STARS 2023: Experiment No. 8

Introduction to System Verilog

(Materials are adapted from ECE270 and ECE337 course materials)

Objective:

- To learn how to convert a given sequential circuit design into System Verilog.
- To learn how to use the System Verilog simulator to synthesize and simulate these designs.
- To learn how to verify the outputs of the sequential circuits on GTKWave.

Lab Tasks:

1. Design a counter that counts to 8'd99 and repeats
   (a) Modify the counter to add up/down counting mode
   (b) Modify the counter to count to an arbitrary number of N
   (c) Modify the counter to count only when an enable E is asserted
2. Implement a pattern detector detecting sequence "1101" using the Mealy and Moore state machine

Lab task1: Design a counter that counts to 8'd99 and repeats

Before you try to implement the specific counter asked in the task, go through the general counter design example given below. Remember, there are multiple ways you can design a counter and you are advised to create your own design based on your understanding.

8-bit binary up counter: The design of a basic binary UP counter is derived as follows:

Q2	Q1	Q0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

- Q0 changes state every clock cycle. The next state equation for Q0 is $Q0^* = \overline{Q0}$

- Q1 changes state every time Q0=1. The next state equation for Q1 is Q1*=Q1^Q0
- Q2 changes state every time Q0=1 AND Q1=1. The next state equation for Q2 is Q2*=Q2^(Q1.Q0)

- The next-state equation for an arbitrary stage "K" (Q_k*) of a binary counter is:
  $Q_K^* = Q_K \wedge (Q_{K-1} \bullet Q_{K-2} \bullet \ldots \bullet Q_1 \bullet Q_0)$

8-bit binary up counter Verilog code:

```
module count8u(input logic clk,

           output logic [7:0]Q

);

logic [7:0] next_Q; always_ff @

(posedge clk) begin

           Q<=next_Q;

end  always_comb  @  (Q)

begin

          next_Q[0]=~Q[0];

          next_Q[1]=Q[1]^Q[0];

          next_Q[2]=Q[2]^(Q[1]&Q[0]);

          next_Q[3]=Q[3]^(Q[2]&Q[1]&Q[0]);

          next_Q[4]=Q[4]^(Q[3]&Q[2]&Q[1]&Q[0]);

          next_Q[5]=Q[5]^(Q[4]&Q[3]&Q[2]&Q[1]&Q[0]);

          next_Q[6]=Q[6]^(Q[5]&Q[4]&Q[3]&Q[2]&Q[1]&Q[0]);

          next_Q[7]=Q[7]^(Q[6]&Q[5]&Q[4]&Q[3]&Q[2]&Q[1]&Q[0]);

end

endmodule
```

Note: You may use reduction operators in the next_Q statements.

8-bit binary down counter: The design of a basic binary DOWN counter is derived as follows:

Q2	Q1	Q0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

$\uparrow$

- Q0 changes state every clock cycle. The next state equation for Q0 is $Q0^* = \overline{Q0}$
- Q1 changes state every time Q0=0. The next state equation for Q1 is $Q1^*=Q1 \wedge (\overline{Q0})$
- Q2 changes state every time Q0=0 AND Q1=0. The next state equation for Q2 is $Q2^*=Q2 \wedge (\overline{Q1}.\overline{Q0})$

- The next-state equation for an arbitrary stage "K" (Q_k^*) of a binary counter is-
  $Q_K^* = Q_K \wedge (\bar{Q}_{K-1} \bullet \bar{Q}_{K-2} \bullet \ldots \bullet \bar{Q}_1 \bullet \bar{Q}_0)$

