ECE385

Spring 2019

Experiment #6
SLC3-Processor

Austin Lee (sal3)

Zuling Chen (zulingc2)

Patrick Wang, Gene Shiue

Tuesday 3PM ABE

Introduction

For this lab, we were given the primary task of reproducing a simplified LC3 processor using the FPGA board and encoded via SystemVerilog. The SLC3 16 bit processor that we were to make had to perform three functions: fetching, decoding, and executing. This was to be done through a combination of state machines, given memory, given tristate and a given Mem2IO (which served as the connection between data signals). The first week's objective was to have an incrementing PC (program counter) and having data show up on the hex displays representing the IR (instruction register), and week 2 was full implementation and demonstration of all functions.

Summary of Operation

The processor that we were to design is a simplified version of the LC-3 (Little Computer 3) that we were introduced to in ECE120 and ECE220. There were some commands that were excluded from this design such as TRAP or RET, but we also needed to add a pause state into the design. The design takes inputs from the 16 switches, the run, continue, and reset button. The outputs would then be displayed on the hex displays and the 12 smaller LED's below the hex displays.

The very first thing that the design does is the FETCH operation. In this operation, the PC (program counter) is loaded into the MAR (Memory Address Register). Then whatever is in the location in the MAR is then loaded in the MDR (Memory Data Register). The IR (instruction register) is then loaded with data from the MDR and PC increases by 1. The data inside IR is basically kept there for future use after being received from the MDR. The PC has to increment at the end of the FETCH instruction so that it doesn't get stuck in one cycle.

The next operation would be DECODE. By looking inside IR, the ISDU module that we implemented would determine which state to go into. This is done by looking at opcodes and following alongside our state diagrams.

The very last state would be our EXECUTE state. Based on what the instructions were from the DECODE operation, different parts of the processor would be selected to pass on certain bits on data. Operations are then performed on these bits of data and the output is eventually forwarded to the Mem2IO module which moves it to the correct place depending on the ISDU. Our extra implementation of the pause button however should turn on the LED lights with the IR if pressed and shouldn't continue until we press the continue button.

Modules, Inputs and Outputs

Module: tristate.sv
Inputs: Clk, tristate_output_enable, [N-1:0] Data_write
Outputs: [N-1:0] Data_read
Inout write: [N-1:0] Data

Description: This code was provided for us and serves as a more complicated MUX that would take and write data to the BUS.
Purpose: This module served as the SRAM tristate buffet. Looking at the inputs if OE = 0; data going to the SRAM is 16bit high. If OE = 1, the data coming in would be the same as data going out, which would be data from the MDR.

Module: slc3.sv
Inputs: [15:0] S, Clk, Reset, Run, Continue
Inout wire [15:0] Data
Output [11:0] LED, [6:0] HEX0-8, CE, UB, LB, OE, WE, [19:0] ADDr
Description: This was the module that connected the tristate, ISDU, Mem2IO, and the datapath together.
Purpose: Although we made an actual top level module, this basically served the same purpose. It made sure the inputs and outputs from each essential module was connected together properly.

Module: fivesext.sv
Inputs: [4:0] IN
Outputs: [15:0] OUT
Description: This served as a sign extension of 5 bits to 16 bits
Purpose: This module takes a 5 bit input and extends to the 16 bits. If the most significant big was a 1, extend it with 1s. Else, if the most significant bit was a 0, extend it with 0s.

Module: ninesext.sv
Inputs: [8:0] IN
Outputs: [15:0] OUT
Description: This served as a sign extension of 9 bits to 16 bits
Purpose: This module takes a 9 bit input and extends to the 16 bits. If the most significant big was a 1, extend it with 1s. Else, if the most significant bit was a 0, extend it with 0s.

Module: elevensext.sv
Inputs: [10:0] IN
Outputs: [15:0] OUT
Description: This served as a sign extension of 11 bits to 16 bits
Purpose: This module takes a 11 bit input and extends to the 16 bits. If the most significant big was a 1, extend it with 1s. Else, if the most significant bit was a 0, extend it with 0s.

Module: reg_16
Inputs: Clk, Load, Reset, [15:0] IN
Outputs: [15:0] OUT.

Description: A simple flipflop that checks for the load signal. If load is high, Dout becomes Din. If we reset it instead, the flipflop's Dout becomes all 0s.
Purpose: We need several 16 bit registers in our design. PC, MAR, MDR, and IR all require a register to either hold the data, change their data, or reset their data.

Module: regFile
Inputs: [15:0] bus, [2:0] dr, sr2, sr1, Clk, Load, Reset,
Outputs: [15:0] sr2out, sr1out
Description: This was basically a big 3:1 Mux that determined what got passed on from the regFile block.
Purpose: Depending on the select signals (ld, dr, sr2, sr1), data that would either be loaded into this mux or be put out in outputs sr2out or sr1out to be used by the slc3 circuit. This basically served as the memory/registers for the slc3.

Module: branchenable
Inputs: Clk, Nin, Zin, Pin, LD_BEN, [2:0] IN
Outputs: BEN
Description: This module was made up of a 1 bit register and a few combination logic elements that took inputs Nin, Zin, Pin and output a branch enable signal.
Purpose: This module controlled the branch enable signal. After internally computing using NZP and the IR, it sets BEN to either 1 or 0, which then gets used the the SLC3 to determine whether or not a branch is required.

Module: NZP
Inputs: Clk, Nin, Zin, Pin, LD_CC
Outputs: Nout, Zout, Pout
Description: This is a flipflop that triggers on the positive edge of the clock to update the inputs to the branchenable module.
Purpose: This module is in place to make sure that values of branchenable module are loaded in correctly at the right edge of the clock when LD_CC is high.

Module: ALU.sv
Inputs: [15:0] A,B, [1:0] select
Outputs: [15:0] OUT
Description: This was basically a MUX that took values of A and B and performed some operation (+,&,~,PASS) on them.
Purpose: This served as our ALU so that we could perform the add operation, the and operation, the not operation, and the just pass operation. This served whenever we wanted to add values, or adjust PC as needed.

Module: datapath.sv
Inputs: input logic Reset, Clk, ld_ir, ld_mdr, ld_mar, ld_pc, ld_regF, ld_cc, ld_ben, ld_led, [15:0] mdrin, marS, pcS, aluS, mdrS, drmS, sr1mS, sr2mS, addr1mS, mioen, [1:0] pcmS, addr2mS, aluSelect
Outputs: ben, [11:0] led, [15:0] irout, mdrout, marout, [15:0] pc
Description: This module served as the interconnections and initiation of several other modules. It is within this file we that connected all the wires between the different MUXs and registers together. It is also within this while where the logic behind loading different MUXs onto the "bus" was. This different data on the bus was then passed along to their appropriate modules
Purpose: This served as our datapath/wiring between all connections. It also controlled when the 4 main MUXs would output their data so that we were not flooding the 'bus' with data from more than one place.

Module: Hexdriver.sv
Inputs: [3:0] In0;
Outputs: [6:0] Out0;
Description: This module was given by course staff to convert data to outputs for the hex display on board.
Purpose: This was used to display the final answers onto our FPGA board.

