EECS150: Lab 4, Circuit Simulation and Testing

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

1 Time Table

ASSIGNED	Friday, February 12nd
DUE	Week 6: February 23rd – 25th, during your assigned lab section

2 Motivation

This lab will introduce the basics of simulation, an essential tool for debugging and verification. So far, we have lacked the ability to properly test our synthesized circuits before putting our designs onto the FPGA. Although the circuits we have designed up to this point in class are relatively simple, you will eventually be working with larger and larger designs. These may take up to half an hour to just synthesize, place/route, and program onto an FPGA, making the current trial and error testing strategy quite impractical. In this lab, you will learn how to simulate a hardware design and write testbenches, both of which are essential in the verification process of large and complex systems.

3 Simulation in the CAD Flow

In Labs 1 through 3, you have learned how to describe your design in high-level behavioral Verilog and observed its functionality in hardware. Furthermore, you have gained insight into the roles of each tool in the CAD tool flow and better understood the underlying architecture of an FPGA. However, you have also struggled with debugging and testing, due to a lack of visibility of the intermediate signals in your design.

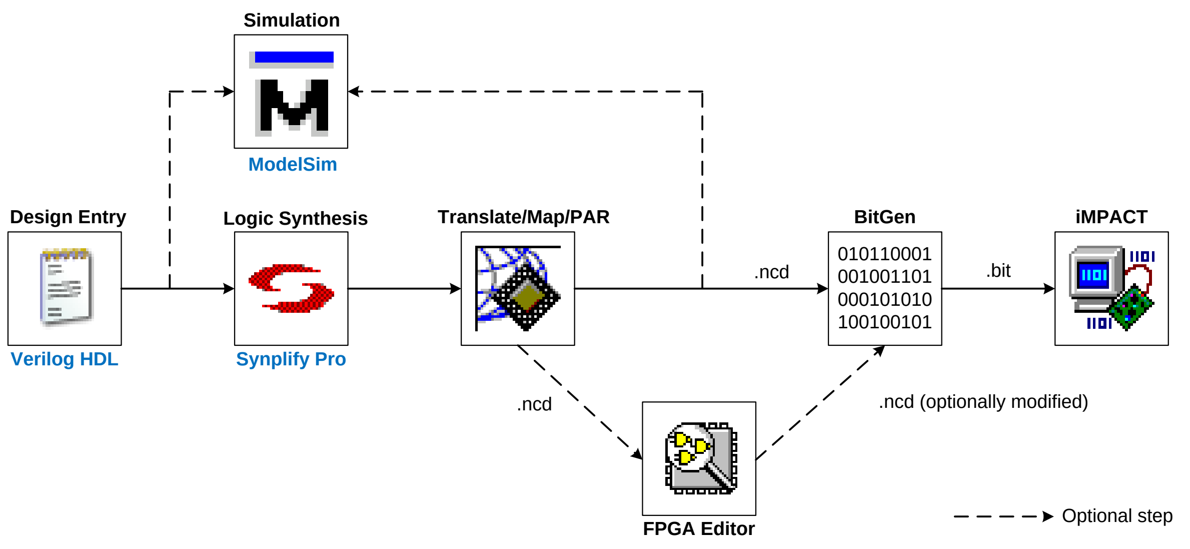
This lab will introduce software **simulation** as a testing and debugging tool. Rather than writing synthesizable Verilog like in Labs 2 and 3, you will instead be writing testbench oriented Verilog which, in general, cannot be mapped into actual hardware. By sacrificing the ability to synthesize the code, however, you gain the ability to use high-level Verilog constructs, such as **for** loops, **while** loops, and even function calls, all to help simplify the simulation and verification process.

4 PreLab

Please make sure to complete the prelab before you attend your lab section. This week's lab will be very long and frustrating if you do not do the prelab ahead of time.

1. **Read this entire handout thoroughly.**
    - Pay particular attention to Section 5 as it describes in detail the circuits and testbenches that you will write and analyze
2. **Examine the Verilog provided for this weeks lab and answer the prelab questions.**
3. **Write your Verilog ahead of time.**

Figure 1 The CAD tool flow



(a) **Lab4BasicTestbench.v**

- Modify the testbench to provide the correct sequence of inputs to make the output of the DUT periodic over 3 cycles.
- See Section 5.1 for details.

