ECE 385
Fall 2021
Experiment #5

Simple Computer SLC-3.2 in SystemVerilog

Michael Stoens (mstoens2)
Ralph Balita (rbalita2)
Lab Section: AB3

1) Introduction: Summary of the basic functionality of the SLC-3 processor

The purpose of experiment 5.1 is to design and implement the *fetch phase (states 18, 33, and 35)* for the 16-bit simple microprocessor (SLC-3) using SystemVerilog. The operations in the *fetch phase* serve as a basis for all other higher level operations (found in the *decode and execute* phases). The purpose of experiment 5.2 is to extend our datapath and control unit modules to implement the decode and execute phases. Combined, experiments 5.1 and 5.2 implement a fully functioning SLC-3. The SLC-3 is a simplified version of the LC-3 that we have learned about in previous courses such as ECE 120 and 220. The SLC-3 is a 16-bit architecture that can perform nine different operations. These operations include eight of the operations from LC3: ADD, AND, NOT, LDR, STR, JSR, JMP, BR, as well as one new operation: PSE. These basic operations can be combined together in a sequence to implement the more advanced functions as seen in this lab such as multiplication, XOR operations and sorting algorithms.

2) Written Description and Diagrams of SLC-3

Summary of Operation

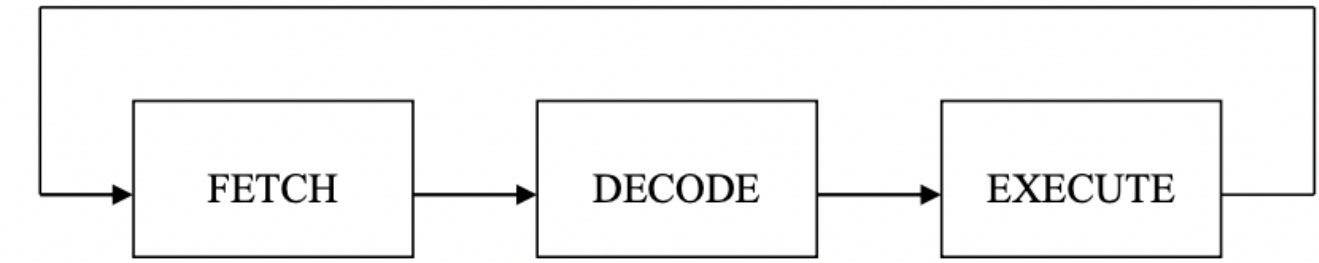
The three phases of the SLC-3 are fetch, decode, and execute. The fetch phase performs a sequence of operations to *fetch* the next instruction from memory and place it into the instruction register (IR). The decode phase utilizes the opcode of the current instruction to determine or *decode* the sequence of states required to complete the given operation. The execute phase then completes or *executes* the previously determined sequence of operations needed to complete the given instruction. At the end of the execute phase, the system returns to the fetch phase to restart the process.

Describe in words how the SLC-3 performs its functions. Fetch - Decode - Execute cycle

Fetch
The fetch phase consists of the 3 operations listed below, each of which is the basis for performing tasks on the SLC-3. First, the Program Counter (PC) is loaded into the Memory Address Register (MAR), the PC is then incremented. Second, the data at the memory location provided to the MAR (the PC value) is loaded into the Memory Data Register (MDR). Third, the data from the MDR is loaded into the IR.
(1) MAR <- PC, PC <- PC + 1 (18) **(Mem. Addr. Reg. takes PC, PC increments)** (2) MDR <- M[MAR] (33) **(Mem. Data Reg takes instruction data from memory)** (3) IR <- MDR (35) **(Instruction data loaded into Instruction Register)**

Decode
The decode phase is a one step phase of updating the Branch Enable Register (BEN). In the Decode state, the control unit utilizes the opcode of the current instruction (IR[15:12]) to determine the next state.
(1) BEN <- IR[11] & N + IR[10] & Z + IR[9] & P **(Update BEN register)**

Execute
The execute phase completes the remaining states(s) and operations need to execute the given instruction. The operations completed in the execute phase consist of four types:(1) Arithmetic Logic Unit (ALU) operations, (2) Data Movement, (3) Program Counter control and (4) Pause. A more in-depth description of execution will be described in the *Datapath Implementation*.
(1) ADD(1), ADDi(1), AND(5), ANDi(5), NOT (9) **(uses ALU to compute)** (2) LDR(6->25->27), STR(7->23->16) **(uses MAR and MDR to access memory)** (3) BR(0), JMP(12), JSR(4->21)**(changes for PC)** (4) PSE(12) **(Display Vector)**



Datapath Implementations.

The output signals of the Instruction Sequence Decoder Unit (ISDU) are used in the datapath to perform various operations including arithmetic operations, accessing memory, and changing the PC. Within the datapath are the registers to hold the various values of the SLC-3, the muxes and decoders needed to control the flow of data to and from those registers or memory locations, as well as the logic for the NZP and BEN.

- Datapath Bus Decoder-Mux.

  The datapath uses a 4:1 mux and a 4:2 decoder to send signals to the bus. This is done because the FPGA on the DE-10 does not have tri-state buffers and therefore the gate signals from the MARMUX, PC, ALU and MDR must be implemented in a different way. By using a Decoder and a Mux we can implement the same functionality as Tri-State buffers and ensure that only one signal is driving the bus at a time.

- Registers.

  To hold values, the datapath has five 16-bit utility registers (PC_Reg, MAR_Reg, MDR_Reg, IR_Reg, and LED_Reg) as well as eight 16-bit general registers (R0-R7). Also within the datapath are four 1-bit registers, which are used to hold the NZP values for conditional codes and the 1-bit BEN register.

- Operational Muxes.

