Khoa Tran and Ravi Sangani
EE 371
May 15, 2021
Lab 4 Report

Procedure

Task #1

In order to approach this task, we first drew up on the block diagram in order to understand the connection between the controller and datapath modules for implementing the bitcounter algorithm. After observing the pseudocode, we then designed the ASMD chart in order to understand the RTL operations needed at certain states and the different transitions between the states in order to correctly increment the count based on if a 1 bit exist on the last index of the given data as the data shifts to the right on the correct state. Besides drawing the ASMD chart, we constructed the finite state machine diagram in order to understand the transitions between the certain states as well as the datapath to see the counter and shifter and their inputs of loading the data and enabling the counter or shifter to start. As seen in figure 1, the block diagram depicts the connection between the modules, while in figure 2, the ASMD chart displays the RTL operations at each state and the input and output transitions of certain logic variables.

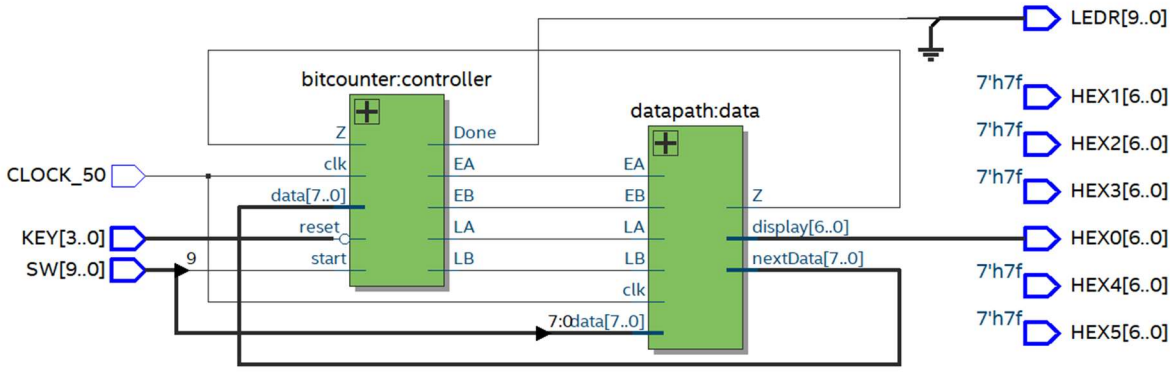

Figure 1: Block diagram for task 1

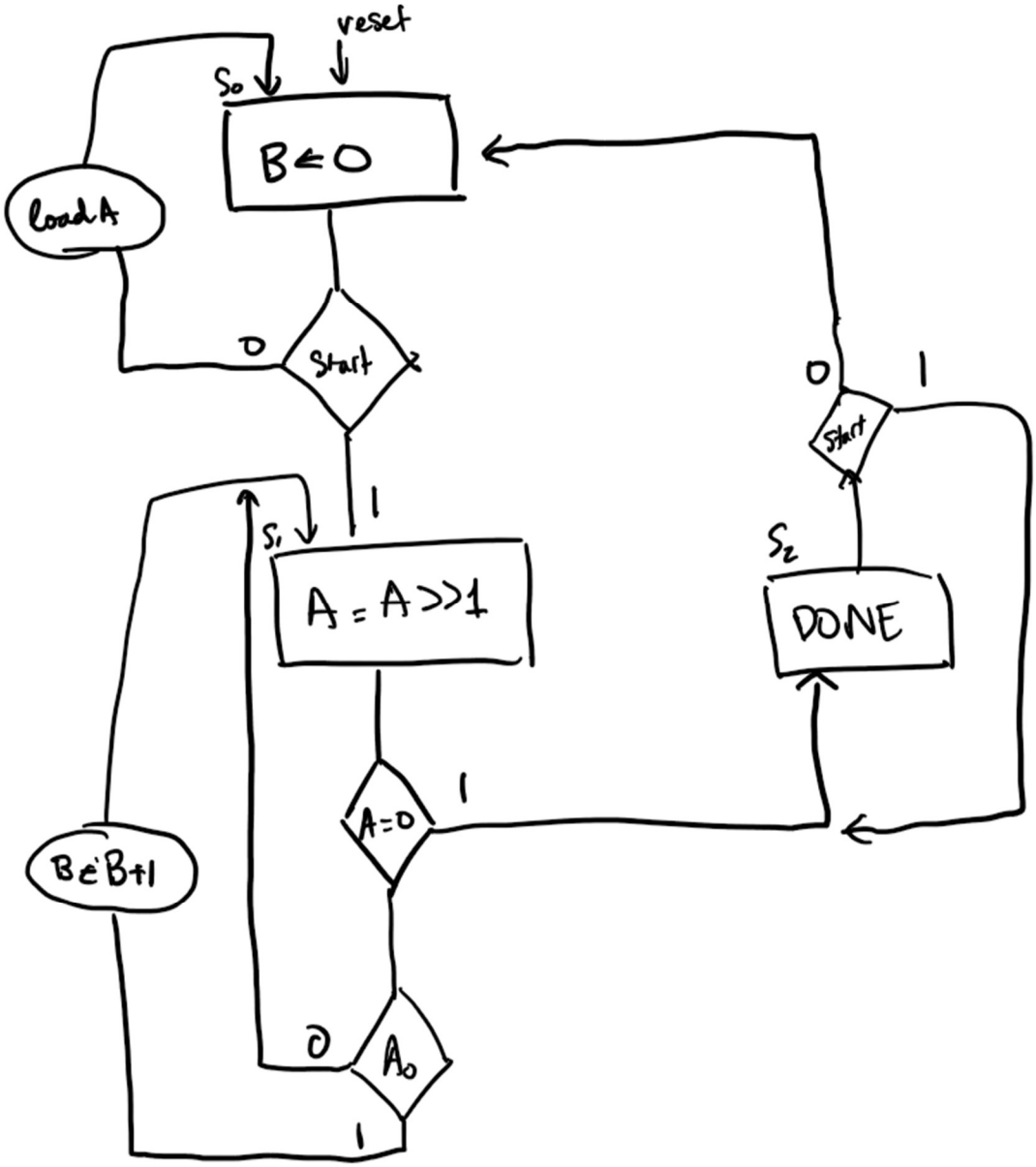

Figure 2: ASMD chart for task 1

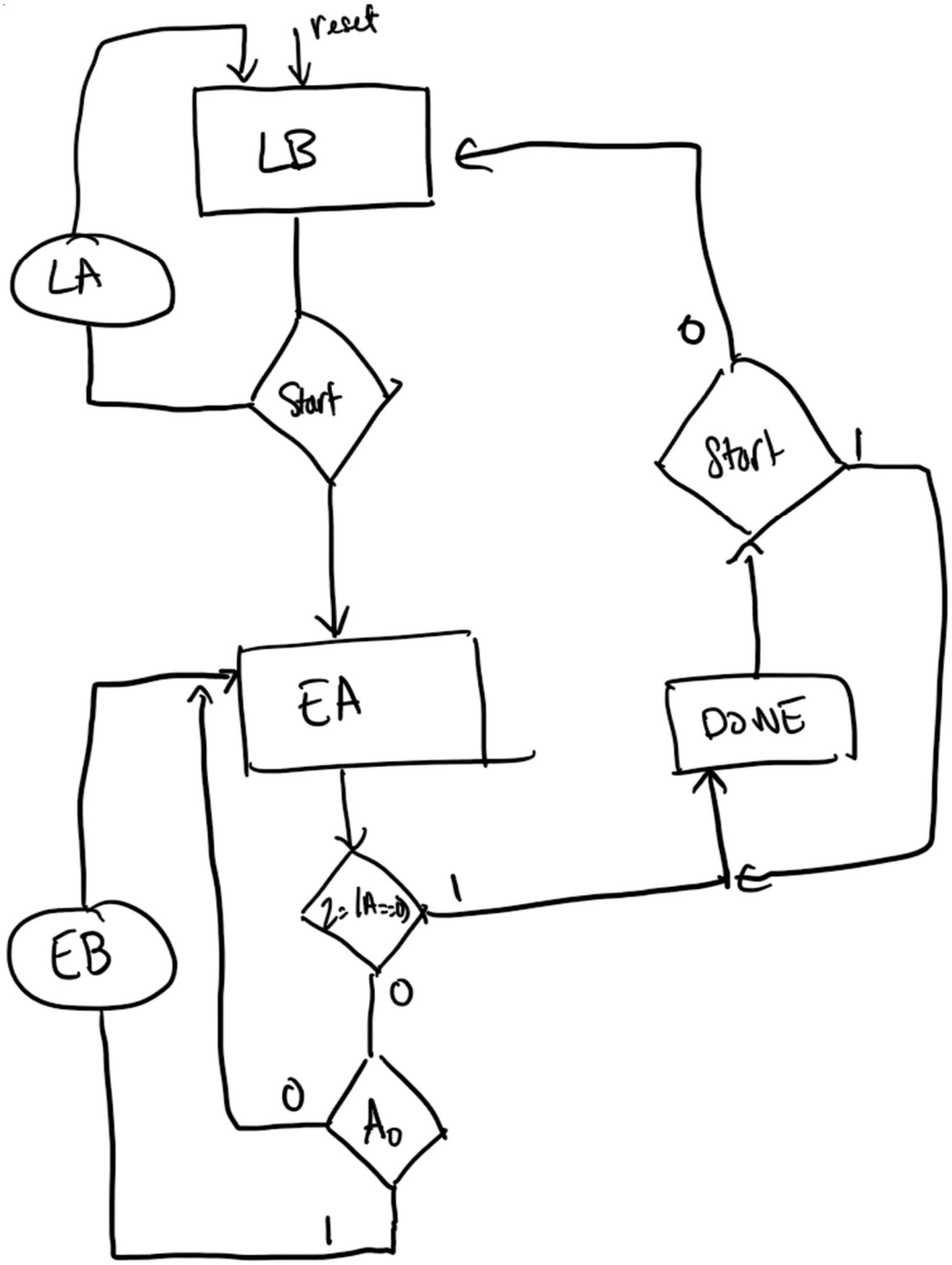

Figure 3: ASM for controller for task 1

Task #2

Approaching this problem, we first looked at the pseudocode for a binary search algorithm on a sorted array. In this problem, our sorted array is a 32x8 ram that has a sorted initial memory initialization. From the pseudocode, we then constructed the ASMD chart as well as the ASM for controller in order to understand the necessary outputs at certain states for the datapath to manipulate and change the value of lower and upper bounds, minimizing the search to a smaller

range of addresses based on given target data and current data at the middle address. The ASMD chart shows the RTL operations at every state, including both datapath and controller operations, while the ASM for controller shows the state transition of the controller as well as the output of enabling lower and upper bounds for the datapath to receive and update the variable accordingly. Overall, the block diagram as well as the ASMD and ASM for controller allowed us to program the binary search algorithm on hardware with minimal issues.

