Brian Dallaire

EE371

4/14/2021

Lab 1 Report

## Procedure

This lab was compromised of four tasks. The first task was to design an FSM for the parking lot indicating when cars enter or exit the lot and code this in System Verilog. The second task was to design a counter that will keep track of the total number of cars in the parking lot. The third task was to create indicators to display the number of cars. The number of cars will be visually indicated by the HEX displays on the DE1_SoC board. The fourth task was to combine the parking lot FSM, counter, and indicator to create a parking lot that tracks when a car enters or exits the lot and keeps track and displays the number of cars. The activation of sensors will be indicated by LEDs on the breadboard of the DE1_SoC.

## Task #1

The first task was to make an FSM for the parking lot. To do this, I started with the key information on what is considered a car entering and exiting the parking lot. There are two sensors, a and b, and the order of which these sensors detect the car will determine if a car has entered or exited the lot. Sensors a and b are the input signals for my FSM. The output signals are enter and exit. After a car enters or exits the lot, the corresponding output signal will be true for one clock cycle.

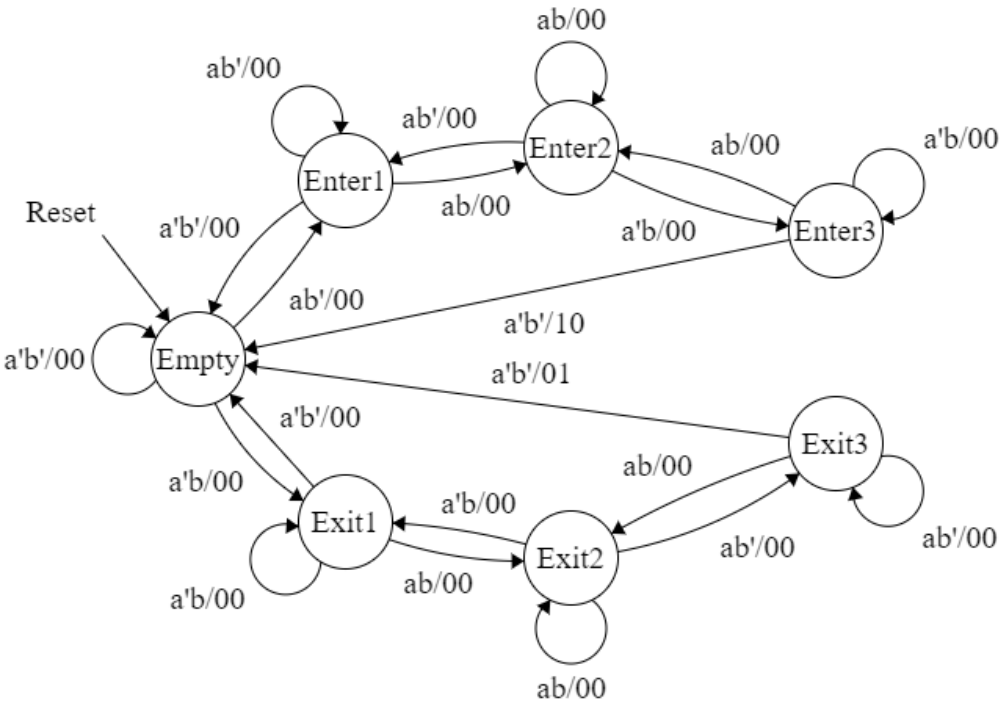


Figure 1: FSM Design for Parking Lot. a and b are input signals and the output signals are represented by enter and exit, respectively.

The main idea for creating this FSM was to differentiate when the car is entering and exiting the lot. In addition, it had to be capable of handling objects that are not cars that activate the sensors, such as pedestrians. To differentiate enter and exit, I created two branches that lead to an output signal that is true. One branch has to follow the exact order that indicates a car enters the lot (a'b' → ab' → ab → a'b → a'b'), and the other branch has to follow the exact order that indicates a car exits the lot (a'b' → a'b → ab → ab' → a'b'). Thus, the total number of states in my FSM is seven states. In the case of a pedestrian, (a'b' → ab' → a'b' → a'b → a'b') the enter and exit outputs do not evaluate to true. Even in the case where a car changes direction (i.e. the car reaches state enter3 but reverses to enter2, then goes to enter3 again and then to empty, fully entering the lot after changing direction once) the FSM will properly output true for enter when the car fully enters the lot.

As for the implementation of this FSM into SystemVerilog code, refer to appendix 1.A. The overall implementation was simple. I used enumerated states to declare all seven states of my FSM. Then, under an always_comb block I established the conditions for each transition between states, with an always_ff block moving the present state to the next state on each positive edge of the clock. The always_ff block also initializes the state to "empty" after a reset. Lastly, I used assign to determine when the output enter and exit are true. Because of the design of my FSM, enter will only be true after it reaches the state "enter3" and the next state is "empty". The output exit will only be true after it reaches the state "exit3" and the next state is

“empty”. In my code, these output signals are represented by incr (increment) for enter and decr (decrement) for exit.

**Task #2**

The second task was to design a counter with two control signals, inc and dec, that can increment and decrement the counter when they are asserted. The max amount the counter can reach must be parameterized, to allow flexibility of the number of cars that can fit in the parking lot. In this lab, the parking lot must be able to change from a max number of 25 cars to 5 cars. To allow this, I used an intermediate logic called totalCars to keep track of the total number of cars and created a global parameter called CAPACITY to set the maximum number of cars allowed in the lot. The totalCars logic is 5-bits, which means the parameter can go up to 31 cars, satisfying the requirement of the lab. Next, to actually count up or down, I used the output logic from the parking lot FSM module, incr and decr, and two output logics, ones (ones place) and tens (tens place), to determine the total number of cars in the lot in a way that is easily displayable for the next task.

```
logic [4:0] totalCars;

always_ff @(posedge clk) begin
   if(reset) begin
      totalCars <= '0;
      ones <= '0;
      tens <= '0;

   end else begin
      if(inc & (totalCars != CAPACITY)) begin
         totalCars <= totalCars + 1'b1;

         if(ones == 4'b1001) begin
            ones <= '0;
            tens <= tens + 4'b0001;
         end else begin
            ones <= ones + 4'b0001;
         end

      end else if (dec & totalCars != 0) begin
         totalCars <= totalCars - 1'b1;

         if(ones == 4'b0000) begin
            ones <= 4'b1001;
            tens <= tens - 4'b0001;
         end else begin
            ones <= ones - 4'b0001;
         end

      end else begin
         totalCars <= totalCars;
      end
   end
end

assign clear = (totalCars == 5'b00000);
assign full = (totalCars == CAPACITY);
```

Figure 2: Counter Implementation in SystemVerilog.

