ECE 150: Fundamentals of Programming (Fall 2016)

Section 001: Project 1

Due on Monday, November 7, 11:59pm ET

The course project is divided over three parts, leading to a simple computer simulator. The overall project will therefore give you better insight into the operation of a computer, as well as teaching you the basics of discrete-event simulation. In this first part of the part of the project you are required to open and read a file containing assembly-language instructions, parse those instructions, and produce some statistical data about the instructions.

Learning Outcomes

After completing this part of the project, students should be able to:

1. Open and read data from files, using C++, with appropriate error checking
2. Parse simplified assembly-language instructions
3. Compute basic statistical data about a file of simplified assembly
4. Develop basic test cases for your code

Requirements

Submission: You should submit the result of this exercise to the project `150-Project-Parser`. Your submission file must be named `parser.cpp` (case sensitive).

Note: We will not be opening the Marmoset submission for any of the project submission requirements until one week before the submission is due. In the case of this part of the project, this means the submission option will not be made available on Marmoset until October 31st. This is intended to encourage students to develop their own test cases and to do coding and testing on their local machine, getting the basic functionality correct, before submitting it to Marmoset for formal testing.

Task Description

Write a program that accepts a single argument, which is the name of a file. The file should contain a series of assembly-language instructions that form a short assembly-language computer program. Your program should open and read this file, one line at a time. It must parse the data in the program using a state-machine approach, as follows. Each line of the file is one of the following:

- A complete assembly instruction and all necessary data, and possibly a comment
- A blank line (possibly including whitespace, but no actual code
- A comment line (to be discussed below)
- A label or directive (to be discussed below)

A complete assembly instruction consists of the instruction (*e.g.*, `LD`, `MUL`, *etc*.) and its associated data (*e.g.*, in the case of a `JMP` instruction, there would be one additional parameter which is the instruction location to which the assembly program is supposed to jump. In assembly-language programming, the instruction is typically referred to as the OPCODE, while its associated data is referred to as the OPERANDS, and we will do so for the remainder of this project description. The OPCODE will always be separated from its OPERANDS with whitespace (either blanks or tabs).

You should treat the OPCODE and OPERANDS as the "events" on which we wish to switch the state machine, and you should design that state machine on paper before attempting to code it. Thus, there will be an initial outer switch statement on the instruction, which can match the various possible OPCODES (see end of this document for the OPCODES), then there will be a secondary set of choices according to the OPERANDS for the particular OPCODE.

OPERANDS will be separate from each other by a comma (“,”) and possibly by additional whitespace. Thus, the assembly instruction:

```
MULi R1, 3, R2
```

should be accepted as identical to

```
MULi R1,3,R2
```

or even

```
MULi            R1,        3,         R2
```

What would not be accepted (and you should issue an error) would be trailing commas, incorrect or missing OPERANDS, *etc*. For example,

```
MULi R1, 3
```

and

```
MULi R1, 3, R2, R4
```

and

```
MULi R1, 3, PC
```

should all be rejected with an error message. Your error message should contain the word “`Error`”, an associated line number, and a brief description of the error. In the above three cases, something like the following would be appropriate:

```
Error on line 1: Missing third OPERAND
Error on line 2: Extra data after instruction
Error on line 3: Expected Register for third OPERAND; saw “PC”
```

You should keep track of the total number of assembly instructions and their category. For the purpose of categories, there are three broad types: Load/Store instructions, Arithmetic-and-Logic-Unit instructions, and Compare-and-Jump instructions. After you have parsed the entire file, you should display on `cout` the totals for each of these categories. Thus, the last thing your program should output is:

```
Total number of assembly instructions: 23
Number of Load/Store: 13
Number of ALU: 7
Number of Compare/Jump: 3
```

Note that sum of the categorized instructions should equal the total.

Blank lines should be ignored, but should not cause errors in your parsing. You may wish to count the number of blank lines, but it is not required.

Comments are invaluable in understanding programs. We will therefore augment our assembly language by using “#” to identify a comment. Anything on a line after a “#” is a comment and should be ignored. For example,

```
# This is my very short assembly program
     MULi R1, 3, R2           # y = x by 3
     MUL  R2, R3, R4          # z = y * w
     SD   R4, R5              # Save z in memory
```

is equivalent to:

```
     MULi R1, 3, R2
     MUL  R2, R3, R4
     SD   R4, R5
```

Labels and Directives

There is a problem we have yet to deal with: where will the code and data actually go? At this stage of the project you are only creating the parser, but at some point we will need to specify where the code and data will go, and it will be necessary for someone writing assembly code to know some of this so as to be able to write instructions. For example, how can the assembly programmer know where "x" will be located, so as to write

```
LD ????, R1
```

