lab7

> *本实验在Linux环境（Ubuntu 22.04）下完成*

实验目的

使用C++编写一个简易的LC-3汇编器，加深对2-Pass汇编过程的理解。

实验方法

Pass 1：建立符号表

需维护的变量

使用C++中的`unordered_map`容器来维护符号表，键为`string`类型的符号名，值为符号对应的地址，平均情况下可以实现$O(1)$时间的插入和查找：

```
std::unordered_map<std::string, int> symbol_table;
```

同样使用`unordered_map`容器维护指令集合，键为`string`类型的指令名，值为`string`类型的二进制`opcode`字符串（以及一些已知的前缀）：

```
const std::unordered_map<std::string, std::string> commands = {
    {"ADD", "0001"},
    {"AND", "0101"},
    {"BR", "0000111"},
    {"BRNZP", "0000111"},
    {"BRN", "0000100"},
    {"BRZ", "0000010"},
    {"BRP", "0000001"},
    {"BRNZ", "0000110"},
    {"BRNP", "0000101"},
    {"BRZP", "0000011"},
    {"JMP", "1100000"},
    {"JSR", "01001"},
    {"JSRR", "0100000"},
    {"LD", "0010"},
    {"LDI", "1010"},
    {"LDR", "0110"},
    {"LEA", "1110"},
    {"NOT", "1001"},
    {"RET", "1100000111000000"},
    {"RTI", "1000000000000000"},
    {"ST", "0011"},
    {"STI", "1011"},
```

```
    {"STR", "0111"},
    {"TRAP", "11110000"},
    {".ORIG", ""},
    {".END", ""},
    {".FILL", ""},
    {".BLKW", ""},
    {".STRINGZ", ""}};
```

此外，为了计算每个符号对应的地址，以及方便后序PC的计算，还需要记录程序起始地址以及每行代码对应的内存地址：

```
int orig_addr;  // 程序起始地址
std::vector<int> line_addr;  // 每行对应的地址
```

字符串分割

函数`assemble`的参数是一个字符串数组，每一行对应一个字符串。由于行内不同token之间以空格隔开，因此考虑先按空格将字符串分割成若干字符串组成的数组。使用C++标准库中的`istringstream`创建一个字符串流，依次读出`token`并加入数组：

```
std::istringstream iss(input_lines[i]);
std::string token;
std::vector<std::string> line;
while (std::getline(iss, token, ' ')) {
    line.push_back(token);
}
```

此外还维护一个数组`file`存储每行分割后的字符串数组，这样在Pass 2就无需重复处理：

```
std::vector<std::vector<std::string>> file;
...
file.push_back(line);
```

每行对应地址的计算

维护一个全局变量`step`，表示下一行与当前行的地址间隔，由此可地推出`line_addr`数组的值：

```
if (i == 1) {
    line_addr[i] = orig_addr;
} else if (i > 1) {
    line_addr[i] = line_addr[i - 1] + step;
}
```

符号处理

每行开头的token要么是指令，要么是伪指令，要么是符号，那么遇到既非指令也不是伪指令就是符号。使用`unordered_map`的`find`方法在指令集中查找：

```cpp
bool is_command(const std::string& str) {
    // @param str: 待判断的字符串
    // @return: str是否为指令(包括支持的伪指令)
    return commands.find(str) != commands.end();
}
```

如果某行开头的token（即`line[0]`）是符号，就要将其与当前行地址的映射加入`symbol_table`中；此后这个行首的符号将没有任何作用，可以删去，从而后序处理中行首总是指令：

```cpp
// label
if (!is_command(line[0])) {                // 不是指令，则为符号
    symbol_table[line[0]] = line_addr[i];  // 存储标签和地址的映射关系
    line.erase(line.begin());              // 移除标签
}
```

不同指令对应的处理

在Pass 1中只需完成地址相关的计算，因此要特别考虑的指令只有如下4个：

1. `.ORIG`指令：由此可获取`orig_addr`的值。
2. `.STRINGZ`指令：需根据字符串（`line[1]`）的大小计算`step`。
3. `.BLKW`指令：需根据分配空间的大小（`line[1]`）计算`step`。
4. `.END`指令：Pass 1结束。

