



leaq 指令能执行加法和有限形式的乘法，在编译如上简单的算术表达式时，是很有用处的。

 **练习题 3.6** 假设寄存器 %rax 的值为 x，%rcx 的值为 y。填写下表，指明下面每条汇编代码指令将存储在寄存器 %rdx 中的值：

表达式	结果
leaq 6(%ax),%rdx	
leaq (%rax,%rcx),%rdx	
leaq (%rax,%rcx,4),%rdx	
leaq 7(%rax,%rax,8),%rdx	
leaq 0xA(,%rcx,4),%rdx	
leaq 9(%rax,%rcx,2),%rdx	

 **练习题 3.7** 考虑下面的代码，我们省略了被计算的表达式：

```
long scale2(long x, long y, long z) {
    long t = _____;
    return t;
}
```

用 GCC 编译实际的函数得到如下的汇编代码：


```
long scale2(long x, long y, long z)
x in %rdi, y in %rsi, z in %rdx
scale2:
    leaq    (%rdi,%rdi,4), %rax
    leaq    (%rax,%rsi,2), %rax
    leaq    (%rax,%rdx,8), %rax
    ret
```

填写出 C 代码中缺失的表达式。

3.5.2 一元和二元操作

第二组中的操作是一元操作，只有一个操作数，既是源又是目的。这个操作数可以是一个寄存器，也可以是一个内存位置。比如说，指令 `incq(%rsp)` 会使栈顶的 8 字节元素加 1。这种语法让人想起 C 语言中的加 1 运算符(++)和减 1 运算符(--)。

第三组是二元操作，其中，第二个操作数既是源又是目的。这种语法让人想起 C 语言中的赋值运算符，例如 `x=y`。不过，要注意，源操作数是第一个，目的操作数是第二个，对于不可交换操作来说，这看上去很奇特。例如，指令 `subq %rax,%rdx` 使寄存器 %rdx 的值减去 %rax 中的值。(将指令解读成“从 %rdx 中减去 %rax”会有所帮助。)第一个操作数可以是立即数、寄存器或是内存位置。第二个操作数可以是寄存器或是内存位置。注意，当第二个操作数为内存地址时，处理器必须从内存读出值，执行操作，再把结果写回内存。

 **练习题 3.8** 假设下面的值存放在指定的内存地址和寄存器中：

地址	值
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

寄存器	值
%rax	0x100
%rcx	0x1
%rdx	0x3