# ECE 385

Fall 2022

Experiment #5

Simple Computer SLC-3.2 in SystemVerilog

Ravi Thakkar, Siraj Khogeer

TA: Gavin Wu

10-11-2022

## Introduction

In this lab, we create a simple sixteen-bit SLC-3 processor using SystemVerilog. SLC-3 is a simplified version of the LC-3 ISA that has less instructions to work with than LC-3. The processor can execute nine instructions through a cycle of fetching the instruction, decoding, then executing the instructions on the registers that comprise the register unit, and then fetch the next instruction.

## Written Description and Diagrams of SLC-3

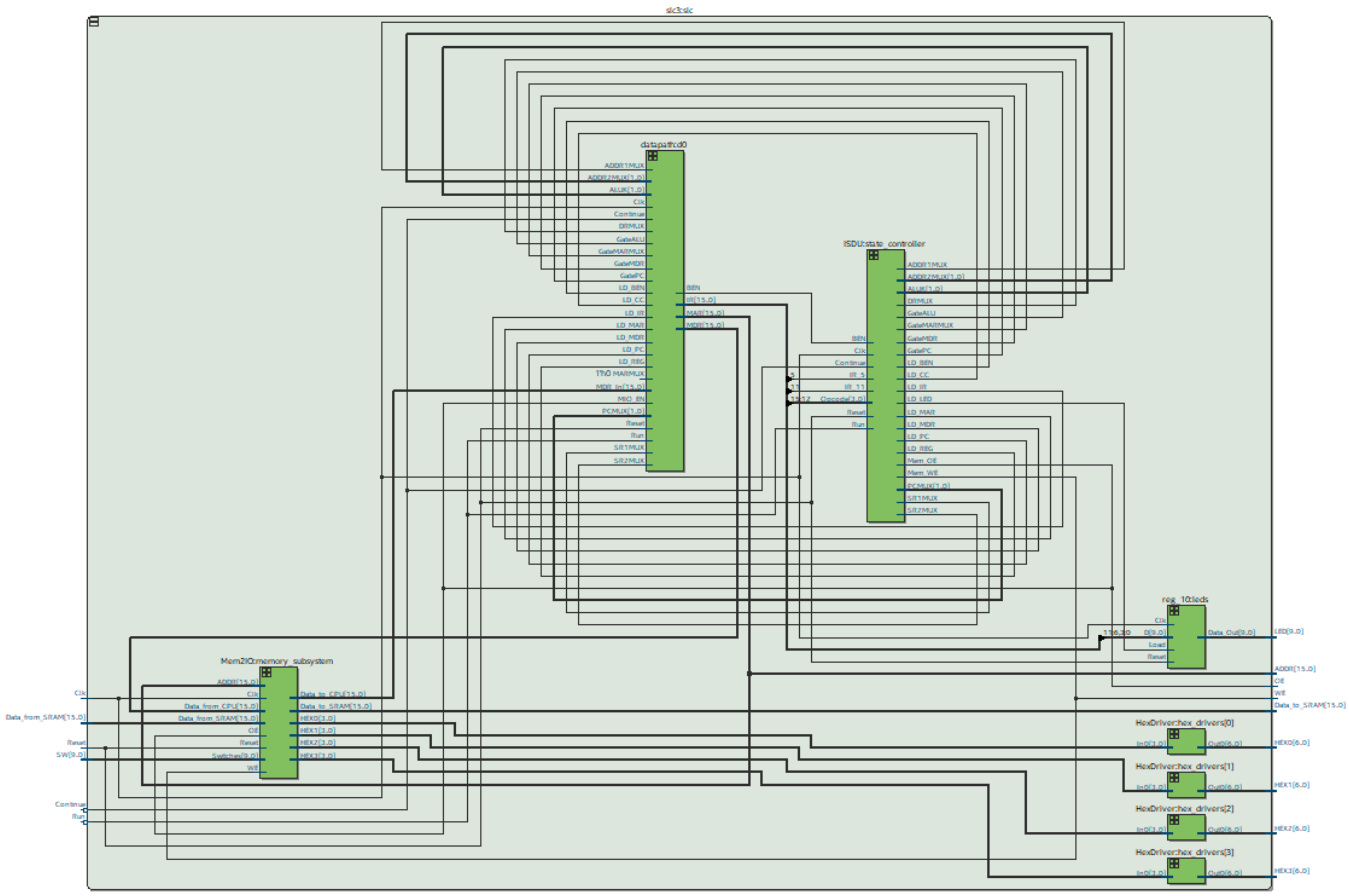
a. Summary of Operation

The SLC-3 processor uses a Fetch-Decode-Execute cycle which is implemented through a Finite State Machine.

