



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

编译系统原理

了解你的编译器 LLVM IR

汤清云 2013536

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 10 月 2 日

摘要

完整的语言处理系统包括预处理器、编译器、汇编器与链接器加载器四个部分。完整编译过程大致为源程序经过预处理器得到经过预处理的源程序，在此程序中头文件、宏定义被展开，编译器处理后则产生了标准汇编语言程序，汇编代码由汇编器处理后产生可重定位的机器代码，最后链接器将其与一些库程序连接在一起形成最终的可执行的目标机器代码。本文在 Linux 环境下以 gcc、llvm 编译器为工具，对简单的 C 程序进行完成的编译过程探究，观察各个部分的输出内容，以期望对编译的完整过程和编译器的工作原理有更深入的认识。

关键词：预处理，编译，汇编，链接，gcc, llvm

目录

一、 引言	1
二、 实验目的	1
(一) 实验描述	1
三、 实验过程	1
(一) 预处理器	1
(二) 编译器	2
1. 词法分析	3
2. 语法分析	4
3. 语义分析	6
4. 中间代码生成	6
5. 代码优化	10
6. 代码生成	13
(三) 汇编器	14
(四) 链接器加载器	16
(五) LLVM IR 自编示例	22
1. 分工介绍	22
2. c 语言程序	22
3. 翻译所得到的中间代码	23
4. 问题解决	25
5. 总结分析	27

一、 引言

本实验以 GCC、LLVM 等编译器为研究对象，深入地探究语言处理系统的完整工作过程，主要回答了以下几个问题：

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？

此外本实验还通过编写 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、 实验目的

(一) 实验描述

本实验以阶乘的 C 源程序为例，使用 gcc、llvm 等编译器探究语言处理过程的完整工作过程，主要了解完整的编译过程以及预处理器、编译器、汇编器、链接器在此过程中的作用。在此实验中通过调整编译器的程序选项获得各阶段的输出，研究它们与源程序的关系。本实验使用的实验环境为 WSL 系统下的 Ubuntu18.04，辅助工具为 Flex+Bison 等分析工具。

具体 c 程序代码如下：

阶乘程序

```
1  #include<stdio.h>
2  int main()
3  {
4      int i,n,f;
5      scanf("%d",&n);
6      i=2;
7      f=1;
8      while(i<=n)
9      {
10         f=f*i;
11         i=i+1;
12     }
13     printf("%d\n",f);
14 }
```

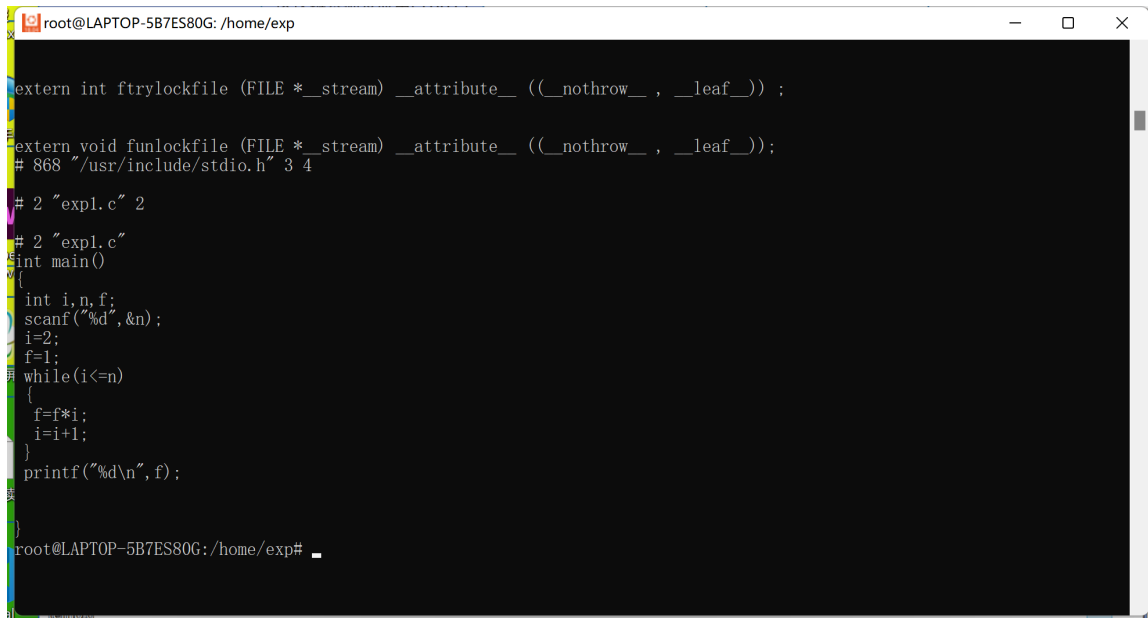
三、 实验过程

(一) 预处理器

预处理器会处理源代码中以 # 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。对于 gcc，我们可以使用如下命令使它输出预处理后的文件：

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

命令行中通过添加参数-E 令 gcc 只进行预处理过程，参数-o 改变 gcc 输出文件名，因此通过如上命令即可得到预处理后文件，其结果如图1所示



```
root@LAPTOP-5B7ES80G: /home/exp# gcc expl.c -E -o expl.i
extern int flockfile (FILE *_stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern void funlockfile (FILE *_stream) __attribute__ ((__nothrow__ , __leaf__));
# 868 "/usr/include/stdio.h" 3 4
# 2 "expl.c" 2
# 2 "expl.c"
int main()
{
    int i,n,f;
    scanf("%d",&n);
    i=2;
    f=1;
    while(i<=n)
    {
        f=f*i;
        i=i+1;
    }
    printf("%d\n",f);
}
root@LAPTOP-5B7ES80G: /home/exp#
```

图 1: 预处理器结果 expl.i

观察预处理文件，可以发现文件长度远大于源文件，结合课上所学我们可以知道，预处理器实际上就是将代码中的头文件（即 `<stdio.h>`）进行了简单的复制粘贴方式的替代。把头文件中的定义全部加到输出文件中，供后续编译程序对其进行处理。

以此输出为例，结合调查资料，可以总结预处理器的作用如下：[1]

- 将源文件中以“include”格式包含的文件复制到编译的源文件中。
- 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”。
- 删除所有的注释“//”和“/* */”。
- 添加行号和文件名标识，比如 `#2 "hello.c" 2`，以便编译时编译器产生调试用的行号信息及编译时产生编译错误或者警告时能够显示行号。

(二) 编译器

编译器做了很多工作，具体编译过程分为六个步骤：[5]

1. **词法分析**：也称作扫描，是编译器的第一个步骤，词法分析器读入组成源程序的字符流，并且将它们组织成为有意义的词素的序列，对于每一个词素，词法分析器产生词法单元作为输出。实际上就是从字符流到单词流的过程。

2. **语法分析**：也称作解析，语法分析器使用由词法分析器生成的各个词法单元的第一个分量来创建树形的中间表示。该中间表示给出了词法分析产生的词法单元流的语法结构。一个常用的表示方法是语法树，树中的每个内部结点表示一个运算，而该结点的子结点表示该运算的分量。

3. **语义分析**：语义分析器使用语法树和符号表中的信息来检查源程序是否和语言定义的语义一致。它同时也收集类型信息，并把这些信息存放在语法树或符号表中，以便在随后的中间代

码生成过程中使用。语义分析的一个重要部分是类型检查，编译器检查每个运算符是否具有匹配的运算分量，比如数组下标必须是整数，若用一个浮点数来做下标，编译器就会报错。程序设计语言可能允许某些类型转换，这被称作自动类型转换。

4. **中间代码生成**：源程序的语法分析和语义分析完成之后，很多编译器生成一个明确的低级的或类机器语言的中间表示，我们可以把这个表示看作是某个抽象机器的程序。该中间表示应该具有两个重要的性质：**易于生成**，且**能够被轻松地翻译为目标机器上的语言**。

5. **代码优化**：机器无关的代码优化步骤试图改进中间代码，以便生成更好的目标代码。代码优化又分为**中间代码优化**和**目标代码的优化**。

6. **代码生成**：代码生成器以源程序的中间表示形式作为输入，并把它映射到目标语言。如果目标语言是机器代码，那么必须为程序使用的每个变量选择寄存器或内存位置，然后，中间指令被翻译成为能够完成相同任务的机器指令序列。

1. 词法分析

根据词法规则识别出源程序中的各个单词 (token)，词法分析器的任务是将源程序转换为单词序列。对于 llvm, 可以使用如下命令获得 token 序列：

```
1 clang -E -Xclang -dump-tokens expl.c
```

