**Illya Kuzmych (2127069), Ruvim Piholyuk (2128297)**
EE/CSE 371 22sp
May 4, 2022
Lab 4 Report

## Design Procedure:

In this lab we were tasked with implementing a Bit-Counting and a binary search algorithm. We utilized the DE1_SoC board to test our modules, from which we used switch 9 to toggle between the bit counting algorithm (`SW[9] = 1`) and the binary search algorithm (`SW[9] = 0`). In order to implement both task 1 and 2 we used an ASMD to plan the design of the algorithm according to the specification. In order to implement task 1 we used a value (`SW[7:0]`) that is loaded into a shift register to check the bottom bit and to increment the total if it was true. Then we shifted the value that was loaded in and repeated the process until the load register had become zero. Then a done signal will be true (`LEDR[9]`) and the total bits are displayed(`HEX0`). Task 2 on the other hand required that we implemented an asynchronous ROM to read from the sorted .mif file that we created. Then we input a value (`SW[7:0]`) that was checked for in the middle of the ROM file and depending on if the value in the ROM was greater or smaller the algorithm would keep cutting down half off the remaining options until the value is found or it does not exist. In the case that the searching is finished, done is displayed (`LEDR[9]`) if the value was found the address of the value is displayed (`HEX1, HEX0`) in hexadecimal format and a found signal is displayed as well (`LEDR[0]`).

### Task #1 (Bit-Counting Algorithm):

In Task #1 we were tasked with implementing a bit-counting algorithm on hardware. Our implementation consisted of a hardware-user interface where the inputs are all directly from the user. The 8-bit input A was controlled by switches 7 through 0 on the DE1_SoC board. The synchronous reset was hooked up to KEY[0], and the start signal was hooked up to KEY[3], which were also put through two DFFs to prevent metastability. Our controller directed from which states to move and which states to go to. We had three total states in our ASMD: S1, S2, S3. In S1 we reset the result to 0 and checked the start signal to see if we needed to load the input value to our shift register. If the start was false, we loaded in a new value, and if it was true, we moved on to S2. In S2, we checked the bottom bit of the right-shift register, then incremented the result if it was high, or shift the register and check the bits until it is equal to 0, then the controller makes S3 the present state, and activates the done signal. Figure 1 shows our ASMD for the bit counting algorithm.

