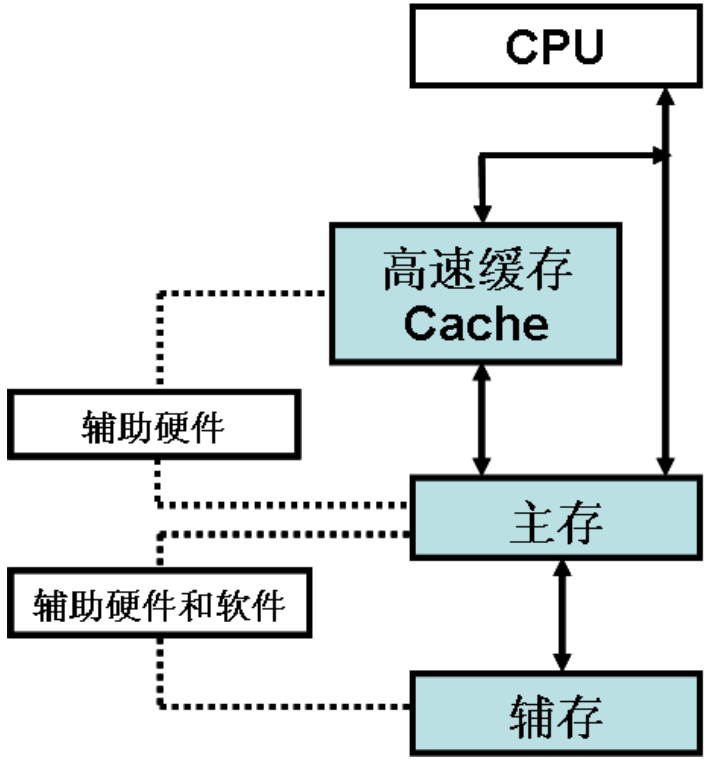
计算机组成原理

第5章 存储系统

llxx@ustc.edu.cn

本章内容

- ✓唐本
  - ✓4.1 概述
  - ✓4.2 主存储器
  - – 4.3 高速缓冲存储器
  - – 4.4 辅助存储器
  - – 4.6 相联存储器
- • COD4
  - – 5.2，5.3，5.7



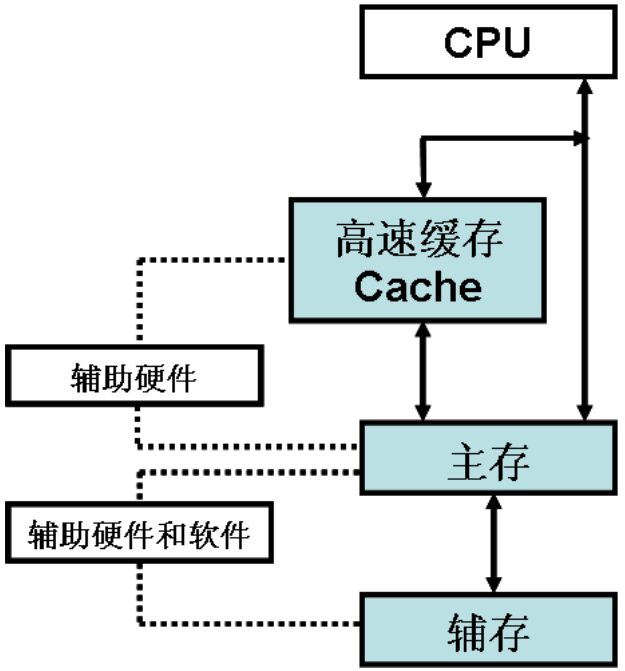


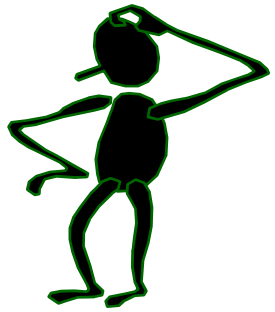

4.3 高速缓冲存储器（Cache）

1. Cache-存储器映象
2. 替换算法

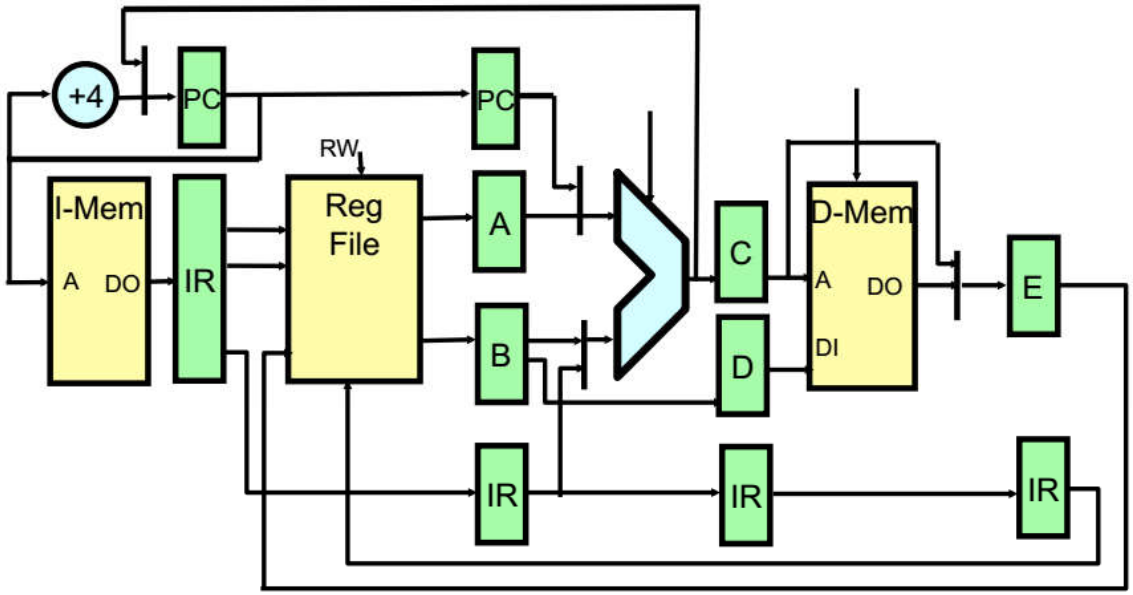
问题

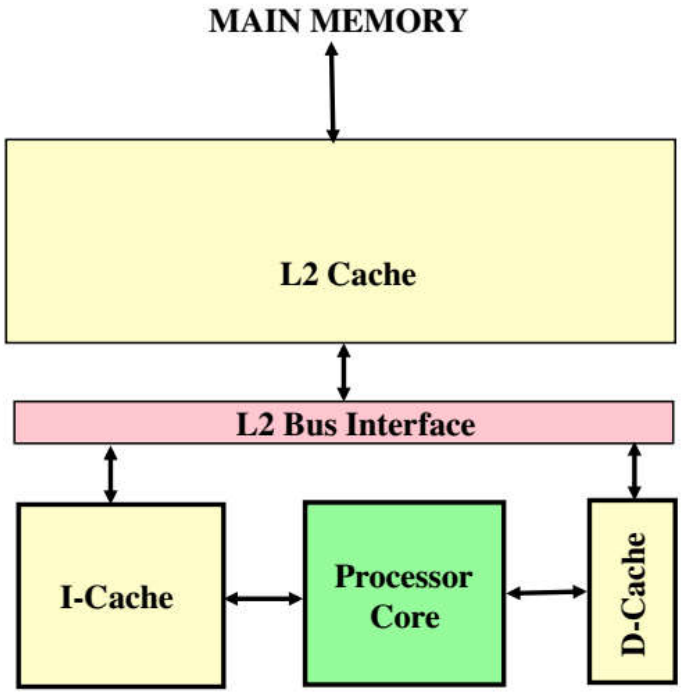
- 为什么需要Cache？
  - 性能、结构
- Cache有效性的理论基础
  - 局部性
- 影响Cache命中率的因素
- Cache的读写操作过程
- Cache的基本结构和映射机制
- Cache的替换策略





Where Does the Memory Hierarchy Fit In?

