ECE385

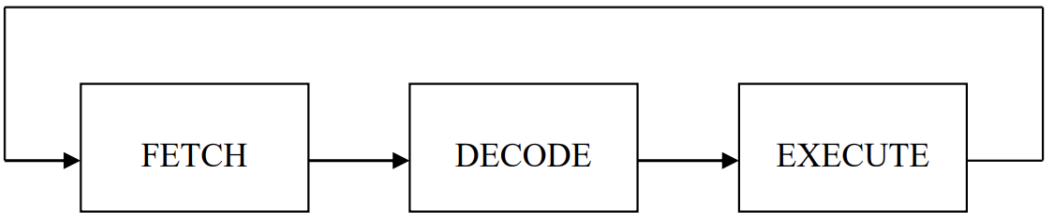
Lab 5

Jiaming Xu

Rui Gong

Introduction

For this lab, there are three main components to design the processor, which are FETCH, DECODE, and EXECUTE. The computer will first fetch an instruction from the memory, decode it to determine the type of the instruction, execute the instruction, and then fetch again.



In week 1, we did the fetch operation. According to the state diagram, we first load the value of PC to MAR, and increment the PC. Then, we load the memory of MAR to MDR, and wait for several clock cycles to make sure it gets to the next state. Then, we load the value of MDR to IR. This is the fetch operation. We transfer the memory of PC into instruction, and store it in the instruction register. In week 2, we first did the decode operation. Based on the opcode (IR[15:12]) and the conditional code (IR[11:9]), we can determine the type of instruction and what operation should be done in the next state. According to the state diagram, we can build the state machine of each operation separately. Based on the SLC3 datapath, we can do the operation step by step. The SLC3 can do 9 distinct operations, such as ADD, AND, NOT, and so on.

Written Description and Diagrams of SLC-3

a. Summary of Operation

The program consists of three main components, fetch, decode, and execute. The computer will fetch an instruction from the PC memory, and decode it to decide which operation should be done, and then, the computer executes the operation. This program can do 9 different

operations: ADD (ADD and ADDi), AND(AND and ANDi), NOT, BR, JMP, JSR, LDR, STR, and PAUSE.

b. Describe in words how the SLC-3 performs its functions. You should describe the Fetch-Decode-Execute cycle as well as the various instructions the processor can perform.

The SLC-3 has 3 main parts, fetch, decode, and execute.

For fetch operation:

We first load the value of PC into MAR and increment PC. Then, we load the memory of MAR into MDR and wait for several clock cycles to make sure it can reach the next state. Finally, we set the IR equals MDR. By doing these operations, we fetch an instruction from the memory.

For decode operation:

By using the opcode (IR[15:12]), we can determine which operation to do, meaning which is the next state in the state machine. If it is BR, we will do IR[11]&N + IR[10]&Z + IR[9]&p -> BEN.

For execute:

In execute, there are different operations depending on the opcode.

ADD:

For ADD operation, if IR[5] is 0, we add the memory of SR1(IR[8:6]) with the memory of SR2(IR[2:0]). If IR[5] is 1, we add the memory of SR1 with sign extend of an immediate number(IR[4:0]). The operation is done by using ALU and the answer is stored in DR(IR[11:9]). After the operation, we set the condition code NZP and go back to state 18 which is loading the value of PC to MAR.

AND:

For ADD operation, if IR[5] is 0, we and the memory of SR1(IR[8:6]) with the memory of SR2(IR[2:0]). If IR[5] is 1, we and the memory of SR1 with sign extend of an immediate number(IR[4:0]). The operation is done by using ALU and the answer is stored in DR(IR[11:9]). After the operation, we set the condition code NZP and go back to state 18 which is loading the value of PC to MAR.

NOT:

For NOT operation, we inverse the memory of SR(IR[8:6]) and store the value to DR(IR[11:9]). The operation is done by using ALU.After the operation, we set the condition code NZP and go back to state 18 which is loading the value of PC to MAR.

LDR:

For LDR, we first load MAR with the sum of the memory of BaseR(IR[8:6]) and sign extend of the offset(IR(5:0]). Then, we set MDR with the memory of MAR and wait for several clock cycles to make sure it reaches the next state. After that, we store the value of MDR to DR(IR[11:9]) and set the condition code. Finally, the program goes back to state 18.

STR:

For STR, we first load MAR with the sum of the memory of BaseR(IR[8:6]) and sign extend of the offset(IR(5:0]). Then, we set MDR equals to memory of SR(IR[11:9]). After that, we load memory of MAR with MDR and go back to state 18.

JSR:

We first store the value of PC into R7, and then load PC with PC+SEXT[offset11](IR[10:0]). After that, the program goes back to state 18.

JMP:

For JMP, we load PC with the memory of BaseR(IR[8:6]). The program goes to the address of PC. After that, the program goes back to state 18.

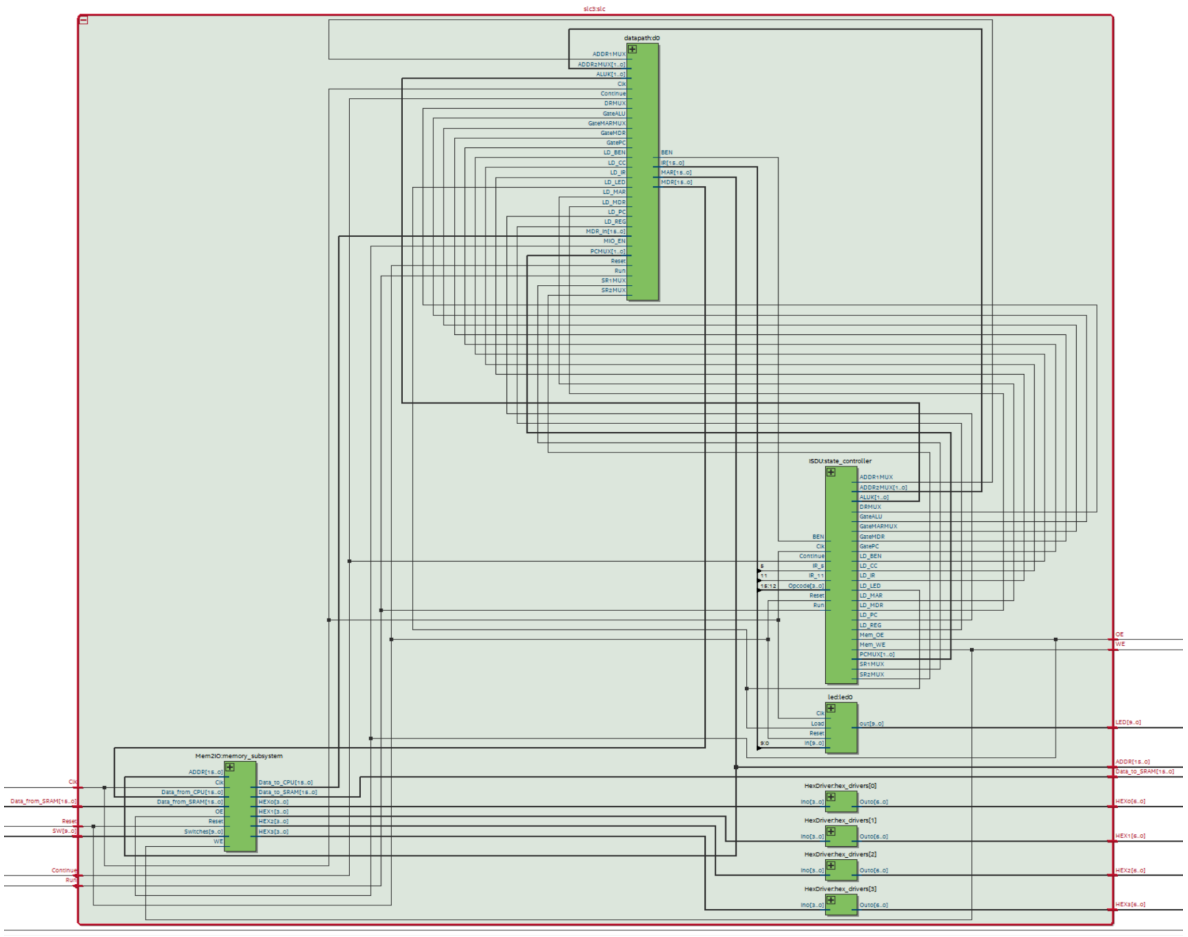
BR:

For BR, we need to make sure it is BEN. We can check this in decode. If IR[11] & N + IR[10] & Z + IR[9] & P != 0, it is BEN. We load PC with PC + SEXT[offset9](IR[8:0]). After that, the program goes back to state 18. If it is not BEN, the program goes back to state 18 directly.

PSE:

If it is PSE operation, the program will stay on the current operation, until we change the continue switch. In the first pause statement, if we press the continue button the program will continue, otherwise it will stay in its current state. For the second state, if we release the button, it will continue and go back to state 18, otherwise, it will still be in its current state.

c./d. Block Diagram of slc3.sv



Block diagram of slc3.sv

e. Written Description of all .sv modules

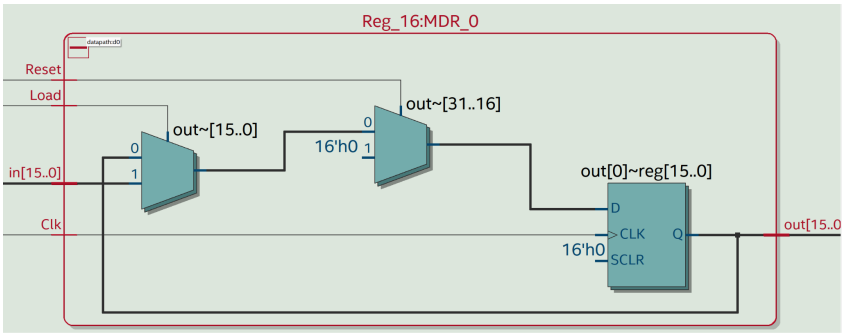
Module: Reg_16

Input: Clk, Reset, Load, [15:0] in

Output: [15:0] out

Description: This module is a 16 bits register module. At the rising edge of clock, this module will either reset the register value based on Reset or load in into the register based on Load.

Purpose: This module reset or load memory to register.



Block Diagram for Reg_16

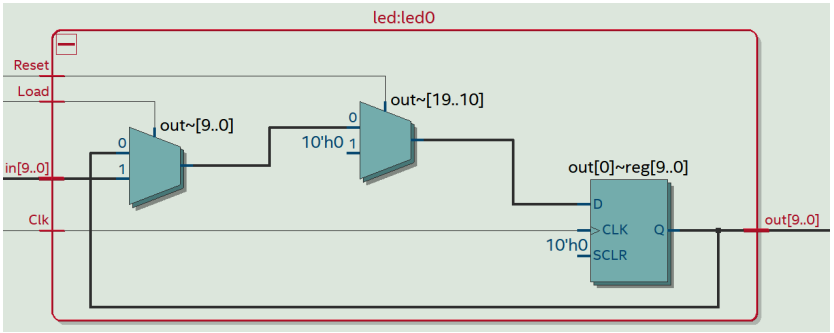
Module: led

Input: Clk, Reset, Load, [9:0] in

Output: [9:0] out

Description: This module is a LED module. At the rising edge of clock, this module will either reset the led value based on Reset or load in into the led based on Load.

