ECE 385

Spring 2024

Experiment #5

Lab 5

Yuxuan Li yuxuan43

Yanghonghui Chen yc47

Section XC

TA: Shixin Chen

1. Introduction

(1) The basic functionality of the SLC-3 processor

The SLC-3 processor is a 16-bit processor with 16-bit PC, 16-bit instructions, and 16-bit registers. The system consists of three main components: central processing unit, the memory that stores instructions and data, and the input/output interface that communicates with external devices. When the processor is running, the computer will first fetch instruction from memory block, then decode to decide which type of instruction it is. The processor then does the execution and do fetching again to start a new cycle.

(2) How the SLC-3 differs from the LC-3 processor you learned in ECE 120/220?

The SLC-3 system is a simplified version of LC-3 processor with less instructions and components. Meanwhile, the SLC-3 system doesn't have R signal that LC-3 system uses to control waiting state, so we do the waiting part by adding extra states. Moreover, we add an extra state name Pause to check and wait for instructions after bench.

2. Written Description and diagram of SLC-3

(1) Summary of operation

The SLC-3 processor system goes through a cycle of fetch-decode-execute. Instructions that should be used are stored in memory subsystem and decoded to be sent to processing unit to execute. The system used a large number of FSM to implement different instructions under different situations. After fetching, processor decode the instruction and use operation code to decide which state it should go to.

(2) Describe in words how the SLC-3 performs its functions.

SLC-3 performs its function using Fetch, Decode and Execute operations. SLC-3 processor consists of CPU, CPU_TO_IO and a Hex Driver block. CPU_TO_IO block acts as the intersection between CPU and physical memory on FPGA board, while Hex driver block maintain the functionality of LED displays on board. Main functionality of processor is achieved by CPU block. The finite states machines are set within CPU block to control operation of processor. Different components in CPU such as ALU, Registers, MAR, MDR and MUXs are connected by CPU's bus, which receives and passes control signals between components. At the beginning of operation, the processor begins at Halt state, then goes through reset state and wait for the Run signal set by user. When Run signal is passed in, the processor will go to Fetch operation under the control of FSM.

In Fetch operation, system loads PC to MAR through the CPU bus on receiving signals coming from FSM. That is, MAR now contains the memory address to read the instruction from. After that, it increments PC, and loads instruction read from memory specified by MAR into MDR. This loading

process needs R signal in Patt and Patel structure, so we used three extra waiting states in finite state machine to do synchronization. Then it loads instruction from MDR to IR through CPU bus. During this operating process, the data path which is CPU bus has played and important role passing signals between different module in CPU, like MAR, MDR, IR and PC. Also, the CPU_TO_IO block is needed when Fetch operation needs to read data using address stored in MAR into MDR, since interaction between control block and physical memory is needed.

In Decode operation, CPU decide which state in FSM it should transfer to by analyzing instruction stored in IR. This operation is specified as state 32 in control unit. It used the 10^{th} to 12^{th} bit in instruction register to do AND operation with NZP bits stored in register, then decide which state to branch to.

The Execute operation do different things depends on which state the FSM is at. Almost every execution state has something in common, for example loading signals from main bus, do operations like adding in ALU, accessing register unit or read from/ write to memory block. Different functionality also can have different state transaction, some may requires to go to pause state, but eventually they all go back to state 18 and start over again to run the next line of instruction.

Instructions the processor can perform:

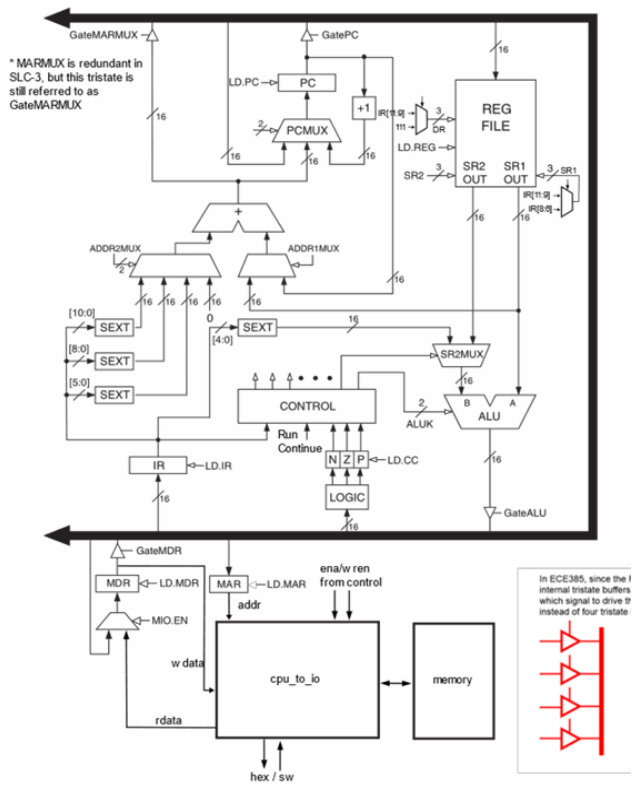
- Opcode: 0001 ADD
  The processor can add value which store in two registers or adding value in register with offset given in instruction. It updates status register by loading CC signal.
- Opcode: 0101 AND
  The processor does logical AND operation using value stored in two registers or AND value in register with offset given in instruction. It updates status register by loading CC signal.
- Opcode: 1001 NOT
  The processor negates SR and stores the negated result into DR. It updates status register by loading CC signal.
- Opcode: 0000 BR
  This operation uses status register that can be modified by LD.CC. If condition codes match the condition stored in the status registers(also mentioned as NZP), take the branch by adding 9-bits offset in instruction to current PC. Otherwise, continues execution. Thus, we can set NZP to 111 to perform an unconditional jump.
- Opcode:1100 JMP
  Copies memory address stored in BaseR specified in instruction into PC, so that it can do unconditional jump.
- Opcode: 0100 JSR
  Stores current PC into Register 7 in register unit, then adds PCoffset11(sign extended) in IR to PC. Usually used when jumping to subroutines.

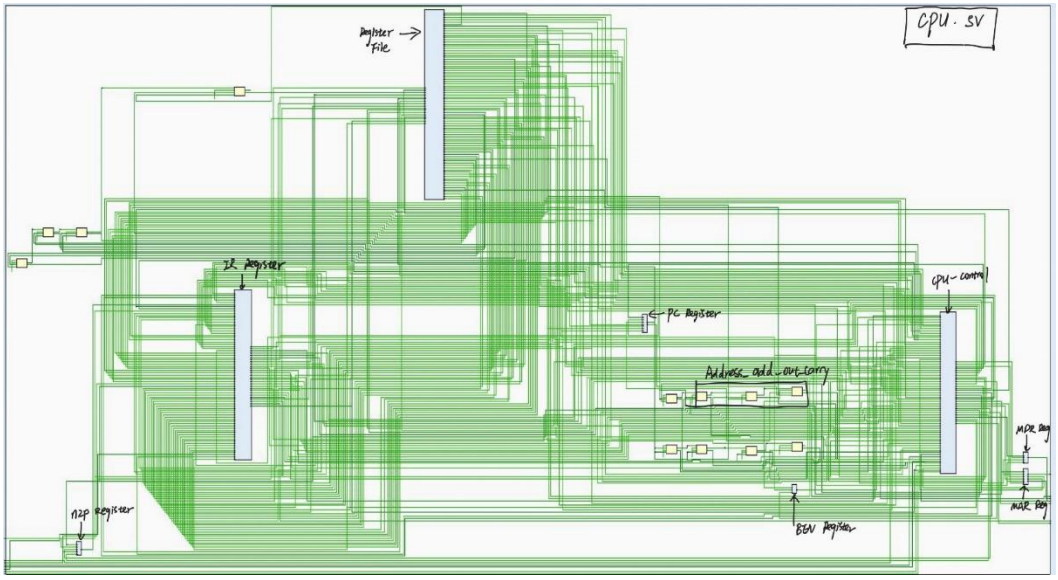
- Opcode: 0110 LDR
  Load using Register unit and 6-bits offset addressing. Loads memory contents in the address of BaseR + SEXT (offset6) into DR. It updates status register by loading CC signal.
- Opcode: 0111 STR
  Store using Register unit and 6-bits offset addressing. Stores the contents within SR at the memory location specified by BaseR + SEXT(offset6).
- Opcode: 1101 PAUSE
  Pauses the Execution operation until the Continue signal is detected. Execution should only relieve Pause state if Continue signal is asserted during the current pause instruction. When there exist multiple Pause states, only one should be erased if only one Continue signal occurs. Control signal to load LED is updated during pause state.

