



南開大學  
Nankai University

计算机学院  
编译原理第一次试验

认识编译器

姓名：王小红 陈小明

学号：20XXXXXXXX

专业：计算机科学与技术

2025 年 9 月 27 日

## 目录

<b>1 第一题：编译过程分析</b>	<b>2</b>
1.1 预处理器	2
1.2 编译器	3
1.3 汇编器	4
1.4 链接器	5
1.5 编译器内部的过程	6
1.5.1 词法分析：代码的“分词”	6
1.5.2 语法分析：构建句法结构	6
1.5.3 语义分析与中间代码	7
1.5.4 GIMPLE：接近源码的高级表示	7
1.5.5 代码优化：从接近高级语言的 GIMPLE 到接近机器语言的 RTL	8
1.5.6 RTL：面向机器的低级表示	9
<b>2 第二题：汇编代码撰写</b>	<b>10</b>
2.1 llvm_IR 设计解释	13
2.1.1 模块头与全局	13
2.1.2 main 的局部变量与入口（prologue）	13
2.1.3 循环控制流（while i < count）	13
2.1.4 条件分支（if/else）与函数调用	14
2.1.5 calculate 函数	14
2.1.6 外部函数声明与元数据	14

# 1 第一题：编译过程分析

## 1.1 预处理器

预处理器是编译过程的第一阶段，主要负责处理源代码中的预处理指令，如宏定义、文件包含、条件编译等。在 GCC 中，预处理器由 `cpp` 工具执行。

以 `fibonacci.c` 为例，源代码包含了 `#include <stdio.h>` 和 `#define MAX_ITERATIONS 10`。预处理器会展开这些指令：将 `stdio.h` 的内容插入到代码中，替换宏定义等。生成的 `fibonacci.i` 文件就是预处理后的中间文件，其中包含了所有必要的头文件定义和宏展开，但不包含注释和预处理指令。

对比 `fibonacci.c` 和 `fibonacci.i`：`fibonacci.c` 中只有几行代码，而 `fibonacci.i` 包含了 `stdio.h` 的完整定义，包括函数声明如 `printf`、`putchar` 等。宏 `MAX_ITERATIONS` 被替换为 `10`。预处理器还添加了行号信息和编译器版本等元数据。

之前 (`fibonacci.c` 的 `main` 函数部分)：

```
1 #define MAX_ITERATIONS 10
2
3 int main() {
4     int a = 0, b = 1, next, i;
5
6     printf("Fibonacci Series up to %d terms:\n", MAX_ITERATIONS);
7
8     for (i = 0; i < MAX_ITERATIONS; i++) {
9         if (i <= 1) {
10             next = i;
11         } else {
12             next = a + b;
13             a = b;
14             b = next;
15         }
16         printf("%d ", next);
17     }
18     printf("\n");
19     return 0;
20 }
```

之后 (`fibonacci.i` 的 `main` 函数部分)：

```
1 int main() {
2     int a = 0, b = 1, next, i;
3
4     printf("Fibonacci Series up to %d terms:\n", 10);
5
6     for (i = 0; i < 10; i++) {
7         if (i <= 1) {
8             next = i;
9         } else {
10             next = a + b;
11             a = b;
```

```
12         b = next;
13     }
14     printf("%d ", next);
15 }
16 printf("\n");
17 return 0;
18 }
```

## 1.2 编译器

编译器将预处理后的代码转换为汇编代码。在 GCC 中，编译器前端（cc1）执行以下阶段：

词法分析：将源代码分解成 token，如关键字（int, for, if）、标识符（main, a, b）、运算符（+, =）、常量（10）等。

语法分析：根据语法规则构建抽象语法树（AST），识别程序结构如函数定义、循环、条件语句。

语义分析：检查类型一致性、作用域等，进行类型检查和符号表构建。例如，确保变量类型匹配，函数调用正确。

中间代码生成：生成中间表示（IR），如 GIMPLE 或 RTL。GCC 使用 GIMPLE 作为高级 IR，然后转换为 RTL。

优化：进行各种优化，如常量折叠（将 MAX\_ITERATIONS 替换为 10）、死代码消除、循环优化等。在 fibonacci 程序中，循环被优化为汇编中的 jmp 和 cmp 指令。

