CS 2110 Homework 6
Subroutines and Calling Conventions

Matthew Jin, Izabela Hadula, Tam Truong, John Ever

Summer 2021

Contents

1 Overview

1.1 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.

1.2 Task

You will implement each of the four subroutines (functions) listed below in LC-3 assembly language. Please see the detailed instructions for each subroutine on the following pages. We have provided pseudocode for each of the subroutines, and suggest that you follow these algorithms when writing your assembly code. Your subroutines must adhere to the LC-3 calling convention.

1. `gcd.asm`
2. `sort.asm`
3. `reverseCopy.asm`

More information on the LC-3 Calling Convention can be found on Canvas in the Lab Guides and in Lecture Slides L12 and L13. More detailed information on each LC-3 instruction can be found in the Patt/Patel book Appendix A (also on Canvas under LC3 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.

1.3 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 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 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!

2 Detailed Instructions

2.1 Part 1

2.1.1 Recursive GCD

For this part of the assignment, you will be recursively computing the greatest common divisor of two integers, both of which will be passed in as arguments. The greatest common divisor d of two integers a and b is defined as the greatest integer that divides both a and b. If you would like additional reading on how Euclid's Greatest Common Divisor Algorithm works, please read `https://crypto.stanford.edu/pbc/notes/numbertheory/euclid.html`.

2.1.2 Pseudocode

Following is the pseudocode for computing the value of the n-th element of the Fibonacci sequence. The pseudocode is as follows:

```
gcd(int a, int b) {
    if (a == b) {
        return a;
    } else if (a < b) {
        return gcd(b, a);
    } else {
        return gcd(a - b, b);
    }
}
```

2.2 Part 2

2.2.1 Recursive Insertion Sort

For this part of this assignment, you will be implementing insertion sort recursively on an array of integers. The parameters for the `sort` subroutine are the memory address of the first element in the array and an integer value representing the number of elements in the array. Following execution of this subroutine, the specified array will be sorted. Pseudocode for this section was adapted from https://www.geeksforgeeks.org/recursive-insertion-sort/; feel free to use this resource for more information about recursive insertion sort.

2.2.2 Pseudocode

The pseudocode for this function is as follows.

```
 sort(arr, n) {
    // base case
    if (n <= 1)
        return;

    //first, we sort the first n-1 elements
    sort(arr, n-1);

    //now, we need to insert the last element where it belongs in the array
    //when subroutine returns, first n-1 elements have been sorted
    last = arr[n-1];
    j = n-2;

    //if an element is greater than 'last', we move it up by one space in the array
    while (j >= 0 && arr[j] > last) {
        arr[j+1] = arr[j];
        j--;
    }

    arr[j+1] = last;
}
```

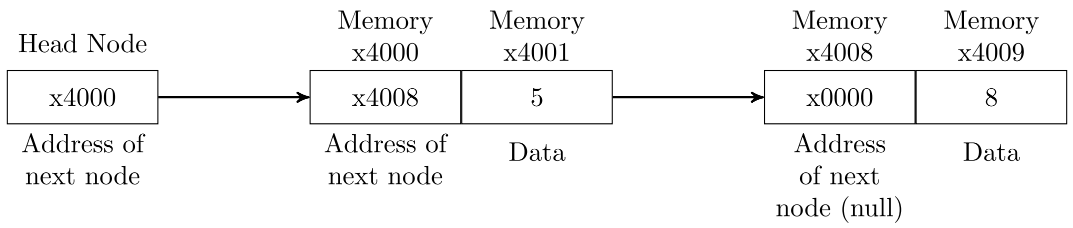
2.3 Part 3

2.3.1 Copy a Linked List in Reverse

For this part of the assignment, you will write a recursive subroutine to copy the data in the nodes of a linked list to an array in reverse order. The parameter of the function is a node representing the head node of the linked list.

2.3.2 Linked List Data Structure

The below figure depicts how each node in our linked list is laid out. Each node will have two attributes: the next node, and a value for that node.



2.3.3 Pseudocode

Here is the pseudocode for this subroutine:

```
reverseCopy(Node head) {
    // note that a NULL address is the same thing as the value 0
    if (head == NULL) {
        return 0;
    }
    Node next = head.next;
    int index = reverseCopy(next);
    ARRAY[index] = head.data;
    return index + 1;
}
```

3 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`

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.