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

Table 1: The SLC-3.2 ISA

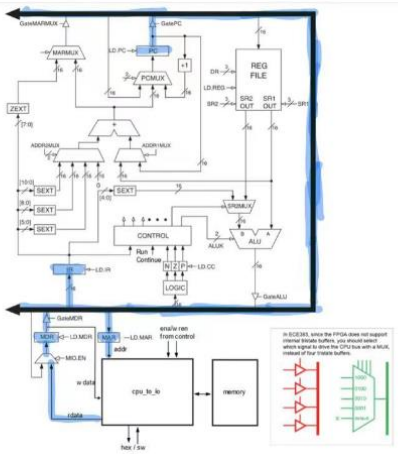
(3) Block diagram of cpu.sv



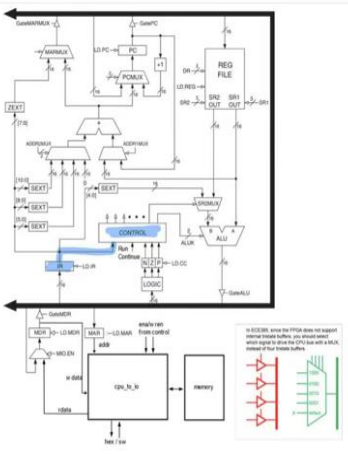


(4) Annotation on the block diagram

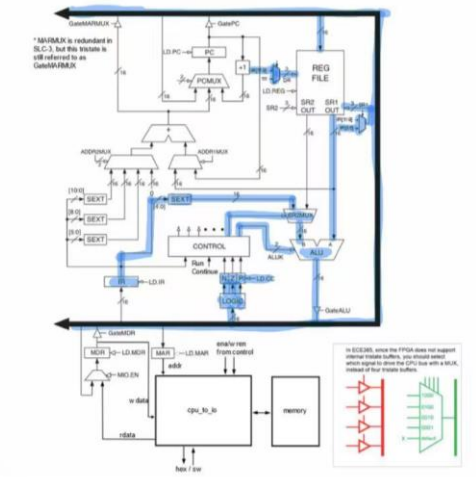
Fetch



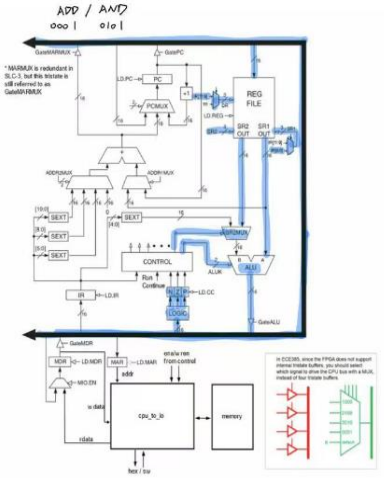
Decode

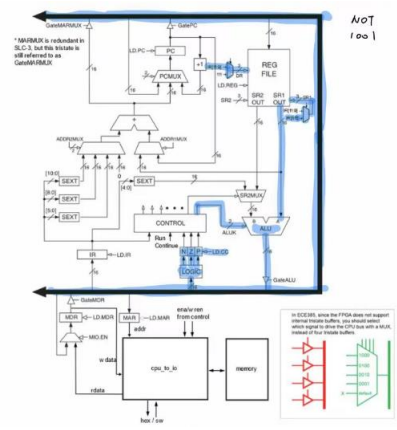


ADDi / ANDi
0001 0101

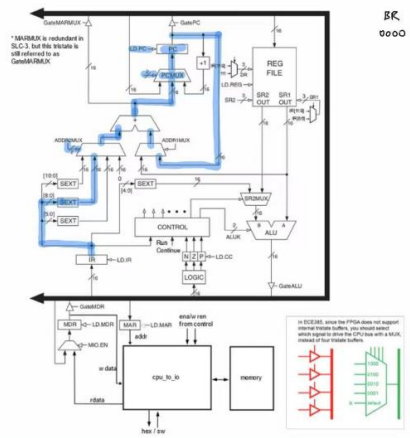


ADD / AND
0001 0101



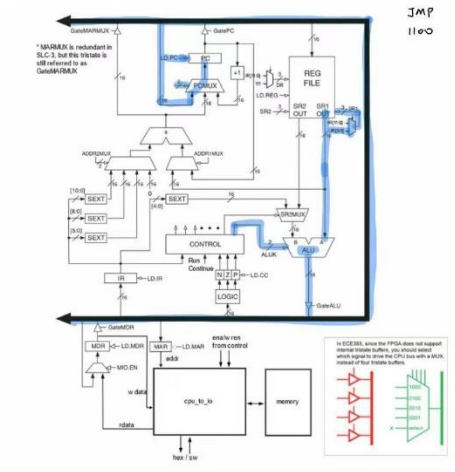


NOT
1001

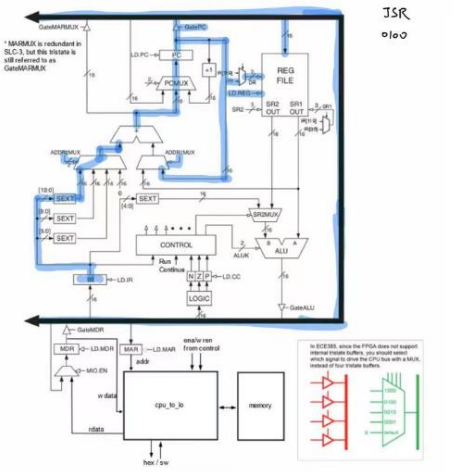


BR
0000

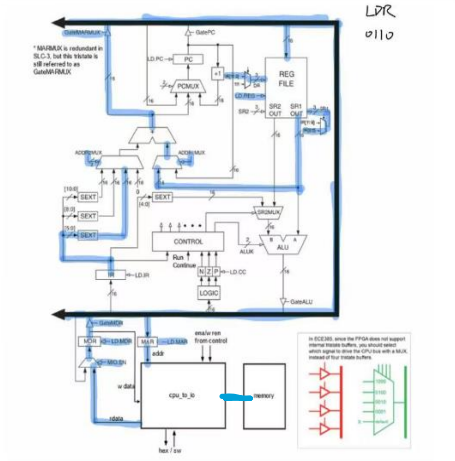
JMP
1100



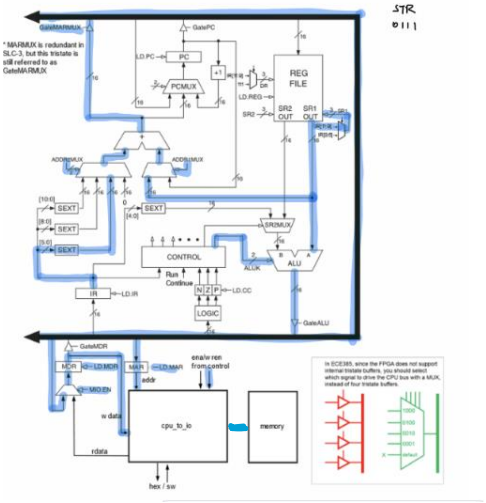
JSR
0100



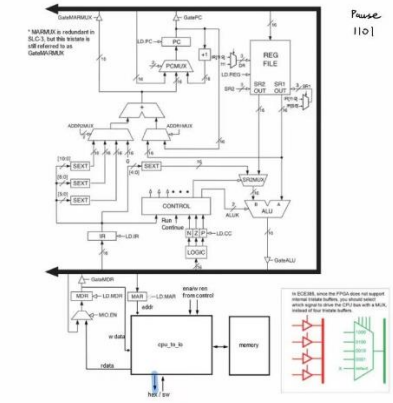
LDR
0110



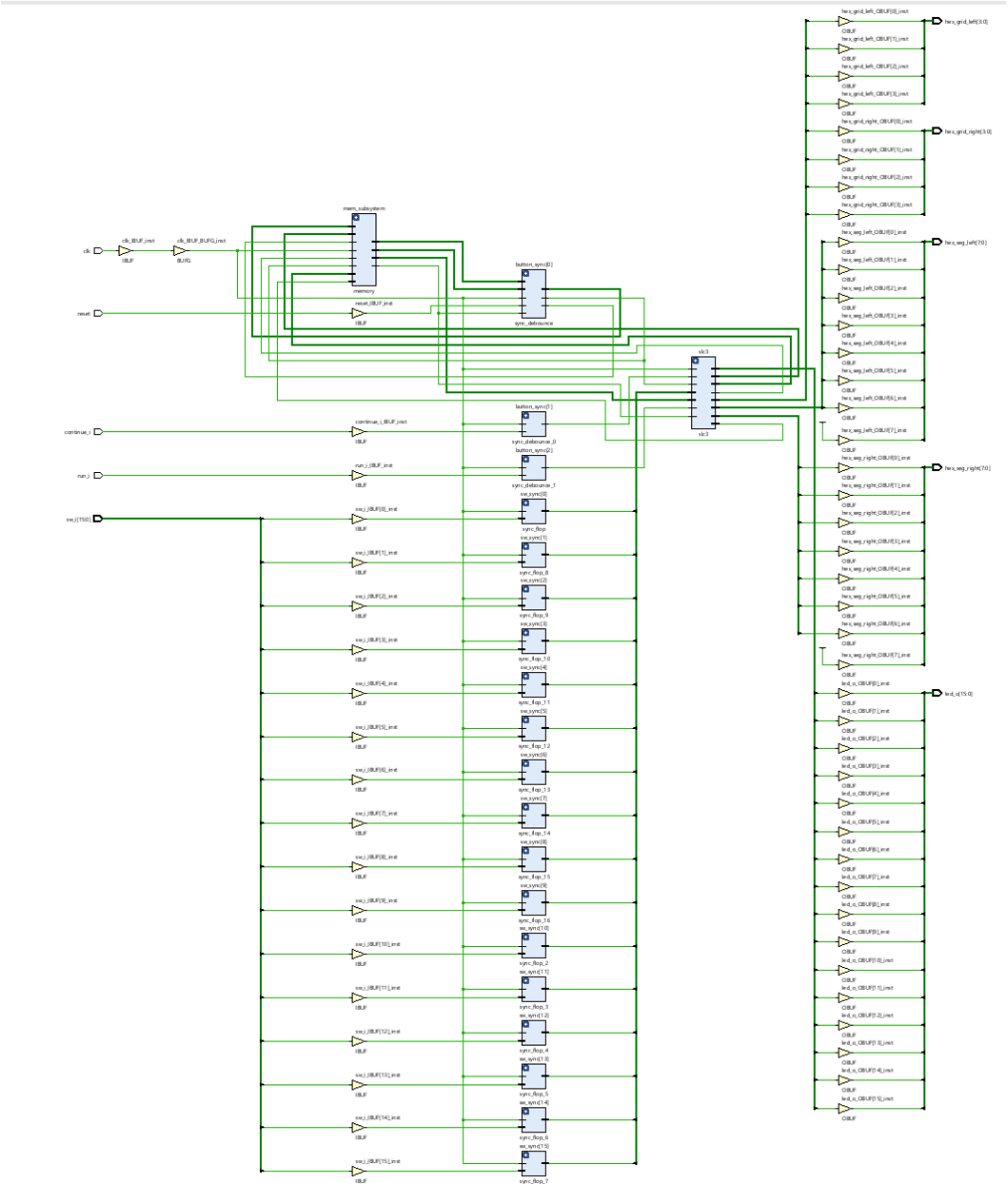
STR
0111



Pause
1101



3. Top Level RTL Diagram



4. Written Description of SystemVerilog Modules

(1) module load_reg

input: clk, reset, load; [DATA_WIDTH-1:0] data_i

output: [DATA_WIDTH-1:0] data_q

Description: DATA_WIDTH here is a parameter for the data width of the register so we don't have to make a new module for every register size. The default value of it is 1. We created a parameter data_d as intermediate variable. If load signal is 1, it loads data_i into data_d. The output data_o of the register is set to zeros if Reset is pressed, while doing a synchronous reset process. In other cases, data_d is loaded to output signal data_q which is to be loaded into register on receiving positive clock signal. Overall, this module store data from

PC, IR, MDR, MAR, and do synchronous Reset and load operation on positive edge of clock signal.

(2) module MUX_2

input: [width-1:0] d0, d1; s

output: [width-1:0] y

Description: implements a 2-1 Mux which has width-bit. Width is a parameter for the data width with default value 16. It select among d0 and d1 on receiving s, and pass it to y.

(3) module MUX_4

input: [width-1:0] d00,d01,d10,d11; [1:0]s

output: [width-1:0] y;

Description: A 4-1 Mux which has width-bit data as input and output. The select signal now has two bits in order to select from four inputs. It is the body of PCMUX and ADDR2MUX.

(4) module MUX_8

input: [width-1:0] In_000,In_001,In_010,In_011,In_100,In_101,In_110,In_111; [2:0]s

output: [width-1:0] y;

Description: A 8-1 Mux which has width-bit data as input and output. Width has a default value of 16 in this case. The select signal now has 3 bits in order to select from 8 inputs. It is used in the structure of ALUA and ALUB MUX.

(5) module MUX_16

input: [width-1:0] d0,d1,d2,d3,d4; [3:0]s

output: [width-1:0] y

Description: A 16-1 Mux which has width-bit data as input and output. The select signal now has 4 bits in order to select from 5 inputs. We use this 16-input MUX to select which signal to drive the CPU bus since FPGA doesn't support tristate buffer.

