
南開大學
Nankai University

# 南　　开　　大　　学

## 计　算　机　学　院

编译系统原理实验报告

---

# 了解你的编译器 &LLVM IR 编程 & 汇编编程

---

2311474 李泽橼

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 9 月 29 日

# 摘要

这次作业由苏威远（2311456）和李泽橏（2311474）共同完成。在第一大节中，介绍了实验目的以及问题的描述，说明了本组的分工情况，配置并说明了本人的实验环境。在任务一中，我了解到预处理阶段单个 `#include <iostream>` 指令触发头文件递归展开，导致代码从 19 行膨胀至 32,273 行，膨胀倍数达 1,698 倍。词法分析将源码转换为 61 个 token 的线性序列，语法分析构建包含 83,715 行的抽象语法树，其中源程序对应的 AST 结构仅占 86 行。中间代码生成阶段的分析表明，O2 优化通过 mem2reg 等 pass 将 14 个栈分配减少至 1 个，内存分配减少 92.9%，18 个存储指令完全消除，同时代码行数从 210 行增长至 366 行，增长 74%，主要源于 C++ 标准库函数的内联展开。代码生成阶段将 LLVM IR 转换为 RISC-V 汇编，O2 版本指令数量增加 48.2% 但文件大小减少 3.0%，store 指令减少 52.9%，反映了寄存器分配策略的根本改变。LLVM IR 编程实验验证了静态单赋值形式的中间表示机制，通过手工构建 phi 节点和基本块实现控制流管理。RISC-V 汇编优化实验显示，寄存器优化在大规模数据 ($n = 10^7$) 上性能提升 27.0%，循环展开在中等规模 ($n = 10^6$) 上最优但在大规模时出现性能反转，反映了代码膨胀对指令缓存的负面影响。

**关键字：编译流程分析，编译器原理，LLVM IR 编程，RISC-V 汇编，**

# 目录

# 一、 实验介绍

## （一） 实验目标

**第一**，通过实际操作验证编译流程的各个阶段，包括预处理、编译、汇编和链接四个主要环节，明确各阶段的输入输出关系及其在整体编译过程中的作用；**第二**，分析预处理器对源代码的宏展开、头文件包含等处理机制，探究其对程序语义的影响；**第三**，研究编译器前端的词法分析、语法分析、语义分析过程，以及后端的中间代码生成、代码优化和目标代码生成机制；**第四**，通过对汇编器和链接器工作原理的实验验证，理解机器代码生成和程序装载的技术细节。

## （二） 实验环境配置

本实验基于 Linux 环境进行，具体配置如下：

**硬件环境**：采用 Windows 11 系统下的 WSL 2（Windows Subsystem for Linux 2）虚拟化环境，提供接近原生 Linux 的性能和兼容性。

**操作系统**：Ubuntu 22.04.5 LTS，内核版本 6.6.87.2-microsoft-standard-WSL2，为实验提供稳定的 POSIX 兼容环境。

**编译工具链**：

- **主编译器**：GCC 11.4.0，用于 C/C++ 源代码的编译分析
- **RISC-V 交叉编译工具链**：riscv64-linux-gnu-gcc 11.4.0，包含对应的汇编器和链接器，支持 RISC-V 64 位指令集架构
- **二进制工具集**：GNU Binutils 2.38，包含汇编器 (as)、链接器 (ld)、目标文件分析工具等
- **模拟器**：QEMU 7.0.0 RISC-V 用户模式模拟器，用于 RISC-V 程序的执行验证

**开发环境**：Visual Studio Code 作为集成开发环境，通过 Remote-WSL 扩展实现与 WSL 环境的无缝集成，便于代码编辑、调试和实验结果观察。

**辅助工具**：包括 objdump、nm 等二进制分析工具，用于反汇编和符号表分析。

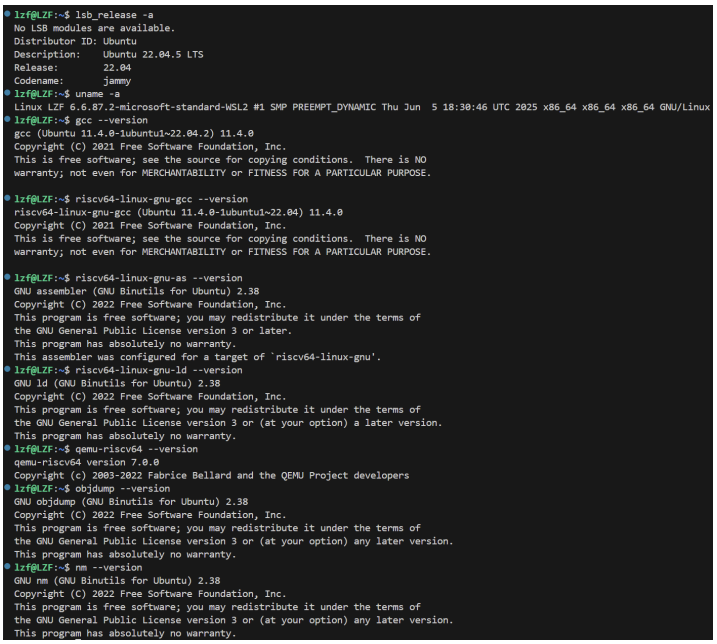



图 1: 环境配置验证

### （三） 分工

本组选题为 RISCV 指令集，小组成员为李泽椹（2311474）和苏威远（2311456）。两人分别独立完成任务一：了解编译器。苏威远（2311456）负责完成任务二：LLVM IR 的程序编写。李泽椹（2311474）负责完成任务三：RISC-V 汇编实现斐波那契数列的设计与优化。

本实验的实验代码文件均已存放到 github 仓库，您可以通过此链接来查阅我的代码文件。

## 二、 任务一：了解编译器

### （一） 预处理阶段

预处理器对源代码执行四类文本级转换操作：宏定义替换、条件编译求值、注释删除、头文件包含展开。

#### 1. 宏定义替换机制

设计包含宏定义的测试程序（Fibonacci_macro.cpp），定义四个宏：

```
#define MAX_ITER 100
#define INPUT cin
#define OUTPUT cout
#define NEWLINE endl
```

预处理后的代码显示所有宏名完全消失，被替换为对应的文本：

```
cin >> n;                  // INPUT → cin
cout << a << endl;         // OUTPUT → cout , NEWLINE → endl
while (i < n && i < 100) {   // MAX_ITER → 100
    ...
    cout << b << endl;
}
```

通过 `grep -E "MAX_ITER|INPUT|OUTPUT|NEWLINE"` 搜索预处理文件，返回 0 个匹配结果，证实宏定义在预处理阶段被彻底替换，不留任何标记。

#### 2. 条件编译的分支选择

设计包含条件编译的测试程序（Fibonacci_conditional.cpp）：

```
#define DEBUG_MODE
#ifdef DEBUG_MODE
    cout << "Debug:i=" << i << ",fib=" << b << endl;
#else
    cout << b << endl;
#endif
```

预处理后的代码仅保留 `#ifdef` 为真的分支，`#else` 分支被完全删除：

```
cout << "Debug:i=" << i << ",fib=" << b << endl;
```

预处理文件中不再包含任何 `#ifdef`、`#else`、`#endif` 指令。

### 3. 注释的完全移除

设计包含单行注释（`//`）和多行注释（`/* */`）的测试程序（Fibonacci_comment.cpp）。预处理后的代码中所有注释被删除：

```
// 预处理前：
int a, b, i, t, n;  // a和b是前两项
/* 初始化变量 */
a = 0;

// 预处理后：
int a, b, i, t, n;
a = 0;
```

通过正则表达式 `grep -E '//|\/\*|\*/'` 搜索预处理文件，匹配数量为 0，证实注释在词法分析前被预处理器完全删除。

### 4. 头文件包含的递归展开

单个 `#include <iostream>` 指令触发了大规模的头文件递归包含，导致代码行数从 19 行膨胀至 32,273 行（膨胀 1,698 倍）。

**展开层次分析** 通过提取 `# linenum filename` 标记并统计路径深度，发现头文件引用最深达 7 层：

表 1: 头文件包含层次分布

| 层级深度 | 典型文件 |
|---|---|
| 0 | 源文件（Fibonacci.cpp）、编译器内置宏 |
| 3 | 系统 C 库头文件（stdio.h, stdlib.h, errno.h） |
| 5 | C++ 标准库头文件（iostream, string, exception） |
| 6 | C++ 实现细节（bits/basic_string.h, bits/locale_facets.h） |
| 7 | 平台配置文件（c++config.h, gthr-default.h） |

**依赖链追踪** 从 `iostream` 开始的引用路径：

1. `iostream` → `ostream` / `istream`（流类定义）
2. → `ios` → `iosfwd`（前向声明）
3. → `c++config.h`（命名空间配置，如 `std::__cxx11` ABI 标签）
4. → `os_defines.h` → `features.h`（系统特性宏）
5. → `bits/c++locale.h`（区域设置）
6. → `gthr-default.h`（POSIX 线程支持）

共产生 1,428 次文件引用标记,但去重后实际引入 189 个独立头文件。其中 `bits/wordsize.h` 被引用 13 次，`c++config.h` 被引用 8 次，说明存在大量重复引用。

### 5. 源代码定位机制

预处理器通过 `# linenum filename flags` 指令维护源码位置映射：

```
# 2 "Fibonacci.cpp" 2
# 2 "Fibonacci.cpp"
using namespace std;
```

第一个 `# 2` 表示" 从包含的头文件返回"，第二个 `# 2` 标记后续代码对应源文件第 2 行。末尾的数字标志含义 [1]：

- `1`：进入新文件
- `2`：返回上一文件
- `3`：系统头文件（编译警告时忽略）
- `4`：C 语言头文件（应视为 extern "C"）

这种机制确保编译错误报告时显示原始源文件位置，而非预处理后的行号。

### 6. 不同算法实现的预处理差异

对比三个版本的预处理输出：

表 2: 三版本预处理后的规模对比

| 版本 | 原始行数 | 预处理后行数 | 头文件部分 |
|---|---|---|---|
| while 循环 | 19 | 32,273 | 32,254 |
| for 循环 | 18 | 32,271 | 32,253 |
| 递归实现 | 16 | 32,269 | 32,253 |

三个版本的头文件展开部分完全一致（前 32,250 行无差异），差异仅体现在用户代码区域：

- **while vs for**：for 版本将循环变量 `i` 的作用域限制在循环内，预处理后表现为声明位置不同
- **while vs recursive**：递归版本新增 `fib` 函数定义，预处理器未对递归调用进行任何优化或展开（递归展开属于编译器优化阶段）

