ECE 385

Spring 2024

Experiment 5 – SLC-3

Lab 5 Report

Ziad AlDohaim (ziada2)
Mohammed Alnemer (mta8)

ECE 385

Lab 5 Report

CANVAS:

-2.0 for Graphical Block Diagram Annotations: Missing, need to show the main data path and control unit for each operation.

1. Introduction

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

The SLC-3 processor is designed as a simplified variant of the LC-3 ISA which we covered in both ECE 120 and 220. It follows the classic fetch-decode-execute cycle characteristic of computing architectures. The SLC-3 processor incorporates a 16-bit architecture, featuring a 16-bit Program Counter (PC), 16-bit instructions, and a 16-bit data path, supporting a variety of operations including arithmetic, logic, and control flow instructions such as ADD, AND, NOT, BR (branch), JMP (jump), and JSR (jump to subroutine). Memory operations are used through instructions like LDR (load register) and STR (store register), with the processor architecture comprising components such as the Instruction Register (IR), Memory Address Register (MAR), Memory Data Register (MDR), an 8x16 general-purpose register file, an Arithmetic Logic Unit (ALU), and a control unit to sort the execution of instructions based on decoded operation codes (opcodes).

This design allows SLC-3 to execute instructions that manipulate data and control program flow, interacting with memory and I/O devices (switches. Buttons. LEDs on FPGA). Instruction execution in the SLC-3 involves sequential stages where an instruction is fetched from memory, decoded to determine its type and the required operations, executed to perform the specified action, and then the cycle repeats for the next instruction. The inclusion of a set of condition codes (N, Z, P for negative, zero, and positive) allows for conditional execution of instructions.

b. Describe how the SLC-3 differs from the LC-3 processor you learned in ECE 120/220.

The SLC-3 is a simplified version of the LC-3 processor. LC-3 provides a good balance between simplicity and realistic features of a computer architecture, making it a common choice for introductory courses in computing systems, like ECE 120/220. While the SLC-3 shares many

core features with the LC-3, such as a 16-bit word size, a set of registers, and a similar instruction set, there are many differences such as,
The SLC-3 only implements a subset of the LC-3's instruction set, focusing on the most fundamental instructions. The control logic or state machine of the SLC-3 is simplified. For example, the LC-3 might include additional states or more complex branching logic within its state machine to handle a wider range of instructions. The SLC-3 simplifies the approach for dealing with interrupts and mainly focuses on the core execution cycle of fetch, decode, and execute. The SLC-3's datapath, which is the collection of hardware components responsible for executing instructions, is less complex, potentially lacking certain elements like advanced addressing modes or supplementary arithmetic units.

2. Written Description and Diagrams of SLC-3

a) Summary of Operation

FUNCTION	DESCRIPTION
ADD (Addition):	Adds the contents of two source registers, SR1 and SR2, and stores the result in the destination register, DR. Updates the condition codes (N, Z, P) in the status register based on the result.
ADDi (Add Immediate):	Adds the content of a source register, SR1, to an immediate value, imm5, which is sign-extended to 16 bits. The result is stored in the destination register, DR, with the status register updated accordingly.
AND (Logical AND):	Executes a bitwise AND operation between two source registers, SR1 and SR2. The result is stored in the destination register, DR, and the status register is updated to reflect the outcome.
ANDi (AND Immediate):	Performs a bitwise AND operation between the contents of a source register, SR, and a sign-extended immediate value, imm5. The result is stored in DR and the status register is updated.
NOT (Bitwise NOT):	Computes the bitwise complement (NOT) of the contents of a source register, SR, and stores the result in the destination register, DR. The status register is updated to indicate the result's sign.

BR (Branch):	Changes the program counter to a new address if the current condition codes (N, Z, P) match the specified condition in the instruction. Unconditional branching is possible by setting all condition codes (NZP) to '1'.
JMP (Jump):	Changes the program counter to the address contained in the base register, BaseR, redirecting program execution to that address, without conditions.
JSR (Jump to Subroutine):	Stores the current program counter in register R7 and then sets the program counter to a new address by adding a sign-extended immediate value, PCoffset11, facilitating subroutine calls.
LDR (Load Register):	Loads the destination register, DR, with the value from memory specified by adding a sign-extended offset, offset6, to the base register, BaseR. The status register is updated to reflect the loaded value.
STR (Store Register):	Stores the content of a source register, SR, into memory at an address calculated by adding a sign-extended offset, offset6, to the base register, BaseR.
PAUSE (Pause Execution):	Halts execution and waits for the 'Continue' signal. Execution can only proceed if 'Continue' is asserted during this instruction. When multiple PAUSE instructions are encountered in succession, each press of the 'Continue' button resolves only one PAUSE. The ledVect12 field can be used to output a user-defined pattern on the board's LEDs to indicate the execution status or position within the program.

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 processor works on the fetch-decode-execute cycle.

Fetch:
(what we worked on for 5.1)

The processor reads the next instruction to execute from memory. This is done by placing the content of the Program Counter (PC), which holds the address of the next instruction, onto the Memory Address Register (MAR). The instruction at the address specified by the MAR is then loaded into the Memory Data Register (MDR). The content of the MDR, which is the fetched instruction, is moved into the Instruction Register (IR) for decoding. The PC is incremented so that it points to the address of the subsequent instruction.

Decode:

The IR is analyzed to determine the type of the instruction that has been fetched. The opcode, which specifies the operation to be performed, is extracted and used to determine the subsequent actions to be taken. Depending on the opcode, other parts of the instruction, such as source and destination registers, immediate values, or offsets, are identified; the opcode can be seen below.

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

Figure 1: Table of available instructions and possible instructions of SLC-3 system

Execute:

The execution phase depends on the specific instruction that was decoded. For arithmetic operations like ADD or AND, the operands are fetched from the register file, the operation is

carried out by the Arithmetic Logic Unit (ALU), and the result is stored back in the destination register. Condition codes (N, Z, P) are set based on the result of the computation.

For memory access instructions like LDR (load) or STR (store), the ALU computes an effective address by adding the base register and the offset, and the data is either read from memory to the register file (in case of LDR) or written from the register to memory (in case of STR).

For control instructions like BR (branch), the processor determines whether to change the flow of execution by adding the offset to the PC if the specified condition codes are met.

For jump and subroutine call instructions like JMP and JSR, the PC is loaded with the address specified by the instruction or computed using the offset, facilitating jumps within the program or calls to subroutines.

The processor repeats this cycle for every instruction, thereby allowing the SLC-3 to perform a wide range of tasks programmatically. The SLC-3 can perform various instructions that are common in many instruction set architectures:

Through the fetch-decode-execute cycle, the SLC-3 carries out these instructions, showing the operation of this processor.

c) Block diagram of cpu.sv, which includes the main data-path and control unit for your CPU.

