ECE 385

Spring 2023

Lab 5

Simple Computer SLC-3.2 in SystemVerilog

Gustavo Fonseca, Hunter Baisden

TA: Hongshuo Zhang

Introduction

The main functionality of the SLC-3 Processor was to act as a microprocessor that uses a simplified version of LC3. This means that it is a 16-Bit processor with 16-Bit instructions and Registers. These instructions include AND, ADD, NOT, etc.

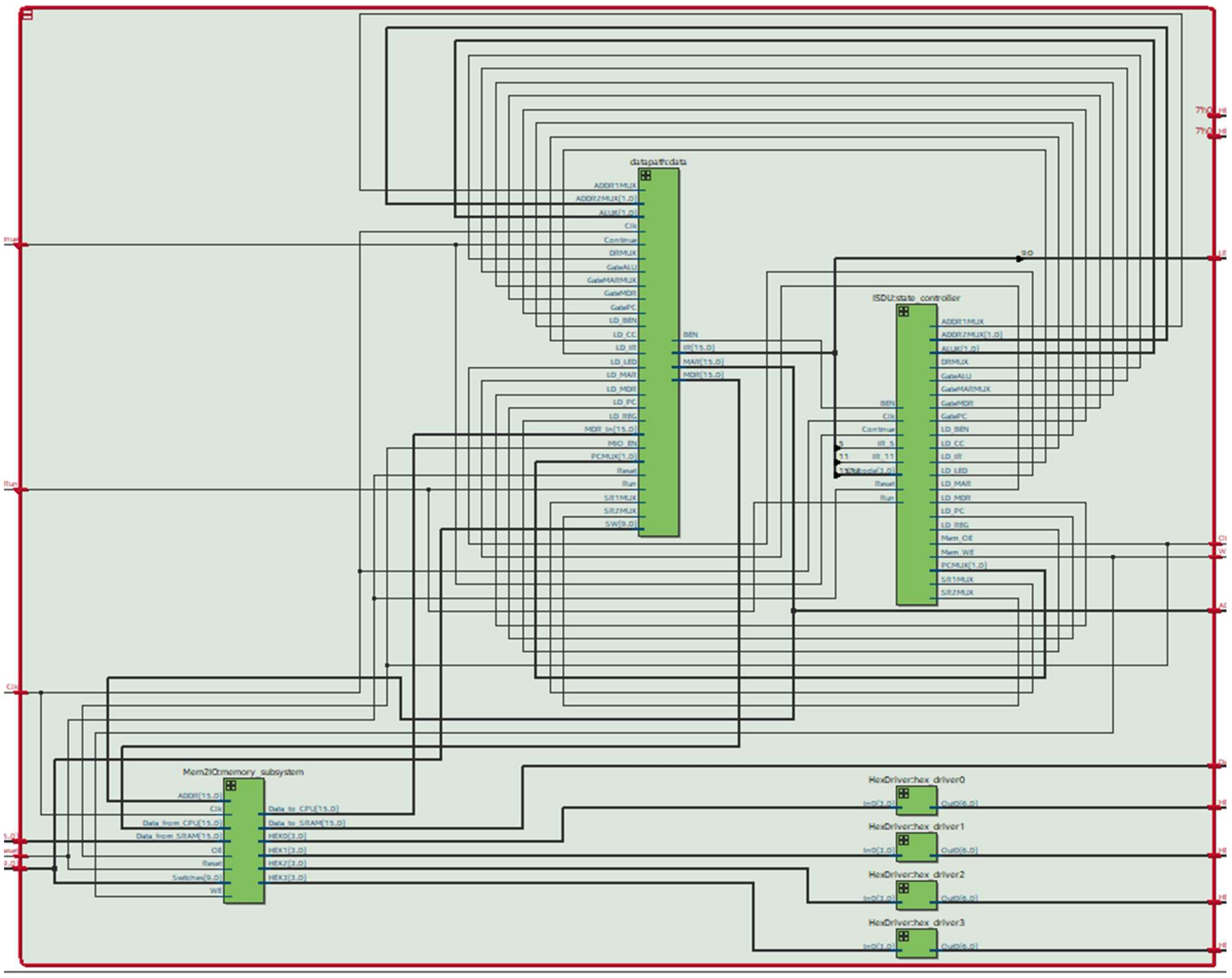
Summary of Operation

1. **Fetch-Decode-Execute**
   The processor operates in 3 main steps: Fetch, Decode, and Execute.
   The fetch step gets the value of the next instruction. It does this by storing value of the Program Counter (PC) in the Memory Address Register (MAR). The value of PC is then incremented by 1. Following this, the value stored within the address of the value stored within MAR is stored within Memory Data Register (MDR). The value within MDR is then stored within the Instruction Register (IR).
   The decode step interprets which instruction should be performed next. The 4 most significant bits (MSB) of the instruction code determines which instruction should be performed. Thus, the 4 MSB of the value stored within IR (IR [15:12]) determines what instruction will be performed. Additionally, the Branch (BR) instruction requires us to define the value of the Branch Enable Condition (BEN). The definition of BEN requires the use of the Status Register (CC), which is a 3-Bit value. This 3-bit value is conventionally called NZP, which is defined by the sign of the value stored into a register in the last instruction: N for negative, Z for zero, and P for positive. The logic controlling BEN is as follows: BEN <= IR[11] & N + IR[10] & Z + IR[9] & P. The value of BEN will always be calculated every decode step but will only be used within the BR instruction.
   The execute step is where the specific instruction is performed. The specifics of each instruction will be discussed below. After the instruction has been executed, the processor will then go back to the fetch step to begin the cycle again.
2. **Specific Instruction Operation**
   a. ADD
   Both ADD and ADDi have the same IR[15:12] value. The processor knows which instruction to perform based on IR[5]. If IR[5] = 0, then ADD is performed. In this case, two registers will be added together. The specific registers are determined by IR[2:0] and IR[8:6] respectively. This value will be store within the register determined by IR[11:9]. The sign of the stored value is then stored within CC (100 if negative, 010 if zero, 001 if positive).
   b. ADDi (Add Immediate)
   If IR[5] = 1, then ADDi is performed. In this case, one register determined by IR[8:6] is added together with the 16-Bit sign-extended value of IR[4:0]. This value will again be stored in the register determined by IR[11:9]. The sign of the stored value is then stored within the CC.
   c. AND
   Both AND and ANDi have the same IR[15:12] value. The processor knows which

instruction to perform based on IR[5]. If IR[5] = 0, then AND is performed. In this case, a bitwise AND operation will be performed between the value of two registers. The specific registers are determined by IR[2:0] and IR[8:6] respectively. This value will be store within the register determined by IR[11:9]. The sign of the stored value is then stored within CC.

d. ANDi (And Immediate)
   If IR[5] = 1, then ANDi is performed. In this case, a bitwise AND operation will be performed between the value of a register and the 16-Bit sign-extended value of IR[4:0]. The specific register is determined by IR[8:6]. This value will again be stored in the register determined by IR[11:9]. The sign of the stored value is then stored within CC.
e. NOT
   This instruction takes the bitwise NOT operation of the value stored within a register, determined by IR[8:6]. This value will be stored within the register determined by IR[11:9]. The sign of the stored value is then stored within CC.
f. BR (Branch)
   This instruction checks the value of BEN. If BEN = 0, then this instruction does nothing, and the processor goes to the fetch state. If BEN = 1, then the value PC is set to PC added with the 16-Bit sign-extended value of IR[8:0].
g. JMP (Jump)
   This instruction takes the value stored within a register and sets PC to that value. The value of the specific register is determined by IR[8:6].
h. JSR (Jump to Subroutine)
   This instruction sets the value of the 7th register to value of PC, then sets the value of PC to PC added with the 16-Bit sign-extended value of IR[10:0].
i. LDR (Load Register)
   This instruction loads a given value into a register. This value is determined by first adding the value of a register and the 16-Bit sign-extended value of IR[5:0]. The specific register is determined by IR[8:6]. We then take the value stored at the address defined by this addition. This value will be stored within the register determined by IR[11:9]. The sign of the stored value is then stored within CC.
j. STR (Store Register)
   This instruction stores the value of a register into memory. The address to be modified is determined by first adding the value of a register and the 16-Bit sign-extended value of IR[5:0]. The specific register is determined by IR[8:6]. The value of a register determined by IR[11:9] is then stored into the calculated address.
k. PAUSE
   This instruction pauses the processor until the user presses the continue button. This instruction also sets the FPGA's LEDs to the value of IR[11:0].

Block Diagram of slc3.sv



Above is the block diagram of the slc3 module. This module has 3 key submodules, all present above: the Datapath, the Mem2IO, and the ISDU. Together, these 3 modules make up the entirety of the slc3 along with the hex drivers for displaying on the FPGA.

