



Datawhale 开源社区

DATAWHALE OPEN SOURCE COMMUNITY

# 深入理解计算机系统 (6)

Computer Systems A Programmer's Perspective

CSAPP

李岳昆、易远哲

realgurk@gmail.com、yuanzhe.yi@outlook.com

2021 年 9 月 25 日



第 I 部分

# 程序的机器级表示-II

算术和逻辑操作

控制



- leaq S, D  $\rightarrow$  Load Effective Address

- ```
leaq 7(%rdx,%rdx,4),%rax
```

- $$Imm(r_b, r_i, s) \rightarrow Imm + R[r_b] + R[r_i] \cdot s$$

除了加载有效地址的功能，leaq 指令还可以用来表示加法和有限的乘法运算：

```
1 long scale(long x, long y, long z){
2     long t = x + 4 * y + 12 * z;
3     return t;
4 }
```

经过编译后，这段代码是通过三条 `leaq` 指令来实现。

scale:

```
leaq (%rdi, %rsi, 4), %rax
leaq (%rdx, %rdx, 2), %rdx
leaq (%rax, %rdx, 4), %rax
ret
```

根据寄存器的使用惯例，参数 x, y, z 分别保存在寄存器 rdi、rsi 以及 rdx 中，还是根据内存引用的计算公式，第一条指令的源操作数就对应于  $x+4*y$ ，具体过程如图所示。

scale:

```
leaq (%rdi, %rsi, 4), %rax → %rdi + 4*%rsi = x + 4*y
leaq (%rdx, %rdx, 2), %rdx → %rdx + 2*%rdx = z + 2*z
leaq (%rax, %rdx, 4), %rax → %rax + 4*%rdx
ret                          = (x + 4*y)
```

## 算术和逻辑操作

## 控制

- 1 首先计算  $3 \times z$  的数值。第二条的 `leaq` 指令执行完毕，此时寄存器 `rdx` 中保存的值是  $3z$ 。
- 2 把  $3z$  作为一个整体乘以 4。

scale:

```
leaq (%rdi, %rsi, 4), %rax → %rdi + 4*%rsi = x + 4*y
leaq (%rdx, %rdx, 2), %rdx → %rdx + 2*%rdx = z + 2*z
leaq (%rax, %rdx, 4), %rax → %rax + 4*%rdx
ret                          = (x + 4*y)
```

$$\text{leaq } (\%rax, \%rdx, 12), \%rax \rightarrow \%rax + 12 * \%rdx = (x + 4 * y) + 12 * z$$

这里主要是由于比例因子取值只能是 1,2,4,8 这四个数中的一个, 因此要把 12 进行分解。



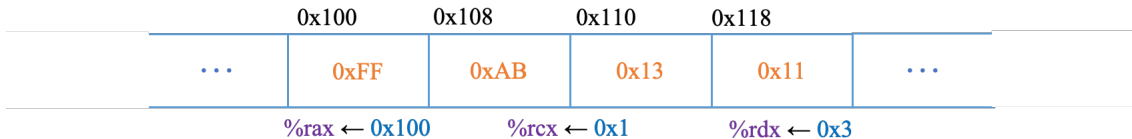
一元操作指令只有一个操作数，因此该操作数既是源操作数也是目的操作数，操作数可以是寄存器，也可以是内存地址。

| 指令           | 影响                    | 描述  |
|--------------|-----------------------|-----|
| <i>INC D</i> | $D \leftarrow D + 1$  | 加 1 |
| <i>DEC D</i> | $D \leftarrow D - 1$  | 减 1 |
| <i>NEG D</i> | $D \leftarrow -D$     | 取负  |
| <i>NOT D</i> | $D \leftarrow \sim D$ | 取补  |

二元操作指令包含两个操作数，第一个操作数是源操作数，这个操作数可以是立即数、寄存器或者内存地址；第二个操作数既是源操作数也是目的操作数，这个操作数可以是寄存器或者内存地址，但不能是立即数。

| 指令               | 影响                        | 描述 |
|------------------|---------------------------|----|
| <i>ADD S, D</i>  | $D \leftarrow D + S$      | 加  |
| <i>SUB S, D</i>  | $D \leftarrow D - S$      | 减  |
| <i>IMUL S, D</i> | $D \leftarrow D * S$      | 乘  |
| <i>XOR S, D</i>  | $D \leftarrow D \wedge S$ | 异或 |
| <i>OR S, D</i>   | $D \leftarrow D   S$      | 或  |
| <i>AND S, D</i>  | $D \leftarrow D \& S$     | 与  |

