

CS 2110 Timed Lab 3: Subroutines and Calling Conventions

Your TAs

Spring 2021

Contents

1	Timed Lab Rules - Please Read	2
2	Overview	2
2.1	Purpose	2
2.2	Task	2
2.3	Criteria	2
3	Detailed Instructions	3
3.1	ABS subroutines	3
3.2	POW3 subroutine	3
3.3	MAP subroutine	4
4	Checkpoints	5
4.1	Checkpoints (70 points)	5
4.2	Other Requirements (30 points)	5
5	Deliverables	5
6	Local Autograder	6
7	LC-3 Assembly Programming Requirements	7
7.1	Overview	7
8	Appendix	8
8.1	Appendix A: LC-3 Instruction Set Architecture	8

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

You are allowed to submit this timed lab starting from the moment your assignment is released until your individual period is over. You have 75 minutes to complete the lab, unless you have accommodations that have already been discussed with your professor. Gradescope submissions will remain open for several days, but you are not allowed to submit after the lab period is over. **You are responsible for watching your own time. Submitting or resubmitting after your due date may constitute an honor code violation.**

If you have questions during the timed lab, you may ask the TAs for clarification in a private Piazza post, though you are ultimately responsible for what you submit. The information provided in this Timed Lab document takes precedence. If you notice any conflicting information, please indicate it to your TAs.

The timed lab 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 for timed labs.**

2 Overview

2.1 Purpose

The purpose of this timed lab is to test your understanding of implementing subroutines in the LC-3 assembly language using the calling convention, from both the callee and caller side.

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—you should 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.

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. Additionally, we will check that you made the correct subroutine calls, so you should not try to implement a recursively subroutine iteratively.

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 three subroutines: ABS, POW3 and MAP. ABS is a non-recursive subroutine that will calculate the absolute value of its argument. POW3 is a recursive subroutine that will compute 3 to the power of its argument. MAP will iterate through an array and apply one of the two subroutines above to each of the elements.

3.1 ABS subroutines

ABS will take one argument that should be loaded from the stack. You should implement the subroutine as shown in the pseudocode below:

```
// checkpoint 1
int ABS(int x) {
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
}
```

Examples:

- ABS(5) returns 5
- ABS(0) returns 0
- ABS(-3) returns 3

Implementing ABS will be the first checkpoint. Please refer to Checkpoints for details on how ABS will be graded.

3.2 POW3 subroutine

POW3 will take one **non-negative integer** argument n and compute 3^n ; that is, 3 raised to the power of the argument. You should implement the subroutine as shown in the pseudocode below:

```
int POW3(int n) {
    if (n == 0) {
        return 1; // (checkpoint 2) base case:  $3^0 = 1$ 
    } else {
        return 3 * POW3(n - 1); // (checkpoint 3) recursive case
    }
}
```

Examples:

- POW3(0) returns 1
- POW3(1) returns 3
- POW3(3) returns 27

POW3 contains two checkpoints. Checkpoint 2 is the base case: returning 1 when $n == 0$. Checkpoint 3 is the recursive case: POW3 should return the correct value for all non-negative integers. Please refer to Checkpoints for details on how POW3 will be graded.

3.3 MAP subroutine

MAP takes two arguments:

- `array`: The address of the first element in the array to be modified
- `length`: The length of the array to be modified

MAP should iterate through the array. For every element at an *even* index, it should apply `ABS` to it and replace the element with the result. For every element at an *odd* index, it should apply `POW3` to it and replace the element with the result.

Hint: remember Homework 1. What's a simple way using bitwise operations to check that a number is even or odd without doing a full mod operation?

You should implement the subroutine as shown in the pseudocode below:

```
// checkpoint 4
void MAP(array, length) {
    i = 0;
    while (i < length) {
        element = arr[i];
        if (i % 2 == 0) {
            result = ABS(element);
        } else {
            result = POW3(element);
        }
        arr[i] = result;
        i = i + 1;
    }
}
```

You do not need to return a value from MAP. While you will still leave a spot for a return value on the stack, it will be ignored.

Correctly implementing MAP is checkpoint 4. Please refer to Checkpoints for details on how MAP 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 ABS, load the parameters `x` and compute the absolute value to return `|x|`.
- Checkpoint 2 (15 points): Implement the base case for POW3 to return 1 when `n == 0`.
- Checkpoint 3 (20 points): Implement the recursive case of POW3 to successfully compute and return 3^n for any non-negative integer `n`.
- Checkpoint 4 (20 points): Implement MAP. You should iterate through the array, and apply the ABS subroutine to elements at all even indices and the POW3 subroutine to elements at all odd indices. The results of each subroutine call must be properly stored back into the array at every index.

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 ABS and POW3, 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 timed lab slot:

1. `t103.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 homework is 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 homework is 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. **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 going to File ↪ Advanced Load and selecting RANDOMIZE for registers and memory.
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. The stack will start 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 HALT or RET instructions before any .fills).
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.

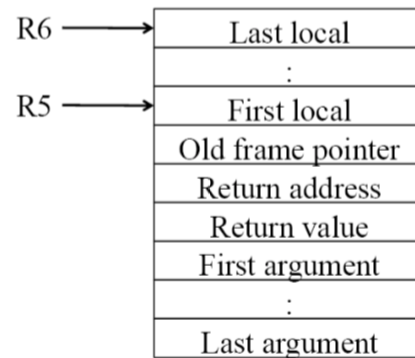
8 Appendix

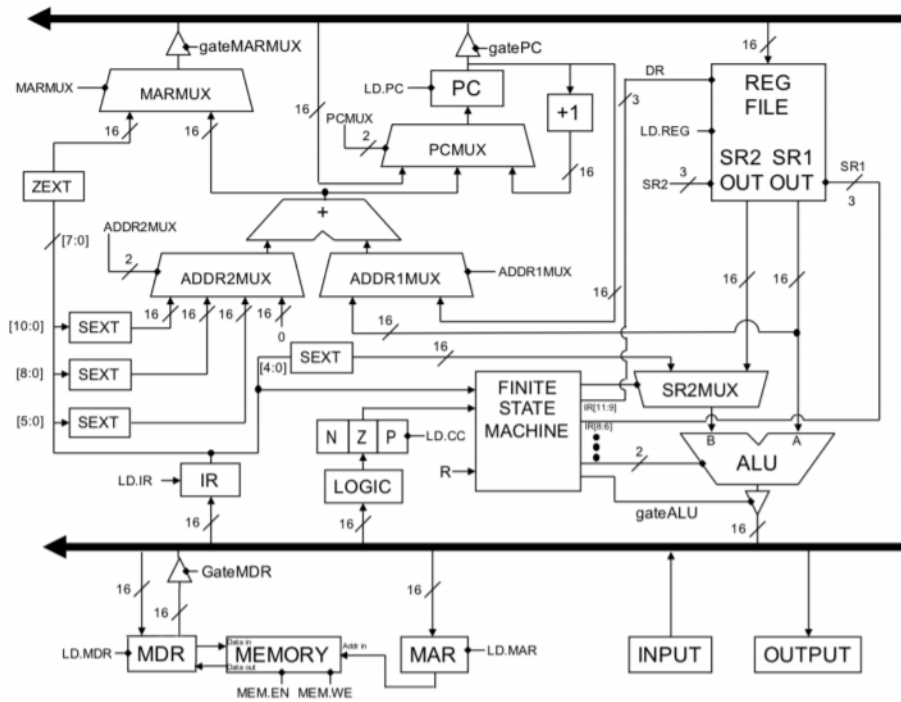
8.1 Appendix A: LC-3 Instruction Set Architecture

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