ECE 385 Lab Report

Spring 2023

Lab 6: Simple Computer SLC-3.2 in SystemVerilog

Name: Zhu Hanggang, Hong Jiadong

Student ID: 3200110457, 3200110970

Prof. Chushan Li, Prof. Zuofu Cheng
ZJU-UIUC Institute
April 6, 2023

1 Introduction

In this experiment, we design a simplified version of the LC-3 microprocessor. The microprocessor contains a 16-bit Program Counter, 16-bit registers, 16-bit ALU, etc. It also has full functionality to interact with SRAM, user input switch, and output LEDs. It supports a subset of LC-3 instructions including ADD, AND, NOT, BR, JMP, JSR, LDR, STR, and Pause.

2 Written Description and Diagrams of SLC-3

2.1 Summary of Operation

SLC-3 supports basic arithmetic operations **ADD, AND, NOT**, They support operations both on the register and intermediate values. To control the instruction flow of the program, **BR, JMP, JSR** is supported. BR sets the PC to a certain address when required conditions are met, JMP directly set the PC to the required address and JSR will store the current address to R7 first and then set the PC to the required address, which is useful for function calls. Further, to interact with SRAM, **LDR, STR** instructions are supported. LDR reads memory content into the register and STR stores register content into memory. In order to allow the user to set switches before the operation and read output after the operation, **Pause** is supported.

2.2 SLC-3 Function Description

SLC-3 performs its function through mainly three operations: *Fetch, Decode, Execute*. Fetch, Decode states are all the same for all instructions and the only difference is Execute state. Once a cycle is completed, a new Fetch-Decode-Execute cycle will re-start for next instruction.

Fetch will read the current instruction from Memory in address PC and increment PC. Specifically, it will Load PC to MAR, take three cycles (three cycles are required to read from SRAM through tristate buffer and synchronizer) to read memory content at address MAR and load the content MDR. Then load MDR to Instruction register(IR) and increment PC.

Deocde will decide which operations should be executed by reading bits 15-12 of IR(opcode) using Instruction Decoder. More specifically, based on opcode, there will be different next state in ISDU.

Execute perform operation based on signals from Instruction Decoder and write the result to desination register or memory. Execute differs for each instruction and the operations of each instruction is listed below.

ADD: Choose DR, SR1, SR2 based on Instruction Decoder and choose add operation in ALU.Pass output from ALU to Register file. Set CC

ADDi: Similar to ADD, except that the second operand will be SEXT(imm5) based on bit 5 of IR (use a mux)

AND: Choose DR, SR1, SR2 based on Instruction Decoder and choose and operation in ALU. Pass output from ALU to register file, Set CC

ANDi: Similar to AND, except that the second operand will be SEXT(imm5) based on bit 5 of IR(use a mux)

NOT: Choose SR, DR based on Instruction Decoder and choose not operation in ALU. Pass output from ALU to Register file, set CC

BR: Evaluate BEN first in one state. If it's 0, go back to start state otherwise, choose ADDR2MUX and ADDR1MUX correctly to get new PC load new PC into PC register.

JMP: Directly choose BR using ADDR1 MUX and load the value to PC

JSR: Pass PC to bus and R7 read data from data bus, calculate new PC and load new PC just like JMP.

LDR: Pass BR + SEXT(offset6) into data bus and MAR will load this value. Take three states to read memory content at address MAR like FETCH After reading, MDR will load the value and pass it to register files

STR: Pass BR + SEXT(offset6) into data bus and MAR will load this value. Pass SR into data bus and MDR will load this value.Take two states to pass MDR to SRAM due to tristate buffer.

PAUSE: Do nothing (all Load signal is off) and wait until Continue is back to 0. Will Continue is set to 1 again, jump to start state S18 and start execution again

2.3 Block Diagram of slc3.sv

Figure 1 shows the block diagram of slc3.sv generated by RTL viewer. It is not very clear but it basically contains one datapath, ISDU, MEM2IO, tristate buffer, some synchronizers and hexdriver to display value.which is the main component of SLC3, Datapath is the main component of SLC3 and ISDU controls the logic of operations. MEM2IO and tristate is used to interact with user input and SDRAM.

