



南開大學

Nankai University

计算机学院  
编译系统原理实验报告

LLVM IR 编程

姓名：刘荟文 孙璐

学号：2114019 2112060

专业：计算机科学与技术 信息安全

指导教师：王刚老师

2023 年 9 月 17 日

# 目录

<b>1 摘要</b>	<b>2</b>
<b>2 编译环境</b>	<b>2</b>
2.1 GCC 编译器	2
<b>3 语言处理系统</b>	<b>2</b>
3.1 工作流程概述	2
3.2 预处理器	3
3.2.1 输出结果分析	3
3.3 编译器	4
3.3.1 词法分析	5
3.3.2 语法分析	5
3.3.3 语义分析	5
3.3.4 中间代码生成	7
3.3.5 代码优化	7
3.3.6 代码生成	12
3.4 汇编器	13
3.4.1 目标文件	13
3.5 链接器加载器	16
3.5.1 静态链接 & 动态链接	16
<b>4 LLVM LR 编程</b>	<b>17</b>
<b>5 整体优化</b>	<b>20</b>
5.1 pipe 优化	20
5.1.1 优化样例	20
<b>6 总结</b>	<b>22</b>
6.1 任务分工	22
6.2 源码链接	22

## 1 摘要

以 GCC 为研究对象,通过斐波那契数列程序在 Ubuntu 虚拟机上对语言处理系统的完整工作流程进行探索,了解预处理器、编译器、汇编器、链接器的原理及功能,重点了解编译器各阶段的详细过程,包括词法分析、语法分析、语义分析、中间代码生成、代码优化等,比较分析-O1、-O2、-O3、-Os 优化差别。编写包含 SysY 语言特性的 C 语言程序,使用 LLVM IR 语言写出中间形式并执行验证。了解 pipe 优化原理,使用并加以分析。

**关键字:** 语言处理系统 编译器 LLVM

## 2 编译环境

### 2.1 GCC 编译器

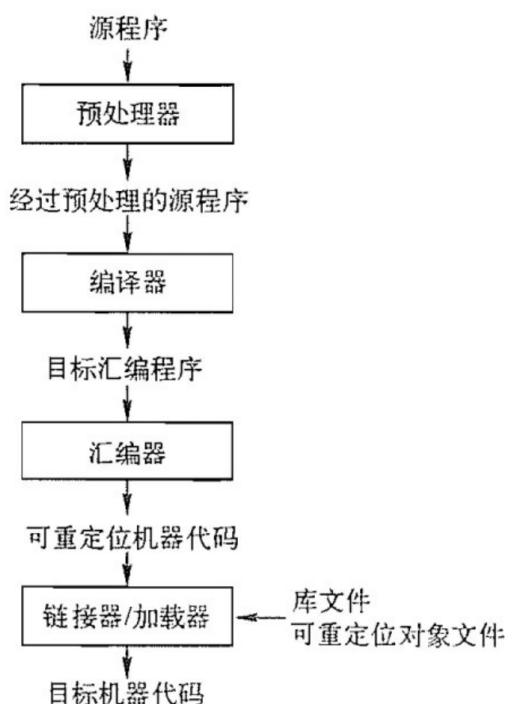
GCC (GNU Compiler Collection) 是一套由 GNU 开发的编译器工具集,最初用于编译 C 语言代码。然而,随着时间的推移和功能的增加, GCC 成为了一个支持多种程序设计语言的编译器集合。

对于程序员来说, GCC 不仅可以将源代码编译成可执行文件,还可以生成中间表示形式(如 GIMPLE),以及汇编代码等。这些中间文件可以用于分析代码、优化算法、进行代码检测和性能调优等目的,帮助程序员更深入地了解编译过程并改进代码质量和性能。

## 3 语言处理系统

### 3.1 工作流程概述

语言处理系统是一种将高级语言源代码转换为计算机可执行代码的工具。通过使用语言处理系统,程序员可选择使用高级语言编写程序,而不需要直接操作底层的机器指令,从而更加高效和方便地开发应用程序,提高软件开发的生产力和品质。以一个 C 程序为例,整体的流程如下图所示:



各阶段的作用如下

- 预处理器

处理源代码中以 `#` 开始的预编译指令，主要的任务包括文件包含、宏替换和条件编译等，例如展开所有宏定义、插入 `#include` 指向的文件等，以获得经过预处理的源程序。

- 编译器

将预处理器处理过的源程序文件翻译成为标准的汇编语言以供计算机阅读。

- 汇编器

将汇编语言指令翻译成机器语言指令，并将汇编语言程序打包成可重定位目标程序。

- 链接器

将可重定位的机器代码和相应的一些目标文件以及库文件链接在一起，形成真正能在机器上运行的目标机器代码

为熟悉实验流程并更深入了解有关知识，小组二人均以阶乘程序和斐波那契数列程序为例进行各阶段任务的尝试。经过实践与讨论分析，最终为了实验报告呈现的效果，决定以斐波拉契数列程序为基准，在不同阶段对程序进行不同操作以探究编译过程。

## 3.2 预处理器

预处理阶段将对源代码进行一些文本替换、宏展开、删除注释、添加行号和文件名标识等操作。主要有以下几个步骤：

- 文件包含 (File Inclusion)

将源代码中的所有 `#include` 指令所引用的文件内容插入到源文件对应位置。

- 宏替换 (Macro Expansion)

根据宏定义 (如 `#define` 等)，将源代码中的所有宏名称替换为其对应的展开代码。

- 条件编译 (Conditional Compilation)

根据条件判断指令 (例如 `#ifdef`、`#ifndef`、`#if` 等)，决定是否编译或排除某些代码块。

执行下列命令对 `fib.cpp` 进行预处理，得到预处理后文件 `fib.i`。

---

```
g++ fib.cpp -E -o fib.i
```

---

### 3.2.1 输出结果分析

- 观察预处理文件 `fib.i`，发现 `fib.i` 文件长度为 32274 行，文件大小远大于源文件，这是对代码中的头文件进行了替代所导致的。`#include<iostream>` 源码长度为 30000 行，在预处理过程中，对 `#include<iostream>` 预编译指令，将被包含的文件内容插入到该指令的位置。值得一提的是，该过程是递归进行的，即直至不存在 `#include` 为止，停止替换包含文件。

```
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "fib.cpp"
# 1 "/usr/include/c++/11/iostream" 1 3
# 36 "/usr/include/c++/11/iostream" 3
# 37 "/usr/include/c++/11/iostream" 3
# 1 "/usr/include/x86_64-linux-gnu/c++/11/bits/c++c
# 278 "/usr/include/x86_64-linux-gnu/c++/11/bits/c++c
```



- 所有的 `#define` 宏定义都会被空行替换，所有使用宏定义的位置的内容都会被其实际所对应的内容替换。
- 所有的注释语句都被替换成空格或者空行，如当前行全为注释语句，会替换成空行。反之若当前行存在非注释的语句，则会将当前的注释替换为空格。需注意预处理器存在尾部空行舍弃问题，即如果代码的尾部存在空行或空格的情况，则末尾的空行和空格会被舍弃。
- `linemakers`

```
# 0 "fib.cpp"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "fib.cpp"
# 1 "/usr/include/c++/11/iostream" 1 3
# 36 "/usr/include/c++/11/iostream" 3
# 37 "/usr/include/c++/11/iostream" 3
# 1 "/usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 1 3
# 278 "/usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
# 278 "/usr/include/x86_64-linux-gnu/c++/11/bits/c++config.h" 3
```

如图部分代码所示，行标格式遵循 `# linenum filename flags`，`flag` 不同值对应含义如下：

- 1：引入一个新文件。
- 2：返回一个新文件（在包含另一个文件之后）。
- 3：以下文本来自系统头文件，因此某些警告应该被阻止。
- 4：以下文本应该被视为包裹在隐式的外部“C”块中。

### 3.3 编译器

编译器接收源代码作为输入，进行一系列词法分析、语法分析、语义分析及优化后生成相对应的汇编代码文件。编译过程可分为以下阶段：

- 词法分析（Lexical Analysis）

编译器首先对源代码进行词法分析，将代码划分成一个个的词法单元 (token)，如关键字、标识符、运算符等。在识别记号的同时，创建符号表，扫描器将标识符放到符号表，将数字和字符串常量存放到文字表等，过滤注释，以备后续编译的继续使用。