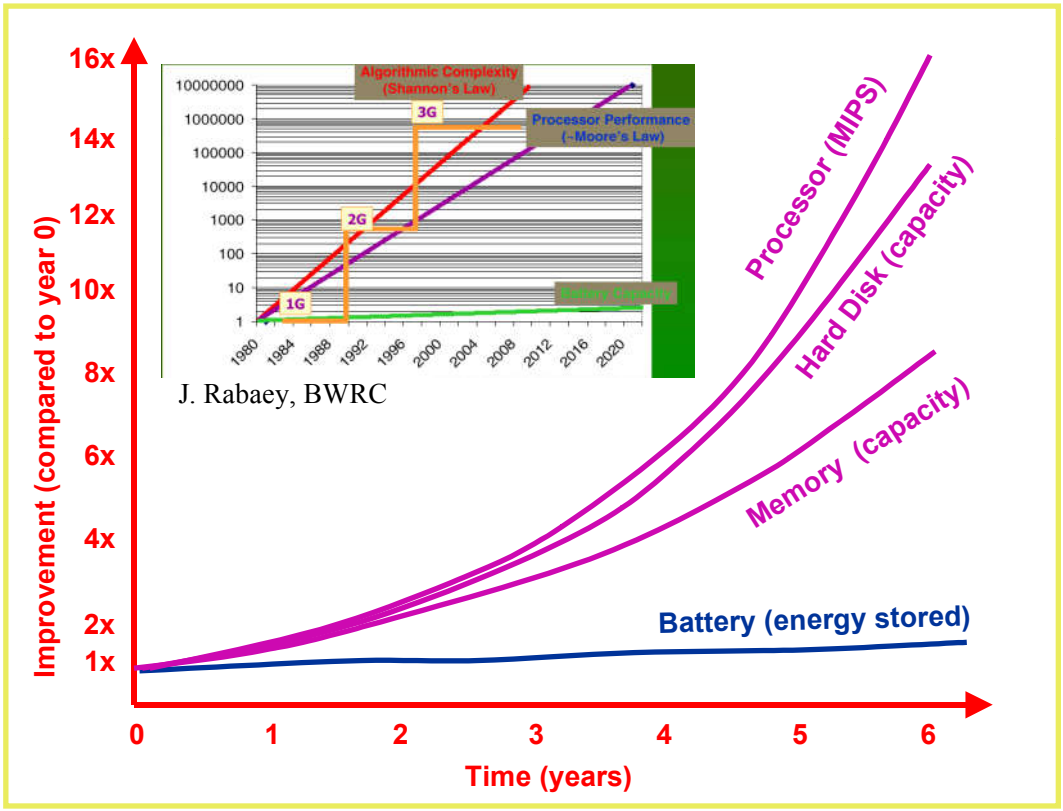




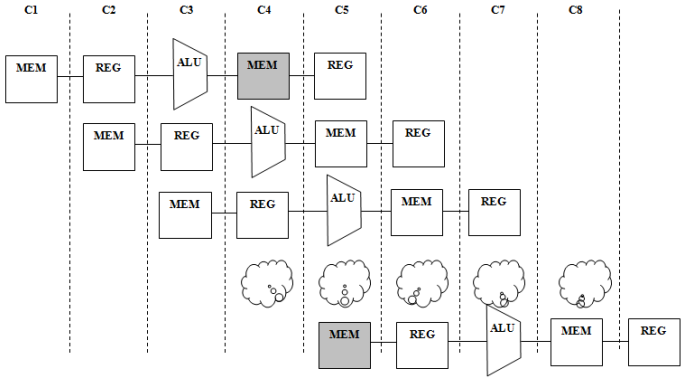
- Alpha 21264
  - 64KB I-Cache, 64KB D-Cache, >1MB L2 Cache

问题的提出（1）

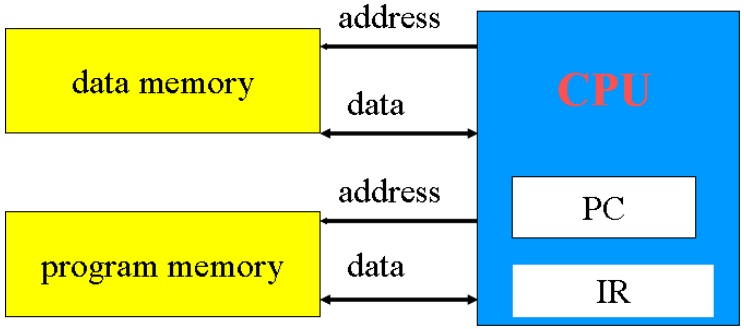
- 主存速度始终跟不上CPU(25MHz的80386之后)的发展
  - 100MHz的Pentium处理器平均10ns就执行一条指令，而DRAM的典型访问时间是60 ~ 120ns。
- 流水线：单周期访存

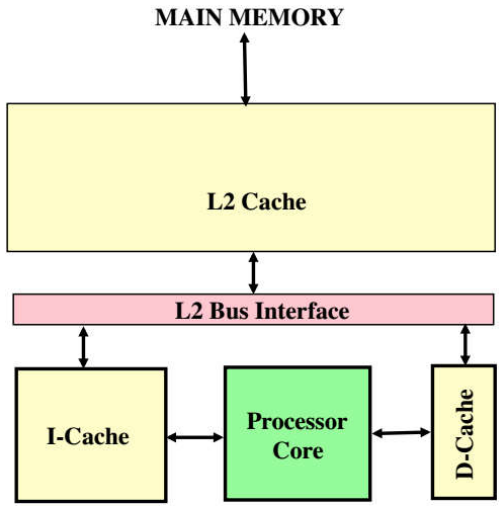


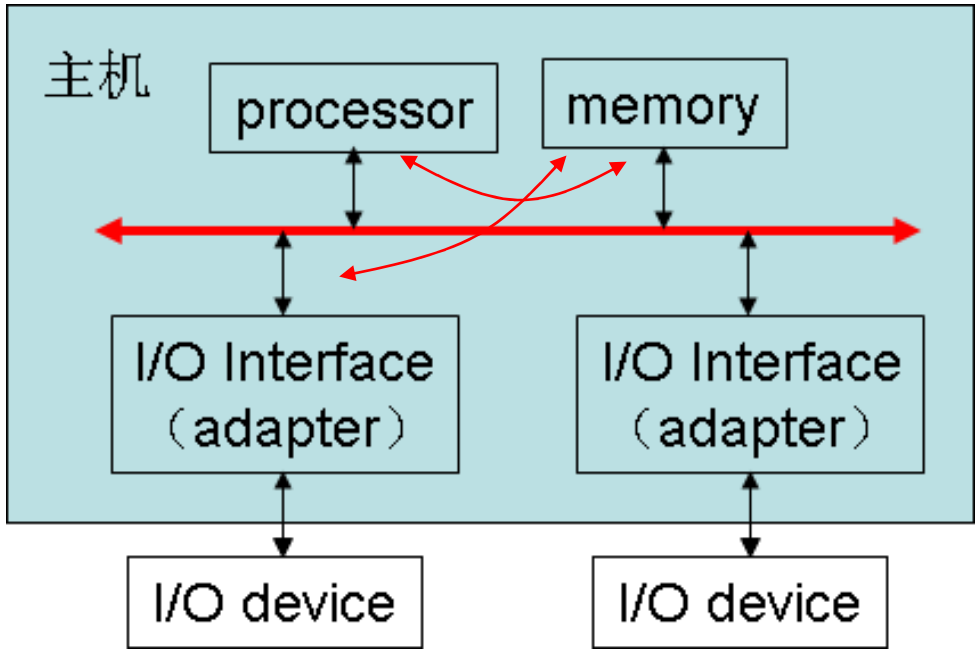
问题的提出（2）



- 结构冲突
  - 访存冲突：指令预取与数据读写
  - 总线占用：CPU和I/O争抢访问主存 →减少访存







程序访问的局部性原理

- 时间局部性：最近的访问项（指令/数据）很可能在不久的将来再次被访问（往往会引起对最近使用区域的集中访问）
- 空间局部性：一个进程访问的各项其地址彼此很近（往往会访问在存储器空间的同一区域）

