521 Digital ASIC Design

Title: HW- 2

For

By: Saroj Shah

Date: Feb 18, 2026

HW-2, Professor Seetal Potluri

Introduction and Summary:

In this HW- 2 assignment, we are designing, and simulating fundamental combinational logic circuits using Verilog HDL and Cadence Xcelium. The following digital building blocks were implemented:

- 2-Input XOR Gate
- 1-Bit Full Adder
- 4-Bit Subtractor (using Full Adders)
- 4-Bit Unsigned Multiplier (using Full Adders)

All designs were written in Verilog at gate-level and structural modeling style. Testbenches were developed for each module to verify correctness through simulation and waveform analysis in SimVision..

Materials:

- Computer system
- Cadence Xcelium (xrun)
- SimVision waveform viewer
- MobaXterm (remote terminal access)

Data:

Please refer to the screenshot provided below:

1. **Two in-put XOR Gate:**

The XOR gate was implemented using the Boolean equation:

$$Y = A'B + AB'$$

The design uses: 2 NOT gates; 2 AND gates; 1 OR gate

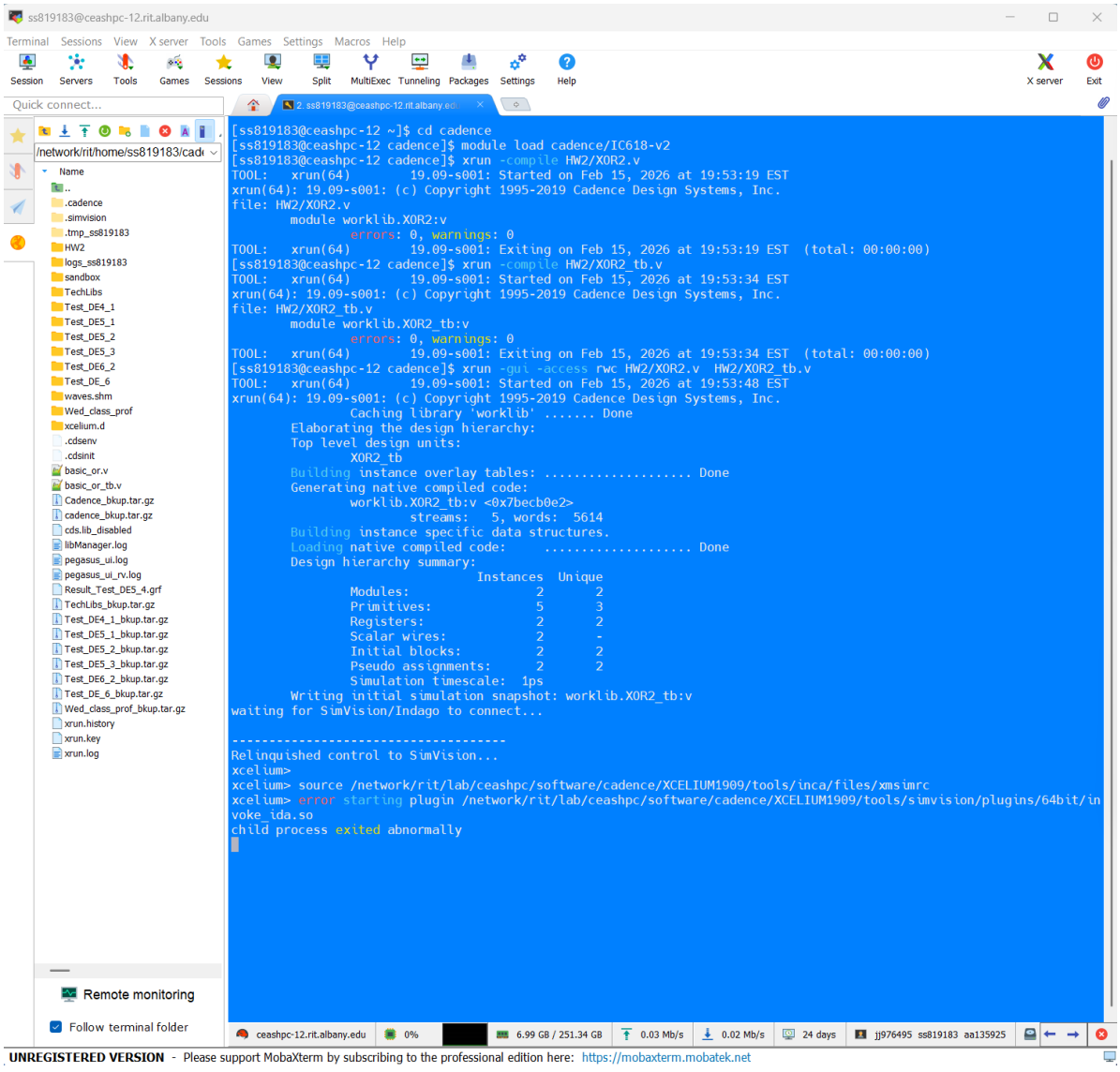


Fig: 1 (MobaXterm window)

The above figure shows the code compiling and Cadence Xcelium running without any error.

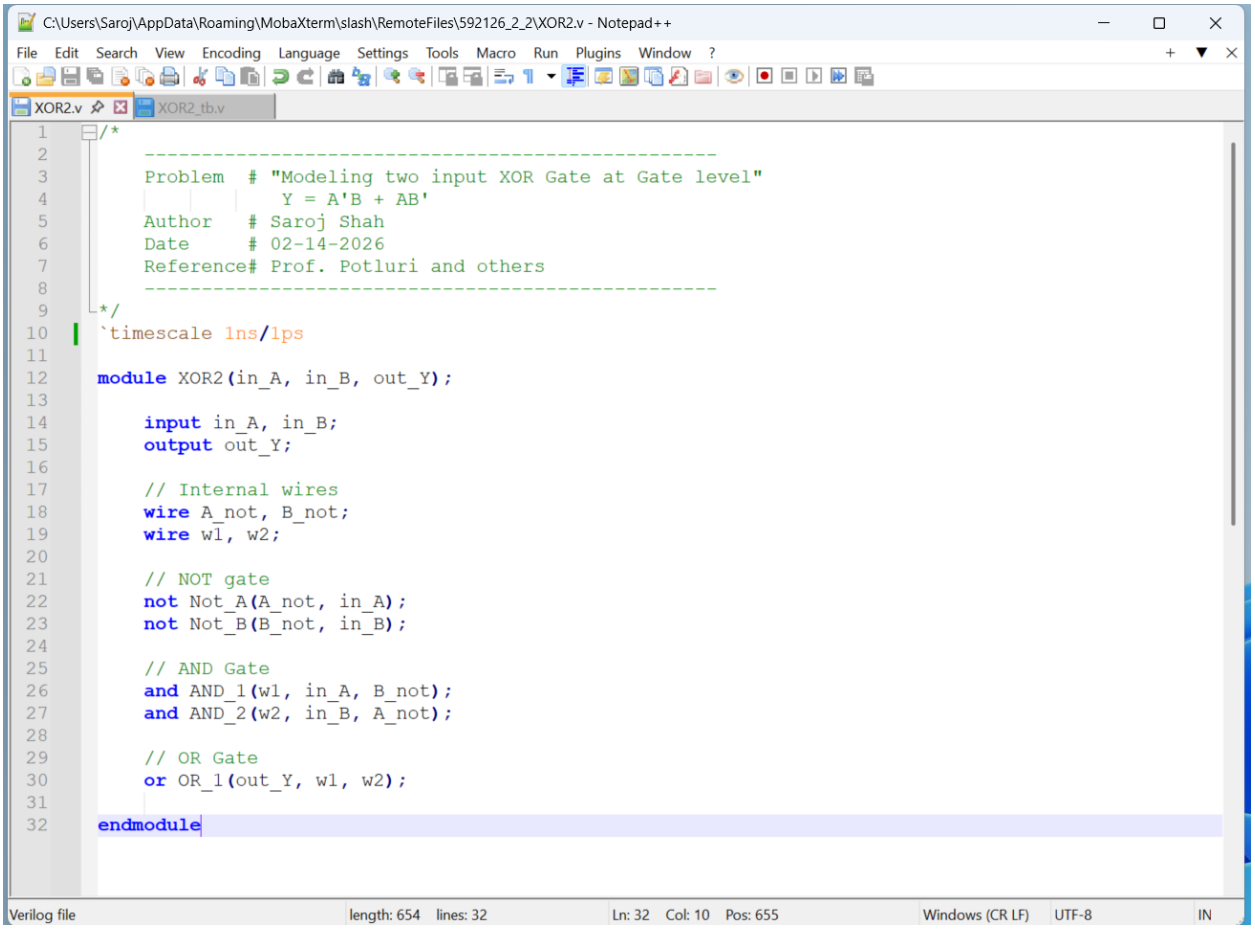


Fig: 2A (Code for XOR2 Gate)

