18-240 Fundamentals of Computer Engineering

Carnegie Mellon

Lab 2a - Tools for Debugging and Testing

Issued: 16 Sep 2010

1. Objective

The purpose of lab 2 is to further develop your combinational logic circuit design, debugging and testing techniques, specifically for larger circuits and with the help of proper tools and methodology. This is Part A of a two-part Lab2 and is to be done during the first week (Week1: 21 - 23 September). Part B is the logic design part and will run for the two weeks after the midterm; it will be handed out separately.

Twenty points of lab 2 will come from **this in-lab assignment**. You will need to perform the task prescribed in this handout and **get checked off during the Week 1 lab period**. Also report on the activity in a one-page write-up (due at the start of the Week 2 lab period).

We are assuming you are proficient in the lab skills used in Lab 0 and Lab 1. If you are still having trouble running a simulation or mapping a design to an FPGA, you will have a hard time finishing in time. You need to catch-up before you show up to lab.

2. Part I: Something Simple to Try—10 points

Figure 1 is the SystemVerilog interface for a BCD to seven-segment decoder module. This module will be useful for board-level output (to be used throughout the rest of the course).

```
module BCDtoSevenSeg
  (output logic [7:0] led,
   output logic [3:0] bar,
   input  logic [3:0] bcd);

...
endmodule // BCDtoSevenSeg
```

Figure 1: SystemVerilog Interface of BCD to Seven-Segment Display

BCD stands for Binary Coded Decimal where each decimal digit is encoded in 4 bits. In BCD, four bits are used to represent the digits 0 through 9. The combinations for 10 through 15 ("A" through "F" in hexadecimal) are not used.

A seven-segment display is one of those ubiquitous LED displays that can show a single digit, composed of seven separate LED elements (and a period, in our case). Figure 2 shows the definition and layout of the segments. The BCD-to-seven-segment decoder takes as input a BCD digit and outputs 8 bits suitable for driving the seven-segment display to display the corresponding digit. This decoder is an entirely combinational circuit -- the truth table has 4 inputs and 8 outputs. The 8 output bits connect to the 8 LED elements such that `led[7]` goes to the 'a' segment, `led[6]` to 'b' etc. Output variable `led[0]` is connected to the decimal point. Each segment is active low, that is, a zero on that output will make the LED glow. For instance, when `led` = `7'b00100100`, the display will show the numeral 2 and the decimal point will be

displayed. However, your module has no input to tell the decoder to turn the decimal point on, so it should always be blank.

You will need to complete the **BCDtoSevenSeg** module. You may use any of the SystemVerilog styles that you have learned -- choose well, as there are easy and hard ways to complete the module. The bar output from the module should be wired such that it is equivalent to the BCD input value. Later, we will connect the **bar** output to 4 separate LEDs in a "bar" display.

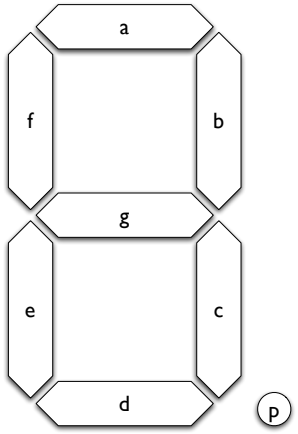

Figure 2: Seven-Segment Display

Figure 3 is the Verilog for our top-level module (top.v). The decoder module, **BCDtoSevenSeg**, is instantiated in this module. Note that we used an alternate method for port specification—look up port specifications under modules in the index of the Verilog book for an explanation. It's just another way to do it. The basic idea is that you can specify connections by giving the names of the two wires to be connected. In the example above, **.bar(barled)** specifies that **barled**, an output of the top module, is connected to the **bar** port of the **MyDisplay** instance of the **BCDtoSevenSeg** display module. Specifying ports this way allows you to specify them in any order in the port list and can save some serious frustration from obscure bugs.

```
module top
  (input  [3:0] switches;
   output [3:0] barled;
   output [7:0] sevenSegmentDisplay;
   output       displayA, displayB, displayC, displayD;
   input        clock, reset);

  logic [7:0] disp;

  BCDtoSevenSeg MyDisplay ( .led(disp), .bar(barled), .bcd(switches));

  LEDController lc(.displayValuesA(disp), .displayValuesB(8'h61),
                   .displayValuesC(8'h63), .displayValuesD(8'h61),
                   .clock(clock), .reset(reset),
                   .displayA(displayA), .displayB(displayB),
                   .displayC(displayC), .displayD(displayD),
                   .sevenSegmentDisplay(sevenSegmentDisplay) );

endmodule
```

