Jiovany Soliman, 1951338

Chloe' Miller, 2031648

EE/CSE 371 Lab 4 Report: Implementing Algorithms in Hardware

November 06, 2023

Design Procedure

The purpose of this lab was to implement algorithms in hardware by translating ASMD charts to code in SystemVerilog. We first implemented a circuit that counts the number of bits set to 1 in an n-bit input A by combining the necessary datapath components and a control circuit FSM. This circuit also displayed the number of 1's counted in A on the 7-segment HEX displays and lit an LED to indicate whenever the algorithm had finished. We then implemented a binary search algorithm, which searches through an unsigned sorted array to locate an 8-bit value A once the user presses Start. Rather than compare each value in the array to the target value, the algorithm simply compares the target value to the middle value between two bounds of the array and determines if it needs to keep searching in the first or second half of the array, repeatedly comparing the target value to the middle value between the two bounds. Lastly, we combined the functionalities of these two algorithms by adding a switch that the user could toggle to run either the bit-counting or the binary search algorithm.

Task #1:

For task 1, we referenced the provided ASM chart to identify our control, status, and external input signals. Then, we added to the ASM chart these signals and RTL to develop the ASMD chart needed for implementing the control and datapath in SystemVerilog. This task was straight-forward and did not take much time to code.

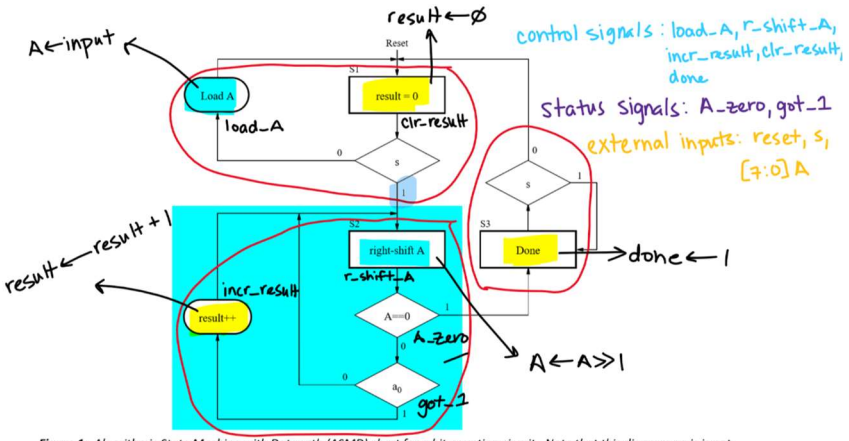


Figure 1: *Algorithmic State Machine with Datapath (ASMD) chart for a bit-counting circuit. Note that this diagram as-is is not sufficient for your lab report; please follow the ASMD chart style shown in lecture.*

Figure 1: ASMD chart for task 1

```
/*
   Module to implmement the control path of task 1.
   External inputs (inputs): clk, reset, s
   Status signals (inputs): A_zero (true if A = 0, false otherwise), got_1 (true if A[0] = 1, false otherwise)
   Control signals (outputs): load_A, r_shift_A, incr_result, clr_result, done
*/

module counter_cntrl(clk, reset, s, A_zero, got_1, load_A, r_shift_A, incr_result, clr_result, done);

   // port definitions
   input  logic clk, reset, s, A_zero, got_1;
   output logic load_A, r_shift_A, incr_result, clr_result, done;

   // define state names and variables
   enum {S1, S2, S3} ps, ns;

   // controller logic with synchronous reset
   always_ff @(posedge clk)
      if(reset)
         ps <= S1;
      else
         ps <= ns;

   // next state logic
   always_comb
      case(ps)
         S1:   ns = s ? S2 : S1;
         S2:   ns = A_zero ? S3 : S2;
         S3:   ns = s ? S3 : S1;
      endcase

   // output assignments
   assign load_A      = (ps == S1) & ~s;
   assign r_shift_A   = (ps == S2);
   assign incr_result = (ps == S2) & got_1;
   assign clr_result  = (ps == S1);
   assign done        = (ps == S3);

endmodule // counter_cntrl
```

