



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

编译系统原理实验报告

预备工作 - 了解编译器

谭易 2312260

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 9 月 29 日

本实验项目本人负责部分（前两部分）同步保存在：[GitHub 仓库](https://github.com/zgyfjch/Compiler-Principle-Learning/tree/main/lab0)

(<https://github.com/zgyfjch/Compiler-Principle-Learning/tree/main/lab0>)

目录

一、探究语言处理系统工作过程	1
(一) 预处理器 (C PreProcessor - cpp)	1
(二) 编译器 (cc1/cc1plus)	1
1. 前端 (Front-end)	2
2. 中端 (Middle-end)	2
3. 后端 (Back-end)	3
4. 其他重要功能 (在 cc1 中)	3
(三) 汇编器 (as)	4
(四) 链接器 (ld)	4
1. 符号解析 (Symbol resolution)	4
2. 重定位 (Relocation)	4
3. 处理静态库与共享库	5
4. GOT/PLT 与运行时绑定	5
5. 可重定位与位置无关代码 (PIC / PIE)	5
6. 链接脚本与控制	5
7. 运行时启动 (CRT, Entry Point)	5
8. 常见错误/警告原因	5
(五) 命令示例	6
二、LLVM IR 中间语言实践	6
(一) 实验环境	6
(二) 实验程序设计	6
1. 原始 SysY 源程序	6
(三) LLVM IR 实现	7
(四) 编译、链接与运行	8
1. 步骤一：将 LLVM IR 编译为 RISC-V 目标文件	8
2. 步骤二：链接目标文件与 SysY 运行库	9
3. 步骤三：在 QEMU 中运行程序	9
三、RISC-V 汇编语言实践	9
(一) 汇编源程序	9
1. 代码结构解析	10
(二) 编译、链接与测试	11
1. 步骤一：编译汇编代码	11
2. 步骤二：链接生成可执行文件	11
3. 步骤三：模拟器运行与结果验证	12

一、 探究语言处理系统工作过程

本部分以 GCC (GNU Compiler Collection) 工具链为研究对象, 通过一个简单的 C 语言程序为例, 深入探究从源代码到可执行文件的完整处理流程。该流程主要分为四个阶段: 预处理、编译、汇编和链接。

(一) 预处理器 (C PreProcessor - cpp)

cpp 是 C 语言的预处理器, 它作为编译过程的第一步, 对源代码文件进行文本层面的处理。它输出的是一个经过完全展开的、不含预处理指令的中间源文件 (通常以 .i 为扩展名, 或直接输出到标准输出流)。其主要任务包括:

- **头文件包含展开 (#include):** 将 #include 指令指定的头文件内容原封不动地插入到指令所在位置。预处理器会根据 <...> 和 "... " 的不同, 按照预设的搜索路径顺序 (如系统头文件目录、用户通过 -I 指定的目录等) 查找并读取头文件。
- **宏定义与宏展开 (#define):** 处理对象宏 (object-like macros) 和函数宏 (function-like macros), 包括带参数的宏和可变参数宏 (...)。它支持 ## (token-concatenation) 和 # (stringizing) 操作符。在展开时, 需要特别注意递归展开的规则和宏的重定义问题。
- **条件编译 (#if, #ifdef, #ifndef, #elif, #else, #endif):** 根据常量表达式的求值结果, 选择性地保留或删除代码块。这一特性常用于实现平台相关的代码、控制调试信息的输出或启用/禁用特定功能。
- **内建宏注入:** 在代码中自动插入一系列预定义的宏, 例如 __FILE__ (当前文件名)、__LINE__ (当前行号)、__DATE__ (编译日期)、__TIME__ (编译时间)、__STDC__ 等标准宏, 以及编译器特定的扩展宏 (如 GCC 的 __GNUC__)。
- **行控制与 #line 指示:** 生成 #line 指令, 用于重置编译器内部记录的文件名和行号。这确保了后续编译阶段在报告错误或警告时, 能够准确地指向原始源代码的位置, 而不是预处理后的中间文件。
- **删除注释与行拼接:** 移除所有 /* ... */ 块注释和 // ... 行注释。同时, 处理行尾的反斜杠, 将其与下一行物理地拼接成一个逻辑行。
- **处理 #pragma 指示 (部分):** 预处理器会识别一些 #pragma 指示并采取相应行动, 或者将它们原样传递给编译器进行后续处理 (例如, #pragma GCC diagnostic 用于控制编译器的诊断信息)。