As shown in Figure 2, if incr is true and totalCars is under the parameter CAPACITY, then totalCars will increment and output logic one of the output logics between ones and tens will increment depending on their current values (i.e. if ones is at $(9)_{10}$, tens will increment by one and ones will go back to $(0)_{10}$). Likewise, if decr is true and totalCars is not at zero, then totalCars will decrement and one of ones and tens will decrement depending on their current values (i.e. if totalCars is at $(20)_{10}$ and decr is true, tens will decrement by one and ones will go to $(9)_{10}$). Two more output signals, clear and full, are indicators for when the parking lot is completely full or empty. These are implemented using assign commands, where if totalCars reached the value of CAPACITY full will be true and if totalCars is zero clear will be true.

**Task #3**

After completing the parking lot FSM and counter, and implementing them both into SystemVerilog successfully, I moved onto the third task of displaying the current number of cars on the HEX displays of the DE1_SoC board. From the previous task, we have several signals that will be required to make this work. The signals clear and full will be necessary to satisfy the unique cases where when full is true, HEX5-HEX2 will display {F, U, L, L} and when clear is true, HEX5-HEX1 will display {C, L, E, A, R}. As shown in figure 3, to implement these unique cases, I used an always_comb block and added local parameters that indicate the bit patterns required to display these letters.

```
module displayCars (clear, full, ones, tens, hexArray);
   input logic clear, full;
   input logic [3:0] ones, tens;
   output logic [5:0][6:0] hexArray;

   parameter BLANK = 7'b1111111;
   parameter C = 7'b1000110;
   parameter L = 7'b1000111;
   parameter E = 7'b0000110;
   parameter A = 7'b0001000;
   parameter R = 7'b0101111;
   parameter F = 7'b0001110;
   parameter U = 7'b1000001;

   always_comb begin

      if(clear)
         hexArray[5:2] = {C, L, E, A};
      else if (full)
         hexArray[5:2] = {F, U, L, L};
      else
         hexArray[5:2] = {BLANK, BLANK, BLANK, BLANK};
   end
```

Figure 3: Implementation of Special Cases for Displaying Cars. Parameters are used to send proper signals to seg-7 displays without cluttering the screen.

Noticeably, the ‘R’ is missing in the case for {C, L, E, A, R}. This is because it will require an extra HEX display compared to {F, U, L, L}, and will require more coding to display ‘R’ on HEX1. HEX1 has to simultaneously display ‘R’ when the signal clear is true, and also show the value zero when clear is false. It is easier to see when splitting the blocks of code to be designated for HEX5-HEX2 and HEX1-HEX0. Thus, I created another always_comb block for HEX1-HEX0 to finish the special case for {C, L, E, A, R} and when the signal clear is false, have HEX1-HEX0 represent the total number of cars. More on this in the results section.

As shown in Figure 4, by using the signal clear to differentiate between the special case and regular case for HEX1-HEX0, I was able to complete the special case for {C, L, E, A, R}. By using case statements for both ones and tens individually, I was able to display the current counter value onto HEX1-HEX0 when clear is false. The combination of these two always_comb blocks will allow my code to cover the special cases and also be a simple counter displayed onto two HEX displays.

```
always_comb begin
    if(clear) begin
        hexArray[1] = R;
        hexArray[0] = 7'b1000000;
    end else begin
        case (tens)
        4'b0000: hexArray[1] = 7'b1000000; // 0
        4'b0001: hexArray[1] = 7'b1111001; // 1
        4'b0010: hexArray[1] = 7'b0100100; // 2
        4'b0011: hexArray[1] = 7'b0110000; // 3
        4'b0100: hexArray[1] = 7'b0011001; // 4
        4'b0101: hexArray[1] = 7'b0010010; // 5
        4'b0110: hexArray[1] = 7'b0000010; // 6
        4'b0111: hexArray[1] = 7'b1111000; // 7
        4'b1000: hexArray[1] = 7'b0000000; // 8
        4'b1001: hexArray[1] = 7'b0010000; // 9
        default: hexArray[1] = 7'bx;
        endcase
        case (ones)
        4'b0000: hexArray[0] = 7'b1000000; // 0
        4'b0001: hexArray[0] = 7'b1111001; // 1
        4'b0010: hexArray[0] = 7'b0100100; // 2
        4'b0011: hexArray[0] = 7'b0110000; // 3
        4'b0100: hexArray[0] = 7'b0011001; // 4
        4'b0101: hexArray[0] = 7'b0010010; // 5
        4'b0110: hexArray[0] = 7'b0000010; // 6
        4'b0111: hexArray[0] = 7'b1111000; // 7
        4'b1000: hexArray[0] = 7'b0000000; // 8
        4'b1001: hexArray[0] = 7'b0010000; // 9
        default: hexArray[0] = 7'bx;
        endcase
    end
end
```

Figure 4: Code Implementation for HEX1-HEX0. Covers the special case for when signal clear is true, and provides a visual of the counter when signal clear is false.

**Task #4**

The last task is to combine every module I have created so far inside DE1_SoC. This was a simple task since each module I have created so far is connected to each other. First, I had to determine the three main inputs into the overall system. Sensor a, sensor b, and reset have to be assigned to switches on the GPIO_0 board. Seen in Figure 5 is the wiring I used for my lab. The leftmost switch is connected to GPIO_0[5], and is assigned to reset. The middle switch is connected to GPIO_0[7] and is assigned to sensor a. The rightmost switch is connected to GPIO_0[9] and is assigned to sensor b. Furthermore, the output pin GPIO_0[26] is connected to the left LED and is assigned to the switch corresponding to sensor a. The output pin GPIO_0[27] is connected to the right LED and is assigned to the switch corresponding to sensor b. This way, when each switch toggles on and off, the corresponding LED will also toggle on and off, indicating which sensors are activated.