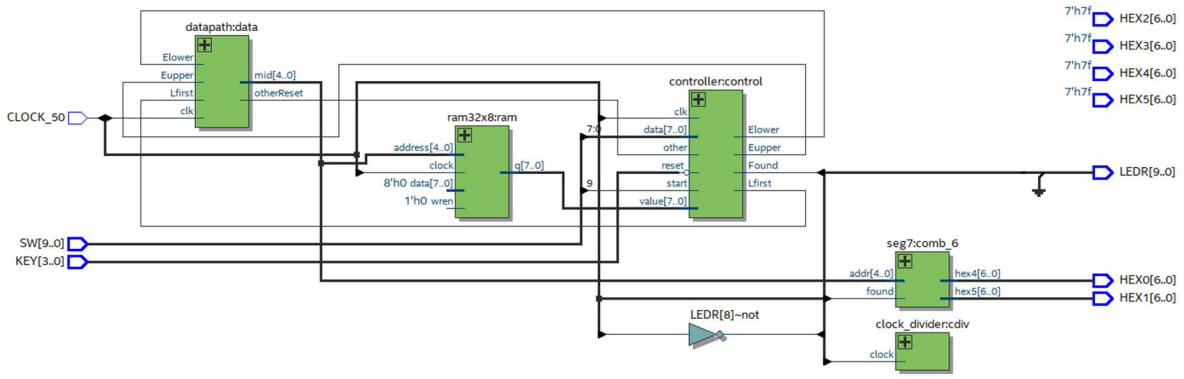

Figure 4: Block diagram for task 2

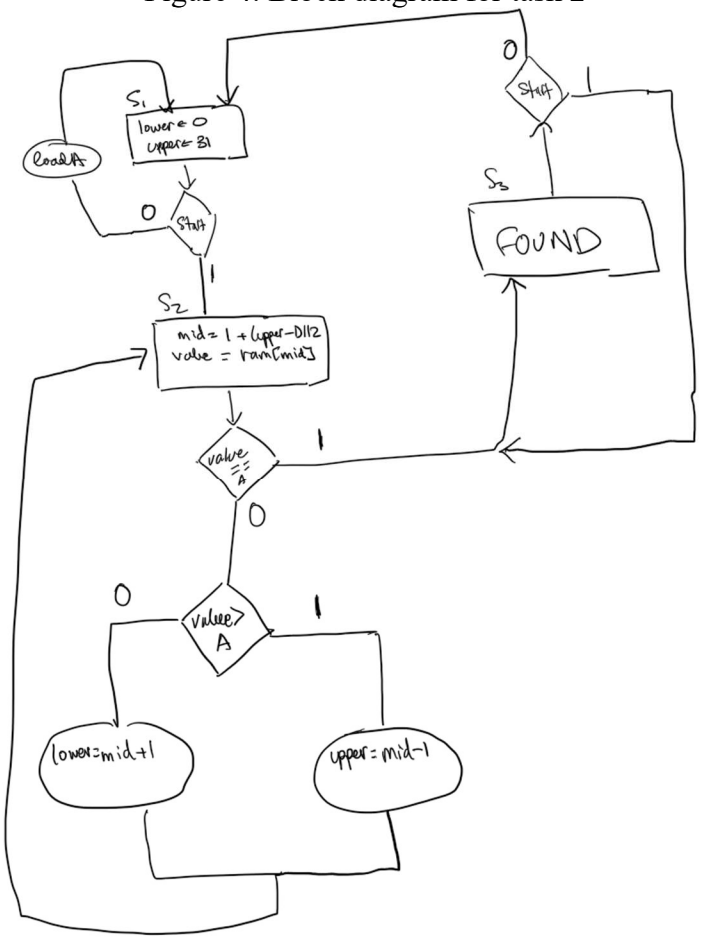

Figure 5: ASMD for task 2

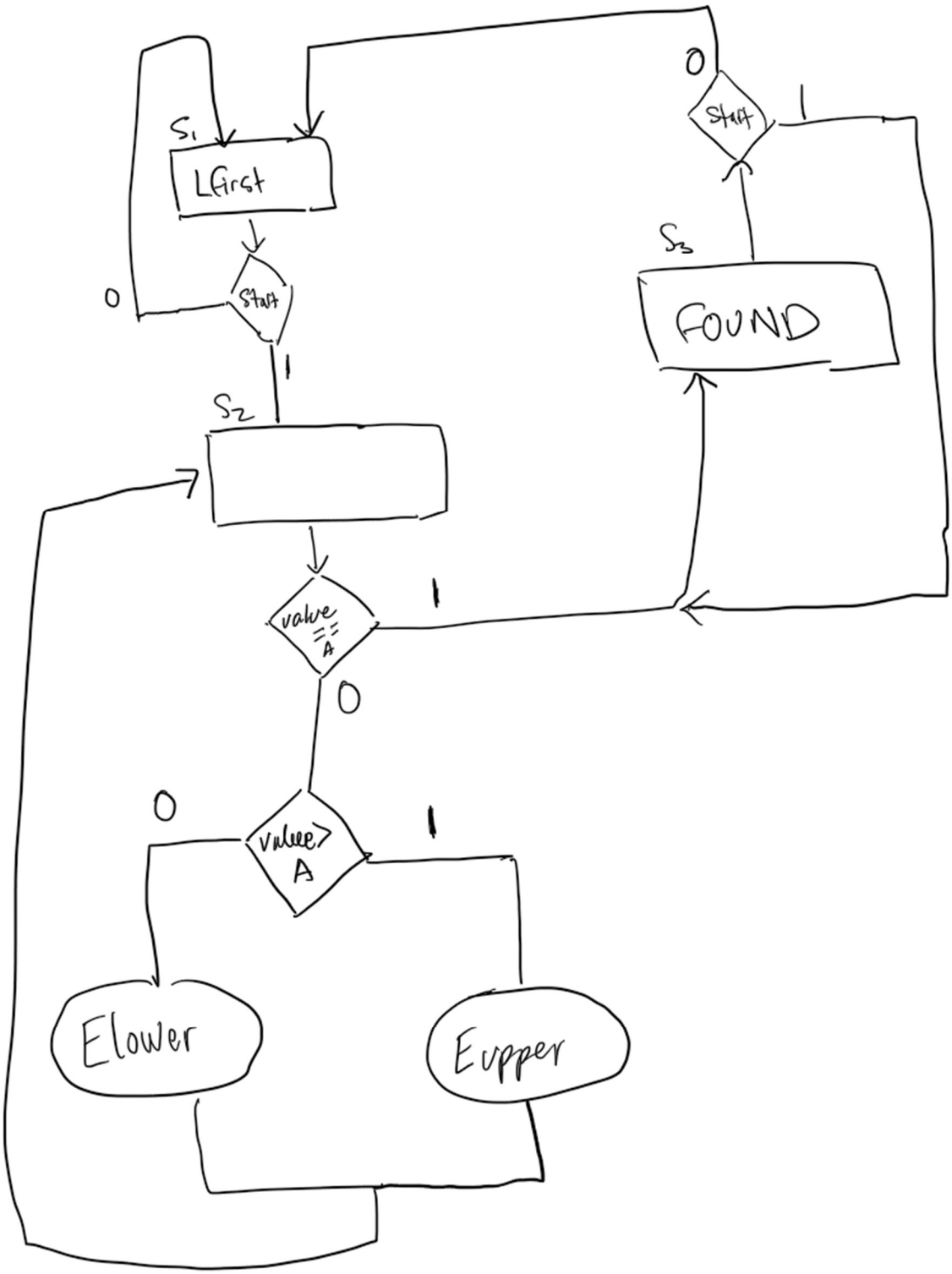

Figure 6: ASM for controller for task 2

Results

Task 1:

For the first part of task 1, we first simulated the bitcontroller module that takes in inputs of reset, start, Z, and data and outputting LA, EA representing loading and enabling A for the shifter of the data input. The module also outputs LB, and EB as the loading and enabling of B for the counter module, representing the count of 1-bit in the data binary input. This module, as seen in figure 7, the data input initially is 3, and based on start, and Z, the progression of present state is next state on the posedge clk.

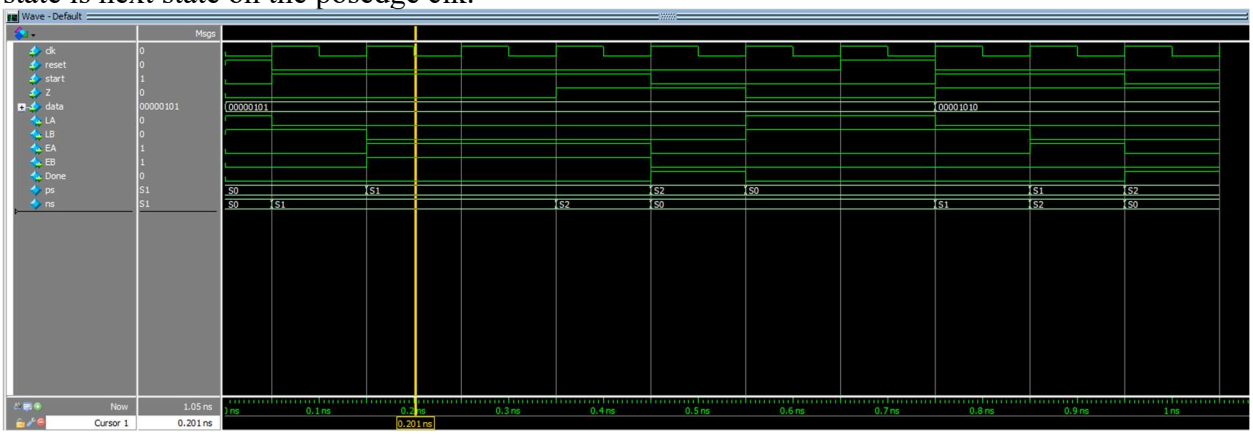

Figure 7: Waveform simulation for controller in task 1

We also tested the waveform of the datapath, making sure the inputs of LA, LB, EA, EB loads the data of A and shifts according to the input of EA, and increment the count of B on EB and displaying the value on the hex display variable named display. The output of Z is also correct, making sure the output is 1 when the data is shifted towards the value of 0.

