



南开大学  
Nankai University

南开大学

计算机学院和网络空间安全学院

编译系统原理实验报告

---

了解你的编译器 & LLVM IR 编程 & 汇编编程

---

陆皓喆

年级：2022 级

学号：2211044

专业：信息安全

指导教师：王刚

2024 年 9 月 23 日

## 摘要

本次实验由陆皓喆和郝志成共同完成。首先，我们介绍了实验目的以及问题的描述。接着说明了本组的分工情况，配置并说明了本人的实验环境。任务一主要完成了了解编译器的任务，主要包括预处理器、编译器、汇编器、链接器、最后执行等过程；任务二主要完成了熟悉 LLVM IR 中间语言，并且编写 LLVM IR 程序小例子，用 LLVM/Clang 编译成目标程序，完成了执行验证；任务三主要完成了斐波那契数列和浮点数的汇编编程的编写，包括自己的编写思路、程序逐行分析、gcc 验证、思考等部分内容。最后是总结部分，总结了本次实验的心得体会。

**关键字：**词法分析，语法分析，语义分析，代码优化，汇编与链接，LLVM IR 编程，汇编编程

# 目录

<b>一、 实验介绍</b>	<b>1</b>
(一) 实验目的	1
(二) 问题描述	1
(三) 分工	1
(四) 实验环境	1
<b>二、 任务一：了解你的编译器</b>	<b>3</b>
(一) 完整编译过程	3
(二) 预处理器	4
1. 预处理阶段功能	4
2. 实验验证与结果分析	4
(三) 编译器	5
1. 词法分析	6
2. 语法分析	9
3. 语义分析	13
4. 中间代码生成	13
5. 代码优化	16
6. 代码生成	20
(四) 汇编器	20
(五) 链接器	22
(六) 执行程序	23
<b>三、 任务二：LLVM IR 编程</b>	<b>24</b>
(一) 任务内容	24
(二) 斐波那契数列程序的编写与改进	24
1. 编写	25
2. 数组和浮点数改进	29
3. 指针优化	33
4. 验证	35
<b>四、 任务三：汇编编程</b>	<b>36</b>
(一) 程序 1：斐波那契数列	36
1. SysY 源程序	36
2. 汇编程序编写思路	37
3. 汇编程序分析	37
4. gcc 验证	41
5. 一些思考	42
(二) 程序 2：浮点数	42
1. SysY 源程序	42
2. 汇编程序编写思路	43
3. 汇编程序分析	43
4. gcc 验证	46
5. 一些思考	46

**五、 总结**

**47**

## 一、 实验介绍

### (一) 实验目的

本次实验是本学期编译系统原理实验课程的**预备实验**，主要目的是让我们熟悉自己的编译器，我认为本次实验有以下目的：

- **理解编译过程：**通过本实验，深入理解从源代码到可执行文件的整个编译过程，包括预处理、编译、汇编和链接等各个阶段。
- **掌握编译器工作机制：**学习如何使用编译器的不同阶段选项来观察和分析程序在每个阶段的输出，从而理解编译器是如何工作的。
- **熟悉 LLVM IR 编程：**通过编写 LLVM IR 程序，熟悉 LLVM IR 这种中间语言，了解其在编译过程中的作用和重要性。
- **熟练编写汇编语言：**通过编写 RISC-V 汇编程序，掌握汇编语言的基本语法和编程技巧，理解底层硬件操作的基本原理。
- **探索编译器优化：**通过实验，探索编译器的优化技术，了解如何通过优化参数来提高程序的性能和效率。

### (二) 问题描述

**任务一**，以 GCC 为研究对象，深入地探究语言处理系统的完整工作过程，其中包括：完整的编译过程、预处理器的任务、编译器的任务、汇编器的任务、链接器的任务、如何执行程序。

**任务二**，通过编写 LLVM IR 程序，熟悉 LLVM IR 中间语言。

**任务三**，通过编写汇编程序，熟悉 RISC-V 汇编语言。

### (三) 分工

本组选题为 C-RISCV 框架，小组成员为陆皓喆 (2211044) 和郝志成 (2212514)。

两人分别独立完成**任务一：了解你的编译器**中的编译过程的复现和该部分实验报告的撰写；郝志成 (2212514) 负责完成**任务二：LLVM IR 编程**中斐波拉契数列程序和数组程序 (扩展) 的代码和报告的编写；

陆皓喆 (2211044) 负责完成**任务三：汇编编程**中斐波那契数列程序和浮点数字程序 (扩展) 的代码和报告的编写；

最后的总结部分由两人共同合作完成。

**本实验的实验代码文件均已存放到 github 仓库，您可以通过[此链接](#)来查阅我的代码文件。**

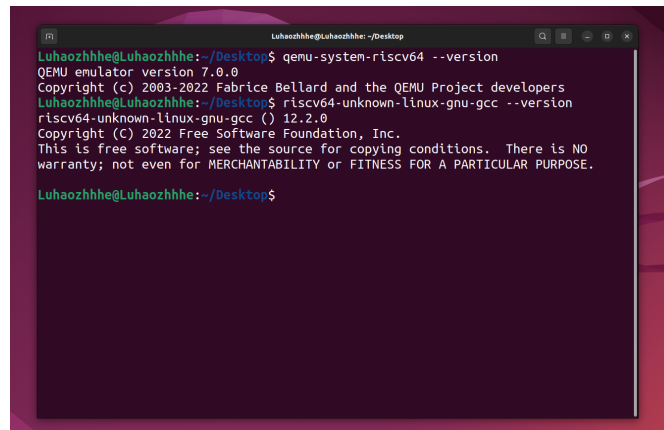
### (四) 实验环境

按照实验指导书的说明，本人选用 ubuntu-22.04.4 版本的虚拟机来进行实验平台环境的配置。实验平台的具体参数如下所示：

设备名称	Luhaozhhe
系统名称	Ubuntu 22.04.4 LTS
操作系统类型	Linux 64 位
虚拟机	VMware Workstation
虚拟机版本	WORKSTATION PRO 17

表 1: 实验平台参数

实验所需环境均已经配置完毕，如图1所示。



```
Luhaozhhe@Luhaozhhe: ~/Desktop
Luhaozhhe@Luhaozhhe:~/Desktop$ qemu-system-riscv64 --version
QEMU emulator version 7.0.0
Copyright (c) 2003-2022 Fabrice Bellard and the QEMU Project developers
Luhaozhhe@Luhaozhhe:~/Desktop$ riscv64-unknown-linux-gnu-gcc --version
riscv64-unknown-linux-gnu-gcc () 12.2.0
Copyright (C) 2022 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.
Luhaozhhe@Luhaozhhe:~/Desktop$
```

图 1: riscv 和 qemu 的环境配置

## 二、 任务一：了解你的编译器

### (一) 完整编译过程

根据理论课上的知识，我们知道，一个完整的编译过程主要分为以下几个过程：

- 预处理
- 编译
- 汇编
- 链接 & 加载

主要流程图如图2所示：

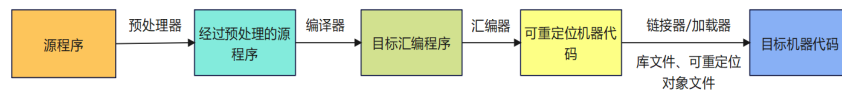
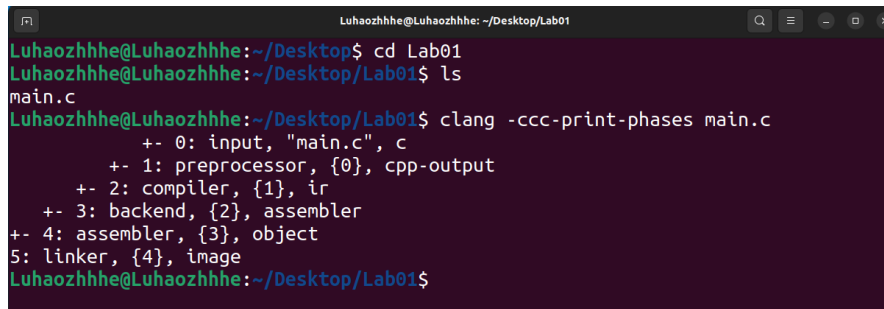


图 2: 编译的主要阶段

我们可以在虚拟机中编写 main.c 文件来对编译过程进行测试，我们的程序源码如下所示：

```
1  main()
2  {
3      int a,b,i,t,n;
4      a=0;
5      b=1;
6      i=1;
7      cin>>n;
8      cout<<a<<endl;
9      cout<<b<<endl;
10     while(i<n)
11     {
12         t=b;
13         b=a+b;
14         cout<<b<<endl;
15         a=t;
16         i=i+1;
17     }
18 }
```

借助 `clang -ccc-print-phases main.c` 指令，我们可以得到编译的整个流程，如图3所示。



```
Luhaozhhe@Luhaozhhe: ~/Desktop/Lab01
Luhaozhhe@Luhaozhhe:~/Desktop$ cd Lab01
Luhaozhhe@Luhaozhhe:~/Desktop/Lab01$ ls
main.c
Luhaozhhe@Luhaozhhe:~/Desktop/Lab01$ 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
Luhaozhhe@Luhaozhhe:~/Desktop/Lab01$
```

图 3: main.c 编译测试

## (二) 预处理器

### 1. 预处理阶段功能

预处理阶段是编译过程的第一个阶段，实际上在编译之前就已经执行了。预处理阶段主要完成以下这些任务：

1. **宏定义替换：**预处理器会查询所有的宏定义，并用其对应的文本替换程序中的宏。举个例子，如果在程序中出现 `#define PAI 3.14`，在进行预处理后，就会将程序中所有的 `PAI` 替换为 `3.14`；
2. **文件包含处理：**对于 `#include` 指令，预处理器会将指定的文件内容插入到该指令所在的位置。这通常用于包含头文件。例如，`#include <stdio.h>` 会将 `stdio.h` 头文件的内容插入到程序中；
3. **条件编译：**预处理器会根据 `#if`、`#ifdef`、`#ifndef`、`#else`、`#elif` 和 `#endif` 指令，决定是否编译代码的特定部分。
4. **去除注释：**预处理过程中，会自动去除掉所有的注释行，包括单行注释和多行注释；
5. **生成预处理后的源代码：**完成以上步骤后，预处理器会生成一个预处理后的文件，这个文件将在后续的编译过程中使用。

预处理阶段生成的代码文件格式一般是 `.i` 或者 `.ii`，然后再传输到下一个步骤进行编译。

我们可以通过使用 `-E` 使 `gcc` 只进行预处理流程，使用参数 `-o` 改变输出的文件名，所以我们使用 `gcc main.c -E -o main.i` 命令来完成我们的预处理操作。

### 2. 实验验证与结果分析

为了验证以上这些预处理器的功能，我们需要对我们的斐波那契数列程序做一些改进。

为了验证宏定义替换的功能，我们在代码中加上一行 `#define mian main`，然后再将源程序中的 `main` 修改为 `mian`；为了验证文件包含处理，我们直接在源程序最前面加上 `#include<stdio.h>`；为了验证我们的去除注释，我们在代码中添加一些单行注释与多行注释。

我们的 `main.c` 代码的前后状态如图4所示。





图 4: main.c 前后状态对比

然后，我们在命令行内输入 `gcc main.c -E -o main.i`，发现生成了一个预处理后的 `main.i` 文件 (图5)，代码行数从原来的 31 行变成了 769 行，说明预处理器成功执行了文件包含处理；我们还发现所有编写的单行、多行注释都消失了，这说明预处理器可以去除注释；还发现我们的宏定义也直接被替换了，说明预处理器可以直接执行宏定义的替换。

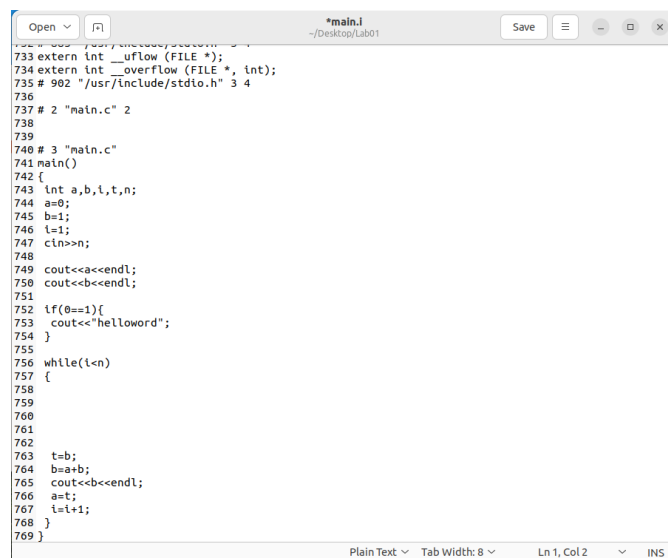


图 5: 生成预处理后的代码

综上所述，预处理器的功能得以验证。

### (三) 编译器

编译器一般分为以下几个步骤，如图6所示。

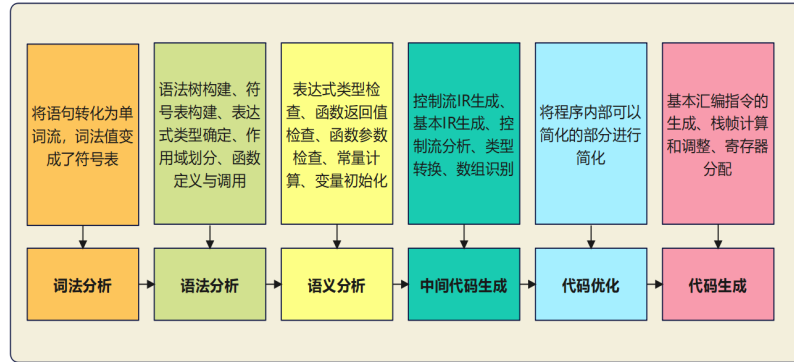


图 6: 编译器的主要流程

下面，我就详细分析一下这六个步骤都干了些什么。

### 1. 词法分析

词法分析的主要功能是：将我们的源程序转化为单词序列，以便于后续的词法分析能够更好地理解代码的结构与语法。

我们还是使用预处理过程中的那一段代码，通过输入命令 `clang -E -Xclang -dump-tokens main.c`，我们运行得到了语法分析的结果，如图7所示。

```

identifier 'endl' Loc=<main.c:9:11>
semi ';' Loc=<main.c:9:15>
while 'while' [StartOfLine] [LeadingSpace] Loc=<main.c:10:2>
l_paren '(' Loc=<main.c:10:7>
identifier 'i' Loc=<main.c:10:8>
less '<' Loc=<main.c:10:9>
identifier 'n' Loc=<main.c:10:10>
r_paren ')' Loc=<main.c:10:11>
l_brace '{' [StartOfLine] [LeadingSpace] Loc=<main.c:11:2>
identifier 't' [StartOfLine] [LeadingSpace] Loc=<main.c:12:3>
equal '=' Loc=<main.c:12:4>
identifier 'b' Loc=<main.c:12:5>
semi ';' Loc=<main.c:12:6>
identifier 'b' [StartOfLine] [LeadingSpace] Loc=<main.c:13:3>
equal '=' Loc=<main.c:13:4>
identifier 'a' Loc=<main.c:13:5>
plus '+' Loc=<main.c:13:6>
identifier 'b' Loc=<main.c:13:7>
semi ';' Loc=<main.c:13:8>
identifier 'cout' [StartOfLine] [LeadingSpace] Loc=<main.c:14:3>
lessless '<<' Loc=<main.c:14:7>
identifier 'b' Loc=<main.c:14:9>
lessless '<<' Loc=<main.c:14:10>
identifier 'endl' Loc=<main.c:14:12>
semi ';' Loc=<main.c:14:16>
identifier 'a' [StartOfLine] [LeadingSpace] Loc=<main.c:15:3>
equal '=' Loc=<main.c:15:4>
identifier 't' Loc=<main.c:15:5>
semi ';' Loc=<main.c:15:6>
identifier 'i' [StartOfLine] [LeadingSpace] Loc=<main.c:16:3>
equal '=' Loc=<main.c:16:4>
identifier 'i' Loc=<main.c:16:5>
plus '+' Loc=<main.c:16:6>
numeric_constant '1' Loc=<main.c:16:7>
semi ';' Loc=<main.c:16:8>
r_brace '}' [StartOfLine] [LeadingSpace] Loc=<main.c:17:2>
r_brace '}' [StartOfLine] Loc=<main.c:18:1>
eof '' Loc=<main.c:33:2>
Luhaozhhe@Luhaozhhe: ~/Desktop/Lab01$
  
