# C Programming Assignment

Due date: 23:59 2021-07-06
Total marks: 7

This assignment aims to give you some experience with C programming and to help you gain better understanding of the instruction format of LC-3 and how assembler works.

**Important Notes**:

- Markers will use instructions that are different from the examples given in the specifications when testing your programs.
- The files containing the examples can be downloaded from course web site and unpacked on a Unix machine with the command below:
    - `tar xvf examplefiles.tar.gz`

In this assignment, you are required to write C programs to implement some of the functions of the LC-3 assembler. That is, the C programs covert LC-3 assembly instructions to machine code that can be executed by the LC-3 simulator.

## Part 1 (3 marks)

LC3Edit is used to write LC-3 assembly programs. After a program is written, we use the LC-3 assembler (i.e. the “Translate → Assemble” function in LC3Edit) to convert the assembly program into a binary executable. The file stores the binary executable is called the *object file*. The object file generated by LC3Edit is named “file.obj” where “file” is the name of the assembly program (excluding the “.asm” suffix). In this specification, a “word” refers to a word in LC-3. That is, a word consists of two bytes. The structure of an object file is as below:

- The first word (i.e. the first two bytes) is the starting address of the program.
- The subsequent words correspond to the instructions in the assembly program and the contents of the memory locations reserved for the program using various LC-3 directives.
- In LC-3, data are stored in Big-endian format. For example, if byte 0x12 in word 0x1234 is stored at address 0x3000, byte 0x34 is stored at address 0x3001. This means, when you read a sequence of bytes from the object file of an LC-3 assembly program, the most significant bit of each word is read first.

In this part of the assignment, you are required to write a C program to generate an object file that can be used by the LC-3 simulator. The detailed requirements are as below:

1. A text file containing some numbers is given. This file is called *numbers file*. Each number in the file is a four-digit hexadecimal number. There is one number in each line of the file. You can regard each number as a word in LC-3.

2. Write a program to (a) read the numbers from the numbers file, (b) create an object file and write the numbers to the object file.

3. The object file is a binary file. It contains a sequence of words (i.e. the numbers from the numbers file). When the words are stored in the object file, they must be stored in Big-endian format. For example, when word 12ab is stored in the object file, 12 (i.e. the byte containing the most significant bit) is stored in the file before ab.

4. The words are stored in the object file one by one. There is no character separating two words in the object file.

   For example, if the contents of the numbers file are as below:

   ```
   1283
   5105
   c140
   2c04
   ```

   The contents of the resulting object file in hexadecimal format should be as follow:

   ```
   12835105c1402c04
   ```

   [Hint: Intel CPU uses little-endian format. So, you probably need to write a word to the object file byte by byte. The encryption/decryption example in the lecture shows how this can be done.]

5. Name this program as part1.c

6. **The names of the numbers file and object file must be provided to the program as command line arguments**. The program should be run using the following command format:

   ```
   ./program_name name_of_numbers_file name_of_object_file
   ```

Here is an example of the execution of the program. In this example, the name of the numbers file and the object file are t1.txt and t1.obj respectively (NOTE: “t1.txt” and “t1.obj” are the exact names of the files). **Markers might use files with different names when testing your program.** The contents of t1.txt are:

```
3020
2407
1283
```

```
5105
c140
2c04
1df7
506f
f025
000C
fff3
```

The execution of the program is shown below. The command line arguments are marked in red.

```
$./part1 t1.txt t1.obj
```

The execution of the program causes the creation of a binary file t1.obj. As t1.obj is a binary file, its contents are not "viewable" in the command window. Program printObjFile.c (included in examplefiles.tar.gz) can be used to print the contents of the object file. To make the contents easy to read, printObjFile.c displays each LC-3 word in one line. You can compile the printObjFile.c program with the command below:

```
$ gcc -o printObjFile printObjFile.c
```

To display the contents of object file t1.obj, use the command below (Note: the name of the object file needs to be given as a command line argument). The outputs are given under the command.

```
$ ./printObjFile t1.obj
3020
2407
1283
5105
c140
2c04
1df7
506f
f025
000c
fff3
```

## Part 2 (3 marks)

In this part of the assignment, you are required to write a C program to translate LC-3's AND, ADD and HALT assembly language instructions into machine code. The detailed requirements are as below:

1. The assembly instructions are stored in a file. This file is called the *instructions file*. Each line of the file stores exactly one instruction. The number of instructions in the file is **NOT** limited.
2. For this part, it should be assumed that the operands of the instructions only use the "register" addressing mode. That is, the values of all the operands are stored in registers.
3. It should be assumed that

   a. the file containing the assembly instructions starts with an ".orig" directive and ends with an ".end" directive
   b. the instructions are valid AND, ADD or HALT instructions
   c. there is exactly one space separating the opcode and the operands of the instruction
   d. the operands are separated by exactly one comma ","
   e. all the characters in the instruction are lower case letters
   f. there are no leading or trailing empty spaces in each line
   g. each line ends with the invisible "\n" character
4. The machine code should be stored in an object file. The structure of the object file should conform to the descriptions given in part 1. That is:
   a. The first word is the starting address of the program. This is followed by the words representing the machine instructions or data.
   b. The words are stored in big-endian format.
   c. The words are stored one by one with NO other value (e.g. a new line character) separating them.
5. Name this program as part2.c
6. **The names of the instructions file and object file must be provided to the program as command line arguments**. The program should be run using the following command format:

```
./program_name name_of_instructions_file name_of_object_file
```

Here is an example of the execution of the program. In this example, the names of the instructions file and the object file are t2.asm and t2.obj respectively (NOTE: "t2.asm" and "t2.obj" are the exact names of the files). Markers might use files with different names when testing your program. The contents of t2.asm are:

```
.orig x3020
add r1,r2,r3
and r0,r4,r5
halt
.end
```

The execution of the program is shown below. The command line arguments are marked in red.

```
$ ./part2 t2.asm t2.obj
```

As explained in Part 1, the contents of `t2.obj` can be viewed with the command below:

```
$ ./printObjFile t2.obj
3020
1283
5105
f025
```

## Part 3 (1 mark)

Expand your program in Part 2 to allow the use of “immediate” addressing mode for operands. That is, the value of an operand is stored in the instruction. It should be assumed that the value operand is given as a decimal number.
Name this program as part3.c

Here is an example of the execution of the program. In this example, the names of the instructions file and the object file are t3.asm and t3.obj respectively (NOTE: “t3.asm” and “t3.obj” are the exact names of the files). Markers might use files with different names when testing your program. The contents of t3.asm are:

```
.orig x3020
add r1,r2,r3
and r0,r4,r5
add r6,r7,#-9
and r0,r1,#15
halt
.end
```

The execution of the program is shown below. The command line arguments are marked in red.

```
$ ./part3 t3.asm t3.obj
```

As explained in Part 1, the contents of `t3.obj` can be viewed with the command below:

```
$ ./printObjFile t3.obj
3020
1283
5105
1df7
506f
f025
```

## Submission

1. Use command “tar cvzf asg.tar.gz part1.c part2.c part3.c” to pack the three C programs to file asg.tar.gz.
2. Submit asg.tar.gz.

## Debugging Tips

1. Debugging is a skill that you are expected to acquire. Once you start working, you are paid to write and debug programs. Nobody is going to help you with debugging. So, you should acquire the skill now. **You can only acquire it by practicing.**
2. If you get a "segmentation faults" while running a program, the best way to locate the statement that causes the bug is to insert "printf" into your program.
3. If you can see the output of the "printf" statement, it means the bug is caused by a statement that appears somewhere after the "printf" statement. In this case, you should move the "printf" statement forward. Repeat this process until you cannot see the output of the "printf" statement.
4. If you cannot see the output of the "printf" statement, it means the bug is caused by a statement that appears somewhere before the "printf" statement.
5. Combining step 3 and 4, you should be able to identify the statement that causes the "segmentation faults".
6. Once you identify the statement that causes the "segmentation faults", you can analyse the cause of bug, e.g. whether the variables have the expected values.

## Suggestions

- The C examples given in the lectures were carefully designed to help you do the assignment. It covers all the techniques that you need for this assignment. So, you should understand those examples before working on the assignment.
- It is unlikely that you know or remember all the details about C. A search on the Internet will normally give you the answer to your question. This is a very useful skill that will benefit you well beyond this course.
- You should create more test cases to test your program. The easiest way is to use LC-3 assembler and simulator.
  - If you use the LC3Edit program to create assembly programs on a PC, you need to be aware that each line ends with the invisible "\r\n" character sequence on a PC.
  - On a Unix machine, each line ends with an invisible "\n" character. The example assembly programs and the assembly programs used by the markers are created on a Unix machine.
- You can use a lot of C library functions. You might want to consider using the functions defined in "string.h" as it has a lot of functions for manipulating strings.

## Further Work

For those who seek challenge, you can think about how to implement a full-fledged LC-3 assembler.