ECE 385

Spring 22

Experiment #5

Simple Computer SLC-3.2 in SystemVerilog

Siddarth Iyer, Geitanksha Tandon

TA: Ruihao Yao (RY)

1. Introduction

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

In this experiment, we designed a simplified 16-bit processor (with 16-bits of instruction and 16-bit data), which is a subset of the LC-3 ISA. There are three main components of the processor: memory, CPU, I/O interface. The processor performs 3 operations: fetch, decode, execute. It first fetches the instruction to be executed from memory, decodes the instruction to determine via the first 4 bits of its 16-bit instruction (Opcode) which instruction needs to be executed, and finally executes the instruction with the help of the datapath. The datapath is the implementation of all the functional modules required by the Computer.

In the first week, we had to implement the fetch phase of the SLC-3 which would go to memory and attempt to access an instruction stored in that memory. This meant familiarizing ourselves with the MEM2IO module which helped us create the RISC CPU version where memory was handled indirectly. We had to implement the sections of our datapath that would help execute the fetch phase.

In the second week, we were to create the functional version of our entire datapath as well as creating all states required for the execution of the SLC-3. This meant we finalized the SLC-3 file implementation of the datapath, as well as created and executed the several states for the ISDU file, with the transitions, inputs, and outputs, as well as how each of these states interacted with the datapath content.

---

2. Written Description and Diagrams of SLC-3

a. Summary of Operation:
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 is a simplified version of the LC3 and operates on the same basis. The CPU consists of a Program Counter(PC), Instruction Register(IR), Memory Address Register(MAR), Memory Data Register(MDR), Instruction Sequencer/Decoder Unit(ISDU), Status Register(NZP), 8x16 register file, and ALU. The PC points to the memory address of the current instruction to be executed. At the start of each cycle, the instruction is fetched and stored in the Instruction Register. The instruction to be executed is determined by an opcode specified by bit 15-12 of IR. The remaining 12 bits provide additional information required to execute the function. The system works with a finite state machine that fetches, decodes, executes, and then fetches again. Listed below are the instructions that can be executed by the SLC3

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

The meaning of these operations is given:

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.

The IR specifies the opcode, with other bits of the IR also used as control signals for the data path
In each particular state, the ISDU (Instruction Sequencer/Decoder Unit) specifies all the control signals to be generated, and provides the necessary inputs to all components of the datapath, including the ALU. The clock cycles usually take a specific time per instruction, with the ones that access memory requiring 3 separate states to successfully access that memory, and write to MDR.

FETCH: The PC address is stored in the MAR (Memory Address Register) {MAR<-PC}. Then, the MDR stores the value at the MAR {MDR <- M(MAR)}. Finally, the IR (Instruction Register) stores the value of MDR {IR<-MDR} so it can be executed, and the PC is incremented by 1 {PC<-PC+1}.

DECODE: The BEN checks whether that is NZP and compares it to the condition code, to check whether a jump is required to take place, and the Opcode is decoded to finalize which state is to be executed. The ISDU receives the information from the IR.

EXECUTE: The correct state is executed depending on the signals given by the ISDU and the state as described from the IR. Then, the correct outputs are given to either the DR or the memory.

Fetch, Load, and Store Operations:

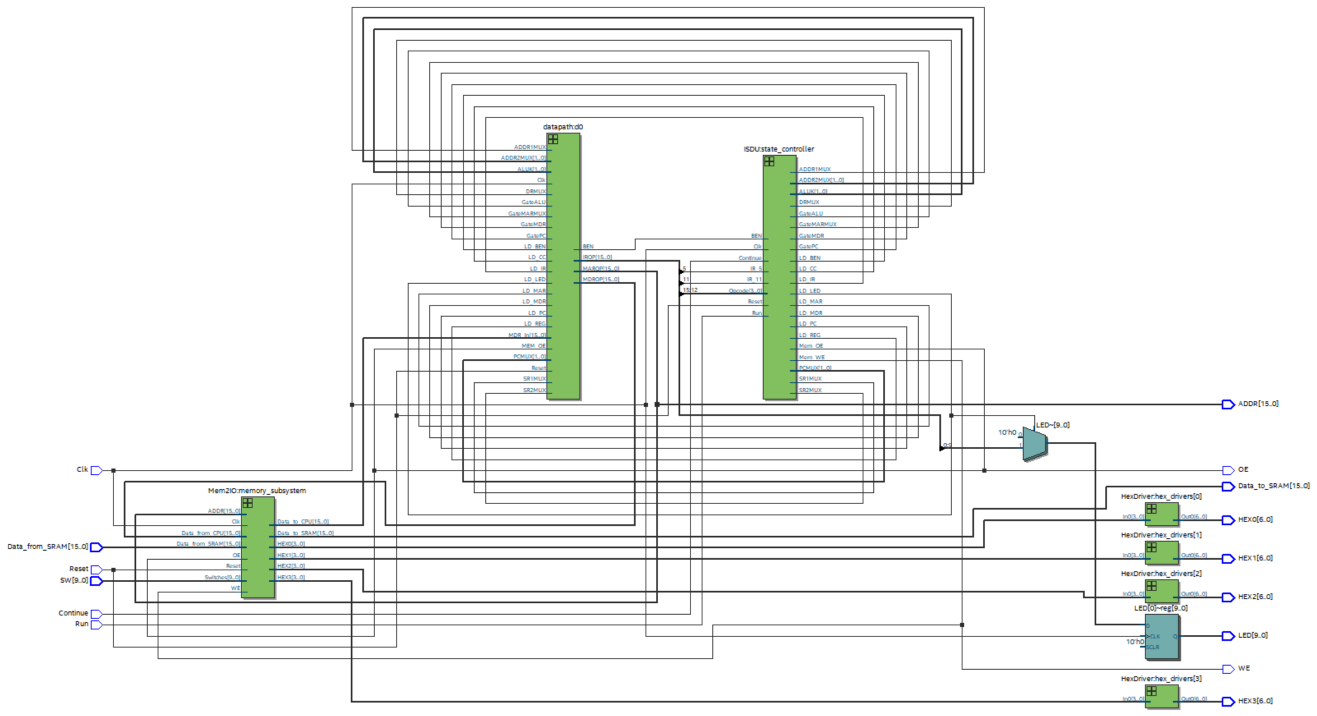
For Fetch, Load (LDR), and Store (STR) operations the ISDU appropriately sets signals for each state of the fetch/load/store sequence by reading writing to and from xFFFF. Also, since the RAM we use does not have an R signal indicating that a read/write operation is ready, so we wait for 3 clock cycles in order to ensure we extract the correct value from memory for any states reading from or writing to RAM.

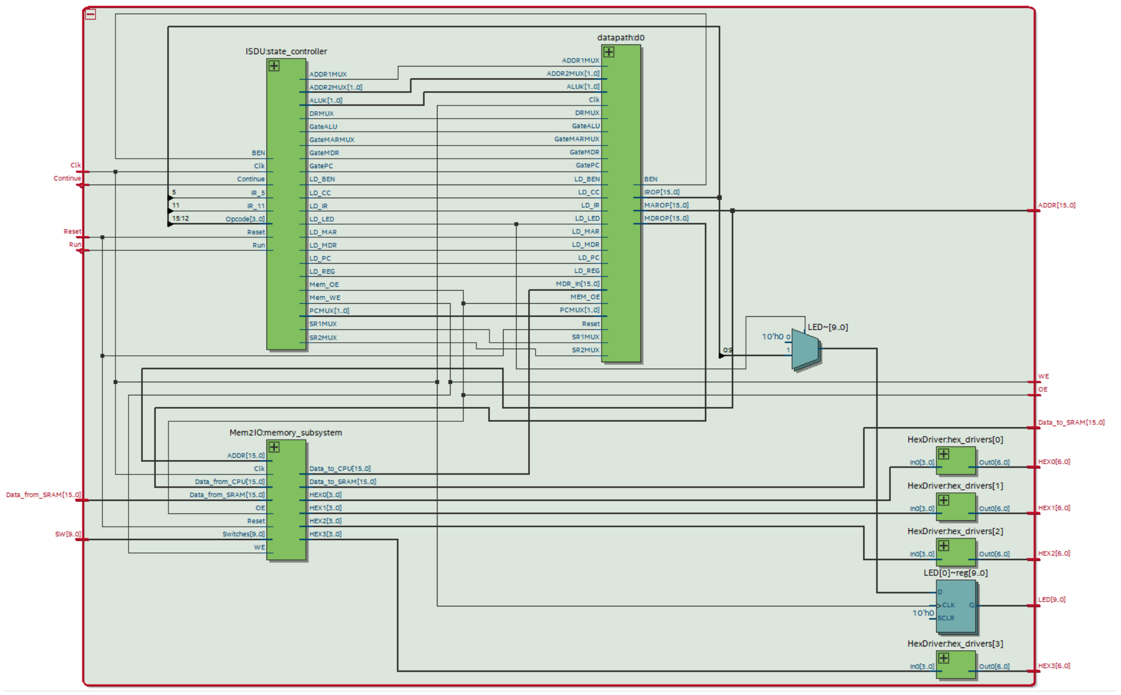
FETCH:
state1: MAR <- PC
state2: MDR <- M(MAR); -- assert Read Command on the RAM
state3: IR <- MDR;
PC <- PC + 1; -- "+1" inserts an incrementer/counter instead of an adder. Go to the next state.
LOAD:
state1: MAR <- (BaseR + SEXT(offset6)) from ALU
state2: MDR <- M(MAR); -- assert Read Command on the RAM
state3: R(DR) <- MDR;
STORE:

state1: MAR <- (BaseR + SEXT(offset6)) from ALU;
MDR <- R(SR)
state2: M(MAR) <- MDR; -- assert Write Command on the RAM.

