```
1   /* Make sure dest updated on each iteration */
2   void combine3w(vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(v);
6       data_t *data = get_vec_start(v);
7       data_t acc = IDENT;
8
9       /* Initialize in event length <= 0 */
10      *dest = acc;
11
12      for (i = 0; i < length; i++) {
13          acc = acc OP data[i];
14          *dest = acc;
15      }
16  }
```

B. combine3 的两个版本有相同的功能，甚至于相同的内存别名使用。

C. 这个变换可以不改变程序的行为，因为，除了第一次迭代，每次迭代开始时从 dest 读出的值和前一次迭代最后写入到这个寄存器的值是相同的。因此，合并指令可以简单地使用在循环开始时就已经在 %xmm0 中的值。

5.5　多项式求值是解决许多问题的核心技术。例如，多项式函数常常用作对数学库中三角函数求近似值。

A. 这个函数执行 $2n$ 个乘法和 n 个加法。

B. 我们可以看到，这里限制性能的计算是反复地计算表达式 xpwr=x*xpwr。这需要一个浮点数乘法(5 个时钟周期)，并且直到前一次迭代完成，下一次迭代的计算才能开始。两次连续的迭代之间，对 result 的更新只需要一个浮点加法(3 个时钟周期)。

5.6　这道题说明了最小化一个计算中的操作数量不一定会提高它的性能。

A. 这个函数执行 n 个乘法和 n 个加法，是原始函数 poly 中乘法数量的一半。

B. 我们可以看到，这里的性能限制计算是反复地计算表达式 result=a[i]+x*result。从来自上一次迭代的 result 的值开始，我们必须先把它乘以 x(5 个时钟周期)，然后把它加上 a[i](3 个时钟周期)，然后得到本次迭代的值。因此，每次迭代造成了最小延迟时间 8 个周期，正好等于我们测量到的 CPE。

C. 虽然函数 poly 中每次迭代需要两个乘法，而不是一个，但是只有一条乘法是在每次迭代的关键路径上出现。

5.7　下面的代码直接遵循了我们对 k 次展开一个循环所阐述的规则：

```
1   void unroll5(vec_ptr v, data_t *dest)
2   {
3       long i;
4       long length = vec_length(v);
5       long limit = length-4;
6       data_t *data = get_vec_start(v);
7       data_t acc = IDENT;
8
9       /* Combine 5 elements at a time */
10      for (i = 0; i < limit; i+=5) {
11          acc = acc OP data[i]   OP data[i+1];
12          acc = acc OP data[i+2] OP data[i+3];
13          acc = acc OP data[i+4];
14      }
15
16      /* Finish any remaining elements */
17      for (; i < length; i++) {
18          acc = acc OP data[i];
19      }
20      *dest = acc;
21  }
```