2.37　A. 这个改动完全没有帮助。虽然 asize 的计算会更准确，但是调用 malloc 会导致这个值被转换成一个 32 位无符号数字，因而还是会出现同样的溢出条件。

B. malloc 使用一个 32 位无符号数作为参数，它不可能分配一个大于 2^{32} 个字节的块，因此，没有必要试图去分配或者复制这样大的一块内存。取而代之，函数应该放弃，返回 NULL，用下面的代码取代对 malloc 原始的调用(第 9 行)：

```
uint64_t required_size = ele_cnt * (uint64_t) ele_size;
size_t request_size = (size_t) required_size;
if (required_size != request_size)
    /* Overflow must have occurred. Abort operation */
    return NULL;
void *result = malloc(request_size);
if (result == NULL)
    /* malloc failed */
    return NULL;
```

2.38　在第 3 章中，我们将看到很多实际的 LEA 指令的例子。用这个指令来支持指针运算，但是 C 语言编译器经常用它来执行小常数乘法。

对于每个 k 的值，我们可以计算出 2 的倍数：2^k(当 b 为 0 时)和 2^k+1(当 b 为 a 时)。因此我们能够计算出倍数为 1，2，3，4，5，8 和 9 的值。

2.39　这个表达式就变成了-(x<<m)。要看清这一点，设字长为 w，$n=w-1$。形式 B 说我们要计算 (x<<w)-(x<<m)，但是将 x 向左移动 w 位会得到值 0。

2.40　本题要求你使用讲过的优化技术，同时也需要自己的一点儿创造力。

K	移位	加法/减法	表达式
6	2	1	(x<<2) + (x<<1)
31	1	1	(x<<5) - x
-6	2	1	(x<<1) - (x<<3)
55	2	2	(x<<6) - (x<<3) - x

可以观察到，第四种情况使用了形式 B 的改进版本。我们可以将位模式[110111]看作 6 个连续的 1 中间有一个 0，因而我们对形式 B 应用这个原则，但是需要在后来把中间 0 位对应的项减掉。

2.41　假设加法和减法有同样的性能，那么原则就是当 $n=m$ 时，选择形式 A，当 $n=m+1$ 时，随便选哪种，而当 $n>m+1$ 时，选择形式 B。

这个原则的证明如下。首先假设 $m>0$。当 $n=m$ 时，形式 A 只需要 1 个移位，而形式 B 需要 2 个移位和 1 个减法。当 $n=m+1$ 时，这两种形式都需要 2 个移位和 1 个加法或者 1 个减法。当 $n>m+1$ 时，形式 B 只需要 2 个移位和 1 个减法，而形式 A 需要 $n-m+1>2$ 个移位和 $n-m>1$ 个加法。对于 $m=0$ 的情况，对于形式 A 和 B 都要少 1 个移位，所以在两者中选择时，还是适用同样的原则。

2.42　这里唯一的挑战是不使用任何测试或条件运算来计算偏置量。我们利用了一个诀窍，表达式 x>>31 产生一个字，如果 x 是负数，这个字为全 1，否则为全 0。通过掩码屏蔽掉适当的位，我们就得到期望的偏置值。

```
int div16(int x) {
    /* Compute bias to be either 0 (x >= 0) or 15 (x < 0) */
    int bias = (x >> 31) & 0xF;
    return (x + bias) >> 4;
}
```

2.43　我们发现当人们直接与汇编代码打交道时是有困难的。但当把它放入 optarith 所示的形式中时，问题就变得更加清晰明了。

我们可以看到 M 是 31；是用 (x<<5)-x 来计算 x*M。