CHITTAGONG UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Department of Electronics and Telecommunication Engineering

VLSI Technology Sessional

ETE 404

Experiment No:10

Introduction to Verilog Testbenches and Functional Verification

---

Submitted by:

Shamanza Chowdhury

ID: 1908008

Submitted to:

Arif Istiaque Rupom

Lecturer

Dept. of ETE,CUET

November 02, 2024

1 Objectives:

- To familiarize with ModelSim.
- To familiarize with writing testbenches.
- To verify combinational circuits imposing test vectors.

2 Apparatus:

- **Software:**ModelSim-Altera

3 2x1 Multiplexer Design:

3.1 2x1 Multiplexer DUT :

Implementing the DUT using the following Verilog code:

Listing 1: 2x1 Multiplexer

```
1    module mux_2x1(input wire a, b, sel, output wire y);
2    assign y = (sel) ? b : a;
3    endmodule
```

3.2 2x1 Multiplexer Testbench :

Creating a testbench to simulate the DUT using the following Verilog code:

Listing 2: 2x1 Multiplexer Testbench

```
1  module mux_2x1_tb;
2   // Inputs to the DUT (reg type for testbench)
3   reg a, b, sel;
4   // Output from the DUT (wire type for testbench)
5   wire y;
6   // Instantiate the DUT
7   mux_2x1 uut (
8     .a(a),
9     .b(b),
10    .sel(sel),
11    .y(y)
```

```
12  );
13  // Test stimulus
14  initial begin
15   // Initialize inputs
16   a = 0;
17   b = 0;
18   sel = 0;
19   // Apply test vectors
20   #10 a = 0; b = 0; sel = 0; // Expect y = a = 0
21   #10 a = 0; b = 1; sel = 0; // Expect y = a = 0
22   #10 a = 1; b = 0; sel = 0; // Expect y = a = 1
23   #10 a = 1; b = 1; sel = 0; // Expect y = a = 1
24   #10 a = 0; b = 0; sel = 1; // Expect y = b = 0
25   #10 a = 0; b = 1; sel = 1; // Expect y = b = 1
26   #10 a = 1; b = 0; sel = 1; // Expect y = b = 0
27   #10 a = 1; b = 1; sel = 1; // Expect y = b = 1
28   // Finish the simulation
29   #10 $finish;
30  end
31  // Monitor the output
32  initial begin
33   $monitor("At time %0t: a = %b, b = %b, sel = %b -> y = %b",$time, a, b, sel, y
        );
34  end
35 endmodule
```

3.3 Behavior Analysis of 2X1 Multiplexer:

Simulation: Simulating the design using ModelSim and analyzing the output waveforms:

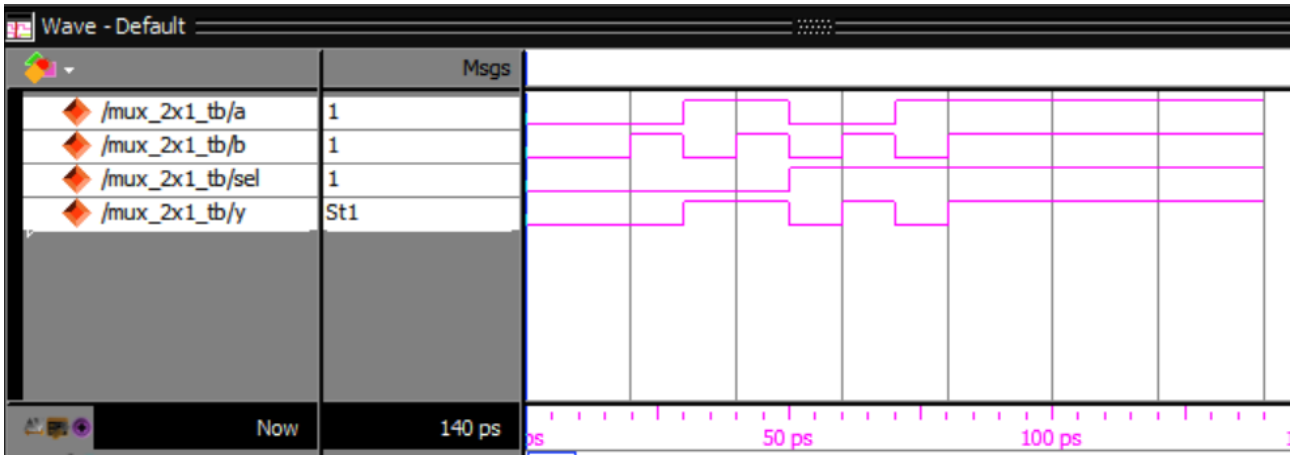

Figure 01:Waveform of 2x1 Multiplexer.

Transcript: Verifying the functionality from the Transcript window:

```
Transcript
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    : F:/vlsi_quartus/exp10/test/mux_2x1_tb.v(29)
#    Time: 90 ps  Iteration: 0  Instance: /mux_2x1_tb
# 1
# Break in Module mux_2x1_tb at F:/vlsi_quartus/exp10/test/mux_2x1_tb.v line 29
```

Figure 02:Transcript of 2x1 Multiplexer.

It can be seen that the output “y” mimics “a” when “sel” is 0 and mimics “b” when “sel” is 1.Hence it has been matched with the test vectors.

4 Full Adder Design:

4.1 Full Adder DUT :

Implementing the DUT using the following Verilog code:

Listing 3: Full Adder

```
1 module full_adder (
2  input a, // First input bit
3  input b, // Second input bit
```

```
 input cin, // Carry input
 output sum, // Sum output
 output cout // Carry output
);
 assign {cout, sum} = a + b + cin;
endmodule
```

4.2 Full Adder Testbench :

Creating a testbench to simulate the DUT using the following Verilog code:

Listing 4: 2x1 Full Adder Testbench

```
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
endmodule
```

4.3 Behavior Analysis of Full Adder:

Simulation: Simulating the design using ModelSim and analyzing the output waveforms:

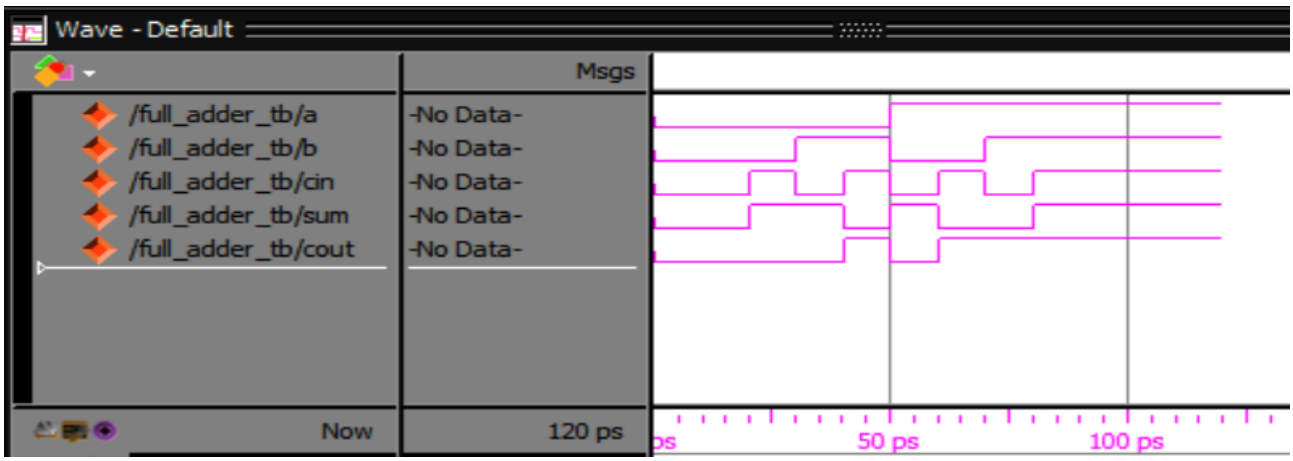

Figure 03:Waveform of Full Adder.