---

c. Block Diagram of slc3.sv: This diagram should represent the placement of all your modules in the slc3.sv. Please only include the slc3.sv diagram and not the RTL view of every module (this can go into the individual module descriptions).





e. Written Description of all .sv modules

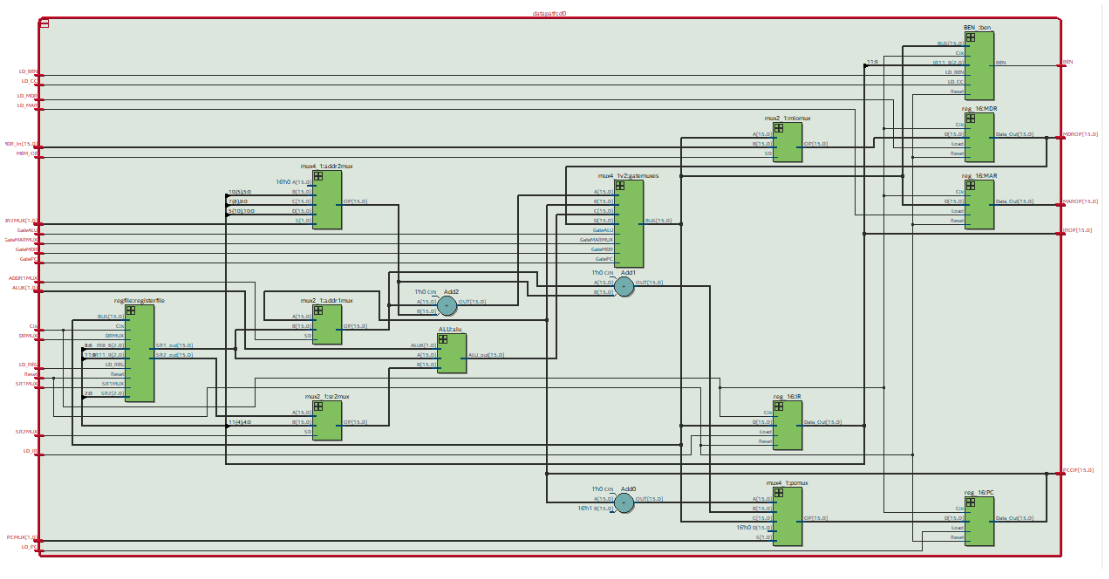
Module Descriptions:

Module: datapath.sv

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

Outputs: BEN, [15:0] IROP, PCOP, MDROP, MAROP.

Description: This module instantiates various 16-bit registers, register file, ALU, multiple MUXes, and BEN to construct the datapath. Four reg_16 16-bit registers are instantiated: IR, PC, MAR, MDR with the contents of the register corresponding to the 4 output 16-bit output signals. The load signal is determined by the input control signals decided by the ISDU. The registerfile is also instantiated and the necessary signals are connected. Multiple MUXes are instantiated, with the select bits coming from the input control signals. For example, the ADDR1MUX has inputs PC and SR1, select bit as ADDR1MUX(input set by ISDU), and outputs ADDR1_out which then serves as part of input for PCMUX. The ALU is instantiated by setting the A and B input as SR1 and the output of SR2MUX, and sending the ALUK signal to determine the operation to be executed. BEN is also instantiated, which is used to check branch conditions and accordingly branch.



Purpose: This module is used to describe the Datapath consisting of the various MUXes, Register file, ALU, registers IR, PC, MAR, MDR, and BEN. It takes in various control signals, and outputs the IR, MDR, PC and MAR.

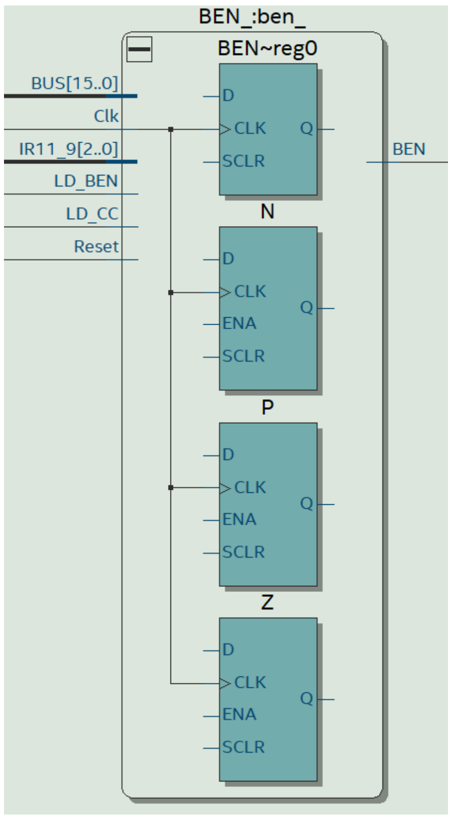
Module: BEN.sv

Inputs: Clk, Reset, LD_BEN, LD_CC, [2:0] IR11_9, [15:0] BUS

Outputs: BEN

Description: This module first checks whether the data on the bus is negative, positive, or zero by checking the MSB and contents of the BUS and accordingly sets the nzp. When LD_CC is high, it loads this nzp value into a 3-bit NZP register. When LD_BEN is high, the NZP register is compared to the nzp of the instruction from IR[11:9] and if the condition is satisfied, it sets BEN to 1, indicating a branch should occur.

Purpose: This module is used to determine whether branch conditions are satisfied and accordingly set the BEN signal indicating whether to branch or not.



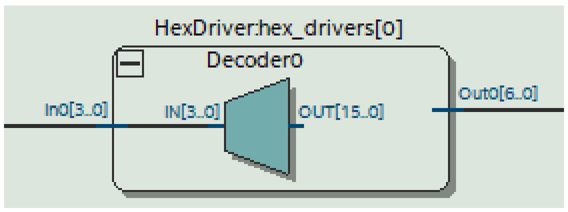
Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This makes use of a unique case switch to set the 7 bits of Out0 to the corresponding 4 bit In0, to display the 4 bit hex number on the 7-segment LED.

Purpose: This module maps a hex number to its corresponding value so it can be displayed on a 7-segment LED.

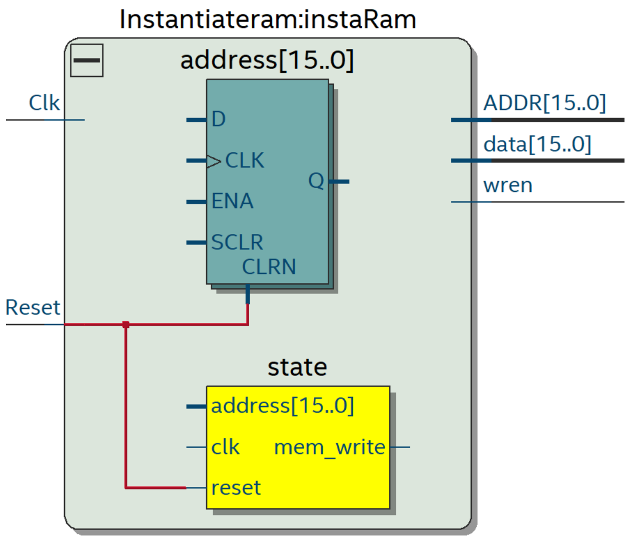


Module: Instantiateram.sv

Inputs: Reset, Clk

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

Purpose: Instantiates on-chip memory with the instructions to be executed, and contains the data/instructions at each memory address.



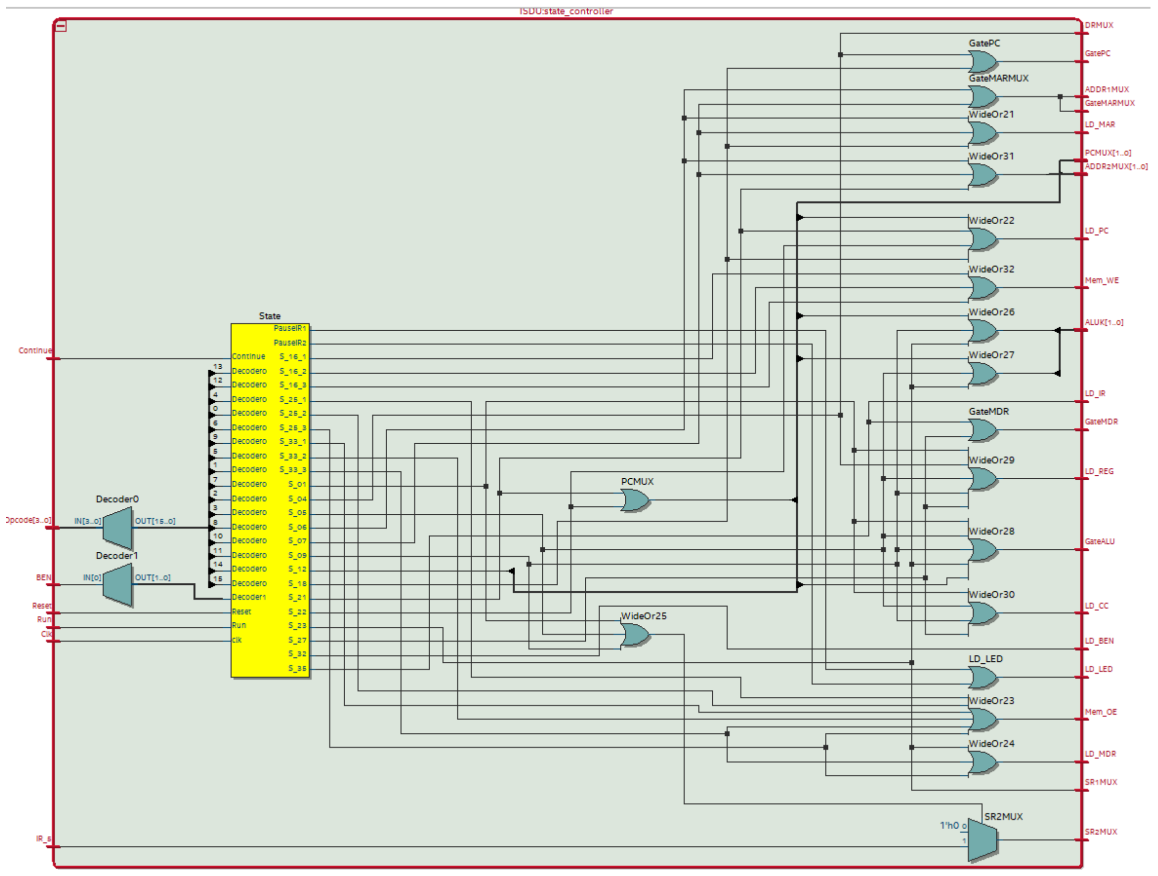
Module: ISDU.sv

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, PCMUX, Mem_OE, Mem_WE [1:0] ADDR2MUX, ALUK

Description: This module describes the ISDU or instruction state decoder unit. First, the 27 states are defined. Next, the next states are set depending on the current state. Further, the default control signals and the control signals for each state are defined. The decode state(state 32) uses the Opcode(IR[15:12]) to determine which instruction to be executed by using a case with Opcode as a parameter, and accordingly sets the next state based on the current opcode.

