ECE385

Fall 2023

Experiment #5

Lab 5

Haoyu Huang Eitan Tse
Lab section: ZY 10/Day & Time: 10.16
Your TA's Name: Yang Zhou

1.Introduction

In this lab, we implement sixteen-bit SLC-3 processor using System Verilog. SLC-3 is a simplified version of the LC-3 ISA that has less instructions and less components compared to LC-3. We added one more Pause instruction to check the code. The processor goes through a cycle of fetching, decoding, then executing the instructions. We can use SLC-3 to do XOR, multiplication, auto-counting and sort.

2. Written description and diagram of SLC-3

summary of operation

SLC-3 will go through the circle of fetching, decoding and executing. It can decode LC3 language. SLC-3 has used many FSM to implement different instructions. In general, it will fetch the instruction, go to the specific state according to opcode, and execute the instruction with the signal of the state.

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 program through fetch-decode-execute. SLC-3 can be divided to the ISDU, DATAPATH, and the MEM2IO modules. The ISDU controls the state of the circuit, the datapath controls differernt component in LC-3 like ALU, REG, MUX, and the MEM2IO controls the memory. For the state part, the program begins in the halt state, and after reset state it will wait for the run signal. When a run signal is high the program will go to Fetch which starts in 18 states.

In fetch state the ISDU provides the signals to load PC->MAR, increment PC, load M[MAR]-> MDR, and MDR -> IR. Since we don't have R signal to wait for MEM2IO. We use an additional wait state to wait signal. These signals will go to the DATAPATH to control registers, MUXes, and adders. The datapath has some MUXes, combinational logic and registers (PC, MAR, MDR, IR are registers in datapath) which can implement the fetch cycle. For example, we have PCMUX which control PC to increase itself. Besides, the control signals and the datapath will interface with the MEM2IO to get the required data. For example, when we implement M[MAR]-> MDR, ISDU will provide a control signal. If we have MAR, the MEM2IO will give back MDR through datapath.

The decode state depends on the value of IR we will jump to different function with the help of ISDU. In state 32, we jump to a different state depending on opcode.
The execution state is unique in different states, but essentially consists of unique cases that have control signals matching what is needed to load, add, or whether to write or read. Different functions may have different states and can require inputs pause state but they will return to state 18 and run for the next line. LD_LED is set in pause state. LD_CC is set in some function during execution. LD_BEN is set in state 32.

Fetch (Reset signal is 0):

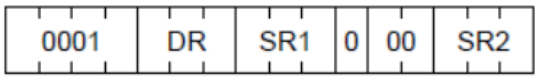
S_18	GatePC, LD_MAR
S_33_1	Mem_OE
S_33_2	Mem_OE
S_33_3	Mem_OE, LD_MDR
S_35	GateMDR, LD_IR

Decode (Reset signal is 0):

S_32: LD_BEN

Instruction that processor can perform (Reset signal is 0):

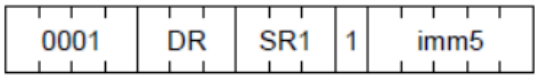
ADD



ADD DR, SR1, SR2

DR ← SR1 + SR2, Setcc

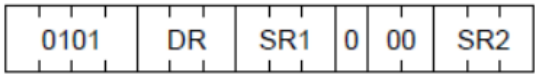
ADD



ADD DR, SR1, *imm5*

DR ← SR1 + SEXT(imm5), Setcc

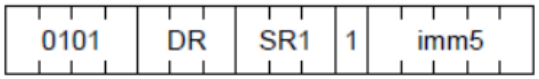
AND



AND DR, SR1, SR2

DR ← SR1 AND SR2, Setcc

AND



AND DR, SR1, *imm5*

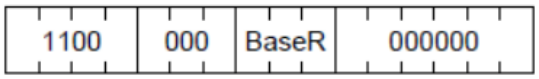
DR ← SR1 AND SEXT(imm5), Setcc

BR

BR{nzp} *PCoffset9*

((n AND N) OR (z AND Z) OR (p AND P)):
PC ← PC + SEXT(PCoffset9)

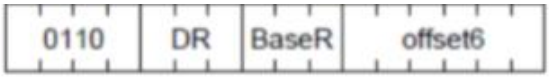
JMP



JMP BaseR

PC ← BaseR

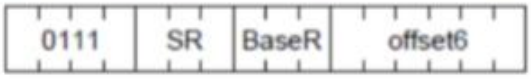
LDR



LDR DR, BaseR, *offset6*

DR ← M[BaseR + SEXT(offset6)], Setcc

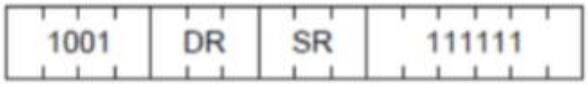
STR



STR SR, BaseR, *offset6*

M[BaseR + SEXT(offset6)] ← SR

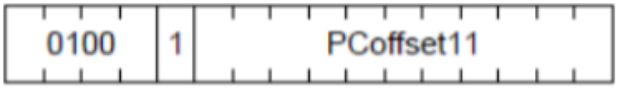
NOT



NOT DR, SR

DR ← NOT SR, Setcc

JSR



JSR *PCoffset11*

R7 ← PC, PC ← PC + SEXT(PCoffset11)

Opcode: 0001 ADD
Depending on IR5 SLC3 professor can add value which store in two registers or add value in register with offset.
S_01: SR2MUX(IR_5) SR1MUX, ALUK, GateALU, LD_REG, LD_CC, LD_BEN

