Jade Jwa

04/14/21

EE 371

Lab 1 Report

Procedure

This lab required us to create a parking lot system that uses two sensors to detect if a car is entering or exiting into a parking lot, and subsequently displaying the number of cars currently in the parking lot onto the FPGA board. There are two sensors, a and b, to help keep track of whether the car is entering or exiting; and these sensors are connected to the LEDs' so that the LEDs' light up whenever the car is passing through a certain sensor. The occupancy of the parking lot is 25 cars, and the system also shows different messages when the parking lot is empty, when the parking lot is full, but otherwise displays the number of cars in the parking lot. In order to implement this said system, I divided up the work for the parking lot into four modules: the DE1_SoC, the counter, the FSM, and the display.

Task 1: Parking Lot Occupancy Counter

I created four modules to create the counter: the DE1_SoC, the Counter, the FSM, and the Display.

The FSM module is used to determine what state the car is in. It takes in the signals of the a and b sensors and brings it together to determine whether or not the car is leaving, entering, or just wandering around the parking lot. It then subsequently returns if the number of cars should be incremented or decremented and returns those values into enter and exit outputs.

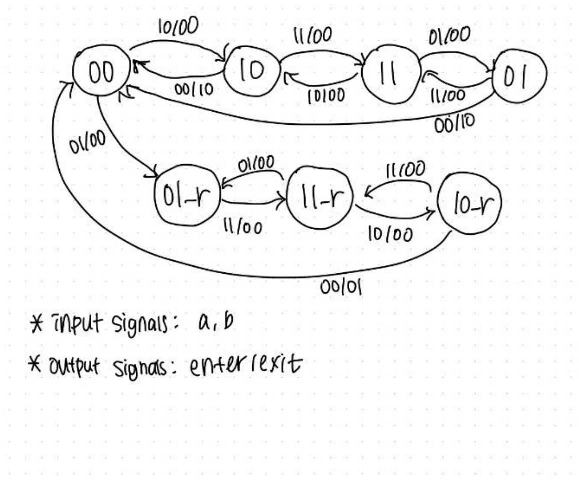


Figure 1. State Diagram of the FSM Module

Figure I shows the state diagram that I used to implement the FSM module. The FSM module requires 7 states; it requires a neutral state at which the car is neither entering nor exiting, and three states for entering and exiting each. The three states are determined upon the LED signals that was inputted – one where only a is triggered, one where only b is triggered, and one where both a and b are triggered. However, both enter and exit need its own states for same input signals because the system needs to differentiate if the car is in the process of entering or in the process of exiting.

After the FSM module, the counter module is used to then determine how many cars are currently at the parking lot. The counter module keeps track of how many cars are in the parking lot by taking in input signals increment and decrement that were delivered from the enter and exit outputs of the FSM module. If it receives a signal to increment, then it increases the number of cars in the parking lot by one, and when it receives a signal to decrement, it decreases the number of cars in the parking lot by one. It then returns the final number of cars in the parking lot in an output called num.

Finally, the display module is the one that puts the number of cars in the parking lot into the HEXs. It uses 7-bit input to correctly display the number of cars if the parking lot is not empty nor full, and the word “empty” when it is unoccupied, and the word “FULL” when there are 25 cars in the parking lot.

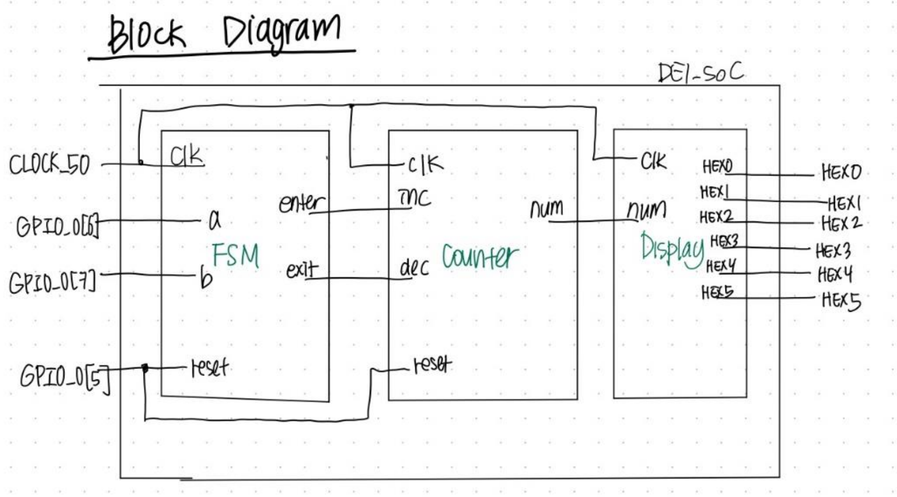


Figure 2.The Block Diagram of the Parking Lot Counter

Above represents the block diagram of the entire system.

Results

After each module was created, I tested to make sure that all parts of the systems are working correctly by running ModelSim tests.

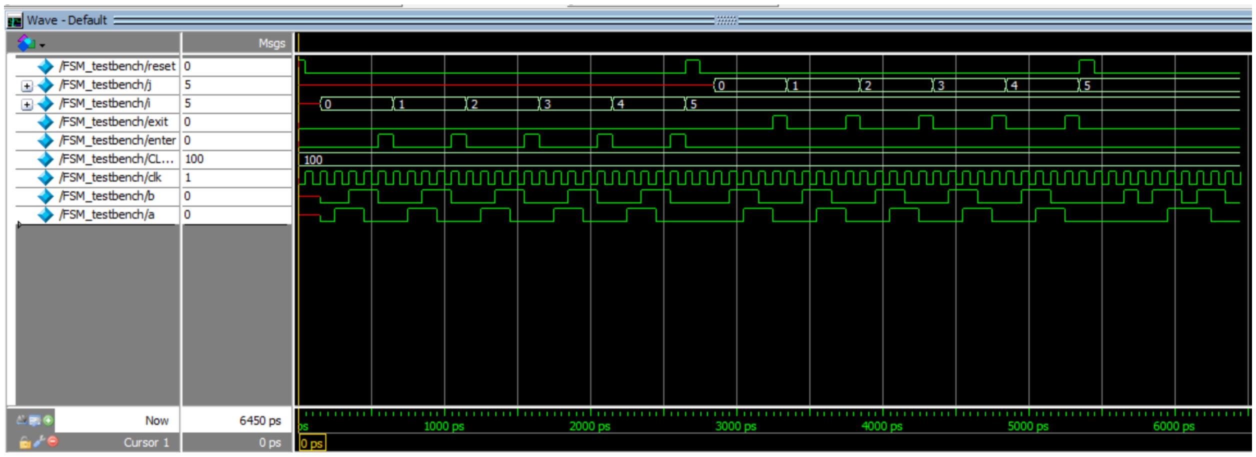


Figure 3. The ModelSim of the FSM Module

Figure 3 is the ModelSim graph of the FSM module. I tested that when the a and b signals are inputted in the correct enter and exit orders, the outputted enter or exit signals are correct.

