Lab 4 Report: Bit-Counting and Binary Search Algorithms in Hardware

Hunter North and Peter Zhong

EE 371

May 09, 2021

Procedure

Task One

Module: shiftReg8

I began this project by developing the 8-bit shift register (see Appendix 1A), which serves the purpose of holding the input data and then shifting it out to classify ones and zero. This module begins by copying the input data into another 8-bit unit. This 8-bit unit will shift data to the right when commanded, and fill zeros in on the left. In addition, it will also be monitoring the size (or count) of bits remaining to be shifted out. Once all data has been shifted out, the shift register is full of zero's and stops functioning. See the diagram in figure 1 to use this module with the correct inputs and outputs.

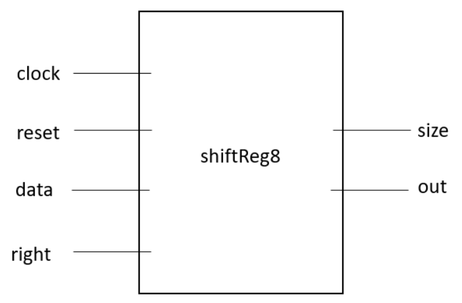


Figure 1. shiftReg8 Diagram. Figure to show the necessary inputs and outputs to use the shiftReg8 module.

Module: counter4

Next, I began developing the system for tracking the number of 1's found in the input data (see Appendix 1C). Since our shift register is constantly shifting out data from our input, I decided to take a counter that would continuously add these values. By adding all of these bits, we would be left with the count of all the ones in our input data. Shown below (in figure 2) is the four-bit counter used to implement the above. The module receives the same clock and reset signals as the shift register, but also receives the (shiftReg8) out data into its own add port. The module will constantly be outputting the running count of one bits that has been identified.

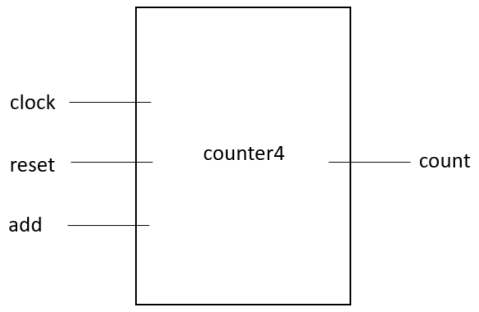


Figure 2. counter4Diagram. Figure to show the necessary inputs and outputs to use the counter4module.

Module: bitCounter

Then I began building the bitCounter module (see Appendix 1E), which serves as a control module to coordinate action between the shiftReg8 and counter4 modules. This module features three different states. The first state is waiting for the user to input the data to be analyzed. Then, a start signal is triggered to send the unit to state 2, which runs our bit counting algorithm. When the algorithm ends, the shift registers size is empty and we move to state3. In state3, the processes are held until the user turn the start signal back off to send the unit back to state 1. The state machine is shown in figure 3.

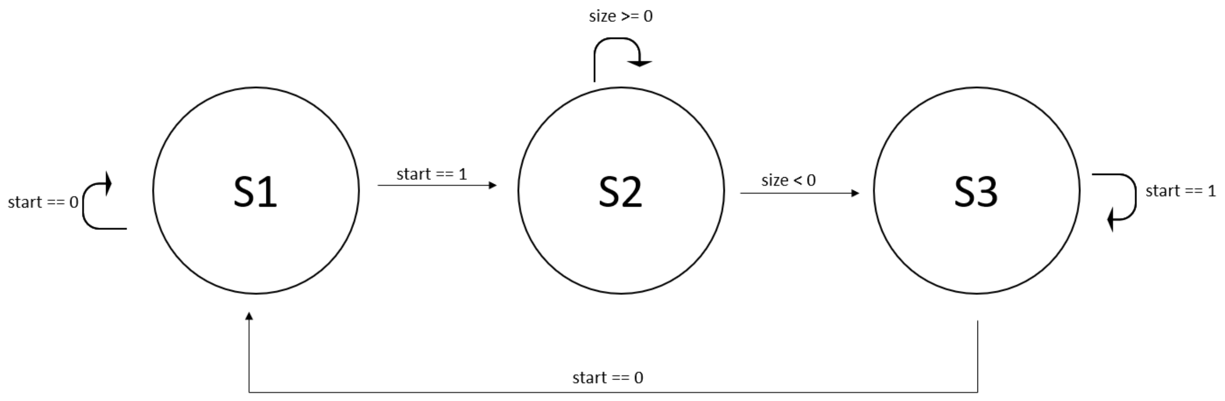


Figure 3. bitCounter FSM. Diagram that shows the conditions required to move the bitCounter module throughout its various states.

To properly use this module, a user will need to supply a clock (preferably 50 MHz) and a reset signal, which will return the module back to state 1. In addition, the user will have to supply an 8-bit data entry to be analyzed. Lastly, a start signal will be used to enter the algorithmic process in state 2 and leave the holding state in state 3. The system will output the count of ones in the data through the result port and will also send out a high signal on the done port when the algorithm ends. A shorter summary of utilizing the module can be found in figure 4.

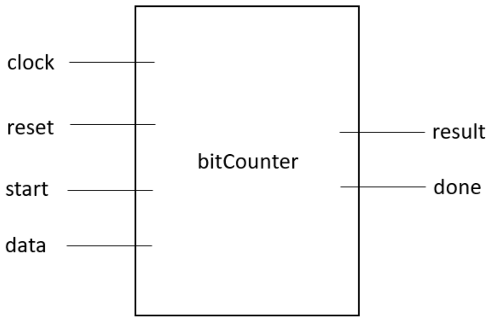


Figure 4. bitCounter Diagram. Figure to show the necessary inputs and outputs to use the bitCounter module.

Shown in figure 5 is an ASMD for our bitCounter module. This diagram was taken directly from the spec for lab 4. It is included to show another view on the actions and processes occurring within the bitCounter module.

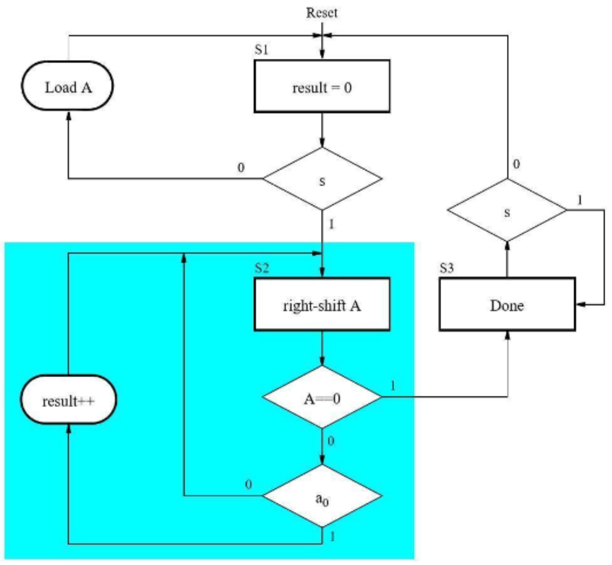


Figure 5. Bit-Counter ASMD. A figure for our implementation of a bit counting algorithm.

Module: seg7

To display the results of our bit counting algorithm, I was going to need a way in which I could translate a numeric (single digit) value onto a HEX display. My intuition was to use these seg7 modules (Appendix 1G) to communicate the result of the bitCounter in hexadecimal format. Seg7 displays receive a single digit, and then light up the hex display to show the number. The finalized block diagram is shown below (figure 6) to give users the idea of what inputs and outputs are required.

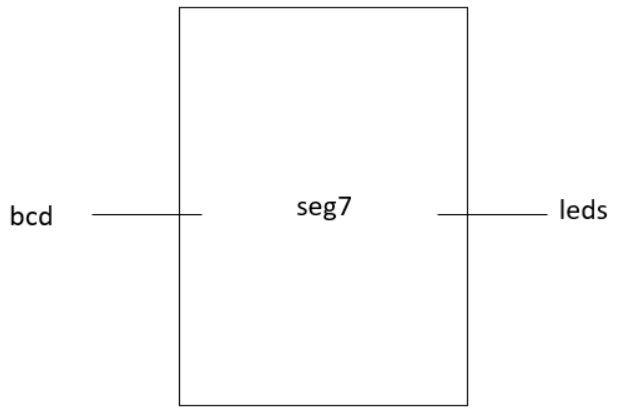


Figure 6. seg7 Diagram. Figure to show the necessary inputs and outputs to use the seg7 module.

Module: DE1_Soc

To conclude the lab task, I needed a module that would instantiate the bitCounter and seg7 display modules while also wiring up physical switches and displays to the actual hardware design. This module does not hold any logic and rather helps structure the overall program. The blow diagram below outlines the required inputs and outputs for the system. In addition, the code for this module is visible in Appendix 1I.

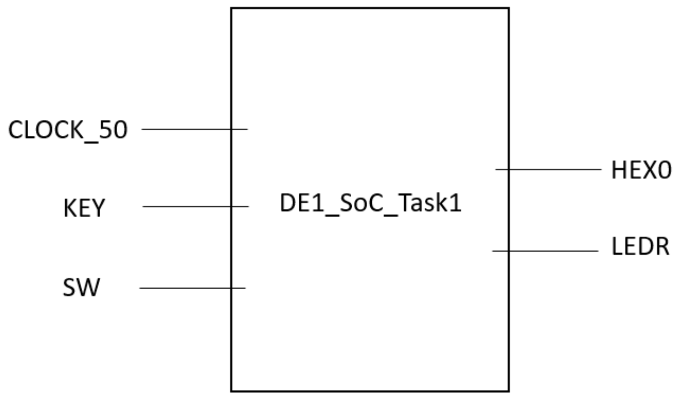


Figure 7. DE1_SoC_Task1 Diagram. Figure to show the necessary inputs and outputs to use the DE1_ SoC_Task1 module.

Task Two

Module: binary_search

Task 2 asks us to implement the binary search algorithm on hardware. To design this ASM, we first drew the ASMD chart (as shown in figure 8) to show the flow and timing of the algorithm. Specifically, we used conditions “q==a” and “size==1” to determine the three possible outcomes of the algorithm: found, keep searching, and not found. From the ASMD chart, I then proceeded to derive the state diagram for the control path of the ASM (see figure 9). The start signal will send the machine from idle state into search state, and the machine will keep searching until either q==a or size == 1. The machine will not return to idle state until the start signal becomes inactive.

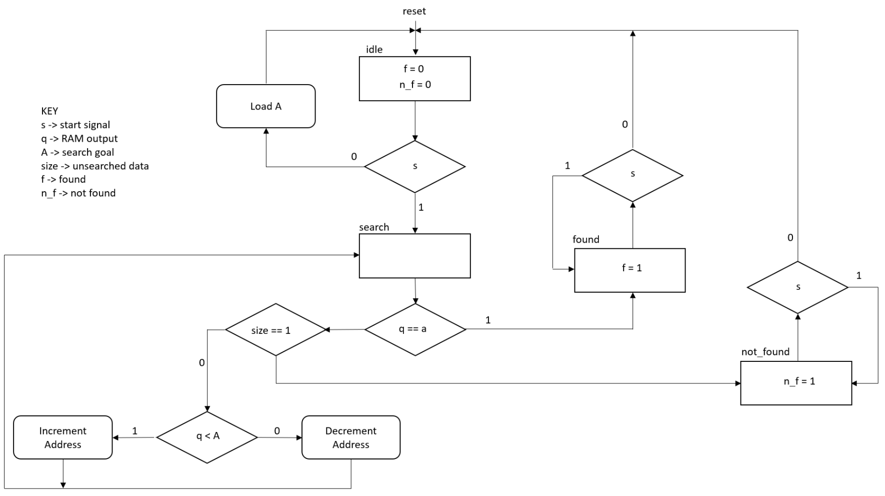


Figure 8. binary_search ASMD. A figure for our implementation of a binary_search algorithm.

