Connor Aksama
EE 371
March 13, 2023
Lab 6 Report

Procedure

This lab is comprised of two tasks. The first task involved repurposing code from Lab 1 to interface with a different GPIO interface. The second task consisted of implementing a parking lot tracker for a 3D simulator of a parking lot.

Task 1

I approached this task by first updating the name of the GPIO interface to V_GPIO and changing the port assignments for the logical controls in my Lab 1 top-level module.

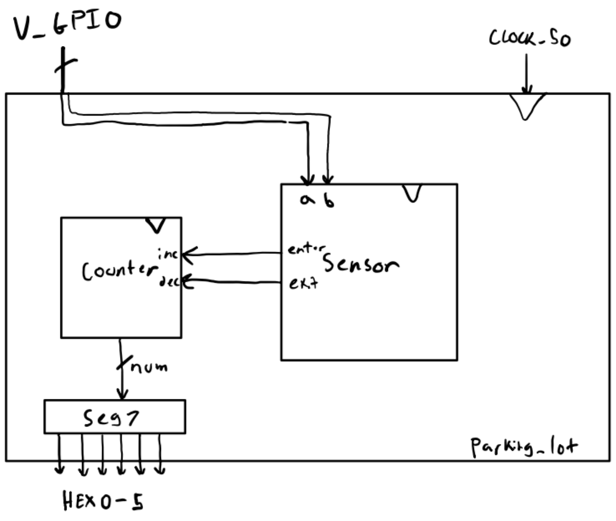


Figure 1. Top-Level Block Diagram for Lab 6 Task 1.

Task 2

I approached this task by first designing the ASMD for the rush hour tracker portion of the task. I determined the necessary states for the system, what control signals would be necessary for the data path, and defined how the output would be reported.

Next, I implemented the car tracking portion of the task, which consisted of instantiating a RAM module, which would read and write data about the total number of cars that have entered at each hour.

After testing both of these modules, I created the top-level module that would handle the I/O connections and the logic for what information would be displayed on the HEX displays as time progressed.

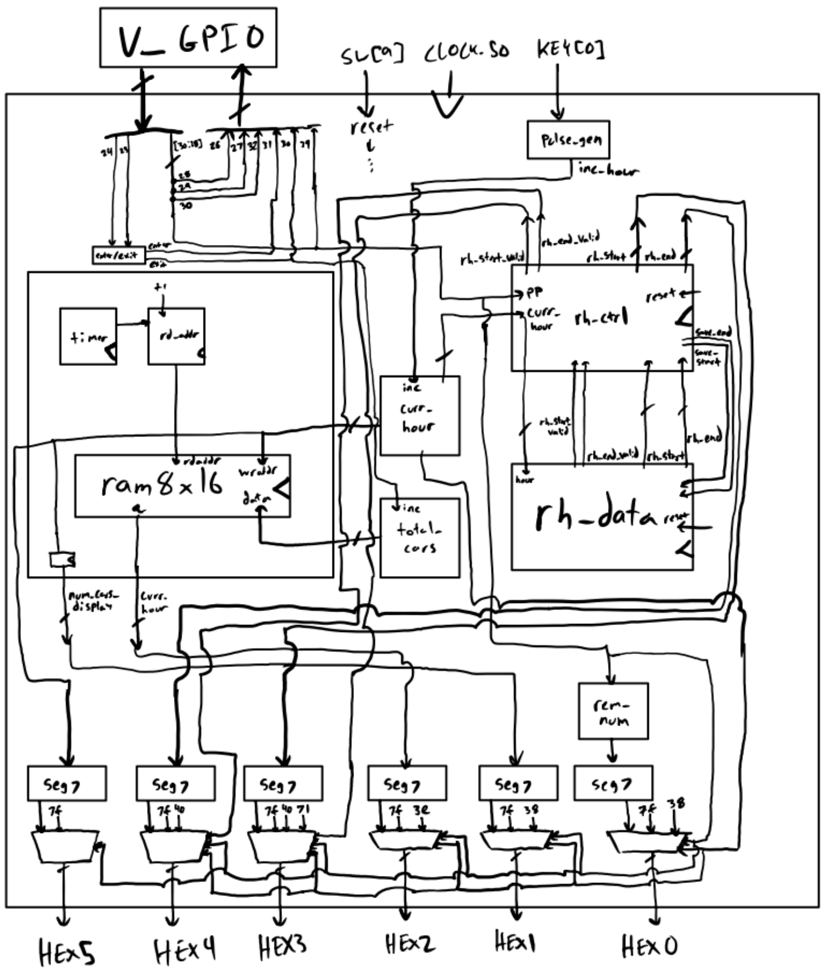


Figure 2. Top-Level Block Diagram for Lab 6 Task 2.

