DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATION ENGINEERING

CHITTAGONG UNIVERSITY OF ENGINEERING AND TECHNOLOGY

CHATTOGRAM – 4349, BANGLADESH

Experiment No. 10

Introduction to Verilog Testbenches and Functional Verification

PRECAUTIONS:

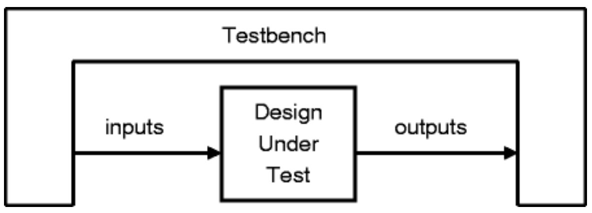
- Students must carefully read the lab manual before coming to lab to avoid any inconveniences.
- Students must carry a flash drive to transfer files and lab manuals.
- Use of mobile phone in lab is strictly prohibited and punishable offence.
- Experiment files must be uploaded to GitHub including home tasks.

OBJECTIVES:

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

HDL TESTBENCHES:

A testbench is a piece of code written in a hardware description language (HDL) like Verilog or VHDL to verify the functionality of a hardware design by applying inputs (stimulus) and observing the outputs. It serves as a virtual environment that simulates the behavior of the design under test (DUT) by generating test patterns, comparing results, and checking for correctness. In simpler terms, a testbench is a verification framework used to simulate a design's behavior and ensure that it works as intended before implementing it on hardware.



The purpose of a testbench is to simulate the DUT with a range of probable inputs. It can check the outputs from the DUT and check whether they meet expected results or not. It is also handy in automation as multiple test cases can be run systematically.

BASIC COMPONENTS OF TESTBENCHES:

Design Under Test (DUT): This is the actual design or module you want to verify. The testbench instantiates the DUT and interacts with it by sending inputs and observing the outputs.

Stimulus Generation: This is the process of generating input signals for the DUT. These signals can be:

- Manually specified: The user defines specific input patterns.
- Randomly generated: Inputs are generated using constrained-random methods to explore different test scenarios.

Monitors: Monitors observe the DUT's output signals. They are responsible for capturing data and passing it to the checkers or scoreboards for validation.

Checkers: Checkers compare the DUT's actual output against the expected output. If the results differ, the checker flags an error. This comparison can be automated to handle complex designs.

Scoreboard: A scoreboard is used to store expected values and check the DUT's outputs over time. It is particularly useful in more complex designs where multiple outputs need to be tracked and validated over a sequence of events.

Clock and Reset Generation: In most digital designs, clocks and resets are needed to drive the DUT. The testbench typically generates these signals.

Assertions: Assertions are conditions that must always hold true in a design. They can be embedded in the testbench or the DUT to check for certain properties or assumptions (e.g., signal timing or valid states).

STRUCTURE OF SIMPLE VERILOG TESTBENCHE:

A simple Verilog testbench consists of the following sections:

1. **Module Declaration:** The testbench itself is declared as a module, but it doesn't have any input or output ports since it acts as a virtual environment.
2. **Instantiation of DUT**: The design you want to verify (DUT) is instantiated inside the testbench.
3. **Initial Block:** The initial block applies stimulus to the inputs of the DUT and contains the sequence of tests.
4. **Clock Generation:** If the design is sequential, a clock signal is usually generated to drive the DUT.
5. **Display and Monitor Statements:** These statements help monitor and display the DUT's output.

Example of a basic testbench is given below:

Multiplexer Design (DUT):

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

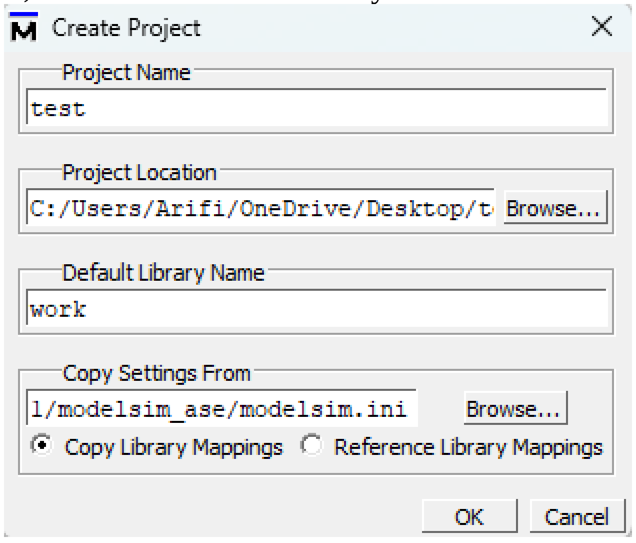
Testbench:

```
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
```