.SV Modules

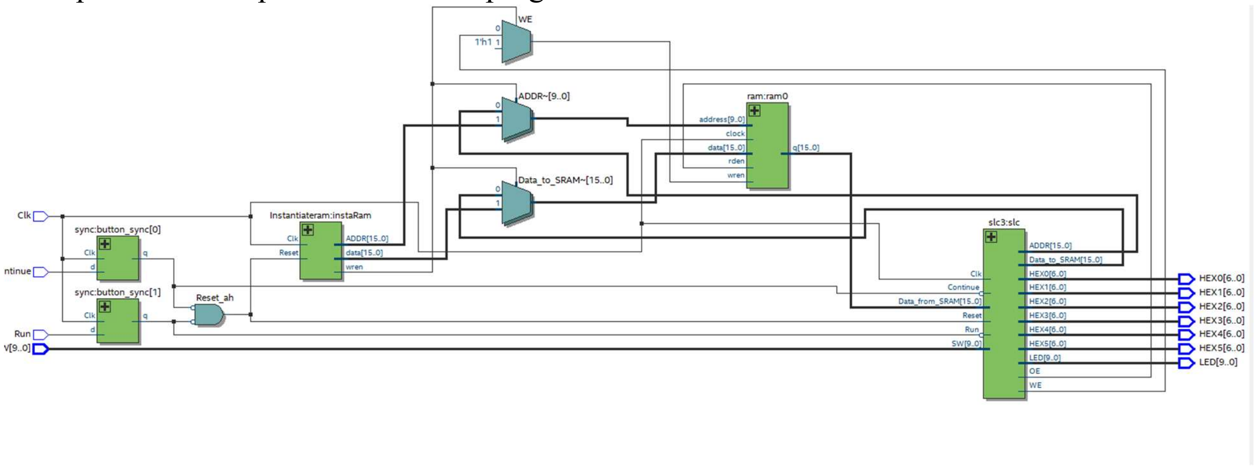
Module: slc_sramtop (in slc_ramtop.sv)

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

Outputs: [6:0] HEX(0-5), [9:0] LED

Description: This module takes in the switches through SW and uses them to select a point in memory to start execution of the slc3. It outputs onto the hex and led displays the data processed by the FPGA. This module also instantiates the on-chip RAM.

Purpose: This module serves as the top level for all the interactions with the FPGA. It is used as the top level in compilation when we program the FPGA.



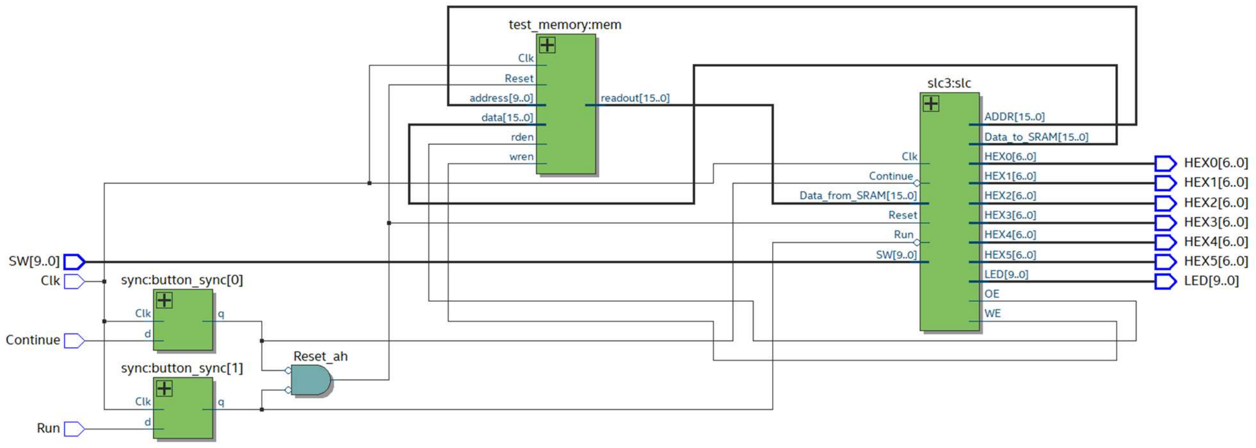
Module: slc_testtop (in slc_testtop.sv)

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

Outputs: [6:0] HEX(0-5), [9:0] LED

Description: This module takes in the switches through SW and uses them to select a point in memory to start execution of the slc3. It outputs onto the hex and led displays the data processed by the FPGA. This module instantiates the test memory that we were given.

Purpose: This module serves as the top level for all the interactions with Model Sim Altera. This module was compiled as the top level when we were testing and debugging with simulations.



Module: slc3 (in slc.sv)

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

Outputs: [6:0] HEX(0-5), [9:0] LED, OE,WE, [15:0] ADDR, [15:0] Data_to_SRAM

Description: This is the next level down from the top level and universal across both the simulation and the FPGA. This module instantiates a Datapath, Mem2IO, and ISDU module to create the entire framework for the slc-3.

Purpose: This module creates the entire hardware framework of the slc3 processor. It lays out the Datapath by calling the Datapath module, the memory by calling the Mem2IO module, and the state controller by calling the ISDU module. This module is shown in the above section.

Module: Instantiateram (in Instantiateram.sv)

Inputs: Clk, [15:0] ADDR, wren

Outputs: [15:0] data

Description: This module is called in the slcramtop top level with inputs Clk, ADDR, and wren and is used to output a 16-bit wide data value. This module does exactly what it states: it instantiates the on-chip RAM on the FPGA.

Purpose: Without an on-chip RAM on the FPGA, running the programs and debugging would be impossible on the FPGA . This module does have an RTL diagram, but it is far too large and complicated to be displayed below this module.

Module: SLC3_2 (in SLC3_2.sv)

Inputs: N/A

Outputs: N/A

Description: This module defines all of the binary values for the opcodes, registers, and nzp values. It also breaks down the IR into the bitwise sections based on the opcode sent in.

Purpose: Without this module, the decode section would fail every time, as it is needed to pull out the different parts of the IR to decode the instruction properly. This sv module does not have a RTL diagram.

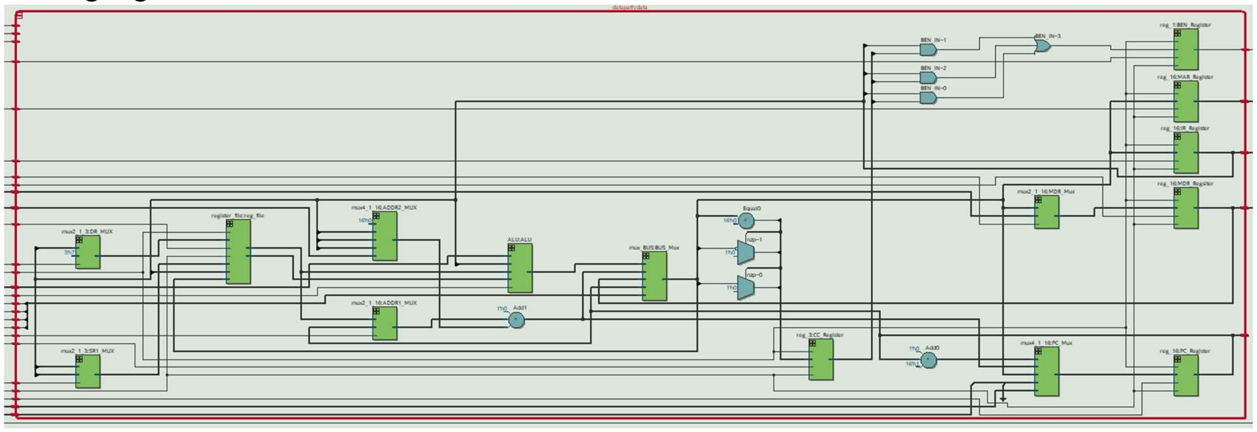
Module: datapath (in Datapath.sv)

Inputs: Clk, Run, [9:0] SW, Continue, Reset, LD_MAR, LD_MAR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, [15:0] MDR_In, [1:0] PCMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, [1:0] ADDR2MUX, [1:0] ALUK, MIO_EN

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

Description: This module lays out all of the hardware of the slc-3 processor based off of the Datapath diagram of the slc-3 diagram given. It instantiates all of the registers and muxes, the ALU, the BEN and CC logic, the bus, and establishes all the connections between these modules.

Purpose: This module serves as 1 of the big 3 modules needed to come together to create the slc-3 processor. This one represents all of the individual components like registers and muxes, coming together as one unit.



