Dylan Thornburg and Alan Rieger
1/16/23

SANTA CLARA UNIVERSITY	**ECEN 122 Winter 2024**	**Hoeseok Yang**
Laboratory #1: A Simple State Machine **For lab section on January 15-29, 2024**		

I. OBJECTIVES

To give you exposure to just enough Verilog to get you going for the rest of the labs.

PROBLEM STATEMENT

We will design and implement a state machine that is an up/down counter. Then we will instantiate this state machine into a larger design that provides the inputs and uses the outputs, so that you can see the system is working in simulation.

ECEN 122 Lab1

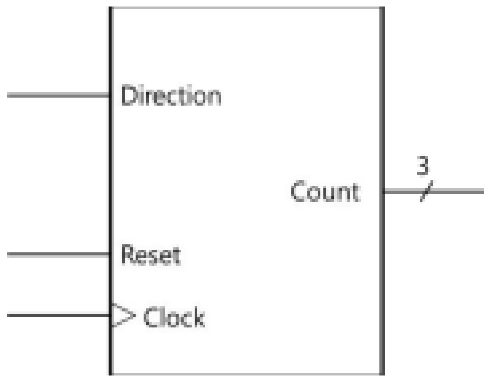
Creating a module

The state diagram above indicates the behavior that we are looking to implement. We can see that there is a single control signal, *x*, that determines whether the counter counts up or counts down. And there is a 3-bit output, *Count*, that cycles between the values 1 and 4, in binary. The 3-bit output of this logic can be determined only by the present state; thus, we know that this state diagram is a Moore machine. Now we need to describe this Moore machine in Verilog.

Verilog is a Hardware Description Language (HDL). Anytime we are going to describe a circuit, we first need to define the boundaries of the circuit, what we would call the interface. Think of drawing a box around the circuit you are describing and being clear about

- o what signals are going in (the inputs) and
- o what signals are coming out (the outputs).

We know the input here is *x*, but maybe we will be a bit more descriptive and choose to call it something like "*Direction*". And we know we have a 3-bit output called *Count*. But state machines also need a clock. (A state diagram does not show this because a state diagram is just enough to express the behavior we are after.) And we are also going to need a *Reset* signal, to make sure we can put our counter into a known state. So, the box around the circuit might look like this:



The first thing we are going to do in Verilog is to define the module interface (input/output).

1. Open a new text file to start writing the Verilog code. By convention we use a suffix of .v for files that contain Verilog code. So, you might call the file *ThreeBitCounter.v* 2. The first line we are going to put in the file will define the actual module itself. It should take the form

module <module name> (<signal name>, <signal name>, <etc.>)**;**

where anything inside <> is a name you define yourself. (I will use bold typeface to indicate keywords that need to be used. You don't need to make them bold yourself.) So maybe we would have

module ThreeBitCounter (Direction, Clock, Reset, Count);

Go ahead and put this in as your first line of code. Feel free to alter the names if you like, as long you continue to use your names throughout the rest of this lab material.[1]

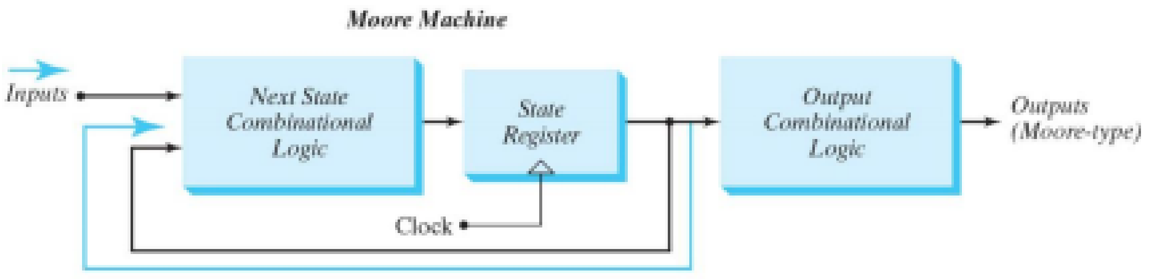
3. Now, even though we have specified four signals on the interface, we have to be more explicit about whether they are inputs or outputs, and whether any of them are buses (as Count is, in our case). So, the next two lines in the file should look something like

input Direction, Clock, Reset;
output reg [2:0] Count;

