南開大學
Nankai University

南开大学

计算机学院和网络空间安全学院

编译系统原理实验报告

了解编译器及 LLVM IR 编程

朱子轩 2110853 计算机科学与技术

刘修铭 2112492 信息安全

年级：2021 级

指导教师：王刚

2023 年 9 月 12 日

摘要

本次实验由朱子轩与刘修铭合作完成，主要通过改进给定的阶乘 C 语言程序代码，在 Ubuntu 虚拟机上对编译器的各个阶段及其功能进行了学习与探索。除了完成实验手册的要求内容外，我们还通过加入注释、引用头文件、宏定义、死代码等部分对编译的预处理部分功能进行验证性探索，加深对于词法分析与语法分析的过程的认识；同时，借助 VS code 的插件，完成了将 CFG 的可视化，进而实现对中间代码生成的多阶段的分析；代码优化部分，我们通过对 O0 O1 O2 等不同优化等级的对比，以程序的运行时间为参考，验证优化效果。除此之外，我们还对 X86、ARM 和 LLVM 的汇编和链接结果分别予以实现，并加以探索。最后，我们学习了 LLVM IR 的语法规则，并编写了一段 LLVM IR 程序，加深对 LLVM IR 中间语言的了解。

关键字： 预处理器，编译器，汇编器，链接器，加载器，优化对比，交叉编译，性能测试，LLVM IR

目录

一、 分工

小组二人均独立完成所给阶乘 C 语言部分的编译复现工作，共同完成 LLVM IR 部分编写工作，并由刘修铭撰写实验报告，朱子轩予以修改补充。

实验所用代码及生成文件均打包附后。

二、 预备工作及实验平台

按照实验指导，我们选用 Ubuntu 虚拟机配置了实验环境。实验平台如下：

设备名称	lxmliu2002-Ubuntu
系统名称	Ubuntu 22.04.3 LTS
操作系统类型	Linux 64 位
GNOME 版本	42.9
窗口系统	X11
虚拟化	VMware

表 1: 实验平台参数

同时，为了完成后续探索工作，我们修改了给定的阶乘程序。修改后的代码如下：

阶乘

```c
// 引用头文件
#include <stdio.h>
#include <time.h>
// 宏定义
#define MAX 10000
int main()
{
    int i, n, f;
    scanf("%d", &n);
    i = 2;
    f = 1;
    clock_t start, end;
    double time;
    start = clock();
    // 死代码
    if (i < 1)
    {
        printf("i小于1\n");
    }
    while (i <= n)
    {
        f = f * i;
        i = i + 1;
    }
    if (f > MAX)
    {
        printf("输出超限\n");
    }
    else
    {
        printf("%d\n", f);
    }
```

```
33      end = clock();
34      time = (double)(end - start) / CLOCKS_PER_SEC;
35      printf("程序运行时间为: %f\n", time);
36      return 0;
37  }
```

三、 实验过程

借助 clang -ccc-print-phases main.c 指令，我们得到编译的整个流程如图1所示：

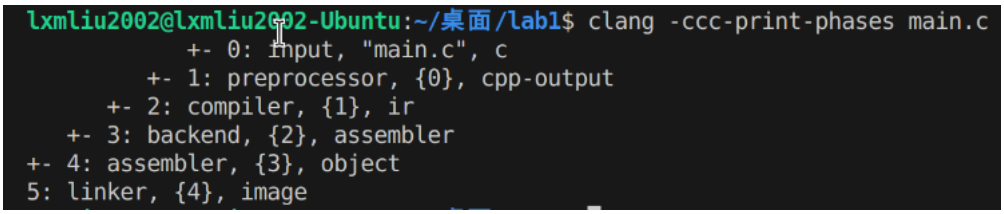


图 1: 编译各阶段

(一) 预处理器

1. 预处理阶段功能

编译器的预处理阶段是编译过程的第一个阶段，它在实际的编译之前执行，主要完成以下任务：

- 处理和预处理以 # 符号开头指令。
- 展开宏定义，将宏名称替换为其定义的文本。
- 根据条件编译指令（如 #ifdef、#ifndef、#if、#elif 等）来控制哪些部分的代码应该包含在编译中。
- 移除注释，删除多余的空格、制表符和换行符，以简化源代码。
- 将指定的头文件内容插入到源代码中，从而形成一个整体的源代码文件，以便后续编译阶段使用。

预处理阶段的主要目标是将源代码准备好，以便后续的编译阶段能够进行词法分析、语法分析和生成中间代码。完成预处理后，生成的预处理文件通常会保存为.i 或.ii 扩展名的文件，然后传递给编译器的下一个阶段进行编译。

2. 实验验证

为了验证以上功能，我们将给定的阶乘代码予以改进，实现了对上述功能的验证。

首先，执行 gcc main.c -E -o main.i 指令，得到预处理后的 main.i 文件。然后观察生成的 main.i 文件并将其与 main.c 文件进行对比，可以发现：

