Lab 5 Report

Noah Conner & Adam Naboulsi
Section ABP

Introduction:

Summary:

Our 16-bit SLC-3 processor is a simplified version of the full LC-3 processor. The computer can perform the following subset of the LC-3 ISA: ADD, ADDi, AND, ANDi, NOT, BR, JMP, JSR, LDR, STR and PAUSE. Having a RISC design, the CPU can perform most instructions in a single clock cycle, disregarding the fetch and decode sequence. The computer is complete with a 1024 word, or 2 Kb, RAM and a memory-mapped input/output. The user can interface with the computer from the FPGA development board's ten switches and two buttons, where KEY0 is the continue button and KEY1 is the run button.

SLC-3 Diagrams & Descriptions:

Summary of Operation

The SLC-3 processor can perform eleven different instructions. Since this is a 16-bit processor, the instructions, memory and registers are all 16-bit. This limits us to how much functionality we can encode in a single instruction. For the purpose of this lab, we implemented a small subset up the LC-3 ISA and avoided more advanced CPU features such as interrupts. Through the FPGA development board, the user is able to input values using the switches, read values from the hex displays and run, reset and continue using the buttons. At startup, the user should reset the program by pressing both buttons and releasing the run button before releasing the continue button. Now the user is able to input the address of whichever program they want to execute and press the run button. Some programs have PAUSE instructions which pause execution until the continue button is pressed. This period allows the user to input or read values from the computer. The LEDs and the leftmost hex drivers provide information for which PAUSE instruction the user is at and whether the pause is for reading, inputting or both.

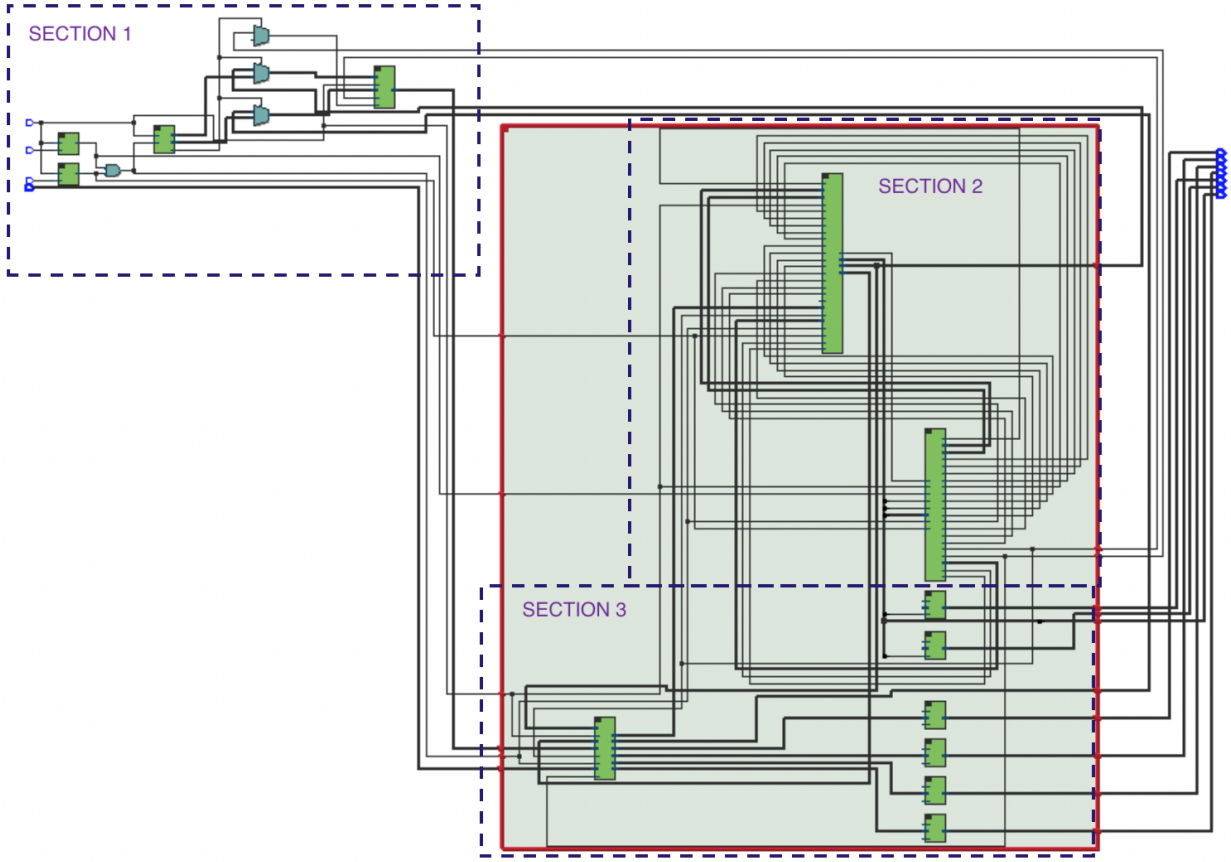
How SLC-3 Performs Functions:

The processor goes through a Fetch-Decode-Execute cycle to perform an instruction. Firstly, the processor fetches the instruction from memory and increments the program counter (PC) which keeps track of where in the program the CPU is at. This is a simple process that takes four states to perform. It first loads the PC into the memory address register (MAR) and increments the PC. Then it retrieves the instruction stored at the address in the MAR and loads that into the memory data register (MDR), then the instruction register (IR). In the Patt and Patel design of the LC-3 processor, an R signal (ready signal) is used with the memory management unit which indicates when the memory is done reading or writing. The memory raises the R signal high once the memory is ready to be read from or has been written to, and indicates to the processor that it is ready to move on from the current state to the next. Our on-chip memory has no R signal, so rather than wait for the R signal in our design, we instead wait a fixed number of cycles (in accordance with the latency of the fully synchronous setup of a read/write on the FPGA memory) within the state that performs a read or write instruction. In our configuration with the on-chip memory timing, we must wait two extra clock cycles for a read and one clock cycle for a write, due to latches used in the on-chip memory. To do this, we extend the read/write states through adding additional states for that specific instruction. This is why the fetch cycle has 2 extra states. Once the instruction is in the IR it can be decoded and executed. Each instruction has a unique 4-bit opcode that tells the state machine which control signals should be activated. The control signals essentially govern how the data paths are configured between the ALU, register bank and data bus e.i. MUX select, bus gates, ALU function and register load signals. As an example, when the state machine comes across an ADD