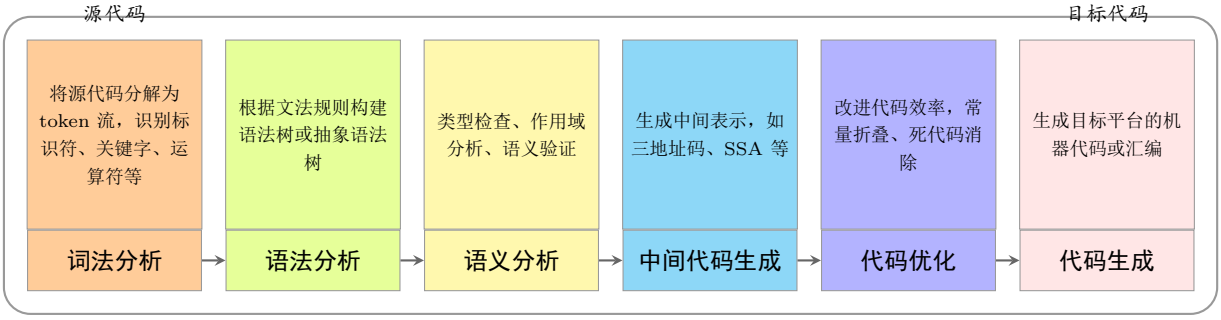
### 7. 预处理阶段的输出特征

预处理器产生的.i 文件具有以下特征：

1. **纯 C++ 代码**：不含任何以 `#` 开头的预处理指令（除位置标记）
2. **文本膨胀**：由于头文件递归展开，代码规模增长约 2000 倍
3. **语义不变**：所有转换均为文本替换，不涉及语义分析或代码优化
4. **可编译性**：输出文件可直接传递给编译器前端进行词法分析

预处理器执行纯文本处理，不理解 C++ 语法，仅进行机械的字符串替换和条件求值。宏替换会识别标识符边界，不会替换字符串字面量或已识别的 token 内部的文本。这一阶段的输出（.i 文件）是纯 C++ 代码，不含任何预处理指令，为后续的词法分析提供了" 干净" 的输入。

## （二） 编译器




### 1. 词法分析：Token 流生成

词法分析器将源代码字符流转换为 token 序列，每个 token 包含三类信息：词法单元类型、词素值、位置属性。

**Token 结构剖析** 以第 13 行的 while 语句为例，词法分析器输出：

```
while 'while'    [StartOfLine] [LeadingSpace]   Loc=<Fibonacci.cpp:13:5>
l_paren '('      [LeadingSpace] Loc=<Fibonacci.cpp:13:11>
identifier 'i'          Loc=<Fibonacci.cpp:13:12>
less '<'         [LeadingSpace] Loc=<Fibonacci.cpp:13:14>
identifier 'n'   [LeadingSpace] Loc=<Fibonacci.cpp:13:16>
r_paren ')'             Loc=<Fibonacci.cpp:13:17>
```

每个 token 的格式为：`< 类型 > '< 词素 >' [属性] Loc=< 位置 >`，其中：

- **类型**：`while`（关键字）、`identifier`（标识符）、`less`（运算符）
- **词素**：源代码中的实际文本（用单引号包围）
- **属性**：`StartOfLine`（行首 token）、`LeadingSpace`（前有空白）
- **位置**：文件名与行列号，如 `Fibonacci.cpp:13:5` 表示第 13 行第 5 列

**Token 类型统计** 对完整 token 流进行分类统计：

表 3: Fibonacci.cpp 的 Token 类型分布

| 类型 | 数量 | 示例 |
|---|---|---|
| 关键字 | 3 | `int`, `while`, `using` |
| 标识符 | 23 | `main`, `a`, `b`, `cin`, `cout` |
| 运算符 | 15 | `=`, `<`, `+`, `<<`, `>>` |
| 字面量 | 3 | `0`, `1`, `1` |
| 分隔符 | 16 | `;`, `{`, `}`, `(`, `)` |
| 其他 | 1 | `eof`（文件结束标记） |
| 总计 | 61 | |

**关键语句的 Token 序列** 手工绘制 `while (i < n)` 的 token 链：


图 2: while 循环条件的 Token 序列

**词法分析器的处理细节**

1. **空白字符处理**：空格、制表符被识别但不生成独立 token，仅作为 `[LeadingSpace]` 属性记录，用于后续格式化输出
2. **复合运算符识别**：`<<` 和 `>>` 被识别为单个 token（`lessless`、`greatergreater`），而非两个 `<` 或 `>`
3. **关键字与标识符区分**：词法分析器维护关键字表，`while` 匹配关键字表因此分类为 `while` 而非 `identifier`
4. **位置追踪精度**：每个 token 记录至字符级别的位置（列号），如 `i` 位于第 13 行第 12 列，`<` 位于第 14 列

**观察到的边界情况**

- **标识符 `cin` 与 `cout`**：被识别为 `identifier` 而非关键字，因为它们是标准库对象而非 C++ 保留字
- **命名空间声明**：`using namespace std;` 生成 5 个 token，`namespace` 是关键字，`std` 是标识符
- **文件结束标记**：最后的 `eof`token（位于 20:2）标记 token 流终止，位置指向文件末尾的虚拟位置

词法分析器输出的 token 流是一维序列，不含任何语法结构信息（如循环嵌套、表达式优先级），这些结构由后续的语法分析阶段构建。

### 2. 语法分析

语法分析器将词法分析生成的 token 流构建为抽象语法树（AST）。通过 `clang -Xclang -ast-dump` 命令生成的 AST 包含 83,715 行，其中大部分为标准库声明，源程序对应的 AST 结构位于最后 86 行。

**AST 节点统计与分布** 对 main 函数的 AST 进行节点类型统计，结果如表4所示：

**函数声明的语法树结构** main 函数的顶层结构表现为：

```
1 FunctionDecl 0x7e93fb8 <line:4:1, line:20:1> main 'int␣()'
2   -CompoundStmt 0x7ea2398 <col:12, line:20:1>|-DeclStmt (变量声明)|-BinaryOperator (初始
      化赋值)|-CXXOperatorCallExpr (输入输出)-WhileStmt (循环结构)
```

每个节点包含三类关键信息：节点类型、内存地址（用于节点间引用）、源码位置映射。

表 4: 斐波那契程序 AST 节点统计

| 节点类型 | 数量 | 对应源码元素 |
|---|---|---|
| VarDecl | 5 | 变量声明（a, b, i, t, n） |
| BinaryOperator (’=’) | 7 | 赋值语句 |
| BinaryOperator (’+’) | 2 | 加法运算（b+a, i+1） |
| DeclRefExpr | 32 | 变量引用 |
| IntegerLiteral | 3 | 整数字面量（0, 1, 1） |
| WhileStmt | 1 | while 循环结构 |
| CompoundStmt | 2 | 复合语句块 |

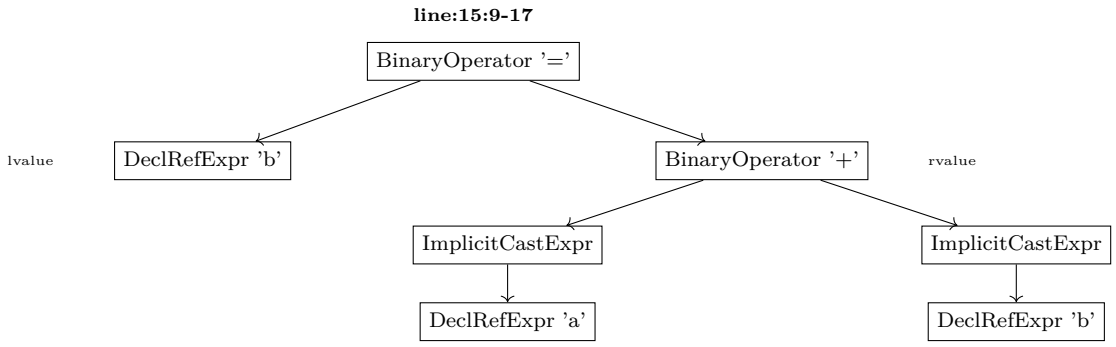



图 3: 表达式 `b = a + b` 的 AST 结构

**表达式树的层次分析：`b = a + b`** 该赋值表达式展现了 AST 的典型嵌套结构，如图3所示：

关键观察：

- **类型转换节点：**每个变量引用都包裹了 `ImplicitCastExpr` 节点，执行 `LValueToRValue` 转换，将左值转换为可参与运算的右值
- **左右值区分：**赋值运算符左侧的 `b` 保持左值属性（可被赋值），右侧表达式计算为右值
- **运算符优先级：**加法节点作为赋值节点的右子树，体现了赋值优先级低于算术运算

**While 循环的控制流结构** WhileStmt 节点包含两个直接子节点：

```
WhileStmt 0x7ea2378 <line:13:5, line:19:5>
  |-BinaryOperator '<' (条件表达式 i < n)
  | |-ImplicitCastExpr (i的类型转换)
  | -ImplicitCastExpr (n 的类型转换)-CompoundStmt (循环体)
    |-BinaryOperator '=' (t = b)
    |-BinaryOperator '=' (b = a + b)
    |-CXXOperatorCallExpr (cout << b << endl)
    |-BinaryOperator '=' (a = t)
    -BinaryOperator '=' (i = i + 1)
```

循环体内的 5 个语句按源码顺序线性排列，形成语句序列而非嵌套结构。条件表达式独立于循环体，便于编译器生成条件跳转指令。

**C++ 特有的运算符重载节点** 输入输出操作被解析为 `CXXOperatorCallExpr` 节点，反映了 C++ 将 `<<` 和 `>>` 作为重载运算符的语言特性。每个输出语句展开为多层函数调用：

表 5: 输出语句的 AST 展开深度

| | |
|---|---|
| **源码** | `cout << b << endl;` |
| **AST 深度** | 7 层 |
| **节点数** | 8 个 |
| **类型转换** | 3 次（函数指针、左值、右值） |

这种复杂性源于 C++ 的运算符重载机制和模板实例化。

### 3. 语义分析

语义分析阶段在语法分析构建的 AST 基础上进行类型检查、符号表管理、作用域解析、控制流验证和语义约束检查，确保程序符合语言的静态语义规则。

**类型系统与一致性检查** 斐波那契程序的 AST 包含 64 个`'int'` 类型标注，表明编译器为每个表达式节点都标注了类型信息。类型检查确保了操作的类型一致性：

表 6: 表达式类型检查结果

| 表达式 | 左操作数 | 右操作数 | 结果类型 |
|---|---|---|---|
| `a + b` | int | int | int |
| `i < n` | int | int | bool |
| `i + 1` | int | int (literal) | int |
| `b = a + b` | int (lvalue) | int (rvalue) | int |

**左值与右值语义分析** 编译器识别了 46 个左值标记，精确区分了可赋值的左值和仅可读取的右值：

- **左值场景**：赋值操作的左侧（如 `DeclRefExpr 'b' lvalue`），必须指向可修改的内存位置
- **右值转换**：10 处 `LValueToRValue` 转换确保算术运算获得值而非地址
- **函数退化**：10 处 `FunctionToPointerDecay` 转换处理 C++ 的函数重载解析

**符号表管理与变量生命周期** 语义分析构建了分层符号表，追踪每个变量的声明、使用和生命周期：

所有变量都被标记为 `used`，表明无死代码。变量 `b` 的高引用频率（6 次）反映了其在算法中的核心地位。

**隐式类型转换与类型安全** 语义分析器插入了 20 个 `ImplicitCastExpr` 节点，执行必要的类型转换：

```
1 double b = 1.5;
2 int t = b;  // 警告：精度损失
```

表 7: 变量语义属性分析

| 变量 | 引用次数 | 初始化 | 作用域 | used 标记 |
|---|---|---|---|---|
| a | 4 | 显式 (=0) | 函数级 | ✓ |
| b | 6 | 显式 (=1) | 函数级 | ✓ |
| i | 4 | 显式 (=1) | 函数级 | ✓ |
| t | 2 | 延迟 (=b) | 块级 | ✓ |
| n | 2 | 输入初始化 | 函数级 | ✓ |