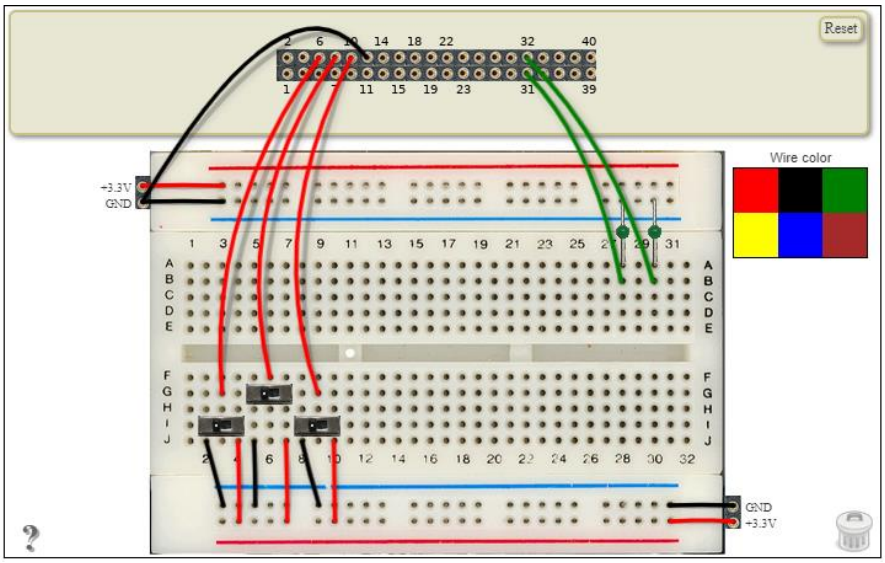


Figure 5: Wiring on DE1_SoC Breadboard for Overall Input Signals and Output LEDs

Now that I determined the overall inputs, I can begin to connect all of the modules I have created together to complete the system I created for this parking lot. First, I need to connect the sensor a and sensor b inputs into the module I created to implement the FSM of the cars entering and exiting the parking lot. The FSM I created is impacted directly by changes in sensors a and b, so this is where these inputs will go first. Next, the FSM module will output increment and decrement logic, which directly impacts the module I created for the counter. The outputs of the FSM module, incr and decr, have to connect to the module for the counter. The counter module will have four different outputs: clear, full, ones, and tens. All of these outputs will be inputted into the last module I created for displaying the special cases and the counter onto the seven segment HEX displays. The output of this module is the HEX displays HEX5 – HEX0, which will correspond to the HEX displays on the board of the DE1_SoC FPGA. A visual of these connections between each module can be seen in Figure 6.

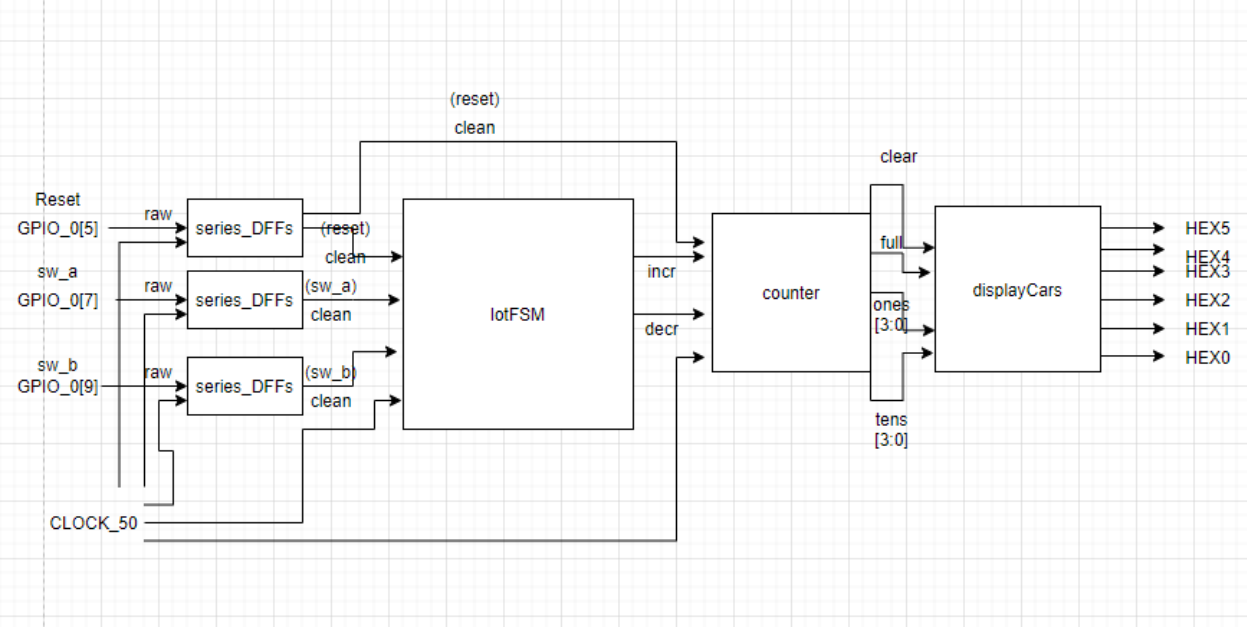


Figure 6: Block Diagram for Parking Lot System

While it may be unnecessary due to the switches being digital, there is a chance the inputs will reach metastability when pressed together too quickly. To prevent this, I also implemented DFFs in series for each overall input. So before the inputs sensor a, sensor b, and reset go into the first module (the module implementing my FSM of the lot), they will first go through two DFFs in series to prevent metastability.

## Results

### Task #1

For the first task of implementing the parking lot FSM, I created a testbench that tests a pedestrian walking by the sensors a and b, and then tested each case where the states transition in a way that enter turns true for one clock cycle and exit turns true for one clock cycle.

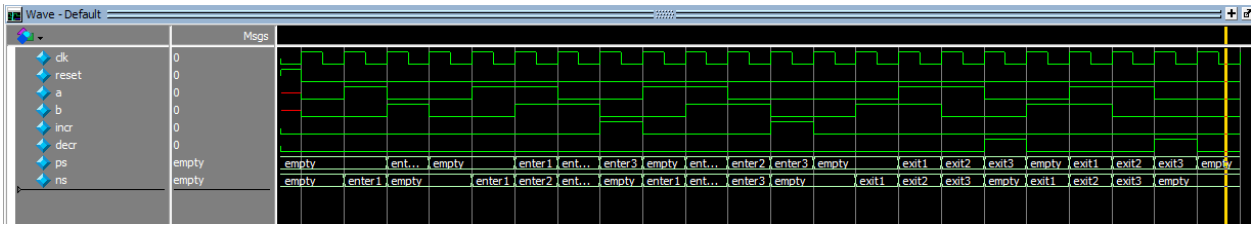


Figure 7: lotFSM ModelSim Simulating the FSM Design for Cars Entering and Exiting the Parking Lot

As shown in Figure 7, if a pedestrian walks through the sensors, incr does not go to true. However, when the sensors are triggered in the right order, indicating a car entered, increment will go to true for one clock cycle. Likewise, decrement will go true for one clock cycle when the car full exits the parking lot.

