CS 2110 Final Exam: LC-3 Assembly

Your Friendly TAs ;)

Spring 2022

Contents

Please take the time to read the entire document before starting the assignment. It is your responsibility to follow the instructions and rules.

1 Rules - Please Read

You are allowed to submit this portion of the final exam starting from the moment your timed lab/coding portion of the exam period begins until your individual period ends. You have 75 minutes to complete *both of the timed lab (LC3-Assembly and C)* portions of the exam, unless you have accommodations that have already been discussed with your professor. Gradescope submissions will close precisely at the end of your allotted exam period. **You are responsible for watching your own time.**

If you have questions during the exam, you may ask the TAs for clarification, though you are ultimately responsible for what you submit. The information provided in this document takes precedence. If you notice any conflicting information, please indicate it to your TAs.

The timed lab/coding section of the final exam is open-resource. You may reference your previous homeworks, class notes, etc., but your work must be your own. Contact in any form with any other person besides a TA is absolutely forbidden. **No collaboration is allowed.**

2 Overview

2.1 Purpose

The purpose of this coding section of the final exam is to test your understanding of coding in LC-3 assembly.

2.2 Task

In this section of the final exam, you will be implementing a short assembly program. Please see the detailed instructions on the following page. We have provided pseudocode for the program—you should follow the algorithm when writing your assembly code.

2.3 Criteria

This section will be graded based on your ability to correctly translate the given pseudocode for an algorithm into LC-3 assembly code. Please use the LC-3 instruction set when writing these programs. Check the Deliverables section for what you must submit to Gradescope.

You must produce and save the correct value for the algorithm, and your code must assemble with no warnings or errors (Complx and the autograder will tell you if there are any). If your code does not assemble, we will not be able to grade that file and you will not receive any points.

3 Detailed Instructions

For this coding section of the final exam, you will be implementing an algorithm `rangeSearch` in assembly that looks for the first element in a list that falls in a given range. This program will have a pointer to the start of the list, size, target, and range value via labels, and you will write this element value to the answer label.

3.1 `rangeSearch` algorithm

The `rangeSearch` algorithm will use a pointer to the start of the list, size of the list, a target value, and a range. The algorithm linearly searches through the list and gets the first element that satisfy the requirement: `target-range` ≤ `element` ≤ `target+range`. Assume all elements in the list are non-negative.

The following pseudocode instruction is provided for reference:

```
Given labels: list, size, target, range, answer
Todo: store the correct list element value to answer

int element = -1;
for (int i = 0; i < size; i++) {
    if (target-range <= list[i] <= target+range) {
        element = list[i];
        break;
    }
}
answer = element;
```

Examples:

- Given list: `[2,3,9]`, size: `3`, target: `10`, and range: `2`, the algorithm should store `9` to `answer`.
- Given list: `[0,10]`, size: `2`, target: `5`, and range: `4`, the algorithm should store `-1` to `answer`.
- Given list: `[4,6]`, size: `2`, target: `6`, and range: `2`, the algorithm should store `4` to `answer`.

Please refer to Grading for details on how `rangeSearch` will be graded.

4 Grading

Point distribution for this coding portion of the final exam is broken down as follows:

- `rangeSearch` (25 points): Storing the correct element value to the label `answer`.

5 Deliverables

Turn in the following file on Gradescope during your assigned final exam coding period:

1. `rangeSearch.asm`

6 Local Autograder

To run the autograder locally, follow the steps below depending upon your operating system:

- Mac/Linux Users:
  1. Navigate to the directory your files are in (**in your terminal on your host machine, not in the Docker container via your browser**)
  2. Run the command `sudo chmod +x grade.sh`
  3. Now run `./grade.sh`
- Windows Users:
  1. In Git Bash (or Docker Quickstart Terminal for legacy Docker installations), navigate to the directory your files are in
  2. Run `chmod +x grade.sh`
  3. Run `./grade.sh`

7 LC-3 Assembly Programming Requirements

7.1 Overview

1. Your code must assemble with **NO WARNINGS OR ERRORS**. To assemble your program, open the file with Complx. It will complain if there are any issues. **If your code does not assemble, you WILL get a zero for that file.**

2. **DO NOT assume that ANYTHING in the LC-3 is already zero.** Treat the machine as if your program was loaded into a machine with random values stored in the memory and register file.

3. You can randomize the memory and load your program by going to File > Advanced Load and selecting `RANDOMIZE` for registers and memory.

4. Do NOT execute any data as if it were an instruction (meaning you should put `HALT` or `RET` instructions before any `.fills`).

5. Do not add any comments beginning with @plugin or change any comments of this kind.

6. You should not use a compiler that outputs LC3 to do this assignment.

7. **Test your assembly.** Don't just assume it works and turn it in.

8. **Comment your code!** (not a hard requirement, but will make your life much easier) This is especially important in assembly, because it's much harder to interpret what is happening later, and you'll be glad you left yourself notes on what certain instructions are contributing to the code. Comment things like what registers are being used for and what less intuitive lines of code are actually doing. To comment code in LC-3 assembly just type a semicolon (;), and the rest of that line will be a comment.

   Avoid stating the obvious in your comments, it doesn't help in understanding what the code is doing.

