lab A - Assembler

Purpose

本次实验的目的在于用高级语言写一个简易的汇编器，执行汇编器的基本功能，在此过程中加深对汇编语言的理解，也可以锻炼编程能力。

Principles

汇编器的工作过程为对代码进行两次遍历：

第一次遍历：

1. 去除所有的注释（以`;`开始），把所有字母转换为大写，用空格替换所有的逗号和`\t\n\r\f\v`等特殊字符，去除每行代码前后的所有空格。但是对于`.STRINGZ`开头的一行代码，只去除前后的空格即可，其后接的字符串不需要做任何改动。
2. 建立标签矢量表，如果一行代码的第一个字符不是操作码或伪指令或`TRAP`指令，就认为它是标签，将它添加到标签矢量表中。不过在此之前需要先判断标签是否合法——是否已经出现过，以及是否符合命名规则（这里我设置的判断规则为以字母或下划线开头且只含字母，数字或下划线）。

第二次遍历：将汇编码翻译为机器码，具体包括翻译操作码和操作数。

在遍历的过程中如果遇到错误，需要报告出错误的类型和位置。

Procedure

Part 1: The first pass

Step 1: Format every line

这一步需要补全框架中的`Trim`函数和`FormatLine`函数。

`Trim`：实现这一功能的思路为找到第一个非空白字符的位置`first`，如果存在，就删除字符串起始位置到`first`的部分，如果不存在就说明字符串为空串或是空白串，直接返回空串即可。如果没有返回，再紧接着找最后一个非空白字符的位置`last`，这里不需要再判断`last`是否存在，因为没有返回说明`first`存在，则`last`一定也存在，此时再删除从`last`到字符串末尾的部分即可。最后返回处理后的字符串。以下为代码实现：

```
// trim string from both left & right
static inline std::string &Trim(std::string &s) {
    // TO BE DONE
    // 寻找第一个非空格字符出现的位置然后删除从0到该位置的所有字符
    auto first = s.find_first_not_of(' ');
    if (first != std::string::npos) {
        s.erase(0, first);
    } else {
        // 如果全是空白字符，则清空字符串
        s.clear();
```

```
        return s;
    }

    // 再删除从最后一个非空格字符出现的位置到字符串末尾的所有字符
    auto last = s.find_last_not_of(' ');
    if (last != std::string::npos) {
        s.erase(last + 1);
    }

    return s;
}
```

`FormatLine`：首先要删除注释，只需要查找`;`的位置`commentPos`，若查找成功，则删除从`commentPos`到字符串末尾的部分，若失败则不需要处理。然后再判断字符串中是否存在`.STRINGZ`，若存在，返回`Trim(formattedline)`(`formattedline`是我在这个函数里设的一个变量，初始化为传入的字符串)即可；若不存在，再做以下处理：将小写字母转化为大写字母，用空格替换`,`及`\t\n\r\f\v`，`Trim(formattedline)`，最后返回`formattedline`即可。以下为代码实现：

```
static std::string FormatLine(const std::string &line) {
    // TO BE DONE
    std::string formattedLine = line;

    // 删除注释
    size_t commentPos = formattedLine.find(";");
    if (commentPos != std::string::npos) {
        formattedLine = formattedLine.substr(0, commentPos);
    }

    // 检查字符串是否含有".STRINGZ"，若含有，则此时直接返回即可
    if (formattedLine.find(".STRINGZ") != std::string::npos) {
        formattedLine = Trim(formattedLine);
        return formattedLine;
    }

    // 把字符串中的小写字母转换成大写字母(使用 std::transform and std:: toupper)
    std::transform(formattedLine.begin(), formattedLine.end(),
formattedLine.begin(), ::toupper);

    // 用空格替换","及', '
    std::replace(formattedLine.begin(), formattedLine.end(), ',', ' ');

    // 用空格替换"\t\n\r\f\v"
    std::string whitespaceChars = "\t\n\r\f\v";
    for (char c : whitespaceChars) {
        std::replace(formattedLine.begin(), formattedLine.end(), c, ' ');
    }

    //使用Trim函数去除字符串前后的空格
    formattedLine = Trim(formattedLine);
    return formattedLine;
}
```

Step 2: Store label

这一步需要补全框架中的`LineLabelSplit`函数，并且需要修改框架中的`AddLabel`函数。

`LineLabelSplit`函数较为简单，只需要先判断字符串的第一个空格前的子串`first_token`是否为标签（满足`IsLC3Pseudo(first_token) == -1 && IsLC3Command(first_token) == -1 && IsLC3TrapRoutine(first_token) == -1`），若是，则直接调用`AddLabel`函数将标签和地址加入到label_map中即可，然后再在字符串中删除`first_token`，如果字符串只含`first_token`，则返回空串，否则返回第一个空格后的子串。代码如下：

```
std::string assembler::LineLabelSplit(const std::string &line, int
current_address) {
    auto first_whitespace_position = line.find(' ');
    auto first_token = line.substr(0, first_whitespace_position);

    if (IsLC3Pseudo(first_token) == -1 && IsLC3Command(first_token) == -1 &&
        IsLC3TrapRoutine(first_token) == -1) {
        // * This is a label
        // save it in label_map
        // TO BE DONE
        // 直接调用AddLabel函数将标签和地址加入到label_map中即可
        label_map.AddLabel(first_token, current_address);

        // remove label from the line
        if (first_whitespace_position == std::string::npos) {
            // nothing else in the line
            return "";
        }
        auto command = line.substr(first_whitespace_position + 1);
        return Trim(command);
    }
    return line;
}
```

