ECE 385

Fall 2019

Experiment #6

Simple Computer SLC-3.2 in SystemVerilog

Xinglong Sun Churan He
ABD T 8AM
Mihir Iyer

Introduction

a. Basic functionality of the SLC-3 processor

In this lab, we use SystemVerilog to design and build an SLC-3 processor on the DE2-115 FPGA board. The 16-bit SLC-3 processor uses a subset of the LC-3 ISA, and it is capable to perform 9 different instructions: ADD, AND, NOT, BR, JMP, JSR, LDR, STR and Pause. With the off-chip SRAM chip, our processor can read/write instructions and data from the memory, and this feature makes executing programs possible. To sum up, the 16-bit processor we built is a von Neumann universal computing machine based on the simplified hardware architecture that we learned from ECE 120.

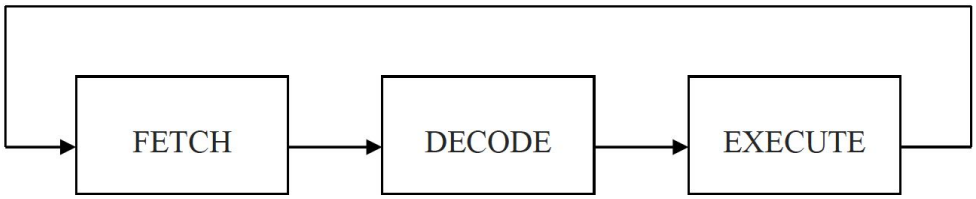
Written Description and Diagrams of SLC-3

a. Summary of User Operation

Operations user can perform include toggling the on-board switches to load in custom inputs, pressing Reset to bring the state machine back to the initial state, pressing Run to start the pre-loaded program, and pressing Continue to step through the program instruction by instruction. User is supposed to have the program already loaded into the memory before performing any on-board actions.

b. Functions can be performed by SLC3

Our SLC3-Processor will first fetch an instruction from the memory, decode it to determine the type of the instruction, execute the instruction, and then fetch again. A simple flowchart is provided as below:



Specific actions and descriptions of each step are:

Fetch:

Instruction to be executed next is fetched from the memory and is prepared for future actions. Unless instructed otherwise, the increment of Program Counter (PC) in FETCH is implicit but necessary. A RTL description is provided as follows:

MAR <- PC; MAR = memory address to read the instruction from

MDR <- M(MAR); MDR = Instruction read from memory
IR <- MDR; IR = Instruction to decode
PC <- (PC + 1)

Decode:

Instruction fetched from the memory is decoded in this step. Specifically, our processor determines what type of instruction this is by looking at its “opocode” (IR[15:12]). We achieve this decoding by including an explicit “decode” state in our FSM which will be explained in more details later.

Execute:

Our processor performs the operation based on the signals from the Instruction Sequencer/Decoder Unit and write the result to the destination register or memory.

ISA of SIC-3 we implement in this lab is acutally a subset of the ISA we saw in ECE120. A table describing each instruction is attached below:

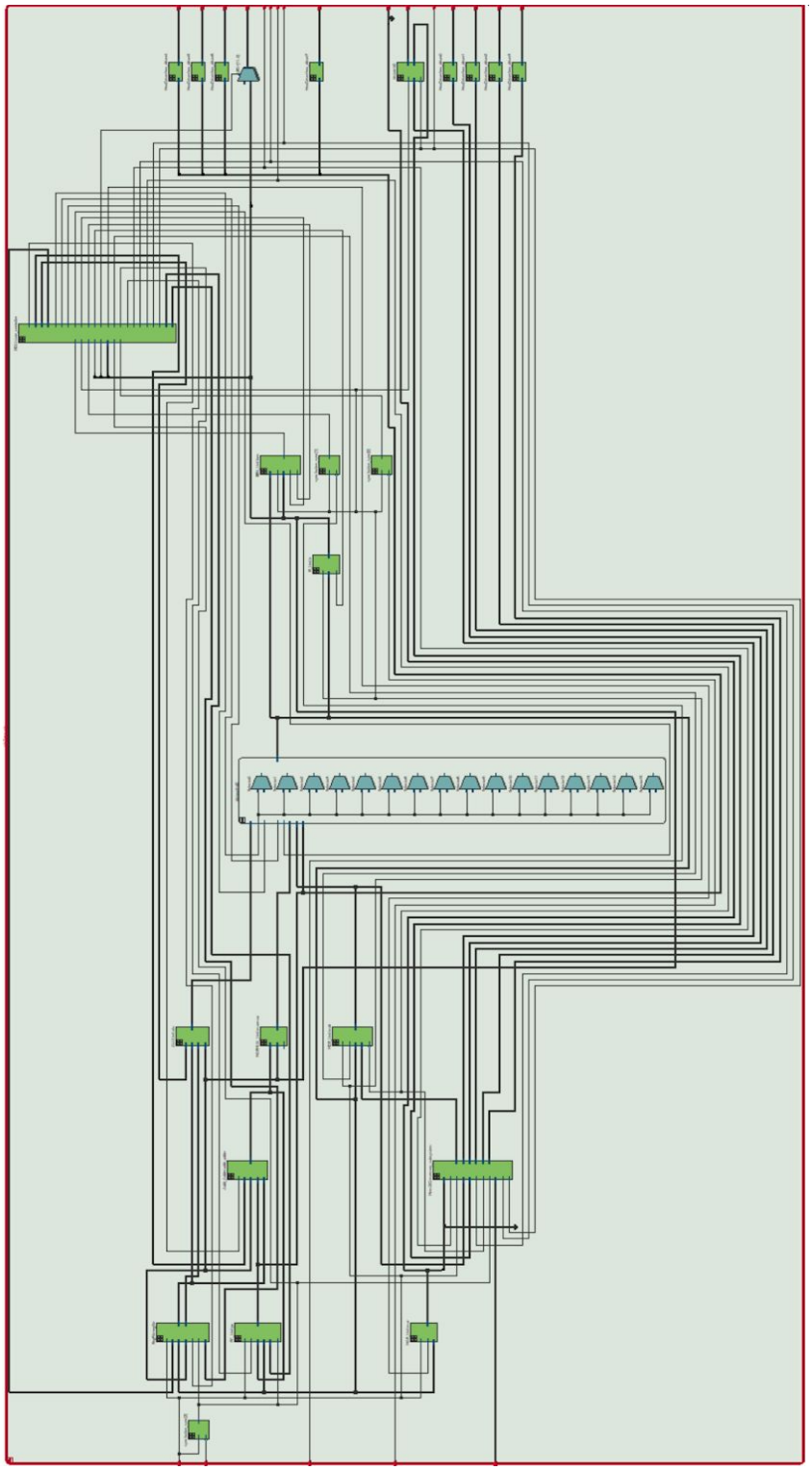
Instruction	Instruction(15 downto 0)																Operation
ADD	0001			DR			SR1			0	00		SR2				R(DR) ← R(SR1) + R(SR2)
ADDi	0001			DR			SR			1	imm5						R(DR) ← R(SR) + SEXT(imm5)
AND	0101			DR			SR1			0	00		SR2				R(DR) ← R(SR1) AND R(SR2)
ANDi	0101			DR			SR			1	imm5						R(DR) ← R(SR) AND SEXT(imm5)
NOT	1001			DR			SR			111111							R(DR) ← NOT R(SR)
BR	0000	n	z	p	PCoffset9												if ((nzp AND NZP) != 0) PC ← PC + SEXT(PCoffset9)
JMP	1100			000			BaseR			000000							PC ← R(BaseR)
JSR	0100	1	PCoffset11														R(7) ← PC; PC ← PC + SEXT(PCoffset11)
LDR	0110			DR			BaseR			offset6							R(DR) ← M[R(BaseR) + SEXT(offset6)]
STR	0111			SR			BaseR			offset6							M[R(BaseR) + SEXT(offset6)] ← R(SR)
PAUSE	1101	ledVect12															LEDs ← ledVect12; Wait on Continue

Actually, we purposely expand our circuit into the ECE120 version for our ISA to support more instructions which could include **JSRR, TRAP, LDI, LEA, STI**. We leave these instructions to be explored and added in the future. In the later sections, we will describe how we implement the expanded version circuit instead of the one required for this lab.

