ECE 385

Fall 2020

Experiment #6

Simple Computer SLC-3.2 in System Verilog

Name: Xiao Shuhong & Lu Yicheng
Lab Section: D225

1. Introduction

1.1. Summarize the basic functionality of the SLC-3 processor

In this lab, we design a simple microprocessor using System Verilog. Three main components form the main design of this processor: the central processing unit (CPU), the memory that stores instructions and data and the I/O interface that communicates with external devices. Our SLC-3 will perform various operations based on the opcodes. As a subset of LC-3 design, we have opcode: ADD, ADDi, AND, ANDi, NOT, BR, JMP, JSR, LDR, STR, PAUSE.

For the basic working mode, our computer will first fetch an instruction from the memory, decode it to determine the type of the instruction, execute the instruction, and the fetch again.

2. Written Description and Diagrams of SLC-3

2.1. Summary of Operation

For this lab, we have three kinds of operations: FETCH, DECODE and EXECUTE. Computer first fetch an instruction from the memory, decode it to determine the type of the instruction, execute the instruction, and the fetch again. For the working detail, we will discuss in the next part.

We have three input: Reset, Run and Continue. For Reset being press, we clear both our 16 bits IR and PC to 16’h0000 and state machine will go back to the Halt state and stop; for Run being press, we begin loop FETCH-DECODE-EXECUTE; the Continue will only be use when we meet PAUSE, if continue is not press, computer will stop when it read PAUSE, otherwise it keep read the next information in the memory.

2.2. Describe function cycle

After the Run Button is press, we will begin the three-operation cycle for each line of instructions in the memory.

For Fetch, we load the PC to MAR as the address to read the instruction from; then we load the instruction read from the memory with address in MAR and store it to MDR; then we sent our instruction from MDR to IR which used for decode; finally we increment PC by 1 for the next FETCH.

For DECODE, we read instruction form IR and decode it, there are 11 instructions we will use for our SLC-3:

ADD and ADDi, with opcode 0001, IR [11:9] as DR, IR [8:6] as SR1; if IR [5]=0, it is ADD and IR [2:0] as SR2, else if IR [5]=1, it is ADDi, IR [4:0] as imm5.

AND and ANDi, with opcode 0101, IR [11:9] as DR, IR [8:6] as SR1; if IR [5]=0, it is AND and IR [2:0] as SR2, else if IR [5]=1, it is ANDi, IR [4:0] as imm5.

NOT, with opcode 1001, IR [11:9] as DR, IR [8:6] as SR.

BR, with opcode 0000, IR [11:9] as nzp, IR [8:0] as PCoffset9.

JMP, with opcode 1100, IR [8:0] as BaseR.

JSR, with opcode 0100, IR [10:0] as PCoffset11.

LDR, with opcode 0110, IR [11:9] as DR, IR [8:6] as BaseR, IR [5:0] as offset6.

STR, with opcode 0111, IR [11:9] as SR, IR [8:6] as BaseR, IR [5:0] as offset6.

PAUSE, with opcode 1101, IR [11:0] as ledVect12.

For EXECUTE, we perform the operation based on the signal from the ISDU and write the result to the destination register or memory, the detailed function for each instruction are shown below.

ADD: add the data in register SR1 and SR2 and store to register DR.

ADDi: add the data in register SR and a binary number imm5, store the result to register DR.

AND: do AND operation between the date in register SR1 and SR2, store the result to register DR.

ANDi: do AND operation between the data in register SR and binary number imm5, store the result to register DR.

NOT: do NOT operation to the data in register SR and store the result to register DR.

BR: compare nzp with NZP, if one of three meets, we change PC to PC+PCoffset9.

JMP: change PC to the address store in register BaseR.

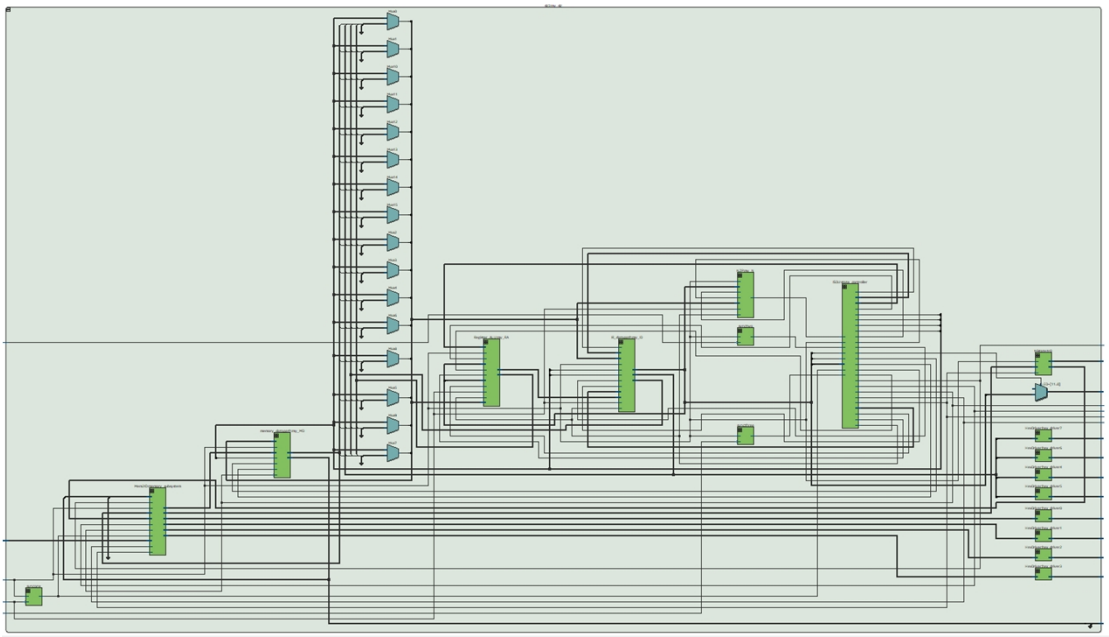
JSR: we first store original PC to register R7 and change PC to PC+PCoffset11.