由于本次实验要求判断`Label`的合法性，所以还需要对框架中的`AddLabel`函数进行修改：

首先需要判断`Label`是否已经出现过，即`label_map`中是否已经有要添加的`Label`了。然后需要判断`Label`是否以字母或下划线开头，只含字母，数字或下划线（我为此增加了一个函数`IsValidLabel`）。代码如下：

```
// 检查Label的合法性
bool IsValidLabel(const std::string &label) {
    // Label的要求有：以字母或下划线开头，只包含字母、数字或下划线，不能重复

    // 先检查Label的首字符是否为字母或下划线
    if (!std::isalpha(label[0]) && label[0] != '_') {
        std::cout << "Label " << label << " is invalid "<< std::endl;
        return false;
    }

    // 再检查Label是否只含字母、数字或下划线
    for (char c : label) {
```

```
        if (!std::isalnum(c) && c != '_') {
            std::cout << "Label " << label << " is invalid "<< std::endl;
            return false;
        }
    }

    return true;
}

// add label and its address to symbol table
void LabelMapType::AddLabel(const std::string &str, const unsigned address) {
    // 先检查Label的合法性
    // 检查Label是否已经被定义过
    if (labels_.find(str) != labels_.end()) {
        printf("Label %s has been defined!\n",str.c_str());
        throw std::invalid_argument(str + "has been defined!");
    }
    else if (!IsValidLabel(str)) {
        // @ Error Invalid Label
        printf("Invalid Label %s!\n",str.c_str());
        //throw std::invalid_argument("Invalid label: " + str);
    }
    else
        labels_.insert({str, address});
}
```

Step 3: Complete the first pass

这一步需要补全框架中的`firstPass`函数，只需要根据`first_token`来修改`current_address`即可。

为了能够处理多个`.ORIG`的情况，在无指令部分全部填充0，我将.ORIG和.END出现的位置分别存在了一个全局变量数组（在第一次遍历前先全部初始化为-1）里而非框架中只用一个变量记录。

我还添加了一个变量`line_number`用来表示读到了文件的哪一行，方便报告错误的位置。以下为处理过程：

用一个`while`循环来从文件中读取汇编码的一行字符串`line`。若读到空串则`continue;`，否则调用`FormatLine`函数处理`line`，若处理后为空串则`continue;`。然后读取字符串第一个空格前的子串`first_token`，根据`first_token`做相应的处理：

- 若为`.ORIG`则`flag = 0`，`orig_value = command.substr(first_whitespace_position + 1)`，`orig_address = RecognizeNumberValue(orig_value)`（`RecognizeNumberValue`为头文件中的函数）。若`orig_address`为`<int>max`则说明有错误，因为出错时`RecognizeNumberValue`的返回值为`<int>max`，此时需要报告错误，否则`current_address = orig_address`最后再`continue;`继续读取下一行。
- 若为`.END`则`flag = 1`，`orin_address = -1`，然后再`continue;`。

  需要注意的时读取过程中如果读取到了`.END`后还没有读取到`.ORIG`就读取到了一行代码，那么需要报告错误，因为一段代码开始需要有`.ORIG`。

  需要注意的是在读到.ORIG或是.END时需要将其存到相应的数组里。
- 若为LC-3的操作数或是`TRAP`指令，则`commands.push_back({current_address, command, CommandType::OPERATION})`然后将`current_address`加1再`continue`即可。

- 若为伪指令`.FILL`，则需要先判断`.FILL`后接的数是否在范围内（-66536~66525，若不在则需要报错，否则将`current_address`加1再`continue`即可。
- 若为伪指令`.BLKW`，则用`.FILL`一样，需要先判断其后接的数（设为n）是否在范围内，不在则报错，在则将`current_address`加上n即可。
- 若为伪指令`.STRINGZ`，则需要先判断其后接的字符串是否合法——是否以双引号开始和结束，若不合法则报错，合法则将`current_address`加上n-3（设字符串长度为n，因为双引号和串尾空字符不需要存储，所以加上n-3）。

以下为代码实现：

```
int assembler::firstPass(std::string &input_filename) {
    std::string line;
    std::fstream input_file(input_filename, std::ios::in | std::ios::out);
    if (!input_file.is_open()) {
        std::cout << "Unable to open file" << std::endl;
        // @ Input file read error
        return -1;
    }
    memset(ORIG,-1,sizeof(int)*10);
    memset(END,-1,sizeof(int)*10);

    int orig_address = -1;
    int current_address = -1;
    int stringz_cnt = 0;//记录出现的.STRINGZ的数目
    int flag = 1;//设置一个变量flag，初始化为1，读取到.ORIG时置为0，读取到.END时置为1，这样就可以不处理.END到.ORIG的部分
    int line_number = 0;//记录读取的行号

    while (std::getline(input_file, line)) {

        line_number++;//每读取一行，将line_number加1
        line = FormatLine(line);
        if (line.empty()) {
            continue;
        }

        auto command = LineLabelSplit(line, current_address);
        if (command.empty()) {
            continue;
        }

        // OPERATION or PSEUDO?
        auto first_whitespace_position = command.find(' ');
        auto first_token = command.substr(0, first_whitespace_position);
        // Special judge .ORIG and .END
        if (first_token == ".ORIG") {
            flag = 0;
            std::string orig_value =
                command.substr(first_whitespace_position + 1);
            orig_address = RecognizeNumberValue(orig_value);

            //std::cout << orig_address<<std::endl;
            if (orig_address == std::numeric_limits<int>::max()) {
                // @ Error address
```

```
                // 打印错误提示信息
                std::cout << "Error: Invalid .ORIG value"<< orig_value << " at
line:" << line_number << std::endl;
                return -2;
            }
            ORIG[ORIG_num++] = orig_address;
            current_address = orig_address;
            continue;
        }

        if (orig_address == -1) {
            // @ Error Program begins before .ORIG
            // 打印错误提示信息
            std::cout << "Error: Program begins before .ORIG" << "at line:" <<
line_number << std::endl;
            return -3;
        }

        if (first_token == ".END") {
            //break;
            commands.push_back({current_address, command, CommandType::PSEUDO});
            END[END_num++] = current_address + stringz_cnt;
            flag = 1;
            orig_address = -1;
            continue;
        }

        // For LC3 Operation
        if (IsLC3Command(first_token) != -1 ||
            IsLC3TrapRoutine(first_token) != -1) {
            commands.push_back(
                {current_address, command, CommandType::OPERATION});
            current_address += 1;
            continue;
        }

        // For Pseudo code
        commands.push_back({current_address, command, CommandType::PSEUDO});
        auto operand = command.substr(first_whitespace_position + 1);
        if (first_token == ".FILL") {
            operand = Trim(operand);
            auto num_temp = RecognizeNumberValue(operand);
            if (operand.length() > 5 || num_temp ==
std::numeric_limits<int>::max()) {
                // @ Error Invalid Number input @ FILL
                // 打印错误信息
                std::cout << "Error: Invalid .FILL value " << operand << " at
line:" << line_number <<std::endl;
                return -4;
            }
            else if (num_temp > 65535 || num_temp < -65536) {
                // @ Error Too large or too small value  @ FILL
                // 打印错误信息
                std::cout << "Error: Too large or too small value " << operand
<< " at line:" << line_number <<std::endl;
```

```
                return -5;
            }
            current_address += 1;
        }
        if (first_token == ".BLKW") {
            // modify current_address
            // TO BE DONE
            // 这与上面的.FILL操作类似，不同的是.FILL只需分配1个存储单元，而.BLKW需要分配n
个存储单元
            auto num_temp = RecognizeNumberValue(operand);
            if (num_temp == std::numeric_limits<int>::max()) {
                // @ Error Invalid Number input @ FILL
                // 打印错误信息
                std::cout << "Error: Invalid .FILL value: " << operand << "at
line:" << line_number <<std::endl;
                return -4;
            }
            else if (num_temp > 65535 || num_temp < -65536) {
                // @ Error Too large or too small value  @ FILL
                // 打印错误信息
                std::cout << "Error: Too large or too small value: " << operand
<< "at line:" << line_number <<std::endl;
                return -5;
            }
            current_address += num_temp;

        }
        if (first_token == ".STRINGZ") {
            // modify current_address
            // TO BE DONE
            // 首先判断字符串的首尾是否为"(需要加转义字符\)
            operand = Trim(operand);
            if (operand[0] != '\"' || operand[operand.size() - 1] != '\"')
            {
                // 输出operand
                // @ Error String format error
                // 打印错误信息
                std::cout << "Error: String format error at line:" <<
line_number <<std::endl;
                return -6;
            }
            //然后改变current_address即可，这里需要减去两个双引号和一个空字符，故加上
operand.size() - 3
            current_address += operand.size() - 3;
            // 求字符串中转义字符的数量
            int escape_count = 0;
            for (int i = 1; i < operand.size() - 1; i++) {
                if (operand[i] == '\\') {
                    escape_count++;
                }
            }
            // 再将current_address减去转义字符的数量
            current_address -= escape_count;
            stringz_cnt++;
        }
```

```
    }
    // OK flag
    return 0;
}
```