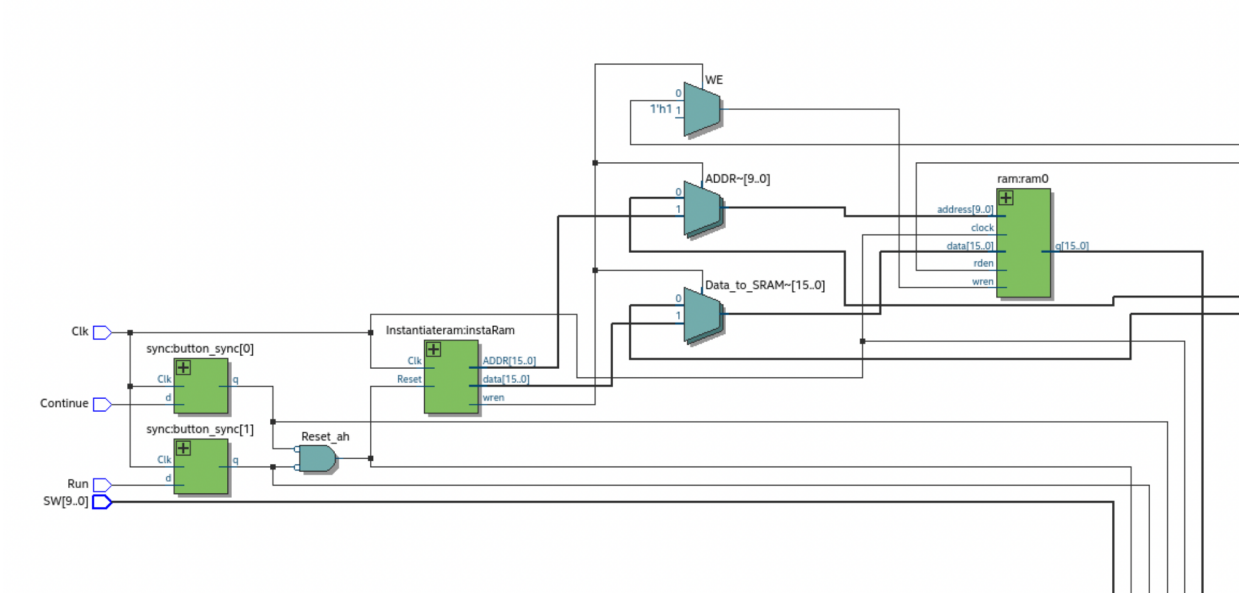
instruction, it configures the data paths so that the two source registers are added via the ALU and the sum is transferred to the destination register via the data bus. Other instructions include ADDi, AND, ANDi, NOT, BR, JMP, JSR, LDR, STR and PAUSE. All instructions execute in one clock cycle with the exception of STR and LDR because they have to wait on SRAM to complete a read or write. These instructions are fairly easy to distinguish except for BR and JMP. They seem very similar, however, there are a couple subtle differences. Firstly, the BR instruction requires that a specified condition be true in order to branch to a certain location. The JMP instruction will always branch to the designated location. The other difference is how they specify where to branch to in memory. The BR instruction accepts a signed 9-bit offset whereas the JMP instruction branches to the value stored in a specified register. This means the JMP instruction is not bound to the PC whereas the BR instruction can only branch a finite distance from the current PC value.

SLC-3 Block Diagram:

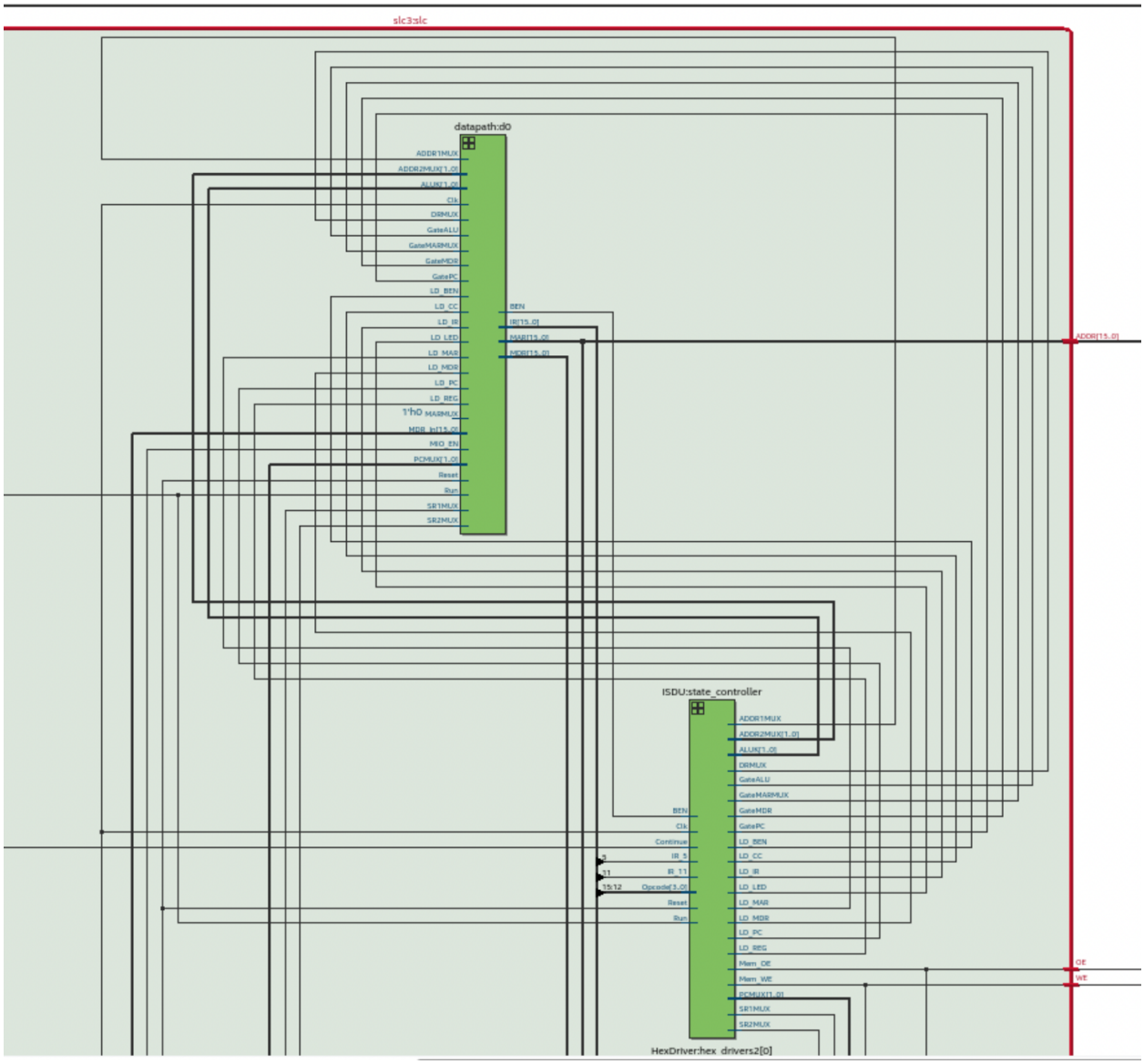
OVERVIEW:



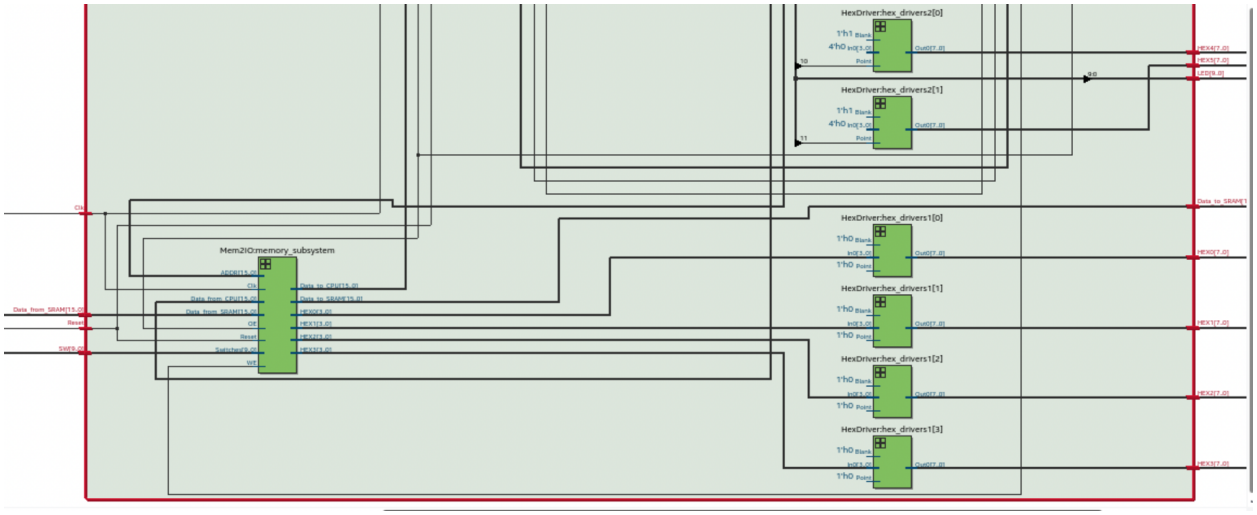
Section 1


WE
1'h1
ADDR~[9..0]
Data_to_SRAM~[15..0]
ram:ram0
address[9..0]
clock
data[15..0]
rden
wren
q[15..0]
Clk
Continue
Run
SW[9..0]
sync:button_sync[0]
sync:button_sync[1]
Clk
d
q
Reset_ah
Instantiateram:instaRam
Clk
Reset
ADDR[15..0]
data[15..0]
wren

Section 2


slc3:slc
datapath:d0
ADDR1MUX
ADDR2MUX[1..0]
ALUK[1..0]
Clk
DRMUX
GateALU
GateMARMUX
GateMDR
GatePC
LD_BEN
LD_CC
LD_IR
LD_LED
LD_MAR
LD_MDR
LD_PC
LD_REG
1'h0 MARMUX
MDR_In[15..0]
MIO_EN
PCMUX[1..0]
Reset
Run
SR1MUX
SR2MUX
BEN
IR[15..0]
MAR[15..0]
MDR[15..0]
ADDR[15..0]
ISDU:state_controller
ADDR1MUX
ADDR2MUX[1..0]
ALUK[1..0]
DRMUX
GateALU
GateMARMUX
GateMDR
GatePC
LD_BEN
LD_CC
LD_IR
LD_LED
LD_MAR
LD_MDR
LD_PC
LD_REG
Mem_OE
Mem_WE
PCMUX[1..0]
SR1MUX
SR2MUX
BEN
Clk
Continue
IR_5
IR_11
15:12 Opcode[3..0]
Reset
Run
5
11
OE
WE
HexDriver:hex_drivers2[0]

Section 3



Module Descriptions:

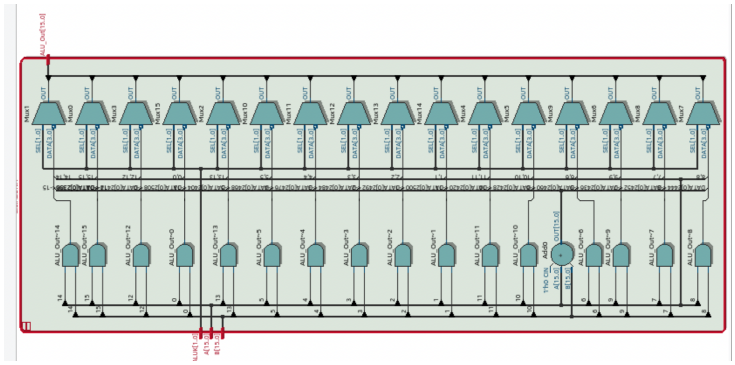
Module: alu.sv

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

Outputs: [15:0]Result

Description: Based on input ALUK, this module computes and returns (outputting to Result) the sum of A and B, the bitwise& of A and B, the not of A, or unmodified A.

Purpose: Serves as the arithmetic logic unit in our SLC-3 processor.



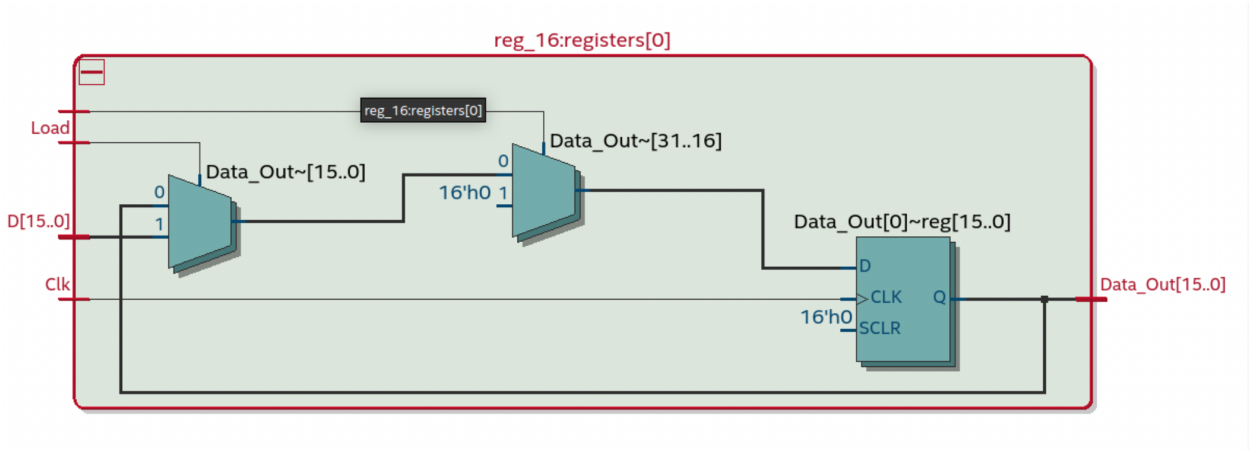
Module: reg_16 (in reg_file.sv)

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

Outputs: [15:0]Dout

Description: Synchronous with the clock edge (always_ff), this module sets Dout to 0 (clears reg) if Reset is high, otherwise, if Ld is high, it sets Dout to input D.

Purpose: Serves as a building block within our register file module described below; a clock synchronized 16-bit register.



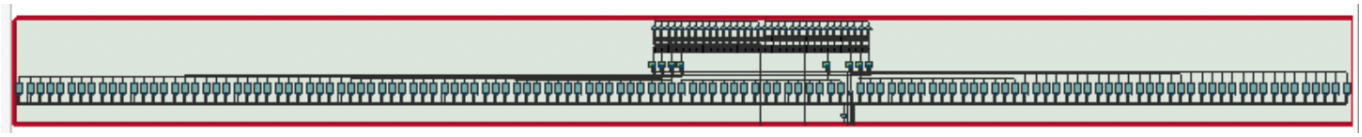
Module: reg_file.sv

Inputs: Clk, Reset, Load, [2:0]DR_ADDR, [2:0]SR1_ADDR, [2:0]SR2_ADDR, [15:0] DR

Outputs: [15:0]SR1, [15:0]SR2

Description: Instantiates 8 16-bit registers using module reg_16 as array. Contains internal logic [7:0]Ld which represents the load signal for each register within the file; [15:0] D [7:0] which represents the Data_In for each register in the file; and [15:0] Dout [7:0] which represents the Data_Out of each register in the file. Sets values of internal logic Ld according to DR_ADDR's input value through ('DR_ADDR') left shifts of input Load into Ld – in always comb. Sets internal logic D at a dereferenced array location [DR_ADDR] to input DR (represents desired data into specified register through DR_ADDR) – always comb. Sets output SR1 to Dout dereferenced at array location [SR1_ADDR]; sets output SR2 to Dout dereferenced at array location [SR2_ADDR] – always comb.