编译器生成的诊断信息：

```
Fibonacci_type_error.cpp:11:17: warning: implicit conversion
turns floating-point number into integer: 'double' to 'int'
            ~   ^
```

诊断信息包含：精确位置（11:17）、问题类型（精度损失）、涉及类型（double→int）、视觉指示符（~ 和 ^ 标记问题位置）。

**作用域解析与名称绑定** 多层作用域测试验证了静态作用域规则：

```
int a = 100;     // 全局: 0x2b5bbb28
int main() {
    int a = 0;    // 函数: 0x2b5bbce8（遮蔽全局）
    {
        int a = 50; // 块级: 0x2b5bbda0（遮蔽外层）
    }
    cout << a << endl;  // 输出0
    cout << ::a << endl;  // 输出100（全局a）,作用域解析运算符显式访问全局a
}
```

每个 a 获得独立的符号表项和内存地址，DeclRefExpr 通过地址精确绑定到正确的声明，实现了词法作用域的语义。

**外部符号依赖分析** 符号表分析揭示了程序的链接需求：

表 8: 符号类型与链接语义

| 符号 | 类型 | 地址 | 链接语义 |
|---|---|---|---|
| main | T | 0x00000000 | 本地定义，程序入口 |
| std::cin | U | undefined | 需要 libstdc++.so |
| std::cout | U | undefined | 需要 libstdc++.so |
| std::endl | U | undefined | 模板实例化，需要链接 |

**控制流语义验证** While 循环的语义约束检查：

- 条件表达式 `i < n` 必须可转换为 `bool` 类型
- 循环体内的所有路径都必须可达
- 循环变量 `i` 的修改确保循环可终止（`i = i + 1` 保证单调递增）

**初始化与定义性赋值分析** 语义分析确保所有变量在使用前被初始化：

- `a, b, i`：声明后立即初始化为字面量
- `n`：第一次使用（条件判断）前通过 `cin` 初始化
- `t`：在循环体内每次使用前都重新赋值

这种定义性赋值分析防止了未初始化变量的使用，保证了程序正确性。

### 4. 中间代码生成与优化变换分析

LLVM 编译器在语义分析完成后生成 LLVM IR 中间表示，并通过一系列优化 pass 对代码进行变换。通过对比不同优化等级下的 LLVM IR 输出，揭示了中间代码生成机制和优化策略的深层原理。

**LLVM IR 结构变化的量化分析** 不同优化等级下的代码特征对比展现了优化过程的复杂性：

表 9: 不同优化等级下 LLVM IR 结构对比

| 指标 | -O0 | -O2 | -O3 |
|---|---|---|---|
| 代码行数 | 124 | 247 | 247 |
| 基本块数量 | 3 | 17 | 17 |
| 内存分配指令 | 6 | 1 | 1 |
| 文件大小 | 8,597B | 15,700B | 15,700B |

-O2 与-O3 在斐波那契程序中产生完全相同的输出，表明该程序的优化潜力在-O2 级别已被充分挖掘，-O3 的额外激进策略未发现进一步优化空间。

**内存分配策略的根本性变革** 两个版本在变量管理上展现了截然不同的策略：

**O0 版本采用直接映射策略：**

```
%1 = alloca i32, align 4  ; 返回值存储
%2 = alloca i32, align 4  ; 变量a
%3 = alloca i32, align 4  ; 变量b
%4 = alloca i32, align 4  ; 变量i
%5 = alloca i32, align 4  ; 变量t
%6 = alloca i32, align 4  ; 变量n
```

每个源程序变量对应一个独立的栈分配，所有操作通过 `load/store` 指令实现。

**O2 版本实现激进的内存消除：**

```
%1 = alloca i32, align 4  ; 仅保留变量n的存储
%2 = bitcast i32* %1 to i8*
call void @llvm.lifetime.start.p0i8(i64 4, i8* nonnull %2) #7
```

通过数据流分析，编译器确定只有输入变量 `n` 需要内存存储，其余变量完全消除或转换为 SSA 形式的虚拟寄存器。

**常量传播与折叠优化** O2 版本展现了编译器的常量分析能力：

```
; 直接输出常量0和1，无需内存加载
%4 = call noundef nonnull align 8 dereferenceable(8)
     %"class.std::basic_ostream"* @_ZNSolsEi(..., i32 noundef 0)
%34 = call noundef nonnull align 8 dereferenceable(8)
      %"class.std::basic_ostream"* @_ZNSolsEi(..., i32 noundef 1)
```

编译器识别出变量 a 和 b 的初值为编译时常量，直接在输出操作中使用字面量，避免了内存分配和加载开销。

**循环结构的多层优化** **循环条件预检查机制：** O2 版本引入了循环预检查优化，在进入主循环前判断是否需要执行：

```
%64 = load i32, i32* %1, align 4, !tbaa !16
%65 = icmp sgt i32 %64, 1           ; 检查n > 1
br i1 %65, label %66, label %104    ; 条件跳转
```

当输入 n <= 1 时，程序直接跳转到退出块，完全跳过循环执行。

**SSA 形式的循环变量管理：**主循环采用 phi 节点实现高效的变量状态管理：

```
66: ; 循环头部
    %67 = phi i32 [ %69, %97 ], [ 0, %60 ]     ; 变量a的SSA形式
    %68 = phi i32 [ %101, %97 ], [ 1, %60 ]    ; 循环计数器i
    %69 = phi i32 [ %70, %97 ], [ 1, %60 ]     ; 变量b的SSA形式
    %70 = add nsw i32 %67, %69                 ; 斐波那契计算
```

临时变量 t 被完全消除，编译器通过 phi 节点的数据流依赖直接管理变量交换，实现了零开销的状态更新。

**函数内联的代价与收益分析** 代码膨胀的主要来源是 C++ 标准库函数的完全内联。以 std::endl 为例，O0 版本的简单调用：

```
%10 = call noundef nonnull align 8 dereferenceable(8)
     %"class.std::basic_ostream"* @_ZNSolsEPFRSoS_E(...)
```

在 O2 版本中展开为包含以下复杂逻辑的代码序列：

1. **字符类型处理：**通过 std::ctype 进行字符宽度转换
2. **虚函数调用解析：**编译时解析虚函数表查找
3. **异常安全检查：**插入 bad_cast 异常处理路径
4. **流缓冲区管理：**显式调用 put 和 flush 操作

这种内联策略虽然增加了代码体积，但消除了函数调用开销和间接寻址成本。

**基本块结构的功能分析** O2 版本的 17 个基本块展现了明确的功能划分：

表 10: O2 版本基本块功能分布

| 基本块范围 | 功能 |
|---|---|
| 0 | 函数入口与初始化 |
| 16-30 | 首次输出”0” 的字符处理 |
| 46-60 | 首次输出”1” 的字符处理 |
| 66 | 循环头部与 phi 节点 |
| 83-97 | 循环内输出的字符处理 |
| 104 | 函数退出与清理 |

基本块数量的急剧增加主要源于 C++ 流操作的内联展开，而非循环优化本身的复杂化。

**优化 Pass 的协同作用机制** 通过 opt 工具分析，LLVM 14.0 的关键循环优化 pass 包括：

- **mem2reg**：将栈变量提升为 SSA 寄存器形式
- **loop-simplify**：规范化循环结构为标准形式
- **indvars**：归纳变量简化和强度削减
- **licm**：循环不变量代码外提
- **sroa**：标量替换聚合对象优化

这些 pass 的顺序执行和相互作用，将源程序的命令式循环结构转换为高度优化的函数式 SSA 表示，在保持语义等价的前提下实现了显著的性能提升。优化后的代码消除了冗余的内存访问、减少了指令数量，并为后续的寄存器分配和指令选择阶段提供了更优的输入。

之后，我通过分析 GCC 的 dump 机制生成不同优化等级下的控制流图，揭示了编译器中间代码生成和优化策略的本质差异。SSA（静态单赋值）阶段的 CFG 对比分析展现了现代编译器优化的核心机制。

**CFG 结构复杂度的量化差异** 使用`-fdump-tree-all-graph` 和`-fdump-rtl-all-graph` 参数生成的 dump 文件数量直接反映了优化复杂度：

表 11: 不同优化等级下的 CFG 特征对比

| 特征指标 | O0 版本 | O2 版本 |
|---|---|---|
| dump 文件总数 | 35 | 169 |
| 初始 CFG 基本块数 | 28 | 59 |
| 主函数 CFG 单元 | 1 个完整图 | 8 个独立子图 |
| 循环标识数量 | 1 个（Loop 1） | 复杂嵌套结构 |

O2 版本产生的 dump 文件数量比 O0 多出 383%，表明激进优化策略引入了大量中间变换阶段。

**SSA 形式下的循环结构分析** SSA 阶段的 CFG 对比揭示了两种根本不同的代码组织策略：

**O0 版本的直观映射**：CFG 呈现线性结构，基本块序列 `bb 2→bb 3→bb 4→bb 5→bb 6` 直接对应源码的顺序执行。循环头部 `bb 11` 集中管理 4 个 phi 节点：

```
# a_5 = PHI <a_10(7), a_28(10)>   // 变量a的SSA合并
# b_6 = PHI <b_11(7), b_24(10)>   // 变量b的SSA合并
# i_7 = PHI <i_12(7), i_29(10)>   // 循环计数器i
# t_8 = PHI <t_4(7), t_2(10)>     // 临时变量t
```

