Justin Sim and Mina Gao (Lab Group 4)
EE 371
March 7, 2024
Lab 6 Report

Procedure

This lab evolved from the foundational concepts introduced in Lab 1, and segmented into two primary tasks: Rush Hour analysis and Car Tracking with RAM. Task 1 is a straightforward process. It was to replicate the Lab 1 setup on a new breadboard configuration within Labsland. In task 2, we are asked to develop a 3D simulation of a restaurant parking lot, incorporating an 8-hour workday scenario to examine rush hour patterns. This involved designing a system to display the hourly car entry rates and identify rush hour periods, indicated by the parking lot reaching full capacity and subsequently emptying. The implementation included constructing an 8x16 RAM module to facilitate data storage and retrieval. After managing the complex module interactions with different states, we were able to successfully demonstrate the 3D parking lot simulation.

Rush Hour

To address the Rush Hour, I began with constructing a ASMD chart and a state diagram, as shown below. The ASMD chart was pivotal in visualizing the logic and transitions necessary for detecting the beginning and end of rush hour within the simulated 8-hour workday of a restaurant parking lot. The state diagram facilitated a clear representation of the system's states, transitions based on the car count within the parking lot, and the conditions under which these transitions occurred. The implementation involved programming FSMs to monitor and record the parking lot's occupancy status, tracking the moment it reached full capacity (3 cars) and the subsequent return to empty. The system incremented the workday hour by hour, closely monitoring car entries and exits. Using separate control and datapath modules, the FSM dynamically adjusted to the current hour, identifying the first instance of rush hour - defined as the lot filling and then emptying. Valid rush hour sequences were identified, and the start and end times of rush hour were displayed on designated HEX displays. If no rush hour occurred, the system was designed to indicate this with a "-" on both outputs.

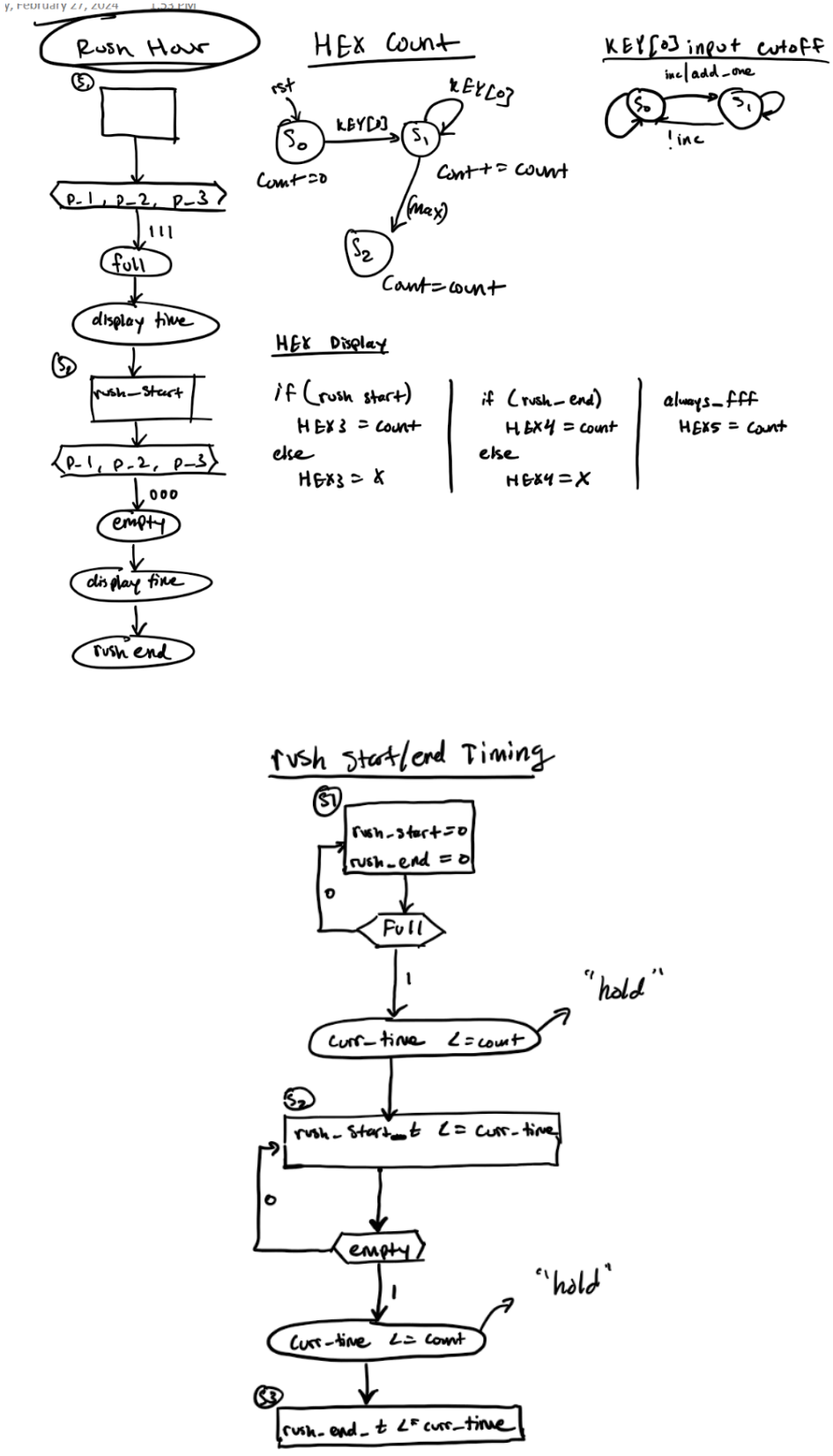

Figure 1: ASMD chart & state diagram for Rush Hour

Car Tracking and RAM

For the Car Tracking task, my approach involved a dual-port 8x16 RAM to record the number of cars entering the parking lot each hour. The design of this component was crucial, it required separate channels for reading and writing, with each RAM address corresponding to a specific business hour. As shown in the figure 2, the design accumulated data on car entries, storing these data in RAM. At the conclusion of the 8-hour period, a counter was used to cycle through the RAM addresses, displaying the total number of cars for each hour on the HEX display. This cycled every second, providing a comprehensive overview of parking lot activity throughout the day by hours.

