组	有效位	标记位	块[0]	块[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

6. 直接映射高速缓存中的冲突不命中

冲突不命中在真实的程序中很常见，会导致令人困惑的性能问题。当程序访问大小为 2 的幂的数组时，直接映射高速缓存中通常会发生冲突不命中。例如，考虑一个计算两个向量点积的函数：

```
1   float dotprod(float x[8], float y[8])
2   {
3       float sum = 0.0;
4       int i;
5
6       for (i = 0; i < 8; i++)
7           sum += x[i] * y[i];
8       return sum;
9   }
```

对于 x 和 y 来说，这个函数有良好的空间局部性，因此我们期望它的命中率会比较高。不幸的是，并不总是如此。

假设浮点数是 4 个字节，x 被加载到从地址 0 开始的 32 字节连续内存中，而 y 紧跟在 x 之后，从地址 32 开始。为了简便，假设一个块是 16 个字节(足够容纳 4 个浮点数)，高速缓存由两个组组成，高速缓存的整个大小为 32 字节。我们会假设变量 sum 实际上存放在一个 CPU 寄存器中，因此不需要内存引用。根据这些假设每个 x[i]和 y[i]会映射到相同的高速缓存组：

元素	地址	组索引	元素	地址	组索引
x[0]	0	0	y[0]	32	0
x[1]	4	0	y[1]	36	0
x[2]	8	0	y[2]	40	0
x[3]	12	0	y[3]	44	0
x[4]	16	1	y[4]	48	1
x[5]	20	1	y[5]	52	1
x[6]	24	1	y[6]	56	1
x[7]	28	1	y[7]	60	1

在运行时，循环的第一次迭代引用 x[0]，缓存不命中会导致包含 x[0]～x[3]的块被加载到组 0。接下来是对 y[0]的引用，又一次缓存不命中，导致包含 y[0]～y[3]的块被复制到组 0，覆盖前一次引用复制进来的 x 的值。在下一次迭代中，对 x[1]的引用不命中，导致 x[0]～x[3]的块被加载回组 0，覆盖掉 y[0]～y[3]的块。因而现在我们就有了一个冲突不命中，而且实际上后面每次对 x 和 y 的引用都会导致冲突不命中，因为我们在 x 和 y 的块之间抖动(thrash)。术语“抖动”描述的是这样一种情况，即高速缓存反复地加载和驱逐相同的高速缓存块的组。

简要来说就是，即使程序有良好的空间局部性，而且我们的高速缓存中也有足够的空间来存放 x[i]和 y[i]的块，每次引用还是会导致冲突不命中，这是因为这些块被映射到了同