



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

编译原理实验报告

了解编译器及 LLVM IR 编程

2011763 黄天昊

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 9 月 30 日

摘要

本次实验主要探究了完整编译过程各个阶段的具体工作，对编译器阶段进行了详细的调试和了解，包括词法分析，语法分析和语义分析等多部分内容，探究的编译器为 gcc 编译器。此外，在了解编译器的基础上，还进一步实现了中间代码和 LLVM IR 编程。

关键字：编译过程，gcc 编译器，LLVM IR 编程

目录

一、 编译过程探究	1
(一) 完整的编译过程	1
(二) 预处理器	1
(三) 编译器	4
1. 词法分析	4
2. 语法分析	5
3. 语义分析	7
4. 中间代码生成	7
5. 代码优化	8
6. 代码生成	10
(四) 汇编器	16
(五) 链接器加载器	18
(六) gcc 编译器开启 O2 优化	24
二、 LLVM IR 编程	25
三、 总结	29

一、 编译过程探究

(一) 完整的编译过程

如图1所示，为完整编译过程的流程图

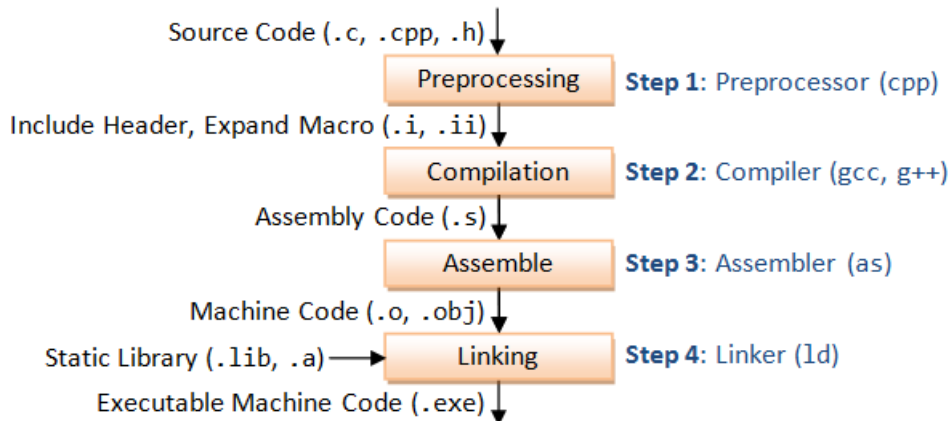


图 1: 编译过程流程图

编译过程分为几个阶段：

1. 词法分析
2. 语法分析
3. 语义分析
4. 中间代码生成
5. 代码优化
6. 目标代码生成

接下来的各部分中将会分析从源程序到目标机器代码的各个阶段，并且对编译过程的各个阶段进行调试和分析。

(二) 预处理器

C 预处理器不是编译器的组成部分，但是它是编译过程中一个单独的步骤。简言之，C 预处理器是一个文本替换工具，它们会指示编译器在实际编译之前完成所需的预处理。

简单来说，预处理就是将要包含 (include) 的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些代码输出到一个“.i”文件中等待进一步处理。

预编译过程主要处理那些源代码文件中以“#”开始的预编译指令。比如“#include”、“#define”等，主要处理规则如下：

- 将所有的“#define”删除，并且展开所有的宏定义
- 处理所有条件预编译指令，比如“#if”、“#ifdef”、“#elif”、“#else”、“#endif”

- 处理”#include” 预编译指令，将被包含的文件插入到该预编译指令的位置。这个过程是递归进行的，也就是说被包含的文件可能还包含其他文件删除所有的注释”//” 和”/* */”
- 添加行号和文件名标识，比如 #2 ”hello.c” 2，以便于编译时编译器产生调试用的行号信息及用于编译时产生编译错误或警告时能够显示行号
- 保留所有的 #pragma 编译器指令，因为编译器需要使用它们

经过预编译后的.i 文件不包含任何宏定义，因为所有的宏已经被展开，并且包含的文件也已经被插入到.i 文件中。所以当我们无法判断宏定义是否正确或头文件包含是否正确的时候，可以查看预编译后的文件来确定问题。

我使用的程序为斐波那契数列，代码如下：

```
1 #include<stdio.h>
2
3 int main() {
4     int a, b, i, t, n;
5     a = 0;
6     b = 1;
7     i = 1;
8     scanf("%d", &n);
9     printf("%d\n", a);
10    printf("%d\n", a);
11    while (i < n) {
12        t = b;
13        b = a + b;
14        printf("%d\n", b);
15        a = t;
16        i = i + 1;
17    }
18    return 0;
19 }
```

通过命令

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

来进行对 main.c 程序的预处理

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。如图2所示

```

C main.i
1 # 0 "main.c"
2 # 0 "<built-in>"
3 # 0 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 0 "<command-line>" 2
6 # 1 "main.c"
7 # 1 "/usr/include/stdio.h" 1 3 4
8 # 27 "/usr/include/stdio.h" 3 4
9 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
10 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
11 # 1 "/usr/include/features.h" 1 3 4
12 # 392 "/usr/include/features.h" 3 4
13 # 1 "/usr/include/features-time64.h" 1 3 4
14 # 20 "/usr/include/features-time64.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
16 # 21 "/usr/include/features-time64.h" 2 3 4
17 # 1 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 1 3 4
18 # 19 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 3 4
19 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
20 # 20 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 2 3 4
21 # 22 "/usr/include/features-time64.h" 2 3 4
22 # 393 "/usr/include/features.h" 2 3 4
23 # 486 "/usr/include/features.h" 3 4
24 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
25 # 559 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
26 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
27 # 560 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4

```

终端 问题 输出 调试控制台 窗口

```

● huangtianhao@huangtianhao-virtual-machine:~/Desktop$ gcc main.c -E -o main.i
○ huangtianhao@huangtianhao-virtual-machine:~/Desktop$

```

图 2: 生成预处理后的 main.i 文件部分代码

此外，我还使用命令

```
cpp main.c main_i.i
```

即直接使用 `cpp.exe` 来对 `main.c` 进行预处理。

结果发现其与之前得到的预处理结果相同，如图3所示

```

C main_ii
1 # 0 "main.c"
2 # 0 "<built-in>"
3 # 0 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 0 "<command-line>" 2
6 # 1 "main.c"
7 # 1 "/usr/include/stdio.h" 1 3 4
8 # 27 "/usr/include/stdio.h" 3 4
9 # 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
10 # 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
11 # 1 "/usr/include/features.h" 1 3 4
12 # 392 "/usr/include/features.h" 3 4
13 # 1 "/usr/include/features-time64.h" 1 3 4
14 # 20 "/usr/include/features-time64.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
16 # 21 "/usr/include/features-time64.h" 2 3 4
17 # 1 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 1 3 4
18 # 19 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 3 4
19 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
20 # 20 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 2 3 4
21 # 22 "/usr/include/features-time64.h" 2 3 4
22 # 393 "/usr/include/features.h" 2 3 4
23 # 486 "/usr/include/features.h" 3 4
24 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
25 # 559 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
26 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
27 # 560 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4

```

终端 问题 输出 调试控制台 窗口

```

● huangtianhao@huangtianhao-virtual-machine:~/Desktop$ gcc main.c -E -o main.i
● huangtianhao@huangtianhao-virtual-machine:~/Desktop$ cpp main.c main_i.i
○ huangtianhao@huangtianhao-virtual-machine:~/Desktop$

```

图 3: 使用 `cpp.exe` 生成预处理后的 main_ii 文件部分代码

(三) 编译器

1. 词法分析

在一个编译器中，词法分析的作用是将输入的字符串流进行分析，并按一定的规则将输入的字符串流进行分割，从而形成所使用的源程序语言所允许的记号 (token) 获得 token 序列，在这些记号序列将送到随后的语法分析过程中，与此同时将不符合规范的记号识别出来，并产生错误提示信息。

源代码程序被输入到扫描器 (Scanner)，扫描器对源代码进行简单的词法分析，运用类似于有限状态机 (Finite State Machine) 的算法可以很轻松的将源代码字符序列分割成一系列的记号 (Token)。

