CHITTAGONG UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Electronics And Telecommunication Engineering

Lab Report

Experiment Name: Introduction to Verilog Testbenches and Functional Verification

Experiment No.:10

Course Title: VLSI Technology Sessional

Course No.: ETE 404

Date of Experiment: 20-10-2024

Date of Submission: 27-10-2024

Submitted By	Submitted To
Name: M.I. Yasir Arafat **ID:** 1908031 **Level:** 4 **Term:** I	**Arif Istiaque Rupom** Assistant Professor, Dept. of ETE,CUET

Objectives:

1. To familiarize with Model-sim.
2. To familiarize with writing test-benches.
3. To verify combinational circuits imposing test vectors.

Required Software:

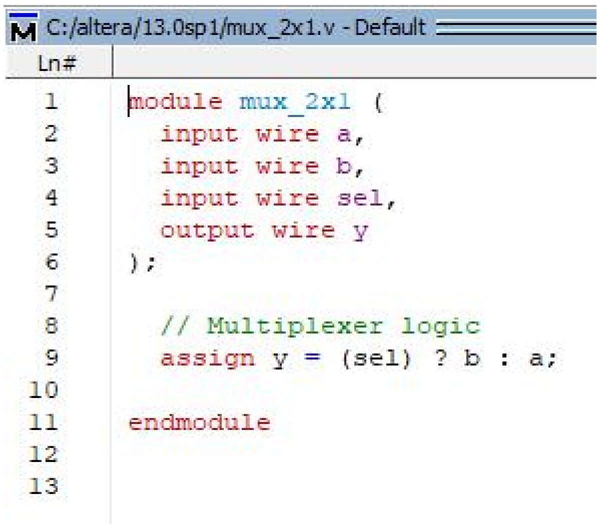
1. Model-sim-Altera

Design Process:

The test-bench codes written in Hardware Description Language (HDL) are run in the Model-sim-Altera software to evaluate the behaviour of Design Under Test (DUT). In this process, four of the components are simulated to check the behaviour with their test benches, which are derived from there truth table. The four components are listed below with their HDL code, test-bench code, simulation waveform, and the transcripts for matching the output.

1. 2X1 Multiplexer

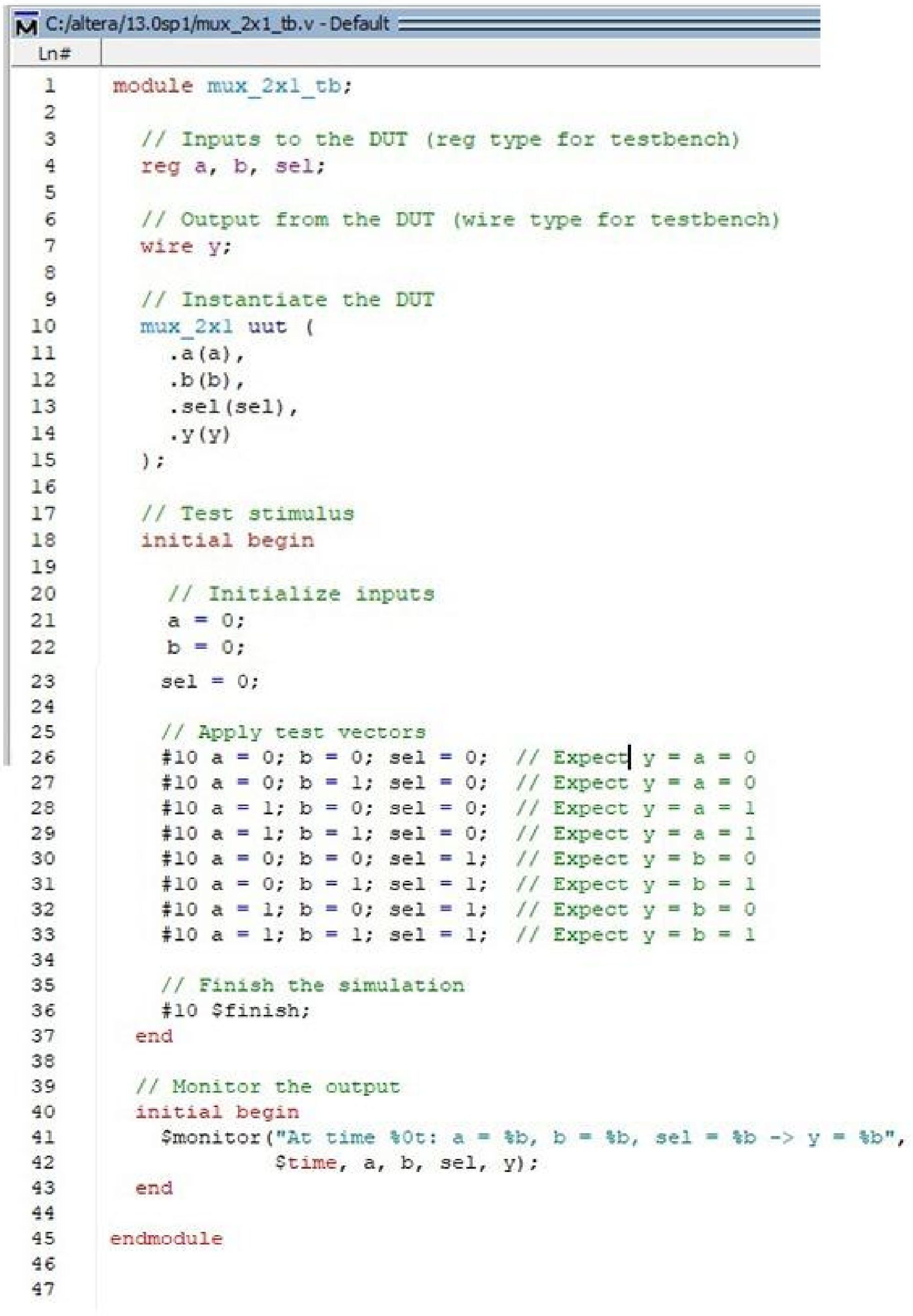
After creating the new file name as ‘mux_2x1’ the code editor window has been opened. The below code has been written in the editor.



```
module mux_2x1 (
  input wire a,
  input wire b,
  input wire sel,
  output wire y
);

  // Multiplexer logic
  assign y = (sel) ? b : a;

endmodule
```

Fig. 1

Then another new file ‘mux_2x1_tb’ has been opened from the project panel to write the test-benches which is shown below.

