Brian Dallaire, Erik Michel
EE 371
5/16/2021

# Lab 4 Report

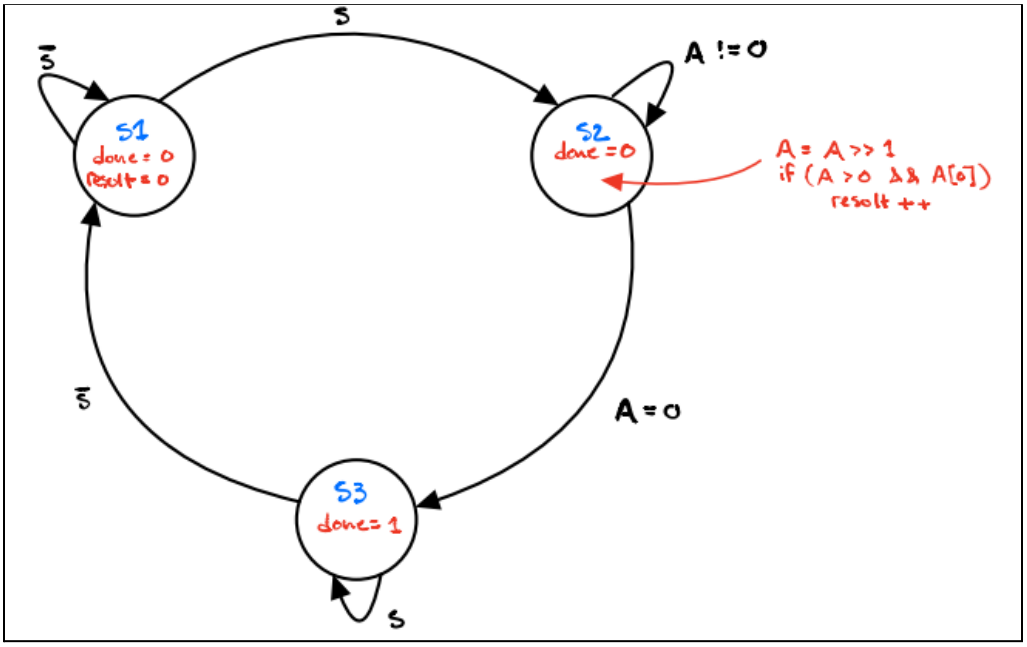
## Procedure

This lab consists of two main tasks. The first task is to design and implement an ON-bit counter. More specifically, to design a module that can count the number of ON bits (bits set to 1) in an n-bit value. The second task was to design and implement a binary search algorithm that will recursively search a 32x8 RAM unit and determine if the 8-bit input value was found or not found within the RAM unit. Overall, this lab was about comprehending, designing, and implementing ASMD charts and understanding the difference between control and datapath circuits and how to put them together in a complete system.

## Task 1

### Procedure

The first task was to design and implement an ON-bit counter for a given n-bit value, was performed by first breaking down the given ASMD chart for the module and converting into its FSM state diagram equivalent as seen in Figure 1:



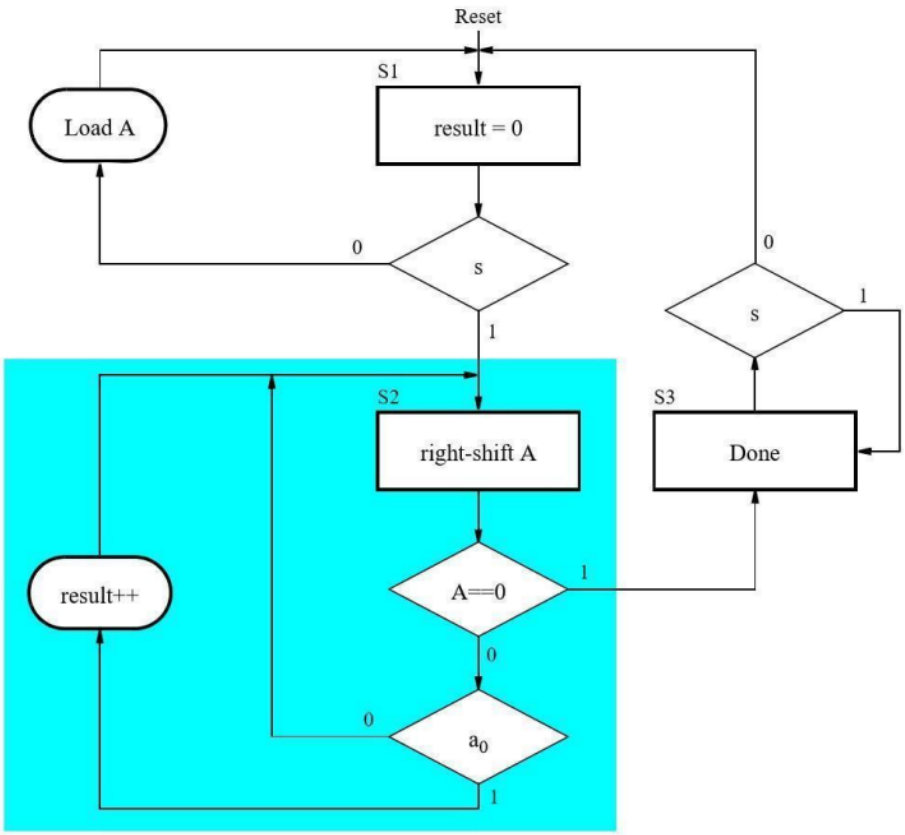



*Figure 1: FSM diagram and ASMD chart for task 1.*

Using the state diagram, the corresponding code was written in the posBitCounter.sv (appx. Task 1.2.) to match the behavior. The code written was a 3-state implementation consisting of an idle/s1 state in which the resulting count was initialized to zero. A counting/s2 state in which the module is actively counting ON bits in an n-bit value. And finally a completion/s3 state in which all ON bits have been counted in the n-bit value and a done signal is active.

A testbench module was created for posBitCounter.sv and the DE1_SoC top-level entity to ensure that the proper behavior is displayed by the posBitCounter main module. In DE1_SoC.sv (appx. Task 1.1.) an instantiation of the posBitCounter module was created and paired with a hexaDig0.sv (appx. Task 1.3.) instantiation to display resulting counts on HEX0.

**Results**

The result was a module, posBitCounter, that could successfully count the number of ON bits in an n-bit value. The testbench module for posBitCounter.sv confirmed that the module functioned properly and that the functionality of the module matched the given specification as can be seen in the ModelSim waveform in Figure 2:

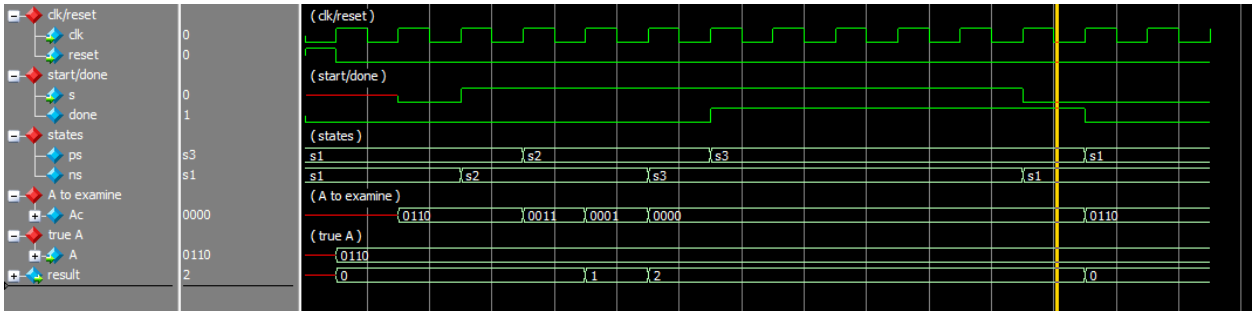



*Figure 2: Waveform produced by posBitCounter.sv testbench module.*

In addition, the DE1_SoC testbench module further confirmed that the module was working as expected and can be seen in Figure 3. The values being output from the result signal matched perfectly with a given number of bits in a value and the count was displayed accordingly on HEX0. In addition, upon completion of a count, LEDR[9] lit up as intended.

