CS 2110 Homework 6
Subroutines and Calling Conventions

Annelise Lloyd, Sebastien Jankowski, Zaid Akkawi,
Jonathan Marto, Max Everest

Spring 2023

Contents

1 TLDR

1.1 Debugging Assembly

7.1 For information on debugging or the autograder, jump to the debugging section. Please note that Docker must be running, but it doesn't need to be within the image.

1.2 Factorial

Given a value n, calculate the factorial by implementing the Factorial and Multiply Subroutines. Return the final answer. The pseudocode can be found in section 3.1.1.

Below is a list of what we have provided you for this method.

- label named "MULTIPLY" where the multiply subroutine begins
- label named "FACTORIAL" where the factorial subroutine begins
- label named "STACK" where the start of the stack is located

1.3 Insertion Sort

Given an array of integers, you will be tasked with implementing an insertion sort to put them in ascending order. The pseudocode can be found in section 3.2.1.

Below is a list of what we have provided you for this method.

- label named "INSERTION_SORT" where the insertion sort subroutine begins
- label named "STACK" where the start of the stack is located
- values of the original array starting at x4000

1.4 Preorder Traversal

For this subroutine, you will write a subroutine to execute a preorder traversal on a binary tree and put the result into an array. The pseudocode can be found in section 3.3.1.

Below is a list of what we have provided you for this method.

- label named "PREORDER_TRAVERSAL" where the preorder traversal subroutine begins
- label named "STACK" where the start of the stack is located
- the nodes of the binary tree with value, address of the left node, and address of the right node, starting at x4000
- the result array starting at x4020

1.5 Advice from your TAs

- **More information on the LC-3 Calling Convention** can be found on Canvas under **Lab slides 9 and 10** and in **Lecture Slides L10 and L11**.
- More detailed information on each LC-3 instruction can be found in the **Patt/Patel book Appendix A (also on Canvas under LC3 Resources)**.

- When converting pseudocode into assembly, become human compilers and translate one line of pseudocode into one (or more) lines of assembly. One line at a time. Don't look at the pseudocode and try to optimize things in your head and write the entire function. Trust us when we say translating one line at a time is the way. So, put on blinders. Ignore previous and future lines of code.
- Don't assume a register still retains a value from a previous LD.
- Don't assume the condition code reflects the register you think it does (NZP).
- Checkout the Appendix for helpful resources
- The local autograder is a shell script which should be run in your local terminal (not the docker terminal).
- Remember that this HW will be demoed. There will be an announcement soon over the demo details!

The above strategy ofcourse produces inefficient code. But, it produces correct assembly code, every time. Not clever code. Not optimized code. Not short code. But correct code. So, if you ever find it difficult to write assembly, just go at it one step at a time :)

psst..also you could come to office hours or post on ed discussion but definitely try that advice first :DD

2 Overview

2.1 Purpose

Now that you've been introduced to assembly, think back to some high level languages you know such as Python or Java. When writing code in Python or Java, you typically use functions or methods. Functions and methods are called subroutines in assembly language.

In assembly language, how do we handle jumping around to different parts of memory to execute code from functions or methods? How do we remember where in memory the current function was called from (where to return to)? How do we pass arguments to the subroutine, and then pass the return value back to the caller?

The goal of this assignment is to introduce you to the Stack and the Calling Convention in LC-3 Assembly. This will be accomplished by writing your own subroutines, calling subroutines, and even creating subroutines that call themselves (recursion). By the end of this assignment, you should have a strong understanding of the LC-3 Calling Convention and the Stack Frame, and how subroutines are implemented in assembly language.

2.2 Task

You will implement each of the three subroutines (functions) listed below in LC-3 assembly language. Please see the detailed instructions for each subroutine on the following pages. The autograder checks for certain subroutine calls with arguments pushed in the correct order, so we suggest that you follow the provided algorithms when writing your assembly code. Your subroutines must adhere to the LC-3 calling convention.

1. `factorial.asm`
2. `insertionSort.asm`
3. `preorderTraversal.asm`

Please read the **TLDR** (listed above) for more helpful resources. Please check the rest of this document for some advice on debugging your assembly code, as well some general tips for successfully writing assembly code.

The assignment is due by 11:59 PM on July 6th, 2023. You could also submit the assignment by July 7th, 2023 for a late penalty of 25% of your overall grade.

