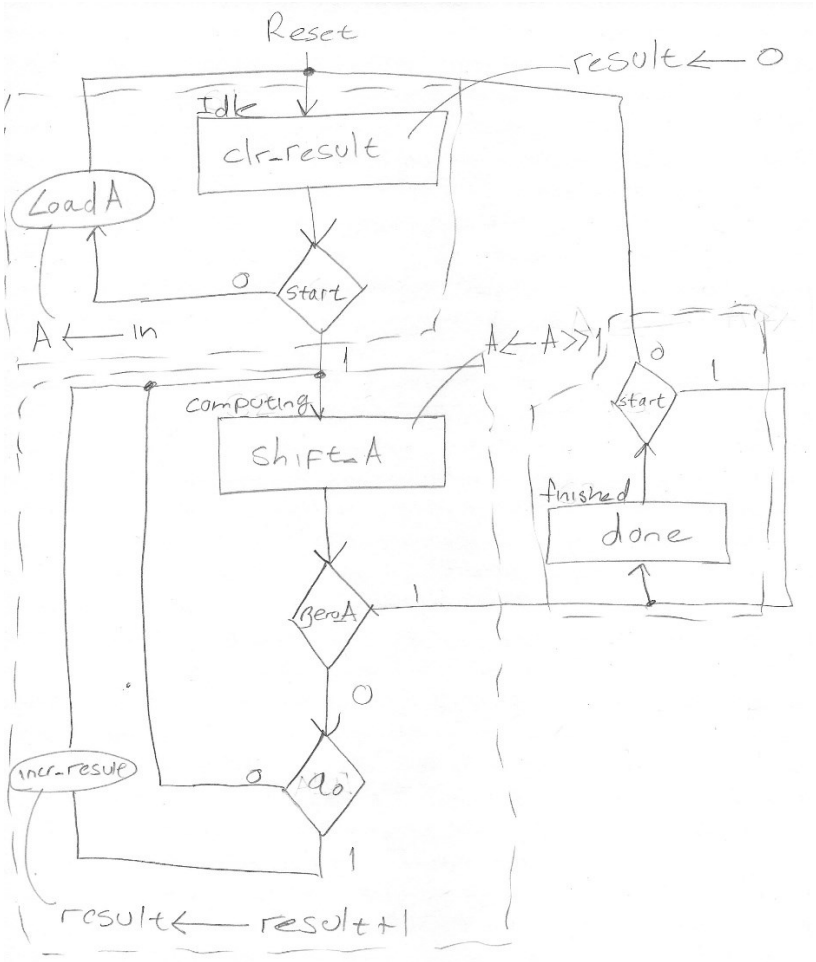
Student Name: Rafael Leyva
Student ID Number: 1975274

EE/CSE 371: laboratory #4, Implementing Algorithms in Hardware

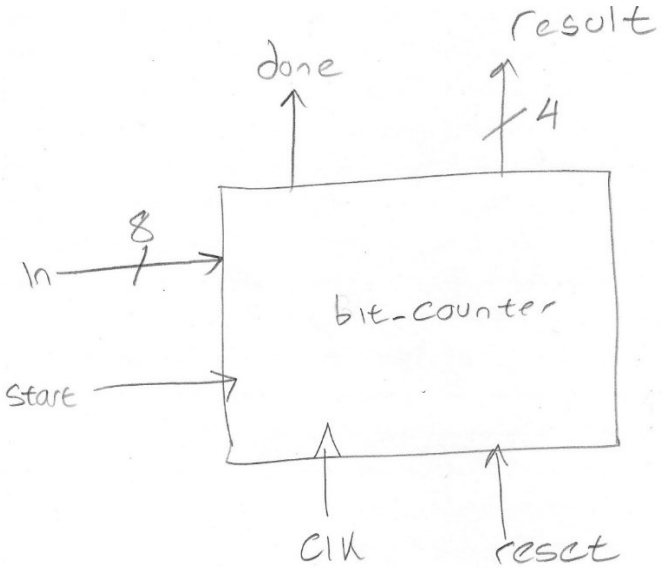
Design Procedure

Task #1

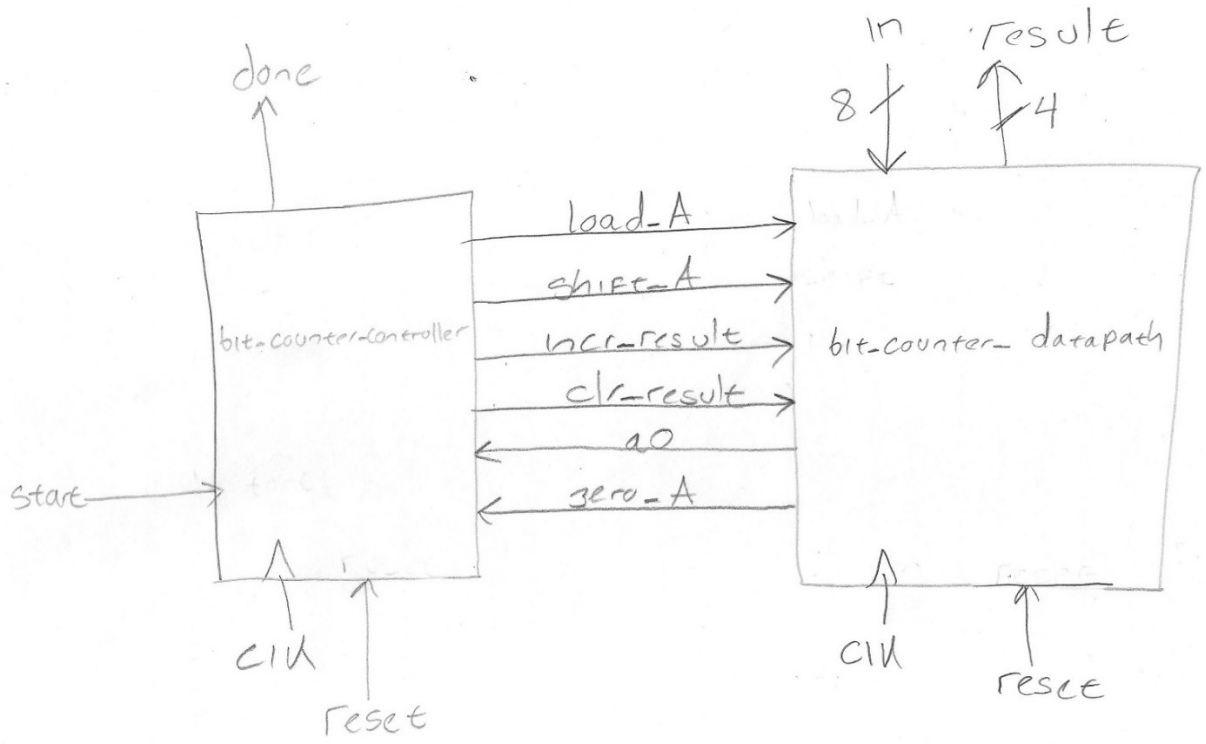
First, the ASMD diagram from the laboratory document is repeated here using the signal names that I used.



