



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

编译原理实验报告

LLVM IR 编程

史文天 乔诣昊

年级：2021 级

专业：信息安全，计算机科学与技术

指导教师：王刚

2023 年 10 月 8 日

摘要

本次实验我们通过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。对编译预处理的功能进行了验证,探索了词法分析和语法分析的过程;通过 Graphviz 工具将 CFG 可视化分析了中间代码生成的多阶段;在代码优化阶段,实践了 O1 O3 和不同优化 pass 方法。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、多种运算、scanf、printf 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,多次运行并成功执行验证。

关键字: LLVM IR, SysY

目录

一、 问题描述	1
二、 实验平台	1
三、 实验分工	1
四、 实验过程	1
(一) 完整编译过程	1
(二) 预处理器	2
1. 预处理阶段功能	2
2. 验证过程及结果分析	2
(三) 编译器	2
1. 词法分析	2
2. 语法分析	3
3. 语义分析	3
4. 中间代码生成	4
5. 代码优化	4
6. 代码生成	5
(四) 汇编器	5
(五) 链接器	11
(六) LLVM IR 程序	18
五、 总结	24
六、 git 库	24

一、问题描述

以你熟悉的编译器，如 GCC、LLVM 等为研究对象，深入地探究语言处理系统的完整工作过程：

1. 完整的编译过程都有什么？
2. 预处理器做了什么？
3. 编译器做了什么？
4. 汇编器做了什么？
5. 链接器做了什么？
6. 通过编写 LLVM IR 程序，熟悉 LLVM IR 中间语言。并尽可能地对其实现方式有所了解。

二、实验平台

设备名称	具体型号
虚拟机	VMware Workstation Pro
操作系统	ubuntu-22.04.2

表 1: 实验平台

三、实验分工

- 史文天：预处理、编译器的分析，llvm 语言的特点分析、llvm 程序的设计。
- 乔诣昊：汇编器、链接器的分析，llvm 程序的实现、调试与测试。

四、实验过程

(一) 完整编译过程

1. **预编译**: 处理源代码中以 # 开始的预编译指令，例如展开所有宏定义、插入 #include 指向的文件等，以获得经过预处理的源程序。
2. **编译**: 将预处理器处理过的源程序文件翻译成为标准的汇编语言以供计算机阅读。
3. **汇编**: 将汇编语言指令翻译成机器语言指令，并将汇编语言程序打包成可重定位目标程序。
4. **链接加载**: 将可重定位的机器代码和相应的一些目标文件以及库文件链接在一起，形成真正能在机器上运行的目标机器代码。

一个 C 程序 hello.c，经历上述 4 个编译阶段最终生成可执行程序，具体流程如下：

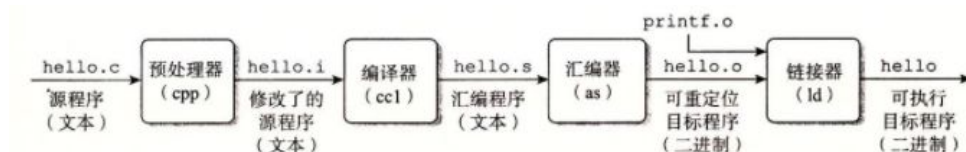


图 1: 代码编译流程图

(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

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

```
720
721 extern char *ctermid (char * __s) __attribute__ ((__nothrow__ , __leaf__))
722 | __attribute__ ((__access__ (__write_only__ , 1)));
723 # 867 "/usr/include/stdio.h" 3 4
724 extern void flockfile (FILE * __stream) __attribute__ ((__nothrow__ , __leaf__));
725
726
727
728 extern int ftrylockfile (FILE * __stream) __attribute__ ((__nothrow__ , __leaf__));
729
730
731 extern void funlockfile (FILE * __stream) __attribute__ ((__nothrow__ , __leaf__));
732 # 885 "/usr/include/stdio.h" 3 4
733 extern int __uflow (FILE *);
734 extern int __overflow (FILE *, int);
735 # 902 "/usr/include/stdio.h" 3 4
736
737 # 2 "main.c" 2
738
739 # 2 "main.c"
740 int main()
741 {
742     int a, b, i, t, n;
743     a = 0;
744     b = 1;
745     i = 1;
746     scanf("%d",n);
747     printf("%d\n%d\n",a,b);
748     while (i < n)
749     {
750         t = b;
751         b = a + b;
752         printf("%d\n",b);
753         a = t;
754         i = i + 1;
755     }
756 }
757
```

图 2: 预处理结果

(三) 编译器

1. 词法分析

利用 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：

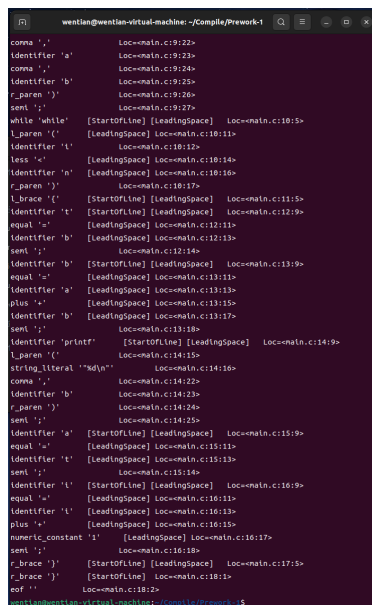


图 3: 词法分析结果

2. 语法分析

将词法分析生成的词法单元来构建抽象语法树。利用 clang -E -Xclang -ast-dump main.c 进行语法分析。

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：

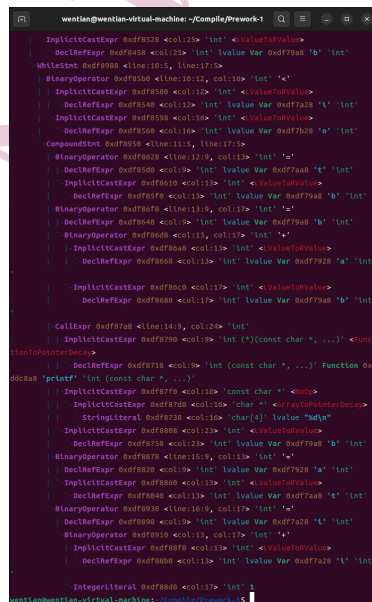


图 4: 语法分析结果

3. 语义分析

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。

4. 中间代码生成

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。可以清楚地看到块之间的逻辑关系。

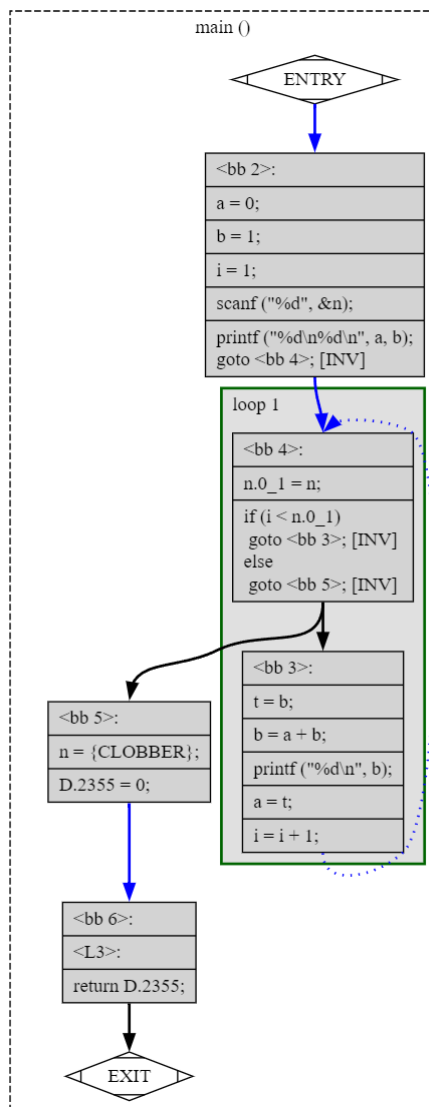


图 5: a-main.c.015t.cfg.dot

可以看到 CFG (控制流程图) 中 `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, 返回值只有编号, 通过数字标识区分不同阶段这些变量有不同值。

