Khoa Tran
EE 371
April 23, 2021
Lab 2 Report

Procedure

Task #1

Approaching this problem, I first drew up the block diagram to figure out the necessary inputs of the 32x4 ram and how the clock progresses each input into the ram, allowing it to output the data from the address given by SW8-4. The ram only has one address input, both for writing and reading. As the write signal is not true, the ram will output the data at the given address, which will initially be 0. However, as write signal is true, given by SW9, then the ram will write the given data, given by SW3-0, to the address location given and output that same value as a read. Block diagram shows that input address is shown on HEX5, HEX4, data on HEX2, and dataOut on HEX0.

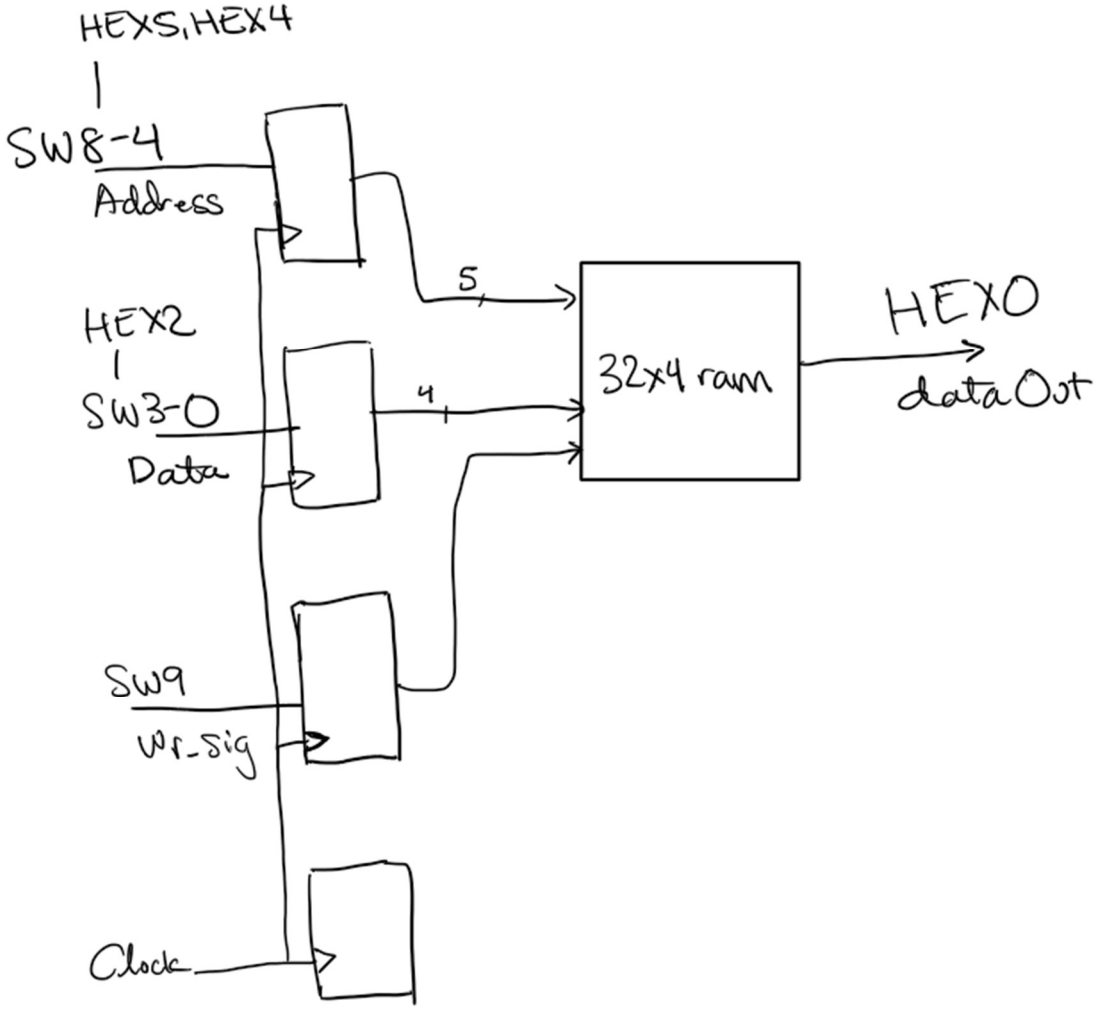

Figure 1: Block diagram for task 1

Task #2

Approaching this problem, I first drew up block diagram for the 32x4 ram as the ram now has an input for read address and write address. The read address is taken from a counter that is implemented by incrementing the value of the address by 1, for every 1 second, running in 1 Hz clock. This can be seen in figure 3. The ram also takes the input of SW8-4 for the write address, and SW3-0 as the data input and KEY[3] as the write signal. The ram inputs the data from the given address, running on CLOCK_50. The output data of the 32x4 ram is the data at the read address of the counter. Using the IP Catalog, I was able to set the memory initialization of the ram so the output data would be different than all zeros to start with, this output can be seen on HEX0. Write address can be seen on HEX5 and HEX4, while read address on HEX3 and HEX2 and input data on HEX1.

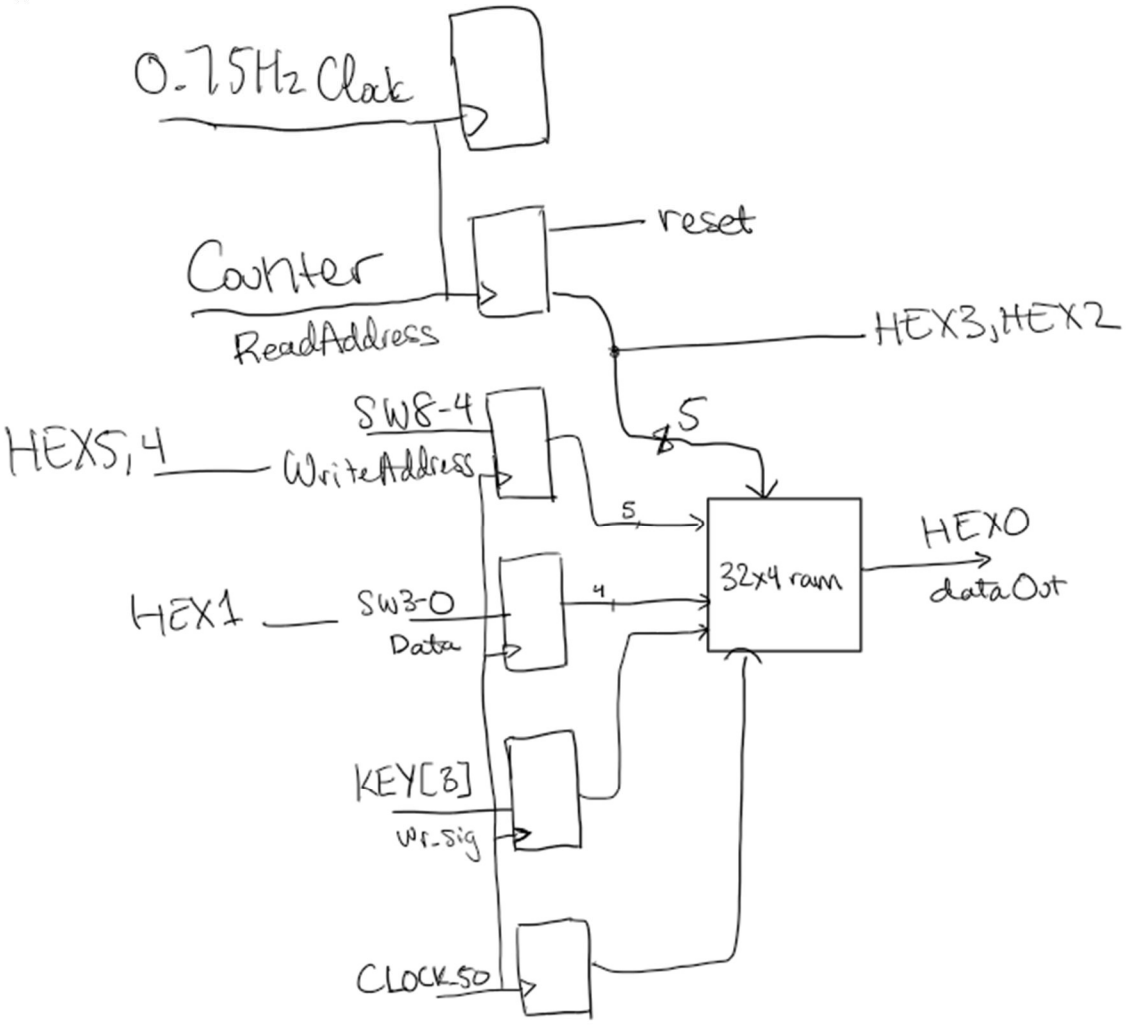

Figure 2: Block diagram for task 2

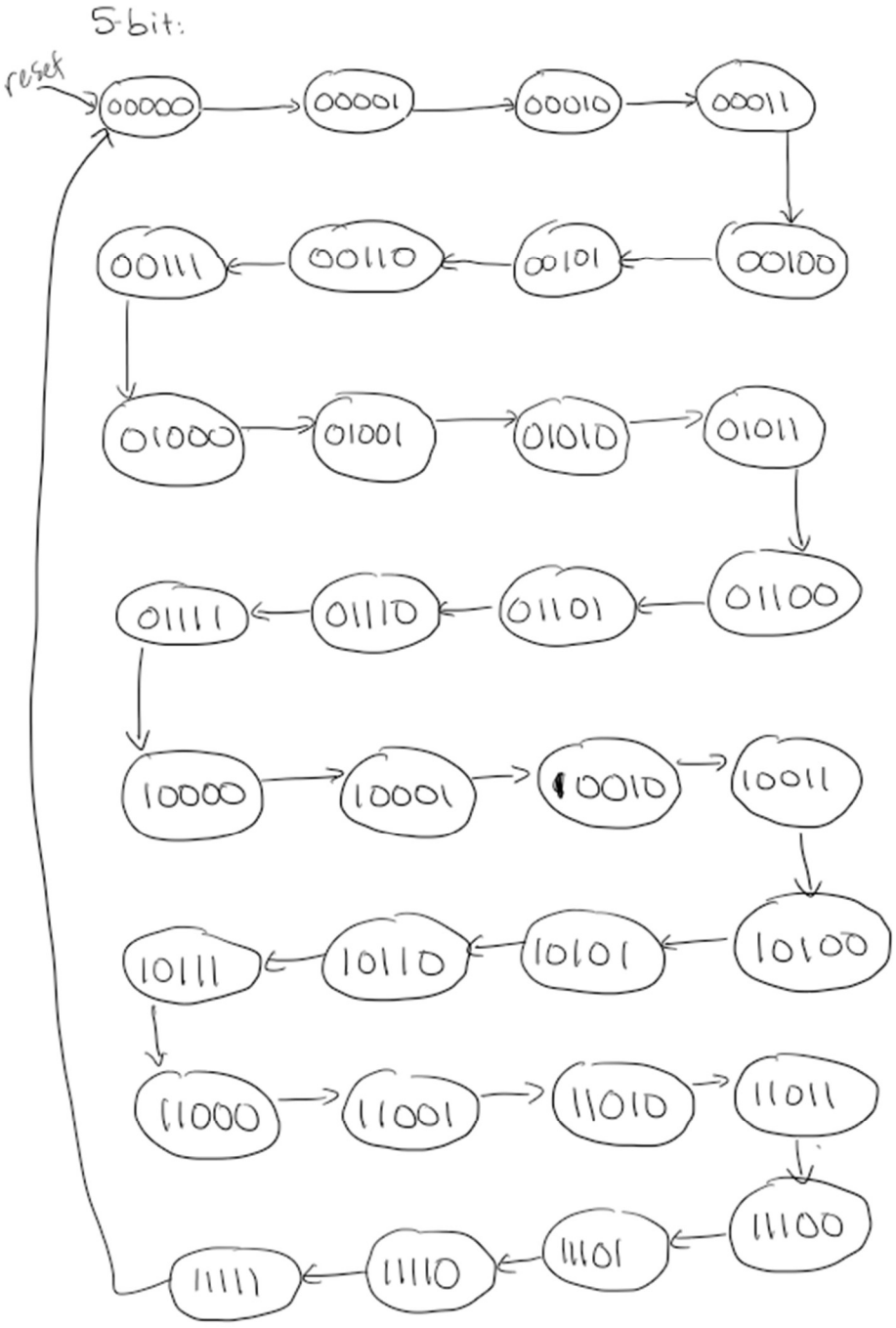

Figure 3: State Diagram for Counter

Task #3

Approaching this problem, I first drew the block diagram to design the interaction between the FIFO controller and the 16x8 ram as the ram requires read and write address as well as write