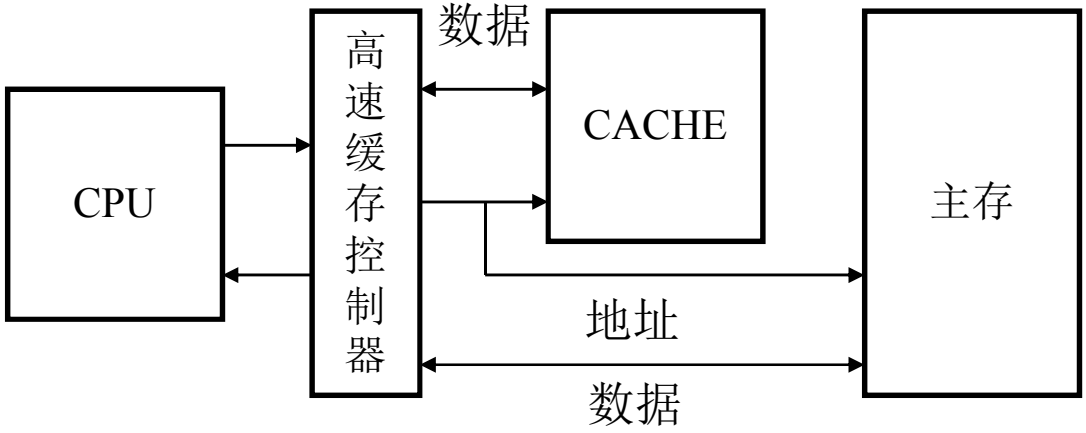
- for i := 0 to 10 do

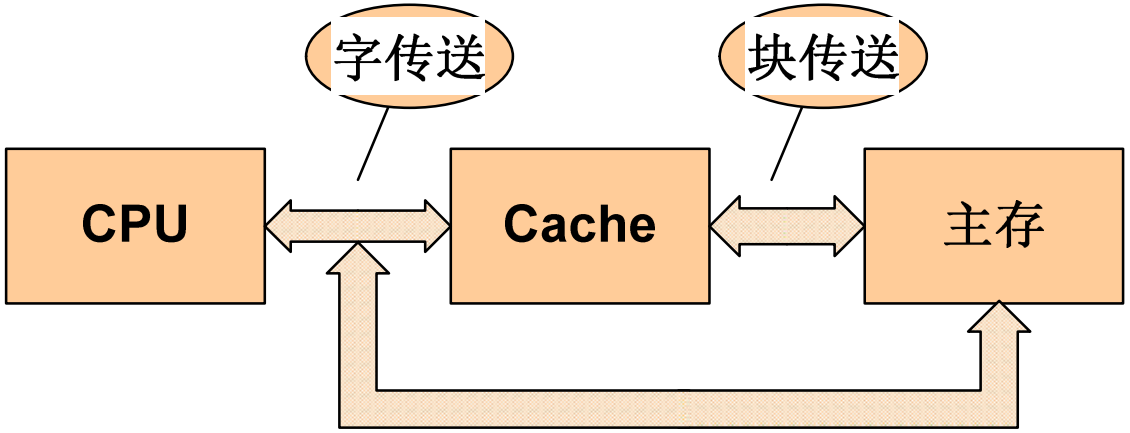
  A[i] := 0;

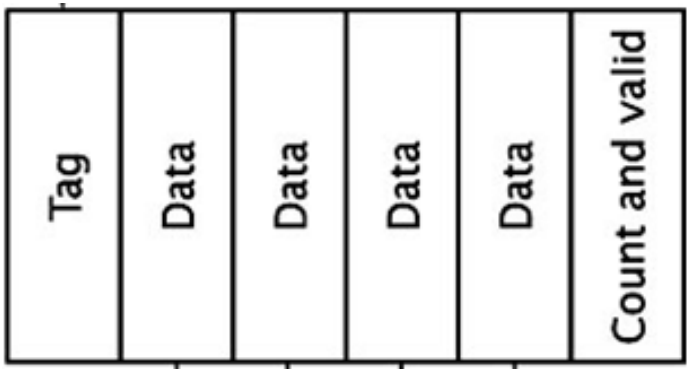
命中、不命中、命中率

- Cache命中（hit）
  - CPU欲访问的数据已在缓存中，即可直接访问Cache
- Cache不命中（miss, 失配，缺失）
  - CPU欲访问的数据不在Cache内，此时需将该数所在的主存整个子块一次调入Cache中。
  - CPU被阻塞（blocking），等待数据调入。
- 命中率（Hit rate）
  - CPU要访问的信息已在Cache内的比率。
    - 通常用命中率来衡量Cache的效率。
- 不命中率（Miss rate）

