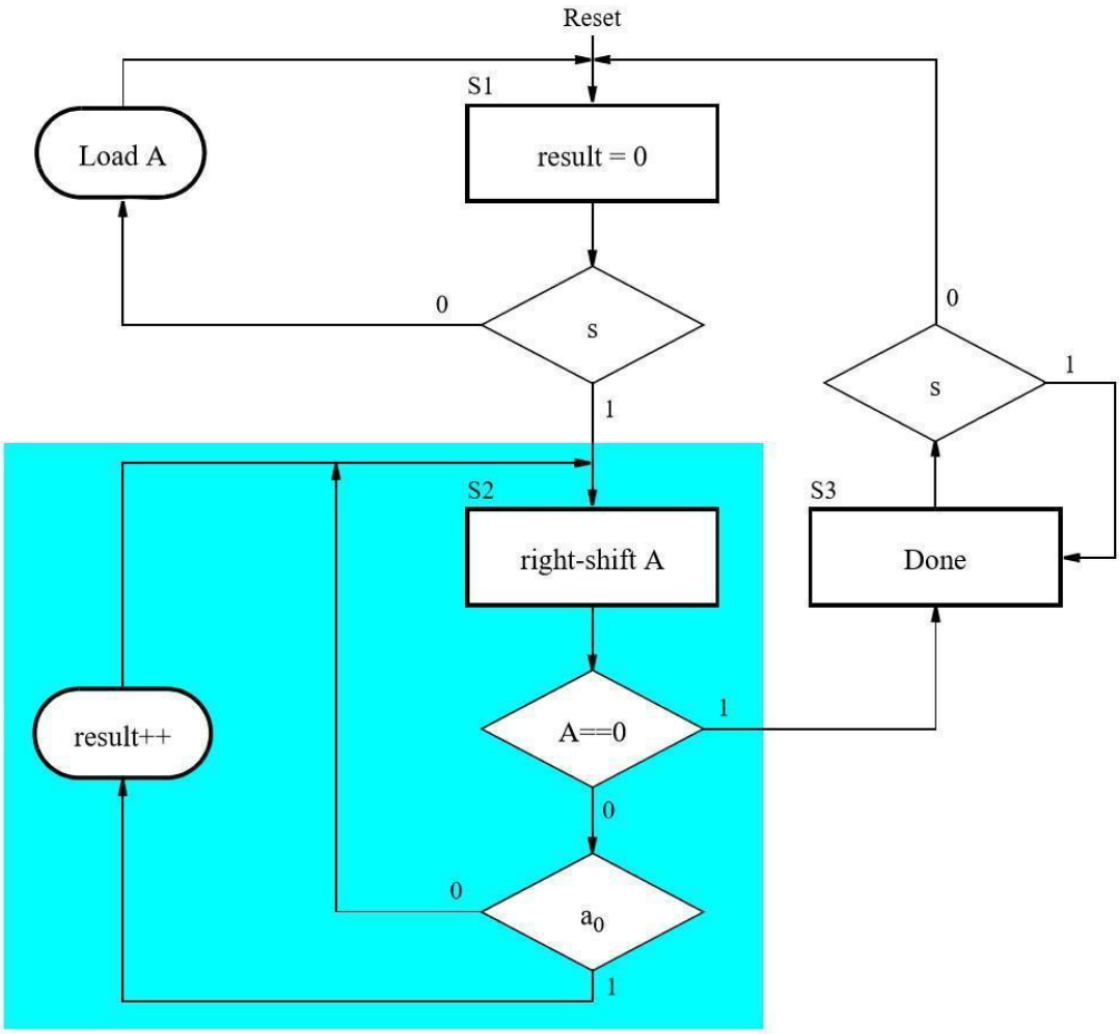
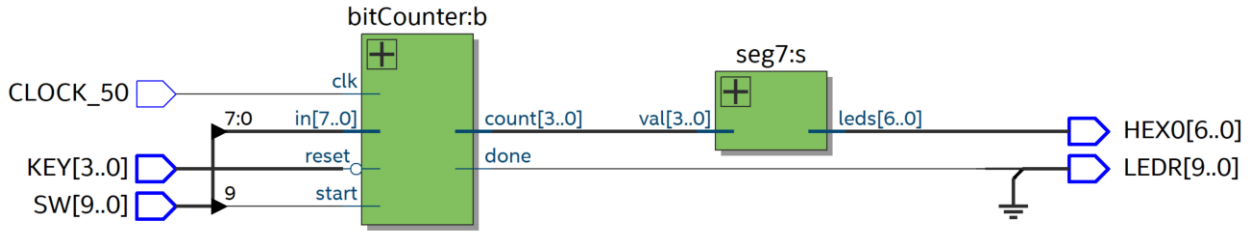
Alan Li
EE 371
February 12th, 2022
Lab 3 Report

