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.*

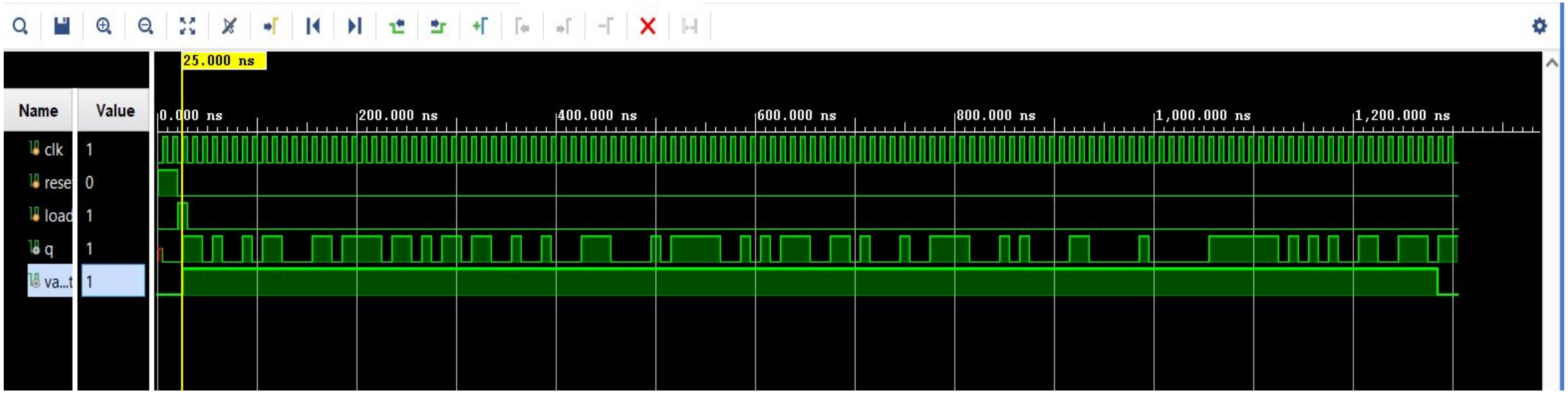
- *Next, we design a modular shift register (`shiftr.v`) that can be configured with any number of flip-flops.*

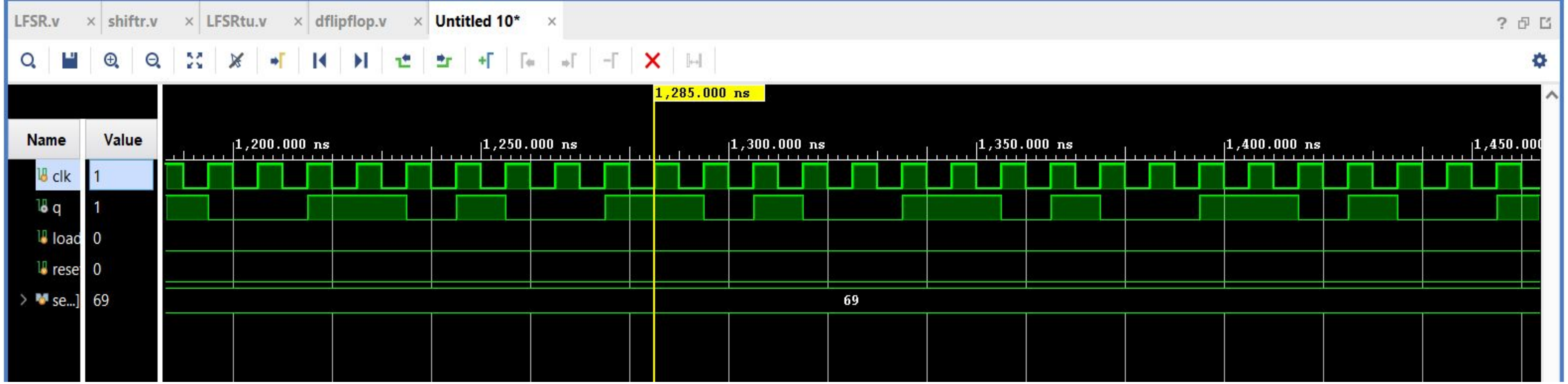
- *The LFSR is built by connecting the shift register with appropriate feedback taps.*

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.