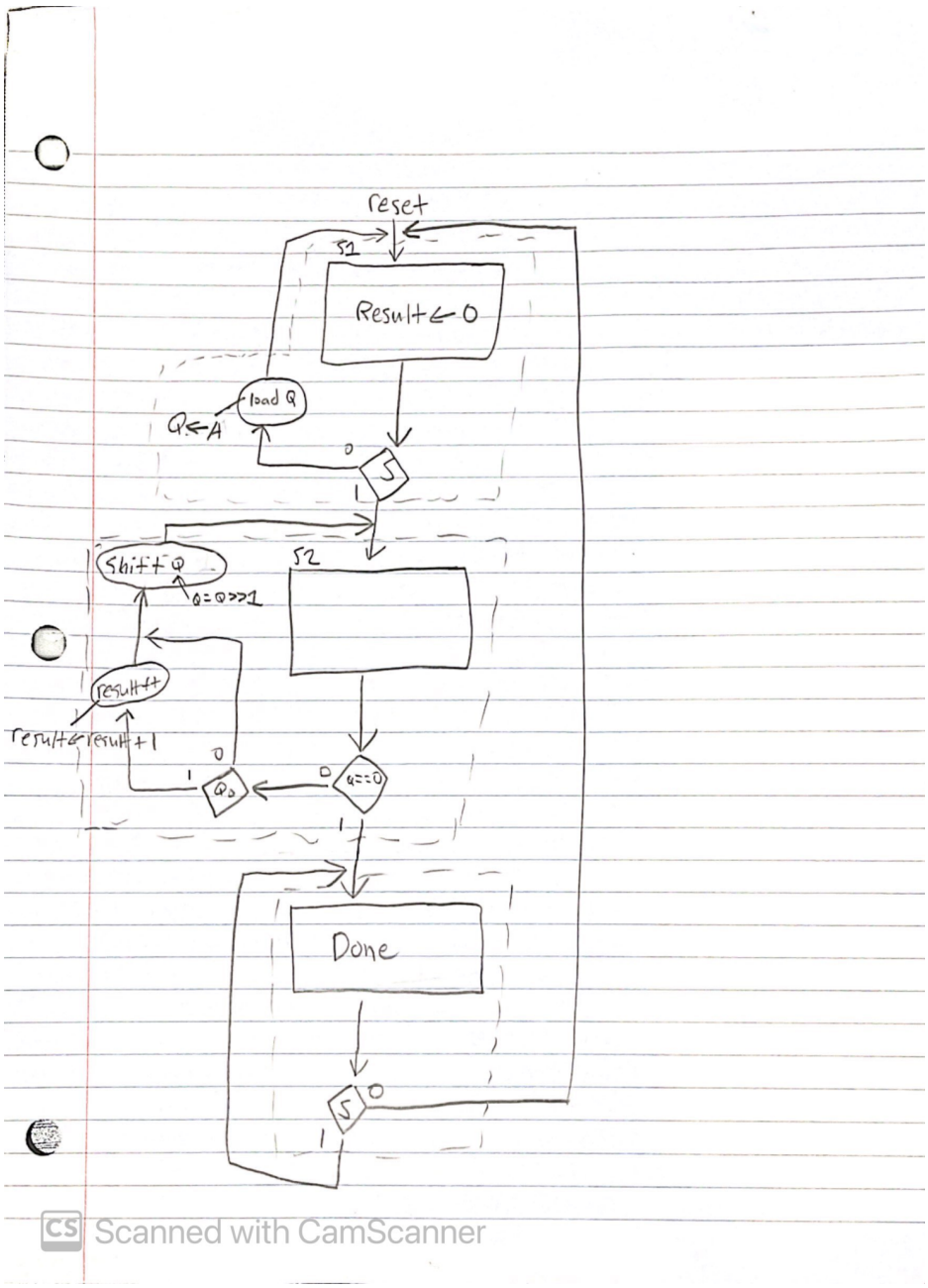



Figure 1. Bit counting algorithm ASMD

**Task #2 (Binary Search Algorithm):**

In Task #2, our goal was to implement another algorithm on hardware, this time binary search. A binary search algorithm searches for a specified value in a sorted array by splitting it into halves. First, it compares the specified value to the middle value of the array or RAM, then checks the upper half or lower half of the array, and then continues recursive calls until it either finds the value and stores the address, or does not find the value and throws up a flag saying the algorithm has finished and the value was not found in the array/RAM. To implement this in hardware, we

first designed an ASMD that portrayed the expected flow and layout of our algorithm. Figure 2 displays the ASMD upon which we based our algorithm on.

