CS 2110 Timed Lab 3: Assembly with Subroutines

Your TAs

Fall 2020

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 Timed Lab Rules - Please Read

1.1 General Rules

1. You are allowed to submit this timed lab starting at the moment the assignment is released. Your submission is due by the end of your lab period, unless you have scheduled an alternate time. Note that Gradescope will remain open for all sections, but you are responsible for watching the clock and submitting on time. **Submitting or resubmitting the assignment after time is up is a violation of the honor code - intentionally doing so may incur a zero on the assignment and might be referred to the Office of Student Integrity.** If you are concerned about your submission time, feel free to ask on Piazza.
2. You can ask TAs clarification questions on Piazza, but you are ultimately responsible for what you submit. **The information provided in this Timed Lab document takes precedence.**
3. Resources you are allowed to use during the timed lab:
   - Assignment files
   - Previous homework and lab submissions (this includes homework PDFs)
   - Class Notes (Open Net, Open Book)
   - Your mind!
4. Resources you are **NOT** allowed to use:
   - Email/messaging
   - Contact in any form with any other person besides TAs

1.2 Submission Rules

1. Follow the guidelines under the Deliverables section.
2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Gradescope. Under no circumstances whatsoever will we accept any email submission of an assignment. Note: if you were granted an extension, you will still turn in your submission via Gradescope.
3. Do not submit links to files. We will not grade assignments submitted this way as it is easy to change the files after the submission period ends.

1.3 Is collaboration allowed?

Absolutely NOT. No collaboration is allowed for timed labs.

2 Overview

2.1 Purpose

The purpose of this timed lab is to test your understanding of the LC-3 calling convention by testing your ability to call and implement recursive subroutines using the stack.

2.2 Task

You will implement the subroutines listed below in LC-3 assembly language. Please see the detailed instructions for the subroutines on the following pages. We have provided pseudocode for the subroutines, and suggest that you follow these algorithms when writing your assembly code. Your subroutines must adhere to the LC-3 calling conventions.

2.3 Criteria

Your assignment will be graded based on your ability to correctly translate the given pseudocode for a subroutine (function) into LC-3 assembly code, following the LC-3 calling convention. Please use the LC-3 instruction set when writing these programs. Check the Deliverables section for what you must submit to Gradescope and the deadlines.

You must produce the correct return values for each function. In addition, registers `R0-R5` and `R7` must be restored from the perspective of the caller, so they contain the same values after the caller's `JSR` subroutine call. Your subroutine must return to the correct point in the caller's code, and the caller must find the return value on the stack where it is expected to be. If you follow the LC-3 calling conventions correctly, all of these things will happen automatically.

While we will give partial credit where we can, 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 Timed Lab, you will be implementing two subroutines named `MOD` and `SUM`. `MOD` is a non-recursive subroutine that will calculate the remainder of **a** divided by **b**. `SUM` will **recursively** iterate through a linked list and sum the mod of the data values at each node.

3.1 MOD subroutine

`MOD` will take two arguments you will need to load from the stack:

- `a`: A positive integer
- `b`: A positive integer less than `A`

You will need to perform the calculation `a % b` using the pseudocode below:

```
MOD(int a, int b) {
    while (a >= b) {
        a = a - b;
    }
    return a;
}
```

Examples:

- `MOD(5, 2)` returns 1
- `MOD(10, 5)` returns 0
- `MOD(13, 5)` returns 3

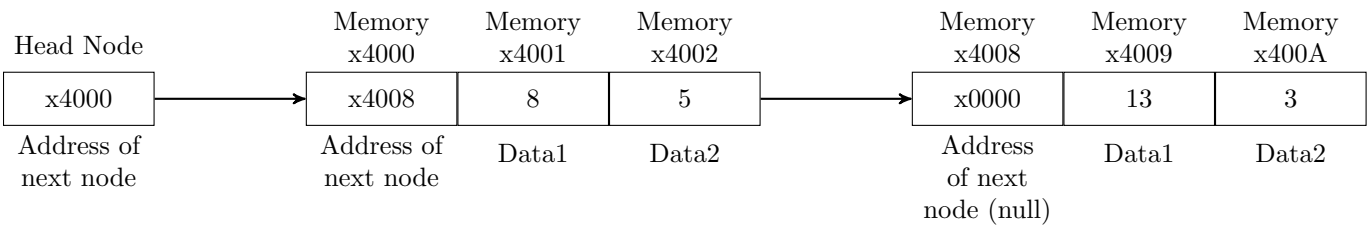
You can assume that `MOD` will only ever be called with positive, nonzero integer arguments.

