ECE 385

Spring 2024

Lab 5

Ziheng Li (zihengl5)

Xin Yang (xiny9)

Section AL1

Prof. Zuofu Chen

Introduction

In this experiment, we implement a simplified version of LC-3 using SystemVerilog. It contains a cpu that can perform basic logic operations like ADD, AND, NOT, and basic memory operations like LDR, STR, along with branching and jump functions. In addition, it includes the functionality of read/write the memory in SRAM, along with an input/output interface that communicates with the external device. Overall, it operates in the order of Fetch–Decode–Execute. Compared to the LC-3 we learned in ECE120, it reduces several instructions like LEA, RTI, and it adds an instruction PAUSE to better observe the result / loading new instructions. It also contains several pause states to fulfill the function of the R signal in the Fetch process.

Written Description and Diagrams of SLC-3

Summary of Operation

First step to operate the SLC-3 is to press Reset (btn_0) to initialize the program counter and registers. Second, we flip the switch to the desired test, and then press Run (btn_1). Some of the tests might require flipping the switch several times and pressing Continue (btn_2).
In this experiment, the 0x0003 I/O test can update the hex_display synchronously when the button changes. For convenience issues, we first run this test; flip the switch to the test we want, then press reset and run the new test.

Fetch-Decode-Execute

The SLC-3 has three upper phases of operation: Fetch–Decode–Execute. During the Fetch phase, the value in the PC is loaded into MAR and then incremented by one. The next step is to load the memory from address in MAR into MDR and wait for 3 cycles to make sure the load is complete (mimic R signal).Then it loads MDR into IR to finish fetching instructions.

The next phase is Decode. In state 32, highest 4 bits of the IR will determine the next state, or in terms of upper level, what execution will the computer perform. This phase also check the flags NZP to determines branch or not if the highest 4 bits of IR is 0000.

In phase 3 Execute, it decides the operation for ALU and various control signals. It will executes the operations specified by IR and stores the result into memory space or into registers. This phase might contains one or multiple states.

Instructions

The SLC-3 has 11 instructions: ADD, ADDi, AND, ANDi, NOT, BR, JMP, JSR, LDR, STR, PAUSE. Besides PAUSE, everything is exactly the same as we learned in ECE120&220. PAUSE state allows the user time to set the switches before an input operation and read the output after an output operation. ledVect12 will also displayed on the board LEDs. The detailed instructions are shown in the table 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

