CS 2110 Project 3: Assembly

Tanya Qin, Sameer Suri, Varun Mehrotra, Calvin Khiddee-Wu, Justin Hinckley

Section A, Fall 2022

Contents

1 General

1.1 Introduction

Hello and welcome to Project 3. In this project, you'll be implementing the Fibonacci sequence, a lowercase converter for strings, and a greatest common divisor calculator in assembly!

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 Ed Discussion 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 7 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 Complx in the Docker Container to debug our code.

2 Fibonacci

To start off, you will implement a function that computes the first n numbers in the Fibonacci sequence. The Fibonacci sequence is important in the field of computer science. It is a special sequence in which the current element is the sum of the previous two elements. You should take in a single value, `N`, compute the first `N` Fibonacci numbers, and store them sequentially in memory starting at the label `RESULT`.

Suggested Pseudocode:

```
n = mem[N];
result = mem[RESULT];

if (n == 1) {
    mem[result] = 0;
} else if (n > 1) {
    mem[result] = 0;
    mem[result + 1] = 1;
    for (i = 2; i < n; i++) {
        x = mem[result + i - 1];
        y = mem[result + i - 2];
        mem[result + i] = x + y;
    }
}
```

Write your code in `fib.asm`.

3 String I/O

Next, you will write a program that takes in a string input from the console, changes any uppercase letters in the string to lowercase, and prints this new string to the console.

- Strings are essentially a contiguous array of ASCII values. KBDR only accepts a single character at a time, so you can use a loop to read all characters and store them before manipulating the string. Although strings are typically null-terminated, you should not expect a null terminator to be provided as part of the input.
- Except for `HALT`, **you may not** use `TRAP` instructions to implement this program. You must perform raw I/O by accessing memory-mapped I/O registers.
- Errors and warnings do not always correctly get cleared between runs of a program in Complx, which can make debugging console programs difficult. We recommend you close and reopen Complx after fixing errors, to make sure the message doesn't show up again for no reason. (You do not need to restart Docker or `cs2110docker.sh`).

Suggested Pseudocode:

```
length = mem[LENGTH];

i = 0;
while (i < length) {
    ready = mem[mem[KBSR]] & x8000;
    if (ready == 0) {
        continue;
    }
    currentChar = mem[mem[KBDR]];
    if (currentChar == '\0') {
        break;
    }
    currentChar = currentChar | 32; // Use DeMorgan's law!
    mem[TEMP + i] = currentChar;
    i++;
}

i = 0;
while (i < length) {
    ready = mem[mem[DSR]] & x8000;
    if (ready == 0) {
        continue;
    }
    currentChar = mem[TEMP + i];
    mem[mem[DDR]] = currentChar;
    i++;
}
```

The input strings will contain only letters, spaces, and numbers, and will be of length `LENGTH`. You should change all letters to lowercase, leaving non-letters as-is. Write your code in `lowercase.asm`.

4 Arithmetic and Bitwise Operations

For this section, you will be implementing a number of different operations that build off each other. Because of this, they are all in the same file, `gcd.asm`. Remember when implementing your program that labels must be unique per file, so **label names cannot be reused between subroutines** in this section. In other words, you cannot redeclare another label with the same name in the same file.

4.1 Bit Shifting

For this section, you will implement bit-shifting in LC-3 assembly as subroutines. As such, values for bit shifting will no longer be placed at label values, but rather, **passed in through registers. Results will also be stored in registers**. You will be given a value, and an amount of bits to shift by. Note that this will be a logical (unsigned) shift; in other words, you should fill in with zeroes for both a left and right shift. **We will not be sign extending**. You may assume that `R6` is correctly initialized with the location of the stack if you want to use the stack to save register values.

Suggested Pseudocode (Shift Left):

```
val = R0;
amt = R1;

while (amt > 0) {
    val = val + val;
    amt = amt - 1;
}

R0 = val;
```

Suggested Pseudocode (Shift Right):

```
val = R0;
amt = R1;

result = 0;

while (amt < 16) {
    result = result + result;
    if (val < 0) { // check if the MSB is set
        result = result + 1;
    }
    val = val + val;
    amt = amt + 1;
}

R0 = result;
```

The operands will be passed in through 2 registers, R0 for the value and R1 for the shift amount. The result should be stored at R0. **You will not have to handle overflow**. Write your code in `gcd.asm`, under the appropriate sections and `.orig` tags (`x3600` for `SHIFTLEFT`, `x3800` for `SHIFTRIGHT`).

