ECE 385

Spring 2023

Lab 5

Experiment #5

Aleena Majeed & Ali Chaudry
aleenam2 & achau9
17:50 - 17:55

1. **Introduction**
    a. **Summarize the basic function of the SLC-3 processor**
2. **Written Description and Diagrams of SLC-3**
    a. **Summary of Operation**
    a. **Describe in words how the SLC-3 performs its functions. You should describe the Fetch-Decode-Execute cycle as well as the various instructions the processor can perform.**
    a. **block Diagram of slc3.sv**
    a. **This diagram should represent the placement of all your modules in the slc3.sv. Please only include the slc3.sv diagram and not the RTL view of every module (this can go into the individual module descriptions).**
    a. **Written Description of all .sv modules**
        . **A guide on how to do this was shown in the lab 2.2 report outline**
    a. **Description of the operation of the ISDU**
        . **Named ISDU.sv, this is the control unit for the SLC-3. Describe in words how the ISDU controls the various components of the SLC-3 based on the current instruction.**
        . **If you prefer to , you can lump this section into the module description section under ISDU.sv**
    a. **State Diagram of ISDU**
2. **Simulations of SLC-3 Instructions**
    a. **Simulate the completion of all 6 test programs, I/O Test 1, I/O Test 2, Self-Modifying doe, XOR, Multiplier and Sort**
    a. **Annotations for the above simulations**
2. **Post-Lab Questions**
    a. **Fill out the Design Resources and Statistics table from Post-Lab question one**
    a. **Answer all the post lab questions**
2. **Conclusion**
    a. **Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it**
    a. **Was there anything ambiguous, incorrect or unnecessarily difficult in the lab manual or given materials which can be improved for next semester.**

INTRODUCTION

WRITTEN DESCRIPTION AND DIAGRAMS OF SLC-3

The SLC-3 Processor consists of three main components, which include the CPU, the memory that contains the data and instructions. And the I/O interface that can communicate with external devices. The computer first fetches an instruction from memory, decodes the instruction, then executes the instruction and repeats the cycle starting back at the fetch operation. The central processing unit utilizes the PC register, the Instruction Register, Memory Address, and Memory Data register. All registers and logical elements in the SLC-3 processor are 16 bits wide. The CPU also contains an Arithmetic logic Unit and the Instruction Sequencer provides proper control signals to the other logical elements in the processor. The processor uses a moore state machine to control the operations in the processor. The processor performs its tasks based on the opcodes from the instruction register which are stored in IR[15:12]. The LC3 processor also contains 8 general-purpose registers that help in the process of executing the assembly program's logical, arithmetic, conditional branching, and subroutine call operations. The processor also uses the program counter to update the count of what instruction it is on.

2. Written Description and Diagrams of SLC-3

a. Summary of Operation

The LC3 instructions consist of ADD, ADDi, AND, NOT, BR, JMP, JSR, LDR, STR, and PAUSE. The instruction register helps specify the operation to perform on the data stored in registers. Each operation takes multiple cycles and the Instruction Decoder needs to provide the appropriate signals at each cycle which is controlled by the control unit in the SLC-3 processor. During a reset, the Instruction Sequencer should reset to the beginning halted state awaiting the user to press the execute button. The program counter should also be reset to zero.

b. Describe in words how the SLC-3 performs its functions. You should describe the Fetch-Decode-Execute cycle as well as the various instructions the processor can perform.

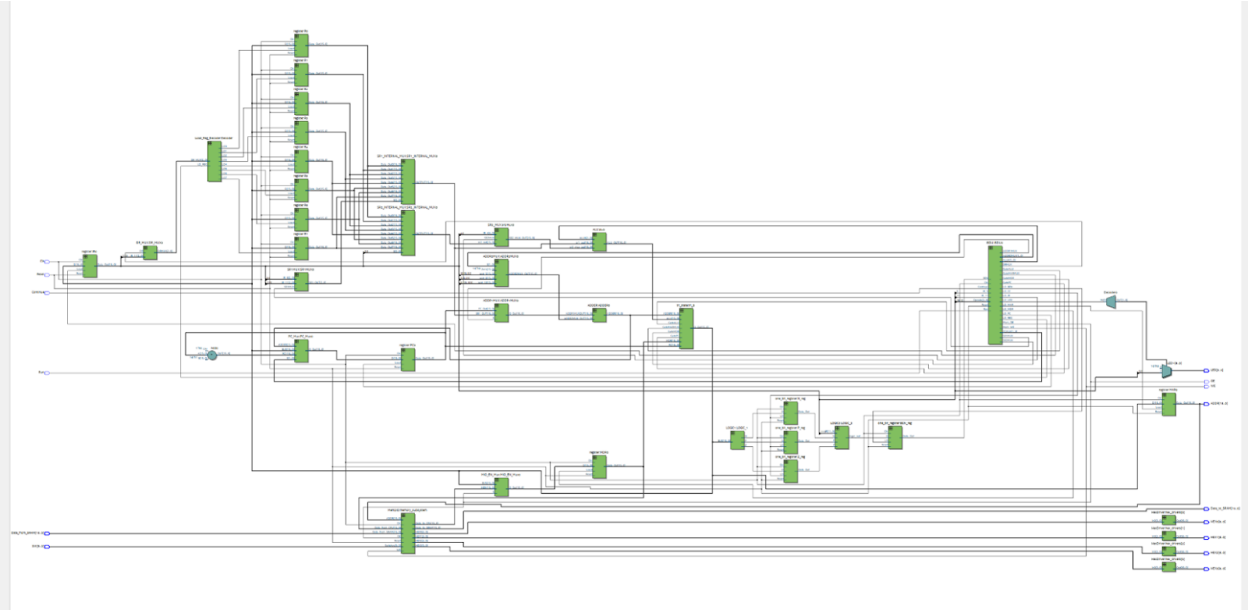
The SLC-3 processor divides its operation into 3 main processes: fetch, decode, and execute. In the fetch stage, The IR, MAR, MDR, and PC registers are central to the function of this stage. First the Memory Address Register stores the Program Counter Register Value, MAR ← PC. Next the the Memory of Memory Address Register is stored into the Memory Data Register, MDR←M(MAR). After this the Memory Data Register's contents are stored into the Instruction Register, IR ← MDR. Finally the Program Counter is updated, PC ← (PC+1). In the decode state the Instruction Register is fed into the Instruction Sequencer/Decoder. The Decoder sends the control signals on what registers to select as source and destination registers, and what operations to perform and what gates should be writing to the BUS. In the execute state, the operation requested is performed based on the signals from the Instruction Sequencer and the result is written into the destination register or memory.

The SLC-3 Processor is able to perform the ADD, ADDi, AND, NOT, BR, JMP, JSR, LDR, STR, and PAUSE functions. Each function has a specific purpose. The ADD function adds the contents of SR1(source register 1) and SR2(source register 2) and stores the result into the DR. At the end of the operation it sets the CC values which is the status register value that indicates if the answer in the destination register is a positive number, negative number, or zero. The ADDi instruction stands for ADD immediate which adds the contents of the SR (Source register) to the signed extended value imm5, and stores the result to the destination register. It also sets the CC value. The AND operation ANDs the content of SR1 and SR2, and stores the result into DR and sets the CC value as well. The ANDi function works like the ADDi function, as it ANDSs the contents of the source register with the signed extended value of imm5, and stores the result into the destination register while setting the CC value. The NOT function negates the values stored in the Source register and stores the result in the destination register while setting the CC value. The BR function is a branch function that branches if any of the condition code matches the condition stored in the Status register(set CC), otherwise the program continues its execution. The program can also unconditionally branch if the NZP value is set to "111". The program determines where to branch to by adding the sign-extended PCoffset9 to the PC. The JMP function jumps to a certain address in the program by copying a memory address from the Base Register to the Program Counter Register. The LDR function loads using a Register offset addressing, It loads the Destination Register with memory contents pointed to by (BaseR + SEXT(offset6)). It also sets the status register. The STR operation stores using Register offset

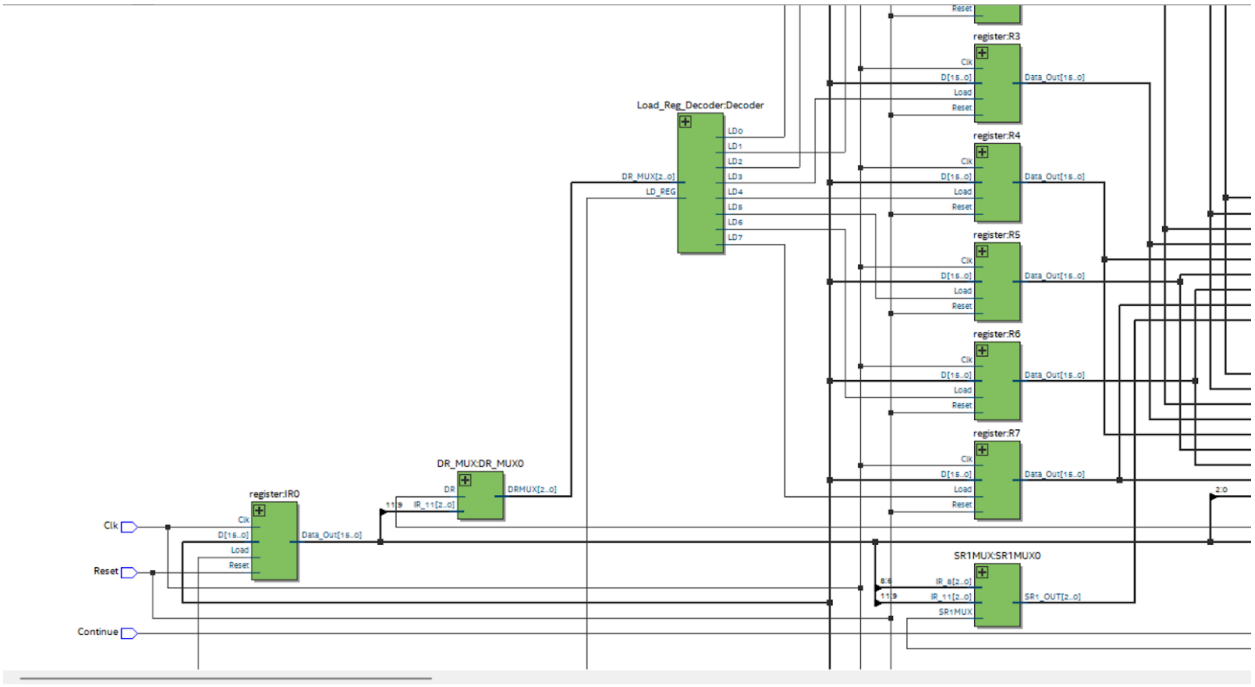
addressing. It stores the contents of the Source Register at the memory location pointed to by (BaseR + SEXT(offset6)). The PAUSE function is responsible for pausing the execution until the continuation is pressed by the user. The execution should only occur if the continue button is pressed during the current pause instruction . During the PAUSE states, the ledVect12 is displayed on the FPGA LEDs.

