



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

---

## 预备实验

---

张政泽

年级：2022 级

专业：信息安全

指导教师：王刚

2024 年 9 月 23 日

# 摘要

关键字：GCC, LLVM/Clang, 预处理器, 编译器, 汇编器, 链接器, RISC-V

# 目录

一、 引言	1
二、 完整的编译过程	2
(一) 预处理器	2
(二) 编译器	4
1. 词法分析	4
2. 语法分析	5
3. 语义分析	7
4. 中间代码生成	7
5. 代码优化	8
6. 代码生成	9
(三) 汇编器	10
(四) 链接器/加载器	12
三、 LLVM IR 程序	12
(一) 阶乘	13
(二) 冒泡排序	15
四、 RISC-V 汇编程序	20
(一) 浮点数累加	20
(二) 冒泡排序	24
(三) 对比分析总结	28

分工：小组成员分别为张政泽和徐俊智，第一部分为两人分别撰写各自的部分，第二部分 LLVM IR 编程部分由队友徐俊智负责，第三部分 RISC-V 汇编编程部分由本人张政泽负责。

## 一、 引言

我们将以一个简单的 SysY 源程序为例，在 vscode 上配置 WSL 环境（Ubuntu22.04），以 GCC/LLVM/Clang 等编译工具为研究对象，逐步探究语言处理系统的完整工作流程。

代码如下：

斐波那契程序

```
1 //包含头文件
2 #include<iostream>
3 using namespace std;
4 //包含宏定义
5 #define str "StringTest"
6 int i=1;
7 //包含普通函数函数
8 int GetN() {
9     int n;
10    cin>>n;
11    return n;
12 }
13 //包含内联函数
14 inline int Fib(int a,int b,int n){
15     int t;
16     while(i<n){
17         t=b;
18         b=a+b;
19         cout<<b<<endl;
20         a=t;
21         i=i+1;
22     }
23 }
24 }
25 int main()
26 {
27     int a,b,t;
28
29     a=0;
30     b=1;
31     int n=GetN();
32     cout<<a<<endl;
33     cout<<b<<endl;
34     Fib(a,b,n);
35     cout<<str;
36 }
37 }
```

代码中包含了头文件、宏定义、普通函数、内联函数、主函数等不同的部分，用来探究语言处理系统如何对各个不同的部分进行处理。代码中包含了

- iostream 头文件
- 一个宏定义 `str "StringTest"`
- 普通函数 `GetN()`，用于获取用户输入
- 内联函数 `Fib()`，接受用户输入并返回斐波那契计算结果

## 二、完整的编译过程

### (一) 预处理器

预处理器的作用是编译前根据预处理指令处理源代码，预处理指令以 `#` 开头，如 `#include`、`#define`、`#if` 等等，对 `#include` 指令会替换对应的头文件，对 `#define` 的宏命令会直接替换相应内容，`#ifdef` 作为条件编译指令，会在条件为 `True` 时对部分程序源代码进行编译，除此之外，在预处理阶段还会删除注释，添加行号和文件名标识。

使用 `g++` 的 `-E` 选项生成预处理后的文件：

```
1 g++ source.cpp -E -o source.i
```

```
root@ZZZZJYMF:/home/Compile# gcc source.cpp -E -o source.i
root@ZZZZJYMF:/home/Compile# ls
SysYTest1.sy SysYTest3.sy source source.i test1.s test2.s test3.s testtest.cpp testtest2 testtest2.s
SysYTest2.sy Token.txt source.cpp test1 test2 test3 testtest testtest.s testtest2.cpp
```

打开 `source.i` 文件，发现整个文件的长度有整整 32293 行，远远超过了初始 `cpp` 文件的大小 (40 行) 文件，经过对比分析发现 `source.i` 文件相较于 `source.cpp` 文件存在以下区别。

- `.cpp` 文件中 `include` 包含的头文件在 `.i` 文件中被替换为了具体的内容

```
C source.i > ...
1 # 0 "source.cpp"
2 # 0 "<built-in>"
3 # 0 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
5 # 0 "<command-line>" 2
6 # 1 "source.cpp"
7
8 # 1 "/usr/include/c++/11/iostream" 1 3
9 # 36 "/usr/include/c++/11/iostream" 3
10
11 # 37 "/usr/include/c++/11/iostream" 3
12
13 # 1 "/usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 1 3
14 # 278 "/usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
15
16 # 278 "/usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
17 namespace std
18 {
19     typedef long unsigned int size_t;
20     typedef long int ptrdiff_t;
21
22
23     typedef decltype(nullptr) nullptr_t;
24
25 }
```

- .i 文件中将.cpp 文件中预定义的宏转换为了具体的值  
宏定义及.cpp 文件内容如下：

```
4    //包含宏定义
5    #define  str "StringTest"
```

而.i 文件中的内容为:

```
27    int main()
28    {
29        int a,b,t;
30
31        a=0;
32        b=1;
33        int n=GetN();
34        cout<<a<<endl;
35        cout<<b<<endl;
36        Fib(a,b,n);
37
38        cout<<str;
39
40    }
41
```

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

    a=0;
    b=1;
    int n=GetN();
    cout<<a<<endl;
    cout<<b<<endl;
    Fib(a,b,n);

    cout<<"StringTest";
}
```

- .i 文件中去除了.cpp 文件中的注释

## (二) 编译器

编译器经过词法分析、语法分析、语义分析、中间代码生成、代码优化以及代码优化等几个阶段，将源代码转换为汇编代码，具体分析如下：

### 1. 词法分析

通过以下将预处理后的程序转换为单词序列 (Tokens)，获得 token 序列：

```
1 clang -E -Xclang -dump-tokens source.cpp
```

```
root@ZZZZJYWF:/home/Compile# clang -E -Xclang -dump-tokens source.cpp
```

部分 token 序列可见 [Token.txt 文件](#) 将获取的 token 序列与上一步生成的.i 文件中的内容进行对比，发现词法分析在预处理器生成的文件的基础上进行了分词操作，将源代码文本分割成了一系列的 token 流，并标识了每一个分词在源代码中的位置；除此之外，还对分词的结果进行了属性的识别，包括标识符 (identifier)、运算符 (更进一步包括 >、<、+ 等)、字面量 (string\_literal)、分隔符 (、;、(、) 等)、高级语言中的关键字 (int、return、while、inline、extern 等)

接下来，以 token 序列中的部分内容为例，分析它们在 token 序列中的表示形式。

- identifier 'n' [LeadingSpace] Loc=<source.cpp:33:6>
  - Token 类型: identifier

- Token 文本: "n"
  - 属性: [LeadingSpace], 表示在此 token 之前存在空白字符
  - 位置信息: Loc=<source.cpp:33:6>, 表明这个 token 的起始位置是 source.cpp 中第 33 行的第 6 个字符。
- while 'while' [StartOfLine] [LeadingSpace] Loc=<source.cpp:18:2>
    - Token 类型: while 关键字
    - Token 文本: "while"
    - 属性: [StartOfLine], 表示此 while 关键字位于新的一行的开始位置。
    - 位置信息: Loc=<source.cpp:18:2>, 表明这个 token 的起始位置是 source.cpp 中第 18 行的第 2 个字符。

## 2. 语法分析

根据语言的语法规则, 用词法分析生成的词法单元来构建抽象语法树 (AST)。LLVM 可以通过以下命令获得相应的 AST:

```
1 clang -E -Xclang -ast-dump source.cpp
```

```
root@ZZZZJYWF:/home/Compile# clang -E -Xclang -ast-dump source.cpp
```

部分 AST 可见 [AST.txt 文件](#)

其中 main 函数对应的语法树的部分为 (为了便于理解在此处对其进行了简化)