Purpose: Serves as a 16bit x 8 register file in our SLC-3 processor; loads inputted values into registers according to DRMUX in the design and outputs stored register values according to SR1MUX and SR1 inputs.



Module: HexDriver.sv

Inputs: [3:0] In0, Point, Blank

Outputs: [7:0] Out0

Description: Uses a unique case statement (based on In0) to map the outputs to the corresponding LED segments (7 total, +1 with point) on the HEX display of the FPGA board. If input Point is high, sets Out0[7] to 0 (displaying dot on corresponding HEX display); if Blank is high, sets Out0[6:0] to 1s (turning off all led segments).

Purpose: Used to output bits of registers in the register unit to the HEX displays on the FPGA board. Inputs point and blank were used to represent the pauseIR states on the segments.

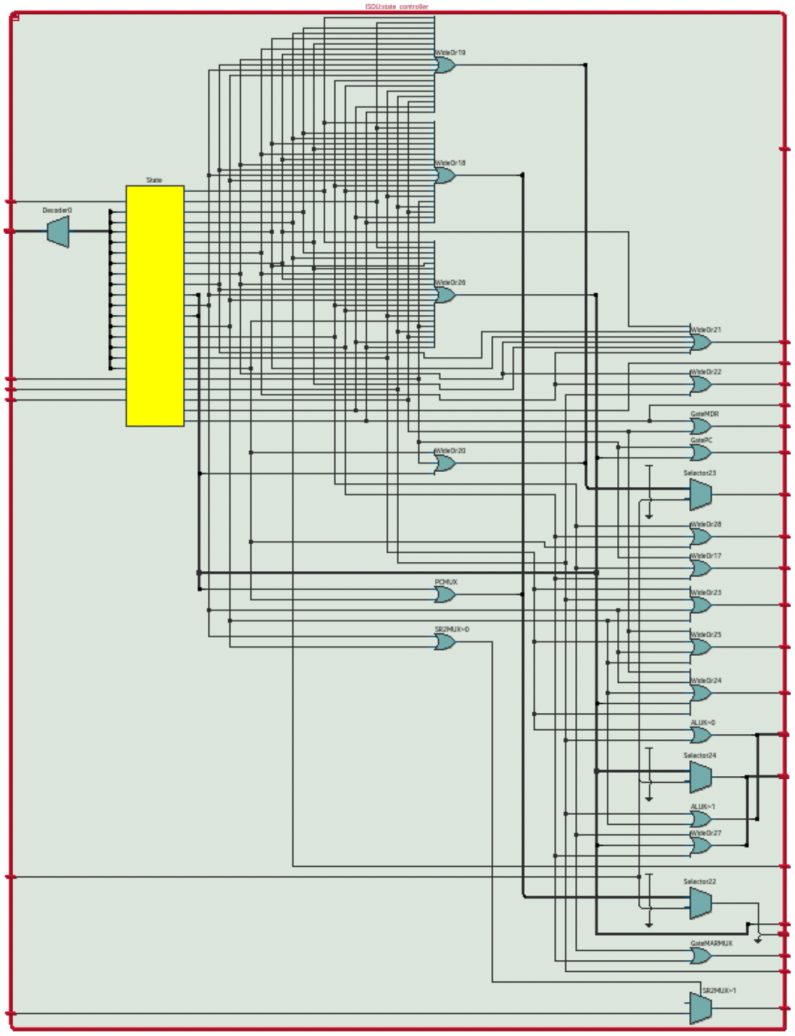
Module: ISDU.sv

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

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, [1:0]ALUK, Mem_OE, Mem_WE

Description: Instantiates all states in our SLC-3 design (including ADD, STR, LDR, etc.) as enum logic as State and Next_state. In an always_ff block, it sets the State = Halted if input Reset is high, otherwise, it sets the State = Next_state. In an always comb block, half the section is designated to setting Next_state according to each state in the enum logic and the other half is designated to setting the outputs of the modules (load signals, gate signals, Mux values, etc.) – will refer to these as functional outputs. Initially it sets all of the LD, Gate, ALUK, MUX, and Mem signals to 0's as default and Next_state = State.

Purpose: Acts as our state machine outputting the relevant functional output values for our datapath and mem2io modules to take in.



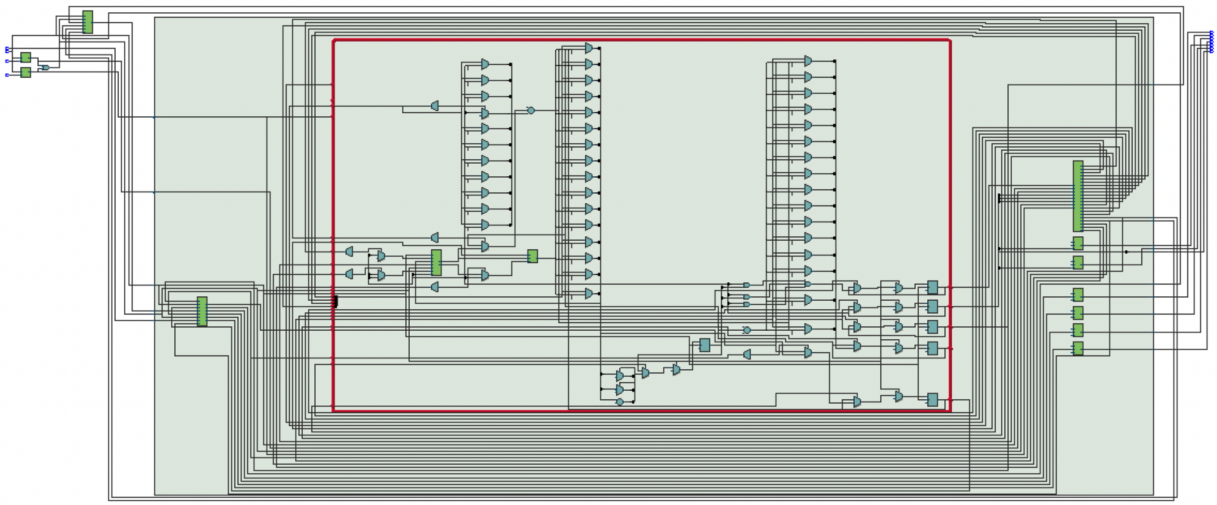
Module: datapath.sv

Inputs: Clk, Reset, 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, [1:0]ALUK, [15:0]MDR_IN, MIO_EN

Outputs: [15:0]MAR, [15:0]MDR, [15:0]IR, [15:0]ProgramCount, BEN

Description: In an always_ff block ,we set the output values of MAR, MDR, IR, BEN, CC(NZP), and PC to the at the positive edge of each clock cycle to 0s if Reset is high, otherwise we set the values to their corresponding next values (represented through internal logic). The next values are calculated in an always_comb block. In the first half of our always comb block, we set the outputs of the addr2mux, addr1mux, sr1mux, sr2mux, and drmux (following our block diagram) using unique case statement according to their corresponding select input values (ADDR2MUX, ADDR1MUX, SR1MUX, SR2MUX, and DRMUX respectively). We also instantiate the BUS of our SLC-3 and use a case statement which sets the values to addr1mux+addr2mux output, the ALU output, MDR, or PC according to which of the GateMARMUX, GateALU, GateMDR, and GatePC gates are high (respectively). In the second half of our always_comb block, we set the internal next logic (MAR_next, MDR_next, IR_next, etc.) to their current values (MAR, MDR, IR,etc.). Then using if-else/unique case statements according to the corresponding load signals, we set the next logic to the correct values (following our SLC-3 design). In this module, we also instantiate an ALU and a register file, again following our SLC-3 design and establishing the data paths relevant to the ALU and register file.

