



南開大學
Nankai University

计算机学院
编译系统原理实验报告

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 9 月 29 日

目录

1 实验目的	2
2 实验环境	2
3 分工说明	2
4 任务一：掌握程序编译过程 from 林盛森	2
4.1 完整的编译过程都有什么？	2
4.2 预处理器做了什么？	3
4.2.1 功能探究	3
4.2.2 实验验证	4
4.3 编译器做了什么？	6
4.3.1 功能探究	6
4.3.2 实验验证	7
4.4 汇编器做了什么？	23
4.4.1 功能探究	23
4.4.2 实验验证	23
4.5 链接器做了什么？	27
4.5.1 功能探究	27
4.5.2 实验验证	27
5 任务二：掌握 LLVM IR 和汇编编程	28
5.1 riscv 汇编编程 from 林盛森	28
5.1.1 代码编写及实验验证	28
5.1.2 RISC-V 64 位汇编语言特性分析	35
5.2 LLVM IR 编程 from 牛帅	36
5.2.1 LLVM IR 概述	36
5.2.2 LLVM IR 编写	38
5.2.3 LLVM IR 改进	42
5.2.4 运行结果	43
6 心得体会	43
7 代码链接	44

1 实验目的

1. 熟悉编译器功能；
2. 掌握 LLVM IR 和汇编编程。

2 实验环境

配置项	规格说明
宿主操作系统	Microsoft Windows 11
虚拟化环境	WSL 2 (Windows Subsystem for Linux 2)
Linux 发行版	Ubuntu 22.04.5 LTS

表 1: 实验环境配置

3 分工说明

在任务一中，两人分别负责各自独立的任务；任务二中，林盛森负责编写 riscv 代码以及相关分析，牛帅负责编写 llvm 代码以及相关分析。

4 任务一：掌握程序编译过程 from 林盛森

4.1 完整的编译过程都有什么？

通常来说，我们要想把源文件（即程序员书写在编辑器里的文本文件）转化成一个可执行的目标文件，需要经由编译器来接手这一过程，而编译器将源程序转化成目标程序的这一过程，主要分为**预处理（Preprocessing）**、**编译（Compilation）**、**汇编（Assembly）**、**链接（Linking）**这四个阶段，而要想执行这四个阶段，分别需要通过**预处理器**、**编译器**、**汇编器**和**链接器**来进行协同工作，这也就组成了编译系统。

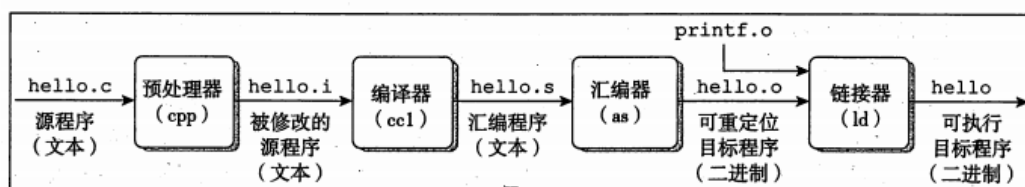


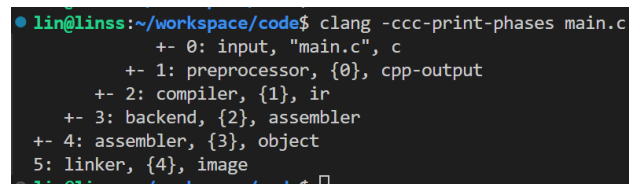
图 4.1: 编译系统

下面，我们以下面这个简单的 `main.c` 源文件为例依次来介绍一下这四个阶段。这是一个简单的斐波那契数列求解器，我们不再进行赘述。

```
1 #include <stdio.h>
2 int main()
```

```
3 {
4     int a, b, i, t, n;
5     a = 0;
6     b = 1;
7     i = 1;
8     scanf("%d", &n);
9     printf("%d\n", a);
10    printf("%d\n", b);
11    while (i < n)
12    {
13        t = b;
14        b = a + b;
15        printf("%d\n", b);
16        a = t;
17        i = i + 1;
18    }
19    return 0;
20 }
```

通过 `clang -ccc-print-phases main.c` 指令，我们可以得到编译的整个流程，如下所示：



```
lin@linss:~/workspace/code$ clang -ccc-print-phases main.c
+- 0: input, "main.c", c
+- 1: preprocessor, {0}, cpp-output
+- 2: compiler, {1}, ir
+- 3: backend, {2}, assembler
+- 4: assembler, {3}, object
5: linker, {4}, image
lin@linss:~/workspace/code$
```

图 4.2: 编译过程

4.2 预处理器做了什么？

4.2.1 功能探究

在我的认知中，预处理器的作用就是把一段代码进行标准化处理，从而方便于后续编译器的操作。具体来说，主要包括**预处理指令的处理**、**注释的删除**、**对源文件进行标识**等操作，并生成另一个 C 程序，并以 `.i` 作为文件扩展名。

通常，在我们的程序中，会有 `#`——这样类似的标识，以上面的 `main.c` 程序为例，我们在第一行就写下了 `#include<stdio.h>` 这句话，这句话中的 **`#include`**，就是预处理指令的显式定义，而预处理器在预处理阶段，就会处理这条指令，把被包含的头文件直接用来替换这条指令，也就是 `stdio` 这个头文件中的内容被加载到了 `main.c` 的头部。而在我们的程序中，还会遇到 **`#if`**、**`#ifdef`**、**`#elif`**、**`#else`**、**`#endif`** 等条件预编译指令，预处理器也会处理这些指令，决定之后是否编译代码的特定部分。除此之外，还会对 **`#define`** 定义的宏进行宏展开。

删除注释这一点比较好理解，主要就是减轻后续编译器的分析负担，简化后续的词法分析。

而对源文件进行标识这一点，主要是预处理器在预处理阶段会对源文件添加行号以及一些特殊的标识，以便于当编译时若程序报错，可以为程序员提供错误信息以及错误的位置等提示。

4.2.2 实验验证

为了验证预处理阶段对原始代码进行的操作，我重新修改了 main.c 文件，如下：

```
1 #include <stdio.h>
2
3 //这是一个计算斐波那契数列的程序 最大输入n为10000 超过10000则取10000
4 #define MAX 10000
5 #define FLAG
6 #ifdef FLAG
7     int myflag=1;
8 #else
9     int myflag=0;
10 #endif
11
12 int main()
13 {
14     int a, b, i, t, n;
15
16     a = 0;
17     b = 1;
18     i = 1;
19     scanf("%d", &n);
20     if(n>MAX) n=MAX;
21     printf("%d\n", a);
22     printf("%d\n", b);
23     while (i < n)
24     {
25         t = b;
26         b = a + b;
27         printf("%d\n", b);
28         a = t;
29         i = i + 1;
30     }
31
32     return 0;
33 }
```

我们运行如下的命令进行单步处理，即只进行预处理阶段：

```
1 gcc -E main.c -o main.i
```

运行之后，出现如下结果，可以看到成功生成了预处理后的 main.i 文件。



图 4.3: main.i 生成

我们点开 main.i 文件，代码行数从原来的 33 行增加到 772 行，原来的 `#include<stdio.h>` 被成功替换成了对应的 `stdio.h` 头文件。除此之外，我们拉到程序的最下边，可以看到，多了一些空行，也就是预处理器对原来的 main.c 中的预处理指令进行了处理，并且还删除了我们添加进去的注释等。可以看到，我们第 3 行添加的注释被预处理器删除掉了。除此之外，我们通过 `#define MAX 10000` 进行的宏定义，把 MAX 定义为 10000，可以看到在后续被使用时所有的 MAX 被成功替换成了 10000，对于接下来的 `#ifdef FLAG` 及之后的这 5 行，由于我们定义了 FLAG，所以走向 if 的对应分支，即保留了 “`int myflag=1;`”，而其他的内容被删除掉了，这也可以在 main.c 中得到验证。

```
743
744 # 7 "main.c"
745 int myflag=1;
746
747
748
749
750 int main()
751 {
752     int a, b, i, t, n;
753
754     a = 0;
755     b = 1;
756     i = 1;
757     scanf("%d", &n);
758     if(n>10000) n=10000;
759     printf("%d\n", a);
760     printf("%d\n", b);
761     while (i < n)
762     {
763         t = b;
764         b = a + b;
765         printf("%d\n", b);
766         a = t;
767         i = i + 1;
768     }
769
770     return 0;
771 }
772
```

图 4.4: main.i 具体内容

4.3 编译器做了什么？

4.3.1 功能探究

编译器在这一过程中的作用主要是对预处理器生成完的.i 程序，主要采取分析-综合模型，进行词法分析、语法分析、语义分析、中间代码生成、中间代码优化和目标代码生成这六个步骤，从而翻译成以.s 为文件扩展名的汇编语言程序。

- **词法分析**：词法分析器通过扫描将源代码的字符序列分割成一系列的记号 (Token)。
- **语法分析**：语法分析器检查代码中的语法错误并确保源代码正确遵循源语言规则，若准确无误，将记号流组装成语法树 (Syntax Tree)。
- **语义分析**：确保 AST 中的构造在语义上是有意义的，包括类型检查、变量使用前声明的检查、控制流检查等。
- **中间代码生成**：将 AST 转换为中间代码（如三地址代码），这种代码形式更接近机器代码，但与目标机器和运行环境无关，便于进行优化。
- **中间代码优化**：改进中间代码，包括删除无用代码、循环优化、常量折叠、强度削弱等多种技术，提高运行效率而不改变程序的功能。
- **目标代码生成**：将优化后的中间代码转换为目标机器代码，通常是汇编代码（查阅资料得知也有少数直接生成机器码），依赖于目标平台。

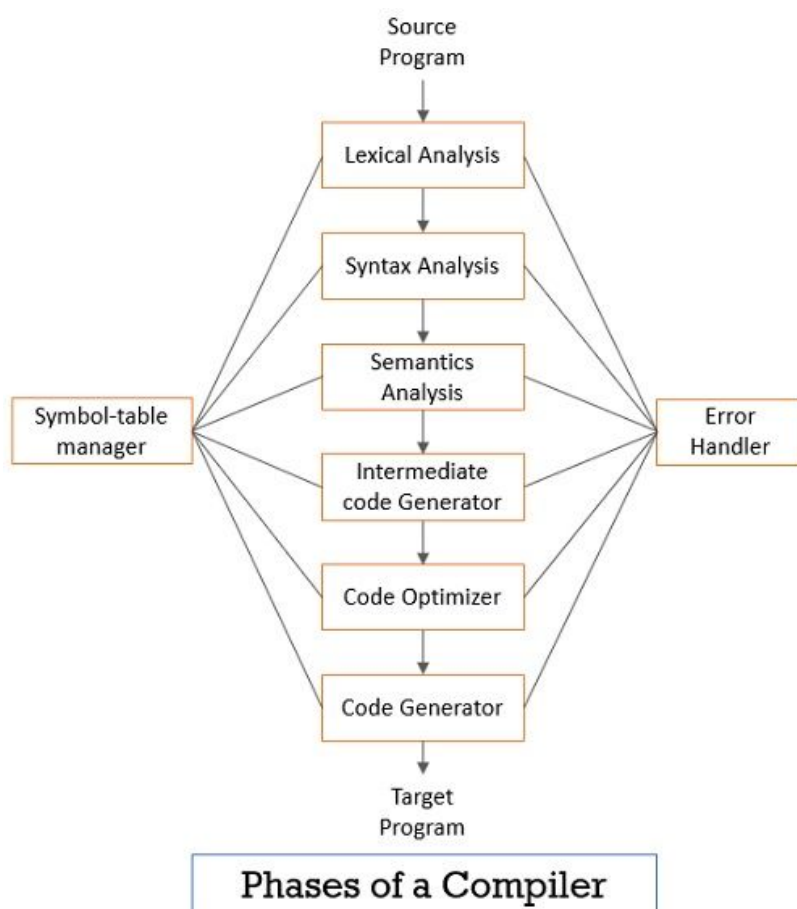


图 4.5: 编译阶段

4.3.2 实验验证

我们可以通过下面的命令，直接生成对应的汇编语言程序。这个命令生成的是特定于当前硬件平台的汇编语言，通过查看生成内容与汇编语言的特点，可以知道生成的汇编语言是特定于 x86-64 架构和 Linux 操作系统的。

```
1 gcc -S main.i -o main.s
```

为了进一步分析编译器在编译阶段对源程序的详细操作，我们采取分布进行的方式，逐层次的对编译器的六个阶段进行分析。

词法分析 运行如下命令，可以得到编译器对源程序进行词法分析后的结果，我们将其保存在 tokens.txt 文件中。

```
1 clang -E -Xclang -dump-tokens main.c 2> tokens.txt
#加2的原因是：-dump-tokens属于标准错误，不是标准输出
```

运行结果片段如下所示，可以看到，每一个 token 信息的尾部，都有该 token 在源文件中所处行号的信息，这进一步验证了预处理阶段预处理器对源文件添加行号以及一些特殊的标识这一操作。该分词结果不仅包含了 main.c 中的内容，也包含了被替换的 stdio.h 中的内容，为了简化，我们在这里只对 main.c 中对应的词法分析结果进行分析：

```
int 'int' [StartOfLine] Loc=<main.c:7:1>
identifier 'myflag' [LeadingSpace] Loc=<main.c:7:5>
equal '=' Loc=<main.c:7:11>
numeric_constant '1' Loc=<main.c:7:12>
semi ';' Loc=<main.c:7:13>
```

图 4.6: 词法分析 1

这里进行了全局变量 flag 的声明与初始化。主要词法序列为：关键字 (int) → 标识符 (myflag) → 运算符 (=) → 常量 (1) → 分隔符 (;)。第一个 token，int 关键字表明整数数据类型，[StartOfLine] 表明该关键字位于行首，Loc 表明了该 token 所处的文件位置以及行列信息，位于 main.c 文件中的第 7 行第 1 列；第二个 token，identifier 指明 flag 是标识符，描述了变量名字，[LeadingSpace] 说明该 token 前面有一个空格；第三个 token，equal 指明这是等号运算符；第四个 token，numeric_constant 表明这是数值常量；第五个 token，semi 就是分号的意思，表明这个分隔符是分号，是语句结束符。

```
int 'int' [StartOfLine] Loc=<main.c:12:1>
identifier 'main' [LeadingSpace] Loc=<main.c:12:5>
l_paren '(' Loc=<main.c:12:9>
r_paren ')' Loc=<main.c:12:10>
l_brace '{' [StartOfLine] Loc=<main.c:13:1>
```

