CS 2110 Project 3: Assembly Programming

Gal Ovadia, Archita Hothur, Annelise Lloyd, Richard So, Rohan Bafna

Fall 2023

Contents

1 General

1.1 Introduction

Hello and welcome to Project 3. In this project, you'll be implementing a word processor. The project has been divided into multiple subroutines that you need to implement.

The project is due October 31st, 11:59 PM EST

Please read through the entire document before starting. Often times we elaborate on instructions later on in the document. **Start early** and if you get stuck, there's always Piazza or office hours.

1.2 General Instructions

- You can create your own labels and fill them with data as you see fit. However, if you have two labels corresponding to the same memory location (e.g., two labels without an instruction between them) this may cause autograder errors.
- You can modify the values stored in some of the helper labels we provide you if you think it'll make it easier for you to program with.
- Take a look at Section 6 for details on the LC-3 Assembly Programming requirements that you must adhere to.

1.3 Running the autograder

Take a look at section 5 for information on how to run the autograder, and how to debug your code. For this project, we will be using LC3Tools, which can be downloaded from Canvas files.

2 Absolute Sum of Array

We'll start with a warm up program before we get to the main chunk of the project. Here you will implement a program that calculates the sum of the absolute values of each element of the array. Memory at the label ARR contains the address of the first element of the array and LEN contains the number of elements present in that array. You should store the resulting sum in the address that the label ANSWER specifies.

Example: array = { 1, 3, -4, -2} sum = 1 + 3 + 4 + 2 = 10

Please write your code is asoluteSum.asm. **Please do not modify the .orig address as it will cause issues in the autograder.**

Suggested Pseudocode:

```
answer = 0;
currNum = 0;
i = 0;
arrLength = arr.length();
while (arrLength > 0) {
    currNum = arr[i];
    if (currNum < 0) {
        currNum = -(currNum);
    }
    answer += currNum;
    i++;
    arrLength--;
}
return
```

3 Word Processor

We're getting into the real meat of the project now. You will be implementing a word processor that takes in an input paragraph, and your job is to divide into lines of 8 characters and perform some modifying to each line. Detailed explanations on what you have to do are explained later. Also, we have divided this segment into smaller subroutines to make it easier. We strongly recommend that you do them in order.

3.1 wordLength

To start off, you will implement a function that calculates the length of a word until space, newline, or null char. The starting address of the word will be present in register R0 when this subroutine is called. Special characters and numbers should be treated as regular letters and should be added to the length. Store the resulting word length back into R0.

Example: "Hello" : wordLength = 5 "this," : wordLength = 5

Please do not modify the .orig address as it will cause issues in the autograder.

Suggested Pseudocode:

```
def wordLength(R0):
    addr = R0
    length = 0
    while (true):
        if (mem[addr] == '\0'):
            break
        if (mem[addr] == '\n'):
            break
        if (mem[addr] == ' '):
            break
        addr += 1
        length += 1
    R0 = length
    return
```

3.2 memcpy

This subroutine uses 3 parameters: starting address, destination address, and length (n). This function should copy a total of n characters starting from the starting address into the address specified by destination address.

Starting address will be present in register R0 Destination address will be present in register R1 Length (n) will be present in register R2

Please do not modify the .orig address as it will cause issues in the autograder.

Suggested Pseudocode:

```
def memcpy(R0, R1, R2):
    sourcePtr = R0
    destPtr = R1
    length = R2
    while (length > 0):
        mem[destPtr] = mem[sourcePtr]
        sourcePtr += 1
        destPtr += 1
        length -= 1
    return
```

3.3 capitalizeLine

This subroutine takes in a starting address of a line in memory and should capitalize all characters in-place until a newline (\n) or null character is encountered (\0). The starting address will be present in register R0.

Example: If the starting address is x4000, the before and after this subroutine is called is shown below:

x4000	's'
x4001	'T'
x4002	'r'
x4003	'!'
x4004	'n'
x4005	'g'
x4006	' '
x4007	'F'
x4008	'u'
x4009	'n'
x400A	'\n'

Before

x4000	'S'
x4001	'T'
x4002	'R'
x4003	'!'
x4004	'N'
x4005	'G'
x4006	' '
x4007	'F'
x4008	'U'
x4009	'N'
x400A	'\n'

After

Note: Capital letters and special characters should NOT be modified.

