End-to-End Digital Communication Pipeline in Verilog

Nabil KANA

Overview:

This project implements a complete digital communication system pipeline using Verilog for hardware-level components and MATLAB for channel simulation. It simulates a realistic transmission system that includes encoding, modulation, noise, and decoding stages. The system successfully recovers the original bitstream, validating correctness and robustness.

Pipeline Components:

1. LFSR-Based Bit Generator

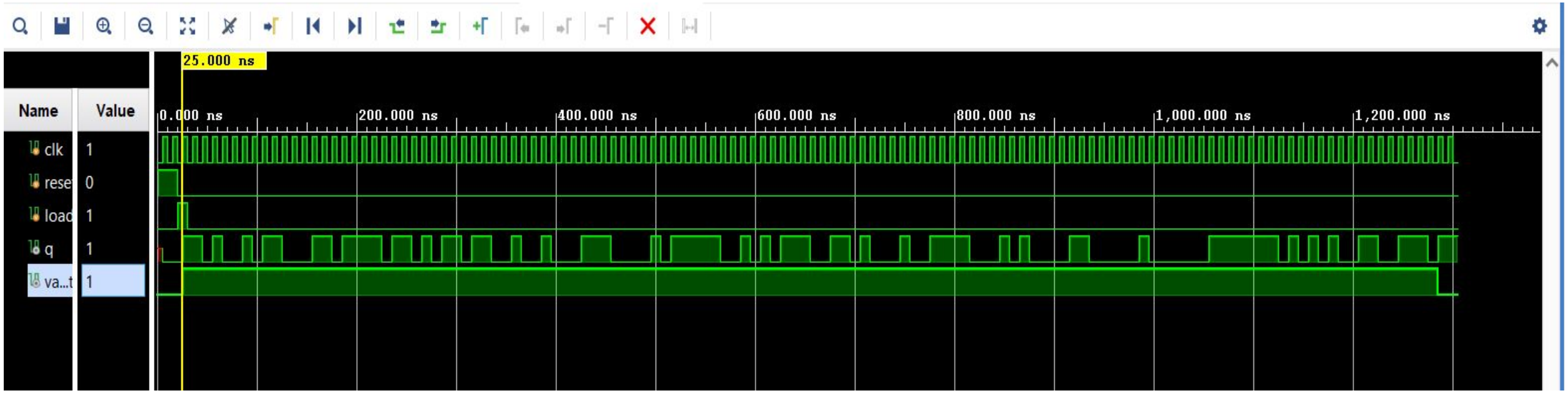
To generate the random input data that will be transmitted through the communication pipeline, we implement a Linear Feedback Shift Register (LFSR). An LFSR is essentially a shift register consisting of n D-type flip-flops, with specific taps feeding back through XOR gates to the least significant bit (LSB) of the register.

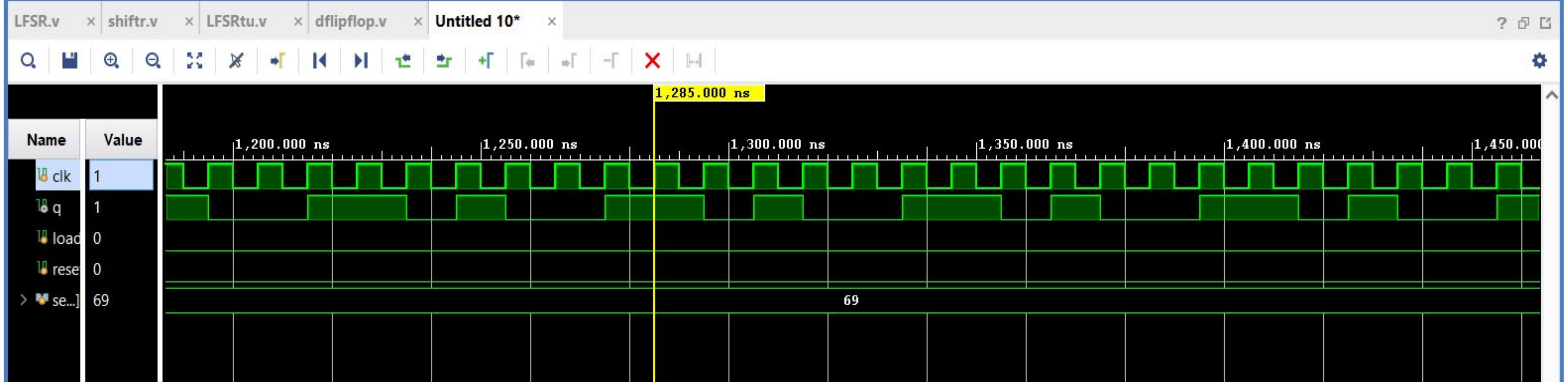
- *We begin by creating a basic D flip-flop module (`dflipflop.v`) using behavioral Verilog.*
- *Next, we design a modular shift register (`shiftr.v`) that can be configured with any number of flip-flops.*
- *The LFSR is built by connecting the shift register with appropriate feedback taps.*

For this project, we used an LFSR of length n = 7, which, for a valid seed, generates a maximum-length sequence of:

2n−1=127 pseudo-random 2^n - 1 = 127 pseudo-random bits

Below are screenshots from the simulation waveform, showing the LFSR output over time. The LFSR starts generating bits at 25 ns and completes a full cycle at 1285 ns, at which point it repeats the same sequence:
(1285−25)T+1=127 bits





2. Convolutional Encoder (Rate 1/3)

To improve resilience against noise and distortions in the transmission channel, we implement a convolutional encoder. This type of encoder introduces redundancy by generating multiple output bits for each input bit, allowing the receiver to detect and correct errors based on known encoding rules.

Our encoder doubles the bitstream length using a fixed polynomial logic. With knowledge of the encoding polynomials and the initial state of the internal shift register, the receiver can later reconstruct the original data—even if some bits are flipped during transmission.

In our design, the encoder outputs two bits for every input bit. For example, when we input a `1`, both output bits are `1`, as expected. When we then input a `0`, the second output returns to `0` due to the updated register state, while the first output remains high. This behavior confirms the encoder is correctly tracking the input bit history across its internal state.

A simulation screenshot below shows this step-by-step response and validates the correctness of the logic.

3.Serializer/interleaver

While the convolutional encoder improves the system’s ability to recover from isolated bit errors, it is not effective against burst errors, where multiple adjacent bits are corrupted. This is because the information for each bit is tightly packed into its neighboring bits, making it vulnerable when consecutive bits are lost or altered.

To address this, we introduce an interleaver module. The interleaver operates on the serialized output of the convolutional encoder, which produces two bits (`v1` and `v2`) for every input bit. These are first flattened into a serial stream by the serializer, then grouped into blocks of 128 bits.

The interleaver stores this block in a matrix and then outputs the data column-wise in sets of 4 bits, effectively spreading out the influence of each original bit across the transmitted stream. This ensures that if a burst of data gets corrupted, the corresponding redundancy needed for recovery will likely remain intact in a different part of the stream.

To support this data flow:

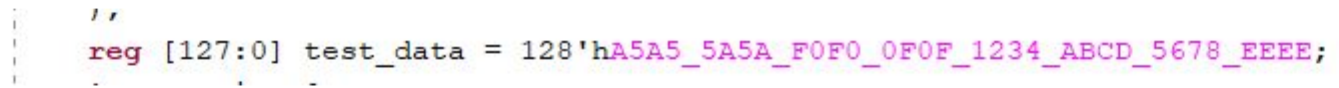
- The serializer and interleaver run at twice the clock speed of the encoder, as they must handle two bits for every encoded input.
- While using different clock domains is not ideal in production systems, for a compact and controlled design like ours, this approach helps keep the pipeline moving efficiently.

We analyze specific output patterns to confirm the expected data rearrangement:

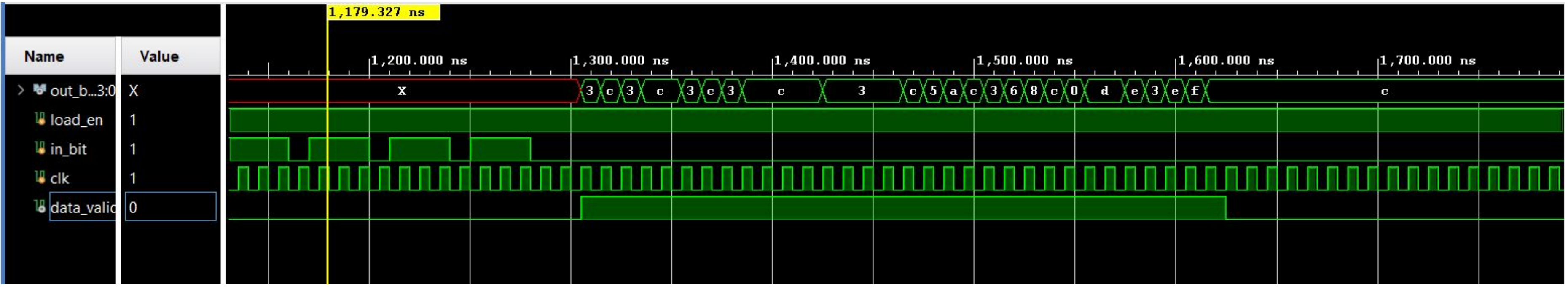
- The 8th, 16th, 24th, and 32nd bits (from the original serial input), when interpreted from MSB to LSB, form the binary sequence `1100`. This translates to the hexadecimal value 3, and it matches the first output value of the interleaver, as we begin reading from the last 4 columns of the matrix.

- Similarly, the 7th, 15th, 23rd, and 31st bits form the pattern `0011`, which corresponds to the hexadecimal value C.

- Another consistency check: the 1st, 9th, 17th, and 25th bits also yield `1100` (MSB to LSB), which again translates to 3, and aligns with the 8th output value of the interleaver.



```
reg [127:0] test_data = 128'hA5A5_5A5A_F0F0_0F0F_1234_ABCD_5678_EEEE;
```



4.Modulation

Our pipeline uses 16-QAM modulation fully implemented in Verilog, where every 4 input bits are mapped to one of 16 constellation points defined by in-phase (I) and quadrature (Q) components.
After modulation, the complex symbols are passed to MATLAB, where noise is added to simulate channel impairments at various Signal-to-Noise Ratios (SNRs). Additionally, MATLAB simulates burst errors to test the system’s resilience to consecutive bit corruptions. The noisy signals are then demodulated in MATLAB using hard-decision logic, converting the received symbols back into bit sequences.