Since *Direction* and *Clock* are both single bit inputs, they can be defined on the same line using the keyword **input**. Since *Count* is an output, it has to be on a separate line, using the keyword **output.** (Based on how we are going to do our implementation in this lab, it also needs to be defined as type **reg** but we won't get into the reasons for that here.) We also have to define *Count* as a 3-bit bus. The notation [2:0] is used for this. It is specifying a range from 2 to 0 (i.e., the values 2, 1, and 0), for a total of three values, representing three bits. We will then be able to use this notation if we ever want to, for example, reference just bit 0 of *Count* by using the form Count[0].[2]

We have now defined the module name and the interface, so we can move on to defining the actual behavior of the module.

The next thing that is typically done in a Verilog module is to define any signals that are going to be internal to the module itself. Moore state machines look like:



[1] See page 6 and 8 of the Verilog reference for syntax and example of module delaration.

[2] See page 7 of the Verilog reference for syntax of input/output declaration of Verilog modules.

We have our inputs, outputs, and clock signals defined already as part of our interface. But we are going to need signals that keep track of our current state (the output of our state register) and the next state value that we calculate as a function of the current state and the current set of inputs; this next state value will then be captured into the state flops (State Register in the figure) on the next rising edge of the clock.

At this point, we need to make a decision about how we are going to encode the states. We know from the diagram above that we have four states. If we take a fully encoded state approach, we can express four different state assignments using just two bits. So, we are going to define our current state and next state values as 2-bit buses. Specifically, we are going to use the value 0 (00 in binary) to represent state A, 1 (binary 01) to represent state B, 2 for state C, and 3 for state D.

4. [State Register] The next line in the Verilog file is going to use the keyword **reg**. Without getting too much into the subtleties and nuances of Verilog, we are generally going to use this keyword to define internal signal/variable names that we will use in our modules. So, our next line may look something like

   ```
   reg [1:0] currState, nextState;
   ```

   You can use whatever names your like for your current state and next states. But note that the declaration above indicates that these two variables are both 2-bit values.[3]

That should cover all the signal name definitions that we will need. Now it is time to start writing the code that will actually express the behavior we want. Verilog is not a sequentially executed language, so there are a number of ways that the behavior here can be expressed. But we will follow a template that is a common approach. We are not going to get into much explanation for why the code is written the way it is; we are going to rely a bit more on learning by example.

5. [State Register] For readability, we generally put in a blank line after the variable declarations, to offset the rest of the code. The line after that is going to be a directive to Verilog that is used whenever we want to model positive edge-triggered flip-flops, which is our standard approach for maintaining state in our state machines. The line should look like

   ```
   always @(posedge Clock)
   ```

   Note that **always** and **posedge** are keywords, whereas "Clock" is the clock signal that we defined on the interface of our module. There's nothing magical about the name "Clock"; the clock signal could really be called anything. But whatever you choose to call it, you would put that name in this line, after the **posedge** keyword. What this line does is cause Verilog to execute the immediately following line (which we'll get to) whenever there is a positive edge on the signal called "Clock".[4]

[3] See pages 10-12 of the Verilog reference for syntax of Verilog signal declarations.

[4] See pages 34-40 of the Verilog reference for more examples of using the **always** statement.

Now we're going to specify/describe the behavior of the state flops. Under normal circumstances we would want, at each rising edge of the clock, the current state to get updated with what we have determined that we want the next state to be. Using simple flip-flop terminology, we want Q (the output) to reflect whatever is on D (the input). In the context of the **always** statement that we've written, we essentially want to say something like Q=D. In other words, Q gets assigned what value is currently on D. However, if the *Reset* signal is asserted then we want our state machine to go to state A, so we will add a bit of code that will take care of that.

