Garrett Tashiro

EE 371

October 12, 2021

Lab 1 Report

Procedure

This lab was comprised of four tasks. The first task was to design an FSM for the parking lot that would tell if a car was entering, or exiting the lot, and from the design write the code for the FSM in System Verilog. The second task was to design a parameterized counter module to keep track the cars entering, or exiting the parking lot. The third task was to create a module that will display the number of cars in the parking lot on the HEX displays of the DE1_SoC board. The fourth task was to combine all of these modules in the DE1_SoC module. The parkFSM, counter, and carHexDisplay modules worked together to show the number of cars entering, or exiting the parking lot, with a max number of cars being set in the parameterized counter module. The DE1_SoC module also had GPIO to control LED's on the virtual breadboard to act as the sensors for the parking lot.

Task #1

The first task was to design an FSM for the parking lot. I read over the section for the first task a few times, looked at the picture, and then drew my own picture to help me visualize how cars would enter, or exit the parking lot. I wrote down the names/labels for each state, along with the outputs for them. Before I started drawing the state machine, I used my drawing to lay out the labels of states with what outputs they would have in what direction. Since we were given two sensors to work with, sensors a and b, I first drew a state diagram with only four states: none, A, B, and Both. I chose not to go with my first idea so I didn't have to have an internal counter inside the FSM to keep track of what the previous states were. I ended up creating an FSM with seven states: none, enterA, enterAB, enterB, exitB, exitAB, exitA. In order for a car to enter, or exit the parking lot it would start in none, trigger the first sensor, then trigger both sensors, then trigger just the second sensor, followed by triggering no sensors and ending up back in the none state. This is shown below in Figure 1.

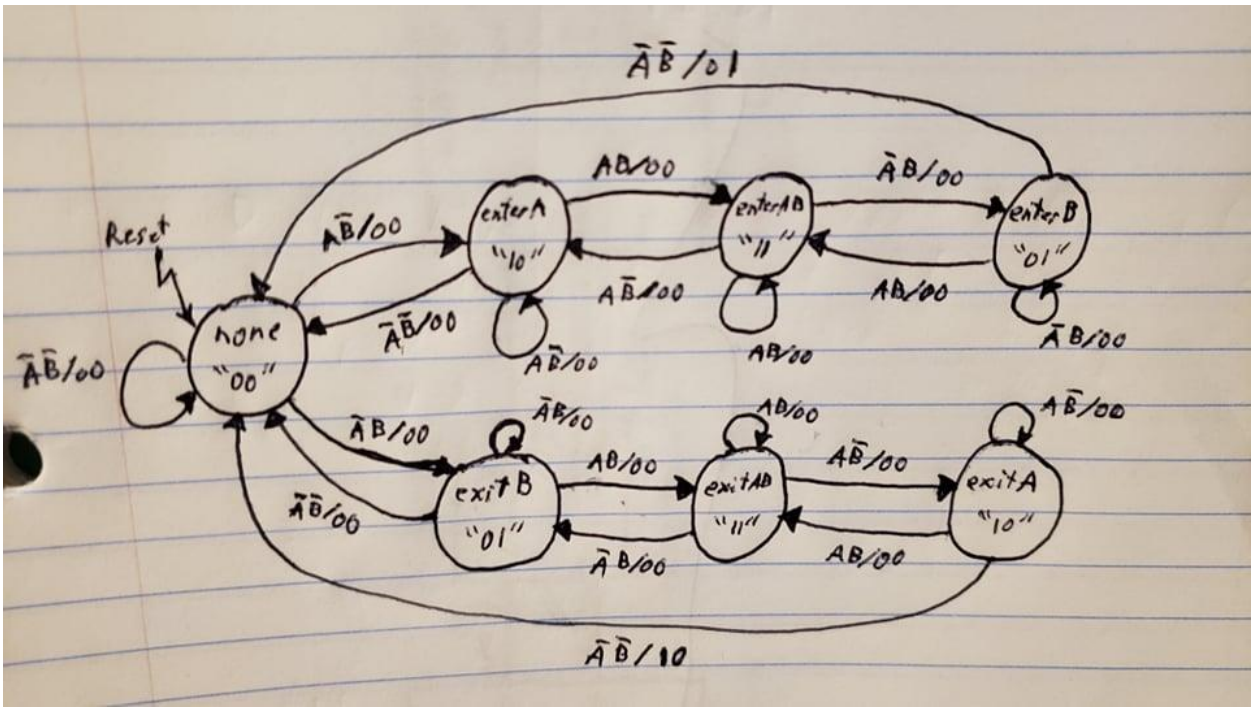

Fig. 1: Parking Lot FSM Design

This FSM design was to have a specific sequence for entering and exiting the parking lot. The design also had to account for the possibility of a pedestrian walking through and trigger the sensors. Having the design be four specific states sequentially allowed accounted for if a pedestrian was to walk through the sensors to enter, or exit the lot, without actually increasing the number of cars in the lot. For a car to enter the lot the sequence of inputs is: $\bar{a}\bar{b} \rightarrow a\bar{b} \rightarrow ab \rightarrow \bar{a}b \rightarrow \bar{a}\bar{b}$. Once a vehicle has gone through the entering sequence the 1-bit output enter is set to 1 to signal that a car has entered the lot. For a car to exit the lot the sequence of inputs is: $\bar{a}\bar{b} \rightarrow \bar{a}b \rightarrow ab \rightarrow a\bar{b} \rightarrow \bar{a}\bar{b}$. Once a vehicle has gone through the exiting sequence the 1-bit output exit is set to 1 to signal that a car has exited the lot. The reason for choosing to have a design like this was to prevent a false positive of entry if a pedestrian was walking in, or out of the lot, or if a vehicle decided to back up before fully entering or exiting the lot. This FSM design will only count cars that have fully entered or exited the lot.

The System Verilog code for this design can be found in Appendix 1.A) parkFSM.sv. For this module I first started with the inputs and outputs that were needed by referring to the drawing of the FSM in Figure 1. Once I had my input and output logic for the module, I created seven enumerated states. I used an always_comb block with a case statement for ps (present state). Inside the always_comb block I tested conditions for each state so that the correct transition to ns (next state) was taken. After the always_comb, I had an always_ff block to update ps on the positive clock edge. If the reset switch was set to high, ps would be set to the “none” state, otherwise ps would be set to ns which was updated in the always_comb.

Task #2

For the second task I first designed the counter to have 1-bit inputs clk, reset, inc, and dec that returned 1-bit clear, full, 5-bit count as outputs. I had count holding the number 25 in binary which is 11001. I was

running into the problem of taking the 5-bits and displaying it onto two separate HEX displays. Anatoliy suggested that instead of having a single 5-bit output from the counter module that I should separate it into two different 4-bit outputs: ones and tens. Following his suggestion helped when creating the carHexDisplay module for displaying to two HEX displays. For this lab, we were told that our parking lot needed to be able to take in a max of 25 cars as a parameter, but for demonstration we could set the parameter to 5. I chose to use "total" logic that is 5-bits. with total be 5-bits, I am able to count up to 31 and surpass the parameter of 25 that was given. The counter module takes the 1-bit outputs enter and exit from parkFSM as 1-bit inputs inc and dec to increase and decrease the count. This module has 1-bit outputs clear and full, as well as 4-bit outputs ones and tens. I had the two 1-bit outputs for if the counter module was at zero, or if it was at it's parameter max. These outputs were important for the carHexDisplay module so it could display to correct output to the HEX5-HEX2. The 4-bit outputs ones and tens are important to send the count over to carHexDisplay and update HEX1-HEX0.