5. 代码优化

先使用指令 `llvm-as main.ll -o main.bc` 得到 LLVM IR 的二进制代码形式。

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。

之后使用其他 pass 模块进行优化, 但优化结果完全一样。推测可能是由于程序比较简单, 优化空间并不是非常大。

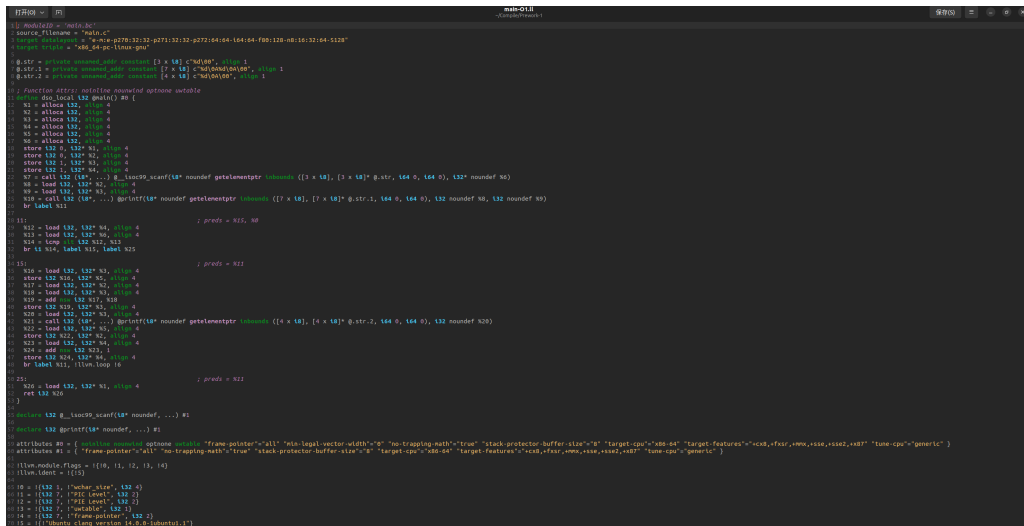


图 6: O1 级别优化

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。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。

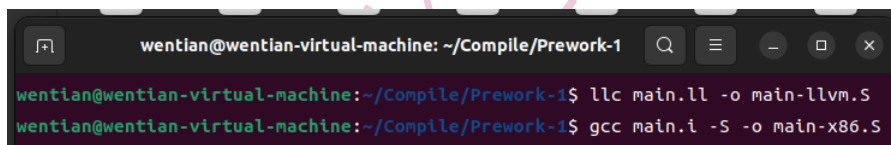


图 7: 代码生成

(四) 汇编器

汇编器的作用就是将汇编语言指令翻译成机器语言指令, 并将汇编语言程序打包成可重定位目标程序。其中每一个汇编语句几乎都对应一条机器指令。

x86 格式汇编可以直接用 `gcc` 完成汇编器的工作, 如使用下面的命令:

```
1 gcc main.S -c -o main.o
```

arm 格式汇编需要用到交叉编译, 如使用下面的命令:

```
1 arm-linux-gnueabi-gcc main.S -o main.o
```

LLVM 可以直接使用 `llc` 命令同时编译和汇编 LLVM bitcode:

```
1 llc test.bc -filetype=obj -o test.o
```

我们可以用 `xxd` 来阅读二进制文件, 由于文本量过大, 所以这里只放出部分的截图。

```

qlao@qlao-virtual-machine:~/桌面/x86$ xxd main.o
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF.....
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  ..>.....
00000020: 0000 0000 0000 0000 0004 0000 0000 0000  .....
00000030: 0000 0000 4000 0000 0000 4000 0e00 0d00  ....@.....@....
00000040: f30f 1efa 5548 89e5 4883 ec20 6448 8b04  ....UH..H.. dH..
00000050: 2528 0000 0048 8945 f831 c0c7 45e8 0000  %(...H.E.1..E...
00000060: 0000 c745 ec01 0000 00c7 45f0 0100 0000  ...E.....E....
00000070: 488d 45e4 4889 c648 8d05 0000 0000 4889  H.E.H..H.....H.
00000080: c7b8 0000 0000 e800 0000 008b 45e8 89c6  .....E...
00000090: 488d 0500 0000 0048 89c7 b800 0000 00e8  H.....H.....
000000a0: 0000 0000 8b45 ec89 c648 8d05 0000 0000  ....E...H.....
000000b0: 4889 c7b8 0000 0000 e800 0000 00eb 2f8b  H......./.
000000c0: 45ec 8945 f48b 45e8 0145 ec8b 45ec 89c6  E..E..E..E...
000000d0: 488d 0500 0000 0048 89c7 b800 0000 00e8  H.....H.....
000000e0: 0000 0000 8b45 f489 45e8 8345 f001 8b45  ....E..E..E...E
000000f0: e439 45f0 7cc9 b800 0000 0048 8b55 f864  .9E.|.....H.U.d
00000100: 482b 1425 2800 0000 7405 e800 0000 00c9  H+.%(...t.....
00000110: c325 6400 2564 0a00 0047 4343 3a20 2855  .%d.%d...GCC: (U
00000120: 6275 6e74 7520 3131 2e34 2e30 2d31 7562  buntu 11.4.0-1ub
00000130: 756e 7475 317e 3232 2e30 3429 2031 312e  untu1~22.04) 11.
00000140: 342e 3000 0000 0000 0400 0000 1000 0000  4.0.....
00000150: 0500 0000 474e 5500 0200 00c0 0400 0000  ....GNU.....
00000160: 0300 0000 0000 0000 1400 0000 0000 0000  .....

```

图 8: main.o

前面得到的 main.o 文件内容我们无法直接读懂，我们可以用下面的命令将前面得到的机器语言文件进行反汇编。由于篇幅原因，此处仅针对 x86 进行反汇编分析，下面的指令也是针对 x86 的。

```
1 objdump -d main.o > main-anti-obj.S
```

得到的反汇编代码如下：

反汇编代码

```

1 main.o:          文件格式 elf64-x86-64
2
3
4 Disassembly of section .text:
5
6 0000000000000000 <main>:
7   0:   f3 0f 1e fa          endbr64
8   4:   55                   push    %rbp
9   5:   48 89 e5             mov     %rsp,%rbp
10  8:   48 83 ec 20          sub     $0x20,%rsp
11  c:   64 48 8b 04 25 28 00  mov     %fs:0x28,%rax
12 13:  00 00
13 15:  48 89 45 f8          mov     %rax,-0x8(%rbp)

