Kiana Peterson and Nattapon Oonlmaom
EE 371
February 14, 2023
Lab 4: Implementing Algorithms in Hardware Report

Procedure
The purpose and objective of this lab were to use algorithmic state machine charts (ASM) to implement algorithms as hardware circuits.

Task 1:
In Task 1, we implemented the bit-counting circuit using the ASMD chart shown in Figure 1 on the lab document that responds to the 8-bit input from SW7-0 with SW0 as a start signal and KEY0 as a reset. When the program is completed, LEDR9 should light up. Looking at the ASMD given, we redrawn our ASMD chart and state diagram, then implemented the bitcounter, datapath, and the shifter for the bitcounter module into SystemVerilog as shown in figure 1.1.

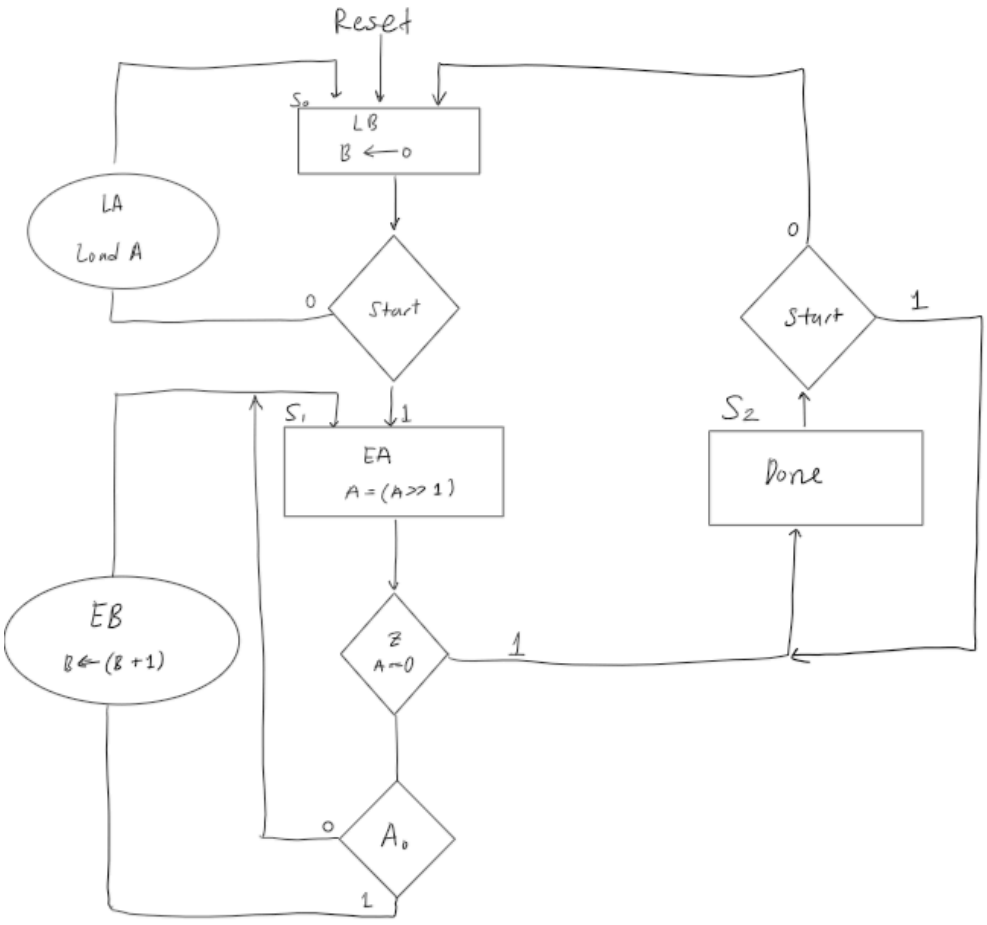

Figure 1.1 ASMD Chart of Task1

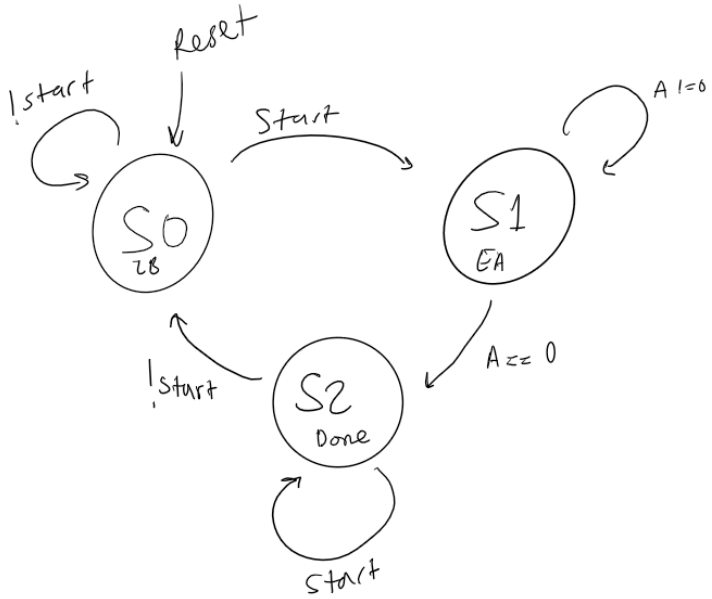

Figure 1.2 State Diagram for Task1

After completing the components of the ASMD chart and the state diagram, we developed a block diagram to represent the DE1_SoC out top-level module. We thought about how the component will work together and instantiated them according to figure 1.3.

