Writing Testbenches in Verilog

Robert Li

1. What is a Testbench

Simulating a design is to apply some input signals as stimuli to the design unit that is under test (i.e. DUT or UUT), and then verify the output against the expected results. With ModelSim-Intel, the input signals can be fed in by graphic waveform input files. With Questa-Intel, testbenches are required to provide input signals to the DUT.

A testbench is actually a Verilog file in which you instantiate the DUT and declare enough variables to provide stimuli to the DUT and to get the output from the DUT. It is much like those of your design modules that contain an instantiated module, but since the testbenches are for simulation, not for synthesis, some techniques used in testbenches cannot be used for synthesizable designs.

2. An Example of a Testbench

Suppose we are required to design hardware, an 8 bits register which has the following ports and requirements:

1) reset_n: Input control signal. When resent_n is 0, the register is reset to 0.
2) Load: Input control signal. When load is 1, the register load value from the data port and store it into a temporary register.
3) data: Input signal. This port provides the value for the register to load.
4) decrement: input signal. When this signal is asserted, the value of temporary register decrease by 1.
5) Increment: input signal. When this signal is asserted, the value of temporary register increase by 1.
6) positive, negative, zero: three output ports. The status of the temporary register. Whether the value of the temporary register is positive, negative or zero.

Our design in Verilog can be:

```
module reg8 (input clk, reset_n, load, increment, decrement, input [7:0] data
        output negative, positive, zero);

reg [7:0] temp/* synthesis noprune */;

always @(posedge clk)
begin
```

```
        if(reset_n==1'b0)
                temp<=8'b0;
        else if (load)
                temp<=data;
        else if (increment)
                temp<=temp+8'b1;
        else if (decrement)
                temp<=temp-8'b1;
end

assign negative=temp[7];
assign zero=(temp==8'b0);
assign positive=(~negative) && (~zero);

endmodule
```

One testbench for the above design can be:

```
1       `timescale 1 ns/100 ps
2
3       module reg8_tb();
4       //input to the DUT are of reg type
5       reg clock;
6       reg reset_n, load,incr,decr;
7       reg [7:0] init_data;
8
9       //outputs from the DUT are types of wire
10      wire neg, pos, zero;
11
12      reg8 DUT(.clk(clock), .reset_n(reset_n), .load(load), .increment(incr),
13      .decrement(decr), .data(init_data),.negative(neg), .positive(pos), .zero(zero));
14
15      always begin
16              #10 clock=!clock;
17      end
18
19      initial begin
20
21              $monitor($time,  "  rest_n=%b, load=%b, pos=%b, neg=%b, zero=%b, data=%d", reset_n,
22              load, pos, neg, zero, init_data);
23
24              clock=1'b0;
25              reset_n=1'b1;
26              load=1'b0;
27              incr=1'b0;
28              decr=1'b0;
29              init_data=8'b00000100;
30              #20 reset_n=1'b0;
31              $display($time, "<<  coming out of reset >>");
32              #20 reset_n=1'b1;
33
34              #20 taskload;
```

```
35
36                  #20 decr=1'b1;
37
38                  wait(zero);
39                  @(posedge clock);
40                  @(posedge clock);
41                  @(posedge clock);
42
43                  decr=1'b0;
44                  init_data=8'b11111011; // -5
45
46                  @(negedge clock);
47                  //to ensure there is a posedge before load is changed to 0
48                  taskload;
49                  incr=1'b1;
50
51                  wait(zero);
52                  @(posedge clock);
53
54                  $stop;
55                  //cannot use $finish, otherwise Questa will quit.
56          end
57
58          task taskload;
59                  begin
60                          load=1'b1;
61                          #20;
62                          load=1'b0;
63                  end
64          endtask
65
66          endmodule
```

Just as mentioned above, this example testbench is a Verilog file. Since it is for simulation, not for synthesis, it has some differences comparing with the other Verilog design files. These can be seen from the following sections as we go through some details of the above example testbench.

