UNIVERSITY OF CALIFORNIA AT BERKELEY
COLLEGE OF ENGINEERING
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ASSIGNED:	Week of 2/11
DUE:	Week of 2/18, 10 minutes after start (xx:20) of *your assigned* lab section.

Lab 4
Debugging & Verification

1.0 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.

2.0 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.

2.1 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 2.2 Debugging Procedure below.

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 32bit adder accepts 2^{64} possible combinations of inputs, so even if it could be run at 10GHz it would take nearly 60 years to plug in all possible 2^{64} inputs, even assuming that a matching 32bit 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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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 2.2.2 Control above.

2.2.4 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.

2.2.5 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!

2.3 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.

1. **Bottom Up Testing:** In this part 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.
2. **Designing Test Hardware:** Rather than simulating this circuit you will perform much faster testing using carefully designed test hardware.
3. **Exhaustive FSM Testing:** 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.

3.0 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 4.0 Lab Procedure 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 **Part2** 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 Part 3
   a. Look at the FSM in Figure 4 and try to devise a **sequence of inputs to test it completely**.
5. You will need the **entire 3hr lab**!
   a. You will need to test and debug both your verilog and ours.

4.0 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.

4.1 Bottom Up Testing

This part of the lab will be entirely in **ModelSim**. You may wish to read the **ModelSim Tutorial** on the course website before jumping in. http://www-inst.eecs.berkeley.edu/~cs150/sp06/Documents.php#Tutorials

You will be testing the three modules that are in the **Lab4Part1.v file**, which together form an **accumulator** very **similar to the one you built in Lab #2**. In order to fully verify that all three modules work, and to save yourself a number of headaches you will be **testing each module separately** as you **move up the hierarchy**.

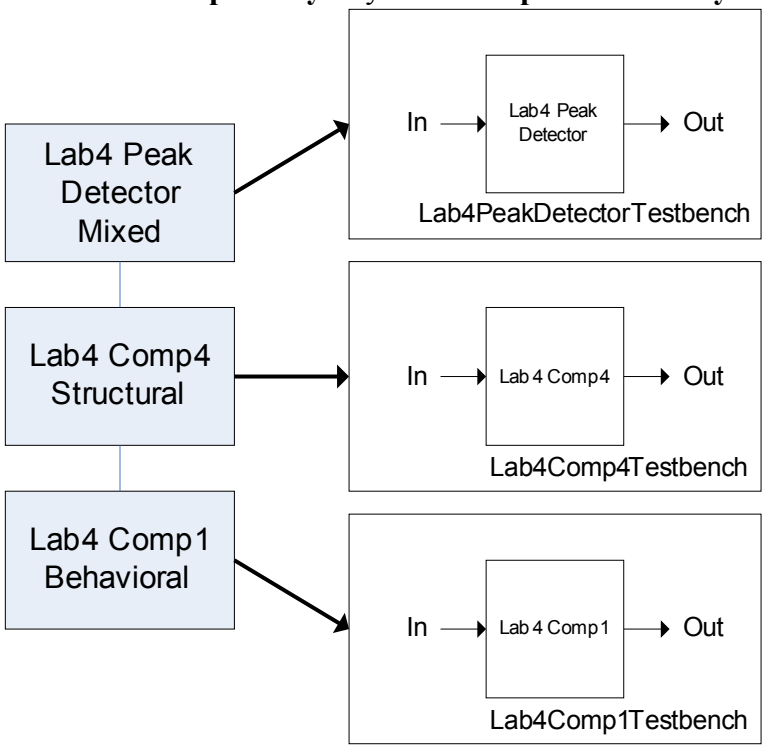

Figure 1: Lab #4 Part1 Module Hierarchy & Testbenches

4.1.1 Lab4Comp1

The first module you will be testing is essentially a duplicate of the `Comp1` module you were asked to build in Lab #2. The main difference is that we asked you to use **structural verilog and primitive gates** in Lab #3, whereas this time we have used **behavioral verilog**. Of course **this version has a bug** which you will need to find and fix before moving on to test the `Lab4Comp4` module.

Signal	Width	Dir	Description
`A`	1	I	The first input
`B`	1	I	The second input
`GreaterIn`	1	I	The `GreaterOut` from the next higher bit
`EqualIn`	1	I	The `EqualOut` from the next higher bit
`GreaterOut`	1	O	Should be `1'b1` whenever `B` > `A`
`EqualOut`	1	O	Should be `1'b1` whenever `B` = `A`

Table 1: Port Specification for Lab4Comp1

Each `Lab4Comp1` module is responsible for **comparing one bit** of `A` to one bit of `B`. In order to generate a useful output however it needs to know the relationship between the **higher order bits** of `A` and `B`, hence the `GreaterIn` and `EqualIn` inputs.