Purpose: The module reset or load value to led.



Block Diagram for led

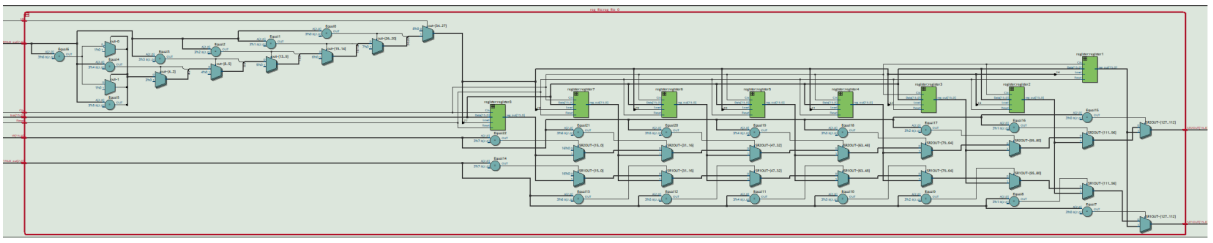
Module: reg_file

Input: Clk, Reset, LD, [15:0] bus, [15:0] IR, [2:0] DRMUX_out, [2:0]SR1MUX_out

Output: [15:0] SR1OUT, [15:0] SR2OUT

Description: In this module, at the rising edge of clock, when LD is 1, we store the value from bus to register based on DR which comes from DRMUX_out. SR1 and SR2 load the memory from the register that we just stored. To decide which register we should use, we use the output of SR1MUX. If it is 000, we load R0. If it is 001, we load R1, and so on. After loading the memory of the register to SR1 and SR2 we can get SR1OUT and SR2OUT.

Purpose: Store values to register file and load the value from register file to SR1 and SR2.



Block Diagram for REGFILE

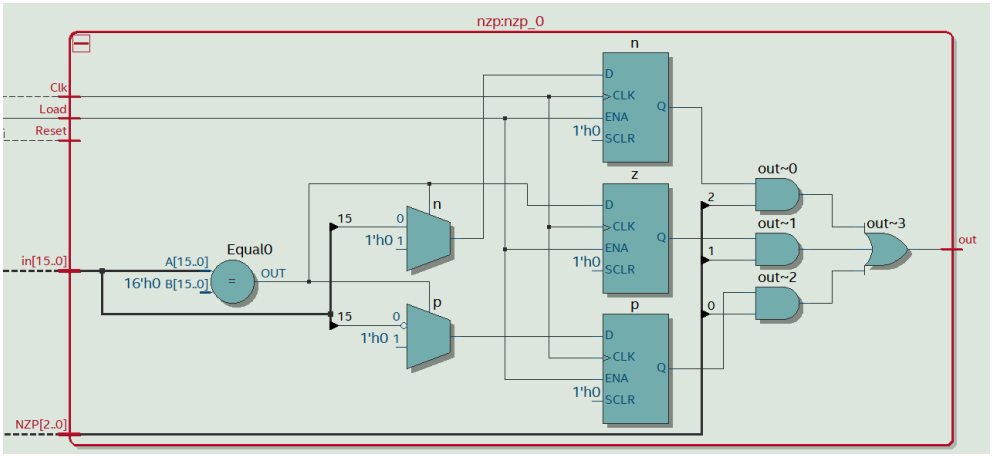
Module: nzp

Input: Clk, Reset, Load, [2:0] NZP, [15:0] in

Output: out

Description: The module is to set up the condition code. At the rising edge of Clk, when load, if in is all 0, we set z = 1. If in[15] = 0, we set p = 1. Else, we set n = 1. We assign the output out = (N&n) | (Z&z) | (P&p), where N is NZP[2], Z is NZP[1], and P is NZP[0].

Purpose: This module loads the condition code and compares it with NZP in BR.



Block Diagram for nzp

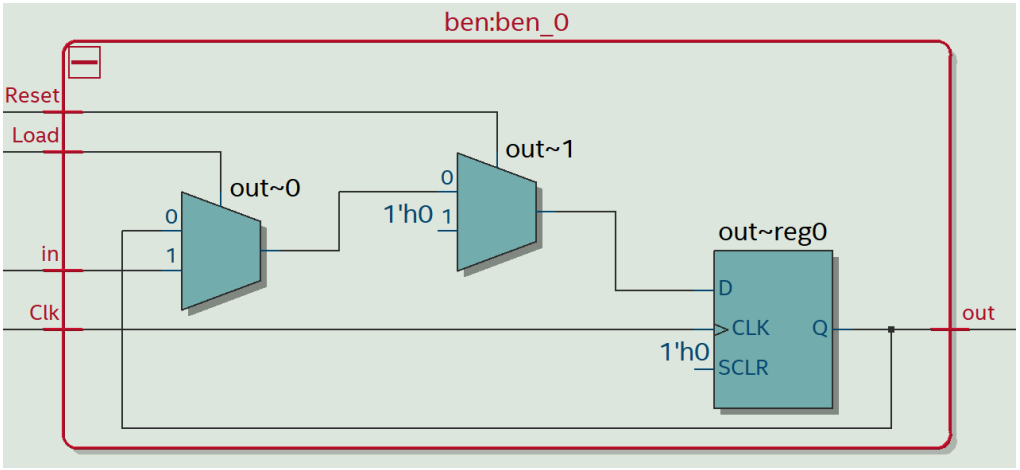
Module: ben

Input: Clk, Reset, Load, in

Output: out

Description: At the rising edge of clock, this module will either reset out value to 0 based on Reset, or load in into out based on Load, or keep out unchanged if neither Reset nor Load.

Purpose: This module reset or load value to ben.



Block Diagram for ben

Module: testbench_lab5

Input: None

Output: None

Description: This module does several tests to the program: Test 1, Test 2, Test 3, XOR Test, Multiplier Test, and Sort Test. It helps us compare the answer with the expected value.

Purpose: This module is used as a test bench for lab5.

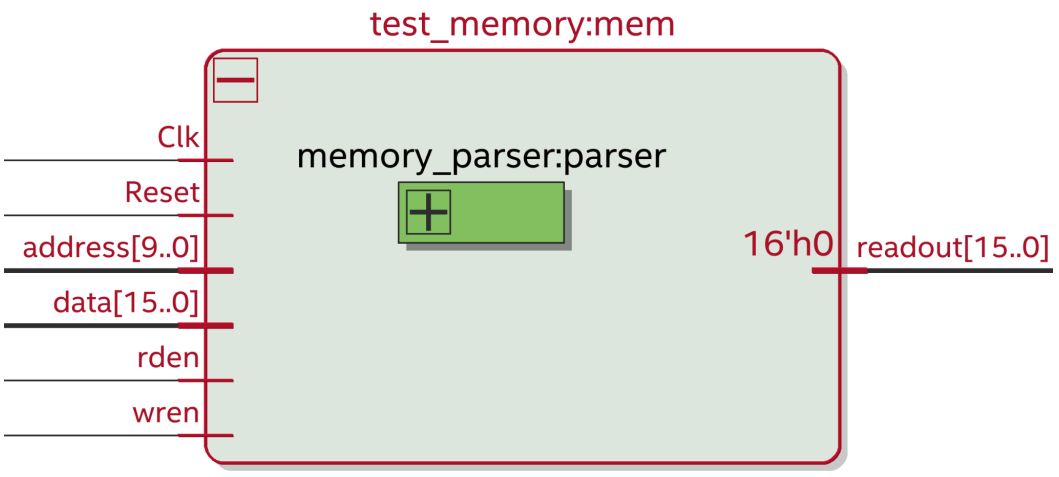
Module: test_memory

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

Output: [15:0] readout

Description: The memory module expands memory to 256 words and initializes the init_external to be 0. It parses memory_contents.mif into machine code and writes into file mem_array.

Purpose: This module helps us to do the simulation on quartus.



Block Diagram for test_memory

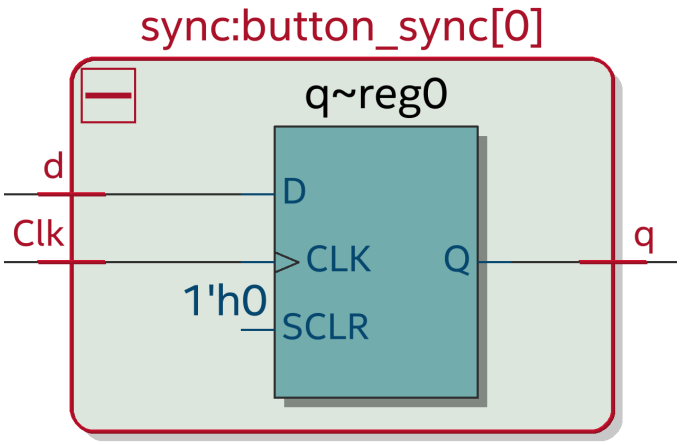
Module: sync

Input: Clk, d

Output: q

Description: This module will implement a Synchronizer, which could bring asynchronous signals into the FPGA and convert them into synchronous signals.

Purpose: Synchronize the asynchronous inputs from FPGA.



Block Diagram for sync

Module: slc3_testtop

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

Output: [9:0] LED, [6:0] HEX0, HEX1, HEX2, HEX3

Description: This module will load the value from switches to determine which operation should be done, and use Run and Continue to control the operation. The output will be displayed on Hexdriver and LED.

Purpose: This is the top level module for both simulation and synthesis using test_memory.

Module: slc3_sramtop

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

Output: [9:0] LED, [6:0] HEX0, HEX1, HEX2, HEX3

Description: This module will load the value from switches to determine which operation should be done, and use Run and Continue to control the operation. The output will be displayed on Hexdriver and LED.

Purpose: This is the top level module for synthesis using physical RAM.

Module: SLC3_2

Input: NONE

Output: NONE

Description: This module describes each operation explicitly. It defines the opcode of each operation and registers. It also tells the structure of each operation instruction.

Purpose: This module tells the instruction of each operation.

