



南开大学  
Nankai University

南 开 大 学

计 算 机 学 院

编译系统原理实验报告

---

Lab0：熟悉编译器功能，掌握 LLVM IR 和汇编编程

---

陈语童

年级：2023 级

学号：2311887

专业：计算机科学与技术

课程教师：李忠伟

2025 年 9 月 29 日

## 摘要

本实验由小组成员两人共同完成。概述部分介绍了实验目的以及其他实验前相关事宜。实验一主要对本课程主要学习的编译流程进行深入探究，研究单位包括预处理器、编译器、汇编器、链接器等，着重分析了编译器的各个步骤所做工作。实验二设计 SysY 程序，并编写等价 LLVM IR 中间语言程序，相关工作包括分析编写思路、验证代码正确性等。实验三沿用实验二中的 SysY 程序，编写等价的 ARM 汇编程序，相关工作与实验二种类相近。最后对本次实验进行总结，记录收获心得。

**关键字：**预处理，编译器，词法分析，语法分析，语义分析，中间代码，代码优化，汇编器，链接器，SysY 编程，LLVM IR 编程，ARM 汇编编程

# 目录

<b>一、 概述</b>	<b>1</b>
(一) 实验目的 . . . . .	1
(二) 小组分工 . . . . .	1
(三) 实验环境 . . . . .	1
<b>二、 实验一：深入了解编译流程和编译器功能</b>	<b>2</b>
(一) 总流程 . . . . .	2
(二) 预处理器 . . . . .	4
1. 预处理的具体功能 . . . . .	4
2. 进行预处理并验证功能 . . . . .	4
(三) 编译器 . . . . .	6
1. 词法分析 . . . . .	7
2. 语法分析 . . . . .	11
3. 语义分析 . . . . .	15
4. 中间代码生成 . . . . .	15
5. 代码优化 . . . . .	20
6. 代码生成 . . . . .	23
(四) 汇编器 . . . . .	23
1. 汇编器的具体功能 . . . . .	23
2. 反汇编与汇编程序分析 . . . . .	24
(五) 链接器 . . . . .	27
(六) 运行程序 . . . . .	27
<b>三、 实验二：LLVM IR 编程</b>	<b>28</b>
(一) SysY 程序编写 . . . . .	28
(二) LLVM IR 编程 . . . . .	31
<b>四、 实验三：ARM 汇编编程</b>	<b>39</b>
(一) SysY 程序 . . . . .	39
(二) ARM 汇编代码设计过程 . . . . .	39
(三) ARM 汇编代码综合展示 . . . . .	45
(四) 正确性验证 . . . . .	45
(五) 思考与启发 . . . . .	46
<b>五、 总结</b>	<b>46</b>

## 一、 概述

### (一) 实验目的

本次实验主要目的是以一个简单的斐波那契数列程序为例, 通过调整编译器的程序选项获得编译器工作过程中的各阶段输出, 借此来探究:

1. 完整的编译过程包含哪些阶段?
2. 预处理器、编译器、汇编器、链接器在这个过程中的功能是什么?
3. 熟悉编译器生成的 LLVM IR 中间语言和汇编目标语言。

更进一步, 我们通过使用 LLVM IR 语言和 ARM 汇编语言编写简单的程序实现 SysY 程序支持的各种语言特性, 来掌握基本的 LLVM IR 语言和 ARM 汇编语言的特点。

### (二) 小组分工

小组成员: 陈语童 (2311887), 强博 (2313825)。

实际实验内容之前的部分, 包括摘要 + 关键字和概述的绝大部分, 由两人共同撰写。

**实验一: 深入了解编译器功能** 由两人各自独立完成实验探究和报告撰写。

**实验二: LLVM IR 编程** 由强博 (2313825) 一人完成实验探究和报告撰写。

**实验三: ARM 汇编编程** 由陈语童 (2311887) 一人完成实验探究和报告撰写。

以上未提及的报告内容撰写部分, 均由两人各自独立完成。

陈语童 (2311887) 的实验相关代码仓库: [NKU-compiler-basics/Lab0](https://github.com/NKU-compiler-basics/Lab0)

强博 (2313825) 的实验相关代码仓库: [NKU-Compiling-2025-Lab/LAB1](https://github.com/NKU-Compiling-2025-Lab/LAB1)

### (三) 实验环境

上一学期某一课程实验已经在本 64 位 x86 机器上配置 Ubuntu 虚拟环境, 系统型号系: Ubuntu 24.04.1 LTS. 安装并配置好本实验需要的新的资源包即可完成环境配置。系统和发行版信息:

```
bugp3ssy666@DESKTOP-ALUTGKM:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 24.04.1 LTS
Release:        24.04
Codename:       noble
```

图 1: 系统和发行版信息

以及内核架构信息:

```
bugp3ssy666@DESKTOP-ALUTGKM:~$ uname -a
Linux DESKTOP-ALUTGKM 6.6.87.2-microsoft-standard-WSL2 #1 SMP PR
EEMPT_DYNAMIC Thu Jun  5 18:30:46 UTC 2025 x86_64 x86_64 x86_64
GNU/Linux
```

图 2: 内核架构信息

上图说明了本虚拟系统为在 Windows 的 WSL2 下运行的 Linux 内核；运行的 CPU 架构、内核架构、硬件平台类型均为“x86\_64”，即在 64 位 x86 CPU 上运行 64 位内核；操作系统环境为“GNU 工具链 + Linux 内核”。

## 二、 实验一：深入了解编译流程和编译器功能

### (一) 总流程

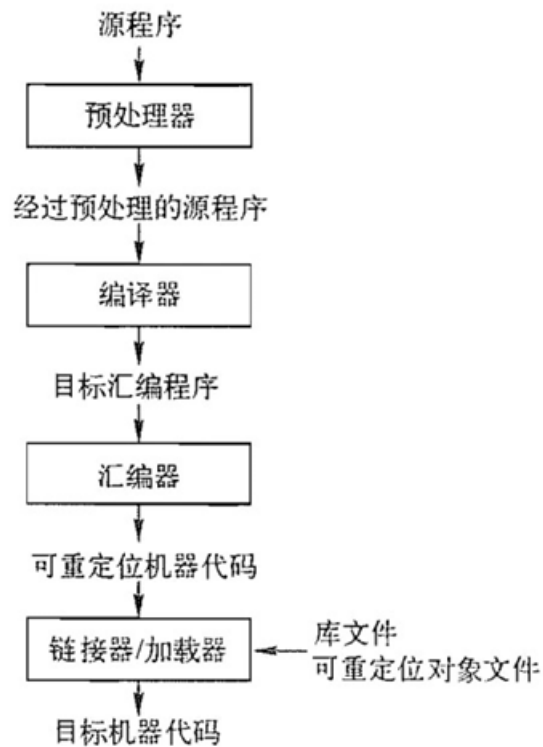


图 3: 完整编译流程图

本课题研究 C 程序的编译过程，完整编译流程如上图所示。简要说，每个阶段的主要作用如下：

1. **预处理器**：处理源代码所有以 **# 开头的预编译指令**（`#include` `#define` `#if` ...）
2. **编译器**：将预处理后的 C 代码翻译成**汇编语言**。
3. **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**可重定位的目标程序**。
4. **链接器**：将可重定位的目标程序（可以是多个）引入**库文件**，生成最终可运行的目标代码。
5. **加载器**：加载资源文件到虚拟内存合适地址，预备启动程序。

为了逐步验证各个阶段产生的作用，由实验提供的简单程序为出发点，作以下两点调整：

- C 语言不支持 `cin`, `cout` 等**输入输出流**, 将此类指令改为 C 语言适用的指令, 如 `printf()`, `scanf()` 等等。
- 加入一些**预编译指令、注释**, 为预处理器阶段验证作补充。

修改后的 C 语言代码如下所示:

一个简单的斐波那契数列 C 程序

```
1 #include <stdio.h>
2
3 #define MAX_N 50
4 #define INIT_A 0
5 #define INIT_B 1
6
7 #define PRINT(x) printf("%d\n", x)
8
9 int main() {
10     int a, b, i, t, n;
11
12     a = INIT_A;
13     b = INIT_B;
14     i = 1;
15
16     scanf("%d", &n);
17     if (n > MAX_N) {
18         printf("Input exceeds maximum value %d\n", MAX_N);
19         return 1;
20     }
21
22     PRINT(a);
23     PRINT(b);
24
25     // this is a friendly annotation *
26     while (i < n)
27     {
28         t = b;
29         b = a + b;
30         PRINT(b);
31         a = t;
32         i = i + 1;
33     }
34 }
```

下面按照步骤编译, 依次验证和探究各个步骤做了什么工作。

## (二) 预处理器

### 1. 预处理的具体功能

**预处理**是 C/C++ 程序**从源代码到编译**的第一步。在这一阶段，编译器的预处理器（cpp）只处理 # 开头的指令以及注释，对源代码进行文本级别的加工，生成一个“纯净”的源文件，交由编译器进行后续分析。

更加具体来说，其主要工作有：

1. **头文件展开**：处理 #include 指令，把指定的头文件内容**直接拷贝**进来。
2. **宏替换/宏展开**：处理 #define 定义的宏，其中对于**对象宏**将其用原先的简单文本替换掉，对于**函数宏**则用带参数的文本替换。
3. **条件编译**：根据条件来**选择性**编译所需要部分的代码，相关常见指令有 #if, #ifdef, #ifndef, #elif, #else, #endif。
4. **注释删除**：所有 /\* ... \*/ 和 // ... 诸如此类的注释信息都会被删除，不进入编译阶段。
5. **\* 其他处理**：其他一些预处理指令的处理，如 #undef 可以取消一个宏定义，#line 指令可以改变编译器记录的行号和文件名，#error 会强制在预处理时报错，#warning 会发出警告（某些编译器支持），等等。
6. **\* 特殊宏替换**：由编译器提供的特殊宏替换，有 \_\_FILE\_\_, \_\_LINE\_\_, \_\_DATE\_\_, \_\_TIME\_\_ 等，不作详细介绍。

在本实验的研究当中，前四点作用是主要研究对象。

### 2. 进行预处理并验证功能

在实验目录下终端中输入以下命令：

```
gcc main.c -E -o main.i
```

对于该命令的理解是：添加参数-E 让 gcc **只进行预处理**过程；参数-o 改变 gcc 输出文件名。  
.i 是标准的预处理后源码文件格式。

执行命令后，观察到目录下新生成一个文件 main.i，说明预处理进行成功。打开进行其功能验证，其中发现：

① 文件中出现诸多以下类型的代码段：

```
13 # 1 "/usr/include/features-time64.h" 1 3 4
14 # 20 "/usr/include/features-time64.h" 3 4
15 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
16 # 21 "/usr/include/features-time64.h" 2 3 4
17 # 1 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 1 3 4
18 # 19 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 3 4
19 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
20 # 20 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 2 3 4
21 # 22 "/usr/include/features-time64.h" 2 3 4
22 # 395 "/usr/include/features.h" 2 3 4
23 # 502 "/usr/include/features.h" 3 4
24 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
25 # 576 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
```

图 4: 预处理器插入的行控制指令

```

124 typedef unsigned long int __nlink_t;
125 typedef long int __off_t;
126 typedef long int __off64_t;
127 typedef int __pid_t;
128 typedef struct { int __val[2]; } __fsid_t;
129 typedef long int __clock_t;
130 typedef unsigned long int __rlim_t;
131 typedef unsigned long int __rlim64_t;
132 typedef unsigned int __id_t;
133 typedef long int __time_t;

```

图 5: 大量定义类型别名的指令

```

341 typedef struct _IO_cookie_io_functions_t
342 {
343     cookie_read_function_t *read;
344     cookie_write_function_t *write;
345     cookie_seek_function_t *seek;
346     cookie_close_function_t *close;
347 } cookie_io_functions_t;
348 # 48 "/usr/include/stdio.h" 2 3 4

```

图 6: 结构体的定义与声明

```

385 extern int renameat (int __oldfd, const char *__old, int __newfd,
386                     const char *__new) __attribute__((__nothrow__, __leaf__));
387 # 184 "/usr/include/stdio.h" 3 4
388 extern int fclose (FILE *__stream) __attribute__((__nonnull__(1)));
389 # 194 "/usr/include/stdio.h" 3 4
390 extern FILE *tmpfile (void)
391     __attribute__((__malloc__)) __attribute__((__malloc__(fclose, 1)));
392 # 211 "/usr/include/stdio.h" 3 4
393 extern char *tmpnam (char[20]) __attribute__((__nothrow__, __leaf__));

```

图 7: 各种函数体声明

其中对于图 4 的指令可以进一步分析：

# < 行号 > "< 文件名 >" < 可选标志 >

其中 < 行号 > 告诉编译器“接下来的代码在源文件里属于哪一行”；< 文件名 > 告诉编译器“这段代码来自哪个源文件”；而 < 可选标志 > 数字是一些特殊标记，用于区分文件种类和状态。其实 < 行号 > 信息在其他指令部分也都有出现。

以上几种代码段基本都来源于 #include 头文件包含的处理，加之以对源代码的解读（涉及到行号信息的获取）。所有声明都是对头文件内容的直接或间接拷贝。这些声明占了约 800 行，可知 <stdio.h> 包还是很庞大的（包括其内部再引用别的头文件）。