2.3 Criteria

Your assignment will be graded based on your ability to correctly translate the given pseudocode for subroutines (functions) 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 deadlines and other related information.

You must obtain the correct 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 before and after the caller's JSR 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 convention correctly, each of these things will happen automatically.

While we will give partial credit where we can, your code must assemble with no warnings or errors. (Complx 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. Each function is in a separate file, so you will not lose all points if one function does not assemble. Good luck and have fun!

3 Detailed Instructions

3.1 Part 1

3.1.1 Factorial

The Factorial of a number is the product of that number and all of the numbers below it. In `factorial.asm`, we want you to implement two subroutines: `MULTIPLY` and `FACTORIAL` (see the following pseudocodes for more details). Note that the numbers that we will give you are strictly greater than or equal to 0. As a reminder, please do **not** change the names of provided subroutines. Otherwise, your submission will not pass the autograder.

For example:
factorial(0) should return 1
factorial(1) should return 1
factorial(2) should return 2
factorial(3) should return 6

3.1.2 Pseudocode

Here are the pseudocodes for these subroutines:

```
MULTIPLY(int a, int b) {
    int ret = 0;
    while (b > 0):
        ret += a;
        b--;
    return ret;
}

FACTORIAL(int n) {
    int ret = 1;
    for (int x = 2; x <= n; x++) {
        ret = MULTIPLY(ret, x);
    }
    return ret;
}
```

Note: since there are two loops in the `MULTIPLY` and `FACTORIAL` subroutines, make sure you use different label names for those loops. In general, you should not have two loops with the same name in the same program.

3.2 Part 2

3.2.1 Insertion Sort

For this part of this assignment, you will be implementing insertion sort on an array of integers in `insertionSort.asm`. As a reminder, please do **not** change the names of provided subroutines. Otherwise, your submission will not pass the autograder.

If you are unfamiliar with insertion sort and would like more information about it, see www.geeksforgeeks.org/recursive-insertion-sort/.

3.2.2 Pseudocode

Here is the pseudocode for this subroutine:

```
INSERTION_SORT(int[] arr (addr), int length) {
    if (length <= 1) {
        return;
    }

    INSERTION_SORT(arr, length - 1);

    int last_element = arr[length - 1];
    int n = length - 2;

    while (n >= 0 && arr[n] > last_element) {
        arr[n + 1] = arr[n];
        n--;
    }

    arr[n + 1] = last_element;
}
```

Let's do an example of insertion sort in action and see how the array looks throughout our above algorithm!

3	5	2	1	7	Given
3	5	2	1	7	INSERTION_SORT(arr, 4)
3	5	2	1		INSERTION_SORT(arr, 3)
3	5	2			INSERTION_SORT(arr, 2)
3	5				INSERTION_SORT(arr, 1)
3					Return
3	5				last_element_movement=0
2	3	5			last_element_movement=2
1	2	3	5		last_element_movement=3
1	2	3	5	7	last_element_movement=0

3.3 Part 3

3.3.1 Preorder Traversal

For this part of the assignment, you will write a recursive traversal subroutine for a binary tree in `preorderTraversal.asm`. The parameters of the function are the tree's root (provided as an address in memory), the array to store the result in (also provided as an address in memory), and the index in the array to store the next result in. As a reminder, please do **not** change the names of provided subroutines or otherwise your submission will not pass the autograder.

If you are unfamiliar with the preorder traversal and would like more information about it, see www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/.