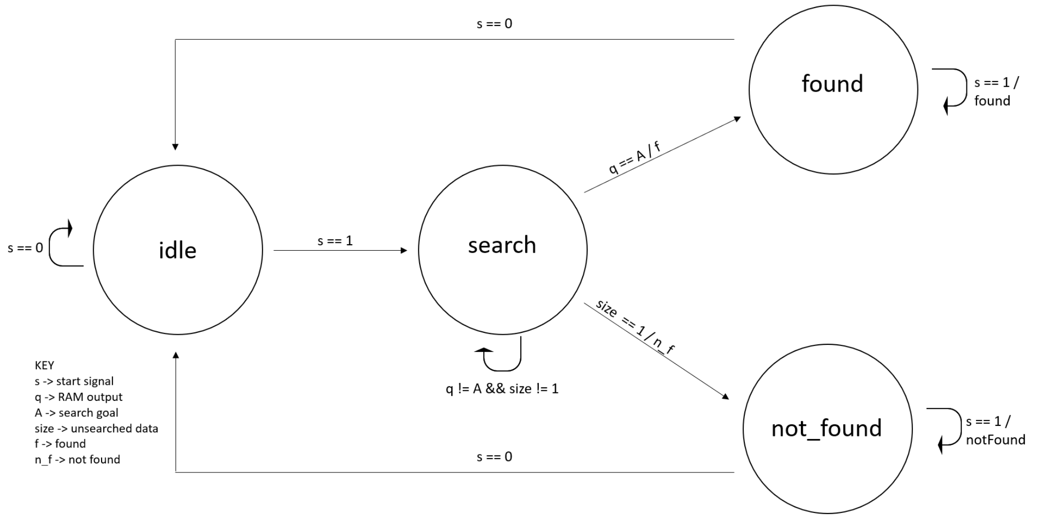


Figure 9. binary_search FSM. Diagram that shows the conditions required to move the binary_search module throughout its various states.

To utilize our module (Appendix 2A), a user must provide the signals shown below in Figure 10. The s_in port is representative of the start signal used to trigger the algorithm. The algorithm will run on the 8-bit input A_in. This system is timed according to the clk signal and is resettable back into the idle state. If A_in is found in our data, the f signal will go high and the address of the data will be sent through addr. If not found, the n_f signal will go high.

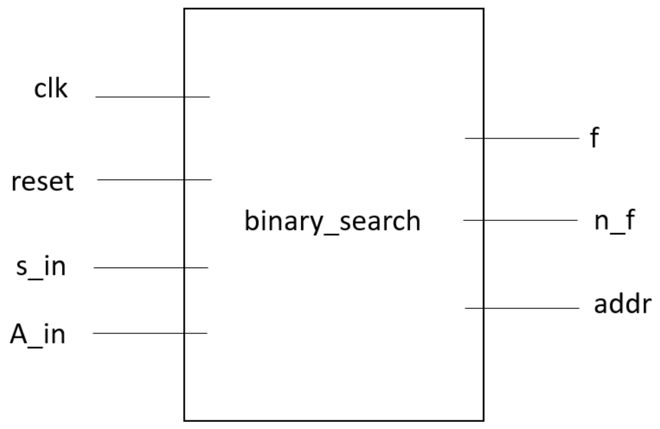


Figure 10. binary_search Diagram. Figure to show the necessary inputs and outputs to use the binary_search module.

In order to hold the data we are searching through, we needed to create some sort of data structure. We decided to create a RAM (see Figure 11 and Appendix 2C) with 32, 8-bit words. This RAM will receive a 5-bit address through the addr_in port and then send out the 8-bit data at said address through data_out. A user will also provide a clock through the clk port to coordinate timing within the system.

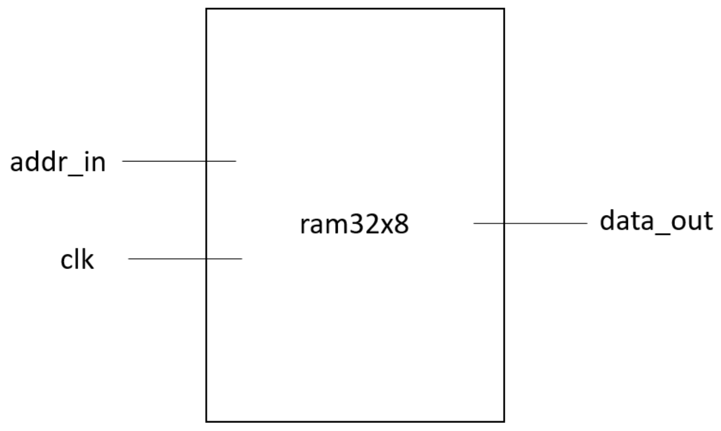


Figure 11. ram32x4 Diagram. Figure to show the necessary inputs and outputs to use the ram32x4 module.

To conclude the lab task, I needed a module that would instantiate the bitCounter and seg7 display modules while also wiring up physical switches and displays to the actual hardware design. This module does not hold any logic and rather helps structure the overall program. The blow diagram below outlines the required inputs and outputs for the system. The module receives a 50 MHz clock, keys to trigger a reset on the system, and switches to input data and start the algorithm. The address of the found data is displayed on HEX0 and HEX1 and the result (found or not found) is shown on the LEDs. In addition, the code for this module is visible in Appendix 2E.

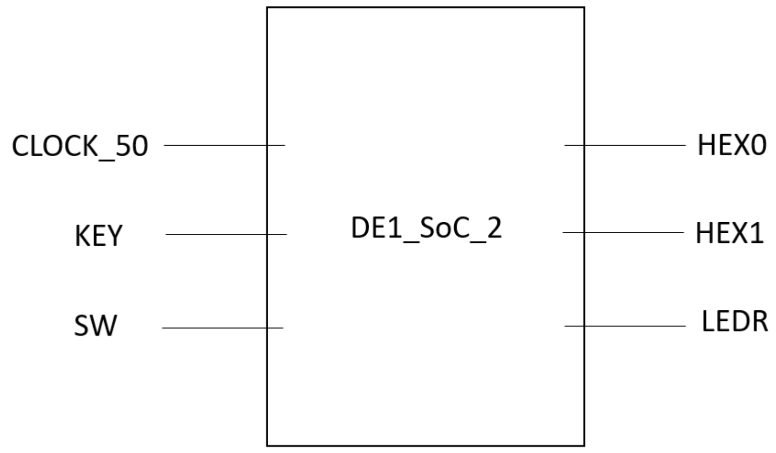


Figure 12. DE1_SoC_Task2 Diagram. Figure to show the necessary inputs and outputs to use the DE1_ SoC_2 module.

Results

Task One - Demonstration Video: https://youtu.be/RKFd2QjFNIw

Module: shiftReg8

The waveform below (Figure 13) shows the successful testing of the shiftReg8 module. In this testbench we test on the input data 10011001. This was my selected test input because it covers a wide range of conditions. It has a leading and trailing 1, which is important for timing issues as it is possible that signals become out of sync and get cut-off early which could lead us to missing out on a 1. It also contains consecutive 1's, which is another unique testing condition that our system must handle. In the simulation, you can see the register data (in the container column) get continuously shifted out on each rising right signal. In addition, with each right signal our size drops by one. Lastly, when our right signal is high when a one is in the rightmost register, the out signal goes high.

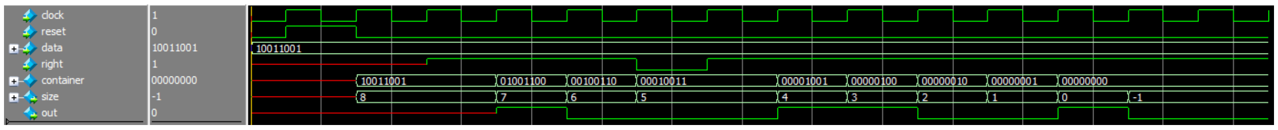


Figure 13. Waveform generated by Appendix 1C that demonstrates a successful test of the shiftReg8 module.

Module: counter4

This waveform below (Figure 14) shows the successful testing of the counter4 module. We began by resetting the counter to zero. Then we keep incrementing the system (with add) so that our count keeps increasing by one. It's also important to note that there is a pause in add signal to ensure that the module doesn't continue adding when we tell it to stop.

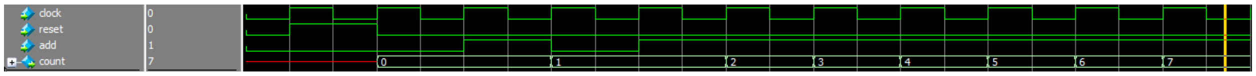


Figure 14. Waveform generated by Appendix 1E that demonstrates a successful test of the counter4 module.

Module: bitCounter

Figure 15 demonstrates a successful test on our bitCounter module. This module ties together the previous two testbenches and adds state logic and transitioning. You can see that when the module is sent to the s2 state (by the start signal), the shift registers and counters become active. Data gets shifted out into the add row which then increments the result section. When the size of the shift register falls past 0, we then transition into the s3 state, where the done signal is held high, and the result data is fixed. Then, the start signal is turned off and the system is sent to the s1 state. Here, the counter is reset and the cycle is ready to be repeated.

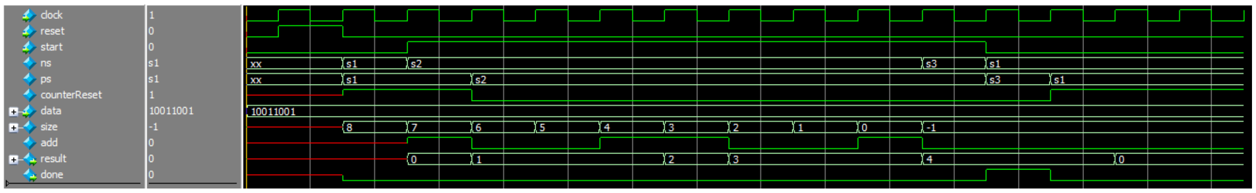


Figure 15. Waveform generated by Appendix 1F that demonstrates a successful test of the bitCounter module.

Module: seg7

The waveform below (Figure 16) shows the successful testing of the seg7 module. This shows how the lights of a given hex display lights up to create the image of a hexadecimal number from 0 to F. Note that a 0 means the led is on and a 1 means the LED is off.

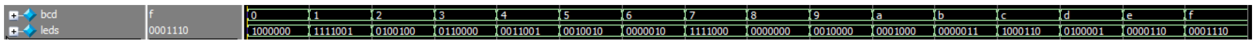


Figure 16. Waveform generated by Appendix 1H that demonstrates a successful test of the seg7 module.

Module: DE1_SoC_Task1

The waveform in Figure 17 confirms the correct implementation of our top-level module. We begin by resetting the entire system (with key0) and entering 10011001 across our switches to serve as our input data. Then, we turn on the bit counting algorithm by turning on sw9. The algorithm runs and hex0 begins updating as the number of located 1's increases. Finally, the algorithm finishes, and our done LED (LEDR9) turns on. At this point, the hex is displaying our final output value, a four (this is also shown in the result row). Next, we return the system to its starting point by turning sw9 back to low – where the result and hex display return to 0.

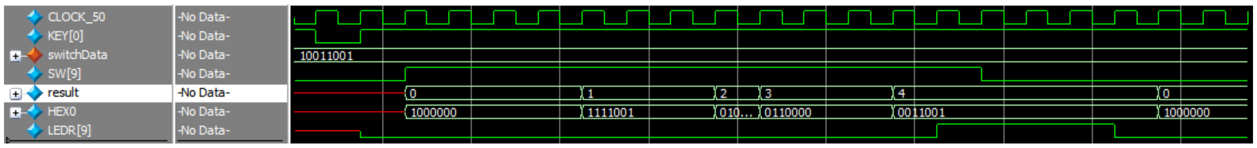


Figure 17. Waveform generated by Appendix 1J that demonstrates a successful test of the DE1_SoC_Task1 module.