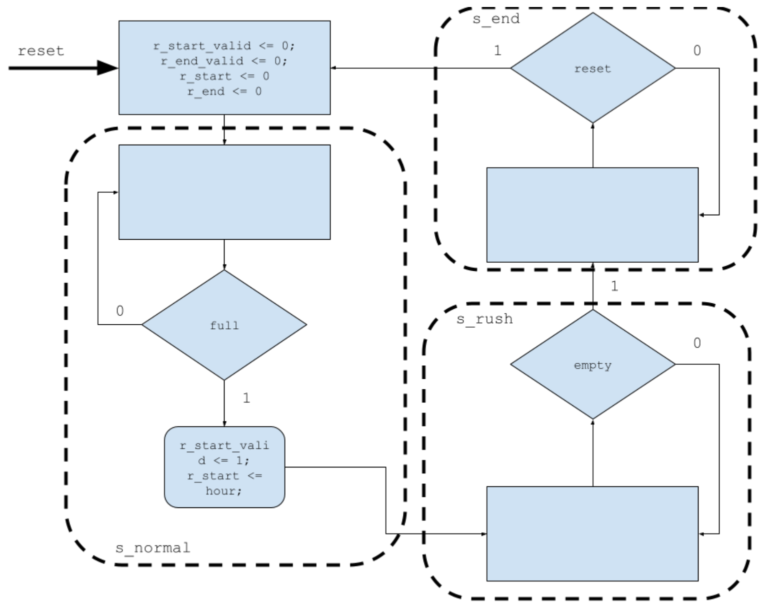


Figure 3. ASMD Chart for the Rush Hour Tracking Module

Results

Task 1

No simulations were required for Task 1 of this lab.

Task 2

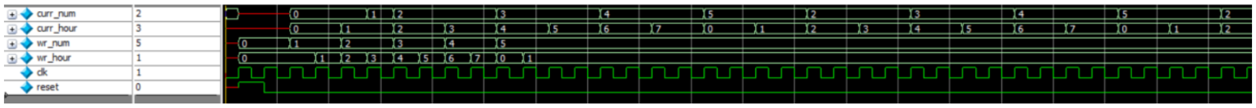
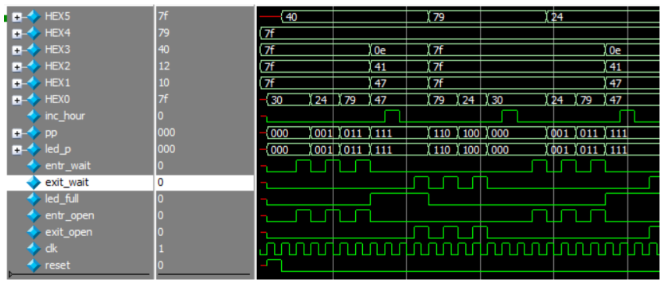


Figure 4. ModelSim Waveforms for the Car Tracking module.

This simulation shows different numbers of cars being written at different hours into the tracking module. Then after the first 8 hours, the simulation shows the cycling output, displaying the record of the total number of cars at each hour.

ModelSim Hexadecimal Code	Seven Segment Display Character
40	0
79	1
24	2
30	3
19	4
12	5
02	6
78	7
00	8
10	9
08	A
03	b
27	c
21	d
06	E
0e	F

Table 1. Conversion between hexadecimal values shown in ModelSim and the character shown on a seven segment display.



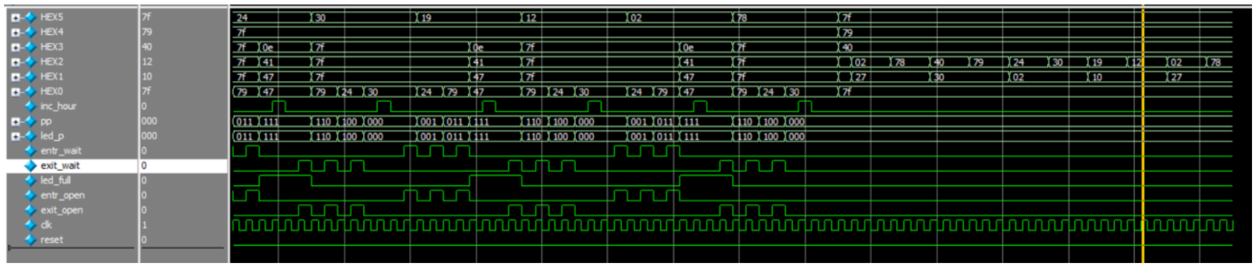


Figure 5. ModelSim Waveforms for the Task 2 top-level module.

This simulation demonstrates filling the parking lot with 3 cars one hour, then removing them the next hour, and repeating until the end of the workday. The simulation demonstrates that the correct numbers are displayed showing the hour and remaining spots, as well as the appropriate messages and external LEDs.

At the end of the 8th hour, the simulation then demonstrates displaying the start and end of rush hour, as well as the number of cars at each hour.

See Table 1 for translations between input numbers and hexadecimal values for the HEX output.