下面看一个例子，一开始，内存以及寄存器中所保存的数据如图所示。

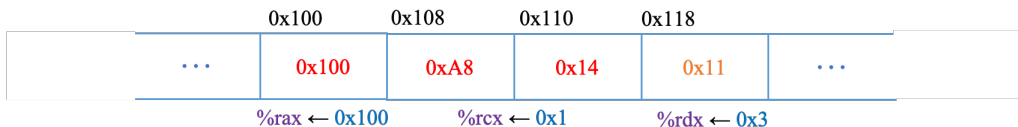


```
addq %rcx, (%rax) → Mem[0x100] = Mem[0x100] + R[%rcx]
subq %rdx, 8(%rax) → Mem[0x108] = Mem[0x108] - R[%rdx]
incq 16(%rax), → Mem[0x110] = Mem[0x110] + 1
subq %rdx, %rax → R[%rax] = R[%rax] - R[%rdx]
```

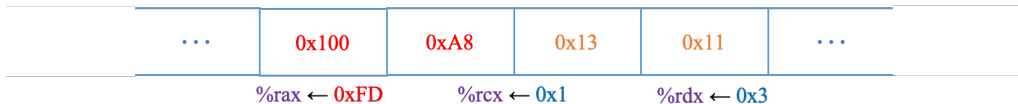
- ② 减法指令 `subq` 是将内存地址 `0x108` 内的数据减去寄存器 `rdx` 内的数据，二者之差在存储到内存地址 `0x108` 处，该指令执行完毕后，内存地址 `0x108` 处所存储的数据由 `0xAB` 变成 `0xA8`。



③ 对于加一指令 incq，就是将内存地址 0x110 内存存储的数据加 1，结果是内存地址 0x110 处所存储的数据由 0x13 变成 0x14。



④ 最后一条加法指令是将寄存器 `rax` 内的值减去寄存器 `rdx` 内的值，最终寄存器 `rax` 的值由 `0x100` 变成 `0xFD`。



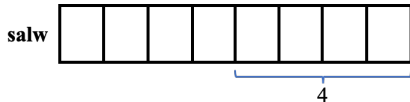
左移指令有两个，分别是 SAL 和 SHL，二者的效果是一样的，都是在右边填零；右移指令不同，分为算术右移和逻辑右移，算术右移需要填符号位，逻辑右移需要填零，这与 C 语言中所讲述的移位操作是一致的。

| 指令          | 影响                       | 描述         |
|-------------|--------------------------|------------|
| $SAL\ k, D$ | $D \leftarrow D \ll k$   | 左移         |
| $SHL\ k, D$ | $D \leftarrow D \ll k$   | 左移，等同于 SAL |
| $SAR\ k, D$ | $D \leftarrow D \gg_A k$ | 算术右移       |
| $SHR\ k, D$ | $D \leftarrow D \gg_L k$ | 逻辑右移       |

- 

- salb
- |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
- 3

- 对于指令 `salw`，移位量则是由寄存器 `cl` 的低 4 位来决定。



以此类推，双字对应的是低 5 位，四字对应的是低 6 位。



接下来，我们通过一个例子来讲述一下移位指令的用途，下面的代码涉及了多种操作。

## 示例代码

```
1 long arith(long x, long y, long z){
2     long t1 = x ^ y;
3     long t2 = z * 48;
4     long t3 = t1 & 0xF0F0F0F;
5     long t4 = t2 - t3;
6     return t4;
7 }
```

我们重点看一下 `z*48` 这行代码所对应的汇编指令。

```
long t2 = z * 48;
```

```
leaq (%rdx, %rdx, 2), %rax
salq $4, %rax
```

这个计算过程被分解成了两步：

- ① 第一步，首先计算  $3 \times z$ ，指令 `leaq` 来实现，计算结果保存到寄存器 `rax`。

```
leaq (%rdx, %rdx, 2), %rax
R[%rdx] + R[%rdx] * 2 = 3 * z → %rax
```