```
1 -FunctionDecl main 'int ()'
   -CompoundStmt
2   |-DeclStmt
3   | |-VarDecl a 'int'
4   | |-VarDecl b 'int'
5   | -VarDecl t 'int'
6   |-BinaryOperator a=0
7   |-BinaryOperator b=1
8
9   |-DeclStmt
10  | -VarDecl n = GetN()
11
12  |-CXXOperatorCallExpr '<<'
13  | |-CXXOperatorCallExpr '<<'
14  | | |-DeclRefExpr 'cout'
15  | | -ImplicitCastExpr 'int' a
16  | -DeclRefExpr 'endl'
17
18  |-CXXOperatorCallExpr '<<'
19  | |-CXXOperatorCallExpr '<<'
20  | | |-DeclRefExpr 'cout'
21  | | -ImplicitCastExpr 'int' b
22  | -DeclRefExpr 'endl'
```

```

23 | -CallExpr  'int'
24 | | -DeclRefExpr  'Fib' 'int'(a,b,n)'
25
26 | -CXXOperatorCallExpr '<<'
27 | | -DeclRefExpr  'cout'
28 | | -StringLiteral  "StringTest"

```

对节点中部分单词含义的解释如下：

1. FunctionDecl: 表示函数声明
2. CompoundStmt: 表示复合语句，如函数体
3. DeclStmt: 变量声明语句
4. VarDecl: 表示变量声明
5. BinaryOperator: 二元运算符
6. IntegerLiteral: 整型字面量
7. CXXOperatorCallExpr: C++ 运算符重载函数调用
8. ImplicitCastExpr: 隐式类型转换
9. DeclRefExpr: 对变量、函数或成员的直接引用
10. callExpr: 函数调用表达式

在上述简化的部分抽象语法树中，已经将各部分对照 cpp 程序源代码按照功能进行了模块的区分，我们以其中的一句输出为例来进行具体的分析

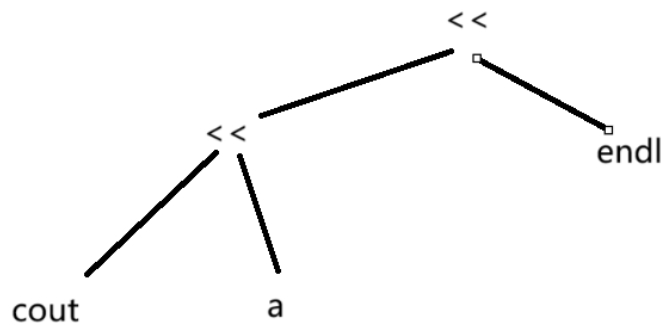
```

1 | -CXXOperatorCallExpr '<<'
2 | | -CXXOperatorCallExpr '<<'
3 | | | -DeclRefExpr  'cout'
4 | | | -ImplicitCastExpr  'int' a3
5 | | | -DeclRefExpr  'endl'

```

上述 AST 语句中，每行中的'|' 字符的个数表示了此元素位于树形结构的层数，再结合其树形结构，由此我们可以得出：在这样一个输出语句当中，根节点为"«" 符号，其两个孩子节点分别为另一个"«" 符号和字符串"endl"，而此输出符号的左右节点又分别为字符"cout" 和字符串"a"，将其表示为树形结构大致如下图所示：





按照中缀表达式的方式将此树形结构转换为字符序列便得到了对应的 C++ 程序源码：  
`cout<<a<<endl`, 其他各部分的树形结构也都可以采用相似的方式转换为更加易读的树形结构。

### 3. 语义分析

使用语法树和符号表中信息来检查源程序是否与语言定义语义一致，进行类型检查、变量声明检查等。

### 4. 中间代码生成

完成上述步骤后，很多编译器会将 AST 转换为一个明确的低级或类机器语言的中间表示（如 LLVM IR）。可以通过 `-fdump-tree-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，生成的 `.dot` 文件可以被 `graphviz` 可视化，能够看到控制流图（CFG），以及各阶段处理中（比如优化、向 IR 转换）CFG 的变化。可以额外使用 `-Ox`、`-fno-*` 等 flag 控制编译行为，使输出文件更可读、了解其优化行为。对于 LLVM，可以通过以下命令生成 LLVM IR：

```
1 clang -S -emit-llvm source.cpp
```

打开 `source.ll` 文件，可以看到生成的 LLVM IR。

```

58
59 ; Function Attrs: mustprogress noline norecurse optnone uwtable
60 define dso_local noundef i32 @main() #5 {
61     %1 = alloca i32, align 4
62     %2 = alloca i32, align 4
63     %3 = alloca i32, align 4
64     %4 = alloca i32, align 4
65     store i32 0, i32* %1, align 4
66     store i32 1, i32* %2, align 4
67     %5 = call noundef i32 @_Z4GetNv()
68     store i32 %5, i32* %4, align 4
69     %6 = load i32, i32* %1, align 4
70     %7 = call noundef nonnull align 8 dereferenceable(8) @"class.std::basic_ostream"* @_ZNSolsEi(@"class.std::basic_o
71     %8 = call noundef nonnull align 8 dereferenceable(8) @"class.std::basic_ostream"* @_ZNSolsEPFRSoS_E(@"class.std::i
72     %9 = load i32, i32* %2, align 4
73     %10 = call noundef nonnull align 8 dereferenceable(8) @"class.std::basic_ostream"* @_ZNSolsEi(@"class.std::basic_
74     %11 = call noundef nonnull align 8 dereferenceable(8) @"class.std::basic_ostream"* @_ZNSolsEPFRSoS_E(@"class.std:
75     %12 = load i32, i32* %1, align 4
76     %13 = load i32, i32* %2, align 4
77     %14 = load i32, i32* %4, align 4
78     %15 = call noundef i32 @_Z3Fibiii(i32 noundef %12, i32 noundef %13, i32 noundef %14)
79     %16 = call noundef nonnull align 8 dereferenceable(8) @"class.std::basic_ostream"* @_ZStlsISt11char_traitsIcEERSt
80     ret i32 0
81 }
82

```

生成的 LLVM IR 文件的大小远小于预处理器生成的.i 文件，此时产生的 LLVM IR 语言已经接近于汇编语言，但比汇编语言的可读性更高

LLVM IR 文件与目标机器代码的关系如下图所示：



相较于直接生成机器代码，采用先生成中间代码再生成机器代码的方式有以下优点：

1. 生成中间代码后可以对中间代码进行优化以提升程序质量和效率
2. 如上图所示，生成的中间代码可以进一步地生成包括 x86、ARM、RISC-V、MIPS、PowerPC 在内的汇编代码，这增强了编译器的移植性

## 5. 代码优化

对中间代码进行与机器无关的优化，生成更好的目标代码。

在 LLVM 中，代码优化可以通过以下三种 pass 来完成：Analysis passes、Transform passes、Utility passes。

- Analysis Passes 不直接修改代码，而是计算其他 pass 可以使用的信息，例如别名分析、控制流分析、循环分析等。
- Transform passes 通过改变程序的结构来优化代码，可以删除无用代码、优化循环、内联函数等。

- Utility passes 提供一些辅助功能，例如为匿名指令分配名称以及模块验证等。

编译器的代码优化有四个等级，分别为 O0、O1、O2 和 O3，不同的等级在代码编译时提供了不同的优化程度，随着优化等级的提高执行效率提高的同时也会带来编译时间的增加。

- O0: 代表无优化，也就是说会保持代码原有特征，通常用于调试
- O1(初级优化): 执行一些基本优化，如删除未使用的变量、内联简单函数等
- O2(中极优化): 在 O1 基础上进行更多的优化操作，如更大范围的内联、循环展开、函数调用图优化等
- O3(高级优化): 在 O2 基础上进行更深入的优化，如更大范围的内联、循环变形、自动向量化等

可以通过以下命令指定使用某个 pass 来进行优化：

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

在此之前我们需要获取.bc 格式的文件用于优化过程，可以通过以下指令进行.ll 文件和.bc 文件的互转：

```
1 llvm-dis a.bc -o a.ll #bc转换为ll
2 llvm-as a.ll -o a.bc #ll转换为bc
```

若希望直接使用.ll 格式文件进行代码优化，可以通过添加命令行参数，指令如下：