这种组织方式保持了源码语义的直接对应关系，便于理解和调试。

**O2 版本的激进重构**：单一主函数被分解为 8 个独立的 CFG 子图，每个子图处理特定的功能模块。这种分解策略源于 C++ 标准库函数的完全内联，将 `std::cout`、`std::endl` 等操作展开为独立的控制流路径。

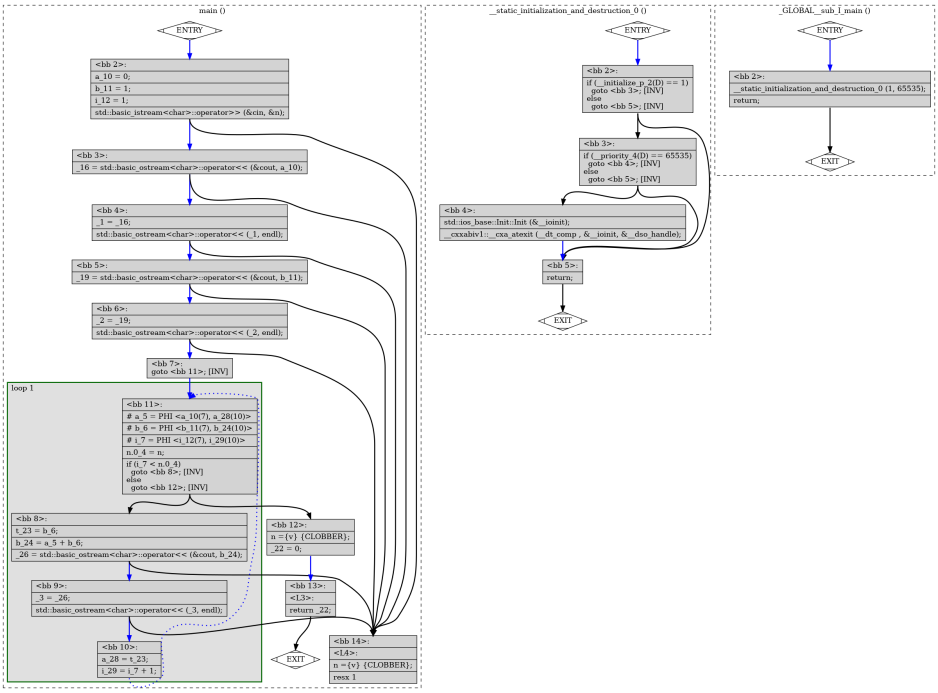



图 4: O0 优化等级下 SSA 阶段 CFG：线性结构与循环头部 phi 节点集中管理

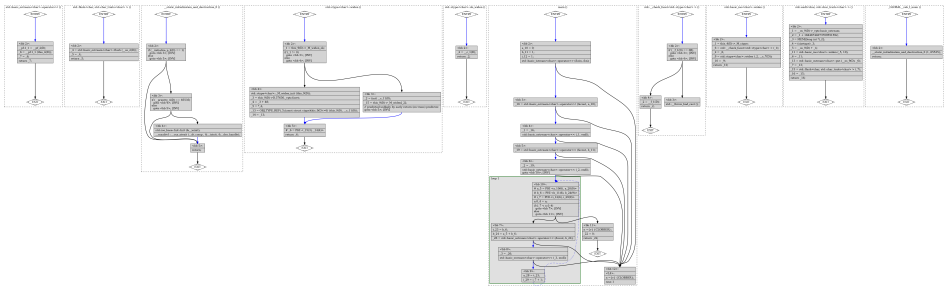


图 5: O2 优化等级下 SSA 阶段 CFG：函数内联导致的多子图分解结构

**控制流复杂度增长的根因分析** O2 版本 CFG 复杂度的急剧增长主要源于两个因素：

1. **函数内联的副作用：**C++ 流操作的完全内联将简单的输出语句展开为包含字符处理、异常检查、缓冲区管理的复杂控制路径
2. **优化准备的细粒度分解：**为支持后续的循环变换、向量化等高级优化，编译器将代码分解为更小的、语义明确的基本块单元

**循环优化潜力的评估** 通过分析 O2 版本中循环优化相关的 dump 文件大小，发现一个有趣的模式：

- `153t.ivcanon`、`169t.cunroll`、`174t.ivopts` 三个阶段的文件大小完全相同（373,709 字节）
- 表明对于斐波那契数列这类简单循环，归纳变量规范化、循环展开、归纳变量优化等变换并未改变 CFG 结构
- 证实了该算法已达到 GCC 循环优化的理论上限

### 5. 代码优化与机器无关变换

代码优化阶段对中间代码执行机器无关的变换以提升程序性能。采用包含递归函数、循环结构、死代码和冗余计算的增强版斐波那契程序，通过对比不同优化等级下的 LLVM IR 输出，分析编译器的具体优化机制。

**LLVM IR 结构变化的量化分析** 测试程序特意引入 `unused_var` 变量、`mul×1` 操作、`CONSTANT_ONE` 常量、冗余的 `double_n = n*2; temp_calc = double_n/2` 计算等优化目标，以观察编译器的处理效果。

表 12: 不同优化等级下 LLVM IR 结构对比

| 优化指标 | O0 版本 | O2 版本 | 变化率 |
|---|---|---|---|
| 代码行数 | 210 | 366 | +74% |
| 内存分配 (alloca) | 14 | 1 | -92.9% |
| 存储指令 (store) | 18 | 0 | -100% |
| Phi 节点 | 0 | 14 | $+\infty$ |
| 基本块数 | 10 | 29 | +190% |
| 函数数量 | 4 | 3 | -25% |

表中数据显示两种截然不同的代码组织策略。O0 版本采用直接映射方式，每个源程序变量对应一个栈分配；O2 版本通过内存消除技术将 14 个栈分配减少至 1 个，同时存储指令完全消除。代码行数增长 74% 的原因是 C++ 标准库函数的完全内联展开，将简单的 `cout`、`endl` 调用替换为包含字符处理、虚函数解析、缓冲区管理的复杂代码序列。

基本块数量从 10 个增长至 29 个反映了控制流的细粒度分解。每个内联的流操作函数引入多个基本块处理不同的执行路径，包括字符编码转换、异常处理、流状态检查等分支。

**SSA 形式转换的内存管理机制** O0 版本采用栈分配加载存储模式管理变量：

```
%2 = alloca i32, align 4     ; 为变量a分配栈空间
%3 = alloca i32, align 4     ; 为变量b分配栈空间
store i32 0, i32* %2         ; 将常量0存储到a的内存位置
%6 = load i32, i32* %2       ; 从内存加载a的值用于计算
%7 = load i32, i32* %3       ; 从内存加载b的值用于计算
%8 = add nsw i32 %6, %7      ; 执行a+b运算
```

此模式下每次变量访问都需要内存操作，产生 18 个 store 指令和 25 个 load 指令。

O2 版本通过 mem2reg pass 将栈变量提升为 SSA 形式的虚拟寄存器：

```
99:                          ; 循环头部基本块
%100 = phi i32 [ %101, %133 ], [ 0, %93 ]     ; 变量a的SSA定义
%101 = phi i32 [ %104, %133 ], [ 1, %93 ]     ; 变量b的SSA定义
%102 = phi i32 [ %106, %133 ], [ 1, %93 ]     ; 循环计数器i
%103 = phi i32 [ %105, %133 ], [ 0, %93 ]     ; 累加器sum
%104 = add nsw i32 %100, %101                 ; 直接寄存器运算
```

每个 phi 节点的两个操作数分别表示变量的初始值（来自基本块 93）和循环更新值（来自基本块 133）。这种表示消除了所有内存分配，将变量状态管理转换为数据流依赖关系。

**死代码消除和常量传播的数据流分析** 编译器通过数据流分析识别并消除无效代码：

表 13: 具体优化技术的量化效果

| 优化目标 | O0 版本检出 | O2 版本处理 |
|---|---|---|
| unused_var 变量 | 存在但未使用 | 完全消除 |
| mul×1 冗余操作 | 2 次 | 0 次 |
| 常量相关操作 | 16 次 | 9 次 |
| if(unused_var==0) 分支 | 存在 | 消除 |

死代码消除 pass 通过活跃变量分析确定 unused_var 从未被读取，因此将其定义和相关的条件分支完全删除。常量传播 pass 识别出 CONSTANT_ONE = 1 的编译时常量性质，将所有引用替换为字面量 1，减少了 7 个常量相关操作。

对于冗余计算 double_n = n*2; temp_calc = double_n/2，编译器通过代数简化识别出 temp_calc 等价于 n，直接使用原变量替代计算结果。强度削减优化将 i*1 操作识别为恒等变换，完全消除此类指令。

**优化 pass 协同失效的根因分析** 单独运行 mem2reg、dce、sccp 等优化 pass 时观察到所有量化指标保持不变的异常现象。通过检查函数属性发现根因：

```
; Function Attrs: mustprogress noinline optnone uwtable
define dso_local noundef i32 @main() #5 {
attributes #5 = { mustprogress noinline norecurse optnone uwtable ... }
```

optnone 属性指示 LLVM 跳过该函数的所有优化变换，noinline 属性阻止函数内联。这些属性由 clang -O0 自动添加以保证调试信息的准确性和单步执行的可预测性。

手动移除这些限制属性后，基础 pass 组合产生显著优化效果：

表 14: 移除 optnone 属性后的优化效果验证

| 版本 | alloca | store | phi | 行数 |
|---|---|---|---|---|
| 原始 O0 | 14 | 18 | 0 | 210 |
| 移除 optnone | 14 | 18 | 0 | 210 |
| mem2reg+sroa+dce+sccp 后 | 1 | 0 | 5 | 158 |

移除限制属性本身不改变代码结构，但使后续优化 pass 能够正常执行。优化后内存分配减少 92.9%，存储指令完全消除，代码行数减少 24.8%，产生 5 个 phi 节点管理数据流。这证实了属性系统对优化行为的控制机制。

**函数内联导致的控制流复杂化** O2 版本的递归函数 fibonacci 通过尾递归优化转换为迭代形式，但主要的代码膨胀来源于 C++ 流操作的内联展开。以 std::endl 为例：

O0 版本的简单函数调用：

```
call @_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
```

在 O2 版本中展开为包含以下操作的复杂序列：

1. 虚函数表查找和字符类型解析
2. std::ctype 模板实例化和字符宽度转换
3. 异常安全检查和 bad_cast 处理路径

4. 流缓冲区的 `put` 和 `flush` 操作

每个内联函数引入 3-5 个额外基本块处理不同执行路径，导致总基本块数从 10 个增长至 29 个。

**编译器前端与后端的优化分工** 对比 `opt` 工具与 `clang -O2` 的效果差异揭示了编译器的分层架构：

表 15: 不同优化工具的效果对比

| 工具 | alloca | store | 行数变化 | 主要策略 |
|---|---|---|---|---|
| `opt -O2`（含 optnone） | 14→14 | 18→18 | 210→210 | 无优化 |
| `opt`（移除 optnone） | 14→1 | 18→0 | 210→158 | 仅内存优化 |
| `clang -O2` | 14→1 | 18→0 | 210→366 | 内存优化 + 函数内联 |

数据表明 `opt`（移除 optnone）与 `clang -O2` 在内存优化效果上完全一致，均将 alloca 指令从 14 个减少至 1 个，store 指令完全消除。差异在于代码组织策略：`opt` 仅执行基本的内存到寄存器变换，代码行数减少至 158 行；`clang -O2` 额外执行 C++ 标准库函数的完全内联，将 `cout`、`endl` 等调用展开为包含字符处理、异常检查的复杂序列，导致代码膨胀至 366 行。

### 6. 代码生成与目标代码转换

代码生成阶段将 LLVM IR 转换为 RISC-V 汇编代码，通过寄存器分配、指令选择和调度优化生成目标机器码。采用原始的斐波那契程序，对比不同优化等级下的汇编输出，分析后端代码生成的具体策略。

**汇编代码结构的量化分析** 两个优化等级产生的汇编代码在规模和指令分布上呈现明显差异：

表 16: 不同优化等级下汇编代码特征对比

| 代码特征 | O0 版本 | O2 版本 | 变化 |
|---|---|---|---|
| 总行数 | 169 | 201 | +18.9% |
| 指令数量 | 85 | 126 | +48.2% |
| 文件大小 | 5705 字节 | 5533 字节 | -3.0% |
| load 指令 (ld/lw) | 26 | 30 | +15.4% |
| store 指令 (sd/sw) | 17 | 8 | -52.9% |
| 算术指令 (add/sub) | 31 | 23 | -25.8% |
| 分支指令 (beq/bne/blt) | 32 | 42 | +31.3% |
| 函数调用 (call) | 10 | 15 | +50.0% |

