Aidan Lee, Aarin Wen
EE 371
November 13, 2024
Lab 4 Report

Procedure:
This lab focused on implementing algorithms as hardware circuits by utilizing Algorithmic State Machine and Datapath (ASMD) charts. ASMD charts such as Figure 1.1 allow the specification of digital systems such that specific hardware implementation can be communicated and thus implemented. Both tasks also focused on breaking down systems into its Datapath and Controller components. Task 1 comprised of creating an 8-bit bit-counting module based on a given ASMD chart that counted how many 1's were in a given input; the implementation was then to be tested and demonstrated on a DE1_SoC board. Task 2 comprised of creating a binary search algorithm through utilization of a Memory Initialization File (MIF) by creating an ASMD chart for the binary search algorithm. Both tasks were then to be implemented and demonstrated on a DE1_SoC board.

Task 1: Bit-Counter
The goal of this task was to implement the bit-counting circuit in System Verilog using the ASMD chart shown below in Figure 1.1.

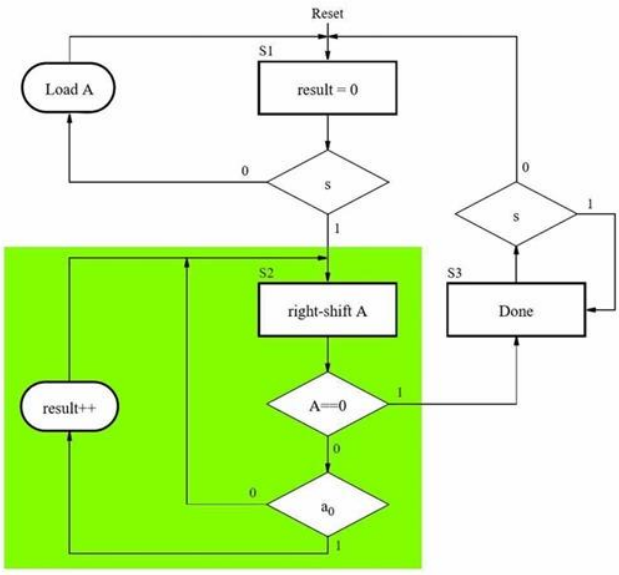

Figure 1.1: Bit-Counter ASMD Chart

The specification requests that all datapath components and an FSM within the control circuit are implemented as separate (sub)modules. The bit-counting system should then be implemented on the DE1_SoC such that complete functionality can be tested. The specification specifically stated that SW[7:0] should correlate to the 8-bit input (A), SW[9] as the start signal (s), KEY[0] as the synchronous reset, and the 50 MHz clock signal (CLOCK_50) to drive the entire system. The functionality was then to be demonstrated on a DE1_SoC video and recorded after all modules were tested.

Task 2: Binary Search Algorithm

The goal of this task was to implement a binary search algorithm that searched through an array to locate a specified 8-bit value A. For the array to look for values in, this was implemented as a 32x8 RAM instantiated within Quartus using the methods learned in lab 2. First, an ASMD chart for the algorithm was to be created with the assumption that the array had a fixed size of 32 elements. Next, an FSM and datapath were to be implemented in SystemVerilog and then connected to a memory block. The algorithm was then to be implemented on the DE1_SoC board with the specifications of: SW9 as 'start', SW[7:0] to specify 'A', KEY0 for reset, CLOCK_50 as the driving clock signal, HEX1 and HEX0 to display the address of data A in hex, and lastly LEDR9 for 'Found' and LEDR8 for 'Not Found'. For ease of testing and debugging, LED1 would indicate when the algorithm finished running and LED0 would light up when a reset signal was received. A MIF file called *my_array.mif* was to be created and filled with an ordered set of 32 8-bit integer numbers. In this case, the MIF file simply held increasing integers 0 to 31, the value of the address number at each address. The functionality was then to be demonstrated on a DE1_SoC video and recorded after all modules are tested. Like the previous task, datapath and controls are to be separated into separate (sub)modules.

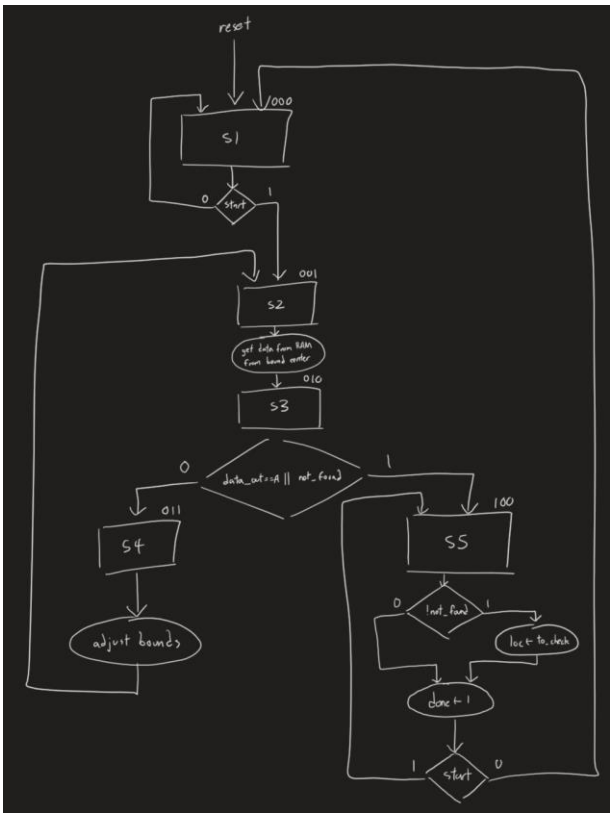

Figure 1.2: Binary Search ASMD Chart

Results:

Task 1: Bit-Counter

We approached the bit-counter by creating bitCounter.sv which served as the main bit-counting module that implemented the ASMD chart provided in Figure 1.1. This module included submodules bitCounter_datapath which contained all datapath elements, bitCounter_controller which contained the FSM and control signal operations and output, and three testbenches which tested the complete functionality, datapath functionality, and controller functionality of the bitCounter module. Although the code in a single file was very long, we decided that to keep it all under bitCounter.sv as it is more

intuitive and easier to use for other systems and projects; there also would not be much use for the controller and datapath as separate files too.

Parameters A_WIDTH and RES_WIDTH were used for code and module versatility, representing input A width and results width respectively which in this lab were set to 8 and 4 bits; 8 bits for the length of A, 4 bits to capture the maximum number of 1's in said input.

The bitCounter_datapath module takes in a clock signal (clk) and all control signals outputted from bitCounter_controller (incr_result, rshift_a, load_A, reset_result, done_). The module then stores and performs all data operations given the inputted control signals, storing and outputting the updated A value for the control unit (A_new), the total count of 1's in the input (result), and 'done' (status signal for bit-counting complete). The testbench tested all data manipulations and data storage by testing each control signal individually on a sample input data A and done signal. The waveform generated can be seen below:

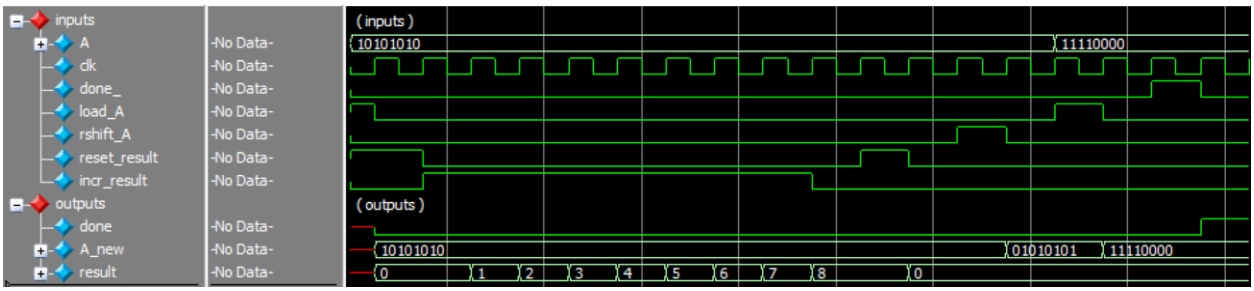

Figure 1.3: bitCounter_datapath Waveform

The waveform shows load_A loading in the input A value provided, and reset_result setting the resetting the result value to 0; both instantiation signals altered the data correctly. The result increments from 0 to 8 during when incr_result is true, and resets to 0 when reset_result is set to true; therefore the incrementing data works as intended. Next, rshift_A successfully set A_new to the input A shifted one value right, 8'b10101010 to 8'b01010101. Lastly, done reflects the input signal done_, therefore showing that the datapath stores the done signal/value correctly. As an additional test, A_new updates to new input A = 8'b11110000 when load_A is set to 1 again, therefore the load signal is verified.

