南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理预习实验

了解你的编译器 & LLVM IR 编程 & 汇编编程（小组合作）

姜宇

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2024 年 9 月 23 日

摘要

关键字：Parallel

目录

一、 小组分工

- **申祖铭：** 负责第 3 小题 Sysy 程序设计以及汇编编程。
- **姜宇：** 负责第 2 小题以及 LLVM IR 编程。

二、 了解 GCC 编译器

GCC（GNU Compiler Collection）是一个广泛使用的开源编译器系统，它支持多种编程语言，包括 C、C++、Fortran 等。GCC 的工作过程可以分为几个主要阶段：预处理、编译、汇编和链接，如图1所示。

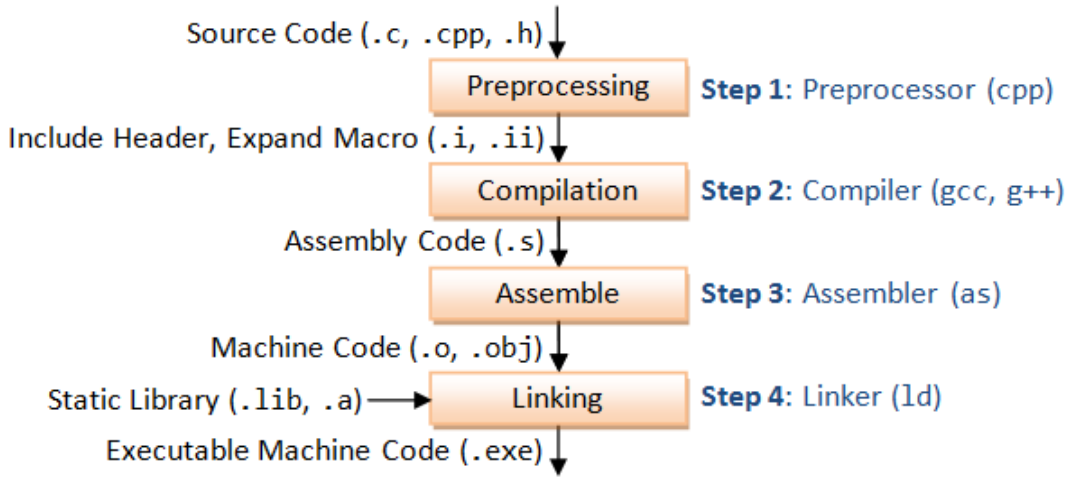

图 1: GCC Compilation Process

这里我所使用实验平台的信息如表二所示，为了解释各个环节的处理过程，这里使用以下求阶乘的代码为例，一步一步探究 GCC 编译器各个工作过程对代码进行了怎么样的处理：

```c
#include <stdio.h>
int main() {
    int num;
    unsigned long long factorial = 1; // 用于存储阶乘结果
    scanf("%d", &num);
    for (int i = 1; i <= num; i++) {
        factorial *= i; // 计算阶乘
    }
    printf("%llu\n", factorial);
    return 0;
}
```

项目	**详细信息**
目标平台	x86_64-linux-gnu
线程模型	posix
支持的 LTO 压缩算法	zlib, zstd
操作系统内核版本	6.8.0-41-generic
GCC 版本	13.2.0 (Ubuntu 13.2.0-23ubuntu4)
GCC 配置选项	–with-pkgversion='Ubuntu 13.2.0-23ubuntu4' –with-bugurl=file:///usr/share/doc/gcc-13/README.Bugs –enable-languages=c,ada,c++,go,d,fortran,objc,obj-c++,m2 –prefix=/usr –with-gcc-major-version-only –program-suffix=-13 –program-prefix=x86_64-linux-gnu- –enable-shared –enable-linker-build-id –libexecdir=/usr/libexec –without-included-gettext –enable-threads=posix –libdir=/usr/lib –enable-nls –enable-clocale=gnu –enable-libstdcxx-debug –enable-libstdcxx-time=yes –with-default-libstdcxx-abi=new –enable-libstdcxx-backtrace –enable-gnu-unique-object –disable-vtable-verify –enable-plugin –enable-default-pie –with-system-zlib –enable-libphobos-checking=release –enable-multiarch –enable-cet –with-arch-32=i686 –with-abi=m64 –enable-multilib –with-tune=generic –enable-offload-targets=nvptx-none,amdgcn-amdhsa –enable-offload-defaulted –without-cuda-driver –enable-checking=release

项目	详细信息
Linux 内核详细版本信息	Linux version 6.8.0-41-generic (buildd@lcy02-amd64-100) (x86_64-linux-gnu-gcc-13 (Ubuntu 13.2.0-23ubuntu4) 13.2.0, GNU ld (GNU Binutils for Ubuntu) 2.42) #41-Ubuntu SMP PREEMPT_DYNAMIC Fri Aug 2 20:41:06 UTC 2024
构建目标	x86_64-linux-gnu

(一) 预处理器（Preprocessor）

预处理阶段是对以“#”开头的代码进行处理以及一些格式上的的处理，主要有以下内容:

- **宏替换:** 删除所有 #define 指令，用宏的定义来替换代码中的宏调用。
- **文件包含:** 处理 #include 预编译命令，将被包含文件插入到该预编译指令的位置。
- **条件编译:** 处理所有预编译指令，如 #if、#ifdef、#ifndef、#else、#elif、#endif 等，决定哪些代码片段需要编译。
- **添加行号:** 添加源代码中的行号和文件信息，以便在编译时产生时能准确定位错误或警告的位置。
- **删除注释:** 从源代码中删除注释，以减少干扰。
- **保留指令:** 保留 #pragma 编译器指令，后续编译过程中使用。

以下是预处理之后的文件内容，由于加入了包含的 <stdio.h> 文件，文件大小个共有 824 行，中间部分进行了省略:

```
# 0 "main.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "main.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
```

…

```
int main() {
    int num;
    unsigned long long factorial = 1;
    scanf("%d", &num);
    for (int i = 1; i <= num; i++) {
        factorial *= i;
    }
    printf("%llu\n", factorial);
    return 0;
}
```

(二) 编译器（Compiler）

编译阶段将预处理器生成的纯文本代码转换为机器无关的中间表示（IR），也就是汇编代码，通过一系列的词法分析、语法分析、语义分析以及进一步优化，生成目标代码，正如老师所将这里对代码进行的处理被称为前端接口。编译过程可以进一步分为几个子阶段：

1. 词法分析

将源代码分解成一系列的词法单元（tokens），例如关键词、标识符、操作符等。该程序的词法分析结果如下，我们可以看到主要包含以下信息：

```
typedef 'typedef'        [StartOfLine]  Loc=</usr/lib/llvm-18/lib/clang/18/
    include/__stddef_size_t.h:18:1>
long 'long'      [LeadingSpace] Loc=</usr/lib/llvm-18/lib/clang/18/include/
    __stddef_size_t.h:18:9 <Spelling=<built-in>:107:23>>
unsigned 'unsigned'      [LeadingSpace] Loc=</usr/lib/llvm-18/lib/clang/18/
    include/__stddef_size_t.h:18:9 <Spelling=<built-in>:107:28>>
```

…

```
identifier 'factorial'   [LeadingSpace] Loc=<main.c:9:22>
r_paren ')'              Loc=<main.c:9:31>
semi ';'                 Loc=<main.c:9:32>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<main.c:10:5>
numeric_constant '0'     [LeadingSpace] Loc=<main.c:10:12>
semi ';'                 Loc=<main.c:10:13>
r_brace '}'      [StartOfLine]  Loc=<main.c:11:1>
eof ''           Loc=<main.c:11:2>
```

- **Token 类型和文本表示：** 每一行包含一个 token 的类型（如 typedef, long, unsigned, int, identifier 等）以及其对应的文本表示，比如：'typedef' 表示类型为 typedef 的 token，它的文本表示为 typedef、'long' 表示类型为 long 的 token，文本表示为 long 等。
- **空格信息：** token 前的空格信息标识了 token 在源文件中的位置关系。[StartOfLine] 表示这个 token 位于行首，[LeadingSpace] 表示这个 token 前面有空格。
- **位置信息（Loc）：** 每个 token 都有一个位置信息，用于标明它在源文件中的位置。格式是文件路径: 行号: 列号。例如 Loc=</usr/include/x86_64-linux-gnu/bits/types.h:18:1> 表示这个 token 位于/usr/include/x86_64-linux-gnu/bits/types.h 文件的第 18 行，第 1 列。
- **typedef 声明：** 通过分析这些 tokens, 可以重建 typedef 声明的结构。例如 typedef unsigned int ___uid_t; 会被分解为一系列 tokens：

```
typedef 'typedef'
unsigned 'unsigned'
int 'int'
identifier '__uid_t'
semi ';'
```

- **文件和行的引用：** 这些 tokens 出现在多个 C 语言标准头文件中，如 ___stddef_size_t.h, ___stdarg_____gnuc_va_list.h 等。这些文件是 C 标准库的一部分，定义了基本数据类型和常用数据结构。

2. 语法分析

根据语法规则，将词法单元构建成抽象语法树（Abstract Syntax Tree，即 AST），类似的抽象语法树如图2所示。

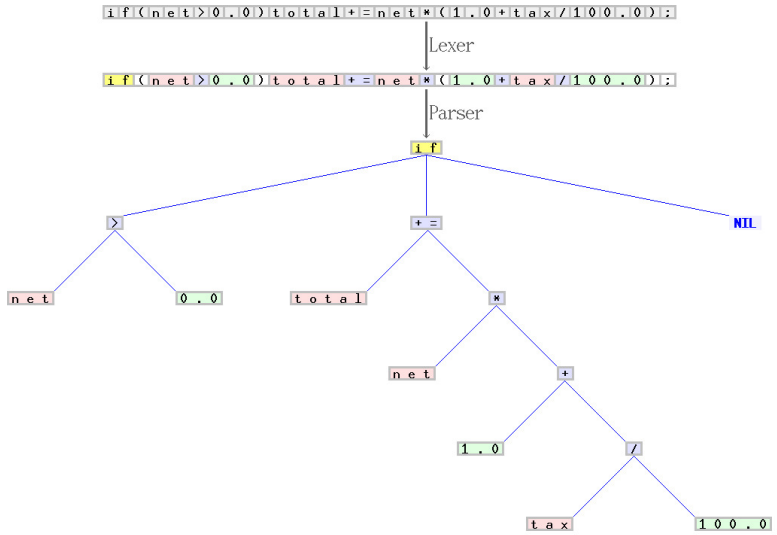

图 2: GCC Compilation Process

这里通过 Clang 编译器前端生成的AST 结构文件。可以看到其有以下结构:

- **根节点 TranslationUnitDecl：**这是整个翻译单元的顶级节点，就在文件的第一行。

```
TranslationUnitDecl 0x6038b4542158 <<invalid sloc>> <invalid sloc>
```

- **类型定义 TypedefDecl：**有很多节点，比如：___int128_t 是 ___int128 的别名，size_t 被定义为 unsigned long 等，这些类型定义通常是隐式的（implicit），用于为基本数据类型或结构定义更具语义的名称。

```
|-TypedefDecl 0x6038b4542988 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x6038b4542720 '__int128'
```

- **内建类型 BuiltinType 和指针类型 PointerType：**在类型定义中，我们看到很多 BuiltinType，例如 char、int、long 和 unsigned long，这些是 C 语言中的基本数据类型。PointerType 表示指向某种类型的指针，例如，char * 是一个指向 char 的指针类型。

```
|-TypedefDecl 0x6038b4542da8 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char
    *'
| `-PointerType 0x6038b4542d60 'char *'
|   `-BuiltinType 0x6038b4542200 'char'
```

- **结构定义 RecordDecl：**用于表示结构体，比如文件中存在 struct _G_fpos_t，这是一个定义在 /usr/include/x86_64-linux-gnu/bits/types/___fpos_t.h 中的结构体，它包含两个字段 ___pos 和 ___state。再比如 _IO_FILE，这个结构体中包含多个字段：_flags、_IO_read_ptr、_IO_write_ptr 等。

```
|-RecordDecl 0x6038b45c46d8 </usr/include/x86_64-linux-gnu/bits/types/__fpos_t.h:10:9, line:14:1>
    line:10:16 struct _G_fpos_t definition
```

```
| |-FieldDecl 0x6038b45c47f0 <line:12:3, col:11> col:11 __pos '__off_t':'long'
| `-FieldDecl 0x6038b45c48b0 <line:13:3, col:15> col:15 __state '__mbstate_t'
```