Opcode: 0101 AND
Depending on IR5 SLC3 professor can add value which store in two registers or add value in register with offset.
S_05: SR2MUX(IR_5) SR1MUX, ALUK, GateALU, LD_REG, LD_CC, LD_BEN

Opcode: 1001 NOT
Negates SR and stores the result to DR. Sets the status register.
S_09: SR2MUX(IR_5) SR1MUX, ALUK, GateALU, LD_REG, LD_CC, LD_BEN

Opcode: 0000 BR Branch.
If any of the condition codes match the condition stored in the status register, take the branch; otherwise, continues execution. (An unconditional jump can be specified by setting NZP to 111.) Branch location is determined by adding the sign extended PCoffset9 to the PC.
S_00: ALL RESET
S_22: LD_PC, ADDR2MUX(2'b10), PCMUX(2'b10)

Opcode:1100 JMP
Copies memory address from BaseR to PC.
S_12: LD_PC, SR1MUX, ADDR1MUX, ADDR2MUX, PCMUX(2'b10)

Opcode: 0100 JSR
Jump to Subroutine. Stores current PC to R(7), adds sign extended PCoffset11 to PC.
S_04: GatePC, LD_REG, DRMUX
S_21: LD_PC, PCMUX(2'b10), ADDR2MUX(2'b11)

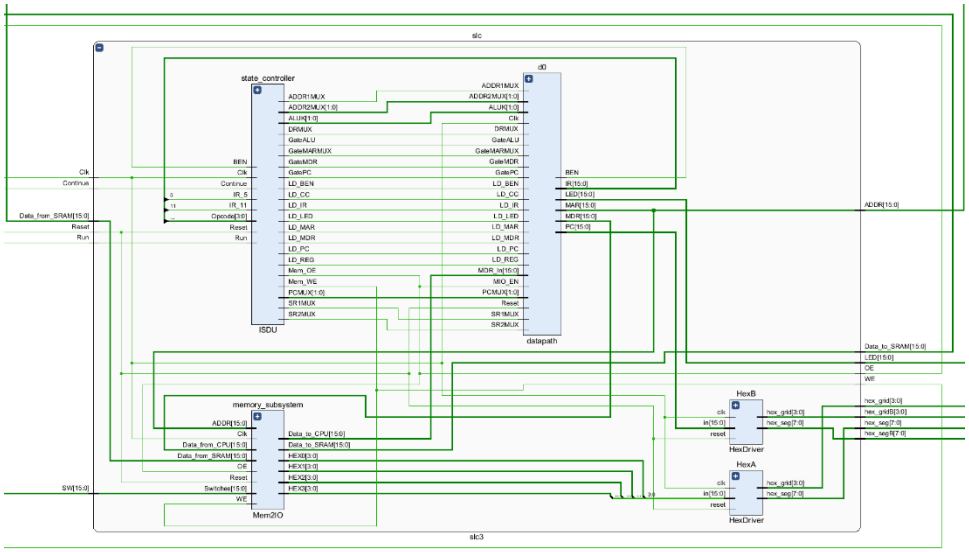
Opcode: 0110 LDR
Load using Register offset addressing. Loads DR with memory contents pointed to by (BaseR + SEXT (offset6)). Sets the status register.
S_06: ADDR1MUX, ADDR2MUX, GateMARMUX, LD_MAR, SR1MUX
S_25_1: Mem_OE
S_25_2: Mem_OE
S_25_3: Mem_OE, LD_MDR
S_27: LD_REG, LD_BEN, LD_CC, GATE_MDR

Opcode: 0111 STR

Store using Register offset addressing. Stores the contents of SR at the memory location pointed to by (BaseR + SEXT(offset6)).
S_07: ADDR1MUX, ADDR2MUX, GateMARMUX, LD_MAR, SR1MUX
S_23: LD_MDR, GATE_ALU, ALUK
S_16_1: Mem_OE, MEM_WE
S_16_2: Mem_OE, MEM_WE
S_16_3: Mem_OE, MEM_WE

Opcode: 1101 PAUSE
Pauses execution until Continue is asserted by the user. Execution should only unpause if Continue is asserted during the current pause instruction; that is, when multiple pause instructions are encountered, only one should be "cleared" per press of Continue. While paused, ledVect12 is displayed on the board LEDs. See I/O Specification section for usage notes.
PAUSE: LD_LED

Block Diagram of SLC3.sv



Written Description of .sv Modules

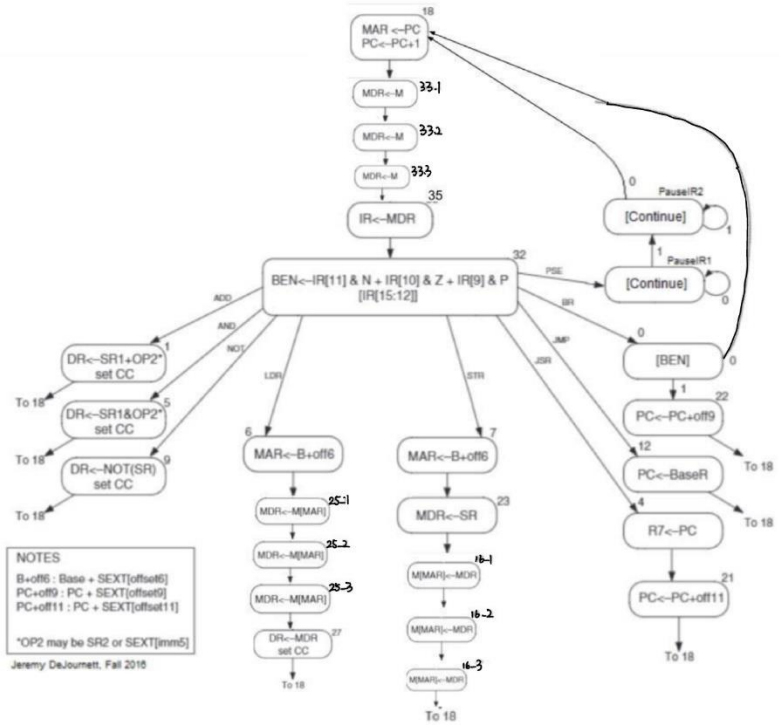
Module: ISDU.sv
Inputs: Clk, Reset, Run, Continue, Opcode[3:0], IR_5, IR_11, BEN