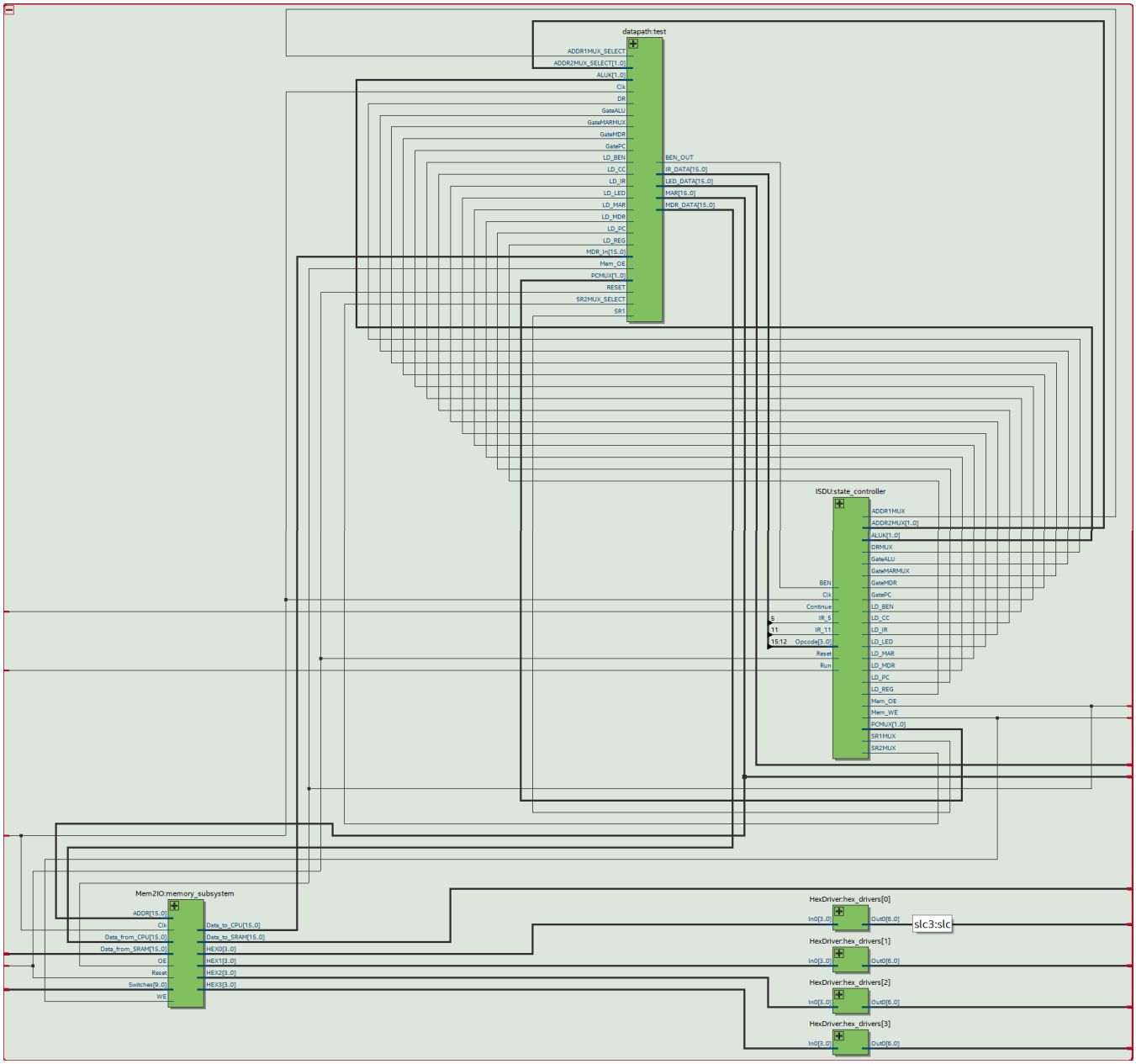
  The operational muxes are used to route the correct data to various components within the datapath based on signals from the control unit. Each of these muxes are implemented in a way that helps us optimize our debugging. By defining any invalid or unused mux select signals as dont care values, the output of the mux from an incorrect input will be easily recognizable on ModelSim.

- NZP and BEN

  Each bit of the NZP logic is checked individually using ‘if, else if, else’ statements and loaded into their respective registers using signals from the control unit. We first check if the most significant bit of the bus is a 1, which would indicate that the value is negative to determine our N value. Next, we check if the value of the bus is zero to determine our Z value. Finally, if the value of the bus is not negative or zero, we have determined that the value must be positive and set our P value. As seen in the Decode State, each of the outputs of the NZP registers is then AND’ed with its corresponding NZP bit of the IR and OR’ed together to determine the input to the BEN register.

MUX	Possible Outputs
PC_MUX:	PC+1, BUS, ADDER
Mem_OE (MDR input Mux):	BUS , MEM1IO
DR (RegFile: Destination Reg):	IR[11:9], register 7
SR1	R0 - R7
EXTERAL_SR1_MUX	IR[11:9], IR[8:6]
SR2	R0 - R7
SR2MUX_SELECT	SR2 OUT, Imm5
ADDR2MUX_SELECT	ADDR2MUX_OUT takes x0000 || SEXT6 || SEXT9 || SEXT11
ADDR1MUX_SELECT	ADDR1MUX_OUT takes PC_DATA || SR1_OUT