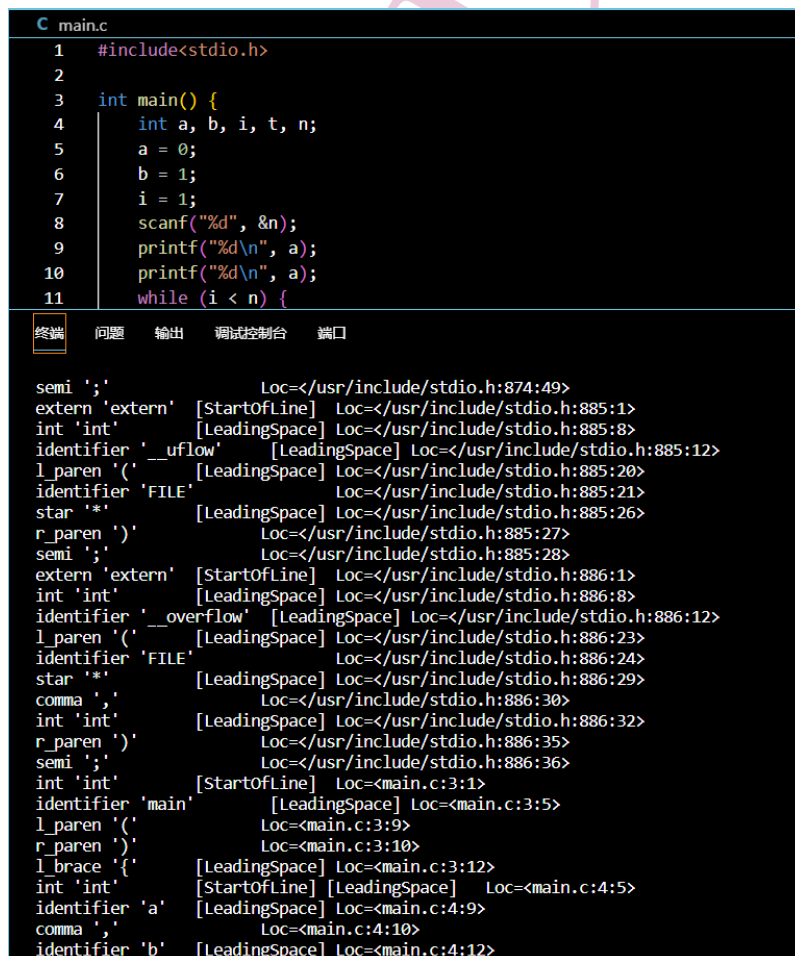
词法分析产生的记号一般可以分为如下几类：关键字、标识符、字面量（包含数字、字符串等）和特殊符号（如加号、等号）。在识别记号的同时，扫描器也完成了其他工作，比如将标识符存放到符号表，将数字、字符串常量存放到文字表等，以备后面的步骤使用。

词法分析可以使用 lex 工具。

对于 LLVM，通过

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

获得 token 序列，结果如图4所示



```
C main.c
1  #include<stdio.h>
2
3  int main() {
4      int a, b, i, t, n;
5      a = 0;
6      b = 1;
7      i = 1;
8      scanf("%d", &n);
9      printf("%d\n", a);
10     printf("%d\n", a);
11     while (i < n) {
    semi ';'                               Loc=</usr/include/stdio.h:874:49>
    extern 'extern' [StartOfLine] Loc=</usr/include/stdio.h:885:1>
    int 'int' [LeadingSpace] Loc=</usr/include/stdio.h:885:8>
    identifier '__uflow' [LeadingSpace] Loc=</usr/include/stdio.h:885:12>
    l_paren '(' [LeadingSpace] Loc=</usr/include/stdio.h:885:20>
    identifier 'FILE' Loc=</usr/include/stdio.h:885:21>
    star '*' [LeadingSpace] Loc=</usr/include/stdio.h:885:26>
    r_paren ')' Loc=</usr/include/stdio.h:885:27>
    semi ';' Loc=</usr/include/stdio.h:885:28>
    extern 'extern' [StartOfLine] Loc=</usr/include/stdio.h:886:1>
    int 'int' [LeadingSpace] Loc=</usr/include/stdio.h:886:8>
    identifier '__overflow' [LeadingSpace] Loc=</usr/include/stdio.h:886:12>
    l_paren '(' [LeadingSpace] Loc=</usr/include/stdio.h:886:23>
    identifier 'FILE' Loc=</usr/include/stdio.h:886:24>
    star '*' [LeadingSpace] Loc=</usr/include/stdio.h:886:29>
    comma ',' Loc=</usr/include/stdio.h:886:30>
    int 'int' [LeadingSpace] Loc=</usr/include/stdio.h:886:32>
    r_paren ')' Loc=</usr/include/stdio.h:886:35>
    semi ';' Loc=</usr/include/stdio.h:886:36>
    int 'int' [StartOfLine] Loc=<main.c:3:1>
    identifier 'main' [LeadingSpace] Loc=<main.c:3:5>
    l_paren '(' Loc=<main.c:3:9>
    r_paren ')' Loc=<main.c:3:10>
    l_brace '{' [LeadingSpace] Loc=<main.c:3:12>
    int 'int' [StartOfLine] [LeadingSpace] Loc=<main.c:4:5>
    identifier 'a' [LeadingSpace] Loc=<main.c:4:9>
    comma ',' Loc=<main.c:4:10>
    identifier 'b' [LeadingSpace] Loc=<main.c:4:12>
```

图 4: 词法分析结果

可以发现输出得到的信息包含 LeadingSpace 和 StartOfLine 等位置信息，其中 StartOfLine 表示的是一个新的 line 开始的位置，而 LeadingSpace 则是表示一个新的切分得到的 token 的开始的位置，其中的计数的 index 是从 1 开始计数，且在计算位置的时候将空格和缩进都包含了进去。

同时，我们也可以发现 clang 词法分析后，已经将 identifier 和 comma 不同的符号类标识了出来。

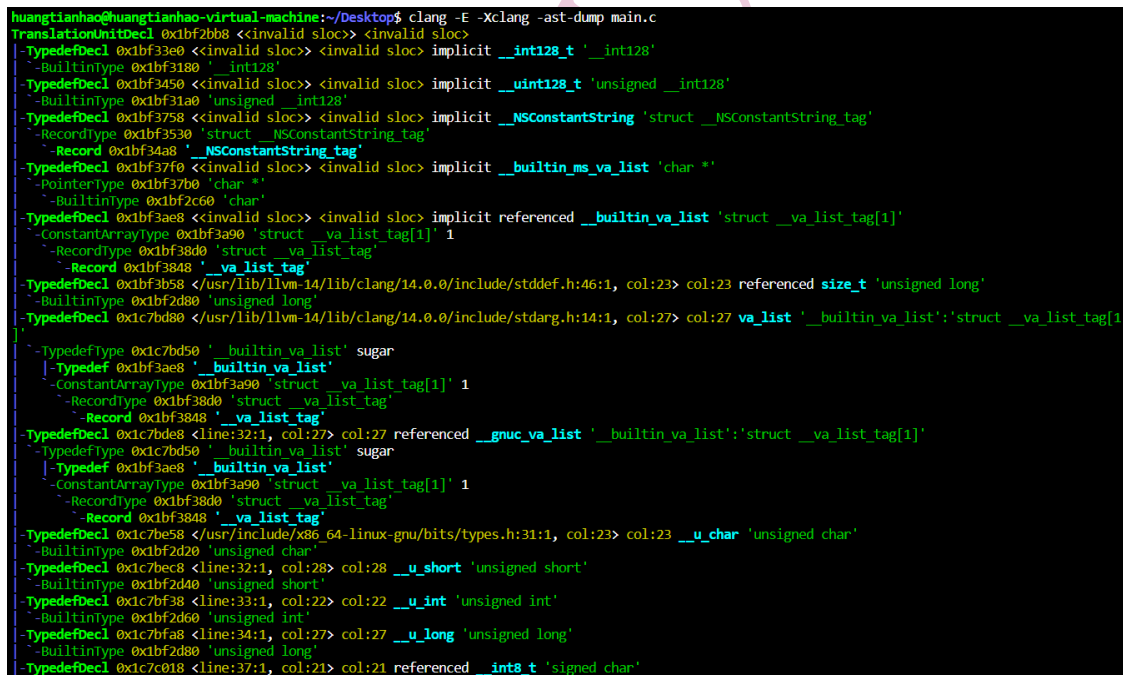
2. 语法分析

语法分析的过程是分析词法分析产生的记号序列，并按一定的语法规则识别并生成中间表示形式，以及符号表。符号表记录程序中所使用的标识符及其标识符的属性。同样，将不符合语法规则的记号识别出其位置并产生错误提示语句。目前大多数开源编译器使用的语法分析方法有两类，一种是自顶向下 (Top-Down Parsing) 的分析方法，另一种是自底向上 (Bottom-Up Parsing) 的分析方法。