```
1 opt -O3 source.ll -o optimized.ll #指定优化级别来启用不同级别优化
2 opt -mem2reg source.ll -o mem2reg.ll #指定特定优化pass(内存到寄存器的晋升Pass)
3 opt -instcount source.ll -o instcount.ll #分析Pass(使用指令计数来获取关于IR的信息)
```

我们以 O3 优化为例，查看优化后的 source\_O3.ll 文件，并与未优化的 source.ll 文件进行对比观察优化效果。

```
1 llvm-as source.ll -o source.bc #ll转换为bc
2 opt -O3 source.bc -o sourceO3.bc #输出重定向到sourceO2.bc文件
3 llvm-dis sourceO3.bc -o sourceO3.ll #bc转换为ll
```

source.ll 及优化生成的 sourceO3.ll 文件可见

source.ll 文件

sourceO3.ll 文件

经过对比未发现两个文件存在明显的差异，猜测可能是因为源程序过于简单。

如果想要查看每个优化传递前后的 LLVM IR，可以使用 llc 工具，并重定向输出到日志文件。2>&1 是重定向描述，意为所有输出（包括错误）都会被写入 main.log。

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

## 6. 代码生成

将优化后的中间代码转换为汇编代码。

```
1 g++ source.i -S -o source.S #生成x86格式目标代码
2 llc sourceO3.ll -o source_llvm.s #LLVM生成目标代码
```

生成的汇编代码文件可见

source.S 文件

source\_llvm.S 文件

### (三) 汇编器

```
1 g++ source.s -c -o source.o # x86格式可以直接使用g++完成汇编
2 llc source.bc -filetype=obj -o source_llvm.o # LLVM可以直接使用llc命令同时编译和汇编LLVM bitcode
```

执行命令后汇编器会基于.s 汇编程序生成.o 文件. 在这一过程中, 汇编语言被翻译成目标机器指令, 最终生成可重定位的机器代码。

汇编语言接近于机器代码, 但仍然包含一些人类可读的伪指令和助记符等, 在汇编器处理的过程中会将这些助记符转换成机器可以执行的二进制代码

根据额外的资料得知, 汇编器在处理过程中的具体功能包括以下几点:

- 指令选择: 即将中间表示中的操作转换为目标架构上的具体机器指令, 具体方法包括树匹配方法 (将 IR 操作表示为树结构并尝试寻找与目标指令集匹配的树模式) 和选择 DAG (IR 操作转换成选择 DAG 结构并寻找将这个 DAG 映射到目标指令集的最优方式)
- 寄存器分配: 在选择之后通过寄存器分配将变量映射到处理器的实际硬件寄存器, 这一步骤旨在通过优化寄存器的使用来减少访问内存的次数进而提高执行速度
- 指令调度: 在这一步中, 通过分析指令的 (数据和控制) 依赖关系并选择合适的调度策略来对指令进行重排, 来提高程序的执行效率和处理器的吞吐量
- 指令编码: 在之前三步的基础上这一过程通过对目标架构指令集进行分析将指令表示转换为目标机器可以识别和执行的格式即最终的二进制代码的形式

我们使用如下指令对生成的 obj 文件进行反汇编得到.s 汇编程序。

```
1 objdump -d source.o > source-anti-obj.S
```

反汇编生成的文件可见source-anti-obj.S 文件

与汇编器处理前的汇编程序进行对比, 发现存在以下的差异:

1. 反汇编代码中的地址与原始汇编代码中的地址不同, 反汇编代码中的地址更加精确, 且更多地利用相对地址。
2. 或许因为程序过于简单的原因, 经过对 main 函数和 Fib 函数的对比后未发现指令经过重排
3. 原始汇编代码中信息更加地丰富 (如包含类型信息、节信息等), 而反编译得到的代码中则缺少这部分内容, 几乎全部是汇编代码

使用 hexdump -C 选项以十六进制形式查看文件内容

```
1 hexdump -C source.o
```

```

root@ZZZZJYWF:/home/Compile# hexdump -C source.o
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |.ELF.....|
00000010 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00 00 |...>.....|
00000020 00 00 00 00 00 00 00 00 20 0a 00 00 00 00 00 00 |.....|
00000030 00 00 00 00 40 00 00 00 00 00 40 00 13 00 01 00 |....@.....@....|
00000040 55 48 89 e5 48 83 ec 10 48 8b 3d 00 00 00 00 48 |UH..H...H.=...H|
00000050 8d 75 fc e8 00 00 00 00 8b 45 fc 48 83 c4 10 5d |.u.....E.H...]|
00000060 c3 66 2e 0f 1f 84 00 00 00 00 00 0f 1f 44 00 00 |.f.....D...|
00000070 55 48 89 e5 48 83 ec 10 c7 45 f8 00 00 00 00 c7 |UH..H...E.....|
00000080 45 fc 01 00 00 00 e8 00 00 00 89 45 f4 8b 75 |E.....E..u|
00000090 f8 48 8b 3d 00 00 00 00 e8 00 00 00 00 48 8b 35 |.H.=.....H.5|
000000a0 00 00 00 00 48 89 c7 e8 00 00 00 00 8b 75 fc 48 |....H.....u.H|
000000b0 8b 3d 00 00 00 00 e8 00 00 00 48 8b 35 00 00 |.=.....H.5..|
000000c0 00 00 48 89 c7 e8 00 00 00 00 8b 7d f8 8b 75 fc |..H.....}.u.|
000000d0 8b 55 f4 e8 00 00 00 00 48 8b 3d 00 00 00 00 48 |.U.....H.=...H|
000000e0 be 00 00 00 00 00 00 00 00 e8 00 00 00 31 c0 |.....1.|
000000f0 48 83 c4 10 5d c3 00 00 00 00 00 00 00 00 00 00 |H...]|.....|
00000100 55 48 89 e5 bf 00 00 00 00 e8 00 00 00 00 48 8b |UH.....H.|
00000110 3d 00 00 00 00 be 00 00 00 00 ba 00 00 00 00 e8 |=.....|
00000120 00 00 00 00 5d c3 66 2e 0f 1f 84 00 00 00 00 00 |....].f.....|
00000130 55 48 89 e5 e8 c7 ff ff ff 5d c3 00 00 00 00 00 |UH.....].....|
00000140 55 48 89 e5 48 83 ec 10 89 7d f8 89 75 fc 89 55 |UH..H...}.u..U|
00000150 f0 8b 04 25 00 00 00 00 3b 45 f0 7d 46 8b 45 fc |...%....;E.}F.E.|

