ELEC 391
Project

John Kim (44135151) - A/D and D/A Conversion, Input/Output format
John Ye (43883347) - Error Encoding/Decoding, Modulation/Demodulation
Mohkam Malhi (49155500) - Transmitter/Receiver, Channel

Objectives, Requirements, and Constraints

Our objective for this project is to design a functioning communication system and verify that it meets the design conditions set for this project. We approached this challenge by creating and simulating a Simulink model, followed by an FPGA implementation of our design. However, there were times where we had to deviate from our original system design in order to implement a system on hardware within our time and resource constraints, and we have updated our original Simulink model as required to reflect those changes.

Audio Bandwidth	Bit error rate	Spectral Mask	Channel	Time delay
4kHz	10^-5	150kHz	δ	15ms

Table 1: The design specification for our design

The basic model of how our communication system works is represented by this figure from the ELEC 391 project handout:

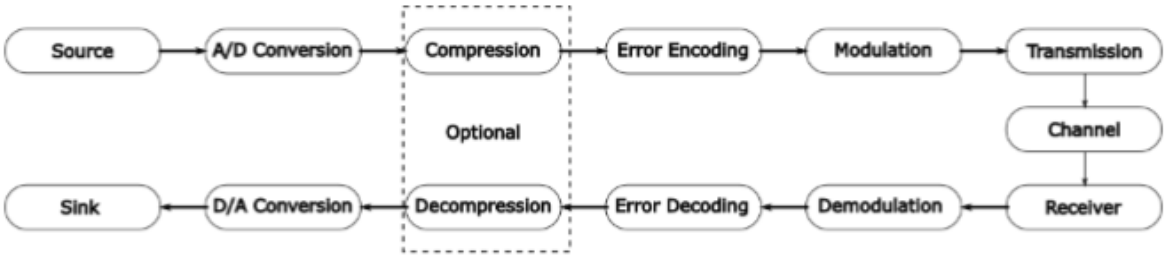


Figure 1: Basic communication system model from ELEC 391 Project handout

The project is then splitted into different subsystems, which were tested extensively before integrating with other subsystems.

There were some challenges and design decisions that arose in the process, described as follows:

We had to make a design decision when designing our transmission filter. As we were using a Square Root Raised Cosine filter, we had to decide how we could implement it, and ended up deciding on utilising lookup tables. Separately, as the channel had a different clock frequency we had to switch clocks at the channel, and also decided to implement a FIFO as it would help reduce error rate. Another decision we had to make was between QPSK and 16-QAM. We ended up deciding on QPSK due to the lower error rate despite it meaning a slower symbol rate being transmitted.

Communication System Design

System Overview

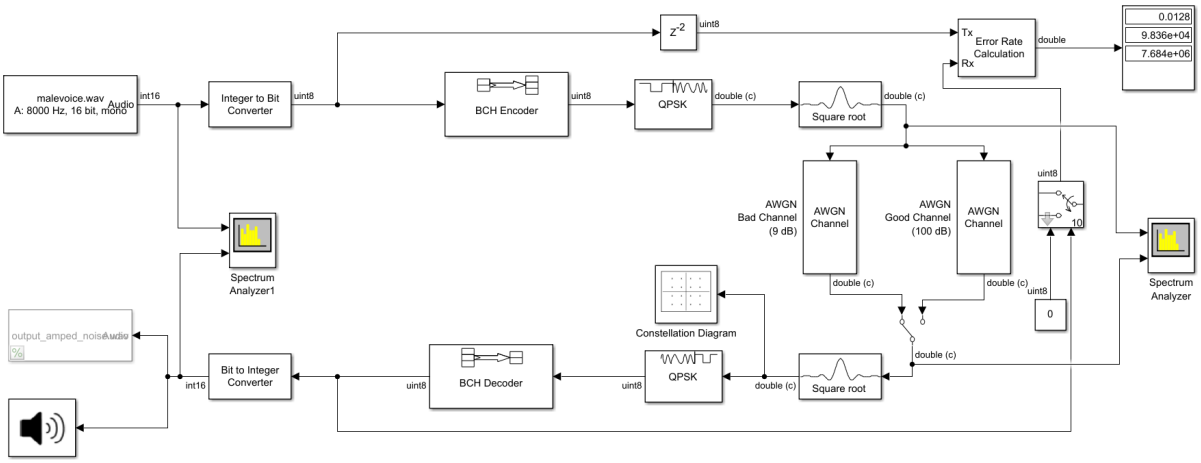


Figure 2: Complete Simulink Model of the communication system

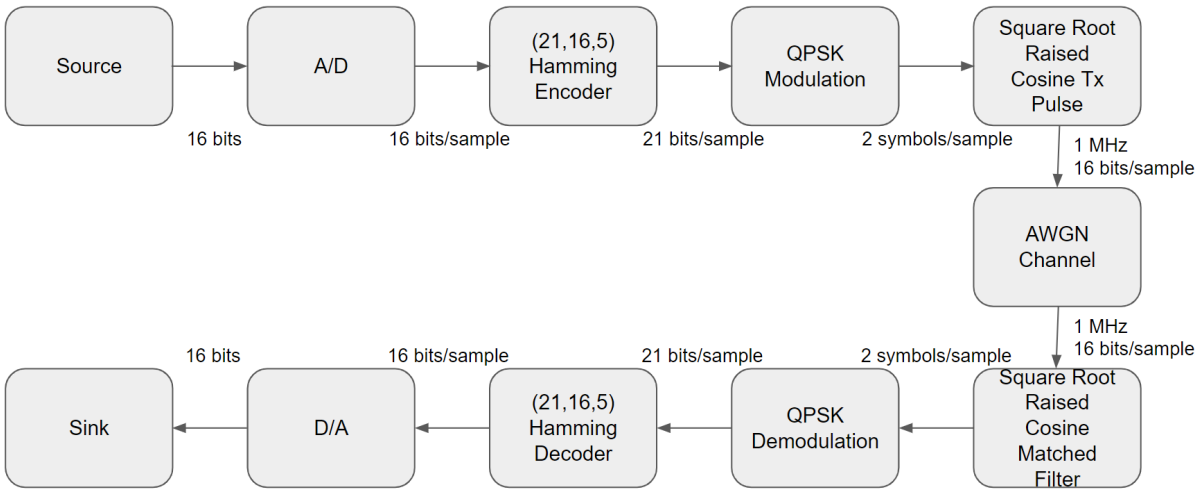


Figure 3: Block diagram representing our system design

On Simulink our source is represented as a From Multimedia File and our sink is a speaker output and a To Multimedia File. This matches what we would see with a hardware implementation, with one difference being an additional microphone input for the source.

We found A/D and D/A converters to be simple to implement on our Simulink model, utilizing Integer to Bit Converter and Bit to Integer Converter.

We designed our system to implement a (21,16,5) Hamming encoder and decoder, a tradeoff we made for the design to be more implementable on hardware in exchange for a higher error rate than if we had used a BCH encoder/decoder. This is because BCH has the capability of correcting up to three errors at once [2] compared to Hamming which can only do one at a time, which would result in a better Bit Error

Rate and overall speed of the system. As Simulink does not allow for modelling a (21,16,5) Hamming encoder, we had to represent it in the model as a (21,16,5) BCH encoder.

We initially modelled a 16-QAM Modulator/Demodulator in Simulink to increase data transmission rate, however in our hardware implementation it was necessary to utilize QPSK modulation/demodulation to lower our Bit Error Rate [3], coming at the expense of speed.

Modelling Square Root Raised Cosine filter blocks helped make it easier to adjust the bandwidth by changing the rolloff factor and filter span in the parameters [4]. However, it was more difficult to implement our desired filter span in our hardware implementation than we had anticipated, so we had to settle for a value less than desired, resulting in a higher bit error rate than we had hoped for.

As our scenario has a 0 transition probability from the bad to good state, we represented the Gilbert channel between 9 dB and 21 dB using a manual switch.

Criterion	**Simulink Perf.**	**FPGA Perf.**
Message Transmission (bit rate)	128 Kbps	16
Transmission Reliability (bit error probability)	0.01281	
Processing Delay	80.735 ms	
Channel Bandwidth	2 kHz	

Table SO1: System performance table

Subsystem Design

1. Source and Sink:

For our source block, we employed a From Multimedia File block in Simulink as shown in Figure 4. We set the output of the source as 16 bit integer with 1024 samples per channel. For the sink, we used a To Multimedia File block with 16 bit audio data output type as shown in Figure 6. The reason why we decided to use 16 bits was because it was possible to have a better sound resolution with 16 bits than sound samples had we used a lower number of bits. With 16 bits, we could represent 65536 amplitude levels which are higher than sound samples with 8 bits or lower. Although the tradeoffs include more computation time and allocation of system resources, we felt it was a reasonable compromise in this case.

