



南开大学
Nankai University

南 开 大 学

网 络 空 间 安 全 学 院

编译原理实验报告

预备工作——了解你的编译器 & LLVM LR 编程 & 汇编编程

阿斯雅 - 2210737

胡博浩 - 2212998

年级：2022 级

专业：信息安全

指导教师：王刚

2024 年 9 月 23 日

摘要

编译器是一种将高级编程语言编写的源代码转换成计算机能够理解和执行的低级机器代码的计算机程序。这一转换过程涉及多个阶段，包括预处理、词法分析、语法分析、语义分析、中间代码生成、优化以及代码生成等关键步骤。LLVM 作为一个先进的编译器和工具链技术，提供了一套模块化的编译器基础设施，它能够支持编译器前端和后端的开发。LLVM 的核心是中间表示（IR），这是一种低级别的编程语言，它为不同的目标指令集提供了一个抽象层。

与此同时，汇编语言作为一种更为接近硬件的编程语言，它与机器语言指令之间存在着直接的对应关系，尽管它比机器指令更易于人类理解和编写。针对不同的指令集架构，有不同的汇编语言；例如，RISC-V 64 位汇编语言，它属于一种精简指令集，特别适用于那些对性能要求极高的应用场景或需要直接进行硬件操作的场合。汇编语言编程要求程序员直接操作寄存器、内存地址以及指令集，这通常需要对硬件架构有深入的了解和掌握。通过汇编语言，程序员能够精细控制硬件，优化程序性能，尽管这通常以牺牲编程便利性和可移植性为代价。

关键字：编译器、LLVM IR、汇编、RISCV64

目录

一、 准备工作	1
(一) 搭建环境	1
(二) 验证环境	1
二、 实验过程	2
(一) 小组分工	2
(二) 问题一	2
1. 预处理	3
2. 编译器	6
3. 汇编器	18
4. 链接器和加载器	20
(三) 问题二	22
1. LLVM IR 程序简述	22
2. 具体代码	24
3. 运行和结果分析	31
4. 语言特性具体分析	33
5. LLVM IR 和 SysY 对比	35
(四) 问题三	37
1. 简单计算器 riscv64 汇编程序	37
2. 数组求和 riscv64 汇编程序	41
三、 实验总结	44

一、准备工作

(一) 搭建环境

根据实验指导书，在 Ubuntu22.04 虚拟机中搭建 Riscv64 交叉编译器环境及 qemu 环境。

```
asy@asy-virtual-machine:~/Desktop$ sudo su
[sudo] password for asy:
root@asy-virtual-machine:/home/asy/Desktop# export PATH=$PATH:/opt/qemu/bin:$PATH
root@asy-virtual-machine:/home/asy/Desktop# qemu-system-riscv64 --version
QEMU emulator version 7.0.0
Copyright (c) 2003-2022 Fabrice Bellard and the QEMU Project developers
root@asy-virtual-machine:/home/asy/Desktop#
```

图 1: riscv64 交叉编译器环境

```
asy@asy-virtual-machine:~/Desktop$ sudo su
[sudo] password for asy:
root@asy-virtual-machine:/home/asy/Desktop# export PATH=riscv64/bin:$PATH
root@asy-virtual-machine:/home/asy/Desktop# riscv64-unknown-linux-gnu-gcc --version
riscv64-unknown-linux-gnu-gcc () 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
root@asy-virtual-machine:/home/asy/Desktop#
```

图 2: qemu-riscv64 环境

(二) 验证环境

根据实验指导书，我们可以写一个 C 语言小程序，然后通过交叉编译器编译并在 qemu 环境中运行目标程序来验证我们搭建的环境。本次验证实验选用的是实验指导书上的例子，如下面所示。

C 语言小程序

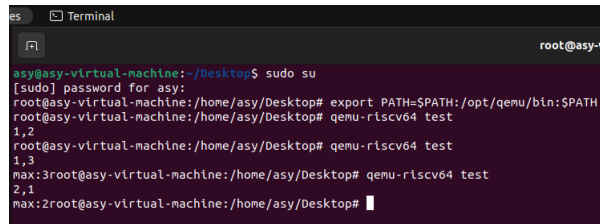
```
1  #include<stdio.h>
2  int a = 0;int b = 0;
3  int max(int a, int b) {
4      if(a >= b) {
5          return a;}
6      else {
7          return b;}
8      }
9  int main() {
10     scanf("%d,%d", &a,&b);
11     printf("max:%d", max(a, b));
12     return 0;
13 }
```

首先第一步，我们要通过 riscv64 交叉编译器来编译该代码，获得目标程序。如下图所示。

```
es  Terminal  9月 16 09:01
root@asy-virtual-machine:/home/asy/Desktop
asy@asy-virtual-machine:~/Desktop$ sudo su
[sudo] password for asy:
root@asy-virtual-machine:/home/asy/Desktop# export PATH=riscv64/bin:$PATH
root@asy-virtual-machine:/home/asy/Desktop# riscv64-unknown-linux-gnu-gcc asy.c -o test -static
root@asy-virtual-machine:/home/asy/Desktop#
```

图 3: riscv64 交叉编译器编译

之后第二步，我们使用 qemu-riscv64 仿真器运行该目标程序，可以从运行结果知道，我们的环境搭建的并没有问题。



```
es Terminal
root@asy-virtual-machine:~#
asy@asy-virtual-machine:~/Desktop$ sudo su
[sudo] password for asy:
root@asy-virtual-machine:~/Desktop# export PATH=$PATH:/opt/qemu/bin:$PATH
root@asy-virtual-machine:~/Desktop# qemu-riscv64 test
1,2
root@asy-virtual-machine:~/Desktop# qemu-riscv64 test
1,3
max:3root@asy-virtual-machine:~/Desktop# qemu-riscv64 test
2,1
max:2root@asy-virtual-machine:~/Desktop#
```

图 4: 运行结果

二、 实验过程

(一) 小组分工

在本次预备实验中，我们小组的小组分工为：对于问题一，小组中的两个人都选取阶乘程序并独立完成对编译器不同阶段的研究。而对于问题二和问题三，由胡博浩同学独立完成问题二的 LLVM LR 中间代码编写，由阿斯雅同学独立完成 riscv64 汇编代码编写并撰写相应的实验报告。实验报告中的其他部分如摘要，总结由小组两个人共同编写。并且本次实验参考了助教提供的实验指导书：[1]，[2]

(二) 问题一

如下图所示，一个完整的编译过程可以分为：

- 1、预处理
- 2、编译
- 3、汇编
- 4、链接加载



图 5: 完整编译过程

我以实验指导书中的阶乘的 main.c 代码为基准，研究了在不同编译阶段代码的变化以此来探究编译原理。

阶乘代码

```
1  #include<stdio.h>
2  int main() {
3      int i, n, f;
4      i = 2;
5      f = 1;
6      scanf("%d",&n);
7      while (i <= n) {
8          f = f * i;
9          i = i + 1;
10     }
11     printf("%d", f);
12     return 0;
13 }
```

1. 预处理

预处理阶段的任务

预处理阶段是编译过程中的初始步骤，其主要任务是对源代码进行一系列文本处理，以准备代码供编译器进一步编译。

在这一阶段，预处理器会解析并执行所有的预处理指令，比如将头文件的内容包含到源文件中，对宏进行展开，处理条件编译指令以根据特定条件决定哪些代码应该被编译，以及去除代码中的注释。此外，预处理器还会在每个源代码行前添加行号和文件名信息，以便在编译后的程序中进行调试时能够追踪到原始的源代码位置。

所谓的预处理指令是 C 和 C++ 语言中的一种特殊指令，它们以井号（#）开头，并由预处理器在编译过程的预处理阶段执行。常见的预处理指令包括：

- `#include`: 包含其他文件的内容，可以是系统头文件（用尖括号 `<>` 包围）或用户定义的头文件（用双引号 `""` 包围）。
- `#define`: 定义宏，可以是一个简单的文本替换，也可以是带有参数的宏，类似于函数。
- `#undef`: 取消已定义的宏。
- 条件编译: 允许根据特定条件包含或排除代码段，如 `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif` 等

验证结果

为了验证预处理器的功能，我们可以在原来的阶乘程序的基础上，通过增加宏定义、文件包含、条件编译、注释等方法修改程序并观察预处理后的结果。

以下是修改后的代码：

修改后的阶乘代码

```
1 #include <stdio.h> // 包含标准输入输出头文件
2
3 #define MAX_VALUE 100 // 宏定义
4 #define DEBUG 1      // 用于条件编译的宏定义
5
6 int main() {
7     int i, n, f;
8     i = 2;
9     f = 1;
10
11     // 如果DEBUG宏被定义，则打印调试信息
12     #if DEBUG
13         printf("Debug\n");
14     #endif
```

```

15
16     scanf("%d", &n); // 读取用户输入
17
18     // 循环计算阶乘
19     while (i <= n) {
20         f = f * i;
21         i = i + 1;
22     }
23
24     printf("Factorial of %d is %d\n", n, f); // 输出结果
25     return 0;
26 }
27 }

```

在 Ubuntu 中，通过添加参数-E 可以令 gcc 只进行预处理过程，因此执行以下命令即可得到预处理文件：

gcc 预处理

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

由于是在 wsl 里，直接使用本机的 visual studio 查看 main.i，如下图所示：

```

# 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
# 27 "/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
# 1 "/usr/include/features.h" 1 3 4
# 392 "/usr/include/features.h" 3 4
# 1 "/usr/include/features-time64.h" 1 3 4
# 20 "/usr/include/features-time64.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 21 "/usr/include/features-time64.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 1 3 4
# 19 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 20 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 2 3 4
# 22 "/usr/include/features-time64.h" 2 3 4
# 393 "/usr/include/features.h" 2 3 4
# 486 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 559 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 560 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 561 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 487 "/usr/include/features.h" 2 3 4
# 510 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 511 "/usr/include/features.h" 2 3 4
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4

```

图 6: main.i 开头

```
721 extern char *ctermid (char * _s) __attribute__ ((__nothrow__ , __leaf__))
722 __attribute__ ((__access__ (__write_only__ , 1)));
723 # 867 "/usr/include/stdio.h" 3 4
724 extern void flockfile (FILE *_stream) __attribute__ ((__nothrow__ , __leaf__));
725
726
727
728 extern int ftrylockfile (FILE *_stream) __attribute__ ((__nothrow__ , __leaf__));
729
730
731 extern void funlockfile (FILE *_stream) __attribute__ ((__nothrow__ , __leaf__));
732 # 885 "/usr/include/stdio.h" 3 4
733 extern int __uflow (FILE *);
734 extern int __overflow (FILE *, int);
735 # 902 "/usr/include/stdio.h" 3 4
736
737 # 2 "main.c" 2
738
739
740
741
742
743 # 6 "main.c"
744 int main() {
745     int i, n, f;
746     i = 2;
747     f = 1;
748
749
750     printf("Debug\n");
751
752
753     scanf("%d", &n);
754
755
756     while (i <= n) {
757         f = f * i;
758         i = i + 1;
759     }
760
761     printf("Factorial of %d is %d\n", n, f);
762     return 0;
763 }
764
765
```

图 7: main.i 结尾

观察预处理文件，可以发现文件长度远大于源文件。由于代码量过于庞大，这里我只展示了开头和结尾的内容。

仔细查看代码，程序对 `include` 指令会替换对应的头文件、导致 `stdio.h` 的内容被包含在预处理文件；所有宏定义将被其值替换；条件编译指令 `#if DEBUG` 将根据 `DEBUG` 宏的值决定是否包含死代码；所有注释都被删除，并添加了行号和文件名标识，很好地验证了预处理的上述功能。

2. 编译器

编译器的功能

编译器是一种软件，其主要功能是将程序员用高级编程语言编写的源代码转换成计算机可以直接执行的机器代码或汇编代码。这个过程涉及多个阶段，包括词法分析、语法分析、语义分析、中间代码生成、代码优化以及目标代码生成。编译器还会进行错误处理，确保源代码中的语法和语义错误得到检测和报告，同

