EECS150: Lab 4, Debugging & Verification

UC Berkeley College of Engineering
Department of Electrical Engineering and Computer Science

September 19, 2008

1 Time Table

ASSIGNED	Friday, September 19^{th}
DUE	Week 5: September 28^{st} – October 4^{th}, 10 minutes after your lab section starts

2 Motivation

Many of you will be very familiar with the process of debugging software, and thanks to the circuits which you have had to build over the last few weeks, you've all become at least minimally familiar with debugging your own circuits. In this lab you will become acquainted with more formal debugging and verification techniques and tools as we ask you to debug and verify a series of modules.

3 Introduction

No matter how carefully you plan and enter your circuit design, it should always come as a major surprise if it works the first time you try it. The larger and more complicated the design, the larger the fraction of the engineering time you should expect to spend on debugging and verification. In a professional setting, a design would not be considered finished without a complete testing regimen to prove that it works acceptably under all circumstances, a process which can easily consume more than 50% of the time required to implement a design.

In the interest of time, we cut a fair number of corners in this class, for example rather than expecting your design to be fully verified (or even fully debugged), we will expect it to appear to work. This is simply because we do not have time to fully examine your testing regimen. **However it is in your best interest to fully verify your modules.** Most students will simply write a piece of Verilog and synthesize it, hopping that it will work and perhaps wasting hours debugging it inefficiently.

We highly recommend that you consider writing an appropriate and complete testbench an integral part of writing a Verilog module. This will save you many sleepless nights.

3.1 Verification Procedure

There are roughly two steps in the verification process:

1. Perform a test.
2. If the test fails, debug the module being tested.

As such there are two very different parts to the verification process, designing tests and actual debugging. We will discuss debugging in Section 3.2.

Because hardware modules are often very much larger and more complex than pieces of software it is often not possible to fully verify a module. For example a 32 bit adder accepts 2^{64} possible combinations

of inputs, so even if it could be run at 10 GHz it would take nearly 60 years to plug in all possible 2^{64} inputs, even assuming that a matching 32 bit adder could be built to test it against. To make matters worse, most circuits have some kind of memory requiring exponentially more time to test. Because of this exhaustive testing only suffices for the most basic of modules, where it can be run easily.

For more complicated modules, hardware engineers rely on bottom up testing and interface contracts to ensure that the modules which they instantiate work as expected, as do the modules with which they must interact. Over the course of this lab and the remainder of the semester you will become intimately familiar with this style of testing, as it is the only way to produce a fully working design.

3.2 Debugging Procedure

Once you know that something is working properly it is often a relatively trying ordeal to hunt down and fix the actual bug. Below is a formalized algorithm that you can use as a starting point for your forays into debugging.

3.2.1 Hypothesis

Before starting to try and debug a design you must have a clear hypothesis of what the problem might be. Even if your hypothesis is very much wrong you should always have something specific that you are looking for when you start a debugging session. "Whatever is wrong" is not a specific enough goal.

3.2.2 Control

With a hypothesis of what is broken in mind, the next step in debugging is to develop a set of test inputs which will test for the specific bug you expect. Usually developing the test inputs is one of the most difficult parts of the debugging and verification process.

The difference between test inputs for general verification and for debugging is simple: inputs for debugging are meant to aid you in testing your hypothesis, whereas inputs for verification should be designed to elicit as wide a range of bugs as possible.

3.2.3 Expected Output

Before actually beginning a test, it is necessary to figure out what the expected result of the test will be. This should be a simple matter of working through the circuit specification by hand using the test inputs, as developed according to Section 3.2.2.

3.2.4 Observe

With a hypothesis in mind and test outputs and expected outputs in hand it is now time to actually run the test. Unfortunately this is usually a very complicated process, made worse by slow simulation times, complex circuits and the difficulty of examining signals in hardware.

To make this step easier, a testbench or test harness can be developed to look for the expected output and produce more meaningful reports of the success or failure of the test. For example if the test succeeded, all we need to know is that it succeeded, not the how or why of it.

3.2.5 Handling Test Results

Ironically a test which fails is a major success during debugging. If the test succeeds, all that has been proved is that the original hypothesis is false and that there is still a bug in the circuit. However if the test fails, that means that the hypothesis has been proven true and the bug has been found.

When we say that "the bug has been found" we simply mean that it has been further localized, that is to say, we have a better idea of what module or what signal is causing the trouble. Fully specifying the bug and identifying the exact fix may require several iterations of this debugging algorithm and many hours of work beyond the first test.

Always be sure that you know exactly what the bug is and have a well designed fix before modifying your code! Making random changes until the problem disappears will simply prolong the problem and frustrate you!

3.3 Types of Debugging (Parts of this Lab)

In this lab, we will introduce you to four specific types of debugging, all of which you will likely be obligated to use during your time in this class.

Bottom Up Testing (see Section 5.1).

- You will take advantage of the hierarchical structure of a design, testing the lower level modules first and moving towards the top step-by-step.

Designing Test Hardware (see Section 5.2).

- Rather than simulating this circuit, you will perform much faster testing using carefully designed test hardware.

Exhaustive FSM Testing (see Section 5.3).

- You will feed a stream of inputs to a Finite State Machine in order to completely map its functionality and draw a bubble-and-arc diagram.

4 Prelab

Please make sure to complete the prelab before you attend your lab section. **You will not be able to finish this lab in 3hrs otherwise!**

1. **Read this handout thoroughly.**
   - Pay particular attention to Section 5 as it describes what you will be doing in detail.
2. **Examine the Verilog** provided for this weeks lab.
   (a) You should become intimately familiar with the **Lab4Part1.v** file as **you will need to debug it.**
   (b) Make sure to read the **Count.v and Register.v** modules in Section 5.2 as you may wish to use them.
3. **Write your Verilog ahead of time.**
   (a) You will need **three separate testbenches** for Part1.
      i. Lab4PeakDetectorTestbench.v, Lab4Comp4Testbench.v and Lab4Comp1Testbench.v
      ii. Refer to **past testbenches** as a starting point.
   (b) **Lab4Part2Tester.v**
      i. **You may need time in lab to debug it.**
      ii. Start with a **timing diagram** and **schematic.**
4. **Prepare your tests** for Section 5.3.
   - Look at the FSM in Figure 3 and try to devise a **sequence of inputs to test it completely.**
5. You will need the **entire 3hr lab!**
   - You will need to test and debug both your Verilog and ours.

5 Lab Procedure

Remember to **manage your Verilog, projects and folders well.** Doing a poor job of managing your files can cost you **hours of rewriting code**, if you accidentally delete your files.