南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译系统原理实验报告

编译系统原理实验一

李子凝

年级：2023 级

专业：信息安全

指导教师：王刚

2025 年 9 月 29 日

摘要

本实验通过系统运用 GCC 和 Clang 编译器的命令行选项，深入研究了从 C 语言源代码到 ARM64 可执行文件的完整编译过程。实验以斐波那契数列生成程序为案例，详细分析了预处理、词法分析、语法分析、语义分析、中间代码生成、代码优化和目标代码生成等各编译阶段的核心机制。通过对比不同优化级别下的中间表示（GIMPLE、SSA、LLVM IR）和汇编代码，揭示了编译器的工作原理和优化策略。进一步地，实验通过手动编写符合 SysY 语言规范的 LLVM IR 程序和 ARM64 汇编程序，验证了中间表示和目标代码的正确性，并使用 QEMU 在 x86 平台上成功运行了 ARM64 架构的可执行文件。实验结果表明，现代编译系统通过多阶段协同工作，能够高效地将高级语言代码转换为优化的机器代码，同时保持跨平台兼容性。

关键字：编译系统 GCC Clang 中间表示 LLVM IR ARM64 代码优化

目录

一、 分工 **1**

二、 实验概述 **1**

（一） 实验目的 1

（二） 实验环境 1

（三） 实验内容与方法 2

1. 编译过程分析 2

2. 实践编程验证 2

三、 了解编译系统 **2**

（一） 编译系统的组成 2

（二） 预处理器 3

1. 预处理器的作用 4

2. 预处理阶段的输出结果分析 4

（三） 编译器 5

1. 词法分析 5

2. 语法分析 6

3. 语义分析 7

4. 中间代码生成 8

5. 代码优化 11

6. 目标代码生成 13

7. ARM 架构目标代码生成的平台选择问题 16

（四） 汇编器 16

1. 交叉编译 16

2. 反汇编 16

3. 目标文件的结构分析 17

4. 汇编器的工作原理和功能总结 19

（五） 链接器 20

1. 链接器的工作原理 20

2. 不同链接参数对程序的影响 21

四、 LLVM IR 程序编写 **22**

（一） 全局变量声明 22

（二） 外部函数声明 22

（三） 简单函数：加法运算 22

（四） 主函数实现 23

1. 局部变量声明与赋值 23

2. 数值运算和赋值语句 23

3. 条件分支语句 23

4. 循环语句 24

5. 类型转换 24

6. 函数调用 24

7. 库函数调用（I/O 操作） 25

8. 数组操作 . . . 25
9. 浮点数操作 . . . 25
（五） LLVM IR 代码的验证与运行（ARM64 平台） . . . 25

五、 ARM64 汇编程序编写 27
（一） 架构声明和全局符号定义 . . . 27
（二） 加法函数实现 . . . 27
（三） 外部库函数声明 . . . 28
（四） 全局变量定义 . . . 28
（五） 主函数实现 . . . 28
1. 函数序言和栈帧分配 . . . 28
2. 变量初始化和算术运算 . . . 28
3. 条件分支实现 . . . 29
4. 循环结构实现 . . . 29
5. 类型转换 . . . 29
6. 函数调用 . . . 29
7. 库函数调用和数组操作 . . . 30
8. 函数收尾 . . . 30
（六） 汇编代码验证和运行 . . . 30

A SysY 示例程序代码 33

B LLVM IR 示例程序代码 34

C ARM64 示例程序代码 36

一、 分工

- **李子凝**：负责 ARM 汇编程序
- **李朝阳**：负责 LLVM IR 程序

二、 实验概述

编译系统是连接高级编程语言和计算机硬件的重要桥梁，它将程序员编写的高级语言代码转换为机器可以直接执行的二进制代码。现代编译器采用多阶段处理架构，通过复杂而精细的转换过程，实现了从源代码到可执行程序的完整转换。本实验通过系统地运用编译器的命令行选项，深入研究编译过程中各阶段的输出结果与源程序之间的内在关联，以直观的方式展现编译器的工作原理和优化机制。

(一) 实验目的

1. **掌握编译系统架构**：系统理解编译器的主要组成部分（预处理器、编译器前端、优化器、编译器后端、汇编器、链接器）及其协同工作机制。

2. **熟悉编译过程各阶段**：深入掌握词法分析、语法分析、语义分析、中间代码生成、代码优化和目标代码生成等核心编译阶段的技术原理和实现方法。

3. **熟练使用编译工具链**：掌握 GCC 和 Clang 编译器的命令行选项，能够获取和分析各编译阶段的输出结果，包括预处理文件、词法单元、抽象语法树、中间表示和汇编代码等。

4. **培养实践分析能力**：通过手动编写 LLVM IR 和 ARM 汇编代码，加深对中间表示和目标代码的理解，培养编译系统分析和优化的实践能力。

(二) 实验环境

实验在标准化的开发环境中进行，具体配置如下：

- **操作系统**：Ubuntu 22.04 LTS（Linux 内核版本 5.15+）
- **编译器工具链**：
  - GCC 11.4.0（主要编译工具）
  - Clang 14.0+（LLVM 工具链，用于中间代码分析）
  - aarch64-linux-gnu-gcc（ARM64 交叉编译工具链）
- **目标架构支持**：
  - x86-64（主机架构，用于编译过程分析）
  - ARMv8-A（目标架构，用于交叉编译实验）
- **辅助工具**：
  - QEMU 6.2+（用于 ARM 程序仿真运行）
  - GNU Binutils 2.38+（objdump、readelf 等二进制分析工具）
  - Graphviz（用于控制流图等可视化）
  - Make 构建工具（用于自动化编译流程）
- **开发环境**：VSCode with C/C++ 扩展，配合终端进行命令行操作

(三) 实验内容与方法

本实验采用“自顶向下、逐层深入”的研究方法，以斐波那契数列程序为案例，系统分析编译过程各环节。主要内容包括：

1. 编译过程分析

预处理分析：使用 gcc -E 选项研究头文件包含、宏展开等预处理操作
词法语法分析：通过 Clang 工具生成词法单元和抽象语法树，分析源代码解析过程
中间代码分析：利用 gcc -fdump-tree 选项研究 GIMPLE、SSA 等中间表示形式
代码优化研究：对比不同优化级别下的代码变换，理解编译器优化策略
目标代码生成：分析中间表示到 x86-64/ARM64 汇编的转换过程

2. 实践编程验证

LLVM IR 编程：手动编写 LLVM IR 代码，理解中间表示设计原理
ARM 汇编编程：编写 ARM64 汇编程序，掌握目标代码编写技巧
交叉编译验证：使用交叉编译工具链生成可执行文件，通过 qemu 验证运行结果

三、 了解编译系统

编译系统是一个复杂的软件系统，由多个相互协作的组件组成，它们共同完成将高级语言代码转换为机器代码的任务。本节将详细介绍编译系统的各个组成部分及其在编译过程中的作用。

(一) 编译系统的组成

编译系统主要由以下几个部分组成：

1. 预处理器（Preprocessor）：负责处理源文件中的预处理指令（如 #include、#define 等），生成预处理后的源代码。

2. 编译器（Compiler）：将预处理后的源代码转换为中间表示（IR），并进行一系列优化，最终生成汇编代码。编译器通常分为前端和后端两个主要部分：

前端（Frontend）：负责词法分析、语法分析、语义分析，并生成中间表示。

优化器（Optimizer）：对中间表示进行各种优化，如常量折叠、死代码消除、循环优化等。

后端（Backend）：将优化后的中间表示转换为目标架构的汇编代码。

汇编器（Assembler）：将汇编代码转换为机器代码，生成目标文件（.o 文件）。

链接器（Linker）：将多个目标文件和库文件链接在一起，生成最终的可执行文件。

编译系统的工作流程可以用下图表示：

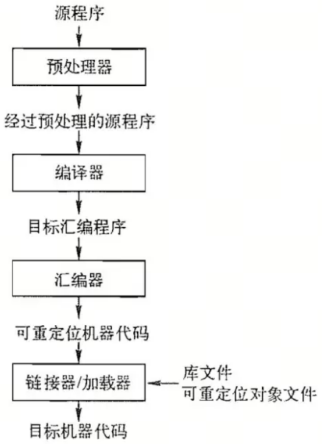
接下来我将以斐波那契数列的 C 语言源码为例，逐步展现编译的各个过程。以下是本次实验使用的 C 源代码文件的完整内容：

```c
#include <stdio.h>

int main()
{
    // 变量声明
    int a, b, i, t, n;  // a和b存储斐波那契数列的当前两项，i是计数器，t是临时变量，n是用户输入的项数

```

图 1: 编译系统工作流程示意图




```
    // 初始化斐波那契数列的前两项
    a = 0;  // 斐波那契数列第一项
    b = 1;  // 斐波那契数列第二项
    i = 1;  // 计数器初始化为1（因为已经有两项了）

    // 获取用户输入的斐波那契数列项数
    scanf("%d", &n);

    // 输出前两项
    printf("%d\n", a);  // 输出第一项：0
    printf("%d\n", b);  // 输出第二项：1

    // 循环计算并输出剩余的斐波那契数列项
    while (i < n)  // 当计数器小于总项数时继续循环
    {
        t = b;          // 临时保存b的值（前一项）
        b = a + b;      // 计算新的斐波那契数：当前项 = 前两项之和
        printf("%d\n", b);  // 输出新计算的斐波那契数
        a = t;          // 更新a为前一项的值
        i = i + 1;      // 计数器加1
    }

    return 0;
}
```

(二) 预处理器

预处理器是编译过程的第一个阶段，它主要负责处理源文件中的预处理指令，并生成预处理后的源代码。预处理器的工作不涉及语法分析或语义分析，它只是对源代码进行文本替换和处理。

在本次实验中，通过执行预处理命令，我成功生成了预处理后的文件`main.i`。

1. 预处理器的作用

预处理器的主要作用包括：

1. 文件包含处理：处理 #include 指令，将指定的头文件内容插入到源文件中。

2. 宏展开：处理 #define 指令，将宏定义展开为其对应的值或代码片段。

3. 条件编译：处理 #ifdef、#ifndef、#if、#else、#elif、#endif 等指令，根据条件决定是否包含某些代码。

4. 行控制：处理 #line 指令，用于控制编译器的行号输出和错误报告。

5. 错误处理：处理 #error 和 #pragma 指令，生成编译时错误信息或向编译器提供特定的指令。

2. 预处理阶段的输出结果分析

通过使用编译器的-E 选项，我们可以只运行预处理器，生成预处理后的源代码文件（通常以.i 为扩展名）。

在本次实验中，我执行了预处理命令 gcc -E -v main.c -o main.i，并通过查看输出信息了解到：现代 GCC 编译器通常直接调用其前端组件 cc1 完成预处理操作，而非独立的 cpp 程序。从终端输出中可以看到具体的调用过程：

```
COLLECT_GCC_OPTIONS='-E' '-v' '-o' 'main.i' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/11/cc1 -E -quiet -v -imultiarch x86_64-linux-
    gnu main.c -o main.i -mtune=generic -march=x86-64 -fasynchronous-unwind-
    tables -fstack-protector-strong -Wformat -Wformat-security -fstack-clash
    -protection -fcf-protection -dumpbase main.c -dumpbase-ext .c
```