# Task1

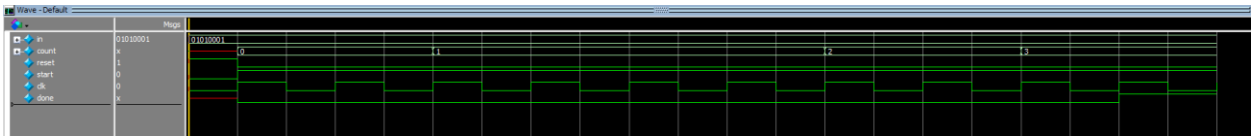
## Section 1: Procedure






For the bit counter, I followed the ASMD diagram provided in the lab document. There were 3 states: initial state (S1), count state (S2), and finish state (S3). At the initial state, the counter value is set to 0 and the system loads the input value. Once the start signal is on, the system goes to count state and checks the last digit of the data. If the last digit is 1, counter increases by 1, otherwise it would not change. Then it shifts the data to right by 1. Once the data becomes 0, count finishes and the system stay at finish state until start signal is off.

## Section 2: Results



The waveform shows that the counter goes to 3 and stopped.

## Section 3: Appendix

```
1  // Alan Li
2  // 02/25/2022
3  // EE 371
4  // Lab #4
5
6  // DE1_SoC takes 3-bit KEY and 10-bit SW as input and return 7-bit HEX and 10-bit LEDR as output.
7  // Switch 9 is used to start counting and when KEY0 is pressed, the system would go back to initial state and count again
8  // The value of counter will be displayed on HEX display, after the counting is done LEDR[9] will turn on
9  // This serves as top-level module for the bitCounter system
10
11 module DE1_SoC (SW, KEY, LEDR, HEX0, CLOCK_50);
12    input logic [9:0] SW;
13    input logic [3:0] KEY;
14    input logic CLOCK_50;
15    output logic [9:0] LEDR;
16    output logic [6:0] HEX0;
17    logic [3:0] count;
18
19    bitCounter b (.in(SW[7:0]), .reset(~KEY[0]), .start(SW[9]), .clk(CLOCK_50), .count, .done(LEDR[9]));
20    seg7 s (.val(count), .leds(HEX0));
21 endmodule
22
```

```
1  // Alan Li
2  // 02/25/2022
3  // EE 371
4  // Lab #4
5
6  // This module displays the value on the hex led display
7  module seg7(val, leds);
8     input logic [3:0] val;
9     output logic [6:0] leds;
10
11    // control the led display with binary numbers
12    always_comb begin case
13    (val)
14    //           Light: 6543210
15       4'b0000: leds = 7'b1000000; // 0
16       4'b0001: leds = 7'b1111001; // 1
17       4'b0010: leds = 7'b0100100; // 2
18       4'b0011: leds = 7'b0110000; // 3
19       4'b0100: leds = 7'b0011001; // 4
20       4'b0101: leds = 7'b0010010; // 5
21       4'b0110: leds = 7'b0000010; // 6
22       4'b0111: leds = 7'b1111000; // 7
23       4'b1000: leds = 7'b0000000; // 8
24       4'b1001: leds = 7'b0010000; // 9
25       4'b1010: leds = 7'b0001000; // A
26       4'b1011: leds = 7'b0000011; // b
27       4'b1100: leds = 7'b1000110; // C
28       4'b1101: leds = 7'b0100001; // d
29       4'b1110: leds = 7'b0000110; // E
30       4'b1111: leds = 7'b0001110; // F
31       default: leds = 7'bX;   endcase
32    end
33 endmodule
34
```