- **函数声明 FunctionDecl：**这表示一个函数，每个 FunctionDecl 节点都包含了函数的参数（ParmVarDecl），返回类型，以及一些属性。比如 printf 函数被声明为：

```
|-FunctionDecl 0x6038b45b3a80 <line:363:12> col:12 implicit used printf 'int (const char *, ...)'
    extern
```

这表示其为可变参数的函数。

- **变量声明 VarDecl：**表示声明变量，文件中存在以下结构：

```
|-VarDecl 0x6038b45ce938 <line:149:1, col:14> col:14 stdin 'FILE *' extern
|-VarDecl 0x6038b45cea00 <line:150:1, col:14> col:14 stdout 'FILE *' extern
|-VarDecl 0x6038b45cea80 <line:151:1, col:14> col:14 stderr 'FILE *' extern
```

stdin、stdout 和 stderr 被声明为外部的（extern）指向 FILE 结构的指针。

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，进行类型检查等。

4. 中间代码生成

这里使用 O2 优化来生存中间代码的多阶段输出，由于中间文件过多，文件含义的解释放在文末，如表七。

5. 代码优化

观察各个 passes 优化编译后的 LLVM 中间表示（LLVM IR，Intermediate Representation），我们可以看到中间表示简洁了很多，没有像预处理的时候文件大小这么臃肿。

```
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-
    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 [6 x i8] c"%llu\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 i64, align 8
  %4 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i64 1, ptr %3, align 8
  %5 = call i32 (ptr, ...) @__isoc99_scanf(ptr noundef @.str, ptr noundef %2)
  store i32 1, ptr %4, align 4
```

```
  br label %6

6:                                                ; preds = %15, %0
  %7 = load i32, ptr %4, align 4
  %8 = load i32, ptr %2, align 4
  %9 = icmp sle i32 %7, %8
  br i1 %9, label %10, label %18

10:                                               ; preds = %6
  %11 = load i32, ptr %4, align 4
  %12 = sext i32 %11 to i64
  %13 = load i64, ptr %3, align 8
  %14 = mul i64 %13, %12
  store i64 %14, ptr %3, align 8
  br label %15

15:                                               ; preds = %10
  %16 = load i32, ptr %4, align 4
  %17 = add nsw i32 %16, 1
  store i32 %17, ptr %4, align 4
  br label %6, !llvm.loop !6

18:                                               ; preds = %6
  %19 = load i64, ptr %3, align 8
  %20 = call i32 (ptr, ...) @printf(ptr noundef @.str.1, i64 noundef %19)
  ret i32 0
}

declare i32 @__isoc99_scanf(ptr noundef, ...) #1

declare i32 @printf(ptr 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"="+cmov,+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"="+cmov
    ,+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 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{i32 7, !"frame-pointer", i32 2}
```

```
!5 = !{!"Ubuntu clang version 18.1.3 (1ubuntu1)"}
!6 = distinct !{!6, !7}
!7 = !{!"llvm.loop.mustprogress"}
```

这里先显示没有优化的 IR，由于 passes 过多这里只分析以下 passes:

- **SimplifyCFGPass**

```
; *** IR Dump After Annotation2MetadataPass on [module] ***
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-
    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 [6 x i8] c"%llu\0A\00", align 1

; Function Attrs: nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i64, align 8
  %4 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  call void @llvm.lifetime.start.p0(i64 4, ptr %2) #3
  call void @llvm.lifetime.start.p0(i64 8, ptr %3) #3
  store i64 1, ptr %3, align 8, !tbaa !5
  %5 = call i32 (ptr, ...) @__isoc99_scanf(ptr noundef @.str, ptr noundef %2)
  call void @llvm.lifetime.start.p0(i64 4, ptr %4) #3
  store i32 1, ptr %4, align 4, !tbaa !9
  br label %6

6:                                                ; preds = %16, %0
  %7 = load i32, ptr %4, align 4, !tbaa !9
  %8 = load i32, ptr %2, align 4, !tbaa !9
  %9 = icmp sle i32 %7, %8
  br i1 %9, label %11, label %10

10:                                               ; preds = %6
  call void @llvm.lifetime.end.p0(i64 4, ptr %4) #3
  br label %19

11:                                               ; preds = %6
  %12 = load i32, ptr %4, align 4, !tbaa !9
  %13 = sext i32 %12 to i64
  %14 = load i64, ptr %3, align 8, !tbaa !5
  %15 = mul i64 %14, %13
  store i64 %15, ptr %3, align 8, !tbaa !5
  br label %16
```

```
42
43  16:                                               ; preds = %11
44    %17 = load i32, ptr %4, align 4, !tbaa !9
45    %18 = add nsw i32 %17, 1
46    store i32 %18, ptr %4, align 4, !tbaa !9
47    br label %6, !llvm.loop !11
48
49  19:                                               ; preds = %10
50    %20 = load i64, ptr %3, align 8, !tbaa !5
51    %21 = call i32 (ptr, ...) @printf(ptr noundef @.str.1, i64 noundef %20)
52    call void @llvm.lifetime.end.p0(i64 8, ptr %3) #3
53    call void @llvm.lifetime.end.p0(i64 4, ptr %2) #3
54    ret i32 0
55  }
56
57  ; Function Attrs: nocallback nofree nosync nounwind willreturn memory(argmem:
        readwrite)
58  declare void @llvm.lifetime.start.p0(i64 immarg, ptr nocapture) #1
59
60  declare i32 @__isoc99_scanf(ptr noundef, ...) #2
61
62  ; Function Attrs: nocallback nofree nosync nounwind willreturn memory(argmem:
        readwrite)
63  declare void @llvm.lifetime.end.p0(i64 immarg, ptr nocapture) #1
64
65  declare i32 @printf(ptr noundef, ...) #2
66
67  attributes #0 = { nounwind uwtable "min-legal-vector-width"="0" "no-trapping-
        math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "
        target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="
        generic" }
68  attributes #1 = { nocallback nofree nosync nounwind willreturn memory(argmem:
        readwrite) }
69  attributes #2 = { "no-trapping-math"="true" "stack-protector-buffer-size"="8"
        "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+
        sse2,+x87" "tune-cpu"="generic" }
70  attributes #3 = { nounwind }
71
72  !llvm.module.flags = !{!0, !1, !2, !3}
73  !llvm.ident = !{!4}
74
75  !0 = !{i32 1, !"wchar_size", i32 4}
76  !1 = !{i32 8, !"PIC Level", i32 2}
77  !2 = !{i32 7, !"PIE Level", i32 2}
78  !3 = !{i32 7, !"uwtable", i32 2}
79  !4 = !{!"Ubuntu clang version 18.1.3 (1ubuntu1)"}
80  !5 = !{!6, !6, i64 0}
81  !6 = !{!"long long", !7, i64 0}
```

```
!7 = !{!"omnipotent char", !8, i64 0}
!8 = !{!"Simple C/C++ TBAA"}
!9 = !{!10, !10, i64 0}
!10 = !{!"int", !7, i64 0}
!11 = distinct !{!11, !12}
!12 = !{!"llvm.loop.mustprogress"}
```

这个 passes 能简化控制流图（CFG），合并冗余的基本块，删除无用的条件跳转等。在初始 IR 中，循环结构通常会有多个分支，控制跳转语句可能会引入多余的跳转逻辑。SimplifyCFGPass 会消除这些不必要的跳转，使代码变得更加线性。

- **删除 noinline 和 optnone 属性**

未优化 IR：

```
; Function Attrs: noinline
    nounwind optnone uwtable
define dso_local i32 @main() #0
    {
```

优化后 IR：

```
; Function Attrs: nounwind
    uwtable
define dso_local i32 @main() #0
    {
```

删除 optnone 允许优化器对函数执行各种优化传递。如果有益，则消除 noinline 允许内联函数。

- **插入 Lifetime Intrinsics**

```
call void @llvm.lifetime.start.p0(i64 4, ptr %2) #3
call void @llvm.lifetime.start.p0(i64 8, ptr %3) #3
...
call void @llvm.lifetime.end.p0(i64 4, ptr %4) #3
call void @llvm.lifetime.end.p0(i64 8, ptr %3) #3
call void @llvm.lifetime.end.p0(i64 4, ptr %2) #3
```

这些内部函数会告知优化器变量的确切生命周期，从而实现更好的内存管理、潜在的堆栈插槽重用，并消除不必要的内存操作。

- **添加 TBAA（基于类型的别名分析）元数据**

```
store i64 1, ptr %3, align 8, !tbaa !5
%7 = load i32, ptr %4, align 4, !tbaa !9
```

优化后附加！tbaa 元数据以加载和存储指令以提供类型信息。TBAA 元数据可帮助编译器了解指针别名，通过假设不同类型的指针不会相互别名，从而允许进行更积极的优化。

- **循环体和递增体的合并**

```
; Block %13 combines multiplication and increment
%14 = load i32, ptr %4, align 4, !tbaa !9
%15 = sext i32 %14 to i64
%16 = load i64, ptr %3, align 8, !tbaa !5
%17 = mul i64 %16, %15
store i64 %17, ptr %3, align 8, !tbaa !5
%18 = load i32, ptr %4, align 4, !tbaa !9
%19 = add nsw i32 %18, 1
```

```
store i32 %19, ptr %4, align 4, !tbaa !9
br label %6, !llvm.loop !11
```

循环体和循环变量的增量已合并为一个基本块%13。合并这些操作减少了基本块和分支的数量，从而在循环中实现更简化和高效的代码执行。

– **重写 Branch 指令**

```
br i1 %9, label %13, label %10
```

块%6 中的条件分支现在直接分支到循环体%13 或出口块%10。直接分支到适当的 blocks 可以简化控制流程并减少不必要的跳转，从而提高性能。

– **提前退出和清理**

```
; Exit block %10
call void @llvm.lifetime.end.p0(i64 4, ptr %4) #3
%11 = load i64, ptr %3, align 8, !tbaa !5
%12 = call i32 (ptr, ...) @printf(ptr noundef @.str.1, i64 noundef %11)
call void @llvm.lifetime.end.p0(i64 8, ptr %3) #3
call void @llvm.lifetime.end.p0(i64 4, ptr %2) #3
ret i32 0
```

退出块%10 现在包含结束变量生命周期并从函数返回的清理代码。将清理代码合并到单个块中可确保资源释放和函数退出高效发生，而不会产生不必要的分支。

– **循环元数据增强**

```
br label %6, !llvm.loop !11
```

循环包含元数据（! llvm.loop）以指导潜在的循环优化。此元数据为优化器提供有关循环行为的其他信息，从而启用展开或矢量化等优化。

– **改进的变量范围界定和寿命分析**

通过显式标记变量生命周期，优化器可以更好地了解何时不再需要变量。这可以减少堆栈使用，消除不必要的加载/存储，并可能改进寄存器分配。

在整体分析上，对于每个 IR 文件分为了六个部分：

- **模块信息：** 介绍了模块来源、数据在内存中的布局例如对齐方式、数据类型的大小等），以及目标平台，是 x86_64 架构、Linux 操作系统。

```
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-
    f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"
```

- **常量字符串定义：** @.str 定义了一个字符串常量"%d"，用于 scanf 函数，读取一个整数。
  @.str.1: 定义了一个字符串常量"%llu
  n"，用于 printf 函数，输出一个 unsigned long long 类型的整数，正是我在源代码中写的 printf("%llu
  n", factorial);。

```
@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@.str.1 = private unnamed_addr constant [6 x i8] c"%llu\0A\00", align 1
```

- **main 函数定义：** 定义了 main 函数，返回类型为 i32，即整型。dso_local: 表示该函数在模块内部使用，不需要进行动态符号解析。

```
define dso_local i32 @main() #0 {
```