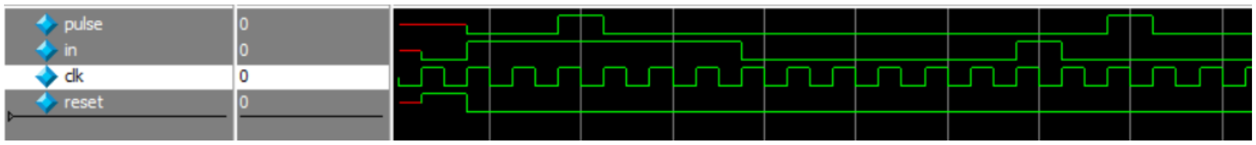


Figure 6. ModelSim Waveform for the Pulse Generator module.

This simluation demonstrates that a one cycle length signal is generated for an arbitrary length continuous input.

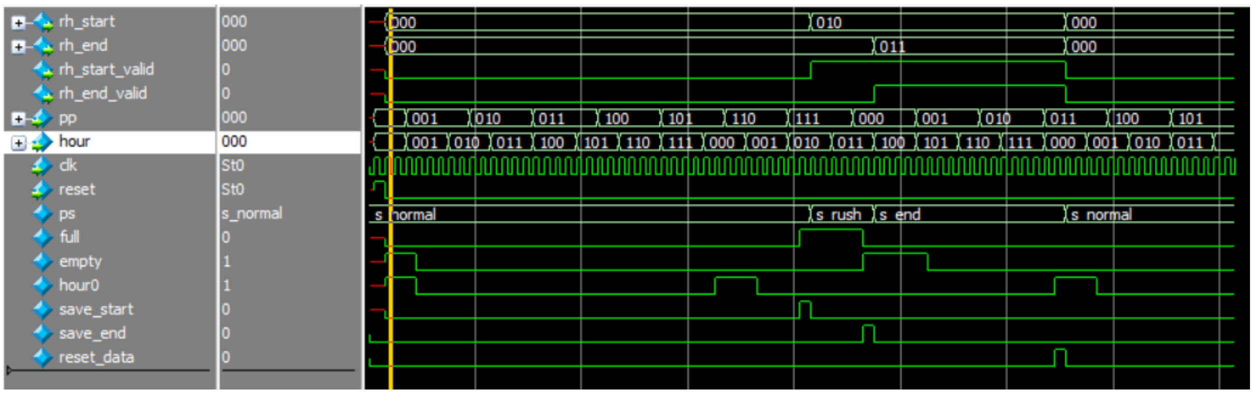


Figure 7. ModelSim Waveform for the Rush Hour Control Path module.

This simulation demonstrates a work day with no rush hour, and a work day with a start and end of a rush hour. The results show that the correct control signals to save the start/end hour are generated, and that the start and end hours are output only if they are set valid.

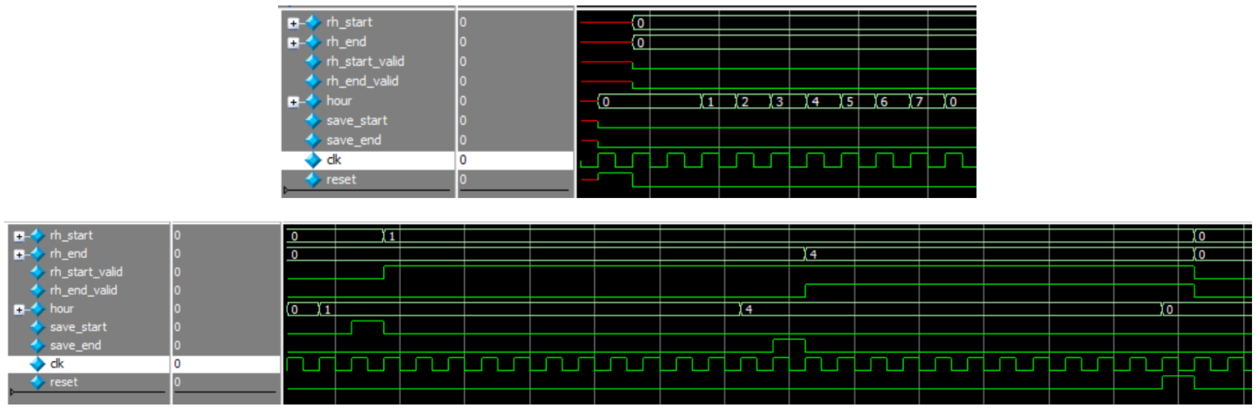


Figure 8. ModelSim Waveform for the Rush Hour Datapath module.

This simulation demonstrates that the start and ending hours for rush hour are only saved by the data path when the corresponding control signals are received by the module.

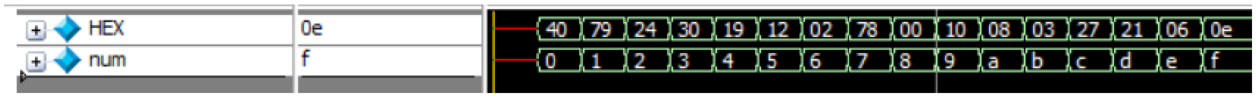

HEX
num
40 79 24 30 19 12 02 78 00 10 08 03 27 21 06 0e
0 1 2 3 4 5 6 7 8 9 a b c d e f

Figure 9. ModelSim Waveform for the single seven segment display module.