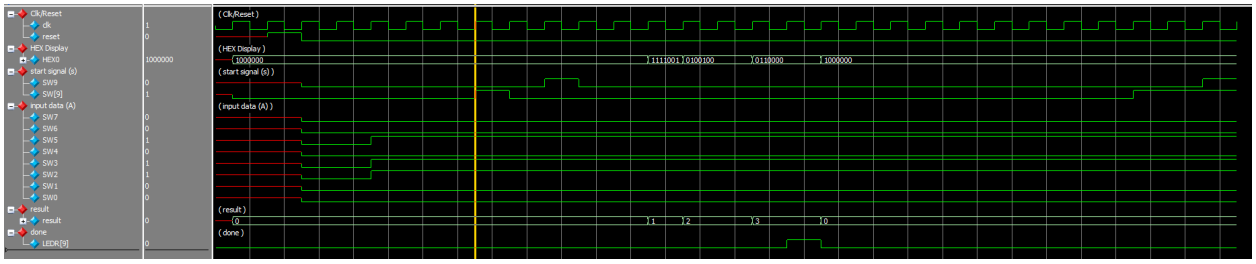



*Figure 3: Waveform produced by DE1_SoC.sv testbench module.*

## Block Diagram - Task 1

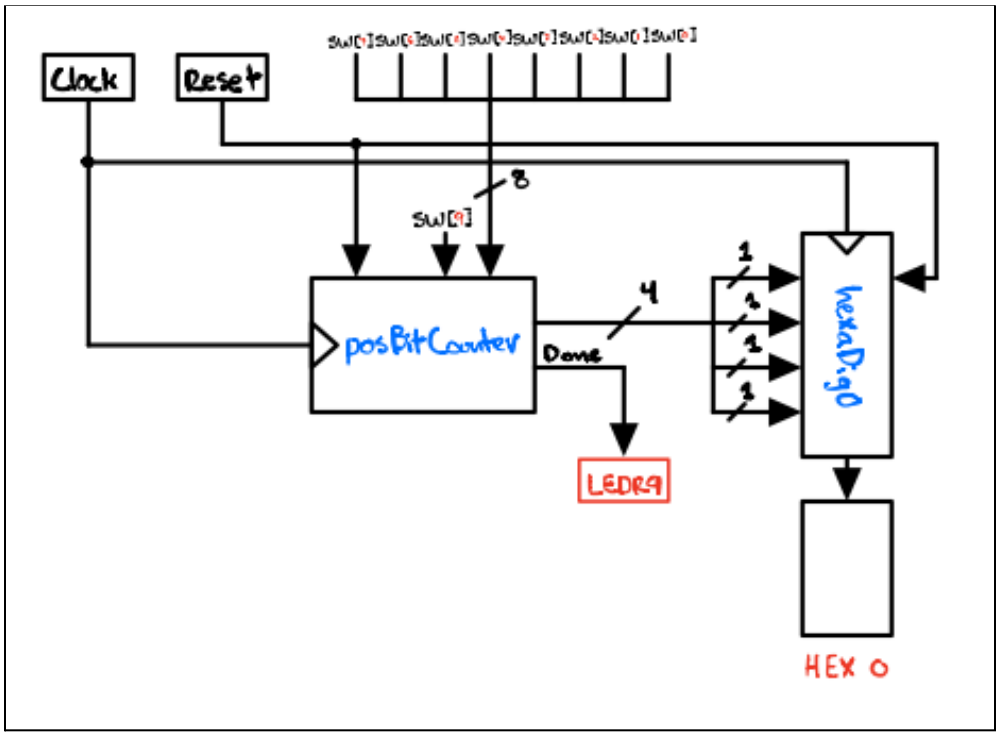



*Figure 4: Block diagram for the task 1 design implementation.*

## Design Decisions - Task 1

The posBitCounter was implemented in this way because it seemed to be the most efficient way to solve the task at hand. The design is very simple and compact. In addition, the way this module was designed made it very versatile when combined with a WIDTH parameter. This parameter allows the module to be used for a wide range of values barring memory constraints.

## Task 2

### Procedure

The second task was to design and implement a binary-search algorithm that searches through an array to locate a specific 8-bit value determined by the switches SW7-0. This algorithm will work on a sorted array, which means every component of the array is in increasing or decreasing order. Shown in Figure 5 is the Memory Initialization File (MIF) that represents the values my RAM will contain initially at each respective address.

| Addr | +0 | +1 | +2 | +3 | ASCII |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | .... |
| 4 | 5 | 6 | 7 | 8 | .... |
| 8 | 9 | 10 | 11 | 12 | .... |
| 12 | 13 | 14 | 15 | 16 | .... |
| 16 | 17 | 18 | 19 | 20 | .... |
| 20 | 21 | 22 | 23 | 24 | .... |
| 24 | 25 | 26 | 27 | 28 | .... |
| 28 | 29 | 30 | 31 | 32 | .... |

*Figure 5. MIF file used in task 2.*

As seen, the contents of the RAM file contain values in incrementing order. This allows for a search algorithm that starts with the middle address, which for my lab was the address “15”. In my binary search algorithm, I have three “pointers”. First, last, and middle. Initially, first is address “0”, and last is address “31”. As shown in Figure 6, I have middle assigned to be the first and last address combined divided by two.

```
logic [5:0] first, last, middle;
logic [4:0] middle2;
logic [7:0] q;

assign middle = (first + last) >> 1;
```

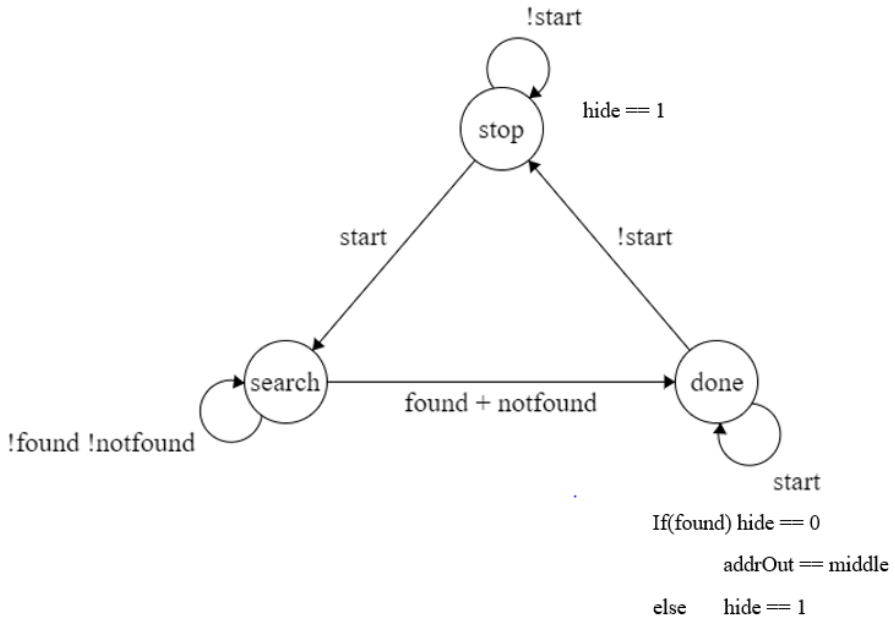
*Figure 6. Updating the middle pointer using the first and last pointers.*

What the algorithm does is compares the current value of the middle address to the input value, and sees if the input value is greater than or less than the current value from the RAM. If it is greater than the input, last will be assigned the value of middle, and first will remain the same. If it is less than the input, first will be assigned to middle, and last will remain the same. For example, if the value we wanted was 24, initially first and last are “0” and “31” and middle is “15”. Since the value at address “15” is 16 which

is less than the input, first will now be “15” and last will now be “31”, making the new middle point to the address “23”. Likewise, the first and last pointers will be moving every clock cycle to update the middle pointer. The middle pointer will then be inputted into the RAM as a read address to read the output value at that address. By updating the pointers with this mechanism, if the value is in the RAM it will eventually be detected. After the value has been detected, middle

Stopping the search when the input value is found from the RAM is straightforward, but finding a stopping place for the RAM when the value cannot be found took some thought. The stopping place originally felt as though it should be when middle was “31” or “0” since this would mean both first and last cannot move any further, but I realized the MIF file may not be in incrementing order by 1 integer. If the values increment by an integer value greater than 1, then the input value may fall between two consecutive values within the RAM. To stop the search in this case, I made another intermediate logic that equates to middle after one clock cycle. When this intermediate logic is equal to middle, it would mean middle has remained the same for more than one search. This also means first and last did not move, and the binary search algorithm ended.

Now that the basic structure of the algorithm has been decided, I had to implement it into SystemVerilog code. To start, I created a FSM and ASMD chart for the binary search system as shown in Figure 7.