示例命令:

```
1 # 将 factorial.c 文件进行预处理, 并将输出保存到 factorial.i
2 gcc -E factorial.c -o factorial.i
```

(二) 编译器 (cc1/cc1plus)

在 GCC 工具链中, 狭义的编译阶段由独立的后端程序如 cc1 (用于 C) 和 cc1plus (用于 C++) 实现。它接收预处理后的 .i 文件, 并负责将其转换为目标平台的汇编代码 (.s 文件)。这个复杂的过程通常分为前端、中端和后端三个部分。

1. 前端 (Front-end)

前端负责词法、语法和语义分析，将源代码从文本形式转化为一种抽象的、适合进一步处理的中间表示 (Intermediate Representation, IR)。

1. **词法分析 (Lexical analysis)**: 将预处理后的字符流分解成一系列有意义的单元，称为“token” (例如，关键字、标识符、字面量、运算符、分隔符等)，并保留其源文件位置信息用于错误提示。
2. **语法分析 (Parsing)**: 根据 C 语言的语法规则，将 token 流组织成一棵抽象语法树 (Abstract Syntax Tree, AST)。此阶段会检测出语法层面的错误，如不合法的表达式、缺少分号等。
3. **语义分析 (Semantic analysis)**: 对 AST 进行检查，确保其在语义上是合法的。主要工作包括：
 - 建立并维护符号表 (Symbol Table)，记录变量、函数、类型等信息。
 - 进行类型检查，确保运算符的操作数类型匹配、函数调用的参数类型与数量正确。
 - 处理作用域和存储类别 (auto, static, extern)。
 - 在早期阶段进行常量求值 (常量折叠)。
 - 处理属性、内联、可见性等语言特性。
4. **生成初级中间表示**: GCC 将 AST 转换为其内部的高层 IR (称为 GENERIC)，这种表示形式与具体语言的语法细节解耦，便于进行统一的高层优化和跨语言处理。

2. 中端 (Middle-end)

中端是编译器的核心优化部分，它对高层 IR 进行一系列的转换和优化，这些优化绝大多数是与目标机器无关的。

1. **从 GENERIC 到 GIMPLE**: 将复杂的 GENERIC 树状结构降低为一种更简单的、类似三地址码的 GIMPLE 形式。在 GIMPLE 中，复杂的表达式被分解为一系列简单的“赋值/调用/分支”语句，这为后续的数据流分析和优化提供了极大便利。
2. **SSA (Static Single Assignment) 构造**: 将 GIMPLE 代码转换为 SSA 形式。在 SSA 形式中，每个变量只被赋值一次。对于控制流合并的节点，会引入 ϕ (phi) 函数来选择正确的变量版本。SSA 极大地简化了许多优化算法的实现。
3. **机器无关优化 (GIMPLE 层)**: 在 GIMPLE-SSA 形式上应用大量的优化过程 (passes)，常见的包括：
 - 常量传播 / 常量折叠
 - 死代码删除 (DCE)
 - 拷贝消除、拷贝传播
 - 公共子表达式消除 (CSE)
 - 循环不变量外提 (LICM)
 - 循环展开、循环合并、循环分裂
 - 函数内联 (inlining) 与尾递归消除

- 基于分析的内联策略与成本估算
- 过程间分析 (Inter-Procedural Analysis, IPA), 例如跨函数常量传播、跨模块内联 (若启用 LTO)

