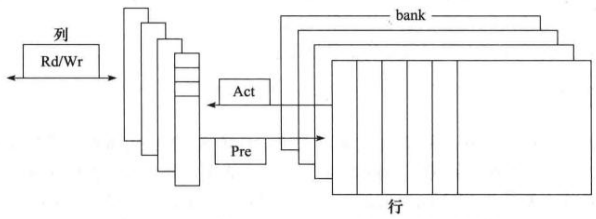
一.存储器

块/行：两级层次结构中存储信息交换的最小单元

平均访问时间=命中率*命中时间+缺失率*缺失代价

DRAM

行填充，可以连续访问



4 DRAM 的内部组织。现代 DRAM 以 bank（存储块）方式组织，典型的 DDR3 中有 4 个 bank。每个 bank 由多个行组成。发送一条 Pre（预充电）命令能够打开或者关闭一个 bank。使用 Act（激活）命令发送一个行地址，将对应的行中的数据传送到一个缓冲器中。当一行数据在缓冲器中时，无论 DRAM 数据宽度（典型情况为 4、8 或 16 位）是多少，都可以通过指定要传送的数据块大小和数据块在缓冲器中的起始地址的方式连续传送相邻地址的数据。与数据块的传送一样，每条命令使用时钟进行同步

闪存

磁盘存储器

每个磁盘的表面划分为同心圆盘，称为**磁道**（track）。每个面通常有几万条磁道。每条磁道同样被划分为用于存储信息的**扇区**（sector）；每条磁道有几千个扇区。每个扇区的容量通常是 512 ~

扇区：磁盘上磁道的基本单位，是磁盘上数据读写的最小单位

访问数据的三步：

1.寻道：将磁头移动到适当的磁道上

2.等待（旋转延时）：磁头到了正确的磁道，必须等待访问的扇区转动到读写头下面（该时间通常是磁盘转动一周时间的一半）

3.传输：传输一块数据需要的时间

二.cache

cache 的基本原理

处理器每次请求一个字，每个块也由一个单独的字组成，注意块是两级层级结构中的最小单元（后面会学到每个块由多个字组成）

直接映射：一种 cache 结果，其中每个存储器地址仅仅对应到 cache 中的一个位置

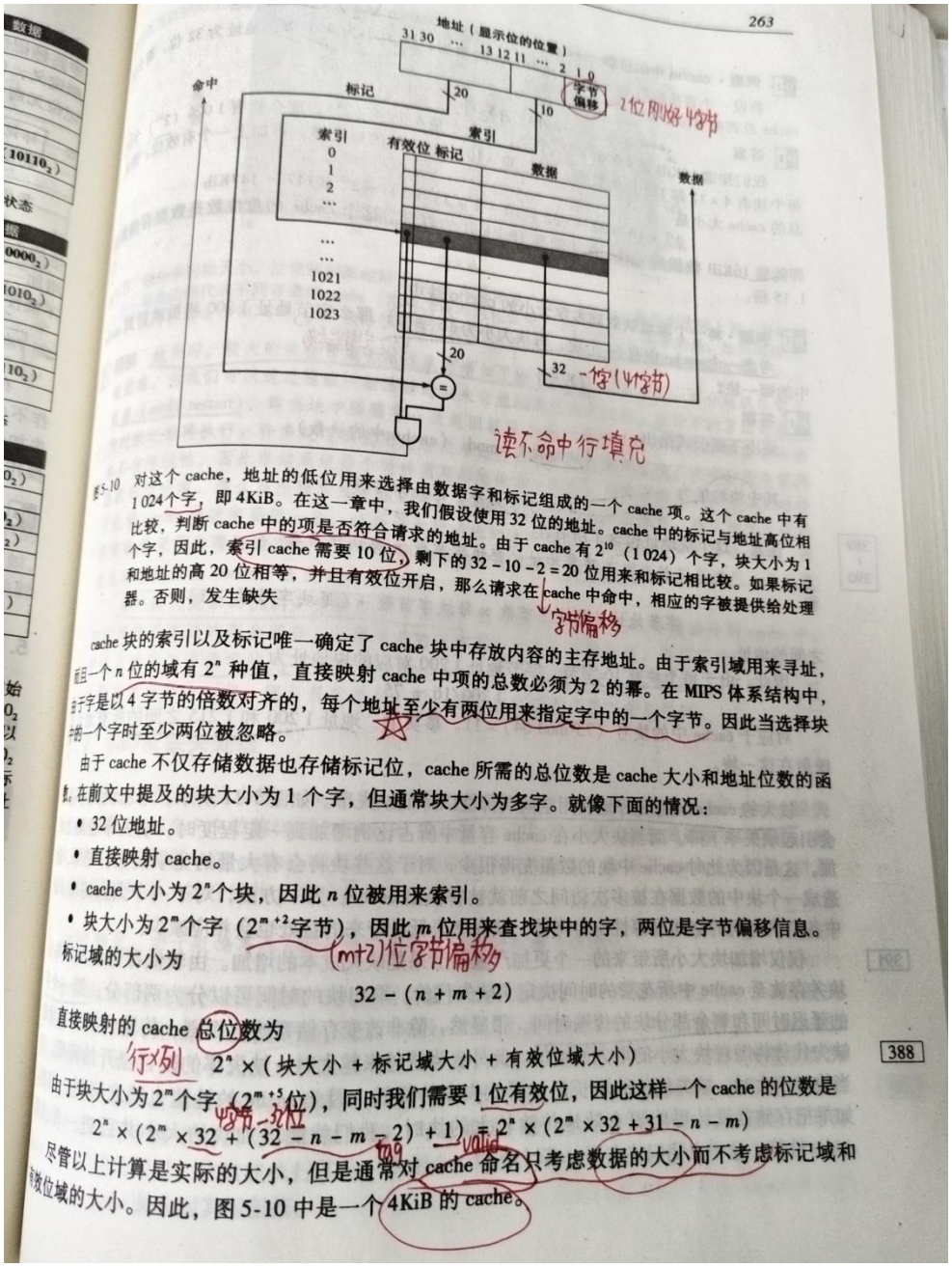
索引：就是我们常说的索引，可以把它看成矩阵中的行，这一点有助于后面相关结构的想象以及相关概念的了解

标记（tag）：只需包含地址的高位，也就是没有用来检索 cache 的那些位，在后面会发现还可能需要减去（字节偏移+字偏移）或者块偏移对应的位

有效位：用来表示一个块的数据是否有效，因为初始化的时候很多数据都是无效的，所以需要 1 位这样的有效位进行标识

映射方法：（块地址）mod（cache 中的块数）

cache 的访问



263

地址（显示位的位置）
31 30 … 13 12 11 … 2 1 0
命中 标记 20 索引 10 字节偏移 2位刚好4字节
索引 有效位 标记 数据 数据
0
1
2
…
1021
1022
1023
20 32 一字(4个字节)
读不命中行填充