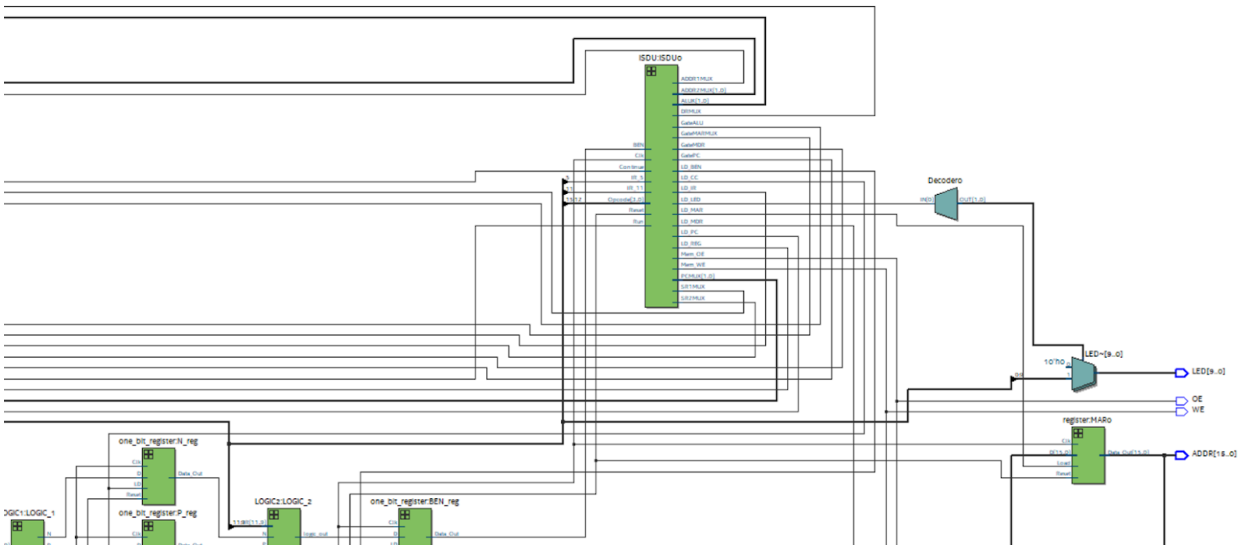
c. block Diagram of slc3.sv

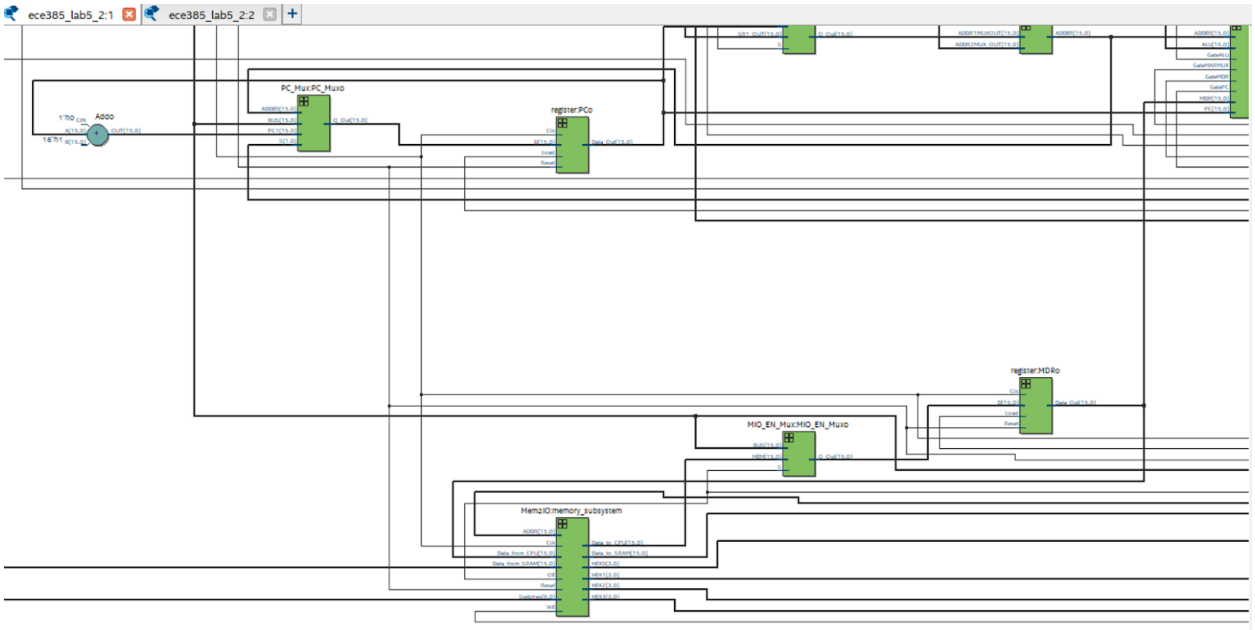
The first figure shown below is a picture of the entire block diagram of the slc3.sv file. Subsequent photos are zoomed in screenshots to show inputs and outputs of the logical elements:

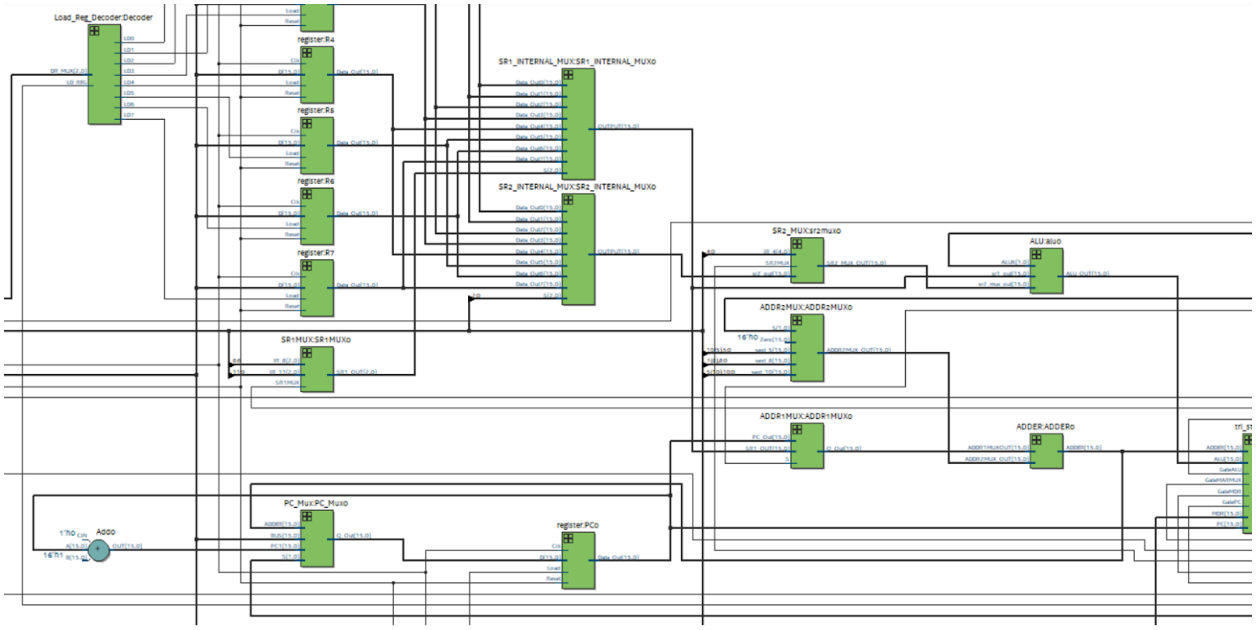


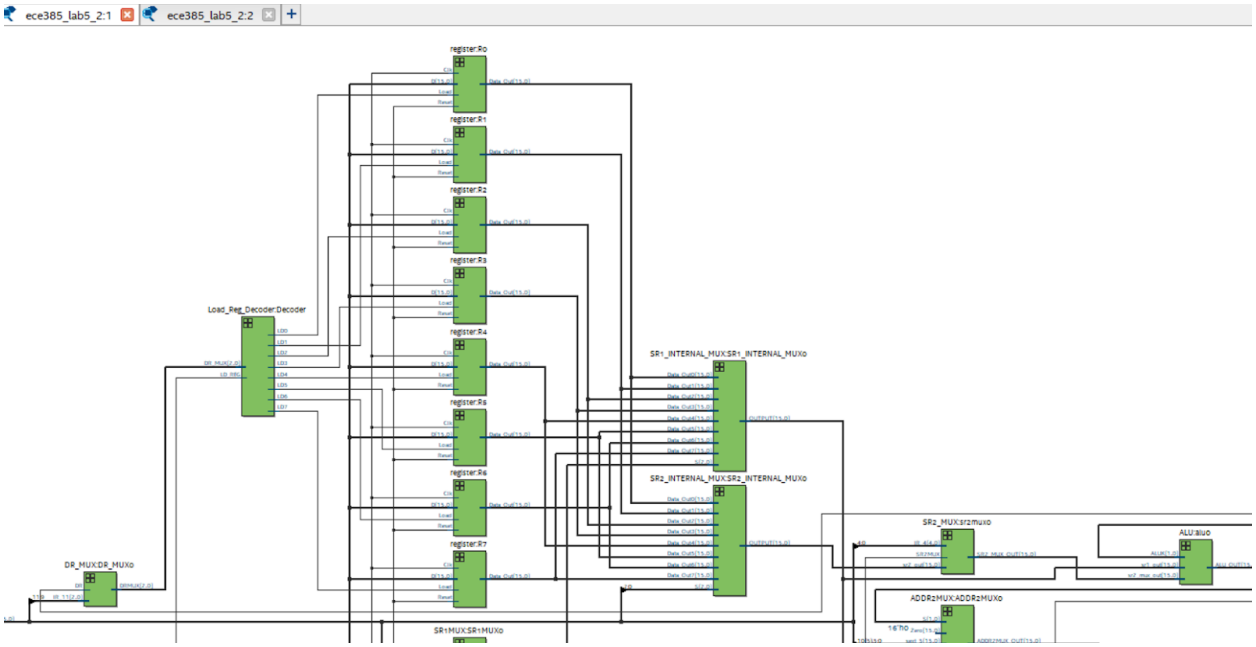
Above is the full RTL block diagram generated from Quaruts - below are zoomed in photos of the different sections of the full block diagram for easier viewing

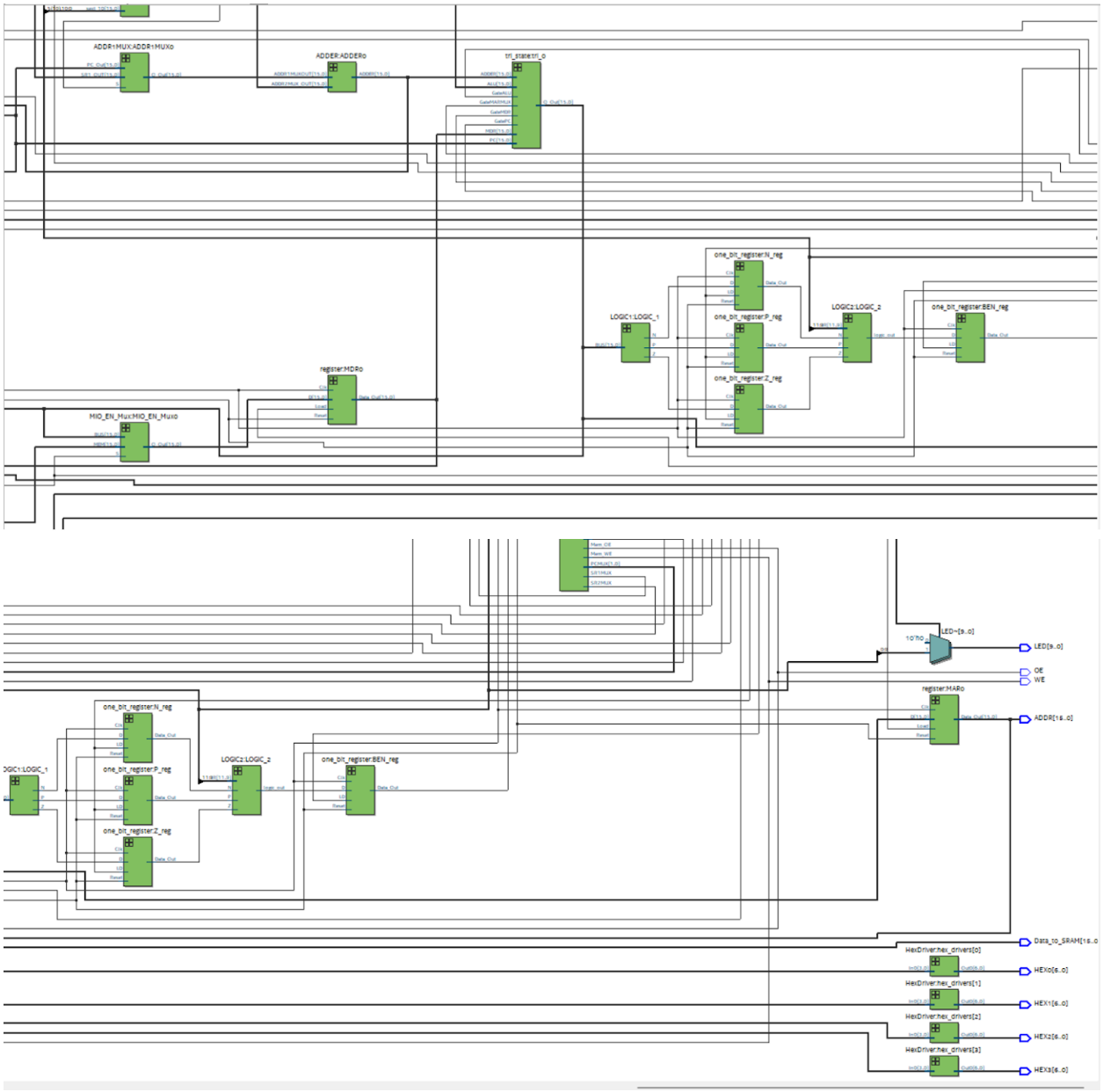



ISDU:ISDUo
Decodero
LED[9..0]
LED[9..0]
OE
WE
register:MARo
ADDR[15..0]
one_bit_register:N_reg
one_bit_register:P_reg
LOGIC2:LOGIC_2
one_bit_register:BEN_reg