LLVM 可以通过如下命令获得相应的 AST:

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

结果如图5和图6所示



```

huangtianhao@huangtianhao-virtual-machine:~/Desktop$ clang -E -Xclang -ast-dump main.c
TranslationUnitDecl 0x1bf2bb8 <invalid sloc> <invalid sloc>
- TypedefDecl 0x1bf33e0 <invalid sloc> <invalid sloc> implicit __int128_t '__int128'
- BuiltinType 0x1bf3180 '__int128'
- TypedefDecl 0x1bf3450 <invalid sloc> <invalid sloc> implicit __uint128_t 'unsigned __int128'
- BuiltinType 0x1bf31a0 'unsigned __int128'
- TypedefDecl 0x1bf3758 <invalid sloc> <invalid sloc> implicit __NSConstantString_tag 'struct __NSConstantString_tag'
- RecordType 0x1bf3530 'struct __NSConstantString_tag'
- Record 0x1bf34a8 '__NSConstantString_tag'
- TypedefDecl 0x1bf37f0 <invalid sloc> <invalid sloc> implicit __builtin_ms_va_list 'char **'
- PointerType 0x1bf37b0 'char **'
- BuiltinType 0x1bf2c60 'char'
- TypedefDecl 0x1bf3ae8 <invalid sloc> <invalid sloc> implicit referenced __builtin_va_list 'struct __va_list_tag[1]'
- ConstantArrayType 0x1bf3a90 'struct __va_list_tag[1]' 1
- RecordType 0x1bf38d0 'struct __va_list_tag'
- Record 0x1bf3848 '__va_list_tag'
- TypedefDecl 0x1bf3b58 </usr/lib/llvm-14/lib/clang/14.0.0/include/stddef.h:46:1, col:23> col:23 referenced size_t 'unsigned long'
- BuiltinType 0x1bf2d80 'unsigned long'
- TypedefDecl 0x1c7bd80 </usr/lib/llvm-14/lib/clang/14.0.0/include/stdarg.h:14:1, col:27> col:27 va_list '__builtin_va_list': 'struct __va_list_tag[1]'
- TypedefType 0x1c7bd50 '__builtin_va_list' sugar
- Typedef 0x1bf3ae8 '__builtin_va_list'
- ConstantArrayType 0x1bf3a90 'struct __va_list_tag[1]' 1
- RecordType 0x1bf38d0 'struct __va_list_tag'
- Record 0x1bf3848 '__va_list_tag'
- TypedefDecl 0x1c7bde8 <line:32:1, col:27> col:27 referenced __gnuc_va_list '__builtin_va_list': 'struct __va_list_tag[1]'
- TypedefType 0x1c7bd50 '__builtin_va_list' sugar
- Typedef 0x1bf3ae8 '__builtin_va_list'
- ConstantArrayType 0x1bf3a90 'struct __va_list_tag[1]' 1
- RecordType 0x1bf38d0 'struct __va_list_tag'
- Record 0x1bf3848 '__va_list_tag'
- TypedefDecl 0x1c7be58 </usr/include/x86_64-linux-gnu/bits/types.h:31:1, col:23> col:23 __u_char 'unsigned char'
- BuiltinType 0x1bf2d20 'unsigned char'
- TypedefDecl 0x1c7bec8 <line:32:1, col:28> col:28 __u_short 'unsigned short'
- BuiltinType 0x1bf2d40 'unsigned short'
- TypedefDecl 0x1c7bf38 <line:33:1, col:22> col:22 __u_int 'unsigned int'
- BuiltinType 0x1bf2d60 'unsigned int'
- TypedefDecl 0x1c7bfa8 <line:34:1, col:27> col:27 __u_long 'unsigned long'
- BuiltinType 0x1bf2d80 'unsigned long'
- TypedefDecl 0x1c7c018 <line:37:1, col:21> col:21 referenced __int8_t 'signed char'

```

图 5: 语法分析部分结果

```

DeclRefExpr @x1cb8a78 <col:20> 'int' lvalue Var @x1cb7e98 'a' 'int'
WhileStmt @x1cb8f88 <line:11:5, line:17:5>
  BinaryOperator @x1cb8bb0 <line:11:12, col:16> 'int' '^'
    ImplicitCastExpr @x1cb8b00 <col:12> 'int' <LValueToRValue>
      DeclRefExpr @x1cb8b40 <col:12> 'int' lvalue Var @x1cb7f98 'i' 'int'
    ImplicitCastExpr @x1cb8b98 <col:16> 'int' <LValueToRValue>
      DeclRefExpr @x1cb8b60 <col:16> 'int' lvalue Var @x1cb8098 'n' 'int'
CompoundStmt @x1cb8bf50 <col:19, line:17:5>
  BinaryOperator @x1cb8c28 <line:12:9, col:13> 'int' '='
    DeclRefExpr @x1cb8b40 <col:9> 'int' lvalue Var @x1cb8018 't' 'int'
    ImplicitCastExpr @x1cb8c10 <col:13> 'int' <LValueToRValue>
      DeclRefExpr @x1cb8bf0 <col:13> 'int' lvalue Var @x1cb7f18 'b' 'int'
  BinaryOperator @x1cb8cf8 <line:13:9, col:17> 'int' '='
    DeclRefExpr @x1cb8c48 <col:9> 'int' lvalue Var @x1cb7f18 'b' 'int'
    BinaryOperator @x1cb8cd8 <col:13, col:17> 'int' '+'
      ImplicitCastExpr @x1cb8ca8 <col:13> 'int' <LValueToRValue>
      DeclRefExpr @x1cb8c68 <col:13> 'int' lvalue Var @x1cb7e98 'a' 'int'
      ImplicitCastExpr @x1cb8cc0 <col:17> 'int' <LValueToRValue>
      DeclRefExpr @x1cb8c88 <col:17> 'int' lvalue Var @x1cb7f18 'b' 'int'
  CallExpr @x1cb8da8 <line:14:9, col:25> 'int'
    ImplicitCastExpr @x1cb8d90 <col:9> 'int (*) (const char *, ...)' <FunctionToPointerDecay>
      DeclRefExpr @x1cb8d18 <col:9> 'int (const char *, ...)' Function @x1c9ce18 'printf' 'int (const char *, ...)'
    ImplicitCastExpr @x1cb8d10 <col:16> 'const char *' <Nop>
    ImplicitCastExpr @x1cb8dd8 <col:16> 'char *' <ArrayToPointerDecay>
      StringLiteral @x1cb8d38 <col:16> 'char[4]' lvalue "Xd\n"
    ImplicitCastExpr @x1cb8e08 <col:24> 'int' <LValueToRValue>
      DeclRefExpr @x1cb8d58 <col:24> 'int' lvalue Var @x1cb7f18 'b' 'int'
  BinaryOperator @x1cb8e78 <line:15:9, col:13> 'int' '='
    DeclRefExpr @x1cb8e20 <col:9> 'int' lvalue Var @x1cb7e98 'a' 'int'
    ImplicitCastExpr @x1cb8e60 <col:13> 'int' <LValueToRValue>
      DeclRefExpr @x1cb8e40 <col:13> 'int' lvalue Var @x1cb8018 't' 'int'
  BinaryOperator @x1cb8f30 <line:16:9, col:17> 'int' '='
    DeclRefExpr @x1cb8e98 <col:9> 'int' lvalue Var @x1cb7f98 'i' 'int'
    BinaryOperator @x1cb8f10 <col:13, col:17> 'int' '+'
      ImplicitCastExpr @x1cb8ef8 <col:13> 'int' <LValueToRValue>
      DeclRefExpr @x1cb8eb8 <col:13> 'int' lvalue Var @x1cb7f98 'i' 'int'
      IntegerLiteral @x1cb8ed8 <col:17> 'int' 1
ReturnsStmt @x1cb8fc8 <line:18:5, col:12>
  IntegerLiteral @x1cb8fa8 <col:12> 'int' 0