为了验证独立 cpp 程序的存在性，我依次输入并执行了以下命令：

```
$ which cpp
/usr/bin/cpp

$ cpp --version
cpp (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

从结果可以看出，系统中确实存在独立的 cpp 程序，但现代 GCC 为了效率和集成度，将预处理功能整合到了编译器前端 cc1 中，不过为了兼容性，系统仍然提供独立的 cpp 程序。

以下是对生成的`main.i`文件的详细分析：

预处理后的 main.i 文件主要包含以下内容：

1. 头文件展开：所有通过 #include 指令包含的头文件（如 stdio.h）的内容都被完整地插入到预处理后的文件中。这些头文件包含了函数声明、宏定义、类型定义等内容。从 main.i 文件的开头部分可以看到头文件展开的情况：

```
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "main.c"
# 1 "/usr/include/stdio.h" 1 3 4
```

```
\# 27 "/usr/include/stdio.h" 3 4
\# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
\# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
\# 1 "/usr/include/features.h" 1 3 4
// ... 更多头文件展开内容 ...
```

2. 宏展开：所有在源文件中定义的宏都被展开为其对应的值或代码片段。

3. 条件编译处理：根据条件编译指令，符合条件的代码段被保留，不符合条件的代码段被删除。

4. 注释删除：源文件中的注释被预处理器删除，不包含在预处理后的文件中。从 main.i 文件的最后部分可以看到，原始 C 代码中的注释已经被删除：

```
    scanf("%d", &n);

    printf("%d\n", a);
    printf("%d\n", b);

    while (i < n)
    {
        t = b;
        b = a + b;
        printf("%d\n", b);
        a = t;
        i = i + 1;
    }

    return 0;
}
```

5. 行号标记：预处理器会在预处理后的文件中插入 #line 指令，用于跟踪源代码的行号，以便编译器在报告错误时能够指出正确的位置。

(三) 编译器

编译器是编译系统的核心组件，负责将预处理后的源代码转换为目标架构的汇编代码。

1. 词法分析

词法分析是编译器处理源代码的第一个阶段，其主要任务是将源代码文本分割成一系列词法单元（Tokens）。实验采用 Clang 编译器的词法分析功能，执行以下命令：

```
clang -E -Xclang -dump-tokens main.c
```

通过运行该命令，我们得到了 main.c 文件的词法单元输出。词法分析器将源代码分解为不同类型的词法单元，包括关键字、标识符、运算符、常量和标点符号等。以下是对几个典型部分的详细分析：

1. 关键字识别 词法分析器能够准确识别 C 语言中的关键字，如 int、return 等。在斐波那契数列程序中，int 关键字用于声明整数类型变量，而 return 关键字用于函数返回。从词法单元输出中可以观察到：

```
identifier 'return' [StartOfLine] [LeadingSpace] Loc=<main.c:30:5>
```

这里的 identifier 'return' 表明词法分析器识别出了 return 关键字，同时记录了其在源代码中的位置（第 30 行第 5 列）。

2. 标识符识别 标识符是程序员定义的变量名、函数名等。在我们的程序中，有 main、a、b、i、t、n、printf 和 scanf 等标识符。

```
identifier 'main' [StartOfLine] Loc=<main.c:2:1>
identifier 'a' [StartOfLine] [LeadingSpace] Loc=<main.c:6:9>
identifier 'b' [StartOfLine] [LeadingSpace] Loc=<main.c:7:9>
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:14:9>
```

这些输出显示了词法分析器对不同标识符的识别结果，包括它们在源代码中的位置信息。

3. 运算符与赋值操作 程序中包含多种运算符，如赋值运算符 =、加法运算符 + 和关系运算符 < 等。词法分析器能够准确识别这些运算符：

```
equal '=' [LeadingSpace] Loc=<main.c:9:7>
plus '+' [LeadingSpace] Loc=<main.c:24:15>
less '<' [LeadingSpace] Loc=<main.c:21:14>
```

4. 常量识别 程序中包含整数常量（如 0、1）和字符串常量（如"%d\n"）。

```
numeric_constant '0' [LeadingSpace] Loc=<main.c:9:9>
numeric_constant '1' [LeadingSpace] Loc=<main.c:10:9>
string_literal '"%d\n"' Loc=<main.c:14:16>
```

5. 标点符号识别 程序中的标点符号如分号、逗号、括号和花括号等也是重要的词法单元。

```
l_paren '(' Loc=<main.c:2:6>
r_paren ')' Loc=<main.c:2:7>
l_brace '{' [StartOfLine] [LeadingSpace] Loc=<main.c:3:5>
r_brace '}' [StartOfLine] Loc=<main.c:31:1>
semi ';' Loc=<main.c:9:10>
comma ',' Loc=<main.c:6:11>
```

通过对词法单元的详细分析，我们可以看出词法分析器如何将源代码文本分解为一系列有意义的词法单元，为后续的语法分析提供基础。每个词法单元都包含了类型信息和位置信息，这对于后续的编译阶段和错误报告至关重要。

2. 语法分析

语法分析阶段根据编程语言的语法规则，将词法单元序列组合成抽象语法树（Abstract Syntax Tree，AST）。实验通过以下命令对源代码进行语法分析：

```
clang -E -Xclang -ast-dump main.c
```

终端输出信息展示了代码的语法组成。例如，main 函数的 AST 结构显示了函数声明、变量声明、赋值语句、函数调用和循环结构等：

```
-FunctionDecl 0x3d292390 <main.c:3:1, line:31:1> line:3:5 main 'int
    ()'-CompoundStmt 0x3d293588 <line:4:1, line:31:1>
    |-DeclStmt 0x3d2926e0 <line:6:5, col:22>
```

```
    | |-VarDecl 0x3d292448 <col:5, col:9> col:9 used a 'int'
    | |-VarDecl 0x3d2924c8 <col:5, col:12> col:12 used b 'int'
    | |-VarDecl 0x3d292548 <col:5, col:15> col:15 used i 'int'
    | |-VarDecl 0x3d2925c8 <col:5, col:18> col:18 used t 'int'
    | -VarDecl 0x3d292648 <col:5, col:21> col:21 used n 'int'|-BinaryOperator 0x3d292738
      <line:9:5, col:9> 'int' '='| |-DeclRefExpr 0x3d2926f8 <col:5> 'int' lvalue Var 0x3d292448 'a'
      'int'|-IntegerLiteral 0x3d292718 <col:9> 'int' 0
    // ... 更多AST节点 ...
    -WhileStmt 0x3d293538 <line:21:5, line:28:5>|-BinaryOperator 0x3d293160 <line:21:12, col:16>
      'int' '<'| |-ImplicitCastExpr 0x3d293130 <col:12> 'int' <LValueToRValue>| |-
      DeclRefExpr 0x3d2930f0 <col:12> 'int' lvalue Var 0x3d292548 'i' 'int'
      | -ImplicitCastExpr 0x3d293148 <col:16> 'int' <LValueToRValue>|-DeclRefExpr 0
      x3d293110 <col:16> 'int' lvalue Var 0x3d292648 'n' 'int'
      -CompoundStmt 0x3d293500 <line:22:5, line:28:5>// ... 循环体 AST 节点...
```

AST 以清晰的层次结构展示了斐波那契数列程序的语法组成，为后续的语义分析提供了基础结构。

3. 语义分析

语义分析阶段对语法分析生成的 AST 树进行语义检查，确保源代码的语义正确性。实验基于语法分析生成的 AST 树，结合-ast-dump 参数提供的语义信息进行初步语义分析。

语义分析器的核心工作包括：

1. 符号表管理与作用域分析：维护不同作用域的符号表，确保变量定义和使用的一致性。如 AST 中展示的：

```
|-VarDecl 0xe02c448 <col:5, col:9> col:9 used a 'int'
```

这里 used 标记表明变量被正确使用，而类型信息‘int’则是语义分析阶段绑定的结果。

2. 类型检查：验证操作数类型是否匹配，确保类型安全。例如赋值语句的类型检查：

```
BinaryOperator 0xe02c738 <line:9:5, col:9> 'int' '='
 |-DeclRefExpr 0xe02c6f8 <col:5> 'int' lvalue Var 0xe02c448 'a' 'int'
 -IntegerLiteral 0xe02c718 <col:9> 'int' 0
```

语义分析器确认了左值 a 和右值 0 都是 int 类型，确保了赋值的合法性。

3. 函数调用检查：验证函数调用的参数类型和数量是否符合函数声明。如 scanf 函数调用：

```
CallExpr 0xe02ce80 <line:14:5, col:19> 'int'
 |-ImplicitCastExpr 0xe02ce68 ... <FunctionToPointerDecay>
 | -DeclRefExpr ... Function 0xe021a10 'scanf' 'int (const char *restrict, ...)'|-ImplicitCastExpr ...
    'const char *' <NoOp>|-StringLiteral 0xe02c878 ... "%d"
 -UnaryOperator 0xe02cdd8 ... 'int *' prefix
    '&'-DeclRefExpr ... 'int' lvalue Var 0xe02c648 'n' 'int'
```

语义分析器检查了字符串字面量“%d”和指针 &n 与 scanf 函数原型的兼容性。

为深入理解语义检查机制，我构建了错误 C 代码文件 error.c，通过编译分析其错误：

```c
int main() {
    int x = 5;

    {
        int x = 10;  // 正确：内层作用域可以重新定义
```

```
        printf("Inner x: %d\n", x);
    }

    // 语义错误：重复定义（在同一作用域）
    int x = 20;  // 错误：重定义'int x'

