

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.

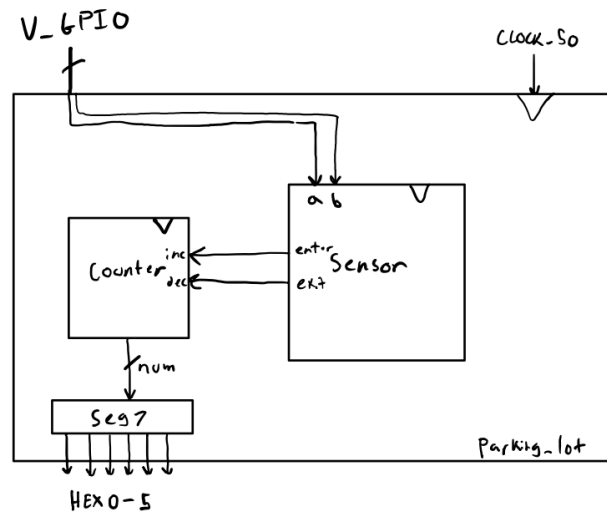


Figure 1. Top-Level Block Diagram for Lab 6 Task 1.

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.

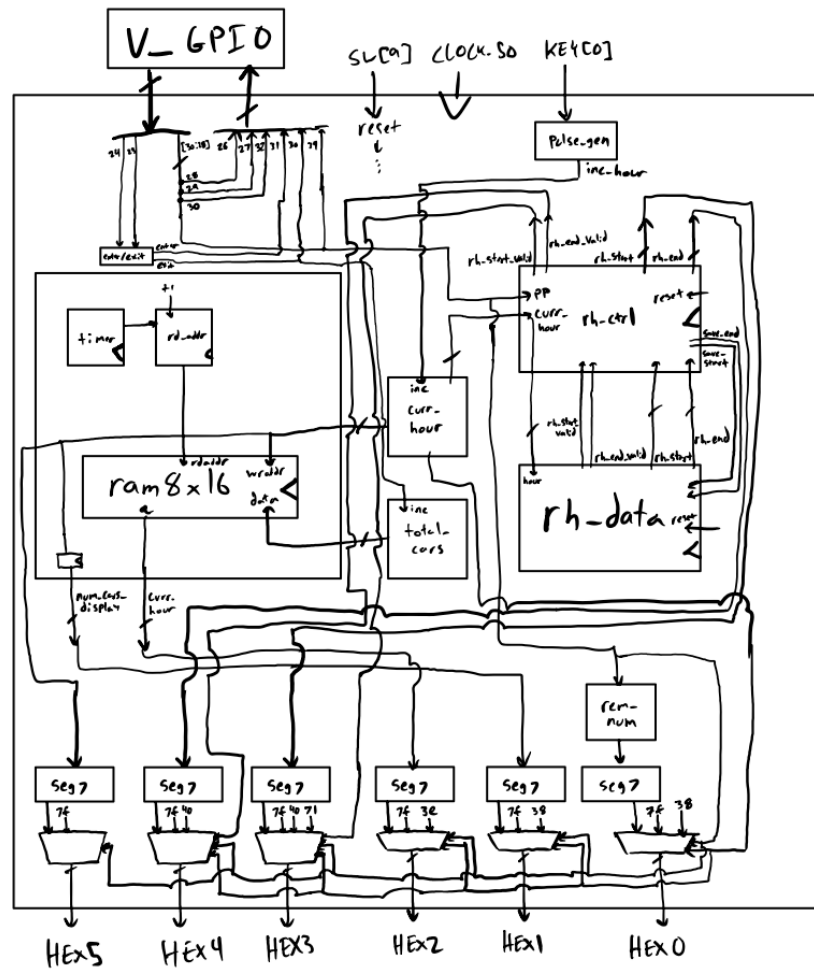


Figure 2. Top-Level Block Diagram for Lab 6 Task 2.

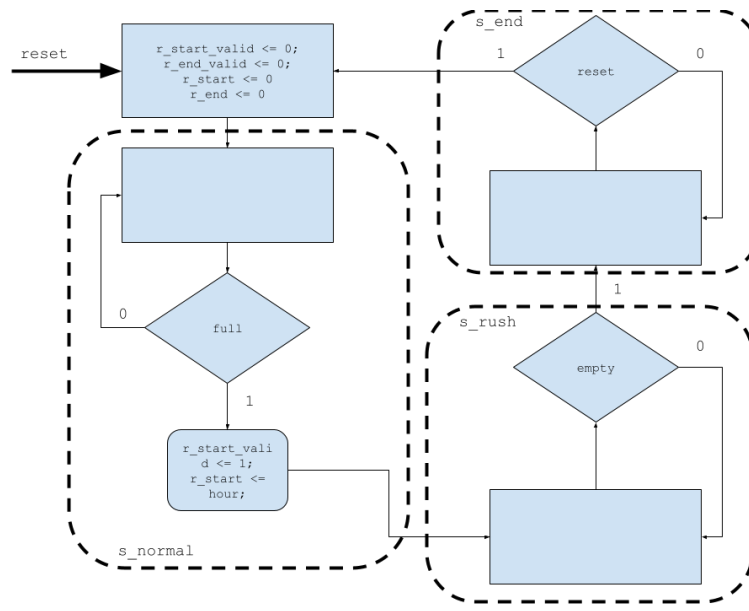


Figure 3. ASMD Chart for the Rush Hour Tracking Module

Results

Task 1

No simulations were required for Task 1 of this lab.