Fig.1 SLC-3 ISA

Block diagram

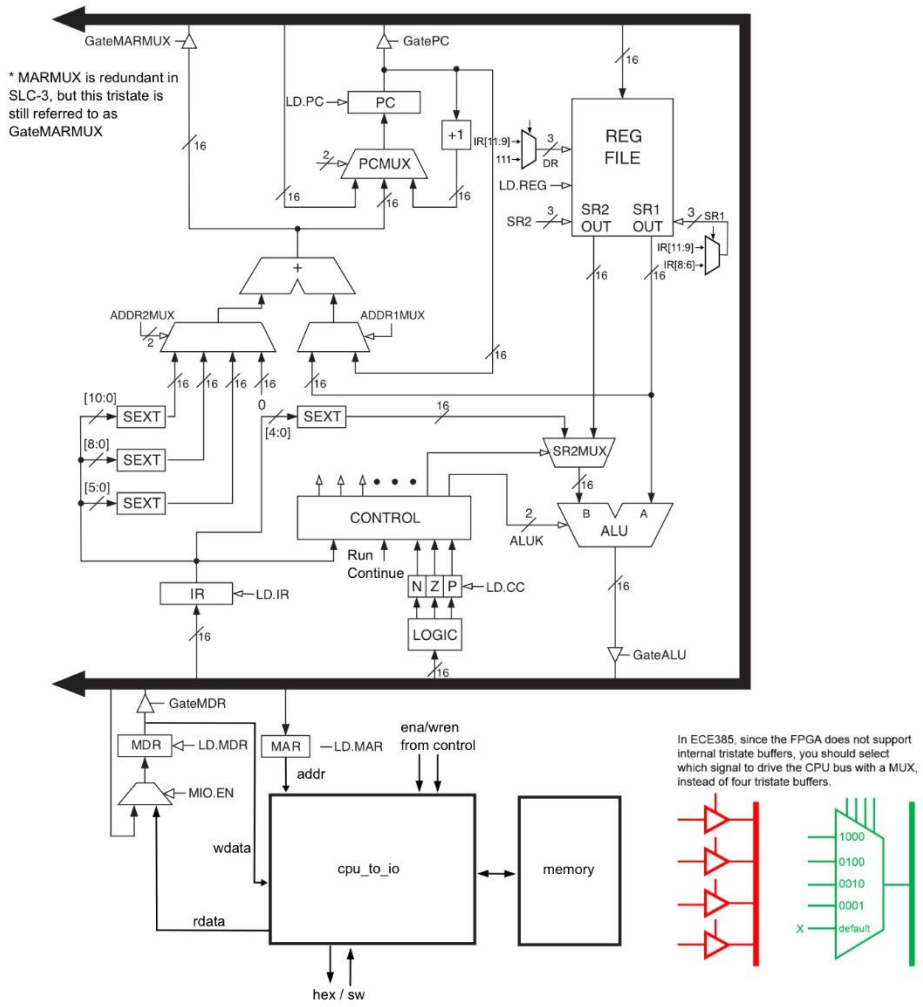

Fig.2 Data path for SLC-3

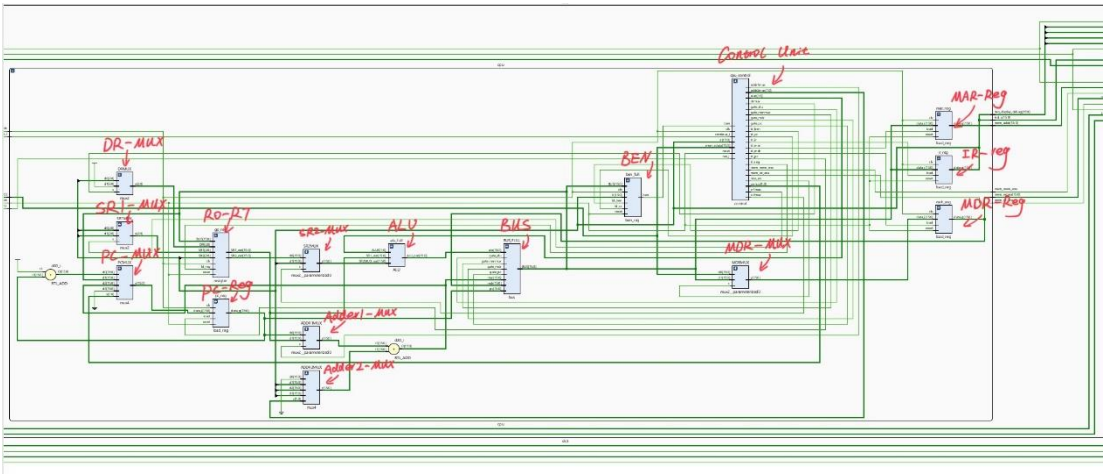

Fig.3 Overall data path for cpu.sv

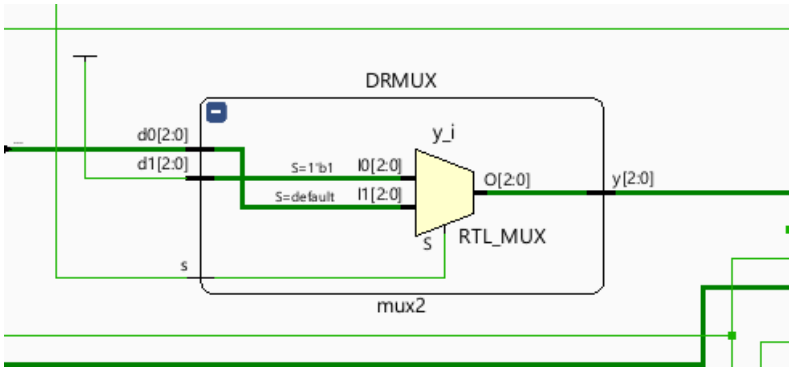

Fig.4 Two to one MUX

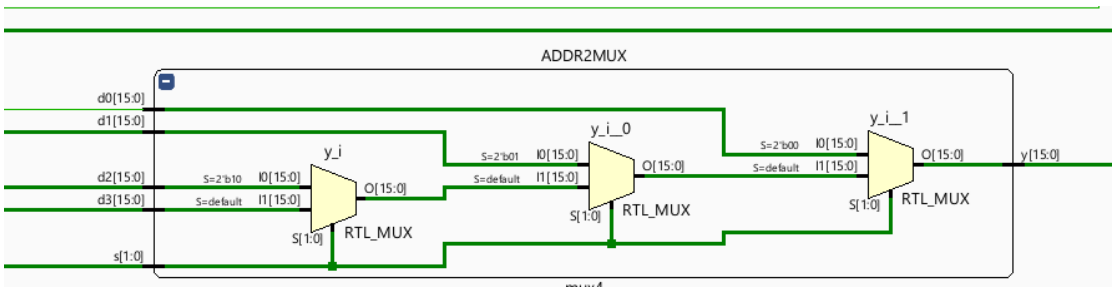

Fig.5 Four to one MUX

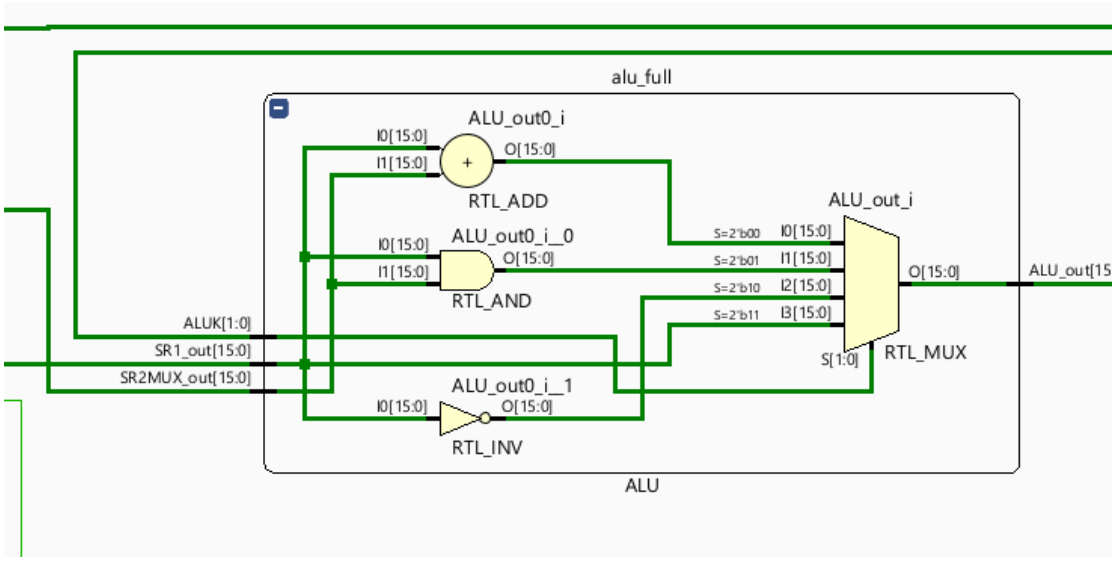

Fig.6 ALU unit

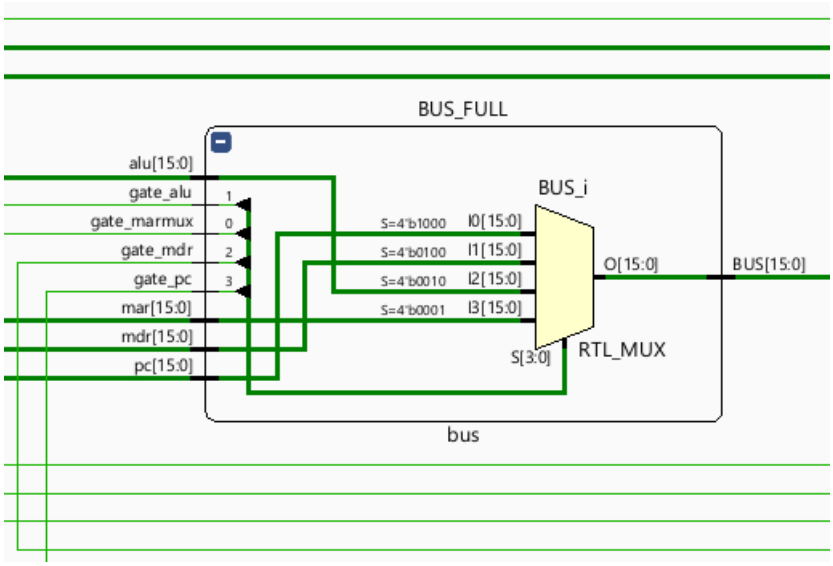

Fig.7 BUS

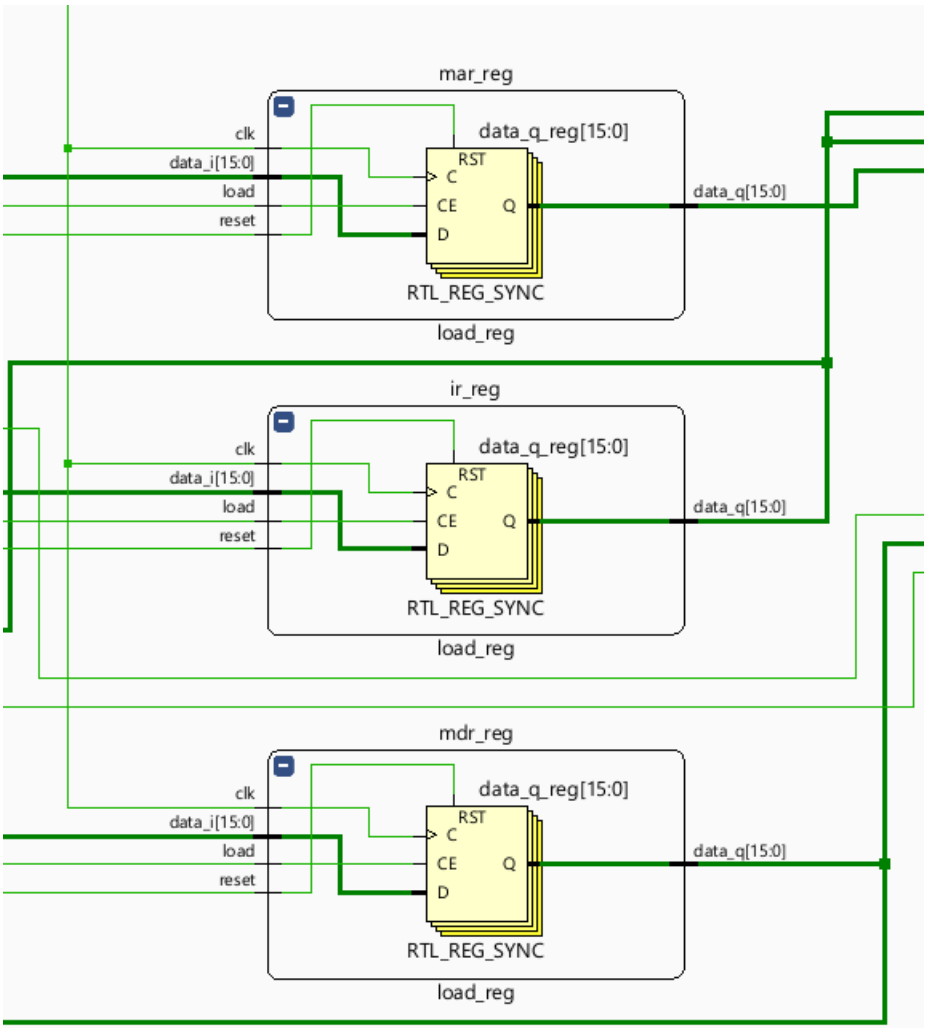