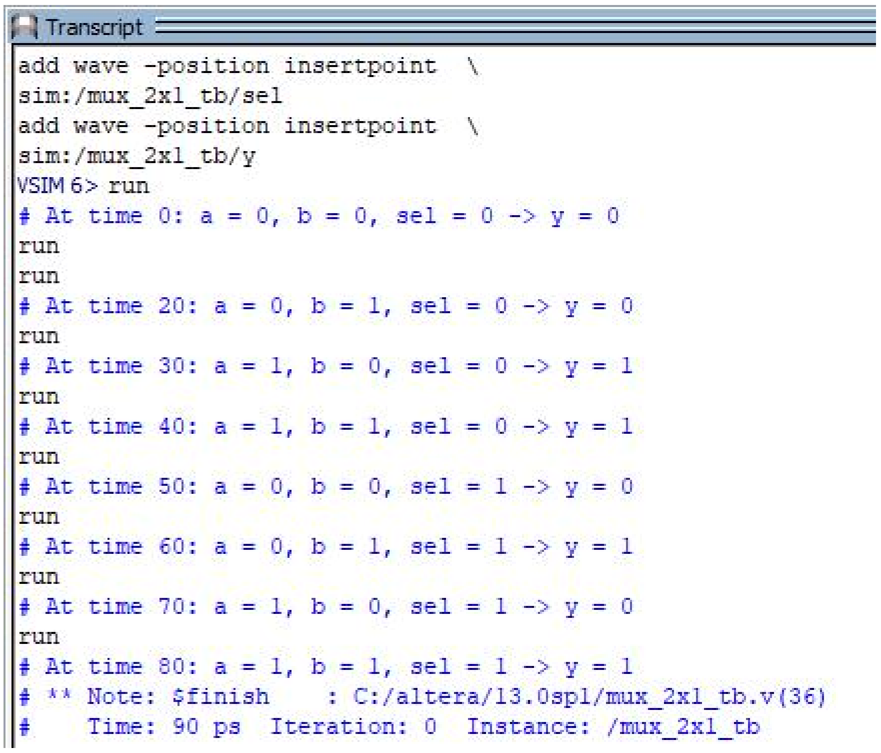


C:/altera/13.0sp1/mux_2x1_tb.v - Default

```
module mux_2x1_tb;

  // Inputs to the DUT (reg type for testbench)
  reg a, b, sel;

  // Output from the DUT (wire type for testbench)
  wire y;

  // Instantiate the DUT
  mux_2x1 uut (
    .a(a),
    .b(b),
    .sel(sel),
    .y(y)
  );

  // Test stimulus
  initial begin

    // Initialize inputs
    a = 0;
    b = 0;
    sel = 0;

    // Apply test vectors
    #10 a = 0; b = 0; sel = 0;  // Expect y = a = 0
    #10 a = 0; b = 1; sel = 0;  // Expect y = a = 0
    #10 a = 1; b = 0; sel = 0;  // Expect y = a = 1
    #10 a = 1; b = 1; sel = 0;  // Expect y = a = 1
    #10 a = 0; b = 0; sel = 1;  // Expect y = b = 0
    #10 a = 0; b = 1; sel = 1;  // Expect y = b = 1
    #10 a = 1; b = 0; sel = 1;  // Expect y = b = 0
    #10 a = 1; b = 1; sel = 1;  // Expect y = b = 1

    // Finish the simulation
    #10 $finish;
  end

  // Monitor the output
  initial begin
    $monitor("At time %0t: a = %b, b = %b, sel = %b -> y = %b",
             $time, a, b, sel, y);
  end

endmodule
```

Fig. 2:

When the compile all was pressed then the code was compiled, and then we could see the simulation from the 'start simulation'. Moreover, we executed the Wave function to visualize the waveform generated from the HDL code. Then, adding the whole Objects to the wave window we got the outcome. The transcript with the proper test benches is shown below in Figure 3.



```
Transcript
add wave -position insertpoint  \
sim:/mux_2x1_tb/sel
add wave -position insertpoint  \
sim:/mux_2x1_tb/y
VSIM 6> run
# At time 0: a = 0, b = 0, sel = 0 -> y = 0
run
run
# At time 20: a = 0, b = 1, sel = 0 -> y = 0
run
# At time 30: a = 1, b = 0, sel = 0 -> y = 1
run
# At time 40: a = 1, b = 1, sel = 0 -> y = 1
run
# At time 50: a = 0, b = 0, sel = 1 -> y = 0
run
# At time 60: a = 0, b = 1, sel = 1 -> y = 1
run
# At time 70: a = 1, b = 0, sel = 1 -> y = 0
run
# At time 80: a = 1, b = 1, sel = 1 -> y = 1
# ** Note: $finish    : C:/altera/13.0sp1/mux_2x1_tb.v(36)
#    Time: 90 ps  Iteration: 0  Instance: /mux_2x1_tb
```

Fig. 3

Before getting the proper waveform, the timing was set to 10ps. The waveform we got is shown below in figure 4.

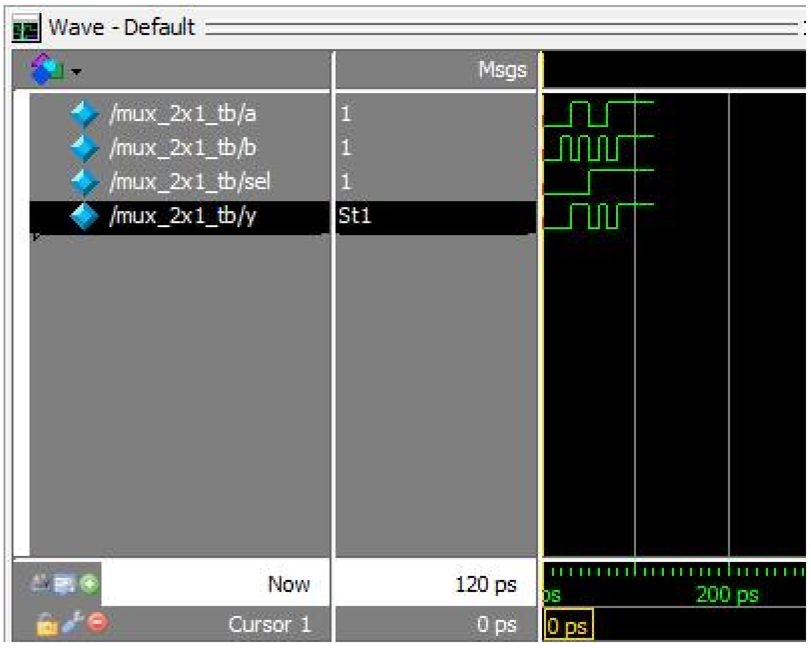


Fig.4

To get the view of the wave differently, such as with 0s and 1s, the Literal format has been executed. The binary format waveform is shown in figure 5.