    // 语义错误：使用未声明的变量
    printf("z: %d\n", z);  // 错误：'z'未声明

    return 0;
}
```

编译该代码产生以下错误：

```
error.c: In function 'main':
error.c:12:9: error: redefinition of 'x'
   12 |     int x = 20;  // 错误：重定义'int x'
      |         ^
error.c:4:9: note: previous definition of 'x' with type 'int'
    4 |     int x = 5;
      |         ^
error.c:15:23: error: 'z' undeclared (first use in this function)
   15 |     printf("z: %d\n", z);  // 错误：'z'未声明
      |                       ^
```

通过分析错误代码的编译输出，我们观察到编译器如何检测两种典型的语义错误：

1. 重复定义错误：编译器在同一作用域中检测到相同名称的变量已存在，并提供了先前定义的位置信息。

2. 未声明标识符错误：编译器在所有可见作用域中查找不到标识符 z 的定义，并报告错误。

错误的 AST 节点会被特殊标记：

```
// 重复定义的变量被标记为invalid
|-VarDecl 0x62085c0 <col:5, col:9> col:9 invalid x 'int'

// 包含未声明变量的表达式被标记为contains-errors
|-CallExpr 0x620d3e8 <line:15:5, col:24> 'int' contains-errors
```

这体现了编译器的错误恢复机制，即使存在错误，编译器仍会尝试分析剩余代码并生成包含错误标记的完整 AST。

语义分析的重要性体现在：语法正确并不意味着语义正确，它能在编译时发现潜在逻辑错误，确保符号定义明确且类型安全。

4. 中间代码生成

中间代码生成阶段将经过语义分析的 AST 转换为中间表示（IR），这是一种介于源代码和目标代码之间的表示形式，通常具有平台无关性。实验从两个方面探究了中间代码生成：

1. GCC 中间代码生成：使用以下命令生成多阶段中间代码（GIMPLE/SSA）：

```
gcc -O0 -fdump-tree-all-graph main.c
```

通过分析生成的中间代码文件和 Graphviz 可视化的.dot 文件，我们系统地研究了 GCC 的中间表示形式及其转换过程。GCC 的编译过程包含以下关键阶段：

初始中间表示（Original）：a-main.c.005t.original 文件展示了非常接近原始 C 代码的中间表示，保留了原始变量名和基本结构，但已将 while 循环转换为 if 条件判断和 goto 语句的组合。

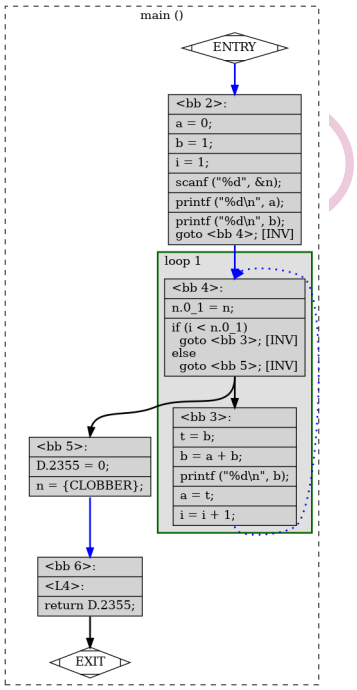
GIMPLE 中间表示：a-main.c.006t.gimple 文件展示了 GCC 的核心中间表示形式，具有三地址码特性，引入了新的临时变量和异常处理框架。

控制流图（CFG）生成：a-main.c.015t.cfg 和 a-main.c.015t.cfg.dot 文件展示了程序的控制流结构，将程序划分为多个基本块，并识别出了循环结构。CFG 的可视化表示清晰地展示了各基本块之间的连接关系，用不同样式的箭头表示不同类型的控制流边。

SSA 形式转换：a-main.c.023t.ssa 文件展示了静态单赋值形式的中间表示，为每个变量的每次赋值创建了新版本（如 a_6、a_18），并引入 PHI 节点处理控制流汇合点的变量值选择。

图2展示了编译过程中某一阶段的控制流图可视化结果，清晰地标识了基本块和循环结构。

图 2: GCC 编译器生成的控制流图可视化示例



通过分析这些中间代码文件，我们可以清晰地看到 GCC 编译器的工作流程：从源代码解析到 GIMPLE 生成，再到控制流图构建和 SSA 转换，每个阶段都有其特定的优化目标和转换策略。

2. LLVM IR 生成：通过以下命令生成平台无关的 LLVM IR 中间代码：

```
clang -S -emit-llvm main.c -o main.ll
```

LLVM IR 具有以下主要特点：

静态单赋值 (SSA) 形式：每个变量只能被赋值一次，每次新赋值都会使用新的编号（如%1、%2、%3 等）

强类型系统：每个操作都明确指定了操作数的类型（如 add nsw i32 %18, %19）

显式内存管理：明确区分内存中的值和寄存器中的值，使用 alloca、load 和 store 指令进行内存操作

基于基本块的控制流表示：使用基本块和跳转指令表示控制流

平台无关与目标特定信息分离：通过 target datalayout 和 target triple 提供目标平台特定信息

生成的 main.ll 文件展示了 LLVM IR 的基本结构：

```
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [3 x i8] c"%d", align 1
@.str.1 = private unnamed_addr constant [4 x i8] c"%d\n", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4  ; a = 0
  %3 = alloca i32, align 4  ; b = 1
  %4 = alloca i32, align 4  ; i = 1
  %5 = alloca i32, align 4  ; t
  %6 = alloca i32, align 4  ; n
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  store i32 1, i32* %3, align 4
  store i32 1, i32* %4, align 4
  %7 = call i32 (i8*, ...) @__isoc99_scanf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %6)
  // ... 更多LLVM IR指令 ...
}
```

为验证 LLVM IR 的平台无关性，实验针对 ARM 架构生成了不同版本的 IR 代码：

32 位 ARM IR：

```
clang -target arm-linux-gnueabi -S -emit-llvm main.c -o main_arm.ll
```

64 位 ARM IR：

```
clang -target aarch64-linux-gnu -S -emit-llvm main.c -o main_arm64.ll
```

根据对比分析，main_arm.ll 与 main_arm64.ll 的主要差异体现在：

1. 目标架构和数据布局：

32 位 ARM：target triple = ”armv4t-none-linux-gnueabi”

64 位 ARM：target triple = ”aarch64-unknown-linux-gnu”

两者具有不同的内存布局和对齐方式定义

2. 函数属性：

32 位 ARM：使用较简单的函数属性集合

64 位 ARM:包含更多架构特定的属性,如”frame-pointer”=”non-leaf” 和”target-features”=”+neon,+outline-atomics,+v8a”

3. 模块标志:

64 位 ARM 包含更多安全相关标志，如 branch-target-enforcement 和 sign-return-address

尽管存在这些架构特定的差异，但两个文件实现的逻辑功能完全相同，都实现了读取用户输入 n 并输出斐波那契数列前 n 项的功能。这充分体现了 LLVM IR 的平台无关性设计理念，使得同一份源代码可以方便地针对不同架构进行编译和优化。

5. 代码优化

代码优化阶段对中间表示进行各种优化，以提高生成代码的性能、减小代码大小或降低能耗。实验基于 ARM64 架构的 LLVM IR 代码进行了深入的优化分析，揭示了优化命令的实际效果和应用条件。

optnone 属性的影响分析 实验最初基于 ARM64 架构的 LLVM IR 代码 main_arm64.ll 进行优化实验，但发现所有优化命令未能产生预期效果。通过分析发现，原始 IR 文件中 main 函数的属性列表包含 optnone 属性:

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  // ... 其他代码 ...
}
```

optnone 属性是 LLVM 中的一种特殊标记，它会明确禁止编译器对函数进行任何优化。这就是为什么应用 Mem2Reg、DCE 等优化命令后，生成的文件与原始文件内容几乎完全相同的原因。

为了使优化命令真正生效，实验通过以下命令移除了 optnone 属性:

```
sed 's/optnone //' main_arm64.ll > main_arm64_no_optnone.ll
```

移除 optnone 属性后，重新应用各种优化命令，终于获得了明显的优化效果。

各优化命令效果详细分析

Mem2Reg 优化(内存访问转寄存器访问) Mem2Reg 优化是最具效果的优化之一,它将栈上的内存访问转换为寄存器访问,主要针对局部变量和临时值。通过分析 main_arm64_mem2reg_effective.ll 文件，我们观察到以下显著变化:

1. 内存分配大幅减少：原始代码中 6 个 alloca 指令（为变量 a、b、i、t、n 和一个未使用变量分配内存）被减少为仅 1 个 alloca（只为输入变量 n 分配内存）

2. 使用 phi 节点替代内存操作：局部变量 a、b、i 不再存储在内存中，而是通过 phi 节点（%.02、%.01、%.0）在寄存器中传递值

3. 代码结构简化：文件行数从 75 行减少到 56 行，循环体结构更紧凑

4. 常量直接内联：初始值 0 和 1 直接在 printf 调用中内联，不再通过内存加载

优化前后的代码对比:

```
; 优化前（main_arm64_no_optnone.ll）
%1 = alloca i32, align 4  ; a
%2 = alloca i32, align 4  ; b
%3 = alloca i32, align 4  ; i
%4 = alloca i32, align 4  ; t
%5 = alloca i32, align 4  ; 未使用变量
%6 = alloca i32, align 4  ; n
store i32 0, i32* %1, align 4
store i32 0, i32* %2, align 4
store i32 1, i32* %3, align 4
store i32 1, i32* %4, align 4

; 优化后（main_arm64_mem2reg_effective.ll）
%1 = alloca i32, align 4  ; 仅为n分配内存
%2 = call i32 (i8*, ...) @__isoc99_scanf(i8* noundef getelementptr inbounds
    ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %1)
%3 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8
    ], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef 0)
%4 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8
    ], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef 1)

; 循环中的phi节点使用
5:                                                ; preds = %8, %0
  %.02 = phi i32 [ 1, %0 ], [ %9, %8 ]   ; b的值在寄存器中传递
  %.01 = phi i32 [ 0, %0 ], [ %.02, %8 ]   ; a的值在寄存器中传递
  %.0 = phi i32 [ 1, %0 ], [ %11, %8 ]   ; i的值在寄存器中传递
```

死代码消除（DCE）、循环简化和 SCCP 在本实验中，DCE、循环简化和 SCCP 优化对代码的影响相对有限：

1. 死代码消除（DCE）：main_arm64_dce_effective.ll 与 main_arm64_no_optnone.ll 文件内容完全相同。这表明原始代码中没有明显的死代码（即不会执行或执行后对程序状态无影响的代码），所有变量和计算结果都在程序流程中被有效使用。

2. 循环简化：main_arm64_loop_simplify_effective.ll 与 main_arm64_no_optnone.ll 文件内容完全相同。这是因为原始代码中的循环结构已经相对简单，满足 LLVM 循环简化的基本要求，只有一个入口点和一个出口点，循环条件判断直接明了。