```



```

14 19: 31 c0 xor %eax,%eax
15 1b: c7 45 e8 00 00 00 00 movl $0x0,-0x18(%rbp)
16 22: c7 45 ec 01 00 00 00 movl $0x1,-0x14(%rbp)
17 29: c7 45 f0 01 00 00 00 movl $0x1,-0x10(%rbp)
18 30: 48 8d 45 e4 lea -0x1c(%rbp),%rax
19 34: 48 89 c6 mov %rax,%rsi
20 37: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # 3e <main+0x3e>
21 3e: 48 89 c7 mov %rax,%rdi
22 41: b8 00 00 00 00 mov $0x0,%eax
23 46: e8 00 00 00 00 call 4b <main+0x4b>
24 4b: 8b 45 e8 mov -0x18(%rbp),%eax
25 4e: 89 c6 mov %eax,%esi
26 50: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # 57 <main+0x57>
27 57: 48 89 c7 mov %rax,%rdi
28 5a: b8 00 00 00 00 mov $0x0,%eax
29 5f: e8 00 00 00 00 call 64 <main+0x64>
30 64: 8b 45 ec mov -0x14(%rbp),%eax
31 67: 89 c6 mov %eax,%esi
32 69: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # 70 <main+0x70>
33 70: 48 89 c7 mov %rax,%rdi
34 73: b8 00 00 00 00 mov $0x0,%eax
35 78: e8 00 00 00 00 call 7d <main+0x7d>
36 7d: eb 2f jmp ae <main+0xae>
37 7f: 8b 45 ec mov -0x14(%rbp),%eax
38 82: 89 45 f4 mov %eax,-0xc(%rbp)
39 85: 8b 45 e8 mov -0x18(%rbp),%eax
40 88: 01 45 ec add %eax,-0x14(%rbp)
41 8b: 8b 45 ec mov -0x14(%rbp),%eax
42 8e: 89 c6 mov %eax,%esi
43 90: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # 97 <main+0x97>
44 97: 48 89 c7 mov %rax,%rdi
45 9a: b8 00 00 00 00 mov $0x0,%eax
46 9f: e8 00 00 00 00 call a4 <main+0xa4>
47 a4: 8b 45 f4 mov -0xc(%rbp),%eax
48 a7: 89 45 e8 mov %eax,-0x18(%rbp)
49 aa: 83 45 f0 01 addl $0x1,-0x10(%rbp)
50 ae: 8b 45 e4 mov -0x1c(%rbp),%eax
51 b1: 39 45 f0 cmp %eax,-0x10(%rbp)
52 b4: 7c c9 jl 7f <main+0x7f>
53 b6: b8 00 00 00 00 mov $0x0,%eax
54 bb: 48 8b 55 f8 mov -0x8(%rbp),%rdx
55 bf: 64 48 2b 14 25 28 00 sub %fs:0x28,%rdx
56 c6: 00 00
57 c8: 74 05 je cf <main+0xcf>
58 ca: e8 00 00 00 00 call cf <main+0xcf>
59 cf: c9 leave
60 d0: c3 ret

```

首先分析一下这个反汇编代码，大致可以分为三部分，第 8 行到第 10 行为一些栈帧操作，第 11 行到第 52 行为主要函数部分，之后的为栈帧清理的结尾内容。

对于主要函数部分有：

- 指令 30-46: 调用一个函数，并存储返回值。
- 指令 4b-5f: 调用另一个函数，并存储返回值。
- 指令 64-78: 调用第三个函数，并存储返回值。
- 指令 7d: 跳转到指令 ae，类似于一个条件跳转。
- 指令 7f-b4: 循环执行一段代码，直到条件不满足，这里最后有一个比较明显的跳转。
- 指令 b6-bb: 返回最终结果。

通过分析代码我们可以知道这个代码就是我们写的源程序的汇编代码，不过是从机器码对应过来的。所以汇编器的功能就是将汇编指令翻译成机器语言指令，把这些指令打包成可重定位目标程序。

我们可以用下面的命令将机器语言文件的符号表信息导出。

```
1 nm main.o > main-nm-obj.txt
```

具体内容如下：

```
1          U __isoc99_scanf
2 0000000000000000 T main
3          U printf
4          U __stack_chk_fail
```

下面是对每个符号的解释：

- U __isoc99_scanf: 这是一个未定义的外部符号，表示在文件中有一个对 __isoc99_scanf 函数的引用。U 代表”未定义”。
- 0000000000000000 T main: 这是一个具有全局可见性的 main 符号，它被定义在文件中，并且是可执行文件的入口点。T 代表”文本段”。
- U printf: 这是一个未定义的外部符号，表示在文件中有一个对 printf 函数的引用。
- U __stack_chk_fail: 这是一个未定义的外部符号，表示在文件中有一个对 __stack_chk_fail 函数的引用。

此外为了更好的分析，我在这里将尝试改变源代码，取不同的小块来进行汇编和反汇编。

尝试 1 源代码

```
1 #include<stdio.h>
2 int main(){
3     return 0;
4 }
```

尝试 1 结果

```

1
2 main2.o:      文件格式 elf64-x86-64
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <main>:
8   0:   f3 0f 1e fa                endbr64
9   4:   55                          push   %rbp
10  5:   48 89 e5                     mov     %rsp,%rbp
11  8:   b8 00 00 00 00              mov     $0x0,%eax
12 d:   5d                          pop     %rbp
13 e:   c3                          ret

```

这里就是简单看一下一个空的主函数是什么样的，结果就是栈的调整。

尝试 2 源代码

```

1 #include<stdio.h>
2 int main(){
3     int i;
4     i=1;
5     return 0;
6 }

```

尝试 2 结果

```

1
2 main2.o:      文件格式 elf64-x86-64
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <main>:
8   0:   f3 0f 1e fa                endbr64
9   4:   55                          push   %rbp
10  5:   48 89 e5                     mov     %rsp,%rbp
11  8:   c7 45 fc 01 00 00 00        movl    $0x1,-0x4(%rbp)
12 f:   b8 00 00 00 00              mov     $0x0,%eax
13 14:   5d                          pop     %rbp
14 15:   c3                          ret

```

然后看一下定义变量与赋值，这里主要变化就是用到了 movl。

尝试 3 源代码

```

1 #include<stdio.h>
2 int main(){
3     int i,n;
4     scanf("%d", &n);

```

```

5      i=1;
6      while (i < n)
7      {
8          i = i + 1;
9      }
10     return 0;
11 }

```

尝试 3 结果

```

1
2 main2.o:      文件格式 elf64-x86-64
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <main>:
8   0:   f3 0f 1e fa      endbr64
9   4:   55              push   %rbp
10  5:   48 89 e5         mov    %rsp,%rbp
11  8:   48 83 ec 10      sub    $0x10,%rsp
12  c:   64 48 8b 04 25 28 00  mov    %fs:0x28,%rax
13 13:  00 00
14 15:  48 89 45 f8      mov    %rax,-0x8(%rbp)
15 19:  31 c0            xor    %eax,%eax
16 1b:  48 8d 45 f0      lea    -0x10(%rbp),%rax
17 1f:  48 89 c6         mov    %rax,%rsi
18 22:  48 8d 05 00 00 00 00  lea    0x0(%rip),%rax      # 29 <main+0x29>
19 29:  48 89 c7         mov    %rax,%rdi
20 2c:  b8 00 00 00 00  mov    $0x0,%eax
21 31:  e8 00 00 00 00  call   36 <main+0x36>
22 36:  c7 45 f4 01 00 00 00  movl   $0x1,-0xc(%rbp)
23 3d:  eb 04           jmp    43 <main+0x43>
24 3f:  83 45 f4 01      addl   $0x1,-0xc(%rbp)
25 43:  8b 45 f0         mov    -0x10(%rbp),%eax
26 46:  39 45 f4         cmp    %eax,-0xc(%rbp)
27 49:  7c f4           jl     3f <main+0x3f>
28 4b:  b8 00 00 00 00  mov    $0x0,%eax
29 50:  48 8b 55 f8      mov    -0x8(%rbp),%rdx
30 54:  64 48 2b 14 25 28 00  sub    %fs:0x28,%rdx
31 5b:  00 00
32 5d:  74 05           je     64 <main+0x64>
33 5f:  e8 00 00 00 00  call   64 <main+0x64>
34 64:  c9             leave
35 65:  c3             ret

