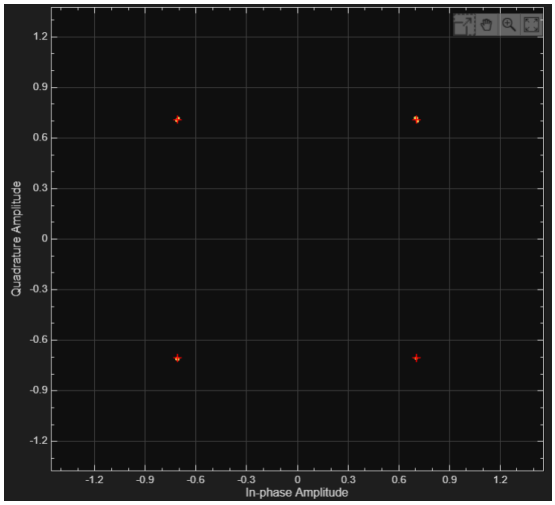


Figure 31: Modulator at SNR 100dB

When there is no noise at 100 dB SNR it is evident that the symbols are perfectly clustered at the 4 points. The figures below show demodulator constellation scatter plots.

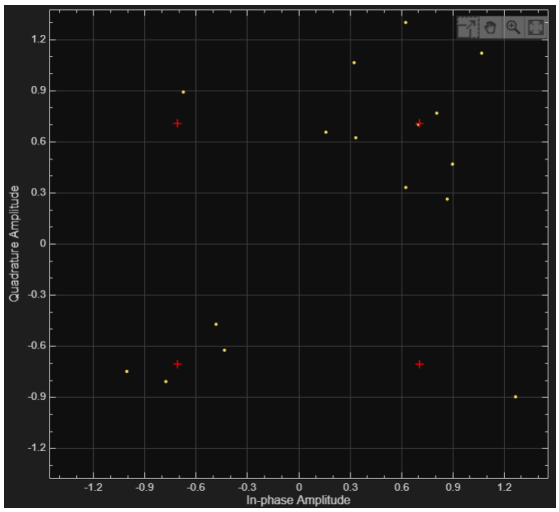


Figure 32: Demodulator at SNR 9dB

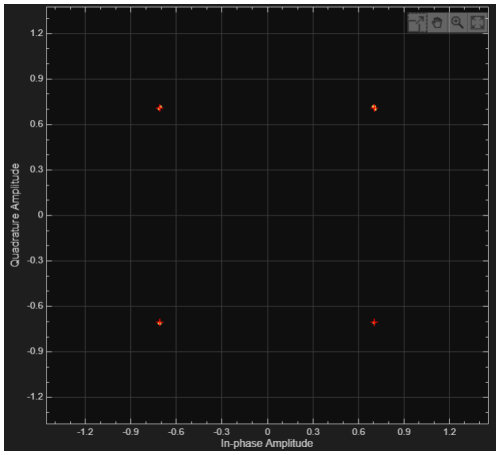


Figure 33: Demodulator at SNR 100dB

The modulator and demodulator constellation scatter plots at 9dB SNR indicate the errors induced from the channel. While the scatter plots are the same at 100 dB SNR because there is no noise.

FPGA: Testing using modelsim we noted the values of the output to expect when reset is toggled and checked the bits from the enc_bits input correctly outputted as I and Q. Also made sure that the analog representation of the signal I and Q looks like the signals observed in the Simulink model. The waveform below shows that modulator output correctly adjusts the symbol rate and correctly represents complex numbers.

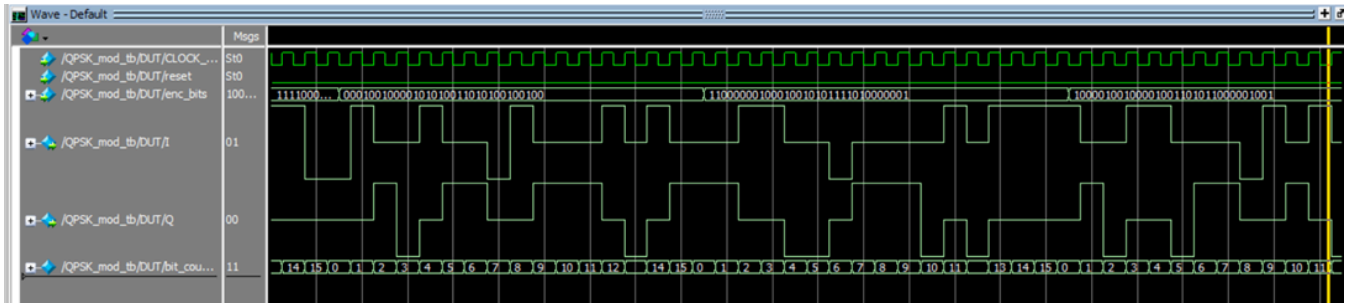


Figure 34: Simulation of QPSK modulator

For the demodulator we checked that the output demod_bits was being correctly outputted when any of the 4 combinations of I and Q were inputs. The simulated waveform below shows the expected results. Every clock cycle of this 32MHz clock, 4 bits of I and Q are demodulated and outputted as demod_bits.

