LabA&S

- 姓名：林佳胜；学号：PB24511997

> 源文件的注释还算详细。
>
> 因为本人未写完（准确来说考虑到时间所以没写）模拟器，所以仅针对汇编器写的实验报告。
>
> 模拟器的坑可能得考完期末考之后再填了。

一. 实验目的

- 本实验旨在通过编写一个 LC-3汇编器（Assembler），深入理解汇编语言与机器语言之间的对应关系，以及汇编器将汇编代码转换为机器代码的工作原理。具体目标包括：
  1. **掌握 ISA**: 深入理解 LC-3 处理器的指令集架构（ISA），包括操作码（OpCode）、寻址模式和指令编码格式。
  2. **理解汇编原理**: 实践汇编器的 "两遍扫描"（Two-Pass）算法机制，理解为什么需要两次扫描来解决前向引用问题。
  3. **符号管理**: 实现符号表（Symbol Table）的构建与管理，正确处理标签（Label）到内存地址的映射。
  4. **自动化测试**: 学会构建 Python 脚本进行自动化测试，提升开发效率和代码质量。

二. 实验过程

本实验使用 C++ 开发汇编器核心逻辑，使用 Python 编写自动化测试脚本。开发流程如下：

> 实验环境：
>
> - `cpp`：见 `CMakeLists.txt` 的要求，由于已经静态链接了可执行文件，如果不再次编译，可认为无环境要求；
> - `py` ：由于脚本使用的都是系统库（不需要额外 install），可以不额外配置环境（尽量是 3.9 以上吧）。

2.1 系统架构设计

汇编器采用经典的 **Two-Pass（双遍扫描）** 架构：

1. **Pass One (第一遍扫描)**:

- 逐行读取汇编源文件，并将原始行内容缓存到内存容器（`source_lines_`）中。
- 识别标签（Labels），并将其与当前的内存地址（`Location Counter`, 即 `lc_`）绑定存入 `symbol_table_` 中。
- 计算每条指令和伪指令占用的内存大小，进而更新 `lc_`。
  - 检查 `.ORIG` 和 `.END` 等段落结构的合法性（参照 `LC3 Tool`，允许多组 `.ORIG 和 .END 代码块`）。

2. **Pass Two (第二遍扫描)**:
   - 遍历内存中的源码缓存（减少 I/O 操作的开销）。
   - 利用 **Pass One** 生成的符号表，将 **Label** 替换为具体的地址偏移或绝对地址。
   - 处理 `.FILL` 等伪指令。
   - 将汇编指令翻译为 16 位的二进制机器码。
   - 输出二进制目标文件 (`.bin`)。

2.2 程序框架

本项目的代码组织结构如下：

```
LabA_S/
├── CMakeLists.txt              # CMake 构建脚本 （配置静态链接）
├── test_runner.py              # Python 自动化测试脚本
├── Inc/                        # [Header Files]
|   ├── types.h                 # 通用的数据类型定义
|   ├── utils.h                 # 通用的工具函数实现（Header-Only 模式）
|   ├── assembler.h             # 汇编器类声明
|   └── lc3_vm.h                # （预留）模拟器类声明
├── Src/                        # [Source Files]
|   ├── assembler.cpp           # 汇编器类实现
|   ├── main.cpp                # 程序入口（允许接收命令行参数）
|   └── lc3_vm.cpp              # （预留）模拟器类实现
└── test_case/                  # [Test set]
    ├── asm/                    # 测试输入（.asm）
    ├── bin/                    # 预期输出（.bin）
    ├── symbol/                 # 预期输出（.txt）
    └── output/                 # 实际输出（.py 生成的）
        ├── bin/                # 生成的机器码
        └── symbol/             # 生成的符号表
```

2.3 遇到的问题与解决

> 有些问题可能忘了，但应该不重要。

2.3 遇到的问题与解决方案

1. **预处理策略的修正 (.STRINGZ)**
   - **问题**：最初的预处理逻辑是将整行代码统一转为大写并去除所有空格。这可能导致 `.STRINGZ` 伪指令后的字符串内容被错误地修改（例如字符串中的空格被删、小写字母被转大写）。
   - **解决**：调整预处理逻辑，对引号内的字符串内容不做任何处理，保留其原始格式。
2. **转义字符的解析**
   - **问题**：`.STRINGZ` 支持 `\n`，`\t`，`\\` 等转义字符，简单的字符串拷贝无法正确处理这些特殊符号。
   - **解决**：在 `PassTwo` 中处理 `.STRINGZ` 时，逐字符扫描字符串。当检测到反斜杠 `\` 时，根据后续字符进行查表（此处是用的 `switch-case` 实现，没用哈希表）映射（如 `n` -> `\n`），将其转换为对应的 ASCII 码写入内存。
3. **异常处理机制的设计**
   - **问题**：在实现过程中，对于“何时抛出异常”与“何时捕获异常”的界限较为模糊，导致早期代码结构混乱。
   - **解决**：确立了分层异常处理原则——底层工具函数（如 `ParseReg`）发现格式错误直接 `throw ErrorCode`；顶层的 `PassOne/PassTwo` 循环负责 `catch` 并记录行号；在最终的 `Run` 中输出错误。
4. **内存地址合法性与越界检查**
   - **问题**：原先的代码处理数字时，会将数字默认截取为 `uint16_t` 或 `int16_t` 的形式，导致无法判断地址是否合法。
   - **解决**：先输出 `int32_t` 的数据，再进行范围检查，保证原始错误输入不被误认为正确输入。
5. `.FILL` **操作数的多样性**