LDR: we first address by add the data in register BaseR and binary number offset6, then we read the data in this address from memory and store it to register DR.

STR: we first read the data in register SR, then get the address by add the data in register SR and binary number offset6, the store the data read to the address in memory.

PAUSE: we show ledVect12 in LEDs on board and wait until Continue button is press.

2.3. Block Diagram of slc3.sv



2.4. Written Description of all .sv modules

Module: lab6_toplevel.sv

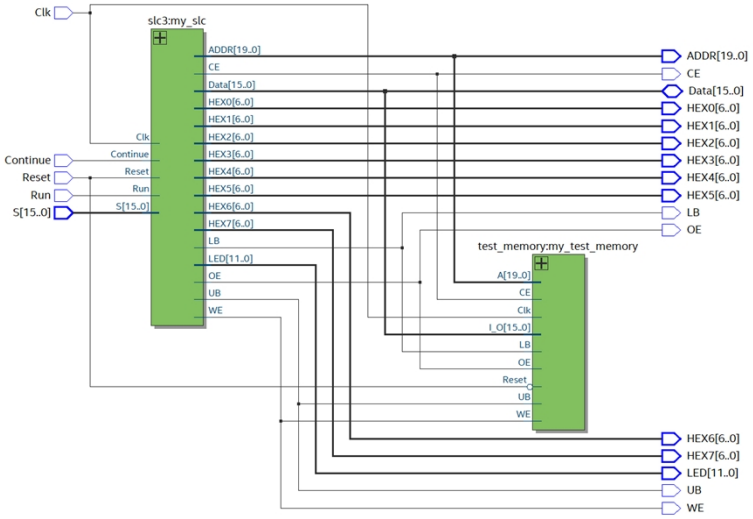
Inputs: [15:0] S, Clk, Reset, Run, Continue

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

INOUT: [15:0] Data

Description: This is the top level of our lab6, contains two parts: slc-3 process and memory.

Purpose: This .sv connect our slc-3 processor and memory.



Module: slc3.sv

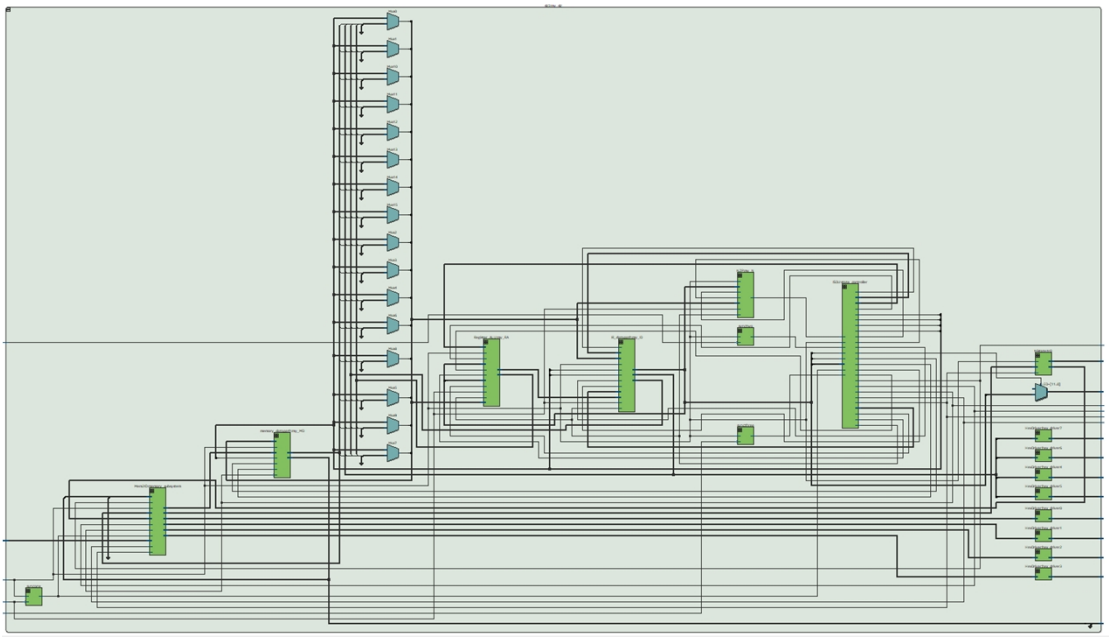
Inputs: [15:0] S, Clk, Reset, Run, Continue

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

INOUT: [15:0] Data

Description: This is our slc-3 processor module contains hexdrivers, Synchronizers, our Datapath (Memory Datapath, IR Datapath, Register File, ALU and NZP), mux to choose which data to bus, Men2IO, tristate and ISDU.