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 program works by splitting itself into three parts. The ISDU, DATAPATH, and the MEM2IO modules. The ISDU controls the state of the circuit, the datapath controls how the data flows and is stored, and the MEM2IO controls the memory and the IO. The program starts in the halt state, and after initialization (Reseting) it waits for a run signal. When a run signal is first pressed the program moves onto state 18 which is the Fetch cycle. In this cycle the ISDU provides the proper control signals to load PC into MAR, increment PC, load M[MAR] into MDR, and then MDR into IR. This is done in multiple steps and when memory is involved, extra wait states are included. These signals are routed into the data path which has the respective IR… registers, MUXes, and adders. The datapath is essentially some registers, MUXes, and combinational logic following the requirements provided. The control signals and the datapath then interface with the MEM2IO to get the required data from memory or to write to it when required. After fetching, the system moves into the decode state, which depending on the value of IR, sets the next state to be equal to one of the 9 functions provided. The ISDU is implemented using two sets of unique cases, one for the next_state, and one for the control signals. After the decode state the system moves onto the execution state which differs for each function, but essentially consists of unique cases that have control signals matching what is needed to load, add, or whatever operation is needed in the data path. Each execution has multiple states and can require inputs, such as the pause state, but they generally return to the fetch state after done and continue with the program. During the execute state LD_CC is set whenever required, and LD_BEN is set during the decode state, depending on the NZP registers.

## Block Diagram of slc3.sv




## Written Description of .sv Modules

Module: **ISDU** in ISDU.sv

Inputs: Clk, Reset, Run, Continue, Opcode[3:0], IR_5, IR_11, BEN

Outputs: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, PCMUX[1:0], DRMUX, SR1MUX, SR2MUX, ADDR1MUX, ADDR2MUX[1:0], ALUK, Mem_OE, Mem_WE

Description:

State Diagram:


This module is the control unit that is based on the state diagram of SLC-3, as seen in the Lab 5 manual. After the current instruction is fetched, the opcode is decoded and the next state of the FSM is designated. Once this next state is designated, certain signals are activated which are passed into the datapath so the instruction can be executed.

The way that the code itself is organized is that we first assign the next state based on the current state, which is done through a case statement. For State 32, which acts as a decoder, we assign the next state based on the opcode that is inputted through another case statement. The second part of the code is another case statement that takes in the current state and changes values for relevant control signals to be passed into the datapath depending on the state diagram provided in the lab manual.

Purpose: The purpose of this module is to implement the finite state machine for SLC-3. The finite state machine passes in control signals based on the current state of the FSM.

Module: **slc3** in slc3.sv

Inputs: SW[9:0], Clk, Reset, Run, Continue, Data_from_SRAM[15:0]

Outputs: LED[9:0], OE, WE, HEX0[6:0], HEX1[6:0], HEX2[6:0], HEX3[6:0], ADDR[15:0], Data_to_SRAM[15:0]

Description: This module instantiates other modules such as the datapath, Mem2IO and ISDU and passes in the same signals into all of these instantiations.

Purpose: This module acts as a top level for all of the necessary modules to implement the SLC-3 processor.

Package: **SLC3_2** in SLC3_2.sv

Inputs: N/A

Outputs: N/A

Description: Assigns opcodes (binary digits) for the nine instructions and provides register and branch condition aliases. Also creates the functions for each of the nine operations.

Purpose: This isn't really a module but it is important to talk about because this is how the opcodes can be decoded and what each operation is set to do is defined.

Module: **slc3_sramtop** in slc3_sramtop.sv

Inputs: SW[9:0], Clk, Run, Continue

Outputs: LED[9:0], HEX0[6:0], HEX1[6:0], HEX2[6:0], HEX3[6:0], test[15:0]

Description: This is the top-level module provided to us to use for when we program the FPGA.

Purpose:  This module allows us to program the FPGA and use its SRAM.