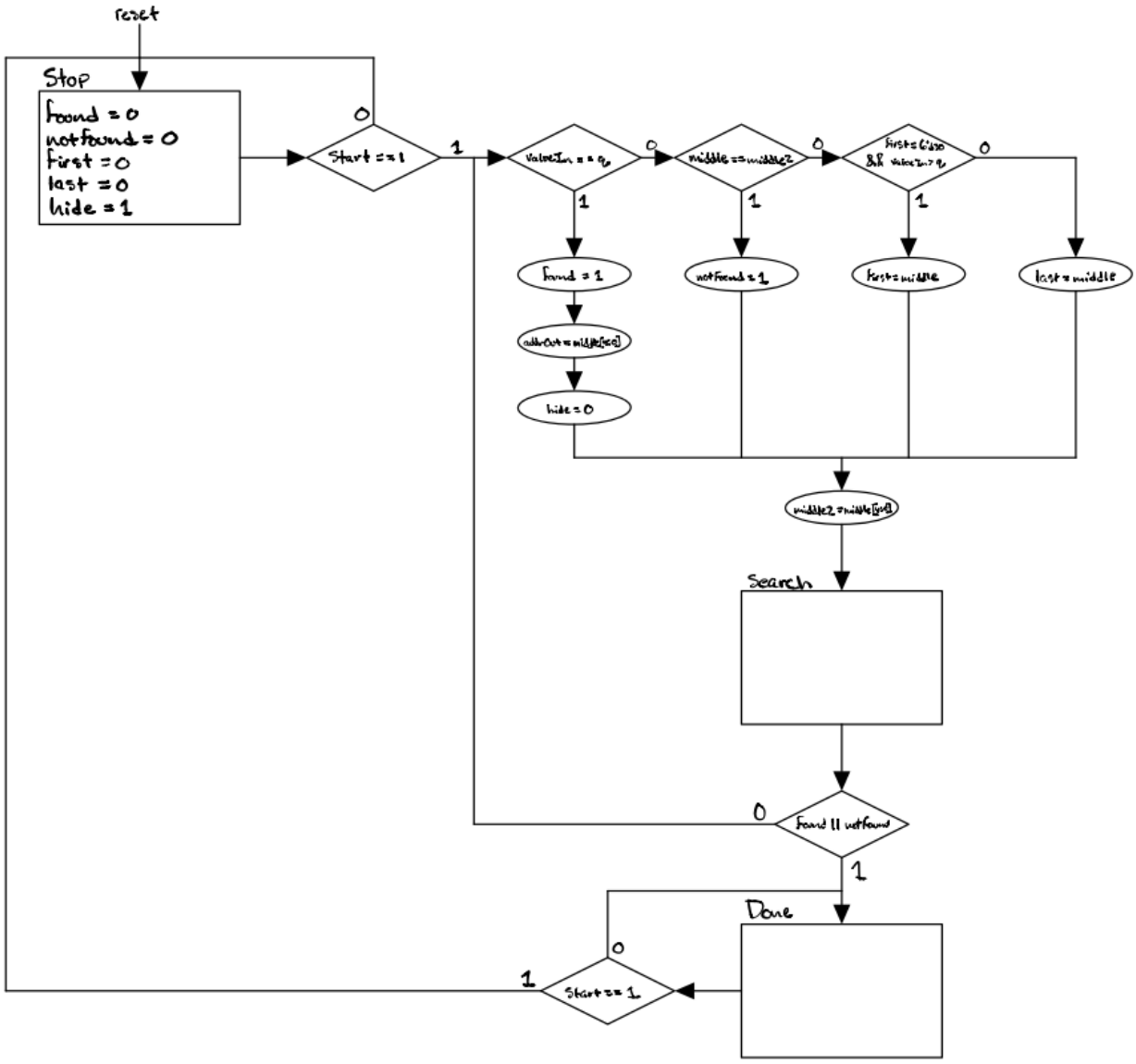



*Figure 7. FSM chart and ASMD chart for task 2.*

The system goes through three states; stop, search, and done. When it is in the stop state, everything remains off and nothing will move. The LEDs representing the found and notfound values will remain off. When the start switch is true, it will move to the search state. In the search state, the binary search algorithm is implemented and will remain in this state until the search is done. The search is done when either found or notfound is true, which is determined through the binary search algorithm I described earlier. It moves to the done state, which maintains the values of found and notfound until the start switch is false. This state was implemented so that the start switch can act as a soft reset. The user will use the switches SW7-0 to determine an input value, then press the start switch. CLOCK_50 is fast enough where it will take no longer than a second to search through the RAM. For the user's convenience, the done state will

maintain the found or notfound values so they can see the respective LEDs turn on. When the start switch is turned off, it goes back to the stop state and the user can input a new value before pressing the start switch again.

Now we needed to combine the control circuit with the datapath circuit as well as the 32x8 RAM. The datapath circuit will be under an always_ff block, and implements the binary search algorithm I described earlier. However, it will only update values such as first, last, and middle when it is in the search state. I simply made it so that when it is in the search state, it will implement my algorithm, and when it is not in the search state, first, last, found, and notfound will all maintain their values. If the system is reset, or when the state moves from done to stop, first will be “0”, last will be “31”, found will be 0 and notfound will be 0. This essentially resets the algorithm, and allows for many values to be checked without pressing the reset key more than one time at the beginning. The 32x8 RAM unit was created using the IP catalog within the Quartus Prime software and the MIF file initialized the values within each address. The write address, write enable, and input data were imputed zeros to prevent the RAM from rewriting any of the values. By combining the RAM, control circuit, and datapath circuit, we successfully created a binary search algorithm that followed the requirements of the lab specifications.

## Results

The second task consisted of five modules. Each of these modules have been simulated in ModelSim to test the behavior of the SystemVerilog code.

The first module was the two DFF's in series used to prevent metastability from the switches. Shown in Figure 9 is the ModelSim simulation of series_dffs.sv, displaying that the input values are output synchronous to the clock two clock cycles after the input timing.

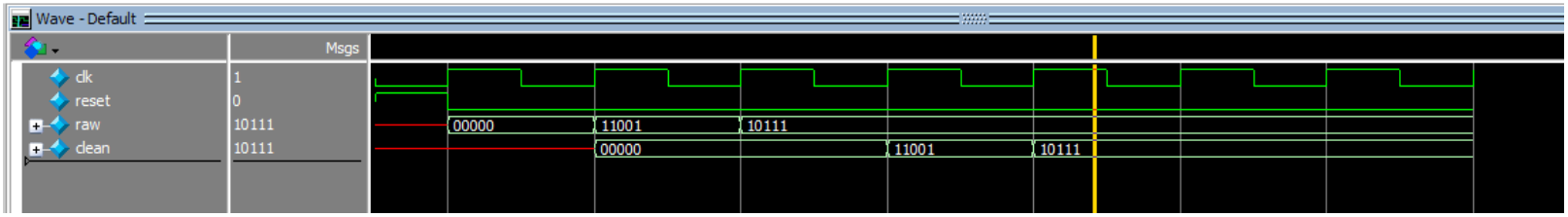



*Figure 9: Waveform produced by series_dffs.sv testbench module.*

The second module was the module used to convert the output address of the binary search to the HEX displays HEX1 and HEX0. In this lab, the HEX displays had to be turned off when the input value was not found or during the binary search procedure, so a new variable was implemented. Shown in Figure 10 is the input values bcd displaying the correct values into the HEX display so that the display is showing the hexadecimal value, as well as turning the HEX display off when it needs to be hidden.

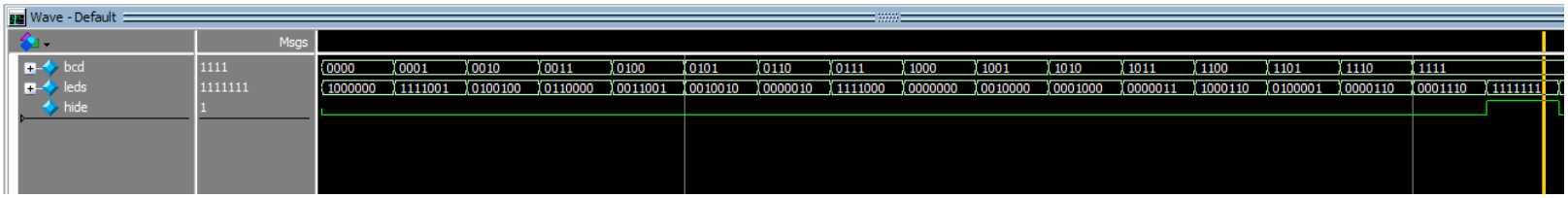



*Figure 10: Waveform produced by seg7hex.sv testbench module.*

The third module was the RAM unit generated using the IP catalog. The ModelSim simulation was used to test if the RAM was behaving as it should. I wrote a lot of data into many addresses, then read them all to make sure a read and write operation worked properly. Then, I rewrote new values into a couple of addresses and read them to make sure the RAM can be rewritten in the same addresses. Then, I read random addresses I did not write into to see if the MIF file I made for this task was being read from properly. The simulation produced results I expected and they can be seen in Figure 11.

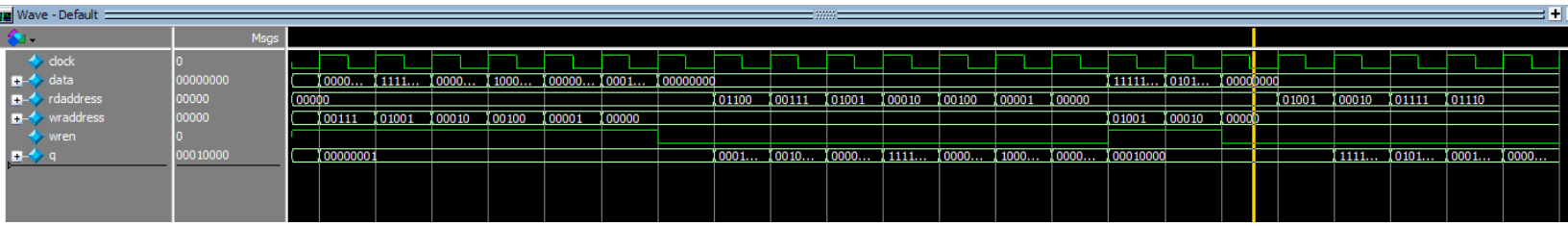



*Figure 11: Waveform produced by ram32x8.sv testbench module.*

The fourth module was binary_search.sv that implemented our binary search algorithm. The MIF file contains values 1-32 from addresses 0-31, and the simulation was used to test if the found, notfound, hide, and addrOut signals output the correct values. The first situation tested was if the input value was 0, which is below the range of values in the RAM. The other situation we tested was if the value was 33, just above the range of values in the RAM. If both of these input values led to notfound being true, hide being true, and addrOut not being displayed, it is a success. The next two situations involved values that could be found in the RAM. Since the “middle pointer” is initially at 15, I tested two cases where the input value is above 15 and below 15. This way, we can see if the algorithm works in both directions. The simulation produced results I expected and they can be seen in Figure 12.