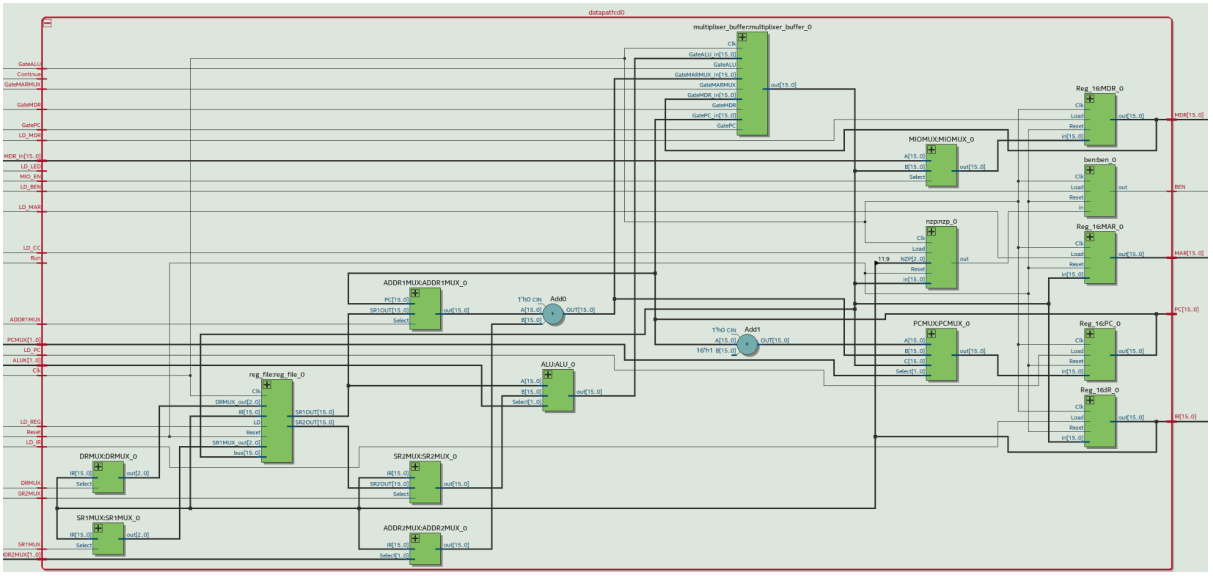
Module: slc3

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

Output: OE, WE, [9:0] LED, [6:0] HEX0, HEX1, HEX2, HEX3, [15:0] ADDR, [15:0] Data_to_SRAM

Description: This module is the slc3 program, which implement user's instruction and provides corresponding outputs. The slc3 program is composed of Led, Datapath, Mem2IO, and ISDU.

Purpose: This is the slc3 program that we want to implement.



Block Diagram for slc3

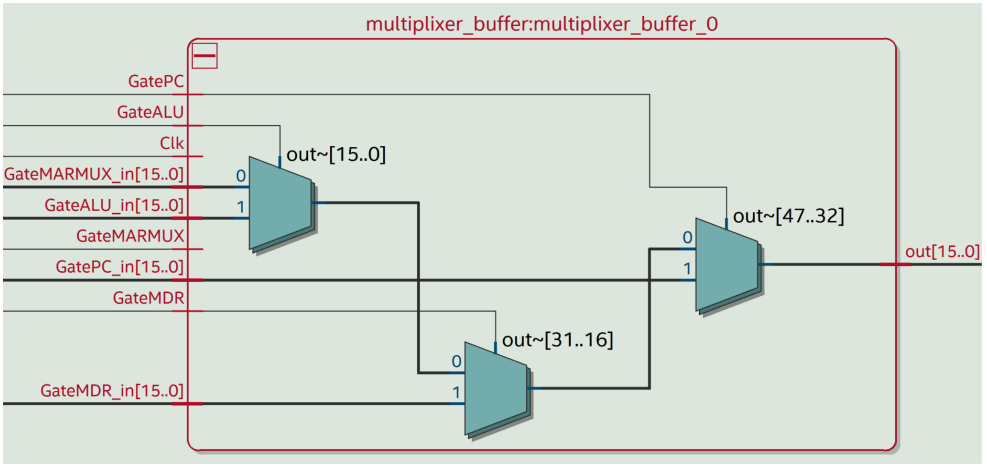
Module: multiplixer_buffer

Input: Clk, GatePC, GateMDR, GateALU, GateMARMUX, [15:0] GatePC_in, GateMDR_in, GateALU_in, GateMARMUX_in

Output: [15:0] out

Description: This module works as the tristate buffer in the SLC3 program. Since we do not have an actual tristate buffer in our FPGA, we use this 5 to 1 mux to implement the tristate buffer.

Purpose: This module works as the tristate buffer, connecting 4 gates to the BUS.



Block Diagram for multiplixer_buffer

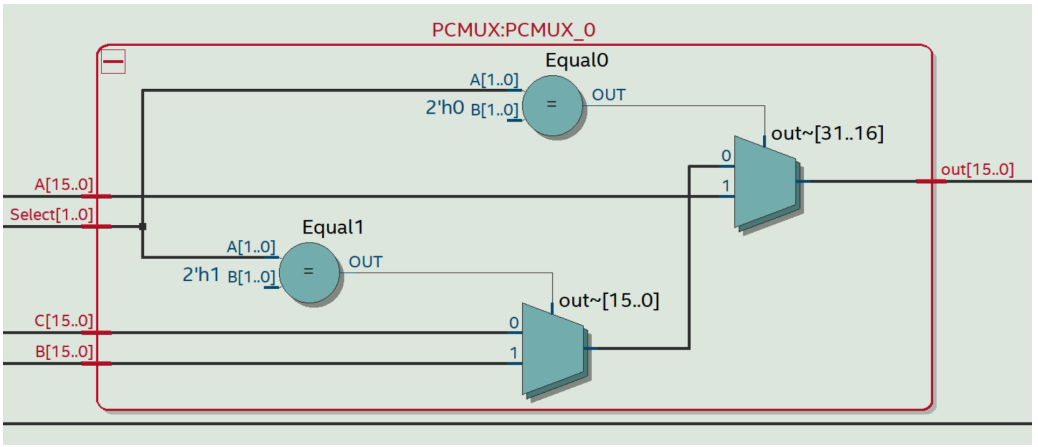
Module: PCMUX

Input: [1:0] Select, [15:0] A, B, C

Output: [15:0] out

Description: This module is the PCMUX, which select from 3 inputs: the BUS, the sum of ADDR1MUX and ADDR2MUX, and the incremented PC, and outputs the selected value into PC.

Purpose: This module works as the PCMUX.



Block Diagram for PCMUX

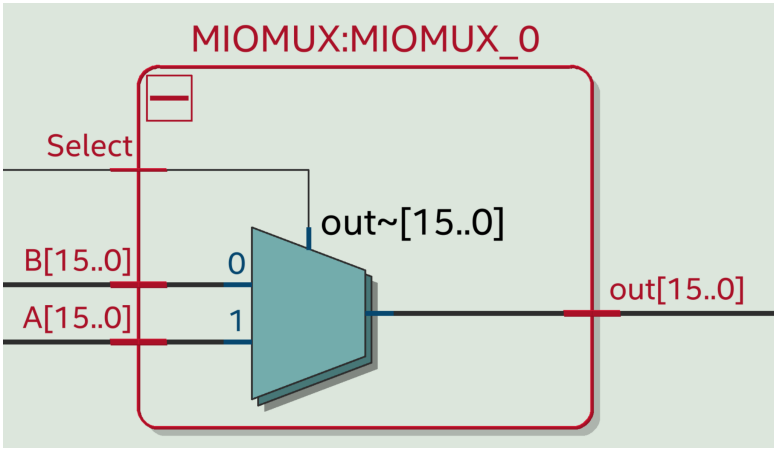
Module: MIOMUX

Input: Select, [15:0] A, B

Output: [15:0] out

Description: This module implements the MIOMUX, which takes two input: the BUS and Data_to_CPU, with select MIO.EN.

Purpose: This module will serve as MIOMUX, and output the selected value into MDR.



Block Diagram for MIOMUX

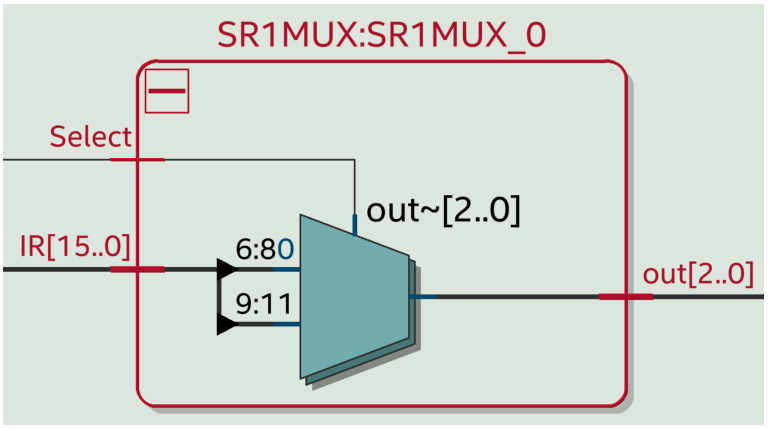
Module: SR1MUX

Input: Select, [15:0] IR

Output: [2:0] out

Description: This module implements the SR1MUX, which takes inputs from IR with Select SR1, and output the selected value, IR[8:6] or IR[11:9], to Regfile.

Purpose: This module will provide one input to the Regfile.



Block Diagram for SR1MUX

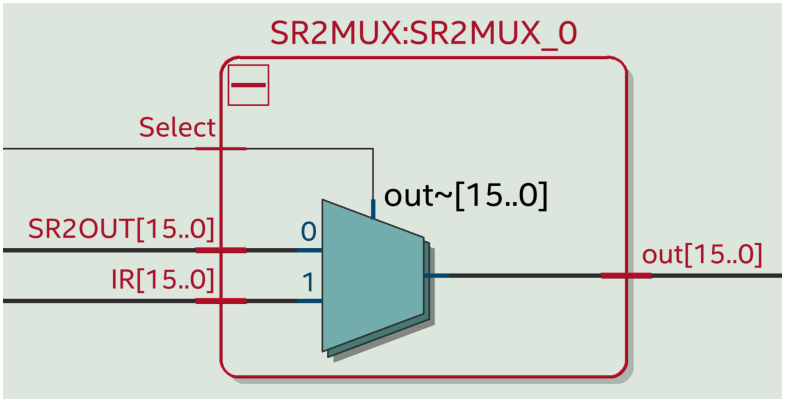
Module: SR2MUX

Input: Select, [15:0] SR2OUT, IR

Output: [15:0] out

Description: This module will implements the SR2MUX, which takes inputs from SR2OUT and sign extended IR[4:0] with Select and output the selected value to ALU.

Purpose: This module will serve as the SR2MUX and provide input to the ALU unit.



Block Diagram for SR2MUX

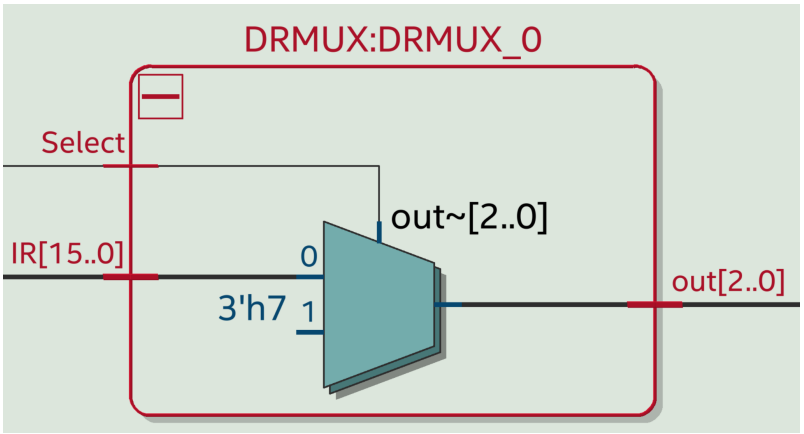
Module: DRMUX

Input: Select, [15:0] IR

Output: [2:0] out

Description: This module will implement the DRMUX, which take 2 inputs: 111 and IR[11:9] with Select.