3. 常量传播（SCCP）：main_arm64_sccp_effective.ll 与 main_arm64_no_optnone.ll 文件内容完全相同。主要原因是虽然变量 a 初始化为 0，b 和 i 初始化为 1，但这些变量在循环中会被不断更新；关键输入 n 是用户提供的运行时变量，而非编译时常量；斐波那契数列计算逻辑依赖于前序计算结果，限制了常量传播的应用范围。

O2 级别优化 O2 是 LLVM 的中级优化级别，包含了多种优化技术的组合，展现了全面的优化效果：

1. 内存操作优化：与 Mem2Reg 类似，仅保留输入变量 n 的内存分配
2. 循环结构优化：使用循环头部预测（.lr.ph）和更高效的分支结构

3. 属性增强：为函数和参数添加了更多优化属性（如 nonnull、dereferenceable、nocapture、readonly 等）

4. 变量管理优化：phi 节点的使用更加精简，变量命名更优化

5. 条件判断优化：添加了早期退出条件（%6 = icmp sgt i32 %5, 1）避免不必要的循环

优化后的 O2 级别代码片段：

```
; 优化后的O2级别代码（main_arm64_O2_effective.ll）
%1 = alloca i32, align 4
%2 = call i32 (i8*, ...) @__isoc99_scanf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef nonnull %1)
%3 = call i32 (i8*, ...) @printf(i8* noundef nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef 0)
%4 = call i32 (i8*, ...) @printf(i8* noundef nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef 1)
%5 = load i32, i32* %1, align 4
%6 = icmp sgt i32 %5, 1  ; 早期退出条件
br i1 %6, label %.lr.ph, label %._crit_edge

.lr.ph:                                           ; preds = %0, %.lr.ph
  %.013 = phi i32 [ %.01011, %.lr.ph ], [ 0, %0 ]
  %.0912 = phi i32 [ %9, %.lr.ph ], [ 1, %0 ]
  %.01011 = phi i32 [ %7, %.lr.ph ], [ 1, %0 ]
  %7 = add nsw i32 %.013, %.01011
  %8 = call i32 (i8*, ...) @printf(i8* noundef nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %7)
  %9 = add nuw nsw i32 %.0912, 1
  %10 = load i32, i32* %1, align 4
  %11 = icmp slt i32 %9, %10
  br i1 %11, label %.lr.ph, label %._crit_edge, !llvm.loop !10
```

6. 目标代码生成

目标代码生成阶段将优化后的中间表示转换为目标架构的汇编代码。实验以 O2 级别优化后的 ARM64 IR 文件 main_arm64_O2.ll 为输入，执行以下命令生成 ARMv8 架构汇编代码：

```
llc main_arm64_O2.ll -o main64.S
```

生成的 main64.S 文件是一个完整的 64 位 ARM 汇编程序，下面结合具体代码对其逻辑进行详细分析：

函数结构分析

函数序言（Function Prologue） 函数序言负责设置栈帧、保存寄存器状态，为函数执行做准备：

```
main:                                   // @main
```

```
        .cfi_startproc
// %bb.0:
        sub     sp, sp, #48                     // 分配48字节栈空间
        stp     x29, x30, [sp, #32]             // 保存帧指针和链接寄存器
        add     x29, sp, #32                    // 设置帧指针
        .cfi_def_cfa w29, 16
        .cfi_offset w30, -8
        .cfi_offset w29, -16
```

这段代码完成了以下关键操作：

1. 通过 sub sp, sp, #48 分配 48 字节的栈空间
2. 使用 stp 指令（存储一对寄存器）将帧指针（x29）和链接寄存器（x30）保存到栈上
3. 设置新的帧指针值，使其指向栈帧底部
4. 生成 DWARF 调用帧信息（.cfi_* 指令），用于调试和异常处理

变量初始化 斐波那契数列计算所需的变量被初始化并存储在栈帧中：

```
        stur    wzr, [x29, #-4]             // a = 0（wzr是零寄存器）
        stur    wzr, [x29, #-8]             // b = 0
        mov     w8, #1
        stur    w8, [x29, #-12]             // i = 1
        mov     w8, #1
        str     w8, [sp, #16]               // t = 1
```

变量初始化的特点：

1. 使用零寄存器（wzr）将 a 和 b 初始化为 0，避免了立即数操作
2. 巧妙地使用栈帧内不同位置存储变量：a、b、i 存储在帧指针的负偏移位置，t 存储在栈指针偏移位置
3. 这里有一个微妙的实现细节：b 初始化为 0，但在后续代码中将被用于输出第一个斐波那契数 0

用户输入与输出前两项 这部分代码负责获取用户输入并输出斐波那契数列的前两个数：

```
        adrp    x0, .L.str
        add     x0, x0, :lo12:.L.str        // 设置scanf格式字符串地址
        add     x1, sp, #8                  // 设置输入变量n的地址
        bl      __isoc99_scanf              // 调用scanf函数获取用户输入

        ldur    w1, [x29, #-8]              // 加载b的值（0）
        adrp    x0, .L.str.1
        add     x0, x0, :lo12:.L.str.1      // 设置printf格式字符串地址
        bl      printf                      // 输出第一个斐波那契数（0）

        ldur    w1, [x29, #-12]             // 加载i的值（1）
        adrp    x0, .L.str.1
        add     x0, x0, :lo12:.L.str.1      // 设置printf格式字符串地址
        bl      printf                      // 输出第二个斐波那契数（1）
```

关键技术点：

1. 使用 adrp 和 add 指令组合计算字符串常量的地址，这是 AArch64 架构中位置无关代码（PIC）的标准实现方式
2. adrp 指令加载字符串所在页的基地址，add 指令添加页内偏移量
3. 函数调用使用 bl 指令（带链接的分支），自动将返回地址保存到链接寄存器（x30）
4. 函数参数传递遵循 AArch64 调用约定：前 6 个整型/指针参数使用 x0-x5 寄存器

循环实现（斐波那契数列计算核心） 循环是程序的核心部分，负责计算并输出剩余的斐波那契数：

```
.LBB0_1:                                // =>This Inner Loop Header: Depth=1
        ldr     w8, [sp, #16]           // 加载t的值
        ldr     w9, [sp, #8]            // 加载n的值
        cmp     w8, w9                  // 比较t和n（循环条件：t < n）
        b.ge    .LBB0_3                 // 如果t >= n，跳转到循环结束

// %bb.2:                               //   in Loop: Header=BB0_1 Depth=1
        ldur    w8, [x29, #-12]         // 加载i的值（当前的斐波那契数）
        str     w8, [sp, #12]           // 保存到临时位置
        ldur    w8, [x29, #-8]          // 加载b的值
        ldur    w9, [x29, #-12]         // 加载i的值
        add     w8, w8, w9              // 计算新的斐波那契数：b + i
        stur    w8, [x29, #-12]         // 保存到i
        ldur    w1, [x29, #-12]         // 加载新计算的斐波那契数
        adrp    x0, .L.str.1
        add     x0, x0, :lo12:.L.str.1  // 设置printf格式字符串
        bl      printf                  // 输出新的斐波那契数
        ldr     w8, [sp, #12]           // 加载临时保存的i的旧值
        stur    w8, [x29, #-8]          // 保存到b
        ldr     w8, [sp, #16]           // 加载t的值
        add     w8, w8, #1              // t++
        str     w8, [sp, #16]           // 保存t
        b       .LBB0_1                 // 跳回循环开始
```

函数尾声（Function Epilogue） 函数尾声负责恢复寄存器状态、释放栈空间并返回：

```
.LBB0_3:
        mov     w0, wzr                 // 设置返回值为0
        ldp     x29, x30, [sp, #32]     // 恢复帧指针和链接寄存器
        add     sp, sp, #48             // 释放栈空间
        ret                             // 返回
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
```

数据段分析 程序中使用的字符串常量定义在只读数据段：

```
        .type   .L.str,@object                  // @.str
        .section        .rodata.str1.1,"aMS",@progbits,1
```

```
.L.str:
        .asciz  "%d"                    // scanf格式字符串
        .size   .L.str, 3

        .type   .L.str.1,@object                // @.str.1
.L.str.1:
        .asciz  "%d\n"                  // printf格式字符串
        .size   .L.str.1, 4
```

7. ARM 架构目标代码生成的平台选择问题

在实验过程中，我们在 ARM 架构目标代码生成阶段遇到了平台选择的弯路。初期我们尝试使用 clang -target arm-linux-gnueabi 命令生成 32 位 ARM 架构的中间代码和汇编代码，但在后续研究中发现，我们需要聚焦于 ARMv8（64 位）架构。

修正思路是使用 clang -target aarch64-linux-gnu 命令生成 64 位 ARM 架构的代码，这样能够更好地支持现代 ARM 处理器的特性和指令集。通过这一修正，我们成功生成了针对 ARMv8 架构优化的高效汇编代码，为后续的性能分析和优化提供了基础。

(四) 汇编器

汇编器是编译过程中的关键组件，它将编译器生成的汇编代码转换为机器代码，并生成可重定位的目标文件（.o 文件）。

1. 交叉编译

我们使用以下命令对 ARM64 架构的汇编代码 main64.S 进行汇编处理：

```
aarch64-linux-gnu-gcc main64.S -c -o main.o
```

这条命令执行了以下操作：

1. 使用 aarch64-linux-gnu-gcc 工具链中的汇编器
2. 输入文件为 main64.S（ARM64 架构汇编代码）
3. -c 选项表示只进行汇编，不执行链接操作
4. -o main.o 指定输出文件名为 main.o

执行成功后，汇编器生成了 main.o 目标文件，该文件包含了机器码指令、数据、符号表和重定位信息等。

2. 反汇编

为了验证汇编器生成的目标文件内容和结构，我们使用反汇编工具对 main.o 文件进行了反编译分析。反汇编是将机器码转换回人类可读的汇编代码的过程，是理解和验证汇编过程的重要手段。

反汇编工具准备 实验中使用了 Makefile 中定义的多种反汇编和分析工具：

```
# 工具定义
OBJDUMP = aarch64-linux-gnu-objdump
READELF = aarch64-linux-gnu-readelf
NM = aarch64-linux-gnu-nm
```

```
STRINGS = strings
```

这些工具可以从不同角度分析目标文件：
objdump -d：显示反汇编代码，展示指令与机器码的对应关系
readelf -a：显示完整的 ELF 文件信息，包括头部、节表、符号表等
nm -a：显示符号表信息
strings：提取文件中的字符串常量