The `MOD` subroutine will consist of two checkpoints. To test your understanding of the LC-3 calling convention, **you will need to place the arguments `a` and `b` into labels `A` and `B`**. You should not change these values before you place them at their respective labels. The second checkpoint will test to see if you are returning the correct value. Please refer to checkpoints 1 and 2 for details on how `MOD` will be graded.

3.2 SUM subroutine

`SUM` will take one argument you need to load onto the stack:

- `node`: A node of the linked list. Please refer to the diagram for more details on the structure of this node.



Each node has three values: the address of the next node, `data1` and `data2`. Please view the diagram above to understand how to index into these values. If `data1` is larger than `data2`, you will need to call `MOD(data1, data2)`, otherwise call `MOD(data2, data1)`. Once you get the return value of the `MOD` subroutine, you should call `SUM(node.next)` on the next node. The `SUM` subroutine will return the sum of the call to `MOD` and the return value of `SUM(node.next)`. Please refer to the pseudocode below.

```
SUM(Node node) {
    if (node == null) {
        return 0;
    }

    int data1 = node.data1;
    int data2 = node.data2;
    int result = 0;

    if (data1 > data2) {

        result = MOD(data1, data2);

    } else {

        result = MOD(data2, data1);

    }

    int sum = SUM(node.next);

    return result + sum;

}
```

The first checkpoint (third overall) for the `SUM` subroutine will check that you handled the base case, when the address of `node` is `x0000`. The second checkpoint will test if you return the correct sum of the MOD's of the data values at each node. Please refer to the checkpoint section of the PDF to see how these will be graded.

4 Checkpoints

4.1 Checkpoints (70 points)

In order to get all of the points for this timed lab, your code must meet these checkpoints:

- Checkpoint 1 (15 points): In `MOD`, load the parameters `a` and `b` off the stack and place their values at labels `A` and `B` respectively.
- Checkpoint 2 (20 points): Implement `MOD` to return `A % B`.
- Checkpoint 3 (10 points): In `SUM`, handle the base case when `node` is null.
- Checkpoint 4 (25 points): Implement `SUM`. If `data1` is greater than `data2`, calculate `MOD(data1, data2)`, otherwise calculate `MOD(data2, data1)`. Call `SUM(node.next)`. Return the sum of the return values of `MOD` and `SUM`.

4.2 Other Requirements (30 points)

Your subroutine must follow the LC-3 calling convention. Specifically, it must fulfill the following conditions:

- Your subroutine must be recursive and call itself according to the pseudocode's description.
- When your subroutine returns, every register must have its original value preserved (except R6).
- When your subroutine returns, the stack pointer (R6) must be decreased by 1 from its original value so that it now points to the return value.
- During the execution of your subroutine, you must make the correct number of calls to `MOD` and `SUM`, corresponding to the pseudocode.
  - If the autograder claims that you are making an unknown subroutine call to some label in your code, it may be that your code has two labels without an instruction between them. Removing one of the labels should appease the autograder.

5 Deliverables

Turn in the following files on Gradescope during your assigned lab section on Monday, October 26$^{\text{th}}$:

1. `tl03.asm`

6 LC-3 Assembly Programming Requirements

6.1 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. **Comment your code!** 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.