3. Major components of a Testbench

3.1 **Set up time scale**

The first line of the example testbench is to set up the timescale for simulator. It uses the compiler directetive " **`*timescale***" to define the time unit and the precision of the time at which the simulator will round the events down to. In our example, the time unit is defined as 1 nanosecond (*ns*) and the precision is 100 ps. Therefore, the following delay instructions in the testbench, #10 and #20, will cause the simulation to wait 10 nanoseconds and 20 nanoseconds respectively.

3.2 Define the testbench module

Line 3 in the example testbench file defines the testbench module as ***reg8_tb().*** Please notice that for a testbench module, input or output ports are not needed, because the DUT will be instantiated in the testbench, and the input and output signals of the instantiated DUT will be declared within the testbench.

The testbench module name can be anything, but the convention for the testbench file is the DUT name followed by a "*_tb*". In our example, the DUT is named *reg8.v*, so the testbench file is named as *reg8_tb.v,* and its top module is named as *reg8_tb()*.

3.3 Declare variables

Line 4 to Line 10 in the example file are for declaring variables for the module. In the example, all the variables declared are for applying stimuli to the instantiated DUT, and for getting results from the instantiated DUT for being monitored on screen or for being viewed in simulator's waveform viewer.

In testbench, signals applies to the instantiated DUT are usually generated or assigned values using *initial* or *always* blocks. And only *reg* type variables can be assigned values within *initial* and *always* blocks. Therefore, in a testbench module, all the variables that are used to feed signals to the instantiated DUT are normally declared as *reg* type.

In a testbench module, all the variables that are used to get results from the instantiated DUT can declared as *wire* type. The values on the *wires* can then be displayed on screen or be viewed in a waveform viewer.

3.4 Instantiate the Design Under Test

In a testbench, the design unit that is to be tested or simulated needs to be instantiated, then stimuli can be applied to it and testing results can be obtained from it for verifying. In the example, Line 12 and Line 13 instantiate module *reg8* by port name connections.

3.5 Generate Clock Signal

Most designs have a clock signal. To test the design, the clock signal can be generated simply using an *always* block in a testbench. Line 15 to Line 17 in the example file generate a 50 MHz clock signal by toggling the signal every 10ns, making the signal cycle period 20ns.

3.6 Initial block in a testbench

Line 19 to Line 56 is an *initial* block. This is the block to generate stimuli and to control applying individual stimuli to the DUT at certain simulation time, and to determine what signals to be monitored and displayed on screen. It is the major part of this testbench and maybe the major part for most testbenches.

In a testbench module, all the *always* blocks and *initial* blocks are executed concurrently when simulaiton starts. In each *always* or *initial* block, the instructions are executed line after line sequentially.

In this initial block, Line 21 and Line 22 use the $*monitor* , a system task, to instruct simulator to display several variables on screen whenever and every time one of their values is changed.

Line 24 to Line 29 are to assign initial values to 6 variables.

Line 30 is to tell the simulator wait for 20ns, which is a clock cycle, then changes the value for signal *reset_n* from its initial 0 to 1. This is basically to tell the simulator to reset the DUT at about 20ns after simulation starts, because the time consumed to execute Line 21 to Line 29 is very minimal.

Line 31 is to tell the simulator to display on screen the information for at what time reset happen.

Line 32 is to wait another 20 ns, that is about 40ns of simulation time, toggle the *reset_n* signal to stop reset action for DUT.

Line 34: wait for 20 ns, then start the *loadtask*. *Loadtask* is a task that is defined in Line 58 to Line 64. It asserts *load* signal, then wait for 20 ns, and then dessert the *load* signal.