With having two 4-bit outputs I used and always_ff block to always update the count on the positive clock edge. I used 5-bit logic I called total to hold the total number of cars in the lot. The 4-bit outputs ones, tens, and then the 5-bit logic total were all zeroed out upon reset. If 1-bit input inc was 1 and total wasn't equal to max, the parameter for number of cars that can be in the lot, total would increase by one and 4-bit output ones was tested to see if it was at decimal value of nine. If the output ones equaled the decimal value of nine, then ones would be zeroed out, and the 4-bit output tens would increase by one, but if it wasn't equal to the decimal value of nine, then ones would increase by. This can be seen in Figure 2 below.

```
//integer total;
logic [4:0] total;

//This always_ff will count up if inc is 1, and count down if dec is 1. If the
//total number of cars is at the max, the count will not increase. If the total
//number of cars is at 0, the count will not decrease. Counting up and down will
//update the 4-bit outputs ones and tens from 0-9.
always_ff @(posedge clk) begin
   if(reset) begin
      total <= '0;
      ones <= '0;
      tens <= '0;
   end

   else begin

      if(inc && (total != max)) begin
         total <= total + 1;

         if(ones == 4'd9) begin
            ones <= '0;
            tens <= tens + 1;
         end

         else begin
            ones <= ones + 1;
         end
      end

      else if(dec && (total != 0)) begin
         total <= total - 1;

         if(ones == 4'd0) begin
            ones <= 4'd9;
            tens <= tens - 1;
         end

         else begin
            ones <= ones - 1;
         end
      end

      else begin
         total <= total;
      end
   end
end
```

Fig. 2: Counter Module always_ff block

I designed the module to decrement in a similar to how I had the module increment. I first tested to see if the 1-bit input dec was 1 and also checked to see that the 5-bit logic total wasn't equal to zero. If total was equal to zero, then total was set to total, and if total wasn't zero then another if statement was used to test if 4-bit output ones equaled decimal zero. If ones did not equal decimal zero, then ones would be decreased by one, and if it did, then ones would be set to decimal value of nice and tens would decrease by one. This can be seen above in Figure 2.

I chose to design the counter this way so that the module could take in the parameter max, and the total number of cars could never exceed that. The total number of cars could also not go below zero. I chose to use two 4-bit outputs for ones and tens for simplicity when displaying the numbers to the HEX displays. I also made multiple test cases to be sure that the 4-bit outputs ones and tens would update properly. The full code for this counter module that was implemented can be found in Appendix 2.A) counter.sv.

Task #3

For the third task of this lab I designed a module that utilized the HEX displays called carHexDisplay. This module had 1-bit inputs clear and full that took the 1-bit output values clear and full from the counter module. The 1-bit inputs clear and full were tested to see if they either of them were 1. If clear is equal to 1, HEX5-HEX2 will display "CLEA". If full is equal to one, HEX5-HEX2 will display "FULL". If neither clear, or full were equal to 1, then HEX5-HEX2 wouldn't display anything and would be blank. I created localparam logic for numbers 0-9, the letters C, L, E, A, r, F, as well as U, and I also had one for blank which displayed nothing on the HEX displays. This can be seen below in Figure 3.

```
//Numbers for HEX displays to count number of cars
localparam logic [6:0] zero = 7'b1000000;   //0
localparam logic [6:0] one = 7'b1111001;    //1
localparam logic [6:0] two = 7'b0100100;    //2
localparam logic [6:0] three = 7'b0110000;  //3
localparam logic [6:0] four = 7'b0011001;   //4
localparam logic [6:0] five = 7'b0010010;   //5
localparam logic [6:0] six = 7'b0000010;    //6
localparam logic [6:0] seven = 7'b1111000;  //7
localparam logic [6:0] eight = 7'b0000000;  //8
localparam logic [6:0] nine = 7'b0010000;   //9

//Letters for hex displays to write out CLEAr and FULL
localparam logic [6:0] C = 7'b1000110;  //C
localparam logic [6:0] L = 7'b1000111;  //L
localparam logic [6:0] E = 7'b0000110;  //E
localparam logic [6:0] A = 7'b0001000;  //A
localparam logic [6:0] R = 7'b0101111;  //r
localparam logic [6:0] F = 7'b0001110;  //F
localparam logic [6:0] U = 7'b1000001;  //U

localparam logic [6:0] blank = 7'b1111111;
```

Fig. 3: Letters, Numbers, and Blank used in carHexDisplay

I used an always_comb block with one case being the 4-bit input ones, and the second being the 4-bit input tens. To set what was displayed on HEX0, the 4-bit input ones was used. The case for ones went through values 0-9, and set led0 to zero-nine, respectively. The tens case was slightly different, since if the 1-bit input clear was 1 then HEX1 will display the letter 'r'. If clear wasn't 1, then HEX1 will display

'0'. Other than this one instance, the tens case is setup the same way the ones case is. These case statements can be seen in Figure 4 and Figure 5 below.

```
always_comb begin
   case(ones)

      4'd0: begin
            led0 = zero;
      end

      4'd1: begin
            led0 = one;
      end

      4'd2: begin
            led0 = two;
      end

      4'd3: begin
            led0 = three;
      end

      4'd4: begin
            led0 = four;
      end

      4'd5: begin
            led0 = five;
      end

      4'd6: begin
            led0 = six;
      end

      4'd7: begin
            led0 = seven;
      end

      4'd8: begin
            led0 = eight;
      end

      4'd9: begin
            led0 = nine;
      end

      default: begin
            led0 = 7'bx;
      end
   endcase
```

Fig. 4: Case Statement for "ones"

To Set Output to HEX0

```
case(tens)

   4'd0: begin
         if(clear == 1) begin
            led1 = R;
            led0 = zero;
         end

         else begin
            led1 = zero;
         end
   end

   4'd1: begin
         led1 = one;
   end

   4'd2: begin
         led1 = two;
   end

   4'd3: begin
         led1 = three;
   end

   4'd4: begin
         led1 = four;
   end

   4'd5: begin
         led1 = five;
   end

   4'd6: begin
         led1 = six;
   end

   4'd7: begin
         led1 = seven;
   end

   4'd8: begin
         led1 = eight;
   end

   4'd9: begin
         led1 = nine;
   end

   default: begin
         led1 = 7'bx;
   end
endcase
end
```

Fig. 5: Case Statement for "tens"

To Set Output to HEX1

I used a second always_comb block that tested if clear was equal to 1, if full was equal to one, or neither of them were one. This always_comb is where I set the outputs for HEX5-HEX2. One reason why I had localparam's for each number, letter, or blank space used, was to just type out the 7-bits for each of those once. I already had a few of the letters, and all of the numbers from a lab in 271, so I just copied those over. I felt like using localparam's made the code inside the case statements a bit nicer, and easier to read. Another reason as to why I implemented the module this way was I switched from having a 5-bit output in my counter module, to having two 4-bit outputs. This required two sets of case statements to set two separate HEX displays. The full code for this module can be found in Appendix 3.A) carHexDisplay.sv.

Task #4

Task 4 is where I put everything together that I made prior in the top module DE1_SoC. I brought each module from tasks 1-3, and well as a double DFF module I made in 271 to prevent metastability. Since I had use the GPIO pins and a breadboard for this part of the virtual lab, I knew I needed to have three 1-bit logics for the switches. SWA, SWB, and reset were set to pins for GPIO_0. SWA was set to GPIO_0[6], SWB was set to GPIO_0[7], and reset was set to GPIO_0[5]. Since I need output to two LED's from the switches, I had GPIO_0[26] = SWA, and GPIO_0[27] = SWB. This allowed the switches to give input to the system through pins 6 and 7, and then output signal to pins 26 and 27 for the LED's. For the setup on the breadboard, the left switch is SWA, the middle-upper switch is reset, and the right switch is SWB. The LED associated with SWA is the one on the left, while the LED associated with SWB is the one on the right. I also color coordinated the wires: yellow is for SWA/sensor A, green is for SWB/sensor B, and reset has the red wire. My breadboard setup can be seen below in Figure 6.

