



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：王泽舜 杨乔钦
学号：2310655 2311838
专业：计算机科学与技术

2025 年 9 月 29 日

目录

1 第一题：编译过程分析	2
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 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
2 第二题：汇编代码撰写	10
2.1 原始 SysY 程序	10
2.2 arm 汇编程序	11
2.2.1 说明	13
2.3 llvm IR 程序	14
2.4 结果展示	16
3 参考指令	16

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

```

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 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

第二题中，王泽舜负责编写与源程序等价的 llvm IR 代码，杨乔钦负责编写与源程序等价的 riscv 汇编代码。

2.1 原始 SysY 程序

我们编写了一个简单的 SysY 程序，功能是读取一个整数数组，计算其元素和，并根据和的大小进行不同的输出操作如果大于 20，会输出和的两倍，如果小于等于 20，会输出和除以 3 的余数。程序中包含了变量声明、数组操作、算术运算、条件判断和函数调用等基本语法结构。以下是该 SysY 程序的完整代码：

逐列访问平凡算法

```
1 // 全局变量和常量声明
2 const int N = 5;
3 int global_array[5] = {2,2,3,4,5};
4
5 // SysY运行时库函数声明 — 必须添加这些声明才能调用库函数
6 void putint(int); // 输出一个整数
7 void putch(int); // 输出一个字符
8
9 // 函数声明
10 int calculate(int a, int b);
11
12 int main() {
13
14     int sum = 0;
15     int i = 0;
16
17     // 使用getarray从输入获取数据，替换原来的固定数组赋值
18     int count = 5;
19
20
21     // 算术运算：计算数组元素的和
22     while (i < count) { // 使用实际读取的元素个数
23         sum = sum + global_array[i];
24         i = i + 1;
25     }
26
27     // 条件判断和输出
28     if (sum > 20) {
29         int result = calculate(sum, 2);
30         putint(result); // 使用putint输出结果而不是直接return
31         putch(10); // 输出换行符
32     } else {
```

```

33     int remainder = sum % 3;
34     putint(remainder); // 使用 putint 输出结果
35     putchar(10);      // 输出换行符
36 }
37
38 return 0; // main 函数返回 0 表示正常结束
39 }
40
41 // 函数定义：复杂计算
42 int calculate(int a, int b) {
43     int result = 0;
44     while (b > 0) {
45         result = result + a;
46         b = b - 1;
47     }
48     return result;
49 }

```

2.2 arm 汇编程序

我编写的 arm 汇编程序如下所示，该程序实现了与上述 SysY 程序相同的功能。它读取一个整数数组，计算其元素和，并根据和的大小进行不同的输出操作。如果和大于 20，则输出和的两倍；如果和小于等于 20，则输出和除以 3 的余数。程序中包含了变量声明、数组操作、算术运算、条件判断和函数调用等基本语法结构。以下是该 arm 汇编程序的完整代码：

逐列访问平凡算法

```

1  .arch armv8-a ; 指定目标架构为 AArch64
2  .file "test.c" ; 源文件名信息
3  .text ; 文本段开始
4  .section .text.startup,"ax",@progbits ; 启动代码段
5  .align 2 ; 对齐
6  .p2align 4,,11 ; 更高对齐
7  .global main ; 导出 main 符号
8  .type main,%function ; 声明 main 为函数
9  main: ; main 函数入口
10 .LFB0: ; 函数框架开始标签
11 .cfi_startproc ; 调试信息开始
12 adrp x0, .LANCHOR0 ; 将 LAUNCHER 基址高位加载到 x0
13 add x1, x0, :lo12:LANCHOR0 ; 将基址低位加到 x0 得到完整地址放 x1
14 stp x29, x30, [sp, -16]! ; 保存帧指针和返回地址并调整栈 (push)
15 .cfi_def_cfa_offset 16 ; 调试: 定义 CFA 偏移
16 .cfi_offset 29, -16 ; 调试: x29 保存位置
17 .cfi_offset 30, -8 ; 调试: x30 保存位置
18 mov x29, sp ; 设置帧指针 x29 = sp
19 ldp w0, w4, [x1] ; 从内存载入两个 32-bit 值到 w0, w4
20 ldp w3, w2, [x1, 8] ; 继续加载另外两个 32-bit 值到 w3, w2
21 add w0, w0, w4 ; 累加 w0 和 w4

```