代码生成：将 IR 转换为目标平台的汇编代码。

从 fibonacci.i 到 fibonacci.s，编译器生成了 x86-64 汇编代码。fibonacci.s 包含了 .text 段、.rodata 段（字符串常量）、main 函数的汇编实现。汇编代码使用了栈来管理局部变量（如 a, b, next, i），并调用 printf 和 putchar。相比 .i 文件，.s 文件是平台特定的机器指令序列。

之前（fibonacci.i 的 main 函数部分）：

```
1 int main() {
2     int a = 0, b = 1, next, i;
3
4     printf("Fibonacci Series up to %d terms:\n", 10);
5
6     for (i = 0; i < 10; i++) {
7         if (i <= 1) {
8             next = i;
9         } else {
10            next = a + b;
11            a = b;
12            b = next;
13        }
14        printf("%d ", next);
15    }
16    printf("\n");
17    return 0;
18 }
```

之后（fibonacci.s 的 main 函数部分）：

```
1  .globl  main
2  .type  main, @function
3  main:
4  .LFB0:
5  .cfi_startproc
6  endbr64
7  pushq %rbp
8  .cfi_def_cfa_offset 16
9  .cfi_offset 6, -16
10 movq %rsp, %rbp
11 .cfi_def_cfa_register 6
12 subq $16, %rsp
13 movl $0, -16(%rbp)
14 movl $1, -12(%rbp)
15 movl $10, %esi
16 leaq .LC0(%rip), %rax
17 movq %rax, %rdi
18 movl $0, %eax
19 call printf@PLT
20 movl $0, -4(%rbp)
21 jmp .L2
22 .L2:
```

### 1.3 汇编器

汇编器将汇编代码转换为机器码，生成目标文件（.o 文件）。在 GCC 中，由 as 工具执行。

fibonacci.s 文件包含了汇编指令，如 mov, add, call 等。汇编器将其翻译成二进制机器码，并组织成 ELF 格式的目标文件，包括代码段 (.text)、数据段 (.rodata) 等。fibonacci.o.asm 是目标文件的反汇编，显示了机器码（如 0xf3 0f 1e fa 对应 endbr64）和对应的汇编指令。目标文件包含了符号表，未解析的外部符号（如 printf, putchar）标记为未定义，需要链接器解决。

对比 fibonacci.s 和 fibonacci.o.asm：.s 是文本汇编，.o.asm 是二进制反汇编，显示了实际的十六进制机器码和地址。

之前（fibonacci.s 的 main 函数部分）：

```
1  .globl  main
2  .type  main, @function
3  main:
4  .LFB0:
5  .cfi_startproc
6  endbr64
7  pushq %rbp
8  .cfi_def_cfa_offset 16
9  .cfi_offset 6, -16
10 movq %rsp, %rbp
11 .cfi_def_cfa_register 6
12 subq $16, %rsp
```

```

13  movl  $0, -16(%rbp)
14  movl  $1, -12(%rbp)
15  movl  $10, %esi
16  leaq   .LC0(%rip), %rax
17  movq   %rax, %rdi
18  movl  $0, %eax
19  call   printf@PLT

```

之后 (fibonacci.o.asm 的 main 函数部分):

```

1  0000000000000000 <main>:
2      0: f3 0f 1e fa          endbr64
3      4: 55                      push   %rbp
4      5: 48 89 e5                mov     %rsp,%rbp
5      8: 48 83 ec 10             sub     $0x10,%rsp
6      c: c7 45 f0 00 00 00 00  movl    $0x0,-0x10(%rbp)
7     13: c7 45 f4 01 00 00 00  movl    $0x1,-0xc(%rbp)
8     1a: be 0a 00 00 00          mov     $0xa,%esi
9     1f: 48 8d 05 00 00 00 00  lea     0x0(%rip),%rax      # 26 <main+0x26>
10    26: 48 89 c7                mov     %rax,%rdi
11    29: b8 00 00 00 00          mov     $0x0,%eax
12    2e: e8 00 00 00 00          call    33 <main+0x33>