Figure 2: Snippet of module counter_cntrl showing the control path of task 1

```
/*
   External inputs (inputs): clk, A
   Control signals (inputs): load_A, r_shift_A, incr_result, clr_result
   Status signals (outputs): A_zero (true if A = 0, false otherwise), got_1 (true if A[0] = 1, false otherwise)
   External outputs (outputs): result
*/

module counter_datapath(clk, A, load_A, r_shift_A, incr_result, clr_result, A_zero, got_1, result);

   // port definitions
   input  logic clk, load_A, r_shift_A, incr_result, clr_result;
   input  logic [7:0] A;
   output logic [3:0] result; // highest value counter can be is 8
   output logic A_zero, got_1;

   logic [7:0] a;

   // datapath logic
   always_ff @(posedge clk) begin
      if(load_A)
         a <= A;

      if(clr_result)
         result <= 0;
      else if(incr_result)
         result <= result + 1;
      else
         result <= result;

      if(r_shift_A)
         a <= (a >> 1);
   end

   assign A_zero = (a == 8'b0);
   assign got_1 = (a[0] == 1'b1);

endmodule // counter_datapath
```

Figure 3: Snippet of module counter_datapath showing the data path of task 1

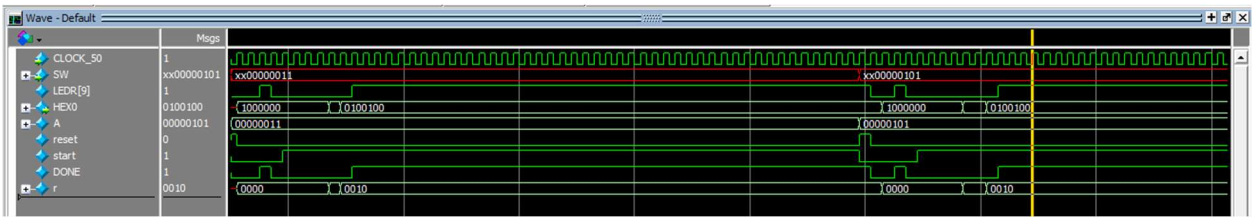


Figure 4: Snippet of task 1 model sim showing result "r" equals 2 reflecting the correct number of bits set to 1 in A.

In ModelSim, we tested our bit-counter algorithm by assigning different values to 8-bit input A, and monitored the result “r”, which indicated how many bits were set to 1 in our input. We can see in the waveform above that “r” provides the correct number of 1 bits for both of our inputs of A = 3 and A = 5 in binary, respectively.

Task #2:

For task 2, we needed to instantiate a 32x8 1-port RAM using a .mif file that we initialized with 32 sorted unsigned values. We then had to implement a binary search algorithm in hardware. We started by drafting an ASMD chart. The first step was to understand our control path that would later be implemented in code using a finite state machine. After we established our control path portion of the ASMD diagram, we started writing down our RTL, which would then be used to establish and implement our data path. It was very helpful to start by drawing the ASMD chart as it clarified our states, control signals, status signals, and finally our RTL. For the algorithm portion, we utilized the commonly used binary search algorithm that’s available at a variety of sources online. The main idea is that you start by checking the middle value of your search space to see if it matches your target value. Afterwards, since the search space is assumed to be sorted, you can simply determine if the target value is before the middle value or after the middle value. Once you identify which half of the search space your value is predicted to be in, you update the edges/limits of your search space to only reflect that new half. Afterwards you again select the middle value of that search space and compare it to your target value. You repeat these steps recursively until you either find your target value at which point you break out of the loop, or until you exhaust the entirety of your search space; which can be checked by comparing the 2 limits of your search space to each other. Finally, we had to include 3 additional states for timing purposes.

