**ECE 385**

Fall 2019

Experiment #6

Simple Computer SLC-3.2 in SystemVerilog

Jacqueline Jiang and JiHo Han

AB6 / Wednesday 12:00 - 2:50 p.m.

Nicholas Cebry and Yujie Zhou

# Introduction

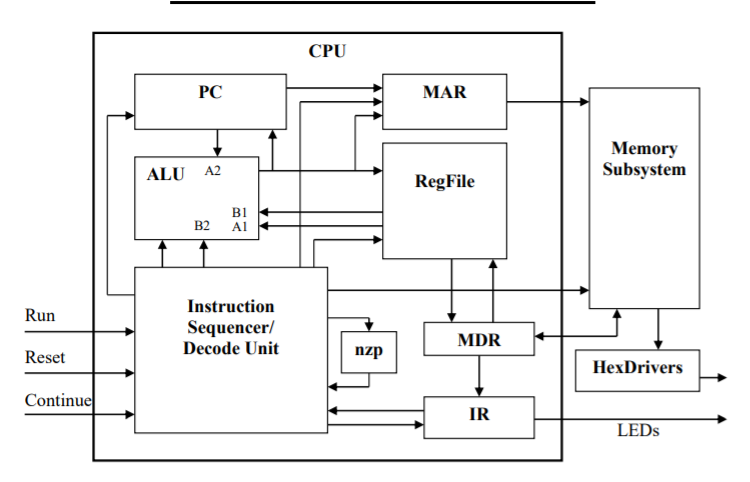
## Summary of the Basic Functionality of the SLC-3 Processor

Using SystemVerilog, we created a simple computer, titled SLC-3, that follows a subset of the LC-3 ISA. This simplified microprocessor is 16-bit addressable and has a data width of 16-bits. The main components of the processor are a central processing unit (CPU), a memory storing both data and instructions, and am I/O interface for communicating with external devices. It is able to accomplish 11 different instructions including add, bitwise and, bitwise not, branch, unconditional jump, jump to the subroutine, load register, store register, and pause. It is controlled by three buttons, reset, run, and continue. Additionally, it takes in switch inputs and can output registers to both the Hex display and LEDs.

# Written Description and Diagram of SLC-3

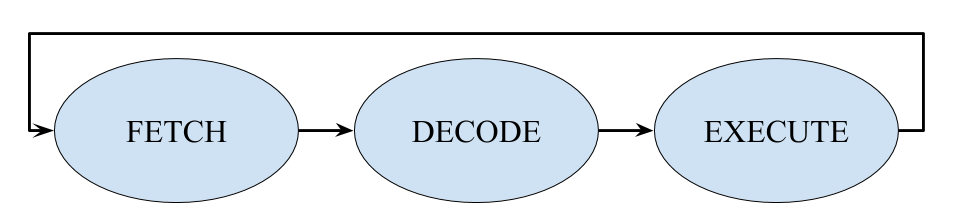
## Summary of Operation

## Our SLC-3 consists of a central processing unit (CPU), a memory storing both data and instructions, and am I/O interface for communicating with external devices, which includes switches, pushbuttons, Hex display, and LEDs. The CPU consists of a Program Counter (PC), an Instruction Register (IR), an Instruction Sequencer and Decoder, a Memory Address Register (MAR), a Memory Data Register (MDR), a Condition Code Registers, eight 16-bit general-purpose registers, and an Arithmetic Logic Unit (ALU). We used SRAM for our memory subsystem. Although our memory is 16-bit addressable, the SRAM is 20-bit addressable, so we zero-extend all addresses that we used when accessing SRAM.



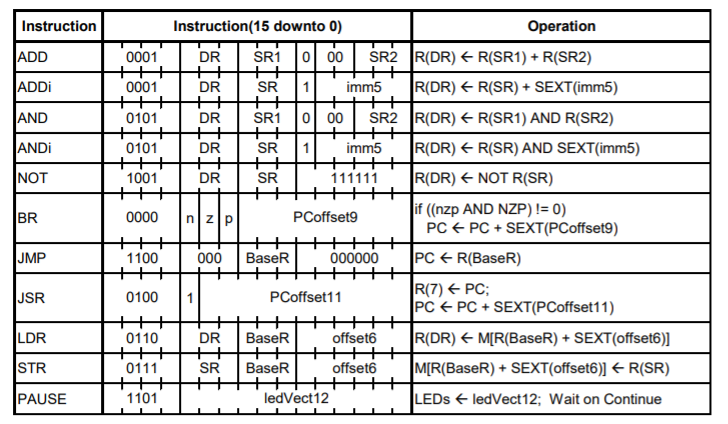
*Figure 1: High-Level Block Diagram of SLC-3*

At the high level, this microprocessor accomplishes three tasks — FETCH, DECODE, and EXECUTE. First, the microprocessor will fetch an LC-3 instruction from the SDRAM and increment the PC. Then, it will decode the fetched instruction using the 4 most significant bits of the instruction and then execute it. This cycle will continue to repeat until the system is reset. Figure 2 demonstrates these three tasks.