Please do not modify the .orig address as it will cause issues in the autograder.

Suggested Pseudocode:

```
def capitalizeLine(R0):
    addr = R0
    while (mem[addr] != '\0' and mem[addr] != '\n'):
        if (mem[addr] >= 'a' and mem[addr] <= 'z'):
            mem[addr] = mem[addr] - 32
        addr += 1
    return
```

3.4 reverseWords

This subroutine will take in a starting address for a line. This subroutine's job is to reverse each word in the line in-place until a newline or null character with the help of a stack. The starting address of the line will be present in R0. The address of the top of the stack will be in R6.

Example: "2110 is a pretty cool class\n" becomes "0112 si a ytterp looc ssalc\n"

Please do not modify the .orig address as it will cause issues in the autograder.

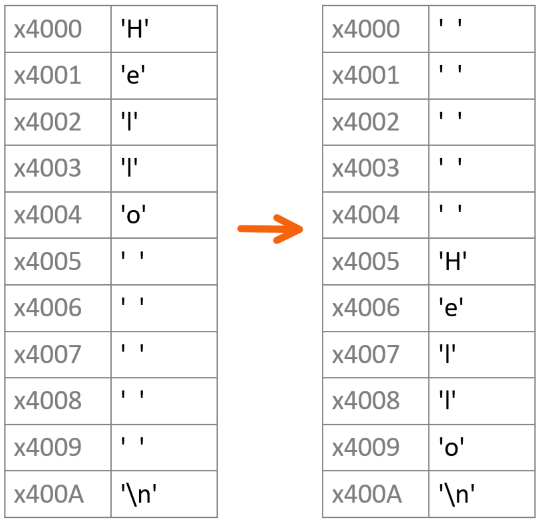
Suggested Pseudocode:

```
def reverseWords(R0):
    i = R0
    while (true):
         if (mem[i] == '\0' or mem[i] == '\n'):
             break
         if (mem[i] == ' '):
             i++
             continue
         start = i
         count = 0
         while (mem[i] != ' ' and mem[i] != '\0' and mem[i] != '\n'):
             stack.push(mem[i])
             i++
             count++
         i = start
         while (count > 0):
             mem[i] = stack.pop()
             i++
             count--
    return
```

3.5 rightJustify

Just like the other subroutines, this one also takes in the starting address of a line and modifies the line in-place. For each space on the right-hand side of the line (before newline or null character), shift all characters in the line to the right by 1, and add a space on the left. Do this until there are no more spaces on the RHS. The starting address of the line will be present in R0.

Example: "Low-level coding, high-level fun!   \n" Should become "   Low-level coding, high-level fun!\n"



x4000	'H'
x4001	'e'
x4002	'l'
x4003	'l'
x4004	'o'
x4005	' '
x4006	' '
x4007	' '
x4008	' '
x4009	' '
x400A	'\n'

Before

x4000	' '
x4001	' '
x4002	' '
x4003	' '
x4004	' '
x4005	'H'
x4006	'e'
x4007	'l'
x4008	'l'
x4009	'o'
x400A	'\n'

After

Please do not modify the .orig address as it will cause issues in the autograder.

Suggested Pseudocode:

```
def rightJustify(R0):
   start = R0
   curr = start
   while (mem[curr] != ’\n’ and mem[curr] != ’\0’):
       curr++
   curr--
   end = curr
   // This loop shifts over the entire string one spacebar at a time,
   // until it is no longer terminated by a spacebar!
   while (mem[end] == ’ ’):
       while (curr != start):
           mem[curr] = mem[curr - 1]
           curr--
       mem[curr] = ’ ’
       curr = end
   return
```

3.6 getInput

This function should read a string of characters from the keyboard and place them somewhere in memory. The string should be terminated by two consecutive ’$’ characters. The ’$’ characters should not be stored in memory. Remember to properly null-terminate your string, and to print out each character as it is typed! **You may assume that the user will always enter a valid input.** The address of the memory location at which the input needs to be stored should be passed in via register R0.

Please do not modify the .orig address as it will cause issues in the autograder.