To have a more clear look at the block diagram, please look at Figure 2

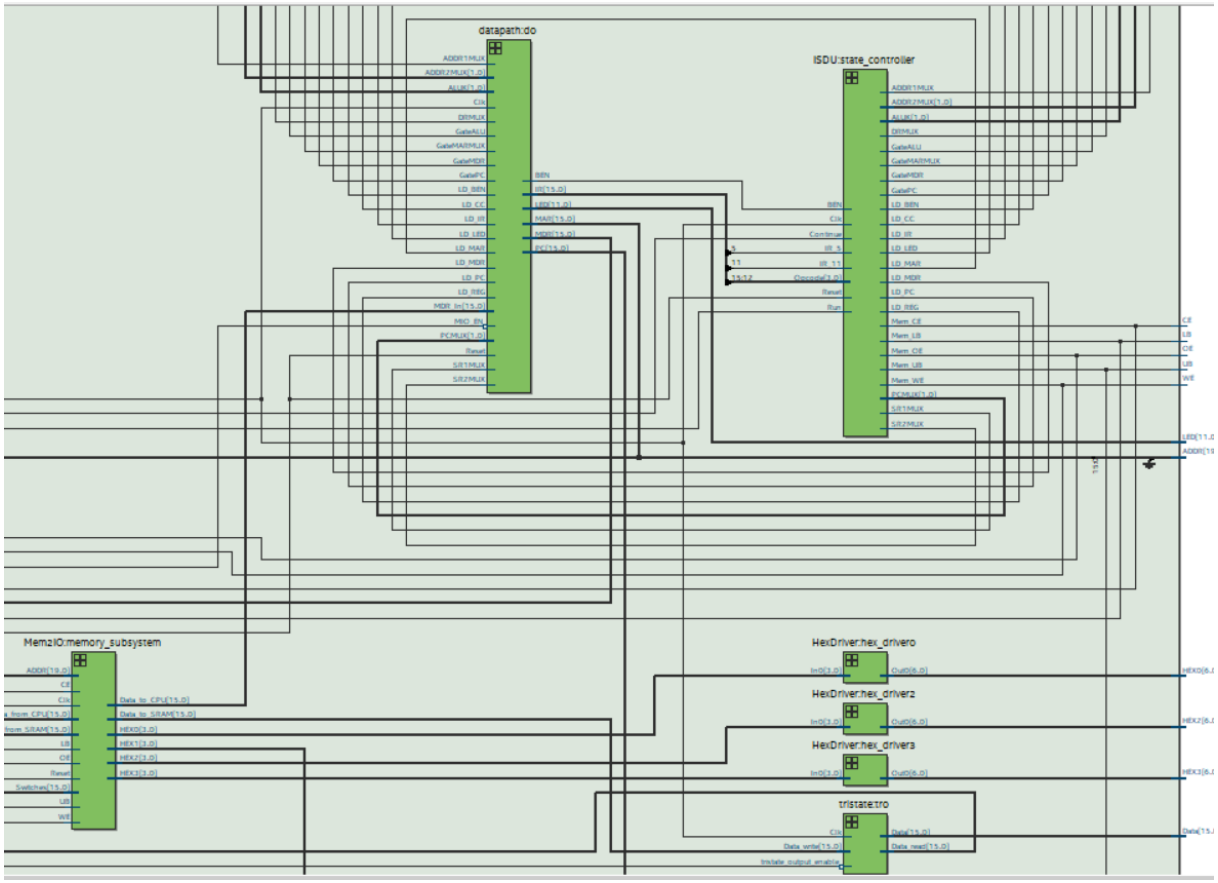

Figure 1: SLC-3 Diagram by generated by RTL viewer

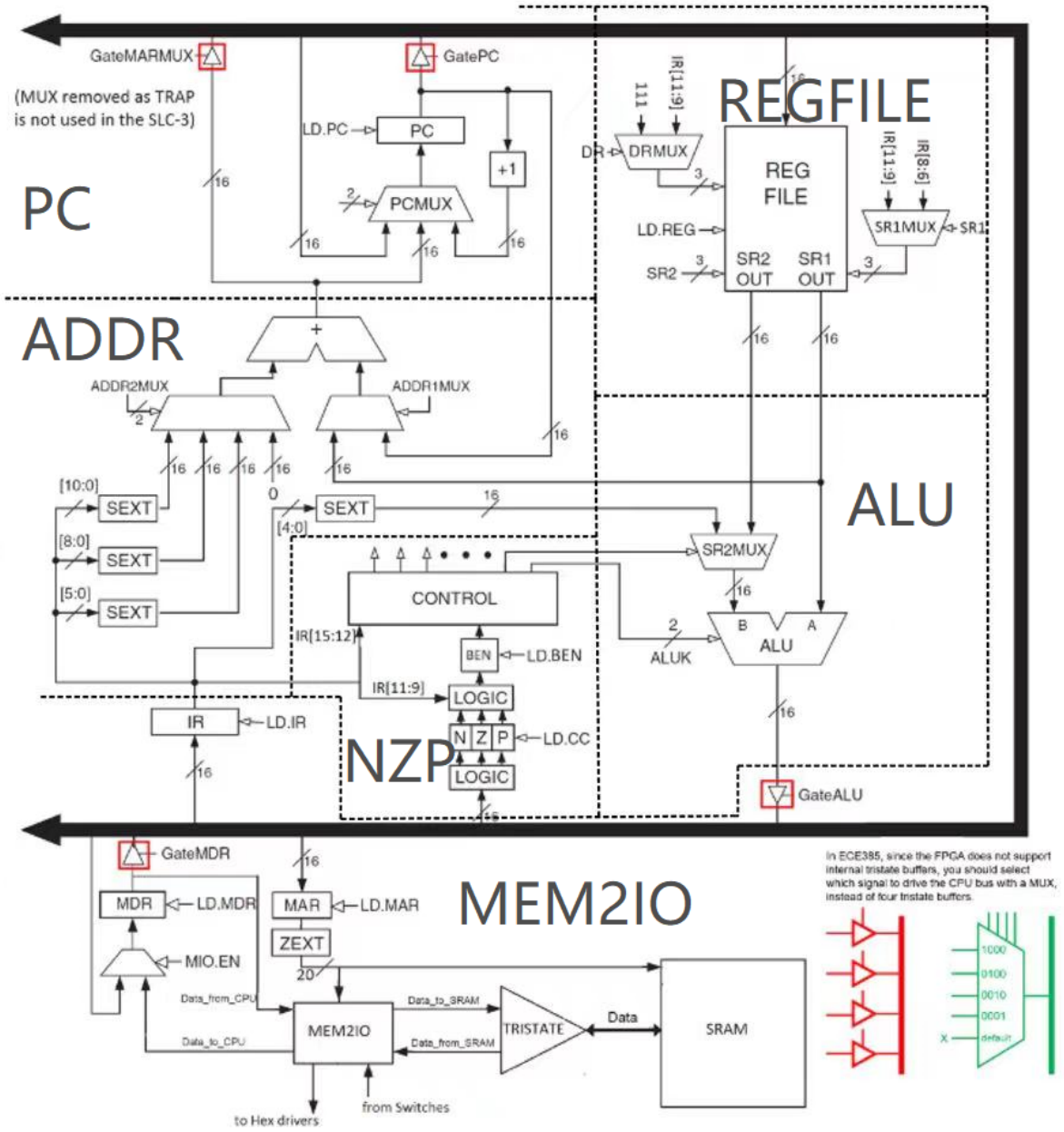

Figure 2: SLC-3 Datapath Division

2.4 Written Description of all .sv modules

2.4.1 Provided module: Overview and SDRAM interaction

The high level of SLC3 is already provided. The module for interacting with SDRAM is also provided. Even though we do not need to implement these, knowing how theses modules works helps us a lot understanding how SLC-3 works. In this section, we present all important modules in SLC3 here. Some smaller modules are described in next subsection.

1. **Module**: lab6_toplevel
   Inputs: [15:0] S, Clk,Run, Reset, Continue.
   Outputs: [11:0] LED, HEX, CE,UB,LB,OE,WE, ADDR
   Description: The toplevel of slc-3. Contains slc3 module and test-memory module. Also has syn-

chronizer for user inputs.
Purpose: Act as toplevel

2. **Module**: slc3
   Inputs: [15:0] S, Clk,Run,Reset, Continue
   Outputs: [11:0] LED, HEX, CE,UB,LB,OE,WE,ADDR
   Description: Contains all important parts of SLC3. Receives user inputs S, Run, Reset, Continue and pass them to ISDU. Depending on output from ISDU. Instruction is implemented in datapath module. SDRAM is connected using Mem2IO and tristate.
   Purpose: Act as the main part of SLC3

