University of California Irvine

EECS 31L: Introduction to Digital Logic Laboratory (Spring 2025)

Lab 1: Logic Block Design

1. Overview

This lab focused on designing and testing digital logic blocks using Verilog: a half adder, a 1-bit full adder, a 4-bit full adder, a 2:1 multiplexer, and a 4:1 multiplexer. Each module was written in its own Verilog file, tested with its own testbench, and simulated to confirm its intended behavior in Vivado.

1.1 Half Adder

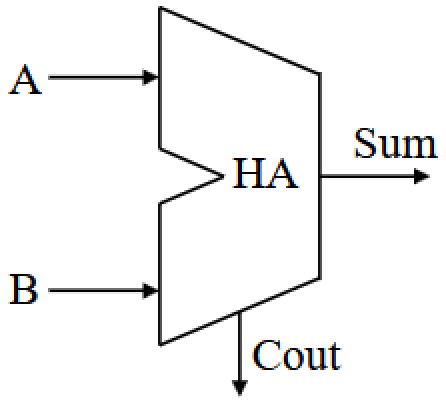


Figure 1.1: Half Adder Block Diagram

A half adder adds two 1-bit binary numbers (A and B) and outputs a 1-bit sum and a 1-bit carry-out.

1.2 1-bit Full Adder

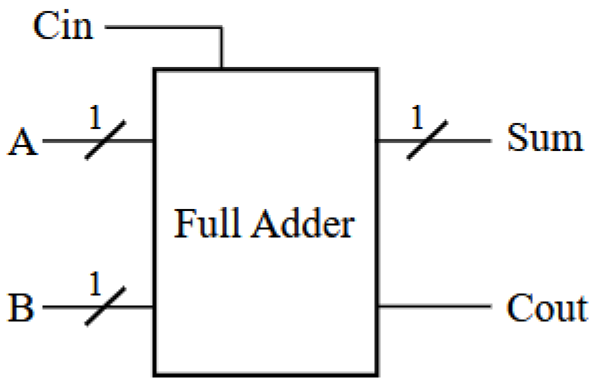


Figure 1.2: 1-bit Full Adder Block Diagram

A full adder is similar to a half adder but includes a third carry-in input. It adds three 1-bit inputs (A, B, and Cin) and outputs a 1-bit sum and carry out.

1.3 4-bit Full Adder

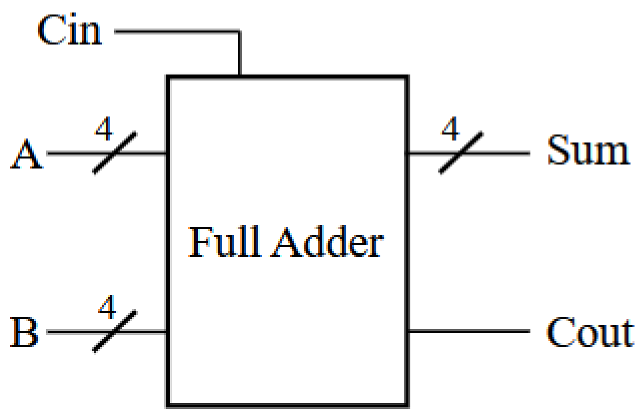


Figure 1.3: 4-bit Full Adder Block Diagram

A 4-bit full adder adds two 4-bit binary numbers and a carry-in. It can be made via a chain of four 1-bit full adders where the carry-out of each connects to the carry-in of the next.

1.4 2:1 Multiplexer

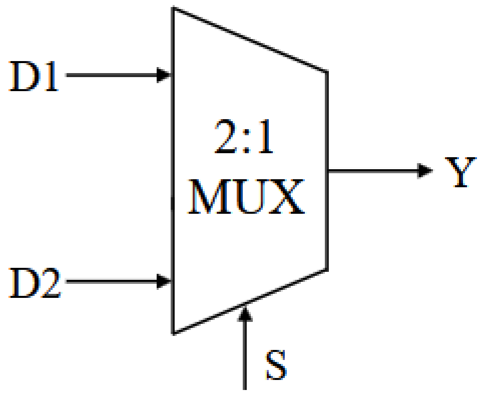


Figure 1.4: 2:1 Multiplexer Block Diagram

A 2:1 multiplexer chooses between two 1-bit inputs (D1 and D2) based on a 1-bit selector S. When S = 0, the output Y = D1 and when S = 1, the output Y = D2.

