ECE 385

Spring 2023

Experiment #5

Simple Computer SLC-3.2 in SystemVerilog

Dylan Bautista, Matthew Wei

Lab Section NL/ Feb 27 11:00 AM

Nick Lu

Experiment #5

Simple Computer SLC-3.2 in SystemVerilog

I. Introduction/Purpose of Circuit:

The SLC-3 processor is a 16-bit microprocessor modeled after the LC-3 hardware design with 16-bit registers and instructions. These 16-Bit instructions are contained within memory written into a .sv file, which can be read by the microprocessor, from which the intended operation can be decoded and executed. These operations dictated by opcodes use and may alter the values within the 8 internal 16-bit registers, as well as the program counter. This execution phase may use the external switches for input, as well as the hex displays and LED's for output. Each time an operation is conducted, and result displayed, the next instruction can commence to be fetched immediately, or once the continue signal is turned off and on again if paused. The reset signal will trigger the SLC-3 top enter a halted state, resetting all internal registers and resetting the program counter to 0.

II. Written description of Circuit

Summary of Operation

The slc3 file contains modules of the slc3 and the main data path. This slc3 file instantiates major top-level entities like the ISDU, MEM2IO, and data path, while connecting all required internal connections as well as switch and output displays. The data path is in charge of linking many of the register and combinational logic units together within the SLC-3. During the execution process, data from the SRAM, using instantiated on-chip memory, is sent to the ISDU to be interpreted based on a variety of preset states. These states dictate the internal connection signals which will be sent back to the slc3 module and thus control where the flow of data is

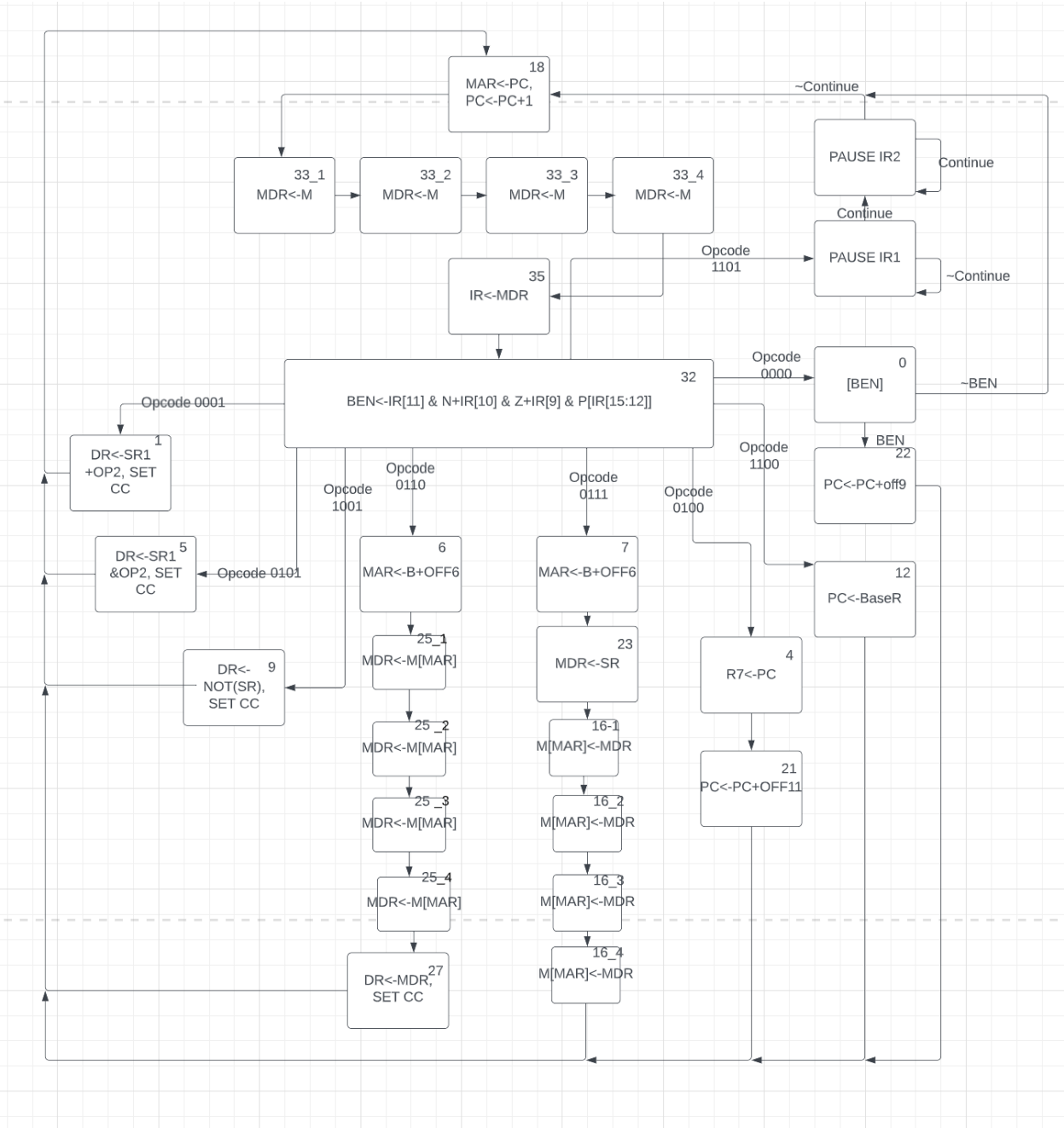
throughout the data path. The MEM2IO system manages I/O between the switches, CPU, and physical on-chip memory, allowing data to be written or read from a predetermined memory address.

The SLC-3 goes through the Fetch-Decode-Execute cycle every time an instruction is conducted in the processor. The Fetch portion gathers the current instruction stored in memory by accessing the memory at the PC value using MAR and MDR, then moving to the next instruction. Decoding involves moving the instruction data from the instruction register to the ISDU. The execute step occurs when the data is read from the ISDU module and values in the data path and slc3 are set accordingly to complete the action. The processor can perform ADD to store the sum of two registers in another register, ADDi to store the sum of a register and a sign extended bit value in another register, AND to store the AND of two registers in another register, ANDi to store the AND of a register and a sign extended bit value in another register, NOT to store the negation of a register in another register, and LDR to load a register with contents at (BaseR+SEXT(offset6)). These all set the status register. There is also BR to branch to the sum of sign extended bit value and PC if condition codes match, JMP to jump to location in a register, JSR to store PC in R7 and add a sign extended bit value to PC, STR to store a register's contents at (BaseR+SEXT(offset6)), and PAUSE to pause next fetch until continue is off and on again and LEDs display ledVect values.