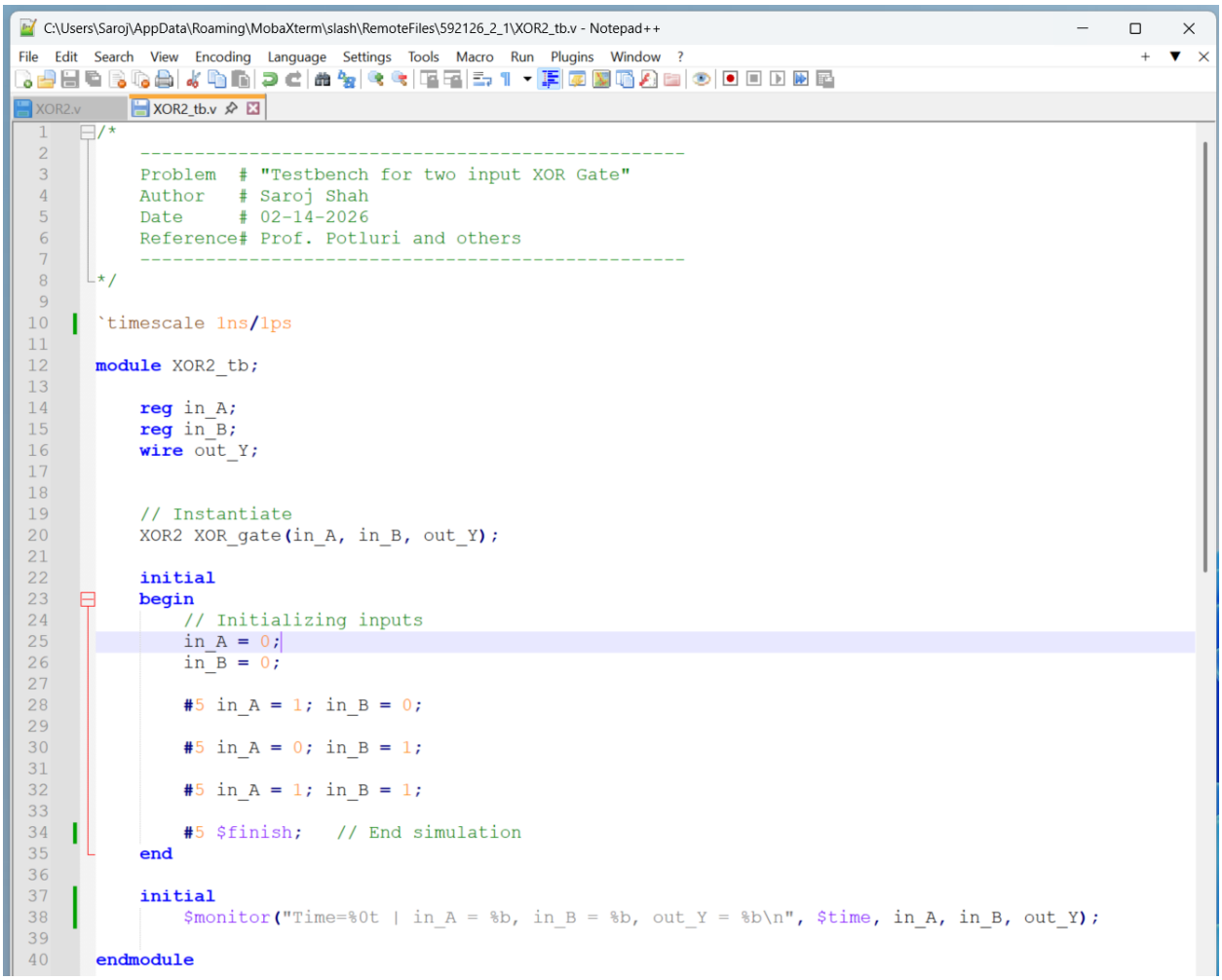
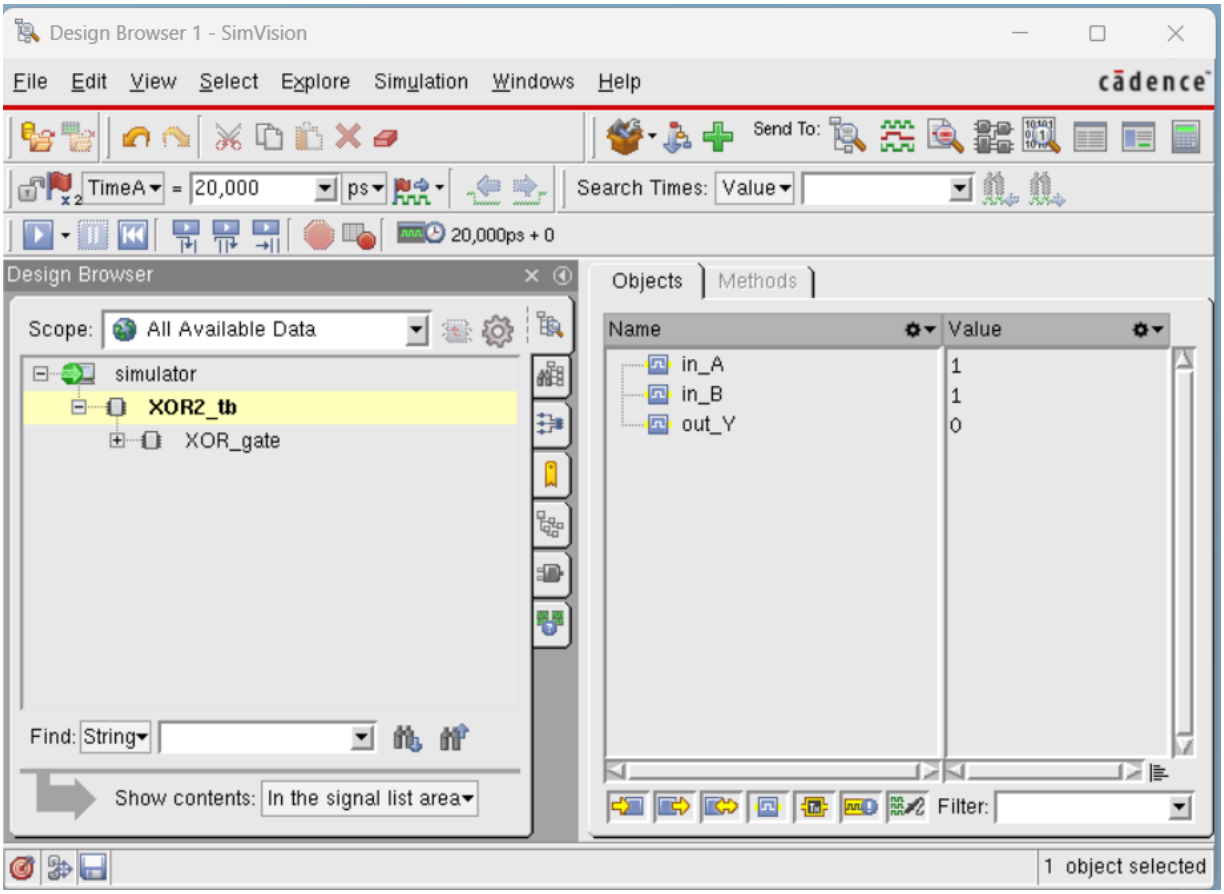


Fig: 2B (Code for XOR2 Testbench Gate)

Figure 2A and 2B shows the code for XOR gate and XOR gate testbench. The testbench code is used to run the simulation.



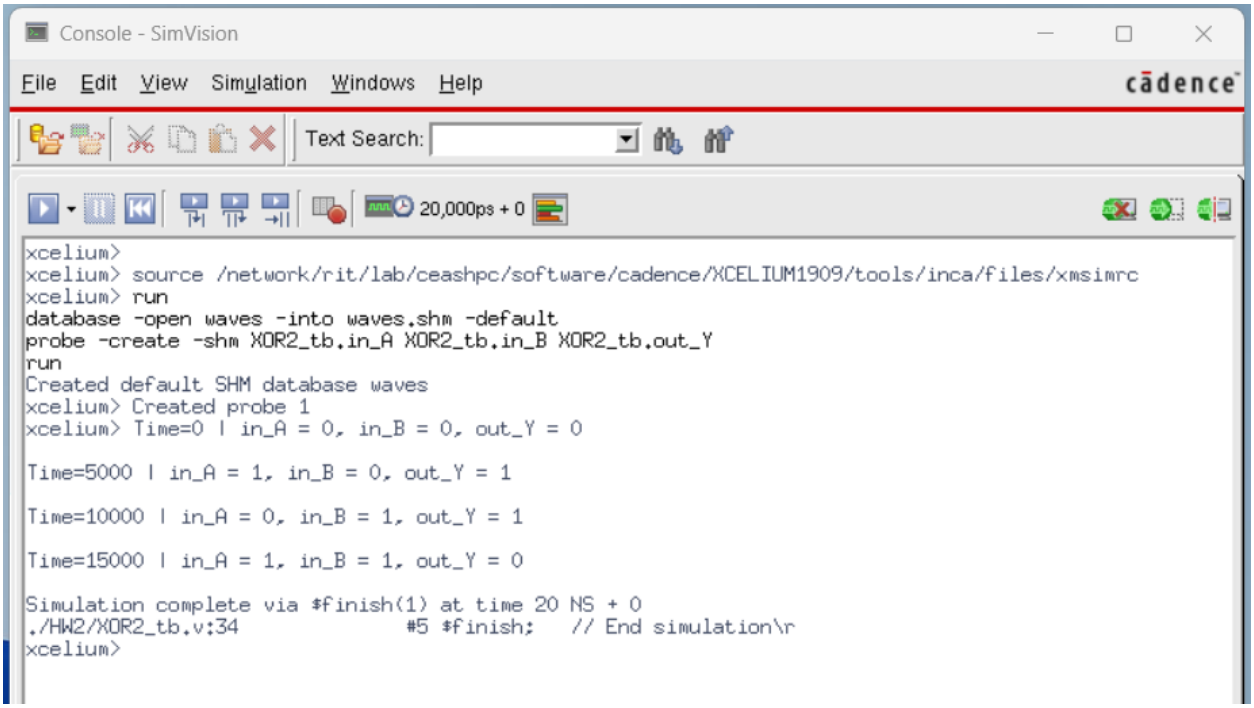


Fig: 3A (Simulation of XOR Gate)