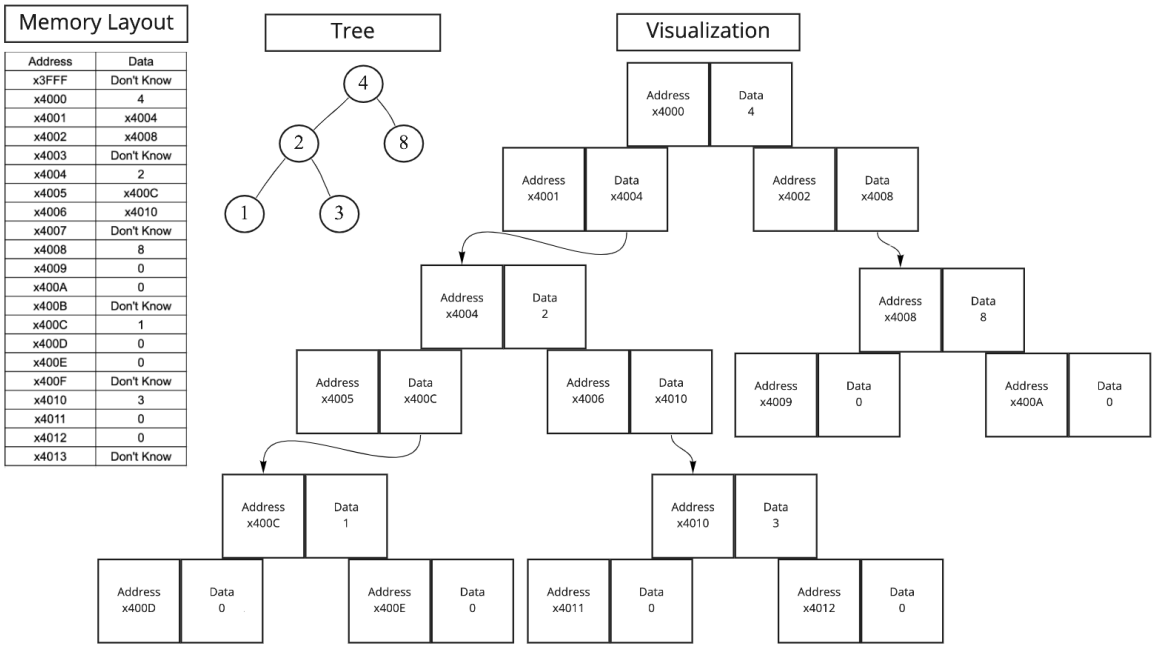
3.3.2 Binary Tree Data Structure

The below figure has 3 parts (the actual tree, the memory layout of that tree, and the visualization of that tree) depicting how each node in our tree is laid out in our memory. Each node will have three attributes: its value, its left node, and its right node. Note that each node is treated as an address in memory.

Given some node that lives at address x:

- mem[x] = node.data
- mem[x + 1] = node.left
- mem[x + 2] = node.right

For example, x4000 holds the data of a node, x4001 holds the address of the left child of that node, and x4002 holds the address of the right child of that node. For a leaf node (a node with no child) such as node 8 in our example, x4008 holds the data and x4009 and x400A both hold 0 since node 8 has no child nodes.



See https://www.geeksforgeeks.org/binary-tree-data-structure/ for more information regarding binary trees.

3.3.3 Pseudocode

Here is the pseudocode for this subroutine:

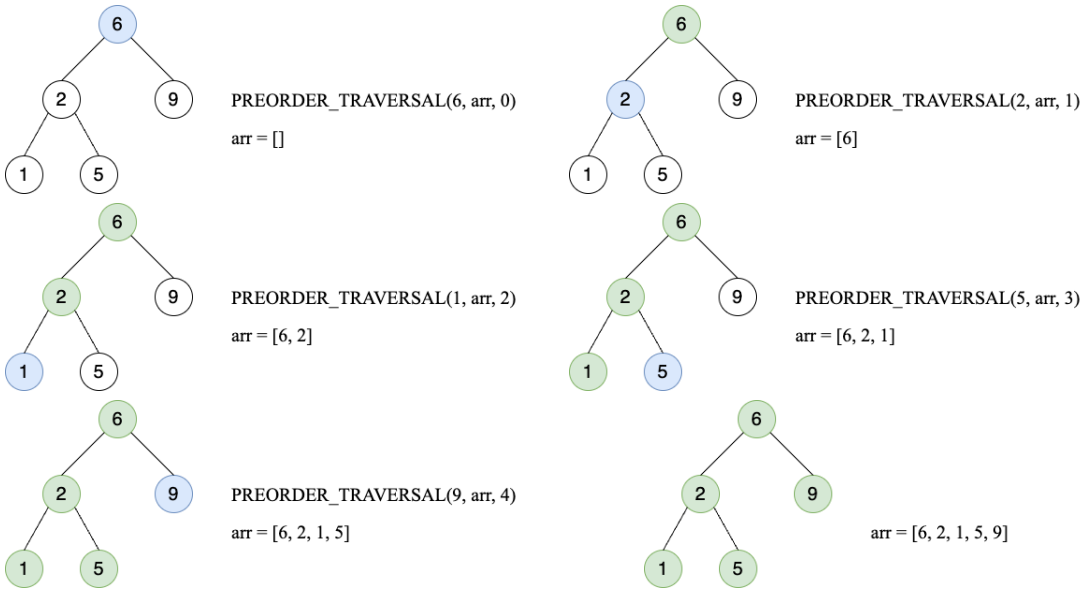
```
PREORDER_TRAVERSAL(Node root (addr), int[] arr (addr), int index) {
    if (root == 0) {
        return index;
    }

    arr[index] = root.data;
    index++;

    index = PREORDER_TRAVERSAL(root.left, arr, index);
    return PREORDER_TRAVERSAL(root.right, arr, index);
}

*Note: 0 is equivalent to NULL
```

