



南开大学
Nankai University

南 开 大 学

计 算 机 与 网 络 空 间 安 全 学 院

编译系统原理第一次实验报告

了解编译器- LLVM IR 编程以及汇编编程

石家伊 刘俊彤

年级：2022 级

专业：信息安全 计算机科学与技术

指导教师：王刚

2024 年 9 月 21 日

摘要

一个简单的 c 程序从编写到运行,中间经过了很多个阶段,包括预处理、编译、汇编、链接、执行。其中汇编器需要经过词法分析、语法分析、语义分析、中间代码生成、代码优化、代码生成六个阶段才能将我们编写的 c 代码转换为对应的更底层的汇编代码。再将汇编码交给汇编器,转化为机器码,最终链接,才得到能够运行的可执行文件。在本次实验中,以一个简单的 c 程序为例,通过生成编译过程的中间产物,来详细探索编译过程。并对于中间代码 LLVM IR 程序和汇编程序进行了学习和编写,为后续更进一步学习编译系统原理相关知识做准备。

关键字: 编译 LLVM-IR 汇编 RISC-V

目录

一、 预备工作以及实验平台	1
(一) 分工情况	1
(二) 刘俊彤实验平台	1
(三) 石家伊实验平台	1
二、 语言处理系统处理过程——石家伊	1
(一) 编写 c 程序	1
(二) 预处理器	3
(三) 编译器	3
1. 词法分析	4
2. 语法分析	4
3. 语义分析	5
4. 中间代码生成	5
5. 代码优化	7
6. 代码生成	10
(四) 汇编器	10
(五) 链接器加载器	10
三、 LLVM IR 编程——石家伊	11
(一) 学习 LLVM IR	11
(二) LLVM IR 编程	16
1. 程序 1——数组,除法,for 循环,打印	16
2. 程序 2——函数调用,输入输出	19
3. 程序 3——浮点数,if 分支判断,输入输出	22
(三) 关于 OpaquePointers	24
四、 RISC-V 汇编编程——刘俊彤	25
(一) RISC-V 指令学习	25
1. 基本知识	25
2. RISC-V 基础整数指令集——RV32I	26
3. RISC-V 乘法除法指令扩展——RV32M	27
4. RISC-V 浮点数扩展——RV32F 和 RV32D	27
(二) -static 选项探索	28

（三） 汇编编程	28
1. 程序 1——数组、除法、for、输出	28
2. 程序 2——函数调用、输入、输出	32
3. 程序 3——浮点数、if 分支判断、输入、输出	34
五、 总结	37
六、 GitHub 链接	38

一、 预备工作以及实验平台

(一) 分工情况

刘俊彤：语言处理系统处理过程，RISC-V 汇编编程

石家伊：语言处理系统处理过程，LLVM IR 编程

(二) 刘俊彤实验平台

设备名称	liu-virtual-machine
系统名称	Ubuntu 22.04.4
操作系统类型	Linux 64 位
vscode	1.93.1
虚拟机	VMware

表 1: 刘俊彤实验平台信息

(三) 石家伊实验平台

设备名称	yayaa-virtual-machine
系统名称	Ubuntu 22.04.4
操作系统类型	Linux 64 位
vscode	1.93.1
虚拟机	VMware

表 2: 石家伊实验平台信息

二、 语言处理系统处理过程——石家伊

从 c 程序到一个可执行的二进制目标程序，需要经过预处理、编译、汇编、链接加载四个阶段。其中的编译器的编译工作，又分为词法分析、语法分析、语义分析、中间代码生成、代码生成六个阶段，最终生成 c 程序对应的汇编语言代码。这一部分将通过一个简单的计算打印斐波那契数列程序，观察各阶段的输出，进行探索学习。

(一) 编写 c 程序

编写一个计算并打印斐波那契数列的 c 语言代码作为源程序。该源代码 main.c 内容如下：

```
1 #include <stdio.h>
2 #include <time.h> // 用于计时
3 #define test "这是一个宏定义~"
4 clock_t start, stop;
5 double duration;
6
7 int main() {
```

```
8     start = clock();
9
10    printf("%s\n", test);
11
12    int a, b, i, t, n;
13    // 常量折叠, 观察优化
14    a = 0 + 1 - 1;
15    b = 1;
16    i = 1;
17    // 一个从未被使用的变量, 观察优化
18    int c = 2000;
19
20    // 使用scanf来读取用户输入
21    scanf("%d", &n);
22    // 使用printf输出ab两个斐波那契数
23    printf("%d\n", a);
24    printf("%d\n", b);
25
26    // 输出后续的斐波那契数
27    while (i < n) {
28        t = b;
29        b = a + b;
30        printf("%d\n", b);
31        a = t;
32        i = i + 1;
33    }
34
35    // 一段死代码, 观察优化
36    if(0==1){ printf("%s\n", "这不对吧? ");}
37
38
39    stop = clock();
40    duration = (double)(stop - start) / CLOCKS_PER_SEC;
41    printf("程序运行时间为: ");
42    printf("%f\n", duration);
43
44    return 0;
45 }
```

这段代码读入一个 n , 计算并打印 n 个斐波那契数列。为了更加明显的观察到后续对代码的处理, 这段代码中做了一些特别准备:

- 为了测试宏定义字符替换, 编写 'define test' 这是一个宏定义~", 直接打印 test, 查看输出内容是否为" 这是一个宏定义~";
- 为了测试后续代码优化后, 程序的运行时间是否有提升, 添加了 clock 计时器来记录程序的运行时间;
- 为测试常量折叠, 使用 $a = 0 + 1 - 1$ 代替了直接将'0' 赋值给 a;

- 为查看未被使用的变量是否会被去除，声明一个变量 c 为 2000，但后续未使用；
- 为测试死代码过滤，添加了 'if(0==1)' 的语句块；

(二) 预处理器

预处理阶段会处理预编译指令，包括绝大多数的 开头的指令，如 `include`、`define`、`if` 等等，对 `include` 指令会替换对应的头文件，对 `define` 的宏命令直接替换相应内容，同时会删除注释，添加行号和文件名标识。

使用-E 选项只进行预处理得到 main.i 预处理后的文件: gcc main.c -E -o main.i

预处理后生成 main.i 文件，代码膨胀到 1075 行。如图1所示，文件开始的一大部分都是头文件加载进来的内容，可以看到预处理器进行了头文件展开。

```

1  # "main.c"
2  # "main.asm"
3  # "main.lib"
4  # "main.obj"
5  # "main.o"
6  # "main.o"
7  # "main.o"
8  # "main.o"
9  # "main.o"
10 # "main.o"
11 # "main.o"
12 # "main.o"
13 # "main.o"
14 # "main.o"
15 # "main.o"
16 # "main.o"
17 # "main.o"
18 # "main.o"
19 # "main.o"
20 # "main.o"
21 # "main.o"
22 # "main.o"
23 # "main.o"
24 # "main.o"
25 # "main.o"
26 # "main.o"
27 # "main.o"
28 # "main.o"
29 # "main.o"
30 # "main.o"
31 # "main.o"
32 # "main.o"
33 # "main.o"
34 # "main.o"
35 # "main.o"
36 # "main.o"
37 # "main.o"
38 # "main.o"
39 # "main.o"
40 # "main.o"
41 # "main.o"
42 # "main.o"
43 # "main.o"
44 # "main.o"
45 # "main.o"
46 # "main.o"
47 # "main.o"
48 # "main.o"
49 # "main.o"
50 # "main.o"
51 # "main.o"
52 # "main.o"
53 # "main.o"
54 # "main.o"
55 # "main.o"
56 # "main.o"
57 # "main.o"
58 # "main.o"
59 # "main.o"
60 # "main.o"
61 # "main.o"
62 # "main.o"
63 # "main.o"
64 # "main.o"
65 # "main.o"
66 # "main.o"
67 # "main.o"
68 # "main.o"
69 # "main.o"
70 # "main.o"
71 # "main.o"
72 # "main.o"
73 # "main.o"
74 # "main.o"
75 # "main.o"
76 # "main.o"
77 # "main.o"
78 # "main.o"
79 # "main.o"
80 # "main.o"
81 # "main.o"
82 # "main.o"
83 # "main.o"
84 # "main.o"
85 # "main.o"
86 # "main.o"
87 # "main.o"
88 # "main.o"
89 # "main.o"
90 # "main.o"
91 # "main.o"
92 # "main.o"
93 # "main.o"
94 # "main.o"
95 # "main.o"
96 # "main.o"
97 # "main.o"
98 # "main.o"
99 # "main.o"
100 # "main.o"

```

图 1: main.i 开头

文件结尾才为主函数内容，如图2所示。可以发现宏定义 `test` 的内容被替换，程序中的注释被删除，但换行和空格等格式保留，且添加了一些行号、标识符、文件名等信息。

```

0032 //编译选项: g++ 11.2.0
0033
0034 int main() {
0035     start = clock();
0036
0037     printf("ks\n", "这是一个宏定义~");
0038
0039     int a, b, i, t, n;
0040
0041     a = 0 + 1 - 1;
0042     b = 1;
0043     i = 1;
0044
0045     int c = 2000;
0046
0047
0048     scanf("%d", &n);
0049
0050     printf("%d\n", a);
0051     printf("%d\n", b);
0052
0053     while (i < n) {
0054         t = b;
0055         b = a + b;
0056         printf("%d\n", b);
0057         a = t;
0058         i = i + 1;
0059     }
0060
0061     if(0==1){ printf("ks\n", "这不可谓不~");}
0062
0063
0064
0065     stop = clock();
0066     duration = (double)(stop - start) /
0067     # 40 "main.c" } 4
0068
0069     # 40 "main.c" (((_clock_t) 1000000))
0070
0071     ;
0072     printf("程序运行时间为:");
0073     printf("%f\n", duration);
0074
0075     return 0;
0076 }

```

图 2: main.i 结尾主函数

(三) 编译器

编译过程分为六步：词法分析、语法分析、语义分析、中间代码生成、代码优化和代码生成六个阶段。下面对每个阶段的工作进行探索。

1. 词法分析

词法分析阶段将源程序转换为单词序列。

通过指令 `clang -E -Xclang -dump-tokens main.c` 获得 token 序列。如图3所示, 图中为 `main` 函数开始处的 token 序列。

```

identifier 'main' [LeadingSpace] Loc=<main.c:7:5>
l_paren '(' [Loc=<main.c:7:9>]
r_paren ')' [Loc=<main.c:7:10>]
l_brace '{' [LeadingSpace] Loc=<main.c:7:12>
identifier 'start' [StartOfLine] [LeadingSpace] Loc=<main.c:8:5>
equal '=' [LeadingSpace] Loc=<main.c:8:11>
identifier 'clock' [LeadingSpace] Loc=<main.c:8:13>
l_paren '(' [Loc=<main.c:8:18>]
r_paren ')' [Loc=<main.c:8:19>]
semi ';' [Loc=<main.c:8:20>]
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:10:5>
l_paren '(' [Loc=<main.c:10:11>]
string_literal '"%s\n"' [Loc=<main.c:10:12>]
comma ',' [Loc=<main.c:10:18>]
string_literal '"这是一个宏定义~"' [LeadingSpace] Loc=<main.c:10:20 <
Spelling=main.c:3:14>>
r_paren ')' [Loc=<main.c:10:24>]
semi ';' [Loc=<main.c:10:25>]
int 'int' [StartOfLine] [LeadingSpace] Loc=<main.c:12:5>
identifier 'a' [LeadingSpace] Loc=<main.c:12:9>
comma ',' [Loc=<main.c:12:10>]
identifier 'b' [LeadingSpace] Loc=<main.c:12:12>
comma ',' [Loc=<main.c:12:13>]
identifier 'i' [LeadingSpace] Loc=<main.c:12:15>
comma ',' [Loc=<main.c:12:16>]
identifier 't' [LeadingSpace] Loc=<main.c:12:18>
comma ',' [Loc=<main.c:12:19>]
identifier 'n' [LeadingSpace] Loc=<main.c:12:21>
semi ';' [Loc=<main.c:12:22>]
identifier 'a' [StartOfLine] [LeadingSpace] Loc=<main.c:14:5>
equal '=' [LeadingSpace] Loc=<main.c:14:7>
numeric_constant '0' [Loc=<main.c:14:9>]
plus '+' [Loc=<main.c:14:11>]
numeric_constant '1' [Loc=<main.c:14:13>]
minus '-' [Loc=<main.c:14:15>]
numeric_constant '1' [Loc=<main.c:14:17>]
semi ';' [Loc=<main.c:14:18>]
identifier 'b' [StartOfLine] [LeadingSpace] Loc=<main.c:15:5>
equal '=' [Loc=<main.c:15:7>]
numeric_constant '1' [Loc=<main.c:15:9>]
semi ';' [Loc=<main.c:15:10>]
identifier 'i' [StartOfLine] [LeadingSpace] Loc=<main.c:16:5>
equal '=' [Loc=<main.c:16:7>]
numeric_constant '1' [Loc=<main.c:16:9>]