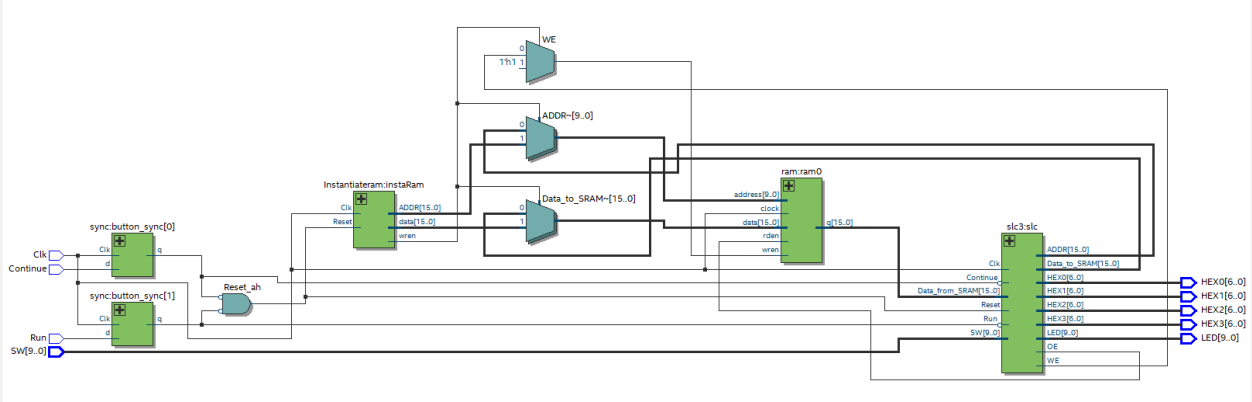
The ISDU is in charge of decoding the 16-bit instruction which is sent in from the instruction register through the linking within the slc3 module. It takes this 16-bit input split into different sections, as well as the BEN value in order to dictate what state the system is in. The fetch states occur regardless of instruction values, but afterwards the opcodes determine what

stae the machine enters into. These states are assigned certain values that control and gate the flow of data within the SLC3. The following is a state diagram of the ISDU.

State Diagram of ISDU



III. Block Diagram:



RTL Block Diagram of SRAMTop

IV. Modules and Simulation:

Modules

addr1_mux.sv

Inputs: [15:0] PC, SR1OUT, S

Outputs: [15:0] Q

Description: Multiplexer to select between two inputs PC and SR1.

Purpose: Used to select the correct value of ADDR1 to be used in the design.

addr2_mux.sv

Inputs: [15:0] IR, [1:0] S

Outputs: [15:0] Q

Description: Multiplexer to select which sign extended bits of IR should be used as ADDR2.

Purpose: Used to select the correct value of ADDR2 to be used in the design.

alu.sv

Inputs: [15:0] A, B, [1:0] S

Outputs: [15:0] Q

Description: Arithmetic unit that incorporates ADD, NOT, and AND.

Purpose: Performs the needed operations between SR1 and SR2 decided by the ALUK select. Output is sent to the bus.

ben.sv

Inputs: [15:0] nzp, IR, LD_BEN, Clk

Outputs: [2:0] BEN

Description: From the input, if the nzp value in the branch instruction matches the actual nzp value from the input (IR).

Purpose: Determines if the branch condition nzp is statisfied.

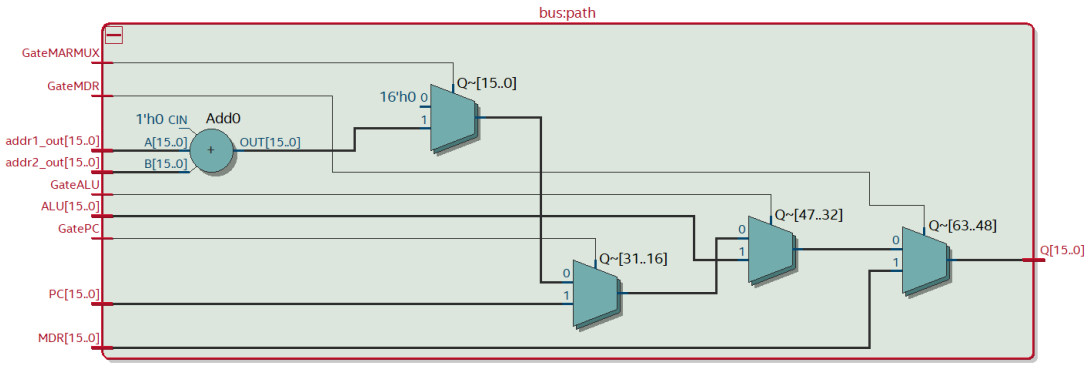
bus.sv

Inputs: logic GateMDR, GateALU, GatePC, GateMARMUX, [15:0] addr1_out, addr2_out, [15:0] MAR, MDR, PC, ALU

Outputs: [15:0] Q

Description: Holds a value that is used for the datapath. This value is transmitted amonst all the other modules. Actual code has code determining which value should be used.

Purpose: Decides whether the value should be MDR, PC, MAR, or ALU.