Task 2

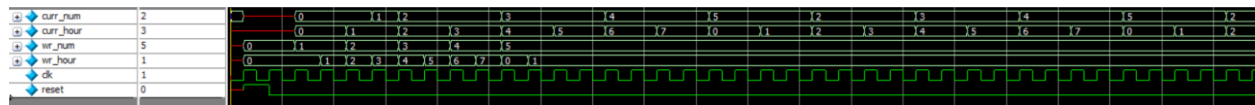


Figure 4. ModelSim Waveforms for the Car Tracking module.

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.

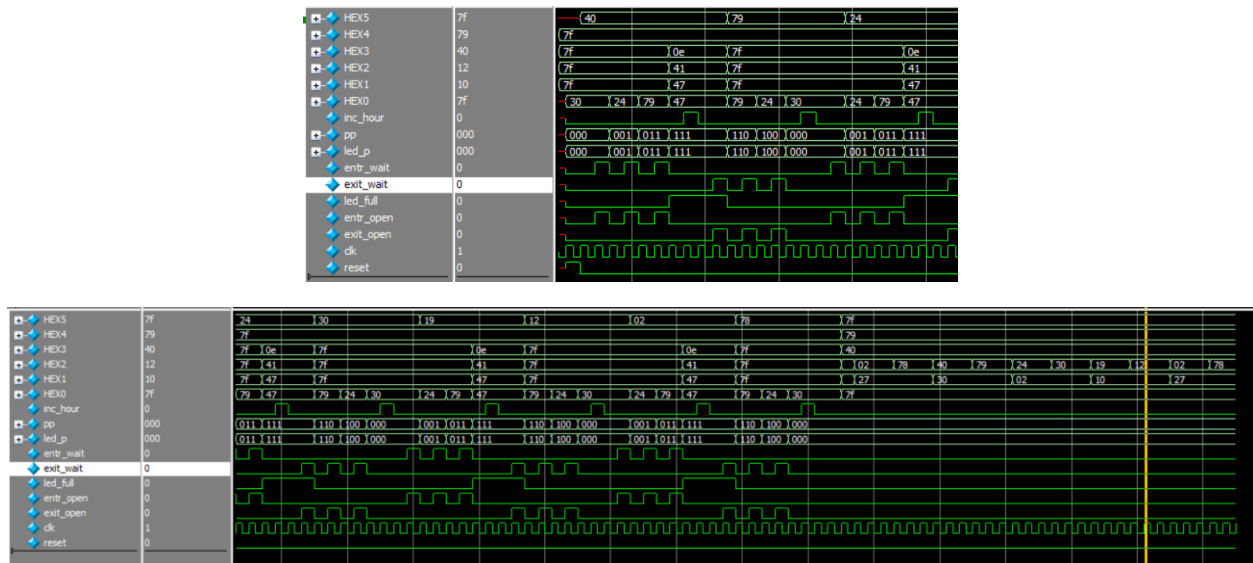


Figure 5. ModelSim Waveforms for the Task 2 top-level module.

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.

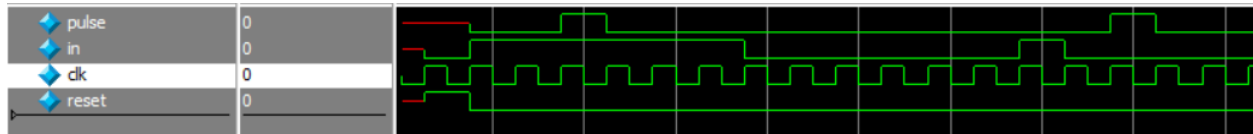


Figure 6. ModelSim Waveform for the Pulse Generator module.

