Ishan Dane NetID: idane
Ismail Memon NETID: ishmemon

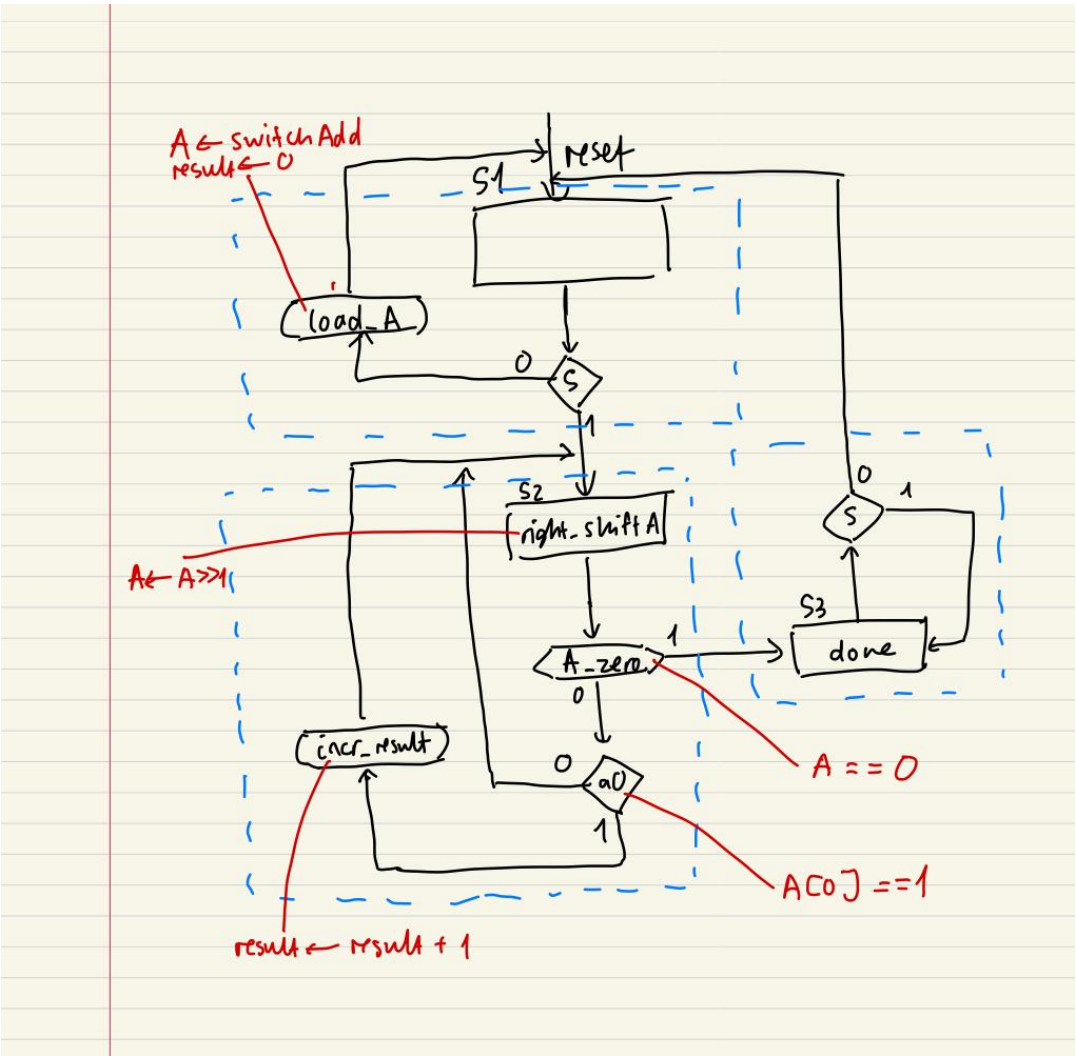
Lab 4

Design Procedure

In Lab 4, we are tasked with creating and implementing ASMD's as a software algorithm that is inputted into the FPGA board. In Task 1, we are provided with an ASMD diagram for a 1 bit counter,which counts the number of 1's out of an 8 bit input, and tasked with implementing its design onto hardware. In Task 2, we are tasked with creating a binary search algorithm that accesses a RAM. In the algorithm, we split the array addresses for the RAM in half again and again based on whether the user input was less than or greater than the value we were assessing. If the data input was found, the address in the array MIF file is displayed on the HEX0 and HEX1. If the data input was not found, LEDR[0] will not display to let the user know that the data input was not in the file.