*Figure 2: Basic Functionality of the Microprocessor*

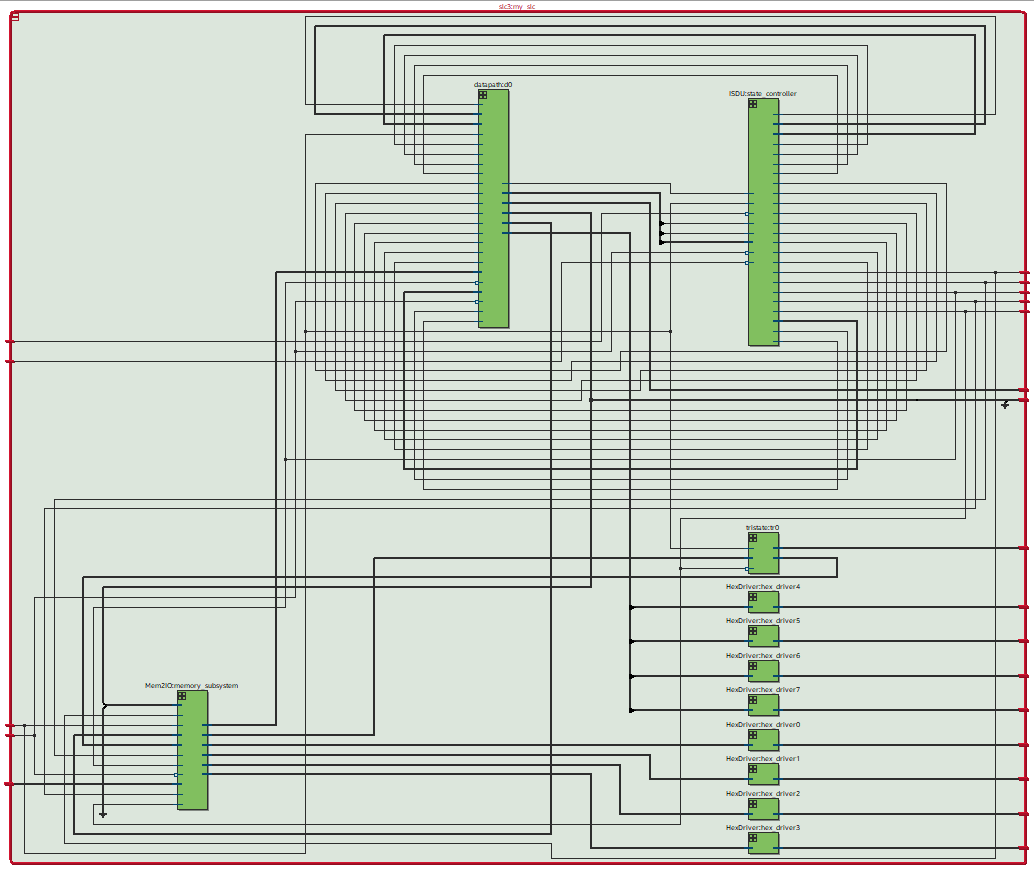
In total, our microprocessor can accomplish 11 different instructions — ADD, ADDi, AND, ANDi, NOT, BR, JMP, JSR, LDR, STR, and PAUSE. Figure 3 shows the SLC-3 ISA. There are two different types of add instructions; both of which set the status register. The ADD instruction can add two registers together and store the result into another register. The ADDi instruction adds a 5-bit immediate value to a register and stores the sum into a destination register. There are also two different types of and instructions that also set the status register. Similar to the ADD instructions, the AND instruction can bitwise and two registers together and store the result into a destination register. The ANDi instruction can bitwise and a 5-bit immediate value to a register and store the result into another register. The NOT instruction will bitwise not a register, store the result into another register, and set the status register. The BR instruction will take in a specific set of condition codes (NZP) and compare them to the status register to see if one of the conditions match. If it does, the program will jump to another instruction, the address of which is a 9-bit offset from an address in a base register; otherwise, it will continue on with its execution as normal. The JMP instruction will unconditionally jump to a location specified by the given register. The JSR instruction will save the current PC in a register and jump over to another address that can have an 11-bit offset away from the current PC. This saved PC address allows the program to return to its current execution after it is done with its subroutine. The LDR instruction will load in data from memory to a specified register and set the status register. The STR instruction will store data from a specified register to memory. The address for memory access for both the LDR and STR instructions is calculated by adding a 6-bit offset to a base register. Additionally, these two instructions load from the on-board switches and store to the Hex display accordingly when the memory address to be accessed is set to 0x0FFFF. The PAUSE instruction will wait for the Continue button to be pressed. In the meantime, it will display the 12 least significant bit on the LEDs. Note that all offsets and immediate values mentioned are signed 2’s complement values.