ece385_lab5_2:1
ece385_lab5_2:2
Addo
PC_Mux:PC_Muxo
register:PCo
register:MDRo
MIO_EN_Mux:MIO_EN_Muxo
Mem2IO:memory_subsystem


Load_Reg_Decoder:Decoder
register:R4
register:R5
register:R6
register:R7
SR1_INTERNAL_MUX:SR1_INTERNAL_MUXo
SR2_INTERNAL_MUX:SR2_INTERNAL_MUXo
SR2_MUX:sr2muxo
ALU:aluo
ADDR2MUX:ADDR2MUXo
ADDR1MUX:ADDR1MUXo
ADDER:ADDERo
SR1MUX:SR1MUXo
PC_Mux:PC_Muxo
Addo
register:PCo


ece385_lab5_2:1
ece385_lab5_2:2
register:Ro
register:R1
register:R2
register:R3
register:R4
register:R5
register:R6
register:R7
Load_Reg_Decoder:Decoder
SR1_INTERNAL_MUX:SR1_INTERNAL_MUXo
SR2_INTERNAL_MUX:SR2_INTERNAL_MUXo
SR2_MUX:sr2muxo
ALU:aluo
DR_MUX:DR_MUXo
ADDR2MUX:ADDR2MUXo
SR1MUX:SR1MUXo


ADDR1MUX:ADDR1MUXo
ADDER:ADDERo
tri_state:tri_o
one_bit_register:N_reg
one_bit_register:P_reg
one_bit_register:Z_reg
LOGIC1:LOGIC_1
LOGIC2:LOGIC_2
one_bit_register:BEN_reg
register:MDRo
MIO_EN_Mux:MIO_EN_Muxo
LED[9..0]
register:MARo
ADDR[19..0]
OE
WE
Data_to_SRAM[15..0]
HexDriver:hex_drivers[0]
HEX0[6..0]
HexDriver:hex_drivers[1]
HEX1[6..0]
HexDriver:hex_drivers[2]
HEX2[6..0]
HexDriver:hex_drivers[3]
HEX3[6..0]

d. Written Description of all .sv modules

- **REG_FILE.sv :**
  - **module DR_MUX()**
    - **input logic DR,**
    - **input logic [2:0] IR_11,**
    - **output logic [2:0] DRMUX**

FUNCTION: The DRMUX module was made to create the Destination Register Mux which is used to select which of the 8 registers will be the Destination register and has an output that is used as an input to the decoder that chooses which register to load/write into based on which register is selected using the LD.REG control signal. The mux chooses either "111" or IR[11:9] and uses the DR select signal coming from the ISDU control unit.

  - **module SR1MUX()**
    - **input logic SR1MUX**
    - **input logic [2:0] IR_11, IR_8,**
    - **output logic SR1_OUT**

FUNCTION: The SR1MUX module was created to help create the SR1MUX which helps choose the correct source register or Base Register based on its inputs, IR[11:9] or IR[8:6], and its select which is SR1 coming from the control unit. The output of this mux acts as a 3-bit wide select that chooses which register to choose the data out from.

  - **module Load_Reg_Decoder()**
    - **input logic [2:0] DRMUX**
    - **input logic LD_REG**
    - **output logic LD0, LD1, LD2, LD3, LD4, LD5, LD6, LD7**

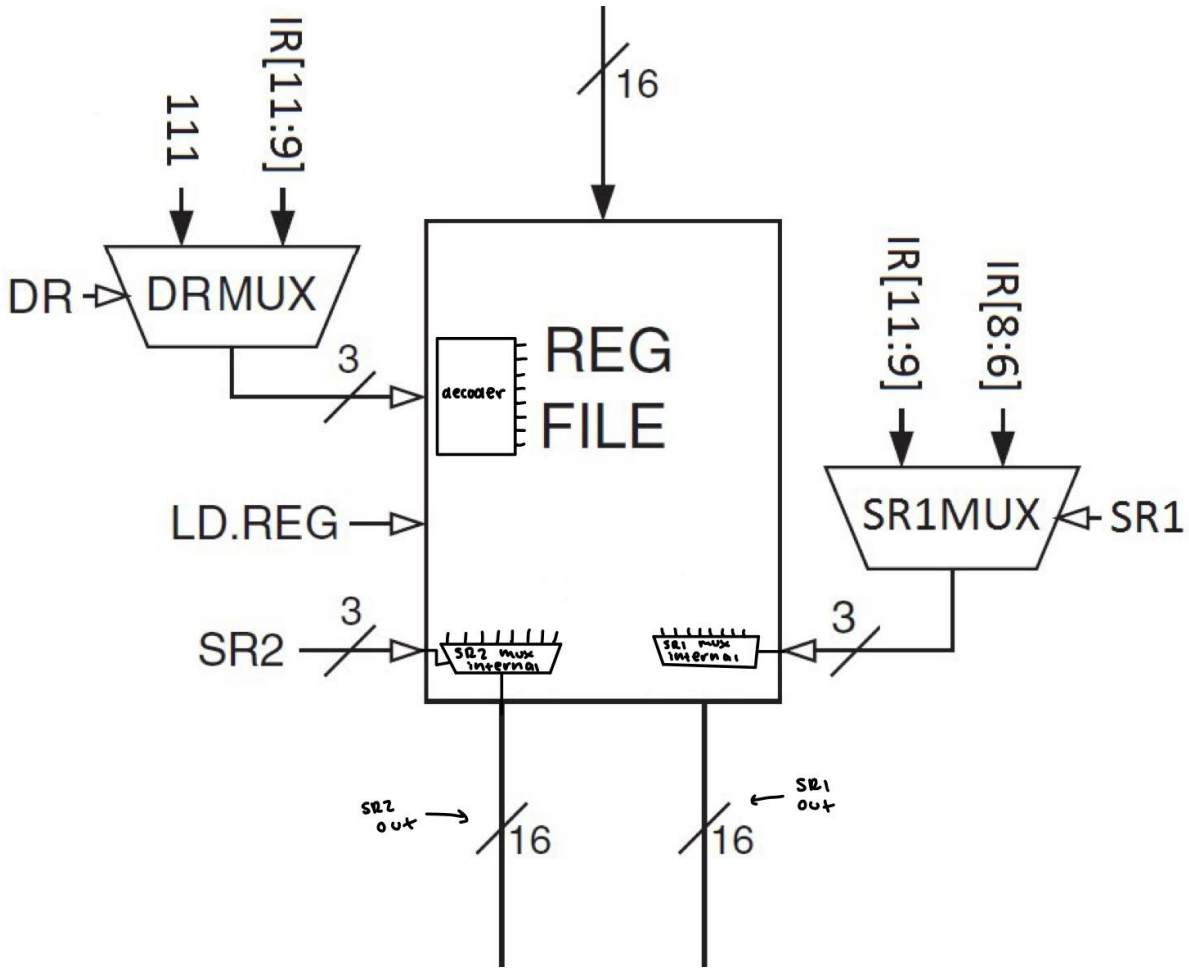
FUNCTION: This module acts as a decoder that has 3-bit wide inputs coming from the DRMUX module and uses LD.REG signal that comes from the control unit. This decoder picks which register's LOAD to turn high so that the right respective register is loaded.

  - **module SR1_INTERNAL_MUX()**
    - **input logic [2:0] S,**
    - **input logic [15:0] Data_Out0, Data_Out1, Data_Out2, Data_Out3, Data_Out4, Data_Out5, Data_Out6, Data_Out7,**
    - **output logic [15:0] OUTPUT**

FUNCTION: This mux is an internal mux in the reg file not shown on the given datapath. The selects of this mux is the output of the SR1MUX. The selects choose which register to output the data out from for source register one.

  - **module SR2_INTERNAL_MUX()**
    - **input logic [2:0] S,**
    - **input logic [15:0] Data_Out0, Data_Out1, Data_Out2, Data_Out3, Data_Out4, Data_Out5, Data_Out6, Data_Out7,**
    - **output logic [15:0] OUTPUT**

FUNCTION: This mux is an internal mux in the reg file not shown on the given datapath. The selector of this mux is the SR2 select, which comes from IR[2:0]. The select chooses which register to output the data out from for source register 2.



Update REG FILE from the LC3 data path - this version includes the internal decoder as well as the two internal muxes: SR2 internal mux and SR1 internal mux

- **control_unit_input.sv :**
  - **module LOGIC1():**
    - **input logic [15:0] BUS,**
    - **output logic N, Z, P**

FUNCTION: This module reads in 16-bit wide data and checks whether it is positive, negative, or zero and then outputs a 1 P, N, or Z, depending on whether the number is positive, negative or zero, respectively. If the number is not positive then P is set to 0, if it is not negative then N is set to 0 and so on.

  - **module one_bit_register()**
    - **input logic Clk, Reset, LD,**
    - **input logic D**
    - **output logic Data_Out**

FUNCTION: This module is used as a template for the 3 one-bit registers, N, Z, and P. These registers take in data from the previous LOGIC1 and are Loaded based on the LD.CC signal sent in from the control unit. They output their data into another logic block. This module is also used to make the BEN register which takes in the data from the second LOGIC2 module and inputs its data out into the control unit. The BEN register is only loaded if the LD.BEN signal coming from the control unit is high.

  - **module LOGIC2()**
    - **input logic [11:9] IR,**
    - **input logic N, Z, P,**
    - **output logic logic_out**

FUNCTION: This module performs the logical operation shown in state 32 in the SLC3 state machine diagram. It only performs the operation and loads the data out into the BEN register depending on the IR[11:9] bits.

- **adder_mux.sv :**
  - **module ADDR2MUX()**
    - **input logic [1:0] S,**
    - **input logic [15:0] Zero,**
    - **input logic [15:0] sext_10,**
    - **input logic [15:0] sext_8,**
    - **input logic [15:0] sext_5,**
    - **output logic [15:0] ADDR2MUX_OUT**

FUNCTION: This module makes the ADDR2MUX which is a component that helps perform an addition operation. It takes 4 inputs three of which are sign extended 16-bit wide numbers from IR[10:0], [8:0], IR[5:0], and input of 0 which is 16 bits wide. The select signal comes from the control unit.