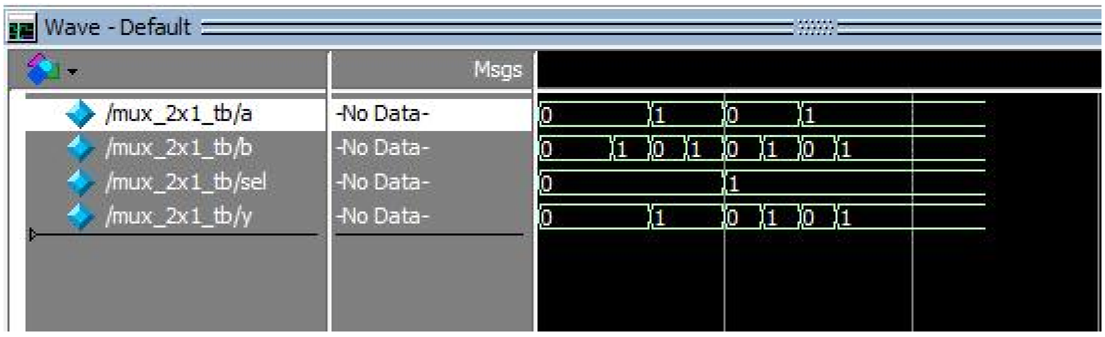
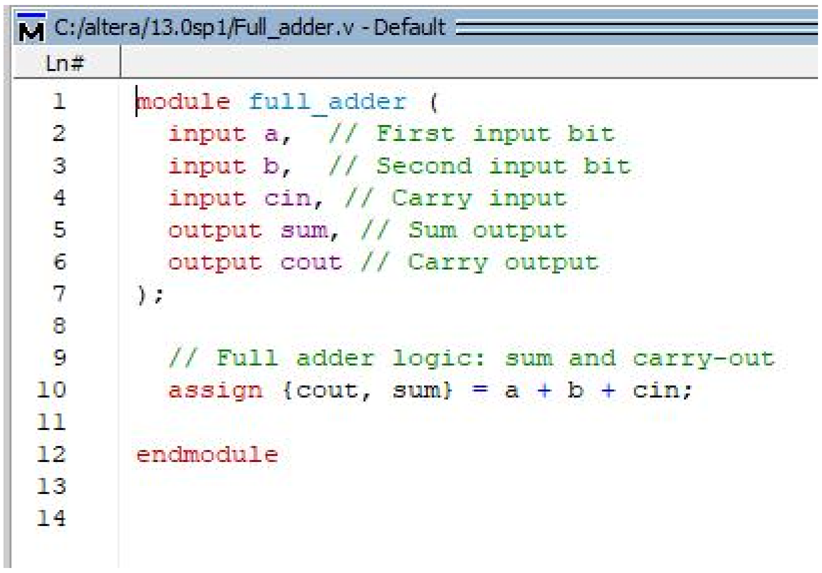


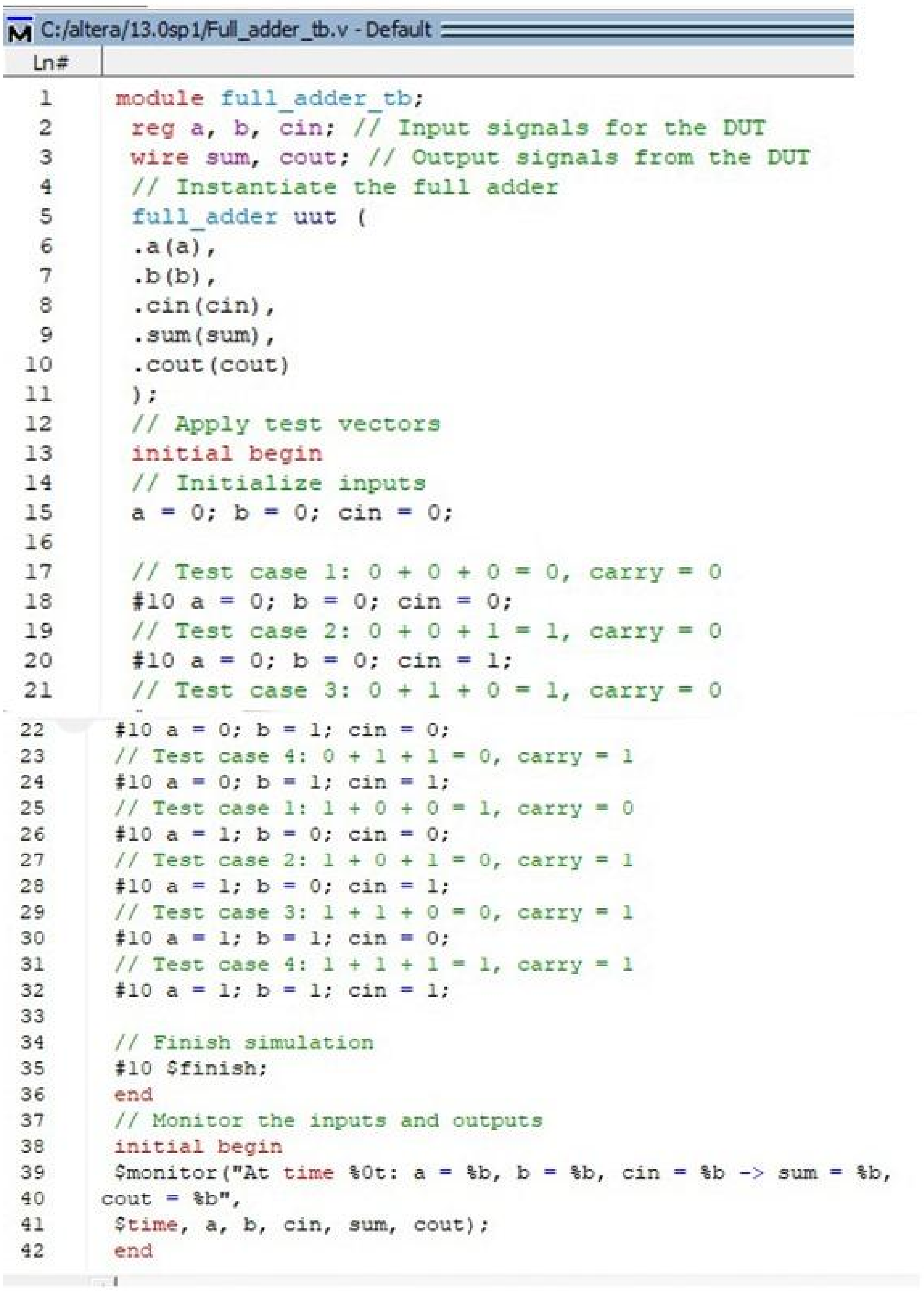
Fig. 5

2. Full Adder

A new file named 'full_adder' was added to the working file panel to write the HDL code of full adder and then visualize the waveform created from the code. The code for Full Adder code is shown below in figure 6.