8-bit binary down counter Verilog code:

```
module count8d(input logic clk,

                        output logic [7:0] Q

);

logic [7:0] next_Q; always_ff @

(posedge clk) begin

      Q<=next_Q;

end  always_comb  @  (Q)

begin

     next_Q[0]=~Q[0];
```

```
        next_Q[1]=Q[1]^ ~Q[0];

        next_Q[2]=Q[2]^( ~Q[1]& ~Q[0]);

        next_Q[3]=Q[3]^( ~Q[2]& ~Q[1]& ~Q[0]);

        next_Q[4]=Q[4]^( ~Q[3]& ~Q[2]& ~Q[1]& ~Q[0]);

        next_Q[5]=Q[5]^( ~Q[4]& ~Q[3]& ~Q[2]& ~Q[1]& ~Q[0]);

        next_Q[6]=Q[6]^( ~Q[5]& ~Q[4]& ~Q[3]& ~Q[2]& ~Q[1]& ~Q[0]);

        next_Q[7]=Q[7]^( ~Q[6]& ~Q[5]& ~Q[4]& ~Q[3]& ~Q[2]& ~Q[1]& ~Q[0]);

end

endmodule
```

With the help of the System Verilog code provided for the 8-bit binary up-counter, design an 8-bit counter that counts to 99, and then starts over from 0. To make it count to 99 and restart, you will need to modify the always_comb block to set the value of next_Q to 0 when Q is equal to 8'd99, otherwise, continue to increment Q by 1 by using the equations for next_Q. You might want to use reduction operators. For instance, instead of typing out something like "x[3] & x[2] & x[1] & x[0]" you can say, instead, "&x[3:0]". The AND prefix is applied to all elements of the bus. This works for all the Boolean binary operators like '|' and '^'. The use of reduction operators makes it possible to succinctly specify the next-state equations for the counter. The reset should be an asynchronous reset. It should not wait for the next rising edge of the clock to take effect.

Required module name: count8du
Required port name:

Port name	Direction	Description
clk	input	The system clock.
rst	input	This is an asynchronous, active-high system reset. When this line is asserted (logic ‘1’), all registers/flip-flops in the device must reset to their initial value.
Q[7:0]	output	8-bit logic output

In the top module, instantiate this module, and connect clk to hz100, rst to reset, and Q to right.

```
count8du u1(.clk(hz100), .rst(reset), .Q(right));
```

To ensure that your counter is working correctly, you should see that the right[7] LED is never illuminated. You will also notice that the right[6] LED blinks at a rate of once per second. This is because it counts from 0 ... 99 (100 steps) repeatedly. Each step of the count takes 1/100 of a second, so every full count from 0 to 99 (100 steps) should take exactly one full second. The right[6] LED is on when the decimal value of the count is between 64 and 99, or approximately 1/3 of the time of the full count. The reset should be an asynchronous reset. It should not wait for the next rising edge of the clock to take effect.

To verify the output on GTKWave, follow the steps-

1. Set the build target to verify and right-click. You have been provided a test bench to verify your circuit. If there is no error, you will see the message on the terminal- no error press any key to close it.
2. Press any key and on the command window that appears, write the command- make verify. A new GTKWave window will open.
3. Under the SST tab, click on your instance name, select all the wires and signals you want to see on the waveform and right-click—insert.
4. At rst=1, q and next_q=0. When rst=0, the count starts. It counts up to 99 and then returns to zero. Under the signals tab, you may select the signal, right-click, and change the data format to decimal/hexadecimal/binary or any format you want.

At this point, you should have a verified counter that is able to count to 8'd99 as indicated by the output, and upon reaching 8'd99, will restart at zero on the next clock toggle.

TA check-off point: Explain how you verified the output on the FPGA to your TA and show the GTKWave output to your TA for check-off.

(a) Modify the counter to add up/down counting mode

For this step, you will modify the count8du module to be able to make it count up or down based on a new input. To start, add a new 1-bit input called DIR to your module's port header (alongside input logic CLK, RST, and output logic [7:0] Q). In the module instantiation in your top module, add the DIR port and connect it to pb[17].

The counter must be a module named count8du, and the only additional line in your top module must be the module instantiation of count8du, i.e. all counter-based logic must be in your count8du module and not your top module.