```

## 1.4 链接器

链接器将目标文件和库链接在一起，生成可执行文件。在 GCC 中，由 ld 工具执行。

链接器解析外部符号，将目标文件中的符号引用与库中的定义匹配。对于 fibonacci.o，链接器链接了 libc 中的 printf 和 putchar 函数，生成了可执行文件 fibonacci.exe。链接器还添加了启动代码（如 \_start 函数，调用 main）、PLT（Procedure Linkage Table）用于动态链接。

fibonacci.exe.asm 显示了完整的可执行文件反汇编，包括 .init, .plt, .text, .fini 段。PLT 允许延迟绑定外部函数，提高加载效率。相比 .o.asm，.exe.asm 包含了所有依赖的库代码和重定位信息，使程序可执行。

对比 fibonacci.o.asm 和 fibonacci.exe.asm：.o.asm 只有 main 函数，而 .exe.asm 包含了整个程序的布局，包括库函数的入口点和动态链接机制。

之前 (fibonacci.o.asm 的 main 函数部分):

```

1  0000000000000000 <main>:
2      0: f3 0f 1e fa          endbr64
3      4: 55                      push   %rbp
4      5: 48 89 e5                mov     %rsp,%rbp
5      8: 48 83 ec 10             sub     $0x10,%rsp
6      c: c7 45 f0 00 00 00 00  movl    $0x0,-0x10(%rbp)
7     13: c7 45 f4 01 00 00 00  movl    $0x1,-0xc(%rbp)
8     1a: be 0a 00 00 00          mov     $0xa,%esi
9     1f: 48 8d 05 00 00 00 00  lea     0x0(%rip),%rax      # 26 <main+0x26>
10    26: 48 89 c7                mov     %rax,%rdi
11    29: b8 00 00 00 00          mov     $0x0,%eax

```

```
12 2e: e8 00 00 00 00      call    33 <main+0x33>
```

之后 (fibonacci.exe.asm 的 main 函数部分):

```
1 00000000000001169 <main>:
2     1169: f3 0f 1e fa      endbr64
3     116d: 55                    push    %rbp
```

## 1.5 编译器内部的过程

预处理之后，编译过程进入核心阶段，代码将经历从高级抽象到低级机器指令的转变。GCC 使用了两种主要的中间表示 (Intermediate Representation, IR) 来完成这个过程：GIMPLE 和 RTL。

### 1.5.1 词法分析：代码的“分词”

编译器首先要像我们阅读文章一样，把一长串的源代码字符流“切分”成一个个有意义的基本单元，这些单元被称为 \*\*词法单元 (Token)\*\*。例如，‘int a = 0;’ 这行代码会被分解为：关键字 ‘int’、标识符 ‘a’、操作符 ‘=’、整型常量 ‘0’ 和分隔符 ‘;’。

我们可以通过 Clang 的 ‘-dump-tokens’ 命令得到 ‘tokens.txt’，其中记录了 ‘fibonacci.c’ 被“分词”后的结果。下面是 ‘main’ 函数开头部分的词法单元：

Listing 1: fibonacci.c 的部分词法单元 (tokens.txt)

```
1 int 'int'      [StartOfLine]  Loc=<fibonacci.c:5:1>
2 identifier 'main'      [LeadingSpace]  Loc=<fibonacci.c:5:5>
3 l_paren '('      Loc=<fibonacci.c:5:9>
4 r_paren ')'      Loc=<fibonacci.c:5:10>
5 l_brace '{'      [LeadingSpace]  Loc=<fibonacci.c:5:12>
6 int 'int'      [StartOfLine] [LeadingSpace]  Loc=<fibonacci.c:6:5>
7 identifier 'a'      [LeadingSpace]  Loc=<fibonacci.c:6:9>
8 equal '='      [LeadingSpace]  Loc=<fibonacci.c:6:11>
9 numeric_constant '0'      [LeadingSpace]  Loc=<fibonacci.c:6:13>
10 comma ',,'      Loc=<fibonacci.c:6:14>
```