Notice that the `GreaterOut` and `EqualOut` outputs from the least significant bit (bit 0), will yield the correct information for the comparison of **all of the bits of** `A` **and** `B`.

For this module you will perform **exhaustive testing**, meaning that you will try all 2^4 = 16 input values in your testbench. This is feasible because there are **so few inputs and no state registers**.

In order to make your life easier, you should **make use of** `if` **statements** and **the** `$display` **process** in Verilog **to display text errors** any time the **actual output** of the `Lab4Comp1` module **differs from the expected output**. For an example of how to use the $display process, see Figure 3 in section 4.1.3 Lab4PeakDetector below or the **IEEE Verilog Reference**:

https://www-inst.eecs.berkeley.edu/~cs150/ProtectedDocs/verilog-ieee.pdf

4.1.2 Lab4Comp4

With a **fully debugged Lab4Comp1 module** in hand you are now ready to debug the Lab4Comp4 module, which instantiates four Lab4Comp1 modules. This module is again very simple, taking two 4 bit inputs and **reporting if the second is greater-than or equal-to the first**.

Signal	Width	Dir	Description
`A`	4	I	The first input
`B`	4	I	The second input
`GreaterEqual`	1	O	Should be `1'b1` whenever `B` ≥ `A`

Table 2: Port Specification for Lab4Comp4

For this module you will perform **exhaustive testing**, meaning that you will try all 2^8 = 256 input values in your testbench. This is feasible because there are **so few inputs and no state registers**.

In order to make your life easier, you should use a `for` **or** `while` **loop to generate the input** values and `if` **statements** and **the** `$display` **process** in Verilog **to display text errors** any time the **actual output** of the `Lab4Comp4` module **differs from the expected output**. For an example of how to use the `$display` process or `for` or `while` loops, see Figure 3 in section 4.1.3 Lab4PeakDetector below or the **IEEE Verilog Reference**:

https://www-inst.eecs.berkeley.edu/~cs150/ProtectedDocs/verilog-ieee.pdf

4.1.3 Lab4PeakDetector

The `Lab4PeakDetector` module should present no challenges to you at this point. It is a simple module that accepts a new input on ever cycle and outputs the largest input it has been given since the last `Reset`.

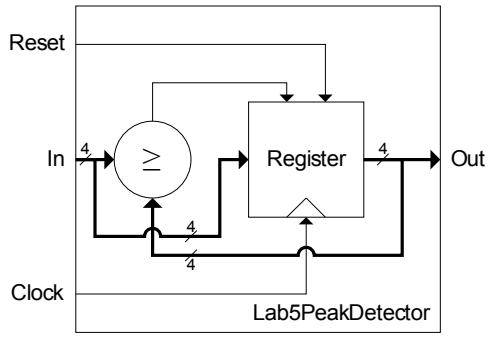

Figure 2: Lab4 Part1 Peak Detector Block Diagram

Since the `Lab4PeakDetector` has **5 inputs and a 4 bit register**, testing all of the possible combinational logic paths would take a **mere 2^9 = 512 inputs**, however **nearly all of the Verilog modules written have significantly more inputs and state information**, making it **impossible to perform exhaustive testing** on these modules.

Therefore in testing the `Lab4PeakDetector` you will use a more advanced testing technique: you will build a testbench that **reads a series of data values from a text file** and **plugs them into the** `Lab4PeakDetector`. This will let you develop more complicated sequences of inputs to perform more **careful, directed testing**.

Figure 3 below is an well commented example of a testbench using the $readmemh process to read hexadecimal test values from a file. Please make sure you understand it. For more information on the `$readmemh` process, please refer to the **IEEE Verilog Reference**:

https://www-inst.eecs.berkeley.edu/~cs150/ProtectedDocs/verilog-ieee.pdf

Testbench.v:

```
// integers can be used to index an array they may not
// be used in synthesis
integer i;

// below is an array of 4-bit values. It contains 16
// elements indexed from 1 to 16. Note that it
// is declared as 'reg', since we assign to it inside of
// initial.
reg[3:0] TestValues[1:16];

initial begin
     // read the file specified and put the values in
     // 'TestValues'
     $readmemh("TestValues.txt", TestValues);

     for(i = 1; i <= 16; i = i + 1) begin
          // Remember to advance the time forward
          #(`Cycle);
          In = TestValues[i];
          $display("In = %d, Peak = %d", In, Peak);
     end
end
```

