Datawhale

Datawhale 开源社区

Datawhale open source community

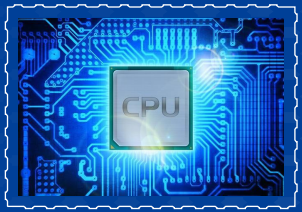
深入理解计算机系统（6）

Computer Systems A Programmer's Perspective CSAPP

李岳昆、易远哲

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

2021 年 12 月 1 日



第 I 部分

程序的机器级表示-II

算术和逻辑操作　　控制

加载有效地址

Datawhale

- 首先我们看一下指令 leaq，它实现的功能是加载有效地址，q 表示地址的长度是四个字，由于 x86-64 位处理器上，地址长度都是 64 位，因此不存在 leab、leaw 这类有关大小的变种。

leaq S, D → Load Effective Address

- 例如下面的这条指令，它表示的含义是把有效地址复制到寄存器 rax 中。

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

- 这个源操作数看上去与内存引用的格式类似，有效地址的计算方式与之前讲到的内存地址的计算方式一致，可以通过下列中的公式计算得到.

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

加载有效地址

Datawhale

假设寄存器 rdx 内保存的数值为 x，那么有效地址的值为 7 + %rdx + %rdx * 4 = 7 + 5x。注意，对于 leaq 指令所执行的操作并不是去内存地址 (5x+7) 处读取数据，而是将有效地址 (5x+7) 这个值直接写入到目的寄存器 rax。

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：

leaq

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

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
```

接下来我们看一下，如何通过 leaq 指令实现算术运算。

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。

```
t = x + 4 * y + 12 * z;

scale:
   x in %rdi,  y in %rsi, z in %rdx
   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）
```

指令 leaq 将该数值保存到目的寄存 rax 中。

接下来关于 z*12 的乘法运算会有一些复杂，需要分成两步：

1. 首先计算 3*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。

```
t = x + 4 * y + 12 * z;