*Figure 3: The SLC-3 ISA*

## 

## Block Diagram of slc3.sv



*Figure 4: Block Diagram of slc3.sv*

## 

## Written Description of all .sv Modules

## Module: ALU from *ALU.sv*

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

Outputs: [15:0]ALUout

Description: This is our arithmetic logic unit. Given the select signal ALUK, it will perform one of 4 different operations on the given operands A and B: addition (00), bitwise and (01), bitwise not A (10), and pass A (11). By default, if none of the operations are selected, it will output don’t-cares. This accomplishes its task asynchronously to the circuit.

Purpose: This ALU is used to perform simple arithmetic operations within the SLC-3. It is mainly used in completing the ADD, ADDi, AND, ANDi, and NOT instructions.

## Module: CC from *CC.sv*

Inputs: [15:0]databus,

Outputs: [2:0]CCout

Description: This intakes the value in the data bus and determines if it has a positive, negative, or zero value. This accomplishes its task asynchronously to the circuit.

Purpose: This module helps calculate our condition codes for our status register.

## Module: CPUbus from *CPUbus.sv*

Inputs: GatePC, GateMDR, GateALU, GateMARMUX, [15:0]PC, [15:0]MDR, [15:0]ALUout, [15:0]ADDER

Outputs: [15:0]databus

Description: This is a multiplexer that determines what is released on the data bus. Because we do not have any internal tristate buffers (because it is not supported), we used a MUX to control what drives the bus. It takes in the values of the gates as select bits. If GatePC is high, the data in the PC register is placed on the data bus. If GateMDR is high, the data in the MDR register is placed on the data bus. If GateALU is high, the data from the ALU is placed on the data bus. If GateMARMUX is high, the data from the address adder is placed on the data bus. Otherwise, the data bus carries don’t care signals. This accomplishes its task asynchronously to the circuit.

Purpose: This module is used as a multiplexer to help control our databus.

Module: HexDriver from *HexDriver.sv*

Inputs: [3:0]In0

Outputs: [6:0]Out0

Description: This is a converter that takes in a 4-bit hexadecimal number and transforms it into an on-board seven-segment display output.

Purpose: This module transforms a hexadecimal input into the corresponding output signal for the seven-segment display.

## Module: ISDU from *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, [1:0]ALUK, Mem\_CE, Mem\_UB, Mem\_LB, Mem\_OE, Mem\_WE

Description: This is our control unit for our SLC-3. It contains a Moore state machine to control the output signals of each state. The state of the machine changes with the positive edge of the clock. See the next section for a more in-depth description of its functionality

Purpose: This module is used as the control unit of our microprocessor, controlling gates, loads, and multiplexers. It is also used as our instruction sequencer and decoder.

## Module: MUX2 from *MUX2.sv*

Inputs: select, [15:0]D0, [15:0]D1

Outputs: [15:0]out

Description: This is a generic 2-to-1 multiplexer. It defaults to a width size of 16, but this can be changed parameters. It will output don’t-cares if the select bit is not valid.

Purpose: This is used to select the values of the DR MUX, SR1 MUX, SR2 MUX, ADDR1 MUX, and MDR MUX in our SLC-3.

## Module: MUX4 from *MUX4.sv*

Inputs: select, [15:0]D00, [15:0]D01, [15:0]D10, [15:0]D11

Outputs: [15:0]out

Description: This is a generic 4-to-1 multiplexer. It defaults to a width size of 16, but this can be changed parameters. It will output don’t-cares if the select bits are not valid.

Purpose: This is used to select the values PC MUX and ADDR2 MUX in our SLC-3.