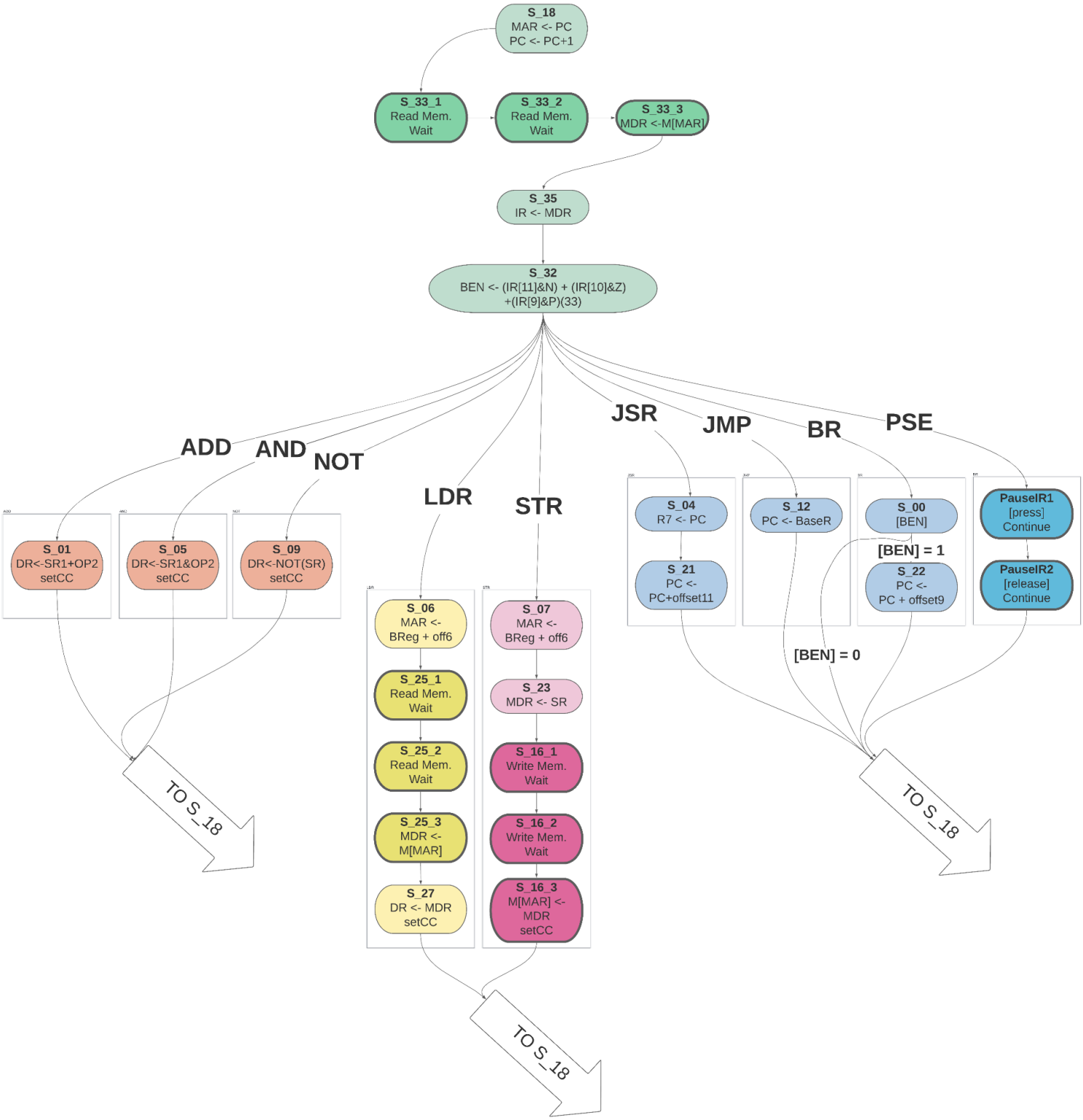
Block Diagram of slc3.sv.



Description of the operation of the ISDU

The SLC-3 control unit, known as the *Instruction Sequence Decoder Unit (ISDU)*, is used to provide the next state logic and datapath signals. As seen in the State Diagram, each execution begins with the Fetch and Decode states. In each of these states, the ISDU outputs the correct datapath signals to perform the operation for that state (register load signals, mux select, ect). In the decode state, the next state is determined using the opcode of the current instruction. Once the decode state determines the next state using the opcode, the state machine proceeds through the sequence of states needed to complete the selected operation while also sending the needed control signals to the datapath. Most states in the ISDU have an unconditional next state (exceptions being the Decode state and the first state of the BR instruction) Similarly, each state has predefined output signals to complete the operation of that state. Both the next state and datapath output logic is implemented using “unique case (state)” statements in SystemVerilog.

State Diagram of ISDU.



ISDU Control Signals

State	Datapath Control Signal
S_01	SR1MUX = 1'b1; SR2MUX = IR_5; ALUK = 2'b00; GateALU = 1'b1; LD_REG = 1'b1; LD_CC = 1'b1;
S_05	SR1MUX = 1'b1; SR2MUX = IR_5; ALUK = 2'b01; GateALU = 1'b1; LD_REG = 1'b1; LD_CC = 1'b1;
S_09	SR1MUX = 1'b1; ALUK = 2'b10; GateALU = 1'b1; LD_REG = 1'b1; LD_CC = 1'b1;

State	Datapath Control Signal
S_18	GatePC = 1'b1; LD_MAR = 1'b1; PCMUX = 2'b00; LD_PC = 1'b1;
S_33_1	Mem_OE = 1'b1;
S_33_2	Mem_OE = 1'b1;
S_33_3	Mem_OE = 1'b1; LD_MDR = 1'b1;
S_35	GateMDR = 1'b1; LD_IR = 1'b1;
S_32	LD_BEN = 1'b1;

State	Datapath Control Signal
S_04	GatePC = 1'b1; DRMUX = 1'b1; LD_REG = 1'b1;
S_21	ADDR2MUX = 2'b11; ADDR1MUX = 1'b0; PCMUX = 2'b10; LD_PC = 1'b1;
S_12	SR1MUX = 1'b1; ALUK = 2'b11; GateALU = 1'b1; PCMUX = 2'b01; LD_PC = 1'b1;
S_00	
S_22	ADDR2MUX = 2'b10; ADDR1MUX = 1'b0; PCMUX = 2'b10; LD_PC = 1'b1;

State	Datapath Control Signal
S_06	SR1MUX = 1'b1; ADDR1MUX = 1'b1; ADDR2MUX = 2'b01; GateMARMUX = 1'b1; LD_MAR = 1'b1;
S_27	SR1MUX = 1'b1; ADDR1MUX = 1'b1; ADDR2MUX = 2'b01; GateMARMUX = 1'b1; LD_MAR = 1'b1;
S_25_1	Mem_OE = 1'b1;
S_25_2	Mem_OE = 1'b1;
S_25_3	Mem_OE = 1'b1; LD_MDR = 1'b1;