图 4.7: 词法分析 2

这里是主函数定义头，主要词法序列为关键字 (int) → 标识符 (main) → 分隔符 (()) → 分隔符 ({}). int 关键字表明整数数据类型，说明该函数的返回值是 int 类型；identifier 指明 main 是标识符，指明了函数名，接下来的 l_paren 和 r_paren 说明这两个 token 是分隔符，是函数参数列表的括号，最后 l_brace 说明该分隔符是代码块开始符，也就是 main 函数开始的标志。


```

int 'int' [StartOfLine] [LeadingSpace] Loc=<main.c:14:5>
identifier 'a' [LeadingSpace] Loc=<main.c:14:9>
comma ',' [LeadingSpace] Loc=<main.c:14:10>
identifier 'b' [LeadingSpace] Loc=<main.c:14:12>
comma ',' [LeadingSpace] Loc=<main.c:14:13>
identifier 'i' [LeadingSpace] Loc=<main.c:14:15>
comma ',' [LeadingSpace] Loc=<main.c:14:16>
identifier 't' [LeadingSpace] Loc=<main.c:14:18>
comma ',' [LeadingSpace] Loc=<main.c:14:19>
identifier 'n' [LeadingSpace] Loc=<main.c:14:21>
semi ';' [LeadingSpace] Loc=<main.c:14:22>

```

图 4.8: 词法分析 3

这里比较简单，定义了 5 个变量，词法序列为关键字 (int) → 标识符 (a) → 分隔符 (,) → 标识符 (b) → 分隔符 (,) → 标识符 (i) → 分隔符 (,) → 标识符 (t) → 分隔符 (,) → 标识符 (n) → 分隔符 (;)。

comma 就是逗号的意思，指明这个分隔符就是“,”，其他的分析与上面大致相同。

```

identifier 'a' [StartOfLine] [LeadingSpace] Loc=<main.c:16:5>
equal '=' [LeadingSpace] Loc=<main.c:16:7>
numeric_constant '0' [LeadingSpace] Loc=<main.c:16:9>
semi ';' [LeadingSpace] Loc=<main.c:16:10>
identifier 'b' [StartOfLine] [LeadingSpace] Loc=<main.c:17:5>
equal '=' [LeadingSpace] Loc=<main.c:17:7>
numeric_constant '1' [LeadingSpace] Loc=<main.c:17:9>
semi ';' [LeadingSpace] Loc=<main.c:17:10>
identifier 'i' [StartOfLine] [LeadingSpace] Loc=<main.c:18:5>
equal '=' [LeadingSpace] Loc=<main.c:18:7>
numeric_constant '1' [LeadingSpace] Loc=<main.c:18:9>
semi ';' [LeadingSpace] Loc=<main.c:18:10>

```

图 4.9: 词法分析 4

这里对 a, b, i 三个变量进行了赋值处理，词法序列模式为：标识符 → 运算符 (=) → 常量 → 分隔符 (;)。没有什么特别值得分析的，不再进行说明。

```

identifier 'scanf' [StartOfLine] [LeadingSpace] Loc=<main.c:19:5>
l_paren '(' [LeadingSpace] Loc=<main.c:19:10>
string_literal '%"d"' [LeadingSpace] Loc=<main.c:19:11>
comma ',' [LeadingSpace] Loc=<main.c:19:15>
amp '&' [LeadingSpace] Loc=<main.c:19:17>
identifier 'n' [LeadingSpace] Loc=<main.c:19:18>
r_paren ')' [LeadingSpace] Loc=<main.c:19:19>
semi ';' [LeadingSpace] Loc=<main.c:19:20>

```

图 4.10: 词法分析 5

接下来是输入处理段，词法序列为标识符 (scanf) → 分隔符 (()) → 常量 ("%d") → 分隔符 (,) → 运算符 (&) → 标识符 (n) → 分隔符 (()) → 分隔符 (;)。

scanf 是标识符，标准库函数名；string_literal 指明"%d" 是字符串常量，用于格式化字符串；amp 就是 & 运算符的英文缩写，表明这是取地址运算符，其他的没有什么特别的就不再分析。

```

if 'if' [StartOfLine] [LeadingSpace] Loc=<main.c:20:5>
l_paren '(' [LeadingSpace] Loc=<main.c:20:7>
identifier 'n' [LeadingSpace] Loc=<main.c:20:8>
greater '>' [LeadingSpace] Loc=<main.c:20:9>
numeric_constant '10000' [LeadingSpace] Loc=<main.c:20:10 <Spelling=main.c:4:13>>
r_paren ')' [LeadingSpace] Loc=<main.c:20:13>
identifier 'n' [LeadingSpace] Loc=<main.c:20:15>
equal '=' [LeadingSpace] Loc=<main.c:20:16>
numeric_constant '10000' [LeadingSpace] Loc=<main.c:20:17 <Spelling=main.c:4:13>>
semi ';' [LeadingSpace] Loc=<main.c:20:20>

```

图 4.11: 词法分析 6

然后是一个 if 的条件判断，词法序列为关键字 (if) → 分隔符 (()) → 标识符 (n) → 运算符 (>) →

常量 (10000) → 分隔符 (()) → 标识符 (n) → 运算符 (=) → 常量 (10000) → 分隔符 (;)。if 关键字指明这是一个条件分支;greater 关键字指明大小判断，这是一个大于号;<Spelling=main.c:4:13> 表明这里有宏替换的痕迹，显示 10000 来自宏定义，来自 main.c 中的第 4 行第 13 列。

```

identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:21:5>
l_paren '(' [LeadingSpace] Loc=<main.c:21:11>
string_literal '"%d\n"' [LeadingSpace] Loc=<main.c:21:12>
comma ',' [LeadingSpace] Loc=<main.c:21:18>
identifier 'a' [LeadingSpace] Loc=<main.c:21:20>
r_paren ')' [LeadingSpace] Loc=<main.c:21:21>
semi ';' [LeadingSpace] Loc=<main.c:21:22>
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:22:5>
l_paren '(' [LeadingSpace] Loc=<main.c:22:11>
string_literal '"%d\n"' [LeadingSpace] Loc=<main.c:22:12>
comma ',' [LeadingSpace] Loc=<main.c:22:18>
identifier 'b' [LeadingSpace] Loc=<main.c:22:20>
r_paren ')' [LeadingSpace] Loc=<main.c:22:21>
semi ';' [LeadingSpace] Loc=<main.c:22:22>

```

图 4.12: 词法分析 7

接下来是初始输出段，词法序列为：标识符 (printf) → 分隔符 (()) → 常量 ("%d\n") → 分隔符 (,) → 标识符 (a/b) → 分隔符 (()) → 分隔符 (;)。printf 是标准输出函数标识符；string_literal 中的 "%d\n" 是格式字符串常量，其中的 "\n" 是转义字符表示换行；这里分别输出变量 a 和 b 的值，也就是斐波那契数列的前两个初始值。

```

while 'while' [StartOfLine] [LeadingSpace] Loc=<main.c:23:5>
l_paren '(' [LeadingSpace] Loc=<main.c:23:11>
identifier 'i' [LeadingSpace] Loc=<main.c:23:12>
less '<' [LeadingSpace] Loc=<main.c:23:14>
identifier 'n' [LeadingSpace] Loc=<main.c:23:16>
r_paren ')' [LeadingSpace] Loc=<main.c:23:17>
l_brace '{' [StartOfLine] [LeadingSpace] Loc=<main.c:24:5>
identifier 't' [StartOfLine] [LeadingSpace] Loc=<main.c:25:9>
equal '=' [LeadingSpace] Loc=<main.c:25:11>
identifier 'b' [LeadingSpace] Loc=<main.c:25:13>
semi ';' [LeadingSpace] Loc=<main.c:25:14>
identifier 'b' [StartOfLine] [LeadingSpace] Loc=<main.c:26:9>
equal '=' [LeadingSpace] Loc=<main.c:26:11>
identifier 'a' [LeadingSpace] Loc=<main.c:26:13>
plus '+' [LeadingSpace] Loc=<main.c:26:15>
identifier 'b' [LeadingSpace] Loc=<main.c:26:17>
semi ';' [LeadingSpace] Loc=<main.c:26:18>
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:27:9>
l_paren '(' [LeadingSpace] Loc=<main.c:27:15>
string_literal '"%d\n"' [LeadingSpace] Loc=<main.c:27:16>
comma ',' [LeadingSpace] Loc=<main.c:27:22>
identifier 'b' [LeadingSpace] Loc=<main.c:27:24>
r_paren ')' [LeadingSpace] Loc=<main.c:27:25>
semi ';' [LeadingSpace] Loc=<main.c:27:26>
identifier 'a' [StartOfLine] [LeadingSpace] Loc=<main.c:28:9>
equal '=' [LeadingSpace] Loc=<main.c:28:11>
identifier 't' [LeadingSpace] Loc=<main.c:28:13>
semi ';' [LeadingSpace] Loc=<main.c:28:14>
identifier 'i' [StartOfLine] [LeadingSpace] Loc=<main.c:29:9>
equal '=' [LeadingSpace] Loc=<main.c:29:11>
identifier 'i' [LeadingSpace] Loc=<main.c:29:13>
plus '+' [LeadingSpace] Loc=<main.c:29:15>
numeric_constant '1' [LeadingSpace] Loc=<main.c:29:17>
semi ';' [LeadingSpace] Loc=<main.c:29:18>
r_brace '}' [StartOfLine] [LeadingSpace] Loc=<main.c:30:5>

```

图 4.13: 词法分析 8

接下来是循环控制段，词法序列为：关键字 (while) → 分隔符 (()) → 标识符 (i) → 运算符 (<) → 标识符 (n) → 分隔符 (()) → 分隔符 ({}). while 是循环控制关键字；less 表示小于运算符 <，用于比较循环计数器 i 和上限 n；l_brace 表示循环体开始的左花括号，整个结构构成了标准的 while 循环条件判断。

接下来是循环体计算段，这是程序的核心逻辑部分：

第 25 行词法序列：标识符 (t) → 运算符 (=) → 标识符 (b) → 分隔符 (;)，这是将 b 的当前值保存到临时变量 t 中

第 26 行词法序列：标识符 (b) → 运算符 (=) → 标识符 (a) → 运算符 (+) → 标识符 (b) → 分隔符 (;)，这是斐波那契数列的核心计算 $b = a + b$

第 27 行词法序列：标识符 (printf) → 分隔符 (() → 常量 ("%d\n") → 分隔符 (,) → 标识符 (b) → 分隔符 ()) → 分隔符 (;)，用于输出当前计算出的斐波那契数值

第 28 行词法序列：标识符 (a) → 运算符 (=) → 标识符 (t) → 分隔符 (;)，将临时保存的旧 b 值赋给 a，完成数值交换

第 29 行词法序列：标识符 (i) → 运算符 (=) → 标识符 (i) → 运算符 (+) → 常量 (1) → 分隔符 (;)，这是循环计数器 i 的自增操作 $i = i + 1$

```
return 'return' [StartOfLine] [LeadingSpace] Loc=<main.c:32:5>
numeric_constant '0' [LeadingSpace] Loc=<main.c:32:12>
semi ';' Loc=<main.c:32:13>
r_brace '}' [StartOfLine] Loc=<main.c:33:1>
eof ' ' Loc=<main.c:33:2>
```

图 4.14: 词法分析 9

最后是循环结束和函数返回段，词法序列为：分隔符 (}) → 关键字 (return) → 常量 (0) → 分隔符 (;) → 分隔符 (}) → 文件结束 (eof)。第一个 r_brace 是循环体的结束花括号；return 是函数返回关键字；常量 0 表示程序正常退出码；第二个 r_brace 是 main 函数体的结束花括号；eof 标记表示源文件结束，至此，整个词法分析完成。

语法分析 对于语法分析，运行下面的命令，可以在终端里输出对语法树的具体描述。

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

我尝试过将其重定向到某个文件中，但会乱码，并且没有颜色渲染，效果不太好，故我这里的结果放的是终端里输出的截图。结果片段如下：

```
linglin@workspace:code$ clang -E -Xclang -ast-dump main.c
-VarDecl @x196bd28 <main.c:7:1, col:12> col:5 used a 'int' cinit
-IntegerLiteral @x196bd90 <col:12> 'int' 1
-FunctionDecl @x196bd108 <line:12:1, line:33:1> line:12:5 main 'int' ()
-CompoundStmt @x196bd3f0 <line:13:1, line:33:1>
-DeclStmt @x196bd458 <line:14:5, col:22>
-VarDecl @x196bd1b8 <col:5, col:9> col:9 used a 'int'
-VarDecl @x196bd238 <col:5, col:12> col:12 used b 'int'
-VarDecl @x196bd258 <col:5, col:15> col:15 used i 'int'
-VarDecl @x196bd338 <col:5, col:18> col:18 used t 'int'
-VarDecl @x196bd3b8 <col:5, col:21> col:21 used n 'int'
-BinaryOperator @x196bd4a8 <line:16:5, col:9> 'int' '+'
-DeclRefExpr @x196bd468 <col:5> 'int' lvalue Var @x196bd1b8 'a' 'int'
-IntegerLiteral @x196bd488 <col:9> 'int' 0
-BinaryOperator @x196bd588 <col:5, col:9> 'int' '+'
-DeclRefExpr @x196bd4c8 <col:5> 'int' lvalue Var @x196bd238 'b' 'int'
-IntegerLiteral @x196bd4e8 <col:9> 'int' 1
-BinaryOperator @x196bd5d8 <line:18:5, col:9> 'int' '+'
-DeclRefExpr @x196bd528 <col:5> 'int' lvalue Var @x196bd258 'i' 'int'
-IntegerLiteral @x196bd548 <col:9> 'int' 1
-CallExpr @x196bd5f8 <line:19:5, col:19> 'int'
-ImplicitCastExpr @x196bd6d8 <col:5> 'int (*) (const char *, ...)' <FunctionToPointerDecay>
-DeclRefExpr @x196bd6d0 <col:5> 'int (const char *, restrict, ...) Function @x196bd24c0 'scanf' 'int (const char *, restrict, ...)'
-ImplicitCastExpr @x196bd638 <col:11> 'const char *' <NoOp>
-ImplicitCastExpr @x196bd6c0 <col:11> 'char *' <ArrayToPointerDecay>
-StringLiteral @x196bdad8 <col:11> 'char[3]' lvalue "3d"
UnaryOperator @x196bd698 <col:17, col:18> 'int' prefix '!' cannot overflow
-DeclRefExpr @x196bdaf8 <col:18> 'int' lvalue Var @x196bd3b8 'n' 'int'
-IntegerLiteral @x196bd548 <col:9> 'int' 1
-IfStmt @x196bd28 <line:20:5, line:4:13>
-BinaryOperator @x196bdca8 <line:20:8, line:4:13> 'int' '>'
-DeclRefExpr @x196bd98 <col:5> 'int' <lvalue to rvalue>
-DeclRefExpr @x196bd59 <col:8> 'int' lvalue Var @x196bd3b8 'n' 'int'
-IntegerLiteral @x196bd70 <line:4:13> 'int' 10000
-BinaryOperator @x196bdd88 <line:20:15, line:4:13> 'int' '+'
-DeclRefExpr @x196bdcc8 <line:20:15> 'int' lvalue Var @x196bd3b8 'n' 'int'
-IntegerLiteral @x196bd6e8 <line:4:13> 'int' 10000
-CallExpr @x196bdd8 <line:21:5, col:21> 'int'
-ImplicitCastExpr @x196bddc0 <col:5> 'int (*) (const char *, ...)' <FunctionToPointerDecay>
-DeclRefExpr @x196bdd48 <col:5> 'int (const char *, ...) Function @x196a1e78 'printf' 'int (const char *, ...)'
-ImplicitCastExpr @x196bd98 <col:12> 'const char *' <NoOp>
-ImplicitCastExpr @x196bd98 <col:12> 'char *' <ArrayToPointerDecay>
-StringLiteral @x196bdd68 <col:12> 'char[4]' lvalue "3d\n"
-DeclRefExpr @x196bd38 <col:20> 'int' <lvalue to rvalue>
-DeclRefExpr @x196bdd88 <col:20> 'int' lvalue Var @x196bd1b8 'a' 'int'
-CallExpr @x196bd6e8 <line:22:5, col:21> 'int'
```