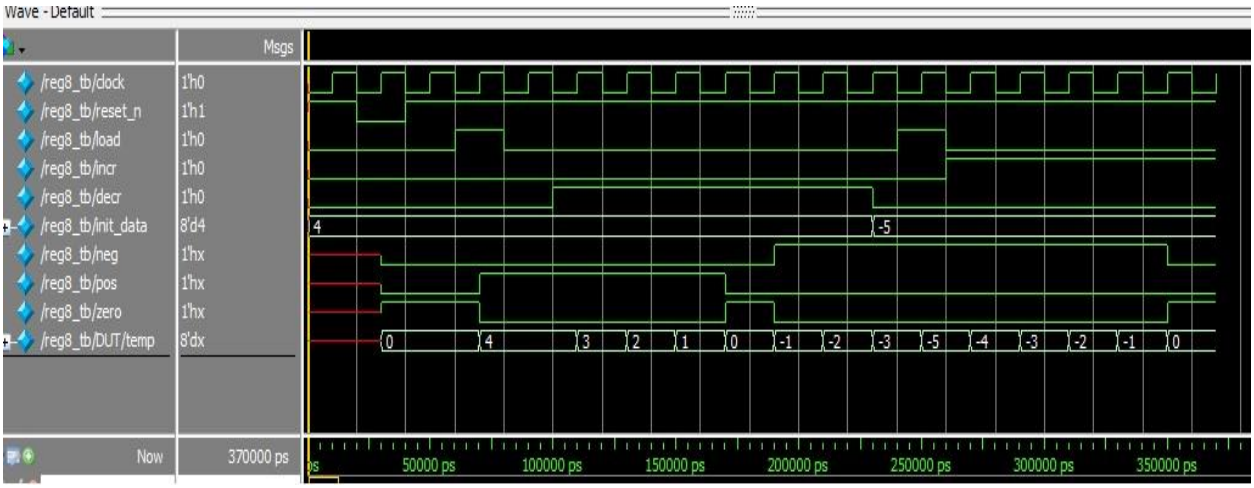
Line 36 is to assert signal *decr*.

Up to this point, the testing plan is: after simulation starts, all the signals are initiated to 0, except the reset signal which is initiated to 1. At about 20 ns, the reset signal is set to 1 to reset the system. Then at about 40 ns, toggles the reset signal to stop the reset action. At about 60 ns, task for loading the initial data start. Loading task lasts about 20ns, then at about 100ns, signal for decrement the temporary register begins. From this moment, the value in the temporary register will be decremented by 1 each clock cycle.

Line 38, and line Line 39 to Line 41: wait till signal *zero* to be 1, and then wait for positive edge of the *clock* signal for 3 times. That is, 3 clock cycles after signal *zero* becomes 1, starts to process instructions from Line 42.

From Line 42 to Line 54, a new value, -5, is loaded to the register and value is incremented by 1 each clock cycle. The simulator wait for signal *zero* become 1 at Line 51, then delay for one more clock cycle at Line 52, finally on Line 54, function *$stop* is called to hall the simulator, put the simulator in interactive mode.

Below is the screenshot of the the simulation result in waveform viewer:



3.7 Time Controls

In Verilog testbench, users have three ways to control timing for signals.

1) Delay control: Use the delay control operator "#" followed by the amount of time unit to delay.
   # (amount_of_time_unit_to_delay);
   The time unit is defined using the compiler directive "`timescale" at the beginning of the testbench. This will the delay the execution of the next instruction for the specified amount of time. To use this control, you need to know the amount of time unit you want to delay first. The amount_of_time_unit_to_delay followed by the "#" can be a literal number, a variable or an expression.
2) Event control: Use the operator "@" followed by the event of a signal transition, i.e., at the positive or negative edge of a signal.
   @ (edge signal1, edge signal2, edge signal3 …);
   This will delay the execution of the next instruction until the specified edge, either *posedge* or *negedge*, of one of the signals in the parenthesis appear.
3) Level control: use keyword *wait* followed by an expression.
   wait (expression);
   This will delay the execution of the next instruction until the expression in the parenthesis is evaluated to be true.