   其余指令都令`step`为1即可。

```cpp
// pseudo command: .ORIG
if (line[0] == ".ORIG") {
    orig_addr = get_immediate_value(line[1]);  // 获取程序起始地址
}
// pseudo command: .STRINGZ
else if (line[0] == ".STRINGZ") {
    line[1].erase(line[1].begin());  // 去除开头引号
    line[1].pop_back();              // 去除末尾引号
    step = line[1].size() + 1;       // 为字符串空出位置
}
// pseudo command: .BLKW
else if (line[0] == ".BLKW") {
    step = get_immediate_value(line[1]);  // 腾出地址空间
}
// pseudo command: .END
else if (line[0] == ".END") {
    break;
}
// 其余指令
else {
    step = 1;
}
```

其中`get_immediate_value`函数可以将以`#`或`x`开头的立即数字符串转化为`int`类型，可直接使用C++标准库中的`stoi`函数实现：

```
int get_immediate_value(const std::string& str) {
    // @param str: 立即数字符串，以'#'或'x'开头
    // @return 立即数的十进制表示
    if (str[0] == '#') {  // 十进制
        return std::stoi(str.substr(1));
    } else {  // 十六进制
        return std::stoi(str.substr(1), nullptr, 16);
    }
}
```

Pass 2：生成机器码

定义了三个函数来实现主要的三个操作：取寄存器、取立即数、取PCoffset

取寄存器

`get_register`函数接收包含寄存器信息的token作为参数，其第2个字符就是寄存器的编号，使用C++标准库中的`bitset`函数可以将其转化为3位二进制形式：

```
std::string get_register(const std::string& str) {
    // @param str: 寄存器字符串，以'R'开头
    // @return 寄存器的二进制字符串
    return std::bitset<3>(str[1] - '0').to_string();
}
```

取立即数

`get_immediate`函数接收包含立即数信息的token以及立即数位数作为参数，并调用前面实现的`get_immediate_value`函数。需要获得的立即数位数只有5、6、8、16这四种情况，因此可以实现如下：

```
std::string get_immediate(const std::string& str, int num_bits) {
    // @param str: 立即数字符串，以'#'或'x'开头
    // @param num_bits: 立即数的位数
    // @return 立即数的二进制字符串
    switch (num_bits) {
        case 5:
            return std::bitset<5>(get_immediate_value(str)).to_string();
        case 6:
            return std::bitset<6>(get_immediate_value(str)).to_string();
        case 8:
            return std::bitset<8>(get_immediate_value(str)).to_string();
        case 16:
            return std::bitset<16>(get_immediate_value(str)).to_string();
        default:
            std::cerr << "Invalid num_bits!" << std::endl;
```

```
            exit(1);
    }
}
```

取PCoffset

PCoffset可能以符号或立即数的方式呈现，故应先判断token是符号还是立即数。鉴于本实验保证输入文件一定是合法的，故可以调用`find`方法判断符号表中是否含有该token，是则为符号，否则为立即数。如果是立即数，直接调用`get_immediate_value`获取其值即可；如果是符号，需要从符号表中获取其地址，并与`curr_pc`相减得到PCoffset：

```
std::string get_pc_offset(const std::string& str, const int curr_pc, const int
num_bits) {
    // @param str: PCoffset的汇编代码
    // @param curr_pc: 当前PC值
    // @num_bits: PCoffset的位数
    // @return PCoffset的二进制字符串

    int offset;                                          // 偏移量
    if (symbol_table.find(str) == symbol_table.end()) {  // PCoffset是立即数
        offset = get_immediate_value(str);
    } else {  // PCoffset是标签
        offset = symbol_table[str] - curr_pc;
    }
    // 根据PCoffset的位数返回结果
    switch (num_bits) {
        case 9:
            return std::bitset<9>(offset).to_string();
        case 11:
            return std::bitset<11>(offset).to_string();
        default:
            std::cerr << "Invalid num_bits!" << std::endl;
            exit(1);
    }
}
```