Cache结构

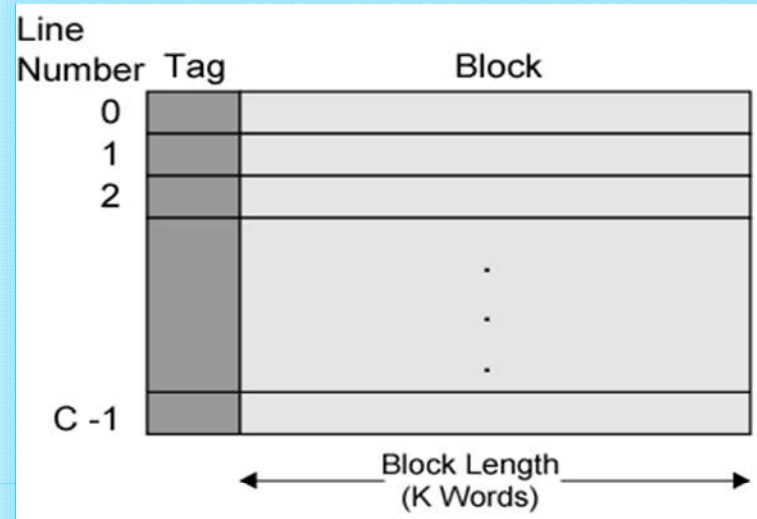




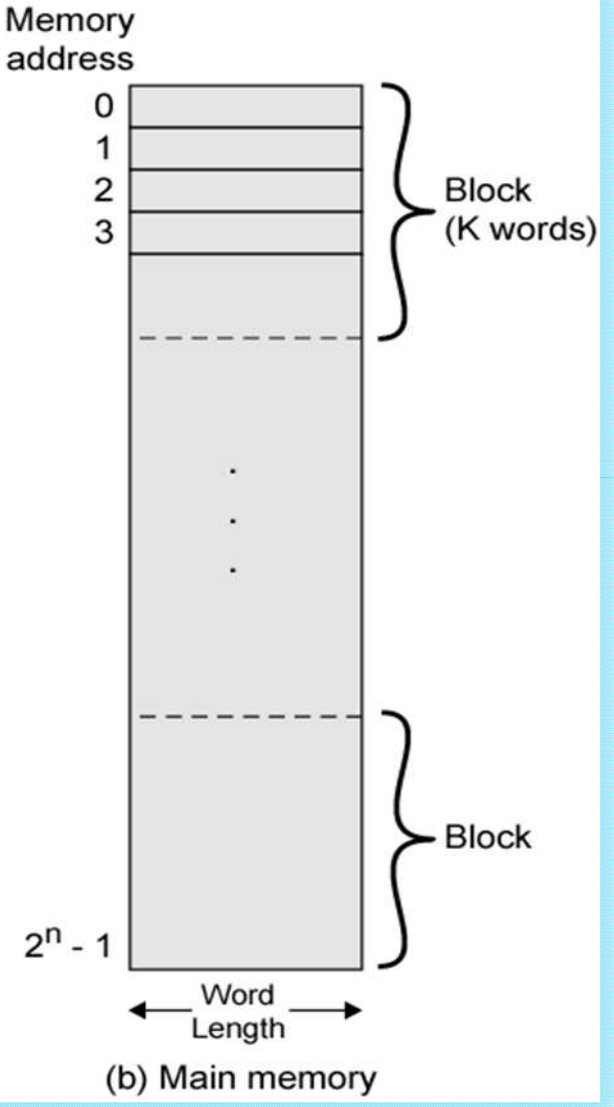


Cache Line =
tag + block(K words) + control bits

Cache Line/Main Memory Structure



(a) Cache



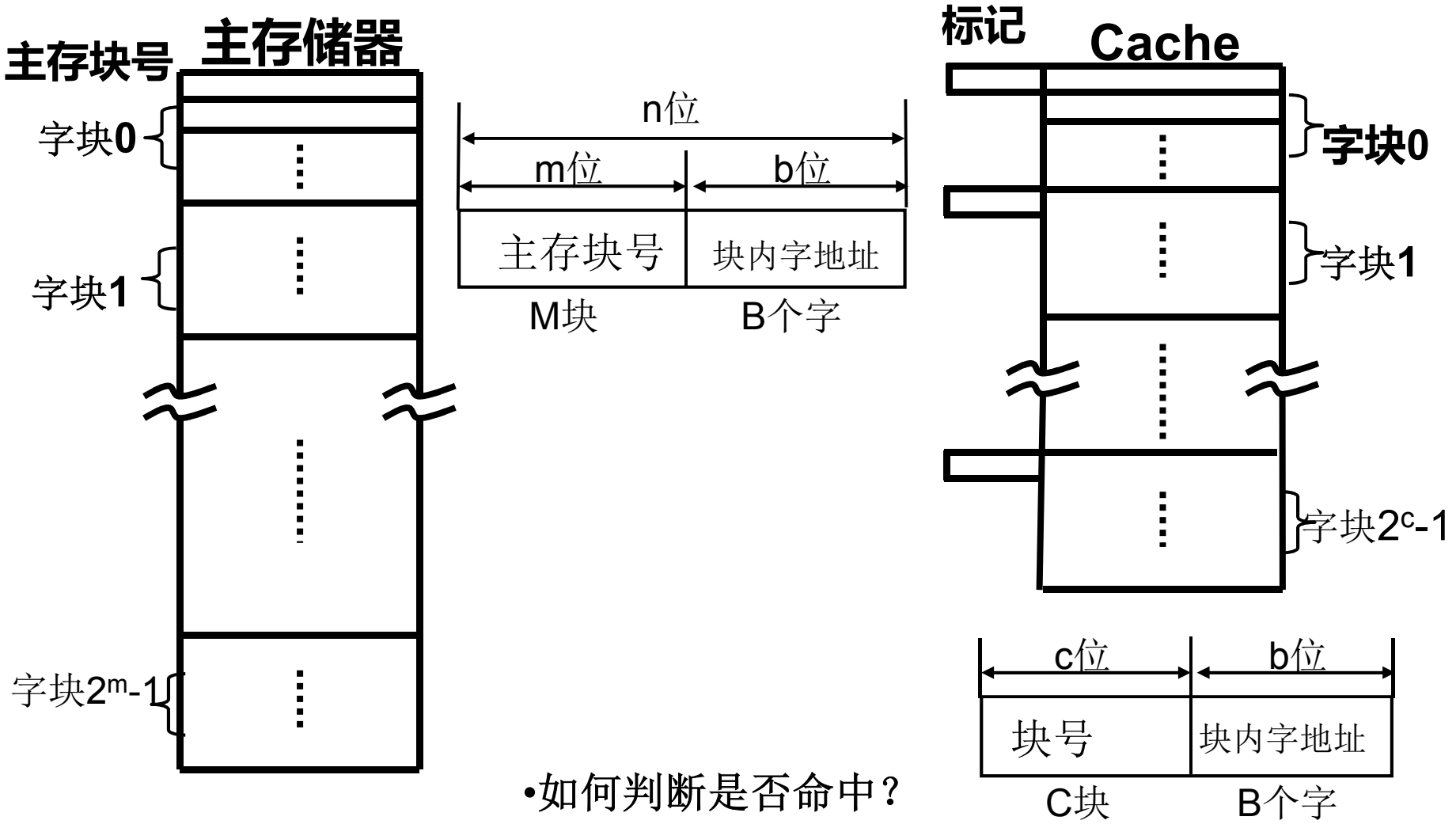
(b) Main memory

Main memory - 2^n words
- $2^n/K$ blocks

Cache has C lines of K words each
$C<M$

Tag – to identifies block

主存与Cache的单元编址模式



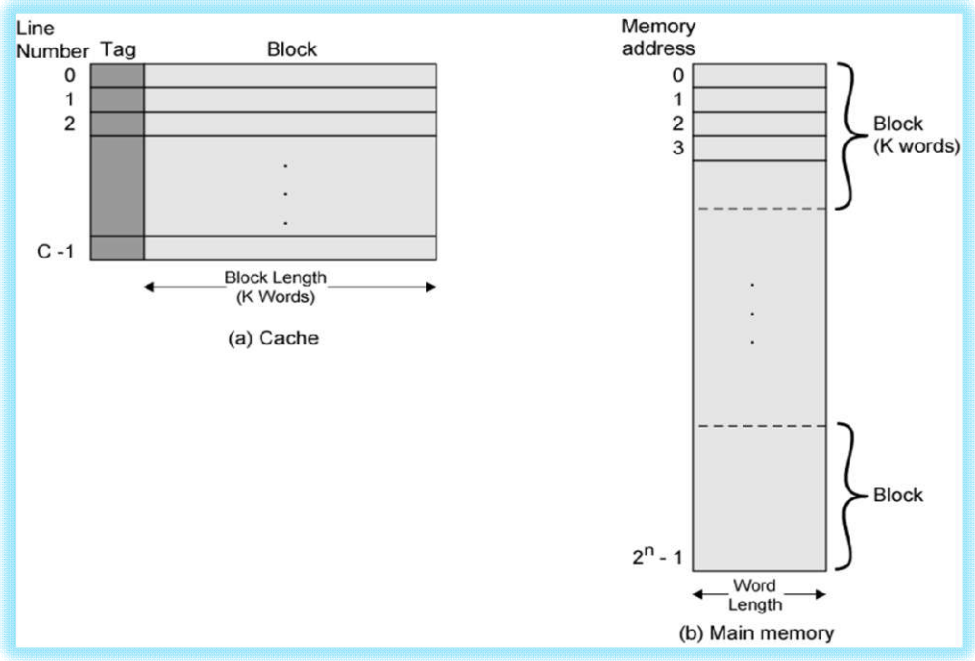
•如何判断是否命中？

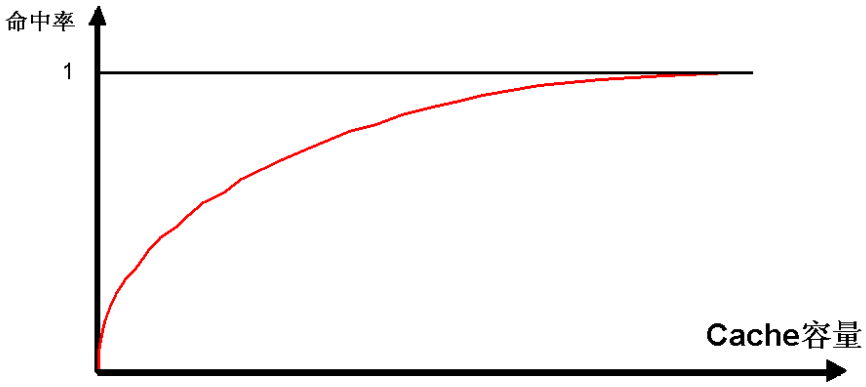
Cache基本结构参数

块（行）大小	4——128字节
命中时间	1——2时钟周期（常规为1）
失配时间	8——100时钟周期
（访问时间）	（6——60时钟周期）
（传送时间）	（2——40时钟周期）
失配率	0.5%——10%
Cache容量	1KB——1MB

