```
4       testq   %rdi, %rdi      Test ls
5       jne     .L3             If nonnull, goto loop
```

第 3 行上的 movq 指令是这个循环中关键的瓶颈。后面寄存器%rdi 的每个值都依赖于加载操作的结果，而加载操作又以%rdi 中的值作为它的地址。因此，直到前一次迭代的加载操作完成，下一次迭代的加载操作才能开始。这个函数的 CPE 等于 4.00，是由加载操作的延迟决定的。事实上，这个测试结果与文档中参考机的 L1 级 cache 的 4 周期访问时间是一致的，相关内容将在 6.4 节中讨论。

5.12.2　存储的性能

在迄今为止所有的示例中，我们只分析了大部分内存引用都是加载操作的函数，也就是从内存位置读到寄存器中。与之对应的是*存储*(store)操作，它将一个寄存器值写到内存。这个操作的性能，尤其是与加载操作的相互关系，包括一些很细微的问题。

与加载操作一样，在大多数情况中，存储操作能够在完全流水线化的模式中工作，每个周期开始一条新的存储。例如，考虑图 5-32 中所示的函数，它们将一个长度为 n 的数组 dest 的元素设置为 0。我们测试结果为 CPE 等于 1.00。对于只具有单个存储功能单元的机器，这已经达到了最佳情况。

```
1   /* Set elements of array to 0 */
2   void clear_array(long *dest, long n) {
3       long i;
4       for (i = 0; i < n; i++)
5           dest[i] = 0;
6   }
```

图 5-32　将数组元素设置为 0 的函数。该代码 CPE 达到 1.0

与到目前为止我们已经考虑过的其他操作不同，存储操作并不影响任何寄存器值。因此，就其本性来说，一系列存储操作不会产生数据相关。只有加载操作会受存储操作结果的影响，因为只有加载操作能从由存储操作写的那个位置读回值。图 5-33 所示的函数 write_read 说明了加载和存储操作之间可能的相互影响。这幅图也展示了该函数的两个示例执行，是对两元素数组 a 调用的，该数组的初始内容为－10 和 17，参数 cnt 等于 3。这些执行说明了加载和存储操作的一些细微之处。

在图 5-33 的示例 A 中，参数 src 是一个指向数组元素 a[0]的指针，而 dest 是一个指向数组元素 a[1]的指针。在此种情况中，指针引用*src 的每次加载都会得到值－10。因此，在两次迭代之后，数组元素就会分别保持固定为－10 和－9。从 src 读出的结果不受对 dest 的写的影响。在较大次数的迭代上测试这个示例得到 CPE 等于 1.3。

在图 5-33 的示例 B 中，参数 src 和 dest 都是指向数组元素 a[0]的指针。在这种情况中，指针引用*src 的每次加载都会得到指针引用*dest 的前次执行存储的值。因而，一系列不断增加的值会被存储在这个位置。通常，如果调用函数 write_read 时参数 src 和 dest 指向同一个内存位置，而参数 cnt 的值为 $n>0$，那么净效果是将这个位置设置为 $n-1$。这个示例说明了一个现象，我们称之为*写/读相关*(write/read dependency)——一个内存读的结果依赖于一个最近的内存写。我们的性能测试表明示例 B 的 CPE 为 7.3。写/读相关导致处理速度下降约 6 个时钟周期。