LabA Lc3Assembler

赖永凡　2022.1

实验要求

补全所给代码，实现一个LC3的汇编器，并通过所给代码的测试。

在介绍我补充的代码之余，我想同时叙述一下汇编器工作的大致历程，以便于更好地理解代码本身和编写的思路。

在实验报告中，我只添加了那些具有代表性的代码，对于剩下那些比较简单或重复性强的代码就不放出来平添繁琐了。

汇编历程

此程序对一份汇编码进行汇编一共需要遍历三次文件，以下将分别介绍。

Scan #0：文件读入及预处理

第一次扫描主要是将原文件读入内存。在读每一行之前都对其进行修剪（*Trim*）操作，即删除每一行首末的空格、制表符之类的无意义文字。
知道Trim函数的功能后，编写起来就十分简单了，只要稍微注意一下firstpos为-1的情况就可以。

```
// trim from left
inline std::string &LeftTrim(std::string &s, const char *t = " \t\n\r\f\v") {
    // TO BE DONE
    int firstpos = s.find_first_not_of(t);
    if (firstpos < 0) return s;
    else {
        s = s.substr(firstpos);
        return s;
    }

}

// trim from right
inline std::string &RightTrim(std::string &s, const char *t = " \t\n\r\f\v") {
    // TO BE DONE
    int lastpos = s.find_last_not_of(t);
    s = s.substr(0,lastpos+1);
    return s;
}

// trim from left & right
inline std::string &Trim(std::string &s, const char *t = " \t\n\r\f\v") {
    return LeftTrim(RightTrim(s, t), t);
}
```

修剪之后需要再将字符串转化为其大写格式，从而方便我们之后对指令的识别。
这里稍微"剧透"一下，最开始这里我只写了一行。然而，在之后测试的过程中，我发现不能一味地将读入的字符串全部转化成大写的形式。究其原因，LC3中存在着 .STRINGZ "#string_content" 的伪指令，而其中的字符串我们原则上是不可以改动的，因此我这一部分添加了一步判断语句，最终只转换两个字符串标志符\"之间的部分。
虽然后面的那个引号可能位于注释中，但其实并不影响最终的结果。

```
267         // Convert `line` into upper case
268         // TO BE DONE
269         int lstringpos = line.find_first_of('\"');
270         int rstringpos = line.find_last_of('\"');
271         if (lstringpos != std::string::npos && rstringpos != std::string::npos && lstringpos != rstringpos) {
272             std::string lpart = line.substr(0, lstringpos);
273             transform(lpart.begin(), lpart.end(), lpart.begin(), ::toupper);
274 
275             std::string midpart = line.substr(lstringpos, rstringpos);
276 
277             std::string rpart = line.substr(rstringpos + 1);
278             transform(rpart.begin(), rpart.end(), rpart.begin(), ::toupper);
279             line = lpart + midpart + rpart;
280         }
281         else {
282             transform(line.begin(), line.end(), line.begin(), ::toupper);
283         }
284 
```

在做完这些改动之后，我们再将句子中的注释从正文部分分离出来，并将两者分别存到两个向量中去。特别地，对于那些没有正文部分的行，将其打上*lcommen*的Tag。

至此，我们就完成了**Scan #0**的所有操作，总结起来，主要有三

1. Trim
2. 字符串转化成大写形式
3. 分离正文和注释

完成这些操作之后，我们对原文件进行了一定程度的规范，删繁就简，大大提高了工作效率。

Scan #1：处理伪指令和标签

经过之前的处理后，我们再次遍历读入的文件。如果某一行之前被打上了*lcommen*的Tag的话，我们就直接跳过这一行，否则，我们就将正文向量中对应的行提取出来。

在对行地址这一问题的处理上，本程序使用了以下方法。首先设置一个向量file_address来储存正文中对应行的地址，其次定义一个变量line_address（初始值为-1）来记录当前处理到的行地址。换言之，原文件中每一行都在file_address中有一个值，而line_address记录的是机器码对应的行的地址，也就是通常意义上的行地址。再具体而言，输出文件中所有指令的line_address都和file_address中储存值相同，而不会在其中出现的行其file_address对应值都是-1。