- **问题**：忘记了 `.FILL` 的操作数既可以是立即数（`x3000`），也可以是标签（`START`）。代码初始版本只处理了数字，导致遇到 Label 时报错。
- **解决**：引入 `try-catch` 尝试机制。优先尝试解析为立即数，若抛出异常，则捕获并在符号表中查找 Label 地址。

6. **环境兼容性 (MinGW DLL 缺失)**
   - **问题**：`CLion` 构建的 `.exe` 依赖动态分发的 `libgcc` 等库，脱离开发环境直接运行（如被 Python 脚本调用）时会因找不到 DLL 而崩溃 (Exit Code 0xC0000005)。
   - **解决**：在 `CMakeLists.txt` 中添加 `-static` 链接选项，强制静态链接依赖库，生成独立的可执行文件。
7. **多代码段 (.ORIG) 支持**
   - **问题**：本汇编器允许源文件中存在多个 `.ORIG ... .END` 块，且往往 `.ORIG` 不是按地址顺序写的，直接写文件可能会产生覆盖或空洞问题。
   - **解决**：替换了原先直接输出到二进制文件的逻辑，改为先输出到 `std::map<uint16_t, uint16_t>` 进行稀疏存储（这会自动按地址排序代码段），最后输出时再统一对未使用的地址空间补 0。
8. **自动化测试的比对挑战**
   - **问题**：测试用例的预期输出为文本文件，无法直接跟二进制文件比对。
   - **解决**：在 Python 测试脚本中增加函数 `compare_bin_with_text`，将二进制数据转换为 16 位宽的 "01" 字符串列表，再与参考答案进行逐行比对。
9. **符号表的有序输出**
   - **问题**：原先使用的 Map 按 Label 字典序存储，导致输出的符号表是按字母排序的，而实验预期是按内存地址升序排列。
   - **解决**：在输出阶段，将符号表数据复制到 `std::vector`，并配合 Lambda 表达式使用 `std::sort` 按地址值进行升序重排。

三. 实验结果

> OUTPUT_DIR = r"test_case\output"
>
> 可以直接在该目录下看输出（包括二进制输出（在 `bin`）和符号表输出（在 `symbol`））。
>
> 不过我的脚本有检测机制，故在 3.1 中展示处理信息， 3.2 中举一个生成的符号表例子。
>
> > 不过，好像没有文件让我验证我的汇编器是否能正确识别错误；
> >
> > 以及，我没写符号表比对的程序，比对仅针对输出的二进制文件，符号表比对是肉眼比对。

3.1 自动化测试

使用 `test_runner.py` 对 `asm` 目录下的所有测试用例进行了验证。

```
 Admin ❯❯ python -u "e:\CS\ics\lab\LabA_S\test_runner.py"
Scanning test cases in test_case\asm...

[*] Testing fibonacci.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.
[*] Testing multiplication.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.
[*] Testing recursion.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.
[*] Testing stack_operations.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.
[*] Testing string_length.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.
```

```
[*] Testing test_add.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.
[*] Testing test_and_not.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.
[*] Testing test_branch.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.
[*] Testing test_jump.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.
[*] Testing test_load_store.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.
[*] Testing test_subroutine.asm ...
    [PASS] (Assembled)     [MATCH] Binary matches expected output.

Summary: 11/11 Passed.
```

3.2 符号表生成

仅以 `fibonacci.asm_symtable.txt` 为例：

```
-- -Symbol Table-- -
LOOP : x300F
DONE : x3014
FAIL : x3019
PASS_MSG : x301C
FAIL_MSG : x3033
--------------------
```

四. 实验总结

> 本次实验算是一次对 `cpp` 的综合运用。
>
> 在本次实验中，我学习了如何组织比较好的项目架构（本次的项目架构我挺喜欢的）：
>
> - 学会了`common` 模块的分离（包括 `types.h` 和 `utils.h`）；
> - 明白了将常用工具函数独立文件（`utils.h`），及其 **Head-Only** 模式。
>
> 以及学会了如何让 **可执行文件** 与 **脚本** 协同工作。
>
> 算是本学期写的收获最大的 Lab 了。

通过本次实验，成功实现了一个功能完备的 LC-3 汇编器，进一步理解了 LC-3 的系统架构。