

Computer Science 141

Lab #6: State Machines “The Easy Way”

October 29 / October 30

This lab should be shorter than usual. We will then redo the second half of last week’s lab, but let Verilog do the hard work for us. We will also add an asynchronous reset.

1 State Machines in Verilog

In last week’s lab, you used D-Flip Flops to store state and wrote your own logic functions to decide state transitions. This week, we’ll let Verilog do all that for us!

In last week’s lab, we stored the current state in a few flip flops. You implemented these flip flops using a register. This time around, we’ll just use the registers directly, you don’t need to add in a lot of Flip Flops and wires this time:

```
reg [(x-1):0] state;    // x is the number of bits you need
                        // to store your states
```

Next, for each state you are currently in, you want to figure out what the next state will be. Note that this may be dependent on your current state and/or your inputs. You can draw out a truth table like we did in last Wednesday’s class. When you write your expression out in Verilog, it should be in the following layout:

```
always @( <conditions> ) begin
  case (state)
    <current_state_0>: <decide next state>
    <current_state_1>: <decide next state>
    ...
    default: state <= <default_state>
  endcase
end
```

Note that all this code does is figure out what your next state is from the current state. You were doing this by hand when you wrote logic functions last week, but Verilog will do it for you this time. Note that you can set the next state by using a simple assignment if your next state is only based on your current state. If the next state is dependent on inputs, you'll want to use an if statement or another case statement.

Also, you don't *have* to use a case statement or this particular layout. You may prefer if statements or some combination of case statements and if statements. The basic idea is to use your current state and your inputs to determine your next state.

Here's a simple example:

```
always @(...) begin
  case (state)
    x'd0: state <= x'd1;           // Whenever we are in state 0, always
                                   //   go to state 1, regardless of
                                   //   the inputs

    x'd1: begin                     // Whenever in state 1...
      if(SomeInput == 1'b1) begin
        state <= x'd0;           // ...go to state 0 if SomeInput is true
      end else begin
        state <= x'd1;           // ...otherwise, state in state 1
      end
    end

    default: state <= x'd0         // If we end up in an unexpected state
                                   //   revert to state 0.
  endcase
end
```

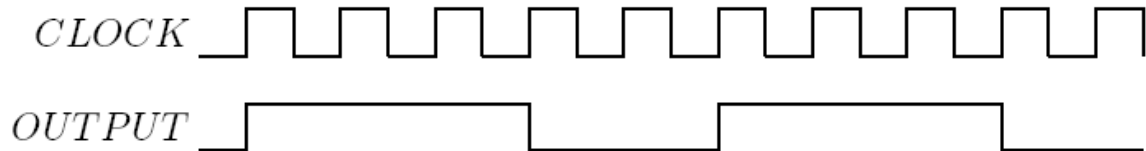
Once your states are working properly, you only need to use assign statements to generate your outputs. You'll definitely want to use your states to decide your outputs; you may want to also use your inputs.

Be sure to also include the default line in your case statement to handle any unintended states. If you use 3 register bits but only use seven states, there's an eighth state that you *could* enter. This may happen when you first start up your board if you do not explicitly specify an initial value for your state. Always use default: when using case statements. It's an excellent habit to have for *any* programming language.

2 Oscillator, Revisited

Here's the assignment from last time:

We will make a oscillating clock that is 1 for three cycles, and 0 for two cycles (and 1 for three cycles, 0 for two cycles, ...). The timing diagram is shown below:



The oscillating module will have two inputs: CLOCK and RESET, and one output: OUTPUT.

You will need the “lab5-template” project again, so unzip another copy from the zip file. When designing your logic, be sure to test your design and make sure it works before trying it out on the board. You will want to use some of the switches and/or buttons and LED segments to interface with your board. Be sure to try out your design by hand with the slow clock and using the oscilloscope with the fast clock.

Finally, be sure to demo your logic, Verilog, and testbench(es) for your TF.

HERE’S THE NEW PART:

First, use the new case statement style to implement this project.

Second, you need to implement an asynchronous reset. You will add an additional RESET input, which will set the output to 0, regardless of state. This is an *asynchronous* reset, so you should revert to a zero input immediately and *not* at the next clock cycle. To do this you will have to modify your always condition a little and add some extra logic for checking the value of the RESET input.

When you demo your project, you will need to see the clock so that you know that your reset is asynchronous, so please display your clock on one of the LED segments. Otherwise, how will you know your reset is asynchronous?

The addition of an asynchronous reset should only require a few extra lines of code, so if you find yourself writing a lot of Verilog, you might want to chat with your friendly TF.

3 Hand In

As always, show your Verilog, testbench(es), and board demo to your friendly TF. Also, make sure your TF has a copy of your project. One day when you are a famous Verilog designer we'll want to post your oscillator code in the Maxwell Dworkin Café...

Updated October 26, 2009, Jian Han