SV Modules

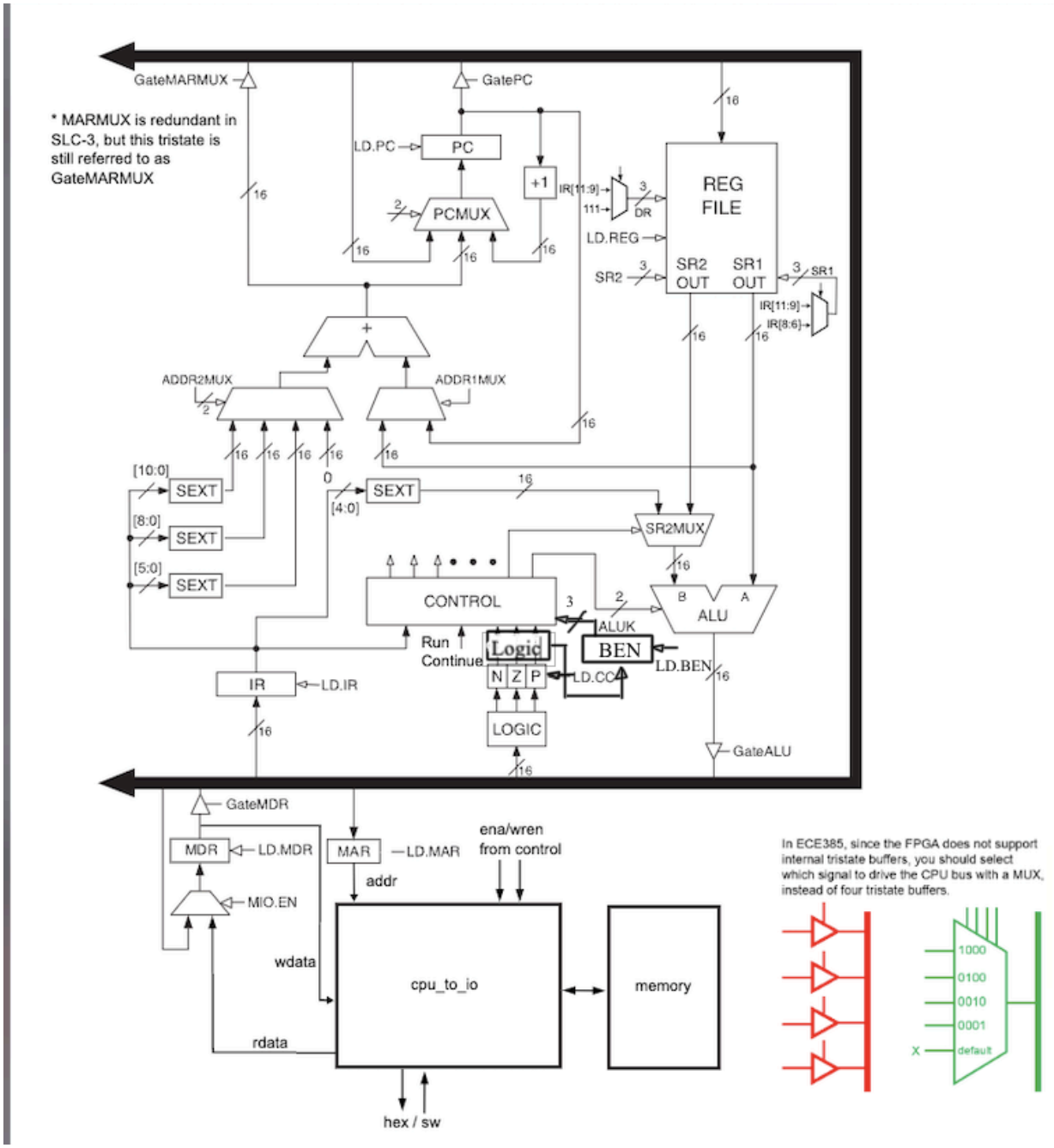

Figure 2: SLC3 BLOCK DIAGRAM

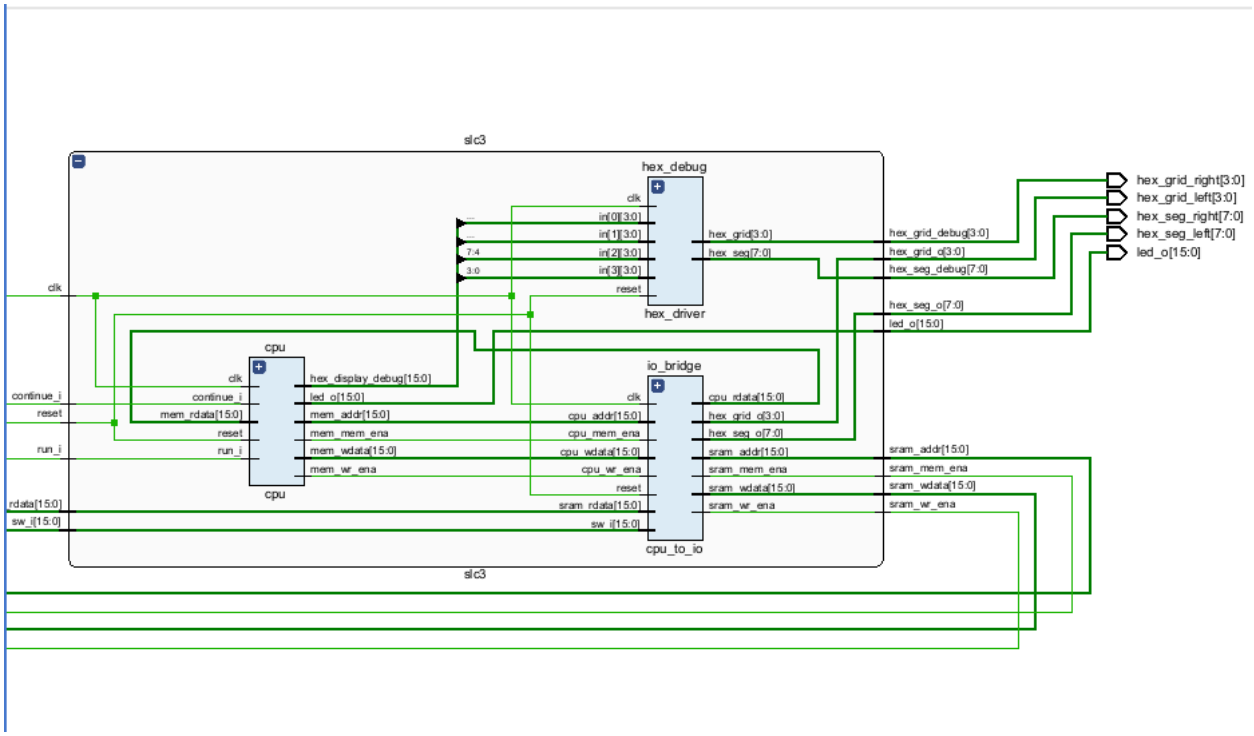

Figure 3: SLC3 TOP RTL Block Diagram (slc3.sv)

processor_top.sv

Inputs:

clk, reset, run_i, continue_i, sw_i[15:0]

Outputs:

led_o[15:0], hex_seg_left[7:0], hex_grid_left[3:0], hex_seg_right[7:0], hex_grid_right[3:0]

Description:

This is the top-level module of an SLC-3 processor system. It connects the CPU core to peripheral I/O devices and manages the overall synchronization and reset logic. The module processes input signals such as 'run_i' and 'continue_i' for controlling the CPU execution flow and reads switch inputs 'sw_i' for configuration or operation directives. The outputs include LED displays 'led_o' for visual status indication and two sets of hex display signals, 'hex_seg_left', 'hex_grid_left', 'hex_seg_right', and 'hex_grid_right', for hexadecimal data output typically used for debugging.