(6) module Register_file

input: clk, reset, ld_reg; [2:0] DR, sr1,sr2; [15:0] bus

output: [15:0] SR1_out, SR2_out

Description: The module consists of an array of 8 registers (R[8]), each 16 bits wide, capable of storing 16-bit values. These registers are indexed from 0 to 7. The LD signal is an 8-bit vector used internally to determine which of the 8 registers is to be loaded with the value on the bus when ld_reg is asserted. This is controlled by decoding the DR input to set one of the bits in LD high, corresponding to the register to be written. When ld_reg is asserted, the DR input is decoded to set the appropriate bit in LD[7:0]. For example, if DR is 3'b010 (2 in decimal), LD becomes 8'b00000100, indicating that the third register (index 2) is to be loaded with the value present on the bus. If ld_reg is not asserted, LD is cleared to 8'b00000000, indicating that no register should be loaded with new data at this time. The actual logic for loading the register with the value from the bus and updating the SR1_out and SR2_out outputs based on the sr1 and sr2 inputs is not shown in the provided code snippet. Typically, this

would be handled by additional logic within the module that uses the ld_reg signal to trigger the loading of the bus value into the appropriate register and to update the outputs based on the selected source registers.

(7) Module cpu

input: clk, reset, run_i, continue_i, [15:0] hex_display_debug, [15:0] led_o, [15:0] mem_rdata,

output: [15:0] mem_wdata, [15:0] mem_addr, mem_mem_ena, mem_wr_ena

Description: The cpu module demonstrates key concepts of computer architecture including the fetch-decode-execute cycle, register file operations, and the use of condition codes for branch decisions. It simulates core CPU operations including instruction fetch, decode, and execute phases. It manages internal state through registers such as the Memory Address Register (MAR), Memory Data Register (MDR), Instruction Register (IR), and Program Counter (PC). Logic is implemented to handle arithmetic and logical operations, conditional branching based on flags (NZP - Negative, Zero, Positive), and memory access (load/store). Instruction execution is controlled by a series of multiplexers and conditionals that guide the flow of data and instructions through the system, emulating how a real CPU functions.

(8) module hex_driver

input: CLK, Reset, [3:0] in[4]

output: [7:0] hex_seg, [3:0] hex_grid

Description: It takes output bits of registers and 3-bits into Hex number and display on LED unit of the FPGA. The input value of module is 4 four-bit signal, totally make up to 16 bits. The module converts upper 4bits and lower 4bits in register value to two hex displayers respectively. The displayed numbers are in Hex format, and user can see value of register A & B separately.