4. **GIMPLE 到 RTL**: 将优化后的 GIMPLE 降低到更接近机器指令的 RTL (Register Transfer Language)。RTL 是一种描述寄存器和内存之间操作的低级 IR。

3. 后端 (Back-end)

后端负责将 RTL 转换为目标体系结构的汇编代码, 并执行机器相关的优化。

1. **指令选择**: 将 RTL 操作映射为目标机器的具体指令, 需要考虑寻址方式、立即数范围等硬件限制。
2. **寄存器分配**: 决定哪些变量或临时值应存放在 CPU 寄存器中, 哪些因寄存器不足而需要“溢出” (spill) 到内存栈上。常用算法包括图着色 (graph coloring) 和基于生命周期的分配算法。
3. **指令调度**: 根据目标 CPU 的流水线特性, 重排指令序列以优化性能, 例如减少流水线停顿、提高指令级并行度、避免数据冒险 (hazards) 等。
4. **平台相关优化**: 利用特定平台的指令或特性进行优化, 如 ARM/x86/RISC-V 特有的寻址模式、条件执行指令、以及处理历史架构中的延迟槽等。
5. **生成汇编代码**: 最后, 将优化后的指令序列输出为文本形式的汇编文件 (.s), 其中包含了指令、伪指令 (如 .text, .data)、标签以及调试信息指示。

4. 其他重要功能 (在 cc1 中)

- **编译器优化级别**: 通过 -O0, -O1, -O2, -O3, -Os, -Ofast 等选项, 可以控制启用哪些优化 passes 及其具体策略。
- **链接时优化 (LTO)**: 将部分优化推迟到链接阶段, 从而实现跨编译单元 (模块) 的内联和分析。
- **剖析引导优化 (PGO)**: 基于程序上一次运行收集的剖析数据 (profile data), 进行更精准的分支预测和代码布局优化。
- **警告/错误处理**: 通过 -Wall, -Wextra, -Werror 等选项控制编译器诊断信息的详细程度和严重性。
- **生成调试信息**: 使用 -g 选项时, 编译器会生成 DWARF 等格式的调试信息, 并将其嵌入到汇编和最终的目标文件中。

常用选项查看中间文件:

- -save-temps: 执行完整的编译链接流程, 但将所有中间文件 (.i, .s, .o) 都保留下来, 便于调试。
- -fdump-tree-all, -fdump-rtl-all: 输出编译器在各个优化阶段的 GIMPLE 和 RTL 中间表示, 用于深入分析编译器的优化行为。

(三) 汇编器 (as)

汇编器 (assembler) 负责将汇编代码 (.s) 翻译成机器语言, 并生成可重定位的目标文件 (.o)。

- **解析汇编指令与伪操作**: 读取汇编源文件, 识别其中的指令和伪操作 (如定义数据、对齐等)。
- **指令翻译 (assembly → machine code)**: 将每一条汇编指令 (如 mov, add) 翻译成其对应的二进制机器码 (操作码 + 编码后的寄存器/立即数)。
- **生成目标文件结构**: 创建符合特定格式 (如 Linux 下的 ELF, Windows 下的 PE, macOS 下的 Mach-O) 的目标文件。
- **符号表 (Symbol Table)**: 创建一个符号表, 记录本文件定义的所有全局符号 (函数名、全局变量) 以及引用的外部符号 (未在本文件中定义的符号)。
- **重定位条目 (Relocation)**: 对于那些地址在汇编时无法确定的引用 (如调用外部函数、访问外部全局变量), 汇编器会生成重定位条目。这些条目记录了需要由链接器在后期进行地址修正的“待办事项”的位置和类型。
- **调试信息的保留**: 将由编译器生成的 DWARF 等调试指示, 从汇编代码中提取并写入目标文件的相应节 (section), 供链接器和调试器使用。

