



南開大學
Nankai University

计算机学院
编译系统原理实验报告

了解你的编译器
LLVM IR 编程 & 汇编编程

姓名：谭凯泽 殷腾骄

学号：2212204 2212202

专业：计算机科学与技术

2024 年 9 月 25 日

目录

1 摘要	2
2 实验平台	2
3 了解你的编译器—GCC	2
3.1 编译流程概述	3
3.2 预处理阶段	7
3.3 编译阶段	8
3.3.1 词法分析	8
3.3.2 语法分析 & 语义分析	9
3.3.3 中间代码生成	10
3.3.4 代码优化	14
3.3.5 代码生成	17
3.4 汇编阶段	17
3.5 链接阶段	19
4 LLVM IR 编程	21
5 SysY 语言以及 ARM/RISC-V 汇编编程	26
5.1 SysY 示例程序	26
5.2 ARM 汇编编程	27
5.3 RISC-V 汇编编程	29
6 实验总结	32

1 摘要

本本实验由谭凯泽和殷腾骄合作完成。实验报告第一部分“了解你的编译器”和第二部分“LLVM IR 编程”由谭凯泽负责，第三部分“汇编编程”由殷腾骄完成。

第一部分中，我们设计了包含头文件、宏、全局变量、函数、注释等语言特性的 C 样例程序，分析了 GCC 编译器将 C 代码转换为可执行文件的各个阶段。我们探讨了预处理器 CPP 的工作，深入 GCC 源码研究了编译过程中的词法分析、语法分析和语义分析的具体实现机制。通过调整编译参数，我们探索了 GCC 中间代码转换和生成，以及代码优化的过程。对于汇编阶段和链接阶段，我们分析了可重定向目标文件和可执行文件的关系，以及 ELF 文件和符号表在链接前后的变化。

第二部分中，我们学习了 LLVM IR 程序的结构，并使用 LLVM 编写了包含输入输出、函数调用、数组、指针等语言特性的程序，以加深对 LLVM 的理解。第三部分汇编编程中，我们对 SysV 语言、ARM 汇编和 RISC-V 汇编进行了实验，展示了在不同架构上实现简单算法过程，并设计代码实例进行说明。

2 实验平台

本次实验在 WSL2 平台上进行，其发行版为 Ubuntu-20.04。使用编译工具的类型和版本信息如表所示。

架构	gcc	clang	llvm
x86_64	9.4.0	18.1.8	18.1.8

表 1: 编译工具的类型和版本

3 了解你的编译器-GCC

我们研究的编译器是 GCC。GCC 是 GNU Compiler Collection 的缩写，用于编译 C、C++ 等语言的源代码。

我们将以一个 C 程序为例，探索一个包含了 C 语言代码、带有特殊文件后缀.c 的文本文件，是如何被著名的编译器 GCC，一步一步转化为一个可执行程序。为了更全面地探索每一个阶段编译器的工作，我们以基础样例程序“阶乘”为基础，增加了函数、头文件、全局变量、宏、递归等语言特性。我们设计的 C 程序如下所示。

C 样例程序 example.c

```
1  #include <stdio.h>
2
3  #define FACTORIAL_LIMIT 10 // 宏定义
4
5  int factorial(int n);
6
7  int global_counter = 0; // 全局变量
8
9  int main() {
10     int n, result;
```

```

11
12     printf("Please enter an integer: ");
13     scanf("%d", &n);
14
15     if (n > FACTORIAL_LIMIT) {
16         printf("Exceeds limit, maximum supported factorial is %d.\n",
17             FACTORIAL_LIMIT);
18         return 1;
19     }
20
21     result = factorial(n);
22     printf("Factorial of %d is: %d\n", n, result);
23     printf("The function was called %d times\n", global_counter);
24
25     return 0;
26 }
27
28 // 递归实现
29 int factorial(int n) {
30     global_counter++;
31     if (n <= 1) {
32         return 1;
33     } else {
34         return n * factorial(n - 1);
35     }
36 }

```

我们对基础样例程序“阶乘”中的改动有：

- 我们使用递归地形式来实现阶乘，并将其设计为一个函数，以观察编译器对函数的处理。
- 我们增加了标准输入输出头文件 `<stdio.h>` 的引入，这对于我们的程序是必要的。
- 我们增加了全局变量 `global_counter`，来追踪阶乘函数的调用次数，以观察编译器对全局变量的处理。
- 我们增加了宏定义 `FACTORIAL_LIMIT`，以观察编译器对宏定义的处理。

3.1 编译流程概述

一个 C 语言程序，需要经过以下步骤，才能生成可执行文件。以我们的 `example.c` 程序为例。

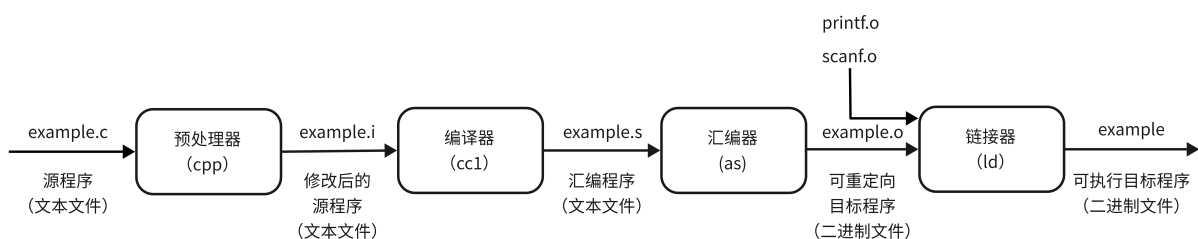


图 3.1: 编译系统 [3]

- **预处理阶段** (Preprocessing phase)：在这个阶段，C 预处理器 (C Pre-processor, cpp) 会根据以 # 开头的预处理指令对原始的 C 程序进行修改。在 example.c 程序中，存在 “#include <stdio.h>” 和 “#define FACTORIAL_LIMIT 10” 两条这样的指令。预处理器会将 #include <stdio.h> 替换为 stdio.h 头文件的内容，这个文件中包含了 printf 和 scanf 等函数的声明 #define FACTORIAL_LIMIT 10 将会把程序中的 FACTORIAL_LIMIT 宏替换成常量 10，相当于进行简单的文本替换。预处理结果会生成一个新的 C 代码文件，通常以.i 为后缀，即 example.i。
- **编译阶段** (Compilation phase)：编译器 (cc1) 会把经过预处理的.i 文件中的代码翻译为汇编代码，该汇编代码文件用.s 作为后缀，即 example.s。
- **汇编阶段** (Assembly phase)：汇编器 (as) 将汇编代码转换为机器代码，并将其打包成一个可重定位目标文件。汇编器会将汇编语言指令转换成机器语言的二进制编码，最终得到汇编好的二进制的目标文件 example.o。
- **链接阶段** (Linking phase)：链接器 (ld) 负责将多个目标文件链接在一起，生成最终的可执行文件。在 example.c 中，我们调用了 printf 和 scanf 函数，这些函数定义在标准 C 库中，它们以 printf.o 和 scanf.o 这样的目标文件存在。所以需要链接器把这些库函数的目标文件与 example.o 合并在一起，生成最终的可执行文件。

对于我们将要探索的编译器 GCC 来说，我们只需要一行命令，就可以完成以上所有步骤，得到可执行文件。

```
1 gcc example.c -o example
```

其中，gcc 表示使用 GCC 编译器，example.c 是我们希望编译的 C 源代码，-o 是一个编译选项，用于指定输出文件的名称，即下一个参数 example。执行上述命令后，就得到一个名为 example 的可执行文件。

在编译系统原理原理的课程中，我们关注的实际上是中间的编译阶段 (Compilation phase)。编译阶段可以细分为以下几个步骤：

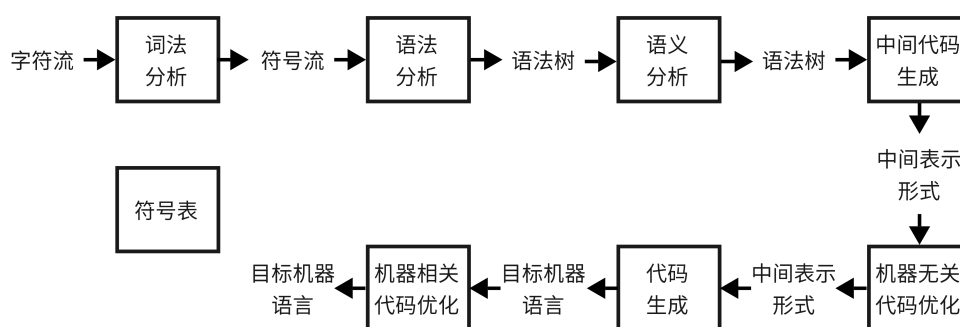


图 3.2: 一个编译器的各个步骤 [1]

- **词法分析** (Lexical Analysis)：经过预处理的源代码会作为字符流进入词法分析器，然后转换为词法单元 (token) 的序列。token 中包含了后续语法分析中所需要的信息。在这个阶段还会开始构建符号表，符号表中记录了每个符号的名称、类型等。
- **语法分析** (Syntax Analysis)：语法分析器会根据输入的词法单元 token 序列生成语法树，语法树是一种中间表示，给出了词法分析产生的词法单元流的语法结构。

- **语义分析 (Semantic Analysis)**: 语义分析器利用语法树和构建的符号表, 检查源程序是否和语言定义的语义一致。同时还会收集类型信息, 用于类型检查, 并检查每个运算符是否具有匹配的运算分量。
- **中间代码生成器**: 为了方便汇编代码的生成和代码优化, 编译器会利用经过语义分析的语法树, 生成一个明确的、低级的或类机器语言的中间表示。这种中间表示与机器无关, 从而可以方便地进行优化。
- **机器无关代码优化**: 编译器会改进中间代码, 以生成更好的目标代码。
- **代码生成**: 经过上述阶段, 编译器就可以根据中间代码生成相应的汇编代码或机器代码, 后续还可能进行机器相关的代码优化。

在进入每个阶段的探索之前, 我们使用以下命令观察 GCC 编译过程的详细信息。

```
1 gcc -v example.c -o example
```

加上编译参数 “-v” 可以输出 GCC 编译过程的详细信息, 包括每个编译阶段的调用命令、参数、库的搜索路径、头文件的搜索路径等。使用上述命令后将得到较长的输出, 以下简要说明其中较为重要的部分。

```
Target: x86_64-linux-gnu
```

