



南开大学  
Nankai University

南 开 大 学  
网 络 空 间 安 全 学 院  
编译原理实验报告

---

预备工作 1——了解编译器及 LLVM IR 编程

---

聂志强 2012307

年级：2020 级

专业：信息安全

指导教师：王刚

2022 年 10 月 2 日

## 摘要

本次实验我通过改进版斐波拉契数列程序在 Ubuntu 虚拟机上对编译器的各个阶段及其功能进行了探索。通过加入宏定义、死代码等部分对编译预处理的功能进行验证，探索了词法分析和语法分析的过程；通过将 CFG 可视化分析了中间代码生成的多阶段；在代码优化阶段，对 O1 O3 和不同优化 pass 方法对比，并通过程序性能（运行时间）进行优化验证，为增强结果稳定性，在原始斐波那契程序基础上进行 10000 次循环并取均值。通过反汇编，对比 x86、arm 和 llvm 汇编和链接结果并得出结论。融合函数、数组、隐式类型转换、9 种运算、“geiint”和“putint”等 SysY 编译器各语言特性编写 LLVM IR 程序，并用 LLVM 编译成目标程序并成功执行验证。

**关键字：**改进版斐波那契, 优化对比, 性能测试, SysY

## 目录

一、 预备工作及实验平台	1
二、 实验过程	1
(一) 完整编译过程	1
(二) 预处理器	1
1. 预处理阶段功能	1
2. 验证过程及结果分析	2
(三) 编译器	3
1. 编译过程——词法分析	3
2. 编译过程——语法分析	3
3. 编译过程——语义分析与中间代码生成	4
4. 编译过程——代码优化	6
5. 编译过程——代码生成	8
(四) 汇编器	8
(五) 链接器	10
(六) 执行	15
(七) LLVM IR 程序	16
三、 总结	20

## 一、预备工作及实验平台

说明：为了方便老师和助教评阅报告，报告中所有完整输出均以附件形式放到文件夹中，报告中仅以部分重要截图展现，辛苦老师和助教批评指正，谢谢

实验平台：

设备名称	lwj-virtual-machine
系统名称	Ubuntu 18.04.3 LTS
操作系统类型	Linux 64 位
Vscode	1.71.2
虚拟机	VMare

表 1: 实验平台参数

## 二、实验过程

以如下斐波拉契数列的 main.c 程序为基准，在下列不同探究阶段进行程序不同改动以探究编译过程

### (一) 完整编译过程

- (1) 预编译
- (2) 编译
- (3) 汇编
- (4) 链接加载

查看编译步骤：clang -ccc-print-phases main.c

```
nie762174555@ubuntu:~$ 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
```

图 1: 编译各阶段

### (二) 预处理器

#### 1. 预处理阶段功能

预处理阶段会处理预编译指令，包括绝大多数的开头的指令，如 include define if 等等，对 include 指令会替换对应的头文件，对 define 的宏命令会直接替换相应内容，同时会删除注释，添加行号和文件名标识。

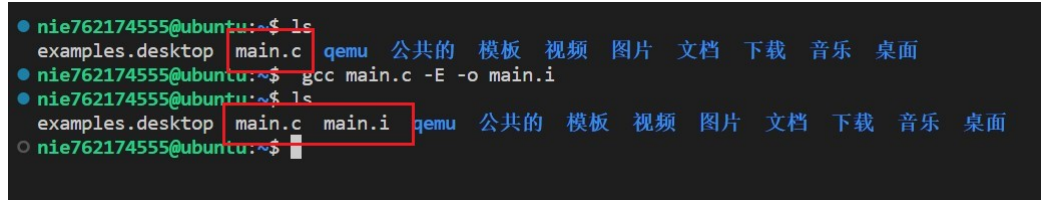
对于 gcc，通过添加参数-E 令 gcc 只进行预处理过程，参数-o 改变 gcc 输出文件名，因此通过命令 gcc main.c -E -o main.i，即可得到预处理后文件。

观察预处理文件，可以发现文件长度远大于源文件，这就是将代码中的头文件进行了替代导致的结果。

## 2. 验证过程及结果分析

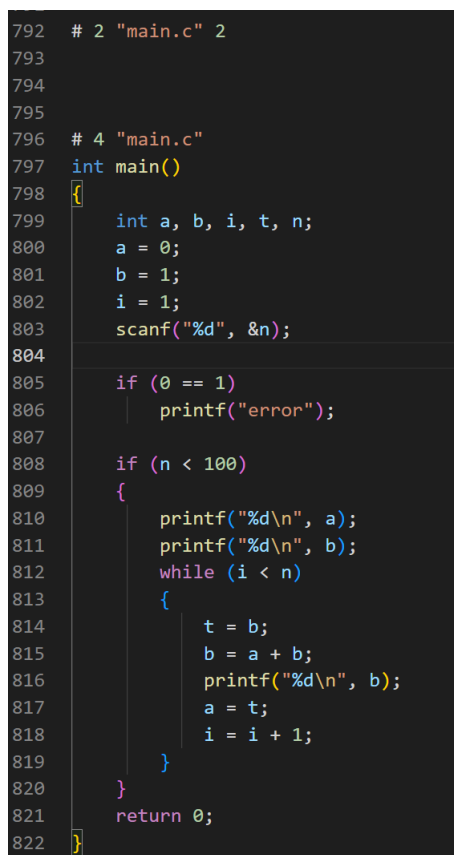
为了验证预处理的上述功能，我们对程序做了以下修改：通过增加宏定义、死代码、注释等进行验证。

执行 `gcc main.c -E -o main.i`，即可得到预处理后文件



```
nie762174555@ubuntu:~$ ls
examples.desktop  main.c  qemu  公共的  模板  视频  图片  文档  下载  音乐  桌面
nie762174555@ubuntu:~$ gcc main.c -E -o main.i
nie762174555@ubuntu:~$ ls
examples.desktop  main.c  main.i  qemu  公共的  模板  视频  图片  文档  下载  音乐  桌面
nie762174555@ubuntu:~$
```

图 2: 预编译前后文件对比



```
792 # 2 "/main.c" 2
793
794
795
796 # 4 "main.c"
797 int main()
798 {
799     int a, b, i, t, n;
800     a = 0;
801     b = 1;
802     i = 1;
803     scanf("%d", &n);
804
805     if (0 == 1)
806         printf("error");
807
808     if (n < 100)
809     {
810         printf("%d\n", a);
811         printf("%d\n", b);
812         while (i < n)
813         {
814             t = b;
815             b = a + b;
816             printf("%d\n", b);
817             a = t;
818             i = i + 1;
819         }
820     }
821     return 0;
822 }
```

图 3: 预处理结果（省略文件头替换内容）

可以观察到，程序对 `include` 指令会替换对应的头文件，对 `define` 的宏命令 `MAX` 直接替换相应内容 100，删除了死代码和注释并添加了行号和文件名标识，从而验证了预处理的上述功能。