如果正文部分第一个字符是'.'的话，我们就认为这个是一个整行的伪指令，并将其打上*lPseudo*的Tag。

- ".ORIG"：将line_address置为其指定的行起始地址，同时将其file_address对应值置为-1。故若后来读取指令时检测到line_address仍为-1的话，就代表程序在".ORIG"前开始了。
- ".END"：本来对该指令的处理仅限于将其file_address对应值置为-1，但这样该行后的语句仍然会被汇编成机器码。为了弥补这一缺陷，我在这里添加了一个结尾标记。于是，在**Scan #1**中扫描每一行时，除了会跳过注释行之外，若检测到结尾标记，就将跳过后面所有的行，直接进入下一步骤。
- ".FILL"&".BLKW"&".STRINGZ"：将line_address的值存入file_address对应项中，然后根据初始化空间的长度修改line_address的值。这里在编写的时候我注意到已给代码的一个问题（注释部分），如果字符串中含有空格的话，按照他这样读取的word就只会包含空格前的一部分。不过修改起来也比较容易，只需要将剩下的内容全部读出来就行了。

```
365         } else if (pseudo_command == ".STRINGZ") {
366             file_address[line_index] = line_address;
367             //line_stringstream >> word;
368             //if (word[0] != '\"' || word[word.size() - 1] != '\"') {
369             //    // @ Error String format error
370             //    return -6;
371             //}
372             //auto num_temp = word.size() - 1;
373             //line_address += num_temp;
374             std::string word;
375             std::string stringword = "";
376             while (line_stringstream >> word) {
377                 stringword = stringword + word + " ";
378             }
379             if (stringword.size() < 1) return -6;
380             stringword = stringword.substr(0, stringword.size() - 1);
381             if (stringword[0] != '\"' || stringword[stringword.size() - 1] != '\"') {
382                 // @ Error String format error
383                 return -6;
384             }
385             line_address = line_address + stringword.size() - 2 - 1;
```

在判别完某行第一个词不是一个伪指令后，我们就将line_address加1，同时将其存入file_address对应项中，接着开始后续的判别。至此，读到的行才严格意义上有了地址。

接下去判断正文行的第一个词是否是LC3中的指令，若是，则将其标上*IOperation*的Tag，然后开始处理正文的下一行。

若该行的第一个词不是LC3中的指令关键词，我们就认为这个是一个标签。对于标签和其对应的地址，我们使用一张表来收集，以便于之后寻址使用。对于第一个词是标签的行，其语法大致可以分为两种：

- label Opcode Oprand
- label Pseudo-Op

无论那种，我们都需要读取行中第二个词才能够判断。如果第二个词是LC3中的指令，那自然就对应着的一种情况。这里所给程序还添加了另一种情况，即标签单独成行的情形，我们也认为这个是用于跳转类型指令的标签。在标准情况下，标签指示的都是当前行的地址。对于标签单独成行的情形，我这里将其指向的是下一个条有意义指令的地址。即若遇到连续两行都是一个标签（应该没有人会这样写程序），第三行才是指令的代码，那两个标签都会指向有指令的那一行。
当然，由于标签并不会在汇编出的机械码中体现，所以单独成行的标签其file_address对应项需要重新置为-1，这也就意味着我们之后不会再读到这一行了。

```
433         // * Label
434         // Store the name of the label
435         auto label_name = word;
436         // Split the second word in the line
437         line_stringstream >> word;
438         if (IsLC3Command(word) != -1 || IsLC3TrapRoutine(word) != -1 || word == "") {
439             // a label used for jump/branch
440             // TO BE DONE
441             if(word == "") {
442                 label_map.AddLabel(label_name, value_tp(vAddress, line_address));
443                 file_address[line_index] = -1;
444                 line_address--;
445             }
446             else {
447                 label_map.AddLabel(label_name, value_tp(vAddress, line_address - 1));
448                 file_tag[line_index] = lOperation;
449             }
450
```

