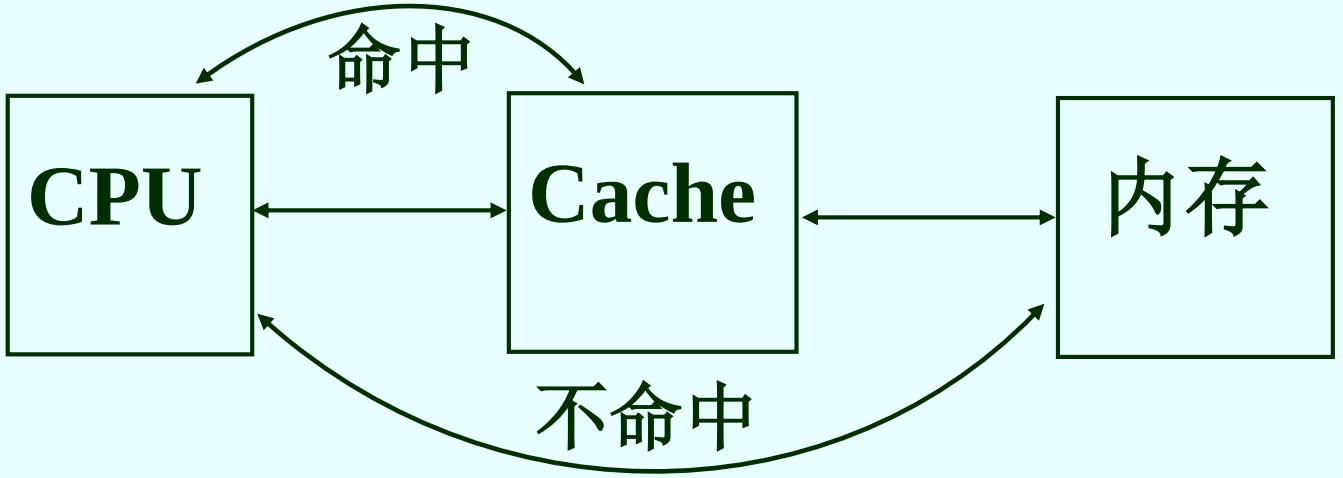
附: 高速缓冲存储器(Cache)

① 概述

- Cache的起源和作用
- MOS工艺半导体存储器
- 双极型高速存储器
- 零等待时间
- Cache与内存的工作流程



② 程序特征与Cache的读写

怎样判断哪些数据和程序代码是经常被访问的？

“程序执行的局部性规律”

描述: 那些最近被使用过的数据和指令被频繁再次使用的规律。

统计表明: 在一个程序的执行过程中, 执行时间的90％花在约10％的程序代码上。

统计值例:

对Gcc编译器、Spice(CAD电路分析软件)、以及TeX(文本处理软件)三个典型测试程序的测试结果, 相应的比例分别为:13%、9.5％、9.3％

包括: 时间局部性规律和空间局部性规律

- 时间局部性规律:

  程序执行过程中近期被访问的信息可能很快将被再次访问; (典型情况: 程序中大量的循环)

- 空间局部性规律:

  那些与被访问的地址相邻近的信息可能很快被访问;

  (典型情况: 程序顺序执行)

按照“局部性规律”, 数据/代码被访问后, 该数据/代码以及临近的数据/代码近期被再次访问的概率, 大于近期未被访问的数据和代码被访问的概率。

因此，一个数据或代码被访问后，就认为是经常会被访问的数据或代码，将其存入Cache，使Cache有更高的命中率

程序执行的局部性规律具有普适性，如：

- LRU淘汰算法
- 动态分支预测技术
- 内存中存放近期被访问的页表，大多数情况都能够满足应用需求(80386/486处理器的TLB表命中率超过90％)。

③ 命中率

- Cache容量与命中率
- Cache的结构与命中率
- 所处理的数据量大小与命中率

④ Cache的淘汰算法

- 先进先出算法
- 随机淘汰
- LFU算法(访问次数最少的内容被淘汰)
- LRU(最近最久未使用淘汰)算法

⑤ Cache命中与否的判断

通过判断该数据/代码的地址是否在Cache中，而不是判断该数据/代码本身。

高速缓存要解决的主要问题:

- 命中率
- 速度
- 一致性

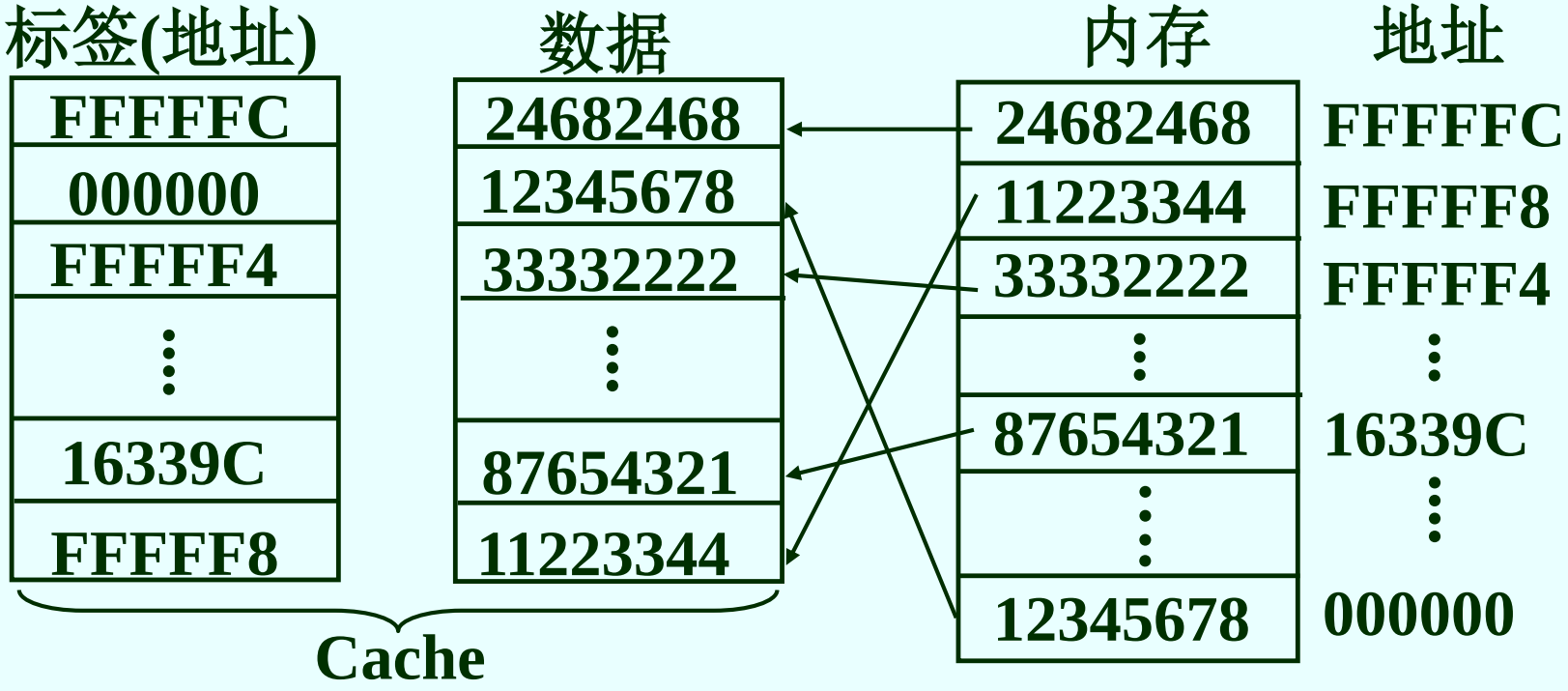
一、高速缓存的结构及工作原理

高速缓存的三种主要结构:

- 全关联式高速缓存
- 直接对应式高速缓存
- 多组关联式高速缓存

1、全关联式高速缓存

高速缓存由两部分组成：地址部分(称为标签)和数据部分, 如下图(假设地址24位, 字长32位):



假设: 处理器读取FFFFFC地址单元

首先用该地址查找Cache, 不命中。将数据24682468读入处理器, 同时将地址FFFFFC写入Cache标签字段, 对应数据24682468写入Cache数据字段。

处理器对存贮器的访问过程:

读操作:

用CPU发出的地址查找Cache的标签字段, 如果存在该地址(命中Cache), 读取Cache中该地址对应的数据部分; 若不存在该地址(不命中), 从主存单元中读取数据, 并同时将该地址和对应的数据分别写入Cache的标签字段和数据部分。

上述过程符合“程序执行的时间局部性规律”, 如果下次再访问该地址, 即可命中。

写操作:

CPU写数据时, 首先查找Cache , 有该地址则命中, 将该数据写入Cache的该地址单元。

全关联式的优缺点:

优点: 直观、结构简单, Cache中数据的存放位置灵活

缺点: 速度慢， Cache越大, 速度越慢。

平均查找次数为Cache容量的一半。

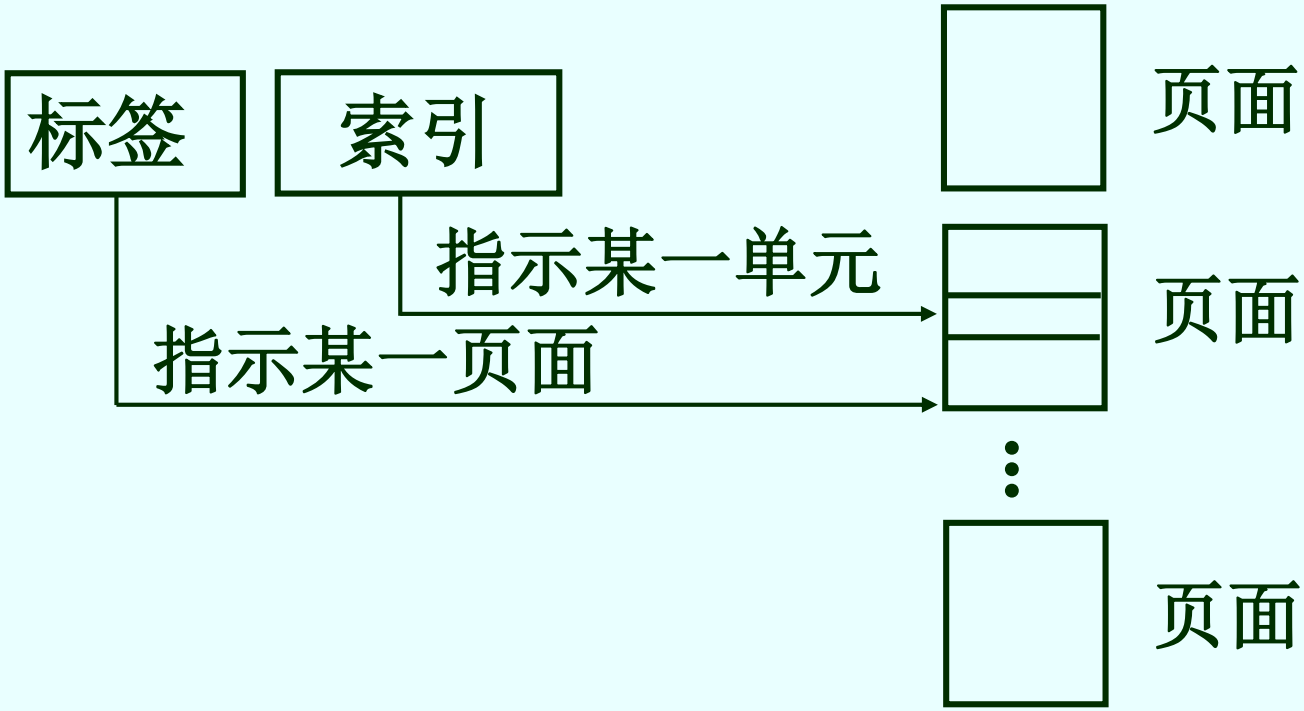
2、直接对应式高速缓存

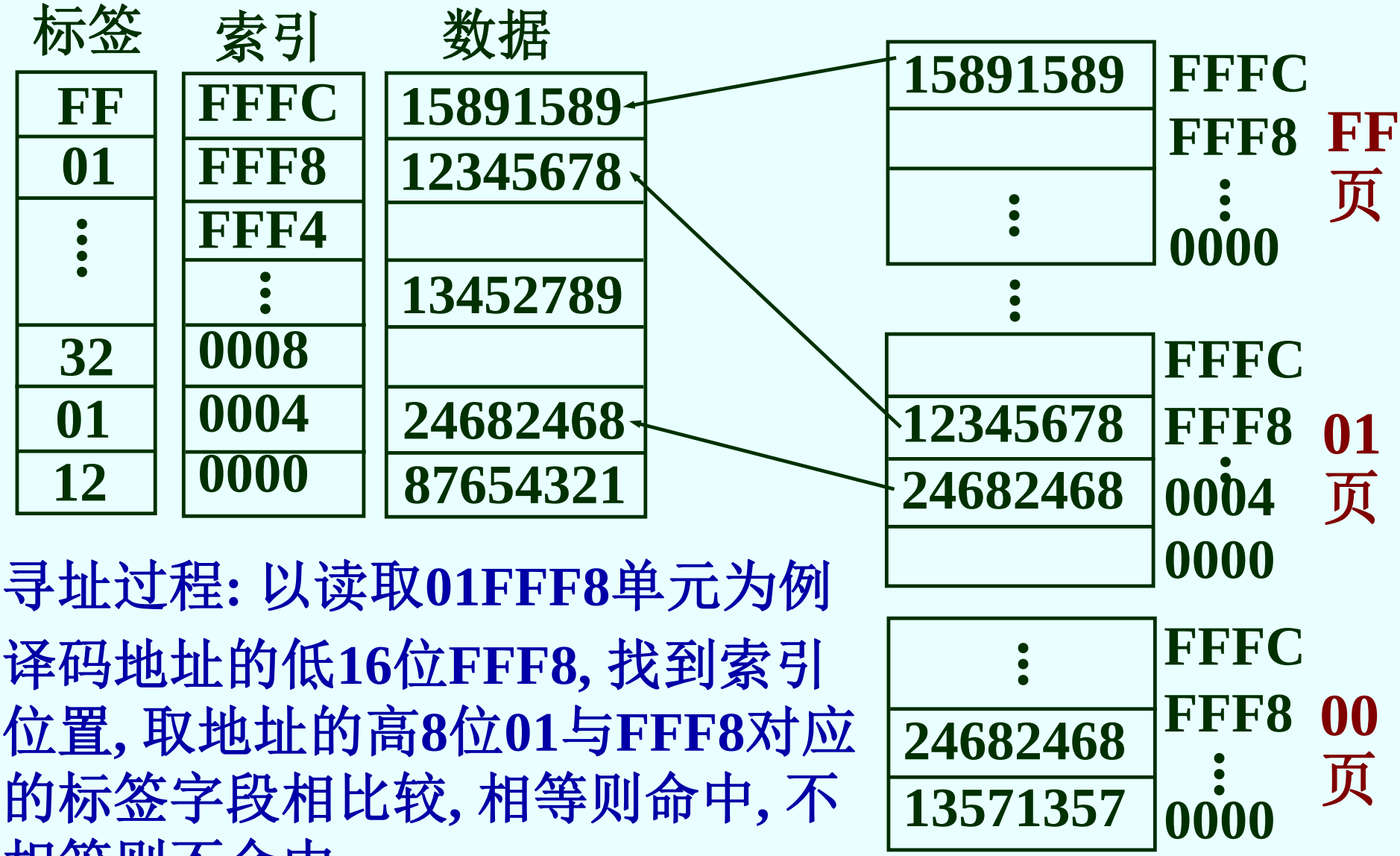
① 结构及工作原理

假设: 主存容量16M(24位地址)，Cache容量为64K, 字长32位。

- 将16M内存空间逻辑上分为每64K为一个页面，共计可分为16M÷64K=256(个页面);
- 将内存空间的地址看作一个二维的地址，即页号和页内地址;
- 与内存二维地址相对应, Cache中的地址也分为页号和页内地址，分别称为标签字段(即页号)和索引字段(即页内地址) 。

- 对于所有总共256页，需要8位地址作为标签(地址的高8位A_{23}~A_{16})，指明访问哪一个页面；每页64K。
- 需要16位地址作为页内地址A_{15}~A_0，指明访问一页的哪个单元。如下图所示：





寻址过程: 以读取01FFF8单元为例

译码地址的低16位FFF8, 找到索引位置, 取地址的高8位01与FFF8对应的标签字段相比较, 相等则命中, 不相等则不命中。

② 特征说明

- 所有页面的相同页内地址竞争同一标签字段

  即：只要索引值相同，256个页面的页地址竞争该索引值对应的标签字段;

- 索引不变，只需修改标签字段

  索引字段保持不变, 变化的是标签(即页号)和数据部分。由于索引值不变，因此索引值不需要逐一比较，一次译码即可找到;

- 只比较一次

  用CPU给出地址的高8 (即页号)与标签字段比较, 相等则命中, 不相等则不命中。

③ 优缺点说明

- 优点

  一次比较即可判断命中与否，速度快。

- 缺点
  - 数据位置固定，灵活性差；
  - 由于“程序执行的时间局部性规律”，命中率可能很低。

如：反复且轮流读取索引值相同但标签不同的两个或多个单元(比如01FFF8和12FFF8，可能出现命中率为0 (如果采用全关联式结构，则每次都能命中)。

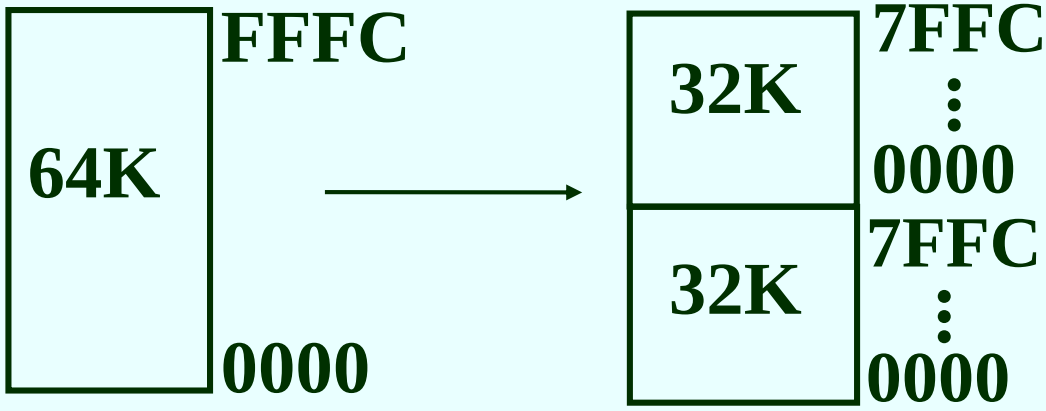
3、多组关联式高速缓存

包括双组关联、四组关联、八组关联等。

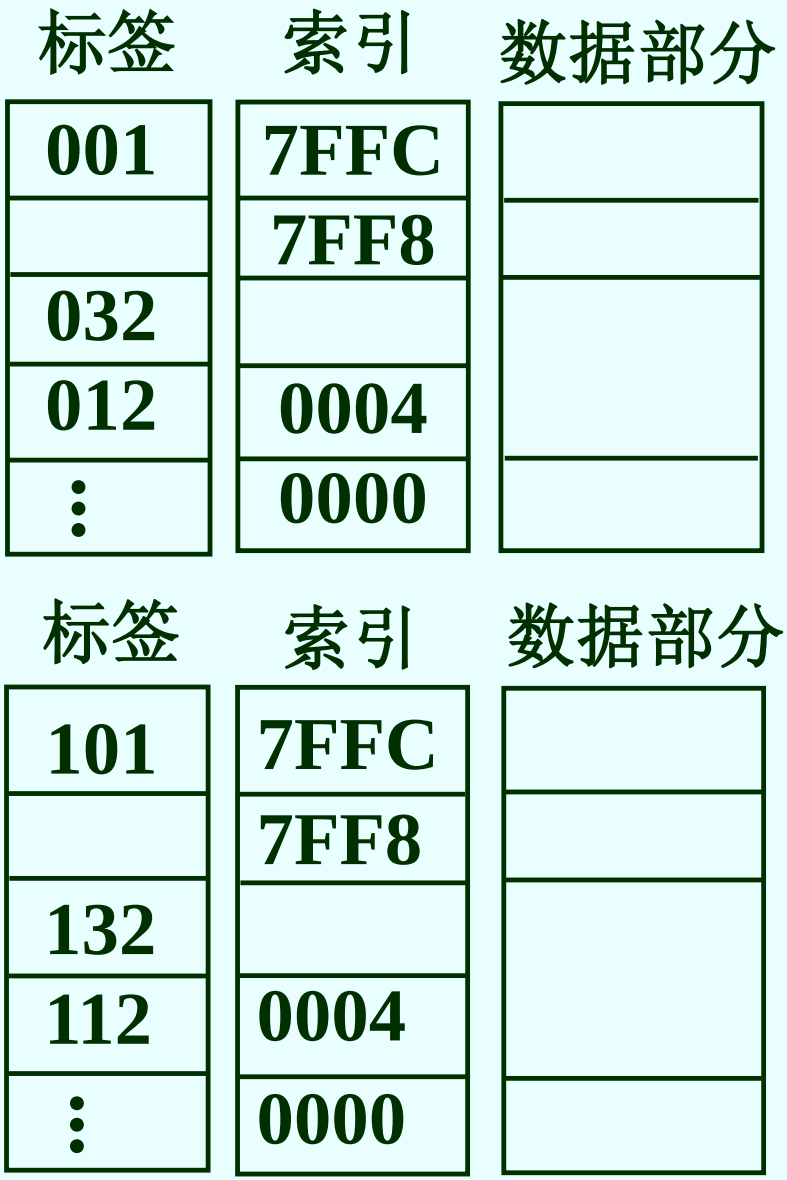
① 结构及工作原理(以双组关联为例)

以双组关联为例，假设: 内存16M，Cache 64K。

将64K的页面逻辑上再分为两个32K。



对32K页面, 页内地址只需15位, 共512个页面, 需9位地址作标签。相对于直接对应式的一个64K的页面来说, 双组关联的构造形式如下图所示:



说明：

① 分为两组后，相当于同一索引字段竞争两个标签位置，使竞争减少一半；

② 如果采用四组关联，则应将64K分为4个16K，依此类推；

③ 如果一直分下去直到不能再分，多组关联式将演变为全关联式。

② 优缺点比较

优点:

与直接对应相比，由比较一次变为比较两次，同一索引值由竞争同一标签位置变为竞争两个标签位置，竞争减少一半。因此，命中率较直接对应式高。

缺点:

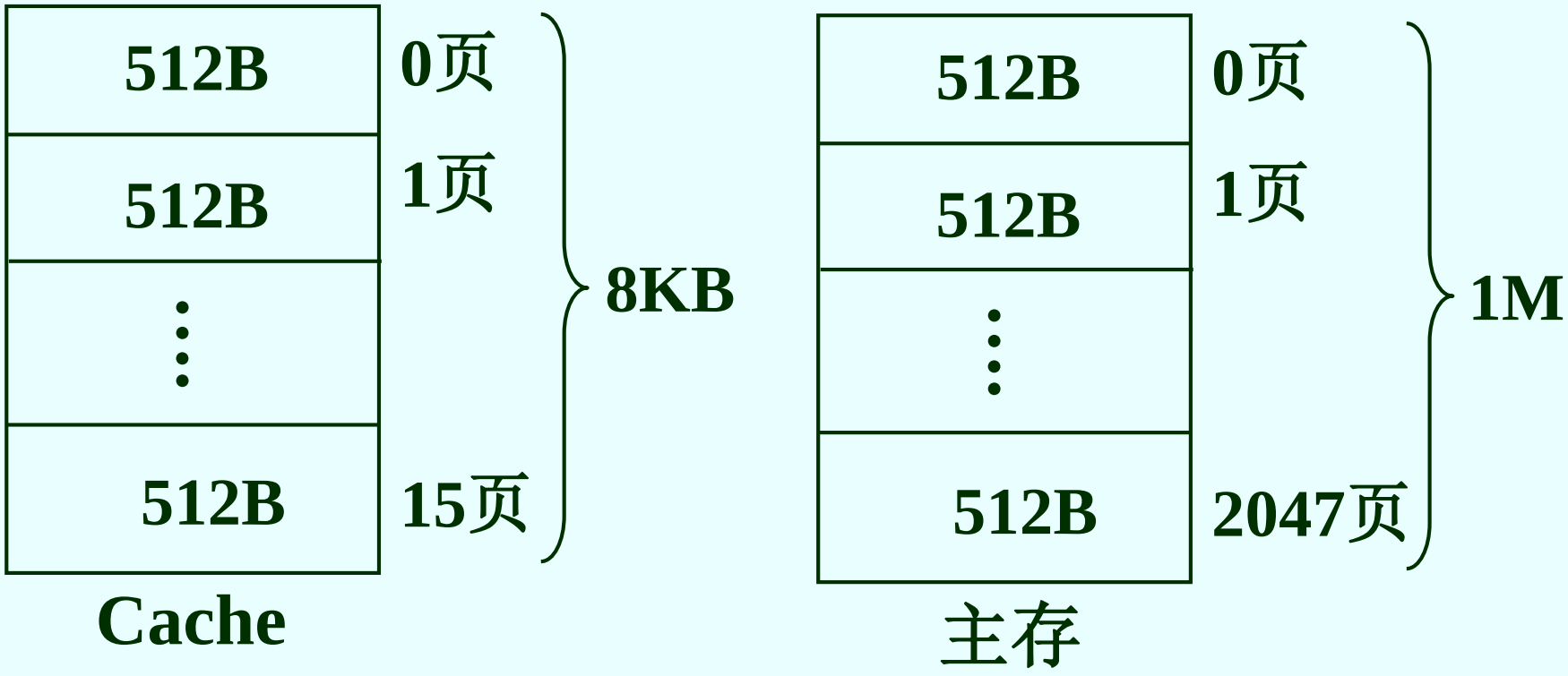
速度较直接对应式慢，但比全关联式快。

优缺点都在前两种结构之间。

基本结构的演进

将主存和Cache划分为若干大小相同的页(或称块)。

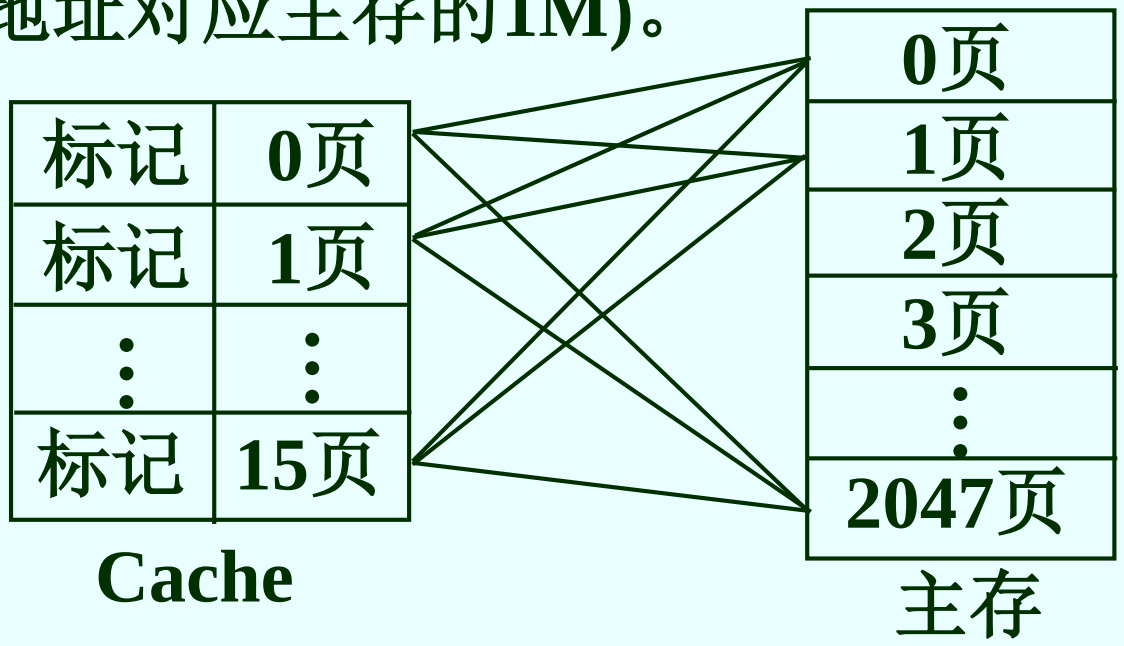
比如: 主存容量1MB, 分为2048页, 每页512B;
Cache容量8KB, 分为16页, 每页512B。



1、全关联式结构

- 主存的每一页可以直接映像到Cache的任一页;
- 主存地址分为页标记(或标签)和页内地址; 共计2048个页, 需要11位作为页标记; 每一页512个字节, 需要9位页内地址。Cache也需要一个11位的标记作为页号(共计20位地址对应主存的1M)。

如图所示:



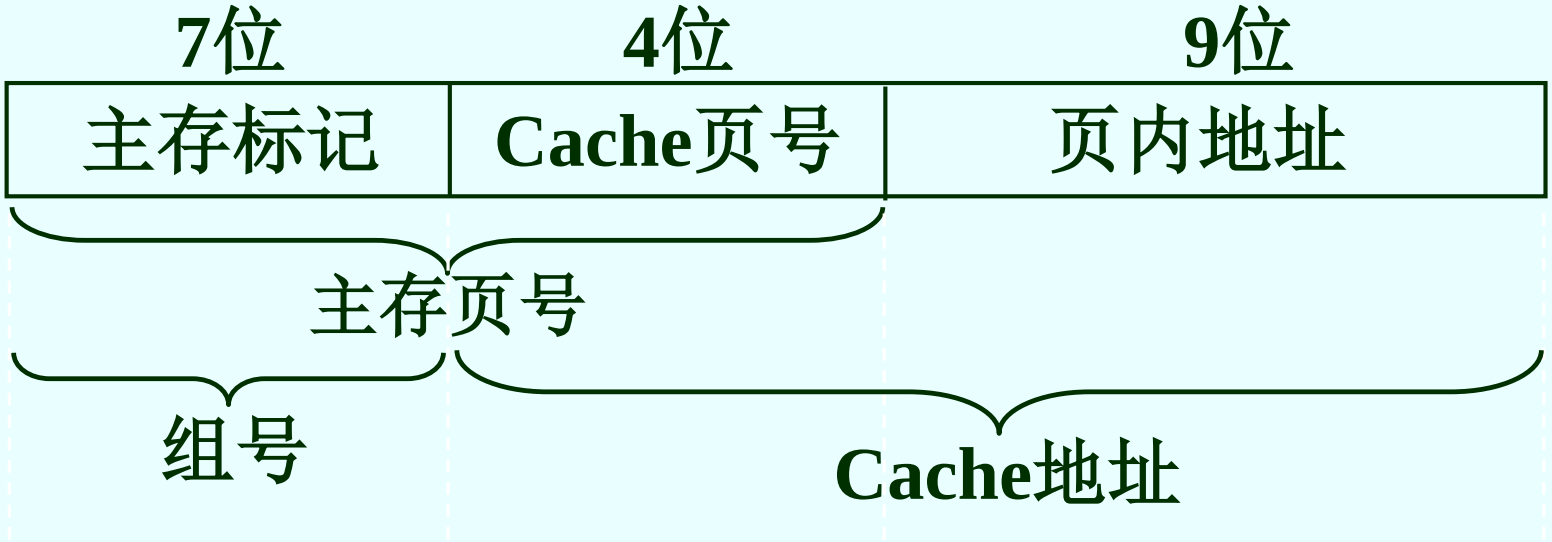
主存页标记	页内地址

2、直接对应式结构

每个主存页只能复制到某个固定的Cache页中(与前述基本的直接对应式结构类似, 必须页号(即标签)相同, 页内地址(即索引)才有效)。

将共计2048个内存页面再分为128组, 每组16个页面, 分别与Cache的16个页对应。

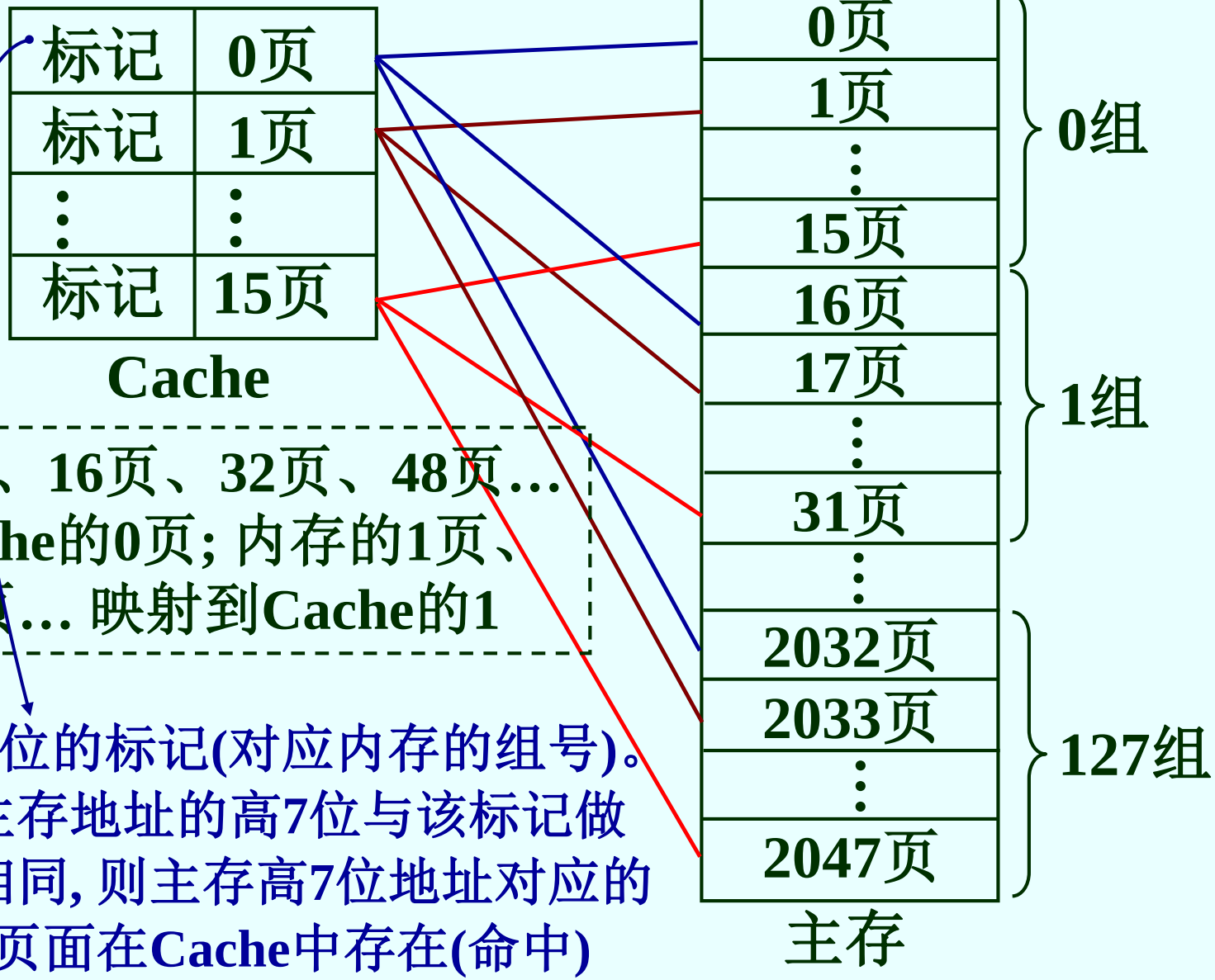
由于分组, 一个内存地址逻辑上被分成三个部分:



对Cache, 同样设置一个7位的标记(对应内存的组号)。
访存时, 将主存地址的高7位与Cache的7位标记做比较, 如果相同, 意味着主存高7位地址对应的组中的某个页面在Cache中存在(命中)

其对应方式是: 内存页号做“模16”运算的结果对应Cache页号。
因此, 内存的0页、16页、32页、48页…映射到Cache的0页; 内存的1页、17页、33页、49页…映射到Cache的1页…。

如下图所示:



内存的0页、16页、32页、48页…映射到Cache的0页; 内存的1页、17页、33页… 映射到Cache的1页…

Cache中的7位的标记(对应内存的组号)。访存时, 将主存地址的高7位与该标记做比较, 如果相同, 则主存高7位地址对应的组中的某个页面在Cache中存在(命中)

3、组关联式结构

以直接对应式为基础, 将高速缓存的页再分组。比如两个页面为一组(双组关联)。

设内存页号j; Cache的组数量u, 则内存块映射到Cache的哪一组的关系式:

$$j \bmod u$$

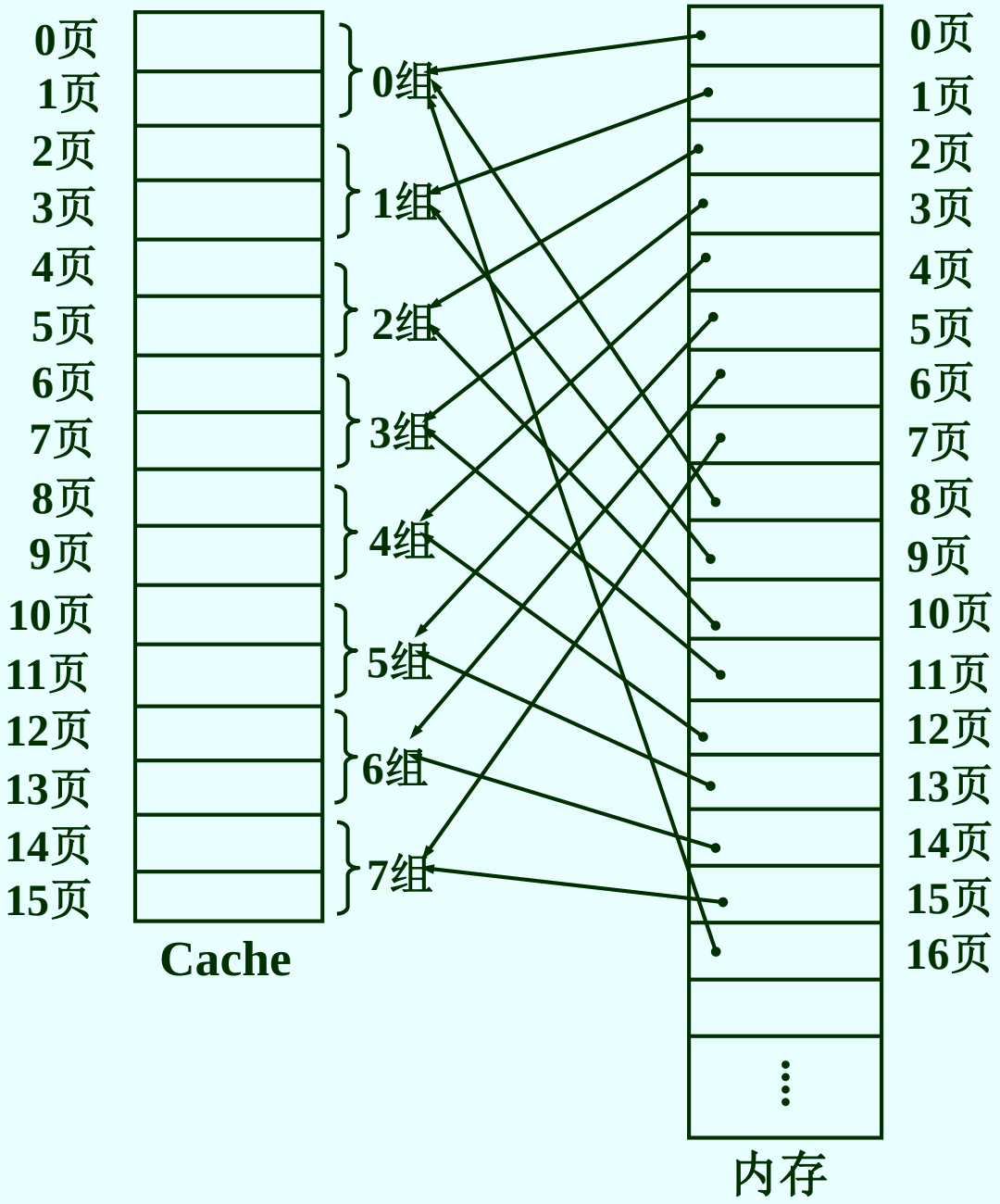
该公式的映射关系如下图所示:

j: 内存页号
u: Cache的组数量

内存的页映射到Cache的哪一组的关系式:

j mod u

Cache的一个组有两个页可供查找, 判断是否命中。



例: 32个字节为一个页，Cache共计16页, 采用双组关联(两页为一组), 按字节编址。问内存129号单元映射到Cache的哪一组?

解: 因为每一块32个字节, 因此129号单元在内存的4页。4 mod 8=4 (组)

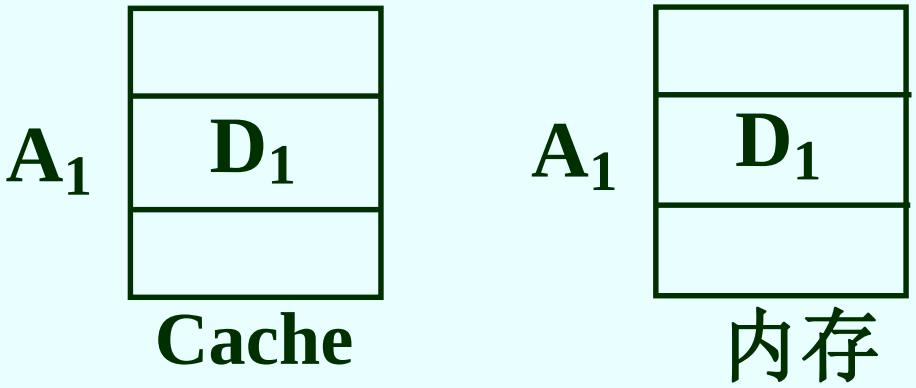
二、高速缓存的数据一致性

1、高速缓存内容丢失

(1) 丢失原因

在正常工作情况下，Cache系统中的数据有两份，一份在Cache中，另一份在主存的对应地址单元中，两份内容是一致的。

比如从主存的A_1单元读取一个数据D_1后，Cache与主存的情况是：



Cache内容与它对应的主存单元内容不一致例:

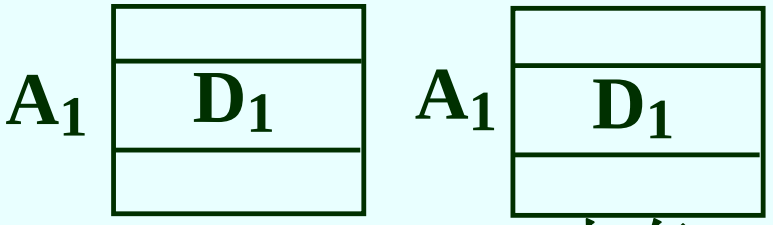
Cache与主存的当前情况:

计算函数 M= (x+y)/f(z)

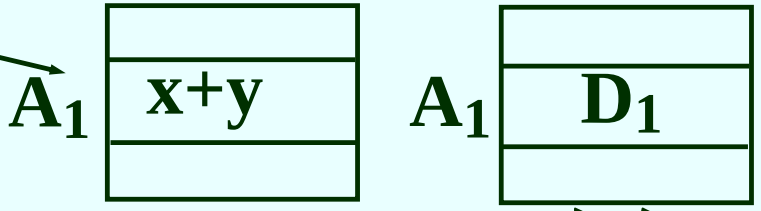
① 计算出中间结果 x+y;

② 将x+y存入A_1单元(命中), Cache与内存不再一致;

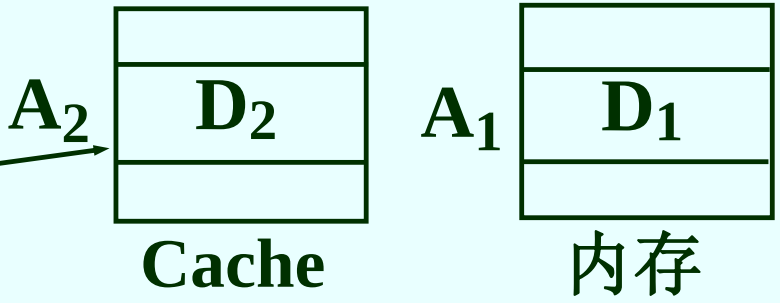
③ 此时, 如果Cache控制器进行Cache更新, 并正好淘汰A_1单元 , 则新数据和地址(假设是D_2和A_2)覆盖x+y和A_1, 导致x+y从Cache中消失(主存也无x+y);



Cache与主存初始状态



存入中间结果后的状态



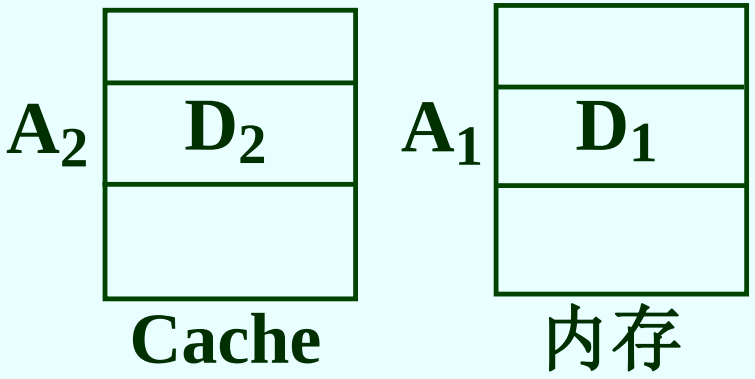
(x+y)被淘汰后的状态

④ 计算出f(z)；

⑤ 从A_1单元读中间结果(x+y)；

⑥ 试图计算 M= (x+y)/f(z)