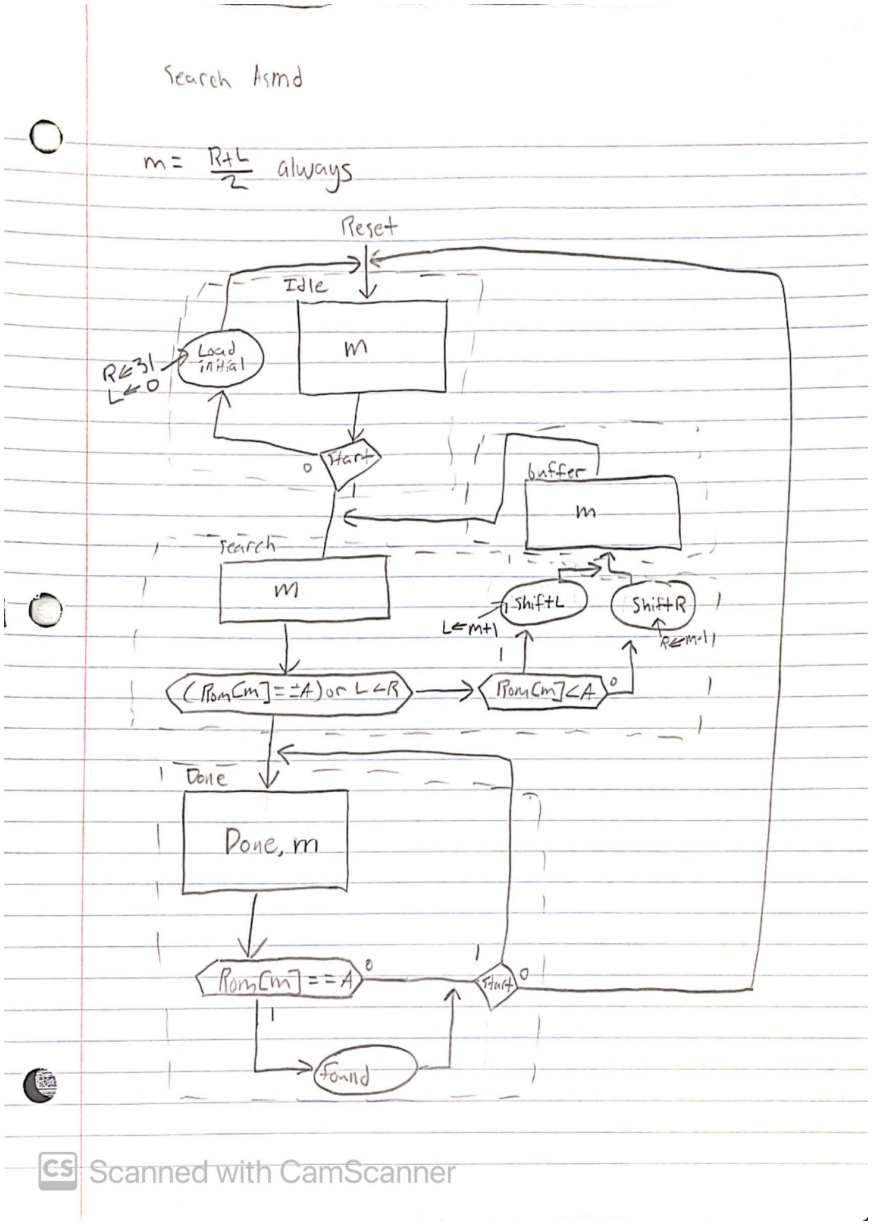



Figure 2. ASMD of the binary search algorithm

Once again, this lab was designed to be interactive, with a specified input by the user utilizing switches 7 through 9 on the DE1_SoC FPGA. To make our system work, we designed it to have 4 states: `idle`, `search`, `searchBuffer`, and a final state, `done`. Initially upon reset or default, our system would start in the `idle` state, from which it only moved from if the `Start` signal was high. In the reset state, `L` was set to 0, and `R` was set to be ROM depth minus 1. `Done` and `Found` are set to false by default. When `Start` is enabled to high, the algorithm goes to the `search` state which at each clock edge performs the necessary RTL operations,

which include checking the value of the current RAM index and incrementing `L` and `R` as necessary depending on what that value is. The search state always go to the `searchBuffer` state, as to include a one clock cycle delay to prevent comparing the wrong value out of the ROM, and reaching the `done` state properly. If `L` is at any point greater than `R`, or `L` is 31 or R is 0, then the circuit goes to the done state, `Found` stays low, `Done` goes high, and nothing is output on the `HEX` displays. One of the significant design choices we made was that our circuit would only perform the algorithm procedures while the `Start` signal is high. If `Start` ever goes low, then the machine goes back to the `idle` state. If the value is found, then Loc stores and outputs that current value to the `HEX` displays, and the circuit reaches the done state, flagging the `Done` and `Found` signals as high. In order to toggle the activity of the FPGA to only focus on either Task #1 or Task #2 depending on the user input, we created a combinational block to vary the outputs based on which state our circuit was in. Figure 3 shows the code we used to toggle between Task #1 and Task #2 in our top-level module.

```
/* Toggle between Task1 and Task2
 * If SW[9] is high, perform task 1
 * Otherwise, do task 2
 */
always_comb begin
   if (SW[9]) begin
      HEX0 = task1leds;
      HEX1 = 7'b1111111; // do not need HEX1 for task1
      LEDR[9] = task1done;
      LEDR[0] = 0;
   end // if
   else if (Found) begin // if the address is found, display it
      HEX0 = locOut0;
      HEX1 = locOut1;
      LEDR[9] = task2done;
      LEDR[0] = Found;
   end // else if
   else begin            // if the address is not found, do nothing
      HEX0 = 7'b1111111;
      HEX1 = 7'b1111111;
      LEDR[9] = task2done;
      LEDR[0] = Found;
   end // else
end // always_comb
```