enable from the FIFO as it determines the location to write and read based on a queue, implementing the idea of first in first out. Since the ram has to read the least recent data that is written to the ram, the FIFO controller takes in inputs of write and read in order to determine if the user wants to write or read and where the respective addresses of where the ram should store the data to continue the queue and where the read address has to be to maintain least recent read. To figure out the FIFO controller, I drew a state diagram showing the different states of the ram and how the different inputs of write and read from SW8 and SW9 affects the output of read and write pointer as the output address for the ram's input and as well as full and empty. This can be seen in figure 5.

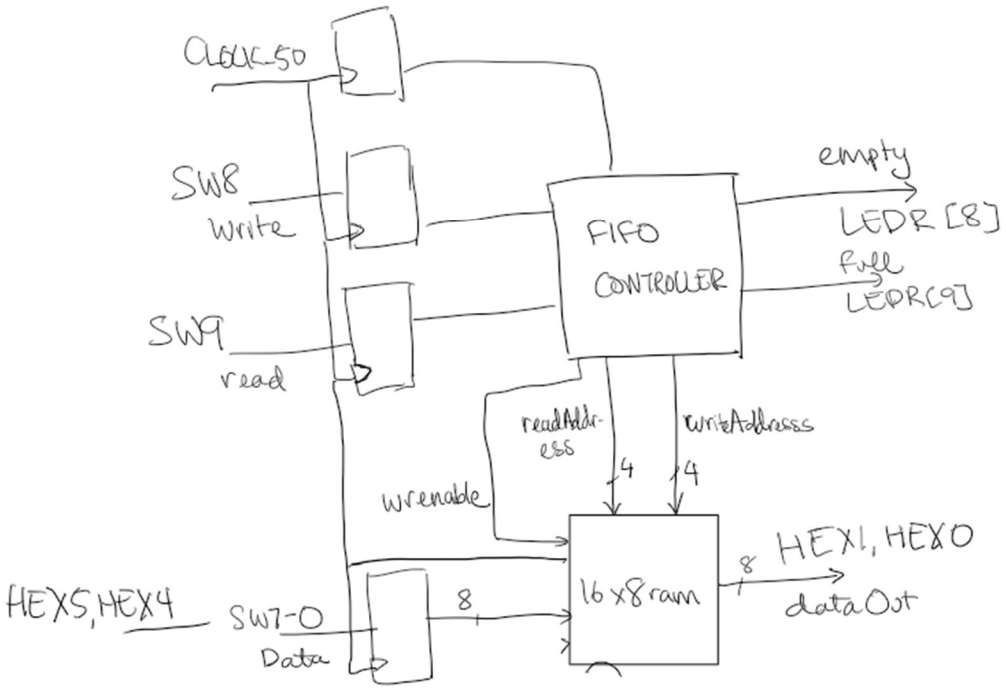

Figure 4: Block diagram for task 3

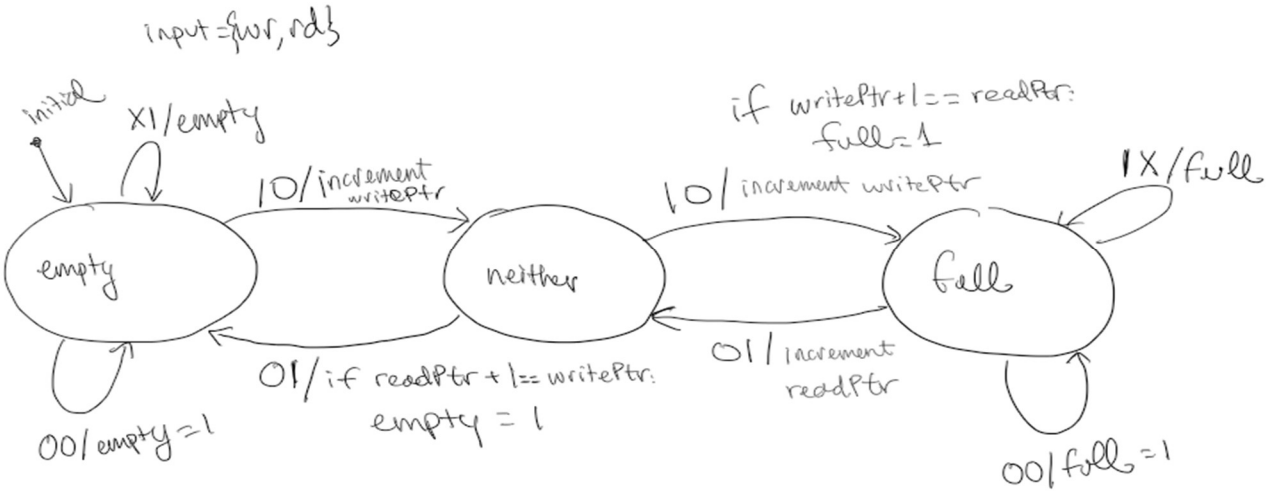

Figure 5: FSM for FIFO Controller

Results

Task 1:

For the first part, I tested if the ram file module as the it goes through a series of inputs on in_data, addr, and wr_sig, seeing if the out_data is correct after each write into the ram. The inputs are given to the ram at the positive edge of the clock, while out_data will always output the data at the given address.

Figure 6: Waveform simulation for ram_file

I also created two files that follows combinational logic, outputing to hex display the hexadecimal value of the address and data given. The waveform simulation is below as the output for the hex display matches the value of the data or address given, seen in figure 7 and 8.

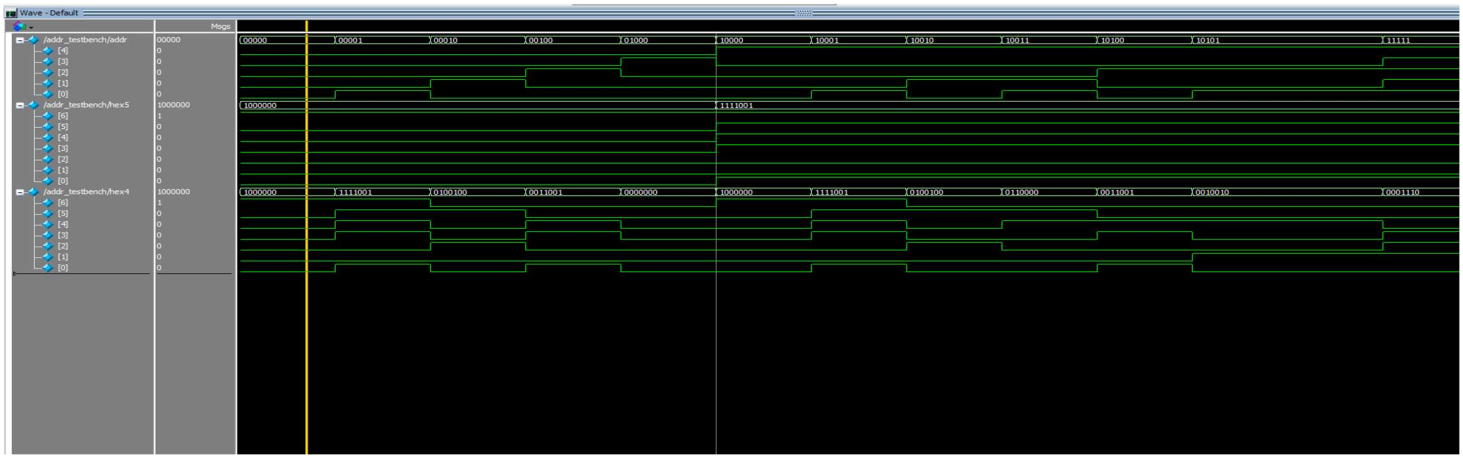

Figure 7: Waveform for display address

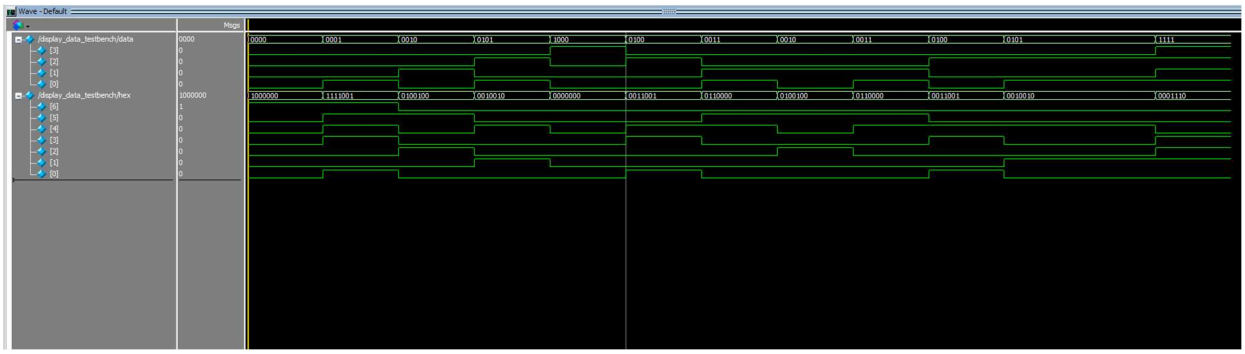

Figure 8: Waveform simulation for display data

For DE1_SoC, The address is taken from SW8-4, as the data is taken from SW3-0. The signal to write to the ram is given from SW9. As the clk that is set to KEY[0] goes to posedge, the ram writes the given data into the given address. However, the output on HEX0 will be the data at the address given by SW8-4. As seen in the simulation in figure 9, HEX5 and HEX4 displays the address value using hexadecimal and HEX2 for input data.

Figure 9: The waveform simulation generated by the DE1_SoC for task1

Task 2:

This task replicated the exact same ram as the previous task with the size of 32x4, however, there are two address inputs, one for the write address and one for the read address. The write address and the data is given from the user using the switches. While the read address is taken from a counter, incrementing by 1 every second. As the output data from the ram is from the read address location. As seen below in figure 10, the counter is incrementing by 1 in order to output

the read address for the ram to receive and output the data.

Figure 10: The waveform simulation generated by counter module

I also created two files that follows combinational logic, outputing to hex display the hexadecimal value of the address and data given. The waveform simulation is below as the output for the hex display matches the value of the data or address given, seen in figure 11 and 12, similar to modules in task 1.