图 5-10 对这个 cache，地址的低位用来选择由数据字和标记组成的一个 cache 项。这个 cache 中有 1 024个字，即 4KiB。在这一章中，我们假设使用 32 位的地址。cache 中的标记与地址高位相比较，判断 cache 中的项是否符合请求的地址。由于 cache 有 2^{10}（1 024）个字，块大小为 1 个字，因此，索引 cache 需要 10 位，剩下的 32 - 10 - 2 = 20 位用来和标记相比较。如果标记和地址的高 20 位相等，并且有效位开启，那么请求在 cache 中命中，相应的字被提供给处理器。否则，发生缺失

字节偏移

cache 块的索引以及标记唯一确定了 cache 块中存放内容的主存地址。由于索引域用来寻址，而且一个 n 位的域有 2^n 种值，直接映射 cache 中项的总数必须为 2 的幂。在 MIPS 体系结构中，字是以 4 字节的倍数对齐的，每个地址至少有两位用来指定字中的一个字节。因此当选择块中的一个字时至少两位被忽略。

由于 cache 不仅存储数据也存储标记位，cache 所需的总位数是 cache 大小和地址位数的函数。在前文中提及的块大小为 1 个字，但通常块大小为多字。就像下面的情况：

- 32 位地址。
- 直接映射 cache。
- cache 大小为 2^n 个块，因此 n 位被用来索引。
- 块大小为 2^m 个字（2^{m+2} 字节），因此 m 位用来查找块中的字，两位是字节偏移信息。

标记域的大小为 (m+2)位字节偏移

$$32 - (n + m + 2)$$

直接映射的 cache 总位数为 行x列

$$2^n \times (\text{块大小} + \text{标记域大小} + \text{有效位域大小})$$

388

由于块大小为 2^m 个字（2^{m+5} 位），同时我们需要 1 位有效位，因此这样一个 cache 的位数是

$$2^n \times (2^m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 31 - n - m)$$

尽管以上计算是实际的大小，但是通常对 cache 命名只考虑数据的大小而不考虑标记域和有效位域的大小。因此，图 5-10 中是一个 4KiB 的 cache。

1.一个块由一个字单独组成，对于一个 32 位地址，想访问具体的字节，那么就需要字节偏移，1 个字是 4 个字节，是 2 的平方，因此需要 2 位来指定字中的一个字节。

2.对于索引的位数，由于 cache 有 1024 块，因此需要 1024 个索引，也就是 2 的 10 次方，因此索引的位数是 10 位。

3.对于标记的位数，剩下的全是标记位，也就是(32-10-2)=20 位

直接映射 cache 的总位数：符号表示的不利于记忆，看文字表示的

$$2^n \times (\text{块大小} + \text{标记域大小} + \text{有效位域大小})$$

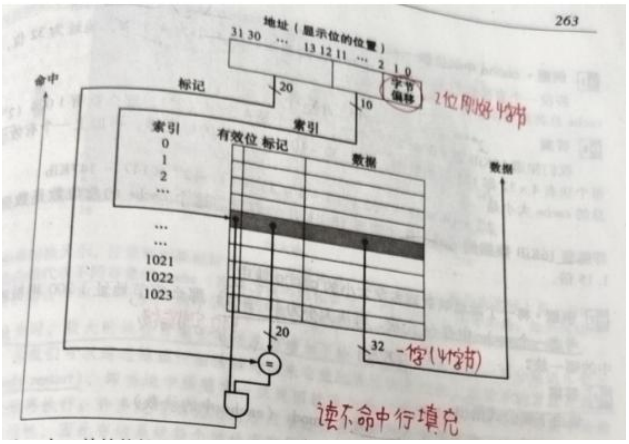
将 cache 这个结构看成矩阵，那么 2 的 n 次方其实就是索引总数，其实就是相乘中对应的是行，后面的全部对应的是列，行*列就得到最终的总位数

区分主存地址和 cache：

以下需要空间想象一下

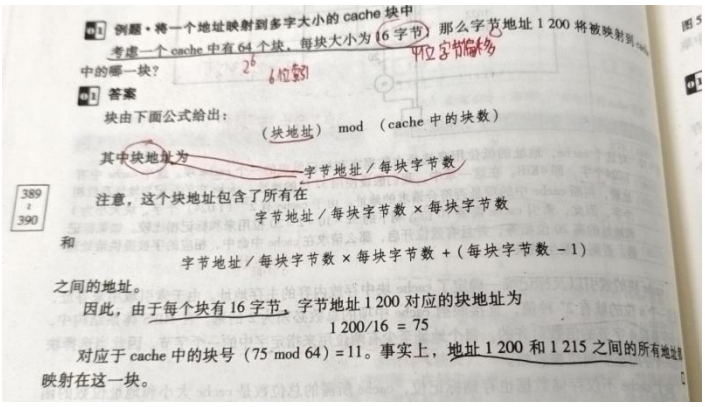
在这个 cache 结构中，一个块对应一个字也就是四个字节，数据域有 32 位也是四个字节，这里给的主存地址是字节地址，但是给的 cache 中数据位其实是四个字节，这里是将这四个字节平放在一起，如果全部纵向铺开，显然地址的低 2 位是用来索引访问的这个字的哪个字节。那么把地址中索引位和字节偏移位拼接在一起，其实就相当于把数据按照字节地址的形式纵向铺开，这也就顺理成章地得出索引位和字节偏移的由来

cache 中数据域的位数根据块的大小直接得到，标记域的位数需要根据主存地址减去索引位数和各种偏移位数（字节偏移，字偏移，块偏移）



cache 的命名：只考虑数据的大小而不考虑标记域有效域的大小，以上有 1024（2 的 10 次方）个索引，每个块的数据域是 1 个字（4 个字节）,因此 cache 的大小为 1024*4=4KiB

将一个地址映射到多字大小的 cache 块中：



以下需要想象 cache 的具体结构

一般给的地址都是字节地址，块地址=字节地址/每块的字节数，块号=块地址 mod(cache 中的块数)（块号其实就是索引中第几个）

判断 cache 是否缺失：

1.通过索引找到对应的块，然后比较标记

2.块地址相同（只是索引相同不行，因为索引=块地址 mod 块数，不同块地址可能对应一个索引），并且之前已经被存放到 cache 中，那么就是命中

cache 的缺失处理

cache 的缺失：由于数据不在 cache 中而导致被请求的数据不能满足
cache 的缺失处理由两部分共同完成：1.处理器控制单元
2.一个进行初始化访问和重新填充 cache 的独立寄存器 发生缺失时，处理器发生阻塞，直到从存储器中取回数据后才响应