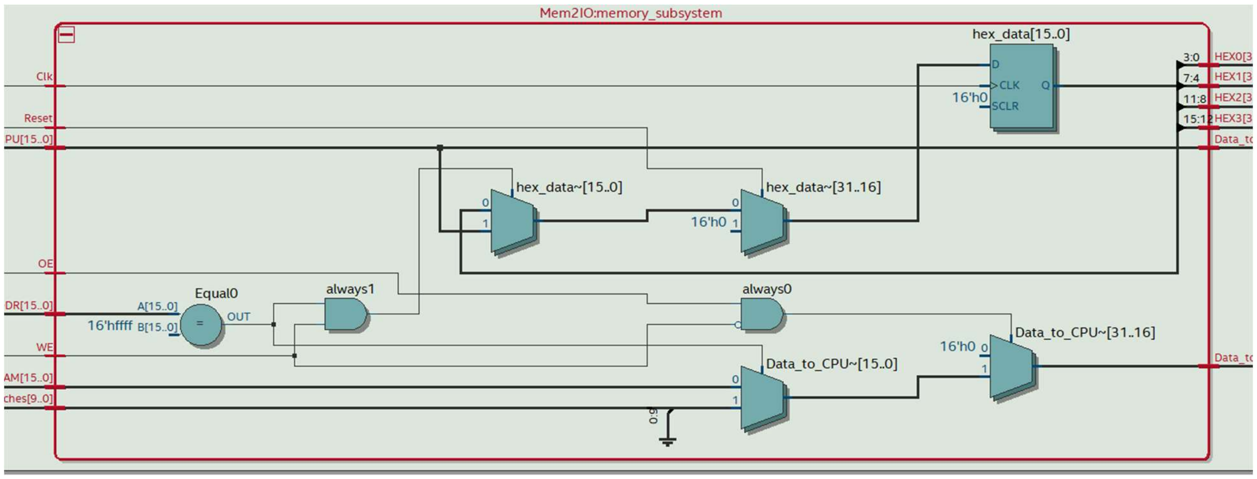
Module: Mem2IO (in Mem2IO.sv)

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

Outputs: [3:0] HEX(0-3), [15:0] Data_to_SRAM, [15:0] Data_to_CPU

Description: This is the next level down from the top level and universal across both the simulation and the FPGA. This module instantiates a connect between the memory and the I/O device, in our case the switches. This module takes in the switches and uses them to output which data should be sent both to the SRAM (Data_to_SRAM) and the CPU (Data_to_CPU). It also outputs to the hex displays to display what is being called back from memory.

Purpose: This module serves as 1 of the big 3 modules needed to come together to create the slc-3 processor. This module represents the connection between the memory (whether it be RAM on the FPGA or test memory in simulation) and the I/O of the switches and buttons on the FPGA.



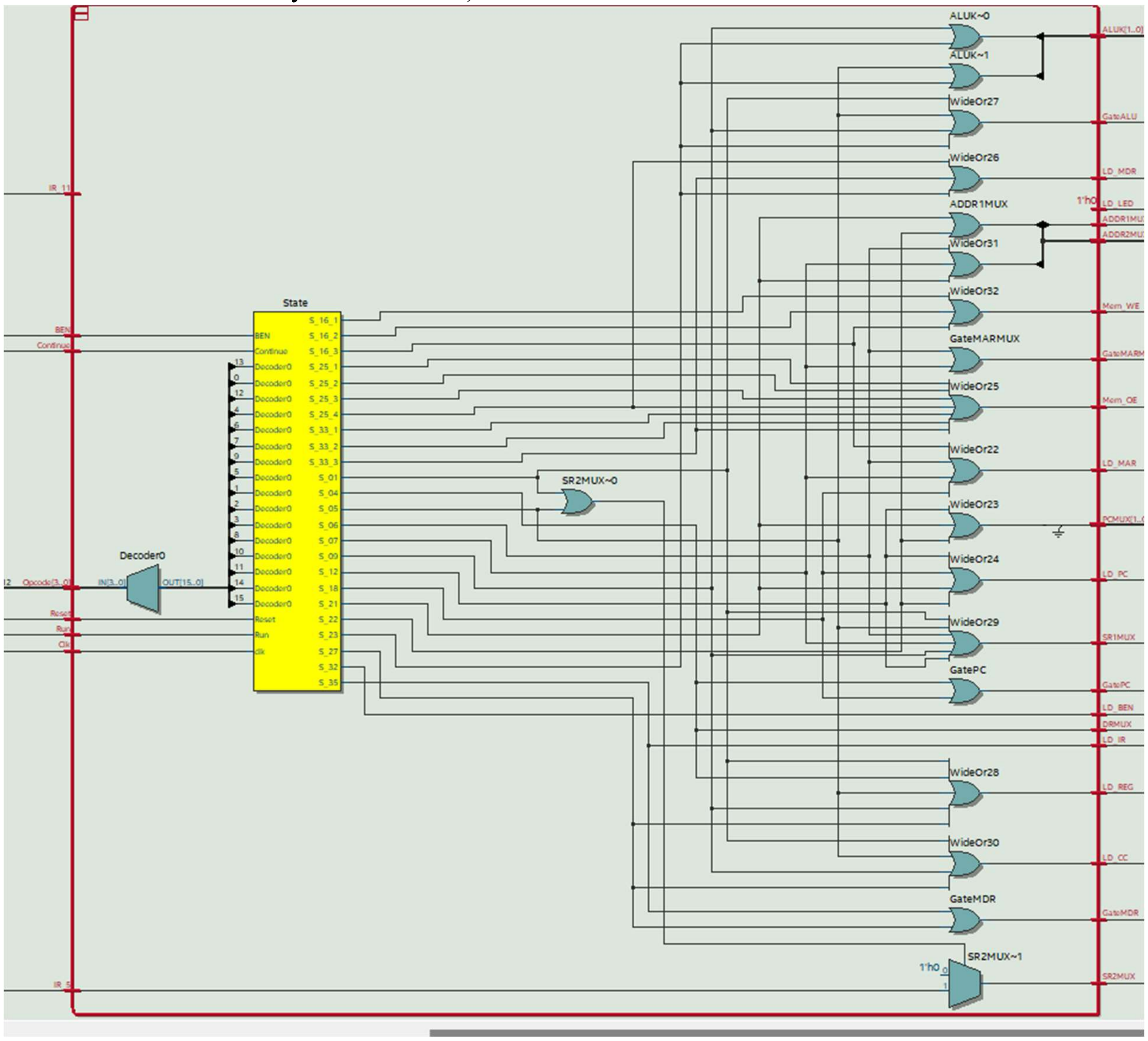
Module: ISDU (in ISDU.sv)

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

Outputs: LD_MAR, LD_MAR, 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, [1:0] ALUK, Mem_OE, Mem_WE

Description: This is the next level down from the top level and universal across both the simulation and the FPGA. This module instantiates a connect between the memory and the I/O device, in our case the switches. This module takes in the switches and uses them to output the corresponding data to both to the SRAM (Data_to_SRAM) and the CPU (Data_to_CPU). It also outputs to the hex displays to display what is being called back from memory.

Purpose: This module serves as 1 of the big 3 modules needed to come together to create the slc-3 processor. This module represents the connection between the memory (whether it be RAM on the FPGA or test memory in simulation) and the I/O of the switches and buttons on the FPGA.



Module: memory_contents (in memory_contents.sv)

Inputs: N/A

Outputs: N/A

Description: This sv module creates an array of memory to be used for the testing and execution of the slc-3 modules.

Purpose: This module is used to create memory array to run programs from, and in our case the test cases. This module does not have an RTL diagram.

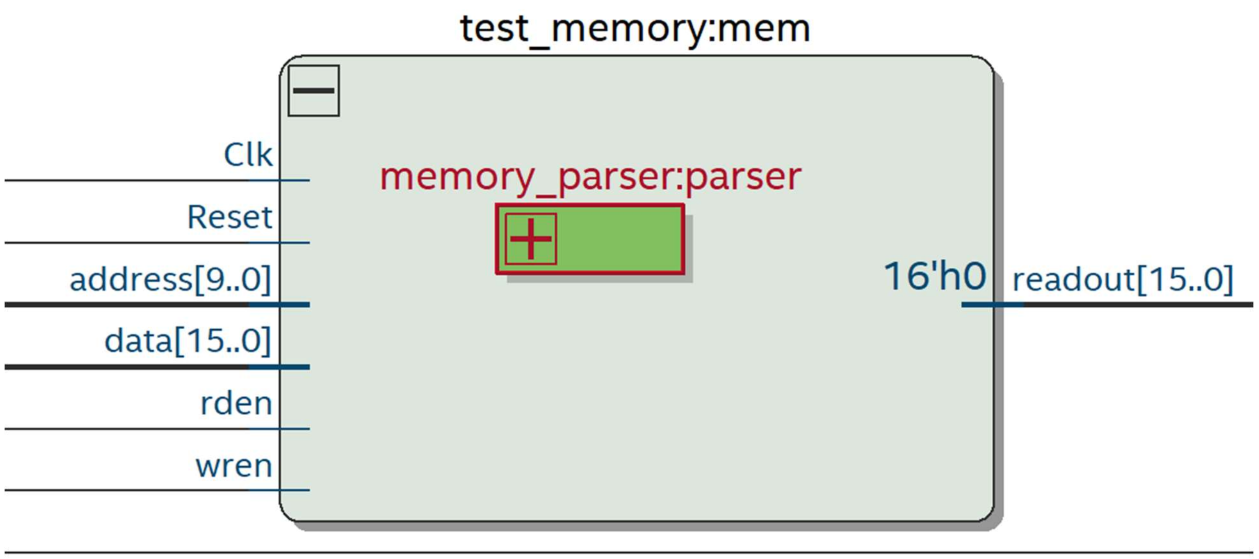
Module: test_memory (in test_memory.sv)

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

Outputs: [15:0] readout

Description: This module takes in data and an address and is used as link between the memory contents sv file and the greater memory of the FPGA or simulation.