### (三) 编译器

#### 1. 编译过程——词法分析

利用 llvm: clang -E -Xclang -dump-tokens main.c, 将源程序转换为单词序列

可以发现, 词法分析过程中, 对源程序的字符串进行扫描和分解, 识别出一个个的单词, 对程序进行分词处理, 并标明每个 token 的类型。部分词法分析结果如下图所示:

```
if 'if' [StartOfLine] [LeadingSpace] Loc=<main.c:15:5>
l_paren '(' [LeadingSpace] Loc=<main.c:15:8>
identifier 'n' Loc=<main.c:15:9>
less '<' [LeadingSpace] Loc=<main.c:15:11>
numeric_constant '100' [LeadingSpace] Loc=<main.c:15:13 <Spelling=main.c:3:13>
r_paren ')' Loc=<main.c:15:16>
l_brace '{' [StartOfLine] [LeadingSpace] Loc=<main.c:16:5>
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:17:9>
l_paren '(' Loc=<main.c:17:15>
string_literal '%d\n' Loc=<main.c:17:16>
comma ',' Loc=<main.c:17:22>
identifier 'a' [LeadingSpace] Loc=<main.c:17:24>
r_paren ')' Loc=<main.c:17:25>
semi ';' Loc=<main.c:17:26>
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:18:9>
l_paren '(' Loc=<main.c:18:15>
string_literal '%d\n' Loc=<main.c:18:16>
comma ',' Loc=<main.c:18:22>
identifier 'b' [LeadingSpace] Loc=<main.c:18:24>
r_paren ')' Loc=<main.c:18:25>
semi ';' Loc=<main.c:18:26>
while 'while' [StartOfLine] [LeadingSpace] Loc=<main.c:19:9>
l_paren '(' [LeadingSpace] Loc=<main.c:19:15>
identifier 'i' Loc=<main.c:19:16>
less '<' [LeadingSpace] Loc=<main.c:19:18>
identifier 'n' [LeadingSpace] Loc=<main.c:19:20>
r_paren ')' Loc=<main.c:19:21>
l_brace '{' [StartOfLine] [LeadingSpace] Loc=<main.c:20:9>
identifier 't' [StartOfLine] [LeadingSpace] Loc=<main.c:21:13>
equal '=' [LeadingSpace] Loc=<main.c:21:15>
identifier 'b' [LeadingSpace] Loc=<main.c:21:17>
semi ';' Loc=<main.c:21:18>
identifier 'b' [StartOfLine] [LeadingSpace] Loc=<main.c:22:13>
equal '=' [LeadingSpace] Loc=<main.c:22:15>
identifier 'a' [LeadingSpace] Loc=<main.c:22:17>
plus '+' [LeadingSpace] Loc=<main.c:22:19>
identifier 'b' [LeadingSpace] Loc=<main.c:22:21>
semi ';' Loc=<main.c:22:22>
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:23:13>
l_paren '(' Loc=<main.c:23:19>
string_literal '%d\n' Loc=<main.c:23:20>
comma ',' Loc=<main.c:23:26>
identifier 'b' [LeadingSpace] Loc=<main.c:23:28>
r_paren ')' Loc=<main.c:23:29>
semi ';' Loc=<main.c:23:30>
identifier 'a' [StartOfLine] [LeadingSpace] Loc=<main.c:24:13>
equal '=' [LeadingSpace] Loc=<main.c:24:15>
identifier 't' [LeadingSpace] Loc=<main.c:24:17>
semi ';' Loc=<main.c:24:18>
identifier 'i' [StartOfLine] [LeadingSpace] Loc=<main.c:25:13>
equal '=' [LeadingSpace] Loc=<main.c:25:15>
identifier 'i' [LeadingSpace] Loc=<main.c:25:17>
plus '+' [LeadingSpace] Loc=<main.c:25:19>
numeric_constant '1' [LeadingSpace] Loc=<main.c:25:21>
semi ';' Loc=<main.c:25:22>
r_brace '}' [StartOfLine] [LeadingSpace] Loc=<main.c:26:9>
r_brace '}' [StartOfLine] [LeadingSpace] Loc=<main.c:27:5>
return 'return' [StartOfLine] [LeadingSpace] Loc=<main.c:28:5>
numeric_constant '0' [LeadingSpace] Loc=<main.c:28:12>
semi ';' Loc=<main.c:28:13>
r_brace '}' [StartOfLine] Loc=<main.c:29:1>
eof '' Loc=<main.c:29:2>
```

图 4: 部分词法分析结果)

#### 2. 编译过程——语法分析

将词法分析生成的词法单元来构建抽象语法树 (Abstract Syntax Tree, 即 AST)。利用 llvm 进行语法分析: clang -E -Xclang -ast-dump main.c

可以发现, 语法分析阶段利用词法分析阶段的单词构成一棵语法分析树, 可以看到明显的层次关系。部分语法分析结果如下图所示:

[illegible]

图 5: 部分语法分析结果

### 3. 编译过程——语义分析与中间代码生成

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、范围检查、数组绑定检查等。

(1) 利用 `llvm clang -S -emit-llvm -Xclang -disable-O0-optnone main.c` 生成 `main.ll` 中间代码:

```
nie762174555@ubuntu:~$ clang -S -emit-llvm main.c
nie762174555@ubuntu:~$ ls
examples.desktop  main.c  main.i  main.ll  gemu  公共的  模板  视频  图片  文档  下载  音乐  桌面
```