这说明表示目标架构是 64 位的 x86_64, 且使用 Linux 系统, 这与我们的实验平台一致。

```
1 COLLECT_GCC_OPTIONS='-v' '-o' 'example' '-mtune=GENERIC' '-march=x86-64'
```

这里展示了当前 GCC 收集到的编译选项, 即 GCC 在执行编译时实际使用的参数。如我们指定的参数 “-v” 和 “-o”。后两个选项为 GCC 默认添加的选项。

```
1 /usr/lib/gcc/x86_64-linux-gnu/9/cc1 -quiet -v -imultiarch x86_64-linux-gnu
  example.c -quiet -dumpbase example.c -mtune=GENERIC -march=x86-64 -auxbase
  example -version -fasynchronous-unwind-tables -fstack-protector-strong
  -Wformat -Wformat-security -fstack-clash-protection -fcf-protection -o
  /tmp/ccz8BSb2.s
```

然后我们能看到 GCC 使用了编译器 cc1 将 C 代码转换成汇编代码, 并输出汇编代码到临时文件 /tmp/ccz8BSb2.s 中。

```
1 ignoring nonexistent directory "/usr/local/include/x86_64-linux-gnu"
2 ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-gnu/9/include-fixed"
3 ignoring nonexistent directory
  "/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/include"
4 #include "... " search starts here:
5 #include <...> search starts here:
6 /usr/lib/gcc/x86_64-linux-gnu/9/include
7 /usr/local/include
8 /usr/include/x86_64-linux-gnu
9 /usr/include
10 End of search list.
```

接着会列出搜索头文件的目录，ignoring 表示 GCC 在搜索头文件时，忽略了某些不存在的目录。其中“#include <...> search starts here:”表示当我们在代码中使用“#include <filename>”引入标准库头文件时，GCC 搜索头文件的顺序。如果 GCC 在以上顺序搜索中没有找到我们引入的头文件，就会报错。

```
1 as -v --64 -o /tmp/cc1kBjNz.o /tmp/ccz8BSb2.s
2 GNU assembler version 2.34 (x86_64-linux-gnu) using BFD version (GNU Binutils for
   Ubuntu) 2.34
```

编译完成后，GCC 调用汇编器 as 来将生成的汇编代码转换成机器代码，得到.o 目标文件。可以看到，汇编器 as 的输入正是编译阶段得到的，用于保存汇编代码的临时文件 ccz8BSb2.s。

```
1 LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/9:/usr/lib/gcc/x86_64-linux-gnu/9/
2 ../../../../x86_64-linux-gnu:/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib:/lib
3 /x86_64-linux-gnu:/lib/./lib:/usr/lib/x86_64-linux-gnu:/usr/lib/./lib/:
4 /usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib:/usr/lib/
```

这里则是 GCC 在链接阶段查找库文件的路径，GCC 会在这些路径中寻找静态库 (.a 文件) 或动态库 (.so 文件)。

```
1 /usr/lib/gcc/x86_64-linux-gnu/9/collect2 -plugin
   /usr/lib/gcc/x86_64-linux-gnu/9/liblto_plugin.so
   -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
   -plugin-opt=-fresolution=/tmp/cciyJmDY.res -plugin-opt=-pass-through=-lgcc
   -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc
   -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id
   --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker
   /lib64/ld-linux-x86-64.so.2 -pie -z now -z relro -o example
   /usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/Scrt1.o
   /usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/crti.o
   /usr/lib/gcc/x86_64-linux-gnu/9/crtbeginS.o -L/usr/lib/gcc/x86_64-linux-gnu/9
   -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu
   -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib -L/lib/x86_64-linux-gnu
   -L/lib/./lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/./lib
   -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../tmp/cc1kBjNz.o -lgcc --push-state
   --as-needed -lgcc_s --pop-state -lc -lgcc --push-state --as-needed -lgcc_s
   --pop-state /usr/lib/gcc/x86_64-linux-gnu/9/crtendS.o
   /usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/crtn.o
```

这是 GCC 调用链接器 collect2 的完整命令。collect2 是 GCC 的链接器驱动程序，它负责调用系统的实际链接器。如果我们使用以下命令

```
1 strace -f gcc -o example example.c 2>&1 | grep execve
```

就可以看到所有由 GCC 通过 execve 系统调用执行的命令，其中显示 GCC 在调用 collect2 后调用了 ld，而 ld 即为系统的实际链接器。这说明 collect2 并非真正的链接器，而是驱动程序，负责控制整个链接过程，真正的链接器是 ld。

还可以使用编译参数“-fdump-tree-all”，这会生成编译过程中的某些中间数据文件。


```
1 gcc -fdump-tree-all example.c
```

使用以上命令后, 我们得到 example.c.004t.original、example.c.005t.gimple 等类似命名的文件。这些文件中包含了 GCC 编译不同阶段的内部数据结构和优化信息, 我们后续会深入观察其中的内容。

对 GCC 编译器的编译流程进行概览后, 我们接下来探索每一阶段编译器对 example.c 文件的具体操作。

3.2 预处理阶段

我们可以使用“-E”参数, 让 GCC 只进行预处理。

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

我们让 GCC 只对 example.c 进行预处理, 并将处理后的结果保存到 example.i 中。打开 example.i 文件后, 我们发现文件增加了许多内容。我们知道预处理阶段, 编译器会将使用“#include”引入的头文件内容复制到 C 源文件中, 并替换定义的宏。在 example.i 文件中, 我们看到

example.i 中的内容

```
1 // 上面是各种别名、函数的声明, 这里暂不列出, 后续讨论
2 // 这里的注释为报告撰写时加入以便说明, 原有的注释已被编译器删去
3 int global_counter = 0;
4
5 // 略去部分内容
6     if (n > 10) {
7         printf("Exceeds limit, maximum supported factorial is %d.\n", 10);
8         return 1;
9     }
10
11 result = factorial(n); // Call function to compute factorial
12 printf("Factorial of %d is: %d\n", n, result);
13 // 略去部分内容
14 // 下面阶乘函数的定义略去
```

在 example.c 中, 我们使用宏定义 FACTORIAL_LIMIT 来限制用户的输入, 并在 main 函数中的 if 条件语句中使用。在 example.i 中, FACTORIAL_LIMIT 已被替换为 10。这说明, 预处理阶段编译器确实对宏定义进行了文本替换。同时也能看到, example.c 中的注释都被 GCC 删去。

关于 example.i 中长篇复杂的各种函数声明, 我们可以在路径“/usr/include”中找到“stdio.h”文件, 对比 /usr/include/stdio.h 和 example.i 的内容。对比后发现, 两个文件中都出现了我们希望引入的“printf”和“scanf”的函数声明, 还有其他我们虽没有使用, 但依然出现的函数声明, 如“fprintf”等。

我们发现 GCC 编译器并非简单地将“#include <stdio.h>”简单地替换为 /usr/include/stdio.h 的内容。根据观察, stdio.h 包含许多 #define 定义的宏, 在插入头文件内容的同时, GCC 的预处理器 CPP 会展开这些宏, 我们在 example.i 中也并未看到这些宏; stdio.h 还包含许多条件编译指令, 如 #if, #ifdef, CPP 在预处理时应该也评估了这些指令, example.i 中并不包含这些条件编译指令; stdio.h 中还包含了注释, CPP 也会将其删除。同时预处理器还会产生类似“9:# 1”/usr/include/stdio.h” 1 3 4”这样的标记, 这表示预处理过程中插入了 stdio.h 头文件的内容。

3.3 编译阶段

使用以下命令

```
1 gcc -S example.c -fverbose-asm -fdump-tree-all -fdump-rtl-all -fdump-ipa-all
```

让 GCC 编译阶段停止在汇编阶段, “-fdump-tree-all -fdump-rtl-all -fdump-ipa-all” 选项让 GCC 列出编译阶段中所有树表示 (tree)、RTL 表示和 IPA 表示的中间文件。对于 example.c, 不添加优化选项, 将得到以下文件。

example.c	example.c.048i.remove_symbols	example.c.271r.split1
example.c.000i.cgraph	example.c.060i.targetclone	example.c.273r.dfini
example.c.000i.ipa-clones	example.c.064i.free-fnsummary1	example.c.274r.mode_sw
example.c.000i.type-inheritance	example.c.068i.whole-program	example.c.275r.asmcons
example.c.004t.original	example.c.074i.hsa	example.c.280r.ira
example.c.005t.gimple	example.c.075i.fnsummary	example.c.281r.reload
example.c.007t.omplower	example.c.076i.inline	example.c.285r.split2
example.c.008t.lower	example.c.078i.free-fnsummary2	example.c.289r.pro_and_epilogue
example.c.011t.eh	example.c.080i.single-use	example.c.292r.jump2
example.c.012t.cfg	example.c.081i.comdats	example.c.305r.stack
example.c.013t.ompexp	example.c.082i.materialize-all-clones	example.c.306r.alignments
example.c.014t.printf-return-value1	example.c.084i.simdclone	example.c.308r.mach
example.c.016i.visibility	example.c.085t.fixup_cfg3	example.c.309r.barriers
example.c.017i.build_ssa_passes	example.c.221t.veclower	example.c.313r.cet
example.c.018t.fixup_cfg1	example.c.222t.cplxlower0	example.c.314r.shorten
example.c.019t.ssa	example.c.224t.switchlower_00	example.c.315r.nothrow
example.c.022i.opt_local_passes	example.c.231t.optimized	example.c.316r.dwarf2
example.c.023t.fixup_cfg2	example.c.233r.expand	example.c.317r.final
example.c.024t.local-fnsummary1	example.c.234r.vregs	example.c.318r.dfinish
example.c.025t.einline	example.c.235r.into_cfglayout	example.c.319t.statistics
example.c.043t.profile_estimate	example.c.236r.jump	example.c.320t.earlydebug
example.c.046t.release_ssa	example.c.248r.reginfo	example.c.321t.debug
example.c.047t.local-fnsummary2	example.c.270r.outof_cfglayout	example.s

图 3.3: GCC 编译中间表示文件

从以上文件列表中, 我们可以猜测 GCC 编译阶段 cc1 所做的工作, 包括但不限于将源代码转换为树的中间表示、产生控制流图 CFG 以及进行优化 optimization 等。我们将在后续对编译各阶段的讨论中探索其中几个文件。