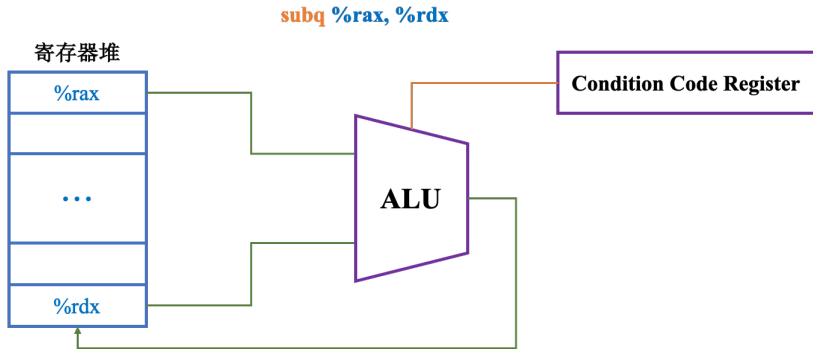
- ```
salq $4, %rax
24 * R[%rax] = 16 * (3 * z) = 48 * z
```

为什么编译器不直接使用乘法指令来实现这个运算呢？主要是因为乘法指令的执行需要更长的时间，因此编译器在生成汇编指令时，会优先考虑更高效的方式。

此外，还有一些特殊的算术指令，对于汇编指令学习，最关键的是了解指令相关的概念，并不需要去记指令的细枝末节，学会查阅指令手册，能够找到需要的信息即可。

指令	描述
imulq $S$	有符号全乘法
mulq $S$	无符号全乘法
cqto	转换为八字
idivq $S$	有符号除法
divq $S$	无符号除法

ALU 除了执行算术和逻辑运算指令外，还会根据该运算的结果去设置条件码寄存器。



接下来，我们详细介绍一下条件码寄存器的相关知识。

条件码寄存器它是由 CPU 来维护的，长度是单个比特位，它描述了最近执行操作的属性。



假如 ALU 执行两条连续的算术指令。

```
t1: addq    %rax, %rbx
t2: subq    %rcx, %rdx
```

t1 和 t2 表示时刻，t1 时刻条件码寄存器中保存的是指令 1 的执行结果的属性，t2 时刻，条件码寄存器的内容被下一条指令所覆盖。

CF	ZF	...	SF	OF
----	----	-----	----	----

**CF** —— Carry Flag（进位标志）  
**ZF** —— Zero Flag（零标志）  
**SF** —— Sign Flag（符号标志）  
**OF** —— Overflow Flag（溢出标志）

- 进位标志，当 CPU 最近执行的一条指令最高位产生了进位时，进位标志 (CF) 会被置为 1，它可以用来检查无符号数操作的溢出。
- 零标志，当最近操作的结果等于零时，零标志 (ZF) 会被置 1。
- 符号标志，当最近的操作结果小于零时，符号标志 (SF) 会被置 1
- 溢出标志，针对有符号数，最近的操作导致正溢出或者负溢出时溢出标志 (OF) 会被置 1。

条件码寄存器的值是由 ALU 在执行算术和运算指令时写入的，下图中的这些算术和逻辑运算指令都会改变条件码寄存器的内容。

Unary Operations	Binary Operations	Shift Operations
<i>INC D</i>	<i>ADD S, D</i>	<i>SAL k, D</i>
<i>DEC D</i>	<i>SUB S, D</i>	<i>SHL k, D</i>
<i>NEG D</i>	<i>IMUL S, D</i>	<i>SAR k, D</i>
	<i>OR S, D</i>	<i>SHR k, D</i>
	<i>XOR S, D</i>	
	<i>AND S, D</i>	



除此之外，还有两类指令可以设置条件码寄存器：`cmp` 指令和 `test` 指令。

- 算术和逻辑操作      控制

```
1 int comp(long a, long b){
2     return (a == b);
3 }
```

```
comq    %rsi, %rdi
sete    %al
movzbl  %al, %eax
ret
```

## 算术和逻辑操作

根据寄存器使用的惯例，参数 a 存放在寄存器 rdi 中，参数 b 存放在寄存器 rsi 中。



```
int comp(long a, long b){
    return (a==b);
}
```

comp:

a in %ordi, b in %rsi

**comq**      %rsi, %rdi

sete %a1

```
movzbl    %al, %eax
```

ret

```
1 int comp(long a, long b){
2     return (a == b);
3 }
```

```
comq    %rsi, %rdi
sete    %al
movzbl  %al, %eax
ret
```