Purpose: This module describes the FSM of the SLC3 including all the states, the transitions, and the required control signals for each state.



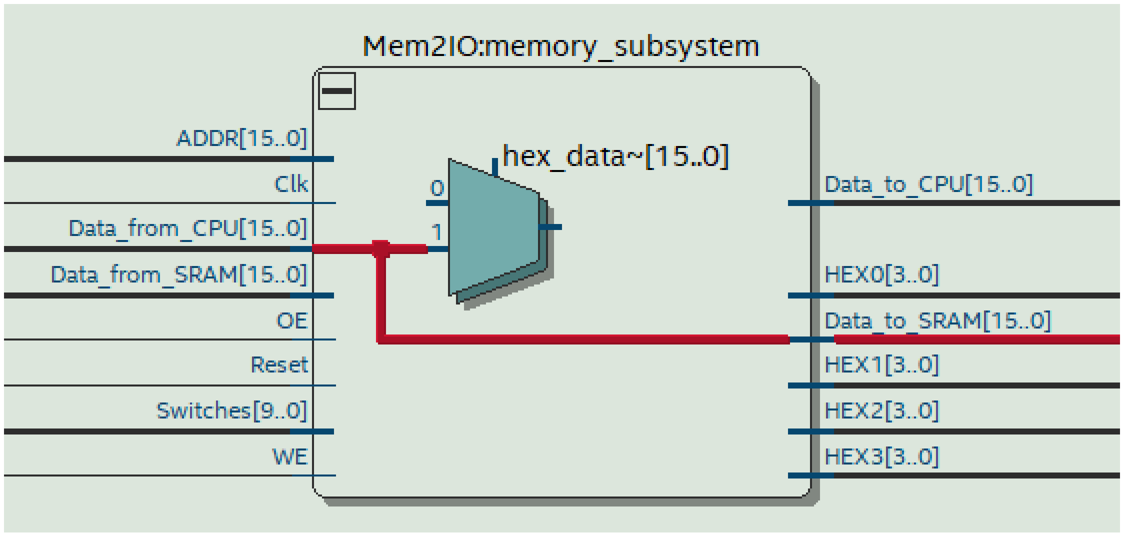
Module: Mem2IO.sv

Inputs: Clk, Reset, OE, WE, [15:0] ADDR, [9:0] Switches, [15:0] 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 used to read data from switches or write data to LEDs. When WE is low and OE is high, if the memory address is 0xFFFF the data sent to the CPU is the value on the switches, else the data to CPU comes from the SRAM. Similarly, if WE is active and the address is 0xFFFF, the data to be displayed on the hex display is set as the data from the CPU.

Purpose: This module is used to describe the MEM2IO which serves as an interface to manage the I/O devices(switches and LEDs).



Module: Mux4_1.sv

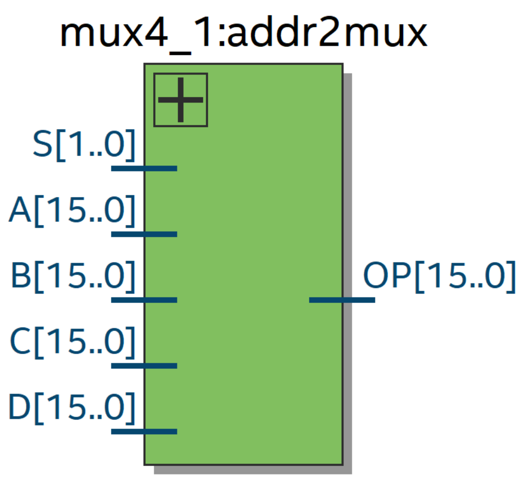
Inputs: A, B, C, D, [1:0] S

Outputs: OP

*Length is a parameter defining the size of ABCD, OP
*Sizes of A, B, C, D, OP are parametrized and set when instantiated as [length-1:0]

Description: This module uses a unique case that sets to output to A, B, C, D depending on the value S.

Purpose: This module creates a 4-to-1 MUX that outputs 1 of 4 inputs A, B, C, D, depending on the 2-bit select bit S



<u>Module: Mux2_1.sv</u>

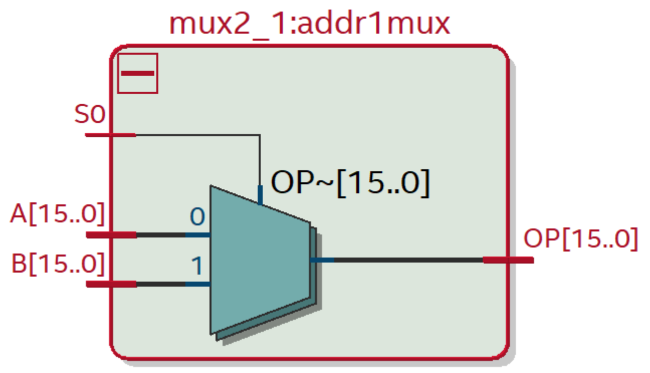
Inputs: A, B, S0

Outputs: OP

*Length is a parameter defining the size of A, B, OP
*Sizes of A, B, OP are parametrized and set when instantiated as [length-1:0]

Description: This module uses a unique case that sets to output to A or B depending on the value S0.

Purpose: This module creates a 2-to-1 MUX that outputs 1 of 2 inputs A or B depending on the 1-bit select bit S0.



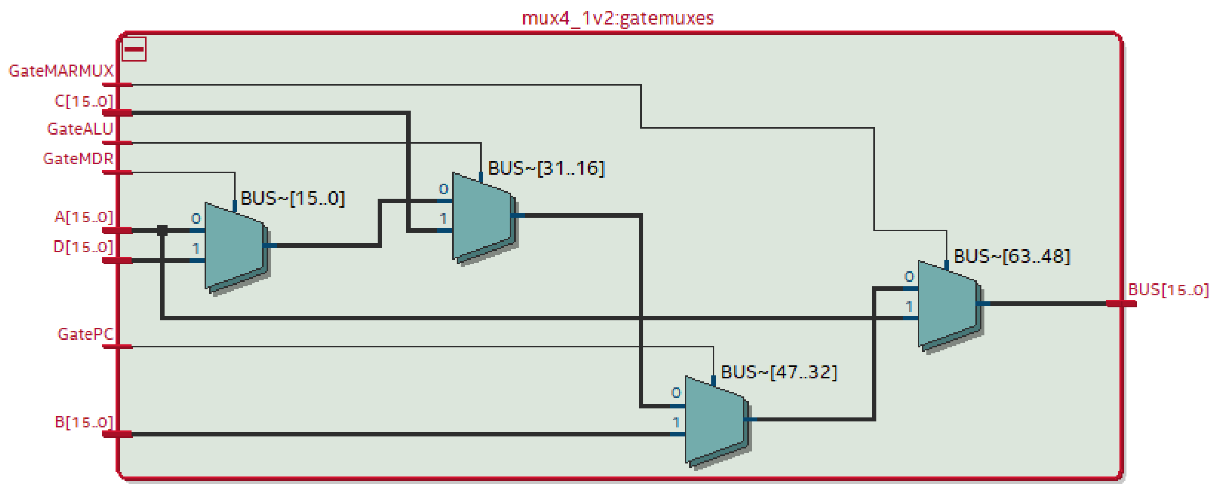
Module: mux4_1v2

Inputs: GateMDR, GateALU, GatePC, GateMARMUX, [15:0] A, B, C, D

Outputs: [15:0] BUS

Description: This module decides which value to output onto the bus. As a default, the bus is set to ADDR1_out + ADDR2_out. If GateMARMUX is high, BUS is set to ADDR1_out + ADDR2_out, if GatePC is high, BUS is set to PC_out, if GateALU is high, BUS is set to ALU_out, and if GateMDR is set to high, BUS is set to MDR_out.

Purpose: Since the FPGA does not have tristate buffers, this module is used to create a mux that determines which value to load on the BUS depending on the values of GateMARMUX, GatePC, GateALU, and GateMDR.