3. **Module**: test_memory
   Inputs: Clk, Reset, CE,UB,LB,OE,WE, Addr
   Outputs: I_O (data)
   Description: This module can't be synthesized. It is used to model the content of SDRAM memory content for simulation. It gives corresponding data for slc3 based on input CE,UB,LB,OE,WE and ADddr.
   Purpose: Act as substition of SDRAM in simulation

4. **Module**: datapath
   Inputs: Clk, Reset, all kinds of LOAD, Gate, MUX signal. MDR_In
   Outputs: BEN, [11:0] LED, [15:0] IR. [15:0] MAR, [15:0] MDR, [15:0] PC.
   Description: The module is responsible to do all operations. Given instruction from ISDU, it loads registers, operate ALU, calcualate BEN etc. More details of this module is in section **Datapath Design: datapath.sv, MUX.sv, register.sv**
   Purpose: To operate all operations of SLC3

5. **Module**: ISDU
   Inputs: Clk, Run, Continue, Reset. Opcode, IR_5, IR_11, BEN
   Outputs: all kinds of LOAD, Gate, MUX signal.
   Description: This is state machine to control the logic of operation. More details of this module can be found in section **Description of operation of ISDU**
   Purpose: To control the overall logic of operations

6. **Module**: Mem2IO
   Inputs: Clk, Reset, [19:0] Addr, CE, UB, LB, OE, WE, Switches, Data from CPU, Data from SRAM
   Outputs: Data to CPU, Data to SRAM, HEX
   Description: This manages all IO with the DE2 physical IO devices with slc3. IO for CPU is memory mapped and when address is set to 0xFFFF, data is read/written from/to Swtich/LED instead of SRAM.
   Purpose: To allow user to pass data to SLC3.

7. **Module**: tristate
   Inputs: Clk, tristate output enable, Data write
   Outputs: Data read, Data.
   Description: This module is between Mem2IO and SDRAM. Basically it interacts with SRAM and

is responsbile to write data from CPU to SRAM or read data from SRAM and pass to CPU.
Purpose: To interact with SRAM

2.4.2 Datapath Design: datapath.sv, MUX.sv, register.sv

The main work of this lab except for the control part is to develop a reasonable implementation of the data path of simplified LC-3 architecture. To achieve this goal, we mainly divide the data path into three System Verilog files, datapath.sv, MUX.sv, register.sv.

The implementation of the register.sv is to define the widely used register data structure in the FPGA circuit design. We found that every register used in the SLC-3 implementation can be represented as one abstract and length edible register. So we simply implement a very basic universal length-edible register.

The MUX.sv is basically the implementation of the structure of the signal transmission part that needs different choices to be made. In this part of the implementation, other than what we did in the register part, we found that it cannot be represented as one simple structure of the multiplexer as there are 4 to 1, 3 to 1, 2 to 1 multiplexers, and even special 4 to 1 multiplexer with 4-bit operation code, the easier way is to design specific multiplexer for every single space that needs multiplexer.

To implement the data path, we defined mainly 7 main parts of the circuit sub-system: MEM2IO, PC, REGFILE, ADDR, ALU, NZP, DataBus. The detailed block diagrams is shown in figure2:

1. **Module**: MUX_IO
   Inputs: [15:0] Data_to_CPU, [15:0] Data_Bus, MIO_EN
   Outputs: [15:0] MDR_Input
   Description: An 2 to 1 MUX, when MIO_EN is 0, MDR_Input will be Data_Bus, when MIO_EN is 1, MDR_Input will be Data_to_CPU.
   Purpose: Deciding data that is selected to MDR.

2. **Module**: MUX_PC
   Inputs: [15:0] Data_Bus, [15:0] Data_Calc, [15:0] PC_next, [1:0] PCMUX
   Outputs: [15:0] PC_Input
   Description: When PCMUX is 10, PC_Input = Data_Calc, which means select the Data from the ADDR part, If PCMUX is 11, then PC_Input = Data_Bus, otherwise, PC_Input = PC_next.
   Purpose: Deciding the value to PC register.

3. **Module**: MUX_Data_Bus
   Inputs: [15:0] PC, [15:0] MDR, [15:0] Data_Calc, [15:0] Data_ALU, [3:0] BUS_TICKET
   Outputs: [15:0] Data_Bus
   Description:If there is more than one 1 in the BUS_TICKET: Data_Bus = 16'h0000, If BUS_TICKET == 4'b0001: Data_Bus = Data_Calc, If BUS_TICKET == 4'b0010: Data_Bus = Data_ALU, If BUS_TICKET == 4'b0100: Data_Bus = MDR, If BUS_TICKET == 4'b1000: Data_Bus = PC
   Purpose: Simplify the four gates and avoid multiple values into the data bus.