00000830 30 00 00 00 00 00 00 00 00 2e 72 65 6c 61 2e 69 |0.....rela.i|
00000840 6e 69 74 5f 61 72 72 61 79 00 5f 5a 34 47 65 74 |nit_array._Z4Get|
00000850 4e 76 00 5f 5a 4e 53 74 38 69 6f 73 5f 62 61 73 |Nv._ZNSt8ios_bas|
00000860 65 34 49 6e 69 74 44 31 45 76 00 5f 5a 4e 53 74 |e4InitD1Ev._ZNSt|
00000870 38 69 6f 73 5f 62 61 73 65 34 49 6e 69 74 43 31 |8ios_base4InitC1|
00000880 45 76 00 2e 72 65 6c 61 2e 74 65 78 74 00 5f 5a |Ev..rela.text._Z|
00000890 53 74 34 63 6f 75 74 00 2e 63 6f 6d 6d 65 6e 74 |St4cout..comment|
000008a0 00 5f 5f 63 78 61 5f 61 74 65 78 69 74 00 5f 5a |. __cxa_atexit._Z|
000008b0 53 74 4c 38 5f 5f 69 6f 69 6e 69 74 00 5f 5f 63 |StL8_ioinit.__c|
000008c0 78 78 5f 67 6c 6f 62 61 6c 5f 76 61 72 5f 69 6e |xx_global_var_in|
000008d0 69 74 00 2e 62 73 73 00 2e 72 65 6c 61 2e 74 65 |it..bss..rela.te|
000008e0 78 74 2e 73 74 61 72 74 75 70 00 2e 67 72 6f 75 |xt.startup..grou|
000008f0 70 00 5f 47 4c 4f 42 41 4c 5f 5f 73 75 62 5f 49 |p._GLOBAL__sub_I|
00000900 5f 73 6f 75 72 63 65 2e 63 70 70 00 5f 5a 53 74 |_source.cpp._ZSt|
00000910 33 63 69 6e 00 6d 61 69 6e 00 2e 6e 6f 74 65 2e |3cin.main..note.|
00000920 47 4e 55 2d 73 74 61 63 6b 00 2e 72 65 6c 61 2e |GNU-stack..rela.|
00000930 74 65 78 74 2e 5f 5a 33 46 69 62 69 69 69 00 5f |text._Z3Fibiii._|
00000940 5a 4e 53 69 72 73 45 52 69 00 5f 5a 4e 53 6f 6c |ZNSirsERi._ZNSol|
00000950 73 45 69 00 2e 72 65 6c 61 2e 65 68 5f 66 72 61 |sEi..rela.eh_fra|
00000960 6d 65 00 5f 5f 64 73 6f 5f 68 61 6e 64 6c 65 00 |me.__dso_handle.|
00000970 5f 5a 53 74 6c 73 49 53 74 31 31 63 68 61 72 5f |_ZStlsIcSt11char_|
00000980 74 72 61 69 74 73 49 63 45 45 52 53 74 31 33 62 |traitsIcEERSt13b|
00000990 61 73 69 63 5f 6f 73 74 72 65 61 6d 49 63 54 5f |asic_ostreamIcT_|
000009a0 45 53 35 5f 50 4b 63 00 2e 73 74 72 74 61 62 00 |ES5_PKc..strtab.|
000009b0 2e 73 79 6d 74 61 62 00 2e 64 61 74 61 00 5f 5a |.symtab..data._Z|
000009c0 53 74 34 65 6e 64 6c 49 63 53 74 31 31 63 68 61 |St4endlIcSt11cha|
000009d0 72 5f 74 72 61 69 74 73 49 63 45 45 52 53 74 31 |r_traitsIcEERSt1|
000009e0 33 62 61 73 69 63 5f 6f 73 74 72 65 61 6d 49 54 |3basic_ostreamIT|

```

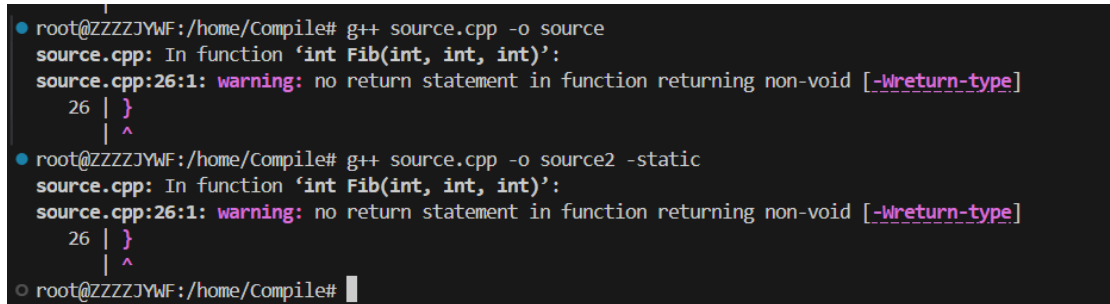
生成的目标文件（Object File）包含了机器代码和导入函数等信息。

#### (四) 链接器/加载器

大型文件通常被分成多个部分进行编译，因此上一步生成的.o 文件不能直接执行，必须和其他库文件通过链接器链接到一起最终形成可执行文件才能执行。

可以通过以下指令来生成可执行文件：

```
1 g++ source.cpp -o source
2 g++ source.cpp -o source2 -static #指定静态链接
```



```
root@ZZZZJYWF:/home/Compile# g++ source.cpp -o source
source.cpp: In function 'int Fib(int, int, int)':
source.cpp:26:1: warning: no return statement in function returning non-void [-Wreturn-type]
    26 | }
        | ^
root@ZZZZJYWF:/home/Compile# g++ source.cpp -o source2 -static
source.cpp: In function 'int Fib(int, int, int)':
source.cpp:26:1: warning: no return statement in function returning non-void [-Wreturn-type]
    26 | }
        | ^
root@ZZZZJYWF:/home/Compile#
```

链接过程分为静态链接和动态链接。在 Windows 操作系统下，静态库通常具有.lib 扩展名，而动态库通常具有.dll 扩展名；而在 Linux 操作系统下，静态库具有.a 扩展名，而动态库（共享库）通常具有.so 扩展名。

静态链接发生在编译时，所有需要使用的库和代码都会在编译过程中被链接到最终的可执行文件当中。这样的方式由于包含了所有需要使用的代码和库文件，因此其生成的可执行文件的大小会增加，但同时程序的启动速度会更快。除此之外，静态链接的可执行文件由于不依赖于系统中的动态链接库，因此可以在没有安装这些库的系统上运行。

动态链接发生在程序运行时，程序在运行时动态地加载其所需要的库，由于不包含所依赖的链接库的文件，因此这样生成的可执行文件较小并且生成的可执行程序不受动态链接库更新的影响，但由于在启动时需要加载动态库，因此可能会增加程序的启动时间。

加载器的主要作用是在执行程序时，将程序的可执行文件加载到内存的数据和代码段中，使得程序可以被处理器执行。

### 三、 LLVM IR 程序

LLVM 的组成部分包括：

1. 前端 (front-end)：将源语言翻译成 IR。
2. 中端 (middle-end)：各种 pass 完成对 IR 的优化。
3. 后端 (back-end)：把优化后的 IR 翻译成目标语言。

LLVM IR 是相对于 CPU 指令集更为高级、但相对于源程序更为低级的代码中间表示的一种语言，由代码生成器自顶向下遍历逐步翻译语法树形成，然后由 LLVM 后端对 LLVM IR 进行优化并编译为相应平台的二进制程序。其组成结构大概为：

1. IR 的基本单位是 module。



2. 一个 module 中拥有多个顶层实体，例如 function 和 global variable
3. function 的 define 中至少有一个 basicblock。
4. 每个 basicblock 中包含多个 instruction。

### (一) 阶乘

我们尝试为阶乘程序编写 LLVM IR 程序。

源代码如下：

```

1 // 阶乘
2 #include <stdio.h>
3 int main() {
4     int i, n, f;
5     printf("请输入一个整数: ");
6     scanf("%d", &n); // 读取整数 n
7     i = 2;
8     f = 1;
9     while (i <= n) {
10         f = f * i; // 计算阶乘
11         i = i + 1;
12     }
13     printf("%d 的阶乘是: %d\n", n, f);
14     return 0;
15 }
```

LLVM IR 代码如下：

```

1 @.str = private unnamed_addr constant [25 x i8] c"请输入一个整数: \00", align
  1
2 @.str1 = private unnamed_addr constant [3 x i8] c"%d\00", align 1
3 @.str2 = private unnamed_addr constant [22 x i8] c"%d 的阶乘是: %d\0A\00",
  align 1
4
5 declare i32 @printf(i8*, ...)
6
7 declare i32 @scanf(i8*, ...)
8
9 define dso_local noundef i32 @main() #0 {
10     ; 变量声明
11     %1 = alloca i32, align 4          ; i -> %1
12     %2 = alloca i32, align 4          ; n -> %2
13     %3 = alloca i32, align 4          ; f -> %3
14     ; 调用 printf , 输出字符串"请输入一个整数: "
15     %4 = call i32 @printf(i8*, ...) @printf(i8* noundef getelementptr inbounds ([25
      x i8], [25 x i8]* @.str, i64 0, i64 0))
16     ; 调用 scanf , 输入 n
17     %5 = call i32 @scanf(i8*, ...) @scanf(i8* noundef getelementptr inbounds ([3 x
      i8], [3 x i8]* @.str1, i64 0, i64 0), i32* noundef %2)
```

