第 8 章 虚拟存储器

如何在有限容量的内存中运行更大的程序呢？早期程序员的做法是把程序分割成许多的片段存储在次级存储器如磁盘中，运行时首先将第 1 个片段放到内存中执行，执行完成后再调用下一个片段。虽然片段在物理内存中的交换可以由操作系统完成，但是程序员必须先把程序进行分割，这是一个十分费时费力的工作。

有没有什么办法将这个过程完全自动化呢？现代的处理器和操作系统通过软硬件协同实现了被称为虚拟存储器的技术，可以让一个程序虚拟地运行在一个完整的地址空间中。比如对于 32 位的系统，地址空间即为 4GB，64 位的系统则有 $2^{64}B$ 非常巨大的地址空间。虚拟化内存技术对于应用程序是完全透明的，背后的工作都由操作系统和硬件自动完成，因此大大方便了应用程序开发者。

那么虚拟存储器是如何实现的呢？这就要引入虚拟地址的概念。在没有使用虚拟地址的系统中，处理器访问内存时输出的地址会直接送到存储器中。如图8.1。而如果使用了虚拟地址，处理器输出的地址则被认为是虚拟地址，需要先进行地址转换，转换后的地址被称为物理地址。如图8.2。

地址转换需要软硬件共同完成。硬件上，由内存管理单元 (Memory Manage Unit, MMU) 进行地址转换。软件上，则由操作系统负责管理应用程序虚拟地址到物理地址的映射关系。当程序访问的数据不在内存中时，操作系统便会自动将数据从磁盘读取数据到内存并添加新的地址映射关系。而当物理内存空间不足时，操作系统则通过将一部分数据从内存写回到磁盘并解除一部分地址映射来释放内存空间。

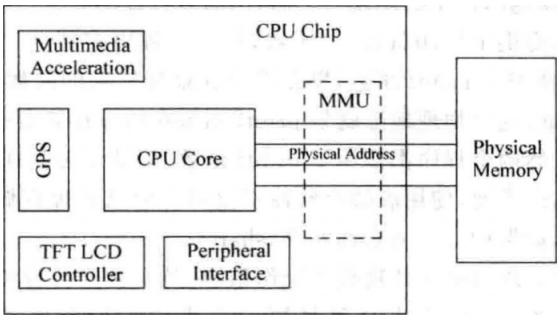

图 8.1: 使用物理地址系统

虚拟存储器不仅解决了内存容量的问题，还带来了非常多的好处，如：

- 地址空间隔离。没有使用虚拟地址时，运行多个程序需要为每个程序分配地址空间，每个程序都只能在这个空间内运行。这便极大地限制了程序的编写。而使用了虚拟地址后，每个程序都认为自己独占整个虚拟地址空间，程序间运行互不干扰。地址空间相互隔离。
- 共享。如操作系统内核提供了 printf 函数，不同程序通过不同的地址进行调用。那么操作系统便可以将这些虚拟地址都映射到同一个物理地址上，从而实现了程序的共享。类似的还可以通过这种方式实现共享内存，进行进程间通信 (Inter-Process

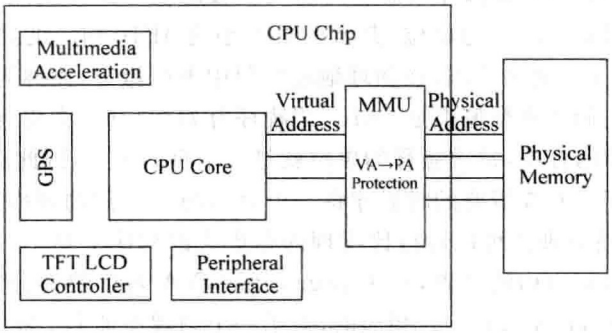

图 8.2: 使用虚拟地址系统

Communication, IPC)。

- 保护。因为进程虚拟地址到物理地址的映射是由操作系统来确定的。因此操作系统便可以通过控制映射关系来保护自己的数据不被恶意程序攻击。进一步也可保护其它进程的数据不被非法访问，或者对不同地址空间设置不同的权限，达到更细化的保护作用。

总之，虚拟存储器是现代操作系统和体系结构非常重要的内容。本部分主要从硬件层面来讲述虚拟存储器的内容。涉及页表，TLB，MIPS 指令集 TLB 异常处理等内容。

8.1 虚拟地址转换

根据上节的介绍，我们知道，实现虚拟存储器就需要实现虚拟地址到物理地址的转换。而实现地址转换的方式可以分为分段和分页两种。这也对应分段式和分页式这两种内存管理的方式。

分段是早期采用的方式。基本思想是把一个程序占用的一段地址空间看成一个段，用起始地址和段长来表示。因此程序运行的地址便可以由段号和段内偏移来确定。只要我们将段的起始地址映射到物理内存的不同位置，便可以完成地址映射。x86 架构中便使用段寄存器和段描述符表等结构来支持分段式内存管理。分段可以实现虚拟存储器的部分功能，但是分段也有一些不足。操作系统每次都需要为一个进程开辟一段连续的内存空间。在进程不断的创建和释放之后。物理内存段之间的空隙便会很多，而这些空隙无法被使用，带来了内存的浪费。这被称为外部碎片问题。

因此人们又发明了**分页**的方法。基本思想是把虚拟地址空间和物理地址空间划分成一个个大小相等的页，如 4KB。操作系统可以控制每一个虚拟页到物理页的映射。因此地址的映射不再是连续的，而是离散的了（其实都是离散的，这里是从控制的粒度的相对大小来说的）。这样的话，物理内存的每一个页都可以被充分利用，而不会带来外部碎片的问题。

而为了存储这种映射关系，便需要被称为页表的数据结构。

8.1.1 页表

对于一个虚拟地址 VA[31:0] 来说，VA[11:0]（假设页大小为 4KB）用来表示页内偏移，称为 page offset。剩余的 VA[31:12] 用于表示位于哪个页，称为 VPN(Virtual Page Numeber)。同样的，对于物理地址 PA 来说，PA[11:0] 表示页内偏移，PA 剩余部分表示位于哪个页。不过由于历史原因，物理页不叫页而叫做帧 (frame)。因此物理地址中对应 VPN 的部分被称为物理帧号 PFN(Physical Frame Number)。注意物理地址并非一定是 32 位的，这要看实际的内存大小。

地址映射时页内偏移是保持不变的，因此重点便在于将虚拟页号 VPN 映射到物理页号 PFN。使用虚拟地址的系统中，一般使用一个表格来存储虚拟页号到物理页号的映射规则，这个表格被称为页表。页表是存放在内存中的。处理器中一般会有一个特殊的寄存器，用来存放当前进程页表的内存起始地址（物理地址）。这个寄存器被称为页表寄存器 (Page Table Register, PTR)。

类似于 cache 的直接映射结构，页表是一个完全直接映射结构。页表存储了所有 VPN 到 FPN 的映射。当地址为 32 位，页大小为 4KB 时，有 1M 个页。因此页表有 1M 个页表项。可以通过 VPN 对页表进行寻址 (索引)，来获得对应的页表项。如图8.3所示，注意图中物理地址为 30 位，表示物理内存为 1GB。图中的 valid 位作用在下一节 PageFault 中讲解。

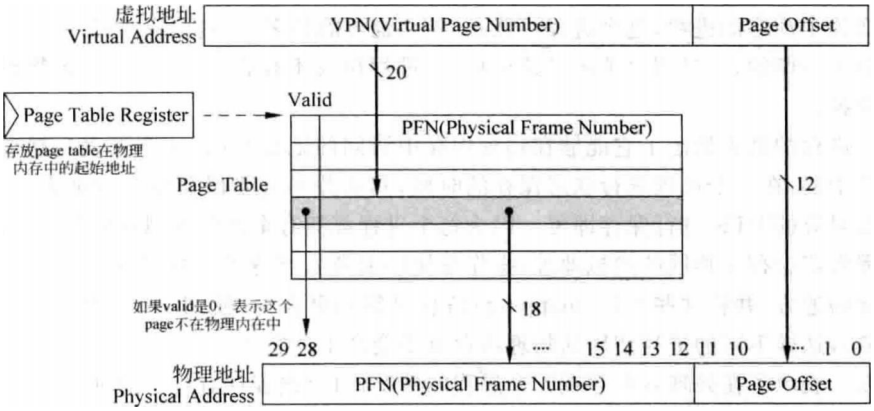

图 8.3: 使用页表进行虚拟地址翻译

从图8.3中来看，似乎一个页表项只需要存储 valid 和 PFN 一共 19 位。但事实上，因为物理内存中数据位宽为 32 位，因此页表项也为 32 位。剩余的位用于存储一些其它信息，如页的属性（是否可读或可写）。因此按目前的讲述，一个页表的大小为 1M*4B = 4MB。

8.1.2 PageFault

到目前为止，我们已经知道如何通过页表查找 VPN 对应的 FPN。但是如果虚拟页对应的物理页还没有被加载到内存中，在页表中该如何表示呢？更进一步，当进程访问一个没有被加载到物理内存的页时，软硬件该怎么处理呢？

第一个问题很简单，类似于 cache，在页表项中我们使用一个 valid 位来表示该虚拟

页是否被加载到内存中。对于第二个问题，我们知道最终需要将该虚拟页从磁盘加载到内存，并在页表中设置映射关系。问题是由谁来执行这些操作呢？很自然，我们使用操作系统来完成这些操作。而为了从进程切换到操作系统执行，硬件需要抛出一个异常。这个异常便被称为缺页异常 (page fault)。

使用虚拟存储器后，物理内存可以看作磁盘的 cache。因为一个程序的某些内容既存在内存中，又存在磁盘中。当存在于内存中时，便可以很快获得数据。而如果不存在于内存中，则会触发 page fault，需要花费毫秒级的时间读取磁盘 (数百万个 CPU 周期)。

既然是一个 Cache，我们便要**考虑写入**时候采用写通 (write through) 还是写回 (write back) 策略。事实上，写通只可能在 L1 和 L2 Cache 之间使用，因为 L2 Cache 的访问时间处于一个可接受的时间范围内，并且可以降低 Cache 一致性管理的难度。越处于存储系统层次的下层，访问时间越长，因此只有写回是可以接受的方法。

因此，类似于 cache，我们在页表项中用 dirty 位表示该页是否被修改。当发生 page fault 时，假如物理内存空间不足，操作系统便需要寻找一个物理页被替换。而如果这个被替换的页是 dirty 的，那么便需要先将其写回磁盘。

当发生 page fault 且物理空间不足时，操作系统需要寻找一个物理页进行替换，因此这就需要操作系统实现一个**替换算法**。Cache 中我们了解过典型的 LRU 算法。但是 LRU 算法需要跟踪最近被使用过的物理页，这对于操作系统来说代价很大甚至是不可能的。因此便需要在硬件上提供支持。可以在页表项中添加一位，用来记录最近是否被访问过。这一位被称为“使用位 (use)”，当每一个页被访问时，硬件将其置为 1。操作系统周期性地将这一位置位 0，因此过一段时间去查看它时便可以知道这一段时间该页是否被访问过。这种方法是近似的 LRU 算法，被大多数操作系统使用。