使用 alloca 分配栈上空间来存储局部变量，i32 是整型变量，i64 是 64 位整数变量:

```
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i64, align 8
%4 = alloca i32, align 4
```

使用 store 指令向内存中存储值:

```
store i32 0, ptr %1, align 4
store i64 1, ptr %3, align 8
```

通过 call 进行函数调用 @__isoc99_scanf 也就是 scanf 函数:

```
%5 = call i32 (ptr, ...) @__isoc99_scanf(ptr noundef @.str, ptr noundef %2)
```

以及其余部分对应的 IR。

- **函数声明：** 声明了两个外部函数 scanf 和 printf。它们使用的是标准库函数，编译和链接时会从标准库中引用。

```
declare i32 @__isoc99_scanf(ptr noundef, ...) #1
declare i32 @printf(ptr noundef, ...) #1
```

- **函数属性：**

```
attributes #0 = { noinline nounwind optnone uwtable ... }
attributes #1 = { ... }
```

- **LLVM 元数据：** 记录模块信息、编译器标识等:

```
!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}
!6 = distinct !{!6, !7}
!7 = !{!"llvm.loop.mustprogress"}
```

6. 代码生成

这里使用 gcc 生成目标代码，为方便阅读和下面的反汇编结果进行对比分析，生成 X86-64 架构下的 intel 汇编语言，跳过了头部的一些声明:

```
main:
.LFB0:
        .cfi_startproc
        endbr64
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        sub     rsp, 32
        mov     rax, QWORD PTR fs:40
        mov     QWORD PTR -8[rbp], rax
        xor     eax, eax
        mov     QWORD PTR -16[rbp], 1
        lea     rax, -24[rbp]
        mov     rsi, rax
        lea     rax, .LC0[rip]
        mov     rdi, rax
        mov     eax, 0
        call    __isoc99_scanf@PLT
        mov     DWORD PTR -20[rbp], 1
        jmp     .L2
.L3:
        mov     eax, DWORD PTR -20[rbp]
        cdqe
        mov     rdx, QWORD PTR -16[rbp]
        imul    rax, rdx
        mov     QWORD PTR -16[rbp], rax
        add     DWORD PTR -20[rbp], 1
.L2:
        mov     eax, DWORD PTR -24[rbp]
        cmp     DWORD PTR -20[rbp], eax
        jle     .L3
        mov     rax, QWORD PTR -16[rbp]
        mov     rsi, rax
        lea     rax, .LC1[rip]
        mov     rdi, rax
        mov     eax, 0
        call    printf@PLT
        mov     eax, 0
        mov     rdx, QWORD PTR -8[rbp]
        sub     rdx, QWORD PTR fs:40
        je      .L5
        call    __stack_chk_fail@PLT
.L5:
        leave
        .cfi_def_cfa 7, 8
```

```
59          ret
60          .cfi_endproc
```

(三) 汇编器（Assembler）

1. 汇编器处理的结果

在本实验平台下，汇编器最终的处理结果是生成一个.o 文件，其中包含 CPU 可执行的机器码。这些机器码直接对应汇编代码中的每条指令。反汇编工具可以将机器码重新转换为汇编代码，但在这个过程中，某些符号信息可能丢失，反汇编代码的可读性通常不如源汇编代码。通过 gcc main.S -c -o main.o 对 x86 格式的汇编直接转化成机器码，然后通过 gcc 进行反汇编查看其处理结果：

```
5  0000000000000000 <main>:
6        0: f3 0f 1e fa                  endbr64
7        4: 48 83 ec 18                  sub    rsp, 0x18
8        8: 48 8d 3d 00 00 00 00         lea    rdi, [rip]        # 0xf
           <main+0xf>
9        f: 64 48 8b 04 25 28 00 00 00   mov    rax, qword ptr fs:[0x28]
10      18: 48 89 44 24 08               mov    qword ptr [rsp + 0x8], rax
11      1d: 31 c0                        xor    eax, eax
12      1f: 48 8d 74 24 04               lea    rsi, [rsp + 0x4]
13      24: e8 00 00 00 00               call   0x29 <main+0x29>
14      29: 8b 4c 24 04                  mov    ecx, dword ptr [rsp + 0x4]
15      2d: 85 c9                        test   ecx, ecx
16      2f: 7e 5e                        jle    0x8f <main+0x8f>
17      31: 8d 71 01                     lea    esi, [rcx + 0x1]
18      34: 83 e1 01                     and    ecx, 0x1
19      37: b8 01 00 00 00               mov    eax, 0x1
20      3c: ba 01 00 00 00               mov    edx, 0x1
21      41: 74 0d                        je     0x50 <main+0x50>
22      43: b8 02 00 00 00               mov    eax, 0x2
23      48: 48 39 f0                     cmp    rax, rsi
24      4b: 74 18                        je     0x65 <main+0x65>
25      4d: 0f 1f 00                     nop    dword ptr [rax]
26      50: 48 0f af d0                  imul   rdx, rax
27      54: 48 8d 48 01                  lea    rcx, [rax + 0x1]
28      58: 48 83 c0 02                  add    rax, 0x2
29      5c: 48 0f af d1                  imul   rdx, rcx
30      60: 48 39 f0                     cmp    rax, rsi
31      63: 75 eb                        jne    0x50 <main+0x50>
32      65: 31 c0                        xor    eax, eax
33      67: 48 8d 35 00 00 00 00         lea    rsi, [rip]        # 0
          x6e <main+0x6e>
34      6e: bf 02 00 00 00               mov    edi, 0x2
35      73: e8 00 00 00 00               call   0x78 <main+0x78>
```

我们对比分析汇编和反汇编的各个部分：

- **文件头部与基本设置**

汇编代码：

```
.file "main.c"
.intel_syntax noprefix
.text
```

在汇编代码中包含信息：源文件名、使用 Intel 汇编语法，不使用前缀、各个段的名称。

反汇编代码：

由于是通过机器码反汇编获得，机器码不包含这些元数据。

- **函数头部**

汇编代码：

```
.globl main
.type main, @function
main:
.LFB0:
```

声明 main 全局符号，声明 main 函数。

反汇编代码：

```
0000000000000000 <main>:
    0: f3 0f 1e fa endbr64
```

反汇编直接从指令 endbr64 开始，表示此处为函数 main 的起始位置。

- **栈帧设置**

汇编代码：

```
push rbp
mov rbp, rsp
sub rsp, 32
```

将当前栈指针保存到 rbp，帧指针抬高 32 个字节，在栈上分配 32 字节的空间。

反汇编代码：

```
4: 48 83 ec 18 sub rsp, 0x18
```

反汇编中的 sub rsp, 0x18 对应栈的调整，但这里分配的是 24 字节（0x18），和汇编代码不同，这是因为汇编器将汇编代码转化成机器语言的时候进行了一定程度的优化。

- **寄存器操作和内存访问**

汇编代码：

```
mov rax, QWORD PTR fs:40
mov QWORD PTR -8[rbp], rax
xor eax, eax
```

从线程局部存储区（fs 段寄存器）加载一个值到 rax，将该值保存到栈上的局部变量，然后将 eax 寄存器清零。

反汇编代码：

```
f: 64 48 8b 04 25 28 00 00 00
    mov rax, qword ptr fs:[0x28]
18: 48 89 44 24 08 mov qword ptr
    [rsp + 0x8], rax
1d: 31 c0 xor eax, eax
```

反汇编与汇编中的内存访问一致，同样访问了 fs 段寄存器偏移量为 40 字节（0x28）的地址。

- **函数调用和参数传递**

汇编代码：

```
lea rax, -24[rbp]
mov rsi, rax
lea rax, .LC0[rip]
mov rdi, rax
call __isoc99_scanf@PLT
```

使用 lea 指令准备参数，将局部变量地址和字符串地址传递给 scanf 函数。

反汇编代码：

```
1f: 48 8d 74 24 04 lea rsi, [rsp + 0x4]
24: e8 00 00 00 00 call 0x29 <main+0x29>
```

反汇编代码解析函数调用位置地址为 0x29，由于机器码无法存储对应函数的名称信息，指挥记录函数调用地址。

- **循环与算术操作**

汇编代码：

```
imul rax, rdx
add DWORD PTR -20[rbp], 1
cmp DWORD PTR -20[rbp], eax
jle .L3
```

这里进行了有符号整数乘法，然后将对应的局部变量加一，然后进行判断循环。

反汇编代码：

```
34: 83 e1 01 and ecx, 0x1
37: b8 01 00 00 00 moveax, 0x1
3c: ba 01 00 00 00 mov edx, 0x1
41: 74 0d je 0x50 <main+0x50>
43: b8 02 00 00 00 mov eax, 0x2
48: 48 39 f0 cmp rax, rsi
4b: 74 18 je 0x65 <main+0x65>
4d: 0f 1f 00 nop dword ptr [rax]
50: 48 0f af d0 imul rdx, rax
54: 48 8d 48 01 lea rcx, [rax + 0x1]
58: 48 83 c0 02 add rax, 0x2
5c: 48 0f af d1 imul rdx, rcx
60: 48 39 f0 cmp rax, rsi
63: 75 eb jne 0x50 <main+0x50>
```

反汇编代码和汇编代码功能一样，但是实现的逻辑和汇编代码有所不同，这里类似的进行了一种循环展开，先对 ecx 取最低位，如果最最低为 1（即奇数），那么不进行跳转，进行一次乘法；如果最低位是 0（即偶数），进行跳转。这之后剩余的循环只剩下偶数次，那么在反汇编中可以看到，它每个循环进行了两次乘法，实现了减少跳转次数。由此可以看出，汇编器对汇编代码进行了一定程度的有优化。

- **栈保护与函数结束**

汇编代码：

```
mov rdx, QWORD PTR -8[rbp]
sub rdx, QWORD PTR fs:40
je .L5
```

使用 lea 指令准备参数，将局部变量地址和字符串地址传递给 scanf 函数。

反汇编代码：

```
78: 48 8b 44 24 08 mov rax, qword ptr [rsp + 0x8]
7d: 64 48 2b 04 25 28 00 00 00 sub rax, qword ptr fs:[0x28]
86: 75 0e jne 0x96 <main+0x96>
```

与汇编代码使用相同的逻辑。

2. 汇编器的功能

汇编器是将汇编语言代码（人类可读的指令和寄存器操作）转换为机器码（CPU 执行的二进制指令）的工具。从以上的分析中我们可以看到，汇编器的核心功能是将汇编语言中的每条指令翻译为相应的机器指令，汇编语言中常常使用符号（如标签、函数名、变量名）来代替内存地址和数据地址，汇编器的任务是将这些符号解析为实际的内存地址。同时汇编器将汇编代码中的不同部分（代码、数据等）分配到合适的段中，然后对指令以及各个跳转的地址进行计算，包括对于一些立即数转化成机器码的格式（比如上面提到的 x24）。最重要的是汇编器会对于汇编代码进行一定程度的优化，我们在上述的例子中可以看到他对于循环进行了循环展开优化。

（四） 链接器（Linker）

链接器用于将编译后的多个目标文件（object files）或库文件（libraries）组合在一起，生成一个可执行文件或库。它的主要作用是解决程序中的符号和地址依赖，并完成最终的代码合成。链接器的两种主要类型：

- **静态链接器：**将库中的代码直接复制到最终的可执行文件中，生成的文件是独立的，不需要在运行时依赖外部库。
- **动态链接器：**将库与程序分离，在程序运行时通过操作系统动态加载库文件，减少了可执行文件的大小，并支持库的共享。

（五） 总结

具体的步骤如图3所示，GCC 的语言处理过程大致分为四个阶段：预处理器将源代码中的宏、文件包含等指令进行展开；编译器将源代码转换为汇编代码，并进行优化；汇编器将汇编代码转换为机器码生成目标文件；最后，链接器将多个目标文件链接成可执行文件。每个阶段都有其明确的职责，通过分阶段的处理方式，提高了编译器的灵活性和可维护性。