The demodulated bits are returned to Verilog for deinterleaving and Viterbi decoding, completing the end-to-end transmission chain. This combined approach enables flexible performance evaluation under different noise and error conditions.

FULL transmitter Pipeline simulation :

On the transmitter side, all modules—including the LFSR, convolutional encoder, serializer, interleaver, and 16-QAM modulator—are integrated into a single top module for simulation and testing.

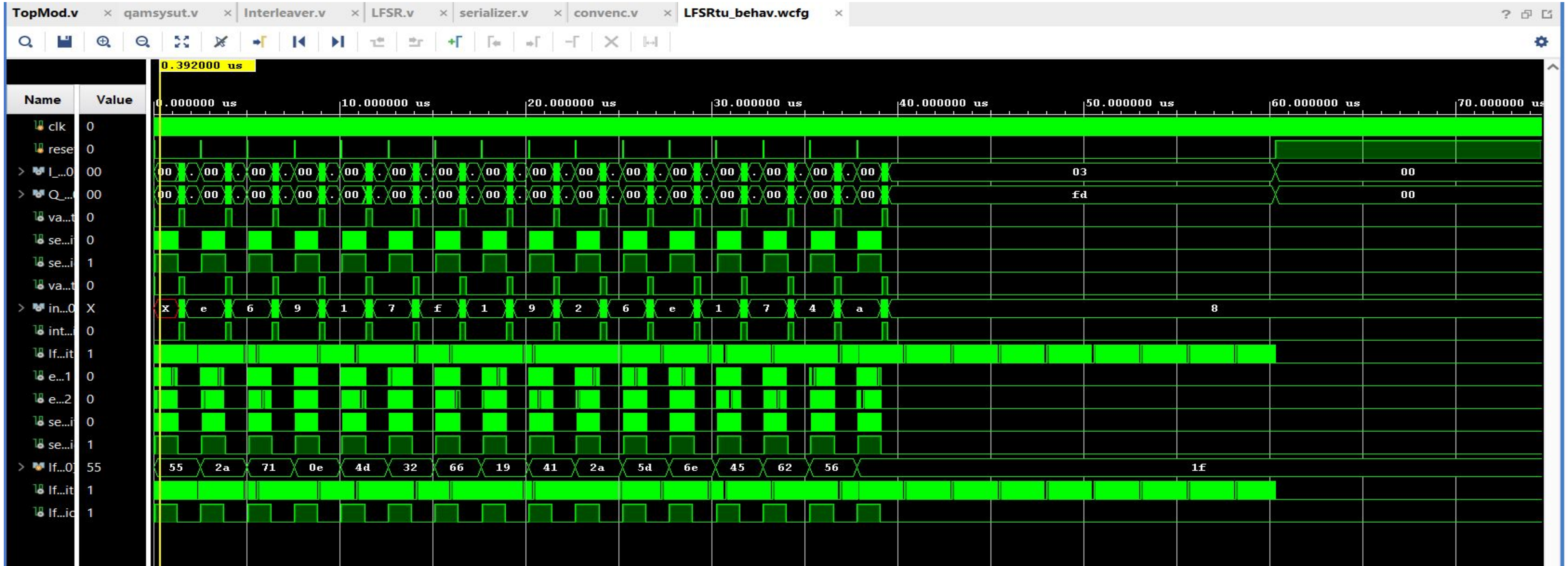
The data generation starts with the LFSR module, which outputs 62 pseudo-random bits per cycle (stopped with a counter) the cycle is repeated 16 times each time with a different seed so we end up with 992 meaningful bits . each 62 bits are then fed into the convolutional encoder, which produces 128 bits per batch (`62 × 2 + 4`), including 4 additional flushing bits to reset the encoder’s internal state.

This 128-bit stream flows through the full transmission pipeline:

Convolutional Encoder → Serializer → Interleaver → 16-QAM Modulator (all implemented in Verilog).

The resulting I/Q values are then sent to MATLAB for channel simulation and noise injection.

The testbench is designed to reset and rerun this process 16 times, resulting in a total of 16 × 62 = 992 original bits transmitted. Each time the LFSR asserts its `valid` signal, the generated bit is captured and stored in a text file, building the ground truth dataset. This file is later used for bit error rate (BER) evaluation by comparing the transmitted bits with the final decoded output on the receiver side.



5.deinterleaver

Once the interleaver is built and verified, implementing the deinterleaver becomes relatively straightforward. The core idea is to reverse the logic applied during interleaving.

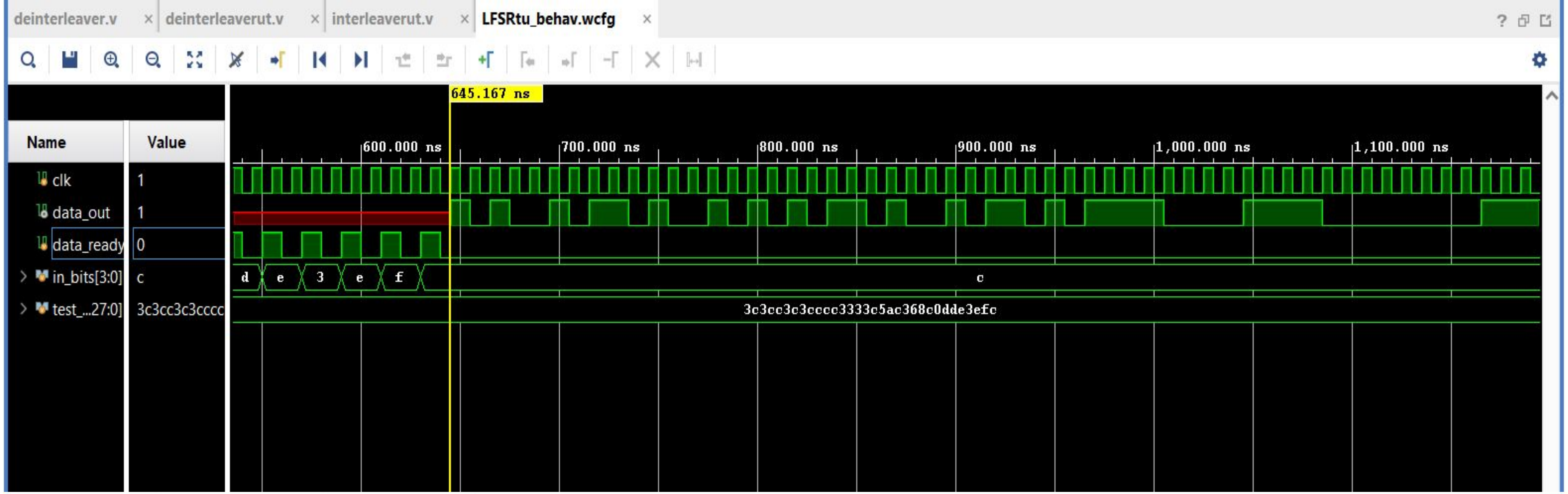
We start by taking sets of 4 bits from the demodulator output and insert them into a matrix. The storage begins from the last column, filling in the first four rows with each bit placed top-down (least significant bit at the top). We continue inserting data in 4-bit vertical chunks, moving column by column from right to left, then filling the next set of rows (e.g., rows 5–8), until the entire matrix is filled.

Once the matrix is populated, we read it out serially row-wise, which reconstructs the original pre-interleaved test sequence.

To verify correctness, we tested the deinterleaver by feeding it the output of the already verified interleaver module. The result:

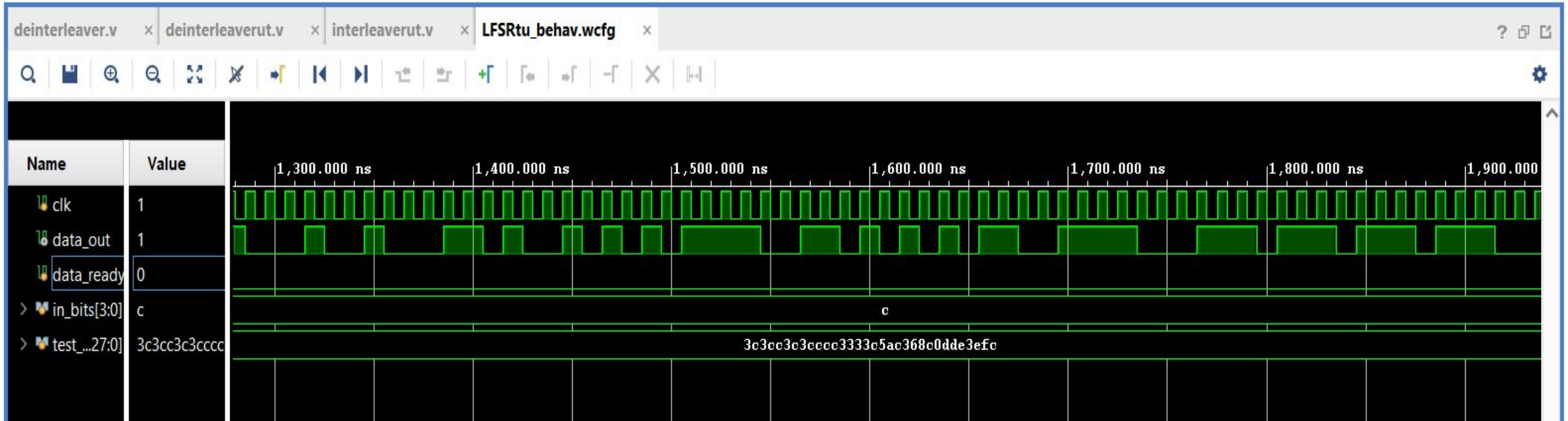
- The deinterleaver output begins with `1010 0101 1010 0101`, which corresponds to A5A5, matching the start of our test vector.

- The final output is `1000 1110 1110 1110 1110`, which corresponds to 8EEEE, the expected end pattern of our test sequence.