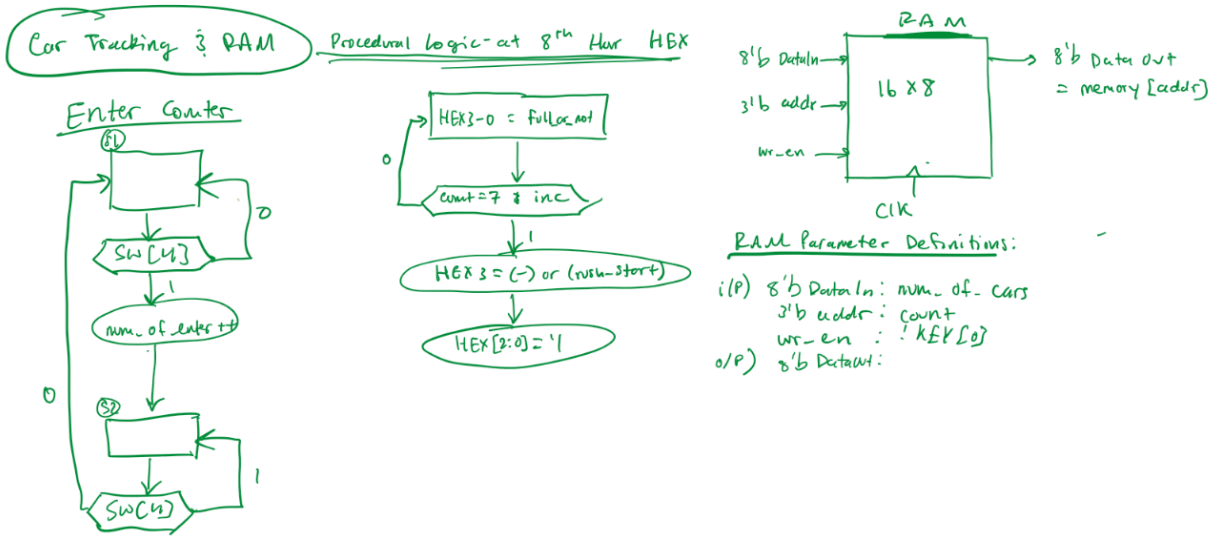

Figure 2: ASMD chart & state diagram for Car Tracking and RAM

Results

Counter_testbench():

Inputs:

1) clk
2) reset
3) inc

Outputs:

1) count

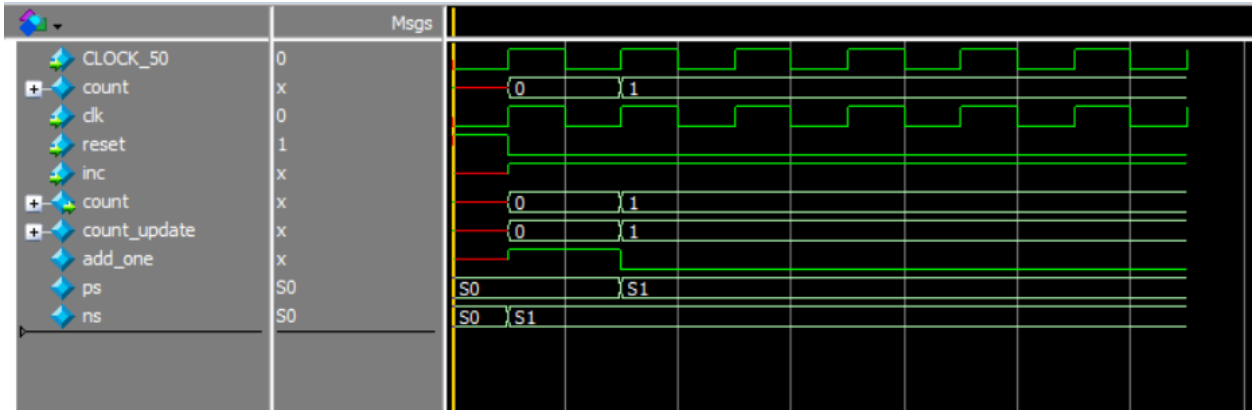

Figure 3: Counter Testbench ModelSim

The counter module is a digital counter designed to track the number of vehicles that have entered or exited the parking lot, with specific behaviors when reaching its maximum capacity or when it is empty. It uses a FSM to ensure that the 'inc' signal is only considered for one clock cycle. It implements logic to handle 'reset', increment ('inc'), and maintain the count within the specified range (0 to MAX) using a given parameter. When 'reset' is high, the counter is set to 0. If 'inc' is high and the current count is less than MAX, the counter increments. If the count reaches MAX, it stays there until reset or another operation changes its state. As shown in Figure 3, after reset when 'inc' is high, the state switches and 'add_one' is high as we keep incrementing. We were able to verify its functionality based on the simulation.

DE1_SoC_testbench()

The DE1_SoC tests two different scenarios,

1) Rush Hour
    a) Show Start of Rush hour (hour when lot turns full) on HEX3
    b) Show End of Rush hour (hour when lot turns empty after being full) HEX4
    c) If no Rush Hour show '-' (7'b1111110) on HEX3 and HEX4
2) Car Tracking and RAM
    a) At the end of the last hour, demonstrate that the HEX displays all the addresses and the corresponding number of cars at that hour

Inputs:

CLOCK_50
[3:0] KEY KEY[0] used as manual hour increment
[9:0] SW SW[9] used as reset

in/outputs:

[35:0] V_GPIO Virtual GPIO for DE1_SoC

Outputs:

IF FINAL HOUR:

HEX5 - (BLANK)

HEX4 - END OF RUSH HOUR (OR - IF NONE)

HEX3 - START OF RUSH HOUR (OR - IF NONE)

HEX2 - RAM ADDRESS

HEX1 - RAM VALUE

HEX0 - (BLANK)

ELSE:

HEX5 - CURRENT HOUR

HEX4 - (BLANK)

HEX3 - (BLANK)

HEX2 - (BLANK)

HEX1 - (BLANK)