Purpose: This module sets all of the datapaths in our SLC-3 design; essentially connects our SLC-3 design establishing all of the units of the processor and connecting them.



Module: MEM2IO.sv

Inputs: [15:0]ADDR, OE, WE, [9:0]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: In an always_comb block,it default sets Data_to_CPU to 0, otherwise if WE (write enable) is low and OE (output enable) is high and ADRR[15:0] is FFFF, it sets it to the value of the switches sign-extended, else (if ADDR != FFFF) it sets it to Data_from_SRAM. To pass data from the CPU to SRAM, in an always_ff it writes 0s to the leds (hex_data–>HEX) when reset is high, and writes Data_from_CPU to the leds when WE is high and ADDR is FFFF.

Purpose: The MEM2IO module acts as a sort of memory management unit for the SLC-3. Since the computer uses a memory-mapped IO it is necessary to have a unit like this to map certain addresses to an input or output interface rather than physical memory. It intercepts a particular address from the address space, sits in between the cpu and the memory (real memory from the board or simulation memory - test mem). It looks for the special address xFFFF (hardcoded); when you do a load from FFFF, mem2io intercepts the load request and instead reads the value of the switches (sign extends it to 16 bits) and sends it over to the cpu; cpu reads from the switches through memory load from address FFFF. Similarly, when we do a memory store, at xFFFF, instead of going into the real memory, it goes into the hex displays.

Module: slc3.sv

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

Outputs: [9:0]LED, OE, WE, [7:0]HEX0, [7:0]HEX1, [7:0]HEX2, [7:0]HEX3, [7:0]HEX4, [7:0]HEX5, [15:0]ADDR, [15:0]Data_to_SRAM

Description: Instantiates the datapath module, the Mem2IO module (memory_subsystem), and the ISDU module (state machine). Also instantiates necessary internal logic for connections between instances of the modules. Also instantiates 6 hex drivers where 4 represent the hex_data from MEM_2IO and the other two display IR[11] and IR[10] in conjunction with the pause states through (dots/blanks on our hex displays); ‘10’ means the previous operation was a write to the hex display, ‘01’ means the next operation is a read from switches, and ‘11’ means both. Lastly, it assigns the LEDs to IR[9:0].

Purpose: Serves as the second highest top-level of our SLC-3 connecting the datapaths, the memory subsystem, the state machine, and setting the hex displays.

Module: slc3_testtop.sv

Inputs: [9:0]SW, Clk, Run, Continue
Outputs: [9:0]LED, [7:0]HEX0, [7:0]HEX1, [7:0]HEX2, [7:0]HEX3, [7:0]HEX4, [7:0]HEX5

Description: Instantiates the slc3 module and test_memory (establishing the processor and creating the simulation memory for testing). Also declares the push button as active high signals. Lastly, instantiates sync modules to synchronize the push buttons (which represent the run and continue signals) with the Clk.

Purpose: Acts as the top-level for simulation purposes of our SLC-3 (uses virtual memory).

Module: slc3_sramtop.sv

Inputs: [9:0]SW, Clk, Run, Continue
Outputs: [9:0]LED, [7:0]HEX0, [7:0]HEX1, [7:0]HEX2, [7:0]HEX3, [7:0]HEX4, [7:0]HEX5

Description: Instantiates the slc3 module, ram/instantiateram (establishing the processor and creating the on-chip memory). Also declares the push button as active high signals. Instantiates sync modules to synchronize the push buttons (which represent the run and continue signals) with the Clk. Lately, in always_comb blocks, it sets the WE (write enable) value, the Data_to_SRAM value, and the ADDR value (internal logic passed in as inputs/outputs to the instantiated modules).

Purpose: Acts as the top-level for our FPGA programmed SLC-3 processor (uses on-chip memory)

Module: Instantiatesram.sv

Inputs: Reset, Clk
Outputs: [15:0]ADDR, [15:0]data, wren

Description: Sets the instruction data values corresponding to the address (PC value) to the desired opcode/operation.
Purpose: Instantiates our on-chip memory.

Module: test_memory.sv

Inputs: Clk, [15:0]data, [9:0]address, rden, wren
Outputs: [15:0]readout

Description: Creates virtual memory similar to the on-Chip memory on the MAX10 board.
Purpose: Virtual memory for simulation purposes of our SLC-3 processor.

Module: memory_contents.sv

Description/Purpose: Memory contents of our test memory; allows us to modify and test specific opcodes.

Module: SLC3_2.sv

Description/Purpose: Reference for functions and constants included in the file (opcodes and operations).

Module: Synchronizers.sv

Inputs: Clk, d

Outputs: q

Description: The synchronizer module outputs a clock-synchronized signal of an asynchronous input signal. Includes 3 sync modules, one with no reset for switches and buttons (sync), one with reset to 0 (sync_r0), and one with reset to 1 (sync_r1).

Purpose: This is used to synchronize all of the user-inputted signals.

ISDU Operation:

Next_state assignments: Initially, in the first half of our always_comb block, our ISDU sets all of the LD, Gate, ALUK, MUX, and Mem signals to 0's as default and Next_state = State. For the Halted state in our design, it sets Next_state = S_18 if input signal Run is high. In state 32 of our design, it uses a case statement to select the next state (ADD, AND, NOT, BR, JMP, JSR, LDR, STR, or PAUSE) according to the 4-bit opcode input. The next_states for the final states of each of these operations is S_18 (state machine returns to beginning of fetch instructions). For part of the fetch section of our machine – state 33 – we split up the instruction (MDR <– M[MAR]) into 3 separate states to ensure MDR takes in the right value. For the LDR instruction, we split up the operation into 5 consecutive states (to ensure mem is loaded correctly). For the STR instruction, we split up operation into 3 states (again to ensure memory is stored correctly).

