Connor Aksama
EE 371
February 12, 2023
Lab 4 Report

Procedure

This lab is comprised of two tasks. The first task involved implementing a bit counting algorithm in hardware using a given ASMD chart. The second task involved designing a binary search algorithm using an ASMD chart, then implementing the algorithm in hardware.

Task 1

I approached this task by first planning out the division between functionality in the controller and the datapath of the hardware implementation. I determined what signals would need to be output from the controller to the datapath, how those signals would affect the data flow, and what feedback would be sent back to the controller. The controller would handle FSM states and transitions, sending signals to the datapath module indicating which state it was currently in. The datapath module would then modify its internally stored data while outputting intermediate results and send feedback to the controller whether the process was complete or not.

After deciding what would be contained within each component of the system, these modules were implemented in Verilog, then simulated and verified using ModelSim.

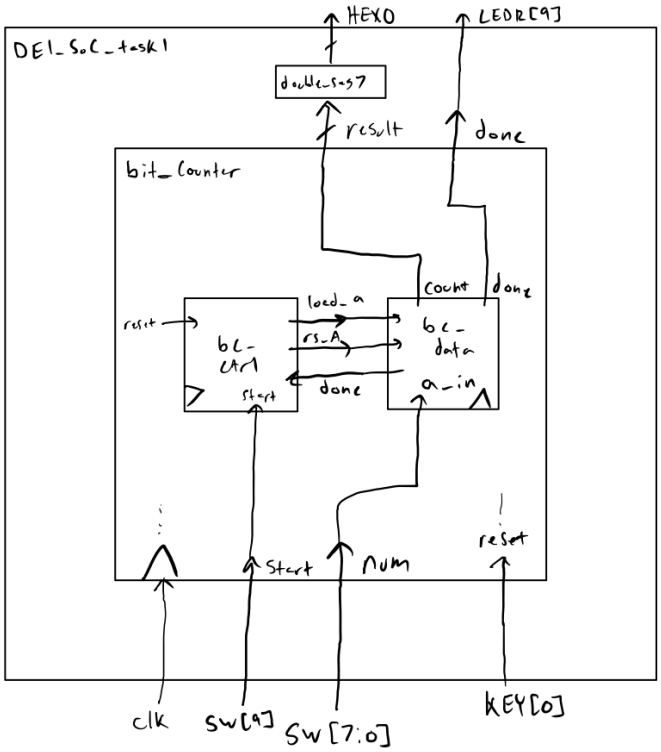


Figure 1. Top-Level Block Diagram for Lab 4 Task 1.

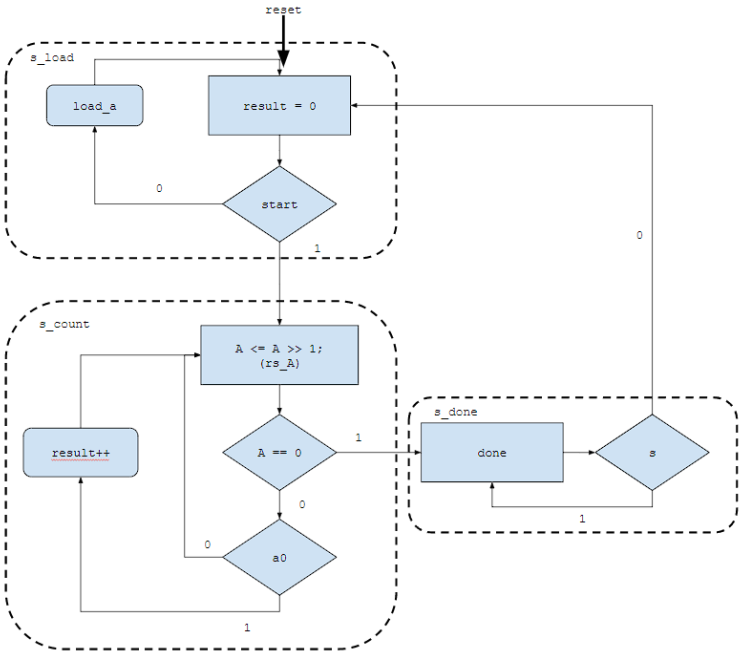


Figure 2. ASMD chart showing the bit counting algorithm.

Task 2

I approached this task by first planning out the high-level algorithm for the binary search. I then drafted an ASM chart to detail each state and the decisions to make at each one. Next, I converted this chart to an ASMD chart by deciding how values would be stored in registers and thinking about when each register would hold a valid value at each state to determine FSM states and timing for transitions between states.

After this ASMD chart was created, similarly to Task 1, distinctions were made between what would belong in a controller module and a datapath module.

These modules were then implemented in Verilog, then tested and simulated in ModelSim before being uploaded and tested on hardware.

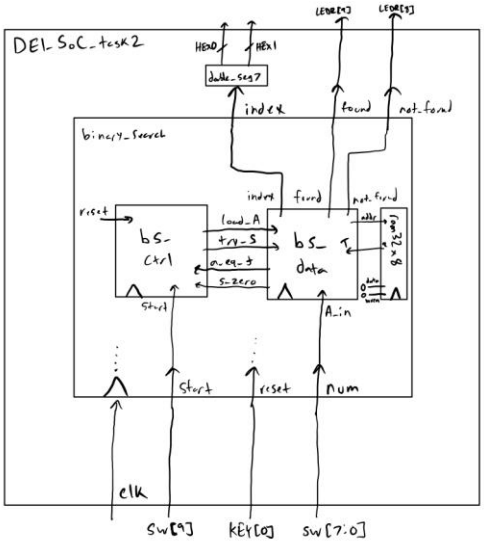


Figure 3. Top-Level Block Diagram for Lab 4 Task 2.