Fig.8 Special Registers

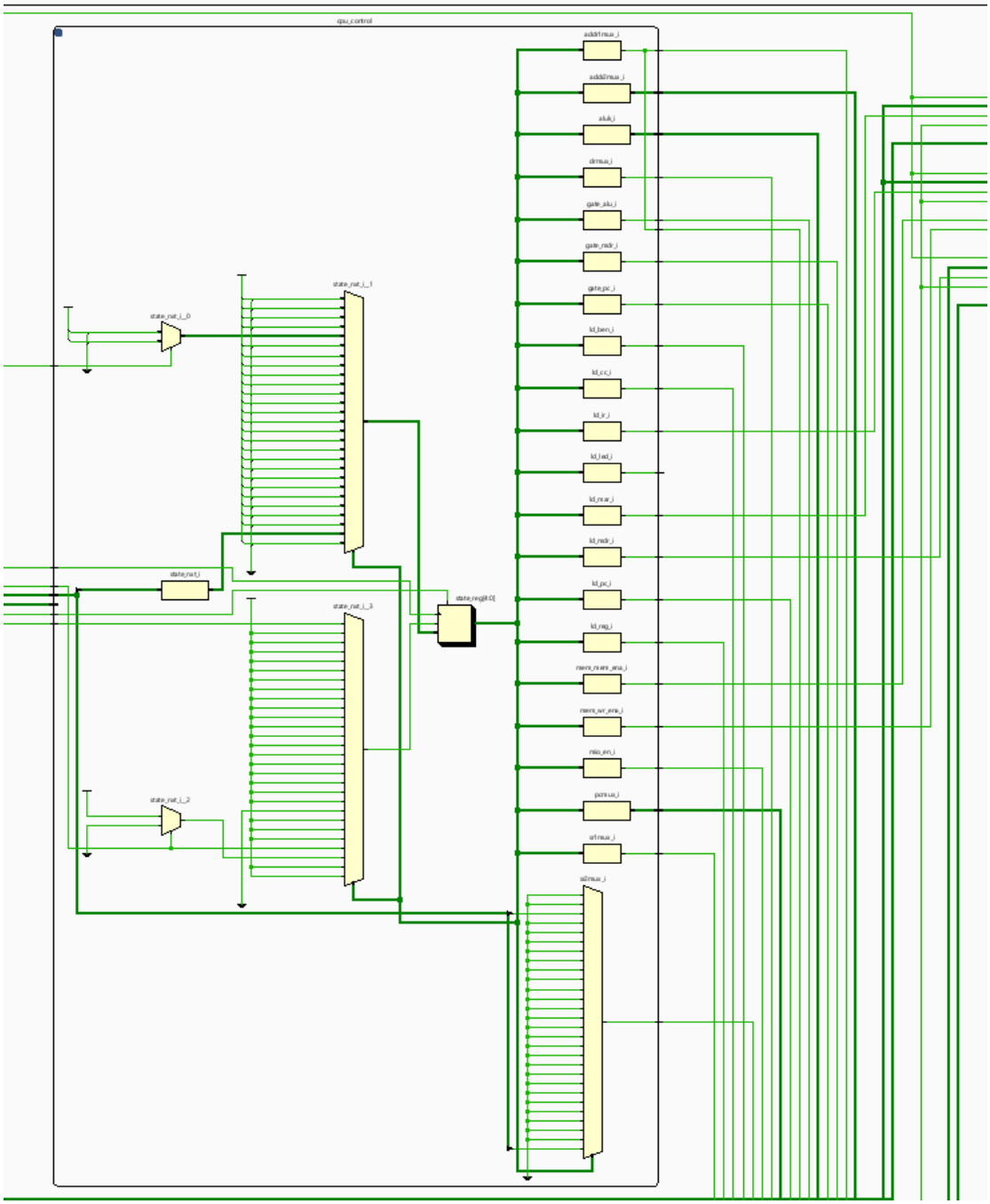

Fig.9 Control Unit block diagram

Annotated Block Diagram

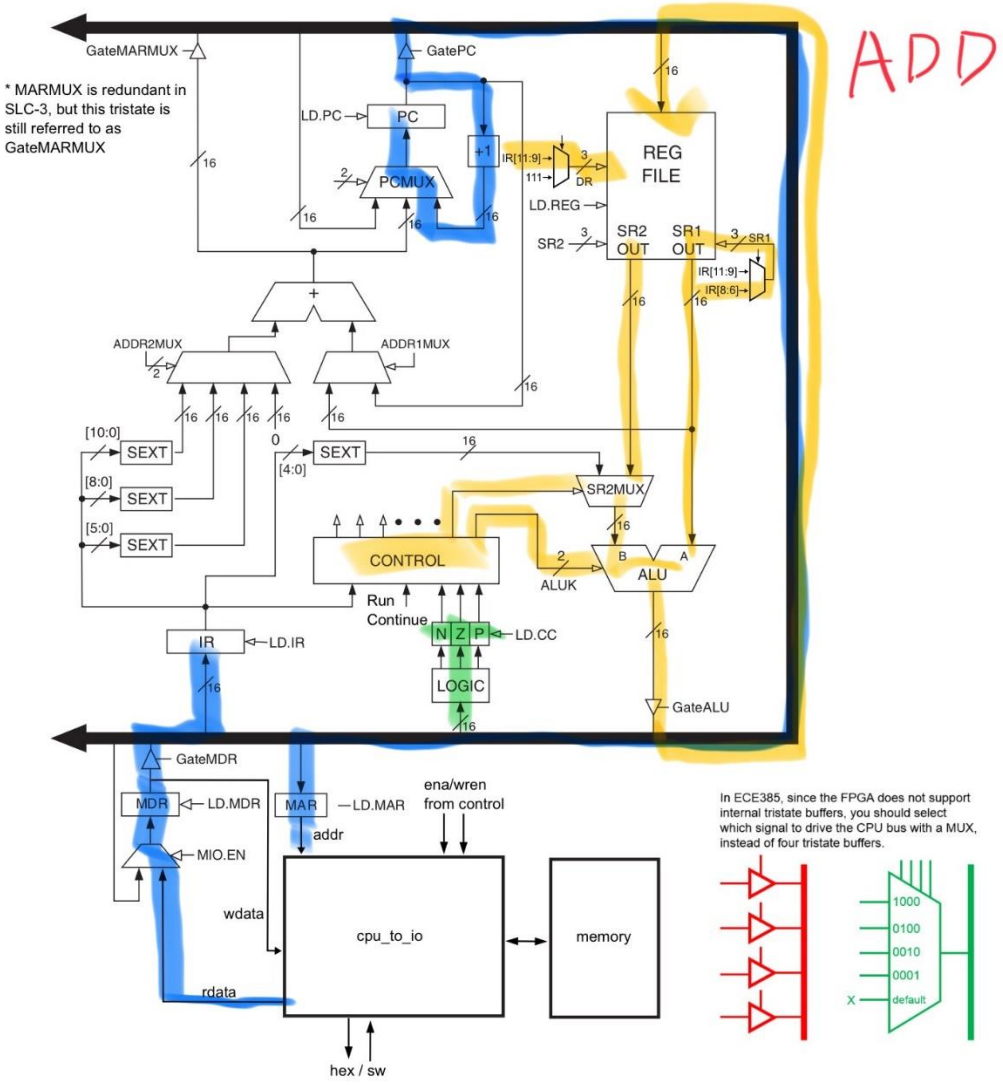

Fig.10 Block diagram for ADD

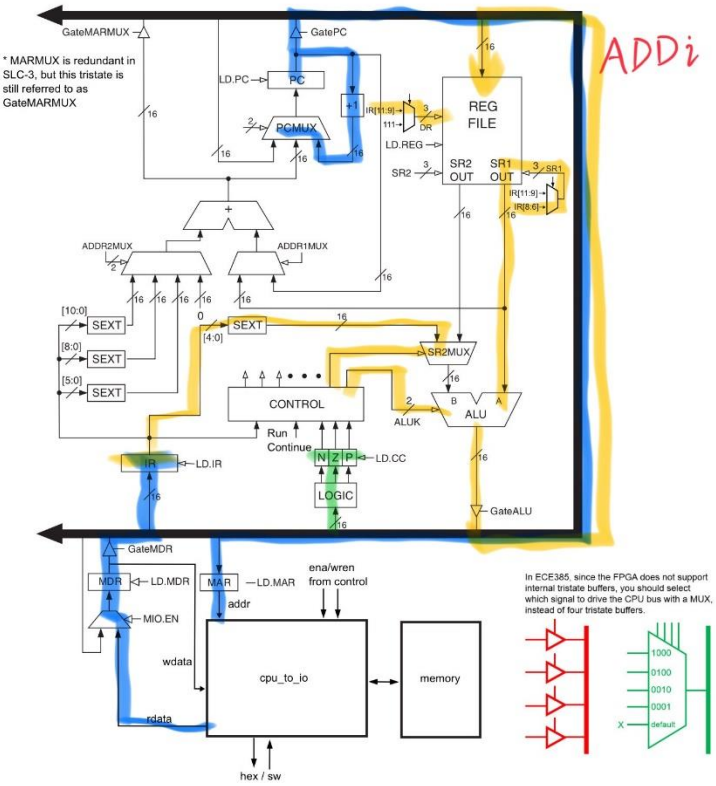

Fig.11 Block diagram for ADDi

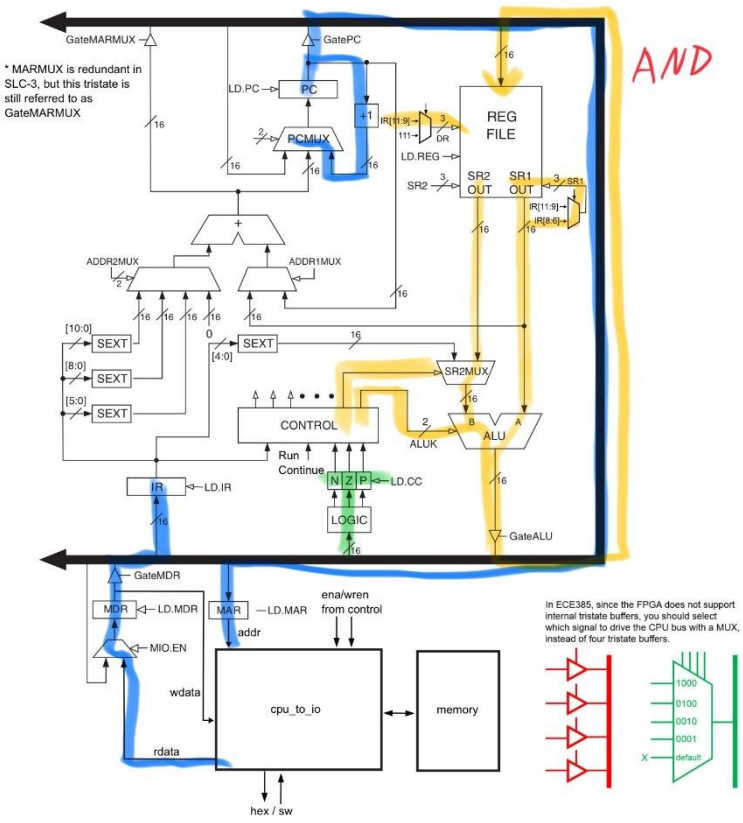

Fig.12 Block diagram for AND

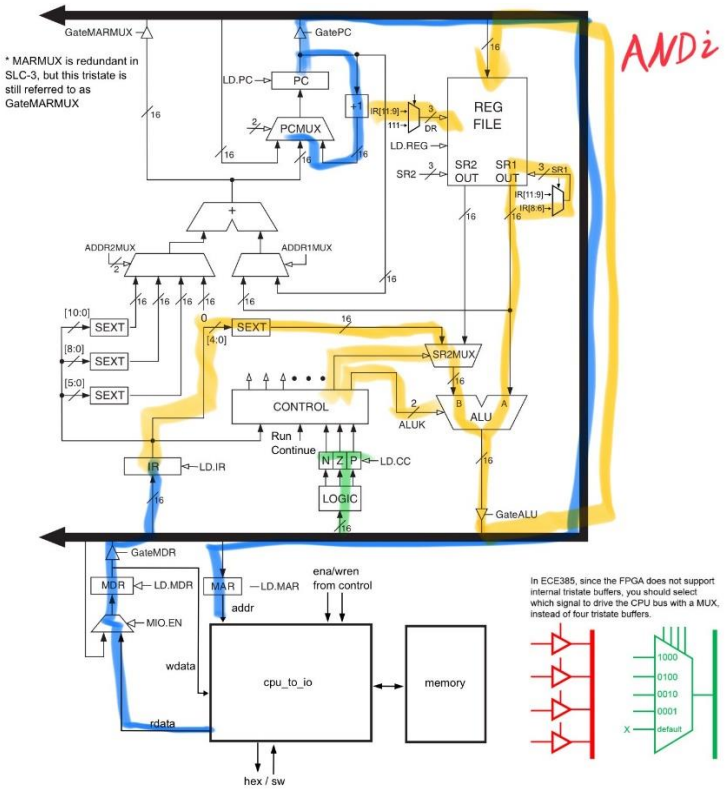

Fig.13 Block diagram for ANDi

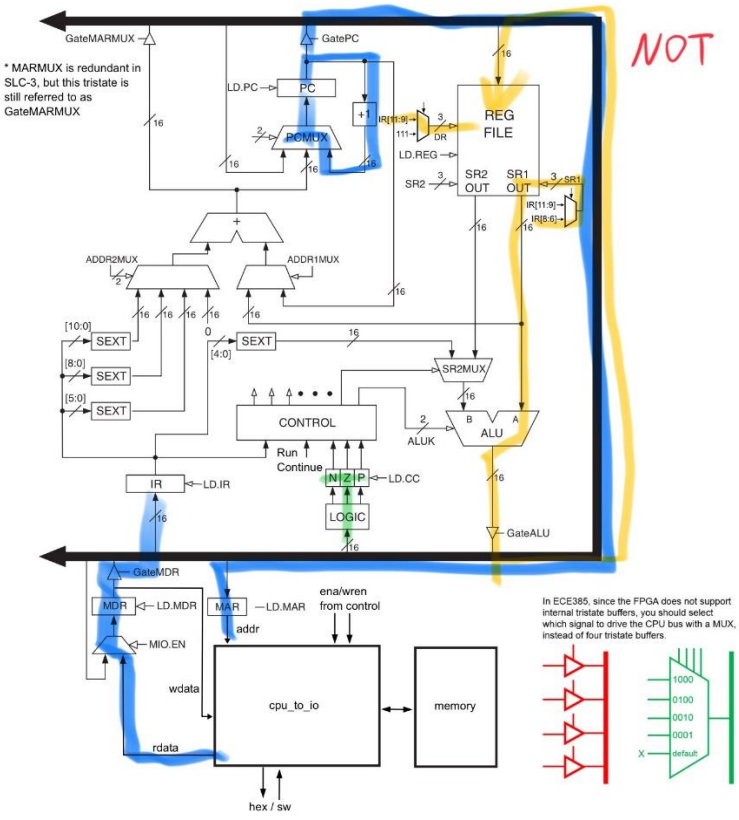