Figure 8: Waveform simulation for datapath in task 1

Lastly, in order to ensure proper operation of the DE1_SoC in task 1, we simulated and made sure the waveform, in figure 9, outputs the correct value of the 1-bit count of a given data from

SW7-0, on the hex0 display. As well as making sure LA, LB, EA, EB is passed and obtains the correct values on the posedge clk. As seen in figure 9, the output on hex0 is the hexadecimal equivalent value to the count of 1-bit on the input binary value.

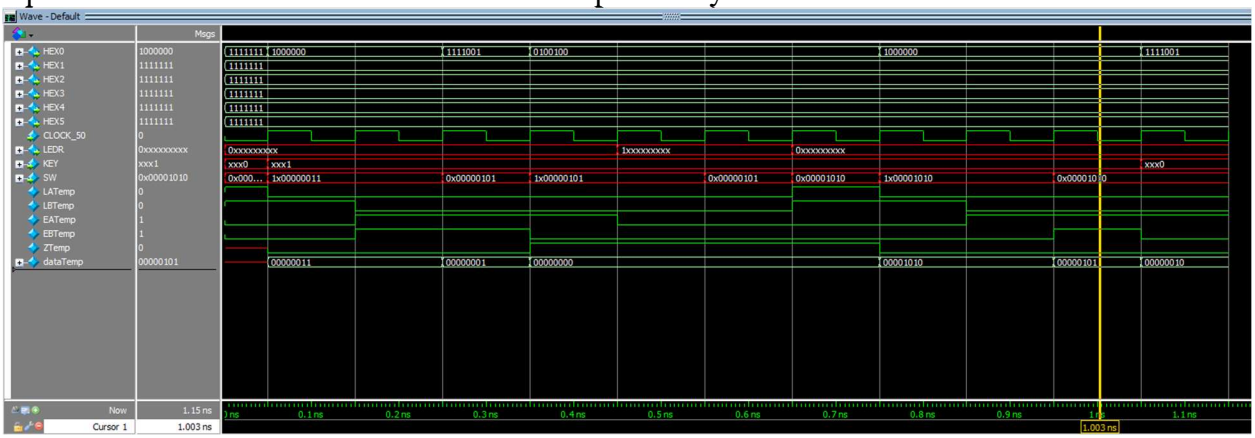

Figure 9: Waveform for DE1_SoC of task 1

Task 2:

First, we tested the simulation generated by the controller as we tested the inputs of start as well as data and value to see if the states are progressing on the condition that data is equal to value. Besides the state transition, the outputs of Found, Lfirst, Elower, and Eupper was correct along the posedge clk with Lfirst being true at S0, Found at the last state, Elower if at second state and data is greater than value, and opposite for Eupper. These outputs are for the controller to manipulate values of upper and lower to continuously input the value of the middle address in the ram.