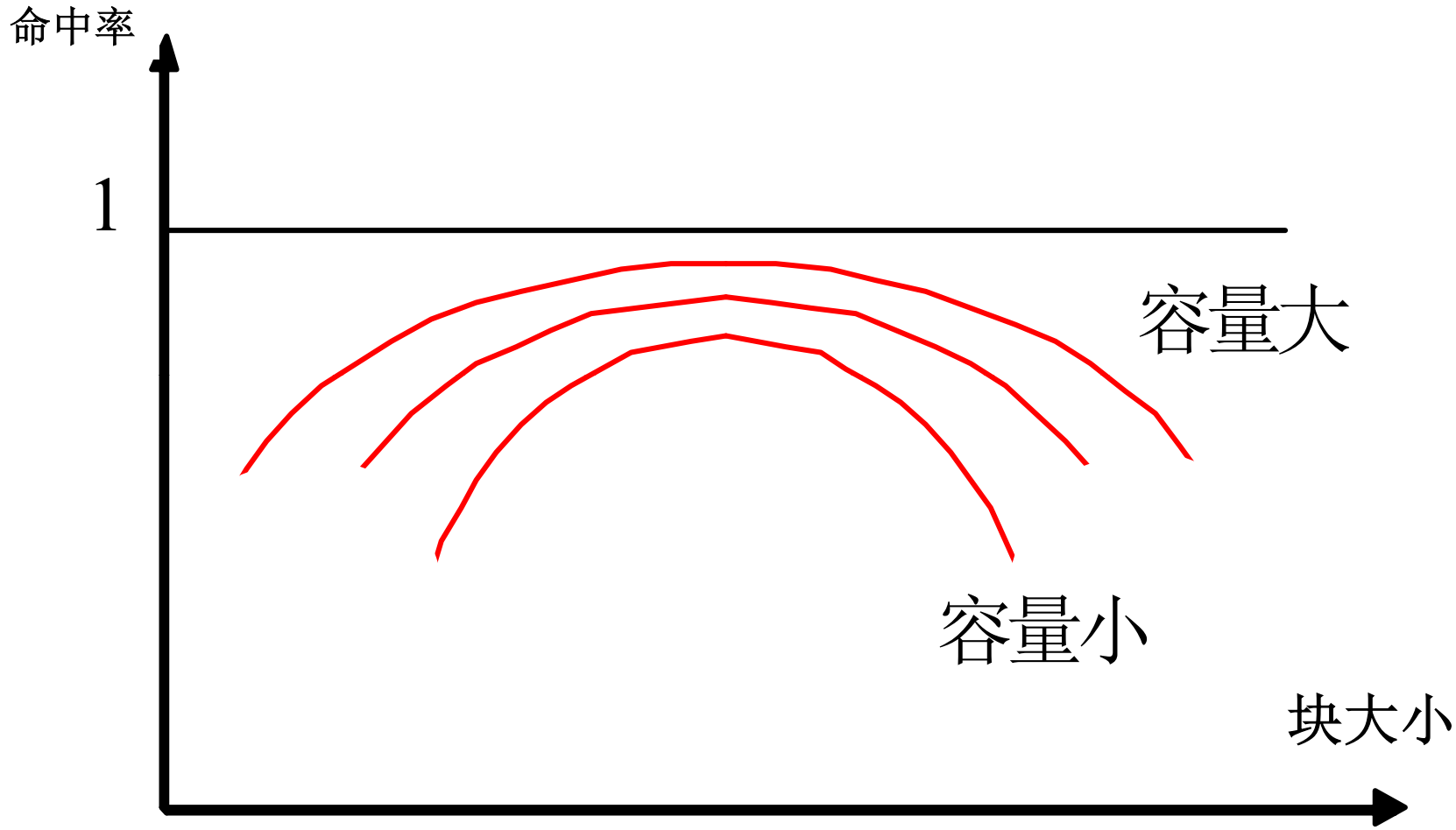
Cache效率

- 容量和块长是影响Cache效率的重要因素。
- Cache容量越大，命中率越高。
  - 当Cahce容量达到一定值时，命中率不会因容量的增大而明显提高。
  - Cache容量大，成本增加。