- 语法分析 (Syntax Analysis)

编译器进行语法分析，根据语法规则检查代码的结构和组织，确认代码是否符合语言的语法规范。常用的语法分析方法有递归下降分析、LR 分析等。

- 语义分析 (Semantic Analysis)

编译器进行语义分析，检查代码中的语义错误和语义约束，如类型匹配、变量声明与使用等。

- 中间代码生成 (Intermediate Code Generation)

编译器生成一种中间表示形式，将源代码转换为更抽象和与底层硬件无关的形式。中间代码通常是一种高级语言形式，如三地址码、抽象语法树等。

- 优化 (Optimization)

编译器进行代码优化，通过应用各种优化技术改进生成的目标代码的性能和效率。优化可以包括删除冗余代码、循环展开、常量折叠等。

### 3.3.1 词法分析

词法分析阶段对源程序进行线性扫描即可完成。将从左到右一个一个字符地读入源程序，对构成源程序的字符流进行扫描和分解，识别出一个一个的字符。产生的记号一般分为：关键字、标识符、字面常量 (数字、字符串等) 和特殊符号 (+、-、\* 等)。对于 LLVM，可通过以下命令获得 token 序列并保存在 tokens.txt 文件中。

---

```
clang -E -Xclang -dump-tokens tokens.cpp 2>&1
```

---

### 3.3.2 语法分析

语法分析阶段采用了上下文无关文法，将词法分析生成的词法单元分解成各类语法短语，来构建抽象语法树 (Abstract Syntax Tree, 即 AST)。语法分析所依据的是语言的语法规则，通过语法分析确定整个输入串是否构成一个语法上正确的程序。

可以通过 `-fdump-tree-original-raw` flag 获得文本格式的 AST 输出。对于 LLVM，可通过下列命令获得相应的 AST 并保存在 AST.txt 文件中：

---

```
clang -E -Xclang -ast-dump fib.cpp 2>&1
```

---

### 3.3.3 语义分析

语义分析阶段使用语法树和符号表中信息是否和语言定义的语义一致，检查合法性。同时收集类型信息，进行标识符属性类型检查和符号解析等，为代码生成阶段收集类型信息。

语法分析仅仅是完成了对表达式的语法层面的分析，但是语法分析并不能确定这个语句是否真正的有意义。编译器所能分析的语义是静态语义，是指在编译器就可以确定的语义，与之对应的动态语

```

namespace 'namespace' [StartOfLine] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:278:1>
identifier 'std' [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:278:11>
l_brace '{' [StartOfLine] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:279:1>
typedef 'typedef' [StartOfLine] [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:280:3>
long 'long' [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:280:11 <Spelling=<built-in>:130:23>>
unsigned 'unsigned' [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:280:11 <Spelling=<built-in>:130:28>>
int 'int' [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:280:11 <Spelling=<built-in>:130:37>>
identifier 'size_t' [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:280:26>
semi ';' Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:280:32>
typedef 'typedef' [StartOfLine] [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:281:3>
long 'long' [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:281:11 <Spelling=<built-in>:124:26>>
int 'int' [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:281:11 <Spelling=<built-in>:124:31>>
identifier 'ptrdiff_t' [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:281:28>
semi ';' Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:281:37>
typedef 'typedef' [StartOfLine] [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:284:3>
decltype 'decltype' [LeadingSpace] Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:284:11>
l_paren '(' Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:284:19>
nullptr 'nullptr' Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:284:20>

```

图 3.1: tokens.txt

```

TranslationUnitDecl 0xe88a08 <<invalid sloc>> <invalid sloc>
| -TypeDefDecl 0xe89270 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| | -BuiltinType 0xe88fd0 '__int128'
| -TypeDefDecl 0xe892e0 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| | -BuiltinType 0xe88ff0 'unsigned __int128'
| -TypeDefDecl 0xe89658 <<invalid sloc>> <invalid sloc> implicit __NSConstantString
| | '__NSConstantString_tag'
| | -RecordType 0xe893d0 '__NSConstantString_tag'
| | -CXXRecord 0xe89338 '__NSConstantString_tag'
| -TypeDefDecl 0xe896f0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| | -PointerType 0xe896b0 'char *'
| | -BuiltinType 0xe88ab0 'char'
| -TypeDefDecl 0xecdf18 <<invalid sloc>> <invalid sloc> implicit referenced
| | __builtin_va_list '__va_list_tag[1]'
| | -ConstantArrayType 0xecdec0 '__va_list_tag[1]' 1
| | | -RecordType 0xe897e0 '__va_list_tag'
| | | -CXXRecord 0xe89748 '__va_list_tag'
| -NamespaceDecl 0xecdf70 </usr/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/x86_64-linux-gnu/c++/11/bits/c++config.h:278:1, line:286:1> line:278:11 std
| | -TypeDefDecl 0xecdff8 <line:280:3, col:26> col:26 referenced size_t 'unsigned long'
| | | -BuiltinType 0xe88bd0 'unsigned long'
| | -TypeDefDecl 0xece068 <line:281:3, col:28> col:28 referenced ptrdiff_t 'long'
| | | -BuiltinType 0xe88b30 'long'
| | -TypeDefDecl 0xece108 <line:284:3, col:29> col:29 referenced nullptr_t
| | decltvp('nullptr'):'std::nullptr_t'

```

图 3.2: AST.txt

义就是只能在运行期才能确定的语义。经过语义分析阶段过程后，整个语法树的表达式都被标识了类型，如果有些类型需要做隐式转换，语义分析程序会在语法树中插入相应的转换节点。语义分析器也会对符号表中的符号类型做了更新。

大多数编译器并没有把词法分析，语法分析，语义分析严格按阶段进行，一般是同时进行的。

### 3.3.4 中间代码生成

完成上述步骤后，大多数编译器会生成一个明确的低级或类机器语言的中间表示。

使用下列命令可以编译 fib.cpp 并生成中间代码的.dot 文件。

---

```
gcc -fdump-tree-all-graph -fdump-rtl-all-graph -O2 -fno-inline -o fib fib.cpp
```

---

上面的命令-fdump-tree-all-graph 可用于生成所有中间代码的.dot 文件,包括树形中间代码;-fdump-rtl-all-graph 可用于生成所有中间代码的.dot 文件, 包括寄存器传输语言的中间代码。

生成的.dot 文件可以被 graphviz 可视化，可以看到控制流图（CFG），以及各阶段处理中（比如优化、向 IR 转换）CFG 的变化。

LLVM 可以通过下面的命令生成 LLVM IR。

---

```
clang -S -emit-llvm fib.cpp
```

---

控制流图的入口是“ENTRY”基本块，表示程序的入口点。当条件满足时，程序会回到循环的起始点（基本块 <bb 10>），然后继续执行循环体内的操作，直到条件不再满足为止。一旦条件不满足，程序将跳出循环并继续执行后续的操作。

### 3.3.5 代码优化

LLVM 可以通过下面的命令生成每个 pass 后生成的 LLVM IR，以观察差别。

---

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

---

使用下列命令进行.ll 与.bc 互转，以统一文件格式。

---

```
llvm-as fib.ll -o fib.bc      # ll 转换为 bc
llvm-dis fib.bc -o fib.ll     # bc 转换为 ll
```