得到错误结果 M= D_1/f(z)



(x+y)被淘汰后的状态

(2) 解决方法

① 直写方式(Write through 通写、透写、直写)

基本思想:

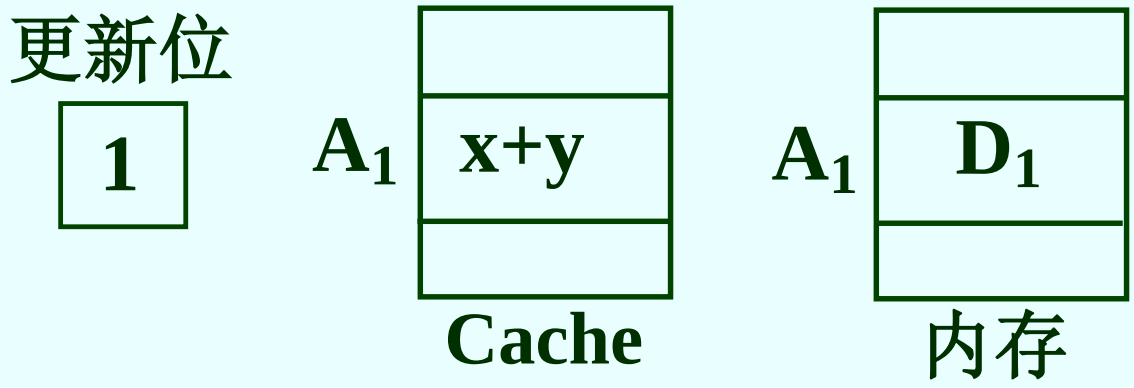
写命中Cache时, 同时将该数据写入对应的主存单元, 使Cache和主存的同一地址中内容保持一致。

如上例中, 将x+y存入Cache A_1单元的同时, 将该数据写入内存的A_1单元。

② 回写方式(Write-Back — 写回)

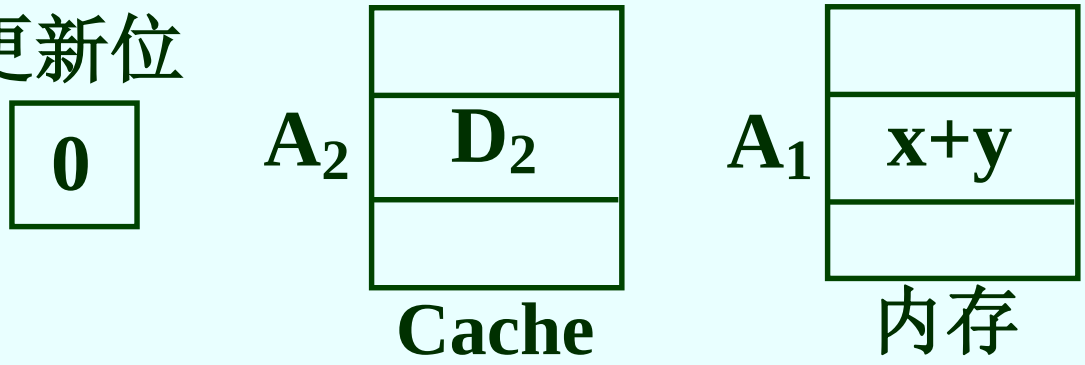
对Cache的每一数据块, 增加了一个“更新位”。当写命中Cache时, 不将该数据立即写入内存, 只将“更新位”置“1”, 用来指明当前Cache内容与对应的主存单元是不一致的。

如下图所示:



如果地址A_1及内容x+y要被新内容(地址A_2和内容D_2)所淘汰，则首先检查A_1的“更新位”，如果为1，表明当前Cache内容与对应的主存内容不一致；则先将A_1内容(如上例的x+y)写入主存A_1单元，然后才将地址A_2和内容D_2写入Cache，同时将“更新位”清0。

如下图所示：



结论：中间结果x+y在Cache被淘汰，但在内存中仍然存在，没有丢失。

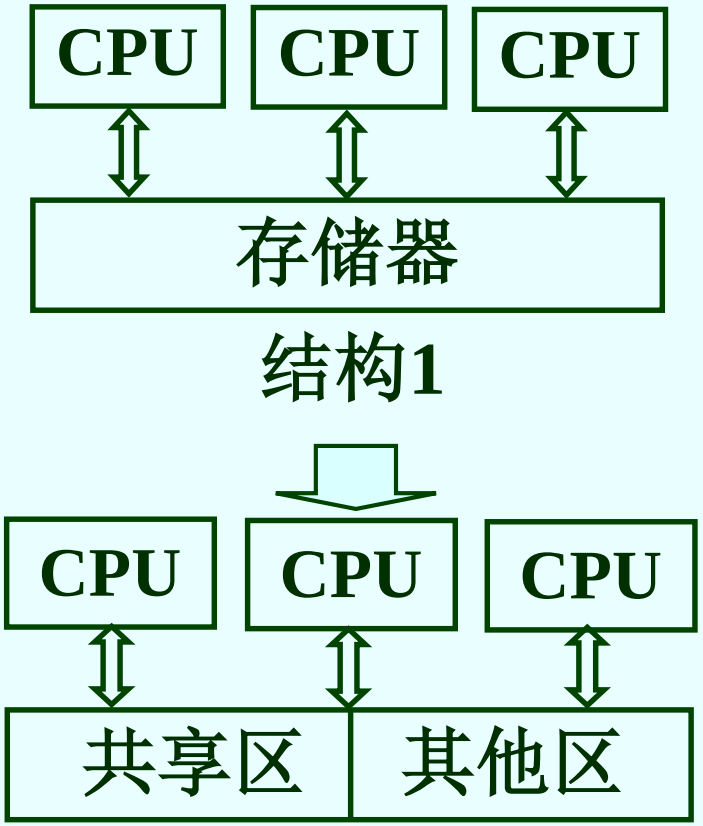
与直写相比，减少了写内存的次数。

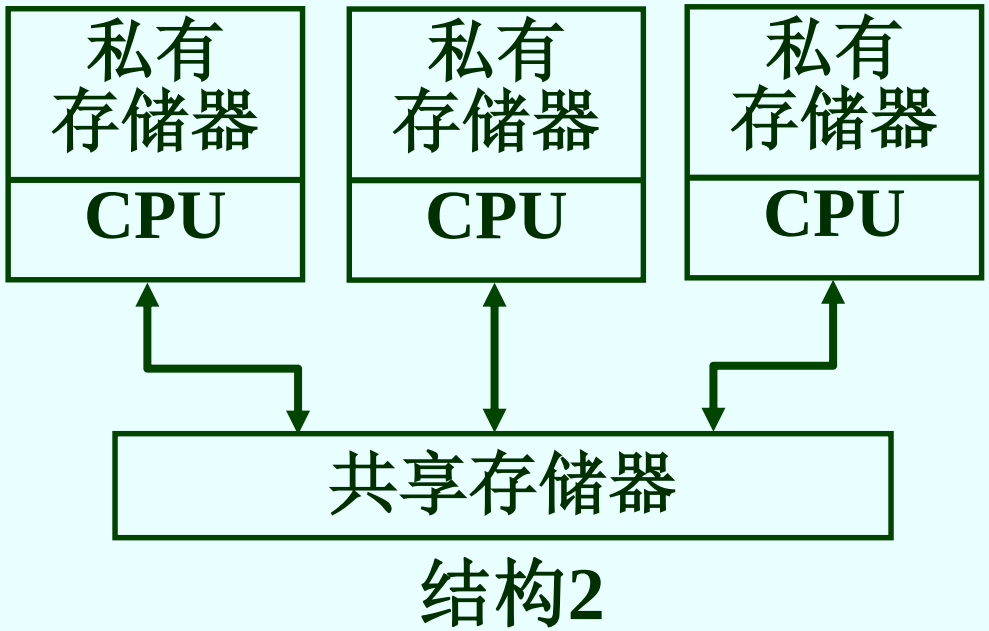
2、Cache内容过时

即Cache内容不能反映当前系统的状况。

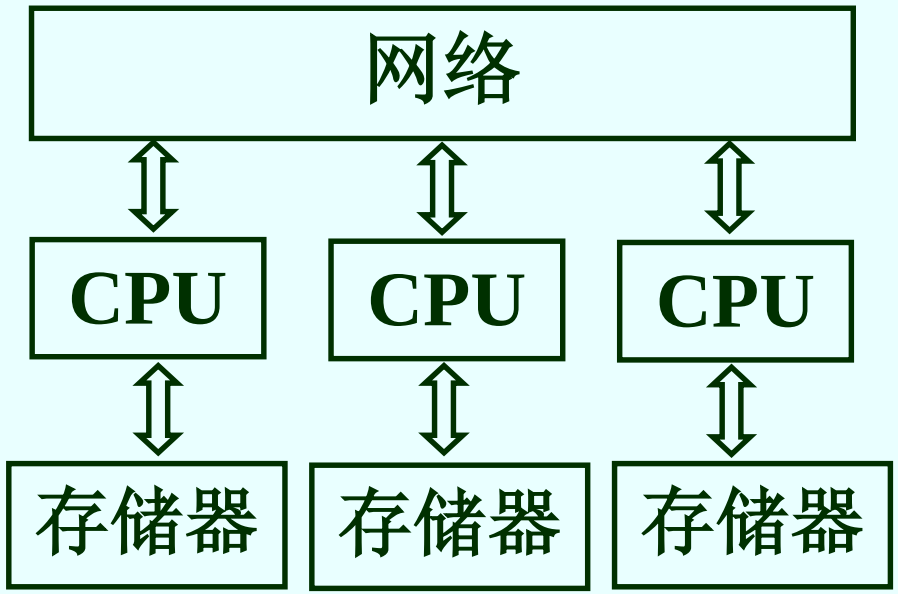
附: 多处理器系统的两种主要结构形态

1、共享存储器(Shared Memory)的多处理器





2、消息传递的多处理机系统



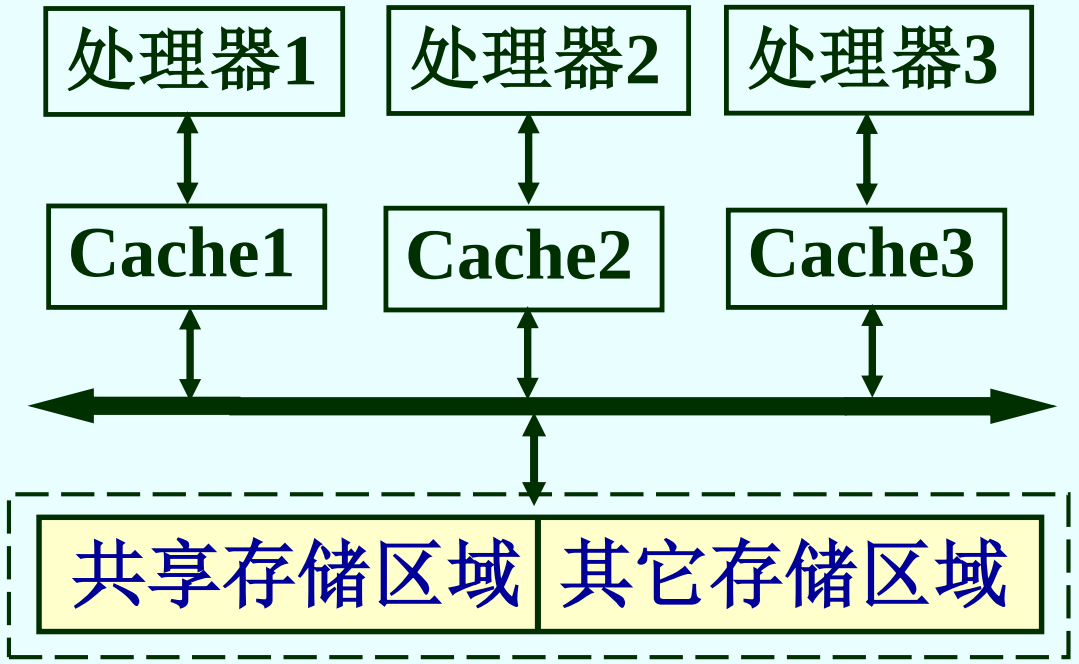
特点:

多台计算机通过网络互连, 构成一个并行计算平台, 计算机之间通过“消息” 传递协同工作。这种基于消息传递的多处理器系统本质上是分布式系统。

① 发生Cache内容过时的条件

- 多机(多处理器)系统
- 各处理器有自己的Cache
- 多处理器共享一部分内存区域

以三个处理器为例, 如下图所示:



多处理器共享存储区的要求:

各处理器的Cache单元内容应与共享区相同地址单元内容一致

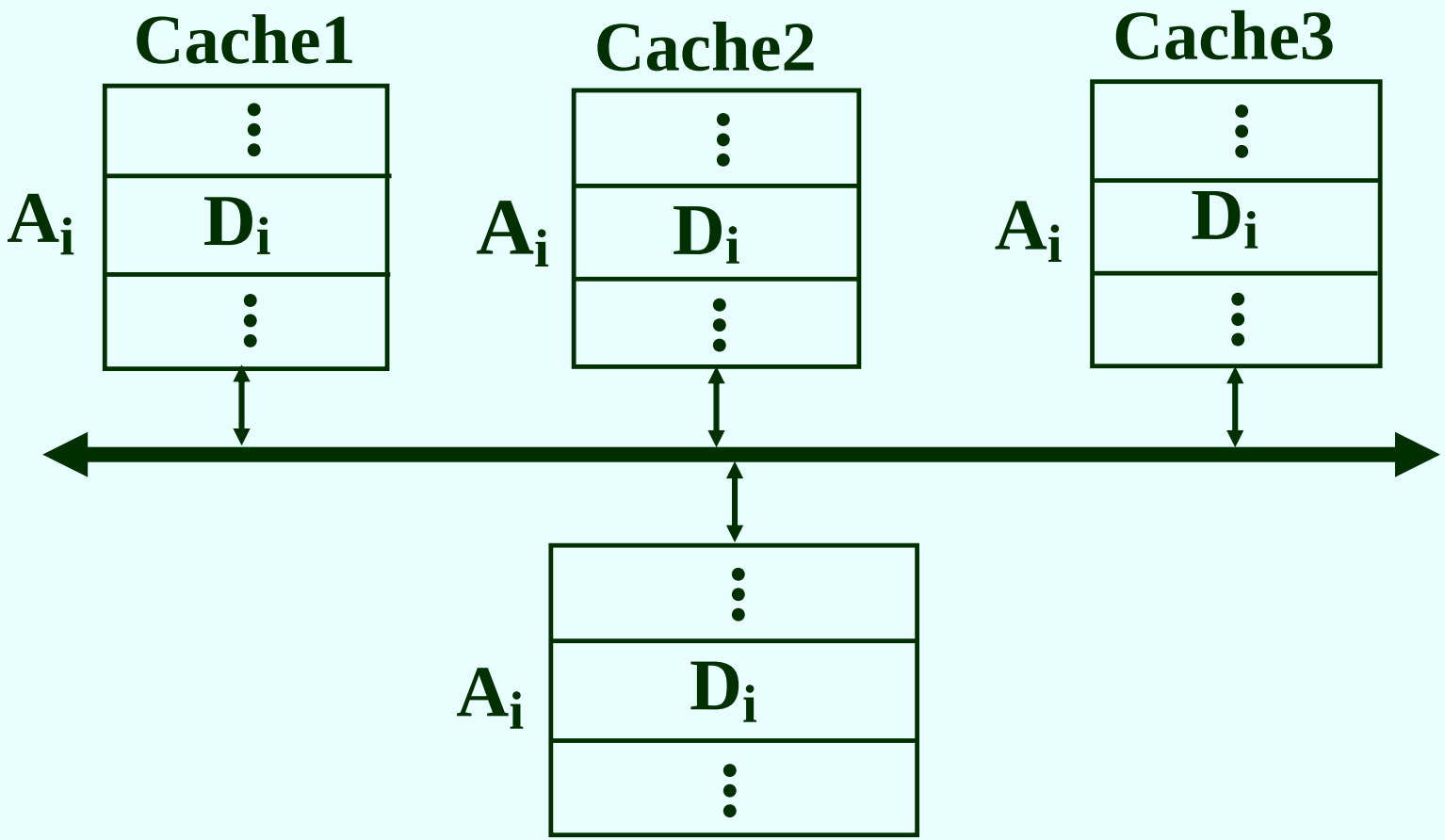
例如:

在内存共享区有一个单元A_i的内容为D_i, 如果在Cache内有该地址和数据, 则三个Cache的A_i单元内都应该为D_i。并且, 三个处理器的Cache必须能够反映当前共享存储区的“更新”情况。

② Cache内容过时原因和后果

如果哪一个Cache的某个单元内容, 不能反映当前共享区的“更新”情况。那么, 该Cache相应单元内容就过时了(该内容也称为“过时数据”)。

例如:

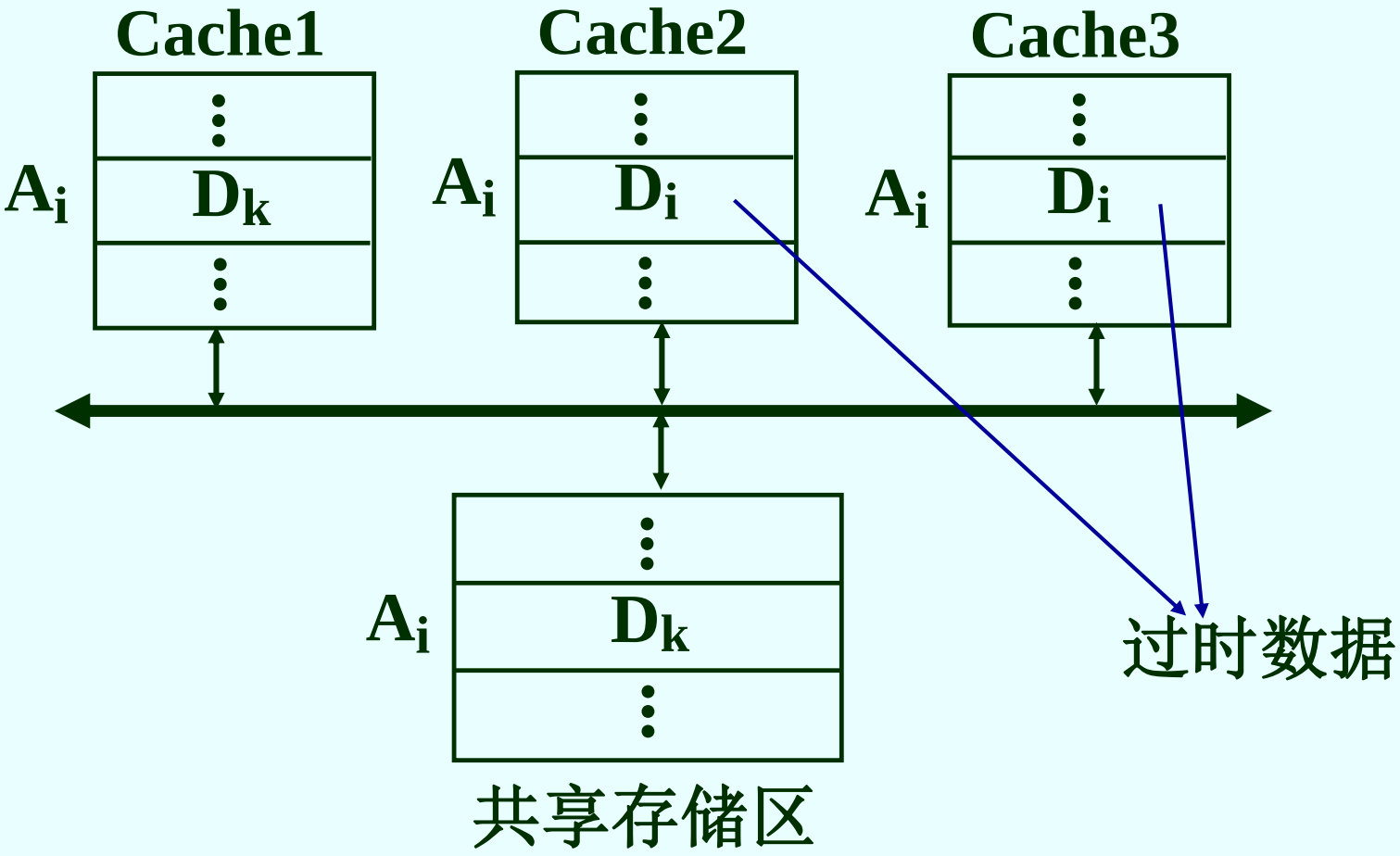


Cache单元与共享区对应单元内容一致

假设:

处理器1向共享区的A_i单元写入数据D_k(如下图):

则Cache2和Cache3的A_i单元内容D_i成为过时数据。



过时的后果:

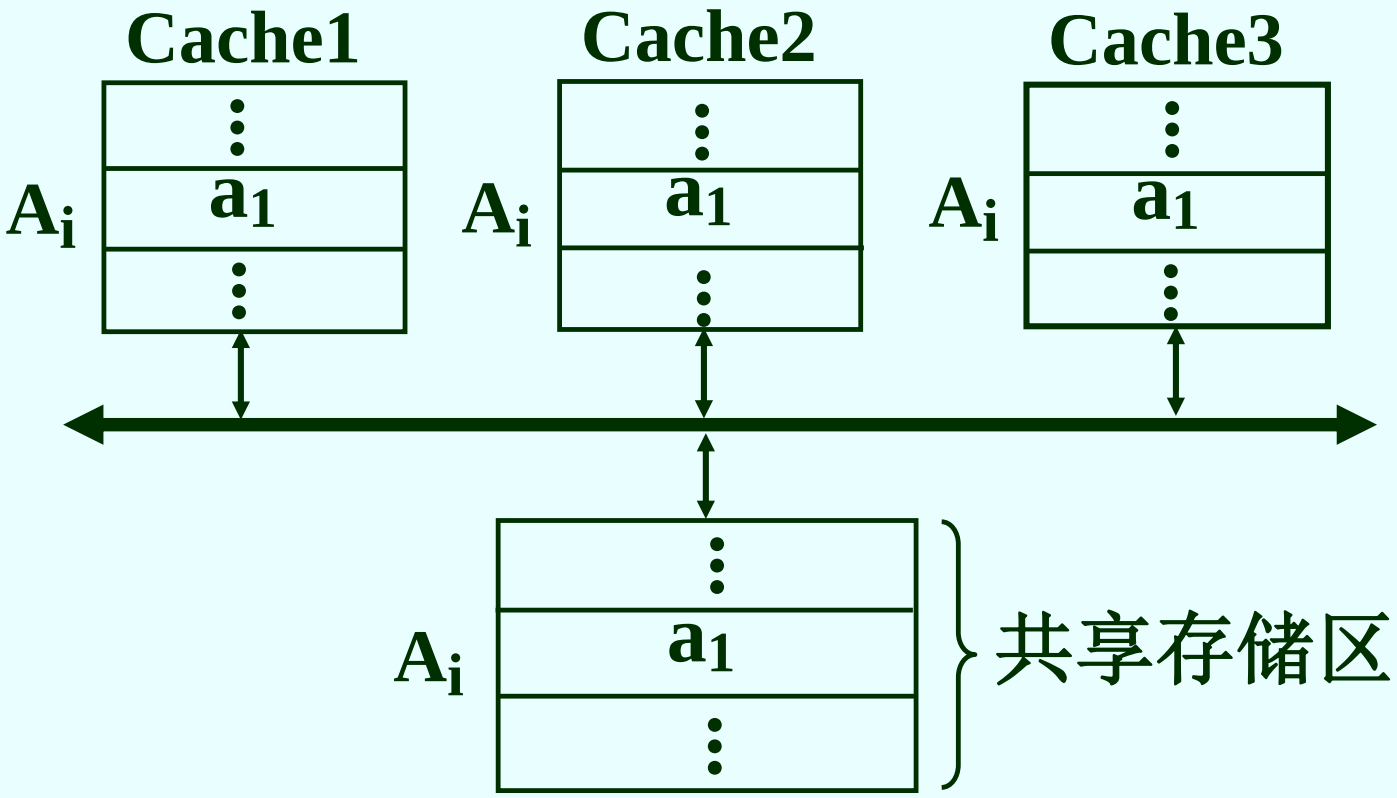
多处理器系统通过共享内存交换数据, 并共同(协同)完成某一项工作, 过时数据可能导致错误结果。

