中国科学技术大学计算机学院

《ICS实验报告》

实验题目：LC3汇编器和模拟器

学生姓名：柯志伟

学生学号：PB20061338

完成时间：2022年1月17日

实验题目

> *LC3*汇编器和*LC3*模拟器

实验目的

> 1. 将*LC3*的汇编代码译为其对应的机器码
> 2. 用高级语言(*C++*)实现对*LC3*机器码执行效果的模拟，熟悉计算机执行机器码的流程

实验环境

> *Visual Studio 2022*

实验过程

1. 汇编器
   首先，机器码每行都是16位的01序列，而汇编代码有操作行，注释行，伪指令行，而每行构成有(以操作行为例)：操作行-->标识符，操作码，操作数，注释；因此要区分出不同行的类型，再根据具体类型，提取该行的各种构成，译为01码

对汇编文件进行三次扫描：

- Scan #0: Read file ,Store comments
- Scan #1: Scan for the .ORIG & .END pseudo code ,Scan for jump label, value label, line comments
- Scan #2: Translate

通读助教的代码，部分代码补全如下：

- Scan#0

```
        // Scan #0:
         // Read file
         // Store comments
         while (std::getline(input_file, line)) {
             // Remove the leading and trailing whitespace
             line = Trim(line);
             if (line.size() == 0) {
                 // Empty line
                 continue;
             }
             std::string origin_line = line;
```

```
            // Convert `line` into upper case
            // TO BE DONE
                for (int i = 0; i < line.size(); i++)
                    line[i] = toupper(line[i]);

            // Store comments
            auto comment_position = line.find(";");
            if (comment_position == std::string::npos) {
                // No comments here
                file_content.push_back(line);
                origin_file.push_back(origin_line);
                file_tag.push_back(lPending);
                file_comment.push_back("");
                file_address.push_back(-1);
                continue;
            } else {
                // Split content and comment
                // TO BE DONE
                    std::string comment_str =
line.substr(comment_position + 1);
                    std::string content_str = line.substr(0,
comment_position);
                // Delete the leading whitespace and the trailing
whitespace
                    comment_str = Trim(comment_str);
                    content_str = Trim(content_str);
                // Store content and comment separately
                file_content.push_back(content_str);
                origin_file.push_back(origin_line);
                file_comment.push_back(comment_str);
                if (content_str.size() == 0) {
                    // The whole line is a comment
                    file_tag.push_back(lComment);
                } else {
                    file_tag.push_back(lPending);
                }
                file_address.push_back(-1);
            }
        }
    } else {
        std::cout << "Unable to open file" << std::endl;
        // @ Input file read error
        return -1;
    }
```

