```
// An Nguyen, Darren Do
// 3/8/2024
// EE 371
// Lab 6, task #2
//
// Inputs:
// CLOCK_50: Timer to indicate the functionality of our module.
// 4-bit KEY: KEY[0] is the only Key used in this lab, it increments the hour
//            count of our module.
// 10-bit SW: SW[9] is set to 'reset', SW[8] is set to 'start' signal.
// 13-bit V_GPIO: Inout signals that tells us the current functionality of the
//                the Parking lot.
//
// Outputs:
// 7-bit HEX0: Tells us how many spots are availble in our parking lot.
//             Goes blank when hour 8 hits.
//             It also tells if the parking lot if full or not.
// 7-bit HEX1: Before hour 8, HEX1 tells us if the parking lot is full.
//             After hour 8, it tells us the total number of cars
//             that has entered our parking lot at certain hours of the day.
// 7-bit HEX2: Before hour 8, HEX2 tells us if the parking lot is full.
//             After hour 8, HEX2 cycles through 7 different hours,
//             which represents the hour of operations during our rush hour day.
//             It also represents the read address needed to get the total car
//             count that has entered the parking lot that day.
// 7-bit HEX3: Before hour 8, HEX3 tells us if the parking lot is full.
//             After hour 8, HEX3 tells us if rush hour ended or not.
// 7-bit HEX4: Before hour 8, HEX4 is blank, since it does not serve a purpose here.
//             After hour 8, HEX4 tells us if rush hour started or not.
// 7-bit HEX5: Before hour 8, HEX5 tells us the current hour of operation.
//             After hour 8, HEX5 goes blank.
// 10-bit LEDR: Only Certain LEDRs are used in this module. Main job
//              is to signal the operations of various V_GPIO wires, i.e.
//              parking lot is full, there's a car entering, parking spot 1 is filled, etc.
//
// Logics:
// 4-bit current_hour: tells us the current hour, range: 0 -> 8.
//                     8th hour is not important to this lab.
// 4-bit rush_start: tells us at what hour, rush hour started, or not.
// 4-bit rush_end: tells us at what hour, rush hour ended, or not.
// 7-bit rush_start_hex: HEX representation of when rush hour started.
// 7-bit rush_end_hex: HEX representation of when rush hour ended.
// 7-bit final_rush_start_hex: Helps us decide if rush hour started, or
//                             just a '-' representing that rush hour did not start.
// 7-bit final_rush_end_hex: Helps us decide if rush hour ended, or
//                           just a '-' representing that rush hour did not end.
// full_final: tells us if the parking lot is full.
// outDD: signal from double flip flop to avoid metastability, used for ~KEY[0].
// no_meta_dd: Stable signal for KEY[0].
// outDD1: signal from double flip flop to avoid metastability, used for V_GPIO[31].
// no_meta_dd1: Stable signal for V_GPIO[31].
// 3-bit num_spots_avail: Tells us how many spots are available in the parking lot.
// 7-bit num_spots_avail_hex: HEX representation of how many spots are available in the parking lot.
// en1: Occurs after hour 7, signifies the start of cycling through RAM, to retreive the total
//      cars that enterred the parking lot at certain hours.
// 3-bit rd_addr_count: Representing the read address (hour), used to retreive the total number of cars that
//                      enterred parking at certain hour of the day.
// 16-bit rd_data: Represents the total car count that has entered the parking at certain time.
// 7-bit rd_data_hex: HEX representation of how many cars entered the parking lot at certain hours.
//                    Count is in HEXADECIMAL.
// 7-bit rd_addr_hex: HEX representation of current hour that's being cycled through.
// 7-bit current_hour_hex: HEX representation of current hour of operation.
// 7-bit HEX0_temp: Temp register for HEX0.
// 7-bit HEX1_temp: Temp register for HEX1.
// 7-bit HEX2_temp: Temp register for HEX2.
// 7-bit HEX3_temp: Temp register for HEX3.