输出: .o 文件 (以 ELF 为例), 包含: ELF 头、程序头表 (虽然在可重定位文件中通常为空白)、节头表 (Section headers) 以及各种节, 如 .text、.data、.rodata、.bss、.symtab、.rela.* 等。

(四) 链接器 (ld)

链接器 (linker) 的主要工作是将一个或多个目标文件 (.o) 以及库文件 (.a, .so) 合并, 生成最终的可执行文件或共享库。这是一个复杂的过程, 核心工作包括:

1. 符号解析 (Symbol resolution)

- 将每个目标文件中引用的未定义符号 (例如在 a.o 中调用了 'printf'), 与所有输入文件中的某个确定定义进行匹配。
- 支持强符号 (strong symbol) 和弱符号 (weak symbol) 的规则, 例如强符号会覆盖同名的弱符号, 允许多个同名的弱符号存在。
- 根据链接命令中文件的顺序和搜索路径 (-L, /etc/ld.so.conf), 在静态库或动态库中查找并解析符号。如果找不到某个符号的定义, 链接器会报告 “undefined reference” 错误。

2. 重定位 (Relocation)

- **地址空间合并与分配**: 将所有输入目标文件中的同名节 (如 .text) 合并, 并为合并后的各个节在最终的虚拟地址空间中分配唯一的运行时地址。
- **地址修正**: 根据前面分配的地址, 遍历所有重定位条目, 计算出符号的最终地址, 并用这个地址去修正代码段或数据段中之前预留的占位符。不同架构有不同的重定位类型 (如 x86_64 的 R_X86_64_PC32, RISC-V 的 R_RISCV_CALL), 链接器会按类型计算并修补指令或数据。

3. 处理静态库与共享库

- **静态链接**：对于静态库 (.a)，链接器从中提取出被引用的目标文件模块，并将其代码和数据直接合并到最终的可执行文件中。结果文件较大，但无外部依赖。
- **动态链接**：对于共享库 (.so)，链接器不会将库代码复制进来，而是在可执行文件中记录一个依赖关系 (DT_NEEDED 条目)。同时，它会生成特殊的代码和数据结构，如过程链接表 (PLT) 和全局偏移表 (GOT)，用于在程序运行时由动态链接器来完成最终的函数地址解析与间接跳转。

4. GOT/PLT 与运行时绑定

- **PLT (Procedure Linkage Table)**：用于实现惰性符号解析 (lazy binding)。对外部函数的第一次调用会通过 PLT 跳转到动态链接器来解析函数地址，后续调用则直接跳转到函数，提高了启动速度。
- **GOT (Global Offset Table)**：存放全局变量和函数的绝对地址。位置无关代码 (PIC) 通过访问 GOT 来间接引用这些全局符号。

5. 可重定位与位置无关代码 (PIC / PIE)

当编译目标是共享库 (-fPIC) 或位置无关可执行文件 (-fPIE) 时，编译器会生成使用相对地址访问内存的代码。链接器在处理这类代码时，会生成相应的相对地址重定位和 GOT/PLT 条目。

6. 链接脚本与控制

通过使用链接脚本 (ld 脚本或 --script 选项)，开发者可以精细控制各个节的布局、符号的导出、内存区域的划分、起始地址的设定等高级链接行为。也支持对节进行垃圾回收 (--gc-sections)、符号版本控制等。

7. 运行时启动 (CRT, Entry Point)

- 链接器会默认链接 C 运行时 (CRT) 的启动文件 (如 crt1.o, crt1.o)，这些文件提供了操作系统加载程序所需的入口点 _start。
- _start 负责初始化 C 库环境 (例如调用 __libc_start_main)，然后调用我们编写的主函数 main()。当 main() 返回后，它会调用退出程序的相关函数。
- 对于动态链接的可执行文件，ELF 文件头中会包含一个 .interp 节，指向动态加载器 (如 /lib64/ld-linux-x86-64.so.2) 的路径。