每一行都清晰地标明了词法单元的类型 (如 ‘keyword’、‘identifier’)、内容 (如 ‘main’) 和它在源文件中的位置。

### 1.5.2 语法分析：构建句法结构

词法分析之后，编译器需要根据语言的语法规则，将离散的词法单元组织成一个具有层级结构的 \*\*抽象语法树 (Abstract Syntax Tree, AST)\*\*。这棵树精确地反映了代码的语法结构。如果代码存在语法错误 (例如，‘for’ 循环缺少括号)，这个阶段就会报错。

ast\_dump.txt 文件展示了 fibonacci.c 的 AST。树的根节点是 TranslationUnitDecl (翻译单元声明)，下面包含了我们定义的 main 函数。

Listing 2: main 函数的抽象语法树片段 (ast\_dump.txt)

```
1 |-FunctionDecl 0x... <<fibonacci.c:5:1, line:22:1>> line:5:5 main 'int ()'
```

```

2 | -CompoundStmt 0x... <line:5:12, line:22:1>|-DeclStmt 0x... <line:6:5, col:32>|
   |   -VarDecl 0x... <col:5, col:9> col:9 used a 'int' cinit
3 |   ...
4 |   |-ForStmt 0x... <line:9:5, line:19:5>
5 |   |   |-DeclStmt 0x... <line:9:10, col:17>
6 |   |   |   -VarDecl 0x... <col:10, col:14> col:14 used i 'int' cinit| |-«<NULL»>| |-BinaryOperator 0x... <col:19,
   |   |   |   col:34> 'int' '<'...

```

从中我们可以看到，‘main’ 函数（‘FunctionDecl’）包含一个复合语句（‘CompoundStmt’），复合语句中又有一个 ‘ForStmt’ 节点，清晰地还原了 ‘for’ 循环的结构。

### 1.5.3 语义分析与中间代码

在 AST 的基础上，编译器进行 \*\* 语义分析 \*\*，检查代码的逻辑是否自治。这包括类型检查（例如，不能把一个字符串赋值给整型变量）、变量是否声明后才使用等。

通过所有检查后，编译器将 AST 转换为一种更接近机器指令的 \*\* 中间表示（IR）\*\*。GCC 使用 GIMPLE 和 RTL 作为其主要的 IR。这部分在之前的章节已有详细阐述，它作为优化的主要载体，对 GIMPLE 形式的代码进行常量传播、冗余消除等操作，然后转换为更低级的 RTL，为生成最终的汇编代码做准备。

### 1.5.4 GIMPLE：接近源码的高级表示

编译器首先将预处理后的代码转换成一种名为 \*\*GIMPLE\*\* 的高级中间表示。GIMPLE 的设计目标是既能保留部分源代码的结构（如循环和条件），又足够简单，以便进行各种与目标机器无关的代码优化。它采用的是一种“三地址码”的形式，即每个指令最多只涉及三个操作数。

在不进行任何优化（‘-O0’）的情况下，‘fibonacci.c’ 的 ‘main’ 函数转换的初始 GIMPLE 代码（位于 ‘fibonacci.c.004t.gimple’）如下所示。可以看到，代码结构与原始 C 代码非常相似，变量定义、循环结构和函数调用都清晰可见。

Listing 3: fibonacci.c.004t.gimple (未优化)

```

1 | ;; Function main (main, funcdef_no=0, decl_uid=2095, cgraph_uid=1, symbol_order=0)
2 |
3 | main ()
4 | {
5 |     int next;
6 |     int b;
7 |     int a;
8 |     int i;
9 |     int D.2104;
10 |
11 | <bb 2> [local count: 1073741824]:
12 |     a = 0;
13 |     b = 1;
14 |     printf ("Fibonacci Series up to %d terms:\n", 10);
15 |     i = 0;
16 |     goto <bb 4>; [100.00%]
17 |
18 | <bb 3> [local count: 976367641]:

```



```
19  if (i <= 1)
20      {
21          next = i;
22      }
23  else
24      {
25          next = a + b;
26          a = b;
27          b = next;
28      }
29  printf ("%d ", next);
30  i = i + 1;
31
32  <bb 4> [local count: 1073741824]:
33  if (i <= 9)
34      {
35          goto <bb 3>; [91.00%]
36      }
37  else
38      {
39          goto <bb 5>; [9.00%]
40      }
41
42  <bb 5> [local count: 97374183]:
43  printf ("\n");
44  D.2104 = 0;
45  return D.2104;
46
47 }
```

### 1.5.5 代码优化：从接近高级语言的 GIMPLE 到接近机器语言的 RTL

GIMPLE 的真正威力在于它是绝大多数优化的载体。当开启优化选项（如 ‘-O2’）时，编译器会执行数百个优化过程（Pass），对 GIMPLE 代码进行分析和转换，以期生成更高效的代码。常见的优化手段包括：

- **常量传播**：将常量值直接替换到使用它的地方。
- **冗余消除**：删除重复的计算。
- **循环优化**：如循环展开、循环不变代码外提等。
- **函数内联**：将小函数的调用直接替换为函数体本身。

经过 ‘-O2’ 优化后，虽然初始 GIMPLE 变化不大，但在后续的优化传递中，代码结构会被极大改变，为生成更高效的底层代码铺平道路。

### 1.5.6 RTL：面向机器的低级表示

当 GIMPLE 阶段的优化完成后，编译器会将其转换为一种更低级的、更接近机器指令的中间表示——\*\*RTL (Register Transfer Language)\*\*。RTL 描述了数据如何在寄存器之间传送和计算，它为目标机器相关的优化（如指令选择和寄存器分配）提供了基础。

RTL 代码看起来更像汇编语言的抽象描述。以下是 ‘-O0’（未优化）和 ‘-O2’（优化）下 ‘main’ 函数最终生成的 RTL 代码（位于 ‘\*.final’ 文件）的片段对比。

**未优化的 Final RTL (‘-O0’):** 代码显得冗长，频繁地在栈（‘[rbp-...]’）和寄存器之间移动数据。这是因为 ‘-O0’ 旨在直接翻译代码，而不关心性能。

Listing 4: fibonacci.c.273r.final (未优化片段)

```

1 (insn 15 14 16 2 (set (reg:SI 91 [ i ])
2   (const_int 0 [0])) "fibonacci.c":9:5 -1
3   (nil))
4 (jump_insn 16 15 17 2 (set (pc)
5   (label_ref 29)) "fibonacci.c":9:5 -1
6   (nil)
7   -> 29)
8 (insn 22 21 23 2 (set (reg:SI 88 [ next ])
9   (reg:SI 91 [ i ])) "fibonacci.c":11:13 -1
10  (nil))
11 (jump_insn 23 22 24 2 (set (pc)
12   (label_ref 27)) "fibonacci.c":11:13 -1
13   (nil)
14   -> 27)

```

**优化后的 Final RTL (‘-O2’):** 代码精简了许多。编译器通过优化，将许多变量直接保存在寄存器中，减少了内存访问。例如，循环计数器和斐波那契数列的值可能长时间驻留在寄存器中，大大提高了执行效率。

Listing 5: fibonacci.c.340r.final (优化片段)

```

1 (insn 7 6 8 2 (set (reg:SI 91)
2   (const_int 1 [0x1])) "fibonacci.c":6:20 -1
3   (nil))
4 (insn 8 7 9 2 (set (reg:SI 92)
5   (const_int 0 [0])) "fibonacci.c":6:20 -1
6   (nil))
7 (insn 9 8 10 2 (set (reg:SI 93)
8   (const_int 0 [0])) "fibonacci.c":9:5 -1
9   (nil))
10 (jump_insn 10 9 11 2 (set (pc)
11   (label_ref 21)) "fibonacci.c":9:5 -1
12   (nil)
13   -> 21)