图 4.15: 抽象语法树层次图

我们将分析的重点部分放在 main 函数中，通过总结，我们可以知道大概如下的层次图：

```

FunctionDecl 'main' (返回int)
├─ CompoundStmt (函数体)
│   ├─ DeclStmt (声明变量)
│   │   ├─ VarDecl 'a' (int)
│   │   ├─ VarDecl 'b' (int)
│   │   ├─ VarDecl 'i' (int)
│   │   ├─ VarDecl 't' (int)
│   │   └─ VarDecl 'n' (int)
│   ├─ BinaryOperator '=' (初始化a=0)
│   ├─ BinaryOperator '=' (初始化b=1)
│   ├─ BinaryOperator '=' (初始化i=1)
│   ├─ CallExpr (调用scanf读取n的值)
│   ├─ IfStmt (检查n是否>10000)
│   ├─ CallExpr (printf输出a的值)
│   ├─ CallExpr (printf输出b的值)
│   └─ WhileStmt (循环)
│       ├─ BinaryOperator '<' (循环条件i<n)
│       └─ CompoundStmt (循环体)
│           ├─ BinaryOperator '=' (t=b)
│           ├─ BinaryOperator '=' (b=a+b)
│           ├─ CallExpr (printf输出b的值)
│           ├─ BinaryOperator '=' (a=t)
│           └─ BinaryOperator '=' (i=i+1)
└─ ReturnStmt (返回0)
  
```

图 4.16: 精简抽象语法树

结构显而易见，接下来我们进一步分析这个抽象语法树每个节点的抽象概念代表了什么意思。

- **FunctionDecl** 表示一个函数声明，是抽象语法树的根节点，包含了函数的名称、返回类型和参数列表等信息；
- **CompoundStmt** 表示复合语句，用于包裹函数体中的多个语句；
- **DeclStmt** 表示声明语句，用于批量声明变量，这里一次性声明了五个整型变量 a、b、i、t、n；
- **VarDecl** 表示具体的变量声明，每个变量都有独立的内存地址标识和源代码位置信息；
- **BinaryOperator** 表示二元运算操作，包括赋值运算、算术运算和比较运算等；
- **DeclRefExpr** 表示对已声明变量的引用，用于在表达式中使用变量；
- **IntegerLiteral** 表示整数字面量常量，如 0、1 等具体数值；
- **CallExpr** 表示函数调用表达式，包括标准库函数 scanf 和 printf 的调用；
- **IfStmt** 表示条件判断语句，用于实现条件分支逻辑；
- **WhileStmt** 表示循环控制语句，用于实现重复执行逻辑；

- **ImplicitCastExpr** 表示编译器自动插入的隐式类型转换，包括函数到指针的转换、左值到右值的转换等；
- **ReturnStmt** 表示函数返回语句，用于结束函数执行并返回结果。

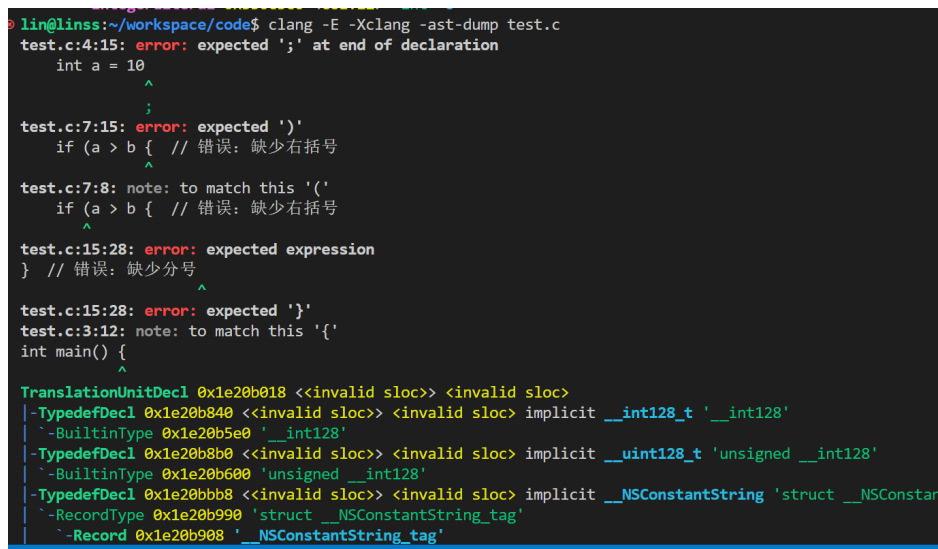
为了进一步验证编译器在语法分析阶段对程序的语法错误检查功能，我们这个编写一个错误的 test.c 程序。

```

1 #include <stdio.h>
2
3 int main() {
4     int a = 10
5     int b = 20; // 错误：上一行缺少分号
6
7     if (a > b { // 错误：缺少右括号
8         printf("a is greater than b\n")
9         // 错误：缺少分号
10    }
11
12    int array[5] = {1, 2, 3, 4 // 错误：数组初始化不完整
13
14    return 0
15 } // 错误：缺少分号

```

我们依旧执行 `clang -E -Xclang -ast-dump main.c` 命令，可以看到在生成抽象语法树之前，打印了一些错误信息。我们可以看到这覆盖了我们设置的一部分信息，但是比如说第 4 行缺少分号的错误被准确报告，但第 8 行 `printf` 语句缺少分号和第 14 行 `return` 语句缺少分号的错误并未显示，这有很大可能的原因是因为语法分析器的错误恢复机制，当遇到一个错误时，它的恢复机制可能会影响后续的分析，这也体现了语法分析器的局限性，存在一定的误导性。



```

lin@linss:~/workspace/code$ clang -E -Xclang -ast-dump test.c
test.c:4:15: error: expected ';' at end of declaration
    int a = 10
              ^
test.c:7:15: error: expected ')'
    if (a > b { // 错误：缺少右括号
              ^
test.c:7:8: note: to match this '('
    if (a > b { // 错误：缺少右括号
        ^
test.c:15:28: error: expected expression
    } // 错误：缺少分号
      ^
test.c:15:28: error: expected '}'
test.c:3:12: note: to match this '{'
int main() {
    ^
TranslationUnitDecl 0x1e20b018 <<invalid sloc>> <invalid sloc>
|-TypeDecl 0x1e20b840 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `--BuiltinType 0x1e20b5e0 '__int128'
|-TypeDecl 0x1e20b8b0 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `--BuiltinType 0x1e20b600 'unsigned __int128'
|-TypeDecl 0x1e20bbb8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `--RecordType 0x1e20b990 'struct __NSConstantString_tag'
| `--Record 0x1e20b908 '__NSConstantString_tag'

```

图 4.17: 语法错误

语义分析 语义分析的核心任务是确保代码的**逻辑正确性**，而不仅仅是语法正确性。语义分析器通过类型检查、作用域分析、控制流分析等手段，确保程序在运行时不会出现未定义的行为或发生逻辑错误。

误等。此外，语义分析的结果会间接影响后续的**中间代码生成**和**代码优化**的效果，所以我认为这一步在编译过程中是很必要的，是编译器深入理解代码的基础。其主要功能包括：

- **类型检查**：验证变量、表达式和操作的数据类型是否匹配，确保类型转换和运算符符合语言规范
- **声明与作用域分析**：检查标识符是否在使用前声明，验证变量和函数的作用域范围是否合法
- **函数调用验证**：确认被调用函数存在且参数数量、类型匹配，检查返回值使用是否正确
- **控制流检查**：确保 break、continue、return 等控制语句在合法上下文中使用，检测不可达代码
- **复合数据类型检查**：验证数组访问和结构体成员访问的合法性，检查复合类型的初始化是否正确
- **常量表达式求值**：在编译时计算常量表达式的值，检查常量变量的不可修改性，防止其被意外修改
- **符号表管理**：维护所有标识符的类型、作用域等属性信息，为后续编译阶段**提供符号查询支持**

中间代码生成 我们运行如下的命令，即可得到 main.ll 文件，存储的就是我们生成的 llvm 中间代码。

```
1 clang -S -emit-llvm main.c
```

代码如下所示：

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

这段中间代码的最初，说明了目标平台的相关信息，datalayout 指定了目标平台的数据布局，包括大小、对齐方式等。target triple 是目标平台的三元组，这里表示 x86_64 架构的 Linux 系统。

```
1 @myflag = dso_local global i32 1, align 4
2 @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
3 @.str.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
```

@myflag: 定义了一个整数类型的全局变量，初始值为 1。此变量在程序中并未被使用，只是为了测试。

@.str: 字符串常量，格式为"%d"，用于格式化输出一个整数，服务于后面的 scanf 函数。

@.str.1: 字符串常量，格式为"%d\n"，用于格式化输出一个整数并换行，服务于后面的 printf 函数。

```
1 ; Function Attrs: noinline nounwind optnone uwtable
2 define dso_local i32 @main() #0 {
```

@main 是程序的入口函数，返回值为 i32（即 32 位整数）。

```
1     %1 = alloca i32, align 4
2     %2 = alloca i32, align 4
3     %3 = alloca i32, align 4
4     %4 = alloca i32, align 4
```

```

5  %5 = alloca i32, align 4
6  %6 = alloca i32, align 4

```

在函数的开头，使用 `alloca` 指令分配了多个局部变量，这些变量都为 `i32` 类型，并且每个都对齐到 4 字节。对应 “`int a, b, i, t, n;`” 这一句；

```

1  store i32 0, i32* %1, align 4
2  store i32 0, i32* %2, align 4
3  store i32 1, i32* %3, align 4
4  store i32 1, i32* %4, align 4

```

这里对局部变量进行了初始化的操作，将整数 0 存储到 `%1` 和 `%2` 中，将整数 1 存储到 `%3` 和 `%4` 中。对应 “`a = 0; b = 1; i = 1;`” 这三句；

```

1  %7 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds ([3 x
    i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %6)

```

`@__isoc99_scanf` 是 C 标准库函数 `scanf` 的调用，用于读取用户输入，`%6` 是一个整型变量，用户输入的整数会存储在这里。对应 “`scanf("%d", &n);`” 这一句；

```

1  %8 = load i32, i32* %6, align 4
2  %9 = icmp sgt i32 %8, 10000
3  br i1 %9, label %10, label %11

```

读取用户输入的值并存储在 `%8` 中。判断输入的值是否大于 10000，如果是，则跳转到标签 `%10`，否则跳转到标签 `%11`。

```

1  10:                                ; preds = %0
2  store i32 10000, i32* %6, align 4
3  br label %11

```

如果输入值大于 10000，将 `%6` 的值设置为 10000。这两段对应 “`if(n>MAX) n=MAX;`” 这一句；另外 “`;`+ 文本” 在 `llvm` 中间代码中表示注释；

```

1  11:                                ; preds = %10, %0
2  %12 = load i32, i32* %2, align 4
3  %13 = call i32 @printf(i8* noundef getelementptr inbounds ([4 x i8], [4
    x i8]* @.str.1, i64 0, i64 0), i32 noundef %12)
4  %14 = load i32, i32* %3, align 4
5  %15 = call i32 @printf(i8* noundef getelementptr inbounds ([4 x i8], [4
    x i8]* @.str.1, i64 0, i64 0), i32 noundef %14)
6  br label %16

```

如果输入值大于 10000 的条件不满足，那么顺序执行 11 标签的内容，调用 `@printf` 输出 `%2` 中的整数（初始为 0），调用 `@printf` 输出 `%3` 中的整数（初始为 1）。对应 “`printf("%d\n", a); printf("%d\n", b);`” 这两句；

```

1  16:                                ; preds = %20, %11
2  %17 = load i32, i32* %4, align 4
3  %18 = load i32, i32* %6, align 4

```



```

4  %19 = icmp slt i32 %17, %18
5  br i1 %19, label %20, label %30

```

判断%4 (i) 中的值是否小于%6 (n) 中的值，如果是，跳转到标签%20，否则结束循环（跳转到%30）。对应 “while (i < n)”；

```

1  20:                                     ; preds = %16
2  %21 = load i32, i32* %3, align 4
3  store i32 %21, i32* %5, align 4
4  %22 = load i32, i32* %2, align 4
5  %23 = load i32, i32* %3, align 4
6  %24 = add nsw i32 %22, %23
7  store i32 %24, i32* %3, align 4
8  %25 = load i32, i32* %3, align 4
9  %26 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds ([4 x i8], [4
    x i8]* @.str.1, i64 0, i64 0), i32 noundef %25)
10 %27 = load i32, i32* %5, align 4
11 store i32 %27, i32* %2, align 4
12 %28 = load i32, i32* %4, align 4
13 %29 = add nsw i32 %28, 1
14 store i32 %29, i32* %4, align 4
15 br label %16, !llvm.loop !6