块（行）容量与命中率



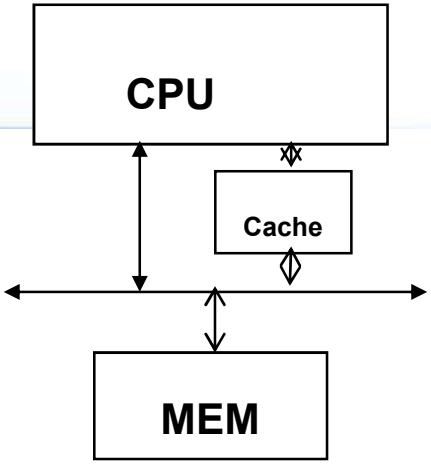
例

- 设Cache的速度是主存的5倍，命中率为95%，则采用Cache后性能提升多少？

  系统平均访问时间=*0.95*t+0.05*5t=1.5t*

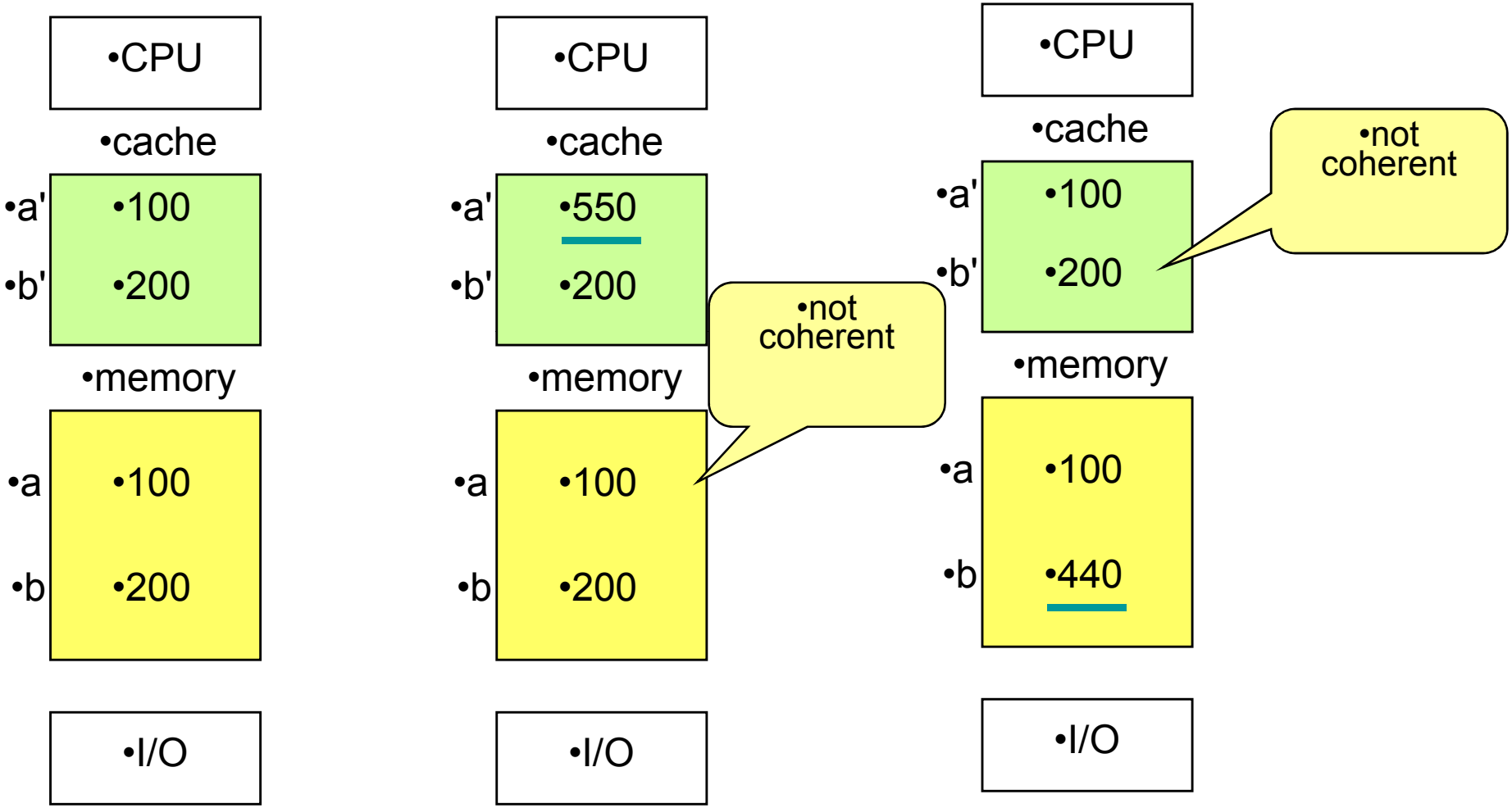
  性能提升=*5t/1.5t=3.33倍*

Cache的读操作

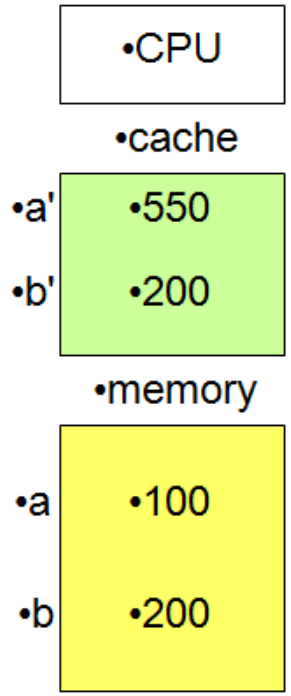

CPU
Cache
MEM

开始
CPU发出访存地址
命中?
Y
N
Cache满?
Y
N
访问Cache取出信息送CPU
访问主存取出信息送CPU
将新的主存块调入Cache中
执行替换算法腾出空位
结束

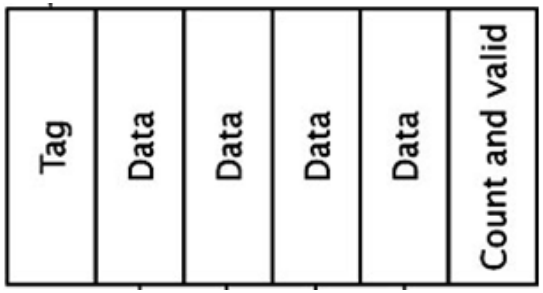
写操作： in single CPU system



Cache的写操作

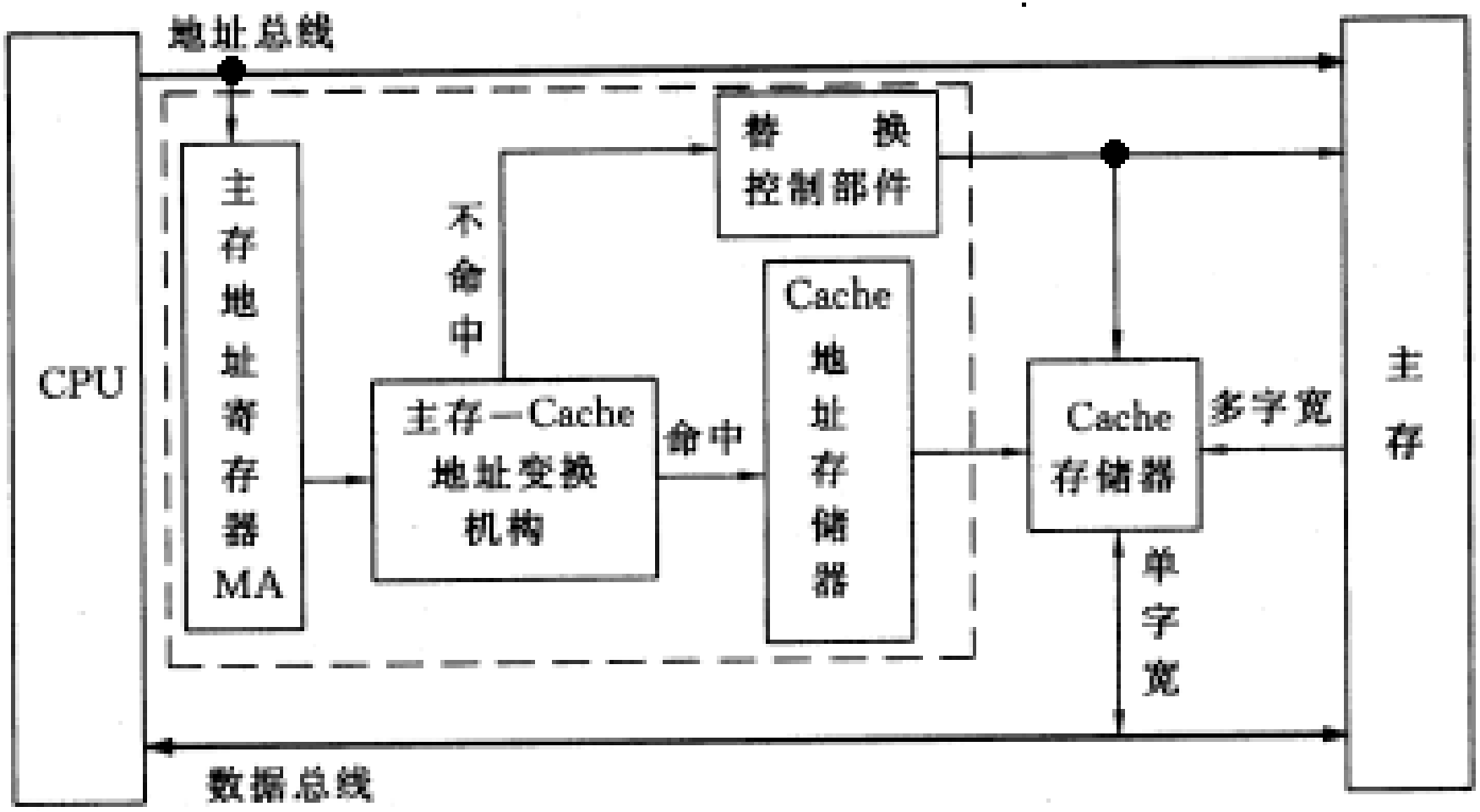


- 命中
  - 写回法（Write-back）：
    - 执行写操作时，信息只写入Cache；
    - 当Cache块被替换时，将该块内容写回主存，然后再调入新页。
  - 写直达法（Write-through、Store-though）
    - 每次写入Cache的同时，也写入主存。
  - 使无效法（Invalidated）
    - 信息只写入主存，同时将相应的Cache块有效位置“0”
    - 影响读操作
- 不命中
  - 写分配（write allocate）：读入后再写。
    - MIPS采用。
  - 写不分配(no write allocate，write around)：只写主存



- 性能比较
  - 写回法的开销是在块替换时的回写时间。
  - 写直达法在每次写入时，都要附加一个比写Cache长得多的写主存时间。
- cache与mem的一致性：
  - 写直达法一致性保持的要好一些。

