Lab EE 371: Parking Lot Occupancy Counter

Autumn 2018

Lab 1

Shawnna Cabanday
Professor Rania Hussein
Teaching Assistant: Shuowei Li
Lab Section: Wednesday from 12:30 - 2:30 PM
Date submitted: Friday, October 5, 2018

Abstract

The purpose of this lab was to review digital design concepts from B EE 271, implement them with System Verilog VHDL programming in Quartus, and use ModelSim to monitor and revise circuit behavior based on simulation waveforms. The primary task of this lab was to design and simulate a parking lot occupancy counter and be able to demonstrate the program on a DE1_SoC FPGA board.

Introduction

Digital logic circuits handle data encoded in binary form through signals that have two values, 0 and 1. This binary logic that deals with determining whether or not input conditions evaluate to an output of either “true” and “false” allows scientists and engineers to develop complex digital logic circuits and computers that can be built using a few types of basic circuits called gates, each performing a specific logic operation. Thus, the purpose of this lab is to practice simulating circuits that require combinational logic for a functional state machine (FSM) in order to simulate a logic system. For this, a parking lot scenario is considered with a single entry and exit gate. Using photo sensors to monitor the activity of the cars, there are two sequences that will indicate whether or not a car enters or exits a lot.

Materials

- ModelSim and Quartus 17.0 Software
- 1 green LED and 1 red LED
- Power cable
- DE1_SoC FPGA board
- Jumper Cables for GPIO_0 Output to LEDs

Procedures

Part 1: Designing a Functional State Machine (FSM)

The task of this lab procedure was to design a functional state machine for the photos sensor system, develop SystemVerilog code for the FSM, and then simulate it in ModelSim to verify existing waveforms. The FSM has two input signals, A and B, and two output signals, enter and exit. The following sequence, as described by the lab 1 guideline, indicates that the car enters a lot:

- Initially, both sensors are unblocked (A and B are “00”)
- Sensor A is blocked (A and B signals are “10”)
- Both sensor are blocked (A and B signals are “11”)
- Sensor A is unblocked (“01”)
- Both sensors unblocked (“00”)

To begin the design, a state diagram was created (Figure 1.1). Next, a state table was created based on the state diagram (Table 1.1). To implement the next state logic in SystemVerilog, the module *parkinglotfsm* was created with inputs clk, reset, A, B, enter, and exit. State names were enumerated using the following stages: *unblocked, sensorB, sensorA, blocked*, with each state corresponding to 00, 01, 10, 11, respectively. Within the module *parkinglotfsm*, next state logic was created using case statements in combination with if-else statements, all coded within an “always @(posedge clk)” block. The outputs for

the enter and exit sensors were assigned based on present state and next state variables, which were embedded into the combinational logic rather than through an assign statement occuring after the combinational logic code. I included both assign statements after the combinational logic portion of the always block for testing purposes, but ultimately preferred the look of the enter and exit assignments within the case statements since it was analogous to the table created. A flip-flop is then implemented to update the present state to the reassigned next state. The resulting waveforms were generated and evaluated based on the state diagram tables.

After reviewing this code, I would have revised the "always @(posedge clk)" block to an always_comb to sustain the purpose of the module, that is, to simulate combinational logic.