---

通过下面的命令指定使用某个 pass 以生成 LLVM IR，以特别观察某个 pass 的差别

---

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

---

在本次实验中，尝试使用-O1、-O2、-O3、-Os 进行优化。

- -O0：不做任何优化，这是默认的编译选项。



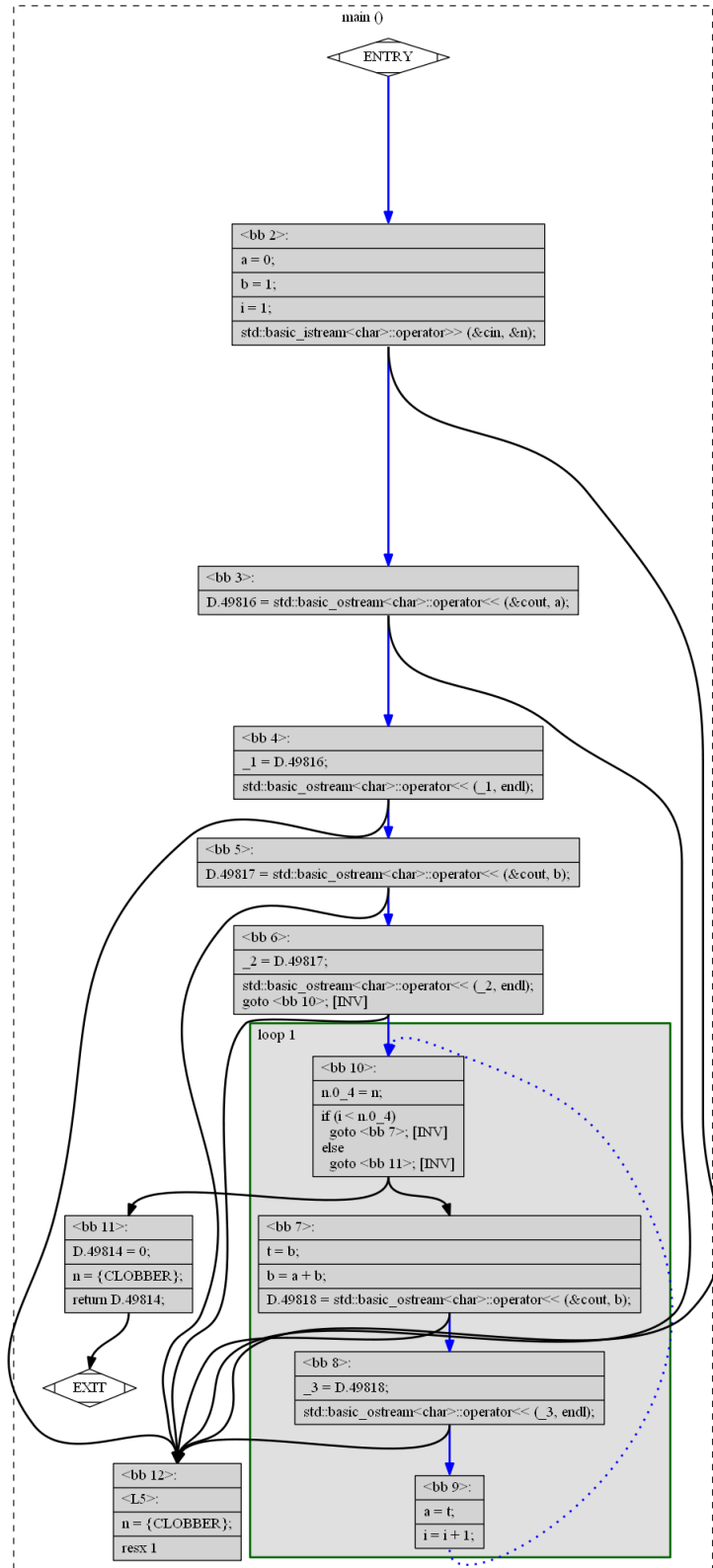


图 3.3: 控制流图

- -O1: 在不影响编译速度的前提下, 尽量采用一些优化算法降低代码大小和可执行代码的运行速度, 主要对代码的分支, 常量以及表达式等进行优化。将常量表达式计算求值, 并用所得到的值来代表表达式, 放入常量表。计算时编译器直接从表中取值而不用访问内存, 从而节省了访问内存的时间。

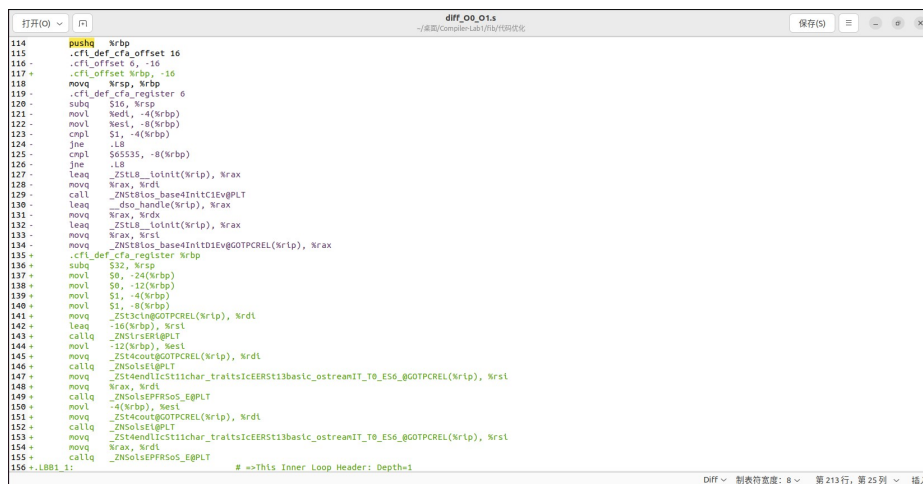
```
.text
.file "fib.cpp"
.section .text.startup,"ax",@progbits
.p2align 4, 0x90 # -- Begin function __cxx_global_var_init
.type __cxx_global_var_init,@function
__cxx_global_var_init: # @__cxx_global_var_init
.cfi_startproc
# %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
movl $_ZStL8__ioinit, %edi
callq _ZNSt8ios_base4InitC1Ev@PLT
movq _ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rdi
movl $_ZStL8__ioinit, %esi
movl $__dso_handle, %edx
callq __cxa_atexit@PLT
popq %rbp
.cfi_def_cfa %rsp, 8
retq
.Lfunc_end0:
.size __cxx_global_var_init, .Lfunc_end0-__cxx_global_var_init
.cfi_endproc
# -- End function

.text
.globl main
.p2align 4, 0x90 # -- Begin function main
.type main,@function
main: # @main
.cfi_startproc
# %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
```

图 3.4: fibO1.s

使用下列指令, 查看-O0 和-O1 优化的区别并存储在 diff\_O0\_O1.s 文件中:

```
diff -u fib.s fibO1.s > diff_O0_O1.s
```



```
diff_O0_O1.s
114 + pushq %rbp
115 + .cfi_def_cfa_offset 16
116 + .cfi_offset 0, -16
117 + .cfi_offset %rbp, -16
118 + movq %rsp, %rbp
119 + .cfi_def_cfa_register 0
120 + subq $16, %rsp
121 + movl %edi, -4(%rbp)
122 + movl %esi, -8(%rbp)
123 + cmpl $1, -4(%rbp)
124 + jne .L8
125 + cmpl $0x5353, -8(%rbp)
126 + jne .L8
127 + leaq _ZStL8__ioinit(%rip), %rax
128 + movq %rax, %rsi
129 + call _ZNSt8ios_base4InitC1Ev@PLT
130 + leaq __dso_handle(%rip), %rax
131 + movq %rax, %rdi
132 + leaq _ZStL8__ioinit(%rip), %rax
133 + movq %rax, %rsi
134 + movq _ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rax
135 + .cfi_def_cfa_register %rbp
136 + subq $32, %rsp
137 + movl $0, -24(%rbp)
138 + movl $0, -12(%rbp)
139 + movl $1, -4(%rbp)
140 + movl $1, -8(%rbp)
141 + movq _ZStL8__ioinit(%rip), %rdi
142 + leaq -16(%rbp), %rsi
143 + callq _ZNSt8ios_base4InitC1Ev@PLT
144 + movl -12(%rbp), %esi
145 + movq _ZStL8__ioinit(%rip), %rdi
146 + callq _ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rdi
147 + movq _ZStL8__ioinit(%rip), %rdi
148 + movq %rax, %rdi
149 + callq _ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rdi
150 + movl -4(%rbp), %esi
151 + movq _ZStL8__ioinit(%rip), %rdi
152 + callq _ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rdi
153 + movq _ZStL8__ioinit(%rip), %rdi
154 + movq %rax, %rdi
155 + callq _ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rdi
156 +.LBB1_1: # ==>This Inner Loop Header: Depth=1
```