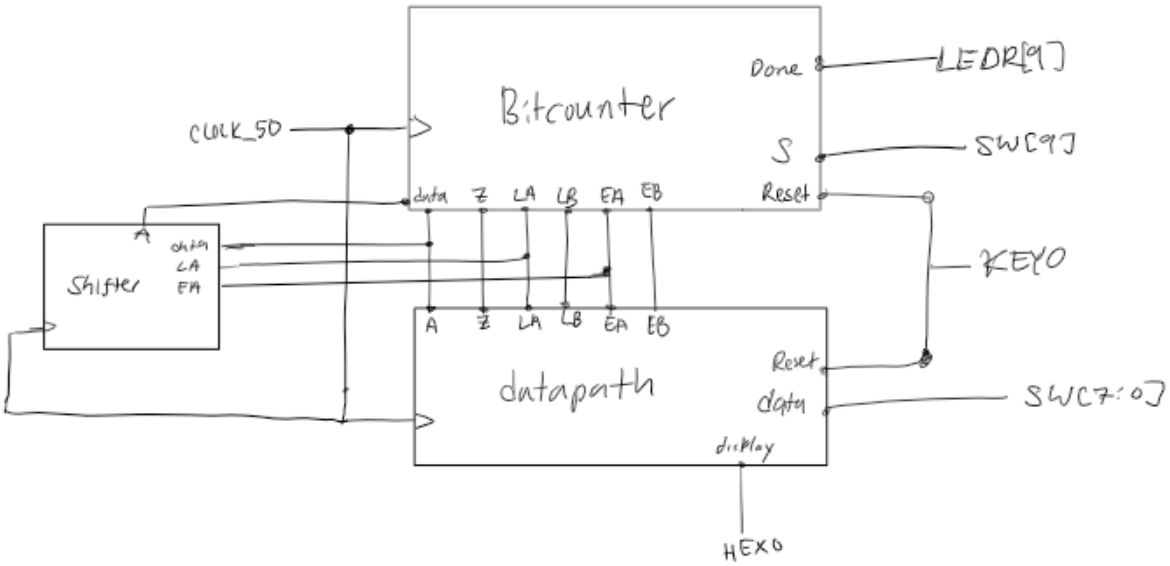

Figure 1.3 Block Diagram of Task1

Task 2:

In Task 2, we implemented a binary search algorithm that searches through an array to locate an 8-bit value A specified via switches SW7-0. When found, LEDR9 should light up and the address location should be shown on the HEX display. However, when not found, LEDR8 should light up. We began by thinking about what components do we need to accomplish this challenge and began by drawing the block diagram as shown in figure 2.1.

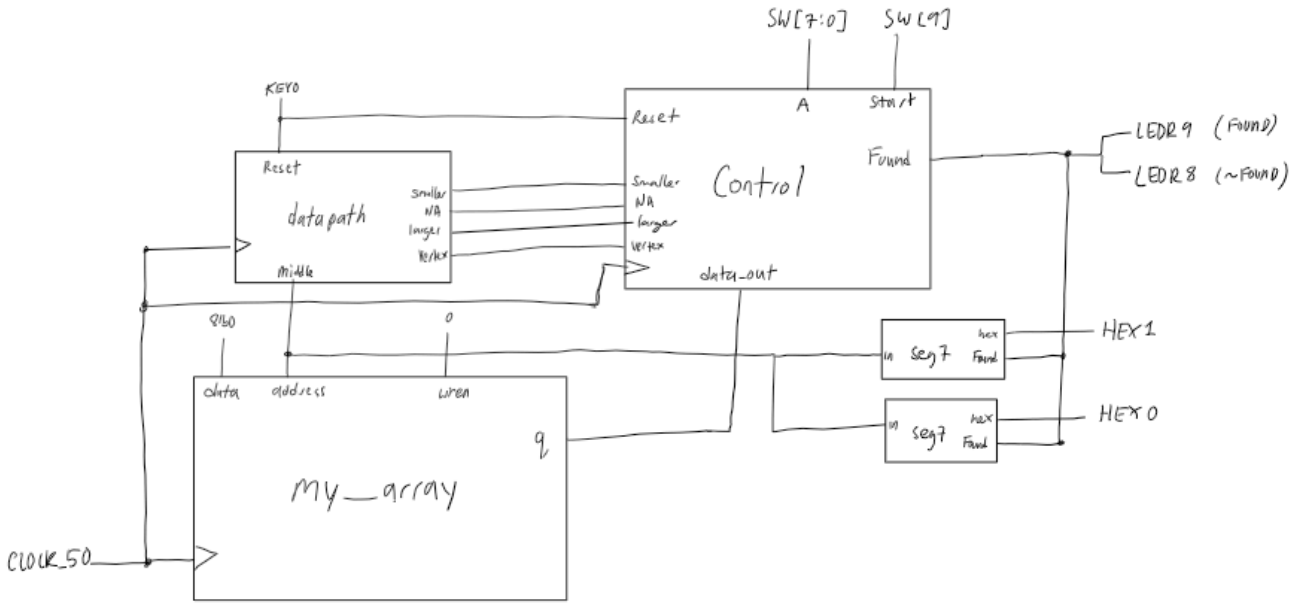

Figure 2.1 Block Diagram of Task2

After knowing what components will be needed and how they will be connected, we designed an ASMD to accomplish the behavior of the binary search algorithm as shown in figure 2.2. Then, we drew a state diagram for the FSM shown in figure 2.3.

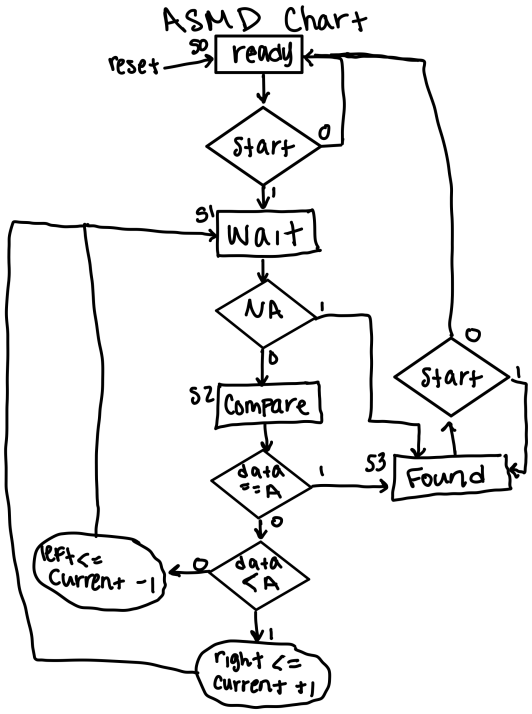

Figure 2.2 ASMD Chart of Task2

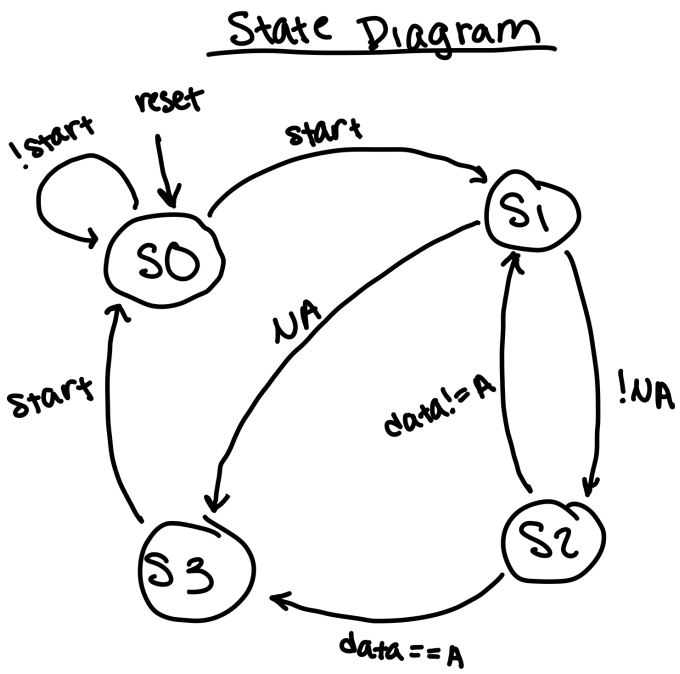

Figure 2.3 State Diagram for Task2

Results

Task 1:

After implementing the modules and instantiating them into the DE1_SoC.sv, we ran the testbench to test the behavior of the DE1_SoC.sv to see if the modules are functioning correctly before running them on the FPGA.

