ECE385

Spring 2020

Experiment #6

# Simple Computer SLC-3.2 in SystemVerilog

Yuantao Lu, Yizhen Lu

Section ABD

TA: Nicholas Cebry, Wenjie Pan

## Introduction

In this lab, we use SystemVerilog to build a microprocessor using the model of a simplified version of 16-bit LC-3 (SLC-3). This processor follows the Von-Neumann model that has three execution cycles: fetch, decode and execute. The structure of this processor contains a Program Counter, an Instruction Register, a Memory Address Register, a Memory Data Register, a Decoder, a status register, a 8x16 regfile, and an Arithmetic Logic Unit. It can perform simple arithmetic and logical operations of 16-bit and store or load into memory. It can also perform condition checks and run simple programs.

## Written Description and Diagrams of SLC-3

### *Summary of Operation*

There are eleven operations that the SLC-3 can perform: ADD, ADDi, AND, ANDi, NOT, BR, JMP, JSR, LDR, STR, PAUSE. All such instructions follow the fetch, decode and execute cycle.

### *Instructions of SLC-3*

Specifically, the operation ADD adds values of two registers together and stores the result in a destination register; ADDi adds a sign extended 5 bit number with the value in one source register and stores the result into another register; AND ands values in two registers and store the value into another. ANDi ands sign extended 5 bit number with the value in one source register and store the result into another; NOT invert the value in one register and store the result into the destination register; BR checks the condition code and moves PC following the

9-bit PCoffset; JMP gives PC the value in a register; JSR stores the current PC value and adds to the PC an offset that is sign extended from a 11-bit number; LDR loads the value that is specified by the address value in a register plus an six bit offset into a destination register; STR stores a value in a register into an address specified by the value of a register and a six bit offset; PAUSE halt the program that is used for user input and debugging.

*The way SLC-3 performs functions*.

The operations of functions in SLC-3 are controlled by the control unit - the state machine of the SLC3.

In general, the Fetch cycle is composed of 3 states that load the desired opcode into IR. In the first state, the state 18, MAR will hold the address of the current PC value and PC would increment and point to the next address.In the second state, the state 33, the memory unit would read the value of address in MAR and load it into MDR. After the value in MDR is correct, in state 35, IR will be loaded with the value in MDR, which is the code of an instruction.

The decode state is state 32. In this state, the BEN value was set by comparing the current NZP value and the IR[11:9]. This is only useful with the instruction BR because the BEN value determines whether the BR instruction should be taken or ignored. Then IR[15:12] that is used for the instruction code is checked and the state will go to the one corresponding to the instruction code.

Generally, there are different execution cycles corresponding to different instructions. They usually have a different path through the state machine. After the instruction was done, the state will go back to state 18 and start fetching the next line of instruction.