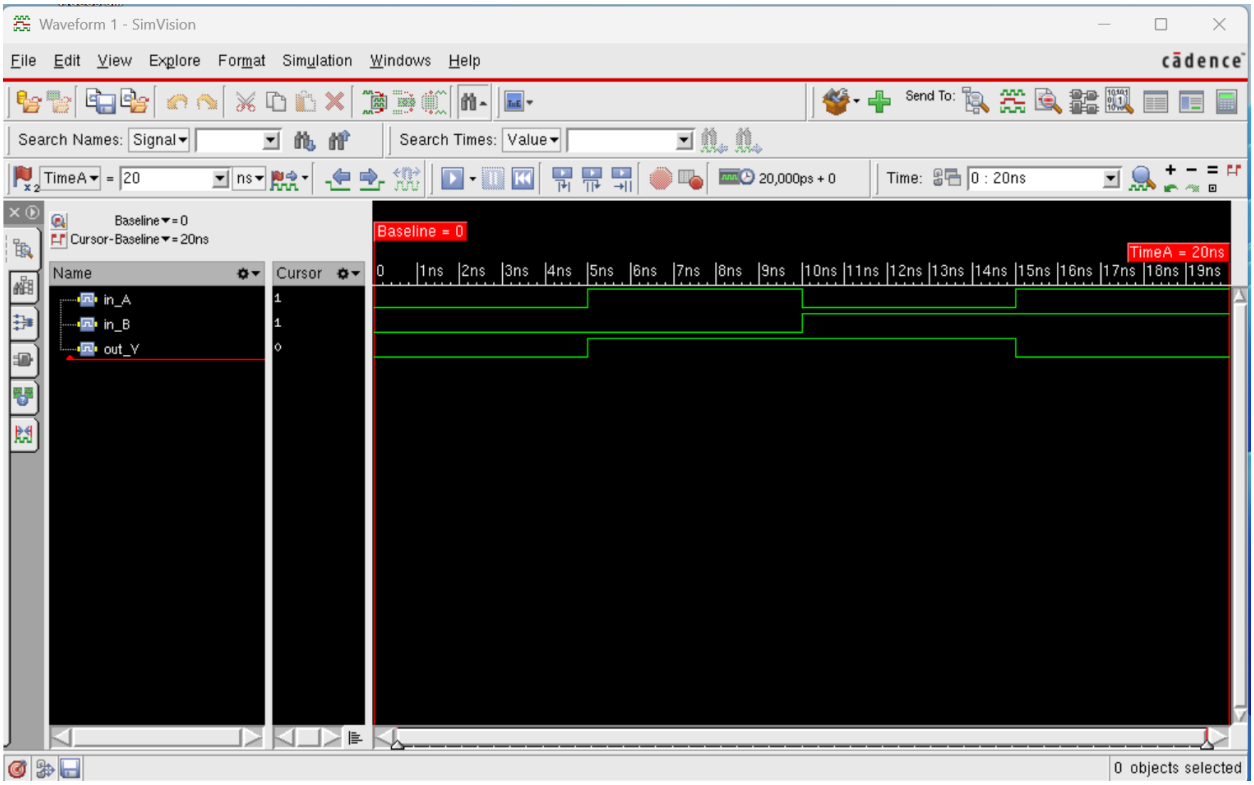


Fig: 3B (Waveform of XOR Gate)

The simulation ran successfully without compilation errors.

Waveform analysis confirms:

- When both inputs are equal (00 or 11), output = 0
- When inputs differ (01 or 10), output = 1

This verifies correct XOR behavior.

2. One-Bit Full adder:

The Full Adder was implemented using:

- $Sum = (A \oplus B) \oplus Cin$
- $Cout = A.B + (A \oplus B).Cin$

The design reuses the XOR2 module and basic AND/OR gates.

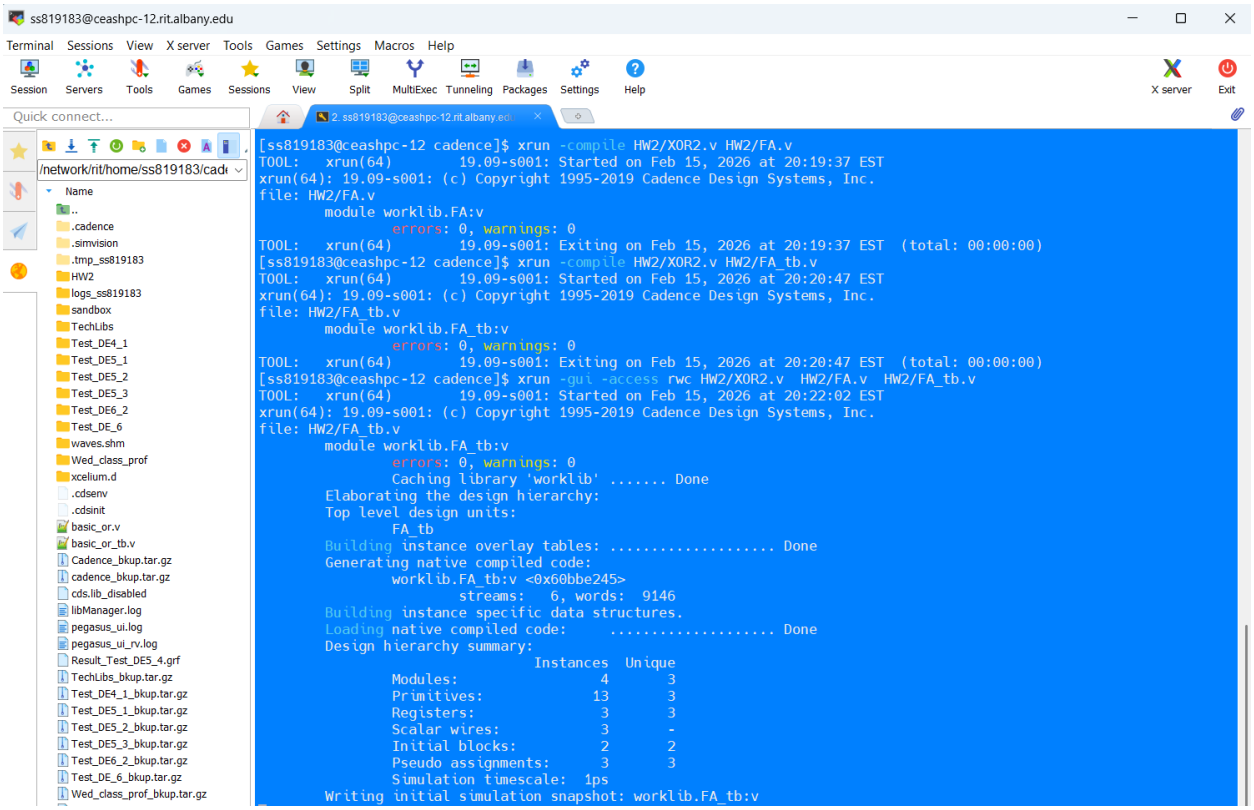


Fig: 4 (MobaXterm window for running FA)

The above Figure 4 shows the MobaXterm window for running one-bit Full Adder.

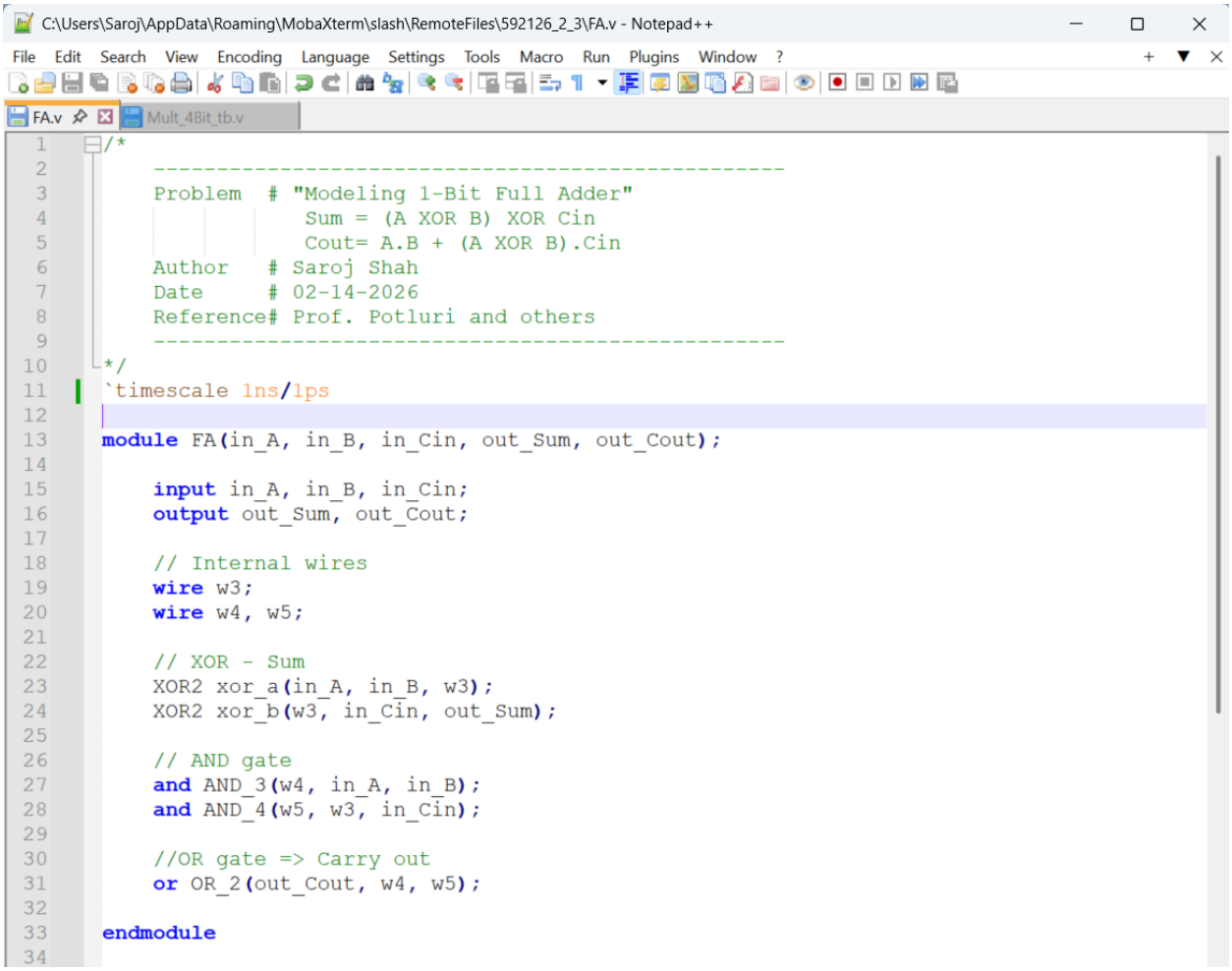


Fig: 5A (Code for 1-Bit Full Adder)