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

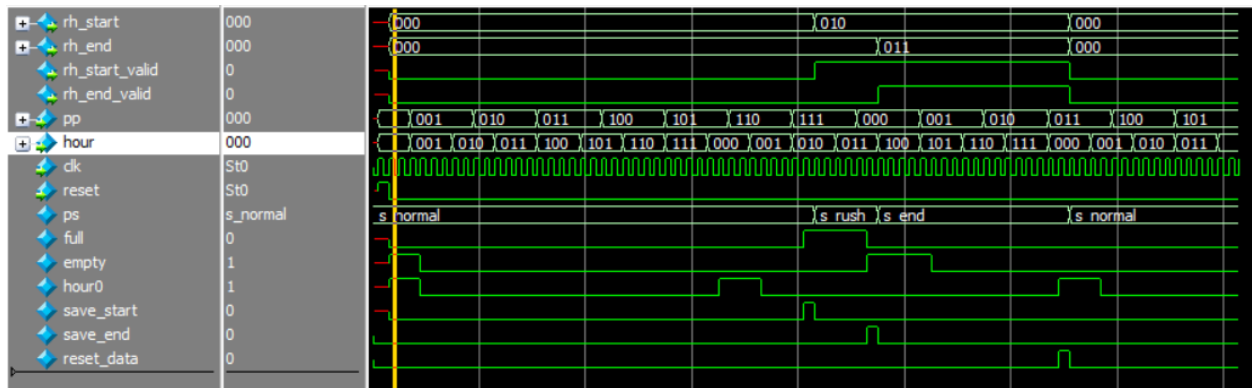


Figure 7. ModelSim Waveform for the Rush Hour Control Path module.

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.

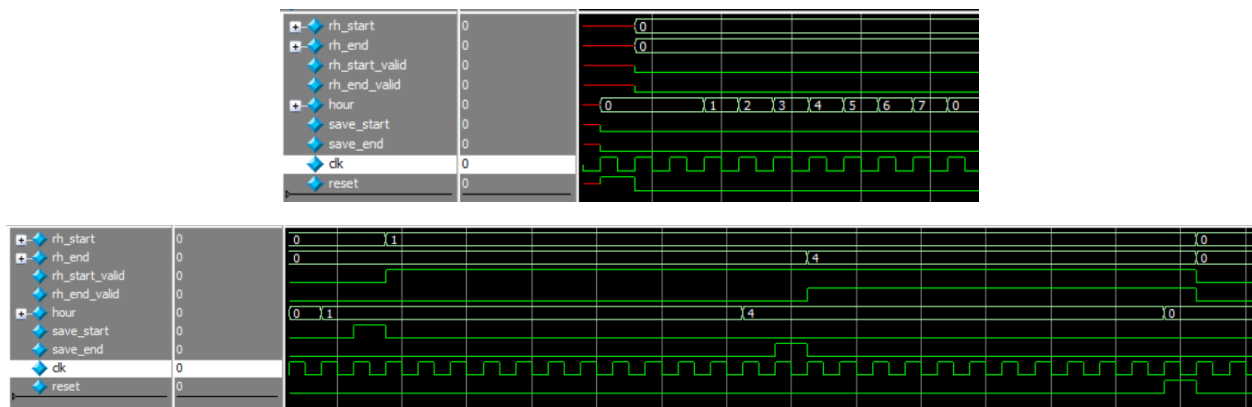


Figure 8. ModelSim Waveform for the Rush Hour Datapath module.

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.

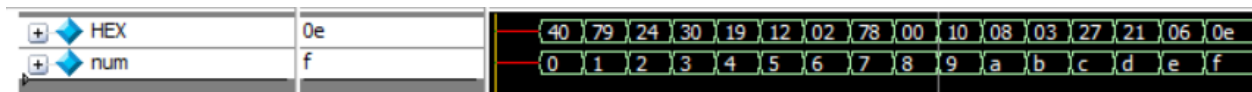


Figure 9. ModelSim Waveform for the single seven segment display module.

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)


```

1  /*****
2
3  car_tracking.sv
4
5  *****/
6
7  /*
8   * Connor Aksama
9   * 03/13/2023
10  * CSE 371
11  * Lab 6
12  */
13
14  /**
15   * Controller module for the car tracking system.
16
17   * Inputs:
18   *   wr_num [16 bit] - The number of cars to track for the current hour
19   *   wr_hour [3 bit] - The hour for which to update the number of cars
20   *   clk [1 bit] - The clock to use for this module.
21   *   reset [1 bit] - Resets the cyclic display to hour 0.
22
23   * Outputs:
24   *   curr_num [16 bit] - The number of cars that have entered in the corresponding
25   *   hour
26   *   curr_hour [3 bit] - The hour at which the corresponding number of cars have
27   *   entered
28  */
29  module car_tracking #(
30      parameter clk_freq = 50000000, duration_sec = 1
31  ) (
32      output logic [15:0] curr_num
33      , output logic [2:0] curr_hour
34      , input logic [15:0] wr_num
35      , input logic [2:0] wr_hour
36      , input logic clk, reset
37  );
38
39  localparam timer_target = clk_freq * duration_sec;
40
41  logic [2:0] rd_addr, rd_out1;
42  logic [31:0] timer;
43  logic reset_timer;
44
45  // RAM module for car tracking data
46  // clk -> clock
47  // wr_num -> data
48  // rd_addr (current hour to read) -> rdaddress
49  // wr_hour (current hour to modify) -> wraddress
50  // wren -> always write
51  // q -> curr_num (current num of cars)
52  ram8x16 ram (
53      .clock(clk)
54      , .data(wr_num)
55      , .rdaddress(rd_addr)
56      , .wraddress(wr_hour)
57      , .wren('1)
58      , .q(curr_num)
59  );
60
61  always_comb begin
62      // Reset timer at target
63      reset_timer = (timer >= timer_target - 1);
64  end
65
66  always_ff @(posedge clk) begin
67      if (reset) begin
68          // Reset cycle to beginning
69          rd_addr <= '0;