*Block Diagram of slc3.sv*

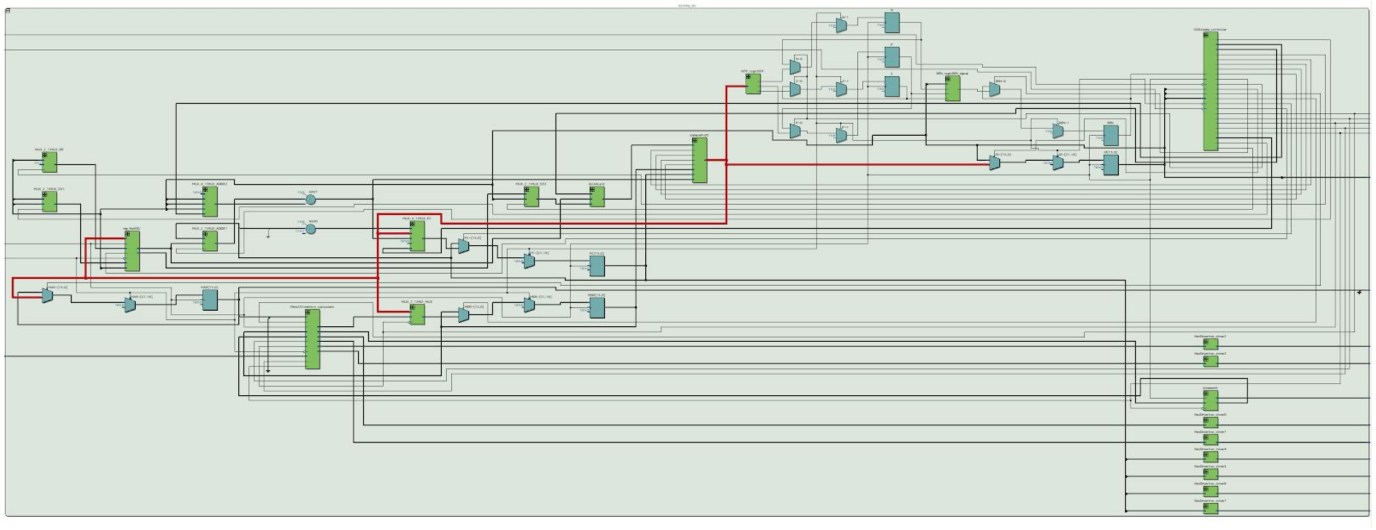


*Figure 1. Overview of  slc3.sv*

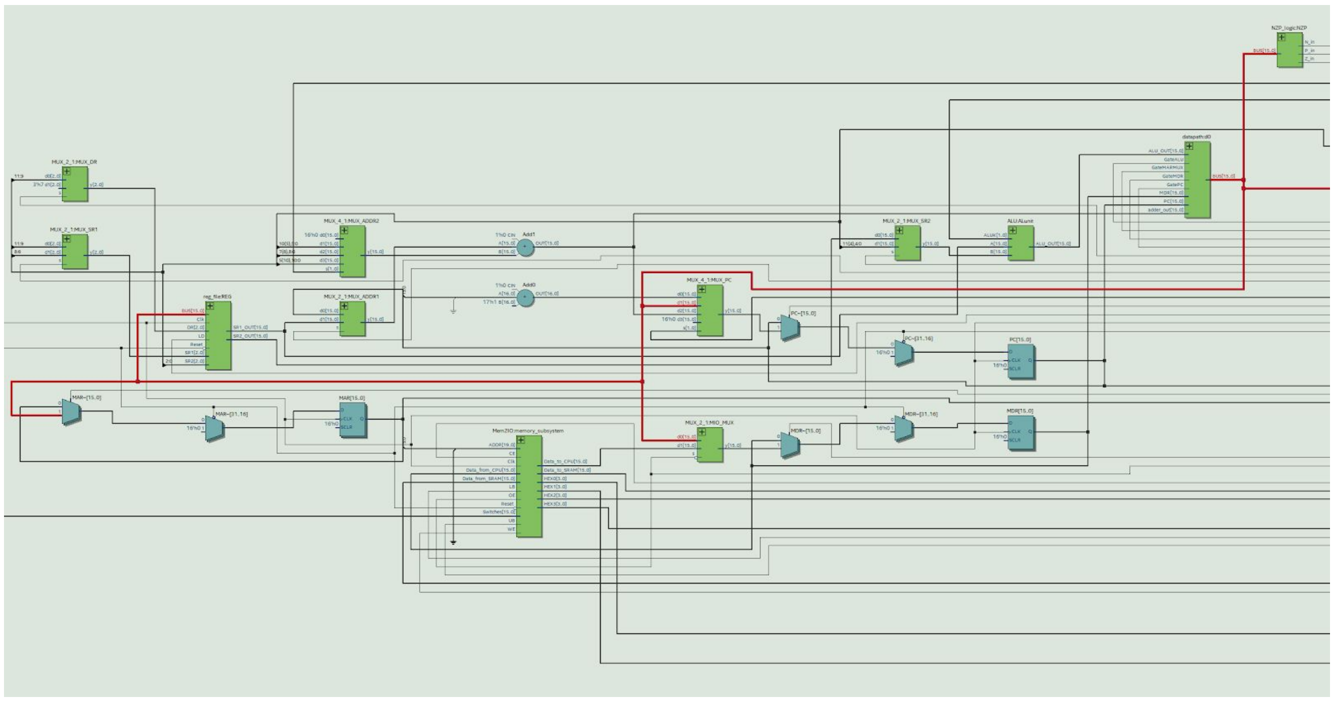


*Figure 2. Left half of the slc3.sv(clearer picture)*

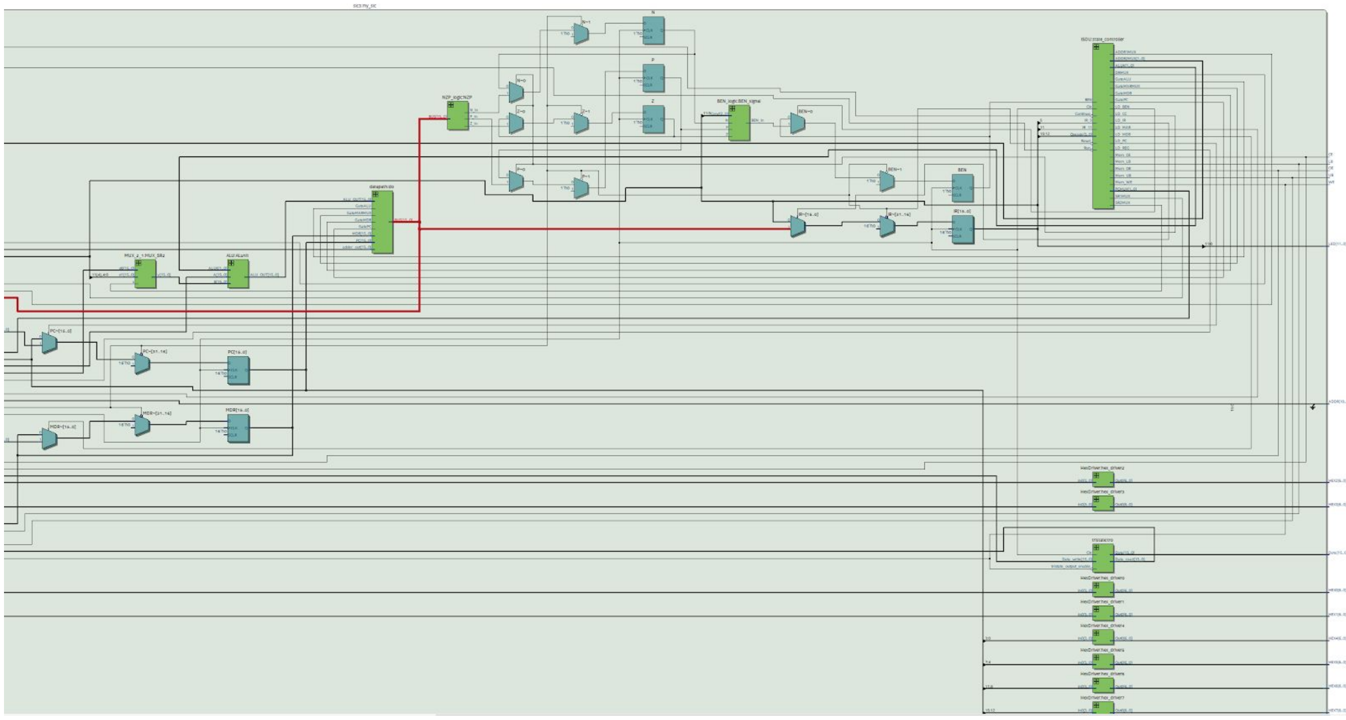


*Figure 3. Right half of slc3.sv(clearer picture)*

We did not create modules for the units of the SLC-3 explicitly. Instead, we used registers to represent the important units like PC, IR, MAR, MDR, and connected them using combinational logic.

<u>*Written Description of all Modules*</u>

Module: *lab6_toplevel.sv*

Inputs: *[15:0] S, [15:0] Data,*

*Clk, Reset, Continue, Run*

Outputs: *[11:0] LED, [19:0] ADDR,*

*[6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,*

*CE, UB, LB, OE, WE*

Description: This is the top level of this project, which connects the SLC-3 CPU module and the test memory module together.

Purpose: This is the top level that connects the input from the board to the CPU and connects the test money to provide memory block when running the simulation.

Module: *slc.sv*

Inputs: *[15:0] S, [15:0] Data,*

*Clk, Reset, Continue, Run*

Outputs: *[11:0] LED, [19:0] ADDR,*

*[6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,*

*CE, UB, LB, OE, WE*

Description: This is the integrated platform for the SLC-3 CPU. When *Reset* is active(Low), all data in the registers are cleared to 0. It takes input from the switches and the SRAM for addresses and instructions. Its outputs include 12 digit LEDs and 20 bit signal as MAR address to fetch data from SRAM.

Purpose: This module has all functions supported by the CPU design. It also enables the CPU to interact with the FPGA board.

Module: *NZP_logic.sv*

Inputs: *[15:0] BUS*

Outputs: *N_in, Z_in, P_in*

Description: This module provides the NZP signal based on the current BUS value, and would then be loaded into NZP registers if needed.

Module: *BEN_logic.sv*

Inputs: *N, Z, P, [2:0] Cond*

Outputs: *BEN_in*

Description: This module generates BEN signal value from judging the condition code provided in the instruction and current NZP values. As long as there's a match, *BEN_in* would be High.