3.3.1 词法分析

预处理后, GCC 调用 cc1 编译器将 example.i 编译成汇编语言代码文件。经过查阅, 我们了解到: GCC 词法分析是由预处理器 CPP 完成 [5], CPP 将源代码转换为 tokens。供后续语法分析使用。词法分析主要由 gcc/c-family/c-lex.c、gcc/c-family/c-common.c 和 gcc/libcpp/lex.c 等文件实现的功能来处理 [6] (文件路径可能有所不同, 比如带有版本号)。lex.c 实现了词法分析的主要框架, 其中的 `_cpp_lex_direct` 和 `_cpp_lex_token` 是较为核心的函数。

`_cpp_lex_direct` 实现了 C 语言的词法分析规则, 它可以识别数字、字母、注释、字符串等 C 语言的 token, 使用 switch-case 来实现读取到不同字符时的处理逻辑。`_cpp_lex_token` 负责从输入中读取下一个 token。而 `_cpp_lex_token` 会调用 `_cpp_lex_direct`, 直到读取到一个完整的 token。识别得到的 token 被保存在一个名为 `cpp_token` 的结构体中, 其中保存了每个 token 的位置信息、类型信息等。

我们知道预处理阶段 CPP 会进行宏定义替换等操作，这显然需要对源代码进行词法分析等操作。而预处理器 CPP 词法分析得到的 tokens 并不会直接被后续的语法分析等阶段所使用，而是经过 c-lex.c 中实现的操作转换成别的形式后使用。c-lex.c 主要在 C 前端和预处理器 CPP 之间提供了接口，核心函数为 `c_lex_with_flags`，它会调用预处理器 CPP 提供的函数 `cpp_get_token` 来获取下一个 token，然后根据 token 的类型进行不同的处理，将其转换为 C 前端处理所需要的形式。

GCC 编译器并没有提供让我们看到词法分析结果的方法，但我们可以通过以下命令

```
gcc -E -fdebug-cpp example.c
```

在标准输出中观察到关于预处理阶段的额外调试信息。截取部分调试信息如下

```
1 {P:example.c;F:<NULL>;L:30;C:1;S:0;M:0x7fdc24fff4e0;E:0,LOC:13834210,R:13834210}
2 int
3 {P:example.c;F:<NULL>;L:30;C:5;S:0;M:0x7fdc24fff4e0;E:0,LOC:13834344,R:13834344}
4 factorial
5 {P:example.c;F:<NULL>;L:30;C:14;S:0;M:0x7fdc24fff4e0;E:0,LOC:13834624,R:13834624}
6 (
7 {P:example.c;F:<NULL>;L:30;C:15;S:0;M:0x7fdc24fff4e0;E:0,LOC:13834658,R:13834658}
8 int
9 {P:example.c;F:<NULL>;L:30;C:19;S:0;M:0x7fdc24fff4e0;E:0,LOC:13834784,R:13834784}
10 n
11 {P:example.c;F:<NULL>;L:30;C:20;S:0;M:0x7fdc24fff4e0;E:0,LOC:13834816,R:13834816}
12 )
```

为了方便观察，在以上展示中调整了部分格式。可以看到，预处理输出的调试信息都有相似格式如 “{P:...}int”，结合源程序 `example.c`，我们发现，P 是源文件名即 `example.c`，L 是行号 30，C 是列号，而 “{...}int”、“{...}factorial” 中的 `int`、`factorial` 就是 `example.c` 中的关键字和标识符。容易看出，以上调试信息就是对阶乘函数声明 “`int factorial(int n)`” 的解析。通过调试信息，我们可以认为预处理阶段 CPP 确实做了词法分析的工作。

3.3.2 语法分析 & 语义分析

GCC 的语法分析和语义分析主要由 `gcc/c/c-parser.c` 中实现的解析器 `parser` 完成，通过一系列函数解析源代码，可以处理各种语法结构和预处理指令，如解析声明或函数定义的函数 `c_parser_declaration_or_fndef` 和解析结构体的函数 `c_parser_struct_or_union_specifier`，然后生成相应的抽象语法树。

`c_parser.c` 中的 `c_parser_file()` 函数是整个解析过程的入口函数，它初始化解析器，并处理预编译头文件，然后调用 `c_parser_translation_unit()` 解析整个 C 源文件，包括源代码和包含的头文件的集合。`c_parser_translation_unit()` 中会循环调用 `c_parser_external_declaration()` 来解析每个外部声明，包括函数定义、全局变量等。`c_parser_external_declaration()` 通过调用 `c_parser_peek_token()` 来获取 token 并检查它是哪种关键字，并调用不同的解析函数来处理。`c_parser_peek_token()` 通过调用 `C_lex_one_token()` 获取下一个 token 并计数，`C_lex_one_token()` 调用 `c_lex_with_flags()` 来获取下一个 token 的类型，并根据类型执行相关操作。`c_lex_with_flags()` 定义在 `c-lex.c` 中，它接着会调用 `c-lex.c` 和 `lex.c` 中定义的词法分析函数来完成词法解析。这印证了 “c-lex.c 主要在 C 前端和预处理器 CPP 之间提供了接口”。同时这也说明，GCC 的词法分析和语法分析是协同进行的，当语法分析需要下一个 token 时，就会调用词法分析器获取 token。

我们可以使用以下命令，来观察 example.c 经过解析后生成的抽象语法树的原始表示形式，即“GENERIC”表示形式。

```
gcc -fdump-tree-original-raw example.c
```

上述命令会生成一个 example.c.004t.original 文件，其中内容十分复杂，不易阅读。实际上，使用编译选项“-fdump-tree-original”可以生成经过格式化和简化的抽象语法树文件，更易阅读。

我们可以使用一些特定脚本 [2] 和 Graphviz 提供的 dot 工具来可视化 example.c.004t.original 文件，得到下图。

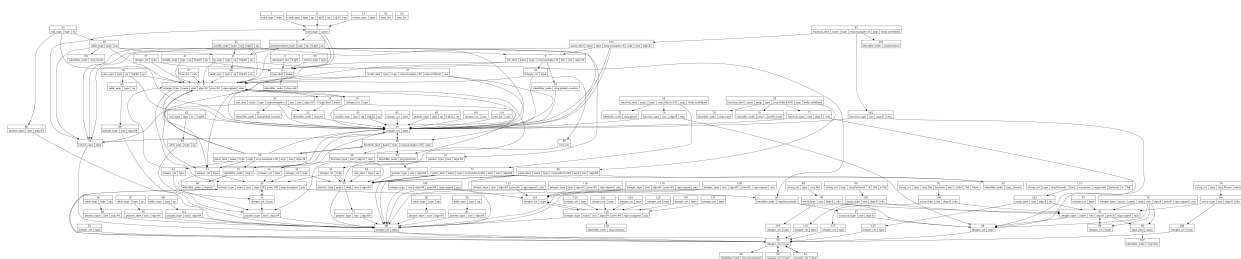


图 3.4: example.c.004t.original 抽象语法树

可以看到非常复杂。由于时间和精力的限制，我们不再详细地分析以上抽象语法树的结构。

3.3.3 中间代码生成

example.c.004t.original 中的 GENERIC 表示形式是 GCC 使用的一种中间表示，可以表示复杂表达式和语法结构，且易于前端生成，但不易于后续的优化 (cite)。GCC 会将其转化为另一种中间表示，即 GIMPLE。GIMPLE 是 GCC 采用的一种中间表示形式，它将 GENERIC 表达式分解为不超过三个操作数的元组（函数除外），并将 GENERIC 中使用的所有控制结构都修改为条件跳转 [4]。

可以使用以下命令查看 GCC 生成的 GIMPLE 形式。

```
gcc -fdump-tree-gimble example.c
```

然后会得到 example.c.005t.gimble 文件。与 .original 文件一样，这样得到的 GIMPLE 表示是经过格式化和简化的。同样的也可以使用“-fdump-tree-gimble-raw”得到原始的 GIMPLE 表示。但为了简化后续的分析，我们选择使用 GENERIC 和 GIMPLE 的简化形式进行比较。简化的 example.c.004t.original 如下所示，仅截取主函数部分。

example.c.004t.original

```
;; Function main (null)
;; enabled by -tree-original
{
  int n;
  int result;

  int n;
  int result;
  printf ((const char * restrict) "Please enter an integer: ");
```

```

11     scanf ((const char * restrict) "%d", &n);
12     if (n > 10)
13     {
14         printf ((const char * restrict) "Exceeds limit, maximum supported factorial
15             is %d.\n", 10);
16         return 1;
17     }
18     result = factorial (n);
19     printf ((const char * restrict) "Factorial of %d is: %d\n", n, result);
20     printf ((const char * restrict) "The function was called %d times\n",
21         global_counter);
22     return 0;

```

简化的 example.c.005t.gimple 如下所示，仅截取主函数部分。

example.c.005t.gimple

```

1  main ()
2  {
3      int D.2326;
4
5      {
6          int n;
7          int result;
8
9          try
10         {
11             printf ("Please enter an integer: ");
12             scanf ("%d", &n);
13             n.0_1 = n;
14             if (n.0_1 > 10) goto <D.2324>; else goto <D.2325>;
15             <D.2324>:
16             printf ("Exceeds limit, maximum supported factorial is %d.\n", 10);
17             D.2326 = 1;
18             // predicted unlikely by early return (on trees) predictor.
19             return D.2326;
20             <D.2325>:
21             n.1_2 = n;
22             result = factorial (n.1_2);
23             n.2_3 = n;
24             printf ("Factorial of %d is: %d\n", n.2_3, result);
25             global_counter.3_4 = global_counter;
26             printf ("The function was called %d times\n", global_counter.3_4);
27             D.2326 = 0;
28             return D.2326;
29         }
30     finally
31     {

```

```

32     n = {CLOBBER};
33 }
34 }
35 D.2326 = 0;
36 return D.2326;
37 }

```

可以看到, GENERIC 形式更接近源代码, 保留了较多的高级语言特性, 其中的变量声明、控制流结构、函数调用形式都与源代码接近一致。而 GIMPLE 中代码已经被大大简化。