Part 2: The second pass

这一部分需要补全框架中的`RecognizeNumberValue, NumberToAssemble, ConvertBin2Hex, TranslatePseudo, TranslateOprand, TranslateCommand`等函数，此外为了便于判断立即数和寄存器的合法性，我还补充了`IsValidImmediateNumber`和`IsValidRegisterNumber`两个函数，并且为了方便打印错误信息，我还添加了`PrintError`函数。

RecognizeNumberValue

首先需要调用`Trim`函数处理传递的字符串，设处理后得到字符串`temp`。若`temp`为空则返回`std::numeric_limits<int>::max()`。然后根据`temp`的首字符进行处理：

- 若首字符为`#`，则说明字符串表示十进制数，然后需要判断`temp[1]`是否为正负号，若为正负号则从`temp[2]`开始计算数值，否则从`temp[1]`开始计算数值。计算数值的方法为用一个变量`number`来记录，初始化为0，遍历字符串，将字符串每一位表示的数值加到`number`上，最后如果`temp[1] == '-'`，还需要将`number`赋值为`-number`。用代码表示为：

  ```
  number = 0;
  int begin = (temp.at(1) == '-' || temp.at(1) == '+') ? 2 : 1;
  for(int i = begin; i <= temp.length()-1; i++) {
      temp_int = CharToDec(temp.at(i));
      if(temp_int == -1 || temp_int > 9)
      {
          printf("Error: invalid number format: ");
          std::cout<< temp <<std::endl;
          return std::numeric_limits<int>::max();
      }
      number = number + pow(10,temp.length()-1-i)*temp_int;
  }
  if(temp.at(1) == '-')
      number = -number;
  return number;
  ```

  需要注意的是在调用`CharToDec`函数处理字符后需要判断得到的`temp_int`是否合法，若`temp_int == -1`说明转换失败，若`temp_int > 9`说明字符串中存在`A-F`中的数，这是不允许在`#`后面出现的。这两种情况都需要报错并返回`std::numeric_limits<int>::max()`。

- 若首字符为`x`或`X`，则说明字符串为十六进制数，处理过程与十进制数处理过程大致相同。不同的是，十六进制数中不允许出现正负号，它使用首位的符号位来表示正负。即`temp[1] > 7`时表示负数，需要用5减去转换得到的数值得到反码表示的数值，最后还需要将`number = -number - 1`。所以它的处理过程如下：

  ```
  number = 0;
  ```

```
            for(int i = 1; i <= temp.length()-1; i++) {
                temp_int = CharToDec(temp.at(i));
                if(temp_int == -1)
                {
                    printf("Error: invalid number format: ");
                    std::cout<< temp <<std::endl;
                    return std::numeric_limits<int>::max();
                }
                if(temp.at(1) > '7') {
                    temp_int = 15 - temp_int;
                }
                number = number + pow(16,temp.length()-1-i)*temp_int;
            }
            if(temp.at(1) > '7') {
                number = -number - 1;
            }
            return number;
```