Purpose: This .sv connect our memory interface, IR, ALU, NZP and register file together to construct the hardware to achieve FETCH-DECODER-EXECTURE cycle.



Module: test_memory.sv

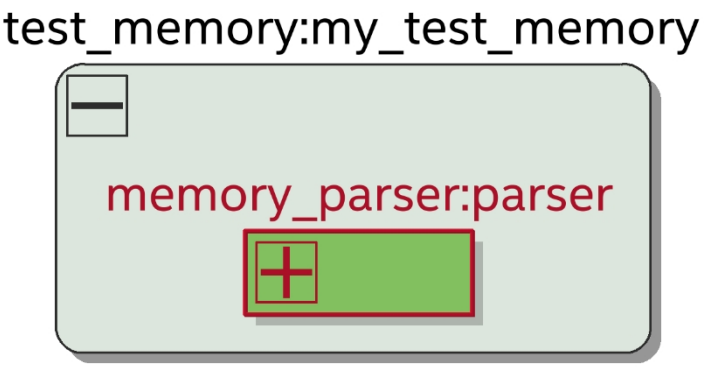
Inputs: Clk, Reset, [19:0] A, CE, UB, LB, OE, WE

Outputs: None

INOUT: [15:0] I_O

Description: This is the simulated memory used for our wave simulation only, a module memory_contents is added to build the entire memory part.

Purpose: This .sv construct the simulated memory with the contains claimed in module memory_contents which we can read and write.



Module: memory_contens.sv

Inputs: None

Outputs: None

Description: This .sv claims the instruction and corresponding address stored in our simulated memory used for wave simulation.

Purpose: This .sv help construct the memory, which we can modify.

Module: SLC3_2.sv

Inputs: None

Outputs: None

Description: This .sv is the memory interface of our design, it connects the memory datapath we defined and our simulated SARM. Also, it leaves connections from switch and to display. When the input memory is xFFFF, the switch number will be read and display, otherwise it stores the data into the SRAM with the address input. For data output from SRAM, it just goes through this put and output to our memory datapath.

Purpose: This module works as an interface to help us communicate with SARM.

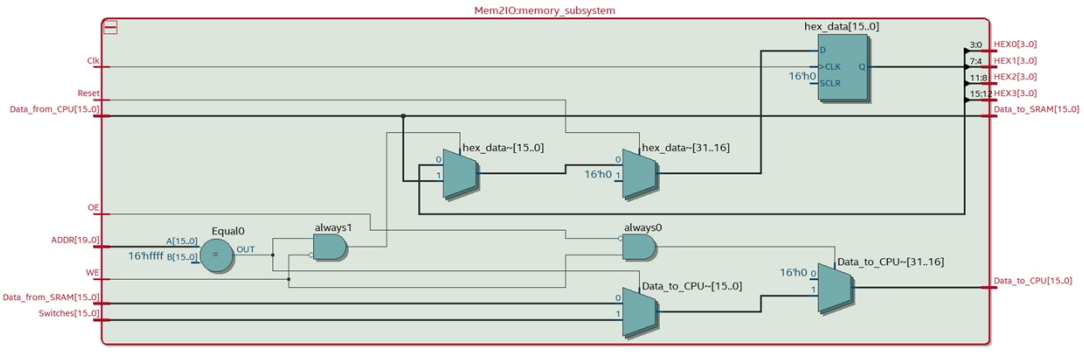
Module: Mem2IO.sv

Inputs: Clk, Reset, [19:0] ADDR, CE, UB, LB, OE, WE, [15:0] Switches, [15:0] Data_from_CPU, [15:0] Data_from_SRAM

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

Description: This .sv will be used as library when decode our instruction, it contain all the binary expression of our 11 instructions.

Purpose: To decode our instruction in IR, then the corresponding register will be used and store right thing in operation EXECUTE.



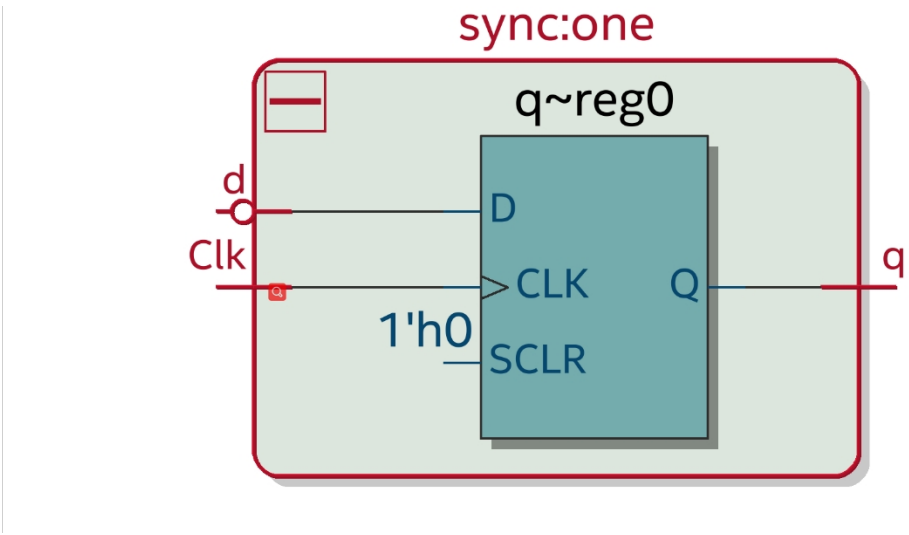
Module: Synchronizers.sv

Inputs: Clk, d

Outputs: q

Description: This .sv contains three different synchronizers. For this lab, we only use the simplest one. It adds a flip flop between input d and output q.