  - **module ADDR1MUX()**
    - **input logic S,**
    - **input logic [15:0] PC_Out,**
    - **input logic [15:0] SR1_Out,**
    - **output logic [15:0] Q_Out**

FUNCTION: This module makes the ADDR1MUX which is a component that helps perform an additional operation. It takes two inputs: a 16-bit immediate value from the PC register and the contents of a register from source register 1. The adder1 component selects one of these two values and produces a 16-bit output that is used as an input to an adder.

  - **module ADDER()**
    - **input logic [15:0] ADDER2MUX_OUT, ADDR1MUXOUT,**
    - **output logic [15:0] ADDER**

FUNCTION: This is an ADDER module which is responsible for adding the output of the ADDR1 and ADDR2 Muxes and outputs the value to either the BUS or the PCMUX.

- **alu.sv :**
  - **module ALU()**
    - **input logic [1:0] ALUK,**
    - **input logic [15:0] sr1_out, sr2_mux_out,**
    - **output logic [15:0] ALU_OUT,**

FUNCTION: This ALU takes in either an immediate sign extended value from the Instruction Register, or a value from the Source Register 2, and performs an operation with the Source Register 1 value, and outputs that value to the BUS. The operation depends on the select forming from the ALUK signal from the control unit. The operations entailed, include, AND, ADD, NOT, etc.

  - **module SR2_MUX()**
    - **input logic SR2MUX,**
    - **input logic [15:0] sr2_out,**
    - **input logic [4:0] IR_4,**
    - **output logic [15:0] sext_4,**
    - **output logic [15:0] SR2_MUX_OUT**

FUNCTION: This mux selects either a value from one of the source registers, which comes from SR2 OUT, or it selects sign extended immediate value coming from the Instriuction register. It chooses one of thes values based on the SR2 select which comes from the control unit.

- **registers_lab5_2.sv :**
    - **module register()**
        - **input logic Clk, Reset, Load,**
        - **input logic [15:0] D,**
        - **output logic [15:0] Data_Out,**

FUNCTION: This module was used as a template for all 16-bit registers, such as the PC, MDR, MAR, IR, and R0-R7 Registers. They all have a data in, data out, clock, reset, and Load signal that is controlled either by the control unit or other logical elements in the datapath.

    - **module PC_Mux()**
        - **input logic [1:0] S,**
        - **input logic [15:0] PC1,**
        - **input logic [15:0] BUS,**
        - **input logic [15:0] ADDER,**
        - **output logic [15:0] Q_Out**

FUNCTION: This module creates the PCMUX which selects the input into the PC Register. It either chooses PC+1, A value from the BUS, or the output of the ADDER that adds the ADDR1MUX and ADDR2MUX output.

    - **module MIO_EN_Mux()**
        - **input logic S,**
        - **input logic [15:0] BUS**
        - **input logic [15:0] MEM,**
        - **output logic [15:0] Q_Out**

FUNCTION: This module creates a MIO Mux which chooses the input into the MDR register, The inputs to this mux are the Data_to_CPU and a value from the BUS. It has an MIO.EN for its select signal which comes from the control unit.

    - **module tri_state()**
        - **input logic GateMDR, GateALU, GatePC, GateMARMUX,**
        - **input logic [15:0] ADDER, MDR, PC, ALU,**
        - **output logic [15:0] Q_Out**

FUNCTION: This is a mux which acts as the BUS for the LC3 Data path as it chooses which GATE to set high that one of the gates can read into the BUS at a time.

- **hex_drivers5.sv :**
  - **module HexDriver()**
    - **input logic [3:0] In0,**
    - **output logic [6:0] Out0**

FUNCTION: This module simply contains if statements to apply the correct input/output logic so that the hex drivers of the FPGA get the correct values depending on which LED values are high.

- **Lab5provided_sp2023/Instantiateram.sv :**
  - **module Instantiateram()**
    - **input Clk, Reset,**
    - **output logic [15:0] ADDR,**
    - **output logic wren,**
    - **output logic [15:0] data**

FUNCTION: This module instantiates the on chip memory for the FPGA. This module contains a read enable, write enable, a 10-bit address bus, and q, which represents a 16-bit data bus.

- **Lab5provided_sp2023/ISDU.sv :**
  - **module ISDU()**
    - **input logic Clk, Reset, Run, Continue, IR_5, IR_11, BEN**
    - **input logic [3:0] Opcode,**
    - **output logic LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED,**
    - **output logic GatePC, GateALU, GateMDR, GateMARMUX,**
    - **output logic [1:0], PCMUX, DRMMUX, SR1MUX, SR2MUX, ADDR1MUX**
    - **output logic [1:0] ADDR2MUX, ALUK,**
    - **output logic Mem_OE, Mem_WE**

FUNCTION: This module is the control unit which implements the state machine for the LC3 processor. There are several inputs and output signals into the control unit which controls the flow of the data in the LC3 Processor The control unit decides when and which registers to load, what operations to perform on data, the selects of muxes, etc. It implements states for memory operations and arithmetic operations and ties all logic elements together. It also uses opcodes from the IR to decide what state to implement. It includes 2 pause states in which the LED signals are set high. Additionally, at each state where a memory read or write takes place, there are 4 wait states implemented in order to allow the computer to take at least 4 cycles to receive the right updated data memory to or from.

- **Lab5provided_sp2023/Mem2IO.sv :**
  - **module Mem2IO()**
    - **input logic Clk, Reset,**
    - **input logic [15:0] ADDR,**
    - **input logic OE, WE,**
    - **input logic [9:0] Switches,**
    - **input logic [15:0] Data_from_CPU, Data_from_SRAM**
    - **output logic [15:0] Data_to_CPU, Data_to_SRAM**
    - **output logic [3:0] HEX0, HEX1, HEX2, HEX3**

FUNCTION: This component controls the input/output operations of the physical I/O devices on the DE10-Lite, specifically the switches and 7-segment displays. The Board Hex display and Board switches are located at the same memory address, which is acceptable as the switches only receive input, while the hex displays only provide output.

- **Lab5provided_sp2023/memory_contents.sv :**
  - **module memory_parser;**

FUNCTION: This module contains the memory contents in the test memory. It can also be used to refer to when testing the model simulation since it shows what states the processor should be going through for each given test.

- **Lab5provided_sp2023/slc3.sv :**
  - **module slc3()**
    - **input logic Clk, Reset, Run, Continue**
    - **input logic [9:0] SW,**
    - **output logic [9:0] LED,**
    - **input logic [15:0] Data_from_SRAM,**
    - **output logic OE, WE**
    - **output logic [6:0] HEX0, HEX1, HEX2, HEX3,**
    - **output logic [15:0] ADDR**
    - **output logic [15:0] Data_to_SRAM**

FUNCTION: This module was used for instantiating all of our modules. It was used as a data path module where all logic elements were instantiated such as the control unit, register file, MAR, MDR, IR, PC, R0-R7 registers, etc. This is the module that connected all modules.

- **Lab5provided_sp2023/SLC3_2.sv :**

FUNCTION: This module includes useful constants, opcode aliases, register aliases, branch condition aliases, etc. This is to be included into the project and should be referenced in files to reference the functions and contents included in the module.

- **Lab5provided_sp2023/slc3_sramtop.sv :**
  - **Module slc3_sramtop()**
    - **input logic [9:0] SW,**
    - **input logic Clk, Run, Continue,**
    - **output logic [9:0] LED,**
    - **output logic [6:0] HEX0, HEX1, HEX2, HEX3**

FUNCTION: This module instantiates the sync module, instantiateram module and slc3 module, along with the ram module. It should be selected at the top level module when testing the processor's function on the FPGA.

- **Lab5provided_sp2023/slc3_testtop.sv :**
  - **Module slc3_testtop()**
    - **input logic [9:0] SW,**
    - **input logic Clk, Run, Continue,**
    - **output logic [9:0] LED,**
    - **output logic [6:0] HEX0, HEX1, HEX2, HEX3**

FUNCTION: This module instantiates the sync module and slc3 module, along with the test_memory module. It should be selected at the top level module when testing the processor's function on the model simulation.

- **Lab5provided_sp2023/synchronizer.sv :** The synchronizer unit was provided code in previous labs that we were just able to copy and paste this module and reuse it as a way to eliminate any errors that could come from multiple signals sharing the same clock.

    - **module sync()**
        - **input logic Clk, d,**
        - **output logic q**

FUNCTION: The synchronizer unit was provided code. It is used to eliminate any errors that could come from multiple signals sharing the same clock.

    - **module sync_r0()**
        - **input logic Clk, Reset, d,**
        - **output logic q**

FUNCTION: The synchronizer unit was provided code. It is used to eliminate any errors that could come from multiple signals sharing the same clock.

    - **module sync_r1()**
        - **input logic Clk, Reset, d,**
        - **output logic q**

FUNCTION: The synchronizer unit was provided code. It is used to eliminate any errors that could come from multiple signals sharing the same clock.

- **Lab5provided_sp2023/test_memory.sv :**
  - **module test_memory()**
    - **input Reset,**
    - **input Clk**
    - **input [15:0] data,**
    - **input [9:0] address,**
    - **input rden,**
    - **input wren,**
    - **output logic [15:0] readout**

FUNCTION: This memory module exhibits comparable characteristics to the SRAM IC found on the DE2 board, and is intended solely for simulation purposes. During simulation, this memory is ensured to perform at least as proficient as the physical memory (which may demand more meticulous handling than this test memory). To utilize this module, it is necessary to create a distinct top-level entity for simulation that establishes a connection between this memory module and your computer.

- **5_2_Testbench.sv :**
  - **module testbench()**

FUNCTION: The testbench is used to test the function of the LC3 processor on the model simulation. The testbench helps with debugging as well as allows the user to see if all the states that are tested are being used and if the correct signals are being output.

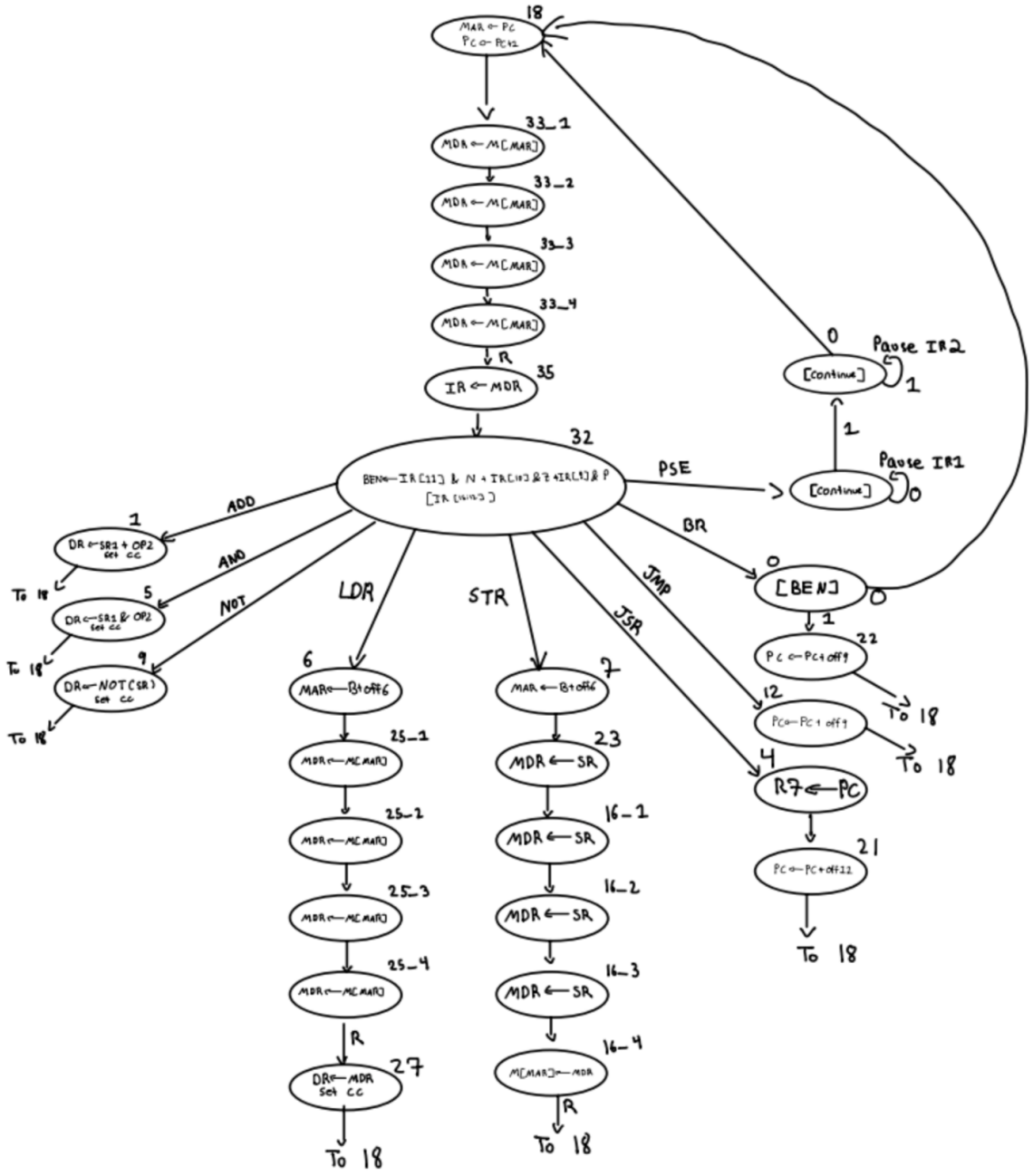
f. Description of the operation of the ISDU