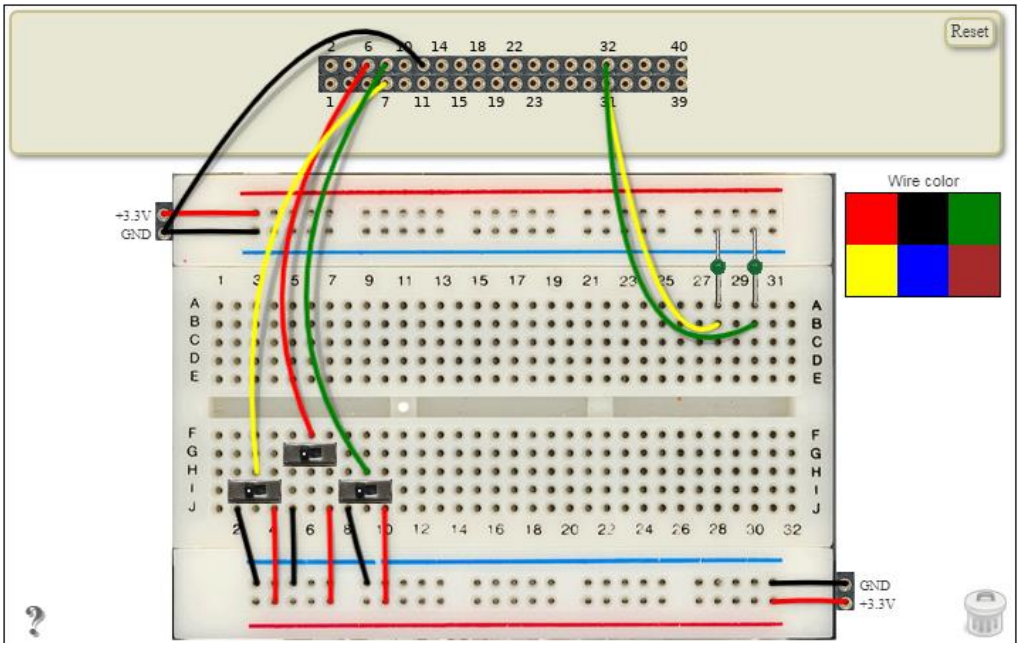

Fig. 6: Breadboard Setup for LED's and Switches

Once I had the inputs from the switches set, I began creating the parking lot sensor system. I first started by creating logic to hold input and output values from each module. I then passed the switch inputs through a double DFF. The outputs sensorA and sensorB from the double DFF's were passed into parkFSM (the parking lot FSM). With the switches passed into parkFSM, the outputs entering and exiting could be 1 or 0 depending on the sequence in which the switches were flipped. These outputs from the FSM were passed to the counter module. The counter module would increase, or decrease depending on if entering, or exiting were 1. This would change the outputs from the counter module, and pennies, and/or dimes would increase, or decrease, depending on if the system was at 0, or the max value based on the parameter. The outputs allOpen, and allFull would change along with the count as well. The counter modules outputs were then passed to the carHexDisplay module. Using what was passed to carHexDisplay, the seven-segment HEX displays, HEX5-HEX0, were updated according to display if the lot was clear, full, as well as the number of cars in it. A block diagram for DE1_SoC is shown below is Figure 7. The full code for the DE1_SoC module can be found in Appendix 4.A) DE1_SoC.sv.

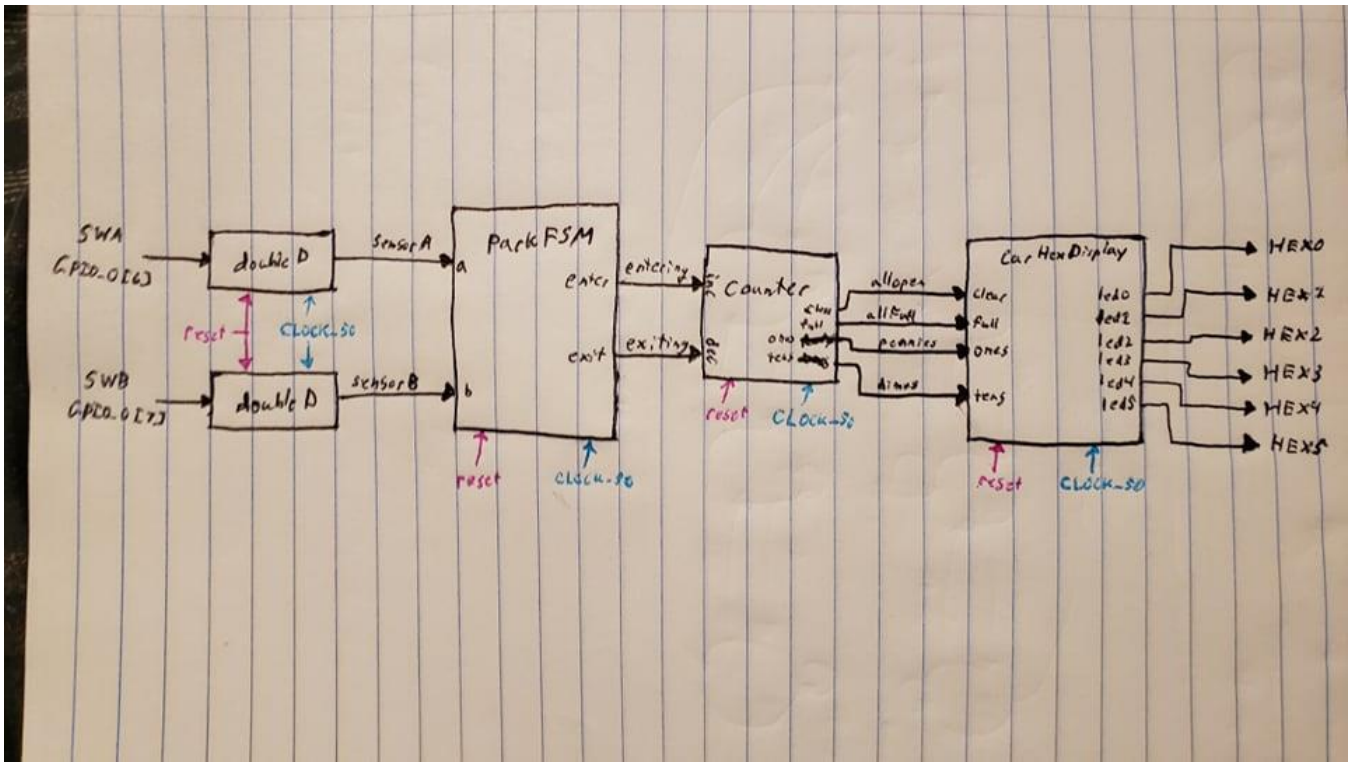

Fig 7: Block Diagram of DE1_SoC Module

Results

Task #1

For the first task of creating the parking lots FSM, I created a testbench that first tested if there was a pedestrian entering the parking lot. I simulated this by triggering only one sensor at a time starting with sensor a, then sensor b. Each sensor is only triggered for one clock cycle, and both were not triggered at the same time, thus, simulating a person walking through the sensor. Next, I simulated a car entering the parking lot, which triggered the same sensors as the pedestrian, but there was a point both sensors were triggered. This causes "enter" to go high for one clock cycle. After that I tested a car exiting the parking lot. Finally, I finished by having a car go through most of the for entering and exiting to be sure that a car could back up and not enter/exit the lot without having enter, or exit go high. All of this can be seen below in Figure 8.