(b) **Lab4AdderTestbench.v**

- Write a testbench that exhaustively tests every possible input combination to the broken adder and checks the output against a behavioral model
- See Section 5.2 for details.
- Write a version of this testbench to instead supply a stream of random inputs
- See Section 5.3 for details

(c) **Lab4FSMTestbench.v**

- Write a testbench that will read input sequences from a file and apply inputs automatically to the state machine
- See Section 5.4 for details.

5 Lab Procedure

5.1 Basic Testbenches and ModelSim Tutorial

Lab4Basic.v implements the circuit shown in Figure 2, with a port specification defined in Table 1. Lab4BasicTestbench.v is a testbench verilog module. It is responsible for instantiating the **Design Under Test** (DUT, for short), Lab4Basic in this case, as well as controlling the values of the inputs to the DUT. Please examine the code and complete the prelab section.

5.1.1 ModelSim Tutorial

This part of the lab is designed to acquaint you with **ModelSim** by simulating Lab4Basic through running Lab4BasicTestbench.

1. Launch **Xilinx ISE** and add all the relevant Verilog files.
2. From within **Xilinx ISE**, select **Behavioral Simulation** from the **Sources for:** pull-down (see Figure 3).

Figure 2 Lab4Basic Schematic diagram

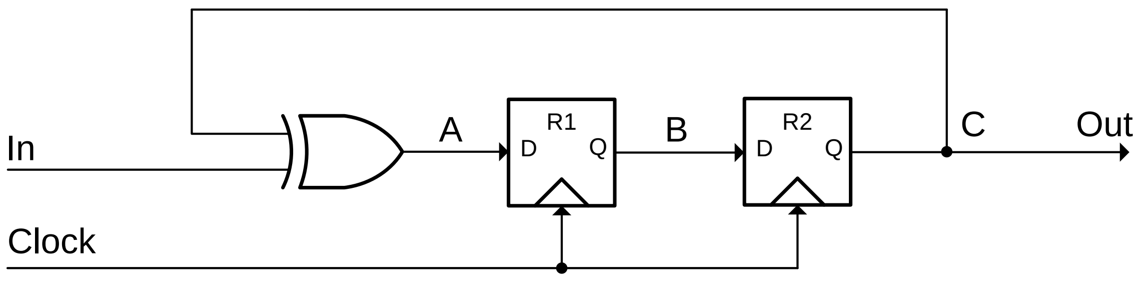
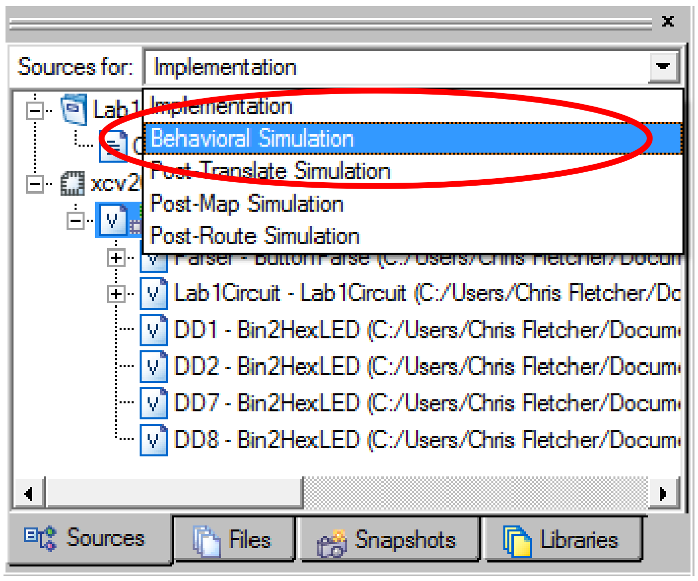


Table 1 Port Specification for Lab4Basic

Signal	**Width**	**Dir**	**Description**
Clock	1	I	The Clock signal
Reset	1	I	The system Reset signal
In	1	I	The input signal to Lab4Basic
Out	1	O	Output signal from Lab4Basic