```

图 7: 词法分析 main.c 的结果

具体内容如下所示，我们来逐一分析一下该词法分析的结果。

- 1 identifier 'main' [StartOfLine] Loc=<main.c:1:1>
- 2 l\_paren '(' Loc=<main.c:1:5>
- 3 r\_paren ')' Loc=<main.c:1:6>
- 4 l\_brace '{' [StartOfLine] Loc=<main.c:2:1>

首先声明了一个标识符 (identifier)main，表示程序的入口点。后面两行中的 paren 用于定义 main 函数的参数列表，无参数。后面的 brace 表示 main 函数代码的开始与结束。

```

1 int 'int' [StartOfLine] [LeadingSpace] Loc=<main.c:3:2>
2 identifier 'a' [LeadingSpace] Loc=<main.c:3:6>
3 comma ',' Loc=<main.c:3:7>
4 identifier 'b' Loc=<main.c:3:8>
5 comma ',' Loc=<main.c:3:9>
6 identifier 'i' Loc=<main.c:3:10>
7 comma ',' Loc=<main.c:3:11>
8 identifier 't' Loc=<main.c:3:12>
9 comma ',' Loc=<main.c:3:13>
10 identifier 'n' Loc=<main.c:3:14>
11 semi ';' Loc=<main.c:3:15>

```

首先是 int 型变量的申明, identifier 'a', identifier 'b', identifier 'i' 和 identifier 't': 分别声明了整型变量 a, b, i, t。comma 表示字符之间的分隔符, semi 表示一个语句的结束符号。

```

1 identifier 'a' [StartOfLine] [LeadingSpace] Loc=<main.c:4:2>
2 equal '=' Loc=<main.c:4:3>
3 numeric_constant '0' Loc=<main.c:4:4>
4 semi ';' Loc=<main.c:4:5>
5 identifier 'b' [StartOfLine] [LeadingSpace] Loc=<main.c:5:2>
6 equal '=' Loc=<main.c:5:3>
7 numeric_constant '1' Loc=<main.c:5:4>
8 semi ';' Loc=<main.c:5:5>
9 identifier 'i' [StartOfLine] [LeadingSpace] Loc=<main.c:6:2>
10 equal '=' Loc=<main.c:6:3>
11 numeric_constant '1' Loc=<main.c:6:4>
12 semi ';' Loc=<main.c:6:5>

```

此处完成了对 a、b 和 i 的变量初始化, 将 a 赋值为 0, b 赋值为 1, i 赋值为 1。其中等号使用 equal 来进行表示。

```

1 identifier 'cin' [StartOfLine] [LeadingSpace] Loc=<main.c:7:2>
2 greatergreater '>>' Loc=<main.c:7:5>
3 identifier 'n' Loc=<main.c:7:7>
4 semi ';' Loc=<main.c:7:8>
5 identifier 'cout' [StartOfLine] [LeadingSpace] Loc=<main.c:8:2>
6 lessless '<<' Loc=<main.c:8:6>
7 identifier 'a' Loc=<main.c:8:8>
8 lessless '<<' Loc=<main.c:8:9>
9 identifier 'endl' Loc=<main.c:8:11>
10 semi ';' Loc=<main.c:8:15>
11 identifier 'cout' [StartOfLine] [LeadingSpace] Loc=<main.c:9:2>
12 lessless '<<' Loc=<main.c:9:6>
13 identifier 'b' Loc=<main.c:9:8>
14 lessless '<<' Loc=<main.c:9:9>
15 identifier 'endl' Loc=<main.c:9:11>
16 semi ';' Loc=<main.c:9:15>

```

此处，使用标识符 cin 来表示输入，使用 greatergreater 来表示 »，表示输入的操作，使用 lessless 表示输出的符号，使用标识符 cout 来表示输出的操作，使用标识符 endl 来表示换行的操作。

```

1 while 'while'      [StartOfLine] [LeadingSpace]    Loc=<main.c:10:2>
2 l_paren '('        Loc=<main.c:10:7>
3 identifier 'i'      Loc=<main.c:10:8>
4 less '<'           Loc=<main.c:10:9>
5 identifier 'n'      Loc=<main.c:10:10>
6 r_paren ')'        Loc=<main.c:10:11>
7 l_brace '{'        [StartOfLine] [LeadingSpace]    Loc=<main.c:11:2>
8 identifier 't'      [StartOfLine] [LeadingSpace]    Loc=<main.c:12:3>
9 equal '='          Loc=<main.c:12:4>
10 identifier 'b'     Loc=<main.c:12:5>
11 semi ';'          Loc=<main.c:12:6>
12 identifier 'b'     [StartOfLine] [LeadingSpace]    Loc=<main.c:13:3>
13 equal '='          Loc=<main.c:13:4>
14 identifier 'a'     Loc=<main.c:13:5>
15 plus '+'          Loc=<main.c:13:6>
16 identifier 'b'     Loc=<main.c:13:7>
17 semi ';'          Loc=<main.c:13:8>
18 identifier 'cout'  [StartOfLine] [LeadingSpace]    Loc=<main.c:14:3>
19 lessless '<<'     Loc=<main.c:14:7>
20 identifier 'b'     Loc=<main.c:14:9>
21 lessless '<<'     Loc=<main.c:14:10>
22 identifier 'endl'  Loc=<main.c:14:12>
23 semi ';'          Loc=<main.c:14:16>

```

此处出现了 while 符号，定义了 while 循环，用 less 来定义 <，其他都和前面的标识符定义相类似，就不再提及了。

```

1 identifier 'a'      [StartOfLine] [LeadingSpace]    Loc=<main.c:15:3>
2 equal '='          Loc=<main.c:15:4>
3 identifier 't'      Loc=<main.c:15:5>
4 semi ';'          Loc=<main.c:15:6>
5 identifier 'i'      [StartOfLine] [LeadingSpace]    Loc=<main.c:16:3>
6 equal '='          Loc=<main.c:16:4>
7 identifier 'i'      Loc=<main.c:16:5>
8 plus '+'          Loc=<main.c:16:6>
9 numeric_constant '1' Loc=<main.c:16:7>
10 semi ';'          Loc=<main.c:16:8>
11 r_brace '}'        [StartOfLine] [LeadingSpace]    Loc=<main.c:17:2>
12 r_brace '}'        [StartOfLine] Loc=<main.c:18:1>
13 eof ''            Loc=<main.c:33:2>

```

此处内容和前面也是类似的，最后的 eof 标识符表示程序的结束。

我们发现，原来的 main.c 文件中的字符串被扫描并分解，识别为一个一个的单词，并且还标明了每个单词的类型。**我们成功验证了词法分析的功能。**

## 2. 语法分析

语法分析的功能是：将前一步词法分析获得的词法单元构建成抽象语法树 (Abstract Tree, 即 AST)。我们可以通过命令行 `clang -E -Xclang -ast-dump main.c` 来完成语法分析的构建。构建结果如图8所示。

```
TranslationUnitDecl @0x19e08 <<invalid sloc>> <invalid sloc>
  TypedefDecl @0x1a638 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
    BuiltinType @0x1a300 '__int128'
  TypedefDecl @0x1a6a0 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
    BuiltinType @0x1a3f0 'unsigned __int128'
  TypedefDecl @0x1a9a8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString_tag 'struct __NSConstantString_tag'
    RecordType @0x1a780 'struct __NSConstantString_tag'
      Record @0x1a6f8 '__NSConstantString_tag'
  TypedefDecl @0x1aa40 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
    PointerType @0x1aa00 'char *'
    BuiltinType @0x19eb0 'char'
  TypedefDecl @0x1ad38 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag[1]'
    ConstantArrayType @0x1ace0 'struct __va_list_tag[1]' 1
      RecordType @0x1ab20 'struct __va_list_tag'
        Record @0x1aa98 '__va_list_tag'
FunctionDecl @0x70900 <<invalid sloc>> <invalid sloc> line:1:1 main 'int ()'
  CompoundStmt @0x71200 <invalid sloc> line:2:1, line:18:1
    DeclStmt @0x70c98 <invalid sloc> line:3:2, col:15
      VarDecl @0x70a00 <col:2, col:6> col:6 used a 'int'
      VarDecl @0x70a80 <col:2, col:8> col:8 used b 'int'
      VarDecl @0x70b00 <col:2, col:10> col:10 used t 'int'
      VarDecl @0x70b80 <col:2, col:12> col:12 used t 'int'
      VarDecl @0x70c00 <col:2, col:14> col:14 used m 'int'
      BinaryOperator @0x70cf0 <col:4:2, col:4> 'int' '-'
        DeclRefExpr @0x70cb0 <col:2> 'int' lvalue Var @0x70a00 'a' 'int'
        IntegerLiteral @0x70cd0 <col:4> 'int' 0
      BinaryOperator @0x70d50 <col:5:2, col:4> 'int' '='
        DeclRefExpr @0x70d10 <col:2> 'int' lvalue Var @0x70a80 'b' 'int'
        IntegerLiteral @0x70d30 <col:4> 'int' 1
      BinaryOperator @0x70db0 <col:6:2, col:4> 'int' '-'
        DeclRefExpr @0x70d70 <col:2> 'int' lvalue Var @0x70b00 't' 'int'
        IntegerLiteral @0x70d90 <col:4> 'int' 1
      WhileStmt @0x711e8 <col:10:2, line:17:2>
        BinaryOperator @0x70ee8 <col:10:8, col:10> 'int' 'c'
        ImplicitCastExpr @0x70eb8 <col:8> 'int' <lvalue to rvalue>
          DeclRefExpr @0x70e78 <col:8> 'int' lvalue Var @0x70b80 't' 'int'
          ImplicitCastExpr @0x70e00 <col:10> 'int' <lvalue to rvalue>
            DeclRefExpr @0x70e98 <col:10> 'int' lvalue Var @0x70c00 'm' 'int'
```

图 8: 语法分析 main.c 的结果

我们对语法分析的结果做一下详细的分析。

```

1 TranslationUnitDecl 0x8c1dd8 <<invalid sloc>> <invalid sloc>
2 | -TypedefDecl 0x8c2600 <<invalid sloc>> <invalid sloc> implicit __int128_t '
  __int128'
3 | '-BuiltinType 0x8c23a0 '__int128'
4 | -TypedefDecl 0x8c2670 <<invalid sloc>> <invalid sloc> implicit __uint128_t '
  unsigned __int128'
5 | '-BuiltinType 0x8c23c0 'unsigned __int128'
6 | -TypedefDecl 0x8c2978 <<invalid sloc>> <invalid sloc> implicit
  __NSConstantString_tag 'struct __NSConstantString_tag'
7 | '-RecordType 0x8c2750 'struct __NSConstantString_tag'
8 | '-Record 0x8c26c8 '__NSConstantString_tag'
9 | -TypedefDecl 0x8c2a10 <<invalid sloc>> <invalid sloc> implicit
  __builtin_ms_va_list 'char *'
10 | '-PointerType 0x8c29d0 'char *'
11 | '-BuiltinType 0x8c1e80 'char'
12 | -TypedefDecl 0x8c2d08 <<invalid sloc>> <invalid sloc> implicit
  __builtin_va_list 'struct __va_list_tag[1]'
13 | '-ConstantArrayType 0x8c2cb0 'struct __va_list_tag[1]' 1
14 | '-RecordType 0x8c2af0 'struct __va_list_tag'
15 | '-Record 0x8c2a68 '__va_list_tag'
```

TranslationUnitDecl 是对整个翻译单元的申明，通常用于表示一个源文件。TypedefDecl 一般用于表示类型的定义申明。\_\_int128\_t 定义为 \_\_int128 类型，这是一个 128 位的整数类型；\_\_uint128\_t 定义为 unsigned \_\_int128 类型，这是一个无符号的 128 位整数类

型; `__NSConstantString` 定义为 `struct __NSConstantString_tag` 类型, 这是一个结构体类型, 通常用于 Objective-C 中的字符串字面量; `__builtin_ms_va_list` 定义为 `char *` 类型, 这是一个指向字符的指针, 用于表示 Microsoft 变长参数列表; `__builtin_va_list` 定义为 `struct __va_list_tag[1]` 类型, 这是一个固定大小的数组, 包含一个结构体元素, 用于表示变长参数列表。

`BuiltinType` 表示内置类型, 如 `__int128` 和 `unsigned __int128`; `RecordType` 表示记录类型, 即结构体或联合体类型; `Record` 表示具体的结构体或联合体定义; `PointerType` 表示指针类型, 如 `char *`; `ConstantArrayType` 表示固定大小的数组类型, 如 `struct __va_list_tag[1]`。

```

1  '-FunctionDecl_0x9189a0_<main.c:1:1, line:18:1>_line:1:1_main_'int ()'
2  _-CompoundStmt_0x9192a8_<line:2:1, line:18:1>
3      |-DeclStmt_0x918d38_<line:3:2, col:15>
4          |-VarDecl_0x918aa0_<col:2, col:6> col:6 used a 'int'
5          |-VarDecl_0x918b20_<col:2, col:8> col:8 used b 'int'
6          |-VarDecl_0x918ba0_<col:2, col:10> col:10 used i 'int'
7          |-VarDecl_0x918c20_<col:2, col:12> col:12 used t 'int'
8          | '-VarDecl_0x918ca0_<col:2, col:14>_col:14_used_n_'int'
9      _-BinaryOperator_0x918d90_<line:4:2, col:4>_'int'_='
10     _-DeclRefExpr_0x918d50_<col:2>_'int'_lvalue_Var_0x918aa0_'a'_int'
11     _-IntegerLiteral_0x918d70_<col:4> 'int' 0
12     |-BinaryOperator_0x918df0_<line:5:2, col:4> 'int' '='
13     | |-DeclRefExpr_0x918db0_<col:2> 'int' lvalue Var 0x918b20 'b' 'int'
14     | | '-IntegerLiteral_0x918dd0_<col:4>_'int'_1
15     _-BinaryOperator_0x918e50_<line:6:2, col:4>_'int'_='
16     _-DeclRefExpr_0x918e10_<col:2>_'int'_lvalue_Var_0x918ba0_'i'_int'
17     _-IntegerLiteral_0x918e30_<col:4> 'int' 1
18     '-WhileStmt_0x919288_<line:10:2, line:17:2>
19     _-BinaryOperator_0x918f88_<line:10:8, col:10>_'int'_<
20     _-ImplicitCastExpr_0x918f58_<col:8>_'int'_<LValueToRValue>
21     _-DeclRefExpr_0x918f18_<col:8> 'int' lvalue Var 0x918ba0 'i' 'int'
22     | '-ImplicitCastExpr_0x918f70_<col:10>_'int'_<LValueToRValue>
23     _-DeclRefExpr_0x918f38_<col:10> 'int' lvalue Var 0x918ca0 'n' 'int'
24     '-CompoundStmt_0x919258_<line:11:2, line:17:2>

```

其中, `FunctionDecl` 表示一个函数声明, `CompoundStmt` 表示一个复合语句, 通常用于函数体; `DeclStmt` 表示一个声明语句块。这里声明了五个变量 `a`, `b`, `i`, `t`, `n`, 它们都是 `int` 类型; `VarDecl` 表示变量声明; 每个变量声明都有一个唯一的地址, 一个在源代码中的位置, 以及一个使用的标识; `BinaryOperator` 表示二元运算符; `DeclRefExpr` 表示对变量的引用; `IntegerLiteral` 表示整数字面量; `WhileStmt` 表示一个 `while` 循环; `ImplicitCastExpr` 表示隐式类型转换, 用于将变量的值转换为需要的类型; `CompoundStmt` 在 `while` 循环中表示复合语句, 即循环体。