```

最后看一下 scanf 和循环，scanf 那里使用了 call，至于循环部分就是值的比较和跳转。

(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。

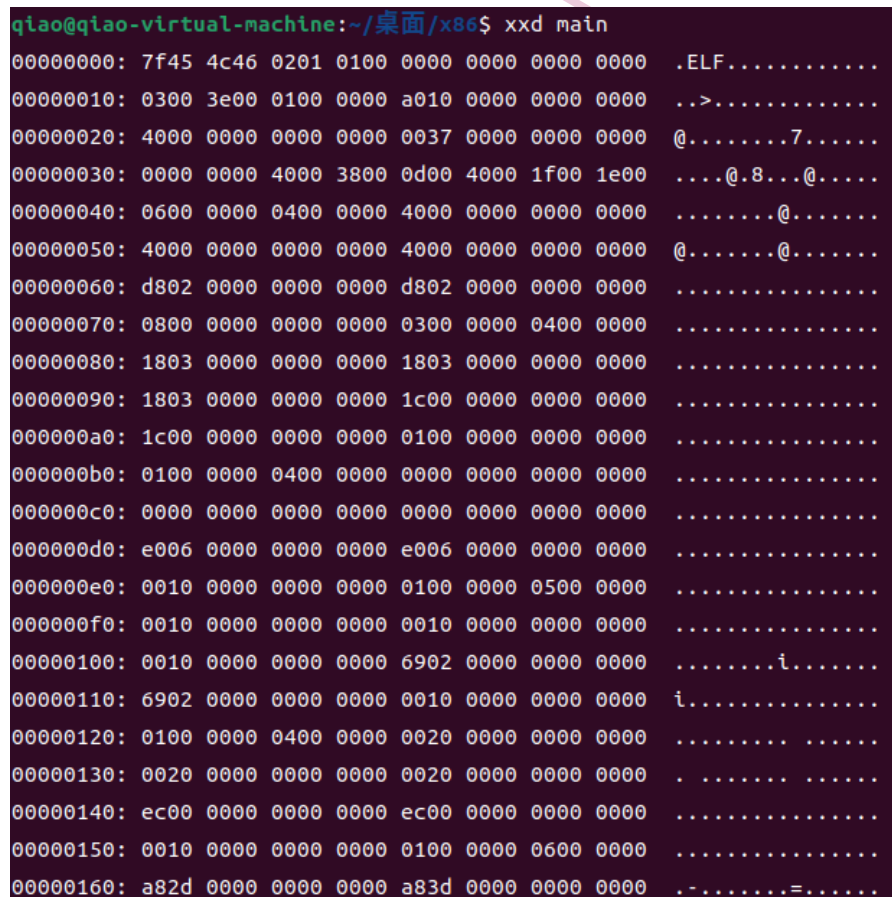
链接器主要工作为：

- 去项目文件里查找目标代码文件里没有定义的变量。
- 扫描项目中的不同文件，将所有符号定义和引用地址收集起来，并放到全局符号表中。
- 计算合并后长度及位置，生成同类型的段进行合并，建立绑定。
- 对项目不同文件里的变量进行地址重定位。

我们可以执行下面的命令便可用机器代码生成可执行文件。

```
1 gcc main.o -o main
```

我们同样使用 xxd 来阅读二进制文件，由于文本量过大，所以这里只放出部分的截图。



```
qiao@qiao-virtual-machine:~/桌面/x86$ xxd main
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF.....
00000010: 0300 3e00 0100 0000 a010 0000 0000 0000  ..>.....
00000020: 4000 0000 0000 0000 0037 0000 0000 0000  @.....7....
00000030: 0000 0000 4000 3800 0d00 4000 1f00 1e00  ....@.8...@....
00000040: 0600 0000 0400 0000 4000 0000 0000 0000  .....@.....
00000050: 4000 0000 0000 0000 4000 0000 0000 0000  @.....@.....
00000060: d802 0000 0000 0000 d802 0000 0000 0000  .....
00000070: 0800 0000 0000 0000 0300 0000 0400 0000  .....
00000080: 1803 0000 0000 0000 1803 0000 0000 0000  .....
00000090: 1803 0000 0000 0000 1c00 0000 0000 0000  .....
000000a0: 1c00 0000 0000 0000 0100 0000 0000 0000  .....
000000b0: 0100 0000 0400 0000 0000 0000 0000 0000  .....
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000  .....
000000d0: e006 0000 0000 0000 e006 0000 0000 0000  .....
000000e0: 0010 0000 0000 0000 0100 0000 0500 0000  .....
000000f0: 0010 0000 0000 0000 0010 0000 0000 0000  .....
00000100: 0010 0000 0000 0000 6902 0000 0000 0000  .....i.....
00000110: 6902 0000 0000 0000 0010 0000 0000 0000  i.....
00000120: 0100 0000 0400 0000 0020 0000 0000 0000  .....
00000130: 0020 0000 0000 0000 0020 0000 0000 0000  . ....
00000140: ec00 0000 0000 0000 ec00 0000 0000 0000  .....
00000150: 0010 0000 0000 0000 0100 0000 0600 0000  .....
00000160: a82d 0000 0000 0000 a83d 0000 0000 0000  .-.....=.....
```

图 9: main

当然我们也无法直接看懂最后得到的可执行文件，我们可以用下面的命令将前面得到的机器语言文件进行反汇编。

```
1 objdump -d main > main-anti-exe.S
```

得到的反汇编代码如下

反汇编代码

```

1  main:      文件格式 elf64-x86-64
2
3
4  Disassembly of section .init:
5
6  0000000000001000 <__init>:
7      1000:      f3 0f 1e fa      endbr64
8      1004:      48 83 ec 08      sub     $0x8,%rsp
9      1008:      48 8b 05 d9 2f 00 00  mov     0x2fd9(%rip),%rax      # 3
          fe8 <__gmon_start__@Base>
10     100f:      48 85 c0      test    %rax,%rax
11     1012:      74 02      je      1016 <__init+0x16>
12     1014:      ff d0      call    *%rax
13     1016:      48 83 c4 08      add     $0x8,%rsp
14     101a:      c3      ret
15
16  Disassembly of section .plt:
17
18  0000000000001020 <.plt>:
19     1020:      ff 35 8a 2f 00 00      push    0x2f8a(%rip)      # 3fb0 <
          _GLOBAL_OFFSET_TABLE_+0x8>
20     1026:      f2 ff 25 8b 2f 00 00      bnd jmp *0x2f8b(%rip)      # 3fb8 <
          _GLOBAL_OFFSET_TABLE_+0x10>
21     102d:      0f 1f 00      nopl    (%rax)
22     1030:      f3 0f 1e fa      endbr64
23     1034:      68 00 00 00 00      push    $0x0
24     1039:      f2 e9 e1 ff ff ff      bnd jmp 1020 <__init+0x20>
25     103f:      90      nop
26     1040:      f3 0f 1e fa      endbr64
27     1044:      68 01 00 00 00      push    $0x1
28     1049:      f2 e9 d1 ff ff ff      bnd jmp 1020 <__init+0x20>
29     104f:      90      nop
30     1050:      f3 0f 1e fa      endbr64
31     1054:      68 02 00 00 00      push    $0x2
32     1059:      f2 e9 c1 ff ff ff      bnd jmp 1020 <__init+0x20>
33     105f:      90      nop
34
35  Disassembly of section .plt.got:
36
37  0000000000001060 <__cxa_finalize@plt>:
38     1060:      f3 0f 1e fa      endbr64
39     1064:      f2 ff 25 8d 2f 00 00      bnd jmp *0x2f8d(%rip)      # 3ff8 <
          __cxa_finalize@GLIBC_2.2.5>
40     106b:      0f 1f 44 00 00      nopl    0x0(%rax,%rax,1)
41