Purpose: This module prepares the *BEN_in* signal to be loaded into the BEN register at the decoding stage.

Module: *HexDriver.sv*

Inputs: *[3:0] In0*

Outputs: *[6:0] Out0*

Description: This module transfers input binary numbers into their corresponding seven-bit binary sequences for LED display.

Purpose: This module helps transfer the data stored in shift registers into Hexadecimal form and into a form that can be displayed on LED and easy to read.

Module: *reg_file.sv*

Inputs: *LD, Clk, Reset, [2:0] DR, SR1, SR2, [15:0] BUS*

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

Description: This is the register file of the CPU. It is made up of eight 16-bit registers, a decoder and two 8-to-1 MUXes. The module is capable of storing data in the BUS to the required register, as well as to read data out of the indicated register. To store data, switch *DR* to point to the destination register, and turn on *LD*, then the data in the BUS would be copied into the register. To read data, simply use either *SR1* or *SR2* to point to the desired register, and the data would be passed out through output ports *SR1_OUT* and *SR2_OUT*. Anytime the *Reset* is active, all data would be cleared to 0 at next clock cycle.

Purpose: The module contains the register file to stage data between memory and the functional units on the chip. It can temporarily store some data between operations.

Module: *MUX_2_1 #(parameter width = 16)*

Inputs: *s, [width-1:0] d0, d1*

Outputs: *[width-1:0] y*

Description: This is a 2-to-1 MUX which outputs *d0* when *s* is 0 and outputs *d1* when *s* is 1.

Module: *MUX_4_1 #(parameter width = 16)*

Inputs: *[1:0] s, [width-1:0] d0, d1, d2, d3*

Outputs: *[width-1:0] y*

Description: This is a 4-to-1 MUX whose output depends on the 2-bit select signal*s*.

Module: *MUX_8_1*

Inputs: *[2:0] s, [15:0] d0, d1, d2, d3, d4, d5,d6, d7*

Outputs: *[15:0] y*

Description: This is an 8-to-1 MUX whose output depends on the 3-bit select signal *s*.