Purpose: This .sv help prevent potential glitch by changing combination input or output to ff one.



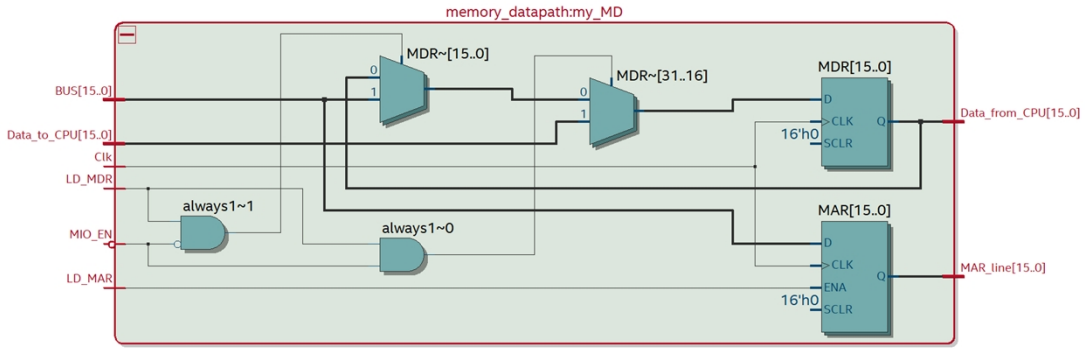
Module: memory_datapath.sv

Inputs: GateMDR, LD_MDR, MIO_EN, LD_MAR, Clk, [15:0] Data_to_CPU, [15:0] BUS

Outputs: [15:0] Data_from_CPU, [15:0] MAR_line

Description: This datapath achieve data (PC) from BUS to MAR, data from BUS to MDR, MAR output and MDR output for the memory part. And all the internal choose logic are implemented.

Purpose: This is the datapath that connect MAR, MDR and BUS together, output of this part will be connected to memory part and BUS.



Module: tristate.sv

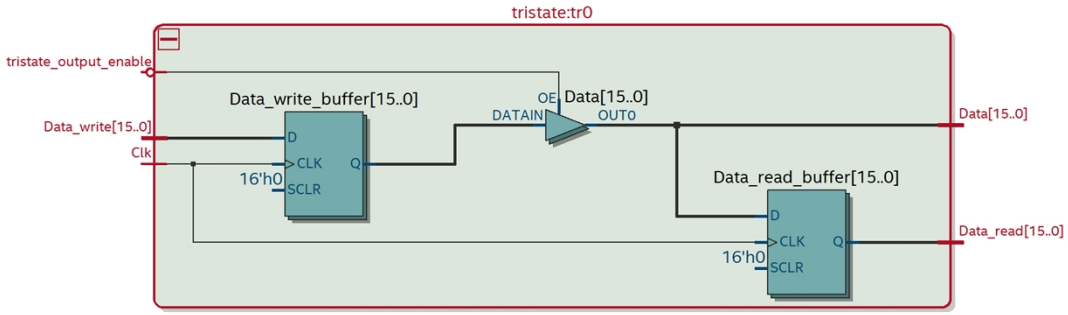
Inputs: Clk, tristate_output_enable, [15:0] Data_write

Outputs: [15:0] Data_read

INOUT: [15:0] Data

Description: This is a tristate buffer for ASRM, two buffer inside are used to store the data as a register, if enable signal is true, tristate allow data pass on both side, otherwise, data path connects tristate and SRAM will be stopped.

Purpose: This tristate is added to avoid mess happened in the wire path Data to avoid unexpected behavior.



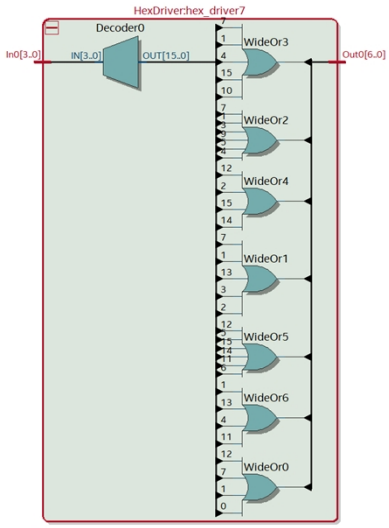
Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0.

Description: This module achieves a mux to exchange output of our circuit to 7-bit display signal.

Purpose: This module helps to light the 7 bits segment display on the FPGA board.