其结果如图1所示

```
root@LAPTOP-5B7ES80G: /home/exp
numeric_constant '2'          Loc=<expl.c:6:4>
semi ';'                      Loc=<expl.c:6:5>
identifier 'f' [StartOfLine] [LeadingSpace] Loc=<expl.c:7:2>
equal '='                      Loc=<expl.c:7:3>
numeric_constant '1'          Loc=<expl.c:7:4>
semi ';'                      Loc=<expl.c:7:5>
while 'while' [StartOfLine] [LeadingSpace] Loc=<expl.c:8:2>
l_paren '('                      Loc=<expl.c:8:7>
identifier 'i'                  Loc=<expl.c:8:8>
lessequal '<='                  Loc=<expl.c:8:9>
identifier 'n'                  Loc=<expl.c:8:11>
r_paren ')'                      Loc=<expl.c:8:12>
l_brace '{'                      [StartOfLine] [LeadingSpace] Loc=<expl.c:9:2>
identifier 'f' [StartOfLine] [LeadingSpace] Loc=<expl.c:10:3>
equal '='                      Loc=<expl.c:10:4>
identifier 'f'                  Loc=<expl.c:10:5>
star '*'                        Loc=<expl.c:10:6>
identifier 'i'                  Loc=<expl.c:10:7>
semi ';'                      Loc=<expl.c:10:8>
identifier 'i' [StartOfLine] [LeadingSpace] Loc=<expl.c:11:3>
equal '='                      Loc=<expl.c:11:4>
identifier 'i'                  Loc=<expl.c:11:5>
plus '+'                        Loc=<expl.c:11:6>
numeric_constant '1'          Loc=<expl.c:11:7>
semi ';'                      Loc=<expl.c:11:8>
r_brace '}' [StartOfLine] [LeadingSpace] Loc=<expl.c:12:2>
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<expl.c:13:2>
l_paren '('                      Loc=<expl.c:13:8>
string_literal '"%d\n"'          Loc=<expl.c:13:9>
comma ','                      Loc=<expl.c:13:15>
identifier 'f'                  Loc=<expl.c:13:16>
r_paren ')'                      Loc=<expl.c:13:17>
semi ';'                      Loc=<expl.c:13:18>
r_brace '}' [StartOfLine] [LeadingSpace] Loc=<expl.c:16:1>
eof ''                          Loc=<expl.c:16:2>
root@LAPTOP-5B7ES80G: /home/exp#
```

图 2: 词法分析结果

可以看到词法分析完成了分词操作，从左到右对源程序进行扫描，按照语法规则识别各类单词，输出了每个单词的对应的属性字、单词自身以及单词在源程序中所在位置。

2. 语法分析

在这一阶段，编译器首先对代码进行语法检查，检查正确后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）。LLVM 可以通过如下命令获得相应的 AST：

```
1 clang -E -Xclang -ast-dump exp1.c
```

对于代码完全正确的程序，可以获得如下图所示的语法分析树：

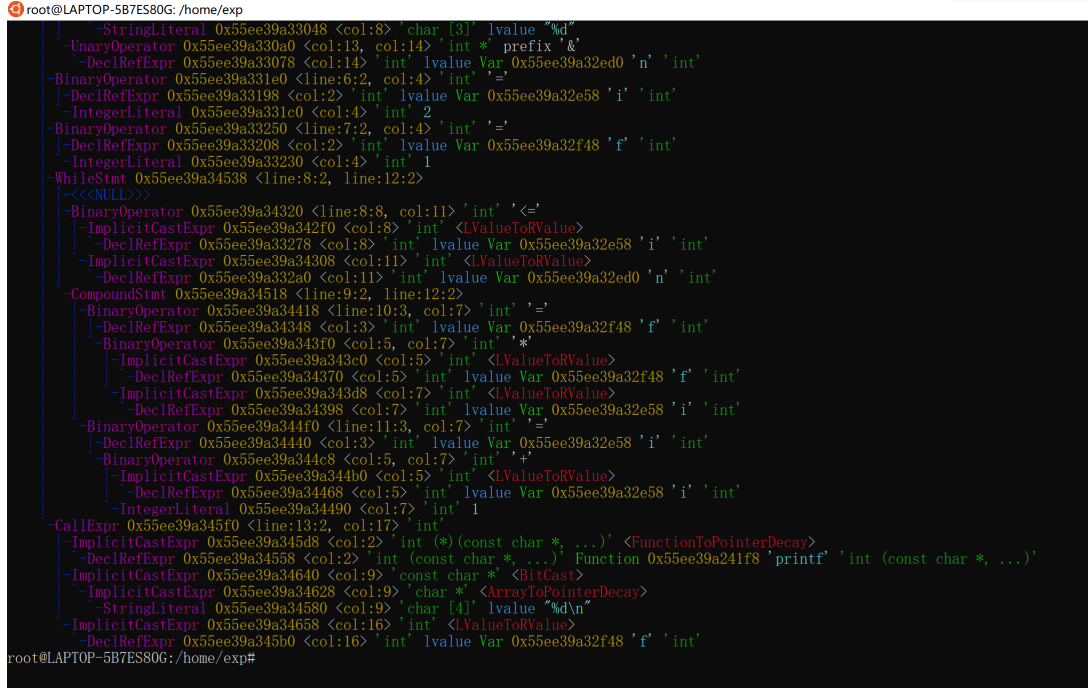


图 3: 语法分析结果

对于 gcc，我们可以使用如下代码获得文本格式的 AST 输出：[2]

```
1 gcc -fdump-tree-original-raw exp1.c
```

这一命令生成一个 exp1.c.003t.original 文件，其中就是源代码经过前端解析后生成的 AST 信息。

```

root@LAPTOP-5B7ES80G: /home/exp
root@LAPTOP-5B7ES80G: /home/exp# cat expl.c.003t.original

; Function main (null)
; enabled by -tree-original

01      statement_list  0 : @2      1 : @3
02      bind_expr       type: @4     vars: @5      body: @6
03      return_expr     type: @4     expr: @7
04      void_type       name: @8     algn: 8
05      var_decl        name: @9     type: @10     scpe: @11
                                srcp: expl.c:4      size: @12
                                algn: 32      used: 1
06      statement_list  0 : @13     1 : @14     2 : @15
                                3 : @16     4 : @17     5 : @18
                                6 : @19     7 : @20     8 : @21
                                9 : @22    10 : @23    11 : @24
                                12 : @25    13 : @26
07      modify_expr     type: @10    op 0: @27    op 1: @28
08      type_decl       name: @29    type: @4
09      identifier_node strg: i      lngt: 1
10      integer_type    name: @30    size: @12    algn: 32
                                prec: 32    sign: signed min : @31
                                max : @32
11      function_decl   name: @33    type: @34    srcp: expl.c:2
                                link: extern
12      integer_cst     type: @35    int: 32
13      decl_expr       type: @4
14      decl_expr       type: @4
15      decl_expr       type: @4

```

图 4: 文本形式 AST 输出

额外探索

而当代码语言出现错漏时，在语法分析这一步中也能分析出错漏并提示，例如我们给出以下代码作为 `expl_wrong.c` 文件代码：

```

expl_wrong.c
1  #include <stdio.h>
2  int main()
3  {
4      int i,n,f;
5      scanf("%d",&n);
6      i=2;
7      f=1;
8      //此处增加一个未定义的j进行判断
9      while(i<=n&&j==1)
10     {
11         f=f*i;
12         i=i+1;
13     }
14     printf("%d\n",f);
15 }

```

我们运行如下命令对 `expl_wrong` 文件进行代码检查：

```
1 gcc -I./math -fsyntax-only expl_wrong.c
```

得到如下图的提示，说明语法分析这一阶段能够对代码进行基本的检测：

```

root@LAPTOP-5B7ES80G:/home/exp# gcc -I./math -fsyntax-only expl_wrong.c
expl_wrong.c: In function 'main':
expl_wrong.c:9:14: error: 'j' undeclared (first use in this function)
    while(i<=n&& j==1)
                   ^
expl_wrong.c:9:14: note: each undeclared identifier is reported only once for each function it appears in
root@LAPTOP-5B7ES80G:/home/exp#

```

图 5: 错误代码语法分析结果

3. 语义分析

使用语法树和符号表中信息来检查源程序是否与语言定义语义一致，如类型检查和转换等，目的是保证语法正确的结构在语义上也是合法的。进行语义分析可以使用语法制导翻译的方法，符号表是存放有关标识符的信息的数据结构，当分析一个标识符的声明时，该标识符的信息被放入符号表，后来使用这个标识符时，语义动作将从符号表获取信息。

4. 中间代码生成

完成上述步骤后，很多编译器会生成一个明确的低级或类机器语言的中间表示。对于 gcc，可以使用如下命令获得中间代码的生成，生成的.dot 文件可以在 vscode 下载 graphviz 插件后查看。可以看到控制流图（CFG），以及各阶段处理中（比如优化、向 IR 转换）CFG 的变化：[\[3\]](#)

```

1 gcc -fdump-tree-all-graph expl.c//能够生成语法分析树
2 gcc -fdump-rtl-all-graph expl.c//能够生成所有的中间代码

```

运行这两条命令后,可以观察到当前文件夹下生成了很多新的文件。选择 expl.c.227t.optimized.dot 打开，其中记载了最优化之后的中间代码，具体如下所示

最优化后的中间代码

```

1  digraph "expl.c.227t.optimized" {
2  overlap=false;
3  subgraph "cluster_main" {
4      style="dashed";
5      color="black";
6      label="main ()";
7      subgraph cluster_0_1 {
8          style="filled";
9          color="darkgreen";
10         fillcolor="grey88";
11         label="loop 1";
12         labeljust=l;
13         penwidth=2;
14         fn_0_basic_block_4 [shape=record,style=filled,fillcolor=lightgrey,
                label="{  FREQ:0  |<bb\ 4>:\1\
15 |#\ i_2\  =\ PHI\  <i_6(2),\ i_12(3)>\>\1\
16 |#\ f_3\  =\ PHI\  <f_7(2),\ f_11(3)>\>\1\
17 |n.0_1\  =\ n;\1\
18 |if\ (i_2\  <=\ n.0_1)\1\
19 \ \ goto\  <bb\ 3>;\ [0.00%]\1\
20 else\1\
21 \ \ goto\  <bb\ 5>;\ [0.00%]\1\