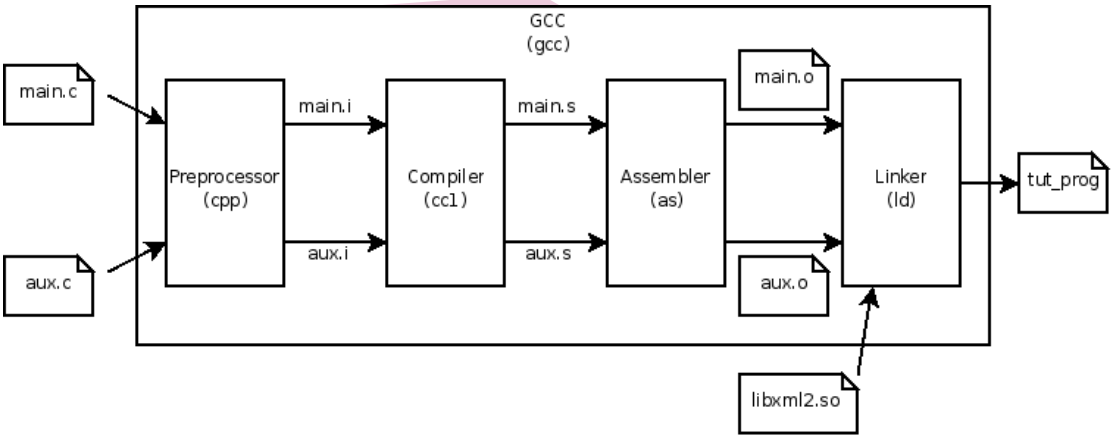

图 3: GCC Preprocessor

三、 LLVM IR 编程

LLVM IR（Intermediate Representation，中间表示）是 LLVM 编译器框架的核心，它是编译器生成目标代码前的一种中间代码。LLVM IR 是一种类似于汇编语言的中间表示形式，由多个基本单元构成。一个完整的 LLVM IR 代码通常由以下几个关键部分组成：

(一) 函数声明

声明外部函数告知 LLVM 编译器这些函数在其他地方定义，可能是标准库函数或者由其他代码提供的函数。在本次作业的函数中，由于 IO 函数的实现需要从 RISC-V 汇编代码一层一层封装，这里是使用直接外部文件定义使用 c 语言的 IO 函数。

```
declare i32 @getint()
declare void @putint(i32)
declare void @putch(i32)
```

(二) 全局变量

在全局范围内声明的变量，通常用于存储程序中多处共享的数据，生命周期贯穿整个程序的执行。

(三) 函数定义

1. **函数头:** 声明返回类型、参数类型、函数名称。
2. **基本块:** 函数体由若干个基本块组成，每个基本块都有一个标签作为入口，例如 entry 标签是整个函数的入口。
3. **指令:** 基本块内包含顺序执行的指令，每个指令可能包括变量加载、存储、计算、控制流等操作，最后通过 ret 指令返回。

(四) 一些函数的定义

1. alloca 指令

- **语法:** <result> = alloca <type>
- **功能:** alloca 指令分配一个局部变量的空间，并返回一个指向该空间的指针。
- **示例:**

```
        %a = alloca i32
        %b = alloca i32
```

在这里，alloca i32 为 i32 类型的变量分配内存空间（相当于在栈上为整数变量分配空间）。%a、%b 等是指向这些内存空间的指针，也就是说%a 等是指向 32 位整数的指针（i32*）。

2. store 指令

- **语法:** store <type> <value>, <type>* <ptr>
- **功能:** 将 <value> 存储到指针 <ptr> 指向的内存位置。
- **示例:**

```
        store i32 42, i32* %a
```

这个指令的意思是将整数 42 存储到指针%a 所指向的内存位置。%a 是一个 i32*，即指向 i32 类型的内存地址。

3. load 指令

- **语法：** <result> = load <type>, <type>* <ptr>
- **功能：** 从指针 <ptr> 指向的内存位置加载一个值，并将其赋给 <result>。
- **示例：**

```
%x = load i32, i32* %a
```

这个指令的意思是从指针%a 所指向的内存位置加载一个 32 位整数值，并将其赋值给%x。%a 的类型是 i32*，所以它指向一个存储 i32 值的内存地址。

4. call 指令

- **语法：** <result> = call <return type> <function>(<args>)
- **功能：** call 指令用于调用函数，并可返回一个值。
- **示例：**

```
%n_value = call i32 @getint()
```

5. br 指令

- **语法：**
  - 无条件跳转：br label <destination>
  - 条件跳转：br i1 <cond>, label <if_true>, label <if_false>
- **功能：** br 是“branch”（分支）指令，它控制程序的跳转。可以是条件跳转或无条件跳转。
- **示例：**

```
br label %while_cond
br i1 %cmp, label %while_body, label %while_end
```

6. icmp 指令

- **语法：** <result> = icmp <predicate> <type> <lhs>, <rhs>
- **功能：** icmp 指令用于整数比较，返回一个布尔值（i1 类型，0 或 1）。
- **示例：**

```
%cmp = icmp slt i32 %i_val, %n_val
```

<predicate>：指定比较类型，例如 eq（等于）、slt（有符号小于）等。

7. add 指令

- **语法：** <result> = add <type> <op1>, <op2>
- **功能：** add 指令执行整数加法运算。
- **示例：**

```
%sum = add i32 %a_val2, %b_val2
```

将两个 i32 整数%a_val2 和%b_val2 相加，结果存储在%sum 中。

8. add 指令

- **语法：** ret <type> <value>
- **功能：** ret 指令用于函数返回，它将控制权交还给调用者，并可以选择性返回一个值。
- **示例：**

```
ret i32 0
```

(五) 示例

1. fibonacci 数列

根据源码编写 LLVM IR 见 二 先使用 llc 将 IR 编译成目标文件

```
llc -march=riscv64 -mattr=+m,+a,+f,+d,+c -filetype=obj fibonacci.ll -o
    fibonacci.o
```

然后将目标文件放在编译链下编译成可执行文件

```
riscv64-unknown-elf-gcc -march=rv64imafdc -mabi=lp64d -o fibonacci
    fibonacci.o support.c
```

最后在 qemu 模拟器上运行这个可执行文件

```
qemu-riscv64 ./fibonacci
```

2. 矩阵乘法

LLVM IR 实现见五，实现了更复杂的循环判断以及 IO，成功实现将 2 * 3 的矩阵和 3 * 2 的矩阵相乘得到最后 2 * 2 的矩阵，使用 double 实现浮点计算。

(六) 遇到问题以及解决

1. 类型问题

在函数调用以及赋值时一开始会出现总是弄不清楚指针和值的问题，在后面不断的调整以及纠正之后解决。

2. 跳转问题

LLVM IR 似乎和汇编不一样，LLVM IR 的基本块结束都要使用 br 跳转指令跳转到另一个块，而不会像汇编一样顺序执行。如果没有跳转，会将标签进行汇编，导致报错 error: expected instruction opcode，即使不需要跳转直接顺序执行也要写一个无条件跳转。

3. 变量赋值问题

在 LLVM IR 中，不同的基本块可以共享变量，但需要通过显式的加载和存储指令来访问这些变量。这是因为 LLVM IR 是一种静态单赋值（SSA）形式的中间表示语言，每个变量只能被赋值一次。因此，变量的值需要通过内存访问来传递。所以在编写 LLVM IR 的时候，为了在不同的基本块之间共享这些变量的值，代码使用了 alloca 指令在栈上分配内存，并通过 load 和 store 指令来访问和修改这些变量的值。

4. 浮点计算的实现

原本 LLVM IR 中有 float 类型，但是经过 llc 生成目标语言和编译链生成 RICS-V 可执行文件的时候 float 类型会导致计算结果是 0.0000，有可能是编译选项的错误，这里的解决方法是使用 double 类型进行计算。

四、 RISC-V 汇编代码

RISC-V 64 位架构（RISCV64）是一种开源指令集，广泛应用于计算机和嵌入式系统。其汇编代码通常分为数据段和代码段。在数据段中，程序员可以定义和初始化字符串、数组等数据。而在代码段中，使用助记符编写指令，指示处理器执行具体的操作，如算术计算和条件跳转。下面是对 RISC-V 汇编代码主要组成部分的介绍，以及一些例子。

(一) 指令标签（Labels）

用于标识代码中的某个位置，可以用于标识函数的入口点，允许其他代码通过调用指令跳转到该函数。例如，可以定义一个标签 myFunction: 来表示一个子程序的开始位置。同时标签也常用于条件跳转指令，如 beq（相等跳转）和 bne（不相等跳转）。通过在代码中设置标签，程序可以根据条件的真假跳转到相应的位置，在实现循环结构时，标签可以用作循环的起始点和结束点，使得代码更加清晰易懂。

```
.section .text          # 代码段
.globl main             # 声明主函数为全局符号
main:                   # 主函数入口
    li t0, 1            # 初始化计数器 t0 为 1
    li t1, 0            # 初始化总和 t1 为 0

loop:                   # 循环标签
    add t1, t1, t0      # 将当前计数器 t0 加入总和 t1
    addi t0, t0, 1      # 计数器加 1
    li t2, 10           # 将 10 加载到 t2
    blt t0, t2, loop    # 如果 t0 小于 10，则跳转到 loop

    # 此时 t1 中存储了 1 到 10 的总和
```

```
    li a7, 10           # 系统调用号 10 (exit)
    ecall               # 触发系统调用
```

在这个示例中，main: 标签标识了程序的入口。首先将计数器 t0 初始化为 1，总和 t1 初始化为 0。在 loop: 标签处，程序进行循环，每次将 t0 的值加到 t1 中，然后 t0 自增 1。使用 blt 指令进行条件跳转，判断 t0 是否小于 10，如果是，则继续循环。当 t0 等于 10 时，循环结束，程序继续执行后续指令。这样，通过标签和条件跳转，代码实现了简单的循环结构。

(二) 段定义 (Segment)

在汇编语言中，段定义是组织程序代码和数据的重要结构，主要分为代码段、数据段和堆栈段。

代码段是程序的核心部分，包含了可执行的指令。每条指令由助记符及其操作数组成，指示计算机执行特定的操作。代码段通常是程序启动后首先被加载到内存中的部分，它负责控制程序的执行流程。在这一段中，我们需要明确指明每个指令的顺序，以及如何通过跳转指令实现程序的逻辑控制，例如循环、条件判断和函数调用等。

```
.section .text          # 代码段
.globl main             # 声明主函数为全局符号
main:                   # 主函数入口
    li a0, 5            # 将数字 5 加载到寄存器 a0
    li a1, 3            # 将数字 3 加载到寄存器 a1
    add a2, a0, a1      # 将 a0 和 a1 相加，结果存入 a2
    # 结果现在在 a2 中
    li a7, 10           # 系统调用号 10 (exit)
    ecall               # 触发系统调用
```

数据段则是用于定义和初始化程序中使用的数据。在这一部分，我们可以声明变量、数组和字符串等，设定初始值。数据段的内容在程序运行时通常会被加载到内存中，以便于后续的读写操作。这一段的设计至关重要，因为合理的内存管理和数据组织能够提高程序的效率和可读性。例如，如果需要在程序中使用一个常量字符串，我们会在数据段中定义该字符串，并为其分配内存空间。

```
.section .data          # 数据段
greeting: .asciz "Hello, World!"  # 定义字符串并以空字符结尾
numbers: .word 1, 2, 3, 4, 5      # 定义整数数组
```

堆栈段则用于管理程序的堆栈。堆栈是一种特殊的内存区域，用于存储临时数据，比如函数调用的参数、返回地址和局部变量。在汇编语言中，堆栈的操作通常涉及到压栈和弹栈，这两种操作分别用于向堆栈中添加数据和从堆栈中移除数据。通过堆栈，程序能够有效地管理函数调用的层次结构，确保在多个函数之间传递数据时，能够保持调用的上下文。因此，堆栈段在程序的执行过程中起着至关重要的作用，尤其是在递归调用和复杂的函数参数传递时。

```
.section .text          # 代码段
.globl main             # 声明主函数为全局符号
main:                   # 主函数入口
    li a0, 10           # 将参数 10 加载到寄存器 a0
    call myFunction     # 调用函数 myFunction
    li a7, 10           # 系统调用号 10 (exit)
```

```
    ecall                   # 触发系统调用

myFunction:                 # 函数入口
    addi sp, sp, -4         # 压栈，分配空间
    sw a0, 0(sp)            # 将参数存入堆栈
    # 这里可以添加其他代码
    lw a0, 0(sp)            # 从堆栈弹出参数
    addi sp, sp, 4          # 弹栈，释放空间
    ret                      # 返回调用
```

