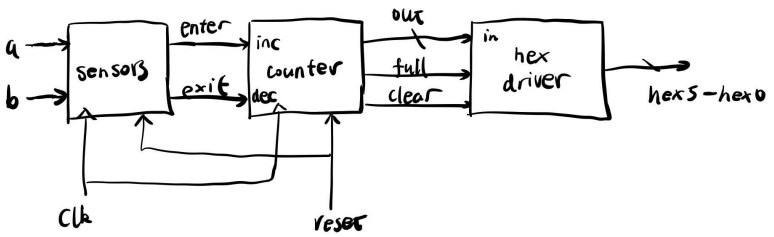
Peter Zhong
EE 371
Apr. 12th, 2021
Lab 1 Report

Procedure

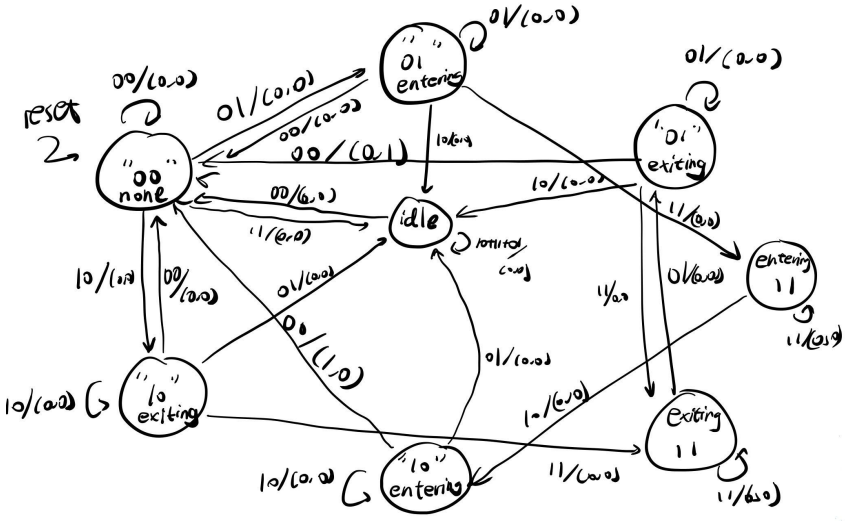
Block diagram for the entire system:



(Figure 1: Block Diagram for the Entire System)

Task #1: Pattern Recognition Module

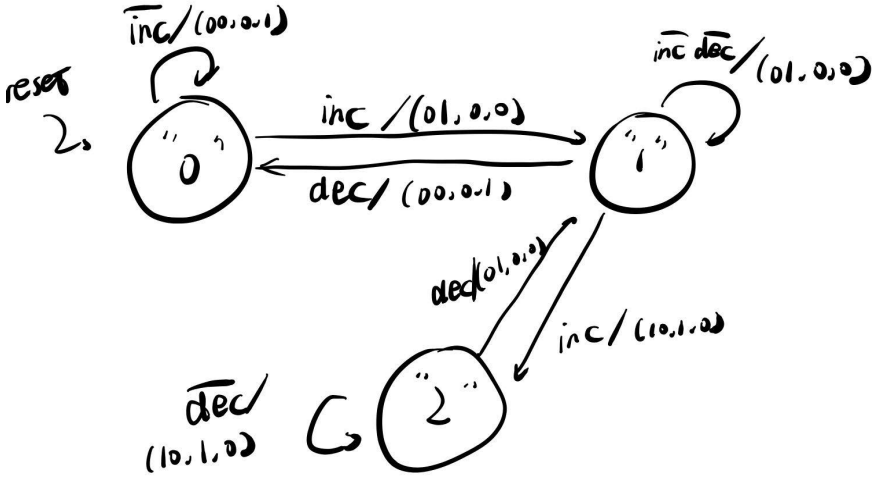
- This module takes the inputs from the two sensors, sends out an “enter” signal when an entering vehicle is detected, and sends out an “exit” signal when an exiting vehicle is detected. These two signals are used by the counter module to determine whether to increment or decrement.
- To implement this module, I designed an eight-state FSM with two separate loops (one for entering and one for exiting) and one idle state to account for the messy and meaningless inputs. The FSM is separated into two loops, since the machine needs to remember whether the current signal is caused by an entering vehicle or an exiting vehicle. The state diagram for this FSM is shown below in Figure 2.