// Summary: The DE1_SoC module is the top-level module for our parking lot system. It is responsible for
// running all the submodules that contribute to our parking lot system as well as
```

```
connecting our
// parking lot logic to the appropriate corresponding FPGA inputs and outputs such as SW, KEY,
// LEDR, or HEX.
`timescale 1 ps / 1 ps
module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, SW, LEDR, V_GPIO);
   input  logic CLOCK_50;
   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   input  logic [3:0] KEY;
   input  logic [9:0] SW;
   output logic [9:0] LEDR;
   inout  logic [35:23] V_GPIO;

   // Generate clk off of CLOCK_50, whichClock picks rate.
   logic [31:0] div_clk;

   parameter whichClock = 24.25; // 0.75 Hz clock // soft answer
   clock_divider cdiv (.clock(CLOCK_50),
                       .reset(reset),
                       .divided_clocks(div_clk));

   // Clock selection; allows for easy switching between simulation and board
   // clocks
   logic clkSelect;
   // Uncomment ONE of the following two lines depending on intention

   //assign clkSelect = CLOCK_50;          // for simulation
   assign clkSelect = div_clk[whichClock]; // for board

   // FPGA output
   assign V_GPIO[26] = V_GPIO[28];  // LED parking 1
   assign V_GPIO[27] = V_GPIO[29];  // LED parking 2
   assign V_GPIO[32] = V_GPIO[30];  // LED parking 3
   assign V_GPIO[34] = V_GPIO[28] & V_GPIO[29] & V_GPIO[30];   // LED full
   assign V_GPIO[31] = V_GPIO[23] & ~V_GPIO[34]; // SW[4];  // Open entrance // tell me when to enter
   assign V_GPIO[33] = V_GPIO[24]; // SW[5]; // Open exit // tell me when to exit
   logic [3:0] current_hour;
   logic [3:0] rush_start, rush_end;
   logic [6:0] rush_start_hex, rush_end_hex;
   logic [6:0] final_rush_start_hex, final_rush_end_hex ;
   logic full_final;
   logic outDD;
   logic outDD1, no_meta_dd1;
   logic no_meta_dd;
   logic [2:0] num_spots_avail;
   logic [6:0] num_spots_avail_hex;
   logic en1;
   logic [2:0] rd_addr_count;
   logic [15:0] rd_data;
   logic [6:0] rd_data_hex, rd_addr_hex, current_hour_hex;
   logic [6:0] HEX0_temp, HEX1_temp, HEX2_temp, HEX3_temp;
   logic outDD11, no_meta_dd11;

   // assigns the “en1” signal (which is essentially an enable signal) to when “current_hour” equals
   // 8, signaling that the day is over.
   assign en1 = (current_hour == 8);

   // module ‘flipFlop’, named ‘DD’ is used as a flip flop for our input of KEY[0] to ensure that
   // the output will be the same after a few clock cycles to prevent metastability
   flipFlop DD (.clk(CLOCK_50), .reset(SW[9]), .in(~KEY[0]), .out(outDD));

   // module ‘user_input’, named ‘no_meta’ is used to ensure that our KEY[0] input will only
   // produce an output for one clock cycle. this is to prevent any unnecessary increments in
   // current hour of the day
   user_input no_meta(.clock(CLOCK_50), .reset(SW[9]), .in(outDD), .out(no_meta_dd));

   // module ‘flipFlop’, named ‘DD_new’ is used as a flip flop for our V_GPIO[31] input that
   // indicates when a car is waiting to enter the car. it is to ensure that the output will be the same
   // after a few clock cycles to prevent metastability.
   flipFlop DD_new (.clk(CLOCK_50), .reset(SW[9]), .in(V_GPIO[31]), .out(outDD1));
```

```

   // module ‘user_input’, named ‘no_meta_new’ is used to ensure that our V_GPIO[31] input will
   // only produce an output for one clock cycle. this is to prevent any unnecessary increments in
   // the total car count
   user_input no_meta_new (.clock(CLOCK_50), .reset(SW[9]), .in(outDD1), .out(no_meta_dd1));

   // module ‘flipFlop’, named ‘DD_newer’ is used as a flip flop for our V_GPIO[33] input that
   // indicates when a car is waiting to enter the car. it is to ensure that the output will be the same
   // after a few clock cycles to prevent metastability.