8. 常见错误/警告原因

- **ABI 不匹配**：例如，链接一个使用硬浮点 (hard-float) ABI 编译的对象文件和另一个使用软浮点 (soft-float) 编译的库。链接器会拒绝链接。必须保持 -mabi、-march 等编译选项一致。
- **未定义符号**：缺少引用的库，或者链接命令中库的放置顺序错误 (通常，被依赖的库要放在依赖它的文件之后)。

- **重定位失败**: 例如, 试图将一个大的地址常量直接放入一个立即数范围受限的指令中, 而代码没有按 PIC 方式编译。

(五) 命令示例

- `gcc -E factorial.c -o factorial.i` → 只运行预处理器 (cpp)。
- `gcc -S factorial.i -o factorial.s` → 只运行编译器 (cc1), 从 `.i` 生成汇编 `.s`。
- `gcc -c factorial.s -o factorial.o` → 只运行汇编器 (as), 从 `.s` 生成目标文件 `.o`。
- `gcc factorial.o -o prog` → 运行链接器 (ld), 生成可执行文件 `prog`。
- `gcc factorial.c -save-temps -o prog` → 执行所有步骤, 并保留所有中间文件 (`.i`, `.s`, `.o`)。

可以看到预处理器的工作完成后, 所有头文件都被拷贝到同一文件, `.i` 文件的内容急剧扩张 (见项目目录)。编译完

二、 LLVM IR 中间语言实践

本部分熟悉 LLVM IR 中间语言, 并针对一个 SysY 示例程序, 手动编写等价的 LLVM IR, 并最终编译链接成可在 RISC-V 平台上运行的程序。

(一) 实验环境

项目	说明
操作系统	Ubuntu 20.04 (Docker 容器)
编译工具	Clang, LLVM, RISC-V 交叉编译工具链
RISC-V ABI	lp64d (硬浮点 ABI)
运行环境	QEMU RISC-V 模拟器
SysY 运行库	'libsysy_riscv.a'

表 1: 实验环境配置

(二) 实验程序设计

为了涵盖 SysY 的基本语言特性, 我们选择一个计算阶乘的程序作为示例。

1. 原始 SysY 源程序

Listing 1: factorial_SysY.sy

```

1 int factorial(int n) {
2     int sum = 1;
3     while (n > 1) {
4         sum = sum * n;
5         n = n - 1;
6     }

```



```

7     return sum;
8 }
9
10 int main() {
11     int x = getint();
12     int res = factorial(x);
13     putint(res);
14     putchar(10); /* for newline */
15     return 0;
16 }

```

功能说明：该程序从标准输入读取一个整数，计算其阶乘，然后将结果输出到标准输出。它涵盖了函数定义与调用、变量赋值、循环语句（‘while’）、整数运算以及对 SysY 运行库函数的调用。

（三） LLVM IR 实现

根据上述 SysY 程序，手动编写等价的 LLVM IR 代码。

Listing 2: factorial_my.ll

```

1 ; ModuleID = 'factorial_my.ll'
2 ; target triple = "riscv64-unknown-elf"
3
4 ; Declare functions from SysY runtime library
5 declare i32 @getint()
6 declare void @putint(i32)
7 declare void @putch(i32)
8
9 ; factorial function implementation
10 define i32 @factorial(i32 %n_arg) {
11     entry:
12         br label %loop_head
13
14     loop_head:
15         %n = phi i32 [ %n_arg, %entry ], [ %n_next, %loop_body ]
16         %sum = phi i32 [ 1, %entry ], [ %sum_next, %loop_body ]
17         %cond = icmp sgt i32 %n, 1
18         br i1 %cond, label %loop_body, label %loop_end
19
20     loop_body:
21         %sum_next = mul nsw i32 %sum, %n
22         %n_next = sub nsw i32 %n, 1
23         br label %loop_head
24
25     loop_end:
26         ret i32 %sum
27 }
28
29 ; main function implementation
30 define i32 @main() {