Module: *datapath.sv*

Inputs: *[15:0] PC, adder_out, ALU_OUT, MDR*

*GatePC, GateMDR, GateALU, GateMARMUX*

Outputs: *[15:0] BUS*

Description: This module is used to represent the BUS. BUS is used for It takes four inputs and four gate signals to determine which input signal should be loaded to the BUS. Only one of the four gate signals would be high at the same time, so just pass the corresponding input to the BUS.

Purpose: The datapath is used to transfer data and control the data flow throughout different CPU components.

Module: *ALU.sv*

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

Outputs: *[15:0] ALU_OUT*

Description: This module is the arithmetic-logic unit of the CPU. It performs arithmetic or bitwise operation on two operands *A, B.* In specific, it can do A+B, A&B, Not A, and Pass A. Signal *ALUK* determines which operation to perform, and the result would be the output.

Purpose: This module enables the CPU to perform arithmetic and logic operations to better handle the instructions.

Module: *tristate.sv (N = 16)*

Inputs: *Clk, tristate_output_enable, [N-1:0] Data_write*

Outputs: *[N-1:0] Data_read*

Bidirectional ports: *[N-1:0] Data*

Description: The tri-state buffer serves as the interface between Mem2IO and SRAM. It also controls when an output signal can be passed into the BUS.

Module: *Mem2IO.sv*

Inputs: *Clk, Reset, CE, UB, LB, OE, WE, [19:0] ADDR,*

*[15:0] Switches, Data_from_CPU, Data_from_SRAM*

Outputs: *[15:0] Data_to_CPU, Data_to_SRAM,*

*[3:0] HEX0, HEX1, HEX2, HEX3*

Description: This is the SRAM and I/O controller. It is used as an interface between the CPU, the SRAM and the input switches on the FPGA board. It can pass data from the input switches to the SRAM or to the CPU as if they were either data from the instruction code, or as the data stored in the SRAM. This helps us manipulate the machine to do indicated tests more easily.

Module: *memory_contents.sv*

Outputs: *[15:0] mem_array[0:size-1] (size=256)*

Purpose: This module provides a block of memory area which is filled with instructions that form a variety of tasks for the CPU to perform.

Module: *test_memory.sv*

Inputs: *Clk, Reset, CE, UB, LB, OE, WE, [19:0] A*

Bidirectional Ports: *[15:0] I_O*

Description: This module provides a memory block that has similar behavior to the SRAM IC on the FPGA board.  It is used to run the simulations. In simulation, this memory works similar to the actual memory so that we can test our SLC design. It is synthesized into a blank module since test memory is not synthesizable. The module interacts with the *memory_contents.sv* file to read and write data into the file.

Package: *SLC_2.sv*

Description: This is a library reference file that is able to transfer all the command instructions in *memory_contents.sv* file into binary machine codes that the CPU can understand.