Transcript: Verifying the functionality from the Transcript window:

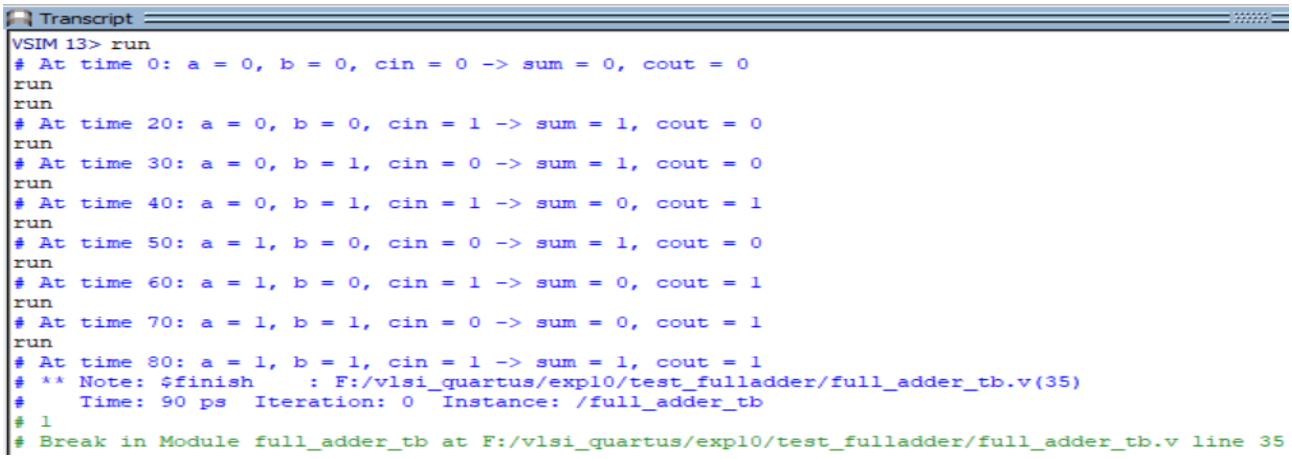

Figure 04:Transcript of Full Adder.

It can be seen that the full adder outputs match the expected test vectors: the 'sum' and 'cout' values correctly represent the binary addition of inputs a, b, and cin, confirming the correct operation of the full adder.

5 2x4 Decoder Design:

5.1 2x4 Decoder DUT :

Implementing the DUT using the following Verilog code:

Listing 5: 2x4 Decoder

```
module decoder_2x4 (
 input [1:0] in, // 2-bit input
 output [3:0] out // 4-bit output
);
 assign out = (in == 2'b00) ? 4'b0001 :
 (in == 2'b01) ? 4'b0010 :
 (in == 2'b10) ? 4'b0100 :
 (in == 2'b11) ? 4'b1000 : 4'b0000;
endmodule
```

5.2 2x4 Decoder Testbench :

Creating a testbench to simulate the DUT using the following Verilog code:

Listing 6: 2x4 Decoder Testbench

```
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
```

5.3 Behavior Analysis of 2X4 Decoder:

Simulation: Simulating the design using ModelSim and analyzing the output waveforms:

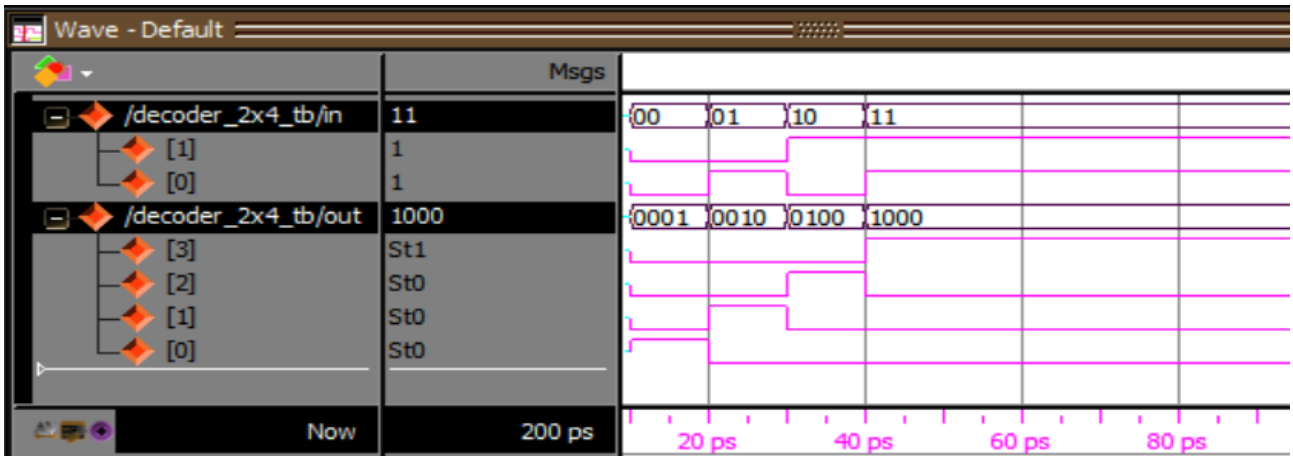

Figure 05:Waveform of 2x4 Decoder.

Transcript: Verifying the functionality from the Transcript window:

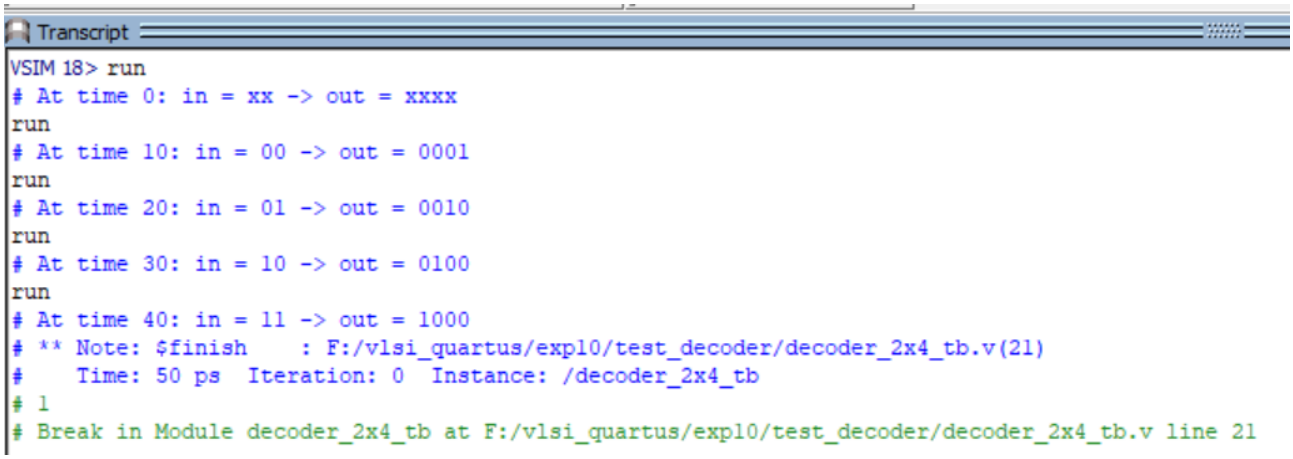

Figure 06:Transcript of 2x4 Decoder.

It can be seen that the decoder outputs match the expected test vectors: 'out' = 0001 for 'in' = 00, 'out' = 0010 for 'in' = 01, 'out' = 0100 for 'in' = 10, and 'out' = 1000 for 'in' = 11, confirming the correct behavior of the decoder.