Fig.14 Block diagram for NOT

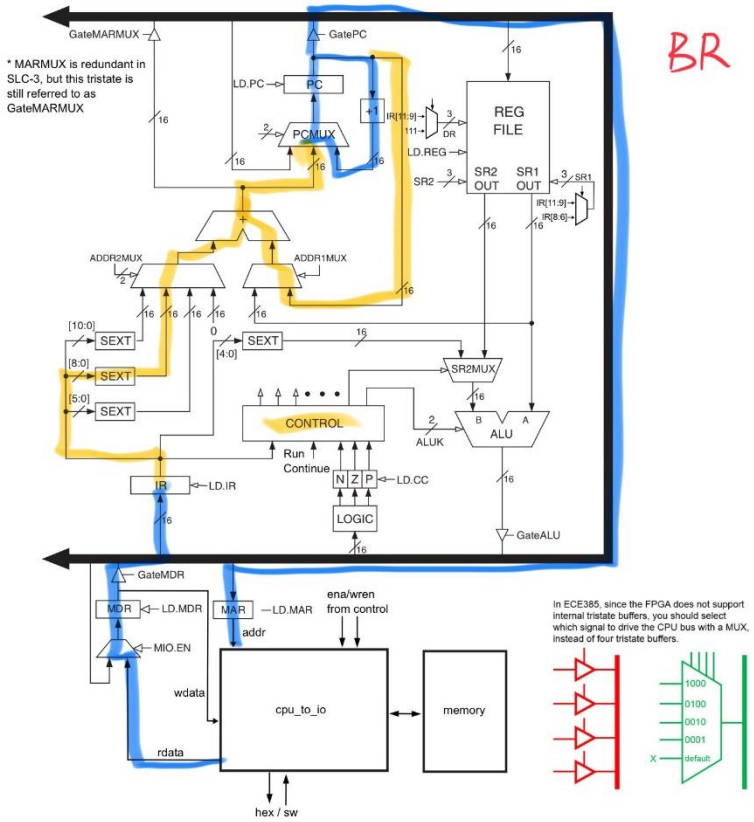

Fig.15 Block diagram for BR

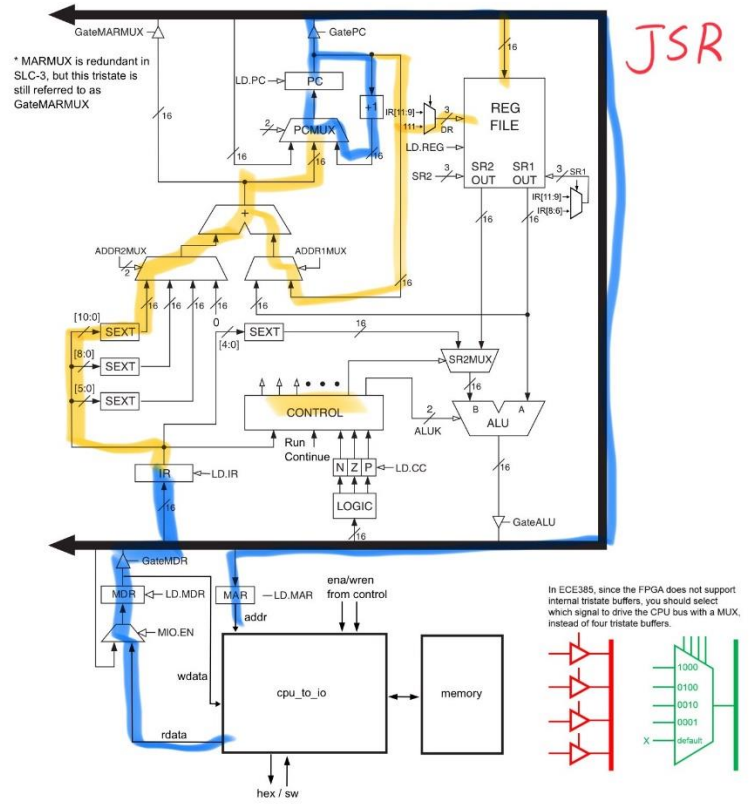

Fig.16 Block diagram for JSR

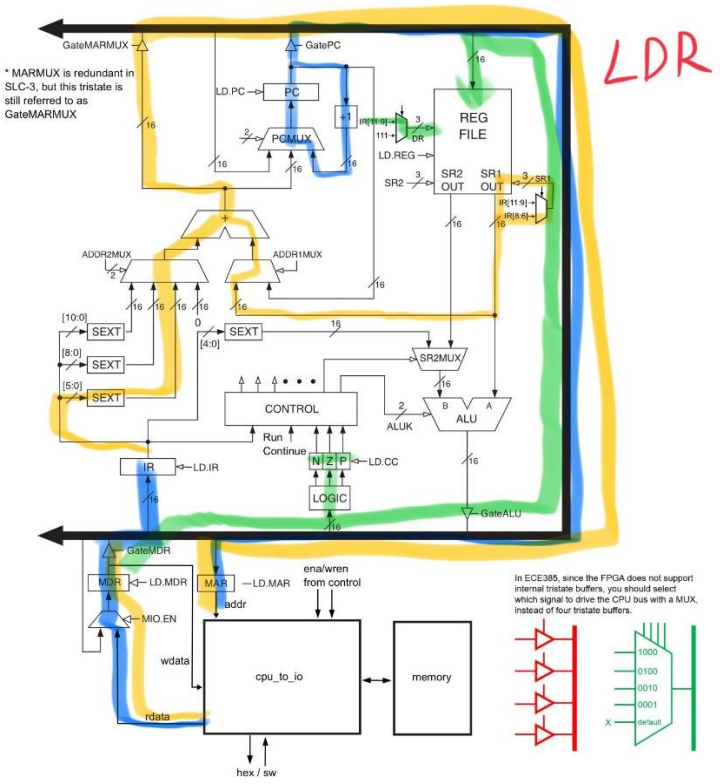

Fig.17 Block diagram for LDR

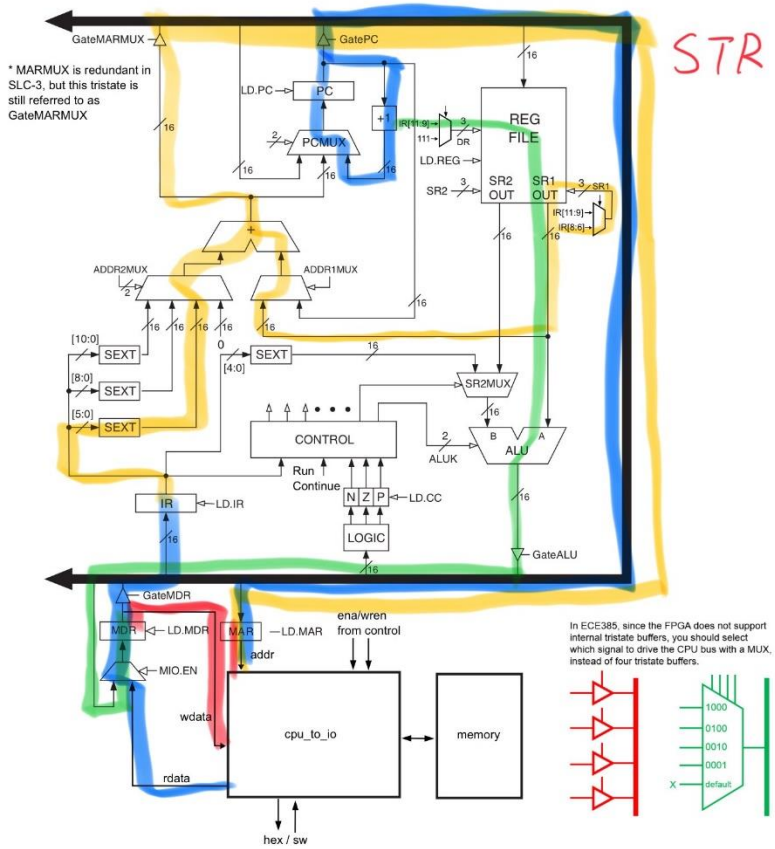

Fig.18 Block diagram for STR

Connection between CPU & IO & memory

From left to right, each block represents Memory, CPU, and IO Bridge.

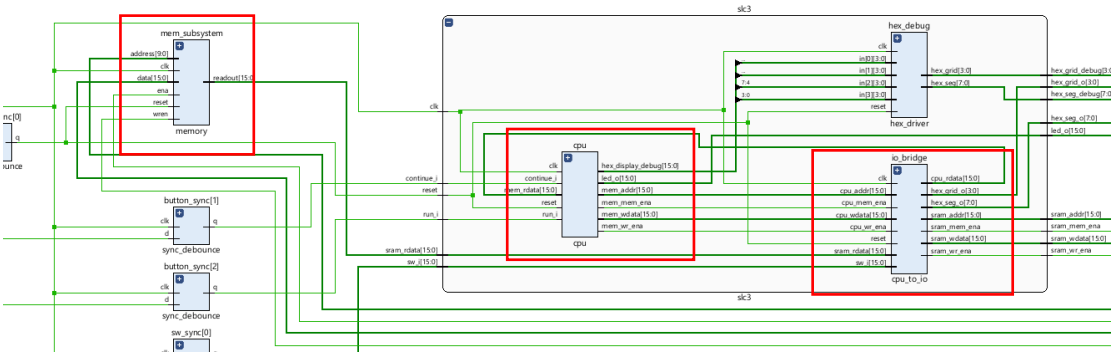

Fig.19 Block diagram for top-level

Written Description of all .sv modules

Module: processor_top.sv

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

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

Description: This is the top-level module of the SLC-3 for both simulation and synthesis using test_memory. The **clk** signal is shared for the lower level modules. In addition, the outputs will enable both the left and right HEX display on board to show values.