- 代码的长度增加，由 37 行变为 1067 行，观察可知，引用的头文件已经插入其中。
- 一些无关内容被预处理删除。
- 宏名称已经被替换。

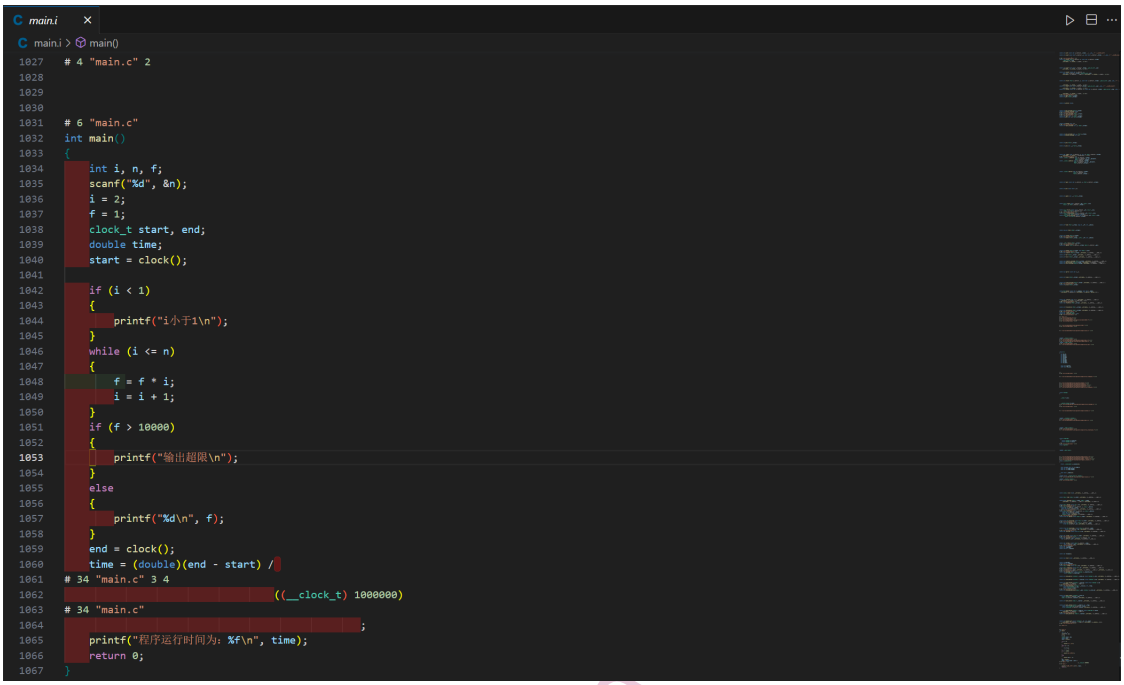


图 2: 预处理结果

根据以上实验结果可知，预处理功能得以验证。

(二) 编译器

1. 词法分析

词法分析阶段的主要目标是将源代码分割成词法单元，以便后续的语法分析阶段能够理解代码的结构和语法。

借助 clang -E -Xclang -dump-tokens main.c 命令，我们得到 token 序列：

词法分析结果

```
int 'int'	[StartOfLine]	Loc=<main.c:6:1>
identifier 'main'	[LeadingSpace]	Loc=<main.c:6:5>
l_paren '('	Loc=<main.c:6:9>
r_paren ')'	Loc=<main.c:6:10>
l_brace '{'	[StartOfLine]	Loc=<main.c:7:1>
int 'int'	[StartOfLine] [LeadingSpace]	Loc=<main.c:8:5>
identifier 'i'	[LeadingSpace]	Loc=<main.c:8:9>
comma ','	Loc=<main.c:8:10>
identifier 'n'	[LeadingSpace]	Loc=<main.c:8:12>
comma ','	Loc=<main.c:8:13>
identifier 'f'	[LeadingSpace]	Loc=<main.c:8:15>
semi ';'	Loc=<main.c:8:16>
identifier 'scanf'	[StartOfLine] [LeadingSpace]	Loc=<main.c:9:5>
l_paren '('	Loc=<main.c:9:10>
string_literal '"%d"'	Loc=<main.c:9:11>
comma ','	Loc=<main.c:9:15>
amp '&'	[LeadingSpace]	Loc=<main.c:9:17>
```

```
identifier 'n'          Loc=<main.c:9:18>
r_paren ')'             Loc=<main.c:9:19>
semi ';'                Loc=<main.c:9:20>
identifier 'i'  [StartOfLine] [LeadingSpace]   Loc=<main.c:10:5>
equal '='       [LeadingSpace] Loc=<main.c:10:7>
numeric_constant '2'    [LeadingSpace] Loc=<main.c:10:9>
semi ';'                Loc=<main.c:10:10>
identifier 'f'  [StartOfLine] [LeadingSpace]   Loc=<main.c:11:5>
equal '='       [LeadingSpace] Loc=<main.c:11:7>
numeric_constant '1'    [LeadingSpace] Loc=<main.c:11:9>
semi ';'                Loc=<main.c:11:10>
identifier 'clock_t'    [StartOfLine] [LeadingSpace]   Loc=<main.c:12:5>
identifier 'start'      [LeadingSpace] Loc=<main.c:12:13>
comma ','               Loc=<main.c:12:18>
identifier 'end'        [LeadingSpace] Loc=<main.c:12:20>
semi ';'                Loc=<main.c:12:23>
double 'double' [StartOfLine] [LeadingSpace]   Loc=<main.c:13:5>
identifier 'time'       [LeadingSpace] Loc=<main.c:13:12>
semi ';'                Loc=<main.c:13:16>
identifier 'start'      [StartOfLine] [LeadingSpace]   Loc=<main.c:14:5>
equal '='       [LeadingSpace] Loc=<main.c:14:11>
identifier 'clock'      [LeadingSpace] Loc=<main.c:14:13>
l_paren '('             Loc=<main.c:14:18>
r_paren ')'             Loc=<main.c:14:19>
semi ';'                Loc=<main.c:14:20>
if 'if' [StartOfLine] [LeadingSpace]   Loc=<main.c:16:5>
l_paren '('     [LeadingSpace] Loc=<main.c:16:8>
identifier 'i'          Loc=<main.c:16:9>
less '<'        [LeadingSpace] Loc=<main.c:16:11>
numeric_constant '1'    [LeadingSpace] Loc=<main.c:16:13>
r_paren ')'             Loc=<main.c:16:14>
l_brace '{'     [StartOfLine] [LeadingSpace]   Loc=<main.c:17:5>
identifier 'printf'     [StartOfLine] [LeadingSpace]   Loc=<main.c:18:9>
l_paren '('             Loc=<main.c:18:15>
string_literal '"i小于1\n"'             Loc=<main.c:18:16>
r_paren ')'             Loc=<main.c:18:28>
semi ';'                Loc=<main.c:18:29>
r_brace '}'     [StartOfLine] [LeadingSpace]   Loc=<main.c:19:5>
while 'while'   [StartOfLine] [LeadingSpace]   Loc=<main.c:20:5>
l_paren '('     [LeadingSpace] Loc=<main.c:20:11>
identifier 'i'          Loc=<main.c:20:12>
lessequal '<='  [LeadingSpace] Loc=<main.c:20:14>
identifier 'n'  [LeadingSpace] Loc=<main.c:20:17>
r_paren ')'             Loc=<main.c:20:18>
l_brace '{'     [StartOfLine] [LeadingSpace]   Loc=<main.c:21:5>
identifier 'f'  [StartOfLine] [LeadingSpace]   Loc=<main.c:22:9>
equal '='       [LeadingSpace] Loc=<main.c:22:11>
identifier 'f'  [LeadingSpace] Loc=<main.c:22:13>
```

```
star '*'        [LeadingSpace] Loc=<main.c:22:15>
identifier 'i'  [LeadingSpace] Loc=<main.c:22:17>
semi ';'                Loc=<main.c:22:18>
identifier 'i'  [StartOfLine] [LeadingSpace]   Loc=<main.c:23:9>
equal '='       [LeadingSpace] Loc=<main.c:23:11>
identifier 'i'  [LeadingSpace] Loc=<main.c:23:13>
plus '+'        [LeadingSpace] Loc=<main.c:23:15>
numeric_constant '1'    [LeadingSpace] Loc=<main.c:23:17>
semi ';'                Loc=<main.c:23:18>
r_brace '}'     [StartOfLine] [LeadingSpace]   Loc=<main.c:24:5>
if 'if' [StartOfLine] [LeadingSpace]   Loc=<main.c:25:5>
l_paren '('     [LeadingSpace] Loc=<main.c:25:8>
identifier 'f'          Loc=<main.c:25:9>
greater '>'     [LeadingSpace] Loc=<main.c:25:11>
numeric_constant '10000'        [LeadingSpace] Loc=<main.c:25:13 <Spelling=
    main.c:5:13>>
r_paren ')'             Loc=<main.c:25:16>
l_brace '{'     [StartOfLine] [LeadingSpace]   Loc=<main.c:26:5>
identifier 'printf'     [StartOfLine] [LeadingSpace]   Loc=<main.c:27:9>
l_paren '('             Loc=<main.c:27:15>
string_literal '"输出超限\n"'           Loc=<main.c:27:16>
r_paren ')'             Loc=<main.c:27:32>
semi ';'                Loc=<main.c:27:33>
r_brace '}'     [StartOfLine] [LeadingSpace]   Loc=<main.c:28:5>
else 'else'     [StartOfLine] [LeadingSpace]   Loc=<main.c:29:5>
l_brace '{'     [StartOfLine] [LeadingSpace]   Loc=<main.c:30:5>
identifier 'printf'     [StartOfLine] [LeadingSpace]   Loc=<main.c:31:9>
l_paren '('             Loc=<main.c:31:15>
string_literal '"%d\n"'         Loc=<main.c:31:16>
comma ','               Loc=<main.c:31:22>
identifier 'f'  [LeadingSpace] Loc=<main.c:31:24>
r_paren ')'             Loc=<main.c:31:25>
semi ';'                Loc=<main.c:31:26>
r_brace '}'     [StartOfLine] [LeadingSpace]   Loc=<main.c:32:5>
identifier 'end'        [StartOfLine] [LeadingSpace]   Loc=<main.c:33:5>
equal '='       [LeadingSpace] Loc=<main.c:33:9>
identifier 'clock'      [LeadingSpace] Loc=<main.c:33:11>
l_paren '('             Loc=<main.c:33:16>
r_paren ')'             Loc=<main.c:33:17>
semi ';'                Loc=<main.c:33:18>
identifier 'time'       [StartOfLine] [LeadingSpace]   Loc=<main.c:34:5>
equal '='       [LeadingSpace] Loc=<main.c:34:10>
l_paren '('     [LeadingSpace] Loc=<main.c:34:12>
double 'double'         Loc=<main.c:34:13>
r_paren ')'             Loc=<main.c:34:19>
l_paren '('             Loc=<main.c:34:20>
identifier 'end'                Loc=<main.c:34:21>
minus '-'       [LeadingSpace] Loc=<main.c:34:25>
```

```
identifier 'start'        [LeadingSpace] Loc=<main.c:34:27>
r_paren ')'               Loc=<main.c:34:32>
slash '/'          [LeadingSpace] Loc=<main.c:34:34>
l_paren '('        [LeadingSpace] Loc=<main.c:34:36 <Spelling=/usr/include/
    x86_64-linux-gnu/bits/time.h:34:25>>
l_paren '('               Loc=<main.c:34:36 <Spelling=/usr/include/x86_64-linux
    -gnu/bits/time.h:34:26>>
identifier '__clock_t'           Loc=<main.c:34:36 <Spelling=/usr/include/
    x86_64-linux-gnu/bits/time.h:34:27>>
r_paren ')'               Loc=<main.c:34:36 <Spelling=/usr/include/x86_64-linux
    -gnu/bits/time.h:34:36>>
numeric_constant '1000000'        [LeadingSpace] Loc=<main.c:34:36 <Spelling=/
    usr/include/x86_64-linux-gnu/bits/time.h:34:38>>
r_paren ')'               Loc=<main.c:34:36 <Spelling=/usr/include/x86_64-linux
    -gnu/bits/time.h:34:45>>
semi ';'                  Loc=<main.c:34:50>
identifier 'printf'       [StartOfLine] [LeadingSpace]   Loc=<main.c:35:5>
l_paren '('               Loc=<main.c:35:11>
string_literal '"程序运行时间为: %f\n"'          Loc=<main.c:35:12>
comma ','                 Loc=<main.c:35:42>
identifier 'time'         [LeadingSpace] Loc=<main.c:35:44>
r_paren ')'               Loc=<main.c:35:48>
semi ';'                  Loc=<main.c:35:49>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<main.c:36:5>
numeric_constant '0'      [LeadingSpace] Loc=<main.c:36:12>
semi ';'                  Loc=<main.c:36:13>
r_brace '}'       [StartOfLine]  Loc=<main.c:37:1>
eof ''           Loc=<main.c:37:2>
```