我们发现, 语法分析过程中, 将我们输入的程序的语法结构进行识别, 并构造出一颗具有层次感的抽象语法树, 为后续的语义分析提供了基础。

另外, 在语法分析的结果中, 出现了图9所示报错信息:

```
Luhaozhhe@Luhaozhhe:~/Desktop/Lab01$ clang -E -Xclang -ast-dump main.c
main.c:1:1: warning: type specifier missing, defaults to 'int' [-Wimplicit-int]
main()
^
main.c:7:2: error: use of undeclared identifier 'cin'
    cin>>n;
    ^
main.c:8:2: error: use of undeclared identifier 'cout'
    cout<<a<<endl;
    ^
main.c:8:11: error: use of undeclared identifier 'endl'
    cout<<a<<endl;
            ^
main.c:9:2: error: use of undeclared identifier 'cout'
    cout<<b<<endl;
    ^
main.c:9:11: error: use of undeclared identifier 'endl'
    cout<<b<<endl;
            ^
main.c:14:3: error: use of undeclared identifier 'cout'
    cout<<b<<endl;
    ^
main.c:14:12: error: use of undeclared identifier 'endl'
    cout<<b<<endl;
               ^
```

图 9: 语法分析中的报错信息

刚开始有些许疑惑, 后来阅读了报错信息后, 发现是因为 c 程序中没有 cin、cout、endl 等字符串, 所以报错了。这也是语法分析的一个功能, 可以帮助我们检验出程序中的错误信息。为了再次检验这个功能, 我们又故意编写了一个错误的程序 test.c(如下所示), 来对该功能进行测试。

```
1  main(){
2      int i;
3      int j;
4      i=j+;
5      i
6      fo(int i){
7          print("111");
8      }
9
10 }
```

我们经过测试, 得到以下结果 (图10):

```

Luhaozhhe@Luhaozhhe:~/Desktop/Lab01$ clang -E -Xclang -ast-dump test.c
test.c:1:1: warning: type specifier missing, defaults to 'int' [-Wimplicit-int]
main(){
^
test.c:4:6: error: expected expression
    i=j+;
      ^
test.c:5:3: error: expected ';' after expression
    i
    ^
test.c:6:2: warning: implicit declaration of function 'fo' is invalid in C99 [-Wimplicit-function-declaration]
    fo(int i){
    ^
test.c:6:5: error: expected expression
    fo(int i){
      ^
test.c:5:2: warning: expression result unused [-Wunused-value]
    i
    ^
TranslationUnitDecl @0x21a1e88 <<invalid sloc>> <invalid sloc>
TypedDecl @0x21a2630 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
  BuiltinType @0x21a23d0 'int128'
TypedDecl @0x21a26a0 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
  BuiltinType @0x21a23f0 'unsigned __int128'
TypedDecl @0x21a29a0 <<invalid sloc>> <invalid sloc> implicit __NSConstantString_tag 'struct __NSConstantString_tag'
  RecordType @0x21a2780 'struct __NSConstantString_tag'
    Record @0x21a26f8 'NSConstantString_tag'
TypedDecl @0x21a2a40 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'char *'
  PointerType @0x21a2a00 'char *'
  BuiltinType @0x21a1eb0 'char'
TypedDecl @0x21a2c00 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag[1]'
  ConstantArrayType @0x21a2ce0 'struct __va_list_tag[1]' 1
    RecordType @0x21a2b20 'struct __va_list_tag'
      Record @0x21a2a98 '__va_list_tag'
FunctionDecl @0x21f8970 <test.c:1:1, line:10:1> main 'int ()'
  CompoundStmt @0x21f8cf8 <col:7, line:10:1>
    DeclStmt @0x21f8ad8 <line:2:2, col:7>
      VarDecl @0x21f8a70 <col:2, col:6> col:6 used i 'int'
    DeclStmt @0x21f8b70 <line:3:2, col:7>
      VarDecl @0x21f8b08 <col:2, col:6> col:6 used j 'int'
    ImplicitCastExpr @0x21f8be8 <line:5:2> 'int' <LValueToRValue>
      DeclRefExpr @0x21f8bc8 <col:2> 'int' lvalue Var @0x21f8a70 'i' 'int'
3 warnings and 3 errors generated.
Luhaozhhe@Luhaozhhe:~/Desktop/Lab01$

```

图 10: test.c 语法分析测试

下面我们对语法分析的报错信息进行分析。

```

1  main.c:1:1: warning: type specifier missing, defaults to 'int' [-Wimplicit-
   int]
2  main(){
3  ^
4  main.c:4:6: error: expected expression
       i=j+;
       ^
5
6
7  main.c:5:3: error: expected ';' after expression
       i
       ^
8
9
10 ;
11
12 main.c:6:2: warning: implicit declaration of function 'fo' is invalid in C99
   [-Wimplicit-function-declaration]
       fo(int i){
13     ^
14
15 main.c:6:5: error: expected expression
       fo(int i){
16     ^
17
18 main.c:5:2: warning: expression result unused [-Wunused-value]
       i

```

可以发现，表明在 main 函数定义时缺少了类型说明符，我们应该用 int main 而不是 main；然后就是 i=j+，加号后面没东西了所以出现了报错；然后是最后一行的 i 没有标点符号，报错；然后就是 for 打成了 fo，因为没有这个词语所以出现了报错。

我们发现，语法分析成功帮我们找出了所有的语法错误。

综上所述，我们成功地验证了语法分析的生成抽象语法树以及检查代码语法的功能。



### 3. 语义分析

使用语法树和符号表中信息来检查源程序是否与语言定义语义一致，进行类型检查等。这一步是很关键的，能够对代码进行检测是否正确，深度理解代码，确保代码可以正常执行。

语义分析主要完成的任务有：

- **类型检查：**确保所有变量和表达式的类型是正确的，并且类型转换是合法的。
- **变量和函数的作用域解析：**确定变量和函数在代码中的作用域，即它们可以被访问和修改的区域，其中包括局部变量、全局变量、参数传递等等。
- **名称解析：**将源代码中的标识符（变量名、函数名等）映射到它们在 AST 中的定义。
- **控制流分析：**分析程序的执行路径，检查是否存在未初始化的变量、未达到的代码、死循环等。
- **数据流分析：**分析数据在程序中的流动，以优化代码和检测错误。
- **符号表的构建：**构建一个数据结构（通常称为符号表），用于存储关于程序中所有符号（变量、函数等）的信息，如类型、作用域、存储位置等。
- **错误检测和报告：**在发现语义错误时，编译器会生成错误消息，告诉我们哪里出了问题，以及可能的原因。

### 4. 中间代码生成

中间代码生成也是很关键的一步，我们使用命令行 `clang -S -emit-llvm main.c` 来生成中间代码，如图11所示。

```

Luhaozhhe@Luhaozhhe:~/Desktop$ clang -S -emit-llvm main.c
main.c:6:5: warning: implicitly declaring library function 'scanf' with type 'int (const char *restrict, ...)' [-Wimplicit-function-declaration]
scanf("%d", &n); // 读取输入
^
main.c:6:5: note: include the header <stdio.h> or explicitly provide a declaration for 'scanf'
main.c:7:5: warning: implicitly declaring library function 'printf' with type 'int (const char *, ...)' [-Wimplicit-function-declaration]
printf("%d\n", a); // 输出 a
^
main.c:7:5: note: include the header <stdio.h> or explicitly provide a declaration for 'printf'
2 warnings generated.

```

图 11: main.ll 生成

运行后我们得到一个 main.ll 文件，内容如下所示。

```

1 ; ModuleID = 'main.c'
2 source_filename = "main.c"
3 target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
  :16:32:64-S128"
4 target triple = "x86_64-pc-linux-gnu"
5
6 @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
7 @.str.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
8
9 ; Function Attrs: noinline nounwind optnone uwtable
10 define dso_local i32 @main() #0 {

```

```

11  %1 = alloca i32, align 4
12  %2 = alloca i32, align 4
13  %3 = alloca i32, align 4
14  %4 = alloca i32, align 4
15  %5 = alloca i32, align 4
16  %6 = alloca i32, align 4
17  store i32 0, i32* %1, align 4
18  store i32 0, i32* %2, align 4
19  store i32 1, i32* %3, align 4
20  store i32 1, i32* %4, align 4
21  %7 = call i32 (i8*, ...) @scanf(i8* noundef getelementptr inbounds ([3 x i8
    ], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %6)
22  %8 = load i32, i32* %2, align 4
23  %9 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
    i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %8)
24  %10 = load i32, i32* %3, align 4
25  %11 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
    i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %10)
26  br label %12
27
28 12:                                     # preds = %16, %0
29  %13 = load i32, i32* %4, align 4
30  %14 = load i32, i32* %6, align 4
31  %15 = icmp slt i32 %13, %14
32  br i1 %15, label %16, label %26
33
34 16:                                     # preds = %12
35  %17 = load i32, i32* %3, align 4
36  store i32 %17, i32* %5, align 4
37  %18 = load i32, i32* %2, align 4
38  %19 = load i32, i32* %3, align 4
39  %20 = add nsw i32 %18, %19
40  store i32 %20, i32* %3, align 4
41  %21 = load i32, i32* %3, align 4
42  %22 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
    i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %21)
43  %23 = load i32, i32* %5, align 4
44  store i32 %23, i32* %2, align 4
45  %24 = load i32, i32* %4, align 4
46  %25 = add nsw i32 %24, 1
47  store i32 %25, i32* %4, align 4
48  br label %12, !llvm.loop !6
49
50 26:                                     # preds = %12
51  ret i32 0
52 }
53
54 declare i32 @scanf(i8* noundef, ...) #1

```

```

55
56 declare i32 @printf(i8* noundef, ...) #1
57
58 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" }
59 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" }
60
61 !llvm.module.flags = !{!0, !1, !2, !3, !4}
62 !llvm.ident = !{!5}
63
64 !0 = !{i32 1, !"wchar_size", i32 4}
65 !1 = !{i32 7, !"PIC_Level", i32 2}
66 !2 = !{i32 7, !"PIE_Level", i32 2}
67 !3 = !{i32 7, !"uwtable", i32 1}
68 !4 = !{i32 7, !"frame-pointer", i32 2}
69 !5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
70 !6 = distinct !{!6, !7}
71 !7 = !{!"llvm.loop.mustprogress"}

```

从上面的中间代码中我们很容易发现，代码已经被翻译成了 `llvm` 格式。我们可以看到，在代码的最开始部分给出了此模块的 ID 名称以及源文件的程序名称，接下来还给出了目标数据布局、目标三元组等信息。

下面所输出的两个 `@` 开头的 `@.str` 和 `@.str1` 是两个全局化的变量，分别用来表示 `scanf` 和 `printf` 的格式化字符串。

第 10 行开始的 `@main` 表示该处为程序的入口点函数，前面一行则给出了有关该函数的一些属性，比如 `noinline`、`nounwind`、`optnone` 和 `uwtable`。

中间代码中还有很多输入输出、循环、退出程序等的代码，我们分部进行分析。利用 `fdump-tree-all-graph` 和 `fdump-rtl-all-graph` 可以获得中间阶段的多个输出，此处我们借用 VS code 中的 CFG 可视化插件来进行分析。

我们使用命令行 `gcc -fdump-tree-all-graph main.c` 进行中间过程的输出，得到了一大堆的文件，我们从中选取几个进行可视化分析即可。

我们分别对 `a-main.c.015t.cfg.dot` 进行可视化分析，结果如图12所示。

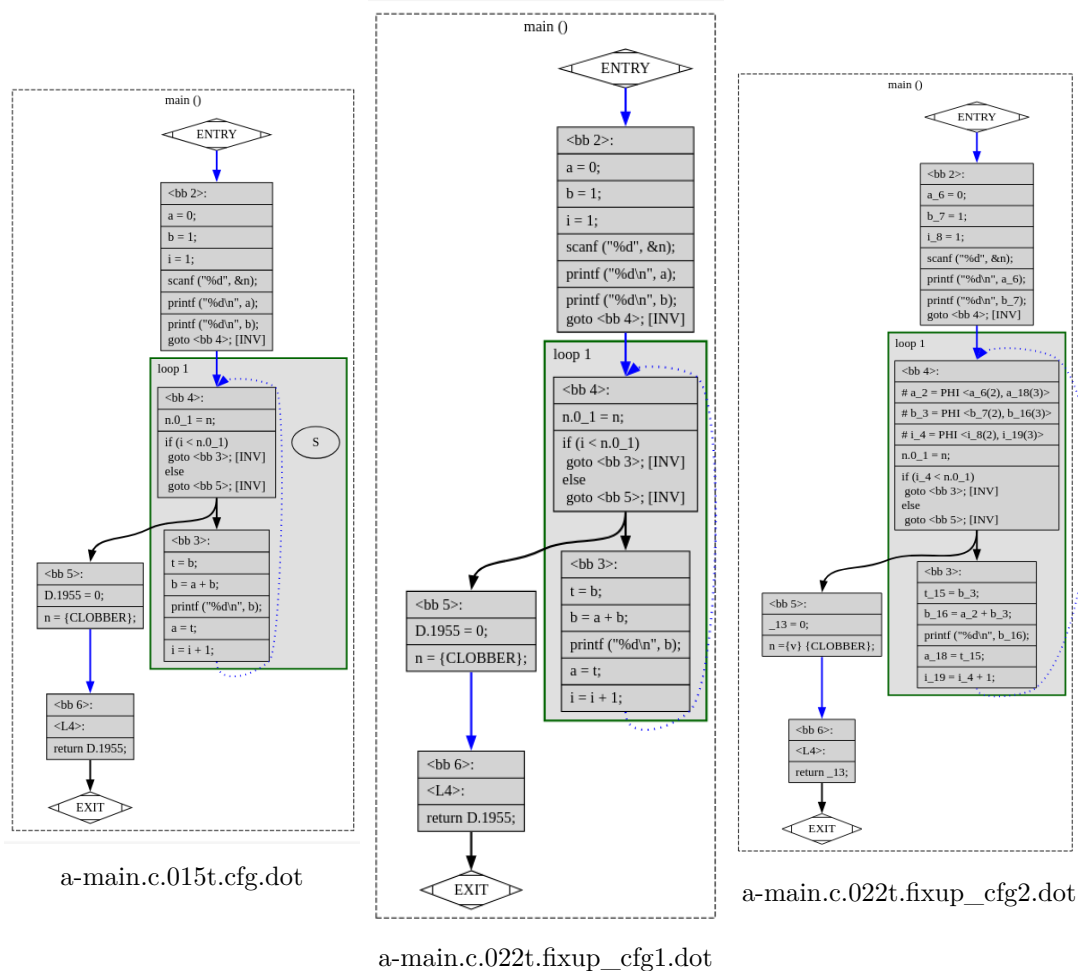


图 12: CFG 可视化分析

我们从上面的可视化结果中可以发现，可视化插件将 dot 代码转化为了控制流图，我们可以轻松分析出代码之间的逻辑关系。

1. 我们可以看出代码中间的跳转与分支关系。比如说，箭头指向的就是代表着语句的跳转方向；loop 代表的就是循环块的意思，一些 if 语句的判定还是在一个框中完成的；
2. 还可以看出，entry 代表着程序的入口处，而 exit 则代表了程序的出口处；
3. 实线所代表的是正常的运行顺序，而虚线则代表了 loop 循环完成一轮后进行的跳转；
4. 可以看到我们定义的变量都被替换掉了名字，比如 `t_15` 等，我猜测可能是因为程序的进行过程中，一个变量所代表的可能不止一个值，会随着程序的运行而不断变化，因此我们需要使用下划线来进行区分。

