Testbench Tutorial

Introduction

A testbench is a Verilog or SystemVerilog module that instantiates and exercises the actual hardware you wish to simulate. Although it is written in Verilog, it uses features of the language that make it unsuitable for synthesis to an FPGA in Quartus Prime. It is only for use within a Modelsim simulation. Compared to using a series of `force` commands in Modelsim, using testbenches has many benefits, including the ability to automate the creation of a large number of test cases and to have two-way interaction between the testing code and the hardware being simulated. This document walks through the creation of testbenches for two sample projects: an 8-bit ripple carry adder, and an 8-bit serial adder.

1 Ripple Carry Adder Testbench

Our first example will be a testbench that verifies that an 8-bit ripple carry adder module, called `add8`, works as intended. It has two 8-bit inputs, and a 9-bit sum output. We start with this simple example because the circuit is purely combinational and there are no clocks involved.

Fundamentally, a testbench is a module that instantiates the hardware you wish to test, and stimulates its inputs in some way. It has no input/output signals of its own – any signals must be generated by the code within. We begin with a simple version of the testbench that just drives the adder with a single test case:

```
module tb(); // no inputs or outputs
  logic [7:0] dut_x, dut_y;
  logic [8:0] dut_out;

  // Instantiate the Design Under Test (the adder module we're simulating)
  add8 DUT ( .x(dut_x), .y(dut_y), .sum(dut_out) );

  // Drive its inputs
  assign dut_x = 8'd5;
  assign dut_y = 8'd7;
endmodule
```

There is no new Verilog syntax here yet. In fact, this looks just like any other module, except that there are no inputs or outputs. Here is what the simulation output looks like. The amount of time you run this simulation for is irrelevant, because the adder generates its output instantly. As you can see, the adder gives the correct result. If it did not, we'd dig deeper to find out why, and fix it.

1.1 Driving Multiple Cases

We'll now replace the two `assign` statements with a more elaborate block of code that automatically drives *all* 65536 possible combinations. Each test case will be provided an (arbitrary) 5ns apart, because otherwise they would all happen at once. For this, we need new Verilog syntax that only works in a Modelsim simulation and not in real hardware.

```
‘timescale 1ns/1ns  // Set default units for delay statements
module tb();
  logic [7:0] dut_x, dut_y;
  logic [8:0] dut_out;

  add8 DUT ( .x(dut_x), .y(dut_y), .sum(dut_out) );

  // initial block: execute this code only once, starting at the beginning of time
  initial begin
    for (integer x = 0; x < 256; x++) begin
      for (integer y = 0; y < 256; y++) begin
        dut_x = x[7:0];
        dut_y = y[7:0];

        #5;  // Wait 5ns (ns because of the timescale directive)
      end
    end
  end
endmodule
```

Rather than `assign` statements, the inputs are now driven from an `initial` block, which executes its contents much like a standard computer program (hence why this won't work in Quartus). Two `for` loops iterate over all possible inputs. We use separate `x` and `y` variables for the loop, rather than the `dut_x` and `dut_y` signals directly, because otherwise neither loop would actually reach the value 256 and would go on forever. The `integer` datatype is simply a shorthand for `logic [31:0]` and has enough bits for this.

There are still a few issues with this testbench that prevent it from being super useful. First, while all 65536 cases are indeed tested, you still need to manually go over each result and inspect it yourself in the ModelSim wave window, which is impractical. Second, it's not obvious how long to run the simulation for. Sure, you can multiply 65536 by 5ns, but we'd like to avoid that. With the above testbench code, the simulation will keep running forever after its last input, continuing to show the same values.

1.2 Automatic Correctness Verification

We'll address those shortcomings in the final version of the testbench for this adder. What we're going to do is make the testbench calculate the *expected* correct answer of the adder by itself (using the + operator), and then compare it to what the adder module outputs. If there's a mismatch, we can actually print a message into the simulator console giving details about what failed, and stop the simulation. This is accomplished using *system functions*, another simulation-only Verilog feature. These are Verilog functions that begin with the $ sign. Here's the new `initial` block of the testbench (everything else remains the same).

```
initial begin
  for (integer x = 0; x < 256; x++) begin
    for (integer y = 0; y < 256; y++) begin
      logic [8:0] realsum;
      realsum = x + y;

      dut_x = x[7:0];
      dut_y = y[7:0];
      #5;
```

```
      if (dut_out !== realsum) begin   // Not a typo
        $display("Mismatch! %d + %d should be %d, got %d instead",
                 x, y, realsum, dut_out);
        $stop();
      end
    end
  end

  $display("Test passed!");
  $stop();
end
```

The `$display` system function behaves just like `printf` does in C, with placeholders that are replaced by actual parameters following the format string. You can use `%0d` instead of `%d` to fix the strange way ModelSim indents the values when printing. The `$stop` function ends the simulation, meaning that you no longer have to pick a specific amount of time to simulate for with the `run` command. Instead, you can use `run -all`.

Note that you can also view the `realsum` signal in the wave window. Since it's a local variable, it will be found in the `sim` window inside several nested entries that have names like `#anonblk#` and `#ublk#`. The signal itself will then be found in the Objects tab.

The `!==` operator lets you compare against `'x'` and `'z'` values.