Purpose: This module is used as the link from the memory contents to the memory modules that is used on the FPGA and the simulation.



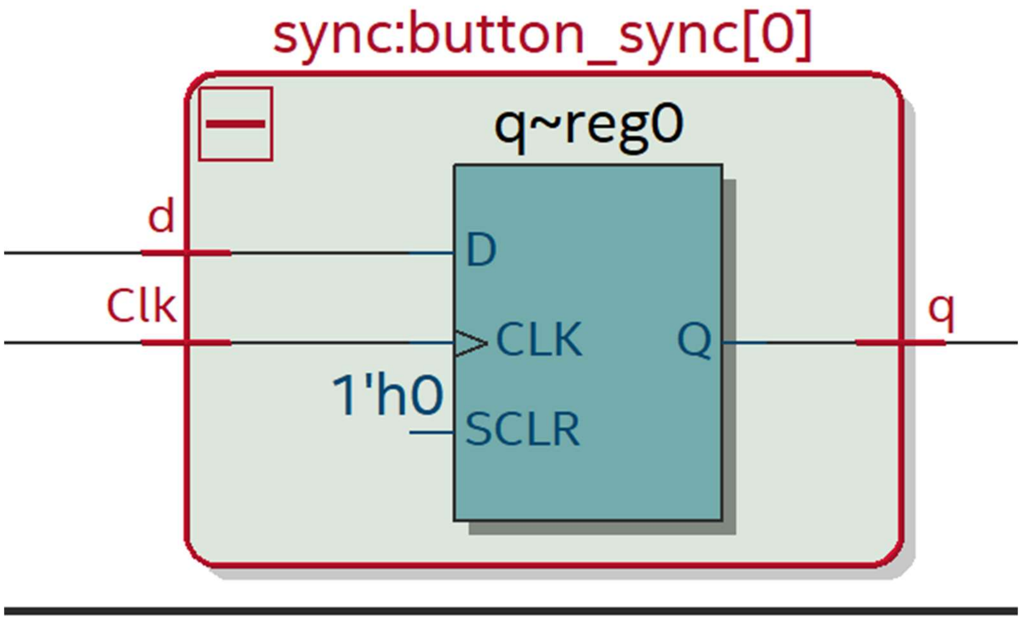
Module: sync (in synchronizers.sv)

Inputs: Clk, d

Outputs: q

Description: This module takes in the clock and input d and outputs q.

Purpose: Without this module, the slc-3 module would function mostly the same, but occasionally the clock would desync and cause the system to crash. This module prevents that from ever happening.



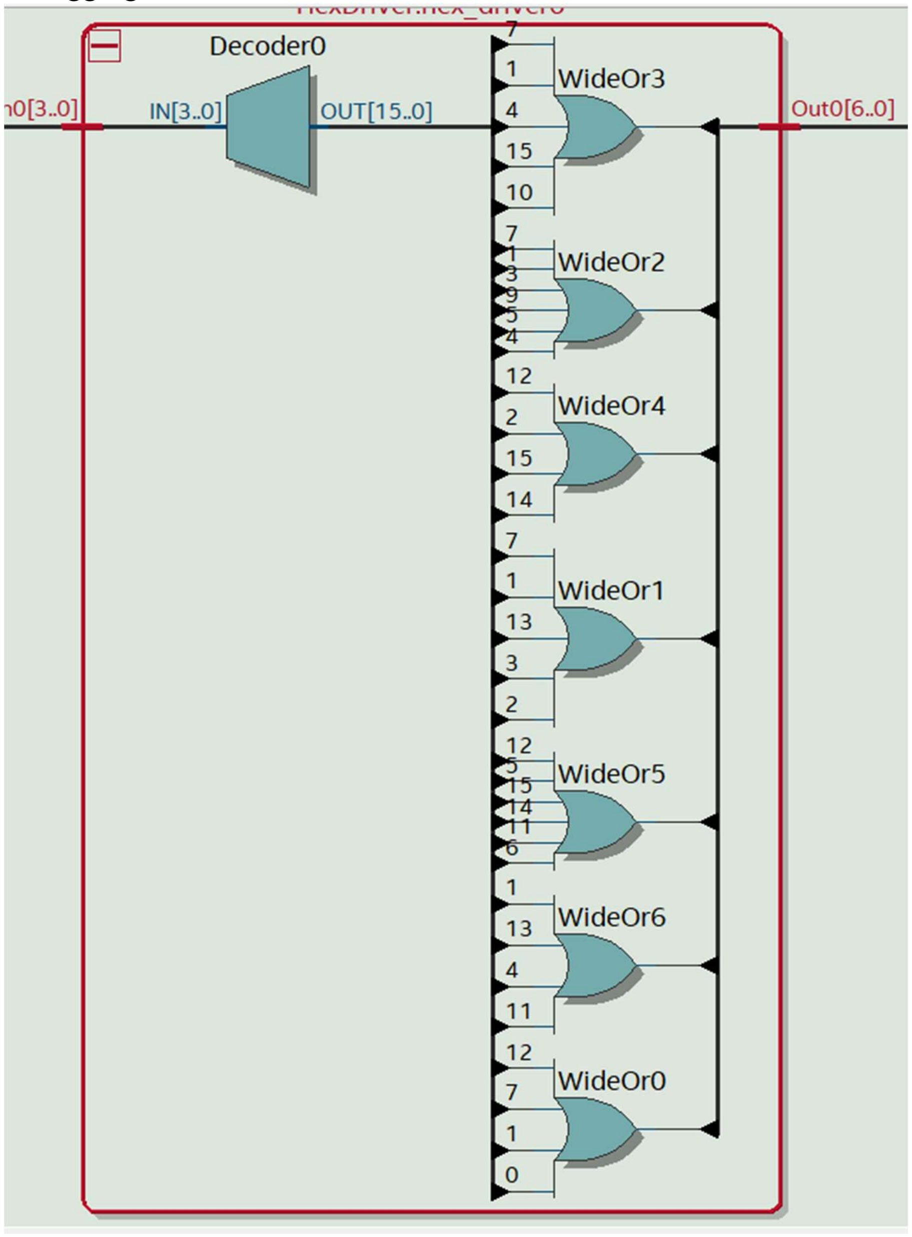
Module: HexDriver (in HexDriver.sv)

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module pairs every unique case of 4-bits (16 in total) into a 7-bit value used to illuminate the hex displays.

Purpose: This was used to output the values we were working with to the hex displays, both for debugging and for the demo itself.



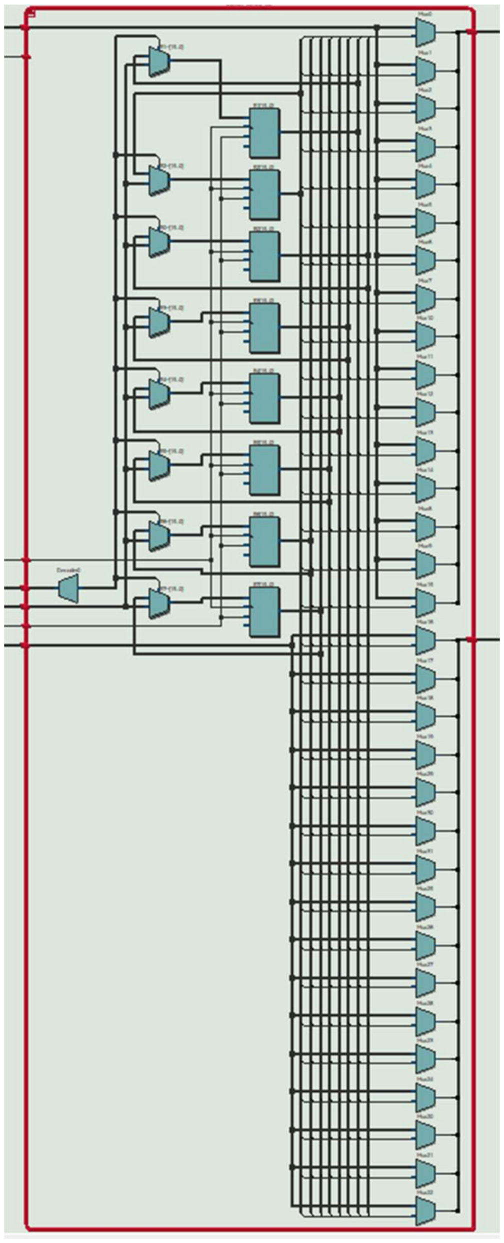
Module: register_file (in register_file.sv)

Inputs: LR_REG, Clk, Reset, [2:0] SR2, [2:0] SR1MUX_OUT, [2:0] DRMUX_OUT, [15:0] bus

Outputs: [15:0] SR1_OUT, [15:0] SR2_OUT

Description: This module instantiated the register file that is used to create and manage the 8 registers used in slc-3, R0-R7. This module loads and resets the given registers based on the value of the DRMUX_OUT. It outputs along SR1 and SR2 for the 2 source registers.