Let's do an example of a preorder traversal in action on a binary tree and see how it builds our result array!



4 Autograder

To run the autograder locally, follow the steps below depending on 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`

Note: The checker may not reflect your final grade on this assignment. We reserve the right to update the autograder as we see fit when grading.

5 Deliverables

Turn in the following files to Gradescope:

1. `factorial.asm`
2. `insertionSort.asm`
3. `preorderTraversal.asm`

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

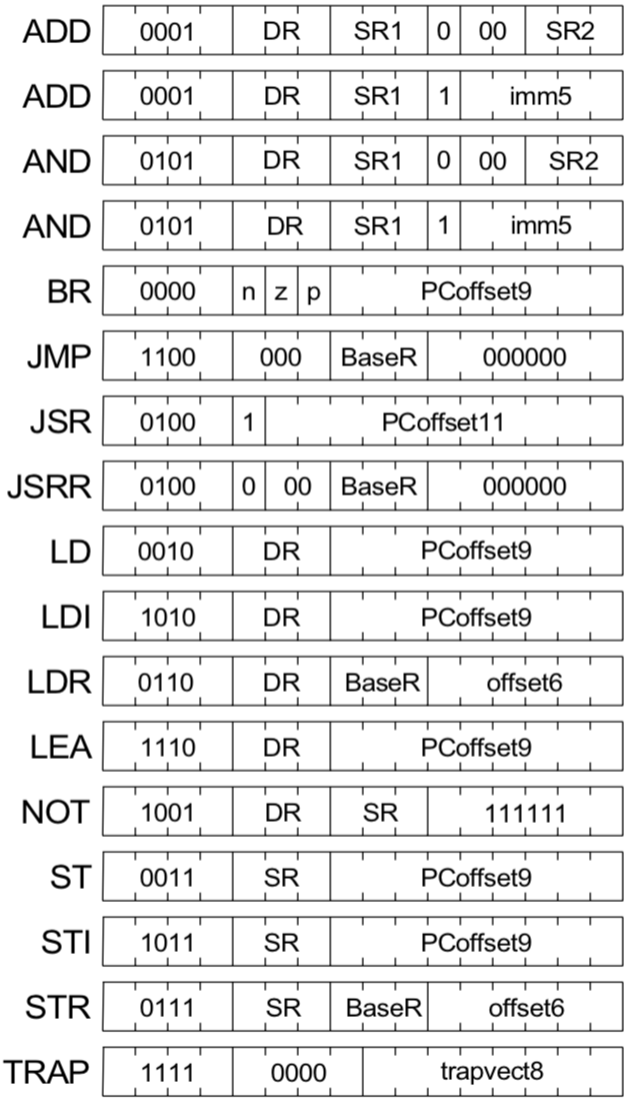
6 Demos

This homework will be demoed. The demos will be ten minutes long and will occur **IN PERSON**. Stay tuned for details as the due date approaches.

- 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 Max Everest (meverest7@gatech.edu) **before** the week of demos.
- If you know you are going to miss your demo, you may cancel your slot on Canvas with no penalty, as long as you cancel 24 hours in advance. However, you are **not** guaranteed another time slot. If you cancel within 24 hours of your demo, it will be counted as a missed demo.
- Your overall homework score will be `((homework_score * 0.5) + (demo_score * 0.5))`, meaning if you received a 90% on your homework, 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 homework is 50%.**
- You will be able to make up one of your missed demos at the end of the semester for half credit.