## Module: Mem2IO from *MEM2IO.sv*

Inputs: Clk, Reset, [19:0]ADDR, CE, UB, LB, OE, WE, [15:0]Switches, [15:0]Data\_from\_CPU, [15:0] Data\_from\_SRAM

Outputs: [15:0]Data\_to\_CPU, [15:0]Data\_to\_SRAM, [3:0]HEX0, [3:0]HEX1, [3:0]HEX2, [3:0]HEX3

Description: This controls loading and storing to both the SRAM and external devices like the switches and Hex display.

Purpose: This controls read-write access to memory and will load switches and store values in the Hex display.

## Module: register\_file from *RegisterFile.sv*

## Inputs: Clk, Reset, Load, [2:0]SR1, [2:0]SR2, [2:0]DR, [15:0]Data\_in

## Outputs: [15:0]SR1out, [15:0]SR2out

## Description: This contains our general purpose registers (R0 ~ R7). It consists of 8 16-bit registers, a decoder to determine when to load any registers, and two multiplexers for determining which register data to output.

Purpose: This is a wrapper to contain the logic necessary to maintain the general purpose registers in our SLC-3.

## Module: register from *Registers.sv*

Inputs: Clk, Reset, Load, [15:0]Data\_in

Outputs: [15:0]Data\_out

Description: This is a positive-edge triggered register that is capable of synchronously resetting and parallel loading. Although it defaults to 16 bits, it has a parameter that allows users to custom the data width to their purpose. When Reset is high, the register will be cleared with zeroes on the positive edge of Clk. When Load is high, the register will be loaded with Data\_in on the positive edge of Clk.

Purpose: This module is used to create registers to hold the values of MAR, MDR, IR, PC, BEN, CC, and our general-purpose registers in our SLC-3.

Module: SLC3\_2 from *SLC3\_2.sv*

Inputs: Clk, Reset, Load, [15:0]Data\_in

Outputs: [15:0]Data\_out

Description: This contains constants and functions that expand to the instructions used in the simulated memory.

Purpose: This is used to simplify the instruction writing for the simulated memory.

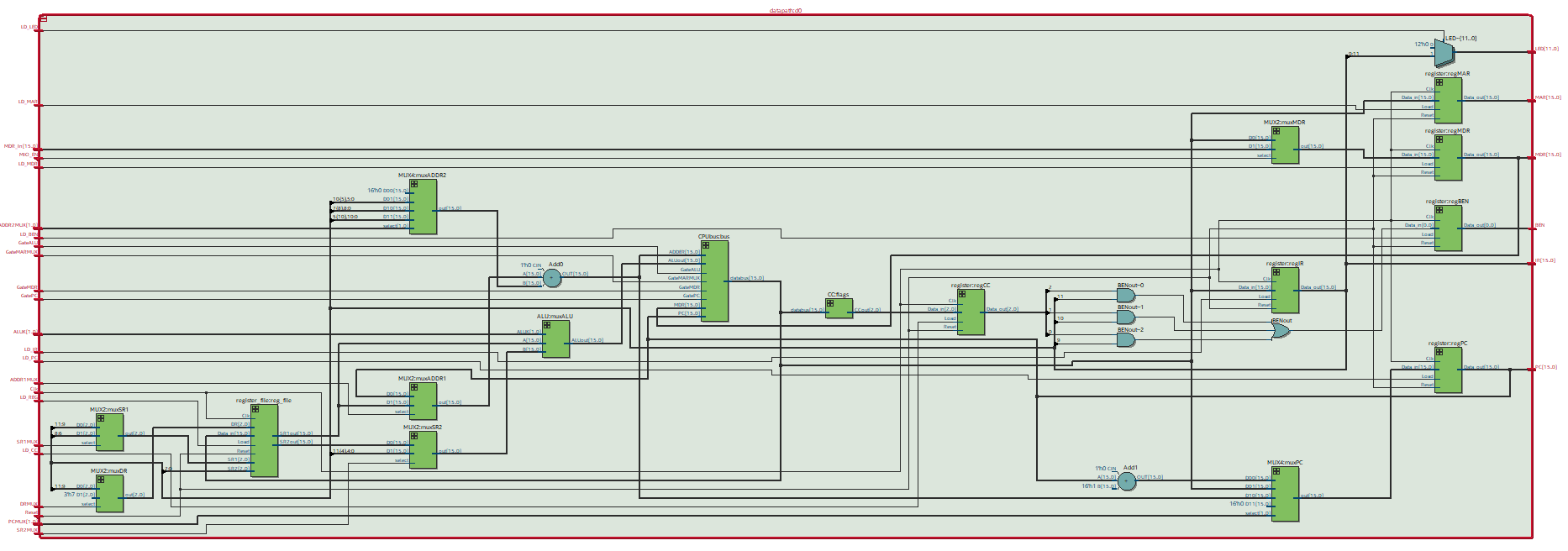
## Module: datapath from *datapath.sv*

Inputs: Clk, Reset, [15:0]MDR\_In, GatePC, GateMDR, GateALU, GateMARMUX, LD\_MAR, LD\_MDR, LD\_IR, LD\_BEN, LD\_CC, LD\_REG, LD\_PC, LD\_LED, [1:0]PCMUX, [1:0]ADDR2MUX, [1:0]ALUK, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, MIO\_EN