2 Serial Adder Testbench

The next testbench will also be for an 8-bit adder, but this one operates one bit at a time and requires a clock and reset signal. You need to generate these within the testbench. Also, since the circuit being tested is synchronized with a clock, you must provide inputs that are also synchronized with the clock. While that can be accomplished by continuing to use the #delay statements, there exists syntax that makes this simpler.

2.1 Module Description

Before describing the testbench, we will briefly introduce the hardware that is to be tested. It is an 8-bit adder that, internally, generates the sum one bit per clock cycle, and outputs the full 9-bit sum along with a *done* signal indicating its completion. A *go* signal is input to tell it to accept the two 8-bit inputs and start computation. Below is the module header – this is all we need to know, along with the meaning of the signals, to write the simulation. Full source code for `add_serial` is available with the supplementary files.

```
module add_serial
(
  input clk, input reset,
  input [7:0] i_x, input [7:0] i_y,
  output [8:0] o_sum,
  input i_go, output logic o_done
);
```

2.2 Generating Clocks

A clock is simply a signal that flips its value every half-period. We can already accomplish this with the new syntax we've learned:

```
logic clk;
initial clk = 1'b0;
always #10 clk = ~clk;
```

An `initial` block sets the clock to a known value at the beginning of simulation. Note that multiple `initial` blocks are allowed in your testbench, as long as they don't try to drive the same signals. They all start running at time=0 and run in parallel. An unconditional `always` block loops its statements forever. The statement being looped is `#10 clk = ~clk`, which waits 10ns and then flips the clock, yielding a 50 MHz signal.

2.3 Waiting and Synchronizing

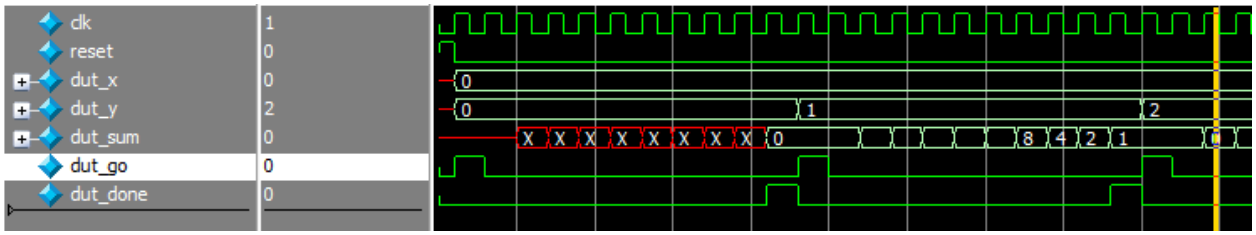
There are statements that let you wait for the next positive (or negative) clock edge, and wait for arbitrary conditions to be satisfied before continuing. The serial adder circuit takes some number of clock cycles to finish before raising its 'done' signal, and we can simply wait for this condition to be true.

```
@(posedge clk);   // Waits until the next positive clock edge
@(negedge clk);   // ... or negative clock edge
wait(dut_done);   // Waits for dut_done == 1'b1
```

2.4 Testbench Code

```
`timescale 1ns/1ns
module tb();
  // Generates a 50MHz clock.
  logic clk;
  initial clk = 1'b0;     // Clock starts at 0
  always #10 clk = ~clk;  // Wait 10ns, flip the clock, repeat forever

  logic reset;
  logic [7:0] dut_x, dut_y;
  logic [8:0] dut_sum;
  logic dut_go, dut_done;
  add_serial DUT
  (
    .clk(clk), .reset(reset), .i_x(dut_x), .i_y(dut_y),
    .o_sum(dut_sum), .i_go(dut_go), .o_done(dut_done)
  );

  initial begin
    dut_go = 1'b0;  // Start with the go signal off
    reset = 1'b1;   // Start with reset on
    @(posedge clk);
    reset = 1'b0;   // Leave it on for a clock cycle and then turn it off

    for (integer i = 0; i < 256; i++) begin
      for (integer j = 0; j < 256; j++) begin
        dut_x = i;      // Set up adder inputs
        dut_y = j;
        dut_go = 1'b1;  // Activate go signal
        @(posedge clk);
        dut_go = 1'b0;  // Turn it off after 1 cycle

        wait(dut_done); // Wait for adder to finish - sum is valid now
        // Add code to validate the sum here, etc
        @(posedge clk); // Wait 1 cycle before sending next inputs
      end
    end
  end
endmodule
```

Here is what the first few testcases look like in the simulator wave window. You can now see visually the effect of the code above: how the reset signal disappears after the first positive clock edge, how the go signal only lasts 1 cycle, and how it waits for the done signal to be 1 (for an entire cycle) before doing the next test case.



Supplementary Files

The full source code for the testbenches *and* both adder implementations is provided as supplementary files to this document. Here is the complete sequence of ModelSim commands required to run each testbench:

```
cd <path to source code>
vlib work
vlog *.sv
vsim -novopt tb
add wave /tb/*   # See testbench inputs/outputs
add wave -group adder /tb/DUT/*  # Optionally, see inner workings too
run -all
```

In your own testbenches, if at this point you need to fix something, recompile, and simulate again:

```
vlog *.sv   # Recompile all files, just one file, or run a 'do' script to do it
restart -f  # Resets simulation back to time=0, keeps list of wave signals
run -all    # Run again
```