   flipFlop DD_newer (.clk(CLOCK_50), .reset(SW[9]), .in(V_GPIO[33]), .out(outDD11));

   // module ‘user_input’, named ‘no_meta_newer’ is used to ensure that our V_GPIO[33] input will
   // only produce an output for one clock cycle. this is to prevent any unnecessary increments in
   // the total car count
   user_input no_meta_newer (.clock(CLOCK_50), .reset(SW[9]), .in(outDD11), .out(no_meta_dd11));

   // module ‘rushhr’, named ‘rush_hour’ is used to implement our rush hour algorithm. There is a
   // controller and datapath component as well as other submodules to help implement the
   // algorithm.  Rush hour starts when the parking lot first becomes full and it ends when the
   // parking lot first becomes empty after it becomes full. If there exists a rush hour, outputs that
   // represent the starting and ending hour of rush hour will be output as a number from 0-7. If
   // there does not exist a rush hour, the outputs will be the number 8 which signifies that a rush
   // hour never happened. Additionally, this module is responsible for counting the total number of
   // cars that have entered the parking lot throughout the day.
   rushhr rush_hour (.clk(CLOCK_50), .reset(SW[9]), .in(no_meta_dd1), .out(no_meta_dd11),
                     .end_hour(rush_end), .start_hour(rush_start), .start(SW[8]),
                     .key0(no_meta_dd), .full_final, .cur_hour(current_hour), .rd_addr(rd_addr_count), .q(rd_data));

   // module ‘count_spots’, named ‘count_it_pls’ is used to keep track of the remaining number of
   // parking spaces in the lot. the number of remaining parking spaces is equal to the number of
   // occupied parking spaces subtracted from 3. a parking space is occupied if it detects a car in a
   // parking space (thanks to the V_GPIOs)
   count_spots Count_it_pls(.car1(V_GPIO[26]), .car2(V_GPIO[27]), .car3(V_GPIO[32]), .spot_left(num_spots_avail));

   // module ‘counter’, named ‘coun’ is used to cycle through the hours of the day in an orderly
   // manner once the day is over (aka after the 8th hour). When the day is over, an enable signal
   // will be sent in to let the module know it’s okay to cycle through the hours. The cycling of the
   // hours are needed for display on HEX2 after the day is completed. The hours cycling is also
   // used for write/read addresses for our RAM.
   counter coun(.clk(clkSelect), .reset(SW[9]), .en(en1), .out(rd_addr_count));

   // module ‘HEX_RAM’, named ‘hex5’ is used to display the data needed for HEX5 during the
   // parking lot simulation. HEX5 displays the current hour of the day so it outputs a 7-segment
   // display of a number from 0-7 depending on the current hour.
   HEX_RAM hex5(.in(current_hour), .out(current_hour_hex));

   // module ‘HEX_RAM’, named ‘hex4’ is used to display the data needed for HEX4 during the
   // parking lot simulation. HEX4 displays the starting hour of rush hour so it outputs a 7-segment
   // display of a number from 0-7 depending on the starting hour or a dash if there’s no
```

```
rush hour.
   HEX_RAM hex4(.in(rush_start), .out(rush_start_hex));

   // module ‘HEX_RAM’, named ‘hex3’ is used to display the data needed for HEX3 during the
   // parking lot simulation. HEX3 displays the ending hour of rush hour so it outputs a 7-segment
   // display of a number from 0-7 depending on the ending hour or a dash if there’s no rush hour.
   HEX_RAM hex3(.in(rush_end), .out(rush_end_hex));

   // module ‘HEX_RAM’, named ‘hex0’ is used to display the data needed for HEX0 during the
   // parking lot simulation. HEX0 displays the remaining number of parking spaces so it outputs a
   // 7-segment display of a number from 1-3 depending on the remaining number of parking
   // spaces.