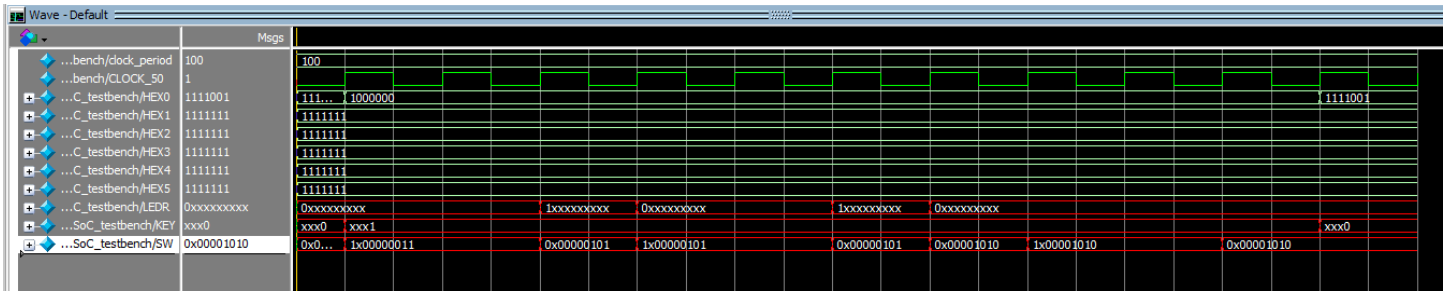

Figure 3.1 DE1_SoC.sv Waveform

The DE1_SoC is functioning correctly. The modules instantiated the values given correctly and are able to output correct values.

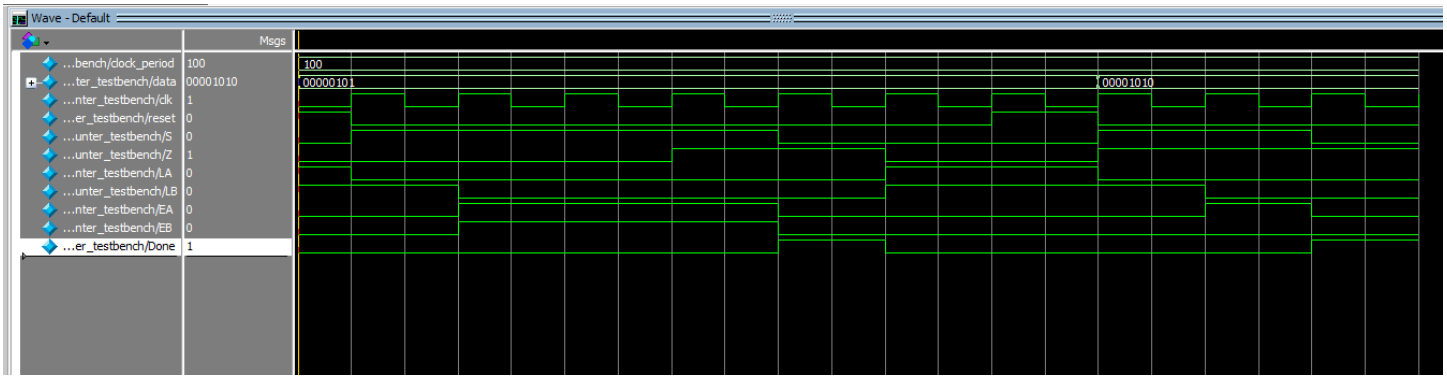

Figure 3.2 bitcounter.sv Waveform

The bitcounter.sv module is functioning correctly. With the possible inputs given, the module could output the values correctly that correspond to the numbers of 1s that will be displayed after instantiations under the DE1_SoC.sv module as the data shifted.

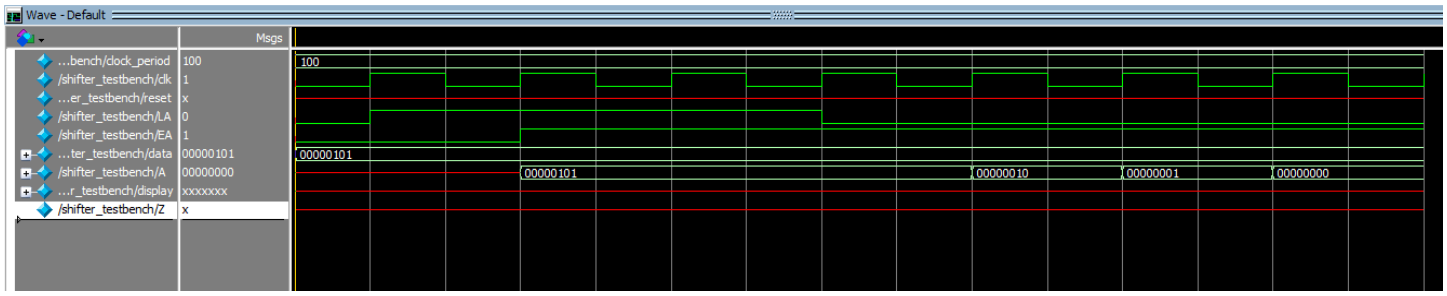

Figure 3.3 shifter.sv Waveform

The shifter.sv module is correct. The module shifts the value given on the testbench correctly as suggested by the ASMD chart. The behavior should be correct.