观察可知，在词法分析阶段，源程序中的字符串被扫描并分解，识别成为一个个单词，并被写明其类型，便于后续的语法分析。

2. 语法分析

将词法分析生成的词法单元来构建抽象语法树（Abstract Syntax Tree，即 AST）。通过 clang -E -Xclang -ast-dump main.c 命令获得相应的 AST，如图3所示：

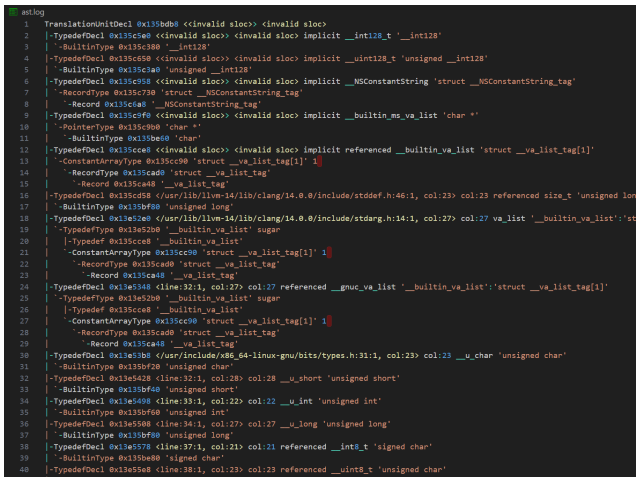


图 3: 语法分析结果

观察可知，他将语法结构加以识别，构成了一颗层次分明的语法树，对于后续的语义分析提供便利。

同时，为了检验其检验语法的功能，我们编写了一段错误程序，以此检测其功能。

错误代码

```
int main()
{
    int i;
    i*;
}
```

输入上述命令后，我们可以得到其语法分析的结果。

错误代码语法分析结果

```
test.c:4:8: error: expected expression
    i *;
      ^
1 error generated.
```