Purpose:

As the top-level, this integrates all the individual components of the SLC-3 processor, including CPU, memory, and I/O interfaces.

Changes:
This module was given to us, we did not have to change anything.

test_memory.sv
Inputs:
clk, reset, data[15:0], address[9:0], ena, wren

Outputs:
readout[15:0]

Description:
The 'test_memory' module emulates a memory component in a test environment. In other words, it's a dummy memory for the simulator to run.

Purpose:
This module is designed for simulation and testing purposes to copy hardware memory components within the simulation.
Changes:
Created for us, did not have to change anything.

cpu.sv
Inputs:
clk, reset, run_i, continue_i,[15:0] mem_rdata

Outputs:
hex_display_debug, led_o, [15:0]mem_wdata, [15:0] mem_addr, mem_mem_ena, mem_wr_ena

Description:
This module represents the central processing unit (CPU) of the SLC-3 processor. It orchestrates the fetch-decode-execute cycle of instructions and manages interactions between the CPU's internal registers, arithmetic logic unit (ALU), and memory. It takes input from the clock signal (clk) to progress through the instruction cycle, a reset signal to initialize the processor state, a run

signal (run_i) to start processing, and a continue signal (continue_i) for pausing and resuming execution. Memory read data is received through mem_rdata, and data to be written to memory, along with the corresponding address, enable and write signals, are output through mem_wdata, mem_addr, mem_mem_ena, and mem_wr_ena, respectively.

Purpose:
The CPU module is the heart of the SLC-3 processor design, managing all major operations such as instruction sequencing, execution, memory access, and I/O control. It enables the simulation of a simple computer architecture for educational purposes, demonstrating core concepts such as instruction flow, memory access, and computation.

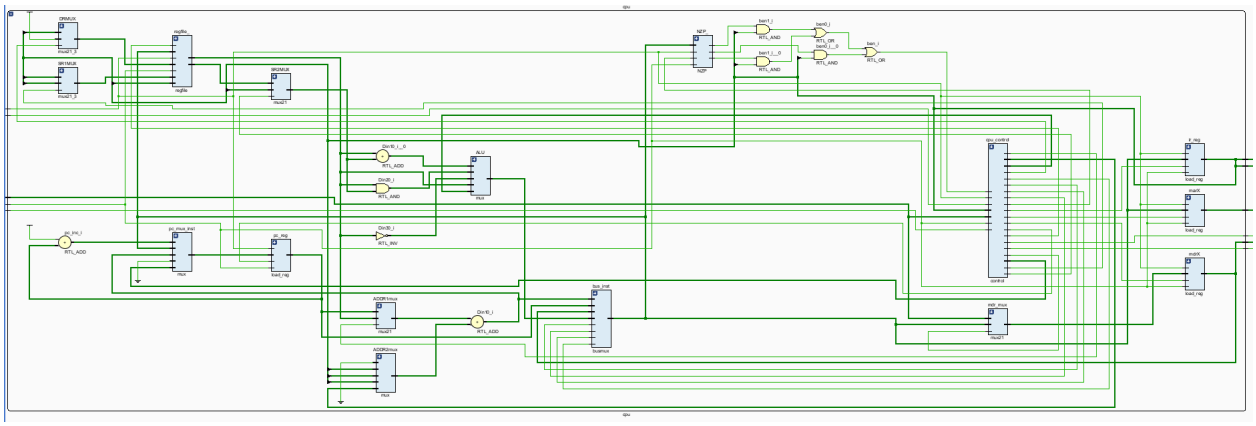

Figure 4: RTL design of CPU (cpu.sv)

Changes:
Created instantiations of almost everything, such as the muxes, register files, NZP, including the logic that goes with it. This file is our

Load_red.sv
Inputs:
clk, reset, load, data_i[DATA_WIDTH-1:0]

Outputs:
data_q[DATA_WIDTH-1:0]

Parameters:
DATA_WIDTH: This parameter defines the width of the data input and output. It must be set to match the size of the data the register is expected to handle.

Description:
This module is a generic loadable register that stores a value of 'DATA_WIDTH' bits. It captures the input data 'data_i' on the rising edge of the clock 'clk' when the load enable signal 'load' is asserted. If 'reset' is asserted, the register is cleared. The current value of the register is continuously output on 'data_q'.

Purpose:
The loadable register serves as a basic storage element in digital circuits, commonly used to hold intermediate values, states, or configuration data within a synchronous system. Its purpose is to provide stable data outputs that only change upon specific control signals, thereby enabling controlled data flow and state retention in sequential logic.

Changes:
Given to us.

regfile.sv
Inputs:
clk, bus[15:0], ld_reg, sr1sel[2:0], sr2sel[2:0]

Outputs:
Dout1[15:0], Dout2[15:0]

Description:
register file in the CPU, which is an array of registers. It uses the 'clk' signal to synchronize the loading of data into the registers. When the 'ld_reg' signal is active, the data present on the 'bus' is written into one of the registers selected by a register-select signal. The module provides two outputs, 'Dout1' and 'Dout2', which are the contents of the registers selected by 'sr1sel' and 'sr2sel. There is also a demux within the register file which is used to select which register to write to by decoding a binary value into the register’s enable.

Purpose:The register file serves as the primary storage of temporary data, allowing access to operands for arithmetic operations, holding intermediate values, and storing results.