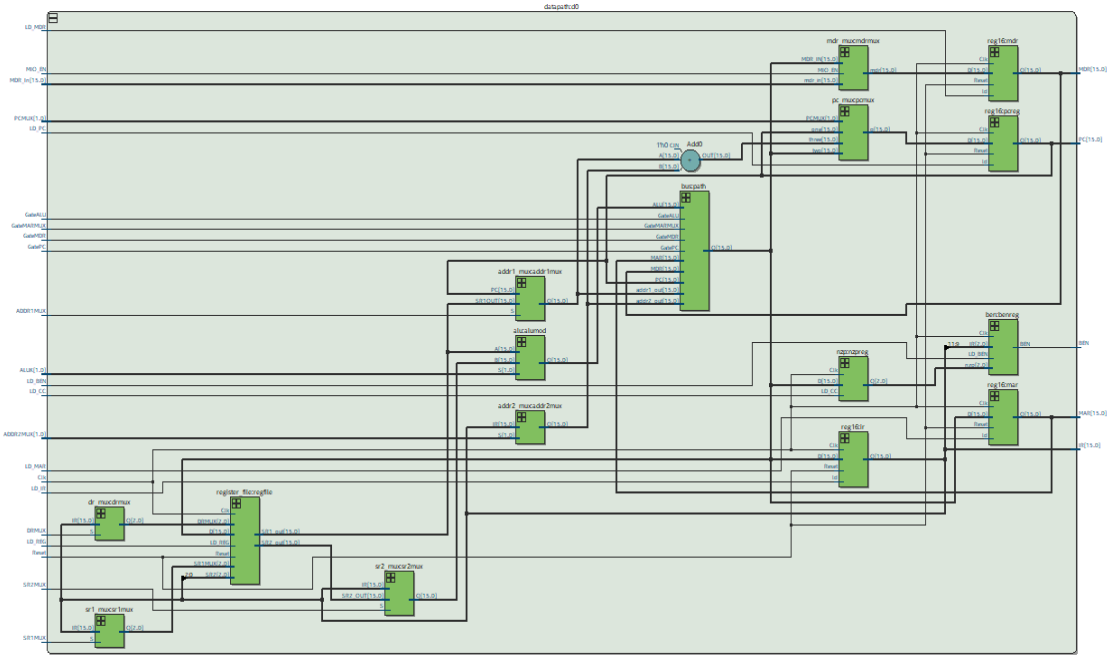
datapath.sv

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

Outputs: BEN, [15:0] MAR, MDR, IR, PC

Description: The datapath essentially is the Patt and Patel diagram. It connects the inputs and outputs of each module to each other so that the design can work properly. Code was made basically looking at the diagram and wiring each input/output to their corresponding locations.

Purpose: Connects all the muxs, registers, register file, bus, ALU, and other components with each other.



dr_mux.sv

Inputs: [15:0] IR, S

Outputs: [2:0] Q

Description: Multiplexer to select whether IR[11:9] or 111 should be used as DR.

Purpose: Used to select the correct value of DR to be used in the design.

HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: Module that holds information about the Hex LED displays on the FPGA board.

Purpose: Used so the correct output can be displayed onto the FPGAHEX displays.

Instantiateram.sv

Inputs: Clk, Reset

Outputs: [15:0] ADDR, wren, [15:0] data

Description: Contains code that will load the contents of memory_contents onto the FPGA board so that the on-chip memory can work.

Purpose: Instantiates the RAM memory contents onto the FPGA board.

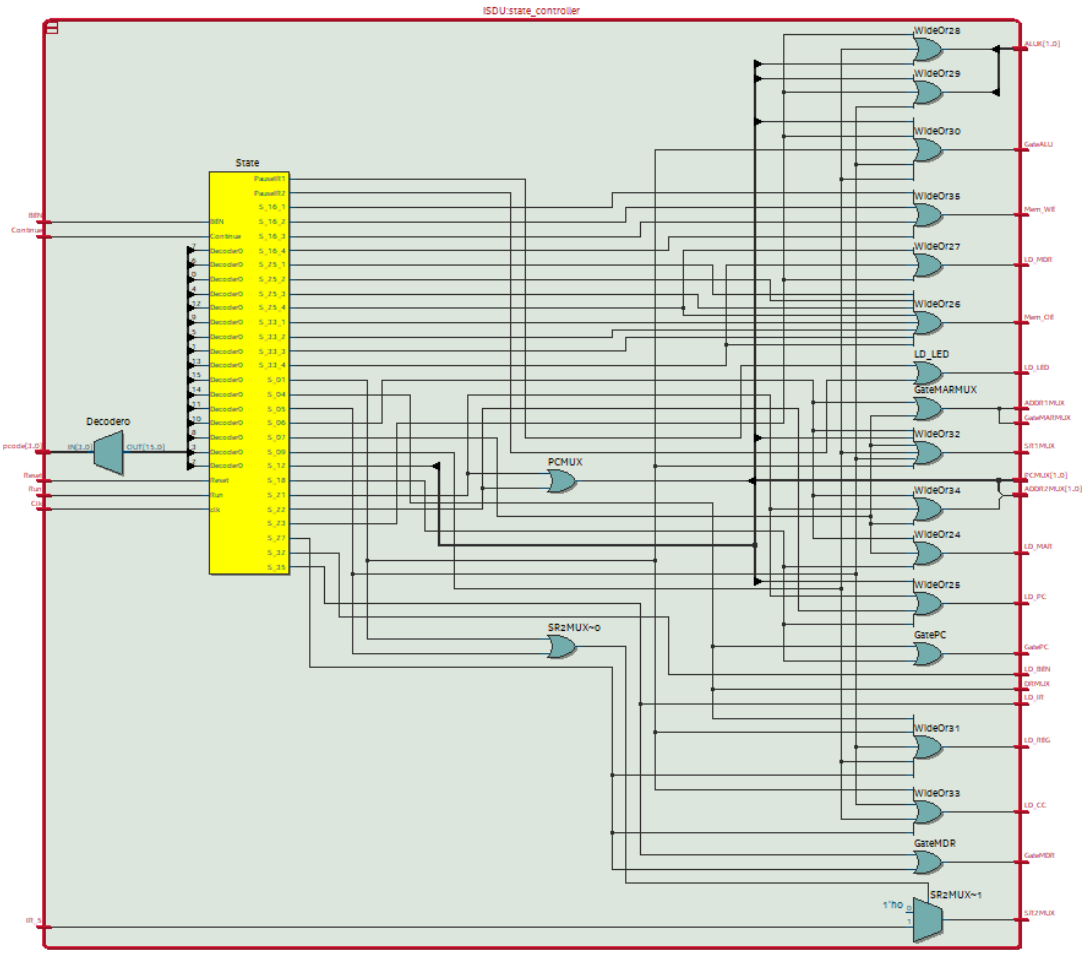
ISDU.sv

Inputs: Clk, Reset, Run, Continue [3:0] Opcode, 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, [1:0] PCMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, [1:0] ADDR2MUX, ALUK, Mem_OE, Mem_WE

Description: Contains the control unit of the design. Essentially, the Patt and Patel control flow with all the states is incorporated into this one module. It guides the whole design and allows it to work the way it does.