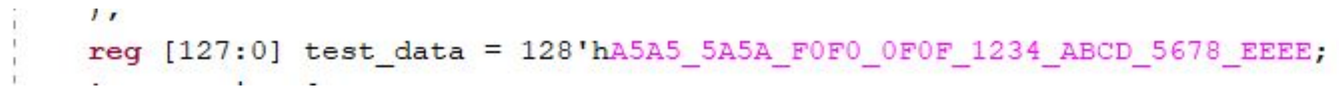
- While using different clock domains is not ideal in production systems, for a compact and controlled design like ours, this approach helps keep the pipeline moving efficiently.

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

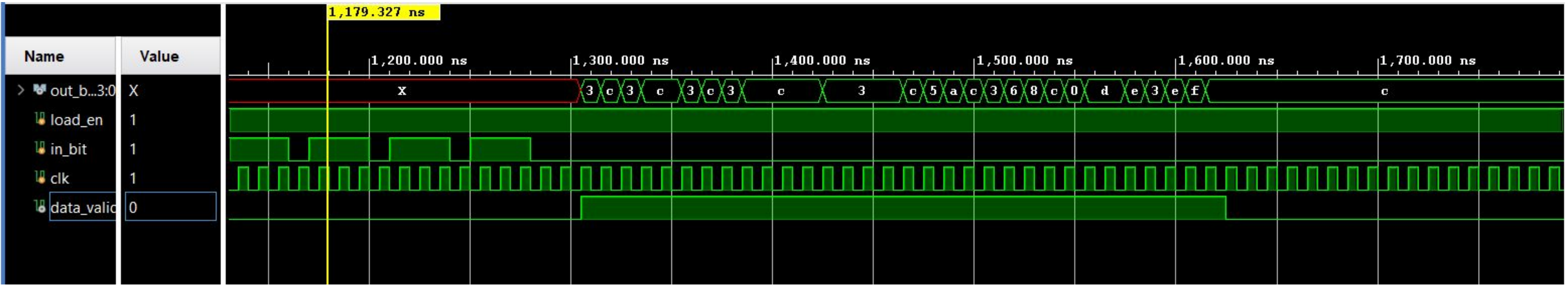
- The 8th, 16th, 24th, and 32nd bits (from the original serial input), when interpreted from MSB to LSB, form the binary sequence `1100`. This translates to the hexadecimal value 3, and it matches the first output value of the interleaver, as we begin reading from the last 4 columns of the matrix.

- 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.

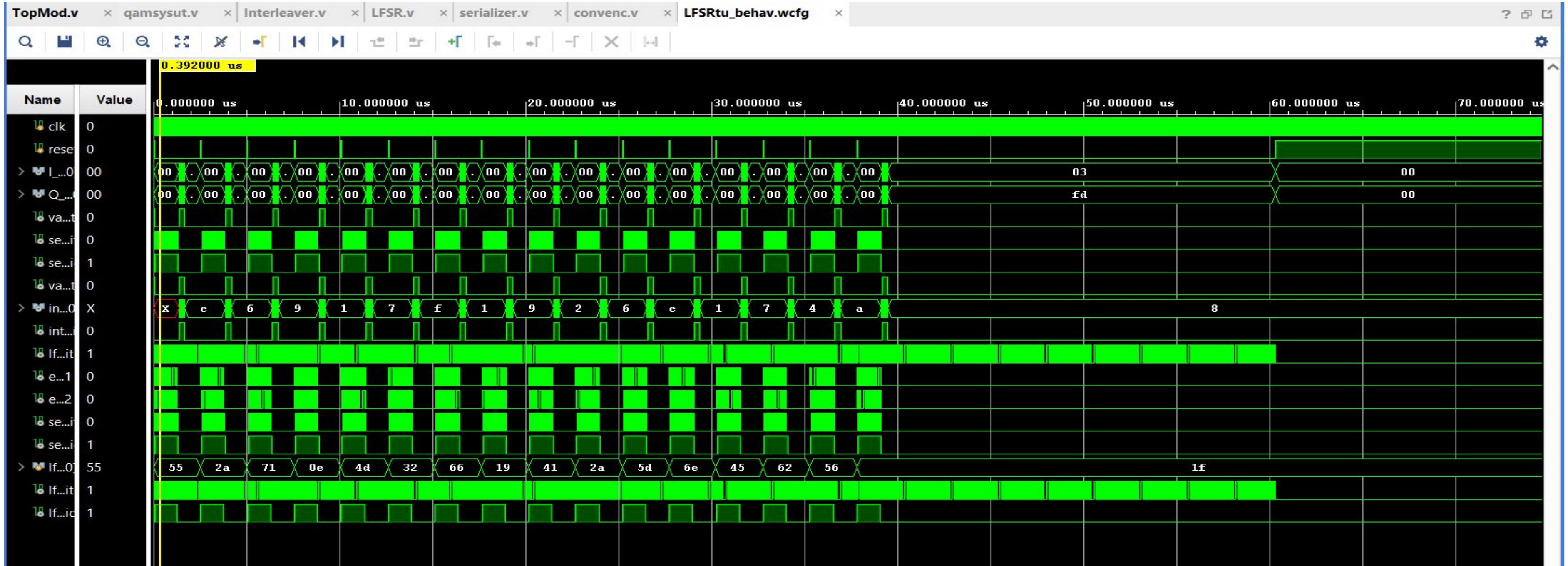
FULL transmitter Pipeline simulation :