6 2 Bit Magnitude Comparator Design:

6.1 Magnitude Comparator DUT :

Implementing the DUT using the following Verilog code:

Listing 7: Magnitude Comparator

```
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 @(*) 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
```

6.2 Magnitude Comparator Testbench :

Creating a testbench to simulate the DUT using the following Verilog code:

Listing 8: Magnitude Comparator Testbench

```
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
```

6.3 Behavior Analysis of Full Adder:

Simulation: Simulating the design using ModelSim and analyzing the output waveforms:

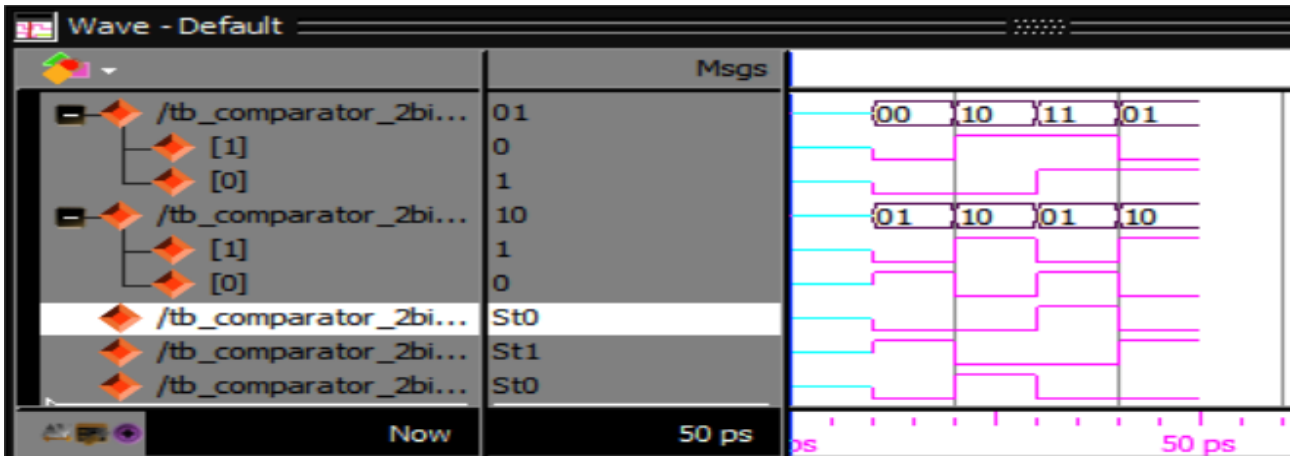

Figure 07:Waveform of Magnitude Comparator.

Transcript: Verifying the functionality from the Transcript window:

```
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
VSIM 8> run
# At time 40: a = 01, b = 10 -> gt = 0, lt = 1, eq = 0
# ** Note: $finish    : F:/vlsi_quartus/exp10/test_comparator/tb_comparator_2bit.v(23)
#    Time: 50 ps  Iteration: 0  Instance: /tb_comparator_2bit
# 1
# Break in Module tb_comparator_2bit at F:/vlsi_quartus/exp10/test_comparator/tb_comparator_2bit.v line 23
```

Figure 08:Transcript of Magnitude Comparator.

It can be seen that the comparator outputs match the expected test vectors: 'gt' = 0, 'lt' = 1, and 'eq' = 0 when a $<$ b; 'gt' = 0, 'lt' = 0, and 'eq' = 1 when a == b; and 'gt' = 1, 'lt' = 0, and 'eq' = 0 when a $>$ b, confirming the correct behavior of the 2-bit comparator.

7 Discussion:

- The experiment was conducted to evaluate the behavior of designs by generating simulation waveforms and transcripts using HDL codes for the multiplexer, full adder, decoder, and magnitude comparator.
- It was observed that the waveforms matched the expected outputs perfectly, as confirmed by the terminal transcripts . This confirmed the functionality and accuracy of the designed modules and testbenches.
- The ModelSim-Altera software was introduced and utilized in this lab for all coding and simulations.
- Project file name and module name should be same.
- The initial step involved writing the Design Under Test (DUT) code, representing the design to be verified. Subsequently, the testbench code was written, containing the design's truth table to facilitate verification.
- The testbench interacted with the DUT by applying input vectors and observing the outputs, while monitors were used to display the DUT's outputs after verification.
- No errors were encountered during compilation or simulation, indicating the correctness of the Verilog code for all modules.