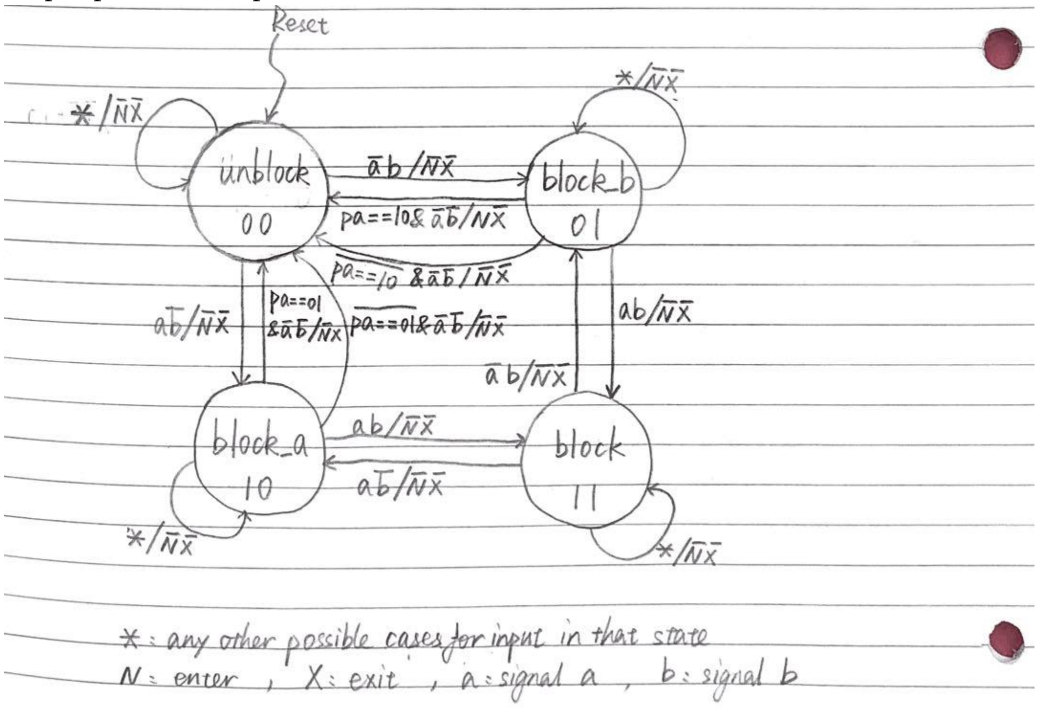
Cynthia Li
EE 371
April 12, 2021
Lab 1 Report

Procedure

The lab comprises of three tasks: 1) design a FSM to simulate a single entry parking lot gate on Quartus in System Verilog, and simulate its functionality on ModelSim; 2) design a counter that tells the parking lot's current occupancy on Quartus, and simulate its functionality on Modelsim; 3) Combining FSM in task#1 and the counter task#2, display the current occupancy status and number of parking lot onto 7-segment display HEX5-HEX0; 4) combine all the previous modules, wire inputs and outputs using GPIO_0.

Task#1
Design the FSM with two input signals, a (SW[0]) and b (SW[1]), and two output signals, enter and exit. An enter of car is defined as signal ab in the sequence 00 – 10 – 11 – 01 – 00, an exit of car is defined as signal ab in the sequence 00 – 01 – 11 - 10 – 00. When ab shows previous patterns, enter and exit will stay true for one clock cycle when a car enter or exits the lot, respectively; otherwise, they will stay false. Note that sequence like 00 – 01 – 00, 00 – 01 – 11 – 01 – 00, etc. are like the cases where the car changes direction halfway and thus will not be considered as an enter or an exit. Also, sequence such as 00-11 or 01-10 cannot happen, because in real cases, cars fly over the light and skip steps, thus, we consider this as a glitch, and it will output previous outputs.

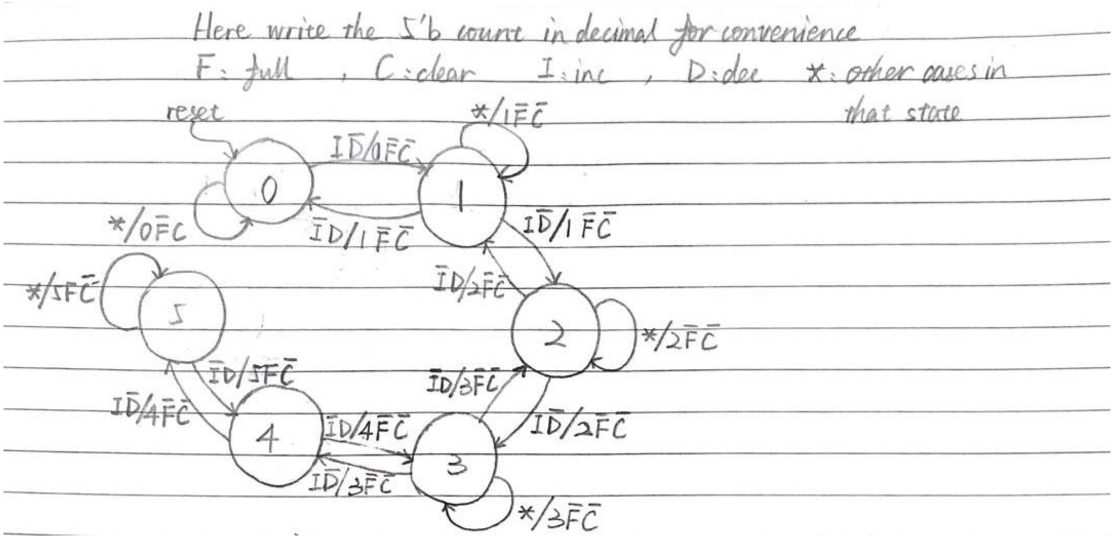


Reset
$*/\bar{N}\bar{X}$
unblock 00
$\bar{a}b/\bar{N}\bar{X}$
block_b 01
$*/\bar{N}\bar{X}$
$pa==10 \& \bar{a}\bar{b}/N\bar{X}$
$\overline{pa==10} \& \bar{a}\bar{b}/\bar{N}\bar{X}$
$a\bar{b}/\bar{N}\bar{X}$
$pa==01 \& \bar{a}\bar{b}/\bar{N}X$
$\overline{pa==01} \& \bar{a}\bar{b}/\bar{N}\bar{X}$
$ab/\bar{N}\bar{X}$
$\bar{a}b/\bar{N}\bar{X}$
block_a 10
$ab/\bar{N}\bar{X}$
$a\bar{b}/\bar{N}\bar{X}$
block 11
$*/\bar{N}\bar{X}$
$*/\bar{N}\bar{X}$
$*$: any other possible cases for input in that state
N: enter , X: exit , a: signal a , b: signal b

Fig.1 state diagram for gate

Task#2

Designed a counter is parameterized with default MAX = 5, with two input signals, inc and dec, which increments and decrements count when asserted, and outputs the occupancy status to full and clear. The count should not exceed MAX and no less than 0, it will stay the same if inc or dec tries to make count cross boundary. Note that default parameter MAX is 5, used for simulation in ModelSim and on FPGA.



Here write the 5'b count in decimal for convenience
F: full , C: clear I: inc , D: dec *: other cases in that state
reset

$$\begin{tikzcd}
5 \arrow[loop left, "*/5F\bar{C}"] \arrow[rd, "I\bar{D}/5F\bar{C}"] & & 0 \arrow[loop left, "*/0\bar{F}C"] \arrow[r, bend left, "I\bar{D}/0\bar{F}\bar{C}"] & 1 \arrow[loop above, "*/1\bar{F}\bar{C}"] \arrow[l, bend left, "\bar{I}D/1\bar{F}\bar{C}"] \arrow[rd, "I\bar{D}/1\bar{F}\bar{C}"] & \\
& 4 \arrow[lu, "\bar{I}D/4\bar{F}\bar{C}"] \arrow[rr, bend left, "I\bar{D}/4\bar{F}\bar{C}"] & & 3 \arrow[ll, bend left, "\bar{I}D/3\bar{F}\bar{C}"] \arrow[r, "\bar{I}D/3\bar{F}\bar{C}"] \arrow[loop below, "*/3\bar{F}\bar{C}"] & 2 \arrow[lu, "\bar{I}D/2\bar{F}\bar{C}"] \arrow[l, "I\bar{D}/2\bar{F}\bar{C}"] \arrow[loop right, "*/2\bar{F}\bar{C}"]
\end{tikzcd}$$

Fig.2 state diagram for counter (MAX=5)

Task #3

Design a display of current parking lot occupancy with a 5-bit input count (since we want the maximum capacity to be 25 in use), occupancy status full and clear, and will outputs 7-bit value in 7-segment form to HEX5 to HEX0.

Task#4

Interconnect the FSM, the counter, the display and GPIO_0. Use GPIO_0 to wire a switch as input reset signal, two switches for input signal a, b, and wire a, b to outputs which are LED lights.