```

图 6: 语法分析部分结果

我们可以看到，在图5中，主要是一些预编译得到的结果生成的语法分析，大多来自于包含的 `stdio.h` 的内容。而图6中，是 `main` 函数的语法分析结果。我们可以发现 clang 语法分析之后，不仅标记了在文本之中代码的行列的值，函数包含的行号等信息，还包含了具体的程序加载地址的值，以及各种符号的含义和参数。

此外，输入命令

```
gcc -fdump-tree-original-raw main.c
```

可以得到文本格式的 AST 输出, 如图7

```

aout
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded Text
00000000 7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00 00
00000010 03 00 3E 00 01 00 00 00 A0 10 00 00 00 00 00 00 00
00000020 40 00 00 00 00 00 00 00 F8 36 00 00 00 00 00 00 00
00000030 00 00 00 00 40 00 38 00 00 40 00 1F 00 1E 00 00
00000040 06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00
00000050 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00000060 D8 02 00 00 00 00 00 00 D8 02 00 00 00 00 00 00
00000070 08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00
00000080 18 03 00 00 00 00 00 00 18 03 00 00 00 00 00 00
00000090 18 03 00 00 00 00 00 00 1C 00 00 00 00 00 00 00
000000A0 1C 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00
000000B0 01 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00
000000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000D0 E0 06 00 00 00 00 00 E0 06 00 00 00 00 00 00
000000E0 00 10 00 00 00 00 00 00 01 00 00 00 05 00 00
000000F0 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00
00000100 00 10 00 00 00 00 00 00 69 02 00 00 00 00 00
00000110 69 02 00 00 00 00 00 00 10 00 00 00 00 00 00
00000120 01 00 00 00 04 00 00 00 20 00 00 00 00 00 00
00000130 00 20 00 00 00 00 00 00 20 00 00 00 00 00 00
00000140 EC 00 00 00 00 00 00 00 EC 00 00 00 00 00 00
00000150 00 10 00 00 00 00 00 00 01 00 00 00 06 00 00
00000160 A8 2D 00 00 00 00 00 A8 3D 00 00 00 00 00 00
00000170 A8 3D 00 00 00 00 00 00 68 02 00 00 00 00 00
00000180 79 02 00 00 00 00 00 00 1B 00 00 00 00 00 00
00000190 02 00 00 00 06 00 00 00 88 2D 00 00 00 00 00
000001A0 B8 3D 00 00 00 00 00 88 3D 00 00 00 00 00
000001B0 F0 01 00 00 00 00 00 F0 01 00 00 00 00 00
000001C0 08 00 00 00 00 00 00 00 04 00 00 00 04 00 00

```

图 7: 二进制文本格式的 AST

但是由于其是二进制文件，我们并不能直观地看到整棵 AST。

3. 语义分析

语义分析是编译过程的一个逻辑阶段，语义分析的任务是对结构上正确的源程序进行上下文有关性质的审查，进行类型审查。语义分析是审查源程序有无语义错误，为代码生成阶段收集类型信息。比如语义分析的一个工作是进行类型审查，审查每个算符是否具有语言规范允许的运算对象，当不符合语言规范时，编译程序应报告错误。如有的编译程序要对实数用作数组下标的情况报告错误。又比如某些程序规定运算对象可被强制，那么当二目运算施于一整型和一实型对象时，编译程序应将整型转换为实型而不能认为是源程序的错误。

语义分析的地位：编译程序最实质性的工作；第一次对源程序的语义作出解释，引起源程序质的变化。

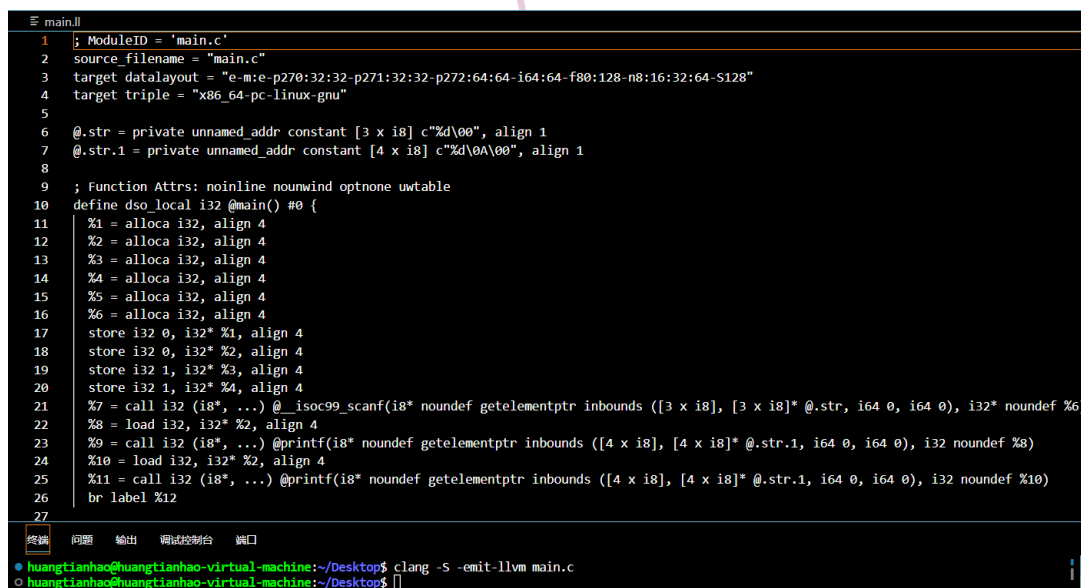
4. 中间代码生成

目前大多数编译器前端会将源程序转换成中间代码的表示形式，本实验中是将要转化为 LLVM IR 的形式。我们可以通过 `-fdump-tree-all-graph` 和 `-fdump-rtl-all-graph` 两个 gcc flag 获得中间代码生成的多阶段的输出。也可以通过 LLVM，运行指令：

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

来生成 LLVM IR。

如图8



```

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 (@__isoc99_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 (@__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* %2, align 4
25     %11 = call i32 (@__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

```

图 8: 中间代码生成

可以看到，首先是几句赋值语句，中间代码中，变量不用字母表示，而是使用 `%i` 的形式。i32 说明是 int 类型，align 4 表示分配 4 字节的内存空间。在这里，`%1` 应该是 `argv` 等 main 函数的变量，`%2 - %6` 是我们分别声明的五个变量。