Module: **datapath** in datapath.sv

Inputs:  Clk, Reset, Run, Continue, LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, GatePC, GateMDR, GateALU, GateMARMUX, SR2MUX, ADDR1MUX, MARMUX, MIO_EN, DRMUX, SR1MUX,  ([1:0] PCMUX, ADDR2MUX, ALUK), [15:0] MDR_In

Outputs: BEN, ([15:0] MAR, MDR, IR, BUSt)

Description: The datapath module is the module where all the data in the SLC-3 moves. The datapath has many inputs that come directly from the ISDU, and some from the top-level module. The datapath takes the control signals front the ISDU and instantiates registers, MUXes, and unique cases to model what each state in the LC3 fsm requires. The LD signals are used to load the specific registers whenever the control circuit says to. There are three main things in the datapath. The registers, the BUS, and the logic. The registers are either instantiated or a reg_file is used. The BUS is an important part of the datapath that is used to connect some parts together such as the IR,MDR,PC, and ALU. The BUS has 4 inputs and 1 output. Initially the BUS was supposed to be a wire with some tri-state buffers, but it was decided to instead implement it with a 4-input MUX. The GATE signals are used as the select

to the MUX, and their respective names are the output, for example GatePC makes BUS = PC. The datapath also has a ALU, which takes SR1_out and SR2_out and performs operations on them. These operations are dictated by ALUK and the registers come from reg_file. Another part of the data path is the MUXes, which are used to select between different operations, inputs, or registers. These have been selected as PCMUX, ADDER MUX etc, which follows the datapath diagram shown in the lab. The MUX signals usually choose between constants, or specific bits of the IR register, for example SR1MUX is used to switch between IR[8:6] and IR[11:9] for the SR1 select to the reg_file. The full description can be seen in the datapath diagram. Finally, the datapath has three signals that interact with the memory. MAR, MDR, and MDR_in. These are 16-bit signals that are used to read and write from the memory using the MEM2IO in the higher level module.

Purpose: This module is used as the datapath of the CPU. It holds all of the registers and most of the combinational logic. The datapath takes signals from the ISDU and memory and it stores, operates, and moves it according to the FSM, and the state diagram. The datapath is used to transfer data between registers, operate on data, and read/write from memory. It is where most data has to flow through.

Module: **reg_file** in reg_file.sv

Inputs: Clk, Reset, LD_REG, [2:0]DR, [2:0]SR1, [2:0]SR2, [15:0]D

Outputs: [15:0]SR1_out, [15:0]SR2_out

Description: This module is used as the register file where R0-R7 are held. This is a synchronous module with Clk and Reset. The reg_file works by having DR dictate the destination (which register to load), and SR1 and SR2 dictate the output. The module instantiates 8 16-bit registers R0-R7. The data input to all registers is D[15:0]. Each register has a load signal l0-l7. A unique statement is used to set the load signals. DR sets L(DR) = LD_REG. This essentially means that DR dictates where LD_REG passes through to. SR1 and SR2 also are used as the case statements for two unique case blocks. SR1 and SR2 are converted to decimal and whichever register corresponds to SR1 or SR2 are outputted to SR1_out and SR2_out.

Purpose: This module is used as the register unit in the datapath of the S-LC-3 implementation. This module holds 8 different registers and controls which are loaded and outputted using the input signals. The data input to all registers is D which is the BUS in the datapath. Used in the implementation of SLC-3, a requirement

Module: **reg_16** in reg_16.sv

Inputs: Clk, Reset, Load, [15:0]D

Outputs: [15:0]Data_Out

Description: 16-bit register that has D inputs and output Data_Out. With reset. 16-bit synchronous register. Load is used to load D into Data_Out.

Purpose: This register is used as the basis for most of the registers in the SLC-3 program. This is used to create the IR,PC,MDR, and MAR registers. Essentially this is a flip-flop that holds its value until a positive edge of a clock cycle and a load signal are imputed. Also used in reg_file.

Module: **reg_3** in reg_3.sv

Inputs: Clk, Reset, Load, [2:0]D

Outputs: [2:0]Data_Out

Description: 3-bit register, that has data inputs And output Data_Out. With reset. 3-bit synchronous register. Load is used to load D into Data_Out

Purpose: Used to hold the NZP register values. Has load set to LD_CC. Loaded when loading register files and when branching.

Module: **reg_1** in reg_1.sv

Inputs: Clk, Reset, D

Outputs: Data_Out

Description: 1-bit register, that has data inputs And output Data_Out. With reset. 1-bit synchronous register

Purpose: This module is used to hold the value of the BEN register, which is the Branch Enable register which is used in the fetch and Jump/Branch instructions.

Module: **HexDriver** in HexDriver.sv

Inputs: In0[3:0]

Outputs: Out0[6:0]

Description: HexDriver used to map the inputs to outputs that can be seen on the hex display.