**Task #2**

For the second task, I implemented the counter. Everytime the lotFSM module sends the increment or decrement signal to be true, the counter must be updated. Represented by Figure 8 and Figure 9 is my simulation showing the functionality of my counter module. As shown in the code for my testbench in appendix 2.B, I changed the parameter for the maximum number of cars in the lot to be 12 for the testbench. This is to shorten the wavelength and also show that the counter will properly update between the ones and tens output signals. The totalCars signal also updates accordingly depending on if increment or decrement is true. In my simulation, I showed that the counter does not increment when reaching max capacity, and the counter does not decrement when it is at zero.

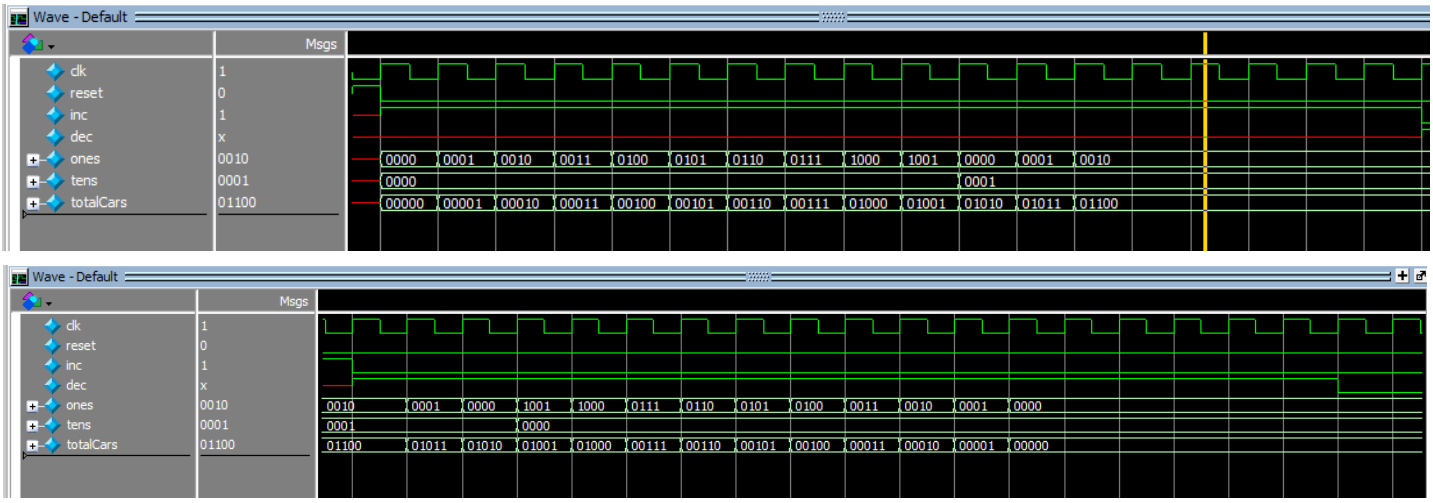


Figure 8: Counter Module ModelSim Simulation. Top is showing a zoomed in view of the counter incrementing upward, and the bottom image is showing a closer view of the counter decrementing downward.

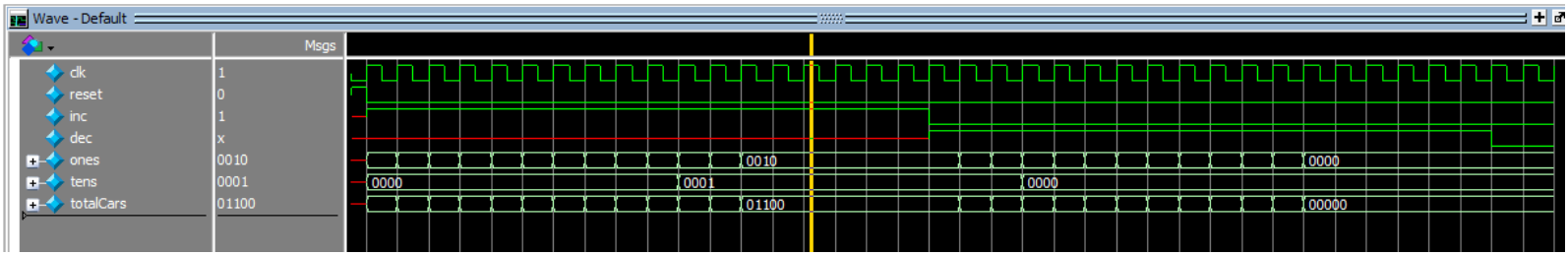


Figure 9: Counter ModelSim Full View

**Task #3**

The third task was displaying the counter values onto the HEX displays. However, the special case requires the HEX displays to show {C, L, E, A, R} in HEX5 – HEX1 when input signal clear is true, and {F, U, L, L} in HEX5 – HEX2 when input signal full is true. The testbench showcases the HEX displays displaying the proper letters when these signals are true, and HEX1 – HEX0 displaying a proper counter when neither of the clear and full signals are true.

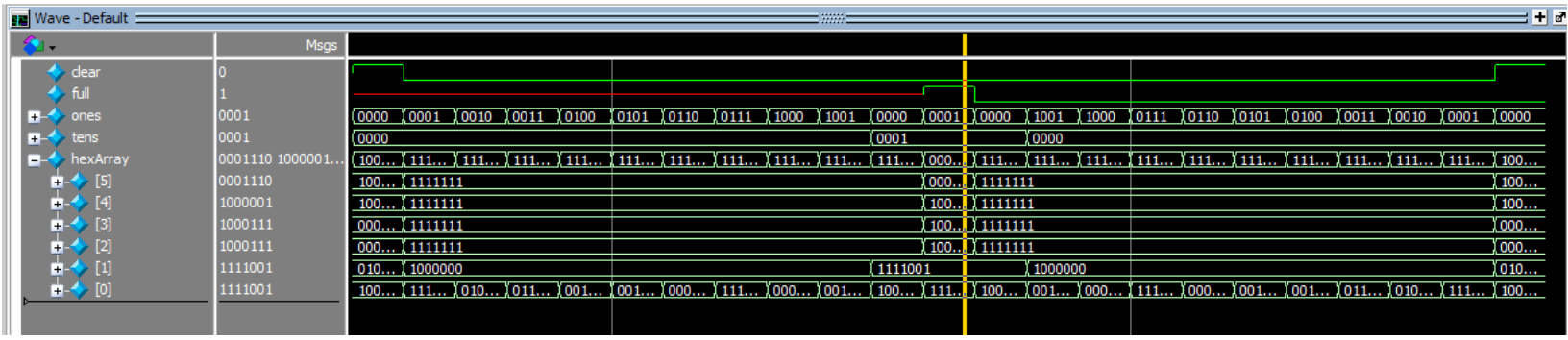


Figure 10: displayHEX ModelSim Simulation.