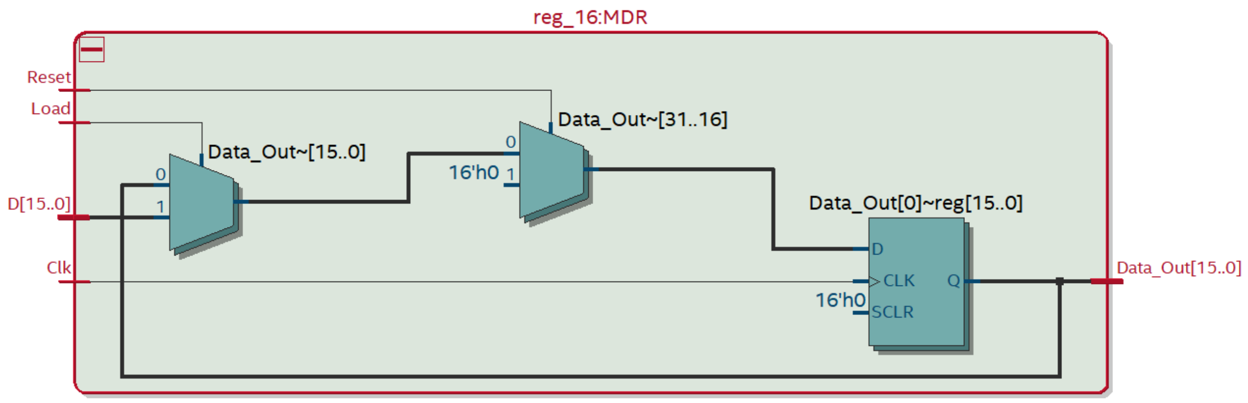
Module: reg_16.sv

Inputs: Clk, Reset, Load, [15:0] D

Outputs: [15:0] Data_Out

Description: This is a positive-edged 16-bit parallel load register. When Reset is high, the register asynchronously resets and sets Data_Out to 0. When Load is high, D loads into Data_Out.

Purpose: This module is used to create a 16 bit register to store various values such as IR, PC, MDR, MAR, as well as to create the register file.



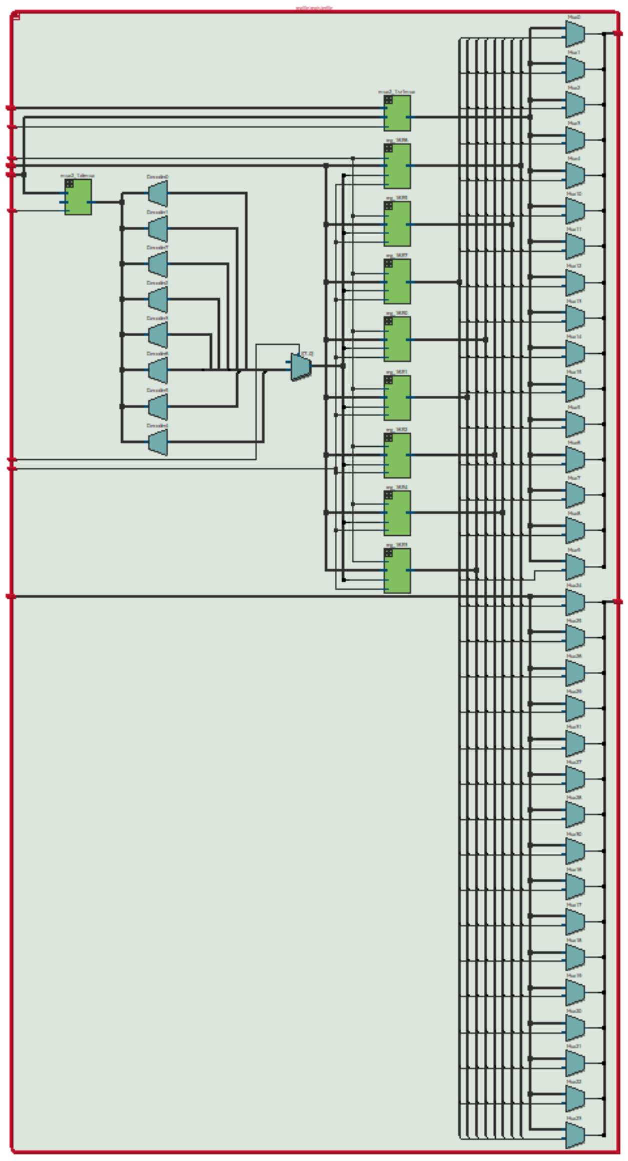
Module: regfile.sv

Inputs: Clk, Reset, LD_REG, DRMUX, SR1MUX, [2:0] IR11_9, IR8_6, SR2, [15:0] BUS,

Outputs: [15:0] SR2_out, SR1_out

Description: This module first instantiates eight 16-bit registers R0-R7, with corresponding load and dataout signals. The drmux is a 2-to-1 mux, that determines the Destination Register using the DRMUX signal, and outputs either IR[11:9] or R7 as the Destination. The load signals are determined by the output of the drmux, with the Load signal corresponding to the DR set to high. The outputs of the register file, SR1 and SR2, are determined by the output of the SR1MUX and the control signal SR2. The sr1mux is 2-1 mux with outputs either IR[8:6] or IR[11:9] depending on the control signal SR1MUX.

Purpose: This module is used to create the register file of the SLC3.



Module: alu.sv

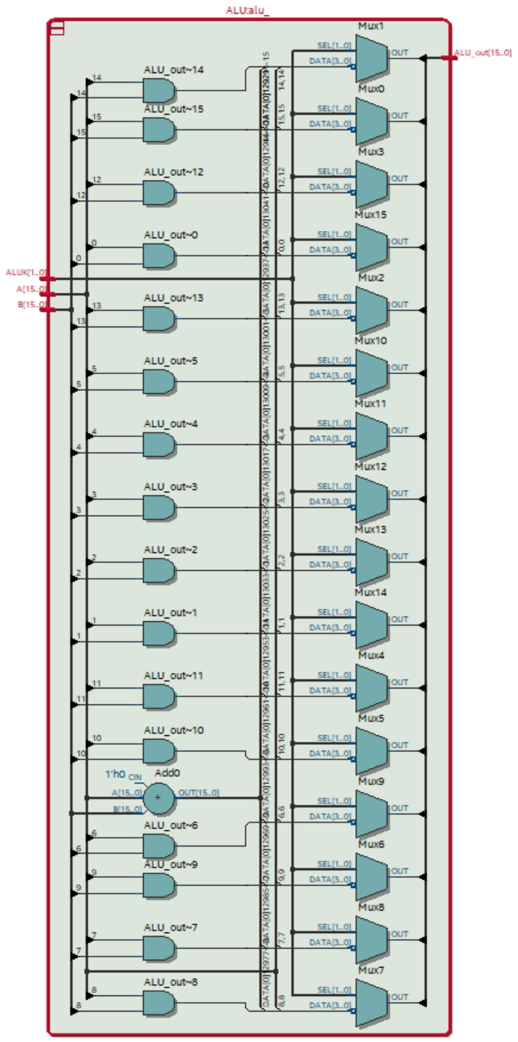
Inputs: [1:0] ALUK, [15:0] A, B

Outputs: [15:0] ALU_out

Description: This module uses the control signal ALUK to determine the operation to be executed by the ALU. When ALUK is 00, the output ALU_out is set to A + B, when

ALUK is 01, the output is set to A & B, when ALUK is 10, the output is set to ~A, and when ALUK is 11, the output is set to A

Purpose: This module is used to create the ALU of the SLC3.



<u>Module: slc3.sv</u>

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

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

Description: This module first instantiates the datapath and passes the control signals from the ISDU as inputs, while also passing the BEN, MAR, MDR, IR and PC as outputs. The ISDU is instantiated, with the IR and BEN passed as inputs, and the corresponding control signals set as outputs. Finally, MEM2IO is instantiated and the data to and from the CPU and SRAM is set. The LEDs are also set to display IR[9:0] if LD_LED is high.

Purpose: Intermediate level module that instantiates the ISDU, datapath, and MEM2IO

Module: slc3_testtop.sv

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

Outputs:OE, WE, [9:0] LED, [6:0] HEX0, HEX1, HEX2, HEX3

Description: This module is the top-level entity that instantiates slc3, and test_memory. The buttons Run and Continue which are initially active low are synched and made to active high, and the final Reset, run and continue buttons are set.

Purpose: This file is the top-level entity when simulating using ModelSim.

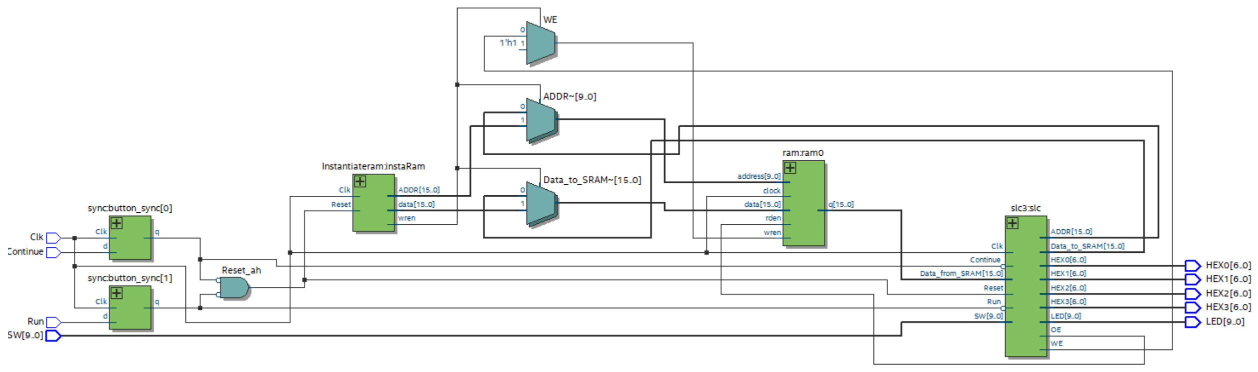
Module: slc3_sramtop.sv

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

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

Description: This module is the top-level entity that instantiates slc3, ram and Instantiateram. The buttons Run and Continue which are initially active low are synched and made to active high, and the final Reset, run and continue buttons are set.

Purpose: This file is the top-level entity when running the SLC3 on the FPGA board.