```

18     ; i、f赋初值
19     store i32 2, i32* %1, align 4      ; i = 2
20     store i32 1, i32* %3, align 4      ; f = 1
21     br label %6
22
23 6:   ; while循环判断                    ; preds = %10, %0
24     %7 = load i32, i32* %1, align 4    ; 读取i
25     %8 = load i32, i32* %2, align 4    ; 读取n
26     %9 = icmp sle i32 %7, %8           ; 判断i是否小于等于n
27     br i1 %9, label %10, label %15
28
29 10:                                     ; preds = %5
30     %11 = load i32, i32* %3, align 4    ; 读取f
31     %12 = load i32, i32* %1, align 4    ; 读取i
32     %13 = mul i32 %11, %12             ; f = f * i
33     store i32 %13, i32* %3, align 4     ; 更新f
34     %14 = add i32 %12, 1               ; i = i + 1
35     store i32 %14, i32* %1, align 4     ; 更新i
36     br label %6
37
38 15:
39     %16 = load i32, i32* %2, align 4    ; 读取n
40     %17 = load i32, i32* %3, align 4    ; 读取f
41     ; 调用 printf , 输出结果
42     %18 = call i32 @printf(i8* noundef getelementptr inbounds ([22
        x i8], [22 x i8]* @.str2, i64 0, i64 0), i32 noundef %16, i32
        noundef %17)
43     ret i32 0
44 }

```

我们将 LLVM IR (test.ll) 转换为目标文件 (test.o)，再链接目标文件生成可执行文件。在编译时使用 -fPIE 选项，确保生成位置无关的代码。

```

1 llc -filetype=obj -relocation-model=pic test.ll -o test.o
2 clang++ -fPIE test.o -o test

```

运行生成的可执行文件。

输入 4，输出 24，运行成功！

我们的源代码保存在 factorial.c 文件中，通过以下命令生成 LLVM IR。

```

1 clang -S -emit-llvm -o factorial.ll factorial.c

```

我们将 Clang 解析源代码生成的 factorial.ll 与自己写的 test.ll 进行比较。

可以发现，在 factorial.ll 的开头定义了模块信息，包括模块 ID、源文件名、目标数据布局和目标三元组。



factorial.ll 的 main 函数会为%1 分配内存但在后续却不使用%1, 猜测是因为编译器保留了这个变量以帮助调试或为了将来可能的优化。

factorial.ll 的 while 循环部分, 在进行乘法、加法操作时使用 nsw 来表示不产生溢出。在循环结束时, 给跳转指令增加了llvm.loop 属性, 表示循环必须有进度, 避免了死循环。

factorial.ll 声明的是 \_\_isoc99\_scanf 函数而不是 scanf 函数, 查阅资料得知, 在 ISO C99 标准中, 对 scanf 函数的行为做了一些修改, 主要是为了提高类型安全性和清晰性。scanf 和 \_\_isoc99\_scanf 之间的区别包括:

- 类型安全性: C99 标准要求格式字符串和相应的参数类型严格匹配。如果类型不匹配, scanf 会返回一个错误, 而不是尝试进行类型转换。
- %n 转换: 在 C99 之前的 scanf 版本中, %n 转换可以用于任何类型的输入, 而在 C99 中, %n 只能用于 int 类型的变量。
- 长度修饰符: C99 标准引入了更多的长度修饰符, 如%zu (用于 size\_t 类型), %j (用于 intmax\_t 和 uintmax\_t 类型), 这些在旧版本的 scanf 中不可用。
- 浮点数输入: 在 C99 之前的 scanf 版本中, %f、%e 和%g 转换可以用于 float、double 和 long double 类型的变量, 而在 C99 中, %f 只能用于 float 类型, %lf 用于 double 类型, %Lf 用于 long double 类型。

在 factorial.ll 的末尾还定义了属性和元数据。属性用于描述函数或模块的特性, 可以影响代码生成和优化。元数据用于附加信息到 LLVM IR 的元素上, 例如函数、基本块等, 通常用于优化、调试或与编译器的其他部分交互。元数据不影响生成的机器代码, 但提供了额外的上下文信息。

## (二) 冒泡排序

我们尝试为冒泡排序程序编写 LLVM IR 程序。

源代码如下:

```
1 // 冒泡排序
2 #include <stdio.h>
3 int main() {
4     float arr[10];
5     printf("请输入10个浮点数: \n");
6     for (int i = 0; i < 10; i++) {
7         scanf("%f", &arr[i]);
8     }
```

```

9
10     for (int i = 0; i < 9; i++) {
11         for (int j = 0; j < 9 - i; j++) {
12             if (arr[j + 1] < arr[j]) {
13                 float temp = arr[j];
14                 arr[j] = arr[j + 1];
15                 arr[j + 1] = temp;
16             }
17         }
18     }

19     printf("排序后的数字为: \n");
20     for (int i = 0; i < 10; i++) {
21         printf("%f", arr[i]);
22     }
23     printf("\n");
24
25     return 0;
26 }
27

```

LLVM IR 代码如下:

```

1  @.str = private unnamed_addr constant [27 x i8] @c"请输入10个浮点数: \00",
    align 1
2  @.str1 = private unnamed_addr constant [3 x i8] @c"%f\00", align 1 ; "%f" 字符
    串
3  @.str2 = private unnamed_addr constant [25 x i8] @c"排序后的数字为: \00",
    align 1
4  @.str3 = private unnamed_addr constant [4 x i8] @c"%f \00", align 1 ; "%f " 格
    式化指令
5  @.str4 = private unnamed_addr constant [2 x i8] @c"\0A\00", align 1 ; 换行符
6
7  declare i32 @printf(i8*, ...)
8
9  declare i32 @scanf(i8*, ...)
10
11 define dso_local noundef i32 @main() #0 {
12     ; 变量声明
13     %1 = alloca [10 x float], align 16 ; arr -> %1
14     %2 = alloca i32, align 4 ; i1 -> %2
15     %3 = alloca i32, align 4 ; i2 -> %3
16     %4 = alloca i32, align 4 ; j -> %4
17     %5 = alloca float, align 4 ; temp -> %5
18     %6 = alloca i32, align 4 ; i3 -> %6
19     ; 调用 printf , 输出字符串"请输入10个浮点数: "
20     %7 = call i32 @i8*, ... @printf(i8* getelementptr inbounds ([27 x i8],
        [27 x i8]* @.str, i64 0, i64 0))
21     ; 调用 printf , 换行
22     %8 = call i32 @i8*, ... @printf(i8* getelementptr inbounds ([2 x i8], [2

```

```

    x i8]* @.str4, i64 0, i64 0))
23   br label %9
24
25   9:   ; i1赋初值                                ; preds = %0
26       store i32 0, i32* %2, align 4              ; i1 = 0
27       br label %10
28
29   10:  ; for_1循环判断                            ; preds = %9, %18
30       %11 = load i32, i32* %2, align 4           ; 读取i1
31       %12 = icmp slt i32 %11, 10                 ; 判断i1是否小于10
32       br i1 %12, label %13, label %21
33
34   13:  ; for_1循环体                              ; preds = %10
35       %14 = load i32, i32* %2, align 4           ; 读取i1
36       %15 = sext i32 %14 to i64                 ; 将i1扩展为i64类型
37       %16 = getelementptr inbounds [10 x float], [10 x float]* %1, i64 0, i64
           %15; 计算arr[i1]的地址
38       ; 调用 scanf , 输入arr[i1]
39       %17 = call i32 (i8*, ...) @scanf(i8* noundef getelementptr inbounds ([3 x
           i8], [3 x i8]* @.str1, i64 0, i64 0), float* noundef %16)
40       br label %18
41
42   18:  ; i1++                                      ; preds = %13
43       %19 = load i32, i32* %2, align 4           ; 读取i1
44       %20 = add i32 %19, 1                       ; i1 = i1 + 1
45       store i32 %20, i32* %2, align 4           ; 更新i1
46       br label %10
47
48   21:  ; i2赋初值                                ; preds = %10
49       store i32 0, i32* %3, align 4              ; i2 = 0
50       br label %22
51
52   22:  ; for_2外层循环判断                        ; preds = %21, %55
53       %23 = load i32, i32* %3, align 4           ; 读取i2
54       %24 = icmp slt i32 %23, 9                 ; 判断i2是否小于9
55       br i1 %24, label %25, label %58
56
57   25:  ; j赋初值                                  ; preds = %22
58       store i32 0, i32* %4, align 4              ; j = 0
59       br label %26
60
61   26:  ; for_2内层循环判断                        ; preds = %25, %52
62       %27 = load i32, i32* %4, align 4           ; 读取j
63       %28 = load i32, i32* %3, align 4           ; 读取i2
64       %29 = sub i32 9, %28                      ; 计算9 - i
65       %30 = icmp slt i32 %27, %29                ; 判断j是否小于9 - i
66       br i1 %30, label %31, label %55
67

```