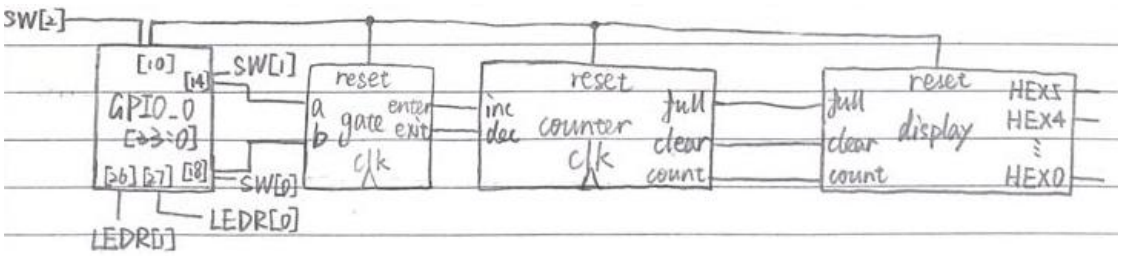


Fig.3 block diagram of top module

The system is designed based on the functionality of each small submodule. The design is basically as described in the specification. Since this is a real world single entry parking lot simulation, though some details are not specified (such as stop incrementing when reaching max capacity, etc.), they are defined based experience with parking lot (the count for cars in the lot cannot exceed max capacity, since there is no room for new car to enter).

Results

Analysis:

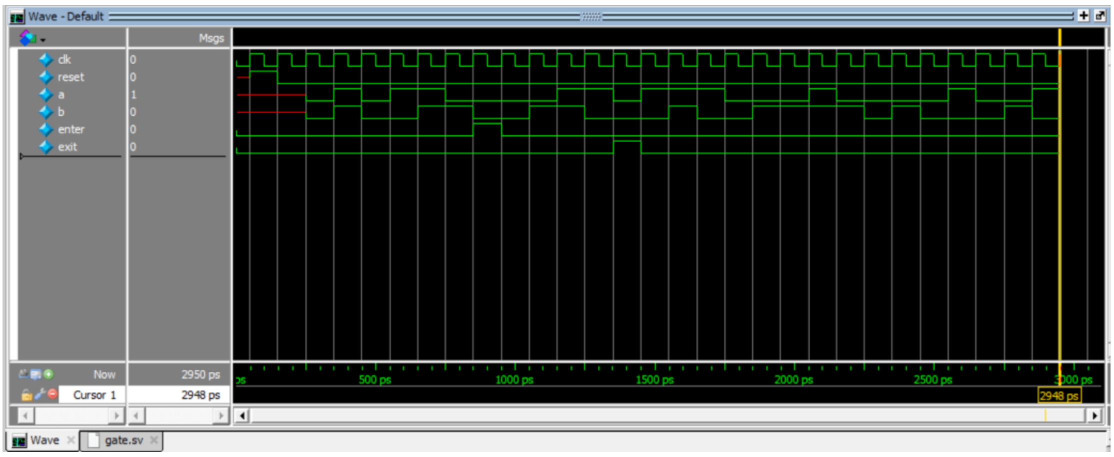


Fig.4 Simulation for gate_testbench

As shown in Appendix. 1.F, 1.G, it tests cases in sequence: standard entering, standard exiting, change direction halfway of entering and exiting, impossible change in reality (10 to 01, the car cannot jump a status).

Expect to see enter to assert true for one cycle after ab sequence “00 – 10 – 11 – 01 - 00” and exit assert true 4 cycle later, other time, both enter and exit are false.

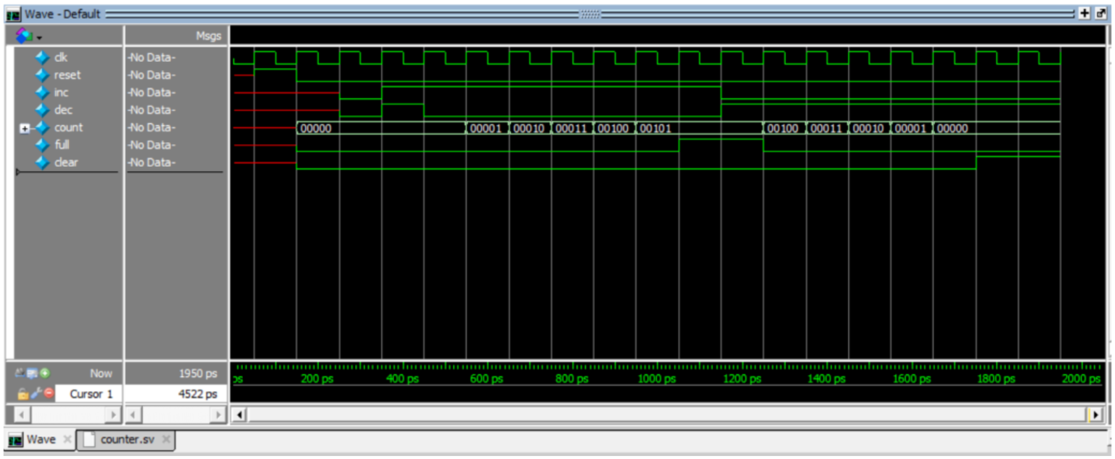


Fig.5 Simulation for counter_testbench

As shown by Appendix 2.C, 2.D, it tests default count, whether increment works, whether count exceeds MAX (5), whether decrement works, whether count will stop decreasing when reaching 0. Also test whether full and clear works correctly.

Expect to see default count is set to 0, then count increments until reaching MAX (5) and stay the same afterwards, full assert true; then, as decrement instruction is given, count decrement to zero and stay zero afterwards, clear assert true.

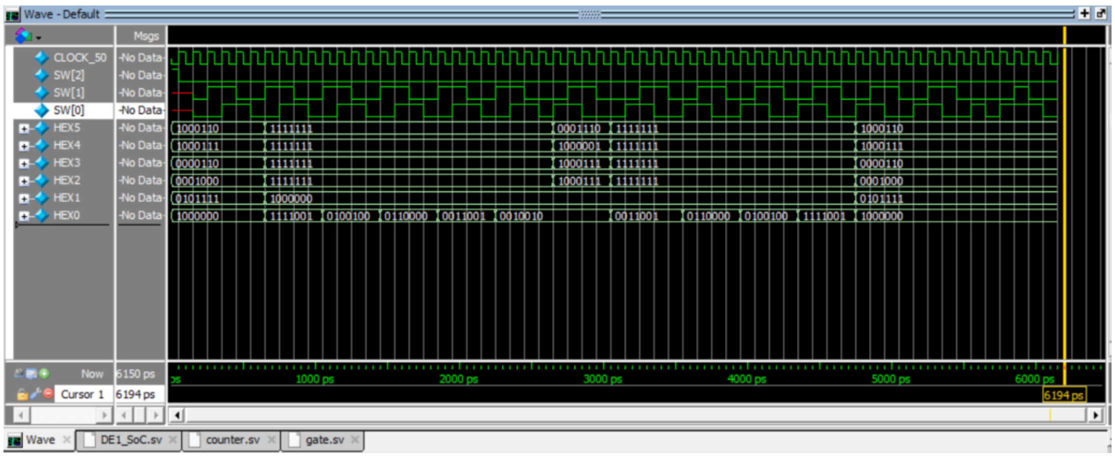


Fig.6 Simulation for DE1_SoC_testbench

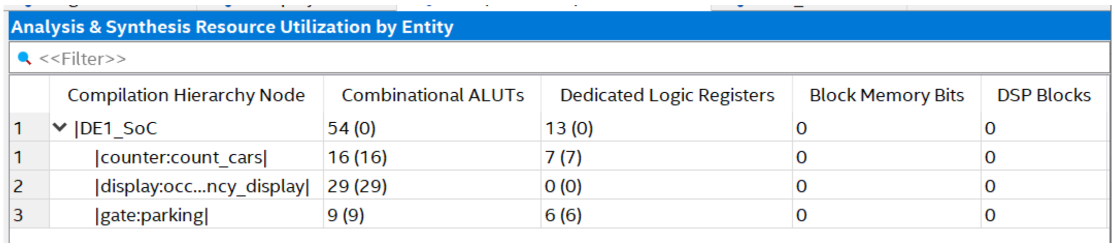
As shown by Appendix 4.B, 4.C, 4.D, 4.E, it tests the default HEX display, the incrementing display, the display after reaching maximum, the decrementing display, the display after reaching zero, and the display for changing direction halfway.

Expect to see HEX display changes from HEX5-HEX0 as “CLEAr0” to HEX5-HEX2 off and HEX1, HEX0 incrementing to “05”, when reaching “05,” HEX5-HEX2 display “FULL.” Then HEX1, HEX0 decrement to one, when reaching zero, HEX5-HEX0 display “CLEAr0.”

Conclusion:

Based on the simulation, we see that this design can successfully simulate a single entry parking lot as specified in the instruction, the only differences are: 1) in the simulation we used maximum capacity of 5, if we want to set it to 25, we need to change the parameter when calling the top module; 2) for the display, when count is zero, it should display “CLEAR,” but after demo, it seems that for 7-segment display, capital A and R looks the same, so I changed the display to “CLEAr.”