时可能生成调试信息以支持程序的调试过程。此外，编译器可能包含性能分析工具，帮助开发者优化代码以提高程序的执行效率。整体而言，编译器的设计目标是确保源代码的准确转换，同时尽可能地提升生成代码的性能和可维护性。

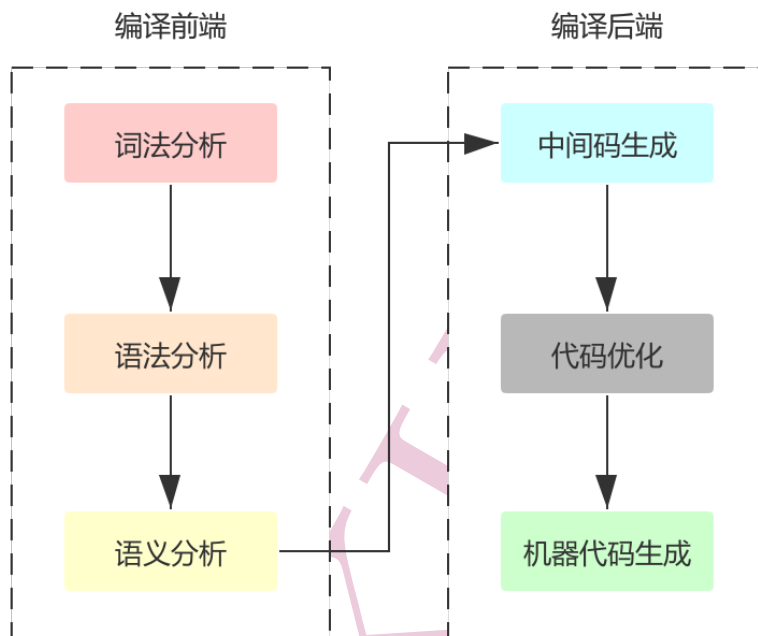


图 8: 编译器工作流程

1. 词法分析

词法分析是编译过程中的初始阶段，它将源程序中的字符序列转换成一系列的词法单元，这些词法单元是编程语言中的基本语法元素，如关键字、标识符、常数、运算符和分隔符等。分析器从源代码的左侧开始，逐个字符地读取输入，识别出每个单词符号，并将它们以二元组的形式输出，包含符号种类的编码和符号的值。词法分析器可以作为函数存在，供语法分析器调用，或者作为一个独立的程序运行，其任务是完成对源代码的词法分析，生成词法单元流供后续编译阶段使用。

在 Ubuntu 中使用 llvm，输入以下命令即可将源程序转换为单词序列

词法分析命令

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

我们尝试对预处理修改后的阶乘文件进行分词，结果如下：

```

huhao@XiaoXinPro:~/test$ clang -E -Xclang -dump-tokens main.c
typedef 'typedef' [StartOfLine] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stddef.h:46:1>
long 'long' [LeadingSpace] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stddef.h:46:9 <Spelling=<built-in>:92:23>
>
unsigned 'unsigned' [LeadingSpace] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stddef.h:46:9 <Spelling=<built-in>
>:92:28>>
int 'int' [LeadingSpace] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stddef.h:46:9 <Spelling=<built-in>:92:37>
>
identifier 'size_t' [LeadingSpace] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stddef.h:46:23>
semi ';' Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stddef.h:46:29>
typedef 'typedef' [StartOfLine] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stdarg.h:14:1>
identifier '__builtin_va_list' [LeadingSpace] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stdarg.h:14:9>
identifier 'va_list' [LeadingSpace] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stdarg.h:14:27>
semi ';' Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stdarg.h:14:34>
typedef 'typedef' [StartOfLine] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stdarg.h:32:1>
identifier '__builtin_va_list' [LeadingSpace] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stdarg.h:32:9>
identifier '__gnuc_va_list' [LeadingSpace] Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stdarg.h:32:27>
semi ';' Loc=</usr/lib/llvm-14/lib/clang/14.0.0/include/stdarg.h:32:41>
typedef 'typedef' [StartOfLine] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:1>
unsigned 'unsigned' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:9>
char 'char' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:18>
identifier '__u_char' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:23>
semi ';' Loc=</usr/include/x86_64-linux-gnu/bits/types.h:31:31>
typedef 'typedef' [StartOfLine] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:1>
unsigned 'unsigned' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:9>
short 'short' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:18>
int 'int' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:24>
identifier '__u_short' [LeadingSpace] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:28>
semi ';' Loc=</usr/include/x86_64-linux-gnu/bits/types.h:32:37>
typedef 'typedef' [StartOfLine] Loc=</usr/include/x86_64-linux-gnu/bits/types.h:33:1>

```

图 9: 部分词法分析结果

经过观察，每个词法单元由两部分组成：一个是表示词法单元类型的标识符，另一个是与该类型相对应的值。表明成功将源程序转换为了单词序列。

2. 语法分析

语法分析是编译过程中的第二个阶段。在语法分析阶段，编译器会将标记序列组织成具有明确语法结构的单元，如表达式、语句和声明的层次关系。这些结构通过语法树或抽象语法树来表示，其中语法树是直接表示编程语言语法规则的树状结构，而抽象语法树是对源代码结构的更高层次抽象。

语法分析器在构建语法树的过程中，会检查代码是否符合语法规则，并构建这个树状结构。这个过程涉及到递归下降分析、LL 解析器和 LR 解析器等技术。语法分析的正确性对于编译器的整体功能至关重要，因为它确保了源程序在结构上是正确的，为后续的编译阶段提供了可靠的基础。

在 LLVM 中，采用上下文无关文法来定义编程语言的语法规则，并使用递归下降分析器来解析源代码。递归下降分析是一种自顶向下的解析方法，它根据上下文无关文法规则，递归地解析源代码中的语法结构。每个文法规则对应一个解析函数，这些函数逐个解析源代码中的标记，并构建抽象语法树。

```

1 // 遇到终结符号 a 时:
2 if (当前输入符号 == a)
3   读入下一个输入符号
4 // 遇到非终结符号 A 时:
5 A()

```

```

6 // 遇到规则  $A \rightarrow s$  时:
7 if(当前输入符号  $\text{notin FOLLOW}(A)$ )
8 error()

```

具体来说，我们使用下面的命令，将源代码转换成词法单元，然后构建抽象语法树，最后打印出这个树状结构。

clang 语法分析

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

在 ubuntu 中使用上述命令，结果如下图所示：

```

huhao@XiaoXinPro:~/test$ clang -E -Xclang -ast-dump main.c
TranslationUnitDecl 0x1de28 <<invalid sloc>> <invalid sloc>
|-TypeDefDecl 0x1de2650 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
|   |-BuiltinType 0x1de23f0 '__int128'
|   |-TypeDefDecl 0x1de26c0 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
|   |   |-BuiltinType 0x1de2410 'unsigned __int128'
|   |-TypeDefDecl 0x1de29c8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
|   |   |-RecordType 0x1de27a0 'struct __NSConstantString_tag'
|   |   |-Record 0x1de2718 '__NSConstantString_tag'
|   |-TypeDefDecl 0x1de2a60 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
|   |   |-PointerType 0x1de2a20 'char *'
|   |   |-BuiltinType 0x1de1ed0 'char'
|   |-TypeDefDecl 0x1de2d58 <<invalid sloc>> <invalid sloc> implicit referenced __builtin_va_list 'struct __va_list_tag[1]'
|   |   |-ConstantArrayType 0x1de2d00 'struct __va_list_tag[1]' 1
|   |   |-RecordType 0x1de2b40 'struct __va_list_tag'
|   |   |-Record 0x1de2ab8 '__va_list_tag'
|   |-TypeDefDecl 0x1de2dc8 </usr/lib/llvm-14/lib/clang/14.0.0/include/stddef.h:46:1, col:23> col:23 referenced size_t 'unsi
gned long'
|   |   |-BuiltinType 0x1de1ff0 'unsigned long'
|   |-TypeDefDecl 0x1de6afc0 </usr/lib/llvm-14/lib/clang/14.0.0/include/stdarg.h:14:1, col:27> col:27 va_list '__builtin_va_l
ist': 'struct __va_list_tag[1]'
|   |   |-TypeDefType 0x1de6af90 '__builtin_va_list' sugar
|   |   |-TypeDef 0x1de2d58 '__builtin_va_list'
|   |   |   |-ConstantArrayType 0x1de2d00 'struct __va_list_tag[1]' 1
|   |   |   |-RecordType 0x1de2b40 'struct __va_list_tag'
|   |   |   |-Record 0x1de2ab8 '__va_list_tag'
|   |-TypeDefDecl 0x1de6b028 <line:3:1, col:27> col:27 referenced __gnuc_va_list '__builtin_va_list': 'struct __va_list_tag[1]'
|   |   |-TypeDefType 0x1de6af90 '__builtin_va_list' sugar
|   |   |-TypeDef 0x1de2d58 '__builtin_va_list'

```

图 10: 语法分析结果 1

```

huhao@XiaoXinPro: ~/test
|-DeclRefExpr 0x1ea7b00 <col:12> 'int' lvalue Var 0x1ea70d8 'i' 'int'
|-ImplicitCastExpr 0x1ea7b58 <col:17> 'int' <LValueToRValue>
|-DeclRefExpr 0x1ea7b20 <col:17> 'int' lvalue Var 0x1ea7158 'n' 'int'
|-CompoundStmt 0x1ea7d18 <col:20, line:22:5>
|   |-BinaryOperator 0x1ea7c40 <line:20:9, col:17> 'int' '='
|   |   |-DeclRefExpr 0x1ea7b90 <col:9> 'int' lvalue Var 0x1ea71d8 'f' 'int'
|   |   |-BinaryOperator 0x1ea7c20 <col:13, col:17> 'int' '*'
|   |   |   |-ImplicitCastExpr 0x1ea7bf0 <col:13> 'int' <LValueToRValue>
|   |   |   |   |-DeclRefExpr 0x1ea7bb0 <col:13> 'int' lvalue Var 0x1ea71d8 'f' 'int'
|   |   |   |   |-ImplicitCastExpr 0x1ea7c08 <col:17> 'int' <LValueToRValue>
|   |   |   |   |-DeclRefExpr 0x1ea7bd0 <col:17> 'int' lvalue Var 0x1ea70d8 'i' 'int'
|   |   |-BinaryOperator 0x1ea7cf8 <line:21:9, col:17> 'int' '='
|   |   |   |-DeclRefExpr 0x1ea7c60 <col:9> 'int' lvalue Var 0x1ea70d8 'i' 'int'
|   |   |   |-BinaryOperator 0x1ea7cd8 <col:13, col:17> 'int' '+'
|   |   |   |   |-ImplicitCastExpr 0x1ea7cc0 <col:13> 'int' <LValueToRValue>
|   |   |   |   |-DeclRefExpr 0x1ea7c80 <col:13> 'int' lvalue Var 0x1ea70d8 'i' 'int'
|   |   |   |-IntegerLiteral 0x1ea7ca0 <col:17> 'int' 1
|   |-CallExpr 0x1ea7e58 <line:24:5, col:43> 'int'
|   |   |-ImplicitCastExpr 0x1ea7e40 <col:5> 'int (*) (const char *, ...)' <FunctionToPointerDecay>
|   |   |   |-DeclRefExpr 0x1ea7d58 <col:5> 'int (const char *, ...)' Function 0x1e8c058 'printf' 'int (const char *, ...)'
|   |   |-ImplicitCastExpr 0x1ea7ea8 <col:12> 'const char *' <NoOp>
|   |   |   |-ImplicitCastExpr 0x1ea7e90 <col:12> 'char *' <ArrayToPointerDecay>
|   |   |   |   |-StringLiteral 0x1ea7db8 <col:12> 'char[23]' lvalue "Factorial of %d is %d\n"
|   |   |   |-ImplicitCastExpr 0x1ea7ec0 <col:39> 'int' <LValueToRValue>
|   |   |   |   |-DeclRefExpr 0x1ea7de8 <col:39> 'int' lvalue Var 0x1ea7158 'n' 'int'
|   |   |   |-ImplicitCastExpr 0x1ea7ed8 <col:42> 'int' <LValueToRValue>
|   |   |   |   |-DeclRefExpr 0x1ea7e08 <col:42> 'int' lvalue Var 0x1ea71d8 'f' 'int'
|   |-ReturnStmt 0x1ea7f10 <line:25:5, col:12>
|   |   |-IntegerLiteral 0x1ea7ef0 <col:12> 'int' 0
huhao@XiaoXinPro:~/test$