1.5 4:1 Multiplexer

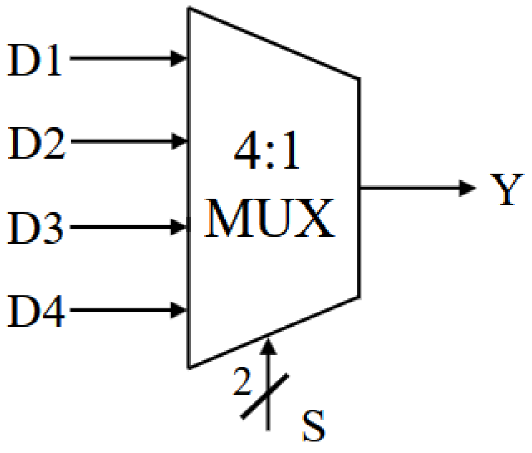


Figure 1.5: 4:1 Multiplexer Block Diagram

A 4:1 multiplexer shows one of four 1-bit inputs (D1, D2, D3, and D4) based on a 2-bit selector S. It routes the selected input to the output Y.

Most of the modules are independent of each other except for the 4-bit full adder. That module was designed by chaining four instances of the 1-bit full adder together. Besides that, the half adder and both multiplexers were standalone designs that were not reused.

2. Hardware Design

2.1 Half Adder

Input		**Output**	
A	**B**	**Sum**	**Cout**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 2.1: Half Adder Truth Table

The half adder adds two 1-bit inputs (A and B) to produce two outputs (Sum and Cout). Its boolean expressions are:

- Sum = A XOR B
- Cout = A AND B

These were translated into Verilog using bitwise operators:

- assign Sum = A ^ B;
- assign Cout = A & B;

2.2 1-bit Full Adder

Input			Output	
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 2.2: 1-bit Full Adder Truth Table

The 1-bit full adder takes three inputs (A, B, and Cin) to produce two outputs (Sum and Cout). Its boolean expressions are:

- Sum = A XOR B XOR Cin
- Cout = (A AND B) OR (A AND Cin) OR (B AND Cin)

These were translated into Verilog using bitwise operators:

- assign Sum = A ^ B ^ Cin;
- assign Cout = (A & B) | (A & Cin) | (B & Cin);

2.3 4-bit Full Adder

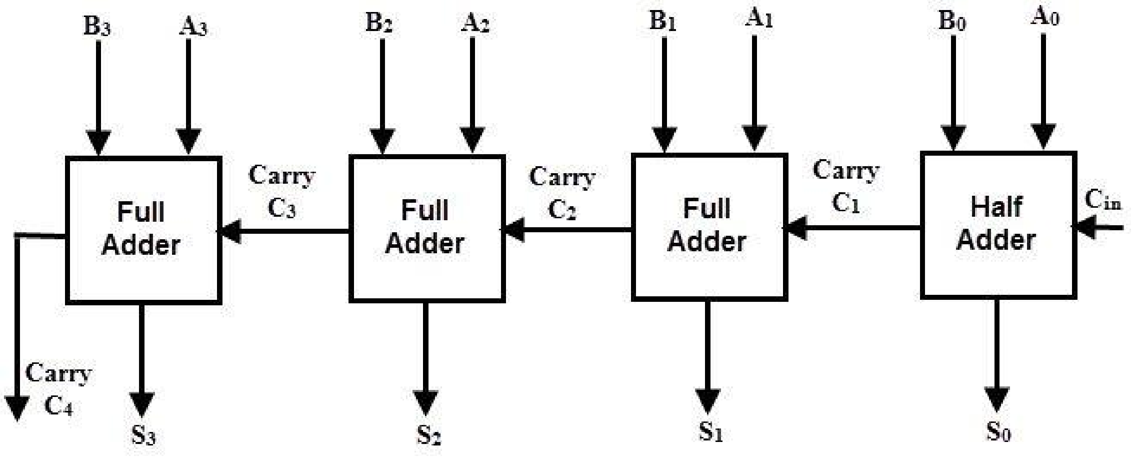
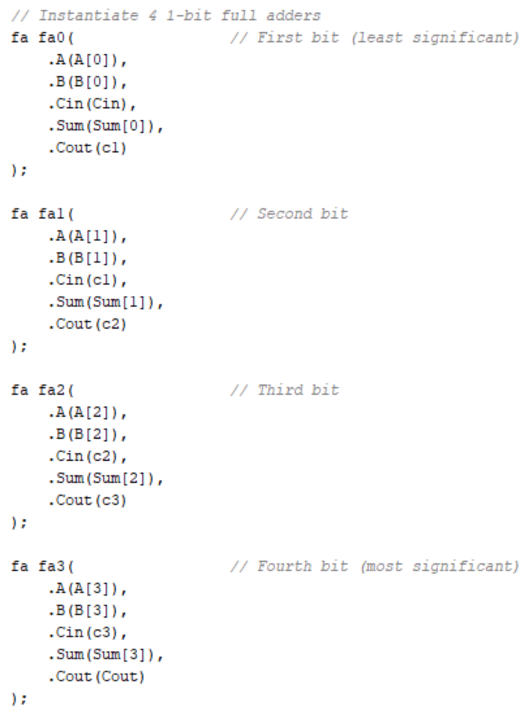