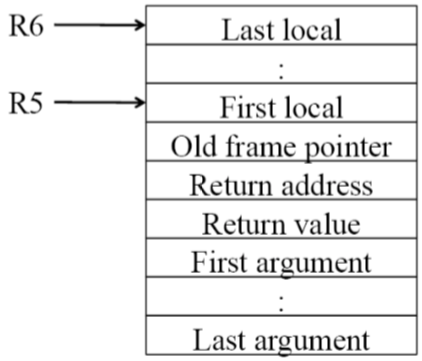
7 Appendix

7.1 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



8 Debugging LC-3 Assembly

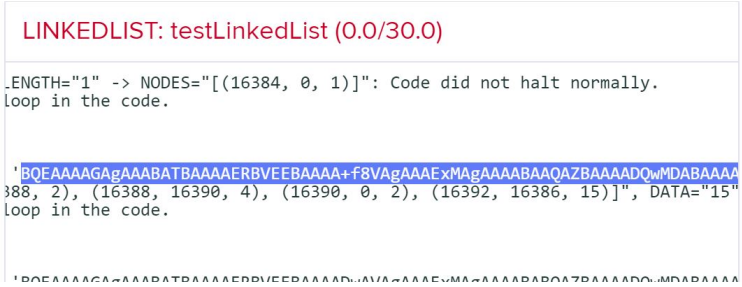
When you turn in your files on Gradescope for the first time, you may not receive a perfect score. Does this mean you change one line and spam Gradescope until you get a 100? No! You can use a handy Complx feature called "replay strings".

1. First off, we can get these replay strings in two places: the local grader, or off of Gradescope. When you run the autograder, you should see an output like this:

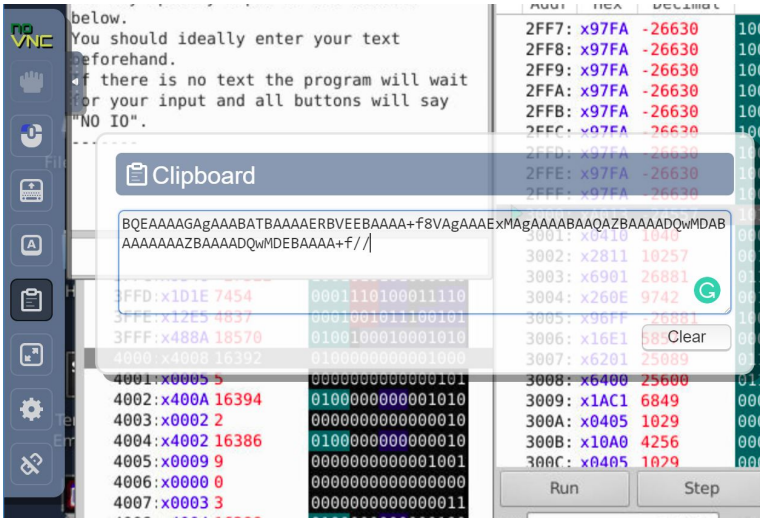
Copy the string, (everything between the apostrophes (')) excluding the initial " b' ".

For example, if your replay string is *b'asknd kansdasd=\\'*, you should paste *asknd kansdasd=* into complex.

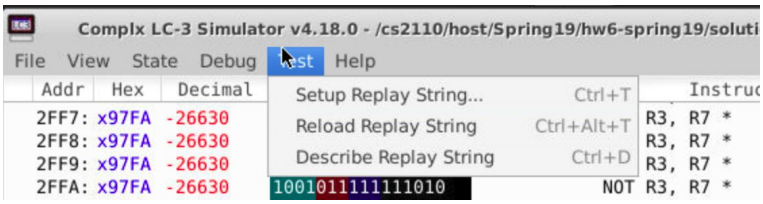
Side Note: If you do not have Docker installed, you can still use the tester strings to debug your assembly code. In your Gradescope error output, you will see a tester string. When copying, make sure you copy from the first letter to the final backslash and again, don't copy the quotations.



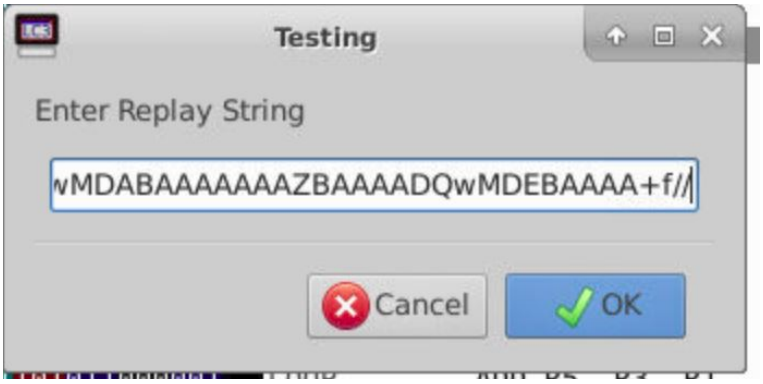
2. Secondly, navigate to the clipboard in your Docker image and paste in the string.