Task #1:

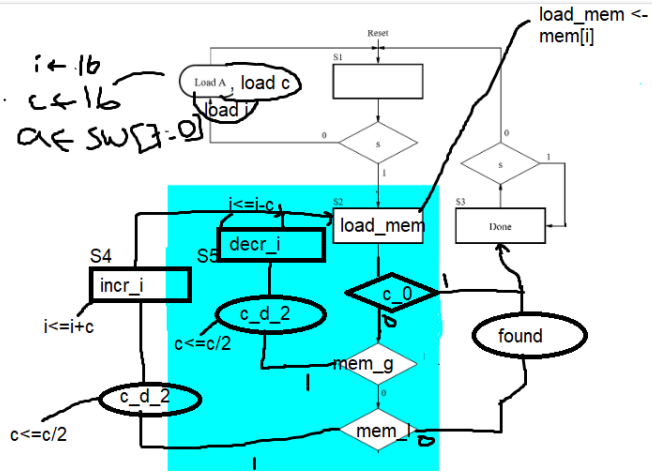
For task 1, we first implemented the ASMD diagram that counted the number of 1's in an 8 bit input provided to the system by SW[7:0] and displayed it on HEX0. We first labeled the status signals: A_zero, a0, s(start); and control signals: right_shiftA, incr_result, load_A, done; involved. From this, we created the datapath module and the controller module for the ASMD diagram. The control signals' logic were implemented in the datapath module and the ASMD states and outputs were assigned in the controller module. These two modules were then connected through the task1 module where they instantiated. The top level DE1_SoC module is used to connect task1 to HEX0 and switches SW[7:0]. Start signal 's' was assigned to ~KEY[3] and 'reset' was assigned to ~KEY[0].



Task #2:

The objective of Task 2 was to implement a binary search algorithm on a sorted array. The array would be stored in the FPGA memory and would be a 32 word memory with 8 bit words. The user entered in a value they wanted to search for using SW[7:0]. They used KEY[0] to reset and KEY[3] to start. When they had not started, the load signals make sure to load 16 (the middle value) to both i and c. When we have started, we load the memory using the memory file and then compare it move on. If our counter is 0 (meaning our iterations are complete) we can go straight to Done and found is not asserted. If our memory value is greater than the target, we halve c and assign i as i-c. If our memory value is less than the target, we halve c and assign i as i+c. This way we are splitting our range in half and looking at the place we want depending on the value of the comparison. We

know this works since the array is sorted, so things only increase as we increase our addresses.



c is the counter of the iteration. The first had 2 sections of 16, second has sections of 8, then 4, then 2, then 1. So we can start at 16 and keep dividing. The i is the memory address.

i starts at 16 and incr or decr depending on whether the load_mem is < or > than a. Once it is neither we found it so we are done.