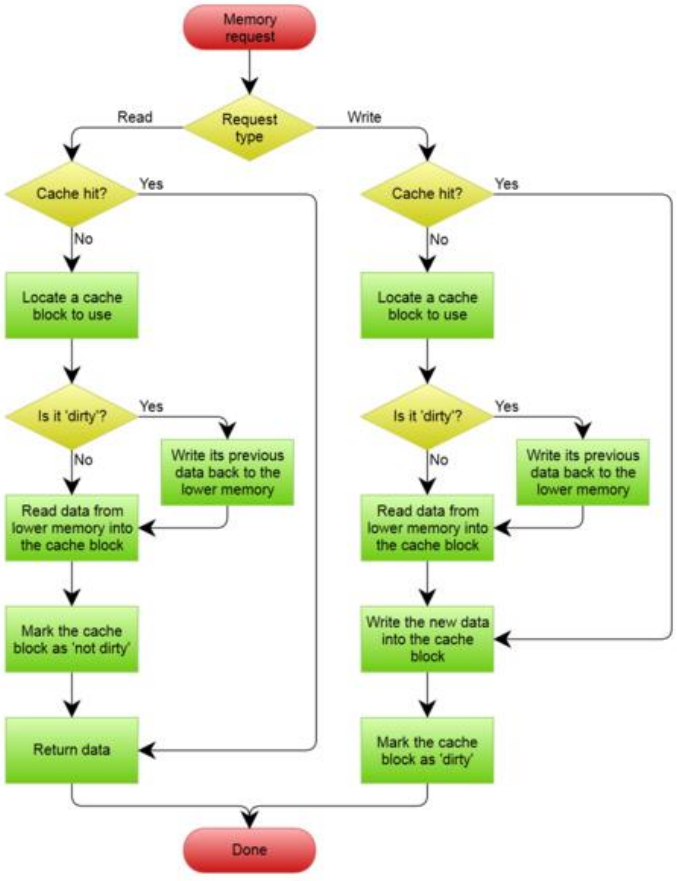
脏位 (dirty):当内存上的某个块需要被新的块替换时，它需要根据脏位判断这个块之前有没有被修改过，如果被修改过，先把这个块更新到主存再替换，否则就直接替换。

写操作处理

cache 与主存不一致：主存和 cache 中相应位置中的值不同
写直达：写操作总是同时更新 cache 和下一存储器层次（保持一致性的最简单的方法）
写缓冲：一个保存等待写入主存数据的缓冲队列
写回：发生写操作时，新值仅仅被写入 cache 块中，只有当修改过的块被替换时才写到较低层存储结构中
写分配：分配 cache 中的一块，数据块从主存中取回，并且再该块中的恰当区域重写数据

写缺失：

1 是写的时候 cache 为空 2 是写的时候 cache 有值但是与内存不一样（不一样的原因可能是因为 cache 存的是中间变量），即是 dirty。
这里的图示显然是用了写回的方法，需要一个写缓冲来保存数据，需要在存储层的较低层进行备份



最小的读/写带宽：

备份数据的思想很重要

读带宽：从存储器读到 cache 的带宽 **写带宽**：从 cache 写到存储器的带宽

思考角度：一般分为指令 cache 和数据 cache

1.读取指令：读取指令有效则无影响，读取指令失败，需要从存储器再次读取指令到 cache，直接读取 1 个块

写直达：1.读命中不访存 2.读数据失败，执行行填充 3.写数据成功，因为是写直达，因此要写入一个字（在本书也就是 4 个字节）到存储器中，注意此时是写有效的概率 4.写数据失败 a.写入内存一个字（4 个字节）b.执行行填充（1 个块）

写回：1.读命中不访存 2.读数据失败 a.替换脏块到存储器（这里其实就是对修改的数据进行备份的操作）b.行填充 3.写数据成功 不访存只写到 cache 中 4.写数据失败 a.替换脏块到存储器 b.行填充

写入内存 1 个字的原因是 lw,sw 指令进行数据交互都是一个字

不管是读操作还是写操作，只要失败(cache 不命中)，都需要执行行填充

注意写直达和写回的区别

写直达：读操作没什么特别的，但是只要是写操作，要同时写入内存 1 个字

写回：不论是读操作还是写操作，只要失败，都要进行脏块的替换

内置 FaseMATH 处理器

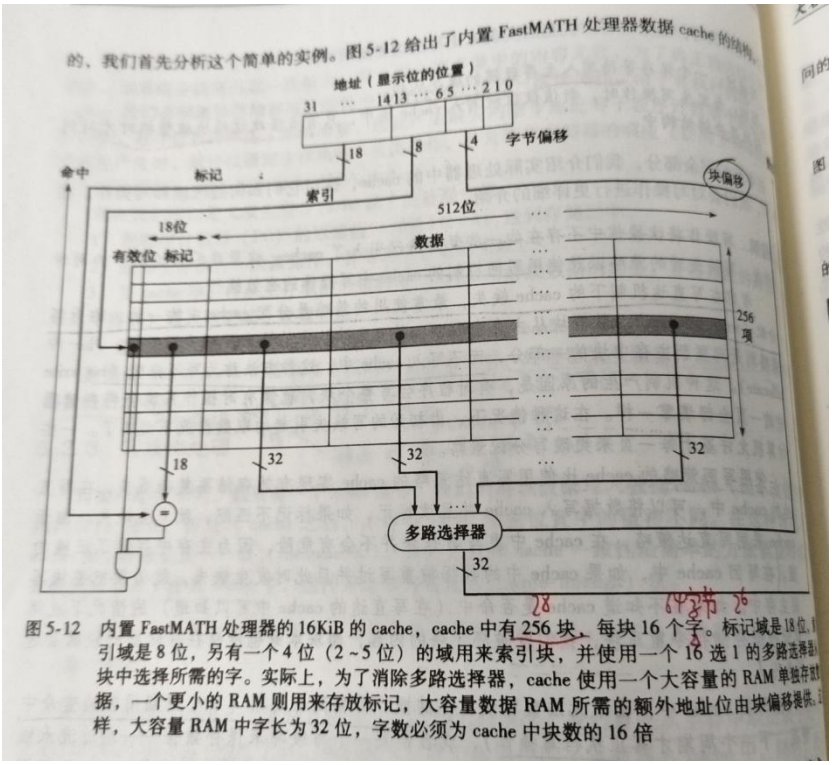


图 5-12 内置 FastMATH 处理器的 16KiB 的 cache，cache 中有 256 块，每块 16 个字。标记域是 18 位，索引域是 8 位，另有一个 4 位（2～5 位）的域用来索引块，并使用一个 16 选 1 的多路选择器从块中选择所需的字。实际上，为了消除多路选择器，cache 使用一个大容量的 RAM 单独存放数据，一个更小的 RAM 则用来存放标记，大容量数据 RAM 所需的额外地址位由块偏移提供。这样，大容量 RAM 中字长为 32 位，字数必须为 cache 中块数的 16 倍