```

22  ldr w1, [x1, 16] ; 载入第五个 32-bit 值到 w1
23  add w0, w0, w3 ; 累加 w3
24  add w0, w0, w2 ; 累加 w2
25  add w0, w0, w1 ; 累加 w1, w0 保存元素和
26  cmp w0, 20 ; 比较和与 20
27  ble .L2 ; 若小于等于 20 跳到 .L2
28  lsl w0, w0, 1 ; 否则将和乘 2 (左移一位)
29  bl  putint ; 调用 putint 输出整数
30  mov w0, 10 ; 将换行符 ASCII 10 放入 w0
31  bl  putchar ; 调用 putchar 输出换行
32  .L3: ; 公共返回标签
33  mov w0, 0 ; 返回值置 0
34  ldp x29, x30, [sp], 16 ; 恢复 x29,x30 并释放栈空间
35  .cfi_remember_state ; 调试信息: 记住状态
36  .cfi_restore 30 ; 恢复调试寄存器 30 信息
37  .cfi_restore 29 ; 恢复调试寄存器 29 信息
38  .cfi_def_cfa_offset 0 ; 恢复 CFA 偏移
39  ret ; 返回调用者
40  .L2: ; 小于等于20的处理分支
41  .cfi_restore_state ; 恢复调试状态
42  mov w1, 3 ; 将除数 3 放入 w1
43  sdiv w1, w0, w1 ; w1 = w0 / 3 (商)
44  add w1, w1, w1, lsl 1 ; w1 = w1 + (w1 << 1) = 3 * 商
45  sub w0, w0, w1 ; 计算余数 w0 = w0 - 3*商
46  bl  putint ; 输出余数
47  mov w0, 10 ; 设置换行字符
48  bl  putchar ; 输出换行
49  b .L3 ; 跳转到公共返回
50  .cfi_endproc ; 结束调试信息
51  .LFE0: ; 函数结束标签
52  .size main, .-main ; 记录 main 大小
53  .text ; 文本段 (下一个函数)
54  .align 2 ; 对齐
55  .p2align 4,,11 ; 对齐
56  .global calculate ; 导出 calculate 符号
57  .type calculate, %function ; 声明 calculate
58  calculate: ; calculate 函数入口
59  .LFB1: ; 函数框架开始
60  .cfi_startproc ; 调试开始
61  mul w0, w1, w0 ; w0 = w0 * w1 (乘法实现)
62  cmp w1, 0 ; 比较参数 w1 与 0
63  csel w0, w0, wzr, gt ; 若 w1>0 返回乘积, 否则返回 0
64  ret ; 返回
65  .cfi_endproc ; 结束调试信息
66  .LFE1: ; 结束标签
67  .size calculate, .-calculate ; 记录大小
68  .global global_array ; 导出全局数组
69  .global N ; 导出常量 N
70  .section .rodata ; 只读数据段

```