Figure 2.3a: 4-bit Full Adder Block Diagram

Figure 2.3a depicts how the 4-bit full adder is built via a chain of four 1-bit full adders. The carry-out of each adder connects to the carry-in of the next one, which processes all four bits sequentially.



```
// Instantiate 4 1-bit full adders
fa fa0(                 // First bit (least significant)
    .A(A[0]),
    .B(B[0]),
    .Cin(Cin),
    .Sum(Sum[0]),
    .Cout(c1)
);

fa fa1(                 // Second bit
    .A(A[1]),
    .B(B[1]),
    .Cin(c1),
    .Sum(Sum[1]),
    .Cout(c2)
);

fa fa2(                 // Third bit
    .A(A[2]),
    .B(B[2]),
    .Cin(c2),
    .Sum(Sum[2]),
    .Cout(c3)
);

fa fa3(                 // Fourth bit (most significant)
    .A(A[3]),
    .B(B[3]),
    .Cin(c3),
    .Sum(Sum[3]),
    .Cout(Cout)
);
```

Figure 2.3b: Code Snippet of 4-bit Full Adder

Figure 2.3b shows how the full adders are instantiated and connected to one another in Verilog. Each full adder takes a pair of bits from A, B, and a carry-in to produce a sum and carry-out. fa0 handles the least significant bit A[0], B[0], and the initial carry-in. fa1, fa2, and fa3 handle the next three bits with each of them taking the carry-out from the previous adder as their carry-in. The final carry-out becomes the 4-bit full adder's overall carry-out.

2.4 2:1 Multiplexer

The 2:1 multiplexer selects between two 1-bit inputs D1 and D2 based on a 1-bit select input S. Its boolean expression is:

- Y = (NOT(S) AND D1) OR (S AND D2)

This expression means that if S = 0, Y = D1 and if S = 1, Y = D2. This is translated into Verilog using bitwise operators:

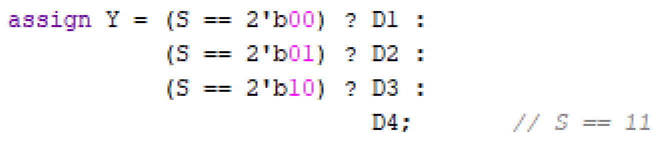
- assign Y = (~S & D1) | (S & D2);

2.5 4:1 Multiplexer

The 4:1 multiplexer selects one of four 1-bit inputs D1, D2, D3, and D4 using a 2-bit selector S. The output Y depends on the binary value of S:

- If S = 00 → Y = D1
- If S = 01 → Y = D2
- If S = 10 → Y = D3
- If S = 11 → Y = D4

This was translated into Verilog using a conditional operator shown in Figure 2.5.



```
assign Y = (S == 2'b00) ? D1 :
           (S == 2'b01) ? D2 :
           (S == 2'b10) ? D3 :
                          D4;       // S == 11
```

Figure 2.5: Code Snippet of 4:1 Multiplexer

This ensures that one input is selected based on the value of S.

3. Simulation Results

3.1 Half Adder

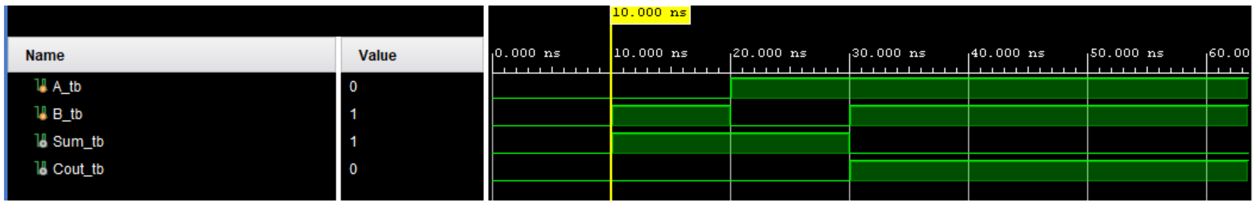


Figure 3.1: Half Adder Waveform

Figure 3.1 shows the simulation results for the half adder. The inputs A_tb and B_tb represent the two 1-bit inputs while Sum_tb and Cout_tb represent the outputs. The goal of this testbench was to verify all possible input combinations of A and B (00, 01, 10, 11) and to confirm that the outputs matched the expected values based on the truth table shown in Figure 2.1.