Purpose: This module is used as a combination of all other modules in the project for an SLC-3.

Module: sync.sv

Inputs: clk, d

Outputs: q

Description: This module takes the signal d from the fpga button and switch. Moreover, outputs q for other modules to use. Firstly, it flops the input d twice. Then change the button when 2^COUNTER_WIDTH stable input cycles are recorded.

Purpose: This module is used as a debouncer and a synchronizer for push buttons and switches.

Module: slc3.sv

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

Outputs: [15:0] led_o, [7:0] hex_seg_o, [3:0] hex_grid_o, [7:0] hex_seg_debug, [3:0] hex_grid_debug, [15:0] sram_wdata, [15:0] sram_addr, sram_mem_ena, sram_wr_ena

Description: This module serves as a top-level port map that wires up the cpu, cpu_to_io, and one HexDriver. The memory control signals and the memory address input are connected from CPU to physical memory.

Purpose: This module is used as a top-level entity.

Module: cpu.sv

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

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

Description: This module basically contains most of the modules of SLC-3. In other words, all the elements that are within the BUS (shown in the block diagram), as well as the bus.

Purpose: This module is a high-level entity that wires up the internal logic of registers and controls.

Module: cpu_to_io.sv

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

Outputs: [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 connects the two physical I/O devices to the same memory address 0xFFFF (-1). One is for input from the switch and one is for output to hex display.

Purpose: This module manages all I/O with the Urbana board's physical I/O devices.

Module: memory.sv

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

Outputs: [15:0] readout

Description: This module controls the memory. When in synthesis mode, it uses init_ram and sram0. And if in simulation mode, it connects to a test_mem_subsystem.

Purpose: This module manages the interface between the CPU and physical memory.

Module: test_memory.sv

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

Outputs: [15:0] readout

Description: This module is used for simulation only when running the program. It can choose memory contents from an external file or use the default initialization.

Purpose: This module provides a simulation environment for testing the behavior of a memory unit which is similar to the memory on FPGA.

Module: instantiate_ram.sv

Inputs: reset, clk,

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

Description: This module increments the address by 1 if init_mem is high. Moreover, the output data will be the memory contents at address and the addr is assigned to the first 10 bit of address.

Purpose: This module is used to initialize the on-chip memory.

Module: control.sv

Inputs: clk, reset, [15:0] ir, ben, [15:0] mem_rdata, 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, [1:0] pcmux, drmux, sr1mux, sr2mux, addr1mux, [1:0] addr2mux, [1:0] aluk, mio_en, mem_mem_ena, mem_wr_ena

Description: This is the control unit for the whole circuit. This module defines the FSM and state transition logic with 2 blocks of always statements.

Purpose: Output the LD, gate_control, mux_selector, and some other signals to control the operation of other modules. See **Control Unit** for details.

Module: load_reg.sv

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

Outputs: [DATA_WIDTH-1:0] data_q

Description: This module implements a register with a parameter to define the length of bit that it can store. When reset is high, the register is set to 0. When load is high the data_i is loaded into the register, however, the output is not shown until a rising edge of clk signal.

Purpose: This module provides a general register with modifiable data width.

Module: hex_driver.sv

Inputs: clk, reset, in[4]

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

Description: The module uses a loop to instantiate four nibble_to_hex converters, mapping each of the 4-bit inputs to their corresponding 7-segment display patterns. Then choose the hex_grid to determine which of the four LEDs is active.

Purpose: Convert binary into hexadecimal and then display them on the LEDs.

Module: types.sv

Inputs: NONE

Outputs: NONE

Description: This module implements the instruction decode types, number of registers, instruction functions, and detailed memory specified to each line in memory space.

Purpose: This module pre-defined the memory contents for program and program testing.

Module: ALU.sv

Inputs: [15:0] SR1_out, [15:0] SR2MUX_out, [1:0] ALUK,

Outputs: [15:0] ALU_out

Description: This module implements the alu logic, it has 4 operations: plus, and, not, pass through. The selector is ALUK.

Purpose: This module is used for logic computations.

Module: ben_reg.sv

Inputs: [15:0] BUS, [15:0] ir, reset, clk, ld_cc, ld_ben,

Outputs: ben

Description: This module implements a register to store NZP flags in SLC-3. The internal nzp flag is defined depending on the instantaneous value on bus. Three flags will be set when BUS contains positive, negative or zero values. Ben is specified in state 32, as ben = ((ir[11] & n) | (ir[10] & z) | (ir[9] & p)). In addition, we only update BEN value when ld_ben is high, and we only update nzp value when ld_cc is high.

Purpose: This module is used to store the value of nzp flags and output a ben signal.

Module: bus.sv

Inputs: [15:0] pc, [15:0] mdr, [15:0] alu, [15:0] mar, gate_pc, gate_mdr, gate_alu, gate_marmux,

Outputs: [15:0] BUS

Description: This is somehow similar to a 4-1 mux. The only differences is there are four selectors, and none of the selectors should be high at same time. Each selector corresponds to its input, if the selector is high, then BUS contains that value.

Purpose: This module contains the BUS of the whole CPU. Four inputs could be filled in the BUS, chosen by four gates.

Module: mux2.sv

Inputs: [width - 1:0] d0, d1, s

Outputs: [width - 1:0] y

Description: There are 2 various inputs with length specified by width, and the selector selects from the 2 inputs and takes it as its output. Output length should be the same as input.

Purpose: As the name suggested, it is a 2-1 mux with changeable length input and output.

Module: mux4.sv

Inputs: [width - 1:0] d0, d1, d2, d3, [1:0]s

Outputs: [width - 1:0] y

Description: There are four various inputs with length specified by width, and the two selectors select from the four inputs and take it as its output. Output length should be the same as input.

Purpose: As the name suggested, it is a 4-1 mux with changeable length input and output.

Module: resigter.sv

Inputs: clk, ld_reg, reset, [15:0] BUS, [2:0] DR, [2:0] SR1, [2:0] SR2

Outputs: [15:0] SR1_out, [15:0] SR2_out

Description: The module contains R0-R7. If reset is high, all register will be cleaned. If ld_reg is high, BUS value will be loaded into register specified by DR. SR1_out and the selector SR1 and SR2 determined SR2_out.

Purpose: This module contains the 7 general purpose register that was used to perform instructions. It can be reset and loaded from BUS value.

Operation of the control unit

The control unit is a central part of the SLC-3. It operates by transitioning through a series of states, each of the states contains various control signals that can enable, disable, or configure the behavior of various components including registers MAR, MDR, IR, PC, and muxes ALU, ALUK.

Initialization

With a reset signal, the control unit enters the halted state, waiting for a run_i signal to start the fetch-decode-execute cycle.

Fetch

Starting with the s_18 state, the control unit initiates the fetch cycle. The current value of the PC is gated to the MAR, and the PC gets incremented. The instruction at the memory location pointed MAR is then fetched into the MDRover a series of wait states (s_33_x states) to account for memory access latency. Finally, the fetched instruction is loaded into the IR in the s_35 state. After fetching the instruction, the control unit decodes the instruction in the s_32 state by examining the opcode, and then sets/compares NZP and BEN values.

Execution

Based on the opcode, the control unit transact to a specific state or a series of states to perform specific execution. For example, an ADD instruction would move the control unit to the s_01 state, where it sets control signals to perform the addition operation in the ALU and then store the result.

Control Signals

Throughout the execution cycle, the control unit sets various control signals that manage data flow and operations within the SLC-3:

- Load Signals (ld_x): These signals control the loading of data into various registers (MAR, MDR, IR, BEN, NZP, GP_REG, PC). These signals were passed into register modules.

