南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理实验报告

了解编译器 &LLVM IR 编程 & 汇编编程实验报告

禹相祐

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

摘要

此次预备实验由禹相祐和查科言二人共同完成，分工为：禹相祐完成有关阶乘的汇编代码实现，查科言完成有关阶乘的 LLVM IR 的实现和 SysY 源程序。这篇报告主要包括七个部分，第一部分是此次实验的简介，第二部分是任务一——了解编译器的相关内容，第三部分是撰写 SysY 源程序，第四部分是任务二——完成阶乘的汇编代码实现以及 LLVM IR 实现，第五部分编写 SysY 的 ARM64 汇编代码，第六部分是 C 语言汇编，第七部分是此次实验的总结部分。

关键字：编译，LLVM IR 编程，汇编编程

目录

一、 实验简介 **1**

（一） 主要任务 . . . 1
（二） 实验目的 . . . 1
（三） 实验分工 . . . 1
（四） 实验环境 . . . 1

二、 实验一：了解你的编译器 **2**

（一） 编译全过程 . . . 2
（二） 预处理阶段 . . . 3
（三） 编译阶段 . . . 6
1. 词法分析 . . . 6
2. 语法分析 . . . 10
3. 语义分析 . . . 15
4. 中间代码生成 . . . 15
5. 代码优化 . . . 19
6. 代码生成 . . . 24
（四） 汇编阶段 . . . 24
（五） 链接 & 加载阶段 . . . 26

三、 SysY 程序设计 **27**

（一） 基础语法 . . . 28
（二） 与 C 的差别 . . . 28
（三） 程序设计 . . . 28

四、 任务二：LLVM IR 编程 **29**

（一） 任务介绍 . . . 29
（二） LLVM IR 的特性和语法 . . . 29
1. 核心特性 . . . 30
2. LLVM IR 语法 . . . 30
（三） 编写 LLVM IR 代码 . . . 32
1. 全局声明与常量 . . . 32
2. 自定义函数声明 . . . 32
3. main 函数实现 . . . 34
（四） 验证结果 . . . 35

五、 任务三：汇编编程 **37**
（一） 任务介绍 37
（二） Arm 架构简要介绍 37
1. 核心特点 37
2. 关键组件 37
（三） 编写汇编代码 38
1. 全局声明与初始化 38
2. 主函数 main 38
3. 自定义函数 40
（四） 验证结果 41

六、 C 语言编程 **43**

七、 总结 **49**

6010NKU

一、 实验简介

(一) 主要任务

1. 以 GCC(或 LLVM/Clang 等你常用的、熟悉的编译工具) 为研究对象, 更深入地探究语言处理系统的完整工作过程: 预处理器做了什么? 编译器做了什么 (包括更细致的编译器各阶段的功能) 做了什么? 汇编器做了什么? 链接器做了什么?

2. 熟悉 LLVM IR 中间语言和汇编语言: 设计 SysY 示例程序涵盖编译器要支持的语言特性 (各种数值运算, 赋值、条件分支、循环等语句函数, 以及其他进阶特性), 编写 LLVM IR 程序以及 ARM 或 RISC-V 汇编程序, 与 SysY 源程序等价、且能链接 SysY 运行库, 用 LLVM/Clang、汇编器编译成成目标程序, 验证运行结果是否正确

(二) 实验目的

以下是我从上述任务总结出来的实验目的:

1. 理解编译全过程：通过观察预处理、编译、汇编、链接各阶段的产物与作用，建立从源代码到可执行文件的整体认知。

2. 掌握编译器工作机制：学会使用编译器不同阶段选项（如 `-E`、`-S`、`-c` 等）拆解流程并分析生成文件，理解关键步骤与常见诊断信息。

3. 熟悉 LLVM IR 编程：能够编写与阅读基础 LLVM IR，理解类型、基本块与控制流等核心概念，并用 `clang`/`opt`/`llc` 生成 IR、进行简单优化并降级到汇编。

4. 熟练编写 ARM 汇编：掌握常用指令与寄存器、栈与调用约定，能将简单算法从 C/IR 映射到 ARM 汇编并进行基础调试，理解与硬件交互的基本原理。

(三) 实验分工

本组选择了 **ARM** 框架，成员为**禹相祐（2312900）**以及**查科言（2312189）**。任务一各自独立完成，任务二具体分工如下：

禹相祐（2312900）负责完成汇编编程的任务部分；

查科言（2312189）负责完成 LLVM IR 编程的任务部分和撰写 SysY 源程序。

最后总结部分由二人共同完成。

(四) 实验环境

表 1: 实验环境

OS 系统	Ubuntu 20.04
OS 类型	Linux(64 位)

二、 实验一：了解你的编译器

(一) 编译全过程

由课上内容，我们可以知道完整的编译过程大概可以分为四部分：

1. 预处理
2. 编译
3. 汇编
4. 链接 & 加载

大致流程图如图1所示：

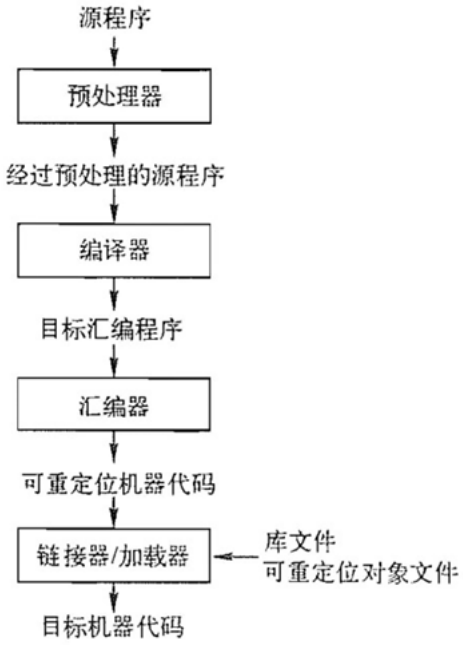


图 1: 完整编译过程

这四个阶段的大致作用如下（下面纯属本人理解，可能不太严谨）：

1. **预处理**: 处理 #include 和 #define 等宏、条件编译，把需要的头文件合并进来，得到“展开后”的源代码，方便后续处理。
2. **编译**: 检查语法和类型，把人能读的代码翻译成更接近机器的表示，通常会生成汇编代码或中间代码，并在这一步报出大多数错误。
3. **汇编**: 把汇编指令变成机器能执行的二进制指令，产物是目标文件（如 `.o/.obj`），此时一些地址还未最终确定。
4. **链接 & 加载**: 把多个目标文件和库“拼起来”，补齐外部函数与变量的位置，生成可执行程序；运行时，操作系统把程序装入内存，准备好起始地址和栈空间，然后开始执行。

那接下来就让我们开始通过具体的代码以及输出文件来一个个部分的进行解释。

(二) 预处理阶段

下面先给出阶乘的 main.cpp 文件的代码：

```cpp
#include<iostream>
using namespace std;

int main()
{
    int i,n,f;
    cin >>n;

    i = 2;
    f = 1;
    while(i<=n)
    {
        f=f*i;
        i=i+1;
    }
    cout << f << endl;

    return 0;
}
```

阶乘 main.cpp 代码

这是一个大一学生看了也都会的代码，先是获取一个输入 n，然后再进入一个 while 循环，最后输出的阶乘结果会存在变量 f 内并输出出来。

那我们就接着进行下一步了：

通过命令 **gcc main.cpp -E -o main.i** 进行预处理得到文件 main.i：

在这指导书也有提及——**-E** 表示 gcc 只进行预处理的过程，而添加**-o** 表示改变 gcc 的输出名，如果命令修改为 **gcc main.cpp -E -o whatever.i**，那我们就会得到一个 whatever.i 文件，其内容是 main.cpp 的预处理结果。

输入命令进行预处理后结果如图：

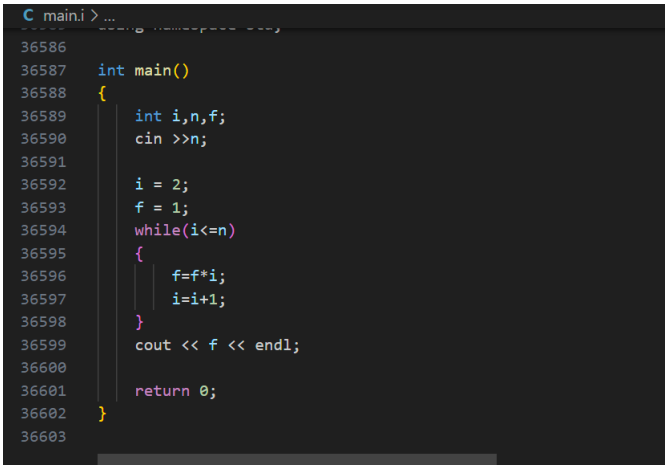


图 2: 预处理后效果图

我们会很惊喜的发现，从原先 main.cpp 的约 20 行变为了如今的约 36600 行。这主要是因为预处理阶段，会将代码原先包含的头文件 **iostream** 内的文件内容直接插入代码中，并置于代码的前面，所以前面的大约 36500 多行都是 iostream 内所包含的内容，这就体现出预处理器的第一个作用——**文件的包含处理**。

我们再将 main.cpp 修改一下，变为如下，我们称其为 **less.cpp**:

```cpp
// less.cpp 去掉include语句 加上define 以及 注释
#define whatever 10

using namespace std;

// 这是很多注释
// 注释注释注释注释
// 依然注释注释注释
int main()
{
    whatever;
    int i,n,f;
    cin >>n;

    i = 2;
    f = 1;
    while(i<=n)
    {
        f=f*i;
        i=i+1;
    }
    cout << f << endl;

    return 0;
}
```

修改后 less.cpp

输入命令：**gcc less.cpp -E -o less.i** 得到：

```
36584
36585
36586   # 5 "less.cpp"
36587   using namespace std;
36588
36589
36590
36591
36592   int main()
36593   {
36594       10;
36595       int i,n,f;
36596       cin >>n;
36597
36598       i = 2;
36599       f = 1;
36600       while(i<=n)
36601       {
36602           f=f*i;
36603           i=i+1;
36604       }
36605       cout << f << endl;
36606
36607       return 0;
36608   }
36609
```

图 3: less.cpp 预处理后效果图

我们会发现，写的注释全部都消失了，且我们通过 **define** 语句定义的变量 **whatever** 从代码中消失并且直接变为了 10，并且明明删除了语句 **#include<iostream>**，却依然有 36000 多行，说明预处理阶段自动帮我们添加了这一句，并依然将文件的内容插入到了我们的代码前方，这就是预处理的第二、第三以及第四个功能，分别是：

- **去除注释**：就像 less.cpp 内写到的那些注释全都被删去了一样，预处理阶段会自动将所有的注释删去；
- **宏定义替换**：就像 less.cpp 内写到的 define whatever 10，预处理阶段会把所有的变量 whatever 替换为 10，将所有的宏定义都用对应的内容替换掉原先的内容；
- **文件包含处理 & 生成预处理后文件**：此处指的是预处理阶段会带有一点的“代码修正”的功能，自动补充所需要的头文件，并在处理后将头文件的内容插入到代码前面，**生成一个预处理后的文件**；

总结一下，**预处理阶段主要就是实现以下四点功能**：

1. **去除注释**：预处理阶段会自动识别所有的注释内容，并且将所有的注释内容（单行注释 + 多行注释）全部删去；
2. **宏定义替换**：预处理阶段会自动识别所有的宏定义语句，并且将所有的宏定义都用对应的内容替换掉原先的内容；
3. **文件包含处理**：预处理阶段会识别 #include 指令，并且将文件的内容（通常是头文件）插入到代码前面;
4. **生成预处理后的代码**：完成以上步骤后，预处理器会生成预处理后的代码，供后续阶段使用。

(三) 编译阶段

编译阶段是此次实验的重点部分，从课内知识我们可以知道，编译阶段又能分为六个部分，如下图4所示：

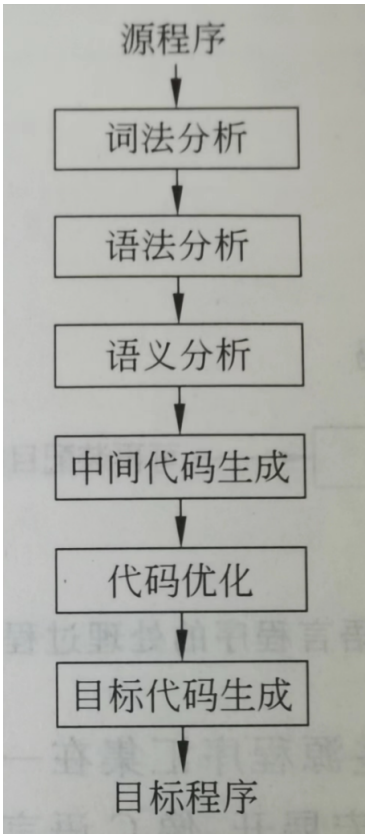


图 4: 编译阶段六个阶段全过程

那我们就一个一个部分来进行实验，看看究竟是在干什么：

1. 词法分析

由课内知识，可以知道词法分析的主要作用是将源程序转化为单词的序列，以便于后续更好地进行词法分析。

然后我们还是对 **main.cpp** 输入以下命令：

clang -E -Xclang -dump-tokens main.cpp, 运行结果如下：

```
r_brace '}'      [StartOfLine]  Loc=</usr/bin/../lib/gcc/x86_64-linux-gnu/13/../../../../include/c++/13/i
am:86:1>
using 'using'    [StartOfLine]  Loc=<main.cpp:3:1>
namespace 'namespace'    [LeadingSpace] Loc=<main.cpp:3:7>
identifier 'std'         [LeadingSpace] Loc=<main.cpp:3:17>
semi ';'                Loc=<main.cpp:3:20>
int 'int'        [StartOfLine]  Loc=<main.cpp:5:1>
identifier 'main'        [LeadingSpace] Loc=<main.cpp:5:5>
l_paren '('             Loc=<main.cpp:5:9>
r_paren ')'             Loc=<main.cpp:5:10>
l_brace '{'      [StartOfLine]  Loc=<main.cpp:6:1>
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<main.cpp:7:5>
identifier 'i'   [LeadingSpace] Loc=<main.cpp:7:9>
comma ','               Loc=<main.cpp:7:10>
identifier 'n'          Loc=<main.cpp:7:11>
comma ','               Loc=<main.cpp:7:12>
identifier 'f'          Loc=<main.cpp:7:13>
semi ';'                Loc=<main.cpp:7:14>
identifier 'cin'         [StartOfLine] [LeadingSpace]   Loc=<main.cpp:8:5>
greatergreater '>>'      [LeadingSpace] Loc=<main.cpp:8:9>
identifier 'n'          Loc=<main.cpp:8:11>
semi ';'                Loc=<main.cpp:8:12>
identifier 'i'   [StartOfLine] [LeadingSpace]   Loc=<main.cpp:10:5>
equal '='        [LeadingSpace] Loc=<main.cpp:10:7>
numeric_constant '2'     [LeadingSpace] Loc=<main.cpp:10:9>
semi ';'                Loc=<main.cpp:10:10>
identifier 'f'   [StartOfLine] [LeadingSpace]   Loc=<main.cpp:11:5>
equal '='        [LeadingSpace] Loc=<main.cpp:11:7>
```

