Khoa Tran and Ravi Sangani

EE 371

May 15, 2021

Lab 4 Report

**Procedure**

**Task #1**

In order to approach this task, we first drew up on the block diagram in order to understand the connection between the controller and datapath modules for implementing the bitcounter algorithm. After observing the pseudocode, we then designed the ASMD chart in order to understand the RTL operations needed at certain states and the different transitions between the states in order to correctly increment the count based on if a 1 bit exist on the last index of the given data as the data shifts to the right on the correct state. Besides drawing the ASMD chart, we constructed the finite state machine diagram in order to understand the transitions between the certain states as well as the datapath to see the counter and shifter and their inputs of loading the data and enabling the counter or shifter to start. As seen in figure 1, the block diagram depicts the connection between the modules, while in figure 2, the ASMD chart displays the RTL operations at each state and the input and output transitions of certain logic variables.

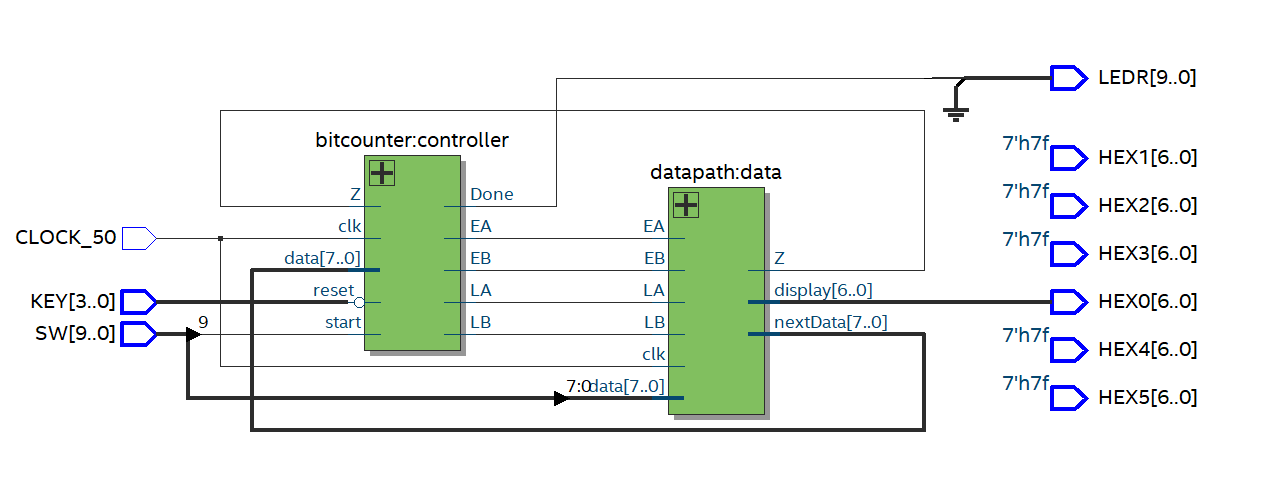
****

Figure 1: Block diagram for task 1

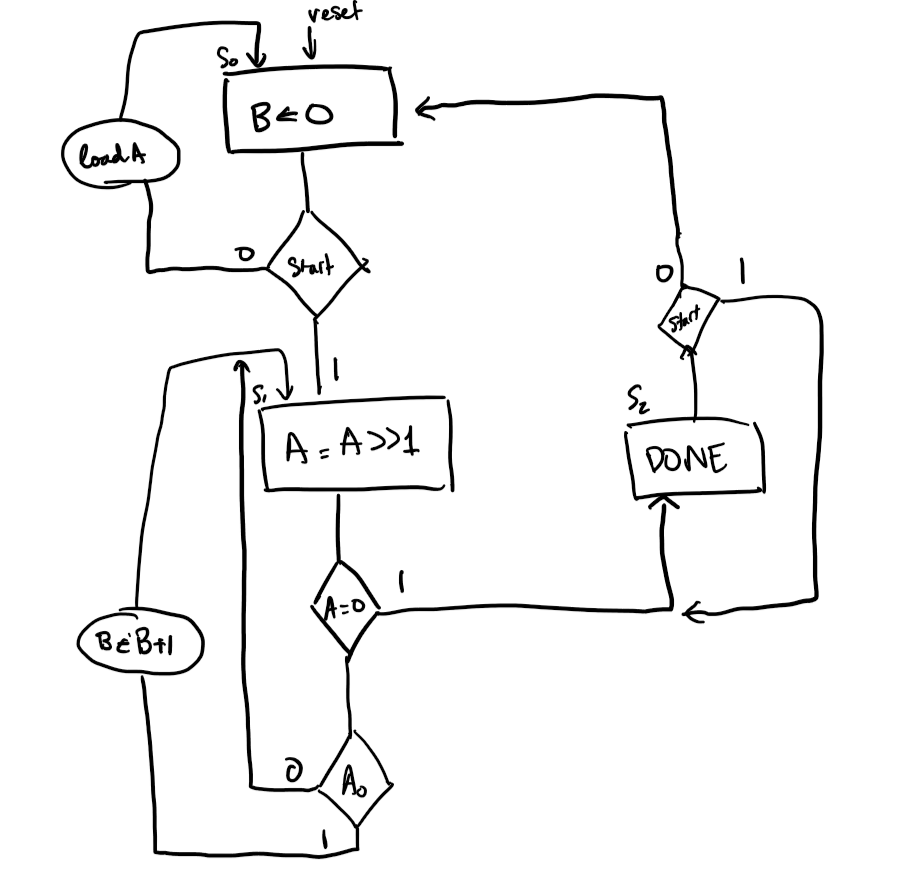


Figure 2: ASMD chart for task 1

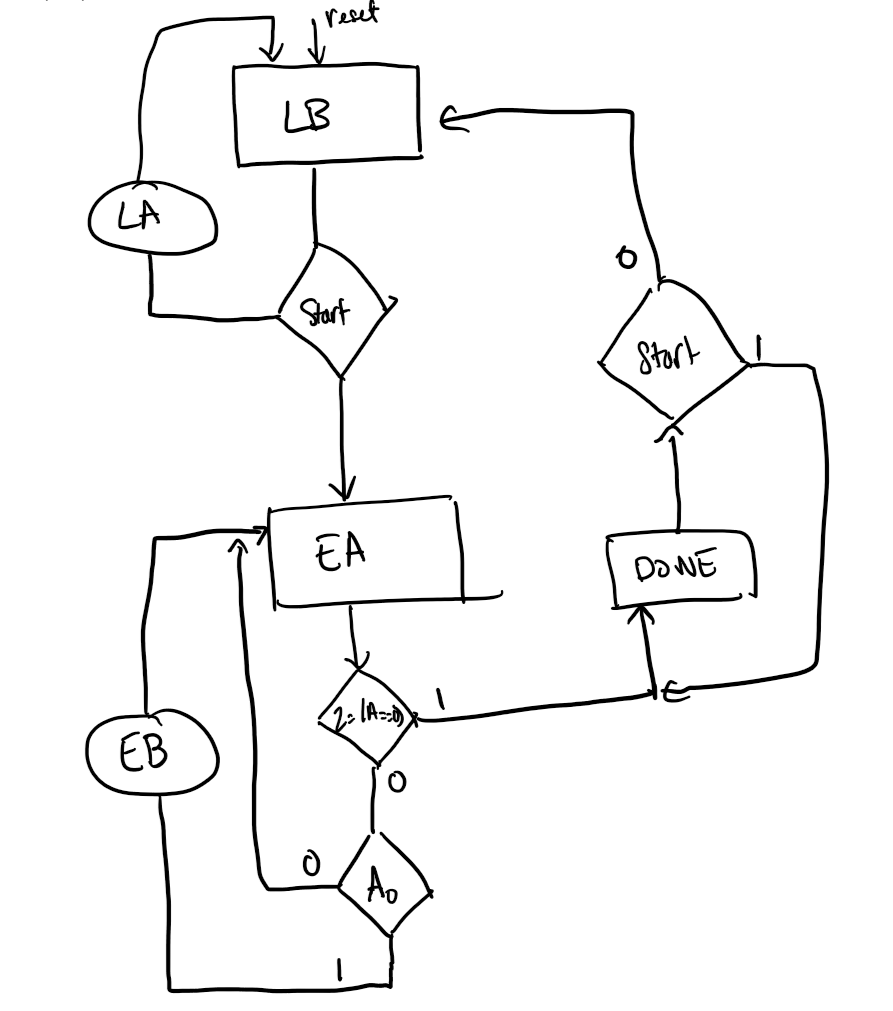


Figure 3: ASM for controller for task 1

**Task #2**

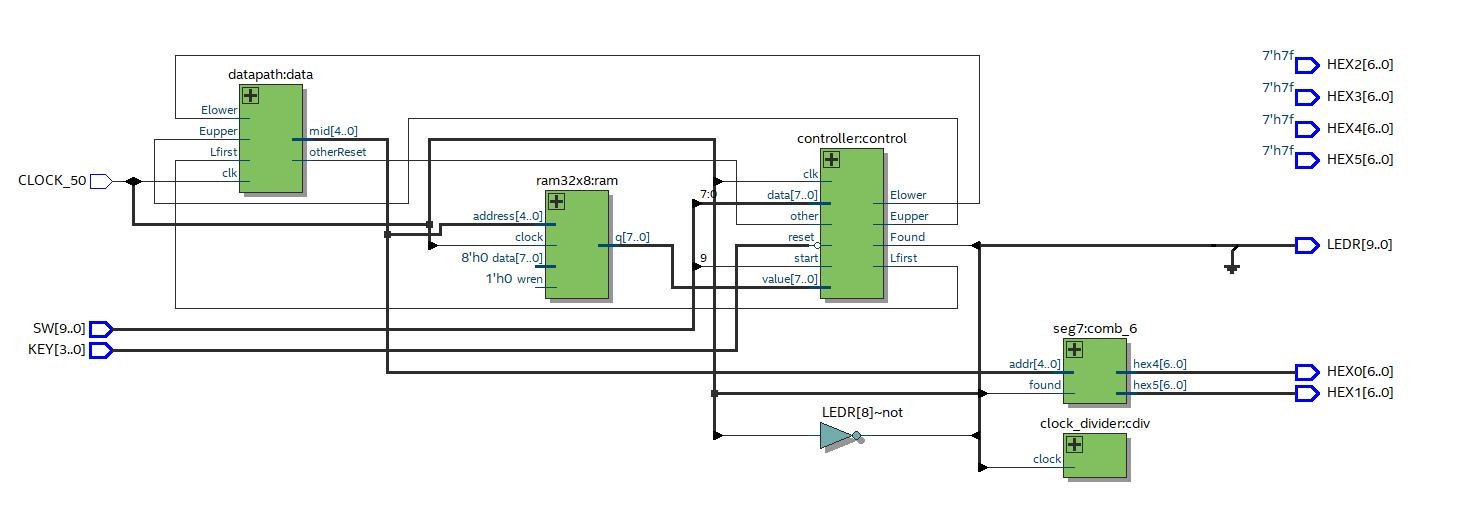
Approaching this problem, we first looked at the pseudocode for a binary search algorithm on a sorted array. In this problem, our sorted array is a 32x8 ram that has a sorted initial memory initialization. From the pseudocode, we then constructed the ASMD chart as well as the ASM for controller in order to understand the necessary outputs at certain states for the datapath to manipulate and change the value of lower and upper bounds, minimizing the search to a smaller range of addresses based on given target data and current data at the middle address. The ASMD chart shows the RTL operations at every state, including both datapath and controller operations, while the ASM for controller shows the state transition of the controller as well as the output of enabling lower and upper bounds for the datapath to receive and update the variable accordingly. Overall, the block diagram as well as the ASMD and ASM for controller allowed us to program the binary search algorithm on hardware with minimal issues. 