```

最终，这些 RTL 指令会被转换成目标平台的汇编代码，完成整个编译过程。通过观察从 GIMPLE 到 RTL 的演变，以及不同优化级别下的差异，我们可以深刻体会到编译器是如何在忠实于原始逻辑的

基础上，对代码进行精雕细琢，以追求极致性能的。

## 2 第二题：汇编代码撰写

下面我们将‘第二问’中的 LLVM IR 注释版（‘test.ll’）直接展示出来，并逐块解释每个基本块如何对应‘test.c’的设计意图。这有助于把抽象的编译器过程和具体的源码意图连接起来。

Listing 6: test.ll

```

1 ; ModuleID = 'test.c' ; 模块来源：对应源文件 test.c
2 source_filename = "test.c" ; 指明原始源文件名
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"
   ; 目标平台上数据布局描述
4 target triple = "x86_64-pc-linux-gnu" ; 目标三元组：架构-厂商-系统
5
6 @N = dso_local constant i32 5, align 4 ; 全局常量 N = 5（只读）
7 @global_array = dso_local global [5 x i32] [i32 2, i32 2, i32 3, i32 4, i32 5], align
   16 ; 全局数组并初始化
8
9 define dso_local i32 @main() #0 { ; 定义 main 函数，返回类型 i32
10   %1 = alloca i32, align 4 ; 分配栈空间给局部变量（sum, i, sum
      的槽, count, result, remainder）
11   %2 = alloca i32, align 4
12   %3 = alloca i32, align 4
13   %4 = alloca i32, align 4
14   %5 = alloca i32, align 4
15   %6 = alloca i32, align 4
16   store i32 0, ptr %1, align 4 ; sum = 0
17   store i32 0, ptr %2, align 4 ; i = 0
18   store i32 0, ptr %3, align 4
19   store i32 5, ptr %4, align 4 ; count = 5（来自N）
20   br label %7 ; 跳转到标签 %7（循环判断开始）
21
22 7:
23   %8 = load i32, ptr %3, align 4 ; 读取当前索引 i
24   %9 = load i32, ptr %4, align 4 ; 读取 count
25   %10 = icmp slt i32 %8, %9 ; 比较 i < count (signed less-than)
26   br i1 %10, label %11, label %20 ; 若 true 跳到循环体
      %11, 否则转到后续 %20
27
28 11: ; preds = %7 — 循环体基本块
29   %12 = load i32, ptr %2, align 4 ; 读取 sum
30   %13 = load i32, ptr %3, align 4 ; 读取索引 i
31   %14 = sext i32 %13 to i64 ; 将 i (i32) 扩展到 i64, 用于
      getelementptr 索引（GEP 要求匹配指针索引类型）
32   %15 = getelementptr inbounds [5 x i32], ptr @global_array, i64 0, i64 %14 ; 计算
      &global_array[i]
33   %16 = load i32, ptr %15, align 4 ; 读取 global_array[i]