伪指令处理

1. `.ORIG`：调用`get_immediate`函数将起始地址转化为16位二进制串即可
2. `.FILL`：其参数可能为符号也可能为立即数，首先按`get_pc_offset`函数的方法来判断。如果是符号，就从符号表中获取其地址，转化为二进制串；如果是立即数，调用`get_immediate`函数即可
3. `.BLKW`：填充16位0直至下一行的起始地址
4. `.STRINGZ`：将每个字符对应的ASCII码利用`bitset`函数转化为16位二进制，最后补一个0即可

代码如下：

```
if (line[0] == ".ORIG") {
    output_lines.emplace_back(get_immediate(line[1], 16));
}
```

```
// .FILL
else if (line[0] == ".FILL") {
    if (symbol_table.find(line[1]) == symbol_table.end()) {
        output_lines.emplace_back(get_immediate(line[1], 16));
    } else {
        output_lines.emplace_back(std::bitset<16>(symbol_table[line[1]]).to_string());
    }
}
// .STRINGZ
else if (line[0] == ".STRINGZ") {
    for (auto c : line[1]) {
        output_lines.emplace_back(std::bitset<16>(c).to_string());
    }
    output_lines.emplace_back(16, '0');
}
// .BLKW
else if (line[0] == ".BLKW") {
    for (int j = line_addr[i]; j < line_addr[i + 1]; j++) {
        output_lines.emplace_back(16, '0');
    }
}
```

指令处理

先根据指令名获取`opcode`（以及一些已知的前缀）：

```
std::string machine_code = commands.at(line[0]);
```

然后根据每种指令的格式，调用上面定义的三个函数`get_register`，`get_immediate`，`get_pc_offset`即可。一些格式相同的指令可以放在一起处理，下面以`ADD`和`AND`指令为例展示代码：

```
// ADD, AND
if (line[0] == "ADD" || line[0] == "AND") {
    machine_code += get_register(line[1]);  // DR
    machine_code += get_register(line[2]);  // SR1
    // 第三个参数是寄存器
    if (line[3][0] == 'R') {
        machine_code += "000";
        machine_code += get_register(line[3]);  // SR2
    }
    // 第三个参数是立即数
    else {
        machine_code += "1";
        machine_code += get_immediate(line[3], 5);  // imm5
    }
}
```

其余指令处理都很类似，不再赘述。

程序测试结果

样例测试

使用`g++`编译：

```
g++ assembler.cpp -o assembler -std=c++20
```

对提供的样例输入进行测试并使用`diff`命令比较其与样例输出的结果：

```
./assembler test_in.asm out.txt && diff out.txt test_out.txt
```

结果为：

```
8c8
< 0000000000000000
---
> 0000000000000000
\ No newline at end of file
```