```

首先将%3 (b) 中的值存储到%5 (t)。接着将%2 (a) 和%3 中的值相加，并将结果存储回%3。输出%3 的值（斐波那契数列求解结果）。更新%2 和%4 的值，%2 设为%5 的值，%4 增加 1。跳转回标签%16，继续循环。对应 “t = b;b = a + b;printf(“%d\n”, b);a = t;i = i + 1;” 所描述的循环体内部。这里很好理解，我们不再赘述。

```

1  30:                                     ; preds = %16
2  ret i32 0
3  }

```

当循环条件不满足时，返回 0 结束程序。

```

1  declare i32 @__isoc99_scanf(i8* noundef, ...) #1
2  declare i32 @printf(i8* noundef, ...) #1

```

这里是外部函数声明,@__isoc99_scanf 是标准库函数 scanf 的声明，用于读取用户输入。@printf 是标准库函数 printf 的声明，用于格式化输出。

```

1  attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all"
    "min-legal-vector-width"="0" "no-trapping-math"="true"
    "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
    "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
2  attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true"
    "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
    "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

```

#0 和 #1 是分别为 main 函数和外部函数 printf、scanf 设置的属性。这些属性告知编译器如何优化

这些函数：

- **noinline**: 禁止内联优化。
- **nounwind**: 表示这些函数不会抛出异常。
- **optnone**: 禁止任何优化。
- **uwtable**: 允许生成 unwind 表（用于异常处理）。

```
1 !llvm.module.flags = !{!0, !1, !2, !3, !4}
2 !llvm.ident = !{!5}
3
4 !0 = !{i32 1, !"wchar_size", i32 4}
5 !1 = !{i32 7, !"PIC Level", i32 2}
6 !2 = !{i32 7, !"PIE Level", i32 2}
7 !3 = !{i32 7, !"uwtable", i32 1}
8 !4 = !{i32 7, !"frame-pointer", i32 2}
9 !5 = !{"Ubuntu clang version 14.0.0-1ubuntu1.1"}
10 !6 = distinct !{!6, !7}
11 !7 = !{"llvm.loop.mustprogress"}
```

这些定义了一些元数据，主要提供给编译器和调试器用于优化、生成代码、调试和分析。它们并不会直接影响程序的运行行为，而是影响编译过程中的生成策略和调试信息。在 LLVM 中间代码中，元数据通常用于描述与程序逻辑无关的附加信息，如编译器设置、优化策略、调试信息等。

比如说,!6 和!7 与**循环优化**相关,尤其是优化器对循环的分析和控制。!7 中”llvm.loop.mustprogress”是一个特定的标志,表示循环必须进行某种进展,即每次迭代循环时,条件必须发生变化(比如我们这里的%4(对应变量 i) 计数器递增)。这是为了确保 LLVM 在优化时不会将循环视为不变的,从而进行某些不合理的优化。而!6 中的 distinct 表示这是一个特殊的标志,和!7 一起用来表示特定循环进度要求。

进行了 llvm 中间代码的分析之后,我们也可以采用 gcc -fdump-tree-all-graph main.c 的命令来生成中间代码的调试信息,这个命令会生成大量.dot 文件,输出的.dot 文件可以通过 Graphviz 工具可视化为图形,描述源代码 main.c 从语法分析到优化的每个中间阶段的树形结构。结果如下所示,

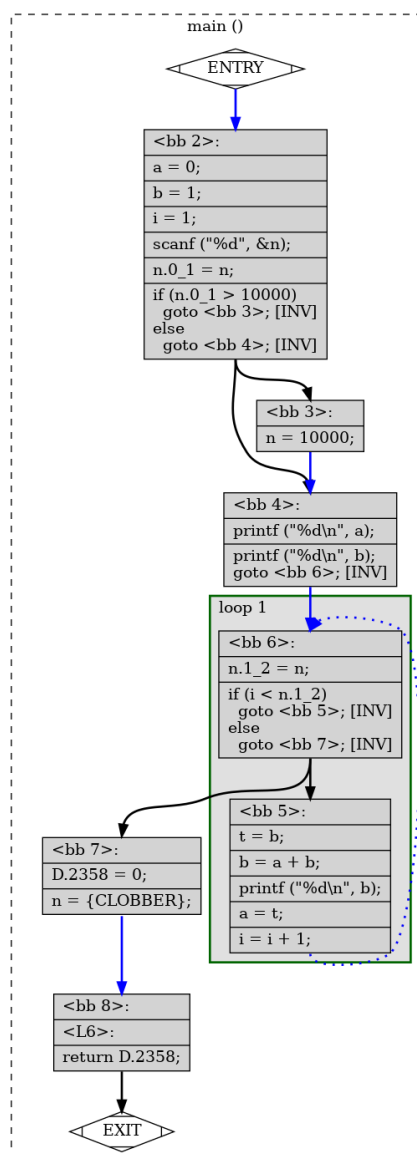


图 4.18: a-main.c.015t.cfg.dot

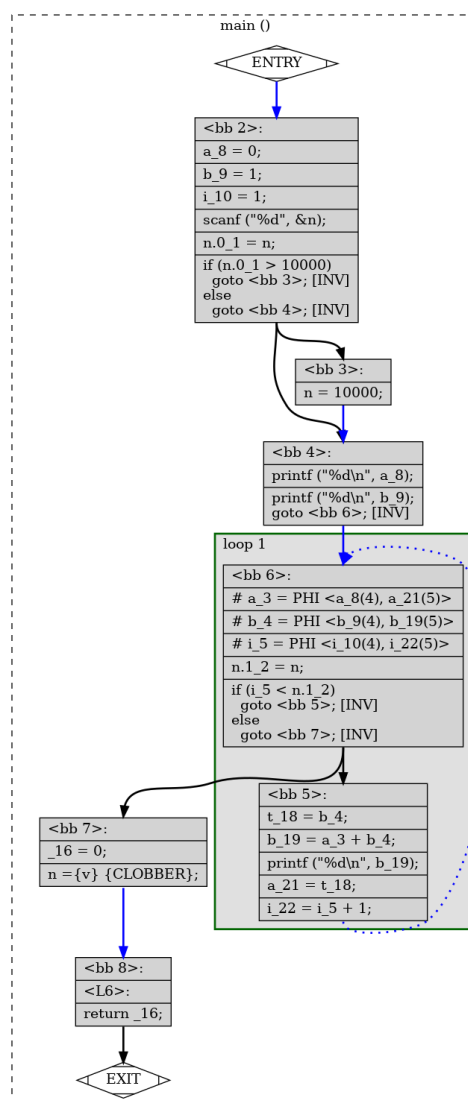


图 4.19: a-main.c.244t.optimized.dot

我们先看 a-main.c.015t.cfg.dot 这张图，描述了程序的初始控制流图，ENTRY 是程序入口点，EXIT 是程序的出口点，从控制流角度分析，该图包括了程序的基本块（如 bb2, bb3, bb4, 等）及其之间的控制流，尤其是跳转、条件分支、循环等。

再来看 a-main.c.244t.optimized.dot 这张图，应该是编译器优化后的流程图。与第一张图相比，第一个不同之处就是对于每一个变量，它的下面多了一个类似角标的标记，可能是编译器对变量进行了重命名或重新分配；第二个不同之处就是在 loop1 处引入了 PHI 节点，在循环优化或条件分支中，常常使用 PHI 节点来表示多个路径合并时变量值的选择，这样做可能有利于循环优化；第三个不同之处就是使用了更多的变量（a_8 和 a_3 是不同的变量），可能有利于程序的优化。

总之，通过编译器的作用，第二张图对应的程序可能在数据流或内存空间上得到了相应的优化。

中间代码优化 采用如下命令进行中间代码优化。

```

1 opt -S -O0 main.ll -o main-O0.ll # 无优化
2 opt -S -O1 main.ll -o main-O1.ll # 基础优化

```

```

3 | opt -S -O2 main.ll -o main-O2.ll # 标准优化
4 | opt -S -O3 main.ll -o main-O3.ll # 激进优化

```

可以看到我们通过运行命令，成功得到了四个结果文件。

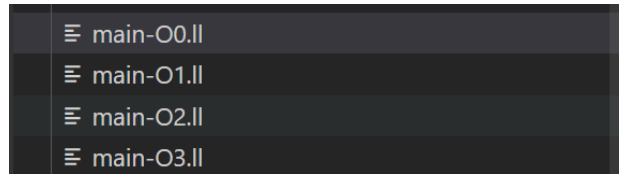


图 4.20: 中间代码优化结果

我们可以使用如下的命令查看未优化版本与 O1 优化版本的不同之处。我也尝试过 O2、O3 对比，但是与 O1 完全一样，可能是程序太简单导致 O1 优化已经达到了上限，O2、O3 更高级的优化在此程序中并未得到体现。

```

1 | diff main-O0.ll main-O1.ll

```

```

lin@linss:~/workspace/code$ diff main-O0.ll main-O1.ll
6c6
< @myflag = dso_local global i32 1, align 4
---
> @myflag = dso_local local_unnamed_addr global i32 1, align 4
11c11
< define dso_local i32 @main() #0 {
---
> define dso_local i32 @main() local_unnamed_addr #0 {
64c64,65
< declare i32 @__isoc99_scanf(i8* noundef, ...) #1
---
> ; Function Attrs: nofree nounwind
> declare noundef i32 @__isoc99_scanf(i8* nocapture noundef readonly, ...) local_unnamed_addr #1
66c67,68
< declare i32 @printf(i8* noundef, ...) #1
---
> ; Function Attrs: nofree nounwind
> declare noundef i32 @printf(i8* nocapture noundef readonly, ...) local_unnamed_addr #1
69c71
< attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"
="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
---
> attributes #1 = { nofree nounwind "frame-pointer"="all" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "
target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

```

图 4.21: 未优化与 O1 优化对比

我们对其进行详细分析：

```

1 | 6c6
2 | < @myflag = dso_local global i32 1, align 4
3 | ---
4 | > @myflag = dso_local local_unnamed_addr global i32 1, align 4

```

在 main-O1.ll 文件中，@myflag 变量的定义发生了变化，新增了 local_unnamed_addr 这个属性，用于表示该全局变量是局部的，并且可以进行优化，用来告诉编译器这个变量不需要具名，可以在链接时进行优化（例如移除或合并）。

```

1 | 11c11
2 | < define dso_local i32 @main() #0 {
3 | ---
4 | > define dso_local i32 @main() local_unnamed_addr #0 {

```

函数 @main 的定义中，添加了 local_unnamed_addr 属性，类似于前面的 @myflag 变量。这意味着 @main 函数在优化过程中可以被优化或重命名，进一步减少不必要的符号链接或函数名称的占用。

```

1 64c64,65
2 < declare i32 @__isoc99_scanf(i8* noundef, ...) #1
3 ____
4 > ; Function Attrs: nofree nounwind
5 > declare noundef i32 @__isoc99_scanf(i8* nocapture noundef readonly, ...)
    local_unnamed_addr #1

```

@__isoc99_scanf 函数声明发生了以下变化：

- **noundef**: 表示该函数的参数不允许为 undefined，即不能传入无效值。
- **nocapture**: 这个标志表示传入的指针参数不会被修改或捕获，用来帮助编译器进行优化。
- **readonly**: 标识该函数的参数是只读的，告诉编译器可以安全地对其进行优化。
- **local_unnamed_addr**: 将该函数的符号标记为可以优化，类似于 @myflag 的处理，表示该函数可以被优化为本地函数。

```

1 66c67,68
2 < declare i32 @printf(i8* noundef, ...) #1
3 ____
4 > ; Function Attrs: nofree nounwind
5 > declare noundef i32 @printf(i8* nocapture noundef readonly, ...) local_unnamed_addr
    #1

```

类似于 @__isoc99_scanf, @printf 的函数声明也进行了优化。增加了 noundef, nocapture, readonly, 和 local_unnamed_addr 标记，以帮助编译器优化对该函数的调用。

```

1 69c71
2 < attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true"
    "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
    "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
3 ____
4 > attributes #1 = { nofree nounwind "frame-pointer"="all" "no-trapping-math"="true"
    "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
    "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

```

添加了 nofree, nounwind 这两个新的属性，表示该函数不会释放内存，也不会抛出异常。这有助于优化函数的调用，避免对函数调用进行不必要的异常处理或内存回收操作。

通过上面的分析，我们可以发现，中间代码优化的本质应该就是添加一些属性值，告诉编译器哪块它可以进行优化，哪块不能进行优化，从而在保证程序正确性的前提下，提高程序的运行效率，具体的优化方式应该是在后续阶段生成目标代码时得到体现。

为了进一步探究优化的效果究竟如何，我们重新编写 main.c 文件，使得其作 100000 次迭代，得到对应的运行总时间来进行比对。

```

1 #include <stdio.h>

```

```
2 #include <time.h>
3 //这是一个计算斐波那契数列的程序 最大输入n为10000 超过10000则取10000
4 #define MAX 10000
5 #define FLAG
6 #ifdef FLAG
7     int myflag=1;
8 #else
9     int myflag=0;
10 #endif
11
12 int main()
13 {
14     int a, b, i, t, n;
15     clock_t start, end;
16     double total_time = 0;
17     int iterations = 100000;
18     scanf("%d", &n);
19     if(n>MAX) n=MAX;
20     for(int j = 0; j < iterations; j++) {
21         a = 0;
22         b = 1;
23         i = 1;
24         //printf("%d\n", a);
25         //printf("%d\n", b);
26         start = clock();
27         while (i < n)
28         {
29             t = b;
30             b = a + b;
31             //printf("%d\n", b);
32             a = t;
33             i = i + 1;
34         }
35         end = clock();
36         total_time += ((double)(end - start)) / CLOCKS_PER_SEC;
37     }
38     printf("total time: %f seconds\n", total_time);
39     return 0;
40 }
```

所有需要的指令如下：

```
1 clang -S -emit-llvm main.c
2 opt -S -O0 main.ll -o main-O0.ll # 无优化
3 opt -S -O1 main.ll -o main-O1.ll # 基础优化
4 opt -S -O2 main.ll -o main-O2.ll # 标准优化
5 opt -S -O3 main.ll -o main-O3.ll # 激进优化
6 llvm-as main-O0.ll -o main-O0.bc
7 llvm-as main-O1.ll -o main-O1.bc
```

```

8 | llvm-as main-O2.ll -o main-O2.bc
9 | llvm-as main-O3.ll -o main-O3.bc
10 | llc main-O0.bc -filetype=obj -o main_llvm-O0.o
11 | llc main-O1.bc -filetype=obj -o main_llvm-O1.o
12 | llc main-O2.bc -filetype=obj -o main_llvm-O2.o
13 | llc main-O3.bc -filetype=obj -o main_llvm-O3.o
14 | clang main_llvm-O0.o -o main_llvm-O0
15 | clang main_llvm-O1.o -o main_llvm-O1
16 | clang main_llvm-O2.o -o main_llvm-O2
17 | clang main_llvm-O3.o -o main_llvm-O3
18 | ./main_llvm-O0
19 | ./main_llvm-O1
20 | ./main_llvm-O2
21 | ./main_llvm-O3