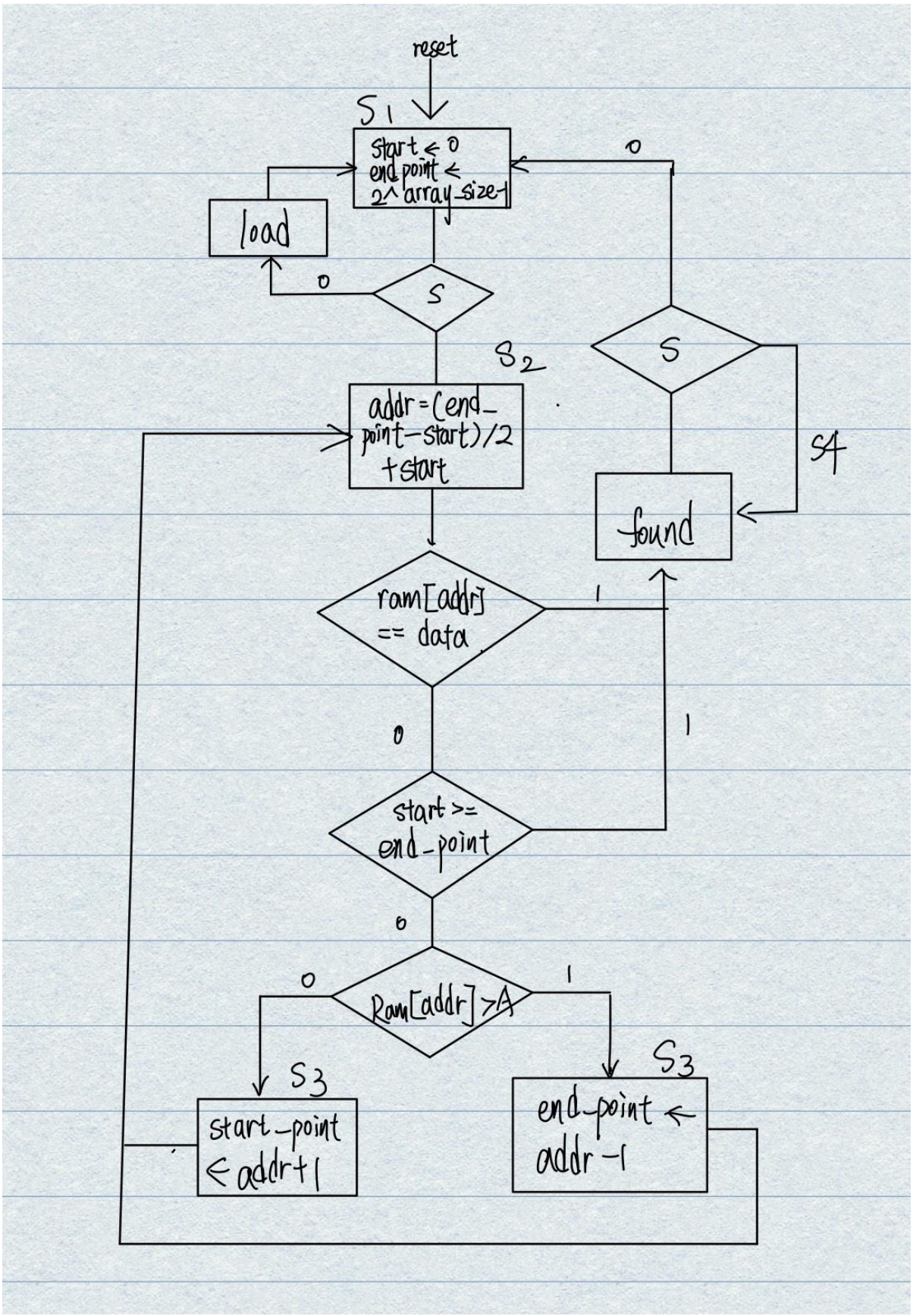
```
// Alan Li
// 02/25/2022
// EE 371
// Lab #4

// bitCounter takes 8-bit in, 1-bit reset, s, clk as input and return 8 bit count and 1-bit done signal
// This module display the number of 1s counted in the input data on the 7-segment display HEX0, and signal that the
// algorithm is finished by lighting up LEDR9.
// The module will not run until a start signal is given

module bitCounter #(parameter width = 8, parameter outWidth = 4)
                   (
                    input logic [width - 1:0] in,  // input value from switches
                    input logic reset, start, clk,
                    output logic [outWidth - 1:0] count,
                    output logic done
                   );

   logic [width - 1:0] A;  // remaining values to count "1"

   // when start signal is triggered, system enter the counting stage, after the counting is finished,
   // system wait until start signal is off. Then system go back to initial state
   enum {S1, S2, S3} ps, ns;

   always_comb begin
      case (ps)
         S1: if (start) ns = S2;  // start
             else ns = S1;
         S2: if (done) ns = S3;
             else ns = S2;
         S3: if (start) ns = S3;
             else ns = S1;  // start set back to 0
      endcase
   end

   // In stage 1, initialize all registers
   // In stage 2, starting the counting process
   always_ff@(posedge clk) begin
      if (reset)
         ps <= S1;
      else
         ps <= ns;

      case (ps)
         S1: begin
             A <= in;
             count <= 0;
             done <= 0;
             end

         S2: begin
             done <= (A == 0);
             if (A[0] == 1)
               count <= count + 1;
             A <= A / 2;  // shift to right by 1 bit
```

```