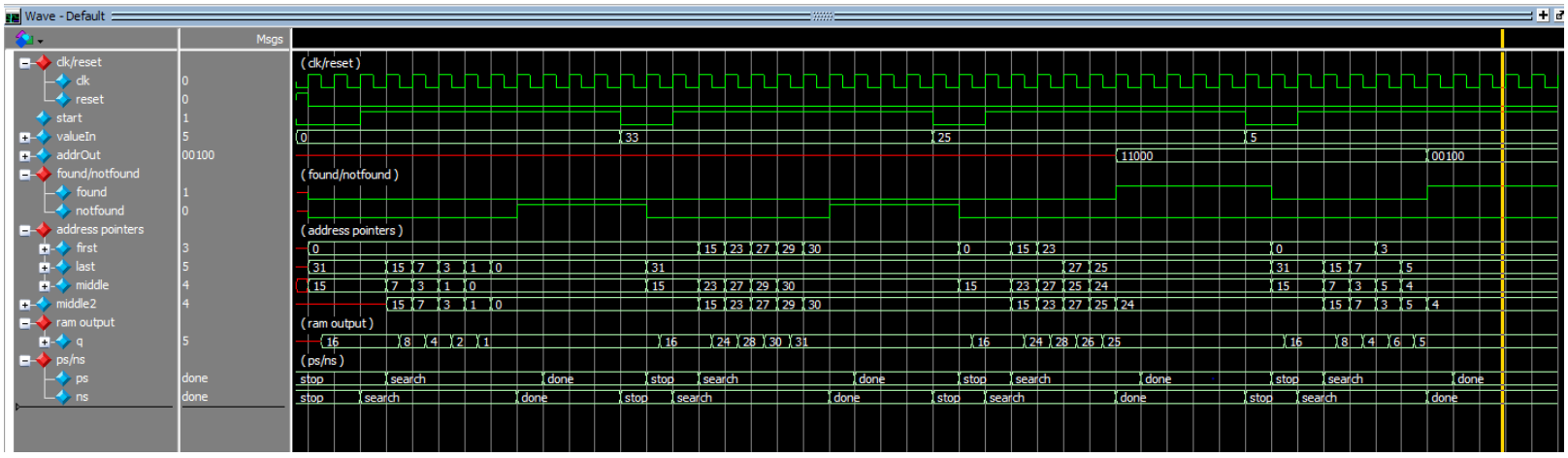



*Figure 12: Waveform produced by binary_search.sv testbench module.*

The last module was DE1_SoC, where all of the modules so far combine to complete the lab. Testing this module was similar to testing binary_search.sv, where we tested multiple situations where notfound should be true, and multiple situations where the value should be found. In the DE1_SoC simulation, the outputs are the LEDR 9-8, HEX1, and HEX0. The simulation produced results I expected and they can be seen in Figure 13.

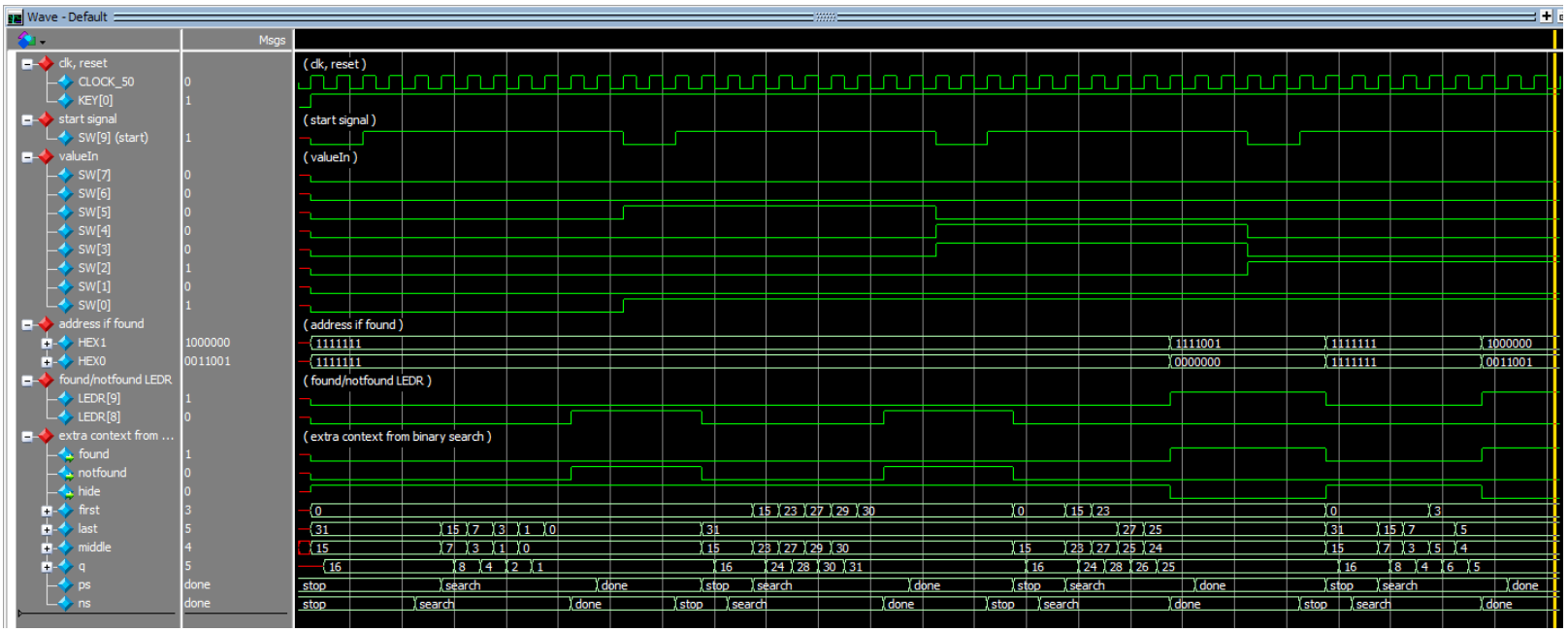



*Figure 13: Waveform produced by binary_search.sv testbench module.*

## Block Diagram - Task 2

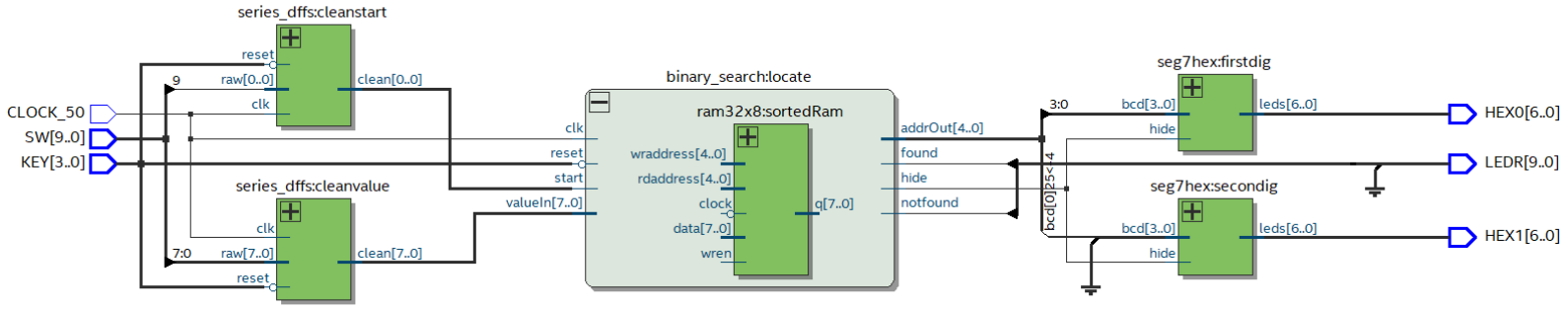



*Figure 14: Block Diagram for the task 2 design implementation.*

**Design Decisions - Task 2**

In this task, we made several key design decisions throughout the process of completing the task. The first design decision was to determine when the search is over when inputting a value that is not within the RAM unit. Initially, we thought the only values that would not be in the RAM would be those that are out of bounds. For example, if the values contained were 1-32, the values 0 and 33 are out of bounds. Thus, the search ends when the “middle” pointer is at address 0 or address 31. This logic has a potential flaw where if the MIF file contains values that are incremented by integer values greater than 1, there will be holes within these values. For example, if the values started from 3 and incremented by 2, an input value such as 4 will not be in the RAM unit. To stop the search in this case, there will be a point where the first and last pointers do not move, which means the middle pointer will stay the same for two searches. Thus, we made another intermediate logic that will equal the value of the middle pointer one clock cycle after the middle pointer changes. This way, when the middle pointer remains the same for more than one search, this intermediate logic will be the same and when both are the same before the intermediate logic changes, the search will end.

Another design decision we made was creating a third state called “done”. The done state was added simply so that the user can see the result of the search. We designed it so that when the user presses the start switch to start the binary search, CLOCK_50 is fast enough to the point where the search is over within a second, so while the start switch is on, it will maintain all of the output values after the search is over until the start switch is off. This was a quality of life improvement made for the user to see the outcome of their search.