```

68 31: ; 比较 arr[j+1] 和 arr[j] ; preds = %26
69 %32 = load i32, i32* %4, align 4 ; 读取 j
70 %33 = add i32 %32, 1 ; j = j + 1
71 %34 = sext i32 %33 to i64 ; 将 j 扩展为 i64 类型
72 %35 = getelementptr inbounds [10 x float], [10 x float]* %1, i64 0, i64
    %34; 计算 arr[j+1] 的地址
73 %36 = load float, float* %35, align 4; 读取 arr[j+1]
74 %37 = load i32, i32* %4, align 4 ; 读取 j
75 %38 = sext i32 %37 to i64 ; 将 j 扩展为 i64 类型
76 %39 = getelementptr inbounds [10 x float], [10 x float]* %1, i64 0, i64
    %38; 计算 arr[j] 的地址
77 %40 = load float, float* %39, align 4; 读取 arr[j]
78 %41 = fcmp olt float %36, %40 ; 判断 arr[j+1] 是否小于 arr[j]
79 br i1 %41, label %42, label %52
80
81 42: ; 条件分支 ; preds = %31
82 %43 = load i32, i32* %4, align 4 ; 读取 j
83 %44 = sext i32 %43 to i64 ; 将 j 扩展为 i64 类型
84 %45 = getelementptr inbounds [10 x float], [10 x float]* %1, i64 0, i64
    %44; 计算 arr[j] 的地址
85 %46 = load float, float* %45, align 4; 读取 arr[j]
86 store float %46, float* %5, align 4 ; temp = arr[j]
87 %47 = add i32 %43, 1 ; j = j + 1
88 %48 = sext i32 %47 to i64 ; 将 j 扩展为 i64 类型
89 %49 = getelementptr inbounds [10 x float], [10 x float]* %1, i64 0, i64
    %48; 计算 arr[j+1] 的地址
90 %50 = load float, float* %49, align 4; 读取 arr[j+1]
91 store float %50, float* %45, align 4; arr[j] = arr[j + 1]
92 %51 = load float, float* %5, align 4; 读取 temp
93 store float %51, float* %49, align 4; arr[j + 1] = temp
94 br label %52
95
96 52: ; j++ ; preds = %42, %31
97 %53 = load i32, i32* %4, align 4 ; 读取 j
98 %54 = add i32 %53, 1 ; j = j + 1
99 store i32 %54, i32* %4, align 4 ; 更新 j
100 br label %26
101
102 55: ; i2++ ; preds = %26
103 %56 = load i32, i32* %3, align 4 ; 读取 i2
104 %57 = add i32 %56, 1 ; i2 = i2 + 1
105 store i32 %57, i32* %3, align 4 ; 更新 i2
106 br label %22
107
108 58: ; 调用 printf , 输出字符串"排序后的数字为: " ; preds = %20
109 %59 = call i32 @printf(i8*, ...) @printf(i8* getelementptr inbounds ([25 x i8],
    [25 x i8]* @.str2, i64 0, i64 0))
110 ; 调用 printf , 换行

```

```

111     %60 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([2 x i8],
112         [2 x i8]* @.str4, i64 0, i64 0))
113     br label %61
114 61: ; i3 赋初值                                ; preds = %58
115     store i32 0, i32* %6, align 4                ; i3 = 0
116     br label %62
117
118 62: ; for_3 循环判断                            ; preds = %61, %72
119     %63 = load i32, i32* %6, align 4             ; 读取 i3
120     %64 = icmp slt i32 %63, 10                   ; 判断 i3 是否小于 10
121     br i1 %64, label %65, label %75
122
123 65: ; for_3 循环体                                ; preds = %62
124     %66 = load i32, i32* %6, align 4             ; 读取 i3
125     %67 = sext i32 %66 to i64                   ; 将 i3 扩展为 i64 类型
126     %68 = getelementptr inbounds [10 x float], [10 x float]* %1, i64 0, i64
127         %67; 计算 arr[i3] 的地址
128     %69 = load float, float* %68, align 4; 读取 arr[i3]
129     ; 调用 printf, 输出 arr[i3]
130     %70 = fpext float %69 to double
131     %71 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
132         [4 x i8]* @.str3, i64 0, i64 0), double noundef %70)
133     br label %72
134
135 72: ; i3++                                        ; preds = %65
136     %73 = load i32, i32* %6, align 4             ; 读取 i3
137     %74 = add i32 %73, 1                         ; i3 = i3 + 1
138     store i32 %74, i32* %6, align 4              ; 更新 i3
139     br label %62
140
141 75:                                            ; preds = %62
142     ; 调用 printf, 换行
143     %76 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([2 x i8],
144         [2 x i8]* @.str4, i64 0, i64 0))
145     ret i32 0
146 }

```

在编写代码的过程中, 我发现直接调用 printf 函数输出 float 类型的浮点数输出的结果均为 0。

查阅资料得知, 在 C 语言中, printf 函数的 %f 格式化指令可以接受 float 或 double 类型的参数。因为 printf 内部处理的是 double 类型的浮点数, 根据 C 语言的默认提升规则 (default argument promotions), float 类型的参数会转换为 double 类型。这种转换确保了无论输入是 float 还是 double, printf 都能以统一的方式处理它们。

```

1 %76 = load float, float* %75, align 4
2 %77 = fpext float %76 to double
3 %78 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([4 x i8]
    ], [4 x i8]* @.str.3, i64 0, i64 0), double noundef %77)

```

修改代码后，我们将 LLVM IR (test.ll) 转换为目标文件 (test.o)，再链接目标文件生成可执行文件。在编译时使用-fPIE 选项，确保生成位置无关的代码。

```
1 llc -filetype=obj -relocation-model=pic test.ll -o test.o
2 clang++ -fPIE test.o -o test
```

运行生成的可执行文件。

## 四、 RISC-V 汇编程序

本次汇编编程部分编写了以下两个 SysY 程序对应的汇编代码，分别用于探究 RISC-V 汇编中的浮点数运算以及数组的特性。

### (一) 浮点数累加

SysY 程序代码如下：

```
1 int main(){
2
3 float sum=0.0;
4
5 for(int i=0;i<10;i++){
6     printf("Please input 第 %d 个:",i);
7     float t=getfloat();
8     sum+=t;
9 }
10 printf("The result is %f",sum);
11 }
```

功能是接受用户输入的 10 组浮点数据，将它们累加后输出结果。

对应编写的 RISC-V 汇编代码如下：

```
1 #####SysYTest2-Riscv 汇编
2
3 .option nopic
4 .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
5 .attribute unaligned_access,0
6 .attribute stack_align,16
7
8 .data
9     zero_float: .float 0.000      # 在数据段定义一个浮点数 0.0
10    sum: .float 0.000
11
12 .LC0:
13     .string "      The result is %.3f.\n"
14 .LC1:
```