36      // In stage 1, initialize all registers
37      // In stage 2, starting the counting process
38      always_ff@(posedge clk) begin
39         if (reset)
40            ps <= S1;
41         else
42            ps <= ns;
43
44         case (ps)
45            S1: begin
46                A <= in;
47                count <= 0;
48                done <= 0;
49                end
50
51            S2: begin
52                done <= (A == 0);
53                if (A[0] == 1)
54                  count <= count + 1;
55                A <= A / 2;  // shift to right by 1 bit
56                end
57         endcase
58      end
59   endmodule
60
61   // tests the previous module
62   module bitCounter_testbench ();
63      logic [7:0] in;
64      logic [3:0] count;
65      logic reset, start, clk, done;
66
67      bitCounter b (.*);
68
69      parameter CLOCK_PERIOD = 100;
70      initial begin
71         clk <= 0;
72         forever #(CLOCK_PERIOD/2) clk <= ~clk;
73      end
74
75      initial begin
76         reset <= 1; start <= 0; in <= 7'b11010001; @(posedge clk);
77         reset <= 0; start <= 1;                    @(posedge clk);
78                                                 @(posedge clk);
79                                                 @(posedge clk);
80                                                 @(posedge clk);
81                                                 @(posedge clk);
82                                                 @(posedge clk);
83                                                 @(posedge clk);
84                                                 @(posedge clk);
85                                                 @(posedge clk);
86                                                 @(posedge clk);
87         $stop;
88      end
89   endmodule
90
```