It may be difficult to see, so refer to Figure 11 below for close ups for each situation. I made 11 the maximum capacity to show that even when full the counter shows the proper count in HEX1 – HEX0 with {F, U, L, L} being displayed in HEX5 – HEX2. Then, when it starts to decrement HEX5 – HEX2 will turn off until it reaches {C, L, E, A, R} when the count is zero.

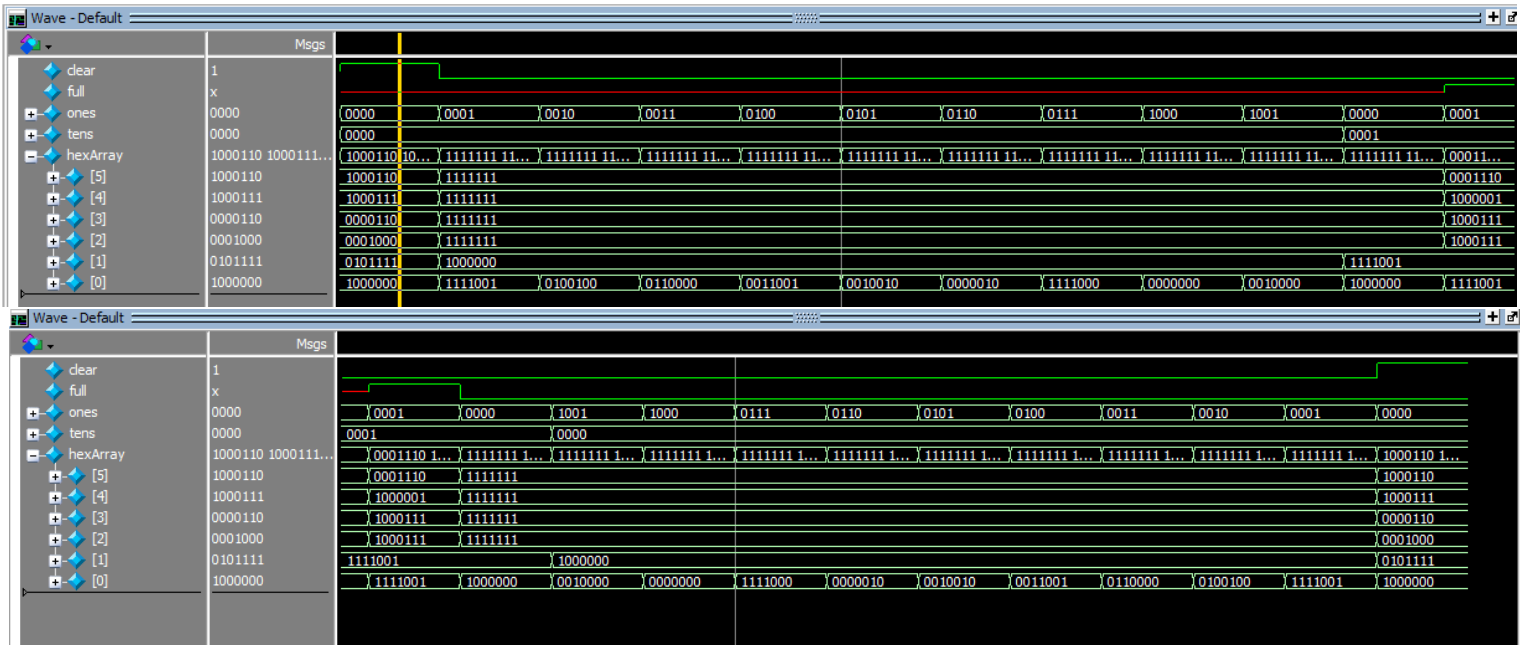


Figure 11: Close Up View of displayHEX ModelSim Simulation. Top image is the count incrementing up starting from the special case “clear”. Bottom image is counting down starting from special case “full”

**Task #4**

Putting it all together in the DE1_SoC module, we have to test and see if the switches assigned to each sensor properly trigger the FSM to change states and output increment or decrement to be true, update the counter, and update the display. Below is the simulation result of my overall DE1_SoC module testbench.

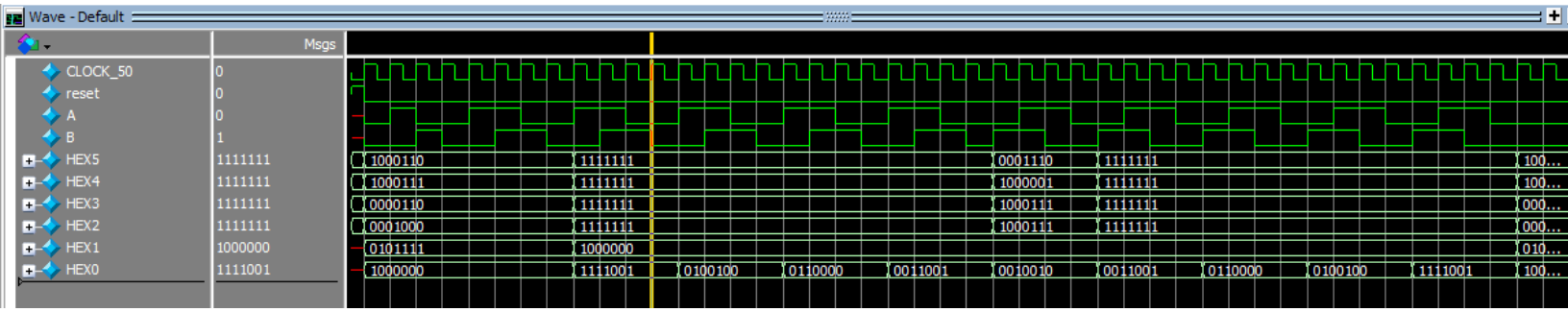


Figure 12: DE1_SoC Testbench ModelSim Simulation

To ensure each module is working properly and connected, I simulated three cases. A case in the beginning where a pedestrian walks through the sensors, which did not update the HEX displays in any way. This means the FSM was designed properly to prevent this case from having increment or decrement be true, which means the counter does not update and the displays do not update. Then, I tested what happens when the counter goes up to the maximum capacity, which is set to 5 in this simulation. The FSM updated properly and the display HEX1-HEX0 provided an accurate counter. After the first count, HEX5 – HEX2 turned off to only show the counter. Once it reached the maximum capacity, HEX5 – HEX2 displayed {F, U, L, L} and HEX1 – HEX0 kept the count displayed. Lastly, I tested what happens when the counter decrements to zero from the max capacity. The counter counted down properly, which means the FSM was designed properly. Once it reached zero, HEX5 – HEX1 updated to {C, L, E, A, R} and HEX0 displayed the number zero. Thus, my simulation compares to the ideal result of this lab.