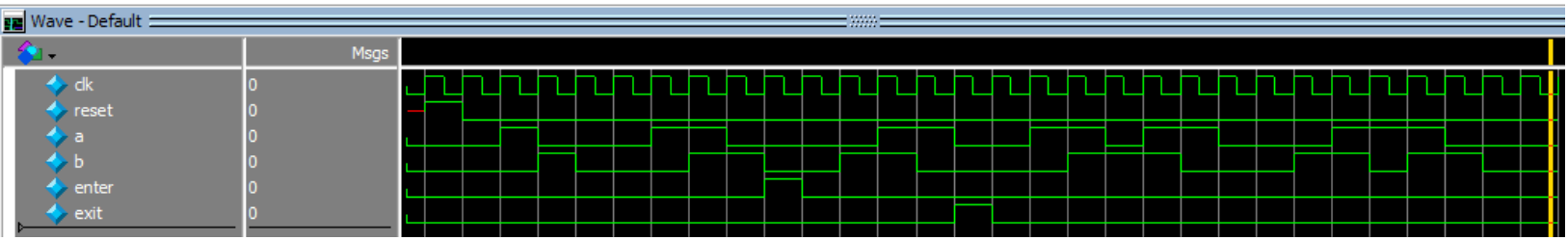

Fig. 8: Simulation of Parking Lot FSM for Different Test Cases

The results from the testbench for Task #1 were as expected. For the pedestrian, both sensors would need to be triggered at the same time after one sensor was trigger, followed by just the second sensor triggered prior to no sensors triggered to be able to have enter, or exit go high for one clock cycle. This is demonstrated with second and third tests with a car triggering the first sensor, both sensors, the second sensor, and then no sensors. This will cause enter, or exit to go high for one clock cycle. The last two tests also resulted in expected behavior, because they don't transition the same way the car entering, and exiting does in the second and third test.

Task #2

The second task was to create a counter module for the system to keep track of cars in the parking lot. I set the parameter max to 11 for this testbench. I first set inc high for one clock cycle. This caused clear to go to 0, and ones to increase by one. Next, I set dec high for three clock cycles. This was to test if ones would decrease after being at 0. I then set inc high for 13 clock cycles to have the max number of cars in the lot so full would go high, and made sure the count didn't exceed the parameter max. These tests can be seen below if Figure 9.

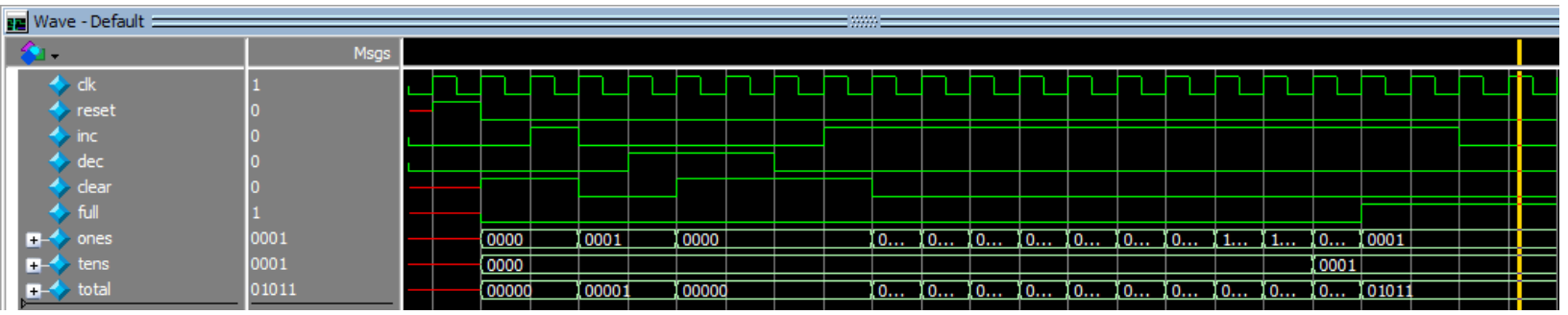

Fig. 9: Simulation of the Counter Module for Different Test Cases

The results from this testbench for Task #2 were as expected. If ones and tens are both 0, then clear will be high, and if you try to decrease while ones is ‘0 and clear is high then nothing will happen. Similarly, you can increase to the max, ones and tens will increase accordingly. Once the max is hit, full is set high, and ones/tens don’t increase anymore.

Task #3

The third task I made a module that could display the number of cars in the lot to the seven-segment HEX displays. I first tested when clear was high to see if HEX5-HEX0 would display “CLEAr0” since there were no cars in the lot and it was clear. I then increased ones and tens all the way up to 15 and set full high to see if the HEX displays would display the correct numbers accordingly. I repeated this test, but in reverse order and decreased from 15 to 0. I first set full low and decreased ones and tens to 0. These tests can be seen below in Figure 10.

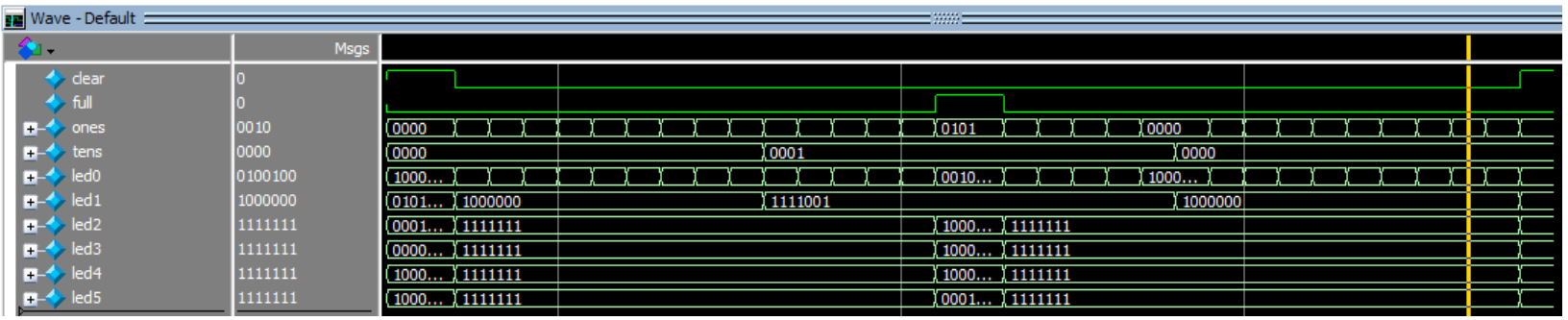

Fig. 10: Simulation of carHexDisplay Module to Visualize HEX Display Updating

The results from this testbench for Task #3 were as expected. If clear is high, HEX5-HEX0 display correctly. If full is high, HEX5-HEX0 display correctly. Increasing, and decreasing ones and tens updates HEX1-HEX0 correctly. If ones and tens are both not 0, and full is low, then HEX5-HEX2 are all ones to produce a blank display.

Task #4

For Task #4, everything was put together in DE1_SoC. I first tested for a pedestrian to make sure the FSM didn’t update. I then had 5 cars enter the lot until the parameter max was reached to see if the HEX displays would display correctly. I then tested 5 cars leaving the lot to reach 0 cars total in the lot to see

if the HEX displays would update accordingly. These three tests that I simulated can be seen below in Figure 11.