```

图 3: token 序列

图中部分详细内容如下:

```

1 int 'int' [StartOfLine] Loc=<main.c:7:1>
2 identifier 'main' [LeadingSpace] Loc=<main.c:7:5>
3 l_paren '(' Loc=<main.c:7:9>
4 r_paren ')' Loc=<main.c:7:10>
5 l_brace '{' [LeadingSpace] Loc=<main.c:7:12>
6 identifier 'start' [StartOfLine] [LeadingSpace] Loc=<main.c:8:5>
7 equal '=' [LeadingSpace] Loc=<main.c:8:11>
8 identifier 'clock' [LeadingSpace] Loc=<main.c:8:13>
9 l_paren '(' Loc=<main.c:8:18>
10 r_paren ')' Loc=<main.c:8:19>
11 semi ';' Loc=<main.c:8:20>
12 identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:10:5>
13 l_paren '(' Loc=<main.c:10:11>
14 string_literal '"%s\n"' Loc=<main.c:10:12>
15 comma ',' Loc=<main.c:10:18>
16 string_literal '"这是一个宏定义~"' [LeadingSpace] Loc=<main.c:10:20 <
    Spelling=main.c:3:14>>
17 r_paren ')' Loc=<main.c:10:24>
18 semi ';' Loc=<main.c:10:25>

```

可以看到词法分析将源代码中的每一个语句都拆成了逐个单词、符号的 token 序列流, 并标明了其类别以及在源代码中的位置。

2. 语法分析

语法分析阶段将词法分析生成的词法单元来构建抽象语法树 (AST)。

通过指令 `clang -E -Xclang -ast-dump main.c` 使用 LLVM 获取相应的 AST。如图4所示,该方法较 gcc 生成的可读性更强,具有明显的层次结构。

[illegible]

图 4: AST

通过指令 `gcc -fdump-tree-original-raw -o ast main.c` 会生成两个文件，一个不可读的二进制文件'ast'，和一个文本格式的 AST 输出：ast-main.c.005t.original，文件内容如5所示。

[illegible]

图 5: 文本格式 AST

3. 语义分析

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。

这一部分没有明确的指令可以看到处理阶段。根据课程学习的内容，可以知道语义分析阶段进行的工作包括进行类型检查、作用域检查、符号表维护等操作。具体比如检查表达式的类型，函数调用参数的类型，变量是否在正确的作用域内声明等。

4. 中间代码生成

语义分析完成后，中间代码生成会将语法树转成中间代码——三地址码。

clang 指令生成 LLVM IR:

通过指令 `clang -S -emit-llvm main.c` 生成 LLVM IR 文件 `main.ll`。文件内容如图6所示。

[illegible]

图 6: LLVM IR: main.ll

LLVM IR 是三地址码的一种形式，每条指令最多包含三个操作数，即两个源操作数和一个目标操作数。对于中间代码将在第二部分继续学习探究。

gcc 获得多阶段输出:

通过指令 `gcc -fdump-tree-all-graph main.c` 获得中间代码生成的多阶段的输出。指令生成了多个文件，文件名有顺序标号。其中的 `.dot` 文件可以用 `graphviz` 可视化，获取控制流图。下面选取几个 `.dot` 文件进行查看。

1、 a-main.c.015t.cfg.dot

这里可以看出 a 常量折叠被优化，直接写了 `a = 0`，并且没有死代码的 `if(0 == 1)` 分支。说明虽然还没有到代码优化的阶段，但到中间代码生成这一阶段时已经对程序做了简单的优化。

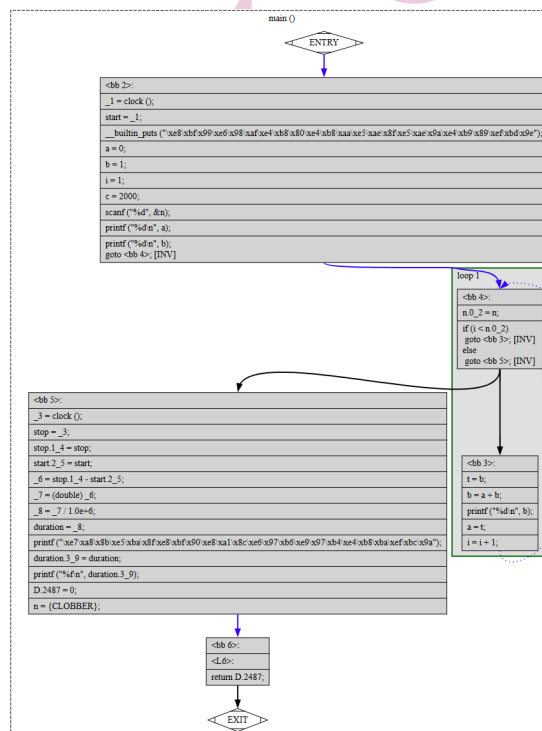


图 7: 控制流图 1

2、 a-main.c.244t.optimized.dot

对比上一个流程图，这里为变量添加下划线数字标识，用来说明不同阶段变量值改变。并且循环外变量 a、b、i 在循环体内被创建了副本并加以标注。

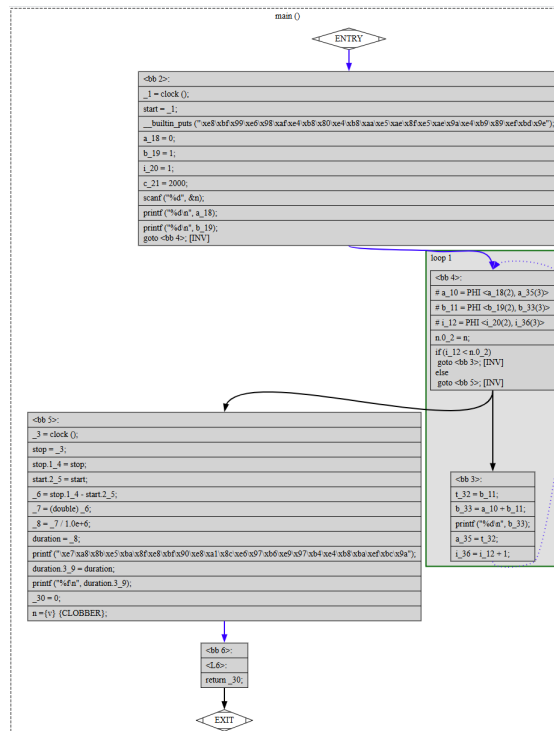


图 8: 控制流图 2

5. 代码优化

代码优化阶段进行与机器无关的代码优化步骤改进中间代码，生成更好的目标代码。

先使用指令 `llc -print-before-all -print-after-all main.ll > a.log 2>&1` 生成每个 pass 后生成的 LLVM IR, 得到 `main.s` 和 `a.log` 文件, 分别为优化后生成的汇编文件和文件。为便于区分, 将 `main.s` 重命名为 `main.llc.s`。

日志文件中包含多次 pass 后的 LLVM IR，与一些标注优化的文字，部分内容如图9所示。

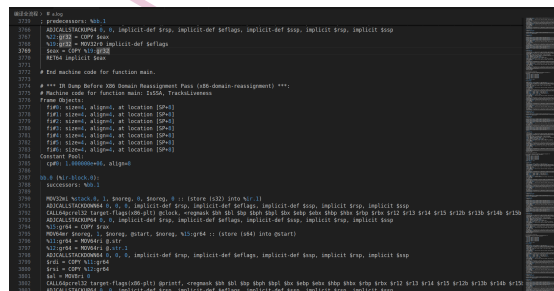


图 9: a.log

接下来来尝试一些特定的 pass，测试他们是否能提升代码的运行时间。

先使用 `llvm-as main.ll -o main.bc` 将 `main.ll` 转换为 `bc` 二进制代码形式的 `main.bc`。

在 LLVM 官网中可以看到 pass 被分为三类：Analysis Passes、Transform Passes 和 Utility Passes。Analysis Passes 用于分析或计算某些信息，以便给其他 pass 使用，如计算支配边界、控制流图的数据流分析等；Transform Passes 都会通过某种方式对中间代码形式的程序做某种变化，如死代码删除，常量传播等。

从 Analysis Passes 和 Transform Passes 两类中各选择两个 pass 进行尝试，通过比较最终生成的可执行代码的运行时间，来衡量优化效果。

Analysis Passes 优化：

对于 Analysis Passes 优化，选用了以下两个优化选项：

- -basic-aa (Basic Alias Analysis)：别名分析，用于确定不同指针或内存引用是否指向同一块内存。如果两个指针不指向同一内存位置，编译器可以对它们各自的操作进行更多的优化。
- -scalar-evolution (Scalar Evolution Analysis)：分析标量变量（尤其是循环中的变量）如何随时间变化。主要用于循环优化，帮助 LLVM 理解循环中的变量值是如何演变的，从而可以进行更高级的优化，比如循环展开、循环向量化等。

生成可执行文件指令如下：

```

1 不 进 行 优 化
2 llc main.bc -filetype=obj -relocation-model=pic -o main.o //同时编译和汇编
   LLVM bitcode 生成main.o
3 gcc main.o -o main -fPIE //连接加载得到可执行文件
4
5 基 本 别 名 分 析 优 化
6 opt -basic-aa main.bc -o main_baa.bc
7 llc main_baa.bc -filetype=obj -relocation-model=pic -o main_baa.o
8 gcc main_baa.o -o main_baa -fPIE
9
10 标 量 演 变 分 析 优 化
11 opt -scalar-evolution main.bc -o main_sea.bc
12 llc main_sea.bc -filetype=obj -relocation-model=pic -o main_sea.o
13 gcc main_sea.o -o main_sea -fPIE

```

运行三种可执行文件，输入不同规模的数据，对比运行时间，如表3所示。

表 3: Analysis Passes 多种优化

数据规模	未优化/s	-basic-aa 优化/s	-scalar-evolution 优化/s
$1 * 10^2$	0.001380	0.000870	0.001393
$1 * 10^3$	0.007498	0.007281	0.006073
$1 * 10^4$	0.071290	0.067091	0.071855
$1 * 10^5$	0.708506	0.650142	0.643686
$1 * 10^6$	7.094876	6.826311	6.958539

计算加速比，绘制折线图10。可以看到二者虽然有一定的优化效果，但除去 basic-aa 在规模 $1*10^2$ 上的表现稍高，其余无论数据规模大小，程序的加速比都不高，主要在 1.2 左右。推测可能是 Analysis Passes 主要是为其它种类 pass 做分析与准备工作的，所以单独使用这类 pass 的优化效果并不理想。

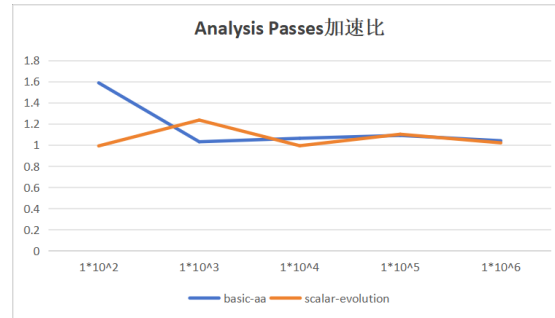


图 10: Analysis Passes 优化加速比

Transform Passes 优化:

对于 Transform Passes 优化，选用了以下两个优化选项：

- -loop-unroll (Unroll loops)：循环展开，用于将循环体中的操作复制多次，减少循环的迭代次数，从而提高程序执行的效率。
- -dce (Dead code elimination)：死代码消除，是一种移除程序中永远不会被执行或对程序结果没有影响的代码的优化技术。通过消除这些代码，可以减小程序的体积，并提高执行效率。

Transform Passes 优化使用的指令如下：

```

1  _____ 不进行优化
2  llc main.bc -filetype=obj -relocation-model=pic -o main.o //同时编译和汇编
   LLVM bytecode 生成main.o
3  gcc main.o -o main -fPIE //连接加载得到可执行文件
4
5  _____ Unroll loops 优化
6  opt -loop-unroll main.bc -o main_lu.bc
7  llc main_lu.bc -filetype=obj -relocation-model=pic -o main_lu.o
8  gcc main_lu.o -o main_lu -fPIE
9
10 _____ Dead code elimination 优化
11 opt -dce main.bc -o main_dce.bc
12 llc main_dce.bc -filetype=obj -relocation-model=pic -o main_dce.o
13 gcc main_dce.o -o main_dce -fPIE

```

运行三种可执行文件，输入不同规模的数据，对比运行时间，如表4所示。

表 4: Transform Passes 优化

数据规模	未优化/s	-loop-unroll 优化	-dce 优化/s
1 * 10 ²	0.001380	0.000709	0.000684
1 * 10 ³	0.007498	0.006453	0.006592
1 * 10 ⁴	0.071290	0.067441	0.070424
1 * 10 ⁵	0.708506	0.691657	0.698041
1 * 10 ⁶	7.094876	6.891539	6.924775

计算加速比，绘制折线图11。可以看到二者有一定的优化效果，但出去在规模 1×10^2 上的表现稍高，其余的数据规模下，程序的加速比都不高，只是比 1 高一点点。推测可能是因为该程序比较简单，优化空间小。

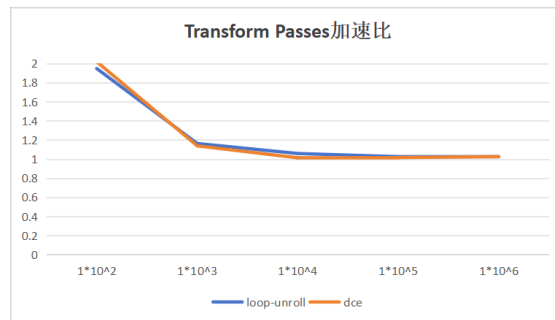


图 11: Transform Passes 加速比

6. 代码生成

代码生成阶段以中间表示形式作为输入，将其映射到目标语言。

通过 `gcc main.i -S -o main.S` 得到源程序对应的汇编代码 `main.s`。

(四) 汇编器

汇编过程实际上把汇编语言程序代码翻译成目标机器指令的过程。其最终生成的是可重定位的机器代码。

通过指令 `gcc main.S -c -o main_step.o` 可以通过汇编器将生成的汇编代码转为机器码文件 `main_step.o`。

汇编器的主要功能是将汇编语言中的助记符 (mnemonics) 转换为机器指令。每一条汇编指令对应一个或多个机器指令。

这一过程中，汇编语言通常使用符号 (如标签和变量名) 来表示内存位置和数据，汇编器会解析这些符号，并为它们分配内存地址。汇编器会生成并维护一个符号表，在指令执行过程中，当引用到符号时，汇编器从符号表中查找符号对应的内存地址或偏移量。

当程序中涉及外部符号 (如外部函数或变量) 时，汇编器会生成重定位信息，标识需要在链接阶段解决的外部符号。在汇编过程中，外部符号的地址是未知的，汇编器会生成重定位表，将相关信息记录下来，交由链接器处理。

汇编器还通常将程序分为不同的段，每个段中包含不同类型的内容，如代码段 `.text`，数据段 `.data`，未初始化数据段 `.bss` 等等。

汇编器处理结束后，生成的文件虽然是机器语言，但还不能执行，需要链接器加载器和加载器的处理。

(五) 链接器加载器

由汇编程序生成的目标文件不能够直接执行。大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。进而链接器对该机器代码进行执行生成可执行文件。

使用指令 `gcc main_step.o -o main_step` 得到可执行文件 `main_step`。

输入 `./main_step` 执行时，加载器将二进制文件载入内存，程序成功执行。

程序运行结果如图12所示。

```

yayaa@yayaa-virtual-machine:~/labs/lab1/编译全流程$ ./main_step
这是一个宏定义~
6
0
1
1
2
3
5
8
程序运行时间为: 0.000278

```

图 12: 程序运行结果

三、 LLVM IR 编程——石家伊

在这一部分中,我选择了对 LLVM IR 进行进一步的学习探索。通过编写三个简单的 c 程序对应的 LLVM 代码,来对这一语言的语法结构进行学习探索。

(一) 学习 LLVM IR

在第一部分的实验流程中,可以获取到斐波那契数列源代码所对应的中间代码文件 main.ll。在了解一些基本知识后,我选择通过逐行分析 main.ll 来学习 LLVM IR 的语法特点。

代码内容与注释如下。

```

1 ; 开头的注释说明LLVM IR 的基本单位称为 module, 标注了该module的名称
2 ; ModuleID = 'main.c'
3
4 ; 标明源程序为"main.c"
5 source_filename = "main.c"
6
7 ; 目标架构的数据布局方式。这个描述了如何在内存中排列数据, 如整数和浮点数的大
  小、对齐方式等。
8 target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
  :16:32:64-S128"
9 ; 定义目标系统的三元组
10 target triple = "x86_64-pc-linux-gnu"
11
12 ; 这一部分先对程序的常量、全局变量; 字符串; 格式化字符;进行了声明。
13 ; 为了便于查看, 调整了声明顺序
14 ; 1、全局变量
15 ; 用于计时的clock_t型全局变量, 可以看到是用64位整数来实现
16 ; dso_local表示这个全局变量在当前的编译单元中是本地的, 不会被其他编译单元引
  用。
17 @start = dso_local global i64 0, align 8
18 @stop = dso_local global i64 0, align 8
19 ; 一个全局双精度浮点变量, 初始值为0.0,用于存储程序运行的时间结果
20 @duration = dso_local global double 0.000000e+00, align 8
21 ; 2、字符串与格式化字符 (宏定义里那个字符串也在这里)
22 ; 用于打印的字符串与格式化字符都被声明为常量, 常量名按顺序标号
23 ; private 表示该字符串常量仅在当前模块内可见, 其他模块无法引用它。
24 ; unnamed_addr 表示LLVM可以自由地优化该常量的地址, 假设多个常量具有相同的内
  容, LLVM可以将它们共享相同的内存地址。
25 ; constant 表示这个是一个不可修改的常量。

```

```

26 ; 对字符串用数组实现，并且都在末尾有“\00”作为字符串结束标识
27 @.str = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
28 @.str.1 = private unnamed_addr constant [25 x i8] c"\E8\BF\99\E6\98\AF\E4\B8
    \80\E4\B8\AA\E5\AE\8F\E5\AE\9A\E4\B9\89\EF\BD\9E\00", align 1
29 ; str2与3可以观察到对于不同的格式化字符是单独声明的，即“%d”与“%d\n”是不相
    关的，并不是在使用的时候拼接“%-”与“\n”
30 @.str.2 = private unnamed_addr constant [3 x i8] c"%d\00", align 1
31 @.str.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
32 @.str.4 = private unnamed_addr constant [25 x i8] c"\E7\A8\8B\E5\BA\8F\E8\BF
    \90\E8\A1\8C\E6\97\B6\E9\97\B4\E4\B8\BA\EF\BC\9A\00", align 1
33 @.str.5 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
34
35 ; Function Attrs: noinline nounwind optnone uwtable
36 ; 定义main函数
37 ; #0 是属性编号(属性声明在末尾)
38 define dso_local i32 @main() #0 {
39     ; 声明局部变量，源代码中有a, b, i, t, n, c
40     %1 = alloca i32, align 4 ; 在本次实验编译出的所有ll文件中都存在，一个为0的
        %1, 零寄存器?
41     %2 = alloca i32, align 4 ; a
42     %3 = alloca i32, align 4 ; b
43     %4 = alloca i32, align 4 ; i
44     %5 = alloca i32, align 4 ; t
45     %6 = alloca i32, align 4 ; n
46     %7 = alloca i32, align 4 ; c
47
48     store i32 0, i32* %1, align 4
49     ; 调用clock函数，之前看到start被声明为全局变量。这里将函数调用先存入%8临时
        寄存器，再存入start，更贴近底层逻辑
50     %8 = call i64 @clock() #3
51     store i64 %8, i64* @start, align 8
52
53     ; 调用printf函数，(i8*, ...)表示 printf 函数接受一个字符指针 (i8*) 作为第一
        个参数，后面可以有任意数量的可变参数 (...)
54     ; 传入了两个参数。传入参数格式如下：
55     ; i8* noundef getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64
        0)
56     ; i8* 是传递给 printf 的参数类型
57     ; noundef 是一个修饰符，表示这个指针参数不能是未定义值.如
58     ; getelementptr inbounds 是LLVM中常用的指令，用于计算数组或结构体中某个元素
        的指针。
59     ; inbounds 确保访问是有效的，超出范围会触发未定义行为
60     ; [4 x i8]: 是 getelementptr 操作的类型参数，要对这样一个数组进行操作
61     ; [4 x i8]* @.str: 这里的[4 x i8]*是一个指针类型，指向@.str，是实际需要被操
        作的数组。
62     ; i64 0, i64 0: 第0个元素的，第0个字节处，即取字符串的起始地址。
63     %9 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
        i8], [4 x i8]* @.str, i64 0, i64 0), i8* noundef getelementptr inbounds

```



```

    ([25 x i8], [25 x i8]* @.str.1, i64 0, i64 0))
64
65 ; 为局部变量赋值
66 ; a用于测试常量折叠的优化, 这里可以看到。直接没有运算+1-1把0给了%2
67 store i32 0, i32* %2, align 4
68 store i32 1, i32* %3, align 4
69 store i32 1, i32* %4, align 4
70 ; 这里是原本的c, 虽然没有使用过该变量, 也没有被去除
71 store i32 2000, i32* %7, align 4
72
73 ; scanf函数的调用, 第二个参数变为一个32位整形指针, 指向%6, 即n
74 %10 = call i32 (@__isoc99_scanf(i8* noundef getelementptr
    inbounds ([3 x i8], [3 x i8]* @.str.2, i64 0, i64 0), i32* noundef %6)
75 ; 打印整型局部变量, 是将其放入另一个局部寄存器在打印
76 ; LLVM 是基于静态单赋值形式 (SSA) 的中间表示, 每个寄存器只能在赋值后使用一
    次, 且一旦赋值, 不可更改。
77 ; 因此, 局部变量值首先存储在内存中的某个位置 (如 %2), 然后在打印之前需要通
    过 load 指令加载该值, 并将其存储到一个新的虚拟寄存器 %11 中。%11 实际
    上 是 SSA 中的一个唯一名称标识符, 用于跟踪变量的值。
78 %11 = load i32, i32* %2, align 4
79 %12 = call i32 (@__printf(i8* noundef getelementptr inbounds ([4 x
    i8], [4 x i8]* @.str.3, i64 0, i64 0), i32 noundef %11)
80 %13 = load i32, i32* %3, align 4
81 %14 = call i32 (@__printf(i8* noundef getelementptr inbounds ([4 x
    i8], [4 x i8]* @.str.3, i64 0, i64 0), i32 noundef %13)
82
83 ; br是跳转指令, 这里无条件。开始循环前要进行条件判断, %15就是这样一个代码块
84 br label %15
85
86 15:
87 ; 循环条件判断
88 ; 先加载i, n的值到临时寄存器
    ; preds = %19, %0
89 %16 = load i32, i32* %4, align 4 ; i
90 %17 = load i32, i32* %6, align 4 ; n
91 ; icmp用于整数比较, slt是小于。小于则置为1
92 %18 = icmp slt i32 %16, %17
93 ; i1 %18表示一个布尔值, 作为条件。为真跳转到%19, 否则%29
94 br i1 %18, label %19, label %29
95
96 19:
97 ; 循环体 ; preds = %15
98 ; t = b: 将%20作为中间寄存器, 取出%3中b的值, 存入%5的t中
99 %20 = load i32, i32* %3, align 4
100 store i32 %20, i32* %5, align 4
101
102 ; b = a + b: 将a、b从内存中取出放入两个临时寄存器
103 %21 = load i32, i32* %2, align 4