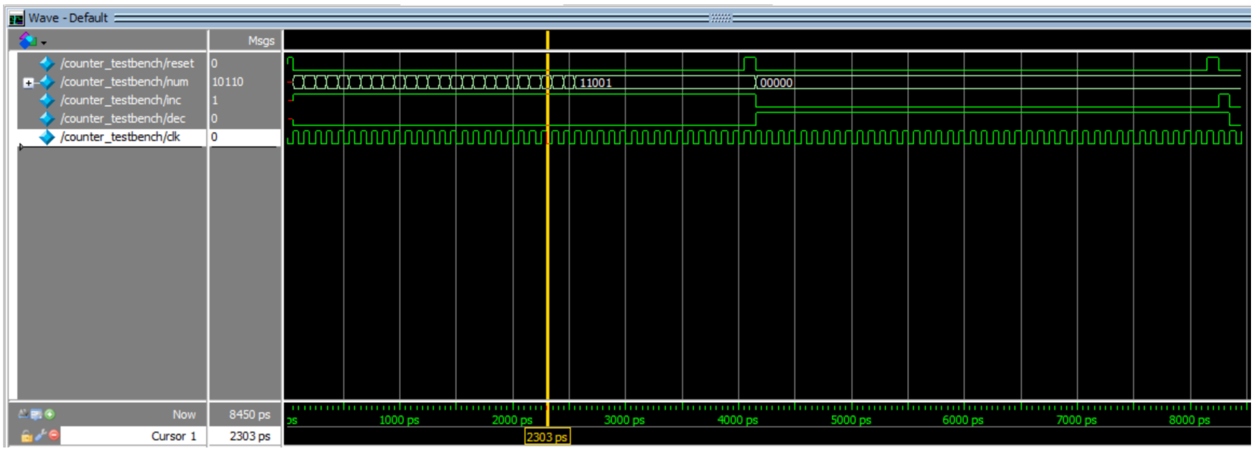
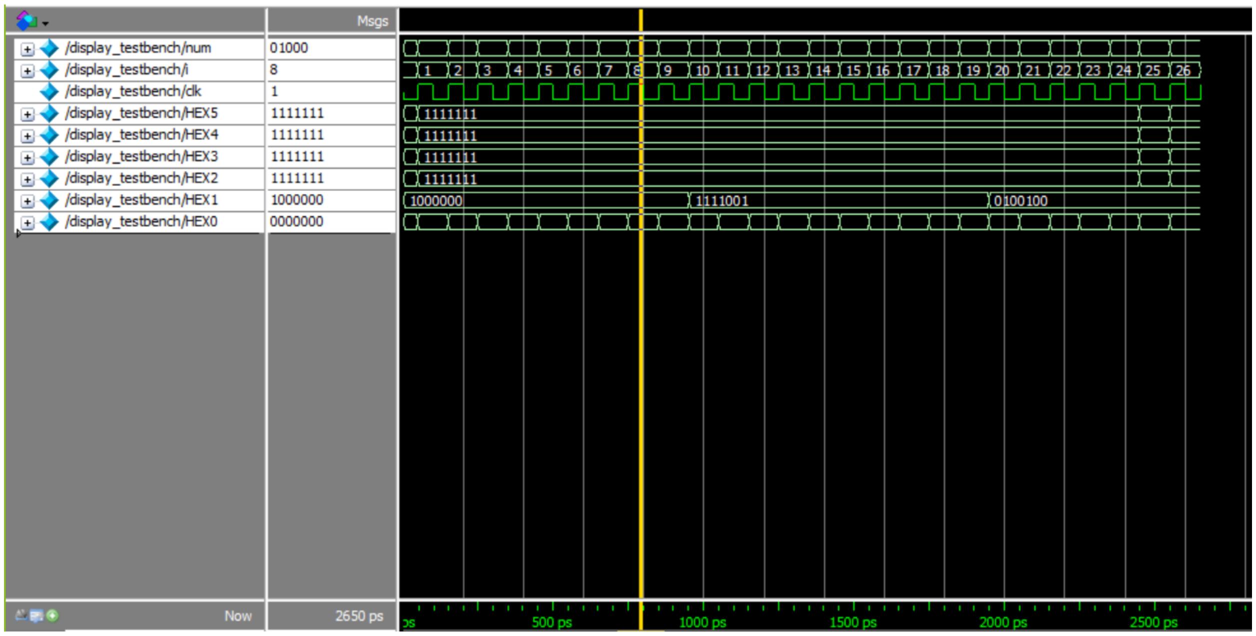


Figure 4. The ModelSim of the Counter Module

Figure 4 is the ModelSim graph of the Counter module. I tested that incrementing 40 times and decrementing 40 times both return correct results of the number of cars in the system. I tested 40 times to make sure that in edge cases of the parking lot being empty and the parking lot being full, the module still works as intended.



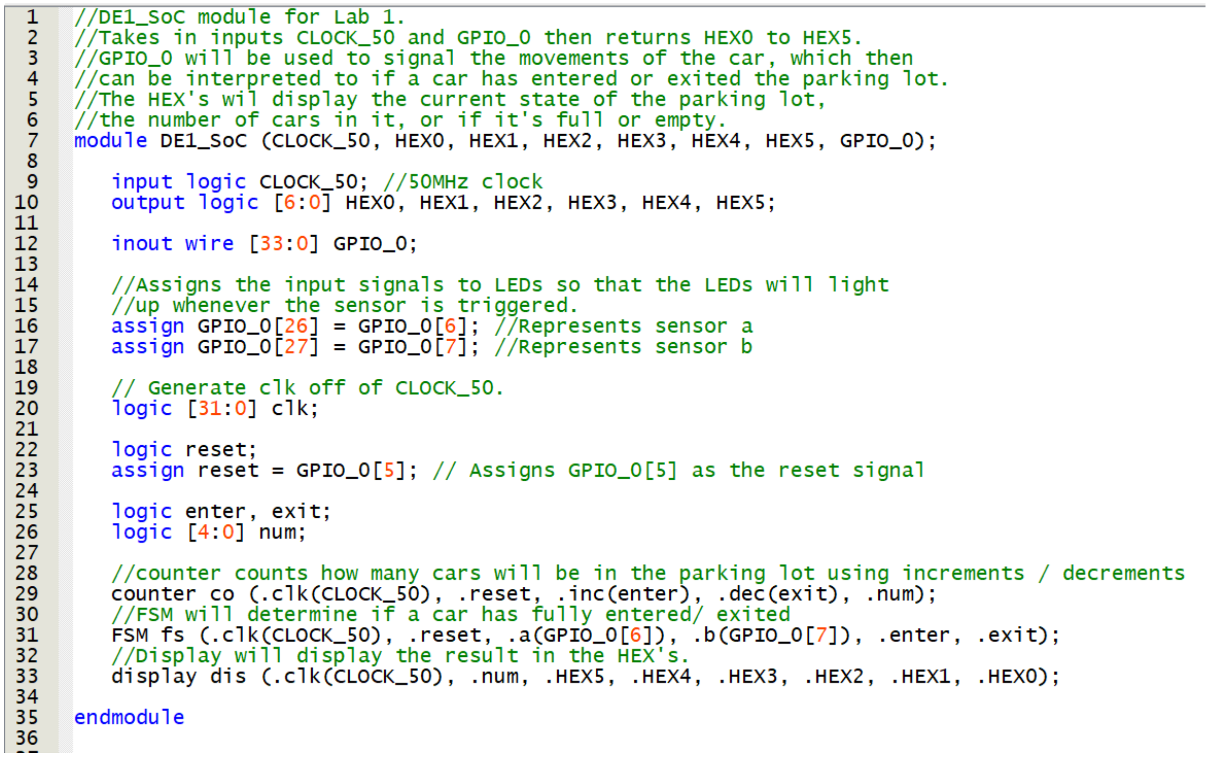
```
                              Msgs
/display_testbench/num        01000
/display_testbench/i          8
/display_testbench/clk        1
/display_testbench/HEX5       1111111
/display_testbench/HEX4       1111111
/display_testbench/HEX3       1111111
/display_testbench/HEX2       1111111
/display_testbench/HEX1       1000000
/display_testbench/HEX0       0000000
Now                           2650 ps
```