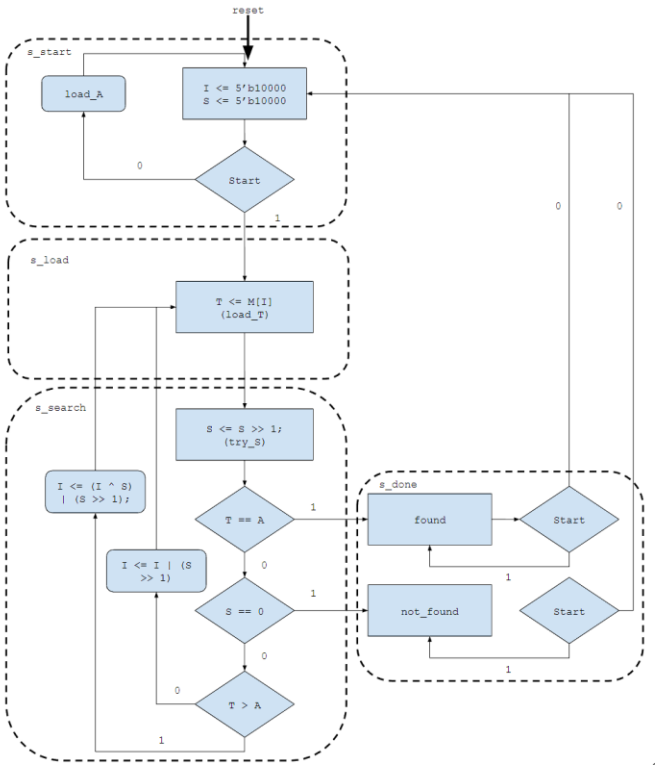


Figure 4. ASMD chart for the Lab 4 Task 2 Design.

Results

Task 1

The following are screenshots from ModelSim simulations for each module used in the bit counter design.

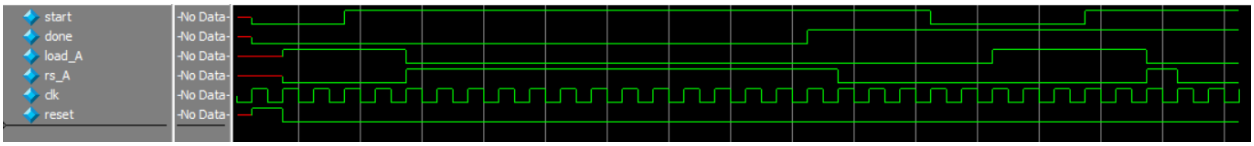


Figure 5. ModelSim Waveform for the Task 1 controller testbench.

This testbench shows the transitions between each of the states in the ASMD chart. After resetting, the system stays in the load state until the start signal is asserted. Following this, the load signal is raised until the done signal is asserted. After this, the controller remains idle until the start signal is lowered, at which point the controller returns to the start state.

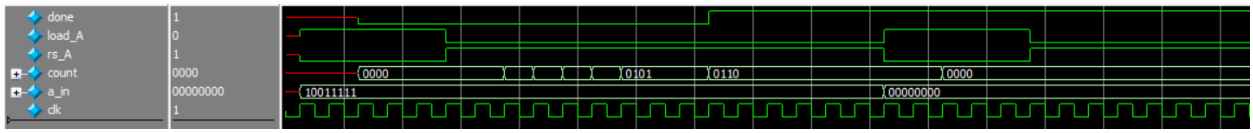


Figure 6. ModelSim Waveform for the Task 1 datapath testbench.

This testbench shows how the internal data of the datapath module is manipulated given external signals from a controller module. While the load signal is asserted, the module samples the input number and holds the count at 0. Once the load signal is low and the right shift signal is high, the process of right shifting and counting bits begins and stops and outputs done once the internal number reaches 0 where it stays until the load signal is asserted again.

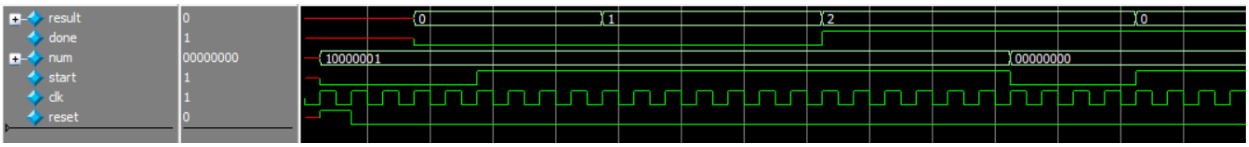


Figure 7. ModelSim Waveform for the Bit Counting module.

This testbench shows the behavior of the design with the controller and datapath modules connected. A number is loaded into the module with the start signal lowered with the result remaining at 0. Once the start signal is asserted, the result begins to increment until the final answer is reached. The system then stays at this state until start is lowered. At this point the next number is loaded in, the result reset, and the process begins again.

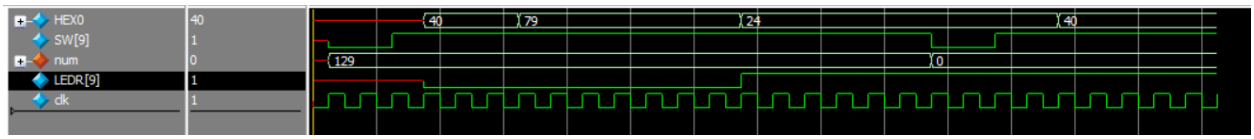


Figure 8. ModelSim Waveform for the Task 1 top-level design module

This testbench shows the correct functionality for the system when connected to the external ports of the board. The testing process is identical to the one shown in Figure 7, except with input and output signals coming from or going to the board's peripherals.

See Figure 13 for the testbench of the double seven-segment display module.

Task 2

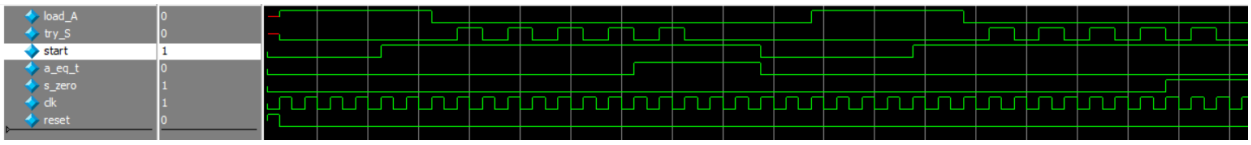


Figure 9. ModelSim Waveform for the Task 2 controller module.