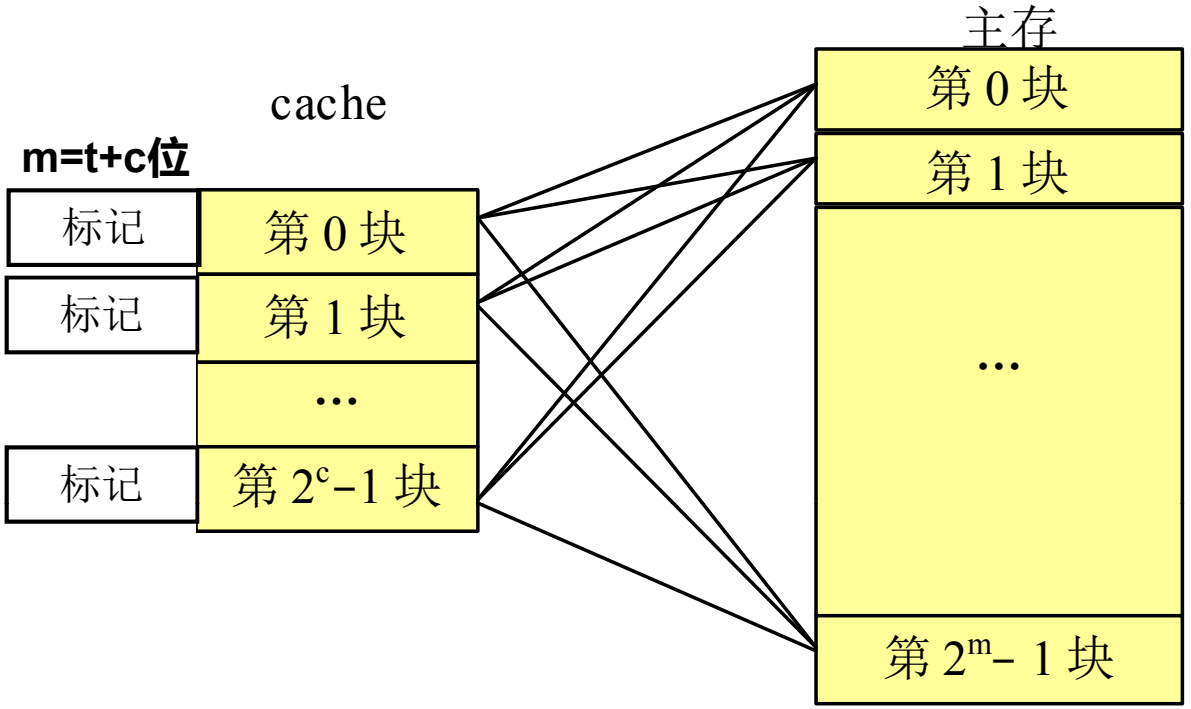
Cache的基本结构


地址总线
CPU
主存地址寄存器MA
不命中
替换控制部件
主存—Cache地址变换机构
命中
Cache地址存储器
Cache存储器
多字宽
主存
单字宽
数据总线

Cache存储体、地址映象变换机构、替换机构

- **Cache存储体**
  - 以块为单位和主存交换信息
  - Cache访存的优先级最高
- **地址映射变换机构**
  - 实现主存块号和Cache块号之间的转换。
    - 三种方式：全相联映射、直接映射、组相联映射。
- **替换机构**
  - Cache内容已满时，无法接受来自主存块的信息，需由Cache内的替换机构按一定的**替换算法**来确定从Cache内移出某个块写回主存。

1. 全相联映像



主存地址	主存字块标记	块内字地址
	m=t+c位	b位

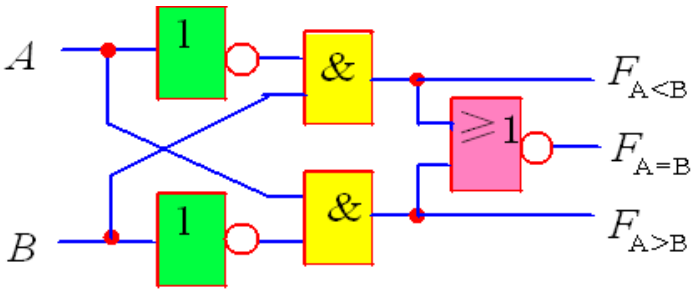
- Cache“标记位”多，比较位数长(m位)
- 比较次数多（m次：全部tag）

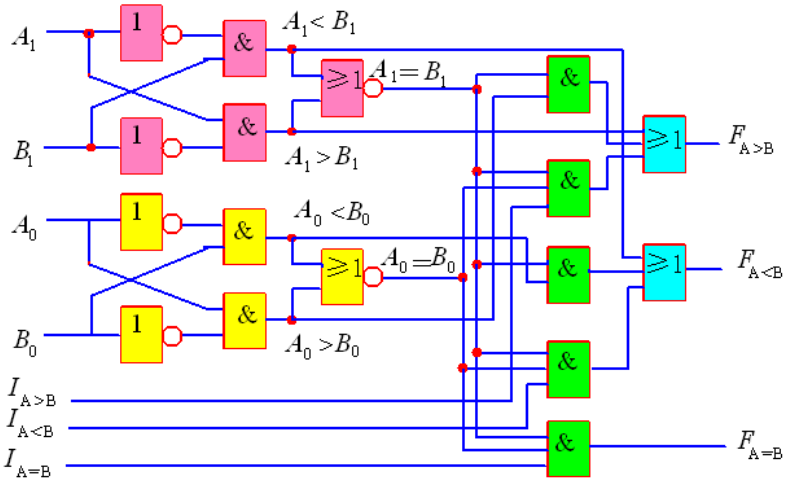
1. 全相联映像（续）

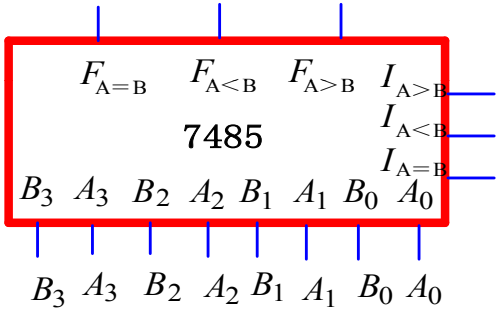

访问顺序 1 2 3 4 5 6 7 8
地址 22 26 22 26 16 4 16 18
字块号
块分配情况
0 1 2 3 4 5 6 7
操作状态 调进 调进 命中 命中 调进 调进 命中 调进