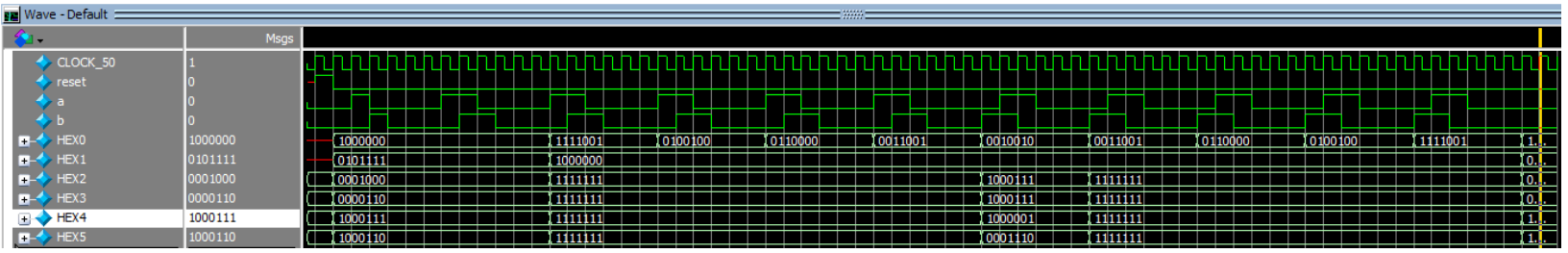

Fig. 11: Simulation of Top Module DE1_SoC

The results from this testbench for Task #4 were as expected. I ran three tests to be sure that the pedestrian wouldn't increase the cars in the lot, that cars could enter the lot which would increase the count to HEX0-HEX1 until the max was reach and then HEX5-HEX2 would display "FULL", and then cars leaving the lot would update HEX1-HEX0 with the count until no cars were left and then HEX5-HEX1 displayed "CLEAr". The HEX displays updated like they should. When there were no cars in the lot, "CLEAr" was displayed on HEX5-HEX1 and HEX0 display 0. As cars entered, HEX5-HEX2 displayed nothing and were blank. Once the max number of cars entered HEX5-HEX2 displayed "FULL" and HEX1-HEX0 displayed the count. Each HEX display updated as it was expected to as cars entered, and exited the lot.

Final Project

The goal of this project was to create a parking lot sensor that detects when a car enters, or exits the parking lot. The system was supposed to keep track of the number of cars in the parking lot. This lab was different since it was virtual, and since we used GPIO_0, which I hadn't written code for before. The use of an FSM in System Verilog was a nice refresher to how they work, and it was a bit challenging trying to remember this programming language, and how to use ModelSim. This lab helped me remember things from Spring quarter when I took EE 271. I learned some new tricks with ModelSim, as well as System Verilog code.

My lab produced the results I was working towards, and I believe I was able to complete the requirements in doing so. The system accounts for cars entering, and exiting, I used a parameterized module, which I had not written one before, and the case for a pedestrian was taken care of and doesn't cause the count of cares to increase, or decrease.

Appendix: System Verilog Code

1.A) parkFSM.sv

```
1   //Garrett Tashiro
2   //October 7, 2021
3   //EE 371
4   //Lab 1, Task 1
5
6   //Module parkFSM takes 1-bit clk, reset, a, and b as input logic and returns 1-bit enter, and exit
7   //as outputs. This module implements a parking lot gate with two sensors to detect is a car is
8   //leaving, or entering the parking lot.
9   module parkFSM(clk, reset, a, b, enter, exit);
10        input logic    clk, reset, a, b;
11        output logic   enter, exit;
12
13
14        enum{none, enterA, enterAB, enterB, exitB, exitAB, exitA} ps, ns;
15
16        //This always_comb is for the parking lots FSM that takes in seven states.
17        //The transition between states is triggered by the 1-bit inputs from
18        //a and b. The FSM starts in state none.
19        always_comb begin
20           case(ps)
21
22              none: begin
23
24                    if(a == 1 && b == 0) begin  //Start enter sequence
25                       exit = 0;
26                       enter = 0;
27                       ns = enterA;
28                    end
29
30                    else if(a == 0 && b == 1) begin  //Start exit sequence
31                       exit = 0;
32                       enter = 0;
33                       ns = exitB;
34                    end
35
36                    else begin  //Stay in none
37                       exit = 0;
38                       enter = 0;
39                       ns = none;
40                    end
41              end
42
43              enterA: begin  //Entering with only sensor a blocked
44
45                    if(a == 1 && b == 1) begin  //Both sensors are triggered for enter
46                       exit = 0;
47                       enter = 0;
48                       ns = enterAB;
49                    end
50
51                    else if(a == 0 && b == 0) begin  //No sensors are triggered
52                       exit = 0;
```

```
                end

                else if(a == 0 && b == 0) begin  //No sensors are triggered
                    exit = 0;
                    enter = 0;
                    ns = none;
                end

                else begin  //Stay in enterA
                    exit = 0;
                    enter = 0;
                    ns = enterA;
                end
            end

            enterAB: begin  //Entering with both sensors blocked

                if(a == 0 && b == 1) begin  //Only sensor b is triggered in enter sequence
                    exit = 0;
                    enter = 0;
                    ns = enterB;
                end

                else if(a == 1 && b == 0) begin  //Only sensor a is triggered in enter sequence
                    exit = 0;
                    enter = 0;
                    ns = enterA;
                end

                else begin  //Stay in enterAB
                    exit = 0;
                    enter = 0;
                    ns = enterAB;
                end
            end

            enterB: begin

                if(a == 0 && b == 0) begin //Enter sequence finished
                    exit = 0;
                    enter = 1;  //Car has entered the lot. Increase enter.
                    ns = none;
                end

                else if(a == 1 && b == 1) begin //Both sensors are triggered for exit
                    exit = 0;
                    enter = 0;
                    ns = enterAB;
                end

                else begin  //Stay in enterB
                    exit = 0;
```

```
                else begin  //Stay in enterB
                    exit = 0;
                    enter = 0;
                    ns = enterB;
                end
            end

            exitB: begin

                if(a == 0 && b == 0) begin  //No sensors triggered
                    exit = 0;
                    enter = 0;
                    ns = none;
                end

                else if(a == 1 && b == 1) begin  //Both sensors are triggered in exit sequence
                    exit = 0;
                    enter = 0;
                    ns = exitAB;
                end

                else begin  //Stay in exitB
                    exit = 0;
                    enter = 0;
                    ns = exitB;
                end
            end

            exitAB: begin

                if(a == 0 && b == 1) begin  //Only sensor B is triggered in exit sequence
                    exit = 0;
                    enter = 0;
                    ns = exitB;
                end

                else if(a == 1 && b == 0) begin  //Only sensor A is triggered in exit sequence
                    exit = 0;
                    enter = 0;
                    ns = exitA;
                end

                else begin  //Stay in exitAB
                    exit = 0;
                    enter = 0;
                    ns = exitAB;
                end
            end

            exitA: begin
```

```
147 
148             exitA: begin
149 
150                 if(a == 0 && b == 0) begin  //Exit sequence finished
151                    exit = 1;  //Car exited lot. Set exit variable to 1.
152                    enter = 0;
153                    ns = none;
154                 end
155 
156                 else if(a == 1 && b == 1) begin  //Both sensors triggered in exit seque
157                    exit = 0;
158                    enter = 0;
159                    ns = exitAB;
160                 end
161 
162                 else begin  //Stay in exitA
163                    exit = 0;
164                    enter = 0;
165                    ns = exitA;
166                 end
167             end
168          endcase
169       end
170 
171       //This always_ff will set ps to none if the 1-bit input reset is
172       //set to 1, otherwise ps will be set to next state upon the
173       //positive clock edge
174       always_ff @(posedge clk) begin
175          if (reset) begin
176             ps <= none;
177          end
178 
179          else begin
180             ps <= ns;
181          end
182       end
183 endmodule
```