Purpose: This module will provide an input to the Regfile.



Block Diagram for DRMUX

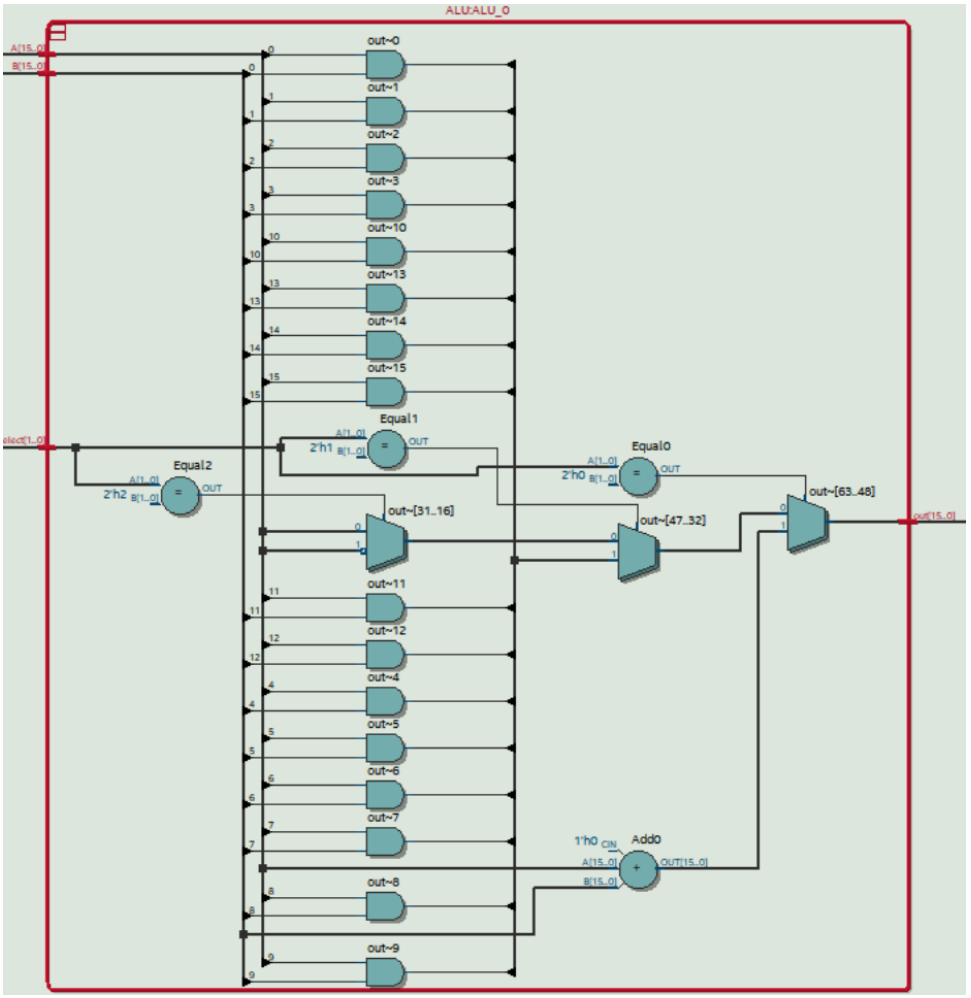
Module: ALU

Input: [15:0] A, B, [1:0] Select

Output: [15:0] out

Description: This module the ALU unit, which could operate 4 functions: ADD, AND, NOT, and pass, which just directly loads the input to the output. The 2-bit select will determine which function to implement.

Purpose: This module will implement the ALU unit and perform 4 functions.



Block Diagram for ALU

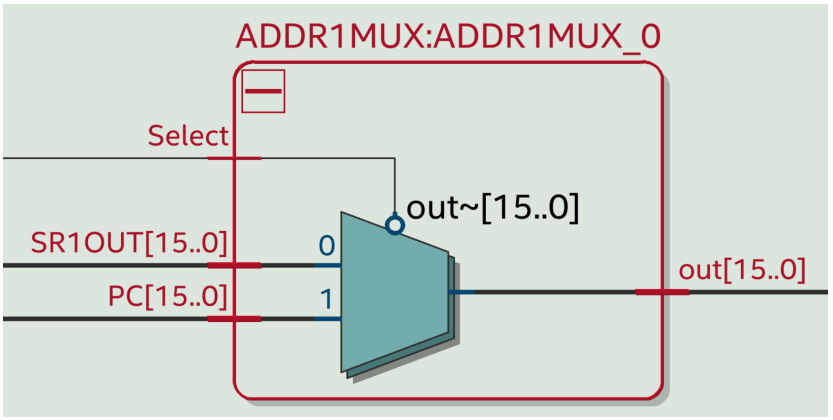
Module: ADDR1MUX

Input: Select, [15:0] PC, SR1OUT

Output: [15:0] out

Description: This module will implement the ADDR1MUX, which takes two inputs: SR1OUT and PC with Select signal from control unit.

Purpose: This module will implement the ADDR1MUX unit, and its output will be added with the output from ADDR2MUX.



Block Diagram for ADDR1MUX

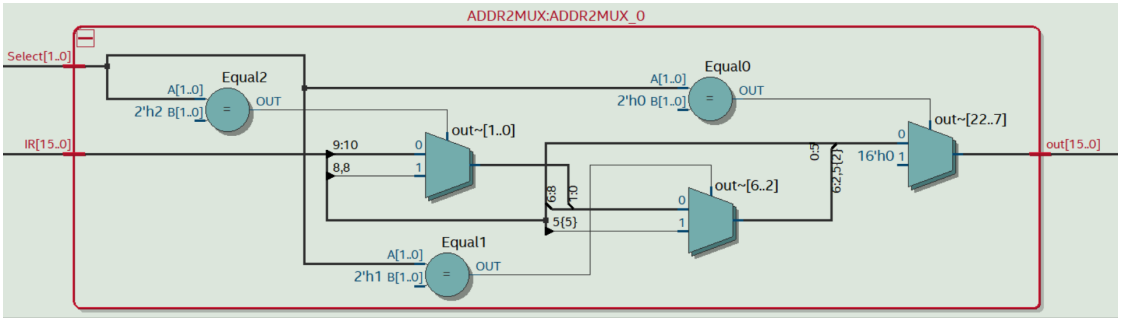
Module: ADDR2MUX

Input: [1:0] Select, [15:0] IR

Output: [15:0] out

Description: This module will implement the ADDR2MUX unit, which takes 4 inputs: sign-exteneded IR[10:0], sign-exteneded IR[8:0], sign-exteneded IR[5:0], and 16-bit 0 with 2-bit Select from control unit.

Purpose: This module will implement the ADDR2MUX unit, and its output will be added with the output from ADDR1MUX.



Block Diagram for ADDR2MUX

Module: memory_contents

Input: NONE

Output: NONE

Description: This module contains the test code that was provided. It includes Basic I/O test 1, Basic I/O test 2, Basic I/O test 3, XOR test, Multiplier test, and Sort test to help us check our program.

Purpose: This module provides test code.

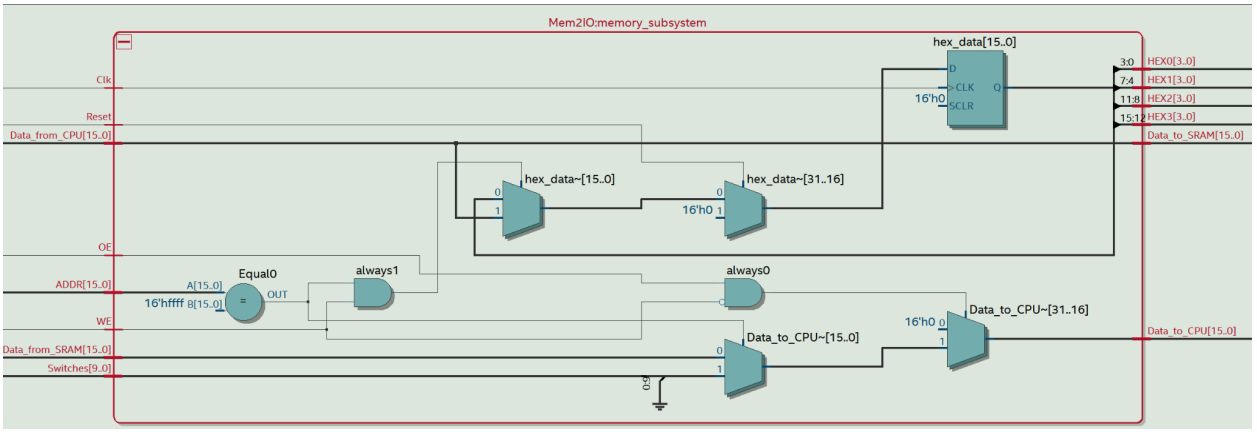
Module: Mem2IO

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

Output: [15:0] Data_to_CPU, Data_to_SRAM, [3:0] HEX0, HEX1, HEX2, HEX3

Description: This module loads data from switches when address is xFFFF, and from SRAM otherwise. It also writes to LEDs when WE is active and address is xFFFF. Hexdriver is also assigned in this module.

Purpose: When the address is xFFFF, the module loads instruction to CPU.



Block Diagram for Mem2IO

Module: ISDU

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

Output: 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, ALUK, Mem_OE, Mem_WE

Description: This module works as the state machine for the program. The detailed description could be found in the ISDU section below.

Purpose: This module will implements the State Machine for this SLC3 program.

Module: Instantiateram

Input: Reset, Clk

Output: wren, [15:0] ADDR, data

Description: This module will instantiate the on-chip memory based on the memory_contents.sv.

Purpose: This module will instantiate the on-chip memory.