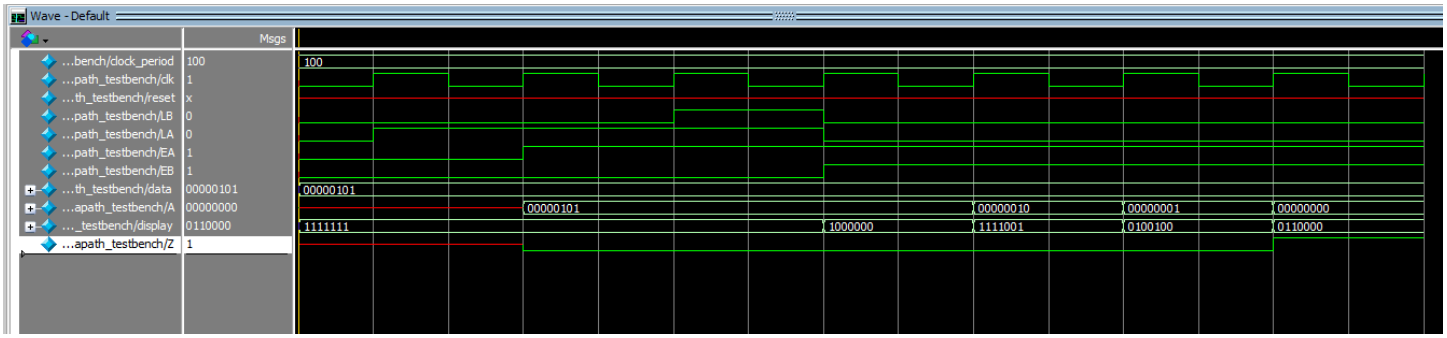

Figure 3.4 datapath.sv Waveform

The datapath.sv module works correctly. The datapath displayed a correct value on the HEX displayed, corresponding to the bit counter given on the testbench.

Task 2:

A similar process to Task 1, after implementing the modules and instantiating them into the DE1_SoC.sv, we ran the testbench to test the behavior of the DE1_SoC.sv to see if the modules are functioning correctly before running them on FPGA.

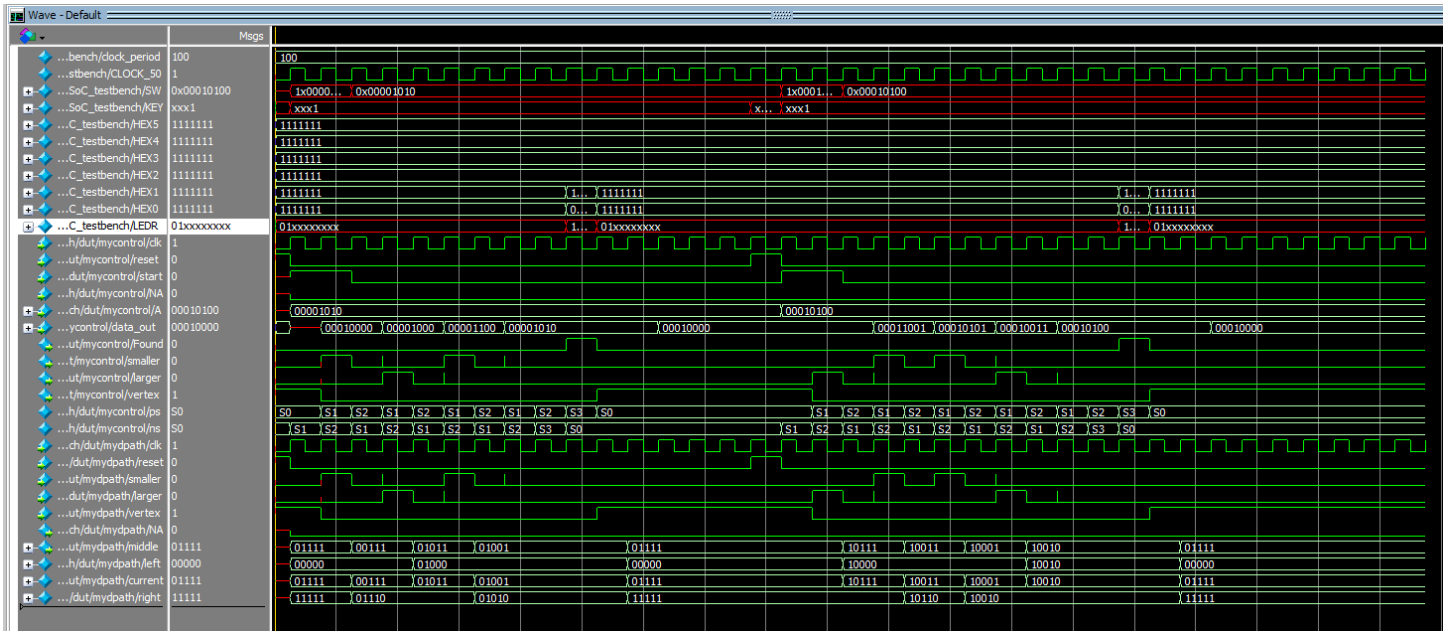

Figure 4.1 DE1_SoC.sv Waveform

The DE1_SoC is functioning correctly. The modules instantiated the values given correctly and are able to output correct values.

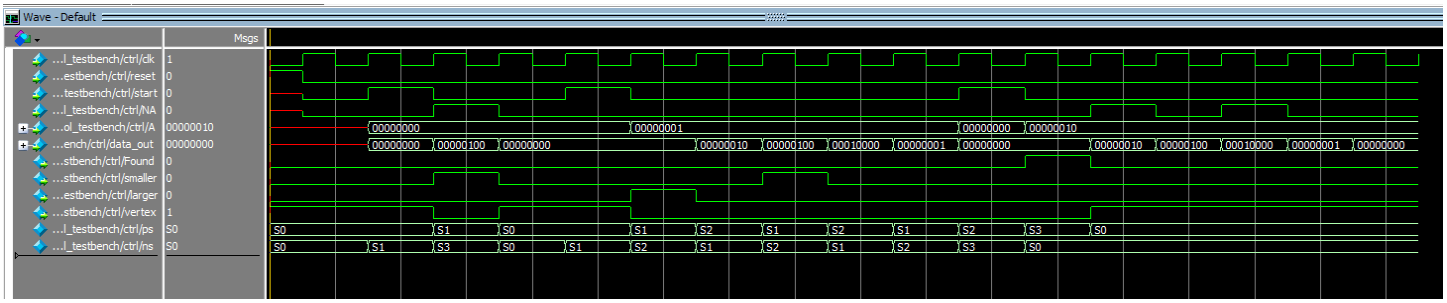

Figure 4.2 control.sv Waveform

The control.sv module is functioning correctly. With the possible inputs given, the module could output the values correctly that correspond to the numbers of 1s that will be displayed after instantiations under the DE1_SoC.sv module as the data shifted.