## 5. 代码优化

这一步简单来说，就是进行与机器无关的代码优化步骤改进中间代码，生成更好的目标代码。我们可以使用 `pass` 来对其进行优化。

在进行代码优化前，我们需要使用命令行 `llvm-as main.ll -o main.bc` 对我们的 ll 格式的文件进行转化，转化为 bc 格式的文件。

然后我们对几种不同的优化方式进行比较。我们分别使用 `opt -S -O1 main.bc -o main-O1.ll`、`opt -S -O2 main.bc -o main-O2.ll`、`opt -S -O3 main.bc -o main-O3.ll` 对原代码进行优化，分别得到 `main-O1`、`main-O2`、`main-O3` 文件。

我们先对生成的 `main-O1.ll` 与 `main.ll` 文件进行比较，如图13所示。

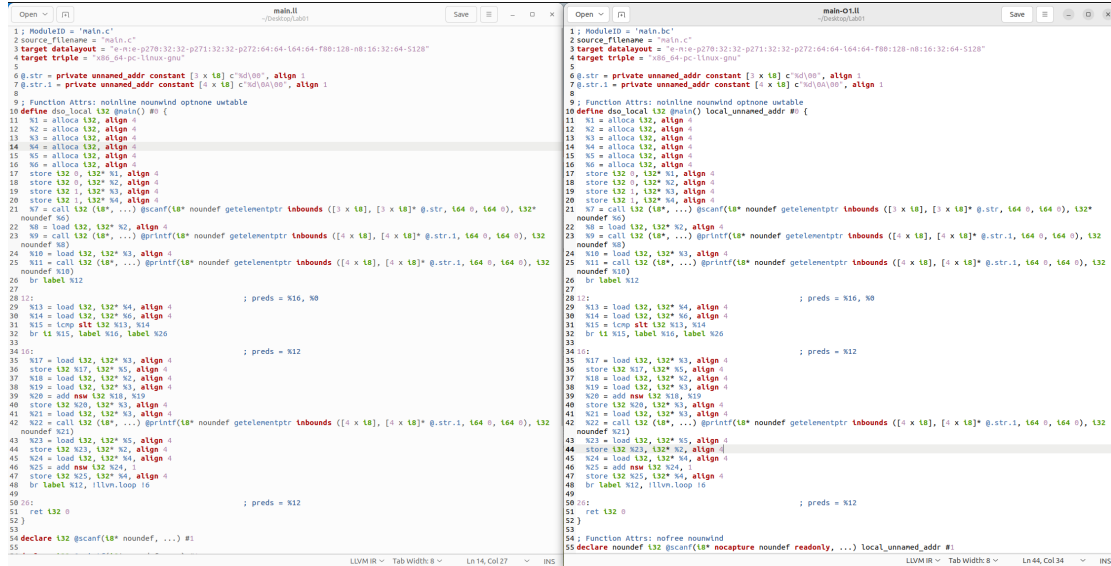


图 13: main 和 main-O1 的差异

然后我们再利用 `diff main.ll main-O1.ll` 命令，对比得到了优化前与优化后的代码差异，如下所示。经过分析，我们发现代码优化之处在于：

```
1 • 10c10
2 < define dso_local i32 @main() #0 {
3 ----
4 > define dso_local i32 @main() local_unnamed_addr #0 {
```

将 `define dso_local i32 @main() #0 {` 换成了 `define dso_local i32 @main() local_unnamed_addr #0 {`，表示在第二个文件中，`main` 函数的定义被添加了 `local_unnamed_addr` 属性。这个属性意味着函数的符号在链接时不会被导出，它只在当前模块内部可见。这个修改就是对我们的原代码进行了一定的优化，提升了运算的效率；

```
1 • 54c54,55
2 < declare i32 @scanf(i8* noundef, ...) #1
3 ----
4 > ; Function Attrs: nofree nounwind
5 > declare noundef i32 @scanf(i8* nocapture noundef readonly, ...)
   ↳ local_unnamed_addr #1
```

`scanf` 函数的声明被修改了。添加了 `nocapture`、`readonly` 属性，并且函数的符号也被标记为 `local_unnamed_addr`。`nocapture` 意味着传递给函数的指针不会被函数内部捕获或存储，`readonly` 意味着函数不会修改通过该指针指向的内存；

```

1 • 56c57,58
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

```

printf 函数的声明也被修改了，添加了相同的属性和 local\_unnamed\_addr;

```

1 • 59c61
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" }

```

在第二个文件中，属性列表被重新排序，并且添加了 nofree 和 nounwind 属性。nofree 意味着函数不会释放任何内存，nounwind 意味着函数不会抛出异常。

然后，我们在比较 O1 优化、O2 优化、O3 优化与未优化情况下程序性能，将中间代码进行汇编、链接之后生成可执行文件，分别运行斐波那契程序并带入不同的 n 进行比较，考虑到每一次运算的时间都不太一样，所以我们修改原来的程序，计算 10000 次运算所需要的平均时间，这样可以减少一定的误差。

修改完的程序如下所示：

```

1  #include<stdio.h>
2  #include<time.h>
3
4  int main() {
5      int a, b, i, t, n;
6      double totalTime = 0.00000; // 用于存储总时间
7      int trials = 10000; // 测试次数
8
9      scanf("%d", &n); // 读取输入
10     clock_t start, end;
11
12     for (int trial = 0; trial < trials; trial++) {
13         start = clock(); // 每次试验开始时记录时间
14
15         a = 0;

```

```

16     b = 1;
17     i = 1;
18
19     while (i < n) {
20         t = b;
21         b = a + b;
22         a = t;
23         i = i + 1;
24     }
25
26     end = clock(); // 每次试验结束时记录时间
27     totalTime += (double)(end - start) / CLOCKS_PER_SEC; //
    ↪ 累加每次试验的运行时间
28 }
29
30 printf(" 平均运行时间为: %f 秒\n", totalTime / trials); // 输出平均运行时间
31 return 0;
32 }

```

我们对三种优化方式分别进行测试, 通过一系列步骤, 分别生成最后的可执行文件, 如图14所示。

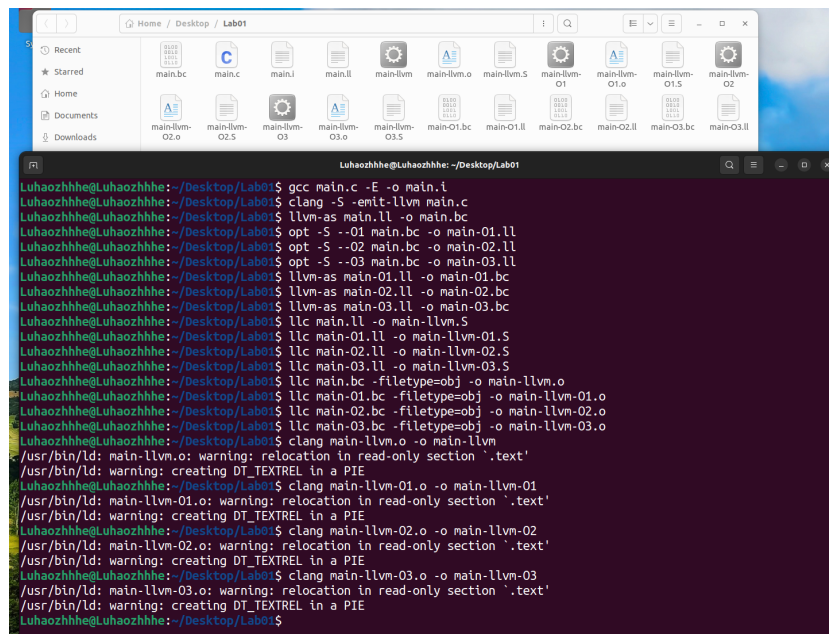


图 14: 三种代码优化生成的可执行文件

我们尝试使用不同的  $n$  值来进行测试, 分别使用  $n=10, 100, 1000, 10000, 100000, 1000000, 10000000$  来进行测试, 结果如图所示 (由于篇幅原因, 此处只展示  $n=10000000$  的情况)。



```
Luhaozhhe@Luhaozhhe: ~/Desktop/Lab01
Luhaozhhe@Luhaozhhe:~/Desktop/Lab01$ ./main-llvm
10000000
平均运行时间为：0.006950 秒
Luhaozhhe@Luhaozhhe:~/Desktop/Lab01$ ./main-llvm-O1
10000000
平均运行时间为：0.006350 秒
Luhaozhhe@Luhaozhhe:~/Desktop/Lab01$ ./main-llvm-O2
10000000
平均运行时间为：0.010194 秒
Luhaozhhe@Luhaozhhe:~/Desktop/Lab01$ ./main-llvm-O3
10000000
平均运行时间为：0.006157 秒
```

图 15: n=10000000 的情况

我们发现，main-llvm 的运行时间为 0.00695 秒，O1 成功对其进行了优化，但是 O2 貌似是反向进行了优化，不但没有提升，反而还减慢了运行的速度。O3 完成的优化是最好的，成功将平均时间优化到了 0.006157 秒。

## 6. 代码生成

以中间表示形式作为输入，将其映射到目标语言。

我们首先利用命令行 `llc main.ll -o main-llvm.S` 生成目标代码，然后利用 `gcc main.i -S -o main-x86.S` 生成 x86 代码，利用 `arm-linux-gnueabi-gcc main.i -S -o main-arm.S` 生成 arm 格式目标代码。生成结果如图16所示。




Name
 main-arm.S
 main-x86.S
 main-llvm.S

图 16: 代码生成

由于生成的代码量过于巨大，此处就不做展示了。

## (四) 汇编器

汇编过程实际上把汇编语言程序代码翻译成目标机器指令的过程，其最终生成的是可重定位的机器代码。汇编器会将汇编程序中的伪指令翻译成等价的机器语言指令，将程序中的分支和数据传输指令中用到的标号都放到一个符号表中，我们通过查表可以生成对应指令的二进制。

我们分别对上一个步骤中的三个生成代码文件进行汇编操作，对于 x86 格式的文件，使用命令行 `gcc main-x86.S -c -o main-x86.o` 生成 main-x86.o 文件；对于 arm 格式的文件，使用命令行 `arm-linux-gnueabi-gcc main-arm.S -c -o main-arm.o` 生成 main-arm.o 文件；对于 llvm 格式的文件，使用命令行 `llc main.bc -filetype=obj -o main-llvm.o` 生成 main-llvm.o 文件。生成结果如图17所示。



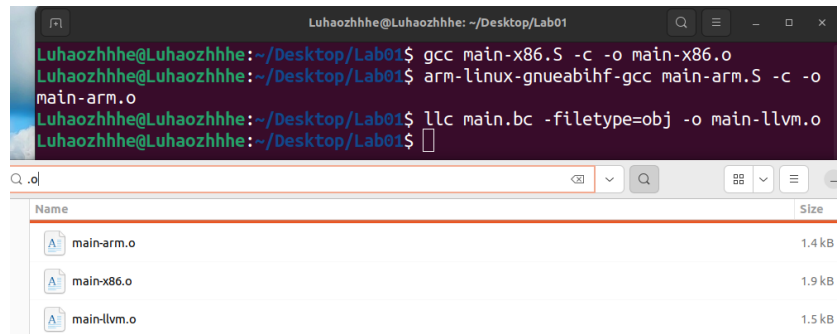


图 17: 用汇编器汇编三种格式的文件

此处我们仅对 x86 格式的文件进行进一步的分析。我们使用命令行 `objdump -d main-x86.o` 对 `main-x86.o` 进行反汇编，得到以下结果，我们对其进行分析。

```

1 0000000000000000 <main>:
2   0:  f3 0f 1e fa      endbr64      # 防止分支预测攻击的指令
3   4:  55              push    %rbp      # 保存旧的基指针
4   5:  48 89 e5        mov     %rsp,%rbp  # 设置新的基指针
5   8:  48 83 ec 40      sub     $0x40,%rsp  # 为局部变量分配
   空间
6  c:  64 48 8b 04 25 28 00 mov     %fs:0x28,%rax # 从线程局部存储读取值
7 13:  00 00
8 15:  48 89 45 f8      mov     %rax,-0x8(%rbp) # 保存线程局部存储的
   值
9 19:  31 c0          xor     %eax,%eax   # 清零eax
10 1b:  66 0f ef c0     pxor    %xmm0,%xmm0 # 清零xmm0
11 1f:  f2 0f 11 45 e0   movsd   %xmm0,-0x20(%rbp) # 将0.0移动到栈上
12 24:  c7 45 d8 10 27 00 00 movl    $0x2710,-0x28(%rbp) # 将10010移动到栈上
13 2b:  48 8d 45 c4      lea     -0x3c(%rbp),%rax # 计算局部变量的地址
14 2f:  48 89 c6        mov     %rax,%rsi   # 将地址移动到rsi
15 32:  48 8d 05 00 00 00 00 lea     0x0(%rip),%rax # 计算函数地址
16 39:  48 89 c7        mov     %rax,%rdi   # 将地址移动到rdi
17 3c:  b8 00 00 00 00   mov     $0x0,%eax   # 清零eax
18 41:  e8 00 00 00 00   call    46 <main+0x46> # 调用函数
19 46:  c7 45 d4 00 00 00 00 movl    $0x0,-0x2c(%rbp) # 将0移动到栈上
20 4d:  eb 76          jmp     c5 <main+0xc5> # 跳转到循环开始
21 4f:  e8 00 00 00 00   call    54 <main+0x54> # 调用函数
22 54:  48 89 45 e8      mov     %rax,-0x18(%rbp) # 保存函数返回值
23 58:  c7 45 c8 00 00 00 00 movl    $0x0,-0x38(%rbp) # 初始化循环变量
24 5f:  c7 45 cc 01 00 00 00 movl    $0x1,-0x34(%rbp) # 初始化循环变量
25 66:  c7 45 d0 01 00 00 00 movl    $0x1,-0x30(%rbp) # 初始化循环变量
26 6d:  eb 16          jmp     85 <main+0x85> # 跳转到循环条件检查
27 6f:  8b 45 cc        mov     -0x34(%rbp),%eax # 加载循环变量
28 72:  89 45 dc        mov     %eax,-0x24(%rbp) # 保存循环变量
29 75:  8b 45 c8        mov     -0x38(%rbp),%eax # 加载循环变量
30 78:  01 45 cc        add     %eax,-0x34(%rbp) # 更新循环变量
31 7b:  8b 45 dc        mov     -0x24(%rbp),%eax # 加载更新后的循环变
   量

```

```

32  7e:  89 45 c8          mov    %eax,-0x38(%rbp) # 保存更新后的循环变
    量
33  81:  83 45 d0 01       addl   $0x1,-0x30(%rbp) # 增加循环计数
34  85:  8b 45 c4          mov    -0x3c(%rbp),%eax # 加载循环条件变量
35  88:  39 45 d0          cmp    %eax,-0x30(%rbp) # 比较循环条件
36  8b:  7c e2          jl     6f <main+0x6f> # 如果小于则跳转
37  8d:  e8 00 00 00 00    call   92 <main+0x92> # 调用函数
38  92:  48 89 45 f0       mov    %rax,-0x10(%rbp) # 保存函数返回值
39  96:  48 8b 45 f0       mov    -0x10(%rbp),%rax # 加载函数返回值
40  9a:  48 2b 45 e8       sub    -0x18(%rbp),%rax # 计算差值
41  9e:  66 0f ef c0       pxor    %xmm0,%xmm0 # 清零xmm0
42  a2:  f2 48 0f 2a c0    cvtsi2sd %rax,%xmm0 # 将差值转换为浮点数
43  a7:  f2 0f 10 0d 00 00 00 movsd  0x0(%rip),%xmm1 # 加载常数到xmm1
44  ae:  f2 0f 5e c1       divsd  %xmm1,%xmm0 # 除以常数
45  b3:  f2 0f 10 4d e0    movsd  -0x20(%rbp),%xmm1 # 加载累加值
46  b8:  f2 0f 58 c1       addsd  %xmm1,%xmm0 # 累加
47  bc:  f2 0f 11 45 e0    movsd  %xmm0,-0x20(%rbp) # 保存累加结果
48  c1:  83 45 d4 01       addl   $0x1,-0x2c(%rbp) # 增加外层循环计数
49  c5:  8b 45 d4          mov    -0x2c(%rbp),%eax # 加载外层循环计数
50  c8:  3b 45 d8          cmp    -0x28(%rbp),%eax # 比较外层循环条件
51  cb:  7c 82          jl     4f <main+0x4f> # 如果小于则跳转
52  cd:  66 0f ef c9       pxor    %xmm1,%xmm1 # 清零xmm1
53  d1:  f2 0f 2a 4d d8    cvtsi2sd %rax,%xmm1 # 将循环次数转换
    为浮点数
54  d6:  f2 0f 10 45 e0    movsd  -0x20(%rbp),%xmm0 # 加载累加结果
55  db:  f2 0f 5e c1       divsd  %xmm1,%xmm0 # 计算平均值
56  df:  66 48 0f 7e c0    movq    %xmm0,%rax # 将平均值转换为整数
57  e4:  66 48 0f 6e c0    movq    %rax,%