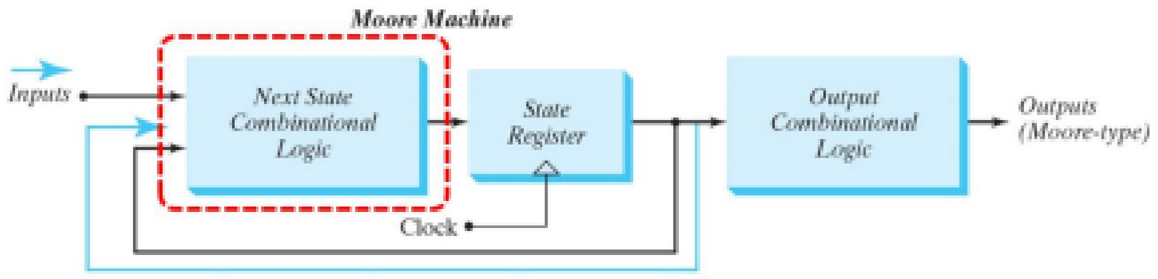
6. [State Register] We will put the following code on the next line, just after the **always** statement that we specified in the previous step. Another formatting convention that we follow is to indent the code that is the subject of the **always** statement. The indentation can either be a few spaces or a tab. Just try to be consistent. The line we need to add is setting the current state to be equal to what has been calculated as the next state, unless the Reset signal is asserted, in which case we need the state to become 0, which is our encoding for state A. Coupled with the **always** statement, so that we can see the indentation, it would look something like

   ```
   always @(posedge Clock)
           if (Reset) currState = 0;
           else currState = nextState;
   ```

   Again, nothing magic about the names *currState* and *nextState*; they are the variable names (identifiers) that we chose to define back in step 4.

At this point we have specified our state register (flip-flops). Now let's turn our focus to specifying

<u>the next state logic</u>:



Before we get into the structure of how to do that in Verilog, let us revisit the state diagram and establish some concepts and terminology around how to interpret what is going on.

When we draw state diagrams, we use lines with arrows (often called arcs) to represent state transitions. The arrows provide direction. They “start” in one state and “end” in another (or sometimes the same) state. And we typically label or annotate them with some condition that must be present in order for that transition to occur. So, if I look at any one arc, I can describe that arc in a way that might look like “if I’m in state W and I see condition Y then I want to go to state Z”, where W is the start of the arc, Y is the condition that labels the arc, and Z is the state where the arc ends.

Looking back at the state diagram that we are trying to describe, let us apply that concept to the two arcs leaving state *A*. One goes to state *B* (in the case of *Direction*==1) and the other goes to state *D*

(in the case of *Direction*=0). The one that goes to state *B* could be described as “if my *currState* value tells me I’m in state *A* and *Direction*==1 then I want to set my *nextState* value to state *B*”. Similarly, the arc to state *D* could be described as “if my *currState* value tells me I’m in state *A* and *Direction*==0 then set *nextState* to state *D*”. I could also combine these two into something that looks like “if I’m currently in state *A*, if (*Direction*==1) then *nextState* = state *B* otherwise nextState = state *D*”.

We could then replicate this same construct for all the arcs leaving all the other states. We’d have a slightly different statement for the case where we’re currently in state *A* versus if we’re in state *B*, or state C, etc. Verilog, and many other programming languages, has a construct called a case statement that matches well with this pattern; we’re checking a specific variable (*currState*) which can have multiple different values, and we want to take a different action depending on what value it currently contains. Which leads us to the actual Verilog structure we will use here.