Name	Value
clk	1
data_out	1
data_ready	0
in_bits[3:0]	c
test_...27:0]	3c3cc3c3cccc

3c3cc3c3cccc3333c5ac368c0dde3efc

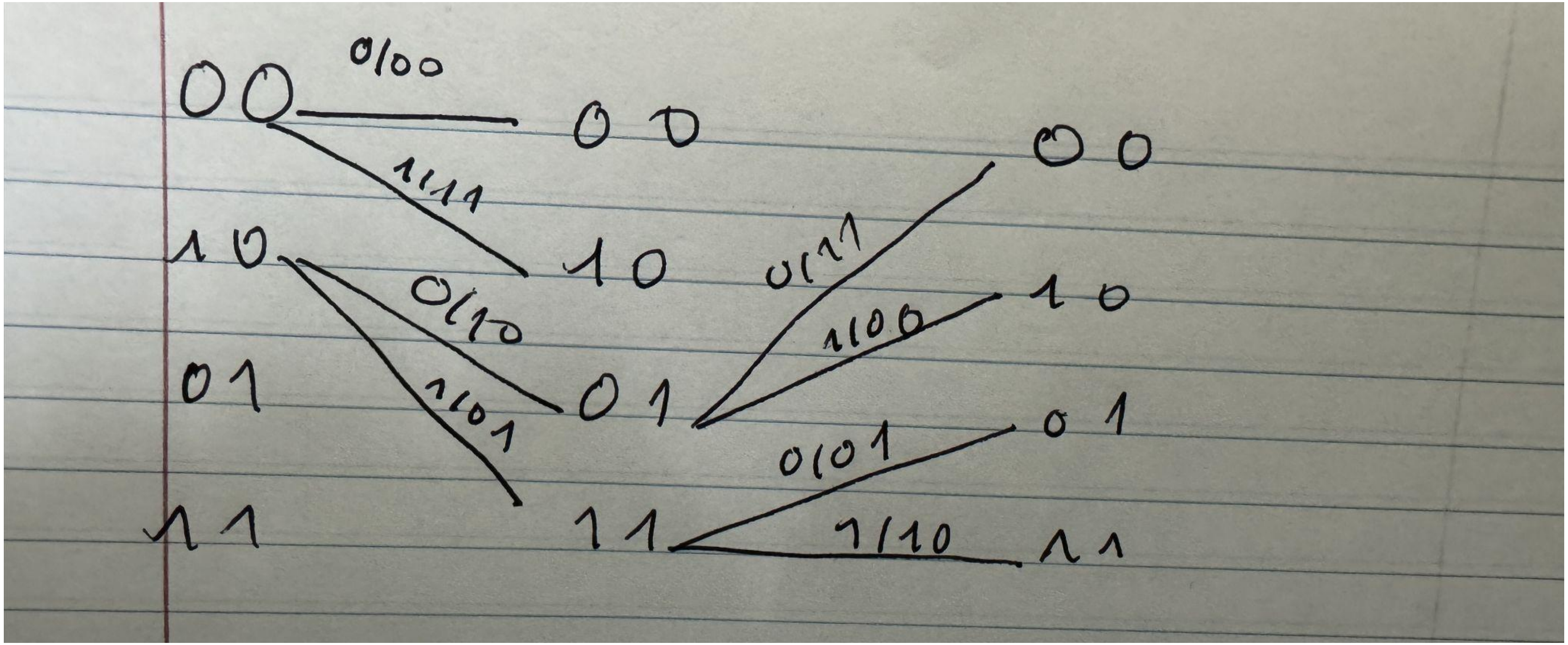


Name	Value
clk	1
data_out	1
data_ready	0
in_bits[3:0]	c
test_...27:0]	3c3cc3c3cccc

3c3cc3c3cccc3333c5ac368c0dde3efc

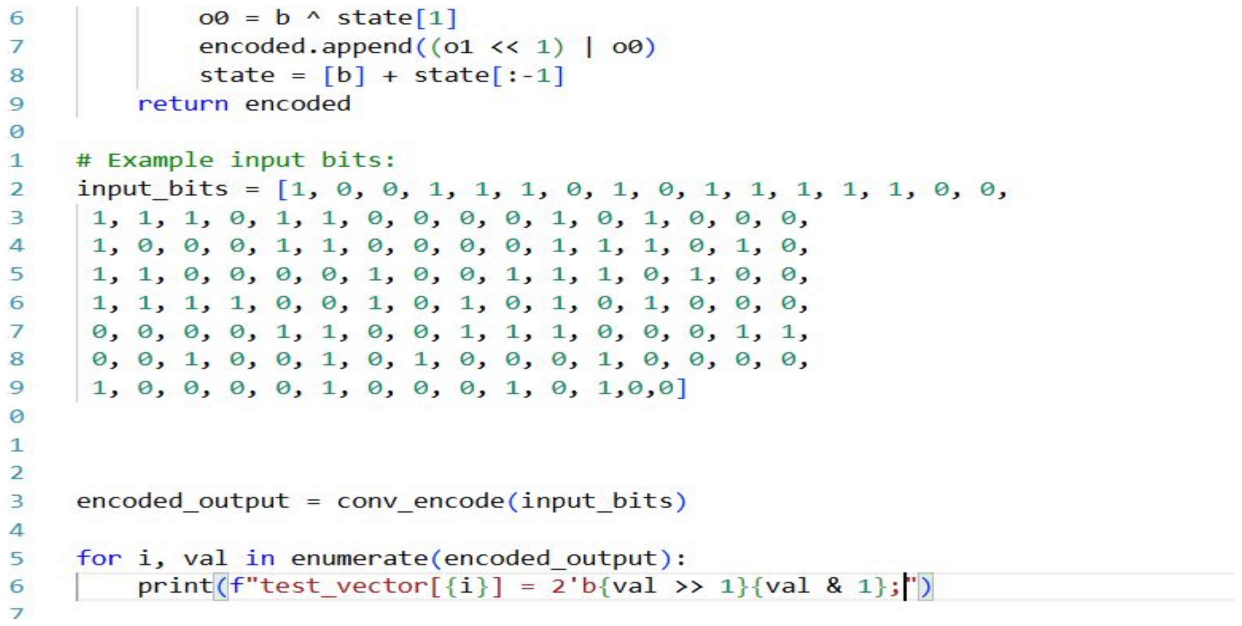
6.Viterbi Decoder

The Viterbi decoder works by tracing the most likely sequence of input bits through a trellis diagram, which maps all possible state transitions defined by the convolutional encoding scheme (shown below). It assumes a known initial state—usually the last two bits of the encoder’s shift register—and, as encoded data arrives, it calculates the Hamming distance between the received bits and the expected outputs for each transition. These distances are used to compute path metrics, which help the decoder decide which paths through the trellis are most likely. For each step, the decoder stores the most probable path to every state—known as the survivor path. After the full block is received, it performs a traceback through these stored paths to recover the original bit sequence. The hardware implementation uses finite state machines for state transitions and includes modules for metric computation, survivor memory, and traceback logic.

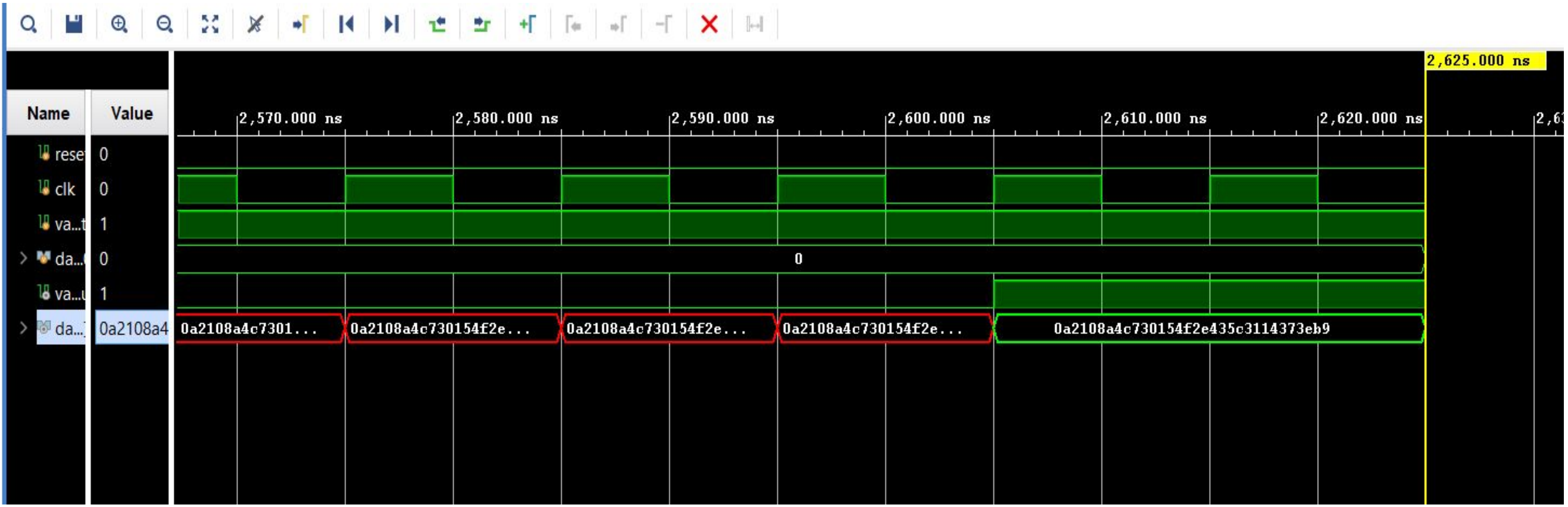


Trellis diagram of the (2,1,3) Viterbi decoder

To test our module, we use a Python script that encodes a set of input bits and generates the corresponding encoded bit pairs. These pairs are then fed as inputs to our Verilog module, enabling verification of correct encoding and decoding functionality through simulation. When examining the output data, the first burst of bits is `1001 1101 0111 1100`, which—when read from left to right—corresponds to the hexadecimal value 9BE3. This value matches exactly the leftmost bits (least significant bits) of our module's output, confirming consistency with the test input. At the end of the output, the last 6 bits are `0101 00`, which correspond to 0A in hexadecimal once padded. Since our data block contains only 62 bits, Vivado automatically pads it with 2 extra bits to form a full byte, ensuring the output is byte-aligned and readable.