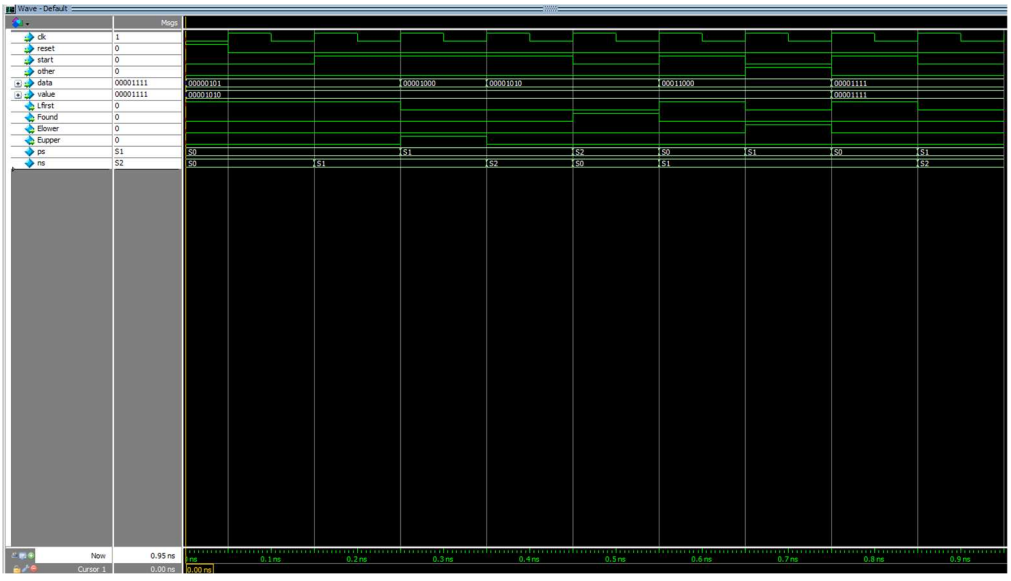

Figure 10: The waveform simulation generated by controller for task 2

As mentioned above, the datapath simulation takes in Lfirst, Elower, and Eupper, and changing lower and upper accordingly to output the middle value that is in between lower and upper.

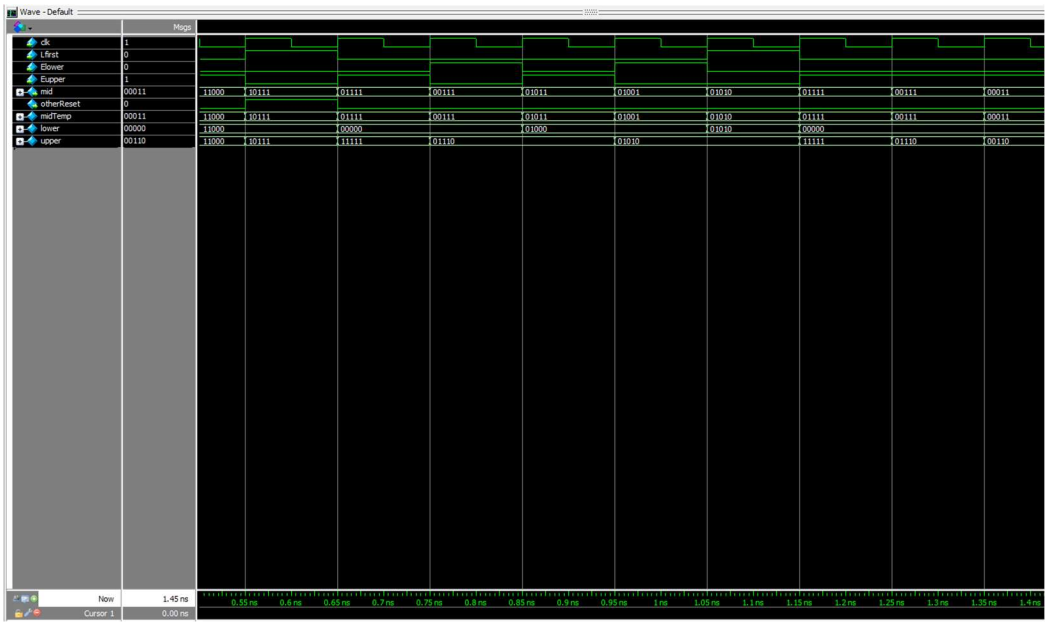

Figure 11: Waveform simulation generated by datapath for task 2

The waveform simulation for seg7 is for the display of the middle address value when the value equals the data, equaling to the found condition. As seen in figure 12, this matches the expected output.

Figure 12: Waveform simulation generated by seg7 for task 2

The simulation for DE1_SoC uses SW7-0 for the data input and SW9 for the start condition, and outputing the value of the address of the target data in the ram on hex0 and hex1, as well as found on LED9 and not found on LEDR8. As seen in figure 13, the not found light is on until the location of the given data from SW7-0 is located and the value is outputed on hex0 and hex1

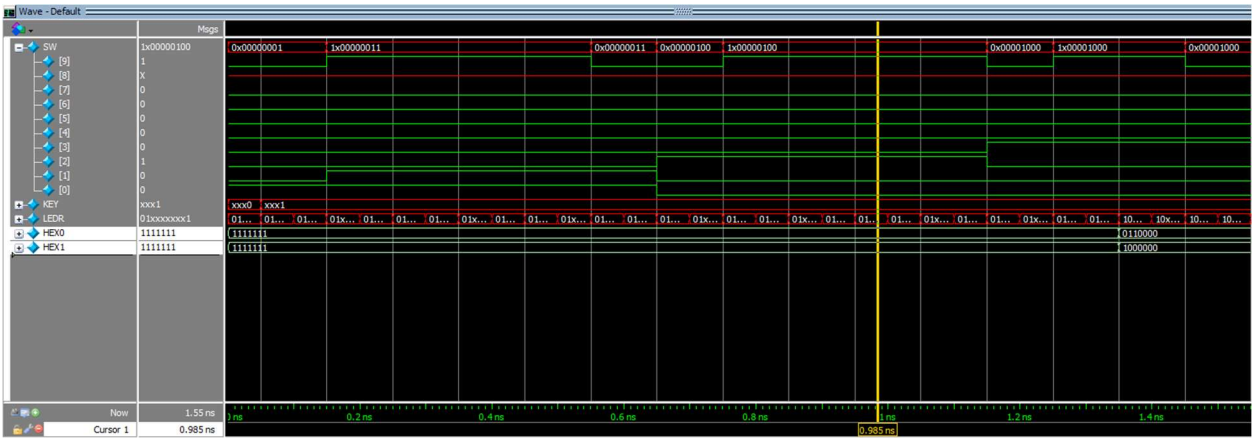

Figure 13: Waveform simulation generated by DE1_SoC for task 2

Lastly, the simulation for ram32x8 uses inputs of data, address, and wren to output the value at the given address. In this simulation, wren is 0, so the ram is only outputing values given in the memory initialization file.