Module: MEM2IO
Inputs: Clk, Reset, CE, UB, LB, OE, WE, [19:0] A, [15:0] Switches, Data_CPU_In, Data_Mem_In
Outputs: 15:0] Data_to_CPU, Data_to_SRAM,[3:0] HEX0, HEX1, HEX2, HEX3
Description: This code was provided by course staff. It served as basically a complicated MUX which decided where to send ata.
Purpose: This served as the connection between the switches, HEX display, CPU, and memory and it decided just where data would be passed.

Module: MUX.sv
Inputs: [15:0] s1,s2,s3, [1:0] select
Outputs: [15:0] OUT
Description: This was a 3:1 MUX that had a 2 bit select signal.
Purpose: This served as our PC_MUX, SR2 Mux, ADDR1 MUX, ADDR2 MUX, BUS, and MDR MUX. All of these MUXs had to take 16 bits of data in and then determine which set of data to pass out.

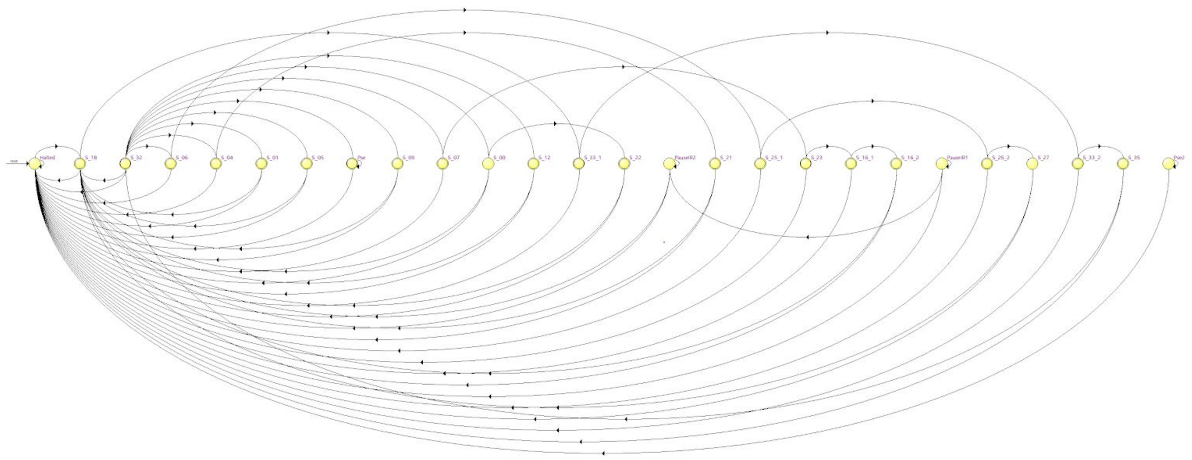
Module: MUX2
Inputs: [2:0] s1,s2

Outputs: [2:0] OUT
Description: This was a 2:1 MUX that had a 1 bit select signal.
Purpose: This MUX was used to make a DR MUX and an SR1MUX. Both of which took 3 bit length data and then determined based on select signals which to pass out.

Module: ISDU.sv
Inputs: Clk, Reset, Run, Continue, BEN, IR_5, IR_11, [3:0] Opcode
Outputs: 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_CE, Mem_UB, Mem_LB, Mem_OE, Mem_WE
Description & Purpose:

The ISDU served as the state machine that controlled the whole processor. Its main input is a [3:0] Opcode that helps it fetch the corresponding instructions from memory. From there it goes into the state that matches that instruction. From there it just follows the path until it gets right back to the start at state_18. Once here, it fetches a new instruction and begins the cycle again. The GATE outputs help to determine what exactly gets output on the data bus. This is an essential function of the ISDU as without it, you would flood the bus with two sets of data which would result in code that doesn't function properly. All the [1:0] selects named after their correspond MUXs work to provide the MUXs with the selection bits which then passes on the corresponding data that we want out. All the Mem_X signals are there to enable writing or reading from memory. All the LD_X signals decide where data that is on the bus goes.