System Overview:

Shown below (Figure 18) is an overview of the structure of my system. DE1_SoC is responsible for receiving the physical input and output ports from our board and distributing them into their respective modules. The bitCounter module functions as the temporary data storage unit that also will perform the bit counting algorithm. The output of this module/algorithm is sent to the seg7 display to generate a hexadecimal number on HEX display 0. The user's interaction with the system structure occurs through the switches (sw7-0 for input data and sw9 for start condition). Lastly, whenever the algorithm is finished, LED9 is turned on.

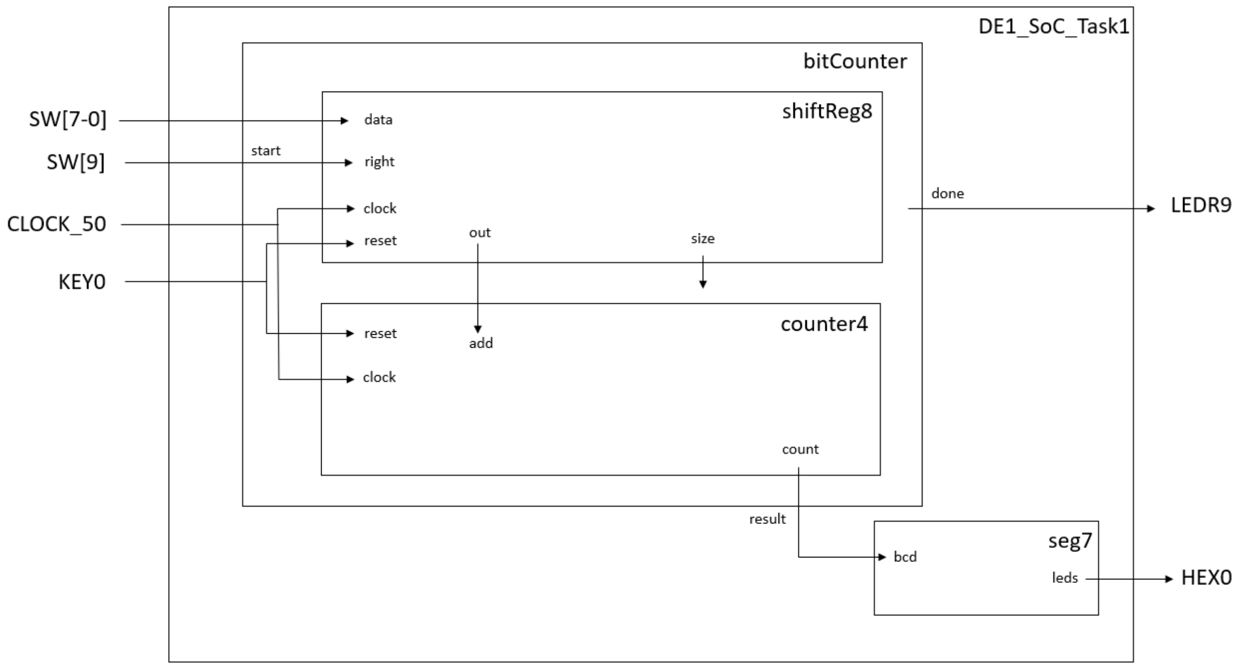


Figure 18. Structural block diagram of the overall system used to implement the bit counter on the DE1_SoC board.

This design fulfills the requirements expected in the lab4 task 1 spec because it successfully implements a bit counter capable of counting the number of high bits in a given 8 bit input. Additionally, this algorithm is implemented using an 8-bit shift register, a counter, and follows the required ASMD diagram shown in Figure 5.

Task Two - Demonstration Video: https://youtu.be/w6Z7fbJeAW8

Module: binary_search

Our RAM is initialized so that the value at every address is twice the value of the address. For example, the value stored at address 18 would be 36. For the binary_search module, I tested 8 different scenarios in order. These scenarios are:

1. Large existing A
2. Small existing A
3. Medium existing A, smaller than middle value
4. Medium existing A, larger than middle value
5. Large nonexistent A
6. Small nonexistent A
7. Medium nonexistent A, smaller than middle value
8. Medium nonexistent A, larger than middle value

As can be seen from the simulation in Figure 19, the module properly outputs true to f when the value is found and true to n_f when the value is not found. The module also outputs the proper address when a number is found and the closest address when the number is not found. The address output stays active even when the number is not found. This is because I did not do any address handling in this module, since it will be easier to enable the hex displays with the f signal in the top-level module.

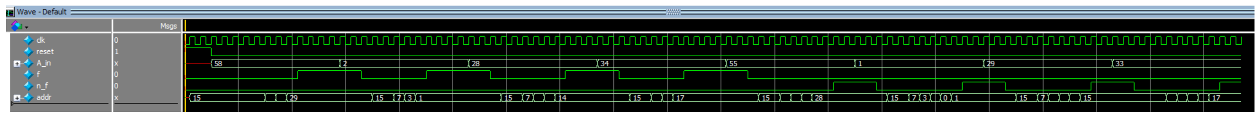


Figure 19. Waveform generated by Appendix 2B that demonstrates a successful test of the binary_search module.

Module: ram32x8

Shown below (Figure 20) is a waveform that tests and verifies the successful implementation of the ram32x8.sv module. This ram is preloaded with a mif file to fill it with default data, so each address holds the data outlined in Figure 21 below. We begin the test by reading from address 1 to see that a 2 is correctly read out. Next, we read from the third address to see a 6. Finally, we read from the 31st address to see a 62.

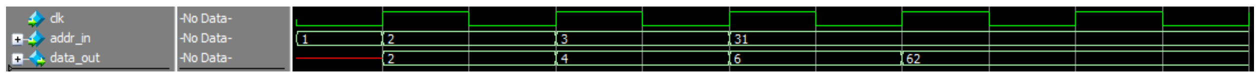
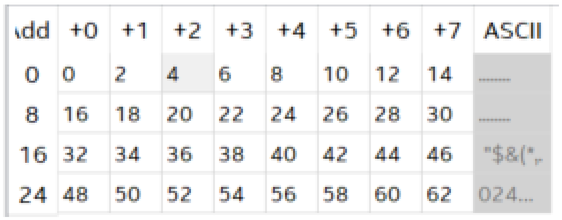


Figure 20. Waveform generated by Appendix 2D that demonstrates a successful test of the ram32x8 module.



dd	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	0	2	4	6	8	10	12	14	....
8	16	18	20	22	24	26	28	30	....
16	32	34	36	38	40	42	44	46	"$&(*,
24	48	50	52	54	56	58	60	62	024...

Figure 21. The mif file that is uploaded into the RAM to give each address its initial data.

Module: DE1_SoC_2

For the top-level module, I used the exact same scenarios from the binary_search module testbench to verify the validity of the interconnections between modules. As can be seen from the waveform above, the module properly outputs found and not found signals, displays the address of the data to the hex displays when the number is found, and turns off the hex displays when the data is not found.

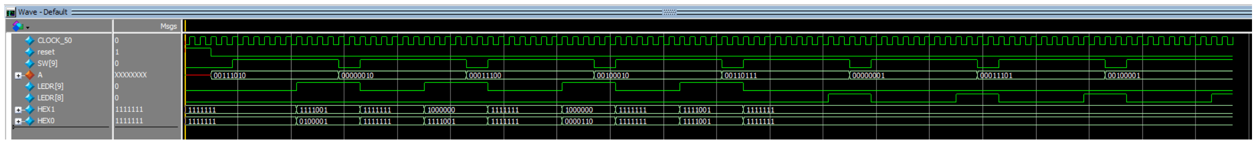


Figure 22. Waveform generated by Appendix 2F that demonstrates a successful test of the DE1_SoC_2 module.

System Overview:

Shown below (Figure 23) is an overview of the structure of my system. DE1_SoC_2 is responsible for receiving the physical input and output ports from our board and distributing them into their respective modules. The ram32x8 module is initialized with the “my_array.mif” file and holds the data to be searched by the binary_search module. The binary_search module implements the binary search algorithm, outputs to LEDR[9] when the data on SW[7:0] is found, and outputs to LEDR[8] when the data is not found. When f is active, the hex displays are turned on, and the address of the data is displayed in hexadecimal.

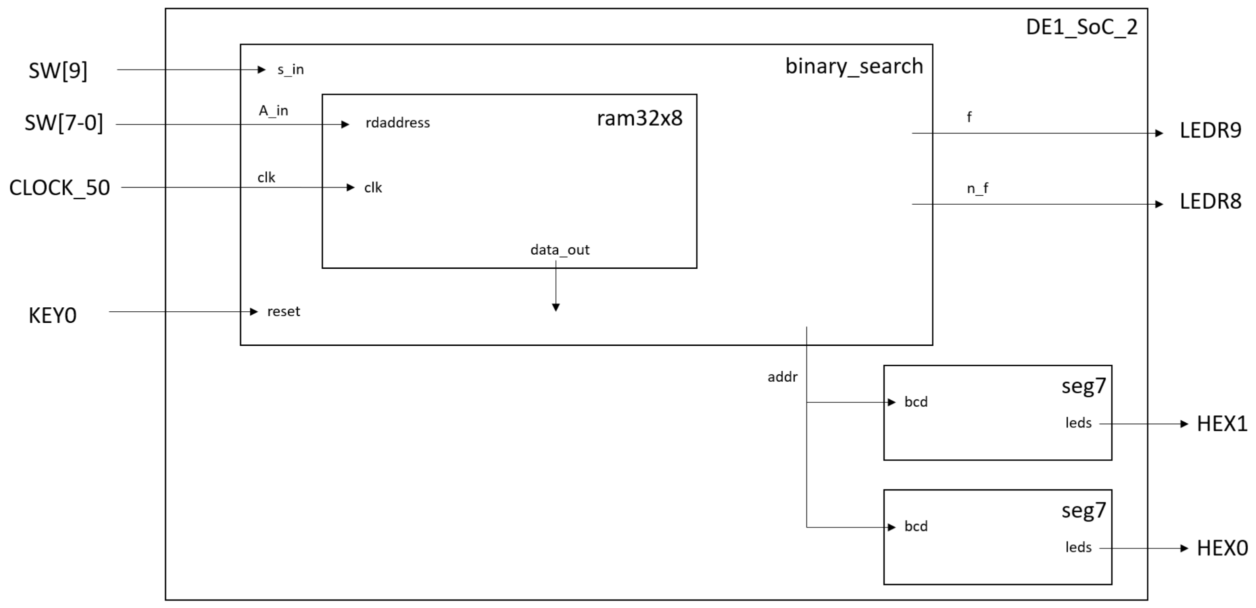
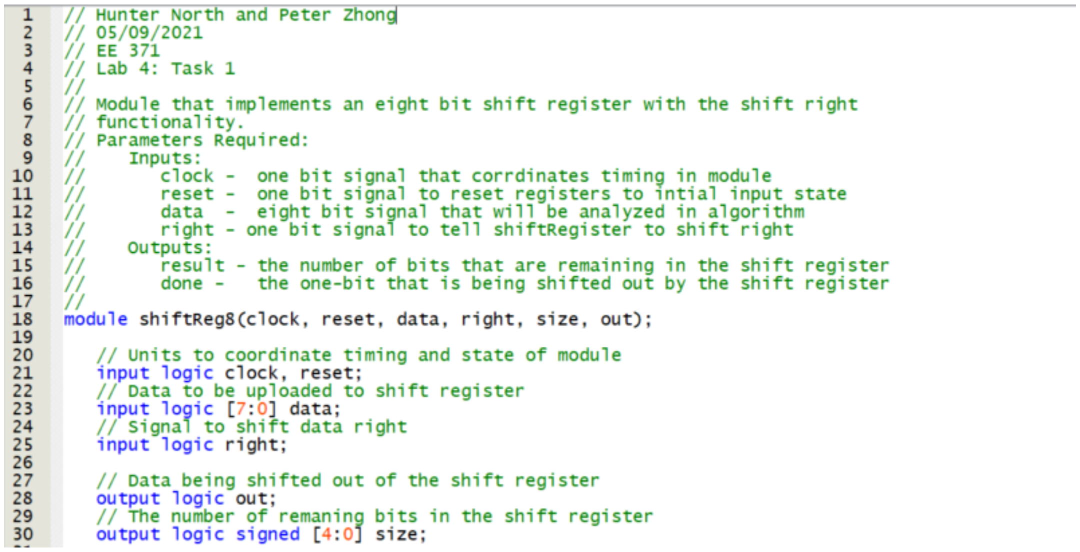