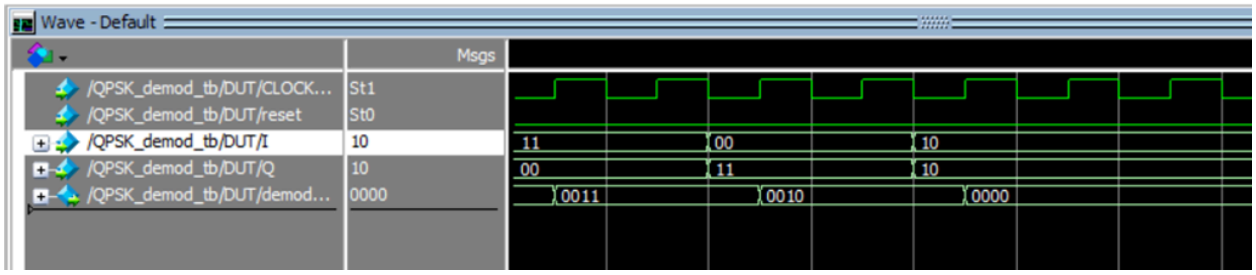


Figure 35: Simulation of QPSK demodulator

The waveform below compares the input to the modulator and output of the demodulator. Looking at 4 bit outputs of demodulated bits until 32 bits and checking with the 32 bit encoded bits that's the input to the modulator and using the BER module we can see that the BER is 0.145

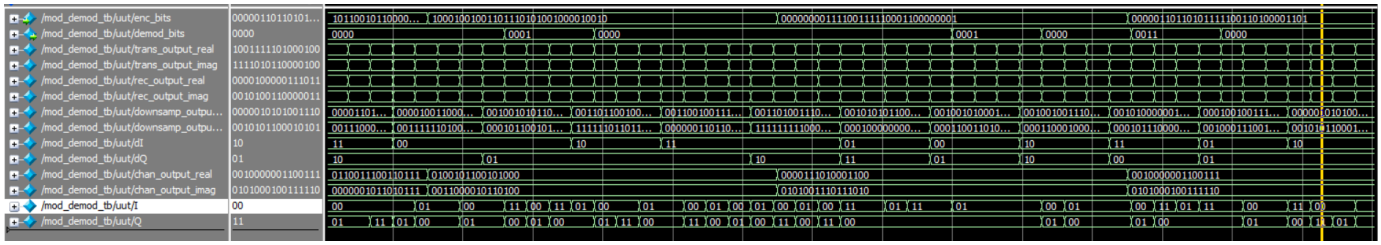


Figure 36: Simulation of whole system without encoder and decoder

VI. Transmitter / Receiver

Simulink:

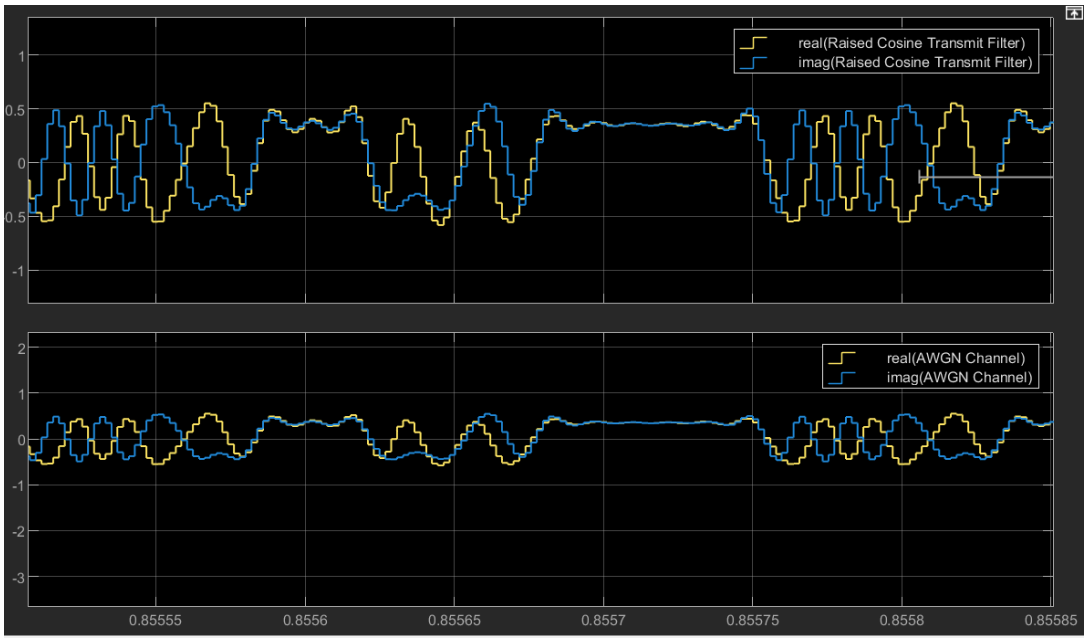


Figure 37: Time domain signal from output of transmitter(top) and output of channel (bottom) at 100dB SNR.