之后的几句 store 代码即为几个变量赋值。21 - 25 行，调用了 scanf 和 printf 函数。

之后 br 即跳转到分支指令，开始 while 循环。

通过命令

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

可以得到控制流图，如图9

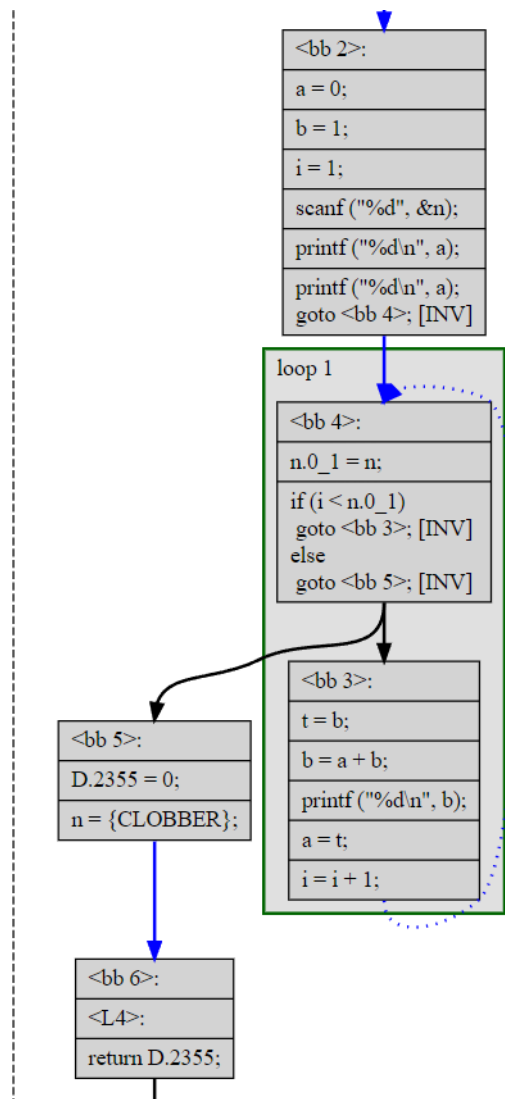


图 9: 控制流图

5. 代码优化

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。

代码优化按照优化的代码块尺度分为：局部优化、循环优化和全局优化。常见的代码优化技术有：删除多余运算、合并已知量和复写传播，删除无用赋值等。

通过指令

```
llc -print-before-all -print-after-all main.ll > a.log 2>&1
```

我们可以看到每个阶段后 LLVM IR 中间代码的变化，并且整个 log 文件的后半部分会给出具体到不同 machine code 的过程 log 展示，所以文件的体量较大。如图10

```

1 *** IR Dump Before Pre-Isel Intrinsic Lowering (pre-isel-intrinsic-lowering) ***
2 ; ModuleID = 'main.ll'
3 source_filename = "main.c"
4 target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
5 target triple = "x86_64-pc-linux-gnu"
6
7 @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
8 @.str.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
9
10 ; Function Attrs: noinline nounwind optnone uwtable
11 define dso_local i32 @main() #0 {
12     %1 = alloca i32, align 4
13     %2 = alloca i32, align 4
14     %3 = alloca i32, align 4
15     %4 = alloca i32, align 4
16     %5 = alloca i32, align 4
17     %6 = alloca i32, align 4
18     store i32 0, i32* %1, align 4
19     store i32 0, i32* %2, align 4
20     store i32 1, i32* %3, align 4
21     store i32 1, i32* %4, align 4
22     %7 = call i32 (@__isoc99_scanf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %
23     %8 = load i32, i32* %2, align 4
24     %9 = call i32 (@__printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %8)
25     %10 = load i32, i32* %2, align 4
26     %11 = call i32 (@__printf(i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 noundef %10)
27     br label %12
28
29 12:                                     ; preds = %16, %0
30     %13 = load i32, i32* %4, align 4
31     %14 = load i32, i32* %6, align 4
32     %15 = icmp slt i32 %13, %14
33     br i1 %15, label %16, label %26

```

图 10: 中间代码优化结果

对于重定向的问题，有以下几点知识点：

- 程序运行后会打开三个文件描述符，分别是标准输入，标准输出和标准错误输出。
- 在调用脚本时，可使用 `2>&1` 来将标准错误输出重定向。
- 只需要查看脚本的错误时，可将标准输出重定向到文件，而标准错误会打印在控制台，便于查看。
- `»log.txt` 会将重定向内容追加到 `log.txt` 文件末尾。
- 通过查看 `/proc/进程 id/fd` 下的内容，可了解进程打开的文件描述符信息。

详见博客[如何理解 Linux shell 中的“2>&1”](#)

在 LLVM 官网对所有 pass 的分类中，共分为三种：Analysis Passes、Transform Passes 和 Utility Passes。Analysis Passes 用于分析或计算某些信息，以便给其他 pass 使用，如计算支配边界、控制流图的数据流分析等；Transform Passes 都会通过某种方式对中间代码形式的程序做某种变化，如死代码删除，常量传播等。其使用的命令是

```
opt <module name> <test.bc> /dev/null
```

这里我们需要的是 bc 文件格式，可以使用如下命令实现 LLVM 的 ll 文件与 bc 文件互转：

```
llvm-dis a.bc -o a.ll # bc 转换为 ll
llvm-as a.ll -o a.bc # ll 转换为 bc
```

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输入如下命令：

```
gcc main.i -S -o main.S # 生成 x86 格式目标代码
arm-linux-gnueabi-gcc main.i -S -o main.S # 生成 arm 格式目标代码
llc main.ll -o main.S # LLVM 生成目标代码
```

使用 gcc 处理后，生成了 main.s 和 main.S 两个文件，经查阅得知，这两种文件都是汇编代码，其区别在于：

- .s 格式的汇编文件中，只能包含纯粹的汇编代码，汇编器只对其进行汇编操作，没有预处理操作
- .S 格式的汇编文件中，还可以使用预处理命令，汇编器会先进行预处理，然后再进行汇编

gcc 编译下的汇编代码为：

```

1      .text
2      .file    "main.c"
3      .globl   main                # — Begin function main
4      .p2align      4, 0x90
5      .type      main,@function
6 main:                                # @main
7      .cfi_startproc
8 # %bb.0:
9      pushq     %rbp
10     .cfi_def_cfa_offset 16
11     .cfi_offset %rbp, -16
12     movq      %rsp, %rbp
13     .cfi_def_cfa_register %rbp
14     subq      $32, %rsp
15     movl      $0, -24(%rbp)
16     movl      $0, -8(%rbp)
17     movl      $1, -4(%rbp)
18     movl      $1, -12(%rbp)
19     movabsq   $.L.str, %rdi
20     leaq      -16(%rbp), %rsi
21     movb      $0, %al
22     callq     __isoc99_scanf@PLT
23     movl      -8(%rbp), %esi
24     movabsq   $.L.str.1, %rdi
25     movb      $0, %al
26     callq     printf@PLT
27     movl      -8(%rbp), %esi
28     movabsq   $.L.str.1, %rdi
29     movb      $0, %al
30     callq     printf@PLT
31 .LBB0_1:                            # =>This Inner Loop Header: Depth=1
32     movl      -12(%rbp), %eax
33     cmpl      -16(%rbp), %eax

```

```

34         jge      .LBB0_3
35 # %bb.2:                                     #   in Loop: Header=BB0_1 Depth=1
36         movl     -4(%rbp), %eax
37         movl     %eax, -20(%rbp)
38         movl     -8(%rbp), %eax
39         addl     -4(%rbp), %eax
40         movl     %eax, -4(%rbp)
41         movl     -4(%rbp), %esi
42         movabsq  $.L.str.1, %rdi
43         movb     $0, %al
44         callq    printf@PLT
45         movl     -20(%rbp), %eax
46         movl     %eax, -8(%rbp)
47         movl     -12(%rbp), %eax
48         addl     $1, %eax
49         movl     %eax, -12(%rbp)
50         jmp      .LBB0_1
51 .LBB0_3:
52         xorl     %eax, %eax
53         addq     $32, %rsp
54         popq     %rbp
55         .cfi_def_cfa %rsp, 8
56         retq
57 .Lfunc_end0:
58         .size    main, .Lfunc_end0-main
59         .cfi_endproc
60                                     # — End function
61         .type    .L.str, @object                # @.str
62         .section .rodata.str1.1, "aMS", @progbits, 1
63 .L.str:
64         .asciz   "%d"
65         .size    .L.str, 3
66
67         .type    .L.str.1, @object                # @.str.1
68 .L.str.1:
69         .asciz   "%d\n"
70         .size    .L.str.1, 4
71
72         .ident   "Ubuntu clang version 14.0.0-1ubuntu1"
73         .section ".note.GNU-stack","",@progbits