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 ($62 \times 2 + 4$), 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

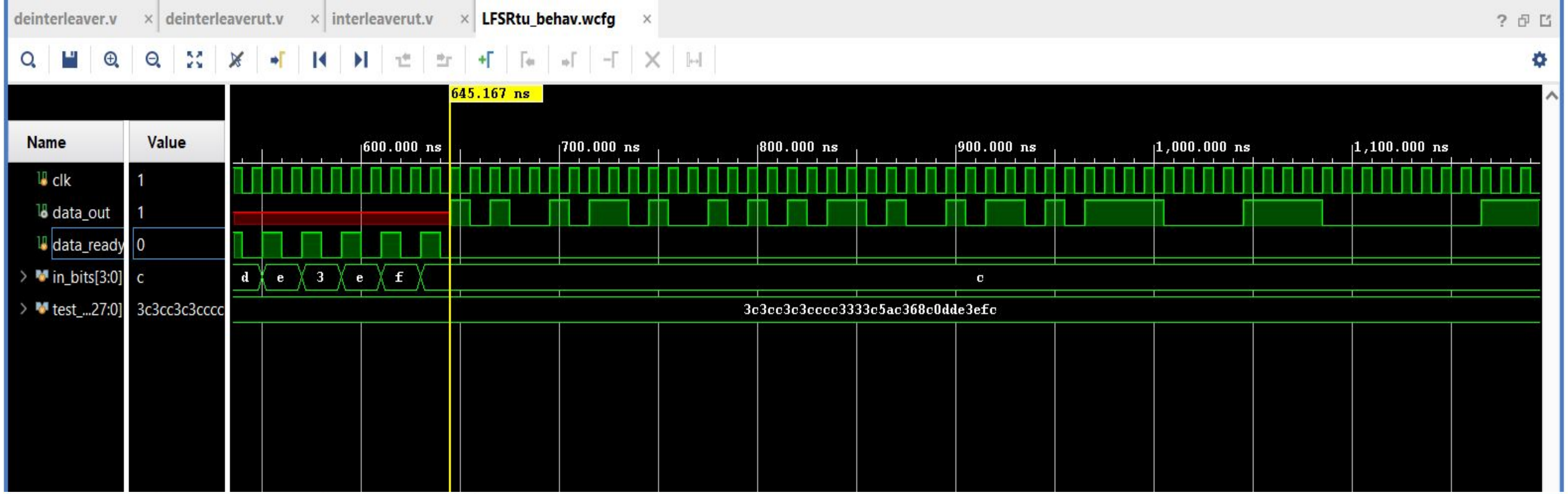
Once the interleaver is built and verified, implementing the deinterleaver becomes relatively straightforward. The core idea is to reverse the logic applied during interleaving.

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.