对于第二个词是伪指令的情况，我们在将标签保存后处理方式就和整行形式的伪指令一样，这里就不再赘述。

至此，我们就完成了**Scan #1**的所有操作，总结如下：

依次读取正文向量中存储的每一行，对每一行依次做以下判断：

1. 判断是否为伪指令，若是，进行相应处理。
2. 将line_address的值赋给当前读到行在file_address向量中的对应项，line_address自增。
3. 判断该行第一个词是否是LC3中的指令，若是，进行相应处理。
4. 认为该行第一个词是一个标签，再接着判断此行的第二个词，并根据结果进行处理。

在完成此步骤之后，我们可以得到每一行在内存中的地址，以及一个标签和指向地址的表，有了这些，我们终于可以开始“汇编翻译”操作了。

Scan #2：翻译

虽说这里主要介绍的是汇编的历程，但我还是想在这里先插入一下关于主菜单的介绍。除了-f和-o及其参数外，在cmd中启动本程序时还允许加入以下指令：

- -h：输出帮助菜单。
- -e：输出错误信息

0：汇编正常	-1：无法打开文件	-2：起始地址非法
-3：程序在.ORIG之前开始	-4：无效的数字输入	-5：跳转标签地址越界
-6：字符串格式错误	-7：立即数越界	-8：寄存器编号非法
-9：含有未定义的标签	-20：输出文件错误	-30：指令中参数个数错误
-50：含有未知的指令	-100：含有未知的伪指令	

- -s：以十六进制输出。
- -d：输出Debug信息。

当然，以上功能仅是锦上添花，在核心功能完成后仅需要加入简单的代码就能实现。例如，在进入**Scan #2**之前，我们就可以判断本次运行是否是在DebugMode下。若是，就将我们之前得到的储存标签及地址的表打出。

在进入正式地将每一句话翻译的部分之前，我想先介绍一下对于我们后面的工作至关重要的几个函数或模块。在他们的帮助下，工作将变得简单而轻松。

- RecognizeNumberValue函数

  这个函数能够将一个字符串转换成整形的数，简而言之就是从字符串中识别出数字来。熟悉C++的人可能会觉得这不就只需要引用一下stoi函数就能解决吗，诚然，我实现此功能的核心部分还是stoi函数，而这个函数功能也确实强大，甚至能够识别出负数。但是，这对于LC3汇编语言来说还不够。我们知道，汇编语言中需要识别的数字有两种情况：

  1. 寄存器编号，例如R0。
  2. 立即数、伪指令中使用的数，这些数会带有提示进制的前缀。

  好在stoi函数具有识别十六进制和二进制数并转成十进制的功能，所以我们只需要讨论字符串的第一个字符的形式并加以分类就能够完成这个函数了。

```
80  int RecognizeNumberValue(std::string s) {
81      // Convert string s into a number
82      // TO BE DONE
83      int num_temp;
84      if(s[0] == 'B') return std::stoi(s.substr(1), 0, 2);
85      if(s[0] == 'X') return std::stoi(s.substr(1), 0, 16);
86      if(s[0] == '#') return std::stoi(s.substr(1));
87      return std::stoi(s, 0, 10);
88  }
89
```

- NumberToAssemble函数

  这个函数有两个重载，一个是将整形的十进制数转化成16位二进制形式的字符串，另一个则是将一个字符串形式的数转化成16位二进制字符串。显然，第二种形式的函数只需要先引用以下前面写好的RecognizeNumberValue函数，再引用一下第一种形式的重载就可以了，因此这里主要介绍前一种重载的编写。