```

程序主要完成了对于计算时间的累加取平均的操作。

程序开始时，通过 `push %rbp` 和 `mov %rsp,%rbp` 设置了栈帧，为局部变量分配了空间（`sub $0x40,%rsp`），并保存了线程局部存储的值。

然后，程序进入了一个嵌套循环结构。外层循环使用 `jmp` 和 `jl` 指令控制跳转，内层循环也使用类似的结构。内层循环中，程序执行了一些累加操作。

对于浮点数的运算，程序使用 `cvtsi2sd` 指令将整数转换为双精度浮点数，并使用 `addsd` 和 `divsd` 指令执行了浮点数的加法和除法运算。

另外，程序计算了内层循环中累加值的平均值。它首先将累加值除以循环次数，然后将结果存储在一个浮点寄存器中，最后将计算得到的平均值转换回整数，并调用另一个函数 `printf` 来输出结果。程序通过 `leave` 和 `ret` 指令退出，恢复了栈帧并返回到调用者。

**我们发现，出现了很多二进制为 0 的地址，我们进一步分析，发现类似于跳转地址、变量、分支标签等内容在此处都为 0，call 指令所对应的地址都是紧跟着它的后一条地址，均可以指向下一条指令。**

## （五）链接器

由汇编程序生成的目标文件不能够直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真

正在机器上运行的代码。进而链接器对该机器代码进行执行生成可执行文件。

我们使用命令行 `clang main-llvm.o -o main-llvm` 生成 `main-llvm` 可执行文件；通过命令行 `gcc main-arm.o -o main-arm` 生成 `main-arm` 可执行文件；通过命令行 `gcc main-x86.o -o main-x86` 生成 `main-x86` 可执行文件。

由于篇幅原因，此处仅分析部分链接器生成的代码。

```

1 Disassembly of section .plt:
2
3 0000000000001020 <.plt>:
4   1020:      ff 35 8a 2f 00 00      push    0x2f8a(%rip)          # 3fb0 <
        _GLOBAL_OFFSET_TABLE_+0x8>
5   1026:      f2 ff 25 8b 2f 00 00    bnd jmp  *0x2f8b(%rip)        # 3fb8 <
        _GLOBAL_OFFSET_TABLE_+0x10>
6   102d:      0f 1f 00                nopl    (%rax)
7   1030:      f3 0f 1e fa             endbr64
8   1034:      68 00 00 00 00          push    $0x0
9   1039:      f2 e9 e1 ff ff ff       bnd jmp 1020 <_init+0x20>
10  103f:      90                      nop
11  1040:      f3 0f 1e fa             endbr64
12  1044:      68 01 00 00 00          push    $0x1
13  1049:      f2 e9 d1 ff ff ff       bnd jmp 1020 <_init+0x20>
14  104f:      90                      nop
15  1050:      f3 0f 1e fa             endbr64
16  1054:      68 02 00 00 00          push    $0x2
17  1059:      f2 e9 c1 ff ff ff       bnd jmp 1020 <_init+0x20>
18  105f:      90                      nop

```

.plt（Procedure Linkage Table）是用于动态链接的跳转指令表，包含了从程序代码到动态库中函数实现的间接跳转。

首先，将全局偏移表（GOT）中的一个地址压入栈，成功将函数的地址传递过去。然后跳转到由 `rip` 寄存器相对地址 `0x2f8b` 处的值指定的地址，将 `0` 压入栈中。接着我们跳转回我们的 `.plt` 的开始部分，压入我们的 `1` 和 `2` 这两个值。

**从中我们可以看到，在经过链接器的链接操作后，补上了一些之前的空白地址，实现了链接器的功能——将目标文件和库文件链接到一起。**

## （六）执行程序

我们在前面的过程中，已经生成了对应的可执行文件。我们直接输入命令行 `./main-llvm` 或者 `./main-x86` 进行测试即可，结果如图所示。

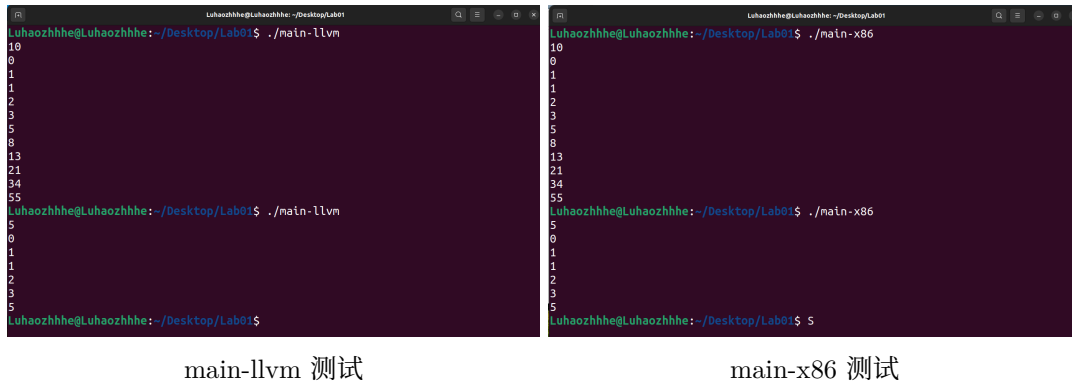


图 18: 执行程序测试

发现我们的程序执行成功了，并且输出的值也是正确的！  
展示一下再第一问的探索中，编写以及生成的文件汇总，如图19所示。

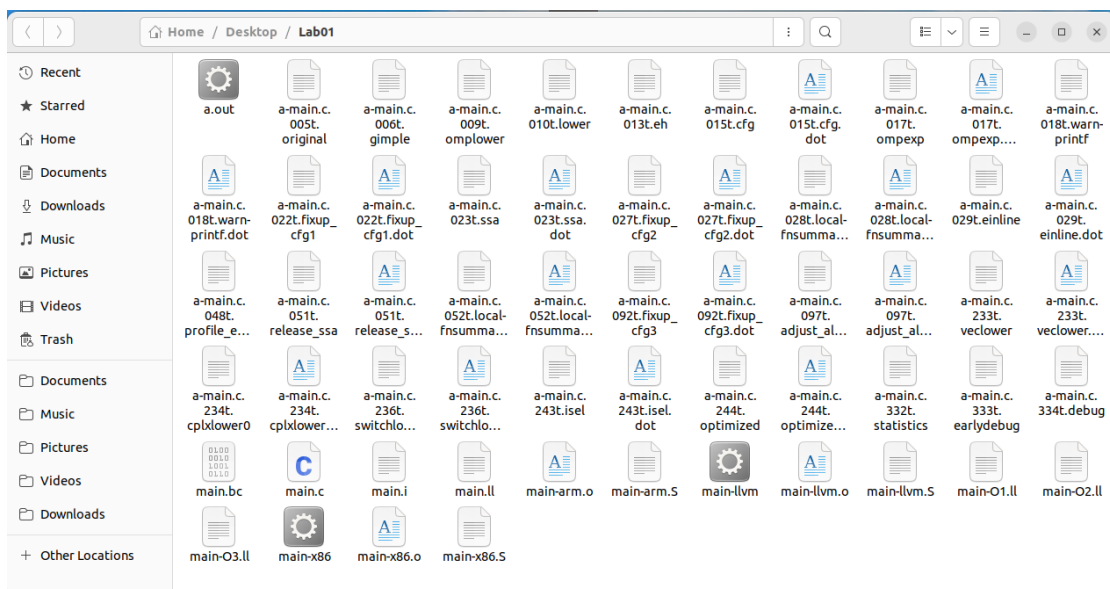


图 19: 程序汇总

### 三、任务二：LLVM IR 编程

### （一）任务内容

熟悉 LLVM IR 中间语言，对你要实现的 SysY 编译器各语言特性，编写 LLVM IR 程序小例子，用 LLVM/Clang 编译成目标程序、执行验证。

## (二) 斐波那契数列程序的编写与改进

源程序如下:

```
1  #include <stdio.h>
2  int main() {
3      int a, b, i, t, n;
```

```

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

```

### 1. 编写

由于对 LLVM IR 的编程方式并不熟悉, 首先使用 clang 产生中间代码的示例, 再比照示例写成基础内容, 数组和浮点数的改进查资料写出。最原始的程序主要分为三部分进行编写: 变量初始化、输入输出、循环递推。

#### 变量初始化

变量初始化对应这部分的代码:

```

1  int a, b, i, t, n;
2  a = 0;
3  b = 1;
4  i = 1;

```

变量初始化包含两个部分: 申请内存和赋初值。申请内存中, 五个变量都是 int 类型, 大小为四个字节, 其在 LLVM IR 中应为 i32 类型, 需要满足四个字节对齐, 因此对应代码:

```

1  %2=alloca i32, align 4 ;a
2  %3=alloca i32, align 4 ;b
3  %4=alloca i32, align 4 ;i
4  %5=alloca i32, align 4 ;t
5  %6=alloca i32, align 4 ;n

```

会给这些临时寄存器在栈上分配内存, 接下来要对 a、b、i 进行赋值:

```

6  store i32 0, i32* %2, align 4 ;a=0
7  store i32 1, i32* %3, align 4 ;b=1
8  store i32 1, i32* %4, align 4 ;i=1

```

#### 输入输出

输入调用 @\_\_isoc99\_scanf 获得输入 (注意要建立一个接受输入的变量, 这里是 %6, 即变量 n, 并且要制定返回类型, 这里为 i32)。

```

1 %7 = call i32 (i8*, ...) @_isoc99_scanf(i8* noundef getelementptr inbounds
   ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %6)
2 #此处%7为返回值临时寄存器，而%6为临时接受值。
3
4 #scanf("%d", &n);

```

这里我们解释一下 scanf 中参数的意义：

call 为调用；

i32 为返回类型，(i8\*, ...) 表示 scanf 采用的参数类型，i8\* 表示第一个参数为一个指向 i8 类型（8 位，对应字符大小，即字符类型，可以理解为 char\*）的指针，... 表示为可变参数函数，可以接受多个参数；

\_\_isoc99\_scanf 是符合 ISO C99 标准的具体函数名；

i8\* noundef getelementptr inbounds ([3 x i8], [3 x i8]\* @.str, i64 0, i64 0) 是计算一个字符指针的位置，该指针指向格式化字符串 ("%d")，noundef 表明必须接受有定义的格式化字符串（即格式化字符串不能为空），getelementptr inbounds ([3 x i8], [3 x i8]\* @.str, i64 0, i64 0) 则是寻址函数，getelementptr 用于计算指针偏移量，inbounds 用于要求边界检查，[3 x i8] 对应三个字符的大小，即"%d" 的大小（不能忘记"\0"），它表示一个三字节的数组，而 [3 x i8]\* @.str 代表格式化字符串的地址，我们通过 getelementptr 计算，而后面的两个 i64 0 则表示数组的起始地址和偏移量，是 getelementptr 的参数。

输出调用 @printf 打印出结果（同样需要声明返回类型、返回值存储的临时变量等，参数含义类似 @scanf）：

```

1 %8 = load i32, i32* %2, align 4
2 %9 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8
   ], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %8)
3 %10 = load i32, i32* %3, align 4
4 %11 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8
   ], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %10)
5 #此处%8、%10为输出的结果，分别从%2、%3，即变量a、b中加载取值，%9、%11为返回值
   临时寄存器
6 #printf("%d\n", a);
7 #printf("%d\n", b);

```

### 循环递推

之后进入 while 循环体，即：

```

1 while (i < n) {
2     t = b;
3     b = a + b;
4     printf("%d\n", b);
5     a = t;
6     i = i + 1;
7 }

```

这里我们需要建立循环条件的比较机制，并在每一个循环体内进行变量的更新、打印的调用。

首先建立循环条件，对于 while 循环，我们可以建立三个标签，分别对应比较条件块、循环体内部块和循环体结束块，第一次进入 while 或者每次循环体结束都会跳转到比较条件块，如果

比较内容成立则进入循环体内部块, 否则跳转至循环体结束块, 在 LLVM IR 中, 我们往往采用临时寄存器当做跳转位置的标签, 比如:

```

1      br label %12
2      # 跳转到标签%12处, 开始循环的比较, 增加一个跳转是为了优化的方便
3  12:
4      # 比较体, 即判断i是否小于n
5      %13 = load i32, i32* %4, align 4
6      %14 = load i32, i32* %6, align 4
7      %15 = icmp slt i32 %13, %14
8      # %13和%14分别load了i和n的值, 而%15则是采用带符号整数比较(icmp)的小于则置位
        (slt) 得到的标志结果, 为真则为1, 否则为0
9      br i1 %15, label %16, label %26
10     # 该句为跳转语句, 如果%15为真则跳转到%16标签处继续循环, 否则跳转到%26标签处
        结束循环。
11  16:
12     # 循环体内部, 即从t=b; 到i=i+1; 的内容, 下部分详细写
13     br label %12
14     # 无条件跳转到标签%12处, 这是while的特点。
15
16  26:
17     # 循环体结束块, 即while后面语句对应的位置, 如果循环结束会跳转到这里

```

接下来我们补齐标签%16 中的内容:

```

1  16:
2      # 循环体内部, 即从t=b; 到i=i+1; 的内容, 下部分详细写
3      %17 = load i32, i32* %3, align 4
4      # %17 = b
5      store i32 %17, i32* %5, align 4
6      # t = %17, 意义在于两个内存数之间不能直接赋值
7      %18 = load i32, i32* %2, align 4
8      # %18 = a
9      %19 = load i32, i32* %3, align 4
10     # %19 = b
11     %20 = add nsw i32 %18, %19
12     # %20 = %18 + %19 = a + b
13     store i32 %20, i32* %3, align 4
14     # b = %20, 综合起来就是计算 b = a + b;
15     %21 = load i32, i32* %3, align 4
16     # %21 = b
17     %22 = call i32 @printf(i8*, ...) @printf(i8* noundef getelementptr @inbounds ([4
        x i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %21)
18     # printf(%21)
19     %23 = load i32, i32* %5, align 4
20     # %23 = t
21     store i32 %23, i32* %2, align 4
22     # a = %23, 综合起来就是a=t
23     %24 = load i32, i32* %4, align 4
24     # %24 = i

```

```

25     %25 = add nsw i32 %24, 1
26     # %25 = %24 + 1
27     store i32 %25, i32* %4, align 4
28     # i = %25, 综合起来就是 i = i + 1;
29     br label %12
30     # 我们可以看到, 在变量赋值过程中, 产生了大量中间结果, 使用临时寄存器存储,
    这是由于指令集的精简性等因素决定的, 在存储在内存的变量间的赋值需要存入
    寄存器再转存入内存, 但在实现编译的意义上来说, 这种代码特性是更方便的。

```

在编写完 main 函数内容之外, 我们还应对 main 函数本身进行编写, 包括其入口、返回值等:

```

1  # 程序入口
2  define i32 @main() #0
3  {
4  # 程序返回
5  ret i32 0
6  }