Figure 4: Block diagram for task 2

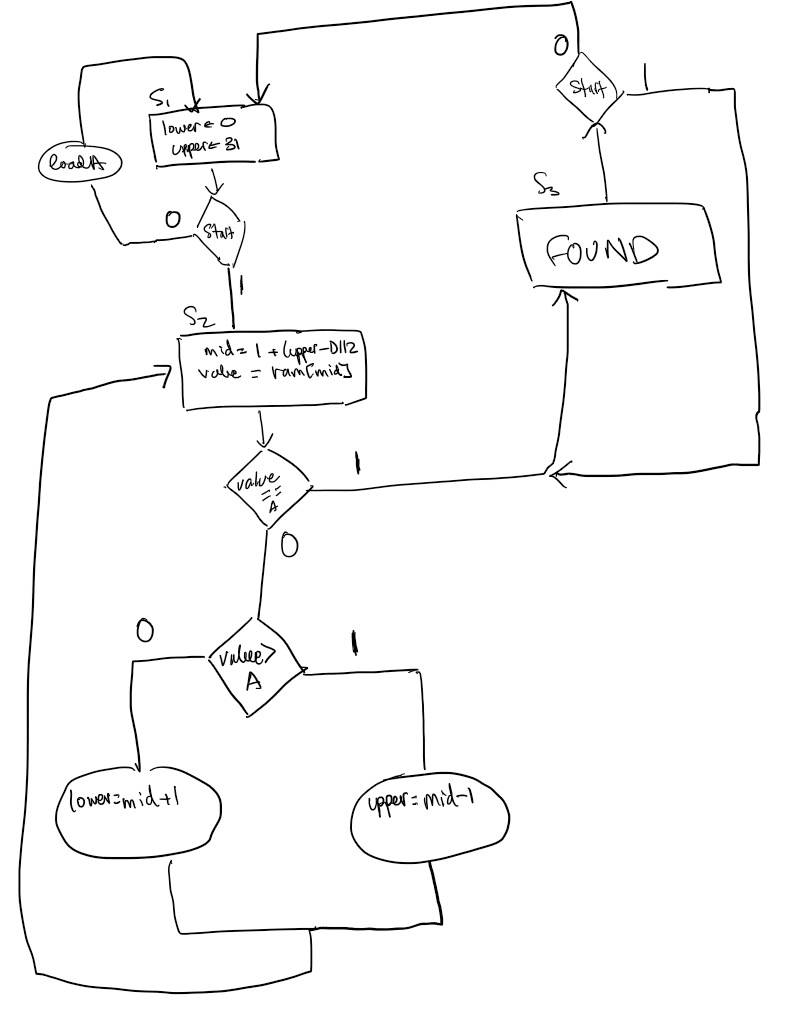
****

Figure 5: ASMD for task 2

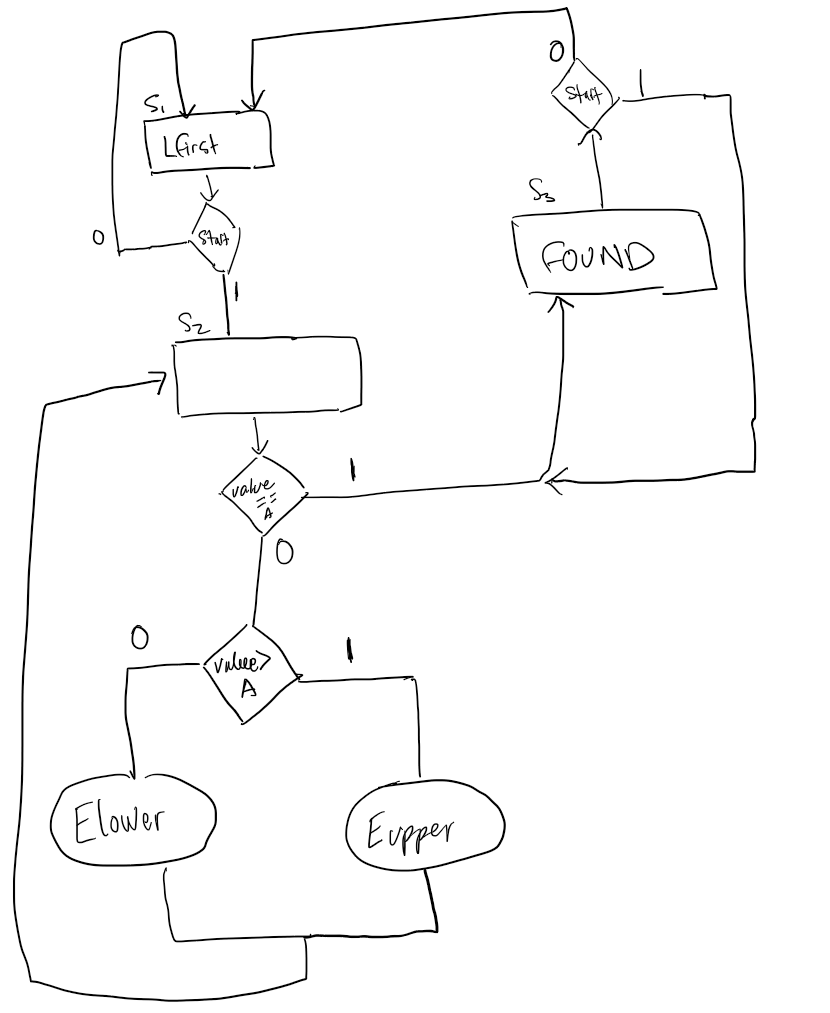


Figure 6: ASM for controller for task 2

**Results**

**Task 1:**

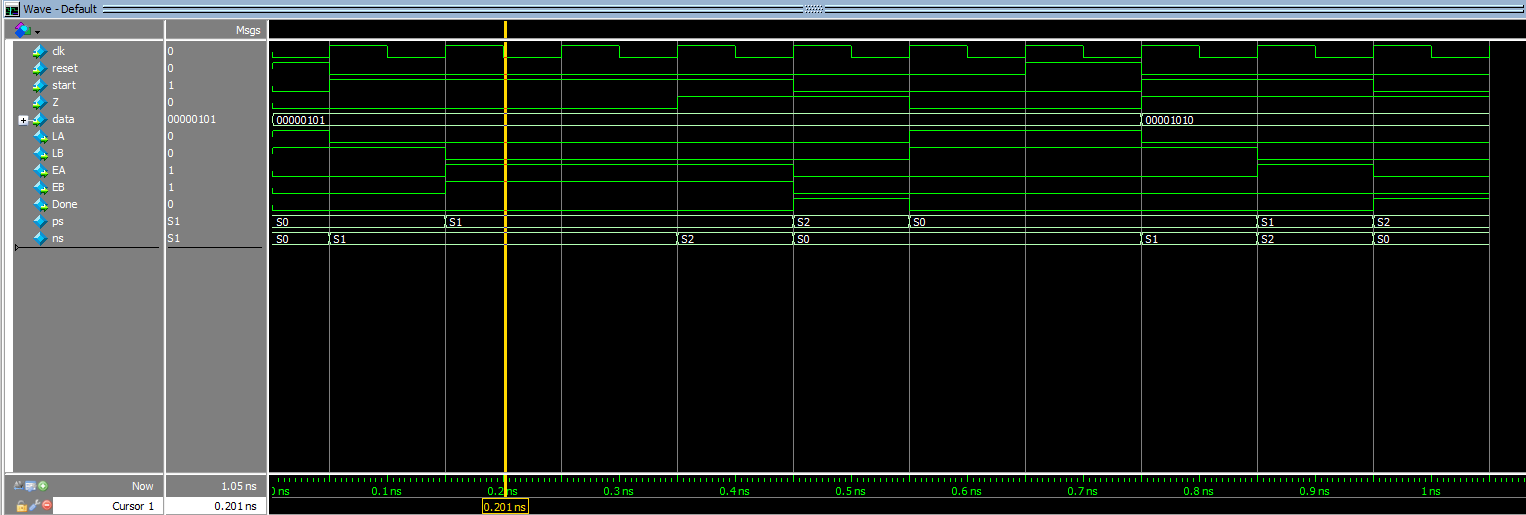
For the first part of task 1, we first simulated the bitcontroller module that takes in inputs of reset, start, Z, and data and outputting LA, EA representing loading and enabling A for the shifter of the data input. The module also outputs LB, and EB as the loading and enabling of B for the counter module, representing the count of 1-bit in the data binary input. This module, as seen in figure 7, the data input initially is 3, and based on start, and Z, the progression of present state is next state on the posedge clk. 

Figure 7: Waveform simulation for controller in task 1

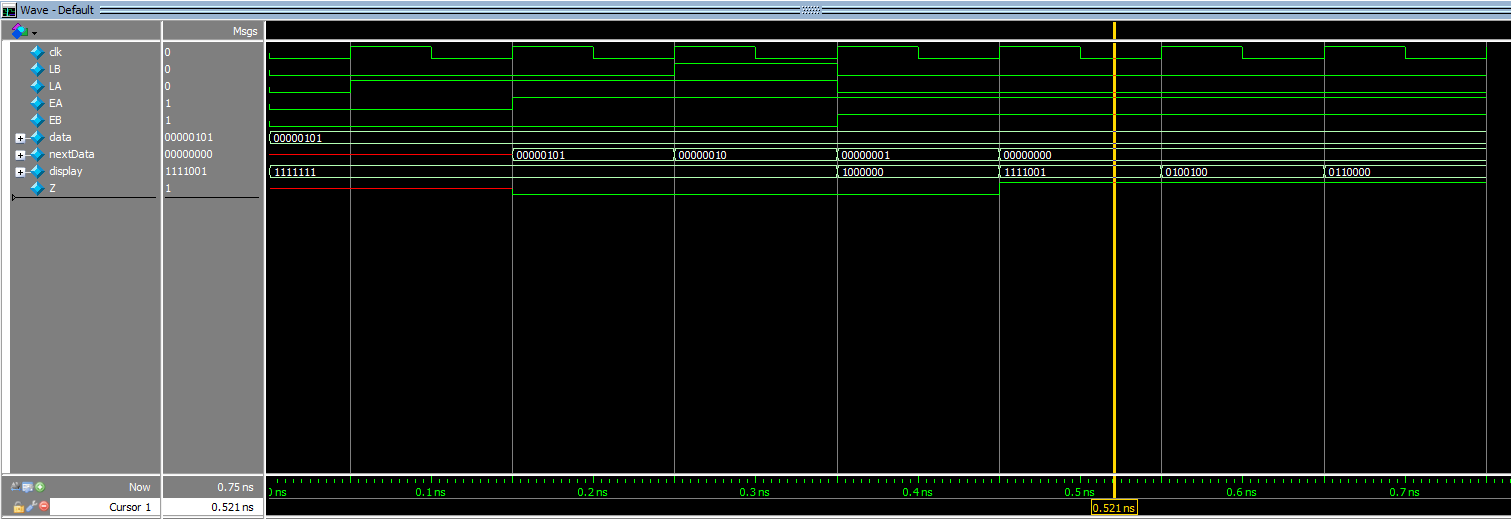
We also tested the waveform of the datapath, making sure the inputs of LA, LB, EA, EB loads the data of A and shifts according to the input of EA, and increment the count of B on EB and displaying the value on the hex display variable named display. The output of Z is also correct, making sure the output is 1 when the data is shifted towards the value of 0.

Figure 8: Waveform simulation for datapath in task 1

Lastly, in order to ensure proper operation of the DE1\_SoC in task 1, we simulated and made sure the waveform, in figure 9, outputs the correct value of the 1-bit count of a given data from SW7-0, on the hex0 display. As well as making sure LA, LB, EA, EB is passed and obtains the correct values on the posedge clk. As seen in figure 9, the output on hex0 is the hexadecimal equivalent value to the count of 1-bit on the input binary value.



Figure 9: Waveform for DE1\_SoC of task 1

**Task 2:**

First, we tested the simulation generated by the controller as we tested the inputs of start as well as data and value to see if the states are progressing on the condition that data is equal to value. Besides the state transition, the outputs of Found, Lfirst, Elower, and Eupper was correct along the posedge clk with Lfirst being true at S0, Found at the last state, Elower if at second state and data is greater than value, and opposite for Eupper. These outputs are for the controller to manipulate values of upper and lower to continuously input the value of the middle address in the ram.