反汇编结果分析 根据 all_results.txt 中的反汇编结果，我们可以看到斐波那契数列程序的汇编指令与机器代码的精确对应关系：

```
0000000000000000 <main>:
   0:   d100c3ff        sub     sp, sp, #0x30                   // 分配48字节栈空间
   4:   a9027bfd        stp     x29, x30, [sp, #32]             // 保存帧指针和链接寄
       存器
   8:   910083fd        add     x29, sp, #0x20                  // 设置新的帧指针
   c:   b81fc3bf        stur    wzr, [x29, #-4]                 // 初始化变量为0
  10:   b81f83bf        stur    wzr, [x29, #-8]                 // 初始化变量为0
  14:   52800028        mov     w8, #0x1                        // #1
  18:   b81f43a8        stur    w8, [x29, #-12]                 // 存储值1
  1c:   52800028        mov     w8, #0x1                        // #1
  20:   b90013e8        str     w8, [sp, #16]                   // 存储值1
  ...
```

从反汇编结果可以看出，每条 ARM64 汇编指令都被准确编码为 4 字节的机器码，这些编码由操作码、寄存器编号、立即数等字段组成，完整实现了斐波那契数列计算程序的逻辑。反汇编结果与原始汇编代码 main64.S 的一致性验证了汇编过程的正确性。

3. 目标文件的结构分析

在完成反汇编过程后，我们接下来对汇编器生成的目标文件（.o 文件）结构进行详细分析。目标文件采用 ELF（Executable and Linkable Format）格式，包含了丰富的信息。

ELF 头部（ELF Header） ELF 头部包含文件的基本信息：

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x0
  ...
```

关键信息包括：

文件类型：REL (Relocatable file)，表示这是一个可重定位目标文件
机器架构：AArch64（ARM64）
数据格式：小端序
入口点地址：0x0（重定位文件没有入口点）

节头表（Section Header Table） 节头表包含每个节的详细描述信息。根据 readelf 输出，main.o 文件包含 13 个节：

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       00000000000000b4  0000000000000000  AX       0     0     4
  [ 2] .rela.text        RELA             0000000000000000  000002d8
       0000000000000120  0000000000000018   I      10     1     8
  [ 3] .data             PROGBITS         0000000000000000  000000f4
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  000000f4
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata.str1.1    PROGBITS         0000000000000000  000000f4
       0000000000000007  0000000000000001 AMS       0     0     1
  ...
```

主要节内容分析

代码节（.text） .text 节包含程序的可执行指令，大小为 180 字节（0x000000b4），具有可执行（X）和已分配（A）属性。通过之前的反汇编结果，我们已经验证了其中的机器码和对应的汇编指令。

只读数据节（.rodata.str1.1） 只读数据节存储格式化字符串，大小为 7 字节，包含两个字符串：“%d”和“%d\n”，用于 scanf 和 printf 函数。

重定位表（.rela.text） 重定位表包含需要在链接时进行重定位的符号信息，共有 12 条记录：

```
Relocation section '.rela.text' at offset 0x2d8 contains 12 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000024  000600000113 R_AARCH64_ADR_PRE 0000000000000000 .rodata.str1.1 + 0
000000000028  000600000115 R_AARCH64_ADD_ABS 0000000000000000 .rodata.str1.1 + 0
000000000030  000c0000011b R_AARCH64_CALL26  0000000000000000 __isoc99_scanf + 0
...
```

主要重定位类型包括：
R_AARCH64_ADR_PRE/R_AARCH64_ADD_ABS：用于解析格式化字符串地址
R_AARCH64_CALL26：用于解析外部函数调用

符号表（.symtab） 符号表包含程序中定义和引用的符号信息，共有 14 个条目：

```
Symbol table '.symtab' contains 14 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     ...
    11: 0000000000000000   180 FUNC    GLOBAL DEFAULT    1 main
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND __isoc99_scanf
    13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

关键符号包括：
main：函数符号，大小 180 字节，位于.text 节
___isoc99_scanf 和 printf：未定义符号（外部函数引用）

4. 汇编器的工作原理和功能总结

通过对汇编过程、反汇编结果和目标文件结构的综合分析，我们可以深入理解汇编器的工作原理和核心功能：

汇编器的工作流程 汇编器的主要工作流程包括：

1. 词法分析：汇编器首先对输入的汇编代码进行词法分析，识别出汇编指令、操作数、标签、伪指令等基本语法单元。

2. 语法分析：根据汇编语言的语法规则，对词法单元进行语法分析，构建语法结构，确保代码符合 ARM64 汇编语言的语法规则。

3. 符号表构建：汇编器在处理汇编代码的过程中，会构建一个符号表，用于记录标签和变量的地址。

4. 指令编码：根据目标架构的指令集架构（ISA），将汇编指令编码为对应的机器指令。

5. 目标文件生成：将编码后的机器指令、数据、符号表等信息按照目标文件格式（如 ELF）组织起来，生成目标文件。

汇编器的核心功能 汇编器在编译过程中执行以下核心功能：

1. 指令编码：汇编器的核心功能是将汇编指令转换为对应的机器码。从反汇编结果可以看到，每条 ARM64 汇编指令都被准确地编码为 4 字节的机器码，完整实现了斐波那契数列计算的逻辑。

2. 符号管理：汇编器收集、处理和记录代码中的所有符号信息，包括符号定义处理、符号引用解析和符号表构建。从 readelf 的符号表输出可以看到，汇编器成功识别并记录了 14 个符号的详细信息。

3. 重定位处理：对于无法在汇编时确定最终地址的指令（如对外部函数的调用或对全局变量的引用），汇编器生成重定位记录。从 readelf 的重定位表可以看到，汇编器为.text 节生成了 12 条重定位记录。

4. 节的创建和管理：汇编器将代码、数据和其他信息组织到不同的节中，如.text（存储机器码指令）、.rodata（存储只读数据）、.symtab（存储符号信息）等。

5. ELF 文件格式生成：最后，汇编器按照 ELF 文件格式规范，将所有信息组织成一个完整的目标文件，包括创建 ELF 头部、节表、数据节等，并确保它们之间的引用关系正确。

（五） 链接器

链接器是编译过程的最后一个阶段，它负责将多个目标文件（.o 文件）和库文件链接在一起，生成最终的可执行文件或共享库。在本次实验中，我通过执行以下命令成功生成了可执行文件：

```
$ aarch64-linux-gnu-gcc main.o -o main
```

随后通过 qemu-aarch64 -L /usr/aarch64-linux-gnu ./main 执行该文件，终端输出结果正确，程序运行正常，能够成功生成并输出斐波那契数列：

```
$ qemu-aarch64 -L /usr/aarch64-linux-gnu ./main
3
0
1
1
2
```

1. 链接器的工作原理

链接器的主要工作流程包括：

符号解析（Symbol Resolution）：链接器会扫描所有输入的目标文件和库文件，为每个符号（如函数名、变量名）找到其定义。符号解析解决了外部引用的问题，确保每个符号引用都能找到对应的符号定义。

重定位（Relocation）：链接器会为每个符号分配一个最终的内存地址，并更新所有对该符号的引用，使其指向正确的地址。重定位解决了符号地址不确定的问题，确保程序在运行时能够正确访问内存。

内存布局（Memory Layout）：链接器会将多个目标文件的相同类型的节（如.text、.data 等）合并，并为合并后的节分配内存地址。内存布局决定了程序在内存中的组织结构。

可执行文件生成：链接器会根据链接脚本或默认的链接规则，将合并后的节、符号表、重定位表等信息按照可执行文件格式（如 ELF）组织起来，生成最终的可执行文件。

main.o 与 main 可执行文件的差异 基于编译链接的基本原理，main.o（目标文件）与 main（可执行文件）之间存在以下主要差异：

文件格式差异 main.o 是一个可重定位目标文件（ELF 类型为 REL），包含未解析的符号引用和重定位信息。而 main 是一个可执行文件（ELF 类型为 EXEC），所有符号引用已解析，重定位已完成。

代码内容差异 main.o 中的函数调用（如 ___isoc99_scanf 和 printf）使用的是相对地址或占位符，在反汇编结果中显示为 bl 0 <___isoc99_scanf>。main 可执行文件中，这些函数调用已被解析为实际的内存地址。

重定位信息 main.o 包含重定位表（.rela.text 和.rela.eh_frame），记录了需要链接器处理的地址修正信息。main 可执行文件中没有重定位表，因为所有重定位已在链接阶段完成。

段信息 main.o 只有基本的代码和数据段，如.text、.data、.bss 等。main 可执行文件包含更多段，如动态链接相关的段（.dynamic、.got、.plt 等）以及程序头表，使操作系统能够正确加载和执行程序。

符号解析 main.o 中的外部符号（如 ___isoc99_scanf 和 printf）标记为 UND（未定义）。main 可执行文件中，这些符号已被解析为实际的函数地址或动态链接表项。

2. 不同链接参数对程序的影响

为了深入了解链接参数对程序的影响，我依次执行了以下命令，生成了不同版本的可执行文件：

```
wayne@coco:~/compiler/lab1$ aarch64-linux-gnu-gcc main.o -o main # 标准动态链接
wayne@coco:~/compiler/lab1$ aarch64-linux-gnu-gcc -flto main.o -o main_lto # 链接时优化
wayne@coco:~/compiler/lab1$ aarch64-linux-gnu-gcc -Os main.o -o main_small # 优化大小
wayne@coco:~/compiler/lab1$ aarch64-linux-gnu-gcc -static main.o -o main_static # 静态链接
```

通过使用不同的链接参数，我们可以控制链接器的行为，从而影响生成的可执行文件的性能、大小和行为。以下是对这些链接参数的详细分析：

不同链接参数的特点分析

无额外参数（aarch64-linux-gnu-gcc main.o -o main） 该命令采用标准链接方式，生成动态链接的可执行文件。其主要优势在于生成的文件体积相对较小，因为它共享系统中的动态链接库。在反汇编分析中，可以观察到该版本包含了动态链接相关的代码和数据段，函数调用通常通过 PLT（过程链接表）进行，体现了动态链接的典型特征。

-flto（链接时优化） 此选项启用链接时优化机制，允许编译器在链接阶段进行跨文件边界的全局优化。该优化方式能够发现更多的优化机会，从而生成效率更高的代码。在反汇编结果中，通常可以观察到函数内联、死代码消除、更高效的寄存器分配等优化效果，这些优化措施显著提升了程序的执行性能。

-Os（优化大小） 该编译选项以优化代码大小为首要目标，特别适合资源受限的嵌入式环境。通过使用更紧凑的指令序列和精简的代码生成策略，虽然可能在一定程度上牺牲执行速度，但能够有效减小最终可执行文件的体积。在反汇编分析中，可以明显看到编译器选择了空间效率更高的指令组合。

-static（静态链接） 静态链接选项将所有依赖的库文件完整地嵌入到可执行文件中，生成完全独立的可执行程序。这种方式的最大优势是不依赖于目标系统中的动态库，具有更好的可移植性。然而，反汇编分析显示这种方式的代价是文件体积显著增大，因为包含了所有必要的库函数代码，使得最终的可执行文件自成体系。

实际文件对比分析 从实际生成的文件信息可以看到：

main_static：体积最大（1079359 字符），包含了完整的 C 标准库等依赖

main_small：体积最小，通过-Os 优化减小了代码大小

main_lto：通过链接时优化，可能在代码效率上有所提升

main：标准动态链接版本，体积适中

四、 LLVM IR 程序编写

在这一节中，我们设计的 SysY 程序在附录1中，该程序包含了全局变量声明、函数实现、条件分支、循环语句、类型转换、函数调用、库函数调用和数组操作等功能模块。下面我将逐个实现设计好的 SysY 程序的 LLVM IR 的中间代码。完整的 LLVM IR 代码参见附录2

(一) 全局变量声明

在 LLVM IR 中，全局变量使用‘@’符号作为前缀。对于 SysY 源程序中的全局变量声明：