Purpose: Acts as the control unit of the design. It tells which signals should be high or low and which signals should have which specific values depending on which state it is currently at.



mdr_mux.sv

Inputs: [15:0] MDR_IN, mdr_in, MIO_EN

Outputs: [15:0] mdr

Description: Multiplexer to select what value MDR should be equal to.

Purpose: Used to select the correct value of MDR to be used in the design.

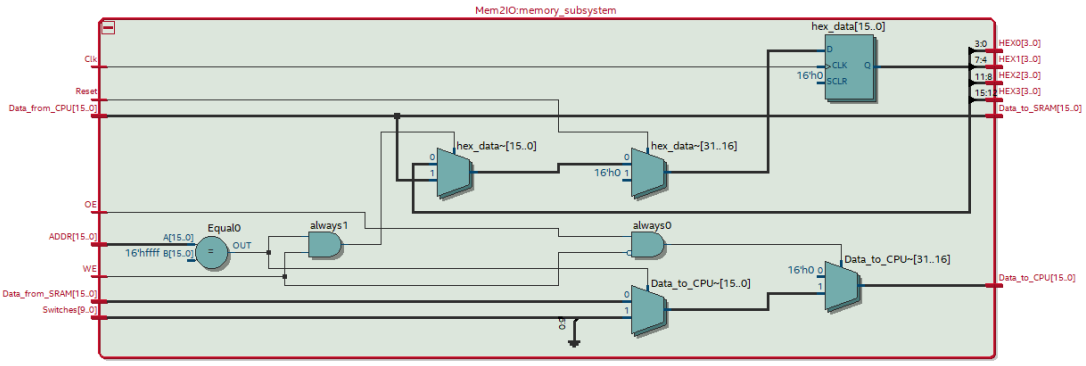
Mem2IO.sv

Inputs: Clk, Reset, [15:0] ADDR, OE, WE, [9:0] Switches, [15:0] Data_from_CPU, Data_from_SRAM,

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

Description: Contains code that determines what to load from the switches and to the LEDs and HexDisplays. Does this by reading from switches if address is xFFFF. If WE is high and address is at xFFFF, it will instead write to the LEDs and HexDisplays.

Purpose: Handles the I/O of the FPGA board by reading from the switches or writing to the LEDs and HexDisplays.



memory_contents.sv

Inputs: None

Outputs: None

Description: Contains initial values of memory to be loaded onto the FPGA board for test programs.

Purpose: Holds and initializes the data needed to run the test programs on the FPGA board.

muxs.sv

Inputs: [1:0] S, [15:0] D1, D2, D3

Outputs: [15:0] Q

Description: Contains all the varying multiplexers needed to implement the SLC-3 design. Includes the mdr_mux, pc_mux, sr1_mux, dr_mux, sr2_mux, addr1_mux, and addr2_mux.

Purpose: Multiplexers select which input is required to go to the output. The varying outputs in this file are routed to multiple areas to be used such as the ALU, register file, datapath, and the bus.

nzp.sv

Inputs: [15:0] D, LD_CC, Clk

Outputs: [2:0] Q

Description: From the input, determines if the value is negative, zero, or positive and sets the nzp value respectivley.

Purpose: Determines the 3-bit value of nzp from the input to be used for branch instructions.

pc_mux.sv

Inputs: [15:0] one, two, three, [1:0] PCMUX

Outputs: [15:0] q

Description: Multiplexer to select which of the three possible value PC should be set to. (Increment, nothing, or ADDR1 + ADDR2).

Purpose: Used to select the correct value of PC to be used in the design.

reg16.sv

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

Outputs: [15:0] Q

Description: Loads a 16-bit register with a input value. Keeps the value the same or changes it depending on the load input. Also has an option to reset the register.

Purpose: Essentially R0-R7. Used later in the register file to create the 8 instances of the LC-3 registers.

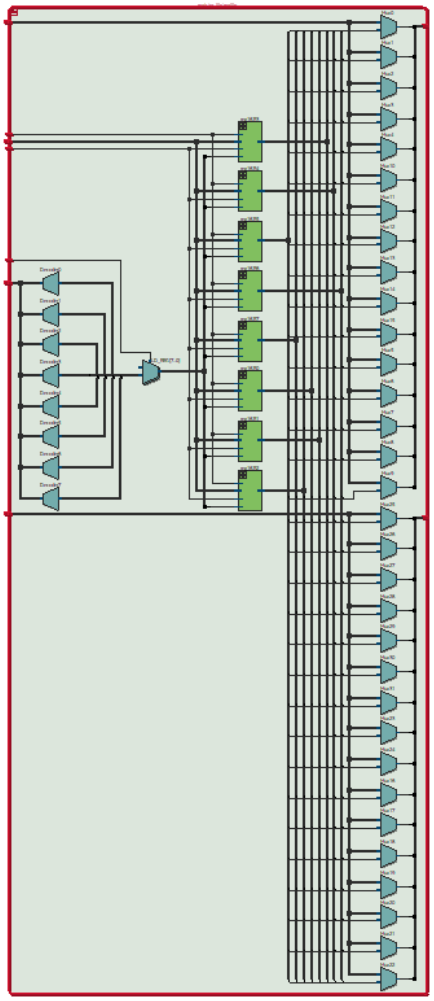
register_file.sv

Inputs: [2:0] DRMUX, SR1MUX, SR2, [15:0] D, LD_REG, Clk, Reset

Outputs: [15:0] SR1_out, SR2_out

Description: Contains and manages the 8 registers in LC-3. Additional inputs select the specific destination and source register as instructions require. The module can manipulate the values within each register as well as assign which register is the source or destination.

Purpose: Manages the data to be stored inside of the 8 registers.



slc3.sv

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

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

Description: Initializes all the modules required for the top-level entity such as the datapath, Mem2IO, and the ISDU as well as assign the HexDisplay and LED values.

Purpose: Acts as the high level design with all the modules needed to incorporate the SLC-3 processor (used in the top-level design later with synchronizers).