Figure 3 Change the Sources to Behavioral Simulation


Sources for:
Implementation
Lab1
xcv2
Implementation
Behavioral Simulation
Post-Translate Simulation
Post-Map Simulation
Post-Route Simulation
Parser - ButtonParse (C:/Users/Chris Fletcher/Docum
Lab1Circuit - Lab1Circuit (C:/Users/Chris Fletcher/Dc
DD1 - Bin2HexLED (C:/Users/Chris Fletcher/Docum
DD2 - Bin2HexLED (C:/Users/Chris Fletcher/Docum
DD7 - Bin2HexLED (C:/Users/Chris Fletcher/Docum
DD8 - Bin2HexLED (C:/Users/Chris Fletcher/Docum
Sources
Files
Snapshots
Libraries

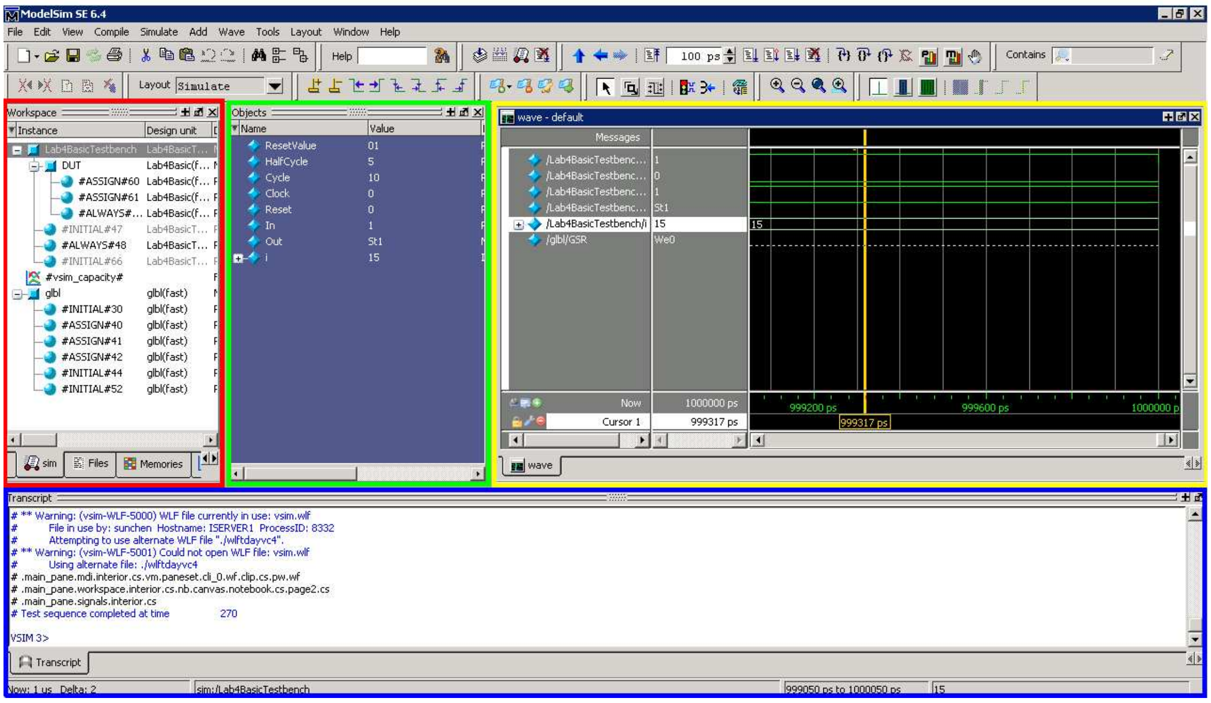
3. Select Lab4BasicTestbench in the **Sources** box.

4. This will change the **Processes For Source** box to show a number of steps involving **ModelSim**. Click on the + next to **ModelSim Simulator** and **Double-click** on the **Simulate Behavioral Model** process.

   (a) It may take ModelSim a few seconds to start up. When it does, you should see something resembling Figure 4.

   (b) ModelSim is composed of multiple windows: the **Transcript Window** showing text messages both from the ModelSim tools and anything printed by the circuit you are simulating, **Wave Window** showing the waveforms from your testbench and any other signals you choose, **Objects Window** which lists the signals in the currently selected module allowing you to drag them to the **Wave Window**, and **Workspace Window** which will let you navigate the tree of modules in your project to change the contents of the **Objects Window**.