这里主要想讲述一下 cache 的结构，每个块 16 个字，数据域有 16*4*8=512 位

16 个字 2 的 4 次方，字偏移有 4 位

一个字 4 个字节 2 的平方，字节偏移有 2 位
cache 有 256 块 2 的 8 次方，因此索引有 8 位
那么标记域就是 32-8-4-2=18 位

cache 性能的评估和改进

平均存储器访问时间(AMAT average memory access time)=命中时间+缺失率*缺失代价

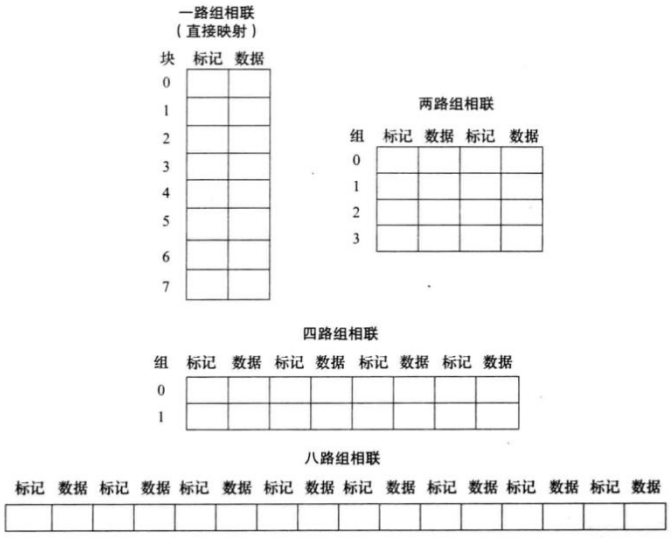
通过更灵活地放置块来减少 cache 的缺失

全相联：一个块可以放置在 cache 中的任何位置，因此需要检索所有的项，为了使检索更有效，它是由一个与 cache 中每个项都相关的比较器并行完成的，适合块数较少的 cache
组相联：每个组有 n 个位置可放，n 路组相联 cache 每个组有 n 块。存储器中每个块对应到 cache 中的唯一的组，并且可以放在这个组的任何一个位置上

都是要先求出块地址，字节地址不可直接用
块地址=字节地址/每块字节数
直接映射：存储块的位置（索引）=块地址 mod（cache 中的块数）
组相联：存储块的组=块地址 mod（cache 中的组数）求出在第几组
直接映射相当于 n=1,全相联相当于 n=n 只有一个组
块的总数=组数*相联度
解题时注意给的主存地址是字节地址、字地址还是块地址（一般不会直接给块地址）

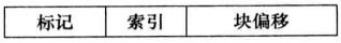
在 cache 中查找一个块

直接映射：索引数=cache 的块数，此时组数等于块数
组相联：索引数=组数，所以要留意索引数的变化，这会影响 cache 中标记位的计算



标记位大小与组相联

对于组相联的 cache，块偏移=字偏移+字节偏移



提高相联度需要更多比较器，同时 cache 块中的标记位数也需要增加。假设一个 cache 有 4 096个块，块大小为 4 个字，地址为 32 位，请分别计算在直接映射、两路组相联、四路组相联和全相联映射中，cache 的总组数以及总的标记位数。

答案

由于块大小为 16（$=2^4$）字节，32 位地址域中的 32 − 4 = 28 位用来提供索引和标记位。直接映射中组数和块数一样，由于 $\log_2(4\,096)=12$，因此有 12 位是索引位；因此总的标记位数是（28 − 12）×4 096 = 16 ×4 096 = 66Kb。

相联度每增加 1 倍，组数就会减少 1/2，因此用来索引 cache 的位数也要相应减 1，而标记位则是增 1。因此，对于一个两路组相联 cache，有 2 048 个组，总的标记位数为（28 − 11）× 2 ×2 048 = 34 ×2 048 = 70Kb。而四路组相联中组数为 1 024，那么总的标记位数为（28 − 10）×4 × 1 024 = 72 × 1 024 = 74Kb。

对于全相联 cache，只有一个有 4 096 个块的组，标记位是 28 位，因此总的标记位数是 28 ×4 096 × 1 = 115Kb。 □

总组数好计算就不过多赘述，来讨论标记位数

直接映射：4096 个组（2 的 12 次方），组数等于块数，一个块 16 个字节，因此字节偏移加字偏移一共 4 位，因此标记位数为(32-4-12)*4096

组相联：注意索引数=组数

两路组相联：2048 个组（2 的 11 次方），标记位数=(32-4-11)*2*2048

四路组相联：1024 个组（2 的 10 次方），标记位数=(32-4-10)*4*1024

全相联：1 个组含 4096 个块（2 的 12 次方），标记位数=(32-4)*4096

替换块的选择

当直接映射的 cache 发生缺失时，被请求的块只能放置于 cache 中的唯一位置，而原先占据的那个位置的块必须被替换掉

最常用的方法是最近最少使用(LRU)，被替换的块是最久没有使用的那一块

关于 cache 缺失的一些理解：

对于读：比如 lw 指令，会给出主存的地址，将主存地址与 cache 中的索引+标记进行比较，如果找到匹配的项，那么直接 cache 命中,反之缺失。但有时候就算找到匹配的项，但是 valid 位为 0，也就是意味着 cache 中存的值无效，与主存中不一样，那么也是 cache 缺失

对于写：比如 sw 命令，会给出主存的地址，将主存地址与 cache 中的索引+标记进行比较，如果找不到匹配的项，那么直接写缺失。但有时候就算找到匹配的项，但是 dirty 位为 1，即意味着 cache 中的数据被修改过，也与主存中的值不同，因此要做一次备份，执行行填充同时将 cache 中被修改的数据写回主存进行备份