图 6: 生成 main.dll 文件

可以发现，语义分析与中间代码生成阶段生成的.ll 文件中，死代码进行了删除，说明在代码优化这一步之前就已经对死代码进行了优化。

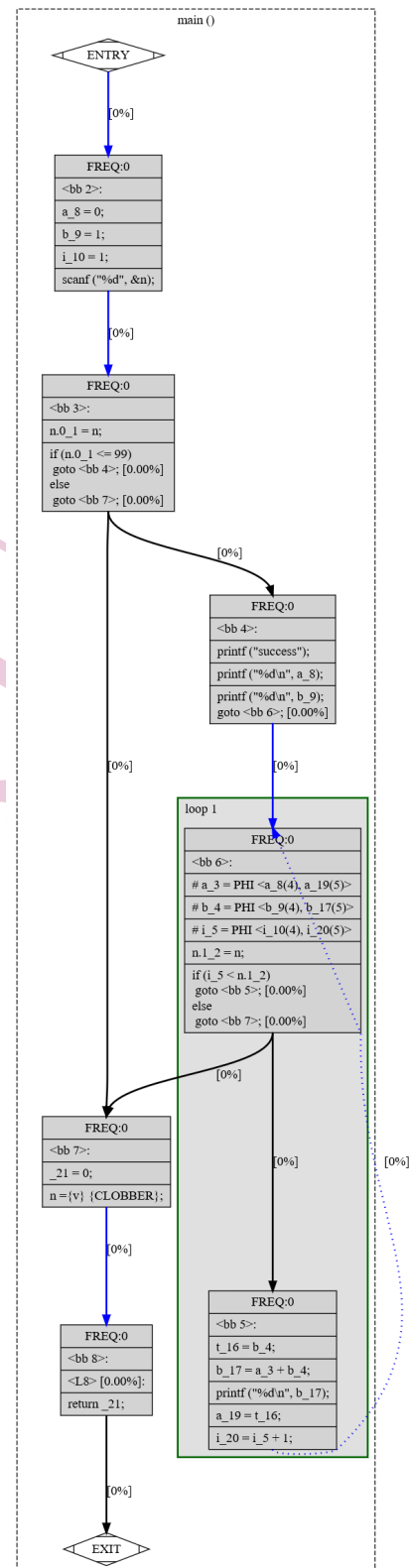
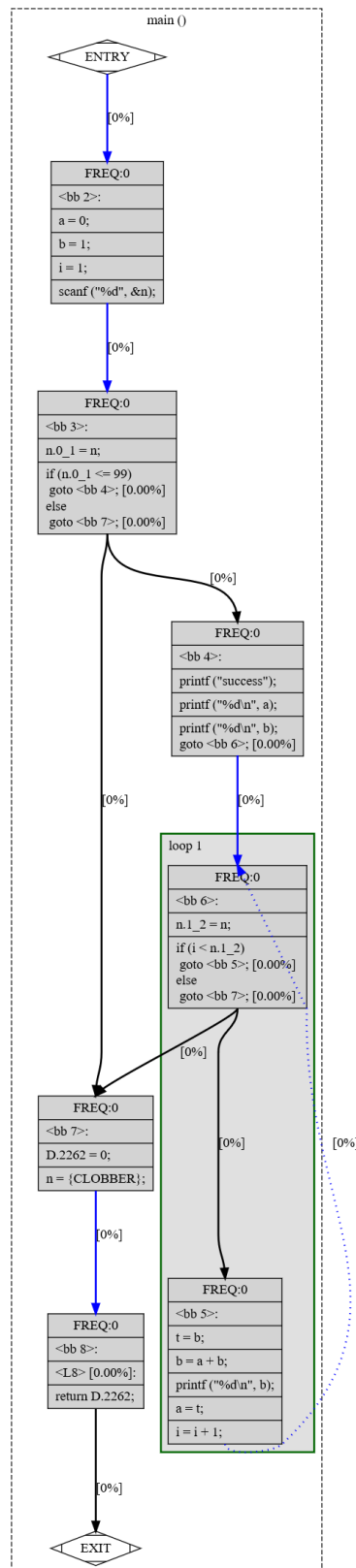
(2) 利用 gcc 通过 `gcc -fdump-tree-all-graph main.c` 获得中间代码生成的多阶段输出, 通过 graphviz 对 CFG 进行可视化, 此处选取 `main.c.011t.cfg.dot` 和 `main.c.086.fixup_cfg4.dot` 进行分析:

可以很明显看到块与块之间的逻辑关系。包括各部分的跳转、分支等。

可以发现控制流图 CFG 的变化是:

- a、b、i 等变量标识符的变化，所有的变量都变成标识符 \_ 编号的形式 (ssa) 了，返回值没标识符就只有 \_ 编号 (由 D.2262 变成 \_21)，通过数字标识区分不同阶段这些变量有不同值。
- loop1 块中增加了 a\_3=PHI<a\_8(4),a\_19(5)> b\_4=PHI<b9(4),b\_17(5)> i\_5=PHI<i\_10(4),i\_20(5)> 注释。例如 a\_3=PHI<a\_8(4),a\_19(5)> 表明表示 a\_3 可能等于块 4 中的 a\_8，也可能

等于块 5 中的 a\_19。(其中 PHI 为希腊字母，大概只是一个标识)



#### 4. 编译过程——代码优化

在使用 pass 进行优化之前，需要先使用指令“`llvm-as main.ll -o main.bc`”得到 LLVM IR 的二进制代码形式

```

nie762174555@ubuntu:~$ llvm-as main.ll -o main.bc
nie762174555@ubuntu:~$ ls
examples.desktop  main.bc  main.c  main.i  main.ll  qemu  公共的  模板  视频  图片  文档  下载  音乐  桌面

```

图 7: 生成 main.bc 二进制文件

(1) O1 O3 不同级别优化生成中间代码对比分析通过指令 `opt -S -O1 main.bc -o main-O1.ll` 对中间代码进行 O1 级别优化：

```

@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@.str.1 = private unnamed_addr constant [8 x i8] c"success\00", align 1
@.str.2 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: noinline nounwind uwtable
define i32 @main() local_unnamed_addr #0 {
    %1 = alloca i32, align 4
    %2 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* nonnull %1)
    %3 = load i32, i32* %1, align 4
    %4 = icmp slt i32 %3, 100
    br i1 %4, label %5, label %.loopexit

; <label>:; preds = %0
    %6 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([8 x i8], [8 x i8]* @.str.1, i64 0, i64 0))
    %7 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str.2, i64 0, i64 0), i32 0)
    %8 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str.2, i64 0, i64 0), i32 1)
    %9 = load i32, i32* %1, align 4
    %10 = icmp sgt i32 %9, 1
    br i1 %10, label %.lr.ph.preheader, label %.loopexit

.lr.ph.preheader:                                ; preds = %5
    br label %.lr.ph

.lr.ph:                                           ; preds = %.lr.ph.preheader, %.lr.ph
    %012 = phi i32 [ %01011, %.lr.ph ], [ 0, %.lr.ph.preheader ]
    %012 = phi i32 [ %13, %.lr.ph ], [ 1, %.lr.ph.preheader ]
    %01011 = phi i32 [ %11, %.lr.ph ], [ 1, %.lr.ph.preheader ]
    %11 = add nsw i32 %012, %01011
    %12 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str.2, i64 0, i64 0), i32 %11)
    %13 = add nsw nsw i32 %012, 1
    %14 = load i32, i32* %1, align 4
    %15 = icmp slt i32 %13, %14
    br i1 %15, label %.lr.ph, label %.loopexit

.loopexit:                                       ; preds = %.lr.ph, %5, %0
    ret i32 0
}

; Function Attrs: nounwind
declare i32 @__isoc99_scanf(i8* nocapture readonly, ...) local_unnamed_addr #1

; Function Attrs: nounwind
declare i32 @__printf(i8* nocapture readonly, ...) local_unnamed_addr #1

attributes #0 = { noinline nounwind uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-prec
attributes #1 = { nounwind "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false

llvm.module.flags = [{!0}]
llvm.ident = [{!1}]

!0 = !{i32 1, !"wchar size", i32 4}

```

图 8: O1 级中间代码优化

在比较 O1 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，为增加稳定性，每组（求  $n$  个斐波那契数）循环 10000 次并取平均值，如下图所示：



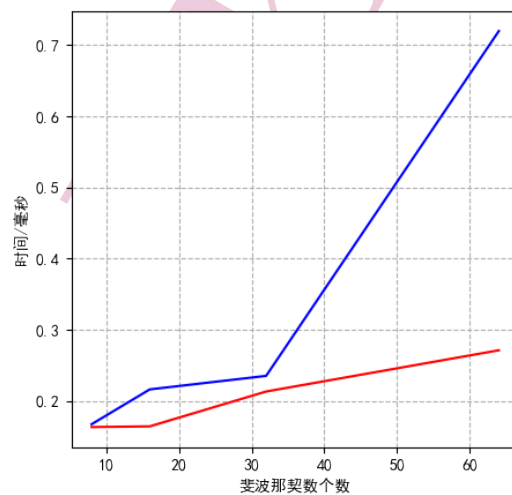


图 9: 性能比较

通过指令 `opt -S -O2 main.bc -o main-O2.ll` 和 `opt -S -O3 main.bc -o main-O3.ll` 对中间代码分别进行 O2 和 O3 级别优化, 优化之后发现和 O1 优化没有区别

