注意，无论是无符号运算还是补码运算，乘以 2 的幂都可能会导致溢出。结果表明，即使溢出的时候，我们通过移位得到的结果也是一样的。回到前面的例子，我们将 4 位模式[1011]（数值为 11）左移两位得到[101100]（数值为 44）。将这个值截断为 4 位得到[1100]（数值为 12=44 mod 16）。

由于整数乘法比移位和加法的代价要大得多，许多 C 语言编译器试图以移位、加法和减法的组合来消除很多整数乘以常数的情况。例如，假设一个程序包含表达式 `x * 14`。利用 $14=2^3+2^2+2^1$，编译器会将乘法重写为 `(x<<3)+(x<<2)+(x<<1)`，将一个乘法替换为三个移位和两个加法。无论 x 是无符号的还是补码，甚至当乘法会导致溢出时，两个计算都会得到一样的结果。（根据整数运算的属性可以证明这一点。）更好的是，编译器还可以利用属性 $14=2^4-2^1$，将乘法重写为 `(x<<4)-(x<<1)`，这时只需要两个移位和一个减法。

练习题 2.38　就像我们将在第 3 章中看到的那样，LEA 指令能够执行形如 `(a<<k)+b` 的计算，这里 k 等于 0、1、2 或 3，而 b 等于 0 或者某个程序值。编译器常常用这条指令来执行常数因子乘法。例如，我们可以用 `(a<<1)+a` 来计算 `3*a`。

考虑 b 等于 0 或者等于 a、k 为任意可能的值的情况，用一条 LEA 指令可以计算 a 的哪些倍数？

归纳一下我们的例子，考虑一个任务，对于某个常数 K 的表达式 `x*K` 生成代码。编译器会将 K 的二进制表示表达为一组 0 和 1 交替的序列：

$$[(0\cdots0)(1\cdots1)(0\cdots0)\cdots(1\cdots1)]$$

例如，14 可以写成$[(0\cdots0)(111)(0)]$。考虑一组从位位置 n 到位位置 m 的连续的 1（$n\geqslant m$）。（对于 14 来说，我们有 $n=3$ 和 $m=1$。）我们可以用下面两种不同形式中的一种来计算这些位对乘积的影响：

形式 A：`(x<<n)+(x<<(n-1))+…+(x<<m)`

形式 B：`(x<<(n+1))-(x<<m)`

把每个这样连续的 1 的结果加起来，不用做任何乘法，我们就能计算出 `x*K`。当然，选择使用移位、加法和减法的组合，还是使用一条乘法指令，取决于这些指令的相对速度，而这些是与机器高度相关的。大多数编译器只在需要少量移位、加法和减法就足够的时候才使用这种优化。

练习题 2.39　对于位位置 n 为最高有效位的情况，我们要怎样修改形式 B 的表达式？

练习题 2.40　对于下面每个 K 的值，找出只用指定数量的运算表达 `x*K` 的方法，这里我们认为加法和减法的开销相当。除了我们已经考虑过的简单的形式 A 和 B 原则，你可能会需要使用一些技巧。

K	移位	加法/减法	表达式
6	2	1	
31	1	1	
−6	2	1	
55	2	2	

练习题 2.41　对于一组从位位置 n 开始到位位置 m 的连续的 1（$n\geqslant m$），我们看到可以产生两种形式的代码，A 和 B。编译器该如何决定使用哪一种呢？

2.3.7　除以 2 的幂

在大多数机器上，整数除法要比整数乘法更慢——需要 30 个或者更多的时钟周期。