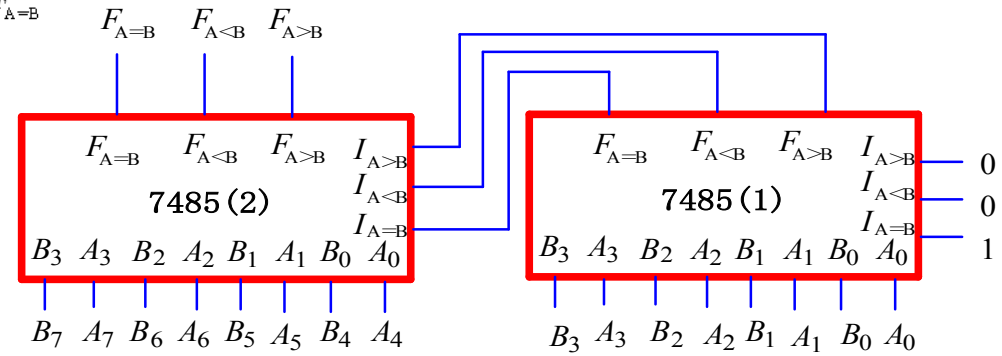
数值比较器

- 1位
- 2位

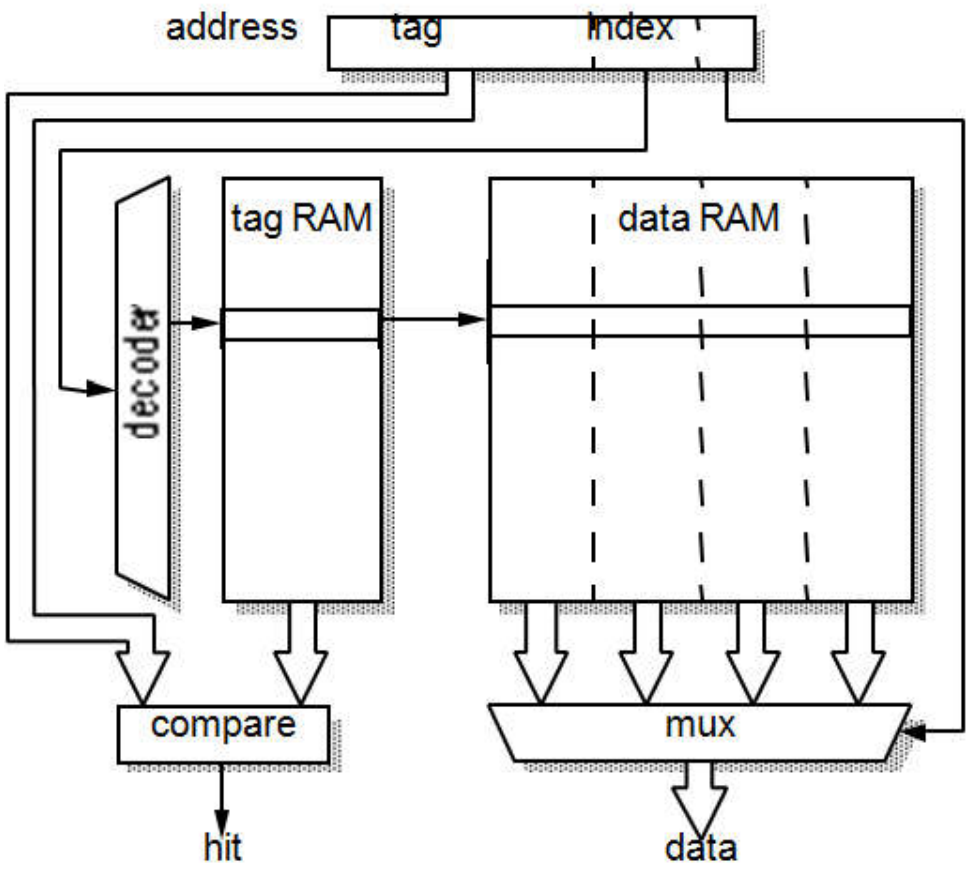

A
B
1
1
&
&
≥1
$F_{A<B}$
$F_{A=B}$
$F_{A>B}$

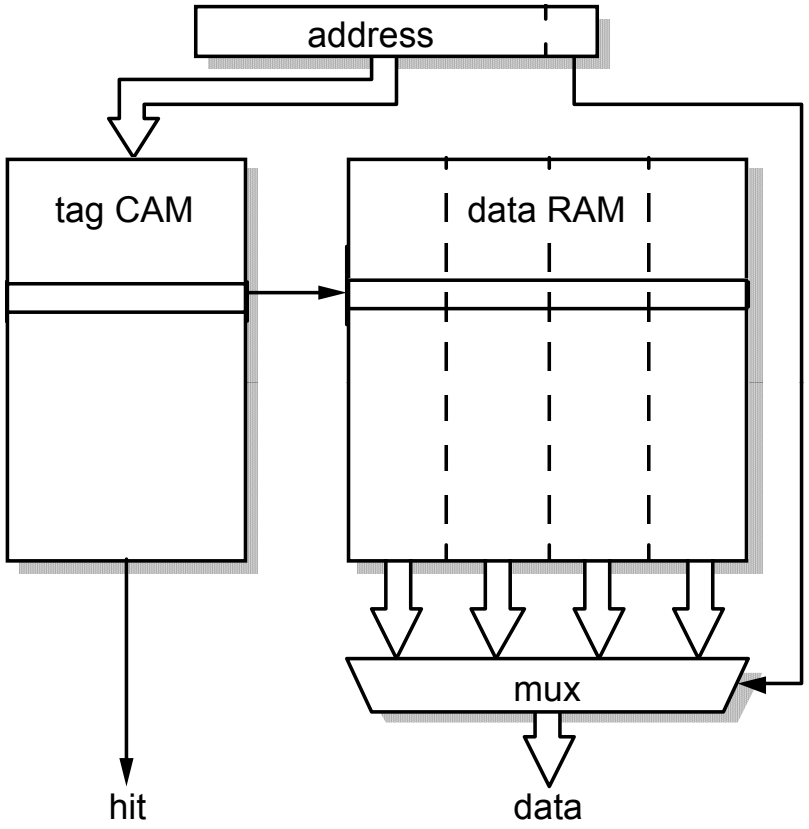

A_1
B_1
A_0
B_0
$A_1 < B_1$
$A_1 = B_1$
$A_1 > B_1$
$A_0 < B_0$
$A_0 = B_0$
$A_0 > B_0$
$I_{A>B}$
$I_{A<B}$
$I_{A=B}$
$F_{A>B}$
$F_{A<B}$
$F_{A=B}$


$F_{A=B}$ $F_{A<B}$ $F_{A>B}$ $I_{A>B}$
$I_{A<B}$
$I_{A=B}$
7485
B_3 A_3 B_2 A_2 B_1 A_1 B_0 A_0
B_3 A_3 B_2 A_2 B_1 A_1 B_0 A_0


$F_{A=B}$ $F_{A<B}$ $F_{A>B}$
7485 (2)
7485 (1)
B_7 A_7 B_6 A_6 B_5 A_5 B_4 A_4
B_3 A_3 B_2 A_2 B_1 A_1 B_0 A_0
0
0
1

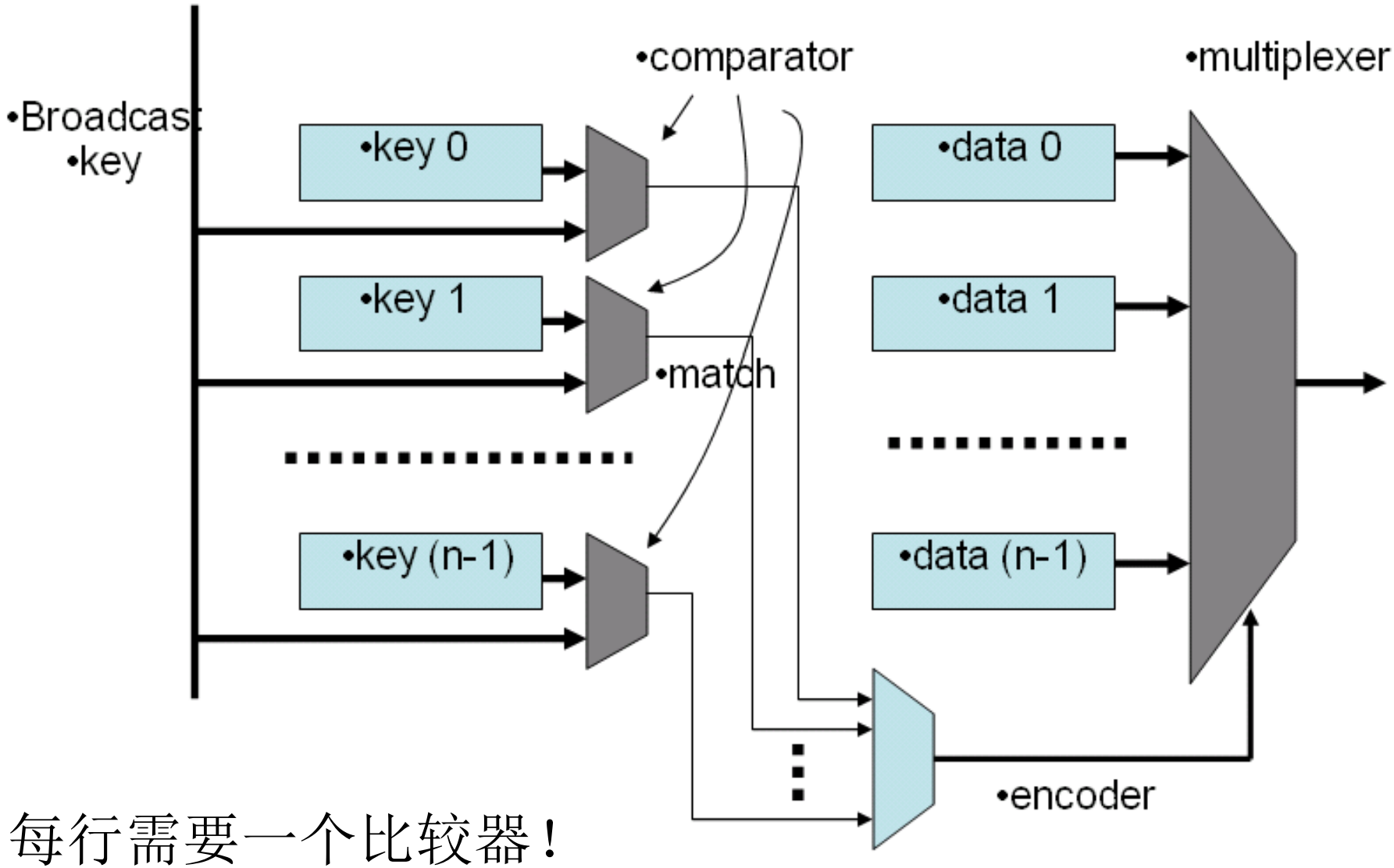
全相联Cache的结构





