Verilog HDL:
Testbenches

Pravin Zode

Outline

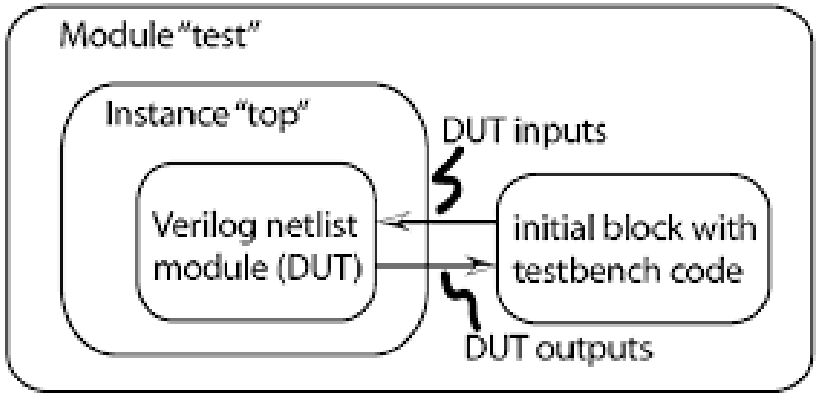
- Basics and Role of Testbench
- Component of Testbench
- Exercises

Introduction

- A testbench is a Verilog module used to verify the functionality of a Device Under Test (DUT)
- It is a non-synthesizable module designed for simulation only.
- Used to apply stimulus and observe DUT responses.
- Essential for validating digital designs before implementation

Role of Testbench

- **Stimulus generation:** Provides input signals to the DUT
- **Response observation:** Captures and checks DUT output
- **No I/O ports:** Unlike hardware modules, a testbench is self-contained
- **Simulation control:** Defines test sequences and simulation duration



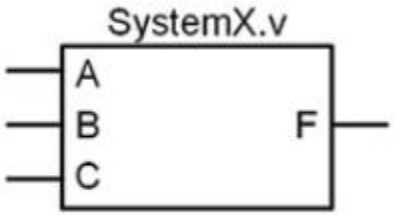
Component of Testbench

- **DUT Instantiation** - Connects DUT signals with testbench variables
- **Stimulus Generation** - Uses initial or always blocks to apply inputs.
- **Monitoring & Verification** - Uses $monitor, $display, or file I/O to observe output.
- **Simulation Control** - Ends simulation using $finish

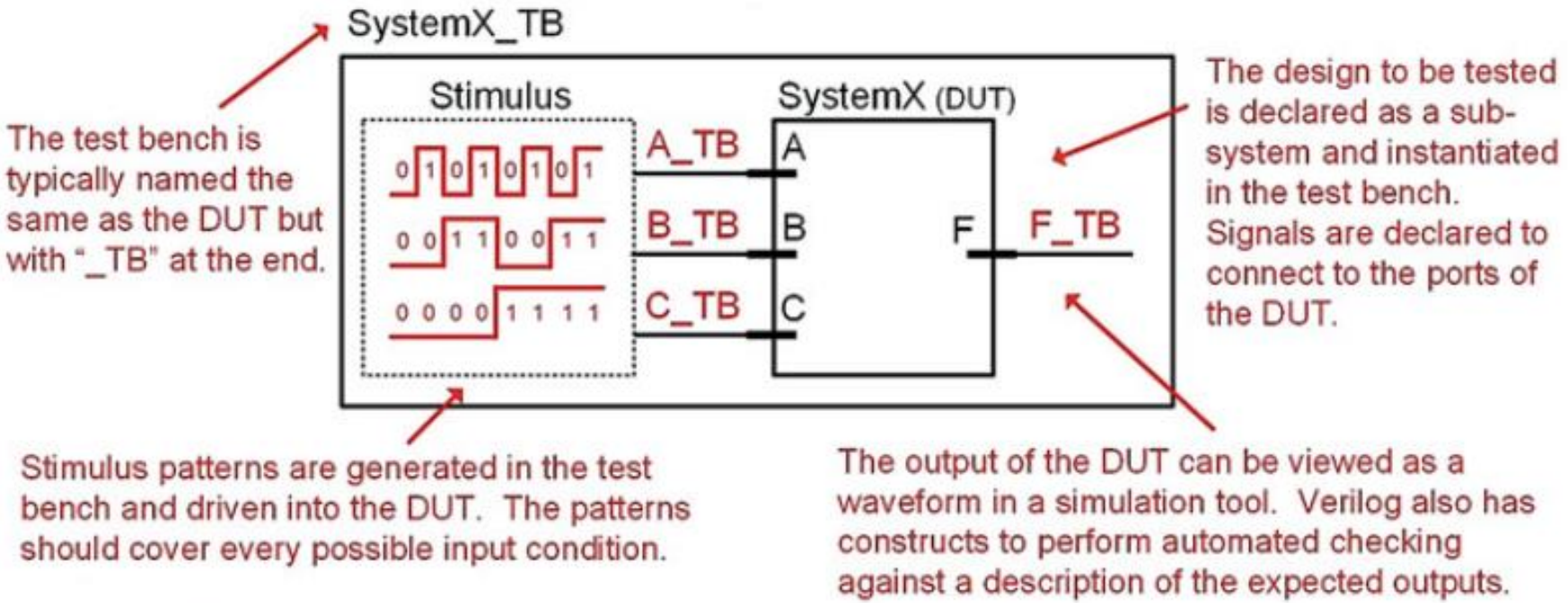
Why Testbench Not Have Inputs and Outputs?