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

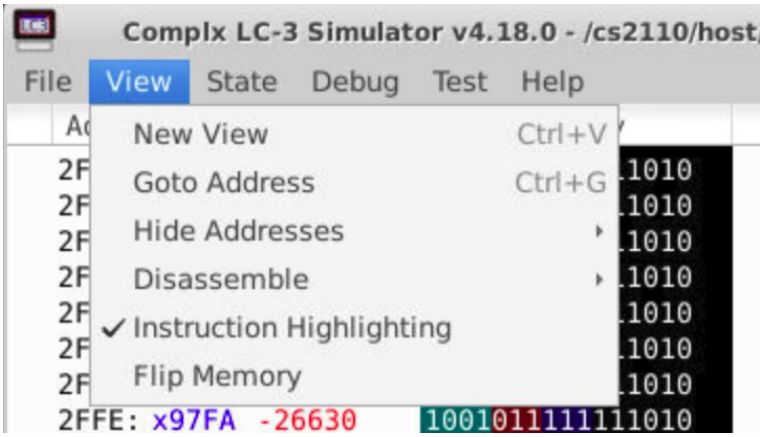


4. Now, paste your tester string in the box!

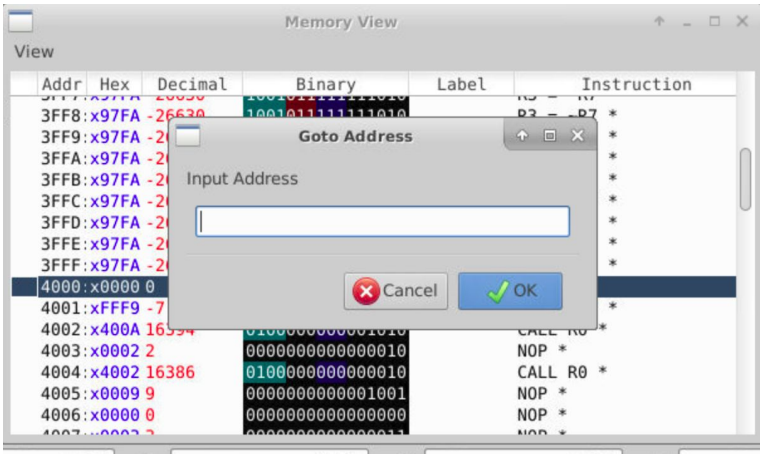


5. Now, Complx is set up with the test that you failed! The nicest part of Complx is the ability to step through each instruction and see how they change register values. To do so, click the step button. To change the number representation of the registers, double click inside the register box.

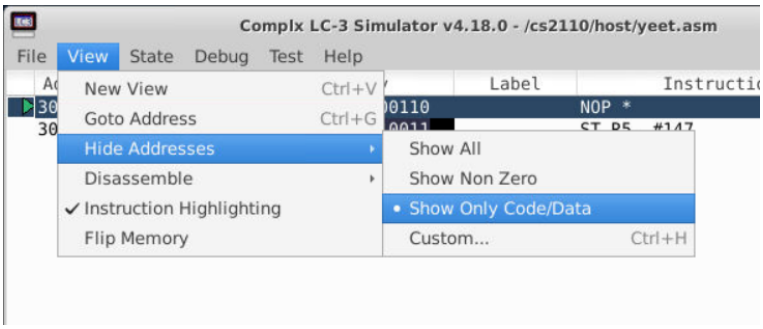
6. If you are interested in looking how your code changes different portions of memory, click the view tab and indicate 'New View'



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.



8.1 Appendix C: LC-3 Assembly Programming Requirements and Tips

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 3. You can randomize the memory and load your program by doing File - Randomize and Load.

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. Start the stack 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 .fills after **HALT** or RET).

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

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

9 Appendix D: Rules and Regulations

9.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. As such, please start assignments early, and ask for help early. 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. 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. 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. You must submit all files listed in the **Deliverables** section individually to Gradescope as separate files.

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.

9.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

9.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own.