- Scan#1

```
 // Scan #1:
    // Scan for the .ORIG & .END pseudo code
```

```
    // Scan for jump label, value label, line comments
    int line_address = -1;
    for (int line_index = 0; line_index < file_content.size();
++line_index) {
        if (file_tag[line_index] == lComment) {
            // This line is comment
            continue;
        }

        auto line = file_content[line_index];

        // * Pseudo Command
        if (line[0] == '.') {
            file_tag[line_index] = lPseudo;
            // This line is a pseudo instruction
            // Only .ORIG & .END are line-pseudo-command
            auto line_stringstream = std::istringstream(line);
            std::string pseudo_command;
            line_stringstream >> pseudo_command;

            if (pseudo_command == ".ORIG") {
                // .ORIG
                std::string orig_value;
                line_stringstream >> orig_value;
                orig_address = RecognizeNumberValue(orig_value);
                if (orig_address == std::numeric_limits<int>::max()) {
                    // @ Error address
                    return -2;
                }
                file_address[line_index] = -1;
                line_address = orig_address;
            } else if (pseudo_command == ".END") {
                // .END
                file_address[line_index] = -1;
                // If set line_address as -1, we can also check if there
are programs after .END
                 line_address = -1;
            } else if (pseudo_command == ".STRINGZ") {
                file_address[line_index] = line_address;
                std::string word;
                line_stringstream >> word;
                if (word[0] != '\"' || word[word.size() - 1] != '\"') {
                    // @ Error String format error
                    return -6;
                }
                auto num_temp = word.size() - 1;
                line_address += num_temp;
            } else if (pseudo_command == ".FILL") {
                // TO BE DONE
```

```
                file_address[line_index] = line_address;
                std::string word;
                line_stringstream >> word;
                auto num_temp = RecognizeNumberValue(word);
                if (num_temp == std::numeric_limits<int>::max()) {
                    // @ Error Invalid Number input @ FILL
                    return -4;
                }
                line_address += 1;
            } else if (pseudo_command == ".BLKW") {
                // TO BE DONE
                file_address[line_index] = line_address;
                std::string word;
                line_stringstream >> word;
                auto num_temp = RecognizeNumberValue(word);
                line_address += num_temp;

            } else {
                // @ Error Unknown Pseudo command
                return -100;
            }

            continue;
        }

        if (line_address == -1) {
            // @ Error Program begins before .ORIG
            // @ Error Program exists after .END
            return -3;
        }

        file_address[line_index] = line_address;
        line_address++;                                 // The address of
next line

        // Split the first word in the line
        auto line_stringstream = std::stringstream(line);
        std::string word;
        line_stringstream >> word;
        if (IsLC3Command(word) != -1 || IsLC3TrapRoutine(word) != -1) {
            // * This is an operation line
            // TO BE DONE
            file_tag[line_index] = lOperation;
            continue;
        }

        // * Label
        // Store the name of the label
        auto label_name = word;
```

```
        // Split the second word in the line
        line_stringstream >> word;
        if (IsLC3Command(word) != -1 || IsLC3TrapRoutine(word) != -1 ||
word == "") {
            // a label used for jump/branch
            // TO BE DONE
            file_tag[line_index] = lOperation;
            label_map.AddLabel(label_name, value_tp(vAddress,
line_address - 1));

        } else {
            file_tag[line_index] = lPseudo;
            if (word == ".FILL") {
                line_stringstream >> word;
                auto num_temp = RecognizeNumberValue(word);
                if (num_temp == std::numeric_limits<int>::max()) {
                    // @ Error Invalid Number input @ FILL
                    return -4;
                }
                if (num_temp > 65535 || num_temp < -65536) {
                    // @ Error Too large or too small value  @ FILL
                    return -5;
                }
                label_map.AddLabel(label_name, value_tp(vValue,
line_address - 1));
            }
            if (word == ".BLKW") {
                // modify label map
                // modify line address
                // TO BE DONE
                label_map.AddLabel(label_name, value_tp(vValue,
line_address - 1));
                std::string word;
                line_stringstream >> word;
                auto num_temp = RecognizeNumberValue(word);

                //TODO: add the restriction to the num_temp
                line_address += num_temp;
                line_address -= 1;

            }
            if (word == ".STRINGZ") {
                // modify label map
                // modify line address
                // TO BE DONE
                label_map.AddLabel(label_name, value_tp(vValue,
line_address - 1));
                std::string word;
                line_stringstream >> word;
```

```
                if (word[0] != '\"' || word[word.size() - 1] != '\"') {
                    // @ Error String format error
                    return -6;
                }
                auto num_temp = word.size() - 1;
                line_address += num_temp;
                line_address -= 1;

            }
        }
    }

    if (gIsDebugMode) {
        // Some debug information
        std::cout << std::endl;
        std::cout << "Label Map: " << std::endl;
        std::cout << label_map << std::endl;

        for (auto index = 0; index < file_content.size(); ++index) {
            std::cout << std::hex << file_address[index] << " ";
            std::cout << file_content[index] << std::endl;
        }
    }
```