图 3.5: diff\_fibO0\_fibO1

O1 在函数切换栈帧的时候, 减少了入栈出栈的过程所用的时间, 也节省了函数栈空间的使用。

- -O2: 会尝试更多的寄存器级的优化以及指令级的优化, 降低代码的大小, 使程序的编译效率大大提升, 从而减少编译的时间, 达到优化的效果。几乎所有的暴力代码开启 O2 优化之后都会大幅提速。

```
.text
.file "fib.cpp"
.section .text.startup,"ax",@progbits
.p2align 4, 0x90 # -- Begin function __cxx_global_var_init
.type __cxx_global_var_init,@function
__cxx_global_var_init: # @__cxx_global_var_init
.cfi_startproc
# %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
movl $_ZStL8__ioinit, %edi
callq __ZNSt8ios_base4InitC1Ev@PLT
movq __ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rdi
movl $_ZStL8__ioinit, %esi
movl $__dso_handle, %edx
popq %rbp
.cfi_def_cfa %rsp, 8
jmp __cxa_atexit@PLT # TAILCALL
.Lfunc_end0:
.size __cxx_global_var_init, .Lfunc_end0-__cxx_global_var_init
.cfi_endproc
# -- End function

.text
.globl main
.p2align 4, 0x90 # -- Begin function main
.type main,@function
main: # @main
.cfi_startproc
# %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
```

图 3.6: fibO2.s

使用下列指令, 查看-O1 和-O2 优化的区别并存储在 diff\_O1\_O2.s 文件中:

```
diff -u fibO1.s fibO2.s > diff_O1_O2.s
```



```
diff_O1_O2.s
~/.桌面/Compiler-Lab1/fib/代码优化
保存(S)

1 --- fibO1.s      2023-09-16 10:50:09.000000000 +0800
2 +++ fibO2.s      2023-09-16 10:50:35.000000000 +0800
3 @@ -16,10 +16,9 @@
4     movq    __ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rdi
5     movl    $_ZStL8__ioinit, %esi
6     movl    $__dso_handle, %edx
7 -     callq  __cxa_atexit@PLT
8     popq    %rbp
9     .cfi_def_cfa %rsp, 8
10    retq
11 +     jmp   __cxa_atexit@PLT # TAILCALL
12 .Lfunc_end0:
13     .size    __cxx_global_var_init, .Lfunc_end0-__cxx_global_var_init
14     .cfi_endproc
15 @@ -99,10 +98,9 @@
16     .cfi_offset %rbp, -16
17     movq    %rsp, %rbp
18     .cfi_def_cfa_register %rbp
19 -     callq  __cxx_global_var_init
20     popq    %rbp
21     .cfi_def_cfa %rsp, 8
22 -     retq
23 +     jmp   __cxa_atexit@PLT # TAILCALL
24 .Lfunc_end2:
25     .size    _GLOBAL__sub_I_fib.cpp, .Lfunc_end2-_GLOBAL__sub_I_fib.cpp
26     .cfi_endproc
```

图 3.7: diff\_fibO1\_fibO2

观察 fibO1.s 与 fibO2.s 文件，发现 O2 级优化会减少循环的展开次数提高程序的编译效率。

- -O3: 该选项在执行-O2 优化选项基础上，采取了很多向量化算法，会提高执行代码的大小，提高代码的并行执行程度，如使用伪寄存器网络，普通函数的内联等，但会降低目标代码的执行时间。

```
.text
.file "fib.cpp"
.section .text.startup,"ax",@progbits
.p2align 4,0x90 # -- Begin function __cxx_global_var_init
.type __cxx_global_var_init,@function
__cxx_global_var_init: # @__cxx_global_var_init
.cfi_startproc
# %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
movl $_ZNSt8ios_base4InitC1Ev@PLT, %edi
callq __ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rdi
movl $_ZNSt8ios_base4InitC1Ev@PLT, %esi
movl $_dso_handle, %edx
popq %rbp
.cfi_def_cfa %rsp, 8
jmp __cxa_atexit@PLT # TAILCALL
.Lfunc_end0:
.size __cxx_global_var_init, .Lfunc_end0-__cxx_global_var_init
.cfi_endproc
# -- End function

.text
.globl main
.p2align 4,0x90 # -- Begin function main
.type main,@function
main: # @main
.cfi_startproc
# %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
```

图 3.8: fibO3.s

- -Os: 用了所有-O2 的优化选项，去掉了那些会导致最终可执行程序增大的优化，尽量缩小代码的大小。

```
.text
.file "fib.cpp"
.section .text.startup,"ax",@progbits
.p2align 4,0x90 # -- Begin function __cxx_global_var_init
.type __cxx_global_var_init,@function
__cxx_global_var_init: # @__cxx_global_var_init
.cfi_startproc
# %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
movl $_ZNSt8ios_base4InitC1Ev@PLT, %edi
callq __ZNSt8ios_base4InitD1Ev@GOTPCREL(%rip), %rdi
movl $_ZNSt8ios_base4InitC1Ev@PLT, %esi
movl $_dso_handle, %edx
popq %rbp
.cfi_def_cfa %rsp, 8
jmp __cxa_atexit@PLT # TAILCALL
.Lfunc_end0:
.size __cxx_global_var_init, .Lfunc_end0-__cxx_global_var_init
.cfi_endproc
# -- End function

.text
.globl main
.p2align 4,0x90 # -- Begin function main
.type main,@function
main: # @main
.cfi_startproc
# %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
```

图 3.9: fibOs.s

针对以上优化结果，发现优化级别越高，生成代码执行效率越高，但对应代码量也会越大，编译

过程所花费时间越长，因此做选择需要进行权衡。

### 3.3.6 代码生成

代码生成阶段以中间表示形式作为输入，将其映射到目标语言。

在目标代码生成阶段，编译器将中间代码翻译成目标指令集：

- 将中间代码的变量映射到寄存器/内存
- 将中间代码的操作映射到指令
- 进行目标指令集相关的优化

```
g++ fib.i -S -o fib_x86.S      # 生成 x86 格式目标代码
arm-linux-gnueabi-gcc fib.i -S -o fib_arm.S      # 生成 arm 格式目标代码
llc fib.ll -o fib_llvm.S      # LLVM 生成目标代码
```