表中数据显示 O2 版本指令数量增加 48.2%，但文件大小反而减少 3.0%。这种反常现象源于指令编码效率的差异：O2 版本使用更多的短编码指令（如 `mv`、`li`），而 O0 版本包含较多的长编码内存寻址指令。

store 指令减少 52.9% 反映了寄存器分配策略的根本改变。O0 版本每次变量操作都需要栈存储，O2 版本通过寄存器分配将大部分变量保持在寄存器中，仅在函数调用边界进行必要的溢出。

**变量存储策略的架构差异** O0 版本采用基于栈偏移的变量管理方式：

```
li      a0, 0                    ; 加载常量0到寄存器
sw      a0, -20(s0)              ; 存储到栈偏移-20（变量a）
sw      a0, -24(s0)              ; 存储到栈偏移-24（变量b的初值）
li      a0, 1                    ; 加载常量1到寄存器
sw      a0, -28(s0)              ; 存储到栈偏移-28（变量b）
sw      a0, -32(s0)              ; 存储到栈偏移-32（变量i）
```

每个源程序变量对应固定的栈偏移位置，变量访问通过 load/store 指令序列实现。栈偏移使用统计显示 5 个变量分别占用-20、-24、-28、-32、-36 位置，形成连续的内存布局。

O2 版本通过寄存器分配消除大部分栈操作：

```
li      a1, 0                    ; 常量0直接用于函数调用
call    _ZNSolsEi@plt            ; 输出操作使用立即数
mv      s0, a0                   ; 将返回值保存到s0寄存器
li      s3, 1                    ; 变量i保存在s3寄存器
addw    s2, a0, s4               ; 斐波那契计算直接使用寄存器
```

寄存器使用统计显示 s0 被引用 24 次，s1 被引用 20 次，s2-s4 各被引用 6-7 次，表明编译器将频繁访问的变量分配到不同的保存寄存器中。

**寄存器分配的资源管理策略** 两个版本在寄存器使用上展现了不同的资源管理策略：

表 17: 寄存器类型使用统计

| 寄存器类型 | O0 版本 | O2 版本 |
|---|---|---|
| 栈指针操作 (sp) | 9 次 | 5 次 |
| 保存寄存器 (s0-s11) | 16 次 | 51 次 |
| 参数寄存器 (a0-a7) | 47 次 | 72 次 |
| 临时寄存器 (t0-t6) | 0 次 | 0 次 |

O0 版本的栈指针操作数量（9 次）几乎是 O2 版本（5 次）的两倍，说明 O0 版本频繁进行栈帧调整和局部变量寻址。O2 版本的保存寄存器使用量增长 218.8%（16→51 次），表明编译器将更多变量分配到需要跨函数调用保持的寄存器中。

参数寄存器使用的增长（47→72 次，+53.2%）反映了 O2 版本在函数调用时的寄存器传递优化。编译器避免了不必要的栈参数传递，直接通过寄存器完成参数和返回值的管理。

**循环控制流的优化实现** O0 版本采用直观的循环控制结构：

```
lw      a0, -32(s0)              ; 加载变量i
lw      a1, -40(s0)              ; 加载变量n
bge     a0, a1, .LBB1_3          ; 比较i>=n, 退出循环
j       .LBB1_2                  ; 无条件跳转到循环体
.LBB1_2:                         ; 循环体标签
lw      a0, -28(s0)              ; 加载变量b
sw      a0, -36(s0)              ; 存储到临时变量t
addw    a0, a0, a1               ; 执行a+b运算
sw      a0, -28(s0)              ; 存储新的b值
```

这种实现直接对应源代码的 while 结构，每次循环迭代都需要 4 个内存访问指令（2 个 load + 2 个 store）。

O2 版本通过更复杂的控制流优化循环执行：

```
li      a1, 2                    ; 加载常量2
blt     a0, a1, .LBB0_15         ; 预检查: 如果n<2直接退出
```

```
li      s3, 1                   ; 将循环计数器i保存在s3
bge     s3, a1, .LBB0_15        ; 循环条件检查：i>=n退出
.LBB0_12:                       ; 循环头部
addw    s2, a0, s4              ; 斐波那契计算：s2 = a + b
mv      s4, s2                  ; 更新：s4 = 新的b值
```

O2 版本引入了循环预检查机制，当输入 n 小于 2 时直接跳过所有计算。循环体内的变量更新完全通过寄存器操作完成，消除了内存访问开销。

**函数调用约定的优化策略** 函数调用处理显示了两种不同的调用约定实现。O0 版本在每次函数调用前后都进行栈 spill/reload 操作：

```
ld      a0, %pcrel_lo(.LBB1_5)(a0)
sd      a0, -56(s0)             ; 函数调用前保存到栈
call    _ZNSolsEi@plt           ; 执行函数调用
ld      a0, -56(s0)             ; 函数调用后从栈恢复
lw      a1, -28(s0)             ; 加载下一个参数
call    _ZNSolsEi@plt           ; 下一个函数调用
```

每个函数调用都伴随 explicit 的栈操作，确保寄存器状态的保持。

O2 版本通过寄存器传递优化减少了栈操作：

```
li      a1, 0                   ; 直接使用立即数作为参数
call    _ZNSolsEi@plt           ; 函数调用
mv      s0, a0                  ; 将返回值直接保存到保存寄存器
mv      a0, s1                  ; 从保存寄存器准备下一个调用参数
call    _ZNKSt5ctypeIcE13_M_widen_initEv@plt
```

通过 `mv` 指令在保存寄存器间直接传递数据，避免了内存访问开销。

**指令调度与代码密度分析** 两个版本在指令组织上呈现不同的密度特征：

表 18: 代码生成效率对比

| 效率指标 | O0 版本 | O2 版本 | 变化原因 |
|---|---|---|---|
| 指令密度 | 0.50(85/169) | 0.63(126/201) | 更多有效计算指令 |
| 内存操作占比 | 50.6%(43/85) | 30.2%(38/126) | 寄存器分配优化 |
| 栈帧利用率 | 8 个偏移位置 | 5 个寄存器 | 变量存储策略变化 |
| 分支密度 | 37.6%(32/85) | 33.3%(42/126) | 控制流复杂度增加 |

O2 版本的指令密度提升 26%（0.50→0.63），表明每行代码包含更多的有效计算指令。内存操作占比下降 40.3%，证实了寄存器分配在减少内存访问方面的效果。

栈帧利用率的变化反映了存储策略的根本转换：O0 版本使用 8 个不同的栈偏移位置管理变量，O2 版本仅使用 5 个保存寄存器就完成了相同的功能，显示了寄存器分配算法的效率。

## （三） 汇编器与可重定位目标文件生成

汇编器将汇编语言源码转换为包含机器指令的可重定位目标文件。通过对比不同优化等级下的汇编处理过程，分析汇编器在段组织、符号管理、重定位信息生成和机器码编码方面的具体机制。

表 19: 汇编器文件格式转换效果对比

| 转换指标 | O0 版本 | O2 版本 | 技术解释 |
|---|---|---|---|
| 汇编文件大小 | 5333 字节 | 5377 字节 | 文本格式输入 |
| 目标文件大小 | 4792 字节 | 5304 字节 | ELF 二进制格式 |
| 压缩效率 | 89% | 98% | 编码密度差异 |
| 文件类型 | ELF 64-bit LSB relocatable | | 可重定位格式 |

**1. 汇编器输入输出的文件格式转换**

汇编器处理展现了文本格式到二进制格式的转换效率差异：

O0 版本的压缩效率为 89%，优于 O2 版本的 98%。这种差异源于指令编码的复杂度：O0 版本主要包含简单的 load/store 指令和短跳转，汇编器能够使用 RISC-V 压缩指令集 (RVC) 的 16 位编码；O2 版本包含更多的长距离跳转和复杂寻址模式，需要 32 位完整指令编码。

目标文件的 ELF 格式标识显示"UCB RISC-V, RVC, double-float ABI"，表明汇编器启用了 RISC-V 压缩指令扩展，支持双精度浮点 ABI，生成与 RISC-V 标准完全兼容的可重定位目标文件。

**2. ELF 段结构的组织与管理**

汇编器将汇编代码组织到标准化的 ELF 段结构中：

表 20: 目标文件段结构对比分析

| 段名称 | O0 大小 | O2 大小 | 段功能 |
|---|---|---|---|
| .text | 0xe4(228 字节) | 0x158(344 字节) | 主函数机器码 |
| .text.startup | 0x58(88 字节) | 0x38(56 字节) | 初始化代码 |
| .data | 0 字节 | 0 字节 | 已初始化数据 |
| .bss | 0 字节 | 0 字节 | 未初始化数据 |
| .sbss | 1 字节 | 1 字节 | 小型 BSS 数据 |
| .init_array | 8 字节 | 8 字节 | 构造函数指针 |
| .eh_frame | 0x78(120 字节) | 0x58(88 字节) | 异常处理信息 |

主要的.text 段大小差异反映了编译器优化策略对机器码长度的影响。O2 版本的.text 段增长 51.3%（228→344 字节），主要原因是函数内联导致的代码膨胀。相反，.text.startup 段在 O2 版本中减少 36.4%（88→56 字节），说明启动代码通过优化变得更紧凑。

.eh_frame 段大小的减少（120→88 字节）表明 O2 版本的异常处理信息更简洁。这是因为优化后的代码结构更规整，减少了异常处理的控制流复杂度。

**3. 符号表的生成与分类管理**

汇编器根据符号的作用域和定义状态生成分类化的符号表：

未定义符号数量从 9 个增加到 12 个,增长 33.3%。新增的符号包括`_ZNKSt5ctypeIcE13_M_widen_initEv`、`_ZNSo3putEc`、`_ZNSo5flushEv`、`_ZSt16__throw_bad_castv`，这些都是 O2 版本中 C++ 标准库函数内联展开后引入的底层实现函数。局部符号减少 1 个的原因是 O2 版本消除了 O0 版本中的`__cxx_global_var_init`函数，该函数被直接内联到`_GLOBAL__sub_I_Fibonacci.cpp`中，体现了编译器在全局变量初始化方面的优化。

表 21: 符号表条目分类统计

| 符号类型 | O0 版本 | O2 版本 | 符号特征 |
|---|---|---|---|
| 局部符号 (l) | 14 个 | 13 个 | 文件内部可见 |
| 全局符号 (g) | 1 个 | 1 个 | 外部可见 (main 函数) |
| 未定义符号 (*UND*) | 9 个 | 12 个 | 需要外部链接 |
| 总计 | 24 个 | 26 个 | 符号表大小 |

### 4. 重定位信息的生成与分类

汇编器为每个需要地址解析的位置生成重定位条目：

表 22: 重定位条目分布与类型分析

| 段名称 | O0 条目数 | O2 条目数 | 主要重定位类型 |
|---|---|---|---|
| .text | 29 个 | 47 个 | 函数调用与分支跳转 |
| .text.startup | 14 个 | 12 个 | 初始化代码地址解析 |
| .init_array | 1 个 | 1 个 | 构造函数指针 |
| .eh_frame | 9 个 | 6 个 | 异常处理地址 |
| 总计 | 53 个 | 66 个 | 重定位复杂度 |