3. 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
   ```

4. **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.

5. Following from 4. You can randomize the memory and load your program by doing File - Randomize and Load.

6. Use the LC-3 calling convention. This means that all local variables, frame pointer, etc... must be pushed onto the stack. Our autograder will be checking for correct stack setup.

7. Start the stack at xF000. **The stack pointer always points to the last used stack location.** This means you will allocate space **first**, then store onto the stack pointer.

8. Do NOT execute any data as if it were an instruction (meaning you should put .fills after **HALT** or RET).

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

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

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

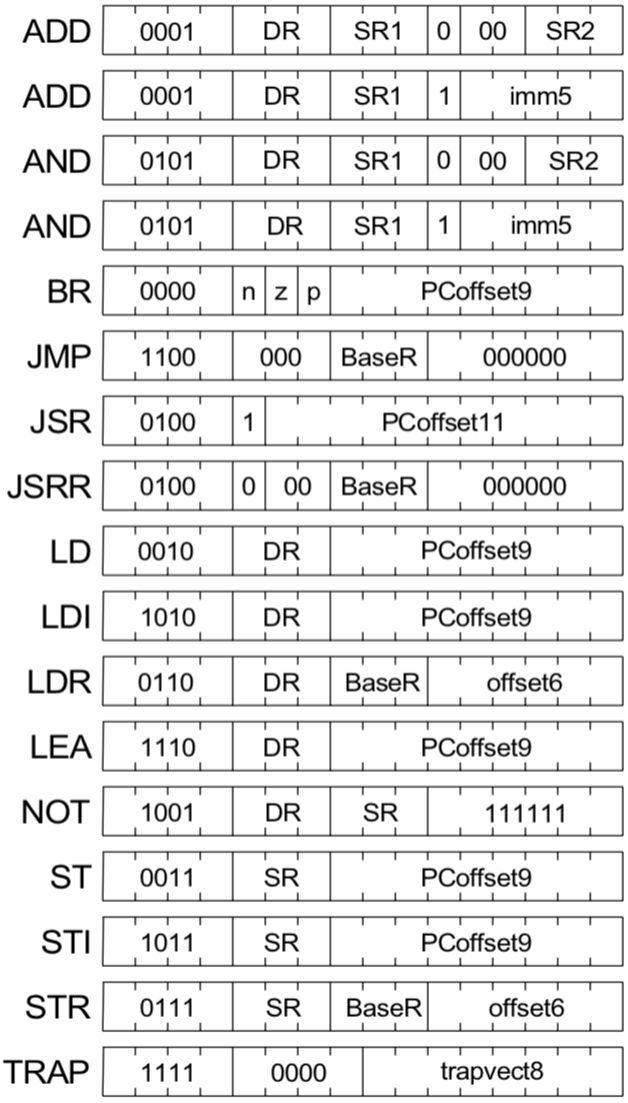
7 Running Local Autograder

When you turn in your files on gradescope for the first time, you might not receive a perfect score. Does this mean you change one line and spam gradescope until you get a 100? No! You can use a handy tool known as tester strings.

1. First off, we can get these tester strings in two places: the local grader or off of gradescope. To run the local grader:
   - Mac/Linux Users:
     (a) Navigate to the directory your timed-lab is in. **In your terminal, not in your browser**
     (b) Run the command `sudo chmod +x grade.sh`
     (c) Now run `./grade.sh`
   - Windows Users:
     (a) On **docker quickstart**, navigate to the directory your timed-lab is in
     (b) Run `./grade.sh`

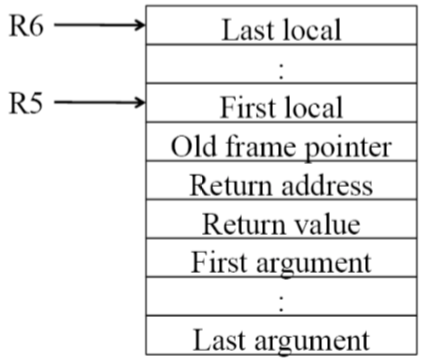
8 Appendix

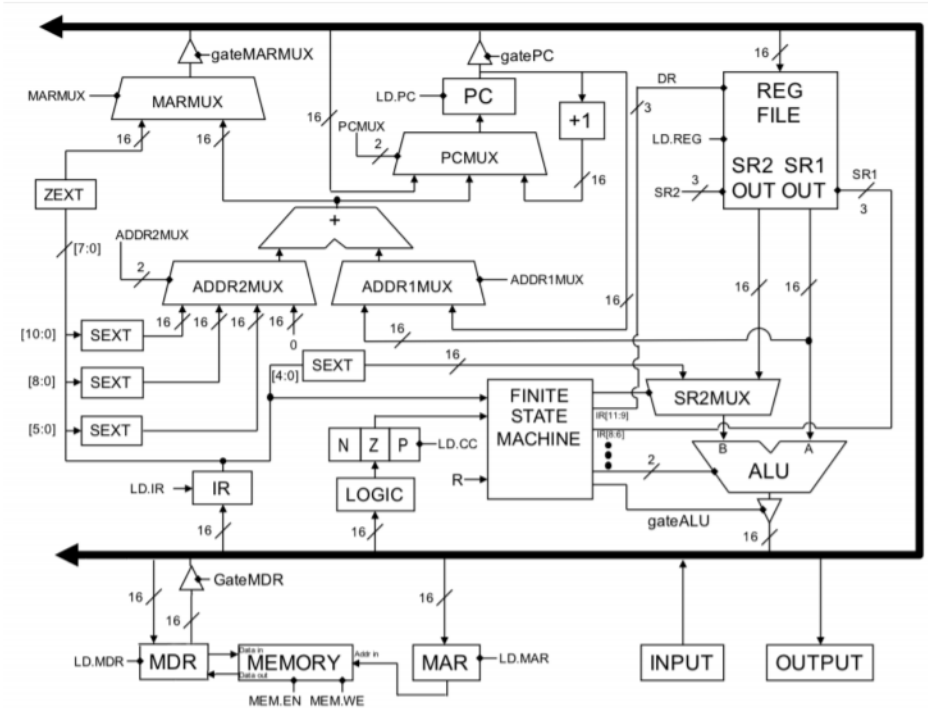
8.1 Appendix A: LC-3 Instruction Set Architecture



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, BUS, ADDER
MARMUX	ZEXT, ADDER
SR2MUX	SR2, SEXT