观察可以看到，在其语法树的前面，会有语法的错误提醒，从而验证语法分析检查代码语法的功能。

3. 语义分析和中间代码生成

语义分析是一个关键的步骤，负责理解代码的含义并检查其语法正确性以及语义是否合法，同时，深度分析并理解代码，确保其能正常执行。

利用 clang -S -emit-llvm main.c 命令生成 LLVM IR 中间代码。

LLVM IR 中间代码生成

```
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-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 [10 x i8] c"i\E5\B0\8F\E4\BA\8E1\0A\00", align 1
@.str.2 = private unnamed_addr constant [14 x i8] c"\E8\BE\93\E5\87\BA\E8\B6\85\E9\99\90\0A\00", align 1
@.str.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@.str.4 = private unnamed_addr constant [28 x i8] c"\E7\A8\8B\E5\BA\8F\E8\BF\90\E8\A1\8C\E6\97\B6\E9\97\B4\E4\B8\BA\EF\BC\9A%f\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i64, align 8
  %6 = alloca i64, align 8
  %7 = alloca double, align 8
  store i32 0, i32* %1, align 4
  %8 = call i32 (i8*, ...) @__isoc99_scanf(i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32* noundef %3)
  store i32 2, i32* %2, align 4
  store i32 1, i32* %4, align 4
  %9 = call i64 @clock() #3
  store i64 %9, i64* %5, align 8
  %10 = load i32, i32* %2, align 4
  %11 = icmp slt i32 %10, 1
  br i1 %11, label %12, label %14

12:                                               ; preds = %0
  %13 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([10 x i8], [10 x i8]* @.str.1, i64 0, i64 0))
  br label %14

14:                                               ; preds = %12, %0
  br label %15

15:                                               ; preds = %19, %14
  %16 = load i32, i32* %2, align 4
  %17 = load i32, i32* %3, align 4
  %18 = icmp sle i32 %16, %17
  br i1 %18, label %19, label %25

19:                                               ; preds = %15
  %20 = load i32, i32* %4, align 4
  %21 = load i32, i32* %2, align 4
```

```
  %22 = mul nsw i32 %20, %21
  store i32 %22, i32* %4, align 4
  %23 = load i32, i32* %2, align 4
  %24 = add nsw i32 %23, 1
  store i32 %24, i32* %2, align 4
  br label %15, !llvm.loop !6

25:                                               ; preds = %15
  %26 = load i32, i32* %4, align 4
  %27 = icmp sgt i32 %26, 10000
  br i1 %27, label %28, label %30

28:                                               ; preds = %25
  %29 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([14 x
       i8], [14 x i8]* @.str.2, i64 0, i64 0))
  br label %33

30:                                               ; preds = %25
  %31 = load i32, i32* %4, align 4
  %32 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
      i8], [4 x i8]* @.str.3, i64 0, i64 0), i32 noundef %31)
  br label %33

33:                                               ; preds = %30, %28
  %34 = call i64 @clock() #3
  store i64 %34, i64* %6, align 8
  %35 = load i64, i64* %6, align 8
  %36 = load i64, i64* %5, align 8
  %37 = sub nsw i64 %35, %36
  %38 = sitofp i64 %37 to double
  %39 = fdiv double %38, 1.000000e+06
  store double %39, double* %7, align 8
  %40 = load double, double* %7, align 8
  %41 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([28 x
       i8], [28 x i8]* @.str.4, i64 0, i64 0), double noundef %40)
  ret i32 0
}

declare i32 @__isoc99_scanf(i8* noundef, ...) #1

; Function Attrs: nounwind
declare i64 @clock() #2

declare i32 @printf(i8* noundef, ...) #1

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "
    min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-
    buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx
```

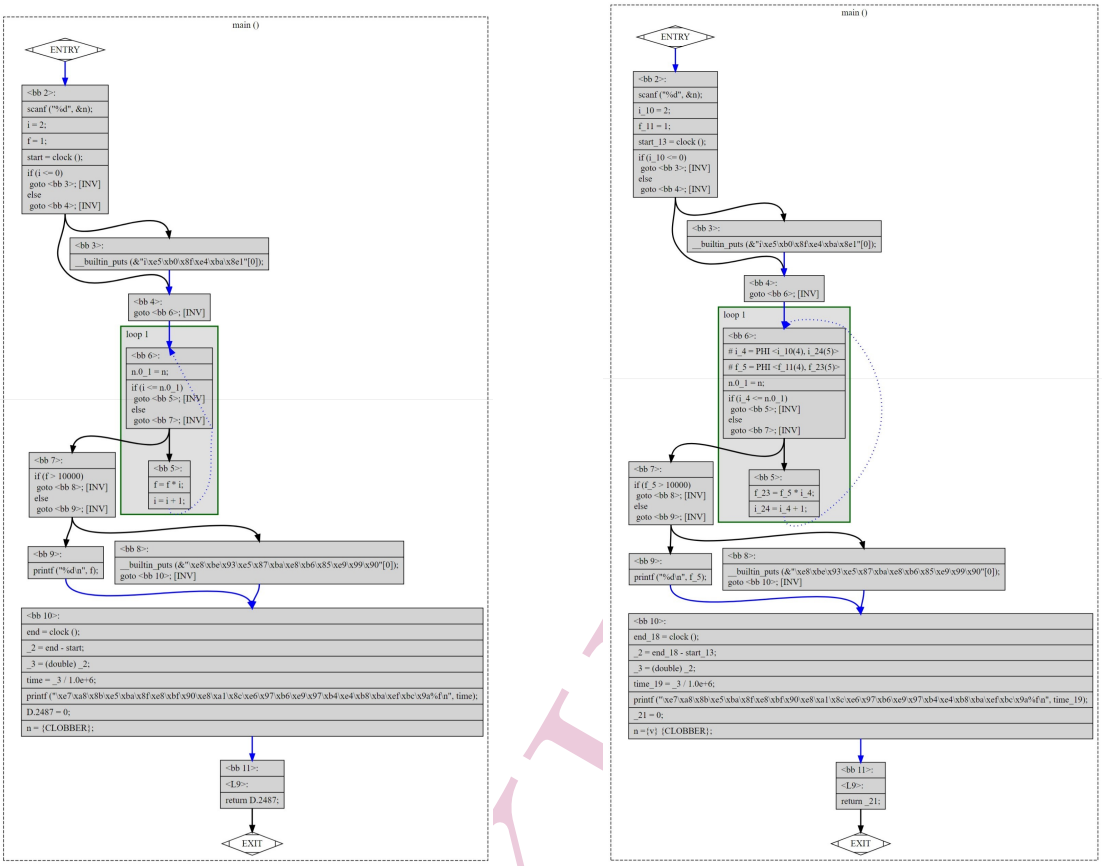
```
    ,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-
    protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8
    ,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #2 = { nounwind "frame-pointer"="all" "no-trapping-math"="true" "
    stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"=
    "+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #3 = { nounwind }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
!6 = distinct !{!6, !7}
!7 = !{!"llvm.loop.mustprogress"}
```

在 LLVM IR 程序的开始，给出了模块的 ID 并记录了源文件的程序名，接着记录了程序数据的字节顺序、类型长度以及程序运行环境等信息。接着是对出现在程序中所有的字符串按顺序定义为全局的常量（以 @ 开头），记录了每个字符串的长度和内容。