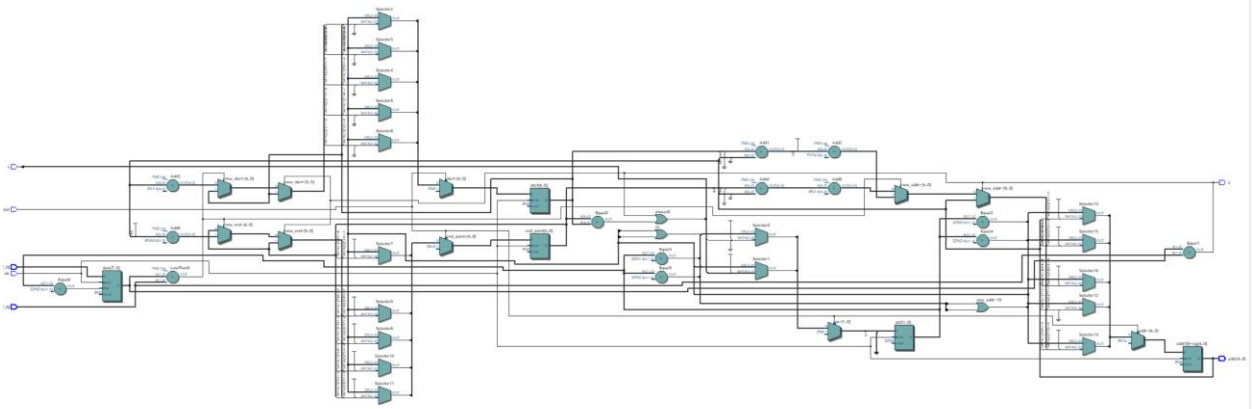
# Task2

## Section 1: Procedure

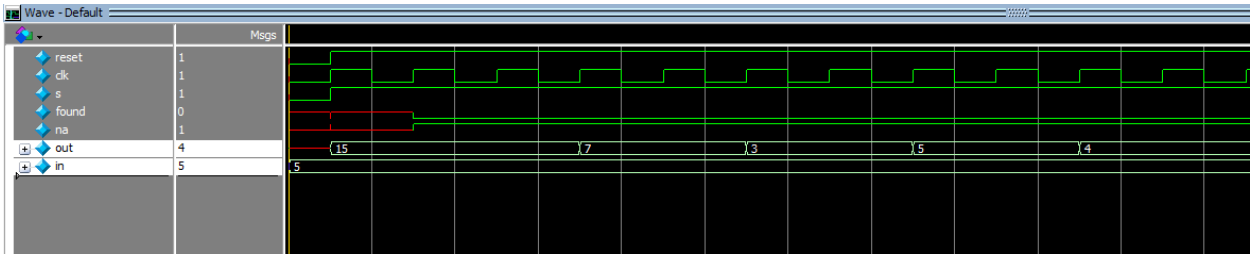
Following the ASMD chart in task1, I draw the ASMD chart for task2.


reset
S1
start ← 0
end_point ←
2^array_size-1
load
0
S
0
S2
addr = (end_
point - start)/2
+start
ram[addr]
== data
1
0
start >=
end_point
1
0
Ram[addr] > A
0
1
S3
start_point
← addr+1
S3
end_point ←
addr -1
S
S4
found
1

I designed 4 states, S1: initial state, only load registers, S2: read data from ram and compare with input data, S3: update search address, S4: finish searching. In search state, the system starts with a search boundary from 0 to 31 and compares the current value from the RAM to the value to find. If they are equal, the search finishes. If the current value is larger, the upper bound for the search becomes 1 less than the current address, else the lower bound becomes 1 greater than the current address. Then the system repeats the cycle until it finds the value or the lower bound is larger than or equal to the upper bound, at which the search finishes.

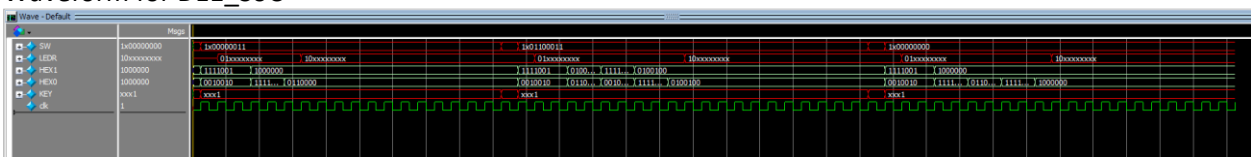


## Section 2: Results




After a few steps, the system find the target data.