```

```

68         rd_out1 <= '0;
69         timer <= '0;
70     end else if (reset_timer) begin
71         // Reset timer, move to next hour
72         timer <= '0;
73         rd_addr <= rd_addr + 3'd1;
74     end else begin
75         // Increment timer
76         timer <= timer + 32'd1;
77     end
78
79     // Sync with RAM output
80     curr_hour <= rd_addr;
81 end
82
83 endmodule // car_tracking
84
85 /*
86  * Testbench to test the functionality of the car_tracking module
87  */
88 `timescale 1 ps / 1 ps
89 module car_tracking_testbench();
90
91     logic [15:0] curr_num;
92     logic [2:0] curr_hour;
93
94     logic [15:0] wr_num;
95     logic [2:0] wr_hour;
96     logic clk, reset;
97
98     car_tracking #(.clk_freq(2), .duration_sec(1)) dut (.*);
99
100     parameter CLOCK_PERIOD = 100;
101     initial begin
102         clk <= '0;
103         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
104     end
105
106     initial begin
107         integer i;
108
109         @(posedge clk) reset <= '1; wr_num <= '0; wr_hour <= '0;
110         @(posedge clk) reset <= '0;
111
112         // Write data to RAM
113         for (i = 0; i < 10; i++) begin
114             @(posedge clk) wr_hour <= i;
115             if (i % 2 == 0) begin
116                 wr_num <= wr_num + 15'd1;
117             end
118         end
119
120         // Cycle through values
121         for (i = 0; i < 30; i++) begin
122             @(posedge clk);
123         end
124
125         $stop;
126     end
127 end
128
129 endmodule // car_tracking_testbench
130
131 /*****
132
133 DE1_SoC.sv
134
135 *****/
136

```

```

137  /*
138  * Connor Aksama
139  * 03/13/2023
140  * CSE 371
141  * Lab 6
142  */
143
144  /**
145   * Top-level module for Lab 6 Task 2. Instantiates rush hour and car tracking modules
   and defines logic to connect I/O to peripherals.
146
147   * Inputs:
148   * SW [10 bit] - The 10 onboard switches, respectively.
149   * KEY [4 bit] - The 4 onboard keys, respectively.
150   * CLOCK_50 [1 bit] - The system clock to use for this module.
151
152   * Outputs:
153   * HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
   display format.
154   * HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
   display format.
155   * HEX2 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
   display format.
156   * HEX3 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
   display format.
157   * HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
   display format.
158   * HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
   display format.
159
160   * Inouts:
161   * V_GPIO [13 bit] - Virtual GPIO Ports
162  */
163  module DE1_SoC #(
164      parameter clk_freq = 50000000, display_duration_sec = 1
165  ) (
166      output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
167      ,input logic [9:0] SW
168      ,input logic [3:0] KEY
169      ,input logic CLOCK_50
170      ,inout logic [35:23] V_GPIO
171  );
172
173  logic clk;
174  assign clk = CLOCK_50;
175
176  // Which parking spaces occupied?
177  logic [2:0] pp;
178  assign pp = V_GPIO[30:28];
179
180  // Car waiting at gate
181  logic entr_wait, exit_wait;
182  assign entr_wait = V_GPIO[23];
183  assign exit_wait = V_GPIO[24];
184
185  // Light LEDs at each parking spot depending on presence
186  assign {V_GPIO[32], V_GPIO[27], V_GPIO[26]} = pp;
187
188  // Full LED
189  logic led_full;
190  assign V_GPIO[34] = led_full;
191
192  // Ctrl for gates
193  logic entr_open, exit_open;
194  assign V_GPIO[31] = entr_open;
195  assign V_GPIO[33] = exit_open;
196
197  // Remaining spot HEX generator
198  logic [1:0] rem_num;