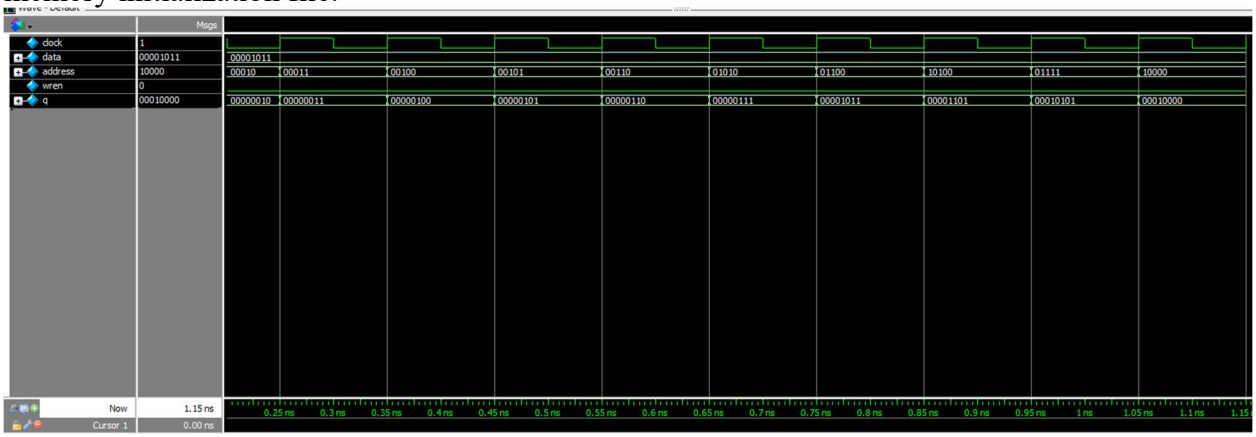

Figure 14: Waveform simulation generated by ram32x8

Final Product

This project had two separate tasks, however, with one overarching goal of implementing a HLSM using a controller and datapath to control the conditions and when certain operations in the datapath should proceed. In the first task, we designed a bitcounter that works like the conditions given in the lab spec. The bitcounter uses a controller to output LA, LB, EA, and EB and the datapath takes these inputs and either shifts A or count B. After the first task, the goal of the second task was to develop a binary search on a sorted 32x8 ram. We were able to achieve this goal with no issues as our controller was able to progress through different states and output Elower or Eupper in order to change the bounds and condense the search to a smaller range. This continues on as we were able to successfully simulate and operate the project on the FPGA perfectly. This project was extremely helpful for us as we were able to independently develop a HLSM using controller and datapath, opening up potential to develop many more different algorithms on hardware.

Appendix: SystemVerilog Code

1) bitcounter.sv (task 1)

```
1   //Khoa Tran and Ravi Sangani
2   //05/14/2020
3   //Lab 4, Task 1
4   //Module bitcounter implements a finite state machine that outputs a series
5   //of different outputs based on present state and input conditions in order
6   //for the datapath to manipulate data at the correct states. In this module
7   //inputs of clk, reset, data, and Z controls the state logic as the outputs
8   //of LA, LB, EA, EB, and Done is for loading and enabling outputs for the inputs in the
9   //datapath as Done is for indicating the program is done counting the number of
10  //1 bit in a 8-bit binary input. Present state progresses to next state at posedge clk
11  //and reset starts the present state at S0.
12  module bitcounter(clk, reset, data, start, Z, LA, LB, EA, EB, Done);
13     input logic clk, reset, start, Z;
14     input logic [7:0] data;
15     output logic LA, LB, EA, EB, Done;
16
17     //present and next state declaration
18     enum {S0, S1, S2} ps, ns;
19
20     //combinational logic going through different cases of present state
21     always_comb begin
22        case(ps)
23           S0: begin
24                 //if start then progress to next state otherwise stay at S0
25                 if (start) ns = S1;
26                 else ns = S0;
27              end
28           S1: begin
29                 //if input Z is 0, stay at S1 otherwise progress to S2
30                 if (Z == 0) ns = S1;
31                 else ns = S2;
32              end
33           S2: begin
34                 //if start then progress to next state otherwise stay at S2
35                 if (start) ns = S2;
36                 else ns = S0;
37              end
38        endcase
39     end
40
41     //sequential logic DFFs for the value of present state
42     always_ff @(posedge clk) begin
43        if (reset)
44           ps <= S0;
45        else
46           ps <= ns;
47     end
48
49     //output load B
50     assign LB = (ps == S0);
51     //output enable A
52     assign EA = (ps == S1);
53     //output Done
54     assign Done = (ps == S2);
55     //output enable B
56     assign EB = (ps == S1 && data[0] == 1);
57     //output load A
58     assign LA = (ps == S0 && start == 0);
59  endmodule
60
61  //module bitcounter_testbench is a testbench for bitcounter
62  //setting a series of inputs on reset, start, data, and Z, in order
63  //to test the progression of the present state and the outputs of
64  //LB, LA, EB, EA, and Done at the posedge clk.
65  module bitcounter_testbench();
66     logic clk, reset, start, Z;
67     logic [7:0] data;
68     logic LA, LB, EA, EB, Done;
69
70     //device under test
71     bitcounter count(.clk, .reset, .data, .start, .Z, .LA, .LB, .EA, .EB, .Done);
72
73     parameter clock_period = 100;
74
75     initial begin
76        clk <= 0;
```



```
77        forever #(clock_period /2) clk <= ~clk;
78     end
79
80     //initial simulation
81     initial begin
82        reset <= 1; start<=0; data<=8'd5; Z <= 0; @(posedge clk);
83        reset <= 0; start<=1; data<=8'd5; Z <= 0; @(posedge clk);
84        reset <= 0; start<=1; data<=8'd5; Z <= 0; @(posedge clk);
85        reset <= 0; start<=1; data<=8'd5; Z <= 0; @(posedge clk);
86        reset <= 0; start<=1; data<=8'd5; Z <= 1; @(posedge clk);
87        reset <= 0; start<=0; data<=8'd5; Z <= 1; @(posedge clk);
88        reset <= 0; start<=0; data<=8'd5; Z <= 0; @(posedge clk);
89        reset <= 1; start<=0; data<=8'd5; Z <= 0; @(posedge clk);
90        reset <= 0; start<=1; data<=8'd10; Z <= 1; @(posedge clk);
91        reset <= 0; start<=1; data<=8'd10; Z <= 1; @(posedge clk);
92        reset <= 0; start<=0; data<=8'd10; Z <= 1; @(posedge clk);
93
94        $stop;
95
96     end
97  endmodule
```

2) datapath.sv (task 1)

```
1   //Khoa Tran and Ravi Sangani
2   //05/14/2020
3   //Lab 4, Task 1
4   //Module datapath implements a datapath logic of a shifter and a counter, incrementing
5   //shifting and loading the data based on the inputs of LB, LA, EA, EB, and data
6   //These inputs are from the fsm indicating when the datapath should increment B or load B
7   //or load A or shift A. Outputs of Z is for if the data value is shifted and equals to 0 to
    stop
8   //counting, as well as display and nextData for the data output and the hex display
9   //equivalent value.
10  module datapath(clk, LB, LA, EA, EB, Z, data, display, nextData);
11     input logic clk, LB, LA, EA, EB;
12     input logic [7:0] data;
13     output logic [7:0] nextData;
14
15     output logic [6:0] display;
16     output logic Z;
17
18     //registers holding hex display value for each decimal equivalent
19     logic [3:0] B;
20     logic [6:0] zero, one, two, three, four, five, six, seven, eight;
21     assign zero = 7'b1000000;
22     assign one = 7'b1111001;
23     assign two = 7'b0100100;
24     assign three = 7'b0110000;
25     assign four = 7'b0011001;
26     assign five = 7'b0010010;
27     assign six = 7'b0000010;
28     assign seven = 7'b1111000;
29     assign eight = 7'b0000000;
30
31     //sequential logic loading B, enabling B, loading A, and enabling A for shifting
32     always_ff @(posedge clk)
33        begin
34           if (LB) B <= 0;
35           //increment B if enable B
36           if (EB) B <= B + 1;
37           if (LA) nextData <= data;
38           //shift data if enable A
39           if (EA) nextData <= nextData>>1;
40        end
41
42     assign Z = (nextData == 0);
43
44     //combinational logic for different cases of B to display the value on the hex board
45     always_comb begin
46              case (B)
47                 4'b0000: begin
48                          display = zero;
49                          end
50                 4'b0001: begin
51                          display = one;
52                          end
53                 4'b0010: begin
54                          display = two;
55                          end
56                 4'b0011: begin
57                          display = three;
58                          end
59                 4'b0100: begin
60                          display = four;
61                          end
62                 4'b0101: begin
63                          display = five;
64                          end
65                 4'b0110: begin
66                          display = six;
67                          end
68                 4'b0111: begin
69                          display = seven;
70                          end
71                 4'b1000: begin
72                          display = eight;
73                          end
74                 default: begin
75                          display = 7'b1111111;
```

```
                        end
                endcase
        end
endmodule

//module datapath_testbench is a testbench for datapath
//setting a series of inputs on LB, EB, LA, EA, and data, in order
//to test the output of nextData, display, and Z on the posedge clk
module datapath_testbench();
    logic clk, LB, LA, EA, EB;
    logic [7:0] data;
    logic [7:0] nextData;

    logic [6:0] display;
    logic Z;

    //device under test
    datapath path(.clk, .LB, .LA, .EA, .EB, .Z, .data, .display, .nextData);

    parameter clock_period = 100;

    initial begin
        clk <= 0;
        forever #(clock_period /2) clk <= ~clk;
    end

    //initial simulation
    initial begin
        LB<=0; data<=8'd5; LA<= 0; EB<=0; EA<=0; @(posedge clk);
        LB<=0; data<=8'd5; LA<= 1; EB<=0; EA<=0; @(posedge clk);
        LB<=0; data<=8'd5; LA<= 1; EB<=0; EA<=1; @(posedge clk);
        LB<=1; data<=8'd5; LA<= 1; EB<=0; EA<=1; @(posedge clk);
        LB<=0; data<=8'd5; LA<= 0; EB<=1; EA<=1; @(posedge clk);
        LB<=0; data<=8'd5; LA<= 0; EB<=1; EA<=1; @(posedge clk);
        LB<=0; data<=8'd5; LA<= 0; EB<=1; EA<=1; @(posedge clk);
        LB<=0; data<=8'd5; LA<= 0; EB<=1; EA<=1; @(posedge clk);
        $stop;

    end
endmodule