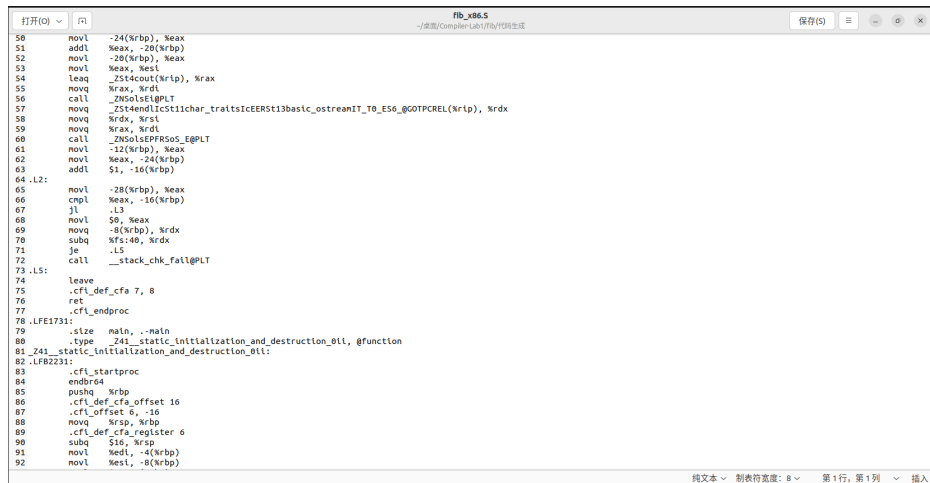


图 3.10: x86 格式目标代码

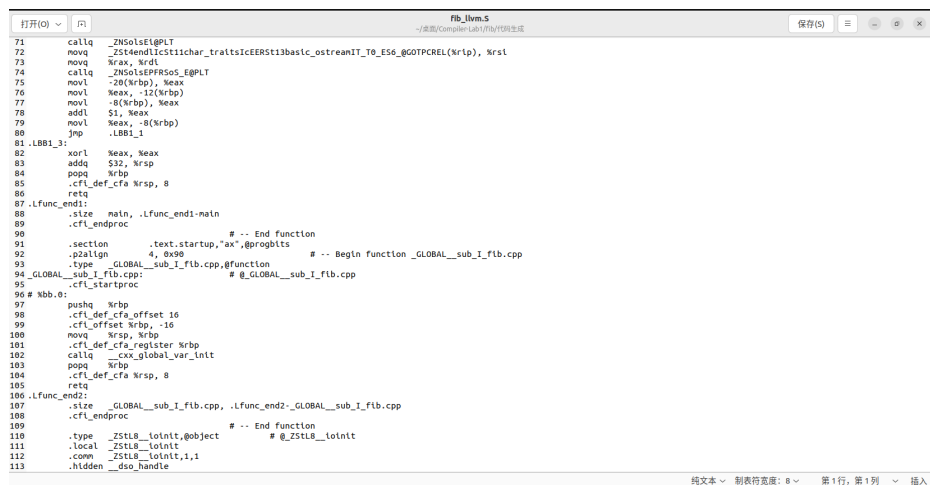


图 3.11: llvm 生成目标代码

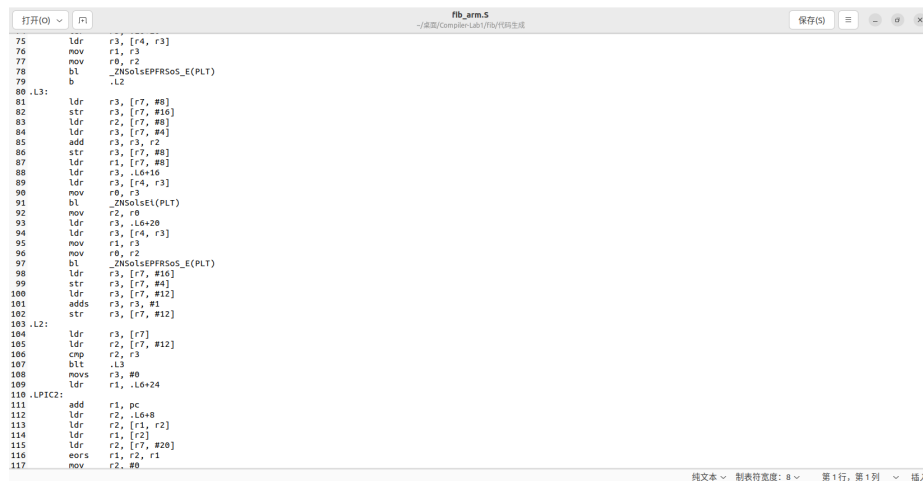


图 3.12: arm 格式目标代码

### 3.4 汇编器

汇编过程实际上把汇编语言程序代码翻译成目标机器码的过程，将符号引用（例如函数和变量名）解析为内存地址或偏移量，每一个汇编语句几乎都对应了一条机器指令，其最终生成的是可重定位的机器代码。

使用下列命令，三种格式汇编完成汇编器的工作。

---

```

g++ fib_x86.S -c -o fib_x86.o
#x86 格式汇编可以直接用 gcc 完成汇编器的工作
arm-linux-gnueabi-gcc fib_arm.S -o fib_arm.o
#arm 格式汇编需要用到交叉编译
llc fib_llvm.bc -filetype=obj -o fib_llvm.o
#LLVM 可以直接使用 llc 命令同时编译和汇编 LLVM bitcode

```

---

使用下列命令，查看反汇编代码得到汇编语言并存储在.txt 文件中

---

```

objdump -d fib_x86.o >fib_x86.txt 2>&1
objdump -d fib_arm.o >fib_arm.txt 2>&1
objdump -d fib_llvm.o >fib_llvm.txt 2>&1

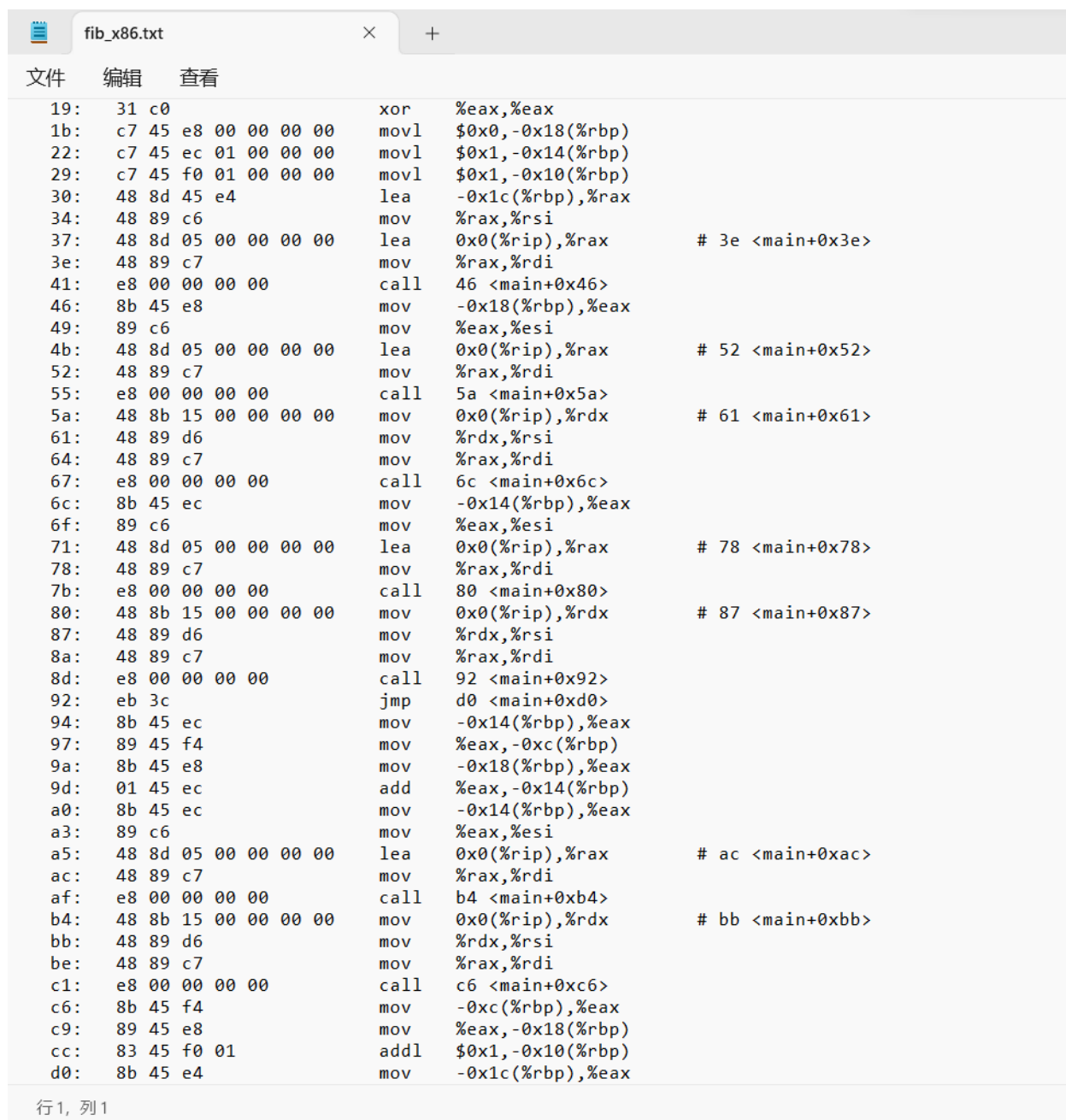
```