Figure 11: Waveform simulation generated by display data 2 module

Figure 12: Waveform simulation generated by addr2 module

Below in figure 13 is the waveform simulation for ram32x4 testing for the output of q, as the inputs of rdaddress, wraddress, data, and wren testing if the q output is correct based on the given read address as the data that outputs is the value from the memory initialization file, for the initial memory of the ram, seen in figure 14.

Figure 13: Waveform simulation generated by ram32x4 module

```
WIDTH=4;
DEPTH=32;

ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
    00  :   0110;
    01  :   0001;
    02  :   0000;
    03  :   0001;
    04  :   0000;
    05  :   0001;
    06  :   0000;
    07  :   0001;
    08  :   0010;
    09  :   0011;
    0A  :   0100;
    0B  :   0101;
    0C  :   0110;
    0D  :   0111;
    0E  :   1000;
    0F  :   1001;
    10  :   1010;
    11  :   1011;
    12  :   0001;
    13  :   0011;
    14  :   0100;
    15  :   0110;
    16  :   1000;
    17  :   1001;
    18  :   1101;
    19  :   1100;
    1A  :   1111;
    1B  :   1100;
    1C  :   0011;
    1D  :   0101;
    1E  :   0111;
    1F  :   0011;
END;
```

Figure 14: ram32x4 memory initialization file

Below is the simulation for DE1_SoC for task2, as SW8-4 is used for the write address and SW3-0 is used for the data displayed on HEX5, HEX4, and HEX1. The KEY0 is used as reset input and KEY 3 is used for wr_en. As seen on HEX0, the ram outputs the data and is displayed on the 7-segment in hexadecimal and the address on HEX3 and HEX2.

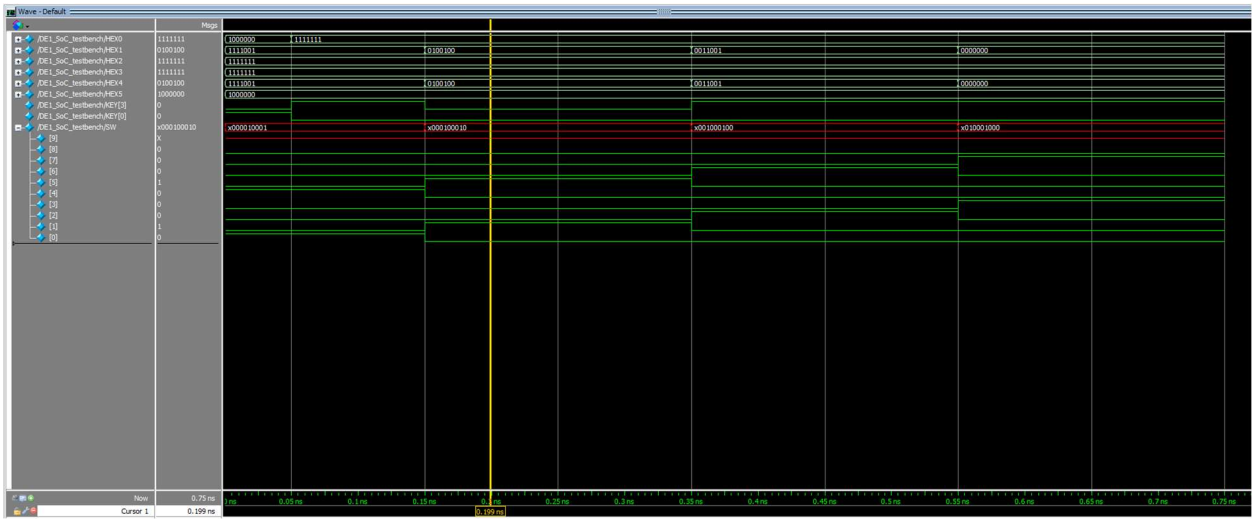

Figure 15: Waveform simulation for DE1_SoC for task 2

Task 3:

This task implements a concept of a queue for the ram, having data being first in, first out. As in the ram will only read the least recent data when the user desires to. As well as only writes when the ram isn't full and the user wants to write. In the simulation below, the FIFO control is being tested in a series of inputs on read, write, and seeing if wr_en, full, empty, write address, and read address is correct along each positive edge of the clock. As seen in the simulation in figure 16, the outputs follow a finite state machine that I designed in the procedure, as the state of empty is initially true and progressing through a series of writes in order for full to output true.

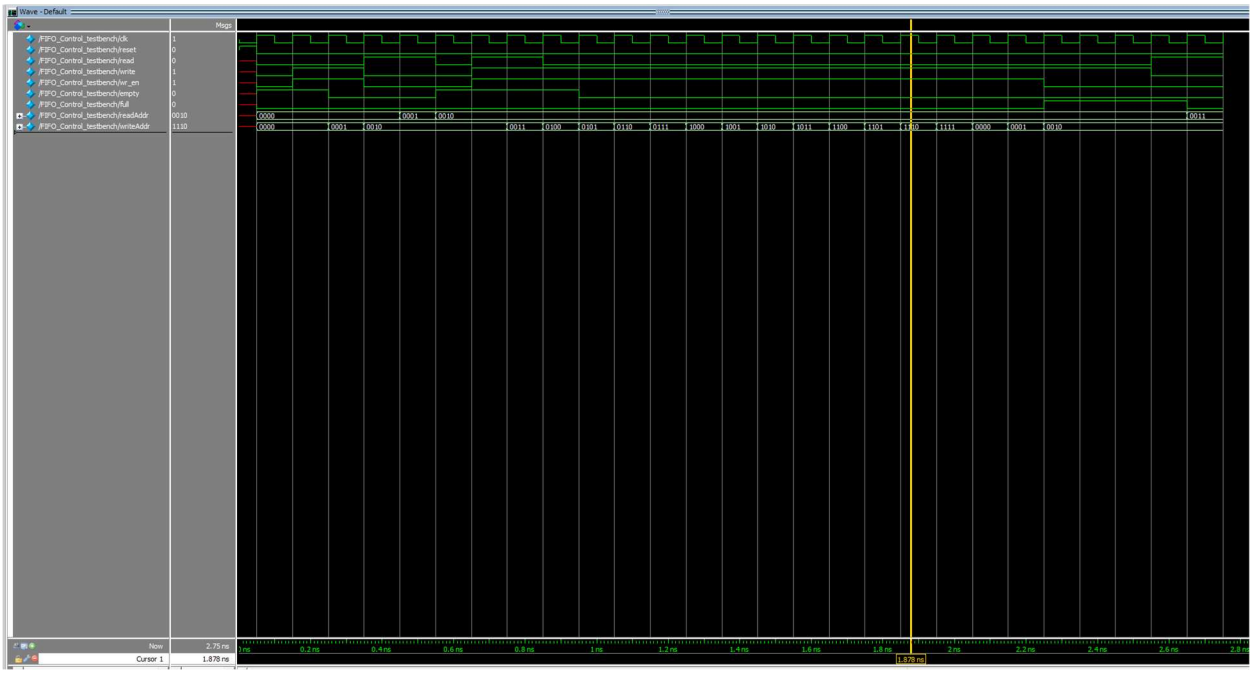

Figure 16: Waveform simulation for FIFO Control module

The simulation below is for the 16x8 ram as test for the output of q, as the inputs of rdaddress, wraddress, data, and wren testing if the q output is correct based on the given read address as the data that outputs is the value from the memory initialization file, for the initial memory of the ram, as it is empty to start with.

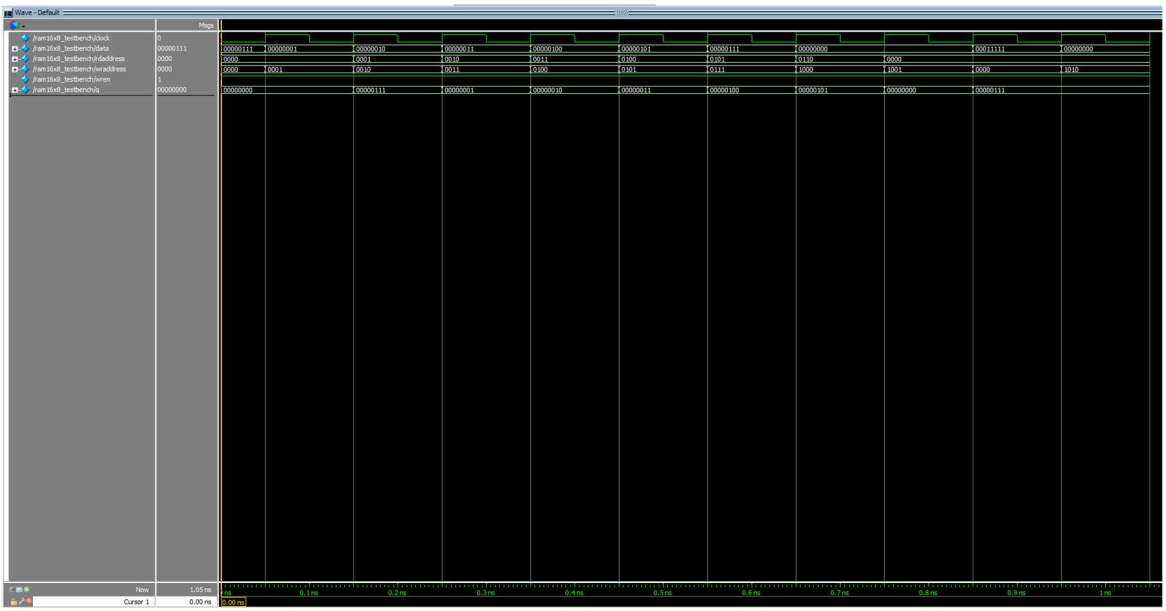

Figure 17: Waveform simulation generated by ram16x8 module

As well as modules for displaying the data of the input and the output data, the addition with the output data module is that it implements SW9, only outputting the value if SW9 is also true as the user intends to read.

Figure 18: Waveform simulation generated by data2 module

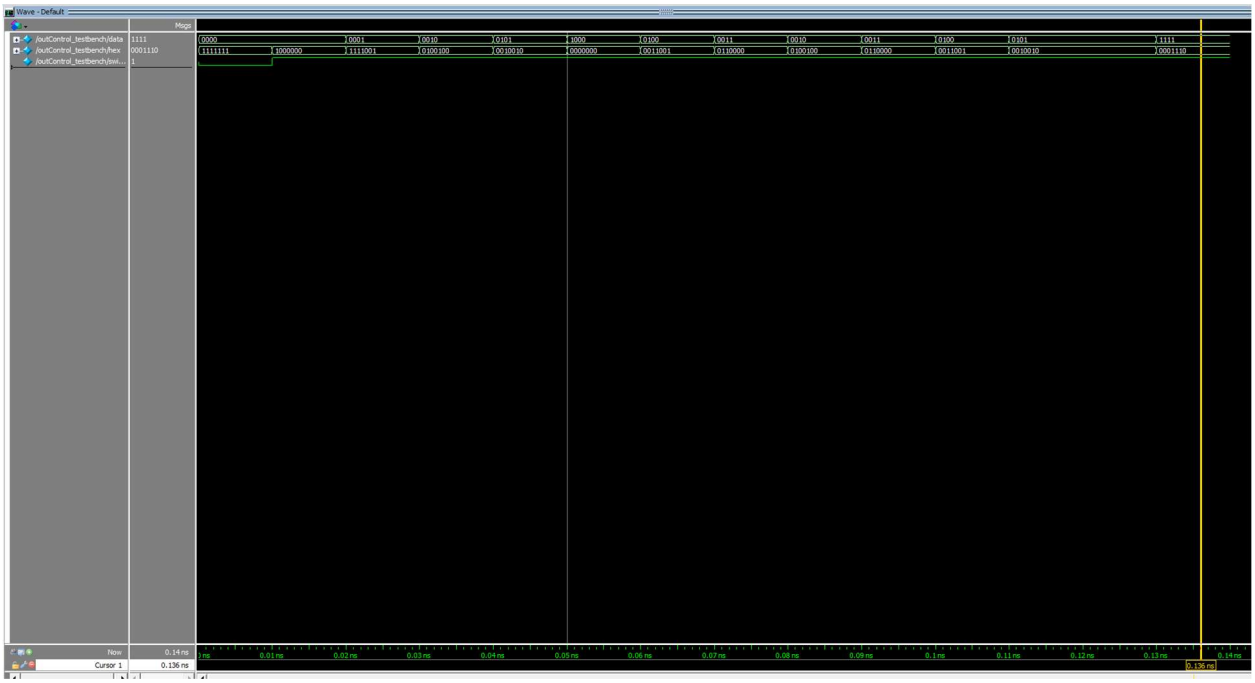

Figure 19: Waveform simulation generated by outControl module

For the simulation of DE1_SoC, the SW7-0 is the data input as SW8 and SW9 is the write and read inputs for the FIFO, as seen in figure 20, the current data output of the FIFO is on HEX1-0, and LEDR9 shows full, and LEDR8 shows empty. As well as HEX5-4 displaying the value of input data in hexadecimal. As the inputs of SW8 and 9, the ram will in written in the order of a queue, implementing first in first out for read data.