State Diagram



Block Diagram

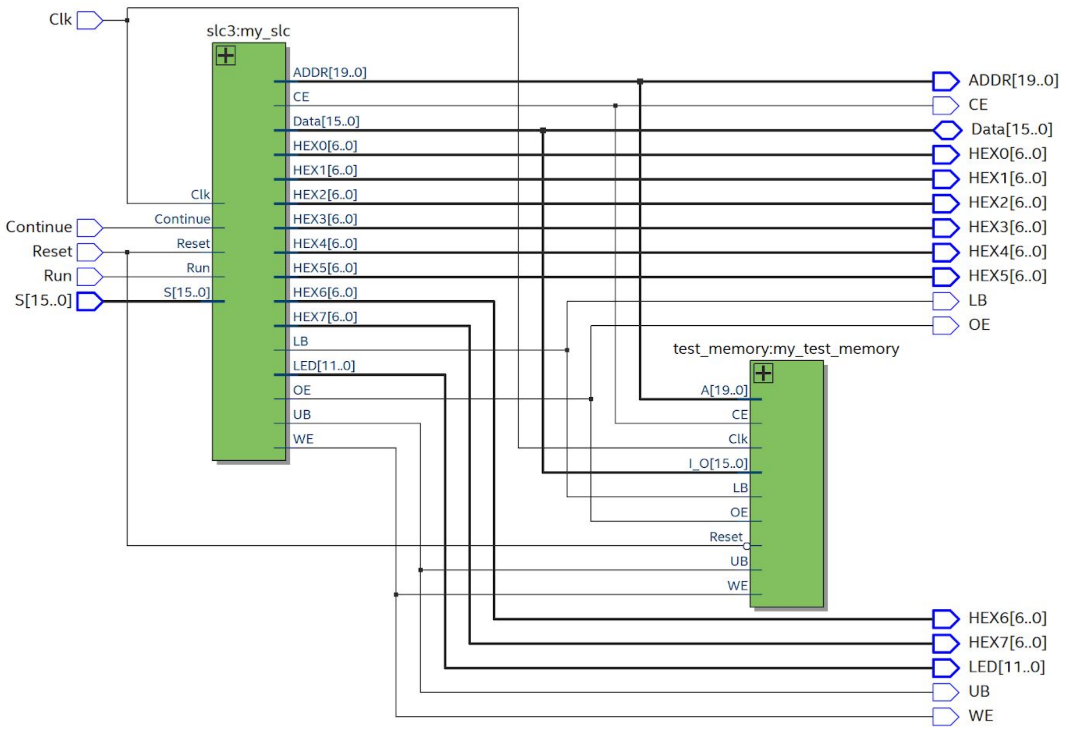


Figure 1: Top-level block diagram

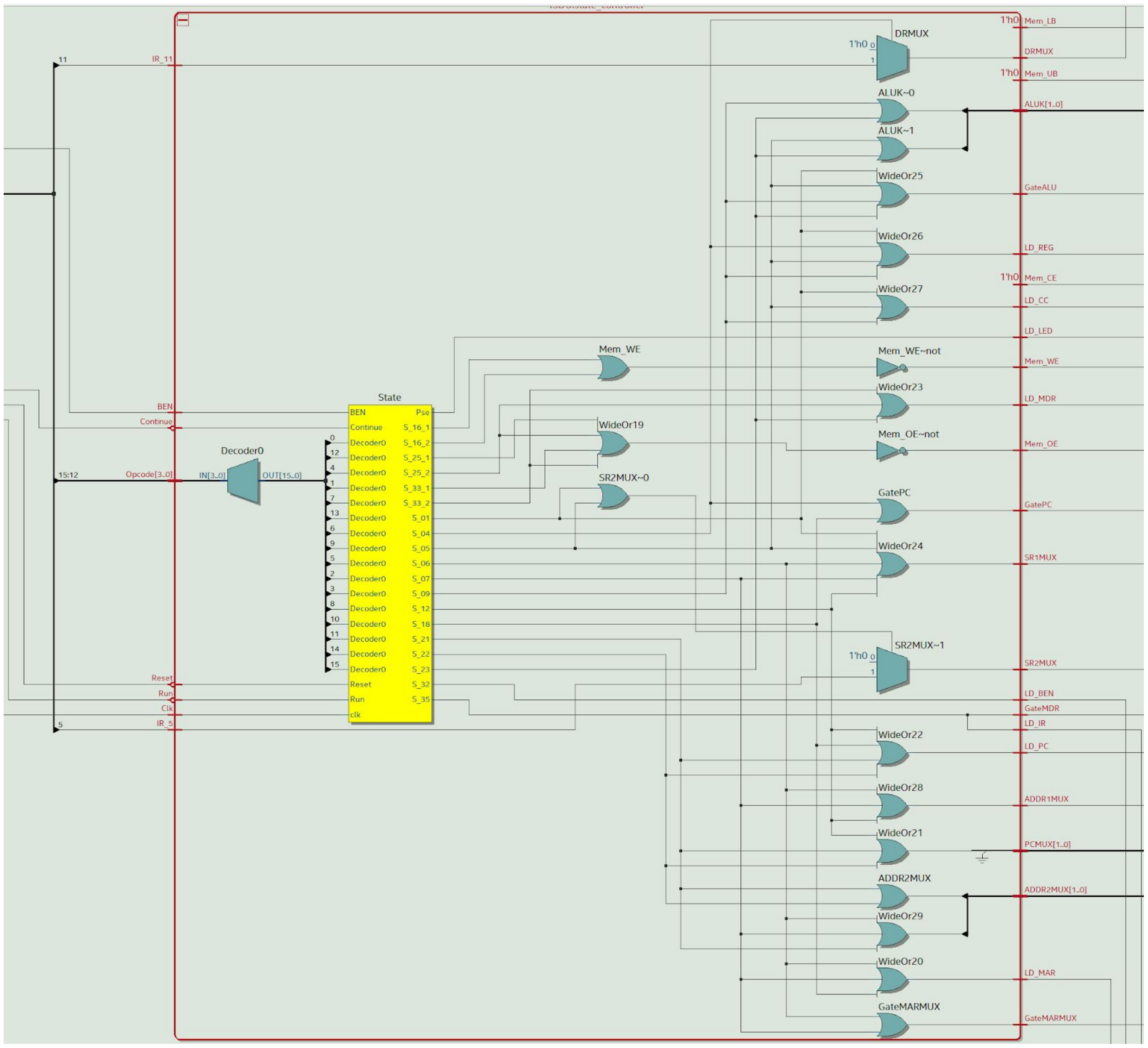


Figure 2: ISDU State Controller

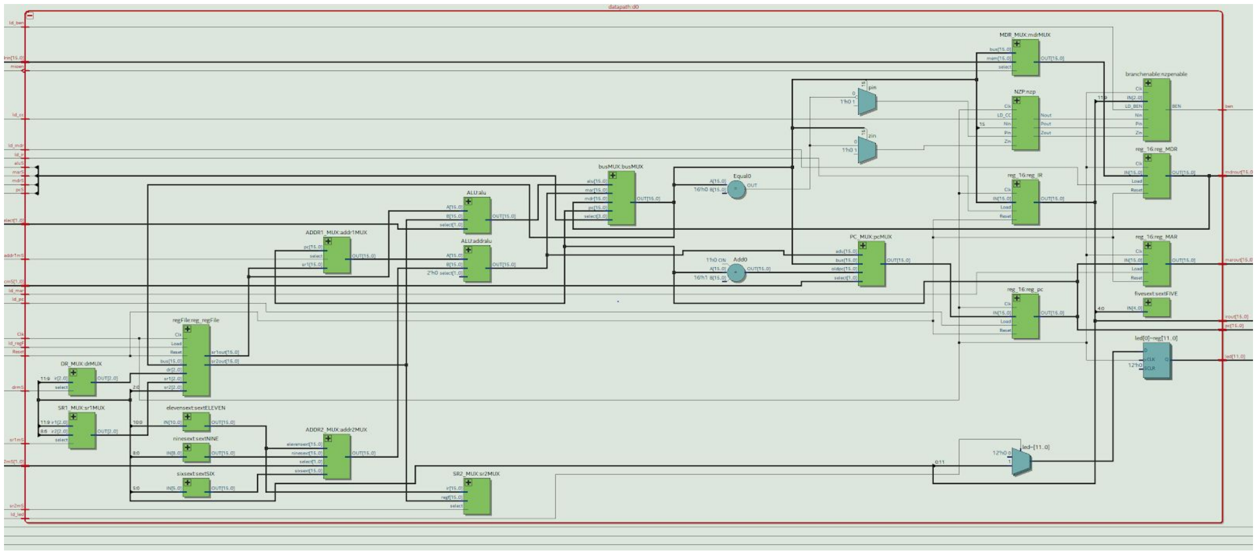


Figure 3: Datapath

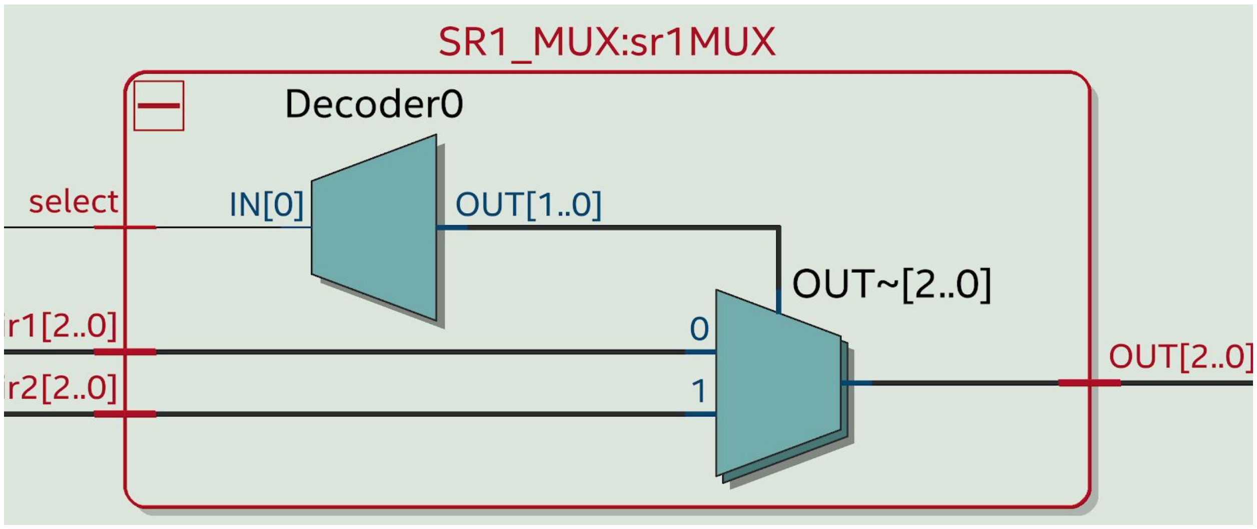


Figure 4: MUX’s (in general)