4 Deliverables

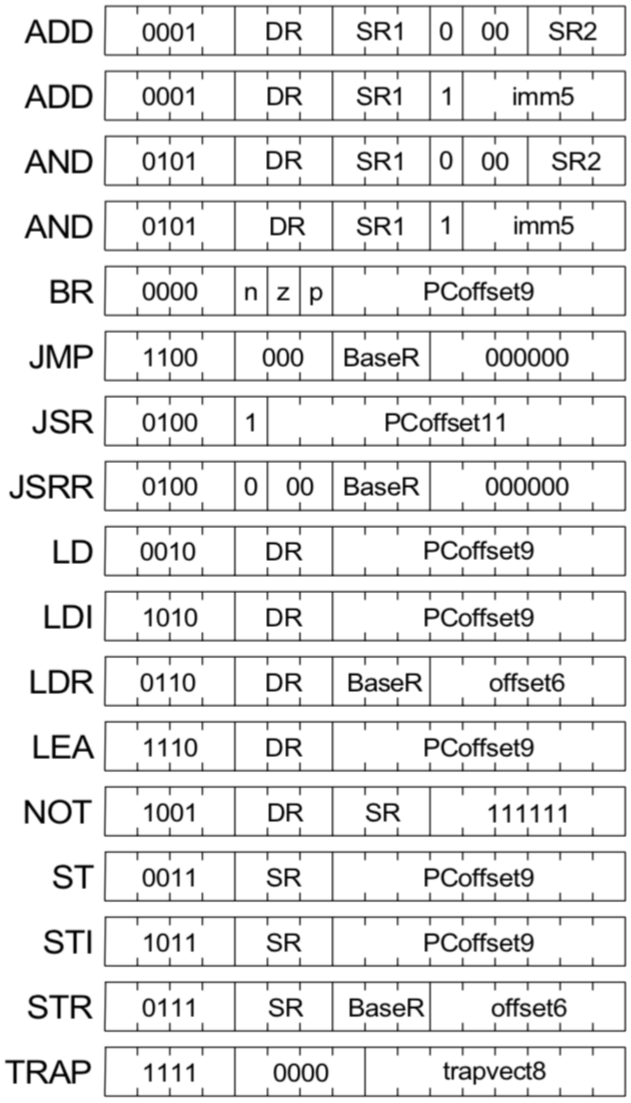
Turn in the following files to Gradescope:

1. `gcd.asm`
2. `sort.asm`
3. `reverseCopy.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.

5 Appendix

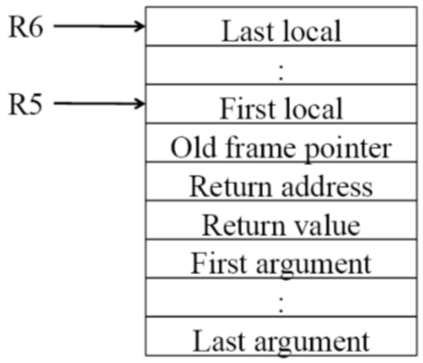
5.1 Appendix A: LC-3 Instruction Set Architecture



ADD	0001	DR	SR1	0	00	SR2
ADD	0001	DR	SR1	1	imm5	
AND	0101	DR	SR1	0	00	SR2
AND	0101	DR	SR1	1	imm5	
BR	0000	n	z	p	PCoffset9	
JMP	1100	000	BaseR	000000		
JSR	0100	1	PCoffset11			
JSRR	0100	0	00	BaseR	000000	
LD	0010	DR	PCoffset9			
LDI	1010	DR	PCoffset9			
LDR	0110	DR	BaseR	offset6		
LEA	1110	DR	PCoffset9			
NOT	1001	DR	SR	111111		
ST	0011	SR	PCoffset9			
STI	1011	SR	PCoffset9			
STR	0111	SR	BaseR	offset6		
TRAP	1111	0000	trapvect8			

Trap Vector	***Assembler Name***
`x20`	GETC
`x21`	OUT
`x22`	PUTS
`x23`	IN
`x25`	HALT

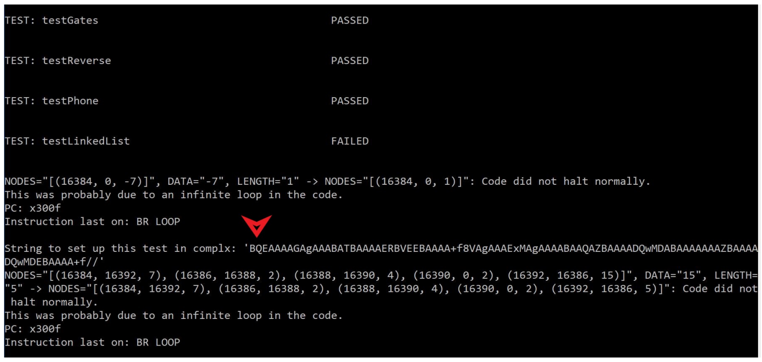
Device Register	***Address***
Keybd Status Reg	`xFE00`
Keybd Data Reg	`xFE02`
Display Status Reg	`xFE04`
Display Data Reg	`xFE06`



6 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".