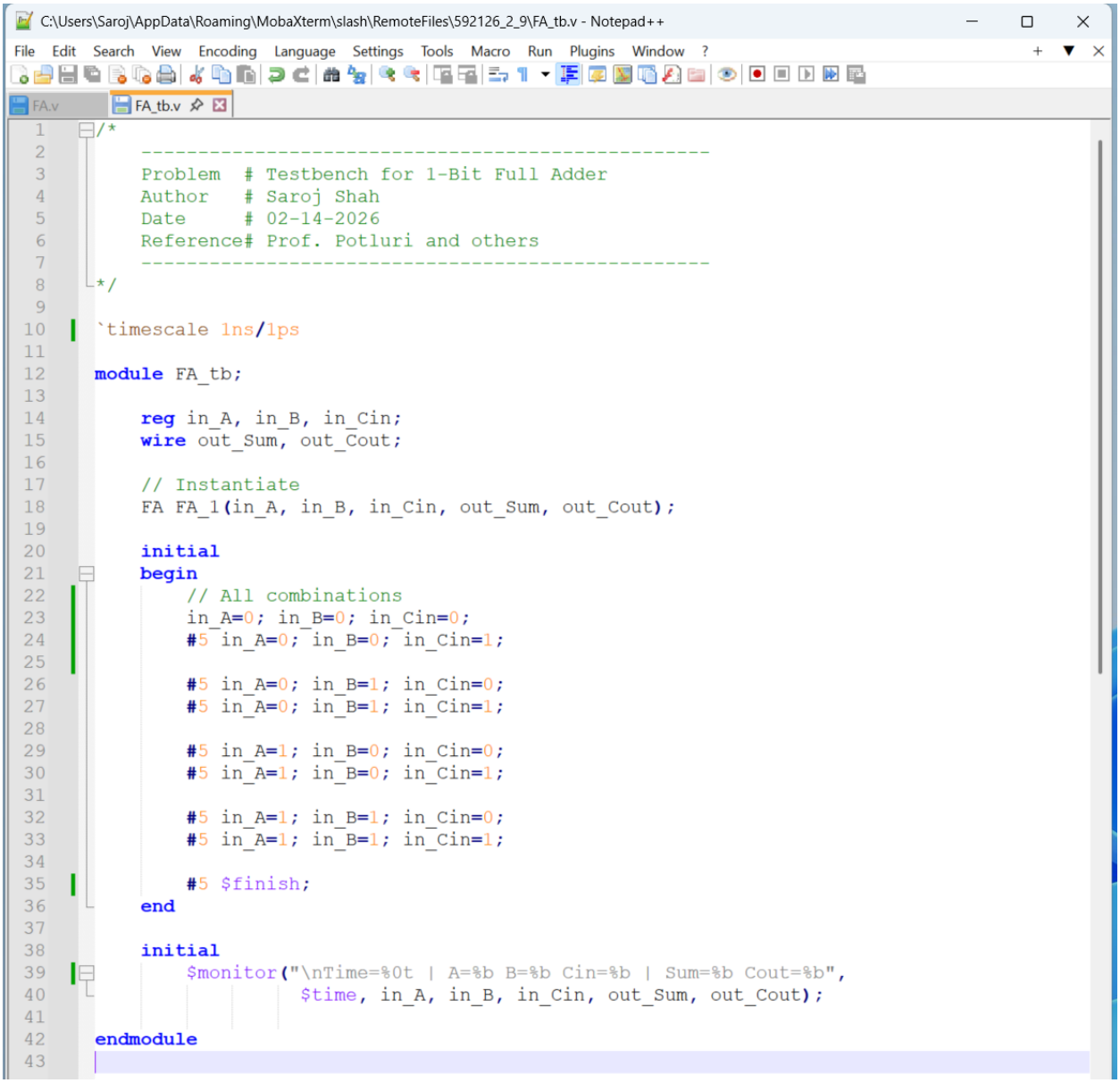


Fig: 5B (Testbench for 1-Bit Full Adder)

Figure 5A and 5B shows the code for one-bit Full Adder and one-bit Full Adder's testbench. The testbench code is used to running the simulation.

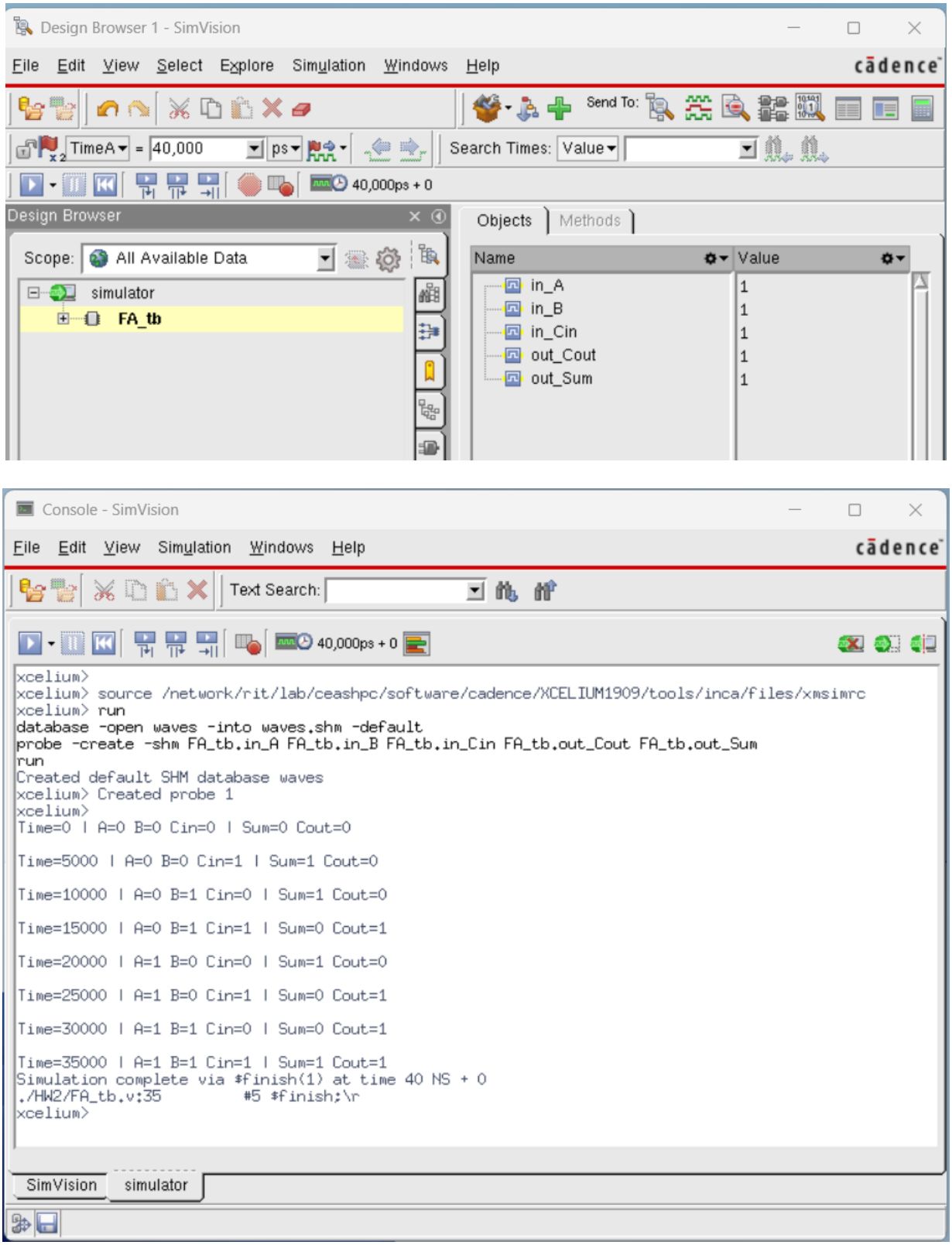


Fig: 6A (Simulation of 1-Bit Full Adder)

```
Time=0 | A=0 B=0 Cin=0 | Sum=0 Cout=0
Time=5000 | A=0 B=0 Cin=1 | Sum=1 Cout=0
Time=10000 | A=0 B=1 Cin=0 | Sum=1 Cout=0
Time=15000 | A=0 B=1 Cin=1 | Sum=0 Cout=1
Time=20000 | A=1 B=0 Cin=0 | Sum=1 Cout=0
Time=25000 | A=1 B=0 Cin=1 | Sum=0 Cout=1
Time=30000 | A=1 B=1 Cin=0 | Sum=0 Cout=1
Time=35000 | A=1 B=1 Cin=1 | Sum=1 Cout=1
```

Fig: 6B (Showing correct calculation)

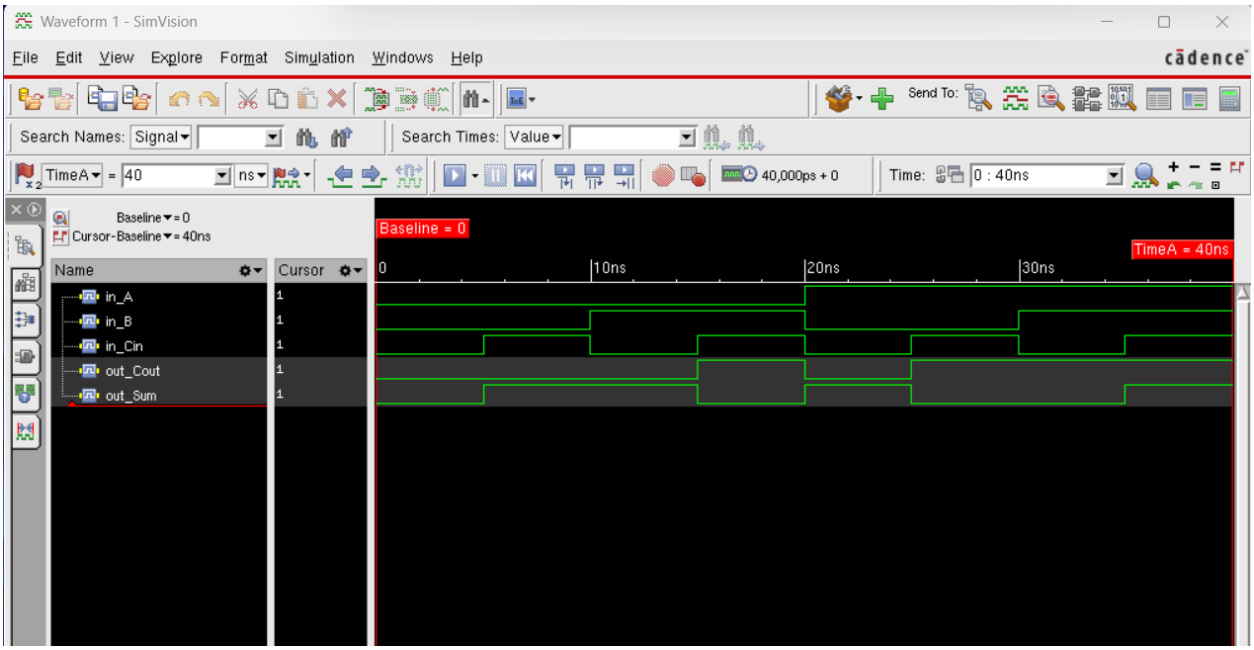


Fig: 6C (Waveform of 1-Bit Full Adder)

The simulation figures shown in figure 6A and 6B verifies correct Full Adder operation. The Sum output follows the XOR relationship of inputs, while the Carry output is asserted whenever at least two inputs are high. Same way the figure 6C, the waveforms clearly show correct propagation of carry and sum bits for all input combinations.