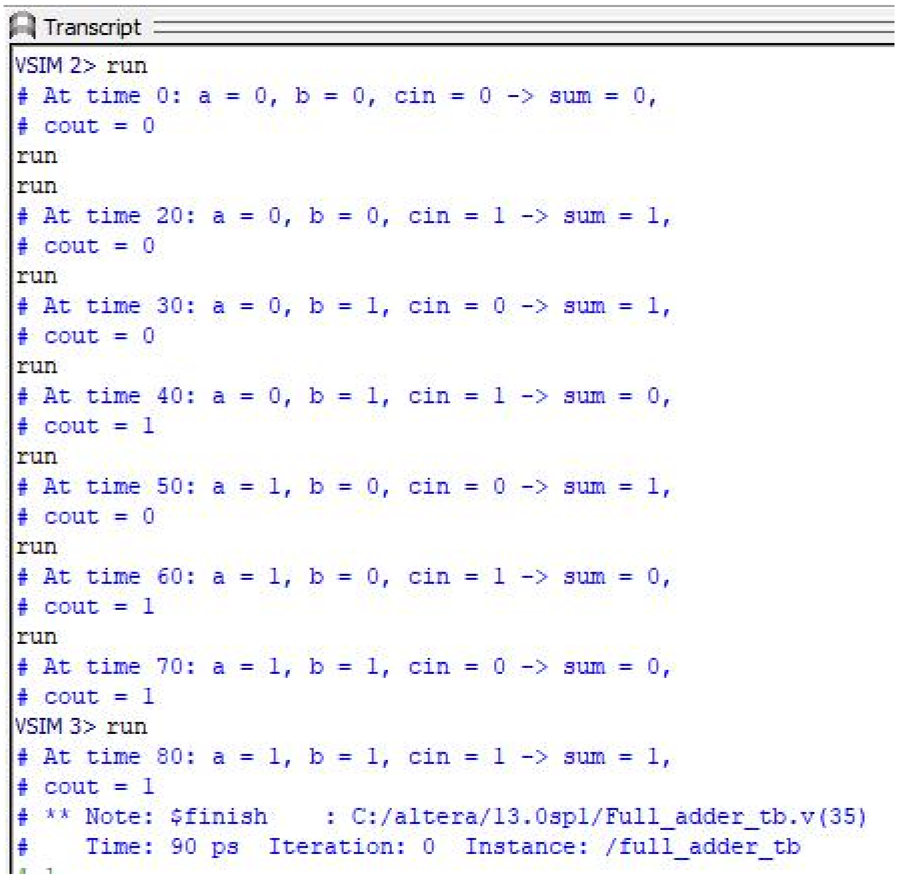
```
module full_adder (
  input a,   // First input bit
  input b,   // Second input bit
  input cin, // Carry input
  output sum, // Sum output
  output cout // Carry output
);

  // Full adder logic: sum and carry-out
  assign {cout, sum} = a + b + cin;

endmodule
```



```
module full_adder_tb;
 reg a, b, cin; // Input signals for the DUT
 wire sum, cout; // Output signals from the DUT
 // Instantiate the full adder
 full_adder uut (
 .a(a),
 .b(b),
 .cin(cin),
 .sum(sum),
 .cout(cout)
 );
 // Apply test vectors
 initial begin
 // Initialize inputs
 a = 0; b = 0; cin = 0;

 // Test case 1: 0 + 0 + 0 = 0, carry = 0
 #10 a = 0; b = 0; cin = 0;
 // Test case 2: 0 + 0 + 1 = 1, carry = 0
 #10 a = 0; b = 0; cin = 1;
 // Test case 3: 0 + 1 + 0 = 1, carry = 0
 #10 a = 0; b = 1; cin = 0;
 // Test case 4: 0 + 1 + 1 = 0, carry = 1
 #10 a = 0; b = 1; cin = 1;
 // Test case 1: 1 + 0 + 0 = 1, carry = 0
 #10 a = 1; b = 0; cin = 0;
 // Test case 2: 1 + 0 + 1 = 0, carry = 1
 #10 a = 1; b = 0; cin = 1;
 // Test case 3: 1 + 1 + 0 = 0, carry = 1
 #10 a = 1; b = 1; cin = 0;
 // Test case 4: 1 + 1 + 1 = 1, carry = 1
 #10 a = 1; b = 1; cin = 1;

 // Finish simulation
 #10 $finish;
 end
 // Monitor the inputs and outputs
 initial begin
 $monitor("At time %0t: a = %b, b = %b, cin = %b -> sum = %b,
cout = %b",
 $time, a, b, cin, sum, cout);
 end
```

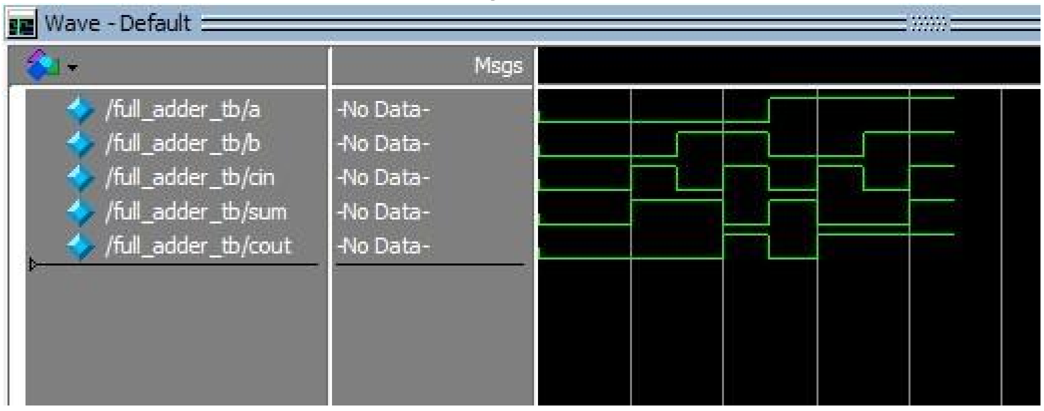
Fig. 6

The testbench code was also run, and then the simulation was done as previously. The waveform we got was matching the transcript with the testbenches. Figure 7 shows the transcript and then the figure 8 shows the waveform which is matching the transcript.



```
Transcript
VSIM 2> run
# At time 0: a = 0, b = 0, cin = 0 -> sum = 0,
# cout = 0
run
run
# At time 20: a = 0, b = 0, cin = 1 -> sum = 1,
# cout = 0
run
# At time 30: a = 0, b = 1, cin = 0 -> sum = 1,
# cout = 0
run
# At time 40: a = 0, b = 1, cin = 1 -> sum = 0,
# cout = 1
run
# At time 50: a = 1, b = 0, cin = 0 -> sum = 1,
# cout = 0
run
# At time 60: a = 1, b = 0, cin = 1 -> sum = 0,
# cout = 1
run
# At time 70: a = 1, b = 1, cin = 0 -> sum = 0,
# cout = 1
VSIM 3> run
# At time 80: a = 1, b = 1, cin = 1 -> sum = 1,
# cout = 1
# ** Note: $finish    : C:/altera/13.0sp1/Full_adder_tb.v(35)
#    Time: 90 ps  Iteration: 0  Instance: /full_adder_tb
```