Outputs: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, PCMUX[1:0], DRMUX, SR1MUX, SR2MUX, ADDR1MUX, ADDR2MUX[1:0], ALUK, Mem_OE, Mem_WE

(For the detail part of how the signal change in ISDU, I have written in what instruction can professor perform)

Description: This module is the control unit that implements the state diagram of SLC-3. The module follows the fetch-decode-execute cycle. In different states, we will have certain signals activated which are passed into the datapath so the instruction can be executed. The module is organized as 2 always blocks. One is to decide the next state, the other is to decide the signal that is activated in the state. We use Enum to represent different state code. It will begin at halted state and will start at S_18 if user presses RUN button. Then it will begin Fetch operation. Then we come to S_32 which is decode state. According to different instruction(opcode) stored in IR. ISDU will jump to state based on different opcode. We have use case statement to connect state with opcode. IR 5 is used in the ADD and AND instruction to the value in register as an offset or. IR 11 is used in JSP operation. Input BEN is used in BR state (S_22). For Pause state, ISDU will come to pause after decoding, Pauses execution until Continue is asserted by the user. Execution should only unpause if continue is asserted during the current pause instruction; that is, when multiple pause instructions are encountered, only one should be "cleared" per press of Continue. While paused, ledVect12 is displayed on the board LEDs. If the user presses the Reset button, it will go back to halted state.

State diagram:



Purpose: The purpose of this module is to implement the FSM for SLC-3. The ISDU will generate different control signals in different states which will control data flow. It also

controls Fetch-Decode-Execute cycle and decides which state we will go to. With ISDU, SLC-3 can store, load value from memory, write to memory and execute instructions stored in the memory

Module: slc3.sv
Inputs: SW[15:0], Clk, Reset, Run, Continue, Data_from_SRAM[15:0]
Outputs: LED[15:0], OE, WE, hex_seg[7:0], hex_segB[7:0], hex_grid[3:0], hex_gridB[3:0] ADDR[15:0], Data_to_SRAM[15:0]
Description: This module defines many signals and instantiates other modules such as the datapath, Mem2IO and ISDU.
Purpose: This module is like the top-level of many sub-modules of SLC processor

Module: datapath.sv
Inputs: Clk, Reset, Run, Continue, LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, GatePC, GateMDR, GateALU, GateMARMUX, SR2MUX, ADDR1MUX, MARMUX, MIO_EN, DRMUX, SR1MUX,
[1:0] PCMUX, ADDR2MUX, ALUK, [15:0] MDR_In
Outputs: BEN, [15:0] MAR, MDR, IR, LED
Description: The datapath module is the module that transfers data between different modules. The inputs of datapath are mainly controled by ISDU,which will change depending on the state. The LD signals are used to load specific registers like PC, MAR, MDR. We instantiate different MUX and registers in DataPath. We implement BUS as a MUX which will take value from different gates. The bus will connect IR, MDR, PC, and ALU. Besides, we used one always_comb block to implement logic like increasing PC address, assigning NZP_com.

Module: register.sv
Parameter w=16
input logic Clk, Reset, Load, [w-1:0] In,
output [w-1:0] Out
Description: The module implements register by setting default bit length to 16. When we instantiate registers, there are all kinds of registers which need different input. W-bit register and have Synchronous reset.
Purpose: The module is used to store data for PC, IR, MDR, MAR. It has Synchronous Reset and will load value input with positive clock signal.

Module: MUX.sv
The module contains 3 MUX modules
1.MUX_BUS.sv
input [3:0] sel, [15:0] PC_bus, [15:0] MAR_bus, [15:0] MDR_bus, [15:0] ALU_bus
Output 15:0] BUS

Description: The module implements a 4-1 Mux which has 15-bit. It is the body of the bus which selects value among different gates.
Purpose: We instantiate bus in datapath.sv

2.MUX2.sv
Parameter w=16
input sel, [w-1:0] A, [w-1:0] B
Output [w-1:0] C
Description: The module implements a 2-1 Mux which has w-bit. It will select value among A, B and C.
Purpose: We instantiate MUX2 (DRMUX, SR2MUX, SR1MUX) in datapath.sv

3.MUX4.sv
input [1:0]sel, [15:0] A, [15:0] B
Output [15:0] C
Description: The module implements a 4-1 Mux which has 15-bit. It is slightly different from MUX bus. We have 2-bit sel signal. It is the body of PCMUX, MUX_adder2.
Purpose: We implement 4-1 MUX.

Module sext1.sv
input [15:0] IR,
output [15:0] sext11, [15:0] sext9, [15:0] sext6, [15:0] sext5
Description: The module concatenates to extend sext to 15 bits. Since IR is signed signal, we need to use if statement to check the bit.
Purpose: The module is used to get the extend bit of partial IR. IR is considered as signed value we need to extend 1 or 0 according to its first bits.