   Good Comment

   ```
   ADD R3, R3, -1          ; counter--
   BRp LOOP                ; if counter == 0 don't loop again
   ```

   Bad Comment

   ```
   ADD R3, R3, -1          ; Decrement R3
   BRp LOOP                ; Branch to LOOP if positive
   ```

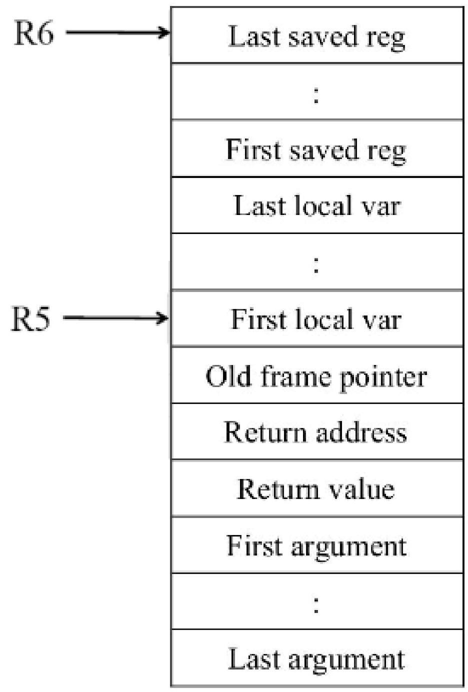
8 Appendix

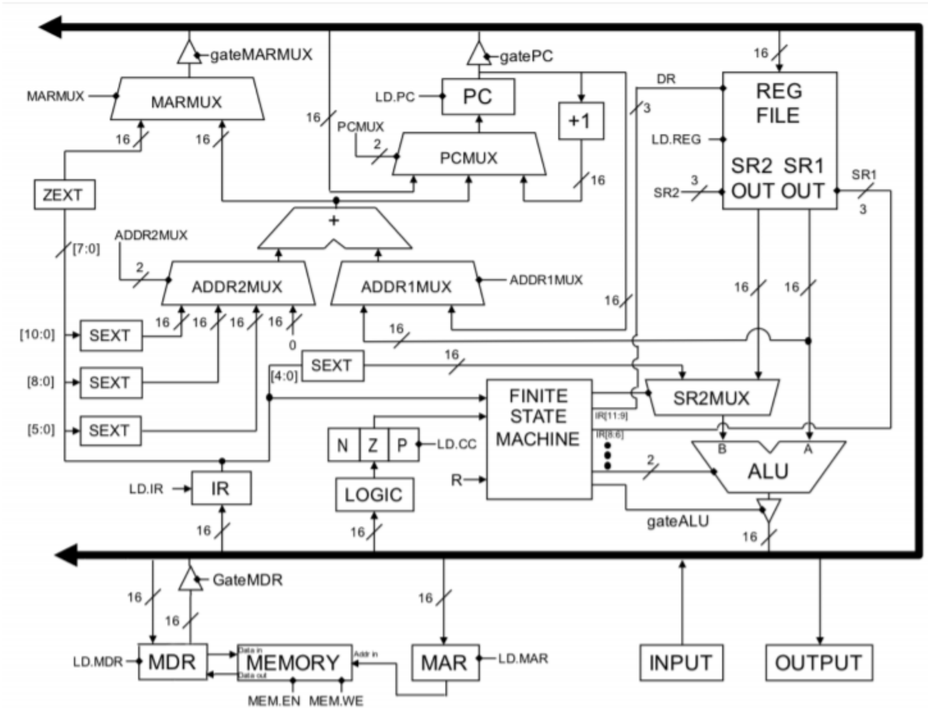
8.1 Appendix A: LC-3 Instruction Set Architecture

Instruction						
ADD	0001	DR	SR1	0	00	SR2
ADD	0001	DR	SR1	1	imm5	
AND	0101	DR	SR1	0	00	SR2
AND	0101	DR	SR1	1	imm5	
BR	0000	n	z	p	PCoffset9	
JMP	1100	000	BaseR	000000		
JSR	0100	1	PCoffset11			
JSRR	0100	0	00	BaseR	000000	
LD	0010	DR	PCoffset9			
LDI	1010	DR	PCoffset9			
LDR	0110	DR	BaseR	offset6		
LEA	1110	DR	PCoffset9			
NOT	1001	DR	SR	111111		
ST	0011	SR	PCoffset9			
STI	1011	SR	PCoffset9			
STR	0111	SR	BaseR	offset6		
TRAP	1111	0000	trapvect8			

Trap Vector	***Assembler Name***
x20	GETC
x21	OUT
x22	PUTS
x23	IN
x25	HALT

Device Register	***Address***
Keybd Status Reg	xFE00
Keybd Data Reg	xFE02
Display Status Reg	xFE04
Display Data Reg	xFE06





Boolean Signals	
LD.MAR	GateMARMUX
LD.MDR	GateMDR
LD.REG	GatePC
LD.CC	GateALU
LD.PC	LD.IR
MEM.EN	MEM.WE

MUX Name	Possible Values
ALUK	ADD, AND, NOT, PASSA
ADDR1MUX	PC, BaseR
ADDR2MUX	ZERO, offset6, PCoffset9, PCoffset11
PCMUX	PC+1, ADDER, BUS
MARMUX	ZEXT, ADDER
SR2MUX	SR2, SEXT