  对于十进制转二进制的问题相信有关算法已有不少，这里用的也是最常见的除二取余的方法。不过，这里还需要考虑形参是一个负数的情况。由于负数的补码可以看做一个无符号整数，这里采用的是现将负数转换成对应无符号数的思想来处理负数的情况。

```cpp
std::string NumberToAssemble(const int &number) {
    // Convert the number into a 16 bit binary string
    // TO BE DONE
    std::string BinNumString = "";
    int num;
    if (number == 0) return "0000000000000000";
    if(number < 0) {
        num = 65536 + number;
    }
    else num = number;

    for( ;num >= 2;num = num / 2) {
        if(num % 2 == 1) BinNumString = "1" + BinNumString;
        else BinNumString = "0" + BinNumString;
    }
    BinNumString = "1" + BinNumString;

    for(int i = BinNumString.size(); i < 16 ; i++) {
        BinNumString = "0" + BinNumString;
    }

    return BinNumString;
}

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 BinNumString = NumberToAssemble(num);
    return BinNumString;
}
```

- ConvertBin2Hex函数

  顾名思义，就是将16位二进制字符串转换成4位16进制字符串形式，用于十六进制模式。比较简单，不多介绍。

- TranslateOprand函数

  这可以说是整个程序的核心，至此我不得不钦佩程序设计者思路的巧妙，用一个函数将所有的Operand统一起来处理，可谓是所有先前准备的集大成者，起到了承上启下的作用。

  这个函数旨在处理所有LC3汇编语言中的Operand，即标签、寄存器、数或立即数。它最多可以传入三个形参：当前地址，Operand字符串，返回字符串的长度。以下将按顺序介绍各自的处理方式。

  1. 标签——需要译出标签所指的地址

     在修剪（*Trim*）完传入的字符串后，我们先默认这是一个标签，因此在先前准备好的标签表中搜索，若能找到，就代表传入的字符串确实是一个标签。
     在标签表中找到这个标签的同时，我们也能获得该标签所在的地址，再根据PCoffset长度（即返回字符串的长度）检查完此地址的合法性后就翻译出PCoffset的值，最后再根据要求返回相应长度的二进制字符串。

```cpp
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 offset = item.getVal() - current_address - 1;

        if (opcode_length == 9) {
            if (offset > 255 || offset < -256) {
            //label out of range
            std::cout << "-5" << std::endl;
            exit(-5);
            }
        }
        if (opcode_length == 11) {
            if (offset > 1023 || offset < -1024) {
            //label out of range
            std::cout << "-5" << std::endl;
            exit(-5);
            }
        }
        if (opcode_length == 6) {
            if (offset > 31 || offset < -32) {
            //label out of range
            std::cout << "-5" << std::endl;
            exit(-5);
            }
        }

        std::string wholenum = NumberToAssemble(offset);
        return wholenum.substr(16 - opcode_length, opcode_length);

    }
```

  2. 寄存器——需要译出三位二进制的寄存器编号

若在标签表中找不到传入的字符串，我们就再进一步检验字符串第一个字符是不是R。若是，就代表传入的是一个寄存器代号。
这种情况比较简单，我们只要判断一下后面的字符是否合法，然后返回识别出的三位二进制字符串就好。

```
186     if (str[0] == 'R') {
187         // str is a register
188         // TO BE DONE
189         int register_index = RecognizeNumberValue(str.substr(1));
190         if (register_index > 7 || register_index < 0) {
191             //illegal register
192             std::cout << "-8" << std::endl;
193             exit(-8);
194         }
195         else return NumberToAssemble(register_index).substr(13, 3);
196
```