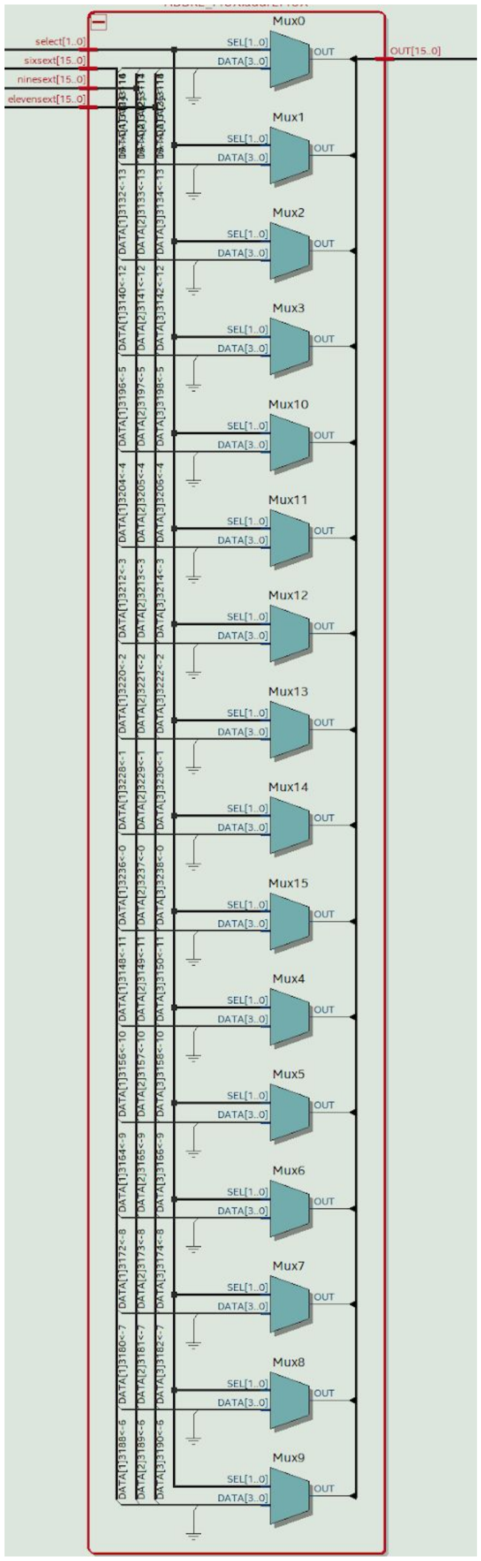


Figure 5: ADDR2 MUX, similar to BUS MUX and PC MUX

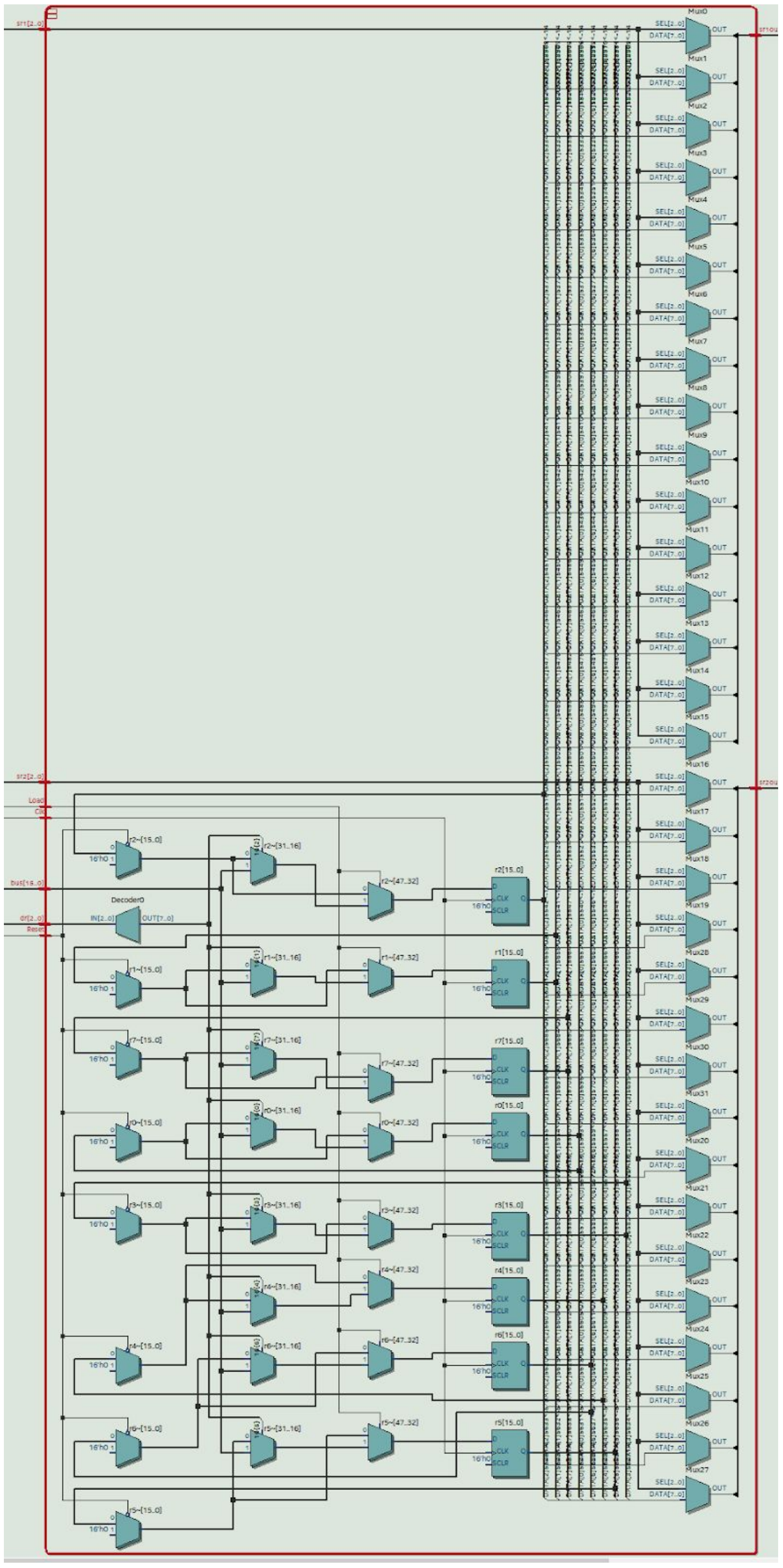


Figure 6: Register File

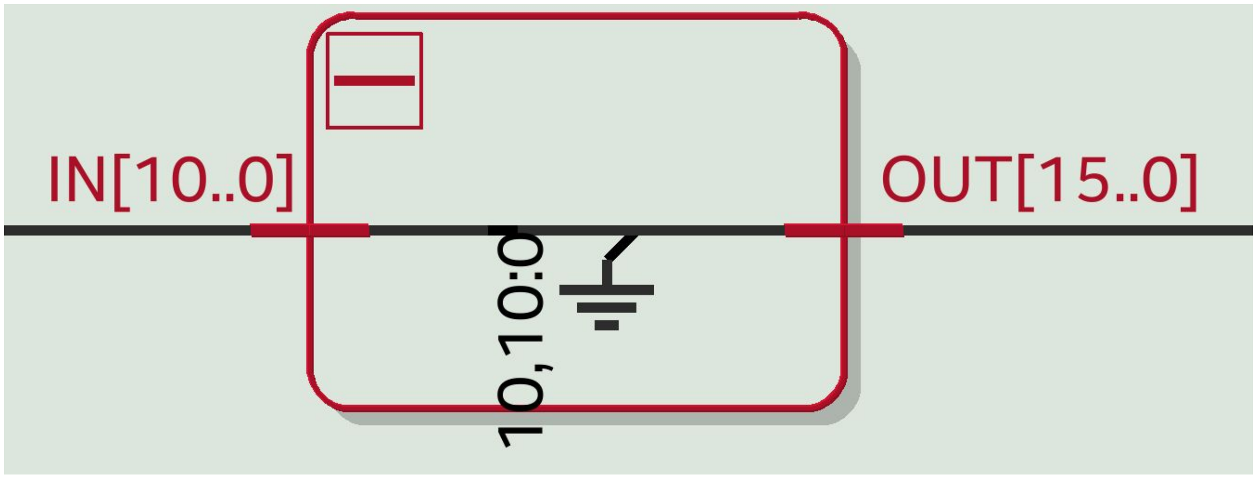


Figure 7: Sign-extend module (in general)