3. Four-Bit Full Subtractor (Using Full Adder):

Subtraction was implemented using 2's complement logic:

$$A-B=A+(B'+1)$$

Implementation steps:

a) Each bit of B is XORed with Bin (control signal).

b) Bin is also connected as the initial carry-in.

c) Four Full Adders are connected in ripple-carry fashion.

d) Final borrow is obtained from the inverted final carry

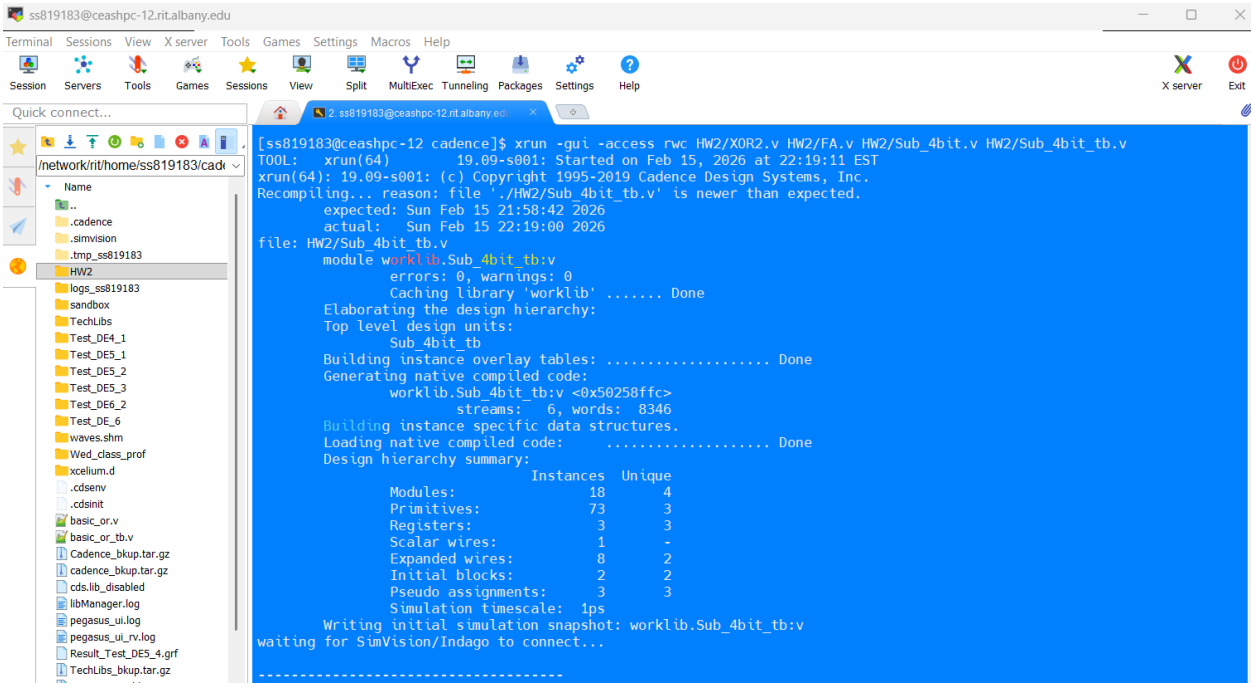


Fig: 7 (MobaXterm window)

The above Figure 4 shows the MobaXterm window for running Four-bit Subtractor.

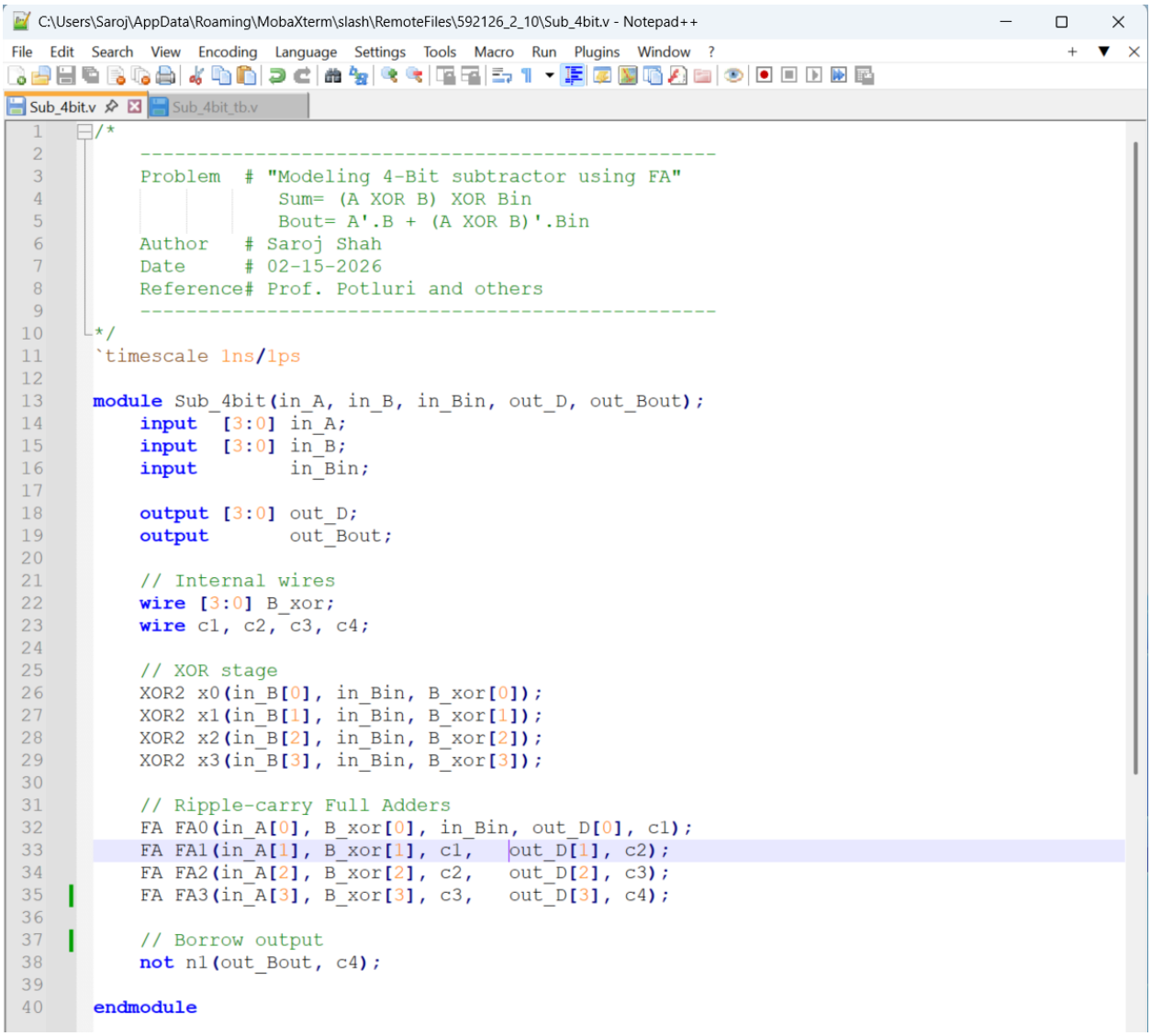


Fig: 8A (Code for 4-Bit Subtractor)