之后声明了返回类型为 i32 的 main 函数，#0 映射到一组函数属性，定义在文件的末尾。在函数内部，依次在栈帧中为函数中出现的所有变量分配了空间，按照源程序的内容初始化相应变量，调用相应的函数。对于每一个分支语句，如 if，while 等，会在可能需要跳转的地方生成标签，生成分支条件并控制跳转到相应的标签。在程序中部分指令后面会带有 nsw 的标记，这个标记指定该操作是不会溢出的，允许进行某些优化。函数的最后返回 0，结束调用。

在程序的最后，给出了所有调用的外部函数的声明和若干组函数属性，以及编译器的配置信息等。

利用 gcc -fdump-tree-all-graph main.c 命令可以得到中间代码生成的多阶段输出，此处借助 VS code 中的插件实现对 CFG 的可视化并加以分析。



(a) a-main.c.022t.fixup-cfg1.dot (b) a-main.c.027t.fixup-cfg2.dot

图 4: CFG 可视化

CFG 图将代码转换为控制流图，可以清晰的反映出代码段之间的跳转关系。

观察生成图片可以发现，控制流、变量名等发生变化，推测其可能为代码生成不同阶段中的不同值。

4. 代码优化

(1) 格式转换

借助 llvm-as main.ll -o main.bc 命令得到 LLVM IR 的二进制代码形式。

(2) 优化比较

借助 opt -S –O0 main.bc -o main-O0.ll 命令，实现对其 O0 优化。得到中间代码后继续处理，得到可执行文件，运行比较优化性能。

借助 opt -S –O1 main.bc -o main-O1.ll 命令，实现对其 O1 优化。得到中间代码后继续处理，得到可执行文件，运行比较优化性能。

借助 opt -S –O2 main.bc -o main-O2.ll 命令，实现对其 O2 优化。得到中间代码后继续处理，得到可执行文件，运行比较优化性能。

经过代码比较可知，O2 级优化与 O1 级优化代码相同，故而可知，O1 级优化已经优化完毕。运行三个可执行文件可知，O1 较 O0 有了一定程度优化，但 O2 级优化出现了不稳定情况。但是综合来看优化效果有限，推测可能是因为程序过于简单所导致。

```
lxmliu2002@lxmliu2002-Ubuntu:~/桌面/lab1$ ./main_O0
10
输出超限
程序运行时间为: 0.000032
lxmliu2002@lxmliu2002-Ubuntu:~/桌面/lab1$ ./main_O1
10
输出超限
程序运行时间为: 0.000016
lxmliu2002@lxmliu2002-Ubuntu:~/桌面/lab1$ ./main_O2
10
输出超限
程序运行时间为: 0.000109
```

图 5: 优化结果比较

5. 代码生成

以中间表示形式作为输入，将其映射到目标语言。

利用 llc main.ll -o main.S 命令生成目标代码；利用 gcc main.i -S -o main.S 命令生成 x86 格式目标代码；利用 arm-linux-gnueabihf-gcc main.i -S -o main.S 命令生成 arm 格式目标代码。由于篇幅有限，代码文件内容在此不做展示，源文件已放于附件之中。

(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。汇编器会将汇编程序中的伪指令翻译成等价的机器语言指令，将分支和数据传输指令中用到的标号都放入一个符号表中，通过查表可以生成对应指令的二进制表示。

分别使用 gcc main.S -c -o main.o 命令、arm-linux-gnueabihf-gcc main.S -o main.o 命令以及 llc main.bc -filetype=obj -o main.o 命令实现对 x86 格式、arm 格式以及 LLVM 格式文件进行汇编，得到 main.o 文件。

使用 objdump -d main_gcc.o 命令、arm-linux-gnueabihf-objdump -d main_arm.o 命令对上述文件进行反汇编。由于内容所限，此处仅对 x86 反汇编进行展示。

x86 反编译结果

```
main_gcc.o:     文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 30             sub    $0x30,%rsp
   c:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  13:   00 00
  15:   48 89 45 f8             mov    %rax,-0x8(%rbp)
  19:   31 c0                   xor    %eax,%eax
  1b:   48 8d 45 d4             lea    -0x2c(%rbp),%rax
  1f:   48 89 c6                mov    %rax,%rsi
  22:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # 29 <main+0x29>
  29:   48 89 c7                mov    %rax,%rdi
```

```
2c:   b8 00 00 00 00          mov    $0x0,%eax
31:   e8 00 00 00 00          call   36 <main+0x36>
36:   c7 45 d8 02 00 00 00    movl   $0x2,-0x28(%rbp)
3d:   c7 45 dc 01 00 00 00    movl   $0x1,-0x24(%rbp)
44:   e8 00 00 00 00          call   49 <main+0x49>
49:   48 89 45 e0             mov    %rax,-0x20(%rbp)
4d:   83 7d d8 00             cmpl   $0x0,-0x28(%rbp)
51:   7f 1f                   jg     72 <main+0x72>
53:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # 5a <main+0x5a>
5a:   48 89 c7                mov    %rax,%rdi
5d:   e8 00 00 00 00          call   62 <main+0x62>
62:   eb 0e                   jmp    72 <main+0x72>
64:   8b 45 dc                mov    -0x24(%rbp),%eax
67:   0f af 45 d8             imul   -0x28(%rbp),%eax
6b:   89 45 dc                mov    %eax,-0x24(%rbp)
6e:   83 45 d8 01             addl   $0x1,-0x28(%rbp)
72:   8b 45 d4                mov    -0x2c(%rbp),%eax
75:   39 45 d8                cmp    %eax,-0x28(%rbp)
78:   7e ea                   jle    64 <main+0x64>
7a:   81 7d dc 10 27 00 00    cmpl   $0x2710,-0x24(%rbp)
81:   7e 11                   jle    94 <main+0x94>
83:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # 8a <main+0x8a>
8a:   48 89 c7                mov    %rax,%rdi
8d:   e8 00 00 00 00          call   92 <main+0x92>
92:   eb 19                   jmp    ad <main+0xad>
94:   8b 45 dc                mov    -0x24(%rbp),%eax
97:   89 c6                   mov    %eax,%esi
99:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # a0 <main+0xa0>
a0:   48 89 c7                mov    %rax,%rdi
a3:   b8 00 00 00 00          mov    $0x0,%eax
a8:   e8 00 00 00 00          call   ad <main+0xad>
ad:   e8 00 00 00 00          call   b2 <main+0xb2>
b2:   48 89 45 e8             mov    %rax,-0x18(%rbp)
b6:   48 8b 45 e8             mov    -0x18(%rbp),%rax
ba:   48 2b 45 e0             sub    -0x20(%rbp),%rax
be:   66 0f ef c0             pxor   %xmm0,%xmm0
c2:   f2 48 0f 2a c0          cvtsi2sd %rax,%xmm0
c7:   f2 0f 10 0d 00 00 00    movsd  0x0(%rip),%xmm1        # cf <main+0xcf
    >
ce:   00
cf:   f2 0f 5e c1             divsd  %xmm1,%xmm0
d3:   f2 0f 11 45 f0          movsd  %xmm0,-0x10(%rbp)
d8:   48 8b 45 f0             mov    -0x10(%rbp),%rax
dc:   66 48 0f 6e c0          movq   %rax,%xmm0
e1:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # e8 <main+0xe8>
e8:   48 89 c7                mov    %rax,%rdi
eb:   b8 01 00 00 00          mov    $0x1,%eax
f0:   e8 00 00 00 00          call   f5 <main+0xf5>
```

```
 f5:   b8 00 00 00 00          mov    $0x0,%eax
 fa:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
 fe:   64 48 2b 14 25 28 00    sub    %fs:0x28,%rdx
105:   00 00
107:   74 05                   je     10e <main+0x10e>
109:   e8 00 00 00 00          call   10e <main+0x10e>
10e:   c9                      leave
10f:   c3                      ret
```