```

经过以上编写, 拼接起来的代码如下:

```

1  define dso_local i32 @main() #0 {
2      %1 = alloca i32, align 4
3      %2 = alloca i32, align 4
4      %3 = alloca i32, align 4
5      %4 = alloca i32, align 4
6      %5 = alloca i32, align 4
7      %6 = alloca i32, align 4
8      store i32 0, i32* %2, align 4
9      store i32 1, i32* %3, align 4
10     store i32 1, i32* %4, align 4
11     %7 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds
        ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %6)
12     %8 = load i32, i32* %2, align 4
13     %9 = call i32 @printf(i8* noundef getelementptr inbounds ([4 x
        i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %8)
14     %10 = load i32, i32* %3, align 4
15     %11 = call i32 @printf(i8* noundef getelementptr inbounds ([4 x
        i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %10)
16     br label %12
17
18 12:
19     %13 = load i32, i32* %4, align 4
20     %14 = load i32, i32* %6, align 4
21     %15 = icmp slt i32 %13, %14
22     br i1 %15, label %16, label %26
23
24 16:
25     %17 = load i32, i32* %3, align 4

```



```

26     store i32 %17, i32* %5, align 4
27     %18 = load i32, i32* %2, align 4
28     %19 = load i32, i32* %3, align 4
29     %20 = add nsw i32 %18, %19
30     store i32 %20, i32* %3, align 4
31     %21 = load i32, i32* %3, align 4
32     %22 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
        i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %21)
33     %23 = load i32, i32* %5, align 4
34     store i32 %23, i32* %2, align 4
35     %24 = load i32, i32* %4, align 4
36     %25 = add nsw i32 %24, 1
37     store i32 %25, i32* %4, align 4
38     br label %12
39
40 26:
41     ret i32 0
42 }

```

在此之上，对程序中引用的函数进行声明，并对程序中的格式化字符串进行处理，就形成了完整代码：

```

1 declare i32 @__isoc99_scanf(i8* noundef, ...) #1
2 declare i32 @printf(i8* noundef, ...) #1
3 @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
4 @.str.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

```

## 2. 数组和浮点数改进

在实现简单的 LLVM IR 编程的基础上，我们改进前面的斐波那契数列程序，包括两方面：使用数组存储中间结果和使用浮点数数据结构。使用数组可以记录更多的数列结果，缺点是空间复杂度增加，使用浮点数可以提高表示变量的范围，能够增加计算范围。

首先是源程序：

```

1 int main() {
2     double a, b, i, t, n;
3     scanf("%lf", &n);
4     double fib[n + 1];
5     fib[0] = 0;
6     if (n > 0) {
7         fib[1] = 1;
8     }
9     for (i = 2; i <= n; i++) {
10        fib[i] = fib[i - 1] + fib[i - 2];
11    }
12    for (i = 0; i <= n; i++) {
13        printf("%lf\n", fib[i]);
14    }

```

```

15     return 0;
16 }

```

只添加了一个 double 类型的 fib 数组，其实前面的变量 a、b 和 t 都可以删掉，但为了方便修改（虚拟寄存器的编号要求比较复杂），在编写的过程中并没有真的删去。

这里我们需要注意的有以下几点：数组空间的申请，数组地址的计算，循环的处理：

首先是数组空间的申请，由于是 double 类型数组，每个 double 类型的变量都应该占 8 个字节，由于 fib 数组应该有  $n+1$  项，应该申请  $8*(n+1)$  字节的空间，而 LLVM IR 中的 malloc 是以 i8（八位，对应一字节）为单位申请的，应该先计算空间的大小，再转化成 i8 单位下的空间大小，进行申请：

```

1 %5 = alloca double* # fib
2 #calculate space of array((n+1)*double)
3 %6 = call i32 @i8*, ... @scanf(i8* getelementptr ([3 x i8], [3 x i8]* @.str,
4     i32 0, i32 0), i32* %4)
5 %7 = load i32, i32* %4
6 %8 = add i32 %7, 1#fib数组从0开始到n，共n+1个double的位置
7 %9 = sext i32 %8 to i64#malloc的参数需要是i64
8 %10 = mul i64 %9, 8#一个double对应八个字节
9 %11 = call noalias i8* @malloc(i64 %10)#申请内存，参数为字节数
10 # malloc返回值为i8*的指针，类似free
11 %12 = bitcast i8* %11 to double*
12 store double* %12, double** %5#fib首地址

```

之后是数组地址的计算，我们以调用 fib[i-1] 的值为例说明，i-1 是一个 i32 类型的偏移值，而 fib 是一个 double 数组，我们可以通过 getelementptr 计算：

```

1 %26 = load double*, double** %5#array柄
2 %27 = load i32, i32* %2#i
3 %28 = sub i32 %27, 1#i-1
4 %29 = getelementptr double, double* %26, i32 %28#%28是变址，存的值按i32识别
5 %30 = load double, double* %29#fib[i-1]

```

fib[i-2] 等地址的计算同理，可以实现数组元素的读取和计算结果存入数组。接下来循环的处理，与 while 循环不同，for 循环包括四个部分，循环量初始化、循环条件比较、循环体执行与循环量更新，以及跳出位置。

以斐波那契数列计算的递推过程为例，具体内容省略，框架如下：

```

1 20:
2     #init
3     store i32 2, i32* %2#i=2
4     br label %21
5 21:
6     #compare look condition
7     %%24=bool(i<n)
8     br i1 %24, label %25, label %38
9 25:
10    #loopbody and update i

```

```

11     #fib[i]=fib[i-1]+fib[i-2]# i+=1
12     br label %21
13 38:
14     #end, other codes

```

综合起来形成如下的代码:

```

1  define i32 @main(){
2      #alloca m
3      %1 = alloca i32#b
4      %2 = alloca i32#i
5      %3 = alloca i32#t
6      %4 = alloca i32#n
7      %5 = alloca double*#fib
8      #calculate space of array(n*double)
9      %6 = call i32 (i8*, ...) @scanf(i8* getelementptr ([3 x i8], [3 x i8]*
10         @.str, i32 0, i32 0), i32* %4)
11
12      %7 = load i32, i32* %4
13      %8 = add i32 %7, 1#fib数组从0开始到n, 共n+1个double的位置
14      %9 = sext i32 %8 to i64#malloc的参数需要是i64
15      %10 = mul i64 %9, 8#一个double对应八个字节
16      %11 = call noalias i8* @malloc(i64 %10)#申请内存, 参数为字节数
17      #malloc返回值为i8*的指针, 类似free
18      %12 = bitcast i8* %11 to double*
19      store double* %12, double** %5#fib首地址
20
21      #init fib[0], fib[1]
22      %13 = load double*, double** %5#指向数组地址的指针, 因此是double**, 也就是
23         说, (%13)*=array, 后文我们使用OpaquePointers优化, 可以有效地减少这种思
24         考量
25      %14 = getelementptr double, double* %13, i32 0#计算从%13初始位置 (i32 0)
26         开始的地址
27      store double 0.0, double* %14
28
29      #下面要判断if, 会产生分支label, 分三部分 比较体/真体/后体
30      #if(n>0)
31      %15 = load i32, i32* %4
32      %16 = icmp sgt i32 %15, 0#大于则置位, 产生一位结果(i1)
33      br i1 %16, label %17, label %20#最后改为寄存器号
34 17:
35      #f[1]=1.0
36      %18 = load double*, double** %5#起始地址
37      %19 = getelementptr double, double* %18, i32 1
38      store double 1.0, double* %19
39      br label %20
40 20:
41      #for:compare,loopbody,end
42      store i32 2, i32* %2#i=2

```

```

39     br label %21
40 21:
41     %22 = load i32, i32* %2#i
42     %23 = load i32, i32* %4#n
43     %24 = icmp sle i32 %22, %23#小于等于则置位
44     br i1 %24, label %25, label %38
45 25:
46     #先计算位置, 再将值取出, 再赋值
47     %26 = load double*, double** %5#array柄
48     %27 = load i32, i32* %2#i
49     %28 = sub i32 %27, 1#i-1
50     %29 = getelementptr double, double* %26, i32 %28#%28是变址, 存的值按i32识
        别
51     %30 = load double, double* %29#fib[i-1]
52     %31 = sub i32 %27, 2#i-2
53     %32 = getelementptr double, double* %26, i32 %31
54     %33 = load double, double* %32#fib[i-2]
55     %34 = fadd double %30, %33#fib[i-1]+fib[i-2]
56     %35 = getelementptr double, double* %26, i32 %27#(fib+i)
57     store double %34, double* %35
58     %36 = load i32, i32* %2
59     %37 = add i32 %36, 1
60     store i32 %37, i32* %2
61     br label %21
62 38:
63     #output for also compare, loopbody, end
64     store i32 0, i32* %2#i=0
65     br label %39
66 39:
67     %40 = load i32, i32* %2#i
68     %41 = load i32, i32* %4#n
69     %42 = icmp sle i32 %40, %41#i<=n
70     br i1 %42, label %43, label %51
71 43:
72     #output
73     %44 = load double*, double** %5
74     %45 = load i32, i32* %2
75     %46 = getelementptr double, double* %44, i32 %45
76     %47 = load double, double* %46
77     %48 = call i32 (i8*, ...) @printf(i8* getelementptr ([5 x i8], [5 x i8]*
        @.str1, i32 0, i32 0), double %47)
78     #i++
79     %49 = load i32, i32* %2
80     %50 = add i32 %49, 1
81     store i32 %50, i32* %2
82     br label %39
83 51:
84     %52 = load double*, double** %5

```

```

85     %53 = bitcast double* %52 to i8*
86     call void @free(i8* %53)
87     ret i32 0
88 }
89
90 declare i32 @scanf(i8*, ...)
91 declare i32 @printf(i8*, ...)
92 declare noalias i8* @malloc(i64)
93 declare void @free(i8*)
94 @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
95 @.str1 = private unnamed_addr constant [5 x i8] c"%lf\0A\00", align 1
96 #声明引用的函数和其中的格式化字符串

```

### 3. 指针优化

在 15.0 以上版本的 clang 对应的 LLVMIR 支持 OpaquePointers 特性, 可以简化我们实现数组和指针的难度, 这种特性支持我们直接使用 ptr 而非复杂的目标类型来声明指针, 在 load、getelementptr 等需要计算指针大小时, 在前面加上命令的类型声明即可, 比如:

```

1 #不优化
2 %7 = load i32, i32* %4
3 #优化
4 %7 = load i32, ptr %4#ptr根据load i32自动识别为一个i32指针

```

类似地, 我们将全部类似情况进行转换:

```

1 define i32 @main(){
2     #alloca m
3     %1 = alloca i32#b
4     %2 = alloca i32#i
5     %3 = alloca i32#t
6     %4 = alloca i32#n
7     %5 = alloca ptr #fib
8     #calculate space of array(n*double)
9     %6 = call i32 (i8*, ...) @scanf(i8* getelementptr ([3 x i8], [3 x i8]*
10         @.str, i32 0, i32 0), i32* %4)
11
12     %7 = load i32, ptr %4
13     %8 = add i32 %7, 1#fib数组从0开始到n, 共n+1个double的位置
14     %9 = sext i32 %8 to i64#malloc的参数需要是i64
15     %10 = mul i64 %9, 8#一个double对应八个字节
16     %11 = call noalias i8* @malloc(i64 %10)#申请内存, 参数为字节数
17     # malloc返回值为i8*的指针, 类似free
18     %12 = bitcast i8* %11 to double*
19     store double* %12, ptr %5#fib首地址
20
21     #init fib[0], fib[1]
22     %13 = load double*, ptr %5#指向数组地址的指针, 因此是double**, 也就是说,
23         (%13)*=array

```

```

22  %14 = getelementptr double, ptr %13, i32 0# 计算从%13初始位置 (i32 0) 开始
    的地址
23  store double 0.0, ptr %14
24
25  #下面要判断if, 会产生分支label, 分三部分 比较体/真体/后体
26  #if(n>0)
27  %15 = load i32, ptr %4
28  %16 = icmp sgt i32 %15, 0#大于则置位, 产生一位结果(i1)
29  br i1 %16, label %17, label %20#最后改为寄存器号
30 17:
31  #f[1]=1.0
32  %18 = load double*, ptr %5#起始地址
33  %19 = getelementptr double, ptr %18, i32 1
34  store double 1.0, ptr %19
35  br label %20
36 20:
37  #for:compare,loopbody,end
38  store i32 2, ptr %2#i=2
39  br label %21
40 21:
41  %22 = load i32, ptr %2#i
42  %23 = load i32, ptr %4#n
43  %24 = icmp sle i32 %22, %23#小于等于则置位
44  br i1 %24, label %25, label %38
45 25:
46  #先计算位置, 再将值取出, 再赋值
47  %26 = load double*, ptr %5#array柄
48  %27 = load i32, ptr %2#i
49  %28 = sub i32 %27, 1#i-1
50  %29 = getelementptr double, ptr %26, i32 %28#%28是变址, 存的值按i32识别
51  %30 = load double, ptr %29#fib[i-1]
52  %31 = sub i32 %27, 2#i-2
53  %32 = getelementptr double, ptr %26, i32 %31
54  %33 = load double, ptr %32#fib[i-2]
55  %34 = fadd double %30, %33#fib[i-1]+fib[i-2]
56  %35 = getelementptr double, ptr %26, i32 %27#(fib+i)
57  store double %34, ptr %35
58  %36 = load i32, ptr %2
59  %37 = add i32 %36, 1
60  store i32 %37, ptr %2
61  br label %21
62 38:
63  #output for also compare, loopbody, end
64  store i32 0, ptr %2#i=0
65  br label %39
66 39:
67  %40 = load i32, ptr %2#i
68  %41 = load i32, ptr %4#n

```

```

69     %42 = icmp sle i32 %40, %41#i<=n
70     br i1 %42, label %43, label %51
71 43:
72     #output
73     %44 = load double*, ptr %5
74     %45 = load i32, ptr %2
75     %46 = getelementptr double, ptr %44, i32 %45
76     %47 = load double, ptr %46
77     %48 = call i32 (i8*, ...) @printf(i8* getelementptr ([5 x i8], [5 x i8]*
        @.str1, i32 0, i32 0), double %47)
78     #i++
79     %49 = load i32, ptr %2
80     %50 = add i32 %49, 1
81     store i32 %50, ptr %2
82     br label %39
83 51:
84     %52 = load double*, ptr %5
85     %53 = bitcast ptr %52 to i8*
86     call void @free(i8* %53)
87     ret i32 0
88 }
89
90 declare i32 @scanf(i8*, ...)
91 declare i32 @printf(i8*, ...)
92 declare noalias i8* @malloc(i64)
93 declare void @free(i8*)
94
95 @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
96 @.str1 = private unnamed_addr constant [5 x i8] c"%lf\0A\00", align 1
97 #声明引用的函数和其中的格式化字符串

```

这是我们最终所使用的程序。

#### 4. 验证

我们对未进行指针优化的程序进行编译和验证，如图20所示。

```
hao@hao-virtual-machine: ~/Documents/InfoSecurity/Repos...
hao@hao-virtual-machine:~/Documents/InfoSecurity/Repository/Compile/Principles_of_Compile/Principles_of_Compile_Systems2024/Lab_pre/Lab_pre_part_2/final_version$ llvm-as f_double.ll -o f_double.bc
hao@hao-virtual-machine:~/Documents/InfoSecurity/Repository/Compile/Principles_of_Compile/Principles_of_Compile_Systems2024/Lab_pre/Lab_pre_part_2/final_version$ llc f_double.bc -o f_double.s
hao@hao-virtual-machine:~/Documents/InfoSecurity/Repository/Compile/Principles_of_Compile/Principles_of_Compile_Systems2024/Lab_pre/Lab_pre_part_2/final_version$ clang-15 -g f_double.s -o f_double -no-pie
hao@hao-virtual-machine:~/Documents/InfoSecurity/Repository/Compile/Principles_of_Compile/Principles_of_Compile_Systems2024/Lab_pre/Lab_pre_part_2/final_version$ ./f_double
8
0.000000
1.000000
1.000000
2.000000
3.000000
5.000000
8.000000
13.000000
21.000000
hao@hao-virtual-machine:~/Documents/InfoSecurity/Repository/Compile/Principles_of_Compile/Principles_of_Compile_Systems2024/Lab_pre/Lab_pre_part_2/final_version$
```