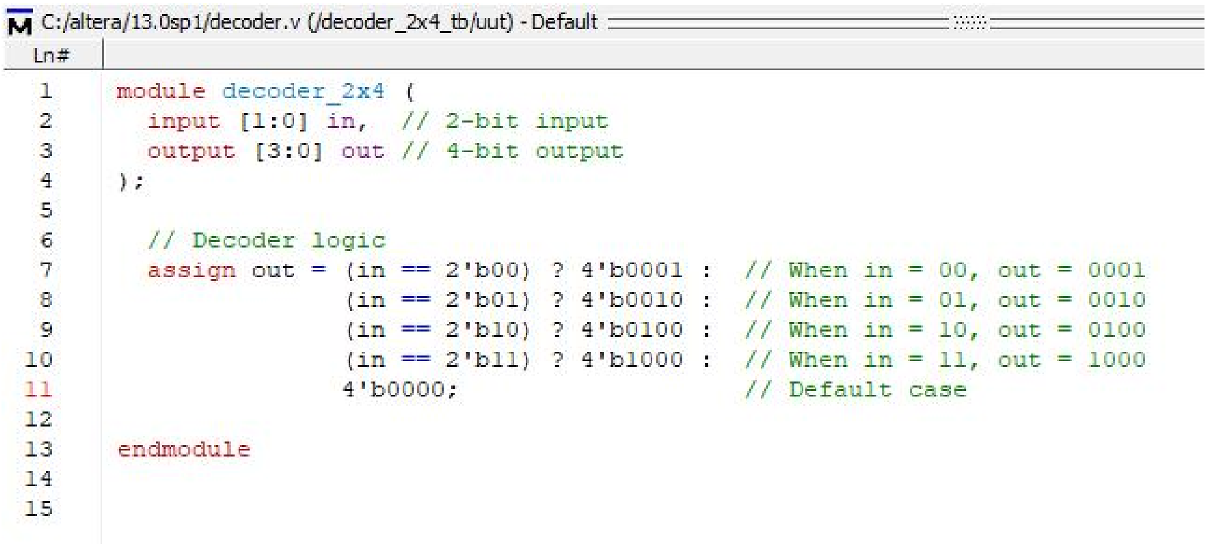
Fig, 7:



Fig, 8

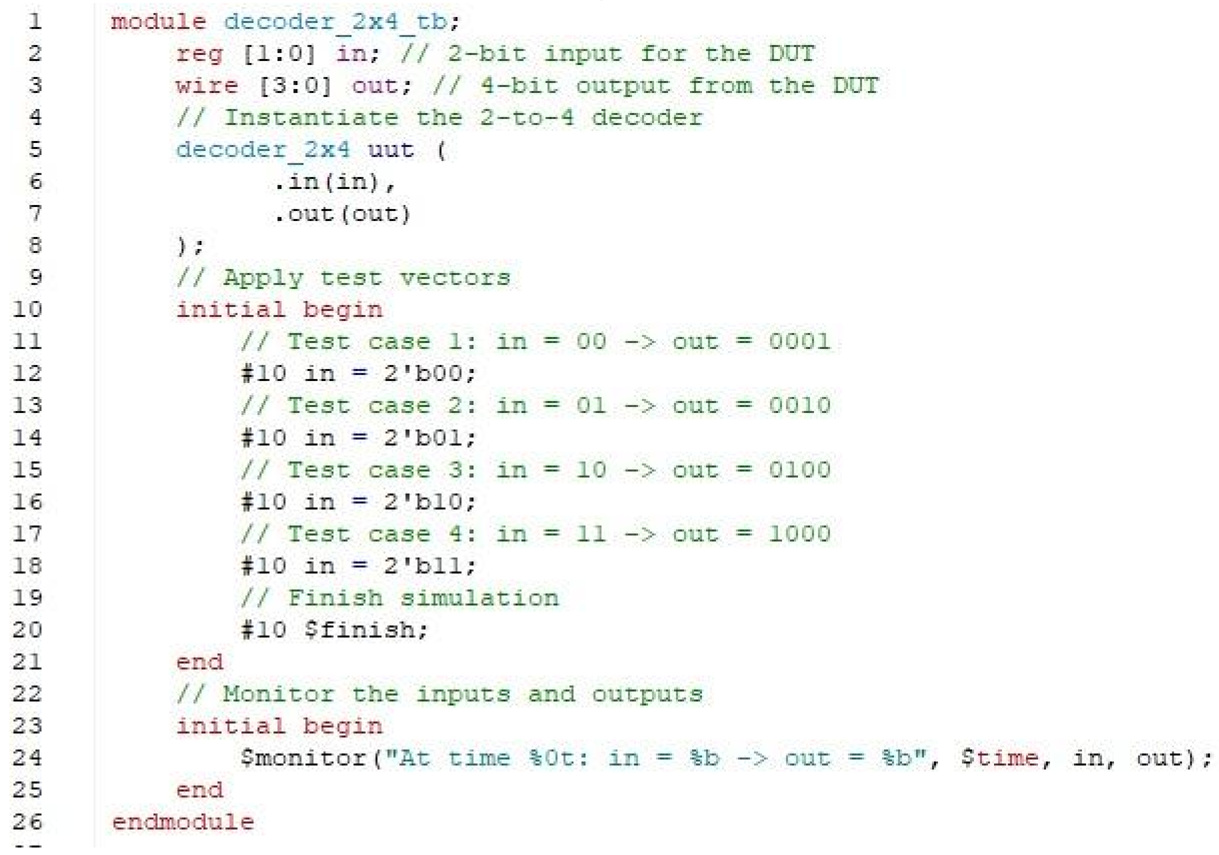
3. 2X4 Decoder

We also worked with the 2x4 decoder HDL to verify the functionality of the code and the output as waveform with the given testbenches. The verilog code is given below in figure 9 along with the testbenches in figure 10.



```
C:/altera/13.0sp1/decoder.v (/decoder_2x4_tb/uut) - Default
Ln#
module decoder_2x4 (
  input [1:0] in,  // 2-bit input
  output [3:0] out // 4-bit output
);

  // Decoder logic
  assign out = (in == 2'b00) ? 4'b0001 :  // When in = 00, out = 0001
               (in == 2'b01) ? 4'b0010 :  // When in = 01, out = 0010
               (in == 2'b10) ? 4'b0100 :  // When in = 10, out = 0100
               (in == 2'b11) ? 4'b1000 :  // When in = 11, out = 1000
               4'b0000;                   // Default case

endmodule
```