比如: 三个处理器共同完成一个复杂函数计算。

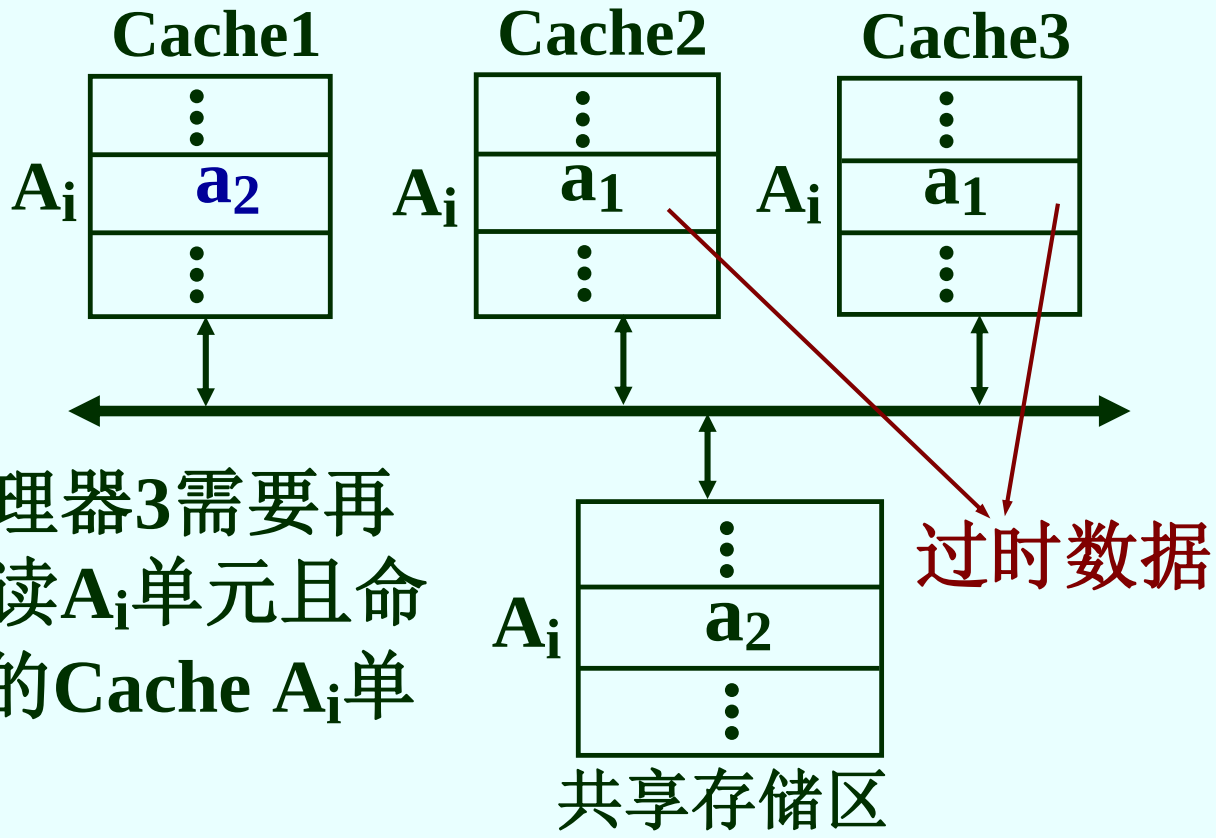
假设:

三个处理器在执行过程中, 共享一个参数C, 该参数随执行过程的推进而变化, 该参数由处理器1提供, 通过共享存储区交换该参数值(处理器2和处理器3通过共享存储区获取该参数)。

如果处理器1计算出$C=a_1$, 并写入共享区单元A_i。当处理器2和处理器3需要该参数时, 读A_i单元, 不命中Cache (初始状态下, Cache2和Cache3还没有地址A_i), 则访问共享区A_i单元, 并将该地址A_i以及内容$C=a_1$存入自己的Cache, 如下图所示:



由于C值是变化, 如果处理器1为C赋了一个新的值a_2, 并写入共享区:



如果处理器2或处理器3需要再次读取该参数, 则读A_i单元且命中Cache, 从自己的Cache A_i单元中读取a_1。

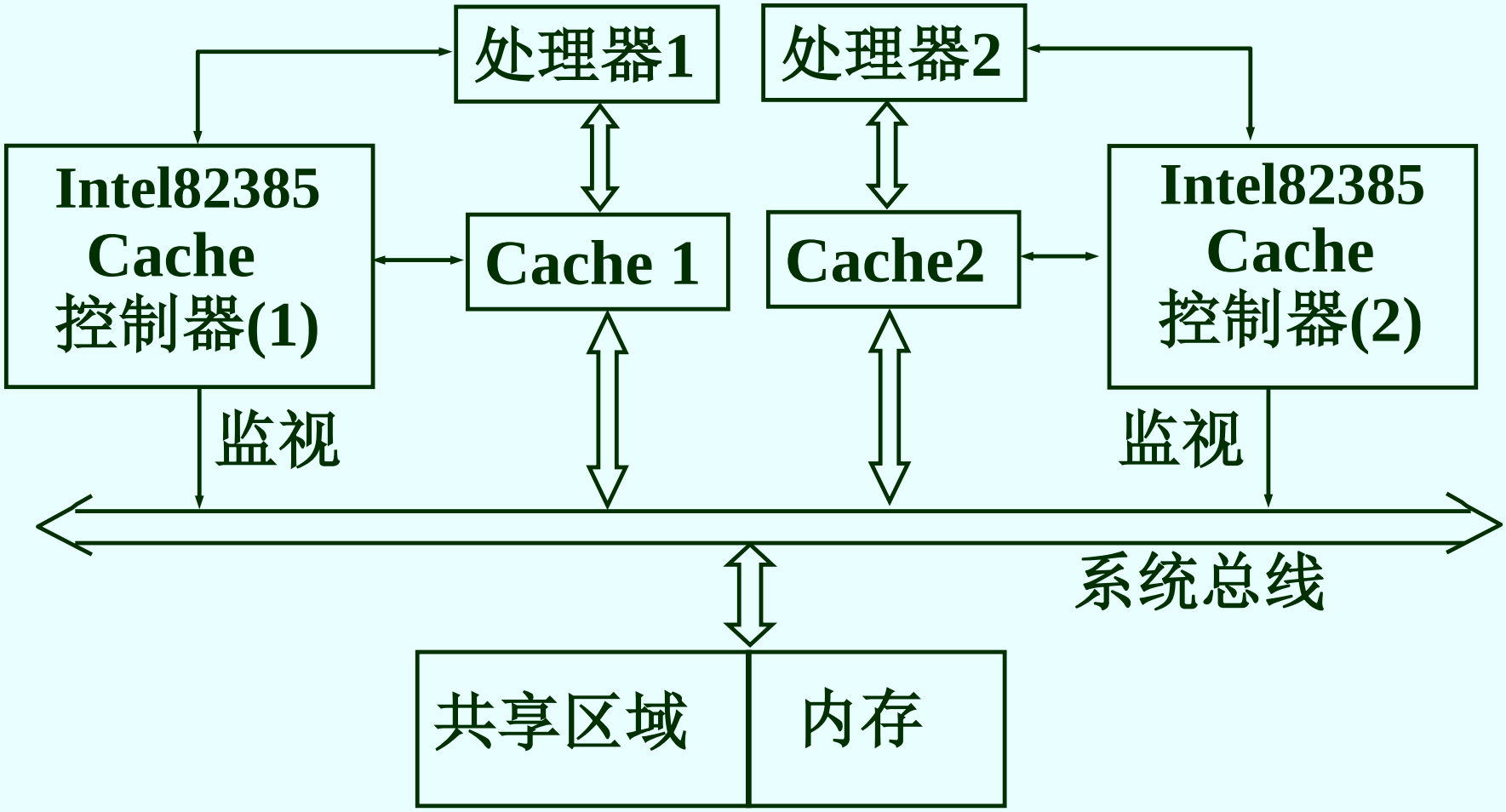
但此时C的最新值应该是a_2, 导致处理器2或处理器3由于读取过时数据(a_1)而产生计算错误。

③ 解决Cache内容过时的方法

- 方法一：总线监视

基于总线监视的Cache内容清除或标识无效

Cache控制器监视系统地址总线。如果有其它处理器向内存共享区中写数据(写入的地址在自身的Cache中存在), 意味着自身Cache的该单元内容将变为过时, 则将该单元内容清除或标识为无效, 使Cache中不再有过时的数据。


处理器1
处理器2
Intel82385
Cache
控制器(1)
Cache 1
Cache2
Intel82385
Cache
控制器(2)
监视
监视
系统总线
共享区域
内存

总线监视示意图

- 方法二: 不可高速用存储器

  一种能够避免过时数据的方法。

  即: 凡是共享区的主存单元内容都不允许进入高速缓存, 因此从根本上消除了“过时”。

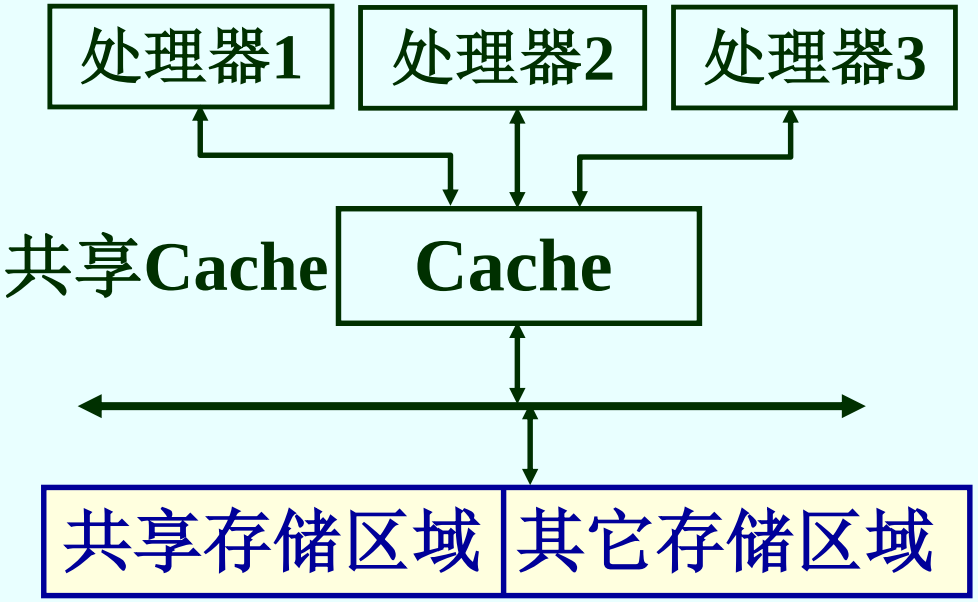
- 方法三: 硬件透明性

  硬件透明性包括:

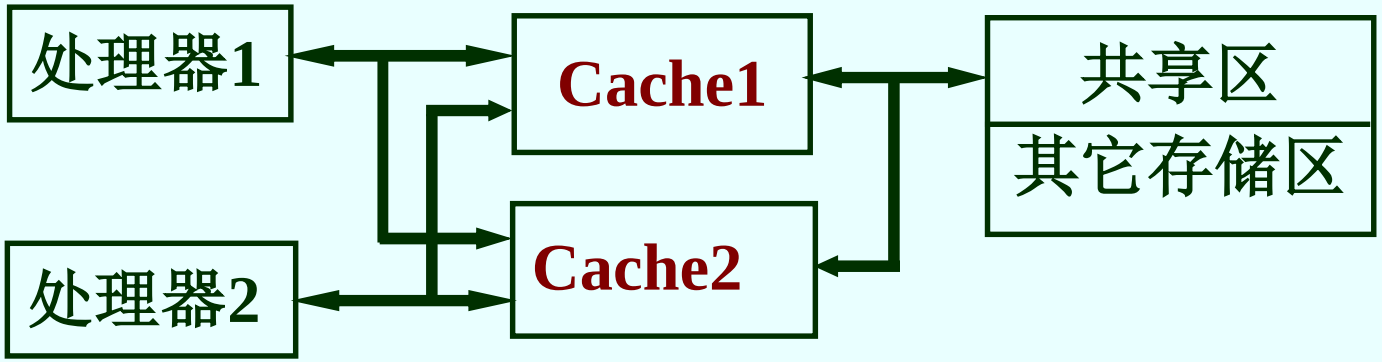
  – 多处理器使用同一个Cache(共享Cache)

  – 广播方式(交叉连接)

– 多处理器使用同一个Cache(共享Cache)



– 广播方式(交叉连接)



- 方法四： Cache清除

  将Cache中所有已经更新过的数据写入主存储器, 并清除Cache的所有内容。

  假如任意一部件写入共享主存之前, 对系统中的所有Cache都进行清除作, 则任何Cache都不会有过时数据存在。

三、Cache主要性能指标

- T_C Cache的访问周期
- T_m 主存的访问周期
- λ Cache的命中率
- S_P Cache的加速比(访问效率)
- T Cache系统等效访问周期(平均访问时间)

$$T = \lambda T_C + (1-\lambda) T_m$$

λ (命中率):

设Nc表示Cache完成存取的总次数, Nm表示主存完成存取的总次数, λ为命中率。则:

$$\lambda = \frac{Nc}{Nc + Nm}$$

越大越好

$$S_P = \frac{T_m}{T} = \frac{T_m}{\lambda T_C + (1-\lambda) T_m} = \frac{1}{(1-\lambda) + \lambda \frac{T_C}{T_m}}$$

器件选定后，$\frac{T_C}{T_m}$为定值，S_P主要取决于λ

– 影响λ的因素:

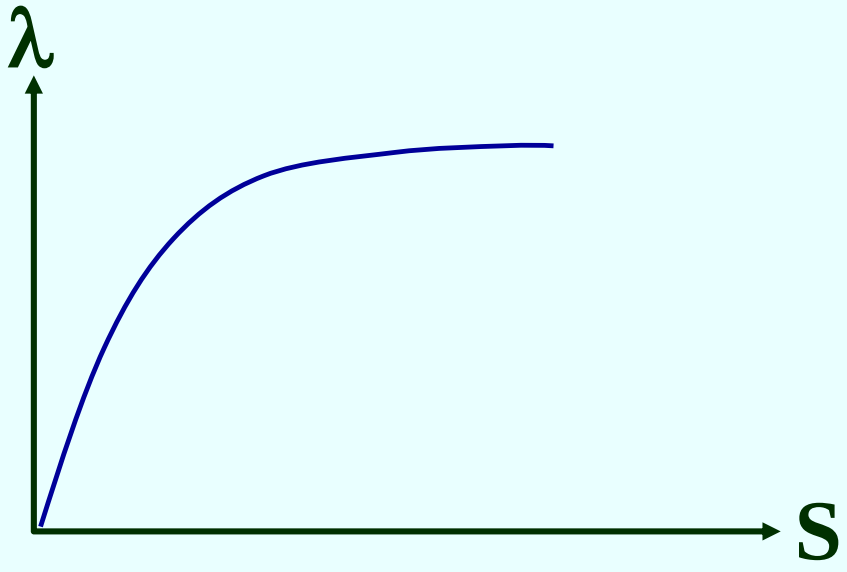
Cache容量S、淘汰算法、Cache结构、数据块的大小等、Cache一致性的解决方法、程序运行情况(比如一个程序反复对某一小块程序或数据进行操作，Cache就会有很高的命中率)等。

例: 不同Cache规模、结构方式以及Cache行大小情况下, 一级Cache的命中率统计:

Cache配置			命中率(%)
规模大小	结构方式	Cache行大小	
64KB	直接对应式	4B	88%
64KB	双组关联式	4B	89%
64KB	四组关联式	4B	89%
64KB	直接对应式	8B	92%
64KB	双组关联式	8B	93%
128KB	直接对应式	4B	89%
128KB	双组关联式	4B	89%
128KB	直接对应式	8B	93%

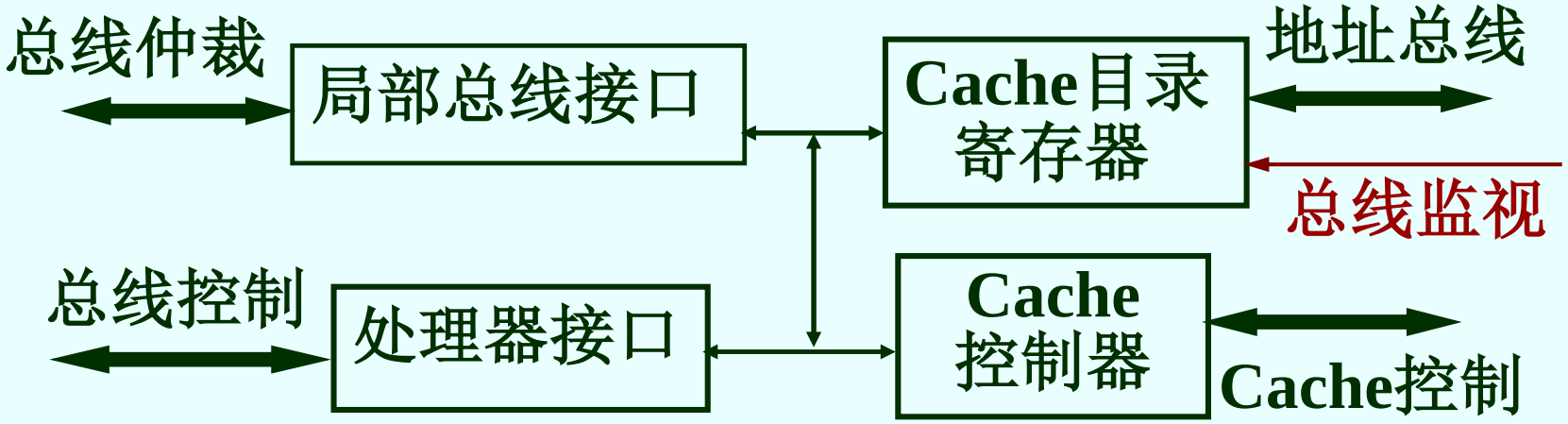
– λ与容量S的关系:

命中率λ与容量S成正比, 但不是线性关系

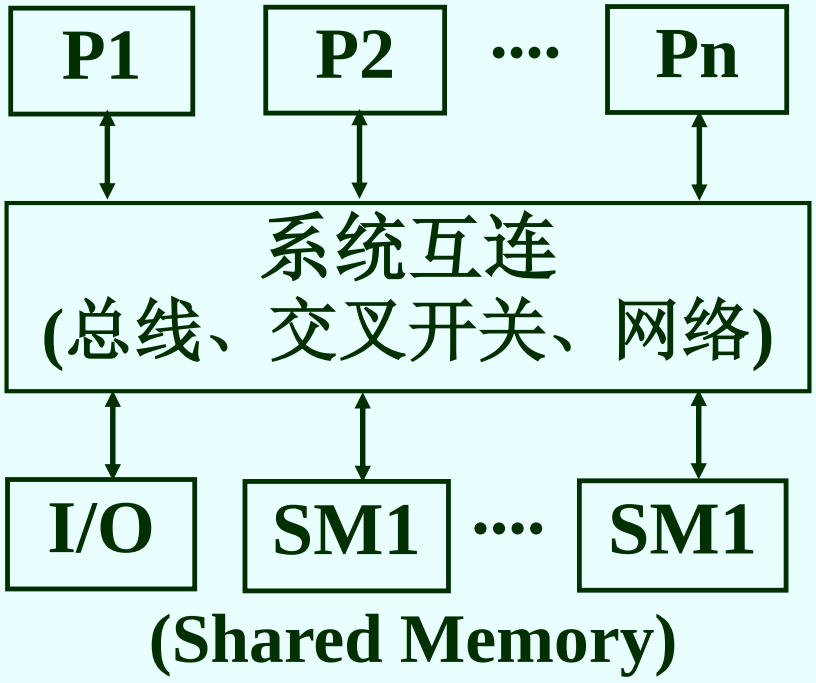


(四) Intel 82385Cache控制器

- 可控制32K的片外Cache
- 直接对应和双组关联可选(通过引脚)
- 采用通写方式
- 使用LRU淘汰算法
- 具有总线监视并标识Cache单元“无效”的功能
- 具有“不可高速用存储器”功能



附：SMP系统中的存储系统



增加系统中处理器数量，系统消耗在内存抢占的时间也随之增加，即多个处理器不能同时读写数据。比如一个CPU正在读一段数据时，其它CPU可以读这段数据，但是当一个CPU在修改(写)某段数据时，它会将这段数据锁定，其它CPU要读写这些数据就必须等待。

CPU越多，处于等待状态的CPU也越多。因此随着CPU数量的增加，整机系统性能无法呈线形增长。

解决上述问题的方法之一是增大CPU的Cache容量, 由于Cache不是共享的, 从而大幅度减少多个CPU争抢内存资源的现象。

目前的SMP系统大多采用侦听算法来保持Cache的一致性(类似总线监视)。Cache越大, 抢占内存概率就越小, 而且由于Cache的数据传输速度高, Cache的增大可以提高CPU的运算效率。

但是, Cache越大, 保持Cache中的数据与内存的一致性的时间消耗也越多(CPU中断当前任务去更新内存数据的频率越高), 导致系统性能下降。

这一矛盾一直制约着SMP系统性能的线性增长。

问题:

1. 从数据存储的角度, Cache也是一种主机内的存储器, 但在评价计算机系统的内存容量时, 并不考虑Cache容量, 原因是什么?
2. 在指令类型相同的情况下, 一个1万行的程序运行一次, 与另一个1百行的程序循环执行100次相比, 哪一个运行速度块, 为什么?

误区:

从一个程序的Cache性能推断其他程序的Cache性能