**Final Product**

The goal of this project was to create a parking lot sensor that detects when a car enters or exits the parking lot and keeping track of the number of cars in the parking lot. It provided a good summary of implementing FSMs into SystemVerilog code, and proved to be a challenge with a new element using the GPIO_0 ports. This lab helped me relearn key things about FSMs I have forgotten from EE 271, and it helped me learn something new with the GPIO_0 ports. It taught me how the inout logic of the GPIO_0 worked, and I was excited to see the breadboard of the DE1_SoC FPGA be utilized again since I only used it twice in EE 271.

Overall, my lab provided the results I wanted and I believe it is sufficient in covering the requirements of this lab. The special cases were covered for and the primary functions work perfectly.

## Appendix

### 1.A) lotFSM.sv (code)

```
// Brian Dallaire
// 04/14/2021
// EE 371
// Lab #1, Task 1

// lotFSM takes 1-bit clk, 1-bit reset, 1-bit a, 1-bit, b as inputs and returns 1-bit incr and 1-bit decr as
// outputs. This module implements the parking lot FSM designed to detect when a car enters or exits the parking
// lot. When a car fully enters the parking lot, the 1-bit output incr will return true, and when a car fully exits
// the parking lot, the 1-bit output will return false.

module lotFSM (clk, reset, a, b, incr, decr);

   input logic clk, reset, a, b;
   output logic incr, decr;

   enum {empty, enter1, enter2, enter3, exit1, exit2, exit3} ps, ns;

   // This always_comb block uses the seven enumerated states from the FSM designed for the parking lot and uses the
   // 1-bit inputs a and b to determine the next state in the FSM. A case statement is used to easily transition the
   // FSM I designed into SystemVerilog code.

   always_comb begin
      case(ps)

      empty  : begin
                  if(a & !b)      ns = enter1;
                  else if(!a & b) ns = exit1;
                  else            ns = empty;
               end

      enter1 : begin
                  if(!a & b | !a & !b) ns = empty;
                  else if(a & b) ns = enter2;
                  else            ns = enter1;
               end

      enter2 : begin
                  if(!a & b)      ns = enter3;
                  else if(a & !b) ns = enter1;
                  else            ns = enter2;
               end

      enter3 : begin
                  if(!a & !b)     ns = empty;
                  else if (a & b) ns = enter2;
                  else            ns = enter3;
               end

      exit1  : begin
                  if(!a & !b | a & !b) ns = empty;
                  else if(a & b) ns = exit2;
                  else            ns = exit1;
               end

      exit2  : begin
```

```
      enter2 : begin
                  if(!a & b)      ns = enter3;
                  else if(a & !b) ns = enter1;
                  else            ns = enter2;
               end

      enter3 : begin
                  if(!a & !b)     ns = empty;
                  else if (a & b) ns = enter2;
                  else            ns = enter3;
               end

      exit1  : begin
                  if(!a & !b | a & !b) ns = empty;
                  else if(a & b) ns = exit2;
                  else            ns = exit1;
               end

      exit2  : begin
                  if(a & !b)      ns = exit3;
                  else if (!a & b)ns = exit1;
                  else            ns = exit2;
               end

      exit3  : begin
                  if(a & b)     ns = exit2;
                  else if(!a & !b)  ns = empty;
                  else  ns = exit3;
               end

      endcase
   end

   // car fully enters the parking lot
   assign incr = (ps == enter3 & ns == empty);

   // car fully exits the parking lot
   assign decr = (ps == exit3 & ns == empty);


   // This always_ff block is used to reset the present state of the FSM to the "empty" state when reset is true,
   // and to update the present state to the next state every positive edge of the clock when reset is false.
   always_ff @(posedge clk) begin
      if(reset) begin
         ps <= empty;

      end else begin
         ps <= ns;
      end
   end

endmodule
```

## 1.B) lotFSM.sv (testbench)

```
// lotFSM_testbench tests multiple important situations of this module. It tests the behavior of the outputs
// for outliar cases and regular cases. For example, the first test tests if a pedestrian walks by the sensors
// and the next two tests are if a car enters and exits the parking lot multiple times.
module lotFSM_testbench();

   logic clk, reset, a, b, incr, decr;

   lotFSM dut (.clk, .reset, .a, .b, .incr, .decr);

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

   initial begin
      reset <= 1;                   @(posedge clk);
      reset <= 0; a <= 0;  b <= 0; @(posedge clk);
                  a <= 1;          @(posedge clk);
                  a <= 0;  b <= 1; @(posedge clk);
                  a <= 0;  b <= 0; @(posedge clk);

      repeat(2) begin
                  a <= 1;          @(posedge clk);
                           b <= 1; @(posedge clk);
                  a <= 0;          @(posedge clk);
                  a <= 0;  b <= 0; @(posedge clk);
      end
                  a <= 0;  b <= 0; @(posedge clk);

      repeat(2) begin
                           b <= 1; @(posedge clk);
                  a <= 1;          @(posedge clk);
                           b <= 0; @(posedge clk);
                  a <= 0;          @(posedge clk);
      end
                  a <= 0;  b <= 0; @(posedge clk);
      $stop;
   end
endmodule
```

## 2.A) counter.sv (code)

```
// Brian Dallaire
// 04/14/2021
// EE 371
// Lab #1, Task 2

// counter is a module that takes 1-bit inputs clk, reset, inc, dec and outputs 1-bit outputs
// clear and full, and 4-bit outputs ones and tens. The purpose of this module is to count the
// total number of cars in the parking lot and increment and decrement accordingly

module counter #(parameter CAPACITY = 25) (clk, reset, inc, dec, clear, full, ones, tens);

   input logic clk, reset, inc, dec;
   output logic clear, full;
   output logic [3:0] ones, tens;
   logic [4:0] totalCars;


   // this always_ff block counts up when increment is true and totalCars is no the max capacity
   // it also updates ones and tens depending on if one is 9 or 0.
   always_ff @(posedge clk) begin
      if(reset) begin
         totalCars <= '0;
         ones <= '0;
         tens <= '0;

      end else begin
         if(inc & (totalCars != CAPACITY)) begin
            totalCars <= totalCars + 1'b1;

            if(ones == 4'b1001) begin
               ones <= '0;
               tens <= tens + 4'b0001;
            end else begin
               ones <= ones + 4'b0001;
            end

         end else if (dec & totalCars != 0) begin
            totalCars <= totalCars - 1'b1;

            if(ones == 4'b0000) begin
               ones <= 4'b1001;
               tens <= tens - 4'b0001;
            end else begin
               ones <= ones - 4'b0001;
            end

         end else begin
            totalCars <= totalCars;
         end
      end
   end

   end

   assign clear = (totalCars == 5'b00000);
   assign full = (totalCars == CAPACITY);

endmodule
```