scale:
   x in %rdi,  y in %rsi, z in %rdx
   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）
```

延拓思考

Datawhale

为什么不能使用下列这条指令，直接一步得到我们期望的结果?

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

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。

一元和二元操作

Datawhale

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
$INC\ D$	$D \leftarrow D+1$	加 1
$DEC\ D$	$D \leftarrow D-1$	减 1
$NEG\ D$	$D \leftarrow -D$	取负
$NOT\ D$	$D \leftarrow \sim D$	取补

二元操作

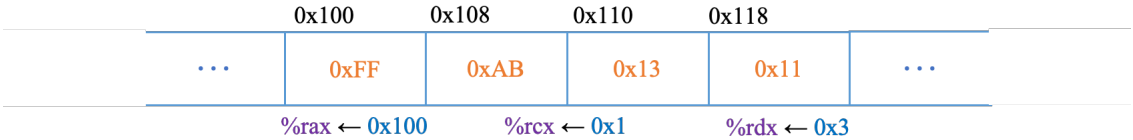
Datawhale

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
$ADD\ S, D$	D ← D+S	加
$SUB\ S, D$	D ← D-S	减
$IMUL\ S, D$	D ← D*S	乘
$XOR\ S, D$	D ← D^ S	异或
$OR\ S, D$	D ← D|S	或
$AND\ S, D$	D ← D&S	与

Datawhale

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。

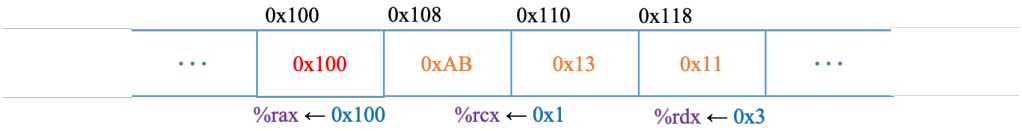


addq %rcx，（%rax） → Mem[0x100] = Mem[0x100] + R[%rcx]
subq %rdx，8(%rax) → Mem[0x108] = Mem[0x108] − R[%rdx]
incq 16(%rax)， → Mem[0x110] = Mem[0x110] + 1
subq %rdx，%rax → R[%rax] = R[%rax] − R[%rdx]

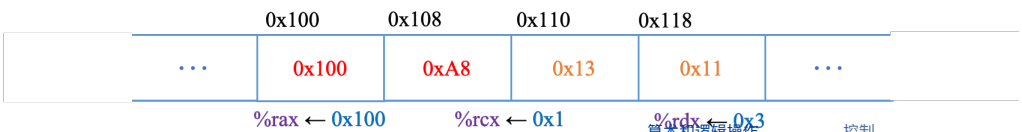
内存与寄存器中

Datawhale

❶ 加法指令 addq 是将内存地址 0x100 内的数据与寄存器 rcx 相加，二者之和再存储到内存地址 0x100 处，该指令执行完毕后，内存地址 0x100 处所存储的数据由 0xFF 变成 0x100。

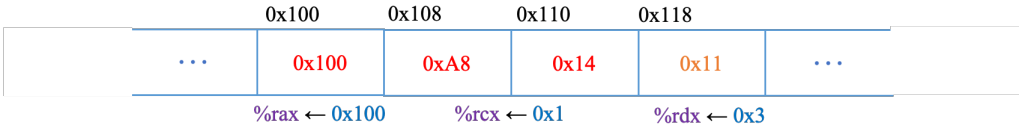


❷ 减法指令 subq 是将内存地址 0x108 内的数据减去寄存器 rdx 内的数据，二者之差在存储到内存地址 0x108 处，该指令执行完毕后，内存地址 0x108 处所存储的数据由 0xAB 变成 0xA8。

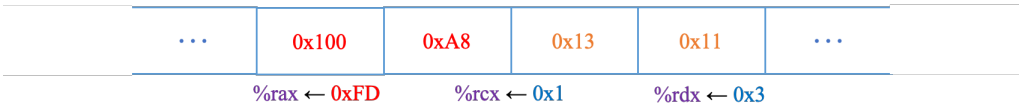


Datawhale

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



❹ 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。



移位操作

Datawhale

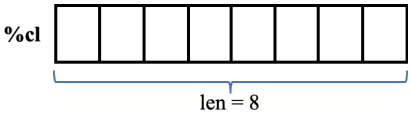
左移指令有两个，分别是 SAL 和 SHL，二者的效果是一样的，都是在右边填零；右移指令不同，分为算术右移和逻辑右移，算术右移需要填符号位，逻辑右移需要填零，这与 C 语言中所讲述的移位操作是一致的。

指令	影响	描述
$SAL\ k, D$	$D \leftarrow D << k$	左移
$SHL\ k, D$	$D \leftarrow D << k$	左移，等同于 SAL
$SAR\ k, D$	$D \leftarrow D >>_A k$	算数右移
$SHR\ k, D$	$D \leftarrow D >>_L k$	逻辑右移

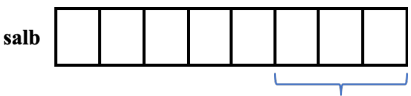
移位操作

Datawhale

- 对于移位量 k，可以是一个立即数，或者是放在寄存器 cl 中的数，对于移位指令只允许以特定的寄存器 cl 作为操作数，其他寄存器不行，这里需要特别注意一下。



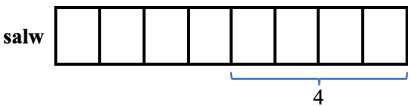
- 由于寄存器 cl 的长度为 8，原则上移位量的编码范围可达 28 - 1 (255)，实际上，对于 w 位的操作数进行移位操作，移位量是由寄存器 cl 的低 m 位来决定，也就是说，对于指令 salb，当目的操作数是 8 位，移位量由寄存器 cl 的低 3 位来决定。



移位操作

Datawhale

- 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。

移位操作

Datawhale

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

```
long arith(long x, long y, long z){
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0xF0F0F0F;
    long t4 = t2 - t3;
    return t4;
}
```

移位操作

Datawhale

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*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
```