This testbench illustrates the state transitions of the controller module in response to sample feedback from a connected datapath module. On reset, the module outputs the load signal. When the start signal is raised, the controller begins oscillating between a signal to try the current bit index (S) and loading in a value from RAM. Once the equality signal or the zero signal is raised, the controller goes idle until the start signal is lowered.

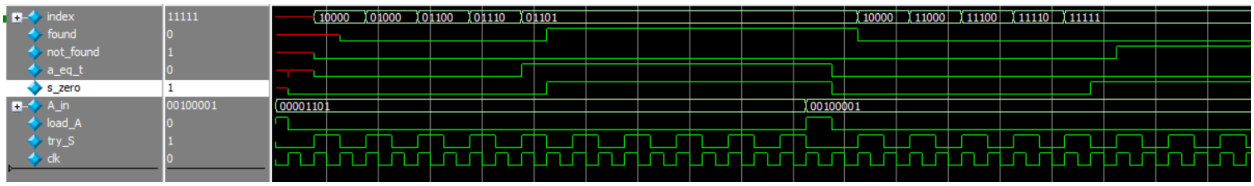


Figure 10. ModelSim Waveform for the Task 2 datapath module.

This testbench shows how data is manipulated in response to signals from a sample connected controller module. On a load signal, the input number is loaded. Then as input signals oscillate between signals to load a value from the connected RAM and to try the current bit index, the output result is shown being updated as the target value is being approached. If the value is found, the module outputs a found signal, and a not found signal otherwise.

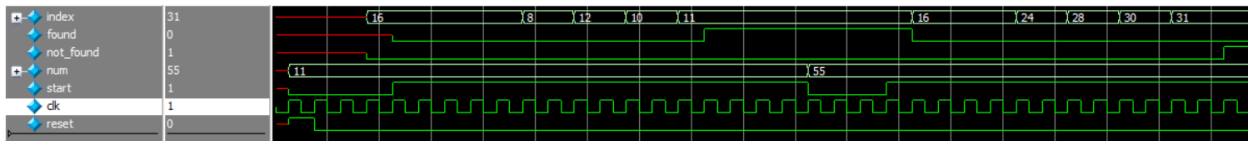


Figure 11. ModelSim Waveform for the binary search module.

This testbench shows the behavior of the design with the controller and datapath modules connected. A number is loaded into the module with the start signal lowered with the result remaining at its start value. Once the start signal is asserted, the result begins to change until the search terminates. The

system then stays at this state until start is lowered. At this point the next number is loaded in, the result reset, and the process begins again.

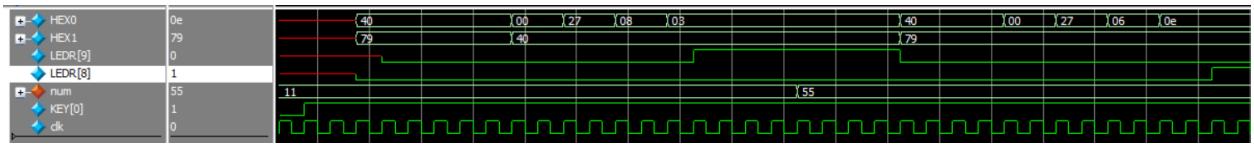


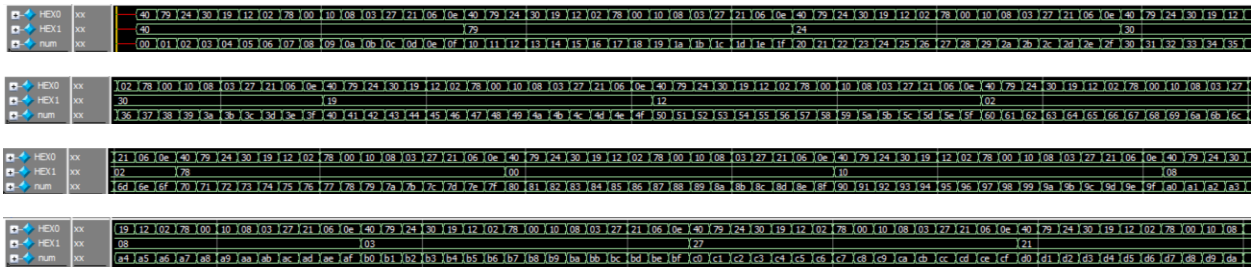
Figure 12. ModelSim Waveform for the Task 2 top-level design module

This testbench shows the correct functionality for the system when connected to the external ports of the board. The testing process is identical to the one shown in Figure 11, except with input and output signals coming from or going to the board's peripherals.

For conciseness, HEX displays are encoded in hexadecimal. Use the following table to convert from hexadecimal code to the corresponding seven segment display:

ModelSim Hexadecimal Code	Seven Segment Display Character
40	0
79	1
24	2
30	3
19	4
12	5
02	6
78	7
00	8
10	9
08	A
03	b
27	c
21	d
06	E
0e	F

Table 1. Conversion between hexadecimal values shown in ModelSim and the character shown on a seven-segment display.



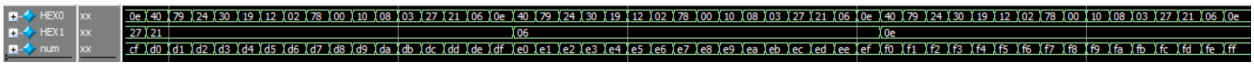


Figure 13. ModelSim Waveform for the Double Seven Segment Display module testbench.

This waveform demonstrates the two digit, hexadecimal display given any number from 0x00 to 0xff. Refer to Table 1 to decode the hexadecimal values to their corresponding characters.

Final Product

The overarching goal of this lab was to gain experience designing, interpreting, and implementing ASMD charts. The main result of Task 1 was a module that implemented a bit counting algorithm using a given ASMD chart. This system would load in an 8-bit number inputted by a user via an array of switches, and when a start signal was triggered with another switch, the number of '1' bits in the number would be shown on the HEX display along with an LED signal indicating whether the process was complete or not.

In Task 2, a binary search algorithm was created using an ASMD chart, then implemented in hardware. This system would take in a target value inputted by a user via an array of switches, and when a start signal was triggered with another switch, the index of the target value in a preprogrammed RAM module would be shown on the HEX display along with LED signals indicating whether the value was successfully found in the RAM or not.