.text 段的重定位条目增长 62.1%（29→47 个），直接反映了 O2 版本代码复杂度的提升。主要的重定位类型包括：

1. **R_RISCV_CALL_PLT**：用于外部函数调用，支持过程链接表 (PLT) 机制
2. **R_RISCV_GOT_HI20/R_RISCV_PCREL_LO12_I**：全局偏移表 (GOT) 访问的高位和低位地址计算
3. **R_RISCV_BRANCH/R_RISCV_RVC_JUMP**：条件分支和无条件跳转的地址重定位
4. **R_RISCV_RELAX**：链接时优化标记，允许链接器进行指令松弛优化

.text.startup 段的重定位条目减少说明初始化代码变得更简洁，而.eh_frame 段的减少表明异常处理机制的简化。

### 5. 机器码编码的指令格式分析

汇编器将 RISC-V 汇编指令转换为对应的机器码格式：

```
O0版本main函数开头的机器码编码:
   0:   7139          addi    sp,sp,-64     ; 16位压缩指令
   2:   fc06          sd      ra,56(sp)     ; 16位压缩指令
   4:   f822          sd      s0,48(sp)     ; 16位压缩指令
   8:   4501          li      a0,0          ; 16位立即数加载
   a:   fea42623      sw      a0,-20(s0)    ; 32位store指令

O2版本main函数开头的机器码编码:
   0:   7139          addi    sp,sp,-64     ; 相同的栈帧设置
   2:   fc06          sd      ra,56(sp)     ; 相同的寄存器保存
   6:   f426          sd      s1,40(sp)     ; 额外的寄存器保存
   8:   f04a          sd      s2,32(sp)     ; 更多寄存器分配
   a:   ec4e          sd      s3,24(sp)     ; 寄存器使用增加
```

机器码分析显示汇编器对指令编码的优化策略：短立即数和寄存器操作使用 16 位 RVC 压缩指令，复杂寻址模式使用 32 位标准指令。O0 版本在偏移 0x8 位置就开始使用 32 位指令进行内存操作，而 O2 版本更多地使用 16 位指令进行寄存器保存，体现了不同的代码密度特征。

主函数大小从 226 字节增长到 344 字节（+52.2%），机器码指令数量的增加主要来源于：循环优化引入的预检查代码、函数内联展开的复杂控制流、以及更多的寄存器 spill/reload 操作。

#### 6. 汇编器的地址空间管理

汇编器生成的目标文件使用虚拟地址空间 0x0 作为基址，所有段的 VMA(虚拟内存地址) 和 LMA(加载内存地址) 都设置为 0x0。这种设计使目标文件具备完全的可重定位性，链接器可以将其装载到任意的内存地址空间。

文件偏移地址的分配体现了汇编器的空间优化：.text 段从 0x40 开始，为 ELF 头部预留空间；各段按照对齐要求紧密排列，最大化存储效率。O2 版本的总文件大小比 O0 版本大 10.7%（4792→5304 字节），主要原因是.text 段的扩展和更多重定位条目的存储开销。

### （四） 链接器

链接器将汇编器生成的可重定位目标文件转换为可执行文件，通过符号解析、地址重定位、段合并等机制实现从相对地址到绝对地址的转换。本节通过对比动态链接与静态链接两种策略，分析链接器的核心工作机制。

#### 1. 链接器输入输出的文件格式转换

链接器处理展现了目标文件到可执行文件的根本性转换。O0 版本的目标文件（5576 字节）经动态链接生成 9264 字节的可执行文件，增长 66.2%；经静态链接生成 1736496 字节的可执行文件，增长 31,041%。文件类型从”ELF 64-bit LSB relocatable” 转换为”ELF 64-bit LSB pie executable”（动态链接）或”ELF 64-bit LSB executable”（静态链接）。

动态链接版本标识为”dynamically linked, interpreter /lib/ld-linux-riscv64-lp64d.so.1”，表明程序运行时需要动态链接器解析外部符号；静态链接版本标识为”statically linked”，包含所有依赖库代码，无需外部解析。两种策略的文件大小差异达 187 倍，反映了链接策略对程序分发和运行环境的不同要求。

#### 2. 重定位处理的核心算法

链接器通过重定位算法将目标文件中的 85 个重定位条目减少至可执行文件中的 21 个，重定位解析率达 75%。目标文件中的重定位需求主要包括三类：

表 23: 重定位类型转换统计

| 目标文件重定位类型 | 数量 | 可执行文件转换结果 |
|---|---|---|
| R_RISCV_CALL_PLT | 10 | R_RISCV_JUMP_SLOT (7 个) |
| R_RISCV_GOT_HI20 | 10 | R_RISCV_64 (8 个) |
| R_RISCV_PCREL_LO12_I | 20 | 链接时完全解析 |
| R_RISCV_RELAX | 24 | 链接时优化消除 |
| 其他类型 | 21 | R_RISCV_RELATIVE (6 个) |

函数调用相关的 `R_RISCV_CALL_PLT` 重定位转换为 `R_RISCV_JUMP_SLOT` 类型，启用过程链接表（PLT）机制实现延迟绑定。全局变量访问的 `R_RISCV_GOT_HI20` 重定位转换为 `R_RISCV_64`

类型，通过全局偏移表（GOT）实现间接寻址。24 个 R_RISCV_RELAX 标记指示链接器在地址确定后执行指令松弛优化，这些标记在链接完成后被消除。

### 3. 符号解析的分层机制

链接器实现三层符号解析架构：本地符号的链接时绑定、全局符号的地址分配、未定义符号的运行时解析。目标文件中的 3 个本地符号（main、_GLOBAL__sub_I_main、_Z41__static_initialization_and_destruction_0ii）在链接时完全解析，获得绝对地址并合并到可执行文件的符号表中。

可执行文件生成 8 个全局符号定义，包括程序入口点 _start（地址 0x950）、用户主函数 main（地址 0xa08）以及数据段边界标记 __DATA_BEGIN__、_edata 等。这些符号的地址在链接时确定，为程序运行和调试器提供固定的引用点。

12 个未定义符号保持 U（未定义）状态，包括 C++ 标准库函数 _ZNSolsEi@GLIBCXX_3.4（cout << int 操作）、_ZSt3cin@GLIBCXX_3.4（cin 对象）、系统库函数 __libc_start_main@GLIBC_2.34 等。这些符号的解析推迟到运行时，由动态链接器从共享库中加载。

### 4. PLT/GOT 延迟绑定机制

过程链接表（PLT）与全局偏移表（GOT）协同实现外部函数调用的延迟绑定机制。PLT 表位于地址 0x8c0-0x94f，包含 7 个函数条目，每个条目占 16 字节，采用统一的三指令序列：auipc + ld + jalr + nop。

以 _ZNSolsEi 函数为例，其 PLT 条目（地址 0x930）包含指令序列：

```
auipc   t3,0x1              ; 计算GOT基址
ld      t3,1808(t3) # 2040  ; 从GOT[2040]加载函数地址
jalr    t1,t3               ; 间接跳转到函数
nop                         ; 对齐填充
```

GOT 表初始化展现延迟绑定的核心机制。地址 0x2018-0x2048 范围内的 7 个函数指针均初始化为 0x8c0，对应 PLT 表头的解析器地址。首次函数调用触发以下执行路径：程序调用 PLT 条目 → 从 GOT 加载地址 0x8c0→ 跳转到 PLT 解析器 → 调用动态链接器解析真实函数地址 → 更新 GOT 条目为真实地址 → 跳转到目标函数。后续调用直接从 GOT 获得真实地址，实现 O(1) 的函数调用开销。

GOT 表内容验证了延迟绑定的技术实现。0xc0080000 的小端序表示对应地址 0x8c0，证实了 GOT 表项在加载时指向 PLT 解析器。地址 0x2058 开始的区域初始化为 0，用于存储全局变量指针，这些指针在程序加载时由动态链接器填充实际地址。

### 5. 静态链接与动态链接的技术权衡

静态链接通过符号合并消除了 PLT/GOT 机制的复杂性。静态版本包含 1590 个代码段符号，比动态版本的 8 个符号增长 198 倍，反映了完整 C++ 标准库的内联包含。函数调用采用直接寻址模式，_ZNSolsEi 的调用指令为 jal ra,59482，直接跳转到绝对地址而非 PLT 条目。

地址空间分配策略的差异体现了两种链接方法的本质区别：

静态链接的 main 函数地址（0x1100e）显著高于动态链接版本（0xa08），原因是大量库代码的预置占用了低地址空间。代码段规模增长 1451 倍，数据段增长 825 倍，BSS 段增长 2050 倍，表明静态链接将完整的运行时环境嵌入到可执行文件中。

库依赖关系的处理方式反映了运行时环境的不同要求。动态链接版本的.dynamic 段声明了 3 个共享库依赖：libstdc++.so.6（C++ 标准库）、libc.so.6（C 库）、ld-linux-riscv64-lp64d.so.1（动态链接器），这些依赖在程序启动时由操作系统动态加载。静态链接版本无.dynamic 段，所有依赖在链接时解析并内联到可执行文件中，程序可在无共享库环境中独立运行。

表 24: 地址空间组织对比

| 组件 | 动态链接 | 静态链接 |
|---|---|---|
| main 函数地址 | 0xa08 | 0x1100e |
| 代码段大小 | 0x23a (570 字节) | 0xc9ef2 (827KB) |
| 数据段大小 | 0x8 (8 字节) | 0x19d8 (6.6KB) |
| BSS 段大小 | 0x10 (16 字节) | 0x8238 (32KB) |
| PLT/GOT 段 | 存在 | 不存在 |

**6. 段结构重组与内存布局**

链接器通过段合并和重新布局优化内存使用效率。动态链接版本生成 21 个段，包括动态链接专用的`.plt` 段（144 字节）、`.got` 段（152 字节）、`.dynamic` 段（528 字节）。静态链接版本简化为 20 个段，消除了动态链接相关段，但`.data.rel.ro` 段扩展至 37KB，存储重定位的只读数据。

段对齐和填充策略反映了不同的内存管理需求。动态链接版本的段边界按页对齐（4KB），便于操作系统的内存保护和共享；静态链接版本采用更紧密的对齐策略，减少内存碎片但牺牲了运行时的灵活性。

段虚拟地址分配展现了链接器的地址空间管理机制。动态链接版本使用基址 0x0 的相对寻址，支持地址空间布局随机化（ASLR）；静态链接版本使用固定的绝对地址，提供可预测的内存布局但降低了安全性。

# 三、 任务二：LLVM IR 的程序编写

LLVM IR 是由代码生成器自顶向下遍历逐步翻译语法树形成的，将任意语言的源代码编译成 LLVM IR，然后由 LLVM 后端对 LLVM IR 进行优化并编译为相应平台的二进制程序。我们通过自己来编写 LLVM IR 代码并运行来学习中间语言的特点。

## （一） 大致编写思路：

编写 LLVM IR 的核心思路是，将高级语言中的概念手动映射到底层的、与具体机器无关的指令集上。为了清晰地展现语言特性，我总共编写了 6 个文件，后面将会讲解其中的核心代码