- 若以上两种情况均不成立，则传入的字符串不合法，返回`std::numeric_limits<int>::max()`。

完整代码如下：

```
static int RecognizeNumberValue(const std::string &str) {
    auto temp = str;
    temp = Trim(temp);
    int number;
    int temp_int;
    // 首先判断temp是否为空串，如果是，则说明输入的字符串为空或空格串，返回最大的int值表示出错
    if(temp.empty()) {
        printf("Error: empty string!\n");
        return std::numeric_limits<int>::max();
    }

    try {
        if(temp.at(0) == '#') {
            //如果str以#开头，则说明它后面接的是十进制数，直接用stoi函数转换即可（需要从下一位开始，不能包含'#'）
            number = 0;
            int begin = (temp.at(1) == '-' || temp.at(1) == '+') ? 2 : 1;
            for(int i = begin; i <= temp.length()-1; i++) {
                temp_int = CharToDec(temp.at(i));
                if(temp_int == -1 || temp_int > 9)
                {
                    printf("Error: invalid number format: ");
                    std::cout<< temp <<std::endl;
                    return std::numeric_limits<int>::max();
                }
                number = number + pow(10,temp.length()-1-i)*temp_int;
            }
            if(temp.at(1) == '-')
                number = -number;
            return number;
            //return std::stoi(str.substr(1),nullptr,10);
        }
        else if(temp.at(0) == 'x' || temp.at(0) == 'X') {
```

```
            //如果str以x或X开头，则说明它后面接的是16进制数，也是调用stoi函数转换，此时进制为16
            number = 0;
            for(int i = 1; i <= temp.length()-1; i++) {
                temp_int = CharToDec(temp.at(i));
                if(temp_int == -1)
                {
                    printf("Error: invalid number format: ");
                    std::cout<< temp <<std::endl;
                    return std::numeric_limits<int>::max();
                }
                if(temp.at(1) > '7') {
                    temp_int = 15 - temp_int;
                }
                number = number + pow(16,temp.length()-1-i)*temp_int;
            }
            if(temp.at(1) > '7') {
                number = -number - 1;
            }
            return number;
            //return std::stoi(str.substr(1),nullptr,16);
        }
        else {
            //当以上两种情况均不成立时，说明输入的字符串错误，返回最大的int值表示出错
            printf("Error: unrecognized number format!\n");
            return std::numeric_limits<int>::max();
        }
    } catch (std::invalid_argument&) {
        printf("Error: invalid number format!\n");
        return std::numeric_limits<int>::max();
    } catch (std::out_of_range&) {
        printf("Error: number out of range!\n");
        return std::numeric_limits<int>::max();
    }
}
```

NumberToAssemble

对于`NumberToAssemble(const int &number)`，因为整数值在内存中是以二进制方式存储的，所以只需要从前往后判断`number`的每一位是否为是0还是1，得到的结果为`bit = (number >> i) & 1;`，再将`bit`加到要返回的字符串`binaryString`后即可。代码实现如下：

```
static std::string NumberToAssemble(const int &number) {
    // Convert `number` into a 16 bit binary string
    // TO BE DONE
    std::string binaryString;
    int bit = 0;
    for (int i = 15; i >= 0; i--) {
        bit = (number >> i) & 1;//每次将number右移i位再与1进行与运算，这样可以判断number的第i位是否为1
        binaryString += std::to_string(bit);//因为i是从15开始递减的，所以将bit添加到binaryString的后面
    }
    return binaryString;
}
```

对于`NumberToAssemble(const std::string &number)`，只需要先调用`RecognizeNumberValue`得到字符串表示的数值，再调用`NumberToAssemble(const int &number)`函数即可，代码如下：

```
static std::string NumberToAssemble(const std::string &number) {
    // Convert `number` into a 16 bit binary string
    // You might use `RecognizeNumberValue` in this function
    // TO BE DONE
    //先调用RecognizeNumberValue函数将字符串转换为int类型的数
    int new_num = RecognizeNumberValue(number);
    //然后直接返回上一个参数为int类型的NumberToAssemble函数即可
    return NumberToAssemble(new_num);
}
```

ConvertBin2Hex

只需要每次读取字符串的4位（如果不足，只处理剩余的字符），将子串转换成十进制数，再将十进制数以16进制的形式写入要返回的字符串后面即可，代码如下：

```
static std::string ConvertBin2Hex(const std::string &bin) {
    // Convert the binary string `bin` into a hex string
    // TO BE DONE
    std::string hex;
    int size = 4;
    int len = bin.length();
    for (int i = 0; i < len; i += size) {
        // 如果剩余的字符数量少于4，只处理剩余的字符
        if (i + size > len) {
            size = len - i;
        }
        std::string nibble = bin.substr(i, size);//从第i个字符开始，提取长为size的子串
        int decimal = std::stoi(nibble, nullptr, 2);//将nibble转换成10进制的形式
        std::stringstream ss;
        // 将十进制数以16进制的形式写入ss中，并且使用大写字母的形式
        ss << std::hex << std::uppercase << decimal;
        hex += ss.str();// 将ss中的内容添加到hex的后面
    }
    return hex;
}
```

TranslatePseudo

首先读取伪指令`pseudo_opcode`，然后根据`pseudo_opcode`进行处理：

- 若`pseudo_opcode == .END`，则需要判断他是否是最后一个.END，这里因为我将.ORIG和.END数组都全部初始化为-1，所以如果.END的下一个.ORIG不是-1，则说明它不是最后一个.END。如果不是，则需要填充0，总共需要填充`ORIG[i+1] - END[i]`行（假设当前是第i个.END，这里的i也需要用一个全局变量，初始化为0，没读取到一个.END就自增1）。
- 若`pseudo_opcode == ".FILL"`，则将其后的字符串转换成二进制码赋给`outline`(返回值)即可。
- 若`pseudo_opcode == ".BLKW"`，则需要先识别其后跟的数值(用n表示)，然后向`outline`中填充n个"0000000000000000\n"，最后还要再删掉最后一个`\n`防止出现空行。
- 若`pseudo_opcode == ".STRINGZ"`，则先将其后跟的字符串去除首尾空格和首尾双引号，然后依次将每个字符的ASCII码的二进制码写入`outline`中，再写入`\n`，最后在串尾添加`\0`即向`outline`中写入`0000000000000000`。需要注意的是转义字符的处理，如果读取到了`\`并且它的后面还要字符，就需要将`\`和它后面的字符作为一个转义字符处理，由于转义字符的数量有限且分布没有规律，所以可以`if else`或是`case`语句罗列所有情况。

  最后需要的是空格的处理，理论上来说空格并不需要额外处理，但是框架有bug，它使用的是`>>`来读取字符串，在遇到空格时会停止，这就导致它无法处理字符串中含有空格的情况，所以我用下面这行代码来替代了`>>`：

  ```
  std::string string_value;
  getline(command_stream, string_value);//读取一行，即读取.STRINGZ后的字符串
  ```

