CPE 的时候，得到令人吃惊的结果：

函数	方法	整数		浮点数	
		+	*	+	*
combine4	累积在临时变量中	1.27	3.01	3.01	5.01
combine5	2×1 展开	1.01	3.01	3.01	5.01
combine6	2×2 展开	0.81	1.51	1.51	2.51
combine7	2×1a 展开	1.01	1.51	1.51	2.51
延迟界限		1.00	3.00	3.00	5.00
吞吐量界限		0.50	1.00	1.00	0.50

整数加的性能几乎与使用 $k\times1$ 展开的版本(combine5)的性能相同，而其他三种情况则与使用并行累积变量的版本(combine6)相同，是 $k\times1$ 扩展的性能的两倍。这些情况已经突破了延迟界限造成的限制。

图 5-27 说明了 combine7 内循环的代码(对于合并操作为乘法，数据类型为 double 的情况)是如何被译码成操作，以及由此得到的数据相关。我们看到，来自于 vmovsd 和第一个 vmulsd 指令的 load 操作从内存中加载向量元素 i 和 $i+1$，第一个 mul 操作把它们乘起来。然后，第二个 mul 操作把这个结果乘以累积值 acc。图 5-28a 给出了我们如何对图 5-27 的操作进行重新排列、优化和抽象，得到表示一次迭代中数据相关的模板(图 5-28b)。对于 combine5 和 combine7 的模板，有两个 load 和两个 mul 操作，但是只有一个 mul 操作形成了循环寄存器间的数据相关链。然后，把这个模板复制 $n/2$ 次，给出了 n 个向量元素相乘所执行的计算(图 5-29)，我们可以看到关键路径上只有 $n/2$ 个操作。每次迭代内的第一个乘法都不需要等待前一次迭代的累积值就可以执行。因此，最小可能的 CPE 减少了 2 倍。

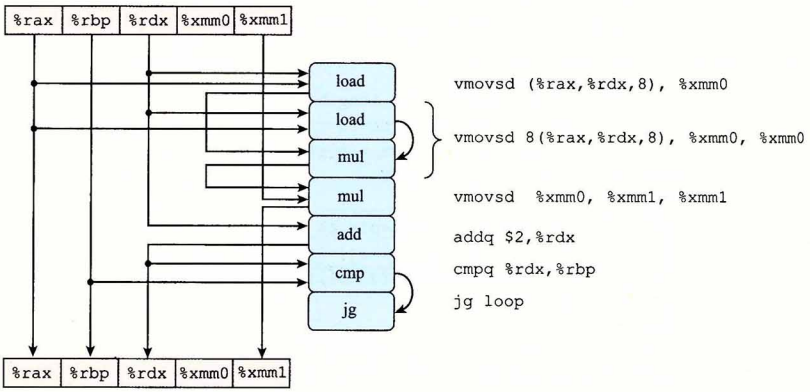

图 5-27　combine7 内循环代码的图形化表示。每次迭代被译码成与 combine5 或 combine6 类似的操作，但是数据相关不同

图 5-30 展示了当数值达到 $k=10$ 时，实现 $k\times1a$ 循环展开并重新结合变换的效果。可以看到，这种变换带来的性能结果与 $k\times k$ 循环展开中保持 k 个累积变量的结果相似。对所有的情况来说，我们都接近了由功能单元造成的吞吐量界限。