


填写下表，给出下面指令的效果，说明将被更新的寄存器或内存位置，以及得到的值：

指令	目的	值
<code>addq %rcx, (%rax)</code>		
<code>subq %rdx, 8(%rax)</code>		
<code>imulq \$16, (%rax, %rdx, 8)</code>		
<code>incq 16(%rax)</code>		
<code>decq %rcx</code>		
<code>subq %rdx, %rax</code>		

3.5.3 移位操作

最后一组是移位操作，先给出移位数，然后第二项给出的是要移位的数。可以进行算术和逻辑右移。移位量可以是一个立即数，或者放在单字节寄存器`%cl`中。（这些指令很特别，因为只允许以这个特定的寄存器作为操作数。）原则上来说，1个字节的移位量使得移位量的编码范围可以达到 $2^8 - 1 = 255$ 。x86-64中，移位操作对 w 位长的数据值进行操作，移位量是由`%cl`寄存器的低 m 位决定的，这里 $2^m = w$ 。高位会被忽略。所以，例如当寄存器`%cl`的十六进制值为`0xFF`时，指令`salb`会移7位，`salw`会移15位，`sall`会移31位，而`salq`会移63位。

如图3-10所示，左移指令有两个名字：`SAL`和`SHL`。两者的效果是一样的，都是将右边填上0。右移指令不同，`SAR`执行算术移位（填上符号位），而`SHR`执行逻辑移位（填上0）。移位操作的目的操作数可以是一个寄存器或是一个内存位置。图3-10中用`>>_a`（算术）和`>>_l`（逻辑）来表示这两种不同的右移运算。

 **练习题 3.9** 假设我们想生成以下C函数的汇编代码：

```
long shift_left4_rightn(long x, long n)
{
    x <<= 4;
    x >>= n;
    return x;
}
```

下面这段汇编代码执行实际的移位，并将最后的结果放在寄存器`%rax`中。此处省略了两条关键的指令。参数`x`和`n`分别存放在寄存器`%rdi`和`%rsi`中。

```
long shift_left4_rightn(long x, long n)
x in %rdi, n in %rsi
shift_left4_rightn:
    movq    %rdi, %rax      Get x
                           x <<= 4
    movl    %esi, %ecx      Get n (4 bytes)
                           x >>= n
```

根据右边的注释，填出缺失的指令。请使用算术右移操作。

3.5.4 讨论

我们看到图3-10所示的大多数指令，既可以用于无符号运算，也可以用于补码运算。