总之 cache 只是为了快速访问主存，我们的核心目的还是访问主存中的数据

三.纠错码

一般采取偶校验，让二进制数中 1 的个数为偶数

汉明编码（SEC/DED）single error correction/double error detection

常见题型：

给了一个汉明码判断其是否出错，如果出错则纠正

汉明码的形式

位置		1	2	3	4	5	6	7	8	9	10	11	12
编码之后的数据位		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
奇偶校验位覆盖范围	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

设置哪些位为奇偶校验位？

将所有编号为 2 的整数次幂的位标记为奇偶校验位（1，2，4，8，16，…）。

这些奇偶校验位管哪一组

各检测位检测的数值

C_1 检测的 g_1 小组包含第 1，3，5，7，9，11，… 位置的二级制编码为X...XXX1

C_2 检测的 g_2 小组包含第 2，3，6，7，10，11，… 位置的二级制编码为X...XX1X

C_4 检测的 g_3 小组包含第 4，5，6，7，12，13，… 位置的二级制编码为X...X1XX

书上例题

例题

假定存在某个单字节数据 10011010_2。首先写出对应的汉明纠错码，然后把第 10 位取反，说明纠错码如何找到并纠正该错误。

答案

将校验位的位置空出来，12 位的码字_ _1 _001 _1010。

位置 1 检查第 1，3，5，7，9，11 位，为使该组为偶校验，我们应当把第 1 位填 0。

位置 2 检查第 2，3，6，7，10，11 位，为使该组为偶校验，我们在第 2 位填入 1。

位置 4 检查第 4，5，6，7，12 位，所以我们在第 4 位填入 1。

位置 8 检查第 8，9，10，11，12，所以我们在第 8 位填入 0。

最终得到的码字为 011100101010。把数据位第 10 位取反之后变成 011100101110。

校验位 1 为 0（**011100101110** 有 4 个 1，为偶性，故该组无错误）。

校验位 2 为 1（**011100101110** 有 5 个 1，为奇性，故该组某个位置上有错误）。

校验位 4 为 1（**011100101110** 有两个 1，为偶性，故该组无错误）。

校验位 8 为 1（**011100101110** 有 3 个 1，为奇性，故该组某个位置上有错误）。

校验位 2 和 10 不正确。因为 2 + 8 = 10，第 10 位肯定是错的。因此，我们将其翻转为 011100101010，即完成了纠错。 □

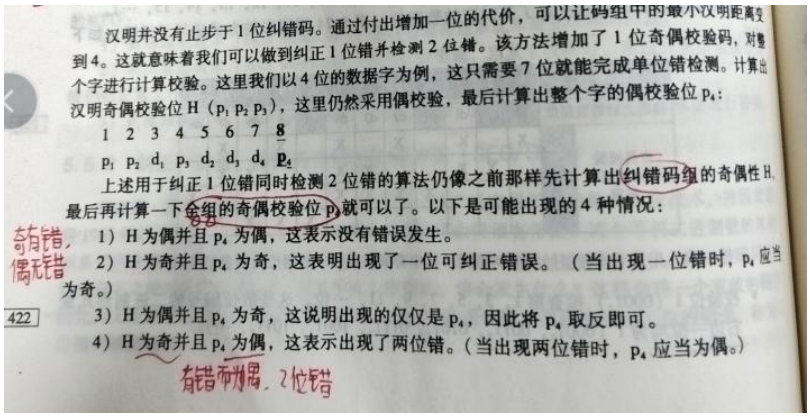
核心思想：分组校验

1.通过校验位提出这一组的各个位，判断其为奇性还是偶性，偶性无错误，奇性有错误

2.将为奇性的校验位相加即为出错位数

SEC/DED

修正 1 位，检测 2 位



汉明并没有止步于1位纠错码。通过付出增加一位的代价，可以让码组中的最小汉明距离变到4。这就意味着我们可以做到纠正1位错并检测2位错。该方法增加了1位奇偶校验码，对整个字进行计算校验。这里我们以4位的数据字为例，这只需要7位就能完成单位错检测。计算出汉明奇偶校验位H（p_1 p_2 p_3），这里仍然采用偶校验，最后计算出整个字的偶校验位 p_4：

1 2 3 4 5 6 7 **8**

p_1 p_2 d_1 p_3 d_2 d_3 d_4 p_4

上述用于纠正1位错同时检测2位错的算法仍像之前那样先计算出纠错码组的奇偶性H，最后再计算一下全组的奇偶校验位 p_4 就可以了。以下是可能出现的4种情况：

1）H为偶并且 p_4 为偶，这表示没有错误发生。

2）H为奇并且 p_4 为奇，这表明出现了一位可纠正错误。（当出现一位错时，p_4 应当为奇。）

3）H为偶并且 p_4 为奇，这说明出现的仅仅是 p_4，因此将 p_4 取反即可。

4）H为奇并且 p_4 为偶，这表示出现了两位错。（当出现两位错时，p_4 应当为偶。）

422

对全组求出一个奇偶校验位 p，求出纠错码组 H（由各个校验位组成）的奇偶性

纠错码组 H 为偶则一定无错，H 为奇 p 的奇偶性决定错误的个数

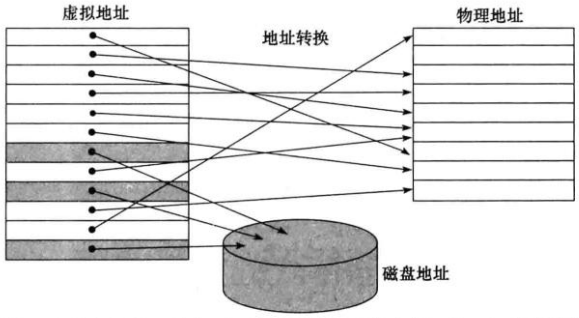
四.虚拟存储器

基本概念

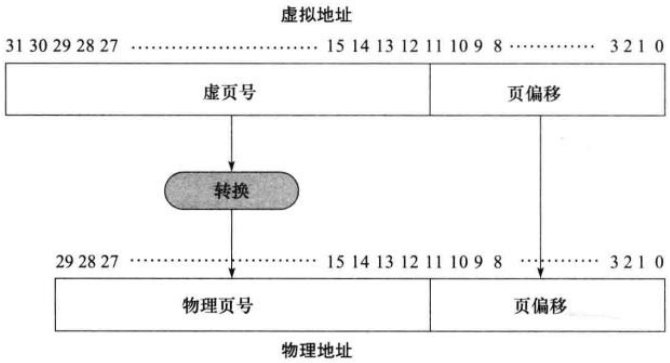
缺页：访问的页不在主存储器中。

虚拟地址：虚拟空间的地址，当需要访问主存时需要通过地址映射转换为物理地址。

地址转换：也称为地址映射。在访问内存时将虚拟地址映射为物理地址的过程。



5-25 在存储器中，主存中的块（称为页）从一组地址（称为虚拟地址）映射到另一组地址（称为物理地址）。访问主存使用物理地址，而处理器产生虚拟地址。虚拟存储器和物理存储器都被划分成页，因此一个虚页被映射到一个物理页。当然，一个虚页也可能不在主存中，因此无法映射到物理地址；在这种情况下，页就被存在磁盘上。物理页可以被两个指向相同物理地址的虚拟地址共享。这种方法用来使两个不同的程序共享数据或代码