Purpose: Maps the input to outputs that can be displayed on the HEX display.

Module: **slc3_testtop** in slc3_testtop.sv

Inputs: SW[9:0], Clk, Run, Continue

Outputs: LED[9:0], HEX0[6:0], HEX1[6:0], HEX2[6:0], HEX3[6:0], test[15:0]

Description: This is the top-level module provided to us to use for when we simulate on Modelsim.

Purpose: This module allows us to simulate the SLC-3 processor on Modelsim.

Module: **Mem2IO** in Mem2IO.sv

Inputs: Clk, Reset, ADDR[15:0], OE, WE, Switches[9:0], Data_From_CPU[15:0], Data_From_SRAM[15:0]

Outputs: Data_to_CPU[15:0], Data_to_SRAM[15:0], HEX0[3:0], HEX1[3:0], HEX2[3:0], HEX3[3:0]

Description: This module interfaces with the I/O for SLC-3, which consists of the switches, the LED, and the hexes using data from the CPU and the SRAM and can also send signals to the CPU and SRAM based on inputs.

Purpose: The purpose of this module is to allow for I/O of the SLC-3 processor and SRAM. Acts as a bridge between Memory, IO, and the datapath/ISDU.

---

Task: **memory_contents** in memory_contents.sv

Inputs: N/A

Outputs: mem_array[0:255]

Description: Instantiates an array of memory locations that are set with specific operations.

Purpose: Allows for all of the tests to ensure the processor is working properly to be loaded onto the on-chip memory.

---

Module: **fourinputmux** in muxes.sv

Inputs:[15:0]input 1, input 2, input 3, input 4

[3:0]select

Outputs: [15:0]out

Description: This module is a four input mux that has 4 select pins.This is used as an alternative to the tri state buffer. Whenever one of the select pins is high, the respective input is passed through to the output.

Purpose: Used to implement the Gate tristate buffers in the datapath. Used to remove floating inputs and implement the tristate buffers, as the FPGA might not support it.

---

Module: syncronizers.sv

Inputs: Clk, d, (in some cases reset)

Outputs: q

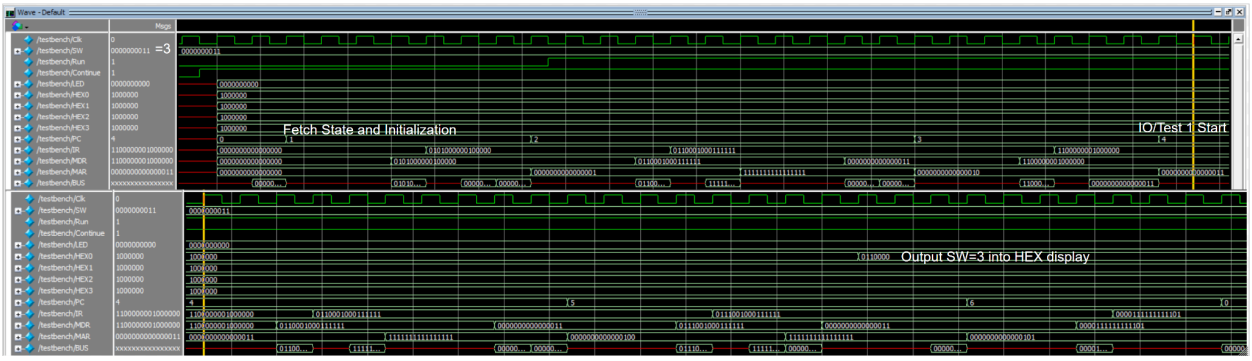
Description: The synchronizers are used to synchronize the button presses with the clock cycles, This is done by essentially passing them into a flip-flop that only registers their value each clock cycle. Some synchronizers have reset q to 1 and some to 0.

Purpose: This is used to avoid timing issues and debouncing such as metastability which is caused when a button press changes when not expected. It is used to make sure the entire design is synchronous.