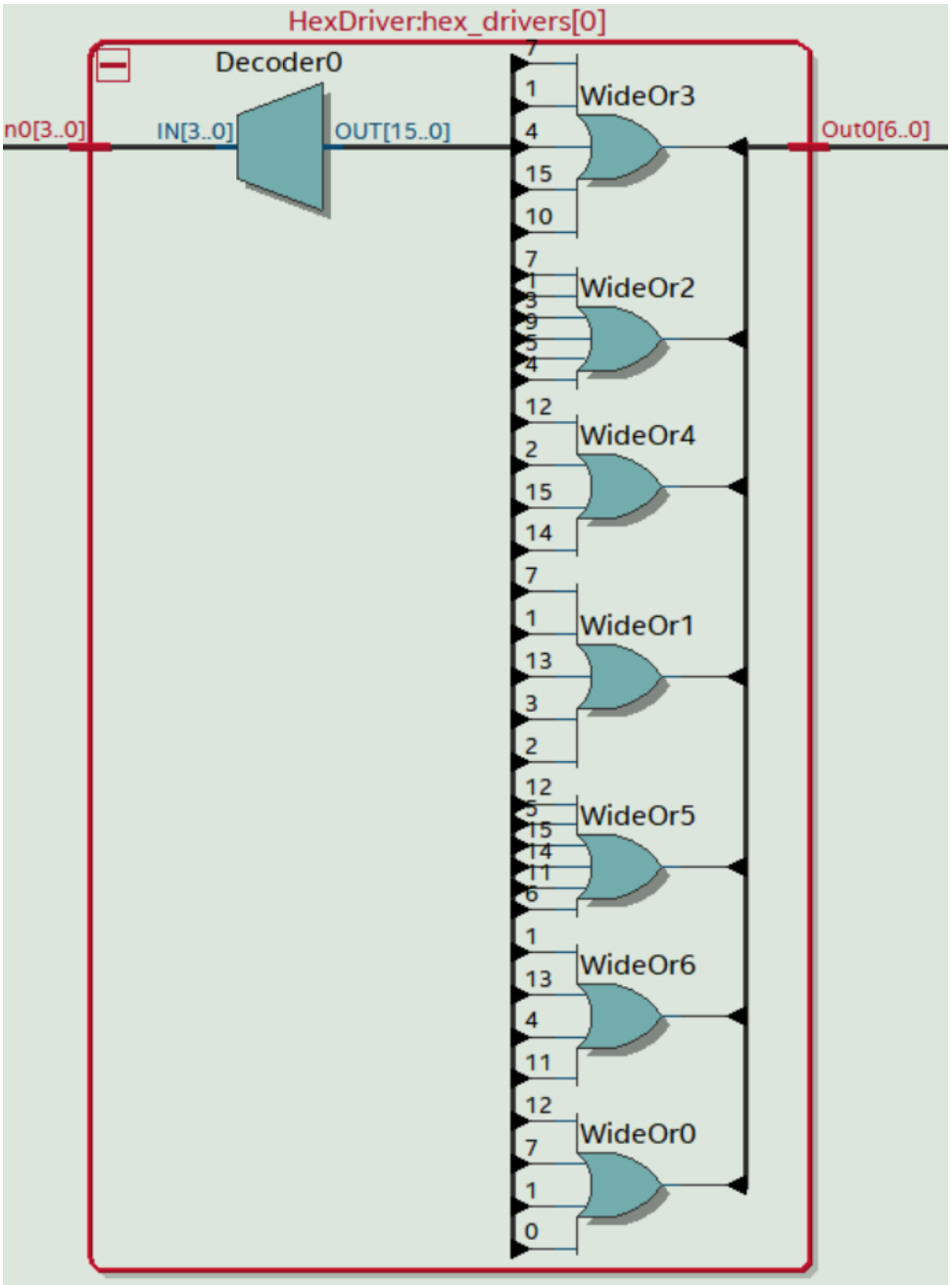
Module: HexDriver

Input: [3:0] In0

Output: [6:0] Out0

Description: This module will implement the HexDriver unit, which could convert the 4-bit input into hexadecimal.

Purpose: This module will implement convert 4-bit binary to Hexadecimal.



Block Diagram for HexDriver

Module: datapath

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

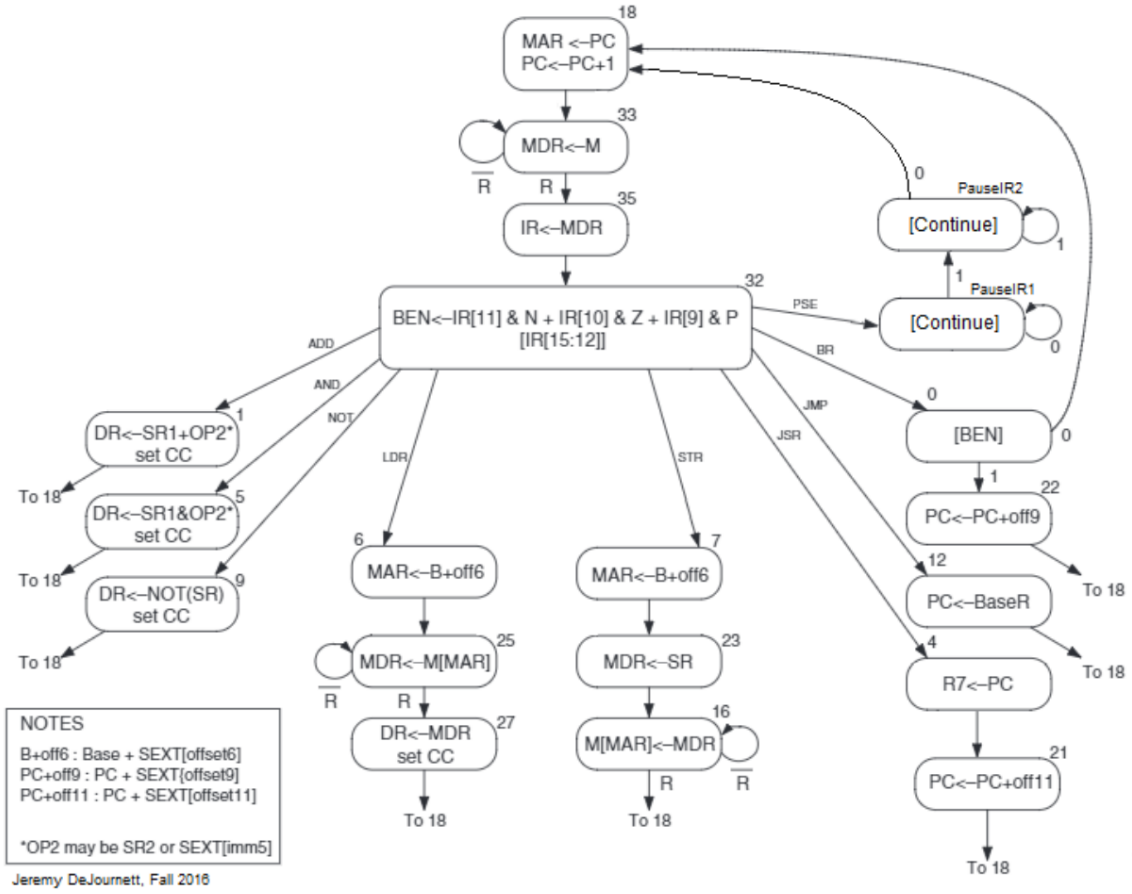
Output: BEN, [15:0] IR, MAR, MDR, PC

Description: This module is the datapath of our program. By using the datapath, the program can do the operations step by step. The output of one function can be the input of the other function.

Purpose: This module will build the datapath of the program.

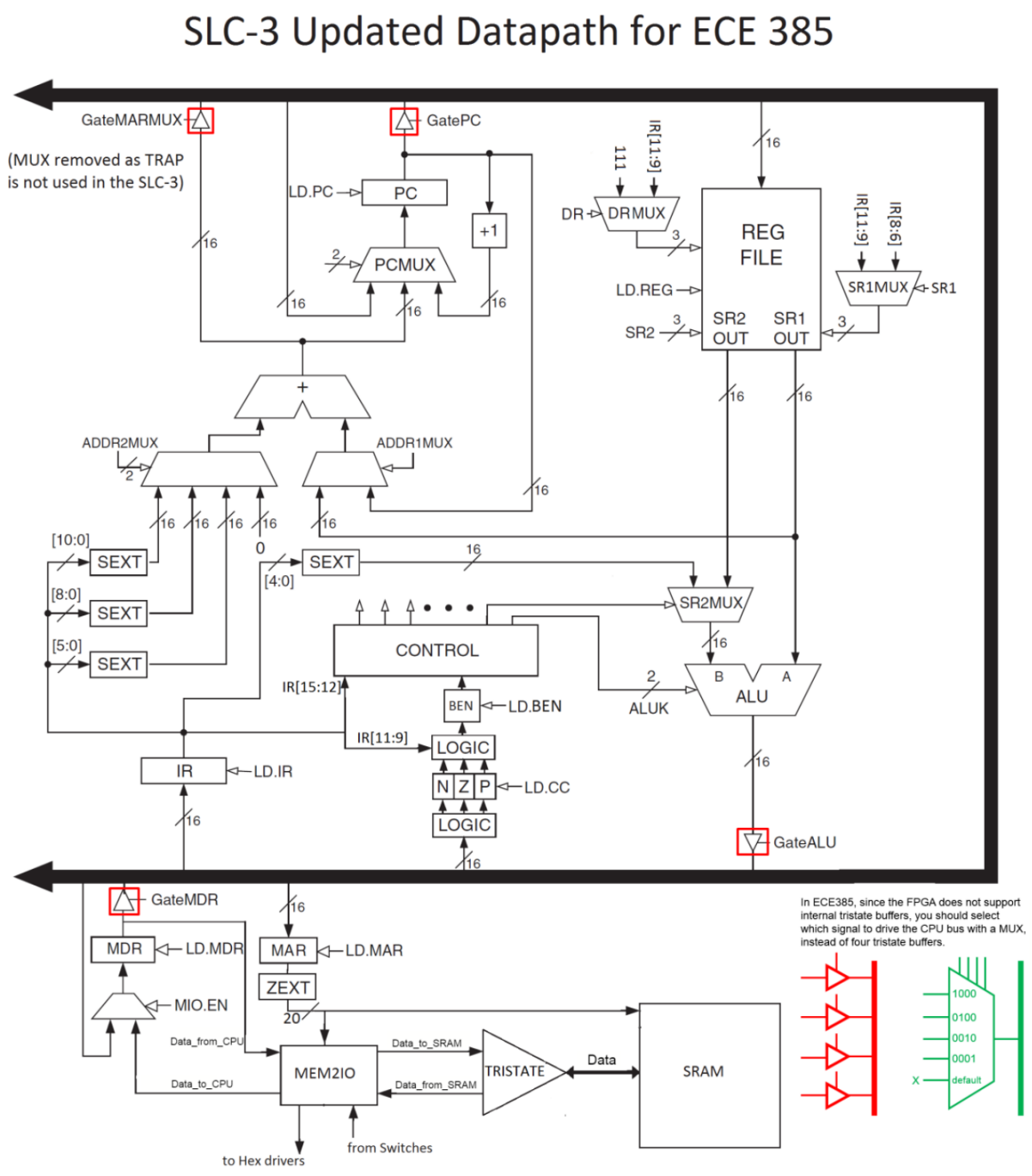
f. Description of the operation of the ISDU (Instruction Sequence Decoder Unit)

The ISDU module works as the control unit, which decodes instructions and sends out controls signals to other modules, in this slc3 program. The ISDU will take inputs from Instruction Register, the user's inputs, Reset signal, Clock signal, and Brench Enable. To implement this ISDU module, we utilizes two-always structure, which is one always-ff and one always-comb. The always-ff block will implement the transfer from current state to the next state on the rising edge of clock. The always-comb block will assign next state logic for each state based on the given slc3 state diagram below, and will also assign controls signals based on current states. One thing that distincts our ISDU design from the state diagram below is that in State33, State16, and State25, since we do not have a Ready signal in our design, we extend those three states for another two states, which make sure that the writing process is completed.