```



```

22 }"];
23
24     fn_0_basic_block_3 [shape=record,style=filled,fillcolor=lightgrey,
25         label="{ FREQ:0 |<bb\ 3>:\1\
26 |f_11\ =\ f_3\ *\ i_2;\1\
27 |i_12\ =\ i_2\ +\ 1;\1\
28 }"];
29
30     fn_0_basic_block_0 [shape=Mdiamond,style=filled,fillcolor=white,label
31         ="ENTRY"];
32
33     fn_0_basic_block_1 [shape=Mdiamond,style=filled,fillcolor=white,label
34         ="EXIT"];
35
36     fn_0_basic_block_2 [shape=record,style=filled,fillcolor=lightgrey,
37         label="{ FREQ:0 |<bb\ 2>:\1\
38 |scanf\ ("%d",\ &n);\1\
39 |i_6\ =\ 2;\1\
40 |f_7\ =\ 1;\1\
41 goto\ \<bb\ 4>;\ [0.00%]\1\
42 }"];
43
44     fn_0_basic_block_5 [shape=record,style=filled,fillcolor=lightgrey,
45         label="{ FREQ:0 |<bb\ 5>:\1\
46 |printf\ ("%d\n",\ f_3);\1\
47 |n\ =\{v\}\ \{CLOBBER\};\1\
48 |_10\ =\ 0;\1\
49 }"];
50
51     fn_0_basic_block_6 [shape=record,style=filled,fillcolor=lightgrey,
52         label="{ FREQ:0 |<bb\ 6>:\1\
53 |<L3>\ [0.00%]:\1\
54 |return\ _10;\1\
55 }"];
56
57     fn_0_basic_block_0:s -> fn_0_basic_block_2:n [style="solid,bold",
58         color=blue,weight=100,constraint=true, label="[0%]"];
59     fn_0_basic_block_2:s -> fn_0_basic_block_4:n [style="solid,bold",
60         color=blue,weight=100,constraint=true, label="[0%]"];
61     fn_0_basic_block_3:s -> fn_0_basic_block_4:n [style="dotted,bold",
62         color=blue,weight=10,constraint=false, label="[0%]"];
63     fn_0_basic_block_4:s -> fn_0_basic_block_3:n [style="solid,bold",
64         color=black,weight=10,constraint=true, label="[0%]"];
65     fn_0_basic_block_4:s -> fn_0_basic_block_5:n [style="solid,bold",
66         color=black,weight=10,constraint=true, label="[0%]"];
67     fn_0_basic_block_5:s -> fn_0_basic_block_6:n [style="solid,bold",
68         color=blue,weight=100,constraint=true, label="[0%]"];

```

```

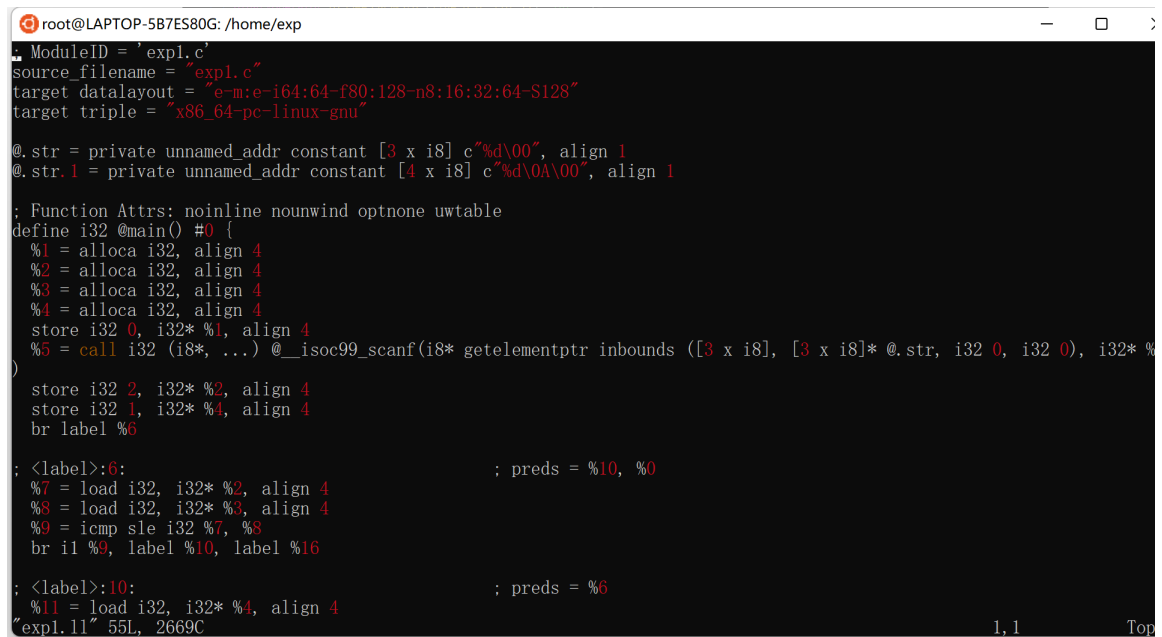
58     fn_0_basic_block_6:s -> fn_0_basic_block_1:n [style="solid,bold",
        color=black,weight=10,constraint=true, label="[0%]"];
59     fn_0_basic_block_0:s -> fn_0_basic_block_1:n [style="invis",
        constraint=true];
60 }
61 }

```

对于 llvm, 我们使用如下命令生成 LLVM IR:

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

运行命令后生成文件 expl.ll



```

root@LAPTOP-5B7ES80G: /home/exp
; ModuleID = 'expl.c'
source_filename = "expl.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

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

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    %5 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i32 0, i32 0), i32* %
)
    store i32 2, i32* %2, align 4
    store i32 1, i32* %4, align 4
    br label %6

; <label>:6:                                ; preds = %10, %0
    %7 = load i32, i32* %2, align 4
    %8 = load i32, i32* %3, align 4
    %9 = icmp sle i32 %7, %8
    br i1 %9, label %10, label %16

; <label>:10:                                ; preds = %6
    %11 = load i32, i32* %4, align 4
    "expl.ll" 55L, 2669C

```

图 6: expl.ll

具体代码为:

```

1 ; ModuleID = 'expl.c'
2 source_filename = "expl.c"
3 target datalayout = "e-m:e-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 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     store i32 0, i32* %1, align 4

```

```

16  %5 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x
    i8], [3 x i8]* @.str, i32 0, i32 0), i32* %3)
17  store i32 2, i32* %2, align 4
18  store i32 1, i32* %4, align 4
19  br label %6
20
21  ; <label>:6:                                ; preds = %10, %0
22  %7 = load i32, i32* %2, align 4
23  %8 = load i32, i32* %3, align 4
24  %9 = icmp sle i32 %7, %8
25  br i1 %9, label %10, label %16
26
27  ; <label>:10:                                ; preds = %6
28  %11 = load i32, i32* %4, align 4
29  %12 = load i32, i32* %2, align 4
30  %13 = mul nsw i32 %11, %12
31  store i32 %13, i32* %4, align 4
32  %14 = load i32, i32* %2, align 4
33  %15 = add nsw i32 %14, 1
34  store i32 %15, i32* %2, align 4
35  br label %6
36
37  ; <label>:16:                                ; preds = %6
38  %17 = load i32, i32* %4, align 4
39  %18 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4
    x i8]* @.str.1, i32 0, i32 0), i32 %17)
40  %19 = load i32, i32* %1, align 4
41  ret i32 %19
42 }
43
44 declare i32 @__isoc99_scanf(i8*, ...) #1
45
46 declare i32 @printf(i8*, ...) #1
47
48 attributes #0 = { noline nounwind optnone uwtable "correctly-rounded-divide
    -sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad
    "="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"
    "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="
    false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false" "
    stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features
    "="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float
    "="false" }
49 attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-
    tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim
    "="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-
    nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math
    "="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target
    -features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-

```

```

    soft-float="false" }
50
51 !llvm.module.flags = !{!0}
52 !llvm.ident = !{!1}
53
54 !0 = !{i32 1, !"wchar_size", i32 4}
55 !1 = !{"clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)"}

```

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。在 LLVM 官网对所有 pass 的分类 3 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。

在本实验中选用 llvm 现有的 pass 进行代码优化探索。在 llvm 的架构中，pass 的作用就是优化 llvm IR, pass 作用于 llvm IR, 并且处理和分析 IR, 寻找优化机会并修改 IR, 从而产生优化的代码。下面的命令行工具 opt 就是用来在 llvm IR 上运行各种优化 pass 的。[4]

LLVM 可以通过下面的命令生成每个 pass 后生成的 LLVM IR, 以观察差别:

```
1 llc -print-before-all -print-after-all exp1.ll > exp1.log 2>&1
```