Figure 3. Combinational function to toggle between task 1 and 2 based on user input from SW[9]

**System block diagram:**

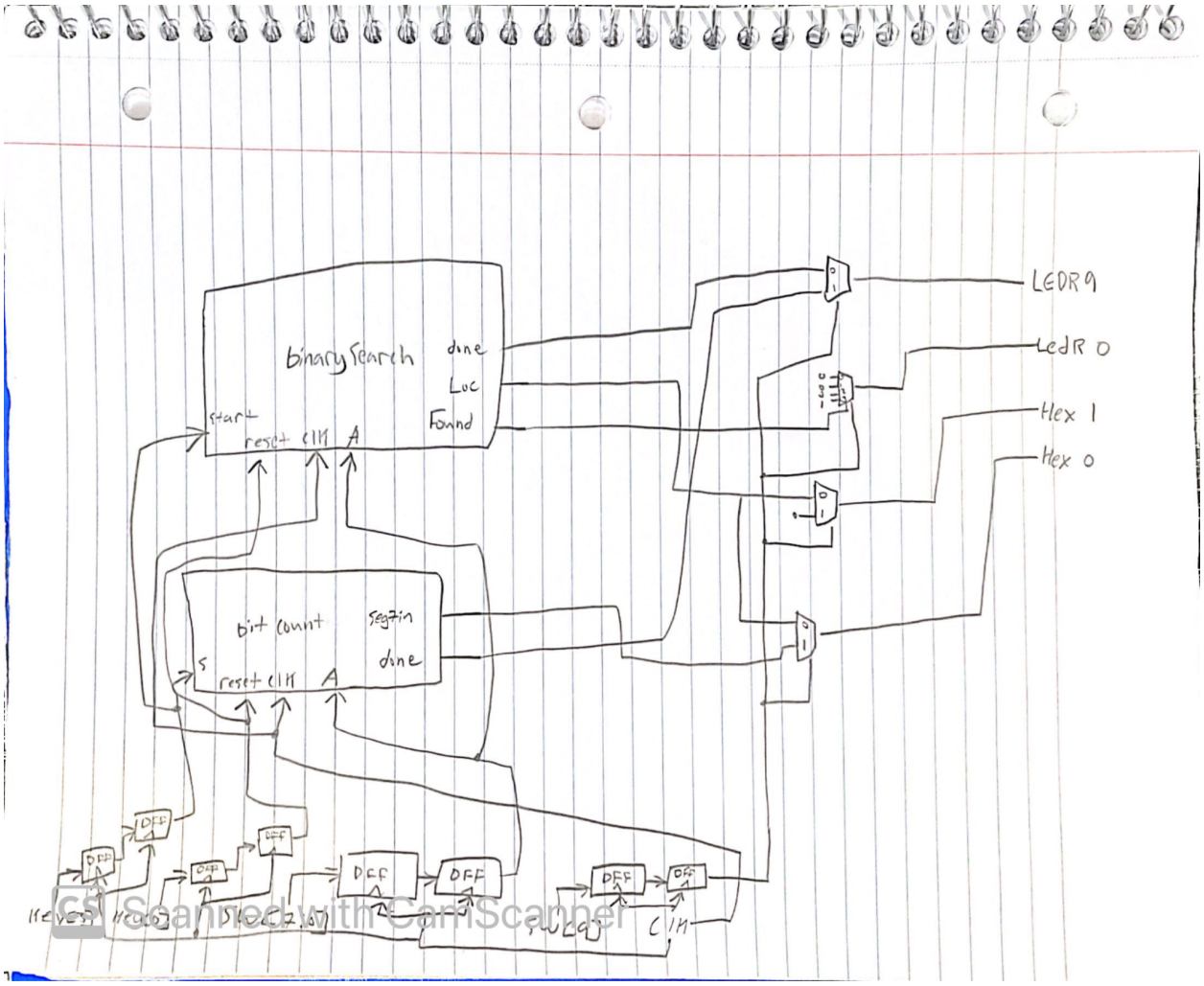



Figure 4. System block diagram for Lab 4 implementing bit counting algorithm and binary search algorithm