**Summary Conclusion**

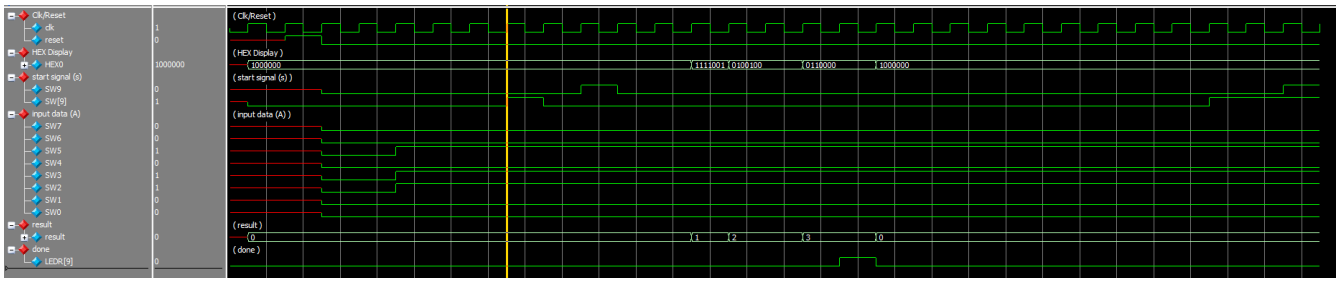
The main goal of this lab was to learn how to read, create, and implement code using ASMD charts. The first task was to read and comprehend the ASMD chart to implement the bit counting module. The second task was more complicated and required us to create our own ASMD chart so that we can code the binary search algorithm. We learned the difference between control circuits and datapath circuits and how they can combine within a module to output the behavior we want.

Overall, my lab provided the results I wanted and I believe it is sufficient in covering the requirements of this lab. The special cases were covered for and the primary functions work perfectly.

# Appendix

## Task 1:

1. DE1_SoC.sv

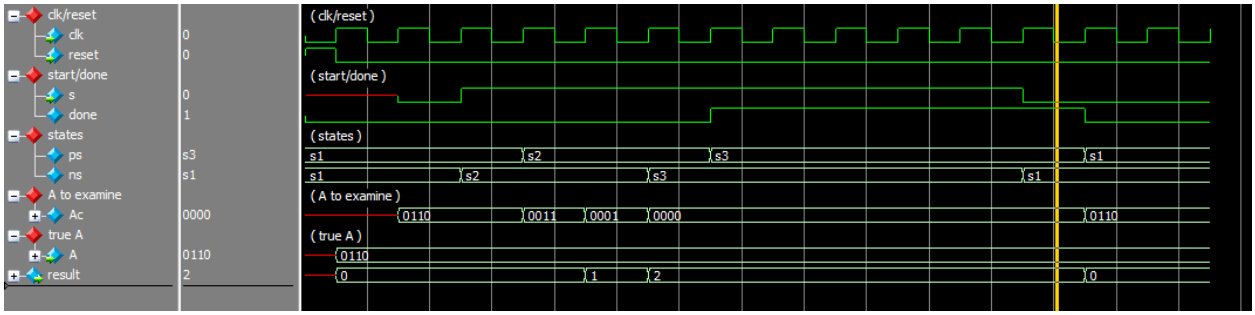



```
// Erik Michel & Brian Masaki
// 4/21/2021
// EE 371
// Lab #4, Implementing Algorithms in Hardware

// DE1_SoC takes a 1-bit CLOCK_50 signal, a 4-bit key signal, and a 10-bit SW signal.
// DE1_SoC returns 7 7-bit HEX signals of which only HEX0 is used to show the
// count of ON bits in a given n-bit value. LEDR9 is returned to display when the
// count of ON bits has finished.
module DE1_SoC (KEY, SW, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, LEDR, CLOCK_50);
   input logic CLOCK_50;
   input  logic [3:0]  KEY;
   input  logic [9:0]  SW;
   output logic [6:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0]  LEDR;

   // initializing logic for filtered switch signals
   logic SW9, SW7, SW6, SW5, SW4, SW3, SW2, SW1, SW0;

   logic [7:0] result;

   assign HEX1 = '1;
   assign HEX2 = '1;
   assign HEX3 = '1;
   assign HEX4 = '1;
   assign HEX5 = '1;

   // initializing the input reset signal by first passing the reset key
   // (KEY[0]) through a DFF filter (2ff's).
   logic reset, resLink1, noFiltR;
   assign noFiltR = !KEY[0];
   noResetFF FFRES1 (.q(resLink1), .d(noFiltR), .clk(CLOCK_50));
   noResetFF FFRES2 (.q(reset), .d(resLink1), .clk(CLOCK_50));

   // Filtering switch input (s) signal through two FF's
   DFF_Filter sw9fil (.clk(CLOCK_50), .reset(reset), .in(SW[9]), .out(SW9));

   // Filtering switch input (data-in/A) signals through two FF's
   DFF_Filter sw7fil (.clk(CLOCK_50), .reset(reset), .in(SW[7]), .out(SW7));
   DFF_Filter sw6fil (.clk(CLOCK_50), .reset(reset), .in(SW[6]), .out(SW6));
   DFF_Filter sw5fil (.clk(CLOCK_50), .reset(reset), .in(SW[5]), .out(SW5));
   DFF_Filter sw4fil (.clk(CLOCK_50), .reset(reset), .in(SW[4]), .out(SW4));
   DFF_Filter sw3fil (.clk(CLOCK_50), .reset(reset), .in(SW[3]), .out(SW3));
   DFF_Filter sw2fil (.clk(CLOCK_50), .reset(reset), .in(SW[2]), .out(SW2));
   DFF_Filter sw1fil (.clk(CLOCK_50), .reset(reset), .in(SW[1]), .out(SW1));
   DFF_Filter sw0fil (.clk(CLOCK_50), .reset(reset), .in(SW[0]), .out(SW0));

   // countOnes takes 1-bit CLOCK_50 signal, 1-bit reset signal, 1-bit SW9 signal
   // 8 1-bit SW7-0 signals concatenated into one 8-bit signal and returns an 8-bit
   // result signal that is a count of the number of ON bits in the concatenated input
   // signal. Also returns a single-bit signal to LEDR[9] signifying that the ON bit
   // count is complete.
   posBitCounter #(.WIDTH(8)) countOnes (.clk(CLOCK_50), .reset(reset), .s(SW9),
                                         .A({SW7, SW6, SW5, SW4, SW3, SW2, SW1, SW0}), .result, .done(LEDR[9]));


   // module used to display the "result" count on hex display HEX0
   hexaDig0 display (.in3(result[3]), .in2(result[2]), .in1(result[1]), .in0(result[0]), .out(HEX0));

endmodule
```

Testbench:

```
// DE1_SoC_testbench tests if the posBitCounter module "countOnes" outputs the proper
// values by displaying the values on the HEX0 display by using a previously used
// hexDig0 module instantiation that displays values on a HEX dsiplay. Various
// combinations of input values were used to ensure that the modules were working
// properly and displaying the correct results.
module DE1_SoC_testbench();
   logic clk;
   logic [3:0]  KEY;
   logic [6:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic [9:0]  SW;
   logic [9:0]  LEDR;

   DE1_SoC dut (.KEY, .SW, .HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .LEDR, .CLOCK_50(clk));

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

   initial begin

      KEY[0] <= 0;                                                                  @(posedge clk);
      KEY[0] <= 1; SW[9] <= 0; // s signal = 0
      // line 87 used to adjust 8-bit input values
      {SW[7], SW[6], SW[5], SW[4], SW[3], SW[2], SW[1], SW[0]} <= 8'b00101100;    @(posedge clk);
                                                                                    @(posedge clk);
      SW[9] <= 0; /* s (start) = 0  */                                            @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
      SW[9] <= 1; /* s (start) = 1 */                                             @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
      SW[9] <= 0; /* s (start) = 0 */                                             @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
                                                                                    @(posedge clk);
      $stop;
   end
endmodule
```

2. posBitCounter.sv




```
// Erik Michel & Brian Masaki
// 4/21/2021
// EE 371
// Lab #4, Implementing Algorithms in Hardware

// posBitCounter takes 1-bit clk and reset signals. Also takes 1-bit s signal
// representing a "start" signal that commences the counting of ON bits. Also
// takes a WIDTH-bit A signal that represents the value to count ON bits from.
// returns a WIDTH-bit result signal that is the count of ON bits in input
// signal A. Returns 1-bit done signal that is ON when count of ON bits in
// signal A has completed.
module posBitCounter #(parameter WIDTH = 4) (clk, reset, s, A, result, done);
   input logic             clk, reset, s; // s is "start" signal
   input logic  [WIDTH-1:0] A;
   output logic [WIDTH-1:0] result; // doesn't have to be WIDHT-bits wide
   output logic done;

   // Intermediate copy of A used to enable shifting on an input
   logic [WIDTH-1:0] Ac;

   // s1: beginning, s2: counting on bits, s3: completion
   enum {s1, s2, s3} ps, ns;

   always_comb begin
      case (ps)
         s1:   begin
                  done = 0;
                  if (s)      ns = s2;
                  else        ns = s1;
               end

         s2:   begin
                  done = 0;
                  if (Ac == 0) ns = s3; // when all ON bits counted
                  else         ns = s2;
               end

         s3:   begin
                  done = 1;
                  if (s)      ns = s3;
                  else        ns = s1; // when !s prepare to start again
               end

      endcase
   end

   always_ff @(posedge clk) begin
      if (reset)
         ps <= s1;
      else
         ps <= ns;
         case (ns)
            s1: begin
                  result <= 0; // reseting ON bit count to zero
                  Ac <= A; // assigning Ac the value of our input data A
                end

            s2: begin
                  Ac <= (Ac >> 1); // shift input data bits each cycle
                  if (Ac > 0 && Ac[0]) begin // input data is still > 0 & if ON bit is detected at Ac[0]
                     result <= result + 1;
                  end
                end

            s3: begin   // no actions required for s3 in this design
                        // simply ps <= ns
                end
         endcase
   end
endmodule
```