It is true that the double data type was available. A double contains 32 bits. With more bits, it was possible to have a larger amplitude spectrum. However, it could take longer computation time from other blocks for processing 32 bits sound samples. Having 16 bits/sample results in a better storage efficiency than 32 bits/sample with the fixed sampling frequency of 22.05 KHz as the 32 bits/sample one would require twice larger storage.

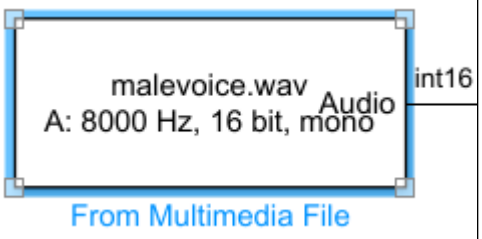


Figure 4: From Multimedia File block

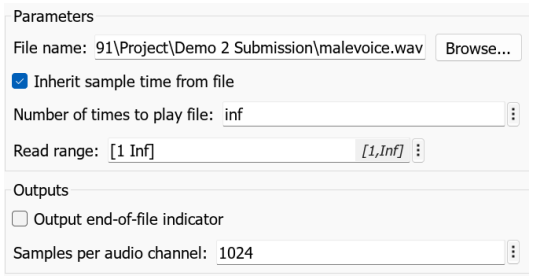
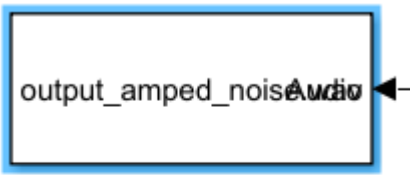


Figure 5: From Multimedia File block parameters



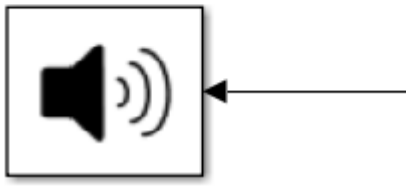


Figure 6: To Multimedia File and Audio Device Writer blocks

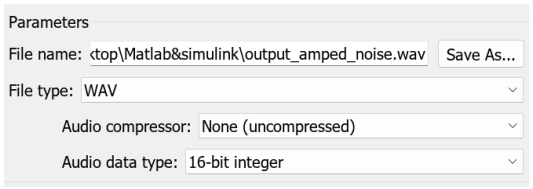


Figure 7: To Multimedia File block parameters

Table SD2: Source/Sink Simulink blocks with parameters

FPGA implementation:

1. Block diagram

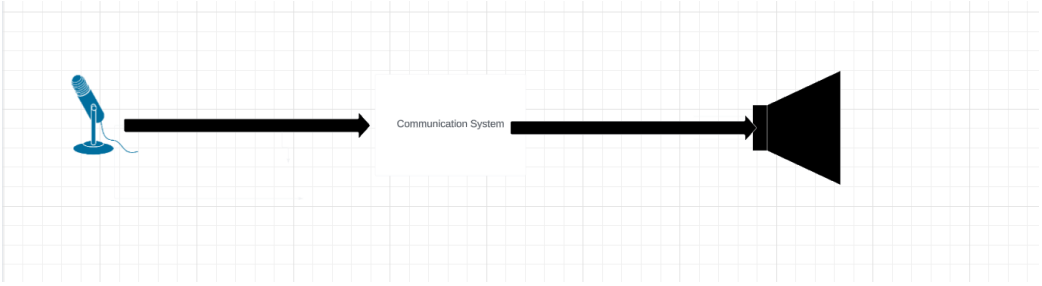


Figure S/S1. Source to sink block diagram

2. *Design justification*

Although our group could not implement a module that could store and play a wav file, we were able to use the microphone and speaker to produce sound by employing the altera audio codec module. We initially planned to create a module for loading a wav file in the ROM of DE1-SOC and sampling from these data. However, within a limited timeframe, we had tasks with higher priority such as building a channel. Therefore, we decided to use the microphone and speaker.

3. *Subsystem components and custom blocks*

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

2. A/D and D/A conversion

We used an integer-to-bit and a bit-to-integer to convert signals from analog to digital and vice versa. We had the blocks inputting/outputting 16 bits per integer value to match the source and sink, which were also 16 bits (signed) long. Similar to the source and sink blocks, we chose 16 bits because of less computation time from some of the other blocks in our communication system and also its ability to represent a relatively wide spectrum of sound amplitude with higher storage efficiency per sample.

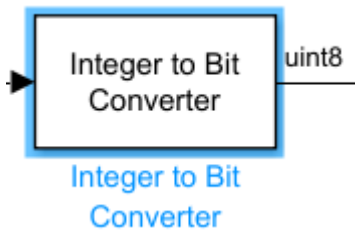


Figure 8: Integer to Bit Converter block

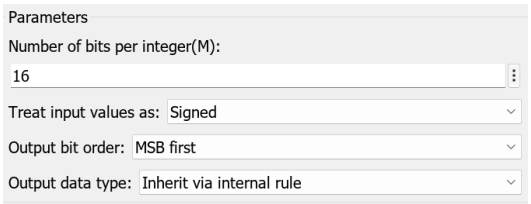


Figure 9: Integer to Bit Converter block parameters

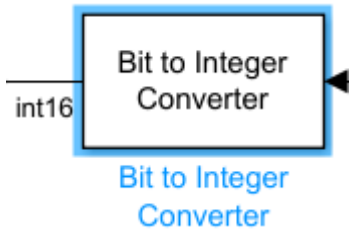


Figure 10: Bit to Integer Converter block

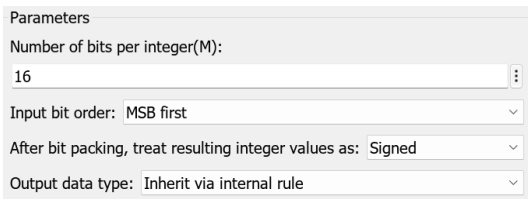
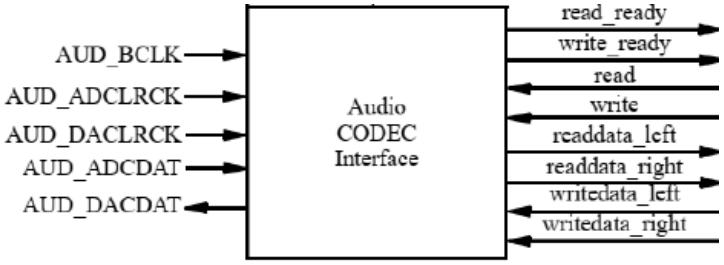


Figure 11: Bit to Integer Converter block parameters

Table 3: A/D and D/A Converter Simulink blocks with parameters

FPGA implementation:

1. Block diagram



2. Design justification

We used the altera codec library for implementing our A/D D/A converter. The interface is provided in our FPGA starter assignment. We modified the data width from 24 bits to 16 bits which matches our design requirements.

The settings for this audio block are shown in table 3. The instantiation and the settings are shown in figure 4 and figure 5. The reason why we set the number of counter init as 15 was because there were 15 bits of sound sample to represent the amplitude of a sound while the very zero index bit of the sound sample was a sign bit. We did not change the sample frequency as 48 kHz was an industry standard. The audio codec block diagram is shown in figure 14.

Bits/sample	16 bits/sample
Sample Frequency	48 kHz

Table 4: FPGA implementation settings of ADC and DAC modules

```
audio_codec codec(
   // Inputs
   CLOCK_50,
   reset,

   read, write,
   writedata_left, writedata_right,

   AUD_ADCDAT,

   // Bidirectionals
   AUD_BCLK,
   AUD_ADCLRCK,
   AUD_DACLRCK,

   // Outputs
   read_ready, write_ready,
   readdata_left, readdata_right,
   AUD_DACDAT
);
```

Figure 12 Audio codec instantiation

```
/*****************************************************************************
 *                           Parameter Declarations                          *
 *****************************************************************************/

parameter AUDIO_DATA_WIDTH = 16;
parameter BIT_COUNTER_INIT = 5'd15;
```

Figure 13 Audio codec settings change

3. *Subsystem components and custom blocks*

The subsystem components of this interface included many sub modules of the altera codec, which is included in the submission.