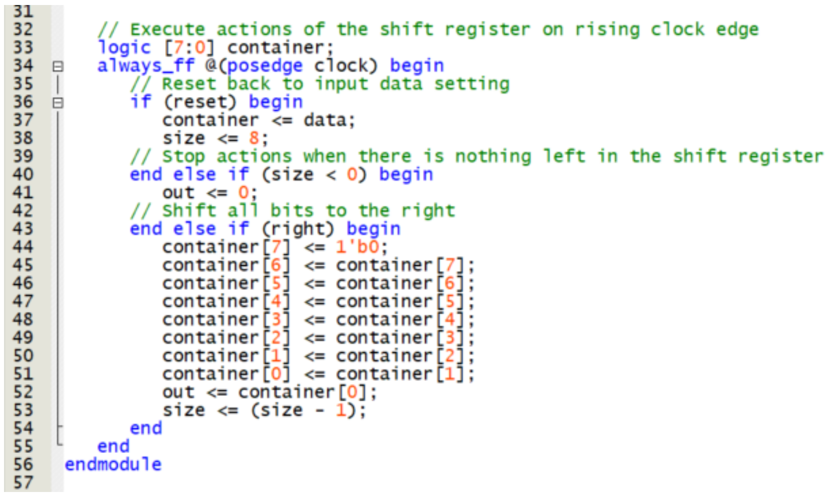
Figure 23. Structural block diagram of the overall system used to implement the bit counter on the DE1_SoC board.

Appendix

1.A – shiftReg8

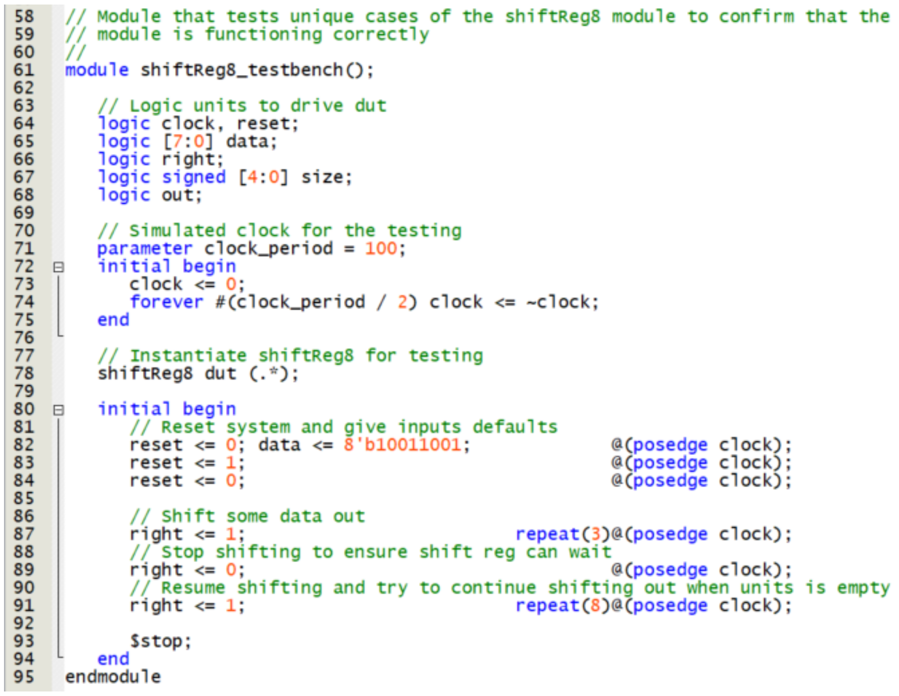


```
// Hunter North and Peter Zhong
// 05/09/2021
// EE 371
// Lab 4: Task 1
//
// Module that implements an eight bit shift register with the shift right
// functionality.
// Parameters Required:
//    Inputs:
//       clock -  one bit signal that corrdinates timing in module
//       reset -  one bit signal to reset registers to intial input state
//       data  -  eight bit signal that will be analyzed in algorithm
//       right - one bit signal to tell shiftRegister to shift right
//    Outputs:
//       result - the number of bits that are remaining in the shift register
//       done -   the one-bit that is being shifted out by the shift register
//
module shiftReg8(clock, reset, data, right, size, out);

   // Units to coordinate timing and state of module
   input logic clock, reset;
   // Data to be uploaded to shift register
   input logic [7:0] data;
   // Signal to shift data right
   input logic right;

   // Data being shifted out of the shift register
   output logic out;
   // The number of remaning bits in the shift register
   output logic signed [4:0] size;
```



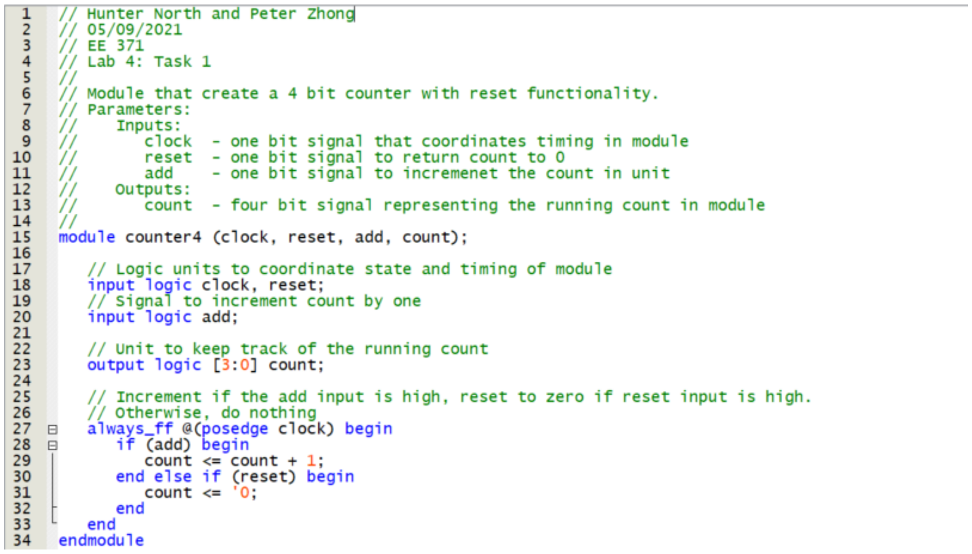
```

   // Execute actions of the shift register on rising clock edge
   logic [7:0] container;
   always_ff @(posedge clock) begin
      // Reset back to input data setting
      if (reset) begin
         container <= data;
         size <= 8;
      // Stop actions when there is nothing left in the shift register
      end else if (size < 0) begin
         out <= 0;
      // Shift all bits to the right
      end else if (right) begin
         container[7] <= 1'b0;
         container[6] <= container[7];
         container[5] <= container[6];
         container[4] <= container[5];
         container[3] <= container[4];
         container[2] <= container[3];
         container[1] <= container[2];
         container[0] <= container[1];
         out <= container[0];
         size <= (size - 1);
      end
   end
endmodule

```

1.B – shiftReg8_testbench



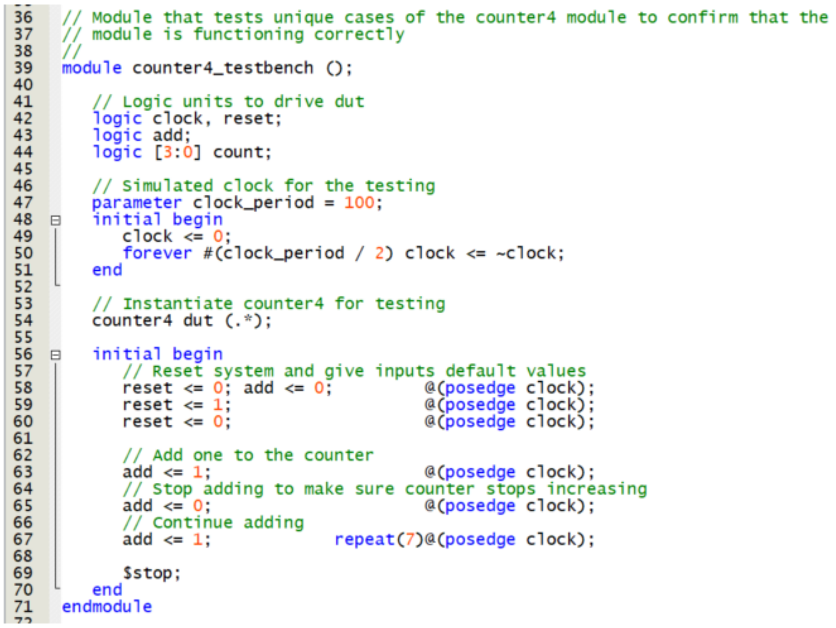
```
// Module that tests unique cases of the shiftReg8 module to confirm that the
// module is functioning correctly
//
module shiftReg8_testbench();

   // Logic units to drive dut
   logic clock, reset;
   logic [7:0] data;
   logic right;
   logic signed [4:0] size;
   logic out;

   // Simulated clock for the testing
   parameter clock_period = 100;
   initial begin
      clock <= 0;
      forever #(clock_period / 2) clock <= ~clock;
   end

   // Instantiate shiftReg8 for testing
   shiftReg8 dut (.*);

   initial begin
      // Reset system and give inputs defaults
      reset <= 0; data <= 8'b10011001;             @(posedge clock);
      reset <= 1;                                  @(posedge clock);
      reset <= 0;                                  @(posedge clock);

      // Shift some data out
      right <= 1;                         repeat(3)@(posedge clock);
      // Stop shifting to ensure shift reg can wait
      right <= 0;                                  @(posedge clock);
      // Resume shifting and try to continue shifting out when units is empty
      right <= 1;                         repeat(8)@(posedge clock);

      $stop;
   end
endmodule
```