<u>SIMULATION USING ModelSim-Altera:</u>

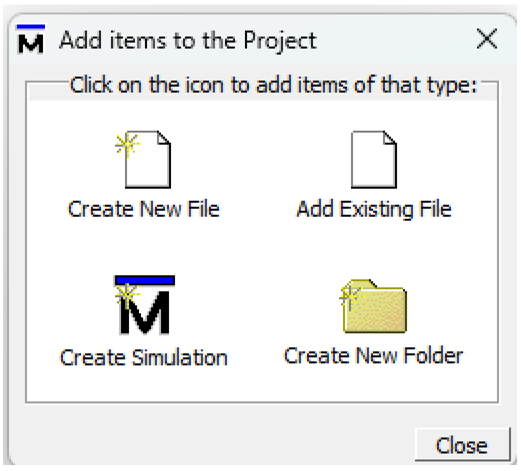
1. Create a new folder at any convenient location. We will create a folder in the **Desktop** and name it **"test"**. Start **ModelSim-Altera** from the start menu by searching **"modelsim"**. **Modelsim-Altera** should start up. Execute **File** > **New** > **Project.**

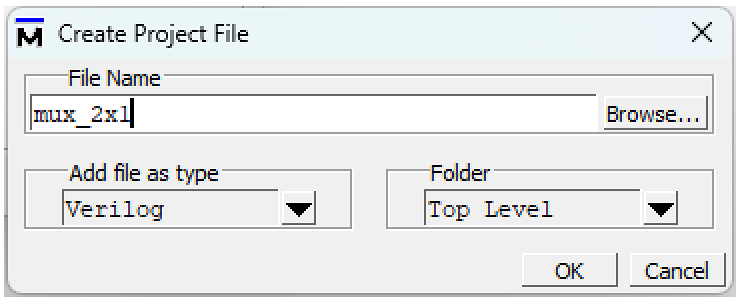
2. Fill up the project name, location and default library names as shown below.


Create Project
Project Name: test
Project Location: C:/Users/Arifi/OneDrive/Desktop/t
Default Library Name: work
Copy Settings From: l/modelsim_ase/modelsim.ini
[✓ Copy Library Mappings] Reference Library Mappings

   Click **OK.** An "Add Item" window should popup.

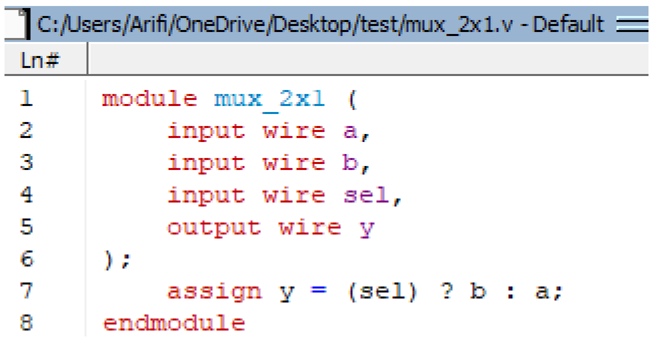
3. In this window we can add any new or existing files to our project, which we will do by clicking on **"Create New File"**, then giving a file name and file type as shown below. Name it **"mux_2x1"**. Then press **OK** and close the **"Add Items"** window by clicking **Close**.