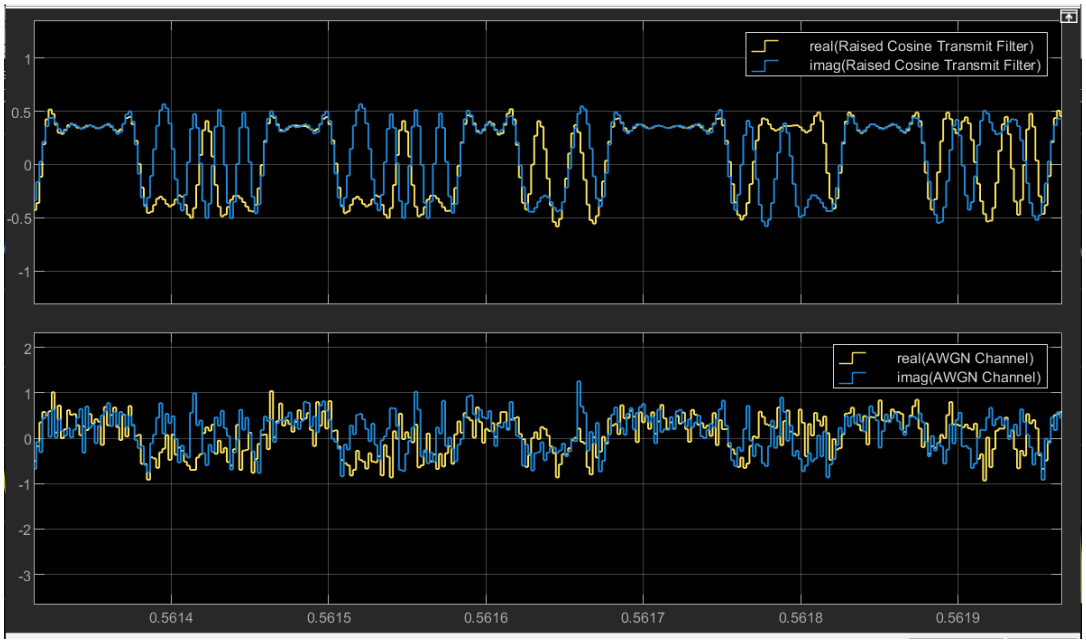


Figure 38: Time domain signal showing output of transmitter (top) and output of channel (bottom) at 9dB SNR.

The effect of the lower signal to noise ratio can clearly be seen by the distortion to the signal coming out of the channel in the 9dB SNR scenario.

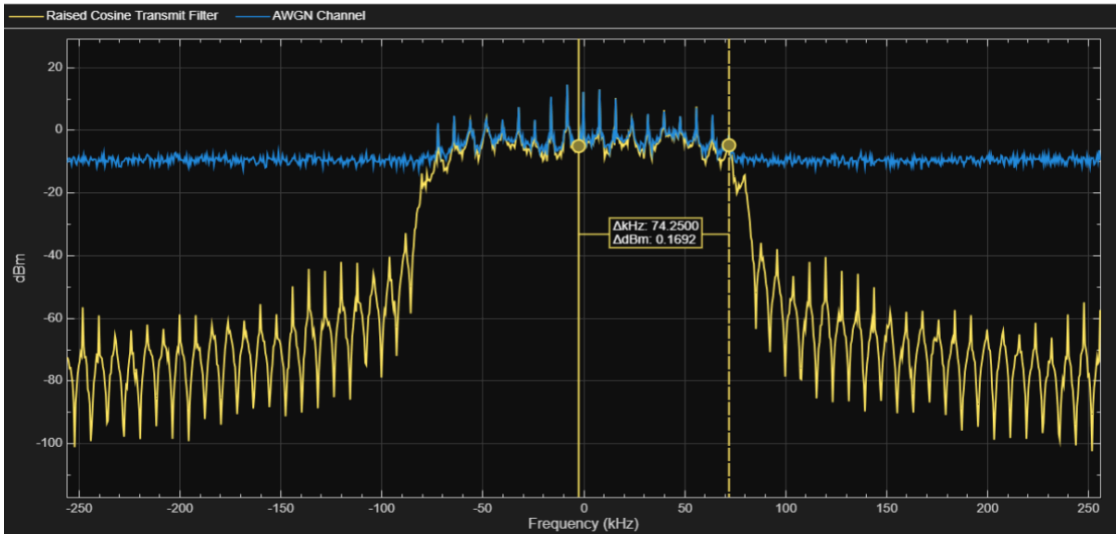


Figure 39: Spectrum analyzer at 9dB SNR.