```

图 11: 语法分析结果 2

可以发现，语法分析阶段利用词法分析阶段的单词构成一棵语法分析树，各个节点按照语法规则排列，从而清晰地展现出代码的层次结构和逻辑关系。

3. 语义分析

语义分析是编译过程中的第三个阶段，它负责检查源代码是否符合编程语言的语义规则。在这个阶段，编译器会分析程序的结构，以确保每个操作和表达式不仅在语法上是正确的，而且在语义上也是合法的。语义分析涉及对程序中的变量、表达式、语句和函数进行详尽的检查，以验证它们是否按照语言规范正确使用。

语义分析会检查类型兼容性，确保赋值语句、函数调用和表达式中的操作数类型匹配，并且操作符适用于这些类型。此外，它还会验证变量在使用前是否已经被声明，并检查它们的作用域，以确认变量只在定义它们的作用域内被访问。类型推断是语义分析的一部分，它允许编译器自动确定某些表达式的类型，从而简化代码编写。

语义分析还会进行控制流分析，以检查程序的逻辑结构是否正确，比如确保每个循环都有终止条件，每个函数都有返回值，以及每个可能的执行路径都是有效的。此外，它还会验证异常处理是否恰当，以及程序中是否存在其他潜在的运行时错误。

总而言之，语义分析通过检查源代码是否符合语言的定义语义，来保证最终生成的程序能够正确执行。如果语义分析发现错误，编译器会报告语义错误，并可能停止编译过程，直到所有错误被修正。

4. 中间代码生成

完成上述步骤后，很多编译器会生成一个明确的低级或类机器语言的中间表示。在这一阶段，编译器将经过语法和语义检查的抽象语法树或语法树转换成一种与源语言和目标机器都无关的内部表示形式，这种形式被称为中间代码。中间代码具有结构简单、含义明确的特点，它简化了源程序中的复杂结构，使其更接近于机器语言，但又不失抽象性，这样的设计使得后续的代码优化和转换更为便捷。中间代码通常是一种低级的、平台无关的指令集，它能够被进一步转换成特定硬件平台上的机器指令。由于中间代码的通用性，它为编译器的设计带来了极

大的灵活性，允许同一编译器前端处理多种编程语言，而后端则可以将中间代码转换成不同硬件架构上的目标代码，从而实现跨平台的编译能力。

在 llvm 中，我们可以通过下面的命令生成 LLVM IR：

clang 中间代码生成

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

结果如下图所示，可以观察到生成的.ll 文件中已经对死代码进行了删除，这表明在代码优化的正式步骤之前，编译器就已经开始执行死代码的优化工作：

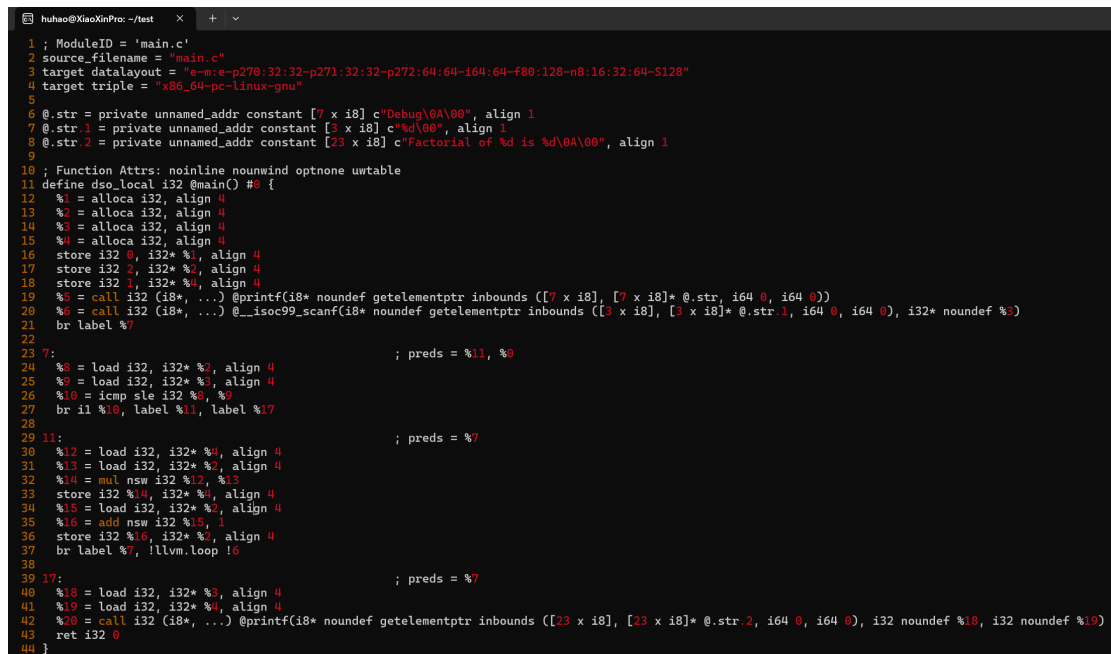


图 12: llvm 中间代码 1

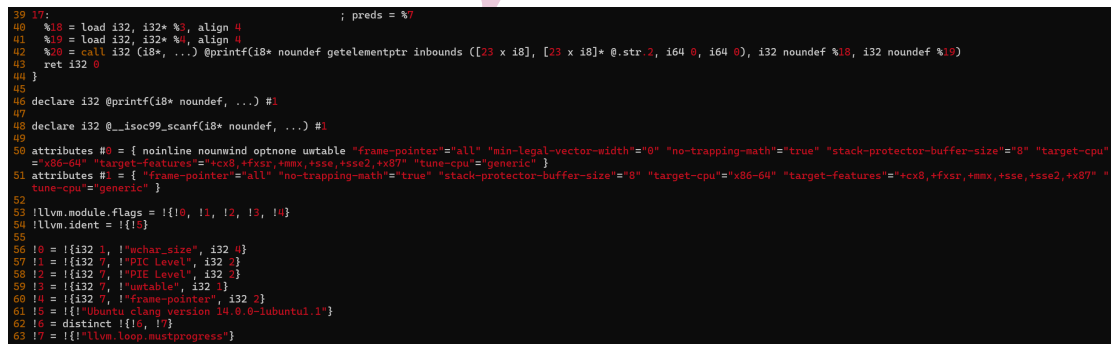


图 13: llvm 中间代码 2

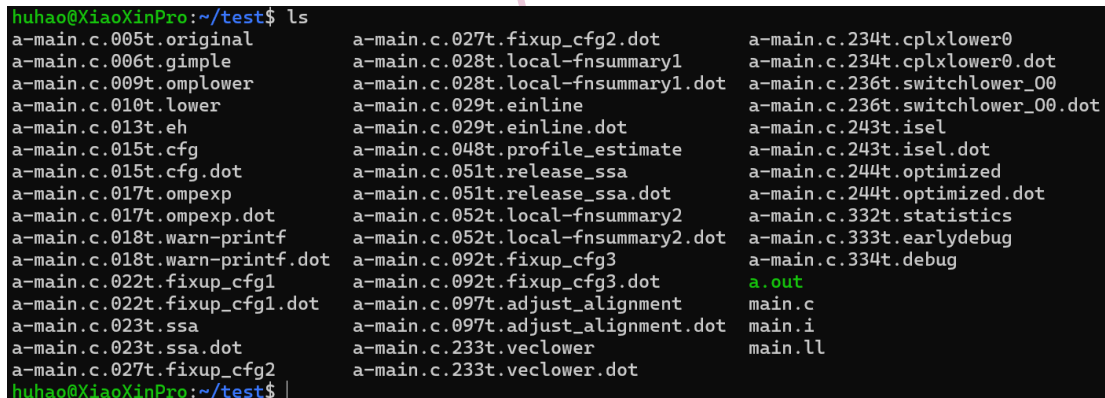
而在 gcc 中，可以通过 fdump-tree-all-graph 和 fdump-rtl-all-graph 两个 gcc flag 获得中间代码生成的多阶段的输出。

- fdump-tree-all: 这个标志告诉 GCC 在编译过程中输出 AST 的不同阶段的文本表示。AST 是源代码的抽象语法树，它保持了源代码的层次结构，并且每个节点都代表了源代码中的一个语法元素。通过这种方式，可以直观地看到源代码是如何被分解成更小的语法元素的，以及这些元素是如何被组织成树状结构的。
- fdump-rtl-all: 这个标志用于生成 GCC 的 RTL 表示的文本表示。RTL 是 GCC 特有的中间表示形式，它表示了源代码的操作顺序和寄存器使用情况。通过这种方式，直观地看到源代码是如何被转换成更接近机器语言的操作序列的，以及这些操作是如何被组织成树状结构的。

gcc 获得多阶段输出

```
1 gcc -fdump-tree-all-graph main.c
2 gcc -fdump-rtl-all-graph main.c
```

本次实验使用 `gcc -fdump-tree-all-graph main.c` 命令来获得输出。



```
huhao@XiaoXinPro:~/test$ ls
a-main.c.005t.original      a-main.c.027t.fixup_cfg2.dot      a-main.c.234t.cplxlower0
a-main.c.006t.gimple        a-main.c.028t.local-fnsummary1     a-main.c.234t.cplxlower0.dot
a-main.c.009t.ompower       a-main.c.028t.local-fnsummary1.dot a-main.c.236t.switchlower_00
a-main.c.010t.lower         a-main.c.029t.einline             a-main.c.236t.switchlower_00.dot
a-main.c.013t.eh            a-main.c.029t.einline.dot         a-main.c.243t.isel
a-main.c.015t.cfg           a-main.c.048t.profile_estimate    a-main.c.243t.isel.dot
a-main.c.015t.cfg.dot       a-main.c.051t.release_ssa         a-main.c.244t.optimized
a-main.c.017t.ompexp        a-main.c.051t.release_ssa.dot     a-main.c.244t.optimized.dot
a-main.c.017t.ompexp.dot    a-main.c.052t.local-fnsummary2     a-main.c.332t.statistics
a-main.c.018t.warn-printf   a-main.c.052t.local-fnsummary2.dot a-main.c.333t.earlydebug
a-main.c.018t.warn-printf.dot a-main.c.092t.fixup_cfg3          a-main.c.334t.debug
a-main.c.022t.fixup_cfg1    a-main.c.092t.fixup_cfg3.dot      a.out
a-main.c.022t.fixup_cfg1.dot a-main.c.097t.adjust_alignment     main.c
a-main.c.023t.ssa           a-main.c.097t.adjust_alignment.dot main.i
a-main.c.023t.ssa.dot       a-main.c.233t.veclower            main.ll
a-main.c.027t.fixup_cfg2    a-main.c.233t.veclower.dot
huhao@XiaoXinPro:~/test$
```

图 14: gcc 多阶段输出产生许多 dot 文件

运行上述命令，生成了一大堆 dot 文件。我们使用 graphviz 可视化，可视化后可以看到控制流图（CFG），以及各阶段处理中（比如优化、向 IR 转换）CFG 的变化。