HEX0 - NUMBER OF AVAILABLE SPOTS

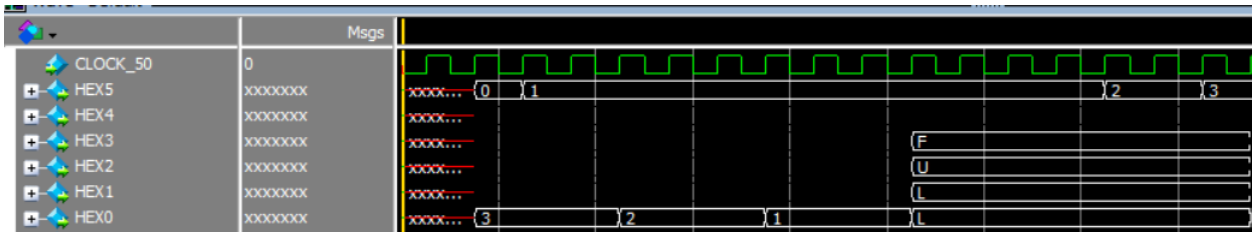

Figure 4: DE1_SoC Rush Hour - FULL Case (NOT FINAL HOUR)

Figure 4 shows the testbench for the start of Rush Hour, and tests the functionality of FULL. As you can see, HEX0 is showing the available number of spots, while it decrements down to 0, where 0 shows FULL on HEX3-HEX0. Additionally, we can see the current hour on HEX5. This testbench shows us that the board is ready to display available number of spots, current hour, and FULL capabilities

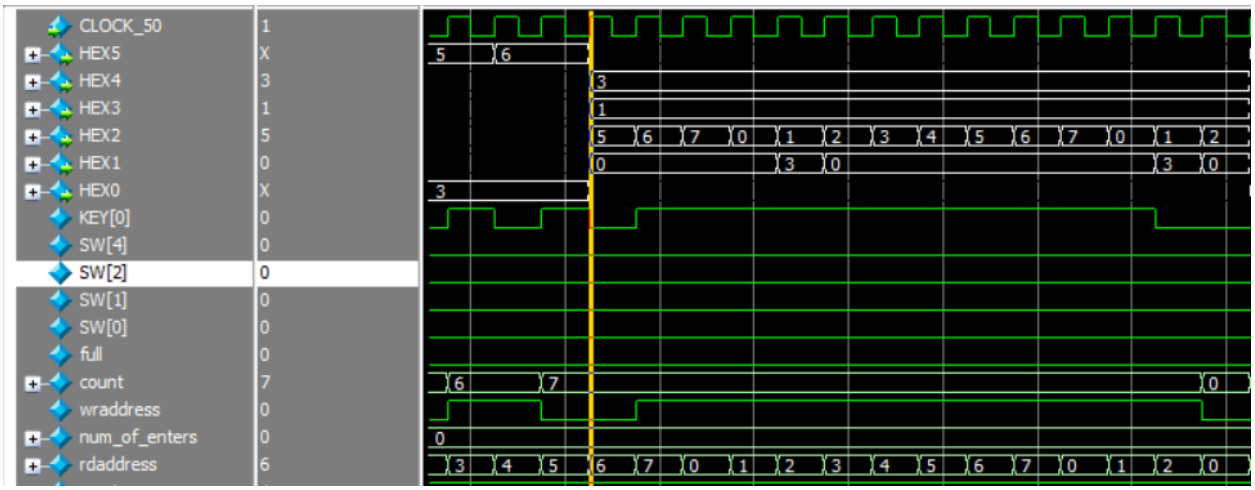

Figure 5: DE1_SoC Rush Hour and Car Track Testbench - (FINAL HOUR) (3 cars at hour 1)

Figure 5 above shows the Rush Hour and Car Tracking capabilities of the board. To see the Rush Hour, we can look at HEX 3 as the start of rush hour (hour 1), which is in line with Figure 4, and HEX4 the end of rush hour (hour 3) which is in line with Figure 6 below. To see the Car Track, we look to HEX2 which cycles through the RAM addresses, and HEX1 which displays the number of cars entered at that hour. We can see that it shows 3 for hour 1 which is in line with Figure 4. This testbench shows us that the board is ready to display the Rush Hour start and end times as well as the final hour cycle through RAM.

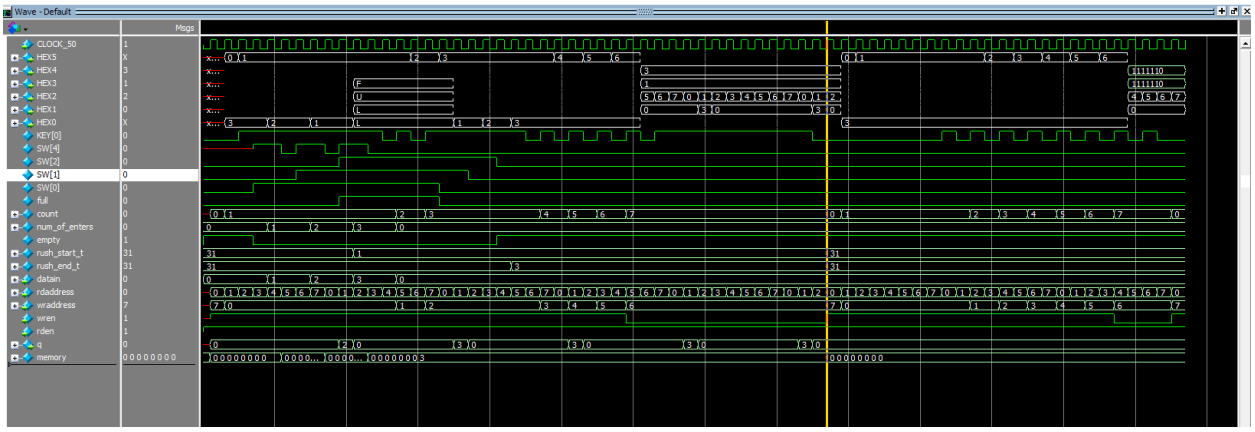

Figure 6: DE1_SoC Rush Hour and Car Track Testbench - FULL

Figure 6 above is primarily used as a reference for Figure 4 and 5. Again, we can see that rush hour ends at hour 3, which is in line with the HEX display output. We also tested the No Rush Hour Case starting with a reset at the yellow line. You can see that the HEX3-4 outputs are 1111110 which signify a '-' in the DE1_SoC HEX display. This testbench shows us that the board is ready to display a No Rush Hour condition, and that the reset is functional for all cases of the spec.

clock_divider_testbench():

clock_divider.sv was useful because it allowed for a longer clock cycle. It slows down the clock, so that we were able to scroll through the addresses at a reasonable rate (~1sec)

Inputs:

1) clk
2) reset

Outputs:

1) [31:0] divided_clocks

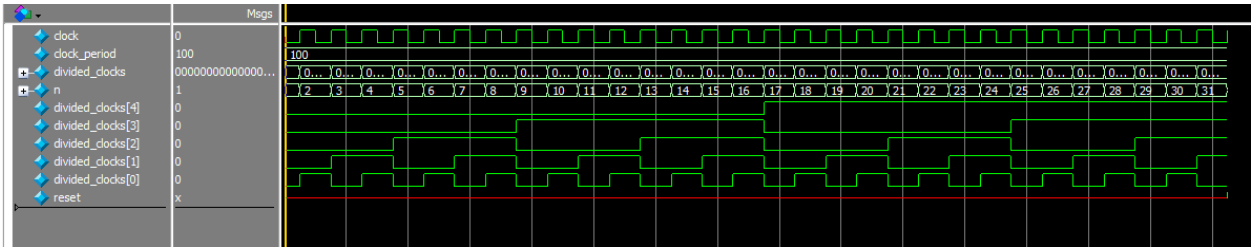

Figure 7: clock_divider testbench Modelsim

The image above shows a testbench for the module. The clock divider has 31 indices, each with a different clock frequency. One can see that divided_clock[1] has double the period of [0], [2] has double the period of [1] and so on. This is consistent with our expectations for the functionality, and it serves a great purpose in customizing the speed of our clock.

display_testbench():

Depending on the 5-bit input (designed to take the 5-bit output 'addr' from the binary search datapath), it will return the corresponding Hexadecimal character, which was helpful for Task 2 where we wanted to convert binary input to something readable on the HEX display.

Input:

1) [4:0] count

Outputs:

1) [6:0] HEX4 (hexadecimal output)
2) [6:0] HEX5 (hexadecimal output)

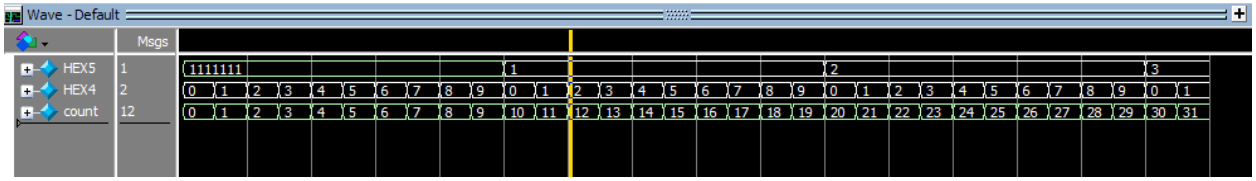

Figure 8: display_testbench Modelsim

The image above shows a testbench for the module. As we updated count, HEX4 and 5 got updated accordingly which aligned with our expectations, meaning that the module can perform 5-bit binary to 7-bit hexadecimal conversion.

RAM8x16_testbench():

A RAM8x16, or "RAM 8 words by 16 bits," is a type of memory module that can store 8 words of data, with each word containing 16 bits. This means it has a total capacity of 8 * 16 = 128 bits. It allows for both read and write operations, where data can be written into specific addresses within the memory array and later read back from those addresses. The RAM8x16 typically operates with clock signals for synchronization and may include additional control signals such as write enable and read enable to control data access. It is commonly used in digital systems for temporary data storage, data buffering, or as part of larger memory structures.

Input:

1) Clk, reset, wren, rden
2) [15:0] datain
3) [2:0] rdaddress, wraddress

Outputs:

3) [15:0

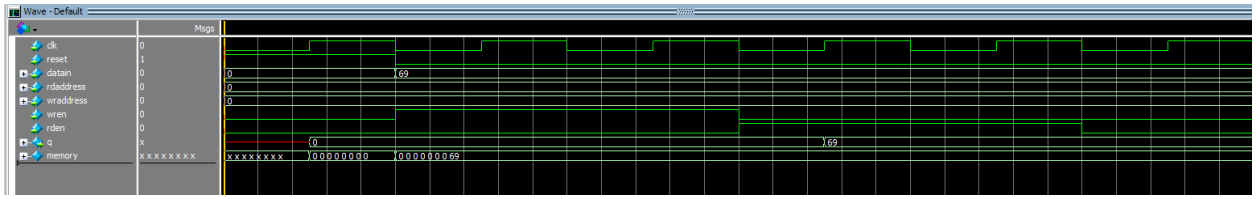

Figure 9: RAM8x16_testbench ModelSim

The image above shows a testbench for the module. Once reset is released and data is written to the ram (69). We can see that the memory instantly adds the datain to the wraddress (0). When rdaddress is 0 and rden becomes 1, we see that q returns the value in memory at address 0 - 69.