Therefore the goal is to provide a module that looks as follows to the external world:



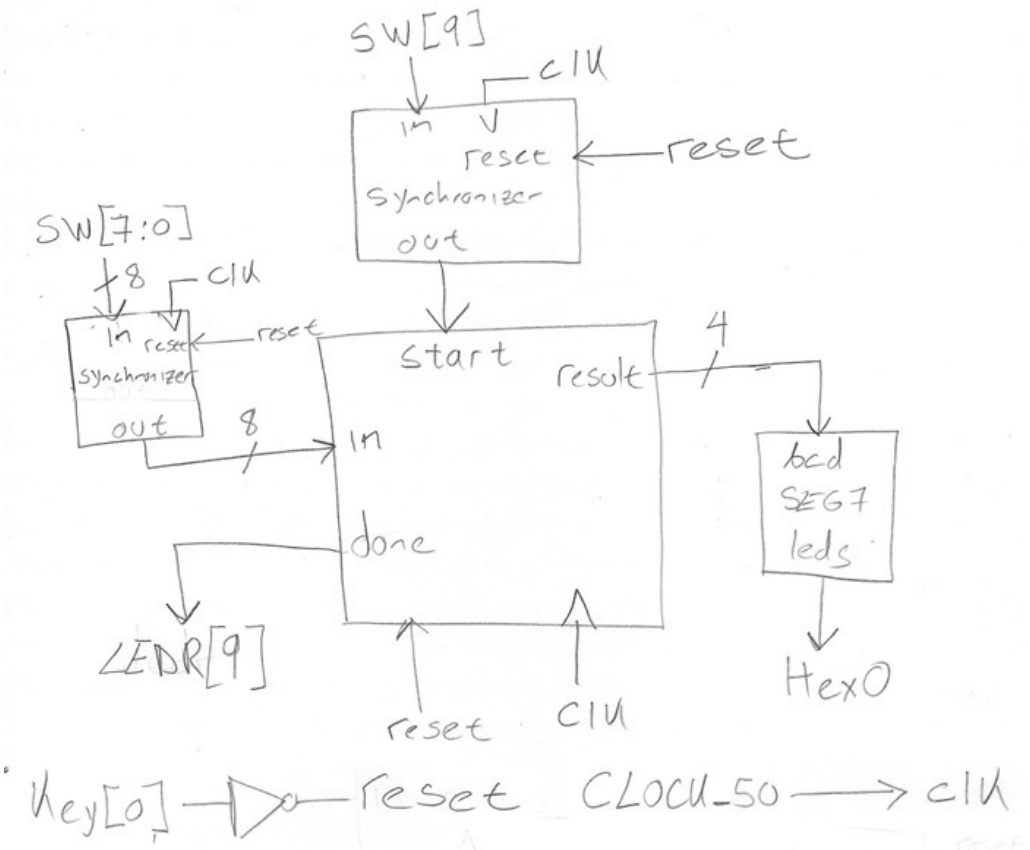
Internally, there will be a Controller (left) and a Datapath (right):



The Systemverilog code for the controller has been attached as *bit_counter_controller.sv* and the Datapath as *bit_counter_datapath.sv*. Likewise, the whole module has been attached as *bit_counter.sv*.

The result is 4 bits, since given an 8-bits input there could be at most 8 bits SET. Therefore we need to represent numbers from 0 to 8, and the smallest number of bits that can represent 9 pattern is 4 bits. This also has a nice side-effect since when designing the *DE1_SoC* module we can directly hook this result to *seg7* from *seg7.sv* without any further conversions.

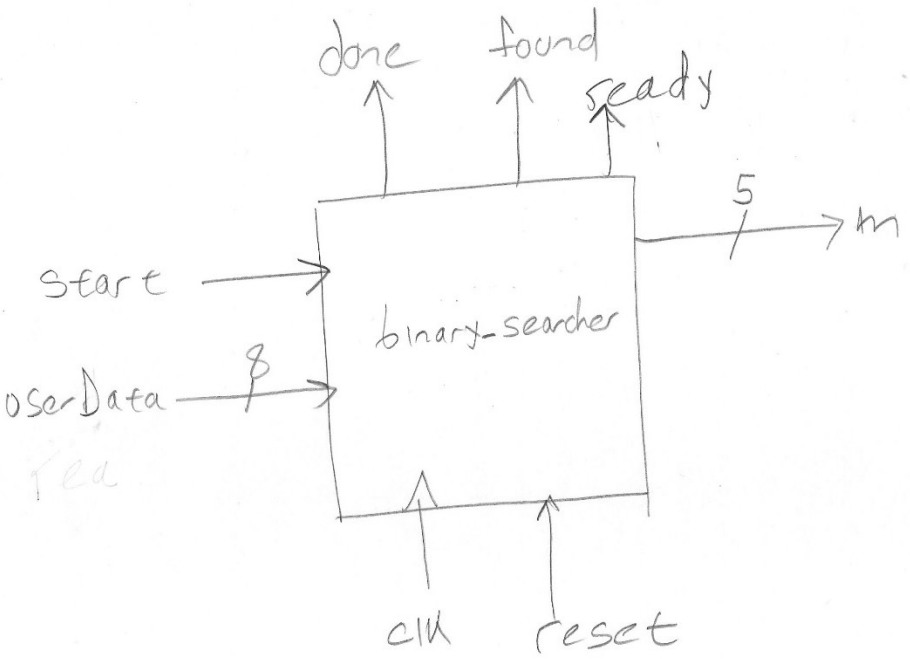
Next, the following connections are made for *DE1_SoC*:



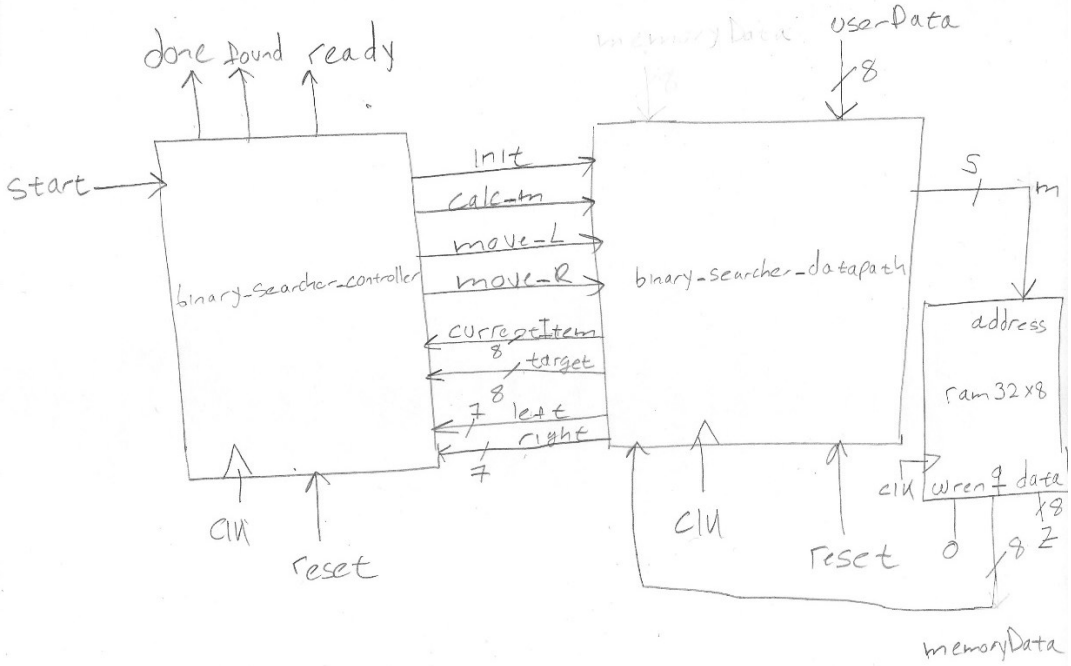
The user can then use SW[7:0] to provide inputs to the machine. Feedback is received on LEDR[9] to signal that the computation has finished and the output is available. The output can be seen in HEX0 which is the number of bits SET in SW[7:0]. Pressing KEY[0] resets the system. The code has been attached as *DE1_SoC_Task1.sv*