在这里我以 `a-main.c.017t.ompexp.dot` 文件为例，通过 vscode 插件来获得控制流图片。

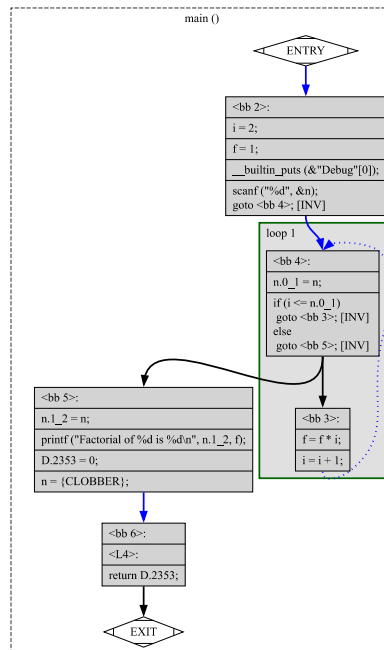


图 15: a-main.c.017t.ompeexp.dot 控制流图

从 CFG 中可以看出，阶乘计算被分成了多个基本块，每个块代表程序的一部分执行过程。并且，控制流图具有如下三个特征：

1. 代码块之间的关系：从 CFG 中可以看到程序执行的不同代码块之间的逻辑关系。每个代码块都用类似 bb_2、bb_3 这样的标识符来表示，显示了程序在执行过程中从一个代码块跳转到另一个代码块的顺序和分支情况。这反映了程序的控制流逻辑，包括跳转和分支指令。
2. SSA（静态单赋值）形式：在 SSA 形式中，每个变量只会被赋值一次，每个变量在整个程序中都是唯一的。例如，原来的变量 i、f 和 n 现在变成了唯一的版本，如 i_4、i_11、n_5，表示这些变量在程序不同阶段的不同值。返回值也类似，变成了类似 _21 这样的编号，表示程序在不同的执行阶段返回的不同值。
3. 变量的阶段：每个变量后面的编号（如 i_4、i_11）用来区分该变量在程序不同位置的值变化。这个编号系统帮助我们理解程序在执行过程中，变量的状态是如何随时间变化的。这对于程序优化和跟踪变量状态非常重要。

通过 CFG 和 SSA 转换，能够清晰地展示代码的执行流程和变量的变化，并查看每个变量在程序执行过程中所处的不同状态。

5. 代码优化

代码优化是编译器在将源代码转换为可执行程序的过程中，对程序进行的一系列改进，目的是提高程序的运行效率、减少资源消耗，以及优化程序的存储和传输。优化可以在不同的层次上进行，包括源代码级别、中间表示级别和目标代码级别。代码优化通常不改变程序的功能，但会改变程序的内部结构和执行方式。

在 LLVM 中，它利用一系列预定义的优化 pass 来改进程序的中间表示 (IR)。这些优化 pass 被设计用来提高程序的性能、减少代码大小、降低内存使用，并改善程序的其他可执行特性。

LLVM 提供不同的优化级别，从-O0（无优化）到-O3（最高优化），以及-Os（优化大小）和-Ofast（不考虑标准兼容性的优化）。每个级别启用不同的优化 pass 组合。

在 Ubuntu 中，在使用 pass 进行优化之前，先使用下述命令得到 LLVM IR 的二进制代码形式

```
1 llvm-as main.ll -o main.bc
```

之后我们可以使用下面的命令生成每个 pass 后生成的 LLVM IR，以观察差别：

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

当然，我们也可以用下面命令来指定获得某个 pass 后的中间代码。

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

6. 目标代码生成

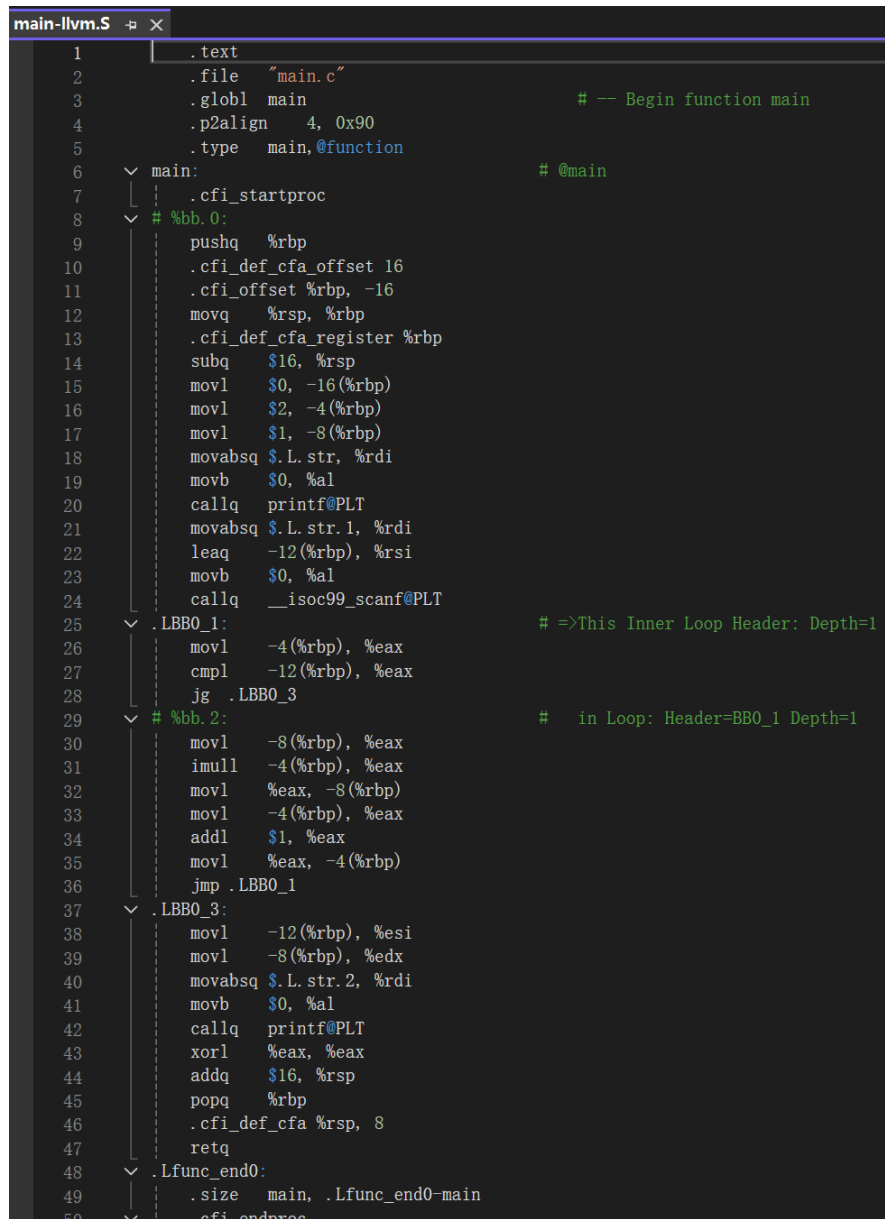
目标代码生成是编译器设计的最后一个阶段，在这个过程中编译器将优化后的中间代码转换成特定计算机硬件能直接执行的机器代码或汇编代码。这个过程确保了程序能在目标机器上高效运行。生成目标代码时，编译器会考虑目标机器的指令集、寄存器、内存结构等因素，以生成最适合该硬件的代码。最终，编译器输出的是汇编语言或机器语言，这些语言描述了程序中的每一条指令和数据操作，能够在目标机器上直接执行。


```

1 gcc main.c -S -o main-x86.S # 生成x86格式目标代码
2 llc main.ll -o main-llvm.S # 生成LLVM目标代码
3 riscv64-unknown-linux-gnu-gcc main.c -S -o main-riscv.S # 生成riscv格式目标代码
4 arm-linux-gnueabi-gcc main.c -S -o main-arm.S # 生成arm格式目标代码

```

由于获取 ARM 架构的汇编代码需要使用 ARM 的交叉编译器，我们将使用 x86、RISC-V 以及 LLVM 生成的目标代码来进行分析。分别执行三个命令，我们可以得到 3 个不同的文件。



```

main-llvm.S
1      .text
2      .file "main.c"
3      .globl main                                # -- Begin function main
4      .p2align 4, 0x90
5      .type main,@function
6
7      main:                                       # @main
8      .cfi_startproc
9      # %bb.0:
10     pushq %rbp
11     .cfi_def_cfa_offset 16
12     .cfi_offset %rbp, -16
13     movq %rsp, %rbp
14     .cfi_def_cfa_register %rbp
15     subq $16, %rsp
16     movl $0, -16(%rbp)
17     movl $2, -4(%rbp)
18     movl $1, -8(%rbp)
19     movabsq $.L.str, %rdi
20     movb $0, %al
21     callq printf@PLT
22     movabsq $.L.str.1, %rdi
23     leaq -12(%rbp), %rsi
24     movb $0, %al
25     callq __isoc99_scanf@PLT
26
27     .LBB0_1:                                    # =>This Inner Loop Header: Depth=1
28     movl -4(%rbp), %eax
29     cmpl -12(%rbp), %eax
30     jg .LBB0_3
31
32     # %bb.2:                                    # in Loop: Header=BB0_1 Depth=1
33     movl -8(%rbp), %eax
34     imull -4(%rbp), %eax
35     movl %eax, -8(%rbp)
36     movl -4(%rbp), %eax
37     addl $1, %eax
38     movl %eax, -4(%rbp)
39     jmp .LBB0_1
40
41     .LBB0_3:
42     movl -12(%rbp), %esi
43     movl -8(%rbp), %edx
44     movabsq $.L.str.2, %rdi
45     movb $0, %al
46     callq printf@PLT
47     xorl %eax, %eax
48     addq $16, %rsp
49     popq %rbp
50     .cfi_def_cfa %rsp, 8
51     retq
52
53     .Lfunc_end0:
54     .size main, .Lfunc_end0-main
55     .cfi_endproc

```

图 16: main-llvm.S

```

main-riscv.S  ✕
1      .file "main.c"
2      .option nopic
3      .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
4      .attribute unaligned_access, 0
5      .attribute stack_align, 16
6      .text
7      .section .rodata
8      .align 3
9      .LC0:
10     .string "Debug"
11     .align 3
12     .LC1:
13     .string "%d"
14     .align 3
15     .LC2:
16     .string "Factorial of %d is %d\n"
17     .text
18     .align 1
19     .globl main
20     .type main, @function
21     main:
22     addi sp, sp, -32
23     sd ra, 24(sp)
24     sd s0, 16(sp)
25     addi s0, sp, 32
26     li a5, 2
27     sw a5, -20(s0)
28     li a5, 1
29     sw a5, -24(s0)
30     lui a5, %hi(.LC0)
31     addi a0, a5, %lo(.LC0)
32     call puts
33     addi a5, s0, -28
34     mv a1, a5
35     lui a5, %hi(.LC1)
36     addi a0, a5, %lo(.LC1)
37     call __isoc99_scanf
38     j .L2
39     .L3:
40     lw a5, -24(s0)
41     mv a4, a5
42     lw a5, -20(s0)
43     mulw a5, a4, a5
44     sw a5, -24(s0)
45     lw a5, -20(s0)
46     addiw a5, a5, 1
47     sw a5, -20(s0)
48     .L2:
49     lw a4, -28(s0)
50     lw a5, -20(s0)

```

图 17: main-riscv.S

```

main-x86.S  + X
1      .file "main.c"
2      .text
3      .section .rodata
4      .LC0:
5      | .string "Debug"
6      .LC1:
7      | .string "%d"
8      .LC2:
9      | .string "Factorial of %d is %d\n"
10     .text
11     .globl main
12     .type main, @function
13     main:
14     .LFB0:
15     | .cfi_startproc
16     | endbr64
17     | pushq %rbp
18     | .cfi_def_cfa_offset 16
19     | .cfi_offset 6, -16
20     | movq %rsp, %rbp
21     | .cfi_def_cfa_register 6
22     | subq $32, %rsp
23     | movq %fs:40, %rax
24     | movq %rax, -8(%rbp)
25     | xorl %eax, %eax
26     | movl $2, -16(%rbp)
27     | movl $1, -12(%rbp)
28     | leaq .LC0(%rip), %rax
29     | movq %rax, %rdi
30     | call puts@PLT
31     | leaq -20(%rbp), %rax
32     | movq %rax, %rsi
33     | leaq .LC1(%rip), %rax
34     | movq %rax, %rdi
35     | movl $0, %eax
36     | call __isoc99_scanf@PLT
37     | jmp .L2
38     .L3:
39     | movl -12(%rbp), %eax
40     | imull -16(%rbp), %eax
41     | movl %eax, -12(%rbp)
42     | addl $1, -16(%rbp)
43     .L2:
44     | movl -20(%rbp), %eax
45     | cmpl %eax, -16(%rbp)
46     | jle .L3
47     | movl -20(%rbp), %eax
48     | movl -12(%rbp), %edx
49     | movl %eax, %esi
50     | leaq .LC2(%rip), %rax