The bitCounter_controller module takes in a clock (clk), reset, s (start signal), and input data 'A' which should be the output from the datapath as those values are updated. The module then utilizes a 3-state FSM and combinational as well as sequential logic to calculate and output control signals incr_result (increment results by 1), rshift_A (right shift A by 1), load_A (load in new A value to system), reset_result (initialize result to 0), and done_ (done signal for datapath to store). The testbench tested all FSM transitions and tested sequences and combinations that outputted an output of 1 for each control signal. The waveform generated can be seen below:

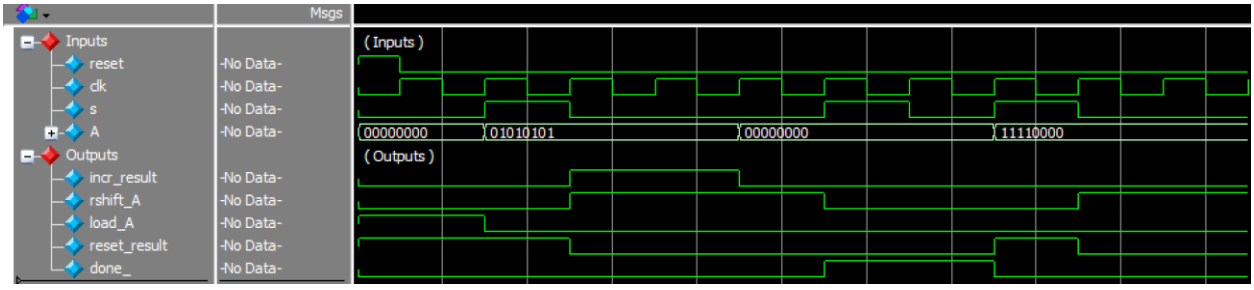

Figure 1.4: bitCounter_controller Waveform

The waveform shows reset_result and load_A being active during initial state S1, which is the expected result as result must be reset to 0 and the data input A must be loaded at the start of the system. When start signal ‘s’ is asserted, the FSM transitions into state S2 where the counting process begins. Since the least significant bit is 1 in state S2, the output signal incr_result and rshift_A are set to true, representing how the count of 1’s (result) should increment, and the A value being analyzed must be shifted right as the right-most data has been analyzed already. When the rest of A is set to 0 while in S2, the FSM transitions into S3 as there are no more 1’s left to possibly add to the results, therefore the count is finished. Since the count is finished, the done_ signal is set to 1, indicating the bit-counting is complete. Lastly, loading in a new value A (8’b00000000 since there wasn’t a clock cycle to allow 8’b11110000 to be loaded) and resetting the system was tested by turning ‘s’ off an on; the expected results similar to the first run are seen as load_A remained off, reset_result turned on, then the state transitioned to S2, and giving the signal that the data should be shifted right after analyzing the left-most bit of 0 which does not signify and incrementing result.

The bitCounter top module then instantiated the datapath and control submodules to cleanly implement the entire ASMD chart. A testbench ‘bitCounter_tb’ was then created which tested all ASMD chart branches and functions. Notably, the test bench tested reset, changing input data without starting, then using the bitCounter system on 8’b00000001 to 8’b00000011 to 8’b00000111 . . . to 8’b11111111, then 8’b11111111 left shifted eventually to 8’b0. The waveform can be seen in the figure below:

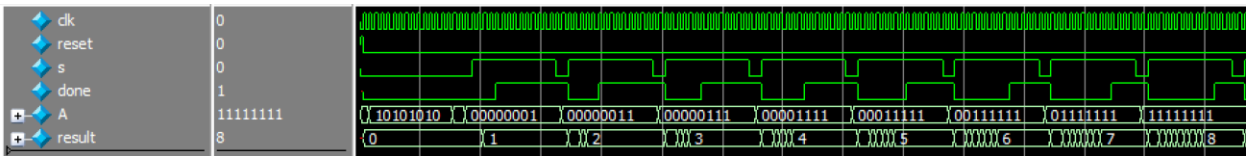

Figure 1.5: bitCounter_tb Waveform Part 1

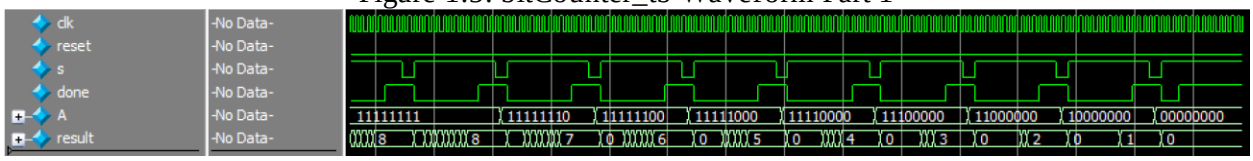

Figure 1.6: bitCounter_tb Waveform Part 2

Starting with the leftmost side of Figure 1.4, the testbench resets to initialize the system then tests if the actual bit-counting operation begins or does not begin without a start signal. The waveform depicts the expected functionality as ‘result’ and ‘done’ both remain at 0 until the start signal is provided later on. Throughout the rest of the waveform, the results can be seen settling to the total number of 1’s contained in the input data A at the time when the start signal began; the done signal also reflects when the system is complete with the counting and result has settled to the correct number. It can be seen that it takes more clock cycles for the operation to complete (seen by the temporary result values and delay before the done signals) as there are increasingly more 1’s to evaluate before the remaining values of A are 0, thus completing the operation. In Figure 1.5, very similar results appear as in Figure 1.4, the only difference being that each operation takes an equivalent amount of time since the operation cannot complete until the remaining bits of input A equal 0; since all of their leftmost bit (besides 8’b0) are 1, the operations all take the same amount of time. As an extra note, the testbench checks incrementing and decrementing the number of 1’s in input A, therefore allowing for an easy to test and read waveform.

After the bitCounter module was thoroughly tested and proved to function as accordingly to the provided ASMD chart, a top level module DE1_SoC_task1.sv was created to implement the functionality on the FPGA board. The top level module instantiated the bitCounter.sv module with the inputs and outputs

corresponding to the DE1_SoC board according to the specification described in the Task 1 procedure of this lab report. Submodule seg7.sv (created and tested in Lab 2) was utilized to display the results of the bit-counting system on hex-display HEX0. A testbench, DE1_SoC_task1_tb, was then created. The testbench implemented the exact same tests as bitCounter_tb by assigning the DE1_SoC components/variable values equal to their corresponding bitCounter.sv variables; doing so allowed the same 'initial begin' code to be used for the DE1_SoC. The waveform can be seen in the figure below:

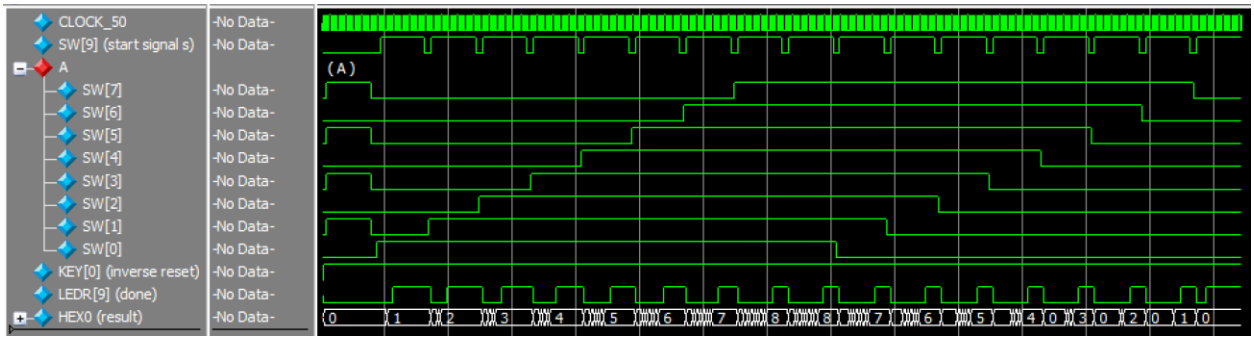

Figure 1.7: DE1_SoC_task1 Waveform

The exact same tests are the bitCounter testbench were implemented and shown in the waveform; all explanations and testing performed on bitCounter_tb apply here. CLOCK_50 can be seen as the driving clock behind all modules, HEX0 displays the resulting total number of ON signals of input A, which itself is given its value by SW[7:0]. In other words, we can say that HEX0 should display the total count of how many switches are flipped on at the time when the start signal (KEY[9]) was turned on. When the operation is complete, LEDR[9] turns on, acting as the done signal. It is important to note that seg7.sv was used to convert 7-bit binary to the HEX display as created and tested in Lab 2. The waveform generated clearly depicts the expected results, as 'result' when 'done' is true is equal to the total number of switches (representing input A) are on at the time of when the start signal (KEY[9]) was switched on. All test cases passed and matched the same tested and verified waveforms and outputs of bitCounter_tb; again, all explanations and testing of bitCounter_tb apply here.

After verifying the DE1_SoC bit-counting functionality, a demonstration video was recorded (found on Canvas), and a block diagram was created as found below:

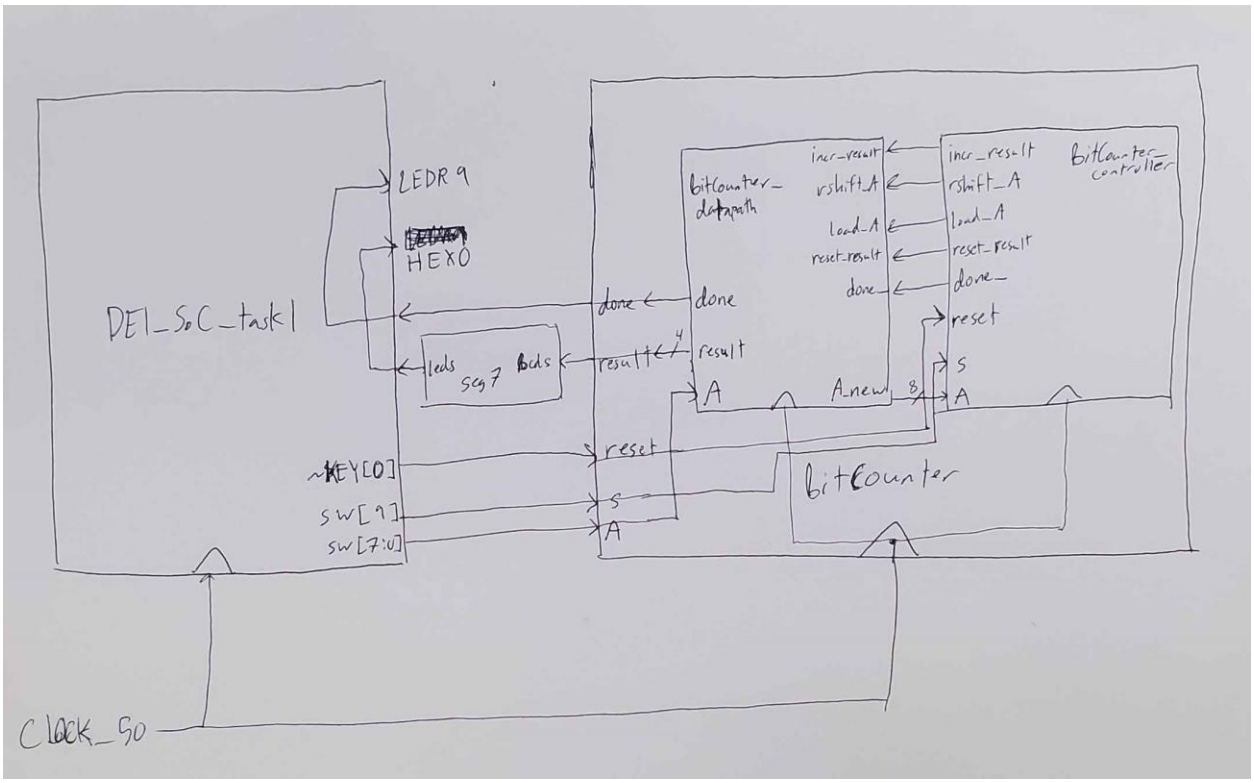

Figure 1.8: DE1_SoC_task1 Block Diagram

Task 2: Binary Search Algorithm

Like task 1, task 2 was implemented by having a top-level module task2.sv, with submodules for the datapath and control modules. The datapath module controls the RAM where the values inside the binary search array are, and the control module is responsible for the FSM changing of states and the bounds where the binary search should look at during any point in the algorithm. This implementation has five-states: one idle, for waiting for a start signal; one state to run the algorithm and fetch a value at the address in between the bounds; one state to determine if the search has finished running; one state to adjust the bounds of the search before running the search again; and one state to stay at when the search has finished running. The algorithm generally works by fetching the value at the centermost address (initially, this will always be 15 since the address width of the RAM is 32) between two bounds (at addresses 0 and 32 initially) and check if the fetched value is the desired value being searched for. If it is not found, the value fetched will be compared to the desired value, and depending on if the fetched value is higher or lower than the desired, the bounds will be updated accordingly. This process repeats until the value is found, or the algorithm can no longer be run because the bounds are too small – meaning that the value was not found in the given array.

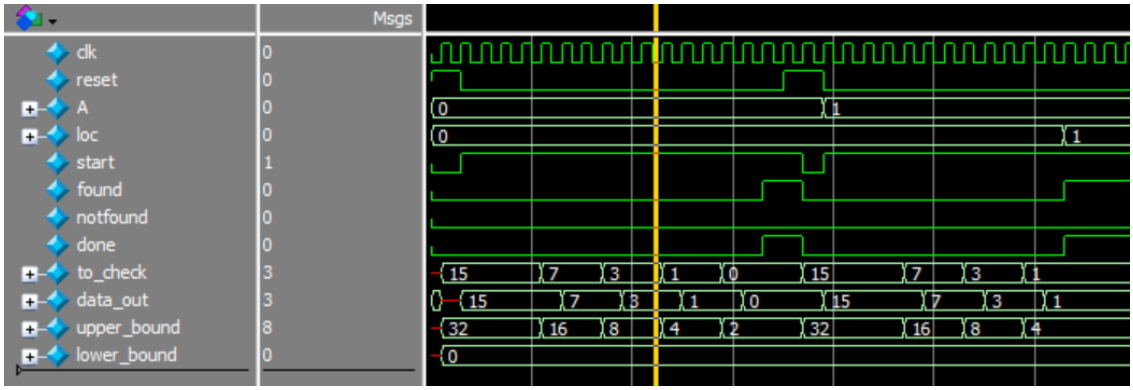

Figure 2.1: task2.sv Waveform, part 1

The testbench for the high-level module tested searching for the lowest value as an edge case, and the second lowest value as one of the values that takes the longest to get to. The upper bound changes to half the width of the current search size, and lower bound values do not range at all, as expected. At the end, when the value is found, the found and done values tick to high to indicate that the algorithm has finished running, and that the given value at A was found.

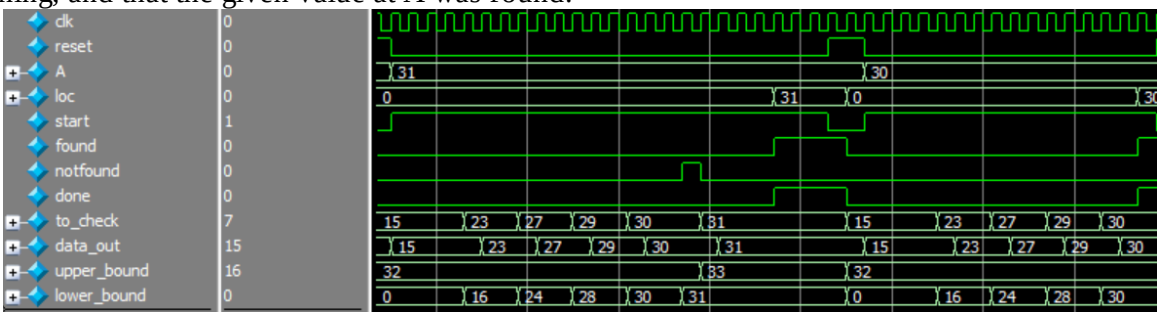

Figure 2.2: task2.sv Waveform, part 2

Next, the testbench tested for the highest and second highest values, for similar reasons as the tests above. For testing the value at array address 31, the notfound signal does tick up for a single clock cycle, but corrects itself quickly and shows the found signal alongside the done signal to indicate that the algorithm has fully and truly found the value and is confident in its result – meaning that the tiny uptick in notfound can be ignored. The address location loc also does not update until the algorithm finishes fully running. This issue is due to the way the edge case for the search of the last element in the array is handled – the bounds are adjusted when looking for values above 30 and creates a minor timing and compatibility issue when paired with the notfound logic. Looking for the value at address 30 works as normal.

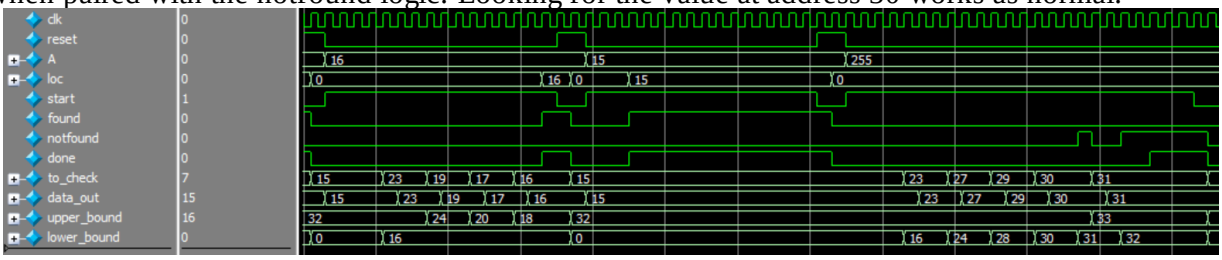

Figure 2.3: task2.sv Waveform, part 3

Finally, the testbench tests what happens when the value at 16 is searched for, since 16 takes the maximum number of iterations to search for and has a combination of up and down binary search iterations. When the value at 15 is searched for, the value is found instantly, as expected. When a value that does not exist in the array, the notfound signal ticks up when the done signal goes high, as expected since the value could not be found.

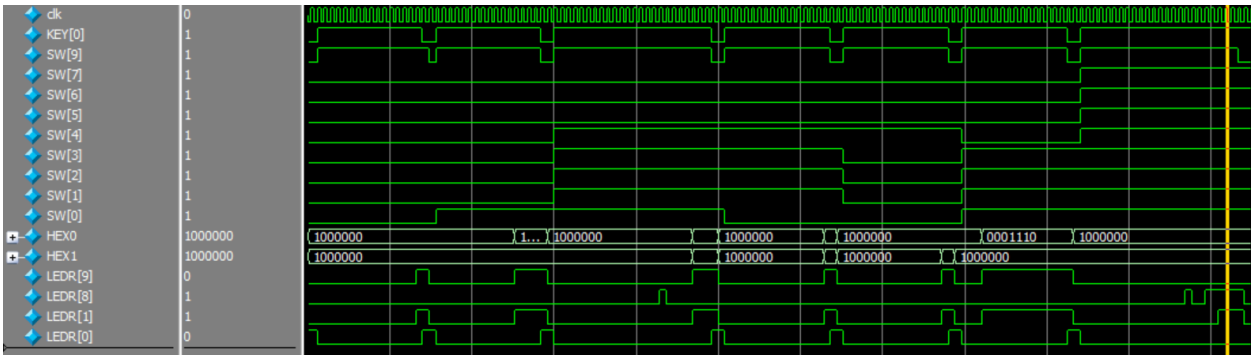

Figure 2.4: DE1_SoC highest level waveform

When tested with the DE1_SoC in mind, the results mirror those of the original tests of task2.sv. The LEDRs linked up to the found, notfound, done, and reset signals all light up during correct situations as expected.

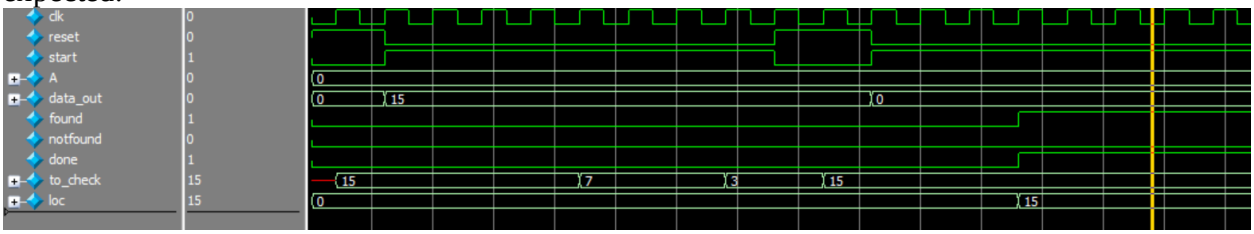

Figure 2.5: Control module waveform

The testbenches for the control module described the behavior when the value returned from the address to_check is not the desired value A, and when the value is in fact the desired value A. When it is not, the to_check halves itself while the value is not found, indicating that the bounds are shifting correctly. When the value is found, the signal found and done goes to high, meaning the module detects the RAM array having the desired value.

