

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

# Lab 6

## Debugging

### Objective

You will explore several techniques for debugging a digital design. You will be provided with four different designs, which for some reason don't work as they should, and you will fix the problems by using the techniques described below.

### Introduction

No matter how careful your digital design could be, it is usually very surprising if it works well the first time you try it. The more complex the design, the more likely you would need to spend a big part of the design time in the debugging process. This is also true in industry, even among the most experienced designers. In fact, the design of a system is incomplete without the design of a testing protocol, which is applied to the system at different stages of development in order to detect bugs and correct them. We could roughly describe the debugging process as having two stages: first, a test is performed, and second, if an error is found, the cause of the error is identified and corrected. Therefore, there are two different concepts involved here, one is how to design an effective test that could detect the errors, and another one is how to trace the error back to its cause so it can be corrected.

In this Lab, we will explore some useful techniques for detecting errors and tracing them back to its cause. We will use tests at different levels of abstraction, i.e., simulations and hardware debugging.

The techniques covered in this Lab are the following:

- 1) Bottom-up testing: take advantage of the hierarchical structure of a design by testing first the lower level, simpler components, and moving to the top one level at a time.
- 2) Designing test hardware.
- 3) Exhaustive testing of finite state machines: feed a sequence of inputs to the FSM that guarantees that every arc in the state diagram is exercised at least once, checking that the output and the next state are correct.
- 4) Observing internal signals: connect internal signals, usually buried inside the chip, to an output pin so that you can watch it in the logic analyzer and gain more understanding of the problem than watching only the outputs.

### Prelab

1. As usual, you should read and understand this handout before coming to the lab.
2. Look at the Verilog files provided and understand what they do.
3. Design and sketch out a schematic for a test harness for part 2 of the lab.
4. Become familiar with suggested references and manuals.
5. Prepare your testing methodology for each part.

## Procedure

The purpose of this lab is to identify errors by applying certain testing and debugging techniques, and eventually correct them. The main goal is not to locate the errors but to learn these techniques, which will be very useful during the project development and beyond. **Due to the nature of this lab, please do not discuss it with other students (except your partner) during or after your lab session.** TA's will not tell you where the bugs are, but will help you with the procedure.

### Part I (Bottom-up testing)

This part of the lab will be performed in ModelSim simulator. Create a project and add the file `lab6_p1.v`. This file contains the Verilog model of a peak detector. This circuit has a 4-bit input, a 4-bit output, and the usual clock and reset inputs. The output is equal to the largest value of the input since the last reset, that's why it is called peak detector. The values are coded in binary (unsigned).

The design has three levels of abstraction (Fig. 1). At the lowest level, there is the module `one_bit_comp`. This module compares two bits `a` and `b`, and asserts the outputs `go` or `eo` if `a>b` or `a=b` respectively. In order to connect several of these modules in cascade, two additional inputs `gi` and `ei` are used to indicate whether the most significant bits of `a` are greater than or equal to `b`, respectively.

At the second level, the module `four_bits_comp` instantiates four one bit comparators to create a four bits comparator. The inputs of this module are two 4-bit buses `a` and `b`, and the output `ge` is asserted when `a>=b`.

At the top of the hierarchy, the module `16p1` instantiates the four bit comparator and connects its output to the clock enable signal of a register. The inputs of the comparator are the global input `in` and the output of the register, which is also the output `peak` of the circuit. This interconnection makes the register load the input data if it is greater than or equal to the register contents.

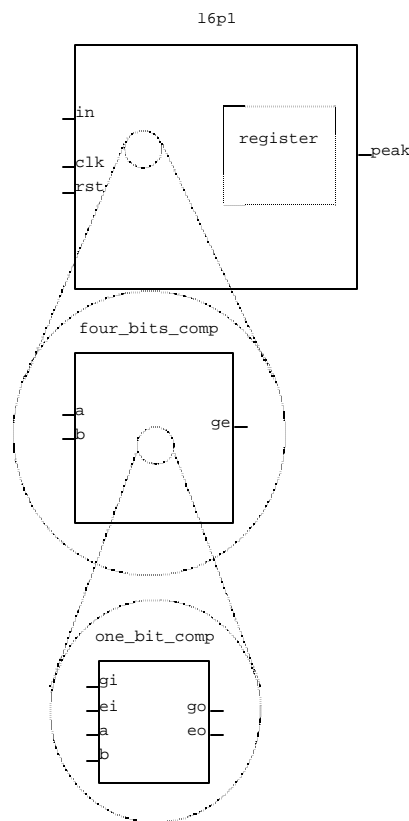


Figure 1: Hierarchic design for Part I.