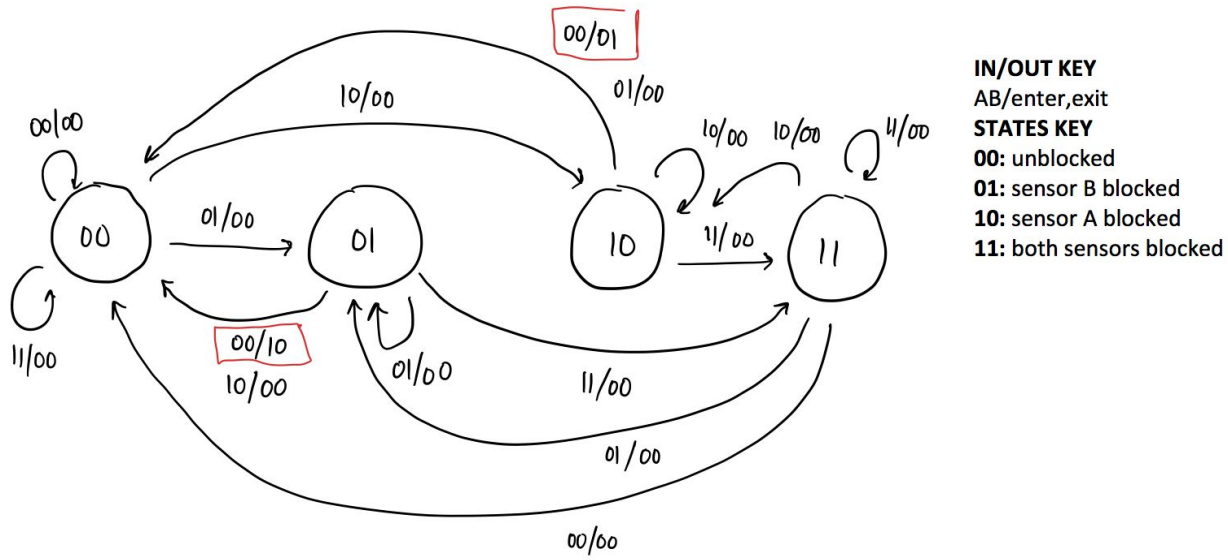
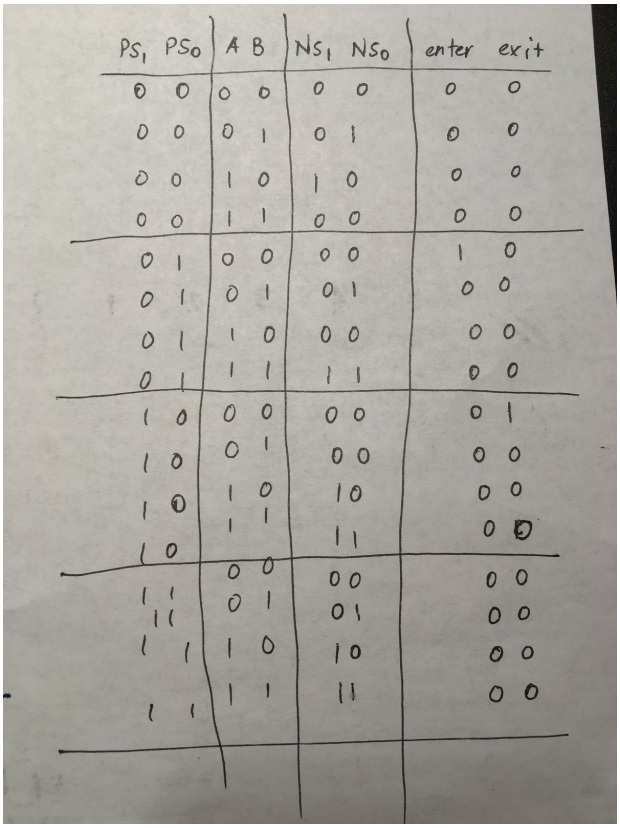


Figure 1.1: State Diagram for FSM



PS_1	PS_0	A	B	NS_1	NS_0	enter	exit
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	0	0	0	0
0	1	1	1	1	1	0	0
1	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	1	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	1	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	1	0	0

Table 1.1: State Table for FSM

Part 2: Designing a Counter

The task of this lab procedure was to design a counter with two control signals, *inc* and *dec*, which increment and decrement depending on the output signals from the previous FSM: enter and exit. For simplicity purposes, neither a state table or state diagram was created for the counter. The counter was built with SystemVerilog code similar to the FSM machine designed in part 1, with the numbers incrementing and decrementing based on if-else statements within a 25 part case statement clause. To implement the next state logic in SystemVerilog, the module *counter* was created with inputs *clk, reset, inc, dec, cout*. The output of the variable cout holds the present number of cars parked in the parking lot in 5-bit binary. It will be used primarily used to connect to the hex displays, which is completed in a separate module titled *hexdisplay*. The *counter* module approaches sequential logic in an always @(posedge clk) block with a case statement that determines the next state assignment based on the present state of *cout* based on whether or not an increment or decrement signal was transmitted from the FSM module *parkinglotfsm*. The *counter* module has a case statement for each numerical value from 0 to 25. After the case statement, I implement an always flip-flop with an always @(posedge clk). On further revision, I would have used the always flip-flop syntax for consistency.

Part 3: Connecting the Entire System

The task of this lab procedure was to finalize the designed system and connect the modules together in main driver module.