```

3. 汇编器

汇编器的功能

汇编器是编译过程中负责将汇编语言代码转换为机器代码的工具。它将汇编语言中的指令和数据转换成目标机器能够理解和执行的指令序列。在现代编译器中，汇编器的作用通常由编译器后端完成，而不需要单独的汇编器工具。

汇编器将汇编语言程序代码翻译成目标机器的指令，并将这些指令打包成可重定位的目标程序。生成的目标文件通常是目标平台的可重定位目标文件（.o 或.obj 文件），包含了机器代码、数据、符号表、重定位信息等。

汇编器处理的结果

x86 格式汇编我们可以直接使用 gcc 完成汇编器的工作，如使用下面的命令：

```
1 gcc main-x86.S -c -o main-x86.o
```

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

```
1 llc main-llvm.bc -filetype=obj -o main-llvm.o
```

riscv 格式汇编需要用到交叉编译，如使用下面的命令：

```
1 riscv64-unknown-linux-gnu-gcc main-riscv.S -c -o main-riscv.o
```

由于篇幅所限，在此我仅针对 x86 进行反汇编分析。

在使用上述 gcc 命令完成 x86 汇编后，利用下面的命令对 main-x86.o 反汇编：

```
1 objdump -d main-x86.o
```

运行完上述指令，会输出以下反汇编结果：

反汇编结果

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

```

12 13: 00 00
13 15: 48 89 45 f8      mov    %rax,-0x8(%rbp)
14 19: 31 c0              xor    %eax,%eax
15 1b: c7 45 f0 02 00 00 00 movl   $0x2,-0x10(%rbp)
16 22: c7 45 f4 01 00 00 00 movl   $0x1,-0xc(%rbp)
17 29: 48 8d 05 00 00 00 00 lea     0x0(%rip),%rax      # 30 <main+0x30>
18 30: 48 89 c7            mov    %rax,%rdi
19 33: e8 00 00 00 00      call   38 <main+0x38>
20 38: 48 8d 45 ec          lea     -0x14(%rbp),%rax
21 3c: 48 89 c6            mov    %rax,%rsi
22 3f: 48 8d 05 00 00 00 00 lea     0x0(%rip),%rax      # 46 <main+0x46>
23 46: 48 89 c7            mov    %rax,%rdi
24 49: b8 00 00 00 00      mov    $0x0,%eax
25 4e: e8 00 00 00 00      call   53 <main+0x53>
26 53: eb 0e              jmp     63 <main+0x63>
27 55: 8b 45 f4            mov     -0xc(%rbp),%eax
28 58: 0f af 45 f0          imul    -0x10(%rbp),%eax
29 5c: 89 45 f4            mov     %eax,-0xc(%rbp)
30 5f: 83 45 f0 01          addl    $0x1,-0x10(%rbp)
31 63: 8b 45 ec            mov     -0x14(%rbp),%eax
32 66: 39 45 f0            cmp     %eax,-0x10(%rbp)
33 69: 7e ea              jle     55 <main+0x55>
34 6b: 8b 45 ec            mov     -0x14(%rbp),%eax
35 6e: 8b 55 f4            mov     -0xc(%rbp),%edx
36 71: 89 c6              mov     %eax,%esi
37 73: 48 8d 05 00 00 00 00 lea     0x0(%rip),%rax      # 7a <main+0x7a>
38 7a: 48 89 c7            mov    %rax,%rdi
39 7d: b8 00 00 00 00      mov    $0x0,%eax
40 82: e8 00 00 00 00      call   87 <main+0x87>
41 87: b8 00 00 00 00      mov    $0x0,%eax
42 8c: 48 8b 55 f8          mov     -0x8(%rbp),%rdx
43 90: 64 48 2b 14 25 28 00 sub     %fs:0x28,%rdx
44 97: 00 00
45 99: 74 05              je      a0 <main+0xa0>
46 9b: e8 00 00 00 00      call   a0 <main+0xa0>
47 a0: c9                leave
48 a1: c3                ret

```

汇编器的具体功能分析

仔细查看代码结构，它展示了基本的栈帧设置、局部变量初始化、函数调用和循环处理。具体来说：

- 0-8 是函数入口及栈帧初始化
- c-22 是局部变量初始化与设置

- 29-33 是加载地址和函数调用
- 55-69 是循环体与乘法运算
- 87-a1 是函数结束及返回

根据这段代码，我们至少可以得出汇编器的以下几个功能：

1. 指令翻译：汇编器将汇编代码中的每条指令翻译成对应的机器码。例如，`mov %rsp, %rbp` 被翻译为 `48 89 e5` 这三个字节的机器码。这些机器码可以直接在 CPU 上执行。
2. 符号管理：汇编器管理程序中的符号，例如函数名称、变量名等。它将符号与内存地址关联起来，并为链接器生成符号表，以便多个目标文件在链接时能够互相引用符号。例如，在这段代码中，函数 `main` 和其他跳转目标的符号会被汇编器管理。
3. 内存地址分配：汇编器负责为指令和数据分配内存地址，并生成适当的偏移量。它处理相对地址和绝对地址的分配，确保指令和数据能够正确定位。例如，`lea 0x0(%rip), %rax` 中的相对地址就是汇编器根据指令位置计算出来的。
4. 节组织：汇编器将代码、数据等内容分别组织到不同的节中，比如代码段（`.text`）、数据段（`.data`）、只读数据段（`.rodata`）等。不同的节有不同的用途，编译器和链接器在生成可执行文件时会根据这些节组织内容。
5. 错误检查：汇编器在翻译时会进行基本的语法检查，确保每条汇编指令的语法正确。如果发现错误，会抛出相应的错误信息，提示开发者修复。

4. 链接器和加载器

链接器、加载器的功能

链接器和加载器是计算机程序执行过程中的最后两步，它们共同确保了编译后的程序能够在操作系统上正确地执行。

链接器的主要功能是将一个或多个由编译器或汇编器生成的目标文件以及库文件合并成一个完整的可执行文件，它通过解析符号引用、合并代码和数据段、重定位地址引用等操作，将分散的代码片段组织成一个连贯的程序实体。

加载器的功能则是将可执行文件中的代码和数据加载到内存中，并准备程序的执行环境，它负责在程序运行时进行内存映射、地址重定位、初始化程序运行所需的各种资源，并将控制权传递给程序的入口点，从而启动程序的执行。

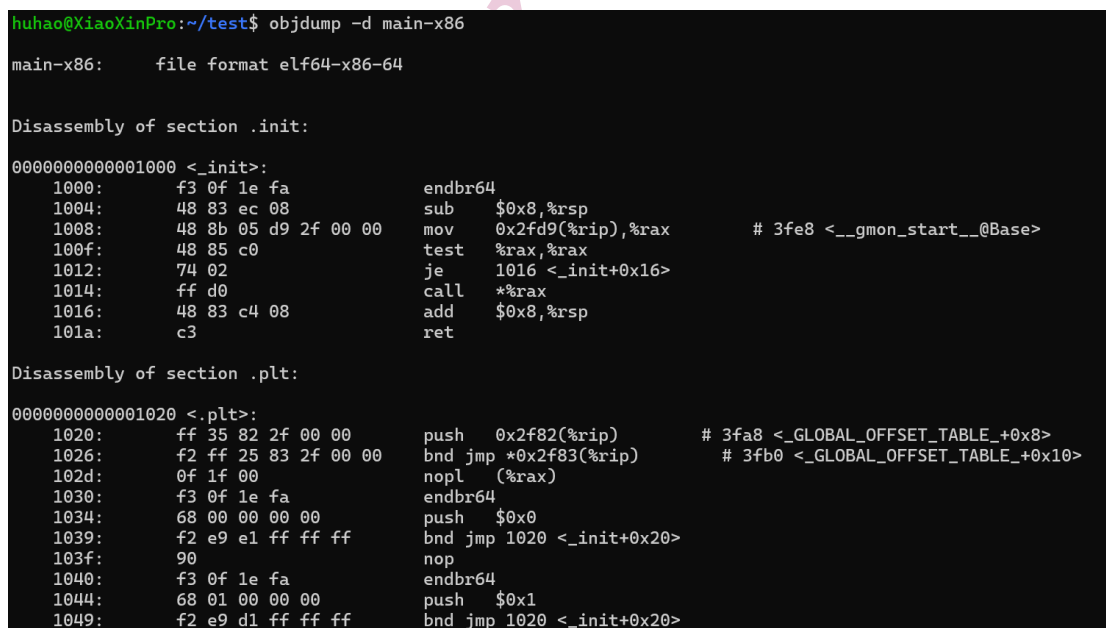
简而言之，链接器负责构建可执行文件，而加载器则负责将可执行文件转换成运行中的程序。

链接器处理的结果及对比分析

与汇编器类似，我们可以使用下面的命令分别生成不同链接的文件

```
1 clang main-llvm.o -o main-llvm
2 gcc main-x86.o -o main-x86
3 gcc main-arm.o -o main-arm
```

在此我以 x86 为例，分析 main-x86 反汇编。执行上述 x86 命令，生成对应的文件后，再使用 `objdump -d main-x86` 指令，即可输出反汇编结果。如下图所示：



```
huhao@XiaoXinPro:~/test$ objdump -d main-x86
main-x86:      file format 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        # 3fe8 <__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 82 2f 00 00      push   0x2f82(%rip)            # 3fa8 <_GLOBAL_OFFSET_TABLE_+0x8>
 1026:    f2 ff 25 83 2f 00 00    bnd jmp *0x2f83(%rip)         # 3fb0 <_GLOBAL_OFFSET_TABLE_+0x10>
 102d:    0f 1f 00                nopl   (%rax)
 1030:    f3 0f 1e fa                endbr64
 1034:    68 00 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 00      push   $0x1
 1049:    f2 e9 d1 ff ff ff      bnd jmp 1020 <_init+0x20>
```

图 19: x86 链接后部分反汇编结果

可以发现，相较于汇编器生成文件的反汇编结果，有以下几个差别，很好地验证了链接器的功能：

- 全局和静态变量的地址是全局的，并且在反汇编代码中可以看到它们的具体地址

- 函数调用会被解析为具体的跳转指令，指向函数的实际地址
- 链接器会包含外部库函数的实现，反汇编代码中会显示这些函数的机器码
- 链接前目标文件中可能包含大量的重定位条目，这些条目指示了哪些地址需要在链接时被修正，但链接后这些重定位条目已经被处理

执行可执行文件

最终我们如愿生成了可执行文件，然后我们执行所生成的可执行文件，如下图所示，我们输入 5 得到 120，说明可阶乘的实现是正确的。（这里 DEBUG 全局设置为了 1，不用在意）

```
huhao@XiaoXinPro:~/test$ ./main-x86
Debug
5
Factorial of 5 is 120
huhao@XiaoXinPro:~/test$
```

图 20: x86 运行结果

(三) 问题二

1. LLVM IR 程序简述

为了展示多种编程语言特性在 LLVM IR 中的表现，我编写了一个综合性的示例，涵盖了从基本的变量定义和算术运算，到更复杂的控制流、数据结构和内存管理等方面。程序通过操作这些元素并输出结果来演示 LLVM IR 如何处理各种编程概念。通过这一个相对紧凑的程序中，我们可以观察到 LLVM IR 如何处理各种常见的编程概念和结构。进而对比 SysY 源代码和生成的 LLVM IR，分析它们之间的关系。

代码展示了许多编程语言中的特性，包括全局变量、结构体、函数调用、循环、条件语句、switch 语句、函数指针调用、动态内存分配等。以下是核心部分的简述：

1. 全局变量和常量：

- 定义了一个全局数组 @arr，存储 5 个整数。
- 定义了换行符和空格字符串常量，供打印时使用。

- 定义了一个结构体 `MyStruct`, 包含一个整型和一个浮点型成员, 并用全局变量 `@global_struct` 进行初始化。

2. 外部函数声明:

- 声明了外部的输入输出函数, 如 `putint`、`putfloat`、`getint`、`getfloat`, 以及动态内存分配函数 `malloc` 和 `free`, 均是通过 `sysY` 运行时库调用。