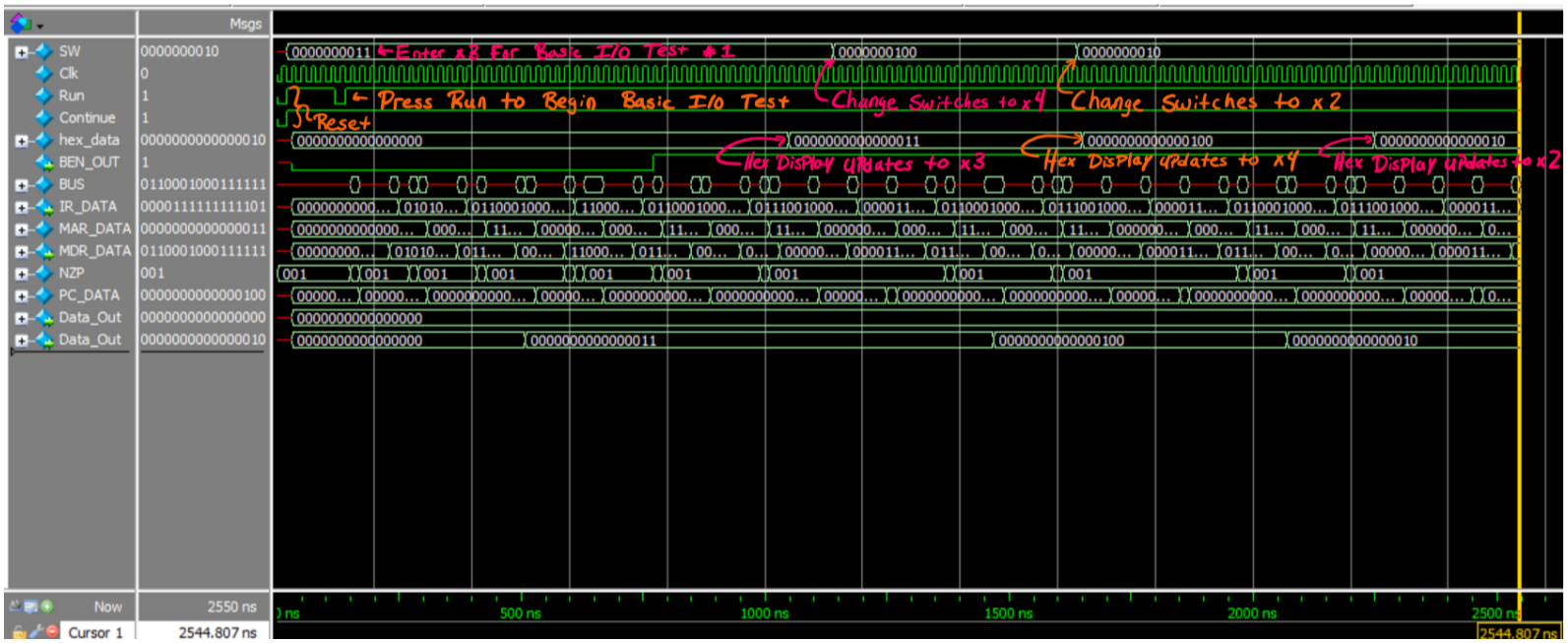
State	Datapath Control Signal
S_07	SR1MUX = 1'b1; ADDR1MUX = 1'b1; ADDR2MUX = 2'b01; GateMARMUX = 1'b1; LD_MAR = 1'b1;
S_23	SR1MUX = 1'b0; ALUK = 2'b11; GateALU = 1'b1; LD_MDR = 1'b1;
S_16_1	Mem_WE = 1'b1;
S_16_2	Mem_WE = 1'b1;
S_16_1	Mem_WE = 1'b1;

Written description of all .SV modules

Module	Inputs	Outputs	Description	Purpose
Syncronizers.sv	Clk, d	q	Synchronizes asynchronous input signals to the Clk signal	Synchronizes the pushbutton signals to the Clk for use with the FPGA
HexDriver.sv	**[3:0]** In0	**[6:0]** Out0	This is used to control the HEX display. Each Input has a hard coded output to display a number on the HEX Display	Translates binary values to signals needed to display those values on the HEX displays of the DE-10
slc3_testtop.sv	Clk, Run, Continue **[9:0]** SW	**[6:0]** HEX0, HEX1, HEX2,HEX3 **[7:0]** LED	Top level module for testing and simulation of the SLC-3 Microprocessor.	Implements all of the other modules and connects all signals.
lsc3_sramtop.sv	Clk, Run, Continue **[9:0]** SW	**[6:0]** HEX0, HEX1, HEX2,HEX3 **[9:0]** LED	Top level module for implementing the SLC-3 Microprocessor on the DE-10	Implements all of the other modules and connects all signals.
test_memory.sv	Reset, Clk, Rden, wren, **[15:0]** data **[9:0]** address	**[15:0]** readout	Test Memory for testing and simulation of the SLC-3 Microprocessor	Instantiated by slc3_testtop in order to create memory for testing of the SLC-3
SLC3_2.sv	None	None	Implements aliases and functions for LC-3 operations	Allows us to call functions instead of individual operations as well as refer to registers by name instead of binary values
slc3.sv	Clk, Reset, Run, Continue **[9:0]** SW **[15:0]** Data_from_SRAM	OE, WE **[6:0]** HEX0, HEX1, HEX2,HEX3 **[15:0]** ADDR, Data_to_SRAM	Initiates MEM2IO, datapath, ISDU, and hexdrivers	This module serves to instantiate the major components of the SLC-3.
memory_contents.sv	None	None	Holds all of the instructions for the SLC3. Also servers are general memory to load and store data	This module contains all of the preprogrammed tests to verify the functionality of our SLC-3
Instantiateram.sv	Clk, Reset	**[15:0]** ADDR, data Wren	Instantiates random access memory on the DE-10	Instantiated by sl3_ramtop in order to create memory on the hardware for the SLC-3
Mem2IO.sv	Clk, Reset, OE, WE **[9:0]** Switches **[15:0]** ADDR, Data_from_CPU, Data_from_SRAM,	**[6:0]** HEX0, HEX1, HEX2,HEX3 **[15:0]** Data_to_SRAM, Data_to_CPU	if the address is xFFFF (-1),MEM_2_IO will read from the switches. else, MEM_2_IO will read from the memory.	MEM2IO serves as an interface between the memory and the FPGA,
datapath.sv	Clk,GateMARMUX, GatePC, GateALU, GateMDR, LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, RESET, Mem_OE, DR, SR1, SR2MUX_Select, ADDR1MUX_Select **[1:0]** PCMUX, ALUK, ADDR2MUX_SELECT	BEN_OUT **[15:0]** MDR_IN, MAR, MDR_DATA, IR_DATA, LED_DATA,	This module initializes the datapath components, including all muxes, registers, and the BUS. A detailed description of the datapath operations can be found under *Datapath Implementations*	Utilizes the control signals from the ISDU in order to control the flow of data in the S-LC3
ISDU.sv	Clk, Reset, Run, Continue, **[3:0]** OpCode, IR_5, IR_11, BEN,	LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateALU, GateMARMUX, MEM_OE, MemWE **[1:0]** PCMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, ADDR2MUX, ALUK,	This module implements a finite state machine to control the output signals for the datapath. It starts with a ***fetch*** state to determine the IR, and current state. Next, it will ***decode*** the opcode and ***execute*** the operation. A detailed description of the ISDU can be found under *Operation of the ISDU*	This module controls the next state logic of the SLC-3. It also generates the control signals needed in each state which are used by the datapath
ALU.sv	**[1:0]** ALUK **[15:0]** ALUK_A, ALUK_B	**[15:0]** ALU_OUT	This module takes 2 16-bit values (A and B) and outputs either an arithmetic ADD, a logic AND, a logic NOT, or pass A depending on the value from the control unit	It is used in the S-LC3 in order to perform arithmetic and logical operations as well as data movement.
Register_File.sv	Clk, Reset, LD_REG, **[2:0]** DRMUX_OUT, EX_SR1MUX_OUT, SR2 **[15:0]** BUS	**[15:0]** SR1_OUT, SR2_OUT	Implements eight general purpose 16-bit registers as well as the logic needed to select a specific register using signals from the datapath	Provides eight 16-bit storage locations for SLC-3 programs to utilize
register_1bit.sv	Clk, Reset, Load, Din	Dout	Implements a 1-bit register with reset and load functionality	Used to implement registers N, Z, P and BEN
Register.sv	Clk, Reset, Load, Shift_In, Shift_En, **[15:0]** Din	Shift_Out **[15:0]** Data_Out	Implements a 16-bit registers with reset and load functionality	Used to implement PC_Reg, MAR_Reg, MDR_Reg, IR_Reg, and LED_Reg, as well as the eight general purpose registers
testbenchlab5.sv	None	None	Testbench used to set values and implement control signals for simulation.	Used to debug and verify if the program is functioning correctly.