The testbench applied a new input combination every 10 ns to ensure that each case could be observed in the waveform. The input transitions and its corresponding outputs are summarized below:

- From 0 to 10 ns: A = 0, B = 0 ⟶ Sum = 0, Cout = 0.
- From 10 to 20 ns: A = 0, B = 1 ⟶ Sum = 1, Cout = 0.
- From 20 to 30 ns: A = 1, B = 0 ⟶ Sum = 1, Cout = 0.
- From 30 to 40 ns: A = 1, B = 1 ⟶ Sum = 0, Cout = 1.

Based on the waveform shown in Figure 3.1, it is clear that the outputs responded correctly after the input changes which confirms that the half adder module functioned as expected.

3.2 1-bit Full Adder

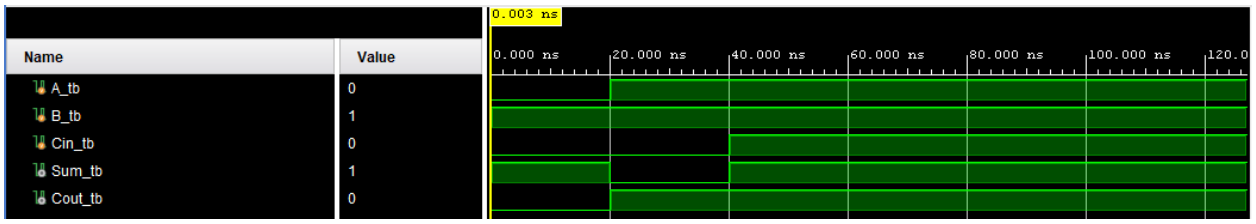


Figure 3.2: 1-bit Full Adder Waveform

Figure 3.2 shows the simulation results for the 1-bit full adder. The inputs A_tb, B_tb, and Cin_tb represent the three 1-bit inputs to the adder while Sum_tb and Cout_tb are the outputs. The goal of this testbench was to simulate three test cases as specified in the lab manual to confirm that the full adder behaves correctly.

The testbench applied a new input combination every 20 ns. The input and output behavior is summarized below:

- From 0 to 20 ns: A = 0, B = 1, Cin = 0 → Sum = 1, Cout = 0.
- From 20 to 40 ns: A = 1, B = 1, Cin = 0 → Sum = 0, Cout = 1.
- From 40 to 60 ns: A = 1, B = 1, Cin = 1 → Sum = 1, Cout = 1.

Based on the waveform shown in Figure 3.2, it is clear that the outputs responded correctly as the inputs changed and matched the expected results from the truth table in Figure 2.2.

3.3 4-bit Full Adder

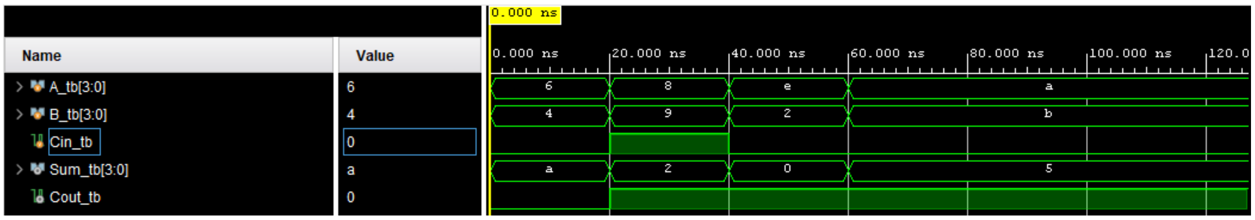


Figure 3.3: 4-bit Full Adder Waveform

Figure 3.3 shows the simulation results for the 4-bit full adder. This module involved four 1-bit full adders whose carry-out values cascading from one adder to the next. The goal of this testbench was to confirm that the design correctly performs 4-bit binary addition with carry propagation.