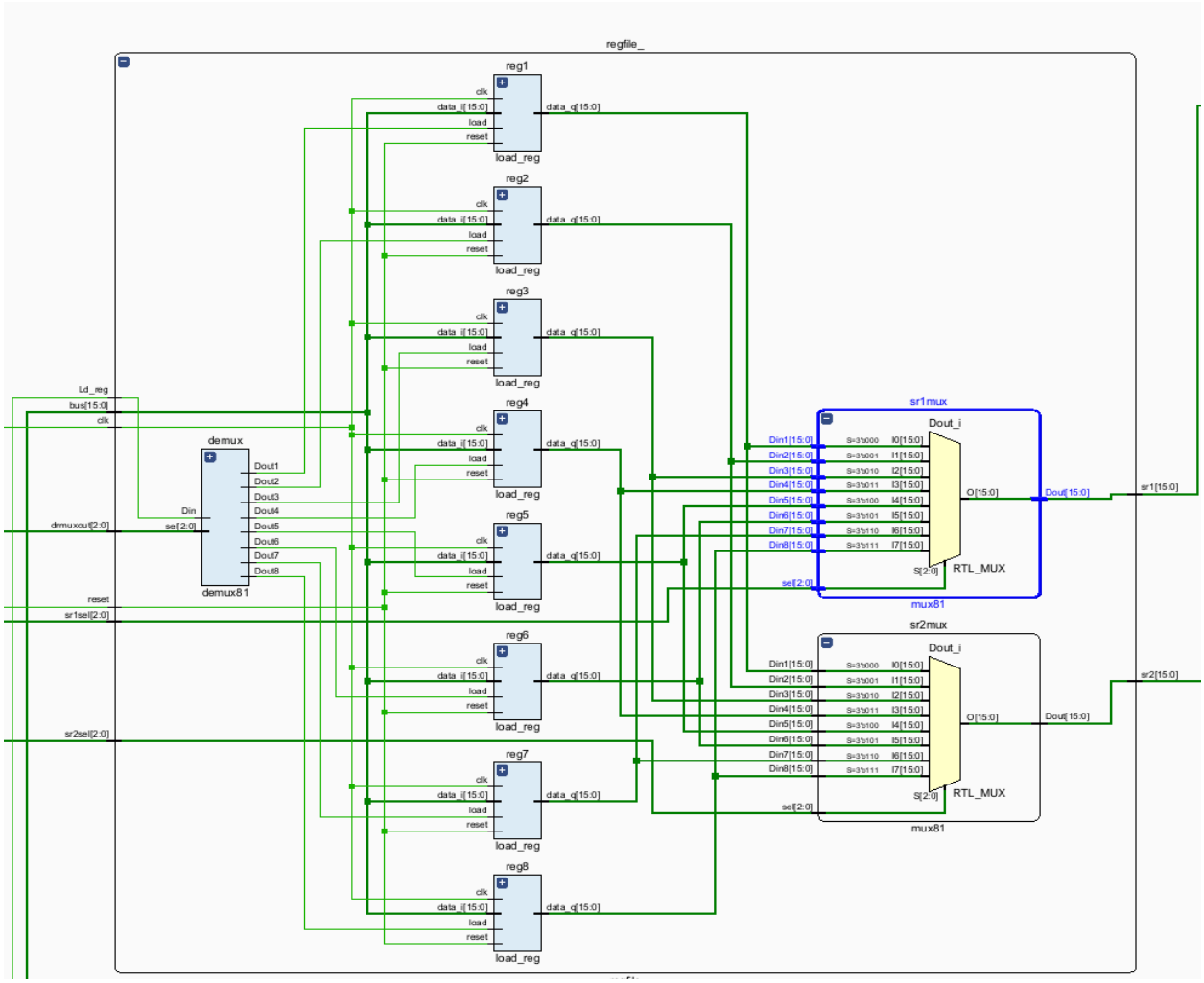

Figure 5: RTL design of Register File (Load_red.sv)

Changes:
Created from scratch.

mux81/mux/mux21_3
Input:
sel[2:0]/sel[1:0]/sel
Din1,Din2,Din3…DinX [15:0]/[15:0]/[2:0]
Output:
Dout[15:0]/Dout[15:0]/Dout[2:0]

Description:
This module is a X-to-1 multiplexer that selects one of the X 16-bit data inputs (Din1, Din2,.. DinX) to pass through to the output (Dout), based on the X-bit select signal (sel). The select input determines which of the data inputs is connected to the output, for a 4-to-1 MUX: '00' selects Din1, '01' selects Din2, '10' selects Din3, and '11' selects Din4. This is expanded based on the number of inputs and selects.

Purpose:
The purpose of a multiplexer is to perform data selection and routing. It allows for selection of data.

Changes:
We made this module from scratch

busmux
Inputs:
Din1[15:0], Din2[15:0], Din3[15:0], Din4[15:0], sel1, sel2, sel3, sel4

Outputs:
Dout[15:0]

Description:
This module is an extended bus multiplexer that selects one of four 16-bit data inputs to pass to the output. Each data input (Din1 to Din4) has an associated select signal (sel1 to sel4). This allows you to simply select which input should be selected. We have conditions to ensure that if more than one is high then simply go to zero, this is an implementation of a bus, shown below.

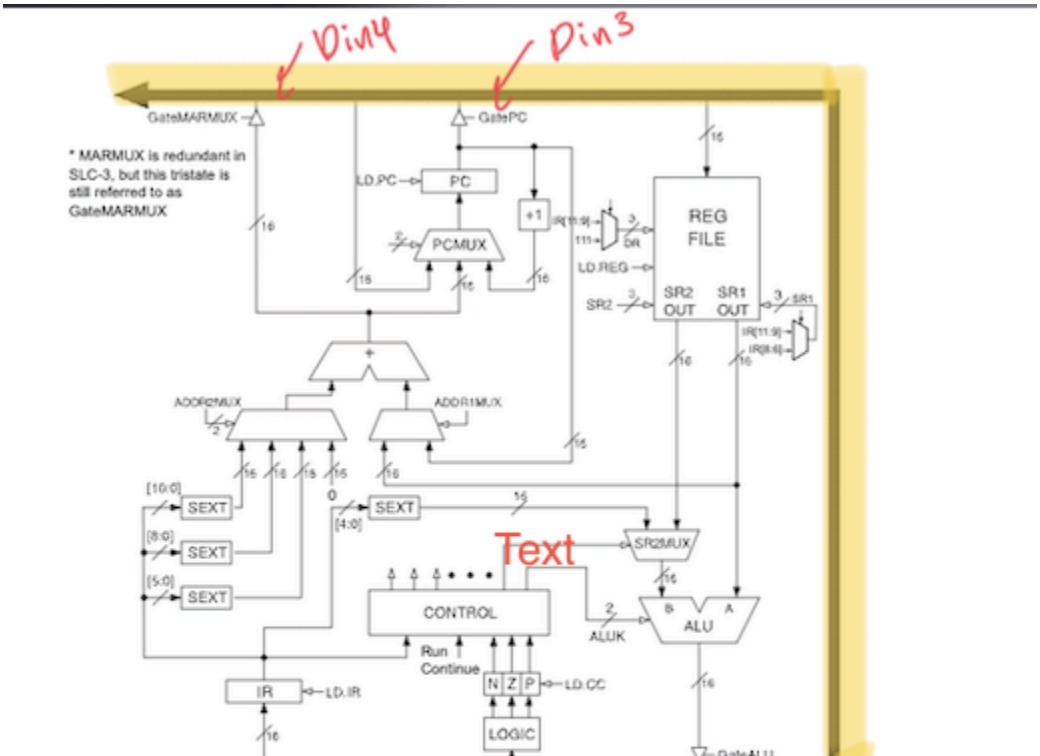

Figure 6: BUSMUX in the Old Datapath for clarification only, not to demonstrate actual data path

Purpose:
The busmux routes data from multiple sources to a single destination, as seen above, based on control signals. To implement this in the FPGA we had to use muxes instead of tristate buffers since the FPGA does not support it.

Changes:
Created this module from scratch, it's simply a mux with 4 select inputs.

hex_driver.sv

Inputs:
Clk, reset, [3:0] in[4],
Outputs:
hex_grid[3:0] and hex_seg[7:0]

Description:
This module takes in an 4 or 8 bit sequence and formats it to be displayed on the hexadecimal LEDs to show the values in hexadecimal.

Purpose: This functionality is particularly useful in debugging and demonstration contexts, where quick and clear visibility of system states or values is necessary.

Changes:
We did not have to change anything.

sync.sv
Input:
Clk, d,
Output**:**
Q

Description:
The purpose of the Synchronizers.sv module is to provide a reliable mechanism for handling signals that cross between different clock domains within a digital system.
Purpose:
By synchronizing these signals to the system's clock, the module helps to prevent metastability, ensuring that the system's operation remains stable and predictable.