因此，目前页表的结构如图8.4。

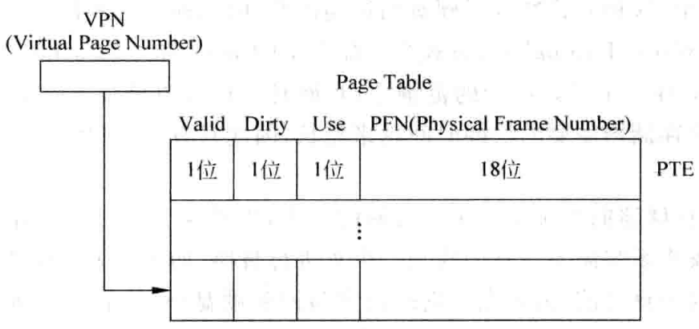

图 8.4: 含 use 位的页表结构

小结

我们目前讲述了在虚拟存储器的系统中，如何将虚拟地址转换成为物理地址，并介绍了 Page Fault 发生时如何进行处理。在处理器中，有一个模块专门负责虚实地址转换，同时处理 Page Fault，这个模块就是之前介绍的内存管理单元 (MMU)。所有支持虚拟存储器的处理器中都会有 MMU 这个模块，下面对目前为止讲述的内容进行一个小结。

1. 当没有 Page Fault 发生时，整个过程如图8.5所示。

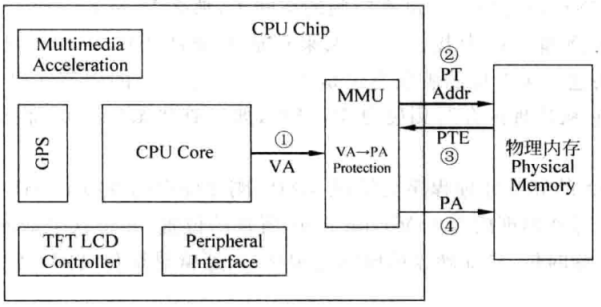

图 8.5: 没有 page fault 时的访问过程

(1) 处理器送出的虚拟地址 VA 首先送到 MMU 中。

(2) MMU 使用页表的基址寄存器 PTR 和 VPN 组成访问页表的地址，这个地址被送到物理内存中。

(3) 物理内存将页表中被寻址到的 PTE(页表项) 返回给 MMU。

(4) MMU 判断 PTE 中的 valid 位，发现其为 1。因此对应的页存在物理内存中。因此使用 PTE 中的 PFN 部分和 VA 的 offset 部分，组成物理地址 (即 PFN, VA[11:0) 并访问内存，得到最终的数据。

2. 当 Page Fault 发生时，整个过程如图8.6所示。

(1-3) 1-3 步骤和上面的过程是一样的。MMU 获得到 PTE。

(4) MMU 判断 PTE 中的 valid 位，发现其为 0。此时 MMU 触发一个 Page Fault 异常发送给处理器，这使得处理器跳转到操作系统 Page Fault 的异常处理程序中。在这一步，MMU 还会把发生 Page Fault 的虚拟地址 VA 页保存到一个专用的寄存器中，供异常处理程序使用。

(5) 假设此时物理内存中已经没有空闲的空间了，那么异常处理程序需要根据替换算法，从物理内存中找到一个不常用的页，将其替换。如过该页的 dirty 位为 1，则还需要将其写回到磁盘中。

(6) Page Fault 的异常处理程序根据 MMU 保存的 VA 来寻址硬盘，找到对应的页，将其写到替换页所在的位置，并将这个新的映射关系写到页表中。因为读取磁盘的时间是最长的，因此这一步花费的时间也是最长的。

(7) 从 Page Fault 的异常处理程序返回时，引起 Page Fault 的指令会被重新取到流水线中执行。因为需要的页已经被放到物理内存了，因此这次访问肯定不会引发 Page Fault，会按照上面的过程进行处理。

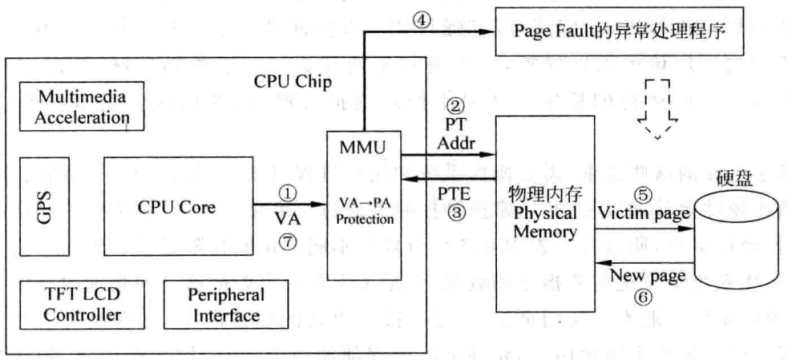

图 8.6: 发生 page fault 时的访问过程

8.1.3 多级页表

前面已经计算过，地址为 32 位且页大小为 4KB 时，一个页表的大小是 4MB。也就是说，对于每个进程都需要在物理内存中为其连续分配 4MB 的空间。而典型的操作系统一般会同时运行上百个进程。因此这种方案会使用大量宝贵的内存空间用于存储页表。而对于 64 位的系统来说，仍旧可能使用 4KB 大小的页。此时一个页表的大小为 $2^{64-12} * 4B = 16PB$，这种方案显然是不可能的。

事实上，一个程序很难用完整个 4GB 的虚拟存储器空间，大部分程序只是用了很少一部分，这就造成了页表中大部分内容其实都是空的，并没有被实际地使用，这样整个页表的利用效率其实是很低的。可以采用很多方法来减少一个进程的页表对于存储空间的需求，最常用的方法是多级页表 (Hierarchical Page Table), 这种方法可以减少页表对于物理存储空间的占用，而且非常容易使用硬件来实现，与之对应的，之前讲述的页表就称为单级页表 (Single Page Table) 也被称为线性页表 (Linear Page Table)。

在多级页表页表的设计中，将原本的单级页表划分成更小的子页表。在执行进程时，不需要一下子将整个页表都放入内存，而是根据需求逐步地放入。并且相邻子页表可以放入内存中的不同位置，然后通过另一个表来存储每个子页表在物理内存中的其实地址。称这个表格为第一级页表，而那些子页表则成为第二级页表。在两级页表的结构中，第一级页表也被叫为页目录表 (Page Directory Table)，第二级页表则简称为页表 (Page Table)。本节主要以二级页表为例，讲解多级页表。

如图8.7显示了使用二级页表进行地址转换的过程。

可以计算出一个页目录表项对应 4MB 的地址空间（二级页表有 1K 项，而一个二级页表项对应 4KB 的空间）。因此二级页表既可以在 4KB 和 4MB 的两种尺度上管理内存。当操作系统创建一个进程时，会先在内存中创建页目录表，并将页目录表的起始地址放到 PTR 寄存器中。通常这个寄存器都是处理器中的一个特殊寄存器，如 x86 中的 CR3 寄存器，ARM 中的 TTB 寄存器。随着进程的执行，操作系统会逐步在物理内存中创建二级页表。每次创建一个二级页表时，操作系统就会将它的起始地址放到对应的页目录表项中。

多级页表还有一个优点，那就是它容易扩展，当处理器的位数增加时，可以通过增加级数的方式来减少页表对于物理内存的占用。

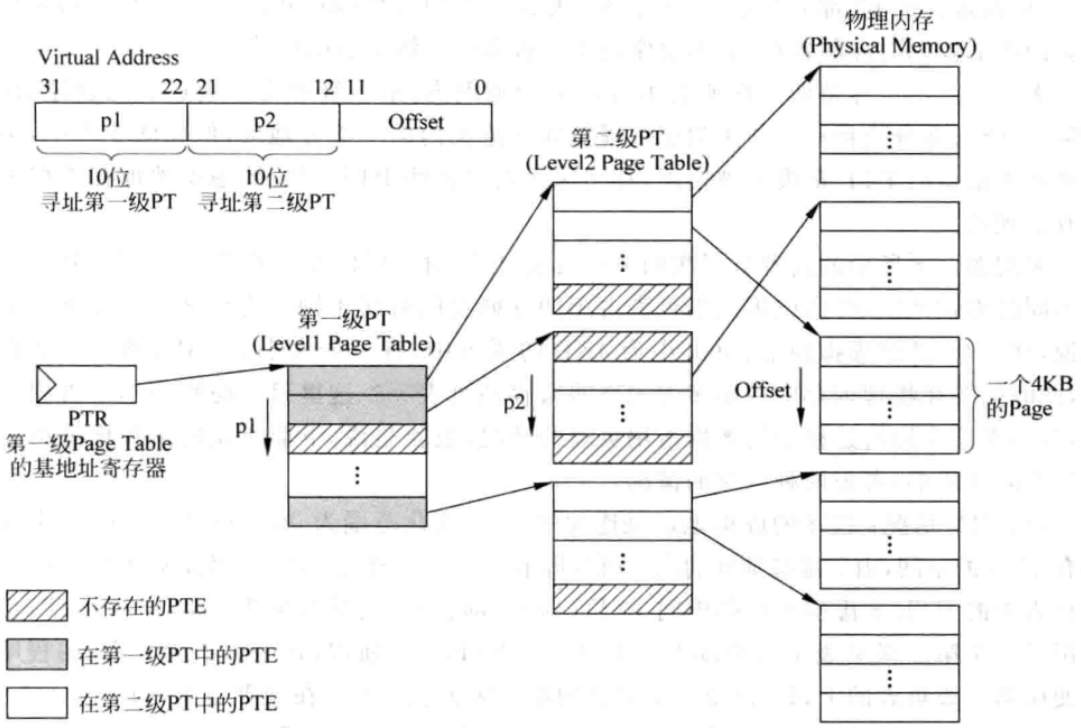

图 8.7: 二级页表进行地址转换

多级页表的结构上是比较简单的，容易使用硬件实现页表的查找，因此在很多硬件实现 Page Table Walk 的处理其中，都采用了这种结构，如 ARM、x86 和 PowerPC 等。所谓的 Page Table Walk 是指当发生 TLB 缺失时，需要从页表中找到对应的映射关系并将其写回到 TLB 的过程。关于什么是 TLB 以及 TLB 缺失会在下一章节介绍。

