CS 2110 Timed Lab 3: Subroutines and Calling Conventions

Nandha Sundaravadivel, Rahul Narayanan, Rishi Nopany

Spring 2023

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

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 lab, 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 given a number represented as a string. Your task will be to compute and the return the digital root of that number.

The digital root of a number is found by repeatedly summing the digits of the original number until it converges onto a single digit.

For example: Input: `12345` Output: `6`

Explanation:

Step 1: `1 + 2 + 3 + 4 + 5 = 15`

Step 2: `1 + 5 = 6`

Since we find a single digit number in step 2, the output is 6

We will implement this function mainly in the function `digital_root` with the assistance of two helper functions

The high level flow of this timed lab is as follows:

1. If our number is less than 10, we have found the digital root
2. If it is not, we will write out our number as a string into memory

   NOTE: This function is given to you, you will only need to call it
3. Iterate through the individual characters in the string created in the previous function to find the sum of the individual numbers
4. Repeat steps 1-3 until the digital root is found

You will implement steps 1 and 4 in the function `digital_root` and step 3 in the function `sum_digits`

3.1 `to_decimal_string` (THIS FUNCTION IS GIVEN TO YOU)

Stores a given positive integer as a string starting at the given memory address including the null terminator

NOTE (PAY CAREFUL ATTENTION HERE): This function technically returns nothing, but, due to the standard calling convention, the stack pointer will be pointing to a space in memory that was allocated for the return value when this function returns.

NOTE 2: Pay extra attention to the result of this function, it will write characters into memory, not numbers. This is because we can use the null terminator character to denote the end of the string

Parameter `address`: Memory address where decimal string should be written to

Parameter `num`: Number that should be returned as a string

Returns: None

Example: `to_decimal_string(x6000, 123)` writes

x6000	'1'
x6001	'2'
x6002	'3'
x6003	\0

3.2 sum_digits

This function takes in an address and returns the sum of the values of each character of the numeric string contained at that address.

Parameter address: The address of the first character of the numeric string

Returns: Sum of each digit found in the numeric string.

Note: The numeric string contains character representations of the digits, not the literal digits themselves. You must return the sum of the numeric value of the digits.

Examples:

x6000	'1'
x6001	'2'
x6002	'3'
x6003	\0

`sum_digits` on `x6000` returns `6`

Pseudocode:

```
sum_digits(int address) {
    int sum = 0;
    char curr = mem[address];
    while (curr != 0){
        sum += int(curr);
        address += 1;
        curr =  mem[address];
    }
    return sum;
}
```

3.3 digital_root

Takes a number and returns the digital root of that number. You are given an address stored in label WRITE_SPACE. This address contains space allocated for up to 10 characters to store the string representations of your number.

Parameter num: The number we would like to take the digital root of. You are given an address stored in label `WRITE_SPACE`. This address contains space allocated for up to 10 characters to store the string representations of your number.

Returns: The digital root of the num

Examples:

- `digital_root` on `15` returns `6`
- `digital_root` on `12` returns `3`
- `digital_root` on `57` returns `3`

Pseudocode:

```
digital_root(int num) {
    int addr = mem[WRITE_SPACE]
    to_decimal_string(addr, num);
    int sum = sum_digits(addr);
    if (sum >= 10){
        return digital_root(sum);
    }
    return sum;
}
```

4 Checkpoints

4.1 Checkpoints (72 points)

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

- Checkpoint 1 (30 points): Implement subroutine `sum_digits` to return the sum of the numbers in a numeric string.
- Checkpoint 2 (22 points): Implement the base case of `digital_root` to successfully compute the digital root if the sum has only one digit.
- Checkpoint 3 (20 points): Implement the recursive case of `digital_root` to successfully compute and return the value of the recursive call.

4.2 Other Requirements (28 points)

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

- Your `digital_root` subroutine must be recursive and call itself, `sum_digits` and `to_decimal_string` 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.
  - 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. `tl03.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`).

