CS2110 Spring 2014
Homework 6

Version: 1

Rules and Regulations

Academic Misconduct
Academic misconduct is taken very seriously in this class. Homework assignments are collaborative. However, each of these assignments should be coded by you and only you. This means you may not copy code from your peers, someone who has already taken this course, or from the Internet. You may work with others **who are enrolled in the course,** but each student should be turning in their own version of the assignment. Be very careful when supplying your work to a classmate that promises just to look at it. If he/she turns it in as his own you will both be charged.

We will be using automated code analysis and comparison tools to enforce these rules. **If you are caught you will receive a zero and will be reported to Dean of Students.**

General Rules

1. In addition 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.
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.

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.
2. When preparing your submission you may either submit the files individually to T-Square (preferred) or you may submit an archive (zip or tar.gz only please) of the files.
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.
4. 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.

Objectives

The goal of this assignment is to help you become comfortable coding in the LC-3 assembly language. This will involve the creating small programs, reading input from the 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 complain if there are any issues.
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 left notes to let you know sections of code or certain instructions are contributing to the code. Comment things like 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 semi-colon (;), 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. You may NOT use the instructions JMP, JSR, or JSRR. Do not write any TRAPs.
6. Do NOT execute any data as if it were an instruction (meaning you should put .fills after halt).
7. Do not add any comments beginning with @plugin or change any comments of this kind.
8. Test your assembly. Don't just assume it works and turn it in.

Assembly Overview

For this assignment, you will be writing 4 assembly programs. The purpose of this assignment is to get your familiar and more comfortable with writing low-level assembly code.

For each part of this homework, we will supply you with pseudo-code, and you must write the equivalent assembly program. Please make your code as similar to the pseudo-code as you can!

You've been given framework files for this assignment, but I'll tell you a little bit about how the assembly files in this class look.

```
.orig x3000
      LEA R0, HW        ;Load the address of the string
      PUTS              ;Output the string
      HALT              ;Stop Running
      HW    .stringz  “Hello World.¥n”
.end
```

This is a simple assembly program that prints “Hello World.” and a new line to the console.
“;” denotes a comment. You don't need a semi-colon after every line, only before comments.
“.orig” is a pseudo-op. Pseudo-ops are special instructions for the assembler that are not actually assembly instructions. “.orig x3000” tells the assembler to place this block of code at x3000, which is where the LC-3 starts code execution. Therefore “LEA R0, HW” is at address x3000, “PUTS” is at address x3001, etc.
Next is your assembly program. You've seen this before. PUTS is just a pseudonym for a TRAP that prints a string whose address is stored in R0, and HALT is a TRAP that stops the LC-3.
“.stringz” is another pseudo-op the stores the following string at that set of memory locations, followed by a zero (That's what the 'z' is for). For example, 'H' is stored at x3003, 'e' is stored at x3004, etc.
“.end” tells the assembler where to stop reading code for the current code block. Every .orig statement must have a corresponding .end statement.
Other pseudo-ops you should know are:
“.fill [value]” - put the given value at that memory location (.fill x6000, .fill 7, etc).
“.blkw [n]” - reserve the next n memory locations, filling them with 0.

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 “State” menu and click on “Randomize”. This randomly assigns values to every memory location and register, and prevents you from assuming values will be initialized to 0.
2. Go to the “File” menu, select “Load Over” and browse for your assembly file. This will load your code over the randomized memory. You will see your code load in the main window.
3. From here, you can run your code. Click the “Run” button to run your code automatically until a HALT instruction or breakpoint is hit. Click “Step” to execute one instruction at a time. Click “Next Line” to fast-forward through subroutines. Notice you can also step back.

Part 1 – Negation (negate.asm)

Write an assembly program that returns the negated value of the number stored at U in ANSWER.

Here's the pseudo-code:

```
/* u is the parameter, answer is the return value */
      answer = -u;
```

U is a label for a .fill in the template file. You will need to load the value at label U into a register. ANSWER is a label for another .fill in the template file. You will need to store the value you get for answer at the address labeled ANSWER.