Appendix

```
1   /*
2   Justin Sim and Mina Gao
3   3/6/2024
4   EE 371 Hussein
5   Lab 6
6
7      This module divides the input clock such that it can slow down the cycle:
8      divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz,[25] = 0.75Hz, ...
9
10     inputs:
11        clock
12     in/outputs:
13        [31:0] divided_clocks           array of varying clock speeds
14
15  */
16  module clock_divider (clock, reset, divided_clocks);
17     input logic reset, clock;
18     output logic [31:0] divided_clocks = 0;
19     always_ff @(posedge clock) begin
20        divided_clocks <= divided_clocks + 1;
21     end
22  endmodule               // clock_divider
23  //-------------------------------------------------------------------------------------------
    -------------
24  // clock_divider_testbench tests the expected behaviors of the clock_divider module
25
26  module clock_divider_testbench();
27  logic reset, clock;
28  logic [31:0] divided_clocks;
29
30  clock_divider dut (.clock, .reset, .divided_clocks);
31
32  // clock setup
33     parameter clock_period = 100;
34     initial begin
35        clock <= 0;
36        forever #(clock_period / 2) clock <= ~clock;
37     end
38     integer n;
39     initial begin
40        for (n=1; n<=31; n++) begin
41        @(posedge clock);
42        end
43        reset = 1;
44        $stop;
45     end
46
47  endmodule               // clock_divider_testbench
```

```
1   /*
2   Mina Gao and Justin Sim
3   3/6/2024
4   EE 371 Hussein
5   Lab 6
6
7      Counter Module:
8         Keeps track of the number of vehicles that have entered or exited
9         the garage. Also has specific behaviors for the Maximum capacity and
10        empty capacity cases.
11
12     inputs:
13        clock
14        reset
15        inc:     1'b increment
16
17     outputs:
18        count:   5'b to represent a number between 0 - 7
19
20  */
21
22  module counter #(parameter MAX = 7) (clk, reset, inc, count);
23     input logic clk, reset;
24     input logic inc;
25     output logic [4:0] count;
26
27     logic [4:0] count_update;
28     logic add_one;
29
30     /* FSM to ensure inc is high for only one clock cycle */
31     enum {S0,S1} ps, ns;
32     always_comb begin
33        case (ps)
34           S0: if (inc) ns <= S1; else ns <= S0;
35           S1: if (!inc) ns <= S0; else ns <= S1;
36        endcase
37     end //always_comb
38     assign add_one = (ps == S0 & inc);
39     always_ff @(posedge clk) begin
40        if (reset) ps <= S0; else ps <= ns; end
41
42
43     /* reset = 1 -> count = 0
44        when count is in the range (0-MAX), count will update accordingly base on
45        inputs(inc)
46        When the count hits its max (25) and inc is 1, count will not get updated
47        and it will hold at MAX until the next input
48     */
49
50     always_ff @(posedge clk) begin
51        if (reset) count_update <= 5'b00000;
52        else if (add_one & count < MAX) count_update <= count_update + 5'd00001;
53        else if (count == MAX) count_update <= MAX;
54        else if (count == 5'b00000) count_update <= 5'b00000;
55        else count_update <= count_update;
56     end
57
58     assign count = count_update;
59  endmodule
60
61  //-------------------------------------------------------------------
62  //counter_testbench tests the expected and unexpected behaviors, reset behaviors, and a
63  // FULL Capacity behavior for MAX = 5
64
65  module counter_testbench();
66     logic clk, reset, inc;
67     logic [4:0] count;
68
69     counter #(5) dut(.clk, .reset, .inc, .count);
70
71     // clock setup
72     parameter clock_period = 100;
73
```

```
74      initial begin
75         clk <= 0;
76         forever #(clock_period / 2) clk <= ~clk;
77      end
78
79      initial begin
80         reset <= 1;                          @(posedge clk);
81                                              @(posedge clk);
82         reset <= 0; inc <= 1;   @(posedge clk); //keep increasing until passing the MAX
83                                              @(posedge clk);
84                     inc <= 1; @(posedge clk);
85                                              @(posedge clk);
86                     inc <= 1; @(posedge clk);
87                                              @(posedge clk);
88                     inc <= 1; @(posedge clk);
89                                              @(posedge clk);
90                     inc <= 1; @(posedge clk);
91                                              @(posedge clk);
92                     inc <= 1; @(posedge clk);
93                                              @(posedge clk);
94                     inc <= 0; @(posedge clk); // keep decreasing until it hits 0
95   //                                         @(posedge clk);
96   //                inc <= 0; dec <= 1;      @(posedge clk);
97   //                                         @(posedge clk);
98   //                inc <= 0; dec <= 1;       @(posedge clk);
99   //                                         @(posedge clk);
100  //                inc <= 0; dec <= 1;      @(posedge clk);
101  //                                         @(posedge clk);
102  //                inc <= 0; dec <= 1;      @(posedge clk);
103  //                                         @(posedge clk);
104  //                inc <= 0; dec <= 1;      @(posedge clk);
105  //                                         @(posedge clk);
106        $stop; // end simulation
107     end
108  endmodule
109
```

```
1   /*
2   Justin Sim and Mina Gao
3   3/6/2024
4   EE 371 Hussein
5   Lab 6
6
7      This is a top level module that combines the controls and datapath
8      for parking lot simulation on DE1_SoC in RemoteHub Lab
9      It has 2 main functionalities:
10        Rush Hour:           Track and displays busiest hour start and end times
11        Car Track and RAM:   Track and displays num of cars entered at each hour
12
13     inputs:
14        CLOCK_50
15        [3:0] KEY         // KEY[0] used as manual hour increment
16        [9:0] SW          // SW[9] used as reset
17
18     in/outputs:
19        [35:0] V_GPIO     Virtual GPIO for DE1_SoC
20
21     outputs:
22        [6:0] HEX         7'b HEX display on DE1_SoC
23        [9:0] LEDR        Signifies Car Presence
24
25  */
26  module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, SW, LEDR, V_GPIO);
27
28     // Define ports
29     input  logic         CLOCK_50;
30     output logic [6:0]   HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
31     input  logic [3:0]   KEY;
32     input  logic [9:0]   SW;
33     output logic [9:0]   LEDR;
34     inout  logic [35:23] V_GPIO;
35
36     // Define intermediate logic
37     bit            full;          // 1 if all spots currently filled
38     bit            empty;           // 1 if all spots currently open