Figure 3: Top-level Module

The four-digit seven-segment LED bank on the Spartan board requires a sequential controller that we have provided (on Blackboard). In the above instantiation, the seven-segment decoded **disp** output from **MyDisplay** will drive the right-most seven-segment LED. The remaining three digits of the seven-segment LED, controlled by **displayValues[B,C,D]** input ports to **LEDController**, are set to display the digits "ECE" in this example.

SystemVerilog doesn't support the specification of pin numbers. We connect top-level I/O ports to specific pins using Xilinx PACE. If you treat the TAs nicely, they will explain how to skip the following process, using a UCF file. Otherwise, the 4-bit input **`switches[0]-switches[3]`** should be tied to the dip-switches on the board through pins **`J14, J13, K14, K13`**, respectively; the 4-bit output **`barled[0]-barled[3]`** should be tied to the LEDs through pins **`P13, N12, P12, P11`**, respectively; and the 8 bits **`sevenSegmentDisplay[0]-[7]`** output should be tied to the 7-segment LEDs through pins **`P16, N16, F13, R16, P15, N15, G13, E14,`** respectively. Look for XilinxPinouts.pdf on Blackboard to find the corresponding pin assignments for **`display[A,B,C,D] (D14, G14, F14, E13)`**, **`clock (T9)`** and **`reset (B6)`**. Ugh! Say "Thank You" to your TAs, if they helped you skip this.

Do this and show your TA: Add **`top.v`**, **`BCDtoSevenSeg.v`** and **`LEDController.v`** to a project in Xilinx ISE. Synthesize, download, and demo your design using the techniques from the previous lab.

Do this and show your TA: ISE generates a log after every successful run. This log is displayed in the console output after synthesis. If your console output is not visible, click on View ➞ Transcript. Here's the information provided:

- Number of slices
- **Number of 4 input LUTs**
- Number of bonded IOBs
- Minimum period
- Minimum input arrival time before clock
- Maximum output required time after clock
- **Maximum combinational path delay**

The two important pieces of information we are looking for are the "Maximum combinational path delay," and the "Number of 4 input LUTs." They reflect the two very fundamental metrics of quality in a hardware design---how fast? how small? A 4-input LUT (lookup table) is a reprogrammable 4-input logic function on the FPGA (remember, we can build 2^(2^n) logic functions of n inputs). The combinational path delay represents the critical path (in nanoseconds) through your combinational circuit. Find the combinational delay and the LUT cost for the design above. Compare it to other teams to see how well your **`BCDtoSevenSeg`** implementation works!

3. Part II: Testing—10 points

In digital design, it is essential to develop good testing practices through effective use of HDL language features (e.g., **`$monitor`** in SystemVerilog) and tools (e.g., waveform viewer). In this part, the TAs will hand out a buggy Verilog module for you to test (that's what happens when you outsource your development!). Before continuing, it's worth thinking about several considerations: Do you want to test using the simulator, or using the board?

The majority of your testing should be done in simulation using well-crafted testbenches. Once you use the board, you give up flexibility and visibility of internal signals. However, successful simulation doesn't tell you anything about whether your design actually works correctly or not (a successfully simulated design can still fail on the board). Ultimately, your strategy should begin by testing as much as possible in simulation before starting testing on the board.

How do I test in the simulator?

The **`$monitor`** statement is a good quick-and-dirty way to debug small designs. Look it up in the book—it was in the lectures too. There can only be one **`$monitor`** statement active at any one time in you description. How does it work? Look it up in the book or lecture notes! Or, read this… Or both! The **`$monitor`** looks like this:

```
$monitor ($time,, "A=%b, B=%b", A, B);
```

In the above declaration, when one of the **`$monitor`** inputs (A or B) changes, then the current time and the quoted string will print. In this case the binary values of A and B are substituted in for the %b. (Similar to how **`printf`** works in C.) Interestingly, it prints this at the end of a time period after all changes have occurred and before the simulator goes on to the next simulated time—nothing else can change during this time. Remember, when you have things happening in arbitrary order during a time period, A and B could change in any order. It's nice to know that if they both are going to change during a time period, **`$monitor`** will print only after both have changed to their new value. (Actually, it prints after all changes have occurred in the current time.)