Suggested Pseudocode:

```
def getInput(R0):
```

```
bufferPointer = R0
while (true):
    input = GETC()
    OUT(input)
    mem[bufferPointer] = input
    if input == '$':
        if mem[bufferPointer - 1] == '$':
            mem[bufferPointer - 1] = '\0'
            break
    bufferPointer += 1
```

3.7 parseLines

This function should parse a string of characters from an initial buffer and place the parsed string in a new buffer. This method should divide the string into lines of 8 characters or less, not including the NewLine character! If a word cannot fit within the current line, add space bars to pad the current line so it is 8 characters long, and place the word on the next line. The address of the buffer containing the unparsed string will be passed in R0, and the address of the destination buffer will be passed through R1.

Please do not modify the .orig address as it will cause issues in the autograder.

For the purpose of this this project, this subroutine assumes that ALL words inputted will have a maximum of 8 characters

This subroutine has already been implemented for you, so you do not need to write it yourself. But please understand what this subroutine does because you will have to call it later in your code.

3.8 wordProcess

You're almost done with the project (yay!!!), but there's one more subroutine you need to write which uses the subroutines you've written so far. We will also use IO to take in user input and print the output for this subroutine. This subroutine will be called by the main subroutine.

This is how this subroutine should work:

- getInput: Reads all the characters input by the user until '$$' and stores it in a buffer in memory, labeled as BUFFER1. In memory, you should not store the '$$', but you should terminate the input provided by the user with the null character \0.
- parseLines: For each word in BUFFER1, calculate the length of the word and add to BUFFER2 according to this:
  - If the word length + the current length of the line is less than 8, then fill in spaces in the line until line length = 8, then add newline char (all into BUFFER2). Note that spaces in between words count towards the line length.
  - To copy the words in BUFFER1 into BUFFER2 use the memcpy subroutine.
  - **For the purpose of this this project, this subroutine assumes that ALL words inputted will have a maximum of 8 characters.**
- For each line resulting from the above operation, read an option number from the console:
  - If 1: Capitalize the line
  - If 2: Reverse each word in the line
  - If 3: Right justify the line
  - Else: Leave the line as it is
- Print the result on the console
- Exit

As always, please do not modify the .orig address as it will cause issues in the autograder.

Suggested Pseudocode:

```
def wordProcess():
    GetInput(x8000)
    OUT(\n)
    ParseLines(x8000, x8500)
    startOfCurrLine = x8500
    PUTS("Enter modifier options.\n")
    while (true):
        option = GETC()
        if (option == '0'):
            pass
        elif (option == '1'):
            CapitalizeLine(startOfCurrLine)
        elif (option == '2'):
             ReverseWords(startOfCurrLine)
        elif (option == '3'):
             RightJustify(startOfCurrLine)
        else:
            continue
        i = 0
        while (i < 9):
            OUT(mem[startOfCurrLine])
            startOfCurrLine++
            i++
        if (mem[startOfCurrLine - 1] == '\0'):
            break
    return
```

3.9 Putting it all together: main

This subroutine is our main subroutine, which is going to call wordProcess(). Since one of your subroutines uses the stack, it has been initialized at the beginning of this subroutine. The stack address has been loaded into R6 for you.

You can also test your various subroutines by changing the address of the subroutine for debugging.

Please follow the comments in the assembly file for this subroutine.

4 Deliverables

Turn in the file `wordProcessor.asm` and `absoluteSum.asm` to Gradescope by the due date.

Note: Please do not wait until the last minute to run/test your project, history has proved that last minute turn-ins will result in long queue times for grading on Gradescope. You have been warned.

5 Debugging

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 LC3 Tools to step through each line in your assembly program to see if the program is behaving the way it's supposed to.

If one of your subroutines is not working, you should try to debug it in isolation to other subroutines. Here's how to do so:

1. In your main function, the following lines jump/call a subroutine

   ```
   LD R5, SUBROUTINE_ADDR
   JSRR R5
   ```

   To jump the subroutine you want to test, you need to change the value at the label SUBROUTINE_ADDR to the address of the subroutine you want to test.

2. Also, remember that we are passing in arguments to each function through registers. So before you jump to the subroutine, make sure you put the correct values into the registers. The description for each subroutine contains which register to use as arguments.

