COMPSCI 210 Assignment 3

Due date: 21:00 15th October 2018

Total marks: 100

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.

Important Notes:

- There are subtle differences between various C compilers. We will use the GNU compiler gcc on login.cs.auckland.ac.nz for marking. Therefore, you MUST ensure that your submissions compile and run on login.cs.auckland.ac.nz. Submissions that fail to compile or run on login.cs.auckland.ac.nz will attract NO marks.
- Markers will compile your program using command "gcc –o name name.c" where name.c is the name of the source code of your program, e.g. part1.c. That is, the markers will NOT use any compiler switches to supress the warning messages.
- 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 Canvas and unpacked on the server with the command below:
    - `tar xvf A3examplefiles.tar.gz`
- As we need to return the assignment marks before the exam of this course, there is NO possibility to extend the deadline for this assignment. NO late assignment will be accepted.

Academic Honesty

Do NOT copy other people's code (this includes the code that you find on the Internet).

We will use Stanford's MOSS tool to check all submissions. The tool is very "smart". Changing the names of the variables, shuffling the statements around and inserting dummy statements will not fool the tool. In previous years, quite a few students had been caught by the tool; and, they were dealt with according to the university's rules at https://www.auckland.ac.nz/en/about/learning-and-teaching/policies-guidelines-and-procedures/academic-integrity-info-for-students.html

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 (27 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 (refer to https://en.wikipedia.org/wiki/Endianness and https://www.webopedia.com/TERM/B/big_endian.html to learn more about 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 create 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 a 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 A3Examples.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 (36 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 (11 marks)

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
```

Part 4 (6 marks)

Expand your program in Part 3 to allow instruction JMP to be converted.
Name this program as part4.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 t4.asm and t4.obj respectively (NOTE: "t4.asm" and "t4.obj" are the exact names of the files). Markers might use files with different names when testing your program. The contents of t4.asm are:

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

```
$ ./part4 t4.asm t4.obj
```

As explained in Part 1, the contents of `t4.obj` can be viewed with the command below:

```
$ ./printObjFile t4.obj
3020
1283
5105
c140
1df7
506f
```

```
f025
```

Part 5 (9 marks)

Expand your program in Part 4 to allow the assembly program to contain directives for reserving memory locations and labels for marking memory locations.

1. Your program should generate a symbol table containing each label in the assembly program and the label's address.
2. The symbol table should be stored in a file named SymbolTable (**NOTE**: "SymbolTable" **is the exact name of the file. It does NOT have any suffix.**).
3. Each line in file SymbolTable contains exactly one label and its address.
4. A label and its address should be separated by exactly one empty space.
5. The order in which the labels appear in file SymbolTable is not important as long as all the labels are in the file.
6. Apart from creating file SymbolTable, your program does NOT need to generate an object file.
7. Name this program as part5.c

It should be assumed that:

- only ".fill" directive will be used for reserving memory locations
- a ".fill" directive is always preceded with a label
- each label consists of at most three characters
- there is exactly one space between a label and the ".fill" directive
- the ".fill" directives appear between the "HALT" instruction and the ".end" directive

Here is an example of the execution of the program. In this example, the name of the file containing the instructions is t5.asm (NOTE: "t5.asm" is the exact name of the file). Markers might use a file with a different name when testing your program. **In the file used by the markers, the value given in the ".orig" directive might be a value different from x3020.** The contents of t5.asm are:

```
.orig x3020
add r1,r2,r3
and r0,r4,r5
jmp r5
add r6,r7,#-9
and r0,r1,#15
halt
abc .fill #12
def .fill #-13
.end
```

The execution of the program is shown below. The command line arguments are marked in red. (Note: As no object file needs to be generated in this part, only the names of the program and the instructions file appear in the command line.)

```
$ ./part5 t5.asm
```

After the execution, file `SymbolTable` is created and the contents of `SymbolTable` are:

```
abc 3026
def 3027
```

Part 6 (9 marks)

Expand your program in Part 5 to allow instruction LD to be converted.

1. Your program should store the machine code of the converted instructions in an object file.
2. To simplify your task, your program should NOT show the contents of the reserved memory locations.
3. Name this program as part6.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 t6.asm and t6.obj respectively (NOTE: “t6.asm” and “t6.obj” are the exact names of the files). Markers might use files with different names when testing your program. **In the file used by the markers, the value given in the “.orig” directive might be a value different from x3020.** The contents of t6.asm are:

```
.orig x3020
ld r2,abc
add r1,r2,r3
and r0,r4,r5
jmp r5
ld r6,def
add r6,r7,#-9
and r0,r1,#15
halt
abc .fill #12
def .fill #-13
.end
```

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

```
$ ./part6 t6.asm t6.obj
```

As explained in Part 1, the contents of `t6.obj` can be viewed with the command below:

```
$ ./printObjFile t6.obj
3020
2407
1283
5105
c140
2c04
1df7
506f
f025
```

2 marks for no compile warning messages when your programs are compiled.

Submission

1. You **MUST** thoroughly test your program on login.cs.auckland.ac.nz before submission. Programs that cannot be compiled or run on login.cs.auckland.ac.nz will **NOT** get any mark.
2. Use command "tar cvzf A3.tar.gz part1.c part2.c part3.c part4.c part5.c part6.c" to pack the six C programs to file A3.tar.gz.
   a. You **MUST** use the tar command on login.cs.auckland.ac.nz to pack the files as files packed using tools on PC cannot be unpacked on login.cs.auckland.ac.nz. You will NOT get any mark if your file cannot be unpacked on login.cs.auckland.ac.nz.
   b. You should submit your source code (i.e. the text files with ".c" suffix).
   c. No marks will be given to people who only submit binary executables.
3. Submit A3.tar.gz using Canvas.
4. **NO** email submission will be accepted.

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 Google 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.