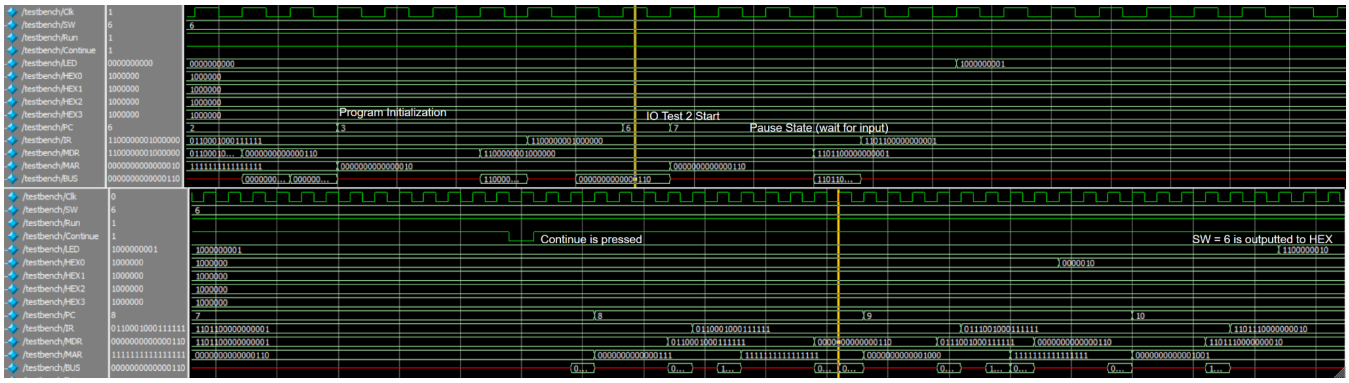
## Simulations of SLC-3 Instructions

a. Simulate the completion of all 6 test programs, I/O Test 1, I/O Test 2, Self-Modifying Code, XOR, Multiplier and Sort.