在 GIMPLE 中, 变量被分解并重命名, 如变量 `n` 被分解为 `n.0_1`、`n.1_2` 和 `n.2_3`, 这符合编译器 SSA 形式, 每个变量只被赋值一次。同时开头定义了一个临时变量 `D.2326` 来存储返回值。GIMPLE 中的控制流被转换为显式的 `goto` 语句, 代替了 GENERIC 中的分支语句。GIMPLE 还引入了 `try` 和 `finally` 块进行异常处理。GIMPLE 中函数调用形式不变, 但其中的参数已经转换成如 `n.1_2` 的形式。

GIMPLE 中间表示会继续被转换成控制流图 CFG 表示。使用命令

```
1 gcc -fdump-tree-cfg-graph example.c
```

GCC 会生成适合与 Graphviz 一起使用的 `.dot` 文件, 在这里是 `example.c.012t.cfg.dot`, 然后就可以使用的 `dot` 工具查看该图。

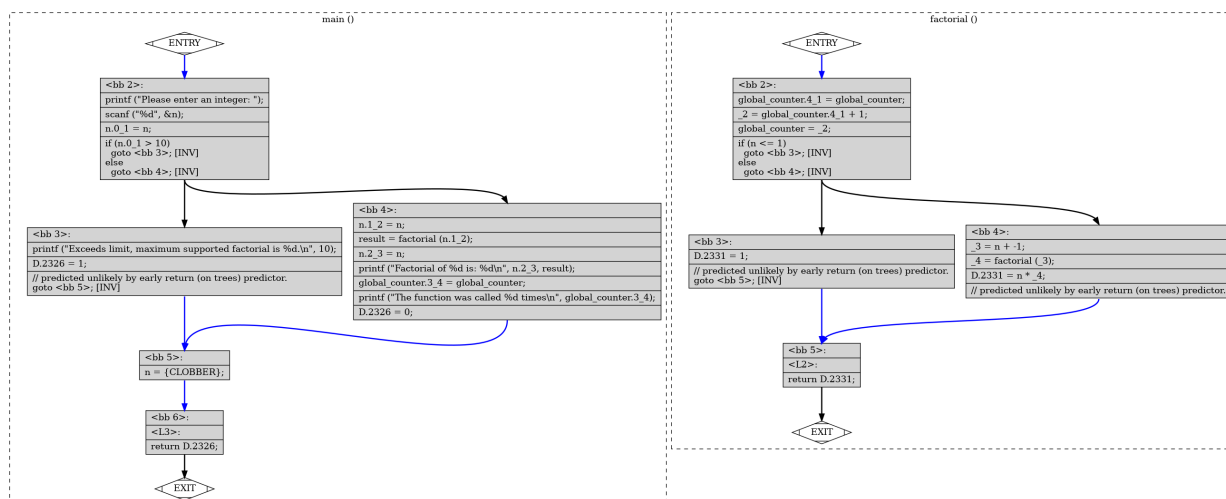


图 3.5: example.c 的 CFG

图中左半部分是 `main()` 函数的控制流, 右半部分是 `factorial()` 函数的控制流, 每一个方框代表一系列顺序执行的指令, 箭头代表控制流的方向。在 `factorial()` 函数的控制流依然能看见全局变量 `global_counter` 和递归调用的形式, 且与 GIMPLE 表示十分相似。

GIMPLE-CFG 表示将在几次优化后被转换为 SSA 形式, 即得到 `example.c.019t.ssa` 文件, 之后继续进行优化, 最后得到 `example.c.231t.optimized` 文件。

然后 GCC 会生成一种更低级的中间表示, 即 Register Transfer Language (RTL)。RTL 也是一种基于三地址码的中间表示, 更接近汇编语言。此时我们得到文件 `example.c.233r.expand`。可以使用类似的命令

```
1 gcc -fdump-rtl-expand-graph example.c
```

让 GCC 生成相应的.dot 文件，从而得到下图。

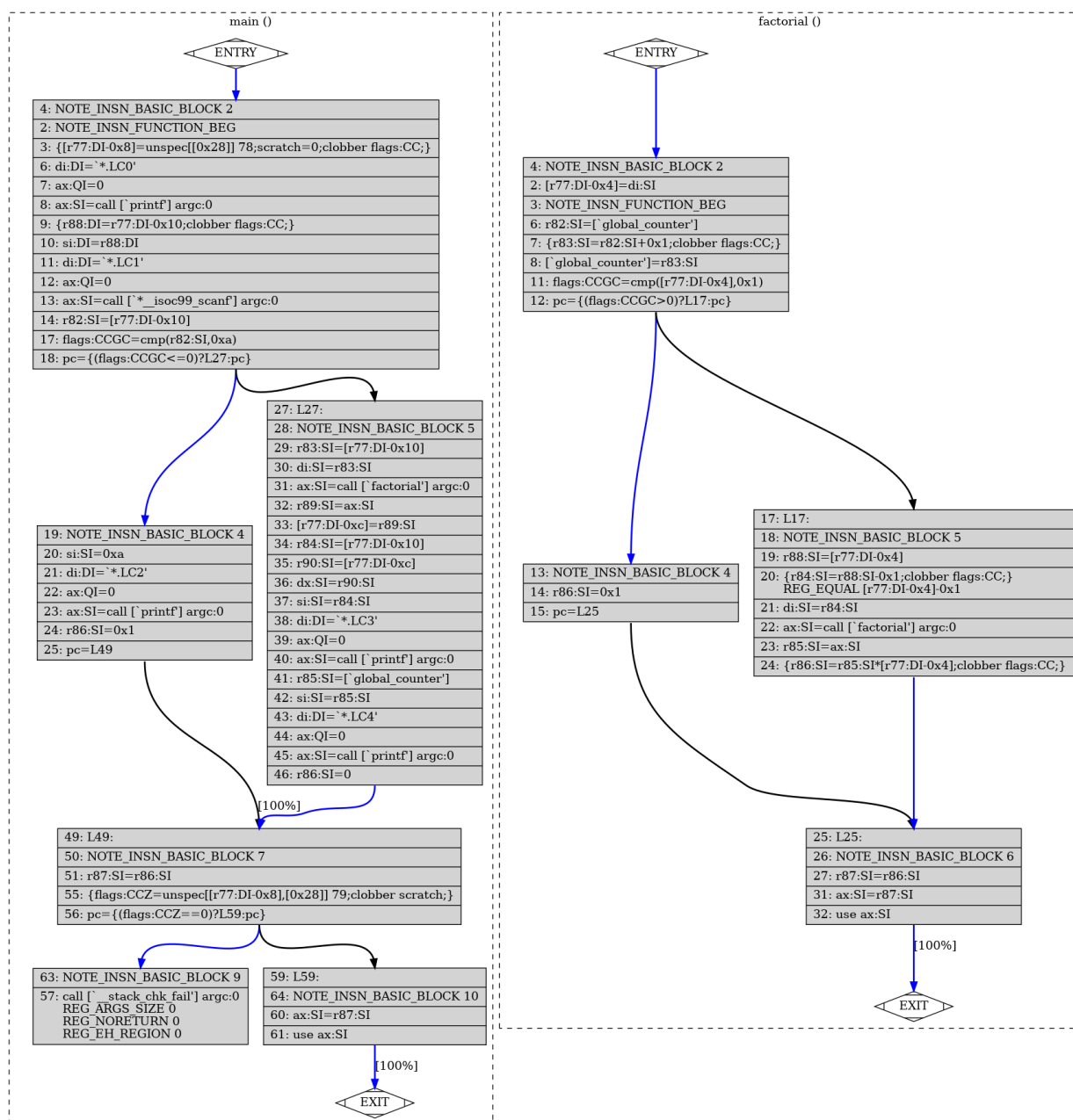


图 3.6: example.c 的 rtl

与 CFG 类似，上图左半部分为 main() 函数的 RTL 表示，右半部分为 factorial() 函数的 RTL 表示，每个方框表示顺序执行的一组 RTL 指令。可以看出，每个块中包含了寄存器、内存和常量之间的操作。在右半部分，我们依然能看到使用 call 指令进行递归调用，以及参数的传递。RTL 中条件跳转更加详细，进一步显示了条件寄存器的使用以及跳转的具体方式。这表明我们的 example.c 源文件已经一步步向更底层的代码进行转化。

RTL 表示经过进一步的优化后，会得到 example.c.317r.final 和 example.c.318r.dfinish 文件。根据文件编号，我们知道此时已经很接近汇编代码了。并且此时 GCC 已不再生成相应的.dot 文件，这说明我们的代码已经被扁平化为线性的指令流了，分支和跳转都已经被内联到指令中，不再以图结构表示控制流 [5]。

3.3.4 代码优化

我们在上一节中间代码生成的讨论中已经看到, GCC 的编译器会将 example.c 在编译阶段转换为不同的中间表示形式并进行不同的优化, 转换过程和优化过程交替进行。实际上, 这些不同的转换和优化被称为 pass。GCC 的编译器过多个 pass 将源代码逐步转化为汇编代码, 并在不同的阶段进行多种不同的优化。对于我们使用的编译器版本, 这些 pass 定义在 gcc9.4.0/gcc/passes.def 中, 形式如下。

```

1 // ...
2 /* Interprocedural optimization passes. */
3 INSERT_PASSES_AFTER (all_small_ipa_passes)
4 NEXT_PASS (pass_ipa_free_lang_data);
5 NEXT_PASS (pass_ipa_function_and_variable_visibility);
6 NEXT_PASS (pass_build_ssa_passes);
7 PUSH_INSERT_PASSES_WITHIN (pass_build_ssa_passes)
8     NEXT_PASS (pass_fixup_cfg);
9     NEXT_PASS (pass_build_ssa);
10    NEXT_PASS (pass_warn_nonnull_compare);
11    NEXT_PASS (pass_early_warn_uninitialized);
12    NEXT_PASS (pass_ubsan);
13    NEXT_PASS (pass_nothrow);
14    NEXT_PASS (pass_rebuild_cgraph_edges);
15 POP_INSERT_PASSES ()
16 // ...