- Scan#2

```
 // Check output file
    if (output_filename == "") {
        output_filename = input_filename;
        if (output_filename.find(".") == std::string::npos) {
            output_filename = output_filename + ".asm";
        } else {
            output_filename = output_filename.substr(0,
output_filename.rfind("."));
            output_filename = output_filename + ".asm";
        }
    }

    std::ofstream output_file;
    // Create the output file
    output_file.open(output_filename);
    if (!output_file) {
        // @ Error at output file
        return -20;
    }

    for (int line_index = 0; line_index < file_content.size();
++line_index) {
```

```
        if (file_address[line_index] == -1 || file_tag[line_index] ==
lComment) {
            // * This line is not necessary to be translated
            continue;
        }

        auto line = file_content[line_index];
        auto line_stringstream = std::stringstream(line);

        if (gIsDebugMode)
            output_file << std::hex << file_address[line_index] << ": ";
        if (file_tag[line_index] == lPseudo) {
            // Translate pseudo command
            std::string word;
            line_stringstream >> word;
            if (word[0] != '.') {
                // Fetch the second word
                // Eliminate the label
                line_stringstream >> word;
            }

            if (word == ".FILL") {
                std::string number_str;
                line_stringstream >> number_str;
                auto output_line = NumberToAssemble(number_str);
                if (gIsHexMode)
                    output_line = ConvertBin2Hex(output_line);
                output_file << output_line << std::endl;
            } else if (word == ".BLKW") {
                // Fill 0 here
                // TO BE DONE
                std::string number_str;
                line_stringstream >> number_str;
                auto num_temp = RecognizeNumberValue(number_str);
                for (int i = 0; i < num_temp; i++) {
                    auto output_line = "0000000000000000";
                    output_file << output_line << std::endl;
                }

            } else if (word == ".STRINGZ") {
                // Fill string here
                // TO BE DONE
                std::string str;
                line_stringstream >> str;
                for (int i = 0; i < str.size(); i++) {
                    char ch = str[i];
                    int num_ch = CharToDec(ch);
                    auto output_line = NumberToAssemble(num_ch);
                    output_file << output_line << std::endl;
```

```
                }

            }

            continue;
        }

        if (file_tag[line_index] == lOperation) {
            std::string word;
            line_stringstream >> word;
            if (IsLC3Command(word) == -1 && IsLC3TrapRoutine(word) ==
-1) {
                // Eliminate the label
                line_stringstream >> word;
            }

            std::string result_line = "";
            auto command_tag = IsLC3Command(word);
            auto parameter_str = line.substr(line.find(word) +
word.size());
            parameter_str = Trim(parameter_str);

            // Convert comma into space for splitting
            // TO BE DONE
            for (int i = 0; i < parameter_str.size(); i++) {
                if (parameter_str[i] == ',')
                    parameter_str[i] = ' ';
            }

            auto current_address = file_address[line_index];

            std::vector<std::string> parameter_list;
            auto parameter_stream = std::stringstream(parameter_str);
            while (parameter_stream >> word) {
                parameter_list.push_back(word);
            }
            auto parameter_list_size = parameter_list.size();
            if (command_tag != -1) {
                // This is a LC3 command
                switch (command_tag) {
                case 0:
                    // "ADD"
                    result_line += "0001";
                    if (parameter_list_size != 3) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0]);
```

```
                    result_line += TranslateOprand(current_address,
parameter_list[1]);
                    if (parameter_list[2][0] == 'R') {
                        // The third parameter is a register
                        result_line += "000";
                        result_line += TranslateOprand(current_address,
parameter_list[2]);
                    } else {
                        // The third parameter is an immediate number
                        result_line += "1";
                        // std::cout << "hi " << parameter_list[2] <<
std::endl;
                        result_line += TranslateOprand(current_address,
parameter_list[2], 5);
                    }
                    break;
                case 1:
                    // "AND"
                    // TO BE DONE
                    result_line += "0101";
                    if (parameter_list_size != 3) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0]);
                    result_line += TranslateOprand(current_address,
parameter_list[1]);
                    if (parameter_list[2][0] == 'R') {
                        // The third parameter is a register
                        result_line += "000";
                        result_line += TranslateOprand(current_address,
parameter_list[2]);
                    }
                    else {
                        // The third parameter is an immediate number
                        result_line += "1";
                        // std::cout << "hi " << parameter_list[2] <<
std::endl;
                        result_line += TranslateOprand(current_address,
parameter_list[2], 5);
                    }
                    break;
                case 2:
                    // "BR"
                    // TO BE DONE
                    result_line += "0000000";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
```

```
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0], 9);
                    break;
                case 3:
                    // "BRN"
                    // TO BE DONE
                    result_line += "0000100";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0], 9);
                    break;
                case 4:
                    // "BRZ"
                    // TO BE DONE
                    result_line += "0000010";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0], 9);
                    break;
                case 5:
                    // "BRP"
                    // TO BE DONE
                    result_line += "0000001";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0], 9);
                    break;
                case 6:
                    // "BRNZ"
                    // TO BE DONE
                    result_line += "0000110";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0], 9);
                    break;
```

```
                case 7:
                    // "BRNP"
                    // TO BE DONE