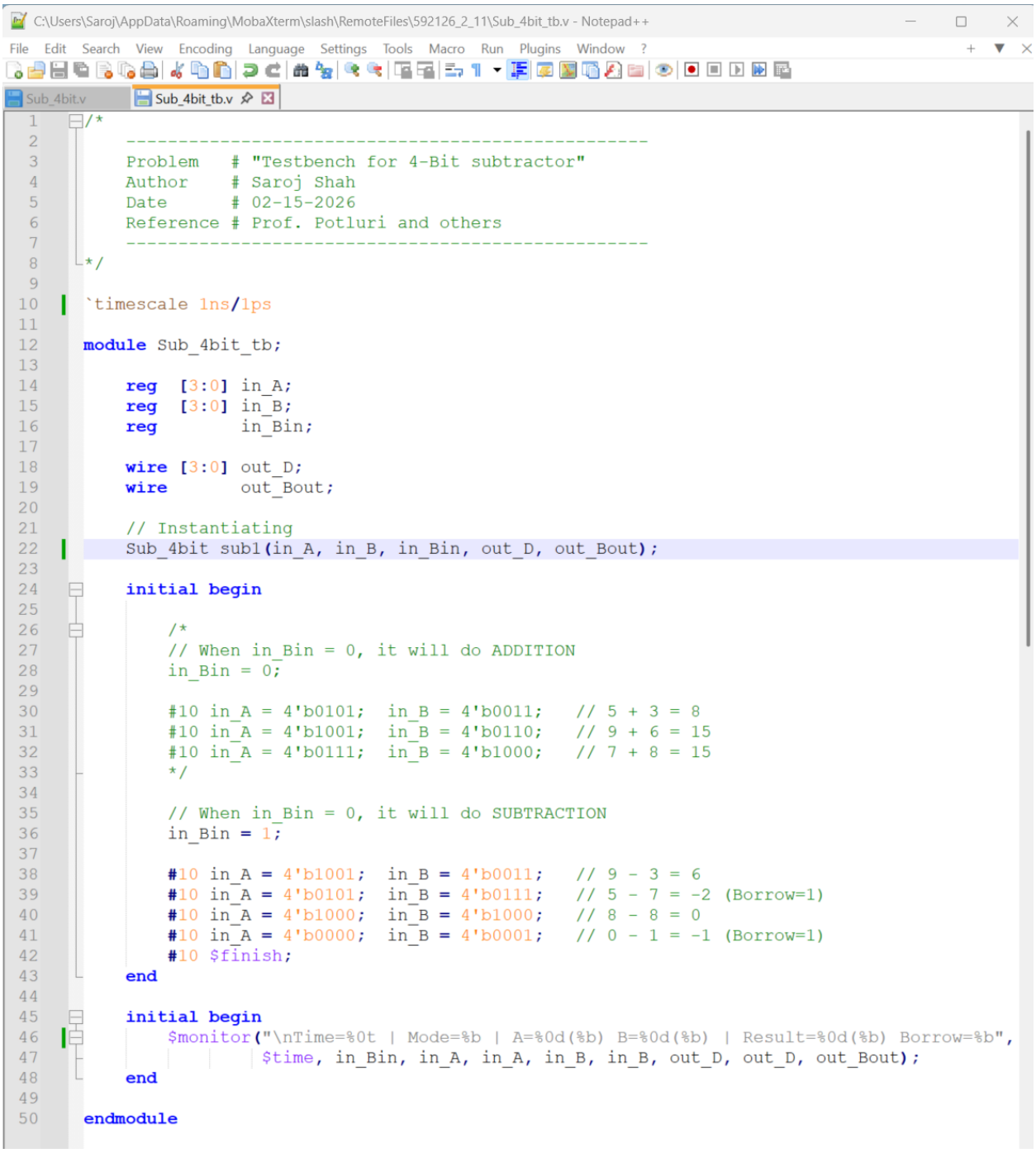


Fig: 8B (Testbench code for 4-Bit Subtractor)

Figure 8A and 8B shows the code for 4-bit subtractor and 4-bit subtractor's testbench. The testbench code is used to running the simulation. The figures are shown below.

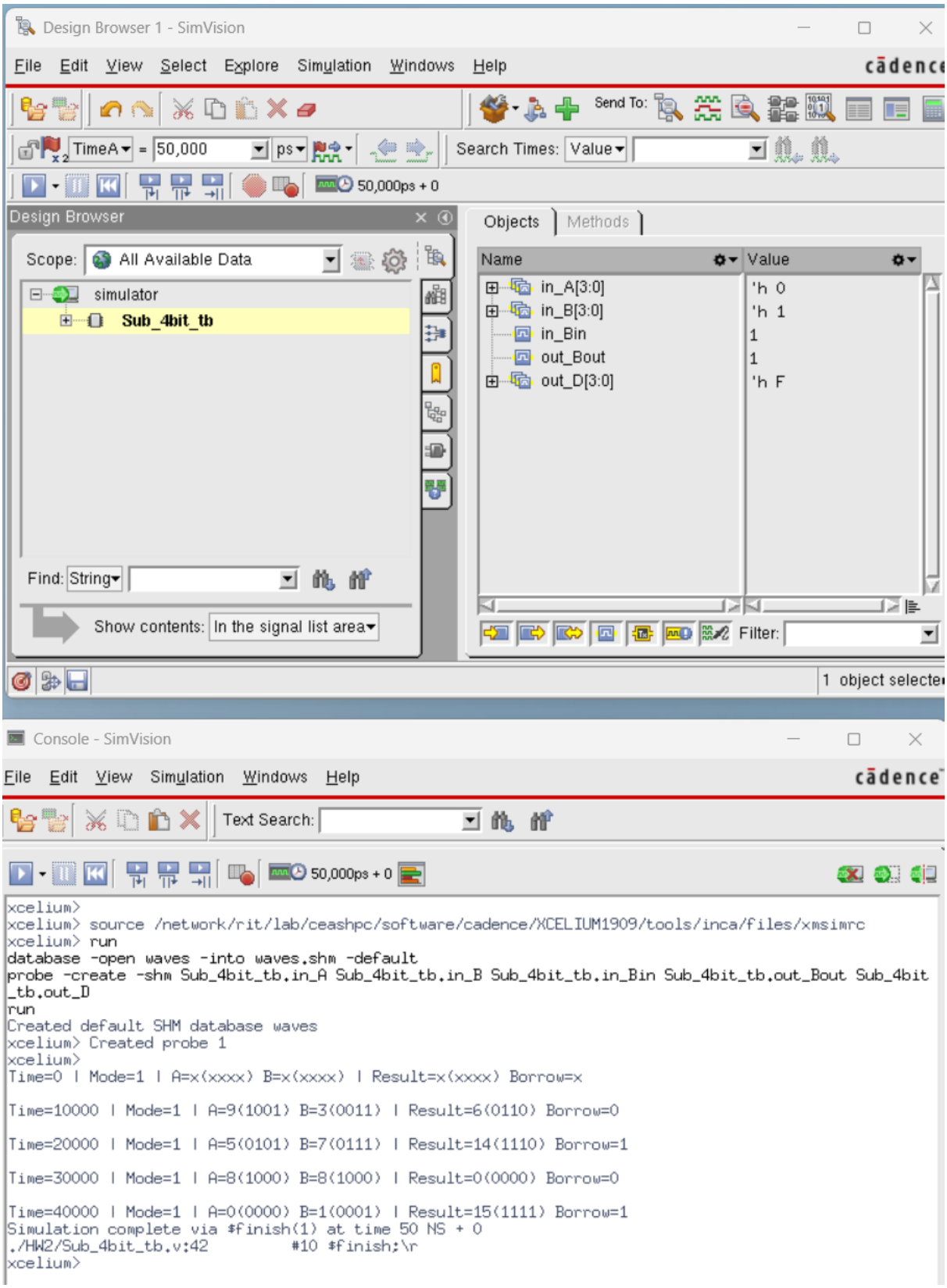


Fig: 9A (Simulation of 4-Bit Subtractor)

```
Time=0 | Mode=1 | A=x(xxxx) B=x(xxxx) | Result=x(xxxx) Borrow=x
Time=10000 | Mode=1 | A=9(1001) B=3(0011) | Result=6(0110) Borrow=0
Time=20000 | Mode=1 | A=5(0101) B=7(0111) | Result=14(1110) Borrow=1
Time=30000 | Mode=1 | A=8(1000) B=8(1000) | Result=0(0000) Borrow=0
Time=40000 | Mode=1 | A=0(0000) B=1(0001) | Result=15(1111) Borrow=1
```

Fig: 9B (Result of 4-Bit Subtractor)

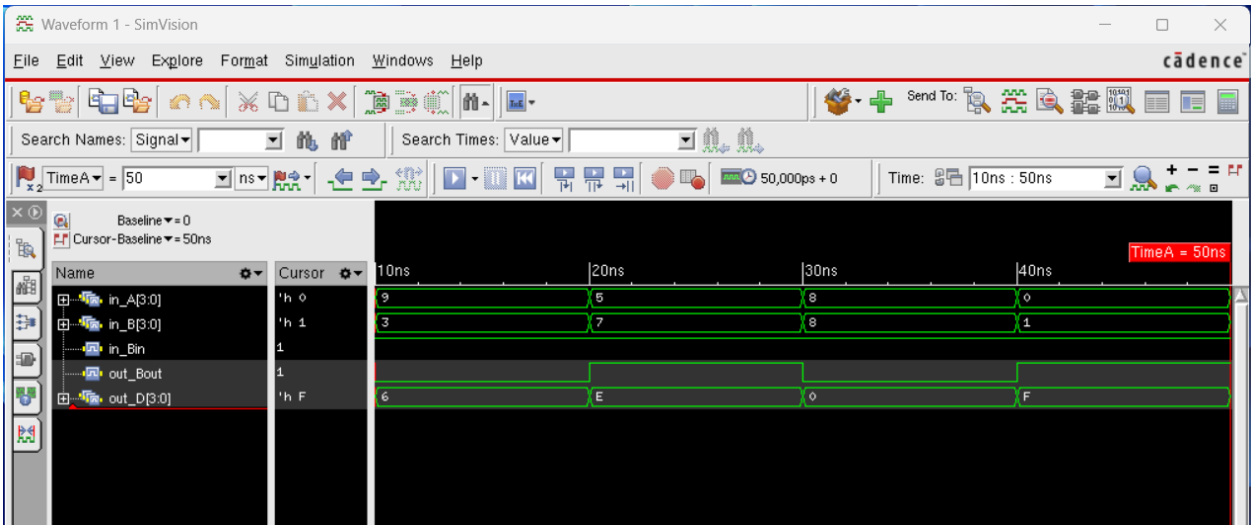


Fig: 9C (Waveform of 4-Bit Subtractor)

Test cases verified:

- $9 - 3 = 6$
- $5 - 7 = -2$ (Borrow = 1)
- $8 - 8 = 0$
- $0 - 1 = -1$ (Borrow = 1)

The simulation figures and waveform shown in figure 9A, 9B and 9C verifies correct operation and result of the Subtractor. The waveforms show correct difference bits and proper borrow generation. The subtractor correctly performs subtraction when Bin = 1.