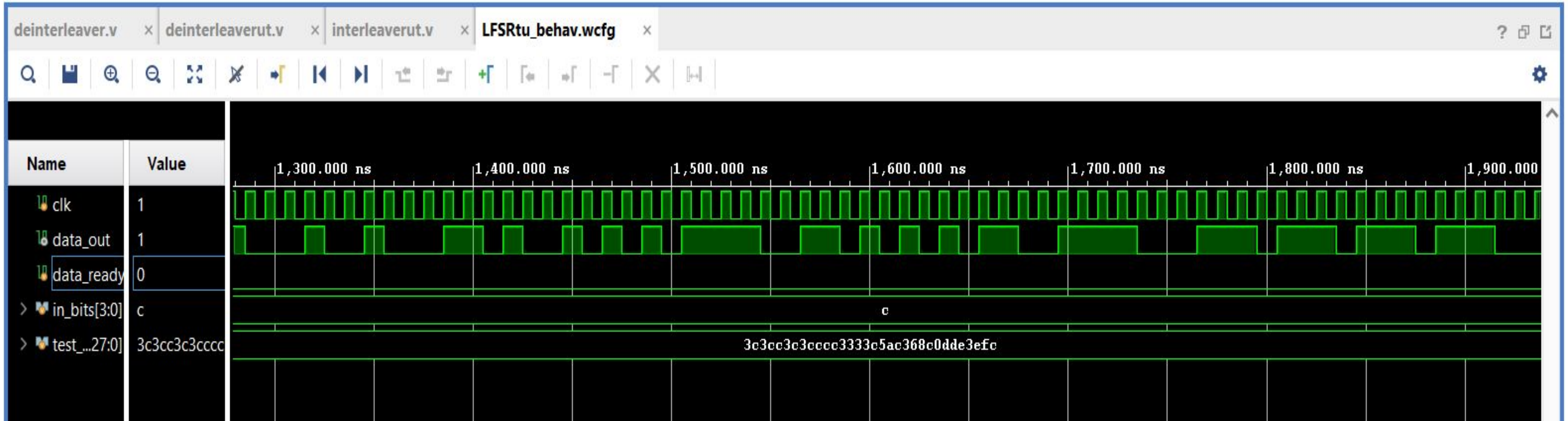
Once the matrix is populated, we read it out serially row-wise, which reconstructs the original pre-interleaved test sequence.

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.