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

The Instruction Sequence Decoder Unit serves as the control unit that describes the SLC3 FSM. First the 27 states of the FSM are defined: Halted, PauseIR1, PauseIR2, S_18, S_33_1, S_33_2, S_33_3, S_35, S_32, S_01, S_05, S_09, S_06, S_25_1, S_25_2, S_25_3, S_27, S_07, S_23, S_16_1, S_16_2, S_16_3, S_04, S_21, S_12, S_00, S_22.

The ISDU has various control signal and outputs set depending on the current state: 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.

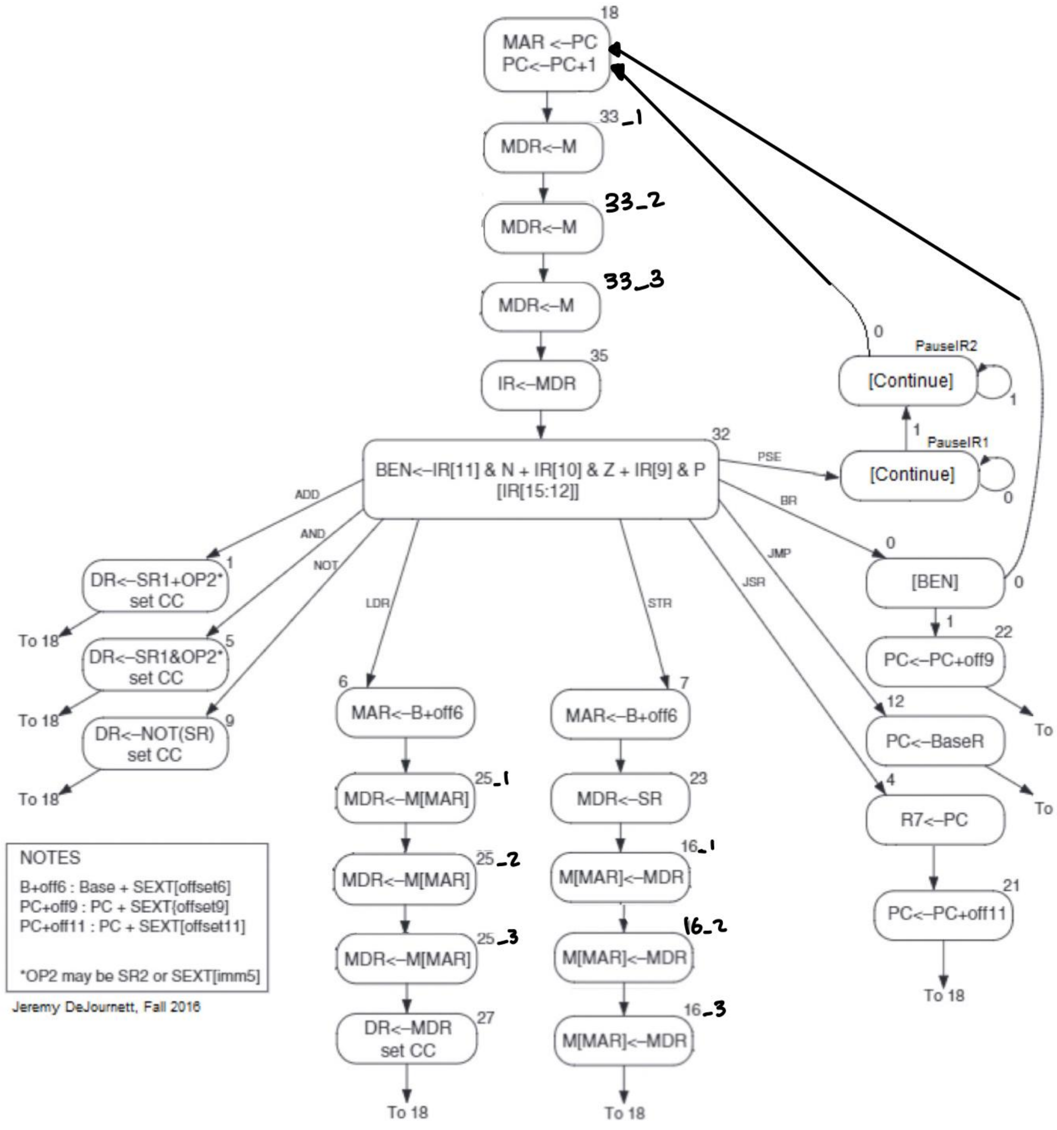
These control signals are all set to 0 by default. Each state of the FSM has corresponding control signals that are required to be set to high during the execution of that state. Using a Case with the parameter as the state, for each state, the required control signals are set to high.

For each state, the next state is typically known, and using a unique case with the state as the parameter, for each current state, the next state is set. For the decode state of State 32, the next state which decides which instruction to be executed is decided by the OPCODE. Again, a case is used with parameter opcode(Based on IR[15:12] of instruction being executed), and the next state is set depending on the opcode. When reset is pressed the FSM transitions to the halted state, while otherwise, on each rising clock cycle, the state becomes the next state.

Certain states such as State 33, 25, and 16 access memory. Since memory access requires 2 clock cycles, and an additional cycle to copy into MDR, these states are split into 3 states. For example, state 25 is split into S_25_1, S_25_2, S_25_3. It is known that memory access does not take beyond 3 clock cycles, and therefore all three states set the same control signals required for the memory access, and the memory is accessed within these 3 states.

---

g. State Diagram of ISDU

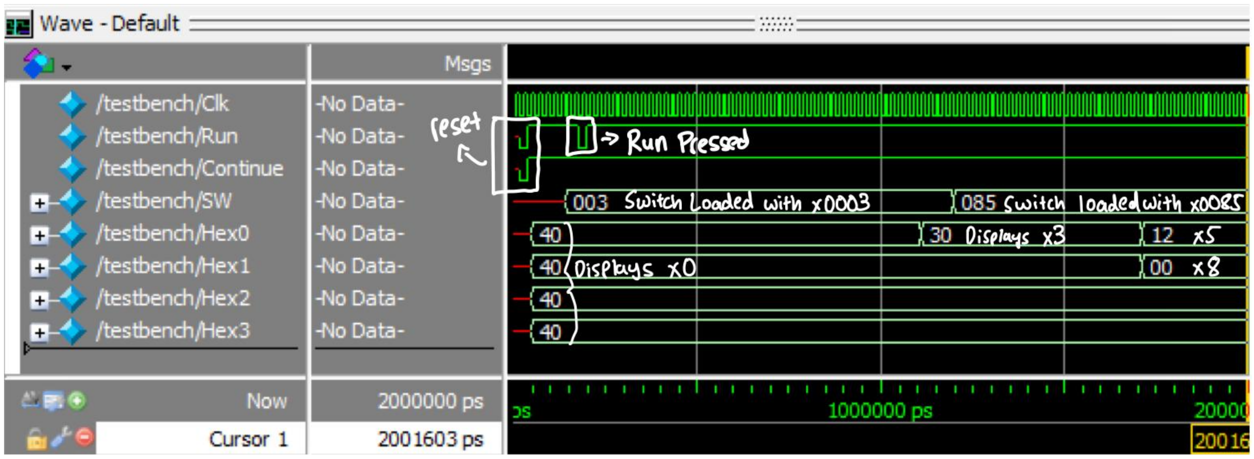


---

3. Simulations of SLC-3 Instructions

a. Simulate the completion of all 6 test programs, I/O Test 1, I/O Test 2, Self-Modifying Code, XOR, Multiplier, and Sort.

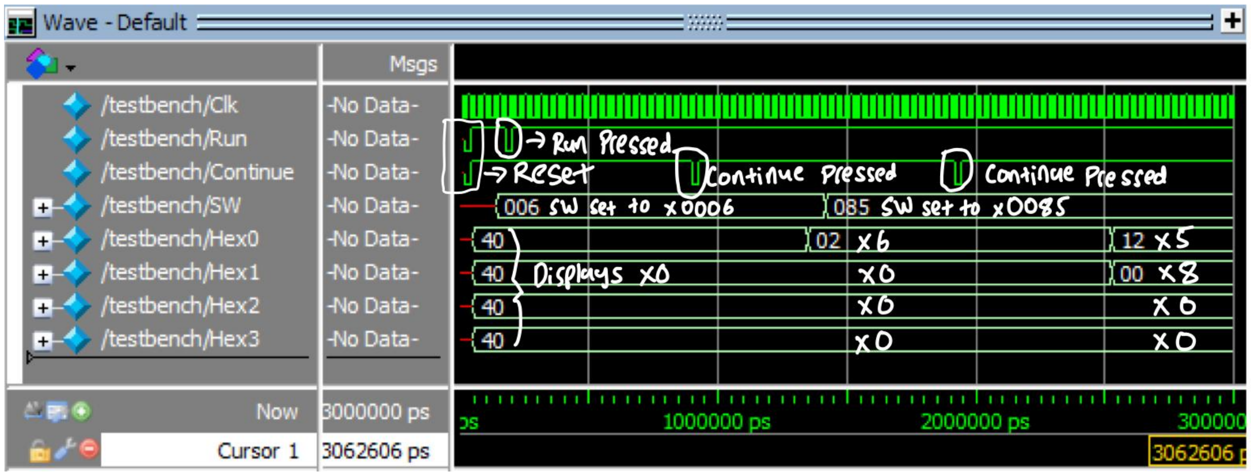
i) Basic I/O Test 1



First, both Run and Continue are set to low to reset the board. At this time, all 4 Hex are loaded with x40 and the 4 hex displays display x0000. The switches are then set to x0003 and then Run is pressed(executes Test 1). After multiple clock cycles the test is executed and Hex0 now stores x30 which corresponds to x3. At this stage, the 4 displays show x0003 corresponding to the value on the switches. The value of the switches is then changed to x85 and after a few clock cycles, the Hex values change to now display x0085 on the HEX displays.