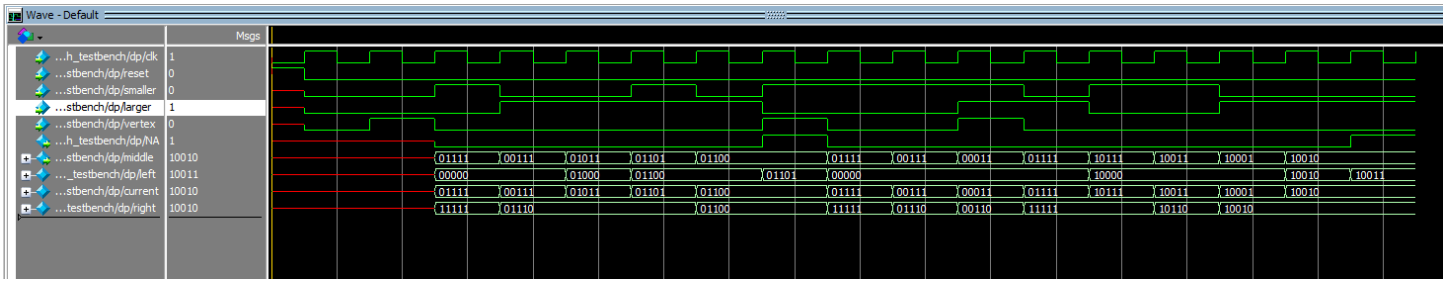

Figure 4.3 datapath.sv Waveform

The datapath.sv module is correct. The module outputs the address accordingly to the value given on the testbench correctly.

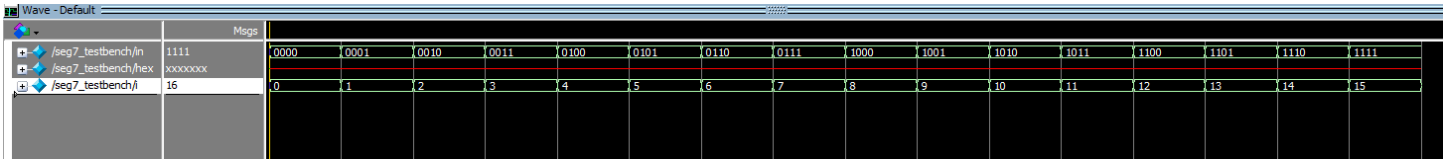

Figure 4.4 seg7.sv Waveform

The seg7.sv module works correctly. The datapath displayed a correct value on the HEX displayed, corresponding to the bit counter given on the testbench.

Final Product

Overall, the product of this project was an implementation of ASMD (Algorithmic State Machine and Datapath). In Task1, the bit-counting circuit responds to the 8-bit input from SW7-SW0 with SW0 as a start signal and KEY0 as a reset. When SW7-SW0 are inputted, the number of 1s corresponding to the bits are shown on the HEX display and when the program is completed, LEDR9 lights up. When reset is pressed, 0 is displayed. In Task2, a binary search algorithm searches through an array to locate an 8-bit value, A, specified by switches SW7-SW0. When found, LEDR9 lights up and the address stored in the 32x8 memory (specified by the .mif file) is shown on the HEX display. However, when not found, LEDR8 lights up.

Demonstrate Link:

Task1: https://drive.google.com/file/d/1v7BqswqTxlyQIJG5GQDFN6z9O8WVYwNt/view

Task2: https://drive.google.com/file/d/1zmyUgDLYj31gEuTOapckLIgL-BA5SoCS/view?usp=sharing

Appendix: SystemVerilog Code

```
// Nattapon Oonlamom and Kiana Peterson
// 02/13/23
// EE 371
// Lab 4: Implementing Algorithms in Hardware (Task1)

/* Overview: This is the top-level module that controls input logics from the FPGA
 * to display the behavior of bitcounter and datapath module onto the HEX display
 * and LEDR accordingly to the inputs given form SW7-0.
 *
 * Overall inputs and outputs listed below:
 * Inputs:
 *   CLOCK_50 (50 MHz Clock)
 *   KEY      (4 bits, active high)
 *   SW       (10 bit, active low)
 * Outputs:
 *   HEXs (six 7-bit, active low)
 *   LEDR (10 bits)
 */
module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   input logic CLOCK_50; // 50 MHz Clock
   output logic [9:0] LEDR;
   input logic [3:0] KEY;
   input logic [9:0] SW;

   // logic to be used
   logic LATemp, LBTemp, EATemp, EBTemp;
   logic ZTemp;
   logic [7:0] dataTemp;

   // instantiate the bitcounter module to outpute LA, LB, EA, EB, and Done
   bitcounter count (.clk(CLOCK_50), .reset(~KEY[0]), .S(SW[9]), .Z(ZTemp), .data(dataTemp),
                     .LA(LATemp), .LB(LBTemp), .EA(EATemp), .EB(EBTemp), .Done(LEDR[9]));

   // instantiate the datapath module with inputs from bitcounter and outputs Z, display, and A
   datapath data (.clk(CLOCK_50), .reset(~KEY[0]), .LB(LBTemp), .LA(LATemp), .EA(EATemp),
                  .EB(EBTemp), .data(SW[7:0]), .display(HEX0), .A(dataTemp), .Z(ZTemp));

   assign HEX1 = 7'b1111111;
   assign HEX2 = 7'b1111111;
   assign HEX3 = 7'b1111111;
   assign HEX4 = 7'b1111111;
   assign HEX5 = 7'b1111111;

endmodule
```

Figure 5 DE1_SoC.sv for Task1

```
// Nattapon Oonlamom and Kiana Peterson
// 02/13/23
// EE 371
// Lab 4: Implementing Algorithms in Hardware (Task1)

/* Overview: This module is an FSM that outputs different states
 * of loading conditions accordinly to the data given.
 *
 * Overall inputs and outputs listed below:
 * Inputs:
 *   data       (8 bits)
 *   clk, reset (1 bit)
 *   S, Z       (1 bit)
 * Outputs:
 *   LA, LB, EA, EB, done (1 bit)
 */
module bitcounter (clk, reset, S, Z, data, LA, LB, EA, EB, Done);
   input  logic [7:0] data;
   input  logic       clk, reset;
   input  logic       S, Z;
   output logic       LA, LB, EA, EB, Done;

   enum {S0, S1, S2} ps, ns;  // present and next state

   // combinational logic
   always_comb begin
      case(ps)
         S0: begin
               if (S)
                  ns = S1;
               else
                  ns = S0;
            end
         S1: begin
               if (Z == 0)
                  ns = S1;
               else
                  ns = S2;
            end
         S2: begin
               if (S)
                  ns = S2;
               else
                  ns = S0;
            end
      endcase
   end

   // sequential logic DFFs
   always_ff @(posedge clk) begin
      if (reset)
         ps <= S0;
      else
         ps <= ns;
   end

   // assigning outputs
   assign LA   = ((ps == S0) && (S == 0));
   assign LB   =  (ps == S0);
   assign EA   =  (ps == S1);
   assign EB   = ((ps == S1) && (data[0] == 1));
   assign Done =  (ps == S2);
endmodule
```