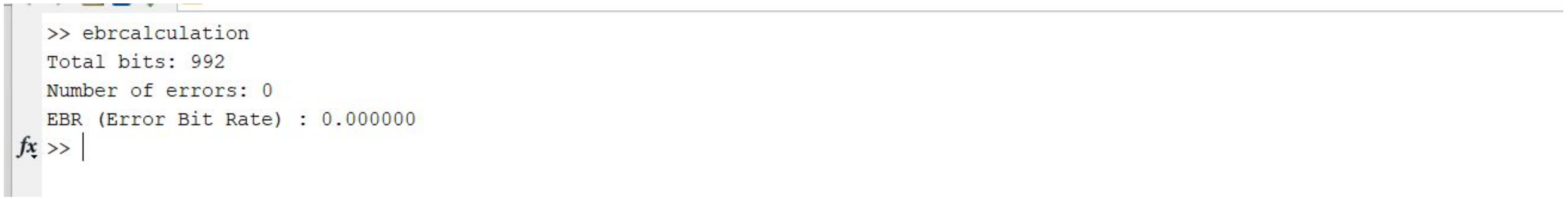


```python
        o0 = b ^ state[1]
        encoded.append((o1 << 1) | o0)
        state = [b] + state[:-1]
    return encoded

# Example input bits:
input_bits = [1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0,
 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,
 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0,
 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0,
 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0,
 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1,
 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1,0,0]



encoded_output = conv_encode(input_bits)

for i, val in enumerate(encoded_output):
    print(f"test_vector[{i}] = 2'b{val >> 1}{val & 1};")
```



FULL receiver Pipeline simulation :

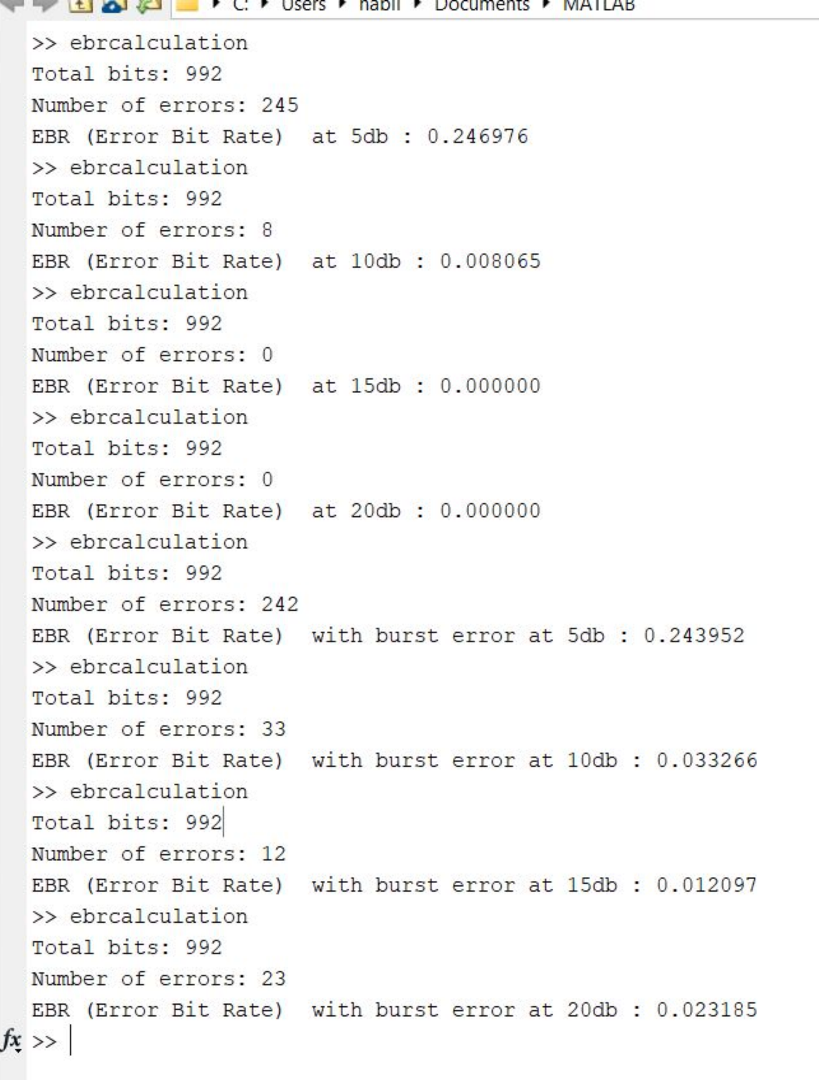
Next, the demodulated bits generated by MATLAB are fed into a top-level Verilog module containing both the deinterleaver and Viterbi decoder. The testbench runs this module and captures the decoded output bits, writing them into a text file then export this file and compare it with the original bits file in matlab to make sure our pipeline works accordingly



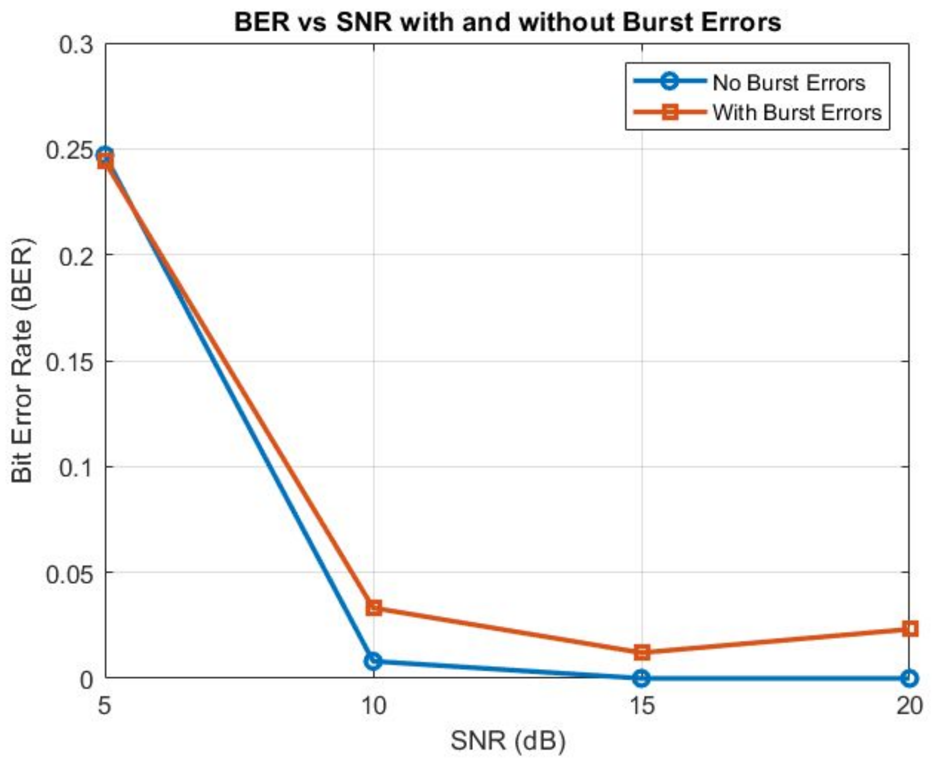
```
>> ebrcalculation
Total bits: 992
Number of errors: 0
EBR (Error Bit Rate) : 0.000000
>> 
```

After successfully decoding 992 randomly generated bits with zero errors, we confirmed that the full pipeline—from encoding to decoding—is functioning correctly under ideal (noise-free) conditions. With this baseline established, we proceeded to evaluate the system's robustness by introducing different levels of noise in MATLAB. For each specified Signal-to-Noise Ratio (SNR), we pass the noisy modulated data through the receiver pipeline, capture the decoded output, and calculate the Error Bit Rate (EBR). This allows us to assess how well the system performs under increasingly challenging channel condition

The results show that our pipeline reliably handles channel noise at 10 dB SNR and above, achieving near-perfect or perfect recovery even under moderate noise. When burst errors are introduced, the system remains resilient—maintaining a low EBR at 10 dB and above—thanks to the combination of convolutional coding and interleaving.However, in more extreme channels with dense or frequent burst errors, more advanced techniques such as deeper interleaving, Reed–Solomon codes, or adaptive decoding could be introduced to further improve reliability and ensure error correction under harsher conditions



```
>> ebrcalculation
Total bits: 992
Number of errors: 245
EBR (Error Bit Rate)  at 5db : 0.246976
>> ebrcalculation
Total bits: 992
Number of errors: 8
EBR (Error Bit Rate)  at 10db : 0.008065
>> ebrcalculation
Total bits: 992
Number of errors: 0
EBR (Error Bit Rate)  at 15db : 0.000000
>> ebrcalculation
Total bits: 992
Number of errors: 0
EBR (Error Bit Rate)  at 20db : 0.000000
>> ebrcalculation
Total bits: 992
Number of errors: 242
EBR (Error Bit Rate)  with burst error at 5db : 0.243952
>> ebrcalculation
Total bits: 992
Number of errors: 33
EBR (Error Bit Rate)  with burst error at 10db : 0.033266
>> ebrcalculation
Total bits: 992
Number of errors: 12
EBR (Error Bit Rate)  with burst error at 15db : 0.012097
>> ebrcalculation
Total bits: 992
Number of errors: 23
EBR (Error Bit Rate)  with burst error at 20db : 0.023185
fx >>
```