Note: HEX3 HEX2 HEX1 HEX0 displays the 16-bit output of the value stored in switches.

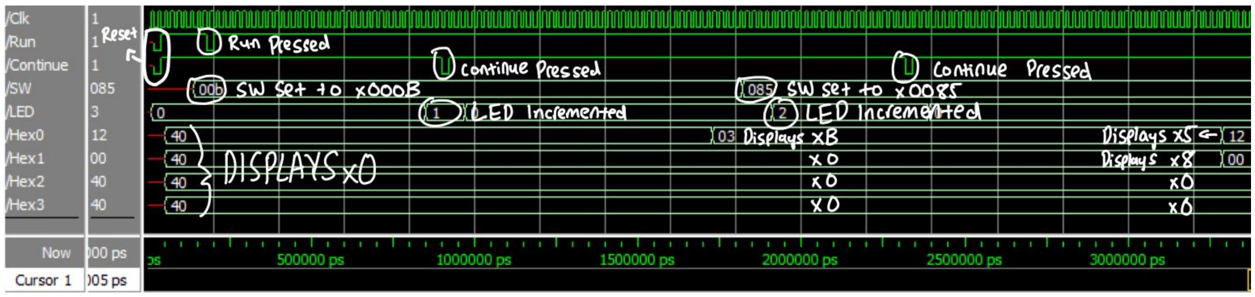
ii) Basic I/O Test 2



First, both Run and Continue are set to low to reset the board. At this time, all 4 Hex are loaded with x40 and the 4 hex displays display x0000. The switches are then set to x0006 and then Run is pressed(executes Test 2). After Continue is pressed the first time, shortly after, the 4 hex values are updated, and the 4 HEX displays show x0006 corresponding to the value of the switches. The SW value is then updated to x0085. The HEX displays continue displaying x0006 until Continue is pressed again, after which shortly after the Hex values update and now displays x0085 corresponding to the value on the switches.

Note: HEX3 HEX2 HEX1 HEX0 displays the 16-bit output of the value stored in switches

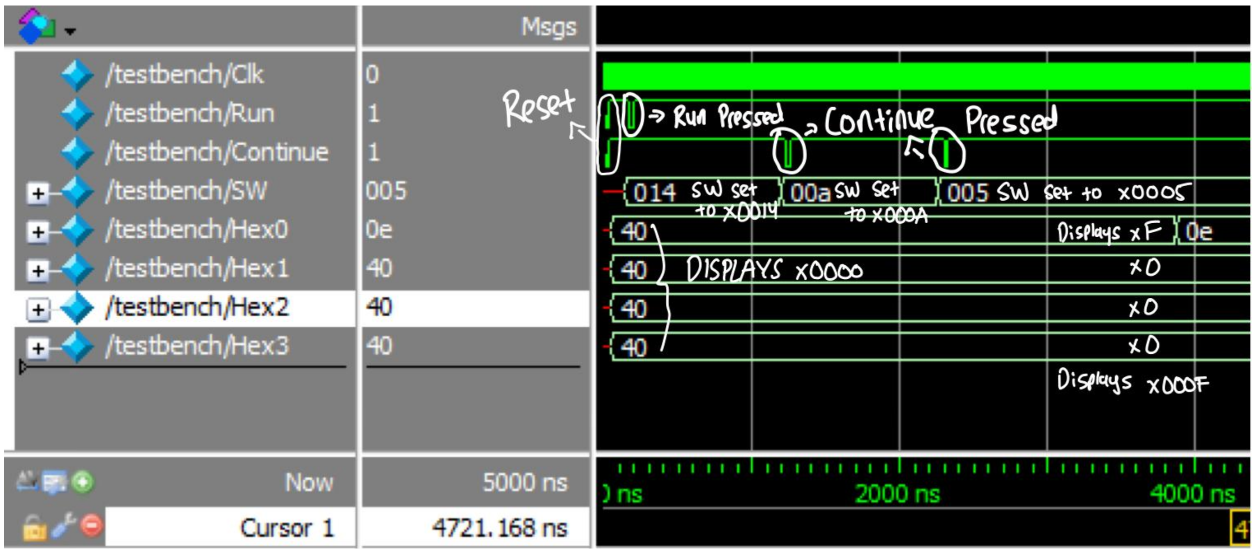
iii) Self-Modifying Code Test



First, both Run and Continue are set to low to reset the board. At this time, all 4 Hex are loaded with x40 and the 4 hex displays display x0000. The switches are then set to x0006 and then Run is pressed(executes Test 2). After Continue is pressed the first time, shortly after, the 4 hex values are updated, and the 4 HEX displays show x0006 corresponding to the value of the switches.Additionally the LED value increments to 1. The SW value is then updated to x0085. The HEX displays continue displaying x0006 until Continue is pressed again, after which shortly after the Hex values update and now displays x0085 corresponding to the value on the switches. Additionally the LED now increments to 2.

Note: HEX3 HEX2 HEX1 HEX0 displays the 16-bit output of the value stored in switches

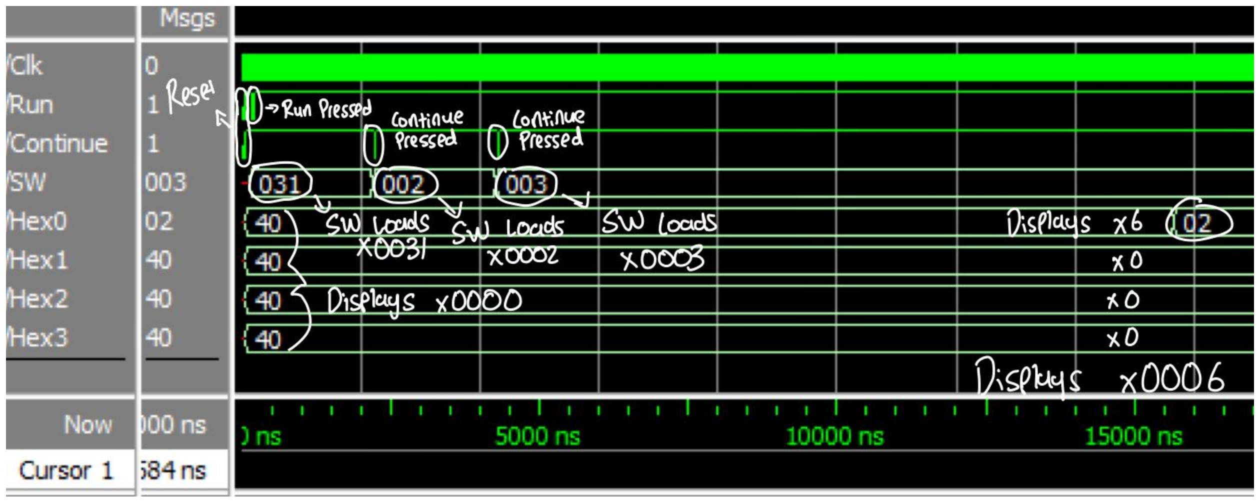
iv) XOR Test



First, both Run and Continue are set to low to reset the board. At this time, all 4 Hex are loaded with x40 and the 4 hex displays display x0000. The switches are then loaded with x0014(XOR Test) and Run is pressed. After a short period, the switch is set to x000A, continue is pressed, then switches are set to x0005 and continue is pressed again. These two values are XORed and the Hex0 updates and stores x0e which corresponds to x000F. Therefore, the hex displays correctly output the result of the XOR.

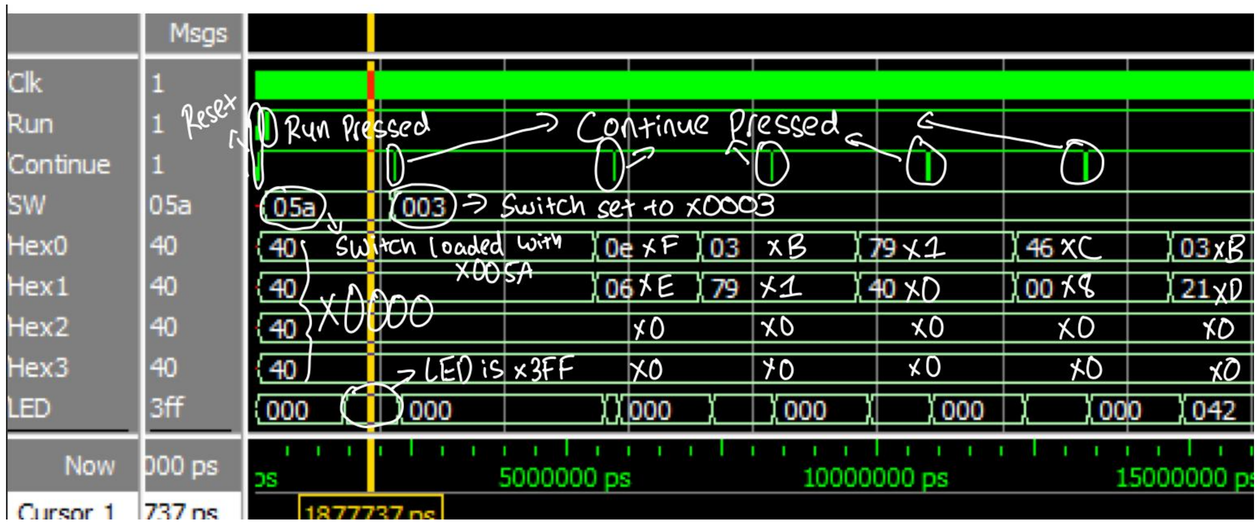
Note: HEX3 HEX2 HEX1 HEX0 displays the 16-bit output of the XOR Calculation