图 5: 词法分析运行结果

我们对以上输出结果尝试分析一下:

```
// 对应using namespace std;
using 'using' [StartOfLine] Loc=<main.cpp:3:1>
namespace 'namespace' [LeadingSpace] Loc=<main.cpp:3:7>
identifier 'std' [LeadingSpace] Loc=<main.cpp:3:17>
semi ';' Loc=<main.cpp:3:20>
// 对应 int main()
int 'int' [StartOfLine] Loc=<main.cpp:5:1>
identifier 'main' [LeadingSpace] Loc=<main.cpp:5:5>
l_paren '(' Loc=<main.cpp:5:9>
r_paren ')' Loc=<main.cpp:5:10>
```

词法分析结果

我们暂时忽略掉 **#include<iostream>** 的词法分析部分，直接看到从 **using namespace std;** 开始的部分,很明显能看出,以上的词法分析输出的结果只是对应两句,即 **using namespace std;** 以及 **int main()**。

可以从上述输出部分看到:

1. **using 'using' [StartOfLine] Loc=<main.cpp:3:1>**

   第一个蓝色的 using 表示这是关键字'using'，第二个 using 就是词法分析拆解出来的第一个 token,【StartOfLine】表明 using 这个词是这一行的第一个 token,然后 Loc=<main.cpp:3.1> 就表示 using 这个词开始的位置是第三行的第一列;

2. **namespace 'namespace' [LeadingSpace] Loc=<main.cpp:3:7>**

   第二行开始的两个 namespace 与第一行的两个 using 同理，然后 [LeadingSpace] 表示 token 'namespace' 前面有一个空格，然后 Loc=<main.cpp:3.7>。这个是因为计算位置的时候是一个一个字母进行计算，我们的原来代码是 **using namespace std;** 很容易知道 using 是五个字母 + 一个空格，所以 namespace 这个 token 会从第七个字母开始，因此 Loc 就是第 3 行第 7 列开始;

3. **identifier 'std' [LeadingSpace] Loc=<main.cpp:3:17>**

   第三行中的 identifier 表示标识符，意思就是将'std' 认为是一个标识符而不再是一个关键字，其他同第二行；

4. **semi ';' Loc=<main.cpp:3:20>**

   第四行中的 semi 就是表示分号，一般也表示一行中最后的一个 token，Loc 依旧同上。

5. **int 'int' [StartOfLine] Loc=<main.cpp:5:1>**

   第五行同第一行，开始的第一个'int' 就是表示识别到了关键词'int', 第二个'int' 就是语法分析拆分的 token，其他也同第一行，[StartOfLine] 表示这是此行的开头的 token，Loc 同理；

6. **identifier 'main' [LeadingSpace] Loc=<main.cpp:5:5>**

   第六行依然同第三行，'identifier' 表示将'main' 识别为标识符，其他同理；

7. **l_paren '(' Loc=<main.cpp:5:9>**

   第七行依然同理，'l_paren' 表示左括号，其他同理；

8. **r_paren ')' Loc=<main.cpp:5:10>**

   第八行同第七行；

到此，这八行就是对前两行代码的词法分析结果，其实还挺容易看懂的，对吧?

那我们接着分析一下剩余的部分，词法分析结果如下：

```
// 对应main后的{
l_brace '{' [StartOfLine] Loc=<main.cpp:6:1>

// 对应int i,n,f;
int 'int' [StartOfLine] [LeadingSpace] Loc=<main.cpp:7:5>
identifier 'i' [LeadingSpace] Loc=<main.cpp:7:9>
comma ',' Loc=<main.cpp:7:10>
identifier 'n' Loc=<main.cpp:7:11>
comma ',' Loc=<main.cpp:7:12>
identifier 'f' Loc=<main.cpp:7:13>
semi ';' Loc=<main.cpp:7:14>

// 对应cin>>n;
identifier 'cin' [StartOfLine] [LeadingSpace] Loc=<main.cpp:8:5>
greatergreater '>>' [LeadingSpace] Loc=<main.cpp:8:9>
identifier 'n' Loc=<main.cpp:8:11>
semi ';' Loc=<main.cpp:8:12>
```

词法分析结果

这上面的一段主要就实现了两个功能，一是声明了 3 个 int 类型的'identifier'——'i','n','f'，二是获取了'n' 的输入，其中用 comma 表示',' 并且用 greatergreater 表示 »；

再然后是：

```
// i = 2;
identifier 'i' [StartOfLine] [LeadingSpace] Loc=<main.cpp:10:5>
equal '=' [LeadingSpace] Loc=<main.cpp:10:7>
```

```
numeric_constant '2' [LeadingSpace] Loc=<main.cpp:10:9>
semi ';' Loc=<main.cpp:10:10>
// f = 1;
identifier 'f' [StartOfLine] [LeadingSpace] Loc=<main.cpp:11:5>
equal '=' [LeadingSpace] Loc=<main.cpp:11:7>
numeric_constant '1' [LeadingSpace] Loc=<main.cpp:11:9>
semi ';' Loc=<main.cpp:11:10>
```

词法分析结果

这一段就实现了对两个 identifier 的赋值，这比较需要注意的就是词法分析会将常数识别为'numeric_constant' 类型，其他都同之前的分析；

还有就是：

```
/*

 while(i<=n)
{
    f = f*i;
    i = i+1;
}

*/
while 'while' [StartOfLine] [LeadingSpace] Loc=<main.cpp:12:5>
l_paren '(' Loc=<main.cpp:12:10>
identifier 'i' Loc=<main.cpp:12:11>
lessequal '<=' Loc=<main.cpp:12:12>
identifier 'n' Loc=<main.cpp:12:14>
r_paren ')' Loc=<main.cpp:12:15>
l_brace '{' [StartOfLine] [LeadingSpace] Loc=<main.cpp:13:5>
identifier 'f' [StartOfLine] [LeadingSpace] Loc=<main.cpp:14:9>
equal '=' Loc=<main.cpp:14:10>
identifier 'f' Loc=<main.cpp:14:11>
star '*' Loc=<main.cpp:14:12>
identifier 'i' Loc=<main.cpp:14:13>
semi ';' Loc=<main.cpp:14:14>
identifier 'i' [StartOfLine] [LeadingSpace] Loc=<main.cpp:15:9>
equal '=' Loc=<main.cpp:15:10>
identifier 'i' Loc=<main.cpp:15:11>
plus '+' Loc=<main.cpp:15:12>
numeric_constant '1' Loc=<main.cpp:15:13>
semi ';' Loc=<main.cpp:15:14>
r_brace '}' [StartOfLine] [LeadingSpace] Loc=<main.cpp:16:5>
```

词法分析结果

上述的词法分析结果主要是实现了 while 循环，其中 lessequal 表示 <=,star 表示乘法 *, plus 表示加法 +；

最后就是：

```
// cout<<f<<endl;
identifier 'cout' [StartOfLine] [LeadingSpace] Loc=<main.cpp:17:5>
lessless '<<' [LeadingSpace] Loc=<main.cpp:17:10>
identifier 'f' [LeadingSpace] Loc=<main.cpp:17:13>
lessless '<<' [LeadingSpace] Loc=<main.cpp:17:15>
identifier 'endl' [LeadingSpace] Loc=<main.cpp:17:18>
semi ';' Loc=<main.cpp:17:22>
// return 0;
return 'return' [StartOfLine] [LeadingSpace] Loc=<main.cpp:19:5>
numeric_constant '0' [LeadingSpace] Loc=<main.cpp:19:12>
semi ';' Loc=<main.cpp:19:13>
}
r_brace '}' [StartOfLine] Loc=<main.cpp:20:1>
eof '' Loc=<main.cpp:20:2>
```

词法分析结果

最后是将 identifier f 输出出来，并且返回 main 函数的数值 0，其中需要注意的是词法分析会用 **eof** 这个标识符表示程序的结束。

至此词法分析的部分结束,我们发现原先 main.cpp 内的所有代码都会被拆解为一个个 token，并标注上类型，方便后续语法的分析。我们也成功对这个程序进行了词法分析的验证。

2. 语法分析

语法分析的功能是**将词法分析的结果用于构建抽象语法树（AST）**以及**检查代码语法**。下面我们通过实验来验证一下这两个功能：

我们通过命令：

clang -E -Xclang -ast-dump main.cpp 来获取语法分析的结果，如图：

图 6: 语法分析运行图

```
// using namespace std;
|-UsingDirectiveDecl 0x5d0edb994e98 <main.cpp:3:1, col:17> col:17 Namespace 0
    x5d0edab479e8 'std'
// int main()
-FunctionDecl 0x5d0edb994f18 <line:5:1, line:20:1> line:5:5 main 'int ()'//
    -CompoundStmt 0x5d0edb9a9190 <line:6:1, line:20:1>
   // int i,n,f;
   |-DeclStmt 0x5d0edb995160 <line:7:5, col:14>
   | |-VarDecl 0x5d0edb994fd8 <col:5, col:9> col:9 used i 'int'
   | |-VarDecl 0x5d0edb995058 <col:5, col:11> col:11 used n 'int'
   | -VarDecl 0x5d0edb9950d8 <col:5, col:13> col:13 used f 'int'// 此处是注释:
      cin>>n|-CXXOperatorCallExpr 0x5d0edb9a4d28 <line:8:5, col:11>
      '__istream_type':'std::basic_istream<char>' lvalue '>>'| |-ImplicitCastExpr
      0x5d0edb9a4d10 <col:9> '__istream_type &(*)(int &)' <FunctionToPointerDecay>
```

语法分析结果

从以上的文本形式的 AST，我们可以一句句来看：

1. **|-UsingDirectiveDecl 0x5d0edb994e98 <main.cpp:3:1, col:17> col:17 Namespace 0x5d0edab479e8 'std'**

   第一行表示这是一个 **UsingDirectiveDecl** 节点，而 **main.cpp:3:1,col:17** 表示源代码中第 3 行第 1 列到第 17 列处使用了 `using namespace std;` 指令。这个指令可以将整个 `std` 命名空间引入当前作用域，允许直接使用标准库中的名称而无需前缀 **std::**；

2. **'-FunctionDecl 0x5d0edb994f18 <line:5:1, line:20:1> line:5:5 main 'int ()'**
   第二行表示这是一个 **FunctionDecl** 节点,定义了一个名为 `main` 的函数。**<line:5:1,line:20:1>** 表示 main 函数全部代码从第 5 行第 1 列到第 20 行第 1 列，返回值类型为 `int`，且缺乏参数；

3. **'-CompoundStmt 0x5d0edb9a9190 <line:6:1, line:20:1>**
   第三行表示这是一个 **CompoundStmt** 节点，表示函数体 {}。

4. **|-DeclStmt 0x5d0edb995160 <line:7:5, col:14>**
   第四行表示这是一个 **DeclStmt** 节点，用于声明变量，表示后续会进行变量的定义；

5. **| |-VarDecl 0x5d0edb994fd8 <col:5, col:9> col:9 used i 'int'**
   第五行定义了一个 int 变量 `i`，且 **used** 符号表示该变量后续被使用了；

6. **| |-VarDecl 0x5d0edb995058 <col:5, col:11> col:11 used n 'int'**
   第六行同第五行；

7. **| '-VarDecl 0x5d0edb9950d8 <col:5, col:13> col:13 used f 'int'**
   第七行同第五、第六行；

8. **|-CXXOperatorCallExpr 0x5d0edb9a4d28 <line:8:5, col:11> '___istream__type':'std::basic__istrea lvalue '»'**
   第八行表示这是一个 **CXXOperatorCallExpr** 节点,用来表示对运算符 >> 的调用。**___istream__type** 表示返回值类型为 `std::basic_istream<char>`，**lvalue** 表示其为一个左值，左值指的是出现在赋值语句左边的变量；