                    result_line += "0000101";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0], 9);
                    break;
                case 8:
                    // "BRZP"
                    // TO BE DONE
                    result_line += "0000011";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0], 9);
                    break;
                case 9:
                    // "BRNZP"
                    result_line += "0000111";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0], 9);
                    break;
                case 10:
                    // "JMP"
                    // TO BE DONE
                    result_line += "1100000";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0]);
                    break;
                case 11:
                    // "JSR"
                    // TO BE DONE
                    result_line += "01001";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
```

```
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0], 11);
                    break;
                case 12:
                    // "JSRR"
                    // TO BE DONE
                    result_line += "0100000";
                    if (parameter_list_size != 1) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0]);
                    result_line += "000000";
                    break;
                case 13:
                    // "LD"
                    // TO BE DONE
                    result_line += "0010";
                    if (parameter_list_size != 2) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0]);
                    result_line += TranslateOprand(current_address,
parameter_list[1], 9);
                    break;
                case 14:
                    // "LDI"
                    // TO BE DONE
                    result_line += "1010";
                    if (parameter_list_size != 2) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0]);
                    result_line += TranslateOprand(current_address,
parameter_list[1], 9);
                    break;
                case 15:
                    // "LDR"
                    // TO BE DONE
                    result_line += "0110";
                    if (parameter_list_size != 3) {
                        // @ Error parameter numbers
```

```
                    return -30;
                }
                result_line += TranslateOprand(current_address,
parameter_list[0]);
                result_line += TranslateOprand(current_address,
parameter_list[1]);
                result_line += TranslateOprand(current_address,
parameter_list[2], 6);
                break;
            case 16:
                // "LEA"
                // TO BE DONE
                result_line += "1110";
                if (parameter_list_size != 2) {
                    // @ Error parameter numbers
                    return -30;
                }
                result_line += TranslateOprand(current_address,
parameter_list[0]);
                result_line += TranslateOprand(current_address,
parameter_list[1],9);
                break;
            case 17:
                // "NOT"
                // TO BE DONE
                result_line += "1001";
                if (parameter_list_size != 2) {
                    // @ Error parameter numbers
                    return -30;
                }
                result_line += TranslateOprand(current_address,
parameter_list[0]);
                result_line += TranslateOprand(current_address,
parameter_list[1]);
                result_line += "111111";
                break;
            case 18:
                // RET
                result_line += "1100000111000000";
                if (parameter_list_size != 0) {
                    // @ Error parameter numbers
                    return -30;
                }
                break;
            case 19:
                // RTI
                // TO BE DONE
                result_line += "1000000000000000";
                if (parameter_list_size != 0) {
```

```
                        // @ Error parameter numbers
                        return -30;
                    }
                    break;
                case 20:
                    // ST
                    result_line += "0011";
                    if (parameter_list_size != 2) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0]);
                    result_line += TranslateOprand(current_address,
parameter_list[1], 9);
                    break;
                case 21:
                    // STI
                    // TO BE DONE
                    result_line += "1011";
                    if (parameter_list_size != 2) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0]);
                    result_line += TranslateOprand(current_address,
parameter_list[1], 9);
                    break;
                case 22:
                    // STR
                    // TO BE DONE
                    result_line += "0111";
                    if (parameter_list_size != 3) {
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0]);
                    result_line += TranslateOprand(current_address,
parameter_list[1]);
                    result_line += TranslateOprand(current_address,
parameter_list[2],6);
                    break;
                case 23:
                    // TRAP
                    // TO BE DONE
                    result_line += "11110000";
                    if (parameter_list_size != 1) {
```

```
                        // @ Error parameter numbers
                        return -30;
                    }
                    result_line += TranslateOprand(current_address,
parameter_list[0],8);
                    break;
                default:
                    // Unknown opcode
                    // @ Error
                    break;
                }
            } else {
                // This is a trap routine
                command_tag = IsLC3TrapRoutine(word);
                switch (command_tag) {
                    case 0:
                    // x20
                    result_line += "1111000000100000";
                    break;
                    case 1:
                    // x21
                    result_line += "1111000000100001";
                    break;
                    case 2:
                    // x22
                    result_line += "1111000000100010";
                    break;
                    case 3:
                    // x23
                    result_line += "1111000000100011";
                    break;
                    case 4:
                    // x24
                    result_line += "1111000000100100";
                    break;
                    case 5:
                    // x25
                    result_line += "1111000000100101";
                    break;
                    default:
                    // @ Error Unknown command
                    return -50;
                }
            }

            if (gIsHexMode)
                result_line = ConvertBin2Hex(result_line);
            output_file << result_line << std::endl;
        }
```

```
    }
```