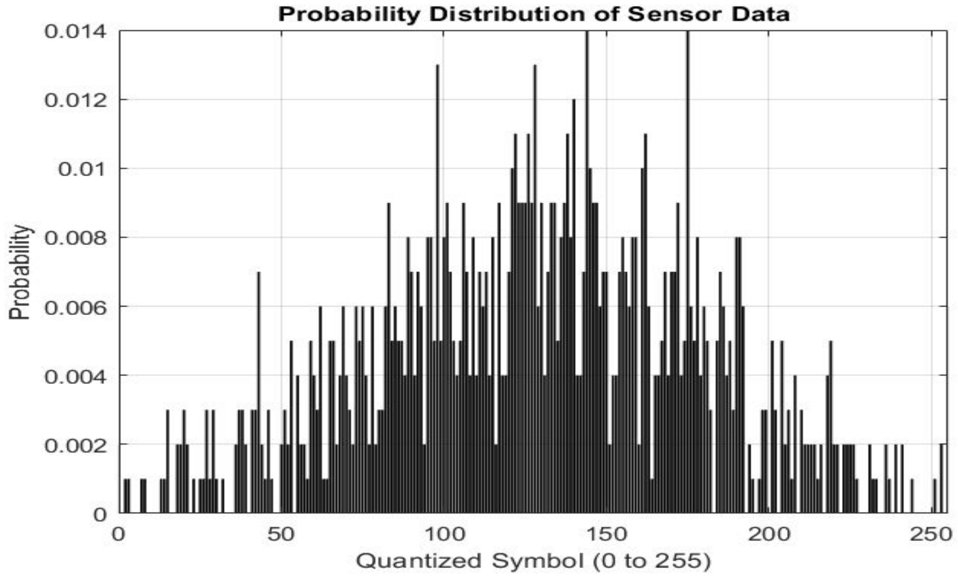


Extended MATLAB Simulation for System Validation and Enhancement

While the Verilog implementation covered the core baseband chain—including encoding, interleaving, modulation, demodulation, and Viterbi decoding—some additional processing stages, such as line coding, matched filtering, and advanced channel modeling, were explored separately in MATLAB. These components are all implementable in hardware and could be integrated into the existing RTL design in a future development phase. The MATLAB simulation not only enabled rapid experimentation with these features but also highlighted the critical role of error correction coding—particularly Viterbi decoding, which was already implemented in the RTL phase—in maintaining reliable communication at low SNR levels. The following section presents this extended MATLAB-based analysis and its implications for the full system design.

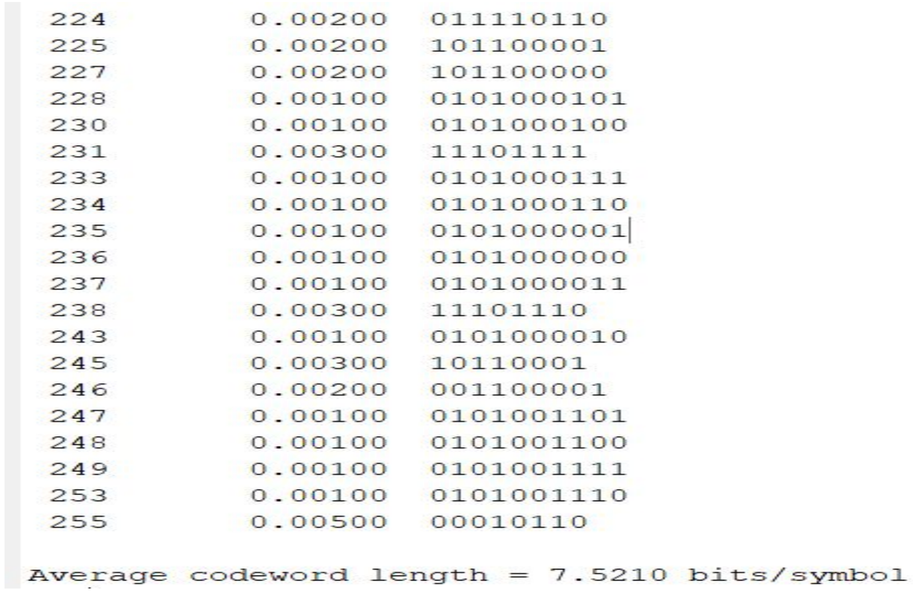
1.Data Generation

We start by modeling temperature sensor data with a Gaussian distribution ranging from 20 to 30 degrees Celsius. This distribution is stationary, as its statistical properties remain constant over time, and it is ergodic with a sufficiently large number of samples — meaning time-averaged properties converge to ensemble averages. The data is then quantized into 256 discrete levels (8 bits per symbol). The computed entropy is 7.4340 bits/symbol, which implies that, in the best-case scenario, 8,000 bits of raw data could be compressed to approximately 7,434 bits.



2.Information source coding

In this phase, Huffman coding was applied to compress the source data. The average codeword length was found to be approximately 7.5210 bits per symbol, which is very close to the source entropy. The coding efficiency, calculated as the ratio of entropy to average codeword length, is approximately 98.8%, indicating that the Huffman coding approach compresses the source data very effectively.



```
224          0.00200   011110110
225          0.00200   101100001
227          0.00200   101100000
228          0.00100   0101000101
230          0.00100   0101000100
231          0.00300   11101111
233          0.00100   0101000111
234          0.00100   0101000110
235          0.00100   0101000001
236          0.00100   0101000000
237          0.00100   0101000011
238          0.00300   11101110
243          0.00100   0101000010
245          0.00300   10110001
246          0.00200   001100001
247          0.00100   0101001101
248          0.00100   0101001100
249          0.00100   0101001111
253          0.00100   0101001110
255          0.00500   00010110

Average codeword length = 7.5210 bits/symbol
```

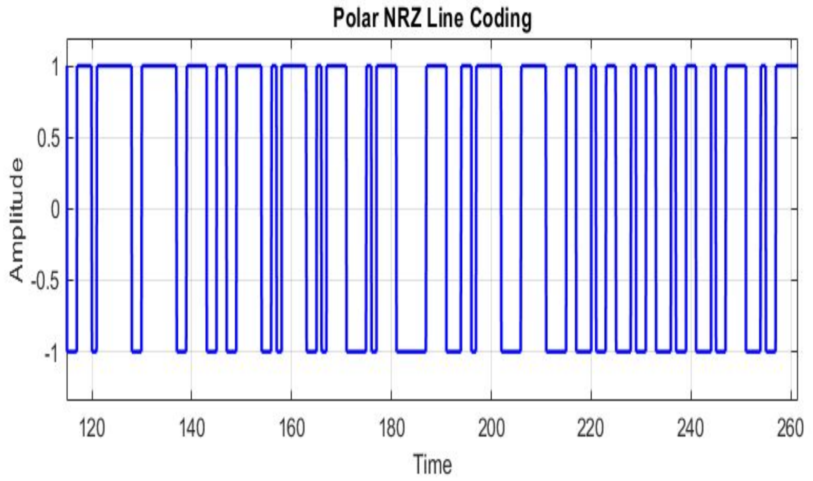
-Huffman Coding Results-

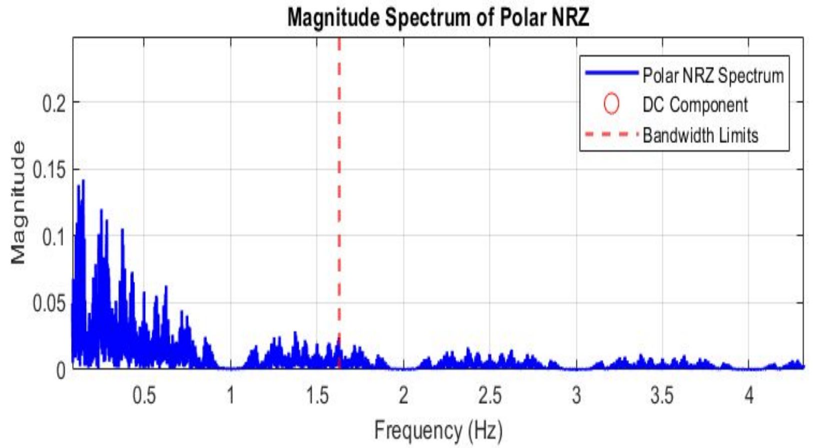
3.Line Coding

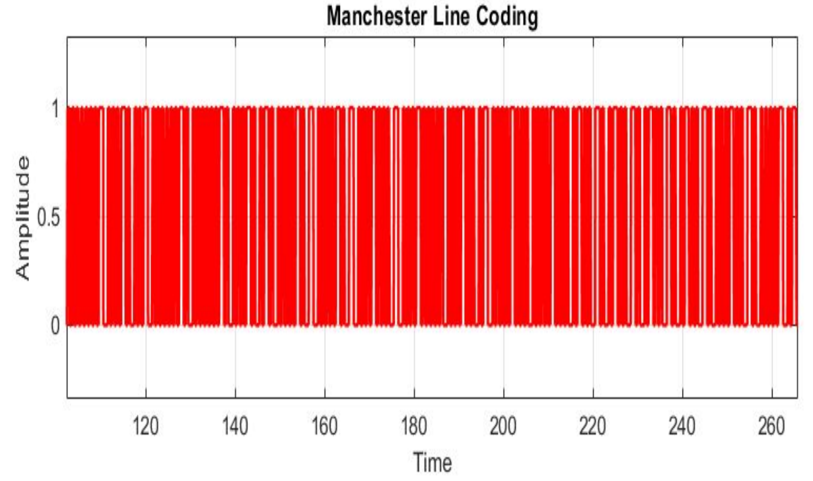
Polar NRZ (Non-Return-to-Zero) line coding represents bits by two voltage levels: a logical ‘1’ is mapped to a positive voltage , and a logical ‘0’ to a negative voltage The signal level remains constant throughout the bit duration, which means fewer transitions but causes a significant DC component in the spectrum. This DC level can lead to baseline drift and makes clock recovery more challenging.