Figure 6 bitcounter.sv

```
// Nattapon Oonlamom and Kiana Peterson
// 02/13/23
// EE 371
// Lab 4: Implementing Algorithms in Hardware (Task1)

/* Overview: This module increments and loads the data
 * accordingly to the loading states given and output
 * a HEX display along with the shifted data information.
 *
 * Overall inputs and outputs listed below:
 * Inputs:
 *   clk, reset     (1 bit)
 *   LB, LA, EA, EB (1 bit)
 *   data           (8 bits)
 * Outputs:
 *   display (7 bits, active low)
 *   A (8 bits)
 *   Z (1 bit)
 */
module datapath (clk, reset, LB, LA, EA, EB, data, display, A, Z);
   input  logic       clk, reset;
   input  logic       LB, LA, EA, EB;
   input  logic [7:0] data;
   output logic [7:0] A;
   output logic [6:0] display;
   output logic       Z;

   // logic for HEX value
   logic [3:0] B;
   logic [6:0] zero, one, two, three, four, five, six, seven, eight;

   // sequential logic
   always_ff @(posedge clk) begin
      // increment load
      if (reset) begin
          B <= 0;
      end
      else if (LB) begin
          B <= 4'b0000;
      end
      else if (EB) begin
          B <= (B + 1);
      end
   end

   // instantiate the shifter module to shift the data to the next
   shifter shift (.clk, .reset, .data, .LA, .EA, .A);

   // assigning output Z
   assign Z = (~|A);

   // logics and booleans for always_comb
   //  active low
   assign zero  = 7'b1000000;
   assign one   = 7'b1111001;
   assign two   = 7'b0100100;
   assign three = 7'b0110000;
   assign four  = 7'b0011001;
   assign five  = 7'b0010010;
   assign six   = 7'b0000010;
   assign seven = 7'b1111000;
   assign eight = 7'b0000000;

   // combinational logic for display
   always_comb begin
      case (B)
         4'b0000: begin
                  display = zero;
                end
         4'b0001: begin
                  display = one;
                end
         4'b0010: begin
                  display = two;
                end
         4'b0011: begin
                  display = three;
                end
         4'b0100: begin
                  display = four;
                end
         4'b0101: begin
                  display = five;
                end
         4'b0110: begin
                  display = six;
                end
         4'b0111: begin
                  display = seven;
                end
         4'b1000: begin
                  display = eight;
                end
         default: begin
                  display = 7'b1111111;
                end
      endcase
   end
endmodule
```

Figure 7 datapath.sv for Task1

```
1  // Nattapon Oonlamom and Kiana Peterson
2  // 02/13/23
3  // EE 371
4  // Lab 4: Implementing Algorithms in Hardware (Task1)
5
6  /* Overview: This module shifts the data accordingly to
7   * the given load conditions and output new appointed data.
8   *
9   * Overall inputs and outputs listed below:
10  * Inputs:
11  *   clk, reset     (1 bit)
12  *   LA, EA         (1 bit)
13  *   data           (8 bits)
14  * Outputs:
15  *   A (8 bits)
16  */
17 module shifter (clk, reset, data, LA, EA, A);
18    input  logic       clk, reset;
19    input  logic       LA, EA;
20    input  logic [7:0] data;
21    output logic [7:0] A;
22
23    // sequential logic
24    always_ff @(posedge clk) begin
25       if (reset) begin
26           A <= 8'b00000000;
27       end
28       else if (LA) begin
29           A <= data;
30       end
31       else if (EA) begin
32           A <= (A >> 1);
33       end
34    end
35
36 endmodule
```

Figure 8 shifter.sv

```
// Nattapon Oonlamom and Kiana Peterson
// 02/14/2023
// Lab 4: Implementing Algorithms in Hardware (Task2)

// This is the top module of task two, connecting all the modules
// It takes in an 8-bit number defined by switches 7-0, then searches for the
// location of the input in the given array. This module then displays the location on the HEX if found
// and lights up LEDR[9]. If not found in the array, LEDR[8] lights up

// Overall inputs and outputs to the DE1_SoC module listed below:
// Inputs: 1 bit CLOCK_50, 10-bit SW, 4-bit KEY
// Outputs: 6 7-bit HEXs, 10-bit LEDR

module DE1_SoC(CLOCK_50, SW, KEY, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0, LEDR);

   input logic CLOCK_50;
   input logic [9:0] SW;
   input logic [3:0] KEY;
   output logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   output logic [9:0] LEDR;

   logic reset;
   logic smaller, larger;|
   logic FoundTemp;
   logic [4:0] L; // The address corresponding to the location of A
   logic [7:0] dataTemp; // The number stored at a specified address

   assign LEDR[9] = FoundTemp; // Lights up if found
   assign LEDR[8] = ~FoundTemp; // Lights up if not in the array
   assign reset   = ~KEY[0]; // Active low

   // Makes the other HEXs blank
   assign HEX5 = 7'b1111111;
   assign HEX4 = 7'b1111111;
   assign HEX3 = 7'b1111111;
   assign HEX2 = 7'b1111111;

   // Controls whether to look to the left or the right
   control mycontrol (.clk(CLOCK_50), .reset(reset), .start(SW[9]), .A(SW[7:0]),
                      .data_out(dataTemp), .Found(FoundTemp), .smaller(smaller),
                      .larger(larger), .NA(NA), .vertex(vertex));

   // Determines the new middle address to be looking at
   datapath mydpath (.clk(CLOCK_50), .reset(reset), .smaller(smaller),
                     .larger(larger), .middle(L), .NA(NA), .vertex(vertex));

   // The 32x8 memory holding an array of unique sorted values
   my_array ram (.clock(CLOCK_50), .data(8'b00000000), .address(L), .wren(1'b0), .q(dataTemp));

   // Displays the address of A if found
   seg7 Ldisplay (.in({3'b000, L[4]}), .Found(FoundTemp), .hex(HEX1));
   seg7 Rdisplay (.in(L[3:0]), .Found(FoundTemp), .hex(HEX0));

endmodule

`timescale 1 ps / 1 ps
module DE1_SoC_testbench();
   logic CLOCK_50;
   logic [9:0] SW;
   logic [3:0] KEY;
   logic [6:0] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
   logic [9:0] LEDR;

   DE1_SoC dut (.*);

   //clock setup
   parameter clock_period = 100;

   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;

   end //initial

   initial begin
         // @(posedge clk) represents one clock cycle, and S, Z, and A are
         // the inputs of that clock cycle. If no inputs are declared before @(posedge clk),
         // the inputs remain the same from last clock cycle
      KEY[0]<=0;                                   @(posedge CLOCK_50);
      KEY[0]<=1; SW[9]<=1; SW[7:0]<= 8'd10;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd10;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd10;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd10;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd10;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd10;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd10;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
      KEY[0]<=0;                                   @(posedge CLOCK_50);
      KEY[0]<=1; SW[9]<=1; SW[7:0]<= 8'd20;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd20;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd20;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd20;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd20;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd20;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd20;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd20;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd20;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                 SW[9]<=0; SW[7:0]<= 8'd20;        @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);
                                                   @(posedge CLOCK_50);

      $stop;
   end
endmodule
```

