CS152 – Computer Architecture and Engineering
University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

Compiled: 2004-09-04 for CS152, Prof. Dave Patterson, John Lazzaro
Sources: http://www.cs.berkeley.edu/~kubitron/courses/cs152-S04/handouts/homeworklab_2.html

Mini Lab 2: **Debugging and FSMs**

Name: _____ **Date** _____

Login: _____ **Prelab Checkoff** _____

Section: _____ **Lab Checkoff** _____

Introduction: Mini Lab 2 is the second part of a 3 part mini lab series designed to orient you with the concept of CAD tool flow as well as let you get hands-on experience with industrial strength CAD tools. We will be going over the creation of test benches, and finite state machines.

This lab is to be done individually and is due at the end of your section. You should be able to show us the steps you took, and explain what they mean.

IMPORTANT: Please read the lab report and answer the following questions before coming to lab.

Prelab Questions:

1. **What are the three main components of an FSM?**

2. **Explain the role of the sensitivity list for an always block. What do the two types of always blocks look like and what are they used for?**

3. **What happens in a combinational always block when a signal is not assigned a value every time the always block executes? Why is this undesirable?**

4. **Following the state machine diagram in the lab report, trace the operation of a 4-bit multiplier multiplying 1101 (multiplicand) and 0110 (multiplier). When the product register is shifted, what should be shifted into the most significant bit?**

Lab Questions:

1. **Bug 1 and Fix**

2. **Bug 2 and Fix**

3. **Bug 3 and Fix**

4. **Bug 4 and Fix**

CS152 – Computer Architecture and Engineering
University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

Compiled: 2004-09-04 for CS152, Prof. Dave Patterson, John Lazzaro
Sources: http://www.cs.berkeley.edu/~kubitron/courses/cs152-S04/handouts/homeworklab_2.html

Mini Lab 2: **Debugging and FSMs**

1 References

Computer Organization and Design: The HW/SW Interface (3rd Edition)

- Appendix B-67 on Finite State Machines
- Section 3.4 on Multiplication

Synplify Manual <http://www-inst.eecs.berkeley.edu/~cs152/handouts/local_only/synplify_ref.pdf>

- Pages 8-15 to 8-19 on Combinational Logic
- Pages 8-20 to 8-21 on Flip-Flops
- Pages 8-26 to 8-28 on Synchronous Sets and Resets
- Pages 8-28 to 8-32 on State Machines (note that on page 8-31, the <= assignments should be =)

2 Finite State Machines in Verilog

Finite state machines are made up of combinational logic, which determines its output and its next state, along with a set of flip-flops, which hold its current state.

2.1 Combinational Logic (Next State and Output functions)

In class we showed how combinational logic can be described with assign statements. Combinational logic can also be described with a combinational always block: always@(a1 or a2 or … or aN), where a1 … aN are signals that form the sensitivity list. The always block runs when any signal of the sensitivity list changes. Because we are using the always block to model combinational logic, whose output is a function of its current inputs, the sensitivity list should contain all the input signals so that whenever any input signal changes, the always block runs to update the output.

A common bug encountered with these blocks is an incomplete sensitivity list, which is created when not all input signals are included. For example, the block

```
always@(a)
    b = a | c
```

contains an incomplete sensitivity list because c is missing. This block does not behave as combinational logic, because when c changes, b does not change.

Within a combinational always block, blocking assignments (=’s), are used. This type of assignment behaves like assignments in most procedural languages. If initially a=0, b=1, and c=2, then after the following is executed:

```
a = b
c = a
```

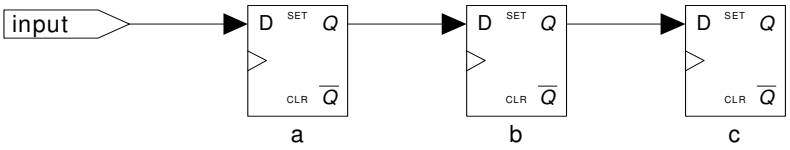
a=1, b=1, and c=1. A is updated with b’s value and then c is updated with a’s new value.

2.2 State (Flip-Flops)

State is described with always@(posedge clk) blocks. This type of always block is only executed on the positive edge of the clock signal. Within these blocks, nonblocking assignments should be used. Nonblocking assignments (<=’s), unlike blocking assignments, perform their assignments simultaneously. For example, if initially a=0, b=1, and c=2, then after the following is executed:

```
a <= b
c <= a
```

a=1, b=1, and c=0. This allows us to describe multiple flip-flops within a single always block:




```
always@(posedge clk)
  begin
    a <= input;
    b <= a;
    c <= b;
  end
```

2.3 FSM Example

In lecture, a stoplight FSM was described. In the following sections, the example will be rewritten in behavioral Verilog.

2.3.1 Module Declaration and State Assignment

In this example, the FSM will be contained within its own module:

```
module trafficLightController(clk, rst, change)
    input clk, rst, change;
    output R, Y, G;
```

To describe three states, instead of fixing an encoding, we will be using constants to make future changes easier

```
    parameter STATE_RED    = 3'b000;
    parameter STATE_YELLOW = 3'b001;
    parameter STATE_GREEN  = 3'b010;
```

2.3.2 State

In lecture, the flip-flops holding state were kept separate from all combinational logic. When using always blocks, simple logic, such as resets, sets, and enables, can be kept within the always@(posedge clk) block to simplify the code.

```
    reg[2:0] state;
    reg[2:0] nextState;
```

```
always@(posedge clk)
    if(rst)
        state <= STATE_RED;
    else if(change)
        state <= nextState;
```