39     logic [4:0]    count;   // variable to hourly count
40     logic          reset;
41     logic [4:0]    rush_start_t, rush_end_t = '1;
42     logic [4:0]    open_spots; // # of available spots
43
44     // Define intermediate logic for slow clock
45     parameter whichClock = 24; // 0.75 Hz clock
46     logic [31:0] div_clk;
47     logic clkSelect;
48
49
50     // Define intermediate logic for Rush Hour Display
51     logic [6:0] disp_rush_start, disp_rush_end, disp_hour;
52
53     // Define intermediate logic for Car Tracker Display
54     logic [6:0] disp_F, disp_U, disp_L, disp_spots;
55
56     // Define intermediate logic for RAM
57     logic [3:0]     rdaddress;
58     logic [15:0]    num_of_enters;
59     logic [15:0]    dout;
60
61
62     // assign value to intermediate logic
63     assign reset = SW[9];
64     assign full = (V_GPIO[30:28] == 3'b111);
65  // assign empty = (SW[2:0] == 3'b000);          // For Simulation
66     assign empty = (V_GPIO[30:28] == 3'b000);    // For Board
67
68     // FPGA input
69     assign LEDR[0] = V_GPIO[28];  // Presence parking 1 Lets you know if there is a car in
    parking spot 1. V_GPIO[28]
70     assign LEDR[1] = V_GPIO[29];  // Presence parking 2 Lets you know if there is a car in
    parking spot 2. V_GPIO[29]
71     assign LEDR[2] = V_GPIO[30];  // Presence parking 3 Lets you know if there is a car in
```

```
    parking spot 3. V_GPIO[30]
72      assign LEDR[3] = V_GPIO[23];  // Presence entrance Lets you know if there is a car
    waiting at the entrance.
73      assign LEDR[4] = V_GPIO[24];  // Presence exit Lets you know if there is a car waiting
    at the exit.
74
75      // Autonomous FPGA output for Board
76      assign V_GPIO[26] = (V_GPIO[28]) ? 1 : 0;
77      assign V_GPIO[27] = (V_GPIO[29]) ? 1 : 0;
78      assign V_GPIO[32] = (V_GPIO[30]) ? 1 : 0;
79      assign V_GPIO[34] = (full) ? 1 : 0;
80      assign V_GPIO[31] = (V_GPIO[23] && V_GPIO[30:28] != 3'b111) ? 1 : 0;
81      assign V_GPIO[33] = (V_GPIO[24]) ? 1 : 0;
82
83      // FPGA output for Simulation
84  // assign V_GPIO[26] = SW[0]; // LED parking 1 Lets you change LED color of parking spot 1
    (0 = green, 1 = red)
85  // assign V_GPIO[27] = SW[1]; // LED parking 2 Lets you change LED color of parking spot 2
    (0 = green, 1 = red)
86  // assign V_GPIO[32] = SW[2]; // LED parking 3 Lets you change LED color of parking spot 3
    (0 = green, 1 = red).
87  // assign V_GPIO[34] = SW[3]; // LED full  Lets you change LED color of the full indicator
    LED (0 = green, 1 = red).
88  // assign V_GPIO[31] = SW[4]; // Open entrance: Opens the entrance gate. When you send a 1,
    the gate will stay open until a car enters the lot.
89  // assign V_GPIO[33] = SW[5]; // Open exit: Opens the exit gate. When you send a 1, the
    gate will stay open until a car leaves the lot
90
91
92      // Instantiate Counter for Rush Hour
93      counter ct (.clk(CLOCK_50), .reset(reset), .inc(!KEY[0]), .count(count));
94
95      // Instantiate clock divider for Car Tracker
96      clock_divider cdiv (.clock(CLOCK_50),
97      .reset(reset),
98      .divided_clocks(div_clk));
99  // assign clkSelect = CLOCK_50; // for simulation
100     assign clkSelect = div_clk[whichClock]; // for board
101
102     /*****************************************RUSH
    HOUR*****************************************/
103
104     enum {S1, S2, S3} ps, ns;
105     always_comb begin
106        case(ps)
107        S1: if (full) ns = S2;
108            else ns = S1;
109        S2: if (empty) ns = S3;
110            else ns = S2;
111        S3: ns = S3;
112        endcase
113     end//always_comb
114
115     // FSM DFF
116     always_ff @(posedge CLOCK_50)
117        if (reset) ps <= S1; //reset
118        else ps <= ns;
119
120     // Controls DFF
121     always_ff @(posedge CLOCK_50)
122        if (reset) begin
123            rush_start_t <= '1;
124            rush_end_t <= '1;
125        end else if (ps==S1 && ns == S2)
126           rush_start_t <= count;
127 //       else if (ps==S2 && ns == S3)
128         else if (ps==S2 && ns == S3)
129           rush_end_t <= count;
130        else begin
131           rush_start_t <= rush_start_t;
132           rush_end_t <= rush_end_t;
133        end
134
```

```
135       display display_time (.clk(CLOCK_50), .count(count), .HEX(disp_hour));
136       display display_rush_start (.clk(CLOCK_50), .count(rush_start_t), .HEX(disp_rush_start));
137       display display_rush_end (.clk(CLOCK_50), .count(rush_end_t), .HEX(disp_rush_end));
138
139    /*************************************CAR TRACKING AND
       RAM*****************************************/
140
141    // count_enter Store the total number of cars entered, depending on SW[2:0]
142       enum {O1,O2} ps0, ns0;
143
144       // FSM to ensure car enter signal if held for one clock cycle only
145       always_comb begin
146          case(ps0)
147    //       O1:   if (SW[4]) ns0 = O2;    // for simulation
148           O1:
149                   if ((V_GPIO[23]) && (V_GPIO[30:28] != 3'b111))
150                       ns0 = O2;
151                   else ns0 = O1;
152                   // num_of_enters = 0;
153          O2:
154                if ((V_GPIO[23]) && (V_GPIO[30:28] != 3'b111))
155                   ns0 = O2;
156                else
157                   ns0 = O1;
158          endcase
159       end //always_comb
160
161       // DFF for Car Tracker FSM
162       always_ff @(posedge CLOCK_50)
163          // if reset or increment hour
164          if (reset) begin
165             ps0 <= O1;
166          end else ps0 <= ns0;
167
168       // DFF for num of enters logic
169       always_ff @(posedge CLOCK_50)
170           if (reset)
171               num_of_enters <= 0;
172           else if (ps0==O1 && (V_GPIO[23]) && (V_GPIO[30:28] != 3'b111))
173               num_of_enters <= num_of_enters + 1;
174           else
175               num_of_enters <= num_of_enters;
176
177       // Defines number of open spots based on V_GPIO i/o
178       always_comb begin
179        // case(SW[2:0]) // for simulation
180        case({V_GPIO[32],V_GPIO[27], V_GPIO[26]})   // for board
181            3'b000: open_spots = 5'd3; // 0 cars
182            3'b001: open_spots = 5'd2; // 1 cars
183            3'b010: open_spots = 5'd2; // 1 cars
184            3'b011: open_spots = 5'd1; // 2 cars
185            3'b100: open_spots = 5'd2; // 1 cars
186            3'b101: open_spots = 5'd1; // 2 cars
187            3'b110: open_spots = 5'd1; // 2 cars
188            3'b111: open_spots = 5'd0; // 3 cars (full)
189           default: open_spots = 5'd3; // Default case
190        endcase end
191
192       // Instantiate RAM to keep track of number of enters each hour
193       /* Inputs:  CLOCK_50, reset,
194                   datain,