Changes:
We did not change anything, taken from previous labs.

cpu_to_io.sv
Inputs:
clk, reset, cpu_addr[15:0], cpu_mem_ena, cpu_wr_ena, cpu_wdata[15:0], sram_rdata[15:0], sw_i[15:0]

Outputs:
cpu_rdata[15:0], sram_addr[15:0], sram_mem_ena, sram_wr_ena, sram_wdata[15:0], hex_grid_o[3:0], hex_seg_o[7:0]

Description:
The 'cpu_to_io' module serves as an interface between the CPU and the I/O devices, including both the on-chip SRAM and the external I/O such as switches and hex displays. It switches between CPU memory access requests and I/O operations. When the CPU addresses the I/O area cpu_to_io redirects the data to or from the appropriate I/O devices.
Purpose:
The primary function of this module is to allow the CPU and various I/O peripherals to communicate, as well as manage memory reads/writes to the SRAM. It ensures that I/O operations do not conflict with memory operations and that data is correctly routed based on the CPU's requests.

Changes:
Given to us, did not have to modify.

control.sv

Inputs:
clk, reset, ir[15:0], ben, mem_rdata[15:0], continue_i, run_i

Outputs:
ld_mar, ld_mdr, ld_ir, ld_ben, ld_cc, ld_reg, ld_pc, ld_led, gate_pc, gate_mdr, gate_alu, gate_marmux, pcmux[1:0], drmux, sr1mux, sr2mux, addr1mux, addr2mux[1:0], aluk[1:0], mio_en, mem_mem_ena, mem_wr_ena

Description:
The 'control' module acts as the central control unit of the SLC-3 processor, decoding instructions and generating signals that orchestrate data flow and operation execution within the CPU. It reads the current instruction from the Instruction Register (ir) and produces control signals that enable or disable various data paths and components in the system. These signals include enabling loads to registers like MAR, MDR, IR, and controlling gate signals that determine which data paths are active for data to flow through. It also determines the function of the ALU and the behavior of the PC, through multiplexers like pcmux.

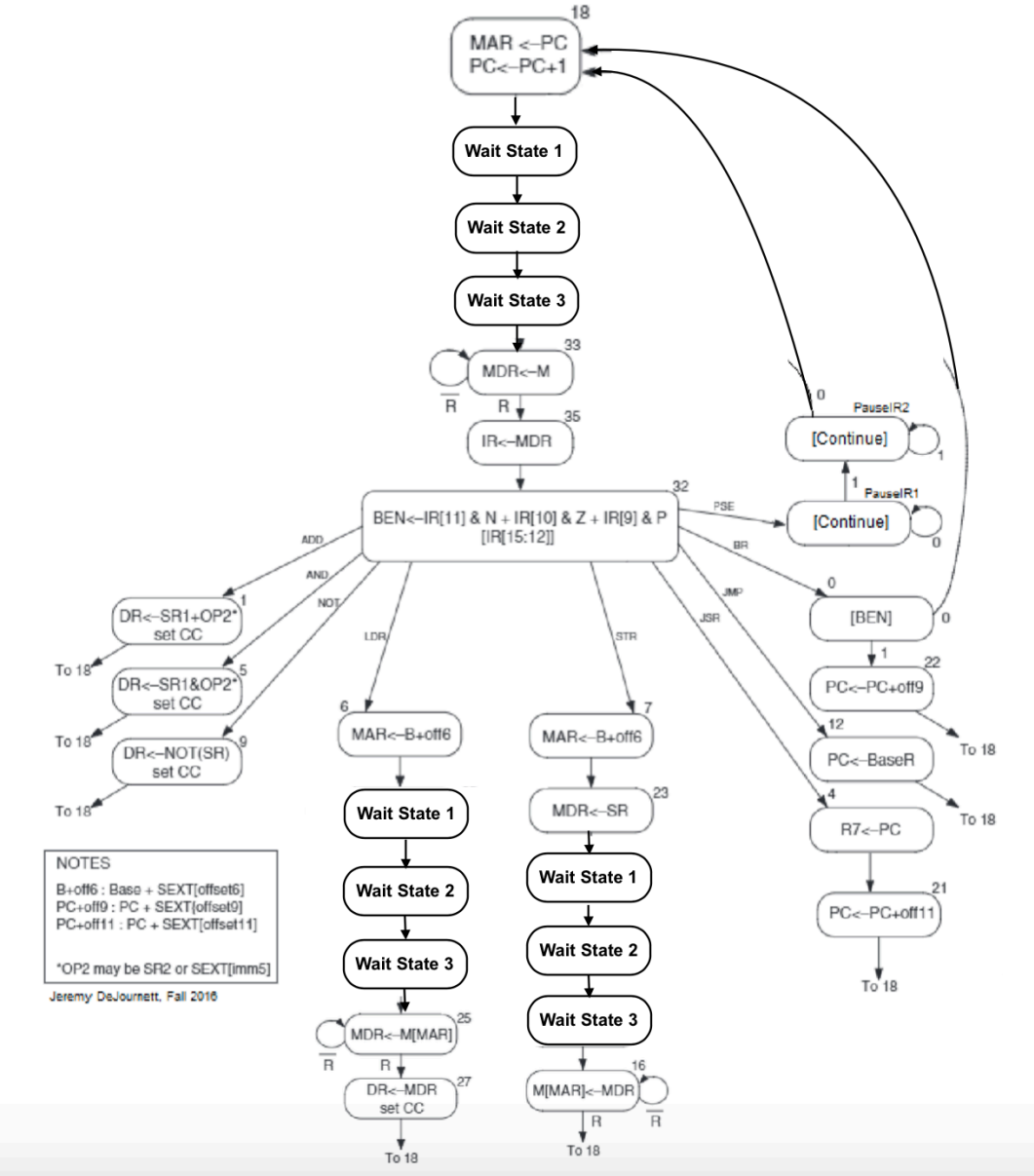

Figure 7: Updated State Diagram for SLC-3.