Figure 9 DE1_SoC.sv for Task2

```
// Nattapon Oonlamom and Kiana Peterson
// 02/14/2023
// Lab 4: Implementing Algorithms in Hardware (Task2)

// This is the datapath module that handles finding the new middle value, whether to moce left or right

// Overall inputs and outputs to the control module listed below:
// Inputs: 1-bit clk, reset, smaller, larger, vertex
// Outputs: 5-bit middle, 1-bit NA

module datapath (clk, reset, smaller, larger, middle, NA, vertex);
   input logic clk, reset, smaller, larger, vertex;
   output logic [4:0] middle; // Address of the data to look at
   output logic NA; // Whether an unexpected value

   logic [4:0] left, current, right;

   assign current = (left + right) / 2; // The current address being looked at (the middle of section)

   always_ff @(posedge clk) begin
      // If at the first middle (beginning of the search)
      if (vertex) begin
         left  <= 5'b00000;
         right <= 5'b11111;
      end

      // If A was smaller than the data at current address, move the right pointer to 1 less than
      // the current middle
      else if (smaller) begin
            right <= current - 1'b1;
      end

      // If A was larger than the data at current address, move the left pointer to 1 more than
      // the current middle
      else if (larger) begin
            left  <= current + 1'b1;
      end
   end

   assign middle = current; // updates the middle pointer to current
   assign NA = (right < left); // NA if the right value is less than the left (improper values in a
array)

endmodule

module datapath_testbench();

   logic clk, reset, smaller, larger, vertex;
   logic NA;
   logic [4:0] middle;

   datapath dp (.*);

   //clock setup
   parameter clock_period = 100;

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

   end

   //initial
   initial begin
                  
      reset<=1;                                                                  @(posedge clk);
      reset<=0;   vertex <= 0; larger <=0; smaller <=0;                          @(posedge clk);
                  vertex <= 1; larger <=0; smaller <=0;                          @(posedge clk);
                  vertex <= 0; larger <=0; smaller <=1;                          @(posedge clk);
                  vertex <= 0; larger <=1; smaller <=0;                          @(posedge clk);
                  vertex <= 0; larger <=1; smaller <=1;                          @(posedge clk);
                  vertex <= 0; larger <=1; smaller <=1;                          @(posedge clk);
                  vertex <= 0; larger <=1; smaller <=0;                          @(posedge clk);
                  vertex <= 1; larger <=0; smaller <=1;                          @(posedge clk);
                  vertex <= 0; larger <=0; smaller <=1;                          @(posedge clk);
                  vertex <= 0; larger <=0; smaller <=1;                          @(posedge clk);
                  vertex <= 1; larger <=1; smaller <=1;                          @(posedge clk);
                  vertex <= 0; larger <=1; smaller <=0;                          @(posedge clk);
                  vertex <= 0; larger <=0; smaller <=1;                          @(posedge clk);
                  vertex <= 0; larger <=0; smaller <=1;                          @(posedge clk);
                  vertex <= 0; larger <=1; smaller <=0;                          @(posedge clk);
                  vertex <= 0; larger <=1; smaller <=0;                          @(posedge clk);
                                                                                 @(posedge clk);
      $stop; //end simulation
   end
endmodule

```

Figure 10 datapath.sv for Task2

```
1   // Nattapon Oonlamom and Kiana Peterson
2   // 02/14/2023
3   // Lab 4: Implementing Algorithms in Hardware (Task2)
4
5   // This is the control module that determines the states. It controls whether
6   // Data is still being compared or is finished.
7
8   // Overall inputs and outputs to the control module listed below:
9   // Inputs: 1-bit clk, 1-bit reset, 1-bit start, 1-bit NA, 5-bit A, 5-bit data_out
10  // Outputs: 1-bit Found, 1-bit smaller, 1-bit larger, 1-bit vertex
11
12  module control(clk, reset, start, A, data_out, Found, smaller, larger, NA, vertex);
13
14     input logic clk, reset;
15     input logic start, NA;
16     input logic [7:0] A, data_out;
17     output logic Found, smaller, larger, vertex;
18
19
20     enum {S0, S1, S2, S3} ps, ns; // present and next state
21
22     // combinational logic
23     always_comb begin
24        case(ps)
25           S0: begin // Begin state
26                 if (start)
27                    ns = S1;
28                 else
29                    ns = S0;
30              end
31           S1: begin // Wait state because of delay from memory
32                  if (NA)
33                     ns = S3;
34                  else
35                     ns = S2;
36              end
37           S2: begin // Compares the middle value to the data at the specified address
38                 if (A == data_out)
39                    ns = S3;
40                 else
41                    ns = S1;
42              end
43           S3: begin // If found
44                 if (start)
45                    ns = S3;
46                 else
47                    ns = S0;
48              end
49        endcase
50     end
51
52     assign Found   = (ps == S3); // If found the location
53     assign smaller = (ps == S1) && (A < data_out) && (A != data_out); // If need to search lower half
54     assign larger  = (ps == S1) && (A > data_out) && (A != data_out); // If need to search the
   larger half
55     assign vertex  = (ps == S0); // If at the original address
56
57     always_ff @(posedge clk) begin
58        if (reset || NA)
59           ps <= S0;
60        else
61           ps <= ns;
62     end
63
```

```
63
64  endmodule
65
66  module control_testbench();
67     logic clk, reset;
68     logic start, NA;
69     logic [7:0] A, data_out;
70     logic Found, smaller, larger, vertex;
71
72     control ctrl (.*);
73
74     //clock setup
75     parameter clock_period = 100;
76
77     initial begin
78        clk <= 0;
79        forever #(clock_period /2) clk <= ~clk;
80
81     end //initial
82
83     initial begin
84
85        reset<=1;                                                                    @(posedge clk);
86        reset<=0; start<=0; NA<=0;                                                   @(posedge clk);
87                                A <= 8'b00000000; data_out<=8'b00000000;
88                  start<=1; NA<=0;                                                   @(posedge clk);
89                                A <= 8'b00000000; data_out<=8'b00000100;
90                  start<=0; NA<=1;                                                   @(posedge clk);
91                                A <= 8'b00000000; data_out<=8'b00000000;
92                  start<=0; NA<=0;                                                   @(posedge clk);
93                                A <= 8'b00000000; data_out<=8'b00000000;
94                  start<=1; NA<=0;                                                   @(posedge clk);
95                                A <= 8'b00000001; data_out<=8'b00000000;
96                  start<=0; NA<=0;                                                   @(posedge clk);
97                                A <= 8'b00000001; data_out<=8'b00000010;
98                  start<=0; NA<=0;                                                   @(posedge clk);
99                                A <= 8'b00000001; data_out<=8'b00000100;
100                 start<=0; NA<=0;                                                   @(posedge clk);
101                               A <= 8'b00000001; data_out<=8'b00010000;
102                 start<=0; NA<=0;                                                   @(posedge clk);
103                               A <= 8'b00000001; data_out<=8'b00000001;
104                 start<=0; NA<=0;                                                   @(posedge clk);
105                               A <= 8'b00000000; data_out<=8'b00000000;
106                 start<=1; NA<=0;                                                   @(posedge clk);
107                               A <= 8'b00000010; data_out<=8'b00000000;
108                 start<=0; NA<=0;                                                   @(posedge clk);
109                               A <= 8'b00000010; data_out<=8'b00000010;
110                 start<=0; NA<=1;                                                   @(posedge clk);
111                               A <= 8'b00000010; data_out<=8'b00000100;
112                 start<=0; NA<=0;                                                   @(posedge clk);
113                               A <= 8'b00000010; data_out<=8'b00010000;
114                 start<=0; NA<=1;                                                   @(posedge clk);
115                               A <= 8'b00000010; data_out<=8'b00000001;
116                 start<=0; NA<=0;                                                   @(posedge clk);
117                               A <= 8'b00000010; data_out<=8'b00000000;
118                                                                                    @(posedge clk);
119        $stop; //end simulation
120
121     end //initial
122
123 endmodule
```