. Named ISDU.sv, this is the control unit for the SLC-3. Describe in words how the ISDU controls the various components of the SLC-3 based on the current instruction.

The control unit of the LC3 processor incorporates a state machine and manages the flow of data through various inputs and output signals. It determines the appropriate registers to load, operations to perform on data, and mux selections, among other things. It establishes states for memory and arithmetic operations and integrates all logic components. It employs opcodes from the IR to determine the state to execute. It has two pause states that activate LED signals. Additionally, during memory read or write operations, four wait states are implemented to ensure that the computer receives the correct updated data memory within a minimum of four cycles.

The ISDU is a system that receives clock, reset, run, and continue signals, as well as a 4-bit OPCODE, IR bits IR[5] and IR[11], and an input BEN. It uses these signals to output instructions and control various signals that load registers, select muxes, and select operations. At the start, all loads and muxes are set to 0, along with other control signals. The ISDU begins in a halted state, then transitions to the fetch stage from state 18 to 32 before decoding the current instruction in state 32. Based on the OPCODE, the ISDU will perform an ADD, AND, BR, JMP, JSR, LDR, STR, NOT, or PAUSE operation. If in the PAUSE state, the ISDU will wait for input from switches and continue running when the continue button is pressed. The IR[5] bit is used to determine whether to perform an ADD or AND operation with immediate sign-extended values with the SR1 register. The BEN input is used in the BEN state to determine the next state based on whether it is high or low. Pressing the reset button clears all state signals and returns the ISDU to the halted state.

g. State Diagram of ISDU



Updated LC3 FSM - this version includes our extra wait states that we have in place of a ready signal - these appear for states 33, 25, and 16

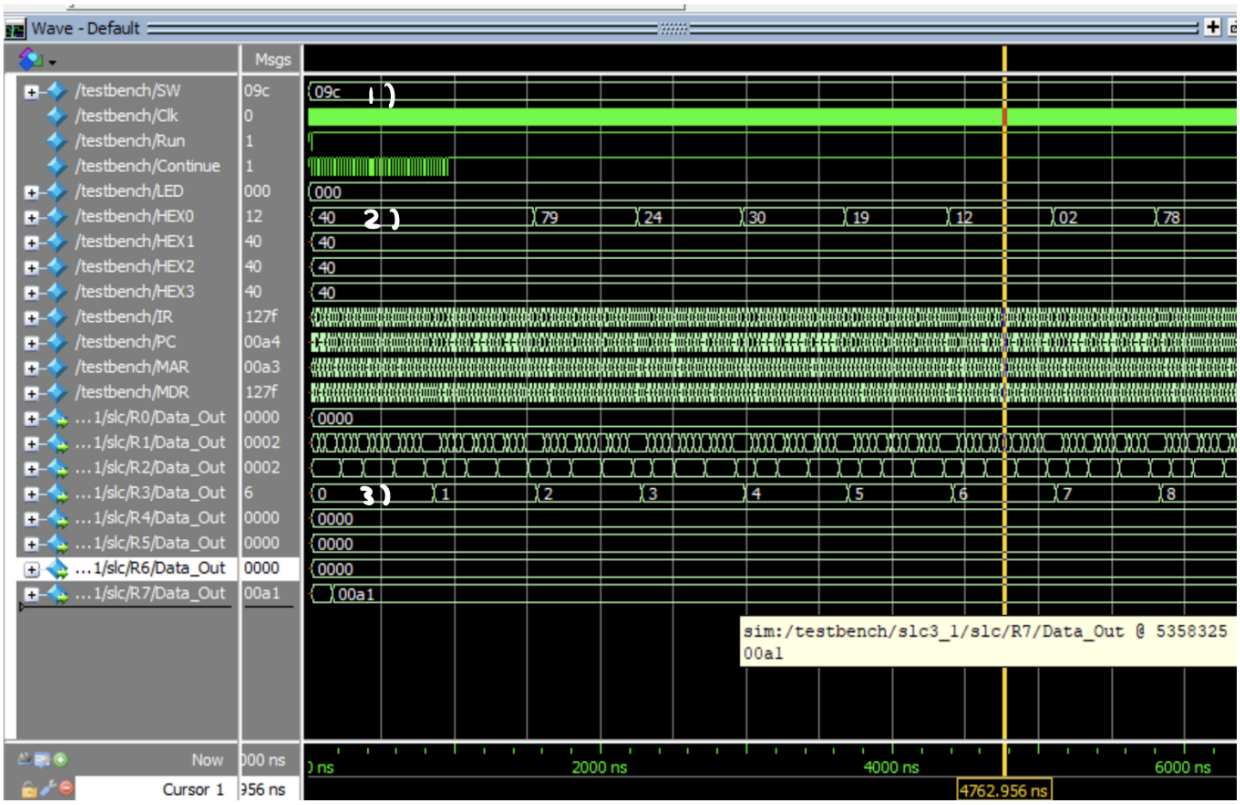
3. Simulations of SLC-3 Instructions

- **e. Simulate the completion of all 6 test programs, I/O Test 1, I/O Test 2, Self-Modifying doe, XOR, Multiplier and Sort**
- **e. Annotations for the above simulations**

```
1  module HexDriver (input  logic [3:0]  In0,
2                    output logic [6:0]  Out0);
3
4     always_comb
5     begin
6        unique case (In0)
7           4'b0000  : Out0 = 7'b1000000; // '0'
8           4'b0001  : Out0 = 7'b1111001; // '1'
9           4'b0010  : Out0 = 7'b0100100; // '2'
10          4'b0011  : Out0 = 7'b0110000; // '3'
11          4'b0100  : Out0 = 7'b0011001; // '4'
12          4'b0101  : Out0 = 7'b0010010; // '5'
13          4'b0110  : Out0 = 7'b0000010; // '6'
14          4'b0111  : Out0 = 7'b1111000; // '7'
15          4'b1000  : Out0 = 7'b0000000; // '8'
16          4'b1001  : Out0 = 7'b0010000; // '9'
17          4'b1010  : Out0 = 7'b0001000; // 'A'
18          4'b1011  : Out0 = 7'b0000011; // 'b'
19          4'b1100  : Out0 = 7'b1000110; // 'C'
20          4'b1101  : Out0 = 7'b0100001; // 'd'
21          4'b1110  : Out0 = 7'b0000110; // 'E'
22          4'b1111  : Out0 = 7'b0001110; // 'F'
23          default  : Out0 = 7'bX;
24       endcase
25    end
26
27 endmodule
```

Above is a key for reading the Hex displays on the Modelsim - showing the hex values for the corresponding binary

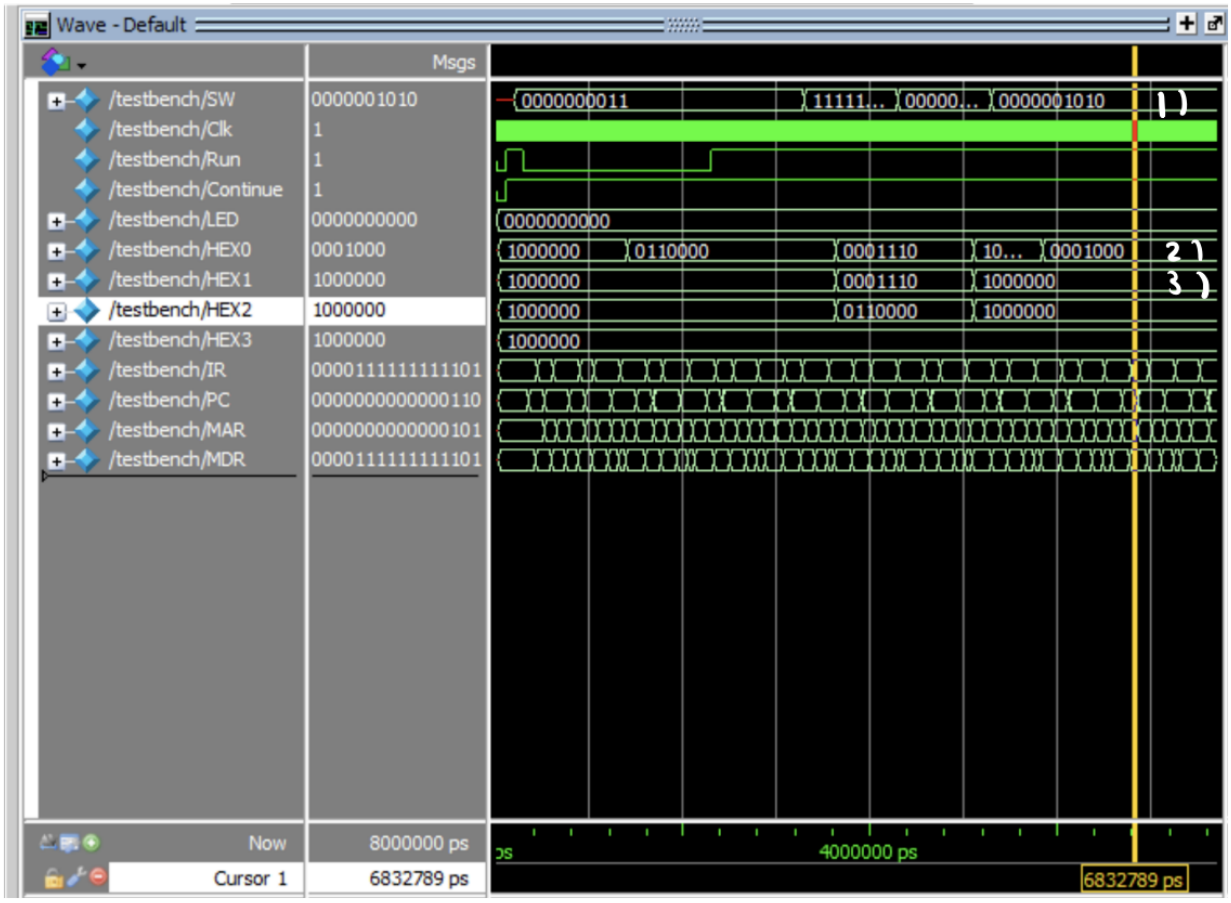
This image shows the contents of the hex driver file. Numbers 0-9 and letters A-F are assigned a certain value in binary corresponding to that number being displayed on the hexdriver on the FPGA. For example, on the RTL simulation, a hex driver value may read “1000000”, but this corresponds to “0” according to the hex driver module, so “0” would be displayed on the hex driver.