Appendix: SystemVerilog Code

(See next page)

```
/**********************************

bc_ctrl.sv

**********************************/

/*
 * Connor Aksama
 * 02/12/2023
 * CSE 371
 * Lab 4
*/

/**
 * Controller module for the bit counting system.

 * Inputs:
 *      start [1 bit] - Signal to begin the binary searching process. While start is 0 and the algo is not in progress,
 *                      the input value num is loaded into the system. When start is 1, the algo begins. Once the search
 *                      is complete, the next value of num will be loaded once start is lowered to 0.
 *      done [1 bit] - 1 if the algo is complete, 0 o.w.
 *      clk [1 bit] - The clock to use for this module.
 *      reset [1 bit] - Resets the system to its initial state before the search process has started.

 * Outputs:
 *      load_A [1 bit] - 1 if A should be sampled from the user during this cycle, 0 o.w.
 *      rs_A [1 bit] - 1 if the next bit of the num should be counted, 0 o.w.
*/
module bc_ctrl (
    output logic load_A, rs_A
    ,input logic start, done, clk, reset
    );

    typedef enum logic [1:0] { s_load, s_count, s_done } state;

    state ps, ns;

    logic start_reg;

    always_comb begin
        // Handle state transitions
        case (ps)
            s_load: begin
                if (start_reg) begin
                    ns = s_count;
                end else begin
                    ns = s_load;
                end
            end

            s_count: begin
                if (done) begin
                    ns = s_done;
                end else begin
                    ns = s_count;
                end
            end

            s_done: begin
                if (!start_reg) begin
                    ns = s_load;
                end else begin
                    ns = s_done;
                end
            end
```

```
66
67          endcase
68
69          // Control signals
70          load_A = (ps == s_load);
71          rs_A = (ps == s_count);
72
73      end
74
75      always_ff @(posedge clk) begin
76          // Register state, start
77          if (reset) begin
78              ps <= s_load;
79          end else begin
80              ps <= ns;
81          end
82
83          start_reg <= start;
84      end
85
86  endmodule  // bc_ctrl
87
88  /*
89   * Testbench to test the functionality of the bc_ctrl module
90   */
91  module bc_ctrl_testbench();
92
93      logic start, done, clk, reset, load_A, rs_A;
94
95      bc_ctrl dut (.*);
96
97      parameter CLOCK_PERIOD = 100;
98      initial begin
99          clk <= 0;
100         forever #(CLOCK_PERIOD / 2) clk <= ~clk;
101     end
102
103     initial begin
104         integer i;
105         // Hold in start state
106         @(posedge clk) reset <= 1'b1; start <= 1'b0; done <= 1'b0;
107         @(posedge clk); reset <= 1'b0;
108         @(posedge clk);
109
110         // Start algo
111         for (i = 0; i < 15; i++) begin
112             @(posedge clk) start <= 1'b1;
113         end
114
115         // Hold in done state
116         for (i = 0; i < 5; i++) begin
117             @(posedge clk) done <= 1'b1;
118         end
119
120         // Back to start state
121         start <= 1'b0;
122         for (i = 0; i < 5; i++) begin
123             @(posedge clk);
124         end
125
126         // Start algo
127         start <= 1'b1;
128         for (i = 0; i < 5; i++) begin
129             @(posedge clk);
130         end
131
132         $stop;
133     end
134
```

```
135 endmodule  // bc_ctrl_testbench
136
137 /**********************************
138
139 bc_data.sv
140
141 **********************************/
142
143 /*
144  * Connor Aksama
145  * 02/12/2023
146  * CSE 371
147  * Lab 4
148 */
149
150 /**
151  * Datapath module for the bit counting system.
152
153  * Inputs:
154  *      a_in [8 bit] - The number to bit count. Latched when load_A is 1.
155  *      load_A [1 bit] - 1 if a_in should be sampled, 0 o.w.
156  *      rs_A [1 bit] - 1 if the next bit should be counted, 0 o.w.
157  *      clk [1 bit] - The clock to use for this module.
158
159  * Outputs:
160  *      done [1 bit] - 1 if the algo is complete, 0 o.w.
161  *      count [4 bit] - The number of 1s in the sampled a_in number. Valid if and only
     if done is 1.
162 */
163 module bc_data (
164     output logic done
165     ,output logic [3:0] count
166     ,input logic load_A, rs_A, clk
167     ,input logic [7:0] a_in
168     );
169
170     logic [7:0] A;
171     logic [3:0] result;
172
173     always_ff @(posedge clk) begin
174         if (load_A) begin
175             // Sample A
176             result <= 4'b0;
177             A <= a_in;
178         end
179
180         if (rs_A) begin
181             // RS and count
182             A <= A >> 1;
183             if (A[0] == 1'b1) begin
184                 result <= result + 1'b1;
185             end
186         end
187
188         done <= (A == 0);
189         count <= result;
190     end
191
192 endmodule  // bc_data
193
194 /*
195  * Testbench to test the functionality of the bc_data module.
196  */
197 module bc_data_testbench();
198
199     logic done, load_A, rs_A, clk;
200     logic [3:0] count;
201     logic [7:0] a_in;
202
```

```
203       bc_data dut (.*);
204
205       parameter CLOCK_PERIOD = 100;
206       initial begin
207           clk <= 0;
208           forever #(CLOCK_PERIOD / 2) clk <= ~clk;
209       end
210
211       initial begin
212           integer i;
213           // Load value, hold in start
214           @(posedge clk) {load_A, rs_A} <= 2'b10; a_in <= 8'b10011111;
215           @(posedge clk);
216           @(posedge clk);
217           @(posedge clk);
218           @(posedge clk);
219           // Begin algo
220           for (i = 0; i < 15; i++) begin
221               @(posedge clk) {load_A, rs_A} <= 2'b01;
222           end
223           // Load value
224           @(posedge clk) {load_A, rs_A} <= 2'b10; a_in <= 8'b00000000;
225           @(posedge clk);
226           @(posedge clk);
227           @(posedge clk);
228           @(posedge clk);
229           // Begin