这一命令生成两个文件，分别为 exp1.log 和 exp1.s

```

root@LAPTOP-5B7ES80G: /home/exp
renamable %rsi = LEA64r %rbp, 1, %noreg, -12, %noreg
MOV32mi %rbp, 1, %noreg, -16, %noreg, 0; mem:ST4[%1]
%al = MOV8ri 0
CALL64prel32 @_isoc99_scanf, <regmask %bb %bl %bp %bp %bx %bp %bx %rbp %rbx %r12 %r13 %r14 %r15 %r12b %r13b %r14b %r15b %r12d %r13d %r14d %r15d %r12w %r13w %r14w %r15w>, implicit %rsp, implicit %ssp, implicit %al, implicit %rdi, implicit %rsi, implicit-def %eax
MOV32mi %rbp, 1, %noreg, -4, %noreg, 2; mem:ST4[%2]
MOV32mi %rbp, 1, %noreg, -8, %noreg, 1; mem:ST4[%4]
Successors according to CFG: %bb.1

bb.1: derived from LLVM BB %6
Predecessors according to CFG: %bb.0 %bb.2
renamable %eax = MOV32rm %rbp, 1, %noreg, -4, %noreg; mem:LD4[%2]
CMP32rm killed renamable %eax, %rbp, 1, %noreg, -12, %noreg, implicit-def %eflags; mem:LD4[%3]
JG 1 %bb.3, implicit killed %eflags
Successors according to CFG: %bb.3 %bb.2

bb.2: derived from LLVM BB %10
Predecessors according to CFG: %bb.1
renamable %eax = MOV32rm %rbp, 1, %noreg, -8, %noreg; mem:LD4[%4]
renamable %eax = IMUL32rm killed renamable %eax, %rbp, 1, %noreg, -4, %noreg, implicit-def dead %eflags; mem:LD4[%2]
MOV32mr %rbp, 1, %noreg, -8, %noreg, killed renamable %eax; mem:ST4[%4]
renamable %eax = MOV32rm %rbp, 1, %noreg, -4, %noreg; mem:LD4[%2]
renamable %eax = ADD32ri8 killed renamable %eax, 1, implicit-def dead %eflags
MOV32mr %rbp, 1, %noreg, -4, %noreg, killed renamable %eax; mem:ST4[%2]
JMP 1 %bb.1
Successors according to CFG: %bb.1

bb.3: derived from LLVM BB %16
Predecessors according to CFG: %bb.1
renamable %rdi = MOV64ri @.str.1
renamable %esi = MOV32rm %rbp, 1, %noreg, -8, %noreg; mem:LD4[%4]
%al = MOV8ri 0
CALL64prel32 @printf, <regmask %bb %bl %bp %bp %bx %bp %bx %rbp %rbx %r12 %r13 %r14 %r15 %r12b %r13b %r14b %r15b %r12d %r13d %r14d %r15d %r12w %r13w %r14w %r15w>, implicit %rsp, implicit %ssp, implicit %al, implicit %rdi, implicit %esi, implicit-def %eax
renamable %eax = MOV32rm %rbp, 1, %noreg, -16, %noreg; mem:LD4[%1]
%rsp = ADD64ri8 %rsp, 16, implicit-def dead %eflags; flags: FrameDestroy
%rbp = POP64ri8 implicit-def %rsp, implicit %rsp; flags: FrameDestroy
RETQ implicit %eax

# End machine code for function main.
root@LAPTOP-5B7ES80G: /home/exp#

```

图 7: exp1.log

```

root@LAPTOP-5B7ES80G: /home/exp
movl    %eax, -4(%rbp)
jmp     .LBB0_1
.LBB0_3:
movabsq $.L.str.1, %rdi
movl    -8(%rbp), %esi
movb    $0, %al
callq   printf
movl    -16(%rbp), %eax
addq    $16, %rsp
popq    %rbp
retq
.Lfunc_end0:
.size   main, .Lfunc_end0-main
.cfi_endproc

.type   .L.str,@object          # -- End function
.section .rodata.str1.1, "aMS",@progbits,1
.L.str:
.asciz  "%d"
.size   .L.str, 3

.type   .L.str.1,@object        # @.str.1
.L.str.1:
.asciz  "%d\n"
.size   .L.str.1, 4

.ident  "clang version 6.0.0-lubuntu2 (tags/RELEASE_600/final)"
.section ".note.GNU-stack","",@progbits
root@LAPTOP-5B7ES80G: /home/exp#

```

图 8: exp1.s

对于 gcc, 可以使用如下命令进行代码优化:

```

1 gcc -O0 exp1.c
2 gcc -Os exp1.c > s.out 2>&1
3 gcc -O3 exp1.c > 3.out 2>&1

```

O0 为不进行优化, 由上至下优化等级不断提高, 会依次生成 a.out;s.out;3.out 三个文件。在运行-O3 命令时返回错误如下图, 故而将源代码改为:

```

1 int temp=scanf("%d",&n);

```

```

root@LAPTOP-5B7ES80G: /home/exp# cat 3.out
exp1.c: In function 'main':
exp1.c:5:2: warning: ignoring return value of 'scanf', declared with attribute warn_unused_result [-Wunused-result]
scanf("%d",&n);

```

图 9: 报错信息

可以通过下面的命令将 ll 形式 LLVM IR 转化为 bc 格式 (即二进制码), 以统一文件格式:

```

1 llvm-as exp1.ll -o exp1.bc

```

图 10: exp1.bc

同样，也可以通过命令指定使用某个 pass 以生成 LLVM IR，以特别观察某个 pass 的差别；这里使用 Transform Passes 中的 -adce 参数，将中间代码中的死代码进行删除：

```
1 opt -adce exp1.bc > exp1_adce.bc
2 llvm-dis exp1_adce.bc -o exp1_adce.ll
```

生成文件及部分内容如下截图

图 11: exp1_adce.ll

与之前生成的未经过死代码删除的 exp1.ll 文件相比可以发现没有变化；说明生成中间代码中没有死代码的出现。

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。编译器将高级语言源程序转换为汇编语言程序体现在文件上为将.i 文件转换为.S 文件。

使用如下命令分别生成 x86 格式、arm 格式和 llvm 的目标格式代码：

```
1 gcc exp1.i -S -o exp1.S//x86
2 arm-linux-gnueabi-gcc exp1.i -S -o exp1_arm.S//arm
3 llc exp1.ll -o exp1_llvm.S//llvm
```

代码部分截图如下



```
root@LAPTOP-5B7ES80G: /home/exp
    jmp     .L2
.L3:
    movl    -12(%rbp), %eax
    imull    -16(%rbp), %eax
    movl    %eax, -12(%rbp)
    addl    $1, -16(%rbp)
.L2:
    movl    -20(%rbp), %eax
    cmpl    %eax, -16(%rbp)
    jle     .L3
    movl    -12(%rbp), %eax
    movl    %eax, %esi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    movq    -8(%rbp), %rdx
    xorq    %fs:40, %rdx
    je      .L5
    call    __stack_chk_fail@PLT
.L5:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size    main, .-main
    .ident   "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
    .section .note.GNU-stack,"",@progbits
root@LAPTOP-5B7ES80G:/home/exp#
```

图 12: exp1_x86.S

```

root@LAPTOP-5B7ES80G: /home/exp
.LPIC2:
    ldr    r3, .L6+12
    add    r3, pc
    mov    r0, r3
    bl     printf(PLT)
    movs   r3, #0
    mov    r0, r3
    ldr    r3, .L6+4
    ldr    r3, [r4, r3]
    ldr    r2, [r7, #12]
    ldr    r3, [r3]
    cmp    r2, r3
    beq    .L5
    bl     __stack_chk_fail(PLT)
.L5:
    adds   r7, r7, #20
    mov    sp, r7
    @ sp needed
    pop    {r4, r7, pc}
.L7:
    .align 2
.L6:
    .word   _GLOBAL_OFFSET_TABLE - (.LPIC0+4)
    .word   __stack_chk_guard(GOT)
    .word   .LC0-(.LPIC1+4)
    .word   .LC1-(.LPIC2+4)
    .size   main, .-main
    .ident  "GCC: (Ubuntu/Linaro 7.5.0-3ubuntu1~18.04) 7.5.0"
    .section .note.GNU-stack,"",%progbits
root@LAPTOP-5B7ES80G: /home/exp#

```

图 13: exp1_arm.S

```

root@LAPTOP-5B7ES80G: /home/exp
    movl   %eax, -4(%rbp)
    jmp    .LBB0_1
.LBB0_3:
    movabsq $.L.str.1, %rdi
    movl   -8(%rbp), %esi
    movb   $0, %al
    callq  printf
    movl   -16(%rbp), %eax
    addq   $16, %rsp
    popq   %rbp
    retq
.Lfunc_end0:
    .size   main, .Lfunc_end0-main
    .cfi_endproc
    # -- End function
    .type   .L.str.@object      # @.str
    .section .rodata.str1.1, "aMS",@progbits,1
.L.str:
    .asciz  "%d"
    .size   .L.str, 3
    .type   .L.str.1.@object    # @.str.1
.L.str.1:
    .asciz  "%d\n"
    .size   .L.str.1, 4
    .ident  "clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)"
    .section .note.GNU-stack,"",@progbits
root@LAPTOP-5B7ES80G: /home/exp#

```

图 14: exp1_llvm.S

(三) 汇编器

汇编过程实际上把汇编语言程序代码翻译成目标机器指令的过程。其最终生成的是可重定位的机器代码。这一步一般被视为编译过程的“后端”，体现在文件上为将.S 文件转换为.o 文件。可以使用如下命令生成.o 文件：

- 1 gcc exp1.S -c -o exp1_x86.o //x86
- 2 arm-linux-gnueabi-gcc exp1_arm.S -o exp1_arm.o //arm 这一步需要用到交叉编译
- 3 llc exp1.bc -filetype=obj -o exp1_llvm.o //llvm

在 gcc 命令中 -c 表示只编译，不链接成为可执行文件。生成的 test.o 为二进制文件，可以使用 vscode 中的 Hexdump 插件查看，也可以使用 GUN 的 objdump 进行反汇编，具体命令为：

```
1 objdump -d expl_x86.o >expl_x86_o.txt 2>&1 \\x86
2 objdump -d expl_arm.o >expl_arm_o.txt 2>&1 \\arm
3 objdump -d expl_llvm.o >expl_llvm_o.txt 2>&1 \\llvm
```