Figure 20: Waveform simulation generated by DE1_SoC for task 3

Final Product

The overarching goal of this project was to design a ram that was able to read and write based on given data and write and read addresses. The project goes further in depth as task 3 implements a system of storing and outputting data that was in a first in first out basis like a queue. The goal was to create a FIFO controller that is a finite state machine, with inputs read and write determining the output and the current state of the state. Overall, I was able to complete the project and done what was asked. There wasn't any issues and everything that was asked in each task was completed to the expectation and simulation and demonstration of the lab allowed me to understand the transitions between modules and systems allowing the finish product to be similar to what is asked.

Appendix: SystemVerilog Code

1) ram_file.sv (task 1)

```
1  //Khoa Tran
2  //04/22/2021
3  //Lab 2, Task 1
4  //This module is a ram file implemented as a 32x4 size ram, with inputs of clk, wr_sig as the
5  //write signal, in_data as the input data, and out_data as the output data.
6  //On the posedge clock, with wr_sig, the ram will write the given data to the given address
7  //in the ram. The output data will always read from the given address.
8  module ram_file #(parameter data_width = 4, addr_width = 5) (clk, wr_sig, in_data, addr, out_data);
9     input logic clk;
10    input logic wr_sig;
11    input logic [data_width - 1:0] in_data;
12    input logic [addr_width - 1:0] addr;
13    output logic [data_width - 1:0] out_data;
14
15    //ram logic
16    logic [data_width -1:0] memory_array [0:2**addr_width - 1];
17
18    //sequential logic, writting data to the given address when wr_sig is true
19    always_ff @(posedge clk)
20       begin
21          if (wr_sig)
22             memory_array[addr] <= in_data;
23       end
24    //output data is the data at the given address in the ram
25    assign out_data = memory_array[addr];
26 endmodule
27
28 //This module is a testbench for the ram_file, testing for inputs
29 //in_data, wr_sig, and addr provides the correct output of the out_data
30 //at the posedge clk.
31 module ram_file_testbench#(parameter data_width = 4, addr_width = 5)();
32    logic clk, wr_sig;
33    logic [data_width - 1:0] in_data, out_data;
34    logic [addr_width - 1:0] addr;
35
36    //device under test
37    ram_file dut(.clk, .wr_sig, .in_data, .addr, .out_data);
38
39    //clock setup
40    parameter clock_period = 100;
41
42    initial begin
43       clk <= 0;
44       forever #(clock_period /2) clk <= ~clk;
45    end
46    //initial simulation
47    initial begin
48       in_data<=4'b0000; wr_sig<=1'b0;  addr<=5'b00000;   @(posedge clk);
49       in_data<=4'b0001; wr_sig<=1'b1;  addr<=5'b00000;   @(posedge clk);
50       #10;
51       in_data<=4'b0010; wr_sig<=1'b0;  addr<=5'b00001;   @(posedge clk);
52       in_data<=4'b0011; wr_sig<=1'b1;  addr<=5'b00001;   @(posedge clk);
53       #10;
54       in_data<=4'b0011; wr_sig<=1'b0;  addr<=5'b00010;   @(posedge clk);
55       in_data<=4'b0100; wr_sig<=1'b1;  addr<=5'b00011;   @(posedge clk);
56       #10;
57       in_data<=4'b0101; wr_sig<=1'b0;  addr<=5'b00100;   @(posedge clk);
58       in_data<=4'b0110; wr_sig<=1'b1;  addr<=5'b00101;   @(posedge clk);
59       #10;
60       $stop; //end simulation
61    end
62 endmodule
63
```

2) DE1_SoC.sv (task 1)

```
1   //Khoa Tran
2   //04/22/2021
3   //Lab 2, Task 1
4   //This module implements a 32x4 ram that takes in data values and an address
5   //and displays the data in hexadecimal on hex2 display as well as the address
6   //in hexadecimal on hex5 and hex4 display. The address is taken from SW8-4, as the
7   //data is taken from SW3-0. The signal to write to the ram is given from SW9.
8   //As the clk that is set to KEY[0] goes to posedge, the ram writes the given data
9   //into the given address. This module continously outputs the current value on
10  //HEX0 in hexadecimal of the ram at the given address from SW3-0.
11  module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
12     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
13     output logic [9:0] LEDR;
14     input logic [3:0] KEY;
15     input logic [9:0] SW;
16
17     logic [3:0] out;
18
19     //instantiating addr module passing SW8-4 to address and outputs to HEX5 and HEX4
20     addr display_addr(.addr(SW[8:4]), .hex5(HEX5), .hex4(HEX4));
21     //instantiating display_data module passing SW3-0 to data and outputs to HEX2
22     display_data data1(.data(SW[3:0]), .hex(HEX2));
23     //instantiating ram_file module passing SW9 as wr_sig, KEY[0] as clk, in data as SW3-0 and
24     //SW8-4 as address, and output the data to logic out
25     ram_file ram(.clk(~KEY[0]), .wr_sig(SW[9]), .in_data(SW[3:0]), .addr(SW[8:4]), .out_data(out));
26     //instantiating display_data to output to HEX0 the out from ram_file
27     display_data data2(.data(out), .hex(HEX0));
28
29     assign HEX1 = 7'b1111111;
30     assign HEX3 = 7'b1111111;
31
32  endmodule
33
34  //Testing the DE1_SoC module by going through a series of inputs on KEY[0] as the
35  //clk and SW[9] as the wr_sig, and SW[8:0] representing the address and the data
36  //seeing if HEX5, HEX4, HEx2, and HEX0 is correct.
37  module DE1_SoC_testbench();
38     logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
39     logic [9:0] LEDR;
40     logic [3:0] KEY;
41     logic [9:0] SW;
42
43     DE1_SoC dut(.HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW);
44
45     integer i;
46     initial begin
47        KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8:4] <= 5'b00001; SW[3:0] <= 4'b0001; #10;
48        KEY[0] <= 1'b0; SW[9] <= 1'b1; SW[8:4] <= 5'b00001; SW[3:0] <= 4'b0001; #10;
49        KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8:4] <= 5'b00010; SW[3:0] <= 4'b0010; #10;
50        KEY[0] <= 1'b0; SW[9] <= 1'b0; SW[8:4] <= 5'b00010; SW[3:0] <= 4'b0010; #10;
51        KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[8:4] <= 5'b00100; SW[3:0] <= 4'b0100; #10;
52        KEY[0] <= 1'b0; SW[9] <= 1'b1; SW[8:4] <= 5'b00100; SW[3:0] <= 4'b0100; #10;
53        KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[8:4] <= 5'b01000; SW[3:0] <= 4'b1000; #10;
54        KEY[0] <= 1'b0; SW[9] <= 1'b1; SW[8:4] <= 5'b01000; SW[3:0] <= 4'b1000; #10;
55        $stop;
56     end
57
58  endmodule
59
60
```

3) display_data.sv (task 1)

```
1   //Khoa Tran
2   //04/22/2021
3   //Lab 2. Task 1
4   //This module controls the logic case for displaying the hex value of the data
5   //on the hex 7-seg display board. With inputs of data as 4 bits, combinational
6   //logic through the cases to have the hex output that matches the input data value
7   module display_data#(parameter data_width = 4) (data, hex);
8      input logic [data_width - 1:0] data;
9      output logic [6:0] hex;
10
11     //logic for 7-seg display for hex values 1-15
12     logic [6:0] zero, one, two, three, four, five, six, seven, eight, nine, A, B, C, D, E, F;
13     assign zero = 7'b1000000;
14     assign one = 7'b1111001;
15     assign two = 7'b0100100;
16     assign three = 7'b0110000;
17     assign four = 7'b0011001;
18     assign five = 7'b0010010;
19     assign six = 7'b0000010;
20     assign seven = 7'b1111000;
21     assign eight = 7'b0000000;
22     assign nine = 7'b0010000;
23     assign A = 7'b0001000;
24     assign B = 7'b0000011;
25     assign C = 7'b1000110;
26     assign D = 7'b0100001;
27     assign E = 7'b0000110;
28     assign F = 7'b0001110;
29
30     //combinational for cases of data values 1-15 to display in hex
31     always_comb begin
32             case (data)
33                4'b0000: begin
34                         hex = zero;
35                         end
36                4'b0001: begin
37                         hex = one;
38                         end
39                4'b0010: begin
40                         hex = two;
41                         end
42                4'b0011: begin
43                         hex = three;
44                         end
45                4'b0100: begin
46                         hex = four;
47                         end
48                4'b0101: begin
49                         hex = five;
50                         end
51                4'b0110: begin
52                         hex = six;
53                         end
54                4'b0111: begin
55                         hex = seven;
56                         end
57                4'b1000: begin
58                         hex = eight;
59                         end
60                4'b1001: begin
61                         hex = nine;
62                         end
63                4'b1010: begin
64                         hex = A;
65                         end
66                4'b1011: begin
67                         hex = B;
68                         end
69                4'b1100: begin
70                         hex = C;
71                         end
72                4'b1101: begin
73                         hex = D;
74                         end
75                4'b1110: begin
76                         hex = E;
```



```
77                         end
78                4'b1111: begin
79                         hex = F;
80                         end
81                default: begin
82                         hex = 7'b1111111;
83                         end
84                endcase
85        end
86  endmodule
87
88  //This module is a testbench for displaying the data as it goes through
89  //the given input data points and test for if the hex output matches the
90  //given input value in hex for the 7-seg display board
91  module display_data_testbench #(parameter data_width = 4)();
92
93     logic [data_width - 1:0] data;
94     logic [6:0] hex;
95
96     //devices under test
97     display_data dut(.data, .hex);
98
99     //initial simulation
100    initial begin
101       data<=4'b0000; #10;
102       data<=4'b0001; #10;
103       data<=4'b0010; #10;
104       data<=4'b0101; #10;
105       data<=4'b1000; #10;
106       data<=4'b0100; #10;
107       data<=4'b0011; #10;
108       data<=4'b0010; #10;
109       data<=4'b0011; #10;
110       data<=4'b0100; #10;
111       data<=4'b0101; #10;
112       data<=4'b0101; #10;
113       data<=4'b1111; #10;
114
115       $stop; //end simulation
116    end
117 endmodule
118
```