```

```

199 logic [6:0] rem_hex;
200 seg7 rem_spot (.HEX(rem_hex), .num({2'b0, rem_num}));
201
202 // Current hour HEX generator
203 logic [2:0] curr_hour;
204 logic [6:0] ch_hex;
205 seg7 hour (.HEX(ch_hex), .num({1'b0, curr_hour}));
206
207 logic reset;
208 assign reset = SW[9];
209
210 logic done;
211
212 // Rush Hour Tracker
213 // rh_start -> rh_start (saved rush hour start)
214 // rh_end -> rh_end (saved rush hour end)
215 // rh_start_valid -> rh_start_valid (1 if rh_start data valid)
216 // rh_end_valid -> rh_end_valid (1 if rh_end data valid)
217 // pp -> pp
218 // curr_hour -> hour (current hour)
219 // clk -> clk
220 // reset -> reset
221 logic [2:0] rh_start, rh_end;
222 logic rh_start_valid, rh_end_valid;
223 rh_ctrl rush_hour (
224     .rh_start(rh_start)
225     ,.rh_end(rh_end)
226     ,.rh_start_valid(rh_start_valid)
227     ,.rh_end_valid(rh_end_valid)
228     ,.pp(pp)
229     ,.hour(curr_hour)
230     ,.clk(clk)
231     ,.reset(reset)
232 );
233
234 logic [6:0] end_hex, start_hex;
235 // Rush Hour End HEX generator
236 seg7 end_hour (.HEX(end_hex), .num({1'b0, rh_end}));
237 // Rush Hour Start HEX generator
238 seg7 start_hour (.HEX(start_hex), .num({1'b0, rh_start}));
239
240 // Num Car Tracker/Display
241 // curr_num -> num_cars_display (Output for saved # of cars for curr_hour_display)
242 // curr_hour -> curr_hour_display (Current hour for which to display # of cars)
243 // total_cars -> wr_num (write the total number of cars entered so far)
244 // curr_hour -> wr_hour (current hour, write total num cars for this hour)
245 // clk -> clk
246 // reset -> reset
247 logic [15:0] total_cars, num_cars_display;
248 logic [2:0] curr_hour_display;
249 car_tracking #(
250     .clk_freq(clk_freq), .duration_sec(display_duration_sec)
251 ) results (
252     .curr_num(num_cars_display)
253     ,.curr_hour(curr_hour_display)
254     ,.wr_num(total_cars)
255     ,.wr_hour(curr_hour)
256     ,.clk(clk)
257     ,.reset(reset)
258 );
259
260 logic [6:0] ncd_hex, chd_hex;
261 // EOD num cars HEX generator
262 seg7 ncd (.HEX(ncd_hex), .num(num_cars_display[3:0]));
263 // EOD curr hour HEX generator
264 seg7 chd (.HEX(chd_hex), .num({1'b0, curr_hour_display}));
265
266 logic inc_hour;
267 // Generate a one cycle pulse for the inc_hour signal (KEY[0])

```

```

268 pulse_gen gen_hour (.pulse(inc_hour), .in(~KEY[0]), .clk(clk), .reset(reset));
269
270 logic inc_cars;
271 pulse_gen gen_cars (.pulse(inc_cars), .in(entr_open), .clk(clk), .reset(reset));
272
273 always_comb begin
274     // Ctrl signals
275     led_full = (pp == 3'b111);
276     entr_open = entr_wait & (pp != 3'b111);
277     exit_open = exit_wait;
278
279     // Find remaining number of spots
280     case (pp)
281         3'b000:
282             rem_num = 2'd3;
283         3'b001, 3'b010, 3'b100:
284             rem_num = 2'd2;
285         3'b011, 3'b101, 3'b110:
286             rem_num = 2'd1;
287         3'b111:
288             rem_num = 2'd0;
289     endcase
290
291     // ctrl driver for HEX outputs
292     if (done) begin
293         HEX5 = ~7'b0;
294         if (rh_end_valid) begin
295             HEX4 = end_hex;
296         end else begin
297             HEX4 = ~7'b1000000;
298         end
299         if (rh_start_valid) begin
300             HEX3 = start_hex;
301         end else begin
302             HEX3 = ~7'b1000000;
303         end
304         HEX2 = chd_hex;
305         HEX1 = ncd_hex;
306         HEX0 = ~7'b0;
307     end else if (led_full) begin
308         HEX5 = ch_hex;
309         HEX4 = ~7'b0;
310         HEX3 = ~7'b1110001; // F
311         HEX2 = ~7'b0111110; // U
312         HEX1 = ~7'b0111000; // L
313         HEX0 = ~7'b0111000; // L
314     end else begin
315         HEX5 = ch_hex;
316         HEX4 = ~7'b0;
317         HEX3 = ~7'b0;
318         HEX2 = ~7'b0;
319         HEX1 = ~7'b0;
320         HEX0 = rem_hex;
321     end
322 end
323
324 always_ff @(posedge clk) begin
325     if (reset) begin
326         // Reset work day
327         curr_hour <= '0;
328         done <= '0;
329     end else if (inc_hour && curr_hour == 3'd7) begin
330         // EOD reached
331         done <= '1;
332     end else if (inc_hour) begin
333         // Increase work hour
334         curr_hour <= curr_hour + 3'b001;
335     end
336 end