```

3) DE1_SoC.sv (task 1)

```
1   //Khoa Tran and Ravi Sangani
2   //05/14/2020
3   //Lab 4, Task 1
4   //This module DE1_SoC controls the logic of inputs from the FPGA using KEY[0] as the reset
5   //SW[9] as the key to start the program, and SW7-0 as the data input for the datapath and
    bitcounter
6   //which together represents a bit counter that counts the number of 1 bit in the input of
    SW7-0 and display
7   //the decimal value on the hex0 board display as well as LEDR[9] to indicate the program is
    done counting.
8   //This module uses a fsm that controls the state of enabling and loading both B and A in
    order to increment
9   //or shift values based on the current state. Using CLOCK_50, bitcounter and datapath works
    in conjunction to
10  //count the number of 1 bit in a given data value in binary.
11  module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
12     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
13     input logic CLOCK_50;
14     output logic [9:0] LEDR;
15     input logic [3:0] KEY;
16     input logic [9:0] SW;
17
18     logic LATemp, LBTemp, EATemp, EBTemp;
19     logic ZTemp;
20     logic [7:0] dataTemp;
21
22     //instantiation of bitcounter, outputing LA, LB, EA, EB, and Done, with inputs from
    datapath for data and Z, as well as KEY[0] for reset and SW[9] for start
23     bitcounter controller(.clk(CLOCK_50), .reset(~KEY[0]), .data(dataTemp), .start(SW[9]), .Z
    (ZTemp), .LA(LATemp), .LB(LBTemp), .EA(EATemp), .EB(EBTemp), .Done(LEDR[9]));
24     //instantiation of datapath, with inputs of LA, LB, EA, and EB from bitcounter, and data
    from the SW7-0 and outputing on Z, display, and nextData
25     datapath data(.clk(CLOCK_50), .LB(LBTemp), .LA(LATemp), .EA(EATemp), .EB(EBTemp), .Z(
    ZTemp), .data(SW[7:0]), .display(HEX0), .nextData(dataTemp));
26
27     assign HEX1 = 7'b1111111;
28     assign HEX2 = 7'b1111111;
29     assign HEX3 = 7'b1111111;
30     assign HEX4 = 7'b1111111;
31     assign HEX5 = 7'b1111111;
32
33  endmodule
34
35  //Testing the DE1_SoC module by implementing a series of inputs on KEY[0] as reset,
36  //SW[9] as start, and SW[7:0] as the data and testing if the outputs of
37  //LEDR[9], HEX0 is correct along the posedge CLOCK_50
38  module DE1_SoC_testbench();
39     logic CLOCK_50;
40     logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
41     logic [9:0] LEDR;
42     logic [3:0] KEY;
43     logic [9:0] SW;
44
45     //device under test
46     DE1_SoC dut (.CLOCK_50, .HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW);
47
48     parameter clock_period = 100;
49
50      initial begin
51        CLOCK_50 <= 0;
52        forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50 ;
53      end
54
55      //initial simulation
56      initial begin
57      KEY[0] <= 0; SW[9] <= 0; SW[7:0] <= 8'd3; @(posedge CLOCK_50);
58      KEY[0] <= 1; SW[9] <= 1; SW[7:0] <= 8'd3; @(posedge CLOCK_50);
59      KEY[0] <= 1; SW[9] <= 1; SW[7:0] <= 8'd3; @(posedge CLOCK_50);
60      KEY[0] <= 1; SW[9] <= 0; SW[7:0] <= 8'd5; @(posedge CLOCK_50);
61      KEY[0] <= 1; SW[9] <= 1; SW[7:0] <= 8'd5; @(posedge CLOCK_50);
62      KEY[0] <= 1; SW[9] <= 1; SW[7:0] <= 8'd5; @(posedge CLOCK_50);
63      KEY[0] <= 1; SW[9] <= 0; SW[7:0] <= 8'd5; @(posedge CLOCK_50);
64      KEY[0] <= 1; SW[9] <= 0; SW[7:0] <= 8'd10; @(posedge CLOCK_50);
65      KEY[0] <= 1; SW[9] <= 1; SW[7:0] <= 8'd10; @(posedge CLOCK_50);
66      KEY[0] <= 1; SW[9] <= 1; SW[7:0] <= 8'd10; @(posedge CLOCK_50);
67      KEY[0] <= 1; SW[9] <= 0; SW[7:0] <= 8'd10; @(posedge CLOCK_50);
```



```
68      KEY[0] <= 0; SW[9] <= 0; SW[7:0] <= 8'd10; @(posedge CLOCK_50);
69      $stop;
70    end
71
72  endmodule
```

4) controller.sv (task 2)

```
1   //Khoa Tran and Ravi Sangani
2   //05/14/2020
3   //Lab 4, Task 2
4   //Module controller implements an fsm that outputs load and enable logic based on inputs and
5   //present state in order for the datapath to manipulate lower and upper bounds of the ram
6   //memory in order to do a binary search for the given data input. Output Lfirst is for
    initial
7   //loading and Elower and Eupper is for enabling the change of either lower and upper,
    updating the
8   //values based on the condition of if data is larger or less than value when its not equal.
9   //Input start and reset is for starting the finite state machine as well as reseting to
    initial state.
10  //Present state progresses on posedge clk and outputs based on present state and comparing
    inputs values.
11  module controller(clk, reset, data, value, start, Lfirst, Found, Elower, Eupper, other);
12     input logic clk, reset, start, other;
13     input logic [7:0] data, value;
14     output logic Lfirst, Found, Elower, Eupper;
15
16     //present and next states
17     enum {S0, S1, S2} ps, ns;
18
19     //combinational logic of state progression
20     always_comb begin
21        case(ps)
22           //initial state
23           S0: begin
24                 if (start) ns = S1;
25                 else ns = S0;
26              end
27           //state of comparing value == data to either continue searching or progress to
    found state
28           S1: begin
29                 if ((value == data) && (~other)) ns = S2;
30                 else ns = S1;
31              end
32           //found state
33           S2: begin
34                 if (start) ns = S2;
35                 else ns = S0;
36              end
37        endcase
38     end
39
40
41     //sequential logic of loading present state
42     always_ff @(posedge clk) begin
43        if (reset || other)
44           ps <= S0;
45        else
46           ps <= ns;
47     end
48
49     //output of first state
50     assign Lfirst = (ps == S0);
51
52     //output of found state
53     assign Found = (ps == S2);
54     //output of Elower if at S1 and value < data
55     assign Elower = (ps == S1) && (value != data) && (value < data);
56     //output of Eupper if at S1 and value > data
57     assign Eupper = (ps == S1) && (value != data) && (value > data);
58  endmodule
59
60  //module controller_testbench is a testbench for controller
61  //setting a series of inputs on reset, start, data, value, and other, in order
62  //to test the progression of the present state and the outputs of
63  //Lfirst, Eupper, Elower, and Found at the posedge clk.
64  module controller_testbench();
65     logic clk, reset, start, other;
66     logic [7:0] data, value;
67
68     logic Lfirst, Found, Elower, Eupper;
69
70     //device under test
71     controller count(.clk, .reset, .data, .value, .start, .Lfirst, .Found, .Elower, .Eupper,
```



```
       .other);
72
73     parameter clock_period = 100;
74
75     initial begin
76        clk <= 0;
77        forever #(clock_period /2) clk <= ~clk;
78     end
79
80     //initial simulation
81     initial begin
82        reset <= 1; other<=0; data<=8'd5; value <= 8'd10; start<=0; @(posedge clk);
83        reset <= 0; other<=0; data<=8'd5; value <= 8'd10; start<=0; @(posedge clk);
84        reset <= 0; other<=0; data<=8'd5; value <= 8'd10; start<=1; @(posedge clk);
85        reset <= 0; other<=0; data<=8'd8; value <= 8'd10; start<=1; @(posedge clk);
86        reset <= 0; other<=0; data<=8'd10; value <= 8'd10; start<=1; @(posedge clk);
87        reset <= 0; other<=0; data<=8'd10; value <= 8'd10; start<=0; @(posedge clk);
88        reset <= 0; other<=0; data<=8'd24; value <= 8'd10; start<=1; @(posedge clk);
89        reset <= 0; other<=1; data<=8'd24; value <= 8'd10; start<=0; @(posedge clk);
90        reset <= 0; other<=0; data<=8'd15; value <= 8'd15; start<=1; @(posedge clk);
91        reset <= 0; other<=0; data<=8'd15; value <= 8'd15; start<=0; @(posedge clk);
92        $stop;
93
94     end
95  endmodule
```