1.B) parkFSM.sv (testbench)

```
//parkFSM_testbench tests all expected, unexpected, and edgecase behavior of the
//parking lot FSM that is implemented in the lab. The testbench starts by first
//testing a pedestrian triggering sensors, then the next two tests are a car entering
//and exiting the lot, while the last two tests are trigger all sensors from entering
//or exiting, but not going from the last sensor to no sensors to enter/exit, then
//back tracking through the states.
module parkFSM_testbench();
      logic clk, reset, a, b, enter, exit;

      parkFSM dut(.clk, .reset, .a, .b, .enter, .exit);

      parameter CLOCK_PERIOD = 100;
      initial begin
            clk <= 0;
            forever #(CLOCK_PERIOD/2) clk <= ~clk;// Forever toggle the clock
      end

      initial     begin
         a <= 0; b <= 0;  repeat(1)    @(posedge    clk);
         reset <= 1;      repeat(1)    @(posedge    clk);
         reset <= 0;      repeat(1)    @(posedge    clk);
         a <= 1;          repeat(1)    @(posedge    clk);  //Pedestrian start
         a <= 0; b <= 1;  repeat(1)    @(posedge    clk);
         //b <= 1;           repeat(1)    @(posedge    clk);
         b <= 0;          repeat(1)    @(posedge    clk);  //Pedestrian end
                          repeat(1)    @(posedge    clk);
         a <= 1;          repeat(1)    @(posedge    clk);  //Enter start
         b <= 1;          repeat(1)    @(posedge    clk);
         a <= 0;          repeat(1)    @(posedge    clk);
         b <= 0;          repeat(1)    @(posedge    clk);  //Enter end
                          repeat(1)    @(posedge    clk);
         b <= 1;          repeat(1)    @(posedge    clk);  //Exit start
         a <= 1;          repeat(1)    @(posedge    clk);
         b <= 0;          repeat(1)    @(posedge    clk);
         a <= 0;          repeat(1)    @(posedge    clk);  //Exit end
                          repeat(1)    @(posedge    clk);
         a <= 1;          repeat(1)    @(posedge    clk);  //Testing from one side to another
         b <= 1;          repeat(1)    @(posedge    clk);  //and back without triggering enter
         a <= 0;          repeat(1)    @(posedge    clk);
         a <= 1;          repeat(1)    @(posedge    clk);
         b <= 0;          repeat(1)    @(posedge    clk);
         a <= 0;          repeat(1)    @(posedge    clk);
                          repeat(1)    @(posedge    clk);
         b <= 1;          repeat(1)    @(posedge    clk);  //Testing from one side to another
         a <= 1;          repeat(1)    @(posedge    clk);  //and back without triggering exit
         b <= 0;          repeat(1)    @(posedge    clk);
         b <= 1;          repeat(1)    @(posedge    clk);
         a <= 0;          repeat(1)    @(posedge    clk);
         b <= 0;          repeat(1)    @(posedge    clk);
                          repeat(1)    @(posedge    clk);

         $stop; // End the simulation.
      end
endmodule
```

2.A) counter.sv

```
1   //Garrett Tashiro
2   //October 7, 2021
3   //EE 371
4   //Lab 1, Task 2
5
6   //counter module takes 1-bit clk, reset, inc, and dec as inputs
7   //and returns 1-bit clear, full, and 4-bit ones and tens. This counter module
8   //is designed to count the number of cars coming and out of the
9   //parking lot. If the lot has no cars, clear will be set to 1 and
10  //can't go below 0. If the max number is reached full will be set
11  //to 1 and no more cars can park.
12  module counter #(parameter max = 5)(clk, reset, inc, dec, clear, full, ones, tens);
13        input logic          clk, reset, inc, dec;
14        output logic         clear, full;
15        output logic [3:0]   ones, tens;
16
17        //integer total;
18        logic [4:0] total;
19
20        //This always_ff will count up if inc is 1, and count down if dec is 1. If the
21        //total number of cars is at the max, the count will not increase. If the total
22        //number of cars is at 0, the count will not decrease. Counting up and down will
23        //update the 4-bit outputs ones and tens from 0-9.
24        always_ff @(posedge clk) begin
25           if(reset) begin
26              total <= '0;
27              ones <= '0;
28              tens <= '0;
29           end
30
31           else begin
32
33              if(inc && (total != max)) begin
34                 total <= total + 1;
35
36                 if(ones == 4'd9) begin
37                    ones <= '0;
38                    tens <= tens + 1;
39                 end
40
41                 else begin
42                    ones <= ones + 1;
43                 end
44              end
45
46              else if(dec && (total != 0)) begin
47                 total <= total - 1;
48
49                 if(ones == 4'd0) begin
50                    ones <= 4'd9;
51                    tens <= tens - 1;
52                 end
53
54                 else begin
55                    ones <= ones - 1;
56                 end
57              end
58
59              else begin
60                 total <= total;
61              end
62           end
63        end
64
65        assign clear = (total == 0);
66        assign full = (total == max);
67  endmodule
```

2.A) counter.sv (testbench)

```
69  //counter_testbench tests for expected, unexpected, and edgecase behavior of the counter.
70  //The testbench is tests to make sure the counter can count up and down, doesn't decrease
71  //below 0, or increase above the max (which for this test was 5).
72  module counter_testbench();
73        logic       clk, reset, inc, dec, clear, full;
74        logic [3:0] ones, tens;
75
76        counter dut(.clk, .reset, .inc, .dec, .clear, .full, .ones, .tens);
77
78        parameter CLOCK_PERIOD = 100;
79        initial begin
80              clk <= 0;
81              forever #(CLOCK_PERIOD/2) clk <= ~clk;// Forever toggle the clock
82        end
83
84        initial     begin
85           inc <= 0; dec <= 0;  repeat(1)    @(posedge    clk); //This is just to test a max of 5
86           reset <= 1;       repeat(1)    @(posedge    clk);
87           reset <= 0;       repeat(1)    @(posedge    clk);
88           inc <= 1;         repeat(1)    @(posedge    clk);
89           inc <= 0;         repeat(1)    @(posedge    clk);
90           dec <= 1;         repeat(1)    @(posedge    clk);
91           dec <= 1;         repeat(1)    @(posedge    clk);
92           dec <= 1;         repeat(1)    @(posedge    clk);
93           dec <= 0;         repeat(1)    @(posedge    clk);
94           inc <= 1;         repeat(7)    @(posedge    clk);
95           inc <= 0;         repeat(2)    @(posedge    clk);
96
97           $stop; // End the simulation.
98        end
99  endmodule
```

