COMPSCI 210 Assignment 2

Due date: 21:00 3rd June 2019
Total marks: 80

This assignment aims to give you some experience with C programming and to help you gain better understanding of the ISA 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 A2Examples.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 emulator. That is, the programs will execute the binary code generated by LC-3 assembler.

Part 1 (40 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 (26 marks)

In this part, you are required to write a C program to implement a LC-3 emulator 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.
- It should be assumed that the “.obj” file contains at most 100 LC-3 machine instructions.
- The state of the emulator 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 emulator 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 emulator should execute each instruction except the “HALT” pseudo instruction in the “.obj” file. For this part, the emulator 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 emulator should output a line consisting of 18 “=” after displaying the state of the LC-3 machine.
- When the execution reaches the “HALT” instruction, the emulator 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 (2 marks)

This part is based on Part 2.

- Name this program as part3.c
- Expand the functionality of the emulator in part 2 to allow the emulator to execute instruction “LEA”.
- For this part, the emulator 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 emulator 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 (2 marks)

This part is based on Part 3.

- Name this program as part4.c
- Expand the functionality of the emulator in part 3 to allow the emulator to execute instruction “LDI”.
- For this part, the emulator 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 emulator 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 (4 marks)

This part is based on Part 4.

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

- For this part, the emulator 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 emulator 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 (2 marks)

This part is based on Part 5.

- Name this program as part6.c
- Expand the functionality of the emulator in part 5 to allow the emulator to execute instruction “NOT”.
- For this part, the emulator 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 emulator 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 (2 marks)

This part is based on Part 6.

- Name this program as part7.c

- Expand the functionality of the emulator in part 7 to allow the emulator to execute instruction “ADD”.
- For this part, the emulator 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 emulator 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 (2 marks)

This part is based on Part 7.