8.2 TLB

到目前为止[1]，对于两级页表的设计，需要访问两次物理内存才可以得到虚拟地址。而物理内存的运行速度相对于处理器本身来说，有几十倍的差距。因此在处理器执行的时候，如果每次送出虚拟地址都需要经过 3 次访存才能获得需要的数据的话，这显然是很慢的 (要知道，每次取指令都需要访问存储器)。此时可以借鉴 Cache 的设计理念，使用一个速度比较快的缓存，将页表中最近使用的 PTE 缓存下来，因为它们在以后还可能继续使用。尤其是对于取指令来说，考虑到程序本身的串行性，会顺序地从一个页内取指令，此时将 PTE 缓存起来是大有益处的，能够加快一个页内 4KB 内容的转换速度。

由于历史的原因，缓存 PTE 的部件一般不称为 Cache，而是称之为 TLB(Translation Lookaside Buffer)，在 TLB 中存储了页表中最近被使用过的 PTE，从本质上来将，TLB 就是页表的 Cache。但是，TLB 又不同于一般的 Cache，它只有时间相关性 (Temporal Locality)，也就是说，现在访问的页，很有可能在以后继续被访问，至于空间相关性 (Spatial Locality)，TLB 并没有明显的规律，因为在一个页内有很多的情况，都可能使程序跳转到其他不相邻的页中取指令或数据，也就是说，虽然当前在访问一个页，但是未必会访问它相邻的页，正因为如此，Cache 设计中很多的优化方法，例如预取 (prefetching)，是没有办法应用于 TLB 中的。

8.2.1 TLB 结构

既然 TLB 本质上是 Cache，那么就有以前讲过的三种组织方法，直接相连 (direct-mapped)、组相连 (set-associative) 和全相连 (fully-associative)，一般为了减少 TLB 缺失 (miss) 发生的频率，会使用全相连的方式来设计 TLB，但是这样导致 TLB 的容量不能太大，因此也有一些设计中采用了组相连的方式来实现容量比较大的 TLB。容量过小的 TLB 会影响处理器的性能，因此在现代的处理器中，很多都采用两级 TLB，第一级 TLB 采用哈佛结构，分为指令 TLB(I-TLB) 和数据 TLB(D-TLB)，一般采用全相连的方式。第二级 TLB 是指令和数据共用，一般采用组相连的方式，这种设计方法和多级 Cache 是一样的。

图8.8显示了 TLB 内容。因为 TLB 采用了全相连的方式，所以相比页表，多了一个 Tag 的项，它保存了虚拟地址的 VPN，用来对 TLB 进行匹配查找，TLB 中其他的项完全来自于页表，每当发生 TLB 缺失时，将 PTE 从页表中搬移到 TLB 内，TLB 中每一项的内容在上面已经介绍过，这里不再过多赘述。

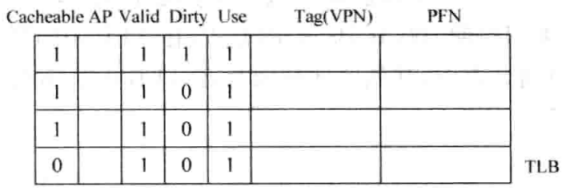

图 8.8: TLB 内容

[1]本节内容大多节选自《超标量处理器设计》，姚永斌著，清华大学出版社

8.2.2 TLB 的写入

涉及到 TLB 写入的知识比较复杂，比如同样面临着写透和写回策略的选择，容易造成混淆。由于我们主要介绍 MIPS 中的 TLB，因此我们可以做一些简化。上节中图8.8显示了 TLB 内容，事实上我们现在可以忽略掉其中的 dirty 位和 use 位，认为 TLB 表项中只包含 VPN，PFN，valid 位和页的属性信息（如是否可读可写等）。因此执行 load/store 类指令不会对 TLB 内容进行修改。也**不会造成 TLB 和页表不一致的问题**。之后们会介绍 MIPS 中如何模拟实现 dirty 位。对一般情况感兴趣的同学，可以参考《超标量处理器设计》3.4.1 节，姚永斌著。

8.2.3 TLB 缺失

当一个虚拟地址查找 TLB，发现需要的内容不在其中时，就发生了 TLB 缺失（miss），由于 TLB 本身的容量很小，所以 TLB 缺失发生的频率还是比较高的。

解决 TLB 缺失的本质就是要从页表中找到对应的映射关系，并将其写回到 TLB 内，这个过程称为 Page Table Walk，可以使用硬件的状态机来完成这个事情，也可以使用软件来做这个事情。它们各有优缺点，在现代的处理器中均有采用，它们各自的工作过程如下。

（1）软件实现 Page Table Walk，软件实现可以保持最大的灵活性，但是一般也需要硬件的配合，这样可以减少软件工作量，一旦发现 TLB 缺失，硬件把产生 TLB 缺失的虚拟地址保存到一个特殊寄存器中，同时产生一个 TLB 缺失类型的异常，在异常处理程序中，软件使用保存在特殊寄存器当中的虚拟地址去寻址物理内存中的页表，找到对应的 PTE，并写回到 TLB 中。很显然，处理器需要支持直接操作 TLB 的指令，如写 TLB、读 TLB 等。对于超标量处理器来说，由于对异常进行处理时，会将流水线中所有的指令进行抹掉，这样会产生一些性能上的损失，但是使用软件方式，可以实现一些比较灵活的 TLB 替换算法，MIPS 和 Alpha 处理器一般采用这种方法处理 TLB 缺失。但是，为了防止在执行 TLB 缺失的异常处理程序时再次发生 TLB 缺失，一般都将这段程序放到一个不需要进行地址转换的区域（这个异常处理程序一般属于操作系统的一部分，而操作系统就放在不需要地址转换的区域），这样处理器在执行这段异常处理程序时，相当于直接使用物理地址来取指令和数据，避免了再次发生 TLB 缺失的情况。

（2）硬件实现 Page Table Walk，硬件实现一般由内存管理单元（MMU）完成。当发现 TLB 缺失时，MMU 自动使用当前的虚拟地址去寻址物理内存中的页表。前面说过，多级页表的最大优点就是容易使用硬件进行查找，只需要使用一个状态机，逐级进行查找就可以了。如果从页表中找到的 PTE 是有效的，那么就将它写回到 TLB 中。这个过程全部都是由硬件自动完成的，软件不需要做任何事情，也就是这个过程对于软件是完全透明的。当然，如果 MMU 发现查找到的 PTE 是无效的，那么硬件就无能为力了，此时 MMU 会产生 PageFault 类型的异常，由操作系统来处理这个情况。使用硬件处理 TLB 缺失的这种方法更适合超标量处理器，它不需要打断流水线，因此从理论上来说，性能也会好一些，但是这需要操作系统保证页表已经在物理内存中建立好了，并且操作系统也

需要将页表的基地址预先写到处理器内部对应的寄存器中（例如 PTR 寄存器），这样才能够保证硬件可以正确地寻址页表，ARM、PowerPC 和 x86 处理器都采用了这种方法。

当发生 TLB 缺失时，如果所需要的 PTE 在页表中，则 TLB 缺失的处理时间大约需要十几个周期。如果由于 PTE 不在页表中而发生 Page Fault，则处理时间就需要成百上千个周期了，此时不管采用硬件处理还是软件处理 TLB 缺失，都不会有明显的差别。TLB 缺失发生的频率对于处理器性能的影响是很大的，在典型的页大小为 4KB 的系统中，只要此时运行的指令或者数据在 4KB 的边界之内，就不会发生 TLB 缺失，一般普通的串行程序都会满足这个规律。当然，TLB 缺失发生的频率还取决于 TLB 的大小以及关联度，还有页的大，小等因素。一旦由 TLB 缺失转变成了 Page Fault，所需要的处理时间就取决于页的替换算法，以及被替换的页是否是脏状态等因素了。

关于 MIPS 如何对 TLB 缺失进行处理，在后面章节会有更详细的介绍。

替换算法

对于组相连（set-associative）或全相连（fully-associative）结构的 TLB，当一个新的 PTE 被写到 TLB 中时，如果当前 TLB 中没有空闲的位置了，那么就要考虑将其中的一个表项（entry）进行替换。理论上说，Cache 中使用的替换方法在 TLB 这里都可以使用，例如最近最少使用算法（Least Recently Used，LRU），但是实际上对于 TLB 来说，随机替换算法（Random）是一种比较合适的方法，当然，在实际的设计中很难实现严格的随机，此时仍然可以采用一种称为时钟算法（Clock Algorithm）的方法来实现近似的随机。它的工作原理本质上就是一个计数器，这个计数器一直在运转，例如每周期加 1，计数器的宽度由 TLB 中表项的个数来决定，例如一个全相连的 TLB 中，表项的个数是 128 个，则计数器的宽度需要 7 位，当 TLB 中的内容需要被替换时，就会访问这个计数器，使用计数器当前的值作为被替换表项的编号，这样就近似地实现了一种随机的替换，这种方法从理论上来说，可能并不能获得最优化的结果，但是它不需要复杂的硬件，也能够获得很好的性能，因此综合看起来是一种不错的折中方法。

8.3 MIPS 虚拟存储器的支持

前面的章节讲述了虚拟存储器的一般设计，本节则讲解 MIPS 体系结构是如何对虚拟存储器提供支持的。

8.3.1 MIPS 虚拟地址空间

在讲解 MIPS 存储管理的机制前，首先介绍 MIPS 的虚拟地址空间布局。以 32 位 MIPS 处理器为例，4GB 地址空间的划分如图8.9所示。

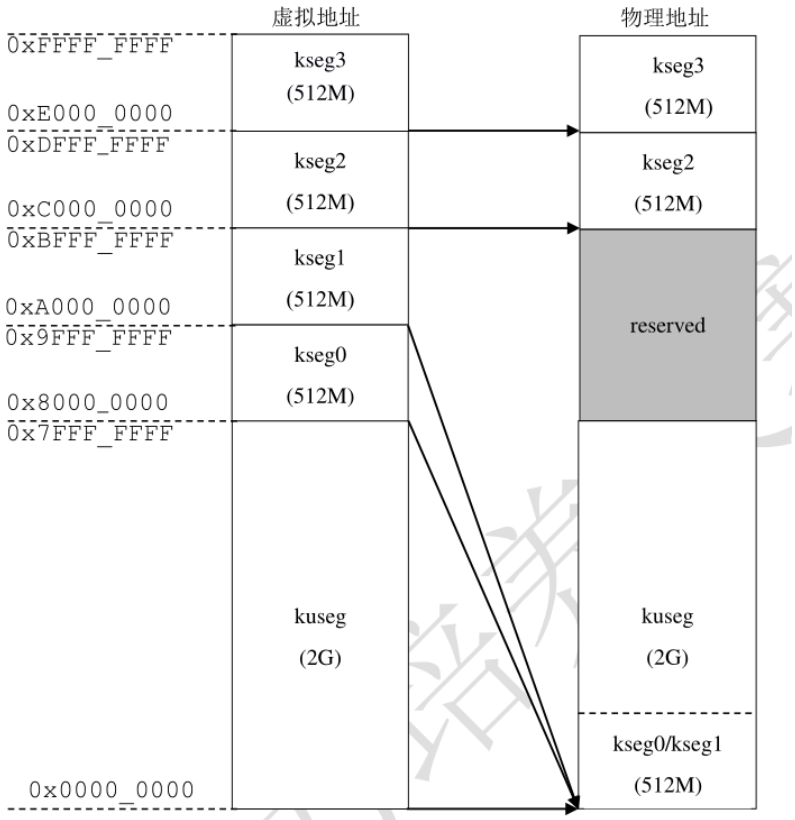

图 8.9: MIPS 虚拟地址空间映射图

32 位 MIPS 处理器将地址空间划分为 5 段，从低地址到高地址，各段的特性如下：

useg 用户模式、监管模式和核心模式都可访问，用户模式下只能访问该段（即低 2GB 空间）。该段地址需要通过 MMU 进行地址映射，其 Cache 算法（是否经过 Cache 缓存、Cache 一致性算法等）由页表中相应字段规定。在 Linux 操作系统中，该段存放用户程序、动态链接库、程序堆栈等。

kseg0 只能由核心模式访问，该段地址不通过 MMU 进行地址映射，其 Cache 算法由 Config 控制寄存器的 K0 域决定。kseg0 直接映射至低 512MB 物理地址空间，映射方法为抹去最高 3 位。在 Linux 操作系统中，该段存放内核代码、数据、异常处理入口。

kseg1 只能由核心模式访问，该段地址不通过 MMU 进行地址映射，也不通过 Cache 进行缓存。ksegl 同样映射至低 512MB 物理地址空间，映射方法为抹去高 3 位。ksegl