  这行代码会读取当前行剩下的所有字符串，不会被空格影响。

完整代码如下：

```
std::string assembler::TranslatePseudo(std::stringstream &command_stream) {
    std::string pseudo_opcode;
    std::string output_line;
    command_stream >> pseudo_opcode;
    if (pseudo_opcode == ".END") {
        if(ORIG[++END_count] != -1) {
            int num_temp = ORIG[END_count] - END[END_count-1];
            //std::cout<<num_temp<<std::endl;
            for (int i = 0; i < num_temp; i++) {
                output_line += "0000000000000000\n";
            }
            //删除最后一个"\n"，防止出现空行
            output_line = output_line.substr(0, output_line.size() - 1);
        }
        else
            output_line = "";
    }
    else if (pseudo_opcode == ".FILL") {
        std::string number_str;
        command_stream >> number_str;
        output_line = NumberToAssemble(number_str);
```

```
        //if (gIsHexMode)//gIsHexMode是一个全局变量，如果它为真，则当前的模式是十六进制模式，输出应该是十六进制
          //  output_line = ConvertBin2Hex(output_line);
    } else if (pseudo_opcode == ".BLKW") {
        // Fill 0 here
        // TO BE DONE
        std::string number_str;
        command_stream >> number_str;
        int num_temp = RecognizeNumberValue(number_str);
        // 因为在first_pass中已经检查过了.BLKW后的数值n的合法性，所以这里不需要再做检查
        for (int i = 0; i < num_temp; i++) {
            output_line += "0000000000000000\n";
        }
        //删除最后一个"\n"，防止出现空行
        output_line = output_line.substr(0, output_line.size() - 1);
    }
    else if (pseudo_opcode == "FILL_ZERO_LINE") {
        std::string number_str;
        command_stream >> number_str;
        int num_temp = RecognizeNumberValue(number_str);
        // 因为在first_pass中已经检查过了.BLKW后的数值n的合法性，所以这里不需要再做检查
        for (int i = 0; i < num_temp; i++) {
            output_line += "0000000000000000\n";
        }
        //删除最后一个"\n"，防止出现空行
        output_line = output_line.substr(0, output_line.size() - 1);
    }
    else if (pseudo_opcode == ".STRINGZ") {
        // Fill string here
        // TO BE DONE
        std::string string_value;
        getline(command_stream, string_value);//读取一行，即读取.STRINGZ后的字符串
        string_value = Trim(string_value);
        // 去除首尾双引号（因为在first_pass中已经检查过了.STRINGZ后字符串的合法性，所以这里不需要再做检查）
        string_value = string_value.substr(1, string_value.size() - 2);
        // 将字符串的每个字符添加到output_line中（每个字符单独占一行，故还需添加"\n"）
        for (size_t i = 0; i < string_value.size(); ++i) {
            char c = string_value.at(i);
            // 需要考虑c是否为转义字符
            if (c == '\\' && i + 1 < string_value.size()) {
                char next_c = string_value.at(i+1);

                if (next_c == 'n') {
                    c = '\n';
                    ++i;
                } else if (next_c == 't') {
                    c = '\t';
                     ++i;
                } else if (next_c == '\"') {
                    c = '\"';
                    ++i;
                } else if (next_c == '\\') {
                    c = '\\';
                    ++i;
```

```
                } else if (next_c == 'r') {
                    c = '\r';
                    ++i;
                } else if (next_c == 'a') {
                    c = '\a';
                    ++i;
                } else if (next_c == 'b') {
                    c = '\b';
                    ++i;
                } else if (next_c == 'f') {
                    c = '\f';
                    ++i;
                } else if (next_c == 'v') {
                    c = '\v';
                    ++i;
                } else if (next_c == '0') {
                    c = '\0';
                    ++i;
                } else if (next_c >= '1' && next_c <= '7') {
                    // 处理八进制转义字符
                    std::string octal_value;
                    while (i < string_value.size() && string_value[i] >= '0' &&
string_value[i] <= '7') {
                    octal_value += string_value[i++];
                }
                c = static_cast<char>(std::stoi(octal_value, nullptr, 8));
                } else if (next_c == 'x') {
                    // 处理十六进制转义字符
                    std::string hex_value;
                    while (i < string_value.size() && isxdigit(string_value[i]))
{
                        hex_value += string_value[i++];
                    }
                    c = static_cast<char>(std::stoi(hex_value, nullptr, 16));
                }
            }
            output_line = output_line + NumberToAssemble(static_cast<int>(c)) +
"\n";//需要先将字符c强制转换为整数形式，即它的ASCII码
        }
        // 串尾添加"\0"
        output_line += "0000000000000000";
    }
    if (gIsHexMode)//gIsHexMode是一个全局变量，如果它为真，则当前的模式是十六进制模式，输出
应该是十六进制
            output_line = ConvertBin2Hex(output_line);
    return output_line;
}
```

IsValidImmediateNumber

这个函数的参数为字符串`oprand`和串长`n`，只需要先调用`RecognizeNumberValue`函数得到字符串表示的数值，再判断其是否在`-2^(n-1)~2^(n-1)-1`之间即可。代码如下：

```
// 检查n位长的立即数是否合法
bool IsValidImmediateNumber(const std::string& operand,int n = 5) {
    int number = RecognizeNumberValue(operand);
    if(number >= -pow(2,n-1) && number <= pow(2,n-1)-1)
        return true;
    else
        return false;
}
```

IsValidRegisterNumber

这个函数的参数为字符串`operand`，只需要判断`operand`是否是`R`加上一个0到7之间的数构成的正则表达式即可，代码如下：

```
//检查寄存器号是否合法
bool IsValidRegisterNumber(const std::string& operand) {
    // Check if the operand is a valid register number
    auto temp = operand;
    temp = Trim(temp);
    std::regex register_regex("^R[0-7]$");//以R开头，后面接一个0-7之间的数字的正则表达式
    return std::regex_match(temp, register_regex);//检查operand是否为R0~R7中的一个
}
```

PrintError

这个函数的参数为`operand_list`，用于打印出错行的代码，只需要依次打印`operand_list`的每一位，中间用空格隔开即可，代码如下：

```
void PrintError(std::vector<std::string> operand_list) {
    int operand_list_size = operand_list.size();
    for(int i = 0;i < operand_list_size;i++)
        std::cout << operand_list.at(i) << " ";
    std::cout << std::endl;
}
```

TranslateOprand

这里我对框架中的代码做了修改，在传入的参数中增加了`operand_list`，即`TranslateCommand`中的`operand_list`，这是为了方便打印错误的位置。

首先要调用`Trim`函数去除传入字符串的首尾空格。

然后需要判断字符串是否为标签，若是，则计算偏移量，判断偏移量是否在合法范围内，若不合法，则报错，若合法，则返回`NumberToAssemble(offset).substr(16 - opcode_length)`，即偏移量的二进制字符串的后`opcode_length`位。