② 在文件最后部分，出现了与原先 C 程序代码相似度很高的代码段：

预处理后的 C 程序代码

```

1 # 9 "main.c"
2 int main() {

```



```
3      int a, b, i, t, n;
4
5      a = 0;
6      b = 1;
7      i = 1;
8
9      scanf("%d", &n);
10     if (n > 50) {
11         printf("Input exceeds maximum value\n", 50);
12         return 1;
13     }
14
15     printf("%d\n", a);
16     printf("%d\n", b);
17
18
19     while (i < n)
20     {
21         t = b;
22         b = a + b;
23         printf("%d\n", b);
24         a = t;
25         i = i + 1;
26     }
27 }
```

这里出现了 # 9 的行号信息，回到[原始代码](#)中，可以观察到 main 函数的确是从第 9 行开始的，可以验证行号信息的正确性。

综合分析这段代码，会发现是和原代码[逻辑一致且正确](#)的“简单斐波那契数列”程序，但有一些细节上的改变：

1. **头文件引用消失：**由上面的分析可知，头文件具体内容已经被全部拷贝至此文件前面部分，所以不再需要声明。
2. **宏定义被替换：**源代码中对参数初始值以及输出函数（printf() -> PRINT()）进行了宏定义（#define），主函数代码中直接使用替换符；而预处理后的代码删除了宏定义，且将替换符全部替换为原始数据或函数。
3. **注释被删除：**源代码添加了一行// ... 注释作为测试，在预处理后的代码中没有出现，说明注释被删除。

这些变化和预处理的各项既定功能是相符合的，则[预处理器功能验证完毕](#)。

### （三） 编译器

编译过程是本课程着重学习和研究的过程。它接收[源代码](#)文件，输出[汇编代码](#)。其过程中主要完成的工作有：

- **翻译：**将原来的高级语言代码翻译成机器能够进一步处理的[汇编代码](#)。

- **检查：**检查原先代码逻辑中的语法错误、语义错误并反馈。
- **优化：**改进程序结构或指令，使生成的代码运行更快或占用更少资源。

接下来逐步分析编译器处理各个阶段完成的工作。

## 1. 词法分析

词法分析：主要工作是把源代码的字符流分割成 **token** 序列（关键字、标识符、常量、操作符、分隔符……），促进后续对于代码结构和语法的理解。

进行词法分析仍从源 C 代码出发，执行以下命令：

```
clang -E -Xclang -dump-tokens main.c
```

命令调用了 Clang 编译器前端。其中参数-E 依旧代表只进行预编译；参数-Xclang 用来告诉 Clang 驱动程序，后面跟着的文件内容直接传给 Clang 前端，而非驱动器自己处理；-dump-tokens 是核心意义的参数，它是 Clang 前端的一个内部调试选项，能够在**词法分析**阶段打印出**所有的 token**。

词法分析的结果会直接在终端打印出来（当然也可以使用 shell 重定向，导出为文本文件），可以截取输出结果中的实际逻辑部分进行分析，探究其含义。

词法分析片段 1

```
1 int 'int'          [StartOfLine]  Loc=<main.c:9:1>
2 identifier 'main'   [LeadingSpace] Loc=<main.c:9:5>
3 l_paren '('         Loc=<main.c:9:9>
4 r_paren ')'         Loc=<main.c:9:10>
5 l_brace '{'         [LeadingSpace] Loc=<main.c:9:12>
```

这一片段对应源代码的：

```
1 int main() {
```

这一行。int 是**关键词 token**，直接被标记为“int”，而 [StartOfLine] 则表示这一关键词出现在该代码行之首（此标记不再说明），Loc 段表示其在源代码文件中的**起始**具体位置，包括文件、行与列信息（例如，Loc=<main.c:9:1> 表示此语句从 main.c 中第 9 行、第 1 列/第 1 个字符起始。此后，Loc 的含义都如此，不再赘述）。对于 main，将其识别为 identifier，即**标识符**，并且冠以 [LeadingSpace] 标记，说明前面有空格（此标记不再说明）。对于三个字符，l\_paren, r\_paren, l\_brace 的 token 命名很直接，分别表示“左括号”“右括号”“左大括号”。

由上面的分析可知，一般对于一个 token 的分析结果有这几个组成成分：

① **token 种类** + ② **'token 内容'** + ③ **[附加信息标记（可选）]** + ④ **Loc=< 源代码位置 >**

有了以上的分析经验，不难将剩下的词法分析结果对应到相应的源代码：

词法分析片段 2

```
1 int 'int'          [StartOfLine] [LeadingSpace]  Loc=<main.c:10:5>
2 identifier 'a'     [LeadingSpace] Loc=<main.c:10:9>
3 comma ',',         Loc=<main.c:10:10>
```

```

4 identifier 'b'      [LeadingSpace] Loc=<main.c:10:12>
5 comma ','          Loc=<main.c:10:13>
6 identifier 'i'      [LeadingSpace] Loc=<main.c:10:15>
7 comma ','          Loc=<main.c:10:16>
8 identifier 't'      [LeadingSpace] Loc=<main.c:10:18>
9 comma ','          Loc=<main.c:10:19>
10 identifier 'n'      [LeadingSpace] Loc=<main.c:10:21>
11 semi ';'          Loc=<main.c:10:22>

```

这一部分对应源代码的：

```

1 int a, b, i, t, n;

```

这一行，是声明全局变量的语句。其中新出现了 comma, semi 两个 token，分别对应源码的逗号和分号分隔符。

### 词法分析片段 3

```

1 identifier 'a'      [StartOfLine] [LeadingSpace] Loc=<main.c:12:5>
2 equal '='          Loc=<main.c:12:7>
3 numeric_constant '0' [LeadingSpace] Loc=<main.c:12:9 <Spelling=main.c
   :4:16>>
4 semi ';'          Loc=<main.c:12:15>
5 identifier 'b'      [StartOfLine] [LeadingSpace] Loc=<main.c:13:5>
6 equal '='          Loc=<main.c:13:7>
7 numeric_constant '1' [LeadingSpace] Loc=<main.c:13:9 <Spelling=main.c
   :5:16>>
8 semi ';'          Loc=<main.c:13:15>
9 identifier 'i'      [StartOfLine] [LeadingSpace] Loc=<main.c:14:5>
10 equal '='          Loc=<main.c:14:7>
11 numeric_constant '1' [LeadingSpace] Loc=<main.c:14:9>
12 semi ';'          Loc=<main.c:14:10>

```

这一部分对应源代码的：

```

1 a = INIT_A;
2 b = INIT_B;
3 i = 1;

```

这几行，对变量赋初值。等号/赋值号被表意地标记为 equal.

值得注意的是，此处出现了 numeric\_constant 这个新 token，和 Spelling 这个在 Loc 参数中出现的附属参数。显而易见的是 numeric\_constant 代表**数值常量**，作为赋值式子中的右值是顺理成章的；而特别地对于 a, b 两参量中的 Spelling 从何而来，经过分析理解，是由于源代码中对于 a, b 两个参数的赋值使用了**宏定义常量**，所以并非像 i 的赋值一样直接使用常数，而是需要宏展开，Spelling 标注的位置就是源代码中**宏定义声明**的位置。

### 词法分析片段 4

```

1 identifier 'scanf'   [StartOfLine] [LeadingSpace] Loc=<main.c:16:5>
2 l_paren '('         Loc=<main.c:16:10>
3 string_literal '%"d"' Loc=<main.c:16:11>

```

```

4 comma ',' Loc=<main.c:16:15>
5 amp '&' [LeadingSpace] Loc=<main.c:16:17>
6 identifier 'n' Loc=<main.c:16:18>
7 r_paren ')' Loc=<main.c:16:19>
8 semi ';' Loc=<main.c:16:20>

```

这一部分对应源代码的：

```

1 scanf("%d", &n);

```

这一行，接收用户输入整型数。出现了新的 token，名为 `string_literal` 和 `amp`，意义比较明确不再赘述。可以发现，函数和变量一样，只是宽泛地、默认地被标记为 `identifier`。

#### 词法分析片段 5

```

1 if 'if' [StartOfLine] [LeadingSpace] Loc=<main.c:17:5>
2 l_paren '(' [LeadingSpace] Loc=<main.c:17:8>
3 identifier 'n' Loc=<main.c:17:9>
4 greater '>' [LeadingSpace] Loc=<main.c:17:11>
5 numeric_constant '50' [LeadingSpace] Loc=<main.c:17:13 <Spelling=main.c
   :3:15>>
6 r_paren ')' Loc=<main.c:17:18>
7 l_brace '{' [LeadingSpace] Loc=<main.c:17:20>
8 identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<main.c:18:9>
9 l_paren '(' Loc=<main.c:18:15>
10 string_literal '"Input_exceeds_maximum_value%d\n"' Loc=<main.c
   :18:16>
11 comma ',' Loc=<main.c:18:50>
12 numeric_constant '50' [LeadingSpace] Loc=<main.c:18:52 <Spelling=main.c
   :3:15>>
13 r_paren ')' Loc=<main.c:18:57>
14 semi ';' Loc=<main.c:18:58>
15 return 'return' [StartOfLine] [LeadingSpace] Loc=<main.c:19:9>
16 numeric_constant '1' [LeadingSpace] Loc=<main.c:19:16>
17 semi ';' Loc=<main.c:19:17>
18 r_brace '}' [StartOfLine] [LeadingSpace] Loc=<main.c:20:5>

```

这一部分对应源代码的：

```

1 if (n > MAX_N) {
2     printf("Input_exceeds_maximum_value%d\n", MAX_N);
3     return 1;
4 }

```

这几行，出现了与判断分支相关的新 token。其中 `if` 关键词直接被标记为 `if`；判断语句中的大于号被标记为 `greater`。以及这里还出现了“满足条件直接结束程序”的 `return` 逻辑，能注意到 `return` 关键词也直接被标记为 `return`。此外，判断句中的常量再次使用到了宏定义，可以观察到 `Spelling` 照常出现。

## 词法分析片段 6

```

1 identifier 'printf'      [StartOfLine] [LeadingSpace]   Loc=<main.c:22:5 <
    Spelling=main.c:7:18>>
2 l_paren '('              Loc=<main.c:22:5 <Spelling=main.c:7:24>>
3 string_literal '%"d\n"'   Loc=<main.c:22:5 <Spelling=main.c:7:25>>
4 comma ','                Loc=<main.c:22:5 <Spelling=main.c:7:31>>
5 identifier 'a'           [LeadingSpace] Loc=<main.c:22:5 <Spelling=main.c:22:11>>
6 r_paren ')'              Loc=<main.c:22:5 <Spelling=main.c:7:34>>
7 semi ';'                 Loc=<main.c:22:13>
8 identifier 'printf'      [StartOfLine] [LeadingSpace]   Loc=<main.c:23:5 <
    Spelling=main.c:7:18>>
9 l_paren '('              Loc=<main.c:23:5 <Spelling=main.c:7:24>>
10 string_literal '%"d\n"'  Loc=<main.c:23:5 <Spelling=main.c:7:25>>
11 comma ','                Loc=<main.c:23:5 <Spelling=main.c:7:31>>
12 identifier 'b'           [LeadingSpace] Loc=<main.c:23:5 <Spelling=main.c:23:11>>
13 r_paren ')'              Loc=<main.c:23:5 <Spelling=main.c:7:34>>
14 semi ';'                 Loc=<main.c:23:13>

```

这一部分对应源代码的:

```

1 PRINT(a);
2 PRINT(b);

```

这两行，输出 a, b 的值。此处使用到了**函数的宏定义**，所以在经过两个函数体的时候都用 Spelling 标记了宏定义声明所在的源码位置。

## 词法分析片段 7

```

1 while 'while'           [StartOfLine] [LeadingSpace]   Loc=<main.c:26:5>
2 l_paren '('              [LeadingSpace] Loc=<main.c:26:11>
3 identifier 'i'           Loc=<main.c:26:12>
4 less '<'                 [LeadingSpace] Loc=<main.c:26:14>
5 identifier 'n'           [LeadingSpace] Loc=<main.c:26:16>
6 r_paren ')'              Loc=<main.c:26:17>
7 l_brace '{'              [StartOfLine] [LeadingSpace]   Loc=<main.c:27:5>
8 identifier 't'           [StartOfLine] [LeadingSpace]   Loc=<main.c:28:9>
9 equal '='                [LeadingSpace] Loc=<main.c:28:11>
10 identifier 'b'           [LeadingSpace] Loc=<main.c:28:13>
11 semi ';'                 Loc=<main.c:28:14>
12 identifier 'b'           [StartOfLine] [LeadingSpace]   Loc=<main.c:29:9>
13 equal '='                [LeadingSpace] Loc=<main.c:29:11>
14 identifier 'a'           [LeadingSpace] Loc=<main.c:29:13>
15 plus '+'                 [LeadingSpace] Loc=<main.c:29:15>
16 identifier 'b'           [LeadingSpace] Loc=<main.c:29:17>
17 semi ';'                 Loc=<main.c:29:18>
18 identifier 'printf'      [StartOfLine] [LeadingSpace]   Loc=<main.c:30:9 <
    Spelling=main.c:7:18>>
19 l_paren '('              Loc=<main.c:30:9 <Spelling=main.c:7:24>>
20 string_literal '%"d\n"'  Loc=<main.c:30:9 <Spelling=main.c:7:25>>