Autocount Modelsim data

AUTOCOUNT:

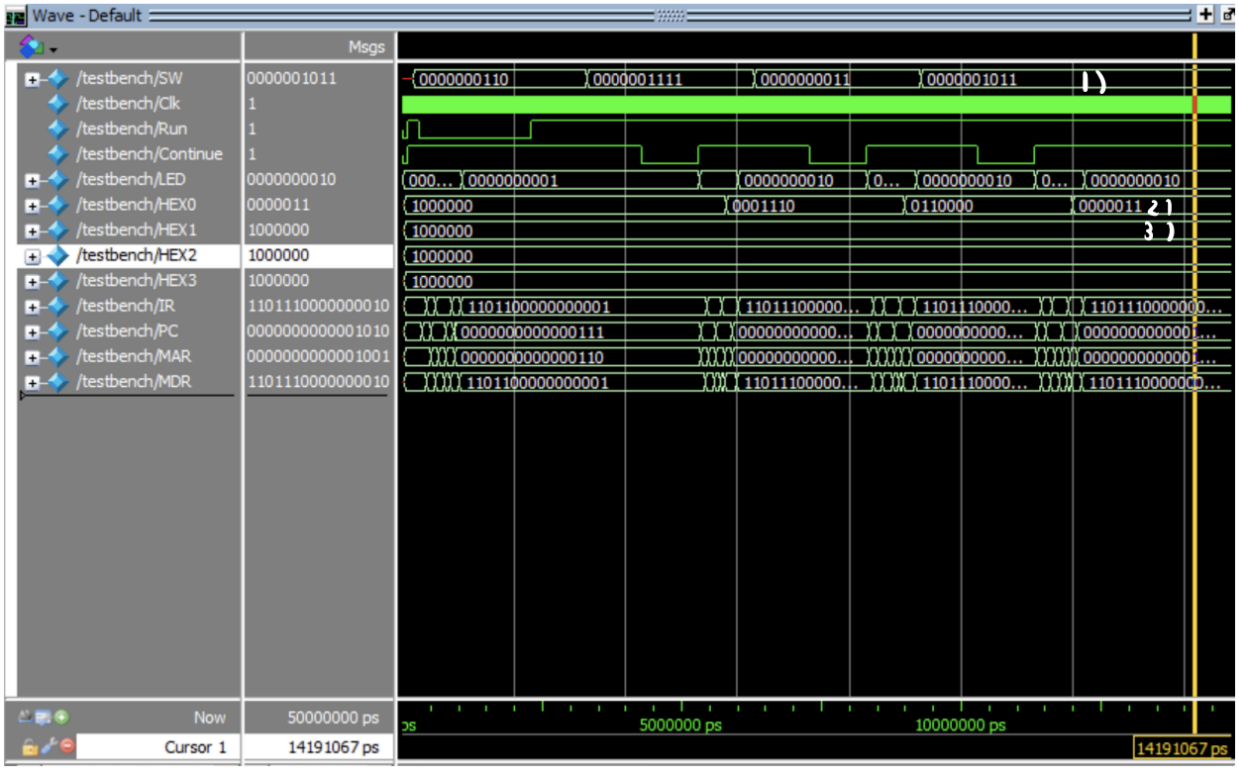
1) This cycle shows the switch value of x09C being loaded into the FPGA which loads in the auto count test.
2) This shows the numbers on the hex display counting from 0. In the hex driver file included in the 5.2 lab project, numbers 0-9 and letters A-F are assigned a certain value in binary corresponding to that number being displayed on the hexdriver on the FPGA. For example, all these cycles start at x40, which means all the hex drivers display x40 in the first cycle. In the hex driver file, x40 corresponds to the number "0", therefore, 0 is being displayed on the hexdrivers. 0 is then followed by 1, 2, 3, 5, 6, 7, and 8.
3) You can also look at the data out of Register 3 which displays the count starting at 0. I changed the settings to display in decimal so the auto count result can be easily read.



I/O test Modelsim data

I/O TEST #1:

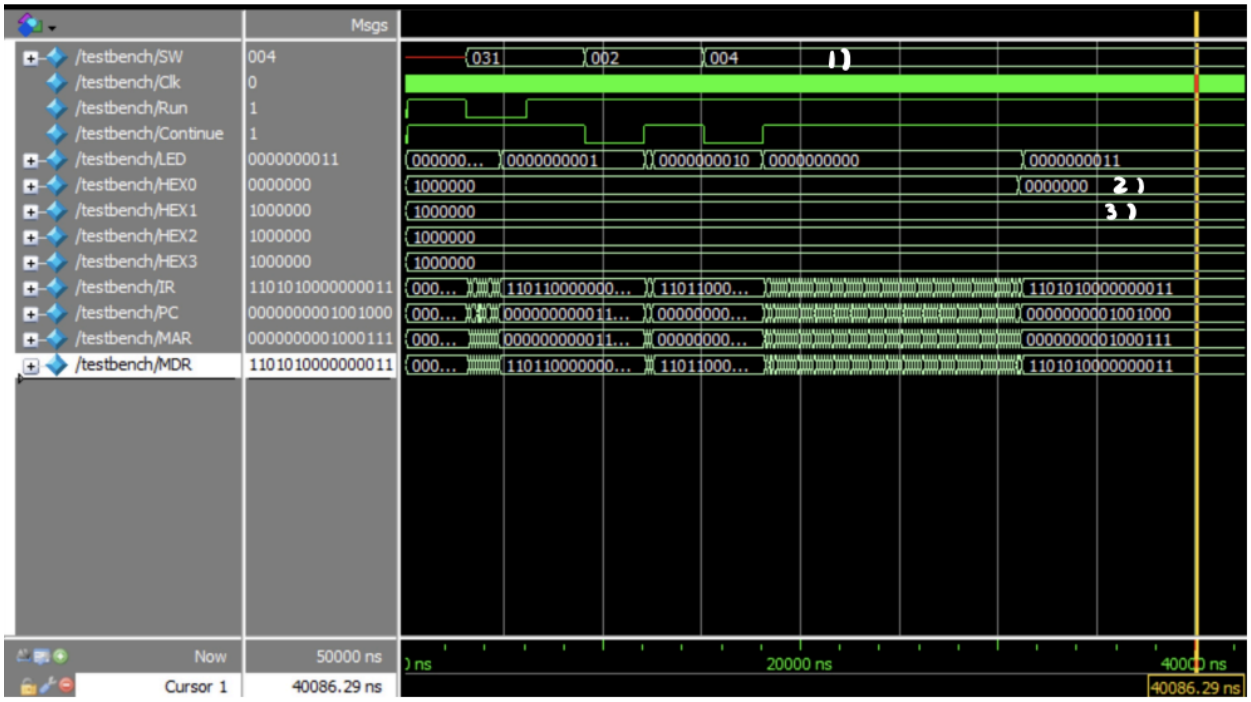
1) This cycle shows the switch value of x3 being loaded into the FPGA which loads in the I/O TEST #1. This allows the user to input switch values and then change the switch values, which results in these values being displayed on the hex. The first value loaded into the switches x3 followed by multiple other switch values ending at xA.
2) This shows the value of the switches being changed on the hex driver. In the hex driver file included in the 5.2 lab project, numbers 0-9 and letters A-F are assigned a certain value in binary corresponding to that number being displayed on the hexdriver on the FPGA. For example, all these cycles start at x40, which means the hex drivers display x40 in the first cycle. In the hex driver file, x40 corresponds to the number "0", therefore, 0 is being displayed on the hexdrivers. 0 is then followed by multiple switch values, ending with xA on the hexdrivers.
3) In this example, Hex Driver 1 and 2 also matter as in one of the cycles the switches display x03FF.



I/O test 2 Modelsim data

I/O TEST #2:

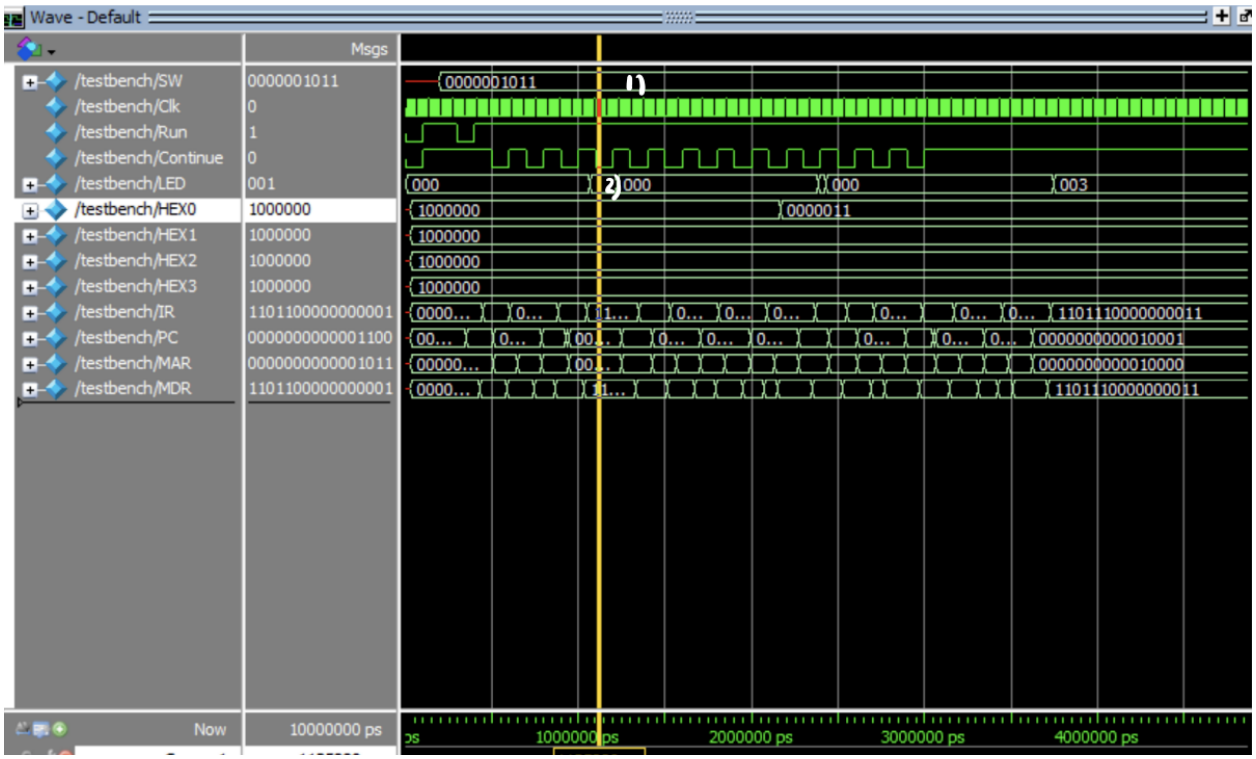
1) This cycle shows the switch value of x6 being loaded into the FPGA which loads in the I/O TEST 2. This allows the user to input switch values and then click the continue key to change the switch values, which results in these values being displayed on the hex. The first value loaded into the switches is x6 followed by multiple other switch values: xF, x3, xB.
2) This shows the value of the switches being changed on the hex driver. In the hex driver file included in the 5.2 lab project, numbers 0-9 and letters A-F are assigned a certain value in binary corresponding to that number being displayed on the hexdriver on the FPGA. For example, all these cycles start at x40, which means the hex drivers display x40 in the first cycle. In the hex driver file, x40 corresponds to the number "0", therefore, 0 is being displayed on the hexdrivers. The only hex driver that changes its value is HEX0 which changes from xF, to x3, to xB.
3) In this example, Hex Driver 1, 2, and 3 are always zero since the largest number input into the switches is xF. Therefore only HEX0 is used.