```

```

42 Disassembly of section .plt.sec:
43
44 0000000000001070 <__stack_chk_fail@plt>:
45   1070:      f3 0f 1e fa          endbr64
46   1074:      f2 ff 25 45 2f 00 00    bnd jmp *0x2f45(%rip)      # 3fc0 <
      __stack_chk_fail@GLIBC_2.4>
47   107b:      0f 1f 44 00 00          nopl  0x0(%rax,%rax,1)
48
49 0000000000001080 <printf@plt>:
50   1080:      f3 0f 1e fa          endbr64
51   1084:      f2 ff 25 3d 2f 00 00    bnd jmp *0x2f3d(%rip)      # 3fc8 <
      printf@GLIBC_2.2.5>
52   108b:      0f 1f 44 00 00          nopl  0x0(%rax,%rax,1)
53
54 0000000000001090 <__isoc99_scanf@plt>:
55   1090:      f3 0f 1e fa          endbr64
56   1094:      f2 ff 25 35 2f 00 00    bnd jmp *0x2f35(%rip)      # 3fd0 <
      __isoc99_scanf@GLIBC_2.7>
57   109b:      0f 1f 44 00 00          nopl  0x0(%rax,%rax,1)
58
59 Disassembly of section .text:
60
61 00000000000010a0 <_start>:
62   10a0:      f3 0f 1e fa          endbr64
63   10a4:      31 ed                xor    %ebp,%ebp
64   10a6:      49 89 d1             mov    %rdx,%r9
65   10a9:      5e                  pop    %rsi
66   10aa:      48 89 e2             mov    %rsp,%rdx
67   10ad:      48 83 e4 f0          and    $0xfffffffffffff0,%rsp
68   10b1:      50                  push   %rax
69   10b2:      54                  push   %rsp
70   10b3:      45 31 c0             xor    %r8d,%r8d
71   10b6:      31 c9                xor    %ecx,%ecx
72   10b8:      48 8d 3d ca 00 00 00    lea    0xca(%rip),%rdi      # 1189
      <main>
73   10bf:      ff 15 13 2f 00 00      call   *0x2f13(%rip)      # 3fd8 <
      __libc_start_main@GLIBC_2.34>
74   10c5:      f4                  hlt
75   10c6:      66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
76   10cd:      00 00 00
77
78 00000000000010d0 <deregister_tm_clones>:
79   10d0:      48 8d 3d 39 2f 00 00    lea    0x2f39(%rip),%rdi    #
      4010 <__TMC_END__>
80   10d7:      48 8d 05 32 2f 00 00    lea    0x2f32(%rip),%rax    #
      4010 <__TMC_END__>
81   10de:      48 39 f8             cmp    %rdi,%rax
82   10e1:      74 15             je     10f8 <deregister_tm_clones+0

```

```

x28>
83 10e3:    48 8b 05 f6 2e 00 00    mov     0x2ef6(%rip),%rax        # 3
    fe0 <_ITM_deregisterTMCloneTable@Base>
84 10ea:    48 85 c0                test    %rax,%rax
85 10ed:    74 09                  je      10f8 <deregister_tm_clones+0
    x28>
86 10ef:    ff e0                  jmp     *%rax
87 10f1:    0f 1f 80 00 00 00 00    nopl    0x0(%rax)
88 10f8:    c3                    ret
89 10f9:    0f 1f 80 00 00 00 00    nopl    0x0(%rax)
90
91 0000000000001100 <register_tm_clones>:
92 1100:    48 8d 3d 09 2f 00 00    lea     0x2f09(%rip),%rdi        #
    4010 <_TMC_END_>
93 1107:    48 8d 35 02 2f 00 00    lea     0x2f02(%rip),%rsi        #
    4010 <_TMC_END_>
94 110e:    48 29 fe              sub     %rdi,%rsi
95 1111:    48 89 f0              mov     %rsi,%rax
96 1114:    48 c1 ee 3f          shr     $0x3f,%rsi
97 1118:    48 c1 f8 03          sar     $0x3,%rax
98 111c:    48 01 c6              add     %rax,%rsi
99 111f:    48 d1 fe              sar     %rsi
100 1122:    74 14                  je      1138 <register_tm_clones+0x38>
101 1124:    48 8b 05 c5 2e 00 00    mov     0x2ec5(%rip),%rax        # 3
    ff0 <_ITM_registerTMCloneTable@Base>
102 112b:    48 85 c0              test    %rax,%rax
103 112e:    74 08                  je      1138 <register_tm_clones+0x38>
104 1130:    ff e0                  jmp     *%rax
105 1132:    66 0f 1f 44 00 00      nopw    0x0(%rax,%rax,1)
106 1138:    c3                    ret
107 1139:    0f 1f 80 00 00 00 00    nopl    0x0(%rax)
108
109 0000000000001140 <__do_global_dtors_aux>:
110 1140:    f3 0f 1e fa          endbr64
111 1144:    80 3d c5 2e 00 00 00    cmpb    $0x0,0x2ec5(%rip)        #
    4010 <_TMC_END_>
112 114b:    75 2b                  jne     1178 <__do_global_dtors_aux+0
    x38>
113 114d:    55                    push    %rbp
114 114e:    48 83 3d a2 2e 00 00    cmpq    $0x0,0x2ea2(%rip)        # 3
    ff8 <__cxa_finalize@GLIBC_2.2.5>
115 1155:    00
116 1156:    48 89 e5              mov     %rsp,%rbp
117 1159:    74 0c                  je      1167 <__do_global_dtors_aux+0
    x27>
118 115b:    48 8b 3d a6 2e 00 00    mov     0x2ea6(%rip),%rdi        #
    4008 <__dso_handle>
119 1162:    e8 f9 fe ff ff        call    1060 <__cxa_finalize@plt>

```



```

120      1167:      e8 64 ff ff ff      call    10d0 <deregister_tm_clones>
121      116c:      c6 05 9d 2e 00 00 01  movb    $0x1,0x2e9d(%rip)      #
      4010 <__TMC_END__>
122      1173:      5d                    pop     %rbp
123      1174:      c3                    ret
124      1175:      0f 1f 00              nopl    (%rax)
125      1178:      c3                    ret
126      1179:      0f 1f 80 00 00 00 00  nopl    0x0(%rax)
127
128      0000000000001180 <frame_dummy>:
129      1180:      f3 0f 1e fa          endbr64
130      1184:      e9 77 ff ff ff      jmp     1100 <register_tm_clones>
131
132      0000000000001189 <main>:
133      1189:      f3 0f 1e fa          endbr64
134      118d:      55                    push    %rbp
135      118e:      48 89 e5              mov     %rsp,%rbp
136      1191:      48 83 ec 20          sub     $0x20,%rsp
137      1195:      64 48 8b 04 25 28 00  mov     %fs:0x28,%rax
138      119c:      00 00
139      119e:      48 89 45 f8          mov     %rax,-0x8(%rbp)
140      11a2:      31 c0                xor     %eax,%eax
141      11a4:      c7 45 e8 00 00 00 00  movl    $0x0,-0x18(%rbp)
142      11ab:      c7 45 ec 01 00 00 00  movl    $0x1,-0x14(%rbp)
143      11b2:      c7 45 f0 01 00 00 00  movl    $0x1,-0x10(%rbp)
144      11b9:      48 8d 45 e4          lea     -0x1c(%rbp),%rax
145      11bd:      48 89 c6              mov     %rax,%rsi
146      11c0:      48 8d 05 3d 0e 00 00  lea     0xe3d(%rip),%rax      # 2004
      <__IO_stdin_used+0x4>
147      11c7:      48 89 c7              mov     %rax,%rdi
148      11ca:      b8 00 00 00 00      mov     $0x0,%eax
149      11cf:      e8 bc fe ff ff      call    1090 <__isoc99_scanf@plt>
150      11d4:      8b 45 e8              mov     -0x18(%rbp),%eax
151      11d7:      89 c6                mov     %eax,%esi
152      11d9:      48 8d 05 27 0e 00 00  lea     0xe27(%rip),%rax      # 2007
      <__IO_stdin_used+0x7>
153      11e0:      48 89 c7              mov     %rax,%rdi
154      11e3:      b8 00 00 00 00      mov     $0x0,%eax
155      11e8:      e8 93 fe ff ff      call    1080 <printf@plt>
156      11ed:      8b 45 ec              mov     -0x14(%rbp),%eax
157      11f0:      89 c6                mov     %eax,%esi
158      11f2:      48 8d 05 0e 0e 00 00  lea     0xe0e(%rip),%rax      # 2007
      <__IO_stdin_used+0x7>
159      11f9:      48 89 c7              mov     %rax,%rdi
160      11fc:      b8 00 00 00 00      mov     $0x0,%eax
161      1201:      e8 7a fe ff ff      call    1080 <printf@plt>
162      1206:      eb 2f                jmp     1237 <main+0xae>
163      1208:      8b 45 ec              mov     -0x14(%rbp),%eax