```



```

104 %22 = load i32, i32* %3, align 4
105 ; 计算a+b: add为加法指令; nsw表示 "No Signed Wrap", 即无符号溢出, 编译器假
    定该操作不会导致有符号整数溢出。
106 %23 = add nsw i32 %21, %22
107 ; 更新内存中b的值, 为本次计算得到的斐波那契数
108 store i32 %23, i32* %3, align 4
109
110 ; 打印更新后的b值
111 %24 = load i32, i32* %3, align 4
112 %25 = call i32 (@i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
    i8], [4 x i8]* @.str.3, i64 0, i64 0), i32 noundef %24)
113
114 ; a = t: 将%26作为中间寄存器, 取出%5中t的值, 存入%2的a中
115 %26 = load i32, i32* %5, align 4
116 store i32 %26, i32* %2, align 4
117
118 ; 取出i, 加上1, 再将新值放进去
119 %27 = load i32, i32* %4, align 4
120 %28 = add nsw i32 %27, 1
121 store i32 %28, i32* %4, align 4
122
123 ; 跳转回%15进行条件判断。
124 ; !llvm.loop !6 : 一个元数据。!llvm.loop 表示该分支指令与循环有关; !6 是指
    向元数据的引用, 该元数据包含了与循环优化相关的信息。
125 br label %15, !llvm.loop !6
126
127 29: ; preds = %15
128 ; 调用clock函数, 记录stop
129 %30 = call i64 @clock() #3
130 store i64 %30, i64* @stop, align 8
131 ; 计算stop-start
132 %31 = load i64, i64* @stop, align 8
133 %32 = load i64, i64* @start, align 8
134 %33 = sub nsw i64 %31, %32 ; 减法sub
135
136 ; sitofp 指令用于将一个有符号整数转换为浮点数, 有符号转换
137 %34 = sitofp i64 %33 to double
138 ; fdiv 是浮点数的除法操作,%34中的double除以1*10的六次方
139 %35 = fdiv double %34, 1.000000e+06
140 ; 时间结果存入duration
141 store double %35, double* @duration, align 8
142 ; 打印字符串与时间
143 %36 = call i32 (@i8*, ...) @printf(i8* noundef getelementptr inbounds ([25 x
    i8], [25 x i8]* @.str.4, i64 0, i64 0))
144 %37 = load double, double* @duration, align 8
145 %38 = call i32 (@i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
    i8], [4 x i8]* @.str.5, i64 0, i64 0), double noundef %37)
146

```

```

147 ; main函数的返回值为0
148 ret i32 0
149 }
150
151 ; Function Attrs: nounwind
152 ; 函数声明
153 declare i64 @clock() #1
154
155 declare i32 @printf(i8* noundef, ...) #2
156
157 declare i32 @__isoc99_scanf(i8* noundef, ...) #2
158
159 ; 函数属性：用于描述函数在编译期间的行为和目标平台的特性。
160 attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "
    min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-
    buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx
    ,+sse,+sse2,+x87" "tune-cpu"="generic" }
161 attributes #1 = { nounwind "frame-pointer"="all" "no-trapping-math"="true" "
    stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features
    "="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
162 attributes #2 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-
    protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8
    ,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
163 attributes #3 = { nounwind }
164
165 !llvm.module.flags = !{!0, !1, !2, !3, !4} ; 声明一些标识，用于记录与整个模
    块相关的编译器设置，通常包括一些平台和环境特定的标志：
166 !llvm.ident = !{!5} ;生成该 LLVM IR 的编译器信息
167
168 !0 = !{i32 1, !"wchar_size", i32 4}
169 !1 = !{i32 7, !"PIC Level", i32 2}
170 !2 = !{i32 7, !"PIE Level", i32 2}
171 !3 = !{i32 7, !"uwtable", i32 1}
172 !4 = !{i32 7, !"frame-pointer", i32 2}
173 !5 = !{!"Ubuntu clang version 14.0.0-1ubuntu1.1"}
174 ; 之前用到的元数据!6，用于标识循环属性。!7中的标识说明这是一个必须有进展的循
    环（不能死循环）
175 !6 = distinct !{!6, !7}
176 !7 = !{!"llvm.loop.mustprogress"}

```

根据上述.ll 文件，我们对 LLVM IR 的特性做以下总结：

1. 首句注释说明 LLVM IR 的基本单位称为 module，生成的文件中标注了该 module 的名称，并且标注了源程序文件的名称。
2. 这个被生成的文件中还标注了目标架构的数据布局方式、目标系统的三元组信息。
3. llvm IR 中注释以 “;” 开头。
4. llvm IR 是基于静态单赋值形式（SSA）的中间表示，即每个值的类型在编写时是确定的。

i1, i32, i64 都是数据类型, 用 i+ 比特位数表示。但也有 float, double 等浮点数类型, void, label 等数据类型。

5. llvm IR 中全局变量和函数都以 @ 开头, 且会在类型 (如 i32) 之前用 global 标明, 局部变量以 % 开头, 其作用域是单个函数, 临时寄存器 (上文中的 %1 等) 以升序阿拉伯数字命名。用于打印的字符串与格式化字符都被单独声明为常量, 常量名按顺序标号, 如 @.str, @.str.1, 字符串会以字符数组的形式存储。
6. 在数据声明时会为数据标注一些属性值, 如 dso_local 表示这个全局变量在当前的编译单元中是本地的; align 属性用于指定变量、指针或数据在内存中的对齐方式。这些属性可以让编译器对程序进行更加大胆的优化。
7. 函数定义: define + 返回值 (i32) + 函数名 (@main) + 参数列表 ((i32 %a, i32 %b)) + 函数体 (ret i32 0)。函数也会被标注属性, 如 0, 1。这些属性的具体内容在结尾处。
8. 对于调用的库函数会在结尾处 declare 声明。调用函数的语法时先声明一个函数返回值类型的临时寄存器, 寄存器 = call + 返回值 (i32) + 函数名 (@clock) + 参数列表 ((i32 %a, i32 %b))。函数的调用返回值将存储在寄存器中。
9. 绝大多数指令的含义就是其字面意思, load 从内存读值, store 向内存写值, add 相加参数, alloca 分配内存并返回地址等。除这些以外 getelementptr inbounds 是 LLVM 中常用的指令, 用于计算数组或结构体中某个元素的指针, 在数组遍历、取值时会经常被使用。
10. br 指令单独使用是无条件跳转, 在后面加上判断条件 (一般是一个布尔值) 就会变成条件跳转, 为真则跳转到前一个标签, 否则跳转到后一个。
11. icmp 指令经常与 br 结合使用, 按照给定的规则如 slt 等比较两个操作数, 并将结果存入临时寄存器。
12. llvm 中存在一种元数据 (Metadata), 是用于为指令、全局变量或函数附加额外信息的机制, 但这些信息不会直接影响程序的语义或生成的代码。主要用于调试信息、优化提示以及其他编译器相关的辅助信息。如 !llvm.loop !6 中 !llvm.loop 表示该分支指令与循环有关; !6 是指向元数据的引用, 该元数据包含了与循环优化相关的具体信息。这些元数据会在文章结尾处声明。
13. 终结指令一定位于一个基本块的末尾, 如 ret 指令会令程序控制流返回到函数调用者。

(二) LLVM IR 编程

1. 程序 1——数组, 除法, for 循环, 打印

第一个程序将数组中的元素打印输出, 涉及到数组、除法、for、输出等语法, 其 c 程序如下:

```

1 #include <stdio.h>
2
3 // 数组 除法 for 循环 打印
4
5 int main() {
6     // 定义一个整型数组, 包含5个元素
7     int numbers[] = {1, 2, 3, 4, 5};
8

```

```

9 // 修改数组的值
10 numbers[1] = 6;
11
12 // 获取数组的大小
13 int size = sizeof(numbers) / sizeof(numbers[0]);
14
15 // 遍历数组并打印每个元素
16 for(int i = 0; i < size; i++) {
17     printf("%d ", numbers[i]);
18 }
19
20 return 0;
21 }

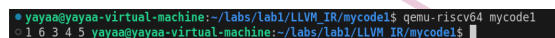
```

编译运行

riscv64-unknown-linux-gnu-gcc mycode1.c -o mycode1 -static

qemu-riscv64 mycode1

得到输出 1 6 3 4 5, 如图13所示。



```

yaya@yaya-virtual-machine:~/labs/lab1/LLVM_IR/mycode1$ qemu-riscv64 mycode1
1 6 3 4 5 yaya@yaya-virtual-machine:~/labs/lab1/LLVM_IR/mycode1$

```

图 13: mycode1

编写其对应的 LLVM IR 代码, 代码内容以及解释如下。

```

1 ; 数组大小不变, 全局处开辟并赋值, 可以在主函数中灵活使用
2 @_.numbers = private unnamed_addr constant [5 x i32] [i32 1, i32 2, i32 3,
   i32 4, i32 5], align 16
3 @.str = private unnamed_addr constant [4 x i8] c"%d \00", align 1
4
5 ; 主函数
6 define dso_local i32 @main() {
7     %1 = alloca i32, align 4
8     store i32 0, i32* %1, align 4
9
10    ; 在栈中为数组开辟空间
11    %2 = alloca [5 x i32], align 16
12    ; 声明两个变量, a分表用于保存数组的长度和当前下标
13    %3 = alloca i32, align 4
14    %4 = alloca i32, align 4
15
16    ; bitcast 类型转换指令。这里将[5 x i32]型转为i8型的指针。便于memcpy函数进行
   字节级别的操作
17    %5 = bitcast [5 x i32]* %2 to i8*
18    ; 调用memcpy函数
19    ; .p0i8.p0i8.i64: 目标地址和源地址分别为两个i8类型的指针, 第三个参数是一个
   64 位整数, 表示要复制的字节数
20    call void @llvm.memcpy.p0i8.p0i8.i64(
21        i8* align 16 %5, ; 目标地址为在栈中开辟的数组

```

```

22     i8* align 16 bitcast ([5 x i32]* @_.numbers to i8*), ; 源地址为全局数
      组, 也用bitcast做了指针类型转换
23     i64 20, ; 共20个字节u需要复制, 数组大小: 32*5 = 160bit/8 = 20字节
24     i1 false) ; 这是一个布尔值参数, 用于指示这次 memcpy 操作是否是“易失性”
      操作。
25         ; false 表示这个操作是“非易失性”的, 意味着这个操作可以被编译
      器优化, 而不会产生其他副作用。
26
27 ; 修改数组的值
28 ; 将numbers[1]的地址存入%6, 修改地址中的值
29 %6 = getelementptr inbounds [5 x i32], [5 x i32]* %2, i64 0, i64 1
30 store i32 6, i32* %6, align 4
31
32 ; 存储数组大小与初始下标零。没有再进行size的计算, 程序已知了数组大小
33 store i32 5, i32* %3, align 4
34 store i32 0, i32* %4, align 4
35
36 ; 跳转到条件判断, 开始for循环
37 br label %7
38
39 7:
40 ; 对比数组大小与当前下标 ;
      preds = %17, %0
41 %8 = load i32, i32* %4, align 4
42 %9 = load i32, i32* %3, align 4
43 ; icmp比较, slt为整数比较, 严格小于
44 %10 = icmp slt i32 %8, %9
45 ; bri条件跳转, %10为真则到%11执行循环体
46 br i1 %10, label %11, label %20
47
48 11:
49 ; 循环体, 打印当前数组元素 ;
      preds = %7
50 %12 = load i32, i32* %4, align 4 ; 取出当前下标值
51 %13 = sext i32 %12 to i64 ; sext 类型转换, 下标转为i64, 便于取地址指针
52 %14 = getelementptr inbounds [5 x i32], [5 x i32]* %2, i64 0, i64 %13 ;
      i64 %13按当前下标取地址指针
53 %15 = load i32, i32* %14, align 4 ; 加载当前地址中的值
54 ; 打印
55 %16 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x
      i8], [4 x i8]* @.str, i64 0, i64 0), i32 noundef %15)
56 ; 调到%17, 进行下标的更新
57 br label %17
58
59 17:
60 ; 更新下标 ; preds = %11
61 %18 = load i32, i32* %4, align 4
62 ; 下标加1, nsw表示编译器可以假设该操作不会发生有符号溢出

```

```

63 %19 = add nsw i32 %18, 1
64 store i32 %19, i32* %4, align 4
65 ; 下标更新后，再次进行条件判断（删除了循环标识）
66 br label %7
67
68 20:
69 ; 程序结束 ; preds = %7
70 ret i32 0
71 }
72
73 ; 函数声明
74 declare void @llvm.memcpy.p0i8.p0i8.i64(i8* noalias nocapture writeonly, i8*
    noalias nocapture readonly, i64, i1 immarg)
75 declare i32 @printf(i8* noundef, ...)