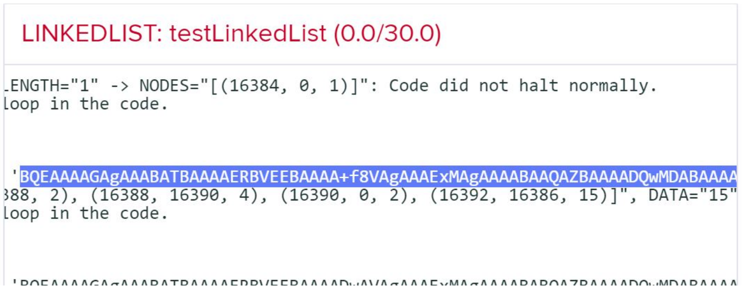
1. First off, we can get these replay strings in two places: the local grader, or off of Gradescope. When you run the autograder, you should see an output like this:



```
TEST: testGates                                         PASSED


TEST: testReverse                                       PASSED


TEST: testPhone                                         PASSED


TEST: testLinkedList                                    FAILED


NODES="[(16384, 0, -7)]", DATA="-7", LENGTH="1" -> NODES="[(16384, 0, 1)]": Code did not halt normally.
This was probably due to an infinite loop in the code.
PC: x300f
Instruction last on: BR LOOP

String to set up this test in complx: 'BQEAAAAGAgAAABATBAAAAERBVEEBAAAA+f8VAgAAAExMAgAAAABAAQAZBAAAADQwMDABAAAAAAAZBAAAA
DQwMDEBAAAA+f//'
NODES="[(16384, 16392, 7), (16386, 16388, 2), (16388, 16390, 4), (16390, 0, 2), (16392, 16386, 15)]", DATA="15", LENGTH=
"5" -> NODES="[(16384, 16392, 7), (16386, 16388, 2), (16388, 16390, 4), (16390, 0, 2), (16392, 16386, 5)]": Code did not
 halt normally.
This was probably due to an infinite loop in the code.
PC: x300f
Instruction last on: BR LOOP
```

   Copy the string, starting with the leading 'B' and ending with the final backslash. Do not include the quotation marks.

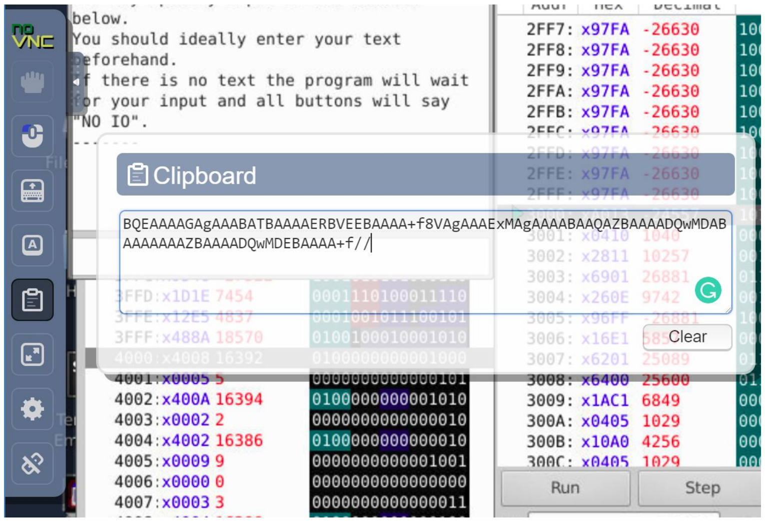
   Side Note: If you do not have Docker installed, you can still use the tester strings to debug your assembly code. In your Gradescope error output, you will see a tester string. When copying, make sure you copy from the first letter to the final backslash and again, don't copy the quotations.