<u>v) Multiplication Test</u>



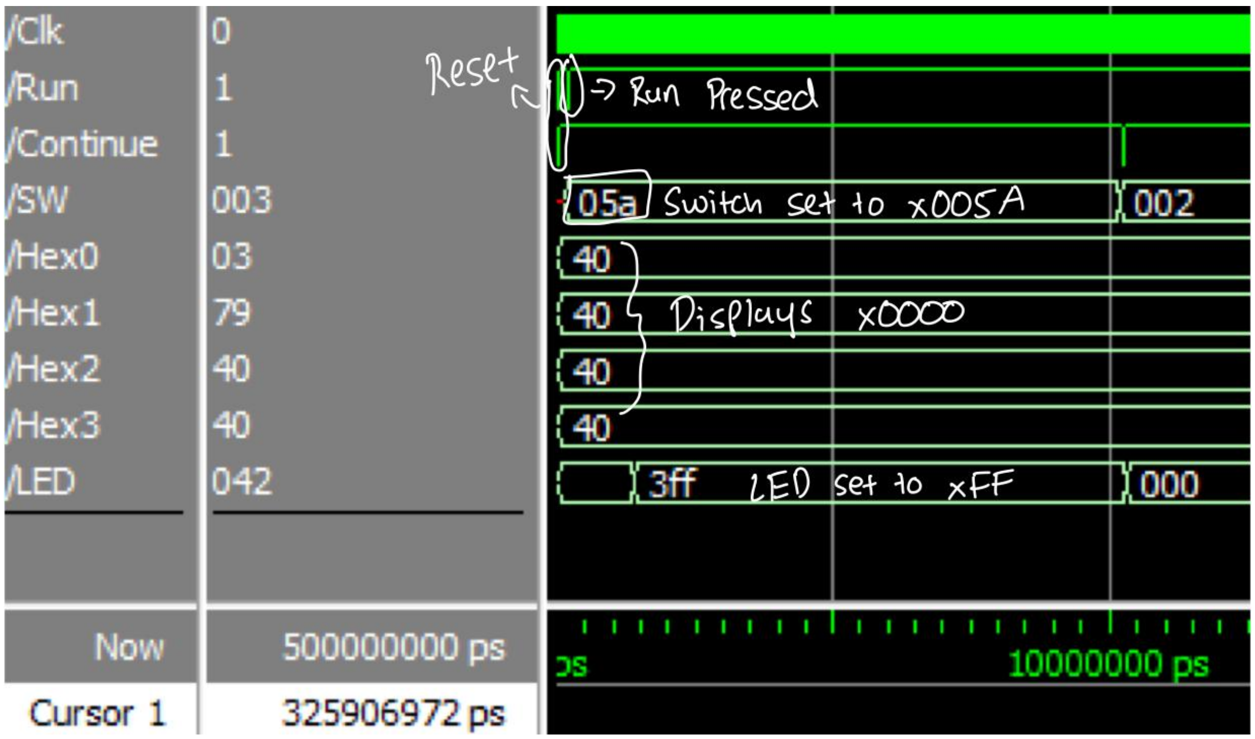
First, both Run and Continue are set to low to reset the board. At this time, all 4 Hex are loaded with x40 and the 4 hex displays display x0000. The switches are then loaded with x0031(Multiplication Test) and Run is pressed. After a short period, the switch is set to x0002, continue is pressed, then switches are set to x0003 and continue is pressed again. These two values are multiplied and the Hex0 updates and stores x02 which corresponds to x0006. Therefore, the hex displays correctly output the result of the multiplication.
Note: HEX3 HEX2 HEX1 HEX0 displays the 16-bit output of the Multiplication Calculation

<u>vi) Sort Test</u>
Simulation without sorting

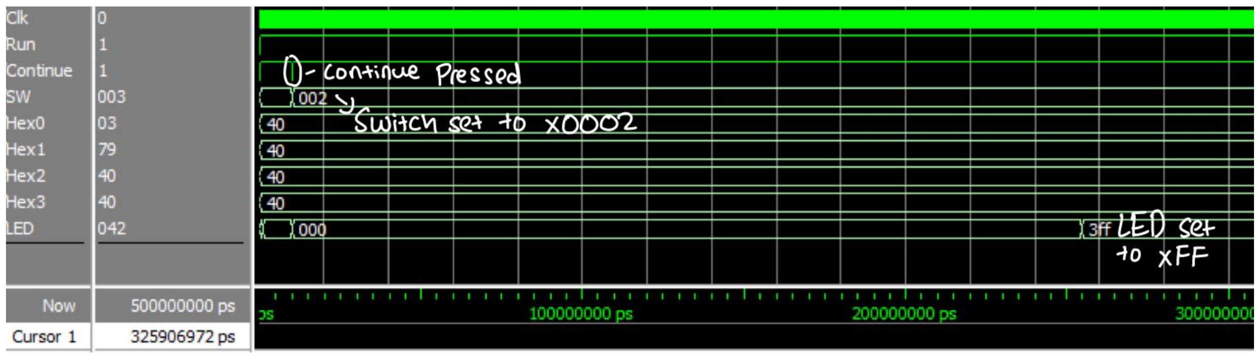


First, both Run and Continue are set to low to reset the board. At this time, all 4 Hex are loaded with x40 and the 4 hex displays display x0000. The switches are then loaded with x005A(Sort Test) and Run is pressed. At this stage, it is currently in the Menu part and LEDs display xFF. Switches are then set to x0003(calls display function), and continue is pressed. The HEX displays first display xEF. Each time continue is pressed the display updates and displays x1B, x01, x8C, xDB. This is unsorted since the sort function(x0002) is not called

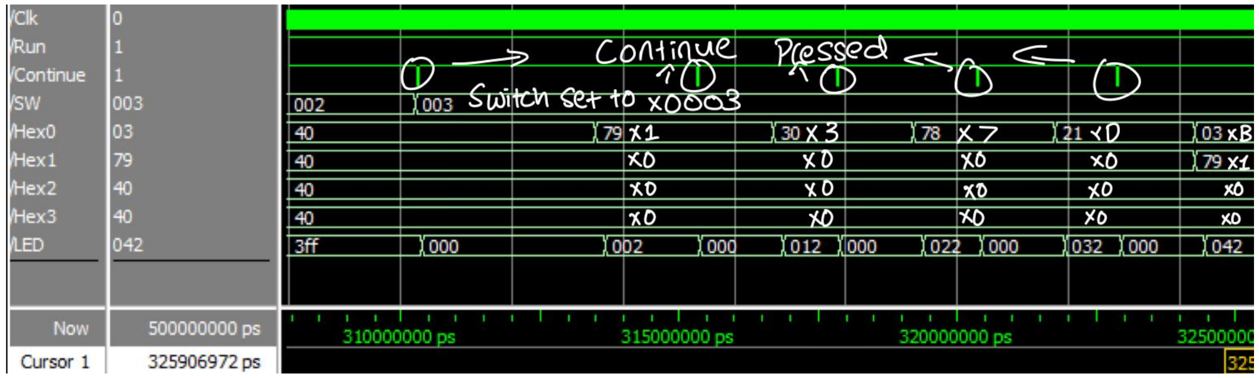
Simulation with sorting



First, both Run and Continue are set to low to reset the board. At this time, all 4 Hex are loaded with x40 and the 4 hex displays display x0000. The switches are then loaded with x005A(Sort Test) and Run is pressed.At this stage, it is currently in the Menu part and LEDs display xFF.



The switches are then set to x0002(Sort function) and continue is pressed to sort the elements. Once the elements have been sorted, the LED displays xFF.



Once the sort function is executed, the switches are set to x0003(display function). Every time continue is pressed, Hex0-3 updates and displays the sorted elements. With each press it displays x01, x03, x07, x0d, x1b. The values are now sorted

---

4. Post-Lab Questions

a. Fill out the Design Resources and Statistics table from Post-Lab question one.

DSP	0
Memory (BRAM)	18432
Flip-Flop	268
Frequency	72.89 MHz
Static Power	89.94 mW
Dynamic Power	0 mW
Total Power	98.63 mW

Total Power	98.63 mW

b. Answer all the post-lab questions.

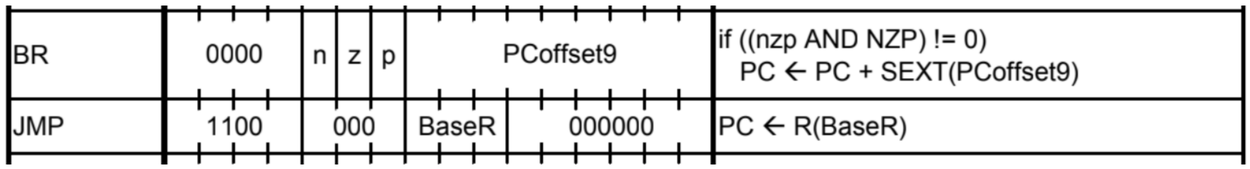
Q) What is MEM2IO used for, i.e. what is its main function?

MEM2IO is a module that serves as an interface to manage the I/O devices on the DE-10 lite board such as the switches and 7-segment displays. In our CPU, since I/O is memory-mapped, when a memory access occurs at an I/O memory address(0xFFFF), MEM2IO sends a signal to pause memory access and use data from the I/O device instead. When a load from memory address 0xFFFF is received(WE is low, OE is high), the data to CPU is loaded with the value from switches, while when a store to address 0xFFFF is received(WE is high), the data from the CPU is loaded and displayed on the HEX displays.

Physical I/O Device	Type	Memory Address	"Memory Contents"
DE10 Board Hex Display	Output	0xFFFF	Hex Display Data
DE10 Board Switches	Input	0xFFFF	Switches(9:0)

Q) What is the difference between BR and JMP instructions?