5) datapath.sv (task 2)

```
1  //Khoa Tran and Ravi Sangani
2  //05/14/2020
3  //Lab 4, Task 2
4  //Module datapath controls the output of the mid address based on lower and upper bounds
5  //and otherReset based on the condition if lower is greater than upper indicating the program
6  //is done searching. This module takes the input of Lfirst, Elower, Eupper in order to
7  //know when and what to update lower and upper bounds to. Lower and upper changes on the posedge clk,
8  //based on input conditions, and output mid as the middle value of lower and upper.
9  module datapath(clk, Lfirst, Elower, Eupper, mid, otherReset);
10    input logic clk;
11    input logic Lfirst, Elower, Eupper;
12    output logic [4:0] mid;
13    output logic otherReset;
14    logic [4:0] midTemp;
15    logic [4:0] lower;
16    logic [4:0] upper;
17
18    //logic storing mid value
19    assign midTemp = (lower + upper)/2;
20
21    //sequential logic of lower and upper based on input conditions
22    always_ff @(posedge clk)
23       begin
24          if (Lfirst) begin
25             lower <= 5'd0;
26             upper <= 5'd31;
27          end
28          if (Elower) lower <= midTemp + 1;
29          if (Eupper) upper <= midTemp - 1;
30       end
31
32    //output mid
33    assign mid = midTemp;
34    //output otherRest based on lower > upper
35    assign otherReset = (lower > upper);
36
37 endmodule
38
39 //module datapath_testbench is a testbench for datapath
40 //setting a series of inputs on Lfirst, Elower, and Eupper, in order
41 //to test the output of mid and otherReset on the posedge clk
42 module datapath_testbench();
43    logic clk, Lfirst, Elower, Eupper;
44
45    logic [4:0] mid;
46    logic otherReset;
47
48
49    //device under test
50    datapath path(.clk, .Lfirst, .Elower, .Eupper, .mid, .otherReset);
51
52    parameter clock_period = 100;
53
54    initial begin
55       clk <= 0;
56       forever #(clock_period /2) clk <= ~clk;
57    end
58
59    //initial simulation
60    initial begin
61       Lfirst<=1; Elower<=0; Eupper<= 0; @(posedge clk);
62       Lfirst<=0; Elower<=1; Eupper<= 0; @(posedge clk);
63       Lfirst<=0; Elower<=1; Eupper<= 0; @(posedge clk);
64       Lfirst<=0; Elower<=0; Eupper<= 1; @(posedge clk);
65       Lfirst<=0; Elower<=0; Eupper<= 1; @(posedge clk);
66       Lfirst<=0; Elower<=0; Eupper<= 1; @(posedge clk);
67       Lfirst<=1; Elower<=0; Eupper<= 0; @(posedge clk);
68       Lfirst<=0; Elower<=0; Eupper<= 1; @(posedge clk);
69       Lfirst<=0; Elower<=1; Eupper<= 0; @(posedge clk);
70       Lfirst<=0; Elower<=0; Eupper<= 1; @(posedge clk);
71       Lfirst<=0; Elower<=1; Eupper<= 0; @(posedge clk);
72       Lfirst<=1; Elower<=0; Eupper<= 0; @(posedge clk);
73       Lfirst<=0; Elower<=0; Eupper<= 1; @(posedge clk);
74       Lfirst<=0; Elower<=0; Eupper<= 1; @(posedge clk);
75       Lfirst<=0; Elower<=0; Eupper<= 1; @(posedge clk);
76       $stop;
77
78    end
79 endmodule
80
```

6) seg7.sv (task 2)

```
1   //Khoa Tran and Ravi Sangani
2   //05/14/2020
3   //Lab 4, Task 2
4   //Module seg7 controls the output to the hex1 and hex0 display based on inputs of found
5   //and address as if found is 0, then output to the display nothing, but if it is found,
6   //then output the value of the address in hexadecimal equivalent on the hex1 and hex0
    display.
7   module seg7(addr, found, hex5, hex4);
8      input logic [4:0] addr;
9      input logic found;
10
11     output logic [6:0] hex5, hex4;
12
13     //logic of hex display values equivalent to hexadecimal value
14     logic [6:0] zero, one, two, three, four, five, six, seven, eight, nine, A, B, C, D, E, F;
15     assign zero = 7'b1000000;
16     assign one = 7'b1111001;
17     assign two = 7'b0100100;
18     assign three = 7'b0110000;
19     assign four = 7'b0011001;
20     assign five = 7'b0010010;
21     assign six = 7'b0000010;
22     assign seven = 7'b1111000;
23     assign eight = 7'b0000000;
24     assign nine = 7'b0010000;
25     assign A = 7'b0001000;
26     assign B = 7'b0000011;
27     assign C = 7'b1000110;
28     assign D = 7'b0100001;
29     assign E = 7'b0000110;
30     assign F = 7'b0001110;
31
32     //combinational logic of outputing hex5, and hex4 based on found condition and cases of
    addr
33     always_comb begin
34        if (~found) begin
35            hex5 = 7'b1111111;
36            hex4 = 7'b1111111;
37        end
38        else begin
39
40        case(addr)
41                  5'b00000: begin
42                            hex5 = zero;
43                            hex4 = zero;
44                            end
45                  5'b00001: begin
46                            hex5 = zero;
47                            hex4 = one;
48                            end
49                  5'b00010: begin
50                            hex5 = zero;
51                            hex4 = two;
52                            end
53                  5'b00011: begin
54                            hex5 = zero;
55                            hex4 = three;
56                            end
57                  5'b00100: begin
58                            hex5 = zero;
59                            hex4 = four;
60                            end
61                  5'b00101: begin
62                            hex5 = zero;
63                            hex4 = five;
64                            end
65                  5'b00110: begin
66                            hex5 = zero;
67                            hex4 = six;
68                            end
69                  5'b00111: begin
70                            hex5 = zero;
71                            hex4 = seven;
72                            end
73                  5'b01000: begin
74                            hex5 = zero;
```

```
                        hex4 = eight;
                        end
                5'b01001: begin
                        hex5 = zero;
                        hex4 = nine;
                        end
                5'b01010: begin
                        hex5 = zero;
                        hex4 = A;
                        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
        end

endmodule

//Module seg7_testbench is a testbench for seg7 testing the outputs of
//hex5 and hex4 is correct based on the given inputs of addr and found
module seg7_testbench();
    logic [4:0] addr;
    logic found;

    logic [6:0] hex5, hex4;

    //instatiation of seg7
    seg7 disp(.addr, .found, .hex5, .hex4);

    initial begin
        addr=5'd0;  found = 0; #10;
        addr=5'd2;  found = 0; #10;
        addr=5'd3;  found = 0; #10;
        addr=5'd4;  found = 0; #10;
        addr=5'd5;  found = 0; #10;
        addr=5'd2;  found = 1; #10;
        addr=5'd3;  found = 1; #10;
        addr=5'd4;  found = 1; #10;
        addr=5'd5;  found = 1; #10;
        $stop; //end simulation
    end
endmodule
```