3. Error Encoding and decoding

Error encoding and decoding is an important part of a digital communication system. To implement this system in FPGA, we first simulated using matlab simulink blocks with the following parameters:

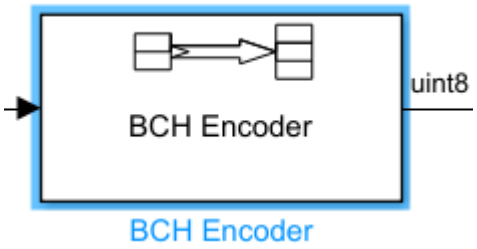


Figure 14: BCH Encoder block

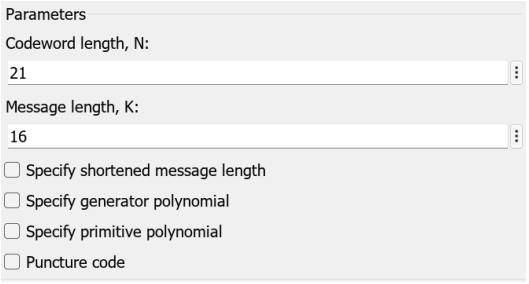


Figure 15: BCH Encoder block parameters

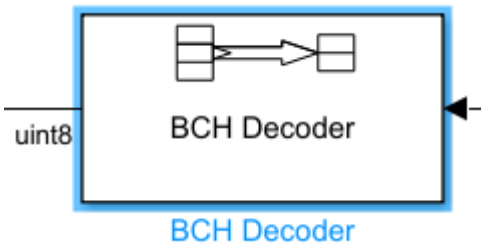


Figure 16: BCH Decoder block

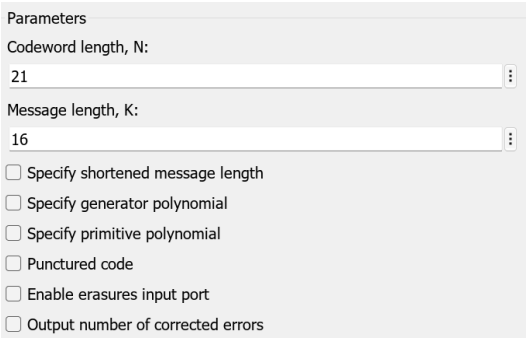


Figure 17: BCH Decoder block parameters

Table 4: Error Correction Encoder/Decoder Simulink blocks with parameters

We used the above blocks in simulation because we couldn't simulate a hamming encoder and decoder with the specified parameters of 16 message length and 21 codeword length. The above simulink block provides identical functionality as a (16,21) hamming blocks.

FPGA Implementation:

1. Block diagram

For our design, the error encoding is implemented using combinational blocks.

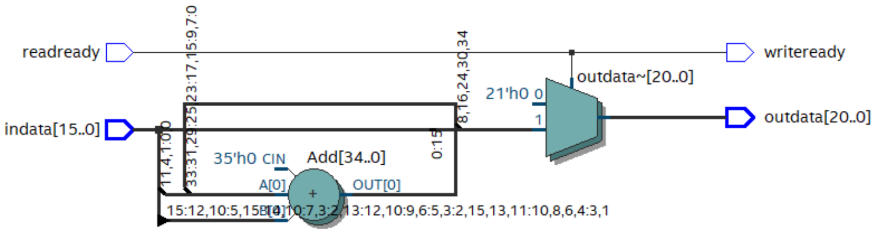


Figure 18: Encoder block diagram

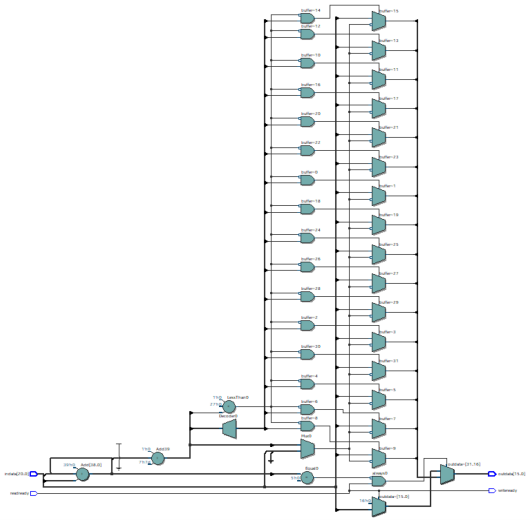


Figure 19: Decoder block diagram

2. Design justifications

Initially, we chose a BCH encoding method which used a 16 bits message and a 31 bits encoded data. This approach is able to detect and correct 3 errors, which would be the ideal choice. However, upon further research, we found that the BCH encoding method is very difficult to implement on the FPGA as it uses many binary multiplication and division. In comparison, the Hamming encoding method with 16 bits message length and 21 encoded data length is a lot easier for implementation (as it is entirely combinational) , albeit it only corrects 1 error at a time.

3. Subsystem components and custom blocks

```
Hamming_encoder encode (datain,encode_out,en_readready,en_writeready);
Hamming_decoder decode (decode_in,dataout,de_readready,de_writeready);
```

Here is the module instantiation for the encoding subsystem. It contain four signals input and outputs and they are connected to the source/sink and the modulation/demodulation respectively.

4. Modulation and demodulation

We decided to use 16-QAM modulation as it transmitted more bits per symbol with a higher symbol rate than QPSK modulation. As 16-QAM modulation had the higher symbol rate, as a trade-off, we expected to have larger error as the distance between two 16-QAM codes became smaller than the QPSK case. However, we observed that this transition barely changed the error rate over 60 seconds of simulations. Both cases yielded bit error rates in the range of 10^{-5}. Also, as the symbol rate increased, the frequency range of modulation became wider. To respond to this change, our group increased the roll-off factor of the raised cosine filter from 0.2 to 0.4 to cover a larger frequency range. However, this did not impact the bit error rate. It stayed within the range of 10^{-5}. Also, these errors could be corrected with the help of the error encoder and decoder. By upgrading from the Hamming encoder to BCH encoder as discussed in the error encoding and decoding section, the error encoder and decoder reduced this potential bit error rate.

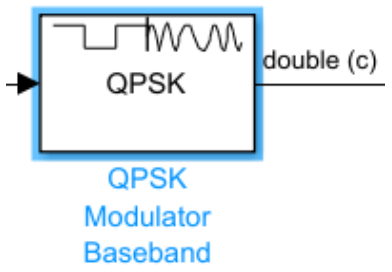


Figure 20: QPSK Modulator block

Figure 21: QPSK Modulator block parameters

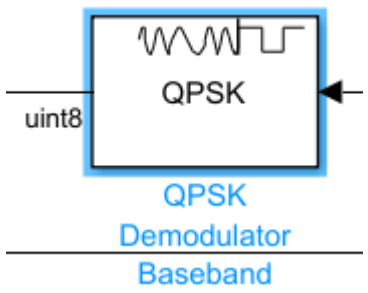


Figure 22: QPSK Demodulator block

Figure 23: QPSK Demodulator block parameters

Table MD5: Modulator/Demodulator Simulink blocks with parameters

FPGA Implementation

1. Block diagram

The block diagram for the QPSK modulator block is given in figure F.1 below. The QPSK demodulator is given in figure F.2.