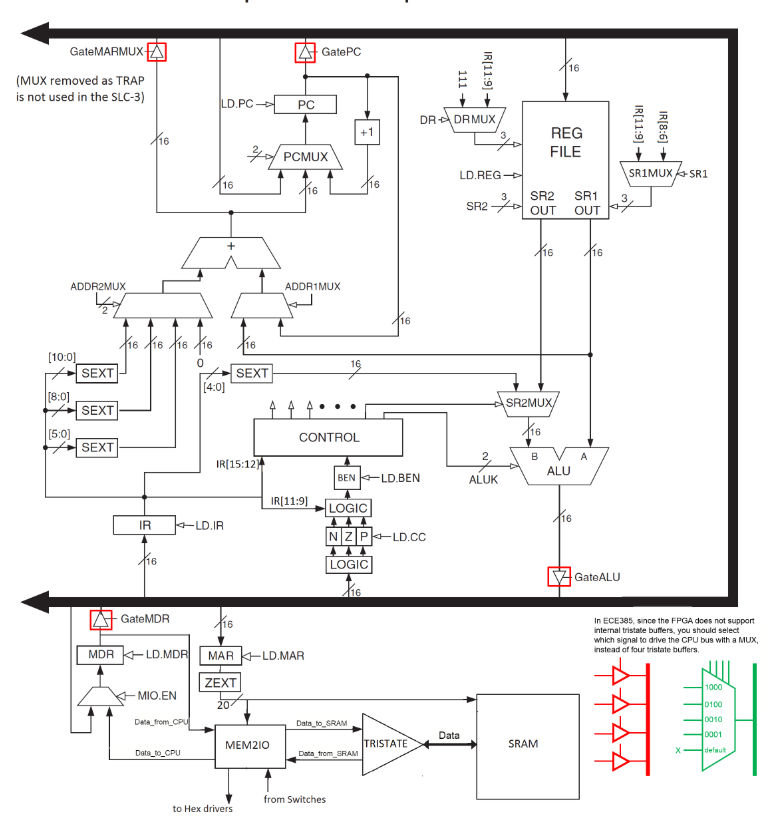
Outputs: [15:0]MAR, [15:0]MDR, [15:0]IR, [15:0]PC, BEN, [11:0]LED

Description: This module contains all of our registers, multiplexers and other logic elements necessary for our SLC-3. Additionally, it controls the LEDs on our device. Figure 5 is the actual block diagram for this module. A more condensed version in shown in Figure 6. This figure gives a more readable look on the actual arrangement of this module. In total, this module contains the MAR, MDR, IR, PC, BEN, CC, Register file, databus, ALU, PC MUX, DR MUX, SR1 MUX, SR2 MUX, ADDR1 MUX, ADDR2 MUX, MDR MUX, and address adder.

Purpose: This is used as a wrapper to contain everything related to our datapath in our SLC-3.



*Figure 5: Block Diagram of our Datapath*



*Figure 6: Datapath of SLC-3*

## 

## Module: lab6\_toplevel from *lab6\_toplevel.sv*

Inputs: [15:0]S, Clk, Reset, Run, Continue, [15:0]Data

Outputs: [11:0]LED, [6:0]HEX0, [6:0]HEX1, [6:0]HEX2, [6:0]HEX3, [6:0]HEX4, [6:0]HEX5, [6:0]HEX6, [6:0]HEX7, CE, UB, LB, OE, WE, [19:0] ADDR, [15:0]Data

Description: This is the top level module which contains the slc3, test\_memory, and synchronizers.

Purpose: This module is used as a top-level wrapper to connect the other modules to the inputs and outputs of the FPGA board. It also set up the virtual memory used in simulations.