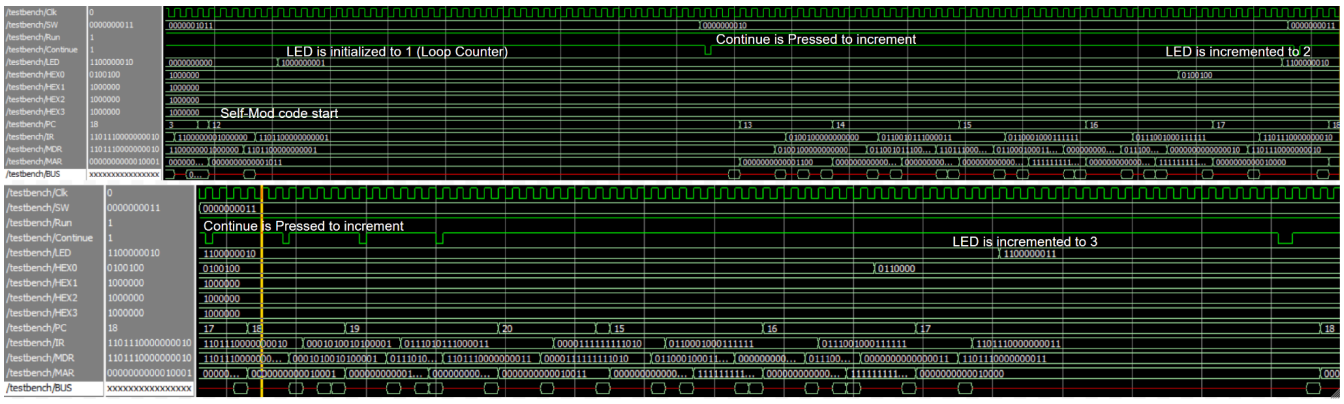
## IO/Test 1



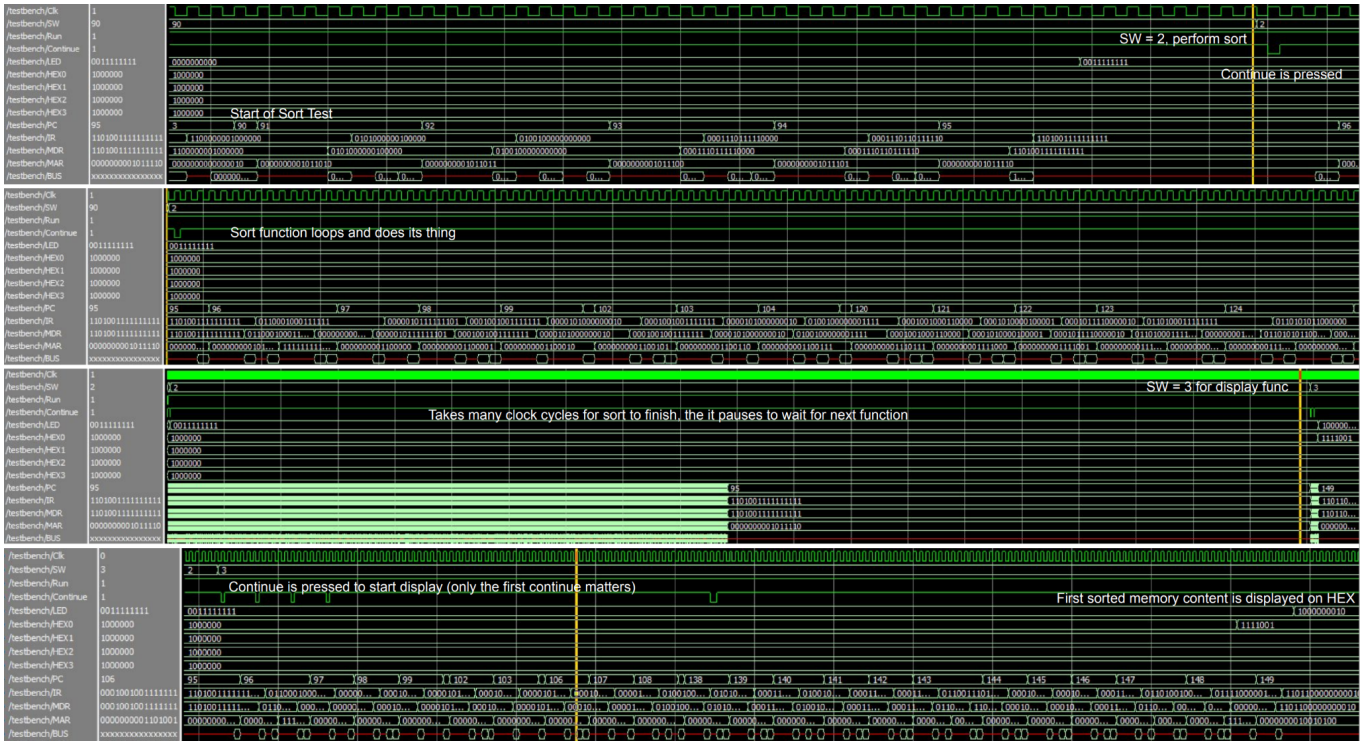

## IO/Test 2



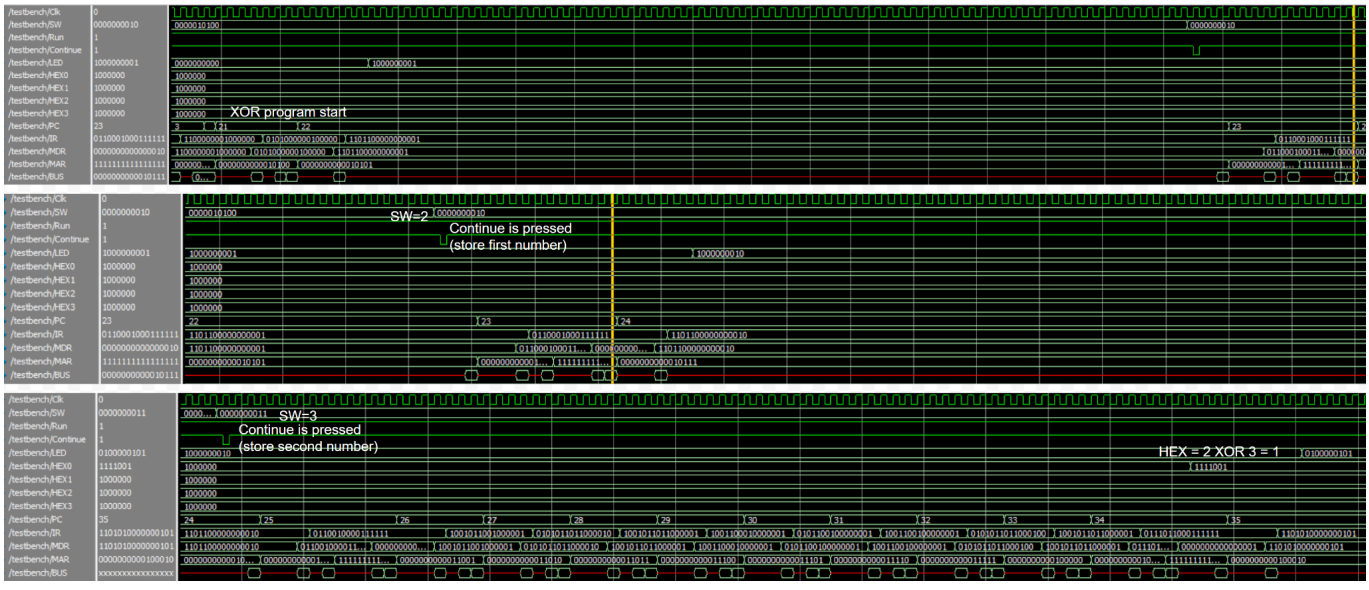

## Self-Modifying Code Test (Ignore multiple continue presses)



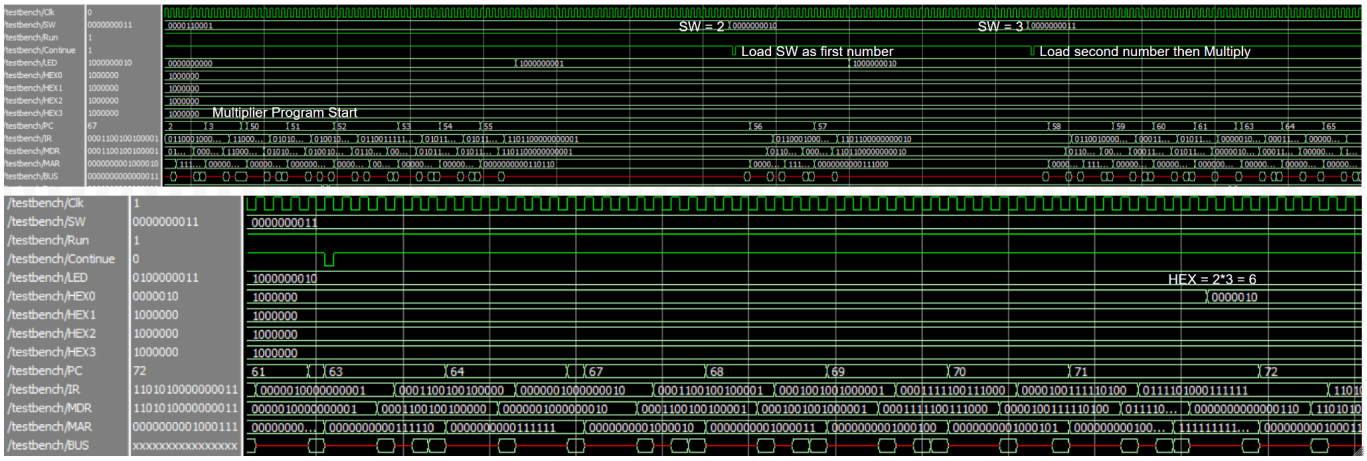
## Sort Test




## XOR Test



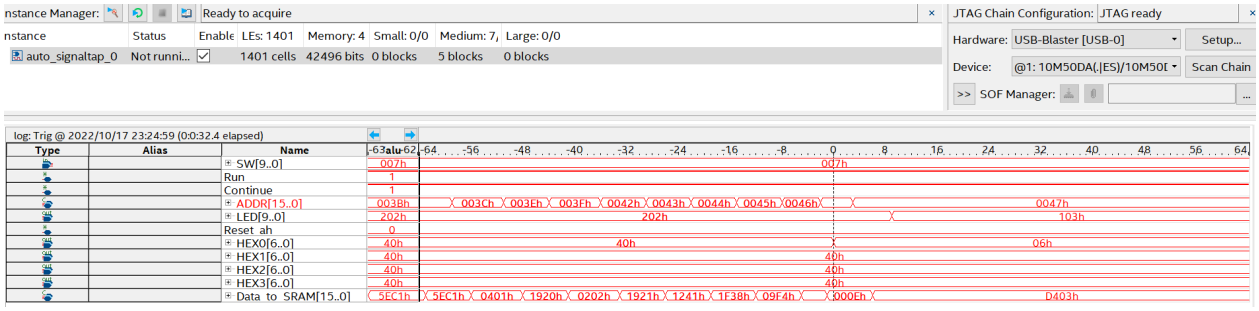

## Multiplier test



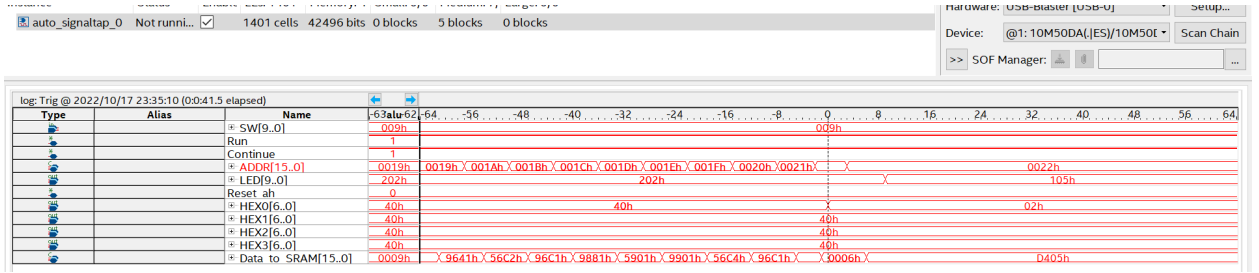

b. Annotations for the above simulations, should include at a minimum, start of the test program, any user input (for example, entering the numbers in the multiplier test), and reading the expected result.

## Extra Credit: SignalTap Traces

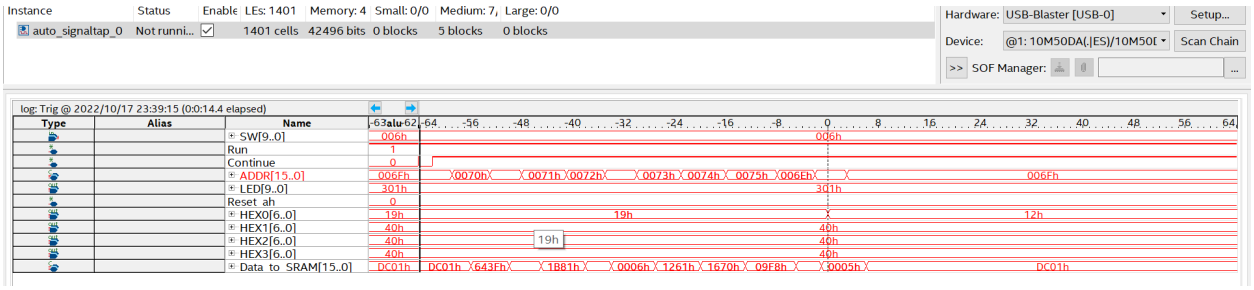
### Multiplier Test




### XOR Test




### Sort Test




## Postlab Questions

1. Refer to the Design Resources and Statistics in IQT.17-19 and complete the following design statistics table

| | |
|---|---|
| LUT | 923 |
| DSP | 0 |
| BRAM | 16,384 |
| Flip-Flop | 268 |
| Frequency Mhz | 91.2 |
| Static Power mW | 89.94 |
| Dynamic Power mW | 0 |
| Total Power mW | 98.7 |

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