```

15     .string "Please input 第 %d 个:"
16 .LC2:
17     .string "%f"
18
19
20 .text
21
22     .global main
23     .type main,@function
24 main:
25     addi sp,sp,-32
26
27 #sum作为局部变量,应在栈中开辟空间保存;但在数据段已经预定义了sum,因此不再重
    复开辟空间
28 #使用fs0和s0两个寄存器, ra寄存器保存返回地址
29     sd s0,8(sp)
30     fsd fs0,16(sp)
31     sd ra,24(sp)
32 #for(int i=0;i<10;i++)
33 #fs0保存sum累加的结果,初始化为0.0
34     la a0,sum
35     flw fs0,0(a0)
36 #s0代表i, 记录循环次数
37     li s0,0
38 #创建局部变量t,存储在栈顶,初始值为0.0
39     la a0,zero_float
40     flw fa0,0(a0)
41     fsd fa0,0(sp)
42 .L1:
43     #准备输出提示字符串
44     mv a1,s0
45     addi a1,a1,1
46     lui a0,%hi(.LC1)
47     addi a0,a0,%lo(.LC1)
48     call putf
49     #////////接收输入数据
50     call getfloat
51     #////////输入结果在fa0
52     #先存到局部变量t中
53     fsd fa0,0(sp)    #加载t
54     #再累加到fs0寄存器,并将值保存到sum中
55     la a0,sum
56     flw fs0,0(a0)    #加载原sum
57     fadd.s fs0,fs0,fa0    #sum=sum+t
58     fsd fs0,0(a0)    #存回sum
59     #////////i++
60     addi s0,s0,1
61     #////////循环10次

```

```
62     li a1,10
63     blt s0,a1,.L1
64     #####调用print函数输出结果
65     la a0,sum
66     flw fs0,0(a0) #加载sum到fs0
67     fmv.d fa5,fs0
68     fcvtd.s fa5,fa5 #扩展为双精度
69     fmv.x.d a1,fa5 #浮点寄存器到整型寄存器
70     la a0,.LC0
71     call putf
72     #####恢复寄存器状态并返回
73     ld ra,24(sp)
74     flw fs0,16(sp)
75     ld s0,8(sp)
76
77     addi sp,sp,32
78     jr ra
79     .size main,.-main
```

由于程序中调用了 SysY 相关库函数,因此在形成可执行文件时需要添加相关参数以链接上 SysY 相关库, 命令如下:

```
1 riscv64-unknown-linux-gnu-gcc -march=rv64gc -static -o sysytest2
   SysYTest2.sy ../SysY/lib/libsysy_rv.a -g
```

部分参数说明:

1. -march=rv64gc: 指定目标架构位 RISC-V64
2. -static: 指定静态链接
3. ../SysY/lib/libsysy\_rv.a: 链接时要用到的静态库文件

执行以下命令在 qemu 模拟器上执行程序:

```
1 qemu-riscv64 test2
```

程序执行结果如下:



```

root@ZZZZJYWF:/home/Compile# qemu-riscv64 test2
Please input 第 1 个:1.1
Please input 第 2 个:1.2
Please input 第 3 个:1.3
Please input 第 4 个:1.4
Please input 第 5 个:1.5
Please input 第 6 个:1.6
Please input 第 7 个:1.7
Please input 第 8 个:1.81
Please input 第 9 个:1.92
Please input 第 10 个:1.2
The result is 14.730.
TOTAL: 0H-0M-0S-0us
root@ZZZZJYWF:/home/Compile#

```

代码分析部分:

- 在数据段我们定义了两个浮点型数据 `zero_float` 和 `sum`，其中 `zero_float` 用于初始化计算中用到的局部变量，而 `sum` 则被用于记录累加的结果。
- 根据不同的任务阶段我简单的将代码段分为以下几个部分：
  - 保存寄存器状态并初始化变量：在 `main` 函数中，使用到了 `s0` 和 `fs0` 寄存器。为了保证这两个寄存器的值在函数调用前后不发生改变，需要先将这两个寄存器中的初始值压入栈中，以便最后进行恢复（当然也会用到 `a0`，`a1` 等寄存器，但查阅资料发现根据 RISC-V 的调用约定，不要求函数调用前后寄存器的值不发生改变）。同时还要保存 `ra` 寄存器的值，该寄存器中存放返回地址，是我们函数调用结束后要跳转到的地址。  
在正式开始循环前，需要将局部变量 `t`（接收用户输入数据的变量）在内存中创建出来。作为局部变量，我选择将其创建在 `main` 函数的栈帧当中，也就是栈顶所预留的空间。这时候就要用到我们在数据段定义的 `zero_float` 来初始化该内存空间。
  - 循环体部分：
    - 输出提示字符串：根据 SysY 运行时库中说明，我们调用 `putf` 函数来输出提示字符串，其 SysY 语句为 `putf("Please input 第%d 个:",i)`，查阅运行时库文档得知第一个参数默认存放在 `a0` 寄存器，第二个参数默认存放在 `a1` 寄存器
    - 获取用户输入：为了获取用户输入的浮点数数据，查阅 SysY 运行时库文档得出要调用 `getfloat` 函数。由于此时输入的数据为浮点数，则会默认将用户的输入数据存放到 `fa0` 寄存器。按照编写的 SysY 程序的逻辑应该将 `fa0` 寄存器的值存放到局部变量 `t` 中，再进行接下来的累加；但从实现的结果来说，可以省去这一步，直接使用 `fa0` 寄存器进行接下来的累加。
    - 进行累加：`fs0` 寄存器被用于存放累加的结果，在这一步我们先将 `sum` 的值从内存中取出来存放到 `fs0` 寄存器当中，再与刚才获取的用户输入进行累加，并将累加的结果存放回变量 `sum` 的地址
  - 输出最终结果：输出最终结果对应的 SysY 语句为 `putf("The result is %f",sum)`，按照刚才的逻辑我们需要将参数分别存放在 `a0` 和 `a1` 寄存器当中，但此时就会出现问

sum 为浮点数，而 a1 寄存器被设定为存放整型数据。为了实现跨类型转换，要用到 riscv 中特殊的数据移动指令 fmv.x.d，用于将浮点数寄存器的值移动到整型寄存器当中

(d) 恢复寄存器状态并跳转

## (二) 冒泡排序

此程序目的是探究 RISC-V 汇编中的数组特性。程序接收用户输入的十组数据并采用冒泡排序的方式将这十组数据按照从小到大的顺序重新排序，最后统一输出。

SysY 程序源代码如下：

```

1  ///冒泡排序程序,十个数
2  int main(){
3
4      int arr[10];
5      for(int i=0;i<10;i++){
6          printf("Please input %dth:",i);
7          arr[i]=getint();
8      }
9
10     for(int i=0;i<9;i++)
11         for(int j=0;j<9-i;j++)
12             {
13                 int temp=arr[j];
14                 if(arr[j]>arr[j+1]){
15
16                     arr[j]=arr[j+1];
17                     arr[j+1]=temp;
18                 }
19             }
20
21     for(int i=0;i<10;i++){
22         printf("arr[%d]: %d.\n",i,arr[i]);
23     }
24
25 }
```

编写的对应 RISC-V 汇编代码如下：

```

1  #####SysYTest3-Riscv 汇编
2
3      .option nopic
4      .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0"
5      .attribute unaligned_access,0
6      .attribute stack_align,16
7
8      .text
9
10     .LC0:
```

```
11     .string "arr%d : %d.\n"
12
13 .LC2:
14     .string "Please input 第 %d 个:"
15
16 .LC1:
17     .string "%d "
18
19     .global main
20     .type main,@function
21 main:
22     addi sp,sp,-104
23 #sp指向arr[0]
24 #保存寄存器状态
25     sd s0,80(sp)
26     sd s1,88(sp)
27     sd ra,96(sp)
28
29 #进行数据的输入
30 #s1代表i, 记录循环次数
31     li s1,0
32 .L1:
33 #输出提示字符串
34     mv a1,s1
35     li a3,8
36     div a1,a1,a3
37     addi a1,a1,1
38     lui a0,%hi(.LC2)
39     addi a0,a0,%lo(.LC2)
40     call putf
41 #进行数据的输入
42
43 #初始化这一部分栈中对应的数据
44 #s0记录输入的数据要存放的位置
45     li a0,0
46     add s0,sp,s1
47     sd a0,0(s0)
48 #s0记录输入的数据要存放的位置
49     call getint
50     sd a0,0(s0)
51     addi s1,s1,8
52     li a2,80
53     blt s1,a2,.L1
54
55 #####冒泡排序部分
56 #s0记录外层循环, s1记录内层循环
57     li s0,0
58 .L2:
```