3. 数或立即数——需要译出指定位数的对应二进制数

单纯地翻译传入的数并返回指定的位数其实很简单，前提是，传入的确实是一个我们要翻译的数。换言之，从函数调用至此，我们已经确定了需要翻译的字符串不是一个**已有的**标签、也不是一个寄存器，那么还剩下两种可能：数或立即数——我们期望的——或一个**不存在的标签**——需要报错。
所以，在毫无防备地就把字符串中的数字识别出来之前，我们应该先判断这是不是一个应该被翻译出来的字符串。

```
197     } else {
198         // str is an immediate number
199         // TO BE DONE
200
201         if (str.find_first_of("XB#") == 0) {
202             if (str[0] == 'X') {
203                 if (str.substr(1).find_first_not_of("-0123456789ABCDEF") != std::string::npos) {
204                     //unknown label
205                     std::cout << "-9" << std::endl;
206                     exit(-9);
207                 }
208             }
209
210             if (str[0] == 'B') {
211                 if (str.substr(1).find_first_not_of("-01") != std::string::npos) {
212                     //unknown label
213                     std::cout << "-9" << std::endl;
214                     exit(-9);
215                 }
216             }
217
218             if (str[0] == '#') {
219                 if (str.substr(1).find_first_not_of("-0123456789") != std::string::npos) {
220                     //unknown label
221                     std::cout << "-9" << std::endl;
222                     exit(-9);
223                 }
224             }
225         }
226         else {
227             //unknown label
228             std::cout << "-9" << std::endl;
229             exit(-9);
230         }
```

这里，我主要是依据LC3汇编语言中数字的句法来判断是否为一个可翻译的数字而不是一个标签的。首先，先判断第一个字符是不是数字句法中第一个位置允许出现的字符，若不是，就可以直接下这是一个未知标签的判断。
对于那些通过判断的字符串，我们也能明白其对应的进制，因此只要能够在其中发现不应该出现的字符，就代表着这个数字表示非法，或这是一个未收录标签。值得一提的是，这样写的话是允许出现$\#-12-3$这种表达式的，至于会识别出什么，就全听从于stoi函数了。其实要改进也很简单，不过这就当做一个彩蛋留存于程序中吧。
做完了判断工作之后，我们就可以认为这是一个需要翻译出来的数。剩下的工作在前面函数的帮助下就很简单了，这里也不再赘述，只是要判断一下数的大小是否能够被指定的字符串长度允许。

至此，我们就完成了TranslateOprand函数的编写，这个函数的强大作用已经呼之欲出了。

终于，我们可以开始最后的翻译工作了。首先，我们需要生成一个输出文件，并给他一个名字。之后我们从头开始再次读取正文向量。和之前一样，若该行全是评论就直接跳过。

之后我们再次判断是否为DebugMode，若是，就将该行正文部分以及此行对应的地址（十六进制形式）在显示屏中打出。

若此行的Tag为*IPseudo*，就读出此行的第一个单词，若该单词的第一个字符不是“.”，代表着这是一个标签，而我们只需要再往后读一个单词就行了，因为若输入汇编代码正确，则标签后面的那个单词一定是伪指令。

- 对于伪指令".FILL"，我们只需要用一下NumberToAssemble函数，并将其返回的结果（16位二进制串）原封不动地输出到输出文件中即可。
- 对于伪指令".BLKW'，我们根据其后的数在输出文件中输出对应行数的0000 0000 0000 0000即可。
- 对于伪指令".STRINGZ"，将其每一个字符都转换成其对应的ACSII值，然后用NumberToAssemble函数获得每一个字符的对应行，并输出即可。值得一提的是，在最后一个字符之后需要再补一行全零行来作为字符串结尾标志符。

若此行的Tag为*IOperation*，同样，先判断第一个单词是不是LC3指令，如果不是，那就一定是第二个单词，否则就直接报错。
这里，我们需要将行中所有的逗号都转换成空格，以便于我们分开每一个参数，且分开后还要再将每一个参数保存起来。
我们现将第一个参数导出，他一定是此行的Opcode。对于所有可能出现的23中指令（对于BR指令，我们将其所有出现的形式都分别列举，因此有8种）和trap routine，我们都需要设置一种情况来实现，因此所谓翻译，实际上就是一个"查字典"的过程，对于每一种指令，我们都有一套翻译的方法。
为了避免繁复，这里只挑选具有代表性的指令来介绍。