说明程序的输出只不过在文件末尾比样例输出多了一个换行，其他完全一致。

自编样例测试

我编写了一个涵盖所有指令情况且具有很多label的`test.asm`文件（并没有任何实际意义，毕竟本实验只需要将其正确地转化为机器码即可）：

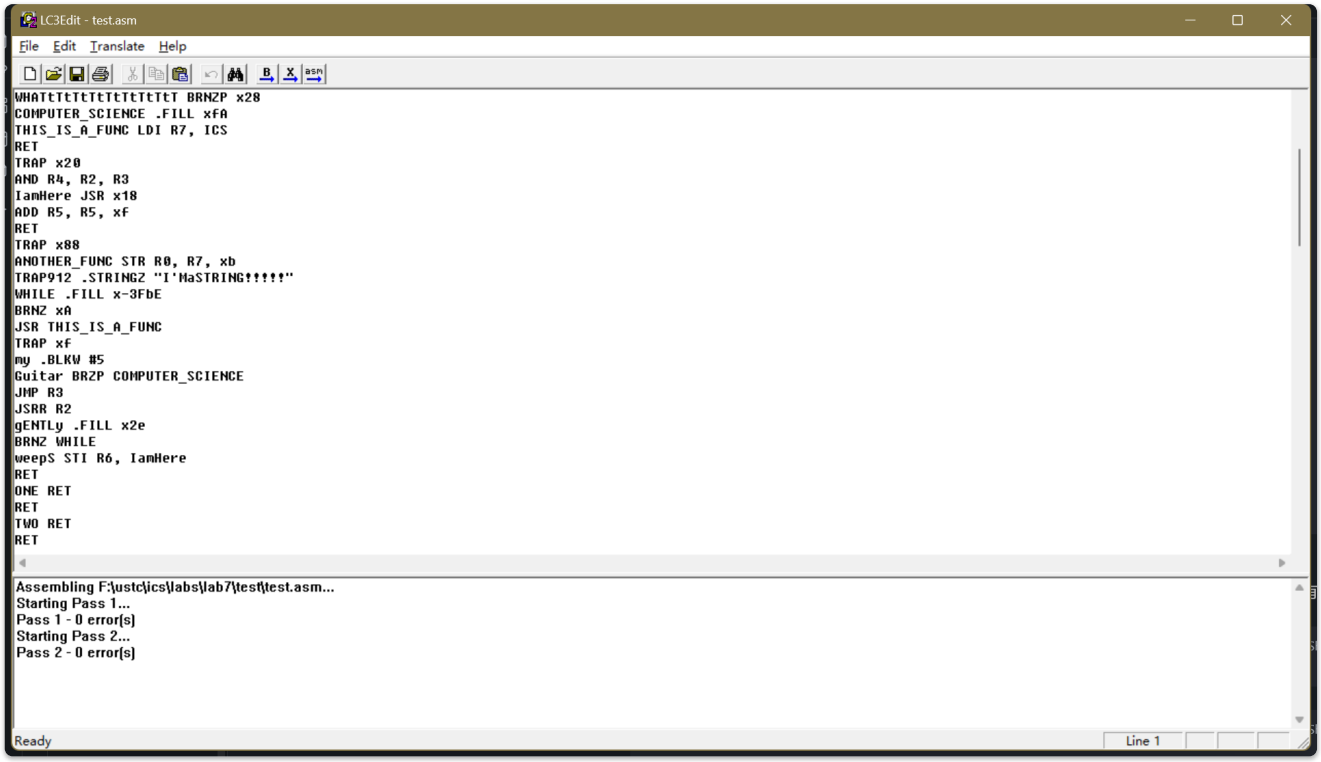
```
.ORIG #189
SOMEthing .FILL x3000
BR SOMEthing
INtheWAY .STRINGZ "sheMoves....."
BRNZP SOMEthing
BRNP INtheWAY
JSR INtheWAY
ICS NOT R6, R7
ADD R3, R4, R5
ADD R6, R2, #-16
WHATtTtTtTtTtTtTtTtT BRNZP x28
COMPUTER_SCIENCE .FILL xfA
NOT R4, R4
BRZP #8
THIS_IS_A_FUNC LDI R7, ICS
TRAP x20
AND R4, R2, R3
```

```
LDI R4, #-126
IamHere JSR x18
BR #-10
BRNZP #99
ADD R5, R5, xf
TRAP x88
ANOTHER_FUNC STR R0, R7, xb
AND R3, R2, #10
TRAP912 .STRINGZ "I'MaSTRING!!!!!"
WHILE .FILL x-3FbE
BRNZ xA
JSR THIS_IS_A_FUNC
TRAP xf
my .BLKW #5
Guitar BRZP COMPUTER_SCIENCE
LEA R2, GOODnight
JMP R3
JSRR R2
gENTLy .FILL ADD63687
BRNZ WHILE
BRP x66
weepS STI R6, IamHere
GOODnight RET
ONE RET
BRZ x-45
TWO RET
JMP R7
LDR R0, R2, x1C
JSRR R4
LD R1, ONE
LDI R7, TWO
ST R1, x8f
AND R1, R0, x0
BRZ THREE
LD R2, WHATtTtTtTtTtTtTtTtT
BR x-fB
ADD63687 .STRINGZ "8Zb.,-=1W*qfR$rjS0p5w2vu+%:#M[eI<"
BRZP xac
LDR R7, R7, #-4
STI R7, x-9f
STI R0 SOMEthing
BRP SOMEthing
TRAP xfF
loVE BRP #120
.STRINGZ "I"
AND R5, R7, x-d
```

```
.STRINGZ "look"
.STRINGZ "at"
BRN #-2
.STRINGZ "the"
THREE .STRINGZ "world"
BRP #-9
.STRINGZ "and"
BRN x-Fa
.STRINGZ "I"
SMILE .STRINGZ "notice"
BR sOMANY7
ST R7, #-234
.STRINGZ "1111111111111111111111111111111222222222222222222222234"
BRNP sOMANY7
ADD R2, R6, x-F
BRZP donTknowWHY
I .FILL x-3
BRNP xb1
donTknowWHY BRN ADD63687
NOBODY BRZ Guitar
tolDyou BRP gENTLy
BRZ SMILE
howTO BRN x99
unfolDyour .STRINGZ "StillMyGuitarGentlyWeeps~"
TRAP12 .STRINGZ "'/?9dJK3n`~|;hellO"
TRAP #37
FINALLYend .STRINGZ ""
ST R1, TWO
ST R7, x-2B
JSR x128h
LDI R2, xF7FG
STR R6, R2, x-1f
BRN xaa
ST R7, TWO
.STRINGZ "it's"
.STRINGZ "turning"
BRZ #100
LD R2, x22
LDI R5, x-3c
JSR #-100
JSR unfolDyour
LEA R2, x-0d
BRNZP NOBODY
xF7FG ADD R0, R0, R0
.FILL xF7FG
LEA R2, x-bF
```

```
LD R5, #-91
TRAP #28
BRNZP x-7C
x128h JSRR R0
BRNP loVE
LEA R6, I
BR howTO
BRP weepS
RTI
ST R2, gENTLy
.FILL #-1111
.BLKW #20
LEA R5, #55
sOMANY7 .BLKW x1f
TRAP x00
whywhywhywhywhy .BLKW xaf
yesyes .STRINGZ "hello_world__________!#$%^&*(()99999"
LEA R7 whywhywhywhywhy
.FILL #999
.FILL Sleep
BRNZP #-90
.FILL x-98
```