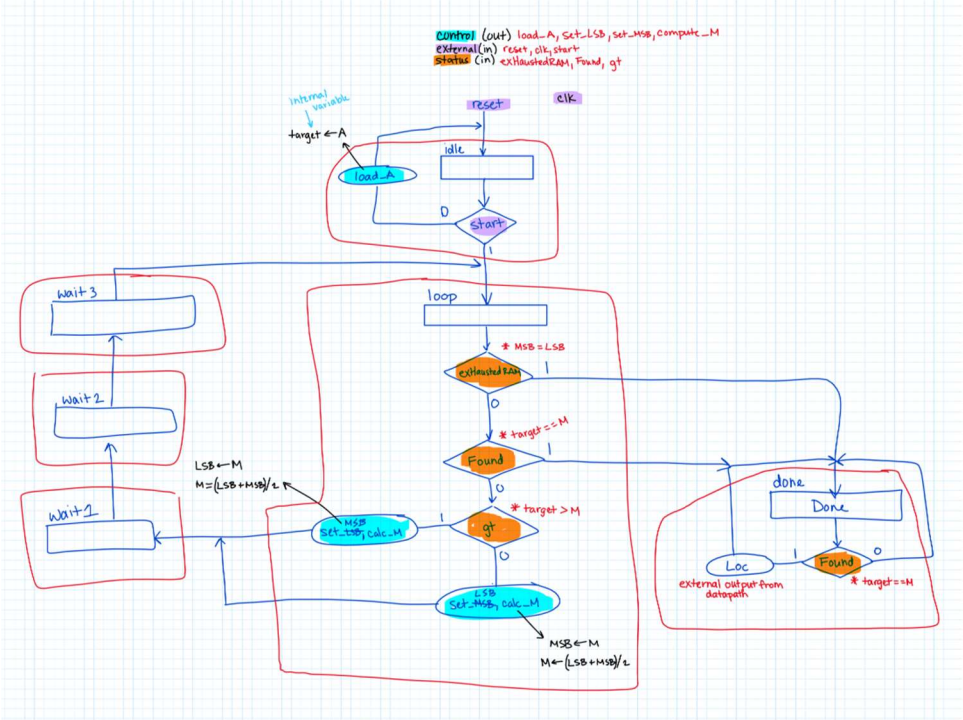


Figure 5: ASMD chart for task 2

```
// Module to implement the finite state machine (control path) portion of task 2, the binary search algorithm.
// CLOCK_50, Reset, Start, exhaustedRAM, Found, gt are single bit inputs.
// Compute_M, Set_LSB, Set_MSB, init are single bit outputs.
`timescale 1 ps / 1 ps
module binaryFSM (CLOCK_50, Reset, Start, exhaustedRAM, Found, gt, Compute_M, Set_LSB, Set_MSB, init);
    // external inputs
    input logic CLOCK_50;
    input logic Reset, Start;

    // status signals
    input logic exhaustedRAM, Found, gt;

    // control signals
    output logic Compute_M, Set_LSB, Set_MSB, init;

    // define enum states
    enum { idle, loop, done, wait1, wait2, wait3 = 6 } ps, ns;

    // next state logic
    always_comb begin
        case(ps)

            idle: ns = Start ? loop : idle;
            loop: begin
                    if(exhaustedRAM)
                        ns = done;
                    else if(~exhaustedRAM & Found)
                        ns = done;
                    else // (~exhaustedRAM & ~Found)
                        ns = wait1;
                end
            wait1: ns = wait2;
            wait2: ns = wait3;
            wait3: ns = loop;
            done: ns = done; // stay here until user presses Reset
            default: ns = idle;
        endcase
    end

    // controller logic with synchronous reset
    always_ff @(posedge CLOCK_50) begin
        if(Reset)
            ps <= idle;
        else
            ps <= ns;
    end

    // output assignments
    assign Compute_M = (ps == loop) & (gt | ~ gt);
    assign Set_MSB = (ps == loop) & ~gt;
    assign Set_LSB = (ps == loop) & gt;
    assign init =  (ps == idle) & ~Start;

endmodule // binaryFSM
```