File Name: mux_2x1
Add file as type: Verilog
Folder: Top Level

4. Double-click the **"mux_2x1.v"** file to open the code editor window. Paste in the previously given code into the editor. It should look like the following:

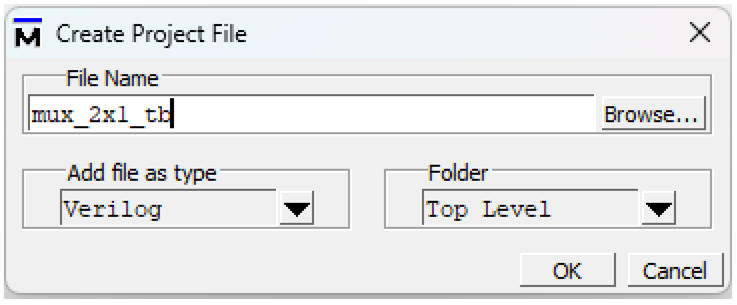


C:/Users/Arifi/OneDrive/Desktop/test/mux_2x1.v - Default

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

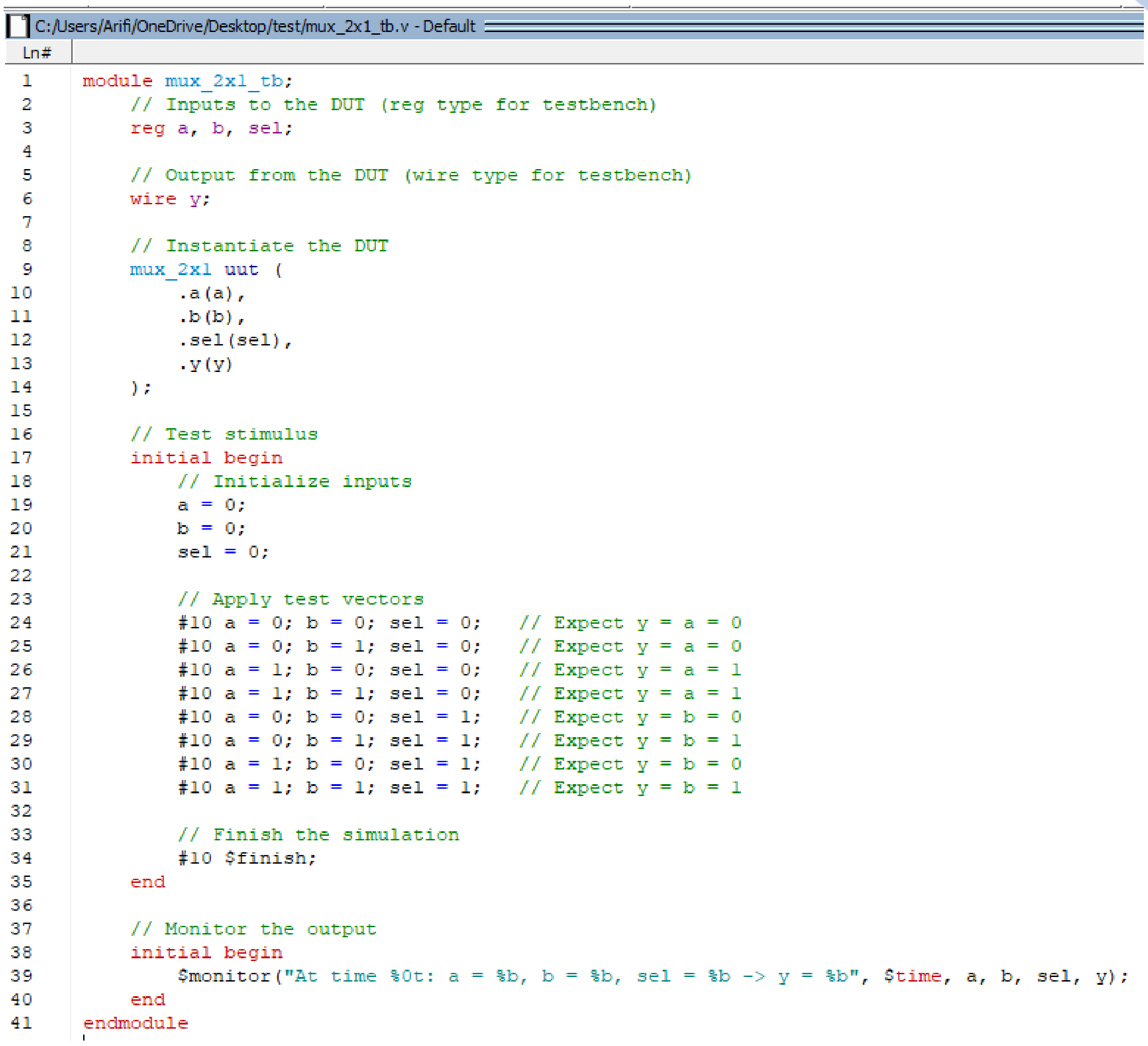
Use Ctrl + S to save the file.