移位操作

❷ 第二步，将寄存器 rax 进行左移 4 位，左移 4 位的操作是等效于乘以 2 的四次方，也就是乘以 16。

```
salq  $4，%rax
```

$$2^4 * R[\%rax] = 16 * (3 * z) = 48 * z$$

通过一条 leaq 指令和一条左移指令，来实现乘法操作。

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。

移位操作

Datawhale

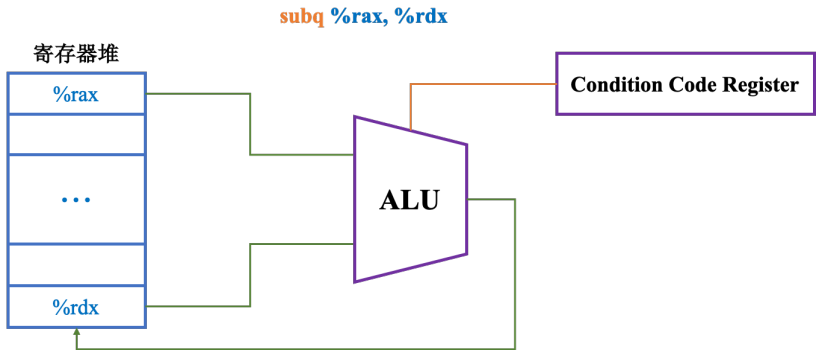
此外，还有一些特殊的算术指令，对于汇编指令学习，最关键的是了解指令相关的基本概念，并不需要去记指令的细枝末节，学会查阅指令手册，能够找到需要的信息即可。

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

条件码

Datawhale

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

Datawhale

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
```

条件码

Datawhale

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 —— 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

Datawhale

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
$INC\ D$	$ADD\ S, D$	$SAL\ k, D$
$DEC\ D$	$SUB\ S, D$	$SHL\ k, D$
$NEG\ D$	$IMUL\ S, D$	$SAR\ k, D$
	$OR\ S, D$	$SHR\ k, D$
	$XOR\ S, D$	
	$AND\ S, D$	

访问条件码

Datawhale

对于不同的指令也定义了相应的规则来设置条件码寄存器。例如逻辑操作指令 xor，进位标志 (CF) 和溢出标志 (OF) 会置 0；对于加一指令和减一指令会设置溢出标志 (OF) 和零标志 (ZF)，但不会改变进位标志 (CF)。

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。

- cmp 指令是根据两个操作数的差来设置条件码寄存器。cmp 指令和减法指令 (sub) 类似，也是根据两个操作是的差来设置条件码，二者不同的是 cmp 指令只是设置条件码寄存器，并不会更新目的寄存器的值。
- test 指令和 and 指令类似，同样 test 指令只是设置条件码寄存器，而不改变目的寄存器的值。

访问条件码

Datawhale

条件码的使用

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

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

图: 对应汇编指令

访问条件码

Datawhale

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

访问条件码

Datawhale

条件码的使用

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

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

图: 对应汇编指令

访问条件码

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 根据需标志 (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。

访问条件码

Datawhale

```
comp:
    a in %rdi, b in %rsi
    comq    %rsi，%rdi → a-b
    sete    %al

    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。

访问条件码

Datawhale

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
```

访问条件码

Datawhale

对比前面相等的情况，可以发现指令有些不同。sete 变成了指令 setl，指令 setl 的含义是如果 a 小于 b，将寄存器 al 设置为 1，其中后缀 l 是 less 的缩写，表示" 在小于时设置"，而不是表示大小 long word，这里特别注意一下。

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。

$$a<b \rightarrow SF \wedge OF$$

$$t = a - b$$

Case 1：a<b	t<0	SF=1	SF^OF=1
Case 2：a>b	t>0	SF=0	SF^OF=0

访问条件码

Datawhale

那么是不是根据符号标志 (SF) 就能够给出判断结论了呢? 我们来看一个例子。

Case 3：a<b　　t<0　　a=-2　　b=127

t=127>0,　　SF=0　OF=1　　SF^OF=1

溢出后，符号标志 SF 不会置一，但溢出标志 OF 会置一。因此仅仅通过符号标志无法判断 a 是否小于 b。

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^OF=0

访问条件码

综合上述所有的情况，根据符号标志 (SF) 和溢出标志 (OF) 的异或结果，可以对 a 小于 b 是否为真做出判断。

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	$D \leftarrow \sim (SF \hat{} OF) \& \sim ZF$	Greater(signed $>$)
setge D	$D \leftarrow \sim (SF \hat{} OF)$	Greater or equal(signed $>=$)
setl D	$D \leftarrow SF \hat{} OF$	Less(signed $<$)
setle D	$D \leftarrow (SF \hat{} OF) \mid ZF$	Less or equal(signed $<=$)

虽然看上去相对复杂一点，不过原理都是一致的。

访问条件码

Datawhale

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
seta D	$D \leftarrow \sim CF \& \sim ZF$	Above(unsigned $>$)
setae D	$D \leftarrow \sim CF$	Above or equal(unsigned $>=$)
setb D	$D \leftarrow CF$	Below(unsigned $<$)
setbe D	$D \leftarrow CF \mid ZF$	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

Datawhale

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
```