将反编译的结果与汇编程序对比，可以发现这里所有涉及到变量、分支标签、调用外部函数对应位置的二进制代码全部为 0，反编译结果中的 call 指令全部指向了它的后一条指令，因为实际的地址是由下一条指令的地址加上 call 后面的地址之和，这里 call 后面的地址为 0，故它们目前均指向下一条指令。

(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。进而链接器对该机器代码进行执行生成可执行文件。

通过 gcc main_gcc.o -o main_gcc 命令、gcc main_arm.o -o main_arm 命令和 clang main_llvm.o -o main_llvm 命令生成不同中间代码版本的 main 可执行文件。

我们可以修改一下参数实现对链接结果的调整。

- -o <output>: 指定输出文件的名称，例如-o main 将生成一个名为 main 的可执行文件。
- -L<path>: 指定链接器搜索库文件的路径，例如-L/usr/lib 将在 /usr/lib 目录下搜索库文件。
- -l<library>: 指定链接器需要链接的库文件，例如-lm 将链接数学库。
- -I<path>: 指定编译器搜索头文件的路径，例如-I/usr/include 将在 /usr/include 目录下搜索头文件。
- -static: 静态链接，将所有的库文件都静态链接到可执行文件中，而不是动态链接。
- -Wl,<option>: 将 <option> 传递给链接器，例如-Wl,-rpath,/usr/local/lib 将设置运行时库搜索路径。
- -g: 生成调试信息，方便调试程序。
- -Wall: 启用所有警告信息。
- -O<level>: 指定优化级别，例如-O2 进行中级优化。
- -std=<standard>: 指定使用的 C 语言标准，例如-std=c11 指定使用 C11 标准。

使用 objdump -d main_gcc 命令、arm-linux-gnueabihf-objdump -d main_arm 命令对上述文件进行反汇编。由于内容所限，此处仅对 x86 反汇编进行展示。