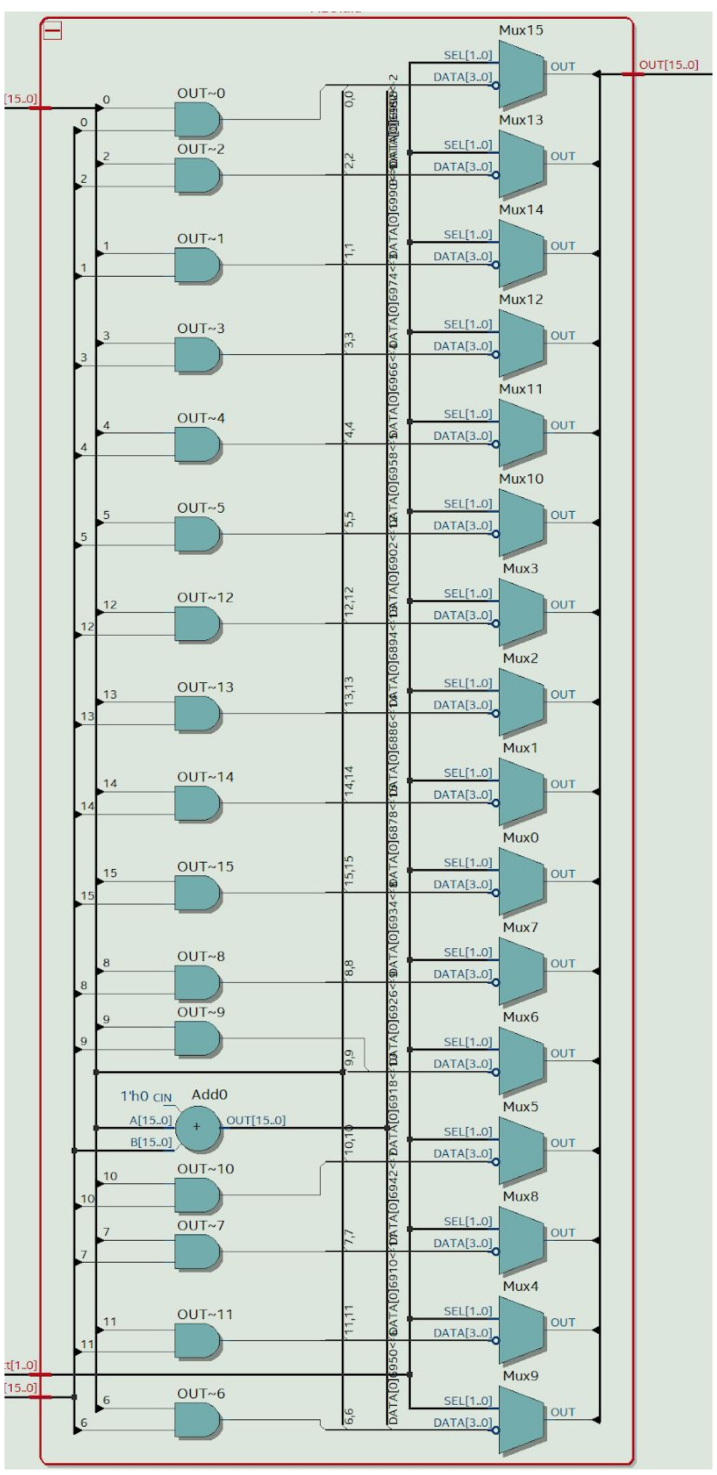


Figure 8: ALU unit

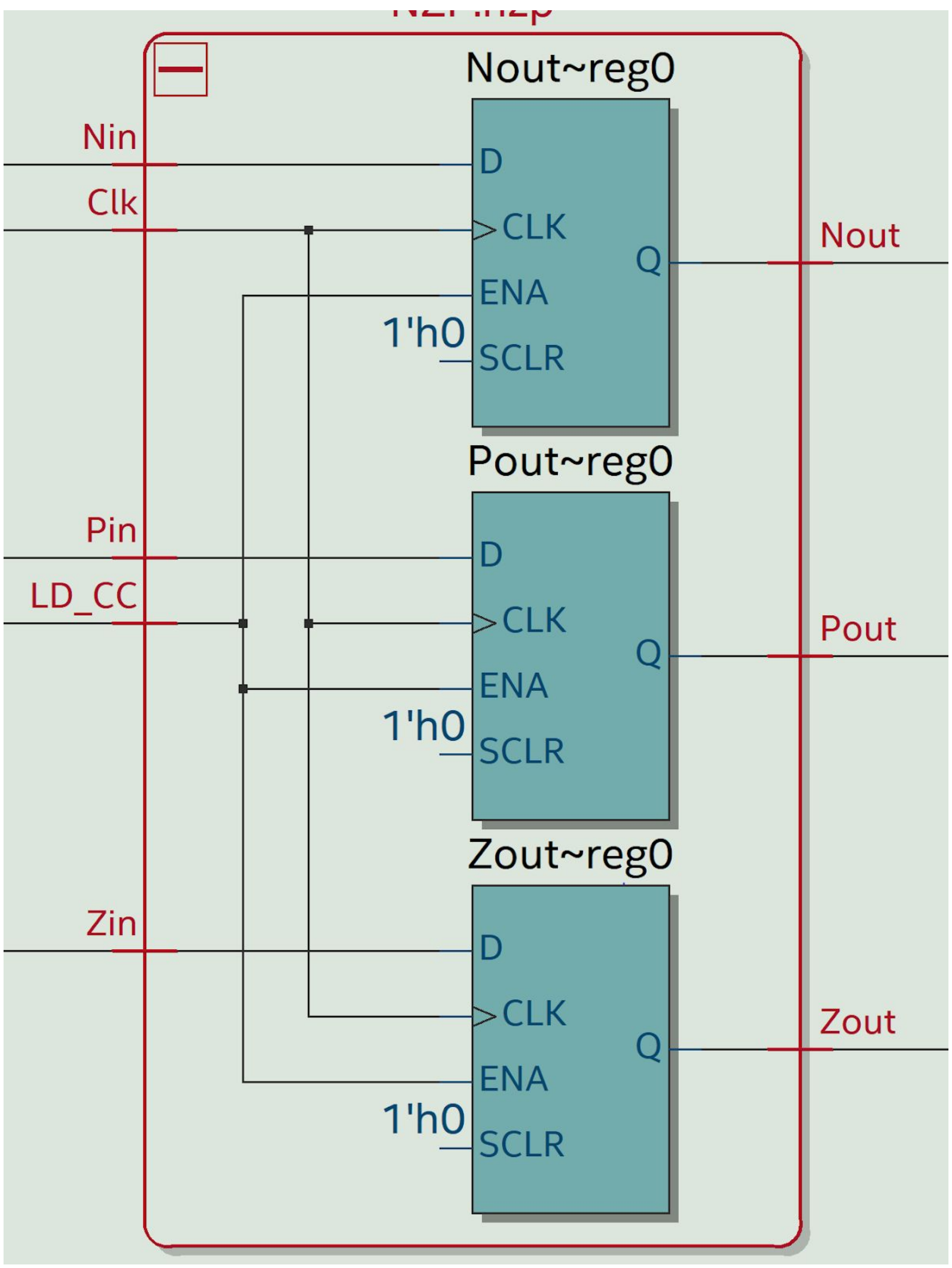


Figure 9: NZP Block