Figure 5. The ModelSim of the Display Module

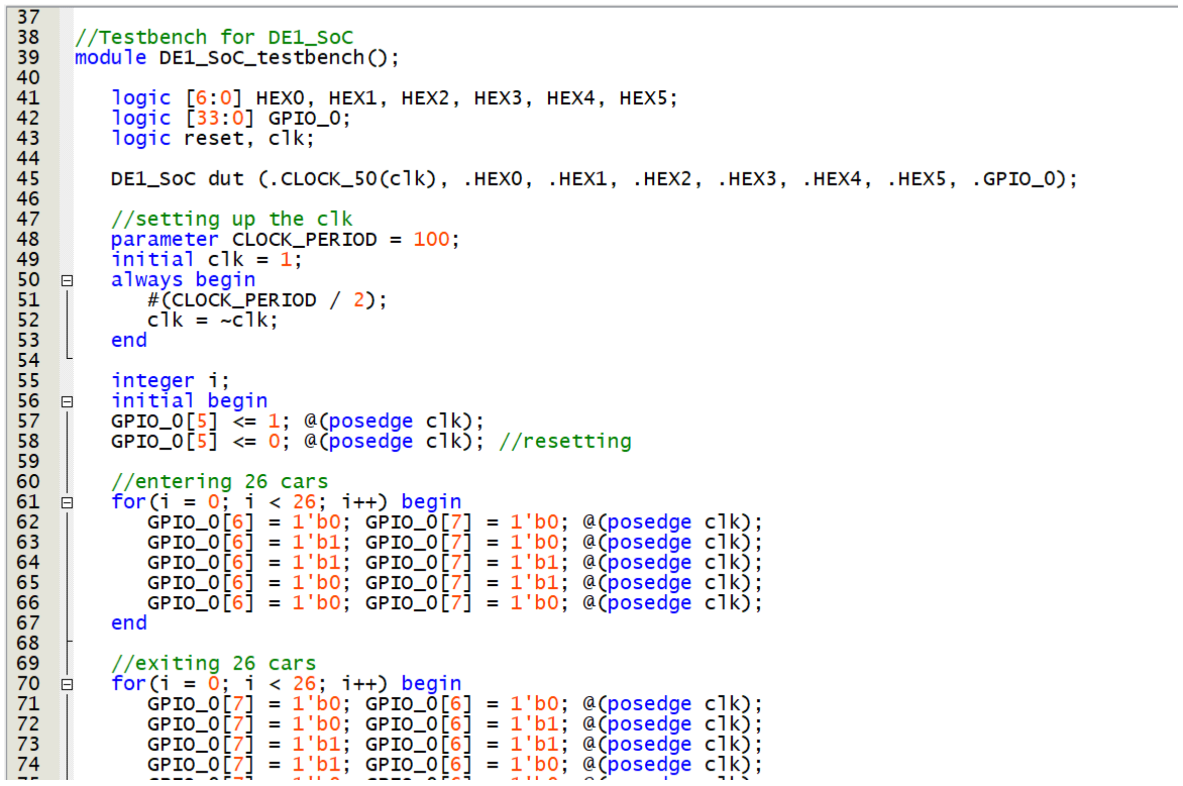
Figure 5 is the ModelSim result of the Display module. Here, I tested that all numbers are displaying correctly onto the HEXs and I was able to check that it indeed does display correctly.

Appendix: SystemVerilog Code

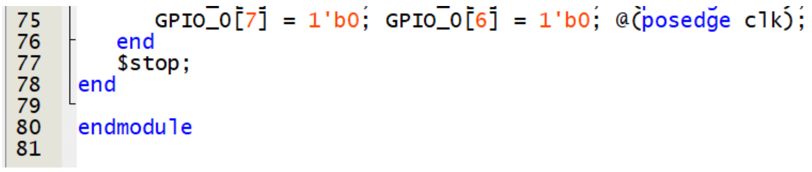
1) DE1_SoC



```
//DE1_SoC module for Lab 1.
//Takes in inputs CLOCK_50 and GPIO_0 then returns HEX0 to HEX5.
//GPIO_0 will be used to signal the movements of the car, which then
//can be interpreted to if a car has entered or exited the parking lot.
//The HEX's wil display the current state of the parking lot,
//the number of cars in it, or if it's full or empty.
module DE1_SoC (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 wire [33:0] GPIO_0;

   //Assigns the input signals to LEDs so that the LEDs will light
   //up whenever the sensor is triggered.
   assign GPIO_0[26] = GPIO_0[6]; //Represents sensor a
   assign GPIO_0[27] = GPIO_0[7]; //Represents sensor b

   // Generate clk off of CLOCK_50.
   logic [31:0] clk;

   logic reset;
   assign reset = GPIO_0[5]; // Assigns GPIO_0[5] as the reset signal

   logic enter, exit;
   logic [4:0] num;

   //counter counts how many cars will be in the parking lot using increments / decrements
   counter co (.clk(CLOCK_50), .reset, .inc(enter), .dec(exit), .num);
   //FSM will determine if a car has fully entered/ exited
   FSM fs (.clk(CLOCK_50), .reset, .a(GPIO_0[6]), .b(GPIO_0[7]), .enter, .exit);
   //Display will display the result in the HEX's.
   display dis (.clk(CLOCK_50), .num, .HEX5, .HEX4, .HEX3, .HEX2, .HEX1, .HEX0);

endmodule