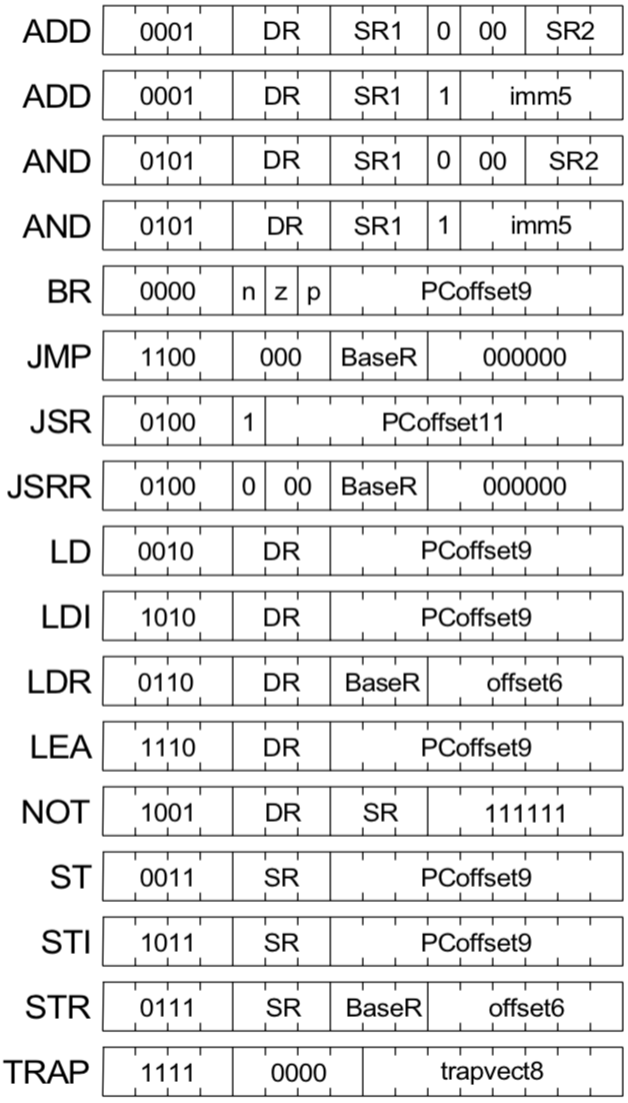
9. Do not add any comments beginning with @plugin or change any comments of this kind.

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.

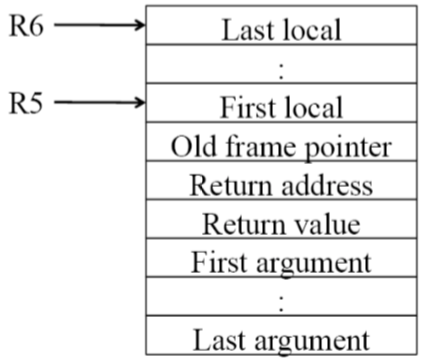
8 Appendix

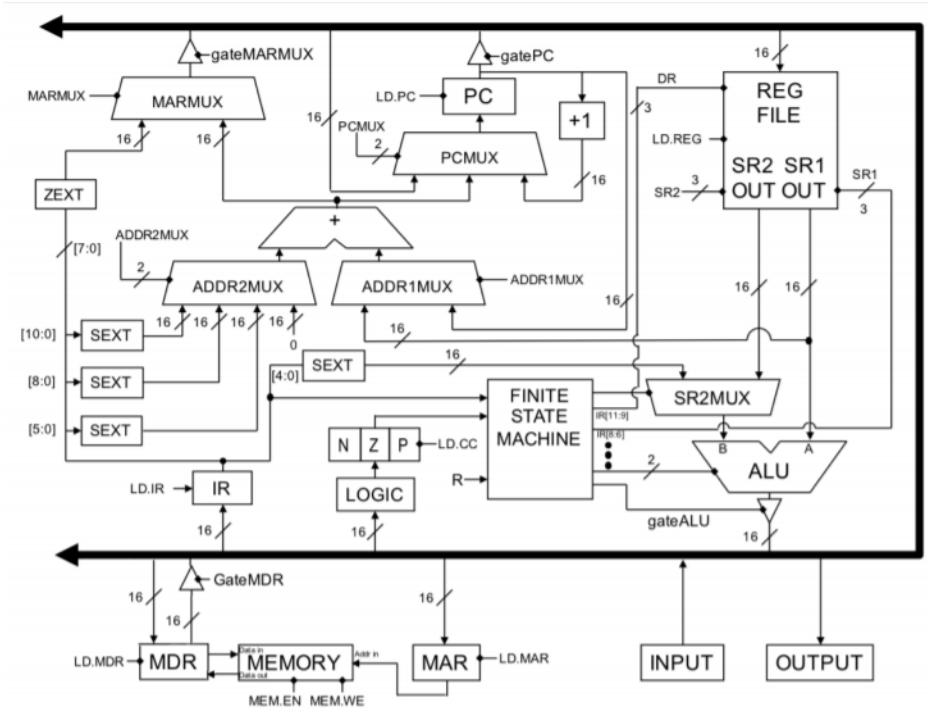
8.1 Appendix A: LC-3 Instruction Set Architecture



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