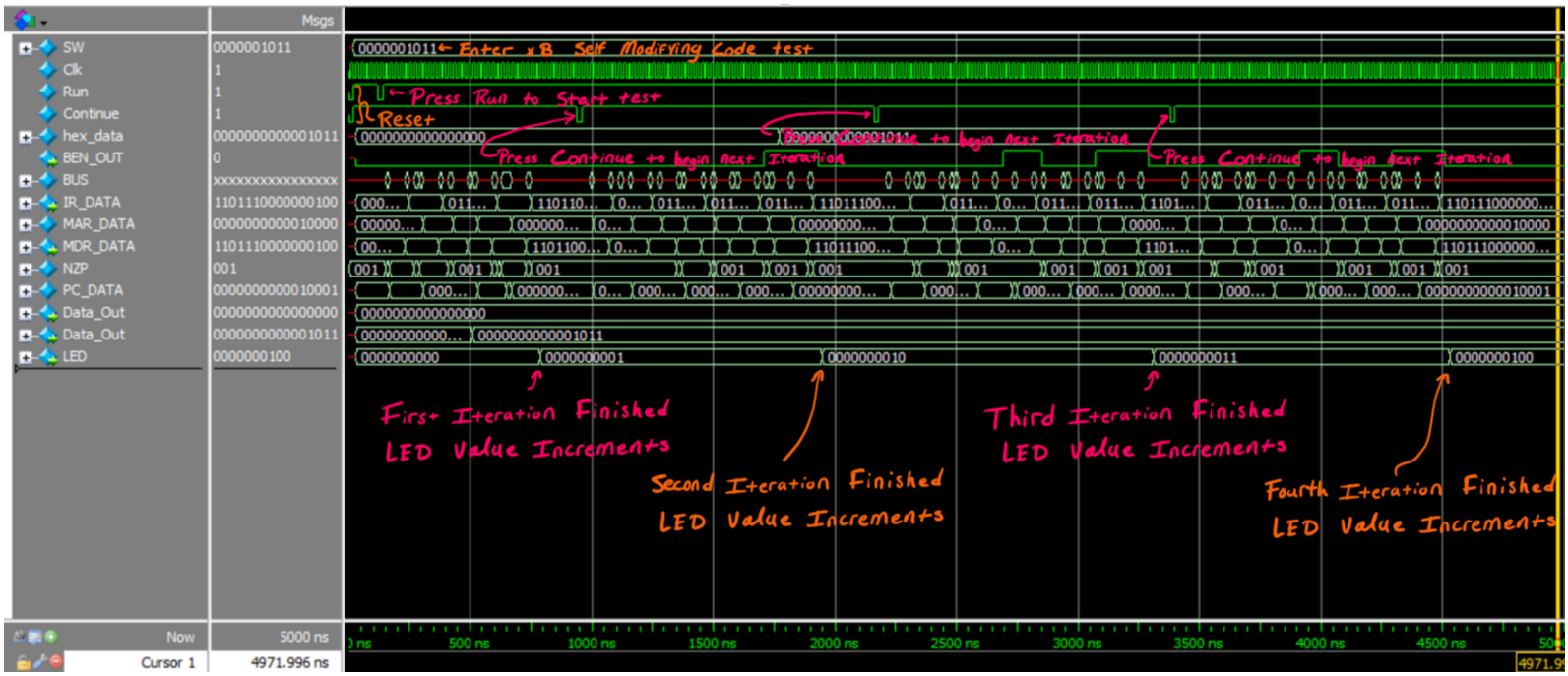
3) Simulations of SLC-3 Instructions

Basic I/O Test #1

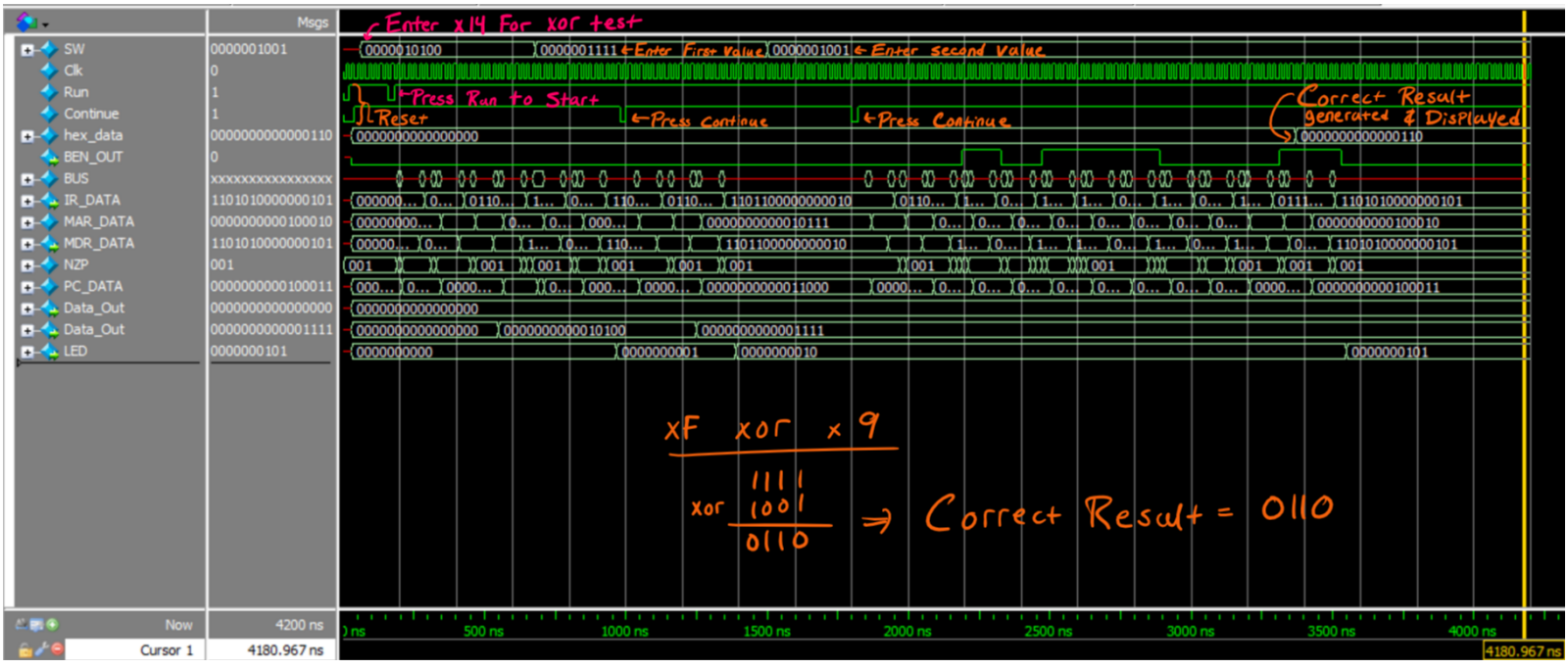


Basic I/O Test #2

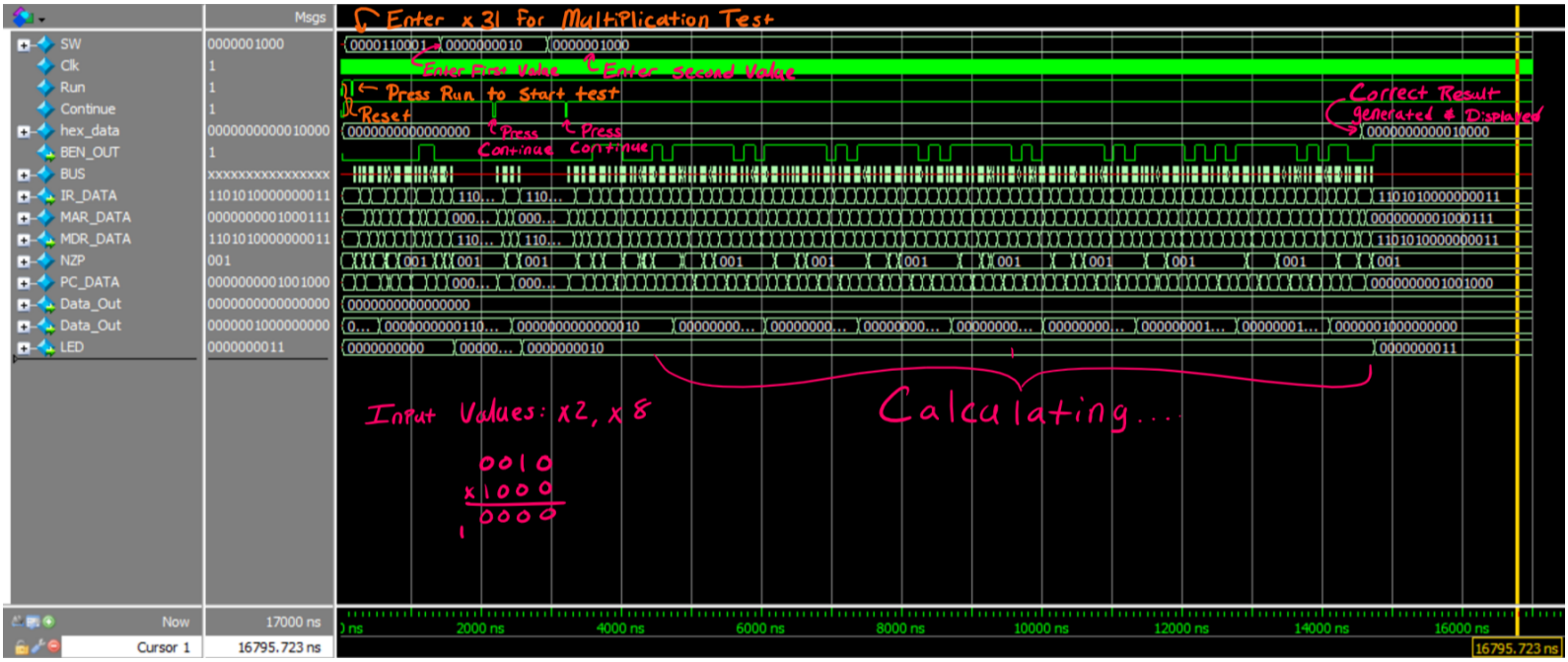
Self Modifying Code Test



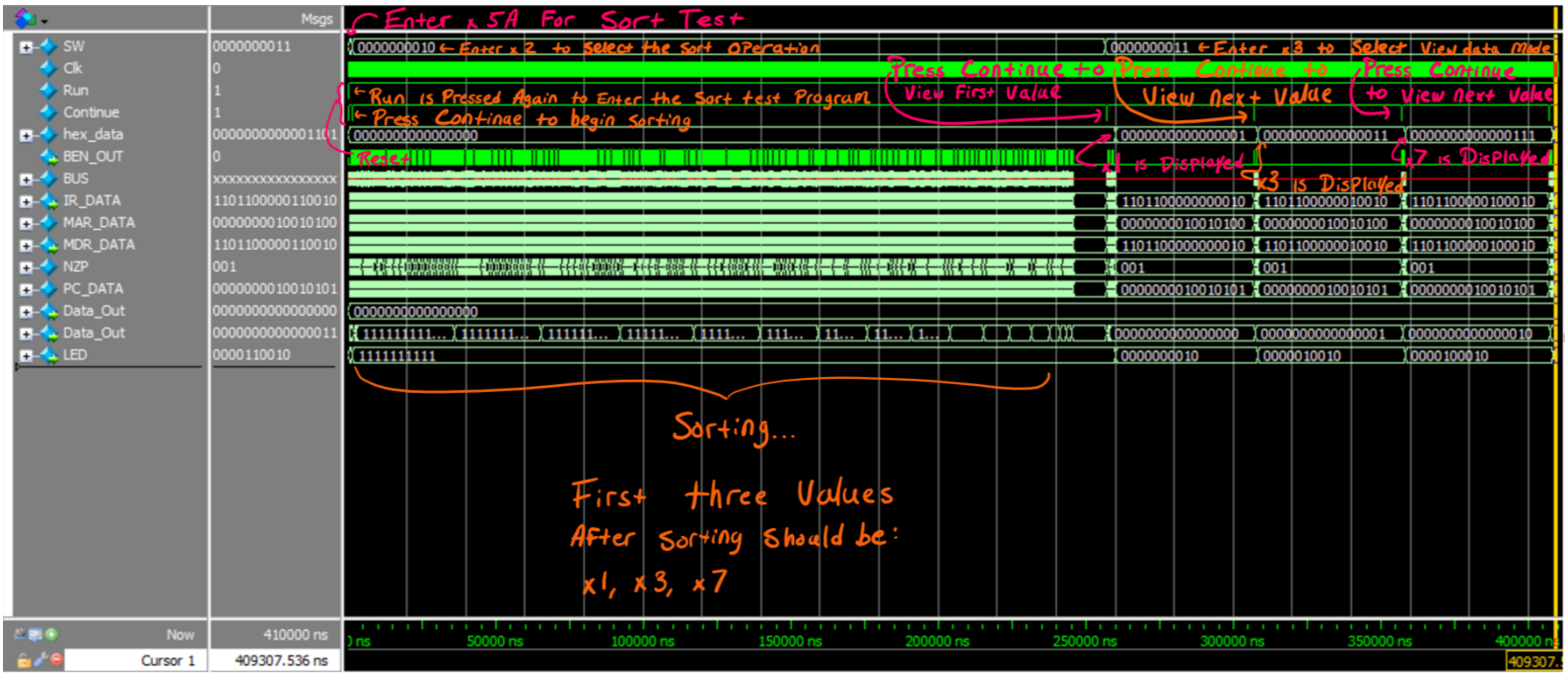
XOR Test



Multiplication Test



Sort Test



4) Post-Lab Questions

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

Design Statistics	
LUT (look up tables)	1113
DSP (Digital signal processors	0
Memory(BRAM)	18,432bits
Flip Flop	1,143
Frequency	95.55MHz
Static Power	89.95mW
Dynamic Power	1.89mW
Total Power	100.68mW

Document any problems you encountered and your solutions to them, and a short conclusion.

One difficulty with this lab was the implementation of the switch signals for our testbench. When trying to simulate our basic I/O test #1, we were setting the correct sequence of buttons and switches, but we were not getting the correct output. After checking over our simulation signals countless times, we realized that although our testbench signals were correct, we were not providing enough delay in between them to give the program enough time to complete the needed operations to generate a correct output. One reason this error was more difficult to detect than it should have been is that we were using the same switch value for our first and last inputs. This caused the output to seem to never update, as the first