(Figure 2: State Diagram for the Pattern Recognition FSM (“sensors” module))

Task #2: Counter Module

- This module takes an "inc" and "dec" signal, and increments/decrements the number currently stored in the machine. The minimum number that can be stored is 0, while the maximum number can be determined by the module's "MAX" parameter.
- This module outputs the current number stored (in 5 bits). It also sends two additional signals, "full" and "clear". When the number stored is equal to MAX, "full" is set to true. When the number stored is zero, "clear" is set to true.
- The "full" and "clear" signals are used by the Hex Display Driver Module to determine whether to display the messages on HEX5-HEX2. The 5-bit output is also used by the driver module to determine the number that should be displayed on HEX1 and HEX0.
- Figure 3 below shows an example state diagram of the counter when MAX = 2.



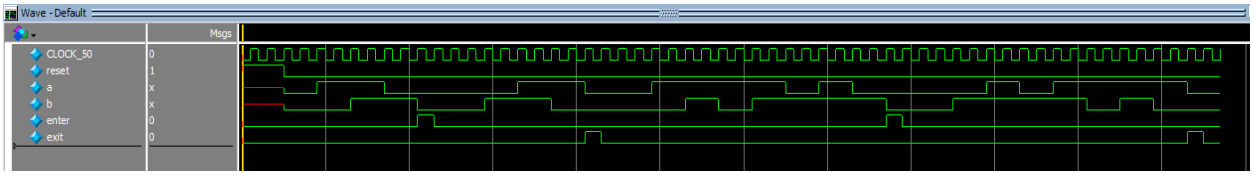
(Figure 3: State Diagram for the Counter Module when MAX = 2)

Task #3: Hex Display Driver Module

- This module drives all 6 of the hex displays on the DE1-SoC board. I designed the module to be a fully combinational module, since it only needs to listen to the inputs given by the counter and does not need to remember anything.
- It displays the "full" and "clear" message according to its "full" and "clear" inputs. It also displays the decimal value of the 5-bit input on HEX1 and HEX0.
- To make the code easier to read, I created 7-bit logics within the module for all numbers and letters used to display messages. This can be seen in the screenshots of the SystemVerilog code included below in the Appendix section.

Results

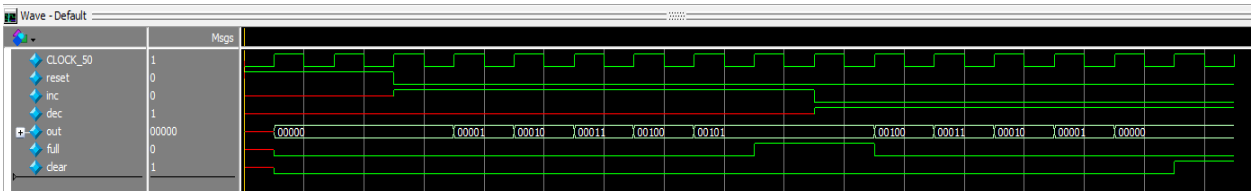
Screenshot for sensors module simulation:



(Figure 4: Screenshot of sensors Module Simulation)

- For the sensors module, I simulated 4 different situations: an entering vehicle, an exiting vehicle, changing direction while entering, and changing direction while exiting. As can be seen from the waveform above, this module correctly outputs one true cycle to either “enter” or “exit” output when an entering or an exiting vehicle is detected.

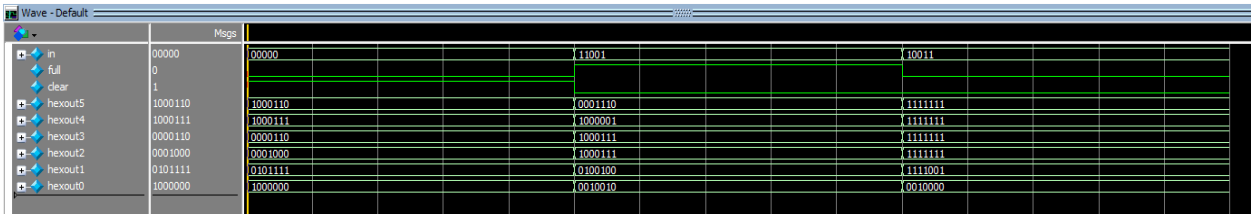
Screenshot for counter module simulation:



(Figure 5: Screenshot of counter Module Simulation)

- For the counter module, I set the maximum value to 5 and tested 2 different situations: incrementing past upper limit, and decrementing past lower limit. As can be seen from the waveform above, the counter behaves properly. It stops incrementing and sets “full” to true when the count reaches the maximum number, and it stops decrementing and sets “clear” to true when the count reaches zero.

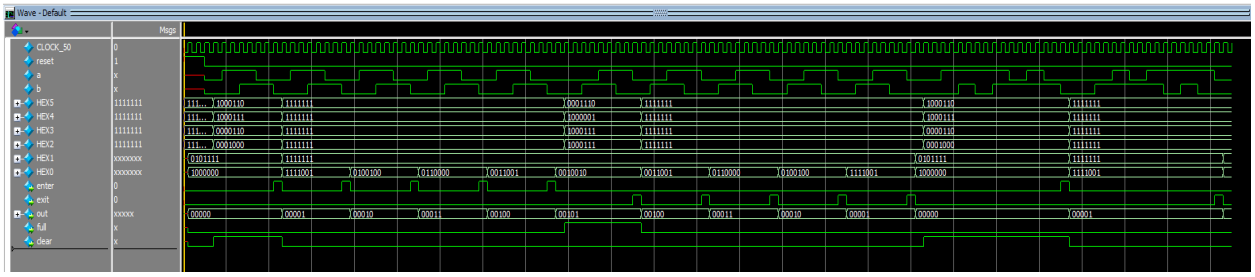
Screenshot for hex_driver module simulation:



(Figure 6: Screenshot of hex_driver Module Simulation)

- For the hex_driver module, I tested 3 different situations: clear, full, and a randomly selected number. As can be from the waveform above, the hex_driver properly outputs different messages to the hex displays at appropriate times.

Screenshot for top-level simulation:



(Figure 7: Screenshot of Top-Level Simulation)

- For my top-level simulation, I tested 4 different situations: entering until full, exiting until clear, messy entering, and messy exiting. As can be seen from my top-level simulation above, my overall top-level module works as intended: it counts cars and displays “full” and “clear” messages accordingly, and it is not disturbed by messy patterns or change of directions upon entering or exiting.

Demonstration Video: https://youtu.be/DxUaFDIO50Q

Overview of the Finished System: The overall goal of this project is to design a parking lot counter that detects entering and exiting vehicles through recognizing patterns from sensors. By using a pattern recognition FSM, a counter, and a hex display driver, I successfully achieved this goal. As described in the lab specifications, my system accurately recognizes entering and exiting vehicles, keeps track of the number of vehicles in the parking lot, and is not disturbed by vehicles changing directions while entering and exiting.