- 具体功能代码

```
void label_map_tp::AddLabel(const std::string &str, const value_tp &val)
{
    labels_.insert(std::make_pair(str, val));
}

value_tp label_map_tp::GetValue(const std::string &str) const {
    // Use (vAddress, -1) to represent the error case
    if (labels_.find(str) == labels_.end()) {
        // not found
        return value_tp(vAddress, -1);
    } else {
        return labels_.at(str);
    }
}

std::ostream &operator<<(std::ostream &os, const StringType &item) {
    switch (item) {
    case sComment:
        os << "Comment ";
        break;
    case sLabel:
        os << "Label";
        break;
    case sValue:
        os << "Value";
        break;
    case sOpcode:
        os << "Opcode";
        break;
    case sOprand:
        os << "Oprand";
        break;
    default:
        os << "Error";
        break;
    }
    return os;
}

std::ostream &operator<<(std::ostream &os, const ValueType &val) {
    switch (val) {
    case vAddress:
        os << "Address";
```

```
        break;
    case vValue:
        os << "Value";
        break;
    default:
        os << "Error";
        break;
    }
    return os;
}

std::ostream &operator<<(std::ostream &os, const value_tp &value) {
    if (value.type_ == vValue) {
        os << "[ " << value.type_ << " -- " << value.val_ << " ]";
    } else {
        os << "[ " << value.type_ << " -- " << std::hex << "0x" <<
value.val_ << " ]";
    }
    return os;
}

std::ostream &operator<<(std::ostream &os, const label_map_tp
&label_map) {
    for (auto item : label_map.labels_) {
        os << "Name: " << item.first << " " << item.second << std::endl;
    }
    return os;
}

int RecognizeNumberValue(std::string s) {
    // Convert string s into a number
    // TO BE DONE
    int Num = 0;
    assert(s[0] == 'X' || s[0] == '#'|| s[0] == 'B');
    if (s[0] == 'X') {
        if (s[1] != '-') {
            assert(s[1] != '+');
            for (int i = 1; i < s.size(); i++)
                Num = Num * 16 + CharToDec(s[i]);
        }
        else {
            for (int i = 2; i < s.size(); i++)
                Num = Num * 16 + CharToDec(s[i]);
        }
    }
    else if(s[0] == '#') {
        if (s[1] != '-') {
            assert(s[1] != '+');
            for (int i = 1; i < s.size(); i++)
```

```
                Num = Num * 10 + CharToDec(s[i]);
        }
        else {
            for (int i = 2; i < s.size(); i++)
                Num = Num * 10+ CharToDec(s[i]);
        }
    }
    else{
        if (s[1] != '-') {
            assert(s[1] != '+');
            for (int i = 1; i < s.size(); i++)
                Num = Num * 2 + CharToDec(s[i]);
        }
        else {
            for (int i = 2; i < s.size(); i++)
                Num = Num * 2 + CharToDec(s[i]);
        }
    }
    if (s[1] != '-')
        return Num;
    else
        return -Num;
}

std::string NumberToAssemble(const int &number) {
    // Convert the number into a 16 bit binary string
    // TO BE DONE
    assert(number <= 32767 && number >= -32768);

    auto number_temp = number;
    std::string str;
    char buf[17];
    buf[16] = 0;
    if (number >= 0) {
        _itoa(number_temp, buf, 2);
        for (int i = 0; strlen(buf) + i < 16; i++)
            str += "0";
        str += buf;
    }

    else {
        number_temp += 32768;
        _itoa(number_temp, buf, 2);
        str += "1";
        str += buf;
    }
    return str.substr(str.size()-16,16);
 }
```

```
std::string NumberToAssemble(const std::string &number) {
    // Convert the number into a 16 bit binary string
    // You might use `RecognizeNumberValue` in this function
    // TO BE DONE
    int Num = RecognizeNumberValue(number);
    std::string str = NumberToAssemble(Num);
    return str;
}

std::string ConvertBin2Hex(std::string bin)  {
    // Convert the binary string into a hex string
    // TO BE DONE
    std::string str;
    std::string str_tmp;
    int Num;
    assert(bin.size() == 16);
    for (int i = 0; i < 4; i++){
        str_tmp = "B" + bin.substr(0, 4);
        Num = RecognizeNumberValue(str_tmp);
        str += DecToChar(Num);
        bin = bin.substr(4);
    }
    return str;
}

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
        // TO BE DONE
        int Num = item.getVal() - (current_address+1);
        std::string str_new = NumberToAssemble(Num);
        return str_new.substr(str_new.size() - opcode_length,
opcode_length);
    }
    if (str[0] == 'R') {
        int Num = CharToDec(str[1]);
        char buf[4];
        char buf_t[4];
        buf_t[0] = 0;
        buf[0] = 0;
        _itoa(Num, buf, 2);
        for (int i = 0; strlen(buf) + i < 3; i++)
            strcat(buf_t, "0");
        strcat(buf_t, buf);
        str = buf_t;
```

```
        return str.substr(str.size() - 3, 3);

    } else {
        // str is an immediate number
        // TO BE DONE
        str = NumberToAssemble(str);
        str = str.substr(str.size() - opcode_length, opcode_length);
        return str;
    }
}
```