Module:ALU.sv
input [15:0] A, [15:0] B, [1:0] ALUK,
output [15:0] Out
Description: The module will compute A+B (ALUK: 00), A+B(ALUK:01), ~A(ALUK:10), A(ALUK:11) by selecting signal ALUK. It implements the ALU part in LC3 datapath.
Purpose: The module will be instantiated in datapath and can be used to transfer values in register, adding two values in register.

Module: REG_file.sv
input Clk, Reset, DR [2:0] SR1_m, [2:0] SR2_m, [15:0] bus_data, LD_REG,
output [15:0] SR1_out, [15:0] SR2_out
Description: This module is used as the register file where R0-R7 are held. This is a synchronous module which has Clk and Reset input. The reg_file uses DR as destination register, and SR1 and SR2 dictate the output.The module holds 15-bit [7:0] array which

indicates 8 different registers. We first use a for loop to reset the array and use another for loop to load the data to a specific register.
Purpose: This module is used as the register unit in the datapath of SLC3 implementation.

Module: Ram
Description: This is not a .sv module but an IP module. We want to introdue its ports. douta is16-bit data bus in/out from the RAM. addra is 10-bit Address bus (in LC-3, the address space is only 16-bit wide, so the addresses take only the 10 least significant bits) ena is Read enable signal. When active, it allows read operations. Active high. wea Write enable signal. When active, it allows write operations. Active high. clka is Clock signal, FPGA BRAMs are synchronous.

Module: SLC3_2.sv
Inputs: N/A
Outputs: N/A
Description: The module implements nine instructions by assigning opcode to each function. The module creates the functions for each of the nine operations.
Purpose: The module functions as bridge between LC3 and systemverilog.

Module: sync
input Clk, d
output q
Description: The module takes in CLK and d. It will use always_ff block to assign d to q.
Purpose: These are synchronizers required for bringing asynchronous signals into the FPGA synchronizer with no reset （for switches/buttons）

Module: Hexdriver
Inputs: Clk, Reset, [3:0] in [4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: The module takes output bits of registers in the register unit. It will change 3-bits into Hex number and display on the FPGA. The input is array type (each 4-bits) which can take in 16 bits in total. It can convert upper 4bits and lower 4bits of register to each hex display respectively. Output is 8-bits hex_seg and 4-bits hex_grid. Hex_seg is used to display numbers in LED. Every LED has 7 lights. Number can be expressed in 7 bits in LED. For example, 0 can be expressed as 11000000. Hex_seg [7] is constant 1 in this case. Since FPGA has multiple HEX displays, hex_grid will instruct the board to use specific displays.
Purpose: The module is used to display Hex numbers on FPGA boards, allowing the user to see the contents of registers A and B.

Module: slc3_sramtop
Inputs: SW [15:0], Clk, Run, Continue

Outputs: LED [15:0], hex_seg[7:0], hex_segB[7:0], hex_grid[3:0], hex_gridB[3:0]
Description: This is the top-level module provided to us to use for when we program the FPGA. We instantiate physical on-chip memory and instantiateram here.
Purpose: SLC-3 Top level module for synthesis using physical RAM

Module: slc3_testtop
Inputs: SW [15:0], Clk, Run, Continue
Outputs: LED [15:0], hex_seg[7:0], hex_segB[7:0], hex_grid[3:0], hex_gridB[3:0]
Description: This is the top-level module provided to us to use for when we run simulation and use for test
Purpose: SLC-3 Top level module for both simulation and synthesis using test memory

Module Mem2IO.sv
Inputs: Clk, Reset, ADDR [15:0], OE, WE, Switches [15:0], Data_from_CPU[15:0], Data_from_SRAM[15:0]
Outputs: Data_to_CPU[15:0], Data_to_SRAM[15:0], HEX0[3:0], HEX1[3:0], HEX2[3:0], HEX3[3:0]
Description: This module interfaces with the I/O for SLC-3. It can help to read and write data to memory and display the hex data and LED.
Purpose: The purpose of this module is to allow for I/O of the SLC-3 processor and SRAM. It connects the data between memory, Datapath and ISDU.