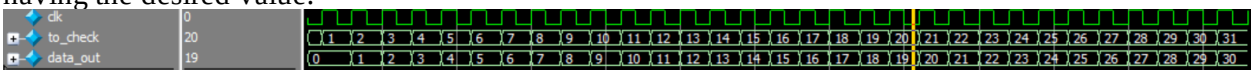

Figure 2.6: Datapath module waveform

Since the majority of the datapath module consists of the ram32x8 file to store values, only a brief iteration of all the values inside the RAM and confirmation was needed to ensure that the datapath was working.

After verifying the DE1_SoC binary search functionality, a demonstration video was recorded (found on Canvas), and a block diagram was created as found below:

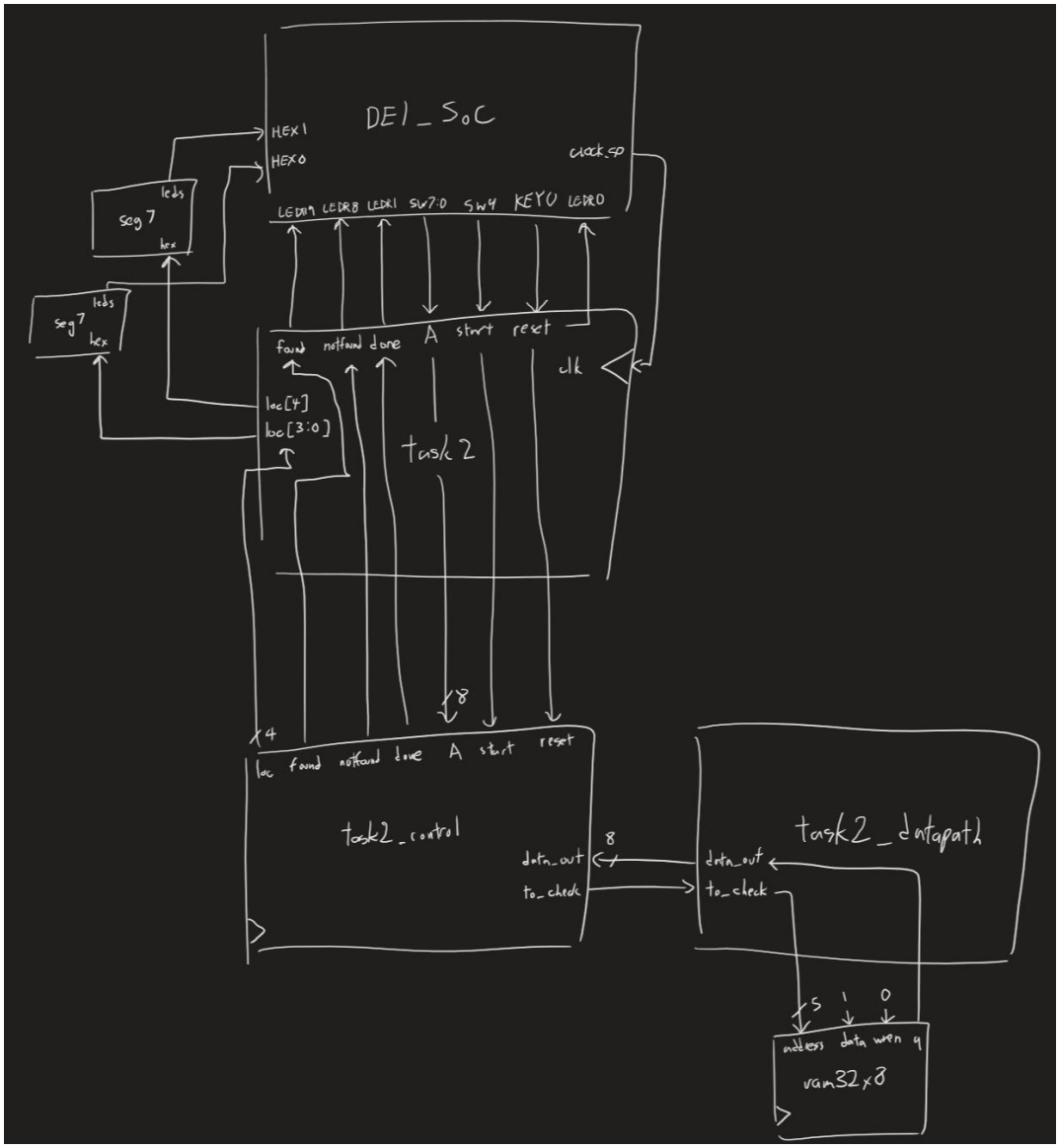

Figure 2.7: Full high-level block diagram for Task 2

Final Product:

Overall, this lab provided a great opportunity to practice utilizing ASMD charts within the hardware-designing workflow, as well as a strong opportunity to practice and understand how datapath and controller components are different and how they work together to create a system's complete functionality. Task 1 allowed us to gain experience in implementation a hard-ware design *given* a specified ASMD chart; it was a completely new experience to have to analyze a very specific implementation and divide and implement it into its datapath and controller components. Task 2 helped develop the skills of drawing ASMD charts before implementing a project and streamlined the process for designing in SystemVerilog.

Appendix: SystemVerilog Code

1) DE1_SoC_task1.sv


```
1   //Aidan Lee, Aarin Wen
2   // 11/13/2024
3   // EE 371
4   // Lab 4
5
6   // Top level module to simulate bitCounter on the DE1_SoC board
7
8   // inputs:
9   // CLOCK_50                  -> clk   (clock signal  for bitCounter)
10  // SW[7:0]                   -> A     (input data    for bitCounter)
11  // ~KEY[0] (pressing KEY[0])  -> reset (reset signal  for bitCounter)
12
13  // outputs:
14  // LEDR[9]  <- done     (bitCounter status signal; counting complete)
15  // HEX[0]   <- results  (bitCounter output; total 1's count of input A)
16
17  module DE1_SoC_task1 (
18  input logic [3:0] KEY,
19  input logic [9:0] SW,
20  input logic CLOCK_50,
21  output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
22  output logic [9:0] LEDR);
23
24     // turn off unused HEX displays
25     assign HEX1 = 7'b1111111;
26     assign HEX2 = 7'b1111111;
27     assign HEX3 = 7'b1111111;
28     assign HEX4 = 7'b1111111;
29     assign HEX5 = 7'b1111111;
30
31     // logic variable to store result
32     logic [3:0] result;
33
34     // parameters, 8-bit wide A, 4 bits to store results
35     parameter A_WIDTH = 8;
36     parameter RES_WIDTH = 4;
37
38     // bitCounter instantiated with...
39     // parameters #(A_WIDTH, RES_WIDTH)
40     // clk      <- CLOCK_50
41     // A        <- SW[7:0]
42     // reset    <- ~KEY[0] (pressing KEY[0])
43     // done     -> LEDR[9]
44     // result   -> result (will display on HEX0 using seg7.sv)
45     bitCounter #(A_WIDTH, RES_WIDTH) bitCounter_
46     (.reset(~KEY[0]), .clk(CLOCK_50), .s(SW[9]), .A(SW[7:0]), .done(LEDR[9]), .result);
47
48     // display result on 7-segment display HEX0
49     // seg7 instantiated with results as input, HEX0 as output
50     seg7 HEX0_result (.bcd(result), .leds(HEX0));
51  endmodule
52
53  // DE1_SoC_task1.sv testbench; runs same test cases as bitCounter.sv
54  // tests reset and s cases as well as A <= 8'b10101010, 8'b10000001, and 8'b0;
55  module DE1_SoC_task1_tb ();
56     logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
57     logic [9:0] LEDR;
58     logic [3:0] KEY;
59     logic [9:0] SW;
60     logic CLOCK_50;
61
62     // instantiate DE1_SoC_task1 dut
63     DE1_SoC_task1 dut (.*);
64
65     // assign DE1_SoC variables to bitCounter counterparts
66     logic clk, s, reset;
67     logic [7:0] A;
68     assign CLOCK_50 = clk;
69     assign SW[7:0] = A;
70     assign SW[9] = s;
71     assign KEY[0] = ~reset;
72
73     // initialize clock simulation
```

```
74      parameter CLOCK_PERIOD = 100;
75      initial begin
76         clk <= 0;
77         forever #(CLOCK_PERIOD/2) clk <= ~clk;
78      end
79
80      // begin tests
81      integer i; // int i for loops
82      // begin tests
83      initial begin
84         // initialize values
85         reset <= 1; s <= 0; A <= 8'b0; @(posedge clk);
86
87         // release reset
88         reset <= 0; @(posedge clk);
89
90         // test if bit-count analysis begins without start signal (SHOULD NOT)
91         // results & done should remain at 0
92         A <= 8'b10101010; repeat (14) @(posedge clk);
93         A <= 8'b00000000; @(posedge clk);
94
95         // loop that uses bitCounter on bits 8'b00000001 to 8'b11111111 by incrementing
96         // results should settle to how many 1's are in the 8b input; expect 1->8
97         // done should have value '1' whenever results are settled
98         // note: 8b'0 will be checked in next loop
99         for (i = 0; i < 8; i = i + 1) begin
100           // turn start signal off before loading in new data
101           s <= 0; @(posedge clk);
102
103           // load in new A value
104           A <= (8'b00000001 << i) | A; @(posedge clk);
105
106           // count 1's and let 'result' and 'done' settle
107           s <= 1; repeat (14) @(posedge clk);
108        end
109
110        // loop that uses left shifting to bit-count 8'b11111111 to 8'b0
111        // results should settle to how many 1's are in the 8b input; expect 8->0
112        // done should have value '1' whenever results are settled
113        for (i = 8; i >= 0; i = i - 1) begin
114           // turn start signal off before loading in new data
115           s <= 0; @(posedge clk);
116
117           // load in new A value
118           A <= 8'b11111111 << (8-i); @(posedge clk);
119
120           // count 1's and let 'result' and 'done' settle
121           s <= 1; repeat (14) @(posedge clk);
122        end
123        $stop;
124
125     end
126  endmodule
```