```

```

21 comma ','          Loc=<main.c:30:9 <Spelling=main.c:7:31>>
22 identifier 'b'     [LeadingSpace] Loc=<main.c:30:9 <Spelling=main.c:30:15>>
23 r_paren ')'        Loc=<main.c:30:9 <Spelling=main.c:7:34>>
24 semi ';'          Loc=<main.c:30:17>
25 identifier 'a'     [StartOfLine] [LeadingSpace] Loc=<main.c:31:9>
26 equal '='          [LeadingSpace] Loc=<main.c:31:11>
27 identifier 't'     [LeadingSpace] Loc=<main.c:31:13>
28 semi ';'          Loc=<main.c:31:14>
29 identifier 'i'     [StartOfLine] [LeadingSpace] Loc=<main.c:32:9>
30 equal '='          [LeadingSpace] Loc=<main.c:32:11>
31 identifier 'i'     [LeadingSpace] Loc=<main.c:32:13>
32 plus '+'          [LeadingSpace] Loc=<main.c:32:15>
33 numeric_constant '1' [LeadingSpace] Loc=<main.c:32:17>
34 semi ';'          Loc=<main.c:32:18>
35 r_brace '}'        [StartOfLine] [LeadingSpace] Loc=<main.c:33:5>

```

这一部分对应源代码的：

```

1  while (i < n)
2  {
3      t = b;
4      b = a + b;
5      PRINT(b);
6      a = t;
7      i = i + 1;
8  }

```

这几行，是进行斐波那契数列计算的循环体。while 循环关键词被直接标记为 while；小于号被标记为 less，加号被标记为 plus。其他内容在前面均有出现。

#### 词法分析片段 8

```

1 r_brace '}'        [StartOfLine] Loc=<main.c:34:1>
2 eof ''            Loc=<main.c:34:2>

```

这是程序的结尾，一个是主函数域的右括号，还有一个 eof 标识符用于表示程序的结束。

综合观察以上的词法分析结果，发现并没有出现源代码中的头文件声明、宏定义、注释，或被替换或被删除。这说明，词法分析之前，编译器已经完成了预处理所做的所有工作，这也间接说明了之前分析的-E 命令参数的实际作用。

当然，一切头文件引用也被包含在词法分析中并罗列在主程序的分析之前，但太过冗杂且基本逻辑是一致的，便不再分析。

至此，编译器的词法分析阶段功能验证完毕。

## 2. 语法分析

语法分析：主要工作是根据语言语法把 token 序列组织成语法树（Parse Tree）或抽象语法树（AST/Abstract Syntax Tree）。它在生成语法树的同时，还能检查并产生语法错误信息，以及做初步的结构检查。

进行语法分析依旧从源 C 代码出发，执行以下命令：

```
clang -E -Xclang -ast-dump main.c
```

命令基本构成和调用方法和词法分析基本相同。其中核心意义参数`-ast-dump` 替换词法分析中的`-dump-tokens`，它能够在语法分析阶段打印出完整的 AST 抽象语法树。

在终端，直接输出 AST 抽象语法树的效果如下图所示：

图 8：在 Ubuntu 终端直接输出 AST

下面对输出的 AST 树进行分析。整个 AST（包括库文件引用）结构是很繁杂的，为缩减报告篇幅，在下面的分析中适当省略部分分支内容。

首先跳过库文件拷贝部分，找到 `main` 函数入口：

```
1 -FunctionDecl 0x6423345a9718 <main.c:9:1, line:34:1> line:9:5 main 'int ()'
```

下面选取一些语句进行 AST 分析。

对于这一声明语句：

```
1 int a, b, i, t, n;
```

可以找到对应的 AST 片段：

#### AST 片段 1

```
1 -FunctionDecl 0x6135e41b0788 <main.c:9:1, line:34:1> line:9:5 main 'int ()'
2 -CompoundStmt 0x6135e41934c8 <col:12, line:34:1>
3   |-DeclStmt 0x6135e41b0ae0 <line:10:5, col:22>
4     |-VarDecl 0x6135e41b0848 <col:5, col:9> col:9 used a 'int'
5     |-VarDecl 0x6135e41b08c8 <col:5, col:12> col:12 used b 'int'
6     |-VarDecl 0x6135e41b0948 <col:5, col:15> col:15 used i 'int'
7     |-VarDecl 0x6135e41b09c8 <col:5, col:18> col:18 used t 'int'
8     |-VarDecl 0x6135e41b0a48 <col:5, col:21> col:21 used n 'int'
```



首先看各个节点的命名。第一行是 main 函数入口，标记为 FunctionDecl，即“函数声明”，也可以视为主程序的**根节点**。下面第一级子节点标记为 CompoundStmt，意为“**复合语句块**”，代表这一整个层级。再下一级节点标记为 DeclStmt，意为“**声明语句**”，说明这一复合语句的具体类型和作用。最低一级节点为若干 VarDecl，意为“变量声明”，每个节点对应原声明语句中的一个变量——a, b, i, t, n 共 5 个变量，对应 5 个节点。

再看节点名后的两个 (/段) 参数。第一个 0x... 是对应变量或函数在当前程序中的指针地址；第二段参数与代码在文本中的位置有关，具体编写/含义各不相同。比如，对于 main 函数，<main.c:9:1, line:34:1> 的含义是“函数从第 9 行第 1 列开始，到第 34 行第 1 列结束”，line:9:5 的含义是其在源代码中的具体位置，即第 9 行第 5 个字符开始。而对于 CompoundStmt 和 DeclStmt 节点，其参数构成相近，尖括号内的前一参数表示语句开始位置，后一参数表示语句结束位置。最后，对于 VarDecl 节点，有三个 col 参数，第一个是所属语句的开始列位置，第二个是此变量开始列位置，第三个是此变量结束列位置。由于上面例句中变量都只有一个字符长度，所以后两个参数自然是一致的。

最后还可以观察到变量声明有几个额外参数。used 表示该变量在整个程序中被使用过；后一个参数是变量名称；最后一个变量声明变量类型。

可见，AST 除了包含基础的程序语法结构信息，还需包含每个语句中的位置、范围、名称、状态等信息，细致到单个变量或函数，整体信息丰富而完善。

同理，还可以再对若干语句进行 AST 对应的分析。

再选择一个判断语句进行分析。以这个判断输入是否越界的判断语句块为例：

```
1 if (n > MAX_N) {
2     printf("Input exceeds maximum value %d\n", MAX_N);
3     return 1;
4 }
```

找到对应的 AST 片段：

AST 片段 2

```
1 -FunctionDecl 0x6135e41b0788 <main.c:9:1, line:34:1> line:9:5 main 'int ()'
2   -CompoundStmt 0x6135e41934c8 <col:12, line:34:1>
3     | - . . . . .
4     | -IfStmt 0x6135e41b1008 <line:17:5, line:20:5>
5     |   | -BinaryOperator 0x6135e41b0e50 <line:17:9, line:3:15> 'int' '>'
6     |   |   | -ImplicitCastExpr 0x6135e41b0e38 <line:17:9> 'int' <LValueToRValue>
7     |   |   |   | -DeclRefExpr 0x6135e41b0df8 <col:9> 'int' lvalue Var 0
8     |   |   |   |   x6135e41b0a48 'n' 'int'
9     |   |   |   | -IntegerLiteral 0x6135e41b0e18 <line:3:15> 'int' 50
10    |   | -CompoundStmt 0x6135e41b0fe8 <line:17:20, line:20:5>
11    |   |   | -CallExpr 0x6135e41b0f58 <line:18:9, col:57> 'int'
12    |   |   |   | -ImplicitCastExpr 0x6135e41b0f40 <col:9> 'int (*) (const char *,
13    |   |   |   |   ...) ' <FunctionToPointerDecay>
14    |   |   |   |   | -DeclRefExpr 0x6135e41b0e70 <col:9> 'int (const char *, ...) '
15    |   |   |   |   |   Function 0x6135e41713b8 'printf' 'int (const char *, ...) '
16    |   |   |   |   |   | -ImplicitCastExpr 0x6135e41b0fa0 <col:16> 'const char *' <NoOp>
17    |   |   |   |   |   |   | -ImplicitCastExpr 0x6135e41b0f88 <col:16> 'char *' <
18    |   |   |   |   |   |   |   ArrayToPointerDecay>
```



```

15 | | -StringLiteral 0x6135e41b0ed0 <col:16> 'char[32]' lvalue "
    | Input exceeds maximum value %d\n"
16 | | -IntegerLiteral 0x6135e41b0f08 <line:3:15> 'int' 50
17 | -ReturnStmt 0x6135e41b0fd8 <line:19:9, col:16>
18 | -IntegerLiteral 0x6135e41b0fb8 <col:16> 'int' 1

```

可以发现，代表判断语句块的 IfStmt 节点和上面分析的 DeclStmt 声明语句在 AST 是同级别的，因为他们都是在 main 函数中最外一层的语句。

第一步，详细分析一下这个条件表达式：

```

1 | -IfStmt . . . . .
2 | | -BinaryOperator 0x6135e41b0e50 <line:17:9, line:3:15> 'int' '>'
3 | | | -ImplicitCastExpr 0x6135e41b0e38 <line:17:9> 'int' <LValueToRValue>
4 | | | | -DeclRefExpr 0x6135e41b0df8 <col:9> 'int' lvalue Var 0
    | | | | x6135e41b0a48 'n' 'int'
5 | | | -IntegerLiteral 0x6135e41b0e18 <line:3:15> 'int' 50

```

顶层 BinaryOperator 节点是识别二元操作符“>”大于号的结果，同时也划定了这个语句的类型。下一级节点有两个，左操作数是 ImplicitCastExpr 即**隐式转换表达式**，后标注了 <LValueToRValue>，含义是将左值转化为右值，具体转化了什么参量稍后说明；右操作数是 IntegerLiteral 即**字面常量**，这里是整数 50。其中左操作数还拥有再下一级的节点 DeclRefExpr，引用了局部变量 n，上面提到的左值转化为右值就是将这个变量的值转化为右值，才能够进行大小比较。

紧接着 BinaryOperator 节点之后顺接了同级的 CompoundStmt 节点，代表判断语句块的主要部分（大括号包括的部分）。

CompoundStmt 节点下的主要部分是对 printf() 函数的调用：

```

1 | | - . . . . .
2 | -CompoundStmt 0x6135e41b0fe8 <line:17:20, line:20:5>
3 | | -CallExpr 0x6135e41b0f58 <line:18:9, col:57> 'int'
4 | | | -ImplicitCastExpr 0x6135e41b0f40 <col:9> 'int (*) (const char *,
    | | | | ...) ' <FunctionToPointerDecay>
5 | | | | -DeclRefExpr 0x6135e41b0e70 <col:9> 'int (const char *, ...) '
    | | | | Function 0x6135e41713b8 'printf' 'int (const char *, ...) '
6 | | | | -ImplicitCastExpr 0x6135e41b0fa0 <col:16> 'const char *' <NoOp>
7 | | | | -ImplicitCastExpr 0x6135e41b0f88 <col:16> 'char *' <
    | | | | ArrayToPointerDecay>
8 | | | | -StringLiteral 0x6135e41b0ed0 <col:16> 'char[32]' lvalue "
    | | | | Input exceeds maximum value %d\n"
9 | | | | -IntegerLiteral 0x6135e41b0f08 <line:3:15> 'int' 50

```

顶层 CallExpr 节点表示**函数调用**，且注明了函数返回类型为 int。下一级节点的第一个比较特殊，是 <FunctionToPointerDecay>，即“函数名衰减为指针”，的隐式转换，这是 C 语言进行函数调用时为了让函数调用合法，自动插入的**函数名 → 函数指针转换**既定操作（转换后形如 int (\*) (const char \*, ...) 指向函数的指针）。后两个同级节点易于理解，是函数中的两个传参，分别对应字符串和常量。

其中字符串参数下属节点较多，着重分析一下。第一层，ImplicitCastExpr <NoOp>，只作**类型微调**，具体是加 const，NoOp 表示“不改变值，只做类型调整”。第二层，ImplicitCastExpr

<ArrayToPointerDecay>,将**数组衰减为指针**,这同样来源于 C 语言的语法规则。第三层,StringLiteral 来到字符串的实际内容。

更多 AST 分析基本原理相似,不再赘述。编译器的**语法分析阶段功能验证完毕**。

### 3. 语义分析

语义分析:主要工作是在 AST 上做语义检查并收集必要信息(类型、作用域、符号表)。换句话说,这一过程需要理解程序的含义并检查正确性,而不仅仅是结构。

具体而言,语义分析所做的任务有:

- **类型检查**:确保操作数和操作符匹配,以及检查函数调用参数类型是否与函数定义匹配。这一步也涉及**隐式类型转换**,前面的**语法分析**阶段中有所提及。
- **检查标识符**:确保每个变量和函数在使用前已经声明并检查是否重复定义。确保每个标识符(变量、函数、类型等)在正确作用域内调用。
- **函数调用检查**:检查参数个数、参数类型是否匹配,检查返回类型是否正确。
- **常量表达式求值**:对常量表达式做计算,生成**常量折叠**。
- **检查控制流语义**:检查 break, continue 等是否在循环内部,return 语句类型是否匹配函数返回类型。
- **构造和维护符号表**:记录每个变量、函数、类型的属性信息,为作用域嵌套和查找提供支持。生成的符号表会用于后续步骤——**中间代码生成和优化**。