5. Now click on the blank space in the **Project** pane on the left. Execute **Project > Add to Project > New File**. Name it **"mux_2x1_tb"**.



File Name: mux_2x1_tb
Add file as type: Verilog
Folder: Top Level

6. Open **"mux_2x1_tb"** by double-clicking it on the **Project** pane and paste in the previously given testbench code into the editor. It should look like the following:

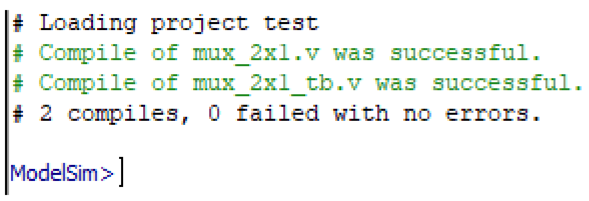


```
module mux_2x1_tb;
    // Inputs to the DUT (reg type for testbench)
    reg a, b, sel;

    // 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
```

Press Ctrl + S to save the file.

7. Click on **“Compile All”** button on the ribbon to compile both the DUT file and testbench. Or you can execute **Compile > Compile All**.

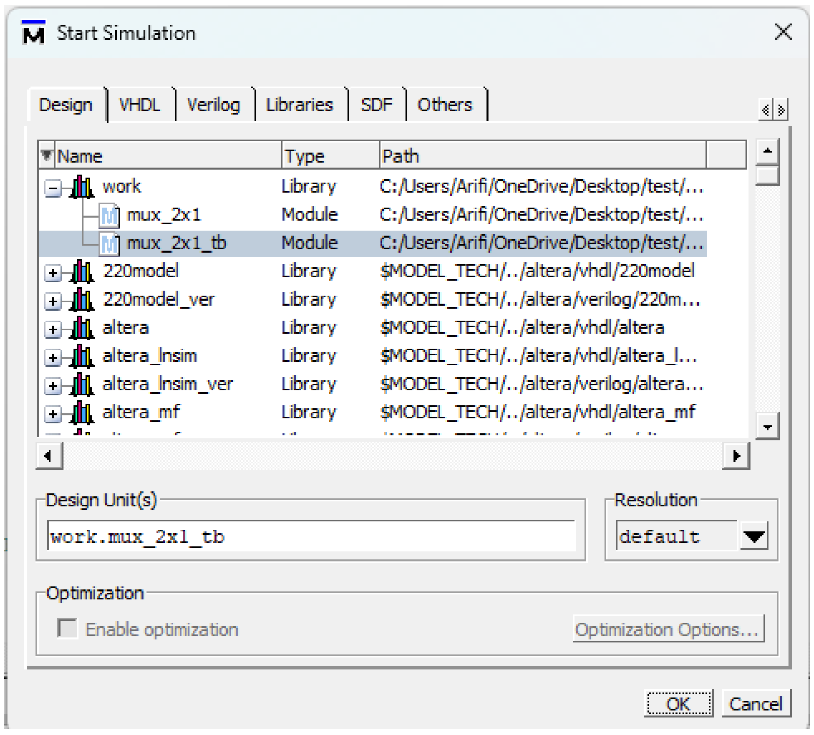
8. In the **Transcript** window at the bottom of the screen, you should find compilation was successful with no errors.



```
# Loading project test
# Compile of mux_2x1.v was successful.
# Compile of mux_2x1_tb.v was successful.
# 2 compiles, 0 failed with no errors.

ModelSim>
```

9. Now, click on the **“Simulate”** button on the ribbon to run the simulation. You can also execute **Simulate > Start Simulation**.