- **Self-Contained Simulation :**
  - A testbench is not a hardware entity; it is purely for simulation.
  - It is used to apply test vectors and observe outputs within the same module.
- **Direct Signal Driving and Observation:**
  - The testbench defines registers (reg) to drive inputs to the DUT.
  - The testbench defines wires (wire) to capture DUT outputs.
- **No Need for Port Declaration :**
  - Since a testbench is not synthesized into hardware, it does not require input and output ports like a synthesizable module.

Testbench Structure



$$F = \sum_{A,B,C}(0,2,6) = A'\cdot B'\cdot C' + A'\cdot B\cdot C' + A\cdot B\cdot C'$$



Testbench Structure

SystemX_TB.v

```
`timescale 1ns/1ps

module SystemX_TB ();

  reg  A_TB, B_TB, C_TB;
  wire F_TB;

  SystemX DUT (.F(F_TB), .A(A_TB), .B(B_TB), .C(C_TB));

  initial
    begin
            A_TB=0; B_TB=0; C_TB=0;
      #10   A_TB=0; B_TB=0; C_TB=1;
      #10   A_TB=0; B_TB=1; C_TB=0;
      #10   A_TB=0; B_TB=1; C_TB=1;
      #10   A_TB=1; B_TB=0; C_TB=0;
      #10   A_TB=1; B_TB=0; C_TB=1;
      #10   A_TB=1; B_TB=1; C_TB=0;
      #10   A_TB=1; B_TB=1; C_TB=1;
    end

endmodule
```

Whenever delay is used, a timescale should be defined.

Type "reg" is used for the inputs of the DUT, "wire" is used for the outputs.

Instantiate the DUT.

An initial block can be used to drive in a series of stimulus patterns. The block contains delayed assignments that will drive in all possible patterns into the combinational logic circuit. This will execute once.

Initial Procedural Statement

- The initial block in Verilog is used to execute a set of procedural statements once at the beginning of simulation
- Ideal for initializing values or setting up conditions for the design

```
initial begin
    // Statements to be executed at the start of simulation
end
```

Initial Procedural Statement

Key Features:

- Executes once at the start of simulation
- Useful for initializing signals, variables, or performing simulation-specific setup
- Can contain multiple sequential statements
- Can be used to generate test vectors or initial conditions for simulation

```
initial begin
    A = 0;    // Initialize variable A to 0
    B = 1;    // Initialize variable B to 1
    #10 A = 1; // After 10 time units, set A to 1
end
```

- Note: The initial block is not synthesizable, meaning it is used for simulation purposes only and does not translate to hardware

Console output

Why Display Values ?

- Helps observe internal signals during simulation
- Useful for debugging and verifying logic

$display

- Prints once when executed
- Good for specific events or checkpoints

$monitor

- Prints automatically whenever any listed signal changes
- Best for continuous signal tracking

```
$display("A = %b, B = %b", A, B);
$monitor("Time = %0t | A = %b, B = %b", $time, A, B);
```

When to Use What?

$display

- For testbench messages, print message once
- Printing at specific simulation times
- Debugging state transitions selectively

$monitor

- To continuously observe signal values
- Debugging unexpected signal changes
- When you want real-time tracking

timescale

‘timescale

- Specifies **time unit** and **time precision** for simulation
- It should be placed at the top of testbench file

```
`timescale <time_unit>/<time_precision>
```

```
`timescale 1ns/1ps
```

- 1ns: each #1 = 1 nanosecond
- 1ps: round-off accuracy = 1 picosecond

Basic Testbench Structure

```
module and_gate ( a,b,y);
input a, b;
output y;
assign y = a & b;
endmodule;
```

```
module andgate_tb;
reg A, B;
wire Y;
and_gate a1 ( .a(A), .b(B), .y(Y));

initial
begin
$monitor(A, B, Y);
  A = 1'b0; B = 1'b0;
  #5
  A = 1'b0; B = 1'b1;
  #5
  A = 1'b1; B = 1'b0;
  #5
  A = 1'b1; B = 1'b1;
 end
endmodule
```