1.C – counter4



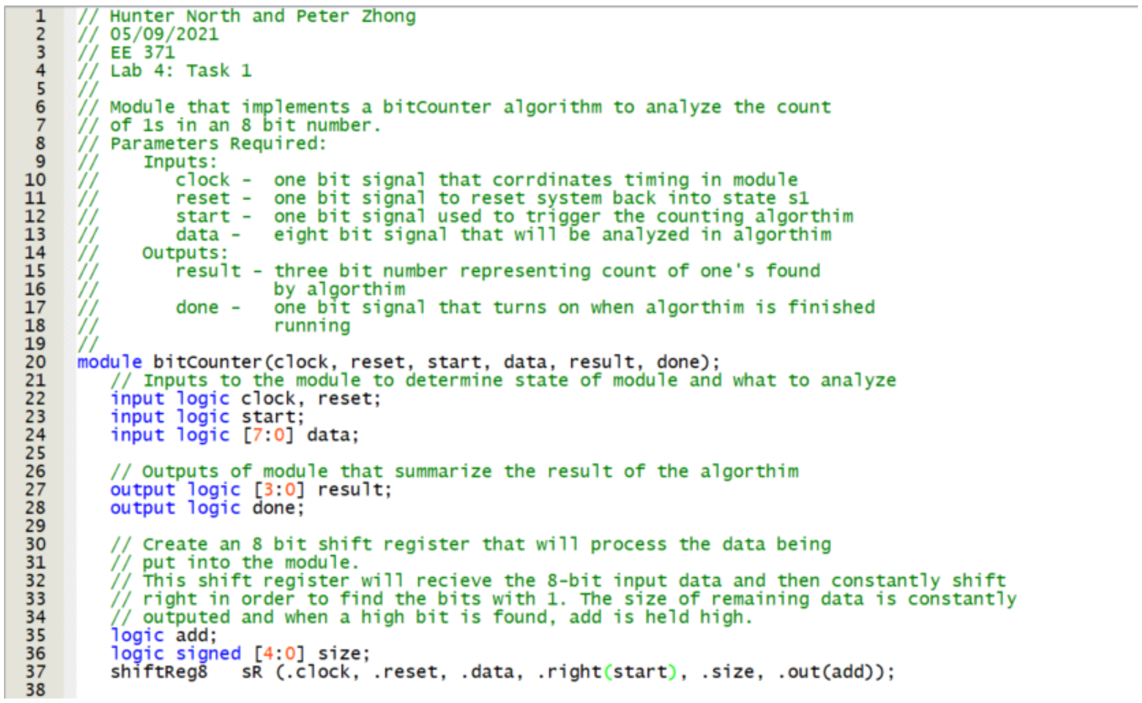
```
// Hunter North and Peter Zhong
// 05/09/2021
// EE 371
// Lab 4: Task 1
//
// Module that create a 4 bit counter with reset functionality.
// Parameters:
//    Inputs:
//       clock  - one bit signal that coordinates timing in module
//       reset  - one bit signal to return count to 0
//       add    - one bit signal to incremenet the count in unit
//    Outputs:
//       count  - four bit signal representing the running count in module
//
module counter4 (clock, reset, add, count);

   // Logic units to coordinate state and timing of module
   input logic clock, reset;
   // Signal to increment count by one
   input logic add;

   // Unit to keep track of the running count
   output logic [3:0] count;

   // Increment if the add input is high, reset to zero if reset input is high.
   // Otherwise, do nothing
   always_ff @(posedge clock) begin
      if (add) begin
         count <= count + 1;
      end else if (reset) begin
         count <= '0;
      end
   end
endmodule
```

1.D – counter4_testbench

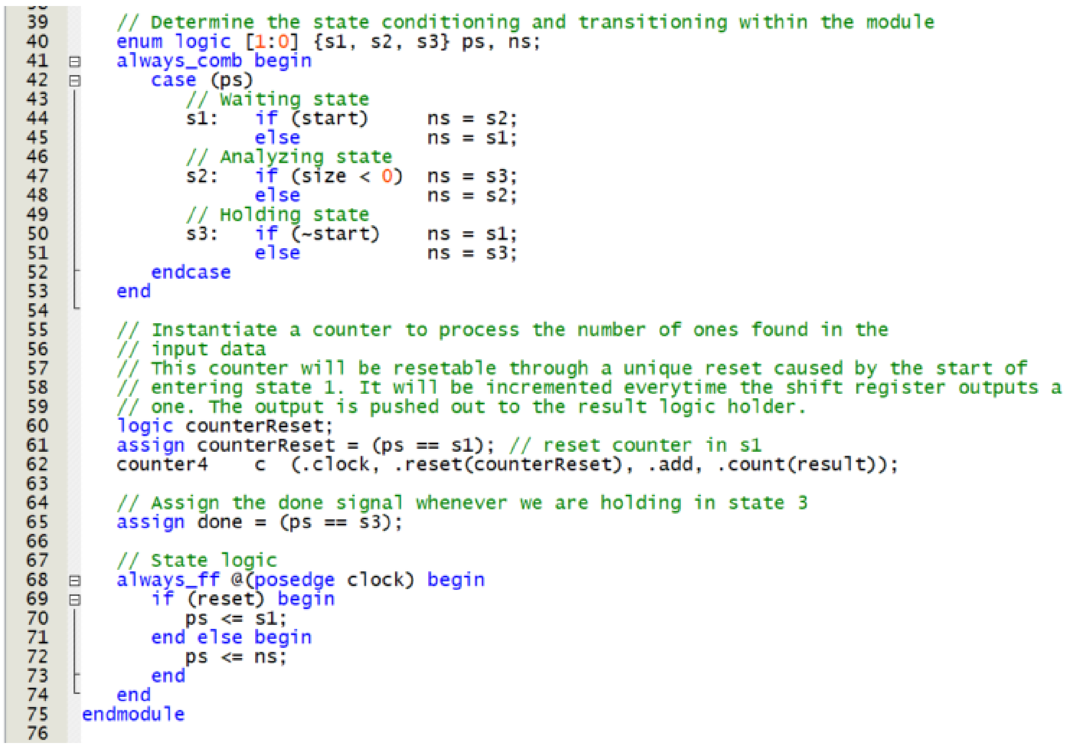


```
// Module that tests unique cases of the counter4 module to confirm that the
// module is functioning correctly
//
module counter4_testbench ();

   // Logic units to drive dut
   logic clock, reset;
   logic add;
   logic [3:0] count;

   // Simulated clock for the testing
   parameter clock_period = 100;
   initial begin
      clock <= 0;
      forever #(clock_period / 2) clock <= ~clock;
   end

   // Instantiate counter4 for testing
   counter4 dut (.*);

   initial begin
      // Reset system and give inputs default values
      reset <= 0; add <= 0;         @(posedge clock);
      reset <= 1;                   @(posedge clock);
      reset <= 0;                   @(posedge clock);

      // Add one to the counter
      add <= 1;                     @(posedge clock);
      // Stop adding to make sure counter stops increasing
      add <= 0;                     @(posedge clock);
      // Continue adding
      add <= 1;            repeat(7)@(posedge clock);

      $stop;
   end
endmodule
```

1.E – bitCounter

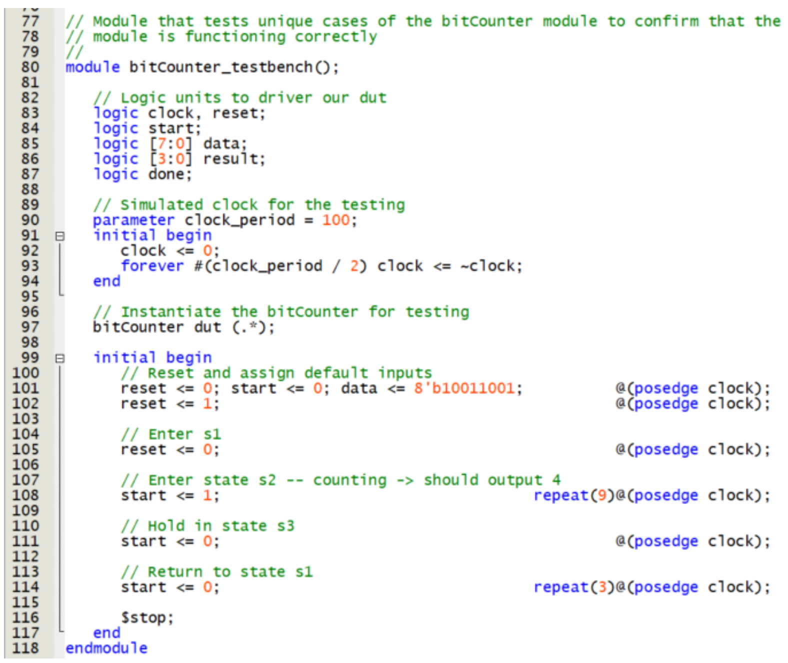


```
// Hunter North and Peter Zhong
// 05/09/2021
// EE 371
// Lab 4: Task 1
//
// Module that implements a bitCounter algorithm to analyze the count
// of 1s in an 8 bit number.
// Parameters Required:
//    Inputs:
//       clock -  one bit signal that corrdinates timing in module
//       reset -  one bit signal to reset system back into state s1
//       start -  one bit signal used to trigger the counting algorthim
//       data -   eight bit signal that will be analyzed in algorthim
//    Outputs:
//       result - three bit number representing count of one's found
//                by algorthim
//       done -   one bit signal that turns on when algorthim is finished
//                running
//
module bitCounter(clock, reset, start, data, result, done);
   // Inputs to the module to determine state of module and what to analyze
   input logic clock, reset;
   input logic start;
   input logic [7:0] data;

   // Outputs of module that summarize the result of the algorthim
   output logic [3:0] result;
   output logic done;

   // Create an 8 bit shift register that will process the data being
   // put into the module.
   // This shift register will recieve the 8-bit input data and then constantly shift
   // right in order to find the bits with 1. The size of remaining data is constantly
   // outputed and when a high bit is found, add is held high.
   logic add;
   logic signed [4:0] size;
   shiftReg8   sR (.clock, .reset, .data, .right(start), .size, .out(add));

```



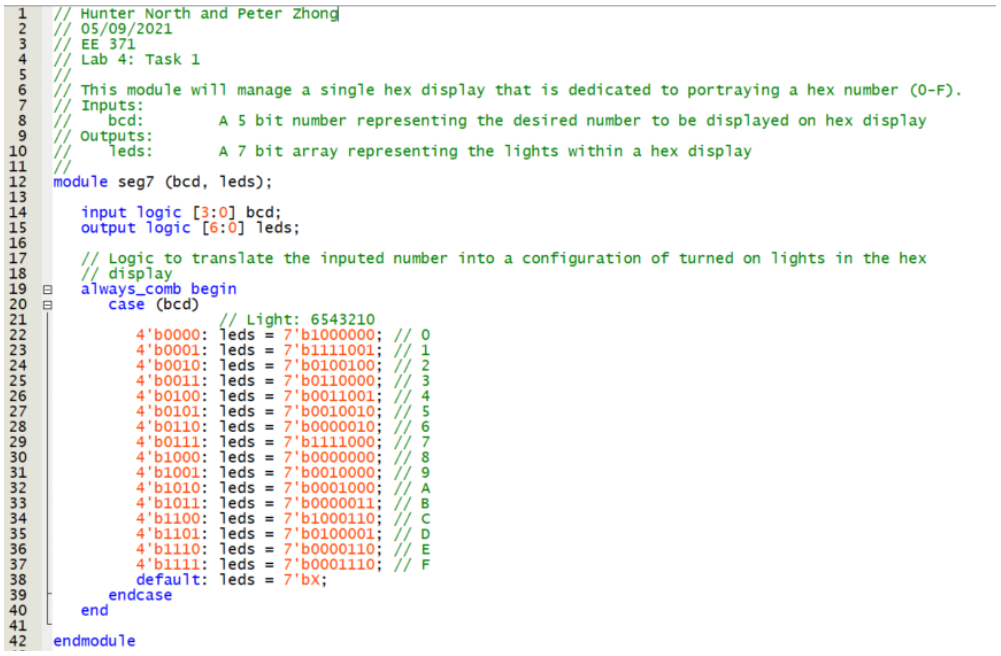
```
   // Determine the state conditioning and transitioning within the module
   enum logic [1:0] {s1, s2, s3} ps, ns;
   always_comb begin
      case (ps)
         // Waiting state
         s1:   if (start)     ns = s2;
               else           ns = s1;
         // Analyzing state
         s2:   if (size < 0)  ns = s3;
               else           ns = s2;
         // Holding state
         s3:   if (~start)    ns = s1;
               else           ns = s3;
      endcase
   end

   // Instantiate a counter to process the number of ones found in the
   // input data
   // This counter will be resetable through a unique reset caused by the start of
   // entering state 1. It will be incremented everytime the shift register outputs a
   // one. The output is pushed out to the result logic holder.
   logic counterReset;
   assign counterReset = (ps == s1); // reset counter in s1
   counter4    c  (.clock, .reset(counterReset), .add, .count(result));

   // Assign the done signal whenever we are holding in state 3
   assign done = (ps == s3);

   // State logic
   always_ff @(posedge clock) begin
      if (reset) begin
         ps <= s1;
      end else begin
         ps <= ns;
      end
   end
endmodule
```