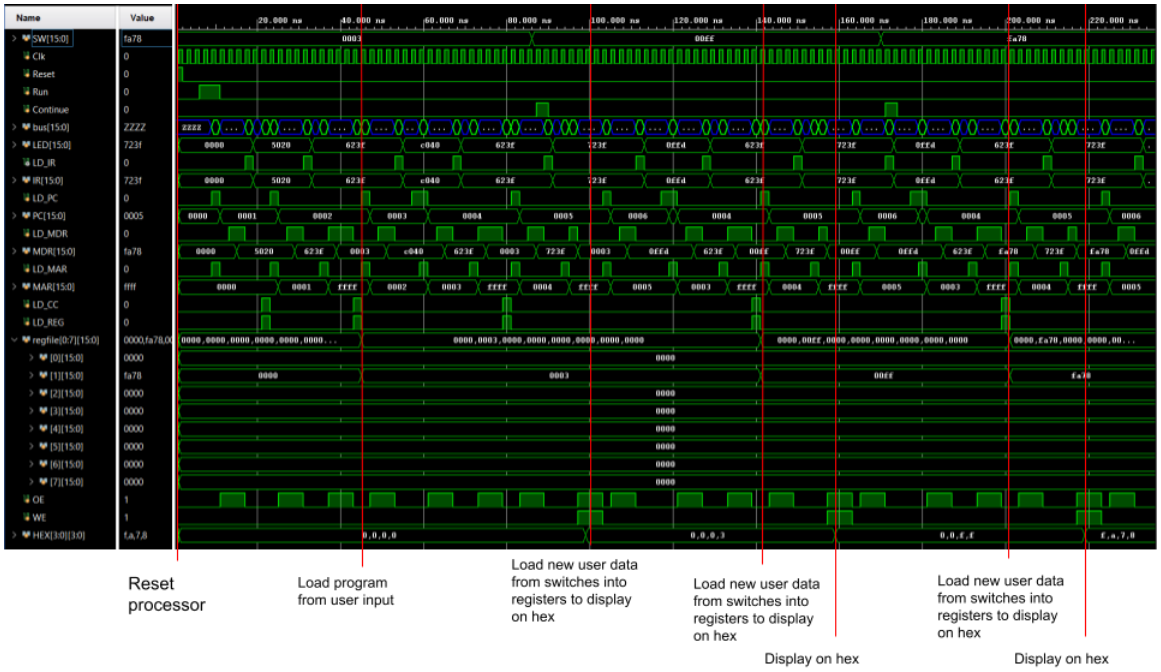
Module: Instantiateram.sv
Input: Reset, Clk,
output [15:0] ADDR, wren, data
Description: The module instantiates memory in many arrays. It is like writing LC3 code. We have an address signal inside module which will go through every line. The LC3 code will be interpreted using SLC2.sv and will be stored as output data.
Purpose: It is used to initiate on-chip memory.

Module: test_memory.sv
input Reset, Clk, datata [15:0], address [9:0], ena, wren,
output [15:0] readout
Purpose: The module instantiates test memory.

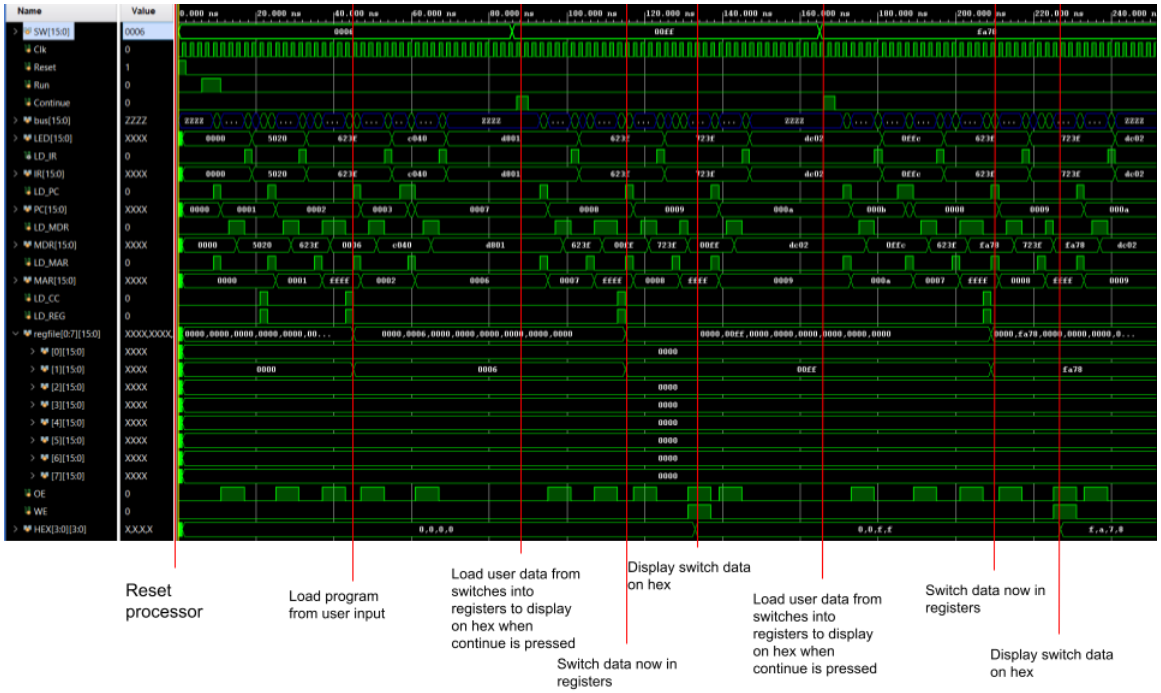
Module: memory_parser.sv
input: NA
output: NA
Description: Instantiates an array of memory locations that can be used to simulation.
Purpose: It has the same memory as on-chip memory which can help to test simulation. It is used for simulation purposes only, to replace SDRAM.

5. Simulations

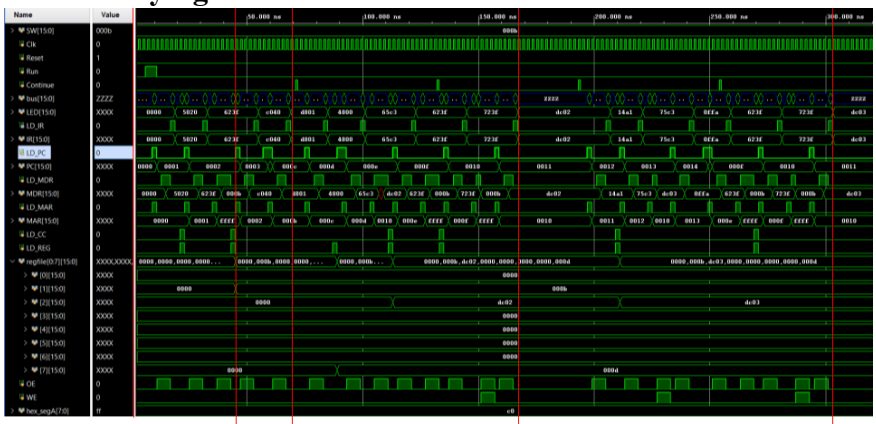
User I/O Test 1



User I/O Test 2



Self-Modifying Code Test



Reset processor

Load program from user input

Wait for continue button to be pressed, LED displays 1 since the lower 10 bits are written to the LEDs.