若不是标签，则根据它的首位字符是否为`R`来判断它是表示寄存器还是立即数。若是寄存器，则需要先调用`IsValidRegisterNumber`函数判断其是否合法，若合法，则将`str[0]`赋值为`#`，然后就可以调用`NumberToAssemble`函数，得到寄存器号的二进制字符串的后`opcode_length`位。若为立即数，则也是需要先调用`IsValidImmediateNumber`函数判断其合法性，若合法则返回`NumberToAssemble(str).substr(16 - opcode_length)`即可。

TranslateCommand

这个函数只需要根据指令类型来填入二进制码即可。具体思路为先根据指令类型填入前四位数，再判断后面接的操作数的数量是否合法，若不合法则报错，合法则调用`TranslateOprand`函数将操作数填入`output_line`中。

- 运算指令：以`ADD`为例，则首先需要将`0001`添加到`output_line`中，然后需要判断`operand_list`是否为3，若不为3则报错。然后调用`TranslateOprand`函数将后面两个表示寄存器的字符串添加到`output_line`中（这里需要调用`TranslateOprand`函数）。然后再判断最后的一个字符串是否以`R`开头即`operand_list[2][0]`是否等于`'R'`，若是，则`oprand_list[2]`表示寄存器，将`000`添加到`output_line`中，再将`TranslateOprand(current_address, operand_list[2],3,ope)`添加到`ouput_line`中，若不是，则将`1`和`TranslateOprand(current_address, operand_list[2],3,ope)`添加到`output_line`中。

  其他运算指令类似。
- 跳转指令
  - `Branch`类型指令

    首先将`0000`添加到`output_line`中，然后判断`oprand_list_size`是否为1，若不是，则报错。然后根据跳转的类型向`output_line`中填入`000`或`001`等。然后再将`TranslateOprand(current_address, operand_list[0],9,ope)`添加到`ouput_line`中。
  - `JMP`指令

    首先将`1100`添加到`output_line`中，然后判断`oprand_list_size`是否为1，若不是，则报错。然后向`output_line`中填入`000`。然后再将`TranslateOprand(current_address,operand_list[0],9,ope)`添加到`ouput_line`中。
  - `JSR`指令

    首先将`0100`添加到`output_line`中，然后判断`oprand_list_size`是否为1，若不是，则报错。然后向`output_line`中填入`1`。然后再将`TranslateOprand(current_address, operand_list[0],11,ope)`添加到`ouput_line`中。
  - `JSRR`指令

    首先将`0100`添加到`output_line`中，然后判断`oprand_list_size`是否为1，若不是，则报错。然后向`output_line`中填入`000`。然后再将`TranslateOprand(current_address,operand_list[0],3,ope)`添加到`ouput_line`中，最后填入`000000`。
- 取数及存数指令

以`LD`指令为例，首先根据指令类型填入前四位数`0010`，然后判断`oprand_list_size`是否为3，若不是，则报错。然后再将`TranslateOprand(current_address, operand_list[0],3,ope)`，`output_line += TranslateOprand(current_address, operand_list[1], 9,ope)`添加到`ouput_line`中。

其他指令类似，不同的是操作数的数目及长度会有不同。

- 返回指令

  `RET`指令：直接向`output_line`中填入`1100000111000000`即可，然后再判断`operand_list_size`是否为0，若不为0，则报错。

  `RTI`指令：直接向`output_line`中填入`1000000000000000`即可，然后再判断`operand_list_size`是否为0，若不为0，则报错。

- `TRAP`指令

  首先向`output_line`中填入`11110000`，然后再判断`operand_list_size`是否为1，若不为1，则报错。然后再向`output_line`中填入`TranslateOprand(current_address, operand_list[0], 8,ope)`。