Appendix

1) sensors.sv

```
// Peter Zhong
// 04/12/2020
// EE 371
// Lab #1. Task 1

// sensors_FSM takes inputs from two sensors and output "1" to either enter or exit for 1 clock cycle
// whenever an entering or exiting vehicle is detected.

module sensors (a, b, enter, exit, clk, reset);

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

   enum {none, entering01, exiting01, entering11, exiting11, entering10, exiting10, idle} ps, ns;

   // Logic for next state: using data from sensors to detect an entering/exiting vehicle.
   always_comb begin
      case(ps)
         none: if (~a & ~b) ns = none;
               else if (~a & b) ns = entering01;
               else if (a & ~b) ns = exiting10;
               else ns = idle;
         entering01: if (~a & ~b) ns = none;
               else if (~a & b) ns = entering01;
               else if (a & ~b) ns = idle;
               else ns = entering11;
         exiting01: if (~a & ~b) ns = none;
               else if (~a & b) ns = exiting01;
               else if (a & ~b) ns = idle;
               else ns = exiting11;
         entering11: if (~a & ~b) ns = none;
               else if (~a & b) ns = entering01;
               else if (a & ~b) ns = entering10;
               else ns = entering11;
         exiting11: if (~a & ~b) ns = none;
               else if (~a & b) ns = exiting01;
               else if (a & ~b) ns = exiting10;
               else ns = exiting11;
         entering10: if (~a & ~b) ns = none;
               else if (~a & b) ns = idle;
               else if (a & ~b) ns = entering10;
               else ns = entering11;
         exiting10: if (~a & ~b) ns = none;
               else if (~a & b) ns = idle;
               else if (a & ~b) ns = exiting10;
               else ns = exiting11;
         idle: if (~a & ~b) ns = none;
               else ns = idle;
      endcase
   end

   //Output logic for entering: outputs 1 to enter when an entering vehicle is detected.
   always_comb begin
      case(ps)
         entering10: if (~a & ~b) enter = 1'b1;
                     else enter = 1'b0;
         default: enter = 1'b0;
      endcase
   end

   //output logic for exiting: outputs 1 to exit when an exiting vehicle is detected.
   always_comb begin
      case(ps)
         exiting01: if (~a & ~b) exit = 1'b1;
                     else exit = 1'b0;
         default: exit = 1'b0;
      endcase
   end

   //DFFs
   always_ff @(posedge clk) begin
      if (reset)
         ps <= none;
      else
         ps <= ns;
   end

endmodule
```

```
// Testbench for the sensors module
module sensors_testbench();
   logic a, b, clk, reset, enter, exit;
   logic CLOCK_50;

   sensors dut (.a(b), .b(a), .clk(CLOCK_50), .reset, .enter, .exit);

   // Setting up a simulated clock.
   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
      reset <= 1;                        repeat(3) @(posedge CLOCK_50);

      //enters
      reset <= 0;       a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
      //exits
                        a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
                        a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                        a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);

      // direction changes while entering
                        a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);

      // direction changes while exiting
                        a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
                        a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                        a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                        a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
      $stop;
   end
endmodule
```

2) counter.sv

```
// Peter Zhong
// 04/12/2020
// EE 371
// Lab #1. Task 2

// counter takes two inputs (inc, dec). It adds 5'b00001 to out when inc is true, and subtracts 5'b00001
// from out when dec is true. Out has a minimum value of 5'b00000 and a maximum value determined by the
// parameter (25 by default).

module counter #(parameter MAX=25) (inc, dec, out, full, clear, clk, reset);

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

   // Sequential logic for counting up and counting down depending on the input.
   always_ff @(posedge clk) begin
      if (reset) begin
         out <= 5'b00000;
         full <= 1'b0;
         clear <= 1'b0;
      end
      else if (inc & out < MAX) begin //increment when not at max
         out <= out + 5'b00001;
         clear <= 1'b0;
      end
      else if (dec & out > 5'b00000) begin // decrement when not at min
         out <= out - 5'b00001;
         full <= 1'b0;
      end
      else if (out == MAX) begin // hold value at max, output full
         out <= MAX;
         full <= 1'b1;
      end
      else if (out == 5'b00000) begin // hold value at min, output clear
         out <= 5'b00000;
         clear <= 1'b1;
      end
      else
         out <= out; // hold value otherwise
   end

endmodule

// Testbench for the counter module
module counter_testbench();
   logic inc, dec, full, clear, reset;
   logic [4:0] out;
   logic CLOCK_50;

   counter #(5) dut (.inc, .dec, .out, .full, .clear, .clk(CLOCK_50), .reset);

   // Setting up a simulated clock.
   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;                       repeat(3) @(posedge CLOCK_50); // reset
      reset <= 0; inc <= 1;             repeat (7) @(posedge CLOCK_50); // inc past max limit
                  inc <= 0; dec <= 1; repeat(7) @(posedge CLOCK_50); // dec past min limit
      $stop;
   end
endmodule
```