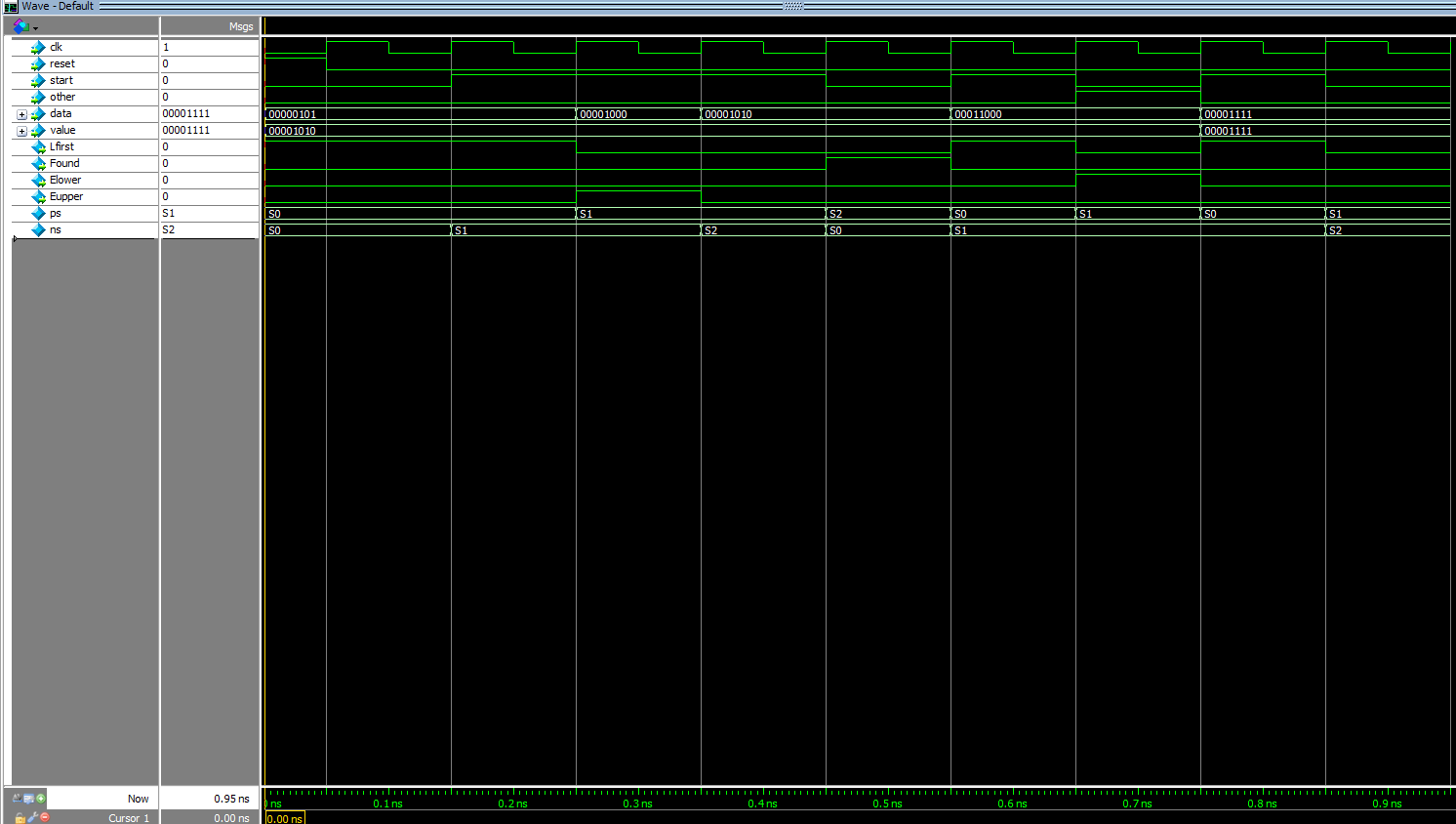


Figure 10: The waveform simulation generated by controller for task 2

As mentioned above, the datapath simulation takes in Lfirst, Elower, and Eupper, and changing lower and upper accordingly to output the middle value that is in between lower and upper.

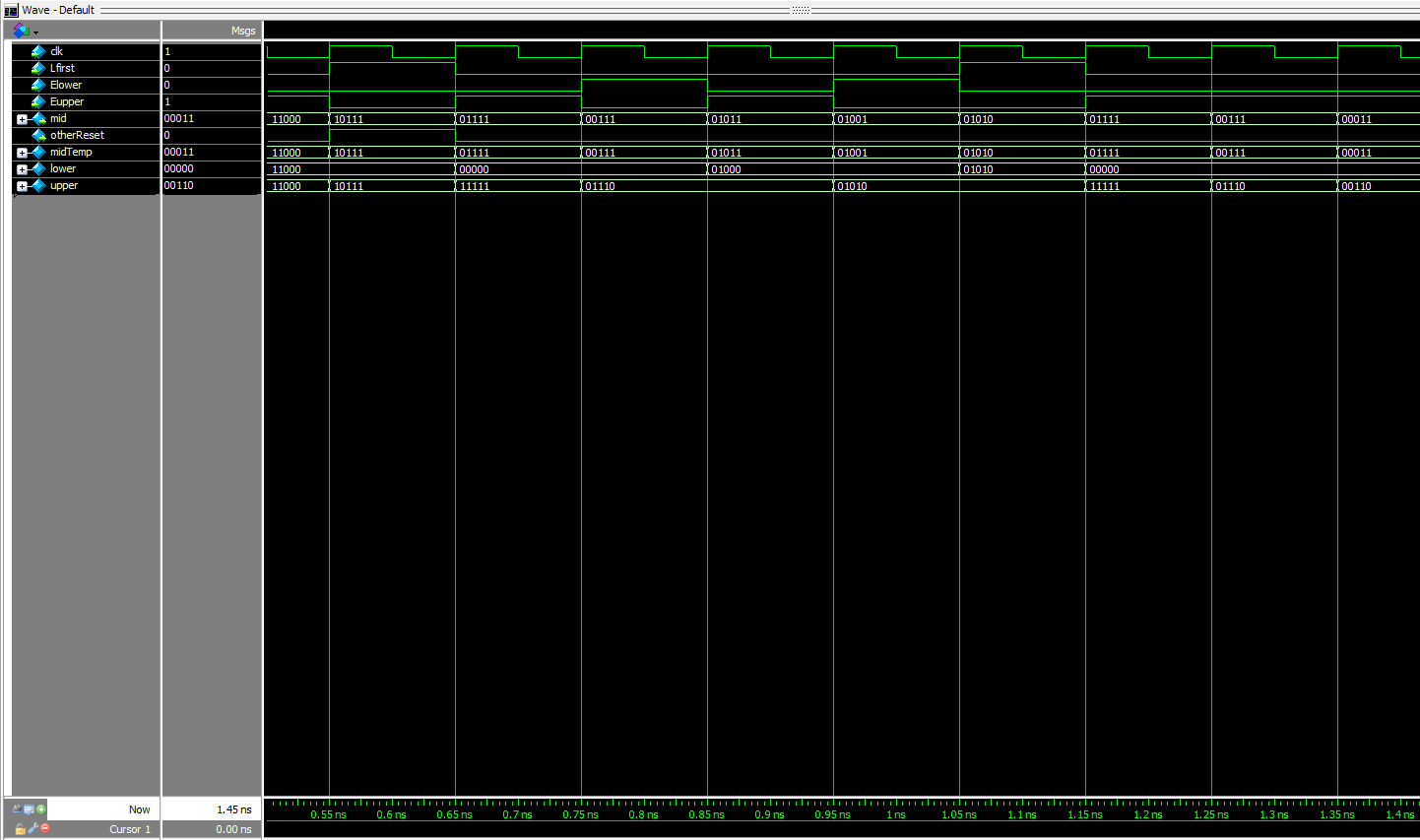


Figure 11: Waveform simulation generated by datapath for task 2

The waveform simulation for seg7 is for the display of the middle address value when the value equals the data, equaling to the found condition. As seen in figure 12, this matches the expected output.

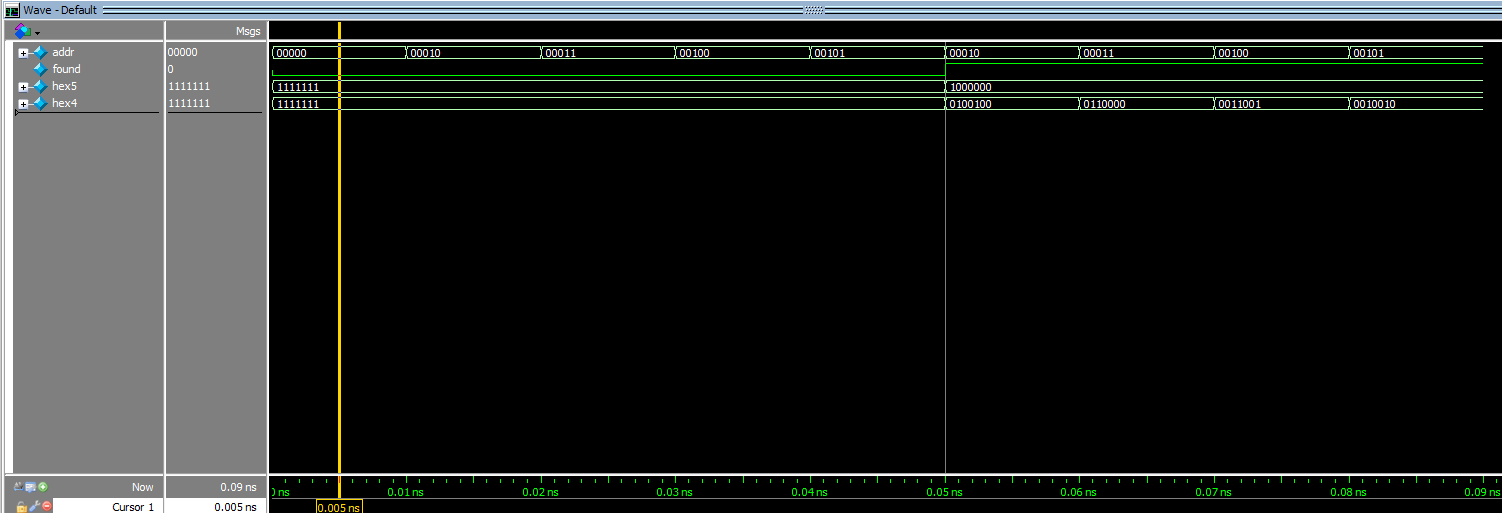


Figure 12: Waveform simulation generated by seg7 for task 2

The simulation for DE1\_SoC uses SW7-0 for the data input and SW9 for the start condition, and outputing the value of the address of the target data in the ram on hex0 and hex1, as well as found on LED9 and not found on LEDR8. As seen in figure 13, the not found light is on until the location of the given data from SW7-0 is located and the value is outputed on hex0 and hex1