Resource Utilization:



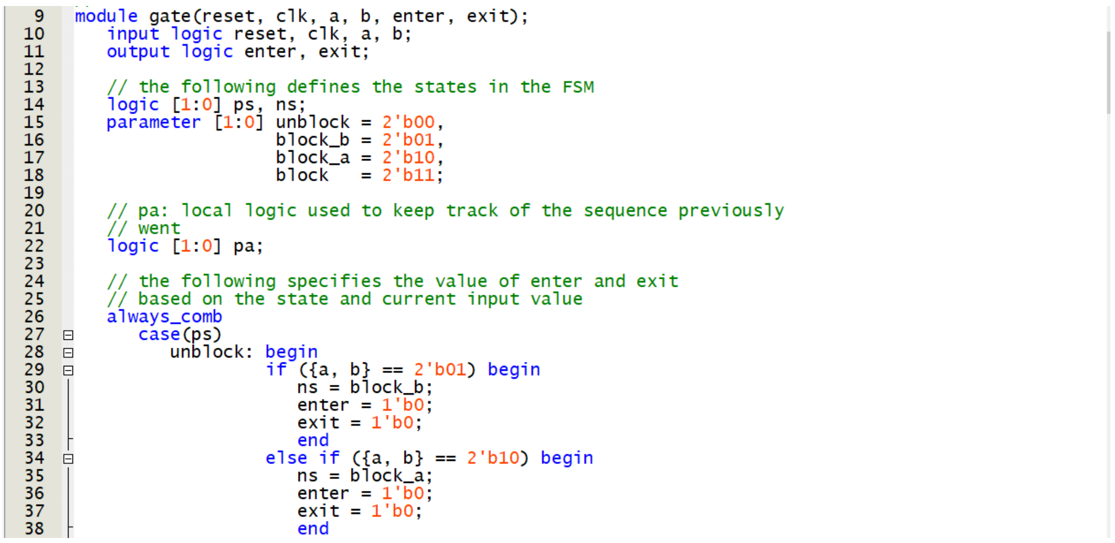
Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks
1	|DE1_SoC	54 (0)	13 (0)	0	0
1	|counter:count_cars|	16 (16)	7 (7)	0	0
2	|display:occ...ncy_display|	29 (29)	0 (0)	0	0
3	|gate:parking|	9 (9)	6 (6)	0	0

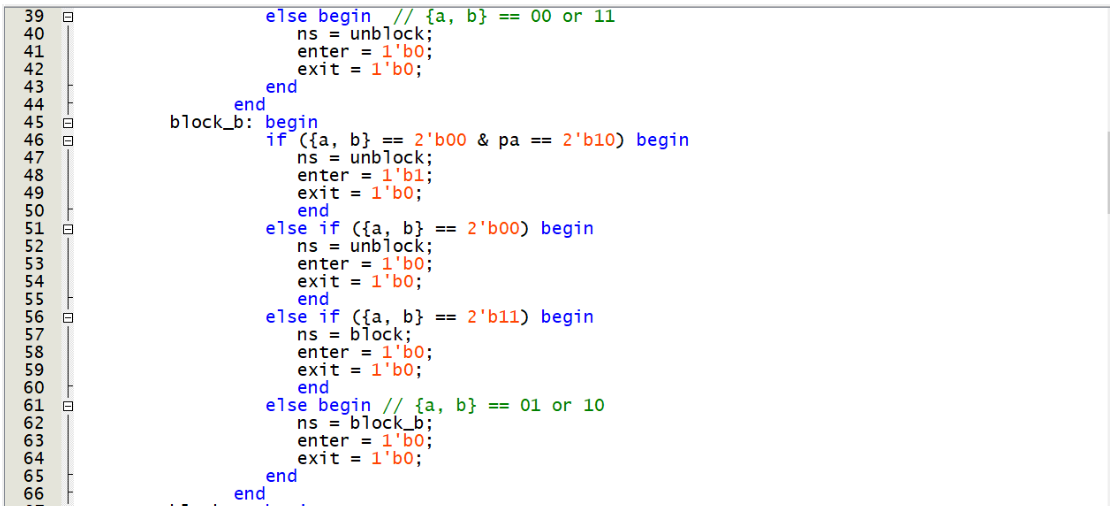
Appendix

1.A



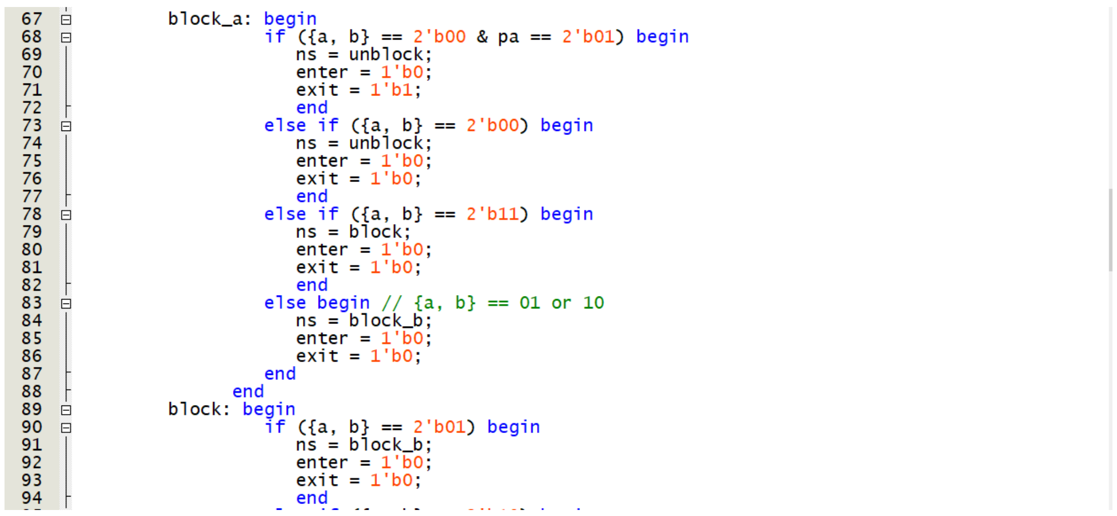
```
module gate(reset, clk, a, b, enter, exit);
   input logic reset, clk, a, b;
   output logic enter, exit;

   // the following defines the states in the FSM
   logic [1:0] ps, ns;
   parameter [1:0] unblock = 2'b00,
                   block_b = 2'b01,
                   block_a = 2'b10,
                   block   = 2'b11;

   // pa: local logic used to keep track of the sequence previously
   // went
   logic [1:0] pa;

   // the following specifies the value of enter and exit
   // based on the state and current input value
   always_comb
      case(ps)
         unblock: begin
                  if ({a, b} == 2'b01) begin
                     ns = block_b;
                     enter = 1'b0;
                     exit = 1'b0;
                     end
                  else if ({a, b} == 2'b10) begin
                     ns = block_a;
                     enter = 1'b0;
                     exit = 1'b0;
                     end
```

1.B



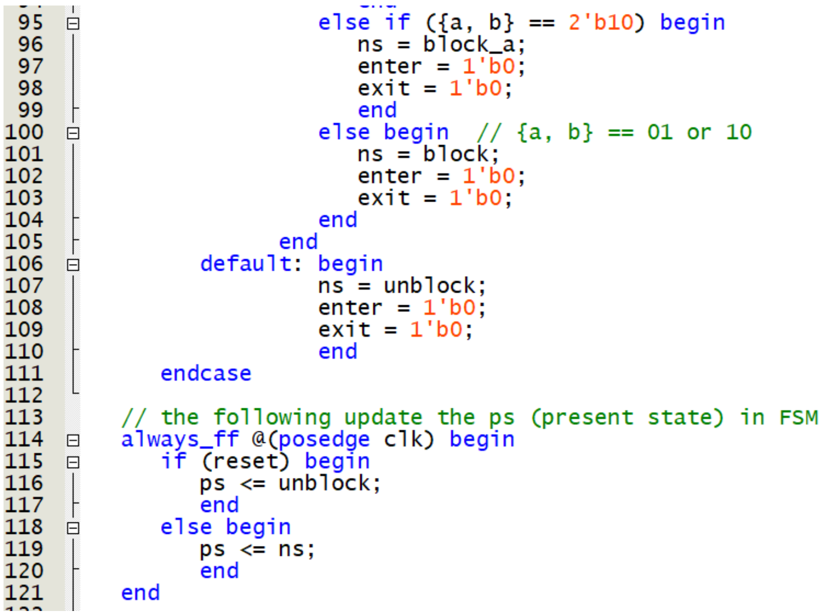
```
                  else begin  // {a, b} == 00 or 11
                     ns = unblock;
                     enter = 1'b0;
                     exit = 1'b0;
                  end
               end
         block_b: begin
                  if ({a, b} == 2'b00 & pa == 2'b10) begin
                     ns = unblock;
                     enter = 1'b1;
                     exit = 1'b0;
                     end
                  else if ({a, b} == 2'b00) begin
                     ns = unblock;
                     enter = 1'b0;
                     exit = 1'b0;
                     end
                  else if ({a, b} == 2'b11) begin
                     ns = block;
                     enter = 1'b0;
                     exit = 1'b0;
                     end
                  else begin // {a, b} == 01 or 10
                     ns = block_b;
                     enter = 1'b0;
                     exit = 1'b0;
                  end
               end
```