Wait for continue button to be pressed. LED displays 2 because the lower 10 bits are written to the LEDs.

Wait for continue button to be pressed. LED displays 2 because the lower 10 bits are written to the LEDs.

Program values do not change afterwards because the continue button hasn't been pressed and it stays in a wait state. R2 is used to track the current data to write to the LEDs. It is incremented over the duration of the program, every time the continue button is pressed. R7 stores the PC value at that particular instant, without changing its value.

Auto-Counting Test

Reset processor

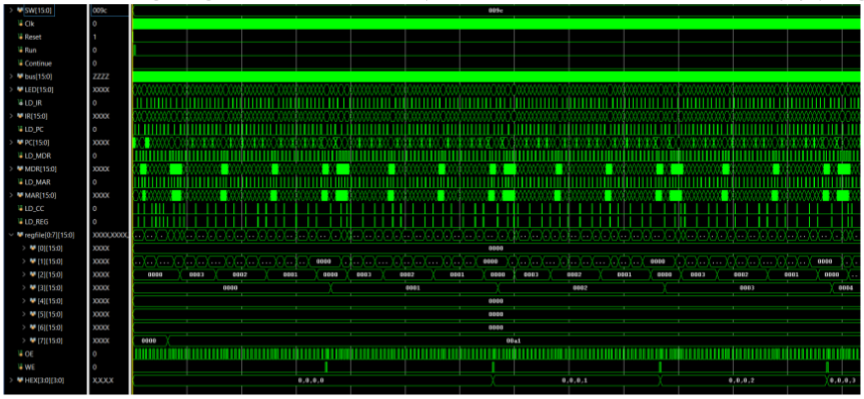
Load program from user input

Reset R1 to use as a counter, R2 and R3 are already reset

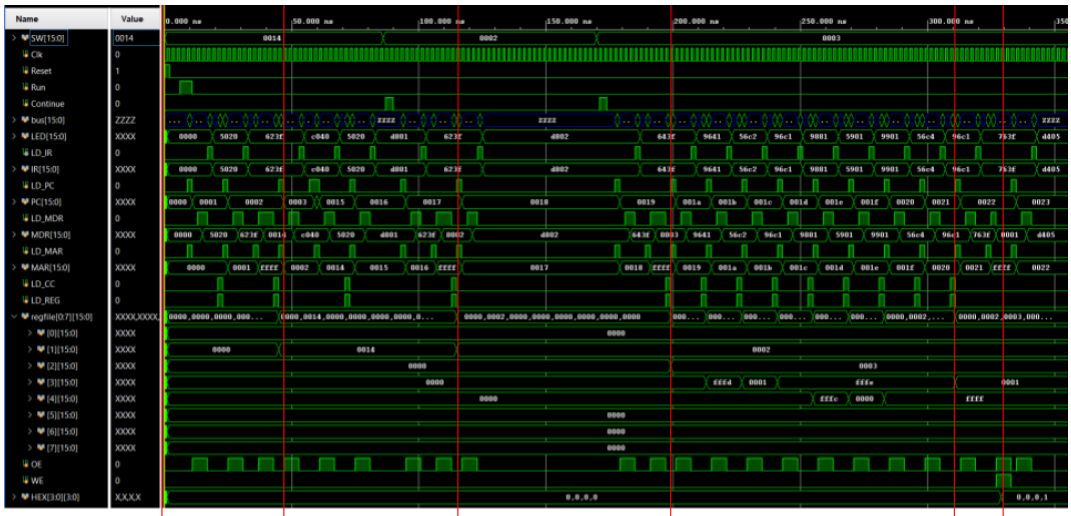
Start R1 counting down

Start R2 counting down

The screenshot is not large enough, so a zoomed-out simulation is provided below to better show the counters and hex display updating.



XOR Test


Reset processor
Load program from user input
Read user input for operand 1 and store in registers
Read user input for operand 2 and store in registers
Start R2 counting down
Write result to hex display

