COMPSCI 210 Assignment 3

Due date: 21:00 29th May 2018
Total marks: 100

This assignment aims to give you some experience with C programming and to help you gain better understanding of the addressing mode 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 machine code that is different from the examples given in the specifications when testing your programs.
- The outputs of your programs will be checked by a program. Thus, your program's outputs MUST be in the EXACTLY SAME FORMAT as shown in the example given in each part. You must make sure that the registers appear in the same order as shown in the example and there are no extra lines.
- The files containing the examples can be downloaded from Canvas and unpacked on server with the command below:
    - `tar xvf A3Examples.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.

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 and shuffling the statements around 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 a LC-3 simulator. That is, the programs will execute the binary code generated by LC-3 assembler.

Part 1 (35 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 binary executable. The binary executable being 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 the "file.obj" 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 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 executable of an LC-3 assembly program from a file, 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 display each word in the ".obj" file of a program in hexadecimal form. That is, the C program should display each binary number stored in the ".obj" file in it corresponding hexadecimal form.

- Name the C program as "part1.c".
- **The name of the ".obj" file (the name of the file INCLUDES the ".obj" suffix) must be given as a command line argument.** The number of instructions in the file is **NOT** limited.
- In the output, each line shows the contents of one word.
- The value of each word must have a "0x" prefix.
- The letter digits "a" to "f" must be shown as lowercase letters.

Here is an example of the execution of the program. In this example, the LC-3 assembly program is as below. The name of the executable of the assembly program is "p1.obj" (markers will probably use a file with a different name and different contents).

```
.ORIG X4500
LD      R0, A
LEA     R1, B
LDI     R2, C
AND     R3, R0, R1
AND     R3, R1, #0
NOT     R4, R3
ADD     R4, R4, #1
BRp     F
ADD     R3, R3, #1
```

```
F       HALT
A       .FILL   X560A
B       .FILL   X4507
C       .FILL   X4501
.END
```

The execution of the program is shown below. In this example, the name of the file containing the machine instructions is p1.obj (NOTE: "p1.obj" is the exact name of the file. That is, the file name does NOT have a '.txt' suffix.). The command line argument is marked in red.

```
$ ./part1 p1.obj
0x4500
0x2009
0xe209
0xa409
0x5601
0x5660
0x98ff
0x1921
0x0201
0x16e1
0xf025
0x560a
0x4507
0x4501
```

Part 2 (46 marks)

In this part, you are required to write a C program to implement a LC-3 simulator that is capable of executing instruction "LD".

- Name the C program as "part2.c".
- **The name of the ".obj" file (the name of the file INCLUDES the ".obj" suffix) must be given as a command line argument.** The number of instructions in the file is **NOT** limited.
- The state of the simulator consists of the contents of the 8 general purpose registers (i.e. R0 to R7), the value of the program counter (i.e. PC), the contents of the instruction register (i.e. IR), and the value of the condition code (i.e. CC).
- The values in R0 to R7, PC and IR should be shown as hexadecimal value. The value of CC is either N, Z or P.
- Before the simulator starts executing a program, it should first display the initial state of the LC-3 machine. In the initial state, R0 to R7 and IR should all be 0; PC should be the starting address of the program to be executed; and CC should be set to Z.
- When displaying the value of R0 to R7, PC, IR and CC, a tab character (denoted as "\t" in C) is used to separate the name of the register and the value of the register.

- Each hexadecimal value must have a “0x” prefix. The letter digits “a” to “f” must be shown as lowercase letters.
- After showing the initial state, the simulator should execute each instruction except the “HALT” pseudo instruction in the “.obj” file. For this part, the simulator should display the hexadecimal code of each LD instruction that has been executed and the state of the LC-3 machine after each LD instruction is executed. The hexadecimal code of an instruction is the hexadecimal form of the 16 bits used to represent the instruction. The hexadecimal code of each LD instruction should be preceded with “after executing instruction\t” (where \t denotes a tab character) and “0x”.
- The simulator should output a line consisting of 18 “=” after displaying the state of the LC-3 machine.
- When the execution reaches the “HALT” instruction, the simulator terminates.