This module shows that the correct HEX patterns are generated for corresponding 4-bit inputs. See Table 1 for translations between input numbers and hexadecimal values for the HEX output.

Final Product

The overarching goal of this lab was to gain experience interfacing with a different GPIO interface from previous labs. In task 1, the same product from Lab 1 was produced, except now being able to interface with the V_GPIO bus. In task 2, I implemented a parking lot simulator/tracker that interfaced with a virtual 3D parking lot simulator. This system was able to control the virtual LEDs based on the detected positions and number of cars, allow them to enter and exit, and display different information about the parking lot using the board HEX displays. The system contains the ability to increment the current work hour, keep track of the number of cars that have entered in each hour, control the enter and exit gates, and keep track of when a rush hour starts and end if applicable. At the end of the workday, the system displays the number of cars that have entered each hour on the HEX display along with the detected start and end of rush hour if applicable.

Appendix: SystemVerilog Code

(See next page)

```systemverilog
/**********************************

car_tracking.sv

**********************************/

/*
 * Connor Aksama
 * 03/13/2023
 * CSE 371
 * Lab 6
*/

/**
 * Controller module for the car tracking system.

 * Inputs:
 *      wr_num [16 bit] - The number of cars to track for the current hour
 *      wr_hour [3 bit] - The hour for which to update the number of cars
 *      clk [1 bit] - The clock to use for this module.
 *      reset [1 bit] - Resets the cyclic display to hour 0.

 * Outputs:
 *      curr_num [16 bit] - The number of cars that have entered in the corresponding hour
 *      curr_hour [3 bit] - The hour at which the corresponding number of cars have entered
*/
module car_tracking #(
        parameter clk_freq = 50000000, duration_sec = 1
    )(
        output logic [15:0] curr_num
        ,output logic [2:0] curr_hour
        ,input logic [15:0] wr_num
        ,input logic [2:0] wr_hour
        ,input logic clk, reset
    );

    localparam timer_target = clk_freq * duration_sec;

    logic [2:0] rd_addr, rd_out1;
    logic [31:0] timer;
    logic reset_timer;

    // RAM module for car tracking data
    // clk -> clock
    // wr_num -> data
    // rd_addr (current hour to read) -> rdaddress
    // wr_hour (current hour to modify) -> wraddress
    // wren -> always write
    // q -> curr_num (current num of cars)
    ram8x16 ram (
        .clock(clk)
        ,.data(wr_num)
        ,.rdaddress(rd_addr)
        ,.wraddress(wr_hour)
        ,.wren('1)
        ,.q(curr_num)
    );

    always_comb begin
        // Reset timer at target
        reset_timer = (timer >= timer_target - 1);
    end

    always_ff @(posedge clk) begin
        if (reset) begin
            // Reset cycle to beginning
            rd_addr <= '0;
```

```
                rd_out1 <= '0;
                timer <= '0;
            end else if (reset_timer) begin
                // Reset timer, move to next hour
                timer <= '0;
                rd_addr <= rd_addr + 3'd1;
            end else begin
                // Increment timer
                timer <= timer + 32'd1;
            end

            // Sync with RAM output
            curr_hour <= rd_addr;
        end

endmodule  // car_tracking

/*
 * Testbench to test the functionality of the car_tracking module
 */
`timescale 1 ps / 1 ps
module car_tracking_testbench();

    logic [15:0] curr_num;
    logic [2:0] curr_hour;

    logic [15:0] wr_num;
    logic [2:0] wr_hour;
    logic clk, reset;

    car_tracking #(.clk_freq(2), .duration_sec(1)) dut (.*);

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

    initial begin
        integer i;

        @(posedge clk) reset <= '1; wr_num <= '0; wr_hour <= '0;
        @(posedge clk) reset <= '0;

        // Write data to RAM
        for (i = 0; i < 10; i++) begin
            @(posedge clk) wr_hour <= i;
            if (i % 2 == 0) begin
                wr_num <= wr_num + 15'd1;
            end
        end

        // Cycle through values
        for (i = 0; i < 30; i++) begin
            @(posedge clk);
        end


        $stop;
    end

endmodule  // car_tracking_testbench

/***********************************

DE1_SoC.sv

***********************************/