## Results:

### Overview:

In this lab, we experimented with and developed our skill in utilizing an ASMD to design and implement specific algorithms. We enhanced our skills with controlling a data path according to an ASMD. We began with using the given ASMD from the lab specifications and created a state machine that controlled the data path for the bit counting algorithm. We also enhanced our skills with writing an ASMD chart based on given specifications in task 2. We had to keep in mind timing when writing the ASMD to account for the time delay of the output of the Rom. Thus, we used a buffer state in order to get the expected values for the comparison in the data path. Also, we were tasked with implementing a single high level design to toggle between the bit counter algorithm and the binary search algorithm based on the state of switch 9 on the DE1_SoC board.

To test our designs, we used high-level testbench modules to ensure the design followed the original ASMDs. The figures and corresponding descriptions below demonstrate the functionality of our project.

**DE1_SoC:**

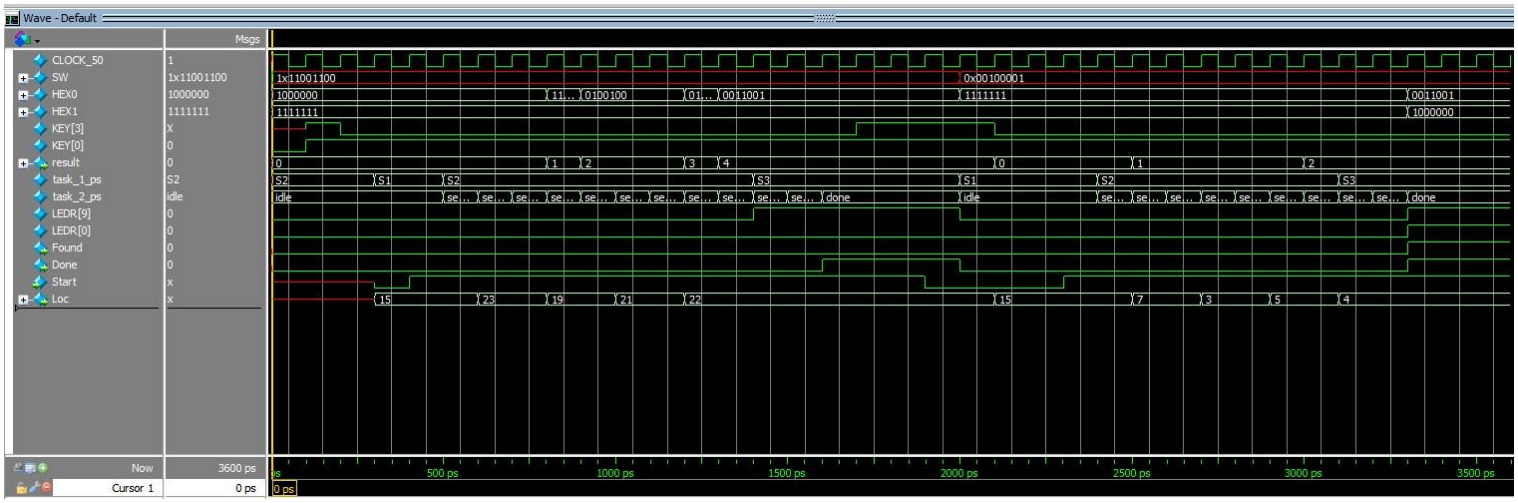



Figure 5. Top-level simulation showing full functionality of the system

In Figure 5 we show our completely functional top-level diagram, demonstrating that the toggle between the activity of each state works. While the toggle itself did not directly impact the internal behavior of each algorithm, it directly affected what was being output on the LEDRs and the `HEX` displays on the board. In the simulation, we first show that Task 1 works as expected, by loading in a value that we want to count bits of, then initializing the start signal to run through and show it works. The `HEX` display works as expected. We then switch the system to task 2 by toggling `SW[9]` to low, and doing a full system reset by setting `KEY[0]` high (active low). Doing so allowed us to reset our switches 7 through 0 to test different values that could be present in the ROM. The values that were in the ROM were found accurately, while values that did not exist were ignored, the `Done` signal went to high, and the `Found` signal stayed low, as demonstrated by the outputs `LEDR[9]` and `LEDR[0]`, respectively.