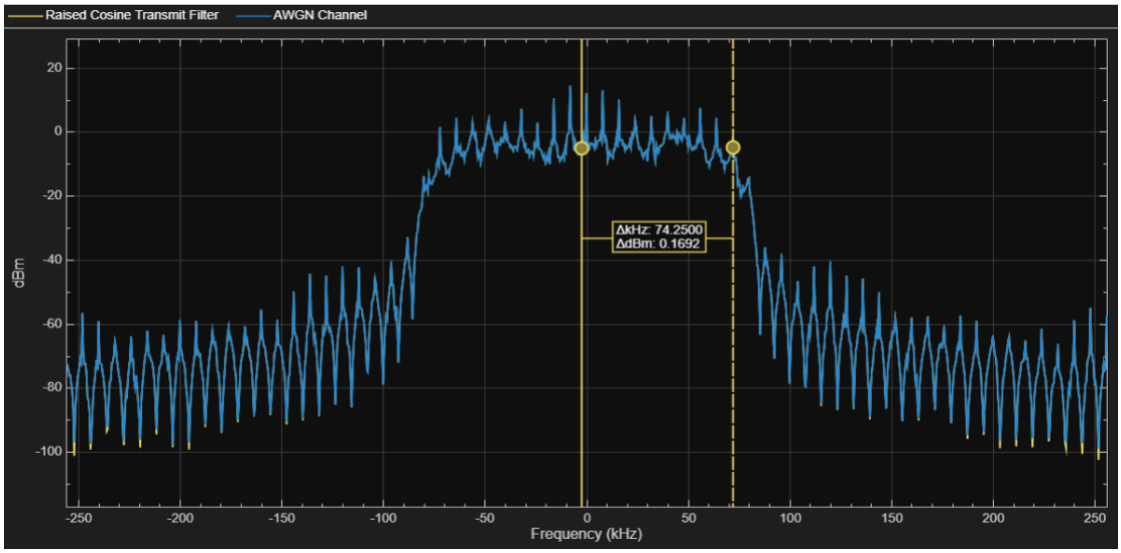


Figure 40: Spectrum analyzer at 100dB SNR.

The figures above show the frequency spectrum displaying the transmitted signal and the AWGN channel at 9 dB SNR and 100 dB SNR.

The fidelity of the signal is preserved much better at the higher signal to noise ratio. The Power of the transmitted signal came out to 12.4dBm, well below the restriction.

FPGA:

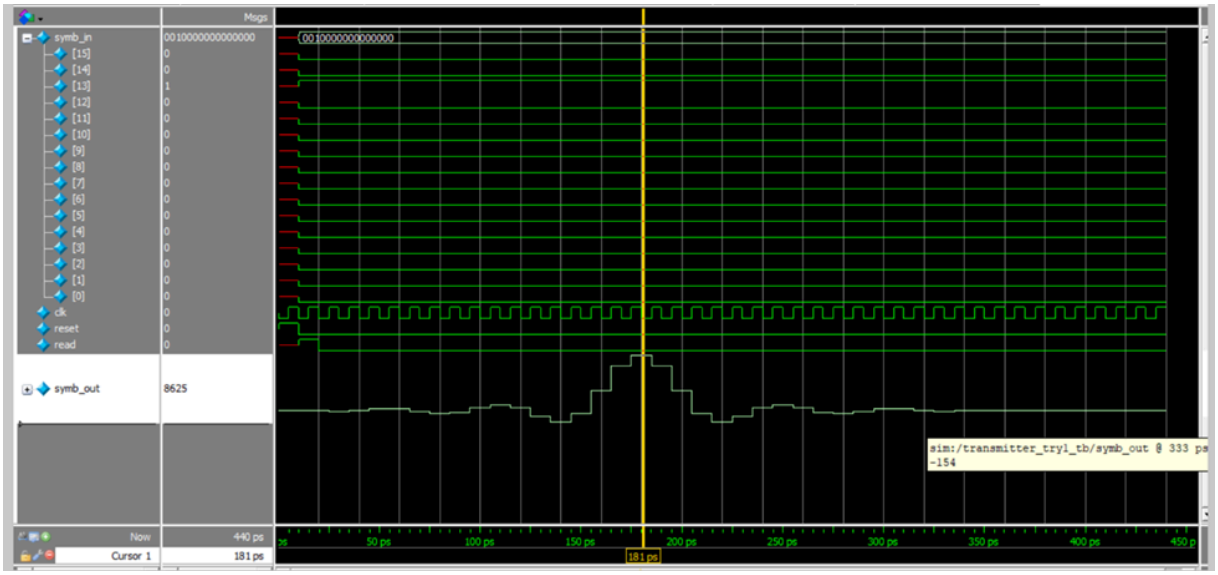


Figure 41: Testbench of transmitter

The transmitter was verified to be a finite impulse response (FIR) filter using the testbench above, an impulse was applied, a one followed by zeroes, and then checked to make sure it was returning the coefficients that were exported from matlab on the other side of the filter.

