CS 2110 Summer 2017
Homework 6

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.

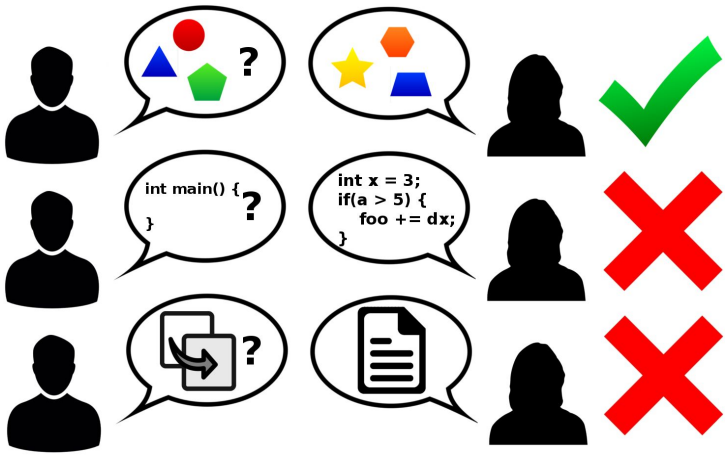
What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own

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 help you become comfortable coding in the LC-3 assembly language. This will involve writing small programs, reading input from keyboard, printing to the console, and converting from high-level code to assembly.

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. **DO NOT** use the following instructions: `JMP`, `JSR`, `JSRR`.
5. **DO NOT** write any `TRAP`s.
6. **DO NOT** execute any data as if it were a function (meaning your `.fill`'s should go after `HALT`).
7. **DO NOT** add any comments beginning with `@plugin` or change any pre-existing comments of this kind.
8. **PLEASE** test your assembly code. Don't assume it just works!

9. Automatic testers for each file have been provided with this assignment, as .xml files. Use the `lc3test` command from the terminal to run these testers on your code. Run "`lc3test`" in the terminal to see usage information. **The grade you receive from an automatic tester is NOT representative of your grade on the assignment. The TAs will check your submission with a different set of automatic tests. Provided tests are there to help you complete the assignment!**

Assembly Overview

For this assignment, you will be writing three assembly programs. The purpose of this assignment is to get you familiar and more comfortable with writing assembly code. For each part of this homework, we will supply you with pseudo-code, and you must write an equivalent assembly program. You have been given skeleton files for this assignment. We recommend following the pseudo-code as closely as possible.

```
.orig x3000
    LEA R0, HW      ; Load the address of the string
    PUTS            ; Output the string
    HALT            ; Stop executing instructions
HW .stringz "Hello World.\n"
.end
```

Above is a simple assembly program that prints "Hello World." and a new line to the console. Semicolons denote comments. You don't need a semicolon after every line, only before comments. `.orig` is a pseudo-op. Pseudo-ops are "guidelines" for the assembler that are not actually assembly instructions. For example, `.orig x3000` tells the assembler to place this block of code at location `x3000`, where the LC-3 starts executing code. Therefore, `LEA R0, HW` is at address `x3000`, `PUTS` is at address `x3001`, and so on.

After the `.orig` line, your actual assembly code begins. `PUTS` is a pseudonym for a `TRAP` call that prints a string whose address is stored in `R0`, and `HALT` is a pseudonym for a `TRAP` that stops the LC-3. `.stringz` is a pseudo-op that stores a particular string of characters in memory, followed by a zero. In the above example, '`H`' is stored at `x3003`, '`e`' is stored at `x3004`, and so on. `.end` tells the assembler where to stop reading code for the current code block. Every `.orig` statement must have a corresponding .end statement. Some other pseudo-ops you should know are:

`.fill` [`value`] which puts a given value into that memory location
`.blkw` [`n`] which reserves the next `n` memory locations, filling them with 0.

Another `TRAP` instruction for interacting with the console is `OUT`. This instruction prints a character from `R0` to the console. Below is a short example program that prints the letter '`A`' to the console. The ASCII value for '`A`' is 65, which is used by `OUT`.

```
.orig x3000
    LD R0, LETTER
    OUT
    HALT
LETTER .fill 65
.end
```

After writing your assembly code, you can test it using Complx. Complx is a full-featured LC-3 simulator, and we recommend running your code in the following fashion:

1. Go to the "File" menu, and select "Randomize and Reload". This randomly assigns values to every memory location and register, and loads your assembly file over the randomized memory.
2. Press the "Run" button to run your code until a `HALT` instruction or breakpoint is hit. You can also click "Step" to execute one instruction at a time, or "Step back" to go back one executed instruction.