1.   C



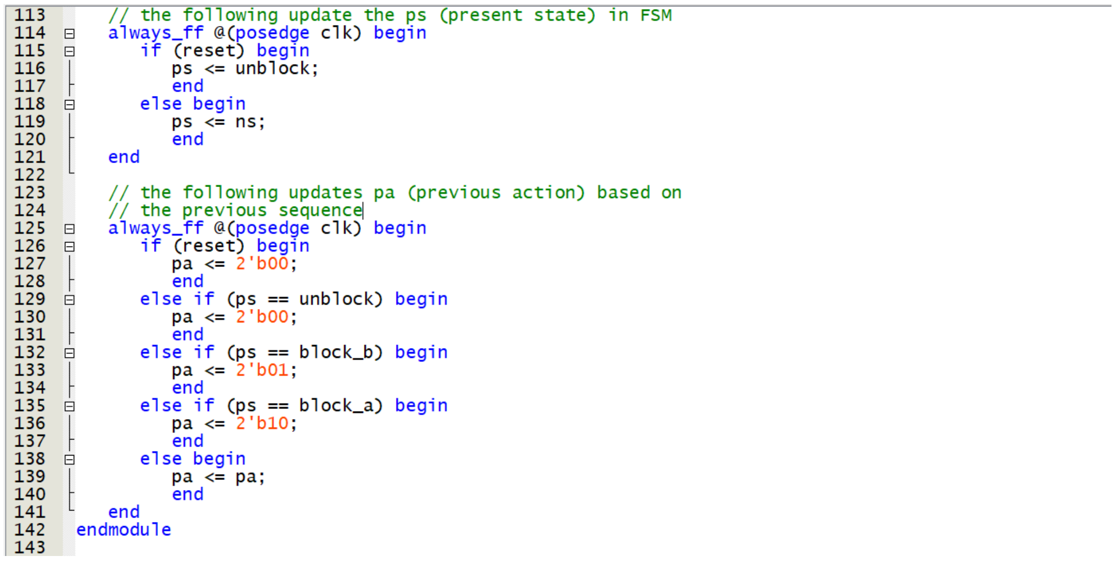
```
         block_a: begin
                  if ({a, b} == 2'b00 & pa == 2'b01) begin
                     ns = unblock;
                     enter = 1'b0;
                     exit = 1'b1;
                     end
                  else if ({a, b} == 2'b00) begin
                     ns = unblock;
                     enter = 1'b0;
                     exit = 1'b0;
                     end
                  else if ({a, b} == 2'b11) begin
                     ns = block;
                     enter = 1'b0;
                     exit = 1'b0;
                     end
                  else begin // {a, b} == 01 or 10
                     ns = block_b;
                     enter = 1'b0;
                     exit = 1'b0;
                  end
               end
         block: begin
                  if ({a, b} == 2'b01) begin
                     ns = block_b;
                     enter = 1'b0;
                     exit = 1'b0;
                     end
```

1.D



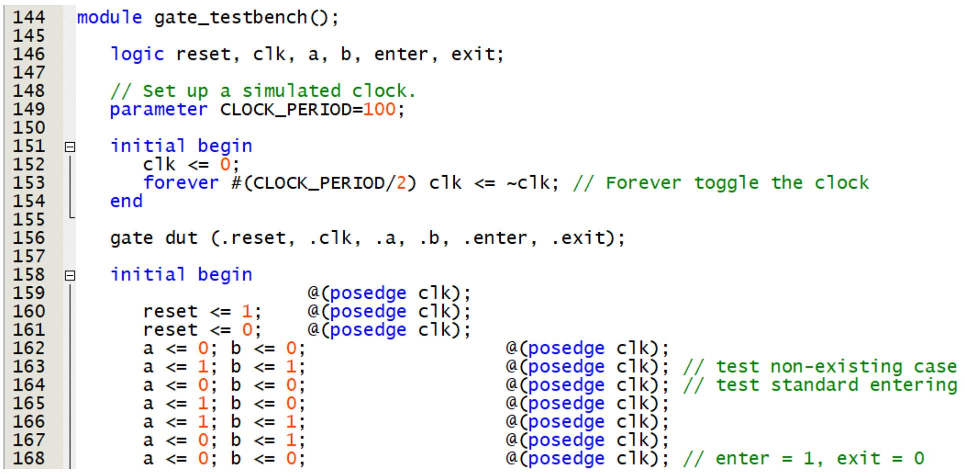
```
                  else if ({a, b} == 2'b10) begin
                     ns = block_a;
                     enter = 1'b0;
                     exit = 1'b0;
                     end
                  else begin  // {a, b} == 01 or 10
                     ns = block;
                     enter = 1'b0;
                     exit = 1'b0;
                  end
               end
         default: begin
                  ns = unblock;
                  enter = 1'b0;
                  exit = 1'b0;
                  end
      endcase

   // the following update the ps (present state) in FSM
   always_ff @(posedge clk) begin
      if (reset) begin
         ps <= unblock;
         end
      else begin
         ps <= ns;
         end
   end
```

1.E



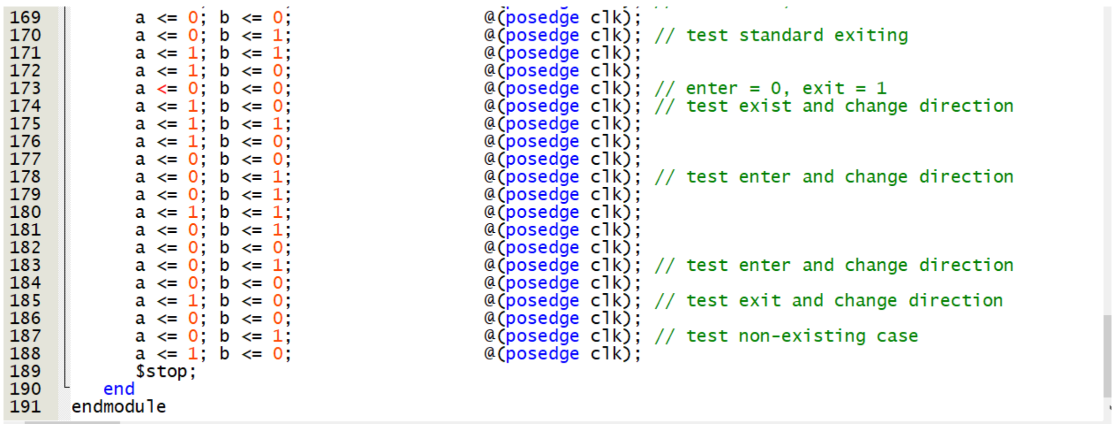
```
   // the following update the ps (present state) in FSM
   always_ff @(posedge clk) begin
      if (reset) begin
         ps <= unblock;
         end
      else begin
         ps <= ns;
         end
   end

   // the following updates pa (previous action) based on
   // the previous sequence
   always_ff @(posedge clk) begin
      if (reset) begin
         pa <= 2'b00;
         end
      else if (ps == unblock) begin
         pa <= 2'b00;
         end
      else if (ps == block_b) begin
         pa <= 2'b01;
         end
      else if (ps == block_a) begin
         pa <= 2'b10;
         end
      else begin
         pa <= pa;
         end
   end
endmodule
```

1.F



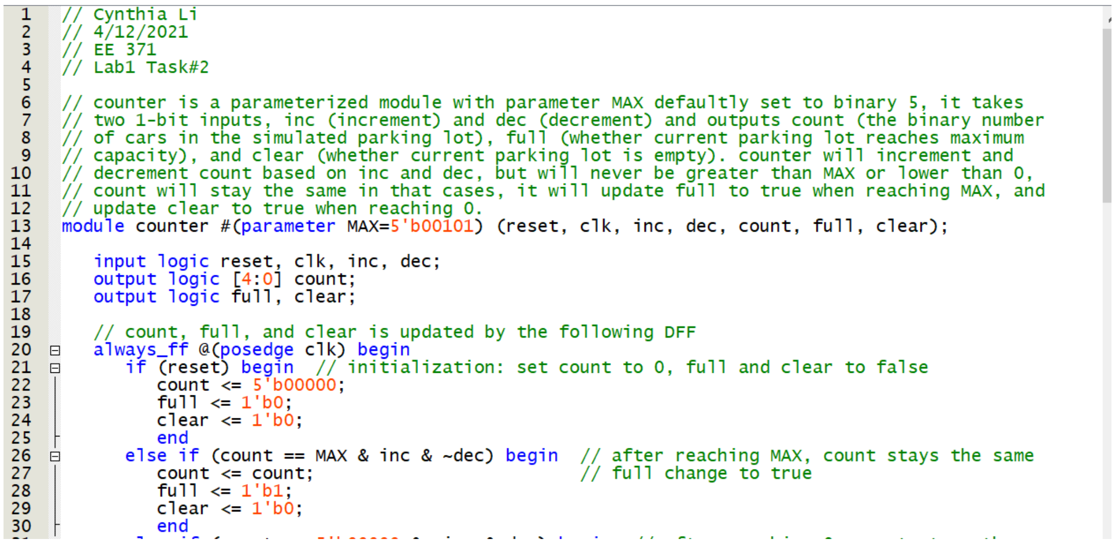
```
module gate_testbench();

   logic reset, clk, a, b, enter, exit;

   // Set up a simulated clock.
   parameter CLOCK_PERIOD=100;

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

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

   initial begin
                     @(posedge clk);
      reset <= 1;    @(posedge clk);
      reset <= 0;    @(posedge clk);
      a <= 0; b <= 0;                  @(posedge clk);
      a <= 1; b <= 1;                  @(posedge clk); // test non-existing case
      a <= 0; b <= 0;                  @(posedge clk); // test standard entering
      a <= 1; b <= 0;                  @(posedge clk);
      a <= 1; b <= 1;                  @(posedge clk);
      a <= 0; b <= 1;                  @(posedge clk);
      a <= 0; b <= 0;                  @(posedge clk); // enter = 1, exit = 0
```