3. 辅助函数:

- `output_newline()` 和 `output_space()` 用于输出换行符和空格, 方便输出格式化。
- `add_ten()` 是一个简单的加法函数, 将输入的整数加 10 并返回。

4. 功能展示函数:

- **全局数组使用:** 调用 `putarray` 输出全局数组 `@arr`。
- **结构体操作:** 通过 `getelementptr` 获取结构体的成员并输出。
- **循环与条件语句:** 实现了一个简单的 `for` 循环, 根据条件输出整数, 跳过特定值。
- **Switch 语句:** 使用 `switch` 实现分支逻辑, 基于输入的整数输出不同的字符串。
- **浮点数和位运算:** 演示了浮点乘法和按位与操作。
- **函数指针调用:** 展示了如何通过函数指针调用 `add_ten` 函数。
- **动态内存分配:** 使用 `malloc` 分配内存, 并对动态数组进行操作后释放内存。

5. 主函数:

- 调用了 `demonstrate_features()` 函数以展示这些语言特性。

总之, 实现了以下功能的展示:

- (1) 基本数据类型: 整型、浮点型
- (2) 复合数据类型: 数组 (全局数组和动态分配的数组)、结构体

- (3) 变量定义和初始化：全局变量和局部变量
- (4) 常量定义：字符串
- (5) 表达式和运算：算术运算、位运算、比较运算
- (6) 控制流语句：if 语句、for 循环、switch 语句
- (7) 函数定义和调用：直接调用和通过函数指针调用
- (8) 指针操作：函数指针、内存操作
- (9) 动态内存管理：malloc 分配内存、free 释放内存
- (10) 输入输出：通过 sysY 运行库调用
- (11) 类型转换：隐式类型转换：如整型和指针类型之间的转换
- (12) 全面使用 OpaquePointers 特性，简化我们编译器实现数组和指针的难度

2. 具体代码

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

```

1 ; LLVM IR 示例：展示各种语言特性
2 ; 目标三元组
3 target triple = "x86_64-pc-linux-gnu"
4
5 ; —— 全局变量和常量 ——
6
7 ; 定义全局整型数组 arr
8 @arr = global [5 x i32] [i32 1, i32 2, i32 3, i32 4, i32 5]
9
10 ; 定义换行符和空格字符串常量
11 ; unnamed_addr 属性表明该变量或常量的地址不会被直接引用, \00 是字符串的 null 终止符
12 @str_newline = private unnamed_addr constant [2 x i8] c"\0A\00"
13 @str_space = private unnamed_addr constant [2 x i8] c" \00"
14
15 ; 定义结构体 MyStruct
16 %MyStruct = type { i32, float }
17
18 ; 定义全局结构体变量 global_struct
19 @global_struct = global %MyStruct { i32 10, float 3.25 }
20
21 ; 定义全局函数指针变量，初始化为 null
22 @function_ptr = global ptr null
23
24 ; 定义 switch 语句用的字符串常量
25 @str_case_ten = private unnamed_addr constant [23 x i8] c"Switch case: Ten
    (10)\0A\00"

```

```

26 @str_case_nine = private unnamed_addr constant [23 x i8] c"Switch case: Nine
    (9)\0A\00"
27 @str_case_eight = private unnamed_addr constant [24 x i8] c"Switch case:
    Eight (8)\0A\00"
28 @str_default_case = private unnamed_addr constant [27 x i8] c"Switch case:
    Default case\0A\00"
29
30 ; —— 外部函数声明 ——
31 declare void @putint(i32)
32 declare void @putfloat(float)
33 declare void @putarray(i32, ptr)
34 declare void @putf(ptr, ...)
35 declare i32 @getint()
36 declare float @getfloat()
37 declare ptr @malloc(i64)
38 declare void @free(ptr)
39
40 ; —— 辅助函数 ——
41
42 ; 输出换行符
43 define void @output_newline() {
44     call void @putf(ptr @str_newline)
45     ret void
46 }
47
48 ; 输出空格
49 define void @output_space() {
50     call void @putf(ptr @str_space)
51     ret void
52 }
53
54 ; 定义 add_ten 函数: int add_ten(int x) { return x + 10; }
55 define i32 @add_ten(i32 %x) {
56     %result = add i32 %x, 10
57     ret i32 %result
58 }
59
60 ; —— 功能展示函数 ——
61
62 define void @demonstrate_features() {
63     ; 使用全局数组
64     call void @putarray(i32 5, ptr @arr)
65     call void @output_newline()
66
67     ; 结构体
68     ; 获取结构体成员
69     ; getelementptr指令用于计算给定类型的元素在内存中的地址,inbounds关键字用于
        优化,i32 0指定结构体实例的索引 (在这个例子中是第一个,也是唯一一个实

```

```

    例)
70  %global_int = load i32, ptr @getelementptr inbounds (%MyStruct, ptr
    @global_struct, i32 0, i32 0)
71  %global_float = load float, ptr @getelementptr inbounds (%MyStruct, ptr
    @global_struct, i32 0, i32 1)
72  ; 输出结构体成员
73  call void @putint(i32 %global_int)
74  call void @output_space()
75  call void @putfloat(float %global_float)
76  call void @output_newline()
77
78  ; 循环和条件语句
79  ; 实现 for (int i = input; i < end; i++) { if (i != 5) printf("%d", i); }
80  ; 为循环变量i分配栈空间
81  %i = alloca i32
82  ; 输入循环开始、结束和跳过的整数
83  %input_int = call i32 @getint()
84  store i32 %input_int, ptr %i
85  %end = call i32 @getint()
86  %not_print = call i32 @getint()
87  br label %loop_start
88
89  loop_start:
90  ; 加载i的值
91  %i_val = load i32, ptr %i
92  ; 比较i是否小于end
93  %cond = icmp slt i32 %i_val, %end
94  ; 根据比较结果跳转
95  br i1 %cond, label %loop_body, label %loop_end
96
97  loop_body:
98  ; 检查i是否等于不需要打印的整数
99  %is_five = icmp eq i32 %i_val, %not_print
100  ; 如果i等于5, 跳过打印
101  br i1 %is_five, label %continue, label %print
102
103  print:
104  ; 打印i的值
105  call void @putint(i32 %i_val)
106  call void @output_space()
107  br label %continue
108
109  continue:
110  ; i自增
111  %next_i = add i32 %i_val, 1
112  store i32 %next_i, ptr %i
113  ; 跳回循环开始
114  br label %loop_start

```

```
115
116 loop_end:
117     call void @output_newline()
118
119     ; Switch语句
120     ; 实现 switch (i) { case 10: ... case 9: ... case 8: ... default: ... }
121     ; 输入整数i
122     %final_i = call i32 @getint()
123     switch i32 %final_i, label %default_case [
124         i32 10, label %case_ten
125         i32 9, label %case_nine
126         i32 8, label %case_eight
127     ]
128
129 case_ten:
130     call void @putf(ptr @str_case_ten)
131     br label %switch_end
132
133 case_nine:
134     call void @putf(ptr @str_case_nine)
135     br label %switch_end
136
137 case_eight:
138     call void @putf(ptr @str_case_eight)
139     br label %switch_end
140
141 default_case:
142     call void @putf(ptr @str_default_case)
143     br label %switch_end
144
145 switch_end:
146     ; 浮点数和位运算操作
147     ; 计算 a * b
148     %a = call float @getfloat()
149     %b = call float @getfloat()
150     %sum = fmul float %a, %b
151     call void @putfloat(float %sum)
152     call void @output_newline()
153     ; 计算 c & d
154     %c = call i32 @getint()
155     %d = call i32 @getint()
156     %and_result = and i32 %c, %d
157     call void @putint(i32 %and_result)
158     call void @output_newline()
159
160     ; 函数指针调用
161     ; 实现 function_ptr = &add_ten; printf("%d", function_ptr(in));
162     ; 将add_ten函数的地址存储到函数指针
```

```

163     store ptr @add_ten, ptr @function_ptr
164     ; 加载函数指针
165     %func = load ptr, ptr @function_ptr
166     ; 通过函数指针调用函数,调用输入函数传入参数in
167     %in = call i32 @getint()
168     %result = call i32 %func(i32 %in)
169     ; 打印结果
170     call void @putint(i32 %result)
171     call void @output_newline()
172
173     ; 动态内存分配
174     ; 实现 int *ptr = (int *)malloc(20); ptr[0] = 10; ptr[1] = 20; ptr[2] =
        30;
175     ; 分配20字节的内存,足够存储5个int类型的值
176     %ptr = call ptr @malloc(i64 20)
177     ; 设置数组元素
178     %element0 = getelementptr i32, ptr %ptr, i32 0
179     store i32 10, ptr %element0
180     %element1 = getelementptr i32, ptr %ptr, i32 1
181     store i32 20, ptr %element1
182     %element2 = getelementptr i32, ptr %ptr, i32 2
183     store i32 30, ptr %element2
184     ; 打印数组
185     call void @putarray(i32 5, ptr %ptr)
186     call void @output_newline()
187     ; 释放内存
188     call void @free(ptr %ptr)
189
190     ret void
191 }
192
193 ; —— 主函数 ——
194 ; 定义主函数: int main() { demonstrate_features(); return 0; }
195 define i32 @main() {
196     call void @demonstrate_features()
197     ret i32 0
198 }

```

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

```

1 // —— 全局变量和常量 ——
2
3 // 定义全局整型数组 arr
4 int arr[5] = {1, 2, 3, 4, 5};
5
6 // 定义结构体类型 MyStruct
7 struct MyStruct {

```

```
8     int a;
9     float b;
10 };
11
12 // 定义全局结构体变量global_struct
13 struct MyStruct global_struct = {10, 3.25};
14
15 // —— 辅助函数 ——
16
17 // 输出换行符
18 void output_newline() {
19     putchar('\n');
20 }
21
22 // 输出空格
23 void output_space() {
24     putchar(' ');
25 }
26
27 // 定义 add_ten 函数: int add_ten(int x) { return x + 10; }
28 int add_ten(int x) {
29     return x + 10;
30 }
31
32 // —— 功能展示函数 ——
33 void demonstrate_features() {
34     int i;
35     int input, end, not_print;
36     int *ptr;
37
38     // 使用全局数组
39     putarray(5, arr);
40     output_newline();
41
42     // 输出结构体成员
43     putint(global_struct.a);
44     output_space();
45     putfloat(global_struct.b);
46     output_newline();
47
48     // 循环和条件语句
49     // for (int i = input; i < end; i++) { if (i != 5) printf("%d", i); }
50     input = getint(); // 输入循环开始的值
51     end = getint();   // 输入循环结束的值
52     not_print = getint(); // 不打印的值
53     for (i = input; i < end; i++) {
54         if (i != not_print) {
55             putint(i);
```

```
56         output_space();
57     }
58 }
59 output_newline();
60
61 // Switch 语句
62 // switch (i) { case 10: ... case 9: ... case 8: ... default: ... }
63 i = getint();
64 switch (i) {
65     case 10:
66         putstr("Switch case: Ten (10)\n");
67         break;
68     case 9:
69         putstr("Switch case: Nine (9)\n");
70         break;
71     case 8:
72         putstr("Switch case: Eight (8)\n");
73         break;
74     default:
75         putstr("Switch case: Default case\n");
76         break;
77 }
78
79 // 浮点数和位运算操作
80 float a, b, sum;
81 int c, d, and_result;
82
83 // 计算 a * b
84 a = getfloat();
85 b = getfloat();
86 sum = a * b;
87 putfloat(sum);
88 output_newline();
89
90 // 计算 c & d
91 c = getint();
92 d = getint();
93 and_result = c & d;
94 putint(and_result);
95 output_newline();
96
97 // 函数指针调用
98 // 实现 function_ptr = &add_ten; printf("%d", function_ptr(in));
99 int (*function_ptr)(int) = add_ten;
100 i = getint();
101 putint(function_ptr(i));
102 output_newline();
103
```



```

104 // 动态内存分配
105 // int *ptr = (int *)malloc(20); ptr[0] = 10; ptr[1] = 20; ptr[2] = 30;
106 ptr = (int *)malloc(20);
107 ptr[0] = 10;
108 ptr[1] = 20;
109 ptr[2] = 30;
110 putarray(5, ptr);
111 output_newline();
112 free(ptr);
113 }
114
115 // —— 主函数 ——
116 int main() {
117     demonstrate_features();
118     return 0;
119 }