Output signals descriptions/purposes: In the second half of the always_comb block we set the functional outputs according to the current state of the machine. The LD_MAR signal corresponds to the load signal for the MAR block in our SLC-3 which allows the MAR to be loaded with the value of the bus if the signal is high. The LD_MDR signal corresponds to the load signal for the MDR block in our SLC-3 which allows MDR to be loaded with either data from MEM2IO or the bus when the signal is high. The LD_IR signal corresponds to the load signal for the IR block in our SLC-3 which allows IR to be loaded with data from the BUS when the signal is high. The LD_BEN signal corresponds to the load signal for the BEN block in our SLC-3 which allows BEN to be loaded with output data from the logic block that takes in IR[11:9] and NZP when the signal is high. The LD_CC signal corresponds to the NZP block in our SLC-3 which allows the NZP values to be set according to the output data of the logic block which takes in the bus data in our diagram when the signal is high. LD_REG corresponds to the load signal of our register file which allows any of its registers to take in a value from the data BUS when the signal is high. LD_PC corresponds to the load signal of the PC block which allows the PC to load in data from PCMUX when the signal is high whether it is from an increment, the bus, or the output of the ADDR2MUX+ADDR1MUX block. LD_LED corresponds to the load signal for our ledVect12 in our

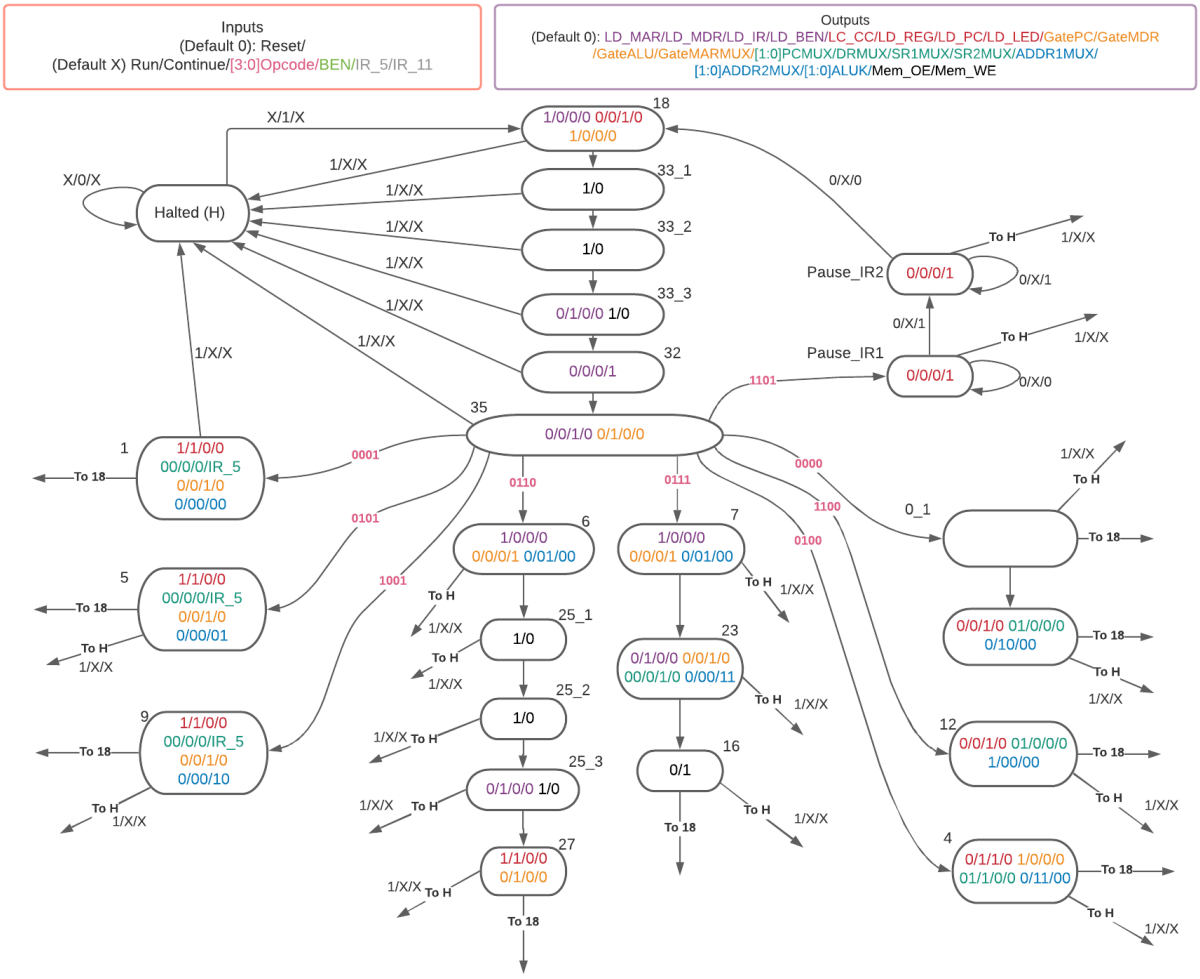
SLC-3 (did not end up using) which allows the vector to load in values when the signal is high. GatePC, GateMDR, GateALU, and GateMARMUX correspond to the (originally tristate buffers) muxes for the data bus in our SLC-3 which allow the data bus to take in the the value of the PC, the value of MDR, the value of the ALU output, or the value of ADDR1MUX + ADDR2MUX correspondingly. The ALUK signal corresponds to the select of our arithmetic logic unit which determines which of the ALU's operations on its inputs A and B should be done. PCMUX corresponds to the select of the PCMUX which selects between the bus data, the output of ADDR1MUX + ADDR2MUX, and the increment of the PC for its data out. DRMUX corresponds to the select signal of the DRMUX in our SLC-3 which allows us to pick between '111' and IR[11:9] to be its output. SR1MUX corresponds to the select signal of our SR1MUX which picks between IR[8:6] and IR[11:9] to be its output. SR2MUX corresponds to the reg selector input of our register file (SR2). ADDR1MUX refers to the select of our ADDR1MUX which picks between 16'b0, IR[5:0] sext16, IR[8:0] sext16, and IR[10:0] sext16 to be its output. ADDR2MUX corresponds to the select of our ADDR2MUX which picks between the value of PC and the SR1 output of our register file to be its output. Mem_OE refers to output enable which indicates to our MEM2IO module when memory should be outputted. Mem_WE refers to write enable which indicates to our MEM2IO module when a write to memory should occur.