2) bitCounter.sv


```
1   //Aidan Lee, Aarin Wen
2   // 11/13/2024
3   // EE 371
4   // Lab 4
5
6   // Main bitCounter module
7   // Counts the number of 1's in input A and outputs total as 'result' and a 'done' signal
8
9   // Parameters: A_WIDTH (width of input A) and RES_WIDTH (width of result)
10
11  // inputs:
12  // 1'b wide: reset, clk, s (start signal),
13  // A_WIDTH wide: A (data input)
14
15  // outputs:
16  // 1'b wide: done (counting complete)
17  // RES_WIDTH wide: result (total count of 1's in A)
18
19  module bitCounter #(parameter A_WIDTH = 8, RES_WIDTH = 4) (
20  input logic reset, clk, s,
21  input logic [A_WIDTH-1:0] A,
22  output logic done,
23  output logic [RES_WIDTH-1:0] result);
24
25     // logic for control signals and updated A value
26     logic incr_result, rshift_A, load_A, reset_result, done_;
27     logic [A_WIDTH-1:0] A_new;
28
29     // initialize datapath
30     // all inputs and outputs instantiated with their respective matching variables
31     bitCounter_datapath     #(A_WIDTH, RES_WIDTH)   datapath
32     (.clk, .incr_result, .rshift_A, .load_A, .reset_result, .done_, .A, .done, .A_new, .
   result);
33
34     // initialize controller
35     // all inputs and outputs besides A instantiated with their respective matching variables
36     // input 'A' is instantiated with 'A_new' which is the updated A value from the datapath
37     bitCounter_controller   #(A_WIDTH, RES_WIDTH)   controller
38     (.reset, .clk, .s, .A(A_new), .incr_result, .rshift_A, .load_A, .reset_result, .done_);
39
40  endmodule
41
42  ////////////////////////////////////////////////////////////////////////////
43
44  // bitCounter datapath module
45  // Contains all datapath elements and data operations of bitCounter,
46  // also outputs updated A value for controller
47
48  // Parameters: A_WIDTH (width of input A) and RES_WIDTH (width of result)
49
50  // inputs:
51  // 1'b wide: clk,
52  //           incr_result, rshift_a, load_A, reset_result, done_ (signals from controller)
53  // A_WIDTH wide: A (input data)
54
55  // outputs:
56  // 1'b wide: done (count complete status signal)
57  // A_WIDTH wide: A_new (updated A value for controller)
58  // RES_WIDTH wide: result (total 1's in A)
59
60  module bitCounter_datapath #(parameter A_WIDTH = 8, RES_WIDTH = 4) (
61  input logic clk, incr_result, rshift_A, load_A, reset_result, done_,
62  input logic [A_WIDTH-1:0] A,
63  output logic done,
64  output logic [A_WIDTH-1:0] A_new,
65  output logic [RES_WIDTH-1:0] result);
66
67     // always_ff block to synchronously manipulate and move data
68     always_ff @(posedge clk) begin
69        done <= done_;    // store 'done'
70
71        if (incr_result)  // increment result
72           result <= result + 1'b1;
```

```
73
74          if (rshift_A)     // right shift A by 1
75             A_new <= A_new >> 1'b1;
76
77          if (load_A)       // load A
78             A_new <= A;
79
80          if (reset_result) // set result = 0
81             result <= 0;
82       end
83
84    endmodule
85
86    ////////////////////////////////////////////////////////////////////////
87
88    // bitCounter controller module
89    // Contains all controller elements of bitCounter including FSM,
90    // also takes in and utilizes updated A value from datapath output
91    // also contains logic and operatiosn to output correct control signals
92
93    // Parameters: A_WIDTH (width of input A) and RES_WIDTH (width of result)
94
95    // inputs:
96    // 1'b wide: reset, clk, s (start signal)
97    // A_WIDTH wide: A (input data)
98
99    // outputs:
100   // 1'b wide:   incr_result,   (increment results)
101   //             rshift_A,      (right shift A by 1 bit)
102   //             load_A,        (load new A value)
103   //             reset_result,  (reset result value to 0)
104   //             done_          (count complete signal)
105
106
107   module bitCounter_controller #(parameter A_WIDTH = 8, RES_WIDTH = 4) (
108   input logic reset, clk, s,
109   input logic [A_WIDTH-1:0] A,
110   output logic incr_result, rshift_A, load_A, reset_result, done_);
111
112      // present and next states S1 S2 S3
113      enum {S1, S2, S3} ps, ns; // S1 = initialization, S2 = analyzing, S3 = finished
114
115      // FSM state-to-state logic
116      always_comb begin
117         case (ps)
118            S1: ns = s ? S2: S1;          // ns to S2 if s,    S1 OW
119            S2: ns = (A == 0) ? S3: S2;   // ns to S3 if A==0, S2 OW
120            S3: ns = s ? S3: S1;          // ns to S3 if s,    S1 OW
121         endcase
122      end
123
124      // combinational logic for output control signals
125      always_comb begin
126         // increment signal if in S2, A != 0 and rightmost bit == 1
127         incr_result    = (ps == S2) & (ns == S2) & (A[0] == 1);
128         // right shift signal if ps is S2 (data analyzing and unfinished)
129         rshift_A       = ps == S2;
130         // load signal if data analyzing hasnt started yet & no start signal
131         load_A         = (ps == S1) & (~s);
132         // reset result signal if data analyzing for current data hasnt started
133         reset_result   = ps == S1;
134         // send done signal if data analyzing is complete
135         done_          = ps == S3;
136      end
137
138      // ps <= ns unless reset, then ps resets to S1
139      always_ff @(posedge clk) begin
140         if (reset)
141            ps <= S1;
142         else
143            ps <= ns;
144      end
145
```

```
146  endmodule
147
148  ////TESTBENCHES/////////////////////////////////////////////////////////////
149
150  // main testbench
151  // tests reset and s cases as well as A <= 8'b10101010, 8'b10000001, and 8'b0;
152  module bitCounter_tb();
153     // variables and parameters
154     parameter A_WIDTH = 8;
155     parameter RES_WIDTH = 4;
156     logic reset, clk, s, done;
157     logic [A_WIDTH-1:0] A;
158     logic [RES_WIDTH-1:0] result;
159
160     // create bitCounter dut
161     bitCounter #(A_WIDTH, RES_WIDTH) dut (.*);
162
163     // initialize clock simulation
164     parameter CLOCK_PERIOD = 100;
165     initial begin
166        clk <= 0;
167        forever #(CLOCK_PERIOD/2) clk <= ~clk;
168     end
169
170     integer i; // int i for loops
171     // begin tests
172     initial begin
173        // initialize values
174        reset <= 1; s <= 0; A <= 8'b0; @(posedge clk);
175
176        // release reset
177        reset <= 0; @(posedge clk);
178
179        // test if bit-count analysis begins without start signal (SHOULD NOT)
180        // results & done should remain at 0
181        A <= 8'b10101010; repeat (14) @(posedge clk);
182        A <= 8'b00000000; @(posedge clk);
183
184        // loop that uses bitCounter on bits 8'b00000001 to 8'b11111111 by incrementing
185        // results should settle to how many 1's are in the 8b input; expect 1->8
186        // done should have value '1' whenever results are settled
187        // note: 8b'0 will be checked in next loop
188        for (i = 0; i < 8; i = i + 1) begin
189           // turn start signal off before loading in new data
190           s <= 0; @(posedge clk);
191
192           // load in new A value
193           A <= (8'b00000001 << i) | A; @(posedge clk);
194
195           // count 1's and let 'result' and 'done' settle
196           s <= 1; repeat (14) @(posedge clk);
197        end
198
199        // loop that uses left shifting to bit-count 8'b11111111 to 8'b0
200        // results should settle to how many 1's are in the 8b input; expect 8->0
201        // done should have value '1' whenever results are settled
202        for (i = 8; i >= 0; i = i - 1) begin
203           // turn start signal off before loading in new data
204           s <= 0; @(posedge clk);
205
206           // load in new A value
207           A <= 8'b11111111 << (8-i); @(posedge clk);
208
209           // count 1's and let 'result' and 'done' settle
210           s <= 1; repeat (14) @(posedge clk);
211        end
212        $stop;
213
214     end
215  endmodule
216
217  // datapath testbench
218  module bitCounter_datapath_tb();
```

```
219      // variables and parameters
220      parameter A_WIDTH = 8;
221      parameter RES_WIDTH = 4;
222      logic clk, incr_result, rshift_A, load_A, reset_result, done_;
223      logic [A_WIDTH-1:0] A;
224
225      logic done;
226      logic [A_WIDTH-1:0] A_new;
227      logic [RES_WIDTH-1:0] result;
228
229      // create bitCounter dut
230      bitCounter_datapath #(A_WIDTH, RES_WIDTH) dut2 (.*);
231
232      // initialize clock simulation
233      parameter CLOCK_PERIOD = 100;
234      initial begin
235         clk <= 0;
236         forever #(CLOCK_PERIOD/2) clk <= ~clk;
237      end
238
239      initial begin
240         // initialize values
241         incr_result <= 0;
242         rshift_A <= 0;
243         load_A <= 0;
244         reset_result <= 1;
245         done_ <= 0;
246
247         // test load_A, new data A = 8'b10101010 should load in.
248         A <= 8'b10101010;
249         load_A <= 1;@(posedge clk);
250         load_A <= 0;@(posedge clk);
251
252         // test incrementing, result should increment from decimal 0 to 8
253         reset_result <= 0;
254         incr_result <= 1; repeat (8) @(posedge clk);
255         incr_result <= 0; @(posedge clk);
256
257         // test reset_result, result should go to 0
258         reset_result <= 1; @(posedge clk);
259         reset_result <= 0; @(posedge clk);
260
261         // test right shift A, A_new should now equal 8'b01010101
262         rshift_A <= 1; @(posedge clk);
263         rshift_A <= 0; @(posedge clk);
264
265         // test loading in new value A
266         A <= 8'b11110000;
267         load_A <= 1;@(posedge clk);
268         load_A <= 0;@(posedge clk);
269
270         // test that done stores and outputs done_ (output from controller)
271         done_ <= 1; @(posedge clk);
272         done_ <= 0; @(posedge clk);
273
274         $stop;
275
276      end
277   endmodule
278
279
280   // Controller testbench
281   module bitCounter_controller_tb();
282       // parameters
283       parameter A_WIDTH = 8;
284       parameter RES_WIDTH = 4;
285
286       // signals for inputs and outputs
287       logic reset, clk, s;
288       logic [A_WIDTH-1:0] A;
289
290       logic incr_result, rshift_A, load_A, reset_result, done_;
291
```

```
292       // instantiate the controller
293       bitCounter_controller #(A_WIDTH, RES_WIDTH) dut3 (.*);
294
295       // initialize clock simulation
296       parameter CLOCK_PERIOD = 100;
297       initial begin
298           clk <= 0;
299           forever #(CLOCK_PERIOD/2) clk <= ~clk;
300       end
301
302       // test sequence
303      initial begin
304          // initialize inputs
305          reset <= 1;
306          s <= 0;
307          A <= 8'b0;
308          @(posedge clk);
309
310          // Release reset, enter S1
311          // Expect: reset_result = 1, all other signals = 0
312          reset <= 0; @(posedge clk);
313
314          // Test S1 to S2 transition with s = 1
315          s <= 1; A <= 8'b01010101; @(posedge clk);
316          // Expect: load_A = 1, all other signals = 0
317
318          // Test S2 state behavior with non-zero A (data analyzing)
319          s <= 0; repeat(2) @(posedge clk);
320          // Expect:
321          // rshift_A = 1
322          // incr_result = 1 since least significant bit of A is 1
323          // all other signals = 0
324
325          // Test S2 to S3 transition when A == 0
326          A <= 8'b0; @(posedge clk);
327          // Expect: done_ = 1 (indicates counting is complete), all other signals = 0
328
329          // Test S3 to S1 transition when s is toggled
330          s <= 1; @(posedge clk);
331          // Expect:
332          // done_ remains 1 if in S3, indicating analyzing is complete
333          // done_ returns to 0 upon resetting back to S1 given s = 0
334          s <= 0; @(posedge clk);
335
336          // test if loading in new value works
337          A <= 8'b11110000;
338          s <= 1; @(posedge clk);
339          s <= 0; repeat(2) @(posedge clk);
340
341         $stop;
342       end
343   endmodule
344
345
```

