CS2110 Fall 2016
Homework 7

Authors: **Roi Atalla**, **Dilara Soylu**, and **Sydney Young**

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 in the file, near the top, for 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. 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% 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.

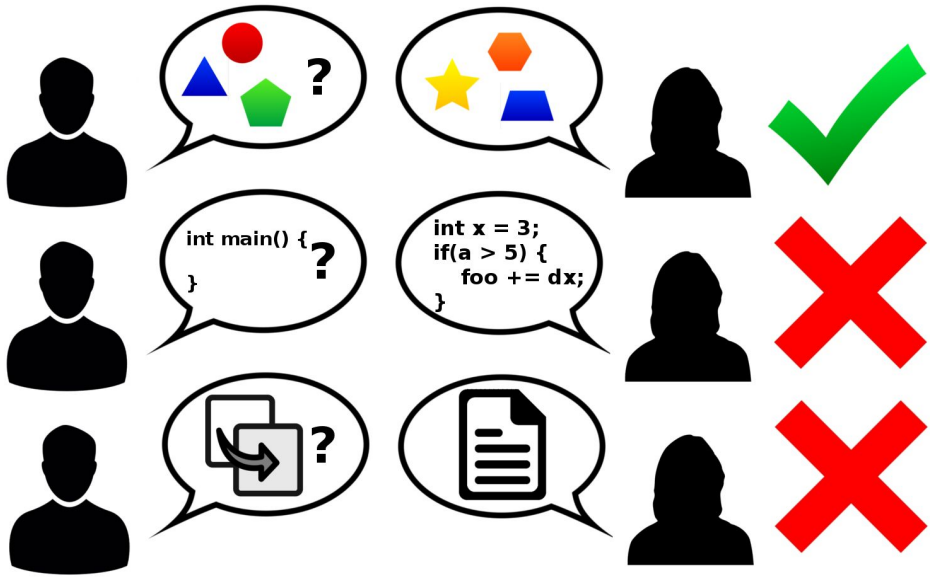
Syllabus Excerpt on Academic Misconduct

> 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, 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 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.

   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 your program was loaded into a machine with random values stored in the memory and register file.

4. Following from 3. You can randomize memory by using the menu option State → Randomize. And then load your program by saying File → Load Over.

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.

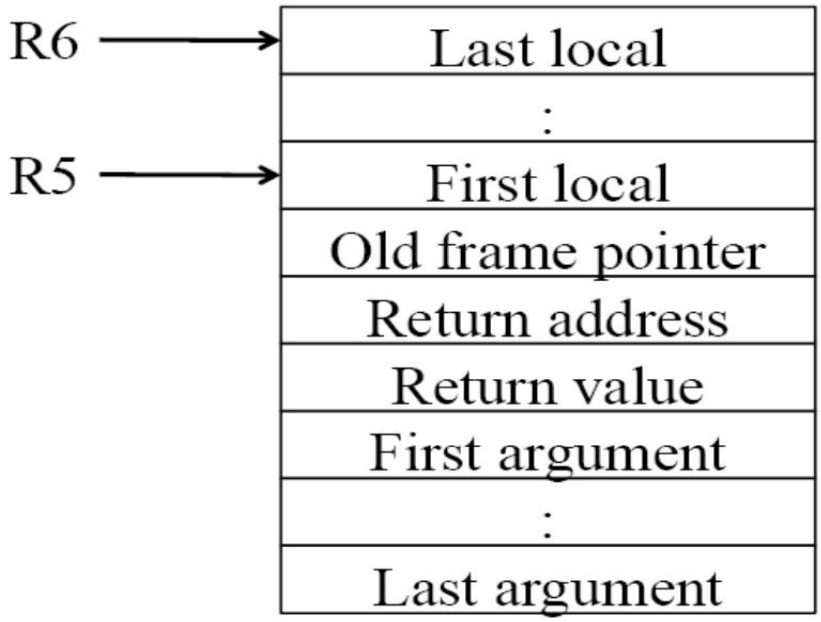
7. Do NOT execute any data as if it were an instruction (meaning you should put .fills **after** HALT or RET).

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 into 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 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.