```

```

164 120b:      89 45 f4      mov    %eax,-0xc(%rbp)
165 120e:      8b 45 e8      mov    -0x18(%rbp),%eax
166 1211:      01 45 ec      add    %eax,-0x14(%rbp)
167 1214:      8b 45 ec      mov    -0x14(%rbp),%eax
168 1217:      89 c6        mov    %eax,%esi
169 1219:      48 8d 05 e7 0d 00 00 lea     0xde7(%rip),%rax      # 2007
    <_IO_stdin_used+0x7>
170 1220:      48 89 c7      mov    %rax,%rdi
171 1223:      b8 00 00 00 00 mov    $0x0,%eax
172 1228:      e8 53 fe ff ff call    1080 <printf@plt>
173 122d:      8b 45 f4      mov    -0xc(%rbp),%eax
174 1230:      89 45 e8      mov    %eax,-0x18(%rbp)
175 1233:      83 45 f0 01   addl    $0x1,-0x10(%rbp)
176 1237:      8b 45 e4      mov    -0x1c(%rbp),%eax
177 123a:      39 45 f0      cmp    %eax,-0x10(%rbp)
178 123d:      7c c9        jl     1208 <main+0x7f>
179 123f:      b8 00 00 00 00 mov    $0x0,%eax
180 1244:      48 8b 55 f8   mov    -0x8(%rbp),%rdx
181 1248:      64 48 2b 14 25 28 00 sub     %fs:0x28,%rdx
182 124f:      00 00
183 1251:      74 05        je     1258 <main+0xcf>
184 1253:      e8 18 fe ff ff call    1070 <__stack_chk_fail@plt>
185 1258:      c9          leave
186 1259:      c3          ret
187
188 Disassembly of section .fini:
189
190 000000000000125c <_fini>:
191 125c:      f3 0f 1e fa   endbr64
192 1260:      48 83 ec 08   sub     $0x8,%rsp
193 1264:      48 83 c4 08   add     $0x8,%rsp
194 1268:      c3          ret

```

通过阅读上面得到的反汇编代码，我们发现了 main 的反汇编代码远多于 main.o 的反汇编代码。此外也变得更为规整了，分了好多的 section，并且前面 main.o 的反汇编代码全部出现在了 section.text 的 0000000000001189 <main> 这里。

- section.init：这个部分包含了程序的初始化代码。当程序加载到内存并开始执行时，初始化代码负责执行一系列初始化任务，例如设置全局变量、初始化库、分配资源等。
- section.plt：这是一个过程链接表，用于支持动态链接和函数调用。它存储了函数指针和相关的跳转指令，以便在运行时正确地将控制流传递给目标函数。
- section.plt.got：这是一个全局偏移表，用于实现动态链接和符号解析。它存储了全局符号的地址，以便在运行时进行符号解析和重定位。
- section.text：这是存放可执行代码的部分，也被称为代码段。它包含了程序的实际执行指令，例如函数的操作指令、控制流语句等。这是程序的主要执行部分。

- section.fini: 这个部分包含了程序的终止代码。当程序执行完毕时, 终止代码负责执行清理任务, 例如释放资源、关闭文件等。

当链接器进行链接的时候, 首先决定各个目标文件在最终可执行文件里的位置。然后访问所有目标文件的地址重定义表, 对其中记录的地址进行重定向。然后遍历所有目标文件的未解决符号表, 并且在所有的导出符号表里查找匹配的符号, 并在未解决符号表中所记录的位置上填写实现地址。最后把所有的目标文件的内容写在各自的位置上, 再作一些另的工作, 就生成一个可执行文件。这里是就一个 main.o 文件, 没有其它的手写文件。

对于库文件, 通常存在两种类型: 静态库和动态库。

对于静态库, 链接器会将库文件的对象代码直接合并到最终的可执行文件中。这意味着最终的可执行文件将包含静态库中所有被使用的函数和数据的副本。静态库的文件扩展名通常为.a (在 Windows 上为.lib)。

对于动态库, 链接器会在最终的可执行文件中创建对库文件的引用。在程序运行时, 操作系统会根据需要加载和链接动态库。这意味着最终的可执行文件大小较小, 因为它只包含对库文件的引用。动态库的文件扩展名通常为.so (在 Windows 上为.dll)。

看完反汇编代码, 我们可以用下面的命令将机器语言文件的符号表信息导出。

```
1 nm main > main-nm-exe.txt
```

具体内容如下:

```
1 000000000000038c r __abi_tag
2 0000000000004010 B __bss_start
3 0000000000004010 b completed.0
4          w __cxa_finalize@GLIBC_2.2.5
5 0000000000004000 D __data_start
6 0000000000004000 W data_start
7 00000000000010d0 t deregister_tm_clones
8 0000000000001140 t __do_global_dtors_aux
9 0000000000003db0 d __do_global_dtors_aux_fini_array_entry
10 0000000000004008 D __dso_handle
11 0000000000003db8 d _DYNAMIC
12 0000000000004010 D _edata
13 0000000000004018 B _end
14 000000000000125c T _fini
15 0000000000001180 t frame_dummy
16 0000000000003da8 d __frame_dummy_init_array_entry
17 00000000000020e8 r __FRAME_END__
18 0000000000003fa8 d _GLOBAL_OFFSET_TABLE_
19          w __gmon_start__
20 000000000000200c r __GNU_EH_FRAME_HDR
21 0000000000001000 T _init
22 0000000000002000 R _IO_stdin_used
23          U __isoc99_scanf@GLIBC_2.7
24          w _ITM_deregisterTMCloneTable
25          w _ITM_registerTMCloneTable
26          U __libc_start_main@GLIBC_2.34
27 0000000000001189 T main
28          U printf@GLIBC_2.2.5
```

```

29 0000000000001100 t register_tm_clones
30      U __stack_chk_fail@GLIBC_2.4
31 00000000000010a0 T _start
32 0000000000004010 D __TMC_END__