2.3.3 Next State Logic

In this combinational always block, the default case is necessary for the circuit to synthesize correctly. If it were left out, then there would be a path through the always block where nextState is not assigned a value. This means that if state were to change to 3'b111 (an invalid state), then nextState would remain as it were before the change. In other words, the circuit would need to contain memory elements (latches) and would not be combinational.

```
always@(state)
    case(state)
    STATE_RED:    nextState = STATE_GREEN;
    STATE_YELLOW: nextState = STATE_RED;
    STATE_GREEN:  nextState = STATE_YELLOW;
    default: nextState = STATE_RED;
    endcase
```

2.3.4 Output Logic

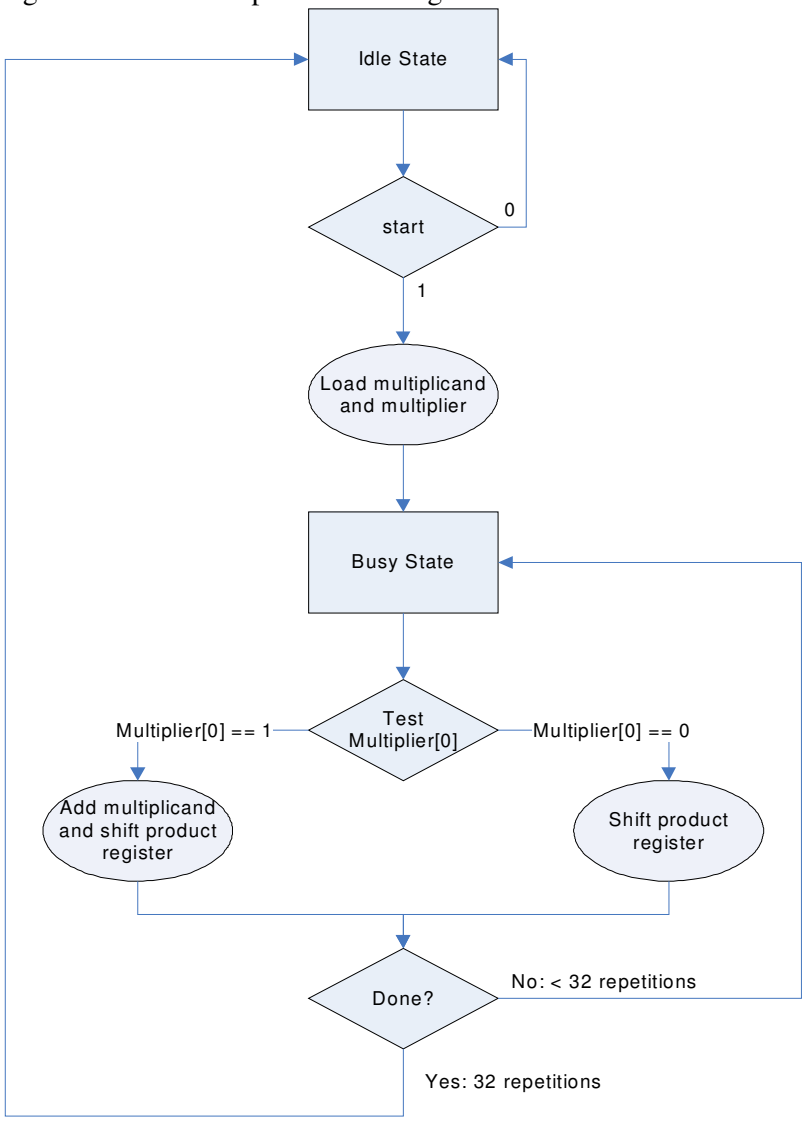
Blocking assignment allows us to assign R, Y, and G to default values and then update them depending on the state.

```
always@(state)
  begin
    R = 1'b0;
    Y = 1'b0;
    G = 1'b0;

    case(state)
    STATE_RED:    R = 1'b1;
    STATE_YELLOW: Y = 1'b1;
    STATE_GREEN:  G = 1'b1;
    endcase
  end
```

3 Multiplication

The type of multiplier you will be debugging is a 32-cycle multiplier, which uses a shared product/multiplier register like the multiplier seen in Figure 3.7 of COD.



4 Lab

In this lab, you will be debugging a completed multiplier in simulation. The code contains a total of 4 bugs. After using test benches we give you to find the first two, you will write a test bench to discover the last two. Start by copying M:\mini_lab2 to your home directory and opening the Project Navigator project file.

4.1 Unit Tests

1. Find and then fix the bug in the adder. Adder_tb.v is the associated test bench.
2. Find and then fix the bug in the product register. Productreg_tb.v is the associated test bench.

4.2 Overall Test Bench

1. Complete the multiplication test bench in multiply_tb.v. Be sure to use both directed and random test vectors.

   This test bench module should include should apply the test vectors to the multiplier, check that the result is correct, and print out the results in the following form:

   number1 * number2 = result
   number3 * number4 = result
   number5 * number6 = result
   number7 * number8 = result

   If there is ever a failure, you should flag it, like this:

   >>>ERROR<<<
   >>>number9 * number10 = result
   >>> Should be: realresult

   You should be able to run this random vector test for a long time, but first debug your test bench and multiplier with a small number of results first. For check off, run a 100 random number loop in addition to a several directed test vectors.

   Hint: You will want to use Verilog system functions and the * operator. There is the $random function to get 32-bit random values, and the $display function to print results. In addition, you may find tasks useful to structure your code and the wait statement useful to wait for the multiplier to finish multiplying. There is a Verilog manual online under the resources section; chapter 17.1 (pg 291) talks about $display, chapter 17.9 (pg 324) talks about $random, chapter 10 (pg 164) talks about tasks, and chapter 9.7.6 (pg 154) talks about wait.

2. Find the remaining two bugs and fix them.