(2) 不同优化 pass 进行中间代码优化对比之后我尝试分别从三类 pass 中各选出一种 pass 来进行优化, 以观察特定 pass 的差异。在 Analysis pass 类中, 我选取了 `-aa-eval` 这一模块进行测试, 使用的指令为 `"opt -S -aa-eval main.bc -o main-eval.ll"` 之后我在后面两类 pass 中各选取了若干个不同的 pass 模块来做优化, 如 `-loop-unroll`, `-codegenprepare` 等, 但得到的 IR 文件内容都没有变化, 推测可能是源程序过于简单, 优化空间小。

## 5. 编译过程——代码生成

以中间表示形式作为输入, 将其映射到目标语言。利用 LLVM 指令 `llc main.ll -o main-llvm.S` 生成目标代码。同理可以利用 `gcc main.i -S -o main-x86.S` 和 `arm-linux-gnueabi-gcc main.i -S -o main-arm.S` 指令分别生成 x86 和 arm 格式的目标代码, 即分别生成 CISC 和 RISC 汇编代码。

```

● nie762174555@ubuntu:~$ llc main.ll -o main-llvm.S
● nie762174555@ubuntu:~$ gcc main.i -S -o main-x86.S
● nie762174555@ubuntu:~$ arm-linux-gnueabi-gcc main.i -S -o main-arm.S
○ nie762174555@ubuntu:~$

```

图 10: 生成目标代码

## (四) 汇编器

汇编过程实际上把汇编语言程序代码(编译过程生成)翻译成目标机器指令的过程, 每一个汇编语句几乎都对应一条机器指令。`llc main-O1.bc -filetype=obj -o main-O1.o` `llc main.bc -filetype=obj -o main-llvm.o`, LLVM 可以直接使用 `llc` 命令同时汇编和链接 LLVM bitcode 生成机器指令。也可以通过 `gcc main-x86.S -c -o main-x86.o` 生成 x86 的目标机器指令, 得到 `main-x86.o` 文件或通过 `arm-linux-gnueabi-gcc main-arm.S -c -o main-arm.o` 生成 arm 的目标机器指令, 得到 `main-arm.o` 文件。

```

● nie762174555@ubuntu:~$ llc main.bc -filetype=obj -o main-llvm.o
● nie762174555@ubuntu:~$ gcc main-x86.S -c -o main-x86.o
● nie762174555@ubuntu:~$ arm-linux-gnueabi-gcc main-arm.S -c -o main-arm.o
○ nie762174555@ubuntu:~$

```

图 11: 汇编器结果

由于篇幅原因, 此处仅针对 x86 进行反汇编分析, 利用 `objdump -d main-x86.o` 对 `main-x86.o` 反汇编, 下图所示为反汇编结果:

反汇编结果

```

1 main-x86.o:          文件格式 elf64-x86-64
2
3
4 Disassembly of section .text:
5
6 0000000000000000 <main>:
7   0:   55                      push   %rbp
8   1:  48 89 e5                mov    %rsp,%rbp
9   4:  48 83 ec 20             sub    $0x20,%rsp
10  8:  64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
11  f:  00 00
12 11:  48 89 45 f8             mov    %rax,-0x8(%rbp)
13 15:  31 c0                   xor    %eax,%eax
14 17:  c7 45 e8 00 00 00 00    movl   $0x0,-0x18(%rbp)
15 1e:  c7 45 ec 01 00 00 00    movl   $0x1,-0x14(%rbp)
16 25:  c7 45 f0 01 00 00 00    movl   $0x1,-0x10(%rbp)

```

17	2c:	48 8d 45 e4	lea	-0x1c(%rbp),%rax	
18	30:	48 89 c6	mov	%rax,%rsi	
19	33:	48 8d 3d 00 00 00 00	lea	0x0(%rip),%rdi	# 3a <main+0x3a>
20	3a:	b8 00 00 00 00	mov	\$0x0,%eax	
21	3f:	e8 00 00 00 00	callq	44 <main+0x44>	
22	44:	8b 45 e4	mov	-0x1c(%rbp),%eax	
23	47:	83 f8 63	cmp	\$0x63,%eax	
24	4a:	7f 73	jg	bf <main+0xbf>	
25	4c:	48 8d 3d 00 00 00 00	lea	0x0(%rip),%rdi	# 53 <main+0x53>
26	53:	b8 00 00 00 00	mov	\$0x0,%eax	
27	58:	e8 00 00 00 00	callq	5d <main+0x5d>	
28	5d:	8b 45 e8	mov	-0x18(%rbp),%eax	
29	60:	89 c6	mov	%eax,%esi	
30	62:	48 8d 3d 00 00 00 00	lea	0x0(%rip),%rdi	# 69 <main+0x69>
31	69:	b8 00 00 00 00	mov	\$0x0,%eax	
32	6e:	e8 00 00 00 00	callq	73 <main+0x73>	
33	73:	8b 45 ec	mov	-0x14(%rbp),%eax	
34	76:	89 c6	mov	%eax,%esi	
35	78:	48 8d 3d 00 00 00 00	lea	0x0(%rip),%rdi	# 7f <main+0x7f>
36	7f:	b8 00 00 00 00	mov	\$0x0,%eax	
37	84:	e8 00 00 00 00	callq	89 <main+0x89>	
38	89:	eb 2c	jmp	b7 <main+0xb7>	
39	8b:	8b 45 ec	mov	-0x14(%rbp),%eax	
40	8e:	89 45 f4	mov	%eax,-0xc(%rbp)	
41	91:	8b 45 e8	mov	-0x18(%rbp),%eax	
42	94:	01 45 ec	add	%eax,-0x14(%rbp)	
43	97:	8b 45 ec	mov	-0x14(%rbp),%eax	
44	9a:	89 c6	mov	%eax,%esi	
45	9c:	48 8d 3d 00 00 00 00	lea	0x0(%rip),%rdi	# a3 <main+0xa3>
46	a3:	b8 00 00 00 00	mov	\$0x0,%eax	
47	a8:	e8 00 00 00 00	callq	ad <main+0xad>	
48	ad:	8b 45 f4	mov	-0xc(%rbp),%eax	
49	b0:	89 45 e8	mov	%eax,-0x18(%rbp)	
50	b3:	83 45 f0 01	addl	\$0x1,-0x10(%rbp)	
51	b7:	8b 45 e4	mov	-0x1c(%rbp),%eax	
52	ba:	39 45 f0	cmp	%eax,-0x10(%rbp)	
53	bd:	7c cc	j1	8b <main+0x8b>	
54	bf:	b8 00 00 00 00	mov	\$0x0,%eax	
55	c4:	48 8b 55 f8	mov	-0x8(%rbp),%rdx	
56	c8:	64 48 33 14 25 28 00	xor	%fs:0x28,%rdx	
57	cf:	00 00			
58	d1:	74 05	je	d8 <main+0xd8>	
59	d3:	e8 00 00 00 00	callq	d8 <main+0xd8>	
60	d8:	c9	leaveq		
61	d9:	c3	retq		

## (五) 链接器

由汇编程序生成的目标文件并不能够直接执行。因为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。进而连接器对该机器代码进行执行生成可执行文件。通过执行 `clang main-llvm.o -o main-llvm` 指令生成 `main-llvm` 可执行文件，同理通过 `gcc main-x86.o -o main-x86` 和 `gcc main-arm.o -o main-arm` 生成 `main-x86` 和 `main-arm` 可执行文件行 `clang main.o -o main`