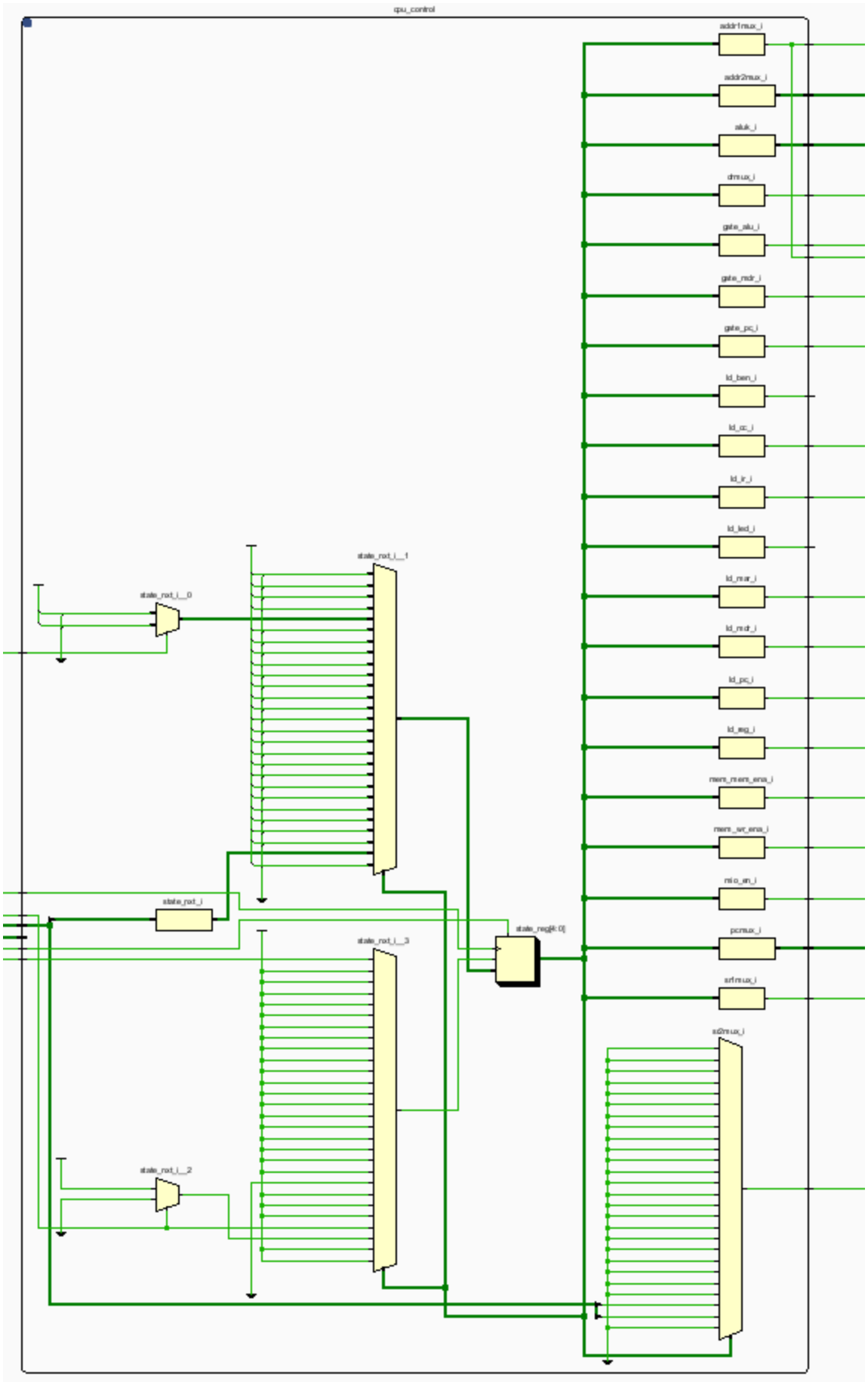
cpu_control
addr1mux_i
addr2mux_i
aluk_i
drmux_i
gate_alu_i
gate_mdr_i
gate_pc_i
ld_ben_i
ld_cc_i
ld_ir_i
ld_led_i
ld_mar_i
ld_mdr_i
ld_pc_i
ld_reg_i
mem_mem_ena_i
mem_wr_ena_i
mio_en_i
pcmux_i
srf mux_i
sr2mux_i
state_nxt_i__1
state_nxt_i__0
state_nxt_i
state_reg[4:0]
state_nxt_i__3
state_nxt_i__2

Figure 8: RTL SYNTHESIS of control.sv

Purpose:
This module is designed to implement the control logic of the SLC-3 processor, translating opcodes and status signals into control signals that direct the operation of the CPU at every clock cycle. It ensures that each instruction is executed correctly, managing the state machine that oversees the fetch, decode, and execute cycles, and handling conditions for branching, memory access, and I/O operations.

Changes:
Mostly given to us, which the professor did not mean to, we only had to add mio_en.

NZP.sv
Inputs:
clk, reset, bus[15:0], ld_cc

Outputs:
N, Z, P

Description:
The NZP module monitors the value present on the bus and updates the condition code flags N (negative), Z (zero), and P (positive). On the rising edge of the 'clk' signal, if the 'ld_cc' (load condition codes) input is asserted, the module evaluates the 16-bit value on the 'bus'. If the value is negative, the N flag is set to high and Z, P flags are set to low. If the value is zero, the Z flag is set to high and N, P flags are set to low. If the value is positive, the P flag is set to high and N, Z flags are set to low. The 'reset', i.e. ‘1’ input resets the condition code flags to zero.

Purpose:
The purpose of the 'NZP' module is to set the processor's condition codes, which are used to make decisions during branch instructions and other conditional operations. It's a crucial component for the execution of instructions that require conditional logic based on the results of arithmetic operations or data status.

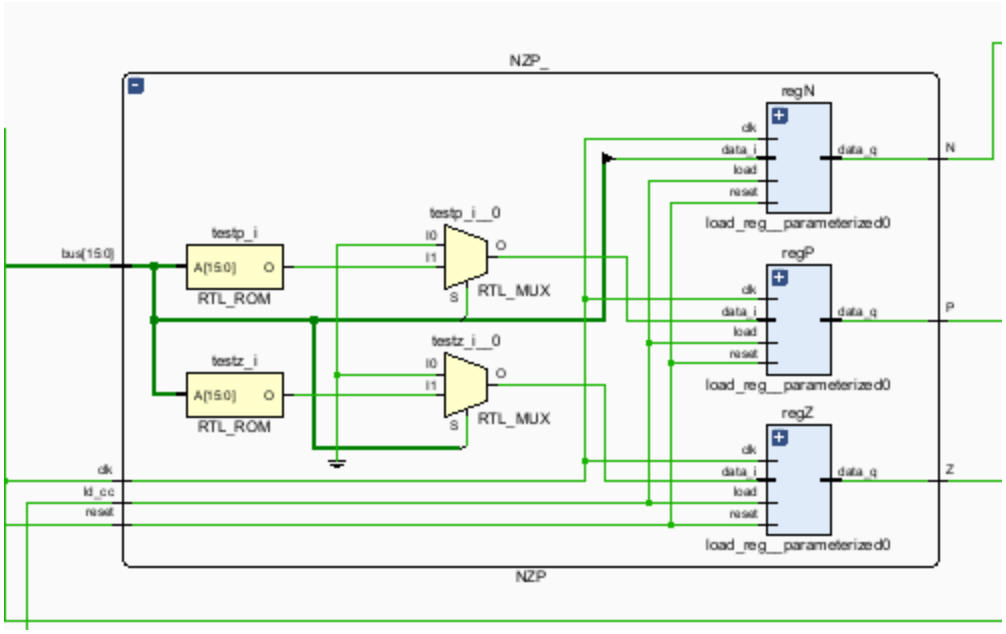

Figure 9: RTL Synthesis of NZP module.

Changes:
Created from Scratch.

d. Like we described in class, annotate graphically on the block diagram how each different execution stages operate. This will likely be multiple versions of the block diagram from 2.c with separate annotations.

e. Block diagram of the top-level (processor_top.sv), which includes connections between the cpu, as well as the cpu_to_io, memory, and other modules.