Figure 4 The ModelSim simulator. The **Transcript Window** is outlined in blue, the **Wave Window** is in gold. The **Objects Window** is in green, and the **Workspace Window** is in Red



Name	Value
ResetValue	01
HalfCycle	5
Cycle	10
Clock	0
Reset	0
In	1
Out	St1
i	15

```
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim.wlf
#        File in use by: sunchen  Hostname: ISERVER1  ProcessID: 8332
#        Attempting to use alternate WLF file "./wlftdayvc4".
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.wlf
#        Using alternate file: ./wlftdayvc4
# .main_pane.mdi.interior.cs.vm.paneset.cli_0.wf.clip.cs.pw.wf
# .main_pane.workspace.interior.cs.nb.canvas.notebook.cs.page2.cs
# .main_pane.signals.interior.cs
# Test sequence completed at time          270

VSIM 3>
```

5. Like **Synplify Pro**, **ModelSim** will syntax check your code and any errors or warnings will show up in the **Transcript Window**.

6. The **Wave Window** is where you will be spending a good majority of your time in, so let us take a moment and familiarize yourself with it.

   (a) You can click the **magnifying glass buttons** to zoom in or out.

   (b) The signal values listed in the second column are those at the **vertical yellow cursor**.

      i. Move the cursor by simply clicking in the wave window.

      ii. To see the signal values in **hexadecimal** select one or more signals, **right-click** and select **Radix → Hexadecimal**.

   (c) You will probably want to **undock** and **maximize** the **Wave Window**.

7. By default, ModelSim adds only the signals from the testbench to the **Wave Window**. Thus, we will need to add the signals A, B, C from Lab4Basic ourselves.
   (a) Go to the **ModelSim Workspace Window**.
   (b) Navigate the **module tree** to the DUT instance.
   (c) **Right-Click** on that module and select **Add** → **To Wave** → **All items in region**.
   (d) You may also add individual signals by selecting them in the **Objects Window** and adding them in a similar manner.
   (e) The added signals will initially have **No Data**. To get waves you will need to **restart the simulation**. Type **restart -f; run 10us** at the prompt in the **Transcipt Window**.
      - This will restart the simulation and then run it for 10 μs.
8. Look at the **Wave Window** again and complete the timing diagram for Lab4Basic on the checkoff sheet.

5.1.2 Basic Testbench Editing

Modify Lab4BasicTestbench such that the inputs to the Lab4Basic DUT change in such a way that the output signal from the DUT repeats with a period of 3 cycles. Use ModelSim to help you with this.

5.2 Broken Adder Exhaustive Testing

To test the functionality of a purely combinational circuit, we can apply all possible inputs by brute force and check the output for each input combination. This is called **exhaustive testing** and is useful for verifying purely combinational circuits with a relatively modest number of inputs.

In this section, we will apply exhaustive testing by permuting through all possible input combinations of a broken 8-bit adder implemented in Lab4Adder.v. This 8-bit adder will produce the correct result almost all the time, but will output the wrong value under several specific input combinations. The port specification for Lab4Adder is given in Table 2.

Table 2 Port Specification for Lab4Adder

Signal	Width	Dir	Description
A	8	I	Input A to the broken adder
B	8	I	Input B to the broken adder
Sum	8	O	Sum output from the broken adder