让我们接下来看剩余的语法分析后的部分：

```
// cin>>n;
| |-\ImplicitCastExpr 0x5d0edb9a4d10 <col:9> '\_\_istream_type \&(*)(int \&)' <
    FunctionToPointerDecay>
    | | -DeclRefExpr 0x5d0edb9975b0 <col:9> '__istream_type &(int &)' lvalue CXXMethod
        0x5d0edb938f88 'operator>>' '__istream_type &(int &)'| |-DeclRefExpr
        0x5d0edb995178 <col:5> 'istream':'std::basic_istream<char>' lvalue Var
        0x5d0edb9949e8 'cin' 'istream':'std::basic_istream<char>'|
        -DeclRefExpr 0x5d0edb995198 <col:11> 'int' lvalue Var 0x5d0edb995058 'n' 'int'
    // int i = 2;
    |-BinaryOperator 0x5d0edb9a4f90 <line:10:5, col:9> 'int' lvalue '='
    | |-DeclRefExpr 0x5d0edb9a4f50 <col:5> 'int' lvalue Var 0x5d0edb994fd8 'i' 'int'
    | -IntegerLiteral 0x5d0edb9a4f70 <col:9> 'int' 2// int f = 1;|-BinaryOperator
        0x5d0edb9a4ff0 <line:11:5, col:9> 'int' lvalue '='| |-DeclRefExpr 0x5d0edb9a4fb0
        <col:5> 'int' lvalue Var 0x5d0edb9950d8 'f' 'int'|
        -IntegerLiteral 0x5d0edb9a4fd0 <col:9> 'int' 1
    while(i<=n)
    |-WhileStmt 0x5d0edb9a5248 <line:12:5, line:16:5>
    | |-BinaryOperator 0x5d0edb9a5080 <line:12:11, col:14> 'bool' '<='
    | | |-ImplicitCastExpr 0x5d0edb9a5050 <col:11> 'int' <LValueToRValue>
    | | | -DeclRefExpr 0x5d0edb9a5010 <col:11> 'int' lvalue Var 0x5d0edb994fd8 'i'
        'int'| | -ImplicitCastExpr 0x5d0edb9a5068 <col:14> 'int' <LValueToRValue>
    | | -DeclRefExpr 0x5d0edb9a5030 <col:14> 'int' lvalue Var 0x5d0edb995058 'n' 'int'//
        int f = f*i;| -CompoundStmt 0x5d0edb9a5228 <line:13:5, line:16:5>
    | |-BinaryOperator 0x5d0edb9a5150 <line:14:9, col:13> 'int' lvalue '='
    | | |-DeclRefExpr 0x5d0edb9a50a0 <col:9> 'int' lvalue Var 0x5d0edb9950d8 'f' 'int'
    | | -BinaryOperator 0x5d0edb9a5130 <col:11, col:13> 'int' '*'| | |-ImplicitCastExpr
        0x5d0edb9a5100 <col:11> 'int' <LValueToRValue>| | |
        -DeclRefExpr 0x5d0edb9a50c0 <col:11> 'int' lvalue Var 0x5d0edb9950d8 'f' 'int'
    | | -ImplicitCastExpr 0x5d0edb9a5118 <col:13> 'int' <LValueToRValue>| |
        -DeclRefExpr 0x5d0edb9a50e0 <col:13> 'int' lvalue Var 0x5d0edb994fd8 'i' 'int'
    // int i = i + 1;
    | -BinaryOperator 0x5d0edb9a5208 <line:15:9, col:13> 'int' lvalue '='| |-DeclRefExpr
        0x5d0edb9a5170 <col:9> 'int' lvalue Var 0x5d0edb994fd8 'i' 'int'|
        -BinaryOperator 0x5d0edb9a51e8 <col:11, col:13> 'int' '+'
    | |-ImplicitCastExpr 0x5d0edb9a51d0 <col:11> 'int' <LValueToRValue>
    | | -DeclRefExpr 0x5d0edb9a5190 <col:11> 'int' lvalue Var 0x5d0edb994fd8 'i' 'int'|
        -IntegerLiteral 0x5d0edb9a51b0 <col:13> 'int' 1
    // cout << f << endl;
    |-CXXOperatorCallExpr 0x5d0edb9a90f8 <line:17:5, col:18> '\_\_ostream_type':'std::
        basic\_ostream<char>' lvalue '<<'
    | |-ImplicitCastExpr 0x5d0edb9a90e0 <col:15> '\_\_ostream_type \&(*)(\_\
        _ostream_type \&(*)(\_\_ostream_type \&))' <FunctionToPointerDecay>
    | | -DeclRefExpr 0x5d0edb9a9060 <col:15> '__ostream_type &(__ostream_type
        &(*)(__ostream_type &))' lvalue CXXMethod 0x5d0edb906e88 'operator<<'
        '__ostream_type &(__ostream_type &(*)(__ostream_type &))'| |-CXXOperatorCallExpr
        0x5d0edb9a7ce8 <col:5, col:13> '__ostream_type':'std::basic_ostream<char>'
        lvalue '<<'| | |-ImplicitCastExpr 0x5d0edb9a7cd0 <col:10> '__ostream_type
        &(*)(int)' <FunctionToPointerDecay>| | |
        -DeclRefExpr 0x5d0edb9a7c48 <col:10> '\_\_ostream\_type \&(int)' lvalue
        CXXMethod 0x5d0edb907fe8 'operator<<' '\_\_ostream\_type \&(int)'
    | | |-DeclRefExpr 0x5d0edb9a5268 <col:5> 'ostream':'std::basic\_ostream<char>'
        lvalue Var 0x5d0edb994a60 'cout' 'ostream':'std::basic\_ostream<char>'
    | | -ImplicitCastExpr 0x5d0edb9a7c30 <col:13> 'int' <LValueToRValue>| |
        -DeclRefExpr 0x5d0edb9a5288 <col:13> 'int' lvalue Var 0x5d0edb9950d8 'f' 'int'
    | -ImplicitCastExpr 0x5d0edb9a9048 <col:18> 'basic_ostream<char, char_traits<char>>
        &(*)(basic_ostream<char, char_traits<char>> &)' <FunctionToPointerDecay>|
        -DeclRefExpr 0x5d0edb9a9020 <col:18> 'basic\_ostream<char, char\_traits<char>>
        \&(basic\_ostream<char, char\_traits<char>> \&)' lvalue Function 0x5d0edb90c060
```

```
        'endl' 'basic\_ostream<char, char\_traits<char>> \&(basic\_ostream<char, char\
        _traits<char>> \&)' (FunctionTemplate 0x5d0edb8e9188 'endl')
29      // return 0;
30      -ReturnStmt 0x5d0edb9a9180 <line:19:5,
            col:12>-IntegerLiteral 0x5d0edb9a9160 <col:12> 'int' 0
```

语法分析结果

限于篇幅的原因，这里就不再一句句进行讲解了，大概介绍一下一些关键字的作用。

表 2: 语法分析关键字

AST 关键字	含义	示例
ImplicitCastExpr	隐式类型转换（如函数指针衰减、值转换）	`cin >> n` 中函数指针衰减（函数-> 函数指针）
CXXMethod	C++ 成员函数（运算符重载等）	`operator>>`
DeclRefExpr	变量或函数引用	`cin`, `n`, `i`, `f`
IntegerLiteral	整数常量（int）	`2`, `1`
BinaryOperator	二元运算符（=, <= 等）	`i = 2`, `f = 1`, `i <= n`
WhileStmt	while 语句结构	`while(i <= n)`

看完上面的内容，我们知道了语法分析可以将词法分析得到的结果转换为一颗抽象语法树 AST，那接下来我们就去验证语法分析的第二个作用——检查代码语法。

接下来我们将我们的 main.cpp 进行略微的修改，改为 **wrong.cpp**，来测试一下语法分析的纠错功能。

wrong.cpp 代码如下：

```cpp
#include<iostream>
using namespace std;
// main拼写错为mian
int mian()
{
    int i,n,f;
    cin >>n;
    i = 2;
    f = 1;
    // while关键字错拼为whlie
    whlie(i<=n)
    {
        f=f*i;
        i=i+1;
    }
    // cout符号反向
    cout >> f >> endl;
    return 0;
}
```

wrong.cpp

依然进行语法分析，得到下图：

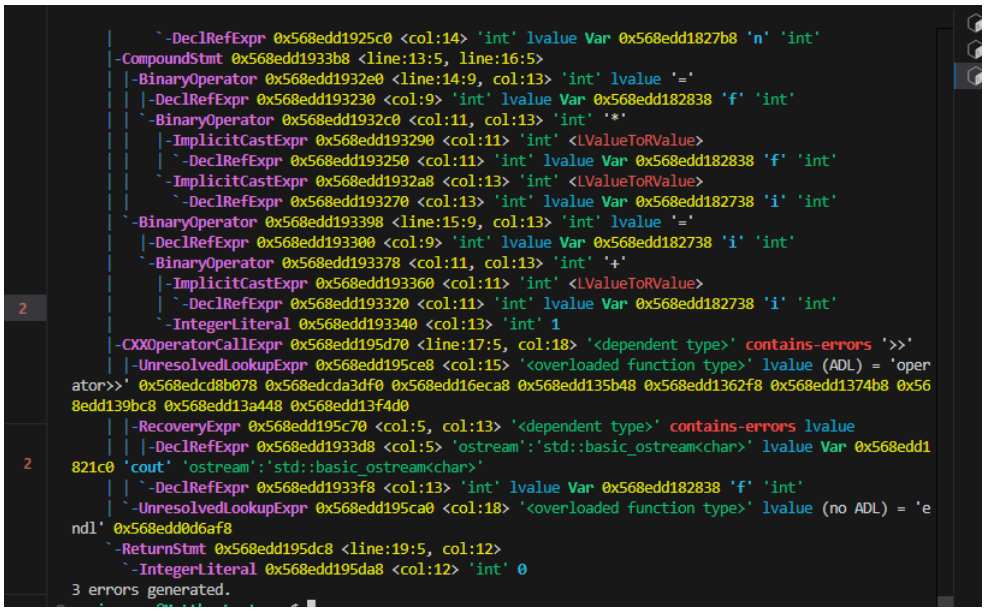


图 7: 语法分析运行图

图中告诉我们，发现了三个 error，那发现的是我们的三个 error 么?
从图中报错信息我们可以获得 3 个 error 来自两个语句:

1. **语句：|-RecoveryExpr ... contains-errors**
   | |-UnresolvedLookupExpr ... = 'whlie' empty

   这表明语法分析发现根本找不到关键字'while'，因为我们错将'while' 写为了'whlie'，找到了第一个我们的 error;

2. **语句：|-CXXOperatorCallExpr ... contains-errors '»'**
   | |-UnresolvedLookupExpr ... = 'operator»' ...
   | |-RecoveryExpr ... contains-errors
   | | |-DeclRefExpr ... 'cout'
   | | '-DeclRefExpr ... 'f'
   | '-UnresolvedLookupExpr ... = 'endl'

   这一处对应的是源代码 **cout«f«endl**, 但我们故意将代码写为了 **cout»f»endl;**, 语法分析发现了两处错误——第一是将第一处'»' 解析成了错误的运算符的调用，而不是运算符重载当作输出来用，这样也算找到了我们的第二个 error! 第二是 endl 需要 std::endl 而不能直接使用 endl，报了未知标识符的错误;

那为什么语法分析找到的第三处错误会是有关 endl 的而不是我们一开始错写的 int mian() 呢?

通过我上 Google 以及询问 GPT 得知：因为 std::endl 并不是一个类似 int 的关键字，而是一个在 <ostream> 内的函数模板，库内有类似 ostream& operator «(ostream&,...) 的重载函数，但是没有 ostream& operator »(ostream&,...) 这样的，所以会报未知标识符的错误。

而为什么找不到 int mian () 这个错误呢? 这个主要是因为这个问题是语义上的错误而非语法上的错误，这个错误最后会影响链接器, 导致其无法找到合适的入口函数, 导致错误。但是并不会影响语法分析的过程，所以语法分析无法发现这个问题。

综上，我们验证了语法分析的两个作用，一个是**生成抽象语法树 AST**, 二是**检查代码语法**。

3. 语义分析

语义分析是编译器在语法分析之后、代码生成之前做的一步检查工作。它的主要作用是确保程序不仅“长得像对的代码”，而且在“意思上也对”，也就是符合语言的语义规则。

1. **主要任务**：语义分析是在语法分析之后进行的，目的是检查程序是否在“意义”上正确，比如类型对不对、变量有没有定义、函数能不能调用等。
2. **符号和作用域**：编译器会建立符号表，记录变量、函数、类等名字和属性。还会检查作用域，确保你用的名字能找到对应的声明。
3. **类型检查**：检查表达式和赋值的类型是否兼容，必要时加上隐式转换，比如数组衰减成指针等。如果发现类型完全对不上，就报错。
4. **常见错误检查**：包括 return 类型不对、break 用在循环外、重载函数没法区分、或者调用了没声明的函数等，这些都是语义分析阶段发现的。
5. **结果和作用**：最后编译器会得到一个带类型信息和符号绑定的抽象语法树（AST），这个结构保证后面的中间代码生成和优化有可靠的依据。

通过语义分析，编译器可以检查类型是否匹配、变量和函数是否存在、控制语句是否合理等等，并为后续生成中间代码和优化提供准确的信息。

4. 中间代码生成

中间代码生成不仅是编译中比较重要的一步，同时也是我们所需要重点掌握的一步。我们可以通过命令：

clang -S -emit-llvm main.cpp 来生成中间代码，得到以下 main.ll 文件：

```
; ModuleID = 'main.cpp'
source_filename = "main.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

module asm ".globl _ZSt21ios_base_library_initv"

%"class.std::basic_istream" = type { ptr, i64, %"class.std::basic_ios" }
%"class.std::basic_ios" = type { %"class.std::ios_base", ptr, i8, i8, ptr, ptr, ptr, ptr }
%"class.std::ios_base" = type { ptr, i64, i64, i32, i32, i32, ptr, %"struct.std::ios_base::_Words", [8 x %"struct.std::ios_base::_Words"], i32, ptr, %"class.std::locale" }
%"struct.std::ios_base::_Words" = type { ptr, i64 }
%"class.std::locale" = type { ptr }
%"class.std::basic_ostream" = type { ptr, %"class.std::basic_ios" }

@_ZSt3cin = external global %"class.std::basic_istream", align 8
@_ZSt4cout = external global %"class.std::basic_ostream", align 8

; Function Attrs: mustprogress noinline norecurse optnone uwtable
```

```
define dso_local noundef i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  %5 = call noundef nonnull align 8 dereferenceable(16) ptr @_ZNSirsERi(ptr noundef
       nonnull align 8 dereferenceable(16) @_ZSt3cin, ptr noundef nonnull align 4
       dereferenceable(4) %3)
  store i32 2, ptr %2, align 4
  store i32 1, ptr %4, align 4
  br label %6

6: ; preds = %10, %0
  %7 = load i32, ptr %2, align 4
  %8 = load i32, ptr %3, align 4
  %9 = icmp sle i32 %7, %8
  br i1 %9, label %10, label %16

10: ; preds = %6
  %11 = load i32, ptr %4, align 4
  %12 = load i32, ptr %2, align 4
  %13 = mul nsw i32 %11, %12
  store i32 %13, ptr %4, align 4
  %14 = load i32, ptr %2, align 4
  %15 = add nsw i32 %14, 1
  store i32 %15, ptr %2, align 4
  br label %6, !llvm.loop !6

16: ; preds = %6
  %17 = load i32, ptr %4, align 4
  %18 = call noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEi(ptr noundef
       nonnull align 8 dereferenceable(8) @_ZSt4cout, i32 noundef %17)
  %19 = call noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEPFRSoS_E(ptr
       noundef nonnull align 8 dereferenceable(8) %18, ptr noundef
       @_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_)
  ret i32 0
}

declare noundef nonnull align 8 dereferenceable(16) ptr @_ZNSirsERi(ptr noundef nonnull
    align 8 dereferenceable(16), ptr noundef nonnull align 4 dereferenceable(4)) #1

declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEi(ptr noundef nonnull
    align 8 dereferenceable(8), i32 noundef) #1

declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEPFRSoS_E(ptr noundef
    nonnull align 8 dereferenceable(8), ptr noundef) #1

```

```
declare noundef nonnull align 8 dereferenceable(8) ptr
    @_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_(ptr noundef nonnull
    align 8 dereferenceable(8)) #1

attributes #0 = { mustprogress noinline norecurse optnone uwtable "frame-pointer"="all"
    "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"
    ="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
     "tune-cpu"="generic" }
attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-protector-
    buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse
    ,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"Ubuntu clang version 18.1.3 (1ubuntu1)"}
!6 = distinct !{!6, !7}
!7 = !{!"llvm.loop.mustprogress"}
```

main.ll

面对这个 llvm 格式的代码，直接观察可能不是很简单，但我们可以通过下列命令 **gcc -fdump-tree-all-graph main.cpp** 获取中间过程的输出，获取后我们会得到.dot 文件，然后再通过 vscode 中的 graphviz 插件将.dot 文件转换为控制流图 CFG，并对 CFG 进行分析。