```

我们选取 $n=10$ 、 100 、 1000 、 10000 这几个规模对四个程序进行测试，结果如下：从折线图来看，优

表 2: Performance Measurements with Different Optimization Levels

n	O0	O1	O2	O3
10	2.51310e-2	2.87720e-2	1.67100e-2	2.04350e-2
100	4.15210e-2	3.46700e-2	3.85630e-2	2.98290e-2
1000	1.18920e-1	1.11801e-1	1.36684e-1	1.06764e-1
10000	8.77061e-1	8.27323e-1	7.55549e-1	7.10892e-1

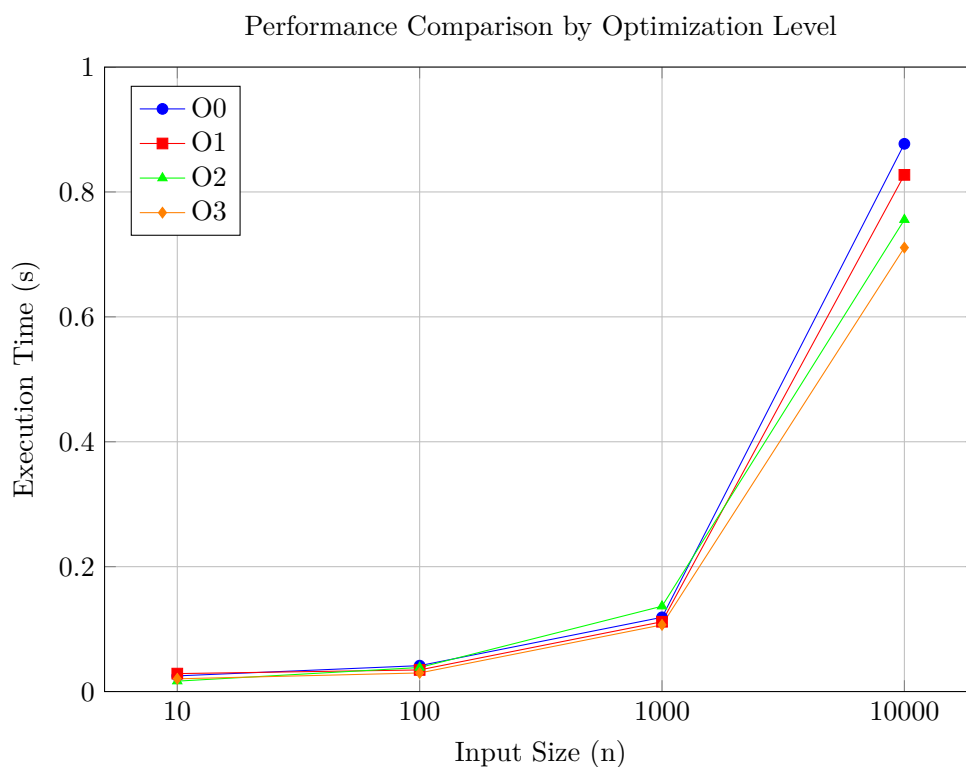


图 4.22: Performance trends with different optimization levels

化效果还是比较明显的，并且随着数据规模的增大，优化效果更加明朗，并且也比较稳定。

基于 g++ 编译器编译时的优化选项进行调研：

-O1 优化选项

-O1 是 GCC/G++ 的默认优化级别，它提供了一组合理的优化选项，旨在不显著增加编译时间的前提下提高程序的性能。-O1 包括的一些优化技术有：

函数内联：对一些小型函数进行内联展开，减少函数调用的开销。

常量传播：将代码中的常量表达式在编译时计算，减少运行时的计算量。

死代码消除：删除程序中永远不会执行的代码，减少程序的大小。

-O2 优化选项

-O2 优化级别在-O1 的基础上，进一步增加了更多的优化技术，以获得更好的性能提升。这些技术包括：

循环优化：包括循环展开、循环交换等，以减少循环的开销并提高缓存的利用率。

分支预测：优化分支指令，减少分支错误预测的可能性。

指令调度：重新排列指令的执行顺序，以提高流水线的利用率。

-O3 优化选项

-O3 优化级别包括-O2 的所有优化，并进一步应用了一些更激进的优化技术，这些技术可能会增加编译时间，但能获得更多的性能提升。例如：

进一步的指令调度和循环优化。

更激进的函数内联：对更大的函数进行内联展开。

浮点运算优化：对浮点运算进行特定的优化，以提高性能。

使用编译器标志

编译器标志是指导编译器应用特定优化技术的重要手段。通过合理使用这些标志，开发者可以显著提高程序的性能，减少资源消耗，并缩短开发周期。例如-ffast-math（链接时优化）和-march=native（为目标 CPU 优化）。

性能关键路径：可以使用-unroll-loops（循环展开）和-ftree-vectorize（循环向量化）等标志进行针对性优化。

内存敏感应用：可以使用-fdata-sections 和-ffunction-sections 标志来启用数据和函数级别的链接优化，从而减少程序的总内存占用。

代码大小优化：可以使用-Os 标志来优化代码大小，同时尽量保持性能。

另外，使用-fopt-info-优化级别标志可以生成优化信息报告。

目标代码生成 采用以下三种指令可以生成不同格式的汇编代码。

```
1 gcc main.i -S -o main_x86.S # 生成 x86 格式目标代码
2 riscv64-unknown-elf-gcc -march=rv64gc -mabi=lp64d main.c -S -o main_riscv.S
   #riscv64 格式
3 llc main.ll -o main_llvm.S # LLVM 生成目标代码
```

运行结果如下，可以看到我们成功生成了 x86 格式、riscv64 格式、LLVM 格式（与一步编译的结果不同，因为我这里用的是 clang 编译器，一步编译用的是 gcc 编译器）的汇编代码，三种代码的风格完全不同。汇编代码的分析我们在此处先暂不提及，放到后面任务二再对 riscv 汇编代码进行分析。

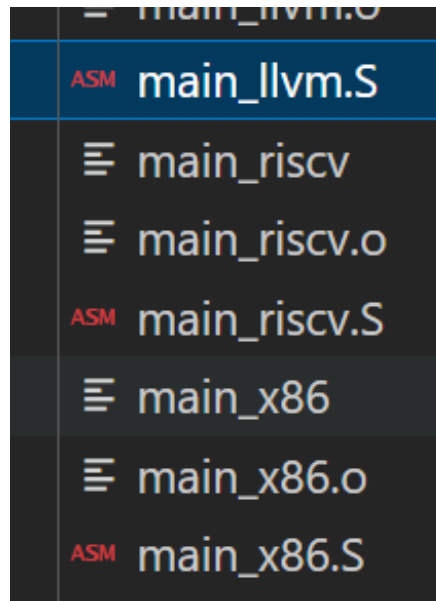


图 4.23: 目标代码生成

4.4 汇编器做了什么？

4.4.1 功能探究

汇编器将 main.s 中的**汇编语言**翻译成**机器语言指令**，把这些指令打包成一种叫做**可重定位目标程序**的格式，**生成重定位信息**，标记需要链接器后续处理的位置，并将结果保存在目标文件 main.o 中，main.o 是一个二进制文件。

关于上面说到的重定位，其实主要就是，我们通过汇编器得到的 main.o 这个二进制文件，并不能直接运行，我们需要后续通过链接外部库，才能够生成可执行的目标程序。而链接的过程，就是把多个文件合并成一个文件，不乏会有很多文件之间的相互调用，比如我们程序中的 printf 函数，是外部文件的库函数，函数调用需要目标函数的地址，而在没有进行链接前，printf 函数在 main.o 中的目标地址为空值，需要后续链接外部库，通过重定位技术，就可以把这个**空白地址替换为真实的内存地址**。

4.4.2 实验验证

汇编阶段运行指令如下：

```
1 gcc main_x86.S -c -o main_x86.o
2 riscv64-unknown-elf-as -march=rv64gc -mabi=lp64d main_riscv.S -o main_riscv.o
3 llc main.bc -filetype=obj -o main_llvm.o
```

结果在图5.30可以得到体现。

可以用下面的指令对 riscv.o 进行反汇编进一步验证。运行之后结果保存在 main_riscv_r.s 中。

```
1 riscv64-unknown-elf-objdump -D main_riscv.o > main_riscv_r.s
```

结果如下，我们逐段来进行分析：

```
1 main_riscv.o:      file format elf64-littleriscv
2 Disassembly of section .text:
```



```

3
4 0000000000000000 <main>:
5     0: 7179          addi   sp,sp,-48
6     2: f406          sd     ra,40(sp)
7     4: f022          sd     s0,32(sp)
8     6: 1800          addi   s0,sp,48

```

首先这里是 main 函数的入口处，我们向上开辟了 48 字节大小的栈空间，用于保存局部变量和保存返回地址等信息。

然后我们把 **ra 寄存器的值（即函数返回地址）** 加载到了栈底（sp 栈指针所指向的位置），将**帧指针寄存器 s0** 的值保存到栈中，帧指针 s0 被用来维护函数调用的栈帧，在进入函数时，s0 通常指向栈的基地址，也就是需要被更新，所以我们要保存 s0，以便函数结束后恢复。

然后将栈指针 sp 加上 48，并将结果存入 s0。这条指令的作用就是将 s0 设置为当前栈帧的基址，然后被用于后续访问局部变量和栈上的数据。

通过这些操作，我们完成了栈帧的设置：为当前函数分配了 48 字节的栈空间，并保存 ra 和 s0 寄存器的值以便函数返回后恢复。

```

1     8: fe042623        sw     zero,-20(s0)
2    c: 4785          li     a5,1
3    e: fef42423        sw     a5,-24(s0)
4   12: 4785          li     a5,1
5   14: fef42223        sw     a5,-28(s0)

```

然后这几条指令就是完成下面的三行代码：

a = 0: 将常量 0 存储到栈上，地址 -20(s0)。

b = 1: 将常量 1 存储到寄存器 a5，并保存到栈 -24(s0)。

i = 1: 将常量 1 存储到寄存器 a5，并保存到栈 -28(s0)

这样，我们完成了局部变量的初始化操作，-20 (s0) 代表目标地址是-20+s0。

```

1   18: fdc40793        addi   a5,s0,-36
2  1c: 85be          mv     a1,a5
3  1e: 000007b7        lui    a5,0x0
4  22: 00078513        mv     a0,a5
5  26: 00000097        auipc  ra,0x0
6  2a: 000080e7        jalr   ra,#26 <main+0x26>

```

首先 18 这条指令对应的操作会把-36 (s0) 对应的地址存到 a5 寄存器中，我们可以发现，通过与原来 main.c 程序的对比，我们声明了 5 个局部变量，分别为 a, b, i, t, n，那么这里的-36 (s0) 对应的就是局部变量 n，即我们将 n 的地址放在了 a5 寄存器中。

然后 1c 这条指令将 a5 寄存器保存的值传入到了 a1 寄存器中，而 a1 寄存器是 RISC-V 中用于传递函数参数的寄存器，所以这里的操作就是传入 scanf 函数的第一个参数，这也说明了从源程序来看，参数是从右向左入栈的。

接着 1e 和 22 这两条指令传入了 scanf 函数的第二个参数，即%d。

但是由于我们没链接外部库，所以这里的跳转指令并不会调用 scanf 函数，这是一个需要被后续加载操作所填充的空白值。

总之，这段代码会将变量 n 的地址传递给 scanf，通过系统调用从用户输入中读取一个整数值。

```

1 2e: fdc42703          lw  a4,-36(s0)
2 32: 6789              lui  a5,0x2
3 34: 71078793          addi a5,a5,1808 # 2710 <.L3+0x265c>
4 38: 00e7d763          bge a5,a4,46 <.L2>

```

$n > \text{MAX}$: 首先加载 n 的值到 $a4$ 寄存器中, 然后将 MAX (即 10000) 加载到寄存器 $a5$ 中, 使用 `bge` 指令判断是否满足 $n > \text{MAX}$, 如果不是, 即 $n \leq \text{MAX}$, 跳转到标签 46 处。

```

1 3c: 6789              lui  a5,0x2
2 3e: 71078793          addi a5,a5,1808 # 2710 <.L3+0x265c>
3 42: fcf42e23          sw  a5,-36(s0)

```

若 $n > \text{MAX}$ 满足, 则顺序执行, 使得 $n = \text{MAX}$, 满足我们限制的 n 不大于 MAX 的要求。

```

1 0000000000000046 <.L2>:
2 46: fec42783          lw  a5,-20(s0)
3 4a: 85be              mv  a1,a5
4 4c: 000007b7          lui  a5,0x0
5 50: 00078513          mv  a0,a5
6 54: 00000097          auipc ra,0x0
7 58: 000080e7          jalr ra # 54 <.L2+0xe>

```

这里的操作跟上面 `scanf` 函数的行为类似, 都是通过传入参数调用函数, 这里进行的操作是 `printf("%d\n", a)` 加载变量 a 的值并传递给 `printf` 函数, 但是这里也是由于没有链接加载, 所以 `printf` 函数这里应该是空白, 需要后续被添加。

```

1 5c: fe842783          lw  a5,-24(s0)
2 60: 85be              mv  a1,a5
3 62: 000007b7          lui  a5,0x0
4 66: 00078513          mv  a0,a5
5 6a: 00000097          auipc ra,0x0
6 6e: 000080e7          jalr ra # 6a <.L2+0x24>

```

`printf("%d\n", b)` 加载变量 b 的值并传递给 `printf` 函数。

```

1 72: a089              j  b4<.L3>

```

无条件跳转到 $L3$, $L3$ 是 `while` 循环条件判断的入口。

```

1 0000000000000074 <.L4>:
2 74: fe842783          lw  a5,-24(s0)
3 78: fef42023          sw  a5,-32(s0)
4 7c: fe842783          lw  a5,-24(s0)
5 80: 873e              mv  a4,a5
6 82: fec42783          lw  a5,-20(s0)
7 86: 9fb9              addw a5,a5,a4
8 88: fef42423          sw  a5,-24(s0)
9 8c: fe842783          lw  a5,-24(s0)
10 90: 85be              mv  a1,a5

```

```

11 92: 000007b7          lui a5,0x0
12 96: 00078513          mv a0,a5
13 9a: 00000097          auipc ra,0x0
14 9e: 000080e7          jalr ra # 9a <.L4+0x26>
15 a2: fe042783          lw a5,-32(s0)
16 a6: fef42623          sw a5,-20(s0)
17 aa: fe442783          lw a5,-28(s0)
18 ae: 2785             addiw a5,a5,1 # 1 <main+0x1>
19 b0: fef42223          sw a5,-28(s0)

```

L4 是循环体的内容，这里我们就不再赘述，逻辑比较简单。

```

1 00000000000000b4 <.L3>:
2 b4: fdc42783          lw a5,-36(s0)
3 b8: fe442703          lw a4,-28(s0)
4 bc: 2701             sext.w a4,a4
5 be: faf74be3          blt a4,a5,74 <.L4>
6 c2: 4781             li a5,0
7 c4: 853e             mv a0,a5
8 c6: 70a2             ld ra,40(sp)
9 c8: 7402             ld s0,32(sp)
10 ca: 6145             addi sp,sp,48
11 cc: 8082             ret

```

最后这里，我们先分析前 4 行，通过 lw 指令分别将 i 和 n 的值加载到了 a4 和 a5 寄存器中，并通过 blt 指令进行比较，如果 i 的值小于 n 的值那么就跳转到 L4（即循环体）中，否则顺序执行，清空栈空间，恢复栈帧并跳转到返回地址，整个程序结束。

```

1 Disassembly of section .sdata:
2
3 0000000000000000 <myflag>:
4 0: 0001             nop
5 ...