   HEX_RAM hex0(.in(num_spots_avail), .out(num_spots_avail_hex));

   // module ‘HEX_RAM’, named ‘hex1’ is used to display the data needed for HEX1 during the
   // parking lot simulation. HEX1 displays and cycles in intervals of 1 second through the total
   // number of cars that have entered the parking lot by a certain hour of the day so it outputs a
   // 7-segment display of a hexadecimal number depending on the total number of cars entered at
   // that hour.
   HEX_RAM hex1(.in(rd_data), .out(rd_data_hex));

   // module ‘HEX_RAM’, named ‘hex2’ is used to display the data needed for HEX2 during the
   // parking lot simulation. HEX2 displays and cycles in intervals of 1 second through the hours of
   // the day so it outputs a 7-segment display of a number from 0-7 depending on the hour of the
   // day.
   HEX_RAM hex2(.in(rd_addr_count), .out(rd_addr_hex));

   // Logic to decide whether rush hour started or ended, or not.
   // If rush hour started and ended, HEX 4 and 3 will output
   // the HEX representation of the hour that rush hour started and
   // the hour that rush hour ended.
   // Else, HEX 4 and 3 will output a '-' to demontstrate
   // that rush hour was not successfully completed.
   always_comb begin
       if ((rush_end == 8) && (rush_start == 8)) begin
           final_rush_start_hex = 7'b0111111; // '-'
           final_rush_end_hex = 7'b0111111;// '-'
       end else begin
           final_rush_start_hex = rush_start_hex;
           final_rush_end_hex = rush_end_hex;
       end
   end

   // logic before hour 8,
   // if the parking lot is full, HEX3 - 0
   // will spell out 'FULL'.
   // Else, HEX0 will show how many spots are availble in
   // the parking lot.
   always_comb begin
      if (V_GPIO[34]) begin
         //full state here
         HEX3_temp = 7'b0001110; // F
         HEX2_temp = 7'b1000001; // U
         HEX1_temp = 7'b1000111; // L
         HEX0_temp = 7'b1000111; // L
      end else begin
         HEX3_temp = '1;
         HEX2_temp = '1;
         HEX1_temp = '1;
         HEX0_temp = num_spots_avail_hex;
      end
   end

   // HEX operations before and after hour 8.
   // Before hour 8:
   // HEX 5: show current hour count.
```

```
    // HEX 4: blank.
    // HEX 3 - 0: functions are outlined in the above always_comb block.
    //
    // After hour 8:
    // HEX 5: blank.
    // HEX 4: show whether or not we started rush hour.
    // HEX 3: show whether or not we ended rush hour.
    // HEX 2: show the current hour that we cycling through.
    //        RANGE: 0 -> 7.
    // HEX 1: show the number of cars that entered the parking
    //        lot during certain hours.
    // HEX 0: blank.
    always_comb begin
       if (~(current_hour == 8)) begin
          HEX5 = current_hour_hex;
          HEX4 = '1;
          HEX3 = HEX3_temp;
          HEX2 = HEX2_temp;
          HEX1 = HEX1_temp;
          HEX0 = HEX0_temp;
       end else begin
          HEX5 = '1;
          HEX4 = final_rush_start_hex;
          HEX3 = final_rush_end_hex;
          HEX2 = rd_addr_hex;
          HEX1 = rd_data_hex;
          HEX0 = '1;
       end

    end


    // FPGA input
    assign LEDR[0] = V_GPIO[28];  // Presence parking 1
    assign LEDR[1] = V_GPIO[29];  // Presence parking 2
    assign LEDR[2] = V_GPIO[30];  // Presence parking 3
    assign LEDR[3] = V_GPIO[23];  // Presence entrance
    assign LEDR[4] = V_GPIO[24];  // Presence exit
    assign LEDR[5] = V_GPIO[34];  // full

 endmodule  // DE1_SoC

 // This test bench is designed specifically to examine how the top module 'DE1_SoC'
 // Interfaces with the submodule fucntions. Essentially, we want to see what would happen
 // when we incorporate the use of switches and keys from DE1_SoC.