最后得到如图 CFG：

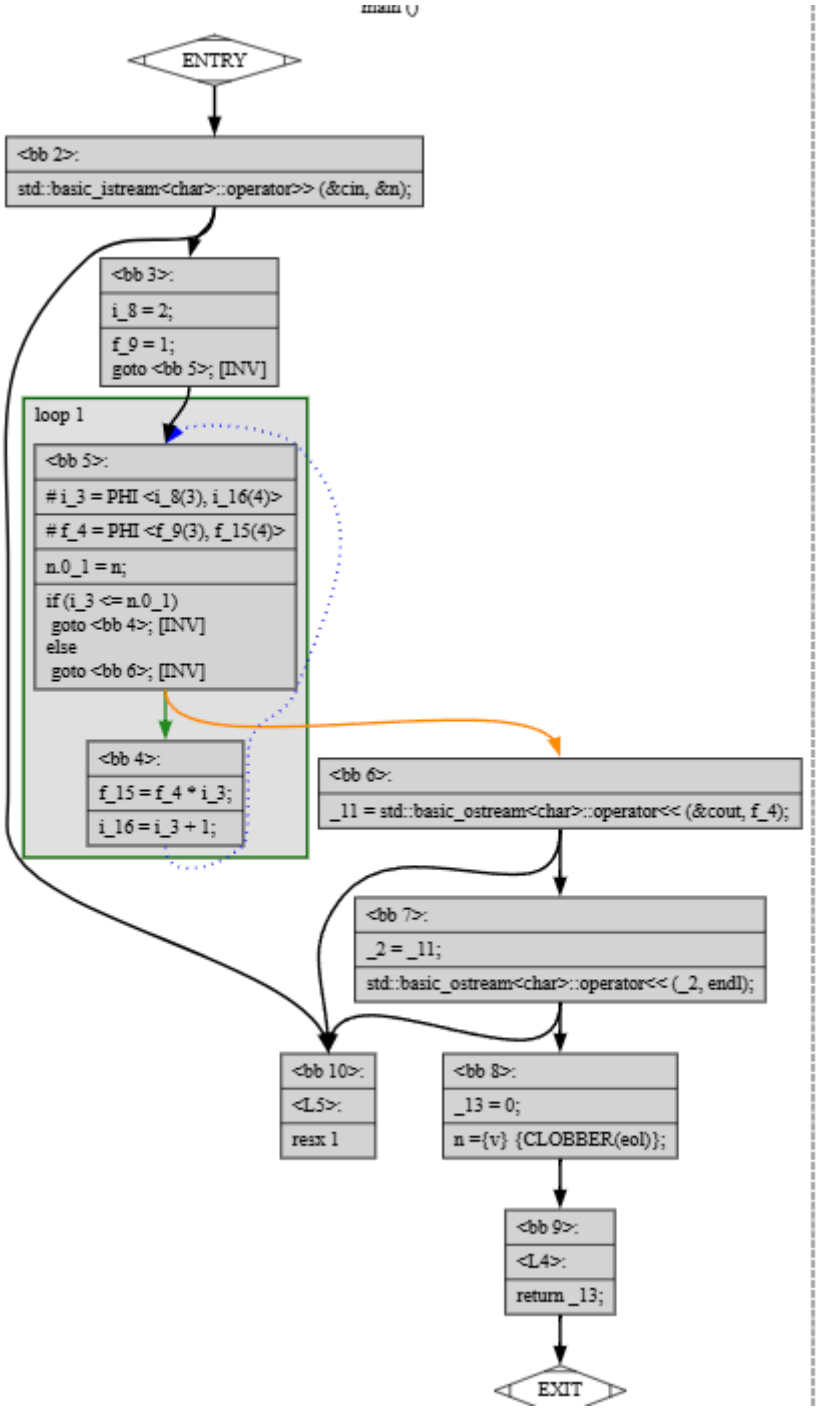


图 8: CFG 可视化图

对于上图，虽然并不清楚具体变量代表着什么，但我们可以很轻松的发现程序的逻辑关系和大概走向。

比如程序从 entry 进入，先是进入到类似 cin»n 一样的输入环节，然后会有一个分支，可以往两个方向运行，然后 loop 1 就是代表一个循环，进入循环后依然有一个分支，接下来还是分支，最后走向 exit。

又比如虚线应该是指代的是 loop 进行完一轮后重新回去进行 loop，并且代码中存在着十分多的 if 语句进行分支的选择。且变量名都被类似 i_3 这样的符号替代，用这样的表示方法我猜测大概是因为程序中变量十分的多，只能用这样的方式进行表示；

总结的来说，我们可以大概看出各个块分别对应什么样的功能：

块编号	对应代码	说明
unknown	ENTRY	main 函数入口
unknown	EXIT	main 函数出口
bb2	cin » n	获取输入 n
bb3	i=2; f=1;	初始化循环 loop1 的变量 i 和 f
bb4	f = f * i; i = i + 1;	循环体 loop1 的操作
bb5	if (i <= n)	是否进行循环 loop1 的判断
bb6	cout « f;	输出循环 loop1 结果 f
bb7	cout « endl;	输出换行
bb8	unknown	unknown
bb9	return 0;	返回值 0
bb10	unknown	unknown

表 3: CFG 各个块对应代码及功能

5. 代码优化

代码优化也是一个很重要的步骤，其能改进中间代码并且生成更好的目标代码。我们这里指的更好的目标代码一般是指实现相同效果但是运行时间更短的代码。

那我们现在就进行测试，分别对代码进行三种优化方式，即-O1 -O2 -O3。

但我们在进行代码优化之前，需要先将.ll 格式的代码转化为.bc 格式的文件，通过命令：

llvm-as main.ll -o main.bc

然后再将 main.bc 通过以下命令：**opt -S -O1 main.bc -o main-O1.ll** 得到文件 main-O1.ll。

然后我们再输入命令 **diff main.ll main-O1.ll**，查看-O1 优化后二者的差距，如图：

```
yuxiangyou@MatthewLaptop:~$ llvm-as main.ll -o main.bc
yuxiangyou@MatthewLaptop:~$ opt -S -O1 main.bc -o main-O1.ll
yuxiangyou@MatthewLaptop:~$ diff main.ll main-O1.ll
1c1
< ; ModuleID = 'main.cpp'
---
> ; ModuleID = 'main.bc'
19c19
< define dso_local noundef i32 @main() #0 {
---
> define dso_local noundef i32 @main() local_unnamed_addr #0 {
53c53
< declare noundef nonnull align 8 dereferenceable(16) ptr @_ZNSirsERi(ptr noundef nonnull align 8 dere
ferenceable(16), ptr noundef nonnull align 4 dereferenceable(4)) #1
---
> declare noundef nonnull align 8 dereferenceable(16) ptr @_ZNSirsERi(ptr noundef nonnull align 8 dere
ferenceable(16), ptr noundef nonnull align 4 dereferenceable(4)) local_unnamed_addr #1
55c55
< declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEi(ptr noundef nonnull align 8 derefe
renceable(8), i32 noundef) #1
---
> declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEi(ptr noundef nonnull align 8 derefe
renceable(8), i32 noundef) local_unnamed_addr #1
57c57
< declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEPFRSoS_E(ptr noundef nonnull align 8
 dereferenceable(8), ptr noundef) #1
---
> declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEPFRSoS_E(ptr noundef nonnull align 8
 dereferenceable(8), ptr noundef) local_unnamed_addr #1
yuxiangyou@MatthewLaptop:~$
```

图 9: -O1 优化后代码差异

完整输出如下：

```
1c1
< ; ModuleID = 'main.cpp'
---
> ; ModuleID = 'main.bc'
19c19
< define dso_local noundef i32 @main() #0 {
---
> define dso_local noundef i32 @main() local_unnamed_addr #0 {
53c53
< declare noundef nonnull align 8 dereferenceable(16) ptr @_ZNSirsERi(ptr noundef
    nonnull align 8 dereferenceable(16), ptr noundef nonnull align 4 dereferenceable(4))
     #1
---
> declare noundef nonnull align 8 dereferenceable(16) ptr @_ZNSirsERi(ptr noundef
    nonnull align 8 dereferenceable(16), ptr noundef nonnull align 4 dereferenceable(4))
     local_unnamed_addr #1
55c55
< declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEi(ptr noundef nonnull
    align 8 dereferenceable(8), i32 noundef) #1
---
> declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEi(ptr noundef nonnull
    align 8 dereferenceable(8), i32 noundef) local_unnamed_addr #1
57c57
< declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEPFRSoS_E(ptr noundef
    nonnull align 8 dereferenceable(8), ptr noundef) #1
---
> declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEPFRSoS_E(ptr noundef
    nonnull align 8 dereferenceable(8), ptr noundef) local_unnamed_addr #1
```

main.ll 与 main-O1.ll 差异

我们一个个差异进行比较：

```
1c1
< ; ModuleID = 'main.cpp'
---
> ; ModuleID = 'main.bc'
```

main.ll 与 main-O1.ll 差异

这个就表示：main.ll 是由 main.cpp 转化而来的，而 main-O1.ll 则是由 main.bc 转化而来的。

```
19c19
< define dso_local noundef i32 @main() #0 {
---
> define dso_local noundef i32 @main() local_unnamed_addr #0 {
```

main.ll 与 main-O1.ll 差异

我们可以发现，-O1 优化时，@main() 后多了一个”local_unnamed_addr”, 这个表示的是假设函数的地址在模块内部是没有命名的，局部是唯一的, 这样可以方便-O1 的优化，可以在最终将函数放在任何位置；

```
53c53
< declare noundef nonnull align 8 dereferenceable(16) ptr @_ZNSirsERi(ptr noundef
    nonnull align 8 dereferenceable(16), ptr noundef nonnull align 4 dereferenceable(4))
     #1
---
> declare noundef nonnull align 8 dereferenceable(16) ptr @_ZNSirsERi(ptr noundef
    nonnull align 8 dereferenceable(16), ptr noundef nonnull align 4 dereferenceable(4))
     local_unnamed_addr #1
```

main.ll 与 main-O1.ll 差异

我们可以发现，依然是多了”local_unnamed_addr”，只不过这次是在函数声明时假设函数的地址在模块内部是没命名且局部唯一；

```
55c55
< declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEi(ptr noundef nonnull
    align 8 dereferenceable(8), i32 noundef) #1
---
> declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEi(ptr noundef nonnull
    align 8 dereferenceable(8), i32 noundef) local_unnamed_addr #1
```

main.ll 与 main-O1.ll 差异

我们可以发现，依然是多了”local_unnamed_addr”；

```
57c57
< declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEPFRSoS_E(ptr noundef
    nonnull align 8 dereferenceable(8), ptr noundef) #1
---
> declare noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEPFRSoS_E(ptr noundef
    nonnull align 8 dereferenceable(8), ptr noundef) local_unnamed_addr #1
```

main.ll 与 main-O1.ll 差异

我们可以发现，依然是多了”local_unnamed_addr”；

那我们就大胆猜想，是不是-O1 优化的思路就是，将函数地址假设在模块内部没有命名并且是局部唯一的，然后去问问 GPT，GPT 给出的回答是：

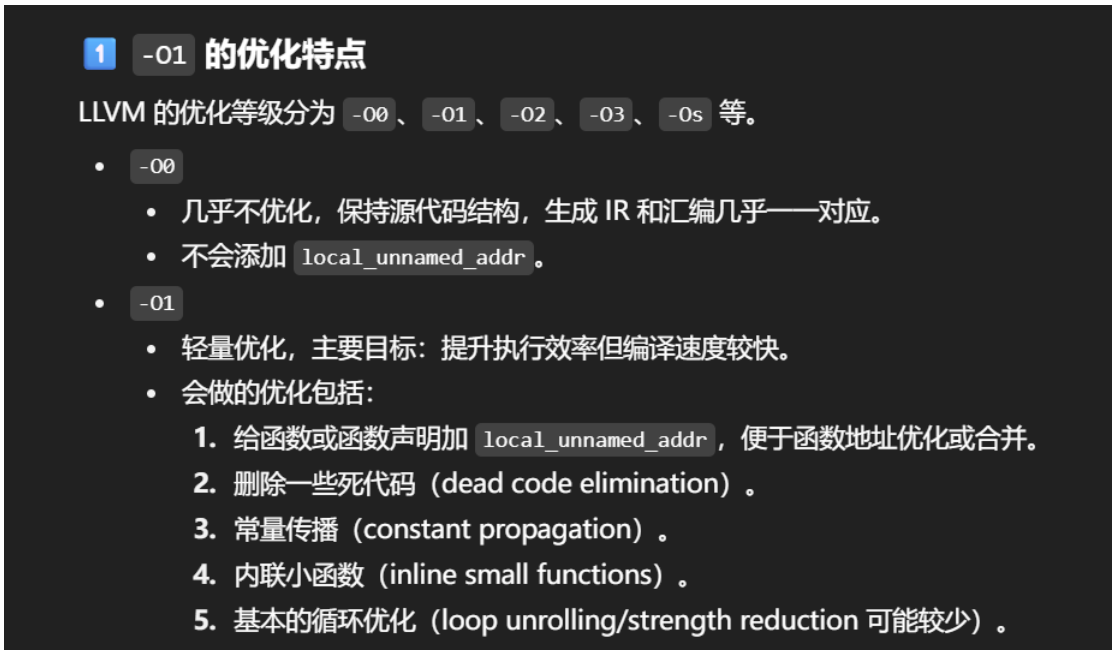


图 10: GPT 有关-O1 优化答案

证明猜想并不太对，但也是一次好的尝试。

然后我们略微修改 main.cpp，增加计时功能，并且测试当 n=10 时未进行优化以及-O1,-O2,-O3 优化后分别的程序运行速度，修改后 test.cpp 如下：

```cpp
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;
// 计算 10! 并 运行 10.000次计算平均用时
int main() {
    int n = 10;
    int trials = 10000;

    long long result = 1;
    double avg_time = 0.0;

    auto start_total = high_resolution_clock::now();

    for (int t = 0; t < trials; t++) {
        result = 1;
        auto start = high_resolution_clock::now();

        for (int i = 2; i <= n; i++) {
            result *= i;
        }

        auto end = high_resolution_clock::now();
        auto duration = duration_cast<nanoseconds>(end - start).count();
```

```
        avg_time += duration;
    }

    auto end_total = high_resolution_clock::now();

    avg_time /= trials;

    cout << "10!=" << result << endl;
    cout << "Avgtime:" << avg_time / 1e6 << "ms" << endl;
    cout << "Totaltime:" << duration_cast<milliseconds>(end_total - start_total).count()
         << "ms" << endl;

    return 0;
}
```

test.cpp

运行结果如图：

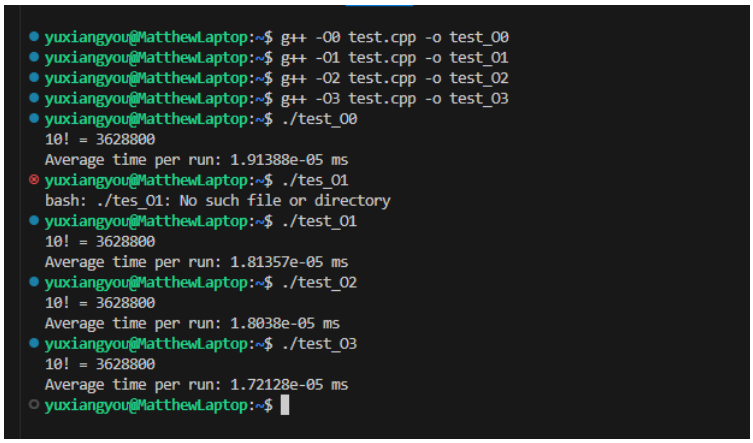


图 11: 代码优化运行时间测试

具体运行时间如下表：

表 4: 不同优化等级计算 10! 的平均运行时间

优化等级	平均每次运行时间 (ms)
-O0	1.91388×10^{-5}
-O1	1.81357×10^{-5}
-O2	1.80380×10^{-5}
-O3	1.72128×10^{-5}

从表中结果可以看出，随着优化等级的提高，平均每次的运行时间也都在下降，只能说对于阶乘这个简单的 cpp 程序，-O1 等级的代码优化就能起到比较好的效果，也算是十分“梦幻”般的结果了。