```

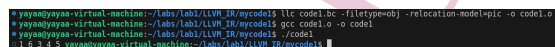
将文件继续编译汇编链接成可执行文件：

```
llvm-as code1.ll -o code1.bc
```

```
llc code1.bc -filetype=obj -relocation-model=pic -o code1.o
```

```
gcc code1.o -o code1
```

./code1 运行程序，如图14所示，程序正确打印出 1 6 3 4 5。



```

yaya@yaya-virtual-machine:~/lab1/LLVM-IR/mycode1$ llc code1.bc -filetype=obj -relocation-model=pic -o code1.o
yaya@yaya-virtual-machine:~/lab1/LLVM-IR/mycode1$ gcc code1.o -o code1
yaya@yaya-virtual-machine:~/lab1/LLVM-IR/mycode1$ ./code1
1 6 3 4 5
yaya@yaya-virtual-machine:~/lab1/LLVM-IR/mycode1$

```

图 14: code1

根据上述.ll 文件，我们对 LLVM IR 的特性做以下总结：

1. 对于大小不变的数组，需要全局处开辟并赋值，然后在主函数使用 memcpy 复制下来，可以进行灵活使用
2. bitcast 是类型转换指令。bitcast 原类型变量名 to 目的类型。
3. 由于运用的 clang 是 14.0.0 版本，代码中都运用了透明指针，所以在使用 memcpy 时较复杂，memcpy 函数的操作以字节级别进行，需要对数组指针进行类型转换。

2. 程序 2——函数调用，输入输出

第二个程序定义函数，将传入的两个参数相加，并在主函数中调用，打印计算结果，涉及到函数调用、输入输出等语法，其 c 程序如下：

```

1 #include <stdio.h>
2
3 // 函数调用 输入输出
4
5
6 // 定义一个函数，该函数接收两个整数作为参数，并返回它们的和
7 int addNumbers(int a, int b) {
8     return a + b; // 返回两个数的和
9 }
10

```

```

11 int main() {
12     int num1; // 定义第一个整数
13     int num2; // 定义第二个整数
14     scanf("%d %d", &num1, &num2);
15
16     // 调用addNumbers函数, 并将结果存储在变量sum中
17     int sum = addNumbers(num1, num2);
18
19     // 打印结果
20     printf("The sum of %d and %d is %d\n", num1, num2, sum);
21
22     return 0; // 程序正常结束
23 }

```

编译运行

riscv64-unknown-linux-gnu-gcc mycode2.c -o mycode2 -static

qemu-riscv64 mycode2

输入 3 4, 得到输出 The sum of 3 and 4 is 7, 如图15所示。

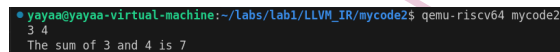


图 15: mycode2

编写其对应的 LLVM IR 代码, 代码内容以及解释如下。

```

1 ; 格式化字符与字符串
2 ; "%d %d"
3 ; "The sum of %d and %d is %d\n"
4 @.str = private unnamed_addr constant [6 x i8] c"%d %d\00", align 1
5 @.str.1 = private unnamed_addr constant [28 x i8] c"The sum of %d and %d is %
   d\0A\00", align 1
6
7 ; addNumbers函数定义
8 ; 这里存在问题: 1、为什么这里从%0开始? 从%1开始会报错。2、为什么没有%2,直接%3
   ? 写成%2会报错
9 define dso_local i32 @addNumbers(i32 noundef %0, i32 noundef %1) {
10     %3 = add nsw i32 %0, %1
11     ret i32 %3
12 }
13
14 ; main函数
15 define dso_local i32 @main() {
16     ; 声明局部变量
17     %1 = alloca i32, align 4
18     store i32 0, i32* %1, align 4
19
20     ; num1;num2;sum
21     %2 = alloca i32, align 4
22     %3 = alloca i32, align 4

```

```

23 %4 = alloca i32, align 4
24 ; 调用scanf, 第一个参数为格式字符@.str, 第二、三个参数为指向%2、%3的指针
25 %5 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds
    ([6 x i8], [6 x i8]* @.str, i64 0, i64 0), i32* noundef %2, i32*
    noundef %3)
26
27 ; 调用addNumbers, num1,num2放入临时寄存器
28 %6 = load i32, i32* %2, align 4
29 %7 = load i32, i32* %3, align 4
30 %8 = call i32 @addNumbers(i32 noundef %6, i32 noundef %7)
31
32 ; 将函数返回结果存入%4, sum
33 store i32 %8, i32* %4, align 4
34
35 ; 调用printf, num1,num2,sum放入临时寄存器
36 %9 = load i32, i32* %2, align 4
37 %10 = load i32, i32* %3, align 4
38 %11 = load i32, i32* %4, align 4
39 ; 第一个参数为字符串@.str.1, 第二、三、四个参数为几个临时寄存器
40 %12 = call i32 @__isoc99_printf(i8* noundef getelementptr inbounds ([28 x
    i8], [28 x i8]* @.str.1, i64 0, i64 0), i32 noundef %9, i32 noundef
    %10, i32 noundef %11)
41
42 ; 结束, 返回
43 ret i32 0
44 }
45
46 ; 函数声明
47 declare i32 @__isoc99_scanf(i8* noundef, ...)
48 declare i32 @__isoc99_printf(i8* noundef, ...)

```

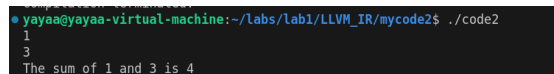
将文件继续编译汇编链接成可执行文件：

```
llvm-as code2.ll -o code2.bc
```

```
llc code2.bc -filetype=obj -relocation-model=pic -o code2.o
```

```
gcc code2.o -o code2
```

./code2 运行程序，如图16所示，输入 1 3，程序正确打印出 The sum of 1 and 3 is 4。



```

yayaa@yayaa-virtual-machine:~/labs/lab1/LLVM_IR/mycode2$ ./code2
1
3
The sum of 1 and 3 is 4

```

图 16: code2

在编写上述.ll 文件时遇到了一些问题：

1. 定义 addNumbers 函数时，参数中的声明从%0 开始，用%1 会报错。
2. 在函数体中，局部变量从%3 开始，用%2 会报错，为什么标号没有连续？

关于这些问题，我在网上并没有找到答案，会在答辩时询问一下助教。

3. 程序 3——浮点数, if 分支判断, 输入输出

第三个程序读入两个浮点数并比较大小, 输出比较结果, 涉及到浮点数, if 分支判断, 输入输出等语法, 其 c 程序如下:

```

1 #include <stdio.h>
2
3 // 浮点数 if分支判断 输入输出
4
5 int main() {
6     float num1, num2;
7
8     // 提示用户输入两个浮点数
9     printf("请输入第一个浮点数: ");
10    scanf("%f", &num1);
11    printf("请输入第二个浮点数: ");
12    scanf("%f", &num2);
13
14    // 使用if判断语句比较两个数
15    if (num1 > num2) {
16        // 如果第一个数大于第二个数, 则打印以下信息
17        printf("%.2f 大于 %.2f\n", num1, num2);
18    } else {
19        // 如果第一个数不大于第二个数 (即小于或等于), 则打印以下信息
20        printf("%.2f 不大于 %.2f\n", num1, num2);
21    }
22
23    return 0;
24 }
```

编译运行

```
riscv64-unknown-linux-gnu-gcc mycode3.c -o mycode3 -static
```

```
qemu-riscv64 mycode3
```

输入 1.1 2.2, 得到输出 “1.10 不大于 2.20”, 如图17所示。



```

yayaa@yayaa-virtual-machine:~/labs/lab1/LLVM_IR/mycode3$ qemu-riscv64 mycode3
请输入第一个浮点数: 1.1
请输入第二个浮点数: 2.2
1.10 不大于 2.20

```

图 17: mycode3

编写其对应的 LLVM IR 代码, 代码内容以及解释如下。

```

1 ; 两个需要打印的字符串, 三个格式化字符
2 ; 请输入第一个浮点数:
3 @.str = private unnamed_addr constant [30 x i8] c"\E8\AF\B7\E8\BE\93\E5\85\A5
   \E7\AC\AC\E4\B8\80\E4\B8\AA\E6\B5\AE\E7\82\B9\E6\95\B0: \00", align 1
4 ; %f
5 @.str.1 = private unnamed_addr constant [3 x i8] c"%f\00", align 1
6 ; 请输入第二个浮点数:
7 @.str.2 = private unnamed_addr constant [30 x i8] c"\E8\AF\B7\E8\BE\93\E5\85\
   A5\E7\AC\AC\E4\BA\8C\E4\B8\AA\E6\B5\AE\E7\82\B9\E6\95\B0: \00", align 1

```

```

8 ; %.2f 大于 %.2f\n
9 @.str.3 = private unnamed_addr constant [18 x i8] c"%.2f \E5\A4\A7\E4\BA\8E
   %.2f\0A\00", align 1
10 ; %.2f 不大于 %.2f\n
11 @.str.4 = private unnamed_addr constant [21 x i8] c"%.2f \E4\B8\8D\E5\A4\A7\
   E4\BA\8E %.2f\0A\00", align 1
12
13 ; Function Attrs: noline nounwind optnone uwtable
14 define dso_local i32 @main() {
15     %1 = alloca i32, align 4
16     store i32 0, i32* %1, align 4
17
18     ; 声明num1与num2, 32位浮点型float指针
19     %2 = alloca float, align 4
20     %3 = alloca float, align 4
21
22     ; 调用print, scanf函数
23     %4 = call i32 @__printf(i8* noundef getelementptr inbounds ([30 x
        i8], [30 x i8]* @.str, i64 0, i64 0))
24     %5 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds
        ([3 x i8], [3 x i8]* @.str.1, i64 0, i64 0), float* noundef %2)
25     %6 = call i32 @__printf(i8* noundef getelementptr inbounds ([30 x
        i8], [30 x i8]* @.str.2, i64 0, i64 0))
26     %7 = call i32 @__isoc99_scanf(i8* noundef getelementptr inbounds
        ([3 x i8], [3 x i8]* @.str.1, i64 0, i64 0), float* noundef %3)
27
28     ; 取出num1, num2到两个临时变量。
29     %8 = load float, float* %2, align 4
30     %9 = load float, float* %3, align 4
31
32     ; fcmp比较。ogt用于比较两个浮点数, 有序大于
33     %10 = fcmp ogt float %8, %9
34
35     ; br跳转。条件为%10这个布尔值。
36     br i1 %10, label %11, label %17
37
38 11:
39     ; 为真                                ; preds = %0
40     ; C 语言中, 当 printf 被调用时, float 类型会被自动转换为 double 类型。
41     ; 所以在调用printf之前fpext做一个类型转换
42     %12 = load float, float* %2, align 4
43     %13 = fpext float %12 to double
44     %14 = load float, float* %3, align 4
45     %15 = fpext float %14 to double
46     %16 = call i32 @__printf(i8* noundef getelementptr inbounds ([18 x
        i8], [18 x i8]* @.str.3, i64 0, i64 0), double noundef %13, double
        noundef %15)
47     ; 为真后要跳到if函数外, 不再执行为假的代码