```

其中 INSERT_PASSES_AFTER(PASS) 指定从某个 Pass 开始插入的位置, NEXT_PASS(PASS) 定义下一个要执行的 Pass, POP_INSERT_PASSES(): 结束嵌套的 Pass 列表。

编译过程中的 pass 可以分成以下几类 [4]:

- Parsing pass
- Gimplification pass
- Inter-procedural optimization passes
- Tree SSA passes
- RTL passes

其中 Parsing pass 只进行一次, 用于解析源代码, 对于我们的 example.c, 它将 C 语言转换为 GENERIC 中间表示; Gimplification pass 将中间表示转换为 GIMPLE 表示; Inter-procedural optimization passes 分析多个函数或多个编译单元之间的相互关系, 来提高程序的整体性能; Tree SSA passes 则是基于 GIMPLE 表示的树形结构进行 SSA 形式转换和优化的过程; RTL passes 则是在 Tree optimization passes 后将其转换为 RTL 表示形式并进行优化的过程。这些 passes 与我们在中间代码生成中观察到的一致。

passes.def 中列出了 GCC 编译过程中所有的 pass, 并定义了这些 pass 的执行顺序。这些 pass 中包括生成 CFG 的 pass_build_cfg、转换为 SSA 形式的 pass_build_ssa 等, 许多 pass 都可以与上一节提到的.cfg 文件和.ssa 文件对应。

passes.def 定义了许多优化相关的 pass, 如 pass_ccp 常量条件传播、pass_dce 死代码消除、pass_tail_recursion 尾递归优化、pass_cse 公共子表达式消除等。我们可以使用编译选项“-O1、-O2”等来开启优

化。使用以上编译选项后，GCC 编译阶段产生的中间文件也会有所不同。我们先来看看对于 example.c，开启或关闭 “O2” 优化时得到的 example.c.231t.optimized 有什么不同。

使用命令

```
gcc -O2 -fdump-tree-all-graph -fdump-ipa-all-graph -fdump-rtl-all-graph example.c
```

得到开启 O2 优化的 example.c.231t.optimized.dot，得到下图。

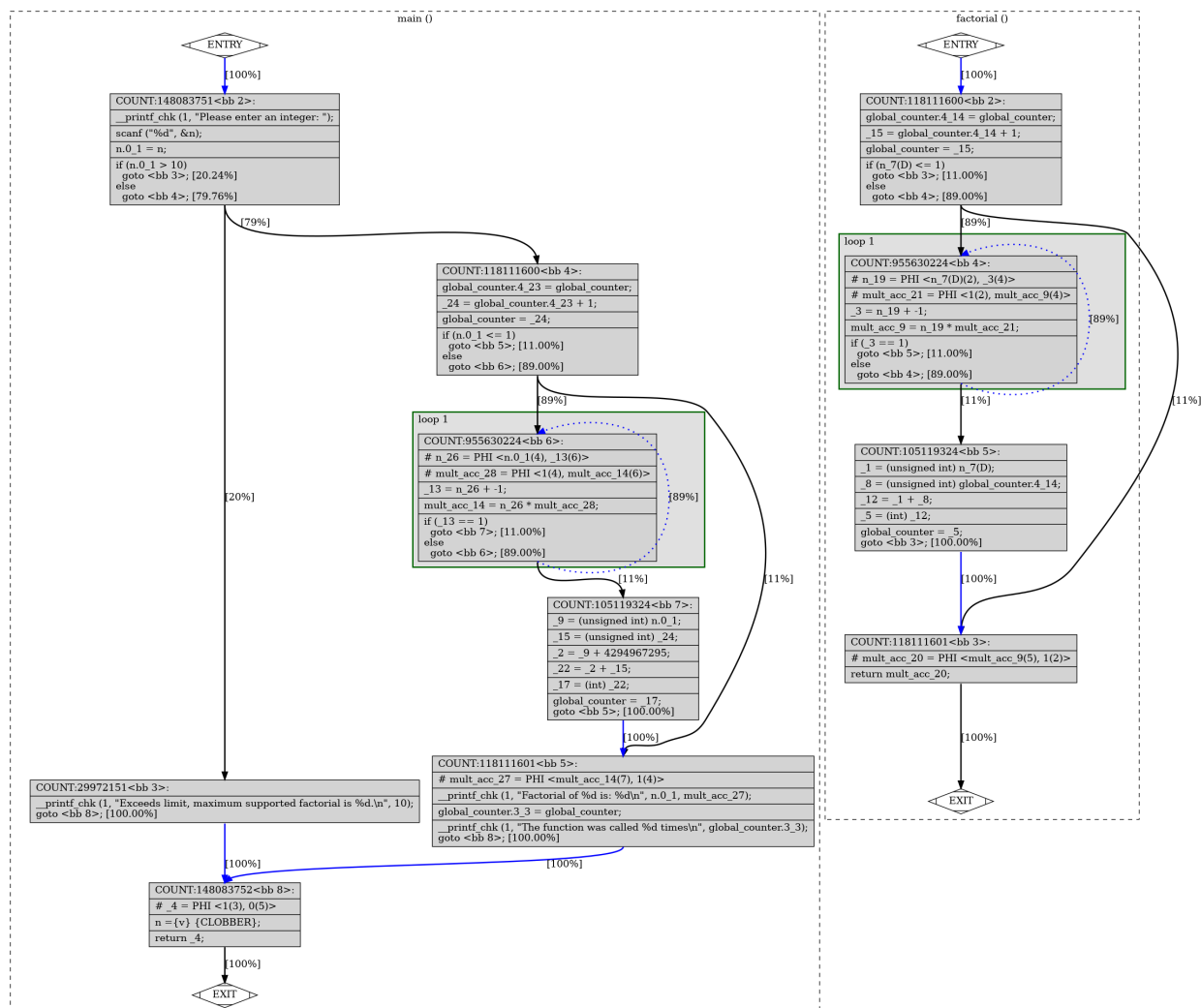


图 3.7: 开启 O2 优化的 example.c.231t.optimized

未开启 O2 优化的如下图。

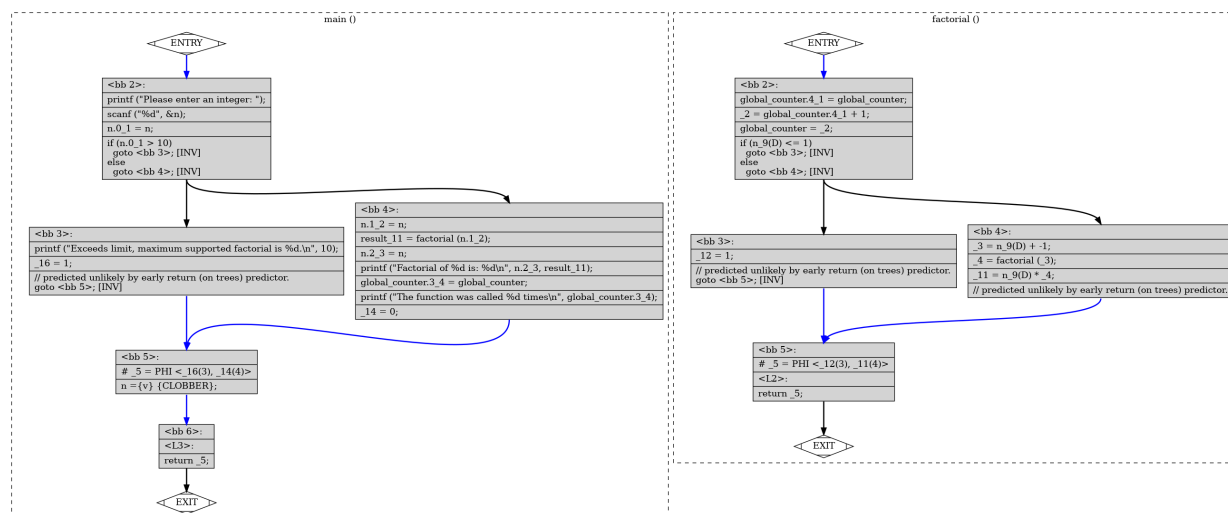


图 3.8: 未开启 O2 优化的 example.c.231t.optimized

对比以上两张 CFG，我们发现 GCC 的编译器做了以下优化：

- 递归的循环展开：在右半部分 factorial() 函数的控制流中，我们看到其递归调用的形式被转换为一个循环，这应该能减少递归形式造成的栈空间的开销。
- 函数内联：左半部分 main() 函数的控制流中，其中调用 factorial() 函数的分支替换成了 factorial() 函数的经过优化的循环形式。
- 分支预测：例如在判断条件 if (n <= 1) 的分支被标记为有不同的执行概率，其他分支和跳转语句也标有相应的执行概率。

在后续的 passes 中编译器还会继续对以上代码进行转换和优化。

我们再来看看课程预习作业 1 中的优化问题。对于以下代码

预习作业 1-hw1.c

```

1  int main(int argc, char *argv[]) {
2      int i;
3      double sum = 0.0;
4      for (int i = 0; i < 100000000; i++) {
5          sum = sum + (1.0 + 1.0);
6      }
7  }

```

我们使用

```

1  gcc -S -O2 hw1.c

```

得到开启 O2 优化编译器输出的汇编代码。对比未开启优化的 hw1.c 的汇编代码，我们发现优化后的代码在 main() 函数中仅进行了“xorl %eax, %eax”将 eax 寄存器清零，并将其作为返回值返回，不进行其他任何实际工作，即程序没有进行循环累加。

我们认为，这是由于以上代码中仅进行了计算，没有接受任何输入也没有任何输出，所以该程序等效为一个不进行任何工作的程序。对于编译器来说，在不改变程序的输入输出的情况下，减少程序

运行的时间空间开销，或许就是好的优化过程。预习作业 1 的例子就是 GCC 编译器死代码消除优化的体现。

3.3.5 代码生成

经过中间代码生成和优化后，GCC 的编译器将代码从 RTL 中间表示转换为对应架构的汇编代码。汇编代码的生成需要参考目标架构的指令集、寄存器等，这些信息保存在 GCC 的 Machine Description 中，而 Machine Description 包括保存指令模式的.md 文件和保存各种宏定义的 C 头文件 [4]。

对于我们使用的 x86_64 架构，其.md 文件等保存在 gcc0.4.0/gcc/config/i386 路径下，在此路径下我们能找到 i386.md 文件、i386.c 文件和 i386.h 文件，还能找到其中 i386.md 描述了 x86_64 架构指令集、寄存器等信息，如定义了与寄存器相关的常量 AX_REG, BX_REG, CX_REG, DX_REG 等。i386.c 和 i386.h 则共同实现了 x86_64 架构相关的逻辑。我们知道 GCC 编译器生成的 RTL 表示已经是与机器相关的，上述这些文件在 RTL 生成和汇编代码生成阶段都用于生成适合目标架构的代码。