Task #2

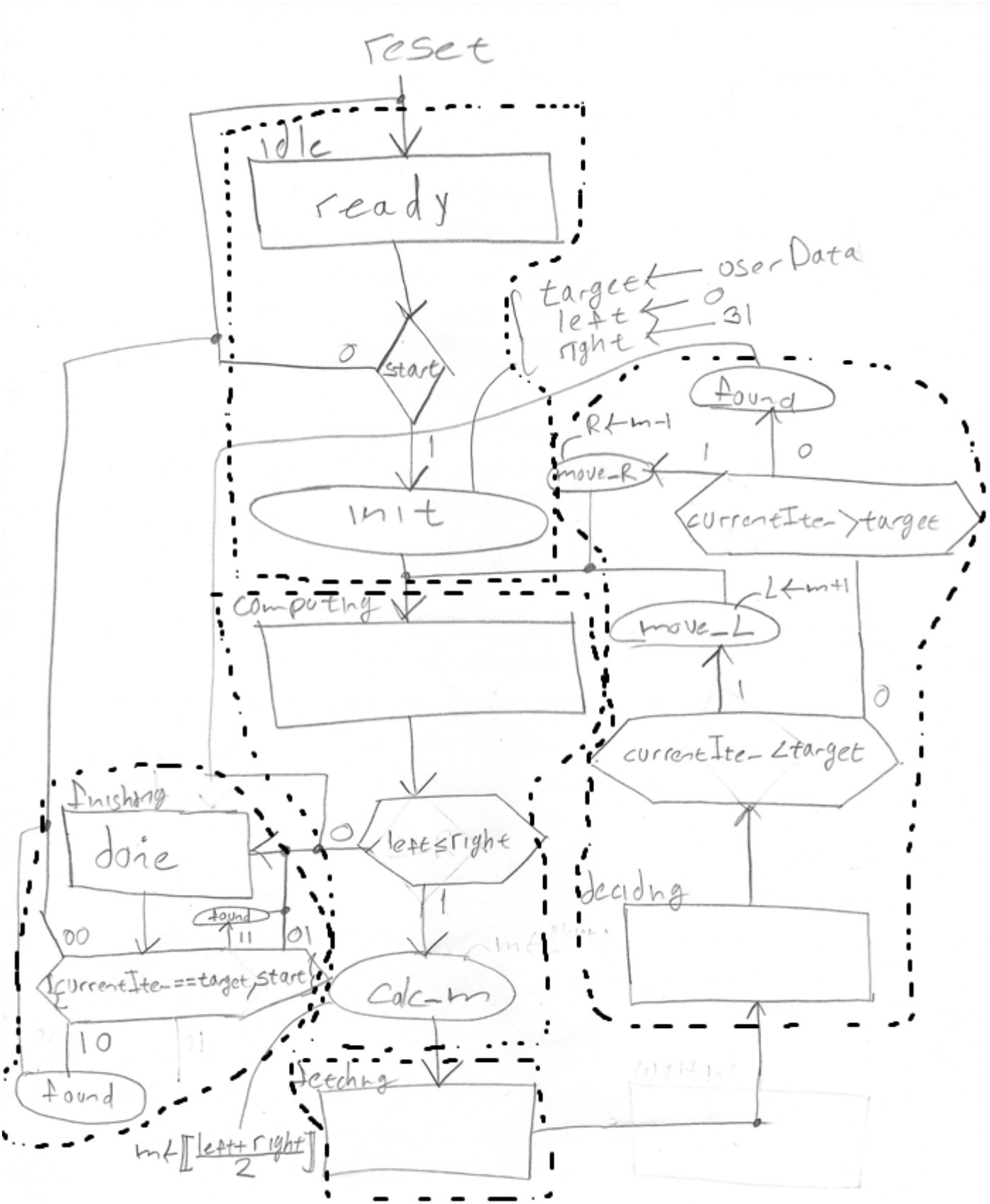
Binary searcher will look like the following to the external world:



The user can input 8 bits of data and a start signal to start the search. The module will assert *done* when the search is complete, and *found* will be asserted if and only if the given item at *userData* was found, and its address in the internal 32x8 RAM will be available as *m.* The following block diagram of a controller, a datapath and a memory module is produced:



Using the signals from the block diagram, the following ASMD chart is made:

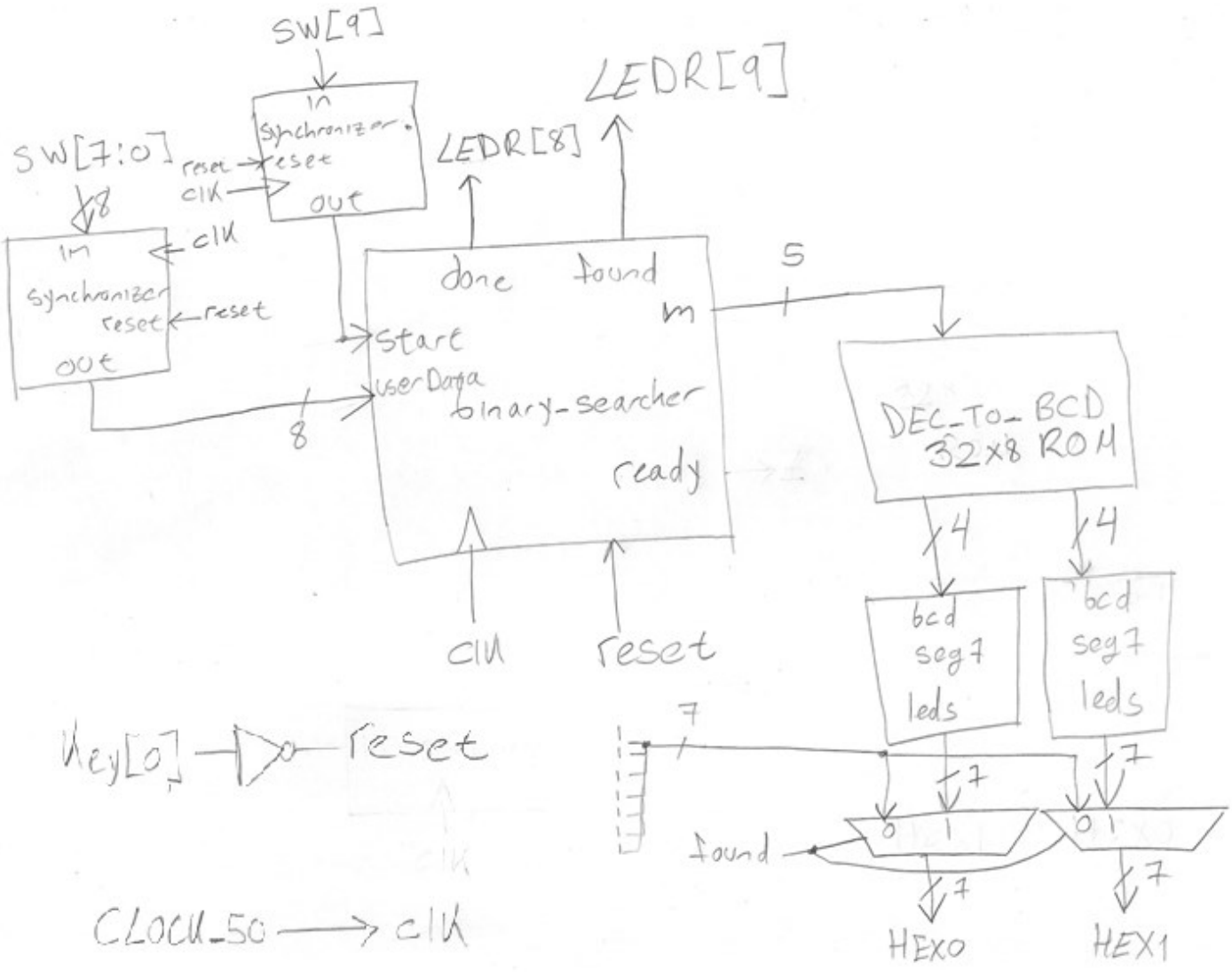


- There is an extra cycle to allow for the memory to load (state *fetching* in the ASMD).
- Found arrives one clock earlier than done due to the conditional output in *deciding*, which slightly decreases latency for a user application using this module.