*Description of the operation of ISDU*

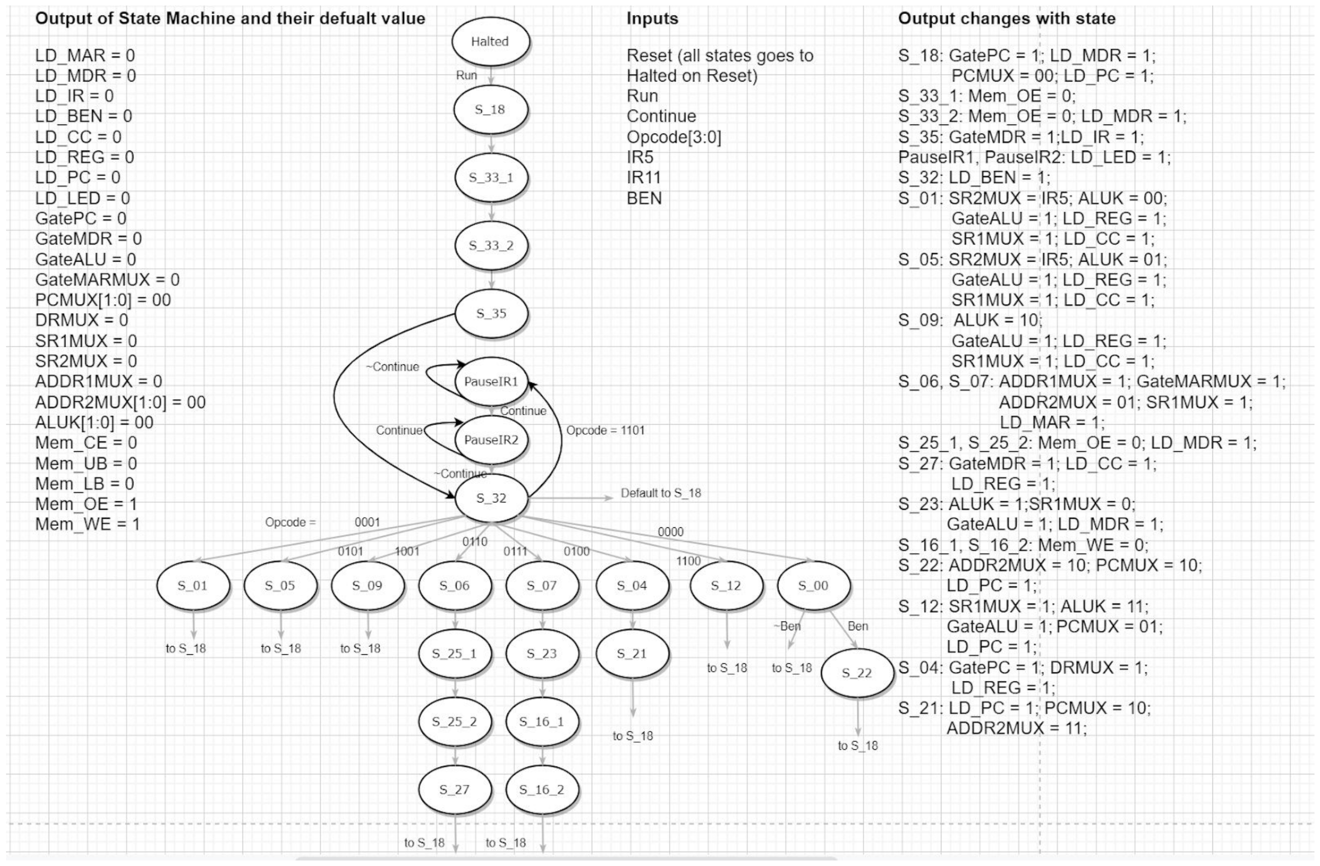
Module: ISDU.sv

Inputs: 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, Gate_PC, Gate_MDR, Gate_ALU, 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 module serves as the control unit of the SLC-3. It has several outputs that serve as the selecting bits for MUXes in the SLC-3. This module is mainly composed of a state machine that is represented by the diagram below. The input group is relatively small when compared to output, but it serves as the essential of state transitions for the state machine. When not running, the control state is always in the Halted until the Run signal is on high on a clock edge. When the reset signal is high on a clock edge, no matter what state the controller is in, the next state will always be the Halted state. Note that this SLC-3 supports the instruction of PAUSE, which is usually used for user input. We have two pause states that are served for PAUSE. The decode state (S_32) looks at the four bit input Opcode that is coming from IR. IR_5 is used to determine between ADD, ADDi and AND, ANDi. BEN is used for Opcode BR and determines whether we need to branch. In the execution states, the output is determined by states. A sequence of states in the execution cycle is often representing an instruction. After such a sequence, the states will go back to the starting state of fetch(S_18). The execution sequence starts at S_01 corresponds to instructions ADD, ADDi; The execution sequence starts at S_05 corresponds to AND, ANDi; the sequence of S_09 corresponds to NOT; the S_06 sequence corresponds to LDR and the S_07 sequence corresponds to STR; the S_04 sequence corresponds to JSR and S_12 sequence corresponds to JMP; the S_00 corresponds to BR. See the State Diagram below for detail.

*State Diagram of ISDU*



**Output of State Machine and their defualt value**

LD_MAR = 0
LD_MDR = 0
LD_IR = 0
LD_BEN = 0
LD_CC = 0
LD_REG = 0
LD_PC = 0
LD_LED = 0
GatePC = 0
GateMDR = 0
GateALU = 0
GateMARMUX = 0
PCMUX[1:0] = 00
DRMUX = 0
SR1MUX = 0
SR2MUX = 0
ADDR1MUX = 0
ADDR2MUX[1:0] = 00
ALUK[1:0] = 00
Mem_CE = 0
Mem_UB = 0
Mem_LB = 0
Mem_OE = 1
Mem_WE = 1

**Inputs**

Reset (all states goes to Halted on Reset)
Run
Continue
Opcode[3:0]
IR5
IR11
BEN

**Output changes with state**

S_18: GatePC = 1; LD_MDR = 1;
PCMUX = 00; LD_PC = 1;
S_33_1: Mem_OE = 0;
S_33_2: Mem_OE = 0; LD_MDR = 1;
S_35: GateMDR = 1;LD_IR = 1;
PauseIR1, PauseIR2: LD_LED = 1;
S_32: LD_BEN = 1;
S_01: SR2MUX = IR5; ALUK = 00;
GateALU = 1; LD_REG = 1;
SR1MUX = 1; LD_CC = 1;
S_05: SR2MUX = IR5; ALUK = 01;
GateALU = 1; LD_REG = 1;
SR1MUX = 1; LD_CC = 1;
S_09: ALUK = 10;
GateALU = 1; LD_REG = 1;
SR1MUX = 1; LD_CC = 1;
S_06, S_07: ADDR1MUX = 1; GateMARMUX = 1;
ADDR2MUX = 01; SR1MUX = 1;
LD_MAR = 1;
S_25_1, S_25_2: Mem_OE = 0; LD_MDR = 1;
S_27: GateMDR = 1; LD_CC = 1;
LD_REG = 1;
S_23: ALUK = 1;SR1MUX = 0;
GateALU = 1; LD_MDR = 1;
S_16_1, S_16_2: Mem_WE = 0;
S_22: ADDR2MUX = 10; PCMUX = 10;
LD_PC = 1;
S_12: SR1MUX = 1; ALUK = 11;
GateALU = 1; PCMUX = 01;
LD_PC = 1;
S_04: GatePC = 1; DRMUX = 1;
LD_REG = 1;
S_21: LD_PC = 1; PCMUX = 10;
ADDR2MUX = 11;