230           for (i = 0; i < 15; i++) begin
231               @(posedge clk) {load_A, rs_A} <= 2'b01;
232           end
233
234           $stop;
235       end
236
237   endmodule
238
239   /**********************************
240
241   bit_counter.sv
242
243   **********************************/
244
245   /*
246    * Connor Aksama
247    * 02/12/2023
248    * CSE 371
249    * Lab 4
250   */
251
252   /**
253    * Instantiates the controller and datapath modules and defines connections.
254
255    * Inputs:
256    *      num [8 bit] - Number to bit count.
257    *      start [1 bit] - Signal to begin the bit counting algo. While start is 0 and the
       algo is not in progress,
258    *                      the input value num is loaded into the system. When start is 1,
       the algo begins. Once the algo
259    *                      is complete, the next value of num will be loaded once start is
       lowered to 0.
260    *      clk [1 bit] - The clock to use for this module.
261    *      reset [1 bit] - Resets the system to its initial state before the algo has
       started.
262
263    * Outputs:
264    *  result [4 bit] - The number of 1s in the sampled num. This result value is valid if
       and only if done is raised.
265    *                  If found and not_found are 0, the search is not complete.
266    *  done [1 bit] - 1 if the algo is complete, 0 o.w.
```

```
267  */
268  module bit_counter(
269      output logic [3:0] result
270      ,output logic done
271      ,input logic [7:0] num
272      ,input logic start, clk, reset
273  );
274
275      // Controller module
276      // In: start, done, clk, reset
277      // Out: load_A, rs_A
278      logic load_A, rs_A, d;
279      bc_ctrl controller (
280           .load_A(load_A)
281          ,.rs_A(rs_A)
282          ,.start(start)
283          ,.done(d)
284          ,.clk(clk)
285          ,.reset(reset)
286      );
287
288      // Datapath module
289      // In: load_A, rs_A, a_in, clk
290      // Out: done, count
291      bc_data datapath (
292           .done(d)
293          ,.count(result)
294          ,.load_A(load_A)
295          ,.rs_A(rs_A)
296          ,.a_in(num)
297          ,.clk(clk)
298      );
299
300      assign done = d;
301
302  endmodule  // bit_counter
303
304  /*
305   * Testbench to test the functionality of the bit_counter module.
306   */
307  module bit_counter_testbench();
308
309      logic [3:0] result;
310      logic done;
311      logic [7:0] num;
312      logic start, clk, reset;
313
314      bit_counter dut (.*);
315
316      parameter CLOCK_PERIOD = 100;
317      initial begin
318          clk <= 0;
319          forever #(CLOCK_PERIOD / 2) clk <= ~clk;
320      end
321
322      initial begin
323          integer i;
324          // Load value
325          @(posedge clk) reset <= 1; num <= 8'b10000001; start <= 0;
326          @(posedge clk) reset <= 0;
327          @(posedge clk);
328          @(posedge clk);
329          @(posedge clk);
330          // Start count
331          @(posedge clk) start <= 1;
332
333          for (i = 0; i < 16; i++) begin
334              @(posedge clk);
335          end
```

```
336           // Load value
337           @(posedge clk) start <= 0; num <= 8'b0;
338           @(posedge clk);
339           @(posedge clk);
340           @(posedge clk);
341           // Start count
342           @(posedge clk) start <= 1;
343           for (i = 0; i < 5; i++) begin
344               @(posedge clk);
345           end
346
347           @(posedge clk);
348           @(posedge clk);
349
350           $stop;
351       end
352
353   endmodule  // bit_counter_testbench
354
355   /**********************************
356
357   DE1_SoC_task1.sv
358
359   **********************************/
360
361   /*
362    * Connor Aksama
363    * 02/12/2023
364    * CSE 371
365    * Lab 4
366   */
367
368   /**
369    * Top-level module for Lab 4 Task 1. Instantiates binary search module and connects I/O
       to peripherals.
370
371    * Inputs:
372    *  SW [10 bit] - The 10 onboard switches, respectively.
373    *  KEY [4 bit] - The 4 onboard keys, respectively.
374    *  CLOCK_50 [1 bit] - The system clock to use for this module.
375
376    * Outputs:
377    *  HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
       display format.
378    *  LEDR [10 bit] - Signal to output to 10 onboard LEDs, respectively.
379   */
380   module DE1_SoC_task1 (
381       output logic [6:0] HEX0
382       ,output logic [9:0] LEDR
383       ,input logic [3:0] KEY
384       ,input logic [9:0] SW
385       ,input logic CLOCK_50
386       );
387
388       // Bit counting module
389       // In: num, start, clk, reset
390       // Out: result, done
391       logic [3:0] result;
392       bit_counter bc (
393            .result(result)
394           ,.num(SW[7:0])
395           ,.done(LEDR[9])
396           ,.start(SW[9])
397           ,.clk(CLOCK_50)
398           ,.reset(~KEY[0])
399       );
400
401       // Seven-Segment display
402       // In: num
```

```
403         // Out: HEX0, HEX1(unused)
404         double_seg7 res_display (.HEX0(HEX0), .HEX1(), .num({4'b0, result}));
405
406     endmodule  // DE1_SoC_task1
407
408     /*
409      * Testbench to test the functionality of the DE1_SoC_task1 module.
410      */
411     module DE1_SoC_task1_testbench();
412
413         logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
414         logic [9:0] LEDR;
415         logic [3:0] KEY;
416         logic [9:0] SW;
417         logic CLOCK_50, clk;
418
419         assign CLOCK_50 = clk;
420
421          DE1_SoC_task1 dut (.*);
422
423         parameter CLOCK_PERIOD = 100;
424         initial begin
425             clk <= 0;
426             forever #(CLOCK_PERIOD / 2) clk <= ~clk;
427         end
428
429         initial begin
430             integer i;
431
432             // Load value
433             @(posedge clk) KEY[0] <= 1'b0; SW[7:0] <= 8'b10000001; SW[9] <= 0;
434             @(posedge clk); KEY[0] <= 1'b1;
435             // Start count
436             @(posedge clk) SW[9] <= 1;
437
438             for (i = 0; i < 16; i++) begin