4) addr.sv

```
1   //Khoa Tran
2   //04/22/2021
3   //Lab 2, Task 1
4   //This module controls the logic case for displaying the hex value of the address
5   //on the hex 7-seg display board. With inputs of address as 5 bits, combinational
6   //logic through the cases to have the hex output that matches the input data value
7   module addr#(parameter addr_width = 5) (addr, hex5, hex4);
8      input logic [addr_width - 1:0] addr;
9      output logic [6:0] hex5, hex4;
10
11     //logic for 7-seg display for hex values 1-5
12     logic [6:0] zero, one, two, three, four, five, six, seven, eight, nine, A, B, C, D, E, F;
13     assign zero = 7'b1000000;
14     assign one = 7'b1111001;
15     assign two = 7'b0100100;
16     assign three = 7'b0110000;
17     assign four = 7'b0011001;
18     assign five = 7'b0010010;
19     assign six = 7'b0000010;
20     assign seven = 7'b1111000;
21     assign eight = 7'b0000000;
22     assign nine = 7'b0010000;
23     assign A = 7'b0001000;
24     assign B = 7'b0000011;
25     assign C = 7'b1000110;
26     assign D = 7'b0100001;
27     assign E = 7'b0000110;
28     assign F = 7'b0001110;
29
30     //combinational for cases of addr values 1-31 to display in hex5-4
31     always_comb begin
32              case (addr)
33                 5'b00000: begin
34                           hex5 = zero;
35                           hex4 = zero;
36                           end
37                 5'b00001: begin
38                           hex5 = zero;
39                           hex4 = one;
40                           end
41                 5'b00010: begin
42                           hex5 = zero;
43                           hex4 = two;
44                           end
45                 5'b00011: begin
46                           hex5 = zero;
47                           hex4 = three;
48                           end
49                 5'b00100: begin
50                           hex5 = zero;
51                           hex4 = four;
52                           end
53                 5'b00101: begin
54                           hex5 = zero;
55                           hex4 = five;
56                           end
57                 5'b00110: begin
58                           hex5 = zero;
59                           hex4 = six;
60                           end
61                 5'b00111: begin
62                           hex5 = zero;
63                           hex4 = seven;
64                           end
65                 5'b01000: begin
66                           hex5 = zero;
67                           hex4 = eight;
68                           end
69                 5'b01001: begin
70                           hex5 = zero;
71                           hex4 = nine;
72                           end
73                 5'b01010: begin
74                           hex5 = zero;
75                           hex4 = A;
76                           end
```

```
77                          hex5 = zero;
78                          hex4 = B;
79                          end
80                  5'b01100: begin
81                          hex5 = zero;
82                          hex4 = C;
83                          end
84                  5'b01101: begin
85                          hex5 = zero;
86                          hex4 = D;
87                          end
88                  5'b01110: begin
89                          hex5 = zero;
90                          hex4 = E;
91                          end
92                  5'b01111: begin
93                          hex5 = zero;
94                          hex4 = F;
95                          end
96                  5'b10000: begin
97                          hex5 = one;
98                          hex4 = zero;
99                          end
100                 5'b10001: begin
101                         hex5 = one;
102                         hex4 = one;
103                         end
104                 5'b10010: begin
105                         hex5 = one;
106                         hex4 = two;
107                         end
108                 5'b10011: begin
109                         hex5 = one;
110                         hex4 = three;
111                         end
112                 5'b10100: begin
113                         hex5 = one;
114                         hex4 = four;
115                         end
116                 5'b10101: begin
117                         hex5 = one;
118                         hex4 = five;
119                         end
120                 5'b10110: begin
121                         hex5 = one;
122                         hex4 = six;
123                         end
124                 5'b10111: begin
125                         hex5 = one;
126                         hex4 = seven;
127                         end
128                 5'b11000: begin
129                         hex5 = one;
130                         hex4 = eight;
131                         end
132                 5'b11001: begin
133                         hex5 = one;
134                         hex4 = nine;
135                         end
136                 5'b11010: begin
137                         hex5 = one;
138                         hex4 = A;
139                         end
140                 5'b11011: begin
141                         hex5 = one;
142                         hex4 = B;
143                         end
144                 5'b11100: begin
145                         hex5 = one;
146                         hex4 = C;
147                         end
148                 5'b11101: begin
149                         hex5 = one;
150                         hex4 = D;
151                         end
152                 5'b11110: begin
```

```
153                             hex5 = one;
154                             hex4 = E;
155                             end
156                     5'b11111: begin
157                             hex5 = one;
158                             hex4 = F;
159                             end
160                     default: begin
161                             hex5 = 7'b1111111;
162                             hex4 = 7'b1111111;
163                             end
164                     endcase
165         end
166     endmodule
167
168     //This module is a testbench for displaying the address as it goes through
169     //the given input address points and test for if the hex output matches the
170     //given address value in hex for the 7-seg display board
171     module addr_testbench#(parameter addr_width = 5)();
172        logic [addr_width - 1:0] addr;
173        logic [6:0] hex5, hex4;
174
175        //devices under test
176        addr dut(.addr, .hex5, .hex4);
177
178        //initial simulation
179        initial begin
180           addr<=5'b00000;   #10;
181           addr<=5'b00001;   #10;
182           addr<=5'b00010;   #10;
183           addr<=5'b00100;   #10;
184           addr<=5'b01000;   #10;
185           addr<=5'b10000;   #10;
186           addr<=5'b10001;   #10;
187           addr<=5'b10010;   #10;
188           addr<=5'b10011;   #10;
189           addr<=5'b10100;   #10;
190           addr<=5'b10101;   #10;
191           addr<=5'b10101;   #10;
192           addr<=5'b11111;   #10;
193           $stop; //end simulation
194        end
195     endmodule
196
```

5) counter.sv (task 2)

```
1  //Khoa Tran
2  //04/22/2021
3  //Lab 2, Task 2
4  //This module implements a counter that increments by 1 at every posedge
5  //clk, if there's a reset, then the count goes back to 0.
6  //As the module takes input reset, clk, and outputs count.
7  module counter#(parameter width = 5) (reset, clk, count);
8     input logic reset, clk; //input reset, and clk
9     output logic [width - 1:0] count;
10
11    //sequential (DFFS) increment count or if reset, count goes back to 0
12    always_ff @(posedge clk)
13    begin
14       if (reset)
15          count <= 0;
16       else
17          count <= count + 1;
18    end
19 endmodule
20
21 //Module test the output of the counter module by running a sequence of inputs
22 //on reset if the output of count is incrementing at each posedge clk
23 module counter_testbench#(parameter width = 5)();
24
25    logic reset, clk;
26    logic [width - 1:0] count;
27
28    //devices under test
29    counter dut (.reset, .clk, .count);
30
31    //clock setup
32    parameter clock_period = 100;
33
34    initial begin
35       clk <= 0;
36       forever #(clock_period /2) clk <= ~clk;
37    end
38    //initial simulation
39    initial begin
40       reset <= 1;          @(posedge clk);
41       reset <= 0;          @(posedge clk);
42       reset <= 0;          @(posedge clk);
43       reset <= 0;          @(posedge clk);
44       reset <= 0;          @(posedge clk);
45       reset <= 0;          @(posedge clk);
46       reset <= 0;          @(posedge clk);
47       reset <= 0;          @(posedge clk);
48       reset <= 0;          @(posedge clk);
49       reset <= 0;          @(posedge clk);
50       reset <= 0;          @(posedge clk);
51       reset <= 0;          @(posedge clk);
52       reset <= 0;          @(posedge clk);
53
54       $stop; //end simulation
55    end
56 endmodule
57
```

6) ram32x4_testbench.sv (task 2)

```
//Khoa Tran
//04/22/2021
//Lab 2, Task 2
//Module testing the ram32x4 by entering a series of inputs on data, rdaddress, wraddress, and wren
//as well to see if the output on q is correct on each posedge clock.
`timescale 1 ps / 1 ps
module ram32x4_testbench();
   reg   clock;
   reg   [3:0]  data;
   reg   [4:0]  rdaddress;
   reg   [4:0]  wraddress;
   reg    wren;
   wire  [3:0]  q;

   ram32x4 dut(.clock(clock), .data(data), .rdaddress(rdaddress), .wraddress(wraddress), .wren(wren), .q(q));

   //clock setup
   parameter clock_period = 100;

   initial begin
      clock <= 0;
       forever #(clock_period /2) clock <= ~clock;
   end
   //initial simulation
   initial begin
      data <= 4'b0111;  rdaddress <= 5'b00000; wraddress <= 5'b00000; wren<=1;   @(posedge clock);
      data <= 4'b0001;  rdaddress <= 5'b00000; wraddress <= 5'b00001; wren<=1;   @(posedge clock);
      data <= 4'b0010;  rdaddress <= 5'b00001; wraddress <= 5'b00010; wren<=1;   @(posedge clock);
      data <= 4'b0011;  rdaddress <= 5'b00010; wraddress <= 5'b00011; wren<=1;   @(posedge clock);
      data <= 4'b0100;  rdaddress <= 5'b00011; wraddress <= 5'b00100; wren<=1;   @(posedge clock);
      data <= 4'b0101;  rdaddress <= 5'b00100; wraddress <= 5'b00101; wren<=1;   @(posedge clock);
      data <= 4'b0111;  rdaddress <= 5'b00101; wraddress <= 5'b00111; wren<=1;   @(posedge clock);
      data <= 4'b0000;  rdaddress <= 5'b00110; wraddress <= 5'b01000; wren<=1;   @(posedge clock);
      data <= 4'b0000;  rdaddress <= 5'b00000; wraddress <= 5'b01001; wren<=1;   @(posedge clock);
      data <= 4'b1111;  rdaddress <= 5'b00000; wraddress <= 5'b00000; wren<=1;   @(posedge clock);
      data <= 4'b0000;  rdaddress <= 5'b00000; wraddress <= 5'b01010; wren<=1;   @(posedge clock);

      $stop; //end simulation
   end
endmodule
```