```
; 全局变量声明
@global_int = global i32 1          ; 定义全局整型变量global_int，初始值为1
@global_float = constant float 2.0  ; 定义全局常量浮点变量global_float，值为
    2.0
```

其中，constant 关键字用于声明常量，确保该值在程序运行过程中不会被修改。

(二) 外部函数声明

为了能够调用 SysY 运行时库中的函数，我们需要在 LLVM IR 中声明这些外部函数：

```
; 外部函数声明
declare i32 @getint()                ; 声明获取整数的函数
declare i32 @getch()                 ; 声明获取字符的函数
declare void @getarray(i32*)         ; 声明获取数组的函数
declare float @getfloat()            ; 声明获取浮点数的函数

declare void @putint(i32)            ; 声明输出整数的函数
declare void @putch(i32)             ; 声明输出字符的函数
declare void @putarray(i32, i32*)    ; 声明输出数组的函数
declare void @putfloat(float)        ; 声明输出浮点数的函数
```

这些声明确保了 LLVM IR 代码能够正确链接到 SysY 运行时库中对应的函数。

(三) 简单函数：加法运算

对于 SysY 源程序中的 `add` 函数，我们在 LLVM IR 中实现如下：

```
; 简单函数：加法运算
define i32 @add(i32 %a, i32 %b) {   ; 定义add函数，接收两个i32类型参数
entry:                              ; 函数入口基本块
  %sum = add i32 %a, %b             ; 执行加法运算，结果存入%sum
  ret i32 %sum                      ; 返回计算结果
}
```

LLVM IR 采用基本块（Basic Block）的结构来组织代码，每个函数至少包含一个入口基本块（通常命名为 entry）。

（四） 主函数实现

主函数是程序的核心部分，我们按照 SysY 源程序的结构，逐步实现各个功能模块。

1. 局部变量声明与赋值

在 LLVM IR 中，局部变量通过 alloca 指令在栈上分配内存空间：

```
; 主函数
define i32 @main() {
entry:
  ; 局部变量声明与赋值
  %a = alloca i32                    ; 为整型变量a分配空间
  %b = alloca i32                    ; 为整型变量b分配空间
  %c = alloca i32                    ; 为整型变量c分配空间
  %f = alloca float                  ; 为浮点变量f分配空间
  %arr = alloca [2 x i32]            ; 为数组arr分配空间
  %arr_ptr = getelementptr inbounds [2 x i32], [2 x i32]* %arr, i32 0, i32 0
      ; 获取数组首元素指针

  store i32 1, i32* %a               ; 存储值1到变量a
  store i32 2, i32* %b               ; 存储值2到变量b
```

对于数组，我们需要使用 getelementptr 指令获取数组元素的指针，这是 LLVM IR 中访问数组元素的标准方式。

2. 数值运算和赋值语句

在 LLVM IR 中，所有操作都需要显式的加载（load）和存储（store）指令：

```
; 赋值语句
%a_val = load i32, i32* %a         ; 从变量a加载值
%b_val = load i32, i32* %b         ; 从变量b加载值
%sum1 = add i32 %a_val, %b_val     ; 执行加法运算
store i32 %sum1, i32* %c           ; 存储结果到变量c
```

3. 条件分支语句

LLVM IR 使用 icmp（整数比较）指令进行条件判断，使用 br（分支）指令实现条件跳转：

```
; 条件分支语句
%c_val = load i32, i32* %c         ; 加载变量c的值
%cmp1 = icmp sgt i32 %c_val, 2     ; 比较c是否大于2（sgt表示有符号大于）
br i1 %cmp1, label %if.then, label %if.else   ; 根据比较结果跳转到不同基本块

if.then:                           ; if分支基本块
  store i32 1, i32* %c             ; 将1存储到变量c
  br label %if.end                 ; 跳转到if结束基本块
```

```
9
10 if.else:                                  ; else分支基本块
11   store i32 2, i32* %c                    ; 将2存储到变量c
12   br label %if.end                        ; 跳转到if结束基本块
13
14 if.end:                                   ; if语句结束基本块
```

4. 循环语句

LLVM IR 中的循环通过基本块和分支指令实现。对于 while 循环，我们设计了三个基本块：循环条件判断块、循环体块和循环结束块：

```
1  ; 循环语句
2  br label %while.cond                      ; 跳转到循环条件判断基本块
3
4  while.cond:                               ; 循环条件判断基本块
5    %a_val2 = load i32, i32* %a             ; 加载变量a的值
6    %cmp2 = icmp slt i32 %a_val2, 3         ; 比较a是否小于3（slt表示有符号小于）
7    br i1 %cmp2, label %while.body, label %while.end   ; 根据比较结果决定是否继
         续循环
8
9  while.body:                               ; 循环体基本块
10   %a_val3 = load i32, i32* %a             ; 加载变量a的值
11   %add1 = add i32 %a_val3, 1              ; a = a + 1
12   store i32 %add1, i32* %a                ; 存储结果到变量a
13   br label %while.cond                    ; 跳转回循环条件判断基本块
14
15 while.end:                                ; 循环结束基本块
```

5. 类型转换

LLVM IR 是强类型语言，类型转换需要使用专门的指令。对于 int 到 float 的转换，我们使用 sitofp（有符号整数到浮点数）指令：

```
1  ; 类型转换: int 转 float
2  %c_val2 = load i32, i32* %c              ; 加载变量c的值
3  %conv1 = sitofp i32 %c_val2 to float     ; 将整数转换为浮点数
4  store float %conv1, float* %f            ; 存储结果到变量f
```

6. 函数调用

在 LLVM IR 中，函数调用通过 call 指令实现：

```
1  ; 函数调用
2  %a_val4 = load i32, i32* %a              ; 加载变量a的值
3  %b_val2 = load i32, i32* %b              ; 加载变量b的值
4  %call_add = call i32 @add(i32 %a_val4, i32 %b_val2)   ; 调用add函数
5  store i32 %call_add, i32* %c             ; 存储函数返回值到变量c
```

7. 库函数调用（I/O 操作）

调用 SysY 运行时库中的 I/O 函数与调用自定义函数类似，也是使用 'call'指令：

```
; 库函数调用（I/O）
%c_val3 = load i32, i32* %c          ; 加载变量c的值
call void @putint(i32 %c_val3)       ; 调用putint函数输出整数
call void @putch(i32 10)             ; 调用putch函数输出换行符（ASCII码10）
```

8. 数组操作

数组元素的访问和修改通过 getelementptr 指令获取元素指针，然后进行加载或存储操作：

```
; 数组操作
%arr_idx0 = getelementptr inbounds [2 x i32], [2 x i32]* %arr, i32 0, i32 0
    ; 获取数组第0个元素的指针
%arr_idx1 = getelementptr inbounds [2 x i32], [2 x i32]* %arr, i32 0, i32 1
    ; 获取数组第1个元素的指针
store i32 1, i32* %arr_idx0          ; 存储1到数组第0个元素
store i32 2, i32* %arr_idx1          ; 存储2到数组第1个元素
call void @putarray(i32 2, i32* %arr_ptr)   ; 调用putarray函数输出数组
```

9. 浮点数操作

浮点数的输入输出通过调用相应的库函数实现：

```
; 浮点数操作
%call_getfloat = call float @getfloat()   ; 调用getfloat函数获取用户输入的浮点
    数
store float %call_getfloat, float* %f     ; 存储输入值到变量f
%f_val = load float, float* %f            ; 加载变量f的值
call void @putfloat(float %f_val)         ; 调用putfloat函数输出浮点数
call void @putch(i32 10)                  ; 输出换行符

ret i32 0                                 ; 主函数返回0
}
```

（五） LLVM IR 代码的验证与运行（ARM64 平台）

对于 ARM64 平台，我们可以通过 LLVM 工具链将 LLVM IR 代码编译为 ARM 架构的可执行文件。我们创建了专门的 Makefile 来简化 ARM64 平台的编译和运行流程 (Makefile 神中神)：

```
CC = aarch64-linux-gnu-gcc
CXX = aarch64-linux-gnu-g++
LLC = llc
QEMU_ARM = qemu-aarch64-static

