ICS Lab Report - labA

SCGY Cao Gaoxiang PB20000061

January 3, 2022

Lab Name

Assembler Lab

Lab Purpose

Implementing a basic LC-3 assembler.

Lab Content

You will get the following C++ project, containing the basic framework of the program. You need to learn how to use Makefile and finish the code.
Here are tasks.

1. Learn how to use Makefile.
2. Read the code, understand the framework of the program, and train the ability to read the program.
3. Replace all **TO BE DONE** in the code with the correct code.

Lab Environment

Operating system: Windows 11 Home Edition build 21H2 (with WSL 2), macOS Monterey.
Software: Visual Studio Code, LC3Tools.

Lab Procedure

1. In fact, for a project of only 4 files, we can perform the operation of Makefile with one command.

   ```
   g++ assembler.cpp main.cpp -o assembler
   clang++ assembler.cpp main.cpp -o assembler -std=c++14
   cl assembler.cpp main.cpp /EHsc
   ```

   Any of them can work. But we still tried to use Makefile.

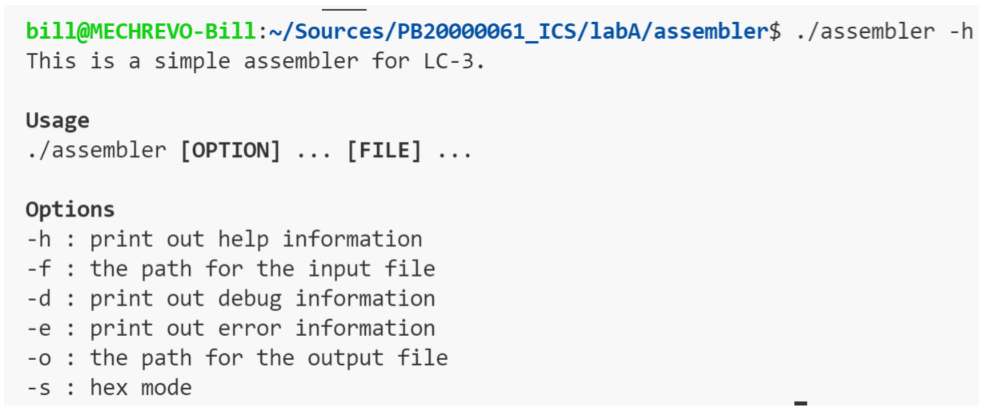
   (a) For Windows, there are two ways to use Makefile. The one is using MinGW, with the following commands.

   ```
   mingw32-make
   ```

   The other is to use WSL, which is the same as the method of Linux.

   (b) For Linux/UNIX, we just need to enter the directory and then type "make". Only note that on macOS, since the default C++ standard of Apple Clang is C++98, we will need to append CFLAGS with "-std=c++14" to build the project.

2. First, we tried to build the project. After handling compile errors, by typing "./assembler -h", we can know the basic usage of the program.



```
bill@MECHREVO-Bill:~/Sources/PB20000061_ICS/labA/assembler$ ./assembler -h
This is a simple assembler for LC-3.

Usage
./assembler [OPTION] ... [FILE] ...

Options
-h : print out help information
-f : the path for the input file
-d : print out debug information
-e : print out error information
-o : the path for the output file
-s : hex mode
```

We don't need to do anything to main.cpp. In assembler.hpp, we need to complete the function Trim. This function is used to eliminate the irrelated characters beside Assembly commands. Using std::string::find(), it's simple to finish the job.

```
inline std::string& RightTrim(std::string& s,
    const char* t = "␣\t\n\r\f\v") {
    if (s.empty()) return s;
    int pos = s.length() - 1;
    std::string tmp(t);
    while (tmp.find(s[pos]) != tmp.npos) --pos;
    s = s.substr(0, pos + 1);
    return s;
}
```

While finishing the code, we need to search for usage of the functions to get the proper error return code for them.

Our main work is in assembler.cpp, which can be devide into several cases.