- 若以上情况都不成立，则报错。

最后，若`gIsHexMode`为真，则调用`ConvertBin2Hex`函数将`output_line`转化为16进制。

以下为完整代码：

```
std::string assembler::TranslateCommand(std::stringstream &command_stream,
                                        unsigned int current_address) {
    std::string opcode;
    std::string br_immediate;
    command_stream >> opcode;
    auto command_tag = IsLC3Command(opcode);

    std::vector<std::string> operand_list;
    std::string operand;
    while (command_stream >> operand) {
        operand_list.push_back(operand);
    }
    auto operand_list_size = operand_list.size();
    std::string output_line;
    std::vector<std::string> ope;
    ope.push_back(opcode);
    for (const auto& op : operand_list) {
        ope.push_back(op);
    }

    if (command_tag == -1) {
        // This is a trap routine
        command_tag = IsLC3TrapRoutine(opcode);
        output_line = kLC3TrapMachineCode[command_tag];
    } else {
        // This is a LC3 command
        switch (command_tag) {
        case 0:
            // "ADD"
            output_line += "0001";
            if (operand_list_size != 3) {
```

```
                // @ Error operand numbers
                printf("Invalid operand numbers at ADD ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += TranslateOprand(current_address,
operand_list[1],3,ope);
            if (operand_list[2][0] == 'R') {
                // The third operand is a register
                output_line += "000";
                output_line += TranslateOprand(current_address,
operand_list[2],3,ope);
            } else {
                // The third operand is an immediate number
                output_line += "1";
                output_line += TranslateOprand(current_address, operand_list[2],
5,ope);
            }
            break;
        case 1:
            // "AND"
            // TO BE DONE
            output_line += "0101";
            if (operand_list_size != 3) {
                // @ Error operand numbers
                printf("Invalid operand numbers at AND ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += TranslateOprand(current_address,
operand_list[1],3,ope);
            if (operand_list[2][0] == 'R') {
                // The third operand is a register
                output_line += "000";
                output_line += TranslateOprand(current_address,
operand_list[2],3,ope);
            } else {
                // The third operand is an immediate number
                output_line += "1";
                output_line += TranslateOprand(current_address, operand_list[2],
5,ope);
            }
            break;
        case 2:
            // "BR"
            // TO BE DONE
            output_line += "0000";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at BR ");
                PrintError(operand_list);
```

```
                exit(-30);
            }
            output_line += "111";
            output_line += TranslateOprand(current_address, operand_list[0],
9,ope);
            break;
        case 3:
            // "BRN"
            // TO BE DONE
            output_line += "0000";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at BRn ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += "100";
            output_line += TranslateOprand(current_address, operand_list[0],
9,ope);
            break;
        case 4:
            // "BRZ"
            // TO BE DONE
            output_line += "0000";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at BRz ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += "010";
            output_line += TranslateOprand(current_address, operand_list[0],
9,ope);
            break;
        case 5:
            // "BRP"
            // TO BE DONE
            output_line += "0000";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at BRp ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += "001";
            output_line += TranslateOprand(current_address, operand_list[0],
9,ope);
            break;
        case 6:
            // "BRNZ"
            // TO BE DONE
            output_line += "0000";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at BRnz ");
```

```
                PrintError(operand_list);
                exit(-30);
            }
            output_line += "110";
            output_line += TranslateOprand(current_address, operand_list[0],
9,ope);
            break;
        case 7:
            // "BRNP"
            // TO BE DONE
            output_line += "0000";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at BRnp ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += "101";
            output_line += TranslateOprand(current_address, operand_list[0],
9,ope);
            break;
        case 8:
            // "BRZP"
            // TO BE DONE
            output_line += "0000";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at BRzp ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += "011";
            output_line += TranslateOprand(current_address, operand_list[0],
9,ope);
            break;
        case 9:
            // "BRNZP"
            output_line += "0000";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at BRnzp ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += "111";
            output_line += TranslateOprand(current_address, operand_list[0],
9,ope);
            break;
        case 10:
            // "JMP"
            // TO BE DONE
            output_line += "1100";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at JMP ");
```

```
                PrintError(operand_list);
                exit(-30);
            }
            output_line += "000";
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += "000000";
            break;
        case 11:
            // "JSR"
            // TO BE DONE
            output_line += "0100";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at JSR ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += "1";
            output_line += TranslateOprand(current_address, operand_list[0],
11,ope);
            break;
        case 12:
            // "JSRR"
            // TO BE DONE
            output_line += "0100";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at JSRR ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += "000";
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += "000000";
            break;
        case 13:
            // "LD"
            // TO BE DONE
            output_line += "0010";
            if (operand_list_size != 2) {
                // @ Error operand numbers
                printf("Invalid operand numbers at LD ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += TranslateOprand(current_address, operand_list[1],
9,ope);
            break;
        case 14:
            // "LDI"
            // TO BE DONE
```

```
            output_line += "1010";
            if (operand_list_size != 2) {
                // @ Error operand numbers
                printf("Invalid operand numbers at LDI ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += TranslateOprand(current_address, operand_list[1],
9,ope);
            break;
        case 15:
            // "LDR"
            // TO BE DONE
            output_line += "0110";
            if (operand_list_size != 3) {
                // @ Error operand numbers
                printf("Invalid operand numbers at LDR ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += TranslateOprand(current_address,
operand_list[1],3,ope);
            output_line += TranslateOprand(current_address, operand_list[2],
6,ope);
            break;
        case 16:
            // "LEA"
            // TO BE DONE
            output_line += "1110";
            if (operand_list_size != 2) {
                // @ Error operand numbers
                printf("Invalid operand numbers at LDR ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += TranslateOprand(current_address, operand_list[1],
9,ope);
            break;
        case 17:
            // "NOT"
            // TO BE DONE
            output_line += "1001";
            if (operand_list_size != 2) {
                // @ Error operand numbers
                printf("Invalid operand numbers at LDR ");
                PrintError(operand_list);
                exit(-30);
            }
```

```
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += TranslateOprand(current_address,
operand_list[1],3,ope);
            output_line += "111111";
            break;
        case 18:
            // RET
            output_line += "1100000111000000";
            if (operand_list_size != 0) {
                // @ Error operand numbers
                printf("Invalid operand numbers at RET ");
                PrintError(operand_list);
                exit(-30);
            }
            break;
        case 19:
            // RTI
            // TO BE DONE
            output_line += "1000000000000000";
            if (operand_list_size != 0) {
                // @ Error operand numbers
                printf("Invalid operand numbers at RTI ");
                PrintError(operand_list);
                exit(-30);
            }
            break;
        case 20:
            // ST
            output_line += "0011";
            if (operand_list_size != 2) {
                // @ Error operand numbers
                printf("Invalid operand numbers at ST ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += TranslateOprand(current_address, operand_list[1],
9,ope);
            break;
        case 21:
            // STI
            // TO BE DONE
            output_line += "1011";
            if (operand_list_size != 2) {
                // @ Error operand numbers
                printf("Invalid operand numbers at STI ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += TranslateOprand(current_address, operand_list[1],
9,ope);
```

```
            break;
        case 22:
            // STR
            // TO BE DONE
            output_line += "0111";
            if (operand_list_size != 3) {
                // @ Error operand numbers
                printf("Invalid operand numbers at STR ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += TranslateOprand(current_address,
operand_list[0],3,ope);
            output_line += TranslateOprand(current_address,
operand_list[1],3,ope);
            output_line += TranslateOprand(current_address,
operand_list[2],6,ope);
            break;
        case 23:
            // TRAP
            // TO BE DONE
            output_line += "11110000";
            if (operand_list_size != 1) {
                // @ Error operand numbers
                printf("Invalid operand numbers at TRAP ");
                PrintError(operand_list);
                exit(-30);
            }
            output_line += TranslateOprand(current_address, operand_list[0],
8,ope);
            break;
        default:
            // Unknown opcode
            // @ Error
            printf("Unknown opcode at line %d\n",current_address);
            exit(-40);
            //break;
        }
    }

    if (gIsHexMode)
        output_line = ConvertBin2Hex(output_line);

    return output_line;
}
```

Changes to the framework

由于本次实验有额外要求，所以不能简单的补全框架，还需要在一些地方进行修改，以下是我改动的一些部分（可能不全，因为我也记不太清哪些代码是我修改过的了）：

1. `TranslateOprand`函数。我在这个函数的参数中增加了变量`std::vector<std::string> operand_list`，这是为了方便报告错误，在使用`TranslateOprand`函数时，如果发现有错误，则直接依次打印`operand_list`的每一个元素，打印出错误的代码。

2. `AddLabel`函数。框架中直接使用了`labels_.insert({str, address})`来实现这一函数，我在这里增加对标签的重复性与合法性的判定，如果标签没有重复过并且合法，才使用`labels_.insert({str, address})`，否则报错并结束程序。