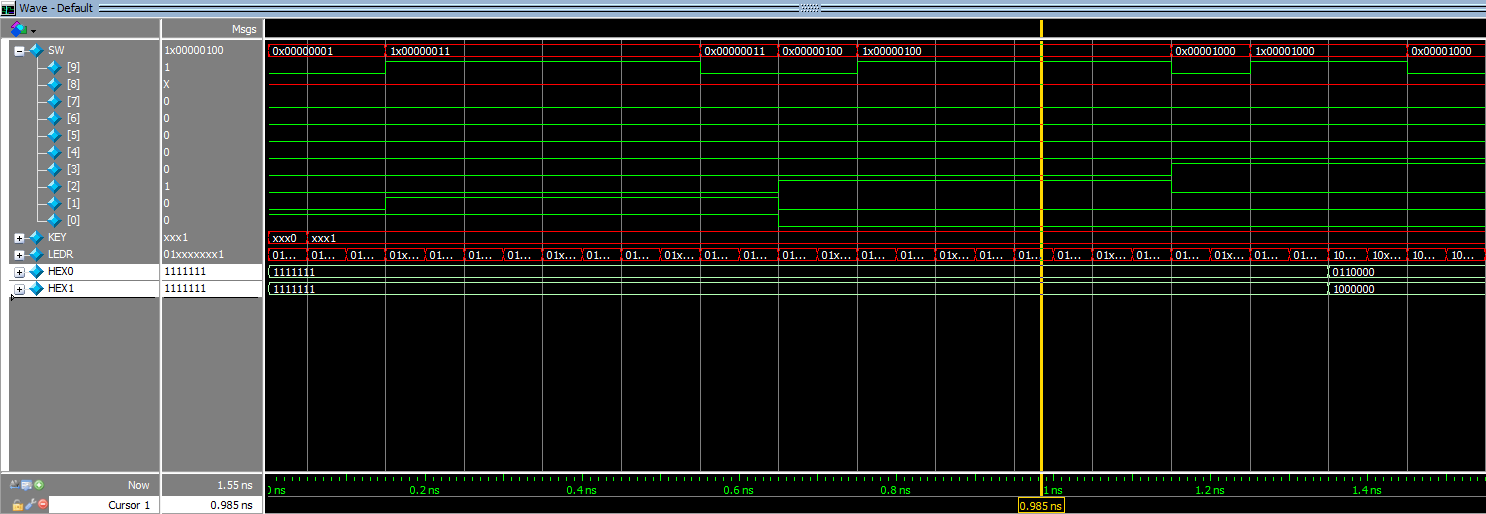


Figure 13: Waveform simulation generated by DE1\_SoC for task 2

Lastly, the simulation for ram32x8 uses inputs of data, address, and wren to output the value at the given address. In this simulation, wren is 0, so the ram is only outputing values given in the memory initialization file.

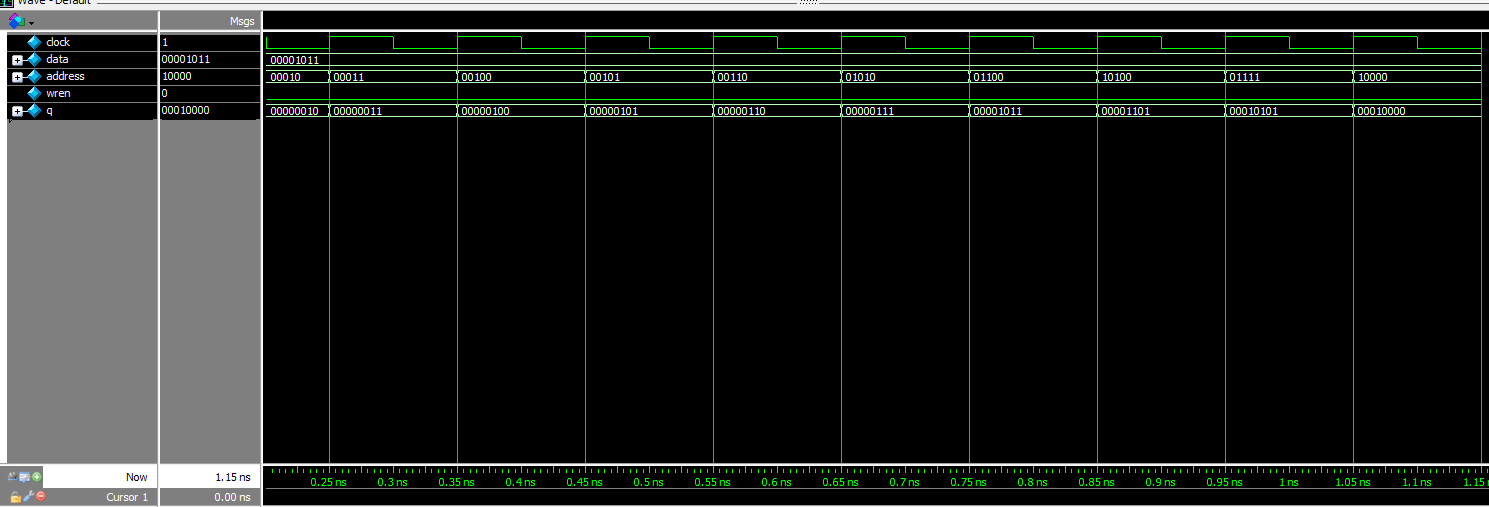


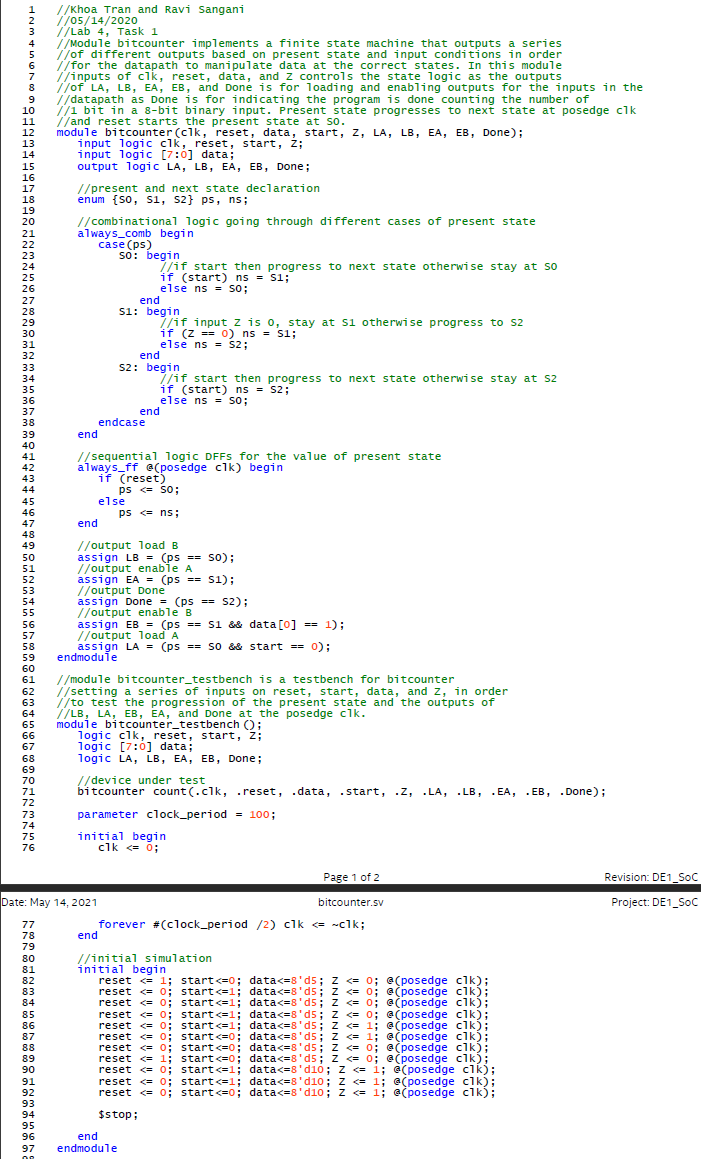
Figure 14: Waveform simulation generated by ram32x8