```

```

31 entry:
32     %input_val = call i32 @getint()
33     %result = call i32 @factorial(i32 %input_val)
34     call void @putint(i32 %result)
35     call void @putch(i32 10) ; ASCII for newline
36     ret i32 0
37 }

```

实现说明：代码使用了 LLVM IR 的 SSA（静态单赋值）形式。‘phi’ 指令被用来处理循环中变量 ‘n’ 和 ‘sum’ 的值合并问题。在循环头（‘loop head’），‘phi’ 根据代码的来源基本块（‘entry’ 或 ‘loop body’）选择正确的值。

（四） 编译、链接与运行

使用 LLVM Clang 和 RISC-V GCC 工具链将 LLVM IR 转换为可执行程序。

1. 步骤一：将 LLVM IR 编译为 RISC-V 目标文件

```

1 clang --target=riscv64-unknown-elf \
2     -march=rv64imafdc \
3     -mabi=lp64d \
4     -c factorial_my.ll -o factorial_rv.o

```

命令参数说明：

- ‘-march=rv64imafdc’：指定目标处理器的架构和扩展。
- ‘-mabi=lp64d’：指定应用程序二进制接口（ABI）。“lp64d” 表示使用硬浮点（hard-float）ABI。
- 指定应用程序二进制接口（ABI），之所以要显式指定，是因为在 RISC-V 和 ARM 等体系结构中，浮点处理单元（FPU）并非是必须的硬件。RISC-V 的硬件浮点功能通过 CPU 的 F/D 扩展来实现。如果编译时不明确指定，默认情况（例如仅使用 ‘-march=rv64i’）可能会采取软件模拟浮点（soft-float）的方式，这会导致生成的对象文件与要求硬件浮点（hard-float）的 SysV 库文件的 ABI 不对齐，从而造成链接错误。‘lp64d’ 确保了使用与库匹配的硬浮点 ABI。

扩展	说明
‘rv’	RISC-V 架构
‘64’	64 位基本整数寄存器宽度
‘i’	基本整数指令集 (Integer base)
‘m’	乘除法指令扩展 (Multiply/Divide)
‘a’	原子指令扩展 (Atomic)
‘f’	单精度浮点扩展 (Float, 32-bit)
‘d’	双精度浮点扩展 (Double, 64-bit)
‘c’	压缩指令扩展 (Compressed, 16-bit)

表 2: RISC-V ‘rv64imafdc’ 架构扩展说明

2. 步骤二：链接目标文件与 SysY 运行库

```
1 riscv64-unknown-elf-gcc \
2     factorial_rv.o path/to/libsysy_riscv.a \
3     -o factorial_rv
```

说明：使用 RISC-V 的交叉编译器 ‘gcc’ 将我们生成的目标文件 ‘factorial_rv.o’ 与 SysY 静态库 ‘libsysy_riscv.a’ 链接在一起，生成最终的可执行文件 ‘factorial_rv’。（请将 ‘path/to/’ 替换为库文件的实际路径）。

3. 步骤三：在 QEMU 中运行程序

```
1 qemu-riscv64 ./factorial_rv
```

在 QEMU RISC-V 模拟器中执行程序。程序会等待用户输入一个数字，按下回车后，计算并输出阶乘结果，然后换行退出。实验结果验证了手动编写的 LLVM IR 的正确性。

```
root@2e38920bc771:~/lab0# make run
clang --target=riscv64-unknown-elf \
  -march=rv64imafdc \
  -mabi=lp64d \
  -c factorial_my.ll -o factorial_rv.o
warning: overriding the module target triple with riscv64-unknown-unknown-elf [-Woverride-module]
1 warning generated.
riscv64-unknown-elf-gcc \
  factorial_rv.o ~/lib/libsysy_riscv.a \
  -o factorial_rv
qemu-riscv64 ./factorial_rv
10
3628800
TOTAL: 0H-0M-0S-0us
```