我们可以使用以下命令直接生成 example.c 对应的汇编文件

```
1 gcc -S example.c
```

“-S” 编译选项意为让编译阶段停止在汇编阶段之前。或者可以将先前经过预处理的文件 example.i 转换为汇编代码。

```
1 gcc -S example.i
```

3.4 汇编阶段

在“编译流程概述”一节我们提到，GCC 调用汇编器 as 来将生成的汇编代码转换成机器代码，得到.o 目标文件。可以使用以下命令手动将先前生成的汇编代码转化为机器代码。

```
1 gcc -c example.s -o example.o
```

也可以直接调用汇编器 as 来把.s 文件汇编成.o 文件

```
1 as example.s -o example.o
```

使用“file”命令查看“example.o”可以得到

```
1 example.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

说明 example.o 文件是 x86_64 架构下的、64 位的、ELF 文件格式的可重定位目标文件。我们使用 readelf 命令检查 example.o 的 ELF 文件头，得到

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
```

```
ABI Version:                0
Type:                       REL (Relocatable file)
Machine:                    Advanced Micro Devices X86-64
Version:                    0x1
Entry point address:        0x0
Start of program headers:    0 (bytes into file)
Start of section headers:    1704 (bytes into file)
Flags:                      0x0
Size of this header:         64 (bytes)
Size of program headers:     0 (bytes)
Number of program headers:    0
Size of section headers:     64 (bytes)
Number of section headers:    14
Section header string table index: 13
```

上述信息描述了 example.o 这个 ELF 文件的类型、数据格式、版本、目标架构等信息，值得注意的是，信息中提到 Entry point address 为 0x0，这可能表示该程序并没有入口点，所以该 ELF 文件无法执行。

使用 objdump 工具对 example.o 进行反汇编。

```
1 objdump -d example.o
```

在标准输出中会出现 objdump 对 example.o 进行反编译得到的汇编代码。用以下命令使用 nm 工具列出目标文件.o 的符号表。

```
1 nm example.o
```

得到以下信息。

```
                U _GLOBAL_OFFSET_TABLE_
                U __isoc99_scanf
                U __stack_chk_fail
000000000000000c3 T factorial
00000000000000000 B global_counter
00000000000000000 T main
                U printf
```

可以看到在 _GLOBAL_OFFSET_TABLE_ 等符号前写的是 U，而我们定义的 factorial() 函数、main() 函数和全局变量 global_counter 前为 T 和 B。T 代表该符号定义在.text 段，即代码段，factorial() 函数和 main() 函数都定义在.text 段。factorial() 函数前的 0xc3 为偏移地址，而 main() 函数没有偏移，说明是第一个函数。B 代表全局变量 global_counter 符号定义在.bss 段，即非初始化数据段。U 则表示该符号未定义，我们知道 printf 函数等符号的定义是在外部库中的，所以在当前文件下并未定义。

3.5 链接阶段

我们知道 GCC 通过调用 collect2 来控制整个链接过程，同时 collect2 会调用系统上真正的链接器如 ld 进行链接。collect2 会根据一定规则先查找系统中的真正的 ld。开始链接时，collect2 首先会对程序进行一次链接，生成一个初步的输出文件它会扫描输出文件，查找具有特定符号名称的函数，这些符号表示 constructor functions，如果 collect2 找到了这些 constructor functions，它会为它们生成一个临时的 .c 文件，包含一个构造函数表，用于记录这些初始化函数。生成临时文件后，collect2 会将这个文件编译，并将它和原程序一起重新链接。constructor functions 实际调用是通过一个名为 __main 的子程序完成的，如果 main 函数是用 GCC 编译，那这个子程序会在 main 函数的开头自动调用 [4]。

可以使用 GCC 来链接 example.o 生成可执行文件 example。

```
1 gcc example.o -o example
```

也可以直接调用链接器 ld 来进行链接，但需要配置更多的参数保证编译正确。

使用 “file” 命令查看可执行文件 example 可以得到

```
example: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=2febd3ce2dec07c63a46c8565d25a97d447c38f3,
for GNU/Linux 3.2.0, not stripped
```

相较于 example.o，可执行文件 example 的信息包括 dynamically linked 动态链接，以及使用了 /lib64/ld-linux-x86-64.so.2 作为动态链接器来加载和执行。如果使用静态链接，即使用命令

```
1 gcc example.o -static -o example
```

得到的可执行文件再用 “file” 命令检查，得到

```
example: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
statically linked,
BuildID[sha1]=ab4413344f5d6adaa22672036c27f40612a01577,
for GNU/Linux 3.2.0, not stripped
```

可以看到显示 statically linked 静态链接，且没有动态链接器。

使用动态链接得到的可执行文件 example 大小为 16872 字节，而使用静态链接得到的可执行文件大小为 1002328 字节。这符合我们的认知，因为静态链接时可执行文件包含了所依赖的所有库的代码。

使用 nm 工具查看动态链接得到的 example 的符号表，得到的符号表较短，且仍存在标记为 U 的、未定义的符号；而静态链接得到的 example 的符号表更长，不存在未定义的符号，每个符号都有自己的定义位置和偏移地址。

使用 readelf 工具检查动态链接得到的 example 的 ELF 文件头，得到

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                               2's complement, little endian
```

Version:	1 (current)
OS/ABI:	UNIX - System V
ABI Version:	0
Type:	DYN (Shared object file)
Machine:	Advanced Micro Devices X86-64
Version:	0x1
Entry point address:	0x10a0
Start of program headers:	64 (bytes into file)
Start of section headers:	14888 (bytes into file)
Flags:	0x0
Size of this header:	64 (bytes)
Size of program headers:	56 (bytes)
Number of program headers:	13
Size of section headers:	64 (bytes)
Number of section headers:	31
Section header string table index:	30

可以看到此时的 ELF 文件已经有了入口地址 0x10a0，且文件类型 Type 变成 DYN (Shared object file)。如果是静态链接得到的 example 的类型 Type 将为 EXEC (Executable file)。

到这里，我们就已经遍历了 GCC 编译器将 C 代码生成可执行文件的全部流程。

4 LLVM IR 编程

我们先从一个最简单的 C 语言的例子来看看 LLVM IR 语言编写的代码是怎么样子的。我们现在有 hello.c 如下。

main.c

```
1  #include <stdio.h>
2  int main() {
3      printf("Hello, world\n");
4      return 0;
5  }
```

使用以下命令可以将 C 代码文件转换为 LLVM IR 代码格式。

```
1  clang -S -emit-llvm hello.c -o hello.ll
```

其中 -S 表示生成汇编代码，-emit-llvm 表示生成 LLVM 中间表示。然后得到

hello.ll

```
1  ; ModuleID = 'hello.c'
2  source_filename = "hello.c"
3  target datalayout =
4      "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
5  target triple = "x86_64-pc-linux-gnu"
6
7  @.str = private unnamed_addr constant [14 x i8] c"Hello, world\0A\00", align 1
8
9  ; Function Attrs: noinline nounwind optnone uwtable
10 define dso_local i32 @main() #0 {
11     %1 = alloca i32, align 4
12     store i32 0, i32* %1, align 4
13     %2 = call i32 @__printf(i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x
14         i8]* @.str, i64 0, i64 0))
15     ret i32 0
16 }
17
18 declare dso_local i32 @__printf(i8*, ...) #1
19
20 attributes #0 = { noinline nounwind optnone uwtable
21     "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"
22     "frame-pointer"="all" "less-precise-fpmad"="false"
23     "min-legal-vector-width"="0" "no-infs-fp-math"="false"
24     "no-jump-tables"="false" "no-nans-fp-math"="false"
25     "no-signed-zeros-fp-math"="false" "no-trapping-math"="false"
26     "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
27     "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false"
28     "use-soft-float"="false" }
29 attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false"
30     "disable-tail-calls"="false" "frame-pointer"="all"
```



```

"less-precise-fpmad"="false" "no-infs-fp-math"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

```

```

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{"clang version 10.0.0-4ubuntu1 "}

```

观察以上 LLVM IR 格式的 hello.c, 可以看到: 使用分号; 作为注释开头, 标注了该 LLVM 模块的名字; 记录了 hello.c 源文件名称; 记录了目标数据布局 target datalayout 和目标架构 target triple; 定义了一个字符串常量 @.str 用于后续调用 printf 函数; 使用 define 定义了 main 函数, 其中完成了变量分配和 printf 函数调用; 使用 declare 声明了外部函数 printf; 使用 attribute 描述了一些函数的属性; 最后记录了包括编译器版本的元数据。

使用 clang 编译得到的 hello.ll 中, 包含了许多与打印 “Hello, world” 无关的信息, 如编译器的版本信息等。这些信息可能是 clang 为了编译的后续流程记录的, 并不是一个 hello.ll 转换为可执行文件所必需的。查阅 LLVM IR 相关概念后, 我们得到了实现打印 “Hello, world” 的 hello.ll, 其中只包含了实现功能必要的代码, 如下所示。

hello.ll

```

; hello.ll
declare i32 @printf(i8*, ...)