439                 @(posedge clk);
440             end
441             // Load value
442             @(posedge clk) SW[9] <= 0; SW[7:0] <= 8'b0;
443             @(posedge clk);
444             // Start count
445             @(posedge clk) SW[9] <= 1;
446             for (i = 0; i < 5; i++) begin
447                 @(posedge clk);
448             end
449
450             @(posedge clk);
451             @(posedge clk);
452
453             $stop;
454         end
455
456     endmodule  // DE1_SoC_task1_testbench
457
458     /*********************************
459
460     double_seg7.sv
461
462     *********************************/
463
464     /*
465      * Connor Aksama
466      * 02/12/2023
467      * CSE 371
468      * Lab 4
469     */
470
471     /**
```

```
472    * Defines data for 2-digit hexadecimal HEX display given a 8-bit unsigned integer
473
474    * Inputs:
475    *      num [8 bit] - An unsigned integer value [0x0-0x7F] to display
476
477    * Outputs:
478    *  HEX0 [7 bit] - A HEX display for the least significant digit of the input num,
       formatted in standard seven segment display format
479    *  HEX1 [7 bit] - A HEX display for the most significant digit of the input num,
       formatted in standard seven segment display format
480   */
481   module double_seg7(
482       output logic [6:0] HEX0, HEX1
483       ,input logic [7:0] num
484       );
485
486       // Drive HEX output signals given num
487       always_comb begin
488           // Light HEX0 using LSD of num
489           case (num[3:0])
490               //     Light: 6543210
491                0: HEX0 = ~7'b0111111;  // 0
492                1: HEX0 = ~7'b0000110;  // 1
493                2: HEX0 = ~7'b1011011;  // 2
494                3: HEX0 = ~7'b1001111;  // 3
495                4: HEX0 = ~7'b1100110;  // 4
496                5: HEX0 = ~7'b1101101;  // 5
497                6: HEX0 = ~7'b1111101;  // 6
498                7: HEX0 = ~7'b0000111;  // 7
499                8: HEX0 = ~7'b1111111;  // 8
500                9: HEX0 = ~7'b1101111;  // 9
501               10: HEX0 = ~7'b1110111;  // A
502               11: HEX0 = ~7'b1111100;  // b
503               12: HEX0 = ~7'b1011000;  // c
504               13: HEX0 = ~7'b1011110;  // d
505               14: HEX0 = ~7'b1111001;  // E
506               15: HEX0 = ~7'b1110001;  // F
507               default: HEX0 = 7'bX;
508           endcase
509
510           // Light HEX1 using MSD of num
511           case (num >> 4)
512               //     Light: 6543210
513                0: HEX1 = ~7'b0111111;  // 0
514                1: HEX1 = ~7'b0000110;  // 1
515                2: HEX1 = ~7'b1011011;  // 2
516                3: HEX1 = ~7'b1001111;  // 3
517                4: HEX1 = ~7'b1100110;  // 4
518                5: HEX1 = ~7'b1101101;  // 5
519                6: HEX1 = ~7'b1111101;  // 6
520                7: HEX1 = ~7'b0000111;  // 7
521                8: HEX1 = ~7'b1111111;  // 8
522                9: HEX1 = ~7'b1101111;  // 9
523               10: HEX1 = ~7'b1110111;  // A
524               11: HEX1 = ~7'b1111100;  // b
525               12: HEX1 = ~7'b1011000;  // c
526               13: HEX1 = ~7'b1011110;  // d
527               14: HEX1 = ~7'b1111001;  // E
528               15: HEX1 = ~7'b1110001;  // F
529               default: HEX1 = 7'bX;
530           endcase
531
532       end
533
534   endmodule  // double_seg7
535
536   /*
537    * Tests the functionality of the double_seg7 module.
538   */
```

```
539  module double_seg7_testbench();
540
541      logic [6:0] HEX0, HEX1;
542      logic [7:0] num;
543
544      double_seg7 dut (.HEX0, .HEX1, .num);
545
546      initial begin
547
548          integer i;
549
550          // Check HEX displays for integers 0x0-0xff
551          for (i = 0; i <= 8'hFF; i++) begin
552              #10 num = i;
553          end
554          #50;
555      end
556
557  endmodule  // double_seg7_testbench
558
559  /**********************************
560
561  binary_search.sv
562
563  **********************************/
564
565  /*
566   * Connor Aksama
567   * 02/12/2023
568   * CSE 371
569   * Lab 4
570  */
571
572  /**
573   * Instantiates the controller and datapath modules and defines connections.
574
575   * Inputs:
576   *      num [8 bit] - An unsigned integer value to search for in the system's RAM.
577   *      start [1 bit] - Signal to begin the binary searching process. While start is 0
       and the search is not in progress,
578   *                      the input value num is loaded into the system. When start is 1,
       the search begins. Once the search
579   *                      is complete, the next value of num will be loaded once start is
       lowered to 0.
580   *      clk [1 bit] - The clock to use for this module.
581   *      reset [1 bit] - Resets the system to its initial state before the search process
       has started.
582
583   * Outputs:
584   *  index [5 bit] - The index of the given num in the system's RAM. This index value is
       valid if and only if found is raised.
585   *                  If found and not_found are 0, the search is not complete.
586   *  found [1 bit] - 1 if the search is complete and num was found in the system's RAM. 0
       otherwise.
587   *  not_found [1 bit] - 1 if the search is complete and num was not found in the
       system's RAM. 0 otherwise.
588  */
589  module binary_search (
590      output logic [4:0] index
591      ,output logic found, not_found
592      ,input logic [7:0] num
593      ,input logic start, clk, reset
594      );
595
596      // Controller Module
597      // In: a_eq_t, s_zero, start, clk, reset
598      // Out: load_A, try_S
599      logic load_A, try_S, a_eq_t, s_zero;
600      bs_ctrl controller (
```

```
601             .load_A(load_A)
602            ,.try_S(try_S)
603            ,.start(start)
604            ,.a_eq_t(a_eq_t)
605            ,.s_zero(s_zero)
606            ,.clk(clk)
607            ,.reset(reset)
608        );
609
610        // Datapath module
611        // In: A_in (user input), load_A, try_S (from controller), clk