3. Finally, you can load your file into LC3 Tools, assemble it and step through line-by-line to see why your subroutine is not behaving the way it should.

Some other points to keep in mind:

The Halt instruction resets your register values. To prevent this, you can do one of two things:

1. You can set a break point before returning from the subroutine.

2. Click on setting and turn on "Stop Execution on reaching HALT".

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 LC3Tools. 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. Do NOT execute any data as if it were an instruction (meaning you should put .fills after **HALT** or RET).
6. Do not add any comments beginning with @plugin or change any comments of this kind.
7. **Test your assembly.** Don't just assume it works and turn it in.

7 Rules and Regulations

7.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit.
2. Please read the assignment in its entirety before asking questions.
3. Please start assignments early, and ask for help early. Do not email us a few hours before the assignment is due with questions.
4. If you find any problems with the assignment, it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

7.2 Submission Conventions

1. In order to submit your assignment, submit the files individually to the Gradescope assignment.
2. Do not submit links to files. The autograder does not understand it, and we will not manually grade assignments submitted this way as it is easy to change the files after the submission period ends.

7.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.
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 we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Gradescope.
3. Projects turned in late receive partial credit within the first 48 hours, as defined in the syllabus. Between 0 and 24 hours late, you can receive a maximum score of 70%. Between 24 and 48 hours late, you can receive a maximum score of 50%. We will not accept projects turned in over 48 hours late.
4. You alone are responsible for submitting your project before the assignment is due; neither Canvas/Gradescope, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until the deadline.

7.4 Is collaboration allowed?

From the syllabus:

- You must submit an assignment or project as your own work. **No collaboration on answers is permitted. Absolutely no code or answers may be copied from others. Such copying is Academic Misconduct.**
- Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "submission of material that is wholly or substantially identical to that created or published by another person or persons").
- Publishing your assignments on public repositories, github, etc, that is accessible to other students is unauthorized collaboration and thus Academic Misconduct.

7.5 Syllabus Excerpt on Academic Misconduct

The goal of all assignments in CS 2110 is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. More importantly, it gives an unfair advantage over students who do the work and follow the rules.

1. **As a Georgia Tech student, you have read and agreed to the Georgia Tech Academic Honor Code.** The Academic Honor Code defines Academic Misconduct as "***any act that does or could improperly distort Student grades or other Student academic records.***"
2. You must submit an assignment or project as your own work. **No collaboration on answers is permitted. Absolutely no code or answers may be copied from others. Such copying is Academic Misconduct**.
3. Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "*submission of material that is wholly or substantially identical to that created or published by another person or persons*").
4. Publishing your assignments on public repositories, github, etc, that is accessible to other students is unauthorized collaboration and thus Academic Misconduct.
5. Suspected Academic Misconduct will be reported to the Division of Student Life Office of Student Integrity. It will be prosecuted to the full extent of Institute policies.
6. Students suspected of Academic Misconduct are informed **at the end of the semester**. Suspects receive an *Incomplete* final grade until the issue is resolved.
7. We use accepted forensic techniques to determine whether there is copying of a coding assignment.
8. Submitting an assignment with code or text from an AI assistant (e.g., ChatGPT) is academic misconduct.
9. **If you are not sure about any aspect of this policy, please ask your lecturer**.

Using AI Assistants

Anything you did not write in your assignment will be treated as an academic misconduct case. If you are unsure where the line is between collaborating with AI and copying AI, we recommend the following heuristics:

Heuristic 1: Never hit "Copy" within your conversation with an AI assistant. You can copy your own work into your own conversation, but do not copy anything from the conversation back into your assignment. Instead, use your interaction with the AI assistant as a learning experience, then let your assignment reflect your improved understanding.

Heuristic 2: Do not have your assignment and the AI agent open at the same time. Similar to the above, use your conversation with the AI as a learning experience, then close the interaction down, open your assignment, and let your assignment reflect your revised knowledge. This heuristic includes avoiding using AI directly integrated into your composition environment: just as you should not let a classmate write content or code directly into your submission, so also you should avoid using tools that directly add content to your submission.

Deviating from these heuristics does not automatically qualify as academic misconduct; however, following these heuristics essentially guarantees your collaboration will not cross the line into misconduct.