1.G



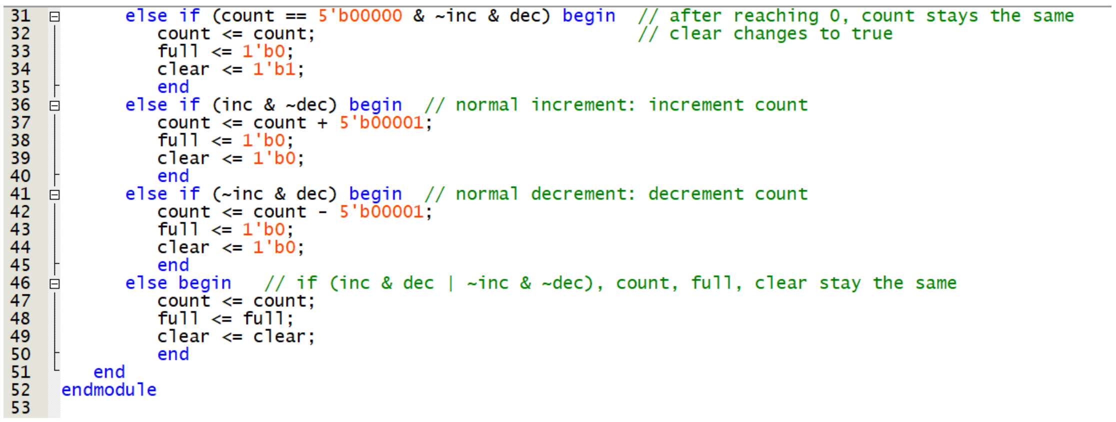
```
        a <= 0; b <= 0;                     @(posedge clk);
        a <= 0; b <= 1;                     @(posedge clk); // test standard exiting
        a <= 1; b <= 1;                     @(posedge clk);
        a <= 1; b <= 0;                     @(posedge clk);
        a <= 0; b <= 0;                     @(posedge clk); // enter = 0, exit = 1
        a <= 1; b <= 0;                     @(posedge clk); // test exist and change direction
        a <= 1; b <= 1;                     @(posedge clk);
        a <= 1; b <= 0;                     @(posedge clk);
        a <= 0; b <= 0;                     @(posedge clk);
        a <= 0; b <= 1;                     @(posedge clk); // test enter and change direction
        a <= 0; b <= 1;                     @(posedge clk);
        a <= 1; b <= 1;                     @(posedge clk);
        a <= 0; b <= 1;                     @(posedge clk);
        a <= 0; b <= 0;                     @(posedge clk);
        a <= 0; b <= 1;                     @(posedge clk); // test enter and change direction
        a <= 0; b <= 0;                     @(posedge clk);
        a <= 1; b <= 0;                     @(posedge clk); // test exit and change direction
        a <= 0; b <= 0;                     @(posedge clk);
        a <= 0; b <= 1;                     @(posedge clk); // test non-existing case
        a <= 1; b <= 0;                     @(posedge clk);
        $stop;
    end
endmodule
```

2.A



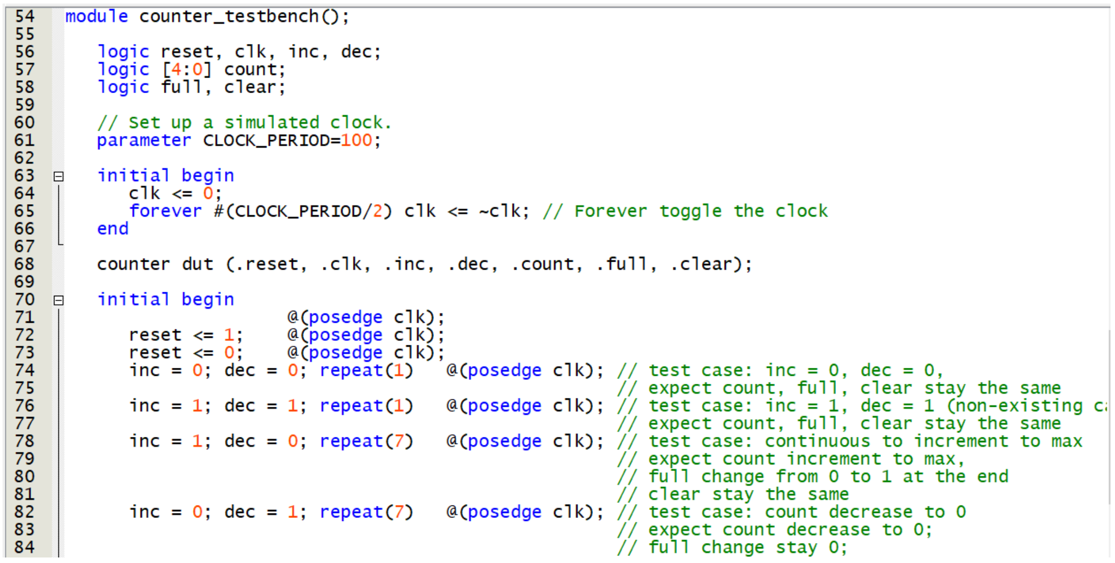
```
// Cynthia Li
// 4/12/2021
// EE 371
// Lab1 Task#2

// counter is a parameterized module with parameter MAX defaultly set to binary 5, it takes
// two 1-bit inputs, inc (increment) and dec (decrement) and outputs count (the binary number
// of cars in the simulated parking lot), full (whether current parking lot reaches maximum
// capacity), and clear (whether current parking lot is empty). counter will increment and
// decrement count based on inc and dec, but will never be greater than MAX or lower than 0,
// count will stay the same in that cases, it will update full to true when reaching MAX, and
// update clear to true when reaching 0.
module counter #(parameter MAX=5'b00101) (reset, clk, inc, dec, count, full, clear);

   input logic reset, clk, inc, dec;
   output logic [4:0] count;
   output logic full, clear;

   // count, full, and clear is updated by the following DFF
   always_ff @(posedge clk) begin
      if (reset) begin  // initialization: set count to 0, full and clear to false
         count <= 5'b00000;
         full <= 1'b0;
         clear <= 1'b0;
         end
      else if (count == MAX & inc & ~dec) begin  // after reaching MAX, count stays the same
         count <= count;                          // full change to true
         full <= 1'b1;
         clear <= 1'b0;
         end
```

2.B



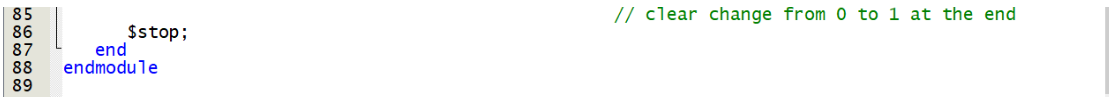
```
      else if (count == 5'b00000 & ~inc & dec) begin  // after reaching 0, count stays the same
         count <= count;                               // clear changes to true
         full <= 1'b0;
         clear <= 1'b1;
         end
      else if (inc & ~dec) begin  // normal increment: increment count
         count <= count + 5'b00001;
         full <= 1'b0;
         clear <= 1'b0;
         end
      else if (~inc & dec) begin  // normal decrement: decrement count
         count <= count - 5'b00001;
         full <= 1'b0;
         clear <= 1'b0;
         end
      else begin   // if (inc & dec | ~inc & ~dec), count, full, clear stay the same
         count <= count;
         full <= full;
         clear <= clear;
         end
   end
endmodule
```

2.C



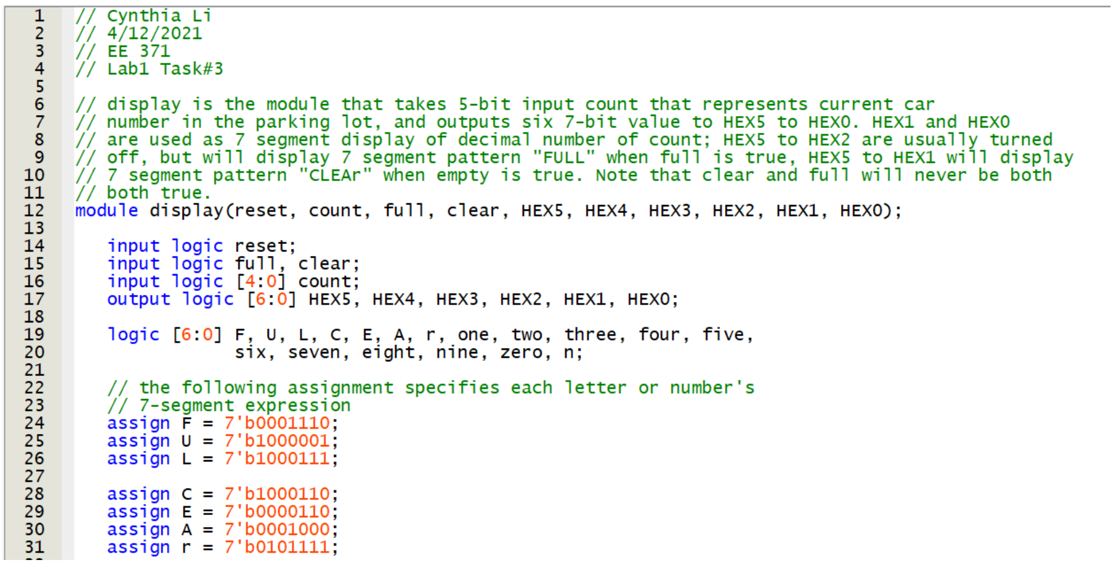
```
module counter_testbench();

   logic reset, clk, inc, dec;
   logic [4:0] count;
   logic full, clear;

   // Set up a simulated clock.
   parameter CLOCK_PERIOD=100;

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

   counter dut (.reset, .clk, .inc, .dec, .count, .full, .clear);

   initial begin
                     @(posedge clk);
      reset <= 1;    @(posedge clk);
      reset <= 0;    @(posedge clk);
      inc = 0; dec = 0; repeat(1)   @(posedge clk); // test case: inc = 0, dec = 0,
                                                    // expect count, full, clear stay the same
      inc = 1; dec = 1; repeat(1)   @(posedge clk); // test case: inc = 1, dec = 1 (non-existing c
                                                    // expect count, full, clear stay the same
      inc = 1; dec = 0; repeat(7)   @(posedge clk); // test case: continuous to increment to max
                                                    // expect count increment to max,
                                                    // full change from 0 to 1 at the end
                                                    // clear stay the same
      inc = 0; dec = 1; repeat(7)   @(posedge clk); // test case: count decrease to 0
                                                    // expect count decrease to 0;
                                                    // full change stay 0;
```