TestValues.txt:

```
0
A
B
6
```

Figure 3: $readmemh Example Testbench & Data File

4.2 Designing Test Hardware

Because it proves beyond all doubt that a circuit works as desired, we really would like to exhaustively test every single Verilog module that we build or use. However simulation runs at about **1 millionth of the speed of actual hardware**. Coupled with circuits like a 16bit adder, which has 32bits of input requiring 2^{32} = **4 billion test vectors**, this seriously hinders our efforts to exhaustively simulate our modules. Therefore we test circuits like the `Lab4Part2Adder` module, a 16bit adder in hardware, **where at 27MHz, 4 billion tests take a mere 2 minutes, 40 seconds**.

In this part of the lab you **will be designing and building specialized piece of test hardware**, `Lab4Part2Tester`, designed to test the `Lab4Part2Adder` module. In order to make this assignment realistic we have given you an **EDIF black box** for the `Lab4Part2Adder`, namely **Lab4Part2Adder.edf**. This file can be **easily synthesized**, but it **cannot be simulated** and it is **nearly impossible to read**.

To help you design your `Lab4Part2Tester`, the `Lab4Part2Adder` has **four different Fail Modes**. The adder will fail in different ways depending on which Fail Mode you select on **SW9[2:1]**. If the **Fail Mode is 2'b00 (0)**, the adder will **work perfectly**, and in **2'b10 (2)** it will fail on the **inputs 0001, 0001**, reporting that their **sum is 0003**, rather than 0002. This information should help you debug your test harness.

In order to help you we have included the **Register.v** and **Counter.v** files which you may wish to use.

Signal	Width	Dir	Description
`A`	16	I	The first input to the adder (**Shown on DD1-DD4**)
`B`	16	I	The second input (**Shown on DD5-DD8**)
`Sum`	16	O	The sum from the adder (possibly incorrect) (**Shown on DD5-DD8 when SW10[1] is on**)
`FailMode`	2	I	Used to set the fail mode (**From SW9[2:1]**)

Table 3: Port Specification for Lab4Part2Adder

In order to make this a realistic test, **the adder may fail anywhere from 0 to 4 times in each fail mode** (except 0), and you will need to know **how the adder has failed**. Thus your tester must be designed to **pause when it encounters an error** and **then continue after you have recorded the error**.

SW1 should `Reset` your `Lab4Part2Tester` to prepare it for **testing a specific fail mode**. `Go` (**SW2**) should then **start the test process**, allowing it to **free run** until the tester discovers an error. **When an error is encountered**, the tester should **pause and assert the** `Error` **output**. You may then use **SW10[1]** to switch between seeing `A` and `B` and seeing the `Sum` as reported by the `Lab4Part2Adder`. When you have **recorded the error on the Checkoff Sheet**, you should **press** `Go` **again to resume testing**.

Signal	Width	Dir	Description
`A`	16	O	The first input to the adder (**Shown on DD1-DD4**)
`B`	16	O	The second input (**Shown on DD5-DD8**)
`Sum`	16	O	The sum from the adder (possibly incorrect) (**Shown on DD5-DD8 when SW10[1] is on**)
`FailMode`	2	I	Used to set the fail mode (**From SW9[2:1]**)
`Go`	1	I	Signal to start or continue testing (**SW2**)
`Clock`	1	I	System Clock
`Reset`	1	I	System Reset (**SW1**)
`Running`	1	O	Indicates that a test has been started and that not all possible inputs have been tested yet (**Shown on D1-D4**)
`Error`	1	O	Indicates that the tester is paused with an error (**Shown on D5-D8**)

Table 4: Port Specification for Lab4Part2Tester

IN ORDER TO PROPERLY SYNTHESIZE A BLACK BOX, SUCH AS THE LAB4PART2ADDER.EDF FILE WE HAVE GIVEN YOU, YOU MUST TAKE A FEW EXTRA STEPS DURING THE XILINX PROJECT NAVIGATOR PROJECT SETUP.

1. Make sure to add the shell Verilog file (**Lab4Part2Adder.v**) to your project.
2. Set the Macro Search Path
   a. Make sure **FPGA_TOP2.v** is highlighted in the **Sources in Project Box**.
   b. **Right-Click** on **Implement Design** in the **Processes for Source Box.**
   c. Go to the **Translate Properties** tab
   d. Set the **Macro Search** Path to the **directory where your copy of Lab4Part2Adder.edf resides**.
3. Your project should now be able to Synthesize and implement properly.