在这个示例中，myFunction 函数使用堆栈来保存参数，并在返回时恢复这些参数，从而管理函数调用的上下文。

(三) 指令（instructions）

汇编指令是汇编语言的基本构建块，主要由助记符和操作数组成。助记符是人类可读的文本，代表计算机能够执行的特定操作，如加法、减法、跳转等。操作数则是指令所需的数据，可能是寄存器、内存地址或立即数。汇编指令的格式通常是“助记符操作数 1, 操作数 2, ...”，并通过这种结构指示计算机执行特定的任务。

```
li a0, 5                # 将立即数 5 加载到寄存器 a0
li a1, 3                # 将立即数 3 加载到寄存器 a1
add a2, a0, a1          # 将 a0 和 a1 相加，结果存入 a2
```

add 指令用于将两个寄存器中的值相加，并将结果存入另一个寄存器。助记符 add 表示加法操作，而操作数则指定了参与运算的寄存器。

```
li a0, 10               # 将立即数 10 加载到寄存器 a0
li a1, 5                # 将立即数 5 加载到寄存器 a1
bne a0, a1, notEqual # 如果 a0 不等于 a1，跳转到 notEqual 标签
# 这里执行 a0 等于 a1 的逻辑

notEqual:               # 标签
    # 处理 a0 不等于 a1 的逻辑
```

bne（branch if not equal）指令用于判断 a0 和 a1 是否不相等。如果不相等，则跳转到 notEqual 标签处执行相应的逻辑。

(四) 伪指令（Pseudo-instructions）

汇编伪指令是汇编语言中的一些指令，它们并不直接对应于机器指令，而是用于指导汇编器如何处理源代码。这些伪指令在程序的编译和链接过程中起着重要的作用，例如定义数据、分配内存或设置段的属性。伪指令使得汇编语言更加灵活和易用，通常以. 开头，如`.globl main` 声明 `main` 为全局符号。

例如，call 伪指令在汇编语言中用于调用函数或子程序，它将当前指令的下一条地址压入堆栈，以便在函数执行完成后可以正确返回。这种机制允许程序在执行过程中跳转到其他代码段，完成特定的功能，然后再返回到调用的位置继续执行。

在 RISC-V 架构中，call 伪指令实际上对应于两个真实的指令：jal（Jump and Link）。jal 指令的作用是跳转到指定的地址，并将返回地址（即当前指令的下一条指令的地址）保存到 ra（返回地址寄存器）中。

```
.section .text              # 代码段
.globl main                 # 声明主函数为全局符号

main:                       # 主函数入口
    li a0, 10               # 将参数 10 加载到寄存器 a0
    call myFunction         # 调用 myFunction 函数
    # 函数返回后继续执行
    li a7, 10               # 系统调用号 10 (exit)
    ecall                   # 触发系统调用

myFunction:                 # 函数入口
    addi a0, a0, 5          # 将参数加 5
    ret                      # 返回到调用点
```

call myFunction 伪指令用于调用 myFunction 函数。在汇编器处理这个伪指令时，它会将其转换为以下真实指令：

使用 jal 指令跳转到 myFunction 的入口，同时将下一条指令的地址保存到 ra 寄存器。在 myFunction 执行完毕后，使用 ret 指令（等效于 jalr 指令）从 ra 寄存器中获取返回地址，跳转回主函数的执行位置。

（五） 单精度浮点数矩阵乘法实现

对于下面单精度浮点数矩阵乘法 sysy 语言代码，我们实现了其汇编版本，这里是其矩阵乘法函数部分。

Listing 1: 矩阵乘法的 SysY 实现

```
const int ROW_A = 2, COL_A = 3; // 矩阵 A 的行数和列数
const int ROW_B = 3, COL_B = 2; // 矩阵 B 的行数和列数

// 定义矩阵乘法函数，计算浮点数矩阵
void matrix_multiply(float A[][COL_A], float B[][COL_B], float C[][COL_B]) {
    // 初始化结果矩阵 C 为 0
    @for (int i = 0; i < ROW_A; i = i + 1) {
        @for (int j = 0; j < COL_B; j = j + 1) {
            C[i][j] = 0.0;
        }
    }

    // 进行矩阵乘法
    @for (int i = 0; i < ROW_A; i = i + 1) {
        @for (int j = 0; j < COL_B; j = j + 1) {
            @for (int k = 0; k < COL_A; k = k + 1) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

```

```
// 主函数: 包含输入、输出和计时
int main() {
    float A[ROW_A][COL_A]; // 矩阵 A
    float B[ROW_B][COL_B]; // 矩阵 B
    float C[ROW_A][COL_B]; // 结果矩阵 C

    // 输入矩阵 A
    putf("请输入矩阵 A 的元素, 共 %d 个元素: \n", ROW_A * COL_A);
    @for (int i = 0; i < ROW_A; i = i + 1) {
        @for (int j = 0; j < COL_A; j = j + 1) {
            A[i][j] = getfloat(); // 读取浮点数
        }
    }

    // 输入矩阵 B
    putf("请输入矩阵 B 的元素, 共 %d 个元素: \n", ROW_B * COL_B);
    @for (int i = 0; i < ROW_B; i = i + 1) {
        @for (int j = 0; j < COL_B; j = j + 1) {
            B[i][j] = getfloat(); // 读取浮点数
        }
    }

    // 启动计时器, 开始记录矩阵乘法的执行时间
    starttime();

    // 调用矩阵乘法函数
    matrix_multiply(A, B, C);

    // 停止计时器
    stoptime();

    // 输出结果矩阵 C
    putf("矩阵乘法结果为: \n");
    @for (int i = 0; i < ROW_A; i = i + 1) {
        @for (int j = 0; j < COL_B; j = j + 1) {
             putf("%f ", C[i][j]); // 使用标准的浮点数输出格式
            if (j < COL_B - 1) {
                putch(32); // 每个元素之间用空格隔开
            } else {
                putch(10); // 每行结束后换行
            }
        }
    }

    return 0;
}
```

以下是实现矩阵乘法的汇编代码:

Listing 2: 矩阵乘法的 RISC-V 汇编实现

```
1
2  #矩阵乘法
3  matrix_multiply:
4      # 初始化目标矩阵C的前4个元素为0
5      sw   zero, 0(a2)            # C[0][0] = 0
6      sw   zero, 4(a2)            # C[0][1] = 0
7      sw   zero, 8(a2)            # C[1][0] = 0
8      sw   zero, 12(a2)           # C[1][1] = 0
9      #用来在对C矩阵按行遍历时作为停止标志
10     addi     a7, a2, 16         # 设置a7指向C矩阵的末尾 (a2 + 16)
11
12     addi     a6, a1, 28         # 设置a6指向B矩阵的末尾 (a1 + 28)
13
14 #外层循环, 用于处理矩阵的行
15 Matrix_Row_Loop:
16     mv   a4, a1                 # 将a4设置为B矩阵的起始地址
17     mv   a5, a0                 # 将a5设置为A矩阵的起始地址
18     addi     a3, a0, 12         # 设置a3指向A矩阵的第一行末尾
19
20 #内层循环, 用于处理矩阵C第一列
21 Matrix_Col_Loop:
22     flw  fa5, 0(a4)             # 加载B的元素到fa5
23     flw  fa3, 0(a5)             # 加载A的元素到fa3
24     flw  fa4, 0(a2)             # 加载C的当前元素到fa4
25     addi     a5, a5, 4          # A矩阵元素指针加4,矩阵A按行遍历
26     addi     a4, a4, 8          # B矩阵元素指针加8,矩阵B按列遍历,由于每行有两个
           元素, 所以是加8
27     fmadd.s fa5, fa5, fa3, fa4  # 计算: fa5 = fa5 * fa3 + fa4
28     fsw  fa5, 0(a2)             # 保存结果到C矩阵
29     bne a5, a3, Matrix_Col_Loop  # 如果a5 != a3, 继续列循环
30
31     addi     a4, a1, 4          # 设置a4指向B矩阵的下一列
32     addi     a5, a0, 0          # 设置a5指向A矩阵的第一列
33 #内层循环, 用于处理矩阵C第二列
34 Next_Column_Loop:
35     flw  fa5, 0(a5)             # 加载A的元素到fa5
36     flw  fa3, 0(a4)             # 加载B的元素到fa3
37     flw  fa4, 4(a2)             # 加载C的当前元素到fa4
38     addi     a4, a4, 8          # B矩阵元素指针加8
39     addi     a5, a5, 4          # A矩阵元素指针加4
40     fmadd.s fa5, fa5, fa3, fa4  # 计算: fa5 = fa5 * fa3 + fa4
41     fsw  fa5, 4(a2)             # 保存结果到C矩阵
42     bne a6, a4, Next_Column_Loop  # 如果a4 != a6, 继续行循环
43
44     addi     a2, a2, 8          # C矩阵指针加8, 处理下一行
45     beq a2, a7, Matrix_Done     # 如果a2 == a7, 结束矩阵计算
46     mv   a0, a3                 # 重置A矩阵指针到第二行
```

```
47      j     Matrix_Row_Loop          # 跳转回到 Matrix_Row_Loop 继续处理下一行
48
49  Matrix_Done:
50      ret                            # 返回函数
```

在汇编语言实现中，矩阵乘法函数通过外层循环 Matrix_Row_Loop 实现对结果矩阵 C 的按行遍历计算。

```
1       mv   a4, a1                    # 将a4设置为B矩阵的起始地址
2       mv   a5, a0                    # 将a5设置为A矩阵的起始地址
3       addi     a3, a0, 12            # 设置a3指向A矩阵的第一行末尾
```

addi a3, a0, 12 将寄存器 a0 加上 12，指向矩阵 A 第一行的末尾地址，存储在寄存器 a3 中，用于循环结束条件的判断。

在每次外层循环中,将计算结果矩阵 C 当前行每列的值,即通过两个内层循环 Matrix_Col_Loop 和 Next_Column_Loop 分别实现对结果矩阵 C 当前行的第一列和第二列的计算，两个内层循环的逻辑相同，在内层循环中，实现输入矩阵 A 的行向量和输入矩阵 B 的列向量内积。

```
1       addi     a5, a5, 4
2       addi     a4, a4, 8
```

addi a5, a5, 4 a5 指向 A 矩阵元素，指针加 4 即矩阵 A 按行遍历每个元素。 *addi a4, a4, 8* a4 指向 B 矩阵元素，矩阵 B 按列遍历, 由于每行有两个元素，所以是加 8。

通过上述循环结构，汇编代码高效地实现了矩阵乘法的功能，充分利用了寄存器和指令集的优势，实现了对矩阵 C 的逐行、逐列计算。

通过如下指令进行编译并运行，要求输入矩阵 A 的形状为 2 行 3 列共 6 个元素，输出矩阵 B 的形状为 3 行 2 列共 6 个元素。

```
1  riscv64-unknown-linux-gnu-gcc matrix.s -o matrix -static
2  qemu-riscv64 matrix
```

输出为运行时间以及计算结果矩阵 C。代码仓库 https://github.com/copsss/bianyi

参考文献

五、 附录

(一) 斐波那契数列的 SysY 实现

```
int main() {
    int a, b, i, t, n;

    a = 0;
    b = 1;
    i = 1;

    // 输入n
    n = getint();

    // 输出初始值
    putint(a);
    putch(10); // 换行
    putint(b);
    putch(10); // 换行

    while (i < n) {
        t = b;
        b = a + b;

        // 输出当前斐波那契数
        putint(b);
        putch(10); // 换行

        a = t;
        i = i + 1;
    }

    return 0;
}
```