图 1: LLVM IR 程序运行结果

三、 RISC-V 汇编语言实践

本部分将展示如何为阶乘程序手动编写等价的 RISC-V 汇编代码，并详细介绍如何通过交叉编译工具链进行编译、链接，最终在 QEMU 模拟器上验证其正确性。

（一） 汇编源程序

首先，我们将整个阶乘计算程序编写在一个汇编文件 main.s 中。该程序不包含独立的 ‘factorial’ 函数，而是在 ‘main’ 函数中直接实现了迭代计算逻辑。

Listing 3: 完整的阶乘汇编程序 main.s

```
1 .text
2 .globl main
3
4 main:
5     # 1. 函数栈帧设置
6     addi sp, sp, -32      # 分配32字节栈空间
7     sd ra, 24(sp)        # 保存返回地址 (return address)
8     sd s0, 16(sp)        # 保存旧的帧指针 (frame pointer)
9     addi s0, sp, 32      # 设置新的帧指针
10
```

```

11  # 2. 输入
12  jal ra, getint      # 调用 getint 函数, 结果存入 a0
13  mv t0, a0          # 将输入的 n 保存到 t0 寄存器 (t0 = n)
14
15  # 3. 变量初始化
16  li t1, 1            # t1 作为阶乘结果 f, 初始化 f = 1
17  li t2, 2            # t2 作为循环计数器 i, 初始化 i = 2
18
19  # 如果 n <= 1, 结果恒为 1, 直接跳转到结束
20  ble t0, t1, end_loop
21
22  # 4. 计算过程 (循环)
23  loop:
24    bgt t2, t0, end_loop # 如果 i > n, 跳出循环
25
26    mul t1, t1, t2       # f = f * i
27    addi t2, t2, 1       # i = i + 1
28
29    j loop              # 继续下一次循环
30
31  # 5. 输出结果
32  end_loop:
33    mv a0, t1            # 将最终结果 f 放入 a0, 准备调用 putint
34    jal ra, putint       # 调用 putint 输出结果
35
36    li a0, 10            # 将换行符 '\n' (ASCII 10) 放入 a0
37    jal ra, putchar     # 调用 putchar 输出换行
38
39  # 6. 收尾
40  li a0, 0              # 设置返回值为 0
41
42  ld ra, 24(sp)         # 从栈中恢复返回地址
43  ld s0, 16(sp)         # 恢复旧的帧指针
44  addi sp, sp, 32       # 释放栈空间
45
46  ret                  # 返回调用者

```

1. 代码结构解析

上述汇编代码的执行流程清晰，主要分为以下几个步骤：

- **函数栈帧设置：**在函数开头，分配栈空间，并保存调用者传递的返回地址 ‘ra’ 和上一级的帧指针 ‘s0’，这是函数调用的标准入口操作（prologue）。
- **输入处理：**通过 ‘jal’ 指令调用 SysY 运行库中的 ‘getint’ 函数，从标准输入读取一个整数。返回值默认存储在 ‘a0’ 寄存器中，随后被移动到 ‘t0’ 作为 ‘n’。
- **变量初始化：**使用 ‘li’ (load immediate) 指令将立即数 ‘1’ 和 ‘2’ 分别加载到 ‘t1’ (阶乘结果) 和 ‘t2’ (循环变量) 中。同时增加了一个边界条件判断，如果输入的 ‘n’ 小于等于 1，则

直接跳转到输出环节。

- **计算过程：**这是一个典型的循环结构。‘loop’ 标签是循环的起点。‘bgt’ 指令用于判断循环是否结束。循环体内使用 ‘mul’ 和 ‘addi’ 指令执行乘法和自增操作。
- **输出结果：**计算结束后，将结果从 ‘t1’ 移回 ‘a0’，并依次调用 ‘putint’ 和 ‘putch’ 函数输出结果和换行符。
- **收尾：**在函数末尾，恢复之前保存的寄存器，释放栈空间，并将返回值 ‘0’ 放入 ‘a0’，最后通过 ‘ret’ 指令返回。这是标准的函数出口操作（epilogue）。