和 kseg0 映射到相同的物理地址，但对 kseg1 的访问无须依赖 Cache，因此 MIPS 处理器程序计数器 (PC) 的复位值 0xbfc00000 就落在 ksegl 段。完成 Cache 初始化等工作后，再跳转到 kseg0 执行，可以利用 Cache 提高访问速度。在 Linux 操作系统中，该段存放启动 ROM 和 IO 寄存器等。

kseg2/3 kseg2 可由核心模式和监管模式访问，kseg3 只能由核心模式访问，这两段通过 MMU 进行地址映射，其 Cache 算法由页表项决定。在 Linux 操作系统中，该段存放动态分配的内核数据。

PageMask

在很多处理器中，还支持容量更大的页，因为随着程序越来越大，4KB 大小的页已经不能够满足要求了，对于一个有着 128 个表项 (entry) 的 I-TLB 来说，只能映射到 128*4KB=512KB 大小的程序，这对于当代的程序来说显然是不够的，因此需要使用容量更大的页，例如 1MB 或是 4MB 大小的页，这样可以使 TLB 映射到更大的范围，避免频繁地对 TLB 进行替换。当然，更大的页也是存在缺点的，对于很多程序来说，如果它利用不到这么大的页，那么就会造成一个页内的很多空间被浪费了，这种现象称为页内的碎片 (Page Fragment)，它降低了页的利用效率，而且，每次发生 Page Fault 时，更大的页也就意味着要搬移更多的数据，需要更长的时间才能将这样大的页从下级存储器 (如硬盘) 搬移到物理内存中，这样使 Page Fault 的处理时间变得更长了。

为了解决这种矛盾，在现代的处理器中都支持大小可变的页，由操作系统进行管理，根据不同应用的特点选用不同的大小的页，这样可以最大限度地利用 TLB 中有限的空间，同时又不至于在页内产生过多的碎片。为了支持这种特性，在 TLB 中需要相应的位进行管理，举例来说，在 MIPS 处理器的 TLB 中，有一个 12 位的 Pagemask 项，它用来指示当前被映射的页的大小，如表8.1所示。

表 8.1: MIPS 使用 PageMask 来指定页大小

Page Mask	Page Size	Page Mask	Page Size
0000_0000_0000	4KB	0000_1111_1111	1MB
0000_0000_0011	16KB	0011_1111_1111	4MB
0000_0000_1111	64KB	1111_1111_1111	16MB
0000_0011_1111	256KB		

采用不同大小的页，在寻址 TLB 时，进行的地址比较也是不同的，例如，当采用 1MB 大小的页时，只需要将 VA[31：20] 作为 Tag，参与地址比较就可以了，虚拟地址剩余的 20 位将用来寻址页的内部。不仅如此，在后文中还会讲到，对 TLB 的寻址还受到其他内容的影响（例如 ASID 和 Global 位），这些都是在真实的处理器中需要考虑的内容。

8.3.2 ASID

前面我们已经了解到，使用虚拟存储器的一个好处就是使得同时运行多个进程更加容易。但是我们马上就会意识到，在使用 TLB 后，因为 TLB 硬件资源是所有进程共用

的，因此不同进程间的 TLB 表项就会相互干扰。一种直接的做法是每次切换进程时都清空 TLB 表项，但这种做法显然效率很低下。一种更加通用的方式是给每个进程分配一个数，称为地址空间 ID 或 ASID(Address Space Identifiers)。TLB 表项中也存储一个该页对应的 ASID。查找 TLB 时则需要当前进程的 ASID 和 TLB 表项中的 ASID 匹配才算命中。这样 TLB 便可以同时容纳不同进程的地址映射关系而不会搞混。当然，在某些情况下，不同进程需要共享一个页。此时，我们可以在 TLB 中引入一个表示全局的 G 位。当 G 位为 1 时，TLB 不需要进行 ASID 比较。

到目前为止，我们的虚拟地址转换系统如图8.10。

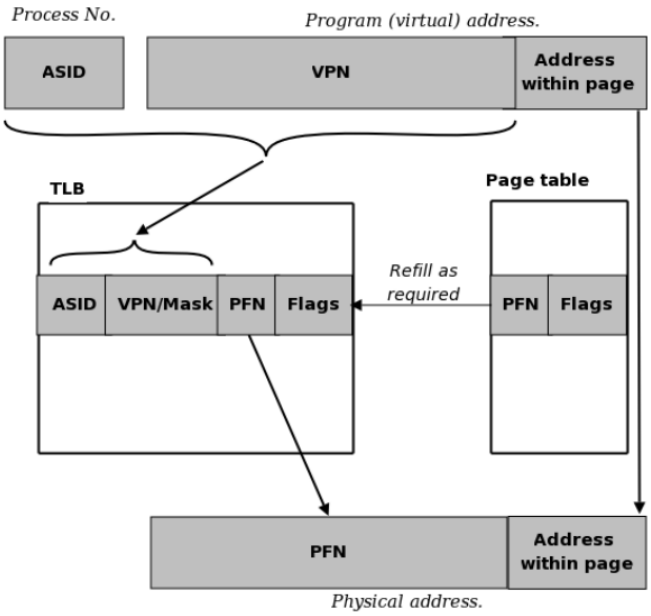

图 8.10: 使用 TLB 和 ASID 的地址转换系统

8.3.3 MIPS TLB 组织

前面我们已经讲过 TLB 是一个全相联的结构，使用相联存储器 (content address memory, CAM) 存储。因此，我们可以将每个 TLB 表项从逻辑上看成两个部分。第一部分作为输入，用于进行命中比较，包含 VPN2、ASID、G bit 和 PageMask 域。第二部分作为输出，主要包含 PFN、valid 位和一些页的属性信息。图8.11显示了 MIPS32 下的 TLB 表项结构。

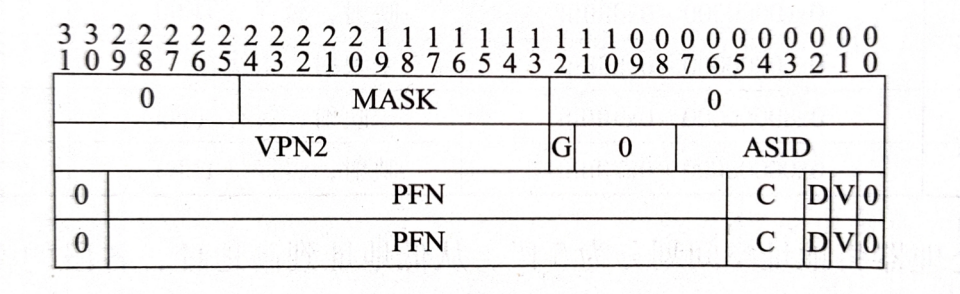

图 8.11: MIPS32 指令系统 TLB 表项结构

你可能注意到图8.11中，包含两份 PFN。这是因为 MIPS 以及当代大多数架构都会一次映射两个连续的 VPN 到两个相互独立的物理地址。这样使得只需要增加少量存储，便可以使得 TLB 可以映射的存储空间加倍。

下面更具体地解释 TLB 每个域的作用：

- VPN2：虚拟页号。VPN2 中的“2”强调每个 VPN2 映射 2 个 PFN(VPN 映射一个 FPN)。虚拟地址的第 12 位即 VA[12] 用于选择 2 个 PFN 中的一个 (奇偶项)。
- PageMask：控制虚拟地址的哪些位用于进行比较。“1”位表示忽略相应的地址位。被忽略位中的最高有效位用来选择奇偶项。
- ASID：标记进程地址空间。只有和当前进程的 ASID（存储在 EntryHi 寄存器中）相同时才参与匹配。ASID 有 8 位：熟悉操作系统的读者可能会意识到 256 作为同时活动的进程的上限对于 Unix 这样的大型系统太小了。但是，只要把同时“活动”理解为“可能在 TLB 中有地址转换项”，256 还算是一个合理的上限。操作系统软件必要时不得不回收 ASID，这将涉及到对 TLB 一些表项进行大清洗。这可是件吃力不讨好的活，但是吃力不讨好的活操作系统不得不干的多了去。256 项应该足以保证 TLB 清洗的次数不会多到造成性能问题。
- G 位：G 位如果为 1 则关闭 ASID 匹配，使得该 TLB 项用于所有的地址空间。
- PFN：物理帧号。
- D 位：写控制位，设为 1 表示允许对该页进行写入。"D" 来源于该位被称为"dirty" 位，下节会详细解释。
- V 位：若该位为零则该表示该项不可用。
- C 域：3 位。主要用于控制是否经过 Cache。如果不经过 Cache，则读写都需要直接访问内存。

模拟"Dirty" 位

我们之前也介绍过页表中包含一个 dirty 位，用于跟踪该页是否被修改过。如果没有没修改过，那么当该页因为 Page fault 且物理空间不足而需要被替换时，就不用写回磁盘。这通常是 CISC CPU 的做法[2]。MIPS CPU 即使在 TLB 中也不支持这个特性。MIPS 体系结构下页表中 D 位和 (TLB 表项中的 D 位) 只用来标识一个页是否可写。所以采用了如下的技巧：

- 当可写的页首次加载进内存时，将其页表项的 D 位清零（表示只读）。
- 当试图对该页写入时，就会发生异常。系统软件将认出这是一个合法的写操作，并利用这个异常将其 D 位置 1（这样 D 位便被用来表示了"Dirty"）。
- 你也要设置 TLB 项中的 D 位，使得之后的写操作可以进行。