**Final Product**

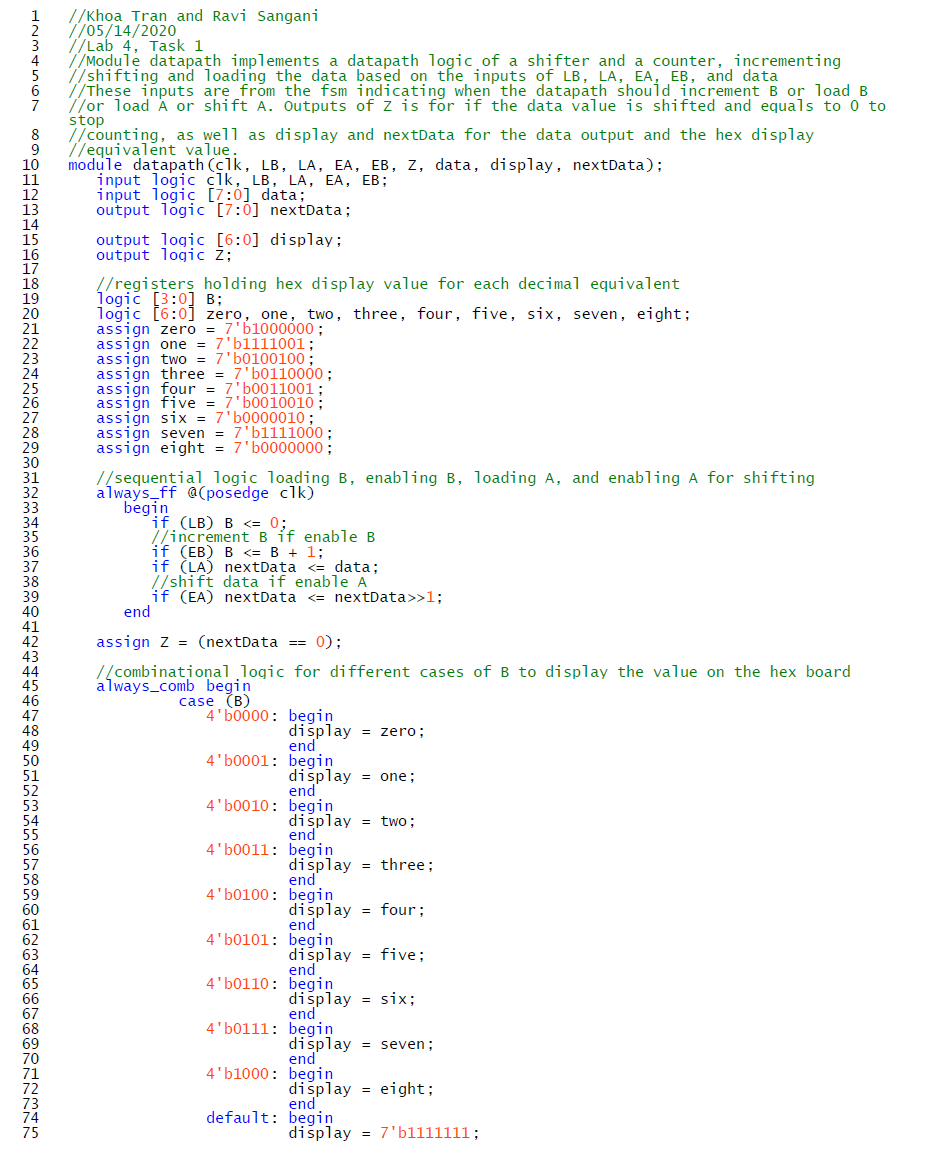
This project had two separate tasks, however, with one overarching goal of implementing a HLSM using a controller and datapath to control the conditions and when certain operations in the datapath should proceed. In the first task, we designed a bitcounter that works like the conditions given in the lab spec. The bitcounter uses a controller to output LA, LB, EA, and EB and the datapath takes these inputs and either shifts A or count B. After the first task, the goal of the second task was to develop a binary search on a sorted 32x8 ram. We were able to achieve this goal with no issues as our controller was able to progress through different states and output Elower or Eupper in order to change the bounds and condense the search to a smaller range. This continues on as we were able to successfully simulate and operate the project on the FPGA perfectly. This project was extremely helpful for us as we were able to independently develop a HLSM using controller and datapath, opening up potential to develop many more different algorithms on hardware.

