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 27 日

目录

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）：

```c
#define MAX_ITERATIONS 10

int main() {
    int a = 0, b = 1, next, i;

    printf("Fibonacci Series up to %d terms:\n", MAX_ITERATIONS);

    for (i = 0; i < MAX_ITERATIONS; i++) {
        if (i <= 1) {
            next = i;
        } else {
            next = a + b;
            a = b;
            b = next;
        }
        printf("%d ", next);
    }
    printf("\n");
    return 0;
}
```

之后（fibonacci.i 的 main 函数部分）：

```c
int main() {
    int a = 0, b = 1, next, i;

    printf("Fibonacci Series up to %d terms:\n", 10);

    for (i = 0; i < 10; i++) {
        if (i <= 1) {
            next = i;
        } else {
            next = a + b;
            a = b;
```

```
            b = next;
        }
        printf("%d ", next);
    }
    printf("\n");
    return 0;
}
```

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）：

```
int main() {
    int a = 0, b = 1, next, i;

    printf("Fibonacci Series up to %d terms:\n", 10);

    for (i = 0; i < 10; i++) {
        if (i <= 1) {
            next = i;
        } else {
            next = a + b;
            a = b;
            b = next;
        }
        printf("%d ", next);
    }
    printf("\n");
    return 0;
}
```

之后（fibonacci.s 的 main 函数部分）：

```
    .globl   main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
    .cfi_def_cfa_register 6
    subq  $16, %rsp
    movl  $0, -16(%rbp)
    movl  $1, -12(%rbp)
    movl  $10, %esi
    leaq  .LC0(%rip), %rax
    movq  %rax, %rdi
    movl  $0, %eax
    call  printf@PLT
    movl  $0, -4(%rbp)
    jmp .L2
.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）：

```
    .globl   main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
    .cfi_def_cfa_register 6
    subq  $16, %rsp
```

```
  movl   $0, -16(%rbp)
  movl   $1, -12(%rbp)
  movl   $10, %esi
  leaq   .LC0(%rip), %rax
  movq   %rax, %rdi
  movl   $0, %eax
  call   printf@PLT
```

之后（fibonacci.o.asm 的 main 函数部分）：

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

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

```
0000000000001169 <main>:
    1169: f3 0f 1e fa           endbr64
    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)

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

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

```
-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
...
|-ForStmt 0x... <line:9:5, line:19:5>
| |-DeclStmt 0x... <line:9:10, col:17>
| | -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洽。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

```
;; Function main (main, funcdef_no=0, decl_uid=2095, cgraph_uid=1, symbol_order=0)

main ()
{
  int next;
  int b;
  int a;
  int i;
  int D.2104;

  <bb 2> [local count: 1073741824]:
  a = 0;
  b = 1;
  printf ("Fibonacci Series up to %d terms:\n", 10);
  i = 0;
  goto <bb 4>; [100.00%]

  <bb 3> [local count: 976367641]:
```

```
  if (i <= 1)
    {
      next = i;
    }
  else
    {
      next = a + b;
      a = b;
      b = next;
    }
  printf ("%d ", next);
  i = i + 1;

 <bb 4> [local count: 1073741824]:
  if (i <= 9)
    {
      goto <bb 3>; [91.00%]
    }
  else
    {
      goto <bb 5>; [9.00%]
    }

 <bb 5> [local count: 97374183]:
  printf ("\n");
  D.2104 = 0;
  return D.2104;

}
```

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)

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

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

Listing 6: test_new.ll

```
; 目标平台信息，消除编译警告
target triple = "x86_64-pc-linux-gnu"
; 全局常量和数组
@N = constant i32 5
@global_array = global [5 x i32] [i32 2, i32 2, i32 3, i32 4, i32 5]
; 外部函数声明
declare void @putint(i32)
declare void @putch(i32)
; main 函数
define i32 @main() {
entry:
    ; 局部变量初始化
    %sum = alloca i32
    %i = alloca i32
    %count = alloca i32

    store i32 0, ptr %sum          ; sum = 0
    store i32 0, ptr %i            ; i = 0
    store i32 5, ptr %count        ; count = 5

    br label %while_check

while_check:
    ; 检查循环条件: i < count
    %i_val = load i32, ptr %i
    %count_val = load i32, ptr %count
    %cmp = icmp slt i32 %i_val, %count_val
    br i1 %cmp, label %while_body, label %while_end

while_body:
    ; 循环体: sum = sum + global_array[i]
    %sum_val = load i32, ptr %sum
    %i_curr = load i32, ptr %i

    ; 计算 global_array[i] 的地址
    %array_ptr = getelementptr [5 x i32], ptr @global_array, i32 0, i32 %i_curr
    %array_val = load i32, ptr %array_ptr

    ; sum = sum + global_array[i]
    %new_sum = add i32 %sum_val, %array_val
```

```
    store i32 %new_sum, ptr %sum

    ; i = i + 1
    %new_i = add i32 %i_curr, 1
    store i32 %new_i, ptr %i

    br label %while_check

while_end:
    ; if (sum > 20)
    %final_sum = load i32, ptr %sum
    %if_cmp = icmp sgt i32 %final_sum, 20
    br i1 %if_cmp, label %then_branch, label %else_branch

then_branch:
    ; int result = calculate(sum, 2);
    %result = call i32 @calculate(i32 %final_sum, i32 2)

    ; putint(result); putch(10);
    call void @putint(i32 %result)
    call void @putch(i32 10)
    br label %exit

else_branch:
    ; int remainder = sum % 3;
    %remainder = srem i32 %final_sum, 3

    ; putint(remainder); putch(10);
    call void @putint(i32 %remainder)
    call void @putch(i32 10)
    br label %exit

exit:
    ret i32 0
}

; calculate 函数
define i32 @calculate(i32 %a, i32 %b) {
entry:
    ; 局部变量
    %result = alloca i32
    %b_local = alloca i32

    store i32 0, ptr %result      ; result = 0
    store i32 %b, ptr %b_local    ; 保存参数 b

    br label %loop_check

loop_check:
```

```
    ; 检查循环条件: b > 0
    %b_val = load i32, ptr %b_local
    %loop_cmp = icmp sgt i32 %b_val, 0
    br i1 %loop_cmp, label %loop_body, label %loop_end

loop_body:
    ; result = result + a
    %result_val = load i32, ptr %result
    %new_result = add i32 %result_val, %a
    store i32 %new_result, ptr %result

    ; b = b - 1
    %b_curr = load i32, ptr %b_local
    %new_b = sub i32 %b_curr, 1
    store i32 %new_b, ptr %b_local

    br label %loop_check

loop_end:
    %final_result = load i32, ptr %result
    ret i32 %final_result
}
```