以上命令生成的.o 文件和反汇编后得到的 txt 文件如下图所示

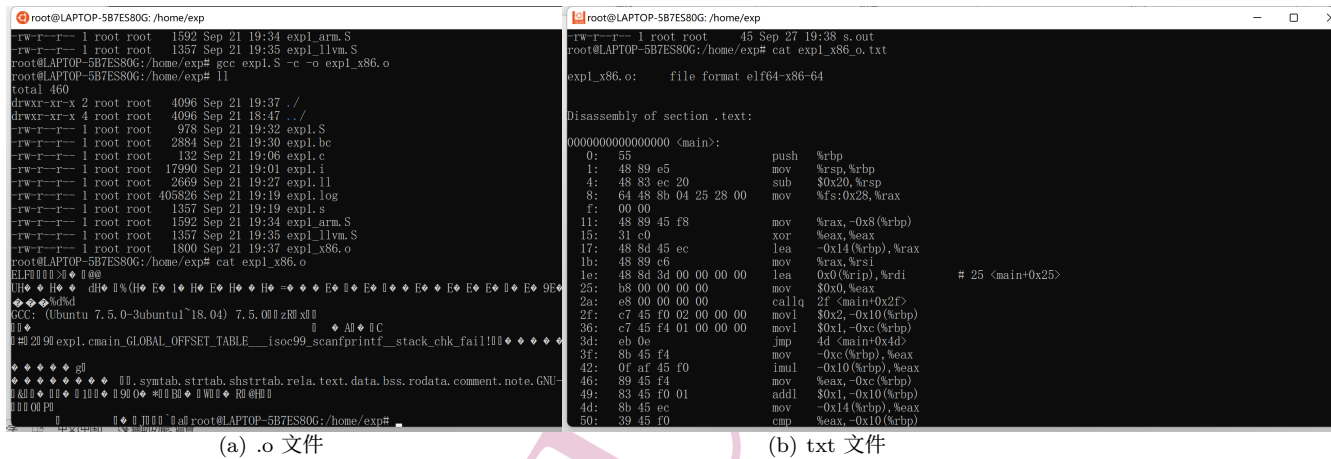


图 15: x86 格式下汇编所得

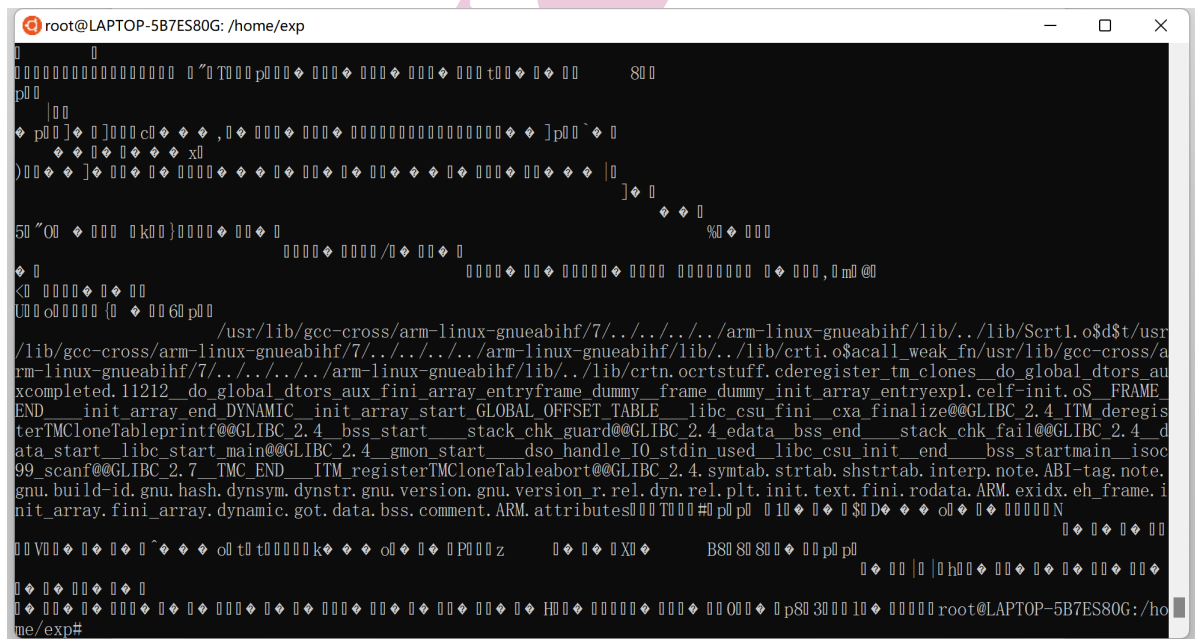


图 16: expl_arm.o

而由于 arm 格式使用的是交叉编译，故而转化为 txt 格式后无法在命令行中显示；此外，观察可知 llvm 和 x86 两种格式下反汇编所得的代码完全一致。

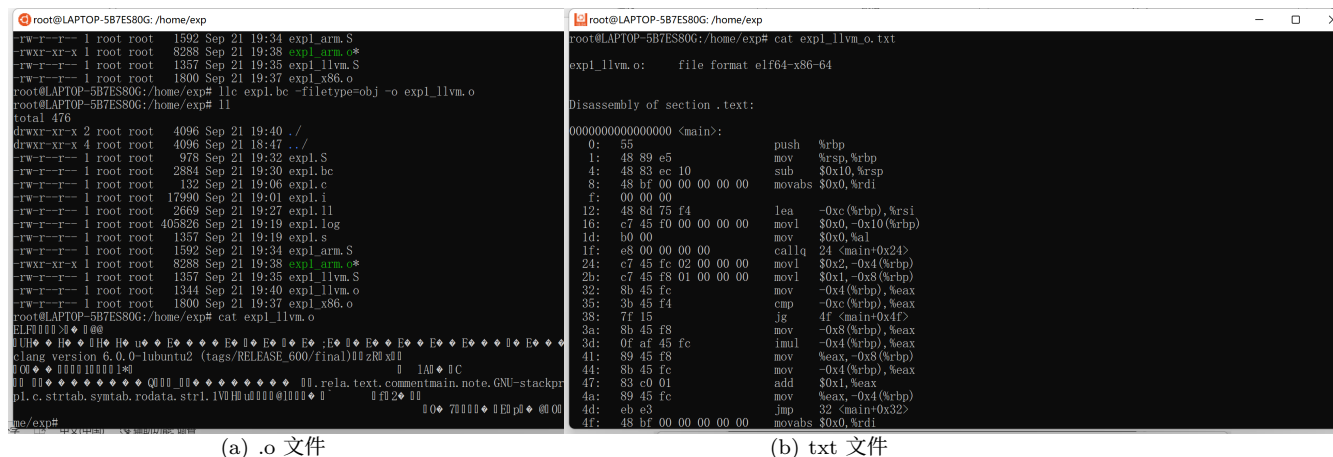


图 17: llvm 格式下汇编所得

(四) 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。代码命令如下：

```
1 gcc exp1_x86.o -o exp1_x86
2 arm-linux-gnueabi-hf-gcc exp1.c -o exp1_arm //对于arm格式，由于其需要交叉编译，故而是从.c文件直接生成32位可执行文件
3 gcc exp1_llvm.o -o exp1_llvm
```

运行以上命令可以生成可执行文件，使用如下命令可以运行生成的可执行文件，此处以 x86 为例：

```
1 | ./exp1_x86
```

```
root@LAPTOP-5B7ES80G:/home/exp# ./exp1_x86_64
362880
root@LAPTOP-5B7ES80G:/home/exp#
```

图 18: exp1 arm.S

执行以下命令对可执行文件进行反汇编：

```
1 objdump -d exp1_x86 >exp1_x86.txt 2>&1
```

生成反汇编代码如下:

```

1  exp1_x86:      file format elf64-x86-64
2
3
4  Disassembly of section .init:
5
6  000000000000005a8 <_init>:
7    5a8:  48 83 ec 08          sub     $0x8,%rsp

```

```

8  5ac:  48 8b 05 35 0a 20 00    mov     0x200a35(%rip),%rax      # 200fe8 <
    __gmon_start__>
9  5b3:  48 85 c0                  test    %rax,%rax
10 5b6:  74 02                     je      5ba <__init+0x12>
11 5b8:  ff d0                     callq   *%rax
12 5ba:  48 83 c4 08               add     $0x8,%rsp
13 5be:  c3                       retq
14
15 Disassembly of section .plt:
16
17 00000000000005c0 <.plt>:
18 5c0:  ff 35 ea 09 20 00        pushq   0x2009ea(%rip)          # 200fb0 <
    _GLOBAL_OFFSET_TABLE_+0x8>
19 5c6:  ff 25 ec 09 20 00        jmpq    *0x2009ec(%rip)        # 200fb8 <
    _GLOBAL_OFFSET_TABLE_+0x10>
20 5cc:  0f 1f 40 00              nopl    0x0(%rax)
21
22 00000000000005d0 <__stack_chk_fail@plt>:
23 5d0:  ff 25 ea 09 20 00        jmpq    *0x2009ea(%rip)        # 200fc0 <
    __stack_chk_fail@GLIBC_2.4>
24 5d6:  68 00 00 00 00          pushq   $0x0
25 5db:  e9 e0 ff ff ff          jmpq    5c0 <.plt>
26
27 00000000000005e0 <printf@plt>:
28 5e0:  ff 25 e2 09 20 00        jmpq    *0x2009e2(%rip)        # 200fc8 <
    printf@GLIBC_2.2.5>
29 5e6:  68 01 00 00 00          pushq   $0x1
30 5eb:  e9 d0 ff ff ff          jmpq    5c0 <.plt>
31
32 00000000000005f0 <__isoc99_scanf@plt>:
33 5f0:  ff 25 da 09 20 00        jmpq    *0x2009da(%rip)        # 200fd0 <
    __isoc99_scanf@GLIBC_2.7>
34 5f6:  68 02 00 00 00          pushq   $0x2
35 5fb:  e9 c0 ff ff ff          jmpq    5c0 <.plt>
36
37 Disassembly of section .plt.got:
38
39 0000000000000600 <__cxa_finalize@plt>:
40 600:  ff 25 f2 09 20 00        jmpq    *0x2009f2(%rip)        # 200ff8 <
    __cxa_finalize@GLIBC_2.2.5>
41 606:  66 90                     xchg    %ax,%ax
42
43 Disassembly of section .text:
44
45 0000000000000610 <_start>:
46 610:  31 ed                     xor     %ebp,%ebp
47 612:  49 89 d1                 mov     %rdx,%r9
48 615:  5e                       pop     %rsi