Purpose: This module is used to control the values of the 8 registers, R0-R7, needed to properly implement an slc-3 processor.



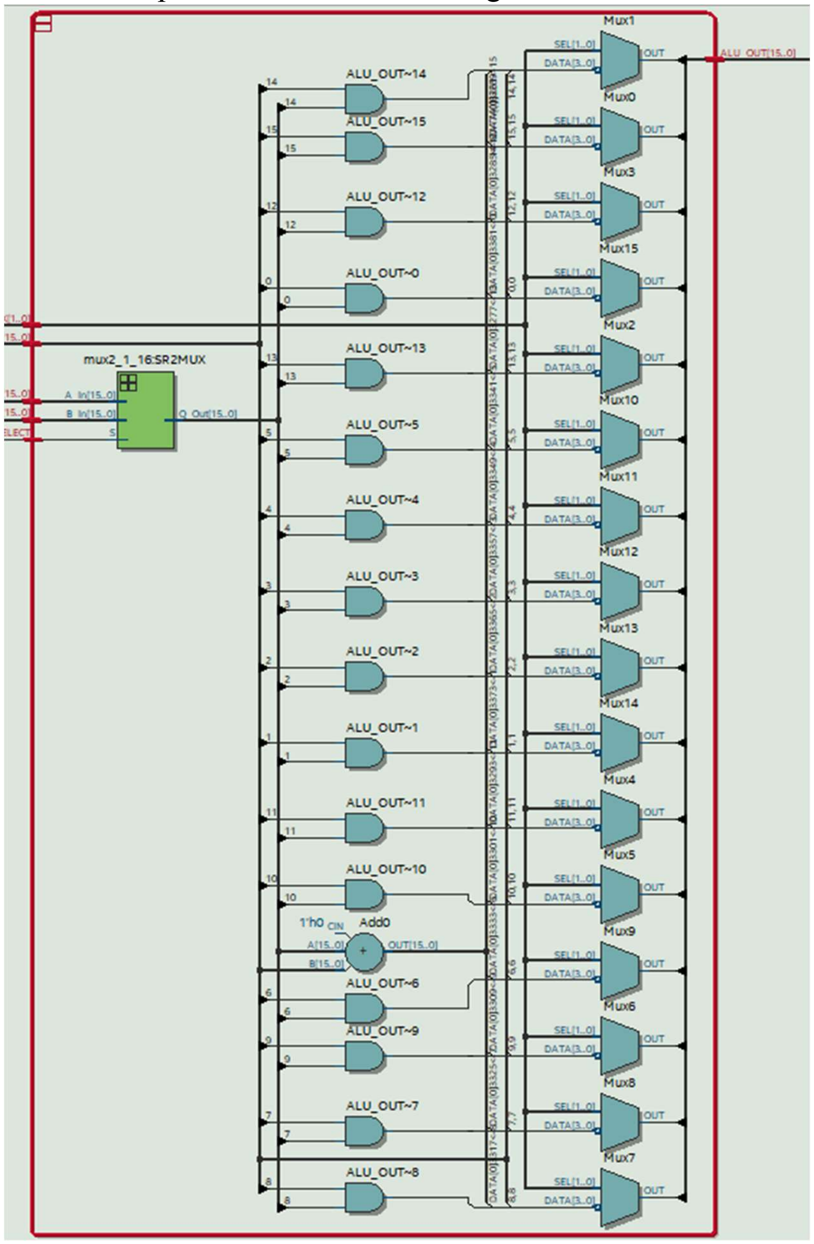
Module: ALU (in ALU.sv)

Inputs: SR2_SELECT, [1:0] ALUK, [15:0] SR1OUT, [15:0] SR2OUT, [15:0] IR_5_Extended

Outputs: [15:0] ALU_OUT

Description: This serves as the SR2 mux and the ALU. The SR2 mux is used to decide what is being sent to ½ of the ALU. This ALU will perform ADD, AND, NOT, and PASS (send the given value through).

Purpose: This module is essential to make sure that the slc-3 processor is performing the correct arithmetic operations on the source register values.



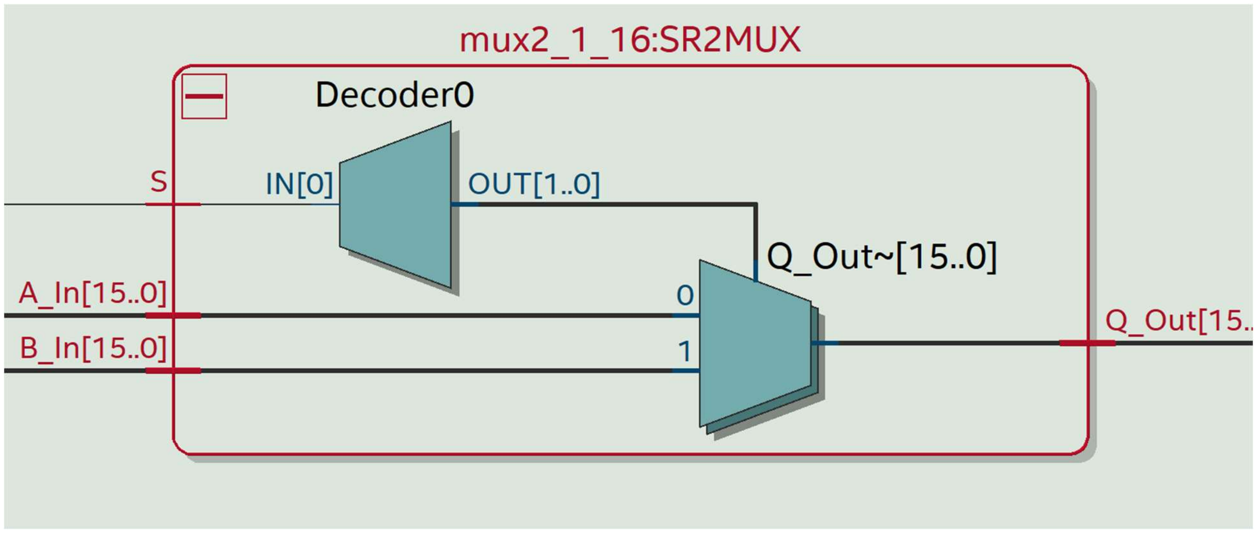
Module: mux(in)_(out)_(bits) (in Mux.sv)

Inputs: S, A_In-D_In

Outputs: Q_Out

Description: It should be noted that I am grouping all the muxes we used into this module. Our naming convention was as follows: A 4-to-1 mux that is 16-bits wide would be instantiated as a module called mux4_1_16 and would have 4 inputs, A_In-D_In, and one output, Q_Out. A 2-to-1 mux that is 3-bits wide would be instantiated as a module called mux2_1_3 and would have 2 inputs, A_In-B_In, and one output, Q_Out. There are 3 types of muxes in Mux.sv.

Purpose: Muxes are an essential part of the datapath framework and as a result, the datapath instantiates 7 of them. One is also instantiated within the ALU.