1.F – bitCounter_testbench



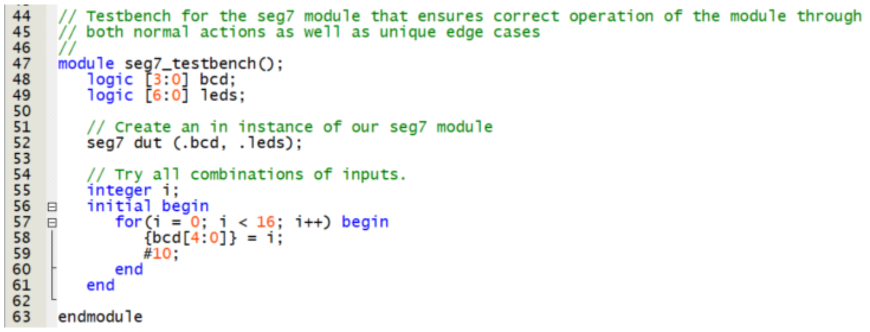
```
// Module that tests unique cases of the bitCounter module to confirm that the
// module is functioning correctly
//
module bitCounter_testbench();

   // Logic units to driver our dut
   logic clock, reset;
   logic start;
   logic [7:0] data;
   logic [3:0] result;
   logic done;

   // Simulated clock for the testing
   parameter clock_period = 100;
   initial begin
      clock <= 0;
      forever #(clock_period / 2) clock <= ~clock;
   end

   // Instantiate the bitCounter for testing
   bitCounter dut (.*);

   initial begin
      // Reset and assign default inputs
      reset <= 0; start <= 0; data <= 8'b10011001;          @(posedge clock);
      reset <= 1;                                           @(posedge clock);

      // Enter s1
      reset <= 0;                                           @(posedge clock);

      // Enter state s2 -- counting -> should output 4
      start <= 1;                                  repeat(9)@(posedge clock);

      // Hold in state s3
      start <= 0;                                           @(posedge clock);

      // Return to state s1
      start <= 0;                                  repeat(3)@(posedge clock);

      $stop;
   end
endmodule
```

1.G – seg7



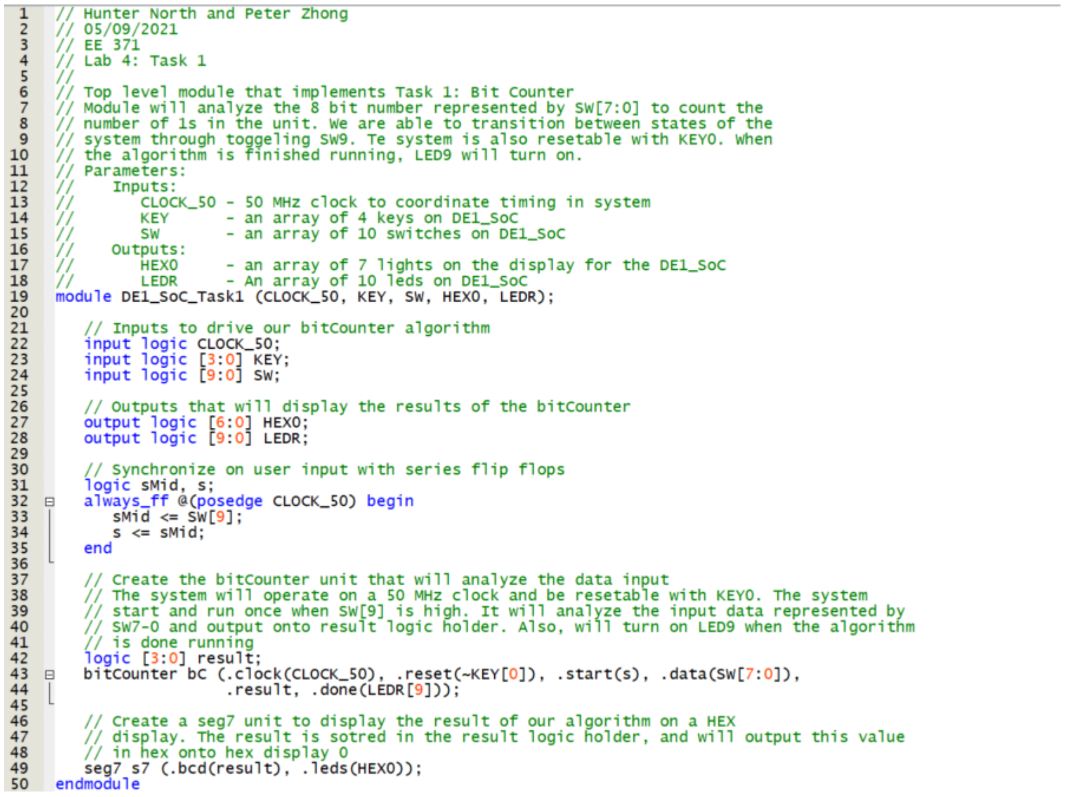
```
// Hunter North and Peter Zhong
// 05/09/2021
// EE 371
// Lab 4: Task 1
//
// This module will manage a single hex display that is dedicated to portraying a hex number (0-F).
// Inputs:
//    bcd:        A 5 bit number representing the desired number to be displayed on hex display
// Outputs:
//    leds:       A 7 bit array representing the lights within a hex display
//
module seg7 (bcd, leds);

   input logic [3:0] bcd;
   output logic [6:0] leds;

   // Logic to translate the inputed number into a configuration of turned on lights in the hex
   // display
   always_comb begin
      case (bcd)
                  // Light: 6543210
         4'b0000: leds = 7'b1000000; // 0
         4'b0001: leds = 7'b1111001; // 1
         4'b0010: leds = 7'b0100100; // 2
         4'b0011: leds = 7'b0110000; // 3
         4'b0100: leds = 7'b0011001; // 4
         4'b0101: leds = 7'b0010010; // 5
         4'b0110: leds = 7'b0000010; // 6
         4'b0111: leds = 7'b1111000; // 7
         4'b1000: leds = 7'b0000000; // 8
         4'b1001: leds = 7'b0010000; // 9
         4'b1010: leds = 7'b0001000; // A
         4'b1011: leds = 7'b0000011; // B
         4'b1100: leds = 7'b1000110; // C
         4'b1101: leds = 7'b0100001; // D
         4'b1110: leds = 7'b0000110; // E
         4'b1111: leds = 7'b0001110; // F
         default: leds = 7'bX;
      endcase
   end

endmodule
```

1.H – seg7_testbench



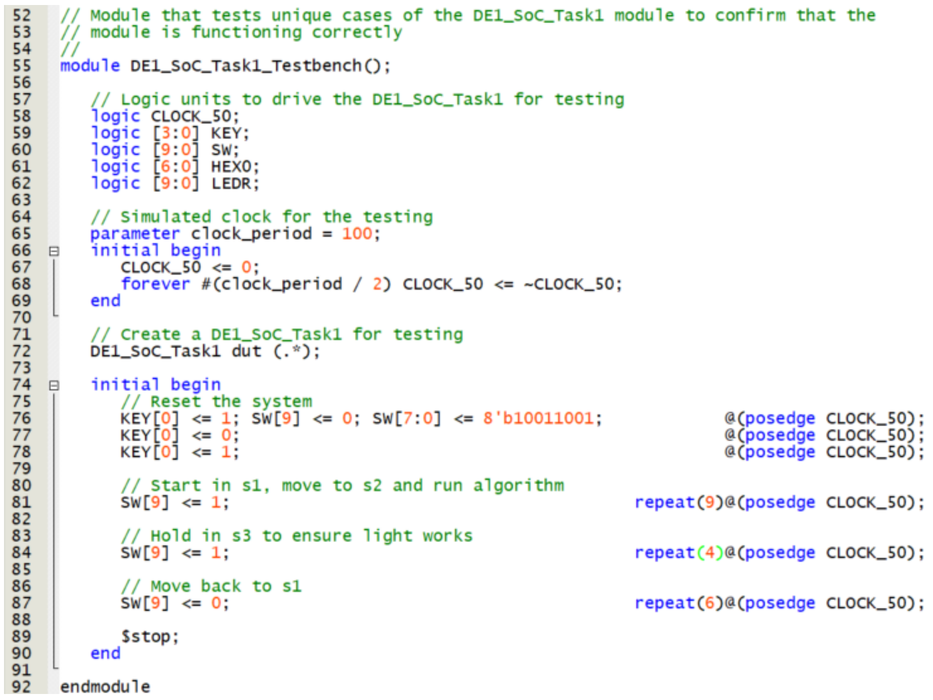
```
// Testbench for the seg7 module that ensures correct operation of the module through
// both normal actions as well as unique edge cases
//
module seg7_testbench();
   logic [3:0] bcd;
   logic [6:0] leds;

   // Create an in instance of our seg7 module
   seg7 dut (.bcd, .leds);

   // Try all combinations of inputs.
   integer i;
   initial begin
      for(i = 0; i < 16; i++) begin
         {bcd[4:0]} = i;
         #10;
      end
   end

endmodule
```

1.I – DE1_SoC_Task1



```
// Hunter North and Peter Zhong
// 05/09/2021
// EE 371
// Lab 4: Task 1
//
// Top level module that implements Task 1: Bit Counter
// Module will analyze the 8 bit number represented by SW[7:0] to count the
// number of 1s in the unit. We are able to transition between states of the
// system through toggeling SW9. Te system is also resetable with KEY0. When
// the algorithm is finished running, LED9 will turn on.
// Parameters:
//    Inputs:
//       CLOCK_50 - 50 MHz clock to coordinate timing in system
//       KEY      - an array of 4 keys on DE1_SoC
//       SW       - an array of 10 switches on DE1_SoC
//    Outputs:
//       HEX0     - an array of 7 lights on the display for the DE1_SoC
//       LEDR     - An array of 10 leds on DE1_SoC
module DE1_SoC_Task1 (CLOCK_50, KEY, SW, HEX0, LEDR);

   // Inputs to drive our bitCounter algorithm
   input logic CLOCK_50;
   input logic [3:0] KEY;
   input logic [9:0] SW;

   // Outputs that will display the results of the bitCounter
   output logic [6:0] HEX0;
   output logic [9:0] LEDR;

   // Synchronize on user input with series flip flops
   logic sMid, s;
   always_ff @(posedge CLOCK_50) begin
      sMid <= SW[9];
      s <= sMid;
   end

   // Create the bitCounter unit that will analyze the data input
   // The system will operate on a 50 MHz clock and be resetable with KEY0. The system
   // start and run once when SW[9] is high. It will analyze the input data represented by
   // SW7-0 and output onto result logic holder. Also, will turn on LED9 when the algorithm
   // is done running
   logic [3:0] result;
   bitCounter bC (.clock(CLOCK_50), .reset(~KEY[0]), .start(s), .data(SW[7:0]),
                  .result, .done(LEDR[9]));

   // Create a seg7 unit to display the result of our algorithm on a HEX
   // display. The result is sotred in the result logic holder, and will output this value
   // in hex onto hex display 0
   seg7 s7 (.bcd(result), .leds(HEX0));
endmodule
```