```

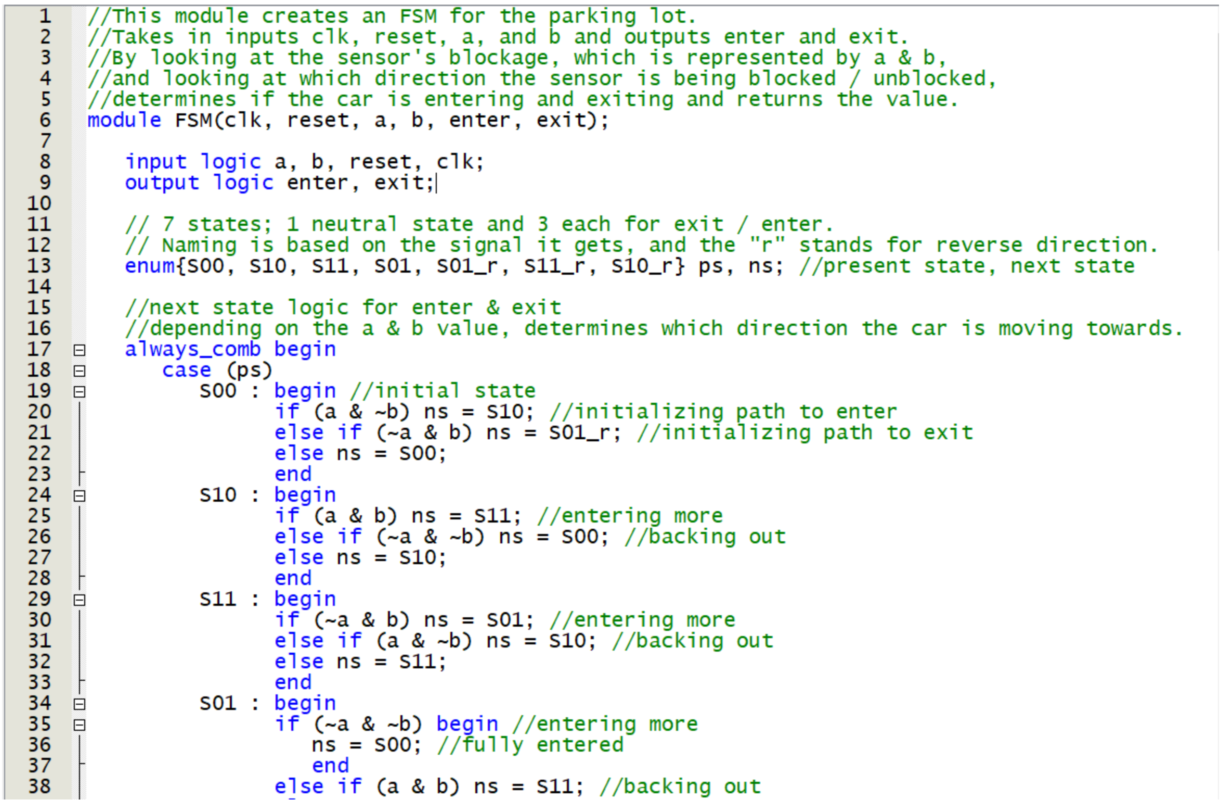


```

//Testbench for DE1_SoC
module DE1_SoC_testbench();

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

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

   //setting up the clk
   parameter CLOCK_PERIOD = 100;
   initial clk = 1;
   always begin
      #(CLOCK_PERIOD / 2);
      clk = ~clk;
   end

   integer i;
   initial begin
   GPIO_0[5] <= 1; @(posedge clk);
   GPIO_0[5] <= 0; @(posedge clk); //resetting

   //entering 26 cars
   for(i = 0; i < 26; i++) begin
      GPIO_0[6] = 1'b0; GPIO_0[7] = 1'b0; @(posedge clk);
      GPIO_0[6] = 1'b1; GPIO_0[7] = 1'b0; @(posedge clk);
      GPIO_0[6] = 1'b1; GPIO_0[7] = 1'b1; @(posedge clk);
      GPIO_0[6] = 1'b0; GPIO_0[7] = 1'b1; @(posedge clk);
      GPIO_0[6] = 1'b0; GPIO_0[7] = 1'b0; @(posedge clk);
   end

   //exiting 26 cars
   for(i = 0; i < 26; i++) begin
      GPIO_0[7] = 1'b0; GPIO_0[6] = 1'b0; @(posedge clk);
      GPIO_0[7] = 1'b0; GPIO_0[6] = 1'b1; @(posedge clk);
      GPIO_0[7] = 1'b1; GPIO_0[6] = 1'b1; @(posedge clk);
      GPIO_0[7] = 1'b1; GPIO_0[6] = 1'b0; @(posedge clk);
```

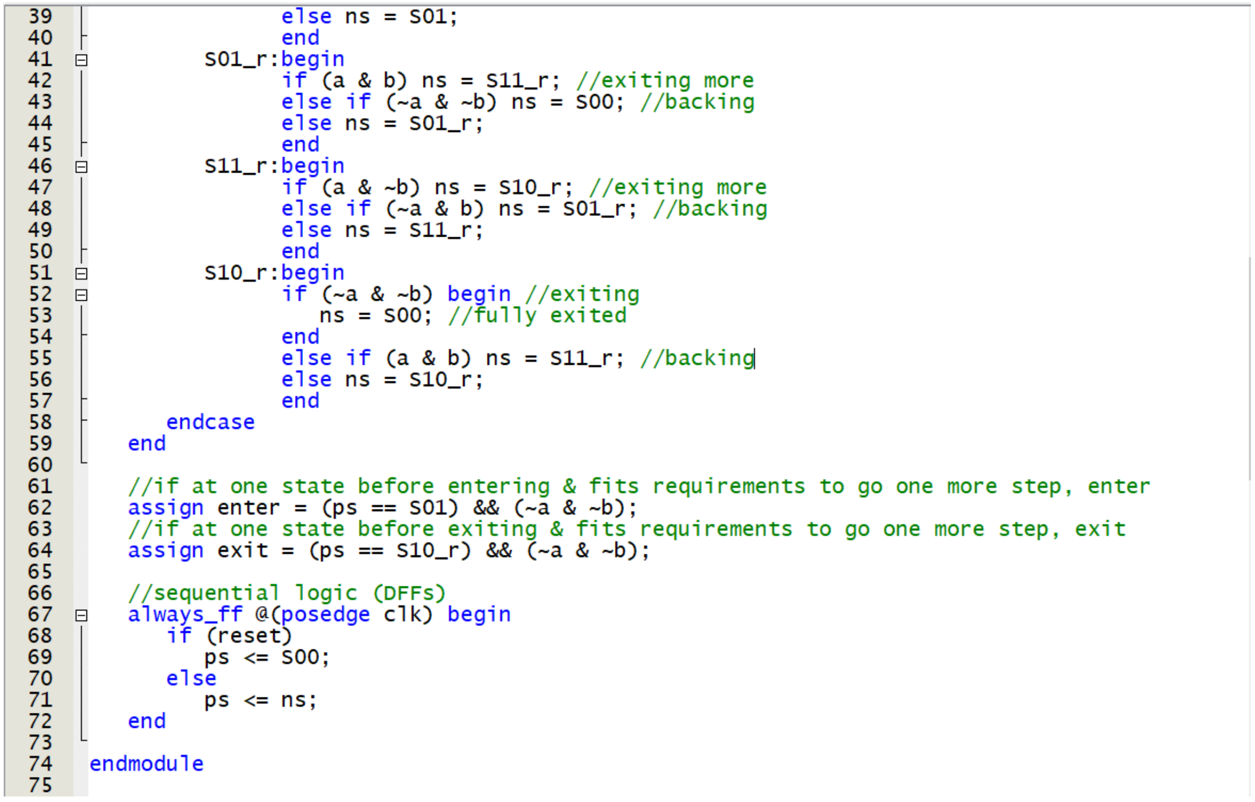


```
        GPIO_O[7] = 1'b0; GPIO_O[6] = 1'b0; @(posedge clk);
    end
    $stop;
end

endmodule
```