## Module: slc3 from *slc3.sv*

Inputs: [15:0]S, Clk, Reset, Run, Continue, [15:0]Data

Outputs: [11:0]LED, [6:0]HEX0, [6:0]HEX1, [6:0]HEX2, [6:0]HEX3, [6:0]HEX4, [6:0]HEX5, [6:0]HEX6, [6:0]HEX7, CE, UB, LB, OE, WE, [19:0]ADDR, [15:0]Data

Description: This contains the high level elements of our SLC-3, such as our memory subsystem, tristate buffer, state controller, and datapath. Additionally, it sets up our hex display to show the data form Mem2IO on one of the displays and the PC on the other display.

Purpose: This is used as a wrapper to hold all of our high level elements of the SLC-3.

## Module: tristate from *tristate.sv*

Inputs: Clk,tristate\_output\_enable, [15:0]Data\_write, [15:0]Data

Outputs: [15:0]Data\_read, [15:0]Data

Description: This is a generic tristate buffer for the SRAM.

Purpose: This is used for communication between the SRAM and Mem2IO.

## Module: memory\_contents from *memory\_contents.sv*

Inputs: none

Outputs: none

Description: This module holds the instructions to test specific codes.

Purpose: This is used as the off-chip memory within simulations.

## Module: test\_memory from *test\_memory.sv*

Inputs: Clk, Reset, [15:0]I\_O, [19:0]A, CE, UB, LB, OE, WE

Outputs: none

Description: This simulates the behavior of the SRAM.

Purpose: This module is for simulation, as it is guaranteed to work at least as well as the actual memory.

Module: sync from *Synchronizers.sv*

Inputs: Clk, d

Outputs: q

Description: This is a positive-edge triggered 1-bit flip-flop that is used to synchronize the inputs with the rest of the circuit. On the positive edge of Clk, data from d is loaded into q.

Purpose: This module is used to synchronize the inputs from the FPGA board to the logic inside the circuit by using flip-flops synced up to the clock.

Module: testbench from *testbench.sv*

Inputs: None

Outputs: None

Description: This is a testbench used to simulate the waveforms of the simplified microprocessor.

Purpose: This module is used to test the written code to see if it is operating as expected.

## Description of the Operation of the ISDU (Instruction Sequence Decoder Unit)

The ISDU is the control unit of the SLC-3. It is a Moore state machine with 27 states. As such the outputs of the state machine are determined by the current state and not any of the inputs. This ISDU includes a decoder that determines what instructions to execute. It also controls when and what is loaded in the registers, selects the outputs of the multiplexers, and determines what is outputted onto the data bus. Additionally, it decides if we need to read or write to the SRAM. In total, the ISDU controls 21 different signals to determine the operation of the SLC-3. Figure 6 shows a simplified version of our state machine. For each state, it includes the exact operation that is performed. This gives a more readable overview of what occurs in each state. However, it does not account for the time we need to wait when we access memory. This is accounted for in Figure 6, which gives a more detailed look at the outputs of each state in each state bubble. In Figure 6, all of the extra states for waiting for the memory access to finish are included. Because our memory subsystem does not indicate when memory access is complete, this delay from cycling through these extra states are necessary to ensure that operation to load and store our data is fully complete.

In normal operation, our state machine will continue on with the FETCH-DECODE-EXECUTE cycle indefinitely after the Run pushbutton is pressed. One detail that was skipped in the making of Figure 6 is the synchronous reset. When the system is reset, the state machine will return to the Halted state in the next clock cycle regardless of which state it is currently in. This detail was omitted to make the state machine more readable when determining the next state of the state machine.