Before completing this, a separate module titled *hexdisplay* was used to capture the output from the *counter* module in order to determine the corresponding hex displays for each of the 6, 7-segment displays. The *hexdisplay* module was created similar to the previous combinational logic modules, which utilized case statements within an always block. With the case statement method, each of the segments in the individual hex displays were manually set. For the ease of reading the code, the 7-bit outputs that correspond with the individual lighting of the LED segments were parameterized with their characteristic output when shown on the FPGA (i.e. zero = 7'b1000000). The display of each hex will be set according to the *cout* value outputted by the *counter* module.

Furthermore, an additional module titled *userInput* was created to minimize metastability issues when the user pressed the switches to simulate the photo sensor sequences. Simply put, *userInput* is a flip-flop module that holds the inputted signal an extra clock length before outputting it its next state.

To connect all the 'helper methods' together, another module was created to act as the primary driver module for all smaller FSM modules. Each of the helper modules are instantiated and input keys and output hex displays are placed within the instantiated statements. Wires are used to transport certain outputs from one module to another, such as the 5-bit *cout* output to the *hexdisplay* module.

Results

Completing the first procedure was one of the simpler tasks of the system, provided that I have had experience completing a lab extremely similar to this one beforehand in EE 271. Therefore, no unexpected issues arose from completing the first task. Figure 3.1 illustrates the resulting waveforms from the FSM and the data matched prior expectations.

The second procedure on the other hand was slightly more difficult for me, simply because I was over-complicating the algorithm of my counter. I was attempting to create a counter that used adder and subtractor modules; however, I kept running into the issue of the counter not incrementing whatsoever. With limited time in mind, I decided to take the simpler route of creating a counter system that was based off entirely of logical steps, that is, through the use of if-else statements. For instance, if we want to increment, what number is expected to come after? Or alternatively, if we want to decrement, what number comes before the current present number we are looking at? Thus, I completed the counter module as described in the procedures. The results of the counter system matched entirely with expectations and behaved accordingly. Figure 3.2 illustrates the resulting waveforms from the counter module.

While this method is a logical and simple way of approaching the *counter* system, I also believe it is extremely inefficient; however, due to time constraints, I decided to stick with the simplest approach. If given more time to revise the code, I would have followed the suggestion provided in the discussions board on canvas by using an adder/subtractor module to produce the *cout* output. This is a much more efficient alternative for larger sized counters with a maximum greater than 25. Figure 2.1 illustrates the waveforms generated by the designed *counter* module through ModelSim.

The final steps were simply then to create a final driver module that would use all the smaller modules to create the entire system. This was not difficult, again, since I had practiced this previously and extensively in B EE 271 and B EE 425. Figure 3.3 illustrates the resulting waveforms from the driver module. This perfectly illustrates possible sequences and inputs by simulation, and it matched according to prior expectations.