Waveform for DE1_SoC



## Section 3: Appendix

| Addr | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 | ASCII |
|---|---|---|---|---|---|---|---|---|---|
| 00 | 00000000 | 00000001 | 00000010 | 00000011 | 00000100 | 00001000 | 00001001 | 00001111 | ..... |
| 08 | 00010001 | 00111000 | 00111011 | 01000000 | 01000011 | 01000100 | 01000101 | 01000110 | .8;@CDEF |
| 10 | 01000111 | 01001000 | 01001010 | 01010001 | 01010010 | 01011000 | 01100011 | 01100110 | GHJQRXcf |
| 18 | 10000000 | 10010001 | 10011001 | 10101011 | 11001101 | 11111000 | 11111110 | 11111111 | ..... |

```
// Alan Li
// 02/25/2022
// EE 371
// Lab #4

// This module displays the value on the hex led display
module seg7(val, leds);
    input logic [3:0] val;
    output logic [6:0] leds;

    // control the led display with binary numbers
    always_comb begin case
    (val)
    //           Light: 6543210
        4'b0000: leds = 7'b1000000; // 0
        4'b0001: leds = 7'b1111001; // 1
        4'b0010: leds = 7'b0100100; // 2
        4'b0011: leds = 7'b0110000; // 3
        4'b0100: leds = 7'b0011001; // 4
        4'b0101: leds = 7'b0010010; // 5
        4'b0110: leds = 7'b0000010; // 6
        4'b0111: leds = 7'b1111000; // 7
        4'b1000: leds = 7'b0000000; // 8
        4'b1001: leds = 7'b0010000; // 9
        4'b1010: leds = 7'b0001000; // A
        4'b1011: leds = 7'b0000011; // b
        4'b1100: leds = 7'b1000110; // C
        4'b1101: leds = 7'b0100001; // d
        4'b1110: leds = 7'b0000110; // E
        4'b1111: leds = 7'b0001110; // F
        default: leds = 7'b1111111;   endcase
    end
endmodule

```

```
// Alan Li
// 02/25/2022
// EE 371
// Lab #4

// binary_search takes 8-bit in, 1-bit reset, clk, and s as input and return 5-bit out, 1-bit na and found as output signal.

module binary_search #(parameter data_width = 8, addr_width = 5) (reset, clk, s, in, found, na, out);
    input logic reset, clk, s;
    input logic [data_width - 1:0] in;
    output logic found, na;
    output logic [addr_width - 1:0] out;

    // addr is the current address that system is checking, q is the data read from ram, z is the signal for whether the data exist in ram
    logic [addr_width - 1:0] addr;
    logic [data_width - 1:0] q;
    logic z;

    // ram32*8 takes addr, clk, 0, 0 as input parameters address, clock, data, wren and q and returns q as output
    ram32x8 r (.address(addr), .clock(clk), .data(0), .wren(0), .q);

    // bsc take reset, s, clk, q, in as input paramter and return addr and z.
    binary_search_control bsc (.*);

    assign out = addr;
    // assign found signal, na as not found signal
    assign found = z;
    assign na = ~z;
endmodule

// tests the previous module
`timescale 1 ps / 1 ps
module binary_search_testbench ();
    logic reset, clk, s, found, na;
    logic [4:0] out;
    logic [7:0] in;

    binary_search bs (.*);

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

    initial begin
        reset <= 0; s <= 0; in <= 8'b00000100;          @(posedge clk);
        reset <= 1; s <= 1;                 repeat(20) @(posedge clk);