---

#### 3.4.1 目标文件

目标文件即源代码编译完成、未进行链接的中间文件。

目标文件中的内容包含有编译后的机器指令代码、数据以及链接所需信息，例如符号表、调试信息、字符串等，并将信息按不同的属性、以节的形式进行存储，有时可称之为段，通常表示一定长度的区域。程序的源代码编译后的机器指令经常放在代码段里，全局变量和局部静态变量会放在数据段.data 里，可执行文件所使用的动态链接库等外来函数与文件的信息存在.idata 中，程序的资源如图标和菜单等存放在.rsrc 中。



```

19: 31 c0          xor    %eax,%eax
1b: c7 45 e8 00 00 00 00  movl   $0x0,-0x18(%rbp)
22: c7 45 ec 01 00 00 00  movl   $0x1,-0x14(%rbp)
29: c7 45 f0 01 00 00 00  movl   $0x1,-0x10(%rbp)
30: 48 8d 45 e4      lea    -0x1c(%rbp),%rax
34: 48 89 c6        mov    %rax,%rsi
37: 48 8d 05 00 00 00 00  lea    0x0(%rip),%rax      # 3e <main+0x3e>
3e: 48 89 c7        mov    %rax,%rdi
41: e8 00 00 00 00    call  46 <main+0x46>
46: 8b 45 e8        mov    -0x18(%rbp),%eax
49: 89 c6          mov    %eax,%esi
4b: 48 8d 05 00 00 00 00  lea    0x0(%rip),%rax      # 52 <main+0x52>
52: 48 89 c7        mov    %rax,%rdi
55: e8 00 00 00 00    call  5a <main+0x5a>
5a: 48 8b 15 00 00 00 00  mov    0x0(%rip),%rdx      # 61 <main+0x61>
61: 48 89 d6        mov    %rdx,%rsi
64: 48 89 c7        mov    %rax,%rdi
67: e8 00 00 00 00    call  6c <main+0x6c>
6c: 8b 45 ec        mov    -0x14(%rbp),%eax
6f: 89 c6          mov    %eax,%esi
71: 48 8d 05 00 00 00 00  lea    0x0(%rip),%rax      # 78 <main+0x78>
78: 48 89 c7        mov    %rax,%rdi
7b: e8 00 00 00 00    call  80 <main+0x80>
80: 48 8b 15 00 00 00 00  mov    0x0(%rip),%rdx      # 87 <main+0x87>
87: 48 89 d6        mov    %rdx,%rsi
8a: 48 89 c7        mov    %rax,%rdi
8d: e8 00 00 00 00    call  92 <main+0x92>
92: eb 3c          jmp    d0 <main+0xd0>
94: 8b 45 ec        mov    -0x14(%rbp),%eax
97: 89 45 f4        mov    %eax,-0xc(%rbp)
9a: 8b 45 e8        mov    -0x18(%rbp),%eax
9d: 01 45 ec        add    %eax,-0x14(%rbp)
a0: 8b 45 ec        mov    -0x14(%rbp),%eax
a3: 89 c6          mov    %eax,%esi
a5: 48 8d 05 00 00 00 00  lea    0x0(%rip),%rax      # ac <main+0xac>
ac: 48 89 c7        mov    %rax,%rdi
af: e8 00 00 00 00    call  b4 <main+0xb4>
b4: 48 8b 15 00 00 00 00  mov    0x0(%rip),%rdx      # bb <main+0xbb>
bb: 48 89 d6        mov    %rdx,%rsi
be: 48 89 c7        mov    %rax,%rdi
c1: e8 00 00 00 00    call  c6 <main+0xc6>
c6: 8b 45 f4        mov    -0xc(%rbp),%eax
c9: 89 45 e8        mov    %eax,-0x18(%rbp)
cc: 83 45 f0 01     addl   $0x1,-0x10(%rbp)
d0: 8b 45 e4        mov    -0x1c(%rbp),%eax

```

行1, 列1

图 3.13: 汇编语言代码





### 3.5 链接器加载器

由汇编程序生成的目标文件不能够直接执行，大型程序经常被分成多个部分进行编译，因此，可重定位的机器代码有必要和其他可重定位的目标文件以及库文件链接到一起，最终形成真正在机器上运行的代码。进而链接器对该机器代码进行执行生成可执行文件。链接器是编译过程的最后一个阶段，它主要执行以下任务：

- 链接库文件

库文件是一组预编译好的可重用代码，包含了常见的函数、过程或类等。链接器将库文件中的函数和变量链接到可执行文件中。它只会链接所需的库文件的部分，以减小可执行文件的大小。

- 符号表处理

合并所有目标文件和库文件的符号表，以便在整个程序中查找符号的定义和引用。这个合并的符号表将用于解析未解析的符号引用和地址重定位。

- 符号解析

链接器通过符号表（Symbol Table）对目标文件中引用和定义的符号进行解析。符号可以是函数名、变量名、常量等，在不同的目标文件中可能会出现重名的符号。链接器通过符号解析将引用符号与定义符号进行匹配，建立符号之间的关联关系。

- 地址重定位

当多个目标文件被合并成一个可执行文件时，各个目标文件的代码和数据段可能会有重叠，需要进行地址调整。计算每个符号的最终地址，将不同目标文件中的相对地址转换为绝对地址或可执行文件内部的相对地址，更新目标文件中的地址引用，确保各个目标文件中的引用正确指向它们的定义，程序在运行时能够正确访问内存。

- 生成可执行文件

将所有目标文件和库文件链接在一起，生成可执行文件。

链接器的工作在程序开发的不同阶段扮演着重要角色，它将多个独立的模块集成为一个可执行文件，使得程序的结构更加清晰、模块化，并提供了代码重用的机制。

使用如下命令可生成 fib 可执行文件

---

```
g++ fib.o -o fib
```

---

在终端输入下列命令输入样例 n = 5 验证 fib 文件

---

```
./fib
```

---

#### 3.5.1 静态链接 & 动态链接

- 静态链接（Static Linking）

在编译链接过程中，将所有目标文件和库文件的代码和数据合并到最终的可执行文件中。在静态链接的方式下，目标文件会被直接复制到可执行文件中，形成一个独立完整的执行映像。当程序运行时，操作系统加载可执行文件并将其完整地装入内存中执行。静态链接的特点包括：

```
judy@judy-virtual-machine: ~/桌面/Compiler-Lab1/fib
judy@judy-virtual-machine: ~/桌面/Compiler-Lab1/fib$ ./fib
5
0
1
1
2
3
5
judy@judy-virtual-machine: ~/桌面/Compiler-Lab1/fib$
```

图 3.16: 运行结果

- 独立性：生成的可执行文件包含了所有需要的代码和数据，可以在没有其他依赖的情况下独立运行，无需额外的库文件支持。
  - 执行速度：由于所有的代码和数据都被静态链接到可执行文件中，程序的执行速度通常比较快。
  - 一致性：由于使用了静态链接，程序的行为在不同的环境下保持一致，不受外部库版本的影响。
- 动态链接（Dynamic Linking）

在程序运行时，由操作系统根据需要在内存中加载所需的共享库文件（也称为动态链接库），并将其链接到可执行文件中。动态链接的特点包括：

- 共享性：多个可执行文件可以共享同一份库文件的代码和数据，减少了磁盘空间的占用和内存的消耗。
- 灵活性：动态链接库可以在不重新编译和链接可执行文件的情况下进行更新，提供了更灵活的软件更新和维护方式。
- 节省资源：由于共享库是按需加载的，只有在运行时需要时才会被加载到内存中，节省了内存资源。
- 库版本管理：动态链接库支持多个版本存在，并提供了版本管理机制，可以根据需要选择特定的库版本。

选择静态链接还是动态链接取决于具体的需求和环境。静态链接适用于需要独立运行、不依赖外部库以及对执行速度有较高要求的场景。而动态链接则适用于需要共享代码、提供灵活的更新和维护方式以及节省内存等资源的场景。

## 4 LLVM LR 编程

编写一个包含 SysY 语言特性的 C 语言程序，并用 LLVM IR 语言写出中间形式。下面编写了一个计算自然数数字和的 C 语言程序

Listing 1: 计算自然数数字和

```
1 #define _CRT_SECURE_NO_WARNINGS // 禁用警告
2 #include <stdio.h>
```

```

3 // 递归函数来计算累加和
4 int sum(int n)
5 {
6     if (n == 1)
7     {
8         return 1; // 基本情况: 当n为1时, 返回1
9     }
10    else
11    {
12        return n + sum(n - 1); // 递归调用, 将n与1到n-1的累加和相加
13    }
14 }
15 int main()
16 {
17     int number;
18     // 从用户输入获取number的值
19     printf("请输入一个正整数: ");
20     scanf_s("%d", &number);
21     if (number < 1)
22     {
23         printf("请输入一个正整数。\\n");
24         return 1; // 如果用户输入的不是正整数, 程序退出并返回错误代码
25     }
26     printf("从1到%d的整数的累加和是 %d\\n", number, sum(number));
27     return 0;
28 }
29
30 }