**binarySearch:**

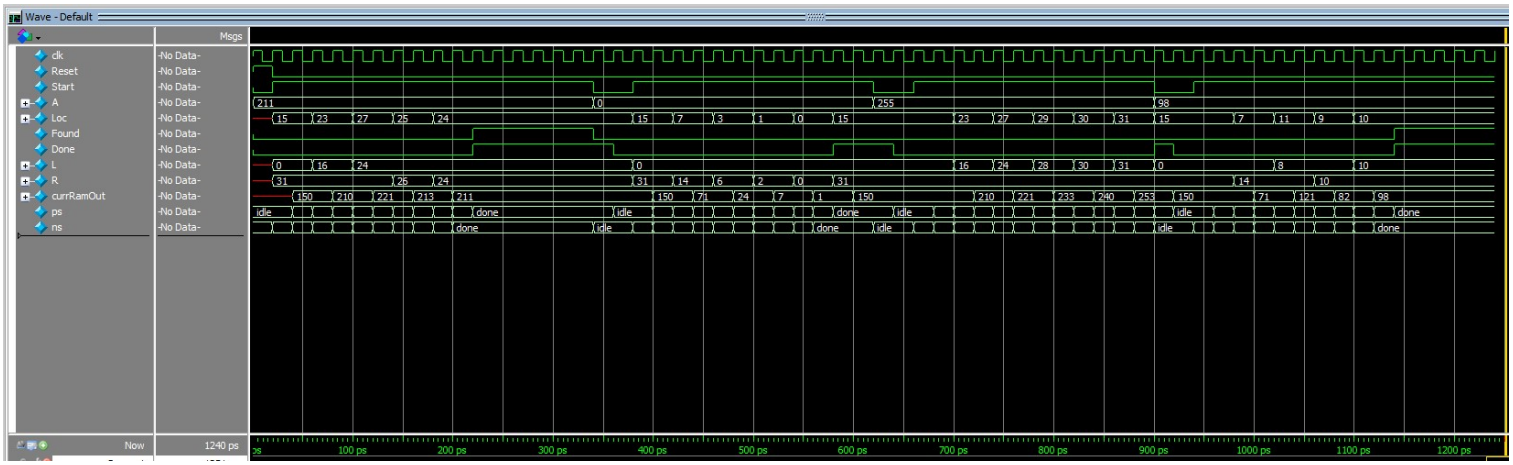



Figure 6. ModelSim simulation of binary search algorithm

In Figure 6 we show a high level simulation of the Binary Search algorithm working according to the given specifications. We begin by toggling reset and then setting the input to 211 which exists in the .mif file. We confirmed the module outputs `Done`, `Found`, and `Loc` once the calculation is complete. Then we send the state machine to `idle` and do the process again with zero which is smaller than any value held in the .mif file. We confirmed that the output is a `Done` signal but `Found` is not high, as we expected. Then we tried again with 255 since it is larger than anything in the .mif file and found the same results as when 0 was input. Finally, we tried again with 98 and confirmed that done and found are both true once binary search is in the done state as we would expect. This gave us confidence to implement the algorithm in our top-level module.