One more thing: it is very important that you use Appendix A in the book to your advantage. Make sure you know what an instruction does before using it! Take a look at this instruction: `LD R2, 5`. What exactly is this instruction doing? Is it loading the number `5` into `R2`? Remember that the bits preceding the destination register are based on the `PCOFFSET9`: this is not an immediate value, you are merely loading data from an address location.

Part 1 - Absolute Value (absolute.asm)

Write an assembly program that returns the absolute value of `U`.

Pseudo-code:

```
if (u < 0) {
    answer = -u;
} else {
    answer = u;
}
```

`U` is a label for a `.fill` in the template file. It will be any integer in the range -32,767 to 32,767, inclusive. You will store your answer in the label `ANSWER`.

Part 2 - Is Multiple (multiple.asm)

Write an assembly program that finds whether `A` is a multiple of `B`. If `A` is a multiple of `B`, then return 1, otherwise return 0.

Pseudo-code:

```
while(a > b) {
    a = a - b;
}
if (a == b) {
    return 1;
} else {
    return 0;
}
```

`A` and `B` are labels for a `.fill` in the template file. They are both integers in the range 1 to 32,767, inclusive. You will store your answer in the label `ANSWER`.

Part 3 - Historical Timeline (timeline.asm)

Write an assembly program that prints out a historical timeline to the console. Given a year, tell us the important dates which precede it and tell us the important dates succeeding it. You will tell us these dates by printing out strings. The function will take in `N`, a number between 1 and 15 inclusively. `N` will then be used to index `YEAR_ARR` and `EVENT_ARR`.

This tables visualizes the arrays and how one indexes them:

N	YEAR_ARR	EVENT_ARR
1	1607	John Smith founded Jamestown in 1607
2	1776	The Declaration of Independence was signed in 1776
3	1788	The Constitution was ratified in 1788
4	1861	The Civil War began in 1861
5	1879	Thomas Edison invented the lightbulb in 1879
6	1885	The Georgia Institute of Technology was founded in 1885
7	1890	Yosemite National Park was created in 1890
8	1917	The US entered WWI in 1917
9	1941	Japan attacked Pearl Harbor in 1941
10	1955	The Civil Rights movement began in 1955
11	1961	The Vietnam War began in 1961
12	1969	Apollo 11 landed on the moon in 1969
13	1975	Bill Gates founded Microsoft Corporation in 1975
14	1985	Super Mario Bros. debuted in 1985
15	1991	The Cold War formally ended in 1991

Pseudo-code:

```
n--;
if (n == 0) {
    puts("Nothing happened before ");
    puts(YEAR_ARR[n]);
    putc('\n');
} else {
    puts("In the years before ");
    puts(YEAR_ARR[n]);
    puts("...");
    putc('\n');
    var i = n;
    while (i > 0) {
        i--;
        puts(EVENT_ARR[i]);
        putc('\n');
    }
}

if (n == max) {
    puts("And nothing happened after ");
    puts(YEAR_ARR[n]);
    putc('\n');
} else {
    puts("And in the years after ");
    puts(YEAR_ARR[n]);
    puts("...");
    putc('\n');
    var i = n;
    while (i < max) {
        i++;
        puts(EVENT_ARR[i]);
        putc('\n');
    }
}
```

Here, `YEAR_ARR` and `EVENT_ARR` are arrays with pointers to different `.stringz` values. `YEAR_ARR` contains the pointers to the years and `EVENT_ARR` contains pointers to the

corresponding events. These values are all supplied with the template file along with the label's for `NEWLINE` and `ELLIPSIS`, as well as a pre-defined value for `MAX`.

For example, if the value of `N` is 4 then the output should be:

```
In the years before 1861...
The Constitution was ratified in 1788
The Declaration of Independence was signed in 1776
John Smith founded Jamestown in 1607
And in the years after 1861...
Thomas Edison invented the lightbulb in 1879
The Georgia Institute of Technology was founded in 1885
```
...continues...
```
The Cold War formally ended in 1991
```

Don't forget the edges case either! For example, when `N` is 1. Make sure to look at the pseudo-code!

Hint: Because `YEAR_ARR` and `EVENT_ARR` are arrays containing pointers to strings, you cannot simply offset the array index by the day and print what is there (since what is there is what we call a *pointer*). It's a step in the right direction though.

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:

- ❏ `absolute.asm` from Part 1 of this assignment
- ❏ `multiple.asm` from Part 2 of this assignment
- ❏ `timeline.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.