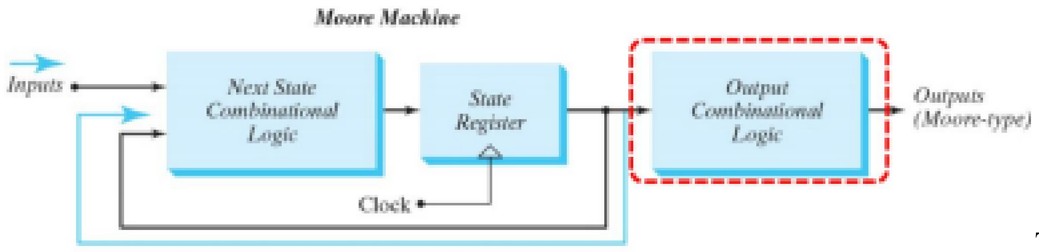
7. [Next State Logic] We are going to start this bit of code with an **always** statement. But since this is just combinational logic it does not depend on the behavior of the clock signal. We are going to use a construct that basically says “execute this code whenever any of the wires in the module change value”. And then we’ll have our case statement, which has the format illustrated here.

```
always @(*)
 case (currState)
```

```
    0: if (Direction==1) nextState = 1 else nextState = 3;
    1: if (Direction==1) nextState = 2 else nextState = 0;
    <2 more lines>
endcase
```

Hopefully you can see the pattern here and fill in the missing two lines.

The last thing we need to do is generate the actual *Count* value outputs:



There are a number of different ways this can be done, but what we'll do here is use a different always block, which has much the same structure as the always block we used in the previous step to specify the logic for nextState.

8. [Output Logic] The idea here is that we are expressing the concept "if I'm in state X I want my output to be Y". Since this is a Moore-style state machine, the output (Count) is not a function of the input (Direction), just what state we're in. Again, just the first few lines are specified. Hopefully you can fill in what's missing.[5]

[5] See page 12 of the Verilog tutorial for examples of specifying Verilog literals.

```
always @(*)
case (currState)
    0: Count = 3'b001;
    1: Count = 3'b010;
    <2 more lines>
endcase
```

A quick note about the fact that both of these always blocks have the same structure. They could have been combined, to generate the assignments for both nextState and Count in the same case statement. If we had done that, we would need to have used begin/end statements to make it clear what needs to happen in each case. Taking that approach would have yielded something that looks like

```
always @(*)
case (currState)
    0: begin
        Count=3'b001;
            if (Direction==1) nextState = 1 else nextState = 3;
    end
```

```
            1: begin
                    Count=3'b010;
                            if (Direction==1) nextState = 2 else nextState = 0;
            end
                    <code for the remaining two cases>
        endcase
```

That covers everything that needs to be specified. The last thing that needs to be done is to define the end of the module

9. Specifying the end of the module is done with the **endmodule** keyword. So, a line or two after the code we've written so far, we would just have

```
        endmodule
```

Putting this all together you should have something that looks like

```
module ThreeBitCounter (Direction, Clock, Reset, Count);

        input Direction, Clock, Reset;
        output reg [2:0] Count;

        reg [1:0] currState, nextState;

        always @(posedge Clock)
        if (Reset) currState = 0;
        else currState = nextState;
```

```
        always @(*)
        case (currState)
                0: begin
                        Count=3'b001;
                        if (Direction==1) nextState = 1; else nextState = 3;
                end
                1: begin
                        Count=3'b010;
                        if (Direction==1) nextState = 2; else nextState = 0;
                end
                2: begin
                        Count=3'b011;
                        if (Direction==1) nextState = 3; else nextState = 1;
                end
                3: begin
                        Count=3'b100;
                        if (Direction==1) nextState =0; else nextState = 2;
                end
            endcase
        endmodule
```

Instantiating the counter module in a larger design and simulating

Now that we have our design written up in Verilog, we'd like to see if it actually works as intended in simulation.

The tool we are going to use in this course is called Vivado. From Lab 0, we already know how to simulate a Verilog design in Vivado. Use the provided test bench (tbThreeBitCounter.v) to run your simulation.