There are some errors buried in the design. Your task is to detect and correct them. The strategy to follow in this case, is to perform a bottom-up testing, i.e., test first the modules at the bottom of the hierarchy, when you are convinced that they work properly, move to the next level in the hierarchy and test those modules, continuing in this way until you reach the top of the hierarchy. In this part, you can perform the tests with simulations only, i.e., you don't need to implement and download the design to the board. Think about a good sequence of inputs for the tests and create a several test benches to simplify the process.

For this part of the lab, we would like you to develop three types of test benches:

1. Exhaustively test the lowest level module (one bit comparator) and have the test bench output a value every time it differs from the expected. For this, use `if` statement and `$display` verilog process.
2. Use `for` or `while` loop to generate a sequence of inputs. This test bench is best suited for testing the four-bit modules of the peak detector.
3. Read a test vector from a file you construct by using `$readmemb` verilog process. For this test bench write your own test vector with at least 10 input values.

Once the design is working properly, you have to show the TA which errors you found, how you corrected them, and the simulation of the corrected version.

The new verilog constructs are summarized below:

```
// the code excerpt below is annotated with some useful
// information. Please study everything carefully and ask
// questions if you do not understand.

integer i; // integers can be used to index an array

// below is an array of 8-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[7:0] ma[1:16];

initial
  begin
    // read the file specified and put the values in 'ma'
    $readmemb("data_vector", ma);

    for(i=1; i<=16; i= i + 1)
      begin
        // remember to advance the time forward
        #(`step * 4)
        in = ma[i];

        $display("in = %d, peak = %d", in, peak);
      end
  end
end
```

## Part II (Testing an adder on FPGA)

In this part of the lab, you will exhaustively test a 16-bit adder circuit, provided to you in `struct_adder.edf` file. You do not have the source for this component, and in your verilog, it will be treated as a “black box” (look into `struct_adder.v` file to learn how this is done). Study `FPGA_top.v` and `user_top.v` carefully. We already provide input and output signals for you. Your job is to design a test harness that will instantiate the module being tested (`struct_adder`) and will try every combination of inputs possible to

determine whether it works correctly. Your test harness must stop, if it finds a pair of inputs that cause the adder to fail, and it should remain in that state until reset button is pressed.

To help you design your test harness, the provided adder has several “failure modes,” controlled by DIPswitches 9-1 and 9-2. If the switches are set to 00, then the adder works correctly. If they are set to 01, the adder will fail on inputs 2 and 2. This information should help to debug your test harness. With the help of your test harness you will have to determine the input values that cause the adder to fail in modes 2 and 3. This ‘failure\_mode’ is one of the inputs to the adder. Do not forget to connect it.

The two 16-bit inputs of your adder will appear on the eight LED 7-segment indicators in hex form. Look into `user_top.v` for suggestion on how to best use tiny LEDs located by the DIP switches.

Since you are not provided with the source code of the adder under test, it is treated as a black box by the Synplify. You can verify that that is the case if you look in the RTL view and attempt to “push” inside of the `struct_adder` module. In order to implement the test harness and the adder on the FPGA, you must refer Xilinx design manager to the provided EDIF file (`struct_adder.edf`). To do so, before you start implementation process (by pressing the PLAY button), select Design>Options, and “Edit Options...” for implementation. In the dialog box that appears, select Translate tab, and specify the name of the directory where your copy of `struct_adder.edf` lives. This will permit Xilinx tools to find the EDIF file and link your test harness with it.

### Part III (Testing finite state machines)

Download the bitfile `fsm.bit` to the Xilinx board. You can do this by running the iMPACT directly from the Start Menu (Start > Programs > Xilinx Foundation > Accessories > iMPACT). When asked what you want to do first, select “Configure Devices”, then select “Slave Serial Mode” and then open the `fsm.bit` file provided. This file contains the design of a sequence detector, which has the state diagram shown in Fig.2. The circuit receives a 1-bit input synchronized with the clock 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 circuits halts and the only way to resume normal operation is by sending a reset signal.