*Figure 4. State Diagram of ISDU*

## Simulation of SLC-3 instructions

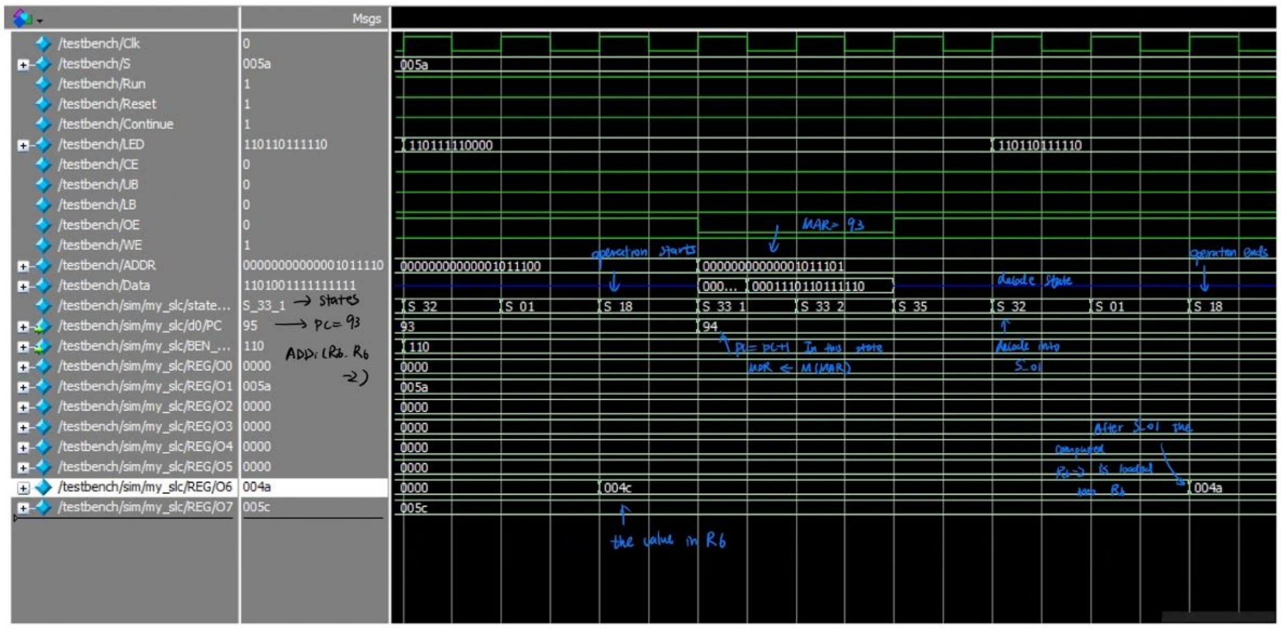

/testbench/Clk 0
/testbench/S 005a
/testbench/Run 1
/testbench/Reset 1
/testbench/Continue 1
/testbench/LED 110110111110
/testbench/CE 0
/testbench/UB 0
/testbench/LB 0
/testbench/OE 0
/testbench/WE 1
/testbench/ADDR 0000000000000001011110
/testbench/Data 1101001111111111
/testbench/sim/my_slc/state... S_33_1 → states
/testbench/sim/my_slc/d0/PC 95 → PC = 93
/testbench/sim/my_slc/BEN_... 110
/testbench/sim/my_slc/REG/O0 0000
/testbench/sim/my_slc/REG/O1 005a
/testbench/sim/my_slc/REG/O2 0000
/testbench/sim/my_slc/REG/O3 0000
/testbench/sim/my_slc/REG/O4 0000
/testbench/sim/my_slc/REG/O5 0000
/testbench/sim/my_slc/REG/O6 004a
/testbench/sim/my_slc/REG/O7 005c
ADDi (R6, R6 -2)
S 32, S 01, S 18, S 33 1, S 33 2, S 35, S 32, S 01, S 18
operation starts
MAR = 93
PC = PC+1 In this state MDR ← M(MAR)
decode state
decode into S_01
After S_01 the computed R6 → is loaded into R6
operation ends
the value in R6