- The state *finishing* looks a bit complicated at first glance. However, it is simply checking if the *currentItem* matches the *target* which dictates if *found* is asserted or not. In addition, this state also polls *start* until it is de-asserted to go back to *idle*.
- In the SystemVerilog implementation, *left* and *right* are declared as signed and are 7 bits long, which is one bit longer than necessary. This is done explicitly to simplify the logic in edge cases. For example, if *left* and *right* match at index 31 and the algorithm tries to increase *left*, we'd like to avoid overflow and infinite loops. If *left* and *right* match at index 0 and the algorithm tries to decrease *right*, we'd like to avoid underflow since it affects the next decision in the algorithm (avoids infinite loops).

The binary search modules is attached as *binary_searcher_controller.sv*, *binary_searcher_datapath.sv* and *binary_searcher.sv* which contain the controller, Datapath, and the top-level module, respectively. Next, the top-level DE1_SoC module for this task makes the following connections (attached as *DE1_SoC_Task2.sv*):

- SW[9] is synchronized and used as a *start* signal.
- SW[7:0] is synchronized and used as the data to be searched for in memory.
- LEDR[8] shows the value of *done* for feedback to the user.
- LEDR[9] shows the value of *found.*
- HEX1-HEX0 display the value of the address where the user data is found in memory, if found.
- KEY[0] is used for reset.
- A small look-up table is used to transform an address into a format that the *seg7* module can use before going to the hexadecimal displays. An alternative was doing modulo and division, but this implementation requires less resources.



Results

Overview

The modules developed in task 1 fit the specifications specified in the laboratory document. The 8-bit input comes from the user from switches SW[7:0], and the reset signal is available by pressing KEY[0]. To start a search, the user must assert the *start* signal through SW[9]. There will be feedback on LEDR[9] to signal the algorithm has finished, and the number of bits SET in the input from SW[7:0] is correctly shown in HEX0. The result is satisfactory.

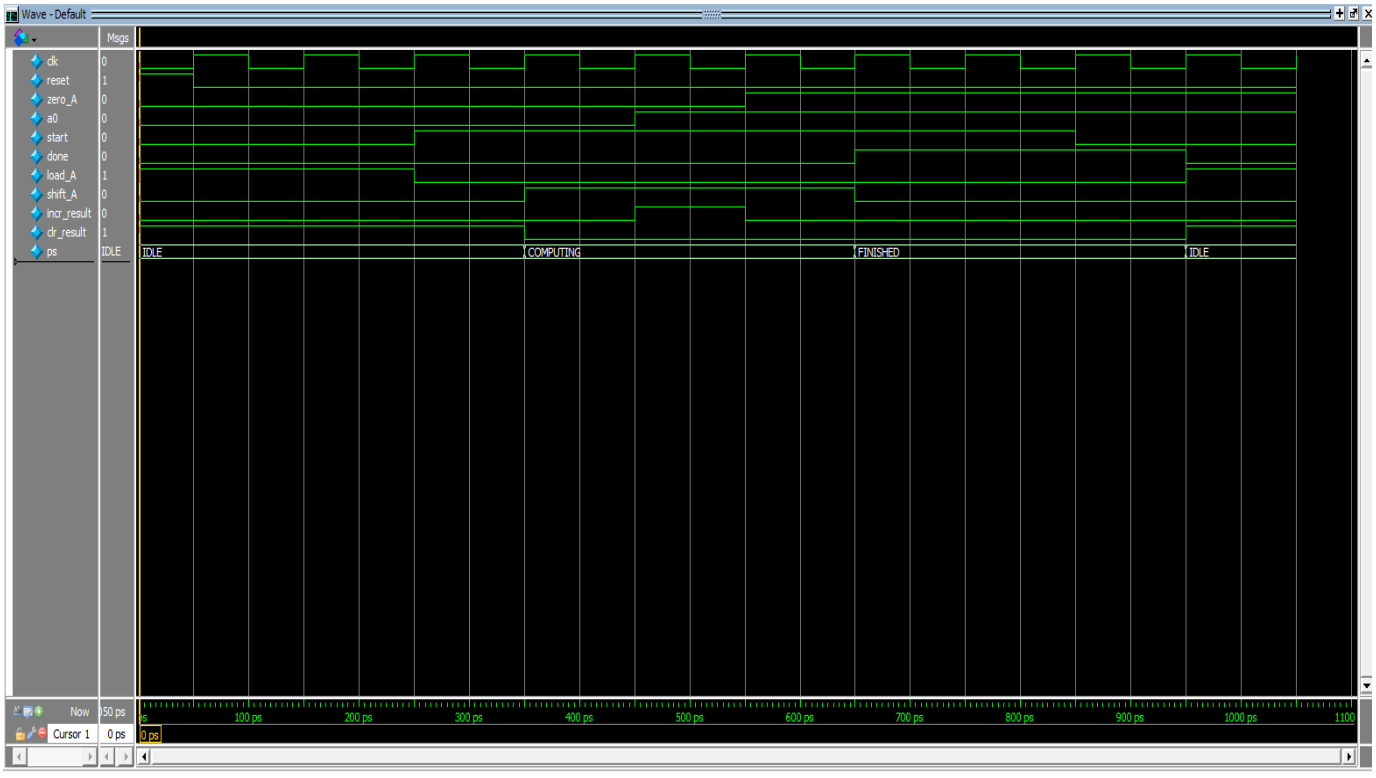
The modules developed in task 2 fit the specifications outlined in the laboratory document. The user can use SW[7:0] to provide a value that they would like to search in the internal RAM, whose values are initialized from *my_array.mif*. The reset signal is available by pressing KEY[0], the output address of the item (if found) will be available at HEX1-HEX0. Feedback will also show in LEDR9 for the *found* signal. In addition, I also added LEDR8 as feedback for the signal *done* that signifies the search has been completed. The result of this task was satisfactory.

Task #1

Controller

The following screenshot is from the *bit_counter_controller* testbench and tests every state and transition:

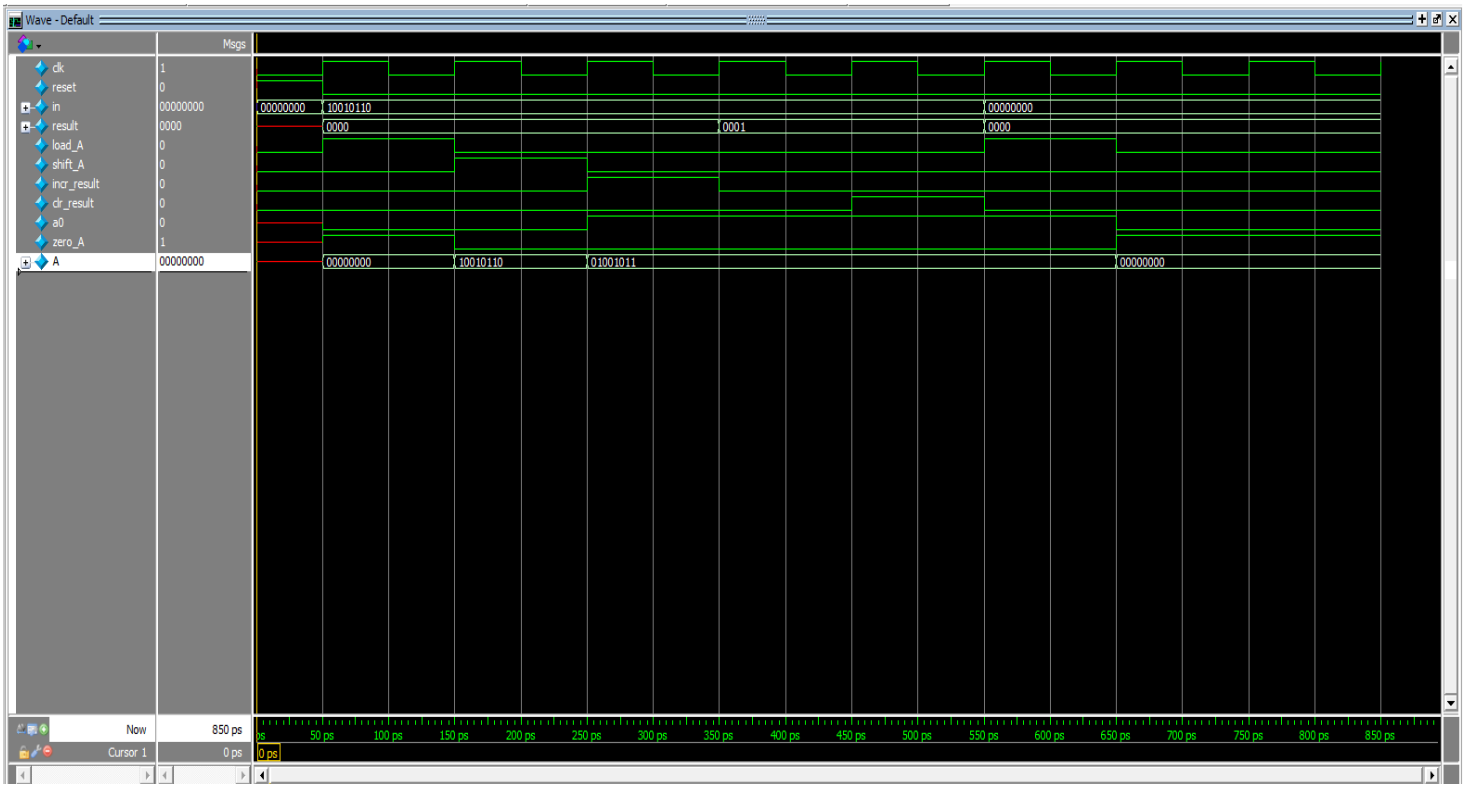
1. The load loop: From 0 to 250ps the system is in IDLE with start being FALSE, the *load_A* is TRUE for this period, also *clr_result* is TRUE too.
2. System starts: *Start* is asserted at 250ps and *load_A* becomes FALSE. The system transitions into COMPUTING. The machine maintains *clr_result* TRUE.
3. Computing without increase: 350 to 450 ps the system is in COMPUTING with *zero_A* and *a0* both FALSE thus *shift_A* is true while *incr_result* is not.
4. Computing with increase: *a0* is asserted after 450ps and so *incr_result* gets asserted.
5. Finishes computing: At 550ps *zero_A* gets asserted and thus *incr_result* gets de-asserted and the system will transition to FINISHED.
6. Start loop: Since *start* is asserted TRUE, the system goes into the *start* loop with *done* also TRUE.
7. Goes back to idle: At 850ps *start* gets de-asserted and the system can now move to IDLE, which it does at 950ps.



Datapath

The testbench makes sure the datapath behaves as designed given each control signal. The simulation can be read from left to right:

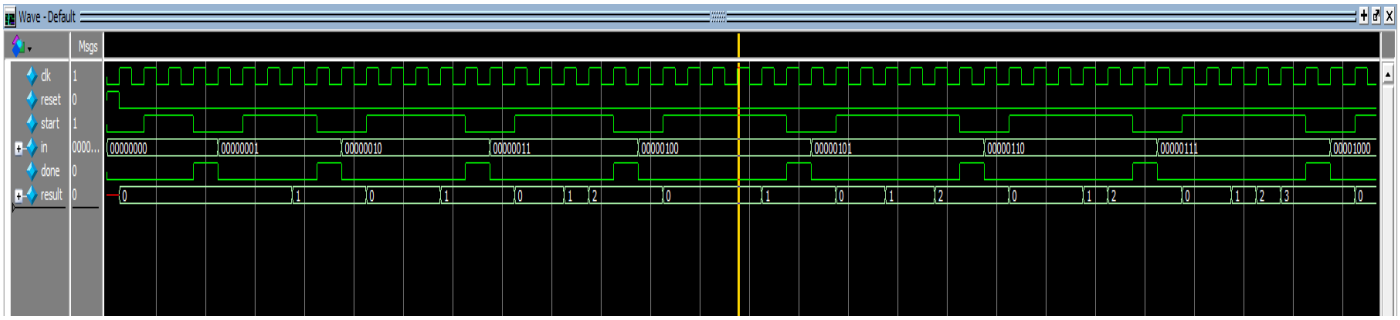
1. Loads b1001_0110 into A. This also means a0 becomes 0b0.
2. Shifts A via *shift_A* at 250ps. Thus A becomes 0b01001011 and a0 becomes 0b1.
3. Increases result via *incr_result* at 350ps, so result becomes 0b0001.
4. Clears result via *clr_result* at 550ps. The signal *result* becomes 0b0000.
5. Finally, loads 0b00000000 into A at 650ps, and *zero_A* becomes TRUE.

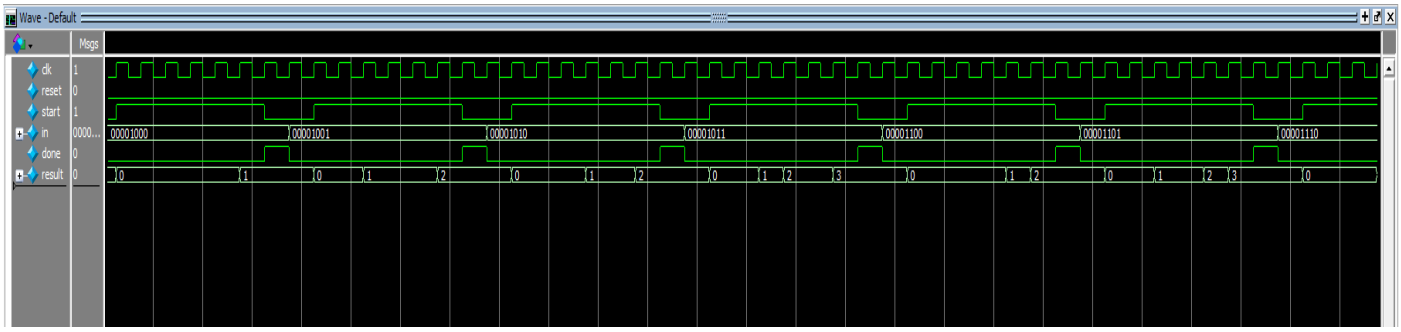


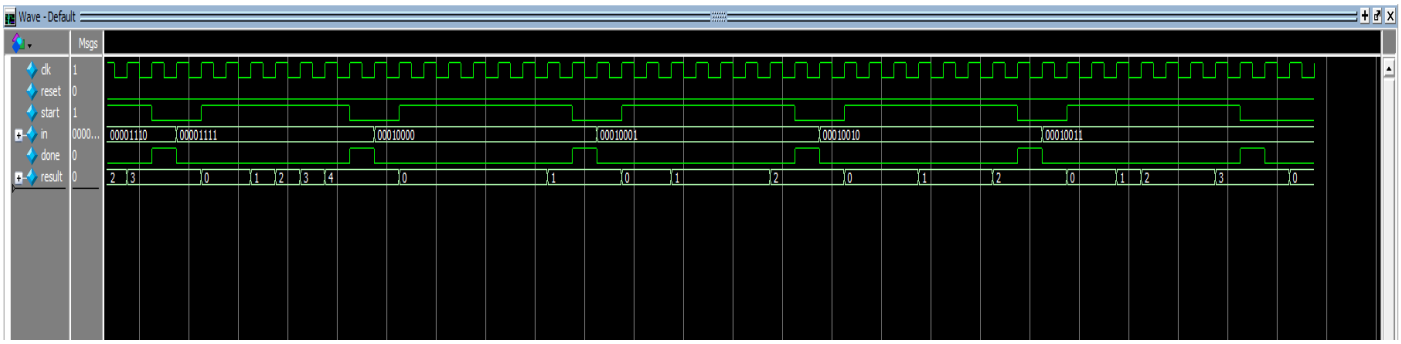
Bit_counter

The testbench goes from *in=0* to *in=19*. We can see that all the results are correct. For example:

1. In=0b0000_0000 gives result=0
2. In=0b0000_0001 gives result=1
3. In=0b0000_0010 gives result=1
4. In=0b0000_0011 gives result=2
5. And the module continues until in=19=0b0001_0011 which gives result=3.