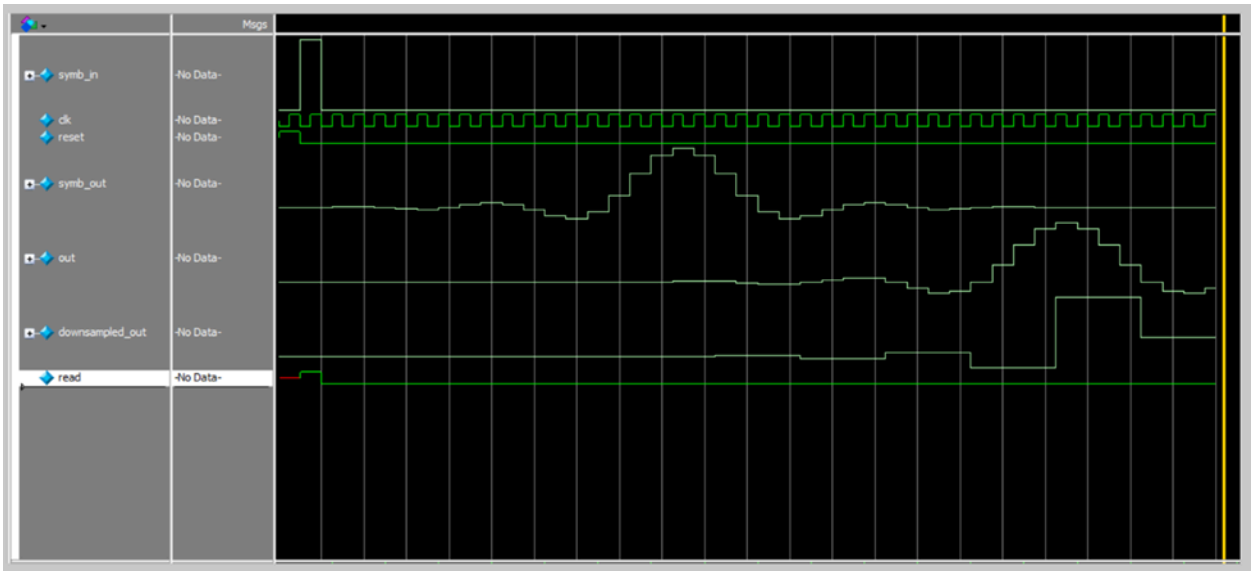


Figure 42: Testbench showing transmitter, receiver, and downsampler testing.

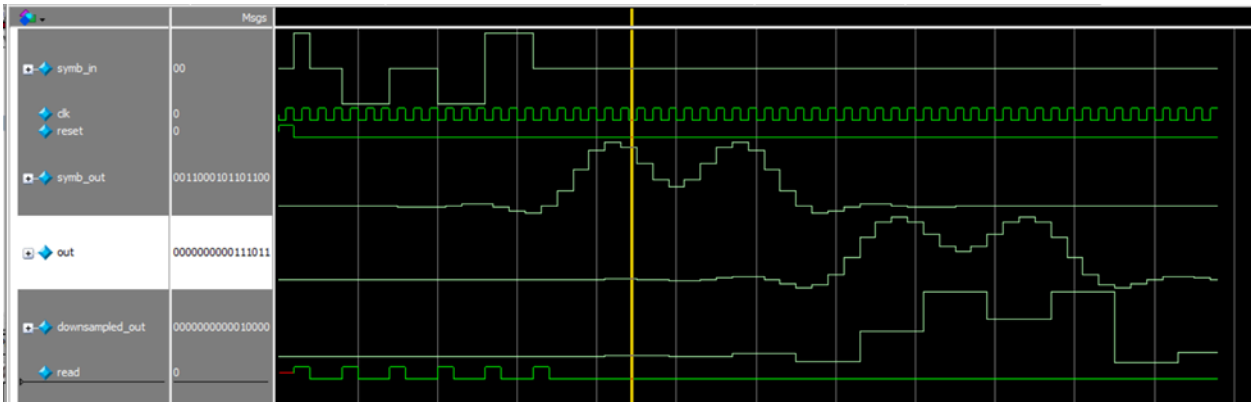


Figure 43: Testbench for ISI with transmitter receiver and downsampler.

The two testbenches above were used to verify that the transmitter, receiver, and downsampler are all working at the same time. We performed a test using just one symbol followed by zeroes, then another with every possible symbol that could be sent from the QPSK modulation to check cases as well as intersymbol interference. Output is as expected with the wave shape resembling that of the raised cosine, some delay present, and returning to a value close to the original after downsampling.