然后我使用LC3Edit工具对其汇编，得到`test.bin`文件：



因为是在Windows下操作，所以我还使用`dos2unix`命令将这些文件转化为Linux格式下的文件，确保行尾没有`'\r'`符：

```
dos2unix test/test.asm test/test.bin
```

然后我使用自己的`assembler`对`test.asm`进行汇编：

```
./assembler test/test.asm test/out.txt
```

最后使用`diff`命令比较两者：

```
diff test/out.txt test/test.bin
```

没有输出，说明两个文件完全相同。

实验中遇到的困难与解决方案

由于本实验对输入文件做了很严格的限制，所以实现起来其实并不困难，于是我主要的精力放在如何简化代码上。在Pass2中，一开始的想法自然是用枚举每一种指令，但是实际上有些指令的格式是完全相同的，比如`ADD`和`AND`，比如`BR`的那若干条指令，比如`LD`和`LDI`和`ST`和`STI`，因此可以把它们合并在一起处理。此外，既然`is_command`函数需要一个存储所有指令的容器，那不妨将这些指令已知的二进制前缀也一并存进去，这样在Pass2一开始就可以获取这些前缀，像`RTI`和`RET`这样完全确定的指令就根本无需处理了。
在进制转化方面，查询得知C++标准库提供了非常有效的函数，因此写起来十分轻松。

实验总结

通过本实验，我通过亲手实现一个简易LC-3汇编器，深入理解了汇编器的工作原理，巩固了理论知识。