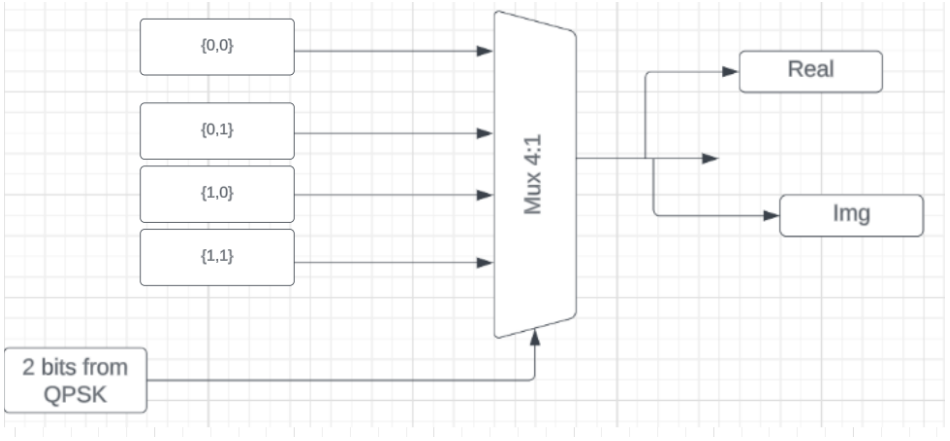


Figure F.1 QPSK modulator block diagram

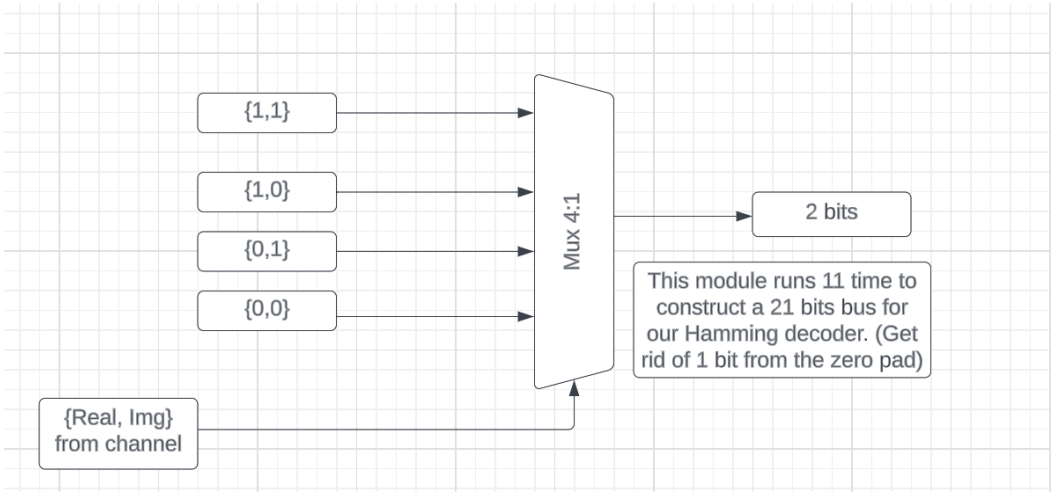


Figure F.2 QPSK demodulator block diagram

2 Design justification

Binary representations of QPSK modulator outputs (00, 01, 10, 11) were inputted into a multiplexer. These binary bits were taken by splitting a Hamming encoded 21 bits + zero pad into pairs of 2 bits. This multiplexer mapped each binary representation into a complex number from the constellation diagram. Then each complex number was split into the real and imaginary parts of this complex number. The QPSK demodulator works in the opposite way. It takes 2 bits every time from the receiver which are binary numbers representing complex values from the channel. Then, it maps back to the binary representation of QPSK.

3. Subsystem components and custom blocks

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

5. Transmitter and receiver

For the transmitter and the receiver, we chose to use square root raised cosine filter blocks. It was efficient and convenient for us to use raised cosine filter blocks because we could adjust our filters to the bandwidth of our channel by changing the roll off factor and filter span of raised cosine filter blocks.

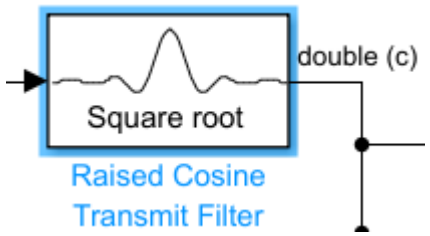


Figure 20: Raised Cosine Transmit Filter block

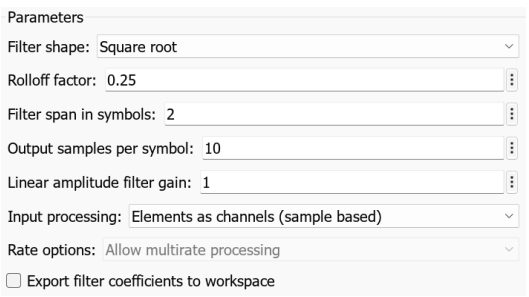


Figure 21: Raised Cosine Transmit Filter block parameters

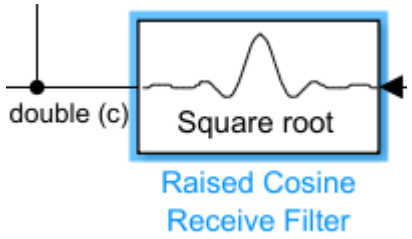


Figure 22: Raised Cosine Receive Filter block

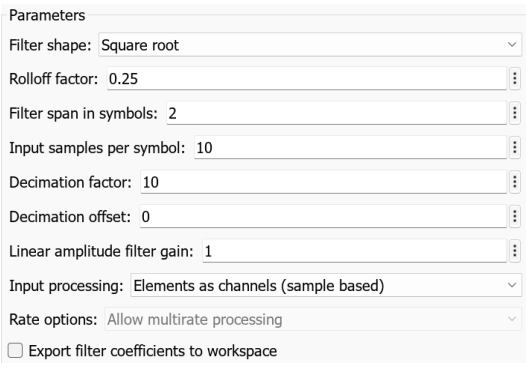


Figure 23: Raised Cosine Receiver Filter block parameters

Table TR6: Transmitter/Receiver Simulink blocks with parameters

(Note: We tried to use the RTL viewer on Quartus to correctly display our transmitter. However, it looked too complex. So we simplified it.)

FPGA Implementation:

1. *Block diagram*

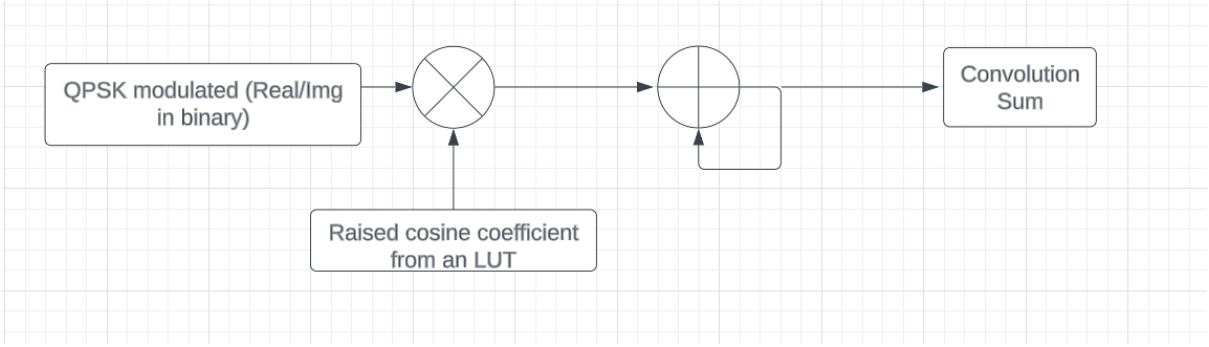


Figure T/R1 Transmitter block diagram

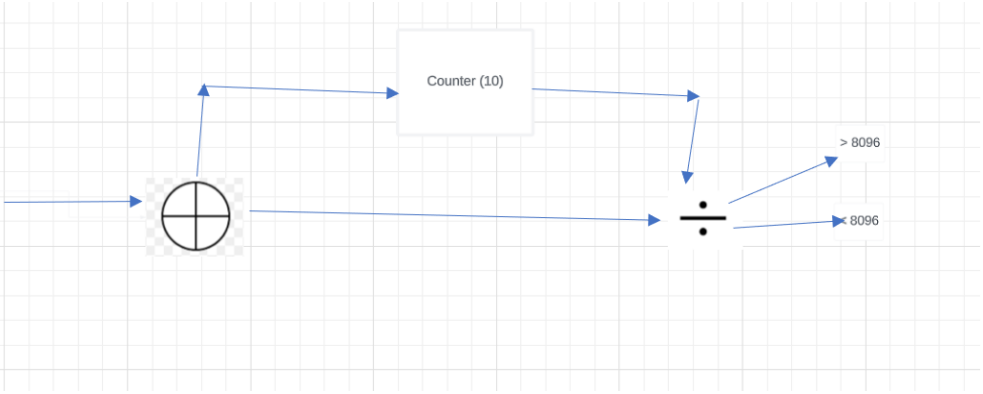
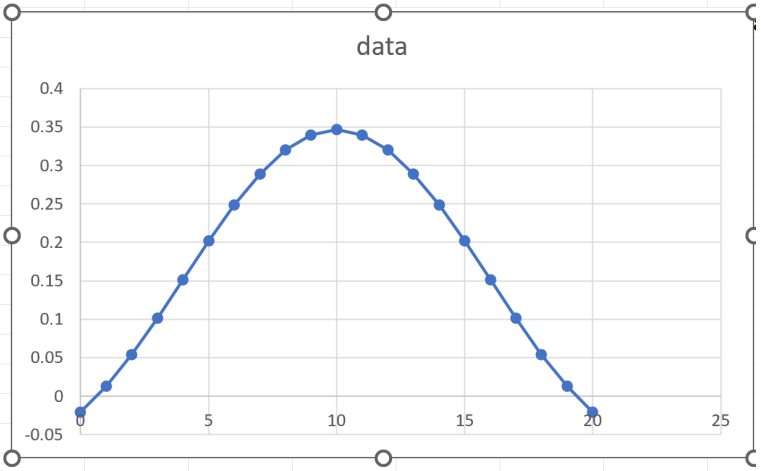


Figure T/R2 Receiver block diagram

2. *Design justification*

We used the coefficient exported from the simulink model to create our design of the transmitter/receiver. Here is the data, obtained from a raised cosine filter transmitter with filter span of 2 and output 10 samples per symbol:

Sample	data	
0	-0.02082	-1665.58
1	0.012759	1020.738
2	0.054111	4328.884
3	0.101202	8096.158
4	0.151371	12109.7
5	0.201529	16122.33
6	0.248403	19872.27
7	0.288811	23104.87
8	0.31993	25594.42
9	0.339547	27163.72
10	0.346247	27699.77
11	0.339547	27163.72
12	0.31993	25594.42
13	0.288811	23104.87
14	0.248403	19872.27
15	0.201529	16122.33
16	0.151371	12109.7
17	0.101202	8096.158
18	0.054111	4328.884
19	0.012759	1020.738
20	-0.02082	-1665.58



We chose the above parameter with consideration of the FPGA compatibility as well as the system resource. We decided that these parameters will be the

best for implementing in the FPGA.With the coefficients, we decided to create a lookup table to transform the bits into a transmissible waveform.