10. The **“Start Simulation”** window will appear. From the **Design** tab select your testbench under “work” or the default library name you provide in step 2. Click **OK.**



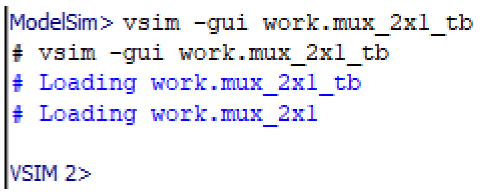
Name	Type	Path
work	Library	C:/Users/Arifi/OneDrive/Desktop/test/...
mux_2x1	Module	C:/Users/Arifi/OneDrive/Desktop/test/...
mux_2x1_tb	Module	C:/Users/Arifi/OneDrive/Desktop/test/...
220model	Library	$MODEL_TECH/../altera/vhdl/220model
220model_ver	Library	$MODEL_TECH/../altera/verilog/220m...
altera	Library	$MODEL_TECH/../altera/vhdl/altera
altera_lnsim	Library	$MODEL_TECH/../altera/vhdl/altera_l...
altera_lnsim_ver	Library	$MODEL_TECH/../altera/verilog/altera...
altera_mf	Library	$MODEL_TECH/../altera/vhdl/altera_mf

Design Unit(s): work.mux_2x1_tb

Resolution: default

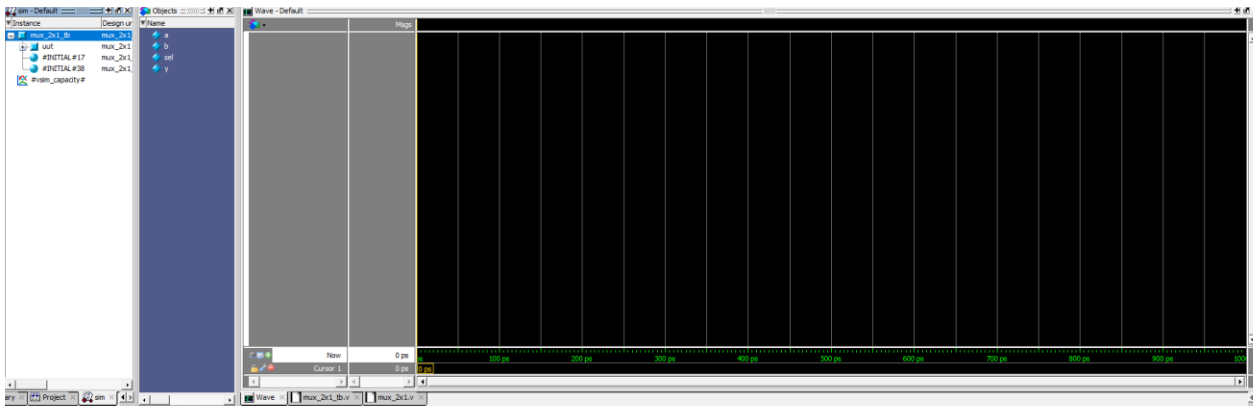
Optimization
Enable optimization

The following message should appear in the **Transcript** window:

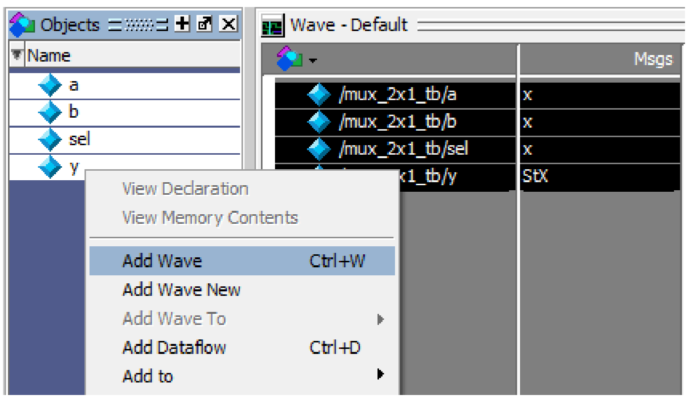


```
ModelSim> vsim -gui work.mux_2x1_tb
# vsim -gui work.mux_2x1_tb
# Loading work.mux_2x1_tb
# Loading work.mux_2x1

VSIM 2>
```

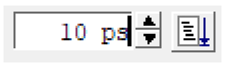
11. Switch to the **Wave** view from the bottom-left of the editor window. If it's not visible, execute **View** > **Wave**.