```

.data 段 (数据段) 该段包含全局变量或静态变量的初始化。比如我们定义的 myflag 变量。

```

1 Disassembly of section .rodata:省略
2 Disassembly of section .comment:省略
3 Disassembly of section .riscv.attributes:省略

```

.rodata 段 (只读数据段)

该段包含常量和字符串，可能是程序中的输出文本。

.comment 段

通常存储与编译器、调试器或程序的元数据相关的注释。此段通常不直接影响程序的执行，但可以包含一些调试信息，可能对调试程序的行为、分析程序运行时信息有帮助。另外有关于编译器版本或优化级别的注释，也有助于理解程序的优化程度。

.riscv.attributes 段

这是特定于 RISC-V 架构的段，包含与 RISC-V 相关的架构特性和属性。这部分主要用于描述指

令集架构（ISA）特性，可能包含不同的特权级别、扩展指令集等信息，可能会影响程序的执行。

4.5 链接器做了什么？

4.5.1 功能探究

正如我们上面所说的那样，main 程序调用了 printf 函数，它存在于一个名为 printf.o 的单独的预编译好了的目标文件中，我们的链接器就负责将这个目标文件与我们的 main.o 文件进行链接，从而实现了 printf 的功能，生成了可执行的目标文件，可以加载到内存中被系统执行。

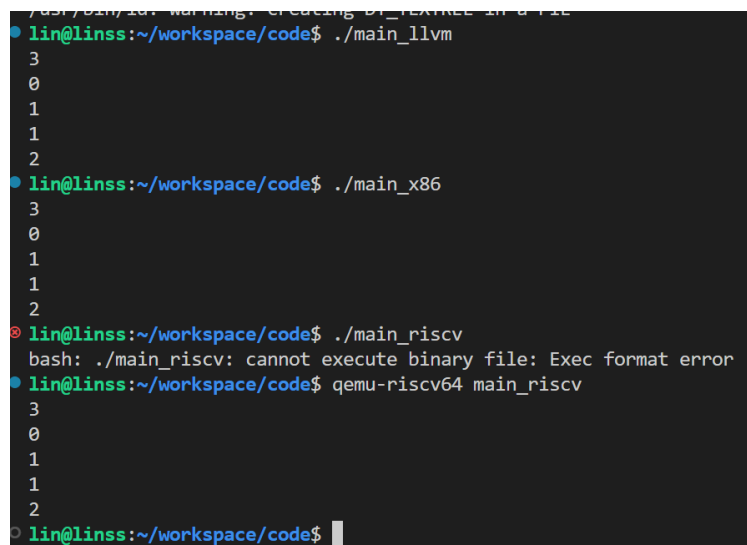
4.5.2 实验验证

链接阶段命令如下：

```
1 gcc main_x86.o -lstlcl++ -o main_x86
2 riscv64-unknown-elf-gcc main_riscv.o -o main_riscv
3 clang main_llvm.o -o main_llvm
```

结果在图5.30可以得到体现，生成了三种可执行程序，都可以成功运行。

值得特别说明的是，由于我们没有 riscv 环境，所以需要使用 qemu 模拟器进行运行，即 qemu-riscv64 main_riscv。



```
lin@linss:~/workspace/code$ ./main_llvm
3
0
1
1
2
2
lin@linss:~/workspace/code$ ./main_x86
3
0
1
1
2
2
lin@linss:~/workspace/code$ ./main_riscv
bash: ./main_riscv: cannot execute binary file: Exec format error
lin@linss:~/workspace/code$ qemu-riscv64 main_riscv
3
0
1
1
2
lin@linss:~/workspace/code$
```

图 4.24: 三种程序运行结果

我们可以执行如下的指令进行反汇编：

```
1 riscv64-unknown-elf-objdump -D main_riscv > main_riscv_r_l.s
```

96	1020e: 3f8000ef	jal 10606 <scanf>
97	10212: fdc42703	lw a4,-36(s0)
98	10216: 6789	lui a5,0x2
99	10218: 71078793	addi a5,a5,1808 # 2710 <exit-0xda10>
100	1021c: 00e7d763	bge a5,a4,1022a <main+0x42>
101	10220: 6789	lui a5,0x2
102	10222: 71078793	addi a5,a5,1808 # 2710 <exit-0xda10>
103	10226: fcf42e23	sw a5,-36(s0)
104	1022a: fec42783	lw a5,-20(s0)
105	1022e: 85be	mv a1,a5
106	10230: 000227b7	lui a5,0x22
107	10234: 5d878513	addi a0,a5,1496 # 225d8 <_clzdi2+0x3c>
108	10238: 3a2000ef	jal 105da <printf>
109	1023c: fe842783	lw a5,-24(s0)

图 4.25: 对 exe 程序反汇编

可以看到我们之前需要调用 scanf 和 printf 函数但却空白的地方，通过链接器的作用，添上了 printf 和 scanf 这两个系统函数的地址，从而实现输入输出的功能。

这也就验证了，通过链接器的链接操作，把我们的.o 程序（可重定位的二进制机器码）中的空白地址替换为了真实的内存地址，实现了链接器的功能，成功地将目标文件和库文件链接到了一起。

至此，我们完成了对一个编译系统的全面探究，任务一产生的所有文件如下所示：

lin@linss:~/workspace/code\$ ls	a-main.c.048t.profile_estimate	a-main.c.334t.debug	main_llvm-01
a-main.c.005t.original	a-main.c.051t.release_ssa	a.out	main_llvm-01.o
a-main.c.006t.gimple	a-main.c.051t.release_ssa.dot	command.txt	main_llvm-02
a-main.c.009t.cmpower	a-main.c.052t.local-fnsummary2	main	main_llvm-02.o
a-main.c.010t.lower	a-main.c.052t.local-fnsummary2.dot	main-00.bc	main_llvm-03
a-main.c.013t.eh	a-main.c.092t.fixup_cfg3	main-00.ll	main_llvm-03.o
a-main.c.015t.cfg	a-main.c.092t.fixup_cfg3.dot	main-01.bc	main_llvm.S
a-main.c.015t.cfg.dot	a-main.c.097t.adjust_alignment	main-01.ll	main_llvm.o
a-main.c.017t.ompexp	a-main.c.097t.adjust_alignment.dot	main-02.bc	main_riscv
a-main.c.017t.ompexp.dot	a-main.c.233t.vecflower	main-02.ll	main_riscv.S
a-main.c.018t.warn-printf	a-main.c.233t.vecflower.dot	main-03.bc	main_riscv.o
a-main.c.018t.warn-printf.dot	a-main.c.234t.cplxlower0	main-03.ll	main_riscv_r.s
a-main.c.022t.fixup_cfg1	a-main.c.234t.cplxlower0.dot	main.bc	main_riscv_r_1.s
a-main.c.022t.fixup_cfg1.dot	a-main.c.236t.switchlower_00	main.c	main_x86
a-main.c.023t.ssa	a-main.c.236t.switchlower_00.dot	main.i	main_x86.S
a-main.c.023t.ssa.dot	a-main.c.243t.isel	main.ll	main_x86.o
a-main.c.027t.fixup_cfg2	a-main.c.243t.isel.dot	main.o	output.png
a-main.c.027t.fixup_cfg2.dot	a-main.c.244t.optimized	main.s	output_optimized.png
a-main.c.028t.local-fnsummary1	a-main.c.244t.optimized.dot	main_llvm	test.c
a-main.c.028t.local-fnsummary1.dot	a-main.c.332t.statistics	main_llvm-00	tokens.txt
a-main.c.029t.einline	a-main.c.332t.statistics	main_llvm-00.o	
a-main.c.029t.einline.dot	a-main.c.333t.earlydebug		
lin@linss:~/workspace/code\$			

图 4.26: 任务一产生的所有文件

5 任务二：掌握 LLVM IR 和汇编编程

5.1 riscv 汇编编程 from 林盛森

5.1.1 代码编写及实验验证

编写如下的 sysY 程序，包含函数调用、数组变量、循环、条件分支、输入输出等，基本覆盖一个程序基础操作。

```

1  const int MAX = 10000;
2
3  // 加法函数
4  int add(int a, int b) {
5      return a + b;
6  }
7
8  void print(int result) {
9      putint(result);
10     putchar(10); // 打印换行符 (ASCII 10)

```

```

11 }
12
13 int main() {
14     int a[2];
15     int i, t, n;
16
17     a[0] = 0;
18     a[1] = 1;
19     i = 1;
20     n = getint();
21     if (n < 0) {
22        _putstr("ERROR!");
23         putchar(10);
24         return 0;
25     }
26     if (n == 0) {
27         print(a[0]);
28         return 0;
29     }
30     if (n > MAX) {
31         n = MAX;
32     }
33     print(a[0]);
34     print(a[1]);
35     while (i < n) {
36         t = a[1];
37         a[1] = add(a[0], a[1]);
38         print(a[1]);
39         a[0] = t;
40         i = add(i, 1);
41     }
42     return 0;
43 }

```

对照 sysY 程序，我们编写如下的 riscv64 汇编代码：

```

1 .section .data
2 fmt_in:      .string "%d"           # scanf 格式字符串
3 fmt_out:     .string "%d\n"         # printf 格式字符串（用于 print 函数）
4 error_msg:   .string "ERROR!\n"     # 错误提示字符串

```

首先，在.data 段中，我们定义了三个字符串：

- `fmt_in`: `"%d"`，给 `scanf` 用，用于读整数。
- `fmt_out`: `"%d\n"`，给 `printf` 用，打印整数 + 换行。
- `error_msg`: `"ERROR!\n"`，当输入 `n < 0` 时打印。

这些是静态字符串，以 `\0` 结尾存在内存中，程序用 `la` 指令取地址。

```

1 .section .text
2 .globl add
3 add:
4     add a0, a0, a1    # a0 = a0 + a1
5     ret

```

接着是.text 段，首先这里实现了我们在定义的函数 add(a, b)，用于计算 $a + b$ 的值。直接用 add 指令计算，在这里就是 $a0=a0+a1$ ，加法结果保存在 a0 中，并可通过 ret 返回到调用处。

```

1 .globl print
2 print:
3     addi sp, sp, -16 # 分配 16 字节栈空间（保持对齐）
4     sd ra, 8(sp)    # 保存返回地址
5     mv a1, a0        # 将参数 result 移到 a1（printf 的第二个参数）
6     la a0, fmt_out   # 加载格式字符串地址
7     call printf      # 调用 printf
8     ld ra, 8(sp)     # 恢复返回地址
9     addi sp, sp, 16  # 释放栈空间
10    ret

```

这里实现了我们定义的 print(result) 函数，用于打印整数并换行。具体方式就是我们调用 printf 函数，这里首先要开辟 16 字节的栈空间为这个 print 子函数中的局部变量等腾出位置，然后保存返回地址、将参数 result 作为第二个参数传给 printf 函数，将输出要用的格式字符串的地址作为第一个参数传入，可以看到这里我们使用了 la 指令，将 fmt_out 这个格式化字符串的地址放到了 a0 中。通过调用 printf 函数，即可打印出结果并进行换行。最后还会恢复返回地址并释放栈空间。

```

1 .globl putstr
2 putstr:
3     addi sp, sp, -16 # 分配 16 字节栈空间
4     sd ra, 8(sp)    # 保存返回地址
5     call printf      # 直接调用 printf，a0 包含字符串地址
6     ld ra, 8(sp)     # 恢复返回地址
7     addi sp, sp, 16  # 释放栈空间
8     ret

```

这里跟上面 print 函数的情况类似，分配栈空间、保存返回地址，也就是保存旧栈的信息，然后这里直接调用 printf 函数，因为在调用 putstr 时，已经把需要输出的内容放到了 a0 中。调用完 printf 之后依旧恢复返回地址并释放栈空间。

```

1 .globl main
2 main:
3     # 强制栈对齐到 16 字节
4     andi sp, sp, -16 # 将 sp 对齐到 16 字节边界
5     # 分配 64 字节栈空间（16 字节对齐）
6     addi sp, sp, -64
7     sd ra, 56(sp)    # 保存返回地址
8     sd s0, 48(sp)    # 保存 n
9     sd s1, 40(sp)    # 保存 a[0]

```

```

10    sd s2, 32(sp)    # 保存 a[1]
11    sd s3, 24(sp)    # 保存 i
12    sd s4, 16(sp)    # 保存 t
13
14    # 初始化
15    li s1, 0          # a[0] = 0
16    li s2, 1          # a[1] = 1
17    li s3, 1          # i = 1

```

这里进入 main 函数，首先强制栈对齐到 16 字节，要不可能违背 riscv 架构的特性，可能会出错，并分配了 64 字节的栈空间，另外我们要注意，一定要分配足够大的栈空间，并且变量之间在栈上的位置不要冲突，否则可能会出现段错误。接着将一些寄存器的值保存到栈中，便于程序结束时恢复，并进行了有关变量的初始化。

```

1    # 读取 n
2    la a0, fmt_in
3    addi a1, sp, 8    # 为 scanf 分配 4 字节空间 (sp+8)
4    call scanf
5    lw s0, 8(sp)      # 加载 n 到 s0
6    sext.w s0, s0     # 符号扩展 32 位到 64 位