```

arm 格式目标代码为:

```

1         .arch   armv7-a
2         .fpu    vfpv3-d16
3         .eabi_attribute 28, 1
4         .eabi_attribute 20, 1
5         .eabi_attribute 21, 1
6         .eabi_attribute 23, 3

```

```

7      .eabi_attribute 24, 1
8      .eabi_attribute 25, 1
9      .eabi_attribute 26, 2
10     .eabi_attribute 30, 6
11     .eabi_attribute 34, 1
12     .eabi_attribute 18, 4
13     .file    "main.c"
14     .text
15     .section      .rodata
16     .align  2
17 .LC0:
18     .ascii  "%d\000"
19     .align  2
20 .LC1:
21     .ascii  "%d\012\000"
22     .text
23     .align  1
24     .global main
25     .syntax unified
26     .thumb
27     .thumb_func
28     .type   main, %function
29 main:
30     @ args = 0, pretend = 0, frame = 24
31     @ frame_needed = 1, uses_anonymous_args = 0
32     push    {r7, lr}
33     sub     sp, sp, #24
34     add     r7, sp, #0
35     ldr     r2, .L6
36 .LPIC4:
37     add     r2, pc
38     ldr     r3, .L6+4
39     ldr     r3, [r2, r3]
40     ldr     r3, [r3]
41     str     r3, [r7, #20]
42     mov     r3, #0
43     movs    r3, #0
44     str     r3, [r7, #4]
45     movs    r3, #1
46     str     r3, [r7, #8]
47     movs    r3, #1
48     str     r3, [r7, #12]
49     mov     r3, r7
50     mov     r1, r3
51     ldr     r3, .L6+8
52 .LPIC0:
53     add     r3, pc
54     mov     r0, r3

```

```

55         bl      __isoc99_scanf(PLT)
56         ldr     r1, [r7, #4]
57         ldr     r3, .L6+12
58 .LPIC1:
59         add     r3, pc
60         mov     r0, r3
61         bl      printf(PLT)
62         ldr     r1, [r7, #4]
63         ldr     r3, .L6+16
64 .LPIC2:
65         add     r3, pc
66         mov     r0, r3
67         bl      printf(PLT)
68         b       .L2
69 .L3:
70         ldr     r3, [r7, #8]
71         str     r3, [r7, #16]
72         ldr     r2, [r7, #8]
73         ldr     r3, [r7, #4]
74         add     r3, r3, r2
75         str     r3, [r7, #8]
76         ldr     r1, [r7, #8]
77         ldr     r3, .L6+20
78 .LPIC3:
79         add     r3, pc
80         mov     r0, r3
81         bl      printf(PLT)
82         ldr     r3, [r7, #16]
83         str     r3, [r7, #4]
84         ldr     r3, [r7, #12]
85         adds    r3, r3, #1
86         str     r3, [r7, #12]
87 .L2:
88         ldr     r3, [r7]
89         ldr     r2, [r7, #12]
90         cmp     r2, r3
91         blt     .L3
92         movs    r3, #0
93         ldr     r1, .L6+24
94 .LPIC5:
95         add     r1, pc
96         ldr     r2, .L6+4
97         ldr     r2, [r1, r2]
98         ldr     r1, [r2]
99         ldr     r2, [r7, #20]
100        eors    r1, r2, r1
101        mov     r2, #0
102        beq     .L5

```

```

103     bl      ___stack_chk_fail(PLT)
104 .L5:
105     mov     r0, r3
106     adds    r7, r7, #24
107     mov     sp, r7
108     @ sp needed
109     pop     {r7, pc}
110 .L7:
111     .align  2
112 .L6:
113     .word   _GLOBAL_OFFSET_TABLE_-(.LPIC4+4)
114     .word   ___stack_chk_guard(GOT)
115     .word   .LC0-(.LPIC0+4)
116     .word   .LC1-(.LPIC1+4)
117     .word   .LC1-(.LPIC2+4)
118     .word   .LC1-(.LPIC3+4)
119     .word   _GLOBAL_OFFSET_TABLE_-(.LPIC5+4)
120     .size   main, .-main
121     .ident  "GCC: (Ubuntu 11.2.0-17ubuntu1) 11.2.0"
122     .section .note.GNU-stack,"",%progbits

```

LLVM clang 编译下的 x86 目标代码:

```

1     .text
2     .file   "main.c"
3     .globl  main                                # — Begin function main
4     .p2align 4, 0x90
5     .type   main, @function
6 main:                                           # @main
7     .cfi_startproc
8 # %bb.0:
9     pushq   %rbp
10    .cfi_def_cfa_offset 16
11    .cfi_offset %rbp, -16
12    movq     %rsp, %rbp
13    .cfi_def_cfa_register %rbp
14    subq     $32, %rsp
15    movl     $0, -24(%rbp)
16    movl     $0, -8(%rbp)
17    movl     $1, -4(%rbp)
18    movl     $1, -12(%rbp)
19    movabsq  $.L.str, %rdi
20    leaq     -16(%rbp), %rsi
21    movb     $0, %al
22    callq    ___isoc99_scanf@PLT
23    movl     -8(%rbp), %esi
24    movabsq  $.L.str.1, %rdi
25    movb     $0, %al
26    callq    printf@PLT

```



```

27     movl    -8(%rbp), %esi
28     movabsq $.L.str.1, %rdi
29     movb    $0, %al
30     callq   printf@PLT
31 .LBB0_1:                                     # =>This Inner Loop Header: Depth=1
32     movl    -12(%rbp), %eax
33     cmpl    -16(%rbp), %eax
34     jge     .LBB0_3
35 # %bb.2:                                     #   in Loop: Header=BB0_1 Depth=1
36     movl    -4(%rbp), %eax
37     movl    %eax, -20(%rbp)
38     movl    -8(%rbp), %eax
39     addl    -4(%rbp), %eax
40     movl    %eax, -4(%rbp)
41     movl    -4(%rbp), %esi
42     movabsq $.L.str.1, %rdi
43     movb    $0, %al
44     callq   printf@PLT
45     movl    -20(%rbp), %eax
46     movl    %eax, -8(%rbp)
47     movl    -12(%rbp), %eax
48     addl    $1, %eax
49     movl    %eax, -12(%rbp)
50     jmp     .LBB0_1
51 .LBB0_3:
52     xorl    %eax, %eax
53     addq    $32, %rsp
54     popq    %rbp
55     .cfi_def_cfa %rsp, 8
56     retq
57 .Lfunc_end0:
58     .size   main, .Lfunc_end0-main
59     .cfi_endproc
60                                     # — End function
61     .type   .L.str, @object           # @.str
62     .section .rodata.str1.1, "aMS", @progbits, 1
63 .L.str:
64     .asciz  "%d"
65     .size   .L.str, 3
66
67     .type   .L.str.1, @object         # @.str.1
68 .L.str.1:
69     .asciz  "%d\n"
70     .size   .L.str.1, 4
71
72     .ident  "Ubuntu clang version 14.0.0-1ubuntu1"
73     .section ".note.GNU-stack","",@progbits

```

(四) 汇编器

汇编过程实际上是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。这一步一般被视为编译过程的“后端”。

1. 指令选择 (Instruction selection)
2. 寄存器分配 (Register allocation)
3. 指令调度 (Instruction scheduling)
4. 指令编码 (Instruction encoding)