State functional output assignments: In state 18, we set GatePC to 1, LD_MAR to 1, PCMUX to 0s, and LD_PC high, which allows MAR to take in PC and then increment PC. In state 33_1 and 33_2 we set Mem_OE high which allows a read from memory. In state 33_3 we set Mem_OE high again and LD_MDR high which allows the read-in memory (particularly the current instruction value) to be assigned to MDR. In state 35, we set GateMDR high and LD_IR high which allows IR to take in MDR's value. In state 32 we set LD_BEN high which allows BEN to be loaded with IR[11:9]&NZP before moving on to the opcode states. In state 1 (ADD) we set SR2MUX to IR_5 (IR[5]), ALUK to 0s, GateALU high, LD_REG high, and LD_CC high which allows an add instruction between A and B (SR1 + SR2 or SR1 + imm5) in the ALU to be performed and loaded into the desired destination register of our reg file. In state 5 (AND) we set SR2MUX to IR_5, ALUK to 01, GateALU high, LD_REG high, and LC_CC high, allowing for an and instruction between A and B (SR1&SR2 or SR1&imm5) in the ALU to be performed and loaded into the desired register of our reg file. In state 9 (NOT), we set SR2MUX high, ALUK to 10, GateALU high, LD_REG high, and LC_CC high which allows a not operation to be performed by the ALU of SR1 in our reg file and stored to the desired destination register in our reg file. In state 6_1 (LDR) we set LD_MAR high, GateMARMUX high, ADDR 2MUX to 01, and ADDR1MUX high to allow MAR to take in the value of a base register + an offset defined by the instruction value. States 6_2 and 6_3 set Mem_OE high to allow for memory to be read at location baseReg+offset through MEM2IO. In state 6_4, we set Mem_OE high and LD_MDR high to allow MDR to take in the memory read in. In the last state of LDR, 6_5, we set LD_REG high, LD_CC high, and GateMDR high to allow the desired destination register to take in MDR. In state7_1 (STR), we set LD_MAR high, GateMARMUX high, ADDR2MUX to 01, and ADDR1MUX high to allow MAR to take in the value of a base register + an offset. In state 7_2, we set LD_MDR high, GateALU high, ALUK to 11, and SR1MUX high to allow MDR to take in the value of a source register from our reg file through our ALU/bus. In the last state of STR, 7_3, we set Mem_WE high to allow M[MAR] to take in MDR. In state 4 (JSR) we set LD_PC high, LD_REG high, GatePC high, ADDR2MUX to 11, PCMUX to 01, and DRMUX high which allows register 7 of our reg file to take in PC and allows PC to take in PC + a sign-extended value from our instruction value. In state 12 (JMP), we set LD_PC high, ADDR1MUX high, and PCMUX to 01 which allows PC to take in a value from a register in our reg file. In state 0 (BR), we set LD_PC high,

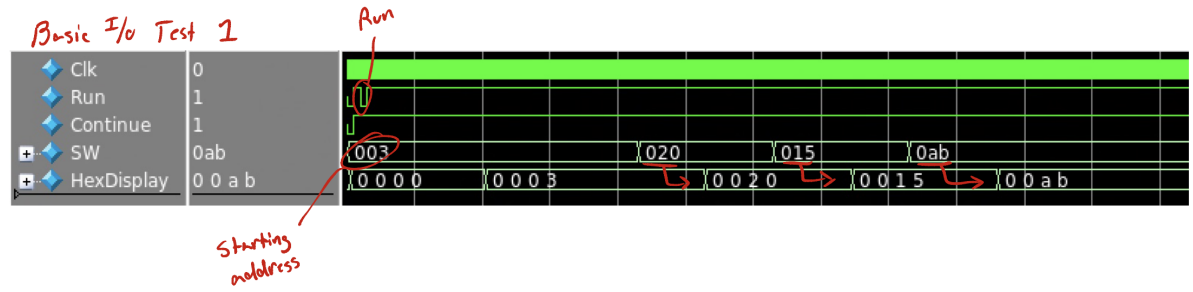
PCMUX to 01, and ADDR2MUX to 10 if BEN is high which allows the PC to take in PC+sign-extended value from instruction value. Finally, our PauseIR1 and PauseIR2 states set LD_LED high to allow ledVect12 to load in a new value (did not end up utilizing in our design).

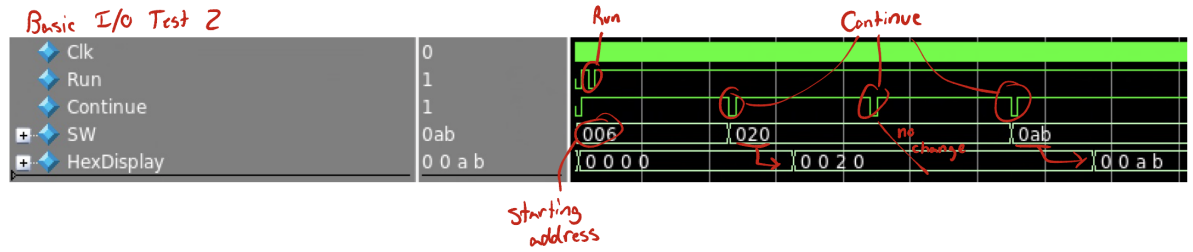
State Diagram ISDU

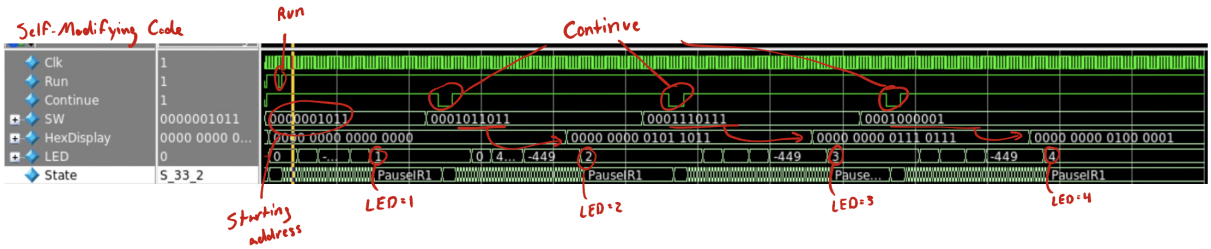
State Diagram:

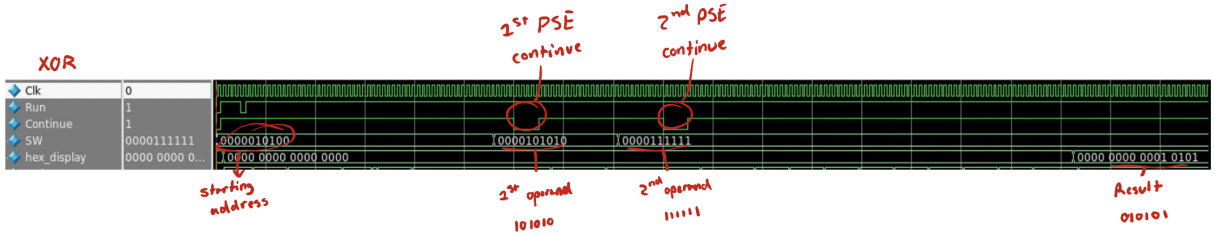


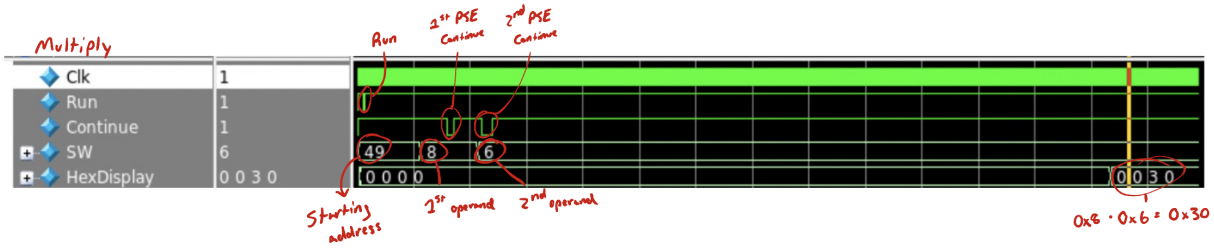
Simulation Test Programs:

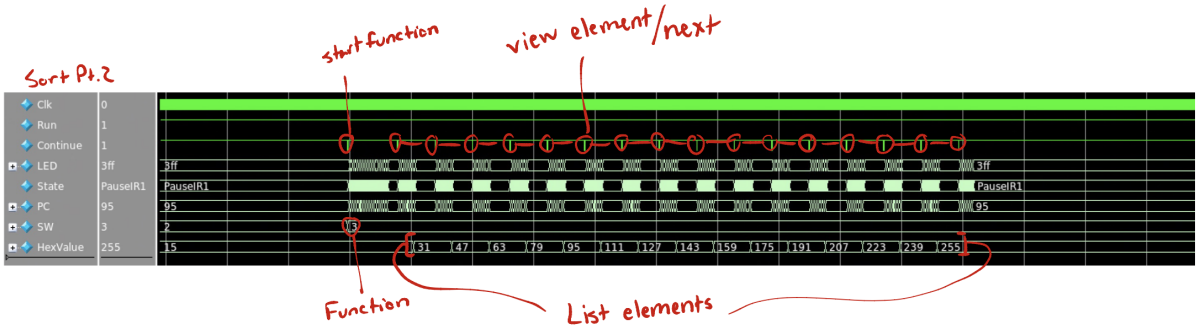


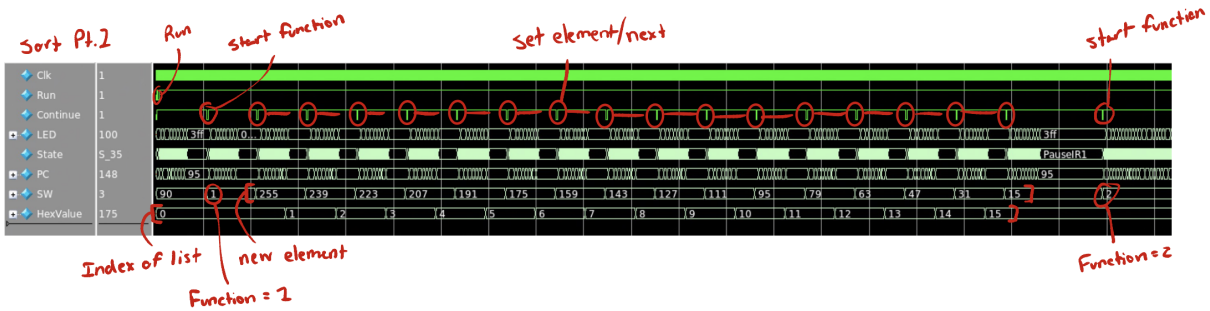








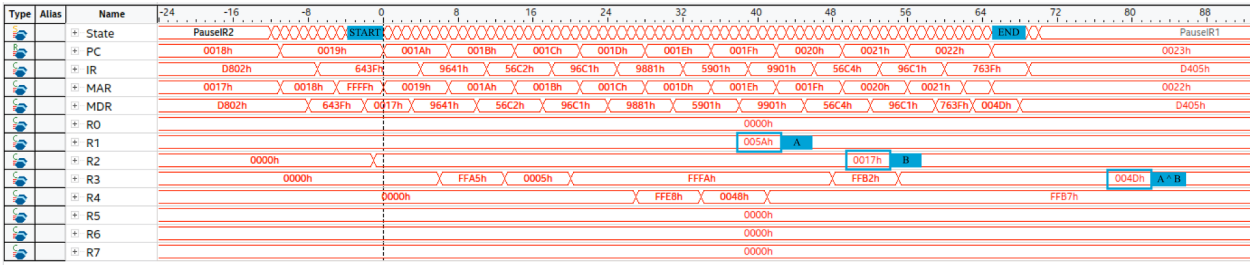




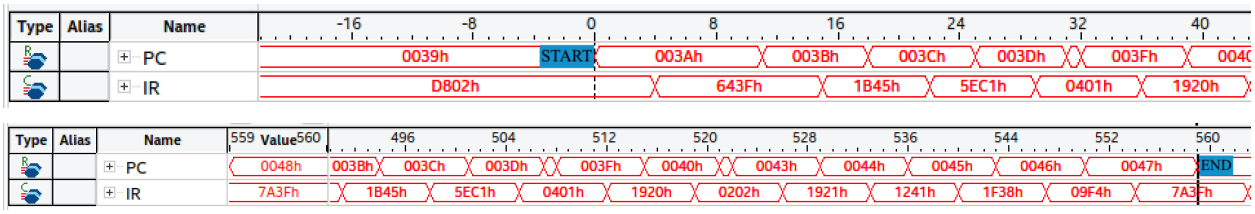
Extra Credit:

From the waveforms below, we can evaluate the performance of our CPU. The number of instructions per second that our CPU is capable of is a great metric for its performance. To calculate this, the number of instructions is divided by the number of cycles they take and that quantity is multiplied by the max frequency of the CPU. In order to count the number of instructions, we implemented a counter variable that we could monitor in SignalTap. This made it easy to keep track of exactly how many instructions were performed. As a start and end point, we used the trigger value of the PC and the PAUSE instruction. The XOR, multiply and sort programs operate at 9.55, 9.75 and 8.60 MIPS, respectively, and the 'average' is 9.30 MIPS. This number is helpful in comparing the performance of CPUs and estimating the runtime of a program.

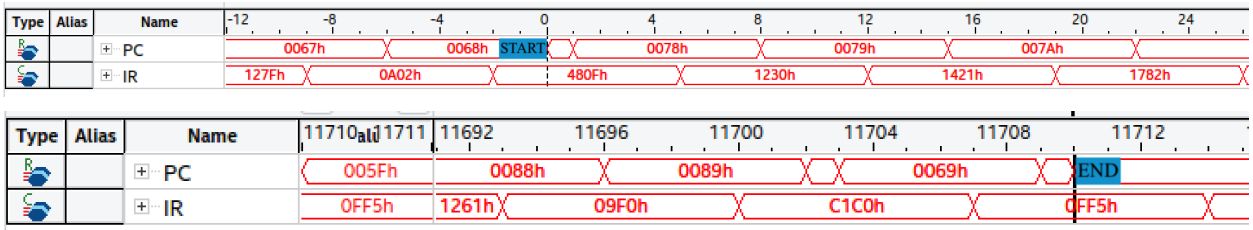
XOR:



Multiply:



Bubble Sort:



Post Lab:

Design Statistics and Resources:

LUT	1165
DSP	0
Memory (BRAM)	16384
Flip-Flop	266
Frequency	68.98 Mhz
Static Power	90.06 mW
Dynamic Power	23.07 mW
Total Power	124.79 mW

Conclusion:

Design Functionality

Overall, our approach to implementing the SLC-3 processor was simple, efficient, and easy to follow. We encountered no problems and finished the lab relatively quickly. Our addition to the provided skeleton consisted of the datapath module, the register file module, and the ALU module. Most of the implementation took place in the datapath module where we instantiated the ALU, the register file and created the necessary connections between the SLC-3's various components. The datapath communicated with the memory management unit (MEM2IO) and the control unit (ISDU) through the secondary top-level (slc3.sv). This partitioning of the SLC-3 simplified the implementation process and allowed for an efficient processor.

Lab Manual Reflection:

The lab manual was clear and detailed. I found the provided state diagram and the updated datapath very useful in implementing the SLC-3 processor. The instruction summary was also great to have as a refresher on the different operations that the LC-3 from ECE 120 could execute. Lastly, the I/O specification section and the description of MEM2IO were great. I cannot think of any specific ambiguous or incorrect parts of the lab manual. All in all, it served as a great guide to implementing the SLC-3.