由于篇幅原因，此处仅针对 `main-x86` 反汇编进行分析，执行 `objdump -d main-x86` 指令，可以发现相较于汇编器生成文件的反汇编结果，可以观察到其他库文件和目标文件的反汇编，验证了链接器的作用。

### 反汇编结果

```

1  main-x86:      文件格式 elf64-x86-64
2  Disassembly of section .init:
3
4  000000000000005a8 <_init>:
5      5a8:  48 83 ec 08      sub    $0x8,%rsp
6      5ac:  48 8b 05 35 0a 20 00  mov    0x200a35(%rip),%rax      # 200
       fe8 <__gmon_start__>
7      5b3:  48 85 c0          test   %rax,%rax
8      5b6:  74 02            je     5ba <_init+0x12>
9      5b8:  ff d0            callq  *%rax
10     5ba:  48 83 c4 08      add    $0x8,%rsp
11     5be:  c3              retq
12
13  Disassembly of section .plt:
14
15  000000000000005c0 <.plt>:
16     5c0:  ff 35 ea 09 20 00  pushq  0x2009ea(%rip)          # 200fb0 <
       _GLOBAL_OFFSET_TABLE_+0x8>
17     5c6:  ff 25 ec 09 20 00  jmpq   *0x2009ec(%rip)        # 200fb8 <
       _GLOBAL_OFFSET_TABLE_+0x10>
18     5cc:  0f 1f 40 00      nopl   0x0(%rax)
19
20  000000000000005d0 <__stack_chk_fail@plt>:
21     5d0:  ff 25 ea 09 20 00  jmpq   *0x2009ea(%rip)        # 200fc0 <
       __stack_chk_fail@GLIBC_2.4>
22     5d6:  68 00 00 00 00    pushq  $0x0
23     5db:  e9 e0 ff ff ff    jmpq   5c0 <.plt>
24
25  000000000000005e0 <printf@plt>:
26     5e0:  ff 25 e2 09 20 00  jmpq   *0x2009e2(%rip)        # 200fc8 <
       printf@GLIBC_2.2.5>
27     5e6:  68 01 00 00 00    pushq  $0x1
28     5eb:  e9 d0 ff ff ff    jmpq   5c0 <.plt>
29
30  000000000000005f0 <__isoc99_scanf@plt>:

```

```

31      5f0:  ff 25 da 09 20 00      jmpq    *0x2009da(%rip)      # 200fd0 <
      __isoc99_scanf@GLIBC_2.7>
32      5f6:  68 02 00 00 00      pushq   $0x2
33      5fb:  e9 c0 ff ff ff      jmpq    5c0 <.plt>
34
35  Disassembly of section .plt.got:
36
37  00000000000000600 <__cxa_finalize@plt>:
38      600:  ff 25 f2 09 20 00      jmpq    *0x2009f2(%rip)      # 200ff8 <
      __cxa_finalize@GLIBC_2.2.5>
39      606:  66 90                  xchg     %ax,%ax
40
41  Disassembly of section .text:
42
43  00000000000000610 <_start>:
44      610:  31 ed                  xor      %ebp,%ebp
45      612:  49 89 d1              mov      %rdx,%r9
46      615:  5e                    pop      %rsi
47      616:  48 89 e2              mov      %rsp,%rdx
48      619:  48 83 e4 f0           and      $0xfffffffffffffff0,%rsp
49      61d:  50                    push     %rax
50      61e:  54                    push     %rsp
51      61f:  4c 8d 05 4a 02 00 00  lea      0x24a(%rip),%r8      # 870 <
      __libc_csu_fini>
52      626:  48 8d 0d d3 01 00 00  lea      0x1d3(%rip),%rcx      # 800 <
      __libc_csu_init>
53      62d:  48 8d 3d e6 00 00 00  lea      0xe6(%rip),%rdi      # 71a <main
      >
54      634:  ff 15 a6 09 20 00      callq   *0x2009a6(%rip)      # 200fe0 <
      __libc_start_main@GLIBC_2.2.5>
55      63a:  f4                    hlt
56      63b:  0f 1f 44 00 00        nopl     0x0(%rax,%rax,1)
57
58  00000000000000640 <deregister_tm_clones>:
59      640:  48 8d 3d c9 09 20 00  lea      0x2009c9(%rip),%rdi      #
      201010 <__TMC_END__>
60      647:  55                    push     %rbp
61      648:  48 8d 05 c1 09 20 00  lea      0x2009c1(%rip),%rax      #
      201010 <__TMC_END__>
62      64f:  48 39 f8              cmp      %rdi,%rax
63      652:  48 89 e5              mov      %rsp,%rbp
64      655:  74 19                je       670 <deregister_tm_clones+0x30>
65      657:  48 8b 05 7a 09 20 00  mov      0x20097a(%rip),%rax      # 200
      fd8 <_ITM_deregisterTMCloneTable>
66      65e:  48 85 c0              test     %rax,%rax
67      661:  74 0d                je       670 <deregister_tm_clones+0x30>
68      663:  5d                    pop      %rbp
69      664:  ff e0                jmpq     *%rax

```

```

70 666: 66 2e 0f 1f 84 00 00 nopw %cs:0x0(%rax,%rax,1)
71 66d: 00 00 00
72 670: 5d pop %rbp
73 671: c3 retq
74 672: 0f 1f 40 00 nopl 0x0(%rax)
75 676: 66 2e 0f 1f 84 00 00 nopw %cs:0x0(%rax,%rax,1)
76 67d: 00 00 00
77
78 0000000000000680 <register_tm_clones>:
79 680: 48 8d 3d 89 09 20 00 lea 0x200989(%rip),%rdi #
201010 <__TMC_END__>
80 687: 48 8d 35 82 09 20 00 lea 0x200982(%rip),%rsi #
201010 <__TMC_END__>
81 68e: 55 push %rbp
82 68f: 48 29 fe sub %rdi,%rsi
83 692: 48 89 e5 mov %rsp,%rbp
84 695: 48 c1 fe 03 sar $0x3,%rsi
85 699: 48 89 f0 mov %rsi,%rax
86 69c: 48 c1 e8 3f shr $0x3f,%rax
87 6a0: 48 01 c6 add %rax,%rsi
88 6a3: 48 d1 fe sar %rsi
89 6a6: 74 18 je 6c0 <register_tm_clones+0x40>
90 6a8: 48 8b 05 41 09 20 00 mov 0x200941(%rip),%rax # 200
ff0 <__ITM_registerTMCloneTable>
91 6af: 48 85 c0 test %rax,%rax
92 6b2: 74 0c je 6c0 <register_tm_clones+0x40>
93 6b4: 5d pop %rbp
94 6b5: ff e0 jmpq *%rax
95 6b7: 66 0f 1f 84 00 00 00 nopw 0x0(%rax,%rax,1)
96 6be: 00 00
97 6c0: 5d pop %rbp
98 6c1: c3 retq
99 6c2: 0f 1f 40 00 nopl 0x0(%rax)
100 6c6: 66 2e 0f 1f 84 00 00 nopw %cs:0x0(%rax,%rax,1)
101 6cd: 00 00 00
102
103 00000000000006d0 <__do_global_dtors_aux>:
104 6d0: 80 3d 39 09 20 00 00 cmpb $0x0,0x200939(%rip) #
201010 <__TMC_END__>
105 6d7: 75 2f jne 708 <__do_global_dtors_aux+0x38>
106 6d9: 48 83 3d 17 09 20 00 cmpq $0x0,0x200917(%rip) # 200
ff8 <__cxa_finalize@GLIBC_2.2.5>
107 6e0: 00
108 6e1: 55 push %rbp
109 6e2: 48 89 e5 mov %rsp,%rbp
110 6e5: 74 0c je 6f3 <__do_global_dtors_aux+0x23>
111 6e7: 48 8b 3d 1a 09 20 00 mov 0x20091a(%rip),%rdi #
201008 <__dso_handle>