7) ram32x8_testbench (task 2)

```
//Khoa Tran and Ravi Sangani
//05/14/2020
//Lab 4, Task 2
//Module ram32x8 testbench test for inputs of data, address, and wren and testing
//if the output on q is correct, displaying the value at the address on the posedge clk
`timescale 1 ps / 1 ps
module ram32x8_testbench();
   reg    clock;
   reg    [7:0]  data;
   reg    [4:0]  address;
   reg      wren;
   wire   [7:0]  q;

   //device under test
   ram32x8 dut(.address(address), .clock(clock), .data(data), .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 <= 8'd11; address <= 5'd0;   wren<=0; @(posedge clock);
      data <= 8'd11; address <= 5'd1;   wren<=0; @(posedge clock);
      data <= 8'd11; address <= 5'd2;   wren<=0; @(posedge clock);
      data <= 8'd11; address <= 5'd3;   wren<=0; @(posedge clock);
      data <= 8'd11; address <= 5'd4;   wren<=0; @(posedge clock);
      data <= 8'd11; address <= 5'd5;   wren<=0; @(posedge clock);
      data <= 8'd11; address <= 5'd6;   wren<=0; @(posedge clock);
      data <= 8'd11; address <= 5'd10;   wren<=0;    @(posedge clock);
      data <= 8'd11; address <= 5'd12;   wren<=0;    @(posedge clock);
      data <= 8'd11; address <= 5'd20;   wren<=0;    @(posedge clock);
      data <= 8'd11; address <= 5'd15;   wren<=0;    @(posedge clock);
      data <= 8'd11; address <= 5'd16;   wren<=0;    @(posedge clock);

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

8) DE1_SoC.sv (task 2)

```
1   //Khoa Tran and Ravi Sangani
2   //05/14/2020
3   //Lab 4, Task 2
4
5   //Module DE1_SoC implements a binary search algorithm on hardware using a 32x8 ram
6   //that has a sorted set of values incrementing with incrementing addresses. The binary
    search algorithm
7   //uses a controller to output conditions and states for the datapath to manipulate lower
    and upper
8   //bound values to output a mid address to the ram to output the value at the mid address.
    This
9   //gets compared with the target data from SW7-0 and continues searching if they don't
    match. Outputs
10  //found on LEDR9 and not found on LEDR8, and the mid address on HEX1 and HEX0 if the data
    is found on the
11  //ram. Also has input of start to indicate when to start the algorithm. Using CLOCK_50,
    states and conditions
12  //progress in order to implement binary search on hardware.
13  module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
14     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
15     input logic CLOCK_50;
16     output logic [9:0] LEDR;
17     input logic [3:0] KEY;
18     input logic [9:0] SW;
19
20     logic [7:0] dataTemp;
21     logic [7:0] valueTemp;
22     logic [4:0] readAddr;
23     logic Lfirst, Elower, Eupper;
24     logic foundTemp;
25
26     logic [31:0] div_clk;
27     parameter whichClock = 12;
28     //instantiation of clock_divider
29     clock_divider cdiv (.clock(CLOCK_50), .divided_clocks(div_clk));
30     logic clkSelect;
31     assign clkSelect = CLOCK_50; // for simulation
32
33     //assign clkSelect = div_clk[whichClock]; // for board
34     assign LEDR[0] = clkSelect;
35     logic otherTemp;
36
37     //instantiation of controller module with data input and ram mid value input as well as
    SW[9] for start, outputing Lfirst, Elower, Eupper for datapath, and found for LEDR
38     controller control(.clk(clkSelect), .reset(~KEY[0]), .data(SW[7:0]), .value(valueTemp), .
    start(SW[9]), .Lfirst, .Found(foundTemp), .Elower, .Eupper, .other(otherTemp));
39     //instantiation of ram module, getting value at mid address from datapath
40     ram32x8 ram(.address(readAddr),  .clock(CLOCK_50), .data(8'd0),   .wren(1'b0), .q(
    valueTemp));
41     //instantiation of datapath using inputs from controller to output mid value to ram and
    otherReset
42     datapath data(.clk(clkSelect), .Lfirst, .Elower, .Eupper, .mid(readAddr), .otherReset(
    otherTemp));
43     //instantiation of seg7, displaying mid address value based on if found condition is true
44     seg7 display(.addr(readAddr), .found(foundTemp), .hex5(HEX1), .hex4(HEX0));
45
46     //found and not found LEDR outputs
47     assign LEDR[9] = foundTemp;
48     assign LEDR[8] = ~LEDR[9];
49
50     //turn of other hex
51     assign HEX2 = 7'b1111111;
52     assign HEX3 = 7'b1111111;
53     assign HEX4 = 7'b1111111;
54     assign HEX5 = 7'b1111111;
55
56  endmodule
57
58  //Module testbench for DE1_SoC, testing for outputs of HEX1, HEX0, LEDR9, LEDR8
59  //is correct along the posedge CLOCK_50 with a series of inputs on
60  //KEY[0] as reset, SW[9] as start, and SW7-0 as the target data to search
61  `timescale 1 ps / 1 ps
62  module DE1_SoC_testbench();
63     logic CLOCK_50;
64     logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
65     logic [9:0] LEDR;
```

```
66      logic [3:0] KEY;
67      logic [9:0] SW;
68
69      DE1_SoC dut (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
70
71      parameter clock_period = 100;
72
73         initial begin
74            CLOCK_50 <= 0;
75            forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
76
77         end //initial
78
79      initial begin
80         KEY[0] <= 1'b0; SW[9] <= 1'b0; SW[7:0] <= 4'b0001;  @(posedge CLOCK_50);
81         KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[7:0] <= 4'b0001;  @(posedge CLOCK_50);
82         KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[7:0] <= 4'd3;  @(posedge CLOCK_50);
83         KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[7:0] <= 4'd3;  @(posedge CLOCK_50);
84         KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[7:0] <= 4'd3;  @(posedge CLOCK_50);
85         KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[7:0] <= 4'd3;  @(posedge CLOCK_50);
86         KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[7:0] <= 4'd3;  @(posedge CLOCK_50);
87         KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[7:0] <= 4'd20;  @(posedge CLOCK_50);
88         KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[7:0] <= 4'd20;  @(posedge CLOCK_50);
89         KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[7:0] <= 4'd20;  @(posedge CLOCK_50);
90         KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[7:0] <= 4'd20;  @(posedge CLOCK_50);
91         KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[7:0] <= 4'd20;  @(posedge CLOCK_50);
92         KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[7:0] <= 4'd24;  @(posedge CLOCK_50);
93         KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[7:0] <= 4'd24;  @(posedge CLOCK_50);
94         KEY[0] <= 1'b1; SW[9] <= 1'b1; SW[7:0] <= 4'd24;  @(posedge CLOCK_50);
95         KEY[0] <= 1'b1; SW[9] <= 1'b0; SW[7:0] <= 4'd24;  @(posedge CLOCK_50);
96
97         $stop;
98      end
99
100  endmodule
101
102  //clock_divider module has inputs of the clock, reset, and the
103  //32 bit divided_clock which allows to sequence through and
104  //output whenever the postive edge has been reached on the clock
105  // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
106  module clock_divider (clock, divided_clocks);
107     input logic clock;
108     output logic [31:0] divided_clocks = 0;
109
110     always_ff @(posedge clock) begin
111        divided_clocks <= divided_clocks + 1;
112     end
113
114  endmodule
115
```