Figure 11 control.sv

```
// Nattapon Oonlamom and Kiana Peterson
// 02/14/2023
// Lab 4: Implementing Algorithms in Hardware (Task2)

// This module shows the logic for 7-segment active low display
// Takes in inputs, in and Found, to display numbers corresponding to the input
// depending on whether the address was found

// Overall inputs and outputs to the seg7 module listed below:
// Inputs: 4-bit in, 1-bit Found
// outputs: 7-bit HEX

module seg7 (in, Found, hex);
   input logic  [3:0] in;
   input logic Found;
   output logic [6:0] hex;


   always_comb begin
      // If not found, keep hex blank
      if (~Found) begin
         hex = 7'b1111111;
      end
      |
      // If the address is found, use the corresponding value to display
      else begin
         case(in)
            4'b0000: hex = 7'b1000000; // 0
            4'b0001: hex = 7'b1111001; // 1
            4'b0010: hex = 7'b0100100; // 2
            4'b0011: hex = 7'b0110000; // 3
            4'b0100: hex = 7'b0011001; // 4
            4'b0101: hex = 7'b0010010; // 5
            4'b0110: hex = 7'b0000010; // 6
            4'b0111: hex = 7'b1111000; // 7
            4'b1000: hex = 7'b0000000; // 8
            4'b1001: hex = 7'b0010000; // 9
            4'b1010: hex = 7'b0001000; // A
            4'b1011: hex = 7'b0000011; // B, shows up as b
            4'b1100: hex = 7'b1000110; // C
            4'b1101: hex = 7'b0100001; // D, shows up as d
            4'b1110:    hex = 7'b0000110; // E
            4'b1111: hex = 7'b0001110; // F
            default:    hex = 7'b1111111; // blank
         endcase
      end
   end

endmodule

```

```
// Testbench for the seg7 module to test all the possible hexcome
// to see of the present state and the next state is set up correctly
module seg7_testbench();

   // Logic to stimulate
   logic [3:0] in;
   logic Found;
   logic [6:0] hex;

   // Instantiates seg7
   seg7 dut (in, hex);

   integer i;
   initial begin
      for (i = 0; i < 2**4; i++) begin
         in = i; #10;
      end
   end // Initial
endmodule
```

Figure 12 seg7.sv

```
// megafunction wizard: %RAM: 1-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: my_array.v
// Megafunction Name(s):
//          altsyncram
//
// Simulation Library Files(s):
//          altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 17.0.0 Build 595 04/25/2017 SJ Lite Edition
// ************************************************************


//Copyright (C) 2017  Intel Corporation. All rights reserved.
//Your use of Intel Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Intel Program License
//Subscription Agreement, the Intel Quartus Prime License Agreement,
//the Intel MegaCore Function License Agreement, or other
//applicable license agreement, including, without limitation,
//that your use is for the sole purpose of programming logic
//devices manufactured by Intel and sold by Intel or its
//authorized distributors.  Please refer to the applicable
//agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module my_array (
    address,
    clock,
    data,
    wren,
    q);

    input [4:0]  address;
    input   clock;
    input [7:0]  data;
    input   wren;
    output   [7:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1    clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [7:0] sub_wire0;
    wire [7:0] q = sub_wire0[7:0];

    altsyncram  altsyncram_component (
              .address_a (address),
              .clock0 (clock),
              .data_a (data),
              .wren_a (wren),
              .q_a (sub_wire0),
              .aclr0 (1'b0),
              .aclr1 (1'b0),
              .address_b (1'b1),
              .addressstall_a (1'b0),
              .addressstall_b (1'b0),
              .byteena_a (1'b1),
              .byteena_b (1'b1),
              .clock1 (1'b1),
              .clocken0 (1'b1),
              .clocken1 (1'b1),
              .clocken2 (1'b1),
              .clocken3 (1'b1),
              .data_b (1'b1),
              .eccstatus (),
              .q_b (),
              .rden_a (1'b1),
              .rden_b (1'b1),
              .wren_b (1'b0));
    defparam
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file = "my_array.mif",
        altsyncram_component.intended_device_family = "Cyclone V",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 32,
        altsyncram_component.operation_mode = "SINGLE_PORT",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "UNREGISTERED",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.ram_block_type = "M10K",
        altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ",
        altsyncram_component.widthad_a = 5,
        altsyncram_component.width_a = 8,
        altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
// Retrieval info: PRIVATE: AclrByte NUMERIC "0"
// Retrieval info: PRIVATE: AclrData NUMERIC "0"
// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: Clken NUMERIC "0"
// Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
// Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
```

Figure 13 my_array.v

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	1	2	3	4	5	6	7	8	.....
8	9	10	11	12	13	14	15	16	.....
16	17	19	20	21	22	23	24	25	.....
24	26	27	28	29	30	31	32	33	..... !

Figure 14 my_array.mif