3. Subsystem components and custom blocks

The transmitter usually has the clock frequency adjustments, but we decided to incorporate that with the channel design in the form of FIFO memory.

The transmitter module consists of two blocks: one that receives the bit data input and one that outputs the waveform. The receiver does that in the reverse order.

6. Channel

For our design, δ channel model was provided. The only state of error available from Gilbert δ channel model is the bad state. The bad state has a SNR of 9dB and $10^{9/10}$ signal to noise ratio, with an input power of $1W$. As such, we can use an AWGN channel with the parameters above to satisfy our design requirement.

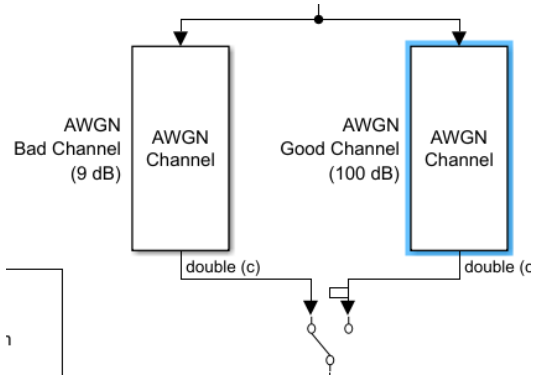
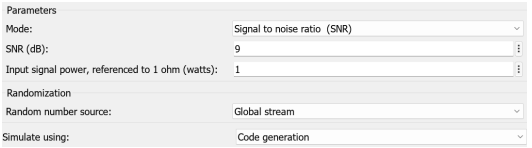


Figure 24: AWGN Channel blocks

Bad Channel parameters



Good Channel parameters

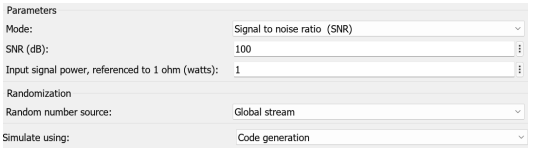
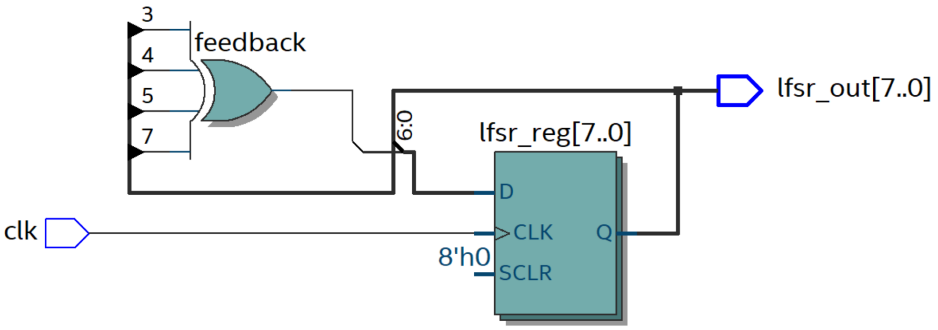


Figure 25: AWGN Channel block parameters

Table C7: Channel Simulink blocks with parameters

FPGA Implementation

1. Block diagram



2. *Design justifications*

To meet the required channel frequency of 1MHz, we decided to use dual ports FIFO memory with different read and write clock signals. To create clock signals with different speeds, we used a clock changer to change the clock from 50MHz (operation clock) to 1MHz channel clock.

We used LFSR for the noise generation as it is easy to implement it in the FPGA. We added 8 bits of zeros to a random 8 bit LFSR register.

3. *Subsystem components and custom blocks*

Clock changer	Converts 50MHz operational frequency to 1MHz channel frequency
FIFO transmitter	Temporary hold the data from the transmitter Parameters: Data width 16, Data Depth 128, Different write/read clk signals.
FIFO receiver	Temporary hold the data for outputting receiver Parameters: Data width 16, Data depth 128, Different write/read clk signals
LSFR	Pseudo random code for the noise generation

Verification

1. Source/Sink

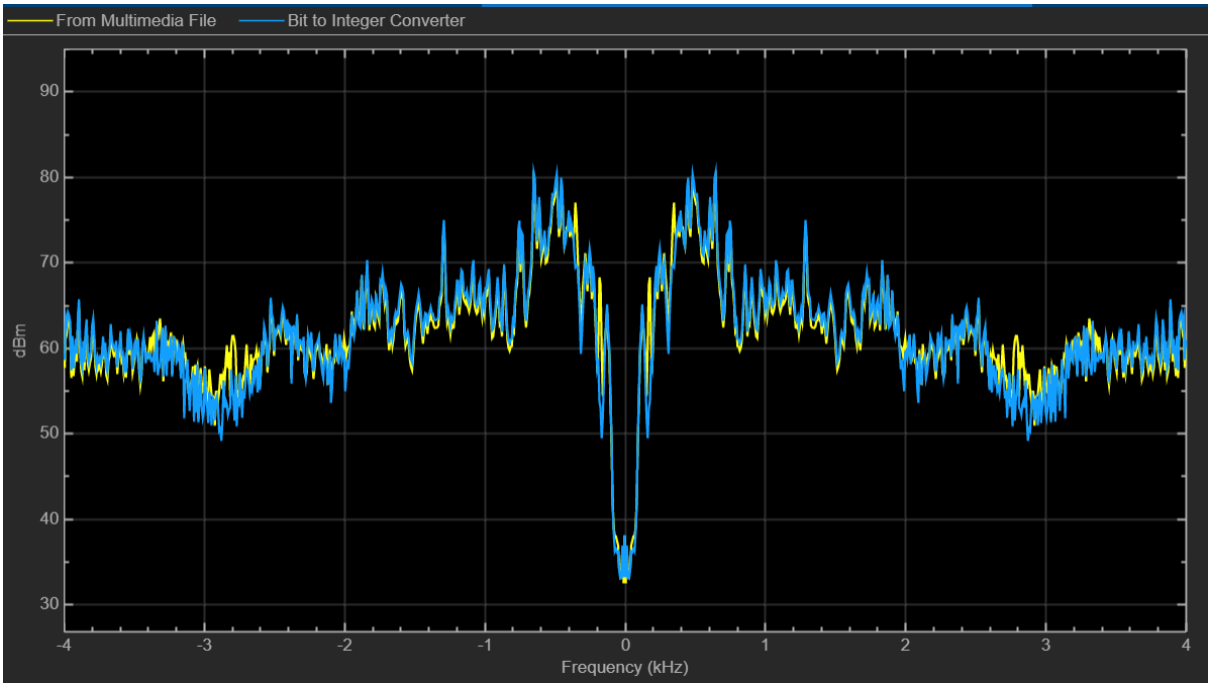


Figure 26: Frequency spectrum of the source (yellow) and sink (blue) with 100 dB SNR

In the figure above, the frequency spectrum of the source is represented in yellow and that of the sink is represented in blue. As expected, at a higher signal to noise ratio the waveforms are almost identical due to the relatively low amount of noise interfering on the signal.

FPGA verification:

For the FPGA verification, we reused our code for individual assignment part1 and tested it for audio input and output. After the test, we determined that the source and sink module is working.

2. A/D and D/A converters

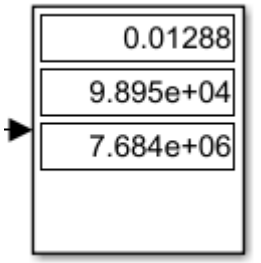


Figure 27: Converter BER at 9 dB

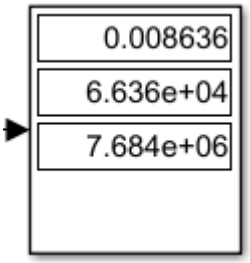


Figure 28: Converter BER at 100 dB

Table ADC/DAC8: Converter BER at 9 dB & 100 dB

As expected, there is a greater Bit Error Rate at a lower Signal to Noise ratio as more noise in a system introduces a greater amount of errors due to misinterpretation of signals.

FPGA verification

For the FPGA verification, we observe the simulation result of part2 and 3 of the individual assignment. We found that the altera codec is able to output integers which help us in implementing the AD/DA converters.