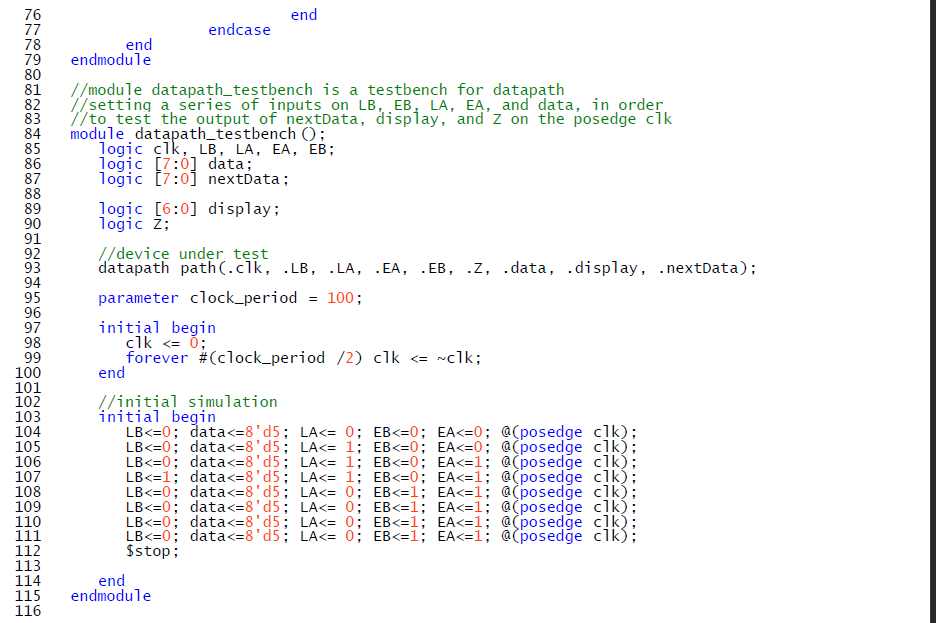
**Appendix: SystemVerilog Code**

1. **bitcounter.sv (task 1)**

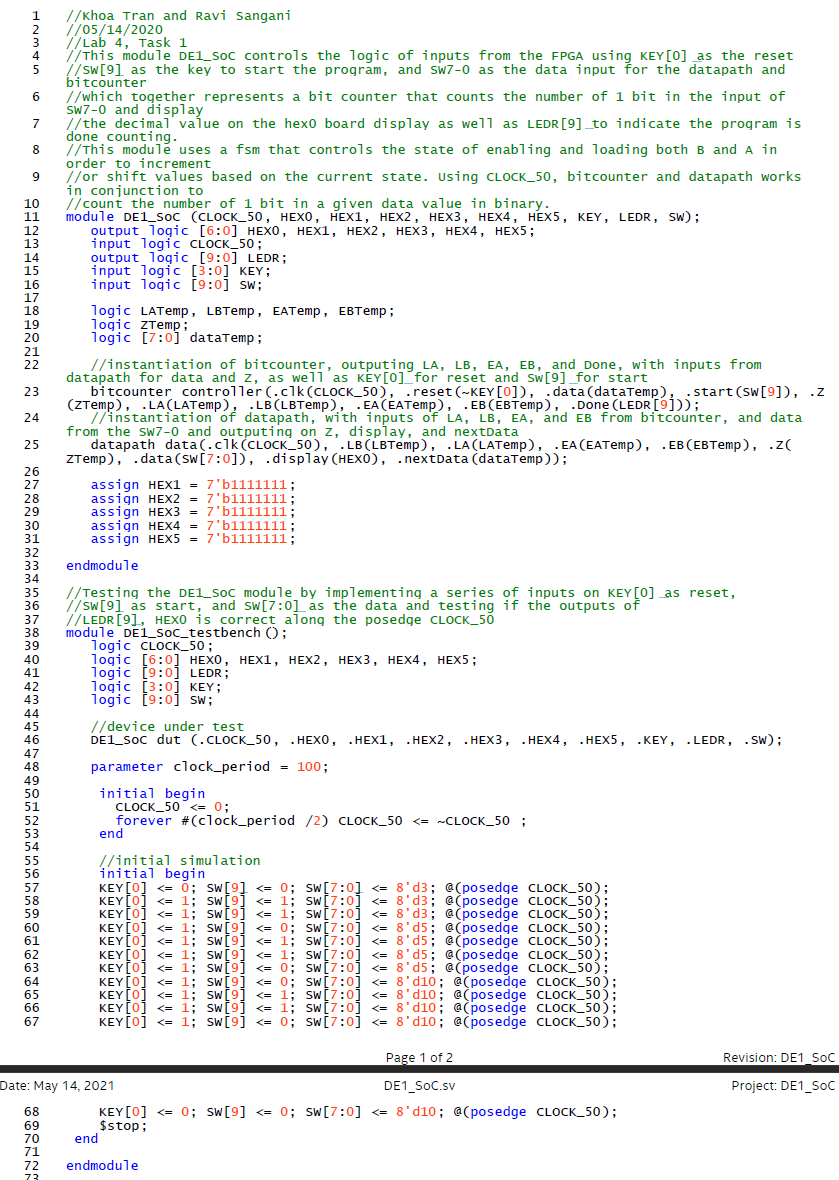
****

1. **datapath.sv (task 1)**

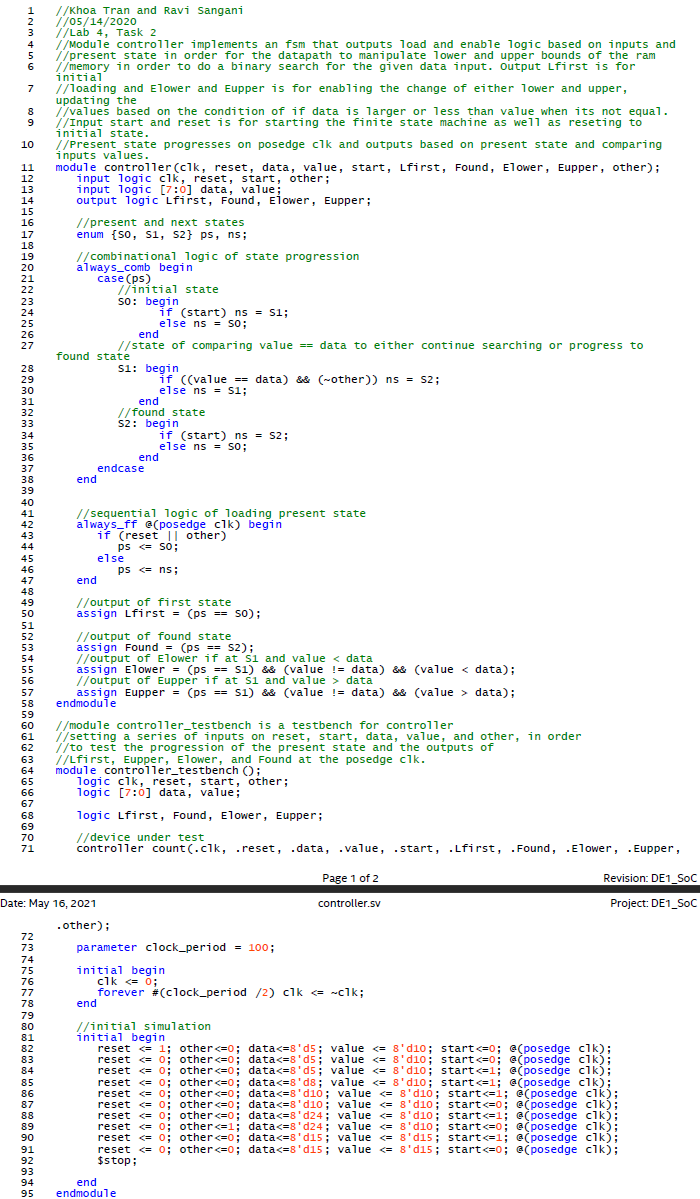