图 20: 改进后的编译与验证

发现程序成功输出了斐波那契数列且为浮点数格式 (如图21所示)。下面我们对进行过指针优化的程序进行编译验证:

```
hao@hao-virtual-machine:~/Documents/InfoSecurity/Repository/Compile/Principles_of_Compile/Principles_of_Compile_Systems2024/Lab_pre/Lab_pre_part_2/FFinal_version$ clang-15 -Xclang -opaque-pointers f.ll -o f
warning: overriding the module target triple with x86_64-pc-linux-gnu [-Woverride-module]
1 warning generated.
hao@hao-virtual-machine:~/Documents/InfoSecurity/Repository/Compile/Principles_of_Compile/Principles_of_Compile_Systems2024/Lab_pre/Lab_pre_part_2/FFinal_version$ '/home/hao/Documents/InfoSecurity/Repository/Compile/Principles_of_Compile/Principles_of_Compile_Systems2024/Lab_pre/Lab_pre_part_2/FFinal_version/f'
5
0.000000
1.000000
1.000000
2.000000
3.000000
5.000000
```

图 21: 指针优化后的程序验证

发现同样能输出正确的结果, 并且这种中间语言实现起来更为方便。

## 四、 任务三：汇编编程

### (一) 程序 1: 斐波那契数列

#### 1. SysY 源程序

我们在前面部分中已经设计过斐波那契数列的相关程序, 因此不再详细解释, 直接给出我设计的 SysY 源程序。

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



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

## 2. 汇编程序编写思路

关于这个程序，我是对照着 SysY 源程序进行编写的，主要思路如下：

首先我们需要确定程序需要做什么，毋庸置疑需要求出斐波那契数列的第  $n$  项，并且需要将前  $n$  项全部打印输出。我们先定义程序所需要的全局变量  $a$ 、 $b$  和  $n$ ，并且给  $a$  和  $b$  赋初值。然后需要设置函数的入口点，于是我们定义了一个 `main` 函数。接着我们需要读取我们输入的  $n$  的值，我准备用 `scanf` 函数来进行读取。

然后就是首先需要输出前两项的值，这个不难，直接输出一下就行了。关键部分就是后面的循环部分，我根据大二上学期《汇编语言与逆向技术》中的知识，准备使用**条件分支指令** `blt` 来进行循环的控制，这样就可以实现 `while` 循环了。

有关程序栈帧问题的分配，我参考了一些成熟的程序，在开始计算时，分配给程序一定大小的栈帧空间，并在结束计算后将其收回，这样可以保证计算的正确性，还可以充分利用对应的空间。

另外，有关性能优化，我在代码中还使用了一系列的 `align` 语句进行数据和代码在内存中对齐，这样可以充分提高我们的代码运算性能，还可以很好地管理我们的内存空间。根据上网搜索可得，`.align 2` 用于 `.data` 段，确保 `word` 定义的变量在 2 的幂次方字节边界上对齐，这通常是 32 位或 64 位处理器的自然对齐边界；`.align 3` 用于 `.rodata.str1.8` 段，确保字符串和其他只读数据在 8 字节边界上对齐，这有助于提高加载这些数据的指令的效率。

## 3. 汇编程序分析

下面我们分块给出设计的对应汇编程序，并做详细解释。

此处涉及到的汇编知识主要是：如何合理的使用栈区的位置，如何编写汇编中的 `while` 循环函数，如何声明一些全局变量等等。

```
1 • .file "Finbonacci.c"
```

该汇编语句声明了源文件的名字为 `Finbonacci.c`。

```

1 • .option nopic
2 .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
3 .attribute unaligned_access, 0
4 .attribute stack_align, 16

```

主要声明了在程序中不使用位置无关的代码，指定了本程序使用的架构为 RISC-V-64，禁止未对齐的访问，以及指定了栈对齐的位数为 16 字节。

```

1 • .globl a # 定义全局变量 a
2 .globl b # 定义全局变量 b
3 .section .data # 开始数据段
4 .align 2 #data 段将字节对齐到 2 的倍数，性能最佳
5 a:
6     .word 0 # 给 a 赋初值 0
7 b:
8     .word 1 # 给 b 赋初值 1

```

声明了全局变量 a 和 b，然后开始数据段，首先将字节地址对齐到 2 的倍数，然后声明 a 和 b 的初始值，分别为 0 和 1。

```

1 • .section .rodata.str1.8,"aMS",@progbits,1 # 只读数据段的声明
2 .align 3 # 该数据段对齐到 8 字节位置最佳
3 .LC0:
4 .string "%d" # 定义字符串常量
5 .align 3
6 .LC1:
7 .string "%d\n" # 定义字符串常量

```

然后开始我们的只读数据段，表示我们不会对其进行数据上的修改。首先将字节地址对齐到 8 的倍数，然后.LC0 和.LC1 分别定义了字符串常量，用于我们的格式化输出。

```

1 • .section .text.startup,"ax",@progbits # 定义代码段
2 .align 1 # 代码段不需要对齐，因此对齐到 1 的倍数即可
3 .globl main # 定义全局变量 main
4 .type main, @function # 声明为全局函数

```

这一段表示我们的代码段开始了，首先将字节地址对齐到 1 的倍数，然后声明 main 全局函数。

```

1 • main:
2     addi sp,sp,-32 # 给局部变量和函数调用分配空间
3     sd s0,16(sp)
4     sd s1,8(sp) # 分别分配 s0、s1 寄存器的栈帧所在位置
5     lui s0,%hi(n)
6     lui a0,%hi(.LC0)

```

```

7  addi a1,s0,%lo(n)
8  addi a0,a0,%lo(.LC0)
9  # 这四行汇编用于将.LC0 的结果存储在 a0 寄存器中，将 n 的结果存储在 s0 寄存器中
10 sd ra,24(sp) # 用于正确找到返回地址
11 call __isoc99_scanf # 调用 scanf 函数，进行格式化输入

```

现在来到了 main 函数的关键部分。首先，将 sp 的地址减去 32，相当于是将 sp 栈帧网上移动了 32 个字节，为局部变量和函数调用的参数在栈上分配空间；然后将寄存器 s0 的值保存到栈指针 sp 指向的地址加上 16 个字节的位置。同理将寄存器 s1 的值保存到栈指针 sp 指向的地址加上 8 个字节的位置。lui 指令用于将全局变量 n 的高 20 位地址加载到寄存器 s0 中，同理后面一条指令是将字符串.LC0 的高 20 位地址加载到寄存器 a0 中。然后将变量 n 的低 12 位地址添加到寄存器 s0 的值中，并将结果存储在寄存器 a1 中，将字符串.LC0 的低 12 位地址添加到寄存器 a0 的值中，并将结果存储回寄存器 a0。这样，a0 就包含了完整的字符串.LC0 的地址。最后，将返回地址寄存器 ra 的值保存到栈指针 sp 指向的地址加上 24 个字节的位置。这样我们就能正确找到返回地址了。最后一步就是调用 C 标准库函数 \_\_isoc99\_scanf，该函数用于从标准输入读取格式化的输入。

```

1 • #output a
2  lui a0,%hi(.LC1)
3  addi a0,a0,%lo(.LC1) # 将.LC1 的值赋值到 a0 寄存器中
4  lui s2,%hi(a)
5  addi a1,s2,%lo(a) # 同理，将 a 的值赋值给 a1 寄存器
6  lw a1,0(a1) # 将 a1 指向的地址加载一个字到寄存器 a1 中
7  call printf # 调用打印函数完成输出

```

这一段就是用于输出我们的 a 的值，首先我们将字符串.LC1 的高 20 位地址加载到寄存器 a0 中，然后将字符串.LC1 的低 12 位地址添加到寄存器 a0 的值中，并将结果存储回寄存器 a0。这样我们就完成了 a0 的赋值。然后后面就是同样的操作，我们分开将我们的 a 导入到 a1 中，a1 就包含了完整的 a。然后我们从寄存器 a1 指向的地址（即变量 a 的地址）加载一个字（32 位）到寄存器 a1 中，最后调用 printf 函数即可完成对 a 的输出。

```

1 • #output b
2  lui a0,%hi(.LC1)
3  addi a0,a0,%lo(.LC1) # 将.LC1 的值赋值到 a0 寄存器中
4  lui s3,%hi(b)
5  addi a1,s3,%lo(b) # 同理，将 b 的值赋值给 a1 寄存器
6  lw a1,0(a1) # 将 a1 指向的地址加载一个字到寄存器 a1 中
7  call printf # 调用打印函数完成输出

```

这一段和上面输出 a 是基本相似的，就是将 a 换成了 b，此处不再赘述。

```

1 • lui s0,%hi(a)
2  addi s0,s0,%lo(a)
3  lw s1,0(s0) # 将 a 的值加载到 s0 寄存器中，然后读取一个字给 s1
4

```

```

5  lui a1,%hi(b)
6  addi a1,a1,%lo(b)
7  lw s2,0(a1) # 将 b 的值加载到 a1 寄存器中，然后读取一个字给 s2
8
9  lui a0,%hi(n)
10 addi a0,a0,%lo(n)
11 lw s4,0(a0) # 将 n 的值加载到 a0 寄存器中
12 li s0,1 # 初始化 s0 寄存器，用于下一轮循环

```

这三段汇编代码很简单，就是将 a、b 和 n 的值加载到我们的寄存器中去。跟前面是一样的方法，首先将 a 的 32 位都存储到寄存器 s0 中，然后从寄存器 s0 指向的地址（即变量 a 的地址）加载一个字（32 位）到寄存器 s1 中。这样我们就完成了对 a 的加载。同理，b 和 a 的加载方式是一模一样的。然后就是 n 的加载，跟 a 和 b 相同，但是在加载之后，我们还需要将 s0 赋值为 1，作用是初始化我们的 s0 寄存器，以便下一次的加载。

```

1 • .L1:
2     mv s3,s2 # 将寄存器 s2 的值赋值给 s3
3     add s2,s2,s1 # 将 s1 的值与 s2 相加，赋值给 s2
4     lui a0,%hi(.LC1)
5     addi a0,a0,%lo(.LC1) #.LC1 赋值到 a0 寄存器上
6     mv a1,s2 # 将 s2 寄存器的值赋值给 a1 寄存器
7     call printf # 调用 printf 函数完成输出
8     mv s1,s3 # 将 s3 的值赋值给 s1，用于下一轮循环
9     addi s0,s0,1 # s0 用于程序的循环计数
10    blt s0,s4,.L1 #关键步骤，用于比较循环次数是否已经达到n的值，
    ↪ 如果达到了就退出循环，如果没有达到就继续循环

```

这一段是汇编中的核心部分，主要用于完成我们的**循环运算操作**。首先，我们将寄存器 s2 的值赋值给 s3，然后将 s1 的值与 s2 相加，赋值给 s2。其中，s1 和 s2 分别保存着斐波那契数列中的连续两个数，用于完成加法操作。此处已经完成了将加法操作后的值赋值给 s2 寄存器。然后后面两行汇编的含义也跟前面的是一样的，就是完成对格式化字符串.LC1 的赋值，将其赋值到了 a0 寄存器上。下一行的作用是，将运算后的 s2 寄存器的值赋值给 a1 寄存器，然后调用 printf 函数将其输出。输出后，我们将运算后的结果 s3 继续赋值给 s1 寄存器，当作是下一轮的运算数，同时将我们的计数寄存器 s0 加上 1，表示已经过了一轮循环。最后使用 blt，对 s0 和 s4 进行比较，如果寄存器 s0 的值（循环计数器）小于寄存器 s4 的值（变量 n 的值），则跳转回标签.L1，继续执行循环。这也完成了我们的循环分支，在循环 n 轮之后，就会跳出循环。

```

1 • ld ra,24(sp)
2   ld s0,16(sp)
3   ld s1,8(sp) # 这三行都用于回复寄存器的状态
4   li a0,0 # 将立即数 0 加载到寄存器 a0 中，确保完成 main 函数的返回
5   addi sp,sp,32 # 恢复原先设置的地址空间
6   jr ra # 跳转到 ra 寄存器的地址，用于返回调用 main 函数的寄存器上
7   .size main,.-main # 标记 main 函数所占空间大小，完成清理

```

该部分用于 main 函数结束后恢复寄存器的状态，设置返回值以及释放我们之前设置的栈空间。首先，从栈指针 sp 指向的地址加上 24 个字节的位置加载一个字（64 位）到寄存器 ra 中。从栈指针 sp 指向的地址加上 16 个字节的位置加载一个字到寄存器 s0 中，用于恢复之前保存的寄存器 s0 的值。从栈指针 sp 指向的地址加上 8 个字节的位置加载一个字到寄存器 s1 中，用于恢复之前保存的寄存器 s1 的值。然后将立即数 0 加载到寄存器 a0 中，这表示 main 函数的返回值为 0，表示我们的程序可以正常退出。addi 操作完成了对栈空间的释放（在程序最开始的时候我们曾经分配了 32 个字节空间的内存）。然后我们跳转到寄存器 ra 存储的地址，即返回到 main 函数的调用者寄存器上，最后标记一下我们的 main 函数的所占空间大小，完成了 main 函数的最后的清理过程。

```

1 • .globl n # 申明全局变量 n
2 .section .sbss,"aw",@nobits # 申明虚拟段，用于链接时的符号解析
3 .align 2 # 虚拟段需要使用 2 字节进行对齐
4 .type n, @object # 为 n 变量申明类型
5 .size b, 4 # 指定 b 的大小为 4 字节
6 n:
7     .zero 4 # 给 n 分配 4 字节的大小空间
8
9 .ident "GCC:()12.2.0" # 申明 GCC 编译版本
10 .section .note.GNU-stack,"",@progbits # 申明栈保护的功能

```

这一段汇编代码首先申明了全局变量 n，然后申明了一个特殊的虚拟段，用于链接时的符号解析。然后首先使数据在内存中按照 2 的倍数对齐，然后为 n 变量申明类型，指定 b 的大小为 4 个字节，给 n 分配 4 个字节的空间，并且将其赋值为 0(初始化)。后面的就是申明一下 GCC 编译的版本，以及栈保护的一些功能的申明。

#### 4. gcc 验证

我们编写完 Finbonacci.s 汇编程序后，用 riscv64-unknown-linux-gnu-gcc 进行验证。输入以下命令行：

```

1 riscv64-unknown-linux-gnu-gcc Finbonacci.s -o Finbonacci -static
2 qemu-riscv64 Finbonacci

```

观察到程序成功编译了，输入 n=10，运算结果如图22所示，说明我们的程序编写成功。

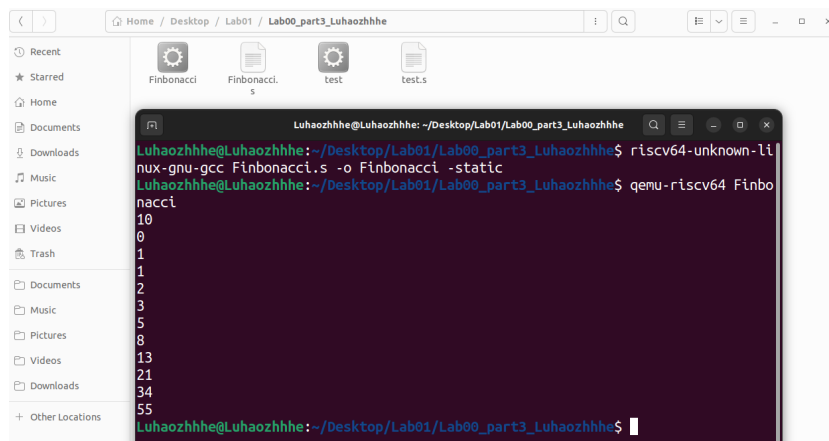


图 22: 测试斐波那契汇编程序

## 5. 一些思考

我编写完程序后，发现该程序其实有很多的不足之处。比如说有关于  $n$  的输入读取，我们没有控制  $n$  的范围。如果输入一个负数或者是一个浮点数，这样可能就会导致程序不能正常输出结果，我们应该在程序中加入一些错误反馈机制，比如输出“Wrong!”等等。