```

```
yayaa@yayaa-virtual-machine:~/labs/lab1$ clang --version
Ubuntu clang version 14.0.0-1ubuntu1.1
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

图 19: clang 版本

```
48 br label %23
49
50 17:
51 ; 为假 ; preds = %0
52 %18 = load float, float* %2, align 4
53 %19 = fpext float %18 to double
54 %20 = load float, float* %3, align 4
55 %21 = fpext float %20 to double
56 %22 = call i32 @__printf(i8* noundef, ...) @printf(i8* noundef getelementptr inbounds ([21 x
    i8], [21 x i8]* @.str.4, i64 0, i64 0), double noundef %19, double
    noundef %21)
57 br label %23
58
59 23:
60 ; 结束, 返回 ; preds = %17, %11
61 ret i32 0
62 }
63
64 ; 函数声明
65 declare i32 @__printf(i8* noundef, ...)
66 declare i32 @__isoc99_scanf(i8* noundef, ...)
```

将文件继续编译汇编链接成可执行文件：

```
llvm-as code3.ll -o code3.bc
```

```
llc code3.bc -filetype=obj -relocation-model=pic -o code3.o
```

```
gcc code3.o -o code3
```

./code3 运行程序，如图18所示：输入 1.25 12.5，程序正确打印出“1.25 不大于 12.50”；输入 1.25 1.25，程序正确打印出“1.25 不大于 1.25”；输入 5.5 1.111，程序正确打印出“5.50 大于 1.11”。

```
yayaa@yayaa-virtual-machine:~/labs/lab1/LLVM_IR/mycode3$ ./code3
请输入第一个浮点数: 1.25
请输入第二个浮点数: 12.5
1.25 不大于 12.50
yayaa@yayaa-virtual-machine:~/labs/lab1/LLVM_IR/mycode3$ ./code3
请输入第一个浮点数: 1.25
请输入第二个浮点数: 1.25
1.25 不大于 1.25
yayaa@yayaa-virtual-machine:~/labs/lab1/LLVM_IR/mycode3$ ./code3
请输入第一个浮点数: 5.5
请输入第二个浮点数: 1.111
5.50 大于 1.11
```

图 18: code3

(三) 关于 OpaquePointers

在先前的实验中，使用的 clang 版本如19所示，一直是 14.0.0，所以前面的 ll 文件中都是透明指针。

使用指令 `sudo update-alternatives --config clang` 选择 15 以上版本的 clang，将 clang 版本切换到 15.0.7。

```
yayenayee@virtual-machine:~/labs/lab1/LLVM_IR/mycode1$ sudo update-alternatives --config clang
有 2 个候选项可用于替换 clang (提供 /usr/bin/clang)。

  选择    路径                优先级    状态
-----
0        /usr/bin/clang-15          100      自动模式
* 1        /usr/bin/clang-14          98       手动模式
2        /usr/bin/clang-15          100      手动模式

要维持当前值[*]请按<回车键>，或者键入选择的编号： 0
update-alternatives: 使用 /usr/bin/clang-15 来在自动模式中提供 /usr/bin/clang (clang)
yayenayee@virtual-machine:~/labs/lab1/LLVM_IR/mycode1$ clang --version
Ubuntu clang version 15.0.7
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

图 20: clang 版本切换

因为 mycode1.c 中是对数组的探索，对指针的运用较多，所以选择该文件，重新为 mycode1.c 生成运用了 OpaquePointers 的中间语言代码 mycode1_Opaque.ll。

如图21所示，图中时调用 memcpy 将全局中的数组复制到 main 函数局部数组中的部分代码，左侧为没有运用 OpaquePointers 的 mycode1.ll，右侧为新生成的 mycode1_Opaque.ll。

<pre>17 %5 = bitcast [5 x i32]* %2 to i8* 18 call void @llvm.memcpy.p0i8.p0i8.i64(19 %5, align 16 %5, 20 %1, align 16 bitcast ([5 x i32]* @_const.main.numbers to i8*), 21 164 20, 22 ! false) 23 24 %6 = getelementptr inbounds [5 x i32], [5 x i32]* %2, 164 0, 164 1 25 store i32 6, i32* %6, align 4 26 store i32 5, i32* %6, align 4 27 store i32 0, i32* %6, align 4 28 br label %7 29 30 7: ; preds = %17, %6</pre>	<pre>17 call void @llvm.memcpy.p0i8.p0i8.i64(18 ptr, align 16 %2, 19 ptr, align 16 @_const.main.numbers, 20 164 20, 21 ! false) 22 23 %5 = getelementptr inbounds [5 x i32], %2, 164 0, 164 1 24 store i32 6, ptr %5, align 4 25 store i32 5, ptr %5, align 4 26 store i32 0, ptr %5, align 4 27 br label %6 28 29 6: ; preds = %10, %5</pre>
--	---

图 21: 两.ll 文件对比

在先前提到过，memcpy 函数的操作以字节级别进行，所以需要对数组指针进行类型转换。可以看到新生成的代码中，由于指针是不透明的形式，都用 ptr 来代表指针，所以这一步操作被省略，直接进行了 memcpy 操作。这样就节省了一段代码运行的时间，并且减少了一个临时寄存器的占用。在大量需要数组指针等操作的程序中，将会提供明显的优化效果。

除此之外，OpaquePointers 也减少了单句代码的长度，让代码变得更加简洁易懂，对开发者而言也是一个优点。

四、 RISC-V 汇编编程——刘俊彤

在这一部分我选择了 RISC-V 汇编语言进行学习探索，以三个简单的 c 程序为例，编写其等价的汇编代码，运行验证。

(一) RISC-V 指令学习

在实验开始之前我首先对于该汇编语言进行了学习，以 32 位 RISC-V 指令集为参考，关键知识如下：

1. 基本知识

RISC-V 是一个近十年诞生的开源指令集架构，和传统增量 ISA 方式不同，RISC-v 是模块化的。其核心是基础 ISA-RV32I，运行一个完整的软件栈，其是固定不变的。

RISC-V 有若干可选的标准扩展，根据应用程序的需要，硬件可以包含或者不包含这些扩展。RISC-V 编译器得知当前硬件包含哪些扩展后，便可以生成当前硬件条件下的最佳代码。

RISC-V 是一个最新的，清晰的，简约的，开源的 ISA，RISC-V 架构师的目标是让它从最小的到最快的所有计算设备上都能有效工作。RISC-V 强调简洁性来保证它的低成本，同时有着大量的寄存器和透明的指令执行速度，从而帮助编译器和汇编语言程序员将实际的重要问题转换为适当的高效代码。

2. RISC-V 基础整数指令集——RV32I

(1)RV32I 的指令

RV32I 有六种基本指令格式：用于寄存器-寄存器操作的 R 类型指令，用于短立即数和访存 load 操作的 I 型指令，用于访存 store 操作的 S 型指令，用于条件跳转操作的 B 类型指令，用于长立即数的 U 型指令和用于无条件跳转的 J 型指令。每种指令的结构如图22所示。

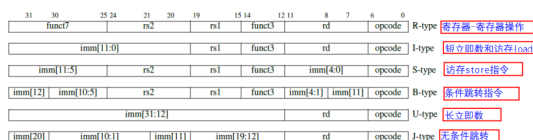


图 22: 指令结构

可以看出 RV32I 指令有以下特点：

- 只有 6 中指令且每条指令都是 32 位长，这样简化了指令解码；
- 提供三个寄存器操作数，对于需要三个不同操作数的指令执行很方便；
- 所有指令中，要读写的寄存器的标识符总是在同一位置，意味着在解码指令之前，就可以先开始访问寄存器；
- 立即数字段总是符号扩展，符号位总是在指令中最高位。这意味着可能成为关键路径的立即数符号扩展，可以在指令解码之前进行；
- 此外，所有位都是 0 或者所有位都是 1 的指令在 RV32I 是非法的；

(2)RV32I 的寄存器

RV32I 有 31 个寄存器加上一个值恒为 0 的 x0 寄存器，0 寄存器有助于加速大概率时间，对于性能提升有很大帮助。0 寄存器可以作为操作数使有限的指令完成更多功能。

(3)RV32I 整数计算

算数指令 (add、sub)，逻辑指令 (and、or、xor)，位移指令 (sll、srl、sra) 以及其立即数版本和传统的 ISA 相同，都是从寄存器读取 32 位数据，将 32 位结果写回目标寄存器。但 RV32I 不同的是，立即数总是会进行符号扩展，也就是我们可以用立即数表示负数，立即数减法就可以用加法实现，所以没有提供立即数版本的 sub 指令。

对于比较判断，程序需要根据比较结果生成布尔值，为应对这种使用场景下，RV32I 提供一个当小于时置位的指令。如果第一个操作数小于第二个操作数，它将目标寄存器设置为 1，否则为 0。有符号数使用 slt，无符号数使用 sltu；同时该指令也有立即数版本 (slti，sltu)。

用于构造大的常量数值和链接，有 lui 加载立即数到高位，auipc 向 pc 高位加上立即数。

(4)RV32I 的 Load 和 Store

除了提供 32 位字 (lw、sw) 的加载和存储外，RV32I 支持加载有符号和无符号字节和半字 (lb、lbu、lh、lhu) 和存储字节和半字 (sb、sh)。

加载和存储的支持的唯一寻址模式是符号扩展 12 位立即数到基地址寄存器,也就是 x86_32 中的位移偏移寻址模式。

(5)RV32I 条件分支

RV32I 可以比较两个寄存器并根据比较结果上进行分支跳转。比较可以是:相等 (beq), 不相等 (bne), 大于等于 (bge), 或小于 (blt)。bge 和 blt 有对应的无符号比较 bgeu 和 bltu。

RISC-V 去掉了其他指令集的条件码设置也没有 loop 等循环指令,只使用少量的简单指令完成分支功能。

(6)RV32I 无条件跳转

跳转并链接指令 (jal) 具有双重功能:若将下一条指令 PC + 4 的地址保存到目标寄存器中,便可以用它来实现过程调用。如果使用零寄存器 (x0) 替换 ra 作为目标寄存器,则可以实现无条件跳转,因为 x0 不能更改。

跳转和链接指令的寄存器版本 (jalr) 同样是多用途的:它可以调用地址是动态计算出来的函数,或者也可以实现调用返回(只需 ra 作为源寄存器,零寄存器 (x0) 作为目的寄存器)。jalr 指令的目的寄存器设为 x0, 就可以实现 switch 和 case 语句的地址跳转功能。

3. RISC-V 乘法除法指令扩展——RV32M

(1) 除法

RV32M 向 RV32I 中添加了整数乘法和除法指令。

除法会得到两个结果——商和余数,在 RV32M 中用不同的指令来得到商和余数,并对于有符号数和无符号数的操作进行了区分,所以 RV32M 中一共有 4 条除法指令:

将商放入目标寄存器:有符号数: div ; 无符号数: divu

将余数放入目标寄存器:有符号数: rem ; 无符号数: remu

(2) 乘法

因为两个 32 位数相乘会得到 64 位结果,而 32 位的 RISC-V 的寄存器是 32 位的。显然 64 位的结果需要两个 32 位寄存器,所以 RV32M 中分别用不同的指令来得到结果的高 32 位和低 32 位。

对于有符号数和无符号数的操作数计算出结果的高 32 位,也用了不同的指令区分。

所以 RV32M 具有四个乘法指令:

mul: 得到积的低 32 位; mulh: 两个有符号数的操作数,得到高 32 位; mulhu: 两个无符号数的操作数,得到高 32 位; mulhsu: 一个有符号操作数和一个无符号操作数,得到高 32 位。