7) DE1_SoC (task 2)

```
1   //Khoa Tran
2   //04/22/2021
3   //Lab 2, Task 2
4   //This module implements a 32x4 ram that takes in data values and an address
5   //and displays the data in hexadecimal on hex1 display as well as the address
6   //in hexadecimal on hex5 and hex4 display. The address is taken from SW8-4, as the
7   //data is taken from SW3-0. The signal to write to the ram is given from KEY3.
8   //As the clk goes to posedge, the ram writes the given data
9   //into the given address if the wren is true. The output of the ram is the data
10  //at the address given from the counter. As each posedge clk, the counter increments
11  //by 1, so the output on the hex0 display goes through the data on the 32x4 ram
12  //built in the IP Catalog Library with the associated ram32x4.mif file for initial
13  //memory data values for each address.
14  module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
15     input logic CLOCK_50;
16     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
17     output logic [9:0] LEDR;
18     input logic [3:0] KEY;
19     input logic [9:0] SW;
20
21     logic reset;
22     logic [31:0] div_clk;
23
24     assign reset = ~KEY[0]; //third switch connected to reset
25     parameter whichClock = 25; // 0.75 Hz clock
26
27     clock_divider cdiv (.clock(CLOCK_50), .divided_clocks(div_clk));
28
29     logic clkSelect;
30
31     //assign clkSelect = div_clk[whichClock]; // for board
32     assign clkSelect = CLOCK_50; //for simulation
33
34     assign LEDR[0] = clkSelect;
35     assign LEDR[3] = CLOCK_50;
36
37     logic [3:0] out;
38     logic [4:0] read;
39
40     //instantiating display_data2, output hex display value of the given data from SW3-0 on
    HEX1
41     display_data2 wdata(.data(SW[3:0]), .hex(HEX1));
42
43     //instantiating addr2, output hex display value on HEX5 and HEX4, from given address
    from SW8-4
44     addr2 wradd(.addr(SW[8:4]), .hex5(HEX5), .hex4(HEX4));
45
46     //instantiating counter, outputing count to logic read for read address
47     counter raddr(.reset(reset), .clk(clkSelect), .count(read));
48
49     //instantiating addr2, output read address on HEX3 and HEX2
50     addr2 readdr(.addr(read), .hex5(HEX3), .hex4(HEX2));
51
52     //instantiating ram32x4, to write data and output data from given address from the
    counter output
53     ram32x4 mem(.clock(CLOCK_50), .data(SW[3:0]), .rdaddress(read), .wraddress(SW[8:4]), .
    wren(~KEY[3]), .q(out));
54
55     //instantiating display_data2, to display on HEX0 the output data from given ram32x4
56     display_data2 rdata(.data(out), .hex(HEX0));
57
58
59  endmodule
60
61  //Testing the DE1_SoC module by going through a series of inputs on KEY[0] as the
62  //reset and KEY[3] as the wren, and SW[8:0] representing the address and the data
63  //seeing if HEX5, HEX4, HEX2, HEX1, and HEX0 is correct.
64  `timescale 1 ps / 1 ps
65  module DE1_SoC_testbench();
66     logic CLOCK_50;
67     logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
68     logic [9:0] LEDR;
69     logic [3:0] KEY;
70     logic [9:0] SW;
71
72     DE1_SoC dut (.CLOCK_50, .HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW);
```

```
73
74      parameter clock_period = 100;
75
76          initial begin
77             CLOCK_50 <= 0;
78             forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
79
80          end //initial
81
82      initial begin
83          KEY[0] <= 1'b1; KEY[3] <= 1'b0; SW[8:4] <= 5'b00001; SW[3:0] <= 4'b0001; @(posedge
    CLOCK_50);
84          KEY[0] <= 1'b0; KEY[3] <= 1'b1; SW[8:4] <= 5'b00001; SW[3:0] <= 4'b0001; @(posedge
    CLOCK_50);
85          KEY[0] <= 1'b0; KEY[3] <= 1'b0; SW[8:4] <= 5'b00010; SW[3:0] <= 4'b0010; @(posedge
    CLOCK_50);
86          KEY[0] <= 1'b0; KEY[3] <= 1'b0; SW[8:4] <= 5'b00010; SW[3:0] <= 4'b0010; @(posedge
    CLOCK_50);
87          KEY[0] <= 1'b0; KEY[3] <= 1'b1; SW[8:4] <= 5'b00100; SW[3:0] <= 4'b0100; @(posedge
    CLOCK_50);
88          KEY[0] <= 1'b0; KEY[3] <= 1'b1; SW[8:4] <= 5'b00100; SW[3:0] <= 4'b0100; @(posedge
    CLOCK_50);
89          KEY[0] <= 1'b0; KEY[3] <= 1'b1; SW[8:4] <= 5'b01000; SW[3:0] <= 4'b1000; @(posedge
    CLOCK_50);
90          KEY[0] <= 1'b0; KEY[3] <= 1'b1; SW[8:4] <= 5'b01000; SW[3:0] <= 4'b1000; @(posedge
    CLOCK_50);
91          KEY[0] <= 1'b0; @(posedge CLOCK_50);
92          KEY[0] <= 1'b0; @(posedge CLOCK_50);
93          KEY[0] <= 1'b0; @(posedge CLOCK_50);
94          KEY[0] <= 1'b0; @(posedge CLOCK_50);
95          KEY[0] <= 1'b0; @(posedge CLOCK_50);
96          KEY[0] <= 1'b0; @(posedge CLOCK_50);
97          KEY[0] <= 1'b0; @(posedge CLOCK_50);
98          KEY[0] <= 1'b0; @(posedge CLOCK_50);
99          KEY[0] <= 1'b0; @(posedge CLOCK_50);
100         KEY[0] <= 1'b0; @(posedge CLOCK_50);
101         KEY[0] <= 1'b0; @(posedge CLOCK_50);
102         KEY[0] <= 1'b0; @(posedge CLOCK_50);
103         KEY[0] <= 1'b0; @(posedge CLOCK_50);
104         KEY[0] <= 1'b0; @(posedge CLOCK_50);
105         KEY[0] <= 1'b0; @(posedge CLOCK_50);
106         KEY[0] <= 1'b0; @(posedge CLOCK_50);
107
108         $stop;
109     end
110
111 endmodule
112
113 //clock_divider module has inputs of the clock, reset, and the
114 //32 bit divided_clock which allows to sequence through and
115 //output whenever the postive edge has been reached on the clock
116 // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
117 module clock_divider (clock, divided_clocks);
118     input logic clock;
119     output logic [31:0] divided_clocks = 0;
120
121     always_ff @(posedge clock) begin
122         divided_clocks <= divided_clocks + 1;
123     end
124
125 endmodule
126
```

8) addr2.sv (task 2)

```
1   //Khoa Tran
2   //04/22/2021
3   //Lab 2, Task 2
4   //This module controls the logic case for displaying the hex value of the address
5   //on the hex 7-seg display board. With inputs of address as 5 bits, combinational
6   //logic through the cases to have the hex output that matches the input data value
7   module addr2#(parameter addr_width = 5) (addr, hex5, hex4);
8      input logic [addr_width - 1:0] addr;
9      output logic [6:0] hex5, hex4;
10
11     //logic for 7-seg display for hex values 1-15
12     logic [6:0] zero, one, two, three, four, five, six, seven, eight, nine, A, B, C, D, E, F;
13     assign zero = 7'b1000000;
14     assign one = 7'b1111001;
15     assign two = 7'b0100100;
16     assign three = 7'b0110000;
17     assign four = 7'b0011001;
18     assign five = 7'b0010010;
19     assign six = 7'b0000010;
20     assign seven = 7'b1111000;
21     assign eight = 7'b0000000;
22     assign nine = 7'b0010000;
23     assign A = 7'b0001000;
24     assign B = 7'b0000011;
25     assign C = 7'b1000110;
26     assign D = 7'b0100001;
27     assign E = 7'b0000110;
28     assign F = 7'b0001110;
29
30     //combinational for cases of addr values 1-31 to display in hex5-4
31     always_comb begin
32              case (addr)
33                 5'b00000: begin
34                           hex5 = zero;
35                           hex4 = zero;
36                           end
37                 5'b00001: begin
38                           hex5 = zero;
39                           hex4 = one;
40                           end
41                 5'b00010: begin
42                           hex5 = zero;
43                           hex4 = two;
44                           end
45                 5'b00011: begin
46                           hex5 = zero;
47                           hex4 = three;
48                           end
49                 5'b00100: begin
50                           hex5 = zero;
51                           hex4 = four;
52                           end
53                 5'b00101: begin
54                           hex5 = zero;
55                           hex4 = five;
56                           end
57                 5'b00110: begin
58                           hex5 = zero;
59                           hex4 = six;
60                           end
61                 5'b00111: begin
62                           hex5 = zero;
63                           hex4 = seven;
64                           end
65                 5'b01000: begin
66                           hex5 = zero;
67                           hex4 = eight;
68                           end
69                 5'b01001: begin
70                           hex5 = zero;
71                           hex4 = nine;
72                           end
73                 5'b01010: begin
74                           hex5 = zero;
75                           hex4 = A;
76                           end
```

```
                5'b01011: begin
                          hex5 = zero;
                          hex4 = B;
                          end
                5'b01100: begin
                          hex5 = zero;
                          hex4 = C;
                          end
                5'b01101: begin
                          hex5 = zero;
                          hex4 = D;
                          end
                5'b01110: begin
                          hex5 = zero;
                          hex4 = E;
                          end
                5'b01111: begin
                          hex5 = zero;
                          hex4 = F;
                          end
                5'b10000: begin
                          hex5 = one;
                          hex4 = zero;
                          end
                5'b10001: begin
                          hex5 = one;
                          hex4 = one;
                          end
                5'b10010: begin
                          hex5 = one;
                          hex4 = two;
                          end
                5'b10011: begin
                          hex5 = one;
                          hex4 = three;
                          end
                5'b10100: begin
                          hex5 = one;
                          hex4 = four;
                          end
                5'b10101: begin
                          hex5 = one;
                          hex4 = five;
                          end
                5'b10110: begin
                          hex5 = one;
                          hex4 = six;
                          end
                5'b10111: begin
                          hex5 = one;
                          hex4 = seven;
                          end
                5'b11000: begin
                          hex5 = one;
                          hex4 = eight;
                          end
                5'b11001: begin
                          hex5 = one;
                          hex4 = nine;
                          end
                5'b11010: begin
                          hex5 = one;
                          hex4 = A;
                          end
                5'b11011: begin
                          hex5 = one;
                          hex4 = B;
                          end
                5'b11100: begin
                          hex5 = one;
                          hex4 = C;
                          end
                5'b11101: begin
                          hex5 = one;
                          hex4 = D;
                          end
```

```
                5'b11110: begin
                          hex5 = one;
                          hex4 = E;
                          end
                5'b11111: begin
                          hex5 = one;
                          hex4 = F;
                          end
                default: begin
                          hex5 = 7'b1111111;
                          hex4 = 7'b1111111;
                          end
                endcase
        end
endmodule

//This module is a testbench for displaying the address as it goes through
//the given input address points and test for if the hex output matches the
//given address value in hex for the 7-seg display board
module addr2_testbench#(parameter addr_width = 5)();
   logic [addr_width - 1:0] addr;
   logic [6:0] hex5, hex4;

   //devices under test
   addr2 dut(.addr, .hex5, .hex4);

   //initial simulation
   initial begin
      addr=5'b00000; #10;
      addr=5'b00001; #10;
      addr=5'b00010; #10;
      addr=5'b00100; #10;
      addr=5'b01000; #10;
      addr=5'b10000; #10;
      addr=5'b10001; #10;
      addr=5'b10010; #10;
      addr=5'b10011; #10;
      addr=5'b10100; #10;
      addr=5'b10101; #10;
      addr=5'b10101; #10;
      addr=5'b11111; #10;
      $stop; //end simulation
   end
endmodule