Module: IR_datapath.sv

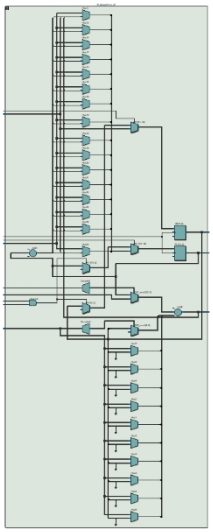
Inputs: GATEMARMUX, GATEPC, LD_PC, ADDR1MUX, LD_IR, reset, Clk, unique_reset, [1:0] PCMUX, [1:0] ADDR2MUX, [15:0] SR1OUT

Outputs: [15:0] IR_line, [15:0] PC_line, [15:0] sum

INOUT: [15:0] BUS

Description: This is the part of Datapath that include PC part and IR part. With IR loaded from BUS, this part achieves 3 kinds of IR splitting and add with register number or PC to output or use for PC update. Also, the PC update logic is contained, with choice of increment by 1, update with the result from IR or read from BUS.

Purpose: This is part of our entire datapath connected PC, IR with outside.



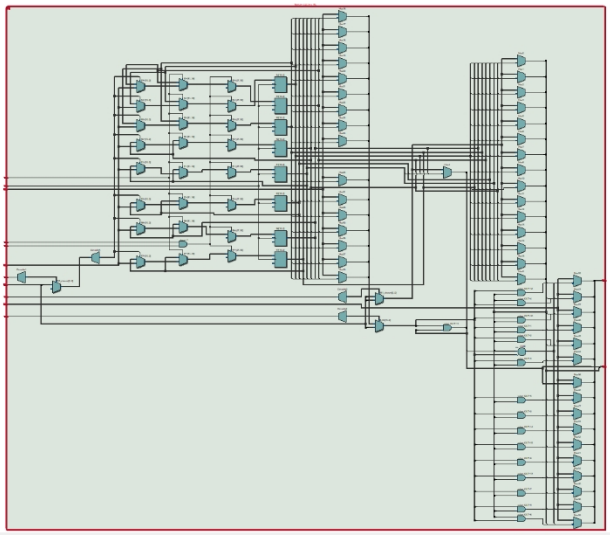
Module: Register_ALU.sv

Inputs: Clk, DR, SR1, LD_REG, SR2MUX, reset, unique_reset, [15:0] value_IN, [2:0]SR2, [1:0]ALUK, [15:0]IR

Outputs: [15:0]value_OUT, [15:0]SR1_OUT

Description: For this part, we design a register file to store and update our 7 registers and an algorithm logic unit.

Purpose: This part will be used when EXECUTE, the data in corresponding register will be use and store, and the calculation is processed by ALU.



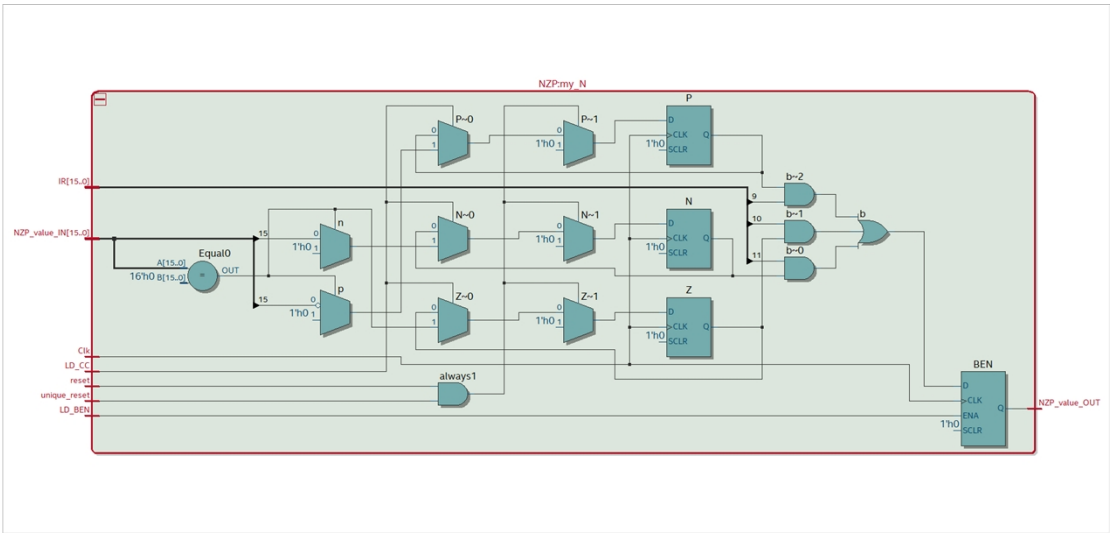
Module: NZP.sv

Inputs: Clk, LD_BEN, LD_CC, [15:0] IR, [15:0] NZP_value_IN, reset, unique_reset

Outputs: NZP_value_OUT

Description: This part contains a combinational logic to check the 16-bit binary number, whether it is negative, zero or positive. And compare it with the corresponding bit in coming instruction (Br) to decide enable signal for BEN.

Purpose: This first logic will check every binary number in BUS whether it is negative, zero or positive and the LD.BEN will be active only when instruction BR.