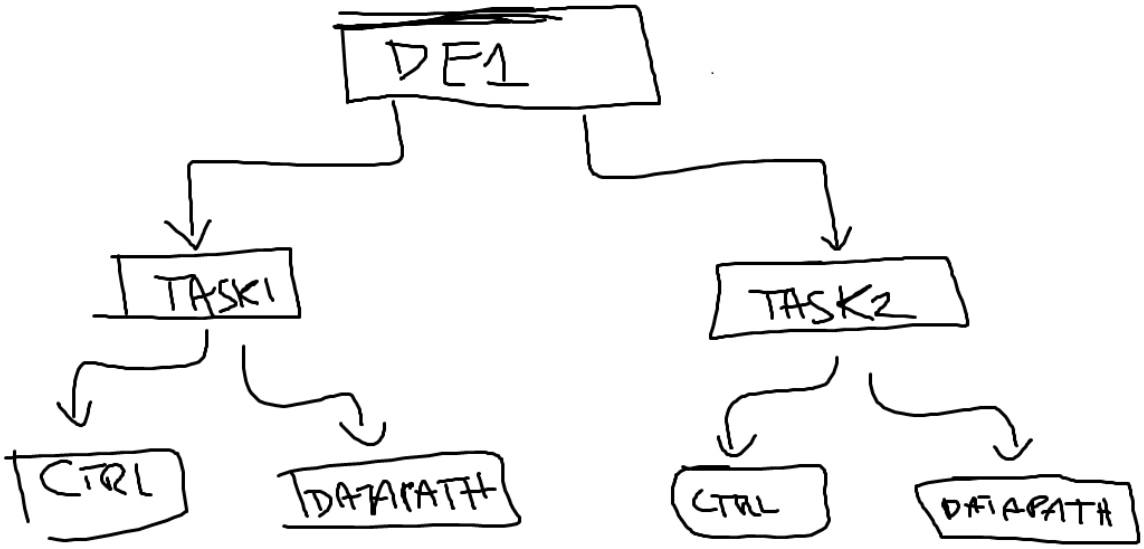
If we have not found it that means we need to check for 2 things, the load_mem is not equal to a AND c = 0 bec 1/2 = 0. So if c is ever 0 we know it's not found.

-

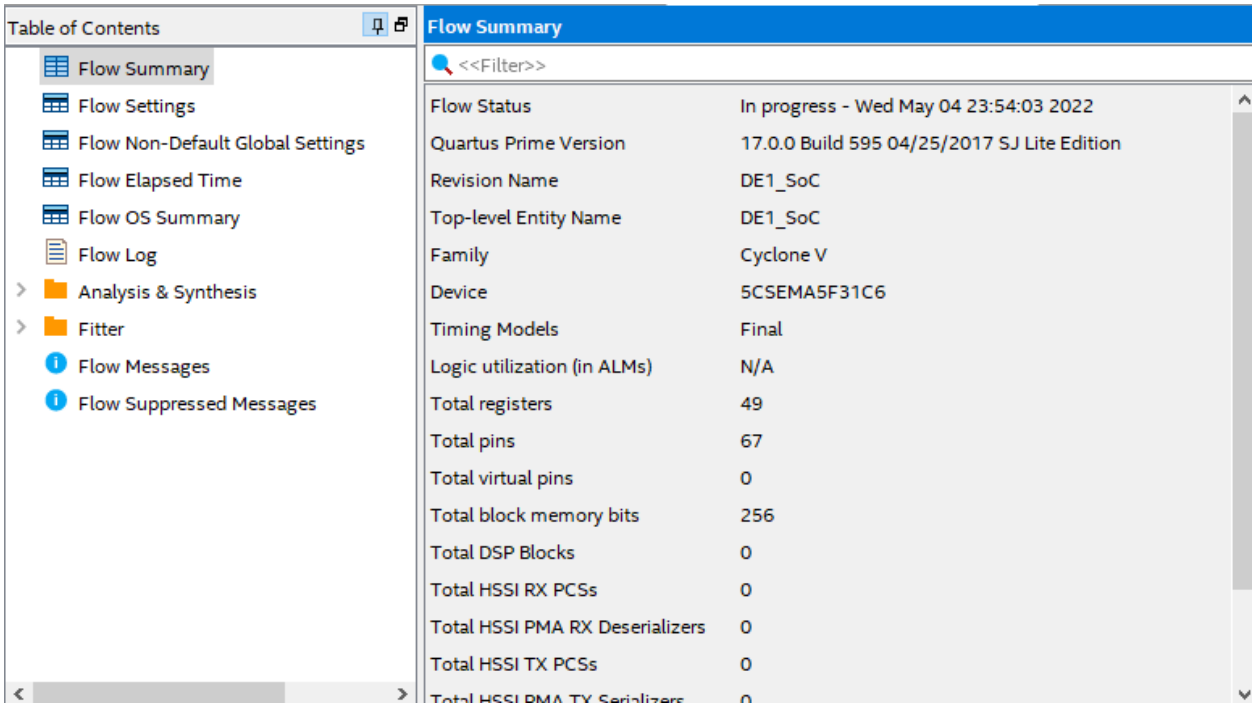
Overall System (Top level):

