CS 2110 Summer 2017
Homework 7

Rules and Regulations

General Rules

1. Starting with the assembly homeworks, Any code you write (if any) must be clearly commented and the comments must be meaningful. You should comment your code in terms of the algorithm you are implementing we all know what the line of code does.

2. 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.

3. Please read the assignment in its entirety before asking questions.

4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.

5. 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.

Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere near the top in other files unless otherwise noted.

2. When preparing your submission you may either submit the files individually to T-Square or you may submit an archive (zip or tar.gz only please) of the files (preferred). You can create an archive by right clicking on files and selecting the appropriate compress option on your system.

3. If you choose to submit an archive please don't zip up a folder with the files, only submit an archive of the files we want. (See **Deliverables**).

4. Do not submit compiled files, that is, .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.

5. Do not submit links to files. We will not grade assignments submitted this way as it is easy to change the files after the submission period ends.

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 T-Square. When you submit the assignment, you should get an email from T-Square telling you that you submitted the assignment. If you do not get this email, that means that you did not complete the submission process correctly. 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 T-Square.

3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% (max points) penalty up until 5:55AM. *So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM.* You alone are responsible for submitting your homework before the grace period begins or ends; neither T-Square, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable. **This 25% is off of the maximum grade you would get**, this means that if you would have gotten a 50% then your late grade will be a 25%.

Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class.

Quizzes, timed labs and the final examination are individual work.

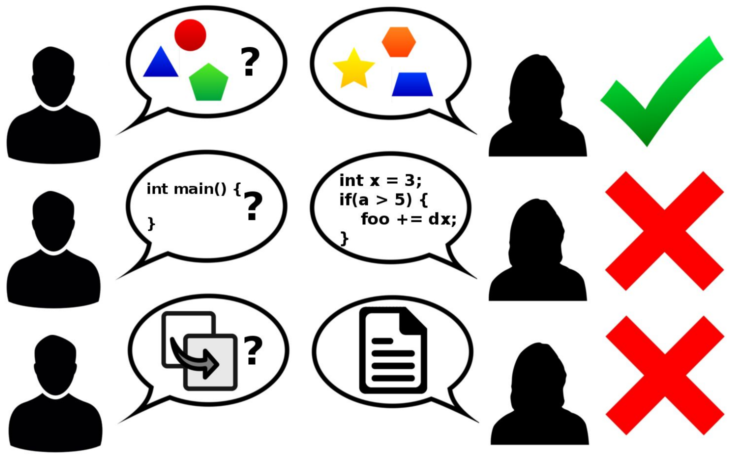
Homework assignments are collaborative. In addition, many, if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using electronic computer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

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, as well as help each other debug code. What you shouldn't be doing, however, is paired programming where you collaborate with each other on a low level. 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, and it is often the case that the recipient will simply modify the code and submit it as their own.



Objectives

The goal of this assignment is to get you familiar with the LC-3 calling convention using assembly subroutines to implement functions. You will be calling functions from within other functions, which will involve using the stack to store arguments, return values, the return address, the old frame pointer, saved registers, and local variables.

Overview

A few requirements:

1. Your code must assemble with **NO WARNINGS OR ERRORS**. To assemble your program, open the file with Complx. It will show any errors in pop-up windows.
2. Comment your code! This is especially important in assembly because it's much harder to interpret what's happening later, and you will be glad you left notes. Comments should show what registers are being used for, and what not so intuitive lines of code are actually doing. To comment code in LC-3 assembly, just type a semicolon (`;`) and the rest of the 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
   ```

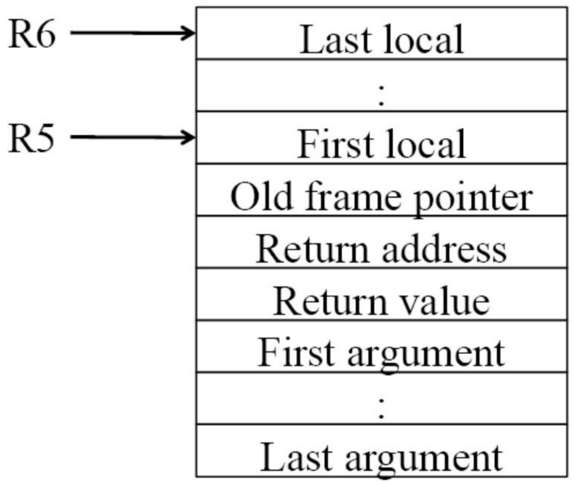
3. **DO NOT assume that anything in the LC-3 is already zero!** Treat the machine as if random values were stored in the register file and memory, and your program was loaded on top. NOTE: To test your code with randomized memory and registers, go to "File -> Randomize and Reload", and then select your assembly file.
4. Following from 3. You can randomize the memory and load your program by doing File → Randomize and Load.
5. 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.
6. 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.
7. **Do NOT** execute any data as if it were an instruction (meaning you should put .fills after HALT or RET).

8. Do not add any comments beginning with @plugin or change any comments of this kind.
9. **Test your assembly**. Don't just assume it works and turn it in.

LC-3 Calling Convention Overview

Since you will be using subroutines, you need a way of preserving the old return addresses and arguments before calling another subroutine which will trash these values. The stack is a convenient place to keep them. It would make sense to use the stack pointer for figuring out where to push the next value, but it is extremely inconvenient for loading a particular value into a register if you have pushed a whole bunch of things onto the stack. You would have to keep track of how many values got pushed into the stack to find the exact address since you will not necessarily be using those values in the order they were pushed into the stack. This is where the frame pointer comes in handy.

In the LC-3 calling convention, the caller pushes all the arguments into the stack before calling the subroutine. Then the callee reserves space for the return value, pushes the return address (R7), and pushes the old frame pointer (R5) into the stack. The frame pointer should then be modified to point to the address right above where the old frame pointer was stored on the stack. You now know precisely where the old frame pointer, return address, and arguments are stored in relative to the frame pointer regardless where the stack pointer is pointing at. Using this will make debugging and cleanup much easier.



Tips and Tricks

- Comment your code! It is very easy to get lost in assembly.
- The first thing you should do when writing any function is to setup the frame at the top of the function and teardown the frame at the bottom. I recommend a common "RETURN" at the bottom, this way multiple ways to return would just branch there.
- Complx is a very powerful debugger, offering the ability to step forward through code, step backwards through code, and setting breakpoints. When you run your code and it doesn't work, step through it line by line and ensure all your registers and the stack are correct at each step.
- **Do not assume registers have the same values when a function returns.** If you store argument 1 in R0, then you call another function, R0 might get overwritten. This is why local variables are stored on the stack. Reload values from the stack, treat registers R0-R4 as random garbage.
- Inspecting the stack in Complx can be done in a separate view: hit Ctrl+V to open a new memory view then hit Ctrl+G to jump to address xF000 for the stack.
- **Testing Your Code:**
    - Tests are run on the terminal: `lc3test xmlfile_test.xml asmfile.asm`
    - **These tests are not comprehensive. If you pass them that means there is a high likelihood of the function being correct.** We will be using our own more comprehensive tests for grading. It is recommended you modify the XML files to create more tests. You may share these tests on Piazza as well.
    - **MAKE SURE COMPLX SAYS AT LEAST VERSION 4.15.6**

Part 1 - McCarthy 91 (mc91.asm)

For this first part, you will be implementing the McCarthy 91 function in assembly. The function is defined as this:

$$M(n) = \begin{cases} n - 10, & \text{if } n > 100 \\ M(M(n+11)), & \text{if } n \leq 100 \end{cases}$$

The neat thing about this function is that for any value of $n \leq 100$, the result of $M(n)$ will be 91 and for any value of $n > 100$, $M(n) = n - 10$. All results after $n = 101$ will continue to increase by one.

Pseudo-code:

```
int mc91(int n) {
    if (n > 100) {
        return n - 10;
    } else {
        return mc91(mc91(n + 11));
    }
}
```

`N` is a label for a `.fill` in the template file. You will store your answer in the label `ANSWER`.

Part 2 - Powers Of 2 (powersOf2.asm)

For this part, you will write a subroutine to calculate powers of 2 with a recursive formula.

Notice that `temp1` and `temp2` here are local variables and must be saved to the stack in its proper location. Failure to do this will result in a loss of points. We are testing if you can follow the calling convention and handling the stack!

Pseudo-code:

```
int powersOf2(int n) {
    int temp1 = 0;
    int temp2 = 0;

    if (n == 0)
        return 1;

    if (n == 1)
        return 2;

    temp1 = powersOf2(n - 1);
    temp2 = powersOf2(n - 2);
    return 3 * temp1 - 2 * temp2;
}
```

`N` is a label for a `.fill` in the template file. You will store your answer in the label `ANSWER`.

Part 3 - BST Height (bst_height.asm)

For this part, you will calculate the height of a binary search tree. Like the previous two problems, we will be providing you with a template file where you call the subroutine with the proper inputs using the LC-3 Calling Convention.

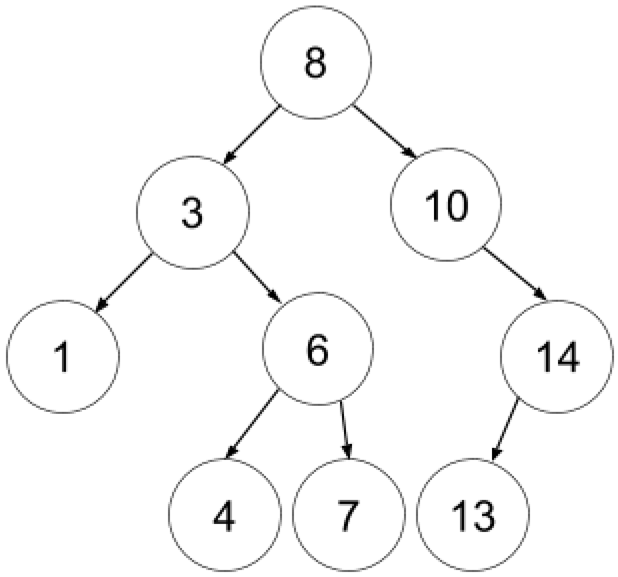
Notice that `leftheight` and `rightheight` here are local variables and must be saved to the stack in its proper location. Failure to do this will result in a loss of points. We are testing if you can follow the calling convention and handling the stack!

Pseudo-code:

```
int bst_height(int* arr, int index){
    if ( *(arr + index) == 0 )
        return 0;
    int leftheight = bst_height(arr, 2 * index);
    int rightheight = bst_height(arr, 2 * index + 1);
    if (leftheight > rightheight)
        return leftheight + 1;
    else
        return rightheight + 1;
}
```

Both inputs for this subroutine are supplied in labels `ARRAY` and `INDEX`, and the result should be stored in `ANSWER`. You must set up the initial call to `bst_height` and store the result in `ANSWER`.

Below is an example of a Binary Search Tree, with height 4.



All you need to know about Binary Search Trees is the following:

- Binary search trees are made of nodes (a circle), which have data (a number) and two children nodes (left and right arrows).
- Binary search trees can be implemented in arrays very easily. If a node is in an array at index i, then the left child node is found at 2*i, and the right child node is found at 2*i+1.
- The first (head) node in a binary search tree implemented in an array is at index 1.
- The height of a node is equal to the maximum height of its children, plus one.

More about Binary Search Trees: http://en.wikipedia.org/wiki/Binary_search_tree

NOTE: Since you are to follow the code on the previous page, you do not need to know exactly how binary search trees work!

Deliverables

Please submit your assignment by the deadline on T-Square and **put your name at the top of EACH file you submit**. You must submit the following files:

- ❏ `mc91.asm` from Part 1 of this assignment
- ❏ `powersOf2.asm` from Part 2 of this assignment
- ❏ `bst_height.asm` from Part 3 of this assignment

Please only submit **the above files or an archive (zip and tar.gz only) of ONLY those files**. Do not submit those files inside a folder. Please do not make a RAR archive.