```

因为这里的符号比较多，所以选取一部分符号进行解释：

- 0000000000004010 B __bss_start：未初始化的全局变量开始位置，存放在 BSS 段。
- 0000000000004010 b completed.0：未初始化的静态变量 completed.0，存放在 BSS 段。
- 0000000000004000 D __data_start：已初始化的全局变量开始位置，存放在数据段(.data)。
- 000000000000125c T _fini：程序的终止函数 _fini。
- 0000000000001000 T _init：程序的初始化函数 _init。
- 0000000000001189 T main：程序的主函数 main。
- 00000000000010a0 T _start：程序的起始函数 _start。

这里最后尝试了一下在链接的时候加入-static，再进行反汇编，但是最后得到的结果却过于庞大，这里只给出部分截图：

```

182943 4b5a97:      74 89      je      4b5a22 <free_mem+0x272>
182944 4b5a99:      48 89 c2    mov     %rax,%rdx
182945 4b5a9c:      48 c1 e2 04 shl     $0x4,%rdx
182946 4b5aa0:      49 83 7c 16 18 00 cmpq    $0x0,0x18(%r14,%rdx,1)
182947 4b5aa6:      74 e8      je      4b5a90 <free_mem+0x2e0>
182948 4b5aa8:      e9 47 fe ff ff jmp     4b58f4 <free_mem+0x144>
182949
182950 Disassembly of section .fini:
182951
182952 0000000000004b5ab0 <_fini>:
182953 4b5ab0:      f3 0f 1e fa endbr64
182954 4b5ab4:      48 83 ec 08 sub     $0x8,%rsp
182955 4b5ab8:      48 83 c4 08 add     $0x8,%rsp
182956 4b5abc:      c3        ret

```

图 10: 部分截图

可以观察到代码达到了 182956 行，查阅资料得知，选项 -static 表示要进行静态链接。在默认情况下，GCC 生成的可执行文件使用动态链接，在运行时需要依赖系统中已安装的共享库。而使用 -static 选项后，GCC 将会尝试将所有依赖项静态链接到可执行文件中，从而在运行时不再需要外部的共享库。静态链接的优点是可执行文件独立于系统环境，更加可移植。然而，由于所有依赖项都被打包到可执行文件中，所以生成的可执行文件可能会较大。

(六) LLVM IR 程序

我们在这里编写的程序的大致流程为：让用户输出一个数（大于等于 0），然后对这个数进行 5 的取模，并将取模结果作为索引，到数组中找对应的数为走的楼梯数，然后调用函数求解有多少种走法（就是经典的走楼梯问题，一次走一步或两步，有多少种走法），最后输出两行，一行为楼梯数，一行为走法数。

例如我输入 2，用 5 取模后为 2，数组索引为 2 的值为 3，所以走 3 阶楼梯，一共有 3 种走法，所以最后会输出两行 3。

我们编写的 LLVM IR 程序代码如下，已经按 LLVM IR 的方式写了详细的注释：

LLVM IR 程序

```

1 ;声明printf函数
2 declare i32 @printf(i8*, ...)
3 ;声明scanf函数
4 declare i32 @scanf(i8*, ...)
5
6 ;用于printf函数
7 @.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
8 ;用于scanf函数
9 @.str2 = private unnamed_addr constant [3 x i8] c"%d\00", align 1
10
11 ;定义全局变量a为5
12 @a = global i32 5
13 ;定义全局变量数组b, 分别为1,2,3,4,5
14 @b = global [5 x i32] [i32 1, i32 2, i32 3, i32 4, i32 5]
15
16 ;这个是走楼梯的函数
17 define i32 @stairs(i32 %n) {
18   entry:
19     ;首先比较n和1,如果n小于等于1,就跳到if.then, 否则跳到if.else
20     %cmp = icmp sle i32 %n, 1
21     br i1 %cmp, label %if.then, label %if.else
22
23   if.then:
24     ;返回1
25     ret i32 1
26
27   if.else:
28     ;首先给pp, p, c, i分配空间, 都是4字节
29     %pp = alloca i32, align 4
30     %p = alloca i32, align 4
31     %c = alloca i32, align 4
32     %i = alloca i32, align 4
33     ;将pp, p, c, i分别赋值为0,1,0,1
34     store i32 0, i32* %pp, align 4
35     store i32 1, i32* %p, align 4
36     store i32 0, i32* %c, align 4
37     store i32 1, i32* %i, align 4
38     ;跳到while.cond
39     br label %while.cond
40
41 ;这里是循环条件
42   while.cond:
43     ;%i的值给%0不断和%n比较, 小于就跳到while.body, 否则跳到while.end
44     %0 = load i32, i32* %i, align 4
45     %1 = icmp sle i32 %0, %n
46     br i1 %1, label %while.body, label %while.end
47

```

```

48 ;这里是循环体
49 while.body:
50 ;%pp给%2,%p给%3
51 %2 = load i32, i32* %pp, align 4
52 %3 = load i32, i32* %p, align 4
53 ;%4为%2和%3相加
54 %4 = add i32 %2, %3
55 ;%4的值给%c, %3的值给%pp, %4的值给%p
56 store i32 %4, i32* %c, align 4
57 store i32 %3, i32* %pp, align 4
58 store i32 %4, i32* %p, align 4
59 ;%5取得i的值, 通过%6实现i++
60 %5 = load i32, i32* %i, align 4
61 %6 = add i32 %5, 1
62 store i32 %6, i32* %i, align 4
63 ;跳到while.cond
64 br label %while.cond
65
66 ;循环结束
67 while.end:
68 ;将%c的值给%ans
69 %ans = load i32, i32* %c, align 4
70 ;跳到if.end
71 br label %if.end
72
73 if.end:
74 ;返回%ans
75 ret i32 %ans
76 }
77
78 ;主函数
79 define i32 @main() {
80 entry:
81 ;给num分配空间, 四字节
82 %num = alloca i32, align 4
83 ;%0得到str的首地址, 之后调用scanf来获取num的值
84 %0 = getelementptr [3 x i8], [3 x i8]* @.str2, i32 0, i32 0
85 %1 = call i32 @i8*, ... @scanf(i8* %0, i32* %num)
86 ;给ans分配空间, 四字节
87 %ans = alloca i32, align 4
88 ;将num的值给%2
89 %2 = load i32, i32* %num, align 4
90 ;将a的值给%3,因为是全局变量, 所以加*
91 %3 = load i32, i32* @a, align 4
92 ;srem是取模运算, 这里相当于把num%a的值给了%4
93 %4 = srem i32 %2, %3
94 ;取得了b[%4]的指针, 即b[num%a]的指针, 最后将地址给%5
95 %5 = getelementptr [5 x i32], [5 x i32]* @b, i32 0, i32 %4

```

```

96 ;%6取得了%5处的值
97 %6 = load i32, i32* %5, align 4
98 ;调用函数, 结果给%7
99 %7 = call i32 @stairs(i32 %6)
100 ;将结果给ans
101 store i32 %7, i32* %ans, align 4
102 ;ans给%8
103 %8 = load i32, i32* %ans, align 4
104 ;输出走多少的台阶
105 %9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
    i8]* @.str, i32 0, i32 0), i32 %6)
106 ;输出有多少种走法
107 %10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4
    x i8]* @.str, i32 0, i32 0), i32 %8)
108 ret i32 0
109 }

```

为了更加直观地观察理解代码, 将其转化为等价的 SysY 代码, 结果如下:

LLVM IR 程序 (ex.ll)

```

1 // 声明printf函数
2 extern int printf(string format, ...);
3 // 声明scanf函数
4 extern int scanf(string format, ...);
5
6 // 定义全局变量a为5
7 int a = 5;
8 // 定义全局变量数组b, 分别为1,2,3,4,5
9 int b[5] = {1, 2, 3, 4, 5};
10
11 // 这个是走楼梯的函数
12 int stairs(int n) {
13     if (n <= 1) {
14         return 1;
15     }
16     else {
17         int pp = 0;
18         int p = 1;
19         int c = 0;
20         int i = 1;
21         while (i <= n) {
22             c = pp + p;
23             pp = p;
24             p = c;
25             i = i + 1;
26         }
27         return c;
28     }
29 }

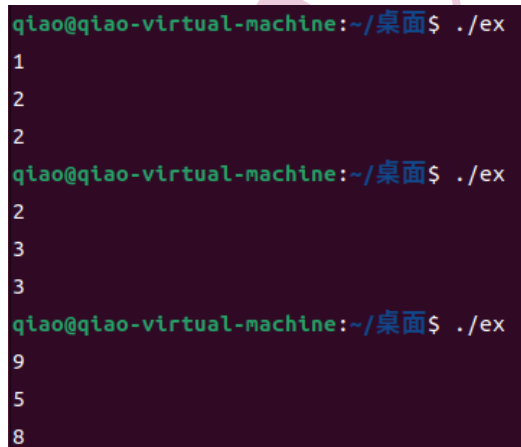
```

```
30
31 // 主函数
32 int main() {
33     int num;
34     scanf("%d", &num);
35     int ans;
36     num = num % a;
37     ans = stairs(b[num]);
38
39     printf("%d\n", b[num]); // 输出走多少的台阶
40     printf("%d\n", ans); // 输出有多少种走法
41
42     return 0;
43 }
```

然后我们使用下面的命令就可以用文件 ex.ll 得到可以执行文件 ex

```
1 clang ex.ll -o ex
```

我们运行程序，选择几个输入进行测试，结果如下：



```
qiao@qiao-virtual-machine:~/桌面$ ./ex
1
2
2
qiao@qiao-virtual-machine:~/桌面$ ./ex
2
3
3
qiao@qiao-virtual-machine:~/桌面$ ./ex
9
5
8
```

图 11: 测试结果

- 第一次我们输入 1，与 5 进行取模运算后还是 1，b[1] 为 2，所以走的楼梯数就是 2，对应的走法就是 2 种，所以输出 2，2，结果正确。
- 第二次我们输入 2，与 5 进行取模运算后还是 2，b[1] 为 3，所以走的楼梯数就是 3，对应的走法就是 3 种，所以输出 3，3，结果正确。
- 第三次我们输入 9，与 5 进行取模运算后为 4，b[4] 为 5，所以走的楼梯数就是 5，对应的走法就是 8 种，所以输出 5，8，结果正确。

经过上面的运行我们可以知道程序的正确性。

语言特性分析：

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，global 表明它是一个全局变量，constant 表示这个全局变量是一个常量，它的值在运行时是不可修改的，此外还可以添加一些修饰符，例如 private

是可选的访问修饰符，用于限制该全局变量的可见性，`private` 可以表示该全局变量只能在当前模块内部访问。

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。在本次所编写的代码中，使用的最多的就是 `i32`，就是相当于 `int` 类型，此外还可以选择 `float` 等。此外在实践中发现如果%后面跟数字，必须从 0 开始，而且不能出现跳跃，像是用了%0 才能使用%1，否则不能成功编译。

2. 数组的定义及访问:

在此次的代码中我们定义的数组为 `b={1, 2, 3, 4, 5}`，其对应 IR 代码为 `@b = global [5 x i32] [i32 1, i32 2, i32 3, i32 4, i32 5]`，其中最开始的 `global` 代表这是全局数组，之后 `[5 x i32]` 表示数组元素个数为 5，类型为 `i32`，后面无个中括号中的内容指明数组对应位置上的元素类型及值，当然类型和前面的类型一样，都是 `i32`，值的话就是像代码中写的一样，为 1, 2, 3, 4, 5。

访问数组元素值时需要使用 `getelementptr` 关键字，像我们这里就是访问 `b[num]`，IR 代码具体有两行，第一行为 `%5 = getelementptr [5 x i32], [5 x i32]* @b, i32 0, i32 %4`，首先有 `[5 x i32]`，这就是数组 `b` 的类型，即 5 个 `i32` 类型变量的数组，`@b` 是一个全局变量，它是一个指向包含 5 个 `i32` 类型元素的数组的指针，`i32 0`：这表示数组索引的第一维度为 0，即数组的起始位置。`i32 %4`：%4 是另一个临时变量，它表示数组索引的第二维度，即数组元素的位置，这里%4 存的就是 `num` 的值，总之这行代码取得了数组指定位置的地址，存到了%5。第二行为 `%6 = load i32, i32* %5, align 4`。这里就是利用地址来完成具体值的读取，通过地址%5，将其中的值给了%6。

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]`”)`Block`”，这边的 `FuncFParams` 表示实参列表。具体而言我们定义了函数 `stairs`，具体代码为 `define i32 @stairs(i32 %n)`，后面跟函数的具体内容。调用的代码为 `%7 = call i32 @stairs(i32 %6)`，使用 `call` 将%6 作为参数，将函数返回结果给%7。

此外我们还使用了 `declare`。`declare` 指令被用于声明函数的存在，以便在当前模块中引用该函数。它告诉编译器或链接器该函数将在其他地方定义或实现。`declare` 指令通常用于外部库函数或在当前模块之外定义的函数。使用 `declare` 指令可以告诉编译器或链接器要寻找这个函数的定义，并在链接过程中解析它。此次我们就用了 `declare i32 @printf(i8*, ...)` 和 `declare i32 @scanf(i8*, ...)`。

4. 语句 (if, while, return, 赋值):

首先是 `if` 语句，此次 `if` 是用两条完成的，第一条为比较，`%cmp = icmp sle i32 %n, 1`，这里用的是 `sle`，就是用来判断小于等于的，成立赋值为真，反之为假，第二条是根据比较的结果进行跳转，`br i1 %cmp, label %if.then, label %if.else`，如果条件成立就跳到第一处，反之跳到第二处。

`while` 处的循环条件处也差不多，第一句为 `%1 = icmp sle i32 %0, %n`，用来计算条件，第二句为 `br i1 %1, label %while.body, label %while.end`，用来根据条件跳跃到指定的地点。

大致可以看出来 `if` 和 `while` 比较类似，都是计算条件，然后进行跳跃，即使用 `sle` 等比较类型进行判断和使用 `br` 进行跳跃。不过 `while` 可以跳回开头。

`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。这里在 `stairs` 函数里就用到了 `ret i32 1` 来返回常数，用 `ret i32 %ans` 来返回变量。

对于赋值语句，主要使用“`store`”和“`load`”关键字，如 `store i32 1, i32* %p, align 4` 将常数 1 保存到地址%`p` 上，而而 `%0 = load i32, i32* %i, align 4` 则将地址%i 上保存的值赋值给变量%0。

5. 表达式：算术运算和关系运算：

此次代码主要实现了两种算术运算：+，%，首先 +，-，*，/，% 对应的关键字分别为：add, sub, mul, div, srem, 因为 +，-，*，/，% 较为相似就只讨论加法，代码为 %4 = add i32 %2, %3, 将 %2 和 %3 的值加起来然后赋值给 %4。其余运算格式类似。

关系运算通过 icmp 来实现，各种关系运算符有对应的关键字，以下为常用的比较关键字。

eq	等于
ne	不等于
slt	有符号小于
sgt	有符号大于
sle	有符号小于等于
sge	有符号大于等于

五、 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。
- 具体分析编译阶段的作用以及结果
- 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、多种运算等 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。

六、 git 库

实验的材料在这个库当中：<https://gitee.com/is-314-double/lab1>