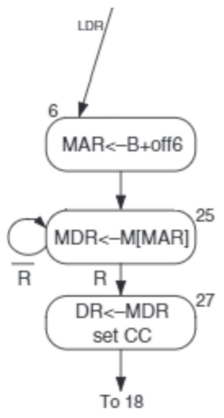
BR or branch is an instruction used to change the PC to prior or future instructions based on if the nzp register matches the condition code in the instruction. For example, if the nzp register matches the condition in the instruction, the new PC is set to the current PC added to a 9-bit sign-extended PCOffset specified in the instruction. If the conditions do not match PC increments as per normal and regular execution continues. In the JMP or jump instruction, the PC is set to a new memory address that is stored in the destination register. JMP unconditionally changes the PC, whereas BR changes PC depending on whether the conditions are satisfied. The new value PC takes is determined by the value stored in the Destination Register for JMP, while it depends on the current PC and the 9 bit offset for BR.

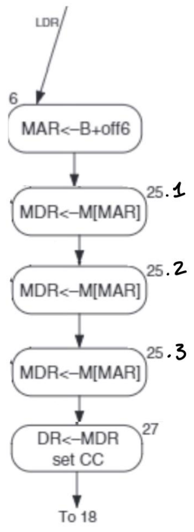


BR	0000	n	z	p	PCoffset9		if ((nzp AND NZP) != 0) PC ← PC + SEXT(PCoffset9)
JMP	1100	000			BaseR	000000	PC ← R(BaseR)

Q) 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?

Typically in LC-3, each instruction takes one clock cycle to occur. However, memory access may take additional clock cycles, in which case, the FSM will need to hold in particular states for much longer. This is taken care of by the R signal (Ready). Since states with memory access may take multiple cycles, the state continues to execute until the memory sends an R signal indicating that the memory access has been completed. In our SLC3, it is known that memory access takes 3 clock cycles at most. Hence, to compensate for the lack of R signal, each state accessing memory is split into 3 intermediate states, with all three states setting the same control signals required for memory access. For example, state 25, which copies the contents of MAR into MDR is split into 3 states. State 6 transitions to the first intermediate state, which then unconditionally transitions to the second and third intermediate states, before finally transitioning to state 27. In this way, the memory access state has 3 cycles to access the memory, and since it is known that the memory access is guaranteed to be completed within the 3 clock cycles, we eliminate the need for an additional Ready signal.

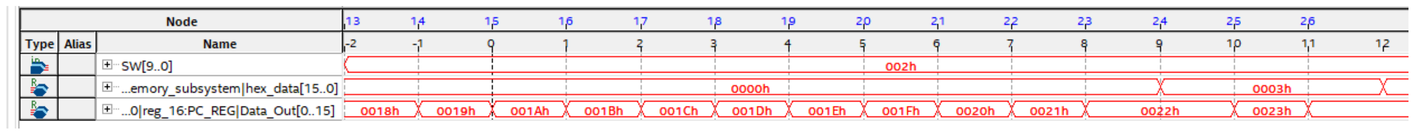




<u>EXTRA CREDIT:</u>

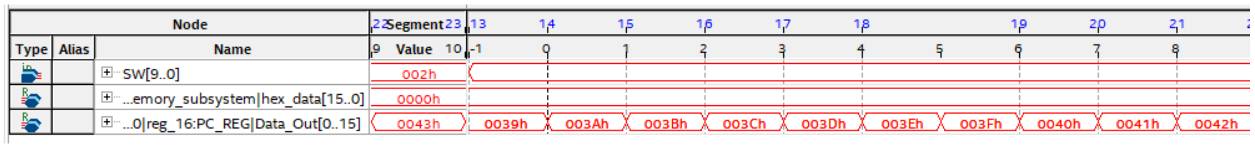
<u>TEST 1: XOR Function Execution:</u>

In this test, we started by inputting two values that are to be XORed together. We then decided to start our signaltap at PC = 0x001A because this is the instance in the memory contents and instructions list at which both values of XOR have been entered, and now we are required to analyze how long it takes for our program to actually give us our final output. My sample values entered were 0x0001, 0x0002 to give us an output of 0x0003 as shown at the clock cycle 9, accessible at clock cycle 10.
Hence, we get the final output at PC = x0023 which means that it took us a total of 9 instructions after the second value was entered to get the final output.
<u>Instructions</u>: 9 (because we ignore instructions required to input the 2 values for XOR)
<u>Cycles:</u> 11
<u>Speed</u>: (9*50,000,000 / 11) = 40.90 MIPS.

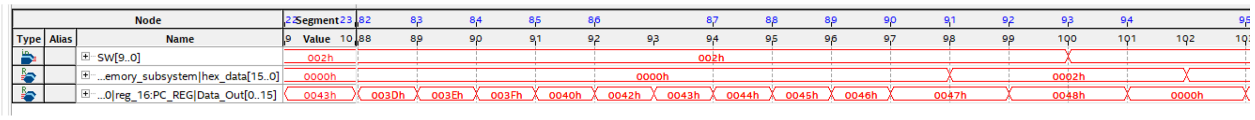


<u>TEST 2: Multiplier Function Execution:</u>

In this test, we were instructed to multiply 2 numbers. We decided to start our signaltap analyzer at PC = x003A because this indicates the position in memory contents when the multiplier and multiplicand has already been entered and now we are required to analyze how long it takes for our program to actually give us our final output. My sample values entered were 0x0001, 0x0002 to give us an output of 0x0002 as shown at the clock cycle 9, accessible at clock cycle 99.
Hence, we get the final output at PC = x0048 which means that it took us a total of 98 instructions after the second value was entered to get the final output. (This is because the last instructions of x0047 takes 2 cycles to complete).
<u>Instructions</u>: 98 (because we ignore instructions required to input the 2 values for Multiplier)
<u>Cycles:</u> 99
<u>Speed</u>: (98*50,000,000 / 99) = 49.49 MIPS.



The beginning section from clock cycle 0, indicating the values that PC starts with.



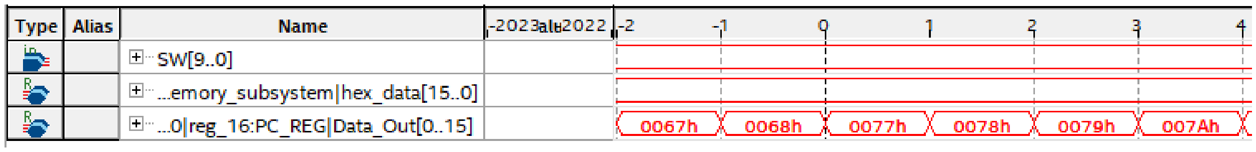
The end section of clock cycle 99, indicating the required output of x0002 is displayed on the switches, and now the PC has gone back to x0000.

<u>TEST 3: Sort Function Execution:</u>

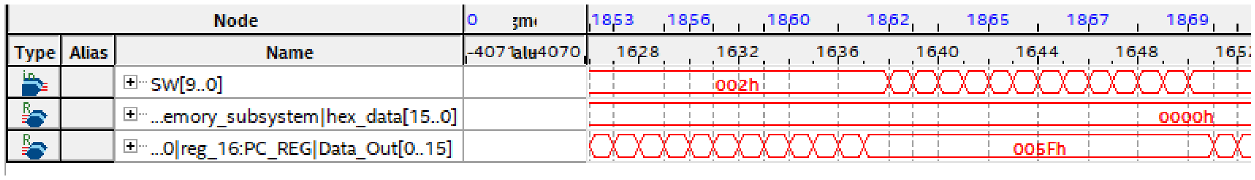
In this test, we were instructed to sort numbers that were given at specified memory locations. We decided to start our signaltap analyzer at PC = x0077 because this indicates the position in memory contents when the sorted values have been given, and we only want to analyze how long the sorting algorithm takes in terms of number of instructions and clock cycles. In our situation, the output itself is not visible because of the fact that the values have been changed at the memory locations themselves which signaltap could not access, but we have included an additional screenshot where we executed the display function again after the output was received in order to ensure that the values at the memory locations were actually sorted.

Hence, we get the final output at PC = x005F which means that it took us a total of 1638 instructions after the second value was entered to get the final output. (This is because the last instructions of x005F takes 13 cycles to complete).

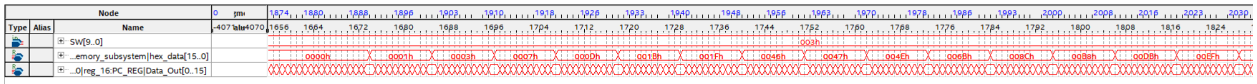
<u>Instructions</u>: 1638
<u>Cycles:</u> 1651 (This is because the last instruction of PC = x005F takes approximately 13 clock cycles to execute)
<u>Speed</u>: (1638*50,000,000 / 1651) = 49.61 MIPS.



The beginning section from clock cycle 0, indicating the values that PC starts with (x0077).



The end section at clock cycle 1651, after which we expect to see the sorted result in memory.



The screenshot of the section beyond clock cycle 1651, after the execution of the display function which shows us that post-algorithm, the values have been sorted.

Now, we finally notice that the MIPS values varied slightly, and the average value of the MIPS is:
MIPS_avg = (40.90 + 49.49 + 49.61) / 3
<u>**MIPS_avg = 46.667 MIPS.**</u>

<u>5. Conclusion</u>

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.
In this lab, we utilize the knowledge we have gained about SystemVerilog, and applied concepts learned from ECE 120 to successfully design a 16-bit SLC-3 processor, synthesize it on Quartus using ModelSim, and implement the processor on our FPGA board. We came across numerous bugs while designing the processor. In our ISDU, for state 12 of JMP, we initially chose the wrong path to load the contents of Base register into PC. We loaded it through the ADDR1MUX rather than passing the value through the ALU. We had the wrong control signals set for this state because of which JMP would not work. To fix this, we looked back at the SLC3 datapath and realized we had to follow a different path and accordingly change the control signals. Additionally, we

noticed that the incorrect value of PC was being loaded. By running simulations on modelsim we were able to debug and realize that we had swapped the inputs of MIOMUX, after which the Processor worked perfectly.