****

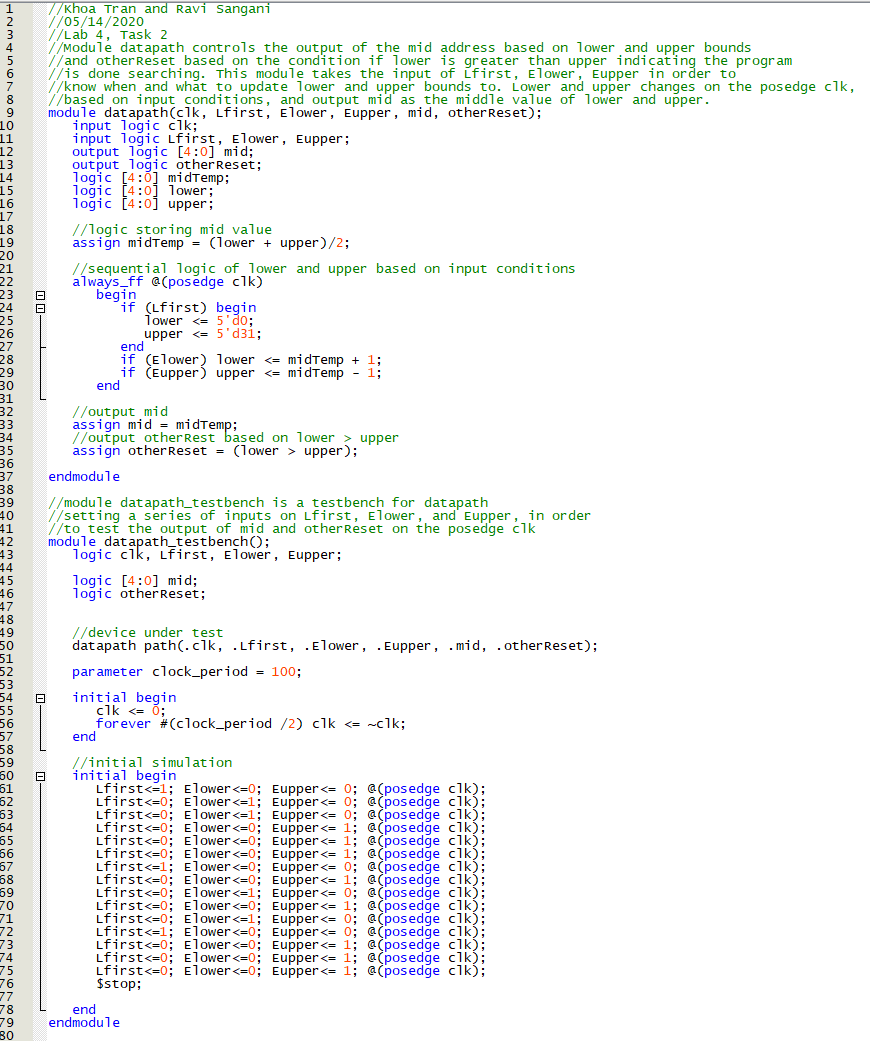
1. **DE1\_SoC.sv (task 1)**



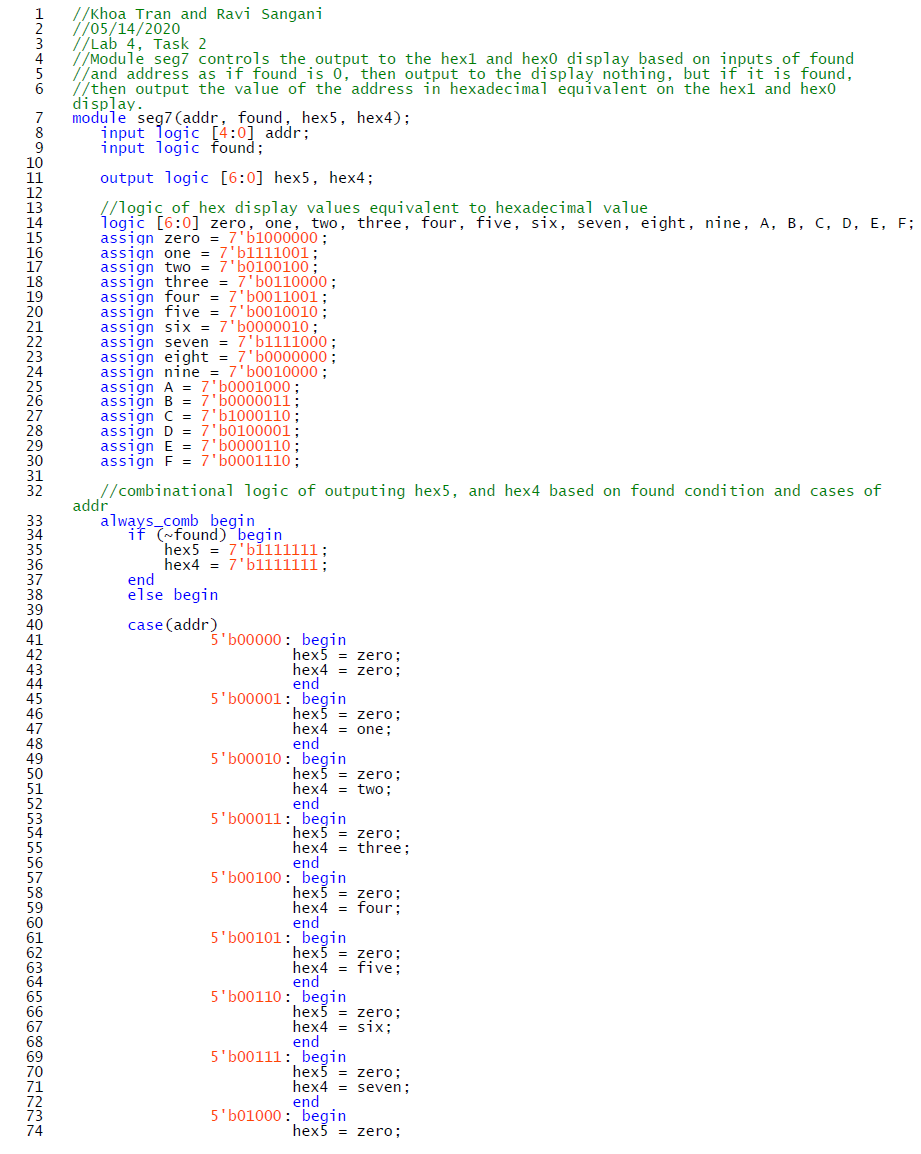
1. **controller.sv (task 2)**

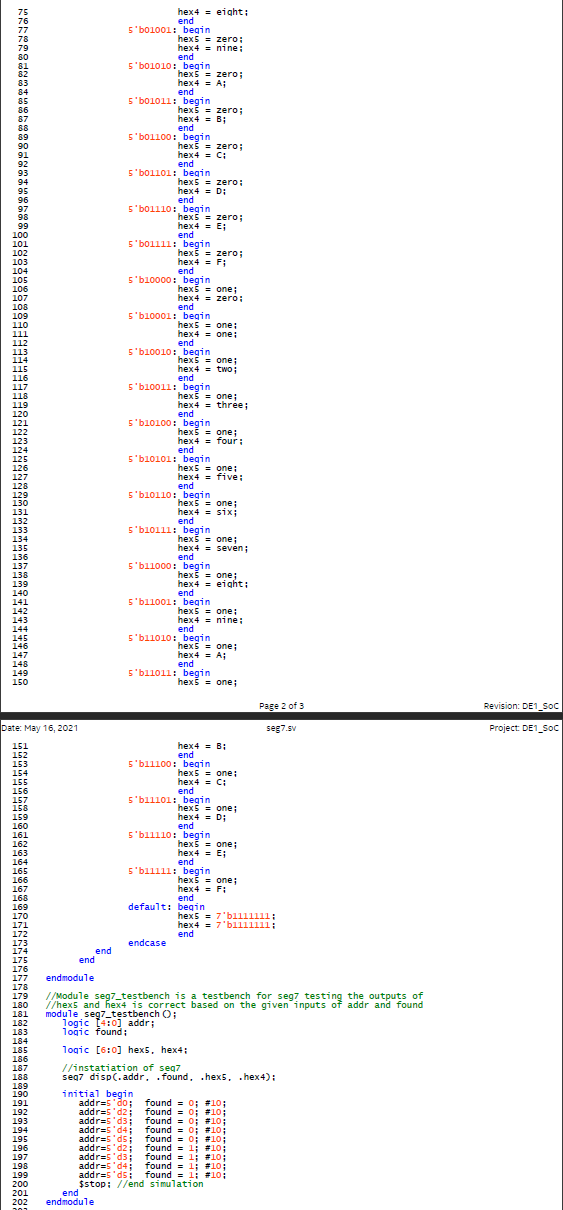
****

1. **datapath.sv (task 2)**