```

```

112 6ee: e8 0d ff ff ff      callq 600 <__cxa_finalize@plt>
113 6f3: e8 48 ff ff ff      callq 640 <deregister_tm_clones>
114 6f8: c6 05 11 09 20 00 01 movb $0x1,0x200911(%rip)      #
      201010 <_TMC_END_>
115 6ff: 5d                  pop    %rbp
116 700: c3                  retq
117 701: 0f 1f 80 00 00 00 00 nopl 0x0(%rax)
118 708: f3 c3              repz retq
119 70a: 66 0f 1f 44 00 00   nopw 0x0(%rax,%rax,1)
120
121 00000000000000710 <frame_dummy>:
122 710: 55                  push   %rbp
123 711: 48 89 e5            mov     %rsp,%rbp
124 714: 5d                  pop     %rbp
125 715: e9 66 ff ff ff      jmpq    680 <register_tm_clones>
126
127 0000000000000071a <main>:
128 71a: 55                  push   %rbp
129 71b: 48 89 e5            mov     %rsp,%rbp
130 71e: 48 83 ec 20          sub     $0x20,%rsp
131 722: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
132 729: 00 00
133 72b: 48 89 45 f8          mov     %rax,-0x8(%rbp)
134 72f: 31 c0                xor     %eax,%eax
135 731: c7 45 e8 00 00 00 00 movl    $0x0,-0x18(%rbp)
136 738: c7 45 ec 01 00 00 00 movl    $0x1,-0x14(%rbp)
137 73f: c7 45 f0 01 00 00 00 movl    $0x1,-0x10(%rbp)
138 746: 48 8d 45 e4          lea     -0x1c(%rbp),%rax
139 74a: 48 89 c6             mov     %rax,%rsi
140 74d: 48 8d 3d 30 01 00 00 lea     0x130(%rip),%rdi      # 884 <
      _IO_stdin_used+0x4>
141 754: b8 00 00 00 00      mov     $0x0,%eax
142 759: e8 92 fe ff ff      callq 5f0 <__isoc99_scanf@plt>
143 75e: 8b 45 e4             mov     -0x1c(%rbp),%eax
144 761: 83 f8 63             cmp     $0x63,%eax
145 764: 7f 73               jg      7d9 <main+0xbf>
146 766: 48 8d 3d 1a 01 00 00 lea     0x11a(%rip),%rdi      # 887 <
      _IO_stdin_used+0x7>
147 76d: b8 00 00 00 00      mov     $0x0,%eax
148 772: e8 69 fe ff ff      callq 5e0 <printf@plt>
149 777: 8b 45 e8             mov     -0x18(%rbp),%eax
150 77a: 89 c6               mov     %eax,%esi
151 77c: 48 8d 3d 0c 01 00 00 lea     0x10c(%rip),%rdi      # 88f <
      _IO_stdin_used+0xf>
152 783: b8 00 00 00 00      mov     $0x0,%eax
153 788: e8 53 fe ff ff      callq 5e0 <printf@plt>
154 78d: 8b 45 ec             mov     -0x14(%rbp),%eax
155 790: 89 c6               mov     %eax,%esi

```

```

156 792: 48 8d 3d f6 00 00 00 lea 0xf6(%rip),%rdi # 88f <
      _IO_stdin_used+0xf>
157 799: b8 00 00 00 00 mov $0x0,%eax
158 79e: e8 3d fe ff ff callq 5e0 <printf@plt>
159 7a3: eb 2c jmp 7d1 <main+0xb7>
160 7a5: 8b 45 ec mov -0x14(%rbp),%eax
161 7a8: 89 45 f4 mov %eax,-0xc(%rbp)
162 7ab: 8b 45 e8 mov -0x18(%rbp),%eax
163 7ae: 01 45 ec add %eax,-0x14(%rbp)
164 7b1: 8b 45 ec mov -0x14(%rbp),%eax
165 7b4: 89 c6 mov %eax,%esi
166 7b6: 48 8d 3d d2 00 00 00 lea 0xd2(%rip),%rdi # 88f <
      _IO_stdin_used+0xf>
167 7bd: b8 00 00 00 00 mov $0x0,%eax
168 7c2: e8 19 fe ff ff callq 5e0 <printf@plt>
169 7c7: 8b 45 f4 mov -0xc(%rbp),%eax
170 7ca: 89 45 e8 mov %eax,-0x18(%rbp)
171 7cd: 83 45 f0 01 addl $0x1,-0x10(%rbp)
172 7d1: 8b 45 e4 mov -0x1c(%rbp),%eax
173 7d4: 39 45 f0 cmp %eax,-0x10(%rbp)
174 7d7: 7c cc jl 7a5 <main+0x8b>
175 7d9: b8 00 00 00 00 mov $0x0,%eax
176 7de: 48 8b 55 f8 mov -0x8(%rbp),%rdx
177 7e2: 64 48 33 14 25 28 00 xor %fs:0x28,%rdx
178 7e9: 00 00
179 7eb: 74 05 je 7f2 <main+0xd8>
180 7ed: e8 de fd ff ff callq 5d0 <__stack_chk_fail@plt>
181 7f2: c9 leaveq
182 7f3: c3 retq
183 7f4: 66 2e 0f 1f 84 00 00 nopw %cs:0x0(%rax,%rax,1)
184 7fb: 00 00 00
185 7fe: 66 90 xchg %ax,%ax
186
187 0000000000000800 <__libc_csu_init>:
188 800: 41 57 push %r15
189 802: 41 56 push %r14
190 804: 49 89 d7 mov %rdx,%r15
191 807: 41 55 push %r13
192 809: 41 54 push %r12
193 80b: 4c 8d 25 96 05 20 00 lea 0x200596(%rip),%r12 # 200
      da8 <__frame_dummy_init_array_entry>
194 812: 55 push %rbp
195 813: 48 8d 2d 96 05 20 00 lea 0x200596(%rip),%rbp # 200
      db0 <__init_array_end>
196 81a: 53 push %rbx
197 81b: 41 89 fd mov %edi,%r13d
198 81e: 49 89 f6 mov %rsi,%r14
199 821: 4c 29 e5 sub %r12,%rbp