## 2.B) counter.sv (testbench)

```
// counter_testbench tests all expected and unexpected behavior of the count module. The primary purpose of this
// testbench is to see the behavior of the counter when it tries to decrement at 0 or increment at max capacity.
module counter_testbench();

   logic clk, reset, inc, dec;
   logic clear, full;
   logic [3:0] ones, tens;

   counter #(.CAPACITY(12)) dut (.clk, .reset, .inc, .dec, .clear, .full, .ones, .tens);

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

   initial begin

      reset <= 1;                                   @(posedge clk);
      reset <= 0; inc <= 1;            repeat(18) @(posedge clk);
                  inc <= 0; dec <= 1;  repeat(18) @(posedge clk);
                            dec <= 0;  repeat(2) @(posedge clk);

      $stop;
   end
endmodule
```

## 3.A) displayHex.sv (code)

```
// Brian Dallaire
// 04/14/2021
// EE 371
// Lab #1, Task 3

// displayCars has 1-bit input logic clear and full, and 4-bit input logic ones and tens. It outputs
// an array of 6, 7-bit HEX displays named hexArray. The purpose of this module is to display the outputs
// of the counter from the previous task. For the special cases where inputs clear or full are true, the
// HEX displays display unique letters. For the case for neither clear or full are true, HEX1 and HEX0 will
// display the total number of cars in the parking lot

module displayCars (clear, full, ones, tens, hexArray);
   input logic clear, full;
   input logic [3:0] ones, tens;
   output logic [5:0][6:0] hexArray;

   parameter BLANK = 7'b1111111;
   parameter C = 7'b1000110;
   parameter L = 7'b1000111;
   parameter E = 7'b0000110;
   parameter A = 7'b0001000;
   parameter R = 7'b0101111;
   parameter F = 7'b0001110;
   parameter U = 7'b1000001;

   // this always_comb block is designated for the displays HEX5-2. When clear or full are true, HEX5-2
   // will show letters that spell out CLEA or FULL. Otherwise, they will remain blank.
   always_comb begin

      if(clear)
         hexArray[5:2] = {C, L, E, A};
      else if (full)
         hexArray[5:2] = {F, U, L, L};
      else
         hexArray[5:2] = {BLANK, BLANK, BLANK, BLANK};
   end
```

```
   // this always_comb block is designated for the displays HEX1-HEX0. When clear is true, HEX1 has to display
   // the letter 'R' and HEX0 has to display the number zero. When clear is not true, HEX1 and HEX0 have to display
   // the total number of cars. hexArray[1] corresponds to HEX1 and represents the tens place, and hexArray[0] represents
   // HEX0 and represents the ones place.
   always_comb begin
      if(clear) begin
         hexArray[1] = R;
         hexArray[0] = 7'b1000000;
      end else begin
         case (tens)
         4'b0000: hexArray[1] = 7'b1000000; // 0
         4'b0001: hexArray[1] = 7'b1111001; // 1
         4'b0010: hexArray[1] = 7'b0100100; // 2
         4'b0011: hexArray[1] = 7'b0110000; // 3
         4'b0100: hexArray[1] = 7'b0011001; // 4
         4'b0101: hexArray[1] = 7'b0010010; // 5
         4'b0110: hexArray[1] = 7'b0000010; // 6
         4'b0111: hexArray[1] = 7'b1111000; // 7
         4'b1000: hexArray[1] = 7'b0000000; // 8
         4'b1001: hexArray[1] = 7'b0010000; // 9
         default: hexArray[1] = 7'bx;
         endcase
         case (ones)
         4'b0000: hexArray[0] = 7'b1000000; // 0
         4'b0001: hexArray[0] = 7'b1111001; // 1
         4'b0010: hexArray[0] = 7'b0100100; // 2
         4'b0011: hexArray[0] = 7'b0110000; // 3
         4'b0100: hexArray[0] = 7'b0011001; // 4
         4'b0101: hexArray[0] = 7'b0010010; // 5
         4'b0110: hexArray[0] = 7'b0000010; // 6
         4'b0111: hexArray[0] = 7'b1111000; // 7
         4'b1000: hexArray[0] = 7'b0000000; // 8
         4'b1001: hexArray[0] = 7'b0010000; // 9
         default: hexArray[0] = 7'bx;
         endcase
      end
   end

endmodule
```

## 3.B) displayHex.sv (testbench)

```
// displayCars_testbench tests irregular and regular cases of this module. For this testbench,
// I saw the effects of what happens when the count goes from zero to over 10. I made the max
// to be 11, so when the count was 11 and full became true, I saw the behaviors of the HEX displays.
// Then I counted back down to 0 and saw the behavior of the HEX displays when it reached zero and
// clear was true.
module displayCars_testbench();

   logic clear, full;
   logic [3:0] ones, tens;
   logic [5:0][6:0] hexArray;

   displayCars dut (.clear, .full, .ones, .tens, .hexArray);

   initial begin

      clear <= 1; ones <= '0; tens <= '0; #10;
      clear <= 0; ones <= 4'b0001;        #10;
                  ones <= 4'b0010;        #10;
                  ones <= 4'b0011;        #10;
                  ones <= 4'b0100;        #10;
                  ones <= 4'b0101;        #10;
                  ones <= 4'b0110;        #10;
                  ones <= 4'b0111;        #10;
                  ones <= 4'b1000;        #10;
                  ones <= 4'b1001;        #10;
                  ones <= 4'b0000; tens <= 4'b0001;          #10;
      full <= 1;  ones <= 4'b0001;        #10;
      full <= 0;  ones <= 4'b0000;        #10;
                  ones <= 4'b1001; tens <= 4'b0000;          #10;
                  ones <= 4'b1000;        #10;
                  ones <= 4'b0111;        #10;
                  ones <= 4'b0110;        #10;
                  ones <= 4'b0101;        #10;
                  ones <= 4'b0100;        #10;
                  ones <= 4'b0011;        #10;
                  ones <= 4'b0010;        #10;
                  ones <= 4'b0001;        #10;
      clear <= 1; ones <= 4'b0000;        #10;

      $stop;
   end
endmodule
```

