9) 调度引发缺页中断的进程，操作系统返回调用它的汇编语言例程。

10) 该例程恢复寄存器和其他状态信息，返回到用户空间继续执行，就好像缺页中断没有发生过一样。

3.6.3 指令备份

当程序访问不在内存中的页面时，引起缺页中断的指令会半途停止并引发操作系统的陷阱。在操作系统取出所需的页面后，它需要重新启动引起陷阱的指令。但这并不是一件容易实现的事。

我们在最坏情形下考察这个问题的实质，考虑一个有双地址指令的CPU，比如Motorola 680x0，这是一种在嵌入式系统中广泛使用的CPU。例如，指令

MOVE.L#6(A1), 2(A0)

为6字节（见图3-28）。为了重启该指令，操作系统要知道该指令第一个字节的位置。在陷阱发生时，程序计数器的值依赖于引起缺页中断的那个操作数以及CPU中微指令的实现方式。

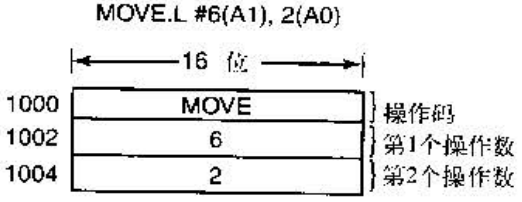

图3-28 引起缺页中断的一条指令

在图3-28中，从地址1000处开始的指令进行了3次内存访问：指令字本身和操作数的2个偏移量。从可以产生缺页中断的这3次内存访问来看，程序计数器可能在1000、1002和1004时发生缺页中断，对操作系统来说要准确地判断指令是从哪儿开始的通常是不可能的。如果发生缺页中断时程序计数器是1002，操作系统无法弄清在1002位置的字是与1000的指令有关的内存地址（比如，一个操作数的位置），还是一个指令的操作码。

这种情况已经很糟糕了，但可能还有更糟的情况。一些680x0体系结构的寻址方式采用自动增量，这也意味着执行这条指令的副作用是会增量一个或多个寄存器。使用自动增量模式也可能引起错误。这依赖于微指令的具体实现，这种增量可能会在内存访问之前完成，此时操作系统必须在重启这条指令前将软件中的寄存器减量。自动增量也可能在内存访问之后完成，此时，它不会在陷入时完成而且不必由操作系统恢复。自动减量也会出现相同的问题。自动增量和自动减量是否在相应访存之前完成随着指令和CPU模式的不同而不同。

幸运的是，在某些计算机上，CPU的设计者们提供了一种解决方法，就是通过使用一个隐藏的内部寄存器。在每条指令执行之前，把程序计数器的内容复制到该寄存器。这些机器可能会有第二个寄存器，用来提供哪些寄存器已经自动增加或者自动减少以及增减的数量等信息。通过这些信息，操作系统可以消除引起缺页中断的指令所造成的所有影响，并使指令可以重新开始执行。如果该信息不可用，那么操作系统就要找出所发生的问题从而设法来修复它。看起来硬件设计者是不能解决这个问题了，于是他们就推给操作系统的设计者来解决这个问题。

3.6.4 锁定内存中的页面

尽管本章对I/O的讨论不多，但计算机有虚拟内存并不意味着I/O不起作用了。虚拟内存和I/O通过微妙的方式相互作用着。设想一个进程刚刚通过系统调用从文件或其他设备中读取数据到其地址空间中的缓冲区。在等待I/O完成时，该进程被挂起，另一个进程被允许运行，而这个进程产生一个缺页中断。

如果分页算法是全局算法，包含I/O缓冲区的页面会有很小的机会（但不是没有）被选中换出内存。如果一个I/O设备正处在对该页面进行DMA传输的过程之中，将这个页面移出将会导致部分数据写入它们所属的缓冲区中，而部分数据被写入到最新装入的页面中。一种解决方法是锁住正在做I/O操作的内存中的页面以保证它不会被移出内存。锁住一个页面通常称为在内存中钉住（pinning）页面。另一种方法是在内核缓冲区中完成所有的I/O操作，然后再将数据复制到用户页面。

3.6.5 后备存储

在前面讨论过的页面置换算法中，我们已经知道了如何选择换出内存的页面。但是却没有讨论当页面被换出时会存放在磁盘上的哪个位置。现在我们讨论一下磁盘管理相关的问题。

在磁盘上分配页面空间的最简单的算法是在磁盘上设置特殊的交换分区，甚至从文件系统划分一块独立的磁盘（以平衡I/O负载）。大多数UNIX是这样处理的。在这个分区里没有普通的文件系统，这样就消除了将文件偏移转换成块地址的开销。取而代之的是，始终使用相应分区的起始块号。