```

3. 运行和结果分析

由于我调用了 SysY 运行时库，将该 IR 文件与 sylib.c 文件一起编译，执行命令“clang-15 111.ll sylib.c -o 111”即可得到可执行文件，结果如下图所示：

```

huhao@XiaoXinPro:~/Lab/Lab1$ clang-15 111.ll sylib.c -o 111
huhao@XiaoXinPro:~/Lab/Lab1$ ./111
5: 1 2 3 4 5

10 0x1.ap+1
-1
10
5
-1 0 1 2 3 4 6 7 8 9
7
Switch case: Default case
3.6
5.9
0x1.53d70ap+4
4 8
0
59
69
5: 10 20 30 0 0

TOTAL: 0H-0M-0S-0us
huhao@XiaoXinPro:~/Lab/Lab1$

```

图 21: llvm ir 运行结果

让我们根据运行结果来分析 LLVM IR 代码的执行过程和输出。

1. 数组输出：

```
5: 1 2 3 4 5
```

表示输出了全局数组 arr 的五个元素，成功展示数组内容。

2. 结构体成员输出:

```
10 0x1.ap+1
```

表示全局结构体 `global_struct` 的整型成员 10 和浮点型成员 3.25, 其中 `0x1.ap+1` 是浮点数的十六进制表示法。

3. for 循环输出:

```
-1
10
5
-1 0 1 2 3 4 6 7 8 9
```

表示在循环中打印了-1 到 10 的所有数值, 但跳过了数字 5。

4. Switch 语句:

```
7
Switch case: Default case
```

表示输入的值 7 不在 switch case 的条件中, 执行了默认分支。

5. 浮点数乘法与逻辑运算:

```
3.6
5.9
0x1.53d70ap+4
4 8
0
```

表示进行了浮点数相乘、整数与运算的输出, 其中浮点数 `0x1.53d70ap+4` 是浮点乘积的十六进制表示, 4 和 8 与运算的结果为 0。

6. 函数指针调用:

```
59
69
```

表示调用了 `add_ten` 函数, `59+10` 结果为 `69`。

7. 动态内存分配输出:

```
5: 10 20 30 0 0
```

表示动态分配的内存成功存储并输出前三个值为 `10`, `20`, `30`, 后两个值为默认初始化的 `0`。

8. 时间输出:

```
TOTAL: 0H-0M-0S-0us
```

程序执行时间统计为零, 这是因为调用 `sysY` 时未启用具体的时间统计逻辑。

整个运行结果与预期一致, 各个功能模块都正常展示了对应的语言特性。**因此, 这个运行结果很好地验证了 LLVM IR 代码的正确性。**

4. 语言特性具体分析

(1) **模块结构:** LLVM IR 文件通常包含全局变量、函数声明和函数定义。本示例中, 我们看到了全局变量 (如 `@arr`)、外部函数声明 (如 `@putint`) 和函数定义 (如 `@demonstrate_features`)。

(2) 数据类型:

- 基本类型: `i32` (32 位整数)、`float` (单精度浮点数)
- 复合类型: `[5 x i32]` (5 个整数的数组)、`%MyStruct` (自定义结构体)
- 指针类型: 使用 `ptr` 表示, 如 `i32*` (整数指针)

(3) 全局变量和常量:

- 全局变量: 使用 `@` 符号前缀, 如 `@arr = global [5 x i32] [i32 1, i32 2, i32 3, i32 4, i32 5]`
- 常量: 使用 `constant` 关键字, 如 `@str_newline = private unnamed_addr constant [2 x i8] c"\0A\00"`

(4) **函数定义**: 格式: `define [返回类型] @[函数名]([参数列表]) { [函数体] }` 例如: `define i32 @add_ten(i32 %x) { ... }`

(5) **基本块**: 函数体由基本块组成, 每个基本块以标签开始, 以终止指令 (如 `br` 或 `ret`) 结束。例如:

```
1 entry:
2   %result = add i32 %x, 10
3   ret i32 %result
```

(6) **SSA 形式**: LLVM IR 使用静态单赋值 (SSA) 形式, 每个变量只被赋值一次。使用 `%` 符号表示 SSA 值。例如: `%sum = fadd float %a_val, %a_val`

(7) **内存操作**:

- `alloca`: 在栈上分配内存, 如 `%i = alloca i32`
- `load`: 从内存加载值, 如 `%i_val = load i32, ptr %i`
- `store`: 存储值到内存, 如 `store i32 0, ptr %i`

(8) **控制流**:

- 条件分支: 使用 `icmp` 和 `br` 指令, 如:

```
1   %cond = icmp slt i32 %i_val, 10
2   br i1 %cond, label %loop_body, label %loop_end
```

- 无条件跳转: 使用 `br` 指令, 如 `br label %loop_start`
- `switch` 语句: 使用 `switch` 指令

(9) **算术和逻辑运算**:

- 整数运算: `add`, `sub`, `mul`, `sdiv`, `srem` 等
- 浮点运算: `fadd`, `fsub`, `fmul`, `fdiv` 等
- 位运算: `and`, `or`, `xor` 等

(10) **函数调用**: 使用 `call` 指令, 如: `call void @putint(i32 %i_val)`

(11) **指针和数组操作**: 使用 `getelementptr` 指令计算地址, 如:

```
1   %element0 = getelementptr i32, ptr %ptr, i32 0
```

(12) 类型转换:

- 整数到浮点: `sitofp`
- 浮点到整数: `fptosi`
- 位扩展: `zext`, `sext`
- 位截断: `trunc`

(13) **结构体操作**: 使用 `type` 关键字定义结构体, 用 `getelementptr` 访问成员:

```
1 %MyStruct = type { i32, float }  
2 %float_val = getelementptr %MyStruct, ptr %struct_ptr, i32 0, i32 1
```

(14) **连接 SysY 时库**: 和调用其他外部函数一样, 但是由于 SysY 库函数未直接在 C 语言中定义, 因此该 IR 文件编译时需要与 `sylib.c` 一起编译。

(15) **OpaquePointers 特性**: OpaquePointers 是 LLVM 15 中引入的一个重要特性, 它对 LLVM IR 的指针表示方式进行了重大改变。OpaquePointers 将所有指针类型统一为单一的 `ptr` 类型, 不再区分具体指向的类型。从而简化 LLVM IR 的类型系统, 提高编译器的性能和灵活性。

通过这个综合性的示例, 我们可以看到 LLVM IR 如何以统一的方式表示各种编程语言构造。

5. LLVM IR 和 SysY 对比

对比 SysY 和 LLVM IR, 分析它们之间的关系如下:

1. 抽象级别

SysY 是一种高级编程语言, 类似于 C 语言的子集。它提供了易于人类理解和编写的语法结构, 如 `if-else` 语句、`for` 循环等。

LLVM IR 是一种中间表示, 处于高级语言和机器代码之间。它更接近底层, 使用基本块、SSA 形式等概念来表示程序逻辑。

2. 类型系统

SysY 有 `int`、`float` 等基本类型, 以及数组类型。

LLVM IR 有更丰富的类型系统, 包括 `i32`、`float` 等基本类型, 以及指针、数组、结构体等复合类型。

3. 控制流

SysY 使用结构化的控制流语句, 如 `if`、`while`、`for` 等。

LLVM IR 使用基本块和分支指令 (如 `br`) 来表示控制流。

4. 变量和内存

SysY 允许直接声明和使用变量。

LLVM IR 使用 `alloca`、`load`、`store` 等指令显式管理内存。

5. 函数

SysY 允许定义和调用函数, 支持参数传递和返回值。

LLVM IR 也支持函数, 但使用更底层的语法, 如 `define` 和 `call` 指令。

6. 转换过程

SysY 代码通常会被编译器前端转换为 LLVM IR。这个过程涉及:

- 语法分析和语义检查
- 生成对应的 LLVM IR 指令
- 将高级控制结构转换为基本块和分支
- 实现变量的内存分配和访问
- 函数调用的参数传递和返回值处理

7. 优化机会

LLVM IR 提供了更多的优化机会, 如常量折叠、死代码消除、循环优化等。这些优化在 SysY 级别可能难以实现。

8. 可移植性

SysY 是特定于编译原理教学的语言。

LLVM IR 是通用的中间表示, 可以被转换为多种目标架构的机器码。

总之, SysY 到 LLVM IR 的转换涉及将高级语言结构映射到更底层的表示。这个过程使得代码更接近机器级别, 同时保留了足够的信息以支持各种优化和代码生成策略。理解这种关系对于编译器开发和优化非常重要。

(四) 问题三

1. 简单计算器 riscv64 汇编程序

汇编代码

```
1 .file "cal.c"
2 .option nopic
3 .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
4 .attribute unaligned_access, 0
5 .attribute stack_align, 16
6 .section .rodata
7 .str1:
8     .string "%d %d %c"
9
10 .str2:
11     .string "%d"
12 .text
13 .globl AD
14 .type AD, @function
15 AD:
16     mv     a5, a0
17     mv     a6, a1
18     addw   a5, a5, a6
19     mv     a0, a5
20     jr     ra
21     .size  AD, .-AD
22
23
24
25 .globl SB
26 .type SB, @function
27 SB:
28     mv     a5, a0
29     mv     a6, a1
30     subw   a5, a5, a6
31     mv     a0, a5
32     jr     ra
33     .size  SB, .-SB
34
35
36 .globl ML
37 .type ML, @function
```

```
38
39 ML:
40     mv     a5, a0
41     mv     a6, a1
42     mulw   a5, a5, a6
43     mv     a0, a5
44     jr     ra
45
46     .size  ML, .-ML
47
48 .globl DV
49 .type  DV, @function
50 DV:
51     mv     a5, a0
52     mv     a6, a1
53     divw   a5, a5, a6
54     mv     a0, a5
55     jr     ra
56
57     .size  DV, .-DV
58
59
60
61 .text
62
63 .globl main
64 .type  main, @function
65
66 main:
67     addi   sp, sp, -32
68     sd     ra, 24(sp)
69     sd     s0, 16(sp)
70     addi   s0, sp, 32
71     addi   a3, s0, -25
72     addi   a4, s0, -24
73     addi   a5, s0, -20
74     mv     a1, a5
75     mv     a2, a4
76
77     lui    a0, %hi(.str1)
78     addi   a0, a0, %lo(.str1)
79     call   __isoc99_scanf
80     lbu    a3, -25(s0)
81     li     a4, 43
82     bne    a3, a4, .A
83     lw     a0, -20(s0)
84     lw     a1, -24(s0)
85     call   AD
```



```

86      mv      a5,a0
87      mv      a1,a5
88      lui      a5,%hi(.str2)
89      addi     a0,a5,%lo(.str2)
90      call     printf
91      j        .end
92
93
94 .A:
95      lbu      a3,-25(s0)
96      li       a4, 45
97      bne      a3,a4,.S
98      lw       a0, -20(s0)
99      lw       a1, -24(s0)
100     call     SB
101     mv       a5,a0
102     mv       a1,a5
103     lui      a5,%hi(.str2)
104     addi     a0,a5,%lo(.str2)
105     call     printf
106     j        .end
107
108
109 .S:
110     lbu      a3,-25(s0)
111     li       a4, 42
112     bne      a3,a4,.D
113     lw       a0, -20(s0)
114     lw       a1, -24(s0)
115     call     ML
116     mv       a5,a0
117     mv       a1,a5
118     lui      a5,%hi(.str2)
119     addi     a0,a5,%lo(.str2)
120     call     printf
121     j        .end
122
123 .D:
124     lbu      a3,-25(s0)
125     li       a4, 47
126     bne      a3,a4,.end
127     lw       a0, -20(s0)
128     lw       a1, -24(s0)
129     call     DV
130     mv       a5,a0
131     mv       a1,a5
132     lui      a5,%hi(.str2)
133     addi     a0,a5,%lo(.str2)
134     call     printf