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：

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

结果为图11

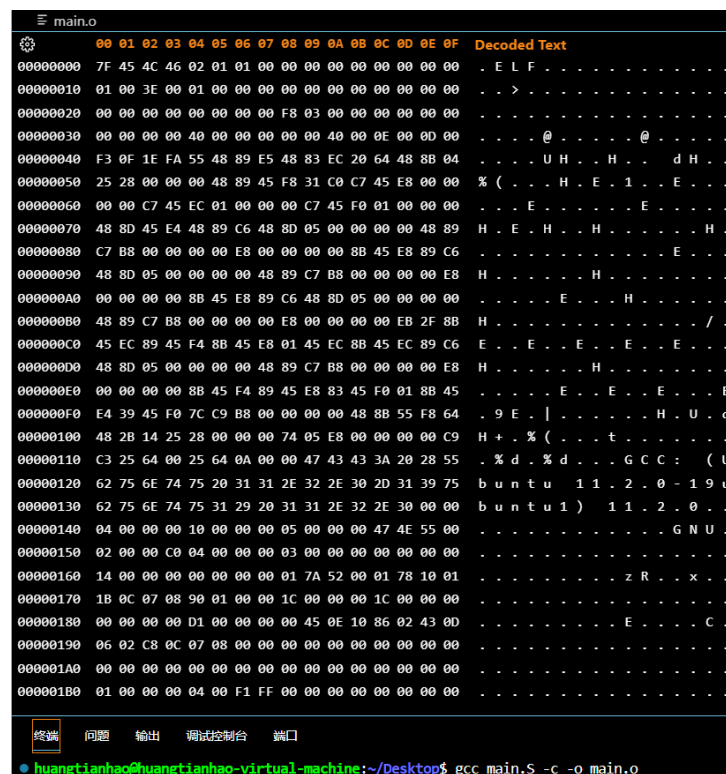


图 11: gcc 汇编结果

由于该文件为二进制文件，我们可以使用指令

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

来对其进行反汇编

得到的反汇编代码为：

```
1 main.o:      file format 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 e8              mov    -0x18(%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	

arm 和 LLVM 的代码在此由于篇幅限制不再展开。

在代码生成阶段得到的.S 格式的文件是程序在 Linux 系统下存储的格式，而在汇编过程进行反编译得到的.S 格式的文件中包含的则是汇编器过程得到的具体的汇编指令。

通过与代码生成部分的对比，我们可以看到指令选择的部分，例如使用了更为简洁的 mov 指令代替了部分 movl 操作长字的指令。而在左侧我们也能够清晰地看到指令的编址以及指令的具体机器代码。

可以看到具体的寄存器分配和寄存器的地址计算，也可以看到将函数的名称换成了具体的 < 地址 + main 的地址 > 与此同时我们可以发现一部分代码调度上的不同编译器的不同，在示例的斐波那契数列代码之中，while 循环判断时，在 gcc 编译之中将 i 的判断条件放在了程序后侧，使用了 jmp 指令进行了跳转，而 LLVM 则是在程序前部直接进行了 cmp 判断。除此之外，LLVM 在 add 指令的处理上和 gcc 也不同，gcc 编译器会直接在寻址的位置进行 add 加法操作，而 LLVM 操作时则需要借助 eax 寄存器的帮助。

gcc 中的 add 操作

```
aa : 83 45 f0 01 addl $0x1 , -0x10(%rbp)
```

LLVM clang 中的 add 操作

```
92: 8b 45 f8 mov -0x8(%rbp) , %eax
```

```
95: 83 c0 01 add $0x1, %eax
```

```
98: 89 45 f8 mov %eax , -0x8(%rbp)
```

(五) 链接器加载器

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。

使用命令:

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

```
clang main.c -o main
```

就可以分别获得 gcc 和 clang 下的可执行文件

在 terminal 中可以直接运行该文件，结果如图12

```

huangtianhao@huangtianhao-virtual-machine: ~/Desktop
huangtianhao@huangtianhao-virtual-machine:~/Desktop$ ./main
5
0
0
1
2
3
5
huangtianhao@huangtianhao-virtual-machine:~/Desktop$ ./main
10
0
0
1
2
3
5
8
13
21
34
55
huangtianhao@huangtianhao-virtual-machine:~/Desktop$

```

图 12: 运行生成的可执行文件

通过执行反编译命令:

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

可以得到将该可执行文件反汇编的结果。

gcc 可执行文件反汇编结果:

```

1 main:      file format 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 $0xfffffffffffffff0,%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+
    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+
    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 e8                mov     -0x18(%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

```

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

```

可以发现在原有的基础上多了许多其他的函数模块, 这些模块都是从其他的库文件中进行加载的。

(六) gcc 编译器开启 O2 优化

O2 优化的具体内容: gcc 将执行几乎所有的不包含时间和空间折中的优化。当设置 O2 选项时, 编译器并不进行循环打开 () loop unrolling 以及函数内联。与 O1 比较而言, O2 优化增加了编译时间的基础上, 提高了生成代码的执行效率。

通过如下的命令过程得到经过 O2 优化过后的汇编代码:

```

gcc main.i -O2 -S -o gcc-o2.S
gcc gcc-o2.S -O2 -c -o gcc-o2.o
objdump -d gcc-o2.o > gcc-o2-anti-obj.S

```

结果如下:

```

1 gcc-o2.o:      file format elf64-x86-64
2
3
4 Disassembly of section .text.startup:
5
6 0000000000000000 <main>:
7   0:  f3 0f 1e fa      endbr64
8   4:  41 55            push    %r13
9   6:  48 8d 3d 00 00 00 00 lea     0x0(%rip),%rdi      # d <main+0xd>
10  d:  4c 8d 2d 00 00 00 00 lea     0x0(%rip),%r13      # 14 <main+0x14>
11 14:  41 54            push    %r12
12 16:  55              push    %rbp
13 17:  53              push    %rbx
14 18:  48 83 ec 18      sub     $0x18,%rsp
15 1c:  64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
16 23:  00 00
17 25:  48 89 44 24 08      mov     %rax,0x8(%rsp)
18 2a:  31 c0            xor     %eax,%eax
19 2c:  48 8d 74 24 04      lea     0x4(%rsp),%rsi
20 31:  e8 00 00 00 00      call    36 <main+0x36>
21 36:  31 f6            xor     %esi,%esi
22 38:  4c 89 ef          mov     %r13,%rdi
23 3b:  31 c0            xor     %eax,%eax
24 3d:  e8 00 00 00 00      call    42 <main+0x42>
25 42:  31 f6            xor     %esi,%esi
26 44:  31 c0            xor     %eax,%eax
27 46:  4c 89 ef          mov     %r13,%rdi
28 49:  e8 00 00 00 00      call    4e <main+0x4e>

```

```

29  4e:  83 7c 24 04 01      cmpl    $0x1,0x4(%rsp)
30  53:  7e 30              jle     85 <main+0x85>
31  55:  bd 01 00 00 00      mov     $0x1,%ebp
32  5a:  bb 01 00 00 00      mov     $0x1,%ebx
33  5f:  31 c0              xor     %eax,%eax
34  61:  0f 1f 80 00 00 00 00 nopl    0x0(%rax)
35  68:  41 89 dc            mov     %ebx,%r12d
36  6b:  01 c3              add     %eax,%ebx
37  6d:  4c 89 ef            mov     %r13,%rdi
38  70:  31 c0              xor     %eax,%eax
39  72:  89 de            mov     %ebx,%esi
40  74:  83 c5 01            add     $0x1,%ebp
41  77:  e8 00 00 00 00      call   7c <main+0x7c>
42  7c:  44 89 e0            mov     %r12d,%eax
43  7f:  39 6c 24 04          cmp     %ebp,0x4(%rsp)
44  83:  7f e3              jg      68 <main+0x68>
45  85:  48 8b 44 24 08      mov     0x8(%rsp),%rax
46  8a:  64 48 2b 04 25 28 00 sub     %fs:0x28,%rax
47  91:  00 00
48  93:  75 0d              jne     a2 <main+0xa2>
49  95:  48 83 c4 18          add     $0x18,%rsp
50  99:  31 c0              xor     %eax,%eax
51  9b:  5b                pop     %rbx
52  9c:  5d                pop     %rbp
53  9d:  41 5c              pop     %r12
54  9f:  41 5d              pop     %r13
55  a1:  c3                ret
56  a2:  e8 00 00 00 00      call   a7 < .LC1+0xa4>

```

我们可以清晰地看到，在函数调用的过程之中，解决了在代码生成部分的问题，不再单纯的使用 rip 寄存器寻址再存到 rax 寄存器中，而是使用了如 r13 等寄存器，直接将其赋值给 rdi 或者 rsi 进行传参，O2 优化后代码会更加精简，它会在编译期间占用更多的内存和编译时间。