SLC3_2.sv

Inputs: None

Outputs: None

Description: Although not a module, it is a file. It contains all the opcode functions and a list of useful parameters.

Purpose: Defines the structure of all the opcodes and how each bit grouping is grouped.

slc3_sramtop.sv

Inputs: [15:0] SW, Clk, Run, Continue

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

Description: Top-level design entity for on-board FPGA syntesis. Requires a different top-level than test simulation.

Purpose: Compiles the code to work properly. Combines all the parts and properly hooks up the inputs and outputs correspondingly. This is the top-level file for FPGA synthesis.

slc3_testtop.sv

Inputs: [15:0] SW, Clk, Run, Continue

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

Description: Top-level design entity for simulation in ModelSim. Requires a different top-level than on-chip FPGA.

Purpose: Compiles the code to work properly. Combines all the parts and properly hooks up the inputs and outputs correspondingly. This is the top-level file for simulations.

sr1_mux.sv

Inputs: [15:0] IR, S

Outputs: [15:0] Q

Description: Multiplexer to select which bits of IR should be used as SR1.

Purpose: Used to select the correct value of SR1 to be used in the design.

sr2_mux.sv

Inputs: [15:0] IR, SR2_OUT, S

Outputs: [15:0] Q

Description: Multiplexer to select whether SR2 or IR should be assigned to SR2.

Purpose: Used to select the correct value of SR2 to be used in the design.

synchronizers.sv

Inputs: Clk, Reset, d

Outputs: q

Description: Contains modules that synchronize the code, so it properly works with the FPGA board.

Purpose: Synchronizes the inputs on the board with the code so the FPGA can correctly be used to take in inputs for the code.

test_memory.sv

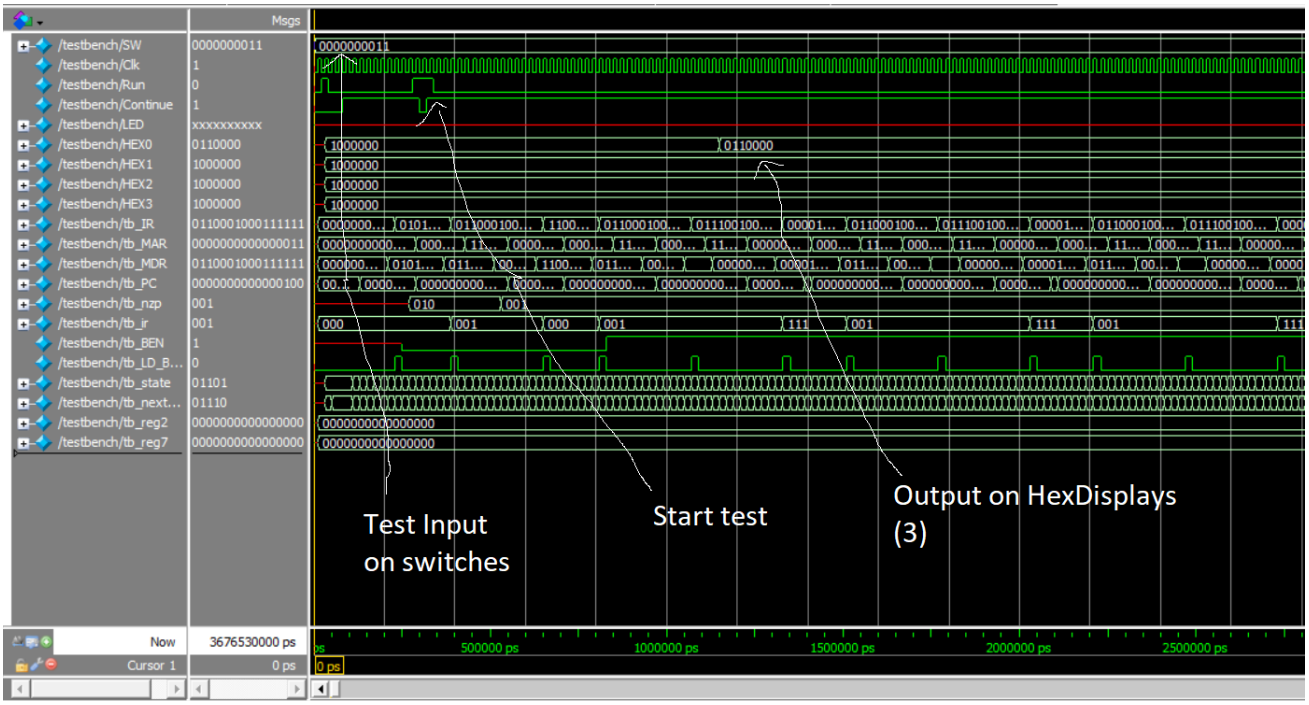
Inputs: Reset, Clk, [15:0] data, [9:0] address, rden, wren

Outputs: [15:0] readout

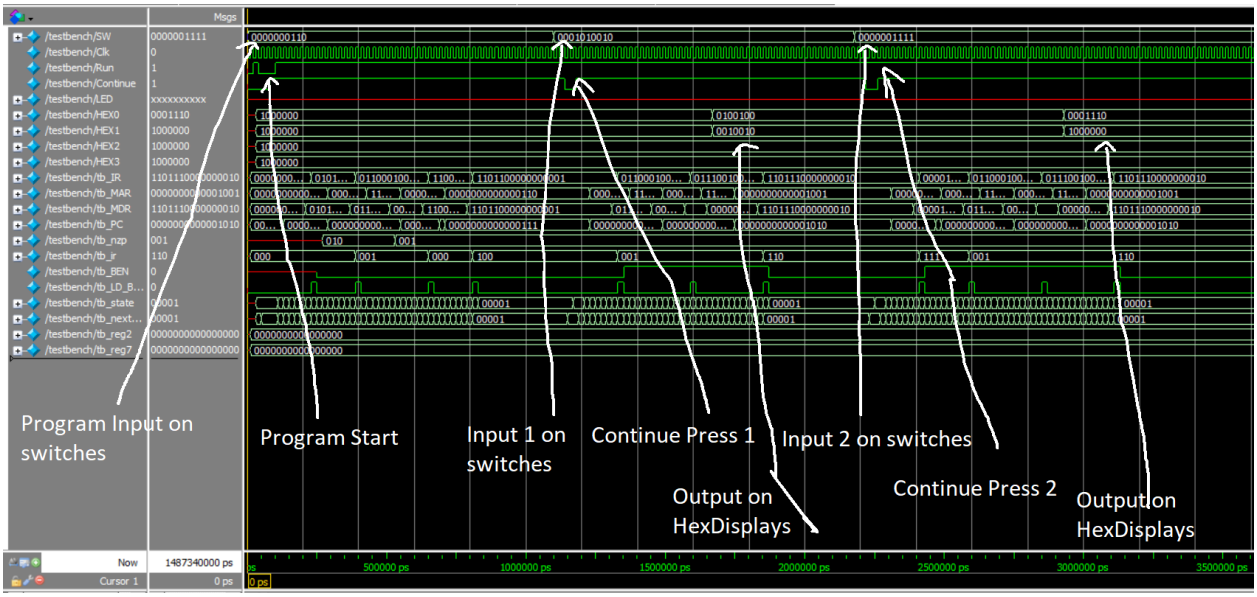
Description: Memory to be used for test simulations when not compiling to the FPGA board.