 // We are testing 2 different cases in this test bench.
 // 1) The start and end of rush hour
 // 2) rush hour doesn't start
 // We are making sure that the SWs and KEYs we use from the FPGA will work accordingly
 // with the code that we wrote the for the submodules that controls the functionality of
 // rush hour and car count.
 module DE1_SoC_testbench();
    logic CLOCK_50;
    logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    logic [3:0] KEY;
    logic [9:0] SW;
    logic [9:0] LEDR;
    wire [35:23] V_GPIO;

    // use SWs to simulate the signals form V_GPIO wires
    // The functions of V_GPIO pins are outlined above
    assign V_GPIO[28] = SW[7];
    assign V_GPIO[29] = SW[6];
    assign V_GPIO[30] = SW[5];
    assign V_GPIO[24] = SW[4];
    assign V_GPIO[23] = SW[3];

    DE1_SoC dut1 (.*);

    parameter clock_period = 100;
    initial begin
       CLOCK_50 <= 0;
       forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;

```

```
   end

   initial begin
      // Initialize the inputs
      SW[9] <= 1; SW[8] <= 0; KEY[0] <= 1; @(posedge CLOCK_50);
      SW[9] <= 0; SW[8] <= 1; @(posedge CLOCK_50);
      // Start of rush hour
      SW[8] <= 0;  @(posedge CLOCK_50);
      // Increment rush hour
      // Increment car count here
      SW[7] <= 1; SW[3] <= 1; @(posedge CLOCK_50);
      KEY[0] <= 0; @(posedge CLOCK_50);
      KEY[0] <= 1; @(posedge CLOCK_50);
      SW[6] <= 1;  SW[3]  <= 1; @(posedge CLOCK_50);
      KEY[0] <= 0; @(posedge CLOCK_50);
      KEY[0] <= 1; @(posedge CLOCK_50);
      // rush hour starts after this point
      SW[5] <= 1;  SW[3]  <= 1; @(posedge CLOCK_50);
      KEY[0] <= 0; @(posedge CLOCK_50);
      KEY[0] <= 1; @(posedge CLOCK_50);
      // start ending rush hour here
      SW[7] <= 0;  SW[4] <= 1; @(posedge CLOCK_50);
      KEY[0] <= 0; @(posedge CLOCK_50);
      KEY[0] <= 1; @(posedge CLOCK_50);
      SW[6] <= 0;  SW[4] <= 1; @(posedge CLOCK_50);
      KEY[0] <= 0; @(posedge CLOCK_50);
      KEY[0] <= 1; @(posedge CLOCK_50);
      // rush hour ends here
      SW[5] <= 0; SW[4] <= 1;  @(posedge CLOCK_50);
      SW[4] <= 0; @(posedge CLOCK_50);
      KEY[0] <= 0; @(posedge CLOCK_50);
      KEY[0] <= 1; @(posedge CLOCK_50);
      KEY[0] <= 0; @(posedge CLOCK_50);
      KEY[0] <= 1; @(posedge CLOCK_50);
      KEY[0] <= 0; @(posedge CLOCK_50);
      KEY[0] <= 1; @(posedge CLOCK_50);
      // Opereation day ends here
      for (int i = 0; i < 10; i++) begin
         @(posedge CLOCK_50);
      end
      // Another case,
      // Testing what happens when we don't
      // finish rush hour
      SW[9] <= 1; SW[8] <= 0;  @(posedge CLOCK_50);
      SW[9] <= 0; @(posedge CLOCK_50);
      SW[8] <= 1;  @(posedge CLOCK_50);
      SW[8] <= 0;  @(posedge CLOCK_50);
      // Increment hour to the end without
      // rush hour functionality.
      for (int i = 0; i < 8; i++) begin
         KEY[0] <= 0; @(posedge CLOCK_50);
         KEY[0] <= 1; @(posedge CLOCK_50);
      end
      for (int i = 0; i < 10; i++) begin
         @(posedge CLOCK_50);
      end
      $stop;
   end


endmodule
```