Example: AND Gate

```
module and_gate ( a,b,y);
input a, b;
output y;
assign y = a & b;
endmodule;
```

```
module andgate_tb;
reg A, B;
wire Y;
and_gate a1 ( .a(A), .b(B), .y(Y));

initial
begin
$monitor(A, B, Y);
  A = 1'b0; B = 1'b0;
  #5
  A = 1'b0; B = 1'b1;
  #5
  A = 1'b1; B = 1'b0;
  #5
  A = 1'b1; B = 1'b1;
 end
endmodule
```

Example: AND Gate

```
1   `timescale 1ns / 1ps
2
3   module and_gate_tb;
4       reg a, b;
5       wire y;
6
7       and_gate uut (.a(a), .b(b), .y(y));
8
9       initial begin
10          $monitor("Time=%0t | a=%b, b=%b => y=%b", $time, a, b, y);
11
12          a = 0; b = 0; #10;
13          a = 0; b = 1; #10;
14          a = 1; b = 0; #10;
15          a = 1; b = 1; #10;
16
17          $finish;
18      end
19  endmodule
```

Example: Adder

```
1   module full_adder(input a, input b, input cin, output sum, output cout);
2     assign {cout, sum} = a + b + cin;
3   endmodule
4
5   module tb_full_adder;
6     reg a, b, cin;
7     wire sum, cout;
8
9     full_adder fa(a, b, cin, sum, cout);
10
11    initial begin
12      $monitor("Time=%0d | a=%b b=%b cin=%b | sum=%b cout=%b",
13               $time, a, b, cin, sum, cout);
14      a = 0; b = 0; cin = 0;
15      #10 a = 0; b = 0; cin = 1;
16      #10 a = 0; b = 1; cin = 0;
17      #10 a = 0; b = 1; cin = 1;
18      #10 a = 1; b = 0; cin = 0;
19      #10 a = 1; b = 0; cin = 1;
20      #10 a = 1; b = 1; cin = 0;
21      #10 a = 1; b = 1; cin = 1;
22      #10 $finish;
23    end
24  endmodule
```

Always Block

- Repeat commands throughout the duration of simulation
- More than one execution block run in parallel

```
1   `timescale 1ns / 1ps
2
3   module and_gate_tb;
4       reg a, b;
5       wire y;
6
7       and_gate uut (.a(a), .b(b), .y(y));
8
9       initial begin
10          a = 0; b = 0;
11      end
12
13      always #5 a = ~a; // Toggle 'a' every 5 time units
14      always #10 b = ~b; // Toggle 'b' every 10 time units
15
16      initial begin
17          #40; // Run simulation for 40 time units
18          $finish;
19      end
20  endmodule
```

Using For loop

```
1   `timescale 1ns / 1ps
2
3   module and_gate_tb;
4       reg a, b;
5       wire y;
6       integer i;
7
8       and_gate uut (.a(a), .b(b), .y(y));
9
10      initial begin
11          for (i = 0; i < 4; i = i + 1) begin
12              {a, b} = i; // Assign different input combinations
13              #10;
14              $display("Time=%0t | a=%b, b=%b => y=%b", $time, a, b, y);
15          end
16          $finish;
17      end
18  endmodule
```

Using Task

```
1   `timescale 1ns / 1ps
2
3   module and_gate_tb;
4       reg a, b;
5       wire y;
6
7       and_gate uut (.a(a), .b(b), .y(y));
8
9       task apply_inputs(input reg ta, input reg tb);
10          begin
11              a = ta; b = tb;
12              #10;
13              $display("Time=%0t | a=%b, b=%b => y=%b", $time, a, b, y);
14          end
15      endtask
16
17      initial begin
18          apply_inputs(0, 0);
19          apply_inputs(0, 1);
20          apply_inputs(1, 0);
21          apply_inputs(1, 1);
22          $finish;
23      end
24  endmodule
```

Using Case

```
1   `timescale 1ns / 1ps
2
3   module and_gate_tb;
4       reg [1:0] test_vector;
5       wire y;
6       reg a, b;
7
8       and_gate uut (.a(a), .b(b), .y(y));
9
10      initial begin
11          for (test_vector = 0; test_vector < 4; test_vector = test_vector + 1) begin
12              {a, b} = test_vector;
13              #10;
14              case (test_vector)
15                  2'b00: $display("a=0, b=0 => y=%b", y);
16                  2'b01: $display("a=0, b=1 => y=%b", y);
17                  2'b10: $display("a=1, b=0 => y=%b", y);
18                  2'b11: $display("a=1, b=1 => y=%b", y);
19              endcase
20          end
21          $finish;
22      end
23  endmodule
```