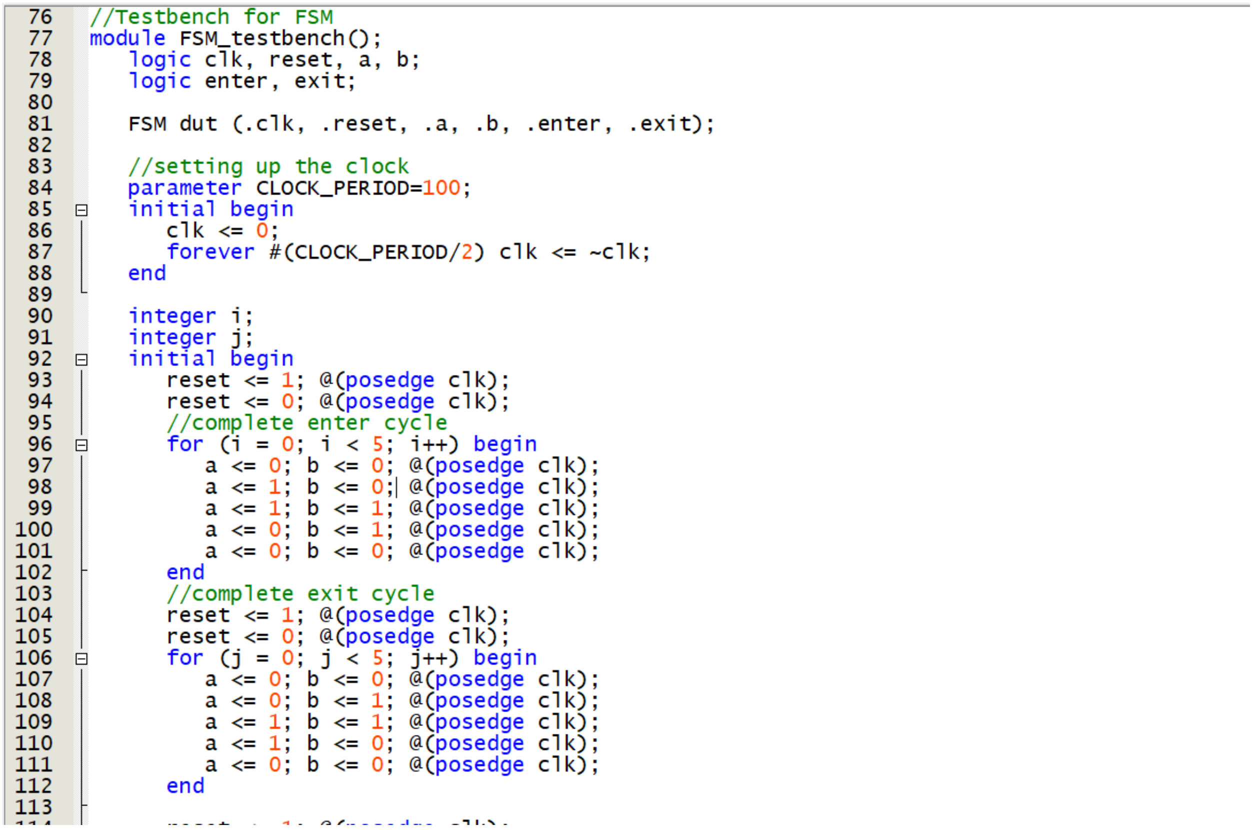
2) FSM



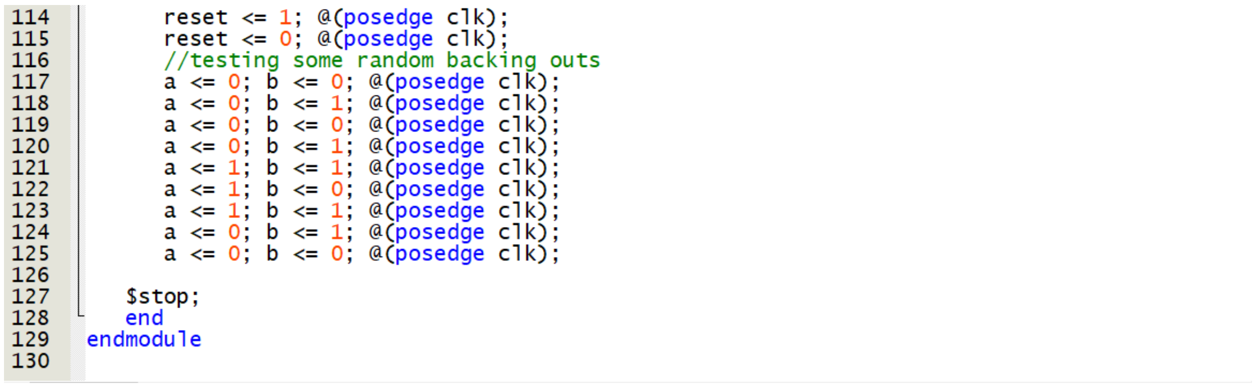
```
//This module creates an FSM for the parking lot.
//Takes in inputs clk, reset, a, and b and outputs enter and exit.
//By looking at the sensor's blockage, which is represented by a & b,
//and looking at which direction the sensor is being blocked / unblocked,
//determines if the car is entering and exiting and returns the value.
module FSM(clk, reset, a, b, enter, exit);

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

   // 7 states; 1 neutral state and 3 each for exit / enter.
   // Naming is based on the signal it gets, and the "r" stands for reverse direction.
   enum{S00, S10, S11, S01, S01_r, S11_r, S10_r} ps, ns; //present state, next state

   //next state logic for enter & exit
   //depending on the a & b value, determines which direction the car is moving towards.
   always_comb begin
      case (ps)
         S00 : begin //initial state
               if (a & ~b) ns = S10; //initializing path to enter
               else if (~a & b) ns = S01_r; //initializing path to exit
               else ns = S00;
               end
         S10 : begin
               if (a & b) ns = S11; //entering more
               else if (~a & ~b) ns = S00; //backing out
               else ns = S10;
               end
         S11 : begin
               if (~a & b) ns = S01; //entering more
               else if (a & ~b) ns = S10; //backing out
               else ns = S11;
               end
         S01 : begin
               if (~a & ~b) begin //entering more
                  ns = S00; //fully entered
                  end
               else if (a & b) ns = S11; //backing out
```



```
               else ns = S01;
               end
         S01_r:begin
               if (a & b) ns = S11_r; //exiting more
               else if (~a & ~b) ns = S00; //backing
               else ns = S01_r;
               end
         S11_r:begin
               if (a & ~b) ns = S10_r; //exiting more
               else if (~a & b) ns = S01_r; //backing
               else ns = S11_r;
               end
         S10_r:begin
               if (~a & ~b) begin //exiting
                  ns = S00; //fully exited
               end
               else if (a & b) ns = S11_r; //backing
               else ns = S10_r;
               end
      endcase
   end

   //if at one state before entering & fits requirements to go one more step, enter
   assign enter = (ps == S01) && (~a & ~b);
   //if at one state before exiting & fits requirements to go one more step, exit
   assign exit = (ps == S10_r) && (~a & ~b);

   //sequential logic (DFFs)
   always_ff @(posedge clk) begin
      if (reset)
         ps <= S00;
      else
         ps <= ns;
   end

endmodule
```