MIPS 这么做的好处是不必考虑每次执行 store 类指令时需要将 TLB 中的 dirty 位置 1（原来的做法），因此也不必考虑其带来的 TLB 和页表的不一致问题。

[2]本节参考《see MIPS run Linux》14.4.7 节, Sweetman

8.3.4 MIPS TLB 管理

当发生 TLB 缺失时，处理器完全可以使用硬件，例如 MMU，自动从页表中找到对应的 PTE 并写回到 TLB 中，只要不发生 Page Fault，整个的过程都是硬件自动完成，不需要软件做任何事情，从这个角度来看，似乎不需要对 TLB 进行什么管理，但是需要注意的是，由于 TLB 是页表的缓存，所以 TLB 中的内容必然是页表的子集，也就是说，如果由于某些原因导致一个页的映射关系在页表中不存在了，那么它在 TLB 中也不应该存在，而操作系统在一些情况下，会把某些页的映射关系从页表中抹掉，例如：

（1）当一个进程结束时，这个进程的指令（code）、数据（data）和堆栈（stack）所占据的页表就需要变为无效，这样也就释放了这个进程所占据的物理内存空间。但是，此时在 I-TLB 中可能存在这个进程的程序（code）对应的 PTE，在 D-TLB 中可能还存在着这个进程的数据（data）和堆栈（stack）对应的 PTE，此时就需要将 I-TLB 和 D-TLB 中，和这个进程相关的所有内容都置为无效，如果没有使用 ASID（进程的编号，后文会进行介绍），最简单的做法就是将 I-TLB 和 D-TLB 中的全部内容都置为无效，这样保证新的进程可以使用一个干净的 TLB；如果实现了 ASID，那么只将这个进程对应的内容在 TLB 中置为无效就可以了。

（2）当一个进程占用的物理内存过大时，操作系统可能会将这个进程中一部分不经常使用的页写回到硬盘中，这些页在页表中对应的映射关系也应该置为无效，此时当然也需要将 I-TLB 和 D-TLB 中对应的内容置为无效，但是，一般操作系统会尽量避免将存在于 TLB 中的页置为无效，因为这些页在以后很可能会被继续使用。

因此，抽象出来，对 TLB 的管理需要包括的内容有如下几点。

- 能够将 I-TLB 和 D-TLB 的所有表项（entry）置为无效；
- 能够将 I-TLB 和 D-TLB 中某个 ASID 对应的所有表项（entry）置为无效；
- 能够将 I-TLB 和 D-TLB 中某个 VPN 对应的表项（entry）置为无效。

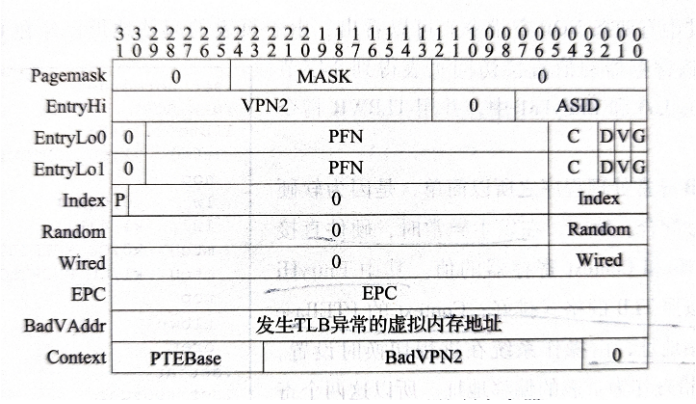

图 8.12: MIPS 指令系统 TLB 相关控制寄存器

MIPS 中定义了专门操作 TLB 的指令，包含 TLBR、TLBWI、TLBWR、TLBP 四条指令。这些指令说明见表8.2（假设 TLB 中表项的个数是 64 个，全相连结构，因此需要 6 位的地址进行寻址）。

同时，为了协助 TLB 指令，MIPS 指令系统中还定义了一组用于访问和控制 TLB 的

表 8.2: MIPS 中对 TLB 进行控制的指令

指令	描述	需要使用的寄存器
TLBP	Probe TLB for matching entry，使用虚拟地址 VA 来查找 TLB 中是否存在它的映射关系，如果存在，将对应 entry 的地址放到 Index 寄存器中	EntryHi：用来存储寻址 TLB 需要使用的 VPN，这个寄存器也包括了 ASID 部分 Index：将 TLB 中被寻址到的 entry 的地址放到这个寄存器中，如果在 TLB 中找到对应的映射关系，则 Index 寄存器的 [32] 置为 0，[5：0] 存储地址；如果没有在 TLB 中找到，则将 Index 寄存器的 [32] 置为 1
TLBR	Read indexed TLB entry，从 TLB 中将 Index 寄存器指定 entry 的内容读出来，放到寄存器 EntryHi 和 EntryLo 中	Index：用来存储寻址 TLB 的地址，会使用这个寄存器的 [5：0] 来寻址 TLB EntryHi：被 Index 寄存器寻址到的 TLB entry 中的 VPN 部分会放到这个寄存器中 EntryLo：被 Index 寄存器寻址到的 TLB entry 中的其他内容会被放到这个寄存器中
TLBWI	Write Indexed TLB entry，向 Index 寄存器寻址到的 TLB entry 中写入 EntryHi 和 EntryLo 的内容	Index：用来存储寻址 TLB 的地址，使用 [5：0] 来寻址 TLB EntryHi：用来存储写入到 TLB entry 中的 VPN EntryLo：用来存储写入到 TLB entry 的其他内容
TLBWR	Write Random TLB entry，使用 Random 寄存器来寻址 TLB，向被寻址的 entry 中写入 EntryHi 和 EntryLo 的内容	Random：一个用来产生随机值的寄存器，只有 [5：0] 有效，用来寻址 TLB，多使用一个计数器来模拟随机的过程 EntryHi：用来存储写入到 TLB entry 中的 VPN EntryLo：用来存储写入到 TLB entry 的其他内容

控制寄存器，包括 EntryHi、EntryLo0、EntryLo1、Pagemask、Index、Random、Wired、EPC、BadVAddr、Context。具体格式如图8.12所示。

MIPS 指令集规定 TLBR、TLBWI、TLBWR 和 TLBP 这 4 条指令用于访问 TLB。TLBR 指令用 Index 寄存器值作为索引将 TLB 表项的值读到 Pagemask、EntryHi、EntyLo0 和 Entrylol 控制寄存器中；TLBWI 和 TLBWR 分别以 Index 寄存器和 Random 寄存器作为索引将上述寄存器的值写到对应 TLB 表项中；TLBP 在 TLB 中查找和 EntryHi 寄存器中的 VPN2 和 ASID 域相匹配的 TLB 表项，若找到则将该表项的索引值写入 Index 寄存器，若没找到则将 Index 寄存器的最高位置为 1。

在与 TLB 直接相关的控制寄存器中，Pagemask、EntryHi、EntryLo 的内容与 TLB 的内容几乎完全一致（除了 G 位），主要用于读写 TLB 表项。写 TLB 时把上述寄存器写到 TLB 某一表项，读 TLB 时将 TLB 表项读到上述寄存器中。

读写 TLB 的哪个表项由 Index、Random 和 Wired 共同控制。除了可以由软件通过 MFC0、MTC0 进行读写访问外，Random 寄存器的值随时钟周期自动变化（之前提到的

Clock 随机替换算法)。Wired 寄存器起到锁定某些 TLB 表项的作用，即指定 Random 寄存器的值只能在 Wired 和 TLB 最大表项减 1 间变化，当软件用 Random 寄存器的值作为索引写 TLB 进行随机替换时，0 到 Wired 寄存器之间的 TLB 表项就不会被替换，因此可以存放一些常用的页表项（如操作系统内核使用的页表项）

EPC 寄存器保存被 TLB 异常打断的指令 PC，BadVAddr 寄存器保存发生 TLB 异常的虚拟内存地址。Context 寄存器用于加速 TLB 重填异常 (TLB miss) 的处理过程，其中 PTEBase 域保存页表起始地址，BadVPN2 域保存发生异常的虚拟页号。因此 Context 直接就是虚拟地址对应 PTE 表项的物理地址。

8.3.5 MIPS TLB 异常

我们前面已经提到了 MIPS 采用了软件的方式来处理 TLB 缺失的情况，这是因为 MIPS 当时的设计理念就是尽可能的让硬件实现起来简单。系统软件可以把 TLB 仅仅看成一个高速的固定大小的转换表。当地址无法转换时，TLB 会触发一个重填异常。为了帮助软件提高效率，TLB 的设计、相关的控制寄存器和重填的细节都需要精心设计。

MIPS 指令系统中 TLB 异常有三种类型：TLB 重填 (refill) 异常指的是 TLB 缺失。TLB 无效 (invalid) 异常指的是相应的物理页不在内存中。TLB 修改 (modified) 异常指的是试图写只读页。其中 TLB 重填有专门的入口地址 (0xbfc00200)，而其他两种异常使用通用异常入口地址 (0xbfc00380)。重填异常有专门入口地址的原因是它发生得最为频繁，专门的入口地址可以避免异常类型的判断。

下面是 MIPS R4000 处理器对应的 TLB 重填异常处理程序（一级页表），该程序仅仅包含 9 条指令。

```
.set noreorder
.set noat
TLBrefill4K:
    mfc0 k1, C0_CONTEXT
    nop
    lw k0, 0(k1)
    lw k1, 4(k1)
    mtc0 k0, C0_ENTRYLO0
    mtc0 k1, C0_ENTRYLO1
    nop
    tlbwr
    eret
.set at
.set reorder
```

该 TLB 异常处理程序之所以简单，是因为软硬件设计的巧妙配合。

- 第一，在发生异常时，硬件直接置好了 EntryHi 和 Context 寄存器的值。其中 EntryHi 的内容已经按照 TLB 的格式排好；Context 的 PTEBase 指向页表起始地址，由操作

系统在进程切换时设置，而 BadVPN2 恰好作为页表的偏移地址。所以这两个寄存器的值可在异常处理时直接使用。

- 第二，页表的每项为 8 字节，已按照 EntryLo0 和 EntryLol 的格式排好，因此取到寄存器后可以直接写入 TLB。
- 第三,TLB 重填异常有专门的入口地址，因此无须判断异常原因。
- 第四，异常处理入口地址在不同 TLB 转换的 kseg0 段，且页表也放在 ksegl 段，因此异常处理程序可保证不会引起新的 TLB 异常。
- 第五，发生异常时，硬件自动将 CPU 状态置为核心态，完成异常处理后 ERET 指令自动将 CPU 状态置为用户态。