3) seg7.sv


```
1   // Aidan Lee, Aarin Wen
2   // 10/18/2024
3   // EE 371
4   // Lab 2
5
6   // Converts decimal number to 7-segment display in HEX
7   // input bcd represents decimal number up to 15 (4 bits)
8   // output leds represents binary output for 7-seg display (7 bits)
9
10  module seg7 (bcd, leds);
11     input logic [3:0] bcd;
12     output logic [6:0] leds;
13
14     // decimal to 7-seg block
15     always_comb begin
16        case (bcd)
17           // Light:          6543210
18           4'b0000: leds = 7'b1000000; // 0
19           4'b0001: leds = 7'b1111001; // 1
20           4'b0010: leds = 7'b0100100; // 2
21           4'b0011: leds = 7'b0110000; // 3
22           4'b0100: leds = 7'b0011001; // 4
23           4'b0101: leds = 7'b0010010; // 5
24           4'b0110: leds = 7'b0000010; // 6
25           4'b0111: leds = 7'b1111000; // 7
26           4'b1000: leds = 7'b0000000; // 8
27           4'b1001: leds = 7'b0010000; // 9
28           4'b1010: leds = 7'b0001000; // A
29           4'b1011: leds = 7'b0000011; // B
30           4'b1100: leds = 7'b1000110; // C
31           4'b1101: leds = 7'b0100001; // D
32           4'b1110: leds = 7'b0000110; // E
33           4'b1111: leds = 7'b0001110; // F
34           default: leds = 7'bx;
35        endcase
36     end
37  endmodule
38
39  // seg7 testbench:
40  module seg7_testbench();
41     logic [3:0] bcd;
42     logic [6:0] leds;
43
44     //dut instantiation
45     seg7 dut (bcd, leds);
46
47     int i;
48     initial begin
49        // increment inputs 0 to 15, expect outputs 0 to F
50        for (i = 0; i <= 15; i++) begin
51           bcd = i; #50;
52        end
53
54        // decrement inputs 15 to 0, expect outputs F to 0
55        for (i = 15; i >= 0; i--) begin
56           bcd = i; #50;
57        end
58
59        $stop;
60     end
61
62  endmodule
```

4) **De1_SoC.sv (task 2)**

```
`timescale 1ns/1ns

// Aidan Lee, Aarin Wen
// 11/13/2024
// EE 371
// Lab 4
//
// Binary search module that returns the address loc in SW[7:0] sorted ascending 32x8 ram module if SW[7:0] value SW[7:0] is found.
// Seaches the middle most value of the array. If not found, the module decides whether to search
// the upper or lower half depending on if the value found at the middle is higher or lower than the
// the desired value. Needs RAM to be sorted ascending to work.
//
// utilizes clk input
// input  reset             resets the search (clears the output) and waits for another SW[9] sigal
// input  [7:0] A           represents the value that you want to find inside the
// input  start             represents the y coordinates of the two endpoints of the line you want to draw
// output found             goes high if the value SW[7:0] was found in the ram
// output final_notfound    goes high if the value SW[7:0] was not found
// output done              goes high if algorithm is finished
// output [4:0] loc         represents the address location where the value is found

module De1SoC (CLOCK_50, KEY, SW, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, LEDR);

    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;

    logic clk, reset, done, found, notfound;
    logic [4:0] loc;
    logic [7:0] result;

    logic start;

    assign reset = ~KEY[0]; // reset on KEY0
    assign start = SW[9]; // start on SW9
    assign LEDR[9] = found; // 9 lights up if found
    assign LEDR[8] = notfound; // 8 lights up if not found
    assign LEDR[1] = done; // 1 indicates finished
    assign LEDR[0] = reset; // 0 indicates currently in reset

    assign clk = CLOCK_50;

   // main binary search algorithm
   //
   // takes in clock and reset
   // takes in SW[7:0] as the value to look for in binary search
   // takes in SW[9] to indicate that the algorithm should SW[9] running
   // gives out found and notfound depending on if the value was found in ram
   // gives out done when finished looking for value
   // gives out loc which is the address where the desired value SW[7:0] is stored
   task2 taskDisp2 (reset, clk, SW[7:0], start, found, notfound, done, loc); // SW[7:0] to specify A
   seg7 disp1 ({3'b0, loc[4]}, HEX1);
   seg7 disp0 (loc[3:0], HEX0);

   // none of the other hexes are used for this task
    assign HEX2 = 7'b1111111;
    assign HEX3 = 7'b1111111;
    assign HEX4 = 7'b1111111;
    assign HEX5 = 7'b1111111;

    // kill all other leds
    assign LEDR[7:2] = 6'b000000;

endmodule

module De1SoC_tb ();
    logic CLOCK_50;
    logic [3:0] KEY;
    logic [9:0] SW;
    logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    logic [9:0] LEDR;

    logic clk;
    assign CLOCK_50 = clk;

    // Clock logic
    parameter CLOCK_PERIOD = 5; // Increase clock frequency
    initial begin
        clk <= 0;
        forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
    end

    De1SoC dut (.*);

    initial begin
        SW[7:0] <= 8'b00000000; SW[9] <= 1'b0;
        KEY[0] <= 0; @(posedge clk);
                     @(posedge clk);

        // test very end
        SW[9] <= 1'b1; KEY[0] <= 1; SW[7:0] <= 8'b00000000;
        repeat (16) @(posedge clk);


        KEY[0] <= 0; @(posedge clk); // check if reset works
        SW[9] <= 1'b0; @(posedge clk);

        // test lowest value of binary search that takes the longest time
        SW[9] <= 1'b1; KEY[0] <= 1; SW[7:0] <= 8'b00000001;
        repeat (16) @(posedge clk);

        SW[9] <= 1'b0;
        KEY[0] <= 0; @(posedge clk);
                     @(posedge clk);

        // test highest value (edge case)
        SW[9] <= 1'b1; KEY[0] <= 1; SW[7:0] <= 8'd31;
        repeat (24) @(posedge clk);

        SW[9] <= 1'b0;
        KEY[0] <= 0; @(posedge clk);
                     @(posedge clk);
```

```
115
116         // test highest value of binary search that takes the longest time
117         SW[9] <= 1'b1; KEY[0] <= 1; SW[7:0] <= 8'd30;
118         repeat (16) @(posedge clk);
119
120         SW[9] <= 1'b0;
121         KEY[0] <= 0; @(posedge clk);
122                      @(posedge clk);
123
124         // test SW[7:0] value of binary search in the middle that takes the longest time
125         // to test that going up and down works
126         SW[9] <= 1'b1; KEY[0] <= 1; SW[7:0] <= 8'd16;
127         repeat (16) @(posedge clk);
128
129         SW[9] <= 1'b0;
130         KEY[0] <= 0; @(posedge clk);
131                      @(posedge clk);
132
133         // test finding the value immediately
134         SW[9] <= 1'b1; KEY[0] <= 1; SW[7:0] <= 8'd15;
135         repeat (16) @(posedge clk);
136
137         SW[9] <= 1'b0;
138         KEY[0] <= 0; @(posedge clk);
139                      @(posedge clk);
140
141         // test value not in array
142         SW[9] <= 1'b1; KEY[0] <= 1; SW[7:0] <= 8'b11111111;
143         repeat (24) @(posedge clk);
144         SW[9] <= 1'b0; @(posedge clk); @(posedge clk);
145
146         $stop;
147     end
148
149 endmodule
```