最重要的是类型检查、标识符检查和符号表的管理。

语义分析没有提供具体的参考命令进行研究,以上是对于语义分析阶段的全部论述。

### 4. 中间代码生成

中间代码生成:主要工作是把语义正确的 AST 翻译成一个与机器无关、便于优化的中间表示(IR)。

中间代码的生成依旧从源 C 代码出发,首先执行以下命令来直接生成中间代码文件:

```
clang -S -emit-llvm main.c
```

命令仍然使用 Clang 前端,其中-S 参数表示只生成汇编/IR,不做汇编成目标文件;-emit-llvm 参数告诉 Clang 不要生成本地汇编,而是生成 LLVM IR. 两者一并使用时,输出文件是 .ll 格式文件(LLVM IR 文本形式);若不添加-S 参数,输出的是 .bc 格式文件(LLVM bitcode 二进制文件)。

执行命令后得到 main.ll. 对于该文件,分段进行分析。

main.ll 段落 1

```
1 ; ModuleID = 'main.c'
2 source_filename = "main.c"
```

```

3 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"
4 target triple = "x86_64-pc-linux-gnu"

```

这一段为**模块基本信息和目标平台设置**。其中 ModuleID 为模块标识符，此时为 main.c。源文件名同名。target datalayout 和 target triple 描述数据在目标平台的布局和目标平台三元组（CPU 架构-供应商-操作系统）。

#### main.ll 段落 2

```

1 @.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
2 @.str.1 = private unnamed_addr constant [32 x i8] c"Input exceeds maximum
  value\0A\00", align 1
3 @.str.2 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

```

这一段定义 C 语言**字符串常量**，供 scanf/printf 使用。其中 [N x i8] 代表长度为 N 的字符数组；\00 为 C 字符串结尾；private unnamed\_addr 表示内部链接，LLVM 可优化地址访问。

#### main.ll 段落 3

```

1 define dso_local i32 @main() #0 { ... }

```

主函数入口。有标明返回类型和函数属性。

#### main.ll 段落 4

```

1 %1 = alloca i32, align 4
2 %2 = alloca i32, align 4
3 ...
4 %6 = alloca i32, align 4

```

这一段进行**栈空间的分配**，alloca 是分配栈上局部变量的指令。

#### main.ll 段落 5

```

1 store i32 0, ptr %1, align 4
2 store i32 0, ptr %2, align 4
3 store i32 1, ptr %3, align 4
4 store i32 1, ptr %4, align 4

```

这一段给变量**赋初值**，可对应到 C 语言代码：

```

1 a = 0; b = 0; i = 1; t = 1;

```

#### main.ll 段落 6

```

1 %7 = call i32 @__isoc99_scanf(ptr noundef @.str, ptr noundef %6)
2 %8 = load i32, ptr %6, align 4
3 %9 = icmp sgt i32 %8, 50
4 br i1 %9, label %10, label %12

```

这一段进行输入数据越界检查（大小比较**判断语句**）。前两行调用 scanf() 读取用户输入的 n 参数大小，第三行 icmp sgt 比较 n > 50 ?，最后一行进行条件跳转——判断语句为真，则

跳转到%10 输出提示并终止，否则跳到 %12 执行后续逻辑。具体%10 和%12 内容如何，下面将分析。根据阅读推测，他们应该代表预处理过后主程序代码的行号。

## main.ll 段落 7

```

1 10:                                     ; preds = %0
2   %11 = call i32 @printf(ptr noundef @.str.1, i32 noundef 50)
3   store i32 1, ptr %1, align 4
4   br label %31

```

这是%10 的实际所指，**处理越界**情况。第一行输出越界提示，而后设置返回值为 1，最后跳转（主函数返回）到%31 结束程序。

## main.ll 段落 8

```

1 12:                                     ; preds = %0
2   %13 = load i32, ptr %2, align 4
3   %14 = call i32 @printf(ptr noundef @.str.2, i32 noundef %13)
4   . . . . .
5   br label %17
6 17:                                     ; preds = %21, %12
7   %18 = load i32, ptr %4, align 4
8   . . . . .
9   br i1 %20, label %21, label %31
10 21:                                    ; preds = %17
11  %22 = load i32, ptr %3, align 4
12  store i32 %22, ptr %5, align 4
13  . . . . .
14  store i32 %30, ptr %4, align 4
15  br label %17, !llvm.loop !6

```

12: 是前面%12 的实际所指。在这之后实际上就是**斐波那契数列计算**的核心逻辑，llvm 代码较为冗长，所以不再逐行解析。

## main.ll 段落 9

```

1 31:                                     ; preds = %10, %17
2   %32 = load i32, ptr %1, align 4
3   ret i32 %32

```

之前提到过的%31 主程序返回逻辑所指。

## main.ll 段落 10

```

1 declare i32 @__isoc99_scanf(ptr noundef, ...) #1
2 declare i32 @printf(ptr noundef, ...) #1

```

这里是**外部函数声明**，声明了 scanf 和 printf，包括参数和返回类型等信息。

## main.ll 段落 11

```

1 attributes #0 = { noinline nounwind optnone uwtable . . . . . }
2 attributes #1 = { . . . . . }

```

这里规定了**函数属性**。其中 #0 规定主函数属性，#1 规定外部函数属性。

## main.ll 段落 12

```

1 !llvm.module.flags = !{!0, !1, !2, !3, !4}
2 !llvm.ident = !{!5}
3
4 !0 = !{i32 1, !"wchar_size", i32 4}
5 !1 = !{i32 8, !"PIC_Level", i32 2}
6 . . . . .
7 !7 = !{"llvm.loop.mustprogress"}

```

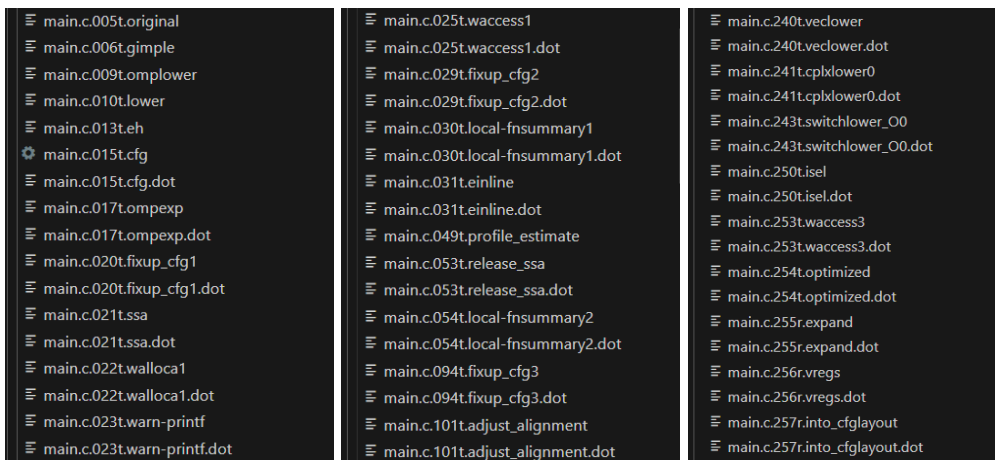
最后是 LLVM Metadata，包含存储编译器、循环属性等信息，用于优化和调试，不影响程序逻辑。

除了直接生成 LLVM IR 文本文件，还可以添加命令参数（gcc flag）获得中间代码生成的多阶段的输出。执行下面的命令：

```
gcc -O0 -fdump-tree-all-graph -fdump-rtl-all-graph main.c -o main
```

和前面不一样的是，这里使用了 GNU 编译器。-O0 限定了优化保持关闭状态，方便完整地分析；-fdump-tree-all-graph 能够输出 GCC 所有阶段的中间树表示（包括：GIMPLE/简化的三地址码表示，CFG/控制流图，SSA/静态单赋值形式），输出 .dot 格式的 Graphviz 文件；-fdump-rtl-all-graph 能够输出所有阶段的 RTL（Register Transfer Level，中间寄存器表示），同样生成 Graphviz 可视化文件。

执行命令之后，可以看到目录中生成了有约百条记录：



```

≡ main.c.005t.original
≡ main.c.006t.gimple
≡ main.c.009t.omplower
≡ main.c.010t.lower
≡ main.c.013t.leh
* main.c.015t.cfg
≡ main.c.015t.cfg.dot
≡ main.c.017t.ompexp
≡ main.c.017t.ompexp.dot
≡ main.c.020t.fixup_cfg1
≡ main.c.020t.fixup_cfg1.dot
≡ main.c.021t.ssa
≡ main.c.021t.ssa.dot
≡ main.c.022t.wallocal1
≡ main.c.022t.wallocal1.dot
≡ main.c.023t.warn-printf
≡ main.c.023t.warn-printf.dot
≡ main.c.025t.waccess1
≡ main.c.025t.waccess1.dot
≡ main.c.029t.fixup_cfg2
≡ main.c.029t.fixup_cfg2.dot
≡ main.c.030t.local-fnsummary1
≡ main.c.030t.local-fnsummary1.dot
≡ main.c.031t.inline
≡ main.c.031t.inline.dot
≡ main.c.049t.profile_estimate
≡ main.c.053t.release_ssa
≡ main.c.053t.release_ssa.dot
≡ main.c.054t.local-fnsummary2
≡ main.c.054t.local-fnsummary2.dot
≡ main.c.094t.fixup_cfg3
≡ main.c.094t.fixup_cfg3.dot
≡ main.c.101t.adjust_alignment
≡ main.c.101t.adjust_alignment.dot
≡ main.c.240t.veclower
≡ main.c.240t.veclower.dot
≡ main.c.241t.cplxlower0
≡ main.c.241t.cplxlower0.dot
≡ main.c.243t.switchlower_O0
≡ main.c.243t.switchlower_O0.dot
≡ main.c.250t.isel
≡ main.c.250t.isel.dot
≡ main.c.253t.waccess3
≡ main.c.253t.waccess3.dot
≡ main.c.254t.optimized
≡ main.c.254t.optimized.dot
≡ main.c.255r.expand
≡ main.c.255r.expand.dot
≡ main.c.256r.vregs
≡ main.c.256r.vregs.dot
≡ main.c.257r.into_cfglayout
≡ main.c.257r.into_cfglayout.dot

```

图 9：部分记录

简而言之，这些记录是 GCC 中间优化/分析阶段的 dump 文件。后缀为 't' 的命名段中的数字代表的就是阶段编号。

找出其中后缀为 cfg.dot 的文件：

```

main.c.015t.cfg.dot
main.c.020t.fixup_cfg1.dot
main.c.029t.fixup_cfg2.dot
main.c.094t.fixup_cfg3.dot

```

分别执行下面的命令：

```
dot -Tpng main.c.015t.cfg.dot -o cfg01.png
```

```
dot -Tpng main.c.020t.fixup_cfg1.dot -o cfg02.png
```

```
dot -Tpng main.c.029t.fixup_cfg2.dot -o cfg03.png
```

```
dot -Tpng main.c.094t.fixup_cfg3.dot -o cfg04.png
```

就可以生成出一系列的**控制流图** (CFG)。这些流程图的变化，一定程度上反映了程序结构随着优化/分析而向着精简与高效进步的过程。

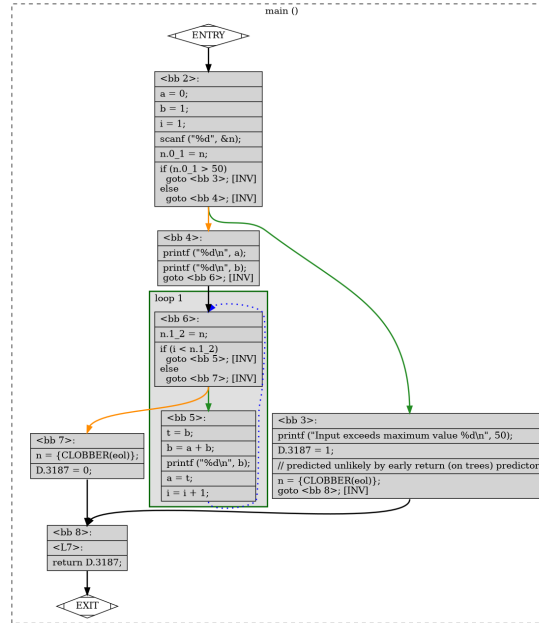


图 10: cfg01.png/cfg02.png (015t/020t)

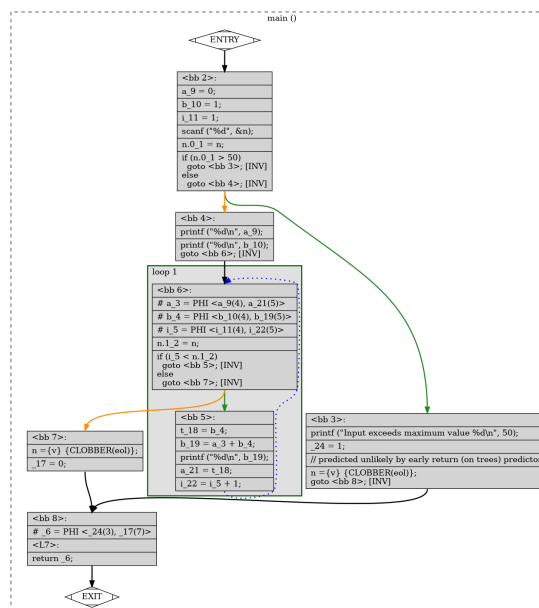


图 11: cfg03.png/cfg04.png (029t/094t)