- Gate Signals (gate_x): These signals enable the output of specific components onto the data bus (PC, MDR, ALU, MARMUX). These signals were passed into BUS module.
- MUX Signals (x_mux): These signals select the source of data for certain operations or data paths, they were passed into mux modules.
- ALU Control (aluk): Determines the operation performed by the ALU (add, and, not, pass).
- Memory Control: Controls access to memory, including enabling memory operations and write-enable signals.

State Diagram of Control Unit

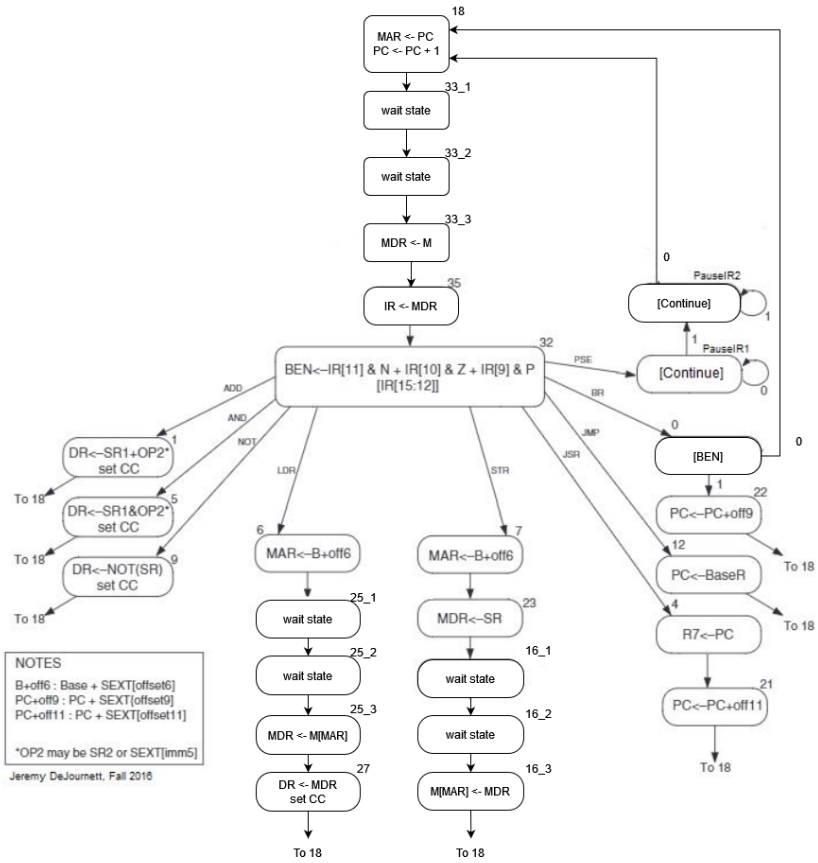

Fig.20 State Diagram for Control Unit

Breadboard View

*Note that hex value stands for the left hex display value.

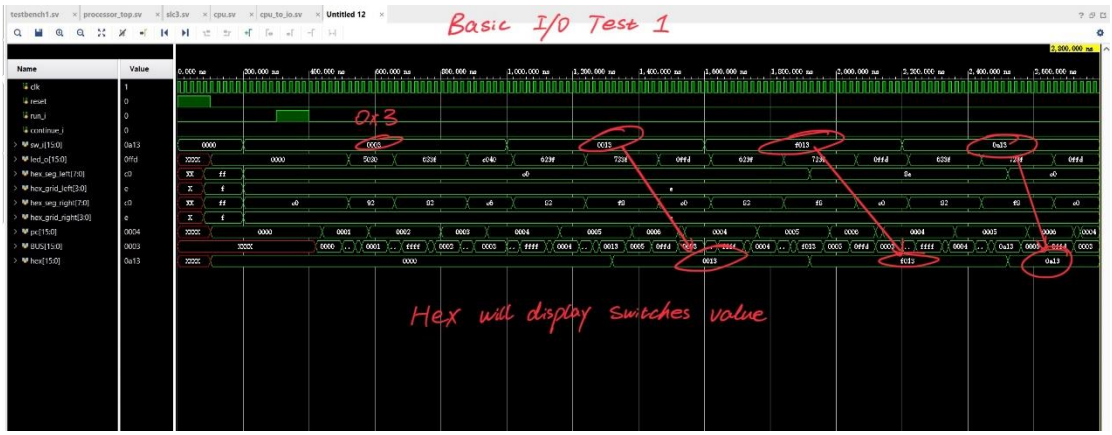

Fig.21imulation for I/O 1

Basic I/O Test 1: The switches were set to 0x0003. Then we press run to enable the execution. As the graph shown, hex value will reflect the switches value.

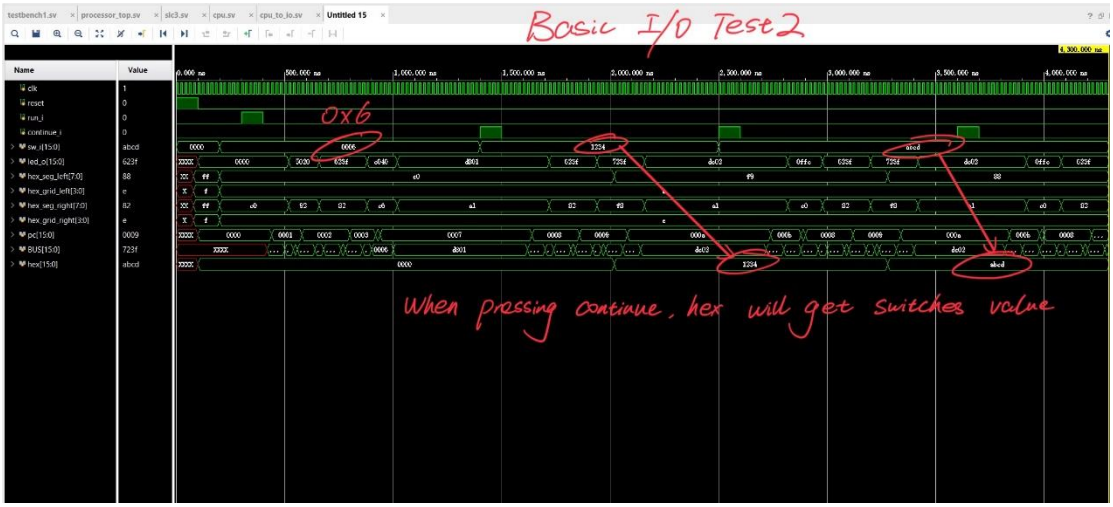

Fig.22 Simulation for I/O 2

Basic I/O Test 2: The switches were set to 0x0006. Then we press run to enable the execution. As the graph shown, hex value will reflect the switches value when we press continue button.

Fig.23 Simulation for self-modifying test

Self-Modifying test: The switches were set to 0x000B. Then we press run to enable the execution. As the graph shown, right LED will increase every time when we press continue.

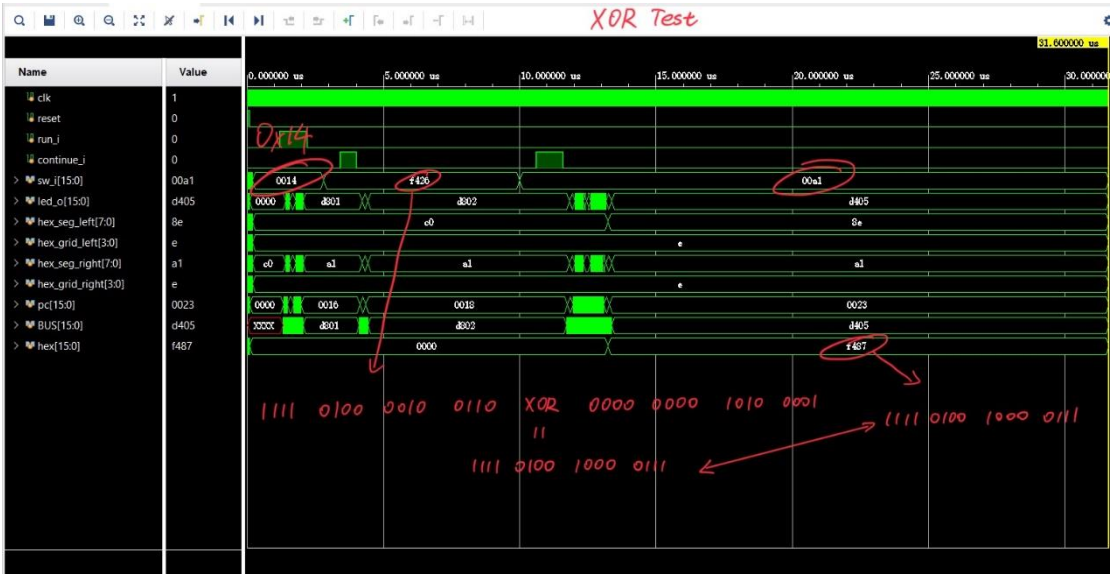

Fig.24 Simulation for XOR test

XOR test: The switches were set to 0x0014, and we press run. We first set the switch to 0xF426 and we press continue. Second time we set the switches to 0x00A1 then press continue. XOR these two values, the result is indeed 0xF487.