(二) 斐波那契数列的 LLVM IR 实现

```
declare i32 @getint()
declare void @putint(i32)
declare void @putch(i32)

define i32 @main() {
    entry:
        %a = alloca i32 ; 这里的alloca是分配内存，%a是一个指针，指向分配
            的内存
```

```
    %b = alloca i32
    %i = alloca i32
    %t = alloca i32
    %n = alloca i32

    store i32 0, i32* %a ; 这里的store是将0存储到%a指向的地址中
    store i32 1, i32* %b
    store i32 1, i32* %i

    %n_value = call i32 @getint()
    store i32 %n_value, i32* %n

    %a_val = load i32, i32* %a
    call void @putint(i32 %a_val)
    call void @putch(i32 10)

    %b_val = load i32, i32* %b
    call void @putint(i32 %b_val)
    call void @putch(i32 10)

    ; 进入while循环, i < n
    br label %while_cond

while_cond:
    %i_val = load i32, i32* %i
    %n_val = load i32, i32* %n
    %cmp = icmp slt i32 %i_val, %n_val
    br i1 %cmp, label %while_body, label %while_end

while_body:
    ; t = b
    %b_val2 = load i32, i32* %b
    store i32 %b_val2, i32* %t

    ; b = a + b
    %a_val2 = load i32, i32* %a
    %sum = add i32 %a_val2, %b_val2
    store i32 %sum, i32* %b

    ; 输出b值
    call void @putint(i32 %sum)
    call void @putch(i32 10)

    ; a = t
    %t_val = load i32, i32* %t
    store i32 %t_val, i32* %a

    ; i = i + 1
```

```
56              %i_val2 = load i32, i32* %i
57              %next_i = add i32 %i_val2, 1
58              store i32 %next_i, i32* %i
59
60              br label %while_cond
61
62          while_end:
63              ret i32 0
64      }
```

(三) 斐波那契数列的汇编实现

Listing 3: 斐波那契数列的 SysY 汇编实现

```
1  .option nopic
2  .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
3  .attribute unaligned_access, 0          # 不允许未对齐的内存访问
4  .attribute stack_align, 16
5  .text                                   # 开始代码段
6
7  .align  3
8  .format_string_scanf:.string "%d"          # 格式化字符串 "%d" 用于整数输
       入
9  .align  3
10 .format_string_printf:.string "%d\n"           # 格式化字符串 "%d\n" 用于
       输出整数并换行
11
12 .align  1
13 .globl  main                            # 定义 main 函数为全局符号
14 .type   main, @function                 # 指定 main 函数的类型为 function
15 main:                                   # main 函数的入口
16
17
18     addi    sp, sp, -64                 # 为栈分配 64 字节, 调整栈指针
19     # 保存现场
20     sd      ra, 56(sp)                  # 保存返回地址 ra 到栈中 sp + 56
21     sd      s0, 48(sp)                  # 保存寄存器 s0 到栈中 sp + 48
22     sd      s1, 40(sp)                  # 保存寄存器 s1 到栈中 sp + 40
23     sd      s2, 32(sp)                  # 保存寄存器 s2 到栈中 sp + 32
24     sd      s3, 24(sp)                  # 保存寄存器 s3 到栈中 sp + 24
25
26
27     # 输入整数
28     lui     a0, %hi(.format_string_scanf)
29     addi    a0, a0, %lo(.format_string_scanf)
30     addi    a1, sp, 12
31     call    __isoc99_scanf
32
```

```
33
34      lui     s2, %hi(.format_string_printf)
35      addi    a0, s2, %lo(.format_string_printf)
36      li      a1, 0
37      call    printf                  # 调用 printf 函数, 打印第一个斐波那
            契数 0
38
39
40      addi    a0, s2, %lo(.format_string_printf)
41      li      a1, 1
42      call    printf                  # 调用 printf 函数, 打印第二个斐波那
            契数 1
43
44      li      s0, 1                   # 初始化循环计数器 s0 = 1 (表示第几
            个斐波那契数)
45      li      s1, 1                   # 初始化当前斐波那契数值为 1
46      li      a5, 0                   # 初始化前一个斐波那契数值为 0
47
48
49  # 循环开始标签
50  fibonacci_loop_start:
51      lw      a4, 12(sp)              # 从栈中加载用户输入的值到 a4
52      bgt     a4, s0, fibonacci_loop_end  # 如果a4大于s0, 则跳转到结束标签
53
54
55      # 恢复现场
56      ld      ra, 56(sp)              # 恢复返回地址 ra
57      ld      s0, 48(sp)              # 恢复寄存器 s0
58      ld      s1, 40(sp)              # 恢复寄存器 s1
59      ld      s2, 32(sp)              # 恢复寄存器 s2
60      ld      s3, 24(sp)              # 恢复寄存器 s3
61      li      a0, 0                   # 将返回值设置为 0
62      addi    sp, sp, 64              # 恢复栈指针
63      jr      ra                      # 返回到调用者
64
65  # 循环结束标签, 继续计算下一个斐波那契数
66  fibonacci_loop_end:
67      addw    s3, a5, s1              # 计算下一个斐波那契数: s3 = a5 + s1
68
69      mv      a1, s3                  # 将新的斐波那契数存储到 a1, 准备打
            印
70      addi    a0, s2, %lo(.format_string_printf) # 设置格式化字符串 "%d\n" 的地
            址到 a0
71      call    printf                  # 打印当前的斐波那契数
72
73      mv      a5, s1                  # 更新前一个斐波那契数 a5 = s1
74      addiw   s0, s0, 1               # 计数器 s0 加 1
75      mv      s1, s3                  # 更新当前的斐波那契数 s1 = s3
```

```
76      j       fibonacci_loop_start        # 跳转回循环开始，继续计算
```

（四） 矩阵乘法的 SysY 实现

Listing 4: 矩阵乘法的 SysY 实现

```
const int ROW_A = 2, COL_A = 3; // 矩阵 A 的行数和列数
const int ROW_B = 3, COL_B = 2; // 矩阵 B 的行数和列数

// 定义矩阵乘法函数，计算浮点数矩阵
void matrix_multiply(float A[][COL_A], float B[][COL_B], float C[][COL_B]) {
    // 初始化结果矩阵 C 为 0
    @for (int i = 0; i < ROW_A; i = i + 1) {
        @for (int j = 0; j < COL_B; j = j + 1) {
            C[i][j] = 0.0;
        }
    }

    // 进行矩阵乘法
    @for (int i = 0; i < ROW_A; i = i + 1) {
        @for (int j = 0; j < COL_B; j = j + 1) {
            @for (int k = 0; k < COL_A; k = k + 1) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// 主函数：包含输入、输出和计时
int main() {
    float A[ROW_A][COL_A]; // 矩阵 A
    float B[ROW_B][COL_B]; // 矩阵 B
    float C[ROW_A][COL_B]; // 结果矩阵 C

    // 输入矩阵 A
    putf("请输入矩阵 A 的元素，共 %d 个元素：\n", ROW_A * COL_A);
    @for (int i = 0; i < ROW_A; i = i + 1) {
        @for (int j = 0; j < COL_A; j = j + 1) {
            A[i][j] = getfloat(); // 读取浮点数
        }
    }

    // 输入矩阵 B
    putf("请输入矩阵 B 的元素，共 %d 个元素：\n", ROW_B * COL_B);
    @for (int i = 0; i < ROW_B; i = i + 1) {
        @for (int j = 0; j < COL_B; j = j + 1) {
            B[i][j] = getfloat(); // 读取浮点数
        }
```

```
    }

    // 启动计时器，开始记录矩阵乘法的执行时间
    starttime();

    // 调用矩阵乘法函数
    matrix_multiply(A, B, C);

    // 停止计时器
    stoptime();

    // 输出结果矩阵 C
    putf("矩阵乘法结果为: \n");
    @for (int i = 0; i < ROW_A; i = i + 1) {
        @for (int j = 0; j < COL_B; j = j + 1) {
            putf("%f ", C[i][j]); // 使用标准的浮点数输出格式
            if (j < COL_B - 1) {
                putch(32); // 每个元素之间用空格隔开
            } else {
                putch(10); // 每行结束后换行
            }
        }
    }

    return 0;
}
```