```



```

405         @(posedge clk);
406         if (entr_open) begin
407             entr_wait <= '0;
408             pp[j] <= '1;
409         end else begin
410             j--;
411         end
412     end
413 end else begin
414     // Three cars leave
415     for (j = 0; j < 3; j++) begin
416         @(posedge clk) exit_wait <= '1;
417         @(posedge clk);
418         if (exit_open) begin
419             exit_wait <= '0;
420             pp[j] <= '0;
421         end else begin
422             j--;
423         end
424     end
425 end
426 @(posedge clk) inc_hour <= '1;
427 @(posedge clk) inc_hour <= '0;
428 end
429
430 for (i = 0; i < 32; i++) begin
431     @(posedge clk);
432 end
433
434 @(posedge clk) reset <= '1; inc_hour <= '0; pp <= '0; entr_wait <= '0; exit_wait
<= '0;
435 @(posedge clk) reset <= '0;
436 for (i = 0; i < 3; i++) begin
437     @(posedge clk) entr_wait <= '1;
438     @(posedge clk);
439     if (entr_open) begin
440         entr_wait <= '0;
441         pp[j] <= '1;
442     end
443 end
444 for (i = 0; i < 8; i++) begin
445     @(posedge clk) inc_hour <= '1;
446     @(posedge clk) inc_hour <= '0;
447 end
448 for (i = 0; i < 32; i++) begin
449     @(posedge clk);
450 end
451 $stop;
452 end
453
454 endmodule // DE1_SoC_testbench
455
456 /*****
457
458 pulse_gen.sv
459
460 *****/
461
462 /*
463  * Connor Aksama
464  * 03/13/2023
465  * CSE 371
466  * Lab 6
467 */
468
469 /**
470  * One cycle pulse generator for arbitrary length input.
471
472  * Inputs:

```

```

473  *      in [1 bit] - Input for which to generate pulse
474  *      clk [1 bit] - The clock to use for this module.
475  *      reset [1 bit] - Resets the cyclic display to hour 0.
476
477  * Outputs:
478  *      pulse [1 bit] - The output signal where pulse is sent
479  */
480  module pulse_gen (
481      output logic pulse
482      ,input logic in, clk, reset
483  );
484
485      typedef enum logic [1:0] { s_wait_rise, s_pulse, s_wait_fall } state;
486
487      state ps, ns;
488      logic in_reg;
489
490      always_comb begin
491          ns = ps;
492
493          // Handle state transitions
494          case (ps)
495              s_wait_rise: begin
496                  if (in_reg) begin
497                      ns = s_pulse;
498                  end
499              end
500              s_pulse: begin
501                  if (~in_reg) begin
502                      ns = s_wait_rise;
503                  end else begin
504                      ns = s_wait_fall;
505                  end
506              end
507              s_wait_fall: begin
508                  if (~in_reg) begin
509                      ns = s_wait_rise;
510                  end
511              end
512          endcase
513
514          pulse = (ps == s_pulse);
515      end
516
517      always_ff @(posedge clk) begin
518          // Update state
519          if (reset) begin
520              ps <= s_wait_rise;
521          end else begin
522              ps <= ns;
523          end
524          // Register input
525          in_reg <= in;
526      end
527
528  endmodule // pulse_gen
529
530  /*
531  * Testbench to test the functionality of the pulse_gen module
532  */
533  module pulse_gen_testbench();
534
535      logic pulse, in, clk, reset;
536
537      pulse_gen dut (.*);
538
539      parameter CLOCK_PERIOD = 100;
540      initial begin
541          clk <= '0;

```



```

542         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
543     end
544
545     initial begin
546         integer i;
547
548         @(posedge clk) reset <= '1; in <= '0;
549         @(posedge clk) reset <= '0; in <= '1;
550
551         for (i = 0; i < 5; i++) begin
552             @(posedge clk);
553         end
554
555         @(posedge clk) in <= '0;
556
557         for (i = 0; i < 5; i++) begin
558             @(posedge clk);
559         end
560
561         @(posedge clk) in <= '1;
562         @(posedge clk) in <= '0;
563
564         for (i = 0; i < 5; i++) begin
565             @(posedge clk);
566         end
567
568         $stop;
569     end
570
571 endmodule // pulse_gen_testbench
572
573 /*****
574
575 rh_ctrl.sv
576
577 *****/
578
579 /*
580  * Connor Aksama
581  * 03/13/2023
582  * CSE 371
583  * Lab 6
584  */
585
586 /**
587  * Controller module for the rush hour system.
588
589  * Inputs:
590  *     pp [3 bit] - Parking spaces currently occupied
591  *     hour [3 bit] - The current hour
592  *     clk [1 bit] - The clock to use for this module.
593  *     reset [1 bit] - Resets the FSM to its initial state.
594
595  * Outputs:
596  *     rh_start [3 bit] - The saved start of rush hour
597  *     rh_end [3 bit] - The saved end of rush hour
598  *     rh_start_valid [1 bit] - 1 if rh_start is valid, 0 o.w.
599  *     rh_end_valid [1 bit] - 1 if rh_end is valid, 0 o.w.
600  */
601 module rh_ctrl (
602     output logic [2:0] rh_start, rh_end
603     ,output logic rh_start_valid, rh_end_valid
604     ,input logic [2:0] pp, hour
605     ,input logic clk, reset
606 );
607
608
609     typedef enum { s_normal, s_rush, s_end } state;
610