重填的过程如图8.13所示。

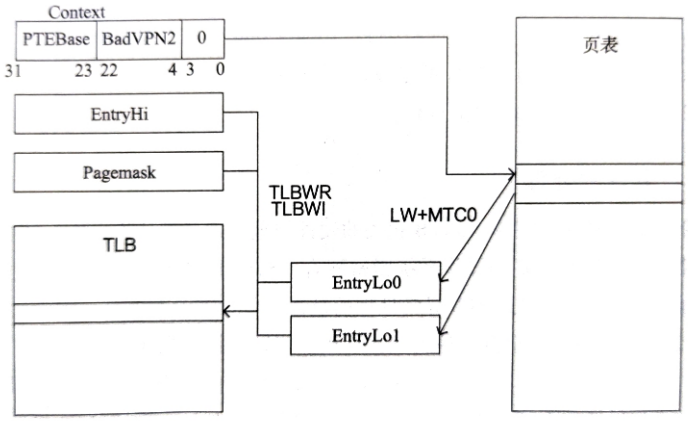

图 8.13: TLB 重填过程

linux 操作系统通常采用两级页表结构，而非 MIPS 早期假设的一级页表。Linux 两级页表结构如图8.14所示。19 位的虚拟页号 (VPN2) 被分为两部分，前 10 位为页目表 (PGD) 索引，后 9 位为二级页表索引。每个物理页的页表项为 4 字节，包括 PFN、C、D、V 和 exts 域。其中 exts 域是软件扩展位，用于维护一些硬件没有实现的功能，例如 ref 位、modified 位等。每个进程的 PGD 表基地址放在进程上下文中，进行切换时就把 PGD 表的基地址写到 Context 寄存器的 PTEBase 域中。

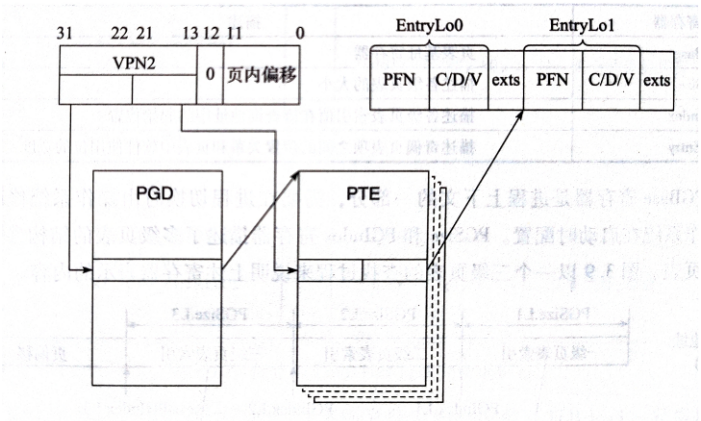

图 8.14: Linux 两级页表结构

Linux 的 TLB 重填异常处理程序如下，共有 18 条指令。

```
mfc0    k0, CP0_BADVADDR      # 取发生TLB缺失的虚拟地址
srl     k0, k0, 22            # 最高10位是第一级页表的索引
lw      k1, pgd_current       # 取当前进程页目录表入口地址
sll     k0, k0, 2             # PGD每项4字节，左移两位得到偏移
addu    k1, k1, k0            # k1指向pgd的一项
mfc0    k0, CP0_CONTEXT       # Context包含发生TLB缺失地址的虚拟页号
lw      k1, 0(k1)             # k1为下一级页表入口地址
srl     k0, k0, 1
and     k0, k0, 0xff8         # 算出二级页表的偏移
addu    k1, k1, k0            # k1指向虚拟地址对应的PTE
lw      k0, 0(k1)             # 成对存放，偶数页
lw      k1, 4(k1)             # 奇数页
srl     k0, k0, 6             # 移除6位exts
mtc0    k0, CP0_ENTRYLO0      # 为TLBWR设置好EntryLo0
srl     k1, k1, 6
mtc0    k1, CP0_ENTRYLO1
tlbwr                         # 写入TLB的一个随机项
eret                          # 异常返回
```

可以看到采用两级页表时，TLB 重填异常处理程序已经比较复杂。若考虑三级页表则需要更多指令。MIPS 原 Context 寄存器的设计不再适合多级页表的快速查找。龙芯处理器通过新增控制寄存器并与专用指令配合的方式来实现多级页表查找的加速。具体内容可以参考《计算机体系结构基础》第二版 p58，胡伟武著。

8.4 加入 TLB 与 Cache

目前我们已经介绍了如何基于 TLB 实现虚拟存储器。再结合上一章介绍的 Cache 有关知识，我们现在即将看见一个完整的计算机存储系统。不过在那之前，我们还面临着一些问题。

TLB(MMU) 负责虚拟地址转换，Cache 负责缓存物理内存的数据。当同时加入这两者时，我们一个很自然的想法就是将他们组织成图8.15所示的结构。事实上，因为这里的 Cache 使用物理地址进行寻址，因此我们把这里的 Cache 称为物理 Cache。

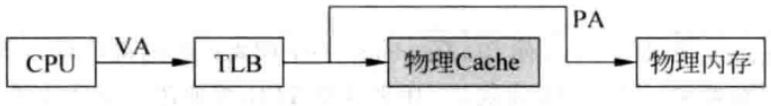

图 8.15: 物理 Cache

很显然，如果不使用虚拟存储器，处理器送出的地址会直接访问物理 Cache，而现在需要先经过 TLB 才能再访问物理 Cache，因此会增加流水线的延迟。如果还想获得和以前一样的运行频率，就需要将访问 TLB 的过程单独使用一级流水线。但这样相当于增加了流水线的级数，对于 I-Cache 来说，增加一级流水线会导致分支预测失败时有更大的惩罚 (penalty)，而对于 D-Cache 来说，增加一级流水线会造成 load 指令的延迟 (latency) 变大。因此 TLB 使用单独流水线不是一种很好的做法。

既然我们最终只是要从虚拟地址获得对应的数据，那么 Cache 可以直接使用虚拟地址吗？当然是可以的，因为这个 Cache 使用虚拟地址来寻址，我们称之为虚拟 Cache。图8.16显示了这种结构。虚拟 Cache 方案，在 Cache 命中时就不需要去访问内存，因此也

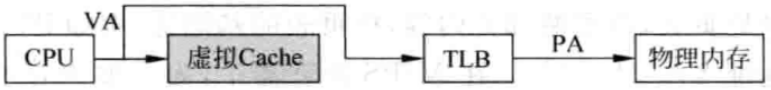

图 8.16: 虚拟 Cache

就不需要进行 TLB 地址转换。在流水线中使用这种虚拟 Cache，不会对时钟周期造成影响。但这种方案也会带来一些问题：

- 同义问题 (synonyms)，也叫重名 (aliasing)。本质为不同的虚拟地址对应着同一个物理地址。这可能带来两个问题。第一，浪费了宝贵的 Cache 空间，降低整体性能。因为 Cache 的不同位置存储了同一物理地址对应的数据。第二，当对某一个虚拟地址执行写操作时，只会修改 Cache 中的一份数据。这导致通过另一个虚拟地址访问时，会获得没有更新的值，从而导致错误。
- 同名问题 (homonyms)。本质为相同的虚拟地址对应着不同的物理地址。这导致不同进程之间的数据会相互造成干扰。

虚拟 Cache 的两种问题都有相应的解决方案，感兴趣的同学可以参考《超标量处理器设计》3.4.2 节。

既然两种方案都有问题，我们应该使用哪一种呢？事实上，我们能从更加底层的角

度来看待这两种方案的异同。我们知道 cache 进行查找时，会先使用地址中的 index 部分索引到一个 cache line（假设是直接映射），然后再比较 cache 中的 tag 部分和地址中的 tag 部分来判断是否命中。因此我们根据 index 和 tag 是虚拟地址还是物理地址，可以细化出以下三种方案。

Physical-indexed, Physical-tagged

图8.17显示了这种方案的示意图。这种方案便是上面介绍的物理 Cache 方案。先经过 TLB 将虚拟地址转换成物理地址，图中的页内偏移 offset 为 k 位。Cache 使用转换后的物理地址，根据其中的 index 找到 cache line，再根据物理 tag 来判断是否命中。整个过程都是串行的，因此时延很高。这种方案在真实的处理器中很少被使用。

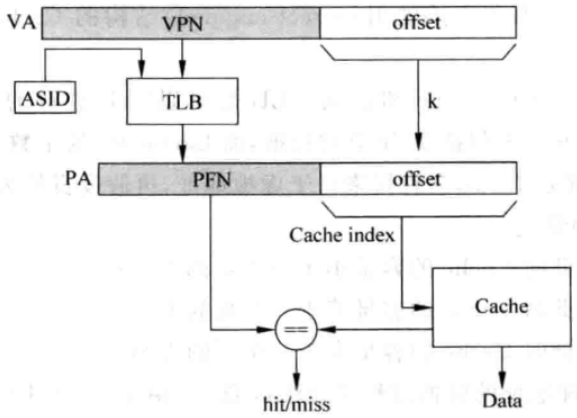

图 8.17: 物理索引-物理标签

Virtual-indexed, Physical-tagged

因为虚拟地址转换时，offset 部分是保持不变的。因此可以利用这点，索引 cache line 时使用虚拟地址，进行 tag 比对时使用物理地址。在这样的方案中，访问 Cache line 和访问 TLB 是并行的，因此可以解决物理 Cache 带来的时延问题。大多数现代处理器都采用了这种方案。

我们假设每个 Cache line 包含 2^b 字节数据，一共有 2^L 个 Cache set。也就是说寻址 Cache 需要的地址长度为 $L+b$，它直接来自虚拟地址。假设一个页大小为 2^k 字节，即 offset 的地址长度为 k。

因此会有以下几种情况：

- $k > L+b$ 此时 Cache 一路大小小于页大小
- $k = L+b$ 此时 Cache 一路大小等于页大小
- $k < L+b$ 此时 Cache 一路大小大于页大小

对于前两种情况，索引 cache line 需要的 index 完全处于 offset 内，因此虚拟地址的 index 和物理地址的 index 是一样的。这种方案本质上就是一个物理 Cache，但可以对 Cache 和 TLB 进行并行访问。图8.18即显示了第二种情况 2。