```

```

49 616: 48 89 e2          mov    %rsp,%rdx
50 619: 48 83 e4 f0       and    $0xfffffffffffff0,%rsp
51 61d: 50              push   %rax
52 61e: 54              push   %rsp
53 61f: 4c 8d 05 ea 01 00 00 lea     0x1ea(%rip),%r8      # 810 <
    __libc_csu_fini>
54 626: 48 8d 0d 73 01 00 00 lea     0x173(%rip),%rcx    # 7a0 <
    __libc_csu_init>
55 62d: 48 8d 3d e6 00 00 00 lea     0xe6(%rip),%rdi    # 71a <main>
56 634: ff 15 a6 09 20 00   callq  *0x2009a6(%rip)    # 200fe0 <
    __libc_start_main@GLIBC_2.2.5>
57 63a: f4              hlt
58 63b: 0f 1f 44 00 00     nopl   0x0(%rax,%rax,1)
59
60 0000000000000640 <deregister_tm_clones>:
61 640: 48 8d 3d c9 09 20 00 lea     0x2009c9(%rip),%rdi    # 201010 <
    TMC_END_>
62 647: 55              push   %rbp
63 648: 48 8d 05 c1 09 20 00 lea     0x2009c1(%rip),%rax    # 201010 <
    TMC_END_>
64 64f: 48 39 f8         cmp    %rdi,%rax
65 652: 48 89 e5         mov    %rsp,%rbp
66 655: 74 19           je     670 <deregister_tm_clones+0x30>
67 657: 48 8b 05 7a 09 20 00 mov     0x20097a(%rip),%rax    # 200fd8 <
    ITM_deregisterTMCloneTable>
68 65e: 48 85 c0         test   %rax,%rax
69 661: 74 0d           je     670 <deregister_tm_clones+0x30>
70 663: 5d              pop     %rbp
71 664: ff e0           jmpq   *%rax
72 666: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
73 66d: 00 00 00
74 670: 5d              pop     %rbp
75 671: c3              retq
76 672: 0f 1f 40 00     nopl   0x0(%rax)
77 676: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
78 67d: 00 00 00
79
80 0000000000000680 <register_tm_clones>:
81 680: 48 8d 3d 89 09 20 00 lea     0x200989(%rip),%rdi    # 201010 <
    TMC_END_>
82 687: 48 8d 35 82 09 20 00 lea     0x200982(%rip),%rsi    # 201010 <
    TMC_END_>
83 68e: 55              push   %rbp
84 68f: 48 29 fe         sub    %rdi,%rsi
85 692: 48 89 e5         mov    %rsp,%rbp
86 695: 48 c1 fe 03      sar    $0x3,%rsi
87 699: 48 89 f0         mov    %rsi,%rax
88 69c: 48 c1 e8 3f      shr    $0x3f,%rax

```

```

89  6a0:  48 01 c6          add    %rax,%rsi
90  6a3:  48 d1 fe          sar    %rsi
91  6a6:  74 18             je     6c0 <register_tm_clones+0x40>
92  6a8:  48 8b 05 41 09 20 00 mov    0x200941(%rip),%rax          # 200ff0 <
    __ITM_registerTMCloneTable>
93  6af:  48 85 c0          test   %rax,%rax
94  6b2:  74 0c             je     6c0 <register_tm_clones+0x40>
95  6b4:  5d                pop    %rbp
96  6b5:  ff e0             jmpq   *%rax
97  6b7:  66 0f 1f 84 00 00 00 nopw   0x0(%rax,%rax,1)
98  6be:  00 00
99  6c0:  5d                pop    %rbp
100 6c1:  c3                retq
101 6c2:  0f 1f 40 00       nopl   0x0(%rax)
102 6c6:  66 2e 0f 1f 84 00 00 nopw   %cs:0x0(%rax,%rax,1)
103 6cd:  00 00 00
104
105 000000000000006d0 <__do_global_dtors_aux>:
106 6d0:  80 3d 39 09 20 00 00 cmpb   $0x0,0x200939(%rip)          # 201010 <
    __TMC_END__>
107 6d7:  75 2f             jne    708 <__do_global_dtors_aux+0x38>
108 6d9:  48 83 3d 17 09 20 00 cmpq   $0x0,0x200917(%rip)          # 200ff8 <
    __cxa_finalize@GLIBC_2.2.5>
109 6e0:  00
110 6e1:  55                push   %rbp
111 6e2:  48 89 e5          mov    %rsp,%rbp
112 6e5:  74 0c             je     6f3 <__do_global_dtors_aux+0x23>
113 6e7:  48 8b 3d 1a 09 20 00 mov    0x20091a(%rip),%rdi          # 201008 <
    __dso_handle>
114 6ee:  e8 0d ff ff ff    callq  600 <__cxa_finalize@plt>
115 6f3:  e8 48 ff ff ff    callq  640 <deregister_tm_clones>
116 6f8:  c6 05 11 09 20 00 01 movb   $0x1,0x200911(%rip)          # 201010 <
    __TMC_END__>
117 6ff:  5d                pop    %rbp
118 700:  c3                retq
119 701:  0f 1f 80 00 00 00 00 nopl   0x0(%rax)
120 708:  f3 c3             repz   retq
121 70a:  66 0f 1f 44 00 00 nopw   0x0(%rax,%rax,1)
122
123 00000000000000710 <frame_dummy>:
124 710:  55                push   %rbp
125 711:  48 89 e5          mov    %rsp,%rbp
126 714:  5d                pop    %rbp
127 715:  e9 66 ff ff ff    jmpq   680 <register_tm_clones>
128
129 0000000000000071a <main>:
130 71a:  55                push   %rbp
131 71b:  48 89 e5          mov    %rsp,%rbp

```

```

132 71e: 48 83 ec 20      sub    $0x20,%rsp
133 722: 64 48 8b 04 25 28 00 mov    %fs:0x28,%rax
134 729: 00 00
135 72b: 48 89 45 f8      mov    %rax,-0x8(%rbp)
136 72f: 31 c0           xor    %eax,%eax
137 731: 48 8d 45 ec      lea    -0x14(%rbp),%rax
138 735: 48 89 c6      mov    %rax,%rsi
139 738: 48 8d 3d e5 00 00 00 lea    0xe5(%rip),%rdi    # 824 <
    _IO_stdin_used+0x4>
140 73f: b8 00 00 00 00    mov    $0x0,%eax
141 744: e8 a7 fe ff ff    callq  5f0 <__isoc99_scanf@plt>
142 749: c7 45 f0 02 00 00 00 movl    $0x2,-0x10(%rbp)
143 750: c7 45 f4 01 00 00 00 movl    $0x1,-0xc(%rbp)
144 757: eb 0e           jmp     767 <main+0x4d>
145 759: 8b 45 f4      mov    -0xc(%rbp),%eax
146 75c: 0f af 45 f0    imul   -0x10(%rbp),%eax
147 760: 89 45 f4      mov    %eax,-0xc(%rbp)
148 763: 83 45 f0 01    addl   $0x1,-0x10(%rbp)
149 767: 8b 45 ec      mov    -0x14(%rbp),%eax
150 76a: 39 45 f0      cmp    %eax,-0x10(%rbp)
151 76d: 7e ea           jle     759 <main+0x3f>
152 76f: 8b 45 f4      mov    -0xc(%rbp),%eax
153 772: 89 c6      mov    %eax,%esi
154 774: 48 8d 3d ac 00 00 00 lea    0xac(%rip),%rdi    # 827 <
    _IO_stdin_used+0x7>
155 77b: b8 00 00 00 00    mov    $0x0,%eax
156 780: e8 5b fe ff ff    callq  5e0 <printf@plt>
157 785: b8 00 00 00 00    mov    $0x0,%eax
158 78a: 48 8b 55 f8      mov    -0x8(%rbp),%rdx
159 78e: 64 48 33 14 25 28 00 xor    %fs:0x28,%rdx
160 795: 00 00
161 797: 74 05           je      79e <main+0x84>
162 799: e8 32 fe ff ff    callq  5d0 <__stack_chk_fail@plt>
163 79e: c9           leaveq  %eax,%edi
164 79f: c3           retq
165
166 00000000000007a0 <__libc_csu_init>:
167 7a0: 41 57           push   %r15
168 7a2: 41 56           push   %r14
169 7a4: 49 89 d7      mov    %rdx,%r15
170 7a7: 41 55           push   %r13
171 7a9: 41 54           push   %r12
172 7ab: 4c 8d 25 f6 05 20 00 lea    0x2005f6(%rip),%r12    # 200da8 <
    __frame_dummy_init_array_entry>
173 7b2: 55           push   %rbp
174 7b3: 48 8d 2d f6 05 20 00 lea    0x2005f6(%rip),%rbp    # 200db0 <
    __init_array_end>
175 7ba: 53           push   %rbx