```

```

611 state ps, ns;
612
613 logic full, empty, hour0;
614 logic save_start, save_end, reset_data;
615
616 // Datapath module
617 // rh_start -> rh_start (saved start hour)
618 // rh_end -> rh_end (saved end hour)
619 // rh_start_valid -> rh_start_valid (rh_start valid?)
620 // rh_end_valid -> rh_end_valid (rh_end valid?)
621 // hour -> hour (current hour of system)
622 // save_start -> save_start (current hour is start)
623 // save_end -> save_end (current hour is end)
624 // clk -> clk
625 // reset | reset_data -> reset (reset FSM to start)
626 rh_data rush_hour_data (
627     .rh_start(rh_start)
628     ,.rh_end(rh_end)
629     ,.rh_start_valid(rh_start_valid)
630     ,.rh_end_valid(rh_end_valid)
631     ,.hour(hour)
632     ,.save_start(save_start)
633     ,.save_end(save_end)
634     ,.clk(clk)
635     ,.reset(reset | reset_data)
636 );
637
638 always_comb begin
639
640     // Handle state transitions
641     case (ps)
642
643         s_normal: begin
644             if (full) begin
645                 ns = s_rush;
646             end else begin
647                 ns = s_normal;
648             end
649         end
650
651         s_rush: begin
652             if (empty) begin
653                 ns = s_end;
654             end else begin
655                 ns = s_rush;
656             end
657         end
658
659         s_end: begin
660             if (hour0) begin
661                 ns = s_normal;
662             end else begin
663                 ns = s_end;
664             end
665         end
666
667     endcase
668
669     // Control signals
670     save_start = (ps == s_normal) & full;
671     save_end = (ps == s_rush) & empty;
672     reset_data = (ps == s_end) & hour0;
673
674 end
675
676 always_ff @(posedge clk) begin
677     // Update FSM
678     if (reset) begin

```

```

680         ps <= s_normal;
681     end else begin
682         ps <= ns;
683     end
684
685     // Register control signals
686     full <= (pp == 3'b111);
687     empty <= (pp == 3'b000);
688     hour0 <= (hour == 3'b000);
689 end
690
691
692 endmodule // rh_ctrl
693
694 /*
695  * Testbench to test the functionality of the rh_ctrl module
696  */
697 module rh_ctrl_testbench();
698
699     logic [2:0] rh_start, rh_end;
700     logic rh_start_valid, rh_end_valid;
701
702     logic [2:0] pp, hour;
703     logic clk, reset;
704
705     rh_ctrl dut (.*);
706
707     parameter CLOCK_PERIOD = 100;
708     initial begin
709         clk <= '0;
710         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
711     end
712
713     initial begin
714         integer i;
715
716         @(posedge clk) reset <= '1; pp <= '0; hour <= '0;
717         @(posedge clk) reset <= '0;
718
719         for (i = 0; i < 40; i++) begin
720             @(posedge clk);
721             @(posedge clk)
722             if (i % 2 == 0) begin
723                 // Increment hour
724                 hour <= hour + 3'd1;
725             end
726             if (i % 3 == 0) begin
727                 // Add car to lot
728                 pp <= pp + 3'd1;
729             end
730         end
731
732         $stop;
733     end
734
735 endmodule // rh_ctrl_testbench
736
737 /*****
738
739 rh_data.sv
740
741 *****/
742
743 /*
744  * Connor Aksama
745  * 03/13/2023
746  * CSE 371
747  * Lab 6
748  */

