CS 2110 Final Exam: LC-3 Assembly

Your Friendly TAs :)

Summer 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 section of the final exam is to test your understanding of coding in LC-3 assembly, including the use of the LC-3 calling convention to write and call subroutines.

2.2 Task

This section of the final exam is divided into two subsections. In the first, you will be implementing a short assembly program that is not a subroutine; in the second, you will be implementing an assembly subroutine. Please see the detailed instructions for each part on the following pages. We have provided pseudocode for both parts—you should follow these algorithms when writing your assembly code. Your subroutine must adhere to the LC-3 calling conventions.

2.3 Criteria

This section will be graded based on your ability to correctly translate the given pseudocode into LC-3 assembly code and follow the LC-3 calling convention for subroutine implementations. 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. Additionally, for your subroutine, 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. The two parts are separated into their own files, so you can still receive credit for one part if the other does not assemble.

3 Detailed Instructions

For this section of the final exam, you will first be implementing an equivalent to the familiar GBA function `setPixel` in assembly. This program will take in as inputs a row, column, height, width, and color value via labels, and you will write this color value to the appropriate index in a `VIDEOBUFFER` array.

In the second part, you will be implementing a recursive subroutine, in accordance with the LC-3 calling convention, to calculate the modulo of two numbers `a` and `b`.

3.1 `setPixel`

`setPixel` will write a given color value to the appropriate index in a `VIDEOBUFFER` array, representing a 2D image with given height and width, as specified by a row and column number. `setPixel` will take 5 arguments via labels. This is **NOT** a subroutine program.

- `COLOR` is the color value to write to `VIDEOBUFFER`
- `HEIGHT` is the height of the `VIDEOBUFFER`
- `WIDTH` is the width of the `VIDEOBUFFER`
- `ROW` is the row number of the pixel in the `VIDEOBUFFER` of which you will write the color value
- `COL` is the column number of the pixel in the `VIDEOBUFFER` of which you will write the color value

The address of the `VIDEOBUFFER` itself is given at the memory location labelled `VIDEOBUFFER`.

Your program should set the value at index `ROW * WIDTH + COL` of the `VIDEOBUFFER` equal to `COLOR`. You may assume that `ROW` and `COL` will be in-bounds. The following pseudocode is provided for reference:

```
offset = 0;
for (i = 0; i < ROW; i++) {
    offset += WIDTH;
}
offset += COL;
VIDEOBUFFER[offset] = COLOR
```

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

3.2 `mod` subroutine

The `mod` subroutine will take one integer argument `a` and one **positive** integer argument `b` and compute `a % b`: that is, the remainder of performing integer division between `a` and `b`. This is a subroutine program. The stack buildup and tear-down portions of the code are included in the file for you. Feel free to replace them with your own stack buildup and tear-down code if you choose. We've only included the stack buildup/tear-down code as the callee, you must still write your own function calls using the appropriate convention as the caller.

You should follow the given pseudocode to implement this subroutine, which accounts for all cases that you are required to handle:

```
mod(a, b) {
    if (a < 0) {
        return mod(a + b, b);
    }
    if (a < b) {
        return a;
    }
    return mod(a - b, b);
}
```

Examples:

- `mod(5, 2)` returns `1`
- `mod(3, 4)` returns `3`
- `mod(-7, 6)` returns `5`

Note: The precise mathematical definition of modulo here is as follows: `mod(a,b)` $= a - (\lfloor a \div b \rfloor \times b)$. As such, this version of modulo will always return a remainder with the same sign as the divisor `b` (always positive in this case), regardless of the sign of the dividend `a`. Additionally, note that while `b` is guaranteed to be positive, `a` may be negative and your subroutine must handle this case (as a reminder, the provided pseudocode does this already).

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

4 Grading

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

- `setPixel` (10 points): Writing the correct color value to the correct index of `VIDEOBUFFER`.
- `mod` return value (15 points): Returning the correct value of `a % b` from the `mod` subroutine.
- Other requirements (5 points): The `mod` subroutine must follow the LC-3 calling convention. Specifically, it must fulfill the following conditions:
  - Your `mod` subroutine must be recursive and call itself according to the pseudocode's description. 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.
  - 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.

5 Deliverables

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

1. `setPixel.asm`
2. `mod.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. 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.

4. 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.

5. 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.

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

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

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

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

10. **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
    ```