Here is an example of the execution of the program. In this example, the LC-3 assembly program is as below. The name of the executable of the assembly program is “p2.obj” (NOTE: “p2.obj” is the exact name of the file. That is, the file name does NOT have a ‘.txt’ suffix). Markers will probably use a file with a different name and different contents.

```
        .ORIG X4500
        LD      R0, A
    F   HALT
    A   .FILL   X560A
    B   .FILL   X4507
    C   .FILL   X4501
        .END
```

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

```
$ ./part2 p2.obj
Initial state
R0      0x0000
R1      0x0000
R2      0x0000
R3      0x0000
R4      0x0000
R5      0x0000
R6      0x0000
R7      0x0000
PC      0x4500
IR      0x0000
CC      Z
==================
after executing instruction     0x2001
R0      0x560a
R1      0x0000
R2      0x0000
R3      0x0000
R4      0x0000
R5      0x0000
R6      0x0000
R7      0x0000
PC      0x4501
IR      0x2001
CC      P
```

```
==================
```

Part 3 (3 marks)

This part is based on Part 2.

- Name this program as part3.c
- Expand the functionality of the simulator in part 2 to allow the simulator to execute instruction “LEA”.
- For this part, the simulator should display the hexadecimal code of each LEA instruction that has been executed and the state of the LC-3 machine after each LEA instruction is executed. The hexadecimal code of each LEA instruction should be preceded with “after executing instruction\t” (where \t denotes a tab character) and “0x”.
- The simulator should output a line consisting of 18 “=” after displaying the state of the LC-3 machine.

Here is an example of the execution of the program. In this example, the LC-3 assembly program is as below. The name of the executable of the assembly program is “p3.obj” (NOTE: “p3.obj” is the exact name of the file. That is, the file name does NOT have a ‘.txt’ suffix.). Markers will probably use a file with a different name and different contents.

```
        .ORIG X4500
        LD      R0, A
        LEA     R1, B
F       HALT
A       .FILL   X560A
B       .FILL   X4507
C       .FILL   X4501
        .END
```

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

```
$ ./part3 p3.obj
after executing instruction     0xe202
R0      0x560a
R1      0x4504
R2      0x0000
R3      0x0000
R4      0x0000
R5      0x0000
R6      0x0000
R7      0x0000
PC      0x4502
IR      0xe202
CC      P
==================
```

Part 4 (3 marks)

This part is based on Part 3.

- Name this program as part4.c
- Expand the functionality of the simulator in part 3 to allow the simulator to execute instruction “LDI”.
- For this part, the simulator should display the hexadecimal code of each LDI instruction that has been executed and the state of the LC-3 machine after each LDI instruction is executed. The hexadecimal code of each LDI instruction should be preceded with “after executing instruction\t” (where \t denotes a tab character) and “0x”.
- The simulator should output a line consisting of 18 “=” after displaying the state of the LC-3 machine.

Here is an example of the execution of the program. In this example, the LC-3 assembly program is as below. The name of the executable of the assembly program is “p4.obj” (NOTE: “p4.obj” is the exact name of the file. That is, the file name does NOT have a ‘.txt’ suffix.). Markers will probably use a file with a different name and different contents).

```
        .ORIG X4500
        LD      R0, A
        LEA     R1, B
        LDI     R2, C
F       HALT
A       .FILL   X560A
B       .FILL   X4507
C       .FILL   X4501
        .END
```

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

```
$ ./part4 p4.obj
after executing instruction     0xa403
R0      0x560a
R1      0x4505
R2      0xe203
R3      0x0000
R4      0x0000
R5      0x0000
R6      0x0000
R7      0x0000
PC      0x4503
IR      0xa403
CC      N
==================
```

Part 5 (3 marks)

This part is based on Part 4.

- Name this program as part5.c
- Expand the functionality of the simulator in part 4 to allow the simulator to execute instruction “AND”.

- For this part, the simulator should display the hexadecimal code of each AND instruction that has been executed and the state of the LC-3 machine after each AND instruction is executed. The hexadecimal code of each AND instruction should be preceded with "after executing instruction\t" (where \t denotes a tab character) and "0x".
- The simulator should output a line consisting of 18 "=" after displaying the state of the LC-3 machine.