```

71 .align 2 ; 对齐
72 .type N, %object ; 声明 N 为对象
73 .size N, 4 ; N 的大小
74 N: ; 常量 N 标签
75 .word 5 ; N = 5
76 .data ; 数据段开始
77 .align 4 ; 数据对齐
78 .set .LANCHOR0, . + 0 ; 设置锚点符号
79 .type global_array, %object ; 声明全局数组
80 .size global_array, 20 ; 数组大小 20 字节
81 global_array: ; 全局数组标签
82 .word 2 ; 元素 0 = 2
83 .word 2 ; 元素 1 = 2
84 .word 3 ; 元素 2 = 3
85 .word 4 ; 元素 3 = 4
86 .word 5 ; 元素 4 = 5
87 .ident "GCC: (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0" ; 编译器标识
88 .section .note.GNU-stack,"",@progbits ; 指示不可执行栈

```

2.2.1 说明

- 调用约定: AArch64 使用寄存器 x0..x7 传递前八个整型/指针参数, 返回值放在 x0 (32-bit 时用 w0)。被调用者保存寄存器 (如 x19..x28) 需在函数入口保存并在返回前恢复。
- 栈帧与栈保护: 函数入口处通过 `sub sp, sp, #64` 分配栈空间, 编译器加入了栈保护 (`_stack_chk_guard` / `_stack_chk_fail`) 以检测栈溢出攻击。
- 数组寻址: 编译器使用位移指令 (例如 `lsl 2`) 将索引乘以 4 来计算整数元素的字节偏移, 从而高效访问数组元素。
- 循环与分支优化: 简单的循环被编译为比较/分支形式; 小的数学变换 (例如将循环展开或用乘法替代循环) 可能由编译器进行优化, 如本例中 `calculate` 使用乘法和条件选择实现语义。
- 运行时库依赖: 汇编中调用的 `putint`、`putch` 等函数由运行时库 (例如 `libsysy_aarch.a`) 提供。链接时必须指定对应目标架构的静态库, 否则会出现未定义引用错误。
- 链接建议: 使用交叉链接器将汇编/目标文件与 AArch64 静态库链接, 例如:

```
aarch64-linux-gnu-gcc test_aarch64.s libsysy_aarch.a -o test_arm.out
```

若库放在子目录则传递相对路径: `lib/.../libsysy_aarch.a`。

- 架构匹配: 务必确保静态库是为目标架构 (AArch64) 构建的; x86 构建的库无法用于 ARM 链接。
- 调试与反汇编: 若需查看生成的可执行的真实指令或符号信息, 可用 `objdump -d / readelf / nm` 等工具检查符号表与节信息。

2.3 llvm IR 程序

王泽舜编写的 llvm IR 程序如下所示，该程序实现了与上述 SysY 程序相同的功能。以下是该 llvm IR 程序的完整代码，说明见注释：

Listing 6: test_new.ll

```

1 ; 目标平台信息，消除编译警告
2 target triple = "x86_64-pc-linux-gnu"
3 ; 全局常量和数组
4 @N = constant i32 5
5 @global_array = global [5 x i32] [i32 2, i32 2, i32 3, i32 4, i32 5]
6 ; 外部函数声明
7 declare void @putint(i32)
8 declare void @putch(i32)
9 ; main 函数
10 define i32 @main() {
11 entry:
12     ; 局部变量初始化
13     %sum = alloca i32
14     %i = alloca i32
15     %count = alloca i32
16
17     store i32 0, ptr %sum      ; sum = 0
18     store i32 0, ptr %i        ; i = 0
19     store i32 5, ptr %count    ; count = 5
20
21     br label %while_check
22
23 while_check:
24     ; 检查循环条件: i < count
25     %i_val = load i32, ptr %i
26     %count_val = load i32, ptr %count
27     %cmp = icmp slt i32 %i_val, %count_val
28     br i1 %cmp, label %while_body, label %while_end
29
30 while_body:
31     ; 循环体: sum = sum + global_array[i]
32     %sum_val = load i32, ptr %sum
33     %i_curr = load i32, ptr %i
34
35     ; 计算 global_array[i] 的地址
36     %array_ptr = getelementptr [5 x i32], ptr @global_array, i32 0, i32 %i_curr
37     %array_val = load i32, ptr %array_ptr
38
39     ; sum = sum + global_array[i]
40     %new_sum = add i32 %sum_val, %array_val
41     store i32 %new_sum, ptr %sum
42
43     ; i = i + 1

```