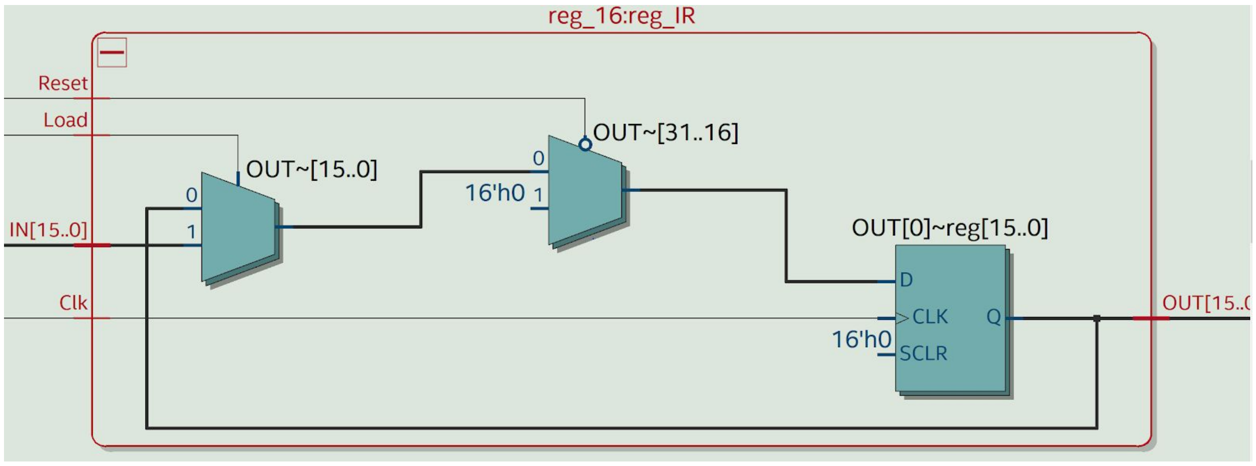


Figure 10: 16-bit register

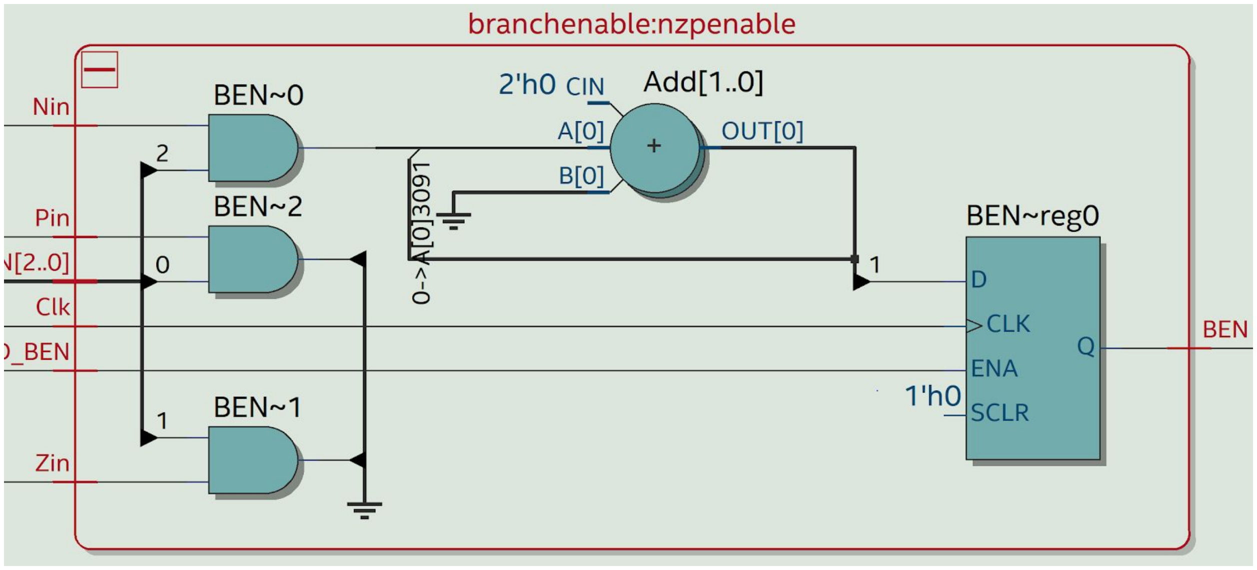
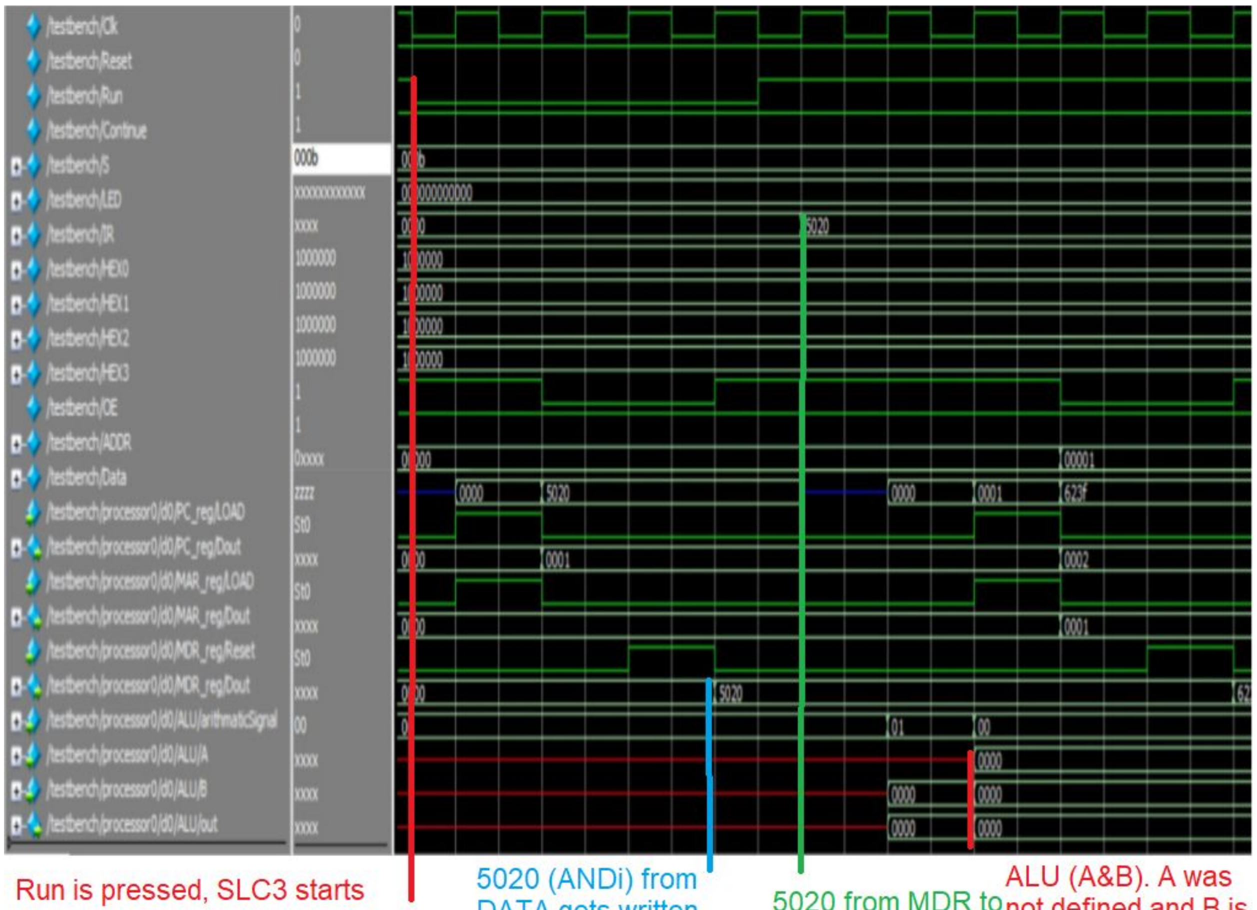