Copy the count8du module below your top module in your top.sv file/simulator tab. In the always_comb block of your module, modify your logic so that:

- If DIR is equal to zero, the value of next_Q is updated to Q - 1. Do not actually use Q 1. Change the equations for the individual bits to do this. If the value of Q is zero, next_Q should be equal to 8'd99.
- Else, if DIR is equal to one, the value of next_Q is updated to Q + 1. Do not actually use Q + 1. Change the equations for the individual bits to do this. If the value of Q is 8'd99, next_Q should be equal to 0.

Similar to how you ensured that your design counted to 8'd99 and then started again at 0 in the previous design, ensure that your counter goes all the way from 8'd99 to 0 when counting down, and from 0 to 8'd99 when counting up, and restarts accordingly.

TA check-off point: Show up and down counting on your FPGA to your TA for check-off.

(b) Modify the counter to count to an arbitrary number of N

For this step, you will modify the count8du module from the previous step to make it count to any number N (inclusive), rather than just 8'd99. To start, add a new 8-bit input called MAX to your module's port header, and in the module instantiation in your top module, add the MAX port and connect it to a Verilog literal 8'd49.

Since you already have logic in place to set next_Q to a specific value (8'd99) from earlier, all you have to do is replace those occurrences of 8'd99 with MAX. As a result, your counter should now reset to zero twice as fast as the previous version.

(c) Modify the counter to count only when an enable E is asserted

For this step, you will modify the count8du module from the previous step to make it count only when an enable input E is asserted. To start, add a new 1-bit input called E to your module's port header, and in the module instantiation in your top module, add the E port and connect it to pb[18] (the Y pushbutton). As a result, your counter should not count as long as pb[18]/Y is not held. It will only start counting when you hold down the button.

TA check-off point: Show the counter outputs to your TA for check-off.

Lab task2: Implement a pattern detector detecting sequence “1101” using the Mealy and Moore state machine

The purpose of an “1101” detector is to assert an output whenever the sequence “1101” is detected in a serial input data stream. Below are the requirements for this circuit-

- The input is serial.
- When the sequence “1101” is detected, ‘1’ should be asserted.
- Must detect overlapping occurrences of the desired 1101 pattern (i.e., an input sequence of “001101**1**101**1**100” should detect the 1101 pattern twice).

Find the state transition diagram for a Serial “1101” sequence detector as a Mealy and Moore state machine in the figure below. The output of a Mealy machine is the function of current state and current inputs whereas Moore machine output is a function of current state only.

To construct the state transition diagram, find the number of states required to detect the 4 bits for each machine. For the Mealy machine, considering S0 to be the idle state, on receiving input ‘1’, it moves to state s1. On receiving input ‘0’, it remains in the idle state. When we write ‘0/0’ in the state transition diagram, it is in the form of ‘input/output’. At S3, bits ‘110’ are already detected. On receiving input ‘1’, all the required bits of the pattern are detected so it will result in output ‘1’.

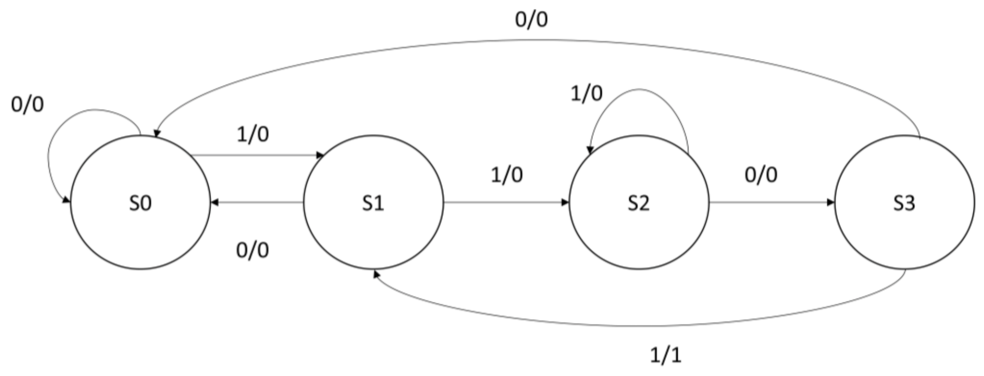

Figure 1: Mealy state transition diagram for 1101 pattern detector

State	Meaning
S0	“XXXX” or IDLE state
S1	“1XXX” or 1 detected
S2	“11XX” or 11 detected
S3	“110X” or 110 detected

Required module name: mealy1101
Required port name:

Port name	Direction	Description
clk	input	The system clock.
n_rst	input	This is an asynchronous, active-low system reset. When this line is asserted (logic ‘0’), all registers/flip-flops in the device must reset to their initial value.
i	input	The serial input for the input stream to process.
o	output	The current result from processing the serial stream so far.