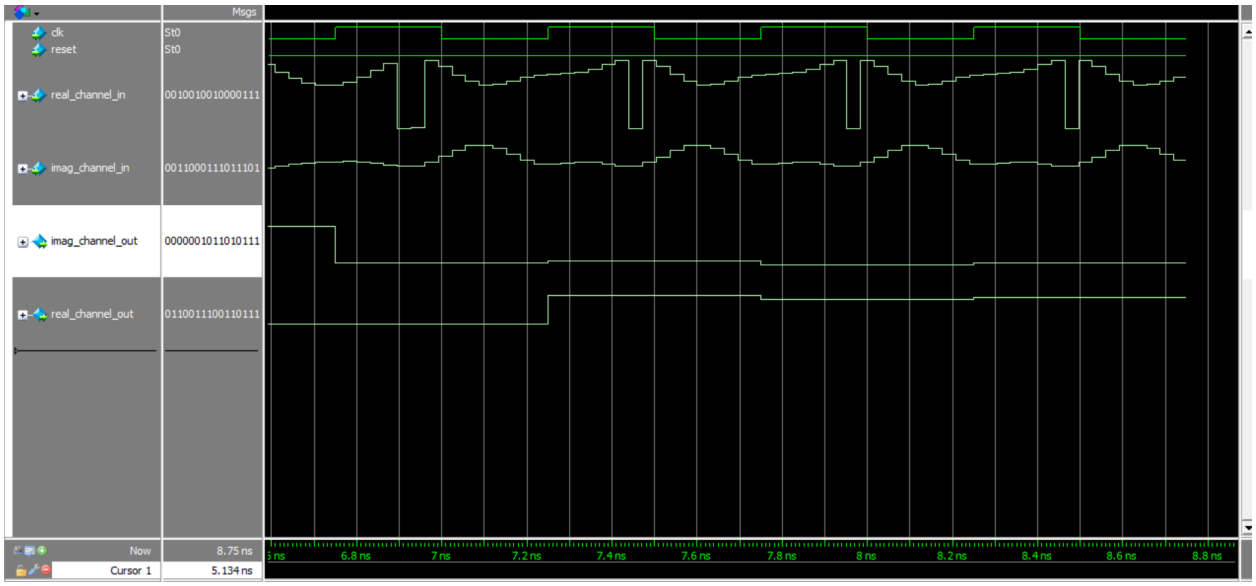


Figure 44: Testbench showing output of transmitter and input of receiver.

The testbench above shows the effect of the noise channel on the transmitter waves.

VII. Channel

Simulink: The variance was calculated as 0.35 via a variance block and display for 9dB SNR. The variance was calculated as 0.23 for 21dB SNR.

FPGA:

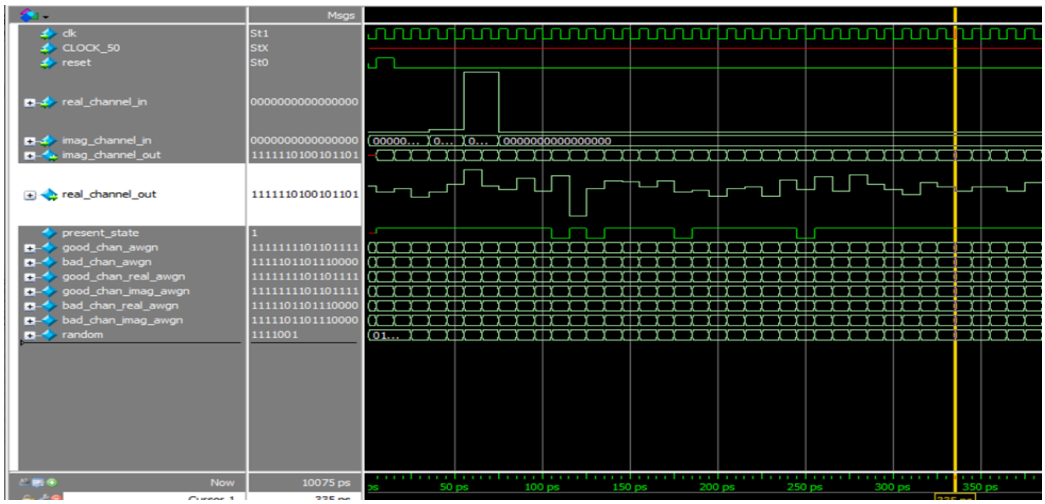


Figure 45: Testbench of channel.v

The above testbench of the channel indicates that a uniform distribution of gaussian noise is being added to the input channel, the present state signal indicates gilbert channel switching between good state and bad state. A basic channel input of zero was used and an arbitrary input pulse to monitor that the output of the channel is the same as the noise being added to it, the state machine is switching correctly in expected time intervals, and random numbers being used to calculate the probabilities are being produced randomly.

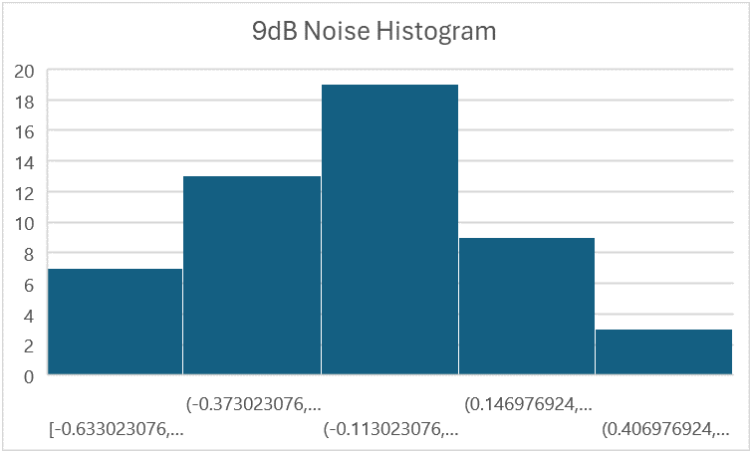


Figure 46: Histogram of n=50 Lookup table noise generated for 9dB SNR.

To verify that the noise follows a gaussian distribution, a histogram was generated using the 9dB SNR noise that was used for the lookup table. Despite some grouping performed by excel a normal distribution can be seen from the graph.

