

**旁注 理解数据传送如何改变目的寄存器**

正如我们描述的那样，关于数据传送指令是否以及如何修改目的寄存器的高位字节有两种不同的方法。下面这段代码序列会说明其差别：

```


1      movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2      movb    $-1, %al                     %rax = 00112233445566FF
3      movw    $-1, %ax                     %rax = 001122334455FFFF
4      movl    $-1, %eax                    %rax = 00000000FFFFFFFF
5      movq    $-1, %rax                    %rax = FFFFFFFF00000000

```

在接下来的讨论中，我们使用十六进制表示。在这个例子中，第1行的指令把寄存器%rax初始化为位模式0011223344556677。剩下的指令的源操作数值是立即数值-1。回想-1的十六进制表示形如FF...F，这里F的数量是表述中字节数量的两倍。因此movb指令（第2行）把%rax的低位字节设置为FF，而movw指令（第3行）把低2位字节设置为FFFF，剩下的字节保持不变。movl指令（第4行）将低4个字节设置为FFFFFFFF，同时把高位4字节设置为00000000。最后movq指令（第5行）把整个寄存器设置为FFFFFFFFFFFFFFFF。

注意图3-5中并没有一条明确的指令把4字节源值零扩展到8字节目的。这样的指令逻辑上应该被命名为movzqlq，但是并没有这样的指令。不过，这样的数据传送可以用以寄存器为目的的movl指令来实现。这一技术利用的属性是，生成4字节值并以寄存器作为目的的指令会把高4字节置为0。对于64位的目标，所有三种源类型都有对应的符号扩展传送，而只有两种较小的源类型有零扩展传送。

图3-6还给出cmtq指令。这条指令没有操作数：它总是以寄存器%eax作为源，%rax作为符号扩展结果的目的。它的效果与指令movslq %eax,%rax完全一致，不过编码更紧凑。

 **练习题 3.2** 对于下面汇编代码的每一行，根据操作数，确定适当的指令后缀。（例如，mov可以被重写成movb、movw、movl或者movq。）

```

mov__    %eax, (%rsp)
mov__    (%rax), %dx
mov__    $0xFF, %bl
mov__    (%rsp,%rdx,4), %dl
mov__    (%rdx), %rax
mov__    %dx, (%rax)

```

**旁注 字节传送指令比较**

下面这个示例说明了不同的数据传送指令如何改变或者不改变目的的高位字节。仔细观察可以发现，三个字节传送指令movb、movsbq和movzbq之间有细微的差别。示例如下：

```

1      movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2      movb    $0xAA, %dl                   %dl = AA
3      movb    %dl,%al                      %rax = 00112233445566AA
4      movsbq  %dl,%rax                     %rax = FFFFFFFF000000AA
5      movzbq  %dl,%rax                     %rax = 00000000000000AA

```

在下面的讨论中，所有的值都使用十六进制表示。代码的头2行将寄存器%rax和%dl分别初始化为0011223344556677和AA。剩下的指令都是将%rdx的低位字节复制到%rax的低位字节。movb指令（第3行）不改变其他字节。根据源字节的最高位，movsbq指令（第4行）将其他7个字节设为全1或全0。由于十六进制A表示二进制值1010，符号扩展会把高位字节都设置为FF。movzbq指令（第5行）总是将其他7个字节全都设置为0。