Figure 3.1: Generated Waveforms for FSM

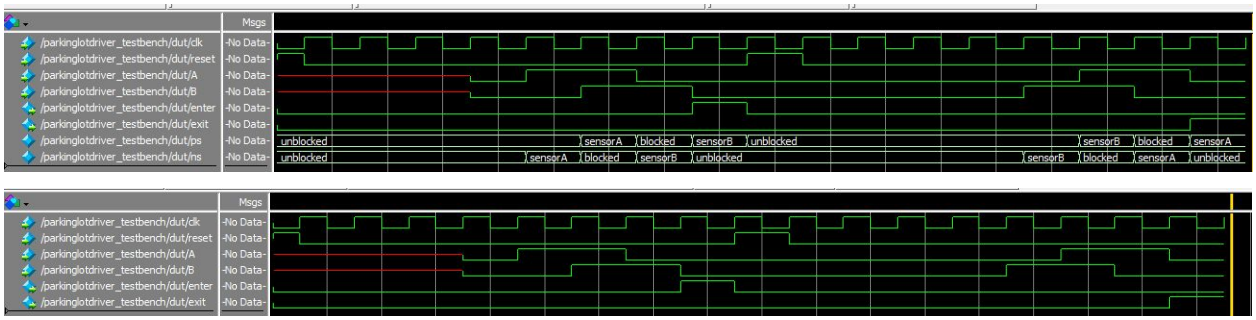


Figure 3.2: Generated Waveforms for Counter System

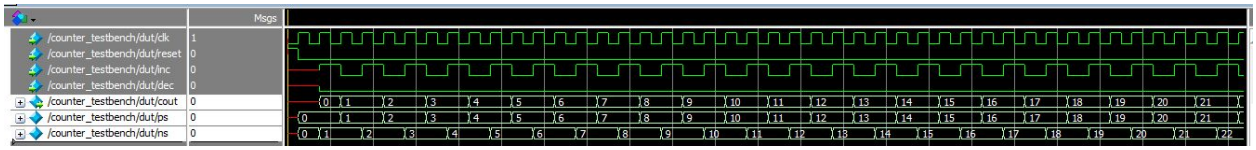
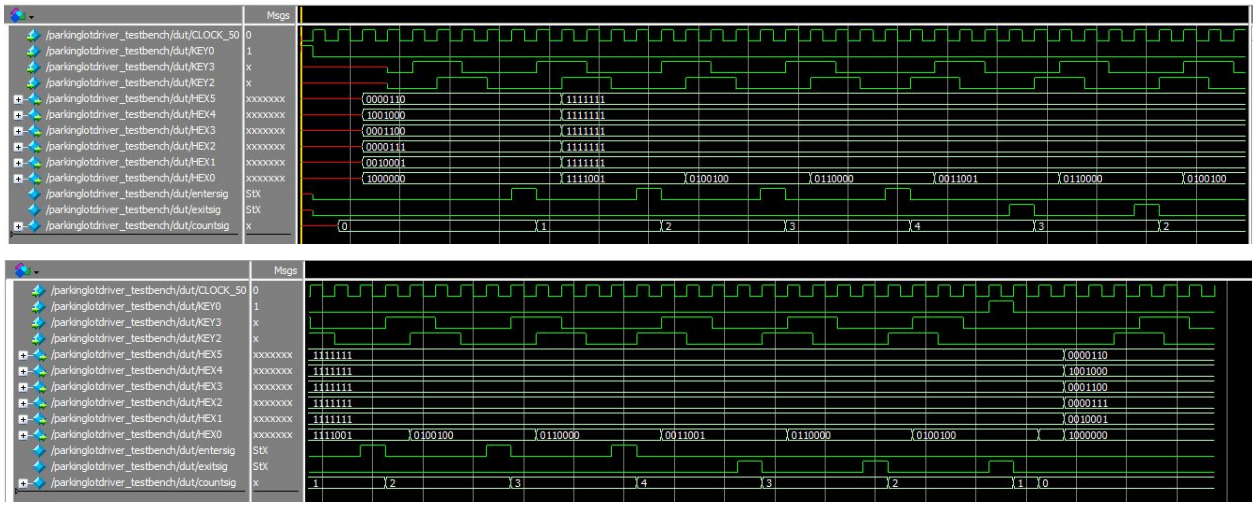


Figure 3.3: Generated Waveforms for Final Driver (Entire System)



Flow Summary:

Flow Summary

<<Filter>>

Flow Status	Successful - Fri Oct 05 22:40:21 2018
Quartus Prime Version	17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name	parkinglotdriver
Top-level Entity Name	parkinglotdriver
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	64 / 32,070 (< 1 %)
Total registers	90
Total pins	48 / 457 (11 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Analysis and Synthesis Resources:

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers
1	|parkinglotdriver	122 (0)	88 (0)
1	|clock_divider:cdiv|	21 (21)	21 (21)
2	|counter:mycounter|	64 (64)	57 (57)
3	|hexdisplay:mydisplay|	29 (29)	0 (0)
4	|parkinglotfsm:myfsm|	6 (6)	6 (6)
5	|userInput:inA|	1 (1)	2 (2)
6	|userInput:inB|	1 (1)	2 (2)

Conclusion

This lab helped to re-polish the most important topics from EE 271. Ultimately, the use of computer aided simulations in Quartus and Verilog programming allows for increased efficiency in the steps and processes needed to develop and verify digital logic circuits on the FPGA. Debugging and determining the major sources for issues is a long process for all simulations and Verilog programming. Creating a system such as the one created in this lab demonstrates the most fundamental steps needed to design, simulate, and implement a digital logic system.

Total Estimated Hours: 13 hours

Code Screenshots:

Parkinglotdriver Module

```
module parkinglotdriver(CLOCK_50, KEY0,
                        KEY3, KEY2,
                        HEX5, HEX4, HEX3, HEX2, HEX1, HEX0,
                        GPIO0, GPIO1);

    input logic CLOCK_50, KEY0, KEY3, KEY2;
    output logic GPIO0, GPIO1;
    output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
    wire entersig, exitsig;
    wire [4:0] countsig;
    wire inputA, inputB;

    logic [31:0] clk;
    parameter whichClock = 20;

    clock_divider cdiv (.clock(CLOCK_50), .divided_clocks(clk));

    userInput inputA (.clk(clk[whichClock]), .D(~KEY3), .Q(inputA));
    userInput inputA (.clk(clk[whichClock]), .D(~KEY2), .Q(inputB));

    parkinglotfsm myfsm (.clk(clk[whichClock]), .reset(~KEY0), .A(inputA), .B(inputB),
                                        .enter(entersig), .exit(exitsig));

    counter mycounter (.clk(clk[whichClock]), .reset(~KEY0), .inc(entersig), .dec(exitsig),
                               .cout(countsig[4:0]));

    hexdisplay mydisplay (.clk(clk[whichClock]), .inputcount(countsig[4:0]),
                          .status5(HEX5), .status4(HEX4), .status3(HEX3), .status2(HEX2),
                          .led1(HEX1), .led0(HEX0));

    assign GPIO0 = ~KEY3;
    assign GPIO1 = ~KEY2;

```

Parking Lot FSM Module

```
84  module parkinglotfsm(clk, reset, A, B, enter, exit);
85
86     input logic clk, reset, A, B;
87     output logic enter, exit;
88
89     //State Variables
90     enum {unblocked, sensorB, sensorA, blocked} ps, ns;
91
92     always @(posedge clk) begin
93        case(ps)
94
95           unblocked:  if({A,B} == 2'b01)   begin    ns = sensorB; enter = 0; exit = 0; end
96                       else if({A,B} == 2'b10) begin    ns = sensorA; enter = 0; exit = 0; end
97                       else              begin       ns = unblocked; enter = 0; exit = 0; end
98
99           sensorB: if({A,B} == 2'b01) begin   ns = sensorB; enter = 0; exit = 0; end
100                    else if({A,B} == 2'b11) begin ns = blocked; enter = 0; exit = 0; end
101                    else if({A,B} == 2'b00) begin ns = unblocked; enter = 1; exit = 0; end
102                    else        begin          ns = unblocked; enter = 0; exit = 0; end
103
104          sensorA: if({A,B} == 2'b10) begin      ns = sensorA; enter = 0; exit = 0; end
105                   else if({A,B} == 2'b11) begin ns = blocked; enter = 0; exit = 0; end
106                   else if({A,B} == 2'b00) begin ns = unblocked; exit = 1; enter = 0; end
107                   else        begin          ns = unblocked; enter = 0; exit = 0; end
108
109          blocked: if({A,B} == 2'b01) begin      ns = sensorB; enter = 0; exit = 0; end
110                   else if({A,B} == 2'b10) begin ns = sensorA; enter = 0; exit = 0; end
111                   else if({A,B} == 2'b11) begin ns = blocked; enter = 0; exit = 0; end
112                   else           begin       ns = unblocked; enter = 0; exit = 0; end
113
114       endcase
115    end
```

Hexdisplay Module

```
260 module hexdisplay(clk, inputcount,
261                   status5, status4, status3, status2,
262                   led1, led0);
263    input clk;
264    input logic [4:0] inputcount;
265    output logic [6:0] status5, status4, status3, status2;
266    output logic  [6:0] led1, led0;
267
268    //                          6543210
269    parameter [6:0]   zero =   7'b1000000,
270                      one =    7'b1111001,
271                      two =    7'b0100100,
272                      three =  7'b0110000,
273                      four =   7'b0011001,
274                      five =   7'b0010010,
275                      six =    7'b0000010,
276                      seven =  7'b1111000,
277                      eight =  7'b0000000,
278                      nine =   7'b0011000,
279                      F =      7'b0001110,
280                      U =      7'b1000001,
281                      L =      7'b1000111,
282                      E =      7'b0000110, //ENPTY
283                      N =      7'b1001000,
284                      P =      7'b0001100,
285                      T =      7'b0000111,
286                      Y =      7'b0010001,
287                      blk =    7'b1111111;
288
289
290    always @(inputcount)
291       case(inputcount)
292          0: begin status5 = E; status4 = N; status3 = P; status2 = T; led1 = Y; led0 = zero; end
293          1: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk; led1 = blk; led0 = one; end
294          2: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk; led1 = blk; led0 = two; end
295          3: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk; led1 = blk; led0 = three; end
296          4: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk; led1 = blk;led0 = four; end
297          5: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk; led1 = blk;led0 = five; end
298          6: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk; led1 = blk;led0 = six; end
299          7: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk; led1 = blk;led0 = seven; end
300          8: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk; led1 = blk;led0 = eight; end
301          9: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk; led1 = blk;led0 = nine; end
302          10: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk;led1 = one; led0 = zero; end
303          11: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk;led1 = one; led0 = one; end
304          12: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk;led1 = one; led0 = two; end
305          13: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk;led1 = one; led0 = three; end
306          14: begin status5 = blk; status4 = blk; status3 = blk; status2 = blk;led1 = one; led0 = four; end
```