```

然后这里是调用 scanf 函数来输入 n，fmt_in 作为第一个参数传入，并为 scanf 分配 4 字节的临时空间，作为第二个参数传入，n 的值保存到 s0 寄存器中，并用符号拓展将 n 的值从 32 位扩展到 64 位，这样可以避免后续 s0 与其他 64 位寄存器比较时可能会发生的问题。

```

1    # 处理 n < 0
2    bgez s0, check_zero # 如果 n >= 0，跳转到检查 n == 0
3    la a0, error_msg    # 加载 "ERROR!\n" 地址
4    call putstr         # 打印错误信息
5    j end              # 直接退出

```

然后这里处理 n<0 的情况，如果 n >= 0，那么就跳转到 check_zero 标签处检查 n == 0，否则 n<0，就通过我们上面定义的 putstr 方式打印信息，直接退出程序。

```

1 check_zero:
2    # 处理 n == 0
3    bnez s0, check_max  # 如果 n != 0，跳转到检查最大值
4    mv a0, s1           # a[0]
5    call print         # 打印 a[0]
6    j end              # 退出

```

那如果 n>=0，就会跳转到这里，继续判断 n 是否等于 0，若等于 0，就只输出 a[0]，接着跳转到 end 标签处退出程序；否则就顺序执行。

```

1 check_max:
2    # 限制 n 最大值
3    li t0, 10000
4    ble s0, t0, fib_start
5    li s0, 10000

```


这里来检查 n 是否超过我们限定的最大值 10000，若超过就顺序执行 `li s0, 10000`，将 n 赋值为 10000；否则就跳转到 `fib_start` 标签处，开始进行斐波那契数列的计算与输出。

```

1 fib_start:
2     # 打印 a[0]
3     mv a0, s1
4     call print
5
6     # 打印 a[1]
7     mv a0, s2
8     call print

```

程序执行到这里时，已经满足 $n > 0$ 的条件，即 $n \geq 1$ ，所以 `a[0], a[1]` 全部输出。

```

1     # 循环
2 loop:
3     bge s3, s0, end    # 如果 i >= n, 结束
4     mv s4, s2          # t = a[1]
5     mv a0, s1          # a[0]
6     mv a1, s2          # a[1]
7     call add           # a[1] = add(a[0], a[1])
8     mv s2, a0          # 更新 a[1]
9     mv a0, s2          # 打印 a[1]
10    call print
11    mv s1, s4           # a[0] = t
12    mv a0, s3           # i
13    li a1, 1
14    call add            # i = add(i, 1)
15    mv s3, a0           # 更新 i
16    j loop

```

接着顺序执行进入到 `loop` 循环中，这里的逻辑比较简单，通过 `bge` 模拟 `while (i < n)` 中的循环条件，并在每次循环中调用两次 `add` 和一次 `print`，不再赘述。对于 `mv a0, s3, li a1, call add, mv s3, a0` 这几条，可以不用通过 `add` 来调用，直接使用 `addi` 立即数加法可以更快一些。

```

1 end:
2     # 恢复寄存器并释放栈
3     ld s4, 16(sp)
4     ld s3, 24(sp)
5     ld s2, 32(sp)
6     ld s1, 40(sp)
7     ld s0, 48(sp)
8     ld ra, 56(sp)
9     addi sp, sp, 64
10    li a0, 0
11    ret

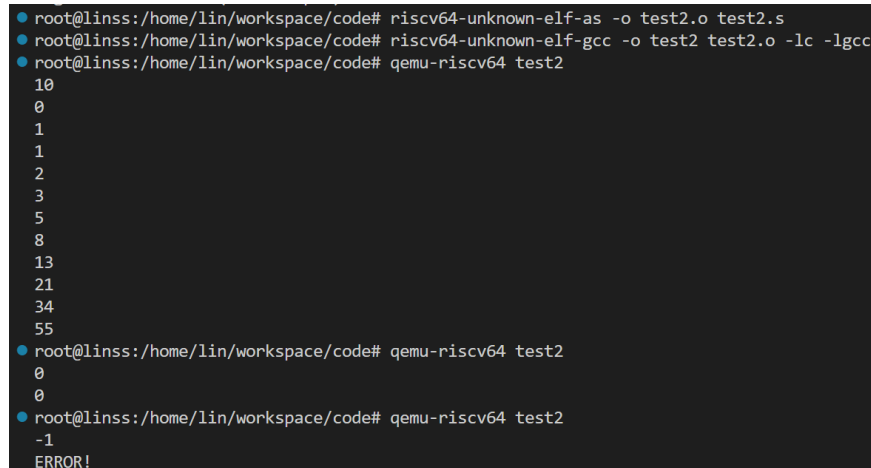
```

最后 `end` 是程序结束的标志，通过 `ld` 恢复寄存器 `s4, s3, s2, s1, s0, ra`，接着释放我们分配的 64 字节的栈空间。`li a0, 0` 设置返回值为 0，通过 `ret` 实现 `return 0`。

可以通过以下命令运行：链接阶段指令中的 `-lc` 指链接 C 标准库，`-lgcc` 指链接 GCC 编译器辅助库，我们程序中的 `printf`、`scanf` 函数需要在此进行加载，从链接库里找到 `printf` 和 `scanf` 函数真实的内存地址。

```
1 riscv64-unknown-elf-as -o test2.o test2.s
2 riscv64-unknown-elf-gcc -o test2 test2.o -lc -lgcc
3 qemu-riscv64 test2
```

运行结果如下，可以看到程序正常运行，结果正确，并且对于边界情况，程序的正确性均得到了验证。



```
root@linss:/home/lin/workspace/code# riscv64-unknown-elf-as -o test2.o test2.s
root@linss:/home/lin/workspace/code# riscv64-unknown-elf-gcc -o test2 test2.o -lc -lgcc
root@linss:/home/lin/workspace/code# qemu-riscv64 test2
10
0
1
1
2
3
5
8
13
21
34
55
root@linss:/home/lin/workspace/code# qemu-riscv64 test2
0
0
-1
ERROR!
```

图 5.27: riscv 程序运行验证

但是当我查看任务要求时说，要链接 `sysY` 库，我上面使用到的 `printf` 和 `scanf` 是属于 C 库的，所以我之后又做了一点修改，改成了基于 `sysY` 库版本的输入及输出，由于篇幅限制，这里只粘贴出修改过的部分：

首先运行如下命令，生成 `libsysy_riscv.a` 这个链接库。

```
1 riscv64-unknown-elf-gcc -c -o sylib.o sylib.c -w -static #生成sysY链接库
2 riscv64-unknown-elf-ar rcs libsysy_riscv.a sylib.o
```

接着通过运行如下图片中的命令，可以看到 `sysY` 库支持的函数，我们选取 `putint`、`putch`、`getint` 作为我们的输入输出函数。

```

root@linss:/home/lin/workspace/code# riscv64-unknown-elf-nm /home/lin/workspace/lib/libsysy_riscv.a
0000000000004020 B _sysy_s
0000000000000000 B _sysy_start
0000000000000694 T _sysy_starttime
00000000000006d8 T _sysy_stoptime
0000000000005020 B _sysy_us
0000000000000400 T after_main
0000000000000344 T before_main
U fprintf
0000000000000088 T getarray
000000000000002c T getch
00000000000000f4 T getfarray
000000000000005a T getfloat
0000000000000000 T getint
U gettimeofday
U printf
00000000000001b4 T putarray
000000000000018e T putch
U putchar
00000000000002ea T putf
0000000000000264 T putfarray
0000000000000232 T putfloat
0000000000000160 T putint
U scanf
U vfprintf

```

图 5.28: sysY 库支持的函数

```

1 .section .data
2
3 error_msg: .string "ERROR!\n"      # 错误提示字符串

```

这里删除了两个格式化字符串，因为 sysY 库的 putint、putch、getint 函数不需要传入格式化字符串的参数。

```

1 .globl print
2 print:
3     addi sp, sp, -16 # 分配 16 字节栈空间（保持对齐）
4     sd ra, 8(sp)    # 保存返回地址
5     call putint     # 调用 sysY 的 putint 函数
6     ld ra, 8(sp)    # 恢复返回地址
7     addi sp, sp, 16 # 释放栈空间
8     ret

```

由于不必传入输入的格式化字符串，所以这里进行删除，而唯一的参数通过 a0 寄存器传递。

```

1 .globl putstr
2 putstr:
3     addi sp, sp, -16 # 分配 16 字节栈空间
4     sd ra, 8(sp)    # 保存返回地址
5     # sysY 库没有直接的字符串输出函数，使用 putch 循环输出
6     mv t0, a0       # 保存字符串地址
7 putstr_loop:
8     lb t1, 0(t0)    # 加载一个字符
9     beqz t1, putstr_end # 如果遇到 0，结束
10    mv a0, t1        # 将字符作为参数
11    call putch       # 输出字符
12    addi t0, t0, 1   # 移动到下一个字符
13    j putstr_loop
14 putstr_end:
15    ld ra, 8(sp)    # 恢复返回地址

```

```

16     addi sp, sp, 16    # 释放栈空间
17     ret

```

由于 `putch` 是一次输出一个字符，所以我们这里要加入循环的逻辑输出所有字符，即一整个字符串，遇到 `\0` 结束。主要的操作就是先初始化栈空间，保存父过程的相应信息，通过循环逐字符进行输出，恢复栈帧。可以看到我们在 `putstr` 中保存了 `ra`，当输出结束时，即在 `putstr_end` 中，通过 `ret` 即可返回到 `putstr` 函数调用处。这里的具体循环逻辑比较简单，不再赘述。

```

1     # 读取 n
2     call getint        # 调用 sysY 的 getint 函数
3     mv s0, a0          # 将输入的 n 保存到 s0

```

对于输入，直接调用 `getint`，只需传入保存输入结果的地址即可。

```

1 fib_start:
2     # 打印 a[0]
3     mv a0, s1
4     call print
5
6     # 打印换行符
7     li a0, 10          # '\n' 的 ASCII 码
8     call putch

```

输出结果也是，只需传入需要输出的信息，但为了与我们设计的 `sysY` 程序逻辑一致，需要手动输出换行符。对于循环中的输出也是一样，我们不再赘述。

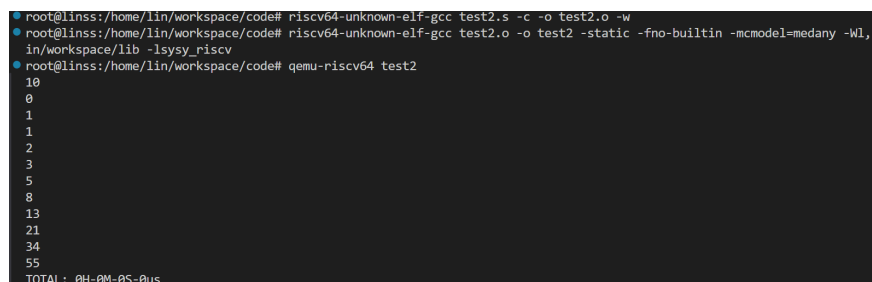
主要修改就是这些。然后我们运行如下的命令，链接我们生成好的 `libsysy_riscv.a` 这个链接库。

```

1 riscv64-unknown-elf-gcc test2.s -c -o test2.o -w
2 riscv64-unknown-elf-gcc test2.o -o test2 -static -fno-builtin -mcmodel=medany
   -Wl,--no-relax -Ttext=0x10000000 -L /home/lin/workspace/lib -lsysy_riscv

```

运行结果如下，我们完成了 `riscv` 汇编语言编译、链接到 `sysY` 库并成功运行的完整流程。



```

root@linss:/home/lin/workspace/code# riscv64-unknown-elf-gcc test2.s -c -o test2.o -w
root@linss:/home/lin/workspace/code# riscv64-unknown-elf-gcc test2.o -o test2 -static -fno-builtin -mcmodel=medany -Wl,--no-relax -Ttext=0x10000000 -L /home/lin/workspace/lib -lsysy_riscv
root@linss:/home/lin/workspace/code# qemu-riscv64 test2
10
0
1
1
2
3
5
8
13
21
34
55
TOTAL: 0H-0M-0S-0us

```

图 5.29: 链接 `sysY` 库结果验证

5.1.2 RISC-V 64 位汇编语言特性分析

为了进一步探究 RISC-V 64 位汇编语言，结合我们的程序，并查阅相关资料，我总结出以下对该架构的描述：

精简规整的指令集 RISC-V 指令集采用 **32 位固定长度编码，格式高度统一**。这一特性在我们的程序中可以得到直接的体现：无论是算术运算指令 `add a0, a0, a1`，还是立即数指令 `addi sp, sp, -64`，其指令格式都保持规整，类似于三地址码，这简化了 CPU 的译码过程，是 RISC-V 追求硬件实现简单性的直接证明。

严格的加载-存储架构 RISC-V 采用严格的加载-存储架构，规定**运算指令仅能操作寄存器**。在我们的程序中，这一原则表现为：所有对变量 `n` 和寄存器 `ra`，以及寄存器 `s0-s4` 的访问都必须通过 `lw` 或 `sd` 指令来显式完成，而 `add` 等计算指令则完全在寄存器间进行，清晰地分离了计算与访存操作。

严格的寄存器使用规范 RISC-V 制定了明确的关于寄存器的应用程序二进制接口规范，比如参数传递经常用到 `a0` 和 `a1` 这两个寄存器，保存寄存器 `s0, s1...` 用来保存关键变量等。在我们的程序中，比如在调用 `scanf` 时使用 `a0, a1` 分别作为第一个和第二个参数传入、`add` 函数使用 `a0` 作为返回值，还有用 `s0-s4` 寄存器来保存 `a[0]`、`a[1]`、`i`、`t`、`n` 这些关键变量，都遵循了这一规范，确保了函数调用过程中寄存器状态的可知性与稳定性，从而避免出错。

系统化的栈管理与内存对齐 RISC-V 的应用程序二进制接口**强制要求栈指针必须 16 字节对齐**。程序在 `main` 函数入口处使用 `andi sp, sp, -16` 强制对齐，并通过 `addi sp, sp, -64` 分配栈空间，系统地保存 `ra` 及 `s0-s4` 寄存器，这体现了其对栈管理和内存对齐规则的严格遵守。我一开始写这个程序的适合，并没有遵守这一点，导致段错误。