Purpose: Used for debugging or for ModelSim simulations (not for on-board testing).

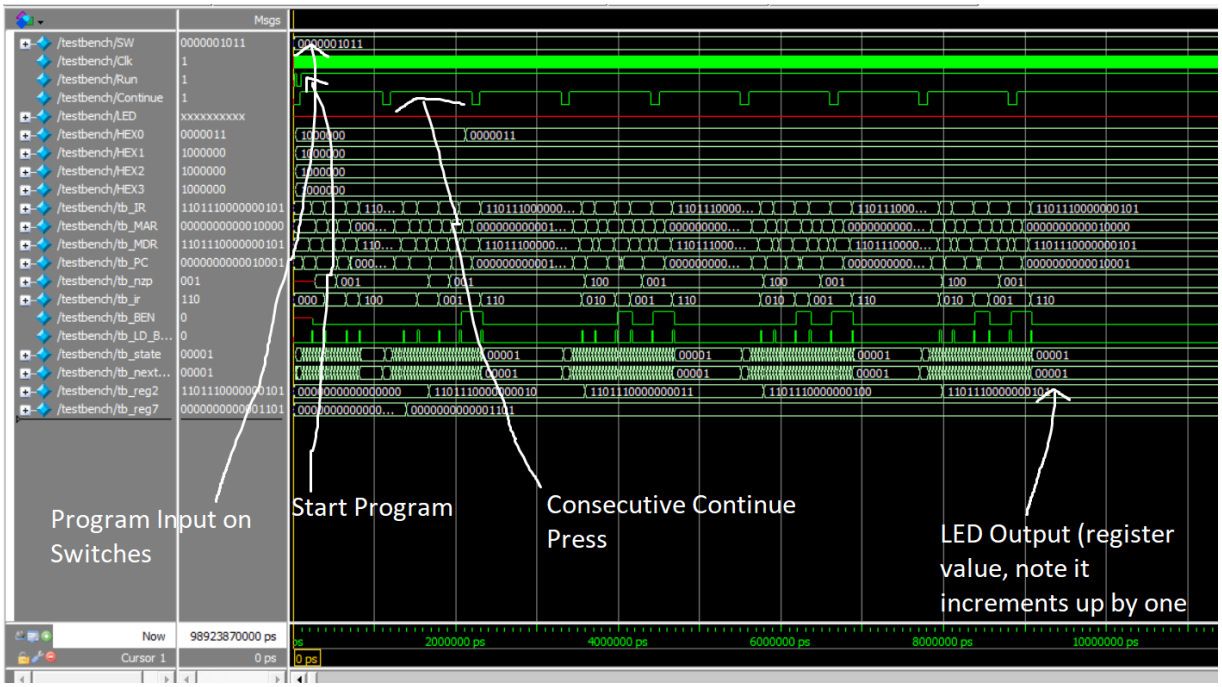
Simulation



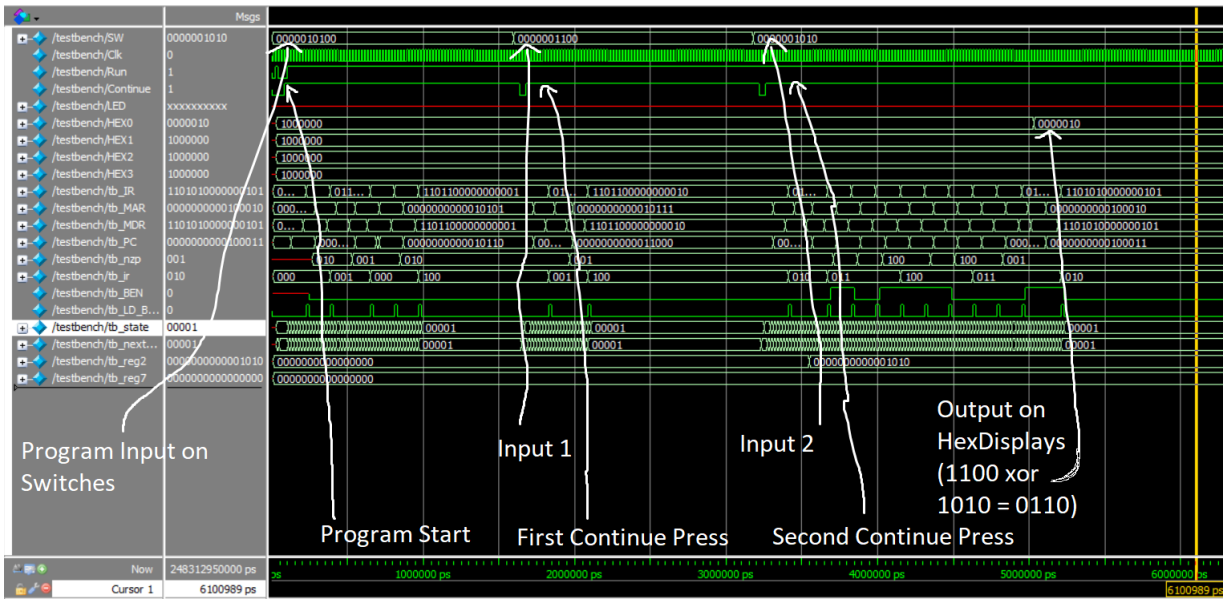
Basic I/O 1 Simulation Results



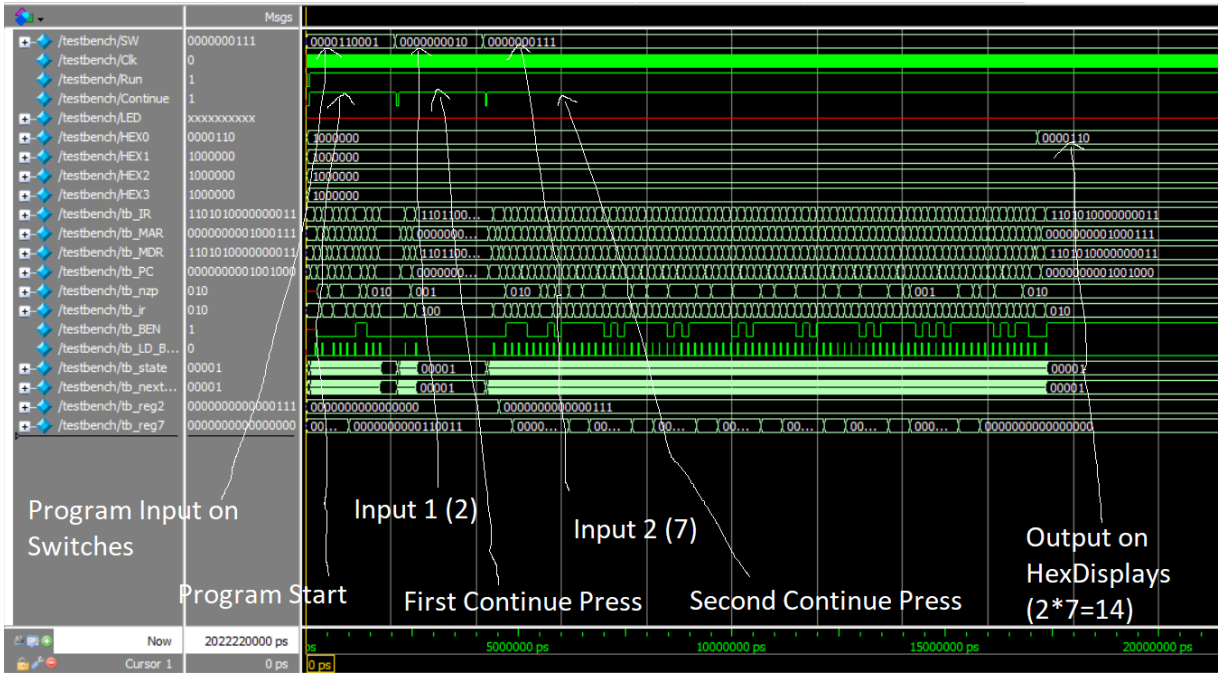
Basic I/O 2 Simulation Results



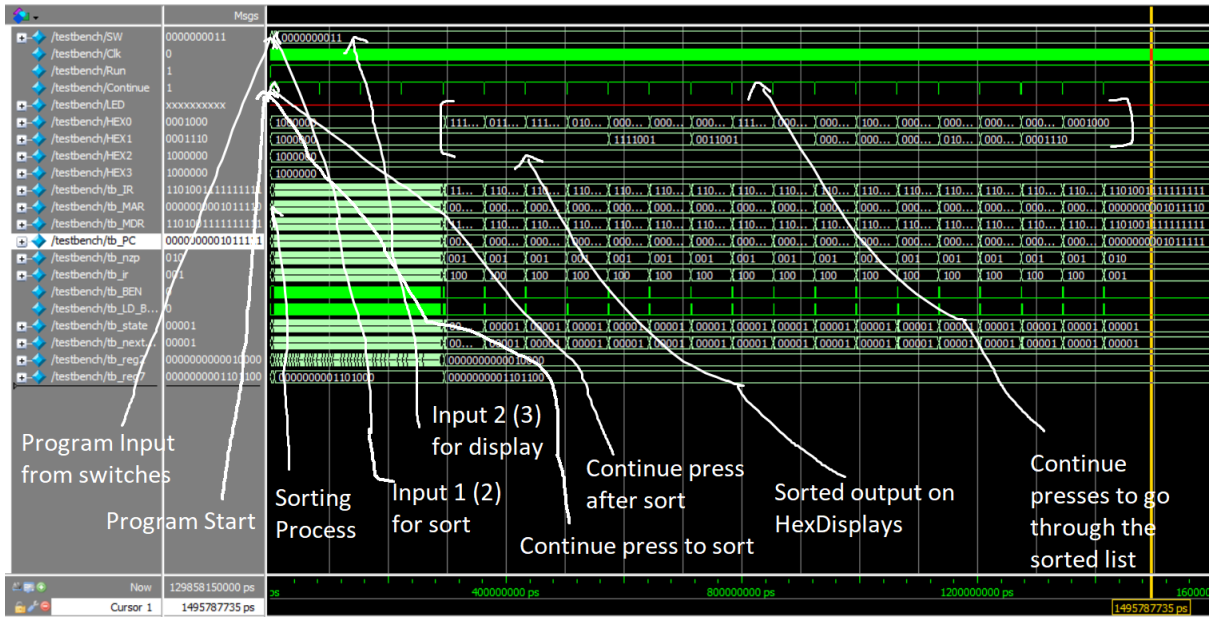
Self-Modifying Code Simulation Results



XOR Simulation Results



Multiplication Simulation Results

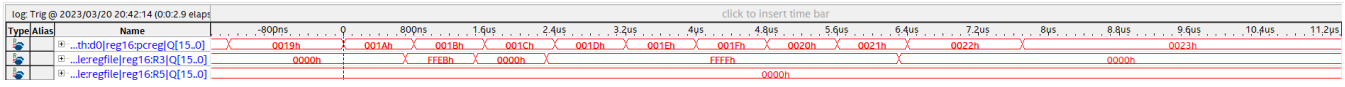


Sort Simulation Results

V. Extra Credit/MIPS

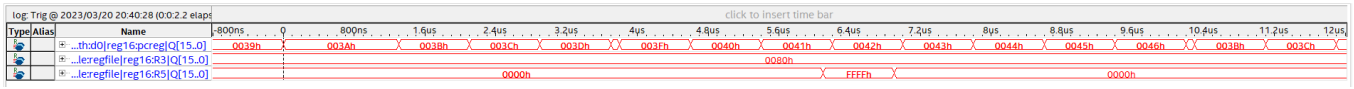
For all calculations below, frequency is 50 MHz and one clock cycle is 20 ns. Inputs for XOR are 0x0014 for the program input and both value inputs. Inputs for multiplication are 0x0031 for the program input and for both value inputs. Inputs for sort are 0x005A for program input and 0x0002 to sort the list. All instruction estimates where calculated through math (whether simple or more complex for multiplication and sort).