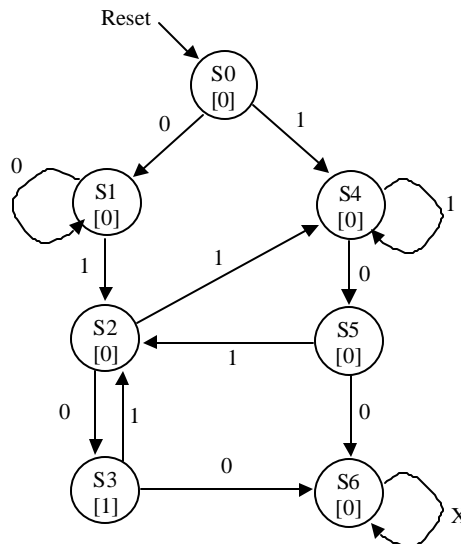


Figure 2: State diagram for Part III.

The bitfile contains a couple of errors, which you should find by performing an exhaustive test on the state machine. The main idea is that you have to exercise every arc and make sure that the state transition as well as the output is correct. Therefore, you should prepare a sequence of inputs that exercises all the arcs and go through it during the test. As you will see, this is not a trivial task; you could deduce that the complexity of the test would grow exponentially with the size of the finite state machine.

To actually perform the test on the board, you will use the dipswitch SW9-1 as the input and the rightmost 7-segment LED. The button SW1 (Reset) will perform the synchronous reset of the state machine, and the button SW2 (Enter) will enable the clock to perform one transition. Additionally, we have wired the state value to the leftmost 7-segment LED, so that you can track the current state (notice that in general the state value is an internal signal that you can't access from outside of the chip, so in real-world applications you may need to modify the code to show those signals and perform the test).

Once you find the errors, you should show them to the TA, together with the sequence of inputs used for the test. You are not required to correct the error, since you won't have access to the source code.

#### **Part IV (Observing internal signals)**

Download a set of wrapper files, which contain the design of an accumulator, similar to the one we've seen in past labs. We've added an edge detector that controls the clock enable signal of the accumulator. Therefore, we can test the circuit manually in the board, one input at a time.

You should first *simulate* the circuit and make sure it works properly. Then you should download it to the board and *test it manually*. You will find that the circuit doesn't work properly. Think about the design and the probable cause of the error. Modify the original source code so that you can look at internal signals in the logic analyzer. Find the signal that's not behaving properly and add a piece of code that corrects the problem.

You have to show the TA the signal on the logic analyzer before the correction. When the problem is solved, you have to show your code and the new design working properly on the board.

To easily look at some internal signal A, you (1) must make sure that it is a wire or a reg (not an input or output), and (2) should add an attribute to its declaration as shown below:

```
wire A /* synthesis syn_probe=1 xc_loc="AU15" */ ;
```

Note, that you must choose an available pin. We suggest the pins in the neighborhood of AV17 and AW17.

## **Acknowledgements**

Original Lab by John Wawrzynek, Gabriel Eirea and Yury Markovskiy.

Name: \_\_\_\_\_

Name: \_\_\_\_\_

Lab Section: \_\_\_\_\_

**Checkoffs: Lab 6**

## 1. Part I.

Show the three test benches you wrote.  
 Show the errors in the code and the corrections.  
 Show the simulation of the corrected design.

TA: \_\_\_\_\_(25%)

## 2. Part II.

Show the diagram of your test harness  
 Demonstrate its operation to the TA.  
 Failure\_mode == 2: A:\_\_\_\_\_ B:\_\_\_\_\_  
 Failure\_mode == 3: A:\_\_\_\_\_ B:\_\_\_\_\_

TA: \_\_\_\_\_(25%)

## 3. Part III.

Show the sequence of inputs used for testing.  
 Show the errors you located in the circuit.

TA: \_\_\_\_\_(25%)

## 4. Part IV.

Show your simulation.  
 Show the misbehaved signal on the logic analyzer.  
 Show the code that corrects the design.  
 Show the corrected version on the board.

TA: \_\_\_\_\_(25%)

**Total Score**

Lab turned in on time

TA: \_\_\_\_\_(100%)

Lab turned in one week late

TA: \_\_\_\_\_(50%)