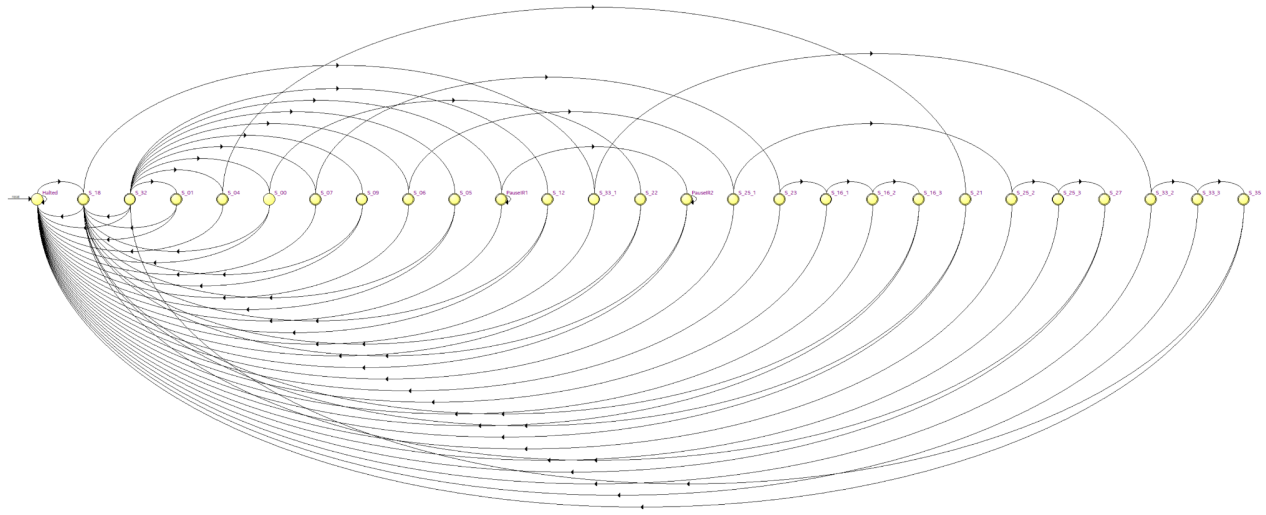
State Diagram of SLC3

To assign accurate control signals based on the current state, we also resort to the given SLC3 Datapath shown below. In our design, since we do not have tristate buffers in our FPGA, which are used to control 4 gates connected to the BUS, we uses a 5-to-1 MUX to implement this function.