## 

## State Diagram of ISDU

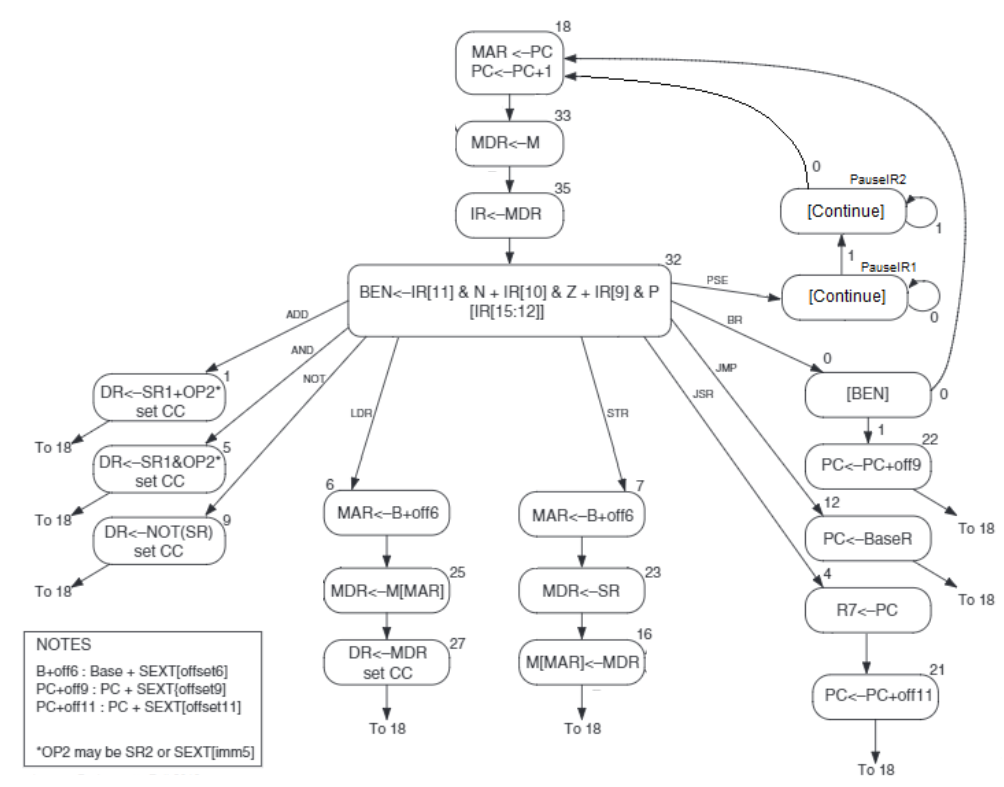


Figure 7: Simplified State Diagram of Instruction Sequence Decoder Unit

## 

Figure 8: State Diagram of Our Instruction Sequence Decoder Unit with Outputs

# Simulations of SLC-3 Instructions

The waveforms below in Figure 9 demonstrate the ADDi, LDR, and JMP instructions. It first adds the value 6 to Register 0, which demonstrates our ability to add and set the status register properly. It then loads the value at address 0x00003, which contains the value 0xFFFF. Afterwards, it does JMP instruction using R0 as a base register. It does an unconditional jump to a Pause instruction, which is incomplete in our simulation.