5) **task2.sv**

```
`timescale 1ns/1ns

// Aidan Lee, Aarin Wen
// 11/13/2024
// EE 371
// Lab 4
//
// Binary search module that returns the address loc in a sorted ascending 32x8 ram module if a value A is found.
// Seaches the middle most value of the array. If not found, the module decides whether to search
// the upper or lower half depending on if the value found at the middle is higher or lower than the
// the desired value. Needs RAM to be sorted ascending to work.
//
// utilizes clk input
// input  reset               resets the search (clears the output) and waits for another start sigal
// input  [7:0] A             represents the value that you want to find inside the
// input  start               represents the y coordinates of the two endpoints of the line you want to draw
// output found               goes high if the value A was found in the ram
// output notfound            goes high at the end when finished if the value A was not found
// output done                goes high if algorithm is finished
// output [4:0] loc           represents the address location where the value is found

module task2 (reset, clk, A, start, found, notfound, done, loc);

   input logic reset, clk, start;
   input logic [7:0] A;

   output logic found, notfound, done;
   output logic [4:0] loc;

   // intermediate values to talk between datapath and control
   logic [4:0] to_check;
   logic [7:0] data_out;

   task2_datapath d_unit (.*);
   task2_control  c_unit (.*);


endmodule

module task2_tb ();
   logic reset, clk;
   logic [7:0] A;
   logic [4:0] loc;
   logic start, found, notfound, done;

    // Clock logic
    parameter CLOCK_PERIOD = 5; // Increase clock frequency
    initial begin
        clk <= 0;
        forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
    end

    task2 dut (.*);

    initial begin
        A <= 8'b00000000; start <= 1'b0;
        reset <= 1; @(posedge clk);
                @(posedge clk);

      // test very end
      start <= 1'b1; reset <= 0; A <= 8'b00000000;
      repeat (16) @(posedge clk);


      reset <= 1; @(posedge clk); // check if reset works
      start <= 1'b0; @(posedge clk);

      // test lowest value of binary search that takes the longest time
      start <= 1'b1; reset <= 0; A <= 8'b00000001;
      repeat (16) @(posedge clk);

      start <= 1'b0;
      reset <= 1; @(posedge clk);
                  @(posedge clk);

      // test highest value (edge case)
      start <= 1'b1; reset <= 0; A <= 8'd31;
      repeat (24) @(posedge clk);

      start <= 1'b0;
      reset <= 1; @(posedge clk);
                  @(posedge clk);

      // test highest value of binary search that takes the longest time
      start <= 1'b1; reset <= 0; A <= 8'd30;
      repeat (16) @(posedge clk);

      start <= 1'b0;
      reset <= 1; @(posedge clk);
                  @(posedge clk);

      // test a value of binary search in the middle that takes the longest time
      // to test that going up and down works
      start <= 1'b1; reset <= 0; A <= 8'd16;
      repeat (16) @(posedge clk);

      start <= 1'b0;
      reset <= 1; @(posedge clk);
                  @(posedge clk);

      // test finding the value immediately
      start <= 1'b1; reset <= 0; A <= 8'd15;
      repeat (16) @(posedge clk);

      start <= 1'b0;
      reset <= 1; @(posedge clk);
                  @(posedge clk);

      // test value not in array
      start <= 1'b1; reset <= 0; A <= 8'b11111111;
      repeat (24) @(posedge clk);
      start <= 1'b0; @(posedge clk); @(posedge clk);

      $stop;
```

```
115         end
116
117     endmodule
```

6) my_array.mif

```
-- 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.

-- Quartus Prime generated Memory Initialization File (.mif)

WIDTH=8;
DEPTH=32;

ADDRESS_RADIX=UNS;
DATA_RADIX=UNS;

CONTENT BEGIN
        0   :   0;
        1   :   1;
        2   :   2;
        3   :   3;
        4   :   4;
        5   :   5;
        6   :   6;
        7   :   7;
        8   :   8;
        9   :   9;
        10  :   10;
        11  :   11;
        12  :   12;
        13  :   13;
        14  :   14;
        15  :   15;
        16  :   16;
        17  :   17;
        18  :   18;
        19  :   19;
        20  :   20;
        21  :   21;
        22  :   22;
        23  :   23;
        24  :   24;
        25  :   25;
        26  :   26;
        27  :   27;
        28  :   28;
        29  :   29;
        30  :   30;
        31  :   31;
END;
```

7) ram32x8.v


```
1   // megafunction wizard: %RAM: 1-PORT%
2   // GENERATION: STANDARD
3   // VERSION: WM1.0
4   // MODULE: altsyncram
5
6   // ============================================================
7   // File Name: ram32x8.v
8   // Megafunction Name(s):
9   //          altsyncram
10  //
11  // Simulation Library Files(s):
12  //          altera_mf
13  // ============================================================
14  // ************************************************************
15  // THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
16  //
17  // 17.0.0 Build 595 04/25/2017 SJ Lite Edition
18  // ************************************************************
19
20
21  //Copyright (C) 2017  Intel Corporation. All rights reserved.
22  //Your use of Intel Corporation's design tools, logic functions
23  //and other software and tools, and its AMPP partner logic
24  //functions, and any output files from any of the foregoing
25  //(including device programming or simulation files), and any
26  //associated documentation or information are expressly subject
27  //to the terms and conditions of the Intel Program License
28  //Subscription Agreement, the Intel Quartus Prime License Agreement,
29  //the Intel MegaCore Function License Agreement, or other
30  //applicable license agreement, including, without limitation,
31  //that your use is for the sole purpose of programming logic
32  //devices manufactured by Intel and sold by Intel or its
33  //authorized distributors.  Please refer to the applicable
34  //agreement for further details.
35
36
37  // synopsys translate_off
38  `timescale 1 ps / 1 ps
39  // synopsys translate_on
40  module ram32x8 (
41     address,
42     clock,
43     data,
44     wren,
45     q);
46
47     input [4:0]  address;
48     input   clock;
49     input [7:0]  data;
50     input   wren;
51     output   [7:0]  q;
52  `ifndef ALTERA_RESERVED_QIS
53  // synopsys translate_off
54  `endif
55     tri1    clock;
56  `ifndef ALTERA_RESERVED_QIS
57  // synopsys translate_on
58  `endif
59
60     wire [7:0] sub_wire0;
61     wire [7:0] q = sub_wire0[7:0];
62
63     altsyncram  altsyncram_component (
64               .address_a (address),
65               .clock0 (clock),
66               .data_a (data),
67               .wren_a (wren),
68               .q_a (sub_wire0),
69               .aclr0 (1'b0),
70               .aclr1 (1'b0),
71               .address_b (1'b1),
72               .addressstall_a (1'b0),
73               .addressstall_b (1'b0),
```

```
74                .byteena_a (1'b1),
75                .byteena_b (1'b1),
76                .clock1 (1'b1),
77                .clocken0 (1'b1),
78                .clocken1 (1'b1),
79                .clocken2 (1'b1),
80                .clocken3 (1'b1),
81                .data_b (1'b1),
82                .eccstatus (),
83                .q_b (),
84                .rden_a (1'b1),
85                .rden_b (1'b1),
86                .wren_b (1'b0));
87        defparam
88           altsyncram_component.clock_enable_input_a = "BYPASS",
89           altsyncram_component.clock_enable_output_a = "BYPASS",
90           altsyncram_component.init_file = "my_array.mif",
91           altsyncram_component.intended_device_family = "Cyclone V",
92           altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
93           altsyncram_component.lpm_type = "altsyncram",
94           altsyncram_component.numwords_a = 32,
95           altsyncram_component.operation_mode = "SINGLE_PORT",
96           altsyncram_component.outdata_aclr_a = "NONE",
97           altsyncram_component.outdata_reg_a = "UNREGISTERED",
98           altsyncram_component.power_up_uninitialized = "FALSE",
99           altsyncram_component.ram_block_type = "M10K",
100          altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ",
101          altsyncram_component.widthad_a = 5,
102          altsyncram_component.width_a = 8,
103          altsyncram_component.width_byteena_a = 1;
104
105
106   endmodule
107
108   // ============================================================
109   // CNX file retrieval info
110   // ============================================================
111   // Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
112   // Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
113   // Retrieval info: PRIVATE: AclrByte NUMERIC "0"
114   // Retrieval info: PRIVATE: AclrData NUMERIC "0"
115   // Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
116   // Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
117   // Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
118   // Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
119   // Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
120   // Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
121   // Retrieval info: PRIVATE: Clken NUMERIC "0"
122   // Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
123   // Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
124   // Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
125   // Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
126   // Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
127   // Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
128   // Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
129   // Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
130   // Retrieval info: PRIVATE: MIFfilename STRING "ram32x8.mif"
131   // Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "32"
132   // Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "2"
133   // Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
134   // Retrieval info: PRIVATE: RegAddr NUMERIC "1"
135   // Retrieval info: PRIVATE: RegData NUMERIC "1"
136   // Retrieval info: PRIVATE: RegOutput NUMERIC "0"
137   // Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
138   // Retrieval info: PRIVATE: SingleClock NUMERIC "1"
139   // Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"
140   // Retrieval info: PRIVATE: WRCONTROL_ACLR_A NUMERIC "0"
141   // Retrieval info: PRIVATE: WidthAddr NUMERIC "5"
142   // Retrieval info: PRIVATE: WidthData NUMERIC "8"
143   // Retrieval info: PRIVATE: rden NUMERIC "0"
144   // Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
145   // Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
146   // Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
```

```
147   // Retrieval info: CONSTANT: INIT_FILE STRING "ram32x8.mif"
148   // Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
149   // Retrieval info: CONSTANT: LPM_HINT STRING "ENABLE_RUNTIME_MOD=NO"
150   // Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
151   // Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "32"
152   // Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT"
153   // Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
154   // Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED"
155   // Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
156   // Retrieval info: CONSTANT: RAM_BLOCK_TYPE STRING "M10K"
157   // Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_A STRING "NEW_DATA_NO_NBE_READ"
158   // Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "5"
159   // Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
160   // Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
161   // Retrieval info: USED_PORT: address 0 0 5 0 INPUT NODEFVAL "address[4..0]"
162   // Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
163   // Retrieval info: USED_PORT: data 0 0 8 0 INPUT NODEFVAL "data[7..0]"
164   // Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL "q[7..0]"
165   // Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL "wren"
166   // Retrieval info: CONNECT: @address_a 0 0 5 0 address 0 0 5 0
167   // Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
168   // Retrieval info: CONNECT: @data_a 0 0 8 0 data 0 0 8 0
169   // Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
170   // Retrieval info: CONNECT: q 0 0 8 0 @q_a 0 0 8 0
171   // Retrieval info: GEN_FILE: TYPE_NORMAL ram32x8.v TRUE
172   // Retrieval info: GEN_FILE: TYPE_NORMAL ram32x8.inc FALSE
173   // Retrieval info: GEN_FILE: TYPE_NORMAL ram32x8.cmp FALSE
174   // Retrieval info: GEN_FILE: TYPE_NORMAL ram32x8.bsf FALSE
175   // Retrieval info: GEN_FILE: TYPE_NORMAL ram32x8_inst.v FALSE
176   // Retrieval info: GEN_FILE: TYPE_NORMAL ram32x8_bb.v FALSE
177   // Retrieval info: LIB_FILE: altera_mf
178
```