```

```

34  %17 = add nsw i32 %12, %16 ; sum + global_array[i] (nsw
    表示有符号溢出未定义行为)
35  store i32 %17, ptr %2, align 4 ; 将新的 sum 存回
36  %18 = load i32, ptr %3, align 4 ; 读取 i
37  %19 = add nsw i32 %18, 1 ; i = i + 1
38  store i32 %19, ptr %3, align 4 ; 存回 i
39  br label %7, !llvm.loop !6 ;
    回到判断块，并附带循环元数据（提供给优化器）
40
41 20: ; preds = %7 — 循环结束后的入口
42  %21 = load i32, ptr %2, align 4 ; 读取 sum
43  %22 = icmp sgt i32 %21, 20 ; 比较 sum > 20 (signed greater-than)
44  br i1 %22, label %23, label %27 ; 若 true 跳到 then 分支 %23, 否则到
    else %27
45
46 23: ; preds = %20 — then 分支
47  %24 = load i32, ptr %2, align 4 ; 读取 sum, 用作参数
48  %25 = call i32 @calculate(i32 noundef %24, i32 noundef 2) ; 调用 calculate(sum, 2)
49  store i32 %25, ptr %5, align 4 ; 将返回值存入临时 %5
50  %26 = load i32, ptr %5, align 4 ; 读取 result
51  call void @putint(i32 noundef %26) ; 调用外部函数 putint(result)
52  call void @putch(i32 noundef 10) ; 调用 putch(10) 输出换行
53  br label %31 ; 跳到统一后继
54
55 27: ; preds = %20 — else 分支
56  %28 = load i32, ptr %2, align 4 ; 读取 sum
57  %29 = srem i32 %28, 3 ; remainder = sum % 3 (有符号取余)
58  store i32 %29, ptr %6, align 4 ; 存入临时 %6
59  %30 = load i32, ptr %6, align 4
60  call void @putint(i32 noundef %30)
61  call void @putch(i32 noundef 10) ; 输出换行
62  br label %31 ; 跳到统一后继
63
64 31: ; preds = %27, %23 — 后继块
65  ret i32 0 ; main 返回 0, 程序结束
66 }
67
68 ; Function Attrs: noinline nounwind optnone uwtable
69 define dso_local i32 @calculate(i32 noundef %0, i32 noundef %1) #0 { ; 定义
    calculate(a=%0, b=%1)
70  %3 = alloca i32, align 4 ; 为局部变量 a,b,result 分配栈空间
71  %4 = alloca i32, align 4
72  %5 = alloca i32, align 4
73  store i32 %0, ptr %3, align 4
74  store i32 %1, ptr %4, align 4
75  store i32 0, ptr %5, align 4 ; result = 0
76  br label %6 ; 跳转到循环判断块
77
78 6: ; preds = %9, %2 — 循环判断

```

```

79  %7 = load i32, ptr %4, align 4          ; 读取 b 的当前值
80  %8 = icmp sgt i32 %7, 0                ; 检查 b > 0
81  br i1 %8, label %9, label %15          ; 若 true 进入循环体
    %9, 否则跳转到结束 %15
82
83  9:                                     ; preds = %6 — 循环体
84  %10 = load i32, ptr %5, align 4        ; 读取 result
85  %11 = load i32, ptr %3, align 4        ; 读取 a
86  %12 = add nsw i32 %10, %11             ; result += a
87  store i32 %12, ptr %5, align 4        ; 存回 result
88  %13 = load i32, ptr %4, align 4        ; 读取 b
89  %14 = sub nsw i32 %13, 1               ; b = b - 1
90  store i32 %14, ptr %4, align 4        ; 存回 b
91  br label %6, !llvm.loop !8            ; 跳回判断块，包含循环元数据
92
93  15:                                     ; preds = %6 — 循环结束后
94  %16 = load i32, ptr %5, align 4        ; 读取 result
95  ret i32 %16                           ; 返回 result
96  }
97
98  declare void @putint(i32 noundef) #1    ; 外部函数声明: putint(int)
99
100 declare void @putch(i32 noundef) #1     ; 外部函数声明: putch(int)
101
102 attributes #0 = { noinline nounwind 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" }
    ; 属性集合 #0
103 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" }
    ; 属性集合 #1
104
105 !llvm.module.flags = !{!0, !1, !2, !3, !4} ; 模块级元数据标记
106 !llvm.ident = !{!5}                       ; 编译器识别标记
107
108 !0 = !{i32 1, !"wchar_size", i32 4}       ; wchar_t 大小元数据
109 !1 = !{i32 8, !"PIC Level", i32 2}        ; PIC 级别
110 !2 = !{i32 7, !"PIE Level", i32 2}        ; PIE 级别
111 !3 = !{i32 7, !"uwtable", i32 2}          ; unwinding table 元数据
112 !4 = !{i32 7, !"frame-pointer", i32 2}     ; frame-pointer 策略
113 !5 = !{!"Ubuntu clang version 18.1.3 (1ubuntu1)"} ; 生成该 IR 的编译器信息
114 !6 = distinct !{!6, !7}                   ; 循环元数据 (self-referential)
115 !7 = !{!"llvm.loop.mustprogress"}         ; 循环必须前进的标记
116 !8 = distinct !{!8, !7}                   ; 另一个循环元数据