总体的编写流程如下：

1. **定义目标架构与数据布局**：首先使用 `target triple` 声明代码最终要运行在何种平台上，这会影响后续的指令选择和内存对齐，我选择的是 `x86_64-unknown-linux-gnu`，其实选什么架构都不影响我们核心代码的编写。

2. **声明全局变量与外部函数**：使用 `@` 定义全局变量（如格式化字符串），使用 `declare` 声明需要引用的外部函数（如 C 标准库的 `printf` 或 SysY 运行库的 `getint`）。

3. **函数与基本块**：程序逻辑被组织在函数 `define` 中，每个函数由一系列基本块 `label` 构成。每个基本块是一段线性执行的指令序列，以一个终结指令（如 `br` 或 `ret`）结束。

4. **变量与内存操作**：LLVM IR 使用静态单赋值形式，但局部变量通常通过在栈上分配内存来模拟。使用 `alloca` 在函数入口处为局部变量分配内存，然后通过 `store` 和 `load` 指令进行存取。

5. **控制流**：高级语言中的条件判断 `if-else` 和循环 `while` 结构，需要通过比较指令 `icmp` 和条件跳转指令 `br` 组合实现。通过跳转到不同的基本块来构建程序的控制流图。

## （二） 具体文件分析

### 1. arithmetic.ll：各种数值运算

该文件演示了加、减、乘、除等基本的二元数值运算。

**核心语言特性**：数值运算。

**对应核心代码**：

```
; 加载 a 和 b 的值
%a_val = load i32, i32* %a, align 4
%b_val = load i32, i32* %b, align 4

; —— 加法 ——
%add_res = add nsw i32 %a_val, %b_val

; —— 减法 ——
%sub_res = sub nsw i32 %a_val, %b_val

; —— 乘法 ——
%mul_res = mul nsw i32 %a_val, %b_val

; —— 有符号除法 ——
%div_res = sdiv i32 %a_val, %b_val
```

**说明**：LLVM IR 为不同的数值运算提供了专门的指令。例如，`add` 用于加法，`sub` 用于减法，`mul` 用于乘法，`sdiv` 用于有符号整数除法。这些指令直接对虚拟寄存器中的值进行操作。‘nsw‘ 标志表示“无有符号回绕 (No Signed Wrap)”，是一种优化提示，告诉编译器该运算不会发生有符号溢出。

### 2. assign_const.ll：常量赋值

该文件演示了最基本的语言特性：将一个常量赋值给一个变量。

**核心语言特性**：常量赋值。

**对应核心代码**：

```
; 为变量 a 分配内存
%a = alloca i32, align 4

; 将常量 100 存储到 a
store i32 100, i32* %a, align 4
```

**说明**：首先使用 `alloca` 在栈上为变量 `%a` 分配了 4 字节的内存空间。然后，使用 `store` 指令将 32 位整型常量 `100` 存入该内存地址。这等价于 C 语言中的 `int a = 100;`。

### 3. assign_var.ll：变量赋值

该文件演示了将一个变量的值赋给另一个变量。

**核心语言特性：**变量间赋值。

**对应核心代码：**

```
    ; 初始化 a = 100
    store i32 100, i32* %a, align 4

    ; 加载 a 的值
    %a_val = load i32, i32* %a, align 4

    ; 将 a 的值赋给 b
    store i32 %a_val, i32* %b, align 4
```

**说明：**为了将变量 a 的值赋给 b，需要先用 load 指令从 %a 的内存地址中读取其值到一个临时虚拟寄存器 %a_val。然后，再用 store 指令将 %a_val 中的值写入 %b 的内存地址。这等价于 C 语言中的 b = a;。

#### 4. if_else.ll：条件分支

该文件演示了 if-else 结构。

**核心语言特性：**条件分支。

**对应核心代码：**

```
    ; 加载 a 和 b 的值进行比较
    %a_val = load i32, i32* %a, align 4
    %b_val = load i32, i32* %b, align 4
    %cmp = icmp slt i32 %a_val, %b_val   ; if (a < b)

    ; 根据比较结果进行分支跳转
    br i1 %cmp, label %if_true, label %if_false
```

**说明：**使用 icmp slt (signed less than) 指令比较 %a_val 和 %b_val。该指令返回一个 i1 类型（布尔值）的结果 %cmp。然后，br (branch) 指令根据 %cmp 的值决定跳转到 %if_true 标签（如果为真）还是 %if_false 标签（如果为假），从而实现了 if-else 的控制流。

#### 5. while_loop.ll：循环语句

该文件演示了 while 循环结构。

**核心语言特性：**循环。

**对应核心代码：**

```
    loop_cond:
    %i_val = load i32, i32* %i, align 4
    %cmp = icmp sle i32 %i_val, 5   ; while (i <= 5)
    br i1 %cmp, label %loop_body, label %loop_end

    loop_body:
    ; ... 循环体内的操作 ...
    br label %loop_cond
```

**说明：**循环是通过基本块之间的跳转实现的。在 %loop_cond 块中，使用 icmp sle (signed less or equal) 判断循环条件。如果条件满足，br 指令跳转到执行循环体的 %loop_body 块；否则，

跳转到 %loop_end 块退出循环。循环体执行完毕后，通过一个无条件 br 指令跳回到 %loop_cond，进行下一次条件判断。

### 6. fib_recursive.ll：递归函数

该文件演示了函数的定义和递归调用。

**核心语言特性：**函数定义与递归。

**对应核心代码：**

```
define i32 @fib(i32 %n) {
        entry:
        %cmp = icmp sle i32 %n, 1
        br i1 %cmp, label %base_case, label %recursive_case

        base_case:
        ret i32 %n

        recursive_case:
        %n_minus_1 = sub i32 %n, 1
        %fib1 = call i32 @fib(i32 %n_minus_1) ; 递归调用

        %n_minus_2 = sub i32 %n, 2
        %fib2 = call i32 @fib(i32 %n_minus_2) ; 递归调用

        %result = add i32 %fib1, %fib2
        ret i32 %result
}
```

**说明：**使用 define 定义了名为 fib 的函数。在函数体内部，通过 call 指令再次调用 @fib 函数自身，从而实现递归。递归的终止条件 (n <= 1) 通过 icmp 和 br 实现，满足条件时执行 %base_case 块并返回，否则执行 %recursive_case 块进行递归调用。

### 7. io.ll：输入输出

该文件演示了如何调用外部库函数（SysY 运行库）来进行输入输出。

**核心语言特性：**外部函数调用 (I/O)。

**对应核心代码：**

```
; 声明 SysY 运行库中的输入输出函数
declare i32 @getint()
declare void @putint(i32)

define i32 @main() {
        entry:
        ; 调用 getint 读取一个整数
        %input_val = call i32 @getint()

        ; 调用 putint 打印读取到的整数
        call void @putint(i32 %input_val)
        ret i32 0
```

```
    }
```

**说明：**首先使用 `declare` 声明了两个外部函数 `@getint` 和 `@putint`，它们的具体实现位于 SysY 运行库中。然后在 `@main` 函数中，通过 `call` 指令直接调用这两个函数来完成读入一个整数并将其打印的功能。这展示了 LLVM IR 如何与外部库进行交互。

### （三） 运行结果验证

采用实验指导书中所给出的命令，将下载好的 sylib.so 挪到该目录下，进行验证

```
lli -load=./sylib.so main.ll
```

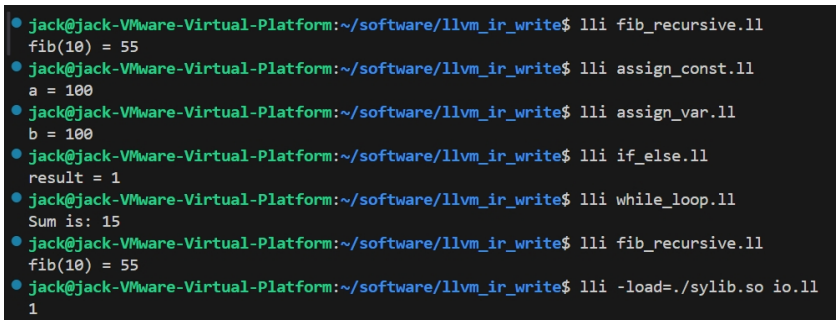



图 6: 运行结果验证

## 四、 任务三：RISC-V 汇编实现斐波那契数列的设计与优化

### （一） 基础版本实现与分析

#### 1. 程序结构与指令解析

RISC-V 汇编程序的基本结构包含段声明、符号定义和指令序列。以下是完整的基础版本实现：

Listing 1: 斐波那契数列基础版本实现

```
    .text                   # 代码段声明
    .align 2                # 4字节对齐(2^2=4)
    .globl main             # main函数全局可见
main:
    # 函数序言
    addi    sp, sp, -32     # 栈指针下移32字节
    sd      ra, 24(sp)      # 保存返回地址(8字节)
    sd      s0, 16(sp)      # 保存帧指针(8字节)
    addi    s0, sp, 32      # 设置帧指针

    # 初始化变量
    li      s1, 0           # s1 = a = 0
    li      s2, 1           # s2 = b = 1
    li      s3, 1           # s3 = i = 1

```

```
    # 读取输入
    call    getint
    mv      s4, a0          # s4 = n

    # 输出初始值a
    mv      a0, s1
    call    putint
    li      a0, 10          # ASCII换行符
    call    putch

    # 输出初始值b
    mv      a0, s2
    call    putint
    li      a0, 10
    call    putch

.L_loop:
    bge     s3, s4, .L_end  # if (i >= n) exit

    # 计算下一个数
    add     t0, s1, s2      # t0 = a + b
    mv      s1, s2          # a = b
    mv      s2, t0          # b = t0

    # 输出新值
    mv      a0, s2
    call    putint
    li      a0, 10
    call    putch

    addi    s3, s3, 1       # i++
    j       .L_loop         # 无条件跳转

.L_end:
    li      a0, 0           # 返回值0
    ld      ra, 24(sp)      # 恢复返回地址
    ld      s0, 16(sp)      # 恢复帧指针
    addi    sp, sp, 32      # 恢复栈指针
    ret                     # 返回
```

### 2. 指令详解与执行流程

**段声明与对齐：** `.text` 指令告知汇编器后续内容为可执行代码段。`.align 2` 确保指令地址为 4 字节对齐，这是 RISC-V 32 位定长指令的要求。对齐可以提高指令获取效率，避免跨缓存行访问。

**栈帧管理：** 函数序言通过 `addi sp, sp, -32` 在栈上分配 32 字节空间。RISC-V 的栈从高地址向低地址增长，因此栈指针减小表示分配。`sd`（Store Doubleword）指令保存 64 位寄存器

值到栈中。保存 ra（返回地址）是必要的，因为后续 `call` 指令会修改 ra 的值。

**寄存器分配策略：**程序使用 s1-s4 存储局部变量，这些是被调用者保存寄存器（callee-saved），在函数调用间保持值不变。相比之下，t0 作为临时寄存器用于中间计算，不需要保存。a0 寄存器遵循调用约定，用于传递第一个参数和返回值。

