的。如果执行一条指令需要1ns，页表查询必须在0.2ns之内完成，以避免映射成为一个主要瓶颈。

第二个问题来自现代计算机使用至少32位的虚拟地址，而且64位变得越来越普遍。假设页长为4KB，32位的地址空间将有100万页，而64位地址空间简直多到超乎你的想象。如果虚拟地址空间中有100万个页，那么页表必然有100万条表项。另外请记住，每个进程都需要自己的页表（因为它有自己的虚拟地址空间）。

对大而快速的页映射的需求成为了构建计算机的重要约束。最简单的设计（至少从概念上）是使用由一组“快速硬件寄存器”组成的单一页表，每一个表项对应一个虚页，虚页号作为索引，如图3-10所示。当启动一个进程时，操作系统把保存在内存中的进程页表的副本载入到寄存器中。在进程运行过程中，不必再为页表而访问内存。这个方法的优势是简单并且在映射过程中不需要访问内存。而缺点是在页表很大时，代价高昂。而且每一次上下文切换都必须装载整个页表，这样会降低性能。

另一种极端方法是，整个页表都在内存中。那时所需的硬件仅仅是一个指向页表起始位置的寄存器。这样的设计使得在上下文切换时，进行“虚拟地址到物理地址”的映射只需重新装入一个寄存器。当然，这种做法的缺陷是在执行每条指令时，都需要一次或多次内存访问，以完成页表项的读入，速度非常慢。

1. 转换检测缓冲区

现在讨论加速分页机制和处理大的虚拟地址空间的实现方案，先介绍加速分页问题。大多数优化技术都是从内存中的页表开始的。这种设计对效率有着巨大的影响。例如，假设一条指令要把一个寄存器中的数据复制到另一个寄存器。在不分页的情况下，这条指令只访问一次内存，即从内存中取指令。有了分页后，则因为要访问页表而引起更多次的访问内存。由于执行速度通常被CPU从内存中取指令和数据的速度所限制，所以每次内存访问必须进行两次页表访问会降低一半的性能。在这种情况下，没人会采用分页机制。

多年以来，计算机的设计者已经意识到了这个问题，并找到了一种解决方案。这种解决方案的建立基于这样一种现象：大多数程序总是对少量的页面进行多次的访问，而不是相反的。因此，只有很少的页表项会被反复读取，而其他的页表项很少被访问。

上面提到的解决方案是为计算机设置一个小型的硬件设备，将虚拟地址直接映射到物理地址，而不必再访问页表。这种设备称为转换检测缓冲区（Translation Lookaside Buffer，TLB），有时又称为相联存储器（associate memory），如图3-12所示。它通常在MMU中，包含少量的表项，在此例中为8个，在实际中很少会超过64个。每个表项记录了一个页面的相关信息，包括虚拟页号、页面的修改位、保护码（读/写/执行权限）和该页所对应的物理页框。除了虚拟页号（不是必须放在页表中的），这些域与页表中的域是一一对应的。另外还有一位用来记录这个表项是否有效（即是否在使用）。

如果一个进程在虚拟地址19、20和21之间有一个循环，那么可能会生成图3-12中的TLB。因此，这三个表项中有可读和可执行的保护码。当前主要使用的数据（假设是个数组）放在页面129和页面130中。页面140包含了用于数组计算的索引。最后，堆栈位于页面860和页面861。

有效位	虚拟页面号	修改位	保护位	页框号
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

图3-12 TLB加速分页

现在看一下TLB是如何工作的。将一个虚拟地址放入MMU中进行转换时，硬件首先通过将该虚拟页号与TLB中所有表项同时（即并行）进行匹配，判断虚拟页面是否在其中。如果发现了一个有效的匹配并且要进行的访问操作并不违反保护位，则将页框号直接从TLB中取出而不必再访问页表。如果虚拟页面号确实是在TLB中，但指令试图在一个只读页面上进行写操作，则会产生一个保护错误，就像对页表进行非法访问一样。

当虚拟页号不在TLB中时发生的事情值得讨论。如果MMU检测到没有有效的匹配项时，就会进行正常的页表查询。接着从TLB中淘汰一个表项，然后用新找到的页表项代替它。这样，如果这一页面很快再被访问，第二次访问TLB时自然将会命中而不是不命中。当一个表项被清除出TLB时，将修改位复