*Figure 4. Simulation of ADDi (ADDi(R6, R6, -2))*

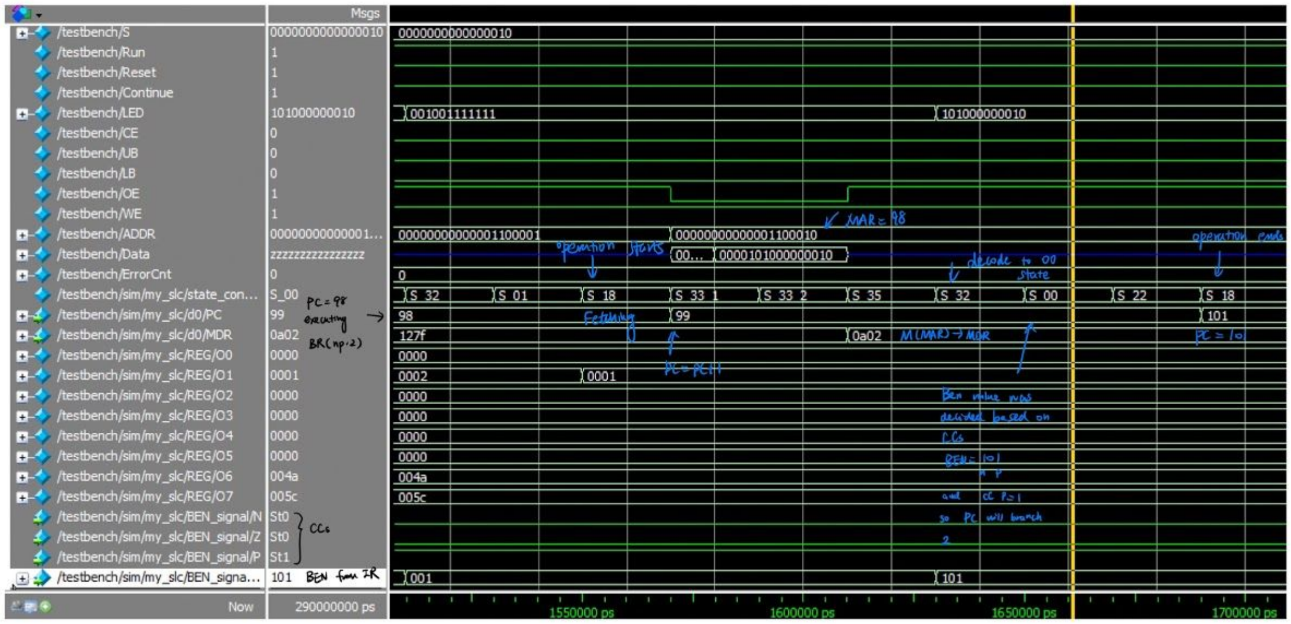
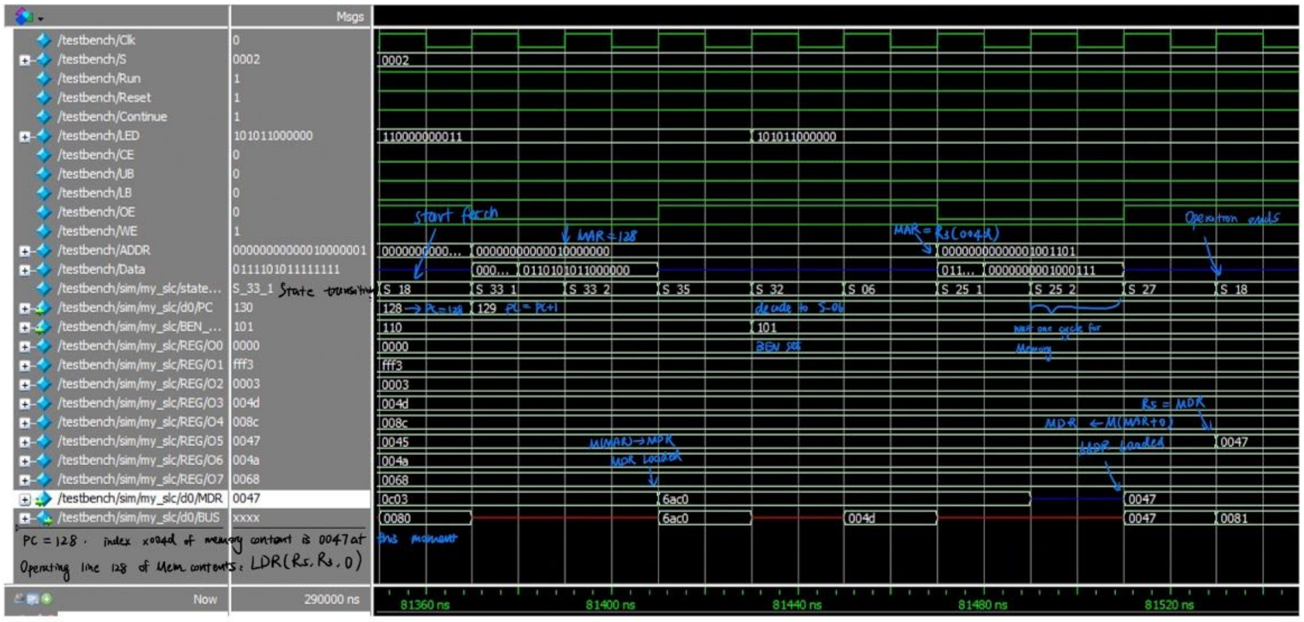