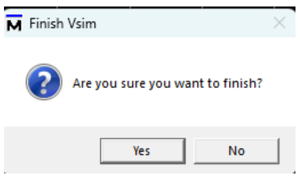
12. Now highlight all the nodes in the **Objects** window, right-click and select **"Add Wave"**. This should add the nodes to the **Wave** window.



13. As the timing was set in **10ps** intervals in the testbench, set **Run Length** to 10ps and click on **"Run"** or press **F9**.

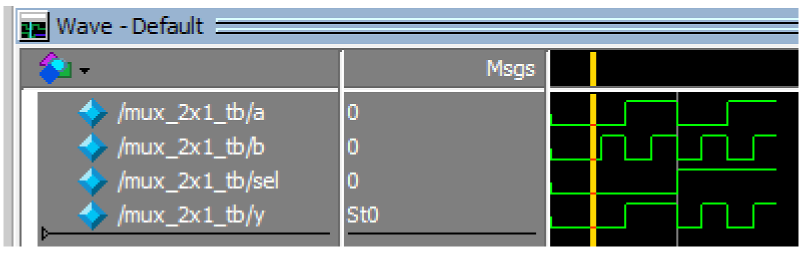


14. You will be able to see the simulation run in 10ps intervals. Keep pressing **Run** and at the end a **"Finish Vsim"** window will appear. Click **No**.



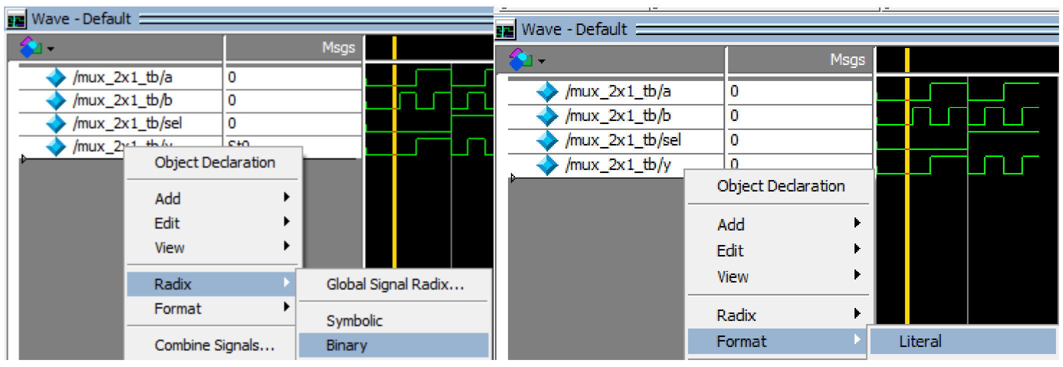
Are you sure you want to finish?

15. Now for the given test vectors, the functionality of the design can be verified from the wave output generated by **ModelSim**. You can check the values by clicking on the wave output and dragging the cursor.

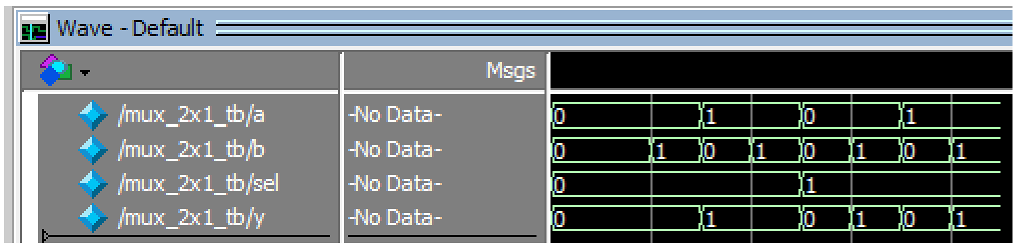


/mux_2x1_tb/a 0
/mux_2x1_tb/b 0
/mux_2x1_tb/sel 0
/mux_2x1_tb/y St0

To view this in a different manner, select all the wave objects, right-click, execute **Radix > Binary** and then execute **Format > Literal**.