**bitCount:**

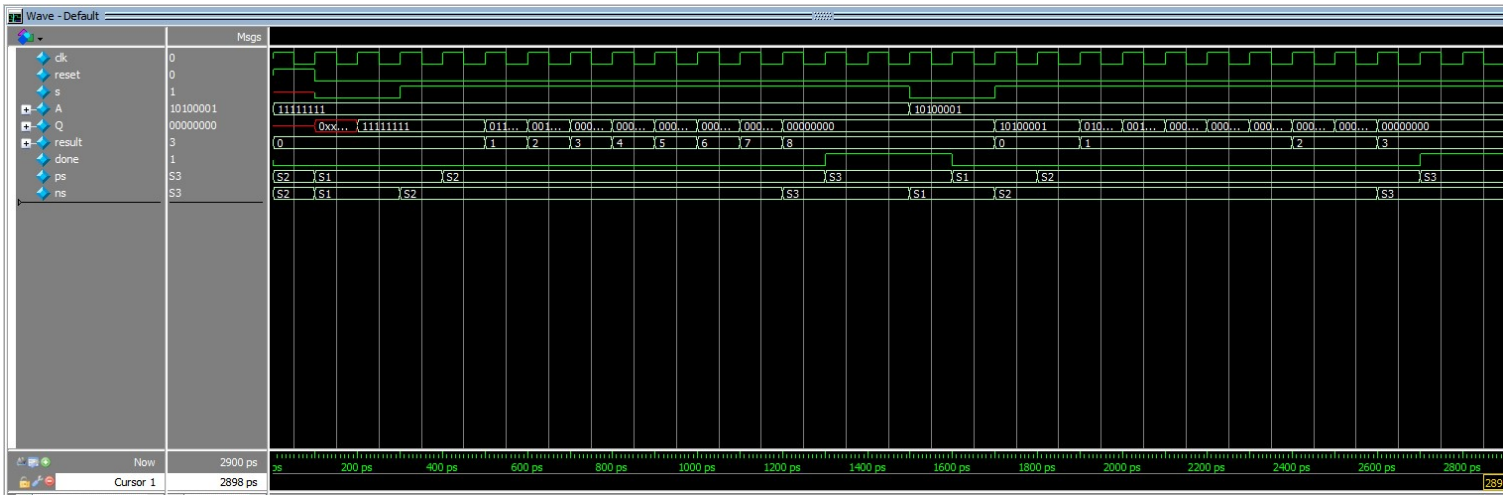
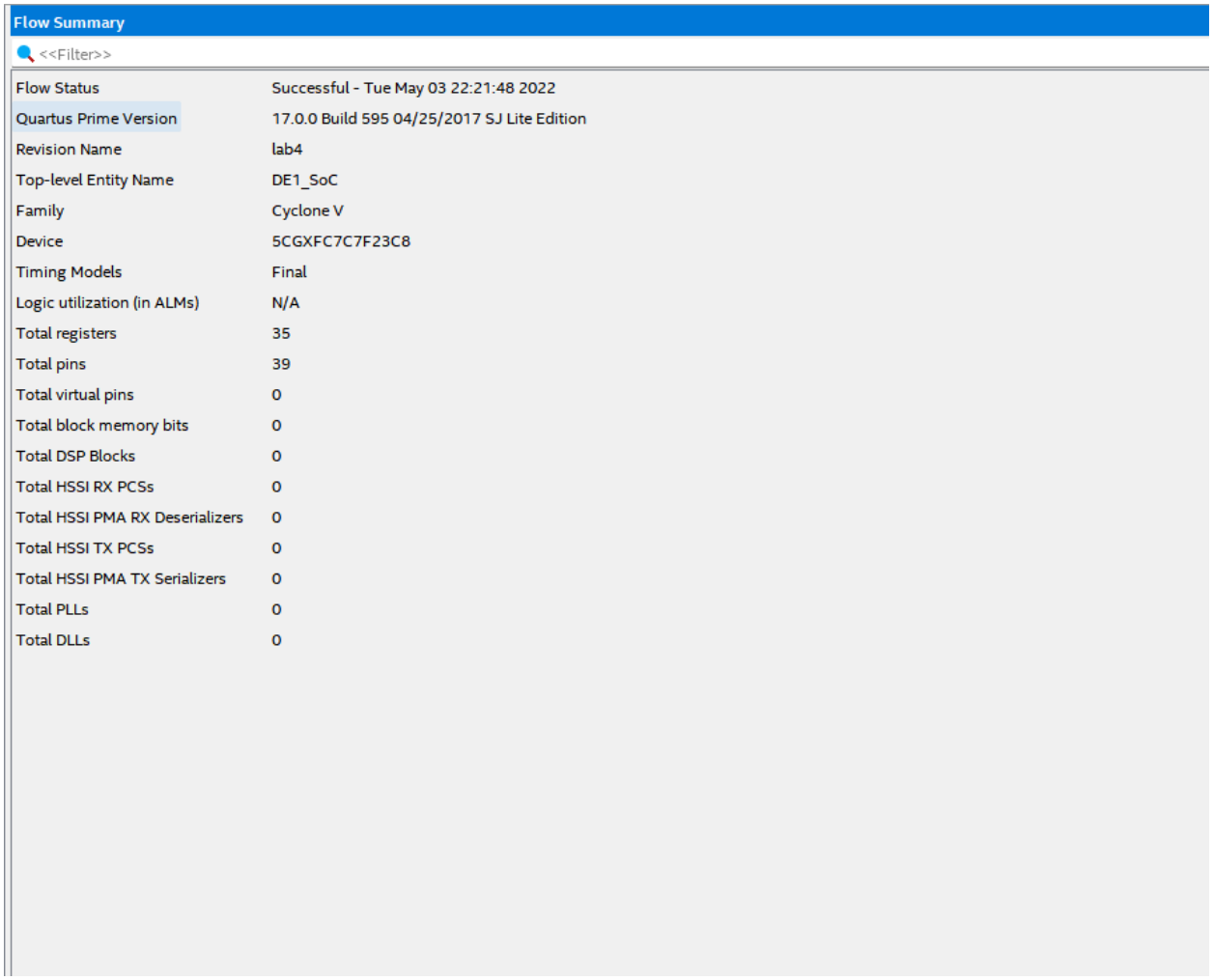