4. **Module**: MUX_SR1
   Inputs: [2:0] IR_11_9, [2:0] IR_8_6, SR1MUX
   Outputs: [2:0] SR1
   Description: When SR1MUX is 1, SR1 = IR[11:9], if SR1MUX is 0, SR1 = IR[8:6]
   Purpose: Selecting the SR1 location.

5. **Module**: MUX_For_DR
   Inputs: [2:0] IR_11_9, DRMUX
   Outputs: [2:0] DR
   Description: When D1MUX is 1, DR = IR[11:9], if D1MUX is 0, DR = 111
   Purpose: Selecting the DR location.

6. **Module**: MUX_RegFile_Write
   Inputs: [2:0] DR, LD_REG,
   Outputs: LD_R0, LD_R1, LD_R2, LD_R3, LD_R4, LD_R5, LD_R6, LD_R7
   Description: The module MUX_RegFile_Write is a multiplexer that selects which register in a register file should be written to based on the value of DR (a 3-bit input). The input LD_REG determines whether the register should be written to (if LD_REG is high) or not (if LD_REG is low). The output signals LD_R0 through LD_R7 indicate which register(s) should be written to based on the value of DR and LD_REG. Each output signal corresponds to a specific register in the register file.
   Purpose: Enable write to REGFILE registers, controlled by MUX_For_DR.

7. **Module**: MUX_RegFile_Read
   Inputs: [2:0] SR, [15:0] R0_Out, [15:0] R1_Out, [15:0] R2_Out, [15:0] R3_Out, [15:0] R4_Out, [15:0] R5_Out, [15:0] R6_Out, [15:0] R7_Out
   Outputs: [15:0] data_out
   Description: When D1MUX is n, then data_out is Rn_Out.
   Purpose: Used in the process of reading a specific REGFILE register data into ALU.

8. **Module**: MUX_SEXT_10, MUX_SEXT_5, MUX_SEXT_8, MUX_SEXT_4
   Inputs: [n:0] IR_n_0
   Outputs: [15:0] IR_n_0_SEXT
   Description: Signed extension for the IR segments.
   Purpose: SEXT operation.

9. **Module**: MUX_For_ADDR1
   Inputs: [15:0] SR1OUT, [15:0] PC, ADDR1MUX
   Outputs: [15:0] ADDR1_to_Adder
   Description: When ADDR1MUX equals to following values, the output would be: 1'b0: ADDR1_to_Adder = SR1OUT; 1'b1: ADDR1_to_Adder = PC;
   Purpose: Select the data into ADDR1.

10. **Module**: MUX_For_ADDR2
    Inputs: [15:0] IR_10_0_SEXT, [15:0] IR_8_0_SEXT, [15:0] IR_5_0_SEXT, [1:0] ADDR2MUX
    Outputs: [15:0] ADDR2_to_Adder
    Description: When ADDR2MUX equals to following values, the output would be:

    2'b00: ADDR2_to_Adder = IR_10_0_SEXT;

    2'b01: ADDR2_to_Adder = IR_8_0_SEXT;

    2'b10: ADDR2_to_Adder = IR_5_0_SEXT;

    2'b11: ADDR2_to_Adder = 16'h0000;
    Purpose: Select the data into ADDR2.

11. **Module**: MUX_For_NZP
    Inputs: [15:0] Data_Bus
    Outputs: [2:0] NZP_In
    Description: The Output value NZP_In would be 010 when Data_Bus == 0, 100 when negative, and 001 when positive.
    Purpose: Judge the value's sign. And transport the signal to the NZP register.

12. **Module**: MUX_For_SR2
    Inputs: [15:0] IR_4_0_SEXT, [15:0] SR2OUT, SR2MUX
    Outputs: [15:0] ALU_In_B

Description: If SR2MUX = 0, ALU_In_B would be SR2OUT, otherwise, ALU_In_B would be IR_4_0_SEXT.
Purpose: Deciding the SR2 value to ALU.

13. **Module**: MUX_For_BEN
Inputs: [2:0] NZP_Out, [2:0] IR_11_9
Outputs: BEN_In
Description: If and only if $\Sigma_{i=0}^{2} NZP\_Out[i] and IR[9+i] = 0$, BEN_In would be 0, otherwise, BEN_In = 1.
Purpose: Specifying BEN value.