## 算术和逻辑操作

接下来的这条指令 `sete` 看起来就有点费解了，这是因为通常情况下，并不会直接去读条件码寄存器。其中一种方式是根据条件码的某种组合，通过 `set` 类指令，将一个字节设置为 0 或者 1。在这个例子中，指令 `sete` 根据零标志 (ZF) 的值对寄存器 `al` 进行赋值，后缀 `e` 是 `equal` 的缩写。如果零标志等于 1，指令 `sete` 将寄存器 `al` 置为 1；如果零标志等于 0，指令 `sete` 将寄存器 `al` 置为 0。

a in %ordi, b in %rsi

**comq**      %rsi, %rdi  $\rightarrow$  a-b

sete %a1

```

if ZF=1, %al=1
if ZF=0, %al=0

```

```
movzbl    %al, %eax
```

ret

然后 mov 指令对寄存器 al 进行零扩展，最后返回判断结果。

下面看一个复杂的例子。

## 另一个例子

```
1 int comp(char a, char b){  
2     return (a < b);  
3 }
```

转成汇编指令如下。

comp:

comb    %sil, %dil

sete    %al

movzbl %al, %eax

ret

相对于相等的情况，判断小于的情况要稍微复杂一点。需要根据符号标志 (SF) 和溢出标志 (OF) 的异或结果来判定。

两个有符号数相减，当没有发生溢出时，如果  $a$  小于  $b$ ，结果为负数，那么符号标志 (SF) 被置为 1；如果  $a > b$ ，结果为正数，那么符号标志 (SF) 就不会被置 1。

$$t = a - b$$

Case 2 :  $a > b$     $t > 0$     $SF = 0$     $SF \wedge OF = 0$



Case 3 :  $a < b$        $t < 0$        $a = -2$        $b = 127$   
 $t = 127 > 0$ ,       $SF = 0$     $OF = 1$        $SF \wedge OF = 1$

当  $a=1$ ,  $b=-128$ , 由于发生了正溢出, 结果  $t=-127$ , 虽然  $a>b$ , 但是由于溢出导致了结果  $t$  小于 0, 此时符号标志 (SF) 和溢出标志 (OF) 都会被置为 1。

Case 4 :  $a > b$        $a=1$        $b=-128$   
 $t=-127 < 0$ ,       $SF=1$   $OF=1$        $SF \wedge OF=0$

对于其他判断情况，都可以通过条件码的组合来实现。

指令	影响	描述
setg $D$	$D \leftarrow \sim (SF \wedge OF) \& \sim ZF$	Greater(signed $>$ )
setge $D$	$D \leftarrow \sim (SF \wedge OF)$	Greater or equal(signed $\geq$ )
setl $D$	$D \leftarrow SF \wedge OF$	Less(signed $<$ )
setle $D$	$D \leftarrow (SF \wedge OF)   ZF$	Less or equal(signed $\leq$ )

虽然看上去相对复杂一点，不过原理都是一致的。

对于无符号数的比较情况，需要注意一下，指令 `cmp` 会设置进位标志，因而针对无符号数的比较，采用的是进位标志和零标志的组合，具体的条件码组合如图所示。

指令	影响	描述
<code>seta D</code>	$D \leftarrow \sim CF \& \sim ZF$	Above(unsigned >)
<code>setae D</code>	$D \leftarrow \sim CF$	Above or equal(unsigned >=)
<code>setb D</code>	$D \leftarrow CF$	Below(unsigned <)
<code>setbe D</code>	$D \leftarrow CF ZF$	Below or equal(unsigned <=)

关于这些条件码的组合并不需要去记住，了解条件语句的底层实现，这对我们深入理解整个计算机系统会有一定的帮助。

## 跳转指令

```
1 long absdiff_se(long x, long y){  
2     long result;  
3     if(x < y){result = y - x;} else{result = x - y;}  
4     return result;  
5 }
```

absdiff\_se:

cmpq %rsi, %rdi

jl .L4

movq %rdi, %rax

subq %rsi, %rax

ret

条件语句  $x$  小于  $y$  由指令 `cmp` 来实现，指令 `cmp` 会根据  $(x-y)$  的结果来设置符号标志 (SF) 和溢出标志 (OF)。图中的跳转指令 `jl`，根据符号标志 (SF) 和溢出标志 (OF) 的异或结果来判断究竟是顺序执行，还是跳转到 L4 处执行。当  $x$  大于  $y$  时，指令顺序执行，然后返回执行结果，L4 处的指令不会被执行；当  $x$  小于  $y$  时，程序跳转到 L4 处执行，然后返回执行结果，跳转指令会根据条件寄存器的某种组合来决定是否进行跳转。