（五） 矩阵乘法 LLVM IR 实现

```
@ROW_A = constant i32 2
@COL_A = constant i32 3
@ROW_B = constant i32 3
@COL_B = constant i32 2

declare void @putf(i8*, ...)
declare void @putch(i32)
declare double @getfloat()
declare void @starttime()
declare void @stoptime()

define void @matrix_multiply(double* %A, double* %B, double* %C) {
entry:
    %i = alloca i32, align 4
    %j = alloca i32, align 4
    %k = alloca i32, align 4
    store i32 0, i32* %i, align 4
    br label %loop1
```

```
19
20 loop1:
21     %0 = load i32, i32* %i, align 4
22     %cmp = icmp slt i32 %0, 2
23     br i1 %cmp, label %loop2, label %endloop1
24
25 loop2:
26     store i32 0, i32* %j, align 4
27     br label %loop3
28
29 loop3:
30     %1 = load i32, i32* %j, align 4
31     %cmp1 = icmp slt i32 %1, 2
32     br i1 %cmp1, label %loop4, label %endloop2
33
34 loop4:
35     %2 = load i32, i32* %i, align 4
36     %3 = load i32, i32* %j, align 4
37     %idx = mul i32 %2, 2
38     %idx1 = add i32 %idx, %3
39     %ptr = getelementptr inbounds double, double* %C, i32 %idx1
40     store double 0.0, double* %ptr, align 8
41     %4 = load i32, i32* %j, align 4
42     %inc = add i32 %4, 1
43     store i32 %inc, i32* %j, align 4
44     br label %loop3
45
46 endloop2:
47     %5 = load i32, i32* %i, align 4
48     %inc2 = add i32 %5, 1
49     store i32 %inc2, i32* %i, align 4
50     br label %loop1
51
52 endloop1:
53     store i32 0, i32* %i, align 4
54     br label %loop5
55
56 loop5:
57     %6 = load i32, i32* %i, align 4
58     %cmp3 = icmp slt i32 %6, 2
59     br i1 %cmp3, label %loop6, label %endloop5
60
61 loop6:
62     store i32 0, i32* %j, align 4
63     br label %loop7
64
65 loop7:
66     %7 = load i32, i32* %j, align 4
```

```
    %cmp4 = icmp slt i32 %7, 2
    br i1 %cmp4, label %loop8, label %endloop6

loop8:
    store i32 0, i32* %k, align 4
    br label %loop9

loop9:
    %8 = load i32, i32* %k, align 4
    %cmp5 = icmp slt i32 %8, 3
    br i1 %cmp5, label %loop10, label %endloop7

loop10:
    %9 = load i32, i32* %i, align 4
    %10 = load i32, i32* %k, align 4
    %idx2 = mul i32 %9, 3
    %idx3 = add i32 %idx2, %10
    %ptr1 = getelementptr inbounds double, double* %A, i32 %idx3
    %val = load double, double* %ptr1, align 8
    %11 = load i32, i32* %k, align 4
    %12 = load i32, i32* %j, align 4
    %idx4 = mul i32 %11, 2
    %idx5 = add i32 %idx4, %12
    %ptr2 = getelementptr inbounds double, double* %B, i32 %idx5
    %val1 = load double, double* %ptr2, align 8
    %mul = fmul double %val, %val1
    %13 = load i32, i32* %i, align 4
    %14 = load i32, i32* %j, align 4
    %idx6 = mul i32 %13, 2
    %idx7 = add i32 %idx6, %14
    %ptr3 = getelementptr inbounds double, double* %C, i32 %idx7
    %val2 = load double, double* %ptr3, align 8
    %add = fadd double %val2, %mul
    store double %add, double* %ptr3, align 8
    %15 = load i32, i32* %k, align 4
    %inc3 = add i32 %15, 1
    store i32 %inc3, i32* %k, align 4
    br label %loop9

endloop7:
    %16 = load i32, i32* %j, align 4
    %inc4 = add i32 %16, 1
    store i32 %inc4, i32* %j, align 4
    br label %loop7

endloop6:
    %17 = load i32, i32* %i, align 4
    %inc5 = add i32 %17, 1
```

```
115     store i32 %inc5, i32* %i, align 4
116     br label %loop5
117
118 endloop5:
119     ret void
120 }
121
122 define i32 @main() {
123 entry:
124     %A = alloca [6 x double], align 16
125     %B = alloca [6 x double], align 16
126     %C = alloca [4 x double], align 16
127     %i = alloca i32, align 4
128     %j = alloca i32, align 4
129
130     call void (i8*, ...) @putf(i8* getelementptr inbounds ([51 x i8], [51 x
            i8]* @.str, i32 0, i32 0), i32 6)
131     store i32 0, i32* %i, align 4
132     br label %loop11
133
134 loop11:
135     %0 = load i32, i32* %i, align 4
136     %cmp = icmp slt i32 %0, 2
137     br i1 %cmp, label %loop12, label %endloop11
138
139 loop12:
140     store i32 0, i32* %j, align 4
141     br label %loop13
142
143 loop13:
144     %1 = load i32, i32* %j, align 4
145     %cmp1 = icmp slt i32 %1, 3
146     br i1 %cmp1, label %loop14, label %endloop12
147
148 loop14:
149     %2 = load i32, i32* %i, align 4
150     %3 = load i32, i32* %j, align 4
151     %idx = mul i32 %2, 3
152     %idx1 = add i32 %idx, %3
153     %ptr = getelementptr inbounds double, double* %A, i32 %idx1
154     %val = call double @getfloat()
155     store double %val, double* %ptr, align 8
156     %4 = load i32, i32* %j, align 4
157     %inc = add i32 %4, 1
158     store i32 %inc, i32* %j, align 4
159     br label %loop13
160
161 endloop12:
```

```
162     %5 = load i32, i32* %i, align 4
163     %inc2 = add i32 %5, 1
164     store i32 %inc2, i32* %i, align 4
165     br label %loop11
166 
167 endloop11:
168     call void (i8*, ...) @putf(i8* getelementptr inbounds ([51 x i8], [51 x
            i8]* @.str1, i32 0, i32 0), i32 6)
169     store i32 0, i32* %i, align 4
170     br label %loop15
171 
172 loop15:
173     %6 = load i32, i32* %i, align 4
174     %cmp3 = icmp slt i32 %6, 3
175     br i1 %cmp3, label %loop16, label %endloop15
176 
177 loop16:
178     store i32 0, i32* %j, align 4
179     br label %loop17
180 
181 loop17:
182     %7 = load i32, i32* %j, align 4
183     %cmp4 = icmp slt i32 %7, 2
184     br i1 %cmp4, label %loop18, label %endloop16
185 
186 loop18:
187     %8 = load i32, i32* %i, align 4
188     %9 = load i32, i32* %j, align 4
189     %idx2 = mul i32 %8, 2
190     %idx3 = add i32 %idx2, %9
191     %ptr1 = getelementptr inbounds double, double* %B, i32 %idx3
192     %val1 = call double @getfloat()
193     store double %val1, double* %ptr1, align 8
194     %10 = load i32, i32* %j, align 4
195     %inc3 = add i32 %10, 1
196     store i32 %inc3, i32* %j, align 4
197     br label %loop17
198 
199 endloop16:
200     %11 = load i32, i32* %i, align 4
201     %inc4 = add i32 %11, 1
202     store i32 %inc4, i32* %i, align 4
203     br label %loop15
204 
205 endloop15:
206     call void @starttime()
207     %12 = getelementptr inbounds [6 x double], [6 x double]* %A, i32 0, i32 0
208     %13 = getelementptr inbounds [6 x double], [6 x double]* %B, i32 0, i32 0
```

```
209     %14 = getelementptr inbounds [4 x double], [4 x double]* %C, i32 0, i32 0
210     call void @matrix_multiply(double* %12, double* %13, double* %14)
211     call void @stoptime()
212     call void (i8*, ...) @putf(i8* getelementptr inbounds ([26 x i8], [26 x
            i8]* @.str2, i32 0, i32 0))
213     store i32 0, i32* %i, align 4
214     br label %loop19
215
216 loop19:
217     %15 = load i32, i32* %i, align 4
218     %cmp5 = icmp slt i32 %15, 2
219     br i1 %cmp5, label %loop20, label %endloop19
220
221 loop20:
222     store i32 0, i32* %j, align 4
223     br label %loop21
224
225 loop21:
226     %16 = load i32, i32* %j, align 4
227     %cmp6 = icmp slt i32 %16, 2
228     br i1 %cmp6, label %loop22, label %endloop20
229
230 loop22:
231     %17 = load i32, i32* %i, align 4
232     %18 = load i32, i32* %j, align 4
233     %idx4 = mul i32 %17, 2
234     %idx5 = add i32 %idx4, %18
235     %ptr2 = getelementptr inbounds double, double* %C, i32 %idx5
236     %val2 = load double, double* %ptr2, align 8
237     call void (i8*, ...) @putf(i8* getelementptr inbounds ([4 x i8], [4 x i8
            ]* @.str3, i32 0, i32 0), double %val2)
238     %19 = load i32, i32* %j, align 4
239     %cmp7 = icmp slt i32 %19, 1
240     br i1 %cmp7, label %cont, label %newline
241
242 cont:
243     call void @putch(i32 32)
244     br label %loop23
245
246 newline:
247     call void @putch(i32 10)
248     br label %loop23
249
250 loop23:
251     %20 = load i32, i32* %j, align 4
252     %inc5 = add i32 %20, 1
253     store i32 %inc5, i32* %j, align 4
254     br label %loop21
```

```
255
256 endloop20:
257     %21 = load i32, i32* %i, align 4
258     %inc6 = add i32 %21, 1
259     store i32 %inc6, i32* %i, align 4
260     br label %loop19
261
262 endloop19:
263     ret i32 0
264 }
265
266 @.str = private unnamed_addr constant [51 x i8] c"请输入矩阵 A 的元素, 共 %d
    个元素: \0A\00", align 1
267 @.str1 = private unnamed_addr constant [51 x i8] c"请输入矩阵 B 的元素, 共 %d
     个元素: \0A\00", align 1
268 @.str2 = private unnamed_addr constant [26 x i8] c"矩阵乘法结果为: \0A\00",
    align 1
269 @.str3 = private unnamed_addr constant [4 x i8] c"%f \00", align 1
```

(六) 矩阵乘法汇编实现

Listing 5: 矩阵乘法的 RISC-V 汇编实现

```
1  .option nopic
2  .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
3  .attribute unaligned_access, 0
4  .attribute stack_align, 16
5  .text
6  .align  1
7  .globl  matrix_multiply
8  .type   matrix_multiply, @function
9  # 定义矩阵A和B的尺寸
10 .globl  COL_B
11 .globl  ROW_B
12 .globl  COL_A
13 .globl  ROW_A
14
15 .section    .srodata, "a"
16 .align  2
17
18 .type   COL_A, @object
19 .size   COL_A, 4
20 COL_A: .word    3
21
22 .type   ROW_A, @object
23 .size   ROW_A, 4
24 ROW_A: .word    2
25
```

```
.type   COL_B, @object
.size   COL_B, 4
COL_B:.word    2

.type   ROW_B, @object
.size   ROW_B, 4
ROW_B:.word    3


#矩阵乘法
matrix_multiply:
    # 初始化目标矩阵C的前4个元素为0
    sw   zero, 0(a2)            # C[0][0] = 0
    sw   zero, 4(a2)            # C[0][1] = 0
    sw   zero, 8(a2)            # C[1][0] = 0
    sw   zero, 12(a2)           # C[1][1] = 0
    #用来在对C矩阵按行遍历时作为停止标志
    addi    a7, a2, 16          # 设置a7指向C矩阵的末尾 (a2 + 16)

    addi    a6, a1, 28          # 设置a6指向B矩阵的末尾 (a1 + 28)

#外层循环, 用于处理矩阵的行
Matrix_Row_Loop:
    mv  a4, a1                  # 将a4设置为B矩阵的起始地址
    mv  a5, a0                  # 将a5设置为A矩阵的起始地址
    addi    a3, a0, 12          # 设置a3指向A矩阵的第一行末尾

#内层循环, 用于处理矩阵C第一列
Matrix_Col_Loop:
    flw  fa5, 0(a4)             # 加载B的元素到fa5
    flw  fa3, 0(a5)             # 加载A的元素到fa3
    flw  fa4, 0(a2)             # 加载C的当前元素到fa4
    addi    a5, a5, 4           # A矩阵元素指针加4,矩阵A按行遍历
    addi    a4, a4, 8           # B矩阵元素指针加8,矩阵B按列遍历,由于每行有两个
        元素, 所以是加8
    fmadd.s fa5, fa5, fa3, fa4  # 计算: fa5 = fa5 * fa3 + fa4
    fsw  fa5, 0(a2)             # 保存结果到C矩阵
    bne a5, a3, Matrix_Col_Loop  # 如果a5 != a3, 继续列循环

    addi    a4, a1, 4           # 设置a4指向B矩阵的下一列
    addi    a5, a0, 0           # 设置a5指向A矩阵的第一列
#内层循环, 用于处理矩阵C第二列
Next_Column_Loop:
    flw  fa5, 0(a5)             # 加载A的元素到fa5
    flw  fa3, 0(a4)             # 加载B的元素到fa3
    flw  fa4, 4(a2)             # 加载C的当前元素到fa4
    addi    a4, a4, 8           # B矩阵元素指针加8
    addi    a5, a5, 4           # A矩阵元素指针加4
```

```
    fmadd.s fa5, fa5, fa3, fa4  # 计算: fa5 = fa5 * fa3 + fa4
    fsw fa5, 4(a2)              # 保存结果到C矩阵
    bne a6, a4, Next_Column_Loop  # 如果a4 != a6, 继续行循环

    addi    a2, a2, 8           # C矩阵指针加8, 处理下一行
    beq a2, a7, Matrix_Done     # 如果a2 == a7, 结束矩阵计算
    mv  a0, a3                  # 重置A矩阵指针到第二行
    j   Matrix_Row_Loop          # 跳转回到 Matrix_Row_Loop 继续处理下一行

Matrix_Done:
    ret                         # 返回函数


# 常量字符串部分

.align  3
.inputA:.string "请输入矩阵A的元素, 共%d个元素: \n"
.align  3
.inA:.string "%f"
.align  3
.inputB:.string "请输入矩阵B的元素, 共%d个元素: \n"
.align  3
.inB:.string "%f "
.align  3
.outputC:.string "矩阵乘法结果为: "
.align  3
.time_message:.string "运算耗费时间: %ld微秒\n"


# 主函数 main 部分

.align  1
.globl  main
.type   main, @function

main:


    addi    sp, sp, -96         # 为栈帧分配96字节
    # 保存现场
    sd      ra, 88(sp)          # 保存返回地址ra到栈中
    sd      s0, 80(sp)          # 保存s0到栈中
    sd      s1, 72(sp)          # 保存s1到栈中
    sd      s2, 64(sp)          # 保存s2到栈中

    # 初始化printf参数
    li      a1, 6               # 常量6
    lui     a0, %hi(.inputA)        # 加载inputA字符串的高位部分到a0
```

```
121     addi    a0, a0, %lo(.inputA)    # 加载inputA字符串的低位部分到a0
122     call    printf                  # 调用printf输出inputA字符串
123
124
125
126     # 准备其他操作
127     lui     s0, %hi(.inA)           # 加载inA字符串的高位部分到s0
128
129
130     # 多次调用scanf获取输入(A:sp+16–sp+36)
131     addi    a1, sp, 16              # 设置a1为sp + 16, 存储第一个输入值
132     addi    a0, s0, %lo(.inA)       # 设置a0为格式化字符串"%f"
133     call    scanf       # 调用scanf
134
135     addi    a1, sp, 20
136     addi    a0, s0, %lo(.inA)
137     call    scanf
138
139     addi    a1, sp, 24
140     addi    a0, s0, %lo(.inA)
141     call    scanf
142
143     addi    a1, sp, 28
144     addi    a0, s0, %lo(.inA)
145     call    scanf
146
147     addi    a1, sp, 32
148     addi    a0, s0, %lo(.inA)
149     call    scanf
150
151     addi    a1, sp, 36
152     addi    a0, s0, %lo(.inA)
153     call    scanf
154
155     li  a1, 6                       # 常量6
156     lui a0, %hi(.inputB)              # 加载inputB字符串的高位部分到a0
157     addi    a0, a0, %lo(.inputB)    # 加载inputB字符串的低位部分到a0
158     call    printf                  # 调用printf输出inputB字符串
159
160     addi    s1, sp, 40              # 设置s1为sp + 40
161
162     mv  s2, s1                      # s2指向s1
163
164     # 多次调用scanf获取输入(B:sp+40–sp+60)
165     mv  a1, s1                      # a1设置为s1
166     addi    a0, s0, %lo(.inA)       # a0设置为格式化字符串
167     call    scanf       # 调用scanf
168
```

```
    addi    a1, s1, 4           # 处理下一次输入
    addi    a0, s0, %lo(.inA)
    call    scanf

    addi    a1, s1, 8           # 处理下一次输入
    addi    a0, s0, %lo(.inA)
    call    scanf

    addi    a1, s1, 12          # 处理下一次输入
    addi    a0, s0, %lo(.inA)
    call    scanf

    addi    a1, s1, 16          # 处理下一次输入
    addi    a0, s0, %lo(.inA)
    call    scanf

    addi    a1, s1, 20          # 处理下一次输入
    addi    a0, s0, %lo(.inA)
    call    scanf

    call    clock               # 第一次调用 clock, 开始计时
    mv      s1, a0              # 将 clock 的返回值 (开始时间) 保存在 s1

    #a0:A(sp+16-sp+36) a1:B(sp+40-sp+60) a2:C(sp-sp+12)
    addi    a0, sp, 16          # 设置a0为sp + 16
    mv  a2, sp                  # 设置a2为sp, 指向输入数据
    mv  a1, s2                  # 设置a1为s2

    call    matrix_multiply     # 调用矩阵乘法函数

    call    clock               # 第二次调用 clock, 结束计时
    mv      s2, a0              # 将 clock 的返回值 (结束时间) 保存在 s2

    # 计算耗时
    sub s2, s2, s1              # 计算时间差 s2 = s2 - s1

    lui a0, %hi(.time_message)  # 加载时间信息字符串
    addi a0, a0, %lo(.time_message)
    mv a1, s2                   # 将计算的周期数传入a1
    call printf                 # 输出计时结果

    lui a0, %hi(.outputC)           # 加载outputC字符串的高位部分到a0
    addi    a0, a0, %lo(.outputC)   # 加载outputC字符串的低位部分到a0
    call    puts                # 输出字符串outputC

    # 打印矩阵C的结果
    flw fa5, 0(sp)              # 加载C[0][0]
    lui s0, %hi(.inB)
```

```
217     addi    a0, s0, %lo(.inB)     # 设置格式化字符串
218     fcvt.d.s fa5, fa5             # 将float转换为double
219     fmv.x.d a1, fa5               # 将fa5移动到a1
220     call    printf                # 打印C[0][0]
221
222     li  a0, 32                    # 打印空格
223     call    putchar
224
225     flw fa5, 4(sp)                # 加载C[0][1]
226     addi    a0, s0, %lo(.inB)     # 设置格式化字符串
227     fcvt.d.s fa5, fa5
228     fmv.x.d a1, fa5
229     call    printf                # 打印C[0][1]
230
231     li  a0, 10                    # 打印换行符
232     call    putchar
233
234     flw fa5, 8(sp)                # 加载C[1][0]
235     addi    a0, s0, %lo(.inB)
236     fcvt.d.s fa5, fa5
237     fmv.x.d a1, fa5
238     call    printf                # 打印C[1][0]
239
240     li  a0, 32
241     call    putchar
242
243     flw fa5, 12(sp)               # 加载C[1][1]
244     addi    a0, s0, %lo(.inB)
245     fcvt.d.s fa5, fa5
246     fmv.x.d a1, fa5
247     call    printf                # 打印C[1][1]
248
249     li  a0, 10                    # 打印换行符
250     call    putchar
251
252     # 恢复寄存器
253     ld  ra, 88(sp)
254     ld  s0, 80(sp)
255     ld  s1, 72(sp)
256     ld  s2, 64(sp)
257
258     li  a0, 0                     # 返回0
259     addi    sp, sp, 96            # 恢复栈指针
260     jr  ra                        # 返回
```