```

下面是对应的 LLVM IR 程序

Listing 2: 计算自然数数字和

```

1
2 ; 定义递归求和函数 sum
3 define dso_local @sum(i32 @noundef %0) #0 {
4     %2 = alloca i32, align 4
5     %3 = alloca i32, align 4
6     store i32 %0, i32* %3, align 4
7     %4 = load i32, i32* %3, align 4
8     %5 = icmp eq i32 %4, 1
9     br i1 %5, label %6, label %7
10
11 6: ; 如果输入等于 1, 直接返回 1
12     store i32 1, i32* %2, align 4
13     br label %13
14
15 7: ; 如果输入不等于 1, 执行递归调用和加法
16     %8 = load i32, i32* %3, align 4
17     %9 = load i32, i32* %3, align 4

```

```

18  %10 = sub nsw i32 %9, 1
19  %11 = call i32 @sum(i32 noundef %10)
20  %12 = add nsw i32 %8, %11
21  store i32 %12, i32* %2, align 4
22  br label %13
23
24 13: ; 返回计算结果
25  %14 = load i32, i32* %2, align 4
26  ret i32 %14
27 }
28
29 ; 主函数 main
30 define dso_local i32 @main() #0 {
31  %1 = alloca i32, align 4
32  %2 = alloca i32, align 4
33  store i32 0, i32* %1, align 4
34  %3 = call i32 (@i8*, ...) @printf(i8* noundef getelementptr inbounds ([19 x i8], [19
    x i8]* @.str, i64 0, i64 0))
35  %4 = call i32 (@i8*, i32*, ...) @bitcast (i32 (...) @scanf_s to i32 (@i8*, i32*,
    ...)) (i8* noundef getelementptr inbounds ([3 x i8], [3 x i8]* @.str.1, i64 0,
    i64 0), i32* noundef %2)
36  %5 = load i32, i32* %2, align 4
37  %6 = icmp slt i32 %5, 1
38  br i1 %6, label %7, label %9
39
40 7: ; 如果输入小于 1, 输出提示并返回 1
41  %8 = call i32 (@i8*, ...) @printf(i8* noundef getelementptr inbounds ([20 x i8], [20
    x i8]* @.str.2, i64 0, i64 0))
42  store i32 1, i32* %1, align 4
43  br label %14
44
45 9: ; 如果输入大于等于 1, 执行递归求和并输出结果
46  %10 = load i32, i32* %2, align 4
47  %11 = load i32, i32* %2, align 4
48  %12 = call i32 @sum(i32 noundef %11)
49  %13 = call i32 (@i8*, ...) @printf(i8* noundef getelementptr inbounds ([28 x i8],
    [28 x i8]* @.str.3, i64 0, i64 0), i32 noundef %10, i32 noundef %12)
50  store i32 0, i32* %1, align 4
51  br label %14
52
53 14: ; 返回 0
54  %15 = load i32, i32* %1, align 4
55  ret i32 %15
56 }
57
58 ; 声明 printf 和 scanf_s 函数
59 declare i32 @printf(i8* noundef, ...) #1
60 declare i32 @scanf_s(...) #1

```

## 5 整体优化

### 5.1 pipe 优化

pipe 优化 (Compile-time Pipe Optimization) 是一种编译器优化技术, 旨在提高代码的执行效率和性能。它通过并行化编译过程中的不同阶段, 将多个优化步骤连接起来, 实现优化的连续流程, 以减少编译时间并提高生成的机器代码质量。

pipe 优化方式采用管道连接, 将不同的优化子阶段连接在一起, 构成一个流水线式的优化管道。每个优化子阶段会接收前一个阶段的输出结果, 进行特定的优化处理, 并将其传递到下一个阶段。这样可以直接在内存中传递数据, 避免了中间文件的存取, 减少 I/O 操作, 大大提高了编译的效率。

编译时 pipe 优化需要处理数据依赖关系, 确保每个子阶段都能获得正确的输入数据。同时, 为了提高优化过程的效率, 编译器还需要进行并发控制, 使得不同的优化子阶段可以在多个处理单元或线程上并行执行。

需要注意的是, 编译时 pipe 优化并非适用于所有情况, 有时候可能会牺牲一定的编译时间来换取更高的优化质量。因此, 在实际应用中, 编译器开发者需要根据具体的编译目标、硬件平台、编译时间要求等因素来选择适当的优化策略和配置。

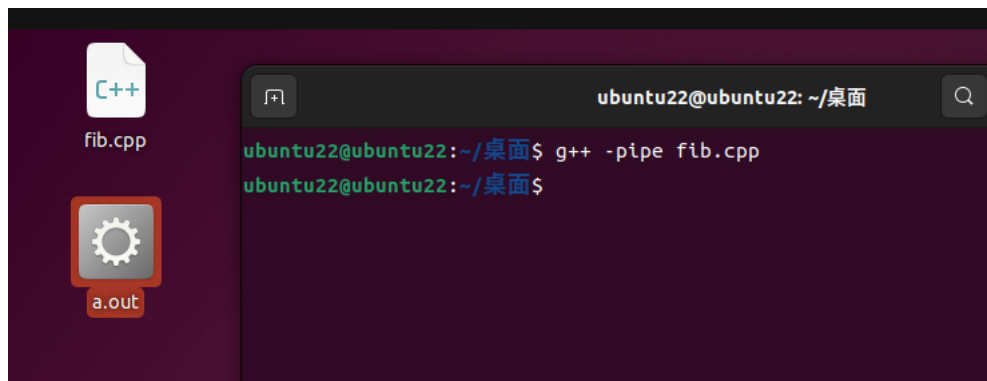
#### 5.1.1 优化样例

调用 pipe 命令生成可执行文件

---

```
g++ -pipe fib.cpp
```

---



使用-ftime-report 来查看调用 pipe 优化和未调用优化的编译器耗时情况:

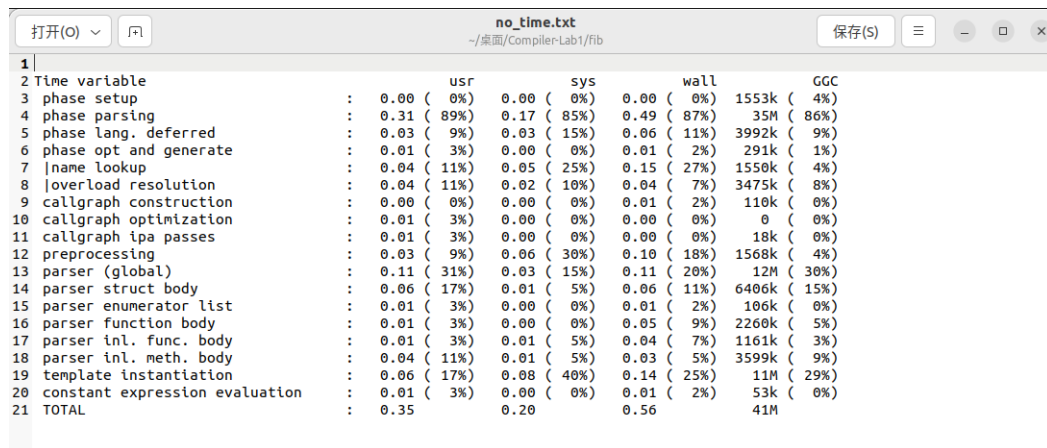
---

```
g++ -ftime-report fib.cpp >no_time.txt 2>&1
g++ -ftime-report -pipe fib.cpp > >pipe_time.txt 2>&1
```