3.A) carHexDisplay.sv

```
  1  //Garrett Tashiro
  2  //October 8, 2021
  3  //EE 371
  4  //Lab 1, Task 3
  5
  6  //carHexDisplay has 1-bit clear, 1-bit full, 4-bit ones, and 4-bit tens as
  7  //inputs and returns 7-bit led0, 7-bit led1, 7-bit led2, 7-bit led3, 7-bit
  8  //led4, and 7-bit led5 as outputs. This module is designed to display the
  9  //ouputs for the count from the counter module to the hex displays. If clear
 10  //is 1 then the hex displays will display "CLEAr" followed by a 0. If full
 11  //is 1 then hex displays will display "FULL" followed by the number of cars
 12  //in the lot. If neither clear, or full are 1, only the number of cars is displayed.
 13  module carHexDisplay(clear, full, ones, tens, led0, led1, led2, led3, led4, led5);
 14        input logic          clear, full;
 15        input logic [3:0]    ones, tens;
 16        output logic [6:0]   led0, led1, led2, led3, led4, led5;
 17
 18        //Numbers for HEX displays to count number of cars
 19        localparam logic [6:0] zero = 7'b1000000;   //0
 20        localparam logic [6:0] one = 7'b1111001;    //1
 21        localparam logic [6:0] two = 7'b0100100;    //2
 22        localparam logic [6:0] three = 7'b0110000;  //3
 23        localparam logic [6:0] four = 7'b0011001;   //4
 24        localparam logic [6:0] five = 7'b0010010;   //5
 25        localparam logic [6:0] six = 7'b0000010;    //6
 26        localparam logic [6:0] seven = 7'b1111000;  //7
 27        localparam logic [6:0] eight = 7'b0000000;  //8
 28        localparam logic [6:0] nine = 7'b0010000;   //9
 29
 30        //Letters for hex displays to write out CLEAr and FULL
 31        localparam logic [6:0] C = 7'b1000110;  //C
 32        localparam logic [6:0] L = 7'b1000111;  //L
 33        localparam logic [6:0] E = 7'b0000110;  //E
 34        localparam logic [6:0] A = 7'b0001000;  //A
 35        localparam logic [6:0] R = 7'b0101111;  //r
 36        localparam logic [6:0] F = 7'b0001110;  //F
 37        localparam logic [6:0] U = 7'b1000001;  //U
 38
 39        localparam logic [6:0] blank = 7'b1111111;
```

```
42          //This always_comb block uses the 4-bit inputs ones, and tens in case statements
43          //to display the number of cars are in the lot on HEX0 and HEX1. If no cars are in
44          //the lot, clear will be 1, HEX1 will be set to 'r' to be at the end of the word
45          //"CLEAr" that is displayed on HEX5-HEX1.
46          always_comb begin
47             case(ones)
48
49                4'd0: begin
50                      led0 = zero;
51                end
52
53                4'd1: begin
54                      led0 = one;
55                end
56
57                4'd2: begin
58                      led0 = two;
59                end
60
61                4'd3: begin
62                      led0 = three;
63                end
64
65                4'd4: begin
66                      led0 = four;
67                end
68
69                4'd5: begin
70                      led0 = five;
71                end
72
73                4'd6: begin
74                      led0 = six;
75                end
76
77                4'd7: begin
78                      led0 = seven;
79                end
80
81                4'd8: begin
82                      led0 = eight;
83                end
84
85                4'd9: begin
86                      led0 = nine;
87                end
88
89                default: begin
90                      led0 = 7'bx;
91                end
92             endcase
```

```
148
149         //This always_comb is to set HEX displays HEX5-HEX2. When clear is 1
150         //HEX5-HEX2 are set to say "CLEA". When full is 1 HEX5-HEX2 are set to
151         //say "FULL". If neither full or clear are 1, then HEX5-HEX2 will be
152         //blank and not display anything.
153         always_comb begin
154
155             if(clear == 1) begin
156                 led5 <= C;
157                 led4 <= L;
158                 led3 <= E;
159                 led2 <= A;
160             end
161
162             else if(full == 1) begin
163                 led5 <= F;
164                 led4 <= U;
165                 led3 <= L;
166                 led2 <= L;
167             end
168
169             else begin
170                 led5 <= blank;
171                 led4 <= blank;
172                 led3 <= blank;
173                 led2 <= blank;
174             end
175         end
176
177         //Causes weird bug. Just stick with always_comb and
178         //get rid of clk and reset up top.
179 //      always_ff @(posedge clk) begin
180 //          if(reset || clear == 1) begin
181 //              led5 <= C;
182 //              led4 <= L;
183 //              led3 <= E;
184 //              led2 <= A;
185 //          end
186 //
187 //          else if(full == 1) begin
188 //              led5 <= F;
189 //              led4 <= U;
190 //              led3 <= L;
191 //              led2 <= L;
192 //          end
193 //
194 //          else begin
195 //              led5 <= blank;
196 //              led4 <= blank;
197 //              led3 <= blank;
198 //              led2 <= blank;
199 //          end
200 //      end
201 endmodule
```

3.B) carHexDisplay.sv (testbench)

```
203   //carHexDisplay_testbench tests to see what happens to the hex displays when clear
204   //is 0 and 1. It also tests to see what happens to the hex displays when full is
205   //set to 0 and 1. The testbench counts from 0-9 for ones, and counts from 0-1 for
206   //tens. This test counted from 0 to 15, then back down to 0, with 15 being the max.
207   module carHexDisplay_testbench();
208         logic           clear, full;
209         logic [3:0]     ones, tens;
210         logic [6:0] led0, led1, led2, led3, led4, led5;
211
212         carHexDisplay dut(.clear, .full, .ones, .tens, .led0, .led1, .led2, .led3, .led4, .led5);
213
214         initial     begin
215            ones <= '0; tens <= '0; clear <= 1; full <= 0;  #10;
216                                                            #10;
217            ones <= 4'd1; clear <= 0;                       #10;
218            ones <= 4'd2;                                   #10;
219            ones <= 4'd3;                                   #10;
220            ones <= 4'd4;                                   #10;
221            ones <= 4'd5;                                   #10;
222            ones <= 4'd6;                                   #10;
223            ones <= 4'd7;                                   #10;
224            ones <= 4'd8;                                   #10;
225            ones <= 4'd9;                                   #10;
226            ones <= '0; tens <= 4'd1;                       #10;
227            ones <= 4'd1;                                   #10;
228            ones <= 4'd2;                                   #10;
229            ones <= 4'd3;                                   #10;
230            ones <= 4'd4;                                   #10;
231            ones <= 4'd5; full <= 1;                        #10;
232                                                            #10;
233            ones <= 4'd4; full <= 0;                        #10;
234            ones <= 4'd3;                                   #10;
235            ones <= 4'd2;                                   #10;
236            ones <= 4'd1;                                   #10;
237            ones <= 4'd0;                                   #10;
238            ones <= 4'd0; tens <= 4'd0;                     #10;
239            ones <= 4'd9;                                   #10;
240            ones <= 4'd8;                                   #10;
241            ones <= 4'd7;                                   #10;
242            ones <= 4'd6;                                   #10;
243            ones <= 4'd5;                                   #10;
244            ones <= 4'd4;                                   #10;
245            ones <= 4'd3;                                   #10;
246            ones <= 4'd2;                                   #10;
247            ones <= 4'd1;                                   #10;
248            ones <= 4'd0; clear <= 1;                       #10;
249         end
250   endmodule
```

4.A) DE1_SoC.sv

```
1   //Garrett Tashiro
2   //October 10, 2021
3   //EE 371
4   //Lab 1, Task 4
5
6   //DE1_SoC has 1-bit CLOCK_50 as input, 34-bit GPIO_0 as inout, and 7-bit
7   //HEX0, HEX1, HEX2, HEX3, HEX4, and HEX5 as outputs. DE1_SoC combines
8   //the modules from previous tasks, as well as GPIO_0 to control LED's
9   //on the breadboard to act as teh parking lots sensors being triggered,
10  //and the HED displays to display the number of cars in the lot and
11  //show if the lot is full or empty (clear). This is the top-level module
12  //for the parking lot sensor system in this lab.
13  module DE1_SoC(HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, GPIO_0, CLOCK_50);
14        // SW and KEY cannot be declared if GPIO_0 is declaredon LabsLand
15        input logic          CLOCK_50;
16        inout logic [33:0]   GPIO_0;  //GPIO uses inout logic instead of input or output
17        output logic [6:0]   HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
18
19        //Assigning 1-bit logic to GPIO inputs, as well as
20        //assigning GPIO outputs for LED's to GPIO inputs
21        //from switches.
22        logic reset, SWA, SWB;
23        assign reset = GPIO_0[5];
24        assign SWA = GPIO_0[6];
25        assign SWB = GPIO_0[7];
26        assign GPIO_0[26] = SWA;
27        assign GPIO_0[27] = SWB;
28
29        //1-bit logic for outputs out from both doubleD's: sensorA, sensorB
30        //1-bit logic for outputs enter and exit from parkFSM: entering, exiting
31        //1-bit logic for outputs clear and full from counter: allOpen, allFull
32        //4-bit logic for outputs ones and tens from counter: pennies, dimes
33        logic sensorA, sensorB, entering, exiting, allOpen, allFull;
34        logic [3:0] pennies, dimes;
35
36        //doubleD SW1 takes 1-bit input CLOCK_50, 1-bit input reset, 1-bit input SWA,
37        //and outputs 1-bit sensorA. SWA is set to GPIO_0[6] to create an output that
38        //prevents metastability.
39        doubleD SW1(.clk(CLOCK_50), .reset(reset), .press(SWA), .out(sensorA));
40
41        //doubleD SW2 takes 1-bit input CLOCK_50, 1-bit input reset, 1-bit input SWB,
42        //and outputs 1-bit sensorB. SWB is set to GPIO_0[7] to create an output that
43        //prevents metastability.
44        doubleD SW2(.clk(CLOCK_50), .reset(reset), .press(SWB), .out(sensorB));
45
46        //parkFSM parkingLotFSM task 1-bit input CLOCK_50, 1-bit input reset, 1-bit input
47        //sensorA, 1-bit input sensorB, and has 1-bit output entering, 1-bit output exiting.
48        //parkingLotFSM takes the 1-bit outputs from doubleD SW1, and doubleD SW2 that to
49        //determine if a car is entering/exiting the parking lot. A vehicle has to trigger
50        //the first sensor, both sensors, just the second sensor, and then no sensors for a
51        //car to enter/exit the lot.
52        parkFSM parkingLotFSM(.clk(CLOCK_50),
53                              .reset(reset),
54                              .a(sensorA),
55                              .b(sensorB),
56                              .enter(entering),
57                              .exit(exiting));
58
59        //counter countU_D takes 1-bit input CLOCK_50, 1-bit input reset, 1-bit input entering,
60        //1-bit input exiting, and has 1-bit output allOpen, 1-bit output allFull, 4-bit output
61        //pennies, 4-bit output and dimes. This modulecounts the number of cars that are in the
62        //lot. If no cars are in the lot, allOut will be high. If the max number of cars are in
63        //the lot, allFull will be high. The 1-bit inputs entering and exiting are used to
64        //increment/decrement the total number of cars in the lot which is output using a combination
65        //of 4-bit outputs pennies, and dimes.
66        counter countU_D(.clk(CLOCK_50),
67                         .reset(reset),
68                         .inc(entering),
69                         .dec(exiting),
70                         .clear(allOpen),
71                         .full(allFull),
72                         .ones(pennies),
73                         .tens(dimes));
74
75        //carHexDisplay displayCars takes 1-bit input allOpen, 1-bit input allFull, 4-bit
76        //input penies, 4-bit input dimes, and 7-bit outputs HEX0-HEX5. This module is designed
77        //to display the total number of cars in the parking lot on HEX1 and HEX0. If there are
78        //no cars in the lot then HEX5-HEX1 will display "CLEAr" while HEX0 displays 0. If the
79        //lot is full then HEX5-HEX2 will display "FULL" while HEX1 and HEX0 will display the
80        //number of cars in the parking lot.
81        carHexDisplay displayCars(.clear(allOpen),
82                                  .full(allFull),
83                                  .ones(pennies),
84                                  .tens(dimes),
85                                  .led0(HEX0),
86                                  .led1(HEX1),
87                                  .led2(HEX2),
88                                  .led3(HEX3),
89                                  .led4(HEX4),
90                                  .led5(HEX5));
91  endmodule
```