(9) module nibble_to_hex

input: [3:0] nibble

output: [7:0] hex

Description: This module is a sub-module within hex_driver. It convert 4-bit nibble signal to 8-bit hex signal by using always comb logic and switch on different cases of nibble's value.

(10) module cpu_to_io

input: clk, reset, [15:0] cpu_addr, cpu_mem_ena, cpu_wr_ena, [15:0] cpu_wdata, [15:0] sw_i, [15:0] sram_rdata

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

Description: This module effectively acts as a bridge, providing the CPU with direct access to specific I/O devices based on predefined memory-mapped addresses, streamlining the interaction between computational and display/logic elements in embedded or educational systems. It routes data from the CPU to SRAM and vice versa, using the cpu_addr to determine the memory address for read or write operations.When the CPU attempts to access the special address 0xFFFF, the module instead reads input from switches (sw_i) for reading

operations, or updates the hex display for writing operations.The hex_display is updated on the edge of the system clock and based on the data written to address 0xFFFF. This display data is then decoded and sent to a hex driver, which controls the hex grid and segment outputs for visual representation. For non-special addresses, the module simply passes through the read and write operations to and from the SRAM, making it transparent to the CPU whether data is coming from or going to SRAM or the hex display mechanism.

(11) module instantiate_ram

input: reset, clk

output: [9:0] addr, wren, [15:0] data

Description: module is designed to initialize a RAM block with predefined data upon system reset. It leverages an external function, memContents, which is assumed to provide the initialization data based on the provided address. This module is a part of a larger system that might be simulating or implementing a memory initialization sequence in hardware. It automates the process of loading initial values into a RAM block at startup, a common requirement in embedded systems and simulations to ensure that memory is in a known state before beginning normal operations.

(12) module memory

input: reset, clk, [15:0] data, [9:0] address, ena, wren

output: [15:0] readout

Description: This module is designed to abstract the functionality of a memory system, with different implementations for synthesis and simulation/test environments. This module interfaces with external logic, providing read and write capabilities to a memory block based on input signals. It supports initialization through a separate instantiate_ram module for loading initial values into memory upon reset. It acts different under Synthesis Environment and Non-Synthesis (Test) Environment. The design effectively separates concerns between hardware implementation and testing, enabling efficient development and verification of memory operations in digital systems.

(13) module sync_debounce

input: clk, d

output: q

Description: This module is designed to mitigate the effects of signal bouncing, typically caused by mechanical switches or noisy digital inputs. It does so by implementing a synchronization and debouncing mechanism using flip-flops and a counter. The module ensures that only stable, debounced signals are passed as output, reducing the likelihood of erroneous readings due to transient signal changes.

(14) module sync_debounce

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

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

Description: This module module serves as the top-level design for a processor

system, incorporating various components to demonstrate the integration of a CPU core with memory and I/O functionalities. It is designed to handle input signals, process them, and provide output signals to LEDs and hexadecimal displays. The module is structured to facilitate simulations and real-world applications involving user interactions and data processing.

5. Description of the operation of the control unit

The control unit is implemented in file control.sv and work as a module within CPU module, managing the execution of instructions and the coordination of various components within the SLC-3.

- **Inputs**: The control unit receives inputs such as the clock signal (clk), reset signal (reset), current instruction register (ir), branch enable signal (ben), memory read data (mem_rdata), and control signals like continue_i and run_i.
- **Outputs**: The control unit outputs signals to control various parts of the system, including load signals (ld_mar, ld_mdr, ld_ir, etc.), gate signals (gate_pc, gate_mdr, gate_alu, etc.), multiplexer controls (pcmux, drmux, sr1mux, etc.), and memory control signals (mem_mem_ena, mem_wr_ena).

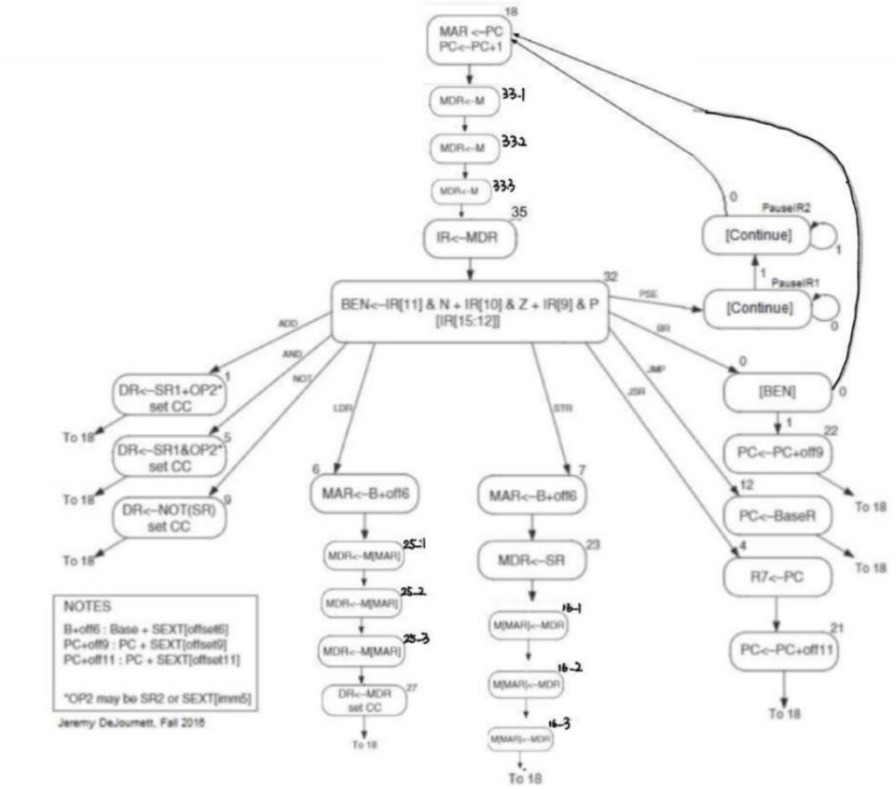
The control logic is implemented as a state machine with enumerated states representing different stages of instruction execution. The key states are:

- **Halted:** The initial state of the machine, waiting for a run_i signal to start execution while doing nothing.
- **Instruction Fetch and Decode:** States like s_18, s_33_1, s_33_2, s_33_3 and s_35 is involved in fetching and decoding the instruction from memory. These states make up the Fetch and Decode operation in processor system, while storing instructions in IR and decoding it for future use in Execute operation.
- **Execution States:** Each instruction type has specific states associated with its execution. For example, s_01 for ADD, s_05 for AND, s_09 for NOT, s_00 for BR (branch), s_12 for JMP (jump), and so forth. These states manage the specific control signals needed to execute the instruction, such as setting up the ALU, accessing memory, or updating the program counter.

As a finite state machine, the control unit moves through its states based on the current state, inputs, and the opcode of the fetched instruction. For instance, after resetting, it transitions from the halted state to s_18 to start the instruction fetch process. For branch instructions, it examines the ben signal to decide whether to take the branch (s_22) or continue sequentially (s_18). For memory access instructions like LDR (s_06) and STR (s_07), it goes through a series of states (s_25_1, s_25_2, s_25_3 for LDR and s_23, s_16_1, s_16_2, s_16_3 for STR) to perform the memory read or write operation, and wait for three sub-states in process to make sure there won’t be synchronization problem between CPU and memory block.

The control unit manages memory access through signals like mem_mem_ena and mem_wr_ena, indicating whether a memory operation is enabled and if it's a write operation, respectively. It controls the ALU operation via the ALUK signal, determining the operation to be performed based on the instruction.