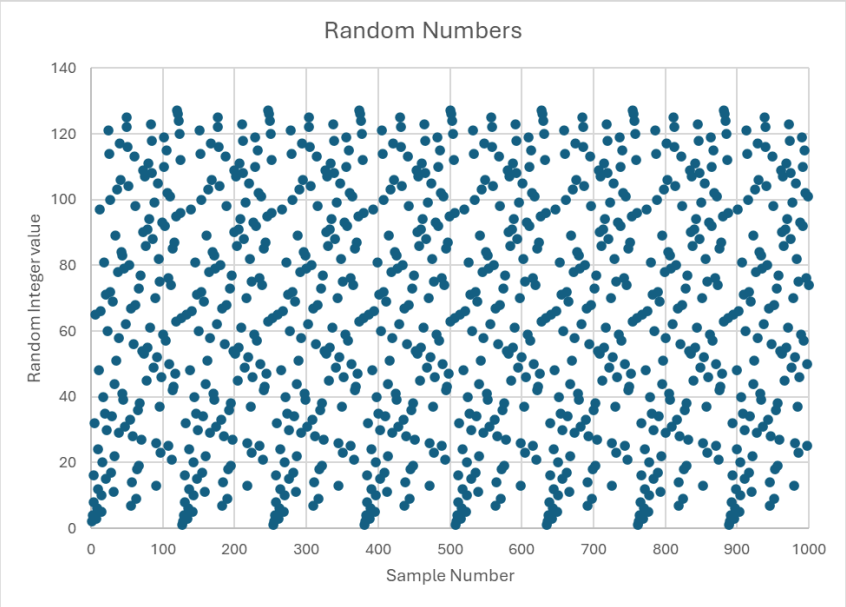


Figure 47: Plot of the random numbers generated using linear feedback shift register (LFSR).

To verify that the random numbers produced by the linear feedback shift register in the gilbert model are indeed random, they are graphed above and shown to follow a random pattern. The limitations of the LFSR are evident though as the numbers appear pseudo random with some local clustering at small denominations.

References

- Viterbi IP core guide
- Hoque, Khondker Shihabul & Mansur, Uddrity. (2022). Implementation of Convolutional Encoder and Viterbi Decoder in Verilog. 10.13140/RG.2.2.27135.48808.
- https://www.mathworks.com/help/signal/ug/fir-filter-design.html
- MATLAB Simulink Documentation
- ELEC391_ProductDocument.pdf
- ELEC391_Project.pdf
- ChatGPT was used to refine some of the codes, written material, and to understand a lot of technical things.
- ChatGPT inputs
    - Provide a structure for implementing a raised cosine filter in Verilog
    - Provide a base code for implementing the downsample module in Verilog
    - Provide a base code for implementing the clock divider in Verilog

Appendix

Appendix A

Deliverable	File Name	Notes
Product Document	391 Final Report	
Product Presentation	391 Final Presentation	
Simulink System File	Team7_Final_Simulink_Model.slx	Please declare a variable called snr =9 or snr=21 in the workspace before simulating
HDL Source Code, including testbenches and *readme files*	Folder called FPGA in the submission has all the files.	

Table 17. Deliverables' file name table

Technical Appendix B

Quartus Reports for the Entire System:

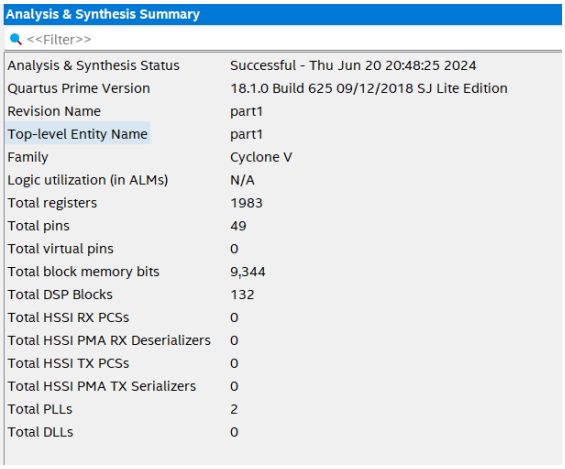


Figure 48: Analysis and synthesis summary from Quartus

Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimate of Logic utilization (ALMs needed)	1239
2		
3	⌄ Combinational ALUT usage for logic	1522
1	-- 7 input functions	18
2	-- 6 input functions	189
3	-- 5 input functions	158
4	-- 4 input functions	112
5	-- <=3 input functions	1045
4		
5	Dedicated logic registers	1983
6		
7	I/O pins	49
8	Total MLAB memory bits	0
9	Total block memory bits	9344
10		
11	Total DSP Blocks	132
12		
13	⌄ Total PLLs	3
1	-- PLLs	3
14		
15	Maximum fan-out node	KEY[0]~input
16	Maximum fan-out	1844
17	Total fan-out	14505
18	Average fan-out	3.80

Figure 49: Analysis and synthesis resource usage summary.