4.B) DE1_SoC.sv (testbench)

```
93  //DE1_SoC_testbench tests all expected, unexpected, and edgecase behavior that the parking
94  //lot sensor system implemented in the lab. This module tests for a pedestrian walking through
95  //the sensors into the lot, then the module has 5 cars enter the lot to reach the max, followed
96  //by 5 cars exiting the lot to show the lots be empty.
97  module DE1_SoC_testbench();
98        logic          CLOCK_50;
99        wire [33:0]    GPIO_0;
100       logic [6:0]    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
101
102       logic reset, a, b;
103
104       DE1_SoC dut(.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .GPIO_0, .CLOCK_50);
105
106       parameter CLOCK_PERIOD = 100;
107       initial begin
108             CLOCK_50 <= 0;
109             forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;// Forever toggle the clock
110       end
111
112       assign GPIO_0[5] = reset;
113       assign GPIO_0[6] = a;
114       assign GPIO_0[7] = b;
115
116       initial     begin
117          a <= 0; b <= 0;          repeat(1)    @(posedge    CLOCK_50);
118          reset <= 1;              repeat(1)    @(posedge    CLOCK_50);
119          reset <= 0;              repeat(1)    @(posedge    CLOCK_50);
120          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);  //Pedestrians
121          a <= 0; b <= 1;          repeat(1)    @(posedge    CLOCK_50);
122          //b <= 1;                   repeat(1)    @(posedge    CLOCK_50);
123          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);
124                                   repeat(2)    @(posedge    CLOCK_50);
125          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);
126          b <= 1;                  repeat(1)    @(posedge    CLOCK_50);
127          a <= 0;                  repeat(1)    @(posedge    CLOCK_50);
128          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);  //1 enter.
129                                   repeat(2)    @(posedge    CLOCK_50);
130          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);
131          b <= 1;                  repeat(1)    @(posedge    CLOCK_50);
132          a <= 0;                  repeat(1)    @(posedge    CLOCK_50);
133          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);  //2 enter
134                                   repeat(2)    @(posedge    CLOCK_50);
135          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);
136          b <= 1;                  repeat(1)    @(posedge    CLOCK_50);
137          a <= 0;                  repeat(1)    @(posedge    CLOCK_50);
138          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);  //3 enter
139                                   repeat(2)    @(posedge    CLOCK_50);
140          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);
141          b <= 1;                  repeat(1)    @(posedge    CLOCK_50);
142          a <= 0;                  repeat(1)    @(posedge    CLOCK_50);
143          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);  //4 enter
144                                   repeat(2)    @(posedge    CLOCK_50);
145          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);
146          b <= 1;                  repeat(1)    @(posedge    CLOCK_50);
147          a <= 0;                  repeat(1)    @(posedge    CLOCK_50);
148          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);  //5 enter. Full.
149                                   repeat(2)    @(posedge    CLOCK_50);
150          b <= 1;                  repeat(1)    @(posedge    CLOCK_50);
151          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);
152          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);
153          a <= 0;                  repeat(1)    @(posedge    CLOCK_50);  //1 exit. 4 total.
154                                   repeat(2)    @(posedge    CLOCK_50);
```

```
155          b <= 1;                  repeat(1)    @(posedge    CLOCK_50);
156          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);
157          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);
158          a <= 0;                  repeat(1)    @(posedge    CLOCK_50);  //2 exit. 3 total.
159                                   repeat(2)    @(posedge    CLOCK_50);
160          b <= 1;                  repeat(1)    @(posedge    CLOCK_50);
161          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);
162          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);
163          a <= 0;                  repeat(1)    @(posedge    CLOCK_50);  //3 exit. 2 total.
164                                   repeat(2)    @(posedge    CLOCK_50);
165          b <= 1;                  repeat(1)    @(posedge    CLOCK_50);
166          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);
167          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);
168          a <= 0;                  repeat(1)    @(posedge    CLOCK_50);  //4 exit. 1 total.
169                                   repeat(2)    @(posedge    CLOCK_50);
170          b <= 1;                  repeat(1)    @(posedge    CLOCK_50);
171          a <= 1;                  repeat(1)    @(posedge    CLOCK_50);
172          b <= 0;                  repeat(1)    @(posedge    CLOCK_50);
173          a <= 0;                  repeat(1)    @(posedge    CLOCK_50);  //5 exit. 0 total. Clear.
174                                   repeat(4)    @(posedge    CLOCK_50);
175
176          $stop; // End the simulation.
177       end
178 endmodule
179
```

4.C) doubleD.sv

```
1   //Garrett Tashiro
2   //October 10, 2021
3   //EE 371
4   //Lab 1, Task 4
5
6   //doubleD has 1-bit clk, reset, and press as inputs, and
7   //returns 1-bit out. This module is a double DFF (two in series)
8   //so the input signal from a switch, or button to prevent metastability.
9   module doubleD(clk, reset, press, out);
10     output logic  out;
11     input  logic  press, reset, clk;
12
13     logic temp1;
14
15     //always_ff replicates a double DFF. The 1-bit input press goes into the
16     //first DFF and the output from the first DFF is the input for the
17     //second DFF.
18     always_ff @(posedge clk) begin
19        if (reset) begin
20                   temp1 <= 0;
21                   out   <= 0;
22              end
23        else begin
24                   temp1 <= press;
25                   out   <= temp1;
26              end
27     end
28
29  endmodule
30
31
```