x86 反编译结果

```
main_gcc:     文件格式 elf64-x86-64

```

```
Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    $0x8,%rsp
    1008:       48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax        # 3
        fe8 <__gmon_start__@Base>
    100f:       48 85 c0                test   %rax,%rax
    1012:       74 02                   je     1016 <_init+0x16>
    1014:       ff d0                   call   *%rax
    1016:       48 83 c4 08             add    $0x8,%rsp
    101a:       c3                      ret

Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:       ff 35 7a 2f 00 00       push   0x2f7a(%rip)        # 3fa0 <
        _GLOBAL_OFFSET_TABLE_+0x8>
    1026:       f2 ff 25 7b 2f 00 00    bnd jmp *0x2f7b(%rip)        # 3fa8 <
        _GLOBAL_OFFSET_TABLE_+0x10>
    102d:       0f 1f 00                nopl   (%rax)
    1030:       f3 0f 1e fa             endbr64
    1034:       68 00 00 00 00          push   $0x0
    1039:       f2 e9 e1 ff ff ff       bnd jmp 1020 <_init+0x20>
    103f:       90                      nop
    1040:       f3 0f 1e fa             endbr64
    1044:       68 01 00 00 00          push   $0x1
    1049:       f2 e9 d1 ff ff ff       bnd jmp 1020 <_init+0x20>
    104f:       90                      nop
    1050:       f3 0f 1e fa             endbr64
    1054:       68 02 00 00 00          push   $0x2
    1059:       f2 e9 c1 ff ff ff       bnd jmp 1020 <_init+0x20>
    105f:       90                      nop
    1060:       f3 0f 1e fa             endbr64
    1064:       68 03 00 00 00          push   $0x3
    1069:       f2 e9 b1 ff ff ff       bnd jmp 1020 <_init+0x20>
    106f:       90                      nop
    1070:       f3 0f 1e fa             endbr64
    1074:       68 04 00 00 00          push   $0x4
    1079:       f2 e9 a1 ff ff ff       bnd jmp 1020 <_init+0x20>
    107f:       90                      nop

Disassembly of section .plt.got:

0000000000001080 <__cxa_finalize@plt>:
    1080:       f3 0f 1e fa             endbr64
    1084:       f2 ff 25 6d 2f 00 00    bnd jmp *0x2f6d(%rip)        # 3ff8 <
```

```
        __cxa_finalize@GLIBC_2.2.5>
    108b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

Disassembly of section .plt.sec:

0000000000001090 <puts@plt>:
    1090:       f3 0f 1e fa             endbr64
    1094:       f2 ff 25 15 2f 00 00    bnd jmp *0x2f15(%rip)        # 3fb0 <
        puts@GLIBC_2.2.5>
    109b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

00000000000010a0 <clock@plt>:
    10a0:       f3 0f 1e fa             endbr64
    10a4:       f2 ff 25 0d 2f 00 00    bnd jmp *0x2f0d(%rip)        # 3fb8 <
        clock@GLIBC_2.2.5>
    10ab:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

00000000000010b0 <__stack_chk_fail@plt>:
    10b0:       f3 0f 1e fa             endbr64
    10b4:       f2 ff 25 05 2f 00 00    bnd jmp *0x2f05(%rip)        # 3fc0 <
        __stack_chk_fail@GLIBC_2.4>
    10bb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

00000000000010c0 <printf@plt>:
    10c0:       f3 0f 1e fa             endbr64
    10c4:       f2 ff 25 fd 2e 00 00    bnd jmp *0x2efd(%rip)        # 3fc8 <
        printf@GLIBC_2.2.5>
    10cb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

00000000000010d0 <__isoc99_scanf@plt>:
    10d0:       f3 0f 1e fa             endbr64
    10d4:       f2 ff 25 f5 2e 00 00    bnd jmp *0x2ef5(%rip)        # 3fd0 <
        __isoc99_scanf@GLIBC_2.7>
    10db:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

Disassembly of section .text:

00000000000010e0 <_start>:
    10e0:       f3 0f 1e fa             endbr64
    10e4:       31 ed                   xor    %ebp,%ebp
    10e6:       49 89 d1                mov    %rdx,%r9
    10e9:       5e                      pop    %rsi
    10ea:       48 89 e2                mov    %rsp,%rdx
    10ed:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    10f1:       50                      push   %rax
    10f2:       54                      push   %rsp
    10f3:       45 31 c0                xor    %r8d,%r8d
    10f6:       31 c9                   xor    %ecx,%ecx
```

```
90      10f8:       48 8d 3d ca 00 00 00     lea    0xca(%rip),%rdi        # 11c9
            <main>
91      10ff:       ff 15 d3 2e 00 00        call   *0x2ed3(%rip)        # 3fd8 <
            __libc_start_main@GLIBC_2.34>
92      1105:       f4                       hlt
93      1106:       66 2e 0f 1f 84 00 00     cs nopw 0x0(%rax,%rax,1)
94      110d:       00 00 00
95
96  0000000000001110 <deregister_tm_clones>:
97      1110:       48 8d 3d f9 2e 00 00     lea    0x2ef9(%rip),%rdi        #
            4010 <__TMC_END__>
98      1117:       48 8d 05 f2 2e 00 00     lea    0x2ef2(%rip),%rax        #
            4010 <__TMC_END__>
99      111e:       48 39 f8                 cmp    %rdi,%rax
100     1121:       74 15                    je     1138 <deregister_tm_clones+0
            x28>
101     1123:       48 8b 05 b6 2e 00 00     mov    0x2eb6(%rip),%rax        # 3
            fe0 <_ITM_deregisterTMCloneTable@Base>
102     112a:       48 85 c0                 test   %rax,%rax
103     112d:       74 09                    je     1138 <deregister_tm_clones+0
            x28>
104     112f:       ff e0                    jmp    *%rax
105     1131:       0f 1f 80 00 00 00 00     nopl   0x0(%rax)
106     1138:       c3                       ret
107     1139:       0f 1f 80 00 00 00 00     nopl   0x0(%rax)
108
109 0000000000001140 <register_tm_clones>:
110     1140:       48 8d 3d c9 2e 00 00     lea    0x2ec9(%rip),%rdi        #
            4010 <__TMC_END__>
111     1147:       48 8d 35 c2 2e 00 00     lea    0x2ec2(%rip),%rsi        #
            4010 <__TMC_END__>
112     114e:       48 29 fe                 sub    %rdi,%rsi
113     1151:       48 89 f0                 mov    %rsi,%rax
114     1154:       48 c1 ee 3f              shr    $0x3f,%rsi
115     1158:       48 c1 f8 03              sar    $0x3,%rax
116     115c:       48 01 c6                 add    %rax,%rsi
117     115f:       48 d1 fe                 sar    %rsi
118     1162:       74 14                    je     1178 <register_tm_clones+0x38>
119     1164:       48 8b 05 85 2e 00 00     mov    0x2e85(%rip),%rax        # 3
            ff0 <_ITM_registerTMCloneTable@Base>
120     116b:       48 85 c0                 test   %rax,%rax
121     116e:       74 08                    je     1178 <register_tm_clones+0x38>
122     1170:       ff e0                    jmp    *%rax
123     1172:       66 0f 1f 44 00 00        nopw   0x0(%rax,%rax,1)
124     1178:       c3                       ret
125     1179:       0f 1f 80 00 00 00 00     nopl   0x0(%rax)
126
127 0000000000001180 <__do_global_dtors_aux>:
```

```
    1180:       f3 0f 1e fa             endbr64
    1184:       80 3d 85 2e 00 00 00    cmpb   $0x0,0x2e85(%rip)        # 4010 <__TMC_END__>
    118b:       75 2b                   jne    11b8 <__do_global_dtors_aux+0x38>
    118d:       55                      push   %rbp
    118e:       48 83 3d 62 2e 00 00    cmpq   $0x0,0x2e62(%rip)        # 3ff8 <__cxa_finalize@GLIBC_2.2.5>
    1195:       00
    1196:       48 89 e5                mov    %rsp,%rbp
    1199:       74 0c                   je     11a7 <__do_global_dtors_aux+0x27>
    119b:       48 8b 3d 66 2e 00 00    mov    0x2e66(%rip),%rdi        # 4008 <__dso_handle>
    11a2:       e8 d9 fe ff ff          call   1080 <__cxa_finalize@plt>
    11a7:       e8 64 ff ff ff          call   1110 <deregister_tm_clones>
    11ac:       c6 05 5d 2e 00 00 01    movb   $0x1,0x2e5d(%rip)        # 4010 <__TMC_END__>
    11b3:       5d                      pop    %rbp
    11b4:       c3                      ret
    11b5:       0f 1f 00                nopl   (%rax)
    11b8:       c3                      ret
    11b9:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

00000000000011c0 <frame_dummy>:
    11c0:       f3 0f 1e fa             endbr64
    11c4:       e9 77 ff ff ff          jmp    1140 <register_tm_clones>

00000000000011c9 <main>:
    11c9:       f3 0f 1e fa             endbr64
    11cd:       55                      push   %rbp
    11ce:       48 89 e5                mov    %rsp,%rbp
    11d1:       48 83 ec 30             sub    $0x30,%rsp
    11d5:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    11dc:       00 00
    11de:       48 89 45 f8             mov    %rax,-0x8(%rbp)
    11e2:       31 c0                   xor    %eax,%eax
    11e4:       48 8d 45 d4             lea    -0x2c(%rbp),%rax
    11e8:       48 89 c6                mov    %rax,%rsi
    11eb:       48 8d 05 16 0e 00 00    lea    0xe16(%rip),%rax        # 2008 <_IO_stdin_used+0x8>
    11f2:       48 89 c7                mov    %rax,%rdi
    11f5:       b8 00 00 00 00          mov    $0x0,%eax
    11fa:       e8 d1 fe ff ff          call   10d0 <__isoc99_scanf@plt>
    11ff:       c7 45 d8 02 00 00 00    movl   $0x2,-0x28(%rbp)
    1206:       c7 45 dc 01 00 00 00    movl   $0x1,-0x24(%rbp)
    120d:       e8 8e fe ff ff          call   10a0 <clock@plt>
    1212:       48 89 45 e0             mov    %rax,-0x20(%rbp)
```

```
1216:   83 7d d8 00          cmpl   $0x0,-0x28(%rbp)
121a:   7f 1f                jg     123b <main+0x72>
121c:   48 8d 05 e8 0d 00 00 lea    0xde8(%rip),%rax        # 200b
    <_IO_stdin_used+0xb>
1223:   48 89 c7             mov    %rax,%rdi
1226:   e8 65 fe ff ff       call   1090 <puts@plt>
122b:   eb 0e                jmp    123b <main+0x72>
122d:   8b 45 dc             mov    -0x24(%rbp),%eax
1230:   0f af 45 d8          imul   -0x28(%rbp),%eax
1234:   89 45 dc             mov    %eax,-0x24(%rbp)
1237:   83 45 d8 01          addl   $0x1,-0x28(%rbp)
123b:   8b 45 d4             mov    -0x2c(%rbp),%eax
123e:   39 45 d8             cmp    %eax,-0x28(%rbp)
1241:   7e ea                jle    122d <main+0x64>
1243:   81 7d dc 10 27 00 00 cmpl   $0x2710,-0x24(%rbp)
124a:   7e 11                jle    125d <main+0x94>
124c:   48 8d 05 c1 0d 00 00 lea    0xdc1(%rip),%rax        # 2014
    <_IO_stdin_used+0x14>
1253:   48 89 c7             mov    %rax,%rdi
1256:   e8 35 fe ff ff       call   1090 <puts@plt>
125b:   eb 19                jmp    1276 <main+0xad>
125d:   8b 45 dc             mov    -0x24(%rbp),%eax
1260:   89 c6                mov    %eax,%esi
1262:   48 8d 05 b8 0d 00 00 lea    0xdb8(%rip),%rax        # 2021
    <_IO_stdin_used+0x21>
1269:   48 89 c7             mov    %rax,%rdi
126c:   b8 00 00 00 00       mov    $0x0,%eax
1271:   e8 4a fe ff ff       call   10c0 <printf@plt>
1276:   e8 25 fe ff ff       call   10a0 <clock@plt>
127b:   48 89 45 e8          mov    %rax,-0x18(%rbp)
127f:   48 8b 45 e8          mov    -0x18(%rbp),%rax
1283:   48 2b 45 e0          sub    -0x20(%rbp),%rax
1287:   66 0f ef c0          pxor   %xmm0,%xmm0
128b:   f2 48 0f 2a c0       cvtsi2sd %rax,%xmm0
1290:   f2 0f 10 0d b0 0d 00 movsd  0xdb0(%rip),%xmm1       #
    2048 <_IO_stdin_used+0x48>
1297:   00
1298:   f2 0f 5e c1          divsd  %xmm1,%xmm0
129c:   f2 0f 11 45 f0       movsd  %xmm0,-0x10(%rbp)
12a1:   48 8b 45 f0          mov    -0x10(%rbp),%rax
12a5:   66 48 0f 6e c0       movq   %rax,%xmm0
12aa:   48 8d 05 74 0d 00 00 lea    0xd74(%rip),%rax        # 2025
    <_IO_stdin_used+0x25>
12b1:   48 89 c7             mov    %rax,%rdi
12b4:   b8 01 00 00 00       mov    $0x1,%eax
12b9:   e8 02 fe ff ff       call   10c0 <printf@plt>
12be:   b8 00 00 00 00       mov    $0x0,%eax
12c3:   48 8b 55 f8          mov    -0x8(%rbp),%rdx
```

```
    12c7:       64 48 2b 14 25 28 00    sub    %fs:0x28,%rdx
    12ce:       00 00
    12d0:       74 05                   je     12d7 <main+0x10e>
    12d2:       e8 d9 fd ff ff          call   10b0 <__stack_chk_fail@plt>
    12d7:       c9                      leave
    12d8:       c3                      ret

Disassembly of section .fini:

00000000000012dc <_fini>:
    12dc:       f3 0f 1e fa             endbr64
    12e0:       48 83 ec 08             sub    $0x8,%rsp
    12e4:       48 83 c4 08             add    $0x8,%rsp
    12e8:       c3                      ret
```