# 编译选项
CFLAGS = -Wall -g
```

```
CXXFLAGS = -Wall -g
LLCFLAGS = -march=aarch64

# 运行时库
SYSY_LIB = lib/libsysy_aarch.a
SYSY_INCLUDE = -Ilib

# 源文件和目标文件
LLVM_IR_SRC = factorial_example.ll
ARM_OUTPUT_ASM = factorial_arm_output.s
ARM_EXE = factorial_arm

# 默认目标
all: $(ARM_EXE)

# 从LLVM IR编译到ARM可执行文件
$(ARM_EXE): $(ARM_OUTPUT_ASM) $(SYSY_LIB)
	$(CC) $(CFLAGS) $< $(SYSY_LIB) -o $@

# 从LLVM IR生成ARM汇编
$(ARM_OUTPUT_ASM): $(LLVM_IR_SRC)
	$(LLC) $(LLCFLAGS) $< -o $@

# 运行ARM程序
run:
	$(QEMU_ARM) -L /usr/arm-linux-gnueabihf/ ./$(ARM_EXE)

# 清理生成的文件
clean:
	rm -f $(ARM_OUTPUT_ASM) $(ARM_EXE)

.PHONY: all run clean
```

通过上述 Makefile，我们可以直接使用以下命令进行编译和运行：

```
# 编译ARM64平台的可执行文件
$ make

# 使用QEMU运行ARM64程序
$ make run
```

编译过程主要分为两个步骤：

1. 使用 LLC 工具将 LLVM IR 代码转换为 ARM64 汇编代码（-march=aarch64 指定目标架构）
2. 使用 aarch64-linux-gnu-gcc 编译器将汇编代码与 libsysy_aarch.a 运行时库链接生成可执行文件

编译完成后，我们可以使用 qemu-aarch64-static 模拟器在 x86 主机上运行 ARM64 架构的可执行文件。

通过这种方式，我们成功地将 LLVM IR 代码编译链接为可执行文件，并验证了其功能正确性。运行结果如图3所示，程序首先输出了 5（这是 `putint(c)` 的结果，其中 `c=add(a,b)=3+2=5`），然后输出换行符，接着输出 `2: 1 2`（这是 `putarray(2, arr)` 的结果，显示数组中有 2 个元素，分别是 1 和 2），程序等待用户输入浮点数（这里输入了 4），随后程序输出了这个浮点数的十六进制表示形式 `0x1p+0`（表示 1）。最后显示了程序运行时间信息。c

```
(base) ➜ task2 qemu-aarch64-static -L /usr/aarch64-linux-gnu/ factorial_arm
5
2: 1 2
4
0x1p+2
TOTAL: 0H-0M-0S-0us
```

图 3: 程序运行结果验证

五、 ARM64 汇编程序编写

完整的 ARM64 汇编代码参见附录3

(一) 架构声明和全局符号定义

汇编代码以.arch armv8-a 开头，这指定了目标架构为 ARMv8-A，即 64 位 ARM 架构。随后使用.text 指令进入代码段，并通过.align 2 实现 4 字节对齐，确保指令在内存中的正确布局。在全局符号定义部分，通过.global add 将 add 函数导出为全局符号，使得其他模块可以调用该函数，同时使用.type add, %function 明确声明 add 为函数类型，这有助于链接器和调试器正确识别符号属性。

```
1  .arch armv8-a
2  .text
3  .align 2
4
5  /* 函数符号导出 */
6  .global add
7  .type add, %function
```

(二) 加法函数实现

add 函数的实现直接使用 add w0, w0, w1 指令将两个参数相加，结果存储在 w0 寄存器中作为返回值。由于这是一个简单的叶子函数，不需要复杂的栈帧管理，因此省略了传统的函数序言和尾声，直接使用 ret 指令返回。

```
1  add:
2      /* 函数序言（不保存帧指针，函数很简单） */
3      add     w0, w0, w1      /* w0 = a + b */
4      ret
5
6      .size add, .-add
```

（三） 外部库函数声明

在调用外部库函数之前，需要使用.extern 指令声明这些外部符号。本代码中声明了 putint、putch、putarray、getfloat 和 putfloat 五个库函数，这些函数提供了基本的输入输出功能。

```
    .extern putint
    .extern putch
    .extern putarray
    .extern getfloat
    .extern putfloat
```

（四） 全局变量定义

全局变量的定义分为两个部分：可读写的整型变量 global_int 定义在.data 段，初始值为 1；只读的浮点常量 global_float 定义在.rodata 段，值为 2.0。这种分段存储体现了不同类型数据的存储特性，.data 段用于可修改的全局变量，.rodata 段用于常量数据.

```
    .data
    .align 2
    .global global_int
global_int:
    .word 1             /* 32-bit integer initialized to 1 */

    .section .rodata
    .align 2
    .global global_float
global_float:
    .float 2.0          /* 32-bit float constant */
```

（五） 主函数实现

1. 函数序言和栈帧分配

main 函数的开始部分展示了标准的函数序言。stp x29, x30, [sp, -48]! 指令同时完成了三个功能：将帧指针 x29 和链接寄存器 x30 压栈保存，为局部变量分配 48 字节的栈空间，并更新栈指针 sp。随后的 mov x29, sp 指令建立新的帧指针，为后续的局部变量访问提供基准地址。

```
main:
    /* 函数序言 */
    stp     x29, x30, [sp, -48]!    /* 保存 fp/lr, 并分配 48 字节栈帧 */
    mov     x29, sp
```

2. 变量初始化和算术运算

使用加载 (ldr)-运算-存储 (str) 的模式实现变量操作。

```
    /* 初始化局部变量 a = 1; b = 2; */
    mov     w0, #1
    str     w0, [x29, #16]      /* store a */
    mov     w0, #2
```

```
    str     w0, [x29, #20]    /* store b */

    /* c = a + b; */
    ldr     w1, [x29, #16]    /* w1 = a */
    ldr     w2, [x29, #20]    /* w2 = b */
    add     w3, w1, w2        /* w3 = a + b */
    str     w3, [x29, #24]    /* c = w3 */
```

3. 条件分支实现

条件分支的实现依赖于比较和条件跳转指令的配合。cmp w3, w4 指令比较变量 c 和立即数 2 的大小，设置相应的条件标志位。随后 b.le .L_else_branch 指令根据比较结果进行跳转：如果 c 小于等于 2，则跳转到 else 分支，否则继续执行 then 分支。

```
    /* if (c > 2) c = 1; else c = 2; */
    mov     w4, #2
    cmp     w3, w4            /* compare c and 2 */
    b.le    .L_else_branch    /* if c <= 2 -> else */
```

4. 循环结构实现

使用标签和条件跳转指令实现 while 循环逻辑。

```
.L_while_begin:
    ldr     w6, [x29, #16]    /* w6 = a */
    mov     w7, #3
    cmp     w6, w7
    b.ge    .L_while_end      /* break if a >= 3 */
```

5. 类型转换

整数到浮点的类型转换使用 scvtf s0, w8 指令实现，该指令将有符号整数寄存器 w8 中的值转换为单精度浮点数，结果存储在浮点寄存器 s0 中。这是 ARM64 架构提供的专用类型转换指令。

```
    /* f = (float)c; */
    ldr     w8, [x29, #24]    /* w8 = c */
    scvtf   s0, w8            /* s0 = (float)w8 */
```

6. 函数调用

按照 ARM64 调用约定，参数通过 w0-w7 寄存器传递，bl 指令实现函数调用。

```
    /* c = add(a, b); */
    ldr     w0, [x29, #16]    /* arg0 = a */
    ldr     w1, [x29, #20]    /* arg1 = b */
    bl      add               /* result in w0 */
```

7. 库函数调用和数组操作

库函数的调用方式与普通函数类似，但参数准备需要符合库函数的接口要求。数组操作通过传递基地址指针的方式实现，add x1, x29, #32 计算数组 arr 的起始地址，然后将地址指针传递给 putarray 函数。

```
    /* putint(c); putch(10); */
    ldr     w0, [x29, #24]    /* arg for putint */
    bl      putint

    /* putarray(2, arr); */
    mov     w0, #2            /* n = 2 */
    add     x1, x29, #32      /* &arr */
    bl      putarray
```

8. 函数收尾

恢复栈帧并返回，返回值 0 存放在 w0 寄存器中。

```
    /* return 0; */
    mov     w0, #0
    ldp     x29, x30, [sp], 48
    ret
```

(六) 汇编代码验证和运行

编译环境的配置通过 Makefile 实现，使用 aarch64-linux-gnu-gcc 作为交叉编译器，针对 ARM64 架构进行编译。编译过程分为两个阶段：首先将汇编代码编译成目标文件，然后链接运行时库生成可执行文件。运行阶段使用 qemu-aarch64 模拟器在 x86 环境中执行 ARM64 程序，通过指定动态链接库路径确保程序正常运行。

```
# 编译器设置
CC = aarch64-linux-gnu-gcc
CFLAGS = -O2
LDFLAGS = -L. -lsysy_aarch -lm

# 源文件和目标文件
SRCS = example.S
OBJS = $(SRCS:.S=.o)
EXEC = example

# 默认目标：编译、链接并运行
all: $(EXEC) run

# 汇编步骤：将 example.S 编译成 example.o
%.o: %.S
	$(CC) $(CFLAGS) -c $< -o $@

# 链接步骤：将目标文件链接成可执行文件 example
```

```
$(EXEC): $(OBJS)
	$(CC) $(OBJS) $(LDFLAGS) -o $(EXEC)

# 运行目标: 用 QEMU 模拟 ARM64 环境运行程序
run: $(EXEC)
	qemu-aarch64 -L /usr/aarch64-linux-gnu ./$(EXEC)

# 清理: 删除目标文件和可执行文件
clean:
	rm -f $(OBJS) $(EXEC)

# 伪目标: 确保没有与文件同名
.PHONY: all clean run
```

程序运行结果显示了完整的执行流程：首先输出 add 函数的计算结果 5，然后输出数组内容”2: 1 2”，接着处理浮点数输入输出，最后显示程序执行时间统计。这些输出结果与 SysY 源代码的预期行为完全一致，证明了汇编代码转换的正确性。特别是浮点数输出”0x1p+2” 是 4.0 的十六进制浮点表示，符合 IEEE754 标准。

```
$ make
aarch64-linux-gnu-gcc -O2 -c example.S -o example.o
aarch64-linux-gnu-gcc example.o -L. -lsysy_aarch -lm -o example
qemu-aarch64 -L /usr/aarch64-linux-gnu ./example
5
2: 1 2
4
0x1p+2
TOTAL: 0H-0M-0S-0us
```

参考文献

1. LLVM 官方文档，https://llvm.org/docs/
2. GCC 官方文档，https://gcc.gnu.org/onlinedocs/
3. ARM 架构参考手册，https://developer.arm.com/documentation/
4. SysY2022 语言定义及运行时的库，课程文件提供。
5. 预备工作 1——了解编译器，LLVM IR 编程及汇编编程，杨俣哲、李煦阳、孙一丁、李世阳、杨科迪、周辰霏、尧泽斌、时浩铭、贺祎昕、张书睿、华志远、李帅东、唐显达、鲁恒泽，2020 年 9 月—2025 年 9 月。

附录说明

附录所示代码均已上传至 GitHub,链接:https://github.com/insung186/Compiler-Homework/tree/main/Lab1

A SysY 示例程序代码

```
// 一、全局变量声明
int global_int = 1;
const float global_float = 2.0;

// 二、简单函数：加法运算
int add(int a, int b) {
    return a + b;  // 三、数值运算：加法
}

// 主函数
int main() {
    // 局部变量声明与赋值
    int a = 1;
    int b = 2;
    int c;
    float f;