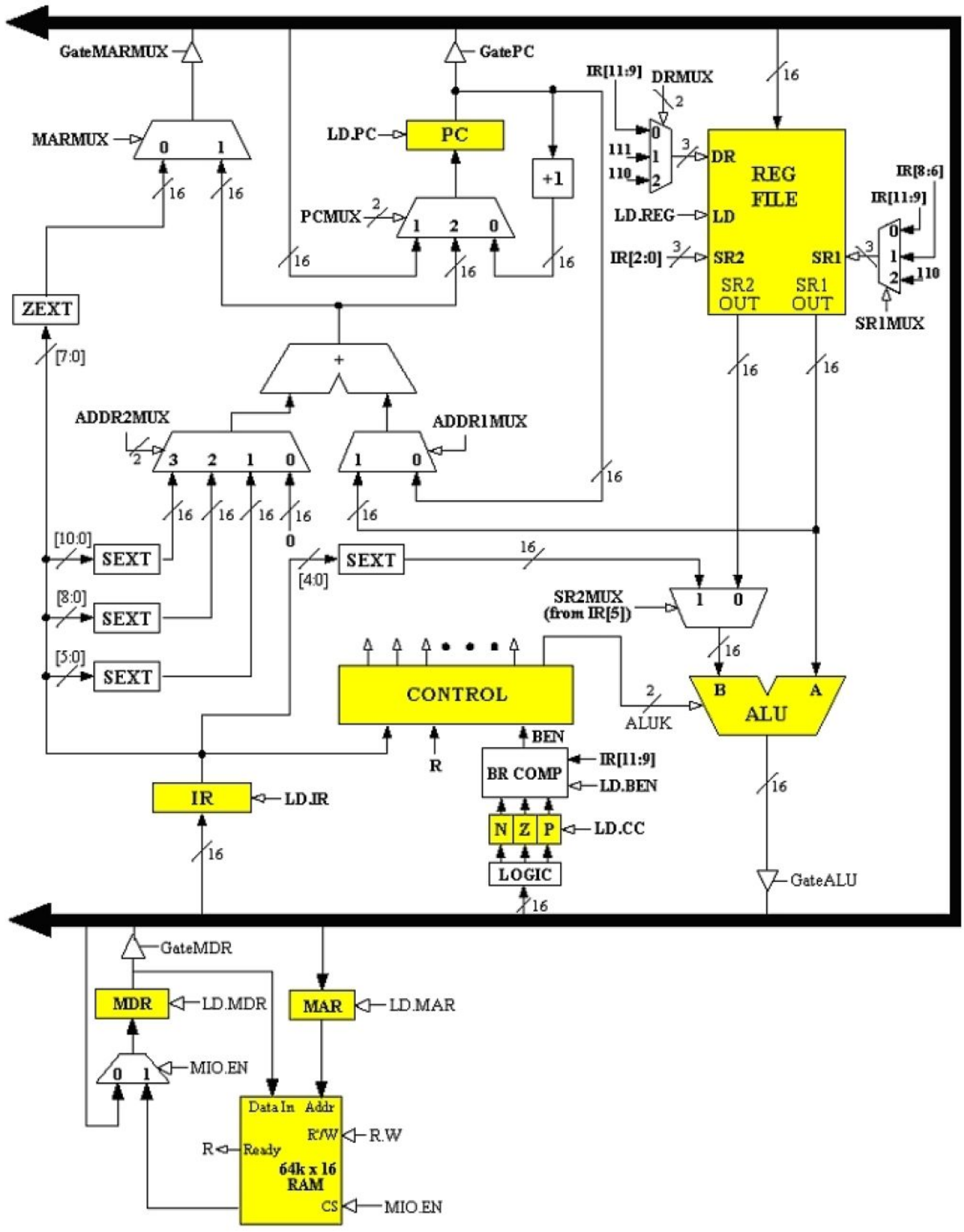
More detailed description for each instruction is provided as follows:

ADD Adds the contents of SR1 and SR2, and stores the result to DR. Sets the status register.
ADDi Add Immediate. Adds the contents of SR to the sign-extended value imm5, and stores the result to DR. Sets the status register.
AND ANDs the contents of SR1 with SR2, and stores the result to DR. Sets the status register.
ANDi And Immediate. ANDs the contents of SR with the sign-extended value imm5, and stores the result to DR. Sets the status register.
NOT Negates SR and stores the result to DR. Sets the status register.
BR Branch. If any of the condition codes match the condition stored in the status register, takes the branch; otherwise, continues execution. (An unconditional jump can be specified by setting **NZP** to 111.) Branch location is determined by adding the sign-extended PCoffset9 to the PC.
JMP Jump. Copies memory address from BaseR to PC.
JSR Jump to Subroutine. Stores current PC to R(7), adds sign-extended PCoffset11 to PC.
LDR Load using Register offset addressing. Loads DR with memory contents pointed to by (BaseR + SEXT(offset6)). Sets the status register.
STR Store using Register offset addressing. Stores the contents of SR at the memory location pointed to by (BaseR + SEXT(offset6)).
PAUSE Pauses execution until Continue is asserted by the user. Execution should only unpause if Continue is asserted during the current pause instruction; that is, when multiple pause instructions are encountered, only one should be “cleared” per press of Continue. While paused, ledVect12 is displayed on the board LEDs. See I/O Specification section for usage notes.

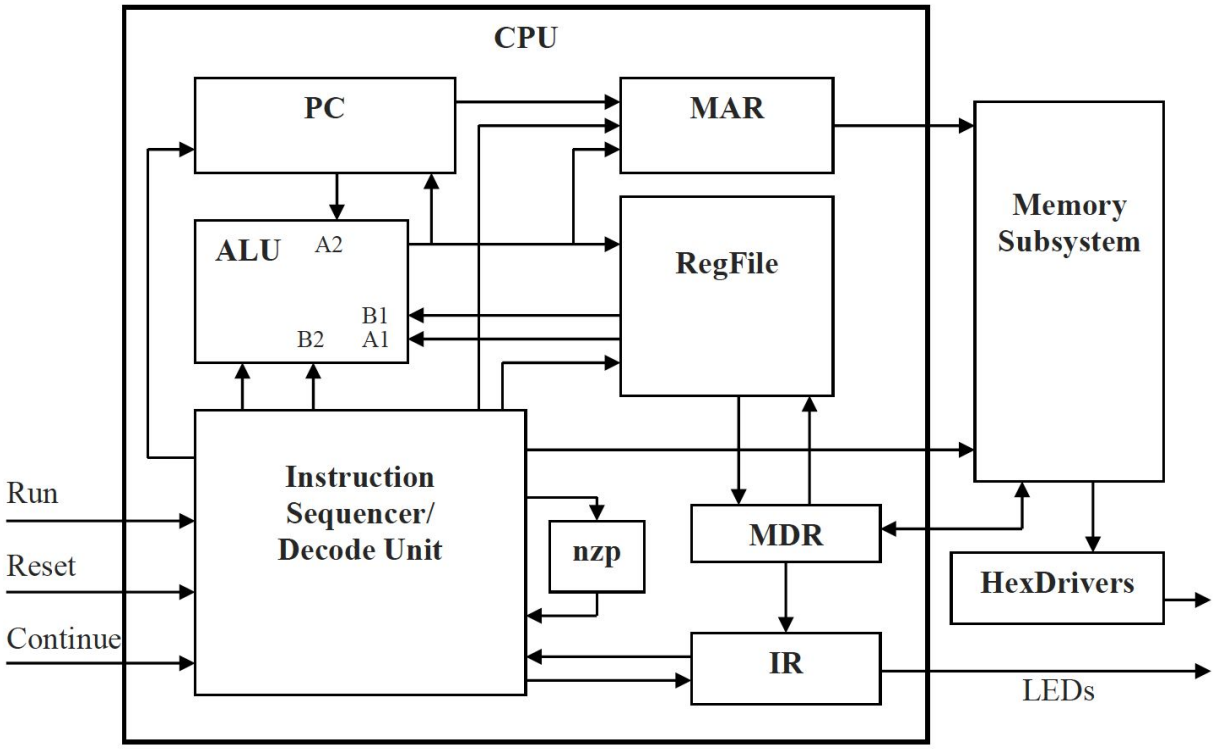
c. Top Level Block Diagram (slc3.sv) Alex



Block Diagram generated in Netlist Viewer



Block Diagram of the circuit we implemented



A simplified block diagram showing the connections between CPU and Mem-to-IO Block

d. Written Description of all .sv modules

Module: Lab6_toplevel.sv Alex

Inputs: [15:0] S,

Clk, Reset, Run, Continue,

Outputs:

[19:0] ADDR,

[11:0] LED,

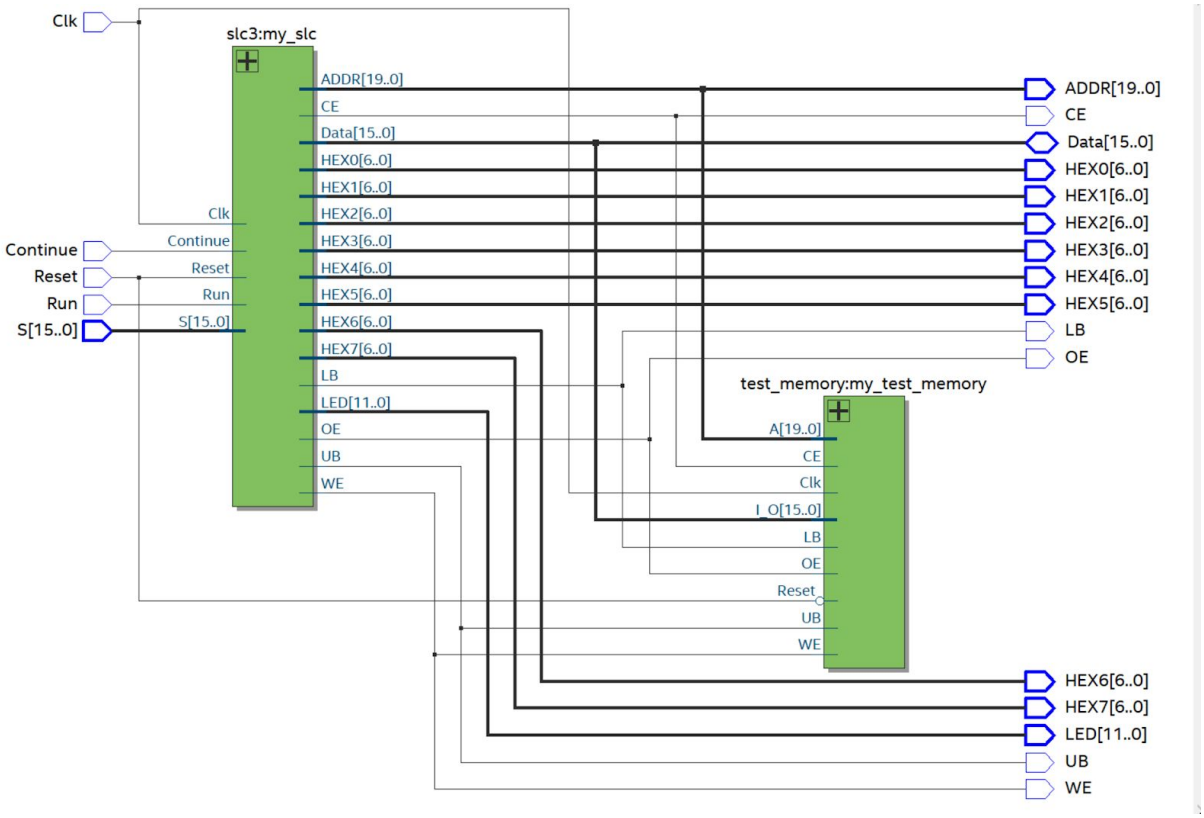
[6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,

CE, UB, LB, OE, WE

InOut:

[15:0] Data

Description: This is the top-level unit in the synthesis. Its most essential part is the slc3.sv unit described below.
Purpose: This unit wraps our CPU unit and a simulated memory unit together for testing and debugging in ModelSim simulation.



Module: Slc3.sv Alex

Inputs:

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

Outputs:

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

InOut:

[15:0] Data

Description: This is the top-level of our CPU circuit. It's 16-bit long central processing unit with 8 separate registers include in the RegisterFile. After hitting the Reset button which brings our circuit back to the initial halted state, user can run the preloaded program by hitting the Run button. Hitting Continue button allows user to step through different checkpoints for testing and displaying values in different states. The reading of switch values(S) is achieved by Memory-Mapped IO we implement. Basically, we associate the memory address xFFFF to switches/LEDs.

Purpose: It wraps up different components described below as an unity, interacts with the memory subsystem by generating memory-related signals, and display the results on LEDs by outputting corresponding hex values.

Module: Slc3_2.sv Alex

Inputs: _

Outputs: _

Description: This file encodes several opcodes and functions used in the instruction writing.

Purpose: This allows users to write program with assembly because it maps various assembly keywords into binary values which could be easily interpreted by our circuit.

Module: BEN_Unit.sv

Inputs:

[15:0] IR, Bus,

Clk, LD_BEN, LD_CC,

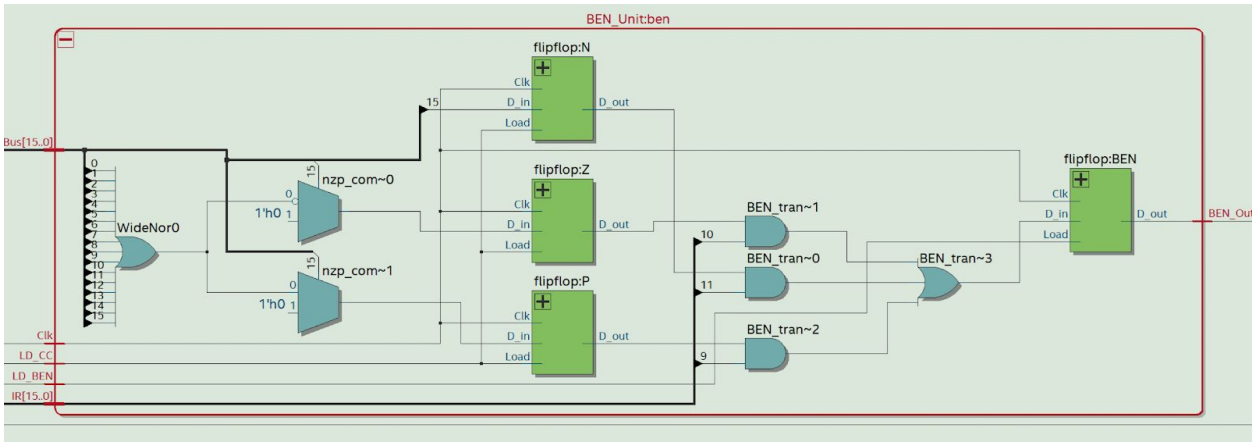
Outputs:

BEN_Out

Description: This module contains two parts, the circuit that determines NZP values and the circuit that determines the actual BEN value. The component stores the value of N, Z, P and BEN is flip-flop. N, Z, P flipflop's value is updated by the LD_CC signal (always in numerical evaluation, read from memory). If LD_CC signal is triggered, a combinational logic would

determine the value of N, Z, P based on the 16-bit binary on Bus. If LD_BEN signal is triggered (in state 32), the combinational logic would calculate IR[11] & N + IR[10] & Z + IR[9] & P to determine the BEN value. If BEN is 1, and the instruction is BR, state 22 would be entered, and a jump of the program counter can be achieved.

Purpose: The BEN_Unit is responsible for the BR instruction, which is primarily used in conditional operations. By determining the N, Z, P of a certain numerical operation, the program could execute the appropriate next instruction.



Module: Datapath.sv Alex

Inputs:

GatePC, GateMDR, GateALU, GateMARMUX,

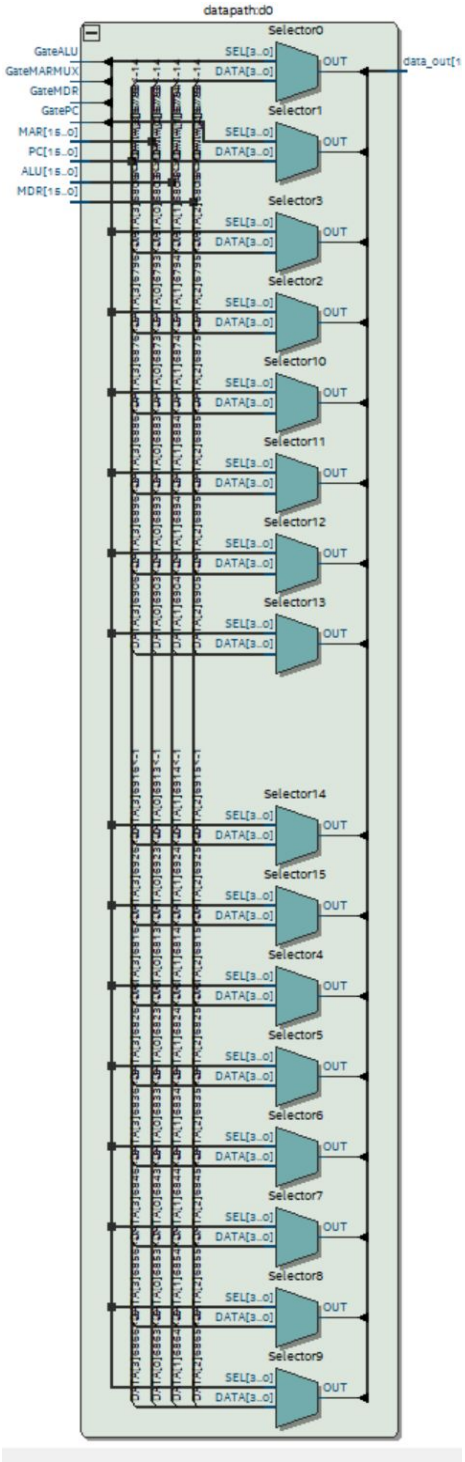
[15:0] MAR, PC, ALU, MDR,

Outputs:

[15:0] data_out

Description: This datapath module implements the data bus shown in bold black in the above block diagram. Since data on the datapath might come from four different sources, we need to ensure that our circuit can only have one source value to avert multiple driver issue. In common practice, this is accomplished by inserting a tri-state buffer between different outputs to datapath. However, in systemVerilog, there's no explicit tristate module. We thus use 4-to-1 Mux as a replacement. Concretely, when GatePC is high, PC value is selected as data_out. Same cases also apply to GateMDR, GateALU, and GateMARMUX.