（七） 中间代码生成各阶段的.dot 文件

后缀	描述
adjust_alignment	调整内存对齐信息。
alias	别名分析，检查不同指针或变量是否指向相同的内存地址。
alignments	变量和数据结构的内存对齐信息。
asmcons	汇编代码约束信息。
backprop	反向传播，通常与数据流分析或值传播有关。
barriers	程序中的屏障信息，可能与指令排序或优化有关。
bbpart	基本块分割信息，优化基本块时的中间结果。
bbro	基本块重排序，优化块执行顺序。
bswap	字节交换优化，特别是在处理大端/小端系统时。
ccp1, ccp2, ccp3, ccp4	条件常量传播阶段，1 到 4 表示不同的传播阶段。
cdce	条件死代码消除，移除条件语句中无用的代码。
cddce1, cddce2, cddce3	条件数据依赖死代码消除的不同阶段。
ce1, ce2, ce3	常量传播和常量表达式优化的不同阶段。
cfg	控制流图（Control Flow Graph）的生成及修改信息。
ch2	可能与循环优化有关，具体是第二个阶段。
ch_vect	向量化的循环优化信息。
cmpelim	比较消除优化，移除不必要的比较操作。
combine	组合优化阶段，将指令组合以减少指令数量。
compgotos	压缩 goto 语句。
copyprop1, copyprop2, copyprop3	复制传播优化，消除多余的复制操作。
cplxlower1	复数运算的优化和降级。
cprop1, cprop2, cprop3	常量传播的多个阶段。
cprop_hardreg	硬件寄存器常量传播。
crited1	关键路径编辑优化，减少关键路径上的指令数。
csa	条件语句优化。
cse1, cse2	公共子表达式消除（Common Subexpression Elimination），第 1 和第 2 阶段。
cselim	消除常量选择运算（例如 select 语句）的优化。
cse_local	局部公共子表达式消除。
cunroll	循环展开优化。
cunrolli	循环展开的指令级优化。
dce2, dce3, dce4, dce6, dce7	死代码消除（Dead Code Elimination）的多个阶段。
debug	调试信息。
dfinish	数据流分析结束信息。
dfinit	数据流分析初始化信息。
dom2, dom3	支配树（dominator tree）分析的不同阶段。
dot	控制流图以 DOT 格式输出。

dse1, dse2, dse3, dse5	存储删除优化（Dead Store Elimination）的不同阶段。
dwarf2	DWARF 调试信息。
ealias	扩展的别名分析。
earlydebug	早期调试信息。
early_objsz	早期对象大小计算。
early_optimizations	早期的编译优化。
eh	异常处理信息。
einline	早期函数内联信息。
endbr_and_patchable_area	与特定 CPU 架构相关的安全性和指令补丁。
esra	扩展的存储器重命名优化。
ethread	线程优化相关信息。
evrp	增强的值范围传播（Enhanced Value Range Propagation）。
expand	中间表示扩展为 RTL（Register Transfer Level）的阶段。
fab1	函数分析块信息。
final	最终生成的汇编代码阶段。
fix_loops	循环修正优化。
fixup_cfg1, fixup_cfg2, fixup_cfg3	修正控制流图的不同阶段。
fnsplit	函数拆分优化。
forwprop1, forwprop2, forwprop3, forwprop4	前向传播优化的多个阶段。
fre1, fre3, fre5	完全冗余消除（Full Redundancy Elimination）的不同阶段。
fwprop1, fwprop2	前向复制传播的不同阶段。
gimple	GIMPLE 中间表示的转换。
ifcombine	条件语句组合优化。
ifcvt	条件转换优化。
iftoswitch	条件语句转换为 switch 语句的优化。
init-regs	寄存器初始化。
into_cfglayout	进入控制流图布局阶段。
ira	集成寄存器分配（Integrated Register Allocation）。
isel	指令选择优化。
isolate-paths	路径隔离优化。
ivcanon	循环增量变量规范化。
ivopts	循环增量变量优化。
jump, jump2, jump_after_combine	跳转优化的不同阶段。
laddress	地址计算优化。
ldist	负载分发优化。
lim2, lim4	循环不变代码移动（Loop Invariant Code Motion）的不同阶段。

local-fnsummary1, local-fnsummary2	局部函数分析的摘要信息。
local-pure-const1, local-pure-const2	局部纯函数和常量分析。
loop, loop2, loopdone, loopinit, loop2_done, loop2_init, loop2_invariant, loop2_unroll	循环优化的不同阶段。
lower	降级优化阶段，将高层语义降低到更简单的形式。
mach	目标机器相关的优化信息。
mergephi1, mergephi2, mergephi3	合并 Phi 函数（SSA 中的一种形式）的不同阶段。
mode_sw	模式切换优化。
modref1, modref2	修改和引用分析。
nothrow	非抛出异常的优化信息。
nrv	返回值优化。
objsz1	对象大小分析。
ompexp, omplower	OpenMP 并行扩展和降低优化。
optimized	经过优化的文件信息。
original	原始文件信息。
outof_cfglayout	控制流图布局的退出阶段。
pcom	预计算优化。
peephole2	窥孔优化（Peephole Optimization）的第二阶段。
phiopt1, phiopt2, phiopt3, phiopt4	Phi 优化的多个阶段。
phiprop	Phi 传播优化。
post_ipa_warn1	跨过程分析后的警告信息。
postreload	重新装载后的优化信息。
powcabs	幂函数和绝对值函数的优化。
pre	部分冗余消除(Partial Redundancy Elimination)。
pro_and_epilogue	函数的前序和后序代码生成阶段。
profile_estimate	性能估算信息。
reassoc1, reassoc2	重新关联优化的不同阶段。
ree	重新评估表达式的优化。
reginfo	寄存器分配信息。
release_ssa	释放 SSA（Static Single Assignment）形式的信息。
reload	寄存器重新加载的阶段。
retslot	返回值槽优化。
rtl_dce	RTL 级别的死代码消除。
sccp	稀疏条件常量传播。
sched2	调度优化的第二阶段。
shorten	缩短分支指令。
sincos	正弦余弦优化。
sink1, sink2	代码下沉优化的不同阶段。
slp1	向量化优化中的超级单指令流（SLP）阶段。

slsr	符号性常量的线性关系替换（Symbolic Linear Substitution）。
split1, split2, split3	指令分割的不同阶段。
sra	结构重排优化（Structure Reordering Analysis）。
ssa	静态单一赋值（Static Single Assignment）形式的生成。
stack	栈布局和优化信息。
statistics	优化阶段的统计信息。
stdarg	可变参数优化。
store-merging	存储合并优化。
strlen1	字符串长度优化的第 1 阶段。
stv1, stv2	向量化存储优化的不同阶段。
subreg1, subreg3	子寄存器优化的不同阶段。
switchconv	switch 语句优化。
switchlower1	switch 语句降低优化的第 1 阶段。
tailc	尾递归优化。
tailr1, tailr2	尾递归消除的不同阶段。
thread1, thread2	线程优化的第 1 和第 2 阶段。
threadfull1, threadfull2	完整的线程优化阶段。
ud_dce	未定义行为的死代码消除。
uncprop1	未使用代码传播优化的第 1 阶段。
veclower21	向量化降低优化。
vect	向量化优化。
vregs	虚拟寄存器信息。
vrp1, vrp2	值范围传播（Value Range Propagation）的第 1 和第 2 阶段。
waccess1, waccess2, waccess3	写访问优化的不同阶段。
walloca1, walloca2	alloca 相关优化的不同阶段。
widening_mul	宽乘法优化。
wrestrict	restrict 关键字相关的优化。
zero_call_used_regs	消除未使用的调用寄存器。

项目地址

参考文献