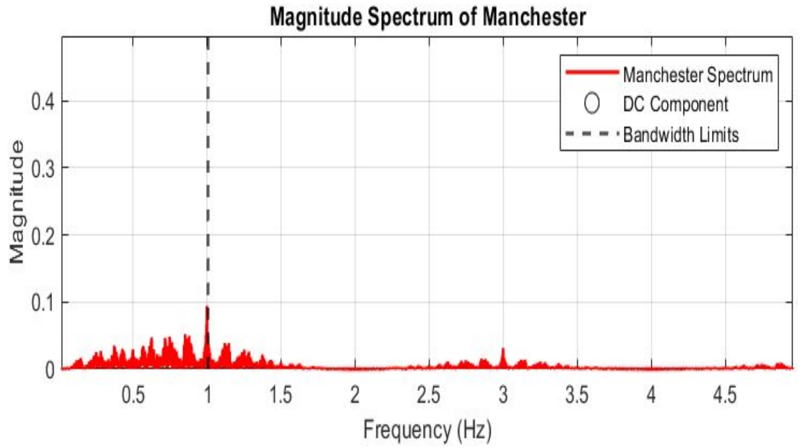
Manchester coding, on the other hand, encodes each bit with a transition in the middle of the bit period: a ‘0’ is represented by a high-to-low transition, and a ‘1’ by a low-to-high transition. This ensures frequent transitions regardless of the data pattern, making it self-clocking and easy to synchronize. The spectrum of Manchester coding shows no DC component because the signal balances positive and negative voltages within each bit, which is why its frequency spectrum stays centered without a DC spike. However, this benefit comes at the cost of roughly doubling the bandwidth compared to Polar NRZ, making Manchester appear denser in the waveform and broader in frequency.

For the final system, we choose Manchester coding because its inherent synchronization capability greatly simplifies clock recovery at the receiver, and its zero DC component avoids baseline wander issues. Although it requires more bandwidth, the reliability it offers is critical for accurate data transmission in sensor networks.


Polar NRZ Line Coding
Amplitude
1
0.5
0
-0.5
-1
120
140
160
180
200
220
240
260
Time


Magnitude Spectrum of Polar NRZ
Polar NRZ Spectrum
DC Component
Bandwidth Limits
Magnitude
0.2
0.15
0.1
0.05
0
0.5
1
1.5
2
2.5
3
3.5
4
Frequency (Hz)


Manchester Line Coding
Amplitude
1
0.5
0
120
140
160
180
200
220
240
260
Time


Magnitude Spectrum of Manchester
Manchester Spectrum
DC Component
Bandwidth Limits
Magnitude
0.4
0.3
0.2
0.1
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Frequency (Hz)

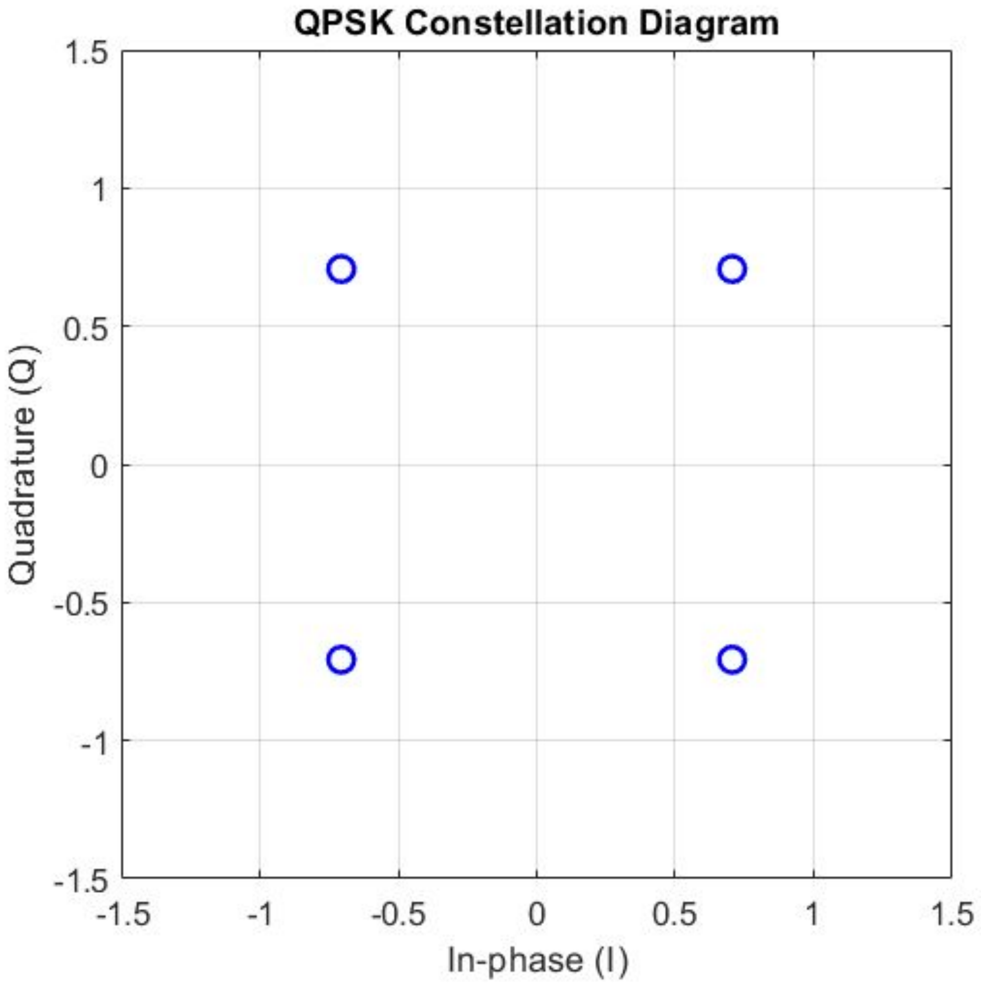
4.Digital Modulation using QPSK

In this part, we take the encoded bitstream and group the bits into pairs for QPSK (Quadrature Phase Shift Keying) modulation. QPSK encodes 2 bits per symbol, offering a good balance between spectral efficiency and robustness against noise. The constellation points are well-separated, reducing the probability of symbol errors compared to denser modulation schemes like 16-QAM, which trade off robustness for higher data rates.

The bit pairs are mapped to constellation points based on Gray coding, which ensures that adjacent symbols differ by only one bit, minimizing bit errors due to symbol misinterpretation.

This approach achieves a balance between error resilience and spectral efficiency, making QPSK a suitable choice for transmitting the line-coded bits over a wireless link in this project.

```
    %%--- matlab snippet – %%
bit_pairs = reshape(bitstream, 2, []).';

gray_map = [0 0; 0 1; 1 1; 1 0];
constellation = [1+1j, -1+1j, -1-1j, 1-1j] / sqrt(2);
modulated_signal = zeros(1, size(bit_pairs,1));
for k = 1:size(bit_pairs,1)
   [~, idx] = ismember(bit_pairs(k,:), gray_map, 'rows');
   modulated_signal(k) = constellation(idx);
end




demod_bits = zeros(size(bit_pairs));
for k = 1:length(modulated_signal)
   [~, idx] = min(abs(modulated_signal(k) - constellation));
   demod_bits(k, :) = gray_map(idx, :);
end

demod_bitstream = reshape(demod_bits.', 1, []);
```

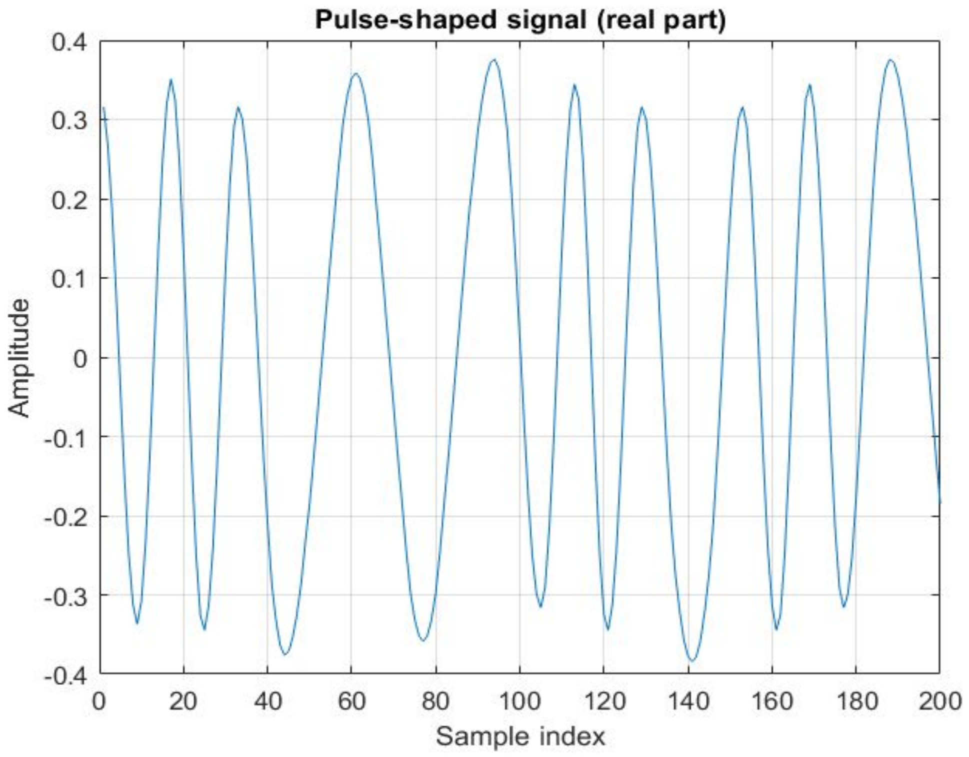

QPSK Constellation Diagram
Quadrature (Q)
In-phase (I)
1.5
1
0.5
0
-0.5
-1
-1.5
-1.5
-1
-0.5
0
0.5
1
1.5