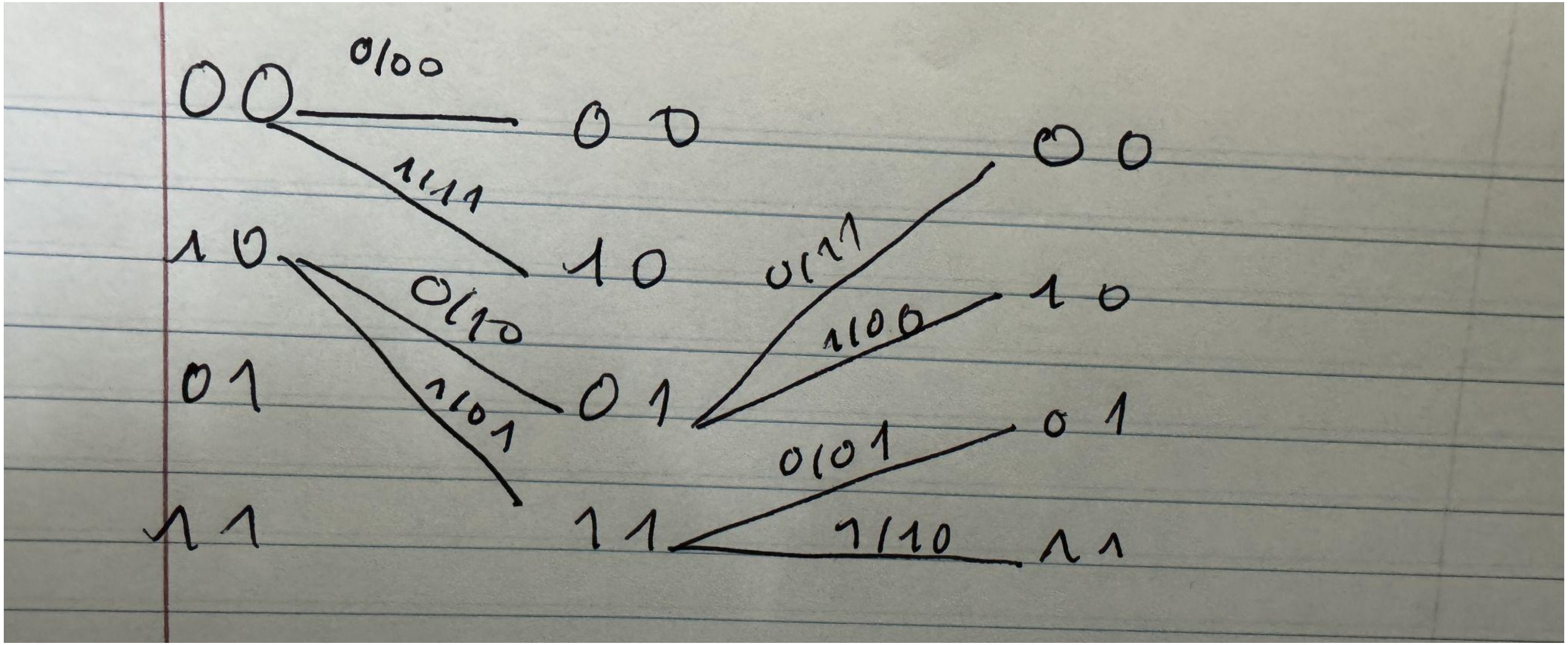
deinterleaver.v
deinterleaverut.v
interleaverut.v
LFSRtu_behav.wcfg
645.167 ns
Name
Value
clk 1
data_out 1
data_ready 0
in_bits[3:0] c
test_...27:0] 3c3cc3c3cccc
600.000 ns
700.000 ns
800.000 ns
900.000 ns
1,000.000 ns
1,100.000 ns
d
e
3
e
f
c
3c3cc3c3cccc3333c5ac368c0dde3efc



deinterleaver.v
deinterleaverut.v
interleaverut.v
LFSRtu_behav.wcfg
Name
Value
clk 1
data_out 1
data_ready 0
in_bits[3:0] c
test_...27:0] 3c3cc3c3cccc
1,300.000 ns
1,400.000 ns
1,500.000 ns
1,600.000 ns
1,700.000 ns
1,800.000 ns
1,900.000
c
3c3cc3c3cccc3333c5ac368c0dde3efc