Figure 6: Snippet of module binaryFSM showing the control path for task 2

```
// Module to implement the data path portion of task 2; binary data search.
// CLOCK_50, Compute_M, Set_LSB, Set_MSB, init are single bit inputs.
// A is an 8 bit input
// Loc is a 5 bit output
// DONE is a single bit output
`timescale 1 ps / 1 ps
module binaryDataPath (CLOCK_50, A, Compute_M, Set_LSB, Set_MSB, init, exhaustedRAM, gt, Found, Loc, DONE);

    // external inputs
    input logic CLOCK_50;
    input logic [7:0] A;

    // control signals (inputs)
    input logic Compute_M, Set_LSB, Set_MSB, init;

    // status signals (outputs)
    output logic exhaustedRAM, gt, Found; // Found will also be an external output

    // external outputs
    output logic [4:0] Loc;
    output logic DONE;

    // internal variables
    logic [4:0] LSB, MSB, M;
    logic [7:0] q;
    logic [7:0] target;

    // datapath logic
    always_ff @(posedge CLOCK_50) begin
        if (init) begin
            LSB <= 0;
            M <= 5'd15;
            MSB <= 5'd31;
            target <= A;
        end

        if (Compute_M)
            M <= (MSB + LSB) / 2;
        if (Set_LSB)
            LSB <= M + 1'b1;
        if (Set_MSB)
            MSB <= M - 1'b1;
    end

    // retrieve value to compare
    RAM_32_8_1port RAM (.address(M), .clock(CLOCK_50), .data(0), .wren(0), .q(q));

    assign exhaustedRAM = (MSB == LSB);
    assign gt = (target > q);
    assign Found = (target == q);
    assign DONE = (MSB == LSB) | Found; // can be DONE without having found the target
    assign Loc = M;

endmodule // binaryDataPath
```

Figure 7: Snippet of module BinaryDataPath showing the data path for task 2

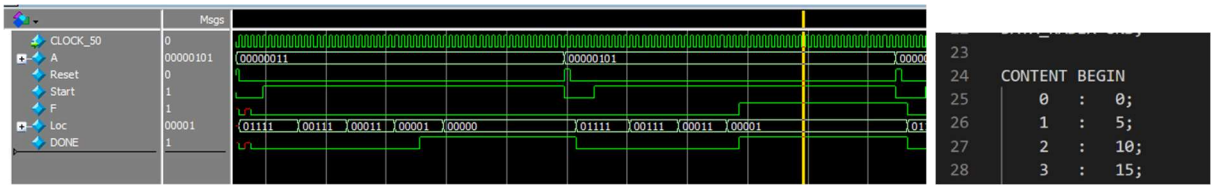


Figure 8: Snippet of binary top-level model sim showing signal F as "1" at the correct .mif file index "1" where A = 5.

In ModelSim, we tested our binary search algorithm by assigning A a value and seeing if our algorithm could find a match to our target value. We initialized our RAM with multiples of 5, as seen in the snippet above to the right. We first tested a value that would not be in the RAM, which was A = 3, and we can see that F (Found) does not become asserted because it is never found in RAM, however DONE still asserts because it indicates that the algorithm has finished comparing all the elements in RAM to the target value. Next, we tested a value that was in RAM (A = 5), and we see that both Found and DONE are asserted as soon as our algorithm finds a match to our target.

DE1_SoC:

The top-level module of this lab is fairly straight forward. We simply assigned SW 0-7 to be our input to both algorithms and SW 9 to be the selector between the counter algorithm (task 1) and the binary search algorithm (task 2). We also had Key 0 as reset and Key 3 as Start to initiate our algorithms. In terms of algorithms instantiation, for counter algorithm, we made one instance of the counter control path and one instance of the counter data path and connected them together. For the binary search algorithm, we made one instance of the binary search control path and one instance of the data path and connected them together. Finally, we made 2 instances of the 7-segment display to show our counter value and the address that gets found by our binary search. We also included 2 LEDRs to demonstrate Done and Found status signals.

```
25      // assign the requested external inputs to the internal control signals
26      assign A = SW[7:0];
27      assign Reset = ~KEY[0];
28      assign Start = ~KEY[3];
29      assign HEX5 = 7'h7F;
30      assign HEX4 = 7'h7F;
31      assign HEX3 = 7'h7F;
32      assign HEX2 = 7'h7F;
33
34      // initalize the counter algorithm FSM and datapath.
35      counter_cntrl cc(.clk(CLOCK_50), .reset(Reset), .s(Start), .A_zero, .got_1, .load_A, .r_shift_A, .incr_result, .clr_result, .done);
36      counter_datapath cdp (.clk(CLOCK_50), .A, .load_A, .r_shift_A, .incr_result, .clr_result, .A_zero, .got_1, .result);
37
38      // initialize the binary search algorithm FSM and datapath.
39      binaryFSM bc (.CLOCK_50, .Reset, .Start, .exhaustedRAM, .Found, .gt, .Compute_M, .Set_LSB, .Set_MSB, .init);
40      binaryDataPath bdp (.CLOCK_50, .A, .Compute_M, .Set_LSB, .Set_MSB, .init, .exhaustedRAM, .gt, .Found, .Loc, .DONE);
41
42      // switch between tasks
43      always_comb begin
44          case(SW[9])
45              0: // counter
46              begin
47                  out1 = result;
48                  out2 = 4'b0000;
49                  d = done;
50                  f = 0;
51              end
52              1: // binary search
53              begin
54                  out1 = Loc[3:0];
55                  out2 = {3'b0, Loc[4]};
56                  d = DONE;
57                  f = Found;
58              end
59              default:
60              begin
61                  out1 = result;
62                  out2 = 4'b0000;
63                  d = done;
64                  f = 0;
65              end
66          endcase
67      end
68
69      // initialize a seven segment display module that outputs either the counter value or address of target value.
70      seg7 seg1(.hex(out1), .leds(HEX0));
71      seg7 seg2(.hex(out2), .leds(HEX1));
72
73      assign LEDR[9] = d; // indicates done state for either counter or binary
74      assign LEDR[0] = f; // off for counter, used for binary to indicate Found
75
76  endmodule // DE1_SoC
```

Figure 9: DE1_SoC top level module

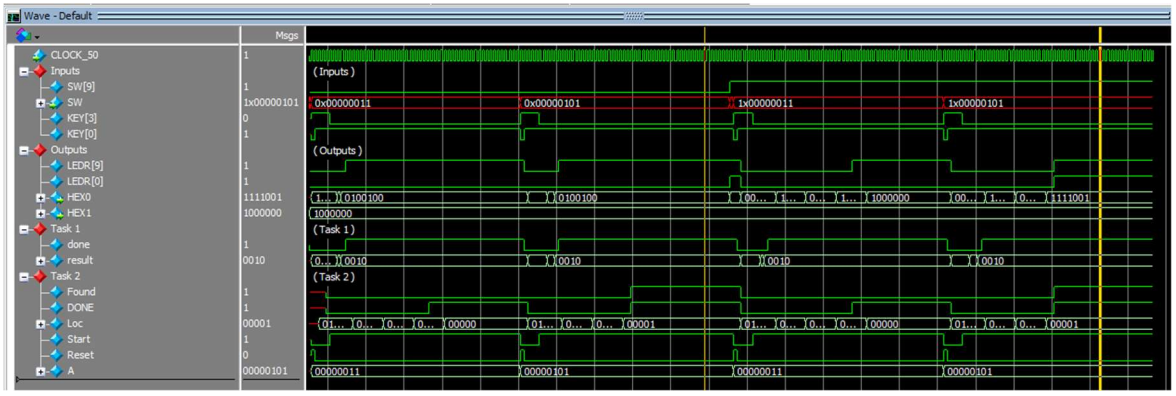
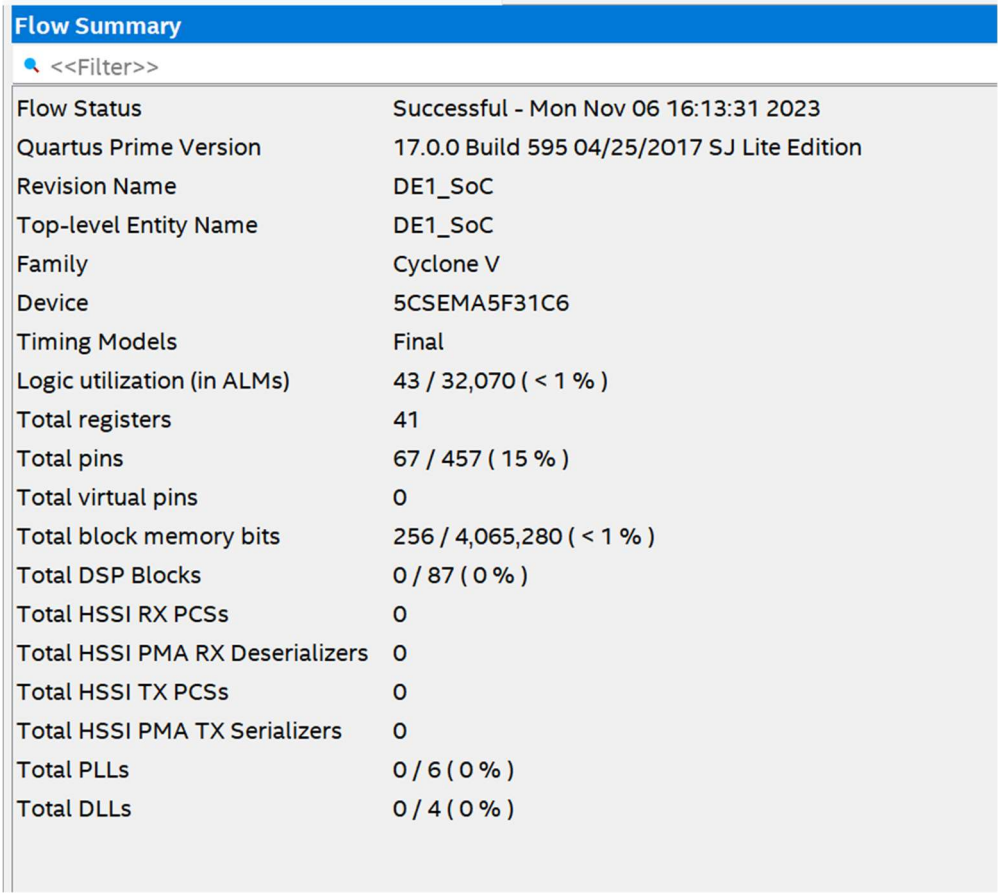