Self-Checking Testbench

- An automated intelligent testbench
- DUT outputs are captured and compared with reference (expected) values
- Verification is done directly at the command line
- Minimal user interaction required
- Does not depend on manual waveform inspection

Self-Checking Testbench (With assert)

```
1   `timescale 1ns / 1ps
2
3   module and_gate_tb;
4       reg a, b;
5       wire y;
6
7       and_gate uut (.a(a), .b(b), .y(y));
8
9       initial begin
10          a = 0; b = 0; #10; assert (y == 0) else $error("Test failed for 0,0");
11          a = 0; b = 1; #10; assert (y == 0) else $error("Test failed for 0,1");
12          a = 1; b = 0; #10; assert (y == 0) else $error("Test failed for 1,0");
13          a = 1; b = 1; #10; assert (y == 1) else $error("Test failed for 1,1");
14
15          $display("All tests passed!");
16          $finish;
17      end
18  endmodule
```

Using File Input (Reading Test Vectors from a File)

```
1   `timescale 1ns / 1ps
2   module and_gate_tb;
3       reg a, b;
4       wire y;
5       integer file;
6
7       and_gate uut (.a(a), .b(b), .y(y));
8
9       initial begin
10          file = $fopen("test_vectors.txt", "r"); // Open file
11          if (file == 0) begin
12              $display("Error opening file");
13              $finish;
14          end
15
16          while (!$feof(file)) begin
17              $fscanf(file, "%b %b\n", a, b);
18              #10;
19              $display("a=%b, b=%b => y=%b", a, b, y);
20          end
21
22          $fclose(file);
23          $finish;
24      end
25  endmodule
```

Test Vector File (test_vectors.txt)

```
0 0
0 1
1 0
1 1
```

Using Random Stimulus

```
1   `timescale 1ns / 1ps
2
3   module and_gate_tb;
4       reg a, b;
5       wire y;
6       integer i;
7
8       and_gate uut (.a(a), .b(b), .y(y));
9
10      initial begin
11          for (i = 0; i < 10; i = i + 1) begin
12              a = $random % 2;
13              b = $random % 2;
14              #10;
15              $display("a=%b, b=%b => y=%b", a, b, y);
16          end
17          $finish;
18      end
19  endmodule
```

Using FSM

```
1   `timescale 1ns / 1ps
2
3   module and_gate_tb;
4       reg a, b;
5       wire y;
6       reg [1:0] state;
7
8       and_gate uut (.a(a), .b(b), .y(y));
9
10      always @ (state) begin
11          case (state)
12              2'b00: {a, b} = 2'b00;
13              2'b01: {a, b} = 2'b01;
14              2'b10: {a, b} = 2'b10;
15              2'b11: {a, b} = 2'b11;
16          endcase
17      end
18
19      initial begin
20          for (state = 0; state < 4; state = state + 1) begin
21              #10;
22              $display("State=%b | a=%b, b=%b => y=%b", state, a, b, y);
23          end
24          $finish;
25      end
26  endmodule
```

Using Function Call

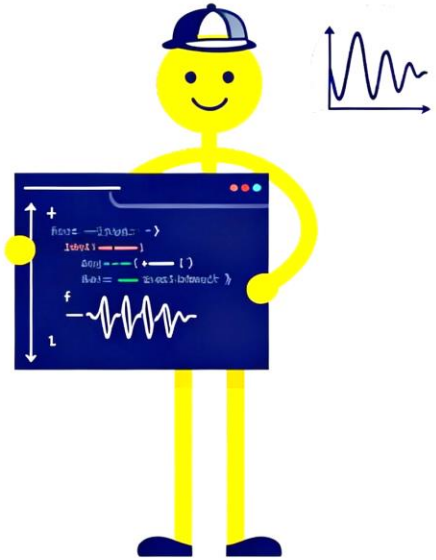
```
1   `timescale 1ns / 1ps
2
3   module and_gate_tb;
4       reg a, b;
5       wire y;
6
7       and_gate uut (.a(a), .b(b), .y(y));
8
9       function bit expected_output(input bit x, input bit y);
10          expected_output = x & y;
11      endfunction
12
13      initial begin
14          repeat (4) begin
15              {a, b} = $random % 4; // Random inputs
16              #10;
17              if (y !== expected_output(a, b))
18                  $error("Mismatch: a=%b, b=%b => y=%b (Expected: %b)", a, b, y, expected_output(a, b));
19              else
20                  $display("a=%b, b=%b => y=%b (Correct)", a, b, y);
21          end
22          $finish;
23      end
24  endmodule
```

Summary

- Verilog testbenches are crucial for simulation and debugging
- Help identify functional issues before synthesis
- Can be extended with advanced verification techniques
- Efficient testbenches improve design reliability

Thank You!

Verilog Testbenches



Your efforts make verification easier !

Thank you !

Happy Learning