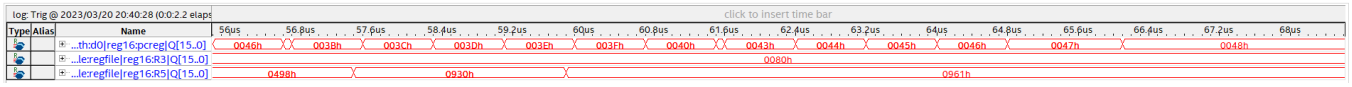
XOR



Instruction of XOR ends around x0022 and starts at x001A, so XOR takes x0022-x001A which is 8 instructions. The trace says it takes about 7.5 us to execute all 8 instructions. 7.5us/20ns is 375 cycles. 8/356*50MHz gives 1.124 MIPs.

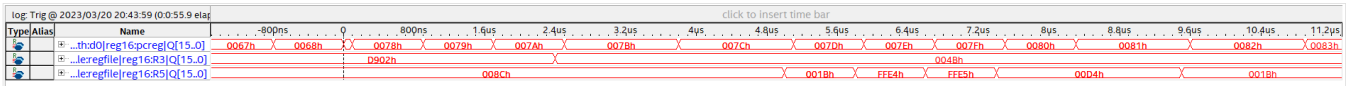
Multiplication

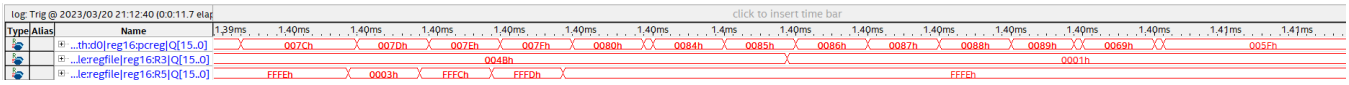




The SignalTap trace is too long for multiplication, so only the beginning and ends are included. The signal trace of multiplication ends around x0047 and starts at x003A. Since there are branches, some instruction need to be repeated so another method of counting instructions was required. We wrote a python script that counted how many times the PC value changed from the traces initial starting program to estimate the instruction number. With the script, we got that 97 instructions were executed, and the trace says it took 66.1 us to complete the operation. Following the same formula, 66.1us/20ns = 3305 cycles and 97/3305*50MHz gives 1.467 MIPs.

Sort





The SignalTap trace for sort is also too long to be shown, so only the beginning and ends are included. The signal trace of sort ends around x0089 and starts at x0077. The instruction estimate was got the same method as multiplication. In total, 1,632 instructions were executed, and the trace says it took 1.4087 ms to complete the operation. Following the same formula as before, 1.4087ms/20ns = 70435 cycles and 1632/70435*50MHz gives 1.157 MIPs.

Average

The average of the MIPs is (1.124 + 1.467 + 1.157)/3 which gives 1.25 MIPs.

VI. Conclusion

Overall, this lab was interesting as a whole processor execution cycle was designed and implemented. This lab was a big jump from making adders and multipliers and really showed how SystemVerilog could be used to make more functional and complex hardware. Introduction of ram initialization and other new topics really made this lab a lot to wrap one's head around. Making the lab's modules and connecting them wasn't too complex as the SLC-3 diagram was given. All we had to do was code the modules and connect each input and output respectively. The hard part of the lab for us was understanding the previously written modules and how to properly create a datapath for the first checkpoint. However, the rest of the lab was relatively simple as most of it involved just following the diagrams and debugging what we wrote. One implementation that we did get stuck on was the LED implementation. For one of the programs, we thought the LED meant the Hex displays and spent quite a while trying to figure out what was wrong with our code. After we realized they were different, we were able to implement it fine.

Regarding functional bugs and issues that arose, there were quite a few. A lot of times our code gave a compile error about hierarchy. This involved us changing the combinational logic of the modules with problems as there was always some error in our code that assumed sequential logic. Other bugs we had showed up after the code compiled. One bug was that our code never repeated indefinitely with the first program. The issues were that we connected the wrong bit lengths to the BR and BEN inputs and outputs as well as forgetting to include some

code in the IDSU. Another bug we encountered was incorrect loading and storing of values. This was due to forgetting to get certain bit values high or low in the IDSU as well.

Post-Lab Questions

	Processor
LUT	946
DSP	0
Memory (BRAM)	18432
Flip-Flip	261
Frequency (MHz)	69.37
Static Power (mW)	90.01
Dynamic Power (mW)	13.18
Total Power (mW)	114.54

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

MEM2IO is used to communicate the inputs and outputs with the CPU. It reads the inputs from switches when the address is xFFFF or it outputs to the Hexdisplays when WE is high and address is xFFFF. This can be done because the memory in memory contents is written to the FPGA's onchip RAM. Essentially, MEM2IO is used to determine the inputs and outputs of the CPU from and to the FPGA board

What is the difference between BR and JMP instructions?

BR or branch jumps to areas in memory given by the current value of PC. It checks for nzp (negative, zero, positive) and if conditions match, then it jumps with the offset given. JMP or jump doesn't require a check for conditions and can jump using an offset whether set or given in a register. Also, BR can only jump using 9-bit values as (offset section in instruction) and JMP can jump using 16-bit values (offset from register).

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 in Patt and Patel tells the CPU whether a read or write operation has been completed. The design in our implementation doesn't utilize this and instead, we compensate it with the inclusion of extra states in the IDSU to make up for the time it would have needed. This implies that no synchronization would be needed during this step since no signal would require its use.