Module: ISDU.sv

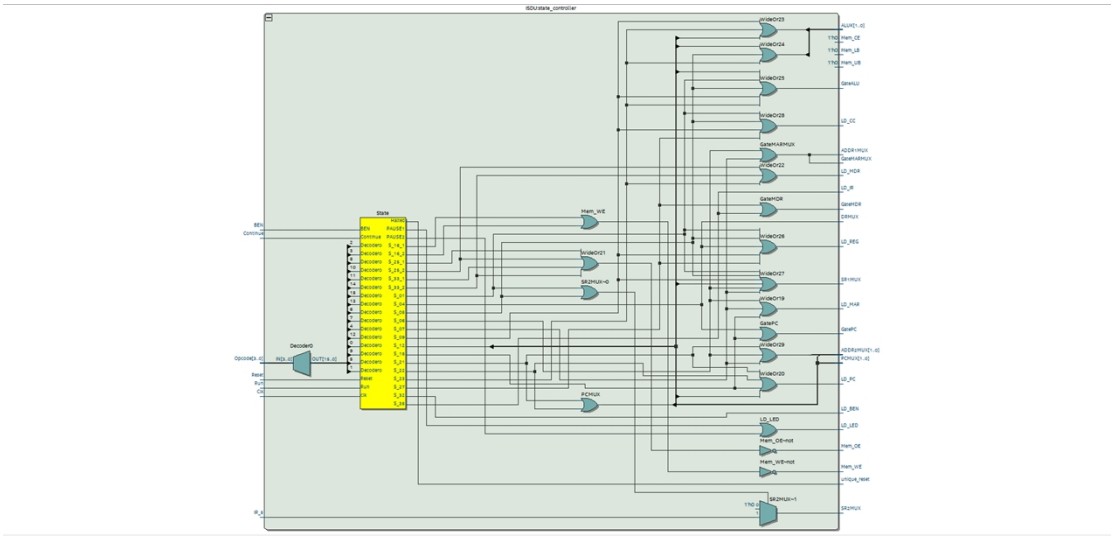
Inputs: Clk, Reset, Run, Continue, IR_5, IR_11, BEN, [3:0] Opcode

Outputs: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, DRMUX, SR1MUX, SR2MUX,

ADDR1MUX, ALUK, Mem_CE, Mem_UB, Mem_LB, Mem_OE, Mem_WE, unique_reset, [1:0] PCMUX, [1:0] ADDR2MUX,

Description: This is our state machine part to achieve the FETCH-DECODE-EXECUTE cycle for our 11 different instructions. In each state, the corresponding control LD signal, Gate signal, Mux signal and Men signal will be set. For state detailed description, we will discuss them at 2.5.

Purpose: This part describes the state change for every instruction and set the corresponding control signal at each state.



2.5. Description of the operation of the ISDU

For our ISDU, we use this part to achieve a state machine for operation cycle. For all the state transition condition, if we do not mention, we define them run one clock cycle and go to next state automatically.

Before the first FETCH, we have an initial state which we named as **Halted**. At this state, we turn all the control signal of except unique_reset which used for Reset operation.

Then, after we press Run, we begin FETCH, at FETCH, we have 4 states, each take one clock cycle: **S_18**, **S_33_1**, **S_33_2**, **S_35**. For **S_18**, we turn GatePC, LD_MAR, LD_PC to 1 and PCMUX to 00 which load PC to MAR, increment PC by 1. For **S_33_1**, **S_33_2,** together we search the data in the address from MAR and load it to MDR, as the existing of tristate buffer, we need two clock cycle to finish memory read, so we turn Mem_OE to 0 at the first state and keep Mem_OE and turn LD_MDR to 1 at second state. For **S_35**, we load MDR to IR, so we turn GateMDR and LD_IR both to 1.

After Fetch, we get IR store our instruction, so we do DECODE, which only contain one state: **S_32**, it only takes one clock cycle and go to next state automatically. For this state, we first check opcode IR [15:12] so we could decide which specific instruction to go in EXECUTE. For opcode 0001, go to ADD or ADDi, 0101 go to AND or ANDi, 1001 go to NOT, 0000 go to BR, 1100 go to JMP, 0100 go to JSR, 0110 go to LDR, 0111 go to STR, 1101 go to

PAUSE. Then, we also decide BEN signal which we have discussed in part about our NZP module.

Then, for EXECUTE part, we discuss our 11 instructions one by one.

For opcode 0001 we go to ADD(i) instruction which is S_01, we do here turn SR1MUX 1, DRMUX 0, SR2MUX IR_5, ALUK 00, GateALU 1, LD_REG 1, LD_CC 1. Then we finish and go to S_18 which begin next FETCH.

For opcode 0101, we go to AND(i), which is S_05 here. We turn SR1MUX 1, DRMUX 0, SR2MUX IR_5, ALUK 01, GateALU 1, LD_REG 1, LD_CC 1. Then we finish and go to S_18 which begin next FETCH.

For opcode 1001, we go to NOT, which is state S_09 here, we turn SR1MUX 1, DRMUX 0, ALUK 10, GateALU 1, LD_CC 1, LD_REG 1. Then we finish and go to S_18 which begin next FETCH.