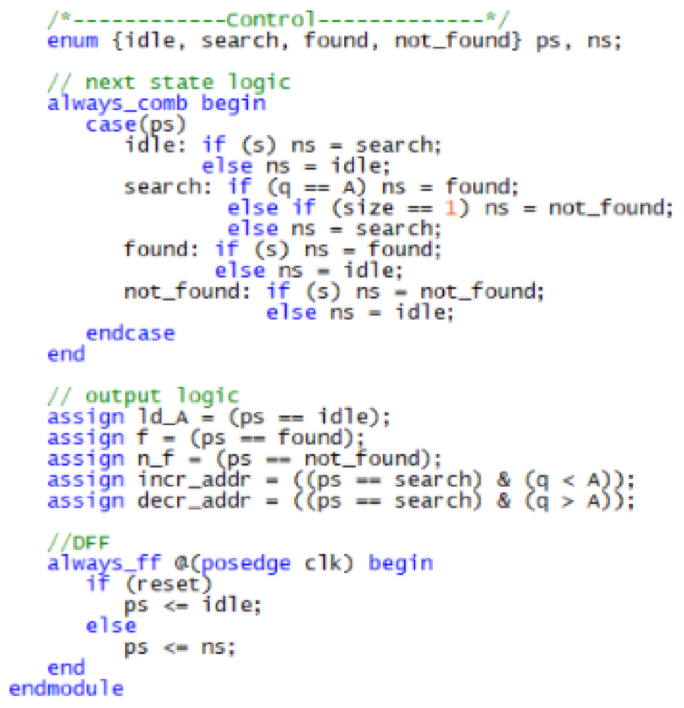
1.J – DE1_SoC_Task1_Testbench



```
// Module that tests unique cases of the DE1_SoC_Task1 module to confirm that the
// module is functioning correctly
//
module DE1_SoC_Task1_Testbench();

   // Logic units to drive the DE1_SoC_Task1 for testing
   logic CLOCK_50;
   logic [3:0] KEY;
   logic [9:0] SW;
   logic [6:0] HEX0;
   logic [9:0] LEDR;

   // Simulated clock for the testing
   parameter clock_period = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period / 2) CLOCK_50 <= ~CLOCK_50;
   end

   // Create a DE1_SoC_Task1 for testing
   DE1_SoC_Task1 dut (.*);

   initial begin
      // Reset the system
      KEY[0] <= 1; SW[9] <= 0; SW[7:0] <= 8'b10011001;            @(posedge CLOCK_50);
      KEY[0] <= 0;                                                 @(posedge CLOCK_50);
      KEY[0] <= 1;                                                 @(posedge CLOCK_50);

      // Start in s1, move to s2 and run algorithm
      SW[9] <= 1;                                         repeat(9)@(posedge CLOCK_50);

      // Hold in s3 to ensure light works
      SW[9] <= 1;                                         repeat(4)@(posedge CLOCK_50);

      // Move back to s1
      SW[9] <= 0;                                         repeat(6)@(posedge CLOCK_50);

      $stop;
   end

endmodule
```

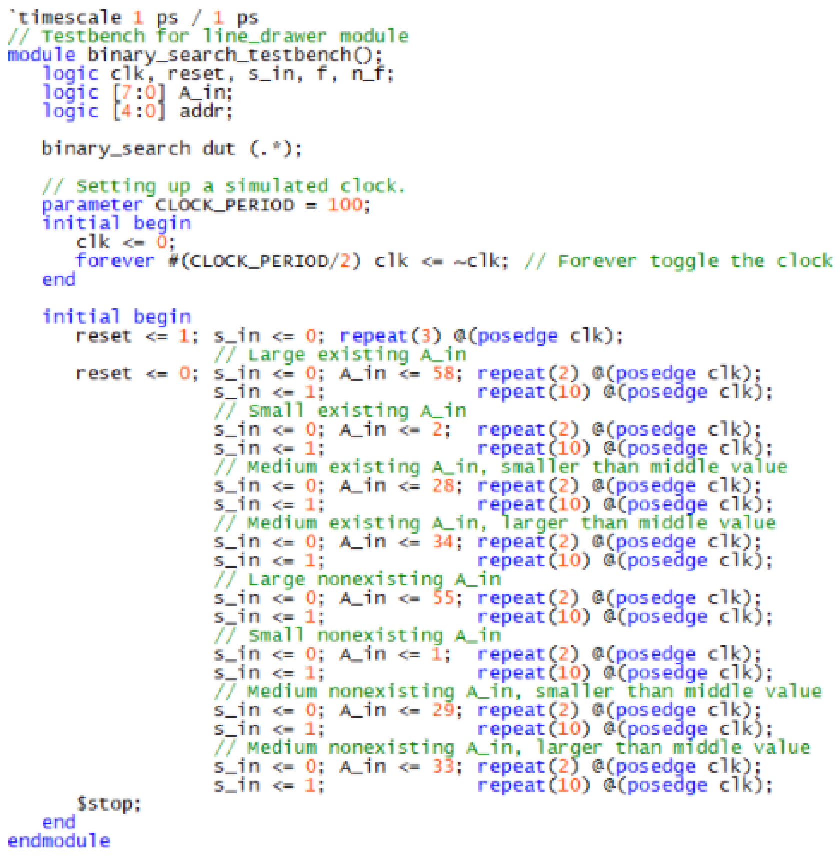
2.A – binary_search

```
// Hunter North and Peter Zhong
// 05/12/2020
// EE 371
// Lab #4. Task 2

// binary_search implements a binary search algorithm on a 32x8 RAM.
// It takes an 8-bit input A_in and start signal s, and outputs f when A_in is in the RAM.
// It outputs n_f when A_in is not in the RAM.

module binary_search(
    input logic clk,
    input logic reset,
    input logic s_in,
    input logic [7:0] A_in,

    output logic f, // found
    output logic n_f, //not found
    output logic [4:0] addr
);

    // synchronizing s_in
    logic s;
    always_ff @(posedge clk) begin
        s <= s_in;
    end

    /*------------Datapath-------------*/
    logic ld_A, incr_addr, decr_addr;
    logic [7:0] A;
    logic [7:0] q;

    // RAM for numbers. No write operations, read only
    // update q at negative edge (~clk) so that q is updated between address update
    // and comparison operation.
    ram32x8 RAM (.clk(~clk), .addr_in(addr), .data_out(q));

    // register for holding A
    always_ff @(posedge clk) begin
        if (reset)
            A <= '0;
        else if (ld_A)
            A <= A_in;
    end

    // register for addr and size
    // left and right for tracking the left and right boundaries
    logic [5:0] size, left, right;
    always_ff @(posedge clk) begin
        if (reset | ld_A) begin
            size <= 32;
            addr <= 15;
            left <= 0;
            right <= 31;
        end
        else if (f)
            addr <= addr;
        // increase left bound, update addr and size accordingly
        else if (incr_addr) begin
            left <= left + (size / 2);
            addr <= (left + (size / 2) + right) / 2;
            size <= right - (left + (size / 2)) + 1;
        end
        // decrease right bound, update addr and size accordingly
        else if (decr_addr) begin
            right <= right - (size / 2);
            addr <= (left + right - (size / 2)) / 2;
            size <= right - (size / 2) - left + 1;
        end
    end
```



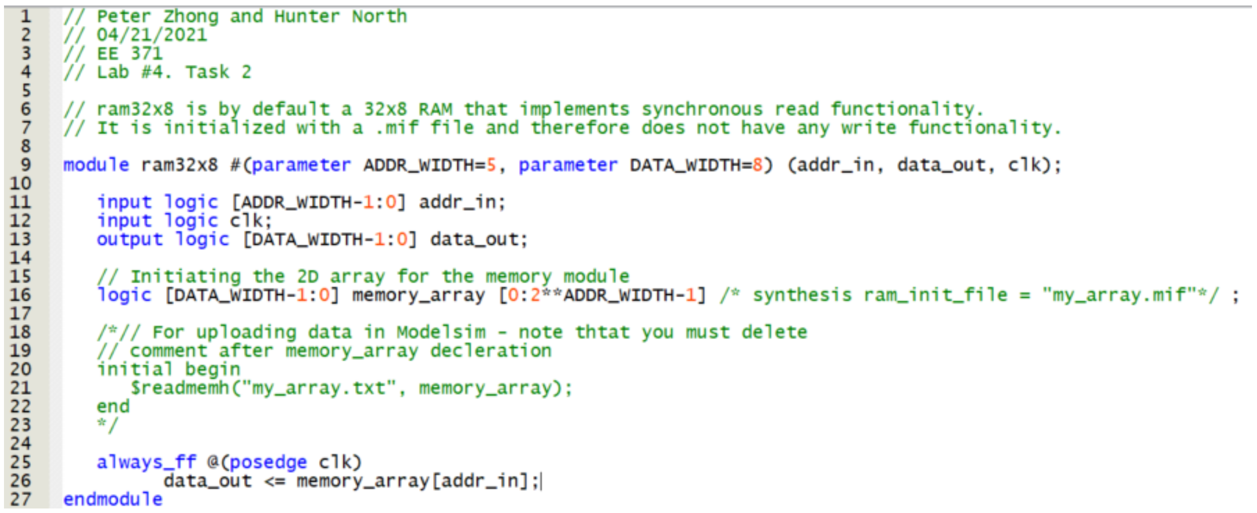
```
  /*------------Control-------------*/
  enum {idle, search, found, not_found} ps, ns;

  // next state logic
  always_comb begin
     case(ps)
        idle: if (s) ns = search;
              else ns = idle;
        search: if (q == A) ns = found;
                else if (size == 1) ns = not_found;
                else ns = search;
        found: if (s) ns = found;
               else ns = idle;
        not_found: if (s) ns = not_found;
                   else ns = idle;
     endcase
  end

  // output logic
  assign ld_A = (ps == idle);
  assign f = (ps == found);
  assign n_f = (ps == not_found);
  assign incr_addr = ((ps == search) & (q < A));
  assign decr_addr = ((ps == search) & (q > A));

  //DFF
  always_ff @(posedge clk) begin
     if (reset)
        ps <= idle;
     else
        ps <= ns;
  end
endmodule
```

2.B -- binary_search_testbench



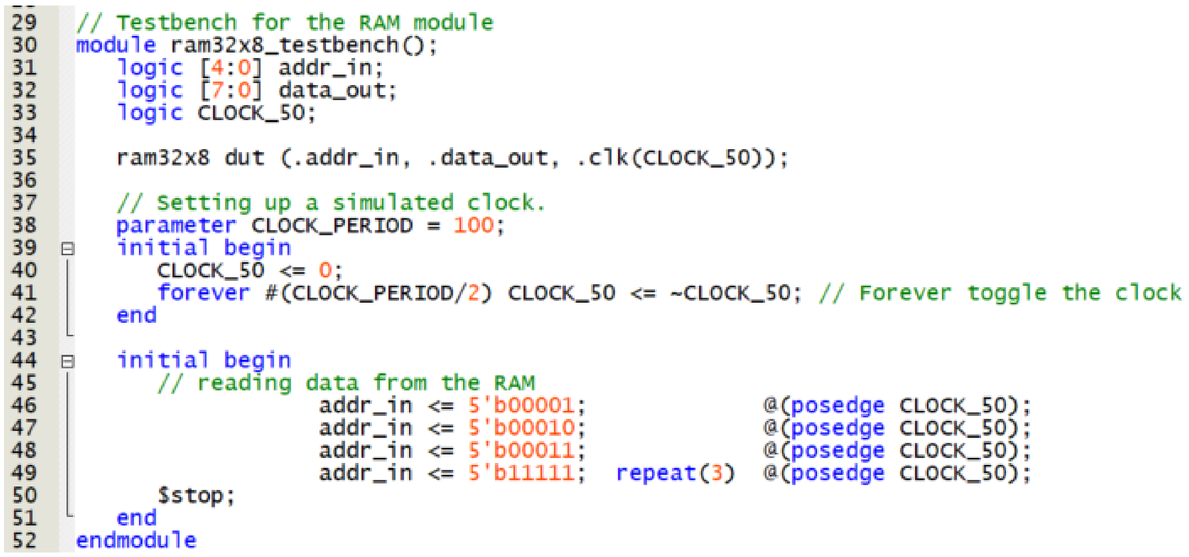
```
`timescale 1 ps / 1 ps
// Testbench for line_drawer module
module binary_search_testbench();
   logic clk, reset, s_in, f, n_f;
   logic [7:0] A_in;
   logic [4:0] addr;

   binary_search dut (.*);

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

   initial begin
      reset <= 1; s_in <= 0; repeat(3) @(posedge clk);
                  // Large existing A_in
      reset <= 0; s_in <= 0; A_in <= 58; repeat(2) @(posedge clk);
                  s_in <= 1;             repeat(10) @(posedge clk);
                  // Small existing A_in
                  s_in <= 0; A_in <= 2;  repeat(2) @(posedge clk);
                  s_in <= 1;             repeat(10) @(posedge clk);
                  // Medium existing A_in, smaller than middle value
                  s_in <= 0; A_in <= 28; repeat(2) @(posedge clk);
                  s_in <= 1;             repeat(10) @(posedge clk);
                  // Medium existing A_in, larger than middle value
                  s_in <= 0; A_in <= 34; repeat(2) @(posedge clk);
                  s_in <= 1;             repeat(10) @(posedge clk);
                  // Large nonexisting A_in
                  s_in <= 0; A_in <= 55; repeat(2) @(posedge clk);
                  s_in <= 1;             repeat(10) @(posedge clk);
                  // Small nonexisting A_in
                  s_in <= 0; A_in <= 1;  repeat(2) @(posedge clk);
                  s_in <= 1;             repeat(10) @(posedge clk);
                  // Medium nonexisting A_in, smaller than middle value
                  s_in <= 0; A_in <= 29; repeat(2) @(posedge clk);
                  s_in <= 1;             repeat(10) @(posedge clk);
                  // Medium nonexisting A_in, larger than middle value
                  s_in <= 0; A_in <= 33; repeat(2) @(posedge clk);
                  s_in <= 1;             repeat(10) @(posedge clk);
      $stop;
   end
endmodule
```

