Sequential Testbenching Tips & FAQs

This handout is a work-in-progress! Any suggestions or tips to improve it are welcomed.

As you will see in labs 3-5, testbenching is an immensely useful tool for debugging sequential logic. In labs 1 and 2, you may have been able to verify your designs through exhaustive print statements, but for labs 3-5 you will need to use waveforms to verify your designs.

As you experienced in labs 1-2, your testbenches consist of applying inputs (usually referred to as “stimulus”) and checking the correctness of the resulting outputs and/or internal wires (usually referred to as “checks”). In previous labs, we were able to check most of the cases by iterating over all possible combinations, but for labs 3-5, this will not be a feasible strategy since sequential designs now depend on the timing of the inputs. You will need to be much more careful and specific about the particular test cases you decide to run. In particular, you will need to design testbenches that provide good “coverage” of the possible input space without actually testing every single possible input. **Your testbenches are only as good as your stimulus!**

You’ll get a sense of how to come up with good stimulus that exercises the “interesting” test cases as you progress through labs 3-5. As a rule of thumb, you’ll want to apply inputs in your testbench, then check the waveforms to verify that the outputs are as expected (in value and in time) based on the inputs. In addition to outputs, you will also want to check that internal signals are behaving as you expect as well. **Don’t underestimate the usefulness of checking internal signals!** Internal signals can tell you a lot about the behavior of your system, including whether state transitions are happening correctly, whether outputs are correct for a given state, and whether internal signals are being propagated to outputs correctly.

When writing your sequential testbenches, you should follow the format given in lecture 5, slide 17. You’ll want to use an initial forever block to set the clock. Then, in another initial block, you’ll want to apply reset, then apply stimulus. At the end of the second initial block, use $stop. The testbench from lecture 5, slide 17 has been duplicated here for your convenience:

```
module test_fsm1 ;
  reg clk, rst, carew ;
  wire [5:0] lights ;
  traffic_light tl(clk, rst, carew, lights) ;

  // clock with period of 10 units
  initial
       forever
       begin
       #5 clk = 1 ; #5 clk = 0 ;
       end
  // input stimuli
  initial begin
       #10 rst = 0 ;               // start w/o reset to show x state
       #20 rst = 1 ;               // reset
       #10 rst = 0 ;               // remove reset
       #30
       // Your stimulus goes here
       #80                         // 8 more cycles
       $stop ;
  end
endmodule
```

Finally, some common errors when writing sequential testbenches:

1) First, make sure that the **clock starts running before anything else happens**. To be safe, let the clock run for a few cycles before applying stimulus.
2) Make sure that your **reset signal is properly being toggled high, then low** before you apply any other stimulus. Applying stimulus before reset goes high then low can lead to abnormal behavior. To be safe, let reset be high for a few cycles, then low for a few cycles, then start your testcases.
3) When signals in your testbench change, they should change **only on rising edges of the clock!** Changing signals on falling edges (or anywhere else) can cause strange behaviors, such as transitioning through 2 states in a single clock, or skipping states, etc.
4) When dealing with waveforms in GTKWave (lab 4-5) for sine_reader.v, analog waveforms will display as unsigned by default. This can cause correct analog waveforms to appear incorrect. You will need to change the display mode from unsigned to signed on every testbench run to avoid this.

Important Final Note

In labs 4 and 5, you can save yourself a lot of time by getting comfortable with writing good testbenches and using the simulator outputs to debug your code. The final synthesized result, once loaded onto the fpga, is a black box and will tell you almost nothing about what you did wrong if it's not working. We sometimes see groups fall down a rabbit hole because they don't want to spend time writing good testbenches (or even using full testbenches that we provide). Instead, they get stuck in a loop synthesizing their design and then debugging solely based on the output from the FPGA, and then synthesizing again, and so on. ***In the testbenches, you have access to literally every single wire in your design which will shorten your debugging time by orders of magnitude.*** *While this may seem pedantic, we can't stress enough the importance of using your testbenches to the fullest extent in this class. If you find yourself stuck in the endless synthesize-debug loop, step back, take a deep breath, and write a good testbench instead. We promise it'll be worth it.*