但实际上根据上学期学并行的经验，当面对较复杂一点的情形时，选择更高等级的（例如-O3 等级的优化）不一定会比低等级（例如-O2 等级的优化）快，这个的原因就比较的复杂了，大概

可以归为以下几类：

1. **优化本身需要开销**：高级优化会做更多复杂处理，比如循环展开、向量化等，这些优化需要额外计算，有时会抵消部分性能提升。
2. **控制流和流水线影响**：优化可能改变循环或判断结构，使 CPU 的分支预测失效或流水线被冲刷，影响效率。
3. **内存访问顺序变化**：优化可能改变数据访问顺序，使原本顺序访问变成不规则访问，增加缓存未命中率。

但从此次实验，我们可以说，代码优化**能改进中间代码并且生成更好的目标代码**。

6. 代码生成

代码生成主要是将中间表示（IR）转换为机器码，其的主要任务是把编译器理解的抽象操作映射为具体 CPU 指令，同时尽量利用寄存器和内存资源提高执行效率。

我们可以利用以下命令将 IR 转换为不同的机器码：

当不显式地指定平台时，命令为 **llc main.ll -o main.S** ，机器会自动识别所在平台并生成对应的机器码，比如直接在 Ubuntu 上运行的话，得到的就是 Linux 平台的机器码。

此处运行命令 **llc main.ll -o main.S** ，如图：

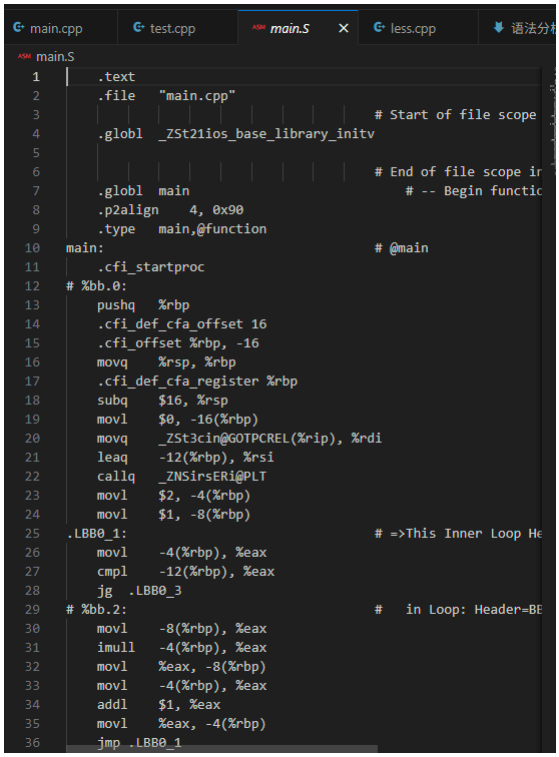


图 12: 生成 main.S 的代码

至此，我们就做完了有关编译阶段的所有实验，也熟悉了六个环节各个环节的作用。

(四) 汇编阶段

实验指导书中提到，汇编过程实际上是把汇编语言程序代码翻译成目标机器指令的过程。其最终生成的是可重定位的机器代码。

那让我们先做完实验再来总结一下汇编阶段具体做了什么并且有什么作用。

我们可以输入命令 **gcc main.S -c -o main.o** 得到汇编器处理后的结果，如图：

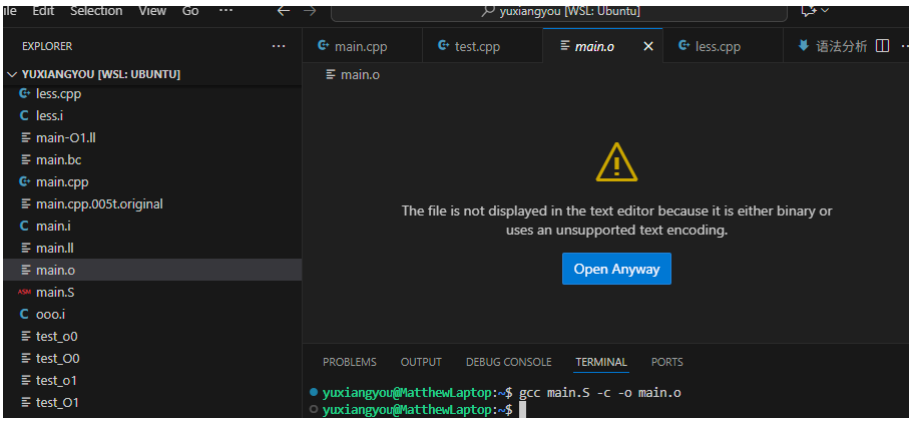


图 13: 生成 main.o

然后我们发现 main.o 无法直接打开，这是因为它是一个二进制文件，我们只需要通过命令 **objdump -d main.o** 就能进行反汇编，查看机器码所对应的汇编指令了，运行命令得到：

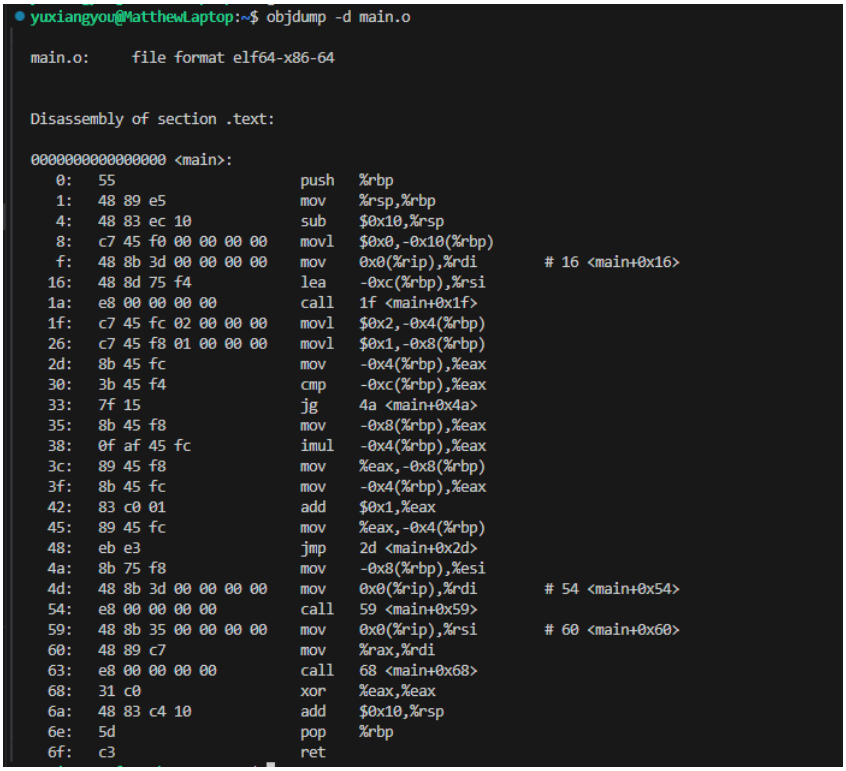


图 14: 反汇编查看汇编指令

完整汇编指令如下：

```
0000000000000000 <main>:
   0: 55 push %rbp // 建立新的栈帧
   1: 48 89 e5 mov %rsp,%rbp // 将rsp挪到rbp位置
   4: 48 83 ec 10 sub $0x10,%rsp // 将rsp-16，留出16B的位置
```

```
// 将0存在栈帧-16B的位置
8: c7 45 f0 00 00 00 00 movl $0x0,-0x10(%rbp)
// 将rip内的值加载到rdi，作为函数第一个参数
f: 48 8b 3d 00 00 00 00 mov 0x0(%rip),%rdi # 16 <main+0x16>
// 取rbp-12B的地址存在rsi
16: 48 8d 75 f4 lea -0xc(%rbp),%rsi
1a: e8 00 00 00 00 call 1f <main+0x1f> // 跳转地址，调用函数
1f: c7 45 fc 02 00 00 00 movl $0x2,-0x4(%rbp) // 局部变量赋值
26: c7 45 f8 01 00 00 00 movl $0x1,-0x8(%rbp) // 同上
2d: 8b 45 fc mov -0x4(%rbp),%eax // 局部变量赋值给eax
30: 3b 45 f4 cmp -0xc(%rbp),%eax // 比较eax与输入值
33: 7f 15 jg 4a <main+0x4a> // 判断条件的跳转
35: 8b 45 f8 mov -0x8(%rbp),%eax // 局部变量赋给eax
38: 0f af 45 fc imul -0x4(%rbp),%eax // 乘法
3c: 89 45 f8 mov %eax,-0x8(%rbp) // eax值赋给rbp-8B
3f: 8b 45 fc mov -0x4(%rbp),%eax // eax = i
42: 83 c0 01 add $0x1,%eax // i++
45: 89 45 fc mov %eax,-0x4(%rbp) // 保存i++到rbp-4B
48: eb e3 jmp 2d <main+0x2d> // 接着loop
4a: 8b 75 f8 mov -0x8(%rbp),%esi //esi存取结果
// 加载地址到rdi
4d: 48 8b 3d 00 00 00 00 mov 0x0(%rip),%rdi # 54 <main+0x54>
54: e8 00 00 00 00 call 59 <main+0x59> // 调用函数
// 加载地址到rsi
59: 48 8b 35 00 00 00 00 mov 0x0(%rip),%rsi # 60 <main+0x60>
60: 48 89 c7 mov %rax,%rdi // 将rax值移动到rdi内
63: e8 00 00 00 00 call 68 <main+0x68> // 调用函数
68: 31 c0 xor %eax,%eax // 异或将eax置0
6a: 48 83 c4 10 add $0x10,%rsp // 释放空间
6e: 5d pop %rbp // 将栈帧恢复
6f: c3 ret // return 0;
```

test.cpp

（五） 链接 & 加载阶段

从指导书我们可以知道，由汇编程序生成的目标文件不能够直接执行。而一个大型的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。

我们可以通过如下命令 **g++ main.o -o main** 获得输出如图：

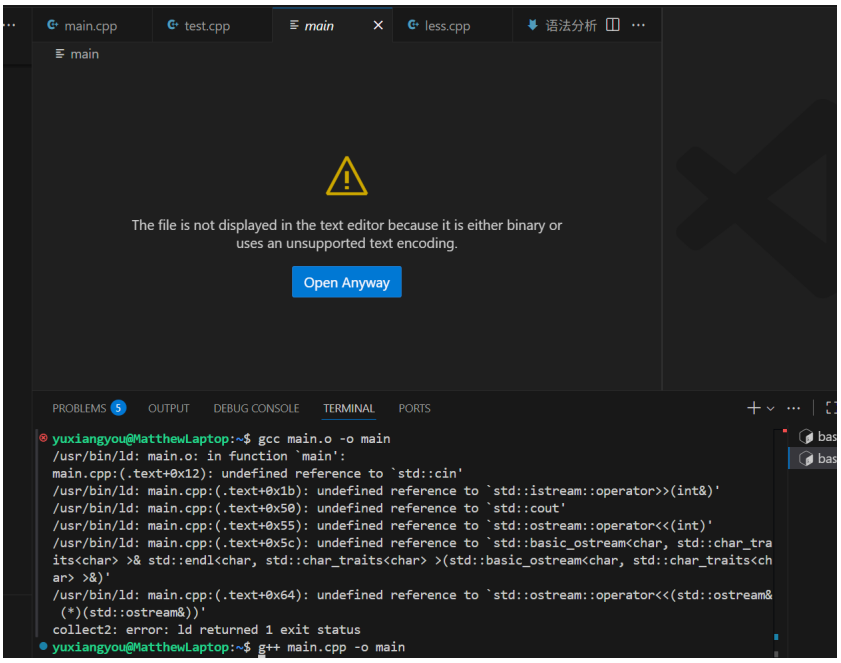


图 15: 输入命令后的运行图

我们可以看到，得到的依然是一个二进制的文件，然后我们再通过命令**./main** 将其运行一下，如图：

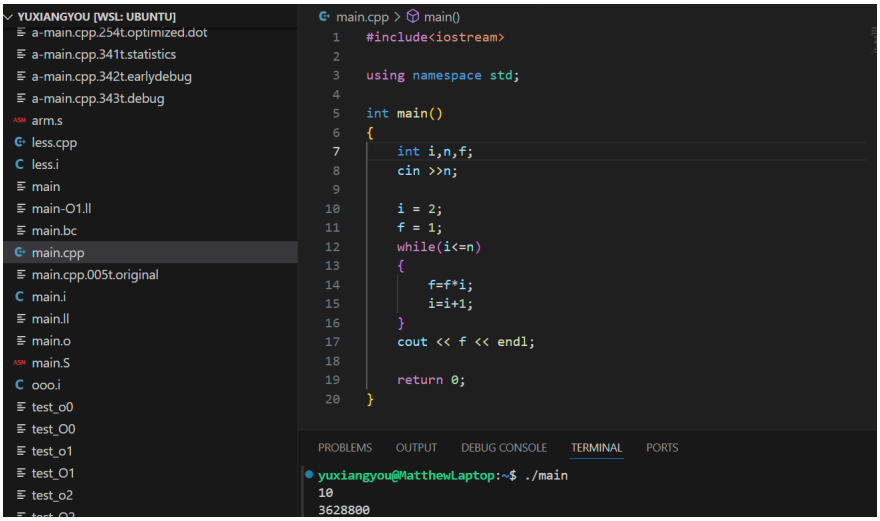


图 16: 运行图

可以发现当我输入了一个数字 10 后，程序就输出了 10!=3628800, 证明运行结果正确！

那至此编译的全过程的实验我们就都做完了，从预处理-> 编译-> 汇编-> 链接-> 加载阶段，一共是五个阶段，而其中编译阶段以及汇编阶段是我们所需要重点掌握的，编译阶段又分为 6 个小的阶段，这些都是我们要在今后的课程中所慢慢学习，慢慢掌握的内容。

三、 SysY 程序设计

当我们输入 5 的时候，输出了斐波那契数列，运行正确！

我阅读了群里发的 SysY 语言的介绍，得知 SysY 是类 C 的简化编程语言，降低了很多语法复杂度，主要聚焦是程序逻辑到中间代码再到目标代码。除去了 C 中复杂的语法，比如指针运算，动态内存等。

(一) 基础语法

基础语法与 C 的类似，但是进行了简化。

- **数据类型：** 仅支持 int 类型，没有 char,float, 指针等
- **变量与常量：** 支持局部变量，全局变量，常量仅支持整数字面值常量
- **控制流：** 仅保留了 3 种基础结构——顺序执行，if-else 分支，while 循环。
- **函数：** 仅支持 main 函数（程序入口），无自定义函数，且 main 的返回值固定为 int

(二) 与 C 的差别

SysY 语言与 C 的差别主要体现在以下三个方面

- **输入输出：** 与 C 的依赖于标准库的输入输出（printf/scanf）不同，SysY 调用专属的运行时函数。
  - **getint()：** 读取一个整数，无参数，返回 int
  - **putint()：** 输出整数 x，无返回值
- **库依赖：** 仅依赖极简的 SysY 运行时库，无其他标准库
- **语法限制：** 没有结构体，数组，宏定义，预处理指令