```

```

749
750 /**
751  * Datapath module for the rush hour system.
752
753  * Inputs:
754  *     hour [3 bit] - The current hour
755  *     save_start [1 bit] - 1 if hour should be saved for start, 0 o.w.
756  *     save_end [1 bit] - 1 if hour should be saved for end, 0 o.w.
757  *     clk [1 bit] - The clock to use for this module.
758  *     reset [1 bit] - Resets the FSM to its initial state.
759
760  * Outputs:
761  *     rh_start [3 bit] - The saved start of rush hour
762  *     rh_end [3 bit] - The saved end of rush hour
763  *     rh_start_valid [1 bit] - 1 if rh_start is valid, 0 o.w.
764  *     rh_end_valid [1 bit] - 1 if rh_end is valid, 0 o.w.
765 */
766 module rh_data (
767     output logic [2:0] rh_start, rh_end
768     ,output logic rh_start_valid, rh_end_valid
769     ,input logic [2:0] hour
770     ,input logic save_start, save_end, clk, reset
771 );
772
773
774     always_ff @(posedge clk) begin
775         // Register hour/valid bits based on ctrl signals
776         if (reset) begin
777             rh_start <= '0;
778             rh_end <= '0;
779             rh_start_valid <= '0;
780             rh_end_valid <= '0;
781         end else if (save_start) begin
782             rh_start <= hour;
783             rh_start_valid <= '1;
784         end else if (save_end) begin
785             rh_end <= hour;
786             rh_end_valid <= '1;
787         end
788
789     end
790
791 end
792 endmodule // rh_data
793
794 /**
795  * Testbench to test the functionality of the rh_ctrl module
796  */
797 module rh_data_testbench();
798
799     logic [2:0] rh_start, rh_end;
800     logic rh_start_valid, rh_end_valid;
801     logic [2:0] hour;
802     logic save_start, save_end, clk, reset;
803
804     rh_data dut (.*) ;
805
806     parameter CLOCK_PERIOD = 100;
807     initial begin
808         clk <= '0;
809         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
810     end
811
812     initial begin
813         integer i;
814
815         @(posedge clk) reset <= '1; save_start <= '0; save_end <= '0; hour <= '0;
816         @(posedge clk) reset <= '0;
817

```

```

818         // Do nothing
819         for (i = 0; i < 10; i++) begin
820             @(posedge clk);
821             hour <= i;
822         end
823
824         // Save start hour
825         @(posedge clk) save_start <= '1;
826         @(posedge clk) save_start <= '0;
827
828         for (i = 0; i < 10; i++) begin
829             @(posedge clk);
830         end
831
832         // Save end hour
833         @(posedge clk); hour <= hour + 3;
834         @(posedge clk); save_end <= '1;
835         @(posedge clk); save_end <= '0;
836
837         for (i = 0; i < 10; i++) begin
838             @(posedge clk);
839         end
840
841         @(posedge clk); hour <= '0; reset <= '1;
842         @(posedge clk); reset <= '0;
843
844         for (i = 0; i < 10; i++) begin
845             @(posedge clk);
846         end
847
848         $stop;
849     end
850
851 endmodule // rh_data_testbench
852
853 /*****
854
855 seg7.sv
856
857 *****/
858
859 /*
860  * Connor Aksama
861  * 03/13/2023
862  * CSE 371
863  * Lab 6
864  */
865
866 /**
867  * Defines data for 1-digit hexadecimal HEX display given a 4-bit unsigned integer
868
869  * Inputs:
870  *     num [4 bit] - An unsigned integer value [0x0-0xF] to display
871
872  * Outputs:
873  *     HEX [7 bit] - A HEX display for the input num, formatted in standard seven
874  *     segment display format
875  */
876 module seg7(
877     output logic [6:0] HEX
878     ,input logic [3:0] num
879 );
880
881     // Drive HEX output signals given num
882     always_comb begin
883         // Light HEX using num
884         case (num)
885             //      Light: 6543210
886             0: HEX = ~7'b0111111; // 0

```

```

886         1: HEX = ~7'b0000110; // 1
887         2: HEX = ~7'b1011011; // 2
888         3: HEX = ~7'b1001111; // 3
889         4: HEX = ~7'b1100110; // 4
890         5: HEX = ~7'b1101101; // 5
891         6: HEX = ~7'b1111101; // 6
892         7: HEX = ~7'b0000111; // 7
893         8: HEX = ~7'b1111111; // 8
894         9: HEX = ~7'b1101111; // 9
895         10: HEX = ~7'b1110111; // A
896         11: HEX = ~7'b1111100; // b
897         12: HEX = ~7'b1011000; // c
898         13: HEX = ~7'b1011110; // d
899         14: HEX = ~7'b1111001; // E
900         15: HEX = ~7'b1110001; // F
901         default: HEX = 7'bX;
902     endcase
903 end
904
905 endmodule // seg7
906
907 /*
908  * Tests the functionality of the seg7 module.
909  */
910 module seg7_testbench();
911
912     logic [6:0] HEX;
913     logic [3:0] num;
914
915     seg7 dut (.HEX, .num);
916
917     initial begin
918
919         integer i;
920
921         // Check HEX displays for integers 0x0-0xff
922         for (i = 0; i <= 15; i++) begin
923             #10 num = i;
924         end
925         #50;
926     end
927
928 endmodule // seg7_testbench
929

```