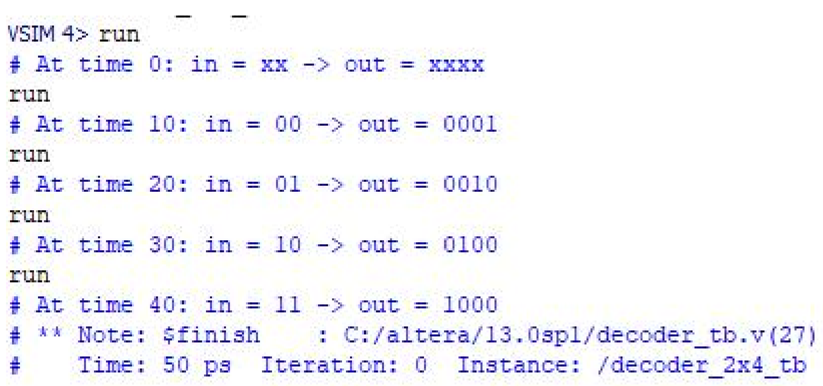
Fig. 9



```
module decoder_2x4_tb;
    reg [1:0] in; // 2-bit input for the DUT
    wire [3:0] out; // 4-bit output from the DUT
    // Instantiate the 2-to-4 decoder
    decoder_2x4 uut (
          .in(in),
          .out(out)
    );
    // Apply test vectors
    initial begin
        // Test case 1: in = 00 -> out = 0001
        #10 in = 2'b00;
        // Test case 2: in = 01 -> out = 0010
        #10 in = 2'b01;
        // Test case 3: in = 10 -> out = 0100
        #10 in = 2'b10;
        // Test case 4: in = 11 -> out = 1000
        #10 in = 2'b11;
        // Finish simulation
        #10 $finish;
    end
    // Monitor the inputs and outputs
    initial begin
        $monitor("At time %0t: in = %b -> out = %b", $time, in, out);
    end
endmodule
```

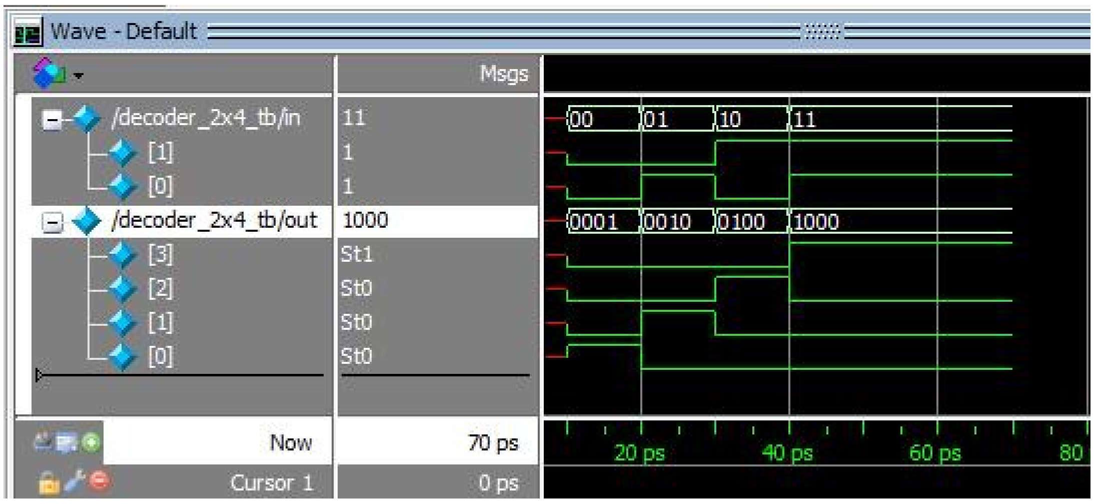
Fig. 10

After running the HDL code, the simulation part was done. The simulation of the decoder output waveform perfectly matched the transcript. Both are shown below in figure 11 and 12. The binary literal waveform has been showed in the figure 12 for better understanding.



```
VSIM 4> run
# At time 0: in = xx -> out = xxxx
run
# At time 10: in = 00 -> out = 0001
run
# At time 20: in = 01 -> out = 0010
run
# At time 30: in = 10 -> out = 0100
run
# At time 40: in = 11 -> out = 1000
# ** Note: $finish    : C:/altera/13.0sp1/decoder_tb.v(27)
#    Time: 50 ps  Iteration: 0  Instance: /decoder_2x4_tb
```