```



```

200      824:  48 83 ec 08          sub    $0x8,%rsp
201      828:  48 c1 fd 03          sar    $0x3,%rbp
202      82c:  e8 77 fd ff ff      callq  5a8 <__init>
203      831:  48 85 ed            test   %rbp,%rbp
204      834:  74 20              je     856 <__libc_csu_init+0x56>
205      836:  31 db              xor     %ebx,%ebx
206      838:  0f 1f 84 00 00 00 00 nopl   0x0(%rax,%rax,1)
207      83f:  00
208      840:  4c 89 fa          mov     %r15,%rdx
209      843:  4c 89 f6          mov     %r14,%rsi
210      846:  44 89 ef          mov     %r13d,%edi
211      849:  41 ff 14 dc      callq  *(%r12,%rbx,8)
212      84d:  48 83 c3 01      add     $0x1,%rbx
213      851:  48 39 dd          cmp     %rbx,%rbp
214      854:  75 ea          jne     840 <__libc_csu_init+0x40>
215      856:  48 83 c4 08      add     $0x8,%rsp
216      85a:  5b              pop     %rbx
217      85b:  5d              pop     %rbp
218      85c:  41 5c          pop     %r12
219      85e:  41 5d          pop     %r13
220      860:  41 5e          pop     %r14
221      862:  41 5f          pop     %r15
222      864:  c3              retq
223      865:  90              nop
224      866:  66 2e 0f 1f 84 00 00 nopw    %es:0x0(%rax,%rax,1)
225      86d:  00 00 00
226
227      0000000000000870 <__libc_csu_fini>:
228      870:  f3 c3          repz retq
229
230      Disassembly of section .fini:
231
232      0000000000000874 <_fini>:
233      874:  48 83 ec 08          sub     $0x8,%rsp
234      878:  48 83 c4 08          add     $0x8,%rsp
235      87c:  c3              retq
236
237 % \end{minted}

```

## (六) 执行

执行./main 如下图所示，main 程序执行成功：

```

nie762174555@ubuntu:~$ ./main
10
success0
1
1
2
3
5
8
13
21
34
55

```

图 12: 执行结果

生成的部分程序展示:

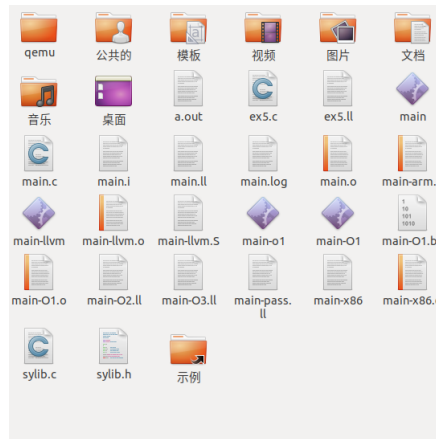


图 13: 文件目录

## (七) LLVM IR 程序

依据已实现的 SysY 语言特性, 小组成员分工如下:

1. 许友锐: (1) 变量、常量的声明及初始化; (2) 函数的定义及调用; (3) 语句: return, 赋值 (=); (4) 表达式: 算术运算, 关系运算和逻辑运算。
2. 聂志强: (1) 数组的声明与数组元素访问; (2) 连接 SysY 运行时库, 实现输入输出函数调用; (3) 语句: if, while, break; (4) int 和 float 之间的隐式类型转换。

所编写的 LLVM IR 代码如下所示:

ex5.ll

```

1 ; 全局常量声明 const float k=2.0;
2 @k = constant float 2.000000e+00, align 4
3 ; 全局变量声明 float a=2.0;
4 @a = global float 2.000000e+00, align 4
5 ; 全局变量声明 float b=2.0;
6 @b = global float 2.000000e+01, align 4
7 ; 全局变量声明 c[3]={1,2,3};
8 @c = global [3 x i32] [i32 1, i32 2, i32 3], align 4
9
10 ; 定义函数 float func(int d)
11 define float @func(i32 %0) {
12     %2 = alloca i32, align 4
13     store i32 %0, i32* %2, align 4

```

```

14  %3 = load float , float* @a, align 4 ; %3存a的值,float类型
15  %4 = load i32 , i32* %2, align 4 ; %4存参数d的值, int类型
16  %5 = sitofp i32 %4 to float ; %4进行类型转换, int转float
17  %6 = fmul float %3, %5
18  %7 = fptosi float %6 to i32 ; %6进行类型转换, float转int
19  store i32 %7, i32* getelementptr inbounds ([3 x i32], [3 x i32]* @c, i64 0,
      i64 1), align 4 ; %7的值存入c[1]
20  %8 = load i32 , i32* getelementptr inbounds ([3 x i32], [3 x i32]* @c, i64
      0, i64 0), align 4 ; %8=c[0]
21  %9 = load i32 , i32* %2, align 4 ; %9=d
22  %10 = srem i32 %8, %9
23  store i32 %10, i32* getelementptr inbounds ([3 x i32], [3 x i32]* @c, i64
      0, i64 2), align 4 ; c[2]=c[0]%d;
24  %11 = load i32 , i32* getelementptr inbounds ([3 x i32], [3 x i32]* @c, i64
      0, i64 1), align 4 ; %11=c[1]
25  %12 = sitofp i32 %11 to float ; %11进行类型转换, int转float
26  %13 = fadd float -2.000000e+00, %12 ; %13=-k
27  %14 = load i32 , i32* getelementptr inbounds ([3 x i32], [3 x i32]* @c, i64
      0, i64 2), align 4
28  %15 = sitofp i32 %14 to float
29  %16 = fadd float %13, %15
30  ret float %16 ;return -k+c[1]+c[2];
31 }
32
33 ; 定义函数 int main()
34 define i32 @main() {
35   %1 = alloca i32, align 4
36   %2 = alloca i32, align 4
37   %3 = alloca i32, align 4
38   store i32 0, i32* %1, align 4 ; *%1=0
39   store i32 0, i32* %2, align 4 ; *%2=0;
40   %4 = call i32 @getint()
41   store i32 %4, i32* %3, align 4 ; *%3=getint();
42   br label %5
43
44 ; 实现while循环while(i<10){...}
45 5:
46   %6 = load i32 , i32* %2, align 4 ; 局部变量i=%6=0
47   %7 = icmp slt i32 %6, 10
48   ; 如果i<10, 则跳转则label%8, 否则跳转至lable%22
49   br i1 %7, label %8, label %22
50
51 ; label%8和label%22本质上是实现了if(b/a==0&& a>21) break 的条件分支
52 8:
53   %9 = load float , float* @a, align 4
54   %10 = fadd float %9, 3.000000e+00
55   store float %10, float* @a, align 4 ; a=a+3;
56   %11 = load float , float* @b, align 4 ; %11=b;

```

```

57  %12 = load float , float* @a, align 4 ; %12=a;
58  %13 = fdiv float %11, %12 ; %13=a/b
59  %14 = fcmp oeq float %13, 0.000000e+00
60  ; 如果满足a/b==0, 则跳转至label%15, 接着判断下一个条件, 否则跳转至label%19
61  br i1 %14, label %15, label %19
62
63  15:
64  %16 = load float , float* @a, align 4
65  %17 = fcmp ogt float %16, 2.100000e+01
66  ; 如果a>21, 则跳转至label%18, 否则跳转至label%19
67  br i1 %17, label %18, label %19
68
69  18:
70  ;实现break, 跳出while循环
71  br label %22
72
73  19:
74  %20 = load i32 , i32* %2, align 4
75  %21 = add nsw i32 %20, 1
76  store i32 %21, i32* %2, align 4 ; 实现i=i+1, 值存入%2
77  br label %5
78
79  22:
80  %23 = load i32 , i32* %3, align 4
81  %24 = call float @func(i32 %23) ; 调用函数func(%23)
82  %25 = fptosi float %24 to i32 ; %24进行类型转换, float转int
83  call void @putint(i32 %25) ; 调用函数putint(%25)
84
85  ret i32 0
86  }
87  ; 函数声明
88  declare i32 @getint()
89  declare void @putint(i32)