195                   rdaddress - given hour (scrolls through 0-7),
196                   wraddress - current hour
197                   wren, rden
198
199          Outputs: dout - number of cars at the given hour
200       */
201          RAM_8x16 ram (
202        .clk(CLOCK_50),          // Clock input
203        .reset(reset),         // Reset input
204        .datain(num_of_enters),// Input data
205        .rdaddress(rdaddress), // Read address (3 bits for 8 locations)
206        // .wraddress(count-1), // Write address (3 bits for 8 locations)
```

```
207       // .wren(count < 7),          // Write enable
208       .wraddress(count), // Write address (3 bits for 8 locations)
209        .wren(count < 8),          // Write enable
210       .rden(count >= 8),          // Read enable
211       .q(dout)     // Output data
212   );
213
214      // DFF to Scroll through RAM read addresses
215      always_ff @(posedge clkSelect) begin
216         if (reset) rdaddress <= 0;
217         else begin
218            if (rdaddress == 7) rdaddress <= 0;
219            else                rdaddress <= rdaddress+1;
220         end
221      end
222
223      // Instantiate Displays for HEX0-5
224      logic [6:0] disp_addr, disp_val;
225      display addr (.clk(CLOCK_50), .count({1'b0,rdaddress}), .HEX(disp_addr));
226      display val  (.clk(CLOCK_50), .count(dout), .HEX(disp_val));
227      display F (.clk(CLOCK_50), .count(5'b01010), .HEX(disp_F));
228      display U (.clk(CLOCK_50), .count(5'b01011), .HEX(disp_U));
229      display L (.clk(CLOCK_50), .count(5'b01100), .HEX(disp_L));
230      display spots (.clk(CLOCK_50), .count(open_spots), .HEX(disp_spots));
231
232   /*****************************PROCEDURAL LOGIC FOR HEX : 8TH
      HOUR*********************************/
233   /* IF FINAL HOUR:
234         HEX5 - X
235         HEX4 - END OF RUSH   (OR -)
236         HEX3 - START OF RUSH (OR -)
237         HEX2 - RAM ADDRESS
238         HEX1 - RAM VALUE
239         HEX0 - X
240      ELSE
241         HEX0 - L IF FULL OR # OPEN SPOTS
242         HEX1 - L IF FULL OR X
243         HEX2 - U IF FULL OR X
244         HEX3 - F IF FULL OR X
245         HEX4 - X
246         HEX5 - CURRENT HOUR
247         */
248      always_ff @(posedge CLOCK_50) begin
249         case(count == 8)  // entering final hour
250            1: begin
251               HEX0 = '1;
252               HEX1 = disp_val;
253               HEX2 = disp_addr;
254               HEX3 = (rush_start_t == '1) ? 7'b1111110 : disp_rush_start;
255               HEX4 = (rush_end_t == '1) ? 7'b1111110 : disp_rush_end;
256               HEX5 = '1;
257               end
258            0: begin
259               HEX0 = (open_spots == 0 || full) ? disp_L : disp_spots;
260               HEX1 = (open_spots == 0 || full) ? disp_L : '1;
261               HEX2 = (open_spots == 0 || full) ? disp_U : '1;
262               HEX3 = (open_spots == 0 || full) ? disp_F : '1;
263               HEX4 = '1;
264               HEX5 = disp_hour;
265               end
266         endcase
267      end // always_ff
268
269   endmodule  // DE1_SoC
270
271   /*****************************************TESTBENCH*****************************************
      ****/
272   //DE1_SoC_testbench tests the expected and unexpected behaviors, reset behaviors
273   // specifically, it tests a Rush Hour
274   // full cycle of scrolling through addresses
275   // and shows a case of no Rush Hour
276
277   `timescale 1 ps / 1 ps
```

```
278
279   module DE1_SoC_tb();
280      logic CLOCK_50;
281      logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
282      logic [3:0] KEY;
283      logic [9:0] SW;
284      logic [9:0] LEDR;
285      wire [35:23] V_GPIO;
286      DE1_SoC dut(.*);
287
288      ///// Testbench Clock Setup/////
289      parameter clock_period = 100;
290      initial begin
291         CLOCK_50 <= 0;
292         forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
293      end
294
295
296      ///// Functions /////
297      // reset cars in lot and time
298      task reset_all; begin
299         SW[2:0] <= 3'b000;
300         KEY[0] <= 0;
301         KEY[3] <= 0; @(posedge CLOCK_50);
302         KEY[3] <= 1; @(posedge CLOCK_50);
303      end endtask
304      // increment hour once
305      task inc_hr; begin
306         KEY[0] <= 0; @(posedge CLOCK_50);
307         KEY[0] <= 1; @(posedge CLOCK_50);
308      end endtask
309      // All Car enter sequence
310      // note: takes 2 clock cycles
311      // per car enter to enable counter
312      task all_in; begin
313         SW[4] <= 1; // open garage
314         SW[0] <= 1; repeat(2)@(posedge CLOCK_50);
315         SW[4] <= 0; @(posedge CLOCK_50);
316         SW[4] <= 1; // open garage
317         SW[1] <= 1; repeat(2)@(posedge CLOCK_50);
318         SW[4] <= 0; @(posedge CLOCK_50);
319         SW[4] <= 1; // open garage
320         SW[2] <= 1; repeat(2)@(posedge CLOCK_50);
321         SW[4] <= 0; @(posedge CLOCK_50);
322      end endtask
323      // All Car exit combination
324      task all_leave; begin
325         SW[0] <= 0; repeat(2)@(posedge CLOCK_50);
326         SW[1] <= 0; repeat(2)@(posedge CLOCK_50);
327         SW[2] <= 0; repeat(2)@(posedge CLOCK_50);
328      end endtask
329
330      // No rush hour: testing display -
331      task no_rush; begin
332         all_leave;
333         repeat(8) inc_hr;
334      end endtask
335
336      initial begin
337         // Case: Rush Hour
338         reset_all;
339         inc_hr;          // rush start: 1
340         all_in;
341         repeat(2)inc_hr; // rush end: 4
342         all_leave;
343         repeat(5)inc_hr; // day end
344         repeat(10)
345         @(posedge CLOCK_50); // cycle through address
346         reset_all;
347         // Case: No Rush Hour
348         no_rush;
349         reset_all;
350         $stop;
```

```
351         end
352         endmodule
353
```

```
/*
Mina Gao and Justin Sim
3/6/2024
EE 371 Hussein
Lab 6

   Display Module:
      Displays a number from 1 - 8 that corresponds to the given count input
      "1" when count = 0
      "8" when count = a MAX parameter in a seperate subsv
      (Note: output needs to manually change depending on MAX parameter)

   inputs:
      clock
      count:      5'b representing number from 0 -25
   outputs:
      [6:0] HEX:  7'b HEX display on DE1_SoC

*/
module display(clk, count, HEX);
   input logic clk;
   input logic [4:0] count;
   output logic [6:0] HEX;

   //for 0-9
   logic [6:0] h0, h1, h2, h3, h4, h5, h6, h7, h8, h9, hOff, hC, hL, hE, hA, hR, hF, hU;

   // assign hex 7'b values that display number 0-9
   assign h0 = 7'b1000000; // 0
   assign h1 = 7'b1111001; // 1
   assign h2 = 7'b0100100; // 2
   assign h3 = 7'b0110000; // 3
   assign h4 = 7'b0011001; // 4
   assign h5 = 7'b0010010; // 5
   assign h6 = 7'b0000010; // 6