Counter Module

```
166     always @(posedge clk) begin
167        case(ps)
168           zero: if(inc) ns = one;
169                 else ns = zero;
170           one: if(inc)   ns = two;
171                 else if(dec)      ns = zero;
172                 else ns = one;
173           two: if(inc)   ns = three;
174                 else if(dec)      ns = one;
175                 else ns = two;
176           three:if(inc)  ns = four;
177                 else if(dec)      ns = two;
178                 else ns = three;
179           four: if(inc)  ns = five;
180                 else if(dec)      ns = three;
181                 else ns = four;
182           five: if(inc)  ns = six;
183                 else if(dec)      ns = four;
184                 else ns = five;
185           six: if(inc)   ns = seven;
186                 else if(dec)      ns = five;
187                 else ns = six;
188           seven: if(inc)    ns = eight;
189                 else if(dec)      ns = six;
190                 else ns = seven;
191           eight:if(inc)     ns = nine;
192                 else if(dec)      ns = seven;
193                 else ns = eight;
194           nine: if(inc)  ns = ten;
195                 else if(dec)      ns = eight;
196                 else ns = nine;
197           ten:  if(inc)  ns = eleven;
198                 else if(dec)      ns = nine;
199                 else ns = ten;
200           eleven:if(inc)    ns = twelve;
201                 else if(dec)      ns = ten;
202                 else ns = eleven;
203           twelve:if(inc)    ns = thirteen;
204                 else if(dec)      ns = eleven;
205                 else ns = twelve;
206           thirteen:if(inc)  ns = fourteen;
207                 else if(dec)      ns = twelve;
208                 else ns = thirteen;
209           fourteen: if(inc)    ns = fifteen;
210                 else if(dec)      ns = thirteen;
211                 else ns = fourteen;
```

Clock Divider and Metastability

```
 84  module parkinglotfsm(clk, reset, A, B, enter, exit);
 85
 86     input logic clk, reset, A, B;
 87     output logic enter, exit;
 88
 89     //State Variables
 90     enum {unblocked, sensorB, sensorA, blocked} ps, ns;
 91
 92     always @(posedge clk) begin
 93        case(ps)
 94
 95           unblocked:  if({A,B} == 2'b01)   begin    ns = sensorB; enter = 0; exit = 0; end
 96                       else if({A,B} == 2'b10) begin    ns = sensorA; enter = 0; exit = 0; end
 97                       else             begin       ns = unblocked; enter = 0; exit = 0; end
 98
 99           sensorB: if({A,B} == 2'b01) begin   ns = sensorB; enter = 0; exit = 0; end
100                    else if({A,B} == 2'b11) begin ns = blocked; enter = 0; exit = 0; end
101                    else if({A,B} == 2'b00) begin ns = unblocked; enter = 1; exit = 0; end
102                    else        begin          ns = unblocked; enter = 0; exit = 0; end
103
104          sensorA: if({A,B} == 2'b10) begin      ns = sensorA; enter = 0; exit = 0; end
105                   else if({A,B} == 2'b11) begin ns = blocked; enter = 0; exit = 0; end
106                   else if({A,B} == 2'b00) begin ns = unblocked; exit = 1; enter = 0; end
107                   else        begin          ns = unblocked; enter = 0; exit = 0; end
108
109          blocked: if({A,B} == 2'b01) begin      ns = sensorB; enter = 0; exit = 0; end
110                   else if({A,B} == 2'b10) begin ns = sensorA; enter = 0; exit = 0; end
111                   else if({A,B} == 2'b11) begin ns = blocked; enter = 0; exit = 0; end
112                   else           begin       ns = unblocked; enter = 0; exit = 0; end
113
114       endcase
115    end
116
117       //alternative method for FSM rather than embedding states
118 // assign enter = (ps == sensorB && {A,B} == 2'b00);
119 // assign exit = (ps == sensorA && {A,B} == 2'b00);
120
121    always_ff @(posedge clk) begin
122
123       if(reset)
124          ps <= unblocked;
125       else
126          ps <= ns;
127    end
128
129 endmodule
```