4.2 Multiplication and Division

Here, you will implement integer multiplication and integer division operators, also as LC-3 assembly subroutines. Again, parameters will be passed in through registers, and results will be stored in registers. Given two values, you will either multiply the first with the second, or divide the first by the second. **You must use the bit-shifting subroutines to implement these subroutines. Additionally, the calls to the bit-shifting subroutines must match the calls made by the pseudocode below.** You may assume that `R6` is correctly initialized with the location of the stack if you want to use the stack to save register values.

Suggested Pseudocode (Multiply):

```
a = R0;
b = R1;

result = 0;
for (i = 0; i < 16; i++) {
    mask = 1 << i;
    if (b & mask != 0) {
        result = result + (a << i);
    }
}

R0 = result;
```

Suggested Pseudocode (Divide):

```
a = R0;
b = R1;

quotient = 0;
remainder = 0;

for (i = 15; i >= 0; i--) {
    quotient = quotient + quotient;
    remainder = remainder + remainder;
    remainder = remainder + ((a >> i) & 1);

    if (remainder >= b) {
        remainder = remainder - b;
        quotient = quotient + 1;
    }
}

R0 = quotient;

/* See more details about this algorithm here:
   https://en.wikipedia.org/wiki/Division_algorithm#Long_division */
```

The operands will be passed in through 2 registers, R0 and R1. The result should be stored at R0. **You will not have to handle overflow.** Write your code in `gcd.asm`, under the appropriate sections and .orig tags (x3200 for `MULTIPLY`, x3400 for `DIVIDE`).

4.3 Greatest Common Denominator

Finally, you will be implementing an iterative algorithm to find the greatest common denominator (GCD) of two values. The two values are stored in memory with the labels `A` and `B`. You will load them from memory, find their GCD using your algorithm, then store the GCD at the label `RESULT`.

You must use the multiply and divide subroutines to implement this program. Additionally, the calls to these subroutines must match the calls made by the pseudocode below.

You must initialize `R6` with the `STACK` label if you use the stack to save register values in this program or any subroutines (`LeftShift`, `RightShift`, `Multiply`, or `Divide`).

Suggested Pseudocode:

```
a = mem[A];
b = mem[B];

R6 = mem[STACK];

while (b > 0) {
    quotient = a / b;
    remainder = a - quotient * b;
    a = b;
    b = remainder;
}

mem[RESULT] = a;
```

You do not need to worry about overflow for negation or subtraction. Write your code in `gcd.asm`.

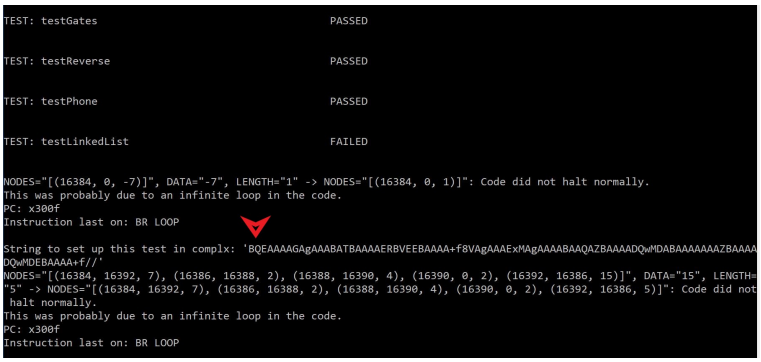
5 Debugging

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 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:

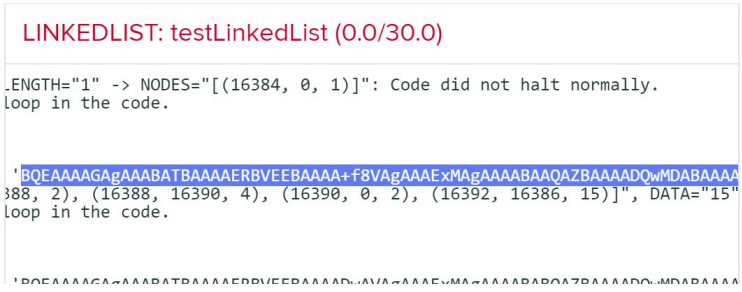
   - Mac/Linux Users:
     (a) **In your terminal, not in your browser**, navigate to the directory your project is in.
     (b) Run the command `sudo chmod +x grade.sh`
     (c) Now run `./grade.sh`
   - Windows Users:
     (a) On **Git Bash, not your browser**, navigate to the directory your project is in.
     (b) Run `./grade.sh`

   When you run the script, you should see an output like this:

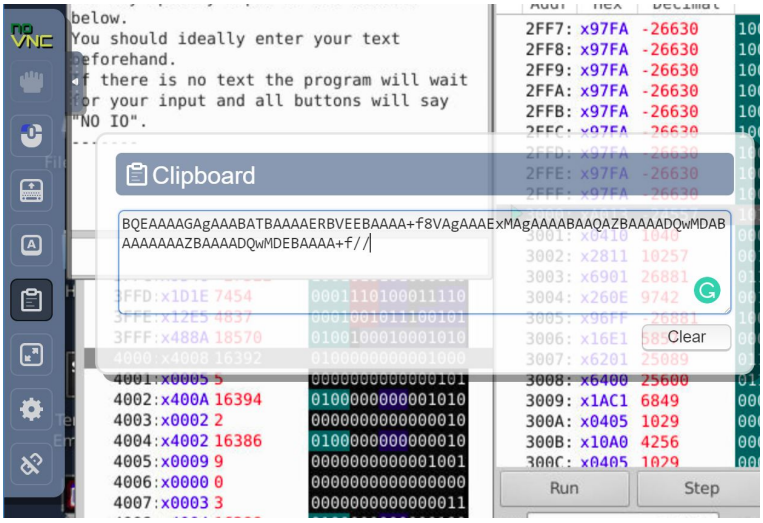


   Copy the tester string, which is everything between the single quotes (’). Do not include the quotations.

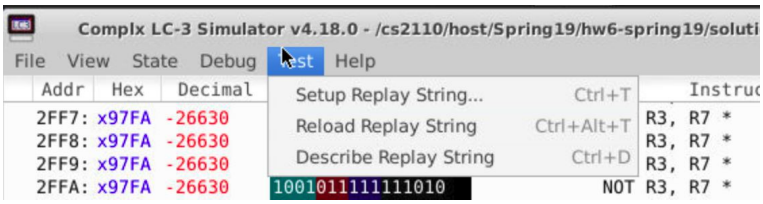
   Side Note: If you do not have Docker installed, you can still use the tester strings to debug your assembly code. In your Gradescope error output, you will see a tester string. When copying, make sure you copy from the first letter to the final backslash and again, don’t copy the quotations.



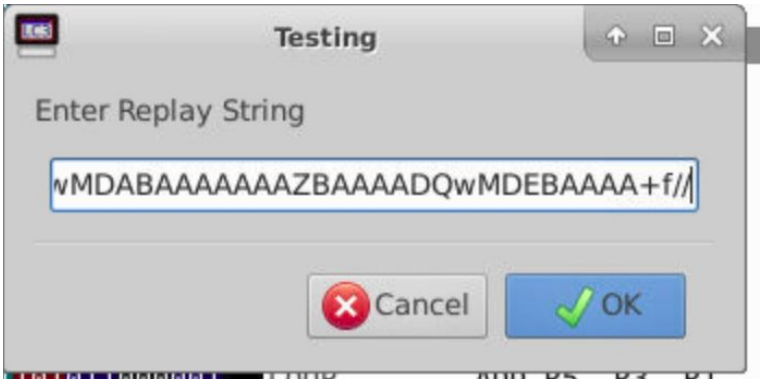
2. Secondly, navigate to the clipboard in your docker image and paste in the string.