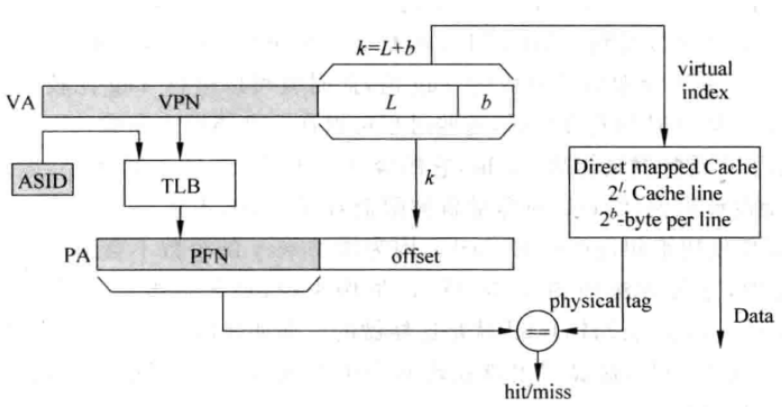

图 8.18: 虚拟索引-物理标签，情况 2

前两种情况已经非常好了，唯一的问题就是限制了 cache 一路的大小。因此为了扩大 cache 的容量，就需要扩大相联度。举例来说，对于 4KB 的页，如果需要一个大小为 32KB 的 Cache，那么就需要 8-way 的设计，Intel 就是这样做的。那如果需要使用一个 4MB 的 Cache，需要 way 的个数是 1024，这显然很难在现实当中实现，因此使用这种方式，对于 Cache 容量的大小是有限制的。

第 3 种情况可以解决 Cache 容量受到限制的问题，但是同样会面临虚拟 cache 中的重名 (aliasing) 问题。可以采用二级 cache 的方式解决该问题。感兴趣的同学可以参考《超标量处理器设计》p91。

Virtual-indexed, Virtual-tagged

这种方式即对应虚拟 cache 方案，会遇到之前提过的同义和同名问题。也可以通过二级 Cache 的方式解决，这里不仔细讲述。

8.5 TLB 实现

8.5.1 verilog 实现

经过这么多介绍，我们现在来讲解如何给我们的 MIPS 处理器添加 TLB 模块。我们要实现的是上一节介绍的 VIPT 方案 (virtual-indexed, physical-tagged)，页大小和 cache 一路大小都是 4KB。加入 TLB 模块后，我们的 cpu 顶层结构如图8.19所示。

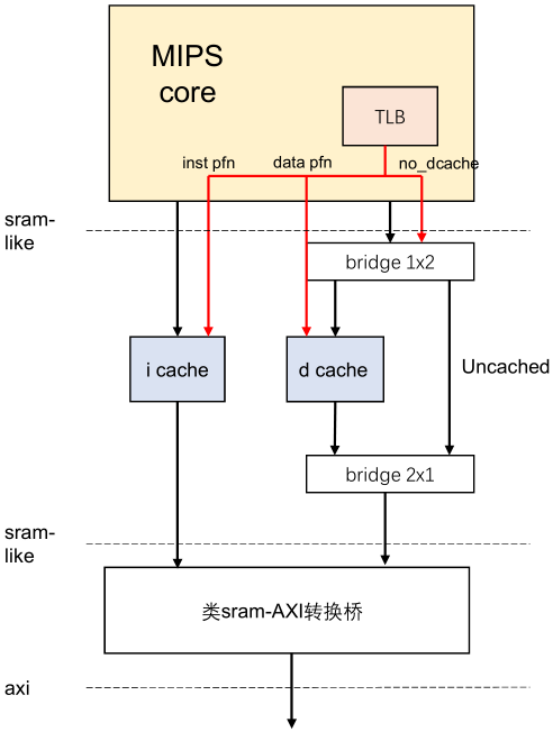

图 8.19: 加入 TLB 后 CPU 顶层结构图

我们实现的 TLB 只是一个组合逻辑的转换表，因此我们将 TLB 模块放入了 MIPS core 内部。具体放在哪看个人自己的实现。我们的 TLB 同时进行指令和数据共用 TLB 表项，最终输出指令和数据虚拟地址对应的物理页号。

因为我们将 TLB 放在了 MIPS core 内部，因此对外部来说，可以不必关心 TLB 的存在，MIPS core 模块输出的地址即为物理地址。只不过，Cache 判断是否命中时需要改为使用 TLB 输出的物理页号。因为某些地址空间访问不经过 cache(如 kseg1 段)，因此我们还需要一个通路直接对外访问内存，不经过 cache。是否经过 cache 的控制信号也由 TLB 输出。(这里只实现了不经过数据 cache，事实上还要实现不经过指令 cache，也就是还要添加一个直接对外访问的通路)

总之，为了添加 TLB，需要改动以下的设计：

- CPU 需要增加 TLBP, TLBR, TLBWI 等指令的解码逻辑
- CPU 需要增加 TLB 有关 CP0 寄存器
- CPU 需要增加数据通路，如处理 TLBR 指令时，写入 CP0 的数据来源需要增加（原来只有 MTC0）

- CPU 需要增加处理 TLB 异常的逻辑
- Cache 判断命中时 tag 比较使用物理页号 PFN
- 添加 TLB 模块

由于每个人的 CPU 实现不一致，本节主要介绍和 TLB 自身有关的部分，CPU 部分请自己实现。本节实现的 TLB 模块只是作为参考，具体接口可能需要根据自己的 CPU 做出调整。

TLB 主要工作

总的来说，TLB 模块主要完成以下工作:

1. 指令和数据的虚拟地址转换（获得物理页号即可）
2. 处理 TLBP, TLBR, TLBWI 为代表的 TLB 指令
   - TLBP: 使用 EntryHi 里的 VPN2 和 ASID 去查找整个 TLB，看是否有某一 TLB 项可以翻译该虚拟地址。如果查找到则将该项 TLB 索引号写入到 cp0 寄存器 Index 中；如果没找到则将 Index 高位置 1，其他位任意值。
   - TLBWI: 以 cp0 寄存器 Index 为索引，将 EntryHi、PageMask、EntryLo0、EntryLo1 写入到寻址到的 TLB 项中。
   - TLBR: 与 TLBWI 相反，是以 Index 为索引，读出寻址到的 TLB 项的值写入到 EntryHi、PageMask、EntryLo0、EntryLo1。
3. 返回异常
   - 指令的 refill, invalid 异常或数据的 refill, invalid, modify 异常。具体参考 MIPS 手册

以上工作可提取其相同部分，细分为以下部分:

1. 根据虚拟地址，查找 TLB 表项，获得索引:
   - 将输入的地址与 TLB 中的每一项做对比，生成一个独热码 (只有 1 位为 1)，然后通过编码器生成一个索引 index。
2. 根据索引，读取 TLB 表项:
   - 根据 index 直接访问 TLB 中对应的项，index 可以来自上一步查找生成的 index，也可以来自 Index CP0 寄存器输入 (TLBR 指令)
3. 根据索引，和输入的 CP0 值，写入 TLB 表项:
   - 对应 TLBWI, TLBWR 指令
4. 计算物理页号，产生异常，输出 TLB 有关 CP0 寄存器的值
   - 如根据读取到的 TLB 表项中的 EntryLo，计算虚拟地址的物理页号。

可以看出，上面过程将虚拟地址转换分为了两个步骤。第一步通过将虚拟地址与 TLB 表项逐一比较，生成独热码，再进一步编码成索引。第二步再通过索引直接读取 TLB 表项的内容，从而获得物理页号的内容。

分成两步有许多好处。第一，更利于代码的重用。因为指令和数据地址翻译乃至 TLBP 指令，第一步都是要生成索引。而指令和数据地址翻译和 TLBR 指令又可以共用第二步通过 index 读取 TLB 表项的逻辑。第二，便于将 TLB 模块**划分流水线**。因为 TLB 模块带

来的时延是非常大的，而第一步和第二步刚好是两步时延比较高的部分，将其拆分成流水线有利于提高时序。提供一个思路：访存请求是在流水线访存阶段发出的，但事实上访存需要的地址在流水线执行阶段便可以获得。从而可以提前将地址传给 TLB 模块。这样 TLB 模块的流水线便被隐藏到了 CPU 的 5 级流水中。当然，我们这里给出的示例代码还是以简单为主。*TLB* 没有划分流水线，所有过程都是组合逻辑。

TLB 输入输出接口

考虑了 TLB 模块的主要功能后，我们来确定 TLB 模块的输入输出接口。

首先，完成指令和数据的虚拟地址翻译需要以下接口。vaddr 代表虚拟地址，pfn 代表物理页号，宽度和 Cache 的 TAG 的宽度一致为 20 位 (用于实现虚拟索引物理标签 VIPT)。no_cache 信号用于表示指令和数据访存是否经过 cache。

```
input wire [31:0] inst_vaddr,
input wire [31:0] data_vaddr,

output wire [`TAG_WIDTH-1:0] inst_pfn,
output wire [`TAG_WIDTH-1:0] data_pfn,
output wire no_cache_i,
output wire no_cache_d,
```

然后，需要处理 TLB 指令需要以下接口。如 TLBP 由 CPU 译码产生，表示当前有一条 TLBP 指令。

```
//TLB指令
input wire TLBP,
input wire TLBR,
input wire TLBWI,
input wire TLBWR,

input wire [31:0] EntryHi_in,
input wire [31:0] PageMask_in,
input wire [31:0] EntryLo0_in,
input wire [31:0] EntryLo1_in,
input wire [31:0] Index_in,
input wire [31:0] Random_in,
output wire [31:0] EntryHi_out,
output wire [31:0] PageMask_out,
output wire [31:0] EntryLo0_out,
output wire [31:0] EntryLo1_out,
output wire [31:0] Index_out
```

其中 in 结尾的变量表示 CP0 寄存器输入，即 CP0 当前的值。而 out 结尾的变量代表 TLB 指令的输出，即 CP0 需要更新的值。

最后 TLB 模块还要输出异常信号，指令请求有两种，数据有三种。

```
//异常
output wire inst_tlb_refill, inst_tlb_invalid,
output wire data_tlb_refill, data_tlb_invalid, data_tlb_modify,
```

宏定义、TLB 表项

为了使之后的代码更清晰，我们定义一些宏，用于表示 TLB 配置和 TLB 有关的 CP0 寄存器的位域。这里定义了 TLB_LINE_NUM 为 8，表示有 8 个 TLB 表项。而剩下的则定义了 CP0 寄存器（同时也对应 TLB 表项）中一些重要的位域。

```
//TLB
//TLB Config
`define TLB_LINE_NUM 8
`define TAG_WIDTH 20
`define OFFSET_WIDTH 12
`define LOG2_TLB_LINE_NUM 3

//index
`define INDEX_BITS `LOG2_TLB_LINE_NUM-1:0
//random
`define RANDOM_BITS `LOG2_TLB_LINE_NUM-1:0
//wired
`define WIRED_BITS `LOG2_TLB_LINE_NUM-1:0

//EntryHi
`define VPN2_BITS 31:13
`define ASID_BITS 7:0
//G bit in TLB entry
`define G_BIT 12

//PageMask
`define MASK_BITS 24:13

//EntryLo
`define PFN_BITS 25:6
`define FLAG_BITS 5:0
`define V_BIT 1
`define D_BIT 2
`define C_BITS 5:3

//context
`define PTE_BASE_BITS 31:23
```

```
‘define BAD_VPN2_BITS 22:4
```