```

9) display_data2.sv (task 2)

```
1   //Khoa Tran
2   //04/22/2021
3   //Lab 2, Task 2
4   //This module controls the logic case for displaying the hex value of the data
5   //on the hex 7-seg display board. With inputs of data as 4 bits, combinational
6   //logic through the cases to have the hex output that matches the input data value
7   module display_data2#(parameter data_width = 4) (data, hex);
8      input logic [data_width - 1:0] data;
9      output logic [6:0] hex;
10
11     //logic for 7-seg display for hex values 1-15
12     logic [6:0] zero, one, two, three, four, five, six, seven, eight, nine, A, B, C, D, E, F;
13     assign zero = 7'b1000000;
14     assign one = 7'b1111001;
15     assign two = 7'b0100100;
16     assign three = 7'b0110000;
17     assign four = 7'b0011001;
18     assign five = 7'b0010010;
19     assign six = 7'b0000010;
20     assign seven = 7'b1111000;
21     assign eight = 7'b0000000;
22     assign nine = 7'b0010000;
23     assign A = 7'b0001000;
24     assign B = 7'b0000011;
25     assign C = 7'b1000110;
26     assign D = 7'b0100001;
27     assign E = 7'b0000110;
28     assign F = 7'b0001110;
29
30     //combinational for cases of data values 1-31 to display in hex5-4
31     always_comb begin
32              case (data)
33                 4'b0000: begin
34                          hex = zero;
35                          end
36                 4'b0001: begin
37                          hex = one;
38                          end
39                 4'b0010: begin
40                          hex = two;
41                          end
42                 4'b0011: begin
43                          hex = three;
44                          end
45                 4'b0100: begin
46                          hex = four;
47                          end
48                 4'b0101: begin
49                          hex = five;
50                          end
51                 4'b0110: begin
52                          hex = six;
53                          end
54                 4'b0111: begin
55                          hex = seven;
56                          end
57                 4'b1000: begin
58                          hex = eight;
59                          end
60                 4'b1001: begin
61                          hex = nine;
62                          end
63                 4'b1010: begin
64                          hex = A;
65                          end
66                 4'b1011: begin
67                          hex = B;
68                          end
69                 4'b1100: begin
70                          hex = C;
71                          end
72                 4'b1101: begin
73                          hex = D;
74                          end
75                 4'b1110: begin
76                          hex = E;
```



```
77                          end
78                 4'b1111: begin
79                          hex = F;
80                          end
81                 default: begin
82                          hex = 7'b1111111;
83                          end
84                 endcase
85         end
86  endmodule
87
88  //This module is a testbench for displaying the data as it goes through
89  //the given input data points and test for if the hex output matches the
90  //given input value in hex for the 7-seg display board
91  module display_data2_testbench#(parameter data_width = 4)();
92
93     logic [data_width - 1:0] data;
94     logic [6:0] hex;
95
96     //devices under test
97     display_data2 dut(.data, .hex);
98
99     //initial simulation
100    initial begin
101       data=4'b0000;  #10;
102       data=4'b0001;  #10;
103       data=4'b0010;  #10;
104       data=4'b0101;  #10;
105       data=4'b1000;  #10;
106       data=4'b0100;  #10;
107       data=4'b0011;  #10;
108       data=4'b0010;  #10;
109       data=4'b0011;  #10;
110       data=4'b0100;  #10;
111       data=4'b0101;  #10;
112       data=4'b0101;  #10;
113       data=4'b1111;  #10;
114
115       $stop; //end simulation
116    end
117 endmodule
118
```

10) FIFO_Control.sv (task 3)

```
1   //Khoa Tran
2   //04/22/2021
3   //Lab 2, Task 3
4   //This module implements a FIFO controller that takes inpuit clk, reset, read, and write,
    outputting
5   //either if the state of the ram is full or empty based on outputs empty and full and also
    outputing
6   //the write address and the read address that maintains the state of first in first out
    data, as data is read by
7   //the order that is entered in. As well as ensuring that write address goes in a queue.
8   module FIFO_Control#(parameter addr_width = 4)(clk, reset, read, write, wr_en, empty, full,
    readAddr, writeAddr);
9      input logic clk, reset;
10     input logic read, write;
11     output logic wr_en;
12     output logic empty, full;
13     output logic [addr_width-1:0] readAddr, writeAddr;
14
15     //logic variables for ptr and nextptr for both read and write, as well
16     //as the result for empty and full
17     logic [addr_width-1:0] writePtr, nextWritePtr;
18     logic [addr_width-1:0] readPtr, nextReadPtr;
19     logic emptyResult, fullResult;
20
21     //assign output of readAddr and writeAddr to the readPtr and writePtr
22     assign readAddr = readPtr;
23     assign writeAddr = writePtr;
24
25     //assign output of wr_en to write and read or write and not full
26     assign wr_en = write & (read | ~full);
27
28     //sequential logic with reset input setting writePtr, readPtr, full, and empty
29     //as well well as setting writePtr, readPtr, full, and empty to the next and result of
30     //their perspective value on a posedge clk
31     always_ff @(posedge clk)
32        begin
33           if (reset)
34              begin
35                 writePtr <= 0;
36                 readPtr <= 0;
37                 full <= 0;
38                 empty <= 1;
39              end
40           else
41              begin
42                 writePtr <= nextWritePtr;
43                 readPtr <= nextReadPtr;
44                 full <= fullResult;
45                 empty <= emptyResult;
46              end
47        end
48
49     //combinational logic
50     always_comb
51        begin
52           //setting the next and result variables to the current, default values
53           fullResult = full;
54           emptyResult = empty;
55           nextWritePtr = writePtr;
56           nextReadPtr = readPtr;
57           case ({write, read})
58              //case of write and not read, if not full then make emptyResult false,
59              //increment nextWritePtr and set fullResult to true on the condition below
60              2'b10:
61                 if (~full)
62                    begin
63                       emptyResult = 0;
64                       nextWritePtr = writePtr + 1;
65                       if (nextWritePtr == readPtr)
66                          fullResult = 1;
67                    end
68              //case of read and not write, if not empty then increment nextReadPtr
69              //and set full to false as well as well as empty to true on condition below
70              2'b01:
71                 if (~empty)
72                    begin
```

```
73                          nextReadPtr = readPtr + 1;
74                          fullResult = 0;
75                          if (nextReadPtr == writePtr)
76                             emptyResult = 1;
77                       end
78                //case of read and write, if not full, then increment write pointer, asserting
79                //full on condition below, as well as if not empty then increment nextReadPtr
80                2'b11:
81                   begin
82                      if (~full)
83                         begin
84                            nextWritePtr = writePtr + 1;
85                            if (nextWritePtr == readPtr)
86                               fullResult = 1;
87                         end
88                      if (~empty)
89                         begin
90                            nextReadPtr = readPtr + 1;
91                         end
92                   end
93                //case of no read and no write, set empty on given condition, otherwise, do
    nothing
94                2'b00:
95                   if (~full & (writePtr == readPtr))
96                      begin
97                         emptyResult = 1;
98                      end
99             endcase
100         end
101   endmodule
102
103   //Module is testbench for FIFO_Control by a series of values on reset, clk, read, and write
    to
104   //test if the outputs of wr_en, empty, full, readAddr, and writeAddr is correct alont each
105   //posedge clk
106   module FIFO_Control_testbench #(parameter addr_width = 4)();
107      logic clk, reset;
108      logic read, write;
109      logic wr_en;
110      logic empty, full;
111      logic [addr_width-1:0] readAddr, writeAddr;
112
113      //device under test
114      FIFO_Control dut(.clk, .reset, .read, .write, .wr_en, .empty, .full, .readAddr, .
    writeAddr);
115
116      //clock setup
117      parameter clock_period = 100;
118
119      initial begin
120         clk <= 0;
121         forever #(clock_period /2) clk <= ~clk;
122      end
123      //initial simulation
124      initial begin
125         reset <= 1;          @(posedge clk);
126         reset <= 0; read <= 0;  write <= 0;    @(posedge clk);
127         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
128         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
129         reset <= 0; read <= 1;  write <= 0;    @(posedge clk);
130         reset <= 0; read <= 1;  write <= 0;    @(posedge clk);
131         reset <= 0; read <= 0;  write <= 0;    @(posedge clk);
132         reset <= 0; read <= 1;  write <= 1;    @(posedge clk);
133         reset <= 0; read <= 1;  write <= 1;    @(posedge clk);
134         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
135         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
136         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
137         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
138         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
139         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
140         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
141         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
142         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
143         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
144         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
145         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
```



```
146         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
147         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
148         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
149         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
150         reset <= 0; read <= 0;  write <= 1;    @(posedge clk);
151         reset <= 0; read <= 1;  write <= 0;    @(posedge clk);
152         reset <= 0; read <= 1;  write <= 0;    @(posedge clk);
153
154         $stop; //end simulation
155      end
156   endmodule
157
```

11) DE1_SoC.sv (task 3)

```
1   //Khoa Tran
2   //04/22/2021
3   //Lab 2, Task 3
4   //This module implements a FIFO queue for the ram size 16x8, as inputs of
5   //SW7-0 provides the data for the ram, SW9 for if the user wants to read, SW8
6   //for if the user wants to write. The concept is that as the user reads and writes
7   //data, the FifO_Control will decide which address to read from and write to based
8   //on the first in first out condition of a queue as it reads the least recent data.
9   //This module also implements an empty and full on LEDR8 and 9 to indicate if the
10  //ram is full or empty, so that the user doesn't read on empty and write on full.
11  //Using CLOCK_50 to simulate and 0.75hz clock on the board, read data is on HEX1 and
12  //HEX0 as write data is on HEX5 and 4. Also has ablity to reset the fifo to default
    conditions.
13  module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
14     input logic CLOCK_50;
15     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
16     output logic [9:0] LEDR;
17     input logic [3:0] KEY;
18     input logic [9:0] SW;
19
20     logic reset;
21     logic [31:0] div_clk;
22
23     assign reset = ~KEY[0]; //third switch connected to reset
24     parameter whichClock = 25; // 0.75 Hz clock
25
26
27     clock_divider cdiv (.clock(CLOCK_50), .divided_clocks(div_clk));
28
29     logic clkSelect;
30
31     assign clkSelect = CLOCK_50; // for simulation
32
33     //assign clkSelect = div_clk[whichClock]; // for board
34
35     //LEDR[0] connected to clk
36     assign LEDR[0] = clkSelect;
37
38     logic [7:0] out;
39     logic wrenable;
40     logic emptyout;
41     logic fullout;
42     logic [3:0] readout;
43     logic [3:0] writeout;
44
45     //instantiating of data2 twice of each 4 bits of the data from SW7-0, showing on HEX5
    and HEX4
46     data2 displaydata1(.data(SW[7:4]), .hex(HEX5));
47     data2 displaydata2(.data(SW[3:0]), .hex(HEX4));
48
49     //instantiating FIFO_Control passing in SW9 for read and SW8 for write, and outputing
    empty on LEDR8 and full on LEDR9, as well as read and write address for the ram
50     FIFO_Control fifo(.clk(clkSelect), .reset(reset), .read(SW[9]), .write(SW[8]), .wr_en(
    wrenable), .empty(LEDR[8]), .full(LEDR[9]), .readAddr(readout), .writeAddr(writeout));
51     //instantiating ram, as data is from the switch and read and write address and wren is
    from the FIFO_Control, outputing q on out
52     ram16x8 mem(.clock(clkSelect), .data(SW[7:0]), .rdaddress(readout), .wraddress(writeout),
     .wren(wrenable), .q(out));
53
54     //instantiating of outControl twice of each 4 bits of the output data from ram, showing
    on HEX1 and HEX0
55     outControl rdata(.data(out[7:4]), .hex(HEX1), .switch(SW[9]));
56     outControl r2data(.data(out[3:0]), .hex(HEX0), .switch(SW[9]));
57
58     //show nothing on hex3 and 2
59     assign HEX3 = 7'b1111111;
60     assign HEX2 = 7'b1111111;
61
62
63  endmodule
64
65  //Testing the DE1_SoC module by implementing a series of input values on KEY0, SW9-0
66  //in order to test if the output on HEX5, HEX4, HEX1, and HEX0 is correct on posedge
67  //CLOCK_50
68  `timescale 1 ps / 1 ps
69  module DE1_SoC_testbench();
```

```
70      logic CLOCK_50;
71      logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
72      logic [9:0] LEDR;
73      logic [3:0] KEY;
74      logic [9:0] SW;
75
76      //device under test
77      DE1_SoC dut (.CLOCK_50, .HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW);
78
79      parameter clock_period = 100;
80
81         initial begin
82            CLOCK_50 <= 0;
83            forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
84
85         end //initial
86
87      //initial simulation
88      initial begin
89            KEY[0] <= 1'b0; SW[9] <= 1'b0; SW[8] <= 1'b0; SW[7:0] <= 8'b00000000; @(posedge
    CLOCK_50);
90            KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
91            KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000010; @(posedge
    CLOCK_50);
92            KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000011; @(posedge
    CLOCK_50);
93            KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[8] <= 1'b0; SW[7:0] <= 8'b00000000; @(posedge
    CLOCK_50);
94            KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[8] <= 1'b0; SW[7:0] <= 8'b00000000; @(posedge
    CLOCK_50);
95            KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[8] <= 1'b1; SW[7:0] <= 8'b00000111; @(posedge
    CLOCK_50);
96            KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[8] <= 1'b0; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
97            KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
98            KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
99            KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
100           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
101           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
102           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
103           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
104           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
105           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
106           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
107           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
108           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
109           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
110           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
111           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
112           KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[8] <= 1'b1; SW[7:0] <= 8'b00000001; @(posedge
    CLOCK_50);
113           $stop;
114        end
115
116     endmodule
117
118     //clock_divider module has inputs of the clock, reset, and the
119     //32 bit divided_clock which allows to sequence through and
120     //output whenever the postive edge has been reached on the clock
121     // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
```



```
122     module clock_divider (clock, divided_clocks);
123        input logic clock;
124        output logic [31:0] divided_clocks = 0;
125
126        always_ff @(posedge clock) begin
127           divided_clocks <= divided_clocks + 1;
128        end
129
130     endmodule
131
```

