累积值。与 combine3 中的循环相比，我们将每次迭代的内存操作从两次读和一次写减少到只需要一次读。

```
    Inner loop of combine4.  data_t = double, OP = *
    acc in %xmm0, data+i in %rdx, data+length in %rax
1   .L25:                              loop:
2     vmulsd  (%rdx), %xmm0, %xmm0      Multiply acc by data[i]
3     addq    $8, %rdx                  Increment data+i
4     cmpq    %rax, %rdx                Compare to data+length
5     jne     .L25                      If !=, goto loop
```

```
1   /* Accumulate result in local variable */
2   void combine4(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       for (i = 0; i < length; i++) {
10          acc = acc OP data[i];
11      }
12      *dest = acc;
13  }
```

图 5-10　把结果累积在临时变量中。将累积值存放在局部变量 acc(累积器(accumulator)的简写)中，消除了每次循环迭代中从内存中读出并将更新值写回的需要

我们看到程序性能有了显著的提高，如下表所示：

函数	方法	整数		浮点数	
		+	*	+	*
combine3	直接数据访问	7.17	9.02	9.02	11.03
combine4	累积在临时变量中	1.27	3.01	3.01	5.01

所有的时间改进范围从 2.2×到 5.7×，整数加法情况的时间下降到了每元素只需 1.27 个时钟周期。

可能又有人会认为编译器应该能够自动将图 5-9 中所示的 combine3 的代码转换为在寄存器中累积那个值，就像图 5-10 中所示的 combine4 的代码所做的那样。然而实际上，由于内存别名使用，两个函数可能会有不同的行为。例如，考虑整数数据，运算为乘法，标识元素为 1 的情况。设 v=[2，3，5]是一个由 3 个元素组成的向量，考虑下面两个函数调用：

```
combine3(v, get_vec_start(v) + 2);
combine4(v, get_vec_start(v) + 2);
```

也就是在向量最后一个元素和存放结果的目标之间创建一个别名。那么，这两个函数的执行如下：

函数	初始值	循环之前	i = 0	i = 1	i = 2	最后
combine3	[2, 3, 5]	[2, 3, 1]	[2, 3, 2]	[2, 3, 6]	[2, 3, 36]	[2, 3, 36]
combine4	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 30]