```

59     li s1,0
60     .L3:
61     #a0记录arr[j],a1记录arr[j+1]
62     mv a2,s1
63     li a4,8
64     mul a2,a2,a4
65     add a3,sp,a2
66     ld a0,0(a3) #a0=[8*j]
67     addi a3,a3,8
68     ld a1,0(a3) #a1=[8*(j+1)]
69     #a3指向【8*(j+1)】,a4为8,a2为8*j
70     #接下来让a4保存【8*j】即temp
71     mv a4,a0
72     #if 语句部分
73     bgt a0,a1,.L4
74     j .L5
75     .L4:#表示要进行交换
76
77     addi a3,a3,-8 #a3指向【8*j】
78     sd a1,0(a3)
79     addi a3,a3,8 #a3指向【8*(j+1)】
80     sd a4,0(a3)
81
82     .L5:
83     addi s1,s1,1
84     li a2,9
85     sub a3,a2,s0
86     blt s1,a3,.L3
87
88     addi s0,s0,1
89     li a2,9
90     blt s0,a2,.L2
91
92     #####逐个输出数组的值
93     li s1,0
94     .L6:
95     add s0,sp,s1
96     ld a1,0(s0)
97     lui a0,%hi(.LC1)
98     addi a0,a0,%lo(.LC1)
99     call putf
100    addi s1,s1,8
101    li a2,80
102    blt s1,a2,.L6
103
104    #恢复寄存器状态
105    ld ra,96(sp)
106    ld s1,88(sp)

```

```
107     ld s0,80(sp)
108     addi sp,sp,104
109     jr ra
110     .size main,.-main
```

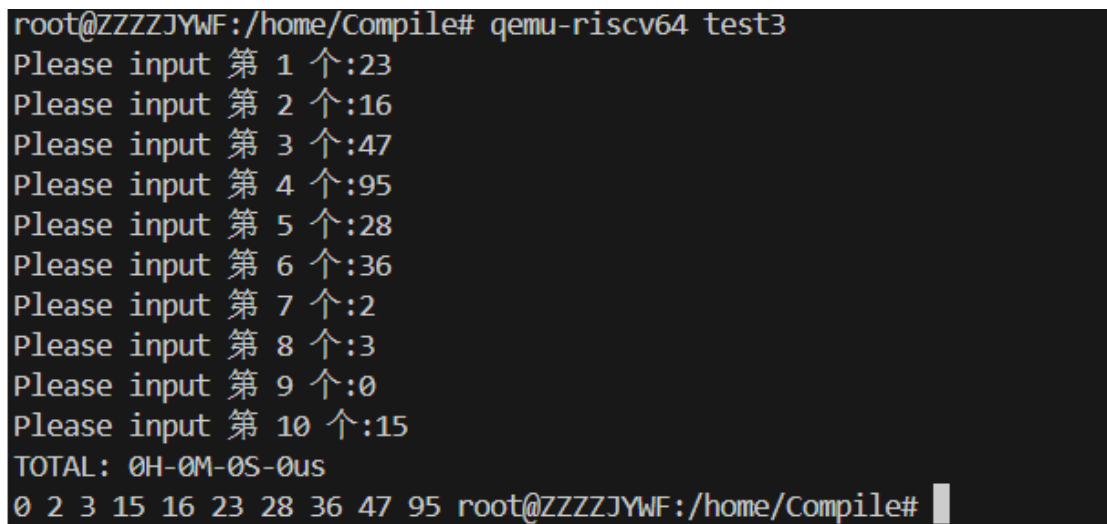
同样，由于程序中调用了 SysY 相关函数，因此在形成可执行文件时需要添加相关参数链接上 SysY 对应的库，命令如下 (参数含义不再重复解释)：

```
1 riscv64-unknown-linux-gnu-gcc -march=rv64gc -static -o sysytest3
    SysYTest3.sy ../SysY/lib/libsysy_rv.a -g
```

执行以下命令来执行程序：

```
1 qemu-riscv64 test3
```

程序执行效果如下：



```
root@ZZZZJYWF:/home/Compile# qemu-riscv64 test3
Please input 第 1 个:23
Please input 第 2 个:16
Please input 第 3 个:47
Please input 第 4 个:95
Please input 第 5 个:28
Please input 第 6 个:36
Please input 第 7 个:2
Please input 第 8 个:3
Please input 第 9 个:0
Please input 第 10 个:15
TOTAL: 0H-0M-0S-0us
0 2 3 15 16 23 28 36 47 95 root@ZZZZJYWF:/home/Compile#
```

1. 在数据段分别定义了三个字符串.L0,.L1,.L2，用于输出提示信息及最后结果。
2. 代码段根据所完成任务可以分为以下几个阶段 (与程序一相似部分不再赘述):
  - (a) 保存寄存器状态
  - (b) 获取用户输入
    - i. 输出提示字符串
    - ii. 获取用户输入：根据 SysY 源程序，arr 数组作为 main 函数当中的局部变量，我们将其存放在 main 函数的栈帧当中 (一块连续的内存区域)
  - (c) 冒泡排序：
    - i. 冒泡排序当中用到了内外两层循环，我们分别使用 s0 和 s1 来记录外层和内层的循环次数
    - ii. RISC-V-64 中通用寄存器为 64 位宽，可装载 8 个字节，为了在每次循环时更加直接地访问到该次循环所对应的位置且便于数据装载过程，我们在栈中开辟空间时选择为数组中的每一个变量开辟 8 字节大小的空间
    - iii. 在交换部分，我们使用到了 a4 寄存器来保存 a[j] 即 SysY 源程序中临时变量 temp 的值

- (d) 输出排序结果
- (e) 恢复寄存器状态

### (三) 对比分析总结

在这一部分, 我们使用成熟的系统编译器将两个 SysY 源程序编译为 RISC-V 汇编程序, 与我们自己编写的汇编程序进行对比及分析。(由于所编写的 SysY 源程序中用到了 `putf`、`getfloat` 等 SysY 库函数, 我们在将其转换为汇编程序前先将这些函数替换为 `c++` 中相同作用的函数, 待转换为可执行程序后再通过反汇编的方式得到 RISC-V 汇编程序) (也尝试过使用参考文档中指定的仓库中的 `SysYc` 来对 SysY 程序进行编译, 但最终未能成功)

两个程序最终生成的反汇编代码可见:

`Program1.s`

`program2.s`

通过分析对比以上两个程序对应的反汇编代码和自己编写的代码, 得出了以下结论:

1. 数组在 RISC-V 中的实现形式为栈中的一块连续的内存区域。(当然也可以将该数组定义为全局变量, 此时的差别在于其不存在于栈中, 但仍为一段连续的内存空间)
2. 在 RISC-V64 汇编中, 通用寄存器为 64 位宽, 而 `int`、`float` 这样的数据大小为 4 个字节, 因此在将这样的数据加载到通用寄存器时, 首先将数据加载到寄存器的低 32 位, 再对其进行符号扩展操作 (`setxt.w` 以及 `fcvt.d.s`) 得到 64 位数据。
3. 在 RISC-V 中, 浮点数与整型数据有较大的差异, 针对于浮点数的一系列操作, RISC-V 也给出了相应的支持, 例如针对于浮点数的浮点数寄存器以及浮点数特有的数据移动及加载指令等。

心得体会:

1. 反汇编得到的代码中所有的局部变量内存空间都是在栈中, 而自己编写的代码部分则在 `main` 函数前预定义了部分变量 (对比后认为这样的定义对应于 C 语言中的全局变量)
2. 自己编写的汇编代码中, 对于整型和浮点型数据 (大小为 4 字节), 一概开辟大小为 8 字节的栈空间 (寄存器为 64 位宽), 而生成的汇编代码中, 对于空间的使用更加地精确, 根据数据的大小开辟相应大小的空间。
3. 或许因为是反汇编得到的代码的原因, 反汇编对应的汇编代码中包含了 `printf` 和 `scanf` 函数的具体实现, 这也导致了在调用这些函数时的差异, 但最终都是通过调用链接库中的函数来实现的。(自己编写的代码中需要 `call` 调用函数; 而反汇编代码则直接跳转到函数实现处执行即可)