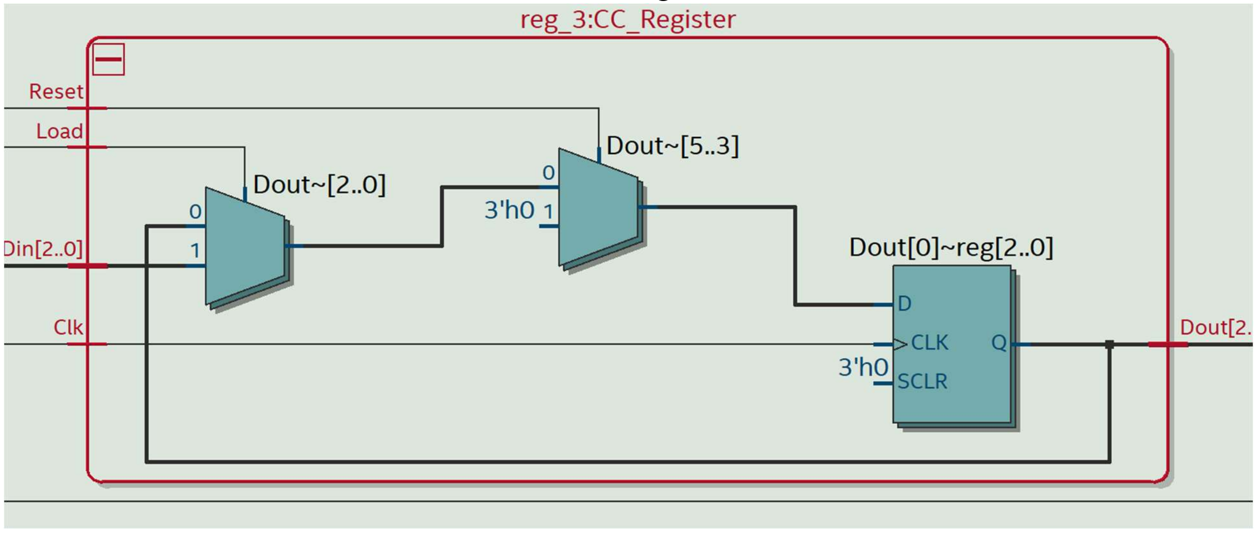
Module: reg_(bits) (in Registers.sv)

Inputs: Clk, Reset, Load, Din

Outputs: Dout

Description: It should be noted that I am grouping all the registers we used into this module. Our naming convention was as follows: a 16-bit register would be instantiated with a module called reg_16 (as it is storing a 16 bit value). Similarly, a 4-bit register would be called reg_4. All registers take in a Clk, Reset, and Load input as well as Din for data in. They output Dout (data out). Din and Dout will be the same width as the naming convention of the register.

Purpose: Registers are an essential part of the Datapath framework and as a result, the Datapath instantiates 6 of them (13 if you include the register file, but this is a separate module). This includes the IR, MAR, MDR, PC, BEN, and CC registers.



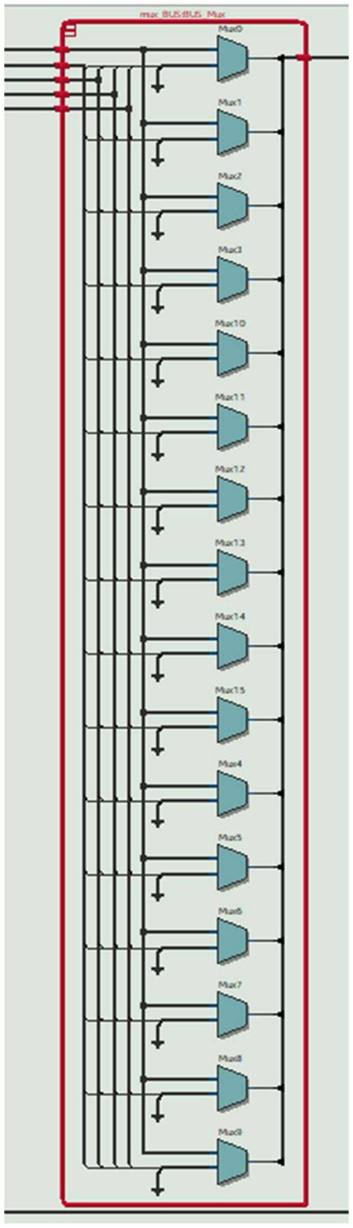
Module: mux_BUS (in Mux.sv)

Inputs: [3:0] S, [15:0] ALU_Bus, [15:0] MDR_Bus, [15:0] PC_Bus, [15:0] MAR_Bus

Outputs: [15:0] bus

Description: This is the mux that was put in place of the tri-state buffers to ensure only one value is on the bus at a given time. This was implemented using a 1 hot method with a select of 4-bits: 0001 meant the bus comes from the ALU, 0010 from the MAR, 0100 from the MDR, and 1000 from the PC.

Purpose: Without a properly functioning bus, nothing within the slc-3 would work as the bus is essential to getting the proper value loaded into IR in the fetch cycle, as well as moving data around in the execute cycle, and much more.


mux_BUS:BUS_Mux
Mux0
Mux1
Mux2
Mux3
Mux10
Mux11
Mux12
Mux13
Mux14
Mux15
Mux4
Mux5
Mux6
Mux7
Mux8
Mux9

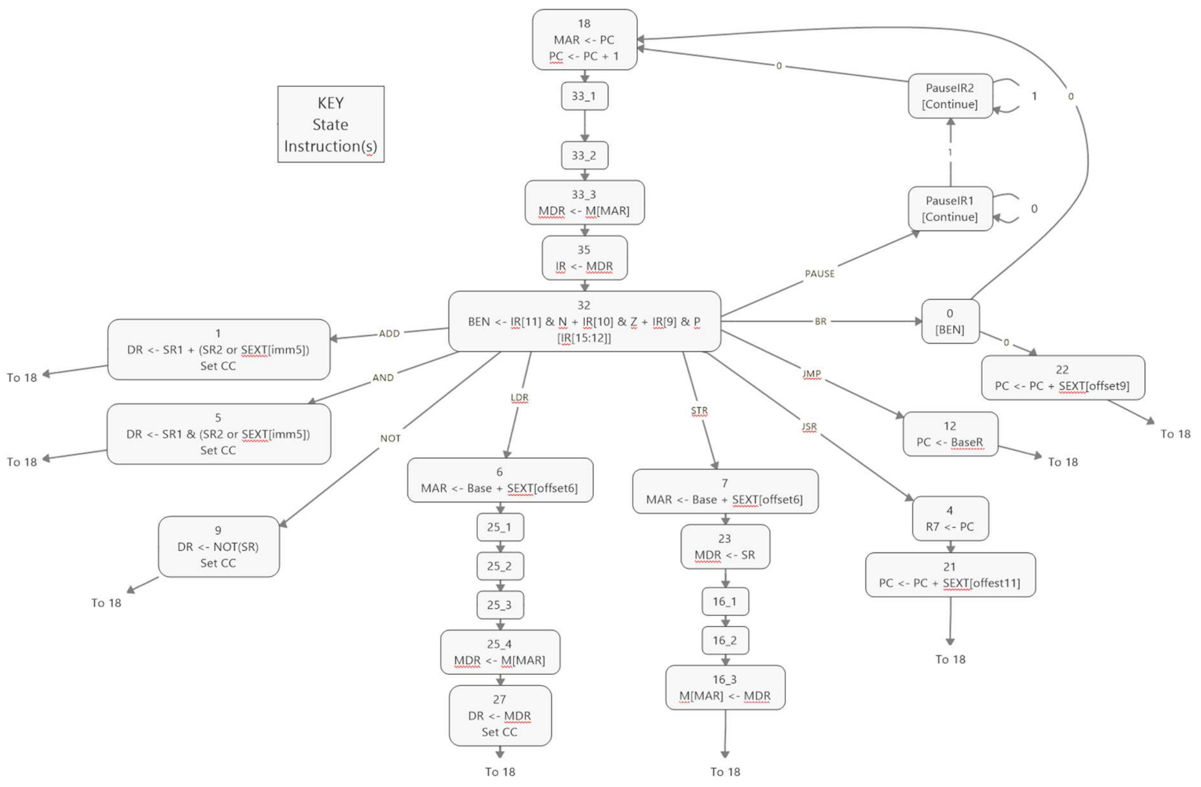
Operation of ISDU

The ISDU's main functionality is to control the Datapath through every state. There are several aspects to control the Datapath: several load signals for the various registers, the control signals for the inputs into the BUS, the control signal to the ALU (ALUK), the selection bits for the various MUX's, and memory write and read commands. By default, these various signals are set as zeros. Based on the current state, the necessary signals are changed.

1. LD_MAR
   The purpose of this is signal was to signal to the MAR to load new incoming data into the register. The states that make this signal high are States 6, 7, 16_3, and 18.
2. LD_MDR
   The purpose of this is signal was to signal to the MAR to load new incoming data into the register. The states that make this signal high are States 23, 25_4, and 33_3.
3. LD_IR
   The purpose of this is signal was to signal to the IR to load new incoming data into the register. The only state that makes this signal high is State 35.
4. LD_BEN
   The purpose of this is signal was to signal to the BEN to load new incoming data into the register. The only state that makes this signal high is State 32.
5. LD_CC
   The purpose of this is signal was to signal to CC to load new incoming data into the register. The states that make this signal high are States 1, 5, 9, and 27.
6. LD_REG
   The purpose of this is signal was to signal to the Register File to load new incoming data into the selected register. The states that make this signal high are States 1, 4, 5, 9, and 27.
7. LD_PC
   The purpose of this is signal was to signal to the PC to load new incoming data into the register. The states that make this signal high are States 18, 12, 21, and 22.
8. GatePC
   The purpose of this signal is to select the value stored within the PC to be sent to the BUS. The states that make this signal high are States 4, and 18.
9. GateMDR
   The purpose of this signal is to select the value stored within the MDR to be sent to the BUS. The states that make this signal high are States 27, and 35.
10. GateALU
    The purpose of this signal is to select the output by the ALU to be sent to the BUS. The states that make this signal high are States 1, 5, 9, and 23.