将其与上面的反编译代码进行比较，可以发现填补了之前一些地址的空白，还有其他库文件的反编译代码，从而验证了链接器的功能。

（五） 执行

在终端中输入./main 即可执行该程序。

```
lxmliu2002@lxmliu2002-Ubuntu:~/桌面/lab1$ ./main
5
120
程序运行时间为: 0.000039
```

图 6: main 程序运行结果

（六） LLVM IR 编程

经过查询可知，LLVM IR 的语法规则如下：

- 模块（Moudle）

```
; ModuleID = 'example'
source_filename = "example"
@global_var = global i32 10
define i32 @main() {
  ; function body
  ret i32 0
}
```

- 全局变量（Global Variable）：以 @ 开头

```
@global_var = global i32 10
@global_array = global [10 x i32] zeroinitializer
```

- 函数（Function）：以 define 关键字开头

```
define i32 @add(i32 %a, i32 %b) {
  ; function body
  %sum = add i32 %a, %b
  ret i32 %sum
}
```

- 基本块（Bassic Block）：以 Label 开头，以终止指令 Terminator Instruction 结束

```
entry:
  ; instructions
  %sum = add i32 %a, %b
  br label %exit
exit:
  ; instructions
  ret i32 %sum
```

- 指令（Instruction）

```
%sum = add i32 %a, %b
%result = mul i32 %sum, 2
%cmp = icmp eq i32 %result, 10
br i1 %cmp, label %true, label %false
```

- 标签（Label）：以% 开头

```
entry:
  ; instructions
  br label %exit
```

基于以上语法规则，我们小组尝试编写的 LLVM IR 程序如下：

LLVM IR 中间代码编写

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
    :16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [13 x i8] c"Hello World\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 1, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = icmp ne i32 %2, 0
  br i1 %3, label %4, label %6

```

```
4:
  %5 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([13 x
      i8], [13 x i8]* @.str, i64 0, i64 0))
  br label %6

6:
  ret i32 0
}

declare i32 @printf(i8* noundef, ...) #1

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "
    min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-
    buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx
    ,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-
    protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8
    ,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
```

编译成可执行文件后运行结果如下:

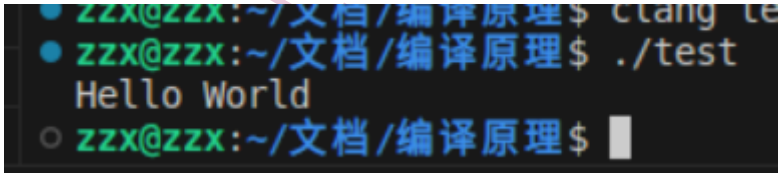


图 7: LLVM IR 程序运行结果

由上面运行结果可以看到，编写的 LLVM IR 程序的正确性得到了验证。

四、 总结

通过本次实验，我们依次完成了预处理、编译、汇编、链接、加载等过程，也对 LLVM IR 中间过程进行分析和编写，使用 gcc 生成控制流图，熟悉了每个中间步骤实现的操作以及产生的文件。也对不同优化情况下的代码对应的汇编结果进行比较，了解了基本的代码优化方向。同时也对 makefile 文件有了一定的了解，可以一定程度上免去重复工作带来的效率低下。对如何构建一个编译器有了初步的认识，这为我们后面实现一个编译器有很大的帮助。