Multiplication Test

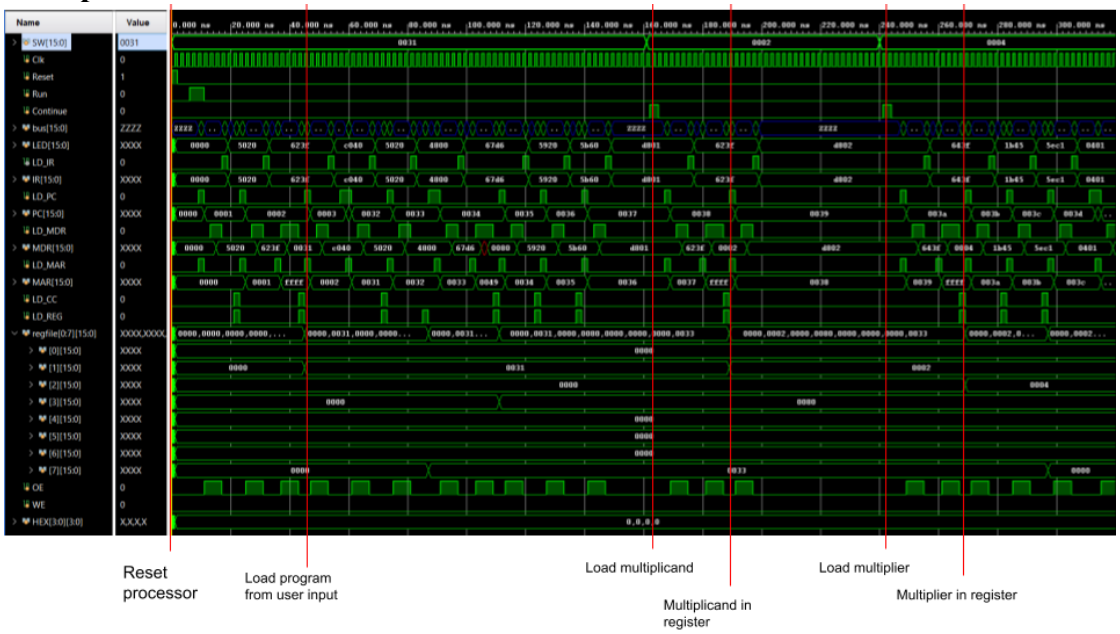
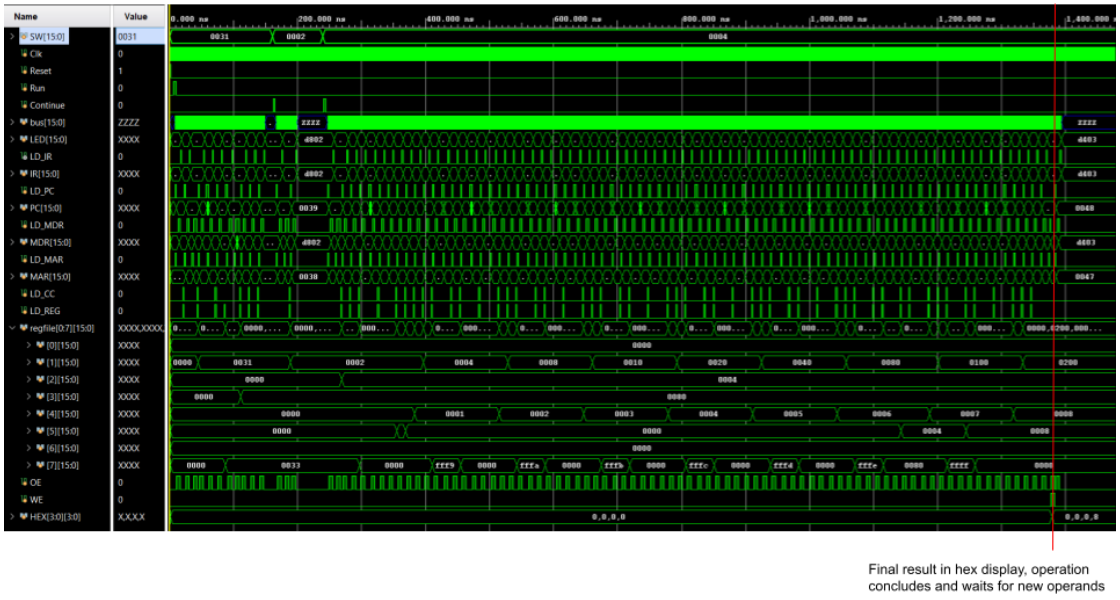


Image is not large enough to encompass the entirety of the algorithm, so a zoomed-out waveform is shown below with the correct result in the hex display. This operation is conducted on 2*4, which produces 8.



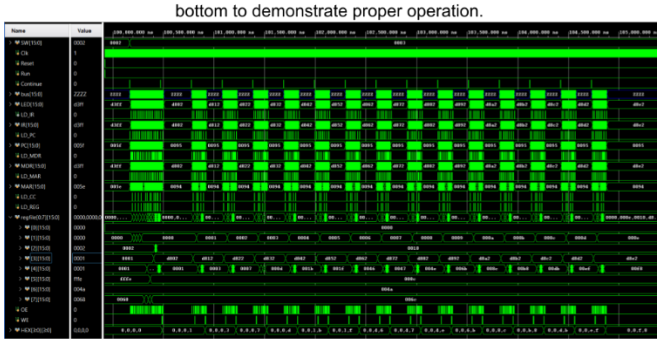
Sort Test

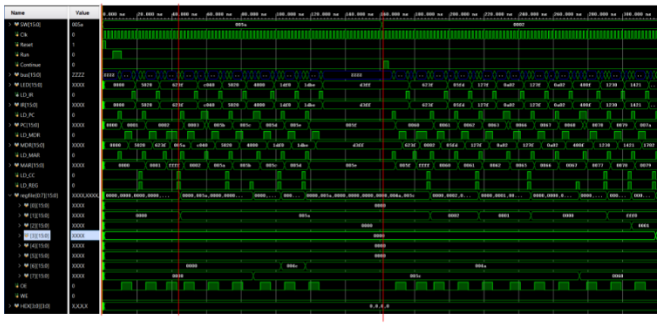
Reset processor

Load program from user input

Load operation from user (display in this case)

Image is not large enough to encompass the entirety of the function. A zoomed-out simulation waveform of the display function is shown here, with hex outputs at the bottom to demonstrate proper operation.