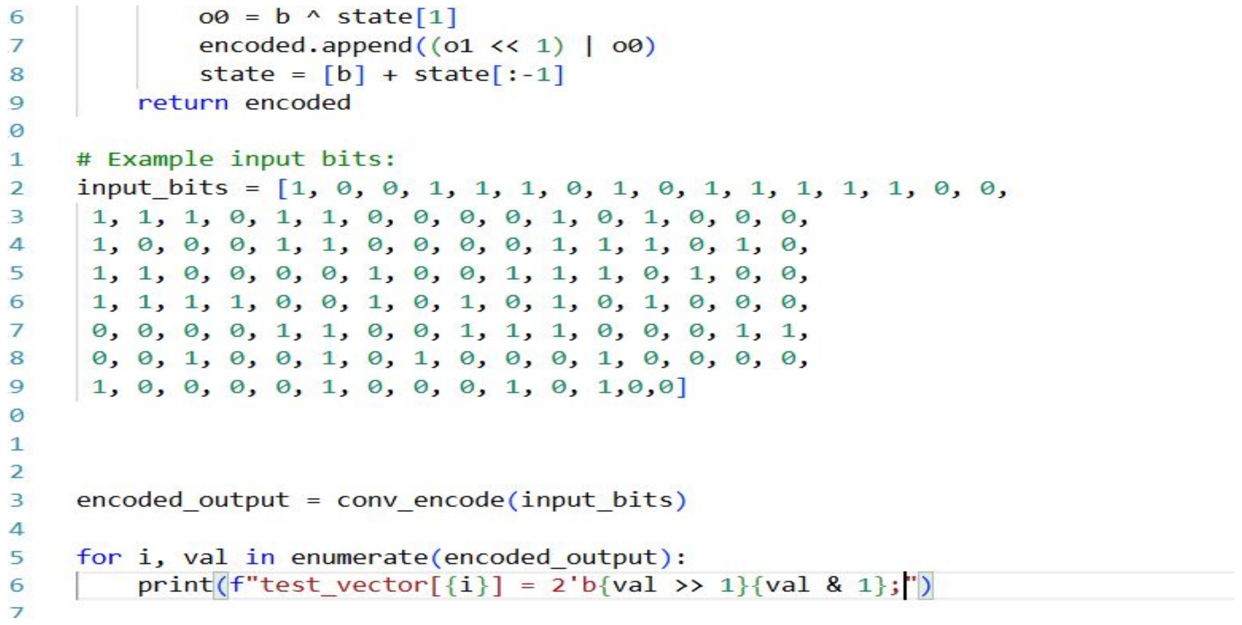
6.Viterbi Decoder

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.

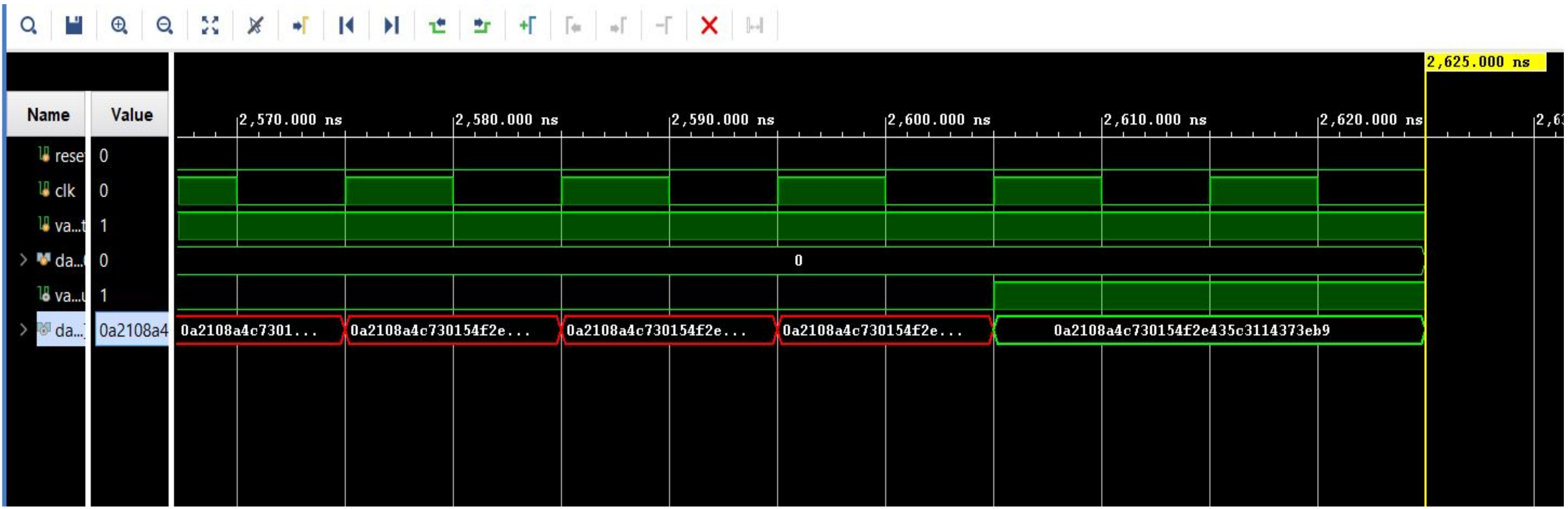


Trellis diagram of the (2,1,3) Viterbi decoder

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.