Multiplication test Modelsim Data

Multiplication Test:

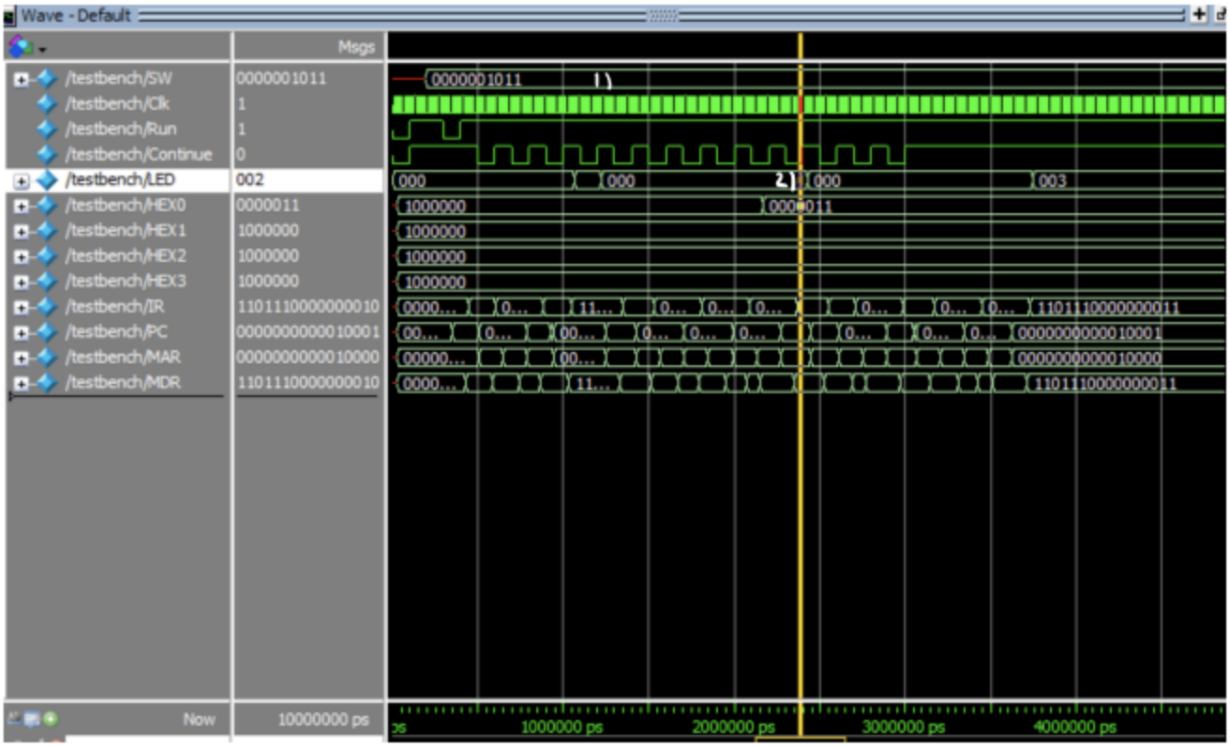
1) The first cycle shows the switch value of x31 being loaded into the FPGA which loads in the Multiplication Test. This allows the user to input one switch value, click continue, input another switch value, click continue, and then the product is displayed on the hex drivers. In this example, it is shown that x2, and x4 are loaded in the switches which means the end value displayed on the drivers should be 8 since 2*4 = 8.
2) This shows the value of HEX0 hex driver. In the hex driver file included in the 5.2 lab project, numbers 0-9 and letters A-F are assigned a certain value in binary corresponding to that number being displayed on the hexdriver on the FPGA. For example, all these cycles start at x40, which means the hex drivers display x40 in the first cycle. In the hex driver file, x40 corresponds to the number “0”, therefore, 0 is being displayed on the hexdrivers. The only hex driver that changes its value is HEX0 which changes from x31, to x2, to 4, to x8. The value x8 is the product of the multiplication.
3) In this example, Hex Driver 1, 2, and 3 are always zero since x0008 needs to be displayed on the Hex Drivers.



Self Modifying test Modelsim Data - picture 1

SELF MODIFYING TEST PART 1ST PICTURE:

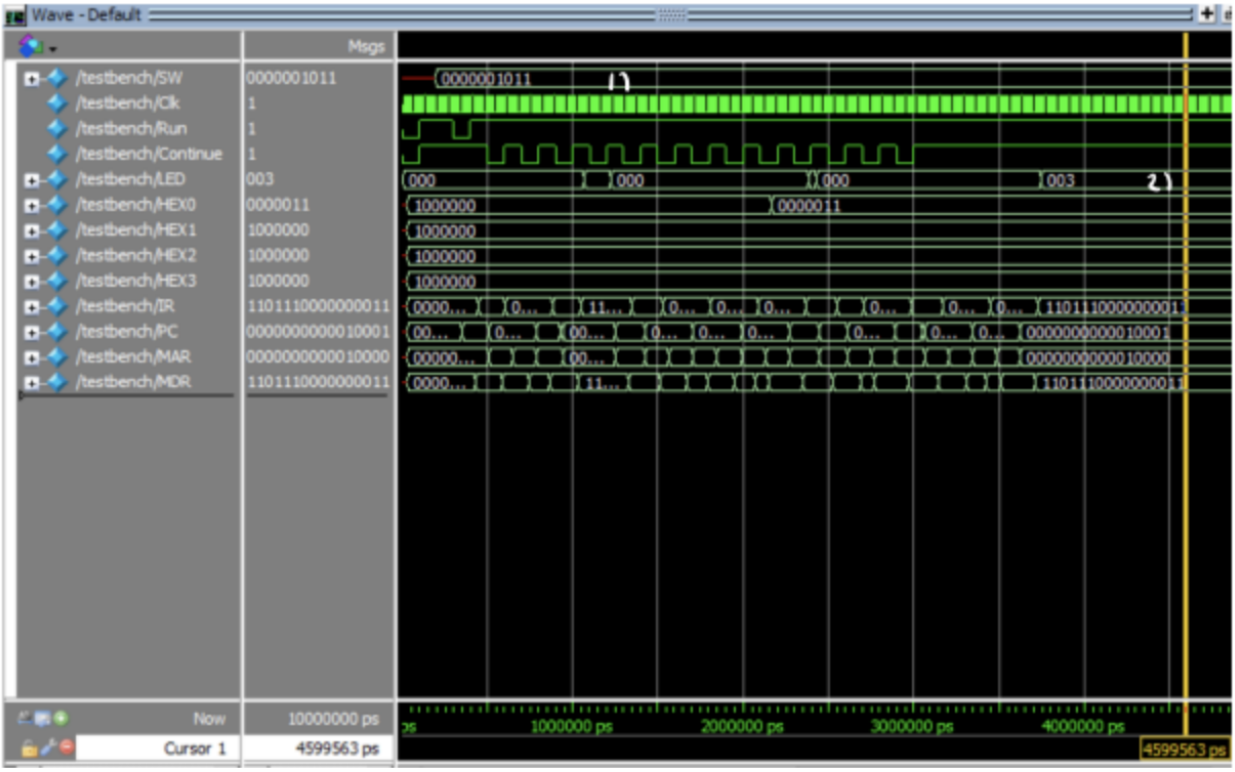
1) This cycle shows the switch value of xB being loaded into the FPGA which loads in the SELF MODIFYING TEST. This allows the user to click continue which causes the LED values to accumulate by 1 each time continue is pressed.
2) The LEDs are only displayed during the pause state, this is why there are zeros being displayed between each LED value that is incrementing. In this cycle, the LED value is 1.



Self Modifying test Modelsim Data - picture 2

SELF MODIFYING TEST PART 2nd PICTURE:

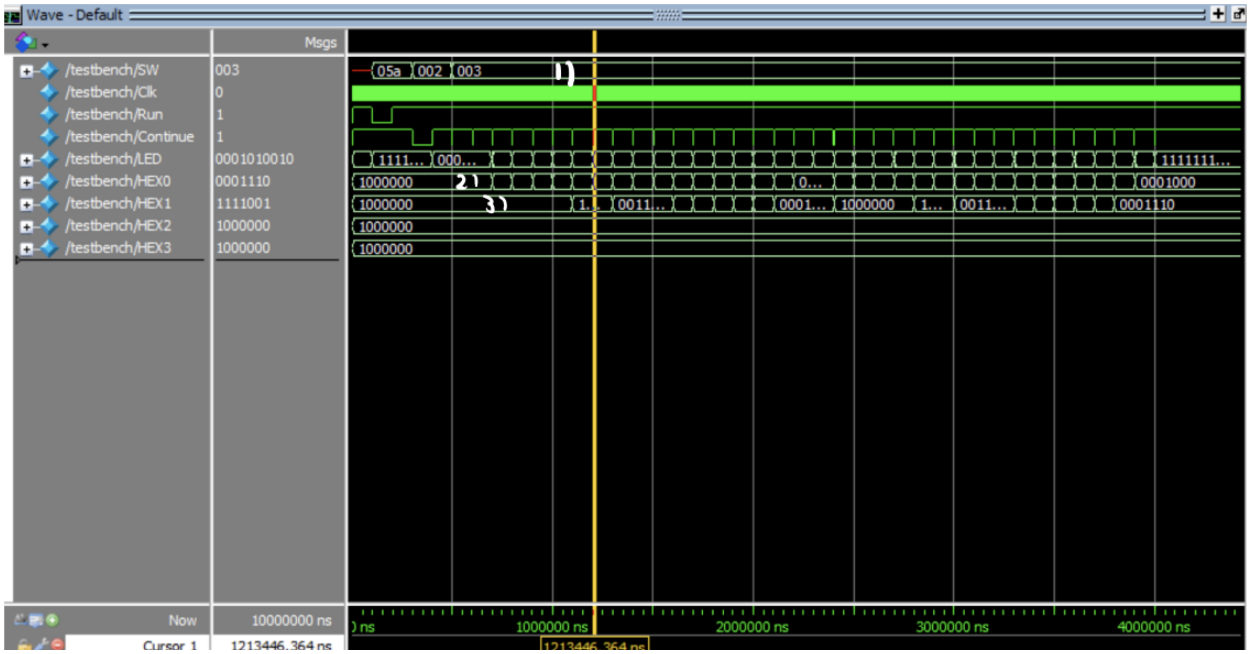
1) This cycle shows the switch value of xB being loaded into the FPGA which loads in the SELF MODIFYING TEST. This allows the user to click continue which causes the LED values to accumulate by 1 each time continue is pressed.
2) The LEDs are only displayed during the pause state, this is why there are zeros being displayed between each LED value that is incrementing. In this cycle, the LED value is 2 since it has accumulated by 1.



Self Modifying test Modelsim Data - picture 3

SELF MODIFYING TEST PART 3rd PICTURE:

1) This cycle shows the switch value of xB being loaded into the FPGA which loads in the SELF MODIFYING TEST. This allows the user to click continue which causes the LED values to accumulate by 1 each time continue is pressed.
2) The LEDs are only displayed during the pause state, this is why there are zeros being displayed between each LED value that is incrementing. In this cycle, the LED value is 3 since it has accumulated by 1.