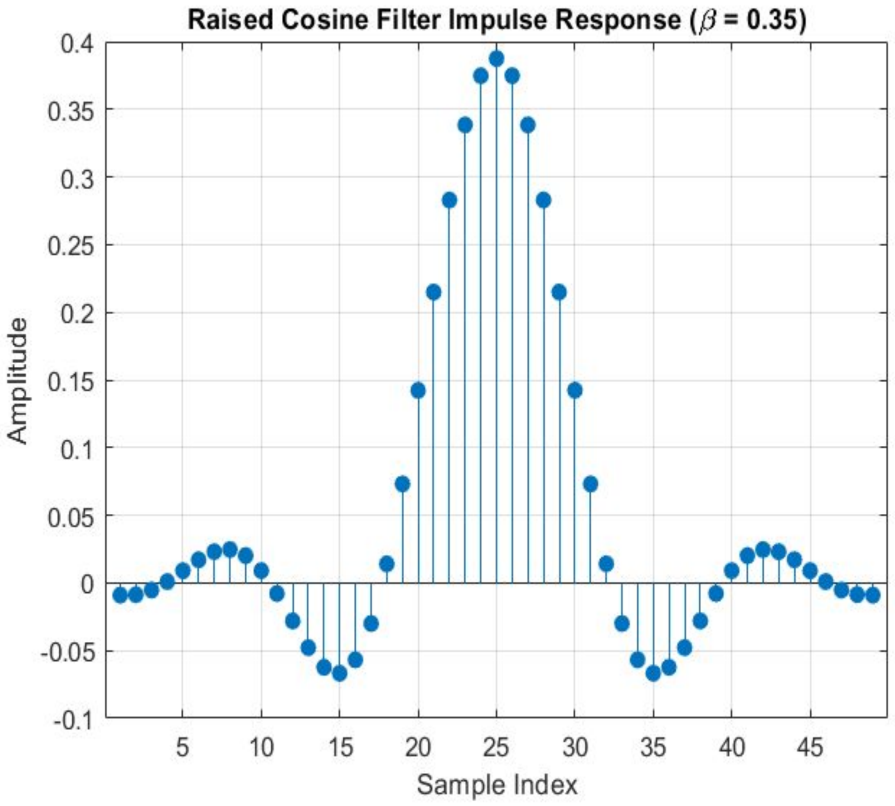
5.Pulse shaping

In order to avoid inter-symbol interference (ISI), the transmitted pulses must be shaped such that they do not interfere with each other at the sampling instants. This requires satisfying the Nyquist first criterion, which states that the pulse should have zero crossings at all integer multiples of the symbol period except at the origin. While the ideal pulse that fulfills this is the sinc function, it is non-causal and infinite in duration, making it impractical for real-time systems. To implement a realizable solution, we use a raised cosine pulse shape, which approximates the ideal sinc behavior while remaining finite and causal when truncated appropriately. This filter ensures minimal ISI and controlled bandwidth by introducing a roll-off factor that trades off sharp frequency cutoff for better time localization. At the receiver, applying a matched filter (another raised cosine) further enhances ISI suppression.

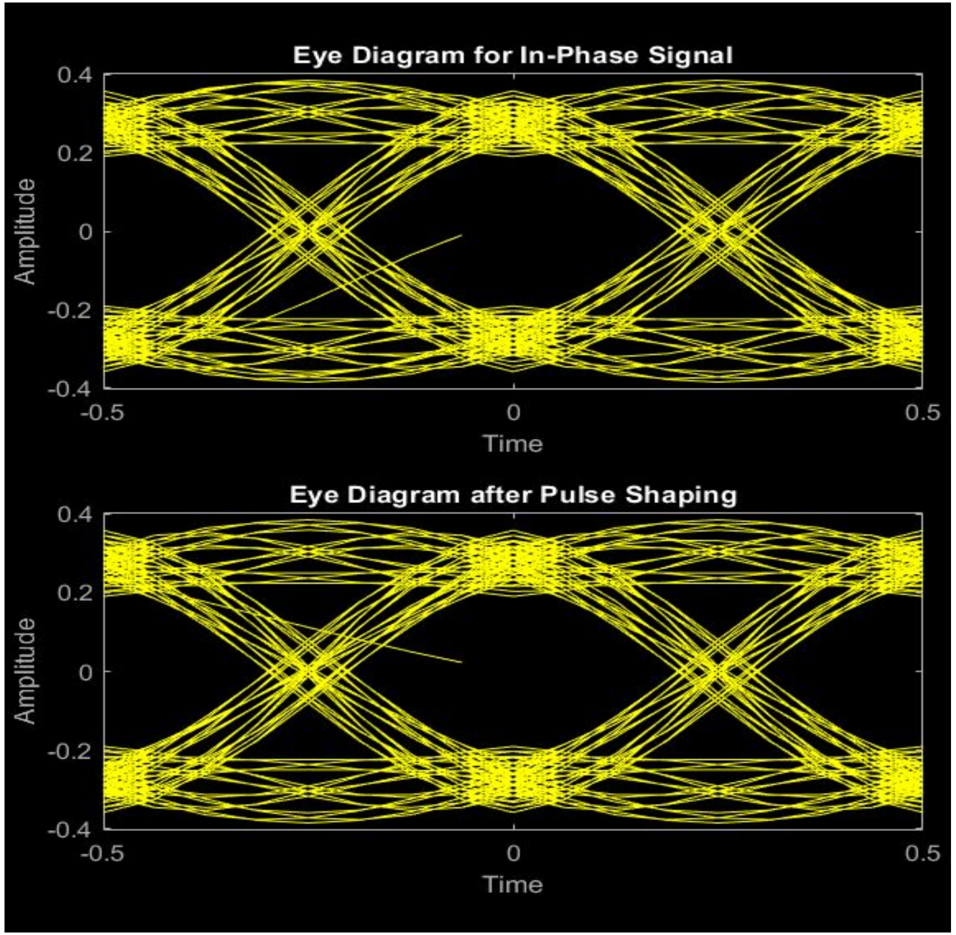
The Raised Cosine filter was designed with a roll-off factor of 0.35 and a span of 6 symbols, creating a balance between bandwidth efficiency and time-domain pulse localization. This roll-off ensures that the spectrum is not overly wide, while still providing smooth pulse transitions to reduce ISI. The span determines how many symbols the filter extends over, affecting the sharpness and latency of the response.

The eye diagram is created by overlapping segments of the received signal, aligned to the symbol period. This shows how the signal varies over time for each bit. When the eye is wide open, it means the pulses from different bits don’t overlap much (low ISI), so the signal at the sampling instant is distinct and easy to interpret as a ‘0’ or ‘1’. If the eye closes, it means pulses interfere with each other, causing ambiguity and errors. Our mostly open eye shows the raised cosine filter effectively shapes pulses to reduce overlap, ensuring clear bit detection.


Pulse-shaped signal (real part)
Amplitude
Sample index
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
20
40
60
80
100
120
140
160
180
200



Raised Cosine Filter Impulse Response (β = 0.35)
Amplitude
Sample Index



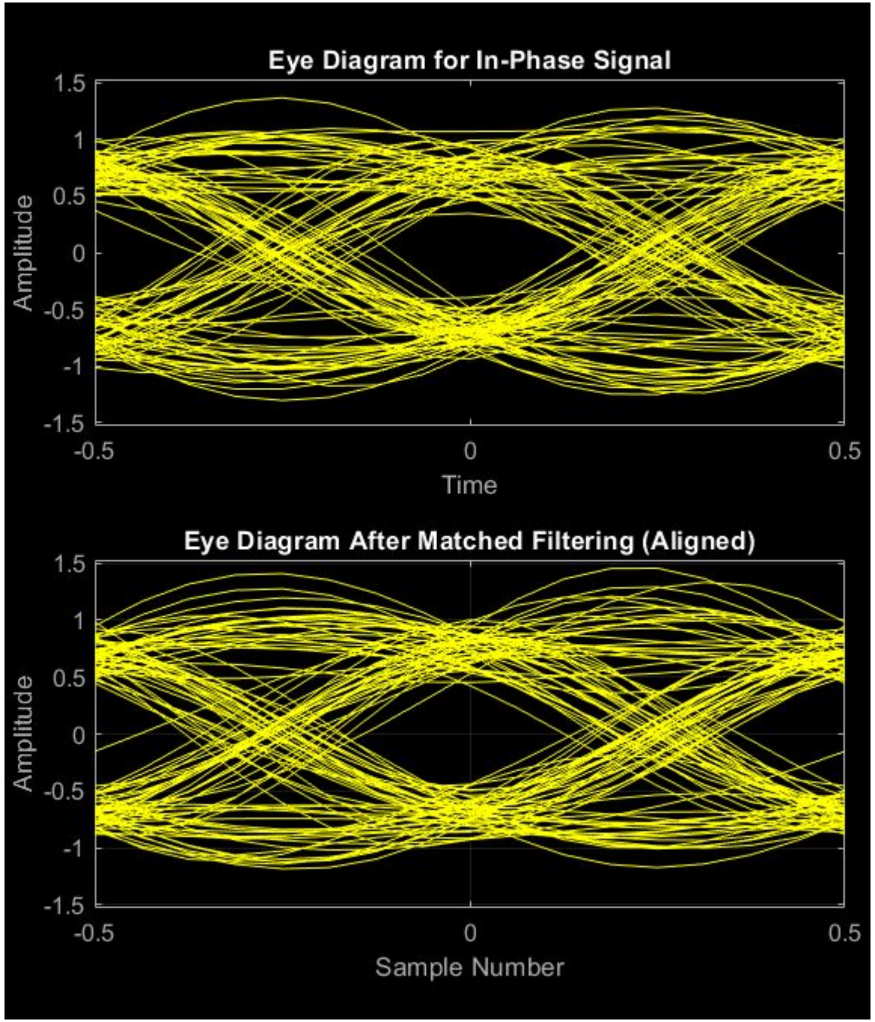
Eye Diagram for In-Phase Signal
Eye Diagram after Pulse Shaping
Amplitude
Time