4.3 Exhaustive FSM Testing

Download the Lab4Part3.bit file to the CaLinx2 board. This will program the board with a very simple circuit, namely the FSM shown in Figure 4 below. You can do this by running the iMPACT directly from the Start Menu (**Start > Programs > Xilinx ISE 6 > Accessories > iMPACT**). In the dialog boxes that appear, select **Configure Devices**, then **Slave Serial Mode** and then open the bitfile file provided.

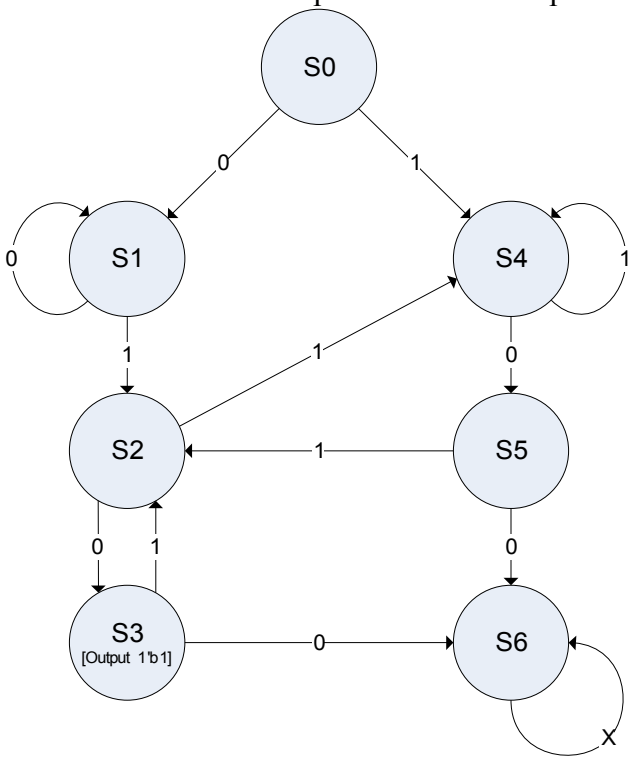

Figure 4: Sequence Detector FSM

This simple FSM is a **sequence detector**, which has the state diagram shown in Figure 4. The circuit receives a 1bit input on every clock cycle and asserts the output when it **detects the sequence 010**, as long as the **sequence 100 has never been received**. If a 100 sequence is received, the circuit halts and the only way to resume normal operation is by resetting it.

The bitfile contains some error, which you should find by **performing an exhaustive test on the state machine**. The idea is to **exercise every arc** and make sure that the **state transition** as well as the **output is correct**.

In order to do this efficiently you should prepare a sequence of inputs that exercises all the arcs and go through it during the test. Preparing this test sequence is not a trivial task and gets exponentially more difficult with the size of the FSM.

To perform the test on the board:

1. The **Input** can be set on **SW9[1]**
    a. The **Input** will appear on **DD7**
2. The **Output** will appear on **DD8**
3. The **State** will appear on **DD1**
4. **SW1** will **Reset** the FSM
5. **SW2** will **Enable** the FSM
    a. The FSM will stay in its current state until you press **SW2**

As you test this FSM, **draw a corrected bubble-and-arc diagram** on the back of your Checkoff Sheet. You will not need to correct the errors in this FSM as we will not be distributing the source code to it.

5.0 LAB 4 CHECK-OFF

ASSIGNED:	Week of 2/11
DUE:	Week of 2/18, 10 minutes after start (xx:20) of *your assigned* lab section.

Man Hours Spent	Total Points	TA Initial	Date	Time
	/ 100		02/ / 07	

NAME		SID		SECTION	

I Bottom Up Testing
- 1 Lab4Comp1 (Testbench & Errors) _____ (10%)
- 2 Lab4Comp4 (Testbench & Errors) _____ (10%)
- 3 Lab4PeakDetector (Testbench & Errors) _____ (10%)

II Designing Test Hardware _____ (40%)

1 Fail Mode 1

A	B	Bad Sum

2 Fail Mode 2

0001	0001	0003

3 Fail Mode 3

III Exhaustive FSM Testing _____ (30%)

1 Draw the corrected FSM Bubble-and-Arc on back of this sheet

RevC – 1/30/2005	Greg Gibeling	Updated to Lab4 Removed Part4 to Lab6
RevB – 7/13/2004	Greg Gibeling	Complete Rewrite of Lab4 Based on the old Lab4
RevA	Multiple	Original Lab4 from Fa02-Fa03 Spring 2004: Greg Gibeling Fall 2003: Greg Gibeling Spring 2003: Sandro Pintz Fall 2002: John Wawrzynek & L.T. Pang