比如说，假设我们用一条 ADD 指令完成等价于 C 表达式 t=a+ b 的功能，这里变量 a、b 和 t 都是整型的。然后，根据下面的 C 表达式来设置条件码：

```
CF    (unsigned) t <  (unsigned) a        无符号溢出
ZF    (t == 0)                            零
SF    (t<0)                               负数
OF    (a<0==b<0) && (t<0 !=a<0)           有符号溢出
```

leaq 指令不改变任何条件码，因为它是用来进行地址计算的。除此之外，图 3-10 中列出的所有指令都会设置条件码。对于逻辑操作，例如 XOR，进位标志和溢出标志会设置成 0。对于移位操作，进位标志将设置为最后一个被移出的位，而溢出标志设置为 0。INC 和 DEC 指令会设置溢出和零标志，但是不会改变进位标志，至于原因，我们就不在这里深入探讨了。

除了图 3-10 中的指令会设置条件码，还有两类指令(有 8、16、32 和 64 位形式)，它们只设置条件码而不改变任何其他寄存器；如图 3-13 所示。CMP 指令根据两个操作数之差来设置条件码。除了只设置条件码而不更新目的寄存器之外，CMP 指令与 SUB 指令的行为是一样的。在 ATT 格式中，列出操作数的顺序是相反的，这使代码有点难读。如果两个操作数相等，这些指令会将零标志设置为 1，而其他的标志可以用来确定两个操作数之间的大小关系。TEST 指令的行为与 AND 指令一样，除了它们只设置条件码而不改变目的寄存器的值。典型的用法是，两个操作数是一样的(例如，testq %rax,%rax 用来检查%rax 是负数、零，还是正数)，或其中的一个操作数是一个掩码，用来指示哪些位应该被测试。

指令		基于	描述
CMP	S_1, S_2	S_2-S_1	比较
cmpb			比较字节
cmpw			比较字
cmpl			比较双字
cmpq			比较四字
TEST	S_1, S_2	$S_1 \& S_2$	测试
testb			测试字节
testw			测试字
testl			测试双字
testq			测试四字

图 3-13　比较和测试指令。这些指令不修改任何寄存器的值，只设置条件码

3.6.2　访问条件码

条件码通常不会直接读取，常用的使用方法有三种：1)可以根据条件码的某种组合，将一个字节设置为 0 或者 1，2)可以条件跳转到程序的某个其他的部分，3)可以有条件地传送数据。对于第一种情况，图 3-14 中描述的指令根据条件码的某种组合，将一个字节设置为 0 或者 1。我们将这一整类指令称为 SET 指令；它们之间的区别就在于它们考虑的条件码的组合是什么，这些指令名字的不同后缀指明了它们所考虑的条件码的组合。这些指令的后缀表示不同的条件而不是操作数大小，了解这一点很重要。例如，指令 setl 和 setb 表示“小于时设置(set less)”和“低于时设置(set below)”，而不是“设置长字(set long word)”和“设置字节(set byte)”。

一条 SET 指令的目的操作数是低位单字节寄存器元素(图 3-2)之一，或是一个字节的内存位置，指令会将这个字节设置成 0 或者 1。为了得到一个 32 位或 64 位结果，我们必须对高位清零。一个计算 C 语言表达式 a < b 的典型指令序列如下所示，这里 a 和 b 都是 long 类型：