@.str = constant [14 x i8] c"Hello, world\0A\00", align 1

define i32 @main() {
entry:
    %0 = call i32 @printf(i8* @.str, ...)
    ret i32 0
}

```

LLVM IR 中存在模块的概念。模块 (module) 是 LLVM IR 的顶层结构, 通常对应于源代码的一个文件或多个文件。在 hello.ll 中该文件就是一个模块。模块中包含全局变量、函数、外部声明等信息。与 C 语言类似, 我们也需要在 LLVM IR 中定义入口函数, 即 main 函数, 也可以声明自定义函数。在 hello.ll 中, 我们使用 define 定义 main 函数。函数内部需要定义基本块, 类似函数体; 基本块内需要至少包含一个入口块, 即 entry, 和一个终结块, 如 ret, 即返回语句。

在 hello.ll 中, 我们定义了全局变量 @.str, 用作 printf 的参数。其中, constant 表示常量, [14 x i8] 表示这是一个长度为 14 的数组, 其中每个元素是 8 位的整数, 字符串前的 c 表示这是一个 C 风格的字符串。在 LLVM IR 中, 使用 @ 定义全局变量, 使用 % 定义局部变量。

由于 printf 是外部函数, 所以需要 declare 声明。declare 用于声明未定义在当前模块中的全局变量或函数, define 用于声明模块内定义的函数。

在 main 函数中，我们使用 call 来调用 printf 函数。

```
1 %0 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x
    i8]* @.str, i32 0, i32 0))
```

其中 i32 是 printf 的返回值，(i8*, ...) 是 printf 的参数列表，“...”表示接受可变数量的参数；“i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0)”是传递给 printf 的第一个参数，它可以得到 @.str 这一格式化字符串的第一个字符的地址。“%0”表示将 printf 的返回值存储在临时变量 %0 中。如果不指定返回值存储的变量，那么 LLVM IR 的解释器会自动为其赋予一个临时变量。为了防止自动赋予的临时变量与我们定义的变量冲突，所以最好手动指定。

为了探索 LLVM IR 函数调用，我们编写了 add.ll 如下。

add.ll

```
1 ; add.ll
2 ; input: 1 2
3 ; output: 3
4
5 define i32 @add(i32 %a, i32 %b) {
6 entry:
7     %0 = add i32 %a, %b
8     ret i32 %0
9 }
10 ; 为了缩减篇幅，报告中省略接受全局变量定义和外部函数声明部分
11 define i32 @main() {
12 entry:
13     ; 为了缩减篇幅，报告中省略接受用户输入和加载变量的部分
14
15     %2 = call i32 @add(i32 %0, i32 %1)
16
17     ; 为了缩减篇幅，报告中省略输出的部分
18
19     ret i32 0
20 }
```

使用“define i32 @add(i32 %a, i32 %b)”来自定义函数，define 后面指定函数的返回值，括号内是函数的参数列表，这里表示接受两个局部变量。在 main 函数中，使用 call 即可调用 add 函数。

对于如何使用函数交换两个变量的值的经典问题，我们发现 LLVM 中也可以定义函数接受指针作为参数，如下所示。

swap.ll

```
1 ; swap.ll
2 ; output: 20 10
3 define void @swap(i32* %a, i32* %b) {
4 entry:
5     %0 = load i32, i32* %a, align 4
6     %1 = load i32, i32* %b, align 4
7 }
```

```

8      store i32 %1, i32* %a, align 4
9      store i32 %0, i32* %b, align 4
10
11      ret void
12  }
```

在 swap 函数中，接收两个指向 i32 的指针作为参数，然后可以使用 load 指令获取保存在该地址的值，从而改变该地址对应变量的值。

为了探索 LLVM 中函数调用的更多细节，我们编写了 fibonacci.ll，使用递归的形式计算 Fibonacci 数，其函数实现如下所示。

fibonacci.ll

```

1      ; fibonacci.ll
2      ; input: 10
3      ; output: 55
4      define i32 @fibonacci(i32 %n) {
5      entry:
6          ; if n <= 1
7          %0 = icmp sle i32 %n, 1
8          br i1 %0, label %end, label %recur
9
10     end:
11         ret i32 %n
12
13     recur:
14         ; fibonacci(n-1)
15         %1 = sub i32 %n, 1
16         %2 = call i32 @fibonacci(i32 %1)
17
18         ; fibonacci(n-2)
19         %3 = sub i32 %n, 2
20         %4 = call i32 @fibonacci(i32 %3)
21
22         ; fibonacci(n-1) + fibonacci(n-2)
23         %5 = add i32 %2, %4
24         ret i32 %5
25     }
```

在 fibonacci 函数中，使用 call 递归调用 fibonacci 函数。该函数中还使用了分支跳转相关指令，“icmp sle”表示整数比较的有符号大于小于，这里比较局部变量 n 与 1 的大小，相当于 if n <= 1。如果比较为真，则将比较结果保存到临时变量 %0 中，然后使用“br”条件分支跳转，如果 %0 为 1，那么跳转到 label %end；如果 %0 为 0，那么跳转到 label %recur。

为了探索 LLVM 中数组相关的细节，我们编写了 bubble_sort.ll，由于代码较长，不在报告中完整展示，仅在以下讲解中展示部分代码。

我们首先设置局部变量 arr_len，用于读取用户输入指定数组长度，并保存输入值。然后使用

```

1      %arr = alloca i32, i32 %arr_len_val, align 4
```

```
%arr_ptr = getelementptr i32, i32* %arr, i32 0
```

来为数组分配内存空间，“alloca”在栈上分配内存，第一个 i32 表示分配内存的元素类型，“i32 %arr_len_val”表示要分配的 i32 类型元素的数量，然后获取数组元素的首地址，相当于“int* arr_ptr = &arr[0];”。其中“getelementptr”用于计算指针偏移，第一个 i32 指示指针类型，“i32* %arr”是计算时的首地址即起点，“i32 0”指偏移量。

和 C 语言类似，我们需要循环读取输入并保存为数组元素，如下所示。

```
1   br label %read_loop
2
3 read_loop:
4   %i = phi i32 [0, %entry], [%i_next, %read]
5   %cmp0 = icmp slt i32 %i, %arr_len_val
6   br i1 %cmp0, label %read, label %sort
7
8 read:
9   %ptr0 = getelementptr i32, i32* %arr_ptr, i32 %i
10  %0 = call i32 @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]*
    @str_input, i32 0, i32 0), i32* %ptr0)
11  ; i = i + 1
12  %i_next = add i32 %i, 1
13  br label %read_loop
```

这里使用“%i”作为循环时的索引，在 read_loop 块中，比较 i 的值和数组长度，从而确定循环次数。其中使用了 phi 指令来初始化和更新 i 的值。“[0, %entry]”表示如果控制流是从 %entry 进入，phi 指令选择值 0，并将值赋值给 i；“[%i_next, %read]”表示当控制流从 %read 进入时，phi 指令选择 %i_next 的值。read_loop 相当于“for (int i = 0; i < n; i=i_next)”，其中“i_next=i+1”的计算在 read 块中完成。read 块内也是循环体的实现。数组排序后输出结果与读取输入时类似。

对于 bubble_sort 函数的实现，我们已经知道如何在 LLVM 中写出类似 for 循环的语句，也知道如何比较两个数的大小、如何交换两个元素，只需要将这些知识结合运用就可以写出正确的排序算法。由于代码较长，报告中不再贴出。

对于编写好的.ll 文件，可以使用以下命令直接解释执行。

```
lli bubble_sort.ll
```

也可以用 clang 将其编译为可执行文件。

```
clang bubble_sort.ll -o bubble_sort
```

我们实现的所有 LLVM IR 代码均已上传至 [Github](#)。

5 SysY 语言以及 ARM/RISC-V 汇编编程

5.1 SysY 示例程序

SysY 语言是 C 语言的一个子集（是编译系统设计赛要实现的编程语言）。每个 SysY 程序的源码存储在一个扩展名为 `sy` 的文件中。该文件中有且仅有一个名为 `main` 的主函数定义，还可以包含若干全局变量声明、常量声明和其他函数定义。SysY 语言支持 `int` 类型和元素为 `int` 类型且按行优先存储的多维数组类型，其中 `int` 型整数为 32 位有符号数；`const` 修饰符用于声明常量。

我们使用 SysY 语言编写了下面的示例程序：

```

1  int n, res;
2
3  int main(){
4      n = getint();
5      int i;
6      i = 2;
7      res = 1;
8      while(i<=n){
9          res = res * i;
10         i = i + 1;
11     }
12     putfloat(res);
13
14     return 0;
15 }
```

该程序实现的是阶乘的功能，从语法上来看和普通的 c 程序基本一致，区别是其输入使用 `get` 从 `.in` 文件中获取输入流，输出使用 `put` 将输出流写入 `.out` 文件。

输入下面的指令将 `.sy` 文件编译成 `riscv` 汇编代码：

```

1  ./SysY_test_single.sh factorial.sy S 01
```

得到的 `.s` 文件（部分）如下所示：

```

1  .text
2  .globl main
3  .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0_zba1p0_zbb1p0"
4
5  main:
6  .main_0:
7      addi    sp,sp,-16
8      sd      ra,8(sp)
9      call    getint
10     addiw   t0,x0,2
```

```

11         bgt     t0,a0,.main_8
12 .main_4:
13         addiw   t0,a0,-3
14         addiw   t1,x0,4
15         blt     t0,t1,.main_12
16 .main_5:
17         addiw   t0,a0,-4
18         addiw   t1,x0,1
19         addiw   t2,x0,2
20 .main_1:
21         mulw    t1,t1,t2
22         addiw   t3,t2,1
23         mulw    t4,t1,t3
24         addiw   t3,t2,2
25         mulw    t5,t4,t3
26         addiw   t3,t2,3
27         mulw    t1,t5,t3
28         addiw   t2,t2,4
29         bgt     t2,t0,.main_11
30 # 后面还有约 500 行代码, 在此省略
31 # ...

```

5.2 ARM 汇编编程

在 ARM 汇编编程部分, 我们根据 `factorial.sy` 编写了等价的 ARM 汇编程序 `factorial.s`, 实现的也是阶乘的功能。

关键代码如下所示:

```

1 .text
2 .global main
3 .type main, %function
4 main:
5     push {r7, lr}
6     sub sp, sp, #8
7     add r7, sp, #0
8     ldr r3, .L5
9     add r3, pc
10    mov r1, r3
11    ldr r3, .L5+4
12    add r3, pc
13    mov r0, r3
14    bl __isoc99_scanf(PLT)
15    movs r3, #2

```

```
16     str r3, [r7, #4]
17     ldr r3, .L5+8
18     add r3, pc
19     movs r2, #1
20     str r2, [r3]
21     b .L2
22 .L3:
23     ldr r3, .L5+12
24     add r3, pc
25     ldr r3, [r3]
26     ldr r2, [r7, #4]
27     mul r2, r3, r2
28     ldr r3, .L5+16
29     add r3, pc
30     str r2, [r3]
31     ldr r3, [r7, #4]
32     adds r3, r3, #1
33     str r3, [r7, #4]
34 .L2:
35     ldr r3, .L5+20
36     add r3, pc
37     ldr r3, [r3]
38     ldr r2, [r7, #4]
39     cmp r2, r3
40     ble .L3
41     ldr r3, .L5+24
42     add r3, pc
43     ldr r3, [r3]
44     mov r1, r3
45     ldr r3, .L5+28
46     add r3, pc
47     mov r0, r3
48     bl printf(PLT)
49     movs r3, #0
50     mov r0, r3
51     adds r7, r7, #8
52     mov sp, r7
53     pop {r7, pc}
54 .L5:
55     .word n-(.LPIC0+4)
56     .word .LC0-(.LPIC1+4)
57     .word res-(.LPIC2+4)
```