Multicorner Timing Analysis Summary

<<Filter>>

	Clock	Setup	Hold	Recovery	Removal	Minimum Pulse Width
1	⌄ Worst-case Slack	12.566	0.294	30.061	0.421	15.838
1	altera_reserved_tck	12.566	0.294	30.061	0.421	15.838
2	⌄ Design-wide TNS	0.0	0.0	0.0	0.0	0.0
1	altera_reserved_tck	0.000	0.000	0.000	0.000	0.000

Figure 50: Multicorner timing analysis summary

MATLAB Function block Code:

```
function [state, noise_level] = gilbert_fading(rand_num, state)%inputs are rand_num, state. Outputs are state, noise_level
    % Transition probabilities
    P_good_to_bad = 0.07;
    P_bad_to_good = 0.15;
    snr = 21;

    % Noise levels
    snr_good = 21; % snr in Good state
    snr_bad = 9;  % snr in Bad state

    switch state
        case 1 % Good state
            if rand_num < P_good_to_bad
                state = 0; % Transition to Bad state
            end
            noise_level = snr_good;
            assignin('base','snr',21); % set value of snr to 21
            %snr = 21;
        case 0 % Bad state
            if rand_num < P_bad_to_good
                state = 1; % Transition to Good state
            end
            noise_level = snr_bad;
            %snr = 9;
            assignin('base','snr',9); % set value of snr to 9
        otherwise
            state = 1; % Default to Good state
            noise_level = snr_good;
            %snr = 21;
            assignin('base','snr',21); % set value of snr to 21
    end
end
```

Figure 51: Matlab Function Block Code

Team Contributions

1. Bobby Smith

Individual Contribution: Made the transmitter/receiver and the noise channel plus gilbert fading in Simulink and FPGA. Assisted with combining parts in final FPGA design and debugging subsystem-combining issues.

Team Effectiveness: This team was very effective in scheduling, completing tasks and was very skilled in assisting each other when somebody was having difficulties with something. Harnoor was especially helpful with debugging issues, and Saksham was very good at scheduling soft deadlines to keep us on track.

Other Comments:

- **How did you ensure that your tasks would integrate with your group?**

Test-Benching as much as possible before trying to integrate parts together.

- **What hurdles did you have to overcome?**

Figuring out the transmitter and receiver was very difficult, but my group members worked with me to figure it out in the lab

- **What did you do to ensure success, or at least improve the likelihood of success?**

Communicate if you are having problems.

- **What did you learn?**

I learned a lot about the insane amount of detail that goes into digital communication in things like making a call with your cell phone

- **Were there any team dynamic issues?**

No, my group supported each other well, everybody worked very hard.

- **Anything else you spent your time on (related to the project)**

I watched a lot of engineering fundamental youtube videos that explained stuff well

2. Harnoor Saigal

Individual Contribution: Worked on encoder, decoder, modulator, demodulator in simulink and FPGA. Contributed in integration and debugging of simulink model and FPGA implementation. I wrote testbenches for encoder, decoder, modulator, demodulator, and the combined system. Added PLL and buffer module.

Team Effectiveness: The team was very effective in completing tasks and dealing with roadblocks in a timely manner.

Other Comments:

- **Were there any team dynamic issues?**

I maintained effective communication with my teammates and being updated on what they are doing

- **What hurdles did you have to overcome?**

I had to overcome understanding and implementing new concepts in a short amount of time. Especially using IP catalog

- **What did you do to ensure success, or at least improve the likelihood of success?**

To ensure success I made robust testbenches.

- **What did you learn?**

I learnt a lot about communication systems using simulink and how they can be implemented in the FPGA and what challenges can be observed in physical implementations

- **Were there any team dynamic issues?**

There were no team dynamic issues.

3. Saksham Mahajan

Individual Contribution: Worked on the A/D and D/A subsystem in both Simulink and FPGA. Wrote the module for Downsampler block and clock divider for implementing the whole system together. Wrote Testbenches for every module of mine and also helped with the noise channel in simulink and FPGA. Worked extensively on all the demo reports as well as all the team presentations. Helped in implementing, debugging, and testing the whole system in both Simulink and FPGA.

Team Effectiveness: The team was very effective in completing the project. All of us worked hard to get to the final outcome. We all helped each other whenever required, in spite of the roles assigned to each one. We took each other's work sometimes, just to balance off the pressure and workload.

Other Comments:

- **How did you ensure that your tasks would integrate with your group?**

I communicated thoroughly with them to ensure what data types their system would require, what approach they are adapting for their roles, etc

- **What hurdles did you have to overcome?**

I had to figure out a lot of stuff that was very new to me in a very short period of time, which was one of the things I had to manage to get the work done before the deadline. Some of the hurdles included: simulating the downsample block, making the simulink ADC output coherent with the rest of the system, and figuring out the parameters for the Audio CODEC.

- **What did you do to ensure success, or at least improve the likelihood of success?**

We had internal deadlines set that were scheduled way before the assignment deadlines, so that if any of us would run into trouble, we all could help him debug the system.

- **What did you learn?**

I thoroughly learned about the communication systems and this course also increased my confidence by a lot with using FPGAs

- **Were there any team dynamic issues?**

No, the team was very smooth in functioning and did not have any dynamic issues.