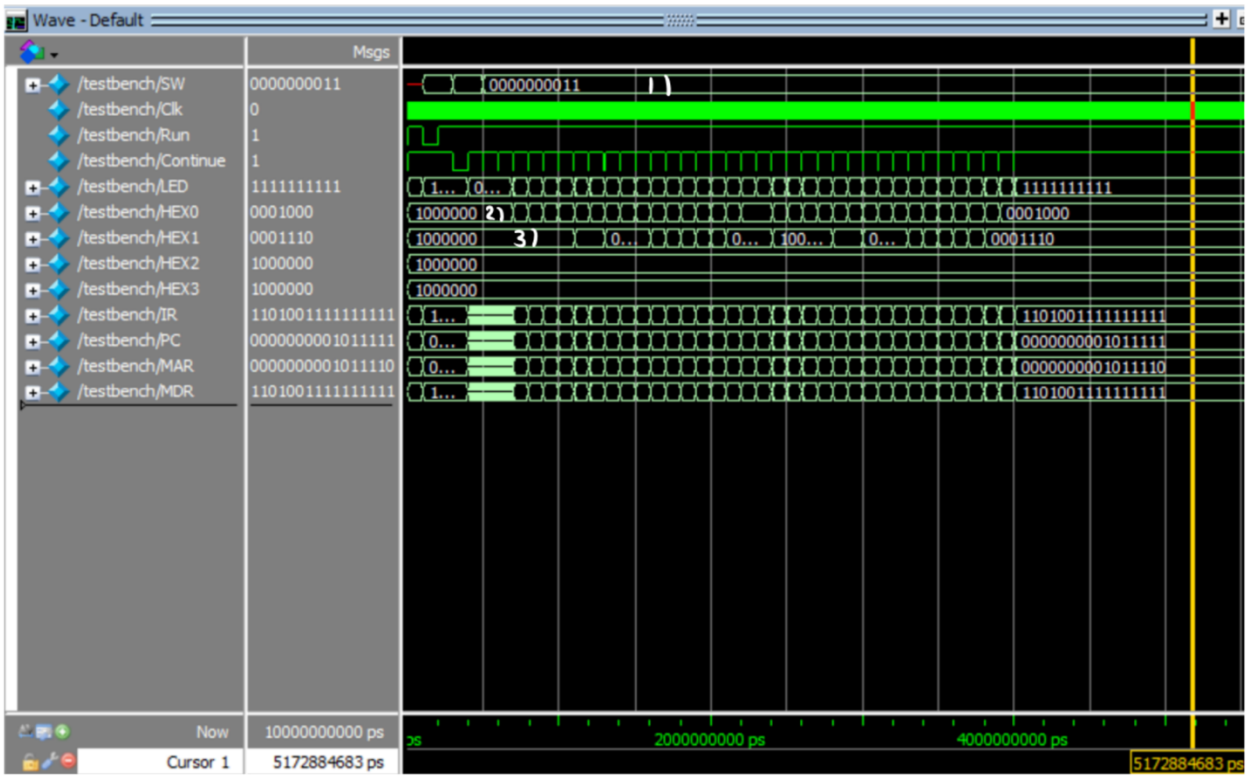


Sort test Modelsim data - picture 1

SORT TEST PART 1st PICTURE:

Multiplication Test:

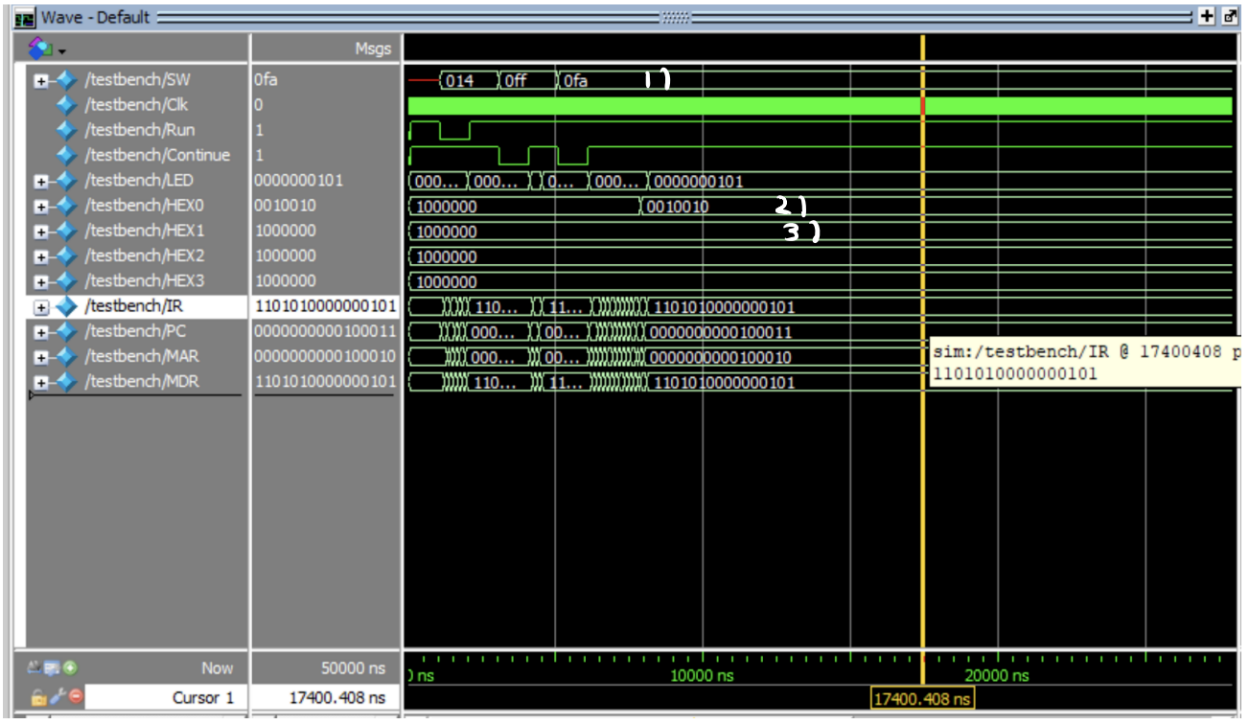
1) The first 3 cycles show the switch value of x5A being loaded into the FPGA which loads in the SORT Test. The next value loaded in the switches is x2, which sorts the array using the bubble sort algorithm. After pressing continue, the user then sets the switches to load in x3 and then presses continue again which displays the sorted array on the hex drivers.
2) This shows the value of the sorted array on the hexdrivers. In the hexdriver file included in the 5.2 lab project, numbers 0-9 and letters A-F are assigned a certain value in binary corresponding to that number being displayed on the hexdriver on the FPGA. For example, all these cycles start at x40, which means the hex drivers display x40 in the first cycle. In the hex driver file, x40 corresponds to the number "0", therefore, 0 is being displayed on the hex drivers. The only hex drivers that change their value are HEX0 and HEX1.
3) In this wave, HEX1 is displaying its values.



Sort test Modelsim data - picture 2

SORT TEST PART 2nd PICTURE:

1) The first 3 cycles show the switch value of x5A being loaded into the FPGA which loads in the SORT Test. The next value loaded in the switches is x2, which sorts the array using the bubble sort algorithm. After pressing continue, the user then sets the switches to load in x3 and then hit continue again which displays the sorted array on the hex drivers.
2) This shows the value of the sorted array on the hexdrivers. In the hexdriver file included in the 5.2 lab project, numbers 0-9 and letters A-F are assigned a certain value in binary corresponding to that number being displayed on the hexdriver on the FPGA. For example, all these cycles start at x40, which means the hex drivers display x40 in the first cycle. In the hex driver file, x40 corresponds to the number “0”, therefore, 0 is being display on the hex drivers. The only hex drivers that changes their value are HEX0 and HEX1. In this wave, HEX0 is displaying its values. This specifically highlights part of the last value in the sorted array which is “a”.
3) In this wave, HEX1 is displaying its values. This specifically highlights part of the last value in the sorted array which is “f”. Therefore, the last value in the sorted array that is displayed is x00fa.



XOR test Modelsim data

XOR TEST:

1) The first cycle shows the switch value of x14 being loaded into the FPGA which loads in the XOR Test. This allows the user to input one switch value, click continue, input another switch value, click continue, and then the XORed value is displayed on the hex drivers. In this example, it is shown that xff, and xfa are loaded in and which means the end value displayed on the drivers should be 5.
2) This shows the value of HEX0 on the hex driver. In the hex driver file included in the 5.2 lab project, numbers 0-9 and letters A-F are assigned a certain value in binary corresponding to that number being displayed on the hexdriver on the FPGA. For example, all these cycles start at x40, which means the hex drivers display x40 in the first cycle. In the hex driver file, x40 corresponds to the number "0", therefore, 0 is being displayed on the hexdrivers. The only hex driver that changes its value is HEX0 which changes to display x5 after the XOR operation is done.
3) In this example, Hex Driver 1, 2, and 3 are always zero since x0005 needs to be displayed on the Hex Drivers.

4. POST LAB QUESTIONS

1. **Fill out the Design Resources and Statistics table from Post-Lab question one**

LUT	1897
DSP	0
Memory (BRAM)	1216512
Flip-Flop	1161
Frequency (MHz)	65.49
Static Power (mW)	89.99
Dynamic Power (mV	9.94
Total Power (mW)	108.77

Above is the table with the statistics from our compilations on Quartus

- **What is MEM2IO used for, i.e. what is its main function?**

  The MEM2IO is the memory mapped bus based I/O, and its purpose is to move data from memory to an I/O device. In this lab, we use it to detect if we want to load or store the values from the memory address 0xFFFF. More specifically, the MEM2IO looks at the address 0xFFFF and determines if it should be loading the value of the switches (the starting address). On the other hand, the MEM2IO looks at the address 0xFFFF and could detect a store, which means that it will store the result of whatever function onto the hex displays. For the lab, the results are actually stored in a register and then displayed onto the switches rather than being stored directly into the SRAM.

- **What is the difference between BR and JMP instructions?**

  The main difference between the BR and JMP instructions is that the branch instruction happens based on the conditions of the previous line of code, more specifically it depends on the condition codes. These condition codes are N (negative), P (positive) and Z (zeros). Thus, depending on if the current value is a positive, negative or zero number, we will branch to a different line in the LC3 code. On the other hand, the JMP instruction happens unconditionally. That is, it will branch to a different line (subroutine) simply by just calling the function and has no regard for the current value of the program.

- **What is the purpose of the R signal in Patt and Patel? How do we compensate for the lack of the signal in our design? What implication does this have for synchronization?**

  In Patt and Patel, R is the ready signal. This ready signal is included because in states 33, 25 and 16, accessing the memory can take multiple cycles so the program waits until the memory makes the R signal high to indicate that reading from the memory has been completed. Since our design does not include this ready signal, we compensate for this by manually creating (at least) 3 wait states in our FSM for states 33, 25, and 16 where we simply stay in this same state for the 3 clock cycles to allot the proper time for memory accessing. The implications that this has for synchronization is that having the wait states ensures that we are accessing the memory at the right time and at the right frequency so that the data being transferred over is accurate.

5. CONCLUSION

- **Discuss functionality of your design. If the parts of your design did not work, discuss what could be done to fix it.**

  In the end, we are able to have our design function properly to execute each LC3 instruction as intended. However, there were some parts of the design that we needed to fix multiple times in order to get it correct. First, we initially had 3 wait states for our states 33, 25 and 16 as we thought that we needed only 3 extra clock cycles for us to be able to access memory. But, ended up needing 4 as it took a little longer for the program to access the memory. Additionally, we had trouble with testing our code in Modelsim. This is because we were unsure of how long to run the continue signals for each test in the test bench, especially for the sort test because there were a lot of instructions that we needed to get through from start to finish. To fix this, we simply had to make sure we ran the model sim for a long enough time so that each instruction could be seen. Finally, we had some difficulty when choosing the selects for our muxes. We made simple copying errors where we accidentally swapped the values of the selects. For example, if the input was 00, our mux selected the input that corresponded to the 01 value instead. This was simply from misreading the Appendix C portion of Patt and Patel.

- **Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.**

Some materials for the lab were confusing initially as we were given multiple diagrams that were different for the same data path. To avoid making the full LC3 data path, we referred to only the SLC3 diagrams. Other than this, nothing was unnecessarily difficult in the lab manual.