Figure 7. ModelSim simulation of bitCount algorithm

In Figure 7, we show the simulated results of our testbench written to test the functionality of our bit-counting algorithm. The testbench is fairly straightforward and simple, as the algorithm itself was also pretty simple. We initialized a value for `A`, which was loaded into a shift register when the s signal was set to 0. Once the `A` value was loaded into the register, we set s to high and let the algorithm run for several clock cycles. Our result came out as expected for both our cases, one checking a `A` value that was all 1s, and another `A` value with 1s in random positions. We then implemented the functional algorithm in our top-level design and hooked everything up appropriately.

**Flow summary:**



Flow Summary

<<Filter>>

| | |
|---|---|
| Flow Status | Successful - Tue May 03 22:21:48 2022 |
| Quartus Prime Version | 17.0.0 Build 595 04/25/2017 SJ Lite Edition |
| Revision Name | lab4 |
| Top-level Entity Name | DE1_SoC |
| Family | Cyclone V |
| Device | 5CGXFC7C7F23C8 |
| Timing Models | Final |
| Logic utilization (in ALMs) | N/A |
| Total registers | 35 |
| Total pins | 39 |
| Total virtual pins | 0 |
| Total block memory bits | 0 |
| Total DSP Blocks | 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 |
| Total DLLs | 0 |

Figure 8. Flow summary of the overall system in Lab 3

## Experience Report:

Our group found the lab to be moderately easy in difficulty. Our biggest hurdle was tackling a timing issue in the binary search algorithm when the output of the ROM was a clock cycle too

slow to get an accurate comparison. To solve this problem we changed our ROM to be asynchronous and added a buffer state so that we ensure we get good values for the comparison that drives the data path. Also, we had difficulty building the ASMD for the binary search algorithm. When following the data path we struggled to find exactly when to stop searching but after referencing the pseudo code from the lecture we were able to build the completed binary search algorithm according to the specifications. One design choice we made was to make the ROM asynchronous in order to help with timing issues as stated before. Also, to always output the address being checked in the binary search algorithm and then only to display it on `HEX1` and `HEX0` if the `Found` signal is true. This lab took us approximately 11 hours, broken down as follows:

- Reading – 30 minutes
- Planning –1 hour
- Design – 1 hour
- Coding – 2.5 hours
- Testing – 2 hours
- Debugging – 4 hours