   assign h7 = 7'b1111000; // 7
   assign h8 = 7'b0000000; // 8
   assign h9 = 7'b0010000; // 9
   assign hOff = 7'b1111111; // no display

   // assign hex 7'b values that display letters (clear, full)
   assign hC = 7'b1000110; // C
   assign hL = 7'b1000111; // L
   assign hE = 7'b0000110; // E
   assign hA = 7'b0001000; // A
   assign hR = 7'b0101111; // r
   assign hF = 7'b0001110; // F
   assign hU = 7'b1000001; // U

   // assign HEX0-HEX5 display according to the count value
   // count value (0-25) is represented in 5 bit binary
   always_comb begin
      case(count)
         // Start: Hour 1
         5'b00000:
            begin
                  HEX = h0;
            end
         5'b00001:
            begin
                  HEX = h1;
            end
         5'b00010:
            begin
                  HEX = h2;
            end
         5'b00011:
            begin
                  HEX = h3;
            end
         5'b00100:
            begin
                  HEX = h4;
```

```
74              end
75           5'b00101:
76              begin
77                    HEX = h5;
78              end
79           5'b00110:
80              begin
81                    HEX = h6;
82              end
83           5'b00111:
84              begin
85                    HEX = h7;
86              end
87           5'b01000:
88              begin
89                    HEX = h8;
90              end
91           5'b01001:
92              begin
93                    HEX = h9;
94              end
95           5'b01010:
96              begin
97                    HEX = hF;
98              end
99           5'b01011:
100             begin
101                   HEX = hU;
102             end
103          5'b01100:
104             begin
105                   HEX = hL;
106             end
107          default: HEX = hOff;
108       endcase
109    end
110 endmodule
111 //------------------------------------------------------------------------
112 //display_testbench tests the expected and unexpected behaviors, reset behaviors, FULL and
    CLEAR Capacity behavior
113
114 module display_testbench();
115    logic clk;
116    logic [6:0] HEX;
117    logic [4:0] count;
118
119    display dut(.*);
120
121    // clock setup
122    parameter clock_period = 100;
123
124    initial begin
125       clk <= 0;
126       forever #(clock_period / 2) clk <= ~clk;
127    end
128
129    initial begin
130                                          @(posedge clk);
131                                          @(posedge clk);
132                   count = 5'b00000;      @(posedge clk);
133                                          @(posedge clk);
134                   count = 5'b00001;      @(posedge clk);
135                                          @(posedge clk);
136                   count = 5'b00010;      @(posedge clk);
137                                          @(posedge clk);
138                   count = 5'b00011;      @(posedge clk);
139                                          @(posedge clk);
140                   count = 5'b00100;      @(posedge clk);
141                                          @(posedge clk);
142                   count = 5'b00101;      @(posedge clk);
143                                          @(posedge clk);
144                   count = 5'b00110;      @(posedge clk);
145                                          @(posedge clk);
```

```
146                     count = 5'b11001;       @(posedge clk);
147                                             @(posedge clk);
148         $stop; // end simulation
149       end
150   endmodule
```

```
1   /*
2   Justin Sim and Mina Gao
3   3/6/2024
4   EE 371 Hussein
5   Lab 6
6
7      RAM 8 words each of length 16 bits
8      to keep track of number of enters each hour
9
10     Inputs:  CLOCK_50, reset,
11                 datain,  - number of cars entered
12                 rdaddress - given hour (scrolls through 0-7),
13                 wraddress - current hour
14                 wren, rden
15
16     Outputs: dout - number of cars at the given hour
17
18  */
19  module RAM_8x16 (
20      input logic clk,          // Clock input
21      input logic reset,        // Reset input
22      input logic [15:0] datain,// Input data
23      input logic [2:0] rdaddress, // Read address (3 bits for 8 locations)
24      input logic [2:0] wraddress, // Write address (3 bits for 8 locations)
25      input logic wren,         // Write enable
26      input logic rden,         // Read enable
27      output logic [15:0] q     // Output data
28  );
29
30      // Internal storage for 8 words of 16 bits each
31      logic [15:0] memory [7:0];
32
33      // Read operation
34      always_ff @(posedge clk or negedge reset) begin
35          if (reset) begin
36              q <= '0; // Reset output to 0 during reset
37          end else if (rden) begin
38              q <= memory[rdaddress];
39          end
40      end
41
42      // Write operation
43      always_ff @(posedge clk or negedge reset) begin
44          if (reset) begin
45              // Reset memory to 0 during reset
46              memory <= '{default: '0};
47          end else if (wren) begin
48              memory[wraddress] <= datain;
49          end
50      end
51
52  endmodule
53
54  // RAM_8x16_tb tests the expected behaviors of the RAM_8x16 module
55  module RAM_8x16_tb;
56
57      // Parameters
58      parameter WIDTH = 16;  // Data width
59      parameter DEPTH = 8;   // Depth of RAM
60
61      // Signals
62      logic clk;
63      logic reset;
64      logic [WIDTH-1:0] datain;
65      logic [2:0] rdaddress;
66      logic [2:0] wraddress;
67      logic wren;
68      logic rden;
69      logic [WIDTH-1:0] q;
70
71      // Instantiate the RAM module
72      RAM_8x16 dut (
73          .clk(clk),
```

```
74            .reset(reset),
75            .datain(datain),
76            .rdaddress(rdaddress),
77            .wraddress(wraddress),
78            .wren(wren),
79            .rden(rden),
80            .q(q)
81        );
82
83        // Clock generation
84        always #5 clk = ~clk;
85
86        // Test stimulus
87        initial begin
88            clk = 0;
89            reset = 1;
90            wren = 0;
91            rden = 0;
92            wraddress = 0;
93            rdaddress = 0;
94            datain = 0;
95
96            #10 reset = 0; // Release reset
97
98            // Write data to RAM
99            wraddress = 0;
100           datain = 16'd69;
101           wren = 1;
102           #20;
103           wren = 0;
104
105           // Read data from RAM
106           rdaddress = 0;
107           rden = 1;
108           #20;
109           rden = 0;
110
111           // Add more test cases here...
112
113           // Finish simulation
114           #10 $stop;
115       end
116
117   endmodule
118
```