14. **Module**: MUX_For_ADDR2
Inputs: [15:0] IR_10_0_SEXT, [15:0] IR_8_0_SEXT, [15:0] IR_5_0_SEXT, [1:0] ADDR2MUX
Outputs: [15:0] ADDR2_to_Adder
Description: When ALUK equals to following values, the output would be:

2'b00: Data_ALU = Add_Out;

2'b01: Data_ALU = And_Out;

2'b11: Data_ALU = SR1OUT;

2'b10: Data_ALU = Not_Out;
Purpose: Deciding the calculation result from ALU results.

2.4.3 Other modules

We make a note of other modules here.

1. **Module**:HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: A hex driver that transform a 4-bit number into a hex 7-segment display.
Purpose: To display hex value on FPGA.

2. **Module**:register
Inputs: [N:0] D, Clk,Reset, Load
Outputs: [N:0] Data_Out,
Description: A synchronous register. When Load is high, Data of D is loaded into the register. All operations on rising edge of Clk. **Purpose**: This module is used to store data of SLC3.

3. **Module**: Synchronizer
Inputs: Clk, d
Outputs: q
Description: q will be d only at rising edge of clock.
Purpose: To synchronize user inputs and SRAM signal.

2.5 Description of operation of ISDU

The ISDU is simply part of the state diagram of LC-3. The inputs of ISDU include user input *Reset, Run, Continue* as well as *IR_5, IR_11, BEN, Opcode* to decide different behaviors or some operations. Based on these inputs, it outputs all necessary choices for the Load register signals, data bus gate signals, MUX choice signals, and memory enable signals. The state machine is a Moore state machine so output only relies on its current state. Each instruction is decomposed into several states of operations and by gradually passing through these states, the operation of instruction is executed.

A simple example of ISDU state control for ADD operation is

$$S_{18} \rightarrow S_{33\_1} \rightarrow S_{33\_2} \rightarrow S_{33\_3} \rightarrow S_{35} \rightarrow S_{32} \rightarrow S_1 A \rightarrow S_{18}(\text{next instruction starts})$$

The first five states are FETCH operations. Specifically, In state S_{18}, We need to pass PC to MAR. So we need to set *GatePC* and *LD_MAR* to 1 to pass PC to the data bus and MAR can load this value. To increment PC, we see *PCMUX* to choose PC + 1 to load into PC and *LD_PC* should also be set to 1.

States $S_{33\_1}, S_{33\_2}, S_{33\_3}$ are used to read data from SRAM. We can do this by setting *Mem_OE* to 0(active low) and set *LD_MDR* to one at the last state. Three states are required as we need to wait for approximately 10ns to let data go to the internal tristate buffer and one clock cycle to let the tristate buffer to pass the data. Since we and synchronizer to *Mem_OE*, *Mem_OE* will be delayed for one cycle to reach tristate buffer so we need to wait for one additional state.

In state S_{35} we simply pass MDR to IR by setting *GateMDR* and *LD_IR* to 1. In state S_{32} we need to state *LD_BEN* to one, as indicated by the state machine diagram. Also, based on *Opcode*, we will go to a different execute state. Since we are implementing ADD operation, we will go to state S_1.

In state S_1, we need to do the add operation and load results into the destination register, so we need to choose appropriate registers by setting *DRMUX* for DR, *SR1MUX* for SR1, *SR2MUX* to choosing between ADD or ADDi. *ALUK* for add operation. To load the value, we need to pass the result to the data bus and load registers by setting *GateALU* and *LD_REG* to 1. Also, *LD_CC* should also be set. Once add operation is over, we will go back to state S_{18} to start the next instruction.

Other instructions are similar except for different outputs depending on what needs to be done.