Testbench:

```
// posBitcounter_testbench tests the function of the posBitCounter module
// by testing if the correct "result" signal is output based on a given
// "A" signal. In addition, this testbench allows for further inspection
// of the timing for when result is being incremented to determine if the
// increments to result are appropriate given the input.
module posBitCounter_testbench();
   parameter WIDTH = 8;

   logic              clk, reset, s, done;      // s is "start" signal
   logic [WIDTH-1:0]  A;
   logic [WIDTH-1:0]  result;             // doesn't have to be WIDHT-bits wide

   posBitCounter #(.WIDTH(WIDTH)) dut (clk, reset, s, A, result, done);

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

   initial begin

      reset <= 1;                              @(posedge clk);
      reset <= 0; A <= 8'b00000110;            @(posedge clk); // testing with n ON bits
      s <= 0;                                  @(posedge clk);
      s <= 1;                                  @(posedge clk); // start signal enable
                                               @(posedge clk);
                                               @(posedge clk); // count process ...
      repeat (5) begin
         @(posedge clk);
      end
                                               @(posedge clk);
      s <= 0;                                  @(posedge clk); // start signal disable
                                               @(posedge clk);
                                               @(posedge clk);

      $stop;
   end
endmodule
```

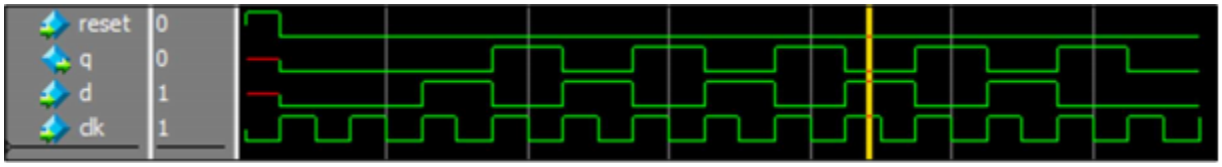
3. hexaDig0.sv


```
// Erik Michel & Brian Masaki
// 4/21/2021
// EE 371
// Lab #4, Implementing Algorithms in Hardware

// hexaDig0 takes a 4 1-bit input signals that together represent
// a hexadecimal value. hexaDig0 outputs a 7-bit out signal to drive
// a hex display based on the 4 1-bit input signals.
module hexaDig0 (in3, in2, in1, in0, out);
   input logic          in3, in2, in1, in0;
   output logic [6:0]   out;

   always_comb begin
      case({in3, in2, in1, in0})

         4'b0000: out = 7'b1000000; // 0

         4'b0001: out = 7'b1111001; // 1

         4'b0010: out = 7'b0100100; // 2

         4'b0011: out = 7'b0110000; // 3

         4'b0100: out = 7'b0011001; // 4

         4'b0101: out = 7'b0010010; // 5

         4'b0110: out = 7'b0000010; // 6

         4'b0111: out = 7'b1111000; // 7

         4'b1000: out = 7'b0000000; // 8

         4'b1001: out = 7'b0010000; // 9

         4'b1010: out = 7'b0001000; // A

         4'b1011: out = 7'b0000011; // b

         4'b1100: out = 7'b1000110; // C

         4'b1101: out = 7'b0100001; // d

         4'b1110: out = 7'b0000110; // E

         4'b1111: out = 7'b0001110; // F

         default: out = 7'bx; // 0

      endcase
   end

endmodule
```

Testbench:

```
// hexaDig0_testbench tests if the proper values are
// output according to the 4 input signals.
module hexaDig0_testbench();
   logic       in3, in2, in1, in0;
   logic [6:0] out;

 hexaDig0 dut (in3, in2, in1, in0, out);

   //Try all combinations with switches I am using (SW7-SW0)
   integer i;
   initial begin
      for(i = 0; i < 16; i++) begin
         {in3,in2,in1,in0} = i; #10;
      end
   end

endmodule
```

4. D_FF.sv

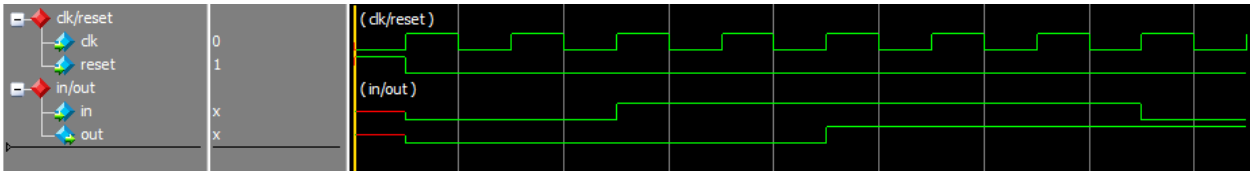



```
// Erik Michel & Brian Masaki
// 4/21/2021
// EE 371
// Lab #4, Implementing Algorithms in Hardware

// D_FF takes a 1-bit clk signal, a 1-bit reset signala 1-bit input 'd',
// and returns a 1-bit signal 'q' on the rising edge of the clk signal.
// 'q' is whatever value 'd' was right before the rising edge of the clock.
// when the reset signal is detected, the value of 'q' will become zero
// regardless of the value of 'd'.
module D_FF (q, d, reset, clk);
   output logic q;
   input logic  d, reset, clk;

   always_ff @(posedge clk) begin
      if (reset)
         q <= 0; //on reset, set to 0
      else
         q <= d; //otherwise out = d
      end
endmodule
```

Testbench:

```
// D_FF_testbench tests that given an input d, the noResetFF
// module will output the proper value for q on the rising edge of
// a clk signal
module D_FF_testbench();
   logic  q;
   logic  d, reset, clk;

   D_FF dut (q, d, reset, clk);

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

   initial begin

      reset <= 1;                  @(posedge clk);
      reset <= 0; d <= 0;          @(posedge clk);
                                   @(posedge clk);

      repeat (5) begin
         d <= 1;                   @(posedge clk);
         d <= 0;                   @(posedge clk);
      end
                                   @(posedge clk);

      $stop;

   end
endmodule
```

5. DFF_Filter.sv

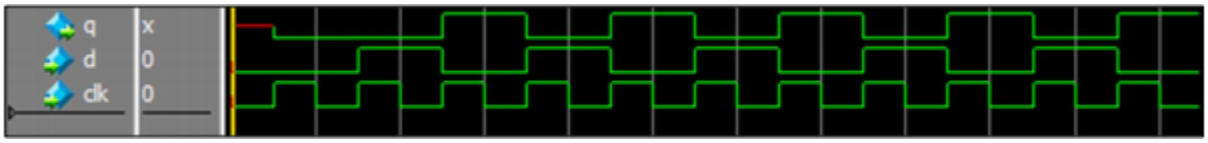



```
// Erik Michel & Brian Masaki
// 4/21/2021
// EE 371
// Lab #4, Implementing Algorithms in Hardware

// DFF_Filter takes a 1-bit clk signal, a 1-bit reset signal, and a
// 1-bit in signal. The in signal is then passed through two instantiated
// flip flop modules to "filter" the input signal and
module DFF_Filter (clk, reset, in, out);
   input logic  clk, reset;
   input logic  in;
   output logic out;

   logic a2b;

   D_FF FF_A (.q(a2b), .d(in), .reset(reset), .clk(clk));
   D_FF FF_B (.q(out), .d(a2b), .reset(reset), .clk(clk));

endmodule
```

Testbench:

```
// DFF_Filter_testbench tets if the DFF filter works properly
// to avoid metastability by passing a signal through two D_FF's
module DFF_Filter_testbench();
   logic  clk, reset;
   logic  in;
   logic out;

   DFF_Filter dut (clk, reset, in, out);

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

   initial begin

      reset <= 1;                @(posedge clk);
      reset <= 0; in <= 0;       @(posedge clk);
                                 @(posedge clk);
      in <= 1;
      repeat (5) begin
         @(posedge clk);
      end
      in <= 0;                   @(posedge clk);

      $stop;

   end
endmodule
```

6. noResetFF.sv




```
// Erik Michel & Brian Masaki
// 4/21/2021
// EE 371
// Lab #4, Implementing Algorithms in Hardware

// noResetFF takes a 1-bit clk signal, a 1-bit input 'd', and returns
// a 1-bit signal 'q' on the rising edge of the clk signal. ''q' is
// whatever value 'd' was before the rising edge of the clock.
module noResetFF (q, d, clk);
    output logic q;
    input logic  d, clk;

    always_ff @(posedge clk) begin
        q <= d;
    end
endmodule
```

Testbench:

```
// noResetFF_testbench tests that given an input d, the noResetFF
// module will output the proper value for q on the rising edge of
// a clk signal
module noResetFF_testbench();
    logic  q;
    logic  d, clk;

    noResetFF dut (q, d, clk);

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

    initial begin

        d <= 0;                     @(posedge clk);
                                    @(posedge clk);
        repeat (5) begin
            d <= 1;                 @(posedge clk);
            d <= 0;                 @(posedge clk);
        end
                                    @(posedge clk);

        $stop;

    end
endmodule
```