12) data2.sv (task 3)

```
1   //Khoa Tran
2   //04/22/2021
3   //Lab 2, Task 3
4   //This module controls the logic case for displaying the hex value of the data
5   //on the hex 7-seg display board. With inputs of data as 4 bits, combinational
6   //logic through the cases to have the hex output that matches the input data value
7   module data2#(parameter data_width = 4) (data, hex);
8      input logic [data_width - 1:0] data;
9      output logic [6:0] hex;
10
11     //logic for 7-seg display for hex values 1-15
12     logic [6:0] zero, one, two, three, four, five, six, seven, eight, nine, A, B, C, D, E, F;
13     assign zero = 7'b1000000;
14     assign one = 7'b1111001;
15     assign two = 7'b0100100;
16     assign three = 7'b0110000;
17     assign four = 7'b0011001;
18     assign five = 7'b0010010;
19     assign six = 7'b0000010;
20     assign seven = 7'b1111000;
21     assign eight = 7'b0000000;
22     assign nine = 7'b0010000;
23     assign A = 7'b0001000;
24     assign B = 7'b0000011;
25     assign C = 7'b1000110;
26     assign D = 7'b0100001;
27     assign E = 7'b0000110;
28     assign F = 7'b0001110;
29
30     //combinational for cases of data values 1-5 to display in hex
31     always_comb begin
32              case (data)
33                 4'b0000: begin
34                          hex = zero;
35                          end
36                 4'b0001: begin
37                          hex = one;
38                          end
39                 4'b0010: begin
40                          hex = two;
41                          end
42                 4'b0011: begin
43                          hex = three;
44                          end
45                 4'b0100: begin
46                          hex = four;
47                          end
48                 4'b0101: begin
49                          hex = five;
50                          end
51                 4'b0110: begin
52                          hex = six;
53                          end
54                 4'b0111: begin
55                          hex = seven;
56                          end
57                 4'b1000: begin
58                          hex = eight;
59                          end
60                 4'b1001: begin
61                          hex = nine;
62                          end
63                 4'b1010: begin
64                          hex = A;
65                          end
66                 4'b1011: begin
67                          hex = B;
68                          end
69                 4'b1100: begin
70                          hex = C;
71                          end
72                 4'b1101: begin
73                          hex = D;
74                          end
75                 4'b1110: begin
76                          hex = E;
```

```
                        end
                4'b1111: begin
                        hex = F;
                        end
                default: begin
                        hex = 7'b1111111;
                        end
                endcase
        end
endmodule

//This module is a testbench for displaying the data as it goes through
//the given input data points and test for if the hex output matches the
//given input value in hex for the 7-seg display board
module data2_testbench#(parameter data_width = 4)();

    logic [data_width - 1:0] data;
    logic [6:0] hex;

    //devices under test
    data2 dut(.data, .hex);

    //initial simulation
    initial begin
        data=4'b0000;  #10;
        data=4'b0001;  #10;
        data=4'b0010;  #10;
        data=4'b0101;  #10;
        data=4'b1000;  #10;
        data=4'b0100;  #10;
        data=4'b0011;  #10;
        data=4'b0010;  #10;
        data=4'b0011;  #10;
        data=4'b0100;  #10;
        data=4'b0101;  #10;
        data=4'b0101;  #10;
        data=4'b1111;  #10;

        $stop; //end simulation
    end
endmodule

```

13) outControl.sv (task 3)

```
1   //Khoa Tran
2   //04/22/2021
3   //Lab 2, Task 3
4   //This module controls the logic case for displaying the hex value of the output data
5   //on the hex 7-seg display board. With inputs of data as 4 bits and the switch,
6   //combinational logic through the cases to have the hex output that matches the input data
    value
7   //and the switch, only outputing an actual hex display value if switch is true. Else, the
    output
8   //display will be value that shows nothing.
9   module outControl#(parameter data_width = 4)(data, hex, switch);
10     input logic [data_width - 1:0] data;
11     input logic switch;
12     output logic [6:0] hex;
13
14     //logic for nothing to zero to 15 with hex display value assigned
15     logic [6:0] nothing, zero, one, two, three, four, five, six, seven, eight, nine, A, B, C,
     D, E, F;
16     assign nothing = 7'b1111111;
17     assign zero = 7'b1000000;
18     assign one = 7'b1111001;
19     assign two = 7'b0100100;
20     assign three = 7'b0110000;
21     assign four = 7'b0011001;
22     assign five = 7'b0010010;
23     assign six = 7'b0000010;
24     assign seven = 7'b1111000;
25     assign eight = 7'b0000000;
26     assign nine = 7'b0010000;
27     assign A = 7'b0001000;
28     assign B = 7'b0000011;
29     assign C = 7'b1000110;
30     assign D = 7'b0100001;
31     assign E = 7'b0000110;
32     assign F = 7'b0001110;
33
34     //combinational for cases of data and switch concatenated together
35     always_comb begin
36               case ({data, switch})
37                  5'b00000: begin
38                            hex = nothing;
39                            end
40                  5'b00001: begin
41                            hex = zero;
42                            end
43                  5'b00011: begin
44                            hex = one;
45                            end
46                  5'b00101: begin
47                            hex = two;
48                            end
49                  5'b00111: begin
50                            hex = three;
51                            end
52                  5'b01001: begin
53                            hex = four;
54                            end
55                  5'b01011: begin
56                            hex = five;
57                            end
58                  5'b01101: begin
59                            hex = six;
60                            end
61                  5'b01111: begin
62                            hex = seven;
63                            end
64                  5'b10001: begin
65                            hex = eight;
66                            end
67                  5'b10011: begin
68                            hex = nine;
69                            end
70                  5'b10101: begin
71                            hex = A;
72                            end
73                  5'b10111: begin
```

```
                        hex = B;
                        end
                5'b11001: begin
                        hex = C;
                        end
                5'b11011: begin
                        hex = D;
                        end
                5'b11101: begin
                        hex = E;
                        end
                5'b11111: begin
                        hex = F;
                        end
                default: begin
                        hex = 7'b1111111;
                        end
                endcase
        end
endmodule

//This module is a testbench for outputing data by passing in
//a series of data inputs and switch inputs and testing if
//the hex output is correct
module outControl_testbench#(parameter data_width = 4)();

   logic [data_width - 1:0] data;
   logic [6:0] hex;
   logic switch;

   //devices under test
   outControl dut(.data, .hex, .switch);

   //initial simulation
   initial begin
      data<=4'b0000; switch<=0; #10;
      data<=4'b0000; switch<=1; #10;
      data<=4'b0001; switch<=1; #10;
      data<=4'b0010; switch<=1; #10;
      data<=4'b0101; switch<=1; #10;
      data<=4'b1000; switch<=1; #10;
      data<=4'b0100; switch<=1; #10;
      data<=4'b0011; switch<=1; #10;
      data<=4'b0010; switch<=1; #10;
      data<=4'b0011; switch<=1; #10;
      data<=4'b0100; switch<=1; #10;
      data<=4'b0101; switch<=1; #10;
      data<=4'b0101; switch<=1; #10;
      data<=4'b1111; switch<=1; #10;
      $stop; //end simulation
   end
endmodule

```

14) ram16x8_testbench.sv (task 3)

```
220   //Khoa Tran
221   //04/22/2021
222   //Lab 2, Task 3
223   //Module testing the ram16x8 by entering a series of inputs on data, rdaddress, wraddress,
      and wren
224   //as well to see if the output on q is correct on each posedge clock.
225   `timescale 1 ps / 1 ps
226   module ram16x8_testbench();
227      reg   clock;
```



```
228      reg   [7:0]  data;
229      reg   [3:0]  rdaddress;
230      reg   [3:0]  wraddress;
231      reg     wren;
232      wire  [7:0]  q;
233
234      ram16x8 dut(.clock(clock), .data(data), .rdaddress(rdaddress), .wraddress(wraddress), .
      wren(wren), .q(q));
235
236      //clock setup
237      parameter clock_period = 100;
238
239      initial begin
240         clock <= 0;
241          forever #(clock_period /2) clock <= ~clock;
242      end
243      //initial simulation
244      initial begin
245         data <= 8'b00000111; rdaddress <= 4'b0000; wraddress <= 4'b0000; wren<=1;  @(posedge
      clock);
246         data <= 8'b00000001; rdaddress <= 4'b0000; wraddress <= 4'b0001; wren<=1;  @(posedge
      clock);
247         data <= 8'b00000010; rdaddress <= 4'b0001; wraddress <= 4'b0010; wren<=1;  @(posedge
      clock);
248         data <= 8'b00000011; rdaddress <= 4'b0010; wraddress <= 4'b0011; wren<=1;  @(posedge
      clock);
249         data <= 8'b00000100; rdaddress <= 4'b0011; wraddress <= 4'b0100; wren<=1;  @(posedge
      clock);
250         data <= 8'b00000101; rdaddress <= 4'b0100; wraddress <= 4'b0101; wren<=1;  @(posedge
      clock);
251         data <= 8'b00000111; rdaddress <= 4'b0101; wraddress <= 4'b0111; wren<=1;  @(posedge
      clock);
252         data <= 8'b00000000; rdaddress <= 4'b0110; wraddress <= 4'b1000; wren<=1;  @(posedge
      clock);
253         data <= 8'b00000000; rdaddress <= 4'b0000; wraddress <= 4'b1001; wren<=1;  @(posedge
      clock);
254         data <= 8'b00011111; rdaddress <= 4'b0000; wraddress <= 4'b0000; wren<=1;  @(posedge
      clock);
255         data <= 8'b00000000; rdaddress <= 4'b0000; wraddress <= 4'b1010; wren<=1;  @(posedge
      clock);
256
257         $stop; //end simulation
258      end
259   endmodule
260
```