3. Next, go to the Test Tab and click Setup Replay String

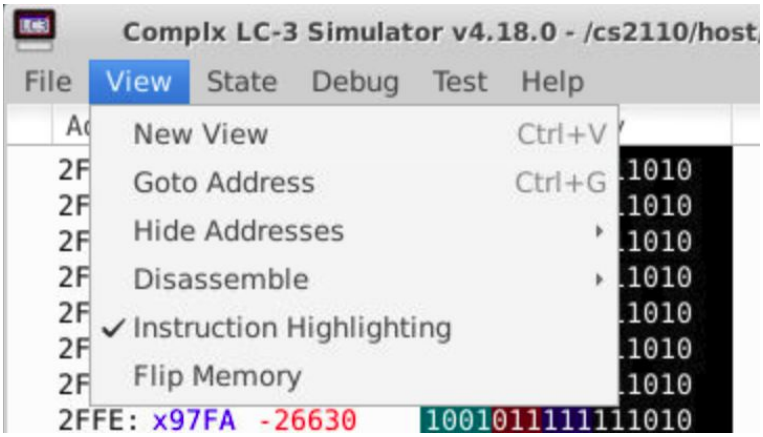


4. Now, paste your tester string in the box!

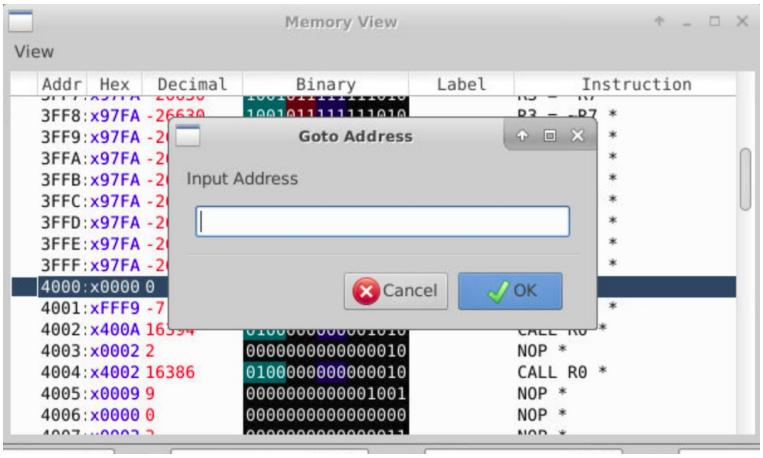


5. Now, complx is set up with the test that you failed! The nicest part of complx is the ability to step through each instruction and see how they change register values. To do so, click the step button. To change the number representation of the registers, double click inside the register box.

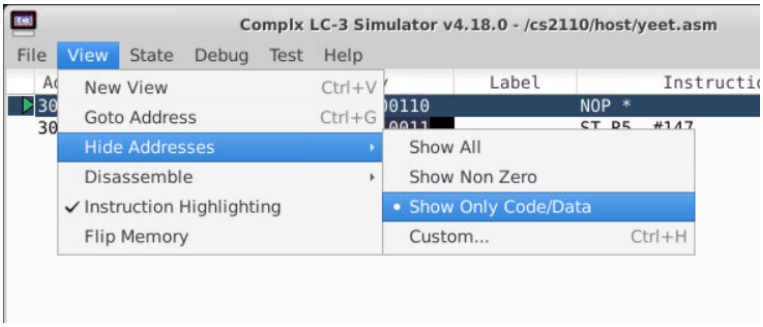
6. If you are interested in looking how your code changes different portions of memory, click the view tab and indicate 'New View'



7. Now in your new view, go to the area of memory where your data is stored by CTRL+G and insert the address



8. One final tip: to automatically shrink your view down to only those parts of memory that you care about (instructions and data), you can use View Tab → Hide Addresses → Show Only Code/Data. Just be careful: if you misclick and select Show Non Zero, it *may* make the window freeze (it's a known Complx bug).



6 Deliverables

Turn in the files `fib.asm`, `lowercase.asm`, and `gcd.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.

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

8 Demos

This project will be demoed. Demos are designed to make sure that you understand the content of the project and related topics. They may include technical and/or conceptual questions.

- Sign up for a demo time slot via Canvas **before** the beginning of the first demo slot. This is the only way you can ensure you will have a slot.
- If you cannot attend any of the predetermined demo time slots, e-mail the Head TA **before** the beginning of the first demo slot.

- If you know you are going to miss your demo, you can cancel your slot on Canvas with no penalty. However, you are **not** guaranteed another time slot. You cannot cancel your demo within 24 hours or else it will be counted as a missed demo.
- Your overall project score will be `((autograder_score * 0.5) + (demo_score * 0.5))`, meaning if you received a 90% on your autograder, but a 30% on the demo you would receive an overall score of 60%. **If you miss your demo you will not receive any of these points and the maximum you can receive on the project is 50%.**
- You will be able to makeup one of your demos at the end of the semester for half credit.

9 Rules and Regulations

9.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.
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.

9.2 Submission Conventions

1. When preparing your submission you may either submit the files individually to Canvas/Gradescope or you may submit an archive (zip or tar.gz only please) of the files. Both ways (uploading raw files or an archive) are exactly equivalent, so choose whichever is most convenient for you.
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.

9.3 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 Canvas/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 Canvas/Gradescope.
3. You are additionally responsible for ensuring that the collaborators list you have provided in your submission is accurate.

4. Projects turned in late receive partial credit within the first 48 hours. We will take off 30% of the points for a project submitted between 0 and 24 hours late, and we will take off 50% of the points for a project submitted between 24 and 48 hours late. We will not accept projects turned in over 48 hours late. This late policy is also in the syllabus.
5. 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.

9.4 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").
- Publishing your assignments on public repositories accessible to other students is unauthorized collaboration and thus Academic Misconduct.

Any of your peers with whom you collaborate in a conceptual manner with must be properly added to a **collaborators.txt** file. Collaborating with another student without listing that student as a collaborator is considered plagiarism.

9.5 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 Honor Code.** The 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*").
4. Publishing your assignments on public repositories 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. **If you are not sure about any aspect of this policy, please ask Dr. Conte.**