The use of Moore models to design sequencer recognizers is generally preferred because you typically don’t want any output signals to change (based on input signal changes) until the machine is clocked to the next state (i.e., the outputs should only be a function of the state variables). In the Moore pattern detector, there will be one more state S4. From the Moore state transition diagram, you can see the output depends only on the state, not on the present inputs.

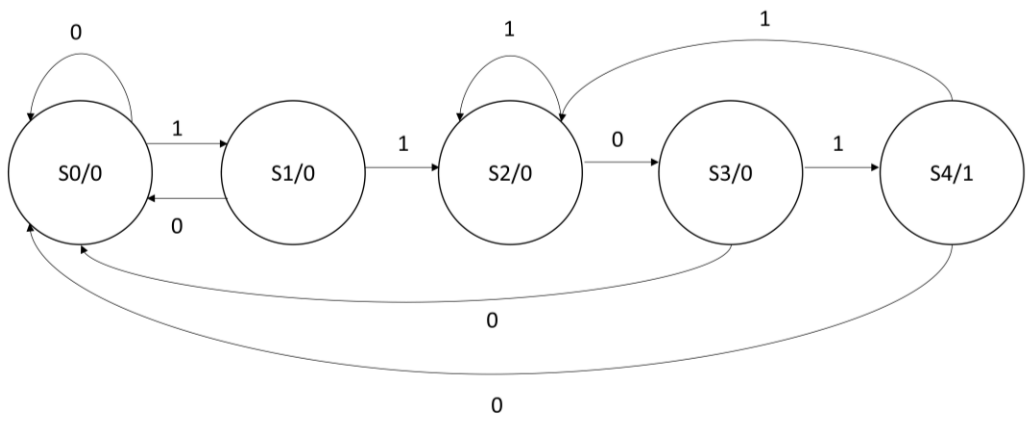

Figure 2: Moore state transition diagram for 1101 pattern detector

State	Meaning	Output
S0	"XXXX" or IDLE state	'0'
S1	"1XXX" or 1 detected	'0'
S2	"11XX" or 11 detected	'0'
S3	"110X" or 110 detected	'0'
S4	"1101" or 1101 detected	'1'

Required module name: moore1101
Required port name:

Port name	Direction	Description
clk	input	The system clock.
n_rst	input	This is an asynchronous, active-low system reset. When this line is asserted (logic '0'), all registers/flip-flops in the device must reset to their initial value.
i	input	The serial input for the input stream to process.
o	output	The current result from processing the serial stream so far.

TA check-off point:

(a) Show the mealy1101 and moore1101 module outputs to your TA. Add your own test case to the testbench and show the output for the input 11011011010.

(b) Create state transition diagram for a "10101" Moore pattern detector and have a TA check off your work.

Example of a moore1101 pattern detector:

```
module moore1101(clk, n_rst, i, o);
input logic clk;
input logic i, n_rst;
output logic o;
typedef enum logic [2:0] {s0,s1,s2,s3,s4} state_t;
logic [2:0] state, next_state;
always_comb begin
next__state=state;
        case(state)
        s0: next_state=i?s1:s0;
         //write for s1, s2, s3, s4 and default
         ………..
         ………..
         ………..
always_ff @ (posedge_clk, negedge n_rst) begin
         //conditional statements for the states
         ………
         ………
         ………
end
always_comb begin
         //conditional statements for output
         ……...

        ...........
        ...........
end
endmodule
```