(三) 程序设计

基于此，我们设计出一个 SysY 程序，涵盖所有基础的语言特性如赋值运算，数值运算，条件分支，循环等，并且**我们在此基础上，实现了进阶的语言特性，自定义函数，一维数组，const 常量和逻辑运算**。

我们设计的 SysY 程序如下:

```
// 进阶特性：const常量定义
const int SIZE = 3; // 数组长度
const int MIN_POSITIVE = 1; // 最小正值判断

// 进阶特性：自定义函数（计算数组中的正偶数和）
int sum_even_positive(int nums[]) {
    int sum = 0;
    int i = 0;
    while (i < SIZE) { // 基础特性：while循环
        // 进阶特性：逻辑运算（&&表示与，||表示或）
        if (nums[i] >= MIN_POSITIVE && (nums[i] % 2 == 0 || SIZE == 1)) {
            sum += nums[i]; // 基础特性：数值运算与赋值
        }
        i++;
```

```
    }
    return sum; // 函数返回值
}

int main() {
    int arr[SIZE]; // 进阶特性：一维数组
    int i = 0;
    int total; // 基础特性：局部变量

    // 基础特性：循环输入数组元素
    while (i < SIZE) {
        arr[i] = getint(); // 基础特性：输入函数
        i++;
    }

    // 进阶特性：调用自定义函数
    total = sum_even_positive(arr);

    // 基础特性：条件分支
    if (total > 0) {
        putint(total); // 基础特性：输出函数
    } else {
        putint(-1); // 输出标识值
    }

    return 0;
}
```

我们的代码很简单，主要逻辑就是把一个一维数组中的偶数加起来，并将结果输出，代码很简单不再解释，接下来我们来编写 LLVM IR 和汇编编程。

四、 任务二：LLVM IR 编程

(一) 任务介绍

LLVM IR 编程是衔接编译器中间表示与目标代码生成的关键环节，任务二要求手动编写 LLVM IR 代码，理解中间语言的特性，与 SysY 语言的映射关系，并掌握编译和运行流程。

我们本次实验主要是通过编写我们设计的 SysY 程序的 LLVM IR 代码，了解中间语言的核心特性和语法规则。

(二) LLVM IR 的特性和语法

在进行实验之前，我首先查阅了一些资料，了解了一下 LLVM IR（LLVM Intermediate Representation）的核心特性和语法规则。

LLVM IR 是 LLVM 项目中的中间表示 i 形式，在编译过程中起到承上启下的作用，可以方便我们的对代码进行分析，优化和生成目标代码。

1. 核心特性

静态类型语言 LLVM IR 是一种静态类型语言，每个值都有明确的类型，并且在编译时就确定下来，不会像动态类型语言一样在运行时进行类型检查。比如 i32 表示 32 位有符号整数，i64 表示 64 位有符号整数，float 表示单精度浮点数，double 表示双精度浮点数，ptr 表示指针类型。

表现形式 LLVM IR 有三种表现形式，分别是二进制形式，文本形式，内存形式。

- 二进制形式：以.bc 为后缀，是紧凑的二进制格式，用于编译器内部的处理和存储。我们可以使用 llvm-as 工具将文本形式的 LLVM IR 转换成二进制的形式，用 llvm-dis 工具再转回来
- 内存形式：在 LLVM 的 C++ 代码库中，以队形和数据结构的形式存在，如基本块（BasicBlock）和指令（Instruction）等类，用于在 LLVM 框架中对 IR 进行操作和优化
- 文本形式：以.ll 为文件后缀，是人类可读的形式，便于编写，阅读和调试。

可移植性和复用性 并且，LLVM IR 不依赖于特定的硬件架构，这使得它可以作为多种高级语言到不同目标机器代码的通用的中间表示，论是编译成 x86、ARM、RISC-V 等架构的代码，都可以先将高级语言转换为 LLVM IR，再通过后端处理生成对应架构的目标代码。

优化多样 LLVM 提供了一系列强大的优化 Pass，比如我们之前提到过的死代码消除，循环展开等，这些优化可以直接作用于 LLVM IR，在生成目标代码之前对代码进行优化，提高程序执行的效率。

2. LLVM IR 语法

标识符 LLVM IR 中的标识符分为两类，分别是全局标识符和局部标识符。

- **全局标识符：**以 @ 开头，用于表示全局变量、全局函数等。
- **局部标识符：**以% 开头，用于表示局部变量、临时值或者虚拟寄存器。

下边给出几个例子：

```
;一个初始值为0的全局32位整数变量
@global_var = global i32 0
;一个名为main的全局函数
define i32 @main()
;%temp 用来存储%a和%b相加的结果
%temp = add i32 %a, %b
```

函数定义 在 LLVM IR 中函数定义使用 define 关键字，格式为 define< 返回值类型 > @< 函数名 >(< 参数列表 >){< 函数体 >}。

```
;一个名为mutiply的函数，接受两个i32类型的参数，计算乘积后通过ret指令返回。
define i32 @multiply(i32 %x, i32 %y) {
    %product = mul i32 %x, %y
    ret i32 %product
}
```

特殊类型声明 LLVM IR 中可以通过 type 关键字自定义一些复杂类型，如结构体、数组等。

```
;声明了一个包含两个 32 位整数的结构体类型
%struct.MyStruct = type { i32, i32 }
```

指令

- **内存操作指令：** alloca 用于在栈上分配内存，返回一个指针类型；load 用于从内存中加载数据；store 用于将数据存储到内存中。
- **算数运算指令：** 常见的有 add（加法）、sub（减法）、mul（乘法）、div（除法）等。
- **控制流指令：** br 用于分支操作，分为条件分支和无条件分支，ret 用于函数返回，格式为 ret< 返回值类型 >< 返回值 >
- **函数调用指令：** 使用 call 关键字，格式为 call < 返回值类型 > @< 函数名 >< >

我们给出一些例子：

```
%ptr = alloca i32 ; 在栈上分配空间
store i32 10, i32* %ptr ; 存储数据
%val = load i32, i32* %ptr ; 读取数据

%r = call i32 @add(i32 3, i32 5) ; 调用函数
ret i32 %r ; 返回结果
```

基本块 基本块是一段顺序执行的代码，以标签 entry 开始，以结束指令如 br 或者 ret 结束，不同基本块直接通过 br 跳转，基本块是我们进行代码优化和分析的基本单元。

接下来的这个例子，定义了一个名为 basic_block_example 的函数，我们写一些注释，用于解释这个函数的含义。

```
define i32 @basic_block_example(i32 %condition) {
entry:
    ;i1 %condition是布尔值的条件判断，判断我们condition参数的值，
    ;如果为真，则挑战到true_block,为假跳转到flase_block
    br i1 %condition, label %true_block, label %false_block

true_block:
    ;计算加法，存储加法结果到局部变量%result中
    %result = add i32 1, 1
    br label %end_block
false_block:
    %result = add i32 2, 2
    br label %end_block
end_block:
    ret i32 %result
}
```

学习完这些，我们终于可以开始编写 LLVM IR 代码了！

(三) 编写 LLVM IR 代码

我们的 LLVM IR 代码的书写可以分为三个部分。

1. 全局声明与常量

首先我们声明外部函数：getint 和 putint, 声明 SysY 运行时的“读取整数”函数，无参数，返回读取到的 i32 类型整数，输出函数类似，但是没有返回值。

并且，我们声明两个全局变量，都是 int 类型的，对应着我们的数组长度和最小正值。

```
; 外部函数声明
declare i32 @getint()
declare void @putint(i32)

; 全局常量
@SIZE = constant i32 3
@MIN_POSITIVE = constant i32 1
```

2. 自定义函数声明

SysY 中是没有自定义函数, 数组，逻辑运算的，这是我们实现的进阶语言特性，这个函数的主要是接受一个 i32 类型的指针，遍历数组后返回正偶数的累加和。

entry: 函数声明和初始化变量 第一部分是函数的声明，传入的参数为 `i32* %nums`，在 entry 块中，分配分配局部变量%sum（存储累加和）和%i（循环计数器），并分别初始化为 0

之后跳转到 loop 块中。

```
define i32 @sum_even_positive(i32* %nums) {
entry:
    %sum = alloca i32
    store i32 0, i32* %sum
    %i = alloca i32
    store i32 0, i32* %i
    br label %loop
```

loop: 循环条件判断 这一部分对应着我们的 `while(i<size)`,首先加载加载%i 的值作为%i$_val$ $\%i\_val < 3$

之后就用上了我们之前学习的条件跳转指令，当成立，即 `i<size`，则跳转到 body 块, 执行循环体，若不成立，跳转到 `end` 块, 循环结束。

```
loop:
    %i_val = load i32, i32* %i
    %cmp = icmp slt i32 %i_val, 3 ; SIZE = 3
    br i1 %cmp, label %body, label %end
```

body: 循环核心 首先，通过 getelementptr 指令计算当前数组元素地址：以数组基地址 %nums（i32* 类型）和循环下标 %i_val 为输入，自动按 i32 类型的字节长度计算偏移量，生成 nums[i] 对应的指针 %elem_ptr；

之后，用 load 指令从 %elem_ptr 读取元素值，存入寄存器 %elem；

最后，通过 icmp（整数比较）和逻辑指令组合筛选正偶数：判断 %elem >= 1（是否为正数）和 %elem % 2 == 0（是否为偶数），结果存入 %final_cond；根据 %final_cond 跳转——成立则进入 add 块累加，否则进入 skip 块跳过。

```
body:
    %elem_ptr = getelementptr i32, i32* %nums, i32 %i_val
    %elem = load i32, i32* %elem_ptr

    %is_positive = icmp sge i32 %elem, 1 ; nums[i] >= MIN_POSITIVE
    %mod = srem i32 %elem, 2
    %is_even = icmp eq i32 %mod, 0
    %size_eq_1 = icmp eq i32 3, 1 ; SIZE == 1
    %or_cond = or i1 %is_even, %size_eq_1
    %final_cond = and i1 %is_positive, %or_cond
    br i1 %final_cond, label %add, label %skip
```

add: 偶数累加 仅当 %final_cond 为 true（元素是正偶数）时执行：加载当前总和 %sum 的值到 %sum_val，与 %elem 相加得到新总和 %new_sum；将 %new_sum 写回 %sum 对应的内存，完成累加后跳转至 skip 块。

```
add:
    %sum_val = load i32, i32* %sum
    %new_sum = add i32 %sum_val, %elem
    store i32 %new_sum, i32* %sum
    br label %skip
```

skip: 计数器更新 无论是否执行累加，都需通过该块更新循环状态：将循环下标 %i_val 加 1 得到 %i_next，更新 %i 的值；随后跳转回 loop 块，重新判断“是否继续循环”（i < 3）。

```
skip:
    %i_next = add i32 %i_val, 1
    store i32 %i_next, i32* %i
    br label %loop
```

end: 函数返回 当循环遍历完所有数组元素(i >= 3)时进入该块:加载最终的总和值到 %ret_sum，通过 ret 指令将 %ret_sum 作为函数返回值，返回给调用者（main 函数）。

```
end:
    %ret_sum = load i32, i32* %sum
    ret i32 %ret_sum
}
```

3. main 函数实现

main 函数的主要作用是完成输入数组，调用自定义函数，输出结果的过程。我们也分几个部分逐步解释

entry: 初始化变量 首先是初始化：分配一块内存存 3 个整数的数组（%arr），以及一个循环计数器（%i，一开始设为 0）和一个存结果的变量（%total），然后转到 input_loop 块开始让用户输入。

```
define i32 @main() {
entry:
    %arr = alloca [3 x i32]
    %i = alloca i32
    store i32 0, i32* %i
    %total = alloca i32
    br label %input_loop
```

input_loop: 判断输入 判断 %i_val < 3(是否需继续输入),如果条件成立则跳转至 input_body 块读输入，不成立则跳转至 after_input 块结束输入。

```
input_loop:
    %i_val = load i32, i32* %i
    %cmp = icmp slt i32 %i_val, 3
    br i1 %cmp, label %input_body, label %after_input
```

input_body: 读取输入 首先,通过调用 @getint() 读取用户输入(%input),之后用 getelementptr 找 arr[i] 的地址即数组里当前要存的位置，存入后使计数器加一，并跳回 input_loop 继续输入。

```
input_body:
    %input = call i32 @getint()
    %elem_ptr = getelementptr [3 x i32], [3 x i32]* %arr, i32 0, i32 %i_val
    store i32 %input, i32* %elem_ptr
    %i_next = add i32 %i_val, 1
    store i32 %i_next, i32* %i
    br label %input_loop
```

after_input: 调用函数计算和 先用 getelementptr 转成指针 %arr_ptr 找到第一个元素的地址，然后调用之前写的 sum_even_positive 函数，把算出来的正偶数和存到 %total 里。

```
after_input:
    %arr_ptr = getelementptr [3 x i32], [3 x i32]* %arr, i32 0, i32 0
    %sum = call i32 @sum_even_positive(i32* %arr_ptr)
    store i32 %sum, i32* %total
    %total_val = load i32, i32* %total
    %cond = icmp sgt i32 %total_val, 0
    br i1 %cond, label %print_sum, label %print_minus1
```

print 和 ret 最后这一部分是结果的判断和函数的结束，先加载 `%total` 的值，判断 `%total_val > 0`，如果成立跳到 print_sum，如果不成立跳转到 print_minus1。

最后主函数返回 `i32 0`，表示程序结束。

```
print_sum:
    call void @putint(i32 %total_val)
    br label %ret

print_minus1:
    call void @putint(i32 -1)
    br label %ret

ret:
    ret i32 0
}
```

（四） 验证结果

如下所示，这是我们手搓实现的 LLVM IR 代码，接下来我们进行运行和验证。

```
; 外部函数声明
declare i32 @getint()
declare void @putint(i32)

; 全局常量
@SIZE = constant i32 3
@MIN_POSITIVE = constant i32 1

; 自定义函数：计算数组中正偶数和
define i32 @sum_even_positive(i32* %nums) {
entry:
    %sum = alloca i32
    store i32 0, i32* %sum
    %i = alloca i32
    store i32 0, i32* %i
    br label %loop

loop:
    %i_val = load i32, i32* %i
    %cmp = icmp slt i32 %i_val, 3 ; SIZE = 3
    br i1 %cmp, label %body, label %end

body:
    %elem_ptr = getelementptr i32, i32* %nums, i32 %i_val
    %elem = load i32, i32* %elem_ptr
    %is_positive = icmp sge i32 %elem, 1 ; nums[i] >= MIN_POSITIVE
    %mod = srem i32 %elem, 2
    %is_even = icmp eq i32 %mod, 0
    %size_eq_1 = icmp eq i32 3, 1 ; SIZE == 1
```

```
    %or_cond = or i1 %is_even, %size_eq_1
    %final_cond = and i1 %is_positive, %or_cond
    br i1 %final_cond, label %add, label %skip

add:
    %sum_val = load i32, i32* %sum
    %new_sum = add i32 %sum_val, %elem
    store i32 %new_sum, i32* %sum
    br label %skip

skip:
    %i_next = add i32 %i_val, 1
    store i32 %i_next, i32* %i
    br label %loop

end:
    %ret_sum = load i32, i32* %sum
    ret i32 %ret_sum
}

; main函数
define i32 @main() {
entry:
    %arr = alloca [3 x i32]
    %i = alloca i32
    store i32 0, i32* %i
    %total = alloca i32
    br label %input_loop

input_loop:
    %i_val = load i32, i32* %i
    %cmp = icmp slt i32 %i_val, 3
    br i1 %cmp, label %input_body, label %after_input

input_body:
    %input = call i32 @getint()
    %elem_ptr = getelementptr [3 x i32], [3 x i32]* %arr, i32 0, i32 %i_val
    store i32 %input, i32* %elem_ptr
    %i_next = add i32 %i_val, 1
    store i32 %i_next, i32* %i
    br label %input_loop

after_input:
    %arr_ptr = getelementptr [3 x i32], [3 x i32]* %arr, i32 0, i32 0
    %sum = call i32 @sum_even_positive(i32* %arr_ptr)
    store i32 %sum, i32* %total
    %total_val = load i32, i32* %total
    %cond = icmp sgt i32 %total_val, 0
```

```
78     br i1 %cond, label %print_sum, label %print_minus1
79
80 print_sum:
81     call void @putint(i32 %total_val)
82     br label %ret
83
84 print_minus1:
85     call void @putint(i32 -1)
86     br label %ret
87
88 ret:
89     ret i32 0
90 }
```