For opcode 0000, we go the BR, first we check the signal BEN which gained from state S_32 at state S_00. If BEN is 0, we back to S_18 which begin next FETCH, otherwise we go to state S_22, increment PC by offset9, we turn ADDR1MUX 0, ADDR2MUX 10, PCMUX 10, LD_PC 1. Then we finish and go to S_18 which begin next FETCH.

For opcode 1100, we go to JMP, which is S_12 here. We change PC to number in BaseR, we turn SR1MUX 1, PCMUX 01, GateALU 1, ALUK 11, LD_PC 1. Then we finish and go to S_18 which begin next FETCH.

For opcode 0100, we go to JSR, which begin with state S_04, here we load PC to R7, so we turn DRMUX 1, GatePC 1, LD_REG 1. Then check IR [11], if 1, we go to state S_21, otherwise we will not discuss it in this lab. In S_21, we increment PC by offset11, so we turn ADDR1MUX 0, ADDR2MUX 11, PCMUX 10, LD_PC 1. Then we finish and go to S_18 which begin next FETCH.

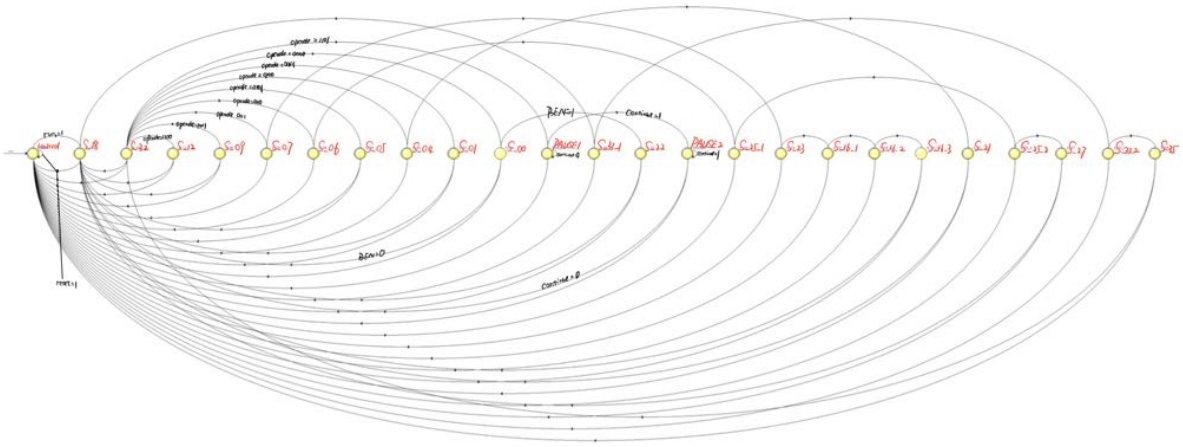
For opcode 0110, we go to LDR, which begin with state S_06. In S_06, we load BaseR+offset6 to MAR, so turn SR1MUX 1, GateMARMUX 1, ADDR1MUX 1, ADDR2MUX 01, LD_MAR 1. Then we go to state S_25_1 and S_25_2, which we read data from memory same as what we do in S_33, at first state, we turn Mem_OE 0, keep Mem_OE and turn LD_MDR 1 at second one. Then we go to state S_27, which we used to store MDR to DR and set CC. So, we turn DRMUX 0, LD_REG 1, LD_CC 1, GateMDR 1. Then we finish and go to S_18 which begin next FETCH.

For 0111, we go to STR, begin with state S_07 which is same as S_06, turn SR1MUX 1, GateMARMUX 1, ADDR1MUX 1, ADDR2MUX 01, LD_MAR 1. Then we go to state S_23, which load SR to MDR, we turn ALUK 11, SR1MUX 0, LD_MDR 1, GateALU 1. Then we go to state S_16_1 and S_16_2, which store our data to memory, still because the buffer, we need two state to finish, at both state, we just turn Mem_WE to 0. Then we finish and go to S_18 which begin next FETCH.

For opcode 1101 we go to PAUSE, which is a stop state and only when we push Continue button, we are able to do next FETCH. We have state PAUSE1 and PAUSE2 here. For both states, we turn LD_LED to 1. In PAUSE1, if we do not press Continue, it just stays here, until we press it and we go to

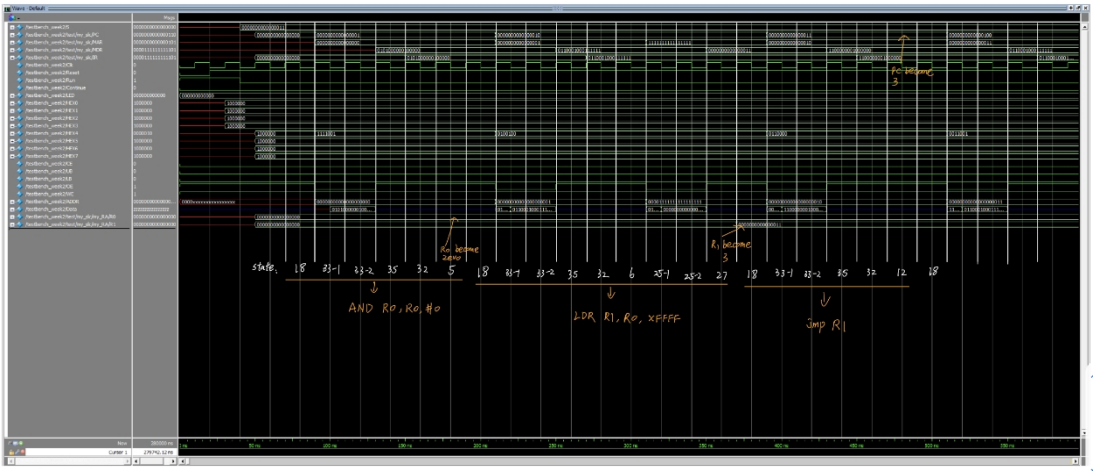
PAUSE2, before we loose the button, we will keep PAUSE2. Then we finish and go to S_18 which begin next FETCH.