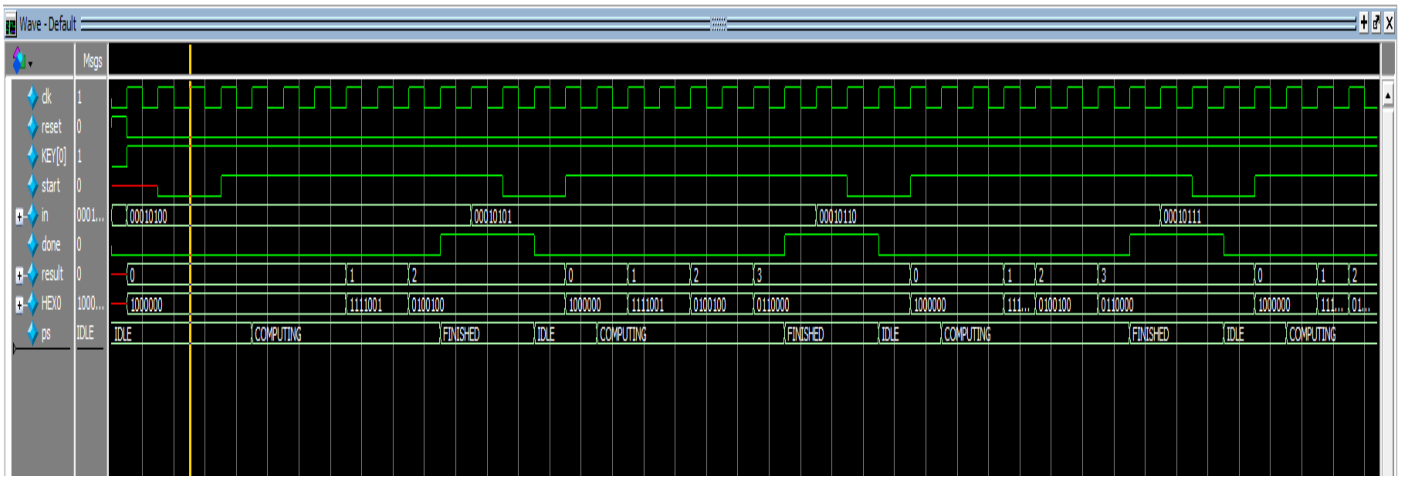


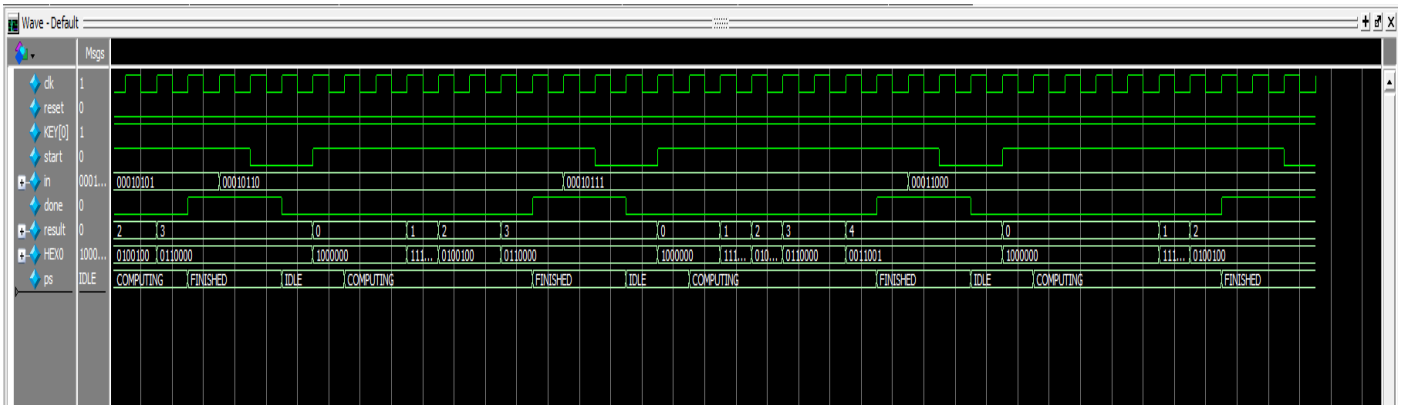


DE1_SoC

The testbench tests inputs from 20 to 24 and correctly outputs the results:

1. In=0b0010100 results in 2 bits set
2. In=0b00010101 results in 3 bits set
3. In=0b00010110 results in 3 bits set
4. In=0b00010111 results in 4 bits set
5. In=0b00011000 results in 2 bits set