```

44     %new_i = add i32 %i_curr, 1
45     store i32 %new_i, ptr %i
46
47     br label %while_check
48
49 while_end:
50     ; if (sum > 20)
51     %final_sum = load i32, ptr %sum
52     %if_cmp = icmp sgt i32 %final_sum, 20
53     br i1 %if_cmp, label %then_branch, label %else_branch
54
55 then_branch:
56     ; int result = calculate(sum, 2);
57     %result = call i32 @calculate(i32 %final_sum, i32 2)
58
59     ; putint(result); putchar(10);
60     call void @putint(i32 %result)
61     call void @putchar(i32 10)
62     br label %exit
63
64 else_branch:
65     ; int remainder = sum % 3;
66     %remainder = srem i32 %final_sum, 3
67
68     ; putint(remainder); putchar(10);
69     call void @putint(i32 %remainder)
70     call void @putchar(i32 10)
71     br label %exit
72
73 exit:
74     ret i32 0
75 }
76
77 ; calculate 函数
78 define i32 @calculate(i32 %a, i32 %b) {
79 entry:
80     ; 局部变量
81     %result = alloca i32
82     %b_local = alloca i32
83
84     store i32 0, ptr %result      ; result = 0
85     store i32 %b, ptr %b_local    ; 保存参数 b
86
87     br label %loop_check
88
89 loop_check:
90     ; 检查循环条件: b > 0
91     %b_val = load i32, ptr %b_local
92     %loop_cmp = icmp sgt i32 %b_val, 0

```



```

93     br i1 %loop_cmp, label %loop_body, label %loop_end
94
95 loop_body:
96     ; result = result + a
97     %result_val = load i32, ptr %result
98     %new_result = add i32 %result_val, %a
99     store i32 %new_result, ptr %result
100
101     ; b = b - 1
102     %b_curr = load i32, ptr %b_local
103     %new_b = sub i32 %b_curr, 1
104     store i32 %new_b, ptr %b_local
105
106     br label %loop_check
107
108 loop_end:
109     %final_result = load i32, ptr %result
110     ret i32 %final_result
111 }

```

2.4 结果展示

我们将上述 SysY 程序编译为 ARM 汇编，并链接运行时库，成功生成了可执行文件。以下是 SysY 程序和 arm 汇编的编译成功控制台信息：

运行成功信息

```

1 yqq@LAPTOP-B4JFEDK6:~/repository/Copiler/1.编译器了解$ aarch64-linux-gnu-gcc
   test_aarch64.s lib.tar.gz/libsysy_aarch.a -o test_arm.out
2 yqq@LAPTOP-B4JFEDK6:~/repository/Copiler/1.编译器了解$ qemu-aarch64 -L
   /usr/aarch64-linux-gnu ./test_arm.out
3 1
4 TOTAL: 0H-0M-0S-0us

```

以下是 llvm 程序编译成功和运行的控制台信息：

运行成功信息

```

1 enovo@wzs:~/lab1$ vim test3.ll
2 enovo@wzs:~/lab1$ clang test3.ll libsysy_x86.a -o my_ir_program3 -static
3 enovo@wzs:~/lab1$ ./my_ir_program3
4 1
5 TOTAL: 0H-0M-0S-0us
6 enovo@wzs:~/lab1$

```

由此可见，SysY 程序和 llvm，arm 汇编程序在相同输入下均能正确输出预期结果，功能实现一致。

3 参考指令

oprule array 内容	SysY 编译输出	arm 汇编编译输出	llvm IR 编译输出
1,2,3,4,5	0	0	0
2,2,3,4,5	1	1	1
3,2,3,4,5	2	2	2
4,2,3,4,5	0	0	0

表 1: 运行结果比较

Listing 7: 汇编指令

```

1  gcc -E fibonacci.c -o fibonacci.i
2  gcc -S fibonacci.i -o fibonacci.s
3  gcc -c fibonacci.s -o fibonacci.o
4  objdump -d fibonacci.o
5  gcc fibonacci.o -o fibonacci
6  objdump -d fibonacci > fibonacci.exe.asm
7  ./fibonacci
8  生成 token 流
9  clang -Xclang -dump-tokens fibonacci.c
10 # 生成 AST
11 clang -Xclang -ast-dump -fsyntax-only fibonacci.c > ast_dump.txt
12 #生成符号表
13 gcc -fdump-translation-unit fibonacci.c
14 gcc -fdump-tree-all -O0 -c fibonacci.c -o fib_O0.o # 无优化
15 gcc -fdump-tree-all -O2 -c fibonacci.c -o fib_O2.o
16 gcc -fdump-rtl-all fibonacci.c -O2 -o /dev/null
17 # 编译llvm代码
18 clang my_ir.ll libsysy_x86.a -o my_ir_program -static

```