Extending the concept

Once you have the basic counter functionality working, see if you can modify the design to do the following:

- Replace the single control input "*Direction*" with two new control signals: *Start* and *Done*.
- Consider state 0, where the counter value is 1, as an idle state. Stay in this state unless/until the *Start* signal is asserted. When *Start* is asserted, move to state 1 (on the next rising edge of the clock, of course).
- Once you have moved to state 1, now you'll react to the *Done* signal. If the Done signal is asserted, go back to state 0. If the *Done* signal is not asserted, go to state 2. ● Similar behavior for state 2: if *Done* isn't asserted, move to state 3, otherwise go back to state 0.
- If you reach state 3, state in state 3 until the *Done* signal is asserted.

Along with these changes to the counter itself, see if you can figure out how to modify the given test bench so that you can simulate and convince yourself that your design is working properly.

II. PRE-LAB

There is no pre-lab for this lab.

III. REPORT

For your lab report, include the completed source code for your working state machine and the screenshot of simulation results. In addition, include answers to the following questions.

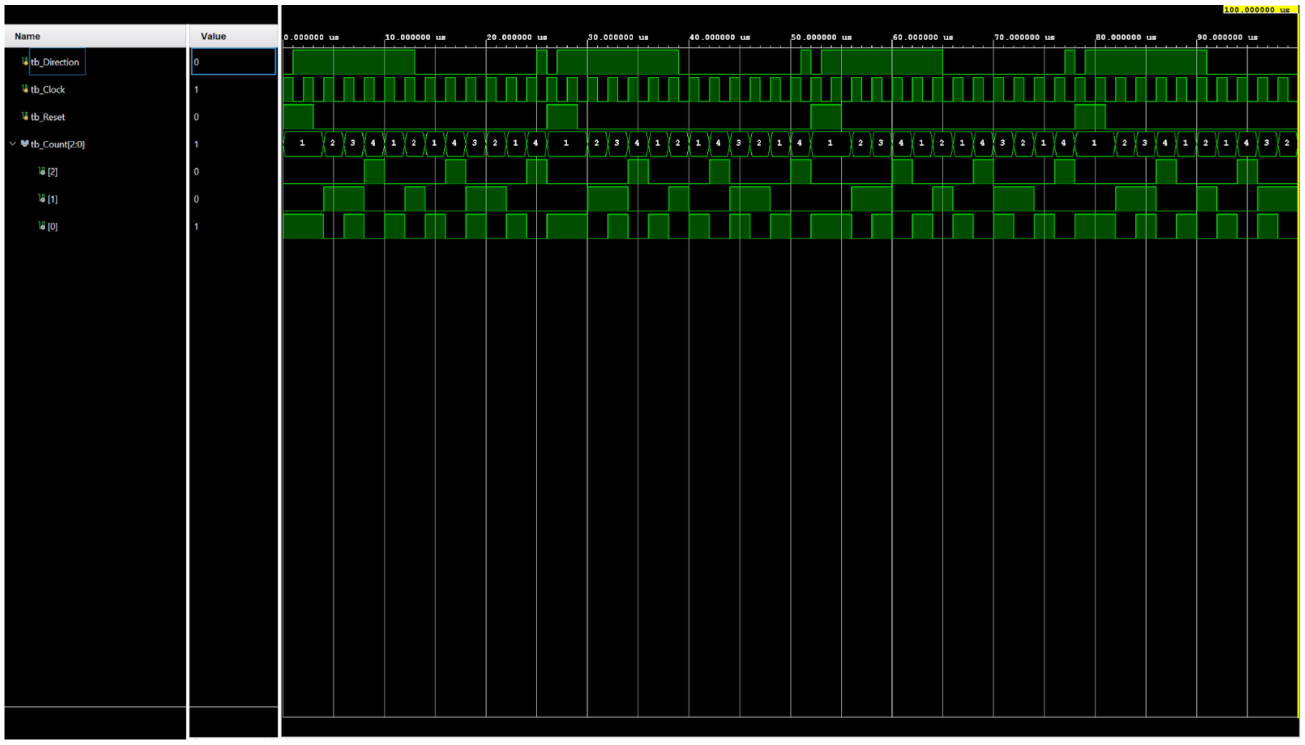
Q1: What problems did you encounter while implementing and testing your system? Did any problems arise when demonstrating for the TA? In other words, did the TA ask you to demonstrate something that you did not think of yourself? What was the scenario that you were asked to demonstrate? Provide some thoughts about why you didn’t think of this yourself.