Here is an example of the execution of the program. In this example, the LC-3 assembly program is as below. The name of the executable of the assembly program is "p5.obj" (NOTE: "p5.obj" is the exact name of the file. That is, the file name does NOT have a '.txt' suffix.). Markers will probably use a file with a different name and different contents).

```
    .ORIG X4500
    LD      R0, A
    LEA     R1, B
    LDI     R2, C
    AND     R3, R0, R1
    AND     R3, R1, #0
F   HALT
A   .FILL   X560A
B   .FILL   X4507
C   .FILL   X4501
    .END
```

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

```
$ ./part5 p5.obj
after executing instruction     0x5601
R0      0x560a
R1      0x4507
R2      0xe205
R3      0x4402
R4      0x0000
R5      0x0000
R6      0x0000
R7      0x0000
PC      0x4504
IR      0x5601
CC      P
==================
after executing instruction     0x5660
R0      0x560a
R1      0x4507
R2      0xe205
R3      0x0000
R4      0x0000
R5      0x0000
R6      0x0000
R7      0x0000
PC      0x4505
IR      0x5660
CC      Z
==================
```

Part 6 (3 marks)

This part is based on Part 5.

- Name this program as part6.c
- Expand the functionality of the simulator in part 5 to allow the simulator to execute instruction “NOT”.
- For this part, the simulator should display the hexadecimal code of each NOT instruction that has been executed and the state of the LC-3 machine after each NOT instruction is executed. The hexadecimal code of each NOT instruction should be preceded with “after executing instruction\t” (where \t denotes a tab character) and “0x”.
- The simulator should output a line consisting of 18 “=” after displaying the state of the LC-3 machine.

Here is an example of the execution of the program. In this example, the LC-3 assembly program is as below. The name of the executable of the assembly program is “p6.obj” (NOTE: “p6.obj” is the exact name of the file. That is, the file name does NOT have a ‘.txt’ suffix.). Markers will probably use a file with a different name and different contents).

```
        .ORIG X4500
        LD      R0, A
        LEA     R1, B
        LDI     R2, C
        AND     R3, R0, R1
        AND     R3, R1, #0
        NOT     R4, R3
F       HALT
A       .FILL   X560A
B       .FILL   X4507
C       .FILL   X4501
        .END
```

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

```
$ ./part6 p6.obj
after executing instruction     0x98ff
R0      0x560a
R1      0x4508
R2      0xe206
R3      0x0000
R4      0xffff
R5      0x0000
R6      0x0000
R7      0x0000
PC      0x4506
IR      0x98ff
CC      N
==================
```

Part 7 (3 marks)

This part is based on Part 6.

- Name this program as part7.c

- Expand the functionality of the simulator in part 7 to allow the simulator to execute instruction “ADD”.
- For this part, the simulator should display the hexadecimal code of each ADD instruction that has been executed and the state of the LC-3 machine after each ADD instruction is executed. The hexadecimal code of each ADD instruction should be preceded with “after executing instruction\t” (where \t denotes a tab character) and “0x”.
- The simulator should output a line consisting of 18 “=” after displaying the state of the LC-3 machine.

Here is an example of the execution of the program. In this example, the LC-3 assembly program is as below. The name of the executable of the assembly program is “p7.obj” (NOTE: “p7.obj” is the exact name of the file. That is, the file name does NOT have a ‘.txt’ suffix.). Markers will probably use a file with a different name and different contents).

```
        .ORIG X4500
        LD      R0, A
        LEA     R1, B
        LDI     R2, C
        AND     R3, R0, R1
        AND     R3, R1, #0
        NOT     R4, R3
        ADD     R4, R4, #1
F       HALT
A       .FILL   X560A
B       .FILL   X4507
C       .FILL   X4501
        .END
```

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

```
$ ./part7 p7.obj
after executing instruction     0x1921
R0      0x560a
R1      0x4509
R2      0xe207
R3      0x0000
R4      0x0000
R5      0x0000
R6      0x0000
R7      0x0000
PC      0x4507
IR      0x1921
CC      Z
==================
```

Part 8 (3 marks)

This part is based on Part 7.

- Name this program as part8.c
- Expand the functionality of the simulator in part 7 to allow the simulator to execute instruction “BR”.