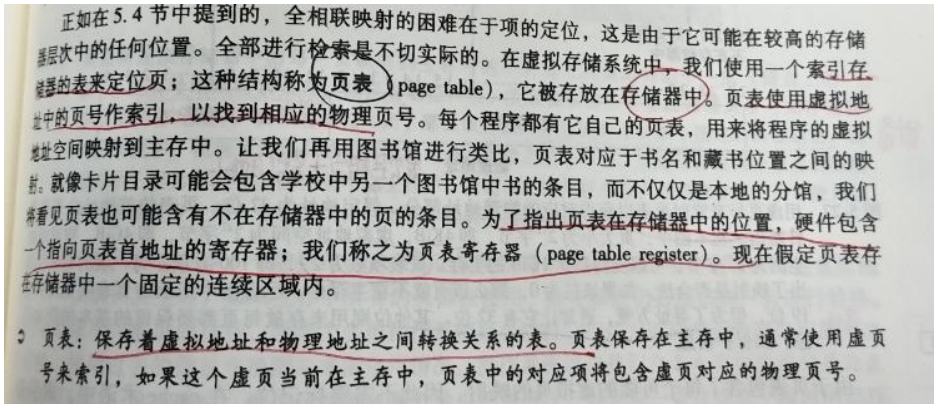
现今所有的虚拟存储器系统将程序重定位为一组固定大小的块（页），因此减少了寻找主存中连续的块来放置程序的必要

页偏移域的位数决定了页的大小，在此图中页的大小为 4KiB

为了弥补较长的访问时间，页应该足够大。能降低缺页率的组织结构具有吸引力，这里用到的主要技术时允许存储器中的页以全相联方式放置。由于写的时间太长，虚拟存储系统都采

用写回机制

页的存放与查找



正如在5.4节中提到的，全相联映射的困难在于项的定位，这是由于它可能在较高的存储器层次中的任何位置。全部进行检索是不切实际的。在虚拟存储系统中，我们使用一个索引存储器的表来定位页；这种结构称为页表（page table），它被存放在存储器中。页表使用虚拟地址中的页号作索引，以找到相应的物理页号。每个程序都有它自己的页表，用来将程序的虚拟地址空间映射到主存中。让我们再用图书馆进行类比，页表对应于书名和藏书位置之间的映射。就像卡片目录可能会包含学校中另一个图书馆中书的条目，而不仅仅是本地的分馆，我们将看见页表也可能含有不在存储器中的页的条目。为了指出页表在存储器中的位置，硬件包含一个指向页表首地址的寄存器；我们称之为页表寄存器（page table register）。现在假定页表存在存储器中一个固定的连续区域内。

- 页表：保存着虚拟地址和物理地址之间转换关系的表。页表保存在主存中，通常使用虚页号来索引，如果这个虚页当前在主存中，页表中的对应项将包含虚页对应的物理页号。

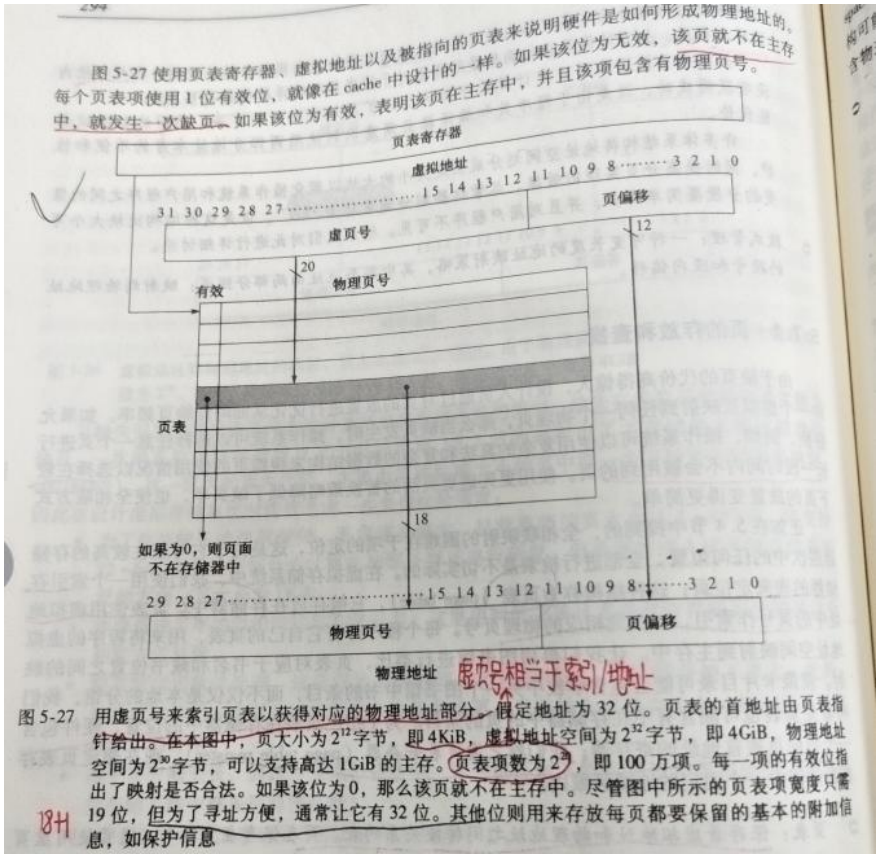


图5-27 用虚页号来索引页表以获得对应的物理地址部分。假定地址为32位。页表的首地址由页表指针给出。在本图中，页大小为2^{12}字节，即4KiB，虚拟地址空间为2^{32}字节，即4GiB，物理地址空间为2^{30}字节，可以支持高达1GiB的主存。页表项数为2^{20}，即100万项。每一项的有效位指出了映射是否合法。如果该位为0，那么该页就不在主存中。尽管图中所示的页表项宽度只需19位，但为了寻址方便，通常让它有32位。其他位则用来存放每页都要保留的基本的附加信息，如保护信息