2.6 State Diagram of ISDU

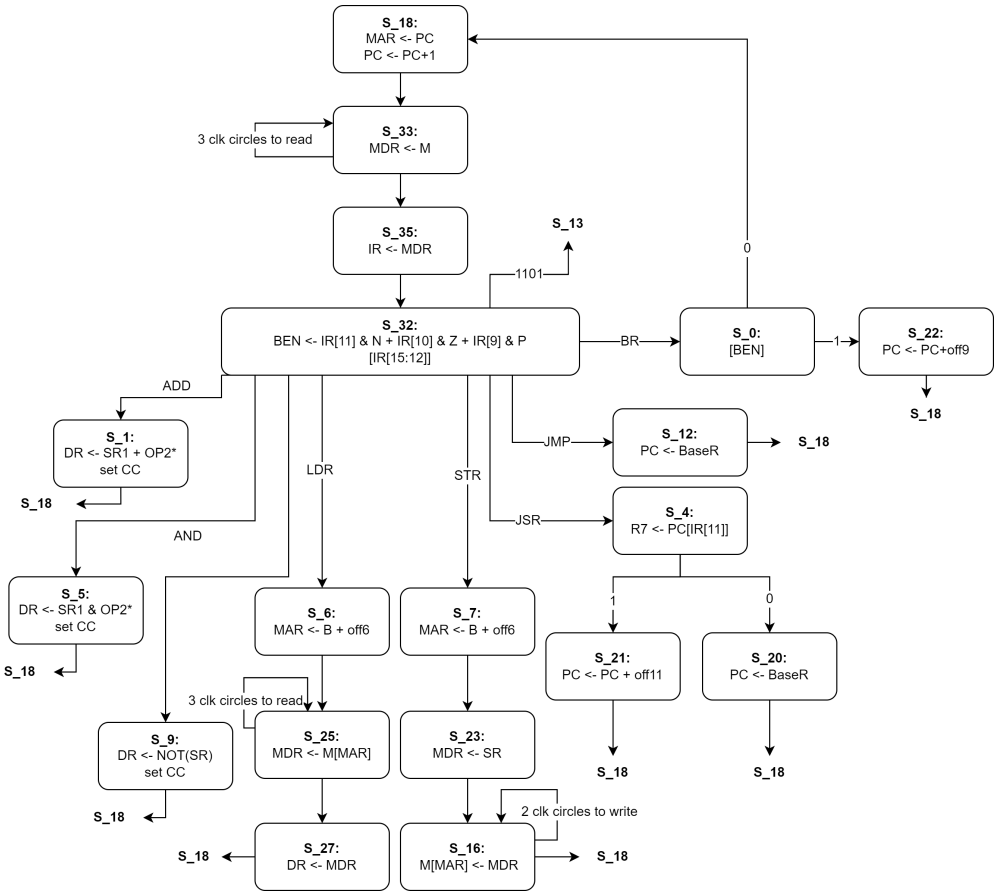

Figure 3: SLC-3 State Diagram

3 Simulations of SLC-3 Instructions

Simulation of Add operation is shown in Figure 4. It adds value of R1 by one. Namely, R1 ¡- R1 + 1.Instruction at S18 annotated. One should note how states change and how add operation is successfully executed by choosing right operands and mux.

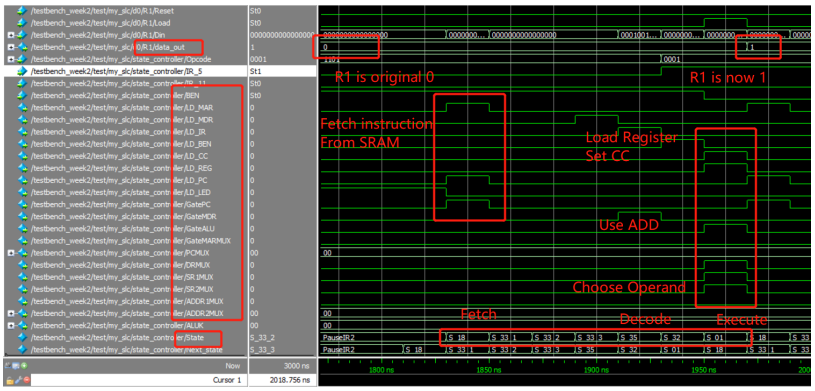

Figure 4: Add Simulation

Simulation of JMP operation is shown in Figure 5. Instruction starts at state S18 annotated. It executes instruction PC ¡- R1. R1 is set to 42 and after JMP, PC is set to 42. One can see how PC is changed by seeing how states change.

