LabA: 汇编器

学号：PB24061302
姓名：赵国华
院系：人工智能与数据科学学院

1 实验目的

- 掌握汇编原理：实现双遍扫描，理解符号表与指令编码
- 理解指令周期：实现指令集模拟器，深入理解取值、译码、执行、访存、写回的全过程
- 系统集成能力：通过脚本或主程序将各个组件串连起来，实现完成的本地测试程序

2 实验思路

1. 设计汇编器的整体架构，确定各个模块的功能和接口。
2. 实现词法分析器，能够识别汇编语言中的各种符号和指令。
3. 实现语法分析器，能够根据汇编语言的语法规则生成抽象语法树。
4. 实现符号表管理模块，能够处理标签和变量的定义与引用。
5. 实现指令编码模块，将汇编指令转换为机器码。

根据以上分析，需要进行模块的拆分：

- 输入输出模块：负责读取汇编代码文件，并将其存储为适合处理的数据结构。将生成的机器码写入输出文件。
- 预处理模块：负责去除注释和空白行，处理大小写不敏感的字符。
- 符号表模块：负责管理标签和变量的定义与引用，构建符号表。
- 指令集模块：负责汇编指令的格式、操作数类型和位字段布局。
- 错误处理模块：负责检测和报告汇编过程中出现的错误。
- 主控制模块：负责协调各个模块的工作流程。

3 实验过程

3.1 编程语言选择与实验环境搭建

- 选择 c++ 作为编程语言，更接近底层，便于加深对汇编器工作原理的理解。
- 选择 VScode+WSL+CMake+Clang 作为实验环境，便于跨平台开发与调试。
- 使用 Git 进行版本控制，便于代码管理与协作开发。

3.2 汇编器设计与实现

3.2.1 Utils 类：文件输入与输出

- 功能：处理文件的读写操作
- public 静态成员函数：
  - std::string readFile(const std::string& filePath)：以二进制模式读取文件内容并返回为字符串
  - bool writeFile(const std::string& filePath, const std::string& content)：将内容写入指定文件

3.2.2 SymbolTable 类：符号表管理

- 功能：管理标签和变量的地址映射
- public 成员函数：
  - void addSymbol(const std::string& name, int address)：添加符号及其地址
  - int getAddress(const std::string& name) const：获取符号的地址
  - bool contains(const std::string& name) const：检查符号是否存在
  - unordered_map<std::string, int> getTable() const：获取完整的符号表

3.2.3 InstructionSet 类：指令集定义与管理

- 功能：定义 LC-3 指令的格式、操作数类型和位字段布局
- 核心数据结构：
  - BitField：描述指令中字段的位置和大小（startBit 和 length）
  - OperandMode：描述特定操作数类型下的指令格式（如 ADD 的寄存器模式和立即数模式）
  - InstructionInfo：包含指令名、操作码、所有支持的操作数模式

- 支持的指令：ADD、AND、NOT、LD、ST、LDR、STR、LDI、STI、BR/BRN/BRZ/BRP/BRNZ/BRN
  JSR、JSRR、JMP、RET、TRAP、HALT、PUTS 等
- public 成员函数：
  - bool isInstruction(const std::string& word) const：判断是否为指令
  - bool isPseudoOp(const std::string& word) const：判断是否为伪指令（.ORIG、.END、.FILL、.BLKW、.STRINGZ）
  - const InstructionInfo* getInstructionInfo(const std::string& mnemonic) const：获取指令信息
  - const OperandMode* getOperandMode(...)：根据操作数类型查找对应的模式
  - const BitField* getBitField(const OperandMode* mode, const std::string& fieldName) const：获取位字段信息