LINKEDLIST: testLinkedList (0.0/30.0)

```
LENGTH="1" -> NODES="[(16384, 0, 1)]": Code did not halt normally.
loop in the code.


 'BQEAAAAGAgAAABATBAAAAERBVEEBAAAA+f8VAgAAAExMAgAAAABAAQAZBAAAADQwMDABAAAA
388, 2), (16388, 16390, 4), (16390, 0, 2), (16392, 16386, 15)]", DATA="15"
loop in the code.
```

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

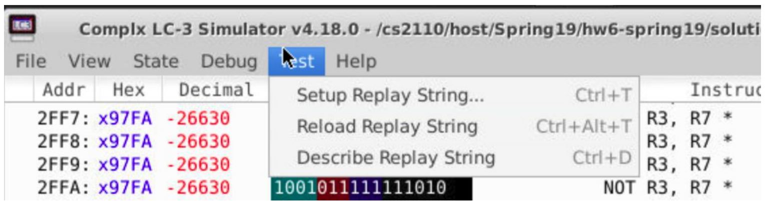


```
below.
You should ideally enter your text
beforehand.
If there is no text the program will wait
for your input and all buttons will say
"NO IO".
```

Clipboard

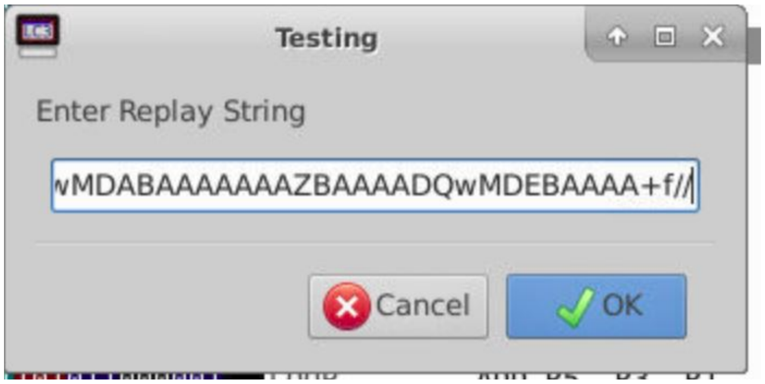
```
BQEAAAAGAgAAABATBAAAAERBVEEBAAAA+f8VAgAAAExMAgAAAABAAQAZBAAAADQwMDAB
AAAAAAAZBAAAADQwMDEBAAAA+f//
```

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


Complx LC-3 Simulator v4.18.0 - /cs2110/host/Spring19/hw6-spring19/soluti
File View State Debug Test Help
Setup Replay String... Ctrl+T
Reload Replay String Ctrl+Alt+T
Describe Replay String Ctrl+D

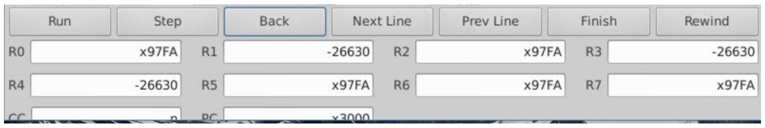
```
Addr   Hex    Decimal                              Instruc
2FF7: x97FA  -26630                          R3, R7 *
2FF8: x97FA  -26630                          R3, R7 *
2FF9: x97FA  -26630                          R3, R7 *
2FFA: x97FA  -26630  1001011111111010    NOT R3, R7 *
```

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



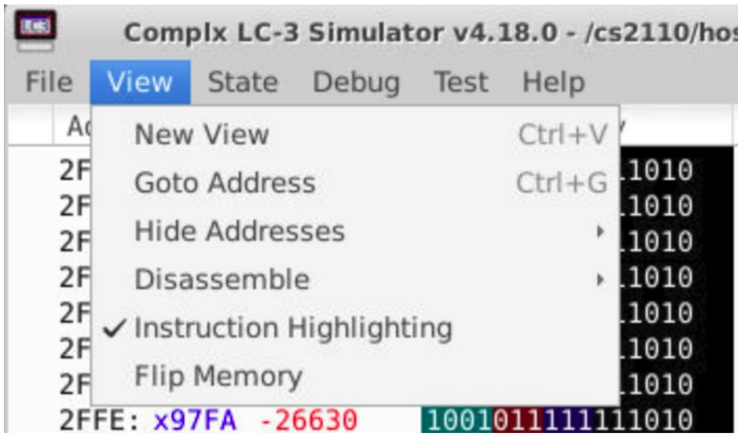
```
vMDABAAAAAAZBAAAADQwMDEBAAAA+f//
```

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.