So, what values do you want to monitor? If some of the outputs are incorrect, maybe you want to look at some of the internal values. Are a few modules connected together? Are the connections between them correct? Is one misnamed? Might you want to have the internal values printed out along with the inputs and outputs of the whole circuit? Use **`$monitor`**. When printing, are they the values you expect? Aaarrrggghhhh!—do you know what values to expect?

There is also a **`$display`** command that works similarly to a **`printf`** in C, except that it only prints out its information when the simulator reaches it. There can be as many **`$display`** commands in your simulation as you like. (They go in **`always`** or **`initial`** blocks.)

Testing with a digital waveform viewer.

A very useful tool in digital systems testing is a waveform viewer. A waveform viewer allows you to see the visual output of digital signals in your design. In essence, it operates like the **`$monitor`** statement but can visually convey how signals concurrently change over time. Some of you may have tried out ModelSim's waveform viewer in Lab 0. This is a very powerful way to quickly visualize internal operation of modules during debugging. We recommend using this technique for debugging larger designs.

So how do I test at the board level?

Very carefully. Realize that testing at the board level for us is looking at the FPGA pins to see what value they have (but we don't give you access to the pins—they're too small to deal with in this class!). One way to do this is to connect some of the internal signals to the FPGA pins. Then you can put some inputs on the circuit and see if the internal value is what you expect. Go by the process of elimination. What inputs cause that signal to change? Does it change as expected? If so, the problem isn't there, try a different place. Trace through the circuit to find out what does/doesn't work.

How do you connect internal signals to the FPGA pins? Just like you did above: declare something to be an output of a module and specify a pin for it. Which pins can you use? Look for **`XilinxPinouts.pdf`** on Blackboard to identify the pins corresponding to other unused LEDs.

Approaches

One approach is a constructive approach. Check each piece exhaustively to see if it works. You might want to work from the inputs to the outputs or vice versa. It's an art … maybe they'll make a TV show about testing ICs rather than cops and robbers. Don't hold your breath. It might also help to bring your SystemVerilog handouts.

Do this: Go to the blackboard and run through the Waveform section at the end of the ModelSim Tutorial in Lab 0.

Do this and show your TA: Your TA will hand out a non-working behavioral Verilog model in a single file called lab2.v, along with three compiled Verilog modules. Within the file, there are possibly 1 or 2 "buggy" modules. The modules are interconnected together to implement a specific 4-bit combinational logic function. Read the comments carefully (they describe what the logic is actually supposed to do). A top-level module called "system" instantiates the modules to implement the function. It is your task to systematically test these modules and tell us what the bugs are. Included in the file is a skeleton for a module tester. Note: you should not be changing any of the modules except the tester. Use any of the techniques we just described to test the model. There are one or two bugs in the three submodules in the description. Where are they? Show us what you did to find this out. You'll probably have to add some Verilog code to what we give you to test it. **You don't have to fix the bug.** Just tell us where and what it is by showing it to us on the ModelSim waveform viewer and/or **`$monitor`** output.

Do this and show your TA: Turn in this simulation output (text or waveform output[1] from your Verilog simulator), and the altered Verilog description (the testing stuff you added). Write on this printout where the error is (maybe you could circle it on the printout).

The tasks requiring check off are due by the end of the lab period. It behooves you to start ahead of time and leave plenty of time for the debugging portion of this lab. The buggy design will be given out during lab.

---

[1] To create a screen capture in Windows: 1. select the window by clicking the top border, 2. press Alt-printscreen, 3. open a PowerPoint or Word file and press Cntrl-V.

There is a separate report required for Part A of Lab 2 due at the start of Week 2. Review the "Lab report guide" on Blackboard. The lab report shall include:

- Your Verilog testbench
- Simulation results (text or waveform) that clearly show errors
- A discussion of your testing process. What was your initial testing plan? How did this change as the lab progressed? Discuss what testing techniques were effective and what were not.
- Answers to any questions explicitly asked in the lab handout