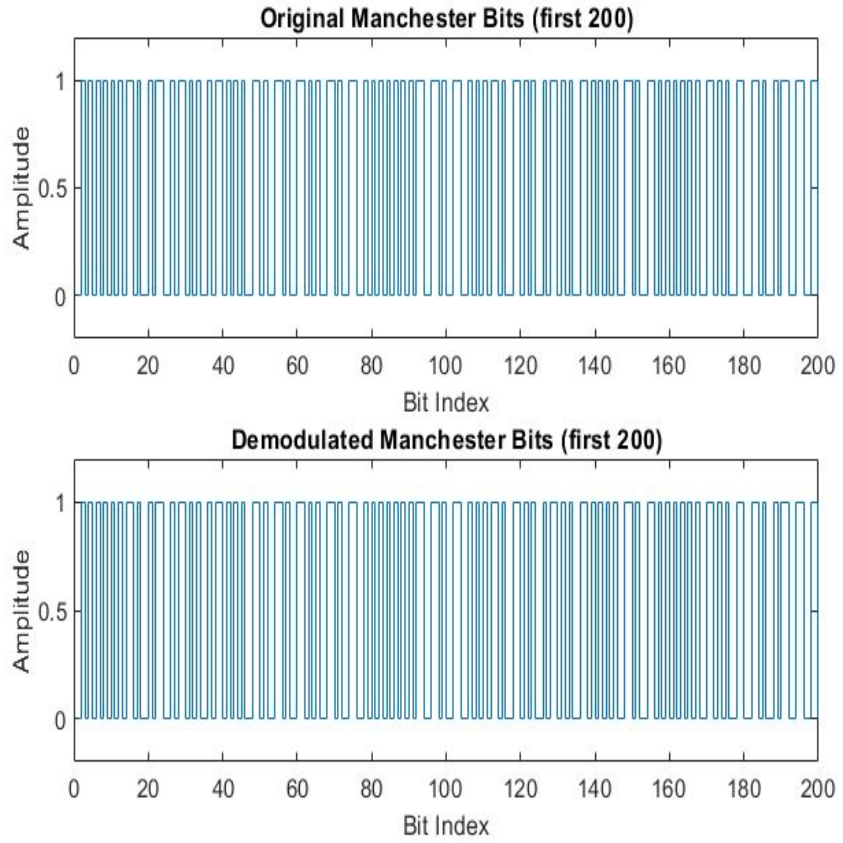
6.AWGN Channel Simulation

The simulation models noise as Additive White Gaussian Noise (AWGN), representing random thermal noise with constant power spectral density and Gaussian amplitude distribution. Noise is added to the normalized QPSK modulated signal using MATLAB's `awgn()` function, which adjusts noise power based on a specified signal-to-noise ratio defined in terms of bit energy to noise spectral density (Eb/N0). A range of (Eb/N0) values from 0 dB to 20 dB in 2 dB increments is used to simulate different channel conditions, from noisy to relatively clean. This range is chosen to observe the bit error rate performance across various noise levels, providing insight into the communication system's robustness and error resilience.

7.Receiver Design

At the receiver, we reverse the entire process by first demodulating and then decoding the received signal. A crucial step is choosing the correct sampling instant. The eye diagram, plotted after compensating for the filter delay, shows the eye opening centered at zero delay. This indicates that the optimal sampling instant occurs at zero offset relative to the compensated signal. Sampling at this point minimizes inter-symbol interference and maximizes the likelihood of correctly detecting each symbol. Since Manchester coding relies on specific signal transitions (pairs of bits 10 for logical 0 and 01 for logical 1), noise in the channel can cause bit flips that create invalid Manchester patterns. These invalid patterns lead to bits being lost or marked as errors, especially at lower SNR values. The decoding process detects such invalid Manchester pairs and replaces them with zeros, which introduces decoding inaccuracies but allows the process to continue.


Eye Diagram for In-Phase Signal
Amplitude
Time
1.5
1
0.5
0
-0.5
-1
-1.5
-0.5
0
0.5
Eye Diagram After Matched Filtering (Aligned)
Amplitude
Sample Number
1.5
1
0.5
0
-0.5
-1
-1.5
-0.5
0
0.5


Original Manchester Bits (first 200)
Amplitude
Bit Index
1
0.5
0
0
20
40
60
80
100
120
140
160
180
200
Demodulated Manchester Bits (first 200)
Amplitude
Bit Index
1
0.5
0
0
20
40
60
80
100
120
140
160
180
200

Next, convolutional error correction is applied through Viterbi decoding, which uses the trellis structure defined by the convolutional encoder at the transmitter. The Viterbi algorithm efficiently finds the most likely transmitted bit sequence given the noisy received bits, significantly improving error correction capabilities. This step is essential because, even when the raw bit error rate (BER) before Viterbi decoding is non-negligible, the Viterbi decoder recovers many of the original bits correctly, reducing the post-decoding BER and thus enhancing overall system performance.

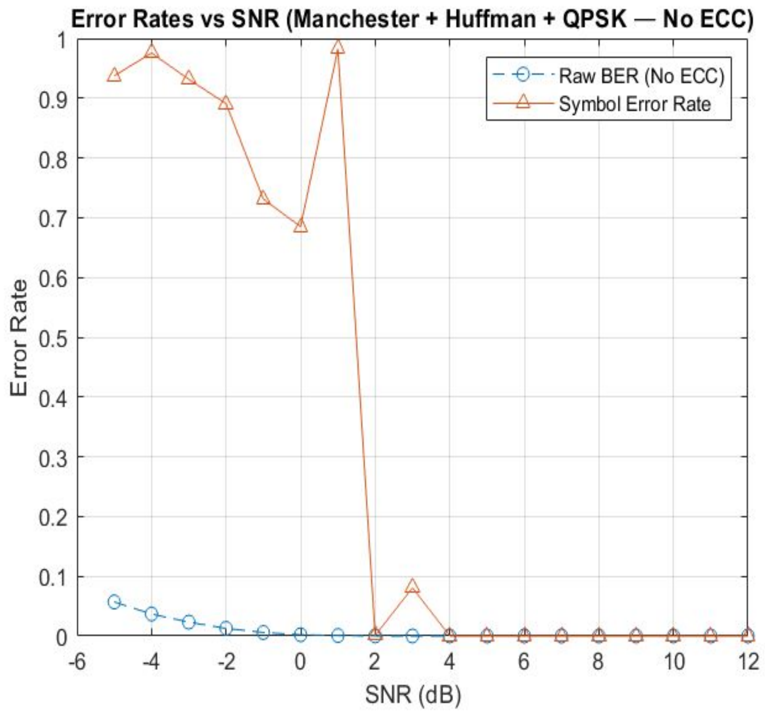
Finally, Huffman decoding is performed using the same dictionary constructed based on the source statistics at the transmitter. Huffman coding reduces the number of bits needed to represent the sensor data by assigning shorter codes to more frequent symbols, improving bandwidth efficiency. Decoding with the correct dictionary ensures accurate reconstruction of the original quantized sensor data symbols.

This multi-stage decoding pipeline—matched filtering, optimal sampling guided by the eye diagram, Gray-coded QPSK demodulation, Manchester decoding, convolutional error correction with Viterbi decoding, and Huffman decoding—ensures that the noisy received signal is processed to recover the transmitted sensor data as accurately as possible.

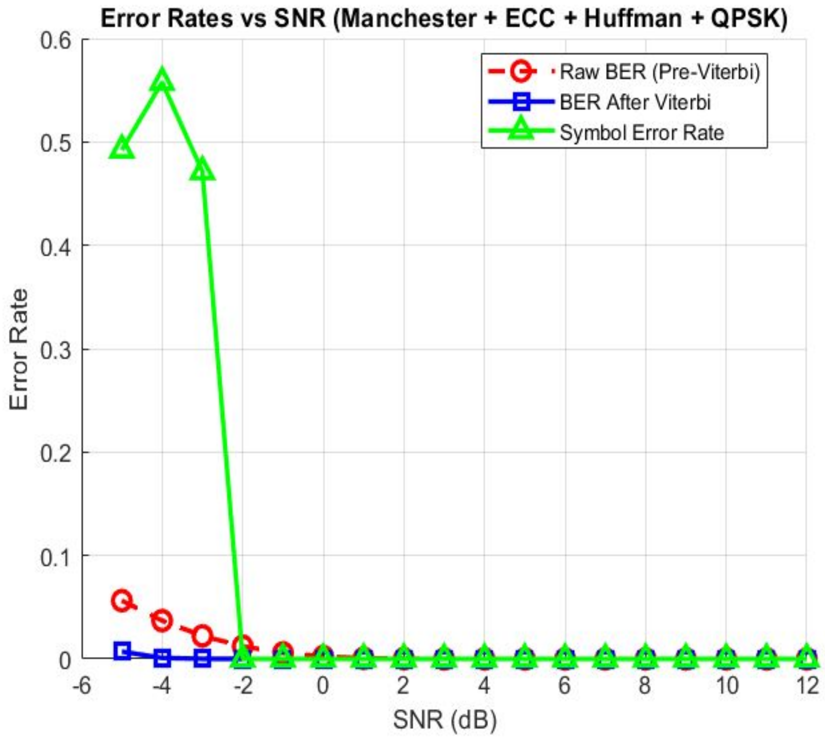
The bit error rate (BER) before and after Viterbi decoding, along with the symbol error rate (SER) of the reconstructed sensor data, serve as key metrics to evaluate system performance. Without error correction, the raw BER remains non-zero until the SNR reaches around 2 dB, meaning bit errors persist at lower SNRs and contribute to significant symbol losses during Manchester decoding. However, with the Viterbi decoder applied, the BER is drastically reduced, and notably, at -2 dB SNR, the BER after decoding is already zero, indicating highly effective error correction even in noisy conditions. As the SNR increases further (2 dB and above), the raw BER decreases naturally, and the SER correspondingly approaches zero, demonstrating the combined benefits of source coding, line coding, modulation, and powerful convolutional error correction for reliable data recovery over a noisy channel.

In terms of bandwidth and data rate, the system's effective bit rate after source coding was estimated at approximately 38,218 bps, with a corresponding symbol rate of about 19,109 symbols/sec based on the QPSK modulation (2 bits per symbol). Given the pulse shaping using a root-raised cosine filter with a roll-off factor of 0.35, the estimated bandwidth required to transmit the signal is around 25.8 kHz. This bandwidth estimation aligns with practical communication systems, where pulse shaping and modulation order control the occupied bandwidth. Source coding with Huffman compression effectively reduces the number of bits to transmit, thus lowering the bandwidth and improving spectral efficiency. The system design balances efficient bandwidth use and reliable transmission, leveraging modulation, line coding, and error correction to maximize data throughput within channel constraints.

This simulation highlights the practical necessity of integrating these coding and decoding techniques to achieve robust, efficient communication over noisy channels. The preamble's role in synchronization and timing recovery, along with the convolutional coding's error resilience, showcases critical design choices for reliable data transmission in real-world systems.



Error Rates vs SNR (Manchester + Huffman + QPSK — No ECC)
Raw BER (No ECC)
Symbol Error Rate
Error Rate
SNR (dB)



Error Rates vs SNR (Manchester + ECC + Huffman + QPSK)
Raw BER (Pre-Viterbi)
BER After Viterbi
Symbol Error Rate
Error Rate
SNR (dB)