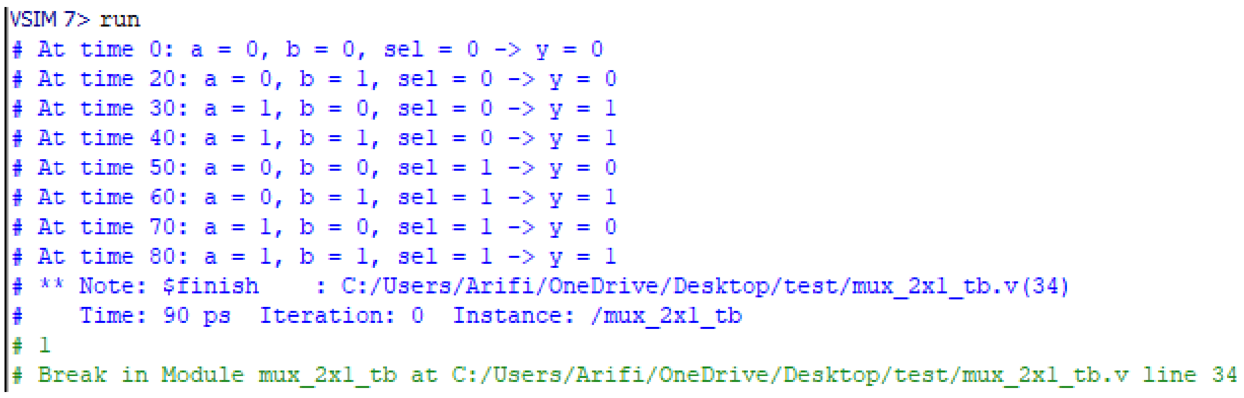


This will give the following outcome:



/mux_2x1_tb/a -No Data-
/mux_2x1_tb/b -No Data-
/mux_2x1_tb/sel -No Data-
/mux_2x1_tb/y -No Data-

16. Functionality can also be verified from the **Transcript** window.



```
VSIM 7> run
# At time 0: a = 0, b = 0, sel = 0 -> y = 0
# At time 20: a = 0, b = 1, sel = 0 -> y = 0
# At time 30: a = 1, b = 0, sel = 0 -> y = 1
# At time 40: a = 1, b = 1, sel = 0 -> y = 1
# At time 50: a = 0, b = 0, sel = 1 -> y = 0
# At time 60: a = 0, b = 1, sel = 1 -> y = 1
# At time 70: a = 1, b = 0, sel = 1 -> y = 0
# At time 80: a = 1, b = 1, sel = 1 -> y = 1
# ** Note: $finish    : C:/Users/Arifi/OneDrive/Desktop/test/mux_2x1_tb.v(34)
#    Time: 90 ps  Iteration: 0  Instance: /mux_2x1_tb
# 1
# Break in Module mux_2x1_tb at C:/Users/Arifi/OneDrive/Desktop/test/mux_2x1_tb.v line 34
```

17. Verify the functionality where the output **"y"** mimics **"a"** when **"sel"** is 0 and mimics **"b"** when **"sel"** is 1.

Now do the same for the following examples:

Full Adder:

```
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
);

    assign {cout, sum} = a + b + cin;

endmodule
```

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
```

2x1 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
```

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
```

2 Bit 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
```

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
```

Rubrics:

Criteria	Below Average (1)	Average (2)	Good (3)
Formatting	Report is not properly formatted, missing objectives, discussions and references.	Report is somewhat formatted but missing proper references.	Report is properly formatted.
Code	HDL codes are properly written.	HDL codes are properly written and simulation results are accurate.	
Writing	Writing is poor and not informative. It does not address the design decisions and contains high plagiarism.	Writing is average and original. Design decisions are somewhat explored.	Writing is excellent with every design decision adequately explored.
Diagram	Diagrams are of bad quality and unreadable.	Diagrams are clear and of high quality.	