Purpose: Due to the use of a datapath, our data value can be quickly transported to other parts of the circuit which are not spatially adjacent. This reduces the number of control signals generated by our ControlUnit.



Module: HexDriver.sv

Inputs:

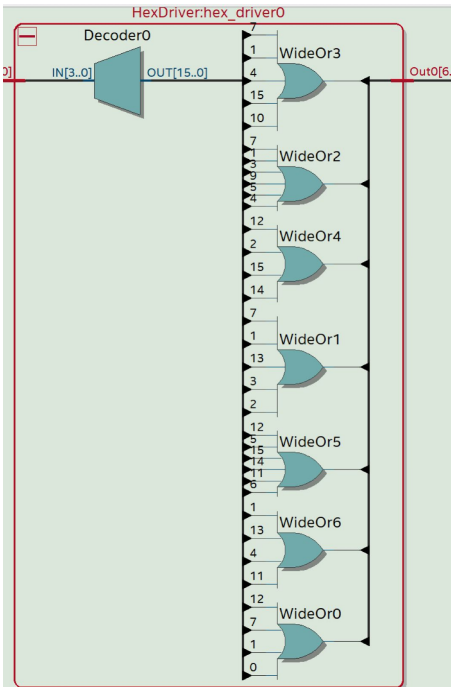
[3:0] In0

Outputs:

[6:0] Out0

Description: HexDriver is used to drive the Hex Display on the FPGA board. A HexDriver can represent a 4-bit binary as its hexadecimal form on the Hex Display.

Purpose: HexDriver can show the PC and IR in the week 1 experiment, and PC and Mem2IO output in the week 2 experiment. It makes debug and get computation result much easier.



Module: IR_Unit.sv Alex

Inputs:

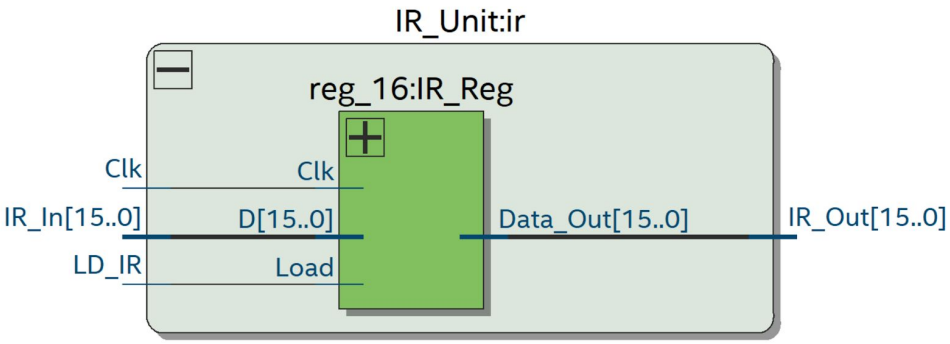
LD_IR, Clk,

[15:0] IR_In

Outputs:

[15:0] IR_Out

Description: IR_Unit is essentially a 16-bit register storing the current instruction to be executed. Instructions will be parallel-loaded into the register when LD_IR is high.
Purpose: IR_Unit stores the current instruction to be executed. It will provide the ISDU with the instruction to be executed and will also provide the datapath with any other necessary data.



Module: ISDU.sv

Inputs:

Clk, Reset, Run, Continue, IR_5, IR_11, BEN,

[3:0] Opcode,

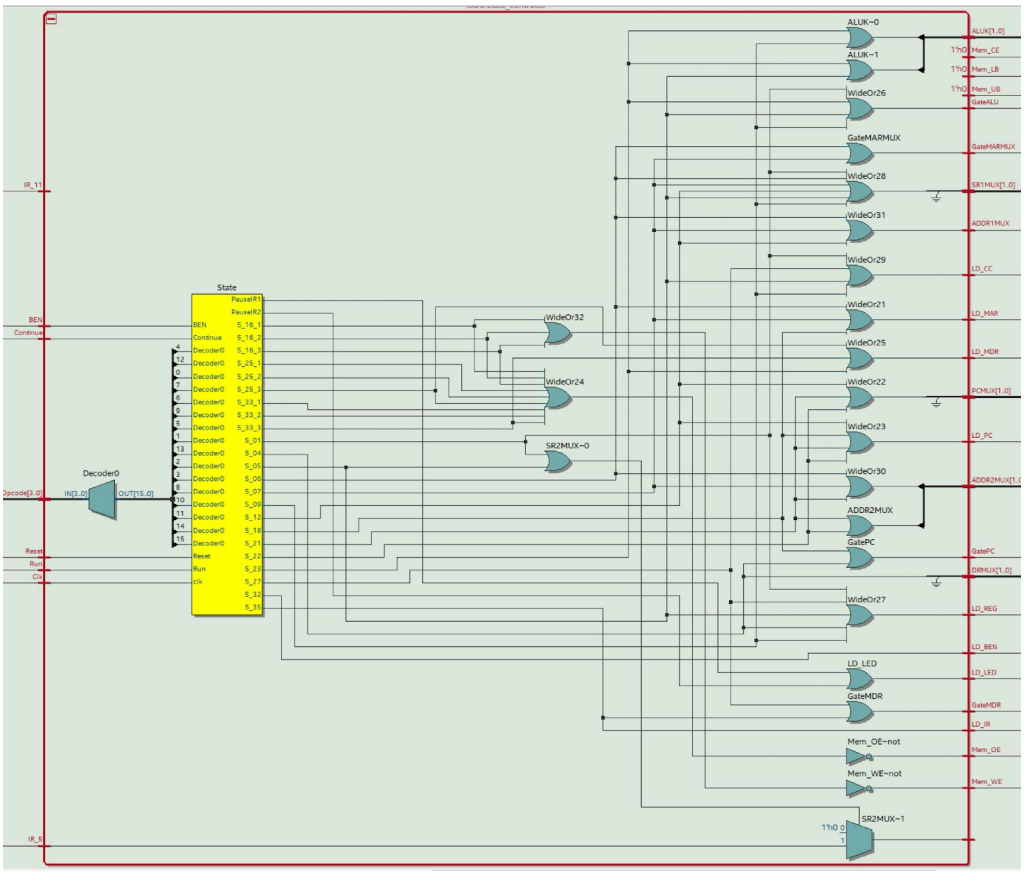
Outputs:

LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, Mem_CE, Mem_UB, Mem_LB, Mem_OE, Mem_WE,

[1:0] PCMUX, DRMUX, SR1MUX, ADDR2MUX, ALUK

Description: The ISDU serves as the main control logic for the SLC-3 Processor. It deals with all human inputs, and controls all LD signals and MUXes. The ISDU logic is mainly a Moore State Machine follow the given state transition diagram. The state at a specific time determines the output signals in order to perform the appropriate operation.

Purpose: The ISDU controls all sub-parts of the SLC-3 to make sure that it can work properly.



Module: MAR_Unit.sv

Inputs:

LD_MAR, Clk,

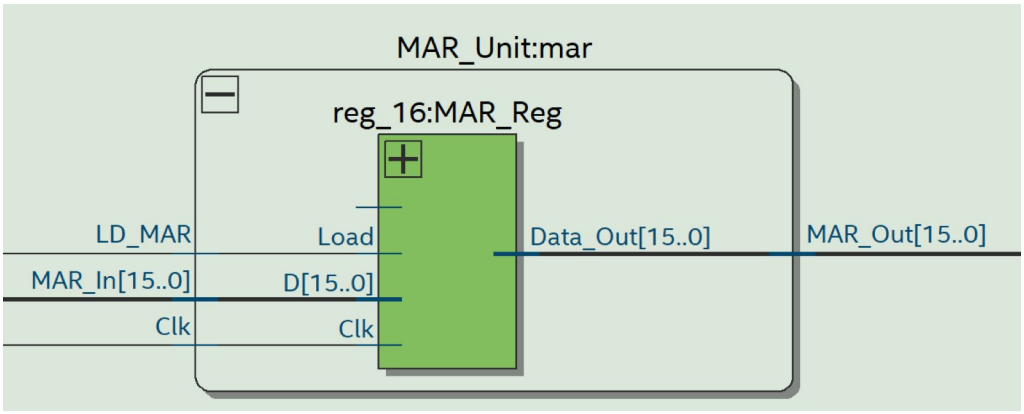
[15:0] MAR_In,

Outputs:

[15:0] MAR_Out

Description: MAR_Unit contains a 16-bit register. When LD_MAR is triggered, the memory address on the Bus would be stored in the MAR_Unit.

Purpose: MAR_Unit stores the memory address for CPU to read/write.