3. Error correction Encoder/Decoder

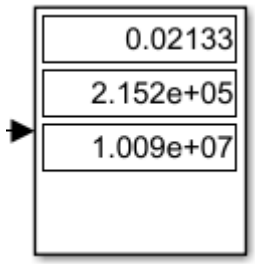


Figure 29: Encoder/Decoder BER at 9 dB

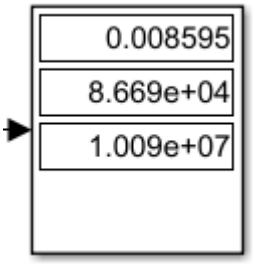
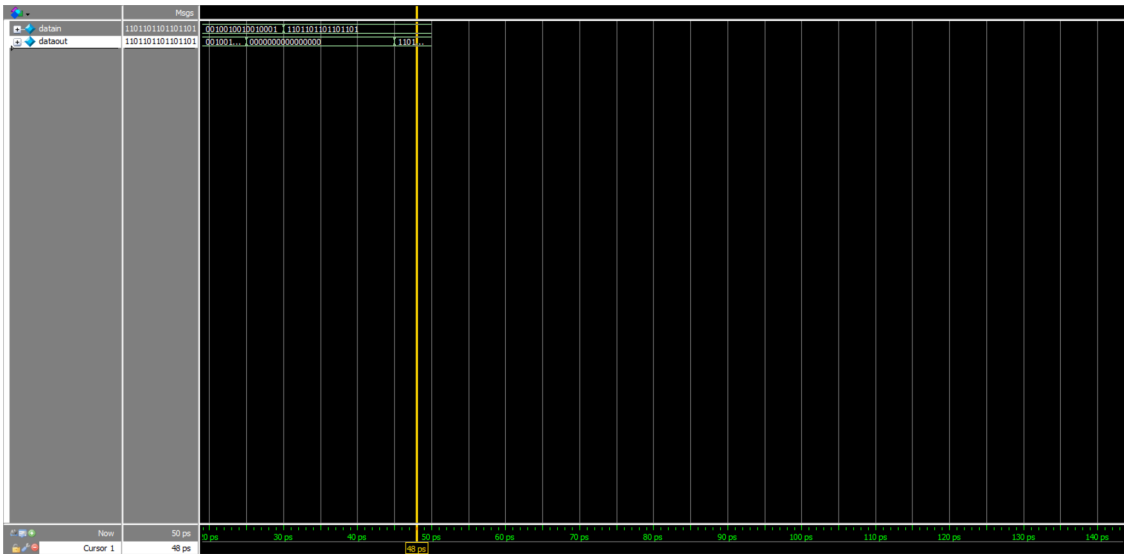


Figure 30: Encoder/Decoder BER at 100 dB

Table E/D9: Encoder/Decoder BER at 9 dB & 100 dB

With the error rate calculation based on the output of the encoder and input of the decoder as shown above, error in encoder/decoders is higher at 9 dB compared to 100 dB. This is expected as a lower Signal to Noise ratio will introduce more errors to the system that may not be corrected. One thing that is interesting to note is that the error for 100 dB is almost identical to that measured previously at the converters, and the error for 9 dB is almost double the error measured at the converters, possibly suggesting that A/D and D/A converters play a significant role in error mitigation when there is a large amount of noise, but not much of a role in error mitigation when there is less noise density.

FPGA Verification



We verified our design of the error encoding module with modelsim simulation. In the test bench, we loaded two different one

4. Modulator/Demodulator

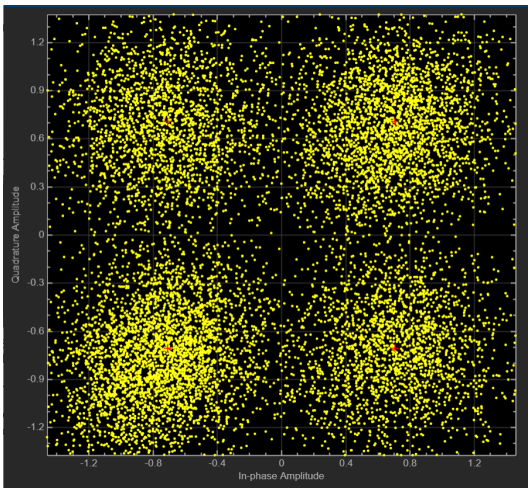


Figure 31: Constellation scatter plot at 9 dB

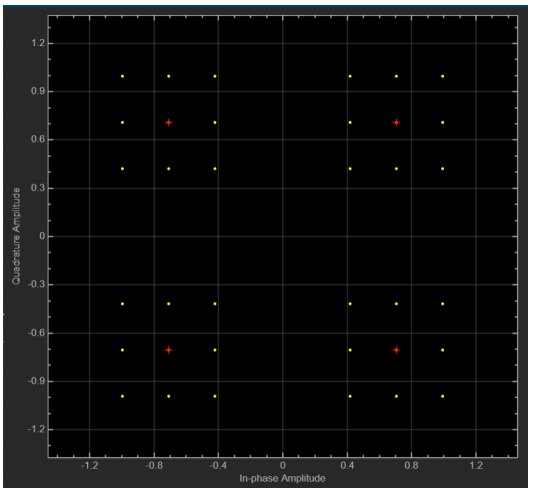
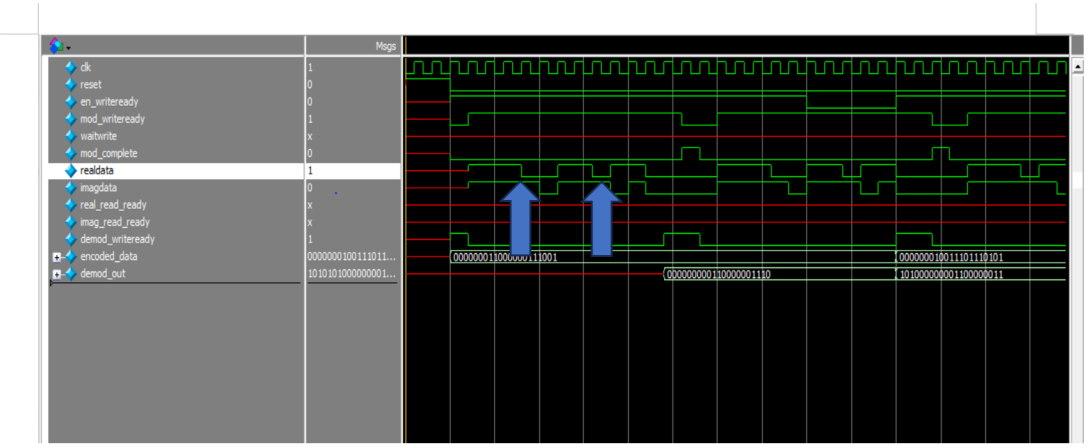


Figure 32: Constellation scatter plot at 100 dB

Table M/D10: Constellation scatter plot for Modulator/Demodulator at 9 dB and 100 dB

A large difference can be observed between the constellation scatter plots at 9 dB and 100 dB SNR as expected. It is clear that the bit error rate for 9 dB would be higher as there is much more of a chance of being assigned to the wrong symbol due to the amount of interference.

FPGA Verification



Our modulation block splits our data into real and imaginary values. The arrows above shows the output of the qpsk modulator. The input 21 bits (pad to 22 bits) is mapped to the 4 quadrants of a cartesian plane 2 bits at a time. The demodulator reverses this process.

Our receiver worked fine when we test-benched it by itself. However, it did not work in conjunction with other modules.The simulated result of our fpga implementation of the modulator and the demodulator is given above. The explanation for this result is given below the result. It is outputting the data in 50MHz operation frequency, which is then adjusted using 48kHz as a driven read clk in a fifo. The module also split the output data into real and imaginary parts, which is then outputted to the other module.