and last switch inputs were the only signals that had enough time to execute. From the error we learned to ensure that we provide enough delay between our inputs, and also to utilize different input values to make troubleshooting easier.

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

MEM2_IO is used to connect signals from the CPU to Memory and vice versa. If the memory address is xFFFF (-1), the MEM_2_IO will read from the switches. If the address is not xFFFF, the MEM_2_IO will read from the memory. Depending on the value of the Write Enable signal, MEM2IO also controls the direction of data flow to/from the memory and CPU.

What is the difference between BR and JMP instructions?

Branch**(BR)** is an operation that allows the user to “branch” or change the PC depending on the condition codes. If the Condition provided in the branch instruction matches with the current values in the condition registers, the PC will be offset by a PC offset of 9-bit sign extended value. On the other hand, Jump**(JMP)** is an operation that allows the user to “jump” or change the PC independently from the conditional codes. Jump will ALWAYS force the value of the PC to take the value in the base register.

The two big differences between the BR and JMP operations are
(1) conditional codes vs no conditional codes, respectively
(2) 9-bit offset vs setting the PC to the value of the base register, respectively

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?

In Patt and Patel, the R signal indicates that the memory is ready for a read/write operation. It is used for operations pertaining to the loading and storing in memory to ensure the memory is ready to perform the read or write operation. In SLC-3, our memory does not have a ready signal. To ensure that the memory is ready to perform a read or write operation, we simply added two more states to each state containing a memory operation to implement a delay. This delay ensures that the memory always has enough time to perform its operations. By compensation for the lack of the Ready signal with a delay, our system remains synchronous and we do not need to handle any asynchronous signals such as the normal ready signal in LC-3..

5) Conclusions

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

The design of both lab 5.1 and 5.2 were straightforward and understandable. Experiment 5.1 eased us into the fetch phase and helped us create the decoder muxes needed in the datapath. Experiment 5.2 led us to complete the decode and execute phase. At first, we were given a blank slate. We understood the mechanics for each module and knew how to utilize ModelSim to best debug our circuit. Our knowledge in using the ModelSim tools allowed us to save many hours troubleshooting our design and ultimately could get it to function properly.