6. State Diagram for Control Units

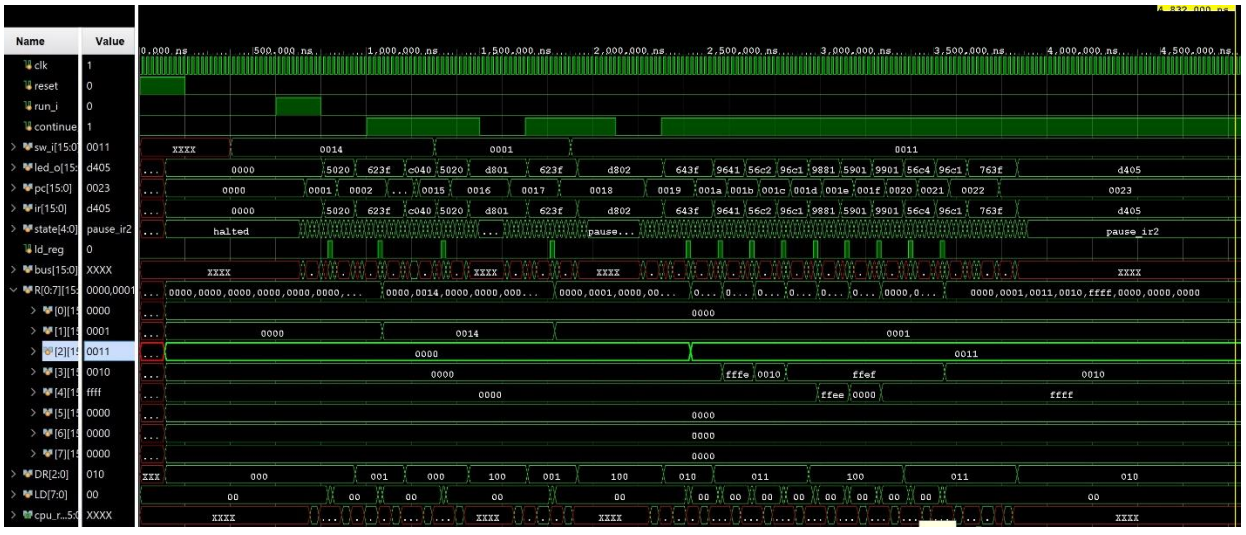


7. Simulation Graph

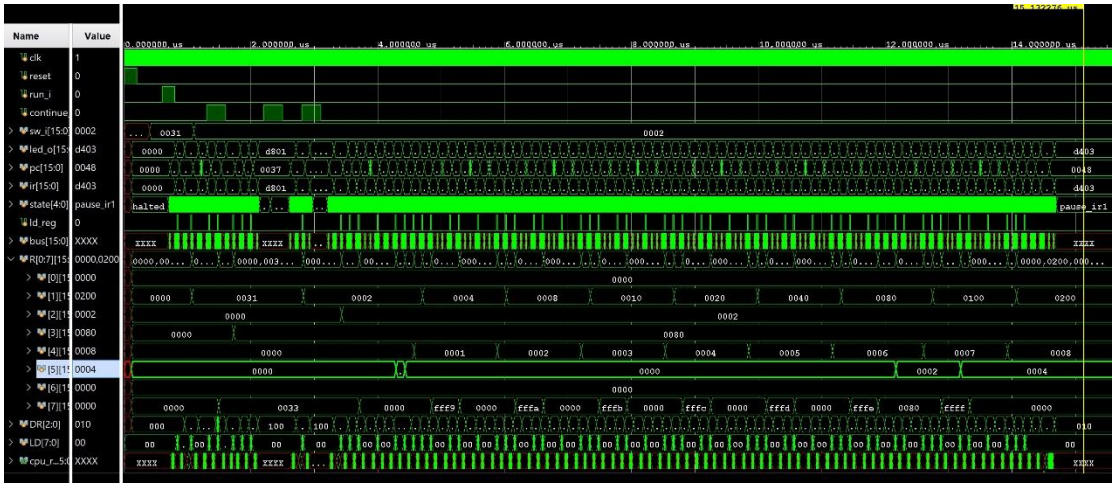
IO_test 1

IO_test 2

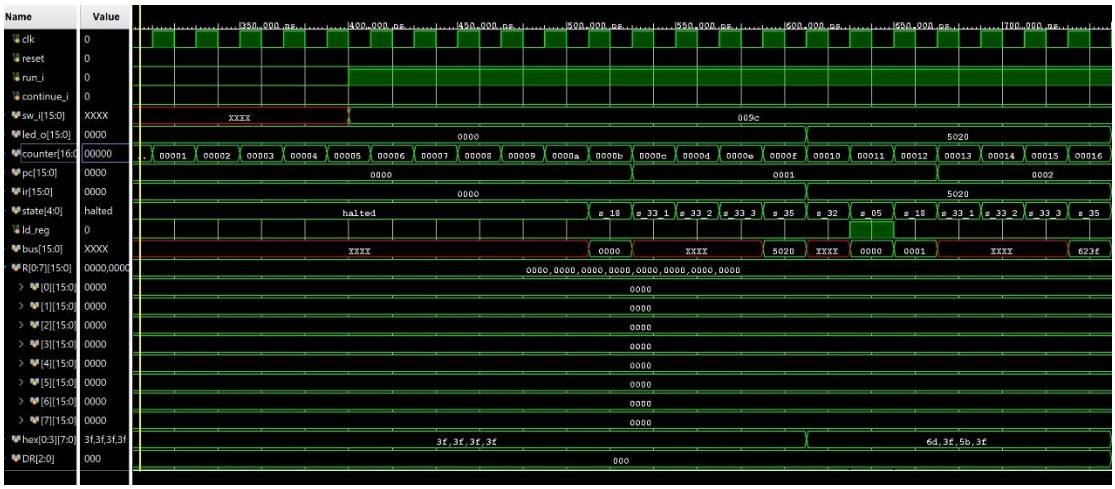
Self-Modifying Code



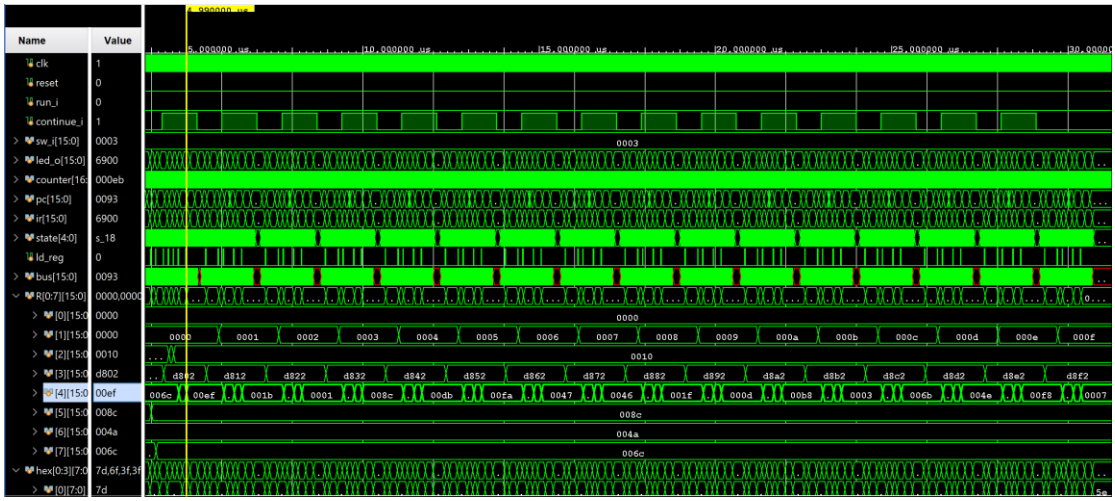
XOR test