```

```systemverilog
/*
 * Connor Aksama
 * 03/13/2023
 * CSE 371
 * Lab 6
*/

/**
 * Top-level module for Lab 6 Task 2. Instantiates rush hour and car tracking modules
 and defines logic to connect I/O to peripherals.

 * Inputs:
 *  SW [10 bit] - The 10 onboard switches, respectively.
 *  KEY [4 bit] - The 4 onboard keys, respectively.
 *  CLOCK_50 [1 bit] - The system clock to use for this module.

 * Outputs:
 *  HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
 display format.
 *  HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
 display format.
 *  HEX2 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
 display format.
 *  HEX3 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
 display format.
 *  HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
 display format.
 *  HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
 display format.

 * Inouts:
 *  V_GPIO [13 bit] - Virtual GPIO Ports
*/
module DE1_SoC #(
        parameter clk_freq = 50000000, display_duration_sec = 1
    )(
        output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
        ,input logic [9:0] SW
        ,input logic [3:0] KEY
        ,input logic CLOCK_50
        ,inout logic [35:23] V_GPIO
    );

    logic clk;
    assign clk = CLOCK_50;

    // Which parking spaces occupied?
    logic [2:0] pp;
    assign pp = V_GPIO[30:28];

    // Car waiting at gate
    logic entr_wait, exit_wait;
    assign entr_wait = V_GPIO[23];
    assign exit_wait = V_GPIO[24];

    // Light LEDs at each parking spot depending on presence
    assign {V_GPIO[32], V_GPIO[27], V_GPIO[26]} = pp;

    // Full LED
    logic led_full;
    assign V_GPIO[34] = led_full;

    // Ctrl for gates
    logic entr_open, exit_open;
    assign V_GPIO[31] = entr_open;
    assign V_GPIO[33] = exit_open;

    // Remaining spot HEX generator
    logic [1:0] rem_num;
	logic [6:0] rem_hex;
	seg7 rem_spot (.HEX(rem_hex), .num({2'b0, rem_num}));

	// Current hour HEX generator
	logic [2:0] curr_hour;
	logic [6:0] ch_hex;
	seg7 hour (.HEX(ch_hex), .num({1'b0, curr_hour}));

	logic reset;
	assign reset = SW[9];

	logic done;

	// Rush Hour Tracker
	// rh_start -> rh_start (saved rush hour start)
	// rh_end -> rh_end (saved rush hour end)
	// rh_start_valid -> rh_start_valid (1 if rh_start data valid)
	// rh_end_valid -> rh_end_valid (1 if rh_end data valid)
	// pp -> pp
	// curr_hour -> hour (current hour)
	// clk -> clk
	// reset -> reset
	logic [2:0] rh_start, rh_end;
	logic rh_start_valid, rh_end_valid;
	rh_ctrl rush_hour (
	    .rh_start(rh_start)
	    ,.rh_end(rh_end)
	    ,.rh_start_valid(rh_start_valid)
	    ,.rh_end_valid(rh_end_valid)
	    ,.pp(pp)
	    ,.hour(curr_hour)
	    ,.clk(clk)
	    ,.reset(reset)
	);

	logic [6:0] end_hex, start_hex;
	// Rush Hour End HEX generator
	seg7 end_hour (.HEX(end_hex), .num({1'b0, rh_end}));
	// Rush Hour Start HEX generator
	seg7 start_hour (.HEX(start_hex), .num({1'b0, rh_start}));

	// Num Car Tracker/Display
	// curr_num -> num_cars_display (Output for saved # of cars for curr_hour_display)
	// curr_hour -> curr_hour_display (Current hour for which to display # of cars)
	// total_cars -> wr_num (write the total number of cars entered so far)
	// curr_hour -> wr_hour (current hour, write total num cars for this hour)
	// clk -> clk
	// reset -> reset
	logic [15:0] total_cars, num_cars_display;
	logic [2:0] curr_hour_display;
	car_tracking #(
	    .clk_freq(clk_freq), .duration_sec(display_duration_sec)
	) results (
	    .curr_num(num_cars_display)
	    ,.curr_hour(curr_hour_display)
	    ,.wr_num(total_cars)
	    ,.wr_hour(curr_hour)
	    ,.clk(clk)
	    ,.reset(reset)
	);

	logic [6:0] ncd_hex, chd_hex;
	// EOD num cars HEX generator
	seg7 ncd (.HEX(ncd_hex), .num(num_cars_display[3:0]));
	// EOD curr hour HEX generator
	seg7 chd (.HEX(chd_hex), .num({1'b0, curr_hour_display}));

	logic inc_hour;
	// Generate a one cycle pulse for the inc_hour signal (KEY[0])
```

```
    pulse_gen gen_hour (.pulse(inc_hour), .in(~KEY[0]), .clk(clk), .reset(reset));

    logic inc_cars;
    pulse_gen gen_cars (.pulse(inc_cars), .in(entr_open), .clk(clk), .reset(reset));

    always_comb begin
        // Ctrl signals
        led_full = (pp == 3'b111);
        entr_open = entr_wait & (pp != 3'b111);
        exit_open = exit_wait;

        // Find remaining number of spots
        case (pp)
            3'b000:
                rem_num = 2'd3;
            3'b001, 3'b010, 3'b100:
                rem_num = 2'd2;
            3'b011, 3'b101, 3'b110:
                rem_num = 2'd1;
            3'b111:
                rem_num = 2'd0;
        endcase

        // ctrl driver for HEX outputs
        if (done) begin
            HEX5 = ~7'b0;
            if (rh_end_valid) begin
                HEX4 = end_hex;
            end else begin
                HEX4 = ~7'b1000000;
            end
            if (rh_start_valid) begin
                HEX3 = start_hex;
            end else begin
                HEX3 = ~7'b1000000;
            end
            HEX2 = chd_hex;
            HEX1 = ncd_hex;
            HEX0 = ~7'b0;
        end else if (led_full) begin
            HEX5 = ch_hex;
            HEX4 = ~7'b0;
            HEX3 = ~7'b1110001;  // F
            HEX2 = ~7'b0111110;  // U
            HEX1 = ~7'b0111000;  // L
            HEX0 = ~7'b0111000;  // L
        end else begin
            HEX5 = ch_hex;
            HEX4 = ~7'b0;
            HEX3 = ~7'b0;
            HEX2 = ~7'b0;
            HEX1 = ~7'b0;
            HEX0 = rem_hex;
        end
    end

    always_ff @(posedge clk) begin
        if (reset) begin
            // Reset work day
            curr_hour <= '0;
            done <= '0;
        end else if (inc_hour && curr_hour == 3'd7) begin