指令	跳转条件	描述
<code>jg Label</code>	$\sim (SF \wedge OF) \& \sim ZF$	Greater(signed $>$ )
<code>jge Label</code>	$\sim (SF \wedge OF)$	Greater or equal(signed $\geq$ )
<code>jl Label</code>	$SF \wedge OF$	Less(signed $<$ )
<code>jle Label</code>	$(SF \wedge OF)   ZF$	Less or equal(signed $\leq$ )

对于代码中的 if-else 语句，当满足条件时，程序沿着一条执行路径执行，当不满足条件时，就走另外一条路径。这种机制比较简单和通用，但是在现代处理器上，它的执行效率可能会比较低。

针对这种情况，有一种替代的策略，就是使用数据的条件转移来代替控制的条件转移。还是针对两个数差的绝对值问题，给出了另外一种实现方式，具体如下所示。

## 代码实现

```
1 long comvdiff_se(long x, long y){  
2     long rval = y - x; long eval = x - y;  
3     long ntest = x >= y;  
4     if(ntest){rval = eval;} return rval;  
5 }
```

我们既要计算  $y-x$  的值，也要计算  $x-y$  的值，分别用两个变量来记录结果，然后再判断  $x$  与  $y$  的大小，根据测试情况来判断是否更新返回值。这两种写法看上去差别不大，但第二种效率更高。第二种代码的汇编指令如下所示。

<b>long</b> cmovdiff_se( <b>long</b> x, <b>long</b> y){	cmovdiff_se:
<b>long</b> rval = y - x;	<b>movq</b> %rsi, %rdx
<b>long</b> eval = x - y;	<b>subq</b> %rdi, %rdx
<b>long</b> ntest = x >= y;	<b>movq</b> %rdi, %rax
<b>if</b> (ntest)	<b>subq</b> %rsi, %rax
rval = eval;	<b>cmpq</b> %rsi, %rdi
<b>return</b> rval;	<b>cmovge</b> %rdx, %rax
}	<b>ret</b>

前面这几条指令都是普通的数据传送和减法操作。cmovge 是根据条件码的某种组合来进行有条件的传送数据，当满足规定的条件时，将寄存器 rdx 内的数据复制到寄存器 rax 内。在这个例子中，只有当 x 大于等于 y 时，才会执行这一条指令。

cmpq %rsi, %rdi  $\rightarrow$  compare x:y

cmovge %rdx, %rax  $\rightarrow \sim (SF \wedge OF)$



更多条件传送指令如下表所示。

指令	移动条件	描述
<code>cmovg S, R</code>	$\sim (SF \wedge OF) \& \sim ZF$	Greater(signed >)
<code>cmovge S, R</code>	$\sim (SF \wedge OF)$	Greater or equal(signed >=)
<code>cmovl S, R</code>	$SF \wedge OF$	Less(signed <)
<code>cmovle S, R</code>	$(SF \wedge OF)   ZF$	Less or equal(signed <=)

为什么基于条件传送的代码会比基于跳转指令的代码效率高呢？这里涉及到现代处理器通过流水线来获得高性能。当遇到条件跳转时，处理器会根据分支预测器来猜测每条跳转指令是否执行，当发生错误预测时，会浪费大量的时间，导致程序性能严重下降。

C 语言中提供了三种循环结构，即 do-while、while 以及 for 语句，汇编语言中没有定义专内的指令来实现循环结构，循环语句是通过条件测试与跳转的结合来实现的。

接下来，我们分别用这三种循环结构来实现 N 的阶乘。

<b>long</b> fact_do( <b>long</b> n){	fact_do:
<b>long</b> result = 1;	n   in   %rdi
<b>do</b> {	<b>movl</b> \$1, %eax
result *= n;	.L2:
n = n - 1;	<b>imulq</b> %rdi, %rax
<b>}</b> <b>while</b> (n > 1);	<b>subq</b> \$1, %rdi
<b>return</b> result;	<b>cmpq</b> \$1, %rdi
	<b>jg</b> .L2

我们可以发现指令 `cmp` 与跳转指令的组合实现了循环操作。

```
fact_do:
    n in %rdi
    movl    $1, %eax
.L2:
    imulq   %rdi, %rax
    subq    $1, %rdi
    cmpq    $1, %rdi
    jg.L2
    rep    ret
```