Part 1 – Operations 1.0 (operations.asm)

You will be writing an operations function that takes two arguments (a and b) and an operation code (op) and returns the result of the requested operation. For this part, you will implement two operations: XOR and MULT. The main function (up at .orig x3000) has been written for you and the skeleton for OPERATIONS, XOR, and MULT are given.

Here's the pseudo-code:

```
int operations(int a, int b, int op) {
     if(op == 0) {
          return xor(a, b);
     } else if(op == 1) {
          return mult(a, b);
     } else {
          return -1;
     }
}

// NOTE: there are two local variables here that must be
// stored on the stack.
int xor(int a, int b) {
     // !(a & b) & !(!a & !b)

     int and1 = a & b;

     int and2 = !a & !b;

     return !and1 & !and2;
}

// NOTE: there is one local variable here that must be
// stored on the stack
int mult(int a, int b) {
     int sum = 0;
     while(b > 0) {
          sum += a;
          b--;
     }
     return sum;
}
```

Part 2 – Extending operations: 2.0

In this part, you will be extending your operations function to support two additional functions: div and mod. **These two must be implemented recursively**. You will be implementing these in operations.asm alongside your Part 1 code. Part 3 is given to test them.

Here's the pseudo-code (operations is updated here):

```
int operations(int a, int b, int op) {
     if(op == 0) {
          return xor(a, b);
     } else if(op == 1) {
          return mult(a, b);
     } else if(op == 2) {
          return div(a, b);
     } else if(op == 3) {
          return mod(a, b);
     } else {
          return -1;
     }
}

// Assume ‘a’ and ‘b’ are positive
// NOTE: there are no local variables in this function
int div(int a, int b) {
     if(a < b) {
          return 0;
     }

     return 1 + div(a - b, b);
}

// Assume ‘a’ and ‘b’ are positive
// NOTE: there are no local variables in this function
int mod(int a, int b) {
     if(a < b) {
          return a;
     }

     return mod(a - b, b);
}
```

Part 3 – Extending operations: 3.0

Lastly for operations, you will be extending it with one more function call. This mystery function implementation is already given to you! All you need to do is call it in the operations function. **Beware: the mystery function depends on your Part 2 DIV and MOD working properly!** This is a great way to test your Part 2 functions.

Here's the pseudo-code (operations is updated here):

```
int operations(int a, int b, int op) {
     if(op == 0) {
          return xor(a, b);
     } else if(op == 1) {
          return mult(a, b);
     } else if(op == 2) {
          return div(a, b);
     } else if(op == 3) {
          return mod(a, b);
     } else if(op == 4) {
          return mystery(a); // NOTE: mystery only takes 'a'
     } else {
          return -1;
     }
}

// Assuming 'a' is positive
// a | result
// 1 | 0
// 2 | 1
// 3 | 7
// 4 | 2
// 5 | 5
// 6 | 8
// 7 | 16
// 8 | 3
int mystery(int a) {
     // implementation given
     // requires working DIV and MOD
}
```

Part 4 – Strings (palindrome.asm)

For this part, you will be playing with strings. The first simple step is to get a recursive string length function working. With this, you will implement a **recursive** palindrome checker that is **case-insensitive**. This means "racecar" and "rAcECaR" are both valid palindromes. The main function has been given to you.

Here's the pseudo-code:

```
void main() {
     addr_of_string string = "rAcECaR";
     int strlen = length(string);
     int answer = palindrome(string, string + strlen - 1);
}

// NOTE: there is one local variables in this function
int length(addr_of_string string) {
     char c = mem[string]. // Load value from memory
     if(c == 0) {
          return 0;
     }
     return 1 + length(string + 1);
}

// NOTE: there are two local variable in this function
int palindrome(addr_of_string start, addr_of_string end) {
     if(start >= end) {
          return 1;
     }

     char c1 = mem[start]; // same as above
     char c2 = mem[end]; // same as above
     if(c1 >= 'a') {
          c1 -= 32; // turn character to upper-case
     }
     if(c2 >= 'a') {
          c2 -= 32; // turn character to upper-case
     }

     if(c1 == c2) {
          return palindrome(start + 1, end - 1);
     }

     return 0;
}
```

Deliverables

Remember to put your name at the top of EACH file you submit.

The files should be names exactly as given.

operations.asm
palindrome.asm