操作被分为四组：加载有效地址、一元操作、二元操作和移位。二元操作有两个操作数，而一元操作有一个操作数。这些操作数的描述方法与 3.4 节中所讲的一样。

| 指令 | | 效果 | 描述 |
|---|---|---|---|
| leaq | $S, D$ | $D \leftarrow \&S$ | 加载有效地址 |
| 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 |
| ADD | $S, D$ | $D \leftarrow D+S$ | 加 |
| SUB | $S, D$ | $D \leftarrow D-S$ | 减 |
| IMUL | $S, D$ | $D \leftarrow D*S$ | 乘 |
| XOR | $S, D$ | $D \leftarrow D \hat{} S$ | 异或 |
| OR | $S, D$ | $D \leftarrow D \mid S$ | 或 |
| AND | $S, D$ | $D \leftarrow D \& S$ | 与 |
| SAL | $k, D$ | $D \leftarrow D << k$ | 左移 |
| SHL | $k, D$ | $D \leftarrow D << k$ | 左移（等同于SAL） |
| SAR | $k, D$ | $D \leftarrow D >>_A k$ | 算术右移 |
| SHR | $k, D$ | $D \leftarrow D >>_L k$ | 逻辑右移 |

图 3-10　整数算术操作。加载有效地址(leaq)指令通常用来执行简单的算术操作。其余的指令是更加标准的一元或二元操作。我们用$>>_A$和$>>_L$来分别表示算术右移和逻辑右移。注意，这里的操作顺序与 ATT 格式的汇编代码中的相反

### 3.5.1　加载有效地址

*加载有效地址*(load effective address)指令 leaq 实际上是 movq 指令的变形。它的指令形式是从内存读数据到寄存器，但实际上它根本就没有引用内存。它的第一个操作数看上去是一个内存引用，但该指令并不是从指定的位置读入数据，而是将有效地址写入到目的操作数。在图 3-10 中我们用 C 语言的地址操作符 &S 说明这种计算。这条指令可以为后面的内存引用产生指针。另外，它还可以简洁地描述普通的算术操作。例如，如果寄存器%rdx 的值为 x，那么指令 leaq 7(%rdx,%rdx,4),%rax 将设置寄存器%rax 的值为 $5x+7$。编译器经常发现 leaq 的一些灵活用法，根本就与有效地址计算无关。目的操作数必须是一个寄存器。

为了说明 leaq 在编译出的代码中的使用，看看下面这个 C 程序：

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

编译时，该函数的算术运算以三条 leaq 指令实现，就像右边注释说明的那样：

```
  long scale(long x, long y, long z)
  x in %rdi, y in %rsi, z in %rdx
scale:
  leaq    (%rdi,%rsi,4), %rax     x + 4*y
  leaq    (%rdx,%rdx,2), %rdx     z + 2*z = 3*z
  leaq    (%rax,%rdx,4), %rax     (x+4*y) + 4*(3*z) = x + 4*y + 12*z
  ret
```