```

```
// Alan Li
// 02/25/2022
// EE 371
// Lab #4

// DE1_SoC takes 3-bit KEY and 10-bit SW as input and return 7-bit HEX and 10-bit LEDR as output.
// The system searches through an array to locate an 8-bit value A specified via switches SW7-0.
// SW[9] drive input start, KEY0 is the reset key
// This serves as top-level module for the bitCounter system
module DE1_SoC (SW, KEY, HEX1, HEX0, LEDR, CLOCK_50);
   input logic [9:0] SW;
   input logic [3:0] KEY;
   input logic CLOCK_50;
   output logic [9:0] LEDR;
   output logic [6:0] HEX1, HEX0;
   logic [4:0] out;
   logic found;

   // bs takes KEY[0], CLOCK_50, SW[9], SW[7:0] as input parameter reset, clk, s, in and return LEDR[9], [8] and out repectively
   binary_search bs (.reset(KEY[0]), .clk(CLOCK_50), .s(SW[9]), .in(SW[7:0]), .found(LEDR[9]), .na(LEDR[8]), .out);
   // s1 takes quotient of value as input and return HEX1 on display
   seg7 s1 (.val(out / 10), .leds(HEX1));
   // s2 takes remainder of value as input and return HEX0 on display
   seg7 s2 (.val(out % 10), .leds(HEX0));
endmodule

// Tests the previous module
`timescale 1 ps / 1 ps
module DE1_SoC_testbench ();
   logic [9:0] SW, LEDR;
   logic [6:0] HEX1, HEX0;
   logic [3:0] KEY;
   logic clk;

   DE1_SoC ds (.SW, .KEY, .HEX1, .HEX0, .LEDR, .CLOCK_50(clk));

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

   initial begin
      KEY[0] <= 0; SW[9] <= 0; SW[7:0] <= 8'b00000011; @(posedge clk);
      KEY[0] <= 1; SW[9] <= 1;                         @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
                                                       @(posedge clk);
      KEY[0] <= 0; SW[9] <= 0; SW[7:0] <= 8'b01100011; @(posedge clk);
      KEY[0] <= 1; SW[9] <= 1;                         @(posedge clk);
```

```
// Alan Li
// 02/25/2022
// EE 371
// Lab #4

// binary_search_control takes in reset, s, clk,8-bit q and in as input and return 5-bit address and z as output
module binary_search_control #(parameter data_width = 8, addr_width = 5) (reset, s, clk, q, in, addr, z);
   input logic s, clk, reset;
   input logic [data_width - 1:0] q, in;
   output logic [addr_width - 1:0] addr;
   output logic z;
   logic [addr_width - 1:0] start, end_point, new_start, new_end, new_addr;
   logic [data_width - 1:0] data;

   // S1: initial state, only load registers
   // S2: read data from ram and compare with input data
   // S3: update search address
   // S4: finish searching
   // control circuit
   enum {S1, S2, S3, S4} ps, ns;


   always_comb begin
      case (ps)
         S1: if (s) ns = S2;
             else ns = S1;
         S2: if (q == data || start == end_point) ns = S4;
             else ns = S3;
         S3: ns = S2;
         S4: if (s) ns = S4;
             else ns = S1;
      endcase
   end

   // implement the ASMD chart for binary searching
   always_comb begin
      z = (q == data);
      case (ps)
         S1: begin
               new_start = 0;
               new_end = 31;
               new_addr = 15; // the address that's in the middle of the array
             end
         S2: begin
               new_start = start;
               new_end = end_point;
               new_addr = addr;
             end
         S3: begin
                  if (z) begin
                     new_start = start;
                     new_end = end_point;
                     new_addr = addr;
                  end
                  else if (q > data) begin  //scan thru the smaller part
                     new_end = addr - 1;
                     new_start = start;
                     new_addr = (start + addr - 1) / 2;
                  end
                  else begin  //scan thru the larger part
                     new_end = end_point;
                     new_start = addr + 1;
                     new_addr = (end_point + addr + 1) / 2;
                  end
```

```
34
35      // implement the ASMD chart for binary searching
36      always_comb begin
37          z = (q == data);
38          case (ps)
39              S1: begin
40                    new_start = 0;
41                    new_end = 31;
42                    new_addr = 15; // the address that's in the middle of the array
43                  end
44              S2: begin
45                    new_start = start;
46                    new_end = end_point;
47                    new_addr = addr;
48                  end
49              S3: begin
50                        if (z) begin
51                           new_start = start;
52                           new_end = end_point;
53                           new_addr = addr;
54                        end
55                        else if (q > data) begin  //scan thru the smaller part
56                           new_end = addr - 1;
57                           new_start = start;
58                           new_addr = (start + addr - 1) / 2;
59                        end
60                        else begin  //scan thru the larger part
61                           new_end = end_point;
62                           new_start = addr + 1;
63                           new_addr = (end_point + addr + 1) / 2;
64                        end
65                    end
66              S4: begin  // stays at the current address
67                    new_start = start;
68                    new_end = end_point;
69                    new_addr = addr;
70                  end
71          endcase
72      end
73
74      // when reset, system go back to initial state, otherwise load the registers and scan the array for the target data
75      always_ff@(posedge clk) begin
76          if (~reset)
77              ps <= S1;
78          else
79              ps <= ns;
80      end
81
82      always_ff@(posedge clk) begin
83          if (~reset) begin
84              start <= 0;
85              end_point <= 31;
86              addr <= 15;
87          end
88          else begin
89              start <= new_start;
90              end_point <= new_end;
91              addr <= new_addr;
92          end
93          if (ps == S1)
94              data <= in;
95      end
96  endmodule
97
```