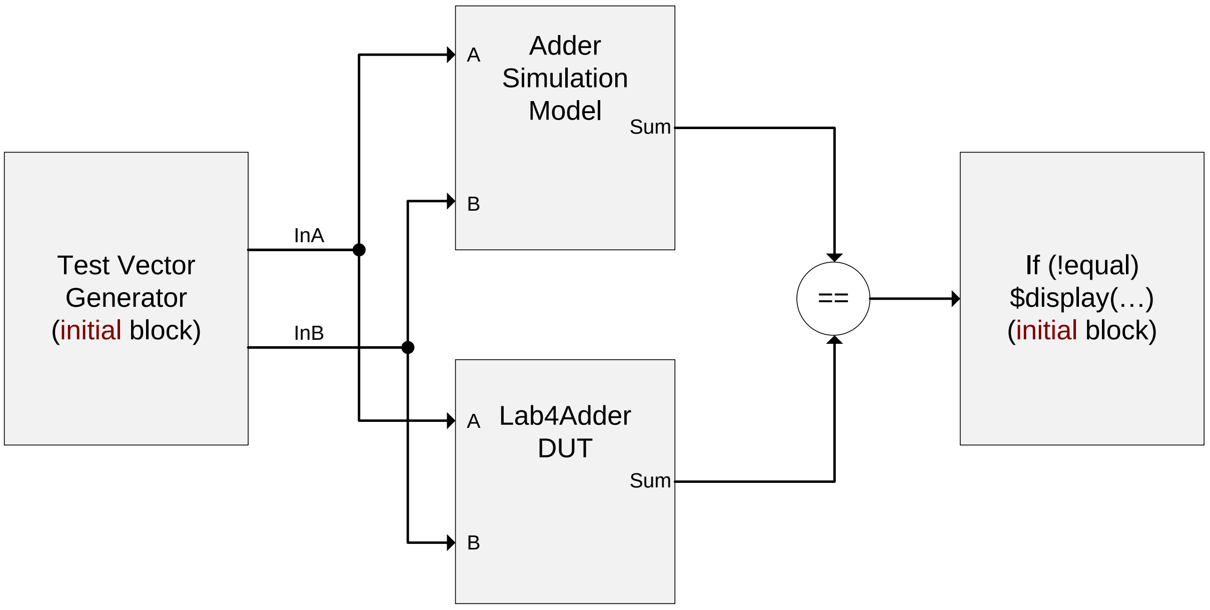
The overall setup of the testbench is shown in Figure 5. Using Lab4BasicTestbench as an example, write a new testbench, Lab4AdderTestbench, that will run through all possible input combinations to the 8-bit adder. Note that you will need to write a simple simulation model of an adder to allow you to check the output of the broken adder with the expected value coming from the simulation model.

In order to make your life easier, you should **make use of** the **$display system function** in Verilog **to display text errors** any time the **actual output** of the Lab4Adder module **differs from the expected output**. After you change each input, **remember to use '#' to advance simulation time by a little bit before you check the output!**

5.3 Broken Adder Randomized Testing

Instead of using a loop to generate every possible input combination, we can also test the adder by generating random test inputs. To generate a random number, you will need to use the $random **system function**. Modify your testbench such that it generates random test inputs. Then, run the simulation until your testbench catches its first error. It may be useful for you to use the $stop **system function** to stop the simulation when you catch the first error.

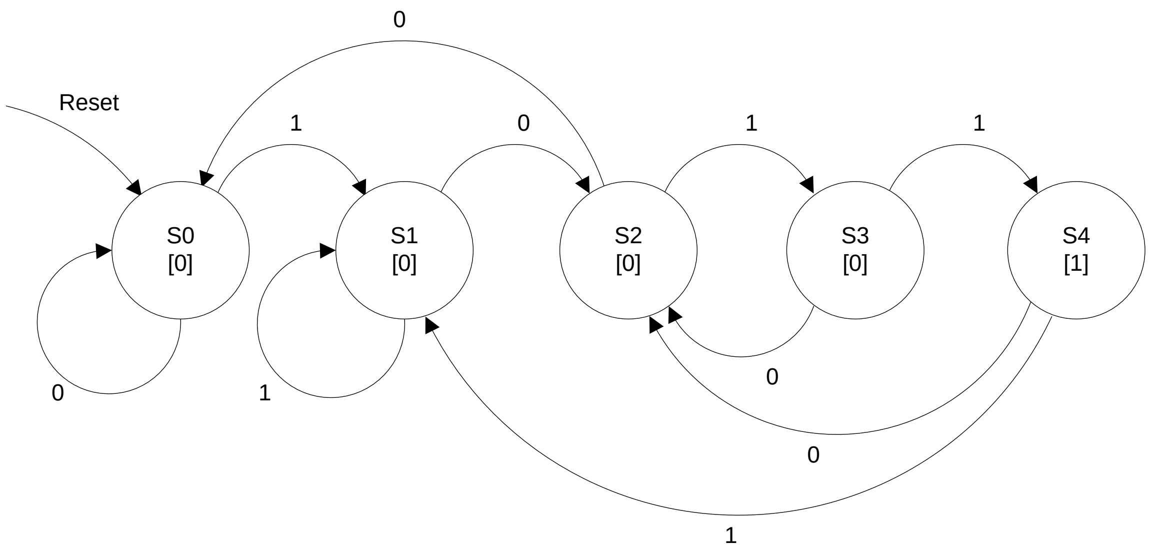
Figure 5 Setup of the Lab4AdderTestbench