**核心算法实现：**斐波那契计算采用滑动窗口方法，仅保留最近两个数。`add t0, s1, s2` 执行 64 位加法运算，`mv`（move）伪指令实际被汇编为 `addi rd, rs, 0`，完成寄存器间数据传送。

**控制流结构：**`bge`（Branch if Greater or Equal）实现条件分支，通过比较 s3 和 s4 决定是否跳转。`j` 是无条件跳转伪指令，实际被汇编为 `jal x0, offset`，形成循环结构。

## （二） 优化策略与实现

### 1. 寄存器优化版本

寄存器优化版本的核心改进在于减少重复指令的执行：

Listing 2: 寄存器优化关键改进

```
    # 预存储常量
    li      s5, 10          # 换行符常量

    # 循环中复用
.L_loop:
    # ... 计算部分相同 ...

    mv      a0, s2
    call    putint
    mv      a0, s5          # 复用预存的换行符
    call    putch
```

原版本每次输出都执行 `li a0, 10` 加载立即数，优化版预先将常量 10 存储在 s5 寄存器中。`li` 伪指令对于小立即数会扩展为 `addi rd, x0, imm`，但对于大立即数可能需要 `lui` 和 `addi` 两条指令。通过寄存器缓存，将循环内的立即数加载替换为寄存器传送，减少了指令译码压力。

### 2. 循环展开版本

循环展开通过在一次迭代中执行多次循环体，减少分支开销：

Listing 3: 循环展开核心结构

```
.L_loop2:
    bge     s3, s4, .L_end

    # 第一次计算
    add     s5, s1, s2
    mv      s1, s2
    mv      s2, s5
    # 输出...
    addi    s3, s3, 1
    bge     s3, s4, .L_end    # 中间检查

    # 第二次计算(展开)
```

```
13      add     s5, s1, s2
14      mv      s1, s2
15      mv      s2, s5
16      # 输出...
17      addi    s3, s3, 1
18      j       .L__loop2
```

展开后每两次计算只需一次循环跳转，理论上减少 50% 的分支指令。中间插入的 `bge` 检查确保不会多输出数据，这是处理非偶数迭代的必要措施。

最后结果如下图7：

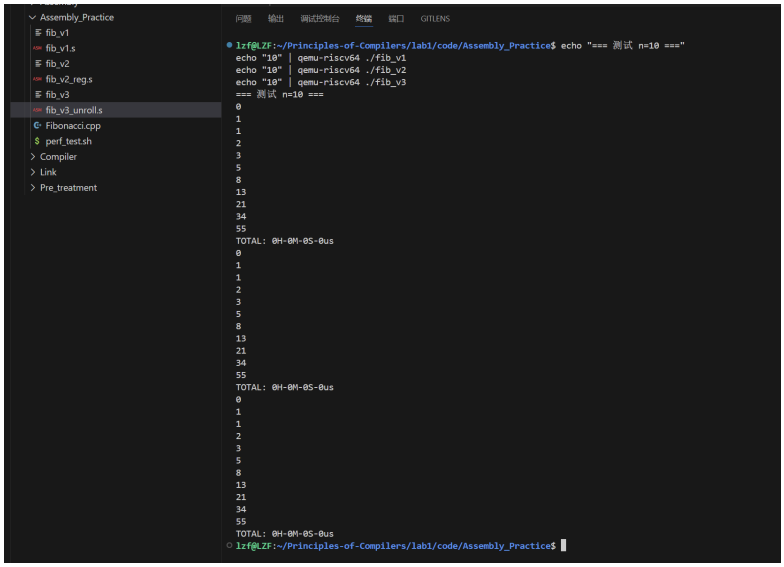



图 7: 结果正确性验证

## （三） 性能分析与对比

### 1. 静态指令分析

表 25: 三个版本的静态指令统计

| 指标 | 基础版 v1 | 寄存器优化 v2 | 循环展开 v3 |
|---|---|---|---|
| main 函数指令数 | 39 | 47 | 57 |
| 栈帧大小 (字节) | 32 | 48 | 48 |
| 使用的 s 寄存器 | 4 个 | 5 个 | 5 个 |
| 循环体指令数 | 11 | 10 | 22 |

基础版本具有最少的静态指令数，但循环体内包含 `li` 指令。寄存器优化版增加了 8 条指令用于额外的寄存器保存/恢复，但循环体减少了 1 条指令。循环展开版指令数最多，循环体扩大一倍，但循环控制指令的执行频率减半。

### 2. 动态执行性能

实验数据呈现三个明显特征：

表 26: 不同输入规模下的执行时间对比

| 输入 n | v1(s) | v2(s) | v3(s) | v2 提升 (%) | v3 提升 (%) |
|---|---|---|---|---|---|
| $10^5$ | 0.092 | 0.088 | 0.087 | 4.3 | 5.4 |
| $10^6$ | 0.486 | 0.457 | 0.438 | 6.0 | 9.9 |
| $10^7$ | 4.659 | 3.403 | 3.877 | 27.0 | 16.8 |

**小规模数据（$n = 10^5$）：**优化效果不明显，三个版本性能接近。此时 I/O 操作（getint/putint/putch）占主导，循环优化的影响被稀释。函数调用开销包括参数传递、跳转和返回，这些开销在总执行时间中占比较大。

**中等规模（$n = 10^6$）：**循环展开版本表现最佳。循环执行次数足够多使优化效果显现，同时代码体积增加尚未造成缓存压力。分支预测器能够有效学习循环模式，减少的分支指令直接转化为性能提升。

**大规模数据（$n = 10^7$）：**出现性能反转，寄存器优化版超过循环展开版。这表明循环展开增加的 57 条指令可能导致指令缓存（I-Cache）性能下降。现代处理器的 L1 指令缓存通常为 32KB-64KB，过大的循环体会增加缓存缺失率，抵消分支减少的收益。

#### 3. 微架构影响分析

RISC-V 指令集的规整性使得优化效果可预测。每条指令固定 32 位长度，简化了取指和译码逻辑。寄存器文件包含 32 个 64 位通用寄存器，提供了充足的寄存器资源进行变量缓存。

QEMU 用户模式模拟器通过动态二进制翻译执行 RISC-V 指令，其性能特征与真实硬件存在差异。真实处理器的分支预测器、乱序执行和多级缓存会产生不同的优化效果。例如，现代处理器的分支预测准确率通常超过 95%，使得循环展开的收益降低。

### （四） 结论

通过三个版本的实现和测试，观察到汇编级优化需要平衡多个因素：指令数量、寄存器压力、缓存效应和分支预测。简单的寄存器优化（常量预存储）在大规模数据上表现最佳，证明了局部性原理的重要性。循环展开作为经典优化技术，在中等规模数据上有效，但需要考虑代码膨胀的副作用。

## 五、 总结

本实验通过系统性分析编译器各阶段的工作机制，建立了从源代码到可执行文件的完整技术认知体系。

### （一） 编译流程的递进式理解

预处理器执行纯文本级操作，将包含宏定义和头文件引用的源码转换为纯 C++ 代码，关键发现是单个 `#include` 指令能够触发 1,698 倍的代码膨胀，反映了现代 C++ 标准库的复杂依赖结构。词法分析器将字符流分解为具有类型、词素、位置属性的 token 序列，实现了从非结构化文本到结构化符号的转换。语法分析器基于 token 流构建层次化的抽象语法树，其中表达式的运算符优先级通过树的深度体现，左值与右值的区分通过节点属性实现。语义分析阶段通过类型检查、作用域解析、符号表管理确保程序的静态正确性，关键机制包括隐式类型转换的插入和定义性赋值分析。

中间代码生成与优化阶段：LLVM IR 的 SSA 形式通过 phi 节点管理控制流汇聚点的变量状态，mem2reg pass 将栈变量提升为虚拟寄存器，实现 92.9% 的内存分配减少和 100% 的存储指令消除。代码膨胀 74% 的现象源于函数内联的激进策略，C++ 流操作的简单调用被展开为包含字符处理、异常检查、缓冲区管理的复杂序列，基本块数量从 3 个增长至 17 个。

代码生成阶段实现了从机器无关的中间表示到特定目标架构机器码的转换。RISC-V 后端通过寄存器分配算法将虚拟寄存器映射到物理寄存器，store 指令减少 52.9% 直接反映了这种映射的效率。指令数量增加 48.2% 但文件大小减少 3.0% 的反常现象，揭示了指令编码密度的差异：优化后的代码更多使用短编码的压缩指令集。

## （二） 优化策略的量化验证

O0 版本保持源码结构的直观映射，每个变量对应独立的栈分配，便于调试但牺牲运行效率。O2 版本通过激进的变换策略，将命令式的循环结构转换为高度优化的函数式 SSA 表示，消除了冗余的内存访问并为后续的寄存器分配创造了优化空间。

LLVM IR 编程实验验证了静态单赋值形式的理论基础。手工构建的 phi 节点精确管理了变量在控制流汇聚点的状态合并，基本块间的跳转指令实现了高级语言控制结构到低级跳转指令的映射。递归函数的实现展现了函数调用栈的管理机制，条件分支通过 icmp 和 br 指令的组合实现了布尔逻辑到机器指令的转换。

RISC-V 汇编优化实验的量化结果展现了微架构对优化效果的影响。寄存器优化通过常量预存储减少了循环内的立即数加载，在大规模数据上实现 27.0% 的性能提升。循环展开在中等规模数据上表现最佳，但在大规模时出现性能反转，这种现象反映了代码膨胀对指令缓存局部性的负面影响。指令缓存的容量限制使得过度的代码展开最终降低了整体性能。

## （三） 编译器设计原理的系统认知

编译器的分层架构体现了复杂系统的模块化设计原则。前端专注于语言特定的分析和验证，后端处理目标架构相关的优化和代码生成，中间层提供统一的抽象接口。这种分离使得编译器能够支持多种源语言和目标架构的组合，同时为优化算法提供了稳定的执行环境。

SSA 形式作为现代编译器的核心中间表示，通过变量的静态单赋值约束简化了数据流分析的复杂度。phi 节点的引入解决了控制流汇聚点的变量状态问题，使得各种优化 pass 能够基于清晰的 def-use 链进行变换。优化 pass 的顺序执行和相互作用机制展现了编译器优化的系统性：每个 pass 在前一个 pass 的基础上进行增量改进，多轮迭代最终收敛到性能最优的代码形式。

链接器作为编译工具链的最后环节,解决了模块化编程的符号解析和地址重定位问题。PLT/-GOT 机制实现的延迟绑定策略体现了系统设计中时间换空间的权衡：通过运行时的间接跳转开销换取了共享库的内存节约和版本管理灵活性。静态链接与动态链接的 1,871 倍文件大小差异量化了这种权衡的代价。

通过对编译器各阶段工作机制的深入分析，建立了对现代编译技术的系统性理解。编译器不仅是源码到机器码的简单翻译器，更是一个复杂的优化引擎，通过多层次的分析和变换实现了程序性能的显著提升。这种理解为后续的编译器设计、程序优化和系统性能调优提供了坚实的理论基础。

# 参考文献

[1] GCC Team. *The C Preprocessor*. Free Software Foundation, 2023. GCC 11.4.0 Documentation.