然后我们用 reg 定义 TLB 的表项，对应各 CP0 寄存器。

```
//TLB
reg [31:0] TLB_EntryHi [‘TLB_LINE_NUM-1:0]; //G位放在EntryHi的第12位
reg [31:0] TLB_PageMask [‘TLB_LINE_NUM-1:0];
reg [31:0] TLB_EntryLo0 [‘TLB_LINE_NUM-1:0];
reg [31:0] TLB_EntryLo1 [‘TLB_LINE_NUM-1:0];
```

查询 TLB 表项

第一阶段：查询生成索引。因为指令和数据地址翻译各需要一个端口，TLBP 指令还需要一个端口，因此这里提供了三个端口 vaddr1, vaddr2, vaddr3。由于代码冗余性较高，这里只展示了 find_mask1 和 find_index1。

首先生成独热码 find_mask，这里用到了 verilog 的 generate for 语句。generate for 语句会直接重复 for 循环内部的电路。判断 TLB 表项是否命中的逻辑如下：判断虚拟地址中的虚拟页号 VPN2(VPN2 是 19 位而 VPN 是 20 位，因为 MIPS 一个 TLB 表项同时映射奇偶两个页) 是否与 TLB 中 EntryHi 部分的 VPN2 一致 (两个 VPN2 都需要和 PageMask 与非一下)。再判断当前 EntryHi 中的 ASID 与 TLB 表项中的 ASID 是否一致，或者 TLB 表项中的 G 位是否为 1。

生成 find_mask 后，再通过编码器生成 find_index。

```
//--------------------------查找逻辑------------------------------
wire [31:0] vaddr1, vaddr2, vaddr3;

assign vaddr1 = inst_vaddr;
assign vaddr2 = data_vaddr;
assign vaddr3 = EntryHi_in;

wire [‘TLB_LINE_NUM-1: 0]  find_mask1, find_mask2, find_mask3;
wire [‘LOG2_TLB_LINE_NUM-1:0] find_index1, find_index2, find_index3; //找到
    的TLB项的索引，通过find_mask生成
wire find1, find2, find3; //是否找到
assign find1 = |find_mask1;

genvar i;
generate
  for (i = 0; i < ‘TLB_LINE_NUM; i = i + 1)
  begin : find
    assign find_mask1[i] = ((vaddr1[‘VPN2_BITS] & ~TLB_PageMask[i][
        ‘VPN2_BITS]) == (TLB_EntryHi[i][‘VPN2_BITS] & ~TLB_PageMask[i][
        ‘VPN2_BITS])) && (TLB_EntryHi[i][‘G_BIT] || TLB_EntryHi[i][
        ‘ASID_BITS] == EntryHi_in[‘ASID_BITS]);
```

```
  end
endgenerate

//编码器，通过mask生成index
assign find_index1=
({3{find_mask1[0 ]}} & 3'd0 ) |
({3{find_mask1[1 ]}} & 3'd1 ) |
({3{find_mask1[2 ]}} & 3'd2 ) |
({3{find_mask1[3 ]}} & 3'd3 ) |
({3{find_mask1[4 ]}} & 3'd4 ) |
({3{find_mask1[5 ]}} & 3'd5 ) |
({3{find_mask1[6 ]}} & 3'd6 ) |
({3{find_mask1[7 ]}} & 3'd7 ) ;
//--------------------------查找逻辑-----------------------------
```

读取 TLB 表项

第二阶段：读取 TLB 表项。这里同样提供了三个端口，前两个对应指令和数据。第三个 index 有两个来源：分别是 TLBP 找到的 index，或者 TLBR 提供的 Index CP0 寄存器。

```
//--------------------------读TLB逻辑-----------------------------
wire [`LOG2_TLB_LINE_NUM-1: 0] index1, index2, index3;

assign index1 = find_index1;
assign index2 = find_index2;
assign index3 = TLBP ? find_index3 : Index_in[`INDEX_BITS];

wire [31:0] EntryLo0_read1;
wire [31:0] EntryLo1_read1;

wire [31:0] EntryLo0_read2;
wire [31:0] EntryLo1_read2;

wire [31:0] EntryHi_read3;
wire [31:0] PageMask_read3;
wire [31:0] EntryLo0_read3;
wire [31:0] EntryLo1_read3;

wire [31:0] EntryLo0_read2;
wire [31:0] EntryLo1_read2;

assign EntryLo0_read1 =TLB_EntryLo0[index1];
```

```
assign EntryLo1_read1 =TLB_EntryLo1[index1];

assign EntryLo0_read2 =TLB_EntryLo0[index2];
assign EntryLo1_read2 =TLB_EntryLo1[index2];

assign EntryHi_read3 =TLB_EntryHi[index3];
assign PageMask_read3 =TLB_PageMask[index3];
assign EntryLo0_read3 =TLB_EntryLo0[index3];
assign EntryLo1_read3 =TLB_EntryLo1[index3];
//-------------------------读TLB逻辑-----------------------------
```

写入 TLB 表项

第三阶段：写入 TLB 表项。这里需要注意一下 TLB_EntryHi 的 G_BIT 的生成。

```
//-------------------------写TLB逻辑-----------------------------
//写TLB表项的index有TLBWI和TLBWR两个来源
wire [`LOG2_TLB_LINE_NUM-1: 0] write_index;

assign write_index = TLBWI ? Index_in[`INDEX_BITS] : Random_in[`INDEX_BITS];

integer tt;
always @(posedge clk)
begin
    if(rst) begin
        for(tt=0; tt<`TLB_LINE_NUM; tt=tt+1) begin
            TLB_EntryHi [tt] <= 0;
            TLB_PageMask[tt] <= 0;
            TLB_EntryLo0[tt] <= 0;
            TLB_EntryLo1[tt] <= 0;
        end
    end
    else if (TLBWI | TLBWR)
    begin
        TLB_EntryHi [write_index][`VPN2_BITS] <= EntryHi_in[`VPN2_BITS] & ~
            PageMask_in[`VPN2_BITS];
        TLB_EntryHi [write_index][`G_BIT] <= EntryLo0_in[0] & EntryLo1_in
            [0];
        TLB_EntryHi [write_index][`ASID_BITS] <= EntryHi_in[`ASID_BITS];
        TLB_PageMask[write_index]          <= PageMask_in;
        TLB_EntryLo0[write_index][`PFN_BITS] <= EntryLo0_in[`PFN_BITS] & ~
            PageMask_in[`MASK_BITS];
        TLB_EntryLo0[write_index][`C_BITS] <= EntryLo0_in[`C_BITS];
        TLB_EntryLo0[write_index][`D_BIT] <= EntryLo0_in[`D_BIT];
```

```
        TLB_EntryLo0[write_index][‘V_BIT] <= EntryLo0_in[‘V_BIT];
        TLB_EntryLo1[write_index][‘PFN_BITS] <= EntryLo1_in[‘PFN_BITS] & ~
            PageMask_in[‘MASK_BITS];
        TLB_EntryLo1[write_index][‘C_BITS] <= EntryLo1_in[‘C_BITS];
        TLB_EntryLo1[write_index][‘D_BIT] <= EntryLo1_in[‘D_BIT];
        TLB_EntryLo1[write_index][‘V_BIT] <= EntryLo1_in[‘V_BIT];
    end
end
//-------------------------写TLB逻辑-----------------------------
```

输出

第四阶段：输出。因为指令和数据地址翻译逻辑一致，这里只显示了数据的部分。需要注意的是 MIPS 规定 kseg0 和 kseg1 段直接映射而不用经过 TLB，生成物理页号时需要选择一下。至于 no_cache 信号的生成，因为 kseg1 段规定是不经过 Cache 的，因此也要单独考虑。其它情况通过 TLB 中读到的 Cache 属性进行判断。

TLB 指令 (TLBP, TLBR) 的输出较为简单，通过读取到的 TLB 表项值生成即可。

生成异常信号时引入了三个新的输入信号 inst_e, mem_read_en 和和 mem_read_en。分别表示指令使能、数据读使能和数据写使能。因为我们的 TLB 模块是组合逻辑，即使在没有数据请求的情况下，同样会进行数据地址翻译。我们只要保证此时不会错误的产生异常信号即可。

```
//-------------------------output--------------------------------
/*data地址映射*/
//获取地址奇偶性
wire data_odd;
assign data_odd = data_vaddr[‘OFFSET_WIDTH];

//地址是否处于kseg0/1段
wire data_kseg01;
wire data_kseg1;
assign data_kseg01 = data_vaddr[31:30]==2’b10 ? 1’b1 : 1’b0;
assign data_kseg1 = data_vaddr[31:29]==3’b101 ? 1’b1 : 1’b0;

//地址虚拟页号
wire [‘TAG_WIDTH-1:0] data_vpn;
assign data_vpn = data_vaddr[31:‘OFFSET_WIDTH];

//获得物理页号
assign data_pfn = data_kseg01? {3’b0, data_vpn[‘TAG_WIDTH-4:0]} :
                ~data_odd   ? EntryLo0_read2[‘PFN_BITS] : EntryLo1_read2[
                    ‘PFN_BITS];
```

```
//获得是否经过Cache属性
wire [5:0] data_flag;
assign data_flag = ~data_odd ? EntryLo0_read2[`FLAG_BITS] : EntryLo1_read2[
    `FLAG_BITS];

assign no_cache_d = data_kseg01 ? (data_kseg1 ? 1'b1 : 1'b0) :
                  data_flag[`C_BITS]==3'b010 ? 1'b1 : 1'b0;

/*TLB指令*/
    //TLBR
assign EntryHi_out = EntryHi_read3;
assign PageMask_out = PageMask_read3;
assign EntryLo0_out = {EntryLo0_read3[31:1], EntryHi_read3[`G_BIT]};
assign EntryLo1_out = {EntryLo1_read3[31:1], EntryHi_read3[`G_BIT]};

    //TLBP
assign Index_out  = find3 ? find_index3 : 32'h8000_0000; //没找到Index最高位
    置1

/*异常*/
    //取指TLB异常
assign inst_tlb_refill = inst_kseg01 ? 1'b0 : (inst_en & ~find1);
assign inst_tlb_invalid = inst_kseg01 ? 1'b0 : (inst_en & find1 & ~inst_flag
    [`V_BIT]);

    //load/store TLB异常
wire data_V, data_D;
assign data_V = data_flag[`V_BIT];
assign data_D = data_flag[`D_BIT];

assign data_tlb_refill = data_kseg01 ? 1'b0 : (mem_read_enM | mem_write_enM)
     & ~find2;
assign data_tlb_invalid = data_kseg01 ? 1'b0 : (mem_read_enM | mem_write_enM
    ) & find2 & ~data_V;
assign data_tlb_modify = data_kseg01 ? 1'b0 : mem_write_enM & find2 & data_V
     & ~data_D;
//--------------------------output--------------------------------
```

到此为止，TLB 模块的功能便完成了。

8.5.2 TLB 测试环境

我们使用国科大 TLB 实验中的软件测试环境。测试环境见github 仓库