Lab report - Assembler

PB20111686 Ruixuan Huang

Lab Goal

Read the given code, understand the framework of the program, and replace all `TO BE DONE` in the code with the correct code. And, learn to use makefile.

Overview

The goal of the program is to realize a small assembler, whose function is transforming assembly language code into binary code. Let's consider the difference between both. In assembly language, apart from binary code, we can use pseudo instructions, comment and most importantly, the labels to replace PCoffset. However, to achieve our goal, we must consider how to transform these characteristics back into binary.

The idea of the framework is to scan three times, because instructions such as `ADD` can be directly translated into binary code, but `BR`, `.FILL` and other instructions like these need to calculate the PCoffset. However, it is impossible to know the offset simply according to the label of the current line. Therefore, preprocessing is required, that is, the first two scans.

In the following paragraphs, I will first talk about the idea of code completion from simple to complex, and then describe the learning process of compiling with makefile.

Completion

Conversion Function

First, let's talk about several auxiliary functions independent of the core.

1. The following function is to convert a string represented number, whose base is decimal or hexadecimal, into a integer type. What deserves consideration is the form of negative numbers.

```
int RecognizeNumberValue(std::string s) {
    // Convert string s into a number
    std::string tmp;
    int size = s.size();
    if (s[0] == 'x' || s[0] == 'X') {
        if (s[1] == '-') {
            tmp = s.substr(2);
        }
        else {
            tmp = s.substr(1);
        }
        return ((s[1] == '-') ? (-1) : 1) * SwitchHexStr2Integer(tmp);
    }
    else if (s[0] == '#') {
        return atoi(s.substr(1).c_str());
    }
    else if (s[0] == '-') {
        return -1 * RecognizeNumberValue(s.substr(1));
    }
    return -1;
}
```

2. The following function is to convert a interger to 16-bit binary string for other functions' using.

```
std::string NumberToAssemble(const int &number) {
    // Convert the number into a 16 bit binary string
    std::string ans = "0000000000000000";
    int std = 32768;
    for (int i = kLC3LineLength; i > 0; i--, std >>= 1) {
        if ((number & std) >> (i - 1)) {
            ans[kLC3LineLength - i] = '1';
        }
    }
    return ans;
}
```

3. The following function is a kind of standardize. Since there may be strange tabs or other invisible characters in the user's input. But the core part of the program overloads the left shifting operator, I suspect the neccessity of this function. However, completing this function isn't completely useless. Through it I learned how to use the `find_first_not_of` function :) .

```
inline std::string &LeftTrim(std::string &s, const char *t = " \t\n\r\f\v") {
    auto begin = s.find_first_not_of(t);
    if (begin == s.npos) {
        s.clear();
    }
    else {
        auto size = s.size();
        s = s.substr(begin, size - begin);
    }
    return s;
}
```

Scanning, or the Mainly Used Function

The purpose of **first scanning** is to read the file and separate the file and comment, which are stored in two `std::vector<std::string>` container named `file_content` and `file_comment`. To be safe, we should convert all operation commands from lowercase to uppercase.

The **second scanning** is mainly used to process pseudo instructions, mark the type of instructions for each line, and record the address corresponding to each label. We should traverse `file_content`, use `line_address` to record the offset of the current statement from the address of the starting statement and use `file_address` to record the offset of the PC relative to the starting address when the `line_index` th instruction is executed. `label_mp` records the relative address corresponding to the label to facilitate the subsequent jump statement to calculate the offset. For each line, if it is a comment, it will not be processed. If it is a pseudo instruction, it needs to be completed in some degree.

In the **third scanning**, the translation results are output to the file. For pseudo instructions like `.STRINGZ` or `.BLKW`, the output behavior is filled with data. For instructions typed `loperation`, we only need to output them by referring to the ISA. It is roughly divided into two steps:

1. `result_line.append(opcode + nonvarible bits)`
2. Judge whether the following parameters are legal. If they are legal, convert the parameters to binary code through the `translateOprand()` function and append them to the `result_line`.

Here I won't paste my code since it's too long and so many repetitions in it. You can see my full code in the attachment.

Using Makefile

This lab also requires me to learn to compile my project using makefile. In fact, in the windows IDE (such as MSVC), we don't have to use tools like makefile to compile our program, but it may be used in other occasions, so we'd better learn it here.

```
CC=g++
CFLAGS=-I. -g
DEPS = assembler.h
OBJ = assembler.o main.o
```

```
%.o: %.cpp $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

assembler: $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS)

all: assembler

.PHONY: clean

clean:
    rm -rf assembler
    rm *.o
```

For some reasons I can't use `make` in the cmd directly, I have to use `mingw32-make` as alternative. The makefile script of the program has already been completed, so we may just execute the command in cmd.

```
cd file_folder
mingw-32_make
```

Then a executable file will be generated.