3) hex_driver.sv

```
// Peter Zhong
// 05/12/2020
// EE 371
// Lab #1. Task 2

// hex_driver outputs correct decimal value to hex displays based on the output given by the counter.
// It also displays "FULL" and "CLEAR" according to the indicator outputs given by the counter.

 module hex_driver (in, full, clear, hexout0, hexout1, hexout2, hexout3, hexout4, hexout5);

   input logic full, clear;
   input logic [4:0] in;
   output logic [6:0] hexout0, hexout1, hexout2, hexout3, hexout4, hexout5;

   // Assigning hex display variables on necessary numbers.
   logic [6:0] hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7, hex8, hex9;
   assign hex0 = 7'b1000000; // 0
   assign hex1 = 7'b1111001; // 1
   assign hex2 = 7'b0100100; // 2
   assign hex3 = 7'b0110000; // 3
   assign hex4 = 7'b0011001; // 4
   assign hex5 = 7'b0010010; // 5
   assign hex6 = 7'b0000010; // 6
   assign hex7 = 7'b1111000; // 7
   assign hex8 = 7'b0000000; // 8
   assign hex9 = 7'b0010000; // 9

   // Assigning hex display variables on necessary letters.
   logic [6:0] hexf, hexu, hexl, hexc, hexe, hexa, hexr, hexoff;
   assign hexf = 7'b0001110; // F
   assign hexu = 7'b1000001; // U
   assign hexl = 7'b1000111; // L
   assign hexc = 7'b1000110; // C
   assign hexe = 7'b0000110; // E
   assign hexa = 7'b0001000; // A
   assign hexr = 7'b0101111; // R
   assign hexoff = 7'b1111111; // off

   // Logic for hexout0: 26 different cases for 26 numbers. (0-25)
   always_comb begin
      case(in)
         5'b00000: hexout0 = hex0;
         5'b00001: hexout0 = hex1;
         5'b00010: hexout0 = hex2;
         5'b00011: hexout0 = hex3;
         5'b00100: hexout0 = hex4;
         5'b00101: hexout0 = hex5;
         5'b00110: hexout0 = hex6;
         5'b00111: hexout0 = hex7;
         5'b01000: hexout0 = hex8;
         5'b01001: hexout0 = hex9;
         5'b01010: hexout0 = hex0;
         5'b01011: hexout0 = hex1;
         5'b01100: hexout0 = hex2;
         5'b01101: hexout0 = hex3;
         5'b01110: hexout0 = hex4;
         5'b01111: hexout0 = hex5;
         5'b10000: hexout0 = hex6;
         5'b10001: hexout0 = hex7;
         5'b10010: hexout0 = hex8;
         5'b10011: hexout0 = hex9;
         5'b10100: hexout0 = hex0;
         5'b10101: hexout0 = hex1;
         5'b10110: hexout0 = hex2;
         5'b10111: hexout0 = hex3;
         5'b11000: hexout0 = hex4;
         5'b11001: hexout0 = hex5;
         default: hexout0 = 7'bx;
      endcase
   end
```

```
   // Logic for hexout1: 26 different cases for 26 numbers. (0-25)
   always_comb begin
      case(in)
         5'b00000: hexout1 = hexr;
         5'b00001: hexout1 = hexoff;
         5'b00010: hexout1 = hexoff;
         5'b00011: hexout1 = hexoff;
         5'b00100: hexout1 = hexoff;
         5'b00101: hexout1 = hexoff;
         5'b00110: hexout1 = hexoff;
         5'b00111: hexout1 = hexoff;
         5'b01000: hexout1 = hexoff;
         5'b01001: hexout1 = hexoff;
         5'b01010: hexout1 = hex1;
         5'b01011: hexout1 = hex1;
         5'b01100: hexout1 = hex1;
         5'b01101: hexout1 = hex1;
         5'b01110: hexout1 = hex1;
         5'b01111: hexout1 = hex1;
         5'b10000: hexout1 = hex1;
         5'b10001: hexout1 = hex1;
         5'b10010: hexout1 = hex1;
         5'b10011: hexout1 = hex1;
         5'b10100: hexout1 = hex2;
         5'b10101: hexout1 = hex2;
         5'b10110: hexout1 = hex2;
         5'b10111: hexout1 = hex2;
         5'b11000: hexout1 = hex2;
         5'b11001: hexout1 = hex2;
         default: hexout1 = 7'bx;
      endcase
   end

   // Logic for hexout5 - hexout2: display letters when full or clear, turn off otherwise.
   always_comb begin
      if (full) begin
         hexout5 = hexf;
         hexout4 = hexu;
         hexout3 = hexl;
         hexout2 = hexl;
      end
      else if (clear) begin
         hexout5 = hexc;
         hexout4 = hexl;
         hexout3 = hexe;
         hexout2 = hexa;
      end
      else begin
         hexout5 = hexoff;
         hexout4 = hexoff;
         hexout3 = hexoff;
         hexout2 = hexoff;
      end
   end

endmodule


module hex_driver_testbench();
   logic full, clear;
   logic [4:0] in;
   logic [6:0] hexout0, hexout1, hexout2, hexout3, hexout4, hexout5;

   hex_driver dut (.in, .full, .clear, .hexout0, .hexout1, .hexout2, .hexout3, .hexout4, .hexout5);

   initial begin
      in = '0; clear = 1; full = 0; #10; // testing clear output
      in = 5'b11001; clear = 0; full = 1; #10; // testing full output
      in = 5'b10011; clear = 0; full = 0; #10; // testing regular output
      $stop;
   end
endmodule
```