- Convertional functions, which is used to convert data from one format to another.

```
int RecognizeNumberValue(std::string s) {
    // Convert string s into a number
    int ans = 0, len = s.length();
    if (tolower(s[0]) == 'x') {
        for (int i = len - 1; i >= 1; --i) {
            if (tolower(s[i]) >= 'a' && tolower(s[i]) <= 'f')
                ans |= (tolower(s[i]) - 'a' + 10)
                    << (4 * (len - 1 - i));
            else if (s[i] >= '0' && s[i] <= '9')
                ans |= (s[i] - '0') << (4 * (len - 1 - i));
            else return std::numeric_limits<int>::max();
        }
    }
    else if (s[0] == '#' || s[0] == '-'
        || (s[0] >= '0' && s[0] <= '9')) {
        bool neg = false;
        int start = 0;
        if (s[0] == '#') start = 1;
        if (s[start] == '-') { neg = true; start = 2; }
        for (int i = start; i < len; ++i) {
            if (s[i] >= '0' && s[i] <= '9')
                ans = ans * 10 + s[i] - '0';
            else return std::numeric_limits<int>::max();
        }
        if (neg) ans = -ans;
    }
    else return std::numeric_limits<int>::max();
    return ans;
}

std::string NumberToAssemble(const int& number) {
    // Convert the number into a 16 bit binary string
    int16_t num = number & 0xffff;
    std::string ans;
    for (int i = 15; i >= 0; --i)
        ans += (bool(num & (1 << i)) + '0');
    return ans;
}

std::string NumberToAssemble(const std::string& number) {
    // Convert the number into a 16 bit binary string
    return NumberToAssemble(RecognizeNumberValue(number));
}

std::string ConvertBin2Hex(std::string bin) {
    // Convert the binary string into a hex string
    std::string ans;
    for (int i = 0; i <= 3; ++i) {
        char tmp = char(((bin[i * 4] - '0') << 3)
            + ((bin[i * 4 + 1] - '0') << 2)
            + ((bin[i * 4 + 2] - '0') << 1)
            + (bin[i * 4 + 3] - '0'));
        tmp = (tmp >= 10) ? (tmp - 10 + 'A') : (tmp + '0');
        ans += tmp;
    }
    return ans;
}
```

- Assembly function, used to convert Assembly functions to LC-3 object code.

  For function *TranslateOprand*, when *str* is a label, we need to calculate the distance between the label and our current instruction.

```
std::string assembler::TranslateOprand(int current_address,
    std::string str, int opcode_length) {
    // Translate the oprand
    str = Trim(str);
    auto item = label_map.GetValue(str);
    if (!(item.getType() == vAddress && item.getVal() == -1)) {
        // str is a label
        int dest = item.getVal() - current_address - 1;
        int range = (1 << (opcode_length - 1)) - 1;
        if (dest > range || dest < -range - 1) {
            std::cout << str << "␣can't␣be␣presented␣by␣"
                << opcode_length
                << "␣of␣␣2's␣complement␣number" << std::endl;
            exit(1);
        }
        std::string ans = NumberToAssemble(dest);
        return ans.substr(ans.length() - opcode_length);
    }
    if (str[0] == 'R') {
        // str is a register
        std::string tmp = NumberToAssemble(str.substr(1));
        return tmp.substr(tmp.length() - 3);
    }
    else {
        // str is an immediate number
        std::string tmp = NumberToAssemble(str);
        return tmp.substr(tmp.length() - opcode_length);
    }
}
```

For function *assemble*, though the code is very long (the finished code has a length of 656 lines), we can find many instructions implemented in a similar way, such as *.FLKW* and *.STRINGZ*, *ADD* and *AND*. What's more, thanks to std::istringstream, we can handle oprands simply.

Correctness Verification

In fact, though the output file expansion name is *.asm*, the program convert an Assembly file into a *.bin* file. So we can put the output file into LC3Tools for correctness verification. After finish labS, it will be more easy to do this.
We have used the Assembly code written in lab4 and lab5 for test cases.