Q2: In the counter design that we’ve done for this lab, we frequently observe the counter values wrapping around from the maximum value to minimum (or vice versa), e.g., 4 →1 or 1→4. If you want to design a Verilog module that can detect these, how would you approach this. In your answer, be specific whether you would describe the module in a Moore machine or a Mealy machine.

Code before any changes:

```
module ThreeBitCounter (Direction, Clock, Reset, Count);
   input Direction, Clock, Reset;
   output reg [2:0] Count;
   reg [1:0] currState, nextState;
   always @(posedge Clock)
      if (Reset) currState = 0;
      else currState = nextState;
   always @(*)
   case (currState)
      0: if (Direction==1) nextState = 1; else nextState = 3;
      1: if (Direction==1) nextState = 2; else nextState = 0;
      2: if (Direction==1) nextState = 3; else nextState = 1;
      3: if (Direction==1) nextState = 0; else nextState = 2;
   endcase
   always @(*)
   case (currState)
      0: Count = 3'b001;
      1: Count = 3'b010;
      2: Count = 3'b011;
      3: Count = 3'b100;
   endcase
endmodule
```

SS:



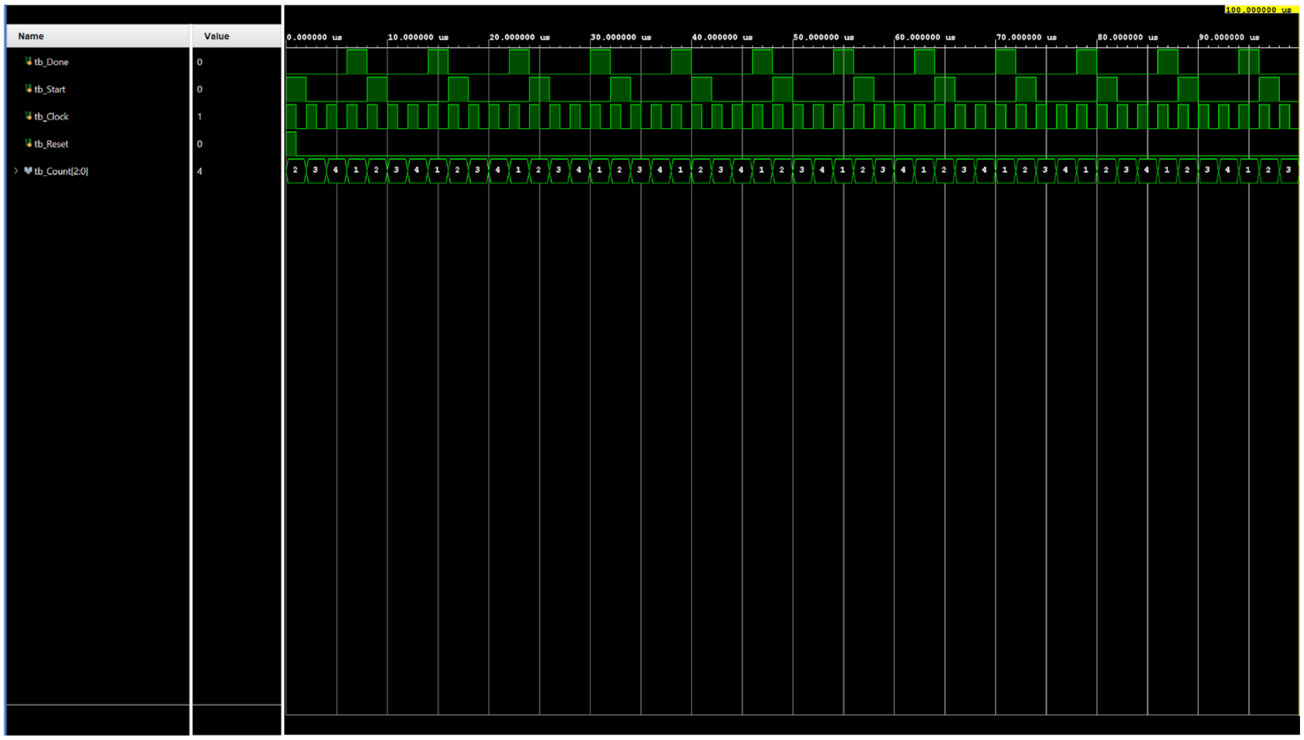
Name	Value
tb_Direction	0
tb_Clock	1
tb_Reset	0
tb_Count[2:0]	1
[2]	0
[1]	0
[0]	1