```

```
134         j         .end
135
136
137 .end:
138     li         a5,0
139     mv         a0,a5
140     ld         ra,24(sp)
141     ld         s0,16(sp)
142     addi       sp,sp,32
143     jr         ra
```

核心代码解释

在第一个程序中，我实现的是一个简单计算器。具体来说就是输入两个整数型操作数和一个字符型操作符，再根据输入的操作符进行加减乘除运算。

首先，我用“.section .rodata”命令指定了”.str1”和”.str2”两个字符串常量是只读数据，不能被修改。之后使用“.text”段定义了代码段。在代码实现中，我是把四种运算各自都写成了一个全局函数，因为其逻辑都相同，所以在此以加法运算为例子解释我的代码。因为是全局函数，所以要使用”.globl”和”.type”标明这个加法函数。而在函数内部，因为输入的两个参数保存在了 a0 和 a1 寄存器，所以使用 mv 指令和 addw 指令进行加法运算。之后要把结果放进 a0 寄存器，然后使用 jr ra 指令跳转到调用函数的地方，接着执行下一个命令。

而因为 main 主函数也是一个函数，所以也要用”.globl”和”.type”定义该函数。在 main 函数内部，因为两个操作数和操作符都是局部变量，所以要使用栈给他们开辟空间。给局部变量开辟空间的同时，也要把一些要用到的寄存器值保存到栈里面，比如本次程序要用到 ra 和 s0 寄存器，所以在程序开始之初，要把他们的值保存到栈里面。接着就要给局部变量分配地址，因为两个操作数都是整数型，所以分别分配四字节空间，而操作符是字符型，所以分配一个字节空间就行。得到了变量的地址之后，就可以调用 scanf 函数来给他们赋值。

紧接着我们使用 lbu 指令和栈来获得输入操作符的值，并配合 bne 指令来根据不同的操作符跳转到不同的代码段。比如当操作符的 ascii 值等于 43 的时候，就代表着输入的操作符是“+”，这时候我们就要调用 AD 函数来对两个操作数进行加法运算，并调用 printf 函数来把最终结果输出到屏幕上。最后，我们需要恢复栈空间，把寄存器恢复到原来相应的值，然后使用 jr ra 指令结束函数调用。

程序运行

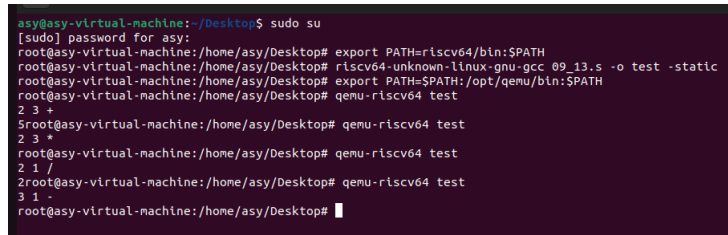
要想运行该汇编代码，首先我们需要 riscv64 交叉编译器来把该汇编代码转成可执行程序。

```
1 riscv64-unknown-linux-gnu-gcc test.s -o test -static
```

之后，要想运行该可执行程序，我们得要在 qemu 环境上去运行。

```
1 qemu-riscv64 test
```

通过运行截图可以知道，汇编代码成功被转成了可执行程序并且可以正确运行在 qemu 环境上面。



```
asy@asy-virtual-machine:~/Desktop$ sudo su
[sudo] password for asy:
root@asy-virtual-machine:/home/asy/Desktop# export PATH=riscv64/bin:$PATH
root@asy-virtual-machine:/home/asy/Desktop# riscv64-unknown-linux-gnu-gcc 09_13.s -o test -static
root@asy-virtual-machine:/home/asy/Desktop# export PATH=$PATH:/opt/qemu/bin:$PATH
root@asy-virtual-machine:/home/asy/Desktop# qemu-riscv64 test
2 3 +
2 3 *
2 1 /
2root@asy-virtual-machine:/home/asy/Desktop# qemu-riscv64 test
```

图 22: 程序运行结果

2. 数组求和 riscv64 汇编程序

汇编代码

```
1 .file "print.c"
2 .option nopic
3 .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
4 .attribute unaligned_access, 0
5 .attribute stack_align, 16
6 .section .rodata
7
8 .str1:
9 .string "please enter the 10 numbers:\n"
10
11 .str2:
12 .string "%d"
13
14 .str3:
15 .string "result is:%d\n"
16
17
18 .text
19 .globl main
20 .type main, @function
21
```

```

22 main :
23     addi    sp, sp, -64
24     sd      ra, 56(sp)
25     sd      s0, 48(sp)
26     sw      zero, 44(sp)
27     sw      zero, 40(sp)
28     addi    s0, sp, 64
29     lui     a0, %hi(.str1)
30     addi    a0, a0, %lo(.str1)
31     call    printf
32     j       .A
33
34 .B:
35     addi    a4, s0, -64
36     lw      a5, -24(s0)
37     slli    a5, a5, 2
38     add     a5, a4, a5
39     mv      a1, a5
40     lui     a0, %hi(.str2)
41     addi    a0, a0, %lo(.str2)
42     call    __isoc99_scanf
43     addi    a4, s0, -64
44     lw      a5, -24(s0)
45     slli    a5, a5, 2
46     add     a5, a4, a5
47     lw      a2, (a5)
48     sext.w  a1, a2
49     lw      a5, -20(s0)
50     addw    a5, a5, a1
51     sw      a5, -20(s0)
52     lw      a5, -24(s0)
53     addiw   a5, a5, 1
54     sw      a5, -24(s0)
55     j       .A
56
57 .A:
58     lw      a5, -24(s0)
59     sext.w  a4, a5
60     li      a5, 9
61     ble     a4, a5, .B
62     lw      a5, -20(s0)
63     mv      a1, a5
64     lui     a0, %hi(.str3)
65     addi    a0, a0, %lo(.str3)
66     call    printf
67     li      a5, 0
68     mv      a0, a5
69     ld      ra, 56(sp)

```

```
70      ld      s0,48(sp)
71      addi    sp,sp,64
72      jr      ra
73      .size   main,.-main
```

核心代码解释

在第二个程序中，我实现的是一个简单的数组求和程序。具体来说就是输入十个整数，然后输出他们求和后的结果。

跟第一个程序一样,首先,我用“.section .rodata”命令指定了”.str1”,”.str2”,”.str3”三个字符串常量是只读数据，不能被其修改。之后使用“.text”段定义了代码段。

```
1      .section      .rodata
2
3      .str1:
4          .string   "please inter the 10 numbers:\n"
5
6      .str2:
7          .string   "%d"
8
9      .str3:
10         .string   "result is:%d\n"
11
12
13      .text
```

而在本次程序中，我除了 main 主函数，没有用到别的函数，所以就只定义了 main 主函数。在 main 函数内部，因为数组是局部变量，所以也要使用栈给他们开辟空间。在给局部变量开辟空间的同时，也要把一些要用到的寄存器值保存到栈里面，比如本次程序要用到 ra 和 s0 寄存器，所以在程序开始之初，要把他们的值保存到栈里面。接着就要给局部变量分配地址，因为数组元素都是整数型，所以每个元素都分配四字节空间。除了数组元素要保存，本次程序中我们还要动态保存两个变量，一个是数组下标变量，而另外一个就是保存数组元素和的变量。因为在整个程序执行过程中，这两个变量是动态更新的，所以我们要不断的使用栈来存取他们。在程序开始之初，使用 zero 寄存器都给他们设置为零。

```
1      sw      zero,44(sp)
2      sw      zero,40(sp)
```

因为下标是从 0 开始的，所以我们每次循环都要跟 9 来进行比较，只要是小

于等于 9，就表示还没遍历完数组。而在循环内部，首先要确定当前数组元素的地址，然后调用 `scanf` 函数获取输入的值，然后再从栈里面获得保存数组元素和的 `SUM` 变量，执行加法操作，然后在保存到栈里面，而保存数组下标的 `index` 变量，每次循环都要加 1，然后保存到栈里面。待遍历完全部十个元素，我们就可以不进入循环，调用 `printf` 函数来输出 `SUM` 变量的值。最后在恢复栈空间，使用 `a0` 寄存器和 `jr` 指令结束函数调用。

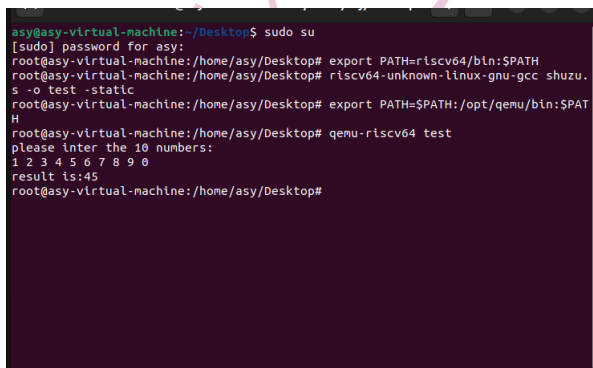
程序运行 要想运行该汇编代码，首先我们需要 `riscv64` 交叉编译器来把该汇编代码转成可执行程序。

```
1 riscv64-unknown-linux-gnu-gcc test.s -o test -static
```

之后，要想运行该可执行程序，我们得要在 `qemu` 环境上去运行。

```
1 qemu-riscv64 test
```

通过运行截图可以知道，汇编代码成功被转成了可执行程序并且可以正确运行在 `qemu` 环境上面。



```
asy@asy-virtual-machine:~/Desktop$ sudo su
[sudo] password for asy:
root@asy-virtual-machine:/home/asy/Desktop# export PATH=riscv64/bin:$PATH
root@asy-virtual-machine:/home/asy/Desktop# riscv64-unknown-linux-gnu-gcc shuzu.s -o test -static
root@asy-virtual-machine:/home/asy/Desktop# export PATH=$PATH:/opt/qemu/bin:$PATH
root@asy-virtual-machine:/home/asy/Desktop# qemu-riscv64 test
please enter the 10 numbers:
1 2 3 4 5 6 7 8 9 0
result is:45
root@asy-virtual-machine:/home/asy/Desktop#
```

图 23: 程序运行结果

三、 实验总结

在本次预备实验的探索过程中，我们不仅对 C-RISCV 框架有了更深入的理解，而且还在虚拟机上成功构建了实验环境。这一成就标志着我们对硬件架构和软件环境配置的掌握迈出了坚实的第一步。

通过亲自动手实践，我们对编译器的各个环节有了更加深刻的认识。从预处理阶段的宏替换和条件编译，到词法分析阶段的标记生成，再到语法分析阶段的语法树构建，每一步都让我们对编译器的工作原理有了更加直观的感受。语义分

析阶段的错误检测和类型检查，以及中间代码生成阶段的抽象语法树到中间表示的转换，都极大地提升了我们对编程语言和编译技术的理解。

在代码优化阶段，我们学习了如何通过各种优化技术提高代码的执行效率和资源利用率。汇编阶段的实践让我们对机器语言有了更直接的接触，而链接阶段的学习则让我们明白了如何将多个编译单元合并为一个可执行的程序。

通过编写 LLVM IR 中间代码和 RISC-V64 汇编代码，我们不仅初步掌握了底层编程语言，而且在不断的调试和优化过程中，也逐渐理解了这些底层语言的执行机制和优化技巧。这些经验对于我们未来在更高层次的编程和系统设计中，将有着不可估量的价值。

我们小组坚信，虽然预备实验只是一个开始，但它为我们打下了坚实的基础。面对未来可能遇到的挑战，我们有信心通过持续的学习和实践，不断克服困难，提升我们的技术能力。我们期待在未来的学习和研究中，能够取得更加显著的成就，并为计算机科学领域做出我们的贡献。

参考文献

- [1] 杨侯哲, 李煦阳, 杨科迪 费迪, 周辰霏, 谢子涵杨科迪, 李君龙, 华志远, 李帅东. 编译器开发环境部署.
- [2] 杨侯哲, 李煦阳, 杨科迪 费迪, 周辰霏, 谢子涵杨科迪, 李君龙, 华志远, 李帅东. 预备实验.

NIJUN