The inputs A_tb[3:0] and B_tb[3:0] are the two 4-bit binary numbers being added while Cin_tb is the initial carry-in. The output Sum_tb[3:0] is the resulting 4-bit sum and Cout_tb is the final carry-out. The testbench applied a new case every 20 ns using the test values specified in the lab manual. The behavior is summarized below:

- From 0 to 20 ns: A = 0110 (6), B = 0100 (4), Cin = 0 → Sum = 1010 (A), Cout = 0.
- From 20 to 40 ns: A = 1000 (8), B = 1001 (9), Cin = 1 → Sum = 0010 (2), Cout = 1.
- From 40 to 60 ns: A = 1110 (E), B = 0010 (2), Cin = 0 → Sum = 0000 (0), Cout = 1.
- From 60 to 80 ns: A = 1010 (A), B = 1011 (B), Cin = 0 → Sum = 0101 (5), Cout = 1.

The outputs match the expected results from the binary addition performed. The waveform confirms that the module properly handles addition with overflow.

3.4 2:1 Multiplexer

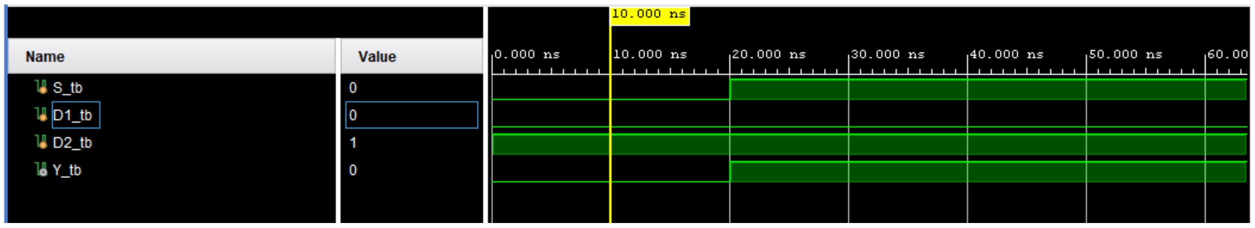


Figure 3.4: 2:1 Multiplexer Waveform

Figure 3.4 shows the simulation results for the 2:1 multiplexer. The inputs D1_tb and D2_tb are the two data inputs, S_tb is the selector signal, and Y_tb is the output. The goal of this testbench was to verify that the output reflects the value of the selector input based on the value of S_tb.

Two test cases were run with different values of S_tb while D1_tb and D2_tb remained as 0 and 1 respectively. The behavior is summarized below:

- From 0 to 20 ns: D1 = 0, D2 = 1, S = 0 → Y = D1 = 0.
- From 20 to 40 ns: D1 = 0, D2 = 1, S = 1 → Y = D2 = 1.

In both cases, the output Y matched the expected values based on the logic of a 2:1 multiplexer.

3.5 4:1 Multiplexer

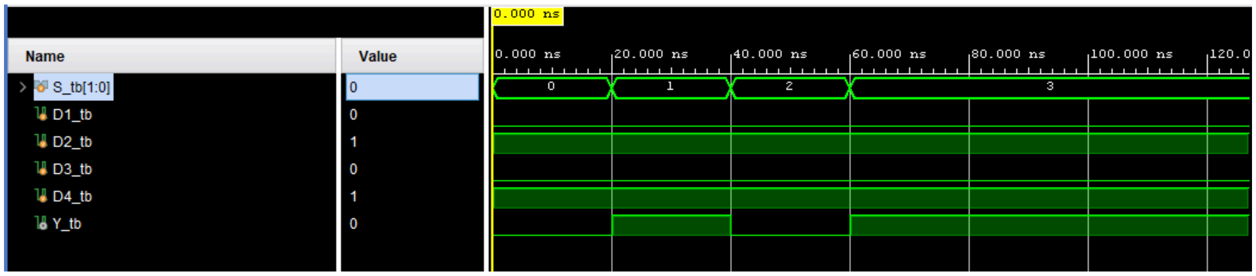


Figure 3.5: 4:1 Multiplexer Waveform

Figure 3.5 shows the simulation results for the 4:1 multiplexer. The data inputs D1_tb through D4_tb were fixed during the simulation while the selector input S_tb[1:0] was varied to test all four cases. The output Y_tb was expected to be the value of the selected input based on the 2-bit selector value.

The select input was designed to update every 20 ns. The behavior is summarized below:

- From 0 to 20 ns: S = 00 → Y = D1 = 0.
- From 20 to 40 ns: S = 01 → Y = D2 = 1.
- From 40 to 60 ns: S = 10 → Y = D3 = 0.
- From 60 to 80 ns: S = 11 → Y = D4 = 1.

These results confirm that the 4:1 multiplexer properly routed the selected input to the output in each case. The output Y_tb responded immediately following each change in the select signal during the simulation.