CODE AFTER CHANGE

```
module ThreeBitCounter (Done, Start, Clock, Reset, Count);
    input Start, Done, Clock, Reset;
    output reg [2:0] Count;
    reg [1:0] currState, nextState;
    always @(posedge Clock)
        if (Reset) currState = 0;
        else currState = nextState;
    always @(posedge Clock)
        if (Start) currState = 1;
        else currState = nextState;
    always @(*)
    case (currState)
        0: if (Done) nextState = 0; else nextState = 0;
        1: if (Done) nextState = 0; else nextState = 2;
        2: if (Done) nextState = 0; else nextState = 3;
        3: if (Done) nextState = 0; else nextState = 3;
    endcase
    always @(*)
    case (currState)
        0: Count = 3'b001;
        1: Count = 3'b010;
        2: Count = 3'b011;
        3: Count = 3'b100;
    endcase
```

```
endmodule


Timescale:
`timescale 100 ns / 1 ps
module tb_ThreeBitCounter ();

    reg tb_Done, tb_Start, tb_Clock, tb_Reset;
    wire [2:0] tb_Count;

    ThreeBitCounter UUT (.Done(tb_Done), .Start(tb_Start), .Clock(tb_Clock), .Reset(tb_Reset),
.Count(tb_Count));

    /* free running clock */
    always begin
        tb_Clock = 1; #10;
        tb_Clock = 0; #10;
    end

    /* done */
    always begin
        tb_Done = 0; #60;
        tb_Done = 1; #20;
    end

    always begin
        tb_Start = 1; #20;
        tb_Start = 0; #60;
    end

    /* reset */
    always begin
        tb_Reset = 1; #10;
        tb_Reset = 0; #1000;
    end
endmodule
```

SS AFTER CHANGE:



Q1: What problems did you encounter while implementing and testing your system? Did any problems arise when demonstrating for the TA? In other words, did the TA ask you to demonstrate something that you did not think of yourself? What was the scenario that you were asked to demonstrate? Provide some thoughts about why you didn't think of this yourself.

Q2: In the counter design that we've done for this lab, we frequently observe the counter values wrapping around from the maximum value to minimum (or vice versa), e.g., 4 →1 or 1→4. If you want to design a Verilog module that can detect these, how would you approach this. In your answer, be specific whether you would describe the module in a Moore machine or a Mealy machine.

Question

1. We encountered the problem of not being able to run the simulation properly because we didn't have the test bench in the file and did not have our actual code under the design sources. Also, we had to extend the simulation time because the default time was too short to see the clock changing at all.
2. When demonstrating our second part of the lab, our timing diagram simulation was messy. To fix that, Rikesh told us to change the timing of the changing of certain bit values.

Specifically, changing the timing of start and done so that they were caught in the rising edge and changing the timing of reset so it is only called at the start. We didn’t think of this ourselves as we were unsure if the changing of the timing is what the TA wanted.

3. We would describe the module in a Mealy machine because you can update the counter as you go from 4 to 1 or 1 to 4 and then you could then check the counter to determine how many times you have wrapped around.