            // EOD reached
            done <= '1;
        end else if (inc_hour) begin
            // Increase work hour
            curr_hour <= curr_hour + 3'b001;
        end

        if (reset) begin
            // Reset # total cars
            total_cars <= '0;
        end else if (inc_cars) begin
            // Car entering
            total_cars <= total_cars + 16'b1;
        end
    end

endmodule  // DE1_SoC

/*
 * Testbench to test the functionality of the DE1_SoC module
 */
`timescale 1 ps / 1 ps
module DE1_SoC_testbench();

    logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
    logic [9:0] SW;
    logic [3:0] KEY;
    logic CLOCK_50;
    wire [35:23] V_GPIO;

    logic clk;
    assign CLOCK_50 = clk;

    logic reset;
    assign SW[9] = reset;

    logic inc_hour;
    assign KEY[0] = ~inc_hour;

    logic [2:0] pp;
    assign V_GPIO[30:28] = pp;

    logic entr_wait, exit_wait;
    assign V_GPIO[23] = entr_wait;
    assign V_GPIO[24] = exit_wait;

    logic [2:0] led_p;
    assign led_p = {V_GPIO[32], V_GPIO[27], V_GPIO[26]};

    logic led_full;
    assign led_full = V_GPIO[34];

    logic entr_open, exit_open;
    assign entr_open = V_GPIO[31];
    assign exit_open = V_GPIO[33];

    DE1_SoC #(.clk_freq(3), .display_duration_sec(1)) dut (.*);

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

    initial begin
        integer i, j;

        @(posedge clk) reset <= '1; inc_hour <= '0; pp <= '0; entr_wait <= '0; exit_wait
        <= '0;
        @(posedge clk) reset <= '0;

        for (i = 0; i < 8; i++) begin
            if (i % 2 == 0) begin
                // Three cars enter
                for (j = 0; j < 3; j++) begin
                    @(posedge clk) entr_wait <= '1;
                    @(posedge clk);
                    if (entr_open) begin
                        entr_wait <= '0;
                        pp[j] <= '1;
                    end else begin
                        j--;
                    end
                end
            end else begin
                // Three cars leave
                for (j = 0; j < 3; j++) begin
                    @(posedge clk) exit_wait <= '1;
                    @(posedge clk);
                    if (exit_open) begin
                        exit_wait <= '0;
                        pp[j] <= '0;
                    end else begin
                        j--;
                    end
                end
            end
            @(posedge clk) inc_hour <= '1;
            @(posedge clk) inc_hour <= '0;
        end

        for (i = 0; i < 32; i++) begin
            @(posedge clk);
        end

        @(posedge clk) reset <= '1; inc_hour <= '0; pp <= '0; entr_wait <= '0; exit_wait <= '0;
        @(posedge clk) reset <= '0;
        for (i = 0; i < 3; i++) begin
            @(posedge clk) entr_wait <= '1;
            @(posedge clk);
            if (entr_open) begin
                entr_wait <= '0;
                pp[j] <= '1;
            end
        end
        for (i = 0; i < 8; i++) begin
            @(posedge clk) inc_hour <= '1;
            @(posedge clk) inc_hour <= '0;
        end
        for (i = 0; i < 32; i++) begin
            @(posedge clk);
        end
        $stop;
    end

endmodule  // DE1_SoC_testbench

/************************************

pulse_gen.sv

************************************/

/*
 * Connor Aksama
 * 03/13/2023
 * CSE 371
 * Lab 6
*/

/**
 * One cycle pulse generator for arbitrary length input.

 * Inputs:
```

```
 *      in [1 bit] - Input for which to generate pulse
 *      clk [1 bit] - The clock to use for this module.
 *      reset [1 bit] - Resets the cyclic display to hour 0.

 * Outputs:
 *      pulse [1 bit] - The output signal where pulse is sent
*/
module pulse_gen (
        output logic pulse
        ,input logic in, clk, reset
    );

    typedef enum logic [1:0] { s_wait_rise, s_pulse, s_wait_fall } state;

    state ps, ns;
    logic in_reg;

    always_comb begin
        ns = ps;

        // Handle state transitions
        case (ps)
            s_wait_rise: begin
                if (in_reg) begin
                    ns = s_pulse;
                end
            end
            s_pulse: begin
                if (~in_reg) begin
                    ns = s_wait_rise;
                end else begin
                    ns = s_wait_fall;
                end
            end
            s_wait_fall: begin
                if (~in_reg) begin
                    ns = s_wait_rise;
                end
            end
        endcase

        pulse = (ps == s_pulse);
    end

    always_ff @(posedge clk) begin
        // Update state
        if (reset) begin
            ps <= s_wait_rise;
        end else begin
            ps <= ns;
        end
        // Register input
        in_reg <= in;
    end

endmodule  // pulse_gen

/*
 * Testbench to test the functionality of the pulse_gen module
 */
module pulse_gen_testbench();

    logic pulse, in, clk, reset;

    pulse_gen dut (.*);

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

    initial begin
        integer i;

        @(posedge clk) reset <= '1; in <= '0;
        @(posedge clk) reset <= '0; in <= '1;

        for (i = 0; i < 5; i++) begin
            @(posedge clk);
        end

        @(posedge clk) in <= '0;

        for (i = 0; i < 5; i++) begin
            @(posedge clk);
        end

        @(posedge clk) in <= '1;
        @(posedge clk) in <= '0;

        for (i = 0; i < 5; i++) begin
            @(posedge clk);
        end

        $stop;
    end

endmodule  // pulse_gen_testbench

/***********************************

rh_ctrl.sv

***********************************/

/*
 * Connor Aksama
 * 03/13/2023
 * CSE 371
 * Lab 6
*/

/**