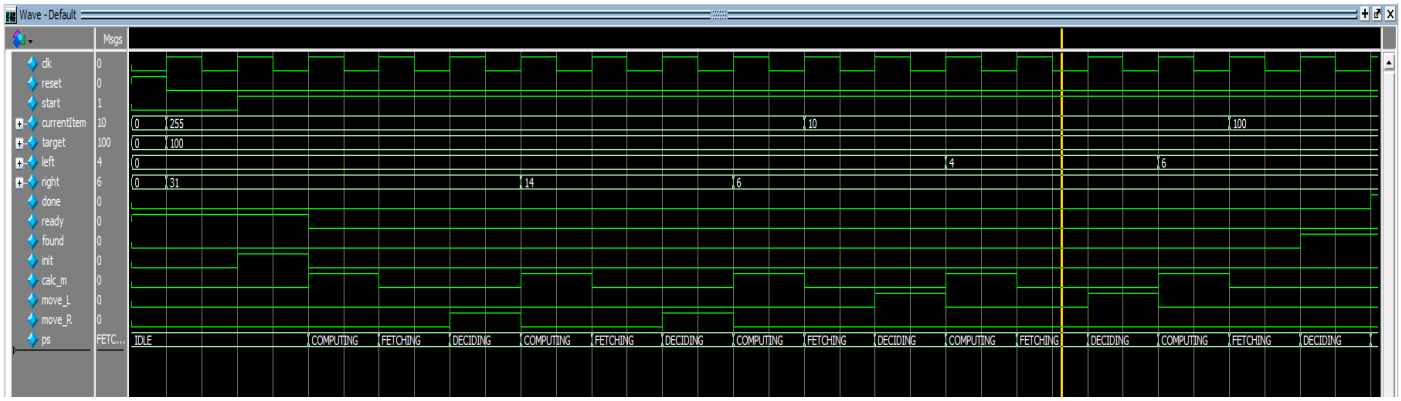


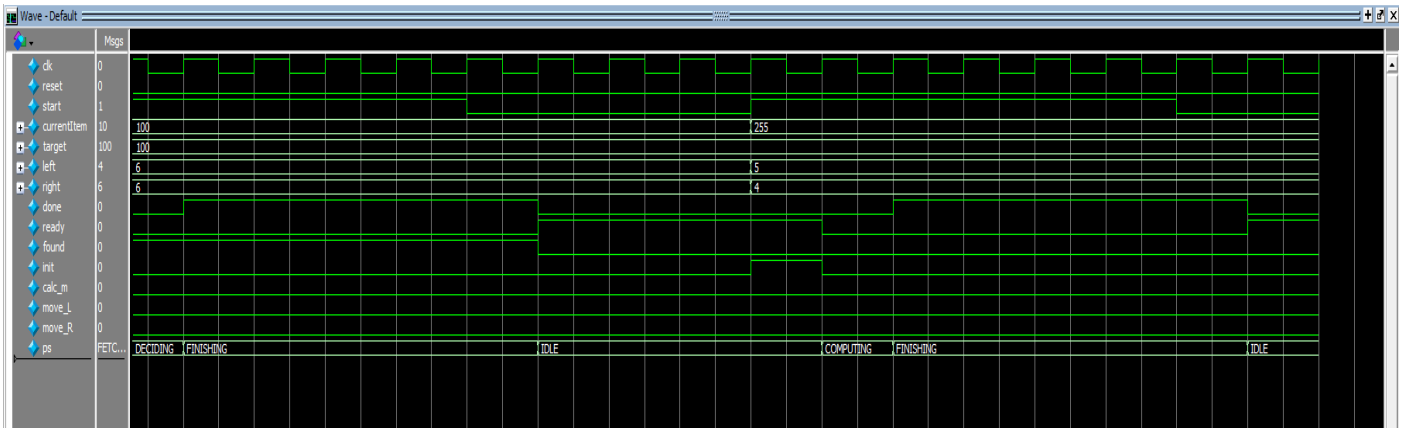
Task #2

Controller

The testbench simulates the scenario of searching for number 100 at index 6. The simulation can be read in order:

1. First target of 100 is loaded, and the current item is simulated to be 255.
2. The controller asserts *move_R* since the current number is too big, so the number must be on the left-side.
3. The *currentItem* becomes 10 and the controller asserts *move_L* since the current number is too low, so the number must be on the right-side.
4. The simulated *currentItem* is still 10, so *move_L* is asserted again.
5. The simulated *currentItem* becomes 100, and the target has been found. Therefore, *found* is asserted and so is *done*.
6. The system goes back to *IDLE*.
7. Now tries to search for a non-existing item. The signal *done* does get asserted but not *found*.

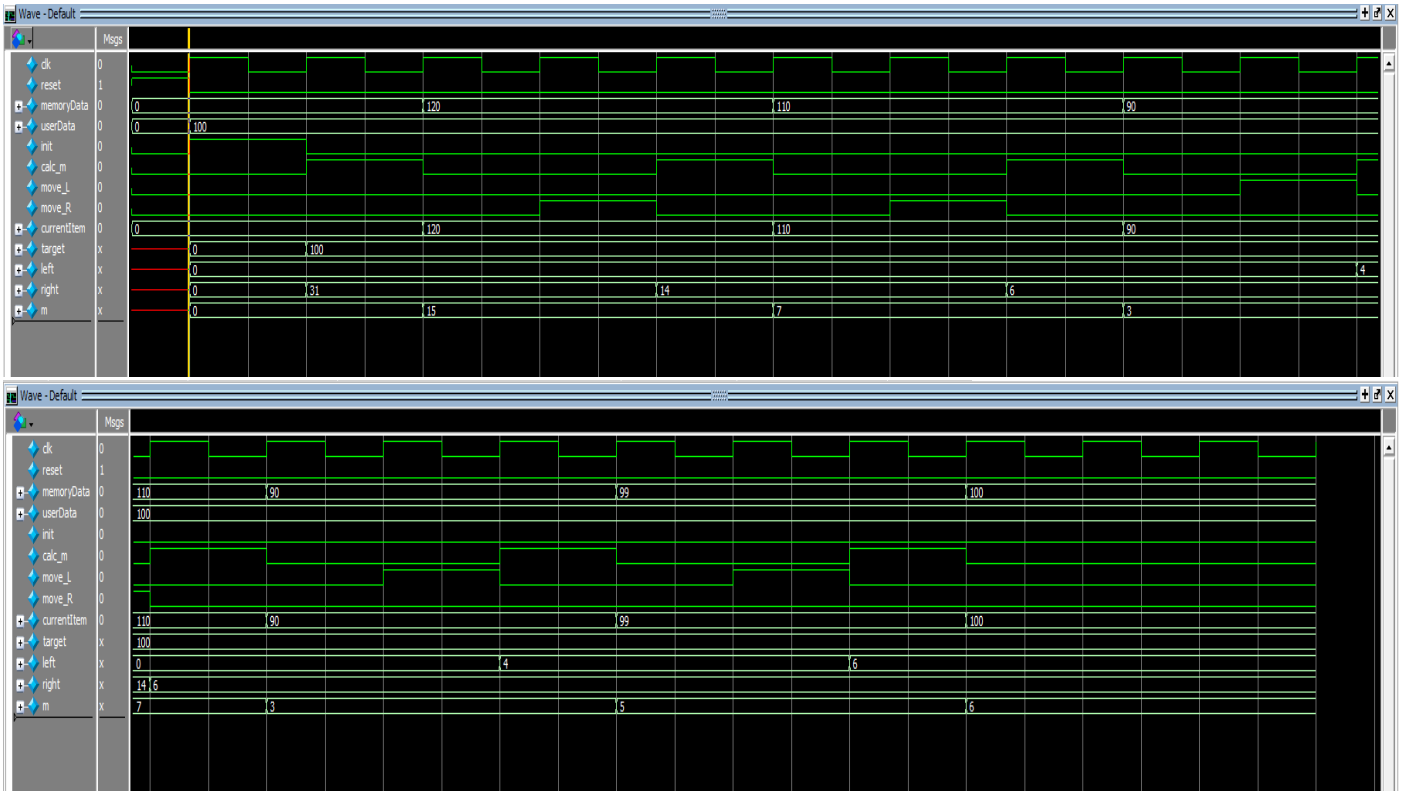




Datapath

Similarly, the testbench simulates the scenario of searching for number 100 at index 6. The simulation can be read in order:

1. Loads target 100. The search space is initially [0, 31].
2. First 120 at index 15 is too big, so the search space becomes [0, 14].
3. Next, 110 at index 7 is still too big, the search space becomes [0, 6].
4. Then, 90 at index 3 is too small, thus the search space becomes [4, 6].
5. Number 99 at index 5 is too small, and the search space becomes [6, 6].
6. Finally, the target 100 is found.