## Task 2:

7. series_dffs.sv (code and testbench)

```
1   // Brian Dallaire
2   // 05/16/2021
3   // EE 371
4   // Lab #4, Task 2
5
6   // series_dffs takes the 1-bit inputs clk and reset and variable-bit input raw, and outputs the variable-bit output clean.
7   // The purpose of this module is that it represents two DFFs in series. When a signal goes through two DFFs, it is very
8   // difficult for it to reach metastability with other signals. I used a global parameter in this module so that multiple
9   // bit inputs are possible without having to call this module more than once.
10
11  module series_dffs # (parameter BITS = 1) (clk, reset, raw, clean);
12
13     input logic clk, reset;
14     input logic [BITS-1 : 0] raw;
15     output logic [BITS-1 : 0] clean;
16     logic [BITS-1 : 0] n1;
17
18     // always_ff block that shows the DFFs displacing the signal. The input signal goes into the first DFF and
19     // the output of the first DFF goes in as the input to the second DFF. The output of the second DFF is the
20     // output clean
21
22     always_ff @(posedge clk) begin
23        if(reset) begin
24           n1    <= '0;
25           clean <= '0;
26        end else
27           n1    <= raw;
28           clean <= n1;
29
30     end
31
32  endmodule
33
34  // series_dffs_testbench tests to see if the flip flops in series works properly even with multiple bit inputs
35  // I tested to see if different values will bug and output incorrectly
36
37  module series_dffs_testbench();
38     logic clk, reset;
39     logic [4:0] raw;
40     logic [4:0] clean;
41
42     series_dffs #(.BITS(5)) dut (.clk, .reset, .raw, .clean);
43
44     parameter CLOCK_PERIOD=100;
45     initial begin
46        clk <= 0;
47        forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
48     end
49
50     initial begin
51        reset <= 1;                    @(posedge clk);
52        reset <= 0; raw <= 5'b00000; @(posedge clk);
53                    raw <= 5'b11001; @(posedge clk);
54                    raw <= 5'b10111; @(posedge clk);
55                           repeat(4) @(posedge clk);
56
57        $stop;
58     end
59  endmodule
60
61
```

8. seg7hex.sv (code)

```
// Brian Dallaire
// 05/16/2021
// EE 371
// Lab #4, Task 2

// seg7hex takes a 4-bit input bcd and outputs a 7-bit output leds. This module takes the 4-bit input that refers to
// either the data or address that is supposed to be displayed onto a 7-segment HEX display. The name seg7hex means
// that the purpose of this module is to convert the 4-bit input into a hexadeicmal value on the HEX displays.

module seg7hex (bcd, hide, leds);
   input logic [3:0] bcd;
   input logic hide;
   output logic [6:0] leds;

   // this always_comb block uses the 4-bit input bcd as a case, and returns the value of leds in a way that
   // displays the hexadecimal value of the 4-bit input. It is clearly visible that all of the one digit
   // hexadecimal values are covered from 0 - F. The hide input determines whether to turn the HEX display
   // off or display the value determined by input bcd.

   always_comb begin
      if(hide) begin
         leds = 7'b1111111;
      end else begin
         case (bcd)
            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'bX;
         endcase
      end
   end
endmodule
```

(testbench)

```
// seg7hex_testbench tests all of the cases of the 4-bit input bcd and tests to see if the output
// leds correspond to the hexadecimal value in the HEX display. The only thing to test for in this
// testbench is to go through all of the cases, and see if the hide input turns the HEX display off.

module seg7hex_testbench();
   logic [3:0] bcd;
   logic [6:0] leds;
   logic hide;

   seg7hex dut (.bcd, .hide, .leds);

   initial begin
      bcd <= 4'b0000; hide <= 0; #10;
      bcd <= 4'b0001; #10;
      bcd <= 4'b0010; #10;
      bcd <= 4'b0011; #10;
      bcd <= 4'b0100; #10;
      bcd <= 4'b0101; #10;
      bcd <= 4'b0110; #10;
      bcd <= 4'b0111; #10;
      bcd <= 4'b1000; #10;
      bcd <= 4'b1001; #10;
      bcd <= 4'b1010; #10;
      bcd <= 4'b1011; #10;
      bcd <= 4'b1100; #10;
      bcd <= 4'b1101; #10;
      bcd <= 4'b1110; #10;
      bcd <= 4'b1111; #10;
      hide <= 1;      #10;
      hide <= 0;      #10;
      $stop;
   end
endmodule
```

## 9. ram32x8.sv (code)

```
// Brian Dallaire
// 05/16/2021
// EE 371
// Lab #4, Task 2

// megafunction wizard: %RAM: 2-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: ram32x8.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 ram32x8 (
    clock,
    data,
    rdaddress,
    wraddress,
    wren,
    q);

    input   clock;
    input [7:0]  data;
    input [4:0]  rdaddress;
    input [4:0]  wraddress;
    input   wren;
    output   [7:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1    clock;
    tri0    wren;
`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 (wraddress),
            .address_b (rdaddress),
            .clock0 (clock),
            .data_a (data),
            .wren_a (wren),
            .q_b (sub_wire0),
            .aclr0 (1'b0),
            .aclr1 (1'b0),
            .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 ({8{1'b1}}),
            .eccstatus (),
            .q_a (),
            .rden_a (1'b1),
            .rden_b (1'b1),
            .wren_b (1'b0));
  defparam
     altsyncram_component.address_aclr_b = "NONE",
     altsyncram_component.address_reg_b = "CLOCK0",
     altsyncram_component.clock_enable_input_a = "BYPASS",
     altsyncram_component.clock_enable_input_b = "BYPASS",
     altsyncram_component.clock_enable_output_b = "BYPASS",
     altsyncram_component.init_file = "my_array.mif",
     altsyncram_component.intended_device_family = "Cyclone V",
     altsyncram_component.lpm_type = "altsyncram",
     altsyncram_component.numwords_a = 32,
     altsyncram_component.numwords_b = 32,
     altsyncram_component.operation_mode = "DUAL_PORT",
     altsyncram_component.outdata_aclr_b = "NONE",
     altsyncram_component.outdata_reg_b = "UNREGISTERED",
     altsyncram_component.power_up_uninitialized = "FALSE",
     altsyncram_component.ram_block_type = "M10K",
     altsyncram_component.read_during_write_mode_mixed_ports = "DONT_CARE",
     altsyncram_component.widthad_a = 5,
     altsyncram_component.widthad_b = 5,
     altsyncram_component.width_a = 8,
     altsyncram_component.width_b = 8,
     altsyncram_component.width_byteena_a = 1;


endmodule
```

(testbench)

```
120
121   // ram32x8_testbench tests both the expected and unexpected cases for this module. In this simulation,
122   // I tested writing into many modules and then reading from the same addresses I wrote into. Then,
123   // I tried rewriting into some of those addresses and rereading to see if they can be overwritten more
124   // than one time. Then, I tried reading addresses I did not touch to see if it reads from the MIF file or
125   // is initialized to zero when unwritten.
126
127   `timescale 1 ps / 1 ps
128   module ram32x8_testbench();
129      reg clock;
130      reg   [7:0]  data;
131      reg   [4:0]  rdaddress;
132      reg   [4:0]  wraddress;
133      reg     wren;
134      wire  [7:0]  q;
135
136      ram32x8 dut (.clock(clock), .data(data), .rdaddress(rdaddress), .wraddress(wraddress), .wren(wren), .q(q));
137
138      parameter CLOCK_PERIOD=100;
139      initial begin
140         clock <= 0;
141         forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
142      end
143
144      initial begin
145         //test writing operation
146         rdaddress <= 5'b0000;
147         data <= 8'b00100000; wraddress <= 5'b1100; wren <= 1; @(posedge clock);
148         data <= 8'b00000101; wraddress <= 5'b0111; wren <= 1; @(posedge clock);
149         data <= 8'b11110000; wraddress <= 5'b1001; wren <= 1; @(posedge clock);
150         data <= 8'b00001010; wraddress <= 5'b0010; wren <= 1; @(posedge clock);
151         data <= 8'b10000000; wraddress <= 5'b0100; wren <= 1; @(posedge clock);
152         data <= 8'b00000100; wraddress <= 5'b0001; wren <= 1; @(posedge clock);
153         data <= 8'b00010000; wraddress <= 5'b0000; wren <= 1; @(posedge clock);
154         data <= 8'b00000000; wraddress <= 5'b0000; wren <= 0; @(posedge clock);
155
156         //test reading operation
157         rdaddress <= 5'b1100; @(posedge clock);
158         rdaddress <= 5'b0111; @(posedge clock);
159         rdaddress <= 5'b1001; @(posedge clock);
160         rdaddress <= 5'b0010; @(posedge clock);
161         rdaddress <= 5'b0100; @(posedge clock);
162         rdaddress <= 5'b0001; @(posedge clock);
163         rdaddress <= 5'b0000; @(posedge clock);
164
165         //test rewriting, rereading, and reading empty spaces
166         data <= 8'b11111111; wraddress <= 5'b1001; wren <= 1; @(posedge clock);
167         data <= 8'b01010011; wraddress <= 5'b0010; wren <= 1; @(posedge clock);
168         data <= 8'b00000000; wraddress <= 5'b0000; wren <= 0; @(posedge clock);
169         rdaddress <= 5'b1001; @(posedge clock);
170         rdaddress <= 5'b0010; @(posedge clock);
171
172         rdaddress <= 5'b1111; @(posedge clock);
173         rdaddress <= 5'b1110; @(posedge clock);
174                              @(posedge clock);
175
176         $stop;
177      end
178   endmodule
179
```