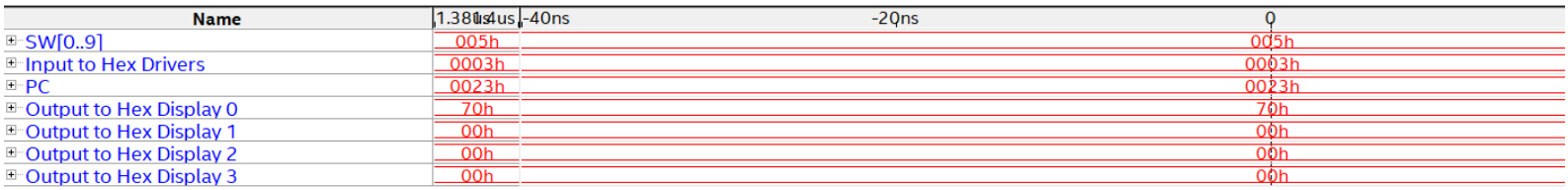
Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.

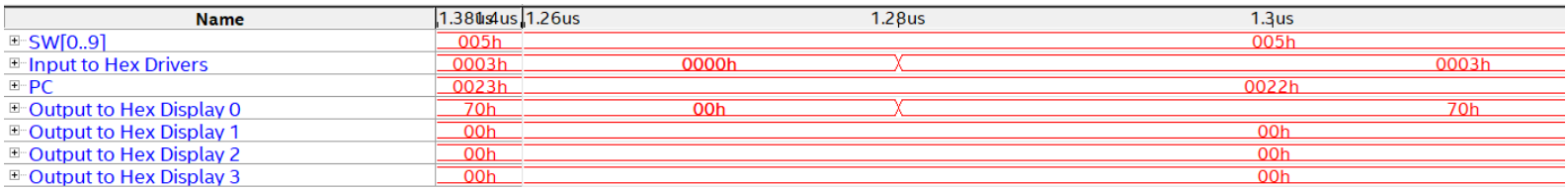
One difficulty that we faced related to the provided material was related to the implementation of extra memory delay states. When we began testing our program on the FPGA, we could not get the Basic I/O test #1 to work properly even though it worked in simulation. In order to troubleshoot the error, we used signal tap to view signals on the FPGA and compare them to the signals in ModelSim. When we viewed the signals on the FPGA, we could see that our instruction register was getting loaded with incorrect values and causing our program to crash. We deduced that the only way for this to happen was if the memory was not ready when the value was loaded into the instruction register. Once we changed the given code to implement a third state for State 33 the test worked properly.

Extra Credit

XOR Test

The XOR test begins at PC = x001A. The Switch values used in this instance were x0006 and x0005. The correct value of the XOR operation is first seen when the PC = x0022. From this we can calculate that there were x0023 - x001A = 9 instructions. By syncing the time bar with our 50MHz clock signal, we can see an accurate representation of how long it took to execute our instructions. In this case, we can also see that this result was computed in 1.3us. By dividing these two terms as (number of instructions)/(Computation Time) we get a result of 6.92E6 Instructions per second, or 6.92MIPS

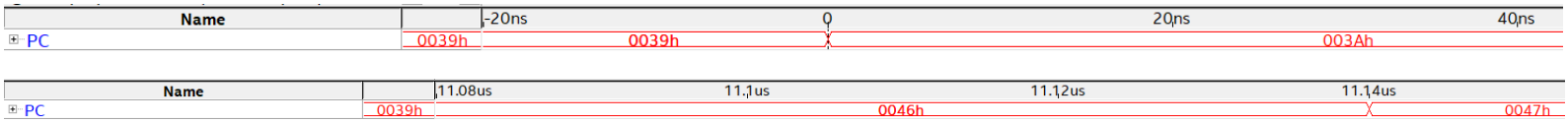




Multiplication Test

The Multiplication test begins at x003A and the output is first generated at x0047. The number of instructions for the Multiplication test is then x0047 - x003A = D or #13 instructions. However, in the Multiplication Test there is a branch condition that causes the program to repeatedly cycle through all but the final instruction multiple times in order to implement multiplication. By analyzing the SignalTap, we can see the PC values x003A through x0046 are repeated eight times. The total number of instructions is then (13 Instructions)(8 Cycles) = 104 Instructions. We can also see that the PC reached x0047 at 11.14us and therefore completed 104 instructions in that time The efficiency of the SLC-3 for the Multiplication test is then (104 Instruction/11.14us) = 9.336MIPS.

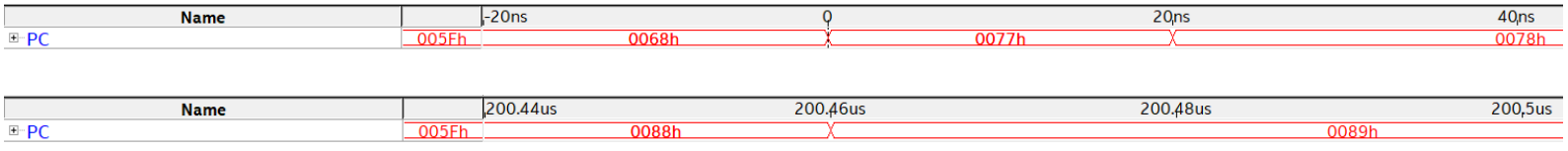
*Note that In this signalTap analysis we were only able to record the PC value due to storage limitations on the DE-10.



Sort Test

The Sort Test begins at x0077 and finishes at x0089. The number of instructions for the sort test is then: x0089 - x0077 = x12 or #18 instructions. The Sort Test implements a bubble sort algorithm which implies that these 18 instructions much be executed N*(N-1) times, where N is the number of items to be sorted. In this case, N = 16 so the total number of instructions can be calculated as (18 Instructions)*16*15 = 4320 Instructions. We can also see that the Sort Algorithm reached its end PC value of x0089 at 200.46us. The performance of the SLC-3 can then be calculated as (4320 Instructions) / (200.46us) = 21.55MIPS

*Note that In this signalTap analysis we were only able to record the PC value due to storage limitations on the DE-10.



Averages

Using the performance test above, we can calculate the average performance of the SLC-3: (6.92 + 9.336 + 21.55)MIPS / 3 = 12.602MIPS