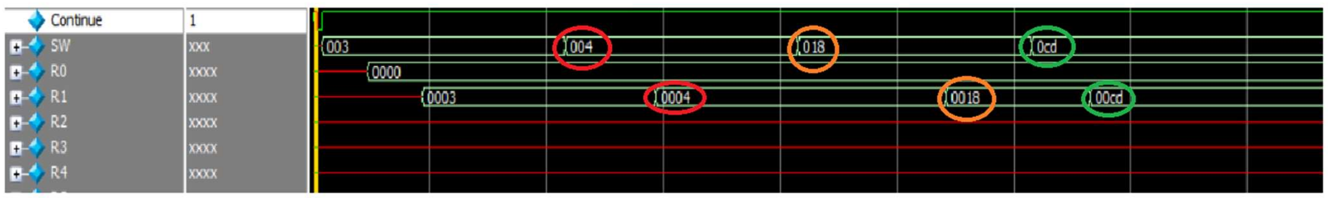
11. GateMARMUX
    The purpose of this signal is to select the output by the Address Adder to be sent to the BUS. The states that make this signal high are States 6, and 7.
12. ALUK
    The purpose of this 2-Bit signal is to determine which operation is to be performed by the ALU. The signal will be 0x00 at State 1. The signal will be 0x01 at State 5. The signal will be 0x10 at State 9. The signal will be 0x11 at State 23.
13. PCMUX
    The purpose of this 2-Bit signal is to act as the selection bits for the 4-to-1 MUX that determines the input into the PC register. The signal will be 0x00 at State 18. The signal will be 0x01 at States 12, 21, and 22.
14. DRMUX
    The purpose of this signal is to act as the selection bit for the 2-to-1 MUX that determines the Direct Register input into the Register File. The signal will be 0 at States 1, 5, 9, and 27. The signal will be 1 at State 4.
15. SR1MUX
    The purpose of this signal is to act as the selection bit for the 2-to-1 MUX that determines what part of the IR is input as the SR1 into the Register File. The signal will be 1 at States 1, 5, 6, 7, 9, and 12. The signal will be 0 at State 23.
16. SR2MUX
    The purpose of this signal is to act as the selection bit for the 2-to-1 MUX that determines what will be the second input into the ALU. The signal will be set to the value of IR[5] at States 1 and 5.
17. ADDR1MUX
    The purpose of this signal is to act as the selection bit for the 2-to-1 MUX that determines whether the value of PC or SR1 will be output into the Address Adder as an input. The signal will be 0 at States 6, 7, 12. The signal will be 1 at States 21, and 22.
18. ADDR2MUX
    The purpose of this 2-Bit signal is to act as the selection bits for the 4-to-1 MUX that determines what part or IR will be output into the Address Adder as an input. The signal will be 0x00 at State 12. The signal will be 0x01 at States 6, and 7. The signal will be 0x10 at State 22. The signal will be 0x11 at State 21.
19. Mem_OE
    The purpose of this signal is to enable the reading of data within memory. This signal will be 1 at States 23, 25_1, 25_2, 25_3, 25_4, 33_1, 33_2, and 33_3.
20. Mem_WE
    The purpose of this signal is to enable the writing of data within memory. This signal will be 1 at States 16_1, 16_2, and 16_3.

State Diagram for Control Unit



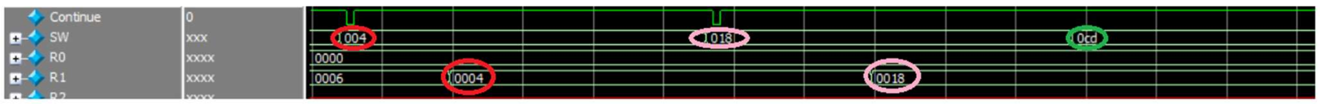
Annotated Simulation Waveforms

1. I/O Test 1



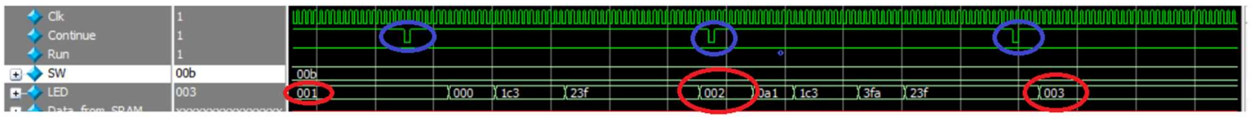
The basic I/O test is shown here. As the switches change, the value in R1 changes WITHOUT continue needing to be pressed. It should be noted that the value of R1 is then displayed on the hex displays

2. I/O Test 2



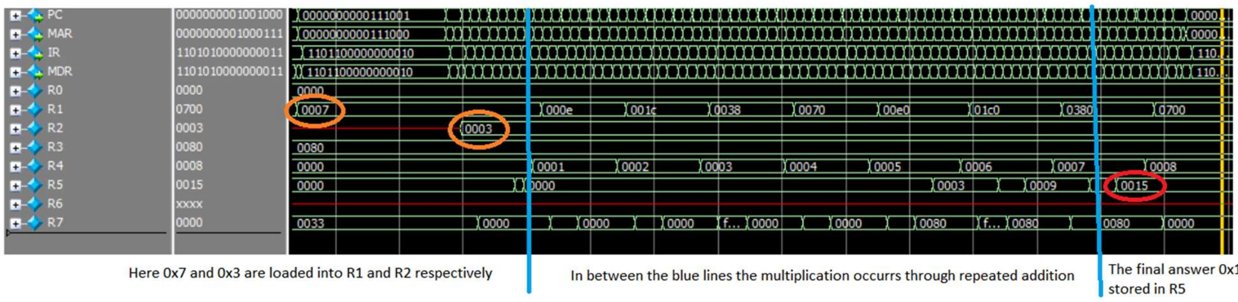
Basic I/O test 2. Here it should be noted that the switches are transfered to R1 only when continue is pressed, as seen above. The last value, 0xcd is loaded into the switches, but continue is never pressed, so it is not loaded into R1. It should be noted R1 is dipslayed on the hex displays.

3. Self-Modifying Code



Self Modifying code test. As continue is pressed, the value the LED eventually sets on is the next incremented value. It should be noted that there are other values taken on by the LED as it does the math, but it always ends at the next value, right after continue is pressed.

4. Multiplication

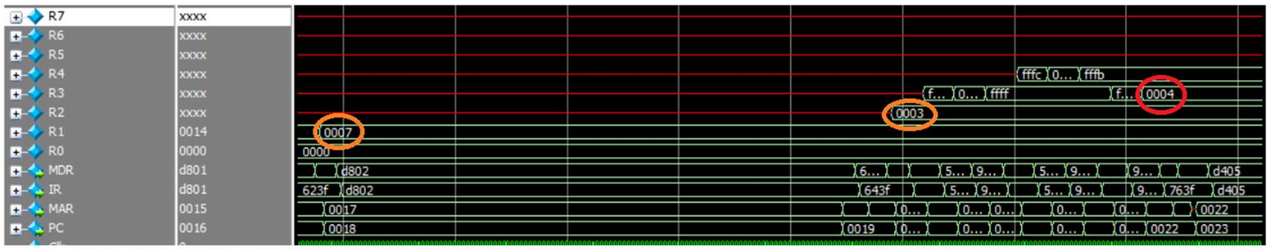


Here 0x7 and 0x3 are loaded into R1 and R2 respectively

In between the blue lines the multiplication occurrs through repeated addition

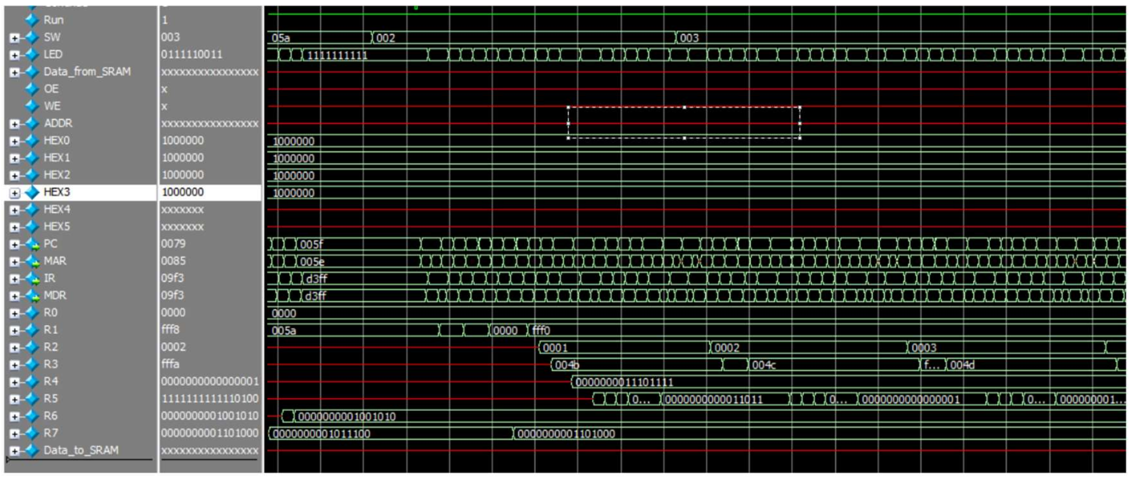
The final answer 0x15 is stored in R5

5. XOR



0x7 is loaded into R1 and 0x3 is loaded into R2. We then XOR them together to get 0x4 inside of R3. This is the expected result