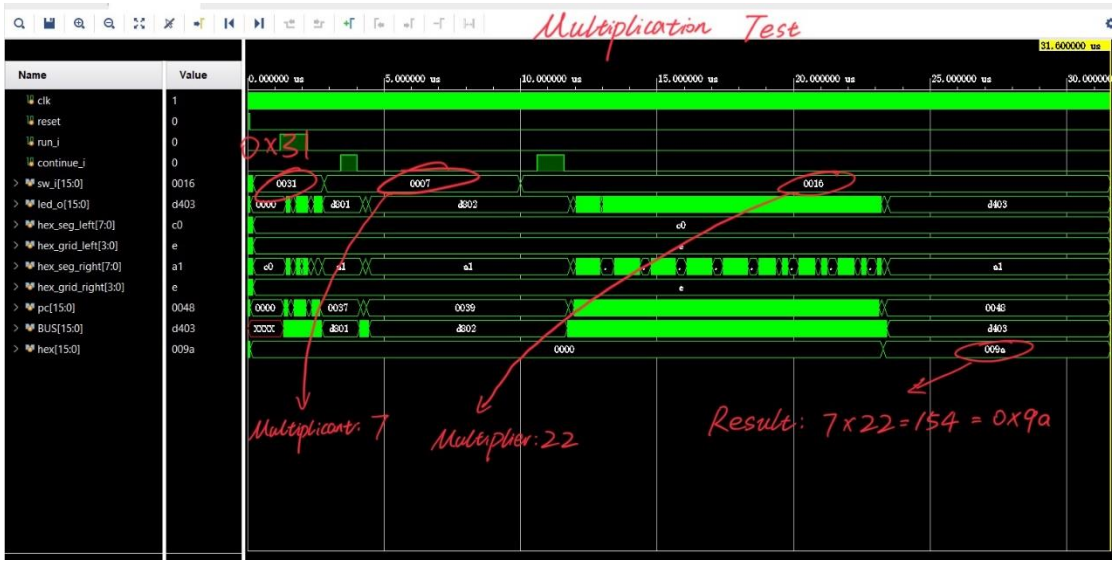

Fig.25 Simulation for Multiplicand test

Multiplicand test: The switches were set to 0x0031, and we press run. We first set the switch to 0x0007 and we press continue. Second time we set the switches to 0x0016 then press continue. Multiply these two values, the result is indeed 0x009A.

Fig.26 Simulation for Auto-counting test

Auto counting test: The switches are set to 0x009C. After we press run, the hex will self-increment.

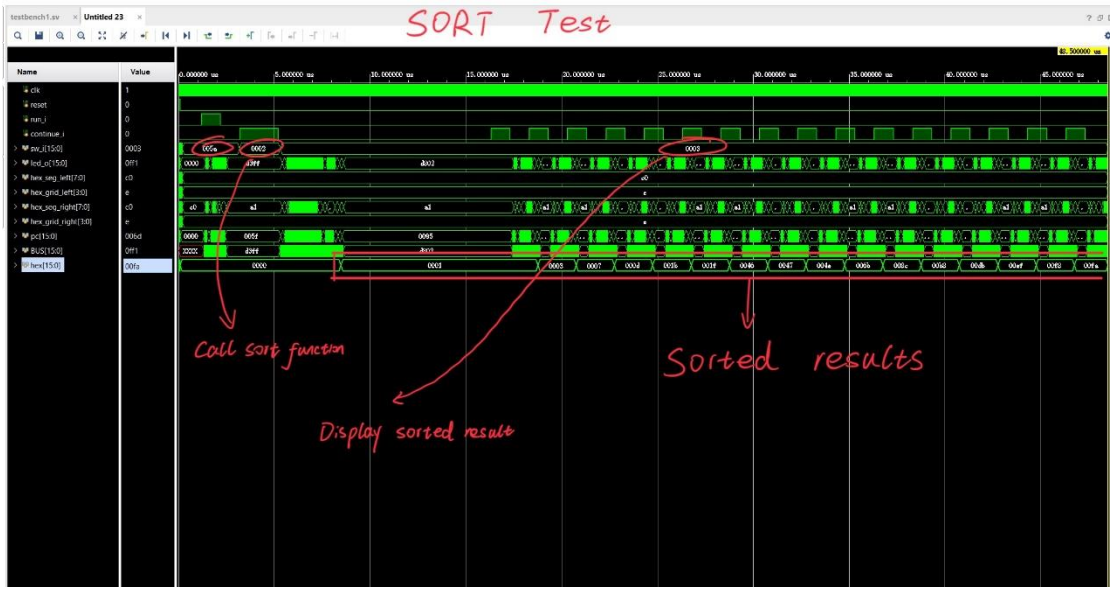

Fig.27 Simulation for Sort test

Sort test: The switches are set to 0x005A, and we press run. At the first checkpoint, we set the switches to 0x0002, and then press continue which is calling the “sort” function. At the second checkpoint, we set the switches to 0x0003, and then press continue which is calling the “display” function. The result after sorting will show up one by one in increasing order when pressing continue buttons.

Fig.28 Simulation for Act-Once test

Act-Once: The switch is set to 0x002A, and we press run. For each press on continue, the hex will increment by one starting from 0x0000.

Bugs Encountered

During our synthesis and simulation process, we came across several bugs. Some of the provided tests required significantly more clock cycles than the standard test bench. Initially, this led us to believe there was an error in our approach; however, we discovered that everything functioned correctly on the Urbana board. By increasing the number of clock cycles in the testbench, we were able to resolve the issue.

Post Lab Question

SLC3	
LUT	411
DSP	0
Memory (BRAM)	1
Flip-Flop	469
Latches	0
Frequency (GHZ)	0.107
Static Power (W)	0.084
Dynamic Power (W)	0.013
Total Power (W)	0.071

Fig.29 Statistics table

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

CPU_TO_IO acts like a bridge, which connects the two physical I/O devices to the same memory, address 0xFFFF (-1). It also outputs external signals to the CPU and takes control signals from the CPU.

What is the difference between BR and JMP instructions?

BR is similar to a “conditional jump” while JMP is similar to an “unconditional jump”. Additionally, BR’s offset is from the last 9 bits of the argument while JMP’s offset comes from the base 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?

The R signal is intentionally designed for a wait signal which means “Ready” to tell the cpu that memory or register is loaded and ready to do next. To compensate for this lack of signal we create 3 more wait states before we do the load instruction. For instance, we have 33_1, 33_2, 33_3 for state 33 which required an R signal before. This affects the performance by increasing the operation time since we tend to wait longer than it actually needs.

Conclusion

In this lab, we successfully built a small microprocessor based on the SLC-3 architecture with SystemVerilog. We embarked on a detailed exploration of computer architecture, translating theoretical knowledge from ECE120 & 220 into a practical implementation. We used an organized approach, beginning with understanding the specifications and breaking down the design into doable jobs. During the experiment, we get to know the operation of the data bus and the interaction between CPU and memory space. We gained invaluable insights into the inner workings of microprocessors, including the critical roles played by each component within the CPU, memory interfacing, and input/output operations.