```

```

176 7bb: 41 89 fd      mov     %edi,%r13d
177 7be: 49 89 f6      mov     %rsi,%r14
178 7c1: 4c 29 e5      sub     %r12,%rbp
179 7c4: 48 83 ec 08   sub     $0x8,%rsp
180 7c8: 48 c1 fd 03   sar     $0x3,%rbp
181 7cc: e8 d7 fd ff ff callq   5a8 <__init>
182 7d1: 48 85 ed      test    %rbp,%rbp
183 7d4: 74 20        je      7f6 <__libc_csu_init+0x56>
184 7d6: 31 db        xor     %ebx,%ebx
185 7d8: 0f 1f 84 00 00 00 00 nopl    0x0(%rax,%rax,1)
186 7df: 00
187 7e0: 4c 89 fa      mov     %r15,%rdx
188 7e3: 4c 89 f6      mov     %r14,%rsi
189 7e6: 44 89 ef      mov     %r13d,%edi
190 7e9: 41 ff 14 dc   callq   *(%r12,%rbx,8)
191 7ed: 48 83 c3 01   add     $0x1,%rbx
192 7f1: 48 39 dd      cmp     %rbx,%rbp
193 7f4: 75 ea        jne     7e0 <__libc_csu_init+0x40>
194 7f6: 48 83 c4 08   add     $0x8,%rsp
195 7fa: 5b           pop     %rbx
196 7fb: 5d           pop     %rbp
197 7fc: 41 5c        pop     %r12
198 7fe: 41 5d        pop     %r13
199 800: 41 5e        pop     %r14
200 802: 41 5f        pop     %r15
201 804: c3          retq
202 805: 90          nop
203 806: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
204 80d: 00 00 00
205
206 0000000000000810 <__libc_csu_fini>:
207 810: f3 c3      repz retq
208
209 Disassembly of section .fini:
210
211 0000000000000814 <_fini>:
212 814: 48 83 ec 08   sub     $0x8,%rsp
213 818: 48 83 c4 08   add     $0x8,%rsp
214 81c: c3          retq

```

实际上，上述代码实现的反汇编使用的是动态链接的反汇编方式，它不会将库的内容复制到可执行程序中，因此反汇编所得到的代码较少，使用如下代码则可以实现静态链接生成.s 文件并进行反汇编：

```

1 gcc expl_x86.o -static -o expl_x86_static
2 objdump -d expl_x86_static > expl_x86_static.txt 2>&1

```

这样生成的代码极长（170774 行），在此不做全部展示，截取部分代码如下图：

```

root@LAPTOP-5B7E580G: /home/exp
expl_x86_static: file format elf64-x86-64

Disassembly of section .init:
0000000000400418 <.init>:
400418: 48 83 ec 08      sub    $0x8,%rsp
40041c: 48 c7 c0 00 00 00 mov    $0x0,%rax
400423: 48 85 c0         test   %rax,%rax
400426: 74 02           je     40042a <_.init+0x12>
400428: ff d0          callq  *%rax
40042a: 48 83 c4 08      add    $0x8,%rsp
40042e: c3            retq

Disassembly of section .plt:
0000000000400430 <.plt>:
400430: ff 25 e2 3b 2d 00 jmpq   *0x2d3be2(%rip) # 6d4018 <_GLOBAL_OFFSET_TABLE_+0x18>
400436: 66 90          xchg   %ax,%ax
400438: ff 25 e2 3b 2d 00 jmpq   *0x2d3be2(%rip) # 6d4020 <_GLOBAL_OFFSET_TABLE_+0x20>
40043e: 66 90          xchg   %ax,%ax
400440: ff 25 e2 3b 2d 00 jmpq   *0x2d3be2(%rip) # 6d4028 <_GLOBAL_OFFSET_TABLE_+0x28>
400446: 66 90          xchg   %ax,%ax
400448: ff 25 e2 3b 2d 00 jmpq   *0x2d3be2(%rip) # 6d4030 <_GLOBAL_OFFSET_TABLE_+0x30>
40044e: 66 90          xchg   %ax,%ax
400450: ff 25 e2 3b 2d 00 jmpq   *0x2d3be2(%rip) # 6d4038 <_GLOBAL_OFFSET_TABLE_+0x38>
400456: 66 90          xchg   %ax,%ax
400458: ff 25 e2 3b 2d 00 jmpq   *0x2d3be2(%rip) # 6d4040 <_GLOBAL_OFFSET_TABLE_+0x40>
"expl_x86_static.txt" 170774L, 9389300C

```

图 19: 静态链接反汇编

(五) LLVM IR 自编示例

1. 分工介绍

在本小节，我与实验队友朱莞尔共同对以下 c 语言程序进行了手动翻译成 llvm 中间代码的工作，具体分工如下：

1. 共同编写 c 语言程序，完成以下六个模块：int 型数据定义、变量与常量的声明和初始化、语句（赋值、if、while、return、for）撰写、表达式（算术运算、逻辑运算、关系运算）撰写、注释和输入输出。朱莞尔负责前三部分，汤清云负责后三部分，两人共同商讨组合成一个整体程序。具体程序代码见后文。

2. 共同研究中间代码语句的撰写模式并进行讨论和查找资料研究，以阶乘程序的中间语言作为样例试着研究明白其逻辑。

3. 进行分工，朱莞尔同学负责对代码的两个 for 循环和之前变量声明进行翻译工作，我负责对内层 while 循环和 if 判定语句以及 printf 等函数调用的模块进行翻译工作。而后两人共同讨论将翻译得到的代码尽心那个组装和 debug，使其最终能够编译为可运行程序。

4. 总结本次实验中遇到的问题和解决的方法以及途径，为之后的学习夯实基础。

2. c 语言程序

本程序的工作为统计 111 444 之间个位、十位、百位三数字各不相同的数字总个数，并按所输入的 temp 进行对总数 sum 运算。特别地，当 temp=1 时，输出结果即为这样数字的总的个数。

```

1 #include <stdio.h>
2 int main()
3 {
4     int sum;//变量声明
5     sum=0;//变量初始化
6     const int one=1;//常量声明与初始化
7     int temp;
8     int ans=scanf("%d",&temp);
9     for(int i=1;i<5;i++)

```



```

10     {
11         for (int a=1;a<5;a++)
12         {
13             int b=1;
14             while(b<5)
15             {
16                 if (i!=a&& i!=b&&a!=b)
17                 {
18                     printf("%d%d%d",i,a,b);
19                     putchar("\n");
20                     sum=sum+1*temp;
21                 }
22                 b=b+1;
23             }
24         }
25     }
26     printf("%d",sum);
27     return 0;
28 }

```

3. 翻译所得到的中间代码

翻译结果如下:

```

1  define i32 @main() #0 {
2      %1 = alloca i32, align 4                ;固定的寄存器
3      %2 = alloca i32, align 4                ;int sum
4      %3 = alloca i32, align 4                ;const int one
5      %4 = alloca i32, align 4                ;int temp
6      %5 = alloca i32, align 4                ;int ans
7      %6 = alloca i32, align 4                ;int i
8      %7 = alloca i32, align 4                ;int a
9      %8 = alloca i32, align 4                ;int b
10     store i32 0, i32* %1, align 4            ;将0 (i32) 存入%1 (i32*)
11     store i32 0, i32* %2, align 4            ;sum=0
12     store i32 1, i32* %3, align 4            ;one=1
13     ;调用@scanf函数, i32表示函数的返回值类型
14     %9 = call i32 @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8
        ], [3 x i8]* @.str, i32 0, i32 0), i32* %4)
15     store i32 %9, i32* %5, align 4            ;%9里存储scanf返回
        值, 赋值给%5即ans
16     store i32 1, i32* %6, align 4            ;i=1
17     br label %10                             ;代码无条件跳转至代码
        块10
18
19 10:                                           ; preds = %51, %0
20     %11 = load i32, i32* %6, align 4
21     %12 = icmp slt i32 %11, 5                ;有符号整数i与5比较
22     br i1 %12, label %13, label %54