2.D



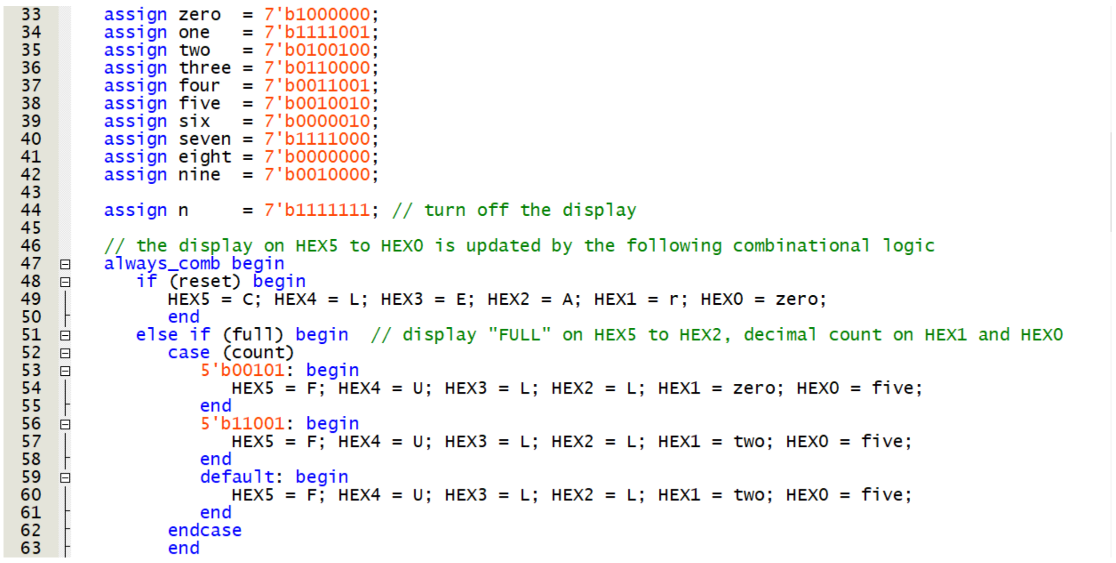
```
                                                    // clear change from 0 to 1 at the end
      $stop;
   end
endmodule
```

3.A



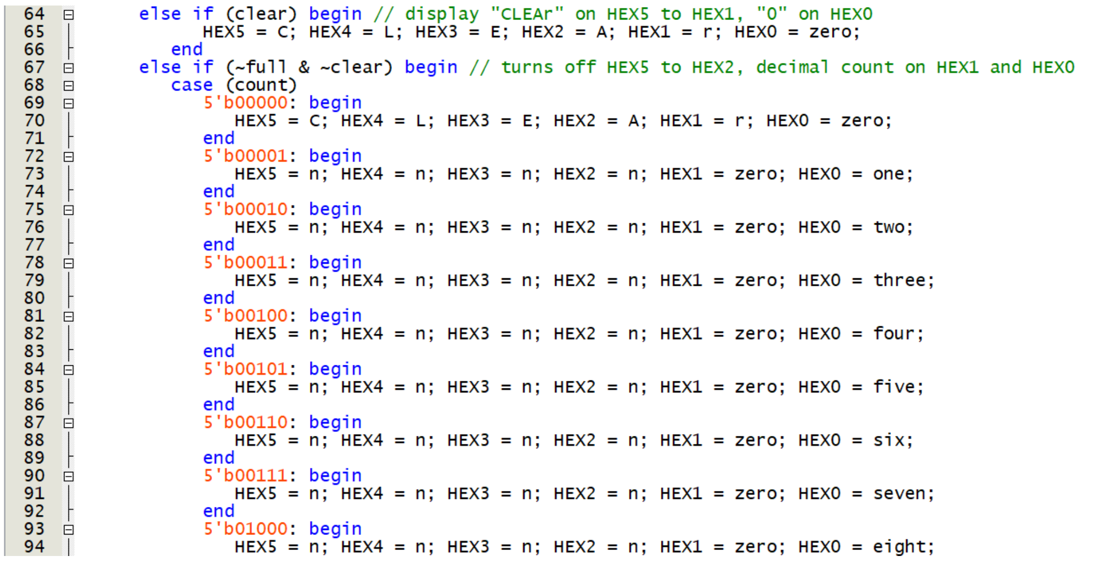
```
// Cynthia Li
// 4/12/2021
// EE 371
// Lab1 Task#3

// display is the module that takes 5-bit input count that represents current car
// number in the parking lot, and outputs six 7-bit value to HEX5 to HEX0. HEX1 and HEX0
// are used as 7 segment display of decimal number of count; HEX5 to HEX2 are usually turned
// off, but will display 7 segment pattern "FULL" when full is true, HEX5 to HEX1 will display
// 7 segment pattern "CLEAr" when empty is true. Note that clear and full will never be both
// both true.
module display(reset, count, full, clear, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);

   input logic reset;
   input logic full, clear;
   input logic [4:0] count;
   output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   logic [6:0] F, U, L, C, E, A, r, one, two, three, four, five,
               six, seven, eight, nine, zero, n;

   // the following assignment specifies each letter or number's
   // 7-segment expression
   assign F = 7'b0001110;
   assign U = 7'b1000001;
   assign L = 7'b1000111;

   assign C = 7'b1000110;
   assign E = 7'b0000110;
   assign A = 7'b0001000;
   assign r = 7'b0101111;
```

3.B



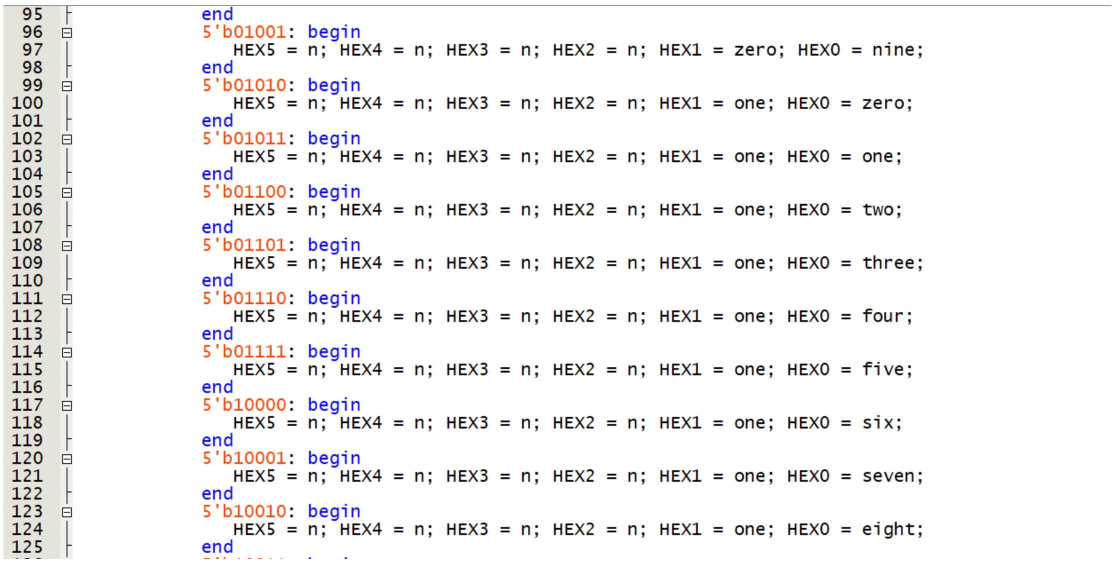
```verilog
assign zero  = 7'b1000000;
assign one   = 7'b1111001;
assign two   = 7'b0100100;
assign three = 7'b0110000;
assign four  = 7'b0011001;
assign five  = 7'b0010010;
assign six   = 7'b0000010;
assign seven = 7'b1111000;
assign eight = 7'b0000000;
assign nine  = 7'b0010000;

assign n     = 7'b1111111; // turn off the display

// the display on HEX5 to HEX0 is updated by the following combinational logic
always_comb begin
   if (reset) begin
      HEX5 = C; HEX4 = L; HEX3 = E; HEX2 = A; HEX1 = r; HEX0 = zero;
      end
   else if (full) begin  // display "FULL" on HEX5 to HEX2, decimal count on HEX1 and HEX0
      case (count)
         5'b00101: begin
            HEX5 = F; HEX4 = U; HEX3 = L; HEX2 = L; HEX1 = zero; HEX0 = five;
         end
         5'b11001: begin
            HEX5 = F; HEX4 = U; HEX3 = L; HEX2 = L; HEX1 = two; HEX0 = five;
         end
         default: begin
            HEX5 = F; HEX4 = U; HEX3 = L; HEX2 = L; HEX1 = two; HEX0 = five;
         end
      endcase
      end
```