我的理解：页表有1位是有效位，剩下的存的是物理页号，但是为了寻址方便所以通常让其有32位，而虚页号相当于索引，用来检索页表

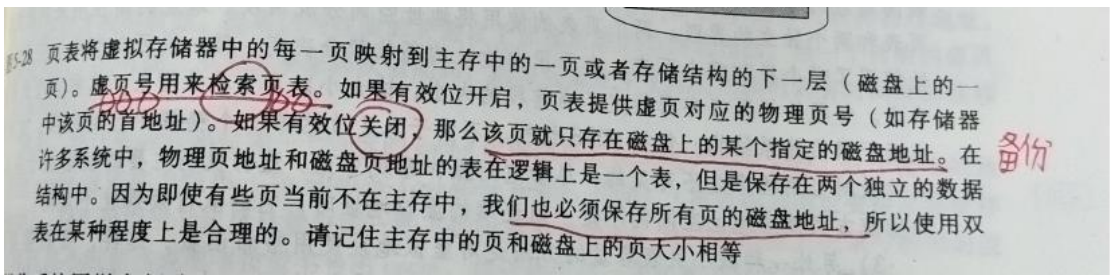
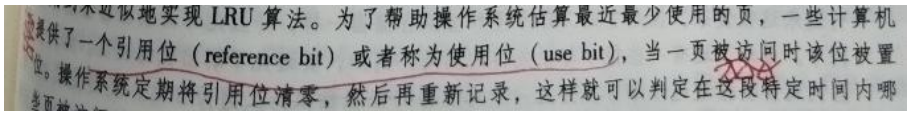


图5-28 页表将虚拟存储器中的每一页映射到主存中的一页或者存储结构的下一层（磁盘上的一页）。虚页号用来检索页表。如果有效位开启，页表提供虚页对应的物理页号（如存储器中该页的首地址）。如果有效位关闭，那么该页就只存在磁盘上的某个指定的磁盘地址。在许多系统中，物理页地址和磁盘页地址的表在逻辑上是一个表，但是保存在两个独立的数据结构中。因为即使有些页当前不在主存中，我们也必须保存所有页的磁盘地址，所以使用双表在某种程度上是合理的。请记住主存中的页和磁盘上的页大小相等

在存储器结构中有一个很重要的思想就是备份，如果数据不在高一级的存储器，那么在更低级的存储器中一定可以找到这个数据。有时候一些替换或者写回操作其实也是为了让改变的数据有个备份

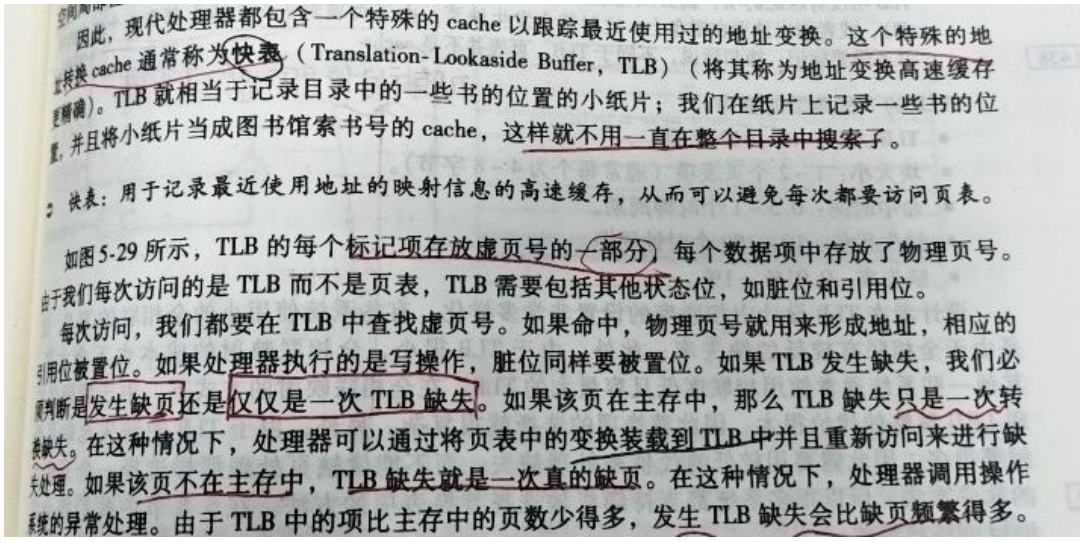
引用位



脏位的存在也是为了备份，让一些修改过的数据不丢失

需要被复制写回。为了追踪读入主存中的页是否被写过，可以在页表中增加一个脏位（dirty bit）。当页中任何字被写时就将这一位置位。如果操作系统选择替换某一页，脏位指明了在把该页所占用的主存让给另一页之前，是否需要将该页写回磁盘。因此，一个修改过的页也通常被称为脏页（dirty page）。

TLB


因此，现代处理器都包含一个特殊的 cache 以跟踪最近使用过的地址变换。这个特殊的地址转换 cache 通常称为快表（Translation-Lookaside Buffer，TLB）（将其称为地址变换高速缓存更精确）。TLB 就相当于记录目录中的一些书的位置的小纸片；我们在纸片上记录一些书的位置，并且将小纸片当成图书馆索书号的 cache，这样就不用一直在整个目录中搜索了。

快表：用于记录最近使用地址的映射信息的高速缓存，从而可以避免每次都要访问页表。

如图 5-29 所示，TLB 的每个标记项存放虚页号的一部分，每个数据项中存放了物理页号。由于我们每次访问的是 TLB 而不是页表，TLB 需要包括其他状态位，如脏位和引用位。

每次访问，我们都要在 TLB 中查找虚页号。如果命中，物理页号就用来形成地址，相应的引用位被置位。如果处理器执行的是写操作，脏位同样要被置位。如果 TLB 发生缺失，我们必须判断是发生缺页还是仅仅是一次 TLB 缺失。如果该页在主存中，那么 TLB 缺失只是一次转换缺失。在这种情况下，处理器可以通过将页表中的变换装载到 TLB 中并且重新访问来进行缺失处理。如果该页不在主存中，TLB 缺失就是一次真的缺页。在这种情况下，处理器调用操作系统的异常处理。由于 TLB 中的项比主存中的页数少得多，发生 TLB 缺失会比缺页频繁得多。