结果表明，015t 和 020t、029t 和 094t 这两对阶段之间的 CFG 是完全一致的，所以变化发生在 020t 至 029t 阶段之间。观察到主要变化有两点：

- 有一些需要初始化数值的变量被添加进来；
- 几乎所有的参数名被修改，特点是都被标记上了各不相同的编号（有些非变量的标识符也被标记了）。

根据上面的发现，推测这一部分是进行了分析上的进阶。原先，仅仅分析全局的逻辑，不考虑具体参量的赋值，特别是初始赋值。分析进阶后，将所有变量的赋值加入考虑，同时为了区分各个参量，类似于给所有变量/标识符赋予独特的 ID，用以在后续的分析过程中更清晰地区分各个变量。

通过上面的分析，切实地观察到了中间代码将 AST 翻译成中间表示的动态过程。编译器的中间代码生成阶段功能验证完毕。

## 5. 代码优化

代码优化：主要工作是在不改变程序语义前提下，提升代码运行效率、缩减代码体积，让生成的代码更快、更小或更省能。

在 LLVM 框架下，优化和分析都是通过 Pass 进行的。官方将 Pass 分为以下三类：

1. **Analysis Pass**：不直接修改代码，只是收集信息，供其他 Pass 使用。一般从中可以分析出代码整体的逻辑架构，各种分支结构等等。
2. **Transform Pass**：对代码进行改造与优化。具体的有如，**常量传播**（比如  $x = 1 + 2$  直接传播为  $x = 3$ ），**死代码删除**（永远不会执行或无副作用的代码），循环优化（这又包含**循环展开**、**循环不变代码外提**等），**内联优化**，公共子表达式消除等。
3. **Utility Passes**：辅助操作，不属于纯粹的分析或优化，但在 IR 处理时提供支持功能。例子有，IR 打印、统计信息输出，Pass 管理相关的功能等。

如果想要观察优化过程中代码本身的变化，可以执行以下命令：

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

这句命令调用了 LLVM 的 llc 工具。-print-before-all 和 -print-after-all 参数分别用于指示打印出**每个 Pass 之前和之后的状态**用来观察与分析。> main.log 意为将结果输出为该文件。2>&1 表示把**标准错误**也重定向到**标准输出**，则所有输出（包括错误信息、调试信息）都会显示在输出文件 main.log 中。

执行此命令之后，将会产生文件 main.log，里面包含了每个 LLVM Pass 之前/之后状态的所有信息。main.log 文件长达数万行，不便于一一分析，此处选取一个明显的变化细节作简单阐述。

在原日志文件中第 875 行之前，每一轮都存在这样一个语句：

```
1 br i1 %20, label %21, label %31
```

它的含义是，循环条件为假时，直接跳到基本块%31。

而从此算起的下一轮中，在第 931 行，这一语句被修改为：

```
1 br i1 %20, label %21, label %.loopexit
```



这表明, LLVM 引入了一个新的块 `.loopexit`. 这一变化是 `loop-simplify pass`/循环优化 Pass 的一个重要步骤, 它确保每个循环都有**唯一的退出块**, 从而便于后续优化 (如循环展开、强度削减)。

除了通过上面命令输出 LLVM Passes 的具体日志, 还可以直接使用不同的编译优化选项来探索优化过程对于代码效率的影响。

注意, 此过程需要使用在之前提及到的 `.bc` 格式文件 (LLVM bitcode 二进制文件)。从 LLVM IR 的 `.ll` 文件生成 `.bc` 文件的命令如下:

```
llvm-as main.ll -o main.bc
```

有了 `.bc` 文件之后就可以根据需要, 改变编译优化选项参数来生成新的 `.ll` 文件, 例如:

```
opt -S -O1 main.bc -o main-O1.ll
```

就生成了 `O1` 编译优化的 LLVM IR. 后续再从 `.ll` 文件出发进行汇编、链接等步骤就可以生成最终优化后的程序。

为了对比不同编译优化选项下程序的性能, 需要进行计时作为对比参量。将原来的程序进行如下的修改, 使其能够完整记录运行时:

修改后的 `main.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4
5 #define MAX_N 100
6 #define INIT_A 0
7 #define INIT_B 1
8
9 #define PRINT(x) printf("%d\n", x)
10
11 int main() {
12     int a, b, i, t, n;
13     struct timeval start, end;
14     double elapsed;
15
16     a = INIT_A;
17     b = INIT_B;
18     i = 1;
19
20     // time start
21     gettimeofday(&start, NULL);
22
23     PRINT(a);
24     PRINT(b);
25
26     // this is a friendly annotation *
27     while (i < n)
28     {
```



```

29     t = b;
30     b = a + b;
31     PRINT(b);
32     a = t;
33     i = i + 1;
34 }
35
36 // time end
37 gettimeofday(&end, NULL);
38
39 // calculate
40 elapsed = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) /
         1000000.0;
41
42 printf("Execution time: %.6f seconds\n", elapsed);
43
44 }

```

因为后续测试数据可能很大（不考虑实际运行的数理正确性，只考虑运行效能），删除了输入数据的越界判断处理。

修改后，生成各个优化选项下的最终可执行文件完整命令行过程如下图所示：

```

bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ clang -S -emit-llvm main.c
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ llvm-as main.ll -o main.bc
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ opt -S -O1 main.bc -o main-O1.ll
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ opt -S -O2 main.bc -o main-O2.ll
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ opt -S -O3 main.bc -o main-O3.ll
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ llc main.bc -filetype=obj -o mainLLVM.o
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ llvm-as main-O1.ll -o main-O1.bc
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ llvm-as main-O2.ll -o main-O2.bc
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ llc main-O1.bc -filetype=obj -o mainLLVM-O1.o
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ llc main-O2.bc -filetype=obj -o mainLLVM-O2.o
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ llc main-O3.bc -filetype=obj -o mainLLVM-O3.o
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ clang mainLLVM.o -o mainLLVM
/usr/bin/ld: mainLLVM.o: warning: relocation in read-only section `.text'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ clang mainLLVM-O1.o -o mainLLVM-O1
/usr/bin/ld: mainLLVM-O1.o: warning: relocation in read-only section `.text'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ clang mainLLVM-O2.o -o mainLLVM-O2
/usr/bin/ld: mainLLVM-O2.o: warning: relocation in read-only section `.text'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ clang mainLLVM-O3.o -o mainLLVM-O3
/usr/bin/ld: mainLLVM-O3.o: warning: relocation in read-only section `.text'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE

```

图 12: 优化程序生成过程

最后，在此目录下依次运行各可执行文件，得到如下的运行时结果：

```

-1262512887
-573849639
-1836362526
1884755131
Execution time: 6.465224 seconds
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$

```

图 13: 无编译优化

```
-1262512887
-573849639
-1836362526
1884755131
Execution time: 5.618795 seconds
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$
```

图 14: -O1 编译优化

```
-1262512887
-573849639
-1836362526
1884755131
Execution time: 5.948980 seconds
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$
```

图 15: -O2 编译优化

```
-1262512887
-573849639
-1836362526
1884755131
Execution time: 5.189250 seconds
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$
```

图 16: -O3 编译优化

实际上，可能由于本机性能问题等，运行时并不稳定，上面的数据只是在相对稳定时截取。但从数据中可以明确观察到的是，采用编译优化选项的程序确实相较于无编译优化的程序有运行速度上的提升。

至此，编译器的代码优化阶段功能验证完毕。

## 6. 代码生成

代码生成：以中间语言或预处理后文件为输入，生成目标语言（汇编语言）。

对于 x86、ARM、LLVM，分别执行下面的命令生成对应的汇编代码：

```
gcc main.i -S -o mainx86.S
aarch64-linux-gnu-gcc main.i -S -o mainARM.S
llc main.ll -o mainLLVM.S
```

生成结果都是.s 文件。

## (四) 汇编器

### 1. 汇编器的具体功能

汇编器接受汇编语言源程序（上面生成的.s 文件），将其转化为目标机器代码的目标文件（.o 文件），为二进制格式。

具体工作包括：

1. **词法/语法分析：**与编译器类似地，汇编器也需要读源代码，把汇编指令、伪指令、标号、注释等进行词法/语法解析，确认代码符合目标架构的汇编语法。
2. **符号收集：**进行第一遍扫描，收集汇编代码中的**符号**（label）和**外部引用**（extern），进行相应的标记或记录。
3. **指令和伪指令翻译：**进行第二遍扫描，把助记符（如 mov, add, jmp）翻译成对应的机器指令编码，并处理伪指令。
4. **地址与重定位信息生成：**汇编器不能完全确定所有地址（例如调用外部函数），所以会生成**重定位信息**，供链接器处理。
5. **生成目标文件（.o）。**

对于 x86、ARM、LLVM 的汇编代码，分别执行下面的命令，将其转化为目标文件：

```
gcc mainx86.S -c -o mainx86.o
```

```
aarch64-linux-gnu-gcc mainARM.S -c -o mainARM.o
```

```
llc main.bc -filetype=obj -o mainLLVM.o
```

生成各自的.o 目标文件。

## 2. 反汇编与汇编程序分析

通过反汇编，可以得到与目标文件对应的原始汇编代码。此处仅对 x86 进行分析，执行下面的代码：

```
objdump -d mainx86.o
```

执行后，汇编代码会直接在终端输出。输出结果如下，已经逐行进行分析并作注释：

mainx86.o 反汇编结果与分析

```

1 0000000000000000 <main>:
2   0:   f3 0f 1e fa                endbr64
3                                   ; CET 指令：用于控制流完整性，与逻辑无关
4   4:   55                          push    %rbp
5                                   ; 保存旧的基指针
6   5:   48 89 e5                    mov     %rsp,%rbp
7                                   ; rbp = rsp, 建立新的栈帧基址
8   8:   48 83 ec 20                 sub     $0x20,%rsp
9                                   ; 栈分配 32 字节局部空间（局部变量和对齐）
10  c:   64 48 8b 04 25 28 00        mov     %fs:0x28,%rax
11 13:   00 00
12                                   ; 读取线程局部存储的值，用于栈保护
13 15:   48 89 45 f8                mov     %rax,-0x8(%rbp)
14                                   ; 将值存到栈帧的 -0x8(%rbp)（在函数返回时检验）
15 19:   31 c0                      xor     %eax,%eax
16                                   ; eax 置零，用于设置返回值或清零寄存器
17 1b:   c7 45 e8 00 00 00 00        movl    $0x0,-0x18(%rbp)
18                                   ; a = 0

```

```

19 22: c7 45 ec 01 00 00 00 movl $0x1,-0x14(%rbp)
20 ; b = 1
21 29: c7 45 f0 01 00 00 00 movl $0x1,-0x10(%rbp)
22 ; i = 1
23 30: 48 8d 45 e4 lea -0x1c(%rbp),%rax
24 ; 为 scanf 准备 n 的地址
25 34: 48 89 c6 mov %rax,%rsi
26 ; rsi = &n, scanf 的第二个参数
27 37: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # 3e <main+0x3e>
28 3e: 48 89 c7 mov %rax,%rdi
29 ; "%d" 的地址, 通过重定位填充
30 41: b8 00 00 00 00 mov $0x0,%eax
31 ; eax 置零, varargs 调用约定
32 46: e8 00 00 00 00 call 4b <main+0x4b>
33 ; 调用函数 scanf("%d", &n)
34 4b: 8b 45 e4 mov -0x1c(%rbp),%eax
35 ; 取出 scanf 写入的 n 赋给 eax
36 4e: 83 f8 32 cmp $0x32,%eax
37 ; 判断 if (n > 50)
38 51: 7e 20 jle 73 <main+0x73>
39 ; 若 eax <= 50 则跳到 address 0x73 (处理正常路
    径), 否则继续 n > 50 的分支
40 53: be 32 00 00 00 mov $0x32,%esi
41 ; 将常数 50 放到 esi, 作为 printf 的参数
42 58: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # 5f <main+0x5f>
43 5f: 48 89 c7 mov %rax,%rdi
44 ; 准备 scanf 字符串参数地址
45 62: b8 00 00 00 00 mov $0x0,%eax
46 ; eax 置零, 可变参数调用约定
47 67: e8 00 00 00 00 call 6c <main+0x6c>
48 ; 调用 printf, 打印越界字符串信息
49 6c: b8 01 00 00 00 mov $0x1,%eax
50 ; 返回值 eax = 1 (准备作为 main 函数返回值)
51 71: eb 70 jmp e3 <main+0xe3>
52 ; 跳转到函数结尾处做清理和返回
53 73: 8b 45 e8 mov -0x18(%rbp),%eax
54 ; 处于正常路径: eax = a, 准备打印 a
55 76: 89 c6 mov %eax,%esi
56 ; a 赋给 esi (printf 第二个参数)
57 78: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # 7f <main+0x7f>
58 7f: 48 89 c7 mov %rax,%rdi
59 ; 准备字符串参数地址
60 82: b8 00 00 00 00 mov $0x0,%eax
61 87: e8 00 00 00 00 call 8c <main+0x8c>
62 ; printf("%d\n", a);
63 8c: 8b 45 ec mov -0x14(%rbp),%eax
64 ; eax = b, 准备打印 b, 下同理
65 8f: 89 c6 mov %eax,%esi

```