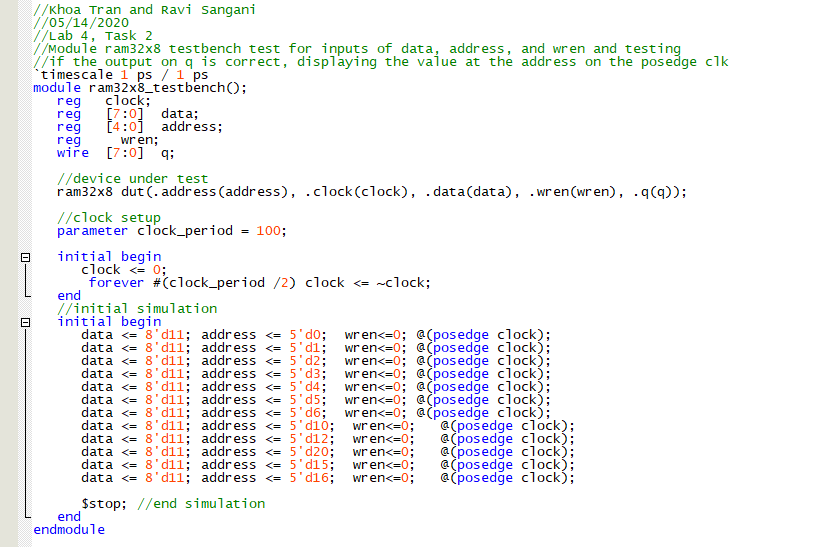
****

1. **seg7.sv (task 2)**

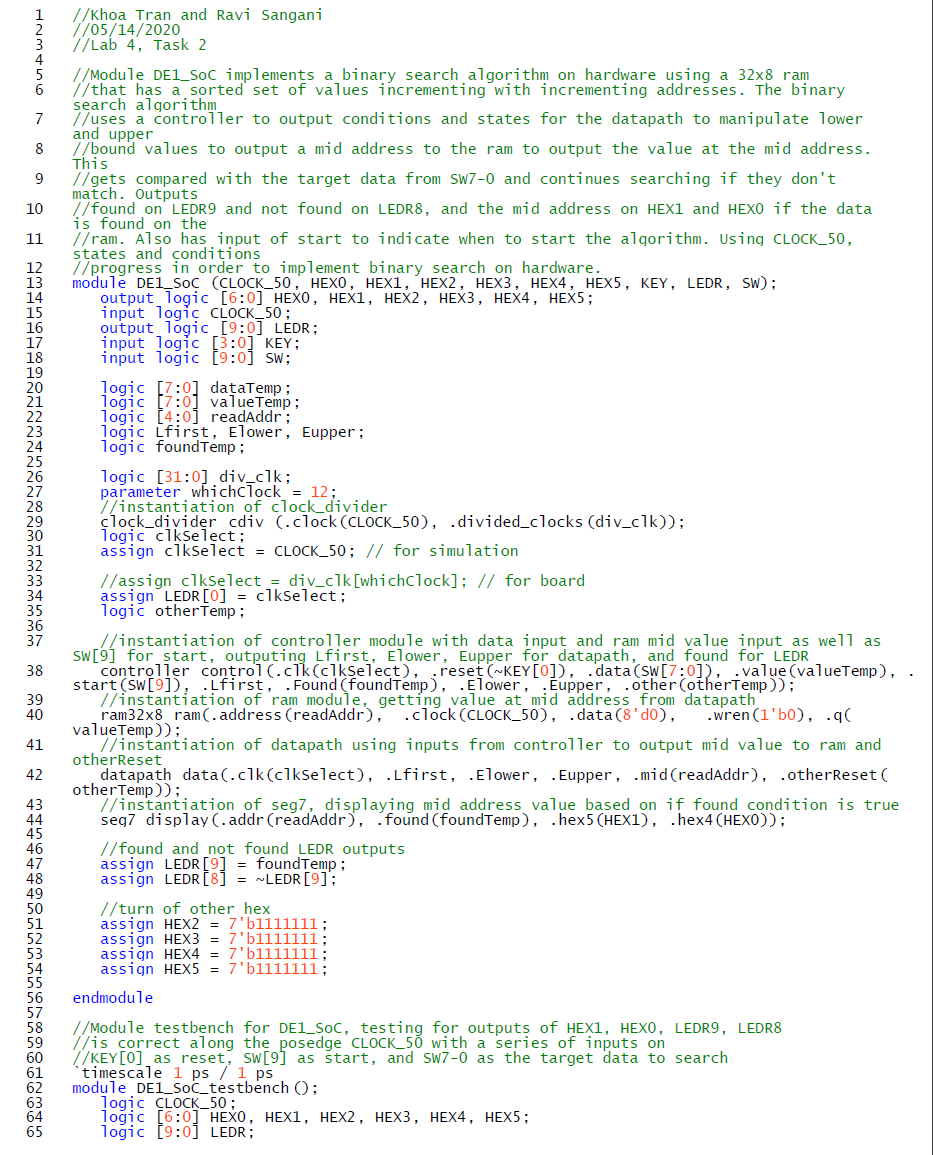
****

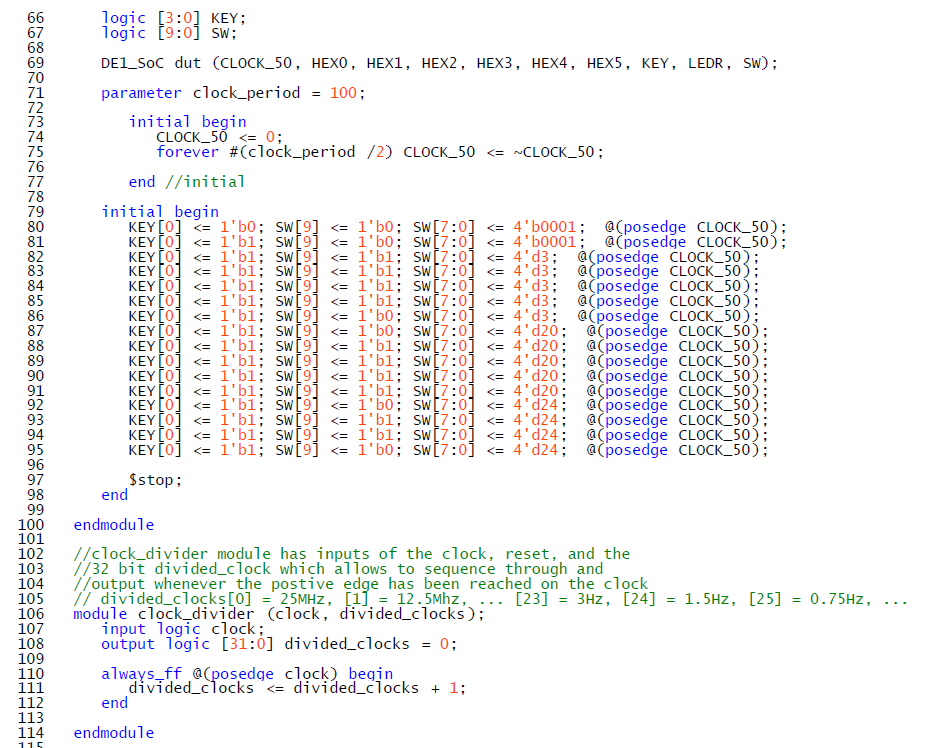
****

1. **ram32x8\_testbench (task 2)**

****

1. **DE1\_SoC.sv (task 2)**

****

****