TLB 可以理解为页表的 cache，所以书中把 TLB 说成记录目录中一些书的位置的小卡片很形象

TLB 缺失有两种情况：1.仅仅是一次 TLB 的缺失 2.缺页（缺页有一些相关处理）

TLB 的结构

之外写 TLB 的必要。

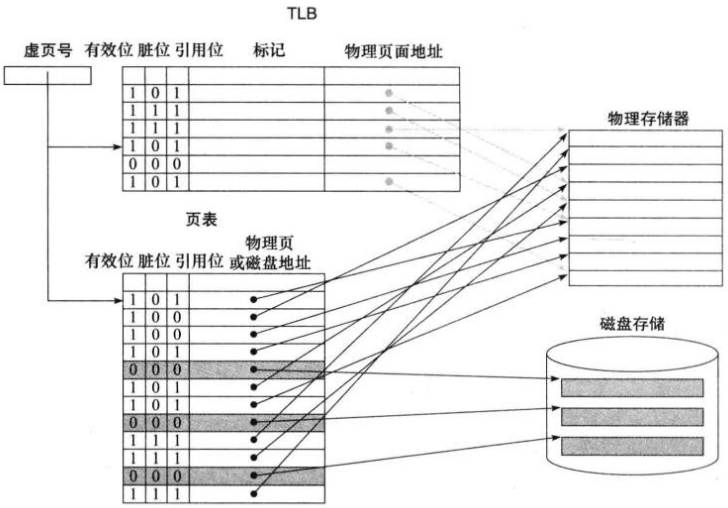


图 5-29 TLB 作为页表的 cache，用于存放映射到物理页中的那些项。TLB 包含了页表中虚页到物理页映射的一个子集。TLB 映射以灰线显示。因为 TLB 是一个 cache，它必须有标记域。如果一个页在 TLB 中没有匹配的项，就必须检查页表。页表或者提供该页的物理页号（可用来创建一个 TLB 项），或者指出该页在磁盘上，这时就会发生缺页。由于页表对于每个虚页都有一个相应的项，并不需要标记；换句话说，不同于 TLB，页表并不是 cache

38

TLB 包含了页表中虚页到物理页映射的一个子集
页表中每个虚页都有一个相应的项
TLB 的标记位存的是虚页号，用来和页表中的进行比对

FastMATH TLB

为了弄清楚这些想法是如何实际应用到处理器中的，我们来进一步研究内置 FastMATH 的 TLB。存储系统页大小为 4KiB，地址空间为 32 位，因此，虚页号长为 20 位，如图 5-30 顶部所示。物理地址和虚拟地址长度相等。TLB 包含了 16 个项，采用全相联映射，由指令和数据访问共享。每个表项宽为 64 位，包含了 20 位的标记位（作为该 TLB 表项的虚页号）、相应的物理页号（也是 20 位）、一个有效位、一个脏位以及一些其他管理操作位。与大多数 MIPS 系统类似，它采用软件来处理 TLB 缺失。

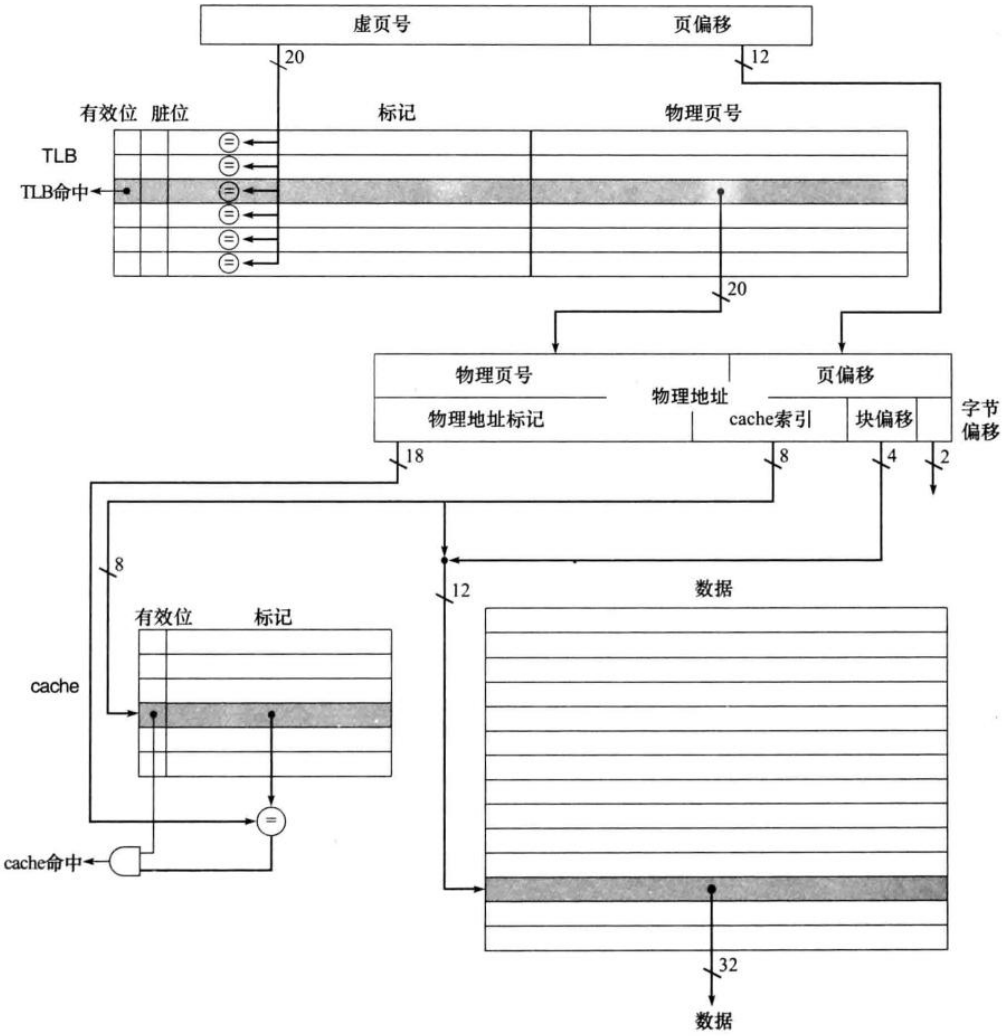


图 5-30　内置 FastMATH 中 TLB 和 cache 实现了从虚拟地址到数据项的转换过程。本图描述了 TLB 和数

注意存储器的层次结构

TLB 缺失，回到页表　页表缺失，则缺页，回到磁盘

虚拟存储器和 cache 系统就像一个层次结构一样共同工作，因此除非数据在主存中，否则它不可能出现在 cache 中。操作系统帮助管理该层次结构，当它决定将某一页移动到磁盘上去时，就在 cache 中将该页中的内容刷新。同时操作系统修改页表和 TLB，而后尝试访问该页上的数据都将发生缺页

任何决定页的访问权限的位不仅要包含在页表中，还要包含在 TLB 中