- 你不可以调用任何库函数。
- 你的代码应该对任意 n 的值都能工作，包括当它不是 K 的倍数的时候。你可以用类似于使用循环展开时完成最后几次迭代的方法做到这一点。
- 你写的代码应该无论 K 的值是多少，都能够正确编译和运行。使用操作 sizeof 来做到这一点。
- 在某些机器上，未对齐的写可能比对齐的写慢很多。(在某些非 x86 机器上，未对齐的写甚至可能会导致段错误。)写出这样的代码，开始时直到目的地址是 K 的倍数时，使用字节级的写，然后进行字级的写，(如果需要)最后采用用字节级的写。
- 注意 cnt 足够小以至于一些循环上界变成负数的情况。对于涉及 sizeof 运算符的表达式，可以用无符号运算来执行测试。(参见 2.2.8 节和家庭作业 2.72。)

**5.18 在练习题 5.5 和 5.6 中我们考虑了多项式求值的任务，既有直接求值，也有用 Horner 方法求值。试着用我们讲过的优化技术写出这个函数更快的版本，这些技术包括循环展开、并行累积和重新结合。你会发现有很多不同的方法可以将 Horner 方法和直接求值与这些优化技术混合起来。

理想状况下，你能达到的 CPE 应该接近于你的机器的吞吐量界限。我们的最佳版本在参考机上能使 CPE 达到 1.07。

**5.19 在练习题 5.12 中，我们能够把前置和计算的 CPE 减少到 3.00，这是由该机器上浮点加法的延迟决定的。简单的循环展开没有改进什么。

使用循环展开和重新结合的组合，写出求前置和的代码，能够得到一个小于你机器上浮点加法延迟的 CPE。要达到这个目标，实际上需要增加执行的加法次数。例如，我们使用 2 次循环展开的版本每次迭代需要 3 个加法，而使用 4 次循环展开的版本需要 5 个。在参考机上，我们的最佳实现能达到 CPE 为 1.67。

确定你的机器的吞吐量和延迟界限是如何限制前置和操作所能达到的最小 CPE 的。

练习题答案

5.1 这个问题说明了内存别名使用的某些细微的影响。

正如下面加了注释的代码所示，结果会是将 xp 处的值设置为 0：

```
4       *xp = *xp + *xp; /* 2x */
5       *xp = *xp - *xp; /* 2x-2x = 0 */
6       *xp = *xp - *xp; /* 0-0 = 0 */
```

这个示例说明我们关于程序行为的直觉往往会是错误的。我们自然地会认为 xp 和 yp 是不同的情况，却忽略了它们相等的可能性。错误通常源自程序员没想到的情况。

5.2 这个问题说明了 CPE 和绝对性能之间的关系。可以用初等代数解决这个问题。我们发现对于 $n \leqslant 2$，版本 1 最快。对于 $3 \leqslant n \leqslant 7$，版本 2 最快，而对于 $n \geqslant 8$，版本 3 最快。

5.3 这是个简单的练习，但是认识到一个 for 循环的 4 个语句(初始化、测试、更新和循环体)执行的次数是不同的很重要。

代码	min	max	incr	square
A.	1	91	90	90
B.	91	1	90	90
C.	1	1	90	90

5.4 这段汇编代码展示了 GCC 发现的一个很聪明的优化机会。要更好地理解代码优化的细微之处，仔细研究这段代码是很值得的。

A. 在没经过优化的代码中，寄存器%xmm0 简单地被用作临时值，每次循环迭代中都会设置和使用。在经过更多优化的代码中，它被使用的方式更像 combine4 中的变量 x，累积向量元素的乘积。不过，与 combine4 的区别在于每次迭代第二条 vmovsd 指令都会更新位置 dest。

我们可以看到，这个优化过的版本运行起来很像下面的 C 代码：