RISC-V 没有选择一句指令将得到的 64 位积写入两个 32 位寄存器,是因为在一条指令中完成把 64 位积写入两个 32 位寄存器的操作会使硬件设计变得复杂。

4. RISC-V 浮点数扩展——RV32F 和 RV32D

RV32F 和 RV32D 使用 32 个独立的 f 寄存器而不是 x 寄存器。这样可以在不增加 RISC-V 指令格式中的寄存器描述符所占空间的情况下将集训期容量和带宽增加到两倍,提高处理性能。但这样做也意味着必须要添加新的指令来加载和存储数据 f 寄存器,还需要添加新指令用于在 x 和 f 寄存器之间传递数据。

如果处理器同时支持 RV32F 和 RV32D 扩展,则单精度数据仅使用 f 寄存器中的低 32 位。f 寄存器中的 f0 不是常量 0,而是和其他 31 个寄存器相同的可变寄存器。

对于 RV32F 和 RV32D, RISC-V 有两条加载指令 (flw、fld) 和两条存储指令 (fsw、fwd),其寻址模式和指令格式和 lw、sw 相同。浮点数的标准算术运算指令有 fadd.s、fadd.d、fsub.s、fsub.d、fmul.s、fmul.d、fdiv.s、fdiv.d。RV32F 和 RV32D 还包括平方根 (fsqrt.s、fsqrt.d) 指令。

它们也有最小值和最大值指令（fmin.s, fmin.d, fmax.s, fmax.d），这些指令在不使用分支指令进行比较的情况下，将一对源操作数中的较小值或较大值写入目的寄存器。

在程序中许多浮点算法在执行完乘法运算后会立即执行一条加法或减法指令。因此 RISC-V 提供了先将两个操作数相乘然后将乘积加上（fmadd.s, fmadd.d）或减去（fmsub.s, fmsub.d）第三个操作数，最后再将结果写入目的寄存器的指令。这些指令比单独使用乘法和加法指令的精确度更高，因为他们少舍入了一次。

RV32F 和 RV32D 没有提供浮点分支指令，提供了浮点比较指令，这些根据两个浮点的比较结果将一个整数寄存器设置为 1 或 0：feq.s、feq.d、flt.s、flt.d、fle.s、fle.d。后续整数分支指令可以根据该整数寄存器值进行分支跳转。

（二） -static 选项探索

当输入如下指令：

```
riscv64-unknown-linux-gnu-gcc test1.s -o test1
```

```
qemu-riscv64 test
```

执行时会报如下错误：

“qemu-riscv64: Could not open '/lib/ld-linux-riscv64-lp64d.so.1': No such file or directory”

输入 “riscv64-unknown-linux-gnu-gcc test1.s -o test1 -static”，则可以正常运行。

查询资料得知编译时的-static 选项是用于指示编译器生成一个静态链接的可执行文件。也就是在编译和连接过程中，编译器会尝试将程序所需要的所有库的静态版本直接包含到最终的可执行文件中，而不是运行时从系统的动态链接库中加载这些库。

而当不使用-static 选项时，运行可执行文件就会报如上错误，在可执行文件过程中需要从系统的动态链接库中加载这些库，因为 qemu 无法找到编译的动态链接器，导致无法加载程序中需要的库。

（三） 汇编编程

1. 程序 1——数组、除法、for、输出

第一个程序为将数组中的元素打印输出，涉及到数组、除法、for、输出等语法，其 c 程序如下：

```
1 #include <stdio.h>
2 // 数组 除法 for 循环
3 int main() {
4     // 定义一个整型数组，包含5个元素
5     int numbers[] = {1, 2, 3, 4, 5};
6     // 获取数组的大小
7     int size = sizeof(numbers) / sizeof(numbers[0]);
8     // 遍历数组并打印每个元素
9     for(int i = 0; i < size; i++) {
10         printf("%d ", numbers[i]);
11     }
12     return 0;
13 }
```

直接将上述程序进行编译运行

```
riscv64-unknown-linux-gnu-gcc test1.c -o test1_1 -static qemu-riscv64 test1_1
```

得到 1 2 3 4 5 的输出，如图23所示。



图 23: 程序 1 源代码执行

编写的其等价的 risc-v 汇编代码以及代码解析如下：

```

1
2      #文件标识、选项和属性声明，指定了目标框架、未使用位置无关代码（PIC
3      ） 、 对齐等
4      .file      "test1.c" #源文件为test1.c
5      .option    nopie #设置汇编器选项 nopie表示不使用位置无关代码（pie）位置
6      无关码是一种一个在内存中任何位置运行的代码，在动态链接时使用
7      .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0" #设置架
8      构属性，arch为目标架构的特性集。这里包括risc-v的若干扩展，如整数
9      （i）、乘除法（m）、原子（a）、浮点（f和d）、压缩（c）和访问控制状
10     态寄存器（zicsr）
11     .attribute    unaligned_access, 0 #设置属性unaligned_access为0，表示生成
12     的代码不包含未对齐内存地址的访问。访问未对齐地址可能导致性能下降
13     或者硬件异常。这样可以保证代码的兼容性和可移植性。
14     .attribute    stack_align, 16 #设置栈对齐属性，值为16，表示栈指针（sp）
15     应该始终保持在16字节的边界上。
16
17     .text #标记代码段开始
18     .section      .rodata #rodata段，存储只读数据
19     .align 3 #指定接下来的数据或代码应该对齐到2的3次方（即8字节）的边界
20
21 .LC1:
22     .string "%d " #字符串常量
23     .align 3
24
25 .LC0:
26     #在当前位置插入五个word大小的整数常量
27     .word 1
28     .word 2
29     .word 3
30     .word 4
31     .word 5
32     .text
33     .align 1
34     .globl main #标记全局
35     .type main, @function #函数类型
36
37 main: #main函数开始
38     #为main函数设置栈帧
39     addi sp, sp, -48 #留出局部变量和寄存器的空间
40     sd ra, 40(sp) #ra寄存器储存到sp+40位置
41     sd s0, 32(sp) #s0储存到sp+32位置

```



```

34      addi    s0,sp,48  #设置新的栈底指针s0
35
36      #初始化局部变量
37      #栈上存好1 2 3 4 5 5 0 a5=5 0和5用于循环计数和循环结束判断
38      lui     a5,%hi(.LC0)  #加载数组.LC0高16位地址到a5
39      addi    a5,a5,%lo(.LC0)  #数组.LC0低16位地址加到a5, 现在a5中包含了.
        LC0的完整地址
40      ld      a4,0(a5)  #从数组加载双字到a4 1 2
41      sd      a4,-48(s0)  # 将a4中的值存储到以s0为基址, 偏移-48字节的位置
42      ld      a4,8(a5)  #从数组加载下一个双字到a4 3 4
43      sd      a4,-40(s0)  #将a4中的值存储到以s0为基址, 偏移-40字节的位置
44      lw      a5,16(a5)  #从数组加载一个字到a5 5
45      sw      a5,-32(s0)  # 将a5中的值存储到以s0为基址, 偏移-32字节的位置
46      li      a5,5  #将立即数5加载到a5中
47      sw      a5,-24(s0)  #将a5中的值存储到以s0为基址, 偏移-24字节的位置
48      sw      zero,-20(s0)  #将0寄存器存储到以s0为基址, 偏移-20字节的位置
49      j       .L2  #跳转到循环开始
50
51      .L3:
52      #打印数组元素
53      lw      a5,-20(s0)  #获取计数器值到a5
54      slli    a5,a5,2  #将a5*4得到待打印数的数组偏移
55      addi    a5,a5,-16  #偏移量减16, 得到在栈的偏移
56      add     a5,a5,s0  #加上栈底地址, 得到要访问数据的地址
57      lw      a5,-32(a5)  #加载数据 (32位)
58      mv      a1,a5  #将数据存到a1
59      lui     a5,%hi(.LC1)  #加载格式字符串.LC1的高16位地址到a5
60      addi    a0,a5,%lo(.LC1)  #将.LC1的低16位地址加到a5上, 得到完整地址,
        并存储到a0
61      call    printf  #调用printf打印数据
62      lw      a5,-20(s0)  #重新加载计数器值
63      addiw   a5,a5,1  #计数器加1
64      sw      a5,-20(s0)  #将计数器值存到栈上
65
66      .L2:
67      lw      a5,-20(s0)  #a5=0
68      mv      a4,a5  #a4=0
69      lw      a5,-24(s0)  #a5=5 a5作为循环计数器
70      sext.w  a4,a4
71      sext.w  a5,a5  #符号扩展
72      blt     a4,a5,.L3  #判断循环条件, 如果a4<a5, 则跳转到L3进行输出
73
74      #循环结束, 完成输出后进行栈恢复
75      li      a5,0  #恢复a5为0
76      mv      a0,a5  #恢复a0为0
77      ld      ra,40(sp)  #恢复返回地址
78      ld      s0,32(sp)  #恢复栈
79      addi    sp,sp,48

```

```

80      jr      ra    #返回函数main的调用位置
81
82      .size    main, .-main
83      .ident   "GCC: () 12.2.0"
84      .section          .note.GNU-stack,"",@progbits

```

将汇编程序进行汇编链接形成可执行文件，运行

```
riscv64-unknown-linux-gnu-gcc test1_2.s -o test1_2 -static qemu-riscv64 test1_2
```

得到 1 2 3 4 5 的正确输出，如图24所示。



图 24: 程序 1 汇编执行结果

将原来的 c 程序使用 -O3 进行编译

```
riscv64-unknown-linux-gnu-gcc -O3 -S test1.c -o test1_3.s
```

得到汇编码如下：

```

1      .file      "test1.c"
2      .option    nopie
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.str1.8,"aMS",@progbits,1
8      .align    3
9      .LC1:
10     .string  "%d "
11     .section          .text.startup,"ax",@progbits
12     .align    1
13     .globl  main
14     .type    main, @function
15 main:
16     li      a5,1
17     slli    a4,a5,33
18     slli    a5,a5,34
19     addi    sp,sp,-64
20     addi    a5,a5,3
21     addi    a4,a4,1
22     sd      a5,16(sp)
23     li      a5,5
24     sd      s0,48(sp)
25     sd      s1,40(sp)
26     sd      s2,32(sp)
27     sd      ra,56(sp)
28     sd      a4,8(sp)
29     sw      a5,24(sp)
30     addi    s0,sp,8
31     addi    s2,sp,28

```

```

32      lui      s1,%hi(.LC1)
33  .L2:
34      lw       a1,0(s0)
35      addi     a0,s1,%lo(.LC1)
36      addi     s0,s0,4
37      call     printf
38      bne      s0,s2,.L2
39      ld       ra,56(sp)
40      ld       s0,48(sp)
41      ld       s1,40(sp)
42      ld       s2,32(sp)
43      li       a0,0
44      addi     sp,sp,64
45      jr       ra
46      .size    main,.-main
47      .ident   "GCC: () 12.2.0"
48      .section .note.GNU-stack,"",@progbits

```

对比两个汇编程序可以发现，使用-o3 优化后程序明显变短了，具体进行的优化如下：

(1) 改善了寄存器使用和栈管理：未优化版本使用了更多的栈空间和寄存器来保存局部变量和临时值。它使用 s0 寄存器作为栈指针的一部分，并在栈上保存了数组元素的副本。但-o3 版本中，减少了栈的使用，并更有效地使用了寄存器。它仅在开始时调整栈指针，并将 printf 的格式字符串高位地址和循环计数器保存在寄存器中，循环打印时只需要加载格式字符串低位地址。此外，循环的迭代是通过调整指针 s0 来实现的，而不是通过数组索引和额外的内存访问。

(2) 指令选择和代码结构：未优化的代码包含了许多加载(ld/lw)和存储(sd/sw)指令，用于在栈和寄存器之间移动数据。这增加了执行时间和内存访问次数。-O3 版本的代码通过更高效的指令序列来减少这些操作。它通过直接计算偏移量来访问数组元素，而不是通过多次加载和存储来间接访问。