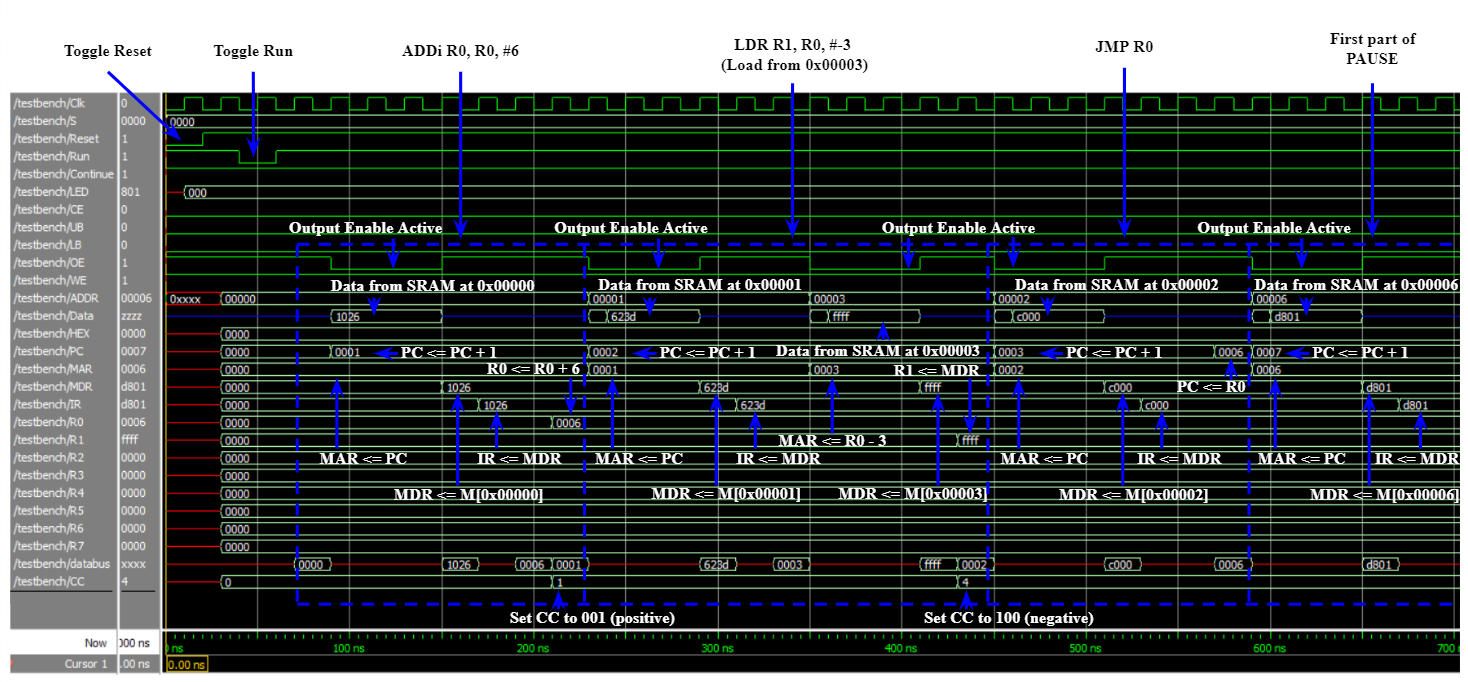


Figure 9: Waveforms of the ADDi, LDR, and JMP Instructions

# Post-Lab Questions

## Design Resources and Statistics Table

|  |  |
| --- | --- |
| **LUT** | 565 |
| **DSP** | 0 |
| **Memory (BRAM)** | 0 |
| **Flip-Flop** | 302 |
| **Frequency (MHz)** | 66.19 |
| **Static Power (mW)** | 98.66 |
| **Dynamic Power (mW)** | 8.2 |
| **Total Power (mW)** | 179.7 |

Figure 10: Design Statistics Table

## The main function of the Mem2IO

The main function of the Mem2IO is in its management of the input and output with the DE2 physical input and output devices. It controls the read-and-write accesses to SRAM. Additionally, it will accommodate for loading switches and storing information to the Hex display if the memory address to access is 0x0FFFF.

## The difference between BR and JMP instructions

The difference between BR and JMP instructions is in its allowance of conditions. On an instruction summary, it is written that if any of the condition codes match the condition stored in the status register, the code will take the branch. On the other hand, jump instruction simply copies the memory address from BaseR to PC - i.e. it is unconditional. JMP might look and function very similar to the BR function if NZP is set to 111, but there is one huge difference that discriminates BR from the NZP. From the detailed operation from the SLC-3.2 ISA, it is noticeable that the value of new PC is limited to the value of the old PC with some offset since the program branches to the location specified by adding the sign-extended value of PCoffset9, which is 9-bit sign-extended 2’s complement as aforementioned, to the PC. On the contrary, JMP can let the program jump to the location specified by the contents of the BaseR.

## Purpose of the R signal in Patt and Patel and its Compensation in our Design

R (Ready) signal is an asynchronous output from the memory to signal to the processor that the read/write instruction has completed and (for a read instruction) that data has been written to the tristate buffer. This can be seen from the given state machine of the LC-3; R states are utilized when there was a read/write for MAR and MDR. However, due to the lack of this signal in this implementation, an alternate solution of just simply waiting a few additional clock cycles for the operation to occur was implemented, as *generally* this is enough time for the operation to successfully finish.

# Conclusion

## The Functionality of the Design

## The main bug that we encountered in this experiment is synchronizing all of our operations in our datapath. Originally, we had all of our multiplexers placed in an always\_comb statement. However, this created race conditions in our code that made our SLC-3 occasionally break. We first found out about these race conditions when testing our code in simulation. We found that our code was consistently outputting the wrong behavior, especially when loading data from our databus. To counter, we pull out all of the intermediate connections to everything in our datapath and found that we were often were loading in data from the data bus before it was ready. To solve this issue, we made multiplexer modules and placed all our multiplexers in parallel by insinuating a module for each. This eliminated the race condition in our code, fixing our bug.

In the end, the circuit and all of its functions worked as expected. Throughout this lab, not only we were able to learn to design as requested from the objective, but we were also able to attain a way to adjust the code based on the bug encountered, which is in other words, being able to utilize the strength of SystemVerilog to repair any possible glitches and errors. Additionally, this lab reminded us what we learned back in ECE 220, especially how the LC-3 functions and the existence of R signal to ensure the stable, correct output. We were able to realize how R signal worked on the LC-3, and how to implement such signal using an alternative methods, such as by allocating more clock cycles instead of R signal.

## Ambiguity and Incorrectness in the Lab Manual

The updated datapath and the given datapath control signals have some inconsistencies in terms of the select bits to the multiplexers. This often led to confusion for things such as the PCMUX when determining the output signals for our ISDU. This can be improved by ensuring that the input ordering matches up correctly in both resources.