跳转指令

Datawhale

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
jg Label	$\sim (SF\hat{}OF)\&\sim ZF$	Greater(signed >)
jge Label	$\sim (SF\hat{}OF)$	Greater or equal(signed >=)
jl Label	$SF\hat{}OF$	Less(signed <)
jle Label	$(SF\hat{}OF)\|ZF$	Less or equal(signed <=)

跳转指令的编码

对于代码中的 if-else 语句，当满足条件时，程序洽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

```
long comvdiff_se(long x, long y){
    long rval = y - x; long eval = x - y;
    long ntest = x >= y;
    if(ntest){rval = eval;} return rval;
}
```

跳转指令的编码

Datawhale

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。

```
long cmovdiff_se(long x, long y){
    long rval = y – x;
    long eval = x – y;
    long ntest = x >= y;
    if (ntest)
        rval = eval;
    return rval;
}
```

```
cmovdiff_se:
    movq    %rsi，  %rdx
    subq    %rdi，  %rdx
    movq    %rdi,   %rax
    subq    %rsi,   %rax
    cmpq    %rsi，  %rdi
    cmovge  %rdx，  %rax
    ret
```

跳转指令的编码

Datawhale

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 $\rightarrow\sim (SF\hat{}OF)$

跳转指令的编码

更多条件传送指令如下表所示。

指令	移动条件	描述
cmovg S, R	$\sim (SF \hat{} OF) \& \sim ZF$	Greater(signed $>$)
cmovge S, R	$\sim (SF \hat{} OF)$	Greater or equal(signed $>=$)
cmovl S, R	$SF \hat{} OF$	Less(signed $<$)
cmovle S, R	$(SF \hat{} OF) \mid ZF$	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。

循环

Datawhale

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导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。

```
long fact_do(long n){
  long result = 1;
  do{
    result *= n;
    n = n - 1;
  } while (n > 1);
  return result;
```

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

循环

Datawhale

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。

```
long fact_do(long n){
    long result = 1;
    do{
        result *= n;
        n = n - 1;
    }while (n > 1);
    return result;
}
```

```
long fact_while(long n){
    long result = 1;
    while (n > 1){
        result *= n;
        n = n - 1;
    }
    return result;
}
```

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:
    movl    $1, %eax
    movl    $2, %edx
    jmp .L2
.L3
    imulq   %rdx, %rax
    addq    $1, %rdx
.L2
    cmpq    %rdi, %rdx
    jle .L3
    rep ret
```

```
fact_for_while:
    movl    $1, %eax
    movl    $2, %edx
    jmp .L2
.L3
    imulq   %rdx, %rax
    addq    $1, %rdx
.L2
    cmpq    %rdi, %rdx
    jl .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 语句

Datawhale

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 }
```

switch 语句

Datawhale

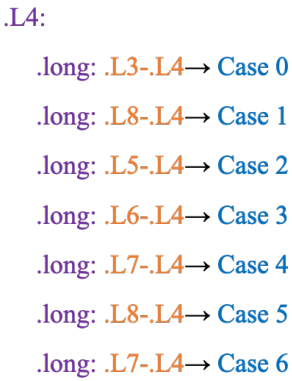
在针对一个测试有多种可能的结果时，switch 语言特别有用，vitch 语句通过跳转表这种数据结构，使得实现更加的高效。接下来我们通看看所对应的汇编指令。

```
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
```

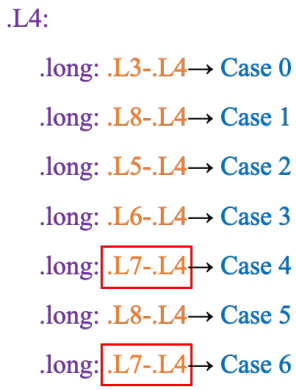
switch 语句

Datawhale

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。



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。



对于缺失的 case1 和 case5 的情况，使用默认情况的标号。

```
.L4:
    .long: .L3-.L4→ Case 0
    .long: .L8-.L4→ Case 1
    .long: .L5-.L4→ Case 2
    .long: .L6-.L4→ Case 3
    .long: .L7-.L4→ Case 4
    .long: .L8-.L4→ Case 5
    .long: .L7-.L4→ 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。