（二） 编译、链接与测试

1. 步骤一：编译汇编代码

使用 RISC-V 交叉编译工具链中的汇编器将 main.s 文件编译为目标文件 main.o。

Listing 4: 编译命令

```
1 riscv64-unknown-elf-gcc main.s -c -o main.o -w
```

参数说明：

- c：指示编译器只进行编译和汇编，不执行链接步骤，生成 .o 格式的目标文件。
- o main.o：指定输出文件的名称为 main.o。
- w：禁止所有警告信息的输出。在调试时可以去掉此选项以查看潜在问题。

2. 步骤二：链接生成可执行文件

将生成的目标文件与 SysY 运行库链接，生成最终的可执行文件。

Listing 5: 链接命令

```
1 riscv64-unknown-elf-gcc main.o -o main -L./lib -lsysy_riscv -static \  
2 -mcmodel=medany -Wl,--no-relax,-Ttext=0x90000000
```

参数说明：

- o main：指定输出的可执行文件名为 main。
- L./lib：添加库文件的搜索路径，这里假设库文件在当前目录下的 ‘lib’ 文件夹中。
- lsysy_riscv：链接名为 ‘libsysy_riscv.a’ 的库文件。
- static：执行静态链接，将所有需要的库代码直接嵌入到最终的可执行文件中。
- mcmodel=medany：指定代码模型为 ‘medany’，允许代码和静态数据位于 2GiB 的地址空间内，适用于中等大小的程序。
- Wl,...：将逗号分隔的参数传递给链接器 (ld)。
- no-relax：禁用链接器的地址松弛 (relaxation) 优化，这可以防止链接器替换某些 RISC-V 指令序列。
- Ttext=0x90000000：通过链接脚本选项，手动设置程序代码段 (.text) 的起始地址。

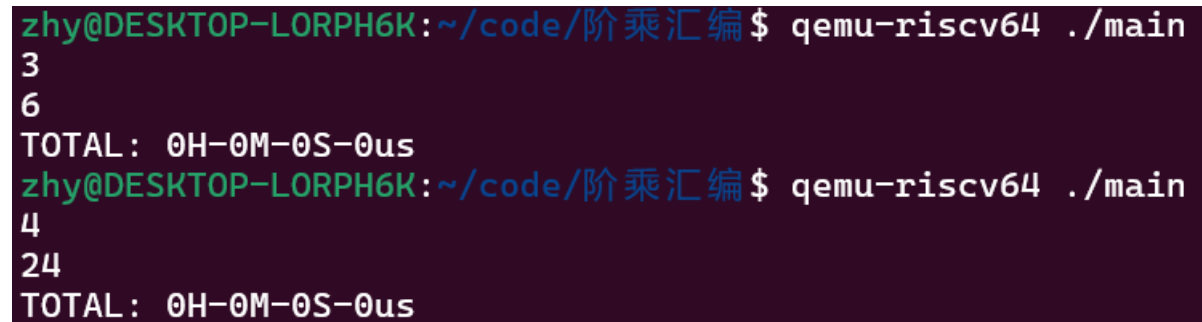
3. 步骤三：模拟器运行与结果验证

使用 QEMU 模拟器运行生成的 RISC-V 64 位可执行程序。

Listing 6: 运行命令

```
1 qemu-riscv64 ./main
```

我们分别输入 ‘3’ 和 ‘4’ 对程序进行测试，程序正确输出了阶乘结果 ‘6’ 和 ‘24’，验证了汇编代码的正确性。



```
zhy@DESKTOP-LORPH6K:~/code/阶乘汇编$ qemu-riscv64 ./main
3
6
TOTAL: 0H-0M-0S-0us
zhy@DESKTOP-LORPH6K:~/code/阶乘汇编$ qemu-riscv64 ./main
4
24
TOTAL: 0H-0M-0S-0us
```

图 2: RISC-V 汇编程序运行结果