Adder Simulation Model
A
B
Sum
Test Vector Generator (initial block)
InA
InB
==
If (!equal) $display(…) (initial block)
Lab4Adder DUT
A
B
Sum

5.4 FSM Mapping

The state transition diagram of an interesting FSM is given in Figure 6. This FSM's function is to output a 1 whenever it sees that the last four inputs were 1011 (if it gets a 1 first, then a 0 on the next cycle, a 1 the cycle after that, and then another 1 on the last cycle, then the FSM should output a 1).

Figure 6 The state transition diagram of the 1011 input pattern detector



In this section, you will be writing a testbench to map out the state transition table of Lab4FSM, an implementation of Figure 6 with incorrect transition arcs. The port specification of Lab4FSM is given in Table 3 and the state encodings are given in Table 4.

Since it is rather tedious to manually set signal values, advance simulation time, then set another signal value over and over again, you will instead read input sequences from a file and have the test-

Table 3 Port Specification for Lab4FSM

Signal	**Width**	**Dir**	**Description**
Clock	1	I	The Clock input to the FSM
Reset	1	I	Reset input to the FSM
In	1	I	The input to the FSM
Out	1	O	Output from the FSM, is a 1 if it sees input sequence 1011
State	3	O	The current state the FSM is in

Table 4 State Encodings for Lab4FSM

State	**State Encoding**
S0	3'b000
S1	3'b001
S2	3'b010
S3	3'b011
S4	3'b100

bench automatically apply the inputs and advance simulation time. In essense, your testbench should automatically perform the following tasks:

1. Read all test input sequences from an input file into a **reg** array
2. Get the first input sequence
3. Apply the Reset signal to the FSM and advance simulation time by one cycle
4. Note the initial state of your FSM
5. Systematically apply each bit of the input sequence to the FSM, remembering to advance simulation time by 1 cycle after each input to allow the state machine to transition to the next text.
6. When finished applying the current input sequence, Reset the FSM and move on to the next input sequence

As an example, if the first input sequence is the 8-bit number 10110010, the testbench should apply a 1 as the input to the state machine the first cycle, 0 on the next cycle, then 1, 1, 0, 0, 1, and finally 0. The testbench should then reset the state machine and apply the next input sequence.

To read input sequences from a file, you will need to need to use $readmemb or $readmemh. The difference between the two system functions is that $readmemb will interpret values from the input file as binary, whereas $readmemh will interpret values as hex. Program 1 shows how to use $readmemb to read the contents of a file into a **reg** array, and then use **$display** to display the contents of everything read, bit by bit. Program 2 shows the contents of a file that was read from.

Using this setup, you will still need to write your own set of input test sequences (start with just one, then add more as you go) and you should use either **$display** or ModelSim's waveform viewer to keep track of the state transitions and help you map out the entire state transition diagram for the broken FSM.

Program 1 $readmemb Example

```
// Integers can be used to index an array, or as a for loop count they may not
// be used in synthesis
integer i, j;

// Declare an array of 8-bit values.  It contains 16 elements indexed 0 to 15.
// Note that it is declared as 'reg,' since we set its value inside of an
// initial block.
reg [7:0] TestValues [0:15];

initial begin
    // Read the file 'TestValues.txt' and put the values
    // in 'TestValueArray'
    $readmemb("TestValues.txt", TestValues);

    //Outer for loop iterates through each 8-bit chunk
    for (i = 0; i < 16; i = i + 1) begin
        //Inner for loop iterates through each bit
        for (j = 0; j < 8; j = j + 1) begin
            $display("TestValues[%d][%d] = %b",i,j,TestValues[i][j]);
        end
    end
end
```

Program 2 An example 'TestValues.txt' corresponding to the $readmemb example. You must **place this file in the same folder as where you made your Xilinx project** or else ModelSim will not be able to find it!

```
10110110
10101011
11010001
11110001
11010101
00110110
```

6 Lab 4 Checkoff

ASSIGNED	Friday, February 12nd
DUE	Week 6: February 23rd – 25th, during your assigned lab section