## 4.A) DE1_SoC.sv (code)

```
// Brian Dallaire
// 04/14/2021
// EE 371
// Lab #1, Task 4

// DE1_SoC takes a 1-bit input CLOCK_50, 34-bit inout logic GPIO_0, and returns 6 different 7-bit displays HEX5 - HEX0. DE1_SoC combines
// the modules created in the previous tasks, and the LED on the GPIO_0 board and the HEX displays visualize the parking lot.

module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, GPIO_0, CLOCK_50);

   input logic CLOCK_50;
   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   inout logic [33:0] GPIO_0;

   logic reset, sw_aRaw, sw_bRaw;
   assign reset = GPIO_0[5];
   assign sw_aRaw = GPIO_0[7];
   assign sw_bRaw = GPIO_0[9];
   assign GPIO_0[26] = sw_aRaw;
   assign GPIO_0[27] = sw_bRaw;

   logic increment, decrement, clearLot, fullLot;
   logic sw_a, sw_b;
   logic [3:0] onePlace, tenPlace;

   // series_dffs take a 1-bit CLOCK_50, 1-bit reset, 1-bit sw_aRAW as inputs and output a clean 1-bit sw_a signal.
   // GPIO_[7] is filtered through two DFFs in series to prevent any chance of metastability
   series_dffs sensor_a (.clk(CLOCK_50), .reset, .raw(sw_aRaw), .clean(sw_a));

   // series_dffs take a 1-bit CLOCK_50, 1-bit reset, 1-bit sw_aRAW as inputs and output a clean 1-bit sw_a signal.
   // GPIO_[9] is filtered through two DFFs in series to prevent any chance of metastability
   series_dffs sensor_b (.clk(CLOCK_50), .reset, .raw(sw_bRaw), .clean(sw_b));

   // lotFSM takes a 1-bit CLOCK_50, 1-bit reset, 1-bit sw_a, 1-bit sw_b signals as inputs and outputs 1-bit outputs
   // increment and decrement. The purpose of this module is to determine if the sensors represented by sw_a and sw_b
   // were triggered in the proper order to indicate if a car fully entered or exited the parking lot
   lotFSM pLot (.clk(CLOCK_50), .reset, .a(sw_a), .b(sw_b), .incr(increment), .decr(decrement));

   // counter takes 1-bit CLOCK_50, 1-bit reset, 1-bit increment, 1-bit decrement as inputs, and outputs 1-bit clearLot,
   // 1-bit fullLot, 4-bit onePlace, and 4-bit tenPlace. The purpose of this module is to increment or decrement the total number
   // of cars in the parking lot when the input bits increment or decrement are true. It will output the total number using a combination
   // of the 4-bit outputs onePlace and tenPlace
   counter #(.CAPACITY(25)) countCars(.clk(CLOCK_50), .reset, .inc(increment), .dec(decrement), .clear(clearLot), .full(fullLot), .ones(onePlace), .tens(tenPlace));

   // for testbench purposes
   // counter #(.CAPACITY(5)) countCars(.clk(CLOCK_50), .reset, .inc(increment), .dec(decrement), .clear(clearLot), .full(fullLot), .ones(onePlace), .tens(tenPlace));


   // displayCars takes a 1-bit clearLot, 1-bit fullLot, 4-bit onePlace, 4-bit tenPlace as inputs and outputs an array of 6,
   // 7-bit outputs called hexArray that correspond to the 6 HEX displays of the DE1_SoC board. The purpose of this module
   // is to display the total count of cars in the parking lot, and also display when the parking lot is empty of full.
   displayCars dispHEX (.clear(clearLot), .full(fullLot), .ones(onePlace), .tens(tenPlace), .hexArray({HEX5, HEX4, HEX3, HEX2, HEX1, HEX0}));

endmodule
```

## 4.B) DE1_SoC.sv (testbench)

```
// DE1_SoC_testbench tests all of the expected and unexpected behavior of the parking lot system.
// For this testbench, I showcased the situation where a pedestrian walks through the sensors,
// cars entering the parking lot enough times to reach maximum capacity, and then from there
// having cars exiting the parking lot enough times to reach an empty lot.

module DE1_SoC_testbench();

   logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic CLOCK_50;
   wire [33:0] GPIO_0;

   logic reset, A, B;

   DE1_SoC dut (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .GPIO_0, .CLOCK_50);

   assign GPIO_0[5] = reset;
   assign GPIO_0[7] = A;
   assign GPIO_0[9] = B;


   parameter CLOCK_PERIOD=100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   initial begin
   reset <= 1;                              @(posedge CLOCK_50);
   reset <= 0; A <= 0; B <= 0;              @(posedge CLOCK_50);


               A <= 1;                      @(posedge CLOCK_50);
               A <= 0;  B <= 1;             @(posedge CLOCK_50);
                        B <= 0;             @(posedge CLOCK_50);
   repeat(5) begin
               A <= 1;                      @(posedge CLOCK_50);
                        B <= 1;             @(posedge CLOCK_50);
               A <= 0;                      @(posedge CLOCK_50);
                        B <= 0;             @(posedge CLOCK_50);
   end

   repeat(5) begin
                        B <= 1;                     @(posedge CLOCK_50);
               A <= 1;                              @(posedge CLOCK_50);
                        B <= 0;                     @(posedge CLOCK_50);
               A <= 0;                              @(posedge CLOCK_50);
   end
                                                    @(posedge CLOCK_50);
                                                    @(posedge CLOCK_50);
   $stop;

   end
endmodule
```

## 4.C) series_dffs.sv (code)

```
// Brian Dallaire
// 04/14/2021
// EE 371
// Lab #1, Task 4

// series_dffs has 1-bit inputs clk, reset, and raw. It outputs a 1-bit output clean. The purpose of this module
// is that it represents two DFFs in series. When a signal goes through two DFFs, it is very difficult for
// it to reach metastability with other signals.

module series_dffs(clk, reset, raw, clean);

   input logic clk, reset, raw;
   output logic clean;
   logic n1;

   // always_ff block that shows the DFFs displacing the signal. The 1-bit input signal goes into the first DFF and
   // the output of the first DFF goes in as the input to the second DFF. The output of the second DFF is the 1-bit
   // output clean
   always_ff @(posedge clk) begin
      if(reset) begin
         n1    <= 0;
         clean <= 0;
      end else
         n1    <= raw;
         clean <= n1;

   end

endmodule
```