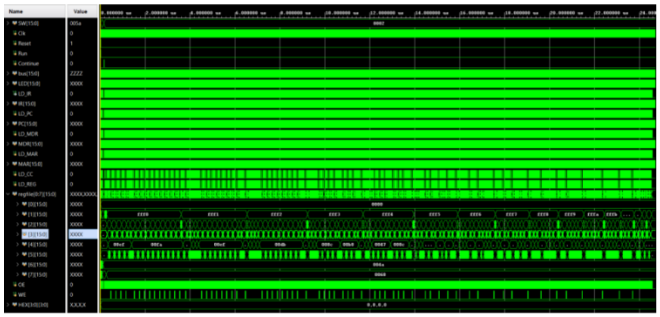


Reset processor

Load program from user input

Load operation from user (display in this case)

Image is not large enough to encompass the entirety of the function. A zoomed-out simulation waveform of the sort function is shown here.



Sort Test Continued

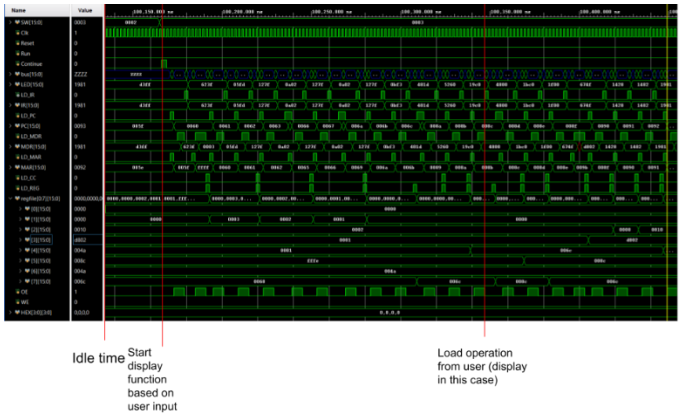
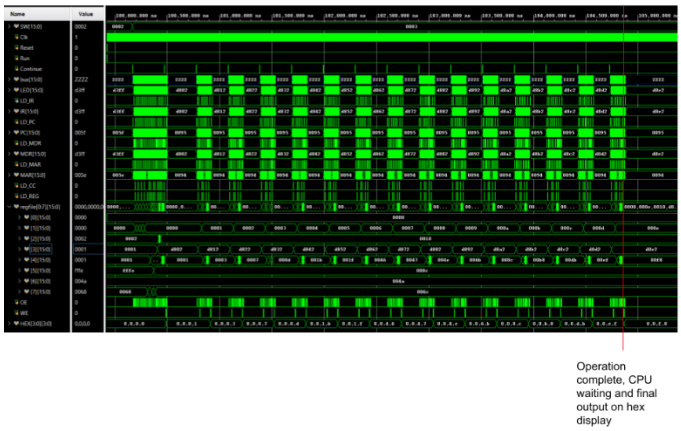


Image is not large enough to encompass the entirety of the function. A zoomed-out simulation waveform of the display function post-sort is shown here, with hex outputs at the bottom to demonstrate proper operation.



7. Answers to Post-Lab Questions

Resource Utilization

LUT	405
DSP	0
Memory (BRAM)	0.50
Flip-Flop	269
Latches	0
Frequency	105.697MHz
Static Power	0.071W
Dynamic Power	0.006W
Total Power	0.077W

What is the MEM2IO used for?
The MEM2IO block acts as a memory interconnect between the initialized BRAM, the processor, and the I/O available on the board. The SRAM is used to store instructions that the processor is to execute. When memory at a certain location (0xFFFF in this case) is attempted to be accessed, the CPU reads from the switches instead of the instantiated BRAM. This is how user input is handled with the SLC-3. The MEM2IO also controls read or write to and from the SRAM using on the OE and WE signals from the processor. It also displays the data from the CPU on the HEX displays when reading from user input. In addition, it has a reset signal for resetting the HEX display.

What is the difference between BR and JMP instructions?
JMP is an unconditional jump to a specified offset from the current program counter. BR either jumps to another point in the program or does not jump based on the BEN 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 implications does this have for synchronization?
The R signal is supposed to be a ready signal from memory, to signal the processor that it is ready to read from or write to. We compensate by adding three wait states every time we want to read or write from memory. This guarantees memory will be ready for read or write by the time the state machine moves the CPU onto performing some action upon the memory. This guarantees that we will not have synchronization issues and reading/writing incorrectly.

8. Conclusions

Haoyu encounters a bug that simulation results are not same as the board performance. I was confused simulation results are perfect but not work on board. I solved the bug by changing the waiting state of my FSM. I found that we must use 3 waiting states to load/write data. However, 2 waiting states are enough for lab5.1. But when it comes to lab5.2, we have more states, and our data flow becomes complex. My device may crash if I only have two waiting states.

Eitan encountered a bug with passing every test except sort and multiply, and found that his ADD/AND/NOT operations were not setting the destination registers correctly. However, all other instructions did so correctly. Once this bug was fixed, his SLC-3 worked perfectly. I also encountered the same bug as Haoyu by using 2 instead of 3 wait states, and was wondering why my data was bad.

Overall, this lab was enjoyable but also a bit challenging. We have gone through what we have learned in ECE220 and combined it with system verilog. It is kind of a challenge to build datapath. Besides, it is good practice to build a state machine and control all signals. We think the lab is pretty good. But I think in the lab manual, the datapath graph is wrong since we didn’t use tri-state component. Besides, in the module there is one signal MARMUX which is useless since we have GATEMAR already.