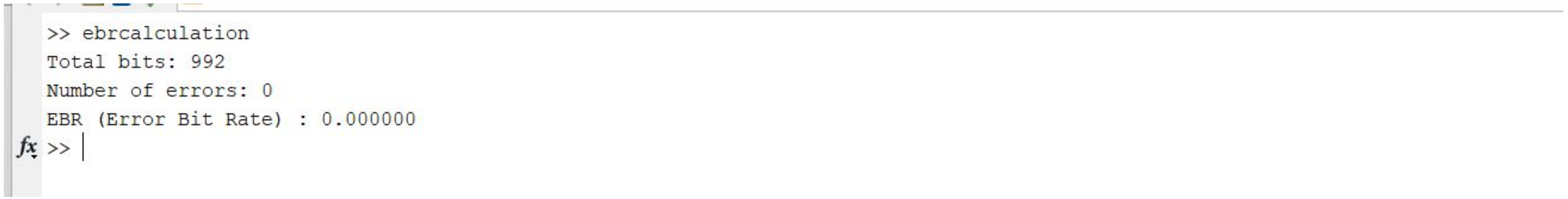


```
        o0 = b ^ state[1]
        encoded.append((o1 << 1) | o0)
        state = [b] + state[:-1]
    return encoded

# 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 :

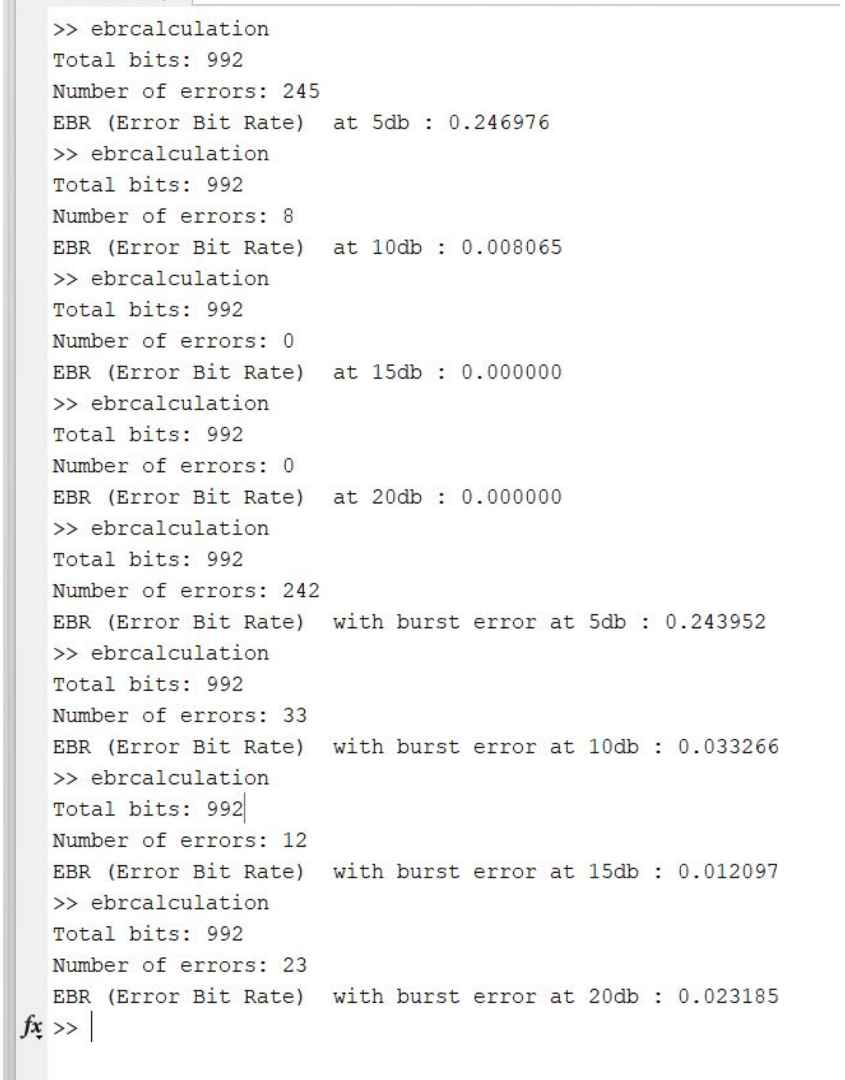
Next, the demodulated bits generated by MATLAB are fed into a top-level Verilog module containing both the deinterleaver and Viterbi decoder. The testbench runs this module and captures the decoded output bits, writing them into a text file then export this file and compare it with the original bits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



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

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
>>
```