Figure 11: Branch Enable block

Simulation Waveforms

ANDi operation


/testbench/Clk
/testbench/Reset
/testbench/Run
/testbench/Continue
/testbench/S
/testbench/LED
/testbench/IR
/testbench/HEX0
/testbench/HEX1
/testbench/HEX2
/testbench/HEX3
/testbench/OE
/testbench/ADDR
/testbench/Data
/testbench/processor0/d0/PC_reg/LOAD
/testbench/processor0/d0/PC_reg/Dout
/testbench/processor0/d0/MAR_reg/LOAD
/testbench/processor0/d0/MAR_reg/Dout
/testbench/processor0/d0/MDR_reg/Reset
/testbench/processor0/d0/MDR_reg/Dout
/testbench/processor0/d0/ALU/arithmaticSignal
/testbench/processor0/d0/ALU/A
/testbench/processor0/d0/ALU/B
/testbench/processor0/d0/ALU/out
5020
0001
0002
623f

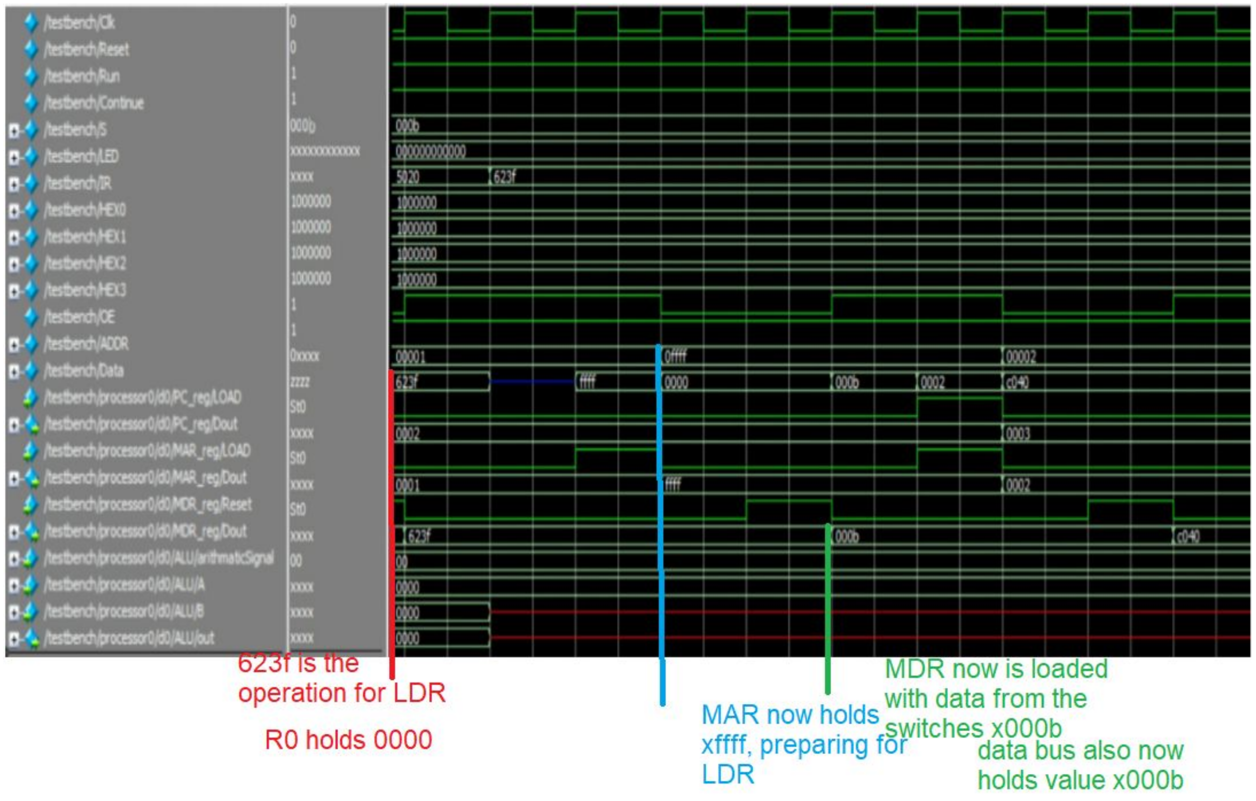
Run is pressed, SLC3 starts

5020 (ANDi) from DATA gets written into MDR

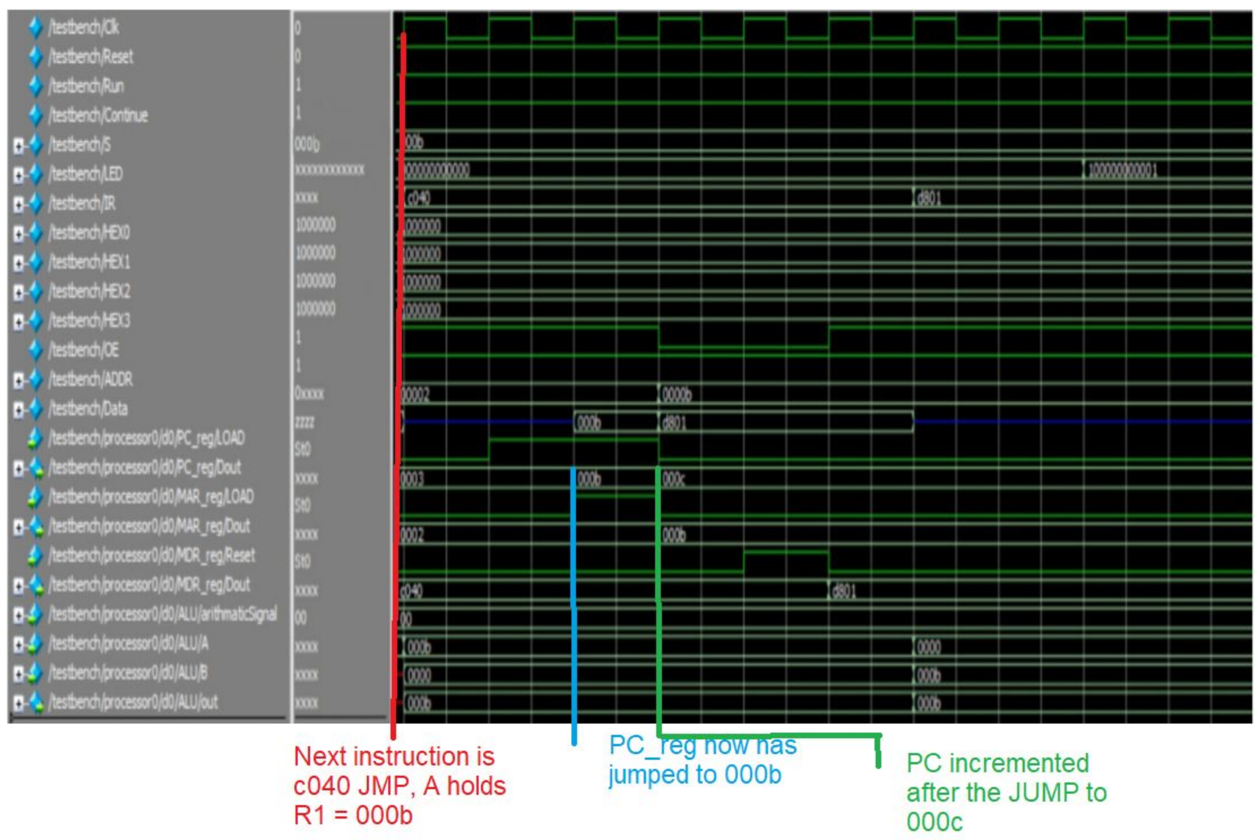
5020 from MDR to IR

ALU (A&B). A was not defined and B is 0, 0 anded with 0 gives output all 0

LDR Operation


/testbench/Clk
/testbench/Reset
/testbench/Run
/testbench/Continue
/testbench/S
/testbench/LED
/testbench/IR
/testbench/HEX0
/testbench/HEX1
/testbench/HEX2
/testbench/HEX3
/testbench/OE
/testbench/ADDR
/testbench/Data
/testbench/processor0/d0/PC_reg/LOAD
/testbench/processor0/d0/PC_reg/Dout
/testbench/processor0/d0/MAR_reg/LOAD
/testbench/processor0/d0/MAR_reg/Dout
/testbench/processor0/d0/MDR_reg/Reset
/testbench/processor0/d0/MDR_reg/Dout
/testbench/processor0/d0/ALU/arithmaticSignal
/testbench/processor0/d0/ALU/A
/testbench/processor0/d0/ALU/B
/testbench/processor0/d0/ALU/out
623f is the operation for LDR
R0 holds 0000
MAR now holds xffff, preparing for LDR
MDR now is loaded with data from the switches x000b
data bus also now holds value x000b

JMP Operation


/testbench/Clk
/testbench/Reset
/testbench/Run
/testbench/Continue
/testbench/S
/testbench/LED
/testbench/IR
/testbench/HEX0
/testbench/HEX1
/testbench/HEX2
/testbench/HEX3
/testbench/OE
/testbench/ADDR
/testbench/Data
/testbench/processor0/d0/PC_reg/LOAD
/testbench/processor0/d0/PC_reg/Dout
/testbench/processor0/d0/MAR_reg/LOAD
/testbench/processor0/d0/MAR_reg/Dout
/testbench/processor0/d0/MDR_reg/Reset
/testbench/processor0/d0/MDR_reg/Dout
/testbench/processor0/d0/ALU/arithmaticSignal
/testbench/processor0/d0/ALU/A
/testbench/processor0/d0/ALU/B
/testbench/processor0/d0/ALU/out
Next instruction is c040 JMP, A holds R1 = 000b
PC_reg now has jumped to 000b
PC incremented after the JUMP to 000c