---

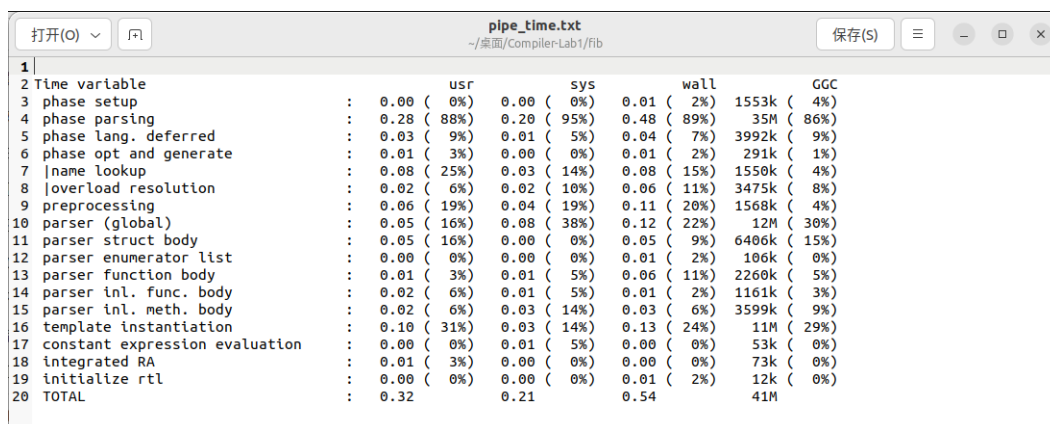
不调用 pipe 优化生成可执行文件 b.out 后, 用 diff 比较优化产生文件是否存在差异

结果无任何输出, 证明两文件完全相同, 即是否使用 pipe 优化, 只影响编译过程中的耗时, 并不对最终的编译结果产生影响, 因为其本质只改变了编译过程中中间文件的读写和存储方式, 降低 I/O 操作的耗时, 提升编译效率。



	usr	sys	wall	GC
1				
2 Time variable				
3 phase setup	0.00 ( 0%)	0.00 ( 0%)	0.00 ( 0%)	1553k ( 4%)
4 phase parsing	0.31 ( 89%)	0.17 ( 85%)	0.49 ( 87%)	35M ( 86%)
5 phase lang. deferred	0.03 ( 9%)	0.03 ( 15%)	0.06 ( 11%)	3992k ( 9%)
6 phase opt and generate	0.01 ( 3%)	0.00 ( 0%)	0.01 ( 2%)	291k ( 1%)
7 lname lookup	0.04 ( 11%)	0.05 ( 25%)	0.15 ( 27%)	1550k ( 4%)
8 joverload resolution	0.04 ( 11%)	0.02 ( 10%)	0.04 ( 7%)	3475k ( 8%)
9 callgraph construction	0.00 ( 0%)	0.00 ( 0%)	0.01 ( 2%)	110k ( 0%)
10 callgraph optimization	0.01 ( 3%)	0.00 ( 0%)	0.00 ( 0%)	0 ( 0%)
11 callgraph ipa passes	0.01 ( 3%)	0.00 ( 0%)	0.00 ( 0%)	18k ( 0%)
12 preprocessing	0.03 ( 9%)	0.06 ( 30%)	0.10 ( 18%)	1568k ( 4%)
13 parser (global)	0.11 ( 31%)	0.03 ( 15%)	0.11 ( 20%)	12M ( 30%)
14 parser struct body	0.06 ( 17%)	0.01 ( 5%)	0.06 ( 11%)	6406k ( 15%)
15 parser enumerator list	0.01 ( 3%)	0.00 ( 0%)	0.01 ( 2%)	106k ( 0%)
16 parser function body	0.01 ( 3%)	0.00 ( 0%)	0.05 ( 9%)	2260k ( 5%)
17 parser inl. func. body	0.01 ( 3%)	0.01 ( 5%)	0.04 ( 7%)	1161k ( 3%)
18 parser inl. meth. body	0.04 ( 11%)	0.01 ( 5%)	0.03 ( 5%)	3599k ( 9%)
19 template instantiation	0.06 ( 17%)	0.08 ( 40%)	0.14 ( 25%)	11M ( 29%)
20 constant expression evaluation	0.01 ( 3%)	0.00 ( 0%)	0.01 ( 2%)	53k ( 0%)
21 TOTAL	0.35	0.20	0.56	41M

图 5.17: 未使用 pipe 优化



	usr	sys	wall	GC
1				
2 Time variable				
3 phase setup	0.00 ( 0%)	0.00 ( 0%)	0.01 ( 2%)	1553k ( 4%)
4 phase parsing	0.28 ( 88%)	0.20 ( 95%)	0.48 ( 89%)	35M ( 86%)
5 phase lang. deferred	0.03 ( 9%)	0.01 ( 5%)	0.04 ( 7%)	3992k ( 9%)
6 phase opt and generate	0.01 ( 3%)	0.00 ( 0%)	0.01 ( 2%)	291k ( 1%)
7 lname lookup	0.08 ( 25%)	0.03 ( 14%)	0.08 ( 15%)	1550k ( 4%)
8 joverload resolution	0.02 ( 6%)	0.02 ( 10%)	0.06 ( 11%)	3475k ( 8%)
9 preprocessing	0.06 ( 19%)	0.04 ( 19%)	0.11 ( 20%)	1568k ( 4%)
10 parser (global)	0.05 ( 16%)	0.08 ( 38%)	0.12 ( 22%)	12M ( 30%)
11 parser struct body	0.05 ( 16%)	0.00 ( 0%)	0.05 ( 9%)	6406k ( 15%)
12 parser enumerator list	0.00 ( 0%)	0.00 ( 0%)	0.01 ( 2%)	106k ( 0%)
13 parser function body	0.01 ( 3%)	0.01 ( 5%)	0.06 ( 11%)	2260k ( 5%)
14 parser inl. func. body	0.02 ( 6%)	0.01 ( 5%)	0.01 ( 2%)	1161k ( 3%)
15 parser inl. meth. body	0.02 ( 6%)	0.03 ( 14%)	0.03 ( 6%)	3599k ( 9%)
16 template instantiation	0.10 ( 31%)	0.03 ( 14%)	0.13 ( 24%)	11M ( 29%)
17 constant expression evaluation	0.00 ( 0%)	0.01 ( 5%)	0.00 ( 0%)	53k ( 0%)
18 integrated RA	0.01 ( 3%)	0.00 ( 0%)	0.00 ( 0%)	73k ( 0%)
19 initialize rtl	0.00 ( 0%)	0.00 ( 0%)	0.01 ( 2%)	12k ( 0%)
20 TOTAL	0.32	0.21	0.54	41M

图 5.18: 使用 pipe 优化

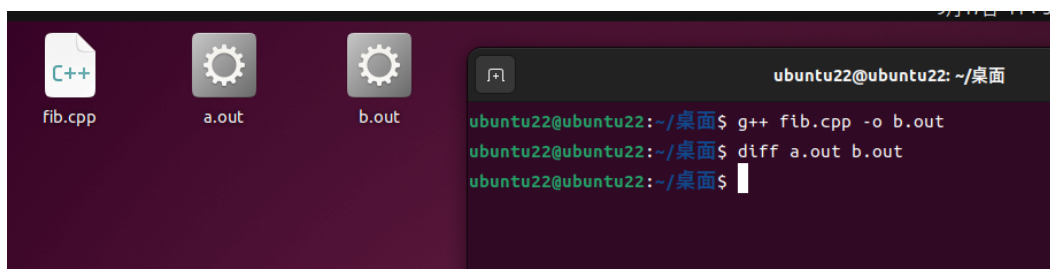


图 5.19: 使用 diff 比较优化产生文件是否存在差异

## 6 总结

经过本次实验，小组成员均全程参与，既有独立思考、又有合作讨论，成功解决实验过程中遇到的各种问题，从而对语言处理系统的完整工作过程有了更深入的了解，掌握了编译器相关的重要知识，对未来在编译原理课程方面进一步的学习奠定下坚实的基础。

### 6.1 任务分工

- 程序的测试及编写、测试结果的分析讨论、资料查询及实验报告撰写均由两人共同合作完成。

### 6.2 源码链接

<https://github.com/jtysxtm/byxtyl>

## 参考文献

- [1] <https://www.docs4dev.com/docs/gcc/6.4.0/preprocessor-output.html>
- [2] <https://gcc.gnu.org/onlinedocs/>
- [3] <https://buaa-se-compiling.github.io/miniSysY-tutorial/>
- [4] <https://github.com/llvm/llvm-project/>
- [5] <https://llvm.org/docs/UserGuides.html>