我们运行命令

```
1 lli -load=./sylib.so even_sum.ll
```

lli 是 LLVM 提供的解释器工具，可以加载 LLVM IR 格式的代码，无需先编译为机器码，并且，我们链接 SysY 的运行时库，这种方式可以快速的测试我们的 LLVM IR 代码。

得到的结果如图17:

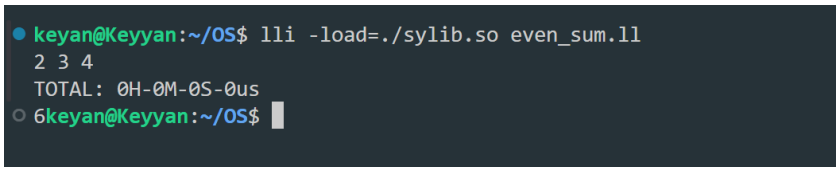


图 17: 运行结果

我们输入 2 3 4，看到输出了 6，证明我们的 LLVM IR 代码编写正确！

五、 任务三：汇编编程

(一) 任务介绍

本次任务即为编写我们设计的 SysY 程序对应的 Arm64 汇编代码，并运行检验正确性。

(二) Arm 架构简要介绍

为了去编写汇编代码，我提前去了解了一下 arm 架构的 ARM64 指令集架构。

1. 核心特点

- **64 位地址：**它支持 64 位虚拟地址空间，同时兼容 32 位数据操作，这就为我们处理 SysY 中的 32 位 int 类型提供了支持。
- **精简指令集：**指令长度固定为 32 位，并且指令功能单一，这就为我们的汇编降低了难度。

2. 关键组件

根据实验指导书中的内容，我们着重了解了 arm64 的寄存器，函数栈帧增长，内存访问等。

寄存器组 ARM64 有 31 个通用 64 位寄存器（X0 X30），可灵活拆分为 32 位使用（对应 W0 W30，适合 SysY 的 int 类型），核心用途如下：

- 参数/返回值寄存器：X0 X7 用于传递函数参数，X0 同时作为函数返回值寄存器。比如 SysY 中 `getint()` 函数的返回值会存于 W0 中。
- 临时寄存器：X9 X15 为临时寄存器，X19 X28 为保存寄存器，函数调用时需保护值，可用于存储 SysY 程序的局部变量如循环计数器、数组下标。
- 栈指针与帧指针：SP（X31）为栈指针，指向栈顶，FP（X29）为帧指针，指向当前函数栈帧底部。

内存访问 支持基地址＋偏移量的灵活寻址，例如访问 SysY 数组 `arr[i]`，我们可以通过 `ldr w1, [x0, w2, lsl 2]` 实现——x0 为数组基地址，w2 为下标 i，lsl #2 表示将 i 左移 2 位（等价于 i*4，计算字节偏移）。

指令

- 算术指令：如 sub，add，srem（取余）等
- 分支指令：b(无条件分支)，cbz(寄存器为 0 则跳转),cmp(比较条件后跳转)，cbnz(寄存器非 0 跳转)。
- 函数调用：bl+ 函数名
- 栈内存读写：str(将寄存器中的值保存到栈地址中)，ldr（从栈中读取数据到寄存器中)。

函数栈增长 ARM64 函数栈遵循向下增长原则，即栈的最高地址为栈顶，低地址为栈底。

当程序启动时，SP 指向栈的初始栈顶，当函数需要局部变量时，SP 向下移动，开辟出空间。当函数执行结束之后，SP 向高地址移动，恢复到函数进入之前的栈顶位置，释放内存。

(三) 编写汇编代码

我们手写 ARM64 汇编代码实现我们之前设计的 SysY 程序。主要分为三部分。

1. 全局声明与初始化

首先声明当前代码段为.text，之后声明 main 为全局符号，再定义 main 的类型为 function 函数，告诉汇编器和链接器，main 是一个函数，需要按照函数的格式进行处理。最后声明 getint 和 putint 为外部函数，他们的实现位于 SysY 运行时库中。

```
1     .section .text
2       .global main
3       .type main, %function
4
5       .extern getint
6       .extern putint
```

2. 主函数 main

我们的主函数负责输入数组，调用自定义函数，输出结果。接下来我们逐步解释。

栈帧初始化 函数进入时需要先初始化栈帧，保存调用者的栈指针 x29 和返回地址 x30, 再进行局部变量的分配。这里我们分配了 16 字节用汉语存储数组，实际上只需要 12 字节就可以，但是为例栈对齐，需要多分配 4 个字节。

```
main:
    stp x29, x30, [sp, -16]! ; 保存帧指针（x29）和链接寄存器（x30，存返回地址），同时栈指针
        （sp）向下移动 16 字节（分配栈空间）
    mov x29, sp ; 更新帧指针（x29）指向当前栈帧底部
    sub sp, sp, #16 ; 栈指针再向下移动 16 字节，分配局部变量空间（存储数组 arr 和临时变量）
    mov w19, 0 ; 初始化循环计数器 i=0（用 w19 寄存器，32 位）
    mov x20, sp ; 保存数组 arr 的基地址（当前栈指针位置，sp 指向 arr[0]）
```

输入循环 我们循环读入 3 个整数，通过 getint 获取输入，用 str 指令存储到数组 arr 的对应位置。

```
input_loop:
    cmp w19, 3 ; 比较 i 与 3
    bge after_input ; 若 i >= 3，跳转到输入结束

    bl getint ; 调用 getint()，返回值存于 w0
    ; 存储输入值到 arr[i]：arr 基地址是 x20，下标 i 左移 2 位（等价于 i*4，因 int 占 4 字
        节）
    str w0, [x20, w19, uxtw #2]
    add w19, w19, 1 ;
    b input_loop ; 跳回循环开始，继续输入
```

调用自定义函数 这一部分实现很简单，调用自定义函数之后，通过 x0 传参，并将返回值保存到 w21 中

```
after_input:
    mov x0, x20 ; 将数组 arr 的基地址存入 x0作为函数参数，
    bl sum_even_positive ; 调用 sum_even_positive
    mov w21, w0 ; 保存返回值到 w21
```

结果输出 我们根据 total 的值判断输出的内容，通过 putint 函数输出结果。其中调用了函数 putint，使用到了分支跳转指令。

```
    cmp w21, 0 ; 比较 total 与 0
    ble print_minus1 ; 若 total <= 0，跳转到输出 -1

print_sum: ; 若 total > 0，输出 total
    mov w0, w21 ; 将 total 存入 w0
    bl putint
    b end_main ; 跳转到程序结束

print_minus1: ; 输出 -1
    mov w0, -1 ; 将 -1 存入 w0
    bl putint
```

清理栈帧 在函数返回之前，我们需要清理栈帧，释放局部变量的空间，即让栈指针上移，回复调用的栈指针，最后通过 ret 返回。

```
end_main:
       mov w0, 0
       add sp, sp, #16
       ldp x29, x30, [sp], 16
       ret
```

3. 自定义函数

我们的自定义函数的主要作用是接受一个数组，便利数组之后并累加其中的正偶数。
我们也是分为几个部分来写。

栈帧初始化 与 main 函数类似，先初始化栈帧，在定义局部变量 i 和 sum，并且保存了一个数组基地址。

```
sum_even_positive:
    stp x29, x30, [sp, -16]! ; 保存帧指针和返回地址，分配 16 字节栈空间
    mov x29, sp ; 更新帧指针
    sub sp, sp, #16 ; 再分配 16 字节栈空间（局部变量）
    mov w19, 0 ;i=0
    mov w20, 0 ;sum=0
    mov x21, x0 ; 保存数组基地址
```

循环遍历数组 在我们的循环开始，需要比较 i 与 3 的大小，如果 i>3，则跳转到循环结束，如果 i<3，则进入循环，读取数组的值。

```
loop_start:
       cmp w19, 3
       bge loop_end

       ldr w22, [x21, w19, uxtw #2] // elem = nums[i]
```

筛选正偶数 对于整数的判断，我们比较数组元素与 1，如果非正，则跳转到 skip_elem; 对于偶数的判断，用与运算 and，结果是 0 的话则是偶数。并且我们需要设定一个逻辑，当数组的长度是 1 的话，也累加，在我们的程序中不出现。

```
       cmp w22, 1
       blt skip_elem

       and w23, w22, 1
       cmp w23, 0
       beq add_elem
       mov w24, 3
       cmp w24, 1
       beq add_elem
       b skip_elem
```

累加与更新 经过上述条件的判断，我们累加正偶数，并且将循环计数器 +1，最后通过跳转指令跳回到循环开始。

```
add_elem:
        add w20, w20, w22

skip_elem:
        add w19, w19, 1
        b loop_start
```

函数返回 在循环结束后，我们将累加的 sum 作为返回值，存到 w0 中，释放局部变量的栈空间，返回栈指针和返回地址，最后返回给调用者 main 函数。

```
loop_end:
        mov w0, w20
        add sp, sp, #16
        ldp x29, x30, [sp], 16
        ret
```

（四） 验证结果

完整的汇编代码如下：

```
        .section .text
        .global main
        .type main, %function

        .extern getint
        .extern putint

main:
        stp x29, x30, [sp, -16]!
        mov x29, sp

        sub sp, sp, #16 // 局部变量空间
        mov w19, 0 // i = 0
        mov x20, sp // arr 数组地址

input_loop:
        cmp w19, 3
        bge after_input

        bl getint
        str w0, [x20, w19, uxtw #2] // arr[i] = getint()
        add w19, w19, 1
        b input_loop

after_input:
        mov x0, x20
```

```
27        bl sum_even_positive
28        mov w21, w0 // total
29
30        cmp w21, 0
31        ble print_minus1
32
33  print_sum:
34        mov w0, w21
35        bl putint
36        b end_main
37
38  print_minus1:
39        mov w0, -1
40        bl putint
41
42  end_main:
43        mov w0, 0
44        add sp, sp, #16
45        ldp x29, x30, [sp], 16
46        ret
47
48        .global sum_even_positive
49        .type sum_even_positive, %function
50  sum_even_positive:
51        stp x29, x30, [sp, -16]!
52        mov x29, sp
53        sub sp, sp, #16
54
55        mov w19, 0 // i = 0
56        mov w20, 0 // sum = 0
57        mov x21, x0 // nums
58
59  loop_start:
60        cmp w19, 3
61        bge loop_end
62
63        ldr w22, [x21, w19, uxtw #2] // elem = nums[i]
64        cmp w22, 1
65        blt skip_elem
66
67        and w23, w22, 1
68        cmp w23, 0
69        beq add_elem
70        mov w24, 3
71        cmp w24, 1
72        beq add_elem
73        b skip_elem
74
```

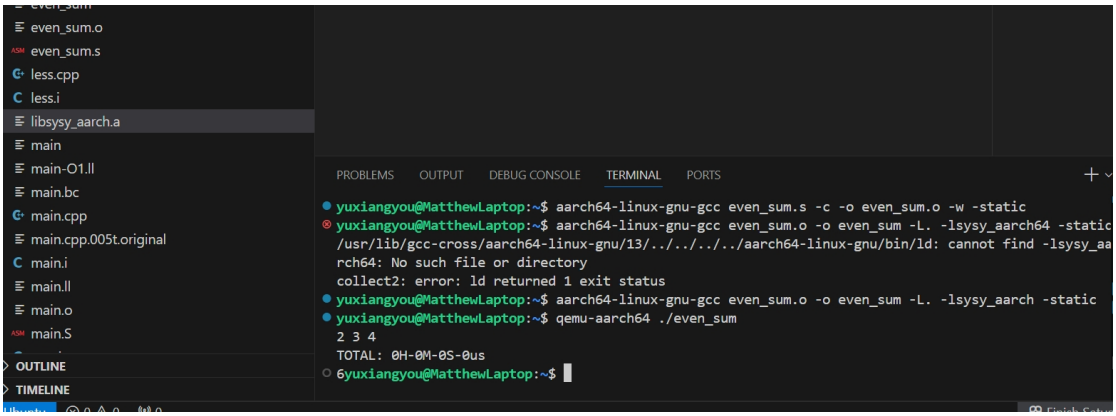
```
add_elem:
        add w20, w20, w22

skip_elem:
        add w19, w19, 1
        b loop_start

loop_end:
        mov w0, w20
        add sp, sp, #16
        ldp x29, x30, [sp], 16
        ret
```

编写完上述汇编代码之后，我们采用以下命令进行测试。

首先，我们使用交叉编译链将我们编写的汇编文件转成目标文件，只执行编译，不进行链接，之后，我们采用交叉编译器生成 ARM64 格式的程序，链接 SysY 静态库，生成可执行文件，之后采用 qemu 模拟器，在我们的 x86 电脑上，模拟 ARM64 指令的执行。

```
aarch64-linux-gnu-gcc even_sum.s -c -o even_sum.o -w -static
aarch64-linux-gnu-gcc even_sum.o -o even_sum -L. -lsysy_aarch -static
qemu-aarch64 ./even_sum
```

运行结果如图所示：



我们输入 **2,3,4, 得到了结果 6，证明我们的汇编程序正确！**

六、 C 语言编程

鉴于我们在前面的实验选择了阶乘作为我们的实验目标，那我们就尝试用汇编的方式再复现一遍代码。

首先我们还是先回顾下之前的 main.cpp 代码的全部代码：

```
#include<iostream>
using namespace std;

int main()
{
```

```
    int i,n,f;
    cin >>n;

    i = 2;
    f = 1;
    while(i<=n)
    {
        f=f*i;
        i=i+1;
    }
    cout << f << endl;

    return 0;
}
```

main.cpp 回顾

既然我们要尝试编写阶乘程序的汇编代码的话，我们就需要先明白这个代码具体做了什么事情，大概是：

1. 先是定义三个 Int 型的变量——i,n,f;
2. 获取 n 的输入;
3. 然后是对 i 和 f 进行赋值;
4. 然后是一个 while 循环——同时要进行 i 与 n 的 <= 的比较;
5. while 循环内会进行两步的赋值——f=f*i 与 i=i+1;
6. 然后是输出 f 以及换行;
7. 最后是 return 0.