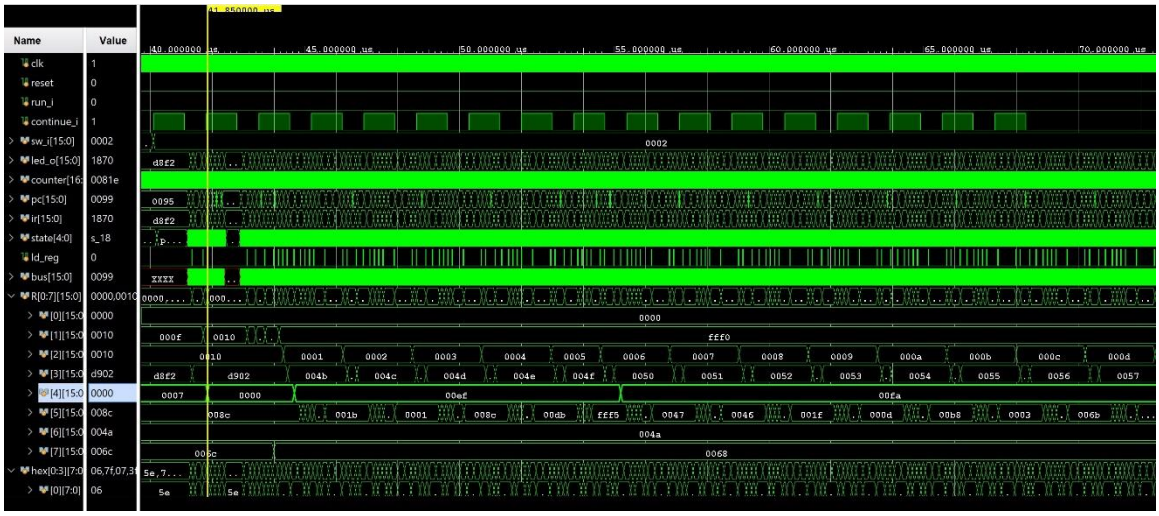
Multiplier test



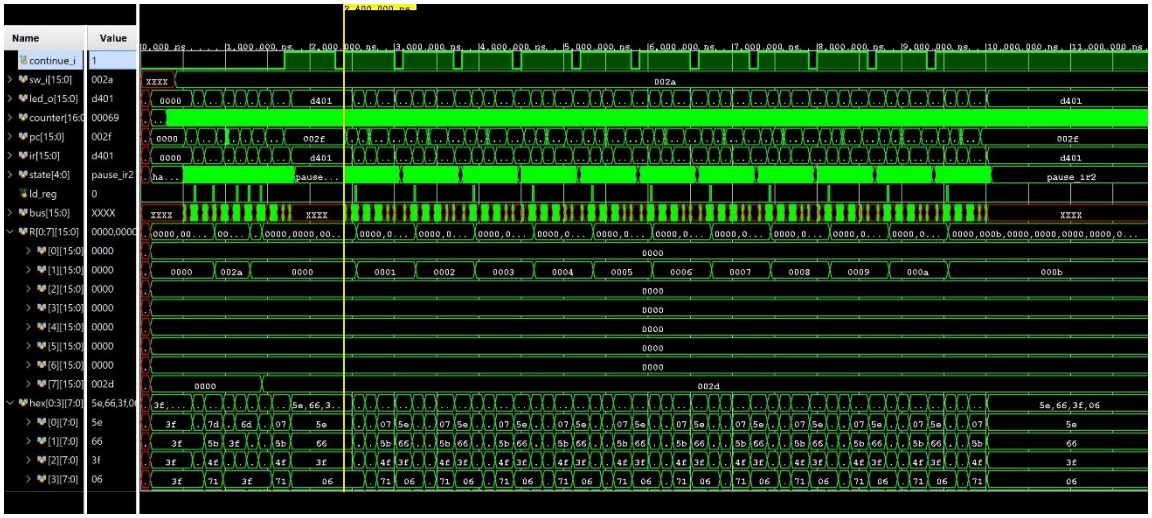
Auto Count test



Before Sort

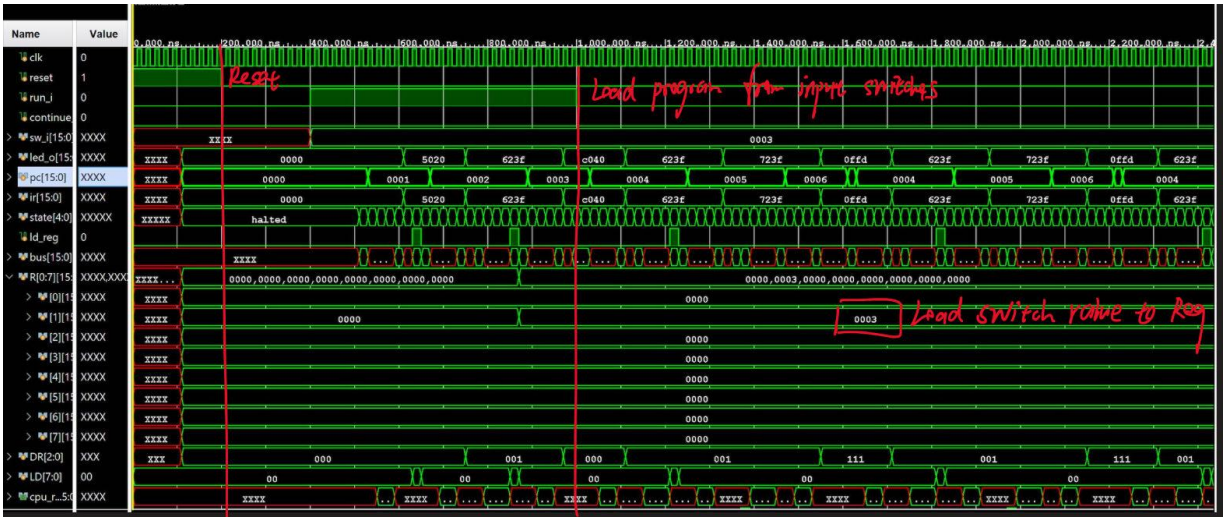


Being Sort

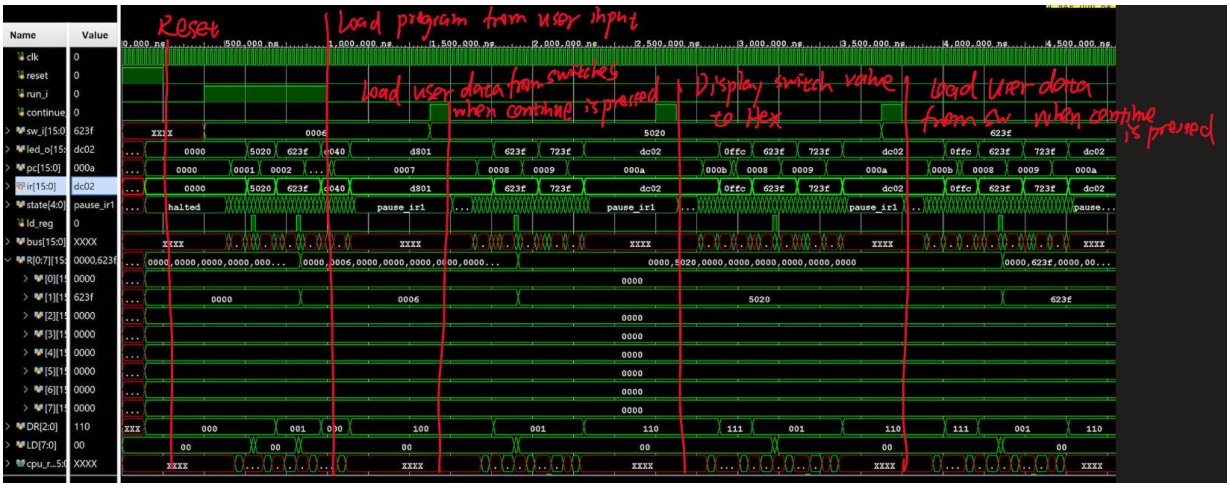


Test at once

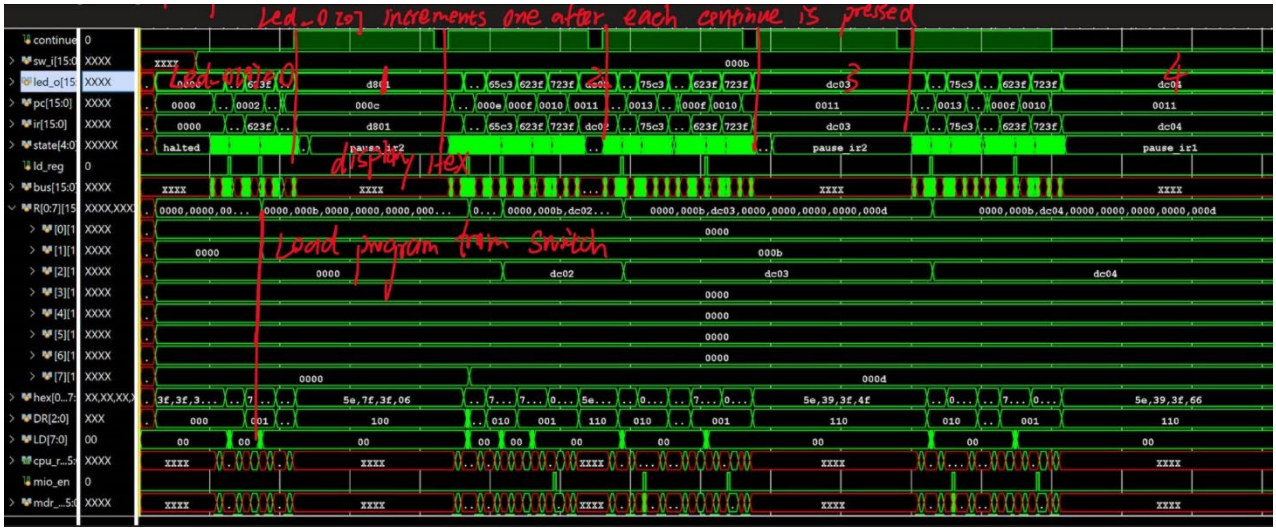
Annotated Version:



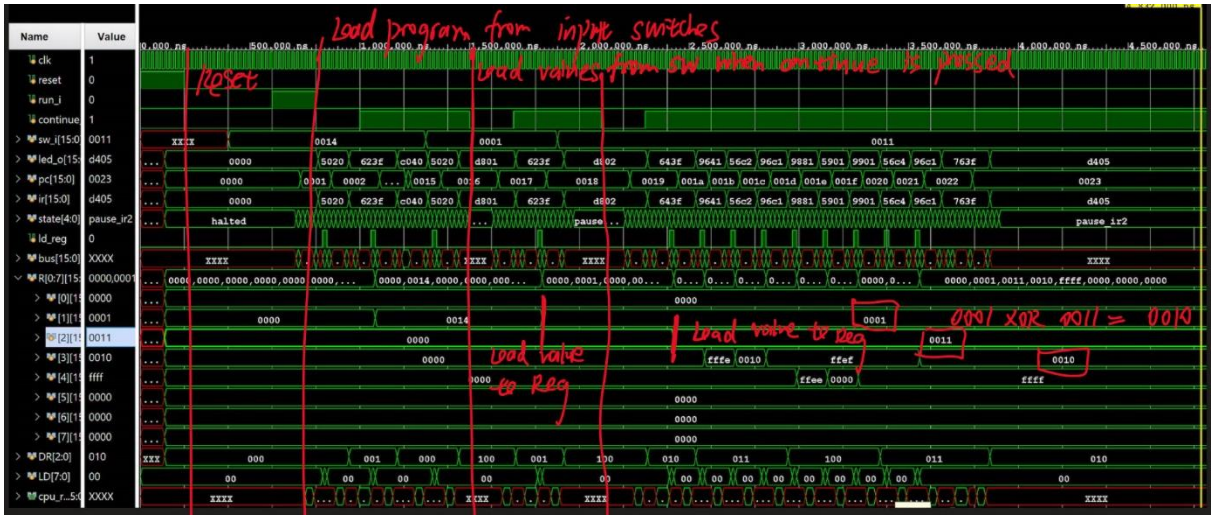
IO_test 1



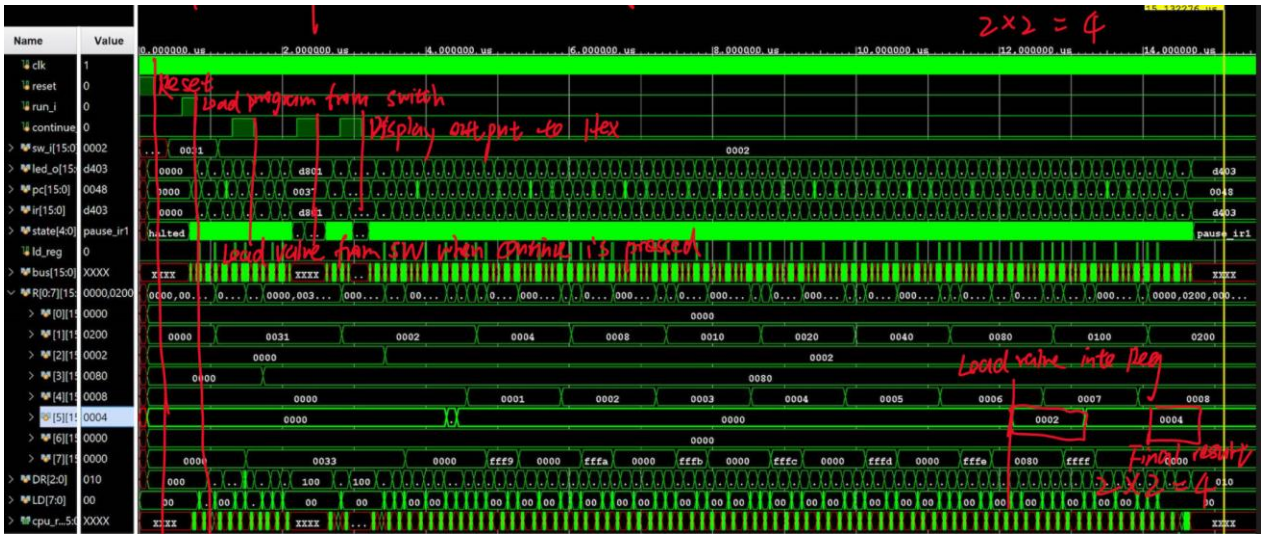
IO_test 2



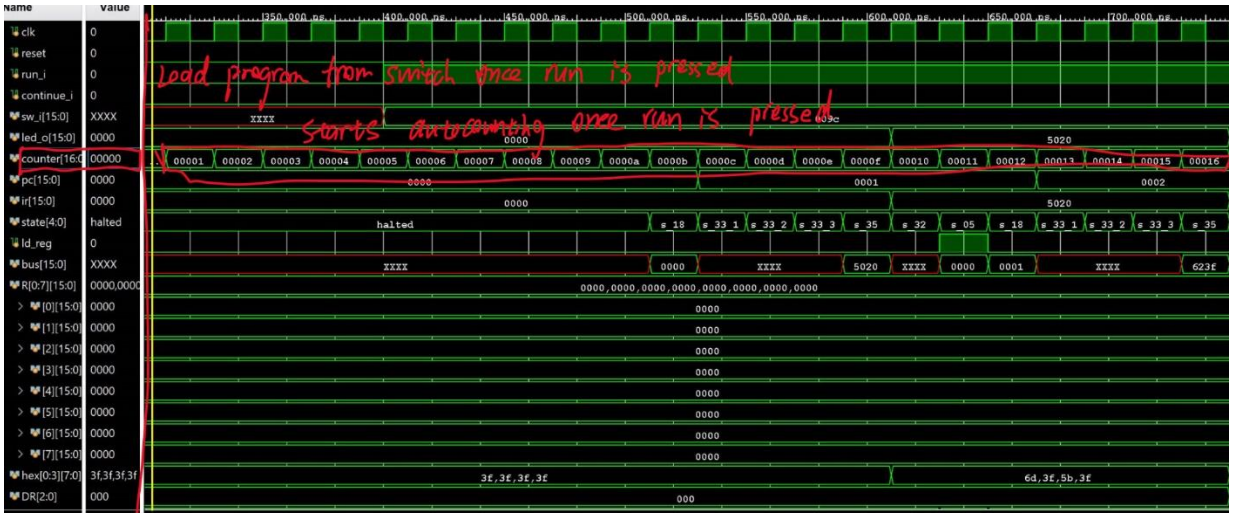
Self-Modifying Code



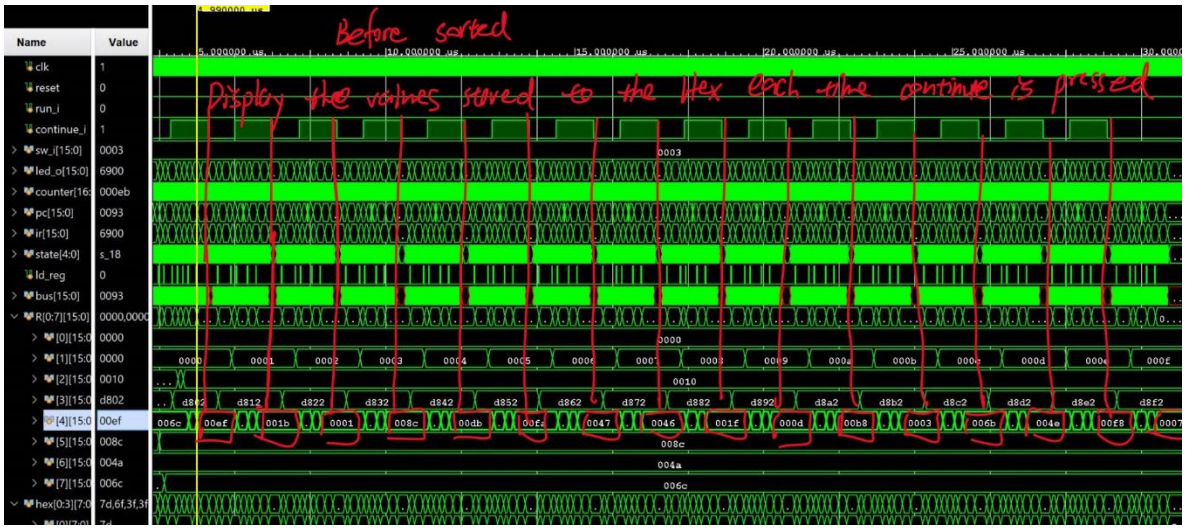
XOR test



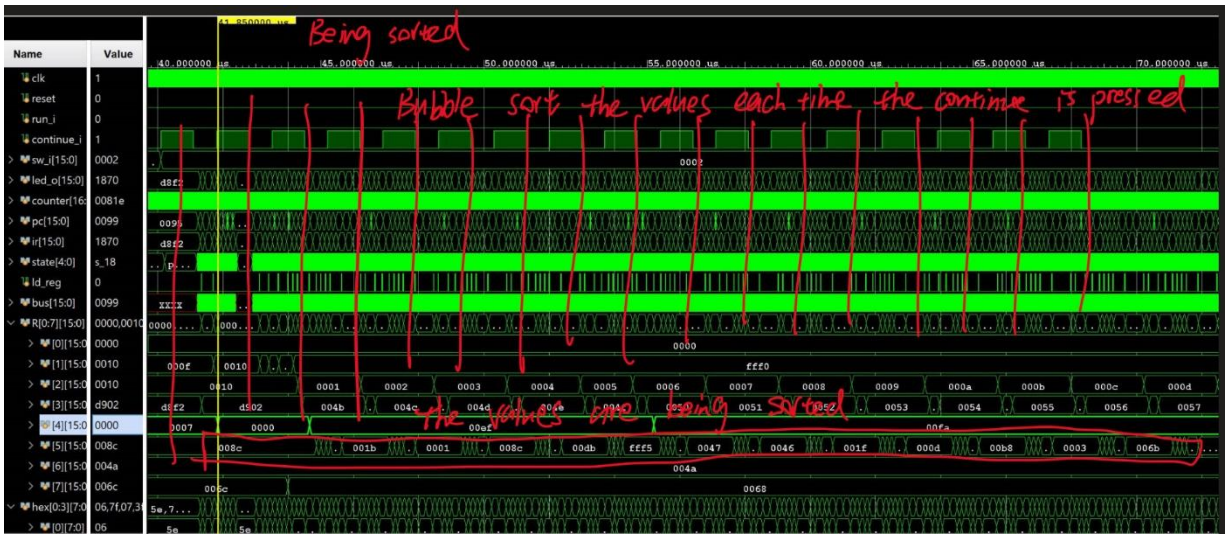
Multiplier test



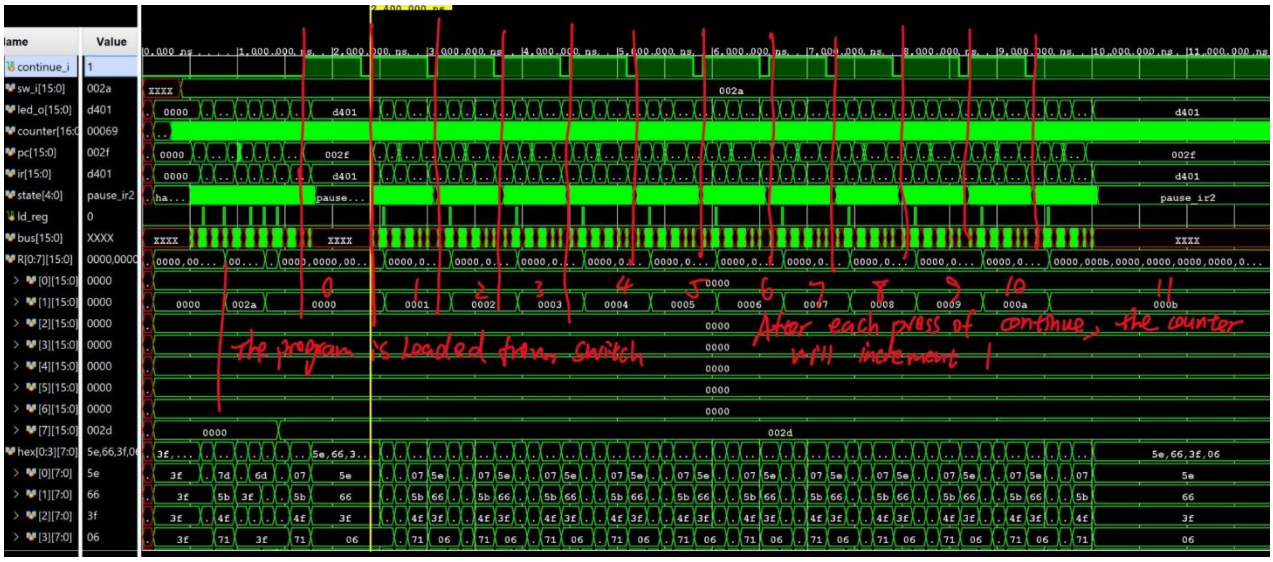
Auto Count test



Before Sort



Being Sort



Test at once

8. Answer to Post-lab Questions

(1) Refer to the Design Resources and Statistics in IVT and complete the following design statistics table.

LUT	*411*
DSP	*0*
Memory (BRAM)	*1*
Flip-Flop	*345*
Frequency	*100.000MHz*
Static Power	*0.071W*
Dynamic Power	*0.010W*
Latches	*0*
Total Power	*0.081W*

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

This unit manages all I/O with the Urbana board's physical I/O devices. CPU_TO_IO block act as the intersection between CPU block and on-chip physical memory and other I/O signals available on FPGA board, namely, the switches and 7-segment displays. It receives enable and data signals from CPU block, and access SRAM when the memory address is 0xFFFF, which signals that CPU should read from switches' memory / write to displays' memory

rather than instantiated BRAM. Meanwhile, it has a reset signal to reset the HEX display. Thus, this block enables user to set input signals manually and see the result of operation intuitively by interacting with CPU, SRAM, and Hex displays.

(3) What is the difference between BR and JMP instructions?

JMP instruction does unconditional jump, while only using program counter signal (PC) and address offset from instruction register (IR). BR do branch conditionally, depending on the value in BEN register to decide whether to jump or not.