Fig. 11

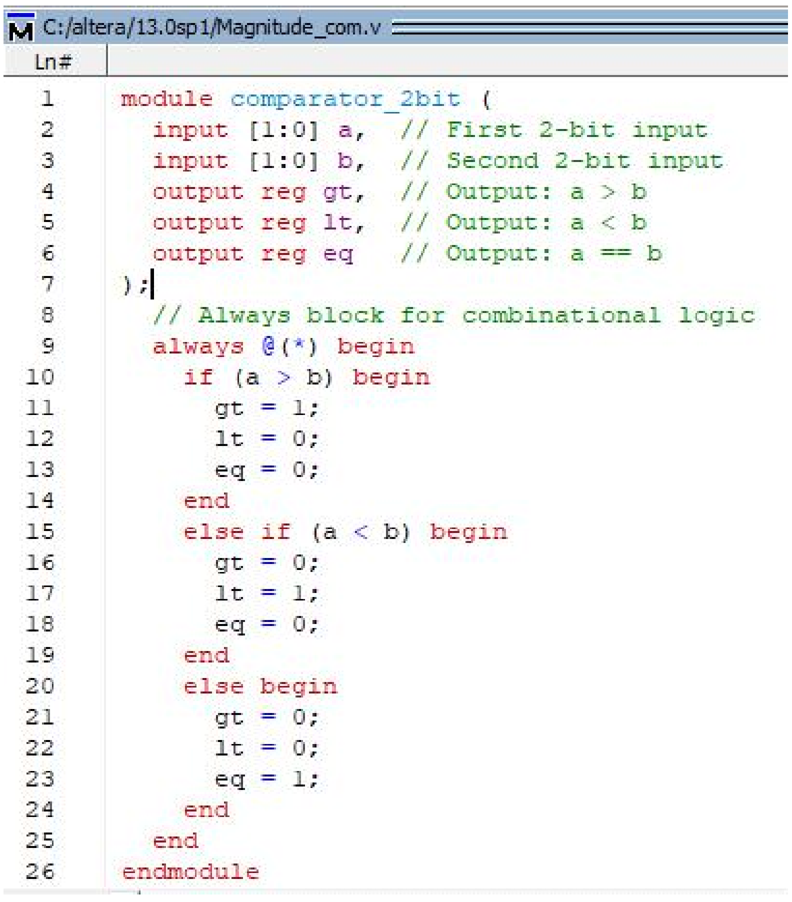


```
Wave - Default
                          Msgs
/decoder_2x4_tb/in        11      00   01   10   11
  [1]                     1
  [0]                     1
/decoder_2x4_tb/out       1000    0001 0010 0100 1000
  [3]                     St1
  [2]                     St0
  [1]                     St0
  [0]                     St0
Now                       70 ps        20 ps     40 ps     60 ps     80
Cursor 1                  0 ps
```

Fig. 12

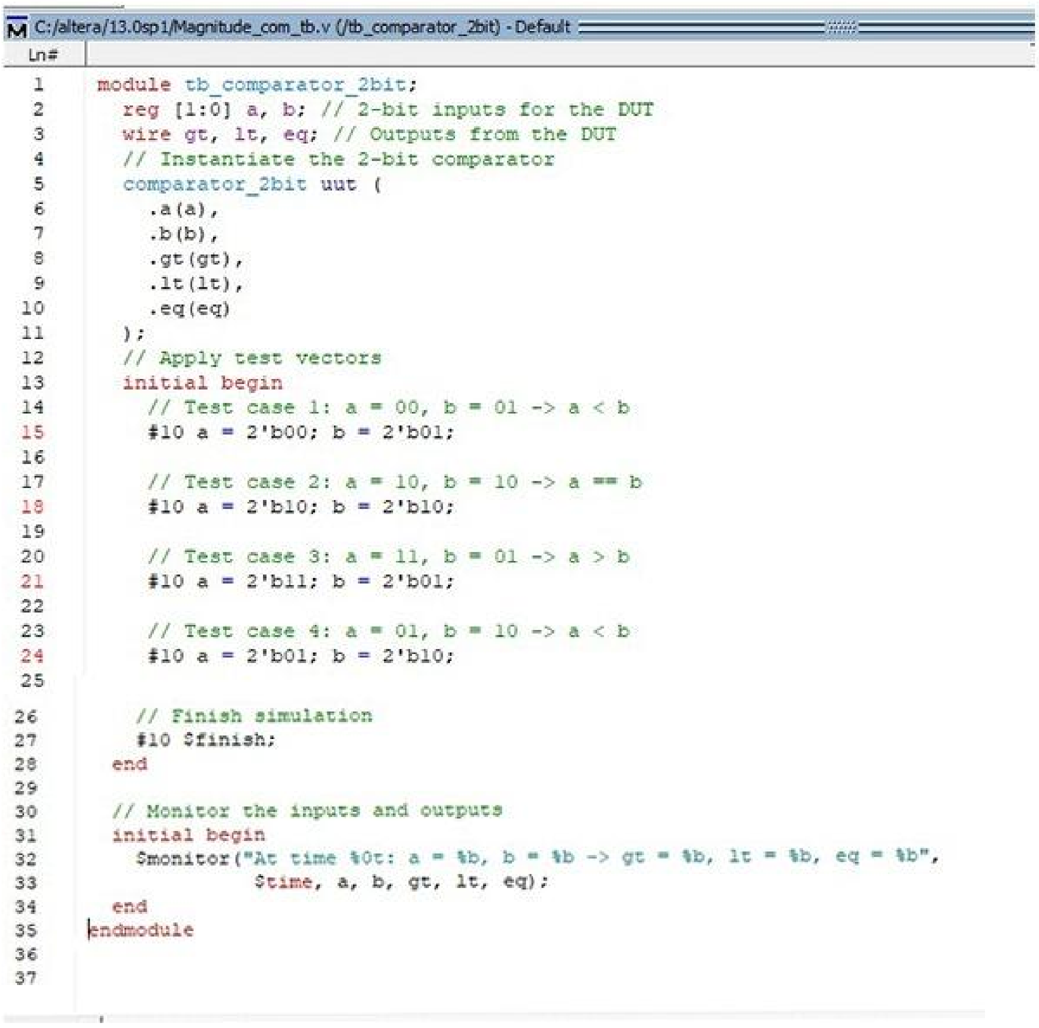
4.2 Bit Magnitude Comparator

A comparator used to compare two binary numbers each of two bits is called a 2-bit magnitude comparator. It consists of four inputs and three outputs to generate less than, equal to, and greater than between two binary numbers. The code regarding the basics is provided as HDL code in the editor depicted in figure 13 along with the testbenches in figure 14.



```verilog
module comparator_2bit (
  input [1:0] a,   // First 2-bit input
  input [1:0] b,   // Second 2-bit input
  output reg gt,   // Output: a > b
  output reg lt,   // Output: a < b
  output reg eq    // Output: a == b
);
  // Always block for combinational logic
  always @(*) begin
    if (a > b) begin
      gt = 1;
      lt = 0;
      eq = 0;
    end
    else if (a < b) begin
      gt = 0;
      lt = 1;
      eq = 0;
    end
    else begin
      gt = 0;
      lt = 0;
      eq = 1;
    end
  end
endmodule
```

Fig. 13


C:/altera/13.0sp1/Magnitude_com_tb.v (/tb_comparator_2bit) - Default
Ln#

```
module tb_comparator_2bit;
  reg [1:0] a, b; // 2-bit inputs for the DUT
  wire gt, lt, eq; // Outputs from the DUT
  // Instantiate the 2-bit comparator
  comparator_2bit uut (
    .a(a),
    .b(b),
    .gt(gt),
    .lt(lt),
    .eq(eq)
  );
  // Apply test vectors
  initial begin
    // Test case 1: a = 00, b = 01 -> a < b
    #10 a = 2'b00; b = 2'b01;

    // Test case 2: a = 10, b = 10 -> a == b
    #10 a = 2'b10; b = 2'b10;

    // Test case 3: a = 11, b = 01 -> a > b
    #10 a = 2'b11; b = 2'b01;

    // Test case 4: a = 01, b = 10 -> a < b
    #10 a = 2'b01; b = 2'b10;

    // Finish simulation
    #10 $finish;
  end

  // Monitor the inputs and outputs
  initial begin
    $monitor("At time %0t: a = %b, b = %b -> gt = %b, lt = %b, eq = %b",
             $time, a, b, gt, lt, eq);
  end
endmodule
```

Fig. 14

The simulation waveform is shown below in figure 15. The transcript which has been run in the terminal is also shown in figure 16.