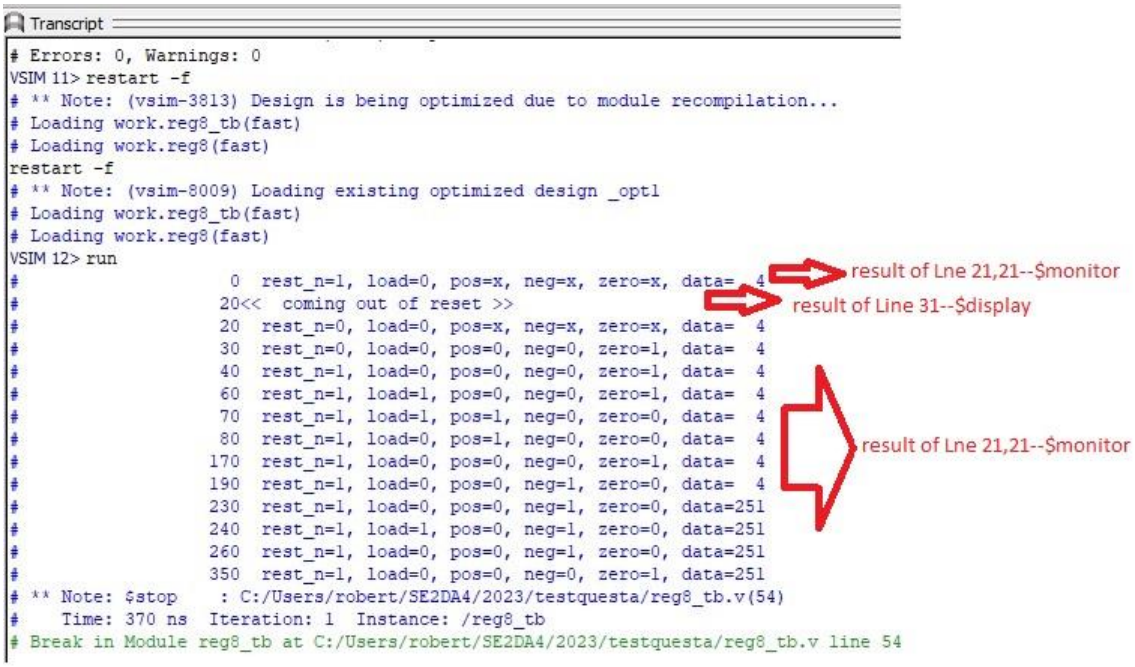
3.8 Display simulation results as text on screen

Many simulators come with a waveform viewer in which the simulation results can be viewed as waveforms. But in case users need to check specific variables, several Verilog system task can be used to display information as text on screen. With Questa-Intel, the following system task can display text in the Transcript widow.

1) **$display**: This system task prints a line each time this task is called. It works much like the printf() function in C, except that $display adds a carriage return and new line character at the end of the line.

2) **$monitor**: It has the same format as $display, but prints a line every time one of its monitored signals changes its value.

   When simulating using the example testbench, the output in the Questa-Intel Transcript window will look like the following screenshot. The way that that $display and $monitor work can be seen from the screenshot.



3) **$write:** This system task works much like $display, except that there is no carriage return and new line character added at the end of the line. This system tasks can be used if users need to print some information on a same line.
4) **$stope:** Works nearly the same as the $display, except that this system task displays the values of its parameters at the end of simulation.

3.9 Tasks

In the example testbench, Line 58 to Line 64 defined a task, and this task is called and used in Line 34 and Line 48.

In Verilog, a task is a group of related codes that would be used repetitively for several times. The task must be defined outside the procedural blocks (i.e., *initial* or *always* blocks), but can be called from within a procedural block. A task is defined by using a pair of keywords *task-endtask*, and its body is enclosed by a pair of *begin-end*.