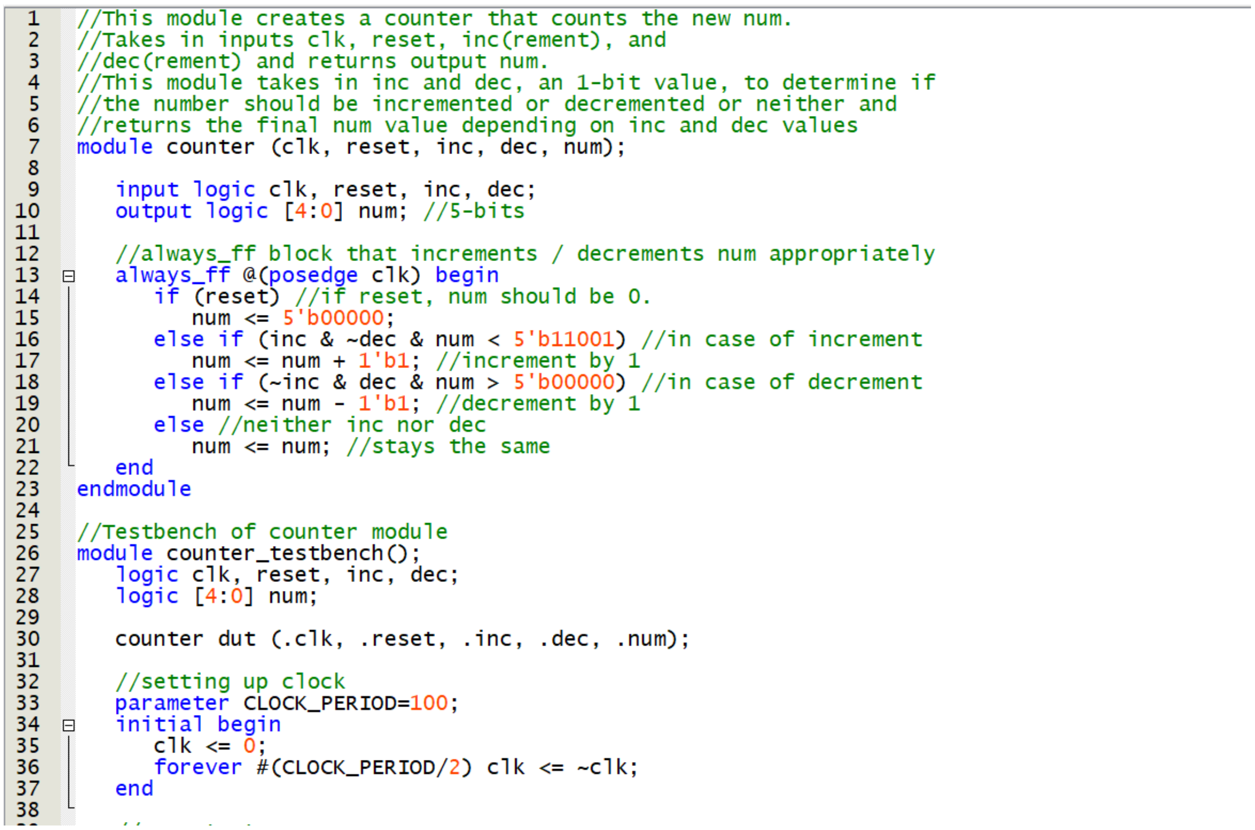


```
//Testbench for FSM
module FSM_testbench();
   logic clk, reset, a, b;
   logic enter, exit;

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

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

   integer i;
   integer j;
   initial begin
      reset <= 1; @(posedge clk);
      reset <= 0; @(posedge clk);
      //complete enter cycle
      for (i = 0; i < 5; i++) begin
         a <= 0; b <= 0; @(posedge clk);
         a <= 1; b <= 0; @(posedge clk);
         a <= 1; b <= 1; @(posedge clk);
         a <= 0; b <= 1; @(posedge clk);
         a <= 0; b <= 0; @(posedge clk);
      end
      //complete exit cycle
      reset <= 1; @(posedge clk);
      reset <= 0; @(posedge clk);
      for (j = 0; j < 5; j++) begin
         a <= 0; b <= 0; @(posedge clk);
         a <= 0; b <= 1; @(posedge clk);
         a <= 1; b <= 1; @(posedge clk);
         a <= 1; b <= 0; @(posedge clk);
         a <= 0; b <= 0; @(posedge clk);
      end

```

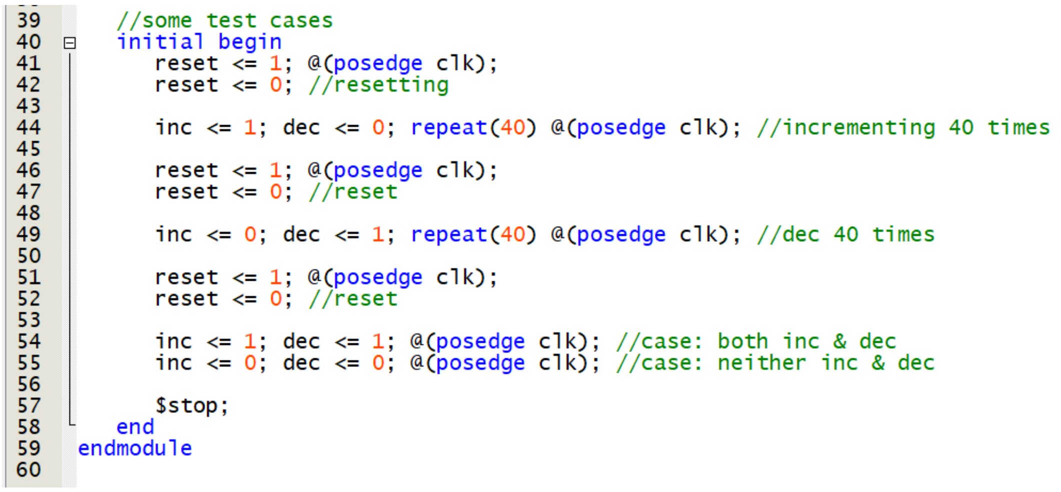


```
      reset <= 1; @(posedge clk);
      reset <= 0; @(posedge clk);
      //testing some random backing outs
      a <= 0; b <= 0; @(posedge clk);
      a <= 0; b <= 1; @(posedge clk);
      a <= 0; b <= 0; @(posedge clk);
      a <= 0; b <= 1; @(posedge clk);
      a <= 1; b <= 1; @(posedge clk);
      a <= 1; b <= 0; @(posedge clk);
      a <= 1; b <= 1; @(posedge clk);
      a <= 0; b <= 1; @(posedge clk);
      a <= 0; b <= 0; @(posedge clk);

   $stop;
   end
endmodule