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

- For this part, the emulator 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 emulator 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
==================
```

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 A2.tar.gz part1.c part2.c part3.c part4.c part5.c part6.c part7.c part8.c" to pack the **SOURCE** code of your completed C programs to file A2.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 A2.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 A2Examples.tar.gz.

- The scripts used by the markers to check your programs' outputs are also packed in A2Examples.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 A2Examples.tar.gz
    - Put A2Examples.tar.gz in the directory in which your programs are stored.
    - Use command "tar xvf A2Examples.tar.gz" to unpack A2Examples.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 emulator 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.

Some Hints

You are strongly encouraged to attempt each part first without reading these hints.
There are many ways to write the programs that satisfy the specifications. The hints given here are just some of the possible implementations. You can implement your programs differently as long as the outputs satisfy the specifications.
These are hints rather than a detailed step by step descriptions on how to implement the assignment. So, you still need to bridge the gaps that are not explicitly stated.

Part 1

1. The ".obj" file is a binary file. To display the contents of the file in hexadecimal, we need to read the information in the ".obj" file byte by byte. Read example program xor.c to see how to do this.
2. The words in the ".obj" file are stored in big-endian format. So, the most significant byte of a word is stored first in the file. For example, if three words, 0x4500, 0x2009 and 0xe209, are stored in the file, the sequence of bytes in the file is "45002009e209". Therefore, we just need to read the bytes from the file one by one, and print out the bytes one by one.
3. Each byte consists of two hexadecimal digits. To print out the contents of a byte as two hexadecimal digits, you can use "printf("%02x", byte)" where "byte" is a variable of "char" type (the size of a "char" type variable is one byte).
4. Example program xor.c shows how to read bytes from a binary file.
5. The pseudo code of a possible implementation is as below:
   do {
       read two bytes from the ".obj" file
       print "0x"
       print the first byte
       print the second byte
       print a new line character, i.e. "\n"
   } while (not the end of the file)
6. The name of the ".obj" file (the name of the file INCLUDES the ".obj" suffix) must be given as a command line argument. Example program xor.c shows how to process a file whose name is given as a command line argument.

Part 2

1. The state of the emulator can be represented by a set of variables.
2. In LC-3, the size of a word is 16, i.e. two bytes. Variables of "unsigned short" type in C consists of two bytes. You can use this type of variables to hold the value of the 8 general purpose registers (i.e. R0 to R7), the value of the program counter (i.e. PC), and the contents of the instruction register (i.e. IR).

   a. You can use an array of type “unsigned short” with 8 elements to hold the values of the 8 general purpose registers of LC-3. The first element holds the value of register 0; the second element holds the value of register 1; and so on.
   b. The condition code only records a single character. So, you can use a variable of type “char” to hold the value of a condition code register.
3. The first two bytes in the “.obj” file is the starting address of the program. Each of the subsequent two-byte blocks is an instruction or pre-set value of the LC-3 program.
4. To obtain the starting address of the program, we need to read the first two bytes from the “.obj” file. Let x be an “unsigned short” type variable for holding the value of the starting address. The first byte read from the “.obj” file should be stored in bits 8 to 15 of x while the second byte read from the “.obj” file should be stored in bits 0 to 7 of x. This is because the data in the “.obj” file are stored in the big-endian format. See slides 85 to 98 on how to use bitwise operations to set/extract values of variables.
5. Initially, the values of all the general purpose registers are 0; the instruction register is also set to 0 (as no instruction has been fetched into the IR); the condition code is “Z”; and the PC is the starting address of the program.
6. You can use an array to simulate the memory storing the machine code in the “.obj” file. Let’s call this array as memory array. Each element of the array is of type “unsigned short” as each word consists of 2 bytes in LC-3.
   a. The first instruction can be stored in the first element of the array; the second instruction can be stored in the second element of the array; and so on.
      i. You can populate the memory array by reading the contents of the “.obj” file into the array. [Note: Like the LC-3 simulator that you have used when writing LC-3 assembly program, you do not need to store the starting address, i.e. the first word in the “.obj” file, in the memory.]
   b. Later, when the emulator executes the program, you can obtain the instruction or the value stored in the memory from the array.
   c. Given a memory address, you can easily work out the index of the array element that holds the value/instruction stored at that memory address. This is because you know the starting address of the program and we have stored the first instruction of the program in the first element of the memory array.
7. The value of the instruction register (IR) can be stored in a variable of type “unsigned short”. To “execute” an instruction, you need to fetch the instruction from the memory array into the variable for IR. The address of the instruction is given in PC.
8. To determine the type of instruction in the IR, we need to check the opcode of the instruction. That is, we need to check the value in bits 12 to 15 of the IR. You can use the bitwise operations (see slides 85 to 98) to find out the value of the opcode of the instruction.
9. Use a loop to load each instruction into the IR. After loading an instruction into the IR, the value of PC should be incremented. Then, the instruction is processed (i.e. “executed”).

a. In order to implement the execution of an instruction, you need to understand the steps that the CPU takes when it executes an instruction. So, it is a good time to review what you have learned in the first part of this course.
b. First, the CPU checks the opcode of the instruction. In part 2, the only instruction that you need to handle is the "LD" instruction. So, you just need to check whether the opcode of the instruction is 0x2. If not, we have reached the end of all the instructions in the program. Therefore, we can terminate the loop.
c. If the opcode is 0x2, the instruction is a "LD" instruction. Once, the CPU knows that the instruction is a "LD" instruction, the CPU carries out the operations according to the specifications of "LD".
    i. The "LD" instruction load the contents of a memory location to a register. The address of the memory location is obtained by PC+offset.
    ii. The value of the offset is stored in bits 0 to 8 of the instruction (i.e. bits 0 to 8 of IR). You can retrieve the offset value using the bitwise operations (see slides 85 to 98).
    iii. Once you work out the address of the memory location, you can figure out the index of the memory array's element that corresponds to the memory location. Therefore, you can retrieve the value to be loaded to a register from that array element.
    iv. The ID of the register that receives the value is stored in bits 9 to 11 of the instruction (i.e. IR). You can use the bitwise operations (see slides 85 to 98) to retrieve the ID.
    v. The ID number is used as the index to find the element in the register array that corresponds to the register in the "LD" instruction.
    vi. You can store the value retrieved from the memory to the register.
    vii. According to the value loaded into the register in the "LD" instruction, you can set the value in the condition code.

Parts 3 to 8 are simple extensions to part 2. To summarise:

1. Extract the opcode value from bits 12 to 15 of IR using the bitwise operations (see slides 85 to 98).
2. From the opcode value, you can figure out the type of the instruction, e.g. LEA, LDI, etc.
3. Once you know the type of the instruction, the format of the instruction is known. That is, the bits allocated for various operands in the instruction.
4. According to the meaning of the instruction, set the value of the register affected by the execution of the instruction.
    a. For example, for instruction "LEA R1, B",
        i. retrieve the value of the offset from the instruction
        ii. compute the address to be loaded into R1 using formula PC+offset
        iii. set the value of the array element holding the value of R1 to the computed address

iv. set the value of the condition code according to the value in R1