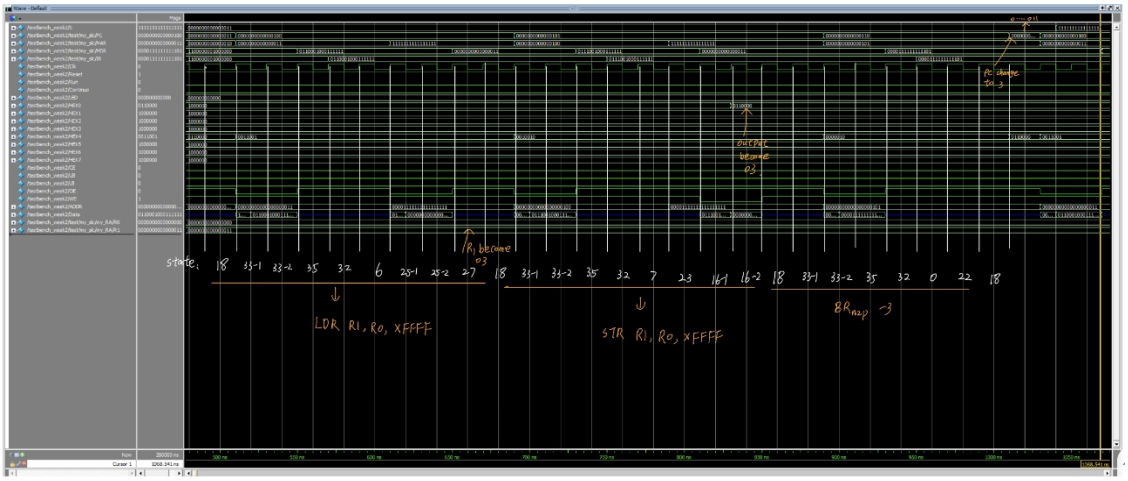
2.6. State Diagram of ISDU



3. simulation of SL-3 instruction

3.1





4. post lab question

4.1

LUT	371+238=609
DSP	0
BRAM	0
Flip-flop	272
Frequency	65.66 MHz
Static Power	98.66 mW
Dynamic Power	8.61 mW
Total Power	177.84 mW

Problem encountered:

1,One problem we meet is that we do not know how to bugs after we finish the whole design since we can only see the results with original waveform. In order to find wrong places, we drag out all the internal signals and use commands “restart -f || run 280us” to see the waveform of the internal signals.

2, Another problem we face is that we can not get expected results when running testbench. This is not because we have the wrong design but because we use 3 cycles to run read or write operations instead of 2. So, we need more time to see the results. Some of the test seems only work well for 2 cycles. So, after we large the time scale, we get the correct results.

4.2

What is MEM2IO used for?

In our opinion, MEM2IO is used to map memory. So that for CPU, we can ignore what external devices are connected to CPU. What CPU need is just store or read the memory.

In our lab, when CPU trying to store data to XFFFF, MEM2IO will display the data. When CPU trying to load data from XFFFF, MEM2IO will give SWITH value to CPU. If the address is not XFFFF, MEM2IO will just act as a bridge connected with CPU and SRAM.

What is the difference between BR and JMP instruction?

BR allows for conditions. However, BR can only change PC in a relatively small range. PC-2^8 < PC < PC+2^8-1

JMP is unconditional but can change PC freely. In fact, we can assign any 16 bits value to PC.

Also, the way to change PC is different. For BR, PC=PC + offset.

For JMP, PC=M(R).

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

Since CPU can not know whether we have already received the data from SRAM or change the data in SRAM. We have to use R signal sent form SRAM to CPU to tell CPU it is now ok to transfer to next state.

In our lab, we just give store, load operations multiple clock cycles. After waiting for that long time, we can guarantee that SRAM have already finished its job.

Since we wait multiple cycles for read and write operation, we can just read the data at rising edge and move to next state. So, it is better for synchronization.

5, conclusion

In this lab, we build a simplified LC3. To build the whole circuit, we separate it by several parts. They are:

1, memory Datapath -- containing the ports that connected to SRAM

2, ISDU -- containing the state machine

3, IR Datapath -- containing the IR, PC.

4, Register ALU -- containing R0-R7,ALU.

5, NZP -- containing the NZP, BEN registers and corresponding combinational logic.

Also, Our CPU supports 9 instructions and can easily connect to external device, like SDRAM, SWITCH, DISPLAY.

We think everything in the lab manual is quite clear and there is no room for improvement.