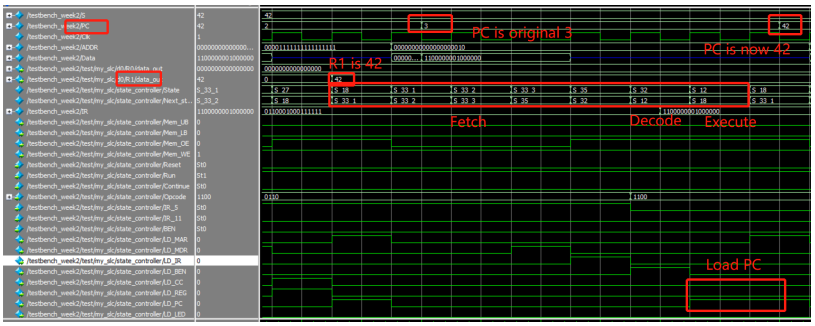

Figure 5: JMP Simulation

Simulation of LDR operation is shown in Figure 6. Instruction starts at state 18 annotated. It executes instruction R1 ¡- M[R0 + 0xFFFF]. R0 is set to zero so it means to load value from switches to R1. One can see how states change and switch is loaded into R1.

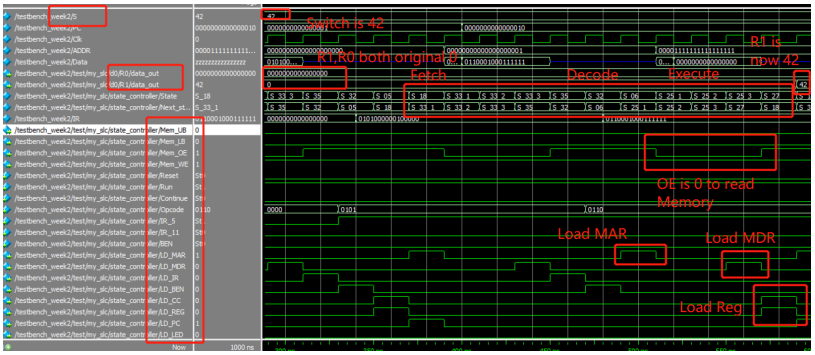

Figure 6: LDR simulation

4 Post-lab Questions

1. **Design statistics table**

LUT	612
DSP	0
Memory(BRAM)	0
Flip-Flop	288
Frequency(Mhz)	64.42
Static Power(mW)	98.66
Dynamic Power(mW)	9.79
Total Power(mW)	178.35

Table 1: Design Statistics Table

2. **Problems encountered and solutions**

   a) **Problem**: Some mistyped values for input, output, and interfaces that lead to an unexpected result.
   Solution: Either check the code carefully or use the simulation result to infer where the mistake is. The general approach is to use the simulation.

   b) **Problem**: User input fails after pressing continue several times.
   Solution: Add synchronizer for user input

   c) **Problem**: After adding a synchronizer for *Mem_OE, Mem_WE*, the simulations go well but FPGA fails for XOR, Sort, and Multiplication test.
   Solution: Instead of setting *Mem_OE* one state beforehand, use three states for memory. It seems that the former tricky strategy doesn't work. It's better to add one more state.

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

   MEM2IO is the interface between the User and Memory. It allows the user to load switches to the CPU and read data from the CPU by showing the hex value on the LED at address 0xFFFF.

4. **What is the difference between BR and JMP instructions?** BR sets the PC to the destination address when certain conditions are met. Examples include the time when the last calculated value is negative, zero, or positive. JMP sets the PC to the destination address directly without any requirements

5. **What is the purpose of the R signal in Patt and Patel? How do we compensate for the lack of signal in our design? What implications does this have for synchronization?**

   R signal is used to indicate when the data is ready to be read/written from SRAM. It indicates that the memory has been successfully fetched. In our design, through empirical study, it is found that data is guaranteed to be fetched with 2 clock cycles. More specifically, there are 10ns for data transmission and one clock cycle for the tri-state buffer. So we just need to use 2 clock cycles for memory read and write. If we add a synchronizer for the memory signal, the memory signal will arrive one clock cycle later. So there will be 3 cycles for reading and 2 cycles for writing.

5 Conclusion

5.1 Functionality of our design

Our design of SLC-3 works pretty well on FPGA. It passes all tests and gives the correct results. By adding synchronizers, the buttons are smooth and no matter how you press the buttons, FPGA always

gives the correct result. Overall, our design of SLC is successful.

5.2 Any possible improvements on the lab

Overall, the manual for this report is well-written and explains things clearly. The provided code template also saves us lots of work and helps us easily understand what we need to do and how we can achieve something. The provided testbench is also very good and helps us a lot in debugging the program. In summary, I think this lab is really good and helps us learn a lot about hardware design.