Design Statistics

LUT	627
DSP	0
Memory (BRAM)	0
Flip-Flop	280
Frequency (MHz)	53.4
Static Power (mW)	98.98
Dynamic Power (mw)	9.10
Total Power (mW)	180.41

Post-lab

The MEM2IO module served as a pretty advanced MUX. The code was provided to us by the course staff and connected all the components (switches, hex display, CPU, memory0. From the I/O devices, if it gets an address while READ ENABLE was high, it would then read the input from the switches. If it gets an address during WRITE ENABLE high, it would write the data into HEX drivers rather than putting it into memory.

The JMP command is a way for the code to go to another location to start new instructions. The JMP command can be called without taking into consideration the previous operations. The BR command only works if BEN conditions are met, and this depends on a value being negative, positive, or zero. JMP loads the PC with some BaseR, but with BR, the PC jumps with a specified offset.

Conclusion

In the end, we could not get our design fully functioning by the time of demo. What was being displayed on our HEX drivers was a backwards six. We were told by the TA that it was because we were trying to simultaneously output two values at once. We ran through our code and see when that would be possible, but we could not find a place in the state diagram where data was being output twice in one instance. We eventually settled on the reasoning of this bug to be an issue with programming the FPGA with the controller given. The control panel given to us was in it's beta development stage and we were warned of its potential glitches. Even when reverting all our week 2 changes in an attempt to reproduce our successful week 1 results, the board did not work. My partner and I both agree that if the course staff knew that an SD card could've prevented the issue, then an SD card should have been provided in the toolbox given to students in the course. An extra \$10-\$15 dollar SD card really wouldn't be that much more to add to a kit containing a several hundred dollar FPGA board. The lab itself had pretty straightforward information given to it; we were provided a state diagram basically and a block diagram and we just had to implement it. So, despite unfortunately not being able to complete the objectives, we believe we were at least partially successful in completing the code aspect of the lab.