Man Hours Spent	**Total Points**	**TA's Initial**	**Date**	**Time**
	/100		/ /	
Name	**SID**	**Section**		

1. PreLab .... ________ (25%)
   (a) Lab4BasicTestbench generates a Clock signal at what frequency?

   ____

   (b) Explain what modifications you must make to the testbench to halve the frequency of the Clock signal that it generates

   ____

   (c) While only looking at the provided code in Lab4BasicTestbench.v, complete the following timing diagram in Figure 7 **for only the inputs** (In, Reset) during the first 10 cycles of simulation
2. Basic Circuit and Testbench .... ________ (15%)
   (a) Using ModelSim's wave viewer, complete the timing diagram in Figure 7 for the first 10 cycles of simulation.
   (b) Show your TA the modified testbench that causes the output signal to repeat itself every 3 cycles
3. Broken Adder Exhaustive Test .... ________ (20%)
   (a) Write down both the inputs and outputs of the cases that the broken adder fails for

   ____

   ____

   ____

   (b) Show your TA the testbench you made for the adder
4. Broken Adder Random Test .... ________ (10%)
   (a) How long did you have to run the randomized test for before it caught the first error?

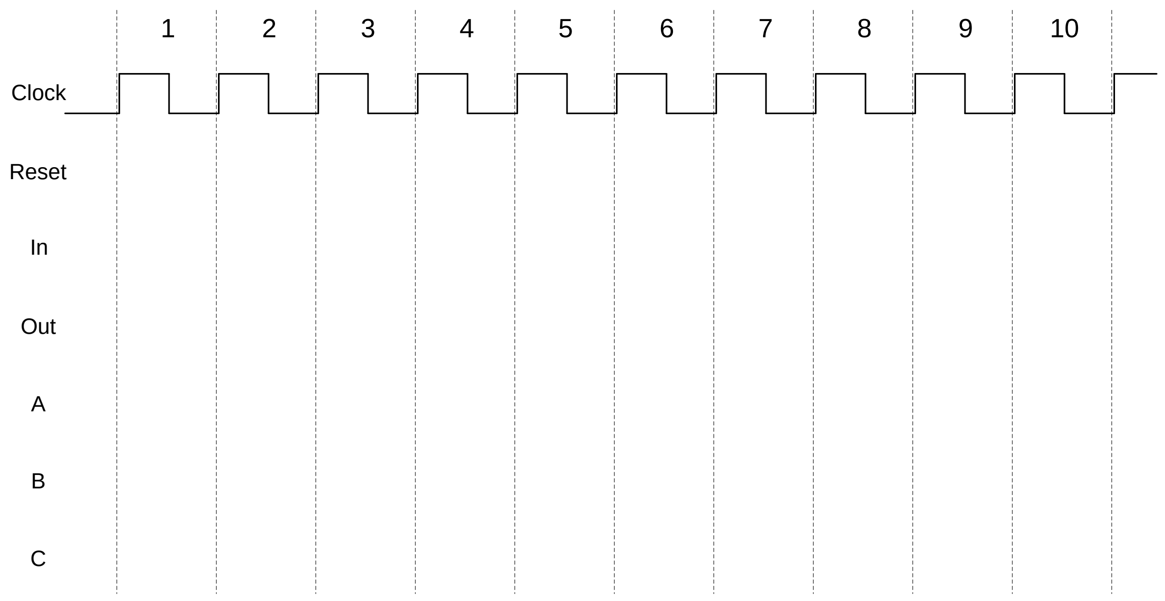
   ____

   (b) Explain to your TA the situations in which a random test might be preferable to an exhaustive test
5. FSM Mapping .... ________ (25%)
   (a) Draw the state transition diagram that the broken FSM implements

(b) Show your TA the testbench that reads input sequences from a file and feeds them to the state machine

(c) What input combination does the broken FSM actually detect?

Figure 7 Fill this out



Rev.	Name	Date	Description
C	Chris Fletcher	2/10/2010	General edits & cleanup for Spring 2010.
B	Chen Sun & John Wawrzynek	2/13/2009	Complete update of Lab 4 from Fall 2002 - Fall 2008, with emphasis on simulation and testing
A	Many Others	-	Lab 4 from Fall 2002 - Fall 2008: Chris Fletcher, Greg Gibeling, John Wawrzynek, Sandro Pintz, & L.T. Pang