What value should the programmer use for "`????`" ? The same problem exists for the various compare-and-jump instructions. To solve this problem, we introduce labels and directives. There are two special directives, as follows:

```
Code: <nnnn>
```

and

```
Data: <nnnn>
```

where `<nnnn>` is some memory address, without the angle brackets. We may now have a short program as follows:

```
# This program computes z = x + y in an infinite loop
Code: 1000
     LD   1500, R1            # Load x
     LD   1501, R2            # Load y
     ADD  R1, R2, R3          # Add x and y
     SD   R3, 1502            # Store z
     JMP  1000                # and do it again, forever

Data: 1500                    # Data starts at 1500
                              # 1500: x
                              # 1501: y
                              # 1502: z
```

Note that although the "`Code:`" and "`Data:`" directives have been placed above their respective sections in the program, there is no requirement to do so. The "`Code:`" directive will occur before the first assembly instruction line, while the "`Data:`" directive is both optional, and if it does occur, it may occur at any location in the assembly program. In particular, some assembly programmers may prefer to place them at the beginning of the file.

In principle, a jump or a compare-and-jump instruction can be implemented by having the assembly-language programmer work out the exact location for a target address (as is the case above). However, to facilitate that, we will add labels, which are simple ASCII strings followed by a colon, with nothing else, except whitespace or comments, on the line. To identify their use within an assembly instruction, whenever they are used, they will be enclosed with square brackets. Therefore, we may now have the above program as follows:

```
# This program computes z = x + y in an infinite loop
Code: 1000
Data: 1500 (x, y, and z in sequence)

StartOfLoop:
     LD   1500, R1            # Load x
     LD   1501, R2            # Load y
     ADD  R1, R2, R3          # Add x and y
     SD   R3, 1502            # Store z
     JMP  [StartOfLoop]       # and do it again, forever
```

Note that the label does not contain an address number. This is because each assembly language instruction is on its own line, with the first one starting at the line identified by the "`Code:`" directive. In the above case, therefore, since "`StartOfLoop:`" is just before the very first instruction, it has the value of `1000` when used within the `JMP` instruction. It is therefore very important that you are able to identify the line number of each assembly instruction.

You should output the values of the code and data directives, together with the names and values of any labels.

Thus, the basic structure of your program is as follows:

```
Make sure program has 1 argument
Treat argument as a filename and open it
while (not read all lines in file) {
     read next line
     separate line into OPCODE and OPERANDS
     switch(OPCODE) {
          cases for various opcodes, organized by OPERAND types
     }
}
```

Assembly instructions to be parsed are contained on the final page of this description.

RISC Machine-Level (Assembly) Instructions

PC: Program Counter
Ri: Register i for any particular i
nnnn, mmmm: Numerical value

Note: when a number is contained within angle brackets (<...>), it means that number is a memory location. The assembly code itself will not contain those angle brackets.

LD <nnnn>, Ri	Ri <-- data at memory location <nnnn>
LD Ri, Rj	Rj <-- data at memory location contained in Rj
LDi nnnn, Ri	Ri <-- nnnn
SD Ri, <nnnn>	Store data in Ri to memory location <nnnn>
SD Ri, Rj	Store data in Ri to memory location contained in Rj
SDi mmmm, <nnnn>	Store the number mmmm to memory location <nnnn>
SDi mmmm, Ri	Store the number mmmm to memory location contained in Ri

ADD Ri, Rj, Rk	Rk <-- Ri + Rj
ADDi Ri, nnnn, Rj	Rj <-- Ri + nnnn
SUB Ri, Rj, Rk	Rk <-- Ri - Rj
SUBi Ri, nnnn, Rj	Rj <-- Ri - nnnn
MUL Ri, Rj, Rk	Rk <-- Ri * Rj
MULi Ri, nnnn, Rj	Rj <-- Ri * nnnn
DIV Ri, Rj, Rk	Rk <-- Ri / Rj
DIVi Ri, nnnn, Rj	Rj <-- Ri / nnnn

JMP <nnnn>	PC <-- nnnn
JZ Ri, <nnnn>	PC <-- nnnn if Ri == 0
JNZ Ri, <nnnn>	PC <-- nnnn if Ri != 0
JGZ Ri, <nnnn>	PC <-- nnnn if Ri > 0
JGEZ Ri, <nnnn>	PC <-- nnnn if Ri >= 0
JLZ Ri, <nnnn>	PC <-- nnnn if Ri < 0
JLEZ Ri, <nnnn>	PC <-- nnnn if Ri <= 0