另外，我们使用内存对齐的方式来进行内存的优化，该程序中的 `align` 是我们根据本程序量身打造的，可能不适用于其他的一些程序。有关于内存优化这一方面的内容，我还需要去进一步了解。

有关于子函数调用的问题我也有所思考。比如说在汇编程序中，我需要调用一个子函数，那么我首先需要给他开一个合适大小的空间进行变量等内容的存放。但是我们运行完该子函数后，应该怎么返回原函数呢？我们在此处需要用栈的功能来实现。我们在调用子函数之前，需要保存我们的原来的寄存器的状态；在子函数调用完毕后，还需要恢复那些寄存器的状态。另外，如果子函数有特定的返回值的话，就需要我们申明特殊的寄存器去进行传输。

有关于数组的编写，我也有所思考。数组和我们正常的成员变量不太一样，由于一个数组可以存放很多内容，所以我们在申明数组的时候也需要开一定的空间进行变量等内容的存放。

## (二) 程序 2：浮点数

### 1. SysY 源程序

为了验证浮点数在 SysY 源程序中的应用，我设计了一个有关于比较输入的两个浮点数的大小关系的程序，其中涉及到了浮点数的运算，由于 SysY 程序与 C 程序类似，所以此处不再赘述编写的过程，直接附上源程序代码：

```
1  #include <stdio.h>
2  int main() {
3      float num1, num2;
4      printf("please input the first float number:");
5      scanf("%f", &num1);
6      printf("please input the second float number: ");
7      scanf("%f", &num2);
8
9      if (num1 > num2) {
```



```

10
11     printf("%.2f is bigger than %.2f\n", num1, num2);
12 }
13 else {
14
15     printf("%.2f is not bigger than %.2f\n", num1, num2);
16 }
17
18     return 0;
19 }

```

## 2. 汇编程序编写思路

首先，我们确定，该程序需要完成的任务是，输入两个浮点数，并且比较这两个浮点数的大小关系，如果第一个比第二个大的话，就输出 bigger than，反之输出 not bigger than。

我们先来定义本程序所需要的变量。首先我们需要一些字符串，我们用 LC0、LC1 等来表示这些字符串。然后我们需要一些寄存器来完成相应的运算，这里就不再提及。还涉及到一些函数，比如打印、读取输入等，都在汇编中一一实现了。

有关栈帧分配的问题，我们还是和上面的程序分配方式一样，不做改变。

有关内存优化上的问题，我们也根据前一个程序中的规则来编写。

重点部分就是浮点数运算的实现，此处我打算使用 `fcvt.d.s` 等语句来实现，将我们输入的浮点数进行转换之后，就可以进行比较运算了。至于我们的条件判断分支，我们就使用 `beq` 判断语句来实现，后续会详细说明实现方式。

## 3. 汇编程序分析

下面我们分块给出设计的对应汇编程序，并做详细解释。

此处涉及到的汇编知识主要是：如何合理的使用栈区的位置，如何编写汇编中的 if 条件分支函数，如何声明一些全局变量、如何使用浮点数进行运算、如何输出浮点数等等。

```

1 • .file "float.c" # 源代码文件名为 "float.c"
2   .option nopic # 不生成位置无关代码
3   .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
   ↪ #指定RISC-V 64位架构，包括整数、乘法、原子操作、浮点、双精度浮点、
   ↪ 压缩指令等扩展
4   .attribute unaligned_access, 0 # 指定了对齐访问的属性
5   .attribute stack_align, 16 # 定义了栈的对齐方式，应该按照 16 字节对齐。
6   .text # 代码段
7   .section .rodata.str1.8,"aMS",@progbits,1 # 定义一个只读数据段
8   .align 3 # 规定按照 8 字节进行对齐

```

该文件的名称为 float.c，且不生成位置无关代码。接着，规定了目标架构的属性为 RISC-V-64，包括整数、乘法、原子操作、浮点、双精度浮点、压缩指令等扩展。后面的内容和前面大致是一致的，指定了对齐访问的属性，定义了栈的对齐方式，应该按照 16 字节进行对齐。text 表示开始一段新的代码段，下面的 section 定义的是一个只读数据段，用于存储字符串常量。其中对齐方式按照 8 字节来进行规定。

```

1 • # 都是一些后续会用到的字符串
2 .LC0:
3     .string "please input the first float number: "
4     .align 3
5
6 .LC1:
7     .string "%f"
8     .align 3
9
10 .LC2:
11     .string "please input the second float number: "
12     .align 3
13
14 .LC3:
15     .string "%.2f is bigger than %.2f\n"
16     .align 3
17
18 .LC4:
19     .string "%.2f is not bigger than %.2f\n"
20     .section .text.startup,"ax",@progbits
21     .align 1
22     .globl main # 定义 main 全局函数
23     .type main, @function

```

.LC0 到.LC4 这五个段都表示了一个固定的字符串，我们在后续的程序中可以调用这些字符串进行输出。前面几个都没有什么可以说明的，最后一个表示的是定义了 main 全局函数，其他的内存对齐都是按照 8 字节进行对齐，而 main 函数则不需要进行对齐。

```

1 • main:
2     lui a0,%hi(.LC0) # 将.LC0 高 20 位地址中的内容加载到我们的 a0 寄存器
3     addi sp,sp,-32 # 调整栈帧的位置，分配运算空间
4     addi a0,a0,%lo(.LC0) # 将.LC0 完整放入 a0 寄存器中
5     sd ra,24(sp) # 完成对 ra 寄存器的保存
6     sd s0,16(sp) # 完成对 s0 寄存器的保存
7     call printf # 调用 printf 函数
8     lui s0,%hi(.LC1) #同理，将.LC1的高20位地址中的内存加载到我们的s0寄存器
9     addi a1,sp,8 # 为 a1 寄存器分配空间
10    addi a0,s0,%lo(.LC1) # 将.LC1 的低 12 位放入寄存器 a0，凑成完整的.LC1
11    call __isoc99_scanf # 调用 scanf 函数，读取输入内容
12    lui a0,%hi(.LC2) # 将.LC2 的低位内容放入 a0 寄存器
13    addi a0,a0,%lo(.LC2) # 将.LC2 完整放入寄存器中
14    call printf # 调用 printf 函数，打印我们的字符串内容
15    addi a1,sp,12 # 为 a1 寄存器分配空间
16    addi a0,s0,%lo(.LC1) # 完整输出.LC1

```



```

17      call __isoc99_scanf # 调用 scanf 函数完成内容的读取
18
19      flw fa5,8(sp) # 读入第一个浮点数
20      flw fa4,12(sp) # 读入第二个浮点数
21      fgt.s a5,fa5,fa4 # 比较 fa5 与 fa4 的大小关系
22      fcvt.d.s fa4,fa4 # 将 fa4 转化为双精度浮点数
23      fcvt.d.s fa5,fa5 # 将 fa5 转化为双精度浮点数
24      fmv.x.d a2,fa4 # 将 fa4 的值移动到 a2 寄存器
25      fmv.x.d a1,fa5 # 将 fa5 的值移动到 a1 寄存器
26      beq a5,zero,.L6 # 核心步骤，将a5与zero寄存器作比较，
    ↪ 如果相等就直接跳转到.L6语句
27      lui a0,%hi(.LC3) # 将.LC3 的高位读入 a0 寄存器
28      addi a0,a0,%lo(.LC3) # 将.LC3 完整读取
29      call printf # 调用 printf 函数完成输出

```

这段 main 函数就是本程序中最核心的一部分，我们对其进行详细的分析。首先，使用 lui 指令将.LC0 高 20 位地址中的内容加载到我们的 a0 寄存器中，然后进行栈帧空间的分配，跟前面的程序是一样的，我们还是给分配 32 位空间，用于存放局部变量和寄存器。然后将.LC0 剩下的 12 位低位也读取进去，这样我们就完成了对字符串.LC0 的完整读取，存放到了 a0 寄存器中。

然后我们完成对 ra 寄存器和 s0 寄存器的位置的保存，这些都是预先需要完成的步骤了，前面的程序也提到过，此处就不再说明。然后就可以调用 printf 函数了，将我们的 a0 寄存器中存放的内容输出。

然后就是写入.LC1 中的内容，这跟前面的也是一样的。此处我们需要读取第一个输入的浮点数，因此我们将其存放在栈指针下面 8 字节的位置，然后正常读取.LC1 中的内容，最后调用 scanf 函数进行格式化字符串的输出。

然后就是.LC2，还是一样，将字符串成功写入 a0 寄存器中，然后调用 printf 函数将我们的前置语句进行输出。然后我们就可以进行第二个浮点数的输入了，我们将其位置分配在 sp 寄存器的后面 12 字节的位置上。最后，调用 scanf 函数，读取我们输入的第二个浮点数。

然后就到了我们的浮点数读取的环节，我们首先使用 flw 命令将栈上的第一个浮点数读取到我们的 fa5 寄存器，然后将栈上的第二个寄存器保存到我们的 fa4 寄存器。保存完之后我们就需要对其进行大小的比较了。利用 fgt.s 命令，比较 fa5 寄存器和 fa4 寄存器中的值的大小，并将比较的结果存放到 a5 寄存器中。然后，利用 fcvt.d.s 命令将 fa4 和 fa5 中的单精度浮点数转化为双精度浮点数，利用 fmv.x.d 命令将 fa4 的值和 fa5 的值移动到寄存器 a2 和 a1。

如果 a5 的值等于 0，那么就是说两个输入的浮点数是相等的，直接跳转到.L6 的位置。然后读取.LC3 中的内容，输出.LC3 的内容即可。

```

1 • .L4:
2
3      ld ra,24(sp) # 恢复寄存器 ra 的地址
4      ld s0,16(sp) # 恢复寄存器 s0 的地址
5      li a0,0 # 为 a0 寄存器赋值为 0，以便作为函数返回值

```

```

6      addi sp,sp,32 # 将 sp 栈帧的位置恢复到初始状态
7      jr ra # 跳转到 ra 中保存的地址执行，返回调用函数的地址

```

运行完 main 函数之后，我们就可以进行内存的恢复工作了。首先将 sp 指向的地址偏移 24 字节的位置加载数据到寄存器 ra，然后将 sp 指向的地址偏移 16 字节的位置加载数据到寄存器 s0，将立即数 0 加载到 a0 寄存器，用作函数返回值。最后，恢复 sp 栈帧的位置。然后跳转到 ra 中保存的地址执行，这样就可以返回到调用函数的位置继续执行了。

```

1 • .L6:
2     lui a0,%hi(.LC4) # 读取.LC4 的字符串高位
3     addi a0,a0,%lo(.LC4) # 读取.LC4 的字符串低位
4     call printf # 调用 printf 函数
5     j .L4 # 判断并跳转到.L4 语句
6     .size main, .-main # 规定 main 函数的大小
7     .ident "GCC: () 12.2.0" # 规定 GCC 编译的版本
8     .section .note.GNU-stack,"",@progbits # 申明栈保护的功能

```

这段代码跟前面的都差不多，就是读取.LC4 中的字符串，然后调用 printf 进行输出，然后跳转到 L4 代码段。最后申明 main 函数的大小，以及 GCC 编译的版本等。

#### 4. gcc 验证

我们编写完 float.s 汇编程序后，用 riscv64-unknown-linux-gnu-gcc 进行验证。输入以下命令行：

```

1 riscv64-unknown-linux-gnu-gcc float.s -o float -static
2 qemu-riscv64 float

```

观察到程序成功编译了，输入两个数——10.9 和 9.9，运算结果如图23所示，说明我们的程序编写成功。

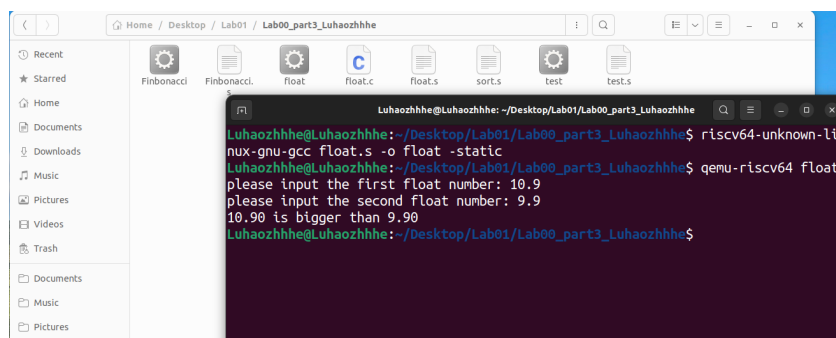


图 23: 测试浮点数汇编程序

#### 5. 一些思考

编写完该汇编程序后，我使用很多组数据进行了测试，发现还是出现了一些小的问题。比如对于很多小数位的数字，我们不能进行完整的输出；对于一些异常输入（比如说中文）等都会使

程序出错。所以我认为需要完善该程序的话，还需要增加一些限定条件，比如遇到错误的输入需要输出报错信息，遇到一些小数点后很多位的数字也需要进行异常处理。

对于栈帧空间的分配这一方面，跟上面的程序大致是一样的，而关于内存对齐这一方面，本汇编程序也是做的比较出色，在字符串处使用 8 字节对齐。

## 五、 总结

**通过本次实验，我们对自己的编译器、LLVM IR 编程以及汇编编程有了更加深刻的理解。**

从**任务一**中，我们首先了解了完整的编译过程，然后分步骤对其进行探究，其中包括了预处理器阶段的功能，编译器中的六大步骤——词法分析、语法分析、语义分析、中间代码生成、代码优化、代码生成，汇编器，链接器以及程序的执行流程。

在预处理器的分析阶段，我充分了解了预处理阶段的功能，并且通过编写程序并进行预处理，验证了这些功能，包括去除注释、文件包含处理、宏定义替换等等；在编译器的分析阶段，我们通过分析词法分析的结果，了解到了词法分析的功能，通过语法分析的实践，了解到了抽象语法树的生成过程以及语法分析检查代码语法的功能。接着我还通过编写一个错误的程序进行语法分析，通过分析输出结果，充分了解了其检查语法的功能；在语义分析阶段，我了解到了其类型检查、数据流分析等功能；在中间代码生成阶段，我们通过 CFG 可视化分析，了解到了代码之间的逻辑关系；在代码优化阶段，通过三种不同形式的优化形式，我们对其中的代码进行了对比，然后通过 10000 次运算均值的比较，对优化有了进一步的了解；代码生成阶段，了解到了 llvm、x86 和 arm 三种形式的目标代码。在汇编器的阶段，我们分析了反汇编的结果；在链接器的阶段，我们分析了部分链接器生成的汇编代码，了解到了链接器的部分功能，最后通过执行程序，成功完成了整个流程。

从**任务二**中，我编写了 LLVM IR 程序并进行了优化和探索，首先根据最简单的斐波那契数列程序，我学习了 LLVM IR 语言的基本写法，在此基础上，我对程序做了数组和浮点数的改进，并使用 OpaquePointers 特性进行指针优化，最后，对编译的程序进行验证，产生了理想的输出结果。

从**任务三**中，我编写了两个汇编程序，首先是斐波那契数列程序。通过 SysY 源程序的编写，我首先明确了汇编程序的编写思路，主要是关于程序的输入和输出、程序栈分配问题以及程序的性能优化。然后我编写了其汇编程序，并且对其进行了逐行的分析。之后，我使用 gcc 命令行对其进行了验证，发现其成功完成编译，最后输入  $n=10$ ，程序输出正确。最后，我还对我编写的程序进行了深入的思考。后面第二个程序——浮点数也是一样的流程，最后输入两个数，发现比较结果正确。