4) DE1-SoC.sv

```
// Peter Zhong
// 04/12/2020
// EE 371
// Lab #1. Task 3 and 4

// DE1_SoC is the top-level module that defines the I/Os for the DE-1 SoC board.
// DE1_SoC takes three switches from the GPIO as inputs, and outputs to 2 LEDs on the breadboard through GPIO and 6 7-bit
// hex displays (HEX0-HEX5). It displays "full" or "clear" messages when the parking lot is either full or clear, and it
// displays the decimal value of the number of cars in the lot accordingly.

module DE1_SoC #(parameter MAX=25) (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, GPIO_0, CLOCK_50);
    output logic [6:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    inout logic [33:0] GPIO_0;
    input logic CLOCK_50;

   // Assigning and clk to CLOCK_50
   logic clk;
   assign clk = CLOCK_50;

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

   // Outputting a and b to breadboard
   assign GPIO_0[26] = GPIO_0[6];
   assign GPIO_0[27] = GPIO_0[10];

   // sensorab takes two switches from the breadboard as the input of the two parking sensors,
   // and outputs to enter and exit when an entering or exiting vehicle is detected.
   sensors sensorab (.a(GPIO_0[6]), .b(GPIO_0[10]), .enter, .exit, .clk, .reset(GPIO_0[14]));

   // counter ct takes enter and exit from sensors, and outputs the car count to counter_out.
   // it als outputs full and clear status to full and clear.
   counter #(MAX) ct (.inc(enter), .dec(exit), .out(counter_out), .full, .clear, .clk, .reset(GPIO_0[14]));

   // hex_driver h1 takes counter_out, full, and clear from ct, and outputs decimal values or status messages to hex displays.
   hex_driver h1 (.in(counter_out), .full, .clear, .hexout0(HEX0), .hexout1(HEX1), .hexout2(HEX2), .hexout3(HEX3), .hexout4(HEX4), .hexout5(HEX5));

endmodule

module DE1_SoC_testbench();
   logic [6:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   wire [33:0] GPIO_0;
   logic CLOCK_50;

   DE1_SoC #(5) dut (.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .GPIO_0, .CLOCK_50);

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

   // Assiging logic to wire
   logic reset, a, b;
   assign GPIO_0[14] = reset;
   assign GPIO_0[10] = a;
   assign GPIO_0[6] = b;
```

(See the remaining parts of the testbench on the page below)

```
    // Testing the module
    initial begin
        // reset
        reset <= 1;                     repeat(3) @(posedge CLOCK_50);

        //enters until full
        reset <= 0;         a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50); // 1st car enters
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50); // 2nd car enters
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50); // 3rd car enters
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50); // 4th car enters
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50); // 5th car enters, full

    // exits until clear
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50); // 1st car exits
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50); // 2nd car exits
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50); // 3rd car exits
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50); // 4th car exits
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50); // 5th car exits, clear

        // direction changes while entering
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
        // direction changes while exiting
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 1; repeat(2) @(posedge CLOCK_50);
                            a <= 1; b <= 0; repeat(2) @(posedge CLOCK_50);
                            a <= 0; b <= 0; repeat(2) @(posedge CLOCK_50);

        $stop;
    end
endmodule
```