Module: MARMUX_Unit.sv Alex

Inputs:

[15:0] IR, Adder_Out,

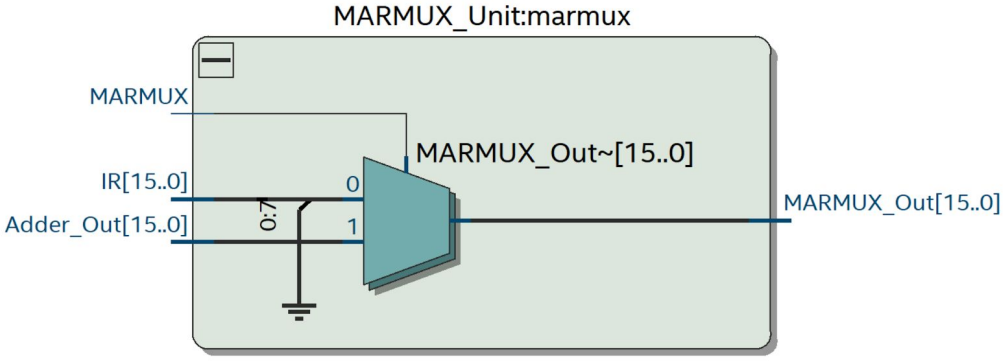
MARMUX

Outputs:

[15:0] MARMUX_Out

Description: MARMUX_Unit is essentially a 2-to-1 MUX that selects between Adder_Out(the output of our address adder) and the 16-bit ZEXT(Zero Extention) of IR[7:0].

Purpose: MARMUX_Unit is for JSRR. For functions required in this specific lab, we just keep MARMUX to be HIGH, which makes the MUX always select from the Adder_Out.



Module: MDR_Unit.sv

Inputs:

MIO_EN, LD_MDR, Clk,

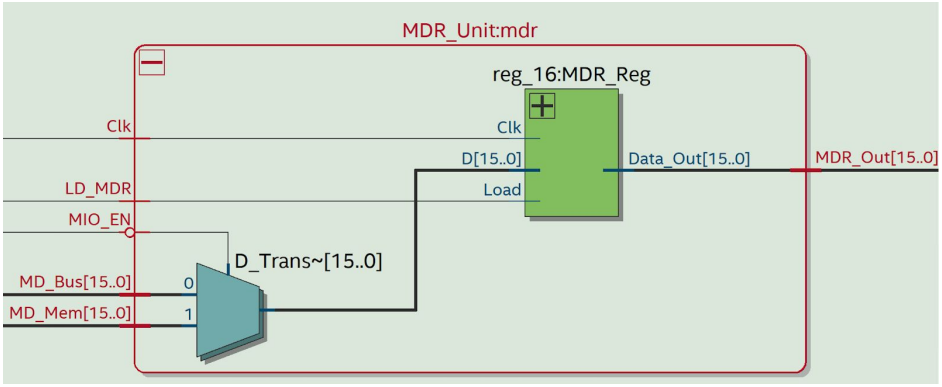
[15:0] MD_Bus, MD_Mem,

Outputs:

[15:0] MD_Out

Description: The MDR_Unit contains a 16-bit register and a Mux. The Mux controls the data source of the register is the Bus or the Memory output.

Purpose: The MDR stores the data to be written to memory and just read from memory like a buffer.



Module: MEM2IO.sv Alex

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 module is inserted between the processor and the external memory. When a memory access occurs at an I/O device address, the unit detects this, and sends a signal to the buffer to deactivate the memory, and instead use the I/O data for the response.

Physical I/O Device	Type	Memory Address	"Memory Contents"
DE2 Board Hex Display	Output	0xFFFF	Hex Display Data
DE2 Board Switches	Input	0xFFFF	Switches(15:0)

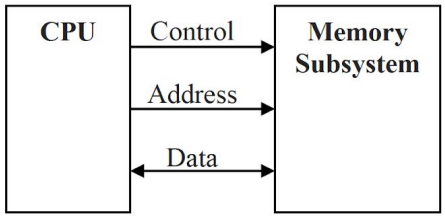


Figure 2: Conceptual Picture of Memory

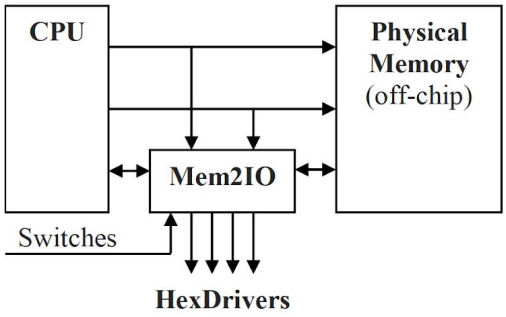
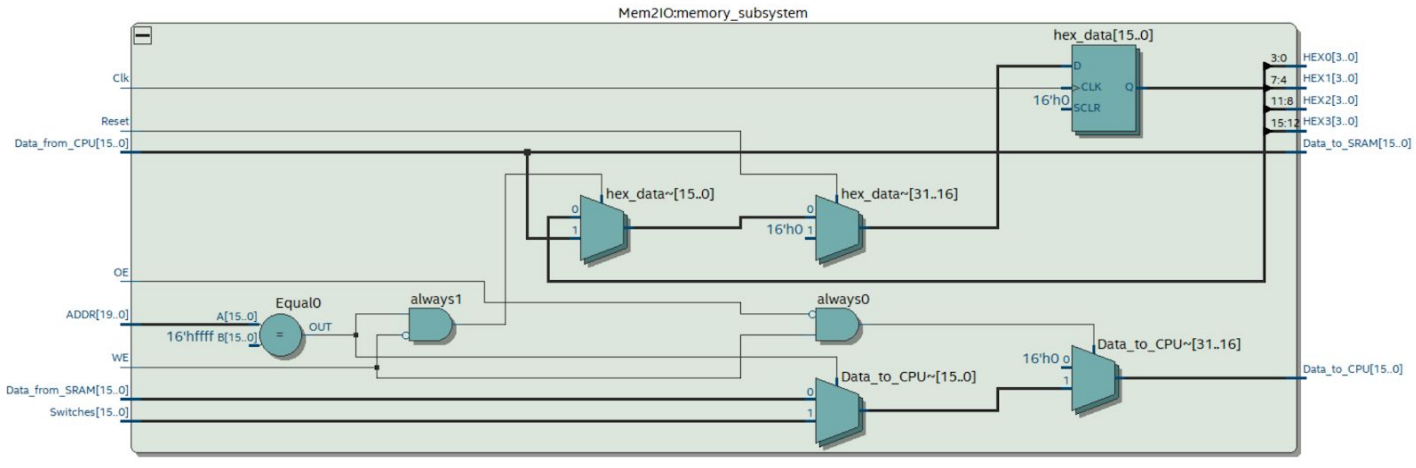


Figure 3: Memory with Mem2IO unit

Purpose: This Mem2IO block serves as a buffer between CPU and the Physical Memory. It determines when we should use the actual memory content and when we should read from device I/O to achieve our Memory-Mapped I/O Design.



Module: Memory_contents.sv

Input: _

Output: _

Description: Memory_contents contains the memory file of the testing program. This part would not be synthesised to hardware logic when uploading to the FPGA board, and it's only useful in the simulation process.

Purpose: Because SRAM is not a part of the FPGA chip, we need to simulate the functionality with test_memory, and the content of the test-memory is in Memory_contents during the on-computer simulation process.

Module: PC_Unit.sv Alex

Inputs:

Ld_PC, Clk, Reset,

[1:0] PC_Sel,

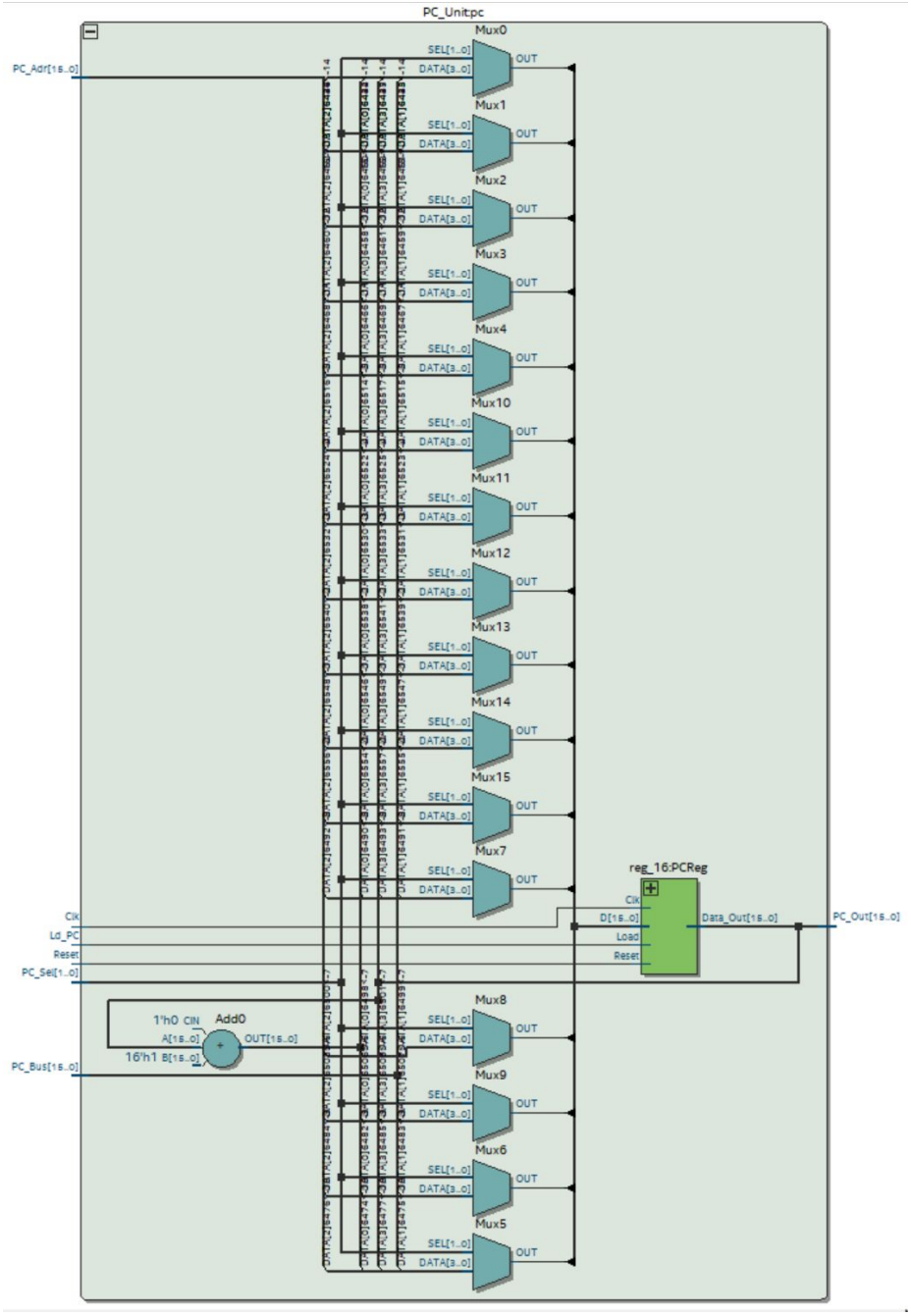
[15:0] PC_Bus, PC_Adr

Outputs:

[15:0] PC_Out

Description: PC_Unit includes a 16-bit register which keeps our current PC value. It's also equipped with the logic to determine what next PC value should be. Basically, when PC_Sel is 00, PC increases by 1. When PC_Sel is 01, PC takes in the value on the bus. When PC_Sel is 10, PC takes in the value of the address adder.

Purpose: Some of our instructions like JMP and BR change the PC address directly to start a program on a different memory address. This PC_Unit implements a simple select logic to give our ISA the option of changing or conditionally changing our PC value based on PC_Sel signal.



Module: Reg_16.sv

Inputs:

Clk, Reset, Load,

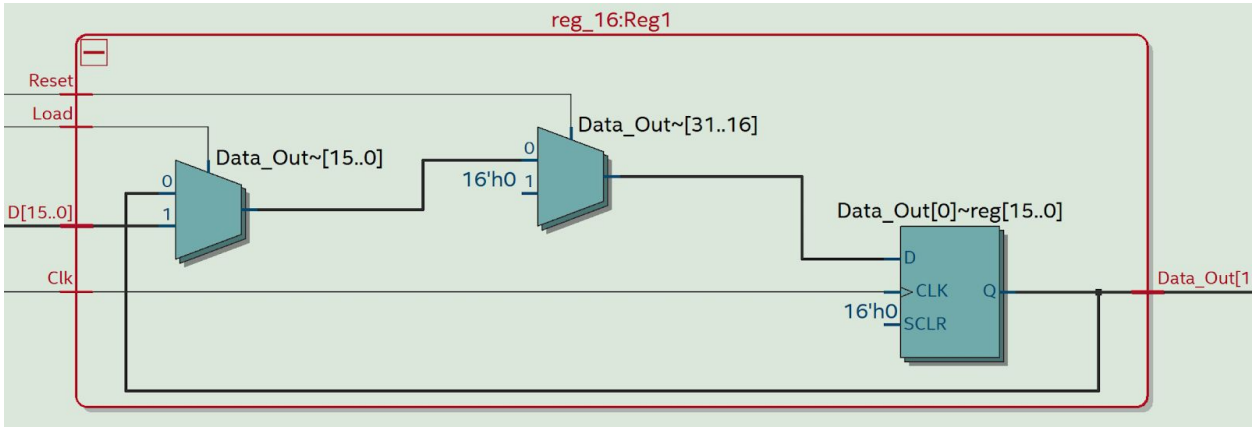
[15:0] D

Outputs:

[15:0] Data_Out

Description: reg_16 is a simple 16-bit register, and it's used in multiple units in the SLC-3 processor. It stores D value when Load is triggered at the Clk's rising edge.

Purpose: reg_16 stores useful value for the PC, IR, MAR, MDR and the register file.



Module: Reg_File.sv Alex

Inputs:

[15:0] IR, D_in,

[1:0] DRMUX, SR1MUX,

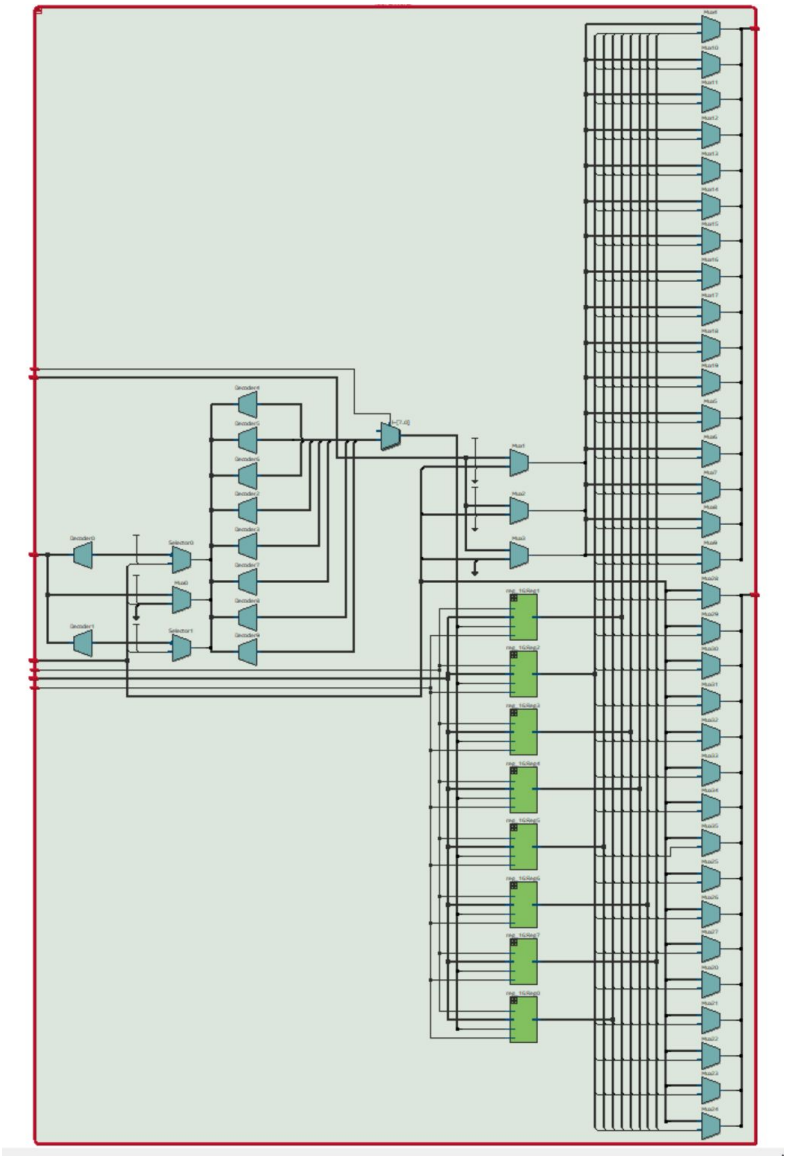
LD_REG, Clk, Reset

Outputs:

[15:0] SR1_Out, SR2_Out

Description: Reg_File Unit includes 8 independent 16-bit registers. DRMUX specifies the destination register that's going to be written by D_in. SR1MUX specifies the first source register to be read from. SR2MUX, although not an explicit input, is always determined by IR[2:0].
Purpose: Reg_File gives the users the freedom of using some immediate memory storage to perform complex logic operations. Several instructions, such as ADD, ADDi, AND, ANDi, store both the operands and result in RegFile to access them immediately.

Generally, since Reg_File is much closer to our CPU than the chip memory, manipulating data from Reg_File is more time-efficient and convenient.



Module: ALUUnit.sv

Inputs:

[15:0] A, B0, IR,

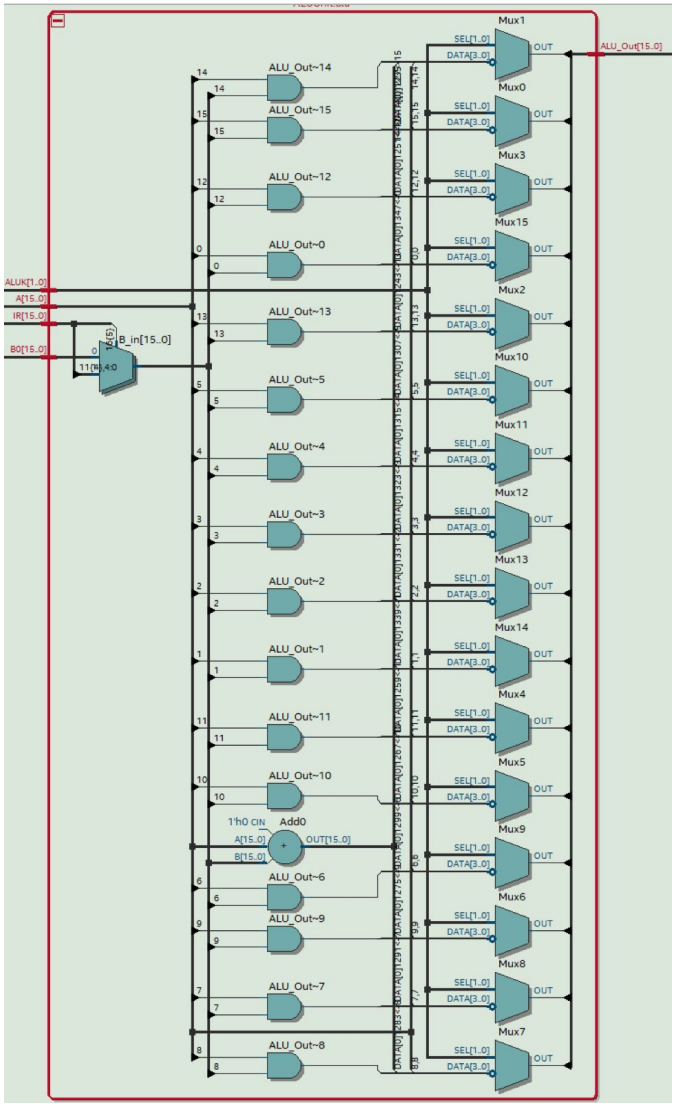
[1:0] ALUK,

Outputs:

[15:0] ALU_Out

Description: ALUUnit is responsible for numerical operations. SR2 is determined by IR[5], a MUX could choose the SR2 from register file or the value from the program instruction, which differentiate ADD/ADDi and AND/ANDi. The ALUUnit is able to perform 4 operations, ADD, AND, NOT and PASS.

Purpose: The ALUUnit is required for all numerical operations related to general purpose registers. WIth the ALUUnit, we are able to compute necessary data when the program runs.



Module: Addr_Adder.sv Alex

Inputs:

[15:0] IR, SR1_In, PC_In,

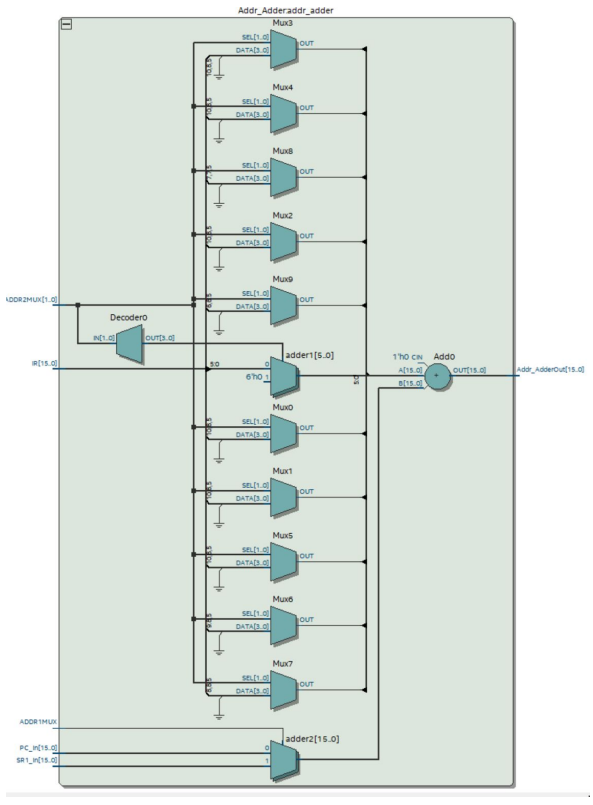
[1:0] ADDR2MUX,

ADDR1MUX

Outputs:

[15:0] Addr_AdderOut

Description: This unit includes a 4-to-1 MUX, 2-to-1 MUX, and a full adder to add values coming from two sources to generate a valid address. SR1_In represents the value from SR1 OUT of our Register File. PC_In represents the current PC value. One of the operands of the AdderUnit is 0 or SEXT version of the IR based on the ADDR2MUX select signals. The other operand of the AdderUnit is either SR1_In or PC_In based on ADDR1MUX.
Purpose: Multiple instructions in our ISA like LDR and STR require adding contents from a register unit and an offset as the address. This allows user to access memory



content far away from the current program and break the code into clear and logical sections.

Module: Synchronizers.sv

Input:

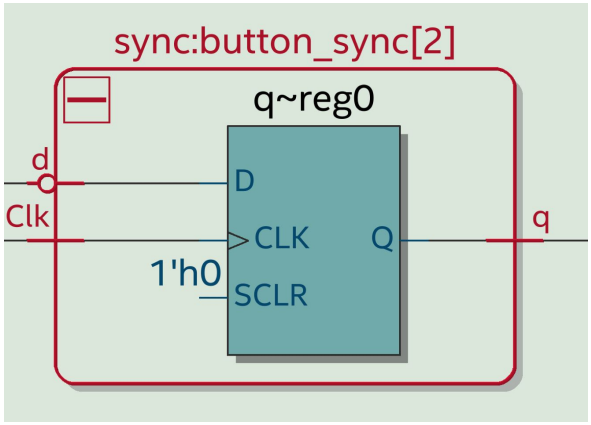
Clk, d,

Output:

q

Description: Synchronizer is basically a flip-flop. It can make asynchronous human input signal be synchronized.

Purpose: Reduce the meta-stable state caused by pressing buttons.



Module: test_memory.sv

Inputs:

Clk, Reset, CE, UB, LB, OE, WE,

[19:0] A

Inout:

[15:0] I_O

Description: The test_memory would not be uploaded to the actual FPGA hardware. It's a simulated SRAM component used for simulation, and it has the exactly same function as the real SRAM. The content is stored in the memory_contents file.

Purpose: The SRAM isn't included in the FPGA chip. To simulate its functionality, we need this virtual SRAM to store the memory data.

Module: tristate.sv

Inputs:

Clk, tristate_output_enable,

[N-1:0] Data_write,

Outputs:

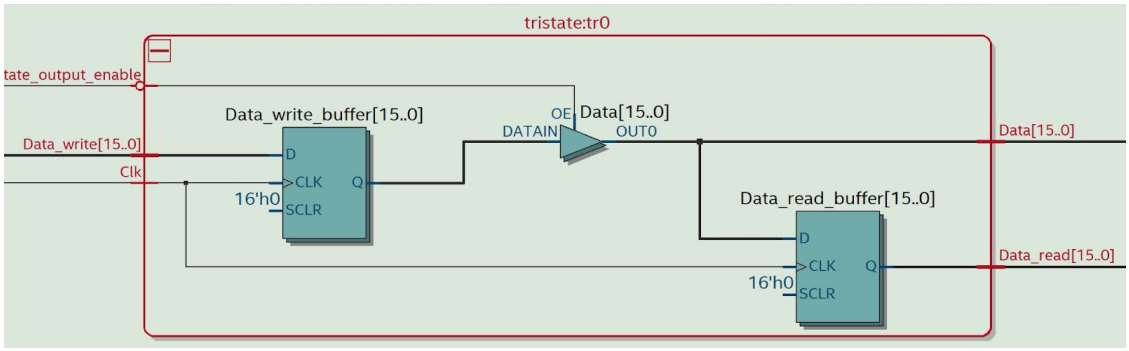
[N-1:0] Data_read,

Inout:

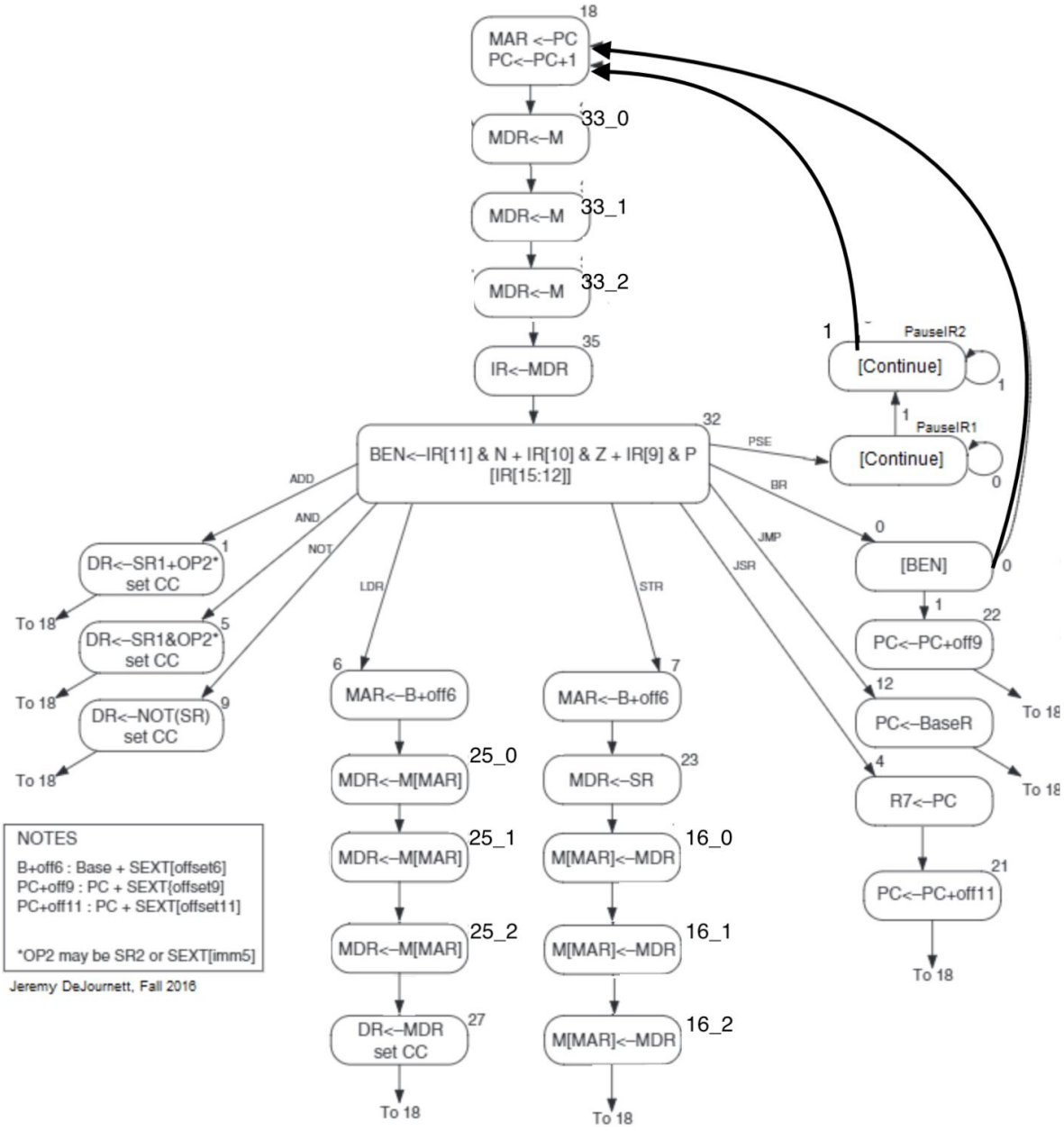
[N-1:0] Data

Description: tristate is the connection buffer from Mem2IO and the SRAM. SRAM's Data pins are inout ports, and Mem2IO has an output Data_to_SRAM and an input Data_from_SRAM. The tristate can connect the Data_to_SRAM with Data or the Data_from_SRAM with Data depending on the tristate_output_enable signal.

Purpose: It splits an inout port to an input port and an output port, which simplifies the design of Mem2IO unit.



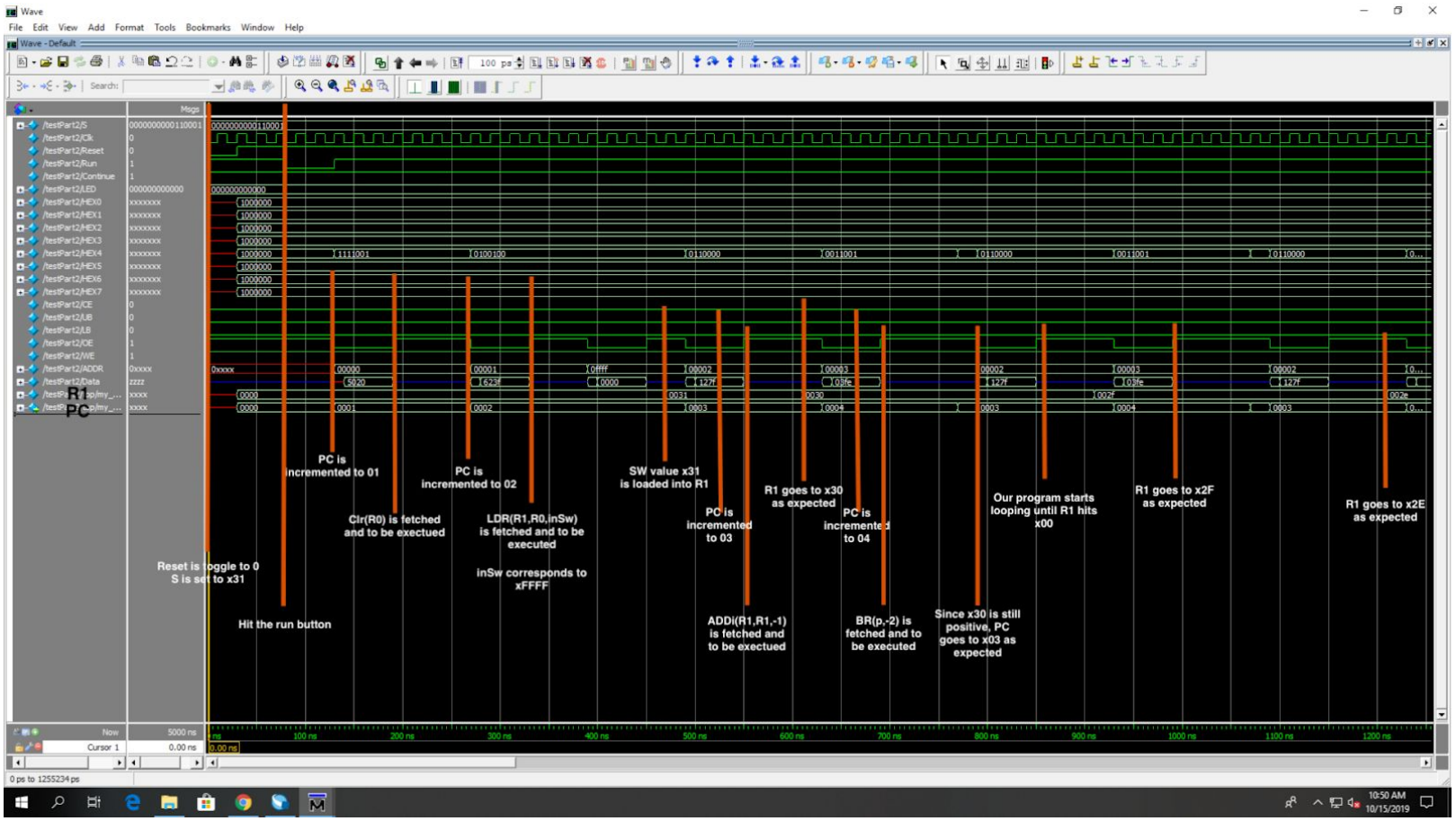
e. FSM Diagram & Description of ISDU



Simulations of SLC-3 Instructions

a. Annotated Waveforms

In our test program, we load the switch value into R1 and continuously subtract 1 from it until R1 hits 0.



Post-Lab Questions

a. Design Resources and Statistics table

LUT	561
DSP	0
Memory (BRAM)	0
Flip-Flop	274
Frequency	68.0MHz
Static Power	98.65mW
Dynamic Power	7.26mW
Total Power	177.36mW

2.) Several questions to notice

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

As elaborated in the above sections, MEM2IO is used here to handle our Memory-Mapped I/O. When xFFFF is input as the address, memory is disabled and our device I/O is used as the response.

- What is the difference between BR and JMP instructions?

Both operations involve moving the PC value directly. However, JMP feeds the content of a register into PC directly. BR, on the other hand, adds the current PC with an offset only if the nzp condition matches.

- 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?

R indicates that our memory content is ready to use. Since memory access takes longer time than usual operation, R signal is necessary in design to avoid the case where we didn't leave enough time to read from or write to memory. In this lab, since there's no R signal coming from the memory, we just leave multiple states for memory-related operations. Specifically, we break each memory-related state into three states to secure the data. This method is also good for synchronization since the operation time is strictly regulated by the clock cycles.

Conclusion

a. Functionality of the design.

Our circuit is able to pass all the assembly test cases provided including the bubble sort. As mentioned above, the functionality or the ISA of our circuit could be expanded into the ECE120 version which supports more complex operations. We leave these options to be explored in the future.

b. Suggestions for future semester

To much files provided for students. I think students should explore by their own and write all of the required files to understand the concept in depth.