3.C



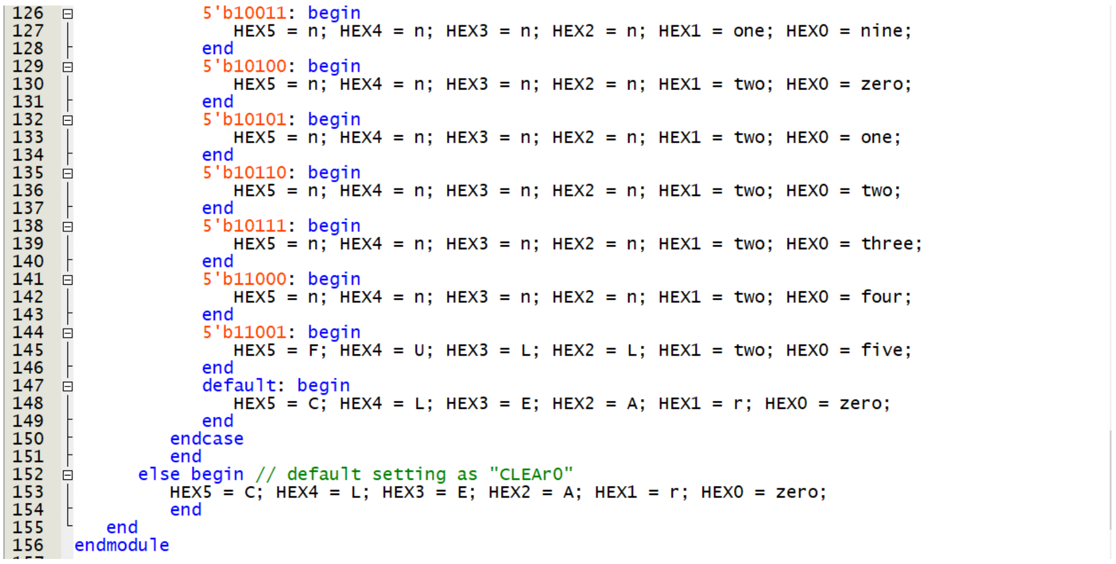
```verilog
   else if (clear) begin // display "CLEAr" on HEX5 to HEX1, "0" on HEX0
         HEX5 = C; HEX4 = L; HEX3 = E; HEX2 = A; HEX1 = r; HEX0 = zero;
      end
   else if (~full & ~clear) begin // turns off HEX5 to HEX2, decimal count on HEX1 and HEX0
      case (count)
         5'b00000: begin
            HEX5 = C; HEX4 = L; HEX3 = E; HEX2 = A; HEX1 = r; HEX0 = zero;
         end
         5'b00001: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = zero; HEX0 = one;
         end
         5'b00010: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = zero; HEX0 = two;
         end
         5'b00011: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = zero; HEX0 = three;
         end
         5'b00100: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = zero; HEX0 = four;
         end
         5'b00101: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = zero; HEX0 = five;
         end
         5'b00110: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = zero; HEX0 = six;
         end
         5'b00111: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = zero; HEX0 = seven;
         end
         5'b01000: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = zero; HEX0 = eight;
```

3.D



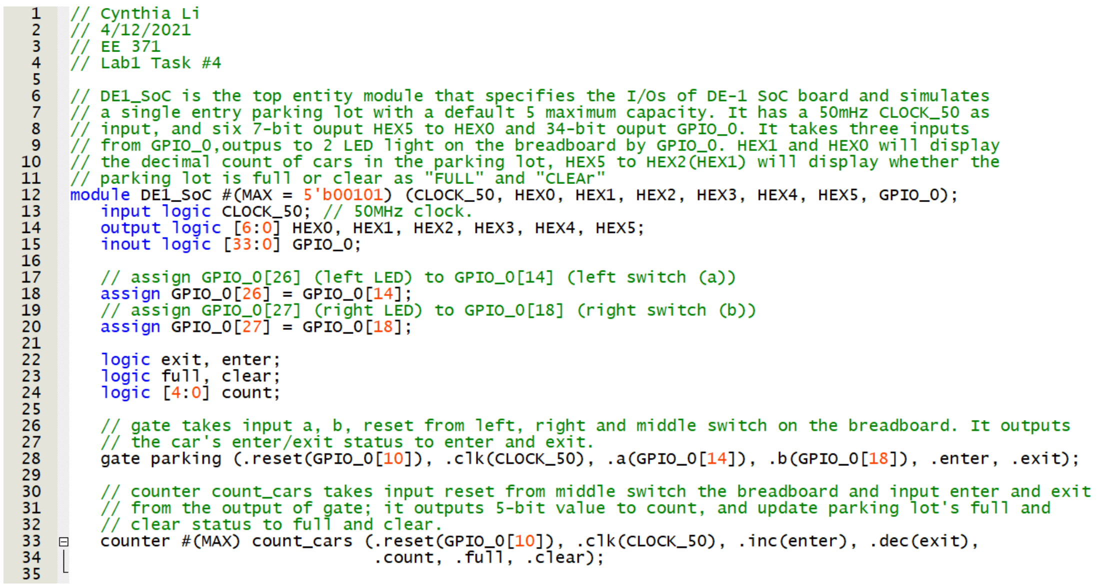
```verilog
         end
         5'b01001: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = zero; HEX0 = nine;
         end
         5'b01010: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = one; HEX0 = zero;
         end
         5'b01011: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = one; HEX0 = one;
         end
         5'b01100: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = one; HEX0 = two;
         end
         5'b01101: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = one; HEX0 = three;
         end
         5'b01110: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = one; HEX0 = four;
         end
         5'b01111: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = one; HEX0 = five;
         end
         5'b10000: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = one; HEX0 = six;
         end
         5'b10001: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = one; HEX0 = seven;
         end
         5'b10010: begin
            HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = one; HEX0 = eight;
         end
```

3.E



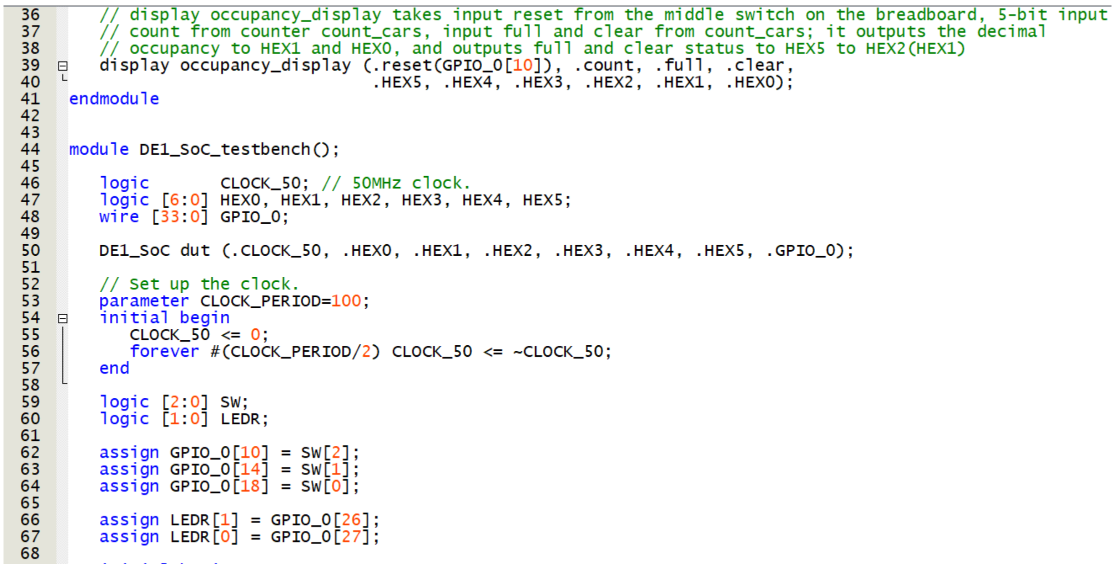
```
            5'b10011: begin
               HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = one; HEX0 = nine;
            end
            5'b10100: begin
               HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = two; HEX0 = zero;
            end
            5'b10101: begin
               HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = two; HEX0 = one;
            end
            5'b10110: begin
               HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = two; HEX0 = two;
            end
            5'b10111: begin
               HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = two; HEX0 = three;
            end
            5'b11000: begin
               HEX5 = n; HEX4 = n; HEX3 = n; HEX2 = n; HEX1 = two; HEX0 = four;
            end
            5'b11001: begin
               HEX5 = F; HEX4 = U; HEX3 = L; HEX2 = L; HEX1 = two; HEX0 = five;
            end
            default: begin
               HEX5 = C; HEX4 = L; HEX3 = E; HEX2 = A; HEX1 = r; HEX0 = zero;
            end
         endcase
         end
      else begin // default setting as "CLEAr0"
         HEX5 = C; HEX4 = L; HEX3 = E; HEX2 = A; HEX1 = r; HEX0 = zero;
         end
   end
endmodule
```