 * Controller module for the rush hour system.

 * Inputs:
 *      pp [3 bit] - Parking spaces currently occupied
 *      hour [3 bit] - The current hour
 *      clk [1 bit] - The clock to use for this module.
 *      reset [1 bit] - Resets the FSM to its initial state.

 * Outputs:
 *      rh_start [3 bit] - The saved start of rush hour
 *      rh_end [3 bit] - The saved end of rush hour
 *      rh_start_valid [1 bit] - 1 if rh_start is valid, 0 o.w.
 *      rh_end_valid [1 bit] - 1 if rh_end is valid, 0 o.w.
*/
module rh_ctrl (
        output logic [2:0] rh_start, rh_end
        ,output logic rh_start_valid, rh_end_valid
        ,input logic [2:0] pp, hour
        ,input logic clk, reset
    );


    typedef enum { s_normal, s_rush, s_end } state;

    state ps, ns;

    logic full, empty, hour0;
    logic save_start, save_end, reset_data;

    // Datapath module
    // rh_start -> rh_start (saved start hour)
    // rh_end -> rh_end (saved end hour)
    // rh_start_valid -> rh_start_valid (rh_start valid?)
    // rh_end_valid -> rh_end_valid (rh_end valid?)
    // hour -> hour (current hour of system)
    // save_start -> save_start (current hour is start)
    // save_end -> save_end (current hour is end)
    // clk -> clk
    // reset | reset_data -> reset (reset FSM to start)
    rh_data rush_hour_data (
        .rh_start(rh_start)
        ,.rh_end(rh_end)
        ,.rh_start_valid(rh_start_valid)
        ,.rh_end_valid(rh_end_valid)
        ,.hour(hour)
        ,.save_start(save_start)
        ,.save_end(save_end)
        ,.clk(clk)
        ,.reset(reset | reset_data)
    );

    always_comb begin

        // Handle state transitions
        case (ps)

            s_normal: begin
                if (full) begin
                    ns = s_rush;
                end else begin
                    ns = s_normal;
                end
            end

            s_rush: begin
                if (empty) begin
                    ns = s_end;
                end else begin
                    ns = s_rush;
                end
            end

            s_end: begin
                if (hour0) begin
                    ns = s_normal;
                end else begin
                    ns = s_end;
                end
            end

        endcase

        // Control signals
        save_start = (ps == s_normal) & full;
        save_end = (ps == s_rush) & empty;
        reset_data = (ps == s_end) & hour0;

    end


    always_ff @(posedge clk) begin
        // Update FSM
        if (reset) begin
```

```
                ps <= s_normal;
            end else begin
                ps <= ns;
            end

            // Register control signals
            full <= (pp == 3'b111);
            empty <= (pp == 3'b000);
            hour0 <= (hour == 3'b000);
        end


    endmodule  // rh_ctrl

    /*
     * Testbench to test the functionality of the rh_ctrl module
     */
    module rh_ctrl_testbench();

        logic [2:0] rh_start, rh_end;
        logic rh_start_valid, rh_end_valid;

        logic [2:0] pp, hour;
        logic clk, reset;

        rh_ctrl dut (.*);

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

        initial begin
            integer i;

            @(posedge clk) reset <= '1; pp <= '0; hour <= '0;
            @(posedge clk) reset <= '0;

            for (i = 0; i < 40; i++) begin
                @(posedge clk);
                @(posedge clk)
                if (i % 2 == 0) begin
                    // Increment hour
                    hour <= hour + 3'd1;
                end
                if (i % 3 == 0) begin
                    // Add car to lot
                    pp <= pp + 3'd1;
                end
            end

            $stop;
        end

    endmodule  // rh_ctrl_testbench

    /**********************************

    rh_data.sv

    **********************************/

    /*
     * Connor Aksama
     * 03/13/2023
     * CSE 371
     * Lab 6
    */
```

```

/**
 * Datapath module for the rush hour system.

 * Inputs:
 *      hour [3 bit] - The current hour
 *      save_start [1 bit] - 1 if hour should be saved for start, 0 o.w.
 *      save_end [1 bit] - 1 if hour should be saved for end, 0 o.w.
 *      clk [1 bit] - The clock to use for this module.
 *      reset [1 bit] - Resets the FSM to its initial state.

 * Outputs:
 *      rh_start [3 bit] - The saved start of rush hour
 *      rh_end [3 bit] - The saved end of rush hour
 *      rh_start_valid [1 bit] - 1 if rh_start is valid, 0 o.w.
 *      rh_end_valid [1 bit] - 1 if rh_end is valid, 0 o.w.
*/
module rh_data (
        output logic [2:0] rh_start, rh_end
        ,output logic rh_start_valid, rh_end_valid
        ,input logic [2:0] hour
        ,input logic save_start, save_end, clk, reset
    );


    always_ff @(posedge clk) begin
        // Register hour/valid bits based on ctrl signals
        if (reset) begin
            rh_start <= '0;
            rh_end <= '0;
            rh_start_valid <= '0;
            rh_end_valid <= '0;
        end else if (save_start) begin
            rh_start <= hour;
            rh_start_valid <= '1;
        end else if (save_end) begin
            rh_end <= hour;
            rh_end_valid <= '1;
        end


    end

endmodule  // rh_data

/*
 * Testbench to test the functionality of the rh_ctrl module
 */
module rh_data_testbench();

    logic [2:0] rh_start, rh_end;
    logic rh_start_valid, rh_end_valid;
    logic [2:0] hour;
    logic save_start, save_end, clk, reset;

    rh_data dut (.*);

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

    initial begin
        integer i;

        @(posedge clk) reset <= '1; save_start <= '0; save_end <= '0; hour <= '0;
        @(posedge clk) reset <= '0;