```

3) Counter

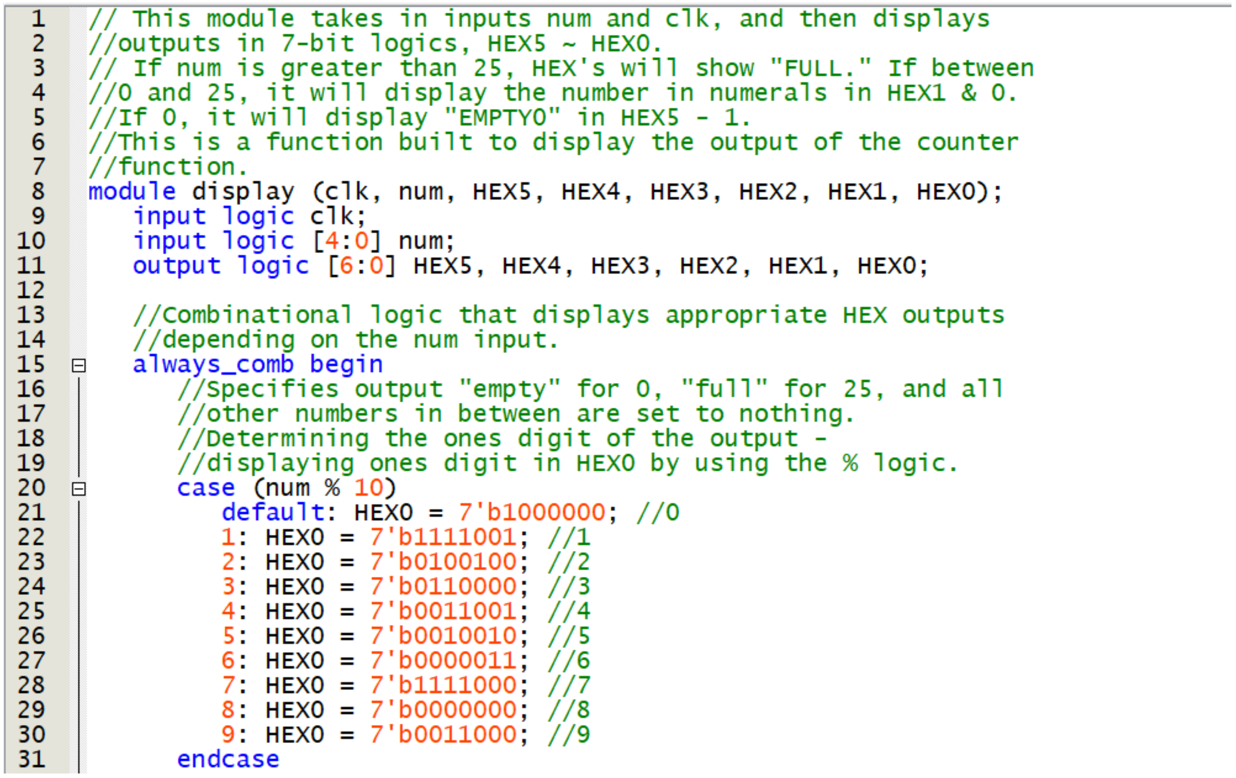


```
//This module creates a counter that counts the new num.
//Takes in inputs clk, reset, inc(rement), and
//dec(rement) and returns output num.
//This module takes in inc and dec, an 1-bit value, to determine if
//the number should be incremented or decremented or neither and
//returns the final num value depending on inc and dec values
module counter (clk, reset, inc, dec, num);

   input logic clk, reset, inc, dec;
   output logic [4:0] num; //5-bits

   //always_ff block that increments / decrements num appropriately
   always_ff @(posedge clk) begin
      if (reset) //if reset, num should be 0.
         num <= 5'b00000;
      else if (inc & ~dec & num < 5'b11001) //in case of increment
         num <= num + 1'b1; //increment by 1
      else if (~inc & dec & num > 5'b00000) //in case of decrement
         num <= num - 1'b1; //decrement by 1
      else //neither inc nor dec
         num <= num; //stays the same
   end
endmodule

//Testbench of counter module
module counter_testbench();
   logic clk, reset, inc, dec;
   logic [4:0] num;

   counter dut (.clk, .reset, .inc, .dec, .num);

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

```

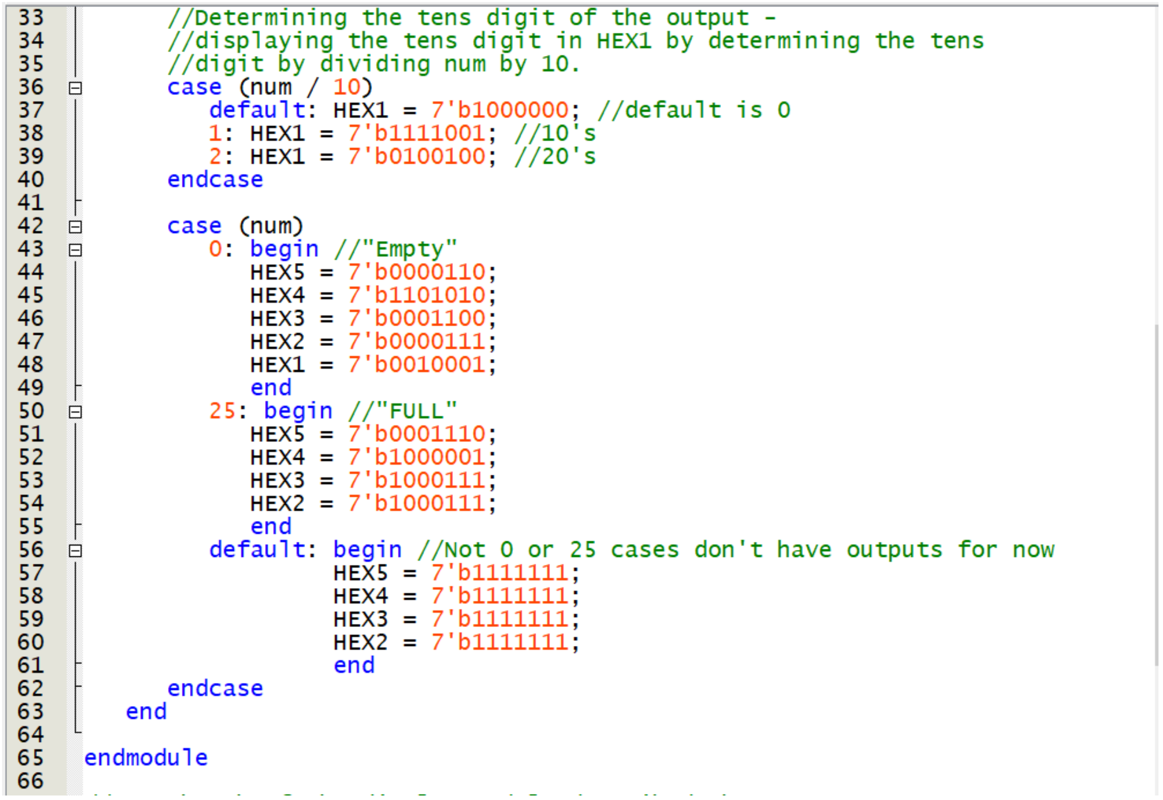


```
   //some test cases
   initial begin
      reset <= 1; @(posedge clk);
      reset <= 0; //resetting

      inc <= 1; dec <= 0; repeat(40) @(posedge clk); //incrementing 40 times

      reset <= 1; @(posedge clk);
      reset <= 0; //reset

      inc <= 0; dec <= 1; repeat(40) @(posedge clk); //dec 40 times

      reset <= 1; @(posedge clk);
      reset <= 0; //reset

      inc <= 1; dec <= 1; @(posedge clk); //case: both inc & dec
      inc <= 0; dec <= 0; @(posedge clk); //case: neither inc & dec

      $stop;
   end
endmodule