5. Transmitter/Receiver

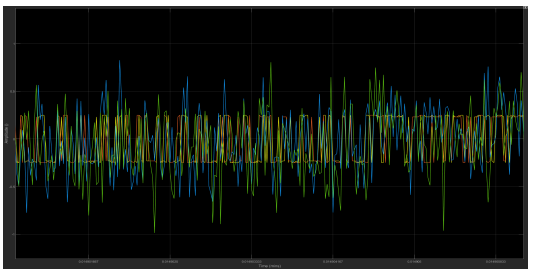


Figure 33: Transmitter/Receiver time domain signals at 9 dB

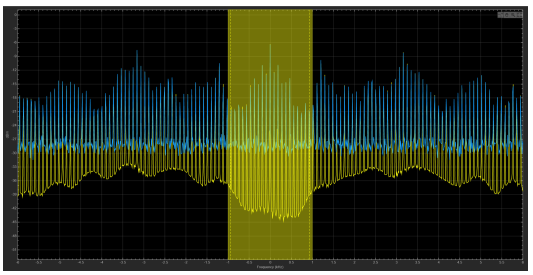


Figure 34: Transmitter/Receiver frequency domain signals at 9 dB

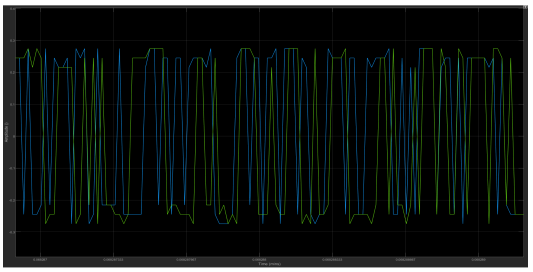


Figure 35: Transmitter/Receiver time

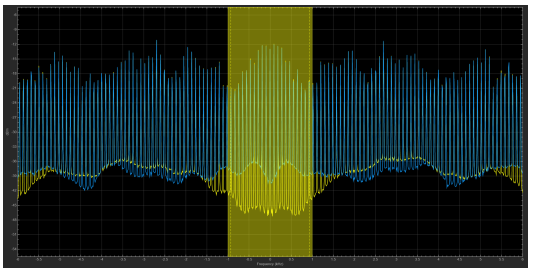


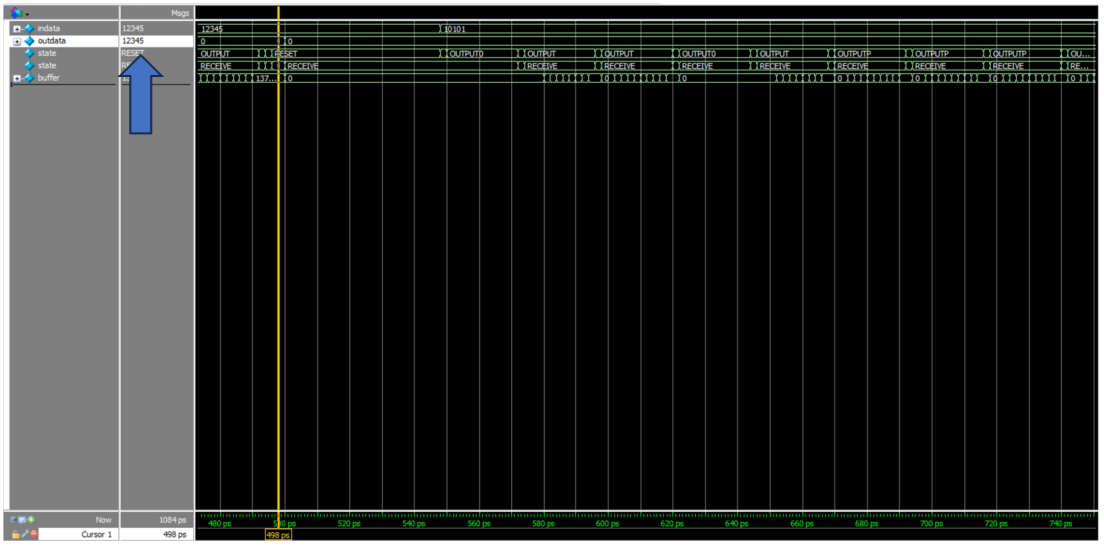
Figure 36: Transmitter/Receiver

domain signals at 100 dB	frequency domain signals at 100 dB

Table T/R11: Transmitter/Receiver signals in the time and frequency domains at 9 and 100 dB

FPGA verification

We are able to export the data coefficient of the raised cosine filter. The lookup table is then implemented into the fpga design.



The raised cosine filter test bench implemented was able to transmit the input data. The transmitter have 4 different modes of output, including two halves of the raised cosine filter, outputting bunch of zeros, as well as the convolution between two of them. The receiver reverses this process by mapping the incoming bits to the look up table.

Through modelsim simulation, we were able to verify that the receiver is correctly converting the symbol back to the original data.

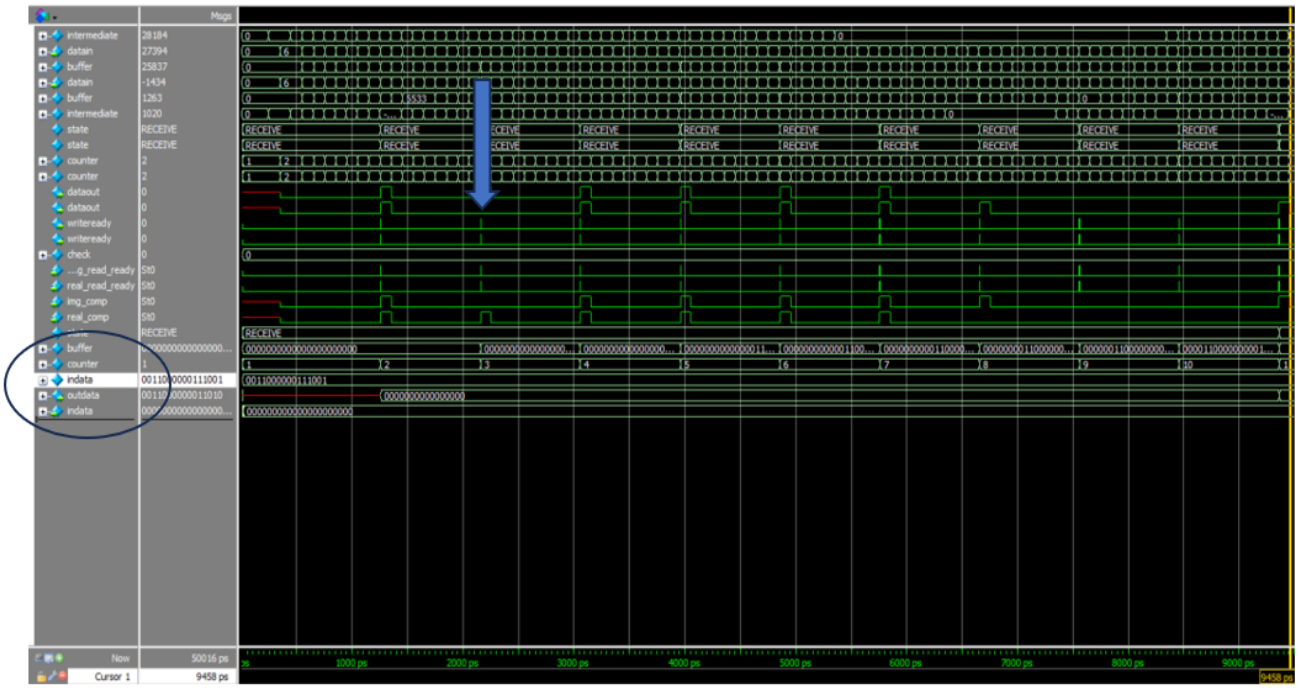
6. Channel

The variance of the AWGN channel can be calculated below:

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

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

FPGA verification



Here is the simulated result of the channel. It includes two fifo memory for adjusting input and output data rate of the module. As shown above, it is able to to turn a fast data input into a slow data input, which is required to meet the 1MHz channel frequency requirement. After an 8 bits long LFSR signal was added to the input of the transmitter, the output data was taken to the QPSK demodulator and Hamming decoder. From the circled part of this test bench result, we can see that the first eight bits of indata was transferred to outdata as they were not affected by the LFSR with SNR of 9dB.

Team Contributions

John Ye
Individual contributions:
Error encoder, Error decoder, Transmitter, Receiver [large amount of time spent], Channel(mainly in conjunction with John Kim[large amount of time spent]), QPSK(in conjunction with John Kim)

Team effectiveness:
Personally, I felt like our team was somewhat effective. We are able to solve some problems very quickly, but we had a lot of difficulties in the other problems. In addition, our team seems to run into an unexpected large amount of issues, which might be due to the somewhat lack of organization of our time and effort.

The team effectiveness was drastically changed after demo 1, as the FPGA implementation became an issue. The skill disparity of the hardware is a lot higher than the matlab simulink.

Other comments:
To ensure a smooth integration with the parts from the other members of the group, I used ready/enable handshake extensively when writing my codes. In the design process, I faced tons of challenges, such as unfamiliarity with the matlab simulink as well as the communication process. I overcame these challenges through spending much time in researching and testing.
I learned that I should make detailed plans before writing any codes to ensure the usability, as well as being better at managing my time and effort spent on parts of the project. I spent much time working on stuff that is very hard to implement on the FPGA, which delayed the writing process of the other modules.

Unfortunately, our team dynamics are not ideal. We didn't set a definite deadline for completing all the modules, which delayed our system integration as some modules were completed later than expected.

Mohkam Malhi:

Tasks I worked on personally include doing online-based research to have a better understanding of the variables we would be working with on this project, adjusting parameters in Simulink blocks to try to find the best working solution to our project targets, and implementing a LFSR to generate random noise to add to our signal passing through the channel.