```

为便于理解, 将其转换成等价的 SysY 代码如下所示:

ex5.c

```

1  const float k=2.0;
2  float a=2.0,b=20.0;
3  int c[3]={1,2,3};
4  float func(int d)
5  {
6      c[1]=a*d;
7      c[2]=c[0]%d;
8      return -k+c[1]+c[2];
9  }
10 int main()
11 {
12     int i,d;

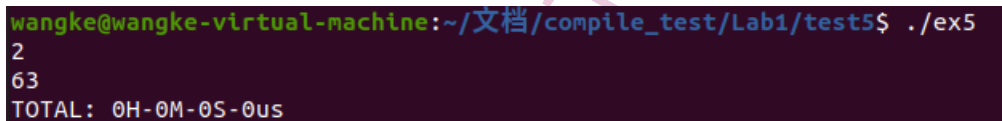
```

```

13     i=0;
14     d=getint();          //调用SysY库函数getint()
15     while(i<10)
16     {
17         a=a+3;
18         if(b/a==0&&a>21)
19         {
20             break;
21         }
22         i=i+1;
23     }
24     putint(func(d));      //调用SysY库函数putint(int)
25     return 0;
26 }

```

将该 IR 文件与 sylib.c 文件一起编译，执行命令 “clang ex5.ll sylib.c -o ex5” 即可得到可执行文件，令输入 d=2，可以看到命令行输出 func(d)=63，执行情况如下图所示：



```

wangke@wangke-virtual-machine:~/文档/compile_test/Lab1/test5$ ./ex5
2
63
TOTAL: 0H-0M-0S-0us

```

图 14: ex5 执行情况

各语言特性分析如下：

(1) 常量、变量的定义及声明：

所有的全局变量（常量）都以 @ 为前缀，const 表明它是一个常量，global 表明它是一个全局变量；以 % 开头的符号表示虚拟寄存器，可将其视作局部变量，使用 alloca 关键字可为其分配空间。与 SysY 相对应的数据类型为 i32 和 float。

(2) 数组的定义及访问：

以全局变量声明 c[3]=1,2,3 为例，其对应 IR 代码为 “@c = global [3 x i32] [i32 1, i32 2, i32 3], align 4”，其中 [3 x i32] 表示数组元素个数为 3，类型为 i32，后面三个中括号中的内容指明数组对应位置上的元素类型及值。访问数组元素值时需要使用 getelementptr 关键字，以访问 C[1] 为例，IR 代码为 “i32\* getelementptr inbounds ([3 x i32], [3 x i32]\* @c, i64 0, i64 1)”，[3 x i32] 表明将计算具有 3 个 i32 元素的数组的地址，基址为 @c，类型为 [3 x i32]\*，后面两个索引中，第二个是真正要访问的下标，getelementptr 返回一个地址，还需要用 i32\* 具体取到该地址上的值。

(3) 函数的定义及调用：

函数的定义格式为 “define” FuncType IDENT “(” [FuncFParams] “)” Block，其中 FuncType 表示函数返回类型，FuncFParams 表明形参列表，各形参用类型加变量名表示。函数的调用格式为 “call” FuncType IDENT “(” [FuncFParams] “)”，这边的 FuncFParams 表示实参列表。

(4) 连接 SysY 时库：

若需调用 SysY 库函数，则应在各函数体外使用 “declare” 关键字声明函数，其格式为 “declare” FuncType IDENT “(” [FuncFParamsType] “)”，FuncFParamsType 为形参类型的列表，调用方式同上。由于 SysY 库函数未直接在 C 语言中定义，因此该 IR 文件编译时需要与 sylib.c 一起编译。该程序调用了 “geiint” 和 “putint” 两个 SysY 函数实现输入输出。

(5) 语句 (while, break, if, return, 赋值):

while, break 和 if 语句常常结合各类表达式一起使用, 进一步的介绍可见 (6) 表示式部分。在 IR 代码中 “br” 关键字即可以用作无条件跳转, 如 “br label %5”, 用于跳转到地址标号 label%5 处; 也可以用作条件跳转, 如 “br i1 %17, label %18, label %19”, 其含义为: 若%17 为 True, 则跳转至 label%18, 否则跳转至 label%19。除此之外, 对于 while 循环中自增变量的设计, 需要有一个固定的地址来保存自增变量值。

return 语句使用关键字 “ret” 即可实现返回功能, 其格式为 “ret” ValType Value, ValType 为返回值类型, Value 为要返回的值。

对于赋值语句, 主要使用 “store” 和 “load” 关键字, 如 “store i32 %0, i32\* %2, align 4” 将参数%0 的值保存到地址%2 上, 而 “%3 = load float, float\* @a, align 4” 则将地址 @a 上保存的值赋值给变量%3。

(6) 表达式: 算术运算, 关系运算和逻辑运算:

IR 代码主要实现了五种算术运算: +, -, \*, /, %, 其对应的关键字分别为: add(fadd), sub(fsub), mul(fmul), div(fdiv), srem, 若有标记 nsw, 表明该算术操作没有符号回绕。

关系运算通过 icmp(fcmp) 来实现, 各种关系运算符有对应的关键字, 如小于号 (<) 对应 “slt”, 等于号 (==) 对应 “oeq”, 大于号 (>) 对应 “ogt”, 如 IR 代码 “%7 = icmp slt i32 %6, 10” 的含义为: 若%6 的值小于 10, 则返回 True 并赋值给%7, 反之返回 False。关系运算常常与分支跳转相结合。

对于逻辑运算, 程序中实现了 “&&” 与运算, 与运算并不显示表示, 而是将串联的两个关系表达式拆分开, 如 exp1 && exp2, 会先对 exp1 作条件判断, 再对 exp2 作条件判断, 共需要三个 label 标识的代码块。

(7) 隐式类型转换:

使用 “sitofp” 和 “fptosi” 关键字即可实现两种类型之间的隐式类型转换, 如 “sitofp i32 %4 to float” 将变量%4 由 i32 类型转为 float, “fptosi float %6 to i32” 将变量%6 由 float 类型转为 i32。

### 三、 总结

在本次实验中, 通过虚拟机平台对斐波拉契数列这个程序进行编译, 学习了编译各个阶段的过程。此外, 为了深刻理解编译器的功能, 做了以下探究:

- 验证预处理阶段的功能;
- 通过性能比较探究 O1 O3 和不同 pass 优化方式比较
- 通过可视化 CFG 探究了中间代码生成的多阶段的输出 (gcc)
- 通过反汇编, 对比 x86、arm 和 llvm 汇编和链接结果并得出结论。
- 融合函数、数组、隐式类型转换、9 种运算、“geint” 和 “putint” 等 SysY 编译器各语言特性编写 LLVM IR 程序, 并用 LLVM 编译成目标程序并成功执行验证。

在本次实验过程中, 还存有疑惑的问题:

- O1 O3 与不同 pass 优化之间的关系
- 初始代码 (.c 文件) 对于不同优化方式的影响