2. 模拟器

计算机执行LC3机器码的流程：从指定的内存开始处(程序开始位置)，逐一取指令，由于其内部的电路逻辑实现对应的功能(ISA所定义的功能)，用高级语言实现时，只需将翻译出每行指令对应的功能并用高级语言直接实现即可

通读助教给的代码，整体框架已完备，程序接受两个文件的输入，一个是作为寄存器开始状态，另一个是内存中的数据(含代码)，在接受两个文件输入完成内存，寄存器的初始化后，即开始main函数的逻辑：逐指令读取并"翻译"、执行，在补充完整具体的功能模块(如VM_STR、VM_LEA、VM_RTI等)后，即完成

部分补全的代码如下：

- main函数

```
int main(int argc, char **argv) {
    po::options_description desc{"\e[1mLC3
SIMULATOR\e[0m\n\n\e[1mOptions\e[0m"};
    desc.add_options()
                          //
        ("help,h", "Help screen")
                           //
        ("file,f", po::value<std::string>()->default_value("input.txt"),
"Input file")              //
        ("register,r", po::value<std::string>()-
>default_value("register.txt"), "Register Status") //
        ("single,s", "Single Step Mode")
                          //
        ("begin,b", po::value<int>()->default_value(0x3000), "Begin
address (0x3000)")
        ("output,o", po::value<std::string>()->default_value(""),
"Output file")
        ("detail,d", "Detailed Mode");

    po::variables_map vm;
```

```
    store(parse_command_line(argc, argv, desc), vm);
    notify(vm);

    if (vm.count("help")) {
        std::cout << desc << std::endl;
        return 0;
    }
    if (vm.count("file")) {
        gInputFileName = vm["file"].as<std::string>();
    }
    if (vm.count("register")) {
        gRegisterStatusFileName = vm["register"].as<std::string>();
    }
    if (vm.count("single")) {
        gIsSingleStepMode = true;
    }
    if (vm.count("begin")) {
        gBeginningAddress = vm["begin"].as<int>();
    }
    if (vm.count("output")) {
        gOutputFileName = vm["output"].as<std::string>();
    }
    if (vm.count("detail")) {
        gIsDetailedMode = true;
    }

    virtual_machine_tp
virtual_machine(gBeginningAddress,gInputFileName,gRegisterStatusFileName
);
    int halt_flag = true;
    int time_flag = 0;
    while(halt_flag) {
        // Single step
        // TO BE DONE
        int16_t next_addr = virtual_machine.NextStep();
        ++time_flag;
        if (next_addr == 0)
            halt_flag = false;

        if (gIsDetailedMode)
            std::cout << virtual_machine.reg << std::endl;

        if (gIsSingleStepMode)
            system("pause");

    }

    std::cout << virtual_machine.reg << std::endl;
    std::cout << "cycle = " << time_flag << std::endl;
```

```
    return 0;
}
```

- 寄存器初始化

```
virtual_machine_tp::virtual_machine_tp(const int16_t address, const
std::string &memfile, const std::string &regfile) {
    // Read memory
    if (memfile != ""){
        mem.ReadMemoryFromFile(memfile);
    }

    // Read registers
    std::ifstream input_file;
    input_file.open(regfile);
    if (input_file.is_open()) {
        int line_count = std::count(std::istreambuf_iterator<char>
(input_file), std::istreambuf_iterator<char>(), '\n');
        input_file.close();
        input_file.open(regfile);
        if (line_count >= 8) {
            for (int index = R_R0; index <= R_R7; ++index) {
                input_file >> reg[index];
            }
        } else {
            for (int index = R_R0; index <= R_R7; ++index) {
                reg[index] = 0;
            }
        }
        input_file.close();
    } else {
        for (int index = R_R0; index <= R_R7; ++index) {
            reg[index] = 0;
        }
    }

    // Set address
    reg[R_PC] = address;
    reg[R_COND] = 0;
}
```