3.2.4 Assembler 类：汇编器的核心

- 功能：整合预处理、第一遍扫描、第二遍扫描和代码生成
- private 成员变量：
  - std::vector<std::string> processedCode：预处理后的代码行
  - SymbolTable symbolTable：符号表，存储标签和变量的地址映射
  - InstructionSet instructionSet：指令集定义
  - std::vector<std::string> machineCodes：生成的机器码
  - int currentLineNum：用于错误报告的当前行号
  - int origAddress：.ORIG 指令指定的起始地址
- private 成员函数：
  - void preprocess(const std::string& sourceCode)：预处理源代码
    - 删除注释（分号后的内容）
    - 删除前导和尾部空白
    - 将所有字符转换为大写（除了.STRINGZ 中的字符串）
    - 将逗号转换为空格
  - void firstPass()：第一遍扫描，构建符号表
    - 遍历所有行，识别标签定义
    - 计算每条指令的地址（考虑.STRINGZ 和.BLKW 占用的空间）
    - 处理.STRINGZ 时正确计算转义序列的实际字符数
    - 处理.BLKW 时累加块的大小

  - void secondPass()：第二遍扫描，生成机器码
    * 处理指令并生成对应的机器码
    * 识别操作数类型（寄存器、立即数、偏移量）
    * 进行指令编码，包括：
      · 设置操作码
      · 处理立即数标志位（ADD/AND 的位 5）
      · 处理分支条件位（BR 指令的位 9-11）
      · 处理特殊标志位（JSR 的位 11）
      · 处理陷阱向量（TRAP 相关指令）
      · 处理 RET 为特殊的 JMP R7
    * 处理伪指令（.ORIG、.FILL、.BLKW、.STRINGZ、.END）
    * 实现转义序列处理（\n、\t 等）
    * 完善的错误检测和异常抛出
  - std::string generateMachineCodeOutput()：生成输出格式的机器码

- public 成员函数：
  - void assemble(std::string inputPath, std::string outputPath)：主汇编接口
    * 读取输入文件
    * 调用 preprocess、firstPass、secondPass
    * 将机器码写入输出文件
    * 异常处理和错误报告

3.2.5 AssemblerException 类：异常处理

- 功能：统一的异常处理机制，包含 8 种错误代码：
  - FILE_READ_ERROR (1)：文件读取失败
  - FILE_WRITE_ERROR (2)：文件写入失败
  - SYNTAX_ERROR (3)：语法错误（如缺少操作数）
  - SEMANTIC_ERROR (4)：语义错误（如未定义的标签）
  - INVALID_OPERAND (5)：无效操作数（如寄存器范围超出）
  - INVALID_INSTRUCTION (6)：未知指令
  - SYMBOL_ERROR (7)：符号表错误（如重复定义）
  - VALUE_OUT_OF_RANGE (8)：值超出范围（如立即数过大）
- 特点：包含行号信息，便于定位错误位置

4 实验结果

4.1 实现的功能特性

- 完整的 LC-3 汇编器实现，支持 15+ 条指令
- 双遍扫描架构：第一遍构建符号表，第二遍生成机器码
- 多模式指令支持：如 ADD/AND 的寄存器和立即数模式
- 完善的伪指令处理：.ORIG、.END、.FILL、.BLKW、.STRINGZ
- 转义序列处理：支持 \n、\t 等特殊字符
- 8 种详细的错误代码和行号定位
- 特殊指令处理：BR/BRN 等分支指令的条件位、RET 作为 JMP R7 的转换
- 大小写敏感处理：保留.STRINGZ 中字符串的原始大小写

4.2 关键算法与实现细节

4.2.1 预处理阶段的大小写处理

预处理需要将指令和操作数转换为大写以便统一处理，但需要保留.STRINGZ 字符串的原始大小写。实现方式：

- 检测行中是否包含.STRINGZ
- 若有，将.STRINGZ 之前的部分转换为大写，之后的部分保留原样
- 若无，全行转换为大写

4.2.2 第一遍扫描的地址计算

不同伪指令占用的地址空间不同：

- 普通指令：1 个地址单位
- .STRINGZ：字符数 +1（null 终止符），需考虑转义序列
- .BLKW：指定的块大小
- .FILL：1 个地址单位