U will be any integer in the range -32767 to 32767, inclusive.

Part 2 – Max (max.asm)

Write an assembly program that returns the max value in an array. Notice that this function returns the max value NOT the index of the max value.

Here's the pseudo-code:

```
int *array; /* This is a pointer! It holds the address of the array. More below. */
int max = array[0]; /* Dereferencing a pointer, see below */
for (int k = 0; k<LENGTH; k++) {
      if (array[k] > max) {
            max = array[k];
      }
}
```

You must store your answer in MAX.

Here, ARRAY and LENGTH are supplied as .fill'd values. The actual array will be defined in another .orig / .end block, as seen in the template file. You should change the array to test your code. Helpful hint: press ctrl+v to open up a new view, then ctrl+g to go to x6000. Now you can watch your array change as you debug your instructions!

ARRAY is any valid address and LENGTH is in the range 0 to 100, inclusive.

Part 3 – Square (square.asm)

Write a program that returns the value of U^2 in ANSWER.
Here's the pseudo-code:

```
/* u and v are parameters, answer is the return value */
int sum = 0;
if (u < 0) {
      u = -u;
}
int counter = u;
while (counter > 0) {
      sum += u;
      counter--;
}
answer = sum;
```

Again, U is supplied as a .fill'd value, and you must store your answer in ANSWER.
U is in the range -181 to 181, inclusive.

Part 4 – Selection Sort (select.asm)

Write a program that sorts an array location at address ARRAY of length LENGTH in ascending order.
Here's the pseudo-code:

```
int *array; /* This is a pointer! It holds the address of the array. More below. */
for (int i = 0; i < LENGTH - 1; i++) {
      int indexOfSmallest = i;
      for (int j = i+1; j < LENGTH; j++) {
            if (array[j] < array[indexOfSmallest]) { /* Dereferencing a pointer,
see below */
                  indexOfSmallest = j;
            }
      }
      int temp = array[i];
      array[i] = array[indexOfSmallest];
      array[indexOfSmallest]=temp;
}
```

Here, ARRAY and LENGTH are supplied as .fill'd values. The actual array will be defined in another .orig / .end block, as seen in the template file. You should change the array to test your code. Helpful hint: press ctrl+v to open up a new view, then ctrl+g to go to x6000. Now you can watch your array change as you debug your instructions!

ARRAY is any valid address and LENGTH is in the range 0 to 100, inclusive.

About Pointers

Pointers are a very valuable tool when programming in C. As you know, every value that we use in a program is stored at a certain address in memory. Pointers are just ways to deal with addresses of values instead of the values themselves.

In our problem, we declare an `int *array;`. The * character, when used in a declaration, signifies that the variable is a pointer to the declared type. This means that the variable holds an address, and at that address in memory we will find a value of the given type. In our case, `array` is a pointer to an `int`, meaning that `array` holds an address where an `int` is stored.

If you want to get the value stored at an address, called dereferencing, you use the * operator. (It may be confusing at first: * used in a declaration means you're declaring a pointer, and anywhere else means you're dereferencing a pointer, unless it's not a pointer – then it's multiplication). For example, if we want to get the value at the address stored in `array` and store it in `x`, we would write
`int x = *array;`.

In this problem, we actually have a list of values starting at the address stored in `array`. Another way to dereference a pointer in C is to use bracket notation. Bracket notation works like this:
`array[n] = *(array + n)`.
Yup, it looks exactly like an array in many other languages, and works exactly like an array would.

So, to get the first value stored at `array`, I would type `array[0]`. The next value in the array would be stored at address `array + 1`. Therefore, to get the second `int` stored at `array`, I would type `array[1]`, and so on. In assembly, to load `array[i]`, you would just load `array` into a register, add `i`, then dereference that value (i.e., get the value stored in memory at the address stored in that register).

If you want to know more about pointers, you can read about it in the textbook, or wait until we cover it in class soon.

Deliverables

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

negate.asm
square.asm
max.asm
select.asm