```

## 2.1 llvm\_IR 设计解释

下面将 ‘test.ll’ 中的主要区域按功能模块逐段解释，并给出对应的 ‘test.c’ 源码，以便把 IR 指令与源码设计联系起来。

### 2.1.1 模块头与全局

对应源码：

```
1 const int N = 5;
2 int global_array[5] = {2,2,3,4,5};
```

N 作为配置常量，global\_array 作为程序的静态输入。放在全局便于多个函数直接引用，且在程序加载时就被初始化。

```
1 @N = dso_local constant i32 5, align 4
2 @global_array = dso_local global [5 x i32] [i32 2, ...], align 16
```

说明：全局符号记录了初始值和对齐，编译器只需为数组生成一次存储位置，运行时按地址访问。

### 2.1.2 main 的局部变量与入口 (prologue)

对应源码片段：

```
1 int sum = 0;
2 int i = 0;
3 int count = 5;
```

设计意图：局部变量用于保存计算状态。在未优化（‘-O0’）模式下，为了便于调试，编译器会把变量保留在栈上。

```
1 %1 = alloca i32
2 store i32 0, ptr %1
3 %4 = alloca i32
4 store i32 5, ptr %4
```

说明：‘alloca’ 在函数栈帧为变量分配空间，‘store’ 用于初始化。高级优化会把这些槽去掉，改用寄存器。

### 2.1.3 循环控制流 (while i < count)

对应源码片段：

```
1 while (i < count) {
2     sum = sum + global_array[i];
3     i = i + 1;
4 }
```

设计意图：按索引遍历数组并累加元素。

IR 实现要点：判断块（‘icmp’+‘br’）、循环体（GEP 地址计算 + ‘load’）、索引扩展（‘sext’）、返回跳转 ‘br’。使用 ‘!llvm.loop’ 元数据提示这是循环。

#### 2.1.4 条件分支（if/else）与函数调用

对应源码：

```
1 if (sum > 20) {  
2     int result = calculate(sum, 2);  
3     putint(result); putchar(10);  
4 } else {  
5     int remainder = sum % 3;  
6     putint(remainder); putchar(10);  
7 }
```

说明：条件编译为 ‘icmp’ 与 ‘br’，函数调用由 ‘call’ 实现，返回值通过 ‘store’/‘load’ 读取（‘-O0’ 保守策略）。求余使用 ‘srem’（有符号取余）。

#### 2.1.5 calculate 函数

对应源码：

```
1 int calculate(int a, int b) {  
2     int result = 0;  
3     while (b > 0) {  
4         result = result + a;  
5         b = b - 1;  
6     }  
7     return result;  
8 }
```

设计意图：用循环实现重复加法并返回结果。IR 保持函数边界，用 ‘alloca’ 存参数与局部变量，循环逻辑与 main 类似。

#### 2.1.6 外部函数声明与元数据

‘putint’ 与 ‘putch’ 在 IR 中用 ‘declare’ 声明，表示定义在模块外，链接时由运行时库提供。模块元数据（‘!llvm.ident’，‘!llvm.module.flags’ 等）记录编译器信息与优化提示。

## 参考文献

- [1] Gene H Golub and James M Ortega. *Scientific computing: an introduction with parallel computing*. Elsevier, 2014.
- [2] Stephen Bassi Joseph, Emmanuel Gbenga Dada, Sanjay Misra, and Samuel Ajoka. Parallel faces recognition attendance system with anti-spoofing using convolutional neural network. In *Illumination of Artificial Intelligence in Cybersecurity and Forensics*, pages 123–137. Springer, 2022.
- [3] Michael J Quinn. *Parallel computing theory and practice*. McGraw-Hill, Inc., 1994.