Our top level module is DE1_SoC.sv. Through this module, we toggle between task 1 and task 2. In both tasks SW[7:0] was used as input, KEY[3] as start and KEY[0] as reset. SW[9] is used to toggle between the 2 tasks. The outputs of task 1 are done1 and result1 and the outputs of task 2 are done2, result2, and found. The seg7 display is instantiated in the top level module to display results: HEX0 is used for task 1 results and HEX0 and HEX1 is used for task 2 results. Found is connected through to LEDR[9] and LEDR[0]. All modules use clk and reset as inputs which synchronizes the circuits.

Block Diagram for System:



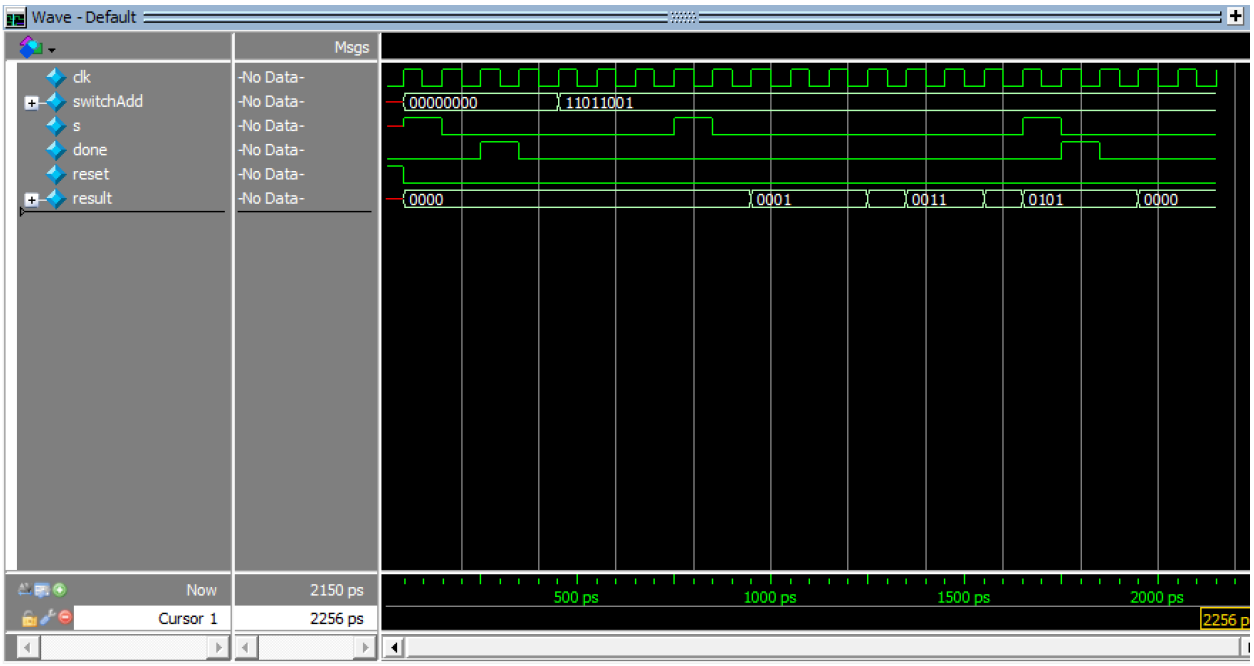
Flow summary:



Results:

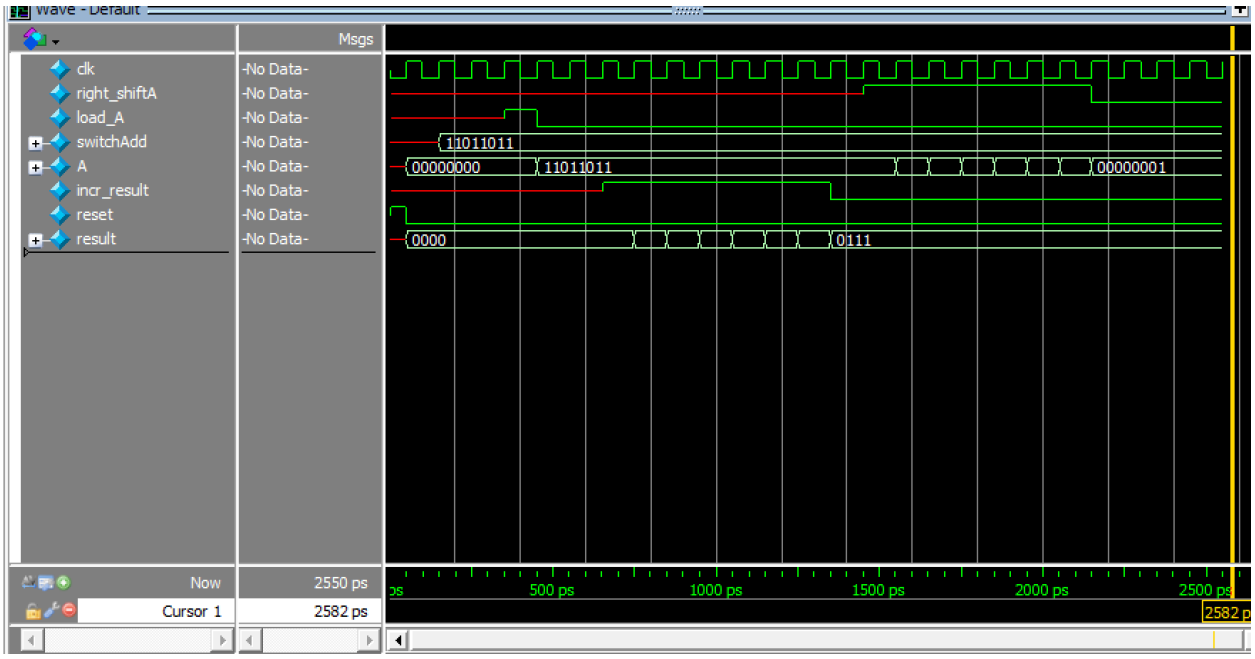
Task1 Modelsim:

To test Task1 , we made a testbench that would simulate an 8 bit input provided by switches SW[7:0] into the system. According to the controller module that implements the ASMD states, signal ‘s’ is used to start the counter that counts the input. When the algorithm is done counting, it outputs a ‘done’ signal. The testbench tests that the correct outputs are provided and the algorithm correctly counts the number of 1’s in the input and outputs done when it finishes counting. The switches are first set to all 0 to test the counter for the 0 case and then an input with 6 1’s were provided with the start signal s to test the counter and algorithm. The results are shown in the figure below.



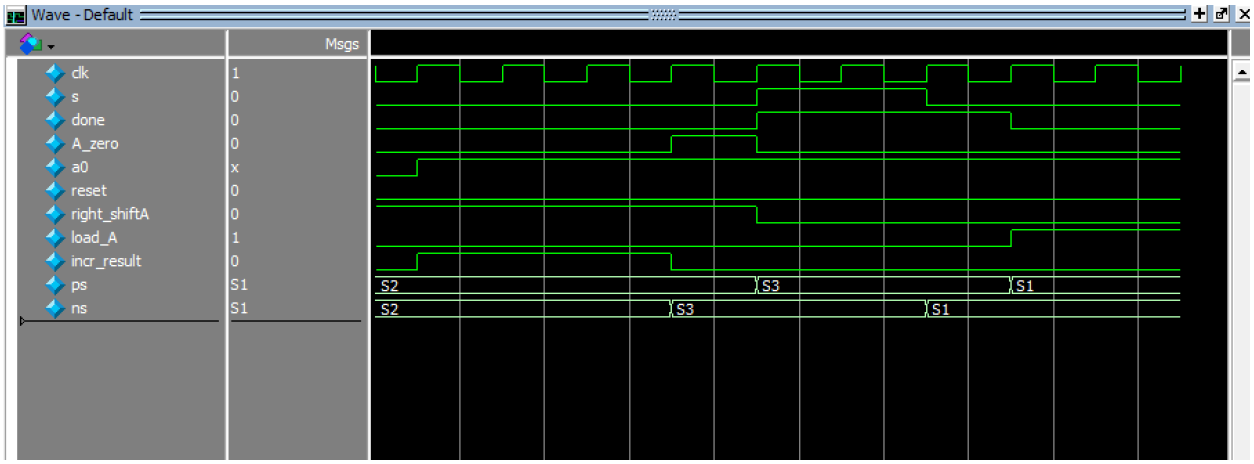
Datapath1 Modelsim:

For this testbench, we tested whether the datapath logic was being correctly outputted. Signals incr_result, load_A and right_shiftA were tested along with the counter result to test that they worked correctly. Signal switchAdd was provided a 8 bit input and load_A was set to 1 for a clock cycle to test correct loading of A. Signal right_shiftA was tested if it shifted A and incr_result was tested to see if it incremented the counter result. The results are shown in the figure below.

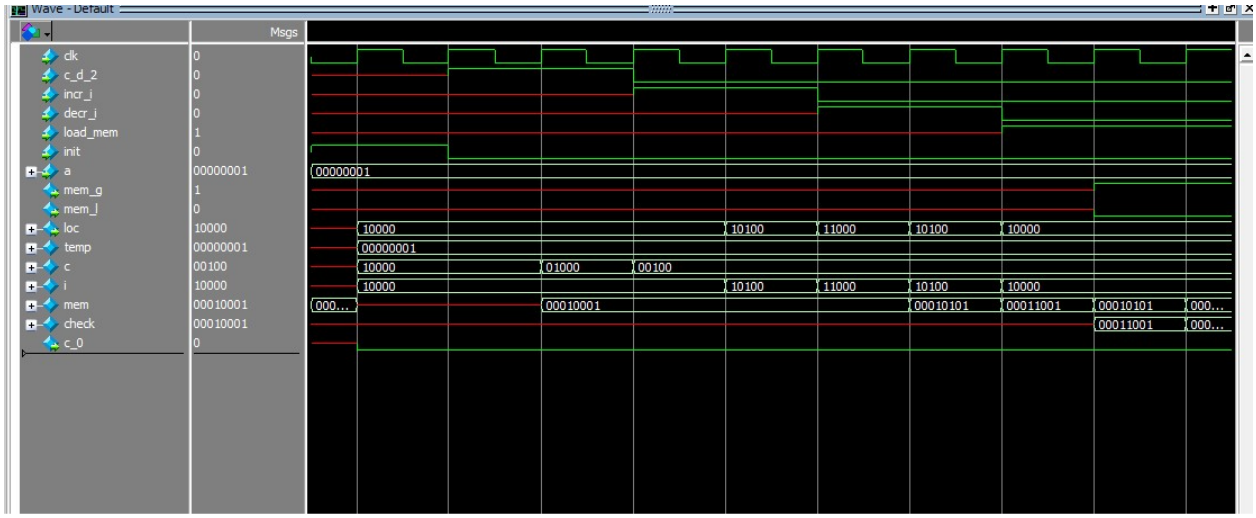


Controller1 Modelsim:

For this testbench, we wanted to test the system to see if the controller module correctly transitioned through the states provided the conditions to transition. The results of the test are shown below which shows the correct transition throughout the states.



Task2 ModelSims:



Experience

We found this lab to be a little simpler than the previous lab but still quite difficult. For task 1 we were provided with the ASMD diagram which made it simpler to separate out the control signals, status signals, external inputs and outputs. This made it much easier to understand and design the controller and datapath modules and implement them together into task1. For task 2, we were also provided with the instructions to create the RAM and implement it. The difficult part was creating the ASMD diagram and implementing the controller and datapath modules. What we found most difficult about this lab was creating the individual testbenches for each task as we often found ourselves making port errors and syntax mistakes. This could have easily been avoided if we had planned out the testbenches and taken more time to carefully debug them instead of rushing them. Furthermore, we ran into issues when we tried to connect the seg7 module individually in the task modules. We fixed this by porting the seg7 module through the top level module instead.

This lab took us approximately 9 hours, broken down as follows:
Reading – 1 hours
Planning – 0.5 hours
Design – 1 hour

Coding – 2.5 hours
Testing – 1 hours
Debugging – 5 hours