MEM2IO is the unit that interfaces with the I/O and the CPU and the memory. Data from the CPU and the memory is inputted into the unit as well as data from the switches, and the output to the CPU and memory can be rewritten, and an output can be sent to the hex display

b. What is the difference between BR and JMP instructions?

BR is a branch instruction, which acts as a conditional instruction evaluating whether an input is negative, zero or positive to make the decision to branch and go to a target address or go to PC+1, but JMP is an unconditional jump instruction. BR supports a smaller offset to a target address than JMP and that is because of the fact that BR has bits to evaluate NZP and JMP doesn't.

c. 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 implications does this have for synchronization?

The R signal is supposed to tell the control unit that the process of reading or writing memory is complete. In our design, we didn't have an R signal so instead we had to use a fixed number of clock signals to wait until the memory completes the R/W process. Since we didn't use an R signal and instead used three clock signals, we don't need to worry about synchronization.

**Conclusion**

Overall, this lab was enjoyable but also a bit challenging. It was really enjoyable due to the fact that we were developing a subset of LC-3 which we learned in ECE 120, so we were already a bit familiar with what was going on in the datapath and the state diagram, but it was also interesting to look back at LC-3 after all of the new things we have learned as now we're halfway through ECE 385 and have learned a lot of new concepts. It was challenging to sift through all of the SystemVerilog code and try to understand what needed to be changed and what didn't, especially in the first week, but it turned out to be pretty simple during the first week. As soon as we got a good grasp of what needed to be done, the rest of the work was not very challenging but it was still a bit tedious because you needed to be careful to get things working right.

One thing that really helped us was basically completing the datapath early on, during the first week, instead of waiting until the second week, which balanced our workload between the two weeks.

Our design at the time of the second demonstration was functional and we didn't have any parts that weren't functional. During the first week, we had some issues with getting it working on the FPGA but we found out that it was due to a simple mistake which was easily corrected.