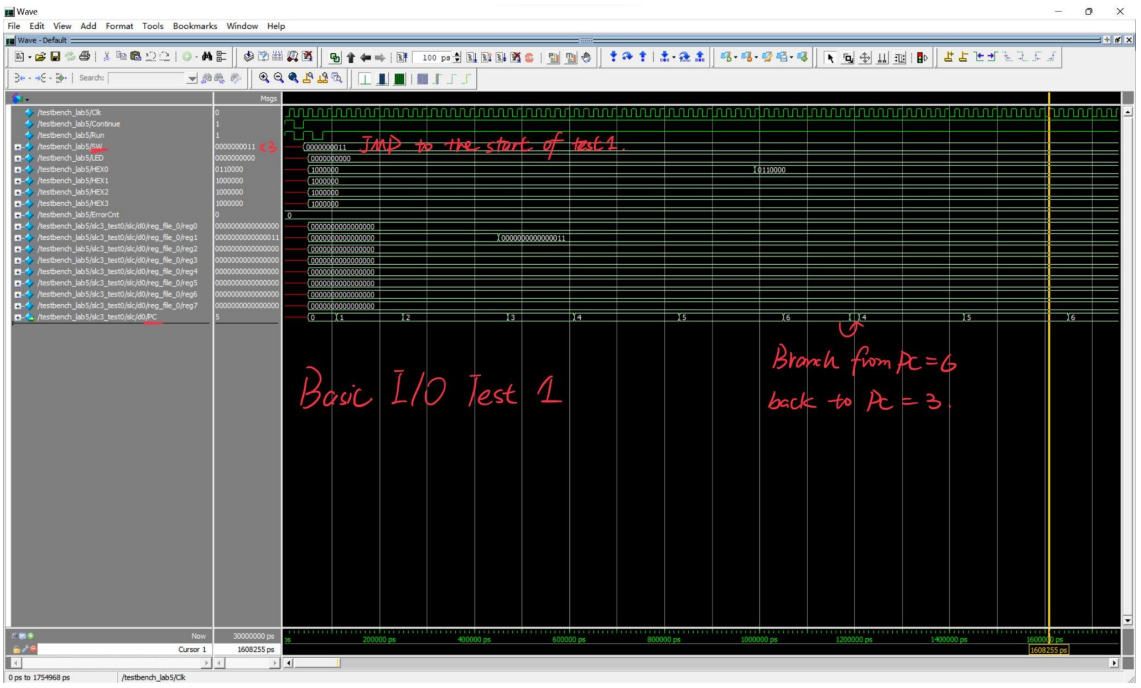
SLC3 Datapath

g. State Diagram of ISDU

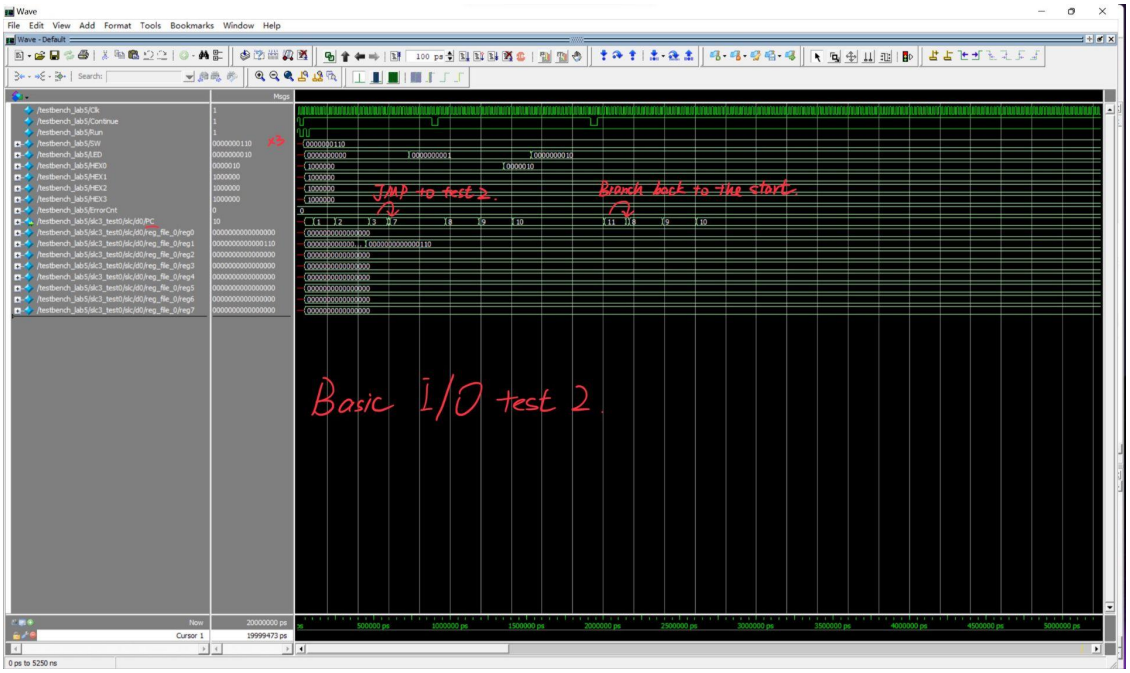


State Diagram of ISDU

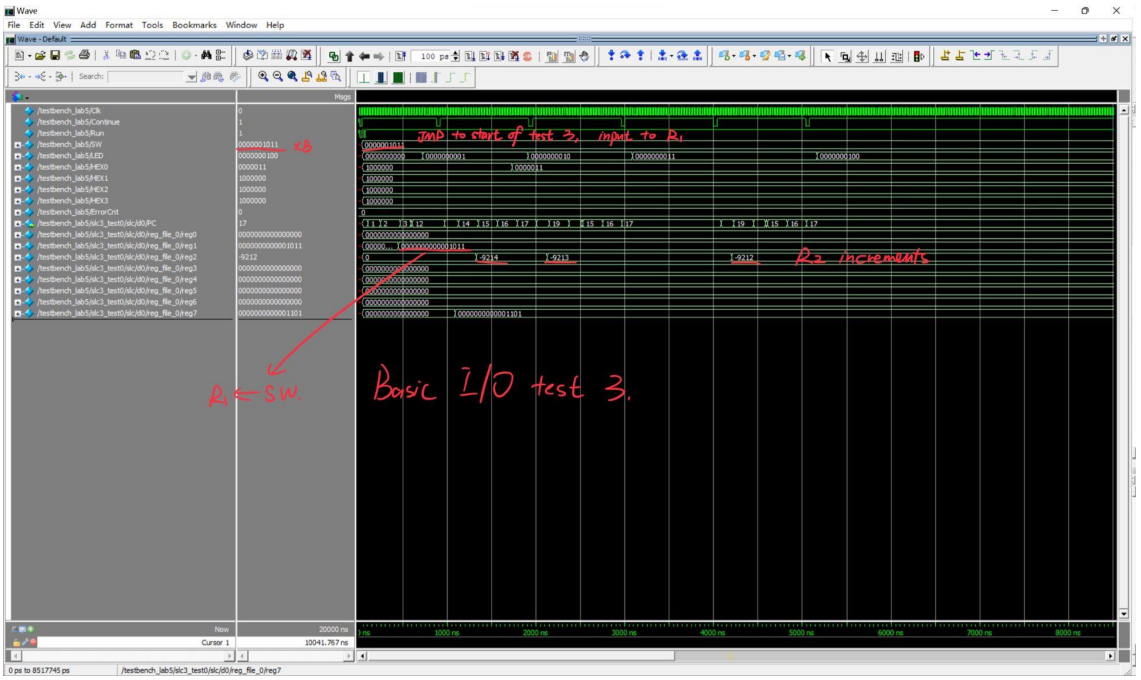
Simulations of SLC-3 Instructions



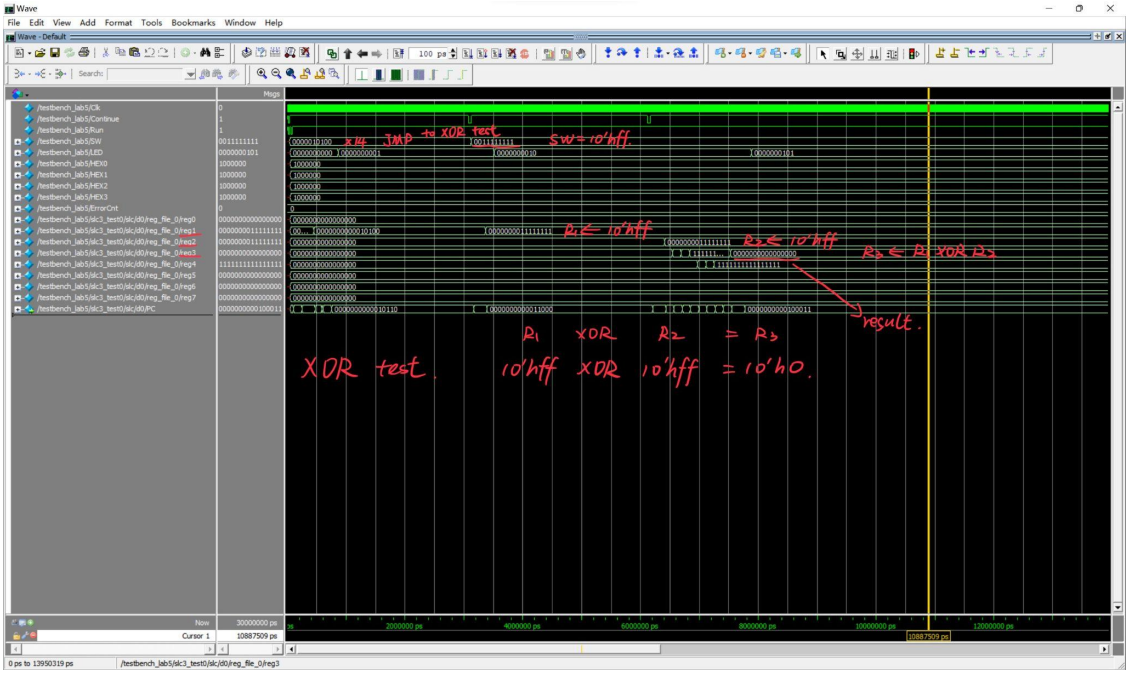
Annotated Basic I/O Test 1 Simulation



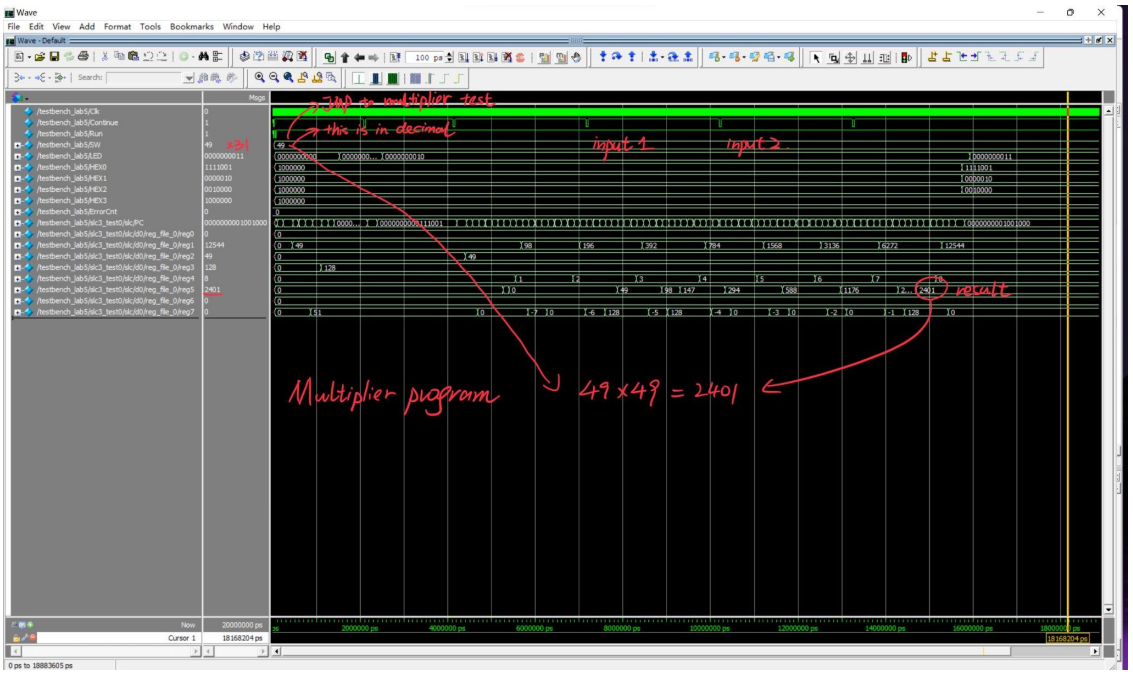
Annotated Basic I/O Test 2 Simulation



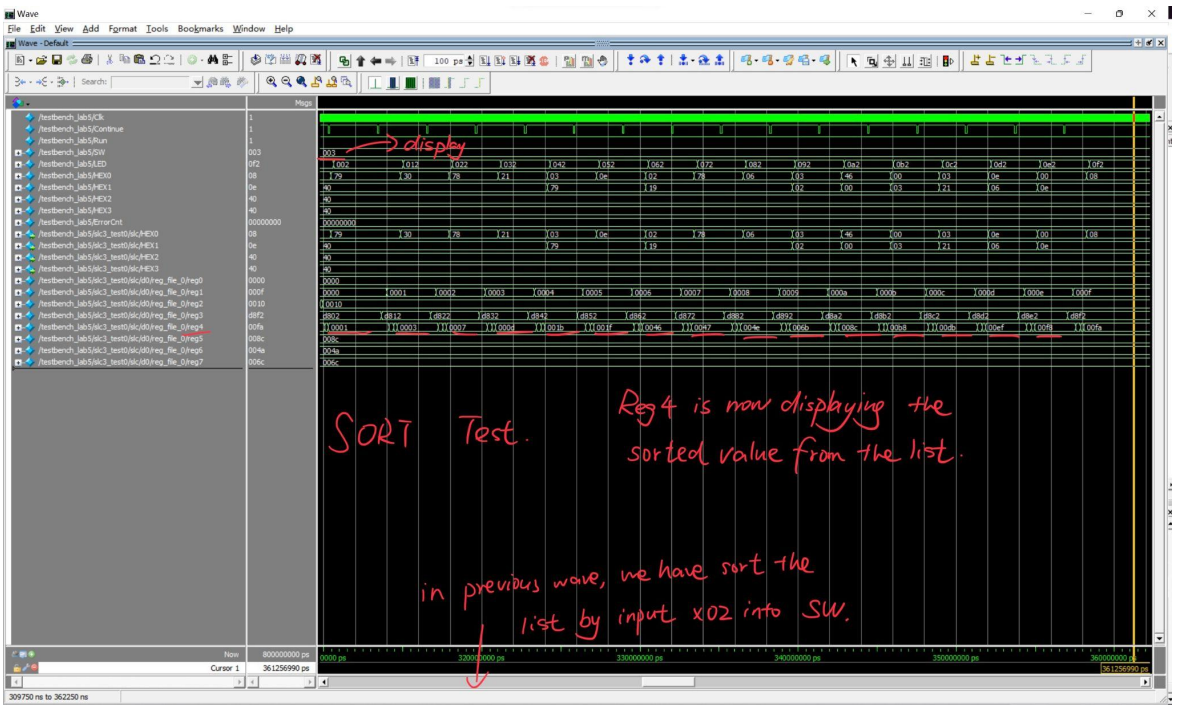
Annotated Basic I/O Test 3 Simulation



Annotated XOR Test Simulation



Annotated Multiplier Test Simulation



Annotated Sort Test Simulation

Post-Lab Questions

Design Resources and Statistics table

LUT	520
DSP	no DSP
Memory (BRAM)	1677312
Flip-Flop	251
Frequency	69.38 MHz
Static Power	89.98 mW
Dynamic Power	4.79 mW
Total Power	106.12 mW

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

The main function of Mem2IO is to load data from switches when address is xFFFF, and from SRAM otherwise. It also writes to LEDs when WE is active and the address is xFFFF. Mem2IO also assigns Hexdriver.

What is the difference between BR and JMP instructions?

For JMP, the program directly goes to the address which is the memory of BaseR. For BR, we need first to check the condition code. If the condition code fits the condition of BR, meaning branch enable, the program goes to the address which is PC + SEXT(PCoffset9).

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

The transition to the next state is asynchronous. It happens at the rising edge of the R signal. To compensate for the lack of the signal in our design, we let the state run for several clock cycles to make sure it reaches the next state. The implication is the state has to wait for 3 clock cycles to get it synchronized.

Conclusion

Our design is a SLC-3 program. It can perform 9 different operations: ADD, AND, NOT, BR, JMP, JSR, LDR, STR, PAUSE. We input the instruction code via switches on the FPGA board. The computer will first fetch the instruction from the memory, decode it to determine the type of the instruction, execute the instruction by the different operations, and then fetch again. In the process of building the program, we have some problems with the register file. We didn't know how exactly it works at first. After reading the SLC-3 datapath carefully, we find that when LD is 1, we can load the memory to the related register based on the value of DR, and using SR1MUX and SR2MUX to get the value of SR1OUT and SR2OUT. For this lab, I think both the state diagram and datapath are very useful. The state diagram can help us build the state machine and do the operations step by step. The datapath diagram can help us know how the SLC3 works and the components we should use to do the operation.

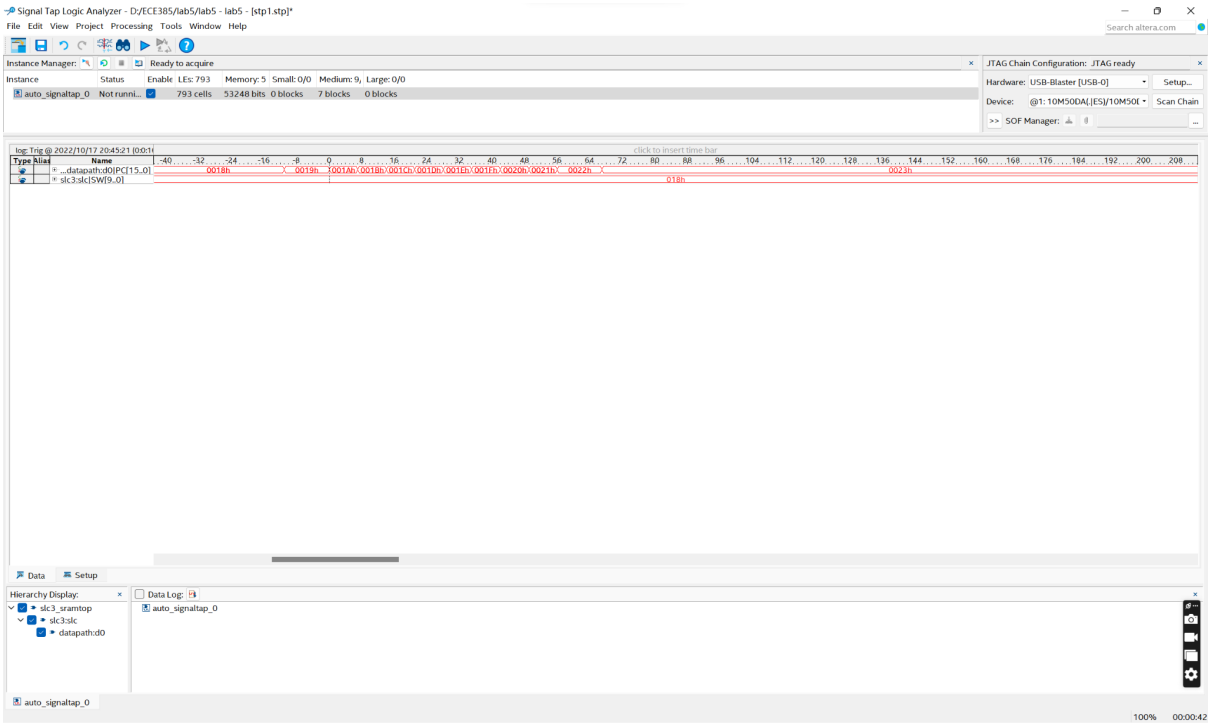
Extra Credit Assignment

XOR Test Signal Tap Analysis

Number of instructions: 15 (From 0x1A to 0x29).

