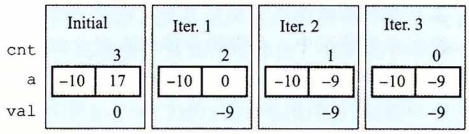
```
1   /* Write to dest, read from src */
2   void write_read(long *src, long *dst, long n)
3   {
4       long cnt = n;
5       long val = 0;
6
7       while (cnt) {
8           *dst = val;
9           val = (*src)+1;
10          cnt--;
11      }
12  }
```

示例A: `write_read(&a[0],&a[1],3)`



示例B: `write_read(&a[0],&a[0],3)`

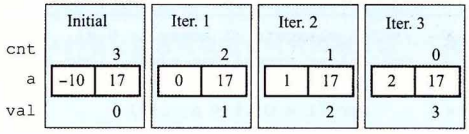


图 5-33　写和读内存位置的代码，以及示例执行。这个函数突出的是当参数 src 和 dest 相等时，存储和加载之间的相互影响

为了了解处理器如何区别这两种情况，以及为什么一种情况比另一种运行得慢，我们必须更加仔细地看看加载和存储执行单元，如图 5-34 所示。存储单元包含一个*存储缓冲区*，它包含已经被发射到存储单元而又还没有完成的存储操作的地址和数据，这里的完成包括更新数据高速缓存。提供这样一个缓冲区，使得一系列存储操作不必等待每个操作都更新高速缓存就能够执行。当一个加载操作发生时，它必须检查存储缓冲区中的条目，看有没有地址相匹配。如果有地址相匹配(意味着在写的字节与在读的字节有相同的地址)，它就取出相应的数据条目作为加载操作的结果。

GCC 生成的 write_read 内循环代码如下：

```
  Inner loop of write_read
  src in %rdi, dst in %rsi, val in %rax
.L3:                          loop:
  movq    %rax, (%rsi)          Write val to dst
  movq    (%rdi), %rax          t = *src
  addq    $1, %rax              val = t+1
  subq    $1, %rdx              cnt--
  jne     .L3                   If != 0, goto loop
```

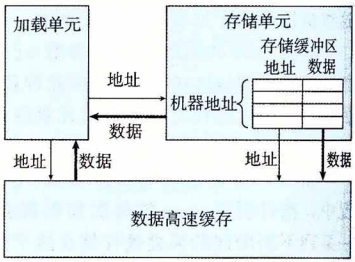


图 5-34　加载和存储单元的细节。存储单元包含一个未执行的写的缓冲区。加载单元必须检查它的地址是否与存储单元中的地址相符，以发现写/读相关

图 5-35 给出了这个循环代码的数据流表示。指令 movq %rax,(%rsi)被翻译成两个操作：s_