Figure 10: Snippet of DE1_SoC top level model sim showing the combined behaviour for task 1 and 2

For the top-level simulation, we combined the counter and the binary search testbench with the additional control of switching between the two algorithms at halfway point through the simulation. In the above ModelSim snippet, 2 yellow lines can be seen indicating the 2 points where the finished algorithm displays the final value. When SW[9] is set to 0, the counter algorithm is being selected and the result signal shows the value 2 since 2 input bits are set to 1. When SW[9] is set to 1, the binary search algorithm is being selected and the Loc signal shows the correct index when the value A = 5 is located as previously shown in the mif file. The LEDRs and HEX displays also display the appropriate behavior for both algorithms as shown in the above snippet. The Start, Done, Reset signals are all correct and behave as expected.

Flow Summary:



Flow Summary

<<Filter>>

Flow Status	Successful - Mon Nov 06 16:13:31 2023
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	43 / 32,070 (< 1 %)
Total registers	41
Total pins	67 / 457 (15 %)
Total virtual pins	0
Total block memory bits	256 / 4,065,280 (< 1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 11: Snippet of the Flow Summary for the top-level module DE1_SoC

Experience Report:

Implementing the 2nd task; binary search algorithm was relatively difficult. We initially encountered issues with identifying which signals are control vs status signals. We also encountered problems with getting the DE1_SoC top level module to work on LabsLand. We also spent a lot of time determining if the ASMD components are combinational or sequential. We had to review the class lecture slides multiple times throughout working on the lab. For teamwork, we used GitHub for easy collaboration of shared files which made code developing and debugging more efficient. It was not very helpful to reference the provided block diagrams since they were incomplete and confusing. For the next lab we would like to get a head start and aim to work together live more frequently to reduce debugging time. Additionally, using GitHub's full functionality for version control may help make our code development process even smoother.

The estimated total time working on this lab was 24 hours broken as follows:

- Reading: 2 hours

- Planning: 2 hours
- Design: 5 hours
- Coding: 6 hours
- Testing: 5 hours
- Debugging: 4 hours.