The Art of Writing Testbenches

Department of Electronics and Communication Engineering
Indian Institute of Technology, Roorkee

ECN 104 Digital Logic Design

Contents

1 Introduction

In the first assignment we studied the basics of Verilog HDL, which we extended in the second assignment and learnt about different modelling techniques. But one thing might still bug you, which is you have designed a module but how to verify it? This is where Verilog testbenches are used, testbenches are verilog module that have sole purpose of testing and verifying the design by simulating it, this can significantly decrease the errors in the design.

2 Types of Verilog Constructs

2.1 Synthesizable Verilog

Anything written such that it can represent a valid circuit will be synthesizable in Verilog, such code is used for designing modules.

Listing 1: Example of synthesizable Verilog code.

```verilog
// andGate4 is a four input AND gate that uses 3 2-input AND
// gates to produce its result. resultTemp0 and resultTemp1 wires
// are used to connect output of two andGate2 to a another andGate2
// This module uses andGate2 module from previous listing.
module andGate4(input0, input1, input2, input3, result);
   // Declare input and output
   input input0, input1, input2, input3;
   output result;

   // Declare wire for connecting gates
   wire resultTemp0;
   wire resultTemp1;

   // Declare instances of 2-input AND gate
   andGate2(input0, input1, resultTemp0);
   andGate2(input2, input3, resultTemp1);

   andGate2(resultTemp0, resultTemp1, result);
endmodule
```

2.2 Non-synthesizable Verilog

Modules written to verify other modules are non-synthesizable, such modules cannot represent a valid digital circuit.

Listing 2: Example of non-synthesizable Verilog code (testbench_1_1.v from Assignment 1).

```verilog
`timescale 1ns/1ps

module testbench_1_1;
   reg [31:0] inputSignal;
   wire [31:0] result;
   reg [31:0]  failureCount;

   initial begin
       failureCount = 0;
       inputSignal = 32'h00000000;
   end

   always #10 inputSignal = inputSignal + 32'h00000111;


   // Initialize module
   module_1_1 uut(.inputSignal(inputSignal), .result(result));


   always @(result) begin
       if (result != inputSignal<<5) begin
           $display("Testcase for input: %d failed, output: %d, expected: %d.",
                    inputSignal, result, inputSignal<<5);
           failureCount = failureCount + 1;
       end
   end

   always @(inputSignal) begin
       if (inputSignal >= 32'h0fffffff) begin
           $display("Test completed, %d failed!", failureCount);
           $finish;
       end
   end
endmodule // testbench1
```

2.3 Common Synthesizable and Non-Synthesizable Verilog constructs

Table 1: List of common operators used in Verilog

Synthesizable	Non-synthesizable
n-Dimensional vectors (e.g. `wire [1023:0] newWire [31:0];`)	Ports having dimensionality greater than 1
All operators	Delay statements
Part select	
If-else, case, casex and casez statements	
wires, regs	

3 Basics of Writing a Testbench

3.1 The clock

Every synchronous design has one or more clocks, these can be initialized by declaring a reg which would hold the state of the clock and changing it at the desired interval, an example of which is shown in Listing 3.

Listing 3: Example clock of time period 15 ns.

```verilog
`timescale 1ns/1ps
// Preceding preprocessing directive is very important when writing a
// testbench, it indicates that all the timings are in steps of 1ns
// and have a resolution of 1ps.

// Timescales in Verilog are specified in the following format:
// `timescale <reference_time>/<precesion>
// e.g.:
//      `timescale 1ns/1ps
//      If a delay statement is used such that, #x.y then the the
//      simulation delay will be x.y ns.
//
//      But the minimum delay you can achieve is log10(x/y), which for
//      this example is 3 (since 1ns/1ps = 1ns/10^-3ns = 10^3).
//
//      Thus, #0.0002 with this example would give you a 0ns delay

module testbench;
   // Declare a new clk as reg
   reg clk;

   // Instantiate the module to test
   testModule uut(clk, ...ports...);

   // Initialize the clock to a value
   initial begin
       clk = 1'b0;
   end

   // Flip the clock every 7.5 ns
   always begin
       #7.500 clk = ~clk;
   end
endmodule // testbench
```

Another and the preffered way to initialize a clock in Verilog is using the forever block, which is described in Listing 4.

Listing 4: Example clock of time period 15 ns using forever statement.

```verilog
`timescale 1ns/1ps

module testbench;
   // Declare a new clk as reg
   reg clk;

   // Instantiate the module to test
   testModule uut(clk, ...ports...);

   initial begin
       clk = 1'b0;              // Initialize the clock
       forever begin
           #7.5 clk = ~clk;
       end
   end
endmodule // testbench
```

3.2 The I/O

For complete testing of a design you'll have to simulate the inputs to the module and verify its output. Inputs to a module are declared as `reg` while the output are declared as `wire`.

Inputs to a module are always changed from a procedural block, this is why they are declared as `reg`, while the outputs are just a connection from the output port of the module under test to your testbench. It is important to note here that output of a module can be left unconnected, but it is almost always a good idea to verify every output of a module.

Listing 5 shows how to declare I/Os of a module.

Listing 5: Example code showing how to initialize and read I/Os of a module.

```
`timescale 1ns/1ps

module testbench;
   reg clk;
   reg [3:0] inputA, inputB;     // Declare 2 4-bit regs for input
   wire resultA, resultB;        // Declare 2 wires for the output

   // Initialize our hypothetical module
   testModule uut(clk, inputA, inputB, resultA, resultB);

   initial begin
       clk = 1'b0;
       forever begin
           #10 clk = ~clk;       // Declare a clk with T = 20ns
       end
   end

   // Note that a module can have multiple initial blocks
   initial begin
       inputA = 4'b0000;
       inputB = 4'b0000;
   end

   initial begin
       #10
       forever begin
           #320
             inputA = inputA + 4'b0001;
       end
   end

   initial begin
       #10
       forever begin
           #20
             inputB = inputB + 4'b0001;
       end
   end
endmodule // testbench
```

In the last example, all the initialization and stimulation was done using `initial` block, however if stimulating inputs using initial blocks is getting cumbersome, you can use `always` block too. One important point to note here is that `always` block in a testbench starts executing simultaneously with all other `initial` and `always` blocks while simulating. Example of using `always` block for simulating a design is provided in Listing 6.

Listing 6: This listing shows how to use always block for simulating inputs to a module.

```
`timescale 1ns/1ps

// Example testbench for an asynchronous 4-bit circuit which produces
// a 4-bit result, the design shifts the input left by 1 bit.
module testbench;
   reg [3:0] inputA;
   wire [3:0] result;
```

```

    testModule uut(inputA, result);

    initial begin
        inputA = 4'b0000;
        forever begin
            #10 inputA = inputA + 4'b0001;
        end
    end

    always @(result)
      if (result != inputA << 1'b1) begin
          // Dislay statements are used to display text to console
          // during the simulation, they are often helpful for
          // debugging and gathering simulation information.

          // You can read more about them here:
          // http://bit.ly/verilogDisplay
          // or a more detailed explanation from:
          // http://verilog.renerta.com/source/vrg00013.htm
          $display("Design failed for input %d!", inputA);
      end
endmodule // testbench
```