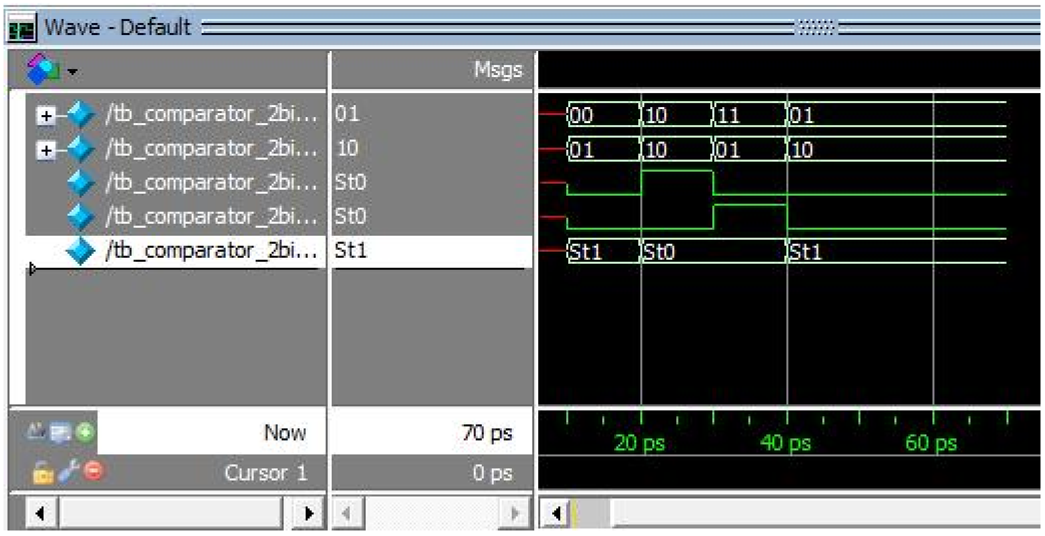
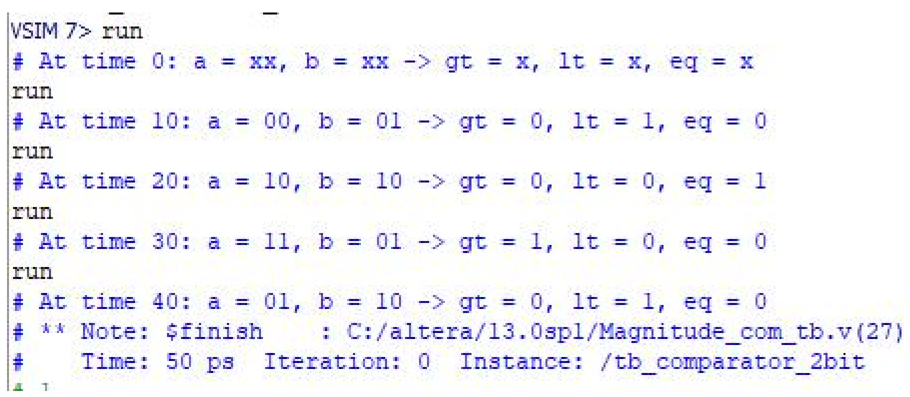


Fig. 15



```
VSIM 7> run
# At time 0: a = xx, b = xx -> gt = x, lt = x, eq = x
run
# At time 10: a = 00, b = 01 -> gt = 0, lt = 1, eq = 0
run
# At time 20: a = 10, b = 10 -> gt = 0, lt = 0, eq = 1
run
# At time 30: a = 11, b = 01 -> gt = 1, lt = 0, eq = 0
run
# At time 40: a = 01, b = 10 -> gt = 0, lt = 1, eq = 0
# ** Note: $finish    : C:/altera/13.0sp1/Magnitude_com_tb.v(27)
#    Time: 50 ps  Iteration: 0  Instance: /tb_comparator_2bit
```

Fig. 16

Result Analysis:

1. 2X1 Multiplexer

If we take the waveform from **figure 5** we can see that the simulation of the waveform of the multiplexer is accurate because of the matching with the transcript in **figure 3**. There are 8 combinations of inputs and outputs. For an example,

- At the beginning when A and B both are 0, and the selection pin is 0 the output remains 0

- But while the A is set to 1 and B is 0, but the selection pin is 0, the output changed to 1.
  This behaviour is just as opposite while we see while selection pin is 1 but the output is 0.

2. Full Adder

From **figure 8** we can see the waveform depicting the input and output combination of a full adder. From the truth table of a Full Adder we can match the transcript as well as the waveform. The truth table is:

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

From the transcript, the waveform can be matched.

- At the beginning, when A= 0 and B = 0, Cin = 0, the Sum and Cout are 0 - When A = 1 and B= 0 , Cin = 0 and the Sum = 1 but Cout = 0

3. 2X4 Decoder

All of the 4 combinations have been tested with the testbenches along with the simulation waveform. The truth table is given as:

A	B	Y0	Y1	Y2	Y3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

To take an example, from the simulation in **figure 12** we can see that,

- When A=0 and B=0 the output binary format shows 0001 in reverse order of the Y0 Y1 Y2 Y3.

4. 2 Bit Magnitude Comparator

The truth table for the 2 bit comparator is as shown below.

A	B	A<B	A=B	A>B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

From the table, it is evident that whenever one bit is 1, then the greater or less than sign comes to be 1, and while both are equal, the comparator shows that they are equal. From the **figure 15** of the simulation waveform of 2 Bit magnitude comparator, we can observe that,

- From the binary literal, we can see clearly that when A is 00 but B is 01 then the less than part (‘lt’ in the transcript) is up to 1 (as we are comparing A with B).
- Again, when both A and B are 10 then the last comparison, which is the equal (‘eq’ in the transcript) is up to 1 and others are 0.

Discussion:

1. The experiment tests the behaviour by generating simulation waveforms and transcripts from the testbenches provided by the HDL codes for multiplexer, full adder, decoder and magnitude comparator.
2. The waveform simulation has been perfectly matched at the end of every simulation with the transcripts that were produced at terminal.