(4) 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 signal means "ready" in Patt and Patel system, and it tells the system whether FSM is ready to go to the next state rather than keep staying in current state. Specifically, it signals the processor that system is ready to do read and write operation that relates to MAR and MDR. We compensate the lack of it by adding three sub-states to do waiting every time we need to access memory for reading or writing. This method ensures that we won't has synchronization problem related to memory accessing by guarantying memory system is ready for read or write after waiting.

9. Bug encountered

(a) Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.

. When I finished the codes first time, I failed to meet the expected results. And I rechecked my codes over ten times. At last, I found there were minor mistakes when zero extending the data from the IR and BUS. I only wrote one zero on the left of each original data, which caused failure. In other aspects, the code worked well.

(b) Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab? manual or given materials which can be improved for next semester? You can also specify what we did right, so it does not get changed.

.It was clear and easy to understand the tasks to be done. The lab manual is helpful when it comes to the well-labeled SLC3 diagram and provided test instructions.

10.Conclusion

In this lab, we went through the LC-3 system learned in ECE220 and combined it with newly learned knowledge like System Verilog and HDL logic. The SLC-3 system we built up is a simplified version of original LC-3 system in Patt and Patel,

but the process building it up was still challenging, especially when we need to understand and modify the FSM in control.sv so that it can implement all functionality required.