Our team worked together to assemble the blocks in our Simulink diagram and write the final product document.

I felt our team had a very good dynamic. Although my leadership role was for transmitter/receiver and channel, as I had not taken CPEN 311 and had taken CPEN 211

almost three years ago, it became clear when it came to hardware implementation that my teammates had a greater proficiency in Verilog than myself, and did not hesitate to effectively change the lead for transmitter/receiver and channel to an entire team section at that point. Although I did not end up writing much usable code apart from the LFSR number randomizer for noise, I still helped out by providing my perspective and opinions on how we could possibly implement different parts of our design during team meetings. As I hoped to help my teammates as much as I could I also prepared the outlines and rough drafts for documentation.

In this process I learned that in a team there will likely be people who are proficient at different tasks and the way to get the most production out of a team is to identify individuals' skills and assign tasks accordingly. I feel scrum meetings were also good as it was an opportunity for everyone to come together and reflect on what everyone in the team has been doing, set deadlines, and support each other towards a common goal.

John (Yohan) Kim

Individual contributions: Source/sink, ADC/DAC, Channel (conjunction with John Ye, large amount of time), Complex modulator and demodulator (conjunction with John Ye)

Team effectiveness: The effectiveness of our team was good. One issue we had was a clock issue. We had a lot of timing differences in our system because many modules in our FPGA system required different clocks. It was different from the simulation in Simulink because Simulink modules were built-in. They were carefully developed by many engineers and scientists. However, many times, in our design, we actually had to design them from scratch and had to consider timing including how many cycles are taken in one module to update output data and be transferred to the next module. The handshake module was very helpful because it ensured that the data was updated successfully before sending it to the next module.

Other comments:

What did you learn: In terms of technical lessons, I learned a lot about clock domain crossing and fifo. It was very educational. Also, I learned in depth about how to implement convolution and computation-heavy modules in FPGA and how effective it can be.

Anything else you spent your time on: I was very happy to brush up my FPGA skills 2 and half years after I learned it last time. To achieve this, I read and watched many Verilog and SystemVerilog tutorial videos.

References

[1] C. Grecu, P. Lusina, "ELEC391 Project Description," 2023.

[2] S. D. Potey and P. M. Dhande, "Error Detection and Correction Capability for BCH Encoder using VHDL," 2019 IEEE 5th International Conference for Convergence in Technology (I2CT), Bombay, India, 2019, pp. 1-4, doi: 10.1109/I2CT45611.2019.9033847.

[3] P. Rajasekhar and V. Rama, "BER performance of 1024 (BPSK, QPSK and QAM) using OFDM over Rayleigh, Rician and Nakagami Fading Channel," *International Journal of Computer Applications*, vol. 164, no. 7, pp. 14–17, 2017. doi:10.5120/ijca2017913515

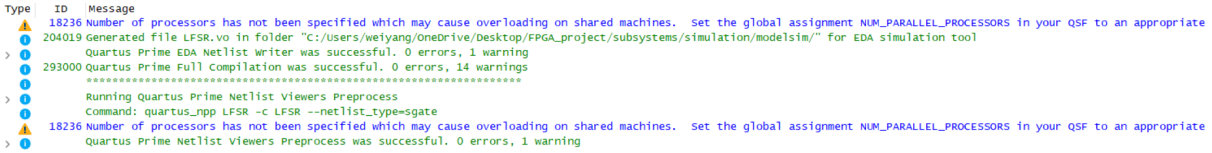
[4] L. Pilato, G. Meoni and L. Fanucci, "Design and quantization limits of root raised cosine digital filter," 2017 3rd International Conference on Frontiers of Signal Processing (ICFSP), Paris, France, 2017, pp. 59-62, doi: 10.1109/ICFSP.2017.8097061.

[5] C. Langton, "Inter Symbol Interference (ISI) and raised cosine filtering", 2002. [Online]. Available: http://complextoreal.com/tutorials/tutorial-14-inter-symbol-interference-isi-and-raised-cosine-filtering/

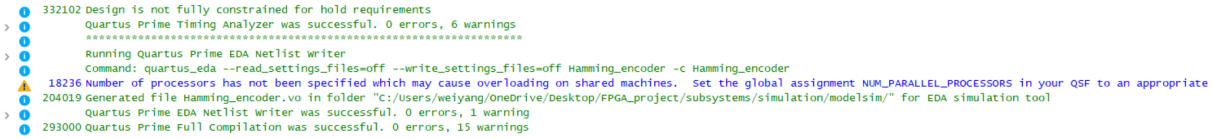
Appendix A: Deliverables

Deliverable	**File Name**	**Notes (optional)**
Product document	Team4_391_Report	The final report of our design
Product presentation	ELEC_391_Project Presentation_Demo_2	The presentation slides
Simulink system file	Team4_Project_Simulink_UpdatedGilbert	The final simulink design with consideration of FPGA compatibility
HDL source code, including testbenches and *readme* files	Altera_ … files Hamming_encoder Hamming_decoder QPSK_Complex QPSK_Complex_Demod Raised_transmitter Raised_receiver Channel_Connection System Fifo ….	The source files for our FPGA design

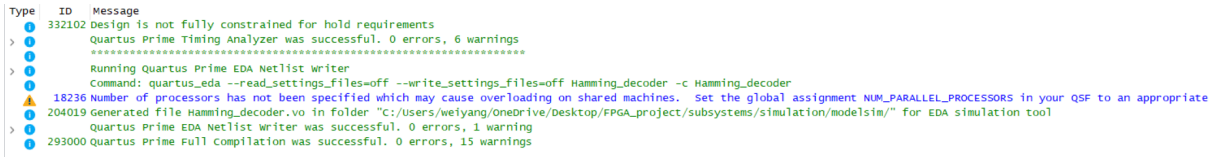
Appendix B: Quartus verification



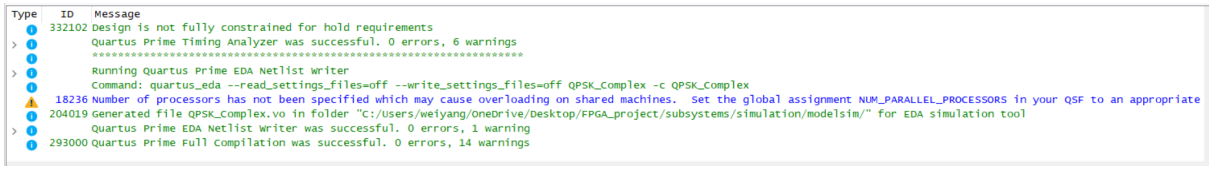
Quartus synthesization of the LFSR module



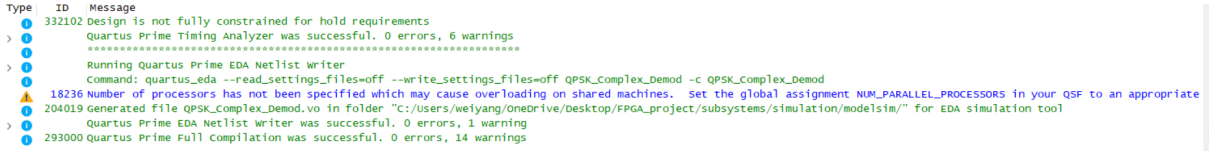
Quartus synthesization of the hamming module



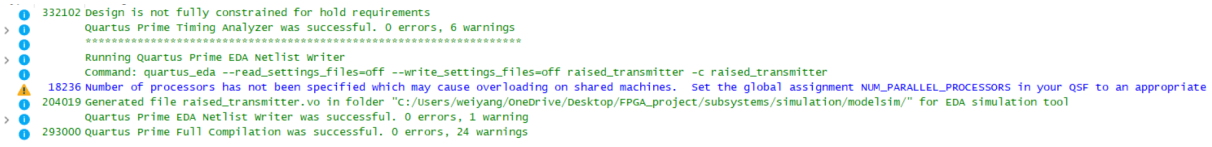
Quartus synthesization of the hamming decoder module



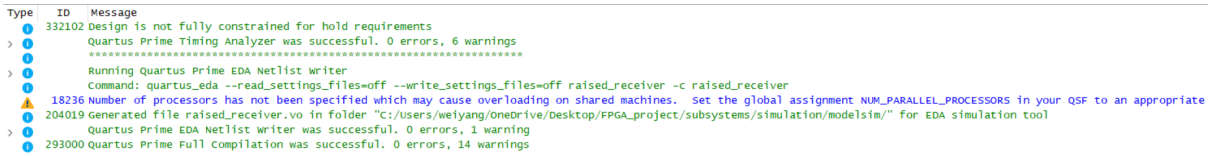
Quartus synthesization of QPSK modulator



Quartus synthesization of QPSK demodulator



Quartus synthesization of raised cosine filter transmitter



Quartus synthesization of raised cosine filter receiver