2. 程序 2——函数调用、输入、输出

第二个程序为将输入的两个整数相加，输出结果，涉及到函数调用、输入、输出等语法，其 c 程序如下：

```

1  #include <stdio.h>
2  // 函数调用 输入输出
3  // 定义一个函数，该函数接收两个整数作为参数，并返回它们的和
4  int addNumbers(int a, int b) {
5      return a + b; // 返回两个数的和
6  }
7  int main() {
8      int num1; // 定义第一个整数
9      int num2; // 定义第二个整数
10     scanf("%d %d", &num1, &num2);
11
12     // 调用addNumbers函数，并将结果存储在变量sum中
13     int sum = addNumbers(num1, num2);
14     // 打印结果

```

```

15     printf("The sum of %d and %d is %d\n", num1, num2, sum);
16     return 0; // 程序正常结束
17 }

```

直接将上述程序进行编译运行

riscv64-unknown-linux-gnu-gcc test2.c -o test2_1 -static qemu-riscv64 test2_1

输入 7 5, 得到和为 12 的输出, 如图25所示。



```

* lianliu-virtual-machine:~/编译原理/lab1/73 riscv64-unknown-linux-gnu-gcc test2.c -o test2_1 -static
* lianliu-virtual-machine:~/编译原理/lab1/73 qemu-riscv64 test2_1
7 5
The sum of 7 and 5 is 12

```

图 25: 程序 2 源代码执行结果

编写其等价的 risc-v 汇编代码以及代码解析如下:

```

1     .file      "test2.c"
2
3     .option    nopie
4     .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
5     .attribute unaligned_access, 0
6     .attribute stack_align, 16
7
8     #addNumber 函数
9     .text
10    .align     1
11    .globl     addNumbers
12    .type      addNumbers, @function
13
14    addNumbers:
15        addw    a0,a0,a1    #a0+a1, 结果存回a0
16        ret     #返回 返回值存储在a0寄存器中
17    .size      addNumbers, .-addNumbers
18    .section   .rodata.str1.8,"aMS",@progbits,1
19    .align     3
20
21    .LC0:
22        .string "%d %d"    #字符串常量
23        .align     3
24
25    .LC1:
26        .string "The sum of %d and %d is %d\n" #字符串常量
27    .section   .text.startup,"ax",@progbits
28    .align     1
29    .globl     main
30    .type      main, @function
31
32    main:
33        addi    sp,sp,-32   #为局部变量和调用预留栈空间
34        lui     a0,%hi(.LC0) #加载格式字符串"%d %d"的高位地址
35        addi    a2,sp,12    #第二个输入整数的栈位置

```

```

36      addi    a1,sp,8    #第一个输入整数的栈位置
37      addi    a0,a0,%lo(.LC0)    #加载格式字符串"%d %d"的低位地址得到字符串
      完整地址
38      sd      ra,24(sp)    #保存返回地址到栈上
39      call    __isoc99_scanf    #调用scanf读取输入到a1、a2位置
40      lw      a1,8(sp)    #得到输入的第一个整数存到a1
41      lw      a2,12(sp)    #得到输入的第二个整数存到a2
42      lui     a0,%hi(.LC1)    #加载字符串"The sum of %d and %d is %d\n"的高
      位地址
43      addi    a0,a0,%lo(.LC1)    #加载格式字符串的低位地址，得到完整地址
44      addw    a3,a1,a2    #计算a1和a2的和，存到a3 内联函数
45      call    printf    #调用printf打印计算结果
46
47      #恢复寄存器以及栈
48      ld      ra,24(sp)
49      li      a0,0
50      addi    sp,sp,32
51      jr      ra
52      .size   main,.-main
53      .ident  "GCC: () 12.2.0"
54      .section .note.GNU-stack,"",@progbits

```

将汇编程序进行汇编链接形成可执行文件，运行

riscv64-unknown-linux-gnu-gcc test2_2.s -o test2_2 -static qemu-riscv64 test2_2

输入 6 9，得到和为 15 的输出，如图26所示。



```

qemu-riscv64 test2_2
6 9
The sum of 6 and 9 is 15

```

图 26: 程序 2 汇编代码运行结果

上述的汇编代码是-o3 优化后的汇编码，使用-o0 没有优化的代码实现加法的部分如下：

```

1      call    __isoc99_scanf
2      lw      a5,-24(s0)
3      lw      a4,-28(s0)
4      mv      a1,a4
5      mv      a0,a5
6      call    addNumbers
7      mv      a5,a0

```

可以明显看到，在未优化的版本中，main 函数调用了 addNumber 函数进行加法计算；而-o3 优化的版本中，编译器对代码进行了内联函数的优化，在 main 函数中没有进行调用 addNumber 函数，而是直接使用 addw a3,a1,a2，将函数中实现的加法在 main 函数中执行，避免了复杂的函数调用过程，使效率提高。

3. 程序 3——浮点数、if 分支判断、输入、输出

第三个程序为比较输入的两个浮点数，并将打印结果输出，涉及到浮点数、if 分支判断、输入、输出等语法，其 c 程序如下：

```

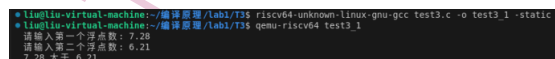
1 #include <stdio.h>
2
3 // 浮点数 if分支判断 输入输出
4
5 int main() {
6     float num1, num2;
7
8     // 提示用户输入两个浮点数
9     printf("请输入第一个浮点数: ");
10    scanf("%f", &num1);
11    printf("请输入第二个浮点数: ");
12    scanf("%f", &num2);
13
14    // 使用if判断语句比较两个数
15    if (num1 > num2) {
16        // 如果第一个数大于第二个数, 则打印以下信息
17        printf("%.2f 大于 %.2f\n", num1, num2);
18    } else {
19        // 如果第一个数不大于第二个数 (即小于或等于), 则打印以下信息
20        printf("%.2f 不大于 %.2f\n", num1, num2);
21    }
22
23    return 0;
24 }

```

直接将上述程序进行编译运行

riscv64-unknown-linux-gnu-gcc test3.c -o test3_1 -static qemu-riscv64 test3_1

输入 7.28 6.21, 得到 7.28 大于 6.21 的输出, 如图27所示。



```

* Linux-virtual-machine:~/编译实验/lab1/3$ riscv64-unknown-linux-gnu-gcc test3.c -o test3_1 -static
* Linux-virtual-machine:~/编译实验/lab1/3$ qemu-riscv64 test3_1
请输入第一个浮点数: 7.28
请输入第二个浮点数: 6.21
7.28 大于 6.21

```

图 27: 程序 3 源代码执行结果

编写其等价的 risc-v 汇编代码以及代码解析如下:

```

1
2     .file      "test3.c"
3     .option   nopie
4     .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
5     .attribute unaligned_access, 0
6     .attribute stack_align, 16
7     .text
8     .section   .rodata.str1.8,"aMS",@progbits,1
9     .align     3
10
11 #字符串常量
12 .LC0:
13     .string    "\350\257\267\350\276\223\345\205\245\347\254\254\344\270

```

```

14         \200\344\270\252\346\265\256\347\202\271\346\225\260: "
15     .align 3
16
17 .LC1:
18     .string "%f"
19     .align 3
20
21 .LC2:
22     .string "\350\257\267\350\276\223\345\205\245\347\254\254\344\272
23         \214\344\270\252\346\265\256\347\202\271\346\225\260: "
24     .align 3
25
26 .LC3:
27     .string "%.2f \345\244\247\344\272\216 %.2f\n"
28     .align 3
29
30 .LC4:
31     .string "%.2f \344\270\215\345\244\247\344\272\216 %.2f\n"
32     .section .text.startup,"ax",@progbits
33     .align 1
34     .globl main
35     .type main, @function
36
37 main:
38     lui a0,%hi(.LC0)
39     addi sp,sp,-32 #预留栈空间
40     addi a0,a0,%lo(.LC0) #加载字符串"请输入第一个浮点数:"地址
41     sd ra,24(sp) #返回地址
42     sd s0,16(sp) #栈帧调整
43     call printf #打印"请输入第一个浮点数:"
44     lui s0,%hi(.LC1)
45     addi a1,sp,8
46     addi a0,s0,%lo(.LC1) #字符串"%f"
47     call __isoc99_scanf #调用scanf函数读取用户输入的浮点数, 存储到
        栈上 (sp+8)
48     lui a0,%hi(.LC2)
49     addi a0,a0,%lo(.LC2) #加载字符串"请输入第二个浮点数:"地址
50     call printf #打印"请输入第二个浮点数:"
51     addi a1,sp,12
52     addi a0,s0,%lo(.LC1) #字符串"%f"
53     call __isoc99_scanf #调用scanf函数读取用户输入的浮点数, 存储到
        栈上 (sp+12)
54
55     flw fa5,8(sp) #从栈上加载第一个浮点数到浮点寄存器fa5
56     flw fa4,12(sp) #从栈上加载第二个浮点数到浮点寄存器fa4
57     fgt.s a5,fa5,fa4 #比较两个浮点数, 结果存储在整数寄存器a5中 (如果
        fa5 > fa4, 则a5非零)
58     fcvt.d.s fa4,fa4 #将单精度浮点数转为双精度浮点数

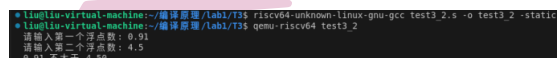
```

```

59      fcvt.d.s      fa5,fa5    #将单精度浮点数转为双精度浮点数
60      fmv.x.d a2,fa4
61      fmv.x.d a1,fa5    #存储到整数寄存器
62      beq      a5,zero,.L6    #如果a5为0, 则跳转L6
63      lui      a0,%hi(.LC3)
64      addi     a0,a0,%lo(.LC3)  #大于, 加载 "%.2f 大于 %.2f\n"
65      call     printf    #打印结果
66
67 .L4:
68     #结束
69     #恢复寄存器以及栈, 跳转返回地址
70     ld       ra,24(sp)
71     ld       s0,16(sp)
72     li       a0,0
73     addi     sp,sp,32
74     jr       ra
75 .L6:
76     lui      a0,%hi(.LC4)
77     addi     a0,a0,%lo(.LC4)    #不大于, 加载 "%.2f 不大于 %.2f\n"
78     call     printf    #打印结果
79     j        .L4    #跳转结束
80     .size    main,.-main
81     .ident   "GCC: () 12.2.0"
82     .section .note.GNU-stack,"",@progbits

```

将汇编程序进行汇编链接形成可执行文件, 运行 `riscv64-unknown-linux-gnu-gcc test3_2.s -o test3_2 -static qemu-riscv64 test3_2` 输入 0.91 4.5, 得到 0.91 不大于 4.50 的输出, 如图28所示。



```

linlin@linlin-virtual-machine:~/编译实验/abi/131 riscv64-unknown-linux-gnu-gcc test3_2.s -o test3_2 -static
linlin@linlin-virtual-machine:~/编译实验/abi/131 qemu-riscv64 test3_2
请输入第一个浮点数: 0.91
请输入第二个浮点数: 4.5
0.91 不大于 4.50

```

图 28: 程序 3 汇编代码执行结果

五、 总结

本次实验深入探索了 C 程序从编写到最终执行的全过程, 尤其是编译过程中的各个阶段, 包括预处理、编译、汇编、链接以及执行。帮助我们理解了计算机程序如何从高级语言转化为机器可执行代码, 也认识到编译系统内部的复杂性和精妙。在本次实验过程中通过对编译的各阶段拆分探索, 我们认识到编译器不仅是将高级语言转换为低级语言的工具, 更是优化代码、提高程序性能的关键。学习 LLVM IR 中间代码编写的过程中, 我们认识到编译器设计的灵活性和可扩展性, 通过中间表示, 编译器可以更容易地进行跨平台编译和代码优化。在学习 RISC-V 汇编语言编写的过程中, 不仅认识了 RISC-V 这样一个新兴的、有巨大发展潜力的指令集, 也在编写中体会到计算机执行复杂代码时实现的底层逻辑。本次实验进一步深刻了我们对编译系统的认识, 为后续更加深入的学习奠定基础。

六、 GitHub 链接

GitHub 仓库连接 [GitHub](#)

NKU