然后就是阅读指导书给出的 arm 汇编代码的教程，一个正常的程序的实现顺序应该是：

```
@ 程序结构大纲：
@ 1. 数据段：定义字符串常量
@ 2. BSS段：定义变量
@ 3. text段：文件以及符号声明
@ 4. 代码段：
@   4.1 函数序言（保存寄存器）
@   4.2 输入处理
@   4.3 循环计算
@   4.4 输出结果
@   4.5 函数收尾
```

那接下来就一块块地去实现 arm 的汇编代码即可，按照顺序应该分别是：
第一步就是实现数据段（.data）的定义：定义程序中会使用到的常量 &string 的数据

```
@ 数据段定义 .data 定义程序中会使用到的常量\&string的数据
.section .data
@ 对于cin>>n的时候，输出的提示词prompt
prompt: .asciz "Enter a number: "
@ 指定输入格式，获取输入为%d，整数
input_format: .asciz "%d"
@ 指定输出格式，输出带%n自动换行
output_format: .asciz "%d\n"
```

arm 汇编代码

第二步就是实现未初始化数据段（.bss），用来存储未初始化的变量，一般预留 4B 作为适合的大小；

```
@ section.bss 存储未初始化的变量
.section .bss
@ 为int类型变量n预存4B的空间
.lcomm n, 4
@ 为int类型变量result预存4B的空间
.lcomm result, 4
```

arm 汇编代码

第三步就是 text 段的部分了，我们要在这一块区域声明 main 函数入口——就是 main 这个符号，然后声明函数 scanf 以及 printf 是别处定义的，为 extern 函数。

```
@ 把接下来代码放到可执行的代码段——text段
.section .text
@ 导出main符号，告诉链接器这个就是程序的入口
.global main
@ extern表示告诉链接器，scanf和printf是外来函数，需要去别的地方找到这两个函数的具体代码
.extern scanf, printf
```

arm 汇编代码

第四步就是代码段的部分了，我们还是一点点进行讲解：

先是 4.1：函数序言的部分，这一部分负责保存函数的返回地址以及需要用的寄存器，并且建立栈帧：

```
main:
    @ 函数序言——保存寄存器
     @ 保存需要使用的寄存器r4,r5和返回地址lr,此处fp是旧的栈指针
    push {r4, r5, fp, lr}
    @ 设置帧指针,将sp的值移动到fp,就是挪动了栈帧指针
    mov fp, sp
```

arm 汇编代码

然后是 4.2: 输入处理的部分，这部分负责处理 cin»n 这样的语句，获取到 n 的实际值：

```
    @ 把提示词prompt的地址加载到r0，就是获取提示信息的内容
    ldr r0, =prompt
```

```
    @ 输出对应的提示信息提示用户输入
    bl printf
    @ 把input_format放在r0，这个input_format在第一段.data有提及过，其实就是把\%d字符串地址
        放到r0，作为scanf第一个参数
    ldr r0, =input_format
    @ 把变量n地址放在r1,作为scanf第二个参数
    ldr r1, =n
    @ 最后调用scanf，从键盘获取输入n
    bl scanf
```

arm 汇编代码

然后是 4.3: 循环计算的部分，这一块就是具体实现 while 循环的汇编代码:

```
    @ 就是一个简单的赋值语句，r4=2实际上就是i = 2
    mov r4, #2
    @ 同上,等效于f=1
    mov r5, #1
    @ 把变量n的地址加载到寄存器r2
    ldr r2, =n
    @ 取出n的值，并且放到r2中，方便后续loop是否继续进行提供判断
    ldr r2, [r2]

loop_start:
    @ 比较i(r4)和n(r2)的大小
    cmp r4, r2
    @ 若i>n，则转到loop_end，实现退出循环
    bgt loop_end
    @ f=f*i，将f值更新为f*i
    mul r5, r5, r4
    @ i=i++，更新r4值
    add r4, r4, #1
    @ 回到循环开头，若i>n则会跳转到loop_end
    b loop_start
```

arm 汇编代码

再然后是 4.4: 输出结果的部分，这个就很名副其实了，就是负责处理 loop_end 后如何去打印结果 f 了:

```
loop_end:
    @ 把result的地址加载到r3
    ldr r3, =result
    @ 把最终结果f存入result
    str r5, [r3]
    @ printf的格式字符串"%d\n"放到r0，作为第一个参数
    ldr r0, =output_format
    @ 把f的值放到r1，作为第二个参数
    mov r1, r5
    @ 调用printf，输出结果
    bl printf
```

```
```

arm 汇编代码

最后就是 4.5: 函数收尾的部分了，这部分主要就是设置 main 函数的返回值 0 以及将存进去的寄存器 pop 出来并 pop 出返回地址:

```
    @ 设置返回值r0=0，对应return 0;
    mov r0, #0
    @ pop出保存了的寄存器r4以及r5and返回地址bx lr
    pop {r4, r5, fp, lr}
    @ 跳转到lr指定的位置，返回调用main函数的部分，即退出main函数
    bx lr;
```

arm 汇编代码

然后完整代码如下:

```
@ 数据段定义 .data 定义程序中会使用到的常量\&string的数据
.section .data
@ 对于cin>>n的时候，输出的提示词prompt
prompt: .asciz "Enter a number: "
@ 指定输入格式，获取输入为%d，整数
input_format: .asciz "%d"
@ 指定输出格式，输出带%n自动换行
output_format: .asciz "%d\n"

@ section.bss 存储未初始化的变量
.section .bss
@ 为int类型变量n预存4B的空间
.lcomm n, 4
@ 为int类型变量result预存4B的空间
.lcomm result, 4

@ 把接下来代码放到可执行的代码段——text段
.section .text
@ 导出main符号，告诉链接器这个就是程序的入口
.global main
@ extern表示告诉链接器，scanf和printf是外来函数，需要去别的地方找到这两个函数的具体代码
.extern scanf, printf

main:
    @ 函数序言——保存寄存器
     @ 保存需要使用的寄存器r4,r5和返回地址lr,此处fp是旧的栈指针
    push {r4, r5, fp, lr}
    @ 设置帧指针,将sp的值移动到fp,就是挪动了栈帧指针
    mov fp, sp
    @ 把提示词prompt的地址加载到r0，就是获取提示信息的内容
    ldr r0, =prompt
     @ 输出对应的提示信息提示用户输入
    bl printf
```

```
    @ 把input_format放在r0，这个input_format在第一段.data有提及过，其实就是把\%d字符串地址
        放到r0，作为scanf第一个参数
    ldr r0, =input_format
    @ 把变量n地址放在r1,作为scanf第二个参数
    ldr r1, =n
    @ 最后调用scanf，从键盘获取输入n
    bl scanf
    @ 就是一个简单的赋值语句，r4=2实际上就是i = 2
    mov r4, #2
     @ 同上,等效于f=1
    mov r5, #1
    @ 把变量n的地址加载到寄存器r2
    ldr r2, =n
     @ 取出n的值，并且放到r2中，方便后续loop是否继续进行提供判断
    ldr r2, [r2]

loop_start:
    @ 比较i(r4)和n(r2)的大小
    cmp r4, r2
     @ 若i>n，则转到loop_end，实现退出循环
    bgt loop_end
    @ f=f*i，将f值更新为f*i
    mul r5, r5, r4
     @ i=i++，更新r4值
    add r4, r4, #1
    @ 回到循环开头，若i>n则会跳转到loop_end
    b loop_start

loop_end:
    @ 把result的地址加载到r3
    ldr r3, =result
    @ 把最终结果f存入result
    str r5, [r3]
    @ printf的格式字符串"%d\n"放到r0，作为第一个参数
    ldr r0, =output_format
     @ 把f的值放到r1，作为第二个参数
    mov r1, r5
     @ 调用printf，输出结果
    bl printf
    @ 设置返回值r0=0，对应return 0;
    mov r0, #0
    @ pop出保存了的寄存器r4以及r5and返回地址bx lr
    pop {r4, r5, fp, lr}
    @ 跳转到lr指定的位置，返回调用main函数的部分，即退出main函数
    bx lr;
```

arm 汇编代码完整版

那至此我们就完成了一个完整的汇编代码的编写，为了验证这个汇编代码的正确性，我使用

了在线网站 **https://cpulator.01xz.net/?sys=arm-de1soc** 来进行 arm 平台汇编代码的编译，编译成功，结果如下图：

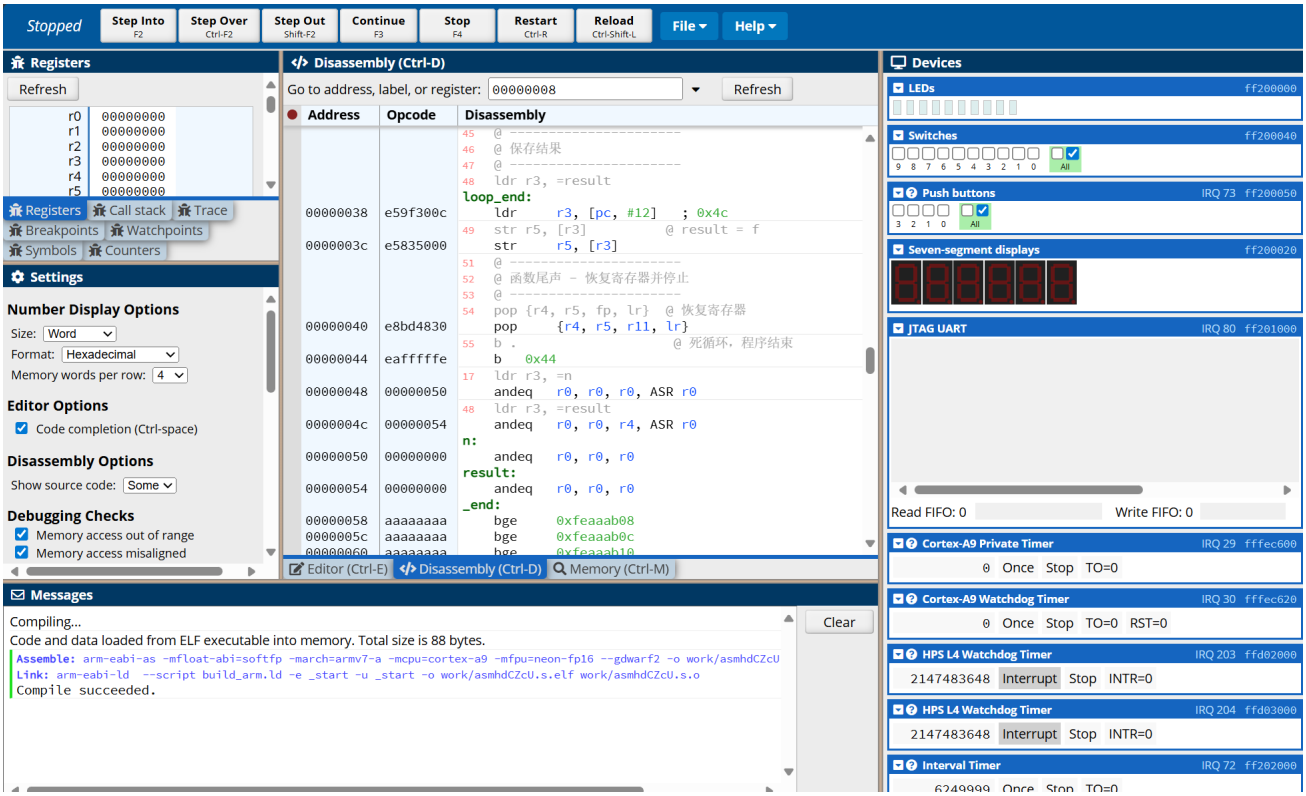


图 18: 网站编译图片

然后我们取值 n=5，此时应该结果为 5!=120，让我们看看运行结果：

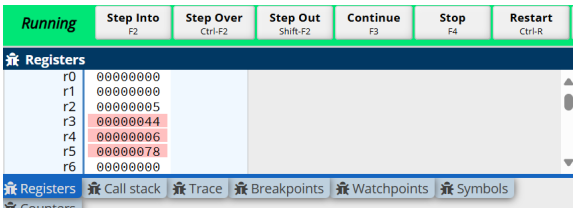


图 19: n=5 运行结果

我们可以发现：寄存器 r2 内就是我们的 n 值 =5，r5 存放着我们的结果 0x78，换算成十进制就是结果 120，证明阶乘实现无误，至此完成！

七、 总结

通过本次实验，我对程序从源代码到可执行文件的整个编译过程有了比较清晰的认识。实验内容包括预处理、编译、汇编、链接，以及加载器将程序载入内存执行的全过程。通过亲手操作，我对每一步在程序生成中的作用理解得更直观。

在本次实验中，我尝试直接使用汇编语言实现 sysy 程序的功能，而不是依赖高级语言。程序的逻辑包括常量定义、数组输入、自定义函数、循环判断以及条件分支等。将这些特性用汇编实现的过程，让我对底层程序运行机制有了更加直观的认识。

在编写过程中,我需要手动完成许多高级语言中自动处理的细节。比如在函数 `sum_even_positive` 中，我必须明确设置栈帧，分配局部变量空间，并通过寄存器传递参数与保存返回值；在实现 `while` 循环时，则需要通过比较指令和条件跳转来构造循环结构；在 `if-else` 语句中，同样需要使用跳转指令和标签来控制不同分支的执行流程。此外，数组访问也需要手动计算偏移量，这让我深刻体会到高级语言对内存操作的封装。

编写过程中我也遇到不少困难。例如，最初我在函数返回值处理上出现了错误，导致主函数无法正确获得结果。经过反复调试，我意识到需要在函数结束前正确恢复栈帧，并把返回值存放在约定的寄存器中。另一个问题是循环标签跳转混乱，我通过在代码中增加中间标签并仔细比对执行流程，才逐步理清逻辑。

总体而言，这次实验让我认识到高级语言代码背后的底层实现。手写汇编的过程虽然繁琐，但也让我更好地理解了编译器在目标代码生成阶段的工作。通过亲手完成 sysy 程序的汇编实现，我不仅掌握了寄存器、栈帧、跳转指令等底层机制，也对程序运行的本质有了更深刻的认识。

实验中，我尝试用 ARM 汇编语言手动实现一个阶乘程序。写代码、调试、运行的过程中，我对寄存器的使用、栈操作和指令执行顺序有了直观感受，也更能理解高级语言背后的底层逻辑。虽然写起来有些繁琐，但每次调试成功都能感受到很大的成就感。

总体来说，这次实验让我收获不少。既巩固了课堂上学到的理论知识，也通过动手实践加深了对编译原理的理解。面对底层程序的调试和运行，我对计算机的执行机制有了更具体的认识，也增加了我今后尝试类似实验的信心。

参考文献