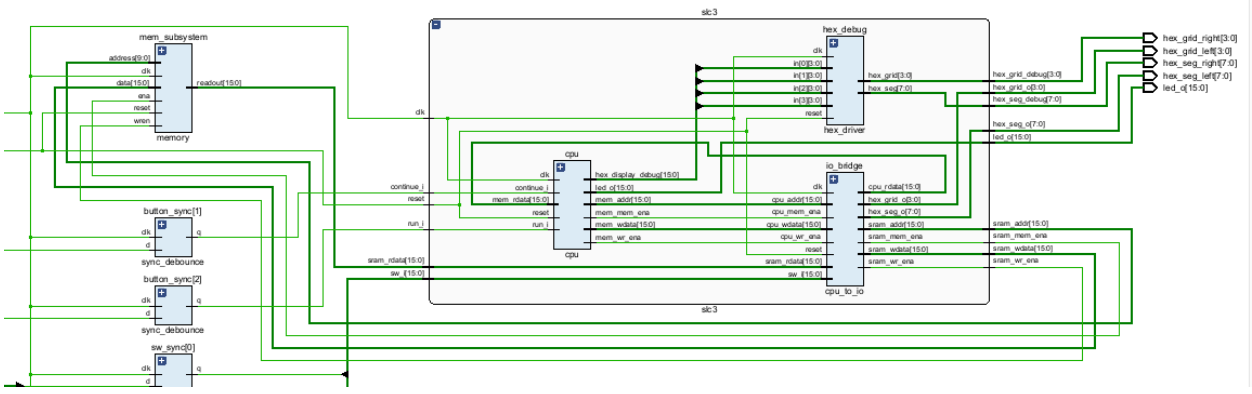

Figure 9: RTL Synthesis of top-level (processor_top.sv)

3. Simulations of SLC-3 Instructions

a. Simulate the completion of all 7 test programs, I/O Test 1, I/O Test 2, Self-Modifying Code, XOR, Multiplier, Auto Count, and Sort.

b. Annotations for the above simulations should include at a minimum: start of the test program, any user input (for example, entering the numbers in the multiplier test), and reading the expected result.

c. Note that for simulation traces which are long, you may truncate out the intermediate portions of the program. For example, you will likely need to do this so that your Sort test simulation is legible.

TEST 1: I/O Test 1: x0003

Figure 10: This I/O Test continuously refreshes the the display values to represent the current value of the switches. In the diagram above, sw_i[15:0] represents the value of the switches and he_display is the display value. When there is a switch change, the display value changes to become that valeu that was switched to.

Test 2: I/O Test 2: x0006

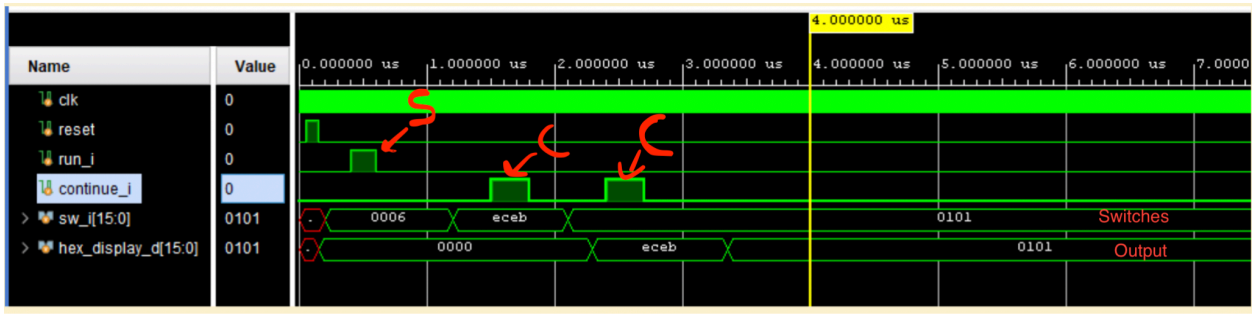

Figure 11: I/O Test 2 is very similar to the first test however it will only change the value of the display value once the continue button is at a high. In the figure above, “S” represents the start of the program, and “C” represents the instant in which “continue” is pressed. We can see that once continue is pressed the value of “hex_display” changes to the value of the switches.

Test 3: Self-Modifying Code Test: x000b

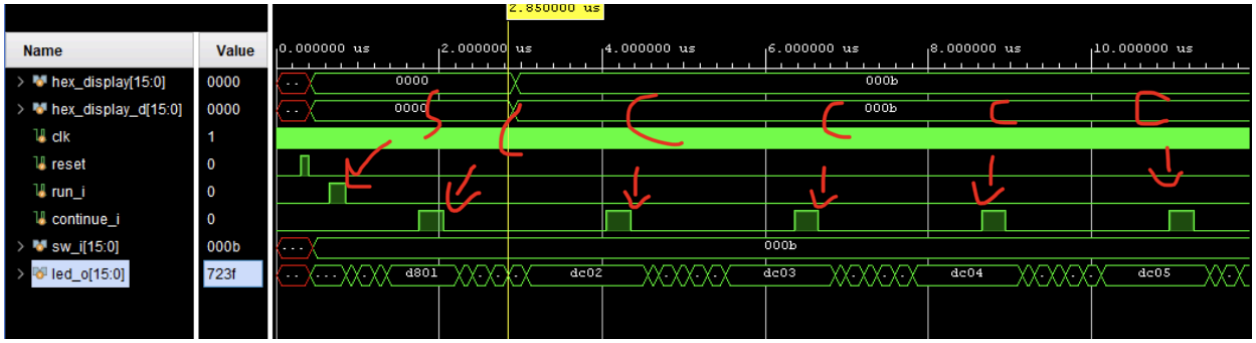

Figure 12: The self-modifying code increments the value of the IR each time continue is pressed. In the diagram "S" represents the start of the program and "C" represents times when continue is high. The "led_o[15:0]" is the value of IR. You can see that everytime continue is pressed the value of IR increments by one.

Test 4: AutoCount Test: x009C

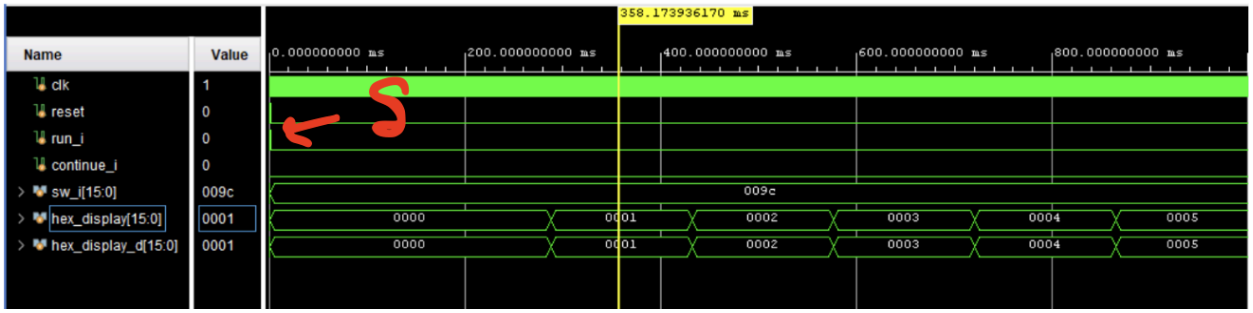

Figure 13: The Auto count will automatically start a counter in the hex display starting from zero without any extra inputs. As soon as we press run, you can see the "hex_display" start incrementing from zero. The "S" represents the start of the program.