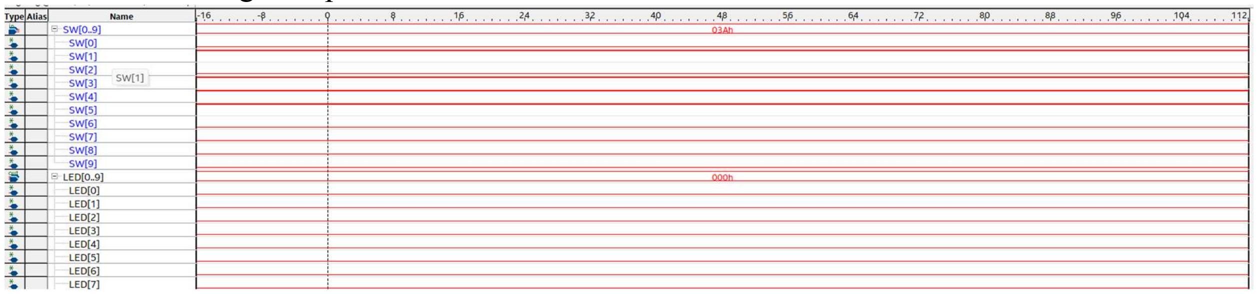
6. Sort



Due to the nature of the sort taking much much longer than the other funcitons, it is hard to show a readable simulation of the sort working in full display. So this is a truncated version. This shows the switches being set to 2 after the sort has been loaded, and continue being pressed. This starts the sort. The switches are changed to 3 for display and then the program waits until the sort is done before continue is pressed again to display the sorted values.

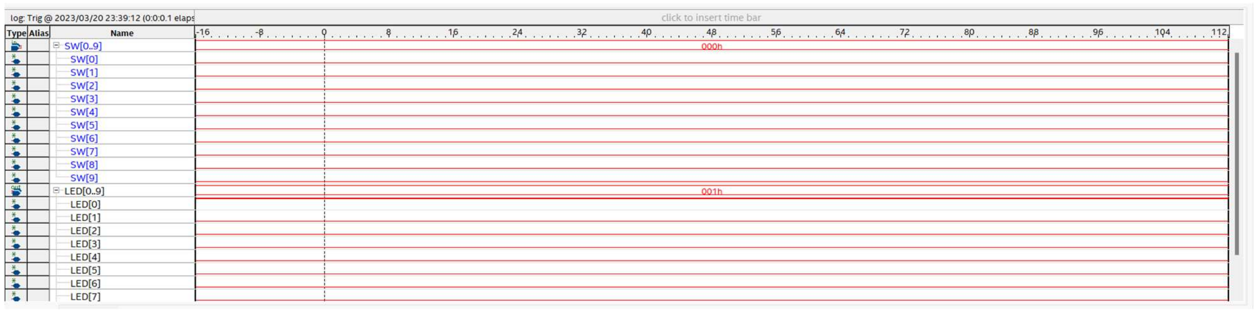
Signal Tap EC

The first signal tap was XOR seen below:



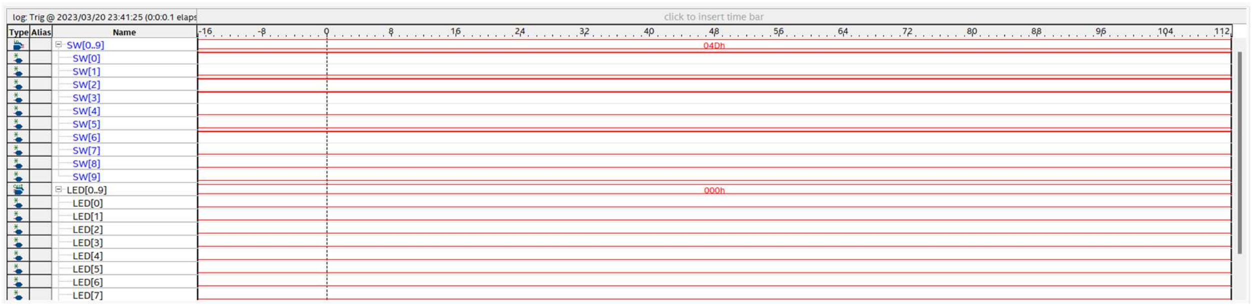
Using this output we can see that the XOR is capable of 7.5 MIPS.

The second signal tap is for multiplier seen below:



Using this plot we can see that multiplier is capable of 8 MIPS

The final signal tap is for the the sort test.



This one is slower than the other two at 10 MIPS.

The average of these is (10+8+7.5)/3 = 17/2 MIPS.

Post-Lab

1. Design Statistics Table

LUT	918
DSP	0
Memory(B-Ram)	0
Flip-Flop	45
Frequency	50 Mhz
Static Power	89.94 mW
Dynamic Power	0 mW
Total Power	98.7 mW

2. Post-Lab Questions

a. **What is MEM2IO used for, i.e. what is its main function?**
The main function of MEM2IO is to act as interface between memory and the input, switches, and the output, the HEX Display of the FPGA. If Read Enable (OE) was enabled, MEM2IO would read the data from the switches when the current address was 0xFFFF, otherwise it would read the data stored within the current address. If Write Enable (WE) was enabled and the current address was 0xFFFF, MEM2IO would display the current data from the CPU on the FPGA's HEX Display, otherwise the HEX Display would hold their previous values.

b. **What is the difference between BR and JMP instructions?**
The main difference between BR and JMP is that BR only changes PC when the specific condition is met, while JMP will always change PC. JMP takes the value

stored within the specified register and sets the value of PC to that value. BR will first check if BEN is low or high. When low, BR will do nothing, sending the processor back to the Fetch State. When high, BR will change the value of PC to PC added with the sign extension of 9 least significant bits of IR (IR[8:0]). Another difference is the range of possible values for PC with these two instructions. JMP can change the PC to any value, the program simply needs to assign the correct value to a register. BR is limited only to addresses 255 higher than PC or 256 lower than PC, this is due to this being the range of a 9-Bit two's complement number.

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 purpose of the R signal in the Patt and Patel version of LC3 is to signal when the data from memory is ready to access. Once the data from memory is accessed, the action of the current state acts correctly. Before the data was ready, the processor still does the current action, but the data sent from memory would be garbage as the memory has not been connected. We compensate for the lack of this signal in our design by adding wait states to any state that reads or writes from memory. We typically had 2 or 3 wait states that would either enable reading or writing to memory. Then our last state would do the actual action needed as the CPU had already connected with memory. The implication of this implementation on synchronization is quite significant. In the original design, in the exact clock cycle that memory could be accessed, the CPU would continue through the state machine. However, in our design, there is always the same amount of wait time based on the amount of wait states we implemented. This throttles the performance significantly. If we had 3 wait states, but that specific address only took 2 wait states to access, then the original implementation would continue at that point, while our implementation always will have to wait 3 states regardless of when the memory was available.

Conclusion

1. **Functionality and Bugs**
   We had several bugs in our development of this Microprocessor. Our first bug was an issue of our implementation of how JMP worked within the Datapath. We originally used the BUS to update PC, we then fixed this by having the signal go through Address Adder and the ADDR1MUX. Another bug we had was that the output of the SR2MUX was only 1-Bit long instead 16-Bits. The final major bug we had was with our implementation of BEN: we originally had the combinational logic that controlled the value of BEN read the value of nzp, which would change after every instruction, instead of the value of CC, which was based on nzp but would only change when LD_CC was high. After fixing all

of these bugs, our processor had full functionality. We were able to perform every test available.

2. **Feedback**
   Overall this lab was very well structured. We liked how there was an intro week where there was time for us use to learn about the large amount of given files and get ahead on the second part of the lab. We felt that there could have been some more guidance on exactly which files were meant to be changed and which files we had to create. Additionally, we felt that test programs instruction were confusing at first. Only after having the basic concept explained to us, we were able to perform the test by ourselves.