8) **task2_control.sv**

```
`timescale 1ns/1ns

// Aidan Lee, Aarin Wen
// 11/13/2024
// EE 371
// Lab 4
//
// Control module for the binary search task for Lab 4. Is in charge of managing states, bounds of the search
// area, etc.
//
// utilizes clk input
// input  reset              resets the search (clears the output) and waits for another start sigal
// input  [7:0] A            represents the desired value that you want to find inside the RAM
// input  start              starts the algorithm. Waits for this value to go to high before starting
// input  [7:0] data_out     the data at address to_check, received by the datapath module
// output found              goes high if the value A was found in the ram
// output notfound           goes high at the end when finished if the value A was not found
// output done               goes high if algorithm is finished
// output to_check           tells the datapath module what address to find the value of next
// output [4:0] loc          represents the address location where the value is found

module task2_control (reset, clk, A, data_out, start, found, notfound, done, to_check, loc);

   input logic reset, clk, start;

   input logic [7:0] A;

   // Data_out - The data out of the RAM module after feeding it in an address
   input logic [7:0] data_out;

   output logic found, notfound, done;
   // The address to check: the middle address between the upper bound and lower bound. 5 bits to match the RAM module
   output logic [4:0] to_check;
   output logic [4:0] loc;

   // upper bounds and lower bounds of the binary search. added an extra bit so that it can hold 32
   logic [5:0] upper_bound, lower_bound;

   // next_bounds - Control signal that reports whether the upper or lower
   // bounds should be adjusted for the next search. 2 bits long so that 2'b11 can
   // signal to stop adjusting bounds
   logic [1:0] next_bounds;

   // control circuit
   logic [2:0] ps, ns;
   parameter S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011, S5 = 3'b100;

    // internal status signal to report the value was not found (INTERNAL ONLY)
    logic change_bounds;

    // state logic
    always_comb begin
        case (ps)
            S1: if (start == 0) ns = S1; // initial state: wait until start signal
                else ns = S2;
            S2: ns = S3; // do algorithm for one clock cycle
            S3: if (data_out == A || notfound) ns = S5; // decide if algorithm should be repeated or if we are done
                else ns = S4;
            S4: ns = S2; // update bounds
            S5: if (start == 0) ns = S1; // done state
                else ns = S5;
        endcase
    end

    // state control ff
    always_ff @(posedge clk) begin
        if (reset)
            ps <= S1;
        else
            ps <= ns;
    end

    // comb logic that assigns control and status signals - prepares what direction
    // next upper and lower bounds should go to if value not found
    always_comb begin
        done = 0; loc = 0; next_bounds = 2'b11;
        case (ps)
            S4: begin
                if (data_out > A) begin
                    next_bounds = 0;
                end
                else if (data_out < A) begin
                    next_bounds = 1;
                end
            end
            S5: begin
                // edge case: if data not found, report 0
                if (!notfound)
                    loc = to_check;

                done = 1;
            end
            default: begin
                done = 0; loc = 0; next_bounds = 2'b11;
            end
        endcase
    end

    // clock logic to make upper and lower bounds assigned on correct clock cycle
    always_ff @(posedge clk) begin
         // default case
         if (reset || start == 0) begin
            upper_bound <= 32;
            lower_bound <= 0;
        end
        // edge case: when value is at upper bound address 31
        else if (change_bounds) begin
            upper_bound <= 33;
            lower_bound <= 31;
        // adjust lower bounds (according to next_bounds)
        end else if (next_bounds == 2'b01) begin
            upper_bound <= upper_bound;
            lower_bound <= to_check + 1;
        // adjust upper bounds (according to next_bounds)
```

```
115         end else if (next_bounds == 2'b00) begin
116            upper_bound <= to_check + 1;
117            lower_bound <= lower_bound;
118         // otherwise, don't touch the bounds. usually for initial run, or when algo finishes
119         end else begin
120            upper_bound <= upper_bound;
121            lower_bound <= lower_bound;
122         end
123     end
124
125     // calculation to determine whether to check the upper half or lower half for binary search
126     assign to_check = ((upper_bound - lower_bound)/2) + lower_bound - 1;
127     // reports that the correct value was found once finished running algorithm
128     assign found = (data_out == A) && done;
129     // signal to change bounds
130     assign change_bounds = ((upper_bound == 32) && (lower_bound == 31));
131     // signal to check that the value was not found (triggers at the end when finished)
132     assign notfound = ((upper_bound - lower_bound) == 1) && (data_out != A);
133
134 endmodule
135
136 module control_tb ();
137
138    logic reset, clk, start;
139
140    logic [7:0] A;
141
142
143    logic [7:0] data_out;
144
145    logic found, notfound, done;
146
147    logic [4:0] to_check;
148    logic [4:0] loc;
149
150    // Clock logic
151    parameter CLOCK_PERIOD = 5; // Increase clock frequency
152    initial begin
153        clk <= 0;
154        forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
155    end
156
157    task2_control dut (.*);
158
159    initial begin
160
161         A <= 8'b00000000; start <= 1'b0; data_out <= 0;
162         reset <= 1; @(posedge clk);
163                     @(posedge clk);
164
165         // check not finding it (should try to go down in address
166         start <= 1'b1; reset <= 0; A <= 8'b00000000; data_out <= 8'd15;
167         repeat (8) @(posedge clk);
168
169         reset <= 1; start <= 1'b0; @(posedge clk);
170                     @(posedge clk);
171
172          // check finding it
173          start <= 1'b1; reset <= 0; A <= 8'b00000000; data_out <= 8'b0;
174          repeat (8) @(posedge clk);
175
176
177
178          $stop;
179     end
180
181 endmodule
```

9) task2_datapath.sv

```
1  `timescale 1ns/1ns
2
3  // Aidan Lee, Aarin Wen
4  // 11/13/2024
5  // EE 371
6  // Lab 4
7  //
8  // Datapath module for the binary search task for Lab 4. Is in charge of storing memory,
9  // and telling the control module what value is at every address it looks for.
10 //
11 // utilizes clk input
12 // input to_check             tells the datapath module what address to find the value of next
13 // output [7:0] data_out      represents the address location where the value is found
14
15 module task2_datapath (to_check, clk, data_out);
16
17    input logic [4:0] to_check;
18    input logic clk;
19    output logic [7:0] data_out;
20
21    // 32x8 ram module, instantiated using the built in methods taught in lab 2
22    // takes in a 5bit address to check the value currently in that address to find the desired value
23    // takes in a clock input
24    // takes in data (doesn't matter. never reading)
25    // takes in wren as a write enable (which is always 0, never going to matter)
26    // outputs 8bit data_out to reveal what's in the ram at the given address
27    ram32x8 ram (.address(to_check), .clock(clk), .data(1'b1), .wren(1'b0), .q(data_out));
28
29 endmodule
30
31 module datapath_tb ();
32
33    logic [4:0] to_check;
34    logic clk;
35    logic [7:0] data_out;
36
37    // Clock logic
38    parameter CLOCK_PERIOD = 5; // Increase clock frequency
39    initial begin
40        clk <= 0;
41        forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
42    end
43
44    task2_datapath dut (.*);
45
46    initial begin
47
48       // check whats in every address
49       for (int i = 0; i < 32; i++) begin
50          to_check = i;
51          @(posedge clk);
52       end;
53       $stop;
54     end
55
56 endmodule
```