Test 5: XOR test: x0014

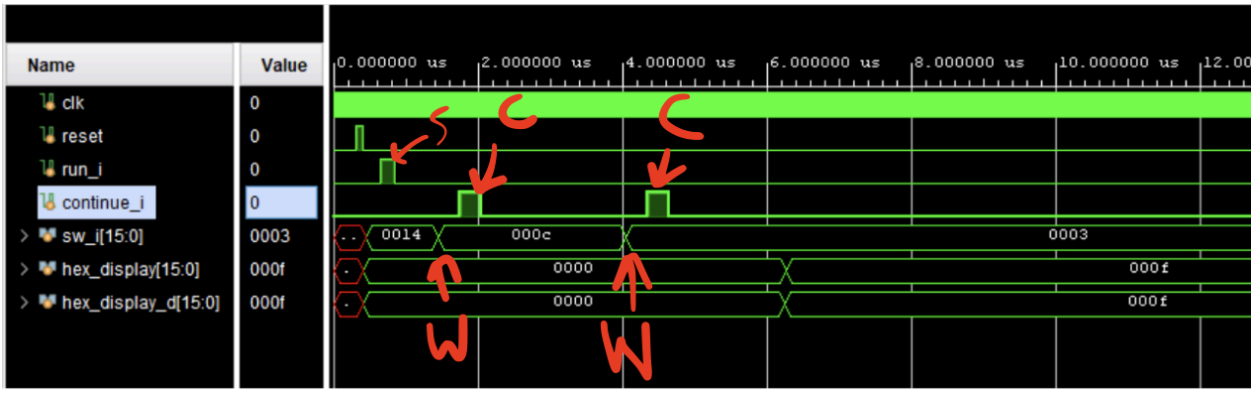

Figure 14: The XOR test will perform the logical XOR operation to two values that it stores from the switches. In the diagram "S" represents the start of the program, "C" represents when continue is high, and "W" represents when we change the value of the switches. In the above

program, we first set the first value to x000c and press continue to store. Then we proceeded to change the value of the switches to x0003 and when we pressed continue the program XOR the two values displayed the value in the hex_display.

Test 6: Multiplication Test: x0031

Figure 15: The multiplication test, similar to the xor test will take in two values but instead of xoring them, it will multiply the two values. In the diagram “S” represents the start of the program, “C” represents when continue is high, and “W” represents when we change the value of the switches. The program first sets the switch values to 2 and then presses continue to store it. Then the program sets the switch value to 4 and the after continue is pressed, the program multiplies to the two values and displays the answer 8 to the hex_display.

Test 7: Bubble Sort: x005a

Unsorted

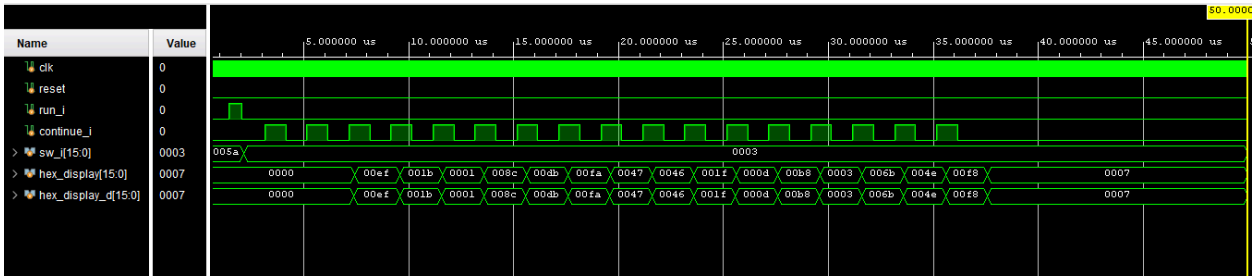

Figure 16: The Sorting Test showcases the sorting of values specified in the lab manual's table. By inputting x0002, the 'sort' function is activated, sorting the list without showing the results on-screen. Following up with x0003 triggers the 'display' function, which, if executed after the sort command, reveals the ordered list on the display, allowing you to view each sorted value by pressing 'continue' (continue is displayed with the ‘continue_i’ wave). Inputting any other value will just cycle the menu back to the beginning.

After Sort

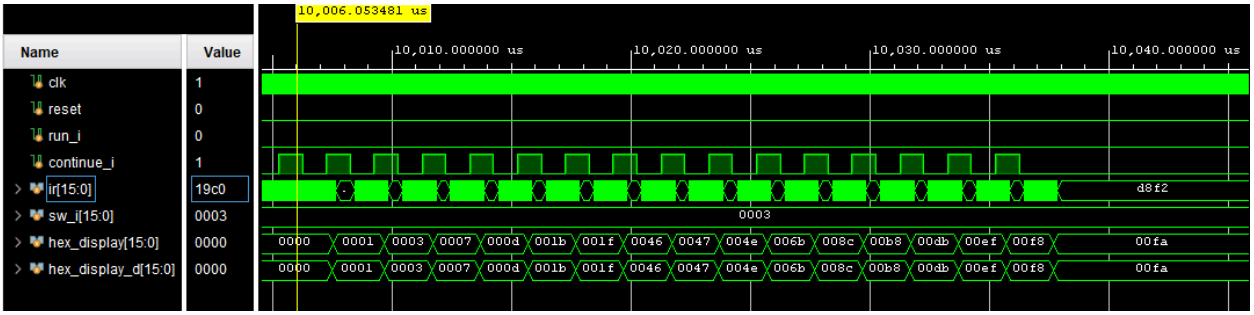

Figure 17: Waveform showing the aftersort.

4. Post-Lab Questions

Design Resources and Statistics table

LUT	408
dsp	0
BRAM	0.5
Flip-Flops	339
Latches	0
Frequency	97.34 Mhz
Static Power	0.071 W
Dynamic Power	0.013 W
Total Power	0.084 W

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

The cpu_to_io module bridges the CPU with external resources like memory and input/output (I/O) devices. It operates based on memory addresses, distinguishing between operations targeting the system's memory and those intended for I/O devices.

• What is the difference between BR and JMP instructions?

The Branch Instruction (BR) is conditional, it tests the condition codes for N,Z,P set by previous operations and based on that decided to branch to a new location or not.

The branching works by adding a sign-extended offset (PCoffset9) to the program counter if the condition is met (N,Z,P). This means that the next instruction is somewhere else in the program.

The Jump Instruction (JMP) is unconditional. It simply causes the program to ‘jump’ to a specified address and continue execution.

The Jumping works by changing the PC value.

The key difference is that BR is conditional and used for branching within code based on conditions, while JMP is unconditional and used for jumping to a new code location regardless of any conditions. BR decides where to go next based on the outcome of previous instructions, while JMP directly goes to an address specified by the contents of a register.

• 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 impact does this have on performance?

R is the ready bit, it indicates when a memory operation (read or write) has been completed.

To compensate for the lack of the signal in our design, we extended specific states in the state machine to accommodate the memory's response time without explicit acknowledgment. Specifically, we’ve adjusted the design to wait for 3 additional clock cycles in State_33 in-memory operations. This approach ensures that memory access will be complete within a fixed number of clock cycles, effectively hardcoding a delay to allow memory operations to be completed.

This impacts performance because since we add more clock cycles, each memory operation takes longer thus making the processor perform slower. However, eliminating the need for an 'R' signal simplifies the interface between the memory and the processor, reducing the hardware complexity and the cost of implementation.

5. Conclusion

The design and implementation of the SLC-3 processor largely met our objectives. However, we faced challenges with the multiplexers, where signal routing errors occasionally led to incorrect outputs. Future improvements should focus on the planning of signal pathways and enhancing our test bench strategy to detect such issues earlier in the development process.

The lab manual was particularly effective in detailing the control unit's operations, which significantly aided our understanding of the processor's functionality. We mainly encountered difficulties with the multiplexers, often struggling to know the correct signal output. However, the control unit's organized structure helped us in debugging and finding the correct signals.