当 `n` 大于 1 时，跳转到 L2 处执行循环，直到 `n` 的值减小到 1，循环结束。

对比 do-while 循环和 while 循环的实现方式，我们可以发现这两种循环的差别在于，N 大于 1 这个循环测试的位置不同。

<pre>long fact_do(long n){     long result = 1;     do{         result *= n;         n = n - 1;     } while (n &gt; 1);     return result; }</pre>	<pre>long fact_while(long n){     long result = 1;     while (n &gt; 1){         result *= n;         n = n - 1;     }     return result; }</pre>
--	---

do-while 循环是先执行循环体的内容，然后再进行循环测试，while 循环则是先进行循环测试，根据测试结果是否执行循环体内容。

我们将这个 for 循环转换成 while 循环。

```
long fact_for(long n){  
    long i;  
    long result = 1;  
    for (i=2; i<=n; i++) {  
        result *= i;  
        return result;  
    }  
}
```

```
long fact_for_while(long n){  
    long i = 2;  
    long result = 1;  
    while (i<=n) {  
        result *= i;  
        i++;  
    }  
    return result;  
}
```

对比 for 循环和 while 循环产生的汇编代码，可以发现除了这一句跳转指令不同，其他部分都是一致的。

fact for while:

```
movl    $1, %eax
```

```
movl    $2, %edx
```

```
jmp .L2
```

.L3

**imulq**    %rdx, %rax

```
addq    $1, %rdx
```

.L2

```
cmpq    %rdi, %rdx
```

j1 .L3

```
rep ret
```

- 这两个汇编代码是采用-Og 选项产生的。
- 综上所述，三种形式的循环语句都是通过条件测试和跳转指令来实现。

## switch

```
1 void switch_eg(long x, long n, long *dest){
2     long val = x;
3     switch(n){
4         case 0: val *= 13; break;
5         case 2: val += 10; break;
6         case 3: val += 11; break;
7         case 4:
8         case 6: val += 11; break;
9         default: val = 0;
10    }
11    *dest = val;
12 }
```



在针对一个测试有多种可能的结果时，switch 语言特别有用，vitch 语句通过跳转表这种数据结构，使得实现更加的高效。接下来我们通看看所对应的汇编指令。

switch\_eg:

n in %rsi

**cmpq**     \$6, %rsi   compare n:6

ja .L8

```
leaq .L4(%rip), %rcx
```

```
movslq (%rcx, %rsi, 4), %rax
```

```
addq    %rcx    %rax
```

```
jmp *%rax
```

指令 `cmp` 判断参数 `n` 与立即数 6 的大小，如果 `n` 大于 6，程序跳转到 `default` 对应的 L8 程序段。`case0 case6` 的情况，可以通过跳转表来访问不同分支。代码将跳转表声明为一个长度为 7 的数组，每个元素都是一个指向代码位置的指针，具体对应关系如图所示。

.L4:

.long: .L3-.L4  $\rightarrow$  Case 0

.long: .L8-.L4  $\rightarrow$  Case 1

.long: .L5-.L4  $\rightarrow$  Case 2

**.long: .L6-.L4 → Case 3**

.long: .L7-.L4  $\rightarrow$  Case 4

.long: .L8-.L4  $\rightarrow$  Case 5

.long: .L7-.L4  $\rightarrow$  Case 6

数组的长度为 7，是因为需要覆盖 Case0 Case6 的情况，对重复的情况 case4 和 case6，使用相同的标号。

.L4:

.long: .L3-.L4  $\rightarrow$  Case 0

.long: .L8-.L4  $\rightarrow$  Case 1

.long: .L5-.L4  $\rightarrow$  Case 2

.long: .L6-.L4  $\rightarrow$  Case 3

**.long: .L7-.L4 → Case 4**

.long: .L8-.L4  $\rightarrow$  Case 5

.long: .L7-.L4 → Case 6

对于缺失的 case1 和 case5 的情况, 使用默认情况的标号。

.L4:

.long: .L3-.L4  $\rightarrow$  Case 0

.long: **.L8-.L4** → Case 1

.long: .L5-.L4  $\rightarrow$  Case 2

.long: .L6-.L4  $\rightarrow$  Case 3

.long: .L7-.L4 → Case 4

.long: **.L8-.L4** → Case 5

.long: .L7-.L4  $\rightarrow$  Case 6

在这个例子中，程序使用跳转表来处理多重分支，甚至当 switch 有上百种情况时，虽然跳转表的长度会增加，但是程序的执行只需要一次跳转也能处理复杂分支的情况，与使用一组很长的 if-else 相比，使用跳转表的优点是执行 switch 语句的时间与 case 的数量是无关的。因此在处理多重分支的时，与一组很长的 if-else 相比，switch 的执行效率要高。