```

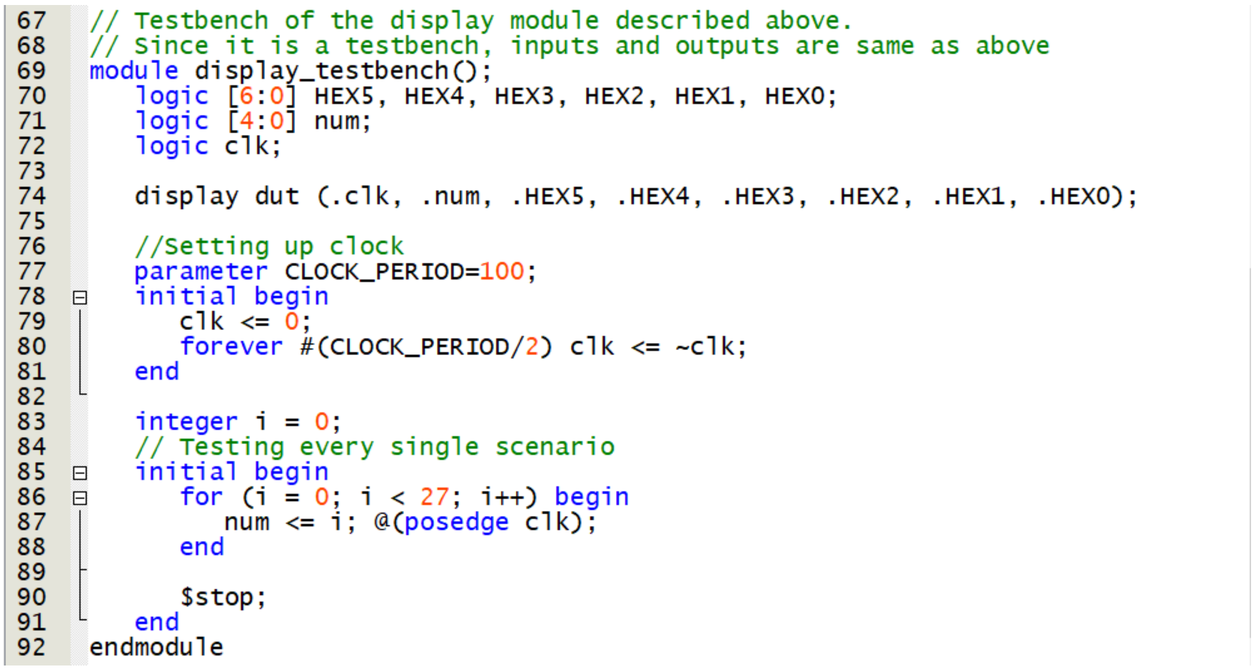
4) Display



```
// This module takes in inputs num and clk, and then displays
//outputs in 7-bit logics, HEX5 ~ HEX0.
// If num is greater than 25, HEX's will show "FULL." If between
//0 and 25, it will display the number in numerals in HEX1 & 0.
//If 0, it will display "EMPTY0" in HEX5 - 1.
//This is a function built to display the output of the counter
//function.
module display (clk, num, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);
   input logic clk;
   input logic [4:0] num;
   output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;

   //Combinational logic that displays appropriate HEX outputs
   //depending on the num input.
   always_comb begin
      //Specifies output "empty" for 0, "full" for 25, and all
      //other numbers in between are set to nothing.
      //Determining the ones digit of the output -
      //displaying ones digit in HEX0 by using the % logic.
      case (num % 10)
         default: HEX0 = 7'b1000000; //0
         1: HEX0 = 7'b1111001; //1
         2: HEX0 = 7'b0100100; //2
         3: HEX0 = 7'b0110000; //3
         4: HEX0 = 7'b0011001; //4
         5: HEX0 = 7'b0010010; //5
         6: HEX0 = 7'b0000011; //6
         7: HEX0 = 7'b1111000; //7
         8: HEX0 = 7'b0000000; //8
         9: HEX0 = 7'b0011000; //9
      endcase
```



```
      //Determining the tens digit of the output -
      //displaying the tens digit in HEX1 by determining the tens
      //digit by dividing num by 10.
      case (num / 10)
         default: HEX1 = 7'b1000000; //default is 0
         1: HEX1 = 7'b1111001; //10's
         2: HEX1 = 7'b0100100; //20's
      endcase

      case (num)
         0: begin //"Empty"
            HEX5 = 7'b0000110;
            HEX4 = 7'b1101010;
            HEX3 = 7'b0001100;
            HEX2 = 7'b0000111;
            HEX1 = 7'b0010001;
            end
         25: begin //"FULL"
            HEX5 = 7'b0001110;
            HEX4 = 7'b1000001;
            HEX3 = 7'b1000111;
            HEX2 = 7'b1000111;
            end
         default: begin //Not 0 or 25 cases don't have outputs for now
                  HEX5 = 7'b1111111;
                  HEX4 = 7'b1111111;
                  HEX3 = 7'b1111111;
                  HEX2 = 7'b1111111;
                  end
      endcase
   end

endmodule

```



```
// Testbench of the display module described above.
// Since it is a testbench, inputs and outputs are same as above
module display_testbench();
   logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   logic [4:0] num;
   logic clk;

   display dut (.clk, .num, .HEX5, .HEX4, .HEX3, .HEX2, .HEX1, .HEX0);

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

   integer i = 0;
   // Testing every single scenario
   initial begin
      for (i = 0; i < 27; i++) begin
         num <= i; @(posedge clk);
      end

      $stop;
   end
endmodule
```