/testbench/S 0000000000000010
/testbench/Run 1
/testbench/Reset 1
/testbench/Continue 1
/testbench/LED 101000000010
/testbench/CE 0
/testbench/UB 0
/testbench/LB 0
/testbench/OE 1
/testbench/WE 1
/testbench/ADDR 00000000000001...
/testbench/Data zzzzzzzzzzzzzzzz
/testbench/ErrorCnt 0
/testbench/sim/my_slc/state_con... S_00
/testbench/sim/my_slc/d0/PC 99
/testbench/sim/my_slc/d0/MDR 0a02
/testbench/sim/my_slc/REG/O0 0000
/testbench/sim/my_slc/REG/O1 0001
/testbench/sim/my_slc/REG/O2 0000
/testbench/sim/my_slc/REG/O3 0000
/testbench/sim/my_slc/REG/O4 0000
/testbench/sim/my_slc/REG/O5 0000
/testbench/sim/my_slc/REG/O6 004a
/testbench/sim/my_slc/REG/O7 005c
/testbench/sim/my_slc/BEN_signal/N St0
/testbench/sim/my_slc/BEN_signal/Z St0
/testbench/sim/my_slc/BEN_signal/P St1
/testbench/sim/my_slc/BEN_signa... 101 BEN from IR
CCs
PC = 98 executing → BR(np, 2)
S 32, S 01, S 18, S 33 1, S 33 2, S 35, S 32, S 00, S 22, S 18
operation starts
MAR = 98
Fetching
PC = PC+1
M(MAR) → MDR
decode to 00 state
Ben value was decided based on CCs BEN = 101 n p and cc P = 1 so PC will branch 2
operation ends
PC = 101
Now 290000000 ps
1550000 ps
1600000 ps
1650000 ps
1700000 ps

*Figure 5. Simulation of BR (BR(np, 2))*


Msgs
/testbench/Clk 0
/testbench/S 0002
/testbench/Run 1
/testbench/Reset 1
/testbench/Continue 1
/testbench/LED 101011000000
/testbench/CE 0
/testbench/UB 0
/testbench/LB 0
/testbench/OE 0
/testbench/WE 1
/testbench/ADDR 0000000000010000001
/testbench/Data 0111101011111111
/testbench/sim/my_slc/state... S_33_1
/testbench/sim/my_slc/d0/PC 130
/testbench/sim/my_slc/BEN_... 101
/testbench/sim/my_slc/REG/O0 0000
/testbench/sim/my_slc/REG/O1 fff3
/testbench/sim/my_slc/REG/O2 0003
/testbench/sim/my_slc/REG/O3 004d
/testbench/sim/my_slc/REG/O4 008c
/testbench/sim/my_slc/REG/O5 0047
/testbench/sim/my_slc/REG/O6 004a
/testbench/sim/my_slc/REG/O7 0068
/testbench/sim/my_slc/d0/MDR 0047
/testbench/sim/my_slc/d0/BUS xxxx
Now 290000 ns
81360 ns
81400 ns
81440 ns
81480 ns
81520 ns
S 18
S 33 1
S 33 2
S 35
S 32
S 06
S 25 1
S 25 2
S 27
start fetch
MAR = 128
State transition
128 → PC = 128
129 PC = PC+1
M(MAR) → MDR
MDR Loaded
decode to S-06
BEN set
MAR = R3(004d)
wait one cycle for Memory
MDR ← M(MAR+0)
MDR loaded
R5 = MDR
Operation ends

PC = 128. index x004d of memory content is 0047 at this moment
Operating line 128 of Mem contents: LDR(R5, R3, 0)

*Figure 5. Simulation of LDR (LDR(R5, R3, 0))*

## Answer to Postlab Questions

1)

| | |
|---|---|
| LUT | 579 |
| DSP | N/A |
| Memory(BRAM) | 0 |
| Flip-Flop | 268 |
| Frequency | 61.42MHz |
| Static Power | 98.66mW |
| Dynamic Power | 7.89mW |
| I/O Thermal Power | 71.97mW |
| Total Power | 178.53mW |

2)

- MEM2IO is the SRAM and I/O controller. It is used as an interface between the CPU, the SRAM and the input switches on the FPGA board. It can pass data from the input switches to the SRAM or to the CPU as if they were either data from the instruction code, or as the data stored in the SRAM. Normally, its function is only to transmit data from memory block to the processor or the other way around. But when the MAR address is 0xFFFF(2's complement of decimal -1), the data being transferred then becomes the value indicated by the switches. This function is helpful when we want the processor to jump to a certain address to execute a specific program or to store certain value into some registers.
- BR is a conditional jump command, it only jumps to the desired address when meeting the expected NZP conditions. In contrast, JMP is an unconditional jump tha takes place as long as the processor executes this instruction. Another difference is that BR jumps based on the given offset, while JMP instruction directly gives the destination address to jump to.
- The R signal in Patt and Patel is provided by the memory unit and used to indicate that the data has already been stored into/ read out of the indicated address and it is safe to perform the next operation. In our design, we use an extra state to offer enough time for storing and loading data in order to compensate for the lack of the R signal. Without the R signal and Patt and Patel, we are assuming that the memory would be ready in that extra clock cycle we left for. If the memory is too slow and would not be ready in that extra clock cycle, the design would fail.

## Conclusion

In this lab, we learned the hardware side of LC3 and actually implemented what we learned from ECE120. We also learned about the timing of the design about what we need to do to make the design run at a relatively high frequency.

The functionality of our design is fine. Because of the slowness of memory access, our design runs at 61.42MHz, which is a little higher than 50MHz. Our design did not make one module per component. Instead, we used modules of registers to represent those components like PC, IR. As a result, our top level view of the design looked messy. To improve, we could write a certain group in one module for its combinational and sequential logic to make the top level look better.

We think this lab provides us with necessary material to implement the SLC-3. But it implemented the whole memory side for us including the tri-state buffer and Mem-IO. We think ECE385 could provide documents for how the memory worked and how to communicate with it in the future to let students implement the memory side by themselves.