对于.STRINGZ 的长度计算，需要逐个扫描字符，识别转义序列（如 \n、\t 等），每个转义序列只计为 1 个字符。

4.2.3 第二遍扫描的机器码生成

关键要点：

- 操作数识别：根据首字符判断类型（R 开头为寄存器，# 或 X 开头为立即数，其他为标签）
- 立即数标志位：ADD 和 AND 指令若第三个操作数为立即数，则设置位 5 为 1
- 分支条件位：BR 指令根据指令名确定位 9-11（NZP 位）：
  - BRn：100、BRz：010、BRp：001、BRNZ：110 等
  - BR（无条件）：111
- 陷阱向量：TRAP 系列指令设置对应的向量号（0x20-0x25）
- RET 特殊处理：转换为固定的机器码 0x1C00（JMP R7）
- 转义序列处理：.STRINGZ 中的 \n 等转换为对应的 ASCII 码

4.3 测试结果

使用提供的 compare_results.py 脚本对比输出与标准答案：

- 测试用例覆盖：fibonacci、加法、与、非、分支、跳转、子程序、栈操作、字符串处理等
- 验证内容：指令编码、地址计算、符号表、特殊指令处理
- 测试工具：Python 脚本自动化对比，显示差异详情和十六进制差值

5 实验总结

5.1 关键问题与解决方案

5.1.1 问题 1：大小写敏感性的处理

汇编指令应该大小写不敏感，但字符串内容需要保留原始大小写。

- 问题：直接转换整行为大写会破坏字符串的大小写
- 解决：在 preprocess 函数中检测.STRINGZ，分别处理指令部分和字符串部分

5.1.2 问题 2：立即数标志位的编码

ADD 和 AND 指令支持两种模式，但立即数模式需要在位 5 设置标志位。

- 问题：简单的位字段定义无法表示固定的标志位
- 解决：在 secondPass 中，根据操作数类型检测，若为立即数则在位 5 设置 1

5.1.3 问题 3：字符串长度的精确计算

.STRINGZ 的字符数需要考虑转义序列，而不是简单地用字符串长度减去引号。

- 问题：直接计数会把转义序列（如 \n）算为 2 个字符
- 解决：逐个字符扫描，识别转义模式并跳过反斜杠

5.1.4 问题 4：RET 指令的特殊处理

RET（返回）指令实际是 JMP R7 的特殊情况。

- 问题：RET 没有操作数，无法通过通用的指令编码路径处理
- 解决：在 secondPass 中特殊检测 RET 指令，直接设置机器码为 0x1C00

5.1.5 问题 5：分支偏移量的范围检查

分支指令使用相对寻址，偏移量必须在指定范围内。

- 问题：无效的偏移量会导致寻址错误
- 解决：计算 pcOffset 后检查范围 $[-2^{n-1}, 2^{n-1}-1]$，超出范围抛出异常

5.2 设计亮点

- **模块化设计**：分离关注点，Utils 处理 I/O、SymbolTable 管理符号、InstructionSet 定义指令格式、Assembler 协调整个流程
- **异常处理**：8 种错误代码覆盖各类问题，包含行号信息便于定位和调试
- **多模式支持**：用 OperandMode 数组实现多种操作数模式，灵活应对不同指令格式
- **位字段精确描述**：BitField 结构明确定义每个字段的位置和大小，便于自动化编码
- **代码一致性**：注释全部中文，便于代码审阅和维护

5.3 学习收获与反思

通过本实验，深入理解了：

- LC-3 汇编语言的完整语法和语义
- 汇编器的设计原理和实现流程（预处理、符号表构建、代码生成）
- 机器码的二进制编码，位字段的精确定位
- 异常处理和错误恢复的重要性

- 模块化设计在复杂系统中的价值
- 测试驱动开发的必要性（自动化对比工具快速反馈）