- ADD指令

  令人莞尔的是，在LC3中使用和执行起来最简单的ADD指令到了翻译中却变成了最复杂的。首先，当我们识别并进入此"条目"时，直接在result_line中生成一个"0101"，此即ADD指令的代码。对于此指令，我们知道他的汇编语法为

  ADD DR SR1 SR2 或 ADD DR SR1 imm5

  编码后为：

  $0001 + \mathrm{DR} + \mathrm{SR1} + 000 + \mathrm{SR2}$ 或 $0001 + \mathrm{DR} + \mathrm{SR1} + 1 + \mathrm{imm5}$

  因此，我们先判断存下的参数是否为3个，若不是，就代表着一定有语法错误。无误后，对于Opcode之后的两个参数，我们都可以通过引用TranslateOprand函数来处理，毕竟他们一定代表着寄存器。之后，再判断第三个，也就是最后一个参数的第一个字符。若为"R"，则表明这是一个寄存器。于是我们在SR1之后输出000，再用TranslateOprand函数输出其标号；若不为"R"，则表明这是一个立即数，于是，我们先输出一个"1"作为标志位，然后才翻译出5位的立即数。注意这里传入TranslateOprand函数的Opcode_length为5，意在提示函数返回字符串的长度。故对于那些不同的返回字符串长度的要求，我们都可以通过改变这一参数来满足。

  至此，对ADD指令的处理就算结束了。

```
646             switch (command_tag) {
647             case 0:
648                 // "ADD"
649                 result_line += "0001";
650                 if (parameter_list_size != 3) {
651                     // @ Error parameter numbers
652                     return -30;
653                 }
654                 result_line += TranslateOprand(current_address, parameter_list[0]);
655                 result_line += TranslateOprand(current_address, parameter_list[1]);
656                 if (parameter_list[2][0] == 'R') {
657                     // The third parameter is a register
658                     result_line += "000";
659                     result_line += TranslateOprand(current_address, parameter_list[2]);
660                 } else {
661                     // The third parameter is an immediate number
662                     result_line += "1";
663                     // std::cout << "hi " << parameter_list[2] << std::endl;
664                     result_line += TranslateOprand(current_address, parameter_list[2], 5);
665                 }
666                 break;
```

- BRZ指令

  进入该进程后，我们首先在result_line中生成一个"0000010"。由于BR指令的语法为：

  BRNZP LABEL

  编码后为：

  $0000 + \mathrm{nzp} + \mathrm{PCoffset9}$

因此我们判断收录的参数是否只有一个，若不是就直接报错。然后只需要对收录的参数（即跳转地址的标签）引用一次TranslateOprand函数即可完成翻译。对于BR指令族的其他指令，我们只需要依据其检测的条件码改变Opcode对应的字符串，而后续操作均一致。值得一提的是，据LC3汇编语法规定，不加任何条件码的指令（即BR LABEL）表示无条件跳转，因此Opcode对应的字符串为“0000111”。

```
708             case 4:
709                 // "BRZ"
710                 // TO BE DONE
711                 result_line += "0000010";
712                 if (parameter_list_size != 1) {
713                     // @ Error parameter numbers
714                     return -30;
715                 }
716                 result_line += TranslateOprand(current_address, parameter_list[0], 9);
717                 break;
```