3. 对于多个.ORIG的情况的处理，我使用了ORIG数组和END数组来存储每一个.ORIG和.END出现的位置，这样就可以在.END和后一个.ORIG之间的无指令部分补充0。然后在`TranslatePseudo`函数中，还需要增加对.END的处理，即对于不是最后一个.END的.END，要在它和后一个.ORIG之间填充0，需要填充填充`ORIG[i+1] - END[i]`行（假设当前是第i个.END，这里的i也需要用一个全局变量，初始化为0，没读取到一个.END就自增1）。

4. 对于`.STRINGZ`的处理。

   - 首先需要注意的是在`FormatLine`函数中，如果检测到了`.STRINGZ`，就只需要删除注释和前后的空格，然后返回，以免改变`.STRINGZ`后接的字符串。框架中给出的要求并未涉及这个。
   - 其次是对于`.STRINGZ`后接的字符串有空格的情况的处理。在检测到`.STRINGZ`后，框架中用的是`>>`来读取字符串，在遇到空格时会停止，这就导致它无法处理字符串中含有空格的情况，所以我使用了`getline(command_stream, string_value)`来直接读取这一行剩下的所有字符，再用`Trim`函数去除`string_value`前后的空格，再去除前后的双引号（因为在`firstPass`中已经检查过字符串的合法性，所以这一步的字符串前后一定是有双引号的）即可得到所求字符串。
   - 最后是对`.STRINGZ`后接的字符串有转义字符的情况的处理。因为转义字符的数量有限且较少，所以我罗列了所有的情况，即读取到`\`时，检查其后面的字符，将两个字符作为一个转义字符，将这个转义字符的ASCII码（零扩展为16位）写入文件中。例如`\`的后面是`n`时，就`0000000000001010`写入文件中。

5. 对于错误的报告。框架中在遇到错误时只是结束了程序，并没有报告错误的类型和位置，我做了修改来报告错误的具体位置和类型。报告错误的类型很容易，只需要在错误出现的位置用`std::cout <<`或是`printf`打印相关信息即可。而对于错误的位置，我写了一个函数`PrintError`，前面也已经介绍过，它的参数是`operand_list`，作用为依次打印`operand_list`的每一个分量，这样就将错误的代码打印了出来，前面提到的修改`TranslateOprand`函数也是为了方便调用这个函数。此外我还写了`IsValidImmediateNumber`来判断立即数或是偏移量字符串是否合法，还有`IsValidRegisterNumber`函数来判断寄存器字符串是否合法，其中合法包括形式正确，范围正确，还有`IsValidLabel`函数来判断标签是否合法。

6. 其他可能遗漏的对框架中的代码做的一些小的修改。

Results

运行结果如下：

首先是将附件中的3个测试样例编译成机器码并与提供的`expected`文件进行比较：

```
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test1.asm -o ./test/actual/test1.bin
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> diff (cat ./test/actual/test1.bin) (cat ./test/expected/test1.bin)
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test2.asm -o ./test/actual/test2.bin
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> diff (cat ./test/actual/test2.bin) (cat ./test/expected/test2.bin)
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test3.asm -o ./test/actual/test3.bin
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> diff (cat ./test/actual/test3.bin) (cat ./test/expected/test3.bin)
```

运行的结果均与提供的一致，正确！（我这里用的是`windows power shell`而不是`linux`，所以比较的时候用的代码与讲义中的不一致，加了`cat`）

然后是一些典型错误的报错：

有`x-100`时：

```
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test4.asm -o ./test/actual/t
est4.bin
Error: Invalid .FILL value X-100 at line:2
```

立即数范围不合理时：

```
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test4.asm -o ./test/actual/t
est4.bin
Error: Invalid immediate number or offset at ADD R1 R1 #16
```

寄存器号不合理时：

```
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test4.asm -o ./test/actual/t
est4.bin
Error: Invalid register number at ADD R1 R1 R8
```

label不合法时：

label重复定义：

```
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test4.asm -o ./test/actual/t
est4.bin
Label LABEL1 has been defined!
```

label全是数字：

```
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test4.asm -o ./test/actual/t
est4.bin
Invalid Label 1234!
```

offfset不合理时：

6位的offset：

```
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test4.asm -o ./test/actual/t
est4.bin
Error: Invalid immediate number or offset at LDR R1 R1 #32
```

9位的offset：

```
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test4.asm -o ./test/actual/t
est4.bin
Error: Invalid immediate number or offset at BR #256
```

11位的offset：

```
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test4.asm -o ./test/actual/t
est4.bin
Error: Invalid immediate number or offset at JSR #1024
```

最后是对于特殊情况的处理，包括`.STRINGZ`后字符串中空格和转义字符的处理，多个.ORIG的处理。

`.STRINGZ`后接的字符串中有空格，转义字符时：

我设置的字符串为：`\nH\r ELLO\n`。

机器码如下：

```
0000000000001010
0000000001001000
0000000000001101
0000000000100000
0000000001000101
0000000001001100
0000000001001100
0000000001001111
0000000000001010
0000000000000000
```

结果正确！

多个.ORIG的处理：

我设置的两个.ORIG分别位于`x3000`和`x3100`处，第一个代码段有17行，以下为编译结果：

```
1010000000001101
1010001000001101
0001010010100001
1001001001111111
0001001001100001
0000010000000110
0101011000000010
0000010000000001
0001100100100001
0001010010000010
0001001001100001
0000111111111001
1011100011110011
1111000000100101
0011000100000000
0011000100000001
0011000100000010
0000000000000000
0000000000000000
0000000000000000
```

```
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
1111111111111111
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000001001000
0000000001000101
```

可以看到，全0行从第18行（第一个代码结束的地方）一直填充到第256行（第二段代码开始的前一行）。结果正确！

此时如果将第二段代码设置为从`x3200`开始，然后在第一段代码中跳转到第二段代码中的label会超出可跳转范围，应报错，以下为运行结果：

```
PS F:\ICSH\Lab\LAB_A_Attachment 2023\LAB_A_Attachment> ./assembler -f ./test/testcases/test4.asm -o ./test/actual/test4.bin
Error: can not reach label at BR A
```

结果符合预期，正确！

综上，本次实验运行结果均正确！