```

66  91:  48 8d 05 00 00 00 00    lea    0x0(%rip),%rax          # 98 <main+0x98>
67  98:  48 89 c7                mov    %rax,%rdi
68  9b:  b8 00 00 00 00          mov    $0x0,%eax
69  a0:  e8 00 00 00 00          call   a5 <main+0xa5>
70                                ; printf("%d\n", b);
71  a5:  eb 2f                  jmp    d6 <main+0xd6>
72                                ; 打印完 a,b 之后跳到循环判断处、
73  a7:  8b 45 ec                mov    -0x14(%rbp),%eax        ; 循环体起始行
74                                ; eax = b
75  aa:  89 45 f4                mov    %eax,-0xc(%rbp)
76                                ; t = b
77  ad:  8b 45 e8                mov    -0x18(%rbp),%eax
78                                ; eax = a
79  b0:  01 45 ec                add    %eax,-0x14(%rbp)
80                                ; b = a + b
81  b3:  8b 45 ec                mov    -0x14(%rbp),%eax
82                                ; eax = b, 再次准备打印 .....
83  b6:  89 c6                  mov    %eax,%esi
84  b8:  48 8d 05 00 00 00 00    lea    0x0(%rip),%rax          # bf <main+0xbf>
85  bf:  48 89 c7                mov    %rax,%rdi
86  c2:  b8 00 00 00 00          mov    $0x0,%eax
87  c7:  e8 00 00 00 00          call   cc <main+0xcc>
88                                ; printf("%d\n", b) , 输出此轮 b
89  cc:  8b 45 f4                mov    -0xc(%rbp),%eax
90                                ; eax = t
91  cf:  89 45 e8                mov    %eax,-0x18(%rbp)
92                                ; a = t
93  d2:  83 45 f0 01            addl    $0x1,-0x10(%rbp)
94                                ; 循环计数参量 i 的累加
95  d6:  8b 45 e4                mov    -0x1c(%rbp),%eax
96                                ; eax = n
97  d9:  39 45 f0                cmp    %eax,-0x10(%rbp)
98                                ; 等同于 cmp n, i, 比较 n 与 i 大小
99  dc:  7c c9                  jl     a7 <main+0xa7>
100                                ; 如果 n < i , 跳回循环体 a7 继续循环计算
101  de:  b8 00 00 00 00          mov    $0x0,%eax
102                                ; 否则循环结束, 设 main 返回值 eax = 0
103  e3:  48 8b 55 f8            mov    -0x8(%rbp),%rdx
104                                ; 将起始时栈上保存的值读入 rdx
105  e7:  64 48 2b 14 25 28 00    sub    %fs:0x28,%rdx
106  ee:  00 00
107                                ; rdx = rdx - [fs:0x28]
108                                ; 比较保存值与当前 TLS canary
109  f0:  74 05                  je     f7 <main+0xf7>
110                                ; 如果相等 (即没有栈破坏), 跳到 f7 正常返回
111  f2:  e8 00 00 00 00          call   f7 <main+0xf7>
112                                ; 如果不等, 调用特定函数异常处理
113  f7:  c9                      leave

```

```

114             ; 恢复栈帧: mov rsp, rbp; pop rbp ...
115 f8:      c3                ret
116             ; 返回

```

通过分析汇编代码，可以看到程序在栈、内存层面上的一些细节操作。总体分析下来，代码逻辑和原始的 C 源码是一致的。

**汇编器功能验证完毕。**

## （五） 链接器

链接器把编译器/汇编器产生的若干**目标文件**（.o）、**静态库**（.a）与**共享库**（.so）合并，做各种符号处理、地址修正、布局修缮等工作，输出最终的**目标文件**或可执行镜像，并为运行时装载留下必要的元数据。

具体的链接流程包括：

1. **输入收集**：读取命令行给定的 object 文件、静态库、共享库，对静态库按需展开。
2. **符号/节汇总**：为每个输入文件构建**符号表**和**节**（有如 .text, .data, .bss, .rodata, 重定位节等），并合并相同类型的节。
3. **地址分配**：分配每个输出节的**虚拟地址**（VMA）和**文件偏移**（LMA），设置段对齐、分页规则、加载段划分等。
4. **符号解析**：对每个**未定义**符号找到“最合适”的替换定义，此外处理强/弱符号优先级、重复定义、符号可见性等问题。
5. **重定位的生成/处理**：把对地址的占位修正为最终地址或产生需要运行时修正的信息，分为**静态重定位**和**动态重定位**两种。
6. **写入输出文件**：编写目标文件/**可执行文件**，包含节、段、符号表、重定位条目、动态节等字段。

对于 x86, ARM 和 LLVM 在汇编器中转化得到的目标文件，分别用下面的指令进行链接：

```
gcc mainx86.o -o mainx86
```

```
aarch64-linux-gnu-gcc -o mainARM mainARM.o -static -lc
```

```
clang mainLLVM.o -o mainLLVM
```

执行后得到没有后缀的**可执行文件**。

.o 目标文件和最终生成的可执行文件都是不可直接阅读的二进制文件，且这一部分不是最主要的学习内容，故不作进一步研究。

## （六） 运行程序

经过以上各个步骤，已经得到了 x86, ARM 和 LLVM 的最终可执行文件。由于处在 Ubuntu 的 Linux 虚拟环境下，直接在当前目录下打开就可以验证程序功能（ARM 需要 qemu 运行）：

```
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ ./mainx86
10
0
1
1
2
3
5
8
13
21
34
55
```

图 17: x86 的可执行文件运行结果

```
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ qemu-aarch64 ./mainARM
10
0
1
1
2
3
5
8
13
21
34
55
```

图 18: ARM 的可执行文件运行结果

```
bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01$ ./mainLLVM
0
1
1
2
3
5
8
13
21
34
55
```

图 19: LLVM 的可执行文件运行结果

对于三个可执行程序，均用测试数据 10 进行测试，都得到正确的输出数字串。说明程序本身逻辑正确，且以上经过的各个处理阶段均没有产生错误。

**至此，实验一全部完成。**

## 三、 实验二：LLVM IR 编程

### (一) SysY 程序编写

首先在第二问中我们小组等价的 SysY 程序如下：

```
1 #include <stdio.h>
2
3 int max(int a, int b);
4 int factorial(int n);
5
6 int main() {
7     int a = 10;
8     int b = 5;
9     int result = 0;
10    int i = 0;
11    int n = 5;
12
13    result = a + b;
14    result = a - b;
15    result = a * b;
16    printf("%d\n", result);
17    result = a / b;
18    result = a % b;
19
20    result = a;
21    result = result + 1;
22    result = result * 2;
23    printf("%d\n", result);
24
25    if (a > b) {
26        result = 1;
27    } else {
28        result = 0;
29    }
30
31    if (a > 0 && b > 0) {
32        result = 1;
33    }
34
35    if (a > 0 || b > 0) {
36        result = 1;
37    }
38
39    if (a > b) {
40        result = max(a, b);
41    } else if (a < b) {
42        result = b;
43    } else {
44        result = 0;
45    }
46
47    printf("%d\n", result);
```



```
48     i = 0;
49     result = 0;
50     while (i <= 10) {
51         result = result + i;
52         i = i + 1;
53     }
54
55     result = 0;
56     for (i = 0; i < 10; i = i + 1) {
57         if (i % 2 == 0) {
58             continue;
59         }
60         result = result + i;
61
62         if (result > 20) {
63             break;
64         }
65     }
66
67     result = factorial(n);
68     printf("%d\n", result);
69
70     int arr[5] = {1, 2, 3, 4, 5};
71     arr[0] = 10;
72     result = arr[0] + arr[1];
73     printf("%d\n", result);
74
75     return 0;
76 }
77
78 int max(int a, int b) {
79     if (a > b) {
80         return a;
81     } else {
82         return b;
83     }
84 }
85
86 int factorial(int n) {
87     if (n <= 1) {
88         return 1;
89     } else {
90         return n * factorial(n - 1);
91     }
92 }
```

上述代码包含了编译器支持的大部分语句操作（包括数值运算，赋值语句，分支语句，循环语句，函数定义和调用等），运行结果如下图：

```

● reca@RecA-computer:~/NKU-Compiling-2025-Lab/LAB1$ ./test
50
22
10
120
12

```

图 20: 源 SysY 程序运行结果

下面展示我们小组完成的与上述程序等价的 LLVM IR 程序和 ARM 汇编程序。

## (二) LLVM IR 编程

强博负责的 LLVM IR 的编程, 完成的等价程序如下:

```

1  @.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
2
3  declare i32 @printf(ptr noundef, ...) #1
4
5  ; 定义函数max
6  define dso_local i32 @max(i32 noundef %0, i32 noundef %1) #0 {
7      %3 = alloca i32, align 4
8      %4 = alloca i32, align 4
9      %5 = alloca i32, align 4
10     store i32 %0, ptr %4, align 4
11     store i32 %1, ptr %5, align 4
12     %6 = load i32, ptr %4, align 4
13     %7 = load i32, ptr %5, align 4
14     %8 = icmp sgt i32 %6, %7
15     br i1 %8, label %9, label %11
16
17  9:
18     %10 = load i32, ptr %4, align 4
19     store i32 %10, ptr %3, align 4
20     br label %13
21
22  11:
23     %12 = load i32, ptr %5, align 4
24     store i32 %12, ptr %3, align 4
25     br label %13
26
27  13:
28     %14 = load i32, ptr %3, align 4
29     ret i32 %14
30 }
31
32 ; 定义函数factorial
33 define dso_local i32 @factorial(i32 noundef %0) #0 {
34     %2 = alloca i32, align 4
35     %3 = alloca i32, align 4

```

```

36     store i32 %0, ptr %3, align 4
37     %4 = load i32, ptr %3, align 4
38     %5 = icmp sle i32 %4, 1
39     br i1 %5, label %6, label %7
40
41 6:
42     store i32 1, ptr %2, align 4
43     br label %13
44
45 7:
46     %8 = load i32, ptr %3, align 4
47     %9 = load i32, ptr %3, align 4
48     %10 = sub nsw i32 %9, 1
49     %11 = call i32 @factorial(i32 noundef %10)
50     %12 = mul nsw i32 %8, %11
51     store i32 %12, ptr %2, align 4
52     br label %13
53
54 13:
55     %14 = load i32, ptr %2, align 4
56     ret i32 %14
57 }
58
59 ;定义主函数
60 define dso_local i32 @main() #0 {
61     ;定义临时变量
62     %1 = alloca i32, align 4
63     %2 = alloca i32, align 4
64     %3 = alloca i32, align 4
65     %4 = alloca i32, align 4
66     %5 = alloca i32, align 4
67     %6 = alloca i32, align 4
68     %7 = alloca [5 x i32], align 16
69     ;变量赋值初始化
70     store i32 0, ptr %1, align 4
71     store i32 10, ptr %2, align 4
72     store i32 5, ptr %3, align 4
73     store i32 0, ptr %4, align 4
74     store i32 0, ptr %5, align 4
75     store i32 5, ptr %6, align 4
76
77     ; 执行result = a + b
78     %8 = load i32, ptr %2, align 4
79     %9 = load i32, ptr %3, align 4
80     %10 = add nsw i32 %8, %9
81     store i32 %10, ptr %4, align 4
82
83     ; 执行result = a - b

```

```
84      %11 = load i32, ptr %2, align 4
85      %12 = load i32, ptr %3, align 4
86      %13 = sub nsw i32 %11, %12
87      store i32 %13, ptr %4, align 4
88
89      ; 执行 result = a * b
90      %14 = load i32, ptr %2, align 4
91      %15 = load i32, ptr %3, align 4
92      %16 = mul nsw i32 %14, %15
93      store i32 %16, ptr %4, align 4
94
95      ; 执行 printf("%d\n", result)
96      %17 = load i32, ptr %4, align 4
97      %18 = call i32 @printf(ptr noundef @.str, i32 noundef %17)
98
99      ; 执行 result = a / b
100     %19 = load i32, ptr %2, align 4
101     %20 = load i32, ptr %3, align 4
102     %21 = sdiv i32 %19, %20
103     store i32 %21, ptr %4, align 4
104
105     ; 执行 result = a % b
106     %22 = load i32, ptr %2, align 4
107     %23 = load i32, ptr %3, align 4
108     %24 = srem i32 %22, %23
109     store i32 %24, ptr %4, align 4
110
111     ; 执行 result = a
112     %25 = load i32, ptr %2, align 4
113     store i32 %25, ptr %4, align 4
114
115     ; 执行 result = result + 1
116     %26 = load i32, ptr %4, align 4
117     %27 = add nsw i32 %26, 1
118     store i32 %27, ptr %4, align 4
119
120     ; 执行 result = result * 2
121     %28 = load i32, ptr %4, align 4
122     %29 = mul nsw i32 %28, 2
123     store i32 %29, ptr %4, align 4
124
125     ; 执行 printf("%d\n", result)
126     %30 = load i32, ptr %4, align 4
127     %31 = call i32 @printf(ptr noundef @.str, i32 noundef %30)
128
129     ; 实现分支判断 if (a > b)
130     %32 = load i32, ptr %2, align 4
131     %33 = load i32, ptr %3, align 4
```