```

58     .word res-(.LPIC3+4)
59     .word res-(.LPIC4+4)
60     .word n-(.LPIC5+4)
61     .word res-(.LPIC6+4)
62     .word .LC1-(.LPIC7+4)
63     .size main, .-main

```

这段代码的核心部分实现了阶乘的计算。首先，函数 `main` 通过 `push` 和 `sub` 指令设置栈帧。接着，使用 `ldr` 和 `mov` 指令加载地址并调用 `__isoc99_scanf` 函数以获取输入值。然后，代码初始化循环变量并进入循环部分，通过 `ldr` 指令加载当前值，使用 `mul` 指令计算阶乘，最后更新循环变量。循环继续执行，直到比较变量和目标值时触发 `cmp` 指令，判断是否需要继续。完成后，结果通过 `printf` 输出，并最终清理栈帧返回。这段代码展示了基本的控制流结构，包括循环和条件判断。

需要注意的是 ARM 汇编中涉及栈操作的 `push` 和 `pop` 指令。其中 `push` 指令将一个或多个寄存器的值保存到栈上，通常用于保存函数调用前的寄存器状态，以便在函数返回时能够恢复。在 `factorial.s` 中，`push r7, lr` 将寄存器 `r7` 和链接寄存器 `lr` 的值压入栈中，保证函数在执行期间能安全使用这些寄存器而不丢失原有值。

同样的，`pop` 指令则从栈中恢复之前保存的寄存器值。在代码结束时，使用 `pop r7, pc` 恢复 `r7` 和程序计数器 `pc`，使程序能够返回到调用此函数的正确位置。通过这两条指令，程序能够在函数调用之间管理寄存器状态，维护函数调用的完整性和正确性。

报告中仅展示了部分代码，完整代码请参见 [Github](#)

5.3 RISC-V 汇编编程

在 RISC-V 汇编编程部分，我们另外设计了两个程序，分别是 `gcd.s` 和 `sort.s`。其中 `gcd.s` 实现的是辗转相除法计算最大公约数的功能，主要测试的是 RISC-V 中的控制流，`sort.s` 实现的是一个冒泡排序的功能，主要测试的是 RISC-V 中数组的实现。

`gcd.s` 中的关键代码如下所示：

```

1  gcd:
2  mv a6, a0
3  mv a5, a1
4  bgt a1, a0, .L1
5  mv a5, a0
6  mv a6, a1
7
8  .L1:
9      beq a6, zero, .L3
10     bgt a6, a5, .L2
11     sub a5, a5, a6
12     j .L1
13
14  .L2:
15     mv a7, a5

```

```

16      mv a5, a6
17      mv a6, a7
18      j .L1
19
20  .L3:
21      sext.w a0, a5
22      ret

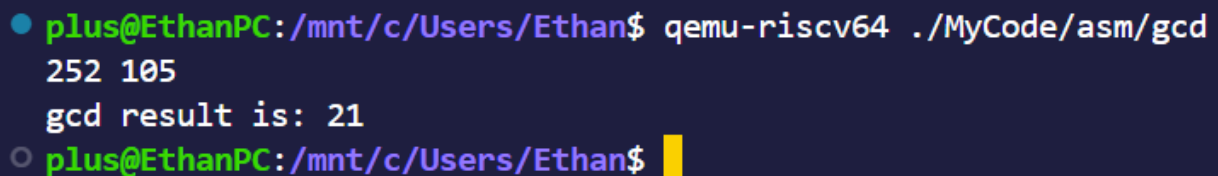
```

这段汇编代码首先对 `scanf` 传来的参数进行比大小，将较大者赋值给 `a5`，将较小者赋值给 `a6`。然后在 `.L1` 中循环用 `a6` 去减 `a5` 直到 `a5` 的值小于 `a6`。接着当 `a5` 的值小于 `a6` 时，将 `a5` 赋值给 `a7`（作为临时存储），将 `a6` 赋值给 `a5`，完成一次辗转相除操作。最后直到 `a6` 的值为 0 的时候跳出循环，`a5` 中的值即为最开始两数的最大公约数。

使用下面的指令将 `gcd.s` 编译成可执行文件：

```
1 riscv64-unknown-linux-gnu-gcc MyCode/asm/gcd.s -o MyCode/asm/gcd -static
```

运行结果如下所示：



```

plus@EthanPC:/mnt/c/Users/Ethan$ qemu-riscv64 ./MyCode/asm/gcd
252 105
gcd result is: 21
plus@EthanPC:/mnt/c/Users/Ethan$

```

`sort.s` 的关键代码如下所示：

```

1  # 加载循环计数器 j 的值
2  lw a5,-24(s0)
3  slli a5,a5,2
4  ld a4,-40(s0)
5  add a5,a4,a5
6  lw a3,0(a5)
7
8  # 加载下一个元素 j + 1
9  lw a5,-24(s0)
10 addi a5,a5,1
11 slli a5,a5,2
12 ld a4,-40(s0)
13 add a5,a4,a5
14 lw a5,0(a5)
15
16 # 比较第 j 个元素与第 j+1 个元素
17 mv a4,a3
18 ble a4,a5,.L4
19
20 # 交换第 j 个和第 j+1 个元素

```

```

21  lw a5,-24(s0)
22  slli a5,a5,2
23  ld a4,-40(s0)
24  add a5,a4,a5
25  lw a5,0(a5)
26  sw a5,-28(s0)
27
28  lw a5,-24(s0)
29  addi a5,a5,1
30  slli a5,a5,2
31  ld a4,-40(s0)
32  add a5,a4,a5
33  lw a4,0(a4)
34  sw a4,0(a5)
35
36  lw a5,-28(s0)
37  sw a5,0(a5)

```

程序首先使用 `lw` 指令读取当前的循环计数器 `j` 的值，并将其乘以 4，以计算出数组中第 `j` 个元素的地址并加载到寄存器 `a3`。接着，它读取 `j + 1` 的值，重复相同的步骤，计算出第 `j + 1` 个元素的地址并加载到寄存器 `a5`。然后，将第 `j` 个元素与第 `j + 1` 个元素进行比较，若第 `j` 个元素小于等于第 `j + 1`，则跳转到不需要交换的步骤；若第 `j` 个元素大于第 `j + 1`，则执行交换操作：先将第 `j` 个元素存储到临时位置，再将第 `j + 1` 个元素移动到第 `j` 的位置，最后将临时存储的第 `j` 个元素放回到第 `j + 1` 的位置。这些步骤是冒泡排序中的关键操作，通过不断比较和交换相邻元素，实现整个数组的排序。

在完整的汇编代码还有一处需要注意：

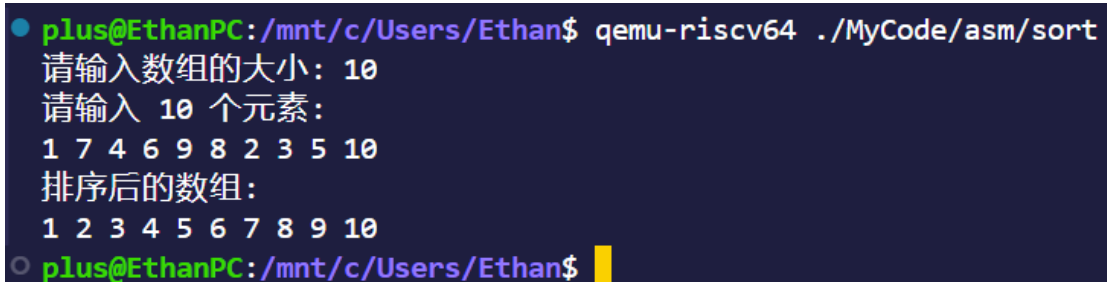
```

1  lui a5,%hi(.LC0)
2  addi a0,a5,%lo(.LC0)

```

这两条指令一起用于加载 `.LC0` 字符串的地址。`lui` 将高 20 位加载到寄存器 `a5` 中，`addi` 将低 12 位加到 `a5` 中，最终形成指向 `.LC0` 的完整地址。这种方式常用于处理大于 2^{12} 的地址，因为 RISC-V 的指令格式限制了立即数的范围（最大为 2^{12} ）。

运行结果如下所示：



```

plus@EthanPC:/mnt/c/Users/Ethan$ qemu-riscv64 ./MyCode/asm/sort
请输入数组的大小：10
请输入 10 个元素：
1 7 4 6 9 8 2 3 5 10
排序后的数组：
1 2 3 4 5 6 7 8 9 10
plus@EthanPC:/mnt/c/Users/Ethan$

```

报告中仅展示了部分代码，完整代码请参见 [Github](#)

6 实验总结

在本次实验中，我们深入研究了 GCC 编译器的工作流程，包括预处理、编译、汇编和链接等阶段。通过分析，我们加深了对源代码到可执行文件转换过程的理解，并深入了解了编译器的核心功能。我们学习了预处理器 CPP、词法分析、语法分析和语义分析的协作机制，以及编译器进行代码优化的具体过程。我们还深入 GCC 源码，了解其功能实现的内部机制。剖析 GCC 这一复杂工具的过程也让我们深刻体会到，“计算机的世界没有秘密”。

此外，我们还实践了 LLVM IR 编程和 ARM/RISC-V 汇编编程，探索了编译器中间表示的结构和特性，以及不同指令集架构的编程方法。这些经验进一步加深了我们对编译器工作机制的理解。

本次实验中我们编写的所有代码均已上传至 [Github](#)。

参考文献

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
- [2] ashimida@. Gcc 源码分析 (二) —词法分析, 2021. <https://blog.csdn.net/lidan113lidan/article/details/119942976>.
- [3] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 3rd edition, 2015.
- [4] Free Software Foundation. *GCC Internal Manual*, 2024. Accessed: 2024-09-17.
- [5] gcc-newbies guide.readthedocs.io. Inside cc1, 2024. <https://gcc-newbies-guide.readthedocs.io/en/latest/inside-cc1.html>.
- [6] Imagine Miracle. Gcc 编译流程：从源代码到可执行程序——浅析编译原理, 2022. https://blog.csdn.net/qq_36393978/article/details/124604885.