612        // Out: index (answer), found, not_found (done & success?)
613        bs_data datapath (
614             .index(index)
615            ,.found(found)
616            ,.not_found(not_found)
617            ,.a_eq_t(a_eq_t)
618            ,.s_zero(s_zero)
619            ,.A_in(num)
620            ,.load_A(load_A)
621            ,.try_S(try_S)
622            ,.clk(clk)
623        );
624
625    endmodule  // binary_search
626
627    /*
628     * The testbench module to test the functionality of the binary_search module.
629    */
630    `timescale 1 ps / 1 ps
631    module binary_search_testbench();
632
633        logic [4:0] index;
634        logic found, not_found;
635        logic [7:0] num;
636        logic start, clk, reset;
637
638        binary_search dut (.*);
639
640        parameter CLOCK_PERIOD = 100;
641        initial begin
642            clk <= 0;
643            forever #(CLOCK_PERIOD / 2) clk <= ~clk;
644        end
645
646        initial begin
647            integer i;
648
649            @(posedge clk) reset <= 1'b1; start <= 1'b0; num <= 11;
650            @(posedge clk) reset <= 1'b0;
651            @(posedge clk);
652            @(posedge clk);
653            @(posedge clk) start <= 1'b1;
654
655            for (i = 0; i < 15; i++) begin
656                @(posedge clk);
657            end
658
659            @(posedge clk) start <= 1'b0; num <= 55;
660            @(posedge clk);
661            @(posedge clk);
662            @(posedge clk) start <= 1'b1;
663
664            for (i = 0; i < 15; i++) begin
665                @(posedge clk);
666            end
667            $stop;
668        end
669
```

```
endmodule  // binary_search_testbench

/**********************************

bs_ctrl.sv

**********************************/

/*
 * Connor Aksama
 * 02/12/2023
 * CSE 371
 * Lab 4
*/

/**
 * Controller module for the binary search system.

 * Inputs:
 *      start [1 bit] - Signal to begin the binary searching process. While start is 0 and the search is not in progress,
 *                      the input value num is loaded into the system. When start is 1, the search begins. Once the search
 *                      is complete, the next value of num will be loaded once start is lowered to 0.
 *      a_eq_t [1 bit] - 1 if the search is complete and the number was found in RAM, 0 o.w.
 *      s_zero [1 bit] - 1 if the search is complete and the number was not found in RAM, 0 o.w.
 *      clk [1 bit] - The clock to use for this module.
 *      reset [1 bit] - Resets the system to its initial state before the search process has started.

 * Outputs:
 *      load_A [1 bit] - 1 if A should be sampled from the user during this cycle, 0 o.w.
 *      try_S [1 bit] - 1 if the next bit index of the RAM address should be tested during this cycle, 0 o.w.
*/
module bs_ctrl (
    output logic load_A, try_S
    ,input logic start, a_eq_t, s_zero, clk, reset
    );

    typedef enum logic [1:0] { s_start, s_load, s_search, s_done } state;

    state ps, ns;

    logic start_reg;

    always_comb begin
        // Handle state transitions
        case (ps)

            s_start: begin
                if (start_reg) begin
                    ns = s_load;
                end else begin
                    ns = s_start;
                end
            end

            s_load: begin
                ns = s_search;
            end

            s_search: begin
                if (a_eq_t | s_zero) begin
                    ns = s_done;
                end else begin
```

```
                    ns = s_load;
                end
            end

            s_done: begin
                if (~start_reg) begin
                    ns = s_start;
                end else begin
                    ns = s_done;
                end
            end

        endcase

        // Control signals to datapath
        load_A = (ps == s_start);

        try_S = (ps == s_search);
    end

    always_ff @(posedge clk) begin
        // Register state and start input
        if (reset) begin
            ps <= s_start;
        end else begin
            ps <= ns;
        end

        start_reg <= start;
    end

endmodule  // bs_ctrl

/*
 * Module to test the functionality of the bs_ctrl module
 */
module bs_ctrl_testbench();

     logic load_A, try_S;
    logic start, a_eq_t, s_zero, clk, reset;

     bs_ctrl dut (.*);

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

     initial begin
         integer i;

          // Hold in start state
          start <= 1'b0; a_eq_t <= 1'b0; s_zero <= 1'b0; reset <= 1'b1;
          @(posedge clk) reset <= 1'b0;
          @(posedge clk);
          @(posedge clk);
          @(posedge clk);
          // Begin the search process
          @(posedge clk) start <= 1'b1;

          for (i = 0; i < 10; i++) begin
              @(posedge clk);
          end
          // Search success
          a_eq_t <= 1'b1;
          // Hold in done state
          for (i = 0; i < 5; i++) begin
              @(posedge clk);
```

```
            end
            // Go back to start
            start <= 1'b0; a_eq_t <= 1'b0; s_zero <= 1'b0;
            for (i = 0; i < 5; i++) begin
                @(posedge clk);
            end
            // Search
            @(posedge clk) start <= 1'b1;

            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end
            // Search failure
            s_zero <= 1'b1;
            for (i = 0; i < 5; i++) begin
                @(posedge clk);
            end

            $stop;
        end

    endmodule  // bs_ctrl_testbench

    /**********************************

    bs_data.sv

    **********************************/

    /*
     * Connor Aksama
     * 02/12/2023
     * CSE 371
     * Lab 4
    */

    /**
     * Datapath module for the binary search system.

     * Inputs:
     *      A_in [8 bit] - The number to search for in the RAM. Latched when load_A is true
     *      load_A [1 bit] - 1 if A_in should be sampled, 0 o.w.
     *      try_S [1 bit] - 1 if the next bit in the index should be tested, 0 o.w.
     *      clk [1 bit] - The clock to use for this module.