- For this part, the simulator should display the hexadecimal code of each BR instruction that has been executed and the state of the LC-3 machine after each BR instruction is executed. The hexadecimal code of each BR instruction should be preceded with "after executing instruction\t" (where \t denotes a tab character) and "0x".
- The simulator should output a line consisting of 18 "=" after displaying the state of the LC-3 machine.

Here is an example of the execution of the program. In this example, the LC-3 assembly program is as below. The name of the executable of the assembly program is "p8a.obj" (NOTE: "p8a.obj" is the exact name of the file. That is, the file name does NOT have a '.txt' suffix.). Markers will probably use a file with a different name and different contents).

```
        .ORIG X4500
        LD      R0, A
        LEA     R1, B
        LDI     R2, C
        AND     R3, R0, R1
        AND     R3, R1, #0
        NOT     R4, R3
        ADD     R4, R4, #1
        BRp     F
        ADD     R3, R3, #1
F       HALT
A       .FILL   X560A
B       .FILL   X4507
C       .FILL   X4501
        .END
```

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

```
$ ./part8 p8a.obj
after executing instruction     0x0201
R0      0x560a
R1      0x450b
R2      0xe209
R3      0x0000
R4      0x0000
R5      0x0000
R6      0x0000
R7      0x0000
PC      0x4508
IR      0x0201
CC      Z
==================
```

Here is another example of the execution of the program. In this example, the LC-3 assembly program is as below. The name of the executable of the assembly program is "p8b.obj" (NOTE: "p8b.obj" is the exact name of the file. That is, the file name does NOT have a '.txt' suffix.). Markers will probably use a file with a different name and different contents).

```
        .ORIG X4500
        LD      R0, A
        LEA     R1, B
```

```
        LDI     R2, C
        AND     R3, R0, R1
        AND     R3, R1, #0
        NOT     R4, R3
        ADD     R4, R4, #1
        BRzp    F
        ADD     R3, R3, #1
F       HALT
A       .FILL   X560A
B       .FILL   X4507
C       .FILL   X4501
        .END
```

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

```
$ ./part8 p8b.obj
after executing instruction     0x0601
R0      0x560a
R1      0x450b
R2      0xe209
R3      0x0000
R4      0x0000
R5      0x0000
R6      0x0000
R7      0x0000
PC      0x4509
IR      0x0601
CC      Z
==================
```

Part 9 (1 mark)

1 mark for having no compile warning messages for all the submitted programs.

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 part7.c part8.c" to pack the eight C programs to file A3.tar.gz. [Note: 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.]
3. Submit A3.tar.gz through Canvas. The markers will only mark your latest submission.
4. **NO** email submission will be accepted.

Resource

- The binary files used in the examples of the specifications are packed in file A3Examples.tar.gz.
- The scripts used by the markers to check your programs' outputs are also packed in A3Examples.tar.gz. To ensure that the outputs of your programs conform to the required output format, I STRONGLY suggest you use the scripts to check the outputs of your programs when you use the examples in the specification to test your programs. If the outputs and the format of the outputs are correct, you should see a "part X passes the test" message. WHEN YOUR PROGRAMS ARE MARKED, OUTPUTS THAT DO NOT CONFORM TO THE REQUIRED FORMAT WILL FAIL THE TEST AND WILL GET 0 FOR THE FAILED CASES.
- Follow the steps below to extract and to use the files in A3Examples.tar.gz
    - Put A3Examples.tar.gz in the directory in which your programs are stored.
    - Use command "tar xvf A3Examples.tar.gz" to unpack A3Examples.tar.gz. After unpacking the file, the ".obj" files and the marking scripts should be extracted to the same directory as your programs.
    - Once you are satisfied with the outputs of your program, you can run the marking scripts to check the outputs and the formats of the outputs of your programs using the marking scripts. There is one marking script for each part. The marking script for part 1 is "m1.bash"; the script for part 2 is "m2.bash"; and so on.
    - To run a script, use command "./mX.bash" where X is a digit. For example, to run the marking script for part 1, use command "./m1.bash".
    - The marking scripts only check the outputs of the programs for the examples given in the specification.
- You should create more test cases to manually test your programs. The easiest way is to use LC-3 assembler to generate the binary ".obj" file and upload the file to login.cs.auckland.ac.nz for testing. You can use the official LC-3 simulator to check whether the outputs of your programs are correct.

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.