- 内存初始化

```
void memory_tp::ReadMemoryFromFile(std::string filename, int
beginning_address) {
        // Read from the file
        // TO BE DONE
        std::ifstream input_file;
        std::string str;
        input_file.open(filename);
           if (input_file.is_open()) {
            int line_count = std::count(std::istreambuf_iterator<char>
(input_file), std::istreambuf_iterator<char>(), '\n');
            input_file.close();
            input_file.open(filename);
            for (int index = 0; index < line_count; ++index) {
                input_file >> str;
                memory[beginning_address + index] = 0;
                for (int i = 0; i < str.size(); i++)
                    memory[beginning_address + index] += (str[i]-'0') *
pow(2, str.size()-i-1);
            }
            input_file.close();
        }
        else {
            ;//PASS
        }
    }

    int16_t memory_tp::GetContent(int address) const {
        // get the content
        // TO BE DONE
        return memory[address];
    }

    int16_t& memory_tp::operator[](int address) {
        // get the content
        // TO BE DONE
        return memory[address];
    }
```

- 具体功能模块

```
namespace virtual_machine_nsp {
template <typename T, unsigned B>
inline T SignExtend(const T x) {
    // Extend the number
    // TO BE DONE
    T signal = x & (1 << (B - 1));
    T y = x;
```

```
    if (signal) {
        for (int i = 0; B + i < sizeof(T) * 8; i++) {
            y = y + (1 << (B + i));
        }
    }
    return y;

}

void virtual_machine_tp::UpdateCondRegister(int regname) {
    // Update the condition register
    // TO BE DONE
    if (reg[regname] > 0) reg[R_COND] = 0x0001;
    else if(reg[regname] < 0) reg[R_COND] = 0x0004;
    else reg[R_COND] = 0x0002;
}

void virtual_machine_tp::VM_ADD(int16_t inst) {
    int flag = inst & 0b100000;
    int dr = (inst >> 9) & 0x7;
    int sr1 = (inst >> 6) & 0x7;
    if (flag) {
        // add inst number
        int16_t imm = SignExtend<int16_t, 5>(inst & 0b11111);
        reg[dr] = reg[sr1] + imm;
    } else {
        // add register
        int sr2 = inst & 0x7;
        reg[dr] = reg[sr1] + reg[sr2];
    }
    // Update condition register
    UpdateCondRegister(dr);
}

void virtual_machine_tp::VM_AND(int16_t inst) {
    // TO BE DONE
    int flag = inst & 0b100000;
    int dr = (inst >> 9) & 0x7;
    int sr1 = (inst >> 6) & 0x7;
    if (flag) {
        // and inst number
        int16_t imm = SignExtend<int16_t, 5>(inst & 0b11111);
        reg[dr] = reg[sr1] & imm;
    }
    else {
        // and register
        int sr2 = inst & 0x7;
        reg[dr] = reg[sr1] & reg[sr2];
    }
```

```
    // Update condition register
    UpdateCondRegister(dr);
}

void virtual_machine_tp::VM_BR(int16_t inst) {
    int16_t pc_offset = SignExtend<int16_t, 9>(inst & 0x1FF);
    int16_t cond_flag = (inst >> 9) & 0x7;
    if (gIsDetailedMode) {
        std::cout << reg[R_PC] << std::endl;
        std::cout << pc_offset << std::endl;
    }
    if (cond_flag & reg[R_COND]) {
        reg[R_PC] += pc_offset;
    }
}

void virtual_machine_tp::VM_JMP(int16_t inst) {
    // TO BE DONE
    int BaseR = (inst >> 6) & 0x7;
    reg[R_PC] = reg[BaseR];
}

void virtual_machine_tp::VM_JSR(int16_t inst) {
    // TO BE DONE
    reg[R_R7] = reg[R_PC];
    int flag = inst & 0b100000000000;
    if (flag) {
        int16_t pc_offset = SignExtend<int16_t, 11>(inst & 0x7FF);
        reg[R_PC] += pc_offset;
    }
    else {
        int BaseR = (inst >> 6) & 0x7;