```

```
            // Do nothing
            for (i = 0; i < 10; i++) begin
                @(posedge clk);
                hour <= i;
            end

            // Save start hour
            @(posedge clk) save_start <= '1;
            @(posedge clk) save_start <= '0;

            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end

            // Save end hour
            @(posedge clk); hour <= hour + 3;
            @(posedge clk); save_end <= '1;
            @(posedge clk); save_end <= '0;

            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end

            @(posedge clk); hour <= '0; reset <= '1;
            @(posedge clk); reset <= '0;

            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end

            $stop;
        end

    endmodule  // rh_data_testbench

    /************************************

    seg7.sv

    ************************************/

    /*
     * Connor Aksama
     * 03/13/2023
     * CSE 371
     * Lab 6
    */

    /**
     * Defines data for 1-digit hexadecimal HEX display given a 4-bit unsigned integer

     * Inputs:
     *      num [4 bit] - An unsigned integer value [0x0-0xF] to display

     * Outputs:
     *      HEX [7 bit] - A HEX display for the input num, formatted in standard seven
     segment display format
    */
    module seg7(
        output logic [6:0] HEX
        ,input logic [3:0] num
        );

        // Drive HEX output signals given num
        always_comb begin
            // Light HEX using num
            case (num)
                //     Light: 6543210
                 0: HEX = ~7'b0111111;  // 0
            1: HEX = ~7'b0000110;  // 1
            2: HEX = ~7'b1011011;  // 2
            3: HEX = ~7'b1001111;  // 3
            4: HEX = ~7'b1100110;  // 4
            5: HEX = ~7'b1101101;  // 5
            6: HEX = ~7'b1111101;  // 6
            7: HEX = ~7'b0000111;  // 7
            8: HEX = ~7'b1111111;  // 8
            9: HEX = ~7'b1101111;  // 9
           10: HEX = ~7'b1110111;  // A
           11: HEX = ~7'b1111100;  // b
           12: HEX = ~7'b1011000;  // c
           13: HEX = ~7'b1011110;  // d
           14: HEX = ~7'b1111001;  // E
           15: HEX = ~7'b1110001;  // F
           default: HEX = 7'bX;
        endcase
    end

endmodule  // seg7

/*
 * Tests the functionality of the seg7 module.
*/
module seg7_testbench();

    logic [6:0] HEX;
    logic [3:0] num;

    seg7 dut (.HEX, .num);

    initial begin

        integer i;

        // Check HEX displays for integers 0x0-0xff
        for (i = 0; i <= 15; i++) begin
            #10 num = i;
        end
        #50;
    end

endmodule  // seg7_testbench
```