```

132     %34 = icmp sgt i32 %32, %33
133     br i1 %34, label %35, label %36
134
135 35:
136     store i32 1, ptr %4, align 4
137     br label %37
138
139 36:
140     store i32 0, ptr %4, align 4
141     br label %37
142
143 37:
144     ; 执行分支判断 if (a > 0 && b > 0)
145     %38 = load i32, ptr %2, align 4
146     %39 = icmp sgt i32 %38, 0
147     br i1 %39, label %40, label %44
148
149 40:
150     %41 = load i32, ptr %3, align 4
151     %42 = icmp sgt i32 %41, 0
152     br i1 %42, label %43, label %44
153
154 43:
155     store i32 1, ptr %4, align 4
156     br label %44
157
158 44:
159     ; 执行分支判断 if (a > 0 || b > 0)
160     %45 = load i32, ptr %2, align 4
161     %46 = icmp sgt i32 %45, 0
162     br i1 %46, label %50, label %47
163
164 47:
165     %48 = load i32, ptr %3, align 4
166     %49 = icmp sgt i32 %48, 0
167     br i1 %49, label %50, label %51
168
169 50:
170     store i32 1, ptr %4, align 4
171     br label %51
172
173 51:
174     ; 执行多重分支判断 if (a > b) ... else if (a < b) ... else ...
175     %52 = load i32, ptr %2, align 4
176     %53 = load i32, ptr %3, align 4
177     %54 = icmp sgt i32 %52, %53
178     br i1 %54, label %55, label %59
179

```

```

180 55:
181     %56 = load i32, ptr %2, align 4
182     %57 = load i32, ptr %3, align 4
183     %58 = call i32 @max(i32 noundef %56, i32 noundef %57)
184     store i32 %58, ptr %4, align 4
185     br label %67
186
187 59:
188     %60 = load i32, ptr %2, align 4
189     %61 = load i32, ptr %3, align 4
190     %62 = icmp slt i32 %60, %61
191     br i1 %62, label %63, label %65
192
193 63:
194     %64 = load i32, ptr %3, align 4
195     store i32 %64, ptr %4, align 4
196     br label %66
197
198 65:
199     store i32 0, ptr %4, align 4
200     br label %66
201
202 66:
203     br label %67
204
205 67:
206     ; 执行 printf("%d\n", result)
207     %68 = load i32, ptr %4, align 4
208     %69 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %68)
209
210     ; 执行赋值语句 i = 0 和 result = 0
211     store i32 0, ptr %5, align 4
212     store i32 0, ptr %4, align 4
213     br label %70
214
215 70:
216     ; 实现循环 while (i <= 10)
217     %71 = load i32, ptr %5, align 4
218     %72 = icmp sle i32 %71, 10
219     br i1 %72, label %73, label %80
220
221 73:
222     ; 执行 result 和 i 的自增
223     %74 = load i32, ptr %4, align 4
224     %75 = load i32, ptr %5, align 4
225     %76 = add nsw i32 %74, %75
226     store i32 %76, ptr %4, align 4
227     %77 = load i32, ptr %5, align 4

```

```

228      %78 = add nsw i32 %77, 1
229      store i32 %78, ptr %5, align 4
230      br label %70, !llvm.loop !6
231
232 79:
233      br label %80
234
235 80:
236      store i32 0, ptr %4, align 4
237      ; 实现循环for (i = 0; i < 10; i = i + 1)
238      store i32 0, ptr %5, align 4
239      br label %81
240
241 81:
242      %82 = load i32, ptr %5, align 4
243      %83 = icmp slt i32 %82, 10
244      br i1 %83, label %84, label %106
245
246 84:
247      ;实现分支if (i % 2 == 0)
248      %85 = load i32, ptr %5, align 4
249      %86 = srem i32 %85, 2
250      %87 = icmp eq i32 %86, 0
251      br i1 %87, label %88, label %89
252
253 88:
254      br label %100
255
256 89:
257      ; 执行result = result + i
258      %90 = load i32, ptr %4, align 4
259      %91 = load i32, ptr %5, align 4
260      %92 = add nsw i32 %90, %91
261      store i32 %92, ptr %4, align 4
262      ; 实现分支if (result > 20) break
263      %93 = load i32, ptr %4, align 4
264      %94 = icmp sgt i32 %93, 20
265      br i1 %94, label %95, label %96
266
267 95:
268      br label %106
269
270 96:
271      br label %97
272
273 97:
274      br label %98
275

```

```

276 98:
277     br label %99
278
279 99:
280     br label %100
281
282 100:
283     ; 实现i自增
284     %101 = load i32, ptr %5, align 4
285     %102 = add nsw i32 %101, 1
286     store i32 %102, ptr %5, align 4
287     br label %81, !llvm.loop !8
288
289 103:
290     br label %81
291
292 104:
293     br label %105
294
295 105:
296     br label %106
297
298 106:
299     ; 函数调用 factorial(n)
300     %107 = load i32, ptr %6, align 4
301     %108 = call i32 @factorial(i32 noundef %107)
302     store i32 %108, ptr %4, align 4
303     %109 = load i32, ptr %4, align 4
304     %110 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef
305         %109)
306
307     ; 实现数组赋值int arr[5] = {1, 2, 3, 4, 5};
308     %111 = getelementptr inbounds [5 x i32], ptr %7, i64 0, i64 0
309     store i32 1, ptr %111, align 16
310     %112 = getelementptr inbounds [5 x i32], ptr %7, i64 0, i64 1
311     store i32 2, ptr %112, align 4
312     %113 = getelementptr inbounds [5 x i32], ptr %7, i64 0, i64 2
313     store i32 3, ptr %113, align 8
314     %114 = getelementptr inbounds [5 x i32], ptr %7, i64 0, i64 3
315     store i32 4, ptr %114, align 4
316     %115 = getelementptr inbounds [5 x i32], ptr %7, i64 0, i64 4
317     store i32 5, ptr %115, align 16
318
319     ; 实现数组元素赋值arr[0] = 10
320     %116 = getelementptr inbounds [5 x i32], ptr %7, i64 0, i64 0
321     store i32 10, ptr %116, align 16
322
323     %117 = getelementptr inbounds [5 x i32], ptr %7, i64 0, i64 0

```



```

323     %118 = load i32, ptr %117, align 16
324     %119 = getelementptr inbounds [5 x i32], ptr %7, i64 0, i64 1
325     %120 = load i32, ptr %119, align 4
326     %121 = add nsw i32 %118, %120
327     store i32 %121, ptr %4, align 4
328
329     %122 = load i32, ptr %4, align 4
330     %123 = call i32 @printf(ptr noundef @.str, i32 noundef
331         %122)
332
333     ret i32 0
334 }
335 !llvm.module.flags = [{!0, !1, !2, !3, !4}]
336 !llvm.ident = [{!5}]
337
338 !0 = [{i32 1, !"wchar_size", i32 4}]
339 !1 = [{i32 8, !"PIC Level", i32 2}]
340 !2 = [{i32 7, !"PIE Level", i32 2}]
341 !3 = [{i32 7, !"uwtable", i32 2}]
342 !4 = [{i32 7, !"frame-pointer", i32 2}]
343 !5 = [{!"Ubuntu clang version 18.1.3 (1ubuntu1)"}]
344 !6 = distinct [{!6, !7}]
345 !7 = [{!"llvm.loop.mustprogress"}]
346 !8 = distinct [{!8, !7}]

```

使用命令: `clang -o mytest test_llvm.ll libsys_x86.a` 和 `lli -load=./sylib.so test_llvm.ll` 进行验证, 最终上述 `test_llvm.ll` 程序进行编译链接后运行结果如下图:

```

● reca@RecA-computer:~/NKU-Compiling-2025-Lab/LAB1$ clang -o mytest test_llvm.ll libsys_x86.a
warning: overriding the module target triple with x86_64-pc-linux-gnu [-Woverride-module]
1 warning generated.
● reca@RecA-computer:~/NKU-Compiling-2025-Lab/LAB1$ ./mytest
50
22
10
120
12
● reca@RecA-computer:~/NKU-Compiling-2025-Lab/LAB1$ lli -load=./sylib.so test_llvm.ll
50
22
10
120
12
TOTAL: 0H-0M-0S-0us

```

图 21: 等价 LLVM 程序运行结果

下面, 对 LLVM 编写的程序中的各语言特性进行说明:

- 局部变量的声明均以% 开头表示虚拟寄存器, 并使用 `alloca` 关键字为其分配对应的内存空间, `i32` 表示的是局部变量的数据类型是 32 位整型变量。
- 数组类型的变量声明使用 “`alloca [5 x i32]`” 表示数组元素个数位 5, 类型为 `i32`。访问数组元素时需要使用 `getelementptr` 关键字, 以语句 “`%116 = getelementptr inbounds [5 x i32], ptr %7, i64 0, i64 0`” 为例, `[5 x i32]` 表示将计算具有 5 个 `i32` 元素的数组的地址, 基址为 `ptr %7`, 后面的两个索引中第二个表示要访问元素的下标。

- 函数定义使用 `define` 关键字，由于本程序均为全局函数，所以函数名前都用 `@` 修饰，调用函数时使用 `call` 关键字，后面接函数返回类型、函数名和实参列表。函数结束后使用关键字 `ret` 实现返回功能
- 分支控制语句和循环语句实现的关键是使用 `br` 关键字实现跳转，可以用于无条件跳转也可以用于条件跳转。
- 本程序实现的算术运算符有 `+`, `-`, `*`, `/`, `%`，其对应的关键字分别为 `add`, `sub`, `mul`, `sdiv`, `srem`。实现关系运算需要声明关键字 `icmp`，后面紧跟需要实现的关系运算，包括小于 (`slt`)、大于 (`sgt`)、小于等于 (`sle`)、等于 (`eq`) 等。

## 四、 实验三：ARM 汇编编程

注：陈语童（2311887）负责此实验的代码编写、探究和报告撰写。

为了更全面地探究编译器支持的 SysY 语言特性，编写了全新的 SysY 程序，用于上面一节的 LLVM IR 程序实验探究，和本节的 ARM 汇编程序实验探究。

### （一） SysY 程序

新的完整的 SysY 程序已经在 3.1 节中给出。

此 SysY 程序所涉及到的语言特性，全面涵盖了赋值、数值运算的基本操作，条件分支、循环等复合语句，数组的声明和调用，以及自定义函数的特性。

直接编译该 SysY 程序，得到下面的输出：

```
● reca@RecA-computer:~/NKU-Compiling-2025-Lab/LAB1$ ./test
50
22
10
120
12
```

图 22: 预期输出结果

则最终编写的汇编程序在经过汇编器转化和最后的链接加载后得到的可执行程序，运行后也应该得到这样的结果。

### （二） ARM 汇编代码设计过程

下面按照自然代码顺序，分块阐述 ARM 汇编代码设计过程。

首先必须明确的是，ARM 汇编中的寄存器编码与 x86 不同。32 寄存器系 `w0`, `w1`, ... , 64 寄存器系 `x0`, `x1`, ... , 编写时需注意与 x86 区分、寄存器之间区分。

汇编文件起始，先参考了前面斐波那契数列生成的 ARM 汇编语言代码，按照规则编写文件头，并全局地声明了一个必要的字面常量：

```

1      .arch armv8-a
2      .file      "example.c"
3
4      .text
5      .section      .rodata
6      .align  3
7  .LC1:
8      .string  "%d\n"

```

其中 `.arch armv8-a` 指明架构是 ARMv8-A (AArch64), `.rodata` 表示只读数据段, 后面声明一个用于 `printf()` 函数中的常量字符串参数 `"%d\n"`, `LC` 表示字面常量。

```

1      .global main
2      .type     main, %function
3  main:

```

进入 `main` 函数。

```

1      sub      sp, sp, #80          ; 在栈上分配 80 字节空间
2      stp      x29, x30, [sp, 64]   ; 保存前一帧的帧指针和返回地址
3      add      x29, sp, 64          ; 设置新的帧指针

```

为函数创建栈帧的基本操作。得到栈的结构大致如下所示:

```

1  sp+0   → 临时变量
2  sp+12  → 用于存放中间结果
3  sp+16  → 其他局部变量
4  ...
5  sp+64  → 保存 x29
6  sp+72  → 保存 x30

```

```

1  mov      w0, 10
2  str      w0, [sp, 20]
3  mov      w0, 5
4  str      w0, [sp, 24]
5  str      wzr, [sp, 12]
6  str      wzr, [sp, 16]
7  mov      w0, 5
8  str      w0, [sp, 28]

```

这一部分与**赋值**的语言特性有关。观察之前工作过程中 ARM 汇编语言代码, 发现与 x86 汇编语言不同的是, 可以使用 `[基址, 偏移量]` 的格式来指示地址, 较为方便。此处就采用这样的写法。其中 `mov`, `str` 分别是 ARM 中数据移动和寄存器中数据写入内存地址的汇编指令。这一部分对应原 SysY 代码的:

```

1      int a = 10;
2      int b = 5;
3      int result = 0;
4      int i = 0;
5      int n = 5;

```

下面三段汇编代码与基本四则运算的语言特性有关。

```
1 ldr    w1, [sp, 20]
2 ldr    w0, [sp, 24]
3 add    w0, w1, w0
4 str    w0, [sp, 12]
```

第一段是  $result = a + b$ ；对应的运算。ldr 是 str 的逆操作，将数据从内存中读取到寄存器中。add 则显然易见地将两个操作数相加存入目标寄存器中的指令。那么上面编写的汇编指令意义就是，先将 a, b 的数据分别读取到寄存器中，再将两个寄存器的值相加存到其中一个寄存器中作为结果，最后再将这个结果寄存器的数值写入内存地址，对应的是 result 变量的地址。

```
1 ldr    w1, [sp, 20]
2 ldr    w0, [sp, 24]
3 sub    w0, w1, w0
4 str    w0, [sp, 12]
```

第二段是  $result = a - b$ ；对应的运算。基本原理一致，这次使用到 sub 减法操作。可以发现三次读取/写入数据的地址和上面加法中是完全一致的，则 result 的值在这个过程中被不可逆地覆盖。

```
1 ldr    w1, [sp, 20]
2 ldr    w0, [sp, 24]
3 mul    w0, w1, w0
4 str    w0, [sp, 12]
```

第三段是  $result = a * b$  对应的运算。ARM 中也有直接的 mul 乘法操作。

```
1 ldr    w1, [sp, 12]
2 adrp   x0, .LC1
3 add    x0, x0, :lo12:.LC1
4 bl     printf
```

这里涉及到对上面计算结果的**输出**，也是 SysY 语言一大特性。

第一行，将要打印的数加载到第二参数上；第二行和第三行共同作用将之前标记为 .LC1 的字符串常量 "%d\n" 加载到第一参数上；最后一行通过 bl 指令（带链接的跳转），直接跳转到外部函数 printf 进行输出。

后续调用自定义的函数时，也可以直接使用 bl 进行跳转。其中字符串常量的加载中，adrp 只将 .L1 标签的页面的基地址加载到 x0 寄存器，通过下一语句的加法，.LC1 标签的低 12 位偏移量被加和到 x0 上，才完成完整地址的计算，指向目标字符常量。

接下来还有除法和取余的运算，和上面四则运算的基本原理相同。但值得注意的是，取余运算没有直接的指令符，需要拆分成“除法 → 乘法 → 减法”三步，才能计算出余数：

```
1 ldr    w0, [sp, 20]
2 ldr    w1, [sp, 24]
3 sdiv   w2, w0, w1          ; 先做整除法，向下取整
4 ldr    w1, [sp, 24]
5 mul    w1, w2, w1          ; 乘法，计算出已经被整除的部分
6 sub    w0, w0, w1          ; 减法，减去已被整除的部分得到余数
```

```
7  str    w0, [sp, 12]
```

此后的自增与左移都能够直接或间接地通过单条运算指令来解决，不再单独展示。

下面展示部分**条件判断**语句的编写。

```
1  ldr    w1, [sp, 20]
2  ldr    w0, [sp, 24]
3  cmp    w1, w0
4  ble    .L2
5  mov    w0, 1
6  str    w0, [sp, 12]
7  b      .L3
8  .L2:
9  str    wzr, [sp, 12]
10 .L3:
11 . . . . .
```

条件判断的逻辑起点是比较。使用 `cmp` 进行两个数的比较，这里是对 `a`, `b` 两个参量比较。之后来到判断之后分支逻辑的核心，`ble` 表示小于等于时跳转，`b` 表示无条件跳转。从源代码分支逻辑的分析出发，关键区别在于不同情况下给 `result` 赋的值不同，那么就需要利用跳转将这一分支实现，两个赋值语句在两种情况下均为“有其中一条语句被执行而另一条不被执行的状态”。所以选择用 `.L2` 标签隔离不满足条件时的赋值语句，同时用 `b` 指令保证满足条件时跳过被隔离的赋值语句。此外还需要注意的点是，赋零不能直接用数字 `0`，而要用 `#0` 立即数 `0`，或者像这里的编写一样，使用 `wzr` 零寄存器来赋值，更高效与保险。综上，这段汇编代码能够成功复现下面原 SysY 代码中的条件分支：

```
1  if (a > b) {
2      result = 1;
3  } else {
4      result = 0;
5  }
```

再往后分析一个代表性条件分支。

```
1  ldr    w0, [sp, 20]
2  cmp    w0, 0
3  ble    .L4
4  ldr    w0, [sp, 24]
5  cmp    w0, 0
6  ble    .L4
7  mov    w0, 1
8  str    w0, [sp, 12]
9  .L4:
10 . . . . .
```

这里的条件分支中有双重条件。思路是，只要有一个条件不被满足，就直接跳转到不满足条件的后续逻辑。所以，在上面编写的代码可以看到，两次 `cmp` 之后都进行了 `ble` 的判断是否跳转。如果两次都满足——都没有跳转，才能执行到 `.L4` 标签前的那一段赋值操作。这段汇编代码能够复现下面原 SysY 代码中的条件分支：

```

1  if (a > 0 && b > 0) {
2      result = 1;
3  }

```

此后条件分支的汇编翻译过程大同小异，最多只是进行进一步的逻辑嵌套，不再赘述。

下面再简单分析一下怎么写**循环语句块**的 ARM 汇编代码。

```

1  str    wzr, [sp, 12]
2  str    wzr, [sp, 16]
3  b      .L10
4  .L11:
5      . . . . . ; 循环内部的实际逻辑
6  .L10:
7  ldr    w0, [sp, 16]
8  cmp    w0, 10
9  ble    .L11

```

一般在每一次循环之后，进行条件的复查，判断是否需要继续进行下一轮循环。所以判断逻辑理所应当地排在循环体实际内容之后，并且在循环实际内容的开头设置标签，方便判断后的跳转。但初始状态，也需要进行一次判断，所以像上面汇编代码中所示，将判断部分也要单独进行标签标记，第一次到达循环体前跳转到判断逻辑进行初次判断，这样才符合原代码逻辑。上面实现的循环在原 SysY 代码中是这样的：

```

1  while (i <= 10) {
2      . . . . . // 实际循环内容
3  }

```

for 循环的实现是同理的，不再介绍如何编写。

在源程序的主函数最后，还进行了**数组的声明和赋值**。

```

1  .LC0:
2      .word    1
3      .word    2
4      .word    3
5      .word    4
6      .word    5

```

数组的声明，在主函数之后单独编写。其中 .LC0 为这里的数组进行字符常量的唯一标识；5 个 .word 是声明 32 位（4 字节）的数据的作用，即 int 类型数据，后面跟随的是实际数值。所以上面的声明用 SysY 写出来就是

```

1  adrp    x0, .LC0
2  add     x0, x0, :lo12:LC0
3  add     x2, sp, 32
4  mov     x3, x0
5  ldp     x0, x1, [x3]
6  stp     x0, x1, [x2]
7  ldr     w0, [x3, 16]
8  str     w0, [x2, 16]

```

实际主函数中，并不能直接使用声明的字符常量数组，而要将其全部复制形成一个新的可以修改值的数组。上面的汇报代码中，前两行之前分析过，是获得字符常量实际地址的作用。下面分配新的数组在栈上的位置，并保存前面获得的字符常量数组地址。后面四行是数组复制的一个以缩减代码量、提高效率的写法。ldp 和 stp 都能够一次读取/加载 16 字节数据，那么为了获得一个 5 项 32 位数据，只需要两组操作即可（一次移动 16 字节 = 16\*8 位 = 4 项），第二组操作进行 16 字节的偏移即可。

因此，int arr[5] = 1, 2, 3, 4, 5; 这一句声明语句不仅需要在主函数后声明常量，还需要写如上的汇编语句将其复制为可操作的局部变量。

后续对于数组数据的修改，和之前将数据直接存储到内存是一个道理，不再赘述了。

示例 SysY 程序还尝试编写了**自定义函数**，函数逻辑主体应该写在主函数结束后。

```

1      .global max
2      .type max, %function
3 max:
4      cmp w0, w1
5      ble .L22
6      ret
7 .L22:
8      mov w0, w1
9      ret

```

先看 max 较大值函数。首先进行一些设置，.global 表示全局函数全局可用；%function 声明这是一个函数。下面进行两个传入参数的大小比较，如果满足条件，默认返回 w0 对应的参数值；否则将另一参数赋给 w0 寄存器并返回值。

```

1      .global factorial
2      .type factorial, %function
3 factorial:
4      stp x29, x30, [sp, -32]!
5      mov x29, sp
6      str w0, [sp, 28]
7      ldr w0, [sp, 28]
8      cmp w0, 1
9      bgt .L23
10     mov w0, 1
11     b .L24
12 .L23:
13     ldr w0, [sp, 28]
14     sub w0, w0, #1
15     bl factorial
16     mov w1, w0
17     ldr w0, [sp, 28]
18     mul w0, w1, w0
19 .L24:
20     ldp x29, x30, [sp], 32
21     ret

```

再看 factorial 阶乘函数，此函数涉及到递归。头部声明不再解析。最开始几行做简单的

栈帧分配操作，并将传入的参数  $n$  入栈。接下来用 `cmp` 和 `bgt`（如果大于跳转）组合做  $n \leq 1$  的比较，处理 0 和 1 的阶乘直接返回 1。而如果大于 1 则跳转到 .L23 进行递归处理（对应 `else` 语句块），易理解接下来的代码在进行下面几件事：计算  $n - 1$ ，以计算得到的  $n - 1$  参数递归地调用 `factorial` 函数，将递归结果与  $n$  相乘并作为返回结果。

在所有用到栈的函数体最后，都需要清理和恢复栈帧。对于本程序，`main` 函数和 `factorial` 函数都用到了栈帧，所以需要依次做如下操作：

```

1 ; function: main
2     ldp    x29, x30, [sp, 64]
3     add    sp, sp, 80
4     ret
5 ; function: factorial
6 .L24:
7     ldp    x29, x30, [sp], 32
8     ret

```

由于篇幅限制不能够逐行解析汇编代码的编写，但未展示的部分与已经分析的部分基本逻辑相近，故不再赘述。

### （三） ARM 汇编代码综合展示

为了节省报告篇幅，完整的 ARM 汇编代码不在此处直接展示。

敬请直接跳转到 GitHub 仓库阅读：[exampleARM.S](#)

### （四） 正确性验证

将代码用以下的指令进行汇编处理和链接加载，获得最终的可执行文件：

```

example-SysY$ aarch64-linux-gnu-gcc exampleARM.S -c -o exampleARM.o
example-SysY$ aarch64-linux-gnu-gcc -o exampleARM exampleARM.o -static -lc
example-SysY$ qemu-aarch64 ./exampleARM

```

图 23: 获得可执行文件

使用 `qemu` 运行 ARM 可执行文件：

```

bugp3ssy666@DESKTOP-ALUTGKM:~/course-compiler-basics/assignment01/example-SysY$ qemu-aarch64 ./exampleARM
50
22
10
120
12

```

图 24: 运行结果

可以看到，这和之前的预期输出结果（图 17）是完全一致的。



### (五) 思考与启发

将高级语言翻译成汇编语言是一个很复杂的过程，因为两者基本逻辑不同。高级语言顺着人自然思考的范式而展开，汇编语言则需要根据机器运行的顺序依次指示。而且汇编语言自由度低，不能够像高级语言那样几乎是随心所欲地重命名参数、函数，比如寄存器等名称是既定不可更改的，会给程序的编写带来一定困惑。

然而编写过程中也感受到，相对于 x86 的汇编语言，ARM 汇编语言似乎能编写地更加美观和简洁，受益于简单易懂的标准参数名称、指令名称，以及一些便捷的逻辑表达方式，比如访存时的方括号格式。

在此之前我修读过汇编语言这门课程，以上的实验让我温故而知新，对于汇编语言有了更深刻的了解和认识。

**至此，实验三全部完成。**

## 五、 总结

本次实验是课程后续主要实验的前置实验，着重分析探究了编译流程和编译器各阶段功能，并进行了 LLVM IR 中间代码程序和 ARM 汇编代码程序的编写。

在实验一中，从理论出发，用各种 Clang/GCC 等命令，将 C 语言程序从 .c 源文件，到中间语言程序文件、汇编语言程序文件、机器码文件，再到最终的可执行文件这一完整编译过程完整走了一遍。在这当中，主要分析了编译器各个阶段做的工作，并进行了机器进程操作层面的前后代码对比，尽可能深入地探究其功能。

在实验三中，根据之前的汇编语言知识和阅读经验，尝试编写和优化 SysY 程序的等价 ARM 汇编语言。结果是成功的，运行后得到了预期结果。在汇编语言的编写过程中，对其有了语法语义上的更深一步了解，并且能够与其他汇编语言联想对比，分析 ARM 汇编语言的特性。

**实验代码的 GitHub 仓库：**

陈语童 (2311887)： [NKU-compiler-basics/Lab0](https://github.com/2311887/NKU-compiler-basics)

强博 (2313825)： [NKU-Compiling-2025-Lab/LAB1](https://github.com/2313825/NKU-Compiling-2025-Lab)