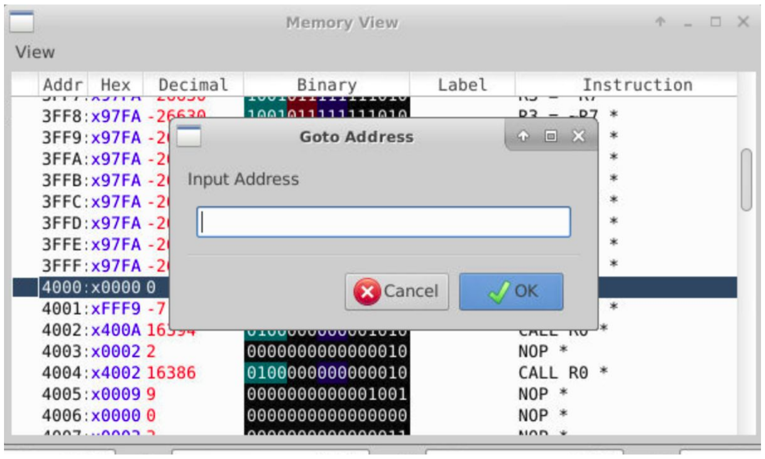
```
R0  x97FA   R1  -26630   R2  x97FA   R3  -26630
R4  -26630  R5  x97FA    R6  x97FA   R7  x97FA
CC  n       PC  x3000
```

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


Complx LC-3 Simulator v4.18.0 - /cs2110/host
File View State Debug Test Help
New View Ctrl+V
Goto Address Ctrl+G
Hide Addresses
Disassemble
✓ Instruction Highlighting
Flip Memory

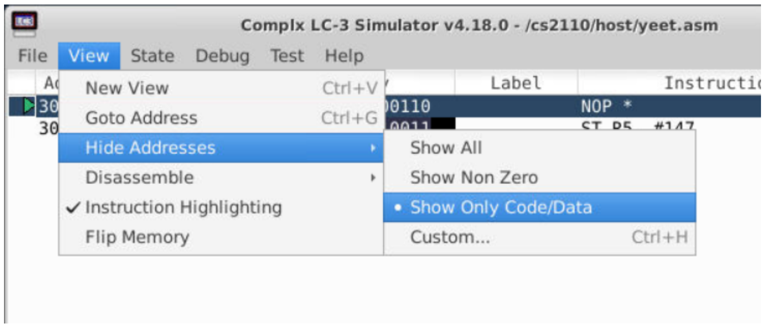
```
2FFE: x97FA -26630  1001011111111010
```

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



```
Addr Hex   Decimal  Binary            Label  Instruction
3FF8:x97FA -26630   1001011111111010         R3 = -R7 *
3FF9:x97FA -2                                         *
3FFA:x97FA -2                                         *
3FFB:x97FA -2                                         *
3FFC:x97FA -2                                         *
3FFD:x97FA -2                                         *
3FFE:x97FA -2                                         *
3FFF:x97FA -2                                         *
4000:x0000 0
4001:xFFF9 -7                                         *
4002:x400A 16394    0100000000001010         CALL R0 *
4003:x0002 2        0000000000000010         NOP *
4004:x4002 16386    0100000000000010         CALL R0 *
4005:x0009 9        0000000000001001         NOP *
4006:x0000 0        0000000000000000         NOP *
```

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.



6.1 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.

7 Appendix D: Rules and Regulations

7.1 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.

7.2 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.

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 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.

7.4 Syllabus Excerpt on Academic Misconduct

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

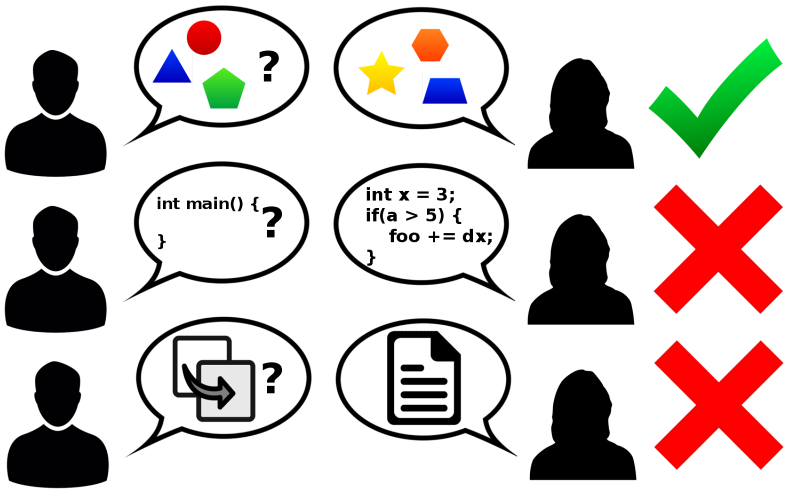
Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to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

7.5 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.


?
int main() {
}
?
int x = 3;
if(a > 5) {
foo += dx;
}
?