     * Outputs:
     *      index [5 bit] - The index of the sampled A_in value in the RAM. Valid if and
     only if found is 1.
     *      a_eq_t [1 bit] - Feedback if the sampled A_in value is equal to the currently
     read value from RAM. 1 if eq, 0 o.w.
     *      s_zero [1 bit] - Feedback if all bits have been tested in the index. 1 if true,
     0 o.w.
     *      found [1 bit] - 1 if the search is complete and the sampled A_in was found in
     RAM, 0 o.w.
     *      not_found [1 bit] - 1 if the search is complete and the sampled A_in was not
     found in RAM, 0 o.w.
    */
    module bs_data (
        output logic [4:0] index
        ,output logic a_eq_t, s_zero, found, not_found
        ,input logic [7:0] A_in
        ,input logic load_A, try_S, clk
        );

        logic [4:0] I, S;
        logic [7:0] T, A;

        // RAM block
        // In: address/I(ndex), clk
```

```
865       // Out: data/wren - unused; q - data
866       ram32x8 ram (
867            .address(I)
868           ,.clock(clk)
869           ,.data(8'b0)
870           ,.wren(1'b0)
871           ,.q(T)
872       );
873
874       assign a_eq_t = (A == T);
875       assign s_zero = (S == '0);
876
877       always_ff @(posedge clk) begin
878
879           if (load_A) begin
880               S <= 5'b10000;
881               I <= 5'b10000;
882               A <= A_in;
883           end
884
885           if (try_S) begin
886               S <= S >> 1;
887               if (T > A) begin
888                   I <= (I ^ S) | (S >> 1);
889               end else if (T < A) begin
890                   I <= I | (S >> 1);
891               end
892           end
893
894           index <= I;
895           found <= (A == T);
896           not_found <= (A != T) & (S == 0);
897       end
898
899   endmodule  // bs_data
900
901   /*
902    * Testbench to test the functionality of the bs_data module
903    */
904   `timescale 1 ps / 1 ps
905   module bs_data_testbench();
906
907       logic [4:0] index;
908       logic found, not_found, a_eq_t, s_zero;
909       logic [7:0] A_in;
910       logic load_A, try_S, clk;
911
912       bs_data dut (.*);
913
914       parameter CLOCK_PERIOD = 100;
915       initial begin
916           clk <= 0;
917           forever #(CLOCK_PERIOD / 2) clk <= ~clk;
918       end
919
920       initial begin
921           integer i;
922
923           // Load value
924           A_in <= 13; load_A <= 1'b1; try_S <= 1'b0;
925
926           // Search
927           for (i = 0; i < 10; i++) begin
928               @(posedge clk); load_A <= 1'b0; try_S <= 1'b0;
929               @(posedge clk); load_A <= 1'b0; try_S <= 1'b1;
930           end
931
932           // Load
933           @(posedge clk) A_in <= 33; load_A <= 1'b1; try_S <= 1'b0;
```

```
934             // Search
935             for (i = 0; i < 10; i++) begin
936                 @(posedge clk); load_A <= 1'b0; try_S <= 1'b0;
937                 @(posedge clk); load_A <= 1'b0; try_S <= 1'b1;
938             end
939 
940             $stop;
941 
942         end
943 
944     endmodule  // bs_data_testbench
945 
946     /**********************************
947 
948     DE1_SoC_task2.sv
949 
950     **********************************/
951 
952     /*
953      * Connor Aksama
954      * 02/12/2023
955      * CSE 371
956      * Lab 4
957     */
958 
959     /**
960      * Top-level module for Lab 4 Task 2. Instantiates binary search module and connects I/O
       to peripherals.
961 
962      * Inputs:
963      *  SW [10 bit] - The 10 onboard switches, respectively.
964      *  KEY [4 bit] - The 4 onboard keys, respectively.
965      *  CLOCK_50 [1 bit] - The system clock to use for this module.
966 
967      * Outputs:
968      *  HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
       display format.
969      *  HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
       display format.
970      *  LEDR [10 bit] - Signal to output to 10 onboard LEDs, respectively.
971     */
972     module DE1_SoC_task2 (
973         output logic [6:0] HEX0, HEX1
974         ,output logic [9:0] LEDR
975         ,input logic [9:0] SW
976         ,input logic [3:0] KEY
977         ,input logic CLOCK_50
978         );
979 
980          assign LEDR[7:0] = 8'b0;
981 
982         // Seven-Segment display
983         // In: num
984         // Out: HEX0, HEX1
985         logic [7:0] index;
986         double_seg7 addr (
987               .HEX0(HEX0)
988              ,.HEX1(HEX1)
989              ,.num(index)
990         );
991 
992         // Binary search module
993         // In: num (switch input), start (switch), clk, reset (key)
994         // Out: index (final answer), found (success), not_found (failure)
995         binary_search bs (
996              .index(index)
997             ,.found(LEDR[9])
998             ,.not_found(LEDR[8])
999             ,.num(SW[7:0])
```

```
        ,.start(SW[9])
        ,.clk(CLOCK_50)
        ,.reset(~KEY[0])
    );

endmodule  // DE1_SoC_task2

/*
 * Testbench to test the functionality of the DE1_SoC_task2 module
 */
`timescale 1 ps / 1 ps
module DE1_SoC_task2_testbench();

    logic [6:0] HEX0, HEX1;
    logic [9:0] LEDR;
    logic [9:0] SW;
    logic [3:0] KEY;
    logic CLOCK_50, clk;

     assign CLOCK_50 = clk;

     DE1_SoC_task2 dut (.*);

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

     initial begin
        integer i;

        // Load num
        @(posedge clk) KEY[0] <= 1'b0; SW[9] <= 1'b0; SW[7:0] <= 8'b00001011;
        @(posedge clk) KEY[0] <= 1'b1;
        @(posedge clk);
        @(posedge clk);
        // Start
        @(posedge clk) SW[9] <= 1'b1;
        // Search
        for (i = 0; i < 15; i++) begin
            @(posedge clk);
        end

        // Load
        @(posedge clk) SW[9] <= 1'b0; SW[7:0] <= 8'b00110111;
        @(posedge clk);
        @(posedge clk);
        // Start
        @(posedge clk) SW[9] <= 1'b1;
        // Search
        for (i = 0; i < 15; i++) begin
            @(posedge clk);
        end
        $stop;
    end

endmodule  // DE1_SoC_task2_testbench
```