4. Four-Bit Multiplier (extra credit):

The multiplier was implemented using:

- Partial products generated with AND gates
- Ripple addition using multiple Full Adders
- Column-wise addition of partial products

The output is an 8-bit product:

$$P = A \times B$$

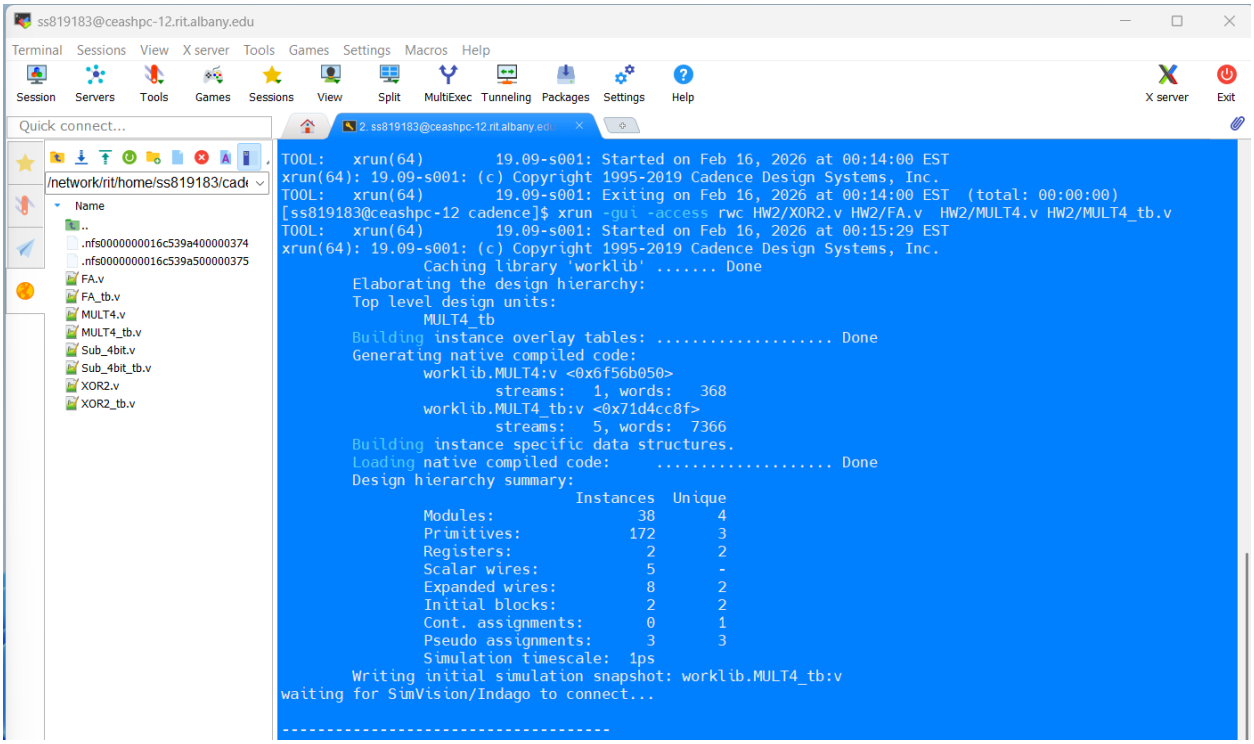
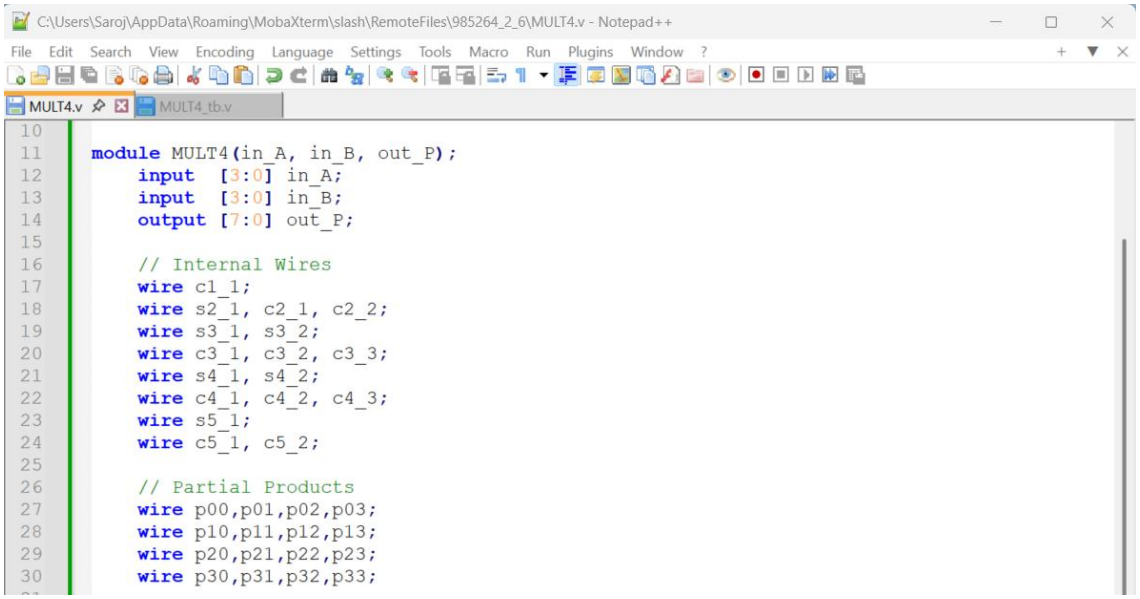


Fig: 10 (MobaXterm window for 4-Bit Multiplier)

The above Figure 4 shows the MobaXterm window for running Four-bit Multiplier




```
10
11  module MULT4(in_A, in_B, out_P);
12      input  [3:0] in_A;
13      input  [3:0] in_B;
14      output [7:0] out_P;
15
16      // Internal Wires
17      wire c1_1;
18      wire s2_1, c2_1, c2_2;
19      wire s3_1, s3_2;
20      wire c3_1, c3_2, c3_3;
21      wire s4_1, s4_2;
22      wire c4_1, c4_2, c4_3;
23      wire s5_1;
24      wire c5_1, c5_2;
25
26      // Partial Products
27      wire p00,p01,p02,p03;
28      wire p10,p11,p12,p13;
29      wire p20,p21,p22,p23;
30      wire p30,p31,p32,p33;
```

```
32      and (p00, in_A[0], in_B[0]);
33      and (p01, in_A[1], in_B[0]);
34      and (p02, in_A[2], in_B[0]);
35      and (p03, in_A[3], in_B[0]);
36
37      and (p10, in_A[0], in_B[1]);
38      and (p11, in_A[1], in_B[1]);
39      and (p12, in_A[2], in_B[1]);
40      and (p13, in_A[3], in_B[1]);
41
42      and (p20, in_A[0], in_B[2]);
43      and (p21, in_A[1], in_B[2]);
44      and (p22, in_A[2], in_B[2]);
45      and (p23, in_A[3], in_B[2]);
46
47      and (p30, in_A[0], in_B[3]);
48      and (p31, in_A[1], in_B[3]);
49      and (p32, in_A[2], in_B[3]);
50      and (p33, in_A[3], in_B[3]);
51
52      // Output bit 0
53      assign out_P[0] = p00;
54
55      // Column 1
56      FA FA1(p01, p10, 1'b0, out_P[1], c1_1);
57
58      // Column 2
59      FA FA2(p02, p11, c1_1, s2_1, c2_1);
60      FA FA3(s2_1, p20, 1'b0, out_P[2], c2_2);
61
62      // Column 3
63      FA FA4(p03, p12, c2_1, s3_1, c3_1);
64      FA FA5(s3_1, p21, c2_2, s3_2, c3_2);
65      FA FA6(s3_2, p30, 1'b0, out_P[3], c3_3);
66
67      // Column 4
68      FA FA7(p13, p22, c3_1, s4_1, c4_1);
69      FA FA8(s4_1, p31, c3_2, s4_2, c4_2);
70      FA FA9(s4_2, c3_3, 1'b0, out_P[4], c4_3);
71
72      // Column 5
73      FA FA10(p23, p32, c4_1, s5_1, c5_1);
74      FA FA11(s5_1, c4_2, c4_3, out_P[5], c5_2);
75
76      // Column 6
77      FA FA12(p33, c5_1, c5_2, out_P[6], out_P[7]);
78
79  endmodule
```

Fig: 11A (Code for 4-Bit Multiplier)

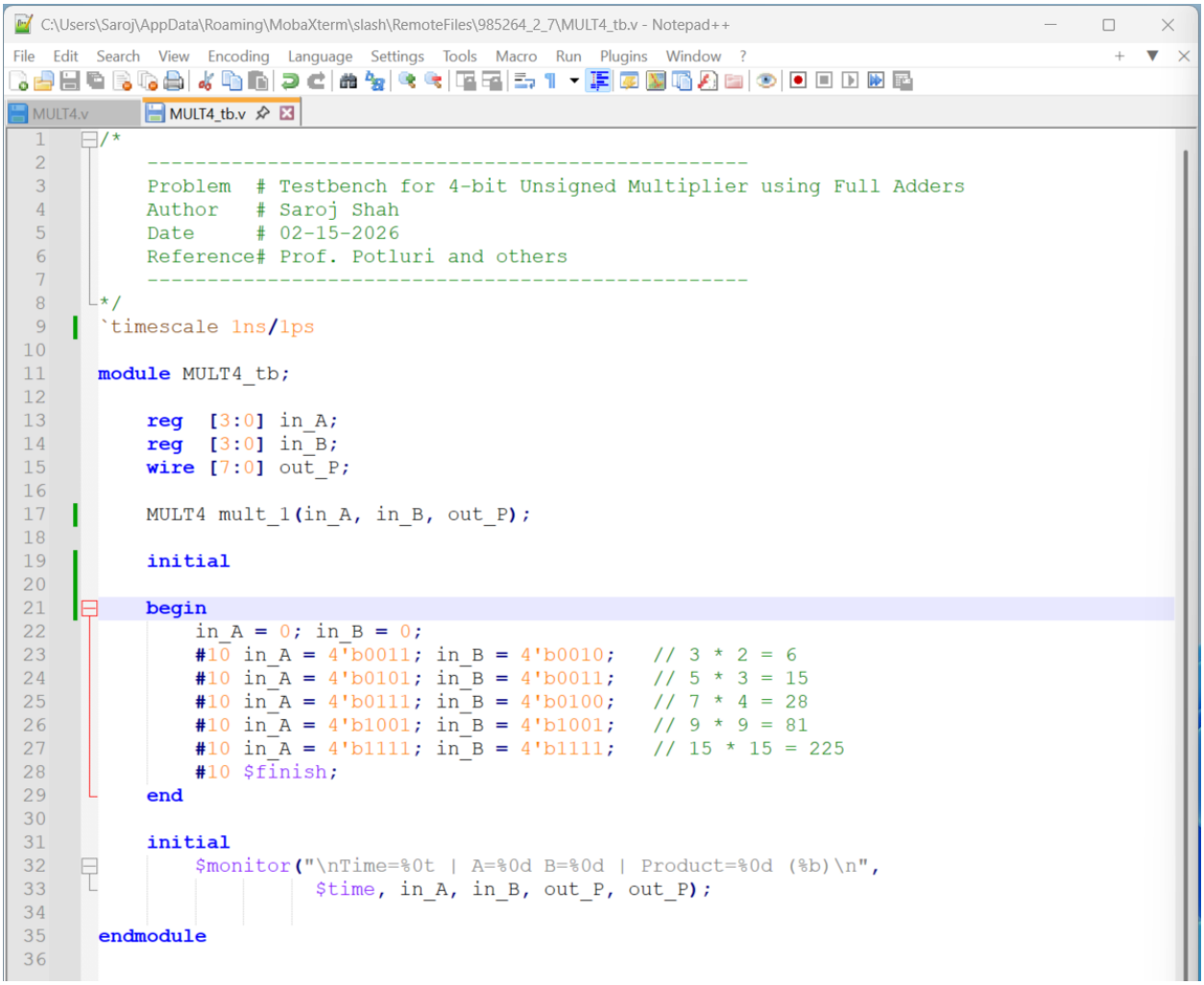


Fig: 11B (Testbench code for 4-Bit Multiplier)

Figure 11A shows the code for multiplier where and 11B shows code for simulation (i.e., its testbench code).