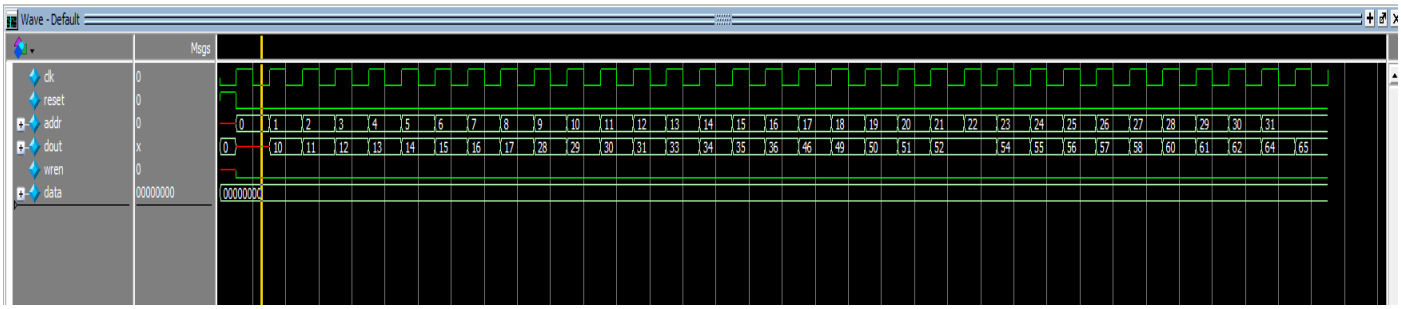


Ram32x8_wrapper

Before making the final *binary_searcher* module, I tested being able to read from memory and made the initialization file worked. The following screenshot shows the memory initialization file *my_array.mif*:

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	10	11	12	13	14	15	16	17	.....
8	28	29	30	31	33	34	35	36	....!"#$
16	46	49	50	51	52	52	54	55	.1234467
24	56	57	58	60	61	62	64	65	89:<=>@A

The file has been attached as *ram32x8_wrapper.sv*, and its testbench goes though memory. The value of *dout* matches the picture above (each output comes 1 clock cycle later than its index because the RAM is registered at its input):

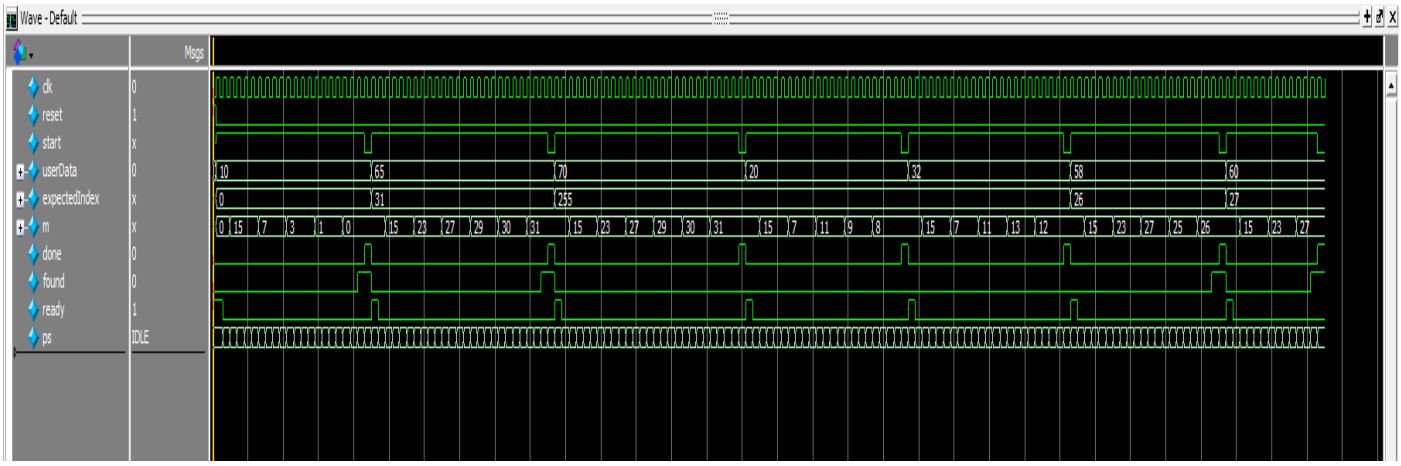


Binary searcher

The testbench shows the module and verifies its behavior. For testbench purposes, the wire *expectedIndex* holds the expected index that *m* should match when *target* is found in memory. Otherwise, *expectedIndex* is 255 (if the item is not in memory). The screenshot shows the following sequence:

1. Finds number 10 at index 0 (*done* gets asserted and *m=0*)
2. Finds number 65 at index 31 (*done* gets asserted and *m*=31)

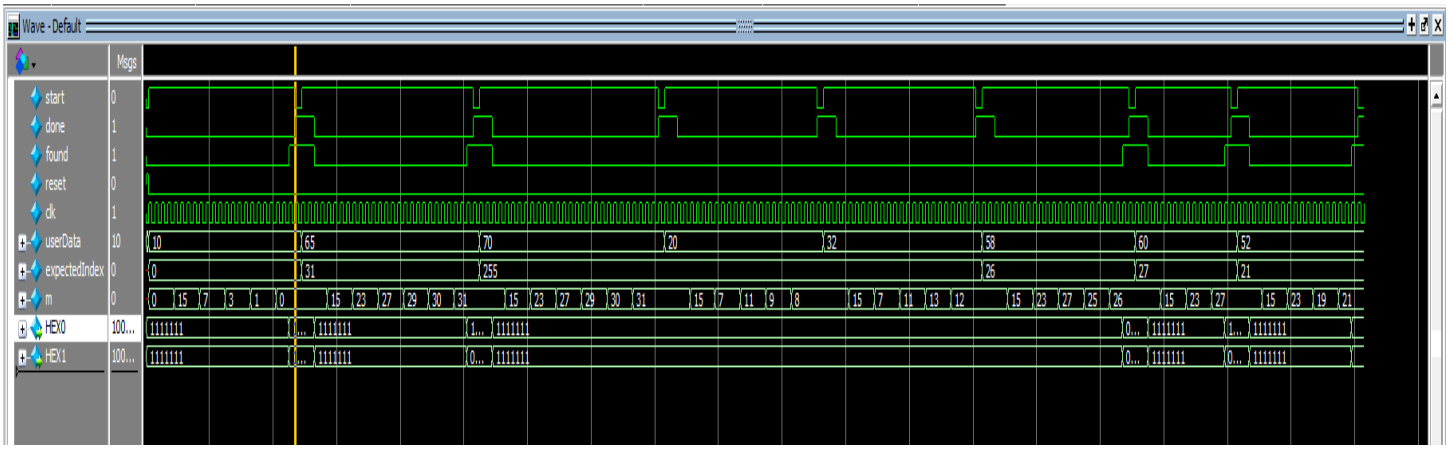
3. Doesn't find 70, 20, or 32 since they are not in memory (*done* is asserted, but not *found*. The value of *m* could be anything since the item wasn't found in either case*)*.
4. Finds 58 at index 26 (*done* gets asserted and *m=26*).
5. Finds 60 at index 27 (*done* gets asserted and *m=27).*



DE1_SoC

Performs the same tests as *binary_searcher,* except that now it also looks up a repeated value in memory (number 52):

1. Finds value 10 at index 0.
2. Finds value 65 at index 31.
3. Can't find values 70, 20, or 32 because these don't exist in *my_array.mif*.
4. Finds value 58 at index 26.
5. Finds value 60 at index 27.
6. Finds value 52 (which is a repeated value) at index 21.



Experience Report

One challenge was getting the memory initialization file to work. I kept trying to read the memory for a while but all I was able to read were zeros. In order to fix this, I noticed an error in the MultiSim console that told me it couldn't read the memory file, and thus it chooses to initialize the memory to zero. By looking up online, it was observed that the memory initialization file must be in the same directory as the simulation. However, with my current project structure this was different than the folder where I hold my SystemVerilog files, which is where Quartus will by default save the memory initialization file. Therefore, I had to move this file to the correct place and add it to the project.

One tip for future laboratories is to remember that the memory initialization file must be in the same directory as the simulation files for MultiSim to work properly. Another tip is to read the error messages

from MultiSim carefully, although sometimes it is easy to miss them due to the small font size and bright font colors.

Feedback for the laboratory is positive, since I now feel more comfortable developing sequential algorithms in hardware, thus I believe this laboratory gave me practice that I highly needed. Another positive is the use of memory modules from the IP catalog and the memory initialization file, since we only had used it once, and I think having an opportunity to practice using these tools is great.

Estimated total time to complete the laboratory: Around 15 hours.