二、 LLVM IR 编程

使用 LLVM IR 中间语言编写简单的程序，体现 SysY 语言的各个语言特性。SysY 语言特性如下：

1. 数据类型：int 整型
2. 变量声明、常量声明，常量、变量的初始化
3. 语句：赋值 (=)、表达式语句、语句块、if、while、return
4. 表达式：算术运算 (+、-、*、/、%，其中 +、- 都可以是单目运算符)、关系运算 (==, >, <, >=, <=, !=) 和逻辑运算 (&& (与)、|| (或)、! (非))
5. 注释
6. 输入输出

7. 函数、语句块（函数声明、函数调用、变量、常量作用域）
8. 数组：数组（一维、二维、）的声明和数组元素访问
9. 浮点数：浮点数常量识别、变量声明、存储、运算

我和费泽锟将会分别编写 LLVM IR 小程序，来体现 sysY 语言特性。
我编写的代码如下：

```

1 #include<stdio.h>
2 float add(float a, float b){
3     return a+b;
4 }
5
6 int main(){
7     float a[5][5];
8     for(int i=0;i<5;i++){
9         for(int j=0;j<5;j++){
10             a[i][j]=i+j+i*j;
11         }
12     }
13     int n;
14     scanf("%d",&n);
15     a[3][3]=add(a[3][3],n);
16     printf("%f",a[3][3]);
17     return 0;
18 }

```

该程序主要是进行了函数的定义与调用，以及二维数组的声明与赋值，算术运算。
其中包含的 sysY 语言特性有：

1. 数据类型：int 整型
2. 变量声明、常量声明，常量、变量的初始化
3. 语句：赋值（=）、表达式语句、语句块、for、return
4. 表达式：算术运算 + *，逻辑运算 <
5. 输入输出
6. 函数、语句块（函数声明、函数调用、变量、常量作用域）
7. 二维数组的声明和数组元素访问
8. 浮点数：浮点数常量识别、变量声明、存储、运算

相应的 LLVM 中间代码为：

```

1 @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
2 @.str.1 = private unnamed_addr constant [3 x i8] c"%f\00", align 1
3
4 ; Function Attrs: noinline nounwind optnone uwtable
5 define dso_local float @add(float noundef %0, float noundef %1) #0 {

```

```

6   %3 = alloca float, align 4
7   %4 = alloca float, align 4
8   store float %0, float* %3, align 4
9   store float %1, float* %4, align 4
10  %5 = load float, float* %3, align 4
11  %6 = load float, float* %4, align 4
12  %7 = fadd float %5, %6
13  ret float %7                                ;add函数实现
14 }
15
16 ; Function Attrs: noinline nounwind optnone uwtable
17 define dso_local i32 @main() #0 {
18   %1 = alloca i32, align 4
19   %2 = alloca [5 x [5 x float]], align 16
20   %3 = alloca i32, align 4
21   %4 = alloca i32, align 4
22   %5 = alloca i32, align 4
23   store i32 0, i32* %1, align 4
24   store i32 0, i32* %3, align 4
25   br label %6                                ;跳转到for循环
26
27 6:
28   %7 = load i32, i32* %3, align 4
29   %8 = icmp slt i32 %7, 5                    ;i与5比较
30   br i1 %8, label %9, label %35              ;分支指令
31
32 9:
33   store i32 0, i32* %4, align 4
34   br label %10
35
36 10:
37   %11 = load i32, i32* %4, align 4
38   %12 = icmp slt i32 %11, 5                  ;j与5比较
39   br i1 %12, label %13, label %31            ;分支指令
40
41 13:
42   %14 = load i32, i32* %3, align 4
43   %15 = load i32, i32* %4, align 4
44   %16 = add nsw i32 %14, %15
45   %17 = load i32, i32* %3, align 4
46   %18 = load i32, i32* %4, align 4
47   %19 = mul nsw i32 %17, %18
48   %20 = add nsw i32 %16, %19
49   %21 = sitofp i32 %20 to float
50   %22 = load i32, i32* %3, align 4
51   %23 = sext i32 %22 to i64
52   %24 = getelementptr inbounds [5 x [5 x float]], [5 x [5 x float]]* %2, i64
      0, i64 %23

```

```

53  %25 = load i32, i32* %4, align 4
54  %26 = sext i32 %25 to i64
55  %27 = getelementptr inbounds [5 x float], [5 x float]* %24, i64 0, i64 %26
56  store float %21, float* %27, align 4
57  br label %28
58
59  28:
60  %29 = load i32, i32* %4, align 4
61  %30 = add nsw i32 %29, 1
62  store i32 %30, i32* %4, align 4
63  br label %10, !llvm.loop !6
64
65  31:
66  br label %32
67
68  32:
69  %33 = load i32, i32* %3, align 4
70  %34 = add nsw i32 %33, 1
71  store i32 %34, i32* %3, align 4
72  br label %6, !llvm.loop !8
73
74  35:
75  %36 = call i32 (i8*, ...) @__isoc99_scanf(i8* noundef getelementptr
    inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %5)
76  %37 = getelementptr inbounds [5 x [5 x float]], [5 x [5 x float]]* %2, i64
    0, i64 3
77  %38 = getelementptr inbounds [5 x float], [5 x float]* %37, i64 0, i64 3
78  %39 = load float, float* %38, align 4
79  %40 = load i32, i32* %5, align 4
80  %41 = sitofp i32 %40 to float
81  %42 = call float @add(float noundef %39, float noundef %41)
82  %43 = getelementptr inbounds [5 x [5 x float]], [5 x [5 x float]]* %2, i64
    0, i64 3
83  %44 = getelementptr inbounds [5 x float], [5 x float]* %43, i64 0, i64 3
84  store float %42, float* %44, align 4
85  %45 = getelementptr inbounds [5 x [5 x float]], [5 x [5 x float]]* %2, i64
    0, i64 3
86  %46 = getelementptr inbounds [5 x float], [5 x float]* %45, i64 0, i64 3
87  %47 = load float, float* %46, align 4
88  %48 = fpext float %47 to double
89  %49 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([3 x
    i8], [3 x i8]* @.str.1, i64 0, i64 0), double noundef %48)
90  ret i32 0
91  }
92
93  declare i32 @__isoc99_scanf(i8* noundef, ...) #1
94
95  declare i32 @printf(i8* noundef, ...) #1

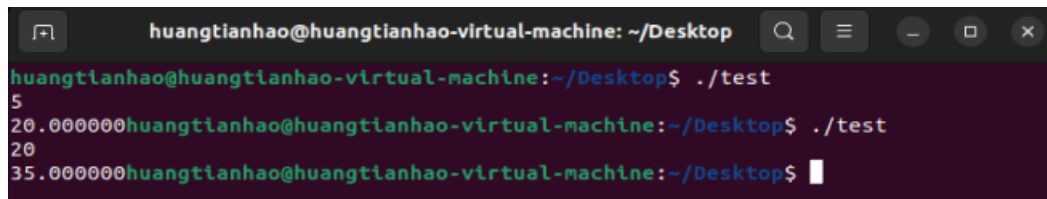
```

经过几条指令之后，可以得到最终的可执行文件。

```
● huangtianhao@huangtianhao-virtual-machine:~/Desktop$ llc test.ll -o test.S
● huangtianhao@huangtianhao-virtual-machine:~/Desktop$ gcc test.S -c -o test.o
● huangtianhao@huangtianhao-virtual-machine:~/Desktop$ gcc test.o -o test
```

图 13: 生成最终可执行文件的指令

在 terminal 运行该文件，测试表明准确无误。



```
huangtianhao@huangtianhao-virtual-machine: ~/Desktop
huangtianhao@huangtianhao-virtual-machine:~/Desktop$ ./test
5
20.000000huangtianhao@huangtianhao-virtual-machine:~/Desktop$ ./test
20
35.000000huangtianhao@huangtianhao-virtual-machine:~/Desktop$
```

图 14: 运行结果

三、 总结

通过本次实验，全面深入地了解了编译过程的具体步骤以及每一步的运行结果，初步了解了 LLVM IR 的编程，能够初步使用 LLVM IR 语言表现其余编程语言的特性。