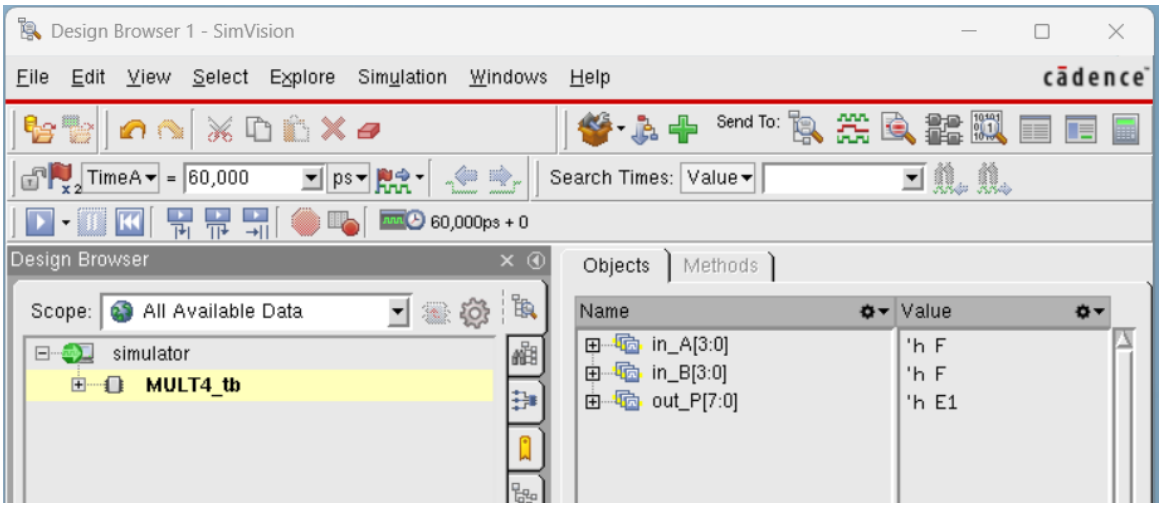
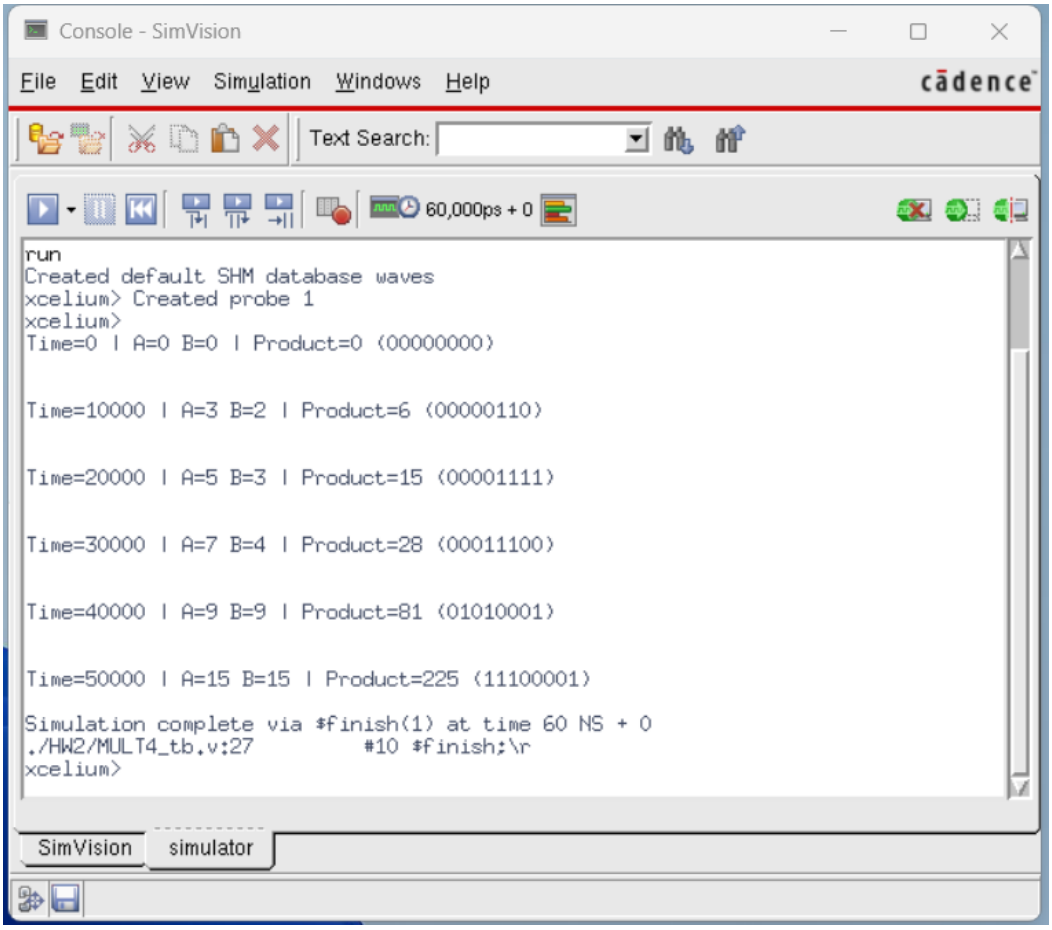


Fig: 12A (Simulation of 4-Bit Multiplier)



```
Time=0 | A=0 B=0 | Product=0 (00000000)

Time=10000 | A=3 B=2 | Product=6 (00000110)

Time=20000 | A=5 B=3 | Product=15 (00001111)

Time=30000 | A=7 B=4 | Product=28 (00011100)

Time=40000 | A=9 B=9 | Product=81 (01010001)

Time=50000 | A=15 B=15 | Product=225 (11100001)
```

Fig: 12B (Result of 4-Bit Multiplier)

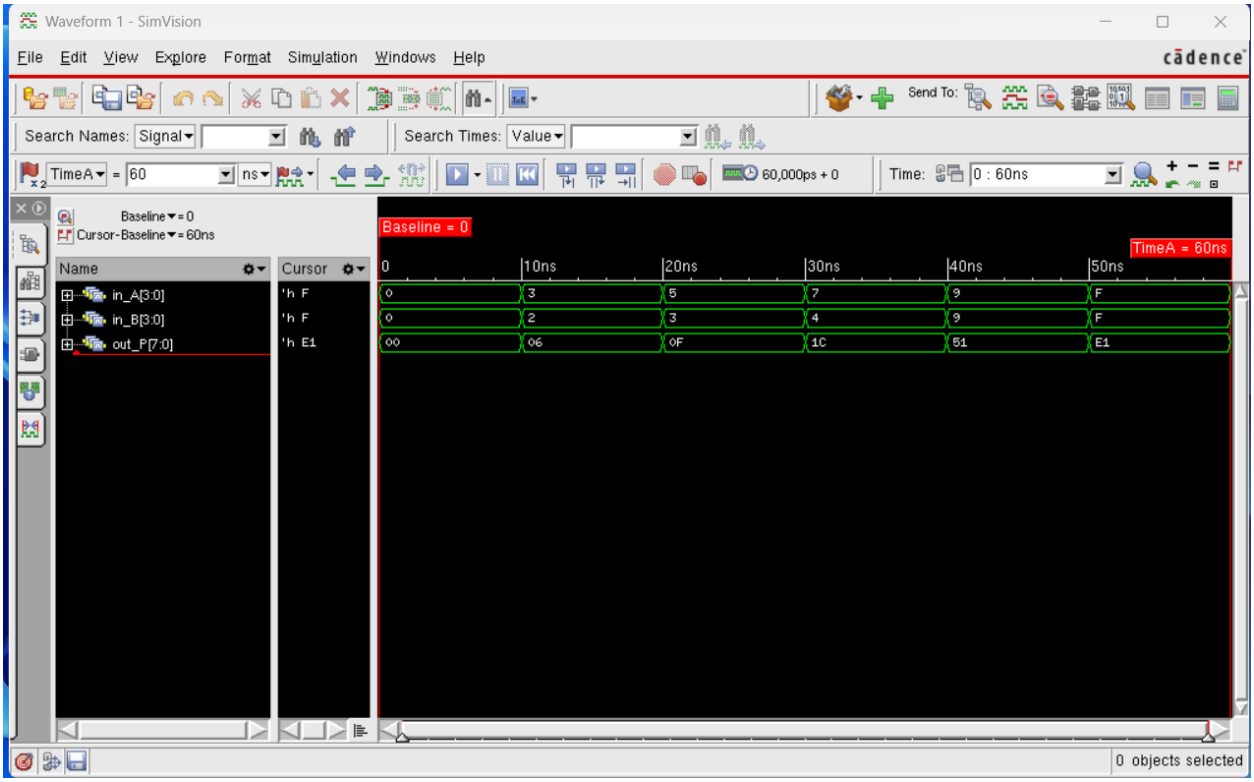


Fig: 12C (Waveform of 4-Bit Multiplier)

Verified test cases:

A	B	Product
3	2	6
5	3	15
7	4	28
9	9	81
15	15	225

The simulation figures and waveform shown in figure 12A, 12B and 12C verifies correct operation and result of the Multiplier. The waveforms also confirm correct bit propagation and addition across all columns. Also note the multiplier correctly generates an 8-bit unsigned result.

Conclusion:

In this assignment, several fundamental combinational logic circuits were successfully designed and simulated using structural Verilog modeling in Cadence Xcelium. The simulation waveforms verified the correct functional behavior for all modules.

Works Cited

Potluri, Seetal. Class Notes, PDF and videos, Spring 2026.