    // 赋值语句
    c = a + b;  // 数值运算：加法

    // 四、条件分支语句
    if (c > 2) {  // 五、关系运算：大于
        c = 1;
    } else {
        c = 2;
    }

    // 六、循环语句
    while (a < 3) {  // 关系运算：小于
        a = a + 1;   // 数值运算：加法
    }

    // 七、类型转换：int 转 float
    f = c;

    // 八、函数调用
    c = add(a, b);

    // 九、库函数调用（I/O）
    putint(c);        // 输出整数
```

```
    putch(10);          // 输出换行符（ASCII 10）

    // 十、数组操作
    int arr[2];
    arr[0] = 1;
    arr[1] = 2;
    putarray(2, arr);  // 输出数组

    // 十一、浮点数操作
    f = getfloat();     // 输入浮点数
    putfloat(f);        // 输出浮点数
    putch(10);

    return 0;
}
```

Listing 1: example.sy 文件内容

B LLVM IR 示例程序代码

```
@global_int = global i32 1
@global_float = constant float 2.0

; 外部函数声明
declare i32 @getint()
declare i32 @getch()
declare void @getarray(i32*)
declare float @getfloat()
declare void @getfarray(float*)

declare void @putint(i32)
declare void @putch(i32)
declare void @putarray(i32, i32*)
declare void @putfloat(float)

declare void @putfarray(i32, float*)

; 简单函数：加法运算
define i32 @add(i32 %a, i32 %b) {
entry:
  %sum = add i32 %a, %b
  ret i32 %sum
}

; 主函数
define i32 @main() {
entry:
```

```
28    ; 局部变量声明与赋值
29    %a = alloca i32
30    %b = alloca i32
31    %c = alloca i32
32    %f = alloca float
33    %arr = alloca [2 x i32]
34    %arr_ptr = getelementptr inbounds [2 x i32], [2 x i32]* %arr, i32 0, i32 0
35
36    store i32 1, i32* %a
37    store i32 2, i32* %b
38
39    ; 赋值语句
40    %a_val = load i32, i32* %a
41    %b_val = load i32, i32* %b
42    %sum1 = add i32 %a_val, %b_val
43    store i32 %sum1, i32* %c
44
45    ; 条件分支语句
46    %c_val = load i32, i32* %c
47    %cmp1 = icmp sgt i32 %c_val, 2
48    br i1 %cmp1, label %if.then, label %if.else
49
50  if.then:
51    store i32 1, i32* %c
52    br label %if.end
53
54  if.else:
55    store i32 2, i32* %c
56    br label %if.end
57
58  if.end:
59
60    ; 循环语句
61    br label %while.cond
62
63  while.cond:
64    %a_val2 = load i32, i32* %a
65    %cmp2 = icmp slt i32 %a_val2, 3
66    br i1 %cmp2, label %while.body, label %while.end
67
68  while.body:
69    %a_val3 = load i32, i32* %a
70    %add1 = add i32 %a_val3, 1
71    store i32 %add1, i32* %a
72    br label %while.cond
73
74  while.end:
75
```

```
  ; 类型转换: int 转 float
  %c_val2 = load i32, i32* %c
  %conv1 = sitofp i32 %c_val2 to float
  store float %conv1, float* %f

  ; 函数调用
  %a_val4 = load i32, i32* %a
  %b_val2 = load i32, i32* %b
  %call_add = call i32 @add(i32 %a_val4, i32 %b_val2)
  store i32 %call_add, i32* %c

  ; 库函数调用 (I/O)
  %c_val3 = load i32, i32* %c
  call void @putint(i32 %c_val3)
  call void @putch(i32 10)  ; 换行符

  ; 数组操作
  %arr_idx0 = getelementptr inbounds [2 x i32], [2 x i32]* %arr, i32 0, i32 0
  %arr_idx1 = getelementptr inbounds [2 x i32], [2 x i32]* %arr, i32 0, i32 1
  store i32 1, i32* %arr_idx0
  store i32 2, i32* %arr_idx1
  call void @putarray(i32 2, i32* %arr_ptr)

  ; 浮点数操作
  %call_getfloat = call float @getfloat()
  store float %call_getfloat, float* %f
  %f_val = load float, float* %f
  call void @putfloat(float %f_val)
  call void @putch(i32 10)  ; 换行符

  ret i32 0
}
```

Listing 2: example.ll 文件内容

C ARM64 示例程序代码

```
    .arch armv8-a
    .text
    .align 2

    /* 函数符号导出 */
    .global add
    .type add, %function

/* ----------------------
   int add(int a, int b) { return a + b; }
```

```
    参数: w0, w1
    返回: w0
    ------------------------ */
add:
    /* 函数序言（不保存帧指针，函数很简单） */
    add     w0, w0, w1      /* w0 = a + b */
    ret

    .size add, .-add

/* ------------------------
    外部运行时库函数（由链接器解析）
    原型（来自 sylib.h / 运行时库文档）:
      int     getint(void);
      int     getch(void);
      float   getfloat(void);
      int     getarray(int a[]);
      int     getfarray(float a[]);
      void    putint(int);
      void    putch(int);
      void    putarray(int, int[]);
      void    putfloat(float);
      void    putf(char[], ...);
    (只在汇编中直接调用需要的几个)
    ------------------------ */
    .extern putint
    .extern putch
    .extern putarray
    .extern getfloat
    .extern putfloat

/* ------------------------
    全局变量（与 example.txt 语义对应）
    int global_int = 1;
    const float global_float = 2.0;
    ------------------------ */
    .data
    .align 2
    .global global_int
global_int:
    .word 1             /* 32-bit integer initialized to 1 */

    .section .rodata
    .align 2
    .global global_float
global_float:
    .float 2.0          /* 32-bit float constant */

```

```
    .text
    .align 2
    .global main
    .type main, %function

/* ----------------------
   int main()
   我们在栈上分配局部变量:
     int a, b, c;  (3 * 4 bytes)
     float f;      (4 bytes)
     int arr[2];   (2 * 4 bytes)
   共计 24 bytes, 向上对齐为 32 bytes, 再加上保存 x29,x30 共 16 bytes -> 48
       bytes
   ---------------------- */
main:
    /* 函数序言 */
    stp     x29, x30, [sp, -48]!   /* 保存 fp/lr, 并分配 48 字节栈帧 */
    mov     x29, sp

    /* 方便注释: 栈帧布局 (低地址 -> 高地址)
       [sp]        : saved x29
       [sp+8]      : saved x30
       [sp+16..31] : padding / locals
       We'll use offsets from x29 (which equals sp) to access locals:
       offset 16: a  (4 bytes)
       offset 20: b  (4 bytes)
       offset 24: c  (4 bytes)
       offset 28: f  (float, 4 bytes)
       offset 32: arr[0] (4 bytes)
       offset 36: arr[1] (4 bytes)
    */

    /* 初始化局部变量 a = 1; b = 2; (来自 example.txt) */
    mov     w0, #1
    str     w0, [x29, #16]    /* store a */
    mov     w0, #2
    str     w0, [x29, #20]    /* store b */

    /* 注: c 未初始化 yet */

    /* c = a + b; */
    ldr     w1, [x29, #16]    /* w1 = a */
    ldr     w2, [x29, #20]    /* w2 = b */
    add     w3, w1, w2        /* w3 = a + b */
    str     w3, [x29, #24]    /* c = w3 */

    /* if (c > 2) c = 1; else c = 2; */
    mov     w4, #2
```

```
106     cmp     w3, w4              /* compare c and 2 */
107     b.le    .L_else_branch      /* if c <= 2 -> else */
108
109     /* then: c = 1; */
110     mov     w5, #1
111     str     w5, [x29, #24]
112     b       .L_after_if
113
114 .L_else_branch:
115     /* else: c = 2; */
116     mov     w5, #2
117     str     w5, [x29, #24]
118
119 .L_after_if:
120     /* while (a < 3) { a = a + 1; } */
121     /* load a, compare with 3, loop */
122 .L_while_begin:
123     ldr     w6, [x29, #16]      /* w6 = a */
124     mov     w7, #3
125     cmp     w6, w7
126     b.ge    .L_while_end        /* break if a >= 3 */
127
128     /* a = a + 1 */
129     add     w6, w6, #1
130     str     w6, [x29, #16]
131     b       .L_while_begin
132
133 .L_while_end:
134     /* f = (float)c;  将 int -> float, 使用 scvtf (signed convert to float)
        */
135     ldr     w8, [x29, #24]      /* w8 = c */
136     scvtf   s0, w8              /* s0 = (float)w8  (float return/passing
        register v0.s) */
137     /* 将 s0 存入局部 f 位置 (在内存存储 32-bit float) */
138     str     s0, [x29, #28]
139
140     /* c = add(a, b); 调用局部函数 add */
141     ldr     w0, [x29, #16]      /* arg0 = a */
142     ldr     w1, [x29, #20]      /* arg1 = b */
143     bl      add                 /* result in w0 */
144     str     w0, [x29, #24]      /* store c = result */
145
146     /* putint(c); putch(10); */
147     ldr     w0, [x29, #24]      /* arg for putint */
148     bl      putint              /* void putint(int) */
149
150     mov     w0, #10
151     bl      putch               /* putch(10) -> newline */
```

```
152
153     /* int arr[2]; arr[0]=1; arr[1]=2; putarray(2, arr); */
154     mov     w0, #1
155     str     w0, [x29, #32]    /* arr[0] = 1 */
156     mov     w0, #2
157     str     w0, [x29, #36]    /* arr[1] = 2 */
158
159     /* prepare arguments for putarray(int n, int arr[]) */
160     mov     w0, #2            /* n = 2 */
161     add     x1, x29, #32      /* &arr (address of arr[0]) -> x1 (pointer) */
162     bl      putarray
163
164     /* f = getfloat(); putfloat(f); putch(10); */
165     bl      getfloat          /* returns float in s0 */
166     /* store returned float into local f (optional) */
167     str     s0, [x29, #28]
168
169     /* call putfloat with s0 already containing the float arg (AArch64 float
            arg v0.s) */
170     /* In assembly, ensure the float is in v0/s0 before call; getfloat
            returned it in s0/v0 already */
171     bl      putfloat
172
173     mov     w0, #10
174     bl      putch
175
176     /* return 0; (main 返回值) */
177     mov     w0, #0
178
179     /* 函数尾: 恢复栈帧并返回 */
180     ldp     x29, x30, [sp], 48
181     ret
182
183     .size main, .-main
```

Listing 3: example.S 文件内容