```

```

23
24 13:                                ; preds = %10
25     store i32 1, i32* %7, align 4      ;a=1
26     br label %14                        ; 往后无条件跳转
27
28 14:                                ; preds = %47, %13
29     %15 = load i32, i32* %7, align 4    ;a与5比较
30     %16 = icmp slt i32 %15, 5
31     br i1 %16, label %17, label %50
32
33 17:                                ; preds = %14
34     store i32 1, i32* %8, align 4      ;b=1
35     br label %18                        ; 无条件跳转
36
37 18:                                ; preds = %43, %17
38     %19 = load i32, i32* %8, align 4
39     %20 = icmp slt i32 %19, 5           ;b跟5比较
40     br i1 %20, label %21, label %46
41
42 21:                                ; preds = %18
43     %22 = load i32, i32* %6, align 4
44     %23 = load i32, i32* %7, align 4
45     %24 = icmp ne i32 %22, %23         ; 比较i与a是否相等
46     br i1 %24, label %25, label %43
47
48 25:                                ; preds = %21
49     %26 = load i32, i32* %6, align 4
50     %27 = load i32, i32* %8, align 4
51     %28 = icmp ne i32 %26, %27         ; 比较i和b是否相等
52     br i1 %28, label %29, label %43
53
54 29:                                ; preds = %25
55     %30 = load i32, i32* %7, align 4
56     %31 = load i32, i32* %8, align 4
57     %32 = icmp ne i32 %30, %31         ; 比较a和b是否相等
58     br i1 %32, label %33, label %43
59
60 33:                                ; preds = %29
61     %34 = load i32, i32* %6, align 4
62     %35 = load i32, i32* %7, align 4
63     %36 = load i32, i32* %8, align 4
64 ; 调用函数@printf
65     %37 = call i32 @@printf(i8* getelementptr inbounds ([7 x i8], [7
        x i8]* @.str.1, i32 0, i32 0), i32 %34, i32 %35, i32 %36)
66     %38 = call i32 @putchar(i32 ptrtoint ([2 x i8]* @.str.2 to i32))
67     %39 = load i32, i32* %2, align 4
68     %40 = load i32, i32* %4, align 4
69     %41 = mul nsw i32 1, %40           ; temp与1相乘

```

```

70 %42 = add nsw i32 %39, %41 ;相加的值赋值给sum
71 store i32 %42, i32* %2, align 4
72 br label %43
73
74 43: ; preds = %33, %29, %25, %21
75 %44 = load i32, i32* %8, align 4
76 %45 = add nsw i32 %44, 1
77 store i32 %45, i32* %8, align 4 ;b的新值赋值给自己
78 br label %18 ;返回代码块18, 使得b
    继续与5比较
79
80 46: ; preds = %18 ;对应b>=5的情况
81 br label %47
82
83 47: ; preds = %46
84 %48 = load i32, i32* %7, align 4
85 %49 = add nsw i32 %48, 1 ;a++
86 store i32 %49, i32* %7, align 4
87 br label %14 ;返回代码块14, 使得a
    继续与5比较
88
89 50: ; preds = %14
90 br label %51 ;对应a>=5的情况
91
92 51: ; preds = %50
93 %52 = load i32, i32* %6, align 4
94 %53 = add nsw i32 %52, 1 ;i++
95 store i32 %53, i32* %6, align 4 ;重新赋值给自己
96 br label %10
97
98 54: ; preds = %10 ;i!=5的情况, 直接跳转
    到代码的最后
99 %55 = load i32, i32* %2, align 4
100 %56 = call i32 @__isoc99_scanf(i8*, ...) @printf(i8* getelementptr @inbounds ([3 x i8], [3
    x i8]* @.str, i32 0, i32 0), i32 %55)
101 ret i32 0 ;主函数结束, 返回0
102 }
103
104 declare i32 @__isoc99_scanf(i8*, ...) #1
105
106 declare i32 @printf(i8*, ...) #1
107
108 declare i32 @putchar(i32) #1

```

4. 问题解决

但当我们对自己编写的中间代码进行汇编链接时, 发现以下错误:

```

root@LAPTOP-5B7ES80G:/home/exp1# vim temp.ll
root@LAPTOP-5B7ES80G:/home/exp1# llvm-as temp.ll -o temp.bc
5llvm-as: temp.ll:13:101: error: use of undefined value '@.str'
      %9 = call i32 @i8*, ...) @_isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]*
      @.str, i32 0, i32 0), i32* %4)

```

图 20: 汇编链接错误一

```

root@LAPTOP-5B7ES80G:/home/exp1# gcc temp.o -o temp
/usr/bin/ld: temp.o: relocation R_X86_64_32 against `.rodata.str.1' can not be used when making a PIE object; recompile
with -fPIC
/usr/bin/ld: final link failed: Nonrepresentable section on output
collect2: error: ld returned 1 exit status
root@LAPTOP-5B7ES80G:/home/exp1#

```

图 21: 汇编链接错误二

在进行了资料查询以及对原本 c 程序进行比较后, 我们发现错误原因在于我们缺少了以下代码, 增加后再次汇编链接成功。代码文件之前增加:

```

1 ; ModuleID = 'test.c'
2 source_filename = "test.c"
3 target datalayout = "e-m:e-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 [7 x i8] c"%d%d\00", align 1
8 @.str.2 = private unnamed_addr constant [2 x i8] c"\0A\00", align 1

```

代码文件末尾增加:

```

1 attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
2 attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
3
4 !llvm.module.flags = !{!0}
5 !llvm.ident = !{!1}
6
7 !0 = !{i32 1, !"wchar_size", i32 4}
8 !1 = !{!"clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)"}

```

添加后再次执行汇编链接, 程序运行成功结果图如下:

```

root@LAPTOP-5B7ES80G:/home/exp1# vim test.c
root@LAPTOP-5B7ES80G:/home/exp1# clang -S -emit-llvm test.c
root@LAPTOP-5B7ES80G:/home/exp1# llvm-as test.ll -o test.bc
root@LAPTOP-5B7ES80G:/home/exp1# llc test.bc -filetype=obj -o test.o
root@LAPTOP-5B7ES80G:/home/exp1# gcc test.o -o test
root@LAPTOP-5B7ES80G:/home/exp1# ./test
1
123
124
132
134
142
143
213
214
231
234
241
243
312
314
321
324
341
342
412
413
421
423
431
432
24

```

图 22: 成功效果图

5. 总结分析

在此次实验过程中, 我和朱莞尔同学一起对 llvm ir 代码进行了学习与探索, 在撰写代码过程中出现多次寄存器使用不匹配问题, 后来逐一修正; 而当多次出现链接异常时, 我与朱莞尔主动在各个平台进行相关资料的查询, 在完成实验的同时深化了自身对中间代码的理解。

在实验完成后, 我们总结了如下 llvm 语言特性: 1. 所有的全局变量都以 @ 为前缀, 后面的 global 关键字表明了它是一个全局变量。

2. 函数定义以 'define' 开头, i32 标明了函数的返回类型, 其中 'add'、'main' 是函数的名字, '@' 是其前缀。

3. 以 % 开头的符号表示为临时寄存器, 必须连续使用。

4. 在进行数值比较时必须使用 i32 型的寄存器, 而不能使用 i32* 型, 否则会出现类型不匹配问题, 故而必须使用 load 指令

5. LLVM IR 的基本单位成为 module (只要是单文件编译就只涉及单 module), 对应 SysY 中的 CompUnit——CompUnit ::= [CompUnit] (Decl | FuncDef), 一个 CompUnit 中有且仅有一个 main 函数定义, 是程序的入口。

6. 一个 module 中可以包含多个顶层实体, 如 function 和 global variable, CompUnit 的顶层变量/常量声明语句 (对应 Decl), 函数定义 (对应 FuncDef) 都不可以重复定义同名标识符 (IDENT), 即便标识符的类型不同也不允许

7. 一个 function define 中至少有一个 basicblock。basicblock 对应 SysY 中的 Block 语句块, 语句块内声明的变量的生存期在该语句块内。Block 表示为 Block ::= " " BlockItem " " ;

BlockItem ::= Decl | Stmt;

8. 每个 basicblock 中有若干 instruction, 且都以 terminator instruction 结尾。SysY 中语句表示为 Stmt ::= LVal " = " Exp " ;"

| [Exp] " ;"

```
| Block  
| " if" ( " Exp " ) " Stmt [ " else" Stmt ]  
| " while" ( " Exp " ) " Stmt  
| " break" ;  
| " continue" ;  
| " return" [Exp] " ;" ;
```

9. llvm IR 中注释以; 开头, SysY 中与 C 语言一致

10. llvm IR 是静态类型的, 即每个值的类型在编写时是确定的

11. llvm IR 中全局变量和函数都以 @ 开头, 且会在类型 (如 i32) 之前用 global 标明, 局部变量以 % 开头, 其作用域是单个函数, 临时寄存器 (上文中的 %1 等) 以升序阿拉伯数字命名

12. 函数定义的语法可以总结为: define + 返回值 (i32) + 函数名 (@main) + 参数列表 ((i32 %a, i32 %b)) + 函数体 (ret i32 0), 函数声明你可以在 main.ll 的最后看到, 即用 declare 替换 define。SysY 中函数定义表示为 FuncDef ::= FuncType IDENT "(" [FuncFParams] ")" Block

13. 终结指令一定位于一个基本块的末尾, 如 ret 指令会令程序控制流返回到函数调用者, br 指令会根据后续标识符的结果进行下一个基本块的跳转, br 指令包含无条件 (br+label) 和有条件 (br+ 标志符 +truelabel+falselabel) 两种

14. i32 这个变量类型实际上就指 32 bit 长的 integer, 类似的还有 void、label、array、pointer 等

15. 绝大多数指令的含义就是其字面意思, load 从内存读值, store 向内存写值, add 相加参数, alloca 分配内存并返回地址等

参考文献

- [1] C 基础: 预处理器. https://blog.csdn.net/qq_43194080/article/details/125507637.
- [2] Gcc - gimple ir 学习一. https://https://blog.csdn.net/qq_36287943/article/details/105458166.
- [3] 了解 gcc 和 llvm·熟悉使用过程·观察中间文件. <https://blog.csdn.net/zhj12399/article/details/123194092>.
- [4] 基于 llvm 的代码优化. <https://blog.csdn.net/zcmuczx/article/details/80855017>.
- [5] 编译器的各个步骤. <https://blog.csdn.net/zoweiccc/article/details/82556601>.

NIJUB