        reg[R_PC] = reg[BaseR];
    }
}

void virtual_machine_tp::VM_LD(int16_t inst) {
    int16_t dr = (inst >> 9) & 0x7;
    int16_t pc_offset = SignExtend<int16_t, 9>(inst & 0x1FF);
    reg[dr] = mem[reg[R_PC] + pc_offset];
    UpdateCondRegister(dr);
}

void virtual_machine_tp::VM_LDI(int16_t inst) {
    // TO BE DONE
    int16_t dr = (inst >> 9) & 0x7;
    int16_t pc_offset = SignExtend<int16_t, 9>(inst & 0x1FF);
    reg[dr] = mem[mem[reg[R_PC] + pc_offset]];
    UpdateCondRegister(dr);
```

```
}

void virtual_machine_tp::VM_LDR(int16_t inst) {
    // TO BE DONE
    int16_t dr = (inst >> 9) & 0x7;
    int16_t BaseR = (inst >> 6) & 0x7;
    int16_t pc_offset = SignExtend<int16_t, 6>(inst & 0x3F);
    reg[dr] = mem[reg[BaseR] + pc_offset];
    UpdateCondRegister(dr);
}

void virtual_machine_tp::VM_LEA(int16_t inst) {
    // TO BE DONE
    int16_t dr = (inst >> 9) & 0x7;
    int16_t pc_offset = SignExtend<int16_t, 9>(inst & 0x1FF);
    reg[dr] = reg[R_PC] + pc_offset;
    UpdateCondRegister(dr);
}

void virtual_machine_tp::VM_NOT(int16_t inst) {
    // TO BE DONE
    int16_t dr = (inst >> 9) & 0x7;
    int16_t sr = (inst >> 6) & 0x7;
    reg[dr] = ~reg[sr];
    UpdateCondRegister(dr);
}

void virtual_machine_tp::VM_RTI(int16_t inst) {
    ; // PASS
}

void virtual_machine_tp::VM_ST(int16_t inst) {
    // TO BE DONE
    int16_t sr = (inst >> 9) & 0x7;
    int16_t pc_offset = SignExtend<int16_t, 9>(inst & 0x1FF);
    mem[reg[R_PC] + pc_offset] = reg[sr];
}

void virtual_machine_tp::VM_STI(int16_t inst) {
    // TO BE DONE
    int16_t sr = (inst >> 9) & 0x7;
    int16_t pc_offset = SignExtend<int16_t, 9>(inst & 0x1FF);
    mem[mem[reg[R_PC] + pc_offset]] = reg[sr];
}

void virtual_machine_tp::VM_STR(int16_t inst) {
    // TO BE DONE
    int16_t sr = (inst >> 9) & 0x7;
    int16_t BaseR = (inst >> 6) & 0x7;
```

```
    int16_t pc_offset = SignExtend<int16_t, 6>(inst & 0x3F);
    mem[reg[BaseR] + pc_offset] = reg[sr];
}

void virtual_machine_tp::VM_TRAP(int16_t inst) {
    int trapnum = inst & 0xFF;
    if (trapnum == 0x25)
        ;
    // TODO: build trap program
    else if (trapnum == 0x20){
        char ch = getchar();
        int16_t num = 0x00FF & ch;
        reg[R_R0] = num;

    }
    else if (trapnum == 0x21)
    {
        char ch = reg[R_R0] & 0xFF;
        putchar(ch);
    }
    else if (trapnum == 0x22)
    {
        char ch;
        int16_t addr = reg[R_R0];
        for (; ch = mem[addr] & 0xFF; addr++) {
            putchar(ch);
        }
    }
    else if (trapnum == 0x23)
    {
        printf(">> ");
        char ch = getchar();
        int16_t num = 0x00FF & ch;
        reg[R_R0] = num;
        putchar(ch);
    }
    else if (trapnum == 0x24)
    {
        char ch;
        int16_t addr = reg[R_R0];
        bool odd = true;
        int16_t num,num_tmp ;
        while (ch = getchar()) {
            if (odd) {
                num = ch & 0xFF;
                odd = false;
            }
            else {
                num_tmp = ch & 0xFF;
```

```
                num = num + num << 8;
                odd = true;
            }
            if (odd) {
                mem[addr] = num;
                addr++;
            }
        }
        if (!odd) {
            mem[addr] = num;
            addr++;
        }
        mem[addr] = 0x0000;

    }
}
```

总结与思考

通过本次实验，更清楚计算机执行机器码的流程，虽然代码主要框架由助教所写(这里感谢助教为我们减轻了难度)，但补全代码然有很多收获；另一方面，通过这次实验算是第一次用C++写代码，读代码(之前没学过，看来寒假要认真地完整学一下了)，万事开头难，其实在课下也零零散散看了一些C++的语法，但始终没实战过，这次实验算是开了个头，感觉收获还是很大的，这里，再次感谢学长这一学期的帮助。