4.A



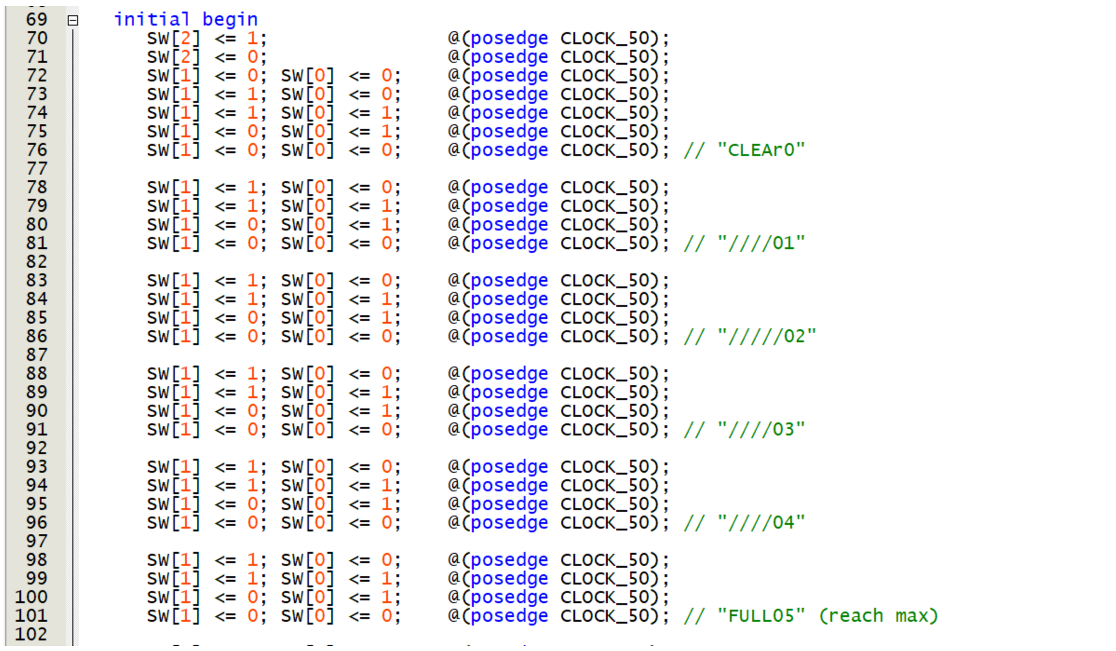
```
// Cynthia Li
// 4/12/2021
// EE 371
// Lab1 Task #4

// DE1_SoC is the top entity module that specifies the I/Os of DE-1 SoC board and simulates
// a single entry parking lot with a default 5 maximum capacity. It has a 50mHz CLOCK_50 as
// input, and six 7-bit ouput HEX5 to HEX0 and 34-bit ouput GPIO_0. It takes three inputs
// from GPIO_0,outpus to 2 LED light on the breadboard by GPIO_0. HEX1 and HEX0 will display
// the decimal count of cars in the parking lot, HEX5 to HEX2(HEX1) will display whether the
// parking lot is full or clear as "FULL" and "CLEAr"
module DE1_SoC #(MAX = 5'b00101) (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, GPIO_0);
   input logic CLOCK_50; // 50MHz clock.
   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   inout logic [33:0] GPIO_0;

   // assign GPIO_0[26] (left LED) to GPIO_0[14] (left switch (a))
   assign GPIO_0[26] = GPIO_0[14];
   // assign GPIO_0[27] (right LED) to GPIO_0[18] (right switch (b))
   assign GPIO_0[27] = GPIO_0[18];

   logic exit, enter;
   logic full, clear;
   logic [4:0] count;

   // gate takes input a, b, reset from left, right and middle switch on the breadboard. It outputs
   // the car's enter/exit status to enter and exit.
   gate parking (.reset(GPIO_0[10]), .clk(CLOCK_50), .a(GPIO_0[14]), .b(GPIO_0[18]), .enter, .exit);

   // counter count_cars takes input reset from middle switch the breadboard and input enter and exit
   // from the output of gate; it outputs 5-bit value to count, and update parking lot's full and
   // clear status to full and clear.
   counter #(MAX) count_cars (.reset(GPIO_0[10]), .clk(CLOCK_50), .inc(enter), .dec(exit),
                              .count, .full, .clear);

```

4.B



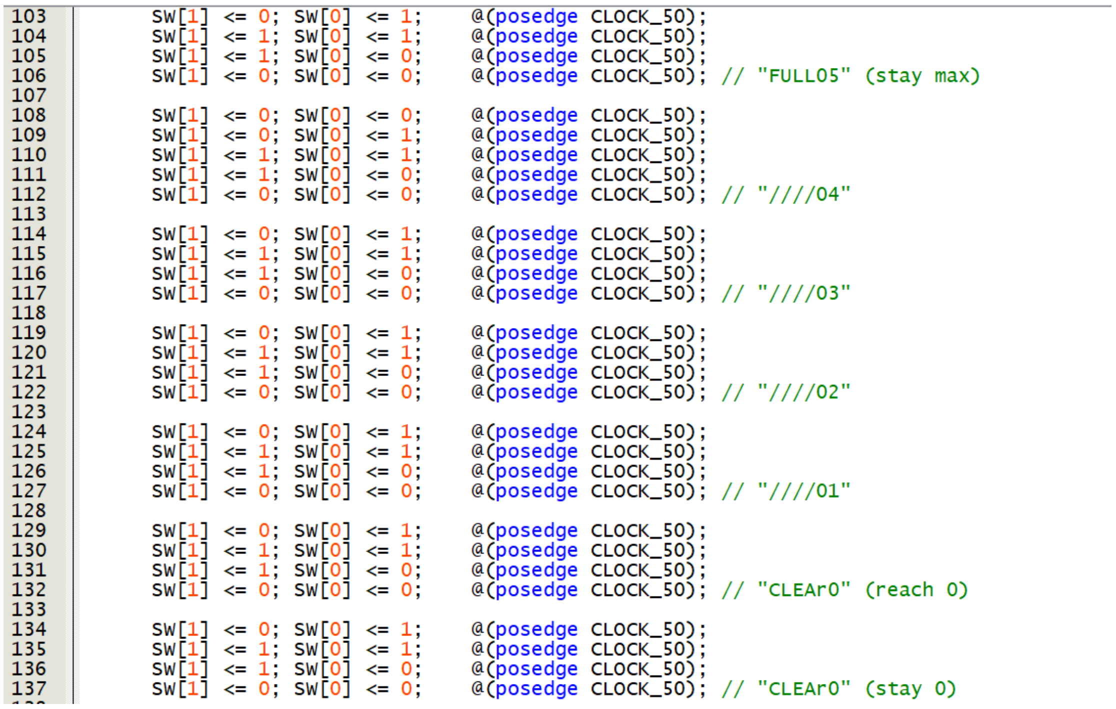
```
   // display occupancy_display takes input reset from the middle switch on the breadboard, 5-bit input
   // count from counter count_cars, input full and clear from count_cars; it outputs the decimal
   // occupancy to HEX1 and HEX0, and outputs full and clear status to HEX5 to HEX2(HEX1)
   display occupancy_display (.reset(GPIO_0[10]), .count, .full, .clear,
                              .HEX5, .HEX4, .HEX3, .HEX2, .HEX1, .HEX0);
endmodule


module DE1_SoC_testbench();

   logic       CLOCK_50; // 50MHz clock.
   logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   wire [33:0] GPIO_0;

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

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

   logic [2:0] SW;
   logic [1:0] LEDR;

   assign GPIO_0[10] = SW[2];
   assign GPIO_0[14] = SW[1];
   assign GPIO_0[18] = SW[0];

   assign LEDR[1] = GPIO_0[26];
   assign LEDR[0] = GPIO_0[27];

```

4.C



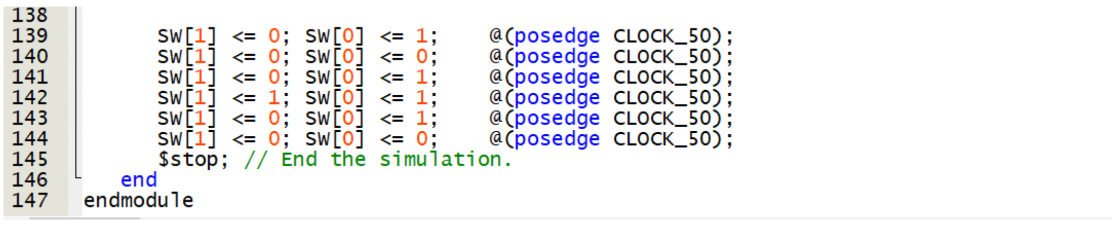
```
  initial begin
     SW[2] <= 1;                   @(posedge CLOCK_50);
     SW[2] <= 0;                   @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "CLEAr0"

     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "////01"

     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "/////02"

     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "////03"

     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "////04"

     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "FULL05" (reach max)

```

4.D



```
     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "FULL05" (stay max)

     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "////04"

     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "////03"

     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "////02"

     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "////01"

     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "CLEAr0" (reach 0)

     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50); // "CLEAr0" (stay 0)
```

4.E



```

     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 1; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 1;       @(posedge CLOCK_50);
     SW[1] <= 0; SW[0] <= 0;       @(posedge CLOCK_50);
     $stop; // End the simulation.
  end
endmodule
```