清晰的函数调用与控制流 RISC-V 使用 **`call/ret` 伪指令**实现标准函数调用与返回，并提供了丰富的条件分支指令。在我们的程序中，函数调用通过 `call` 指令完成，循环与条件判断则通过 `bgez`、`bnez` 以及 `j` 等指令实现，展现了其控制流指令集的清晰与高效。

5.2 LLVM IR 编程 from 牛帅

5.2.1 LLVM IR 概述

LLVM IR 是相对于 CPU 指令集更为高级、但相对于源程序更为低级的代码中间表示的一种语言，是由**代码生成器**自顶向下遍历**语法树**形成的，具有**类型化、可扩展性和强表现力**的特点。LLVM IR 本身更贴近汇编语言，指令集相对底层，能灵活地进行低级操作。LLVM IR 作为现代编译器技术的核心，为多种编程语言提供了强大的编译基础设施，是理解现代编译器工作原理的关键。

为了了解 LLVM IR 中间语言，我们设计 SysY 示例程序 `example.sy` 源代码如下：

```
1 int global_var = 10;
2 int global_array[5] = {1, 2, 3, 4, 5};
3 int add(int a, int b);
4 void print_result(int result);
5
6 int main() {
7     int local_var = 5;
8     int i = 0;
9     int sum = 0;
10    int result1, result2, result3, result4;
11    int func_result;
12    int local_array[3];
```

```
13     int final_result;
14
15     result1 = global_var + local_var;
16     result2 = global_var - local_var;
17     result3 = global_var * local_var;
18     result4 = global_var / local_var;
19
20     func_result = add(result1, result2);
21
22
23     if (func_result > 20) {
24         sum = sum + 10;
25     } else {
26         sum = sum + 5;
27     }
28
29
30     for (i = 0; i < 5; i = i + 1) {
31         sum = sum + global_array[i];
32     }
33
34
35     i = 0;
36     while (i < 3) {
37         sum = sum + i;
38         i = i + 1;
39     }
40
41
42     local_array[0] = sum;
43     local_array[1] = sum * 2;
44     local_array[2] = sum * 3;
45
46
47     final_result = local_array[0] + local_array[1] + local_array[2];
48
49
50     print_result(final_result);
51
52     return 0;
53 }
54
55
56 int add(int a, int b) {
57     return a + b;
58 }
59
60
61 void print_result(int result) {
```

```

62     putint(result);
63     putchar(10);
64 }

```

这个 sy 程序是一个综合示例，首先设置一个**整型全局变量**与一个**整型全局数组**，接着声明两个**函数**。进入主函数后，对**局部变量初始化**，接着进行四种基本**算术运算**，然后**调用函数**并根据函数调用结果进行**条件判断**，再进行 **for 循环**与 **while 循环**，最后进行**数组操作**并**调用输出函数**打印最终结果。在主函数外进行**加法函数**与**打印函数**的实现。

5.2.2 LLVM IR 编写

接下来我将根据 y 源代码编写 LLVM IR 中间代码：

```

1 @global_var = dso_local global i32 10, align 4
2 @global_array = dso_local global [5 x i32] [i32 1, i32 2, i32 3, i32 4, i32 5], align
  16
3 @.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

```

- @global_var 定义全局整型变量，值为 10
- @global_array 定义全局数组，包含 5 个整数 [1,2,3,4,5]
- @.str 用于格式化字符串常量，用于 printf 输出

接下来进入 main 函数：

```

1 %1 = alloca i32, align 4 ; 函数返回值
2 %2 = alloca i32, align 4 ; local_var
3 %3 = alloca i32, align 4 ; i
4 %4 = alloca i32, align 4 ; sum
5 ...
6 按变量出现顺序以此类推

```

为主函数中的局部变量使用 alloca 指令在栈上分配空间。

```

1 store i32 0, ptr %1, align 4 ; 返回值=0
2 store i32 5, ptr %2, align 4 ; local_var = 5
3 store i32 0, ptr %3, align 4 ; i = 0
4 store i32 0, ptr %4, align 4 ; sum = 0

```

利用 store 命令为部分局部变量进行变量初始化。

```

1 #global_var + local_var
2 %12 = load i32, ptr @global_var, align 4
3 %13 = load i32, ptr %2, align 4
4 %14 = add nsw i32 %12, %13 ; 检查符号溢出
5 store i32 %14, ptr %5, align 4
6 ...
7 算术运算以此类推

```

为 result1 4 进行基本的四则运算。

```
1 %24 = load i32, ptr %5, align 4 ;
2 %25 = load i32, ptr %6, align 4 ;
3 %26 = call i32 @add(i32 noundef %24, i32 noundef %25)
4 store i32 %26, ptr %9, align 4 ;
```

将 result1 与 result2 传参调用 add 函数。

```
1 %27 = load i32, ptr %9, align 4
2 %28 = icmp sgt i32 %27, 20 ;
3 br i1 %28, label %29, label %32 ;
4
5 29: ; preds = %0
6   %30 = load i32, ptr %4, align 4
7   %31 = add nsw i32 %30, 10
8   store i32 %31, ptr %4, align 4
9   br label %35
10
11 32: ; preds = %0
12   %33 = load i32, ptr %4, align 4
13   %34 = add nsw i32 %33, 5
14   store i32 %34, ptr %4, align 4
15   br label %35
```

如果 func_result > 20, 跳转到标签%29, 否则跳转到标签%32, 再分别给出%29 与%32 的内容。

```
1 ; 循环初始化
2 store i32 0, ptr %3, align 4
3 br label %36
4
5 ; 循环条件检查
6 36:
7   %37 = load i32, ptr %3, align 4
8   %38 = icmp slt i32 %37, 5 ;
9   br i1 %38, label %39, label %49 ;
10
11 ; 循环体
12 39:
13   %40 = load i32, ptr %4, align 4
14   %41 = load i32, ptr %3, align 4
15   %42 = sext i32 %41 to i64 ;
16   %43 = getelementptr inbounds [5 x i32], ptr @global_array, i64 0, i64 %42
17   %44 = load i32, ptr %43, align 4
18
19   %45 = add nsw i32 %40, %44
20   store i32 %45, ptr %4, align 4
21   br label %46
22
23 ; 循环递增
```



```

24 46:
25   %47 = load i32, ptr %3, align 4
26   %48 = add nsw i32 %47, 1 ; i = i + 1
27   store i32 %48, ptr %3, align 4
28   br label %36, !llvm.loop !6 ;

```

进入 for 循环后：

- 进行循环变量 i 的初始化
- 检查循环条件 i<5
- 进入循环体，访问数组 global_array[i]，将其累加至 sum
- 进行循环变量 i 的递增，然后跳回条件检查

```

1 ; 循环初始化
2 store i32 0, ptr %3, align 4
3 br label %50
4
5 ; 循环条件检查
6 50:
7   %51 = load i32, ptr %3, align 4
8   %52 = icmp slt i32 %51, 3 ; i < 3
9   br i1 %52, label %53, label %59
10
11 ; 循环体
12 53:
13   ; sum = sum + i
14   %56 = add nsw i32 %54, %55
15   store i32 %56, ptr %4, align 4
16   ; i = i + 1
17   %58 = add nsw i32 %57, 1
18   store i32 %58, ptr %3, align 4
19   br label %50, !llvm.loop !8

```

进入 while 循环后：

- 进行循环变量 i 的初始化
- 检查循环条件 i<3
- 进入循环体，将 i 其累加至 sum，然后进行循环变量 i 的递增，然后跳回条件检查

```

1
2   %60 = load i32, ptr %4, align 4
3   %61 = getelementptr inbounds [3 x i32], ptr %10, i64 0, i64 0
4   store i32 %60, ptr %61, align 4
5
6

```

```

7  %62 = load i32, ptr %4, align 4
8  %63 = mul nsw i32 %62, 2
9  %64 = getelementptr inbounds [3 x i32], ptr %10, i64 0, i64 1
10 store i32 %63, ptr %64, align 4
11
12
13 %65 = load i32, ptr %4, align 4
14 %66 = mul nsw i32 %65, 3
15 %67 = getelementptr inbounds [3 x i32], ptr %10, i64 0, i64 2
16 store i32 %66, ptr %67, align 4
17
18
19 %68 = getelementptr inbounds [3 x i32], ptr %10, i64 0, i64 0
20 %69 = load i32, ptr %68, align 4
21 %70 = getelementptr inbounds [3 x i32], ptr %10, i64 0, i64 1
22 %71 = load i32, ptr %70, align 4
23 %72 = add nsw i32 %69, %71
24 %73 = getelementptr inbounds [3 x i32], ptr %10, i64 0, i64 2
25 %74 = load i32, ptr %73, align 4
26 %75 = add nsw i32 %72, %74
27 store i32 %75, ptr %11, align 4

```

为数组 `local_array` 进行数组初始化, 再进行数组元素求和, 结果存入 `final_result`。

```

1  %76 = load i32, ptr %11, align 4
2  call void @print_result(i32 noundef %76)
3  ret i32 0

```

最后调用 `print_result` 函数并返回结果, `main` 函数编写完毕。

```

1  define dso_local i32 @add(i32 noundef %0, i32 noundef %1) #0 {
2    %3 = alloca i32, align 4 ;
3    %4 = alloca i32, align 4 ;
4    store i32 %0, ptr %3, align 4
5    store i32 %1, ptr %4, align 4
6    %5 = load i32, ptr %3, align 4
7    %6 = load i32, ptr %4, align 4
8    %7 = add nsw i32 %5, %6 ;
9    ret i32 %7
10 }

```

实现 `add` 函数, 对传入的参数 `a` 与 `b` 进行相加操作。

```

1  define dso_local void @print_result(i32 noundef %0) #0 {
2    %2 = alloca i32, align 4
3    store i32 %0, ptr %2, align 4
4    %3 = load i32, ptr %2, align 4
5    %4 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %3)
6    ret void
7  }

```

实现 `print_result` 函数，实现对传入的参数 `result` 的打印操作。

综上所述，我们将分步编写的 LLVM IR 中间代码进行拼接，即可形成最终代码。

5.2.3 LLVM IR 改进

上述 LLVM IR 代码使我们根据原本的 `sy` 程序所反推出来的，实际上该代码也存在着如下许多可以优化改进的地方：

1. 冗余加载

我们可以看到代码中存在大量不必要的 `load/store` 操作，如在为数组赋值时多次加载 `sum` 的值（即`%4`），我们可以修改为如下代码，只加载一次，然后重复使用该值：

```
1 %60 = load i32, ptr %4, align 4    ; 只加载一次
2 %61 = getelementptr inbounds [3 x i32], ptr %10, i64 0, i64 0
3 store i32 %60, ptr %61, align 4
4 %63 = mul nsw i32 %60, 2           ; 使用%60而不是重新加载
5 %64 = getelementptr inbounds [3 x i32], ptr %10, i64 0, i64 1
6 store i32 %63, ptr %64, align 4
7 %66 = mul nsw i32 %60, 3           ; 使用%60
8 %67 = getelementptr inbounds [3 x i32], ptr %10, i64 0, i64 2
9 store i32 %66, ptr %67, align 4
```

2. 乘法操作优化：

因为我们的乘法操作属于两个已经赋值的变量相乘，所以无需保证结果不会发生有符号整数溢出，因此我们可以使用能耗更低的二进制左移方法进行编译计算：

```
1 ; 原始代码
2 %tmp = mul nsw i32 %x, 8
3
4 ; 优化后
5 %tmp = shl i32 %x, 3
```

3. 循环优化：我们可以将循环内不变的计算移动到循环外部，显著提升程序性能。例如我们可以将在 `for` 循环中的数组地址外提出来：

```
1 br label %36
2
3 36:
4   %37 = load i32, ptr %3, align 4
5   %38 = icmp slt i32 %37, 5
6   br i1 %38, label %39, label %49
7
8 39:
9   ; 数组地址计算外提
10  %array_base = getelementptr inbounds [5 x i32], ptr @global_array, i64 0, i64 0
11
```

```

12 ; 剩余操作
13 %40 = load i32, ptr %4, align 4
14 %41 = load i32, ptr %3, align 4
15 %42 = sext i32 %41 to i64
16 %43 = getelementptr i32, ptr %array_base, i64 %42
17 %44 = load i32, ptr %43, align 4
18 %45 = add nsw i32 %40, %44
19 store i32 %45, ptr %4, align 4
20 br label %46

```

5.2.4 运行结果

综上所述，我们成功改进了 LLVM IR 代码，下面我们可以运行该代码来检测结果。我们依次通过运行：

1. `llc example.ll -o example.S`
2. `clang -c example.S -o example.o`
3. `clang -no-pie example.o -o example`
4. `./example`

最终代码运行结果如下图所示：

```

root@LAPTOP-1TQ9KDNS:~/Flex-Bison-example/example# clang -no-pie example.o -o example
root@LAPTOP-1TQ9KDNS:~/Flex-Bison-example/example# ./example
138

```

图 5.30: 运行结果

到此，我们成功运行了我们编写的 LLVM IR 代码，验证了其正确性。

6 心得体会

通过这次实验，我深刻理解了编译器是如何将一个源程序一步一步地生成至目标可执行程序的过程，了解了 riscv 汇编程序的语言特性，并在此基础上实现了我自己的 riscv 汇编程序的编写。完成本次实验，我大概用了三天的时间。

第一天我**完成了环境的配置**，并参阅飞书群中的教程，收集所有可能需要使用到的命令，尝试把整个流程走了一遍，在其中也对于命令行的运用更加熟练，了解到了各条命令中某个标志的作用，比如说在进行词法分析时，使用 `clang -E -Xclang -dump-tokens main.c 2> tokens.txt` 命令将词法分析的结果重定向到 `tokens.txts` 时，我发现一开始其中没有结果，需要加 2，这是因为 `-dump-tokens` 属于标准错误，不是标准输出。

第二天我**基本对任务一进行了具体的实现并完成了任务一的编写**，这是我第一次对于编译器有了这么彻底的认知，使得我更明确了这学期《编译系统原理》这门课的课程目标，对于实现一个自己的编译器需要进行哪些环节或工具的实现有了更为清晰的认知。

第三天我**完成了任务二的实现与实验报告的编写**，在此过程中，我了解了 riscv 这种架构的语言特性以及其具有的优势，我还了解到如何通过命令查看 `sysY` 库具有的系统函数，并在此基础上，设计

了 sysY 源程序，根据我的 sysY 源程序，完成了 riscv 汇编语言的编写，并成功汇编并链接到 C 库和 sysY 库，正确性都得到了验证。

并且在此过程中，我和我的队友积极交流，互相分享，互相帮助。

7 代码链接

[点击访问 GitHub 主页](#)