测试显示，对于整数这个函数的 CPE 等于 1.50，对于浮点数据 CPE 等于 3.00。对于数据类型 double，内循环的 x86-64 汇编代码如下所示：

```
    Inner loop of inner4.  data_t = double, OP = *
    udata in %rbp, vdata in %rax, sum in %xmm0
    i in %rcx, limit in %rbx
1   .L15:                                   loop:
2     vmovsd  0(%rbp,%rcx,8), %xmm1           Get udata[i]
3     vmulsd  (%rax,%rcx,8), %xmm1, %xmm1     Multiply by vdata[i]
4     vaddsd  %xmm1, %xmm0, %xmm0             Add to sum
5     addq    $1, %rcx                        Increment i
6     cmpq    %rbx, %rcx                      Compare i:limit
7     jne     .L15                            If !=, goto loop
```

假设功能单元的特性如图 5-12 所示。

A. 按照图 5-13 和图 5-14 的风格，画出这个指令序列会如何被译码成操作，并给出它们之间的数据相关如何形成一条操作的关键路径。

B. 对于数据类型 double，这条关键路径决定的 CPE 的下界是什么？

C. 假设对于整数代码也有类似的指令序列，对于整数数据的关键路径决定的 CPE 的下界是什么？

D. 请解释虽然乘法操作需要 5 个时钟周期，但是为什么两个浮点版本的 CPE 都是 3.00。

* 5.14　编写习题 5.13 中描述的内积过程的一个版本，使用 6×1 循环展开。对于 x86-64，我们对这个展开的版本的测试得到，对整数数据 CPE 为 1.07，而对两种浮点数据 CPE 仍然为 3.01。

A. 解释为什么在 Intel Core i7 Haswell 上运行的任何(标量)版本的内积过程都不能达到比 1.00 更小的 CPE 了。

B. 解释为什么对浮点数据的性能不会通过循环展开而得到提高。

* 5.15　编写习题 5.13 中描述的内积过程的一个版本，使用 6×6 循环展开。对于 x86-64，我们对这个函数的测试得到对整数数据的 CPE 为 1.06，对浮点数据的 CPE 为 1.01。

什么因素制约了性能达到 CPE 等于 1.00？

* 5.16　编写习题 5.13 中描述的内积过程的一个版本，使用 6×1a 循环展开产生更高的并行性。我们对这个函数的测试得到对整数数据的 CPE 为 1.10，对浮点数据的 CPE 为 1.05。

** 5.17　库函数 memset 的原型如下：

```
void *memset(void *s, int c, size_t n);
```

这个函数将从 s 开始的 n 个字节的内存区域都填充为 c 的低位字节。例如，通过将参数 c 设置为 0，可以用这个函数来对一个内存区域清零，不过用其他值也是可以的。

下面是 memset 最直接的实现：

```
1   /* Basic implementation of memset */
2   void *basic_memset(void *s, int c, size_t n)
3   {
4       size_t cnt = 0;
5       unsigned char *schar = s;
6       while (cnt < n) {
7           *schar++ = (unsigned char) c;
8           cnt++;
9       }
10      return s;
11  }
```

实现该函数一个更有效的版本，使用数据类型为 unsigned long 的字来装下 8 个 c，然后用字级的写遍历目标内存区域。你可能发现增加额外的循环展开会有所帮助。在我们的参考机上，能够把 CPE 从直接实现的 1.00 降低到 0.127。即，程序每个周期可以写 8 个字节。

这里是一些额外的指导原则。在此，假设 K 表示你运行程序的机器上的 sizeof(unsigned long)的值。