- RTI指令

  由于RTI指令翻译后一定是1000000000000000，因此我们只需要再检查一下保存的参数是否为0个即可。

- TRAP指令

  该指令的语法为：

  TRAP trapvector8

  翻译后为：

  11110000 + trapvect8

  因此在确定参数只有一个后，我们对参数引用TranslateOprand函数即可将trapvector翻译出来。

- trap routine

  由于trap routine中每一句都有对应的翻译，因此我们只需要输出对应的字符串即可。

```
921             } else {
922                 // This is a trap routine
923                 command_tag = IsLC3TrapRoutine(word);
924                 switch (command_tag) {
925                     case 0:
926                     // x20
927                     result_line += "1111000000100000";
928                     break;
929                     case 1:
930                     // x21
931                     result_line += "1111000000100001";
932                     break;
933                     case 2:
934                     // x22
935                     result_line += "1111000000100010";
936                     break;
937                     case 3:
938                     // x23
939                     result_line += "1111000000100011";
940                     break;
941                     case 4:
942                     // x24
943                     result_line += "1111000000100100";
944                     break;
945                     case 5:
946                     // x25
947                     result_line += "1111000000100101";
948                     break;
949                     default:
950                     // @ Error Unknown command
951                     return -50;
952                 }
```

- 其余情况

  表明出现了未知的指令，直接报错即可。

- 总结

  在编写每一个函数的翻译过程时，我们虽要做到“因地制宜”，但基本思路如下：

  1. 直接输出Opecode对应的字符串；
  2. 根据该指令的语法，判断参数是否正确；
  3. 依次翻译每一个参数：对于寄存器，我们直接引用TranslateOprand函数；对于标签或立即数，我们需要在引用TranslateOprand函数时指定返回字符串的长度，从而同语法相匹配。
  4. 最后，由于这是Switch中的子模块，所以记得Break;

由于之前的结果均保存在result_line中，因此，若是在十六进制模式下，我们只需要再引用一次ConvertBin2Hex函数将result_line转化成十六进制形式。做完以上工作后，我们也就完成了对一行的翻译，终于可以将result_line和换行符输出到Output_file中，并开始下一行的翻译工作了。

当所有行都成功翻译后，程序结束，return 0。

至此，对于汇编历程的介绍告一段落。在其中，我还融入了对大部分要求完成的程序的介绍，虽难以做到面面俱到，但我力求通俗且涵盖了所有*Tricky*之处。这些基本上都是$Assembler.cpp$文件中的代码（$Trim$位于$Assembler.h$中），但由于$main.cpp$和$Assembler.h$基本上都是一些类的定义和主界面的实现，且没有别的要求完成的代码部分，限于篇幅，恕不介绍。稍微不负责任地说，上文中那些未提及代码的"OOP函数"，基本上都是在这些两个文件里定义的，但读者应该能够通过猜字面意思或结合语境知道它们的意思和大概的实现方式。

实验心得

同样限于篇幅，这里我就试将万千感慨融进寥寥几行之间。

1. 由于我之前未接触过C++，因此最初对于完成附加实验有所抵触。但出于对自身能力提升的希冀~~(给分太香了)~~，我还是下决心着手研究LabA。从结果来看，这个入门作业还算是不负有心人。
2. 其实上文中的代码，大部分都不是一蹴而就，而是在后续测试中经多次修改后完成的。这次实验，需要在后续测试中下一番功夫。因为无论测试时输入的是正确代码还是错误代码，都能帮助你反思程序中的漏洞和查错机制是否完善，而这些在首次撰写时往往会被人忽略。可以说，这个程序虽然通过了所给代码的测试，但一定还在存在着某些问题，确切地说是能够容忍某些有问题的表达而不出错，因此，还不足以做到和LC3Tools中的汇编器一模一样。不过，我可以断言，对于正确的代码，这个程序**没有理由**会出错。
3. 虽然我从最初的面对代码一脸茫然，到后来的渐渐摸清思路，再到最后能够填补空白甚至依据自己的理解修改部分原先所给代码，但是我明白，这终究是**站在巨人的肩膀上**，再次重申，我十分钦佩原作者构建的整体思路，同时感谢他为我们创建好的类和函数，当然，还有恰到好处的提示。
4. 总而言之，这次实验我既看见了差距，也看到了自己的成长。最后还是要**感谢这门课，感谢苗老师和所有的助教，愿新年快乐，平安顺意。**

以上。