2.C -- ram32x8



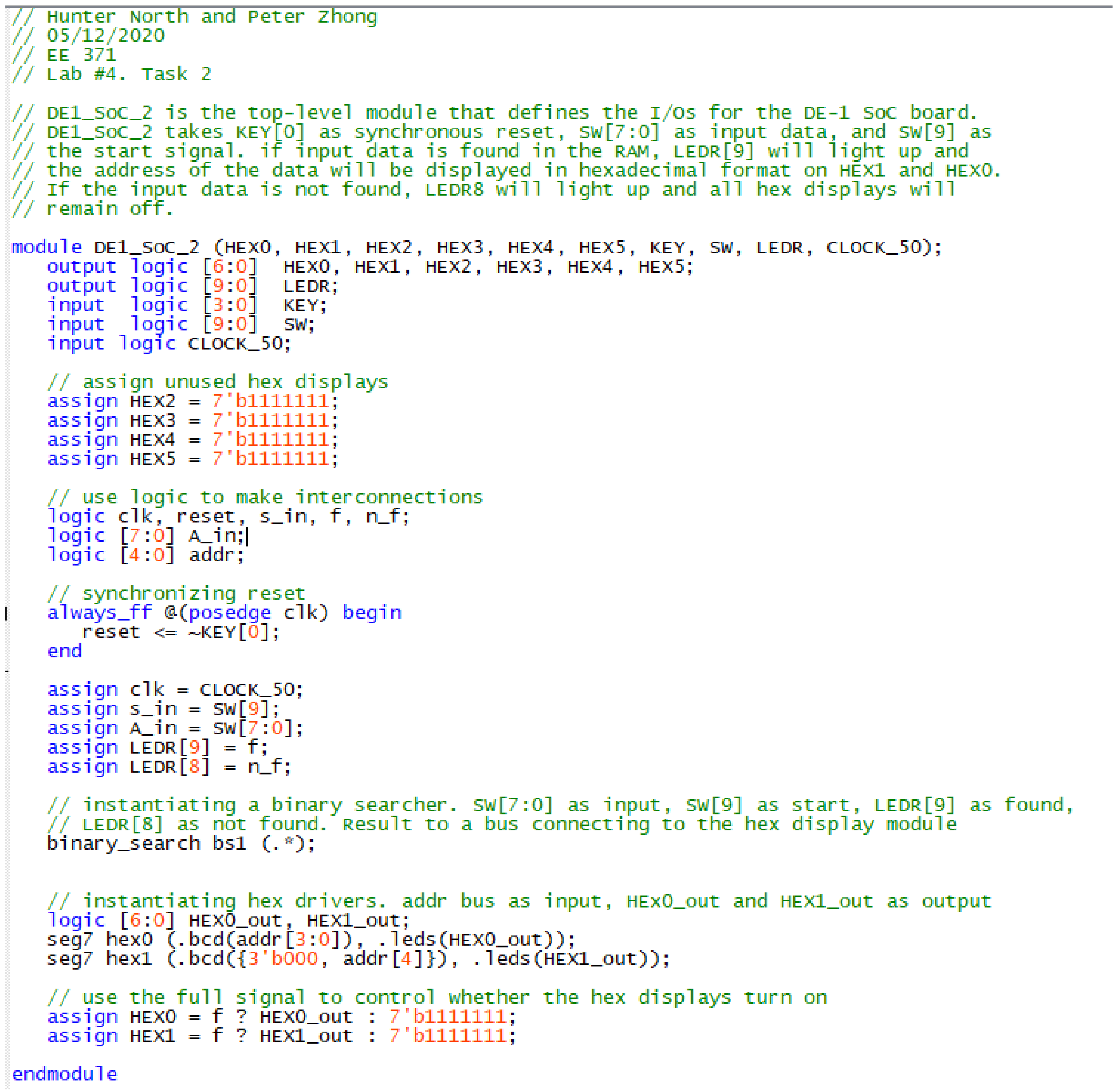
```
// Peter Zhong and Hunter North
// 04/21/2021
// EE 371
// Lab #4. Task 2

// ram32x8 is by default a 32x8 RAM that implements synchronous read functionality.
// It is initialized with a .mif file and therefore does not have any write functionality.

module ram32x8 #(parameter ADDR_WIDTH=5, parameter DATA_WIDTH=8) (addr_in, data_out, clk);

   input logic [ADDR_WIDTH-1:0] addr_in;
   input logic clk;
   output logic [DATA_WIDTH-1:0] data_out;

   // Initiating the 2D array for the memory module
   logic [DATA_WIDTH-1:0] memory_array [0:2**ADDR_WIDTH-1] /* synthesis ram_init_file = "my_array.mif"*/ ;

   /*// For uploading data in Modelsim - note thtat you must delete
   // comment after memory_array decleration
   initial begin
      $readmemh("my_array.txt", memory_array);
   end
   */

   always_ff @(posedge clk)
         data_out <= memory_array[addr_in];
endmodule
```

2.D -- ram32x8_testbench



```
// Testbench for the RAM module
module ram32x8_testbench();
   logic [4:0] addr_in;
   logic [7:0] data_out;
   logic CLOCK_50;

   ram32x8 dut (.addr_in, .data_out, .clk(CLOCK_50));

   // Setting up a simulated clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   initial begin
      // reading data from the RAM
                    addr_in <= 5'b00001;              @(posedge CLOCK_50);
                    addr_in <= 5'b00010;              @(posedge CLOCK_50);
                    addr_in <= 5'b00011;              @(posedge CLOCK_50);
                    addr_in <= 5'b11111;   repeat(3)  @(posedge CLOCK_50);
      $stop;
   end
endmodule
```

2.E - DE1_SoC_2



```
// Hunter North and Peter Zhong
// 05/12/2020
// EE 371
// Lab #4. Task 2

// DE1_SoC_2 is the top-level module that defines the I/Os for the DE-1 SoC board.
// DE1_SoC_2 takes KEY[0] as synchronous reset, SW[7:0] as input data, and SW[9] as
// the start signal. if input data is found in the RAM, LEDR[9] will light up and
// the address of the data will be displayed in hexadecimal format on HEX1 and HEX0.
// If the input data is not found, LEDR8 will light up and all hex displays will
// remain off.

module DE1_SoC_2 (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 unused hex displays
   assign HEX2 = 7'b1111111;
   assign HEX3 = 7'b1111111;
   assign HEX4 = 7'b1111111;
   assign HEX5 = 7'b1111111;

   // use logic to make interconnections
   logic clk, reset, s_in, f, n_f;
   logic [7:0] A_in;
   logic [4:0] addr;

   // synchronizing reset
   always_ff @(posedge clk) begin
      reset <= ~KEY[0];
   end

   assign clk = CLOCK_50;
   assign s_in = SW[9];
   assign A_in = SW[7:0];
   assign LEDR[9] = f;
   assign LEDR[8] = n_f;

   // instantiating a binary searcher. SW[7:0] as input, SW[9] as start, LEDR[9] as found,
   // LEDR[8] as not found. Result to a bus connecting to the hex display module
   binary_search bs1 (.*);


   // instantiating hex drivers. addr bus as input, HEX0_out and HEX1_out as output
   logic [6:0] HEX0_out, HEX1_out;
   seg7 hex0 (.bcd(addr[3:0]), .leds(HEX0_out));
   seg7 hex1 (.bcd({3'b000, addr[4]}), .leds(HEX1_out));

   // use the full signal to control whether the hex displays turn on
   assign HEX0 = f ? HEX0_out : 7'b1111111;
   assign HEX1 = f ? HEX1_out : 7'b1111111;

endmodule
```

2.F --- DE1_SoC2_testbench

```
`timescale 1 ps / 1 ps
// testbench for the top-level module
module DE1_SoC_2_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_2 dut (.*);

   // assigning logics to make the code more readable
   logic reset;
   assign KEY[0] = ~reset;

   logic [7:0] A_in;
   assign SW[7:0] = A_in;

   logic s_in;
   assign SW[9] = s_in;

   // Setting up a simulated clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      CLOCK_50 <= 0;
      forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
   end

   // Trying all combinations of inputs (x and y).
   initial begin
      reset <= 1; s_in <= 0; repeat(3) @(posedge CLOCK_50);
                  // Large existing A_in
      reset <= 0; s_in <= 0; A_in <= 58; repeat(2) @(posedge CLOCK_50);
                  s_in <= 1;             repeat(10) @(posedge CLOCK_50);
                  // Small existing A_in
                  s_in <= 0; A_in <= 2;  repeat(2) @(posedge CLOCK_50);
                  s_in <= 1;             repeat(10) @(posedge CLOCK_50);
                  // Medium existing A_in, smaller than middle value
                  s_in <= 0; A_in <= 28; repeat(2) @(posedge CLOCK_50);
                  s_in <= 1;             repeat(10) @(posedge CLOCK_50);
                  // Medium existing A_in, larger than middle value
                  s_in <= 0; A_in <= 34; repeat(2) @(posedge CLOCK_50);
                  s_in <= 1;             repeat(10) @(posedge CLOCK_50);
                  // Large nonexisting A_in
                  s_in <= 0; A_in <= 55; repeat(2) @(posedge CLOCK_50);
                  s_in <= 1;             repeat(10) @(posedge CLOCK_50);
                  // Small nonexisting A_in
                  s_in <= 0; A_in <= 1;  repeat(2) @(posedge CLOCK_50);
                  s_in <= 1;             repeat(10) @(posedge CLOCK_50);
                  // Medium nonexisting A_in, smaller than middle value
                  s_in <= 0; A_in <= 29; repeat(2) @(posedge CLOCK_50);
                  s_in <= 1;             repeat(10) @(posedge CLOCK_50);
                  // Medium nonexisting A_in, larger than middle value
                  s_in <= 0; A_in <= 33; repeat(2) @(posedge CLOCK_50);
                  s_in <= 1;             repeat(10) @(posedge CLOCK_50);
      $stop;
   end
endmodule
```