10. binary_search.sv (code)

```
// Brian Dallaire
// 05/16/2021
// EE 371
// Lab #4, Task 2

// binary_search takes the 1-bit inputs clk, reset, start and the 8-bit input valueIn and outputs the
// 5-bit output addrOut and the 1-bit outputs found, notfound, and hide. The main purpose of this module
// is to recursively search through the RAM unit to see if a value equivalent to valueIn is within the
// RAM. If it is in the RAM, found will return true as well as the address of the value from the RAM.
// If it is not in the RAM, notfound will return true and hide will remain true, keeping any displays
// to remain off.

module binary_search(clk, reset, start, valueIn, addrOut, found, notfound, hide);

   input logic clk, reset, start;
   input logic [7:0] valueIn;
   output logic [4:0] addrOut;
   output logic found, notfound, hide;

   logic [5:0] first, last, middle;
   logic [4:0] middle2;
   logic [7:0] q;

   // assigns the middle pointer to (first + last) / 2
   assign middle = (first + last) >> 1;

   // ram32x8 takes the 1-bit inputs clk and wren and the 8 bit input data and the 5-bit inputs
   // rdaddress and wraddress and outputs the 8-bit output q. The main purpose of this module is
   // that it is the RAM unit used in task 2 and contains the values from the MIF file. data,
   // wraddress, and wren are all set to zero and this is purely a read only RAM unit.
   ram32x8 sortedRam (.clock(~clk), .data(8'b0), .rdaddress(middle[4:0]), .wraddress(5'b0), .wren(1'b0), .q);

   enum {stop, search, done} ps, ns;

   // this always comb block implements the FSM used for the binary search algorithm. It has
   // three states and only proceeds to search during the "search" state. The 1-bit input start
   // determines if it stays or moves from the states stop and done.
   always_comb begin
      case(ps)
      stop   :  if(start) ns = search;
                else      ns = stop;

      search :  if(found | notfound)
                          ns = done;
                else      ns = search;

      done   : if(start) ns = done;
               else      ns = stop;
      endcase
   end

   // this always_ff block is used to move the present state to the next state every clock
   // cycle. If reset is true, it will set the present state to the initial state which is
   // the "stop" state.
   always_ff @(posedge clk) begin
      if(reset)
         ps <= stop;
      else
         ps <= ns;
   end


   // this always_ff block is the datapath circuit for the binary search algorithm. When reset,
   // the values of first and last will be set to 0 and 31, making the middle pointer 15. While
   // the next state is search, it will progress through the binary search and eventually end the
   // search, making either found or notfound true. When found is true, hide will be false and the
   // current address for the middle pointer will be assigned to the output addrOut.
   always_ff @(posedge clk) begin
      if(reset || (ps == done && ns == stop)) begin
         first <= '0;
         last  <= 6'b011111;
         found <= 0;
         notfound <= 0;
         hide <= 1;
      end else if(ns == search) begin
         if(valueIn == q) begin
            found <= 1;
            addrOut <= middle[4:0];
            hide <= 0;
         end else if(middle == middle2) begin
            notfound <= 1;
         end else if(first == 6'd30 && valueIn > q) begin
            first <= 6'd31;
         end else if(valueIn > q) begin
            first <= middle;
         end else begin
            last <= middle;
         end
         middle2 <= middle[4:0];

      end else begin
         first <= first;
         last <= last;
         found <= found;
         notfound <= notfound;
      end
   end
endmodule
```

(testbench)

```
 98
 99  // binary_search_testbench tests multiple cases to see if the binary search algorithm works properly.
100  // This testbench assumes that the MIF file increments by 1 and starts from 0 and ends at the value 32.
101  // It tests cases where the input value is 0 and 33, and cases where the value is above 15 and below 15,
102  // but within the boundaries of the RAM unit.
103
104  `timescale 1 ps / 1 ps
105  module binary_search_testbench();
106
107     logic clk, reset, start;
108     logic [7:0] valueIn;
109     logic [4:0] addrOut;
110     logic found;
111     logic notfound;
112     logic hide;
113
114     binary_search dut (.clk, .reset, .start, .valueIn, .addrOut, .found, .notfound, .hide);
115
116     parameter CLOCK_PERIOD = 100;
117     initial begin
118        clk <= 0;
119        forever # (CLOCK_PERIOD/2) clk <= ~clk; //Forever toggle clock
120     end
121
122     initial begin
123        reset <= 1; start <= 0; valueIn <= 8'b00000000;           @(posedge clk);
124        reset <= 0;                                    repeat(2)  @(posedge clk);
125                    start <= 1;                        repeat(10) @(posedge clk);
126                    start <= 0; valueIn <= 8'b00100001; repeat(2)  @(posedge clk);
127                    start <= 1;                        repeat(10) @(posedge clk);
128                    start <= 0; valueIn <= 8'b00011001; repeat(2)  @(posedge clk);
129                    start <= 1;                        repeat(10) @(posedge clk);
130                    start <= 0; valueIn <= 8'b00000101; repeat(2)  @(posedge clk);
131                    start <= 1;                        repeat(10) @(posedge clk);
132
133     $stop;
134     end
135  endmodule
136
```

## 11. DE1_SoC.sv (code)

```
// Brian Dallaire
// 04/23/2021
// EE 371
// Lab #2, Task 3

// DE1_SoC takes a 1-bit input CLOCK_50, 4-bit input KEY, 10-bit input SW and returns the six, 7-bit outputs
// HEX5-HEX0 and 10-bit output LEDR. DE1_SoC combines all of the modules in Task 2 and uses HEX1 and HEX0 to
// display the output address if applicable and LEDR9, LEDR8 if the input value determined by SW7-0 is found
// or not found. The output address is represented in hexadecimal.

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

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

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

   logic start, found, notfound, hide;
   logic [7:0] valueIn;
   logic [4:0] addrout;

   // series_dffs has a 1-bit input CLOCK_50, KEY[0], and 1-bit input SW[9] and ouptuts the 1-bit output
   // start. This ensures SW[7:0] outputs a clean signal and prevents metastability using two flip-flops in series
   series_dffs #(.BITS(1)) cleanstart (.clk(CLOCK_50), .reset(~KEY[0]), .raw(SW[9]), .clean(start));

   // series_dffs has a 1-bit input CLOCK_50, KEY[0], and 8-bit input SW[7:0] and ouptuts the 8-bit output
   // valueIn. This ensures SW[7:0] outputs a clean signal and prevents metastability using two flip-flops in series
   series_dffs #(.BITS(8)) cleanvalue (.clk(CLOCK_50), .reset(~KEY[0]), .raw(SW[7:0]), .clean(valueIn));

   // binary_search has the 1-bit inputs clk, reset, and start and 8-bit input valueIn and outputs the 5-bit output
   // addrOut and the 1-bit outputs found, notfound, and hide. The main purpose of this module is to perform the
   // binary search algorithm to find the value of valueIn within the RAM unit.
   binary_search locate (.clk(CLOCK_50), .reset(~KEY[0]), .start, .valueIn, .addrOut, .found, .notfound, .hide);

   // seg7hex takes the 4-bit input which is the first 4 bits of addrOut and the 1-bit input hide and outputs
   // the corresponding hexadecimal value to the 7-segment display HEX0. Hide determines if the display is
   // on or off.
   seg7hex firstdig (.bcd(addrout[3:0]), .hide, .leds(HEX0));

   // seg7hex takes the 4-bit input which is the first 4 bits of addrOut and the 1-bit input hide and outputs
   // the corresponding hexadecimal value to the 7-segment display HEX0. Hide determines if the display is
   // on or off.
   seg7hex secondig (.bcd({3'b000, addrout[4]}), .hide, .leds(HEX1));

   assign LEDR[9] = found;
   assign LEDR[8] = notfound;

endmodule
```

(testbench)

```
// DE1_SoC_testbench tests the expected and unexpected situations of this task. For this simulation,
// I tested what happens if I input values outside the RAM from both directions. I also tested values
// that were both above the initial starting address and below. This way, I can see that the task works
// properly in all cases.

`timescale 1 ps / 1 ps
module DE1_SoC_testbench();

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

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

   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever # (CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; //Forever toggle clock
   end

   initial begin

      KEY[0] <= 0;                                                    @(posedge CLOCK_50);
      KEY[0] <= 1; SW[9] <= 0; SW[7:0] <= 8'b00000000; repeat(2)  @(posedge CLOCK_50);
                   SW[9] <= 1;                         repeat(10) @(posedge CLOCK_50);
                   SW[9] <= 0; SW[7:0] <= 8'b00100001; repeat(2)  @(posedge CLOCK_50);
                   SW[9] <= 1;                         repeat(10) @(posedge CLOCK_50);
                   SW[9] <= 0; SW[7:0] <= 8'b00011001; repeat(2)  @(posedge CLOCK_50);
                   SW[9] <= 1;                         repeat(10) @(posedge CLOCK_50);
                   SW[9] <= 0; SW[7:0] <= 8'b00000101; repeat(2)  @(posedge CLOCK_50);
                   SW[9] <= 1;                         repeat(10) @(posedge CLOCK_50);

   $stop;
   end
endmodule
```