Number of cycles: 67.

CPU is capable of 11.194 MIPS (millions of instructions per second) on the XOR test (15 instructions / 67 cycles * 50,000,000 cycles per second)



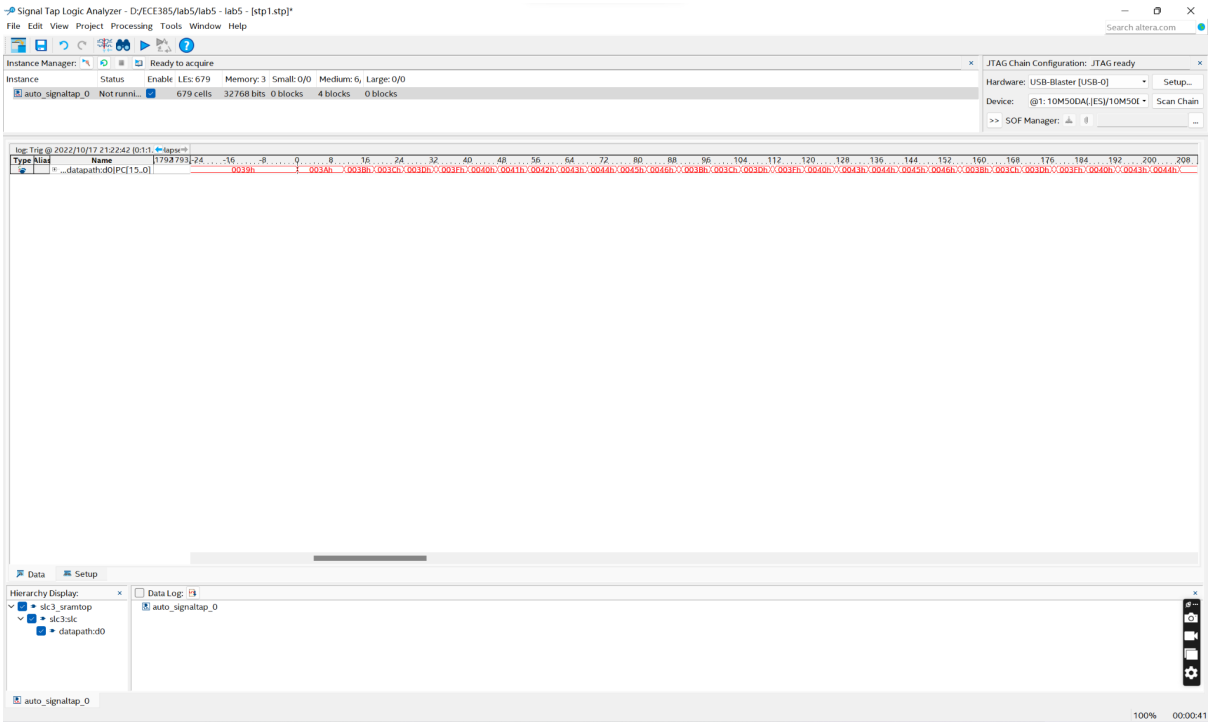
Signal Tap Anlysis for XOR Test

Multiplier Test Signal Tap Analysis

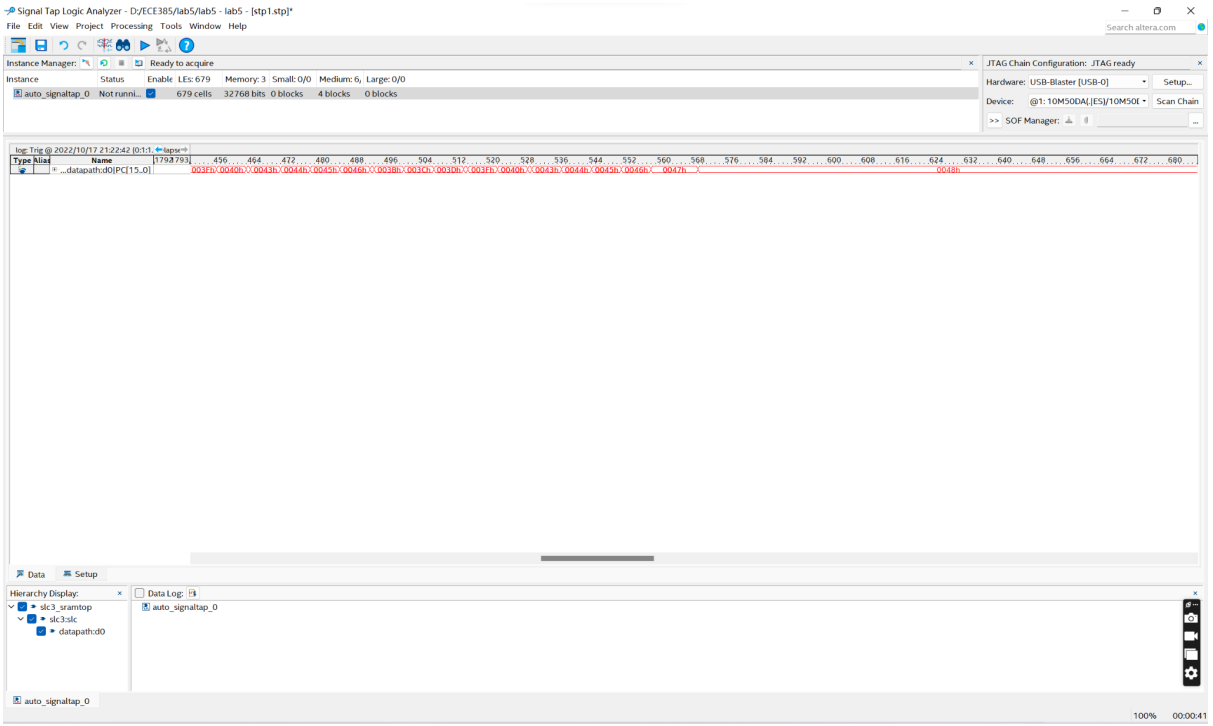
Number of instructions: 122 (Estmated based inputs to FPGA).

Number of cycles: 568.

CPU is capable of 10.739 MIPS (millions of instructions per second) on the XOR test (122 instructions / 568 cycles * 50,000,000 cycles per second)



Start of Signal Tap Anlysis for Multiplier Test



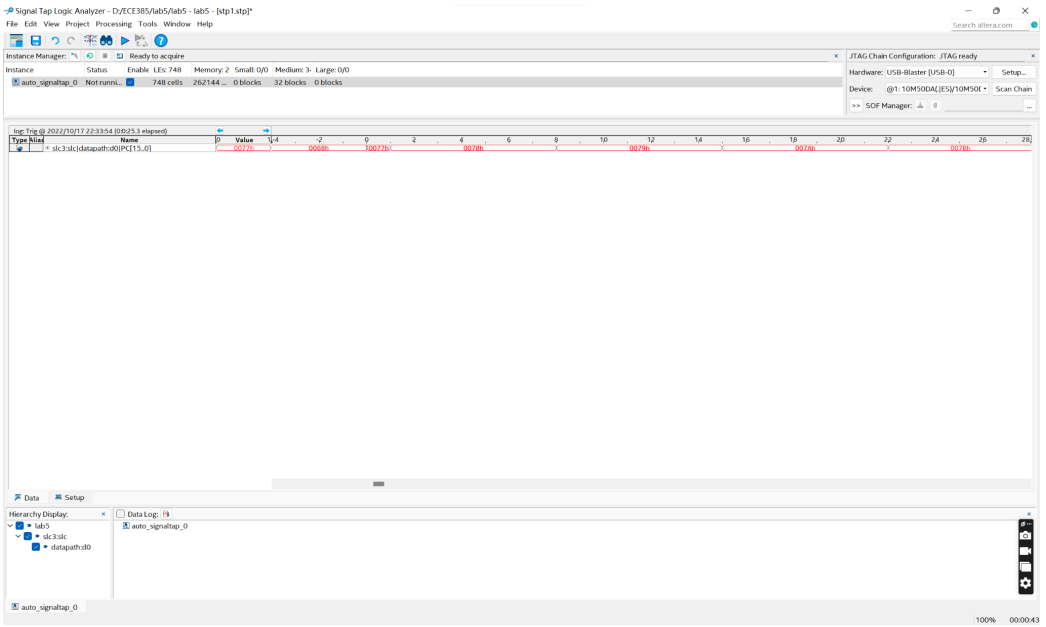
End of Signal Tap Anlysis for Multiplier Test

Sort Test Signal Tap Analysis

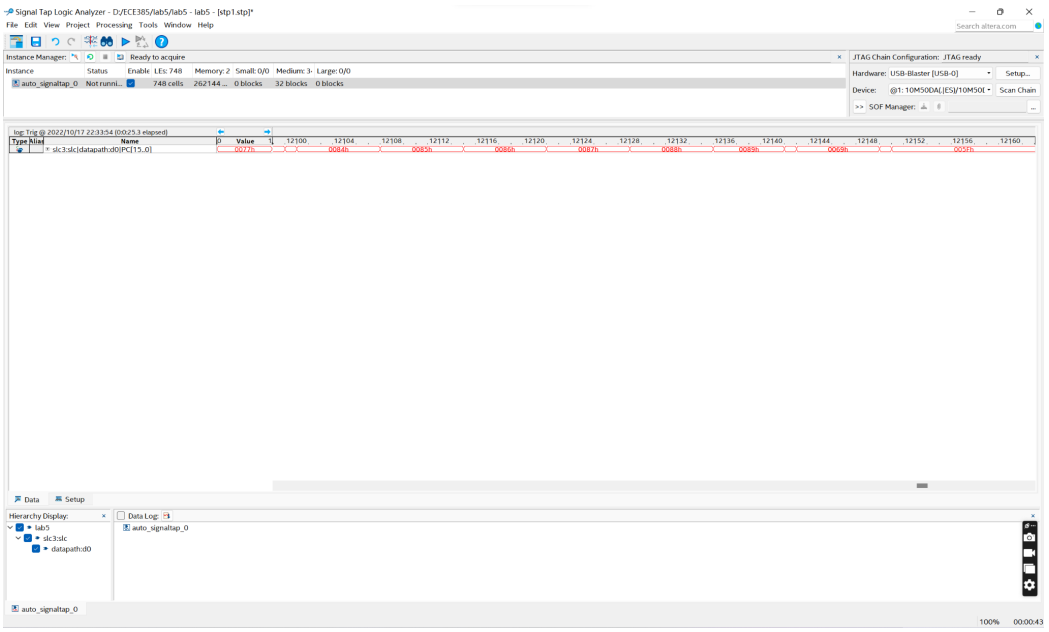
Number of instructions: 2176 (N(n+n-1+n-2+...2+1), where we use N=16 and n=16 to estimate number of instructions).

Number of cycles: 12150.

CPU is capable of 8.955 MIPS (millions of instructions per second) on the XOR test (2176 instructions / 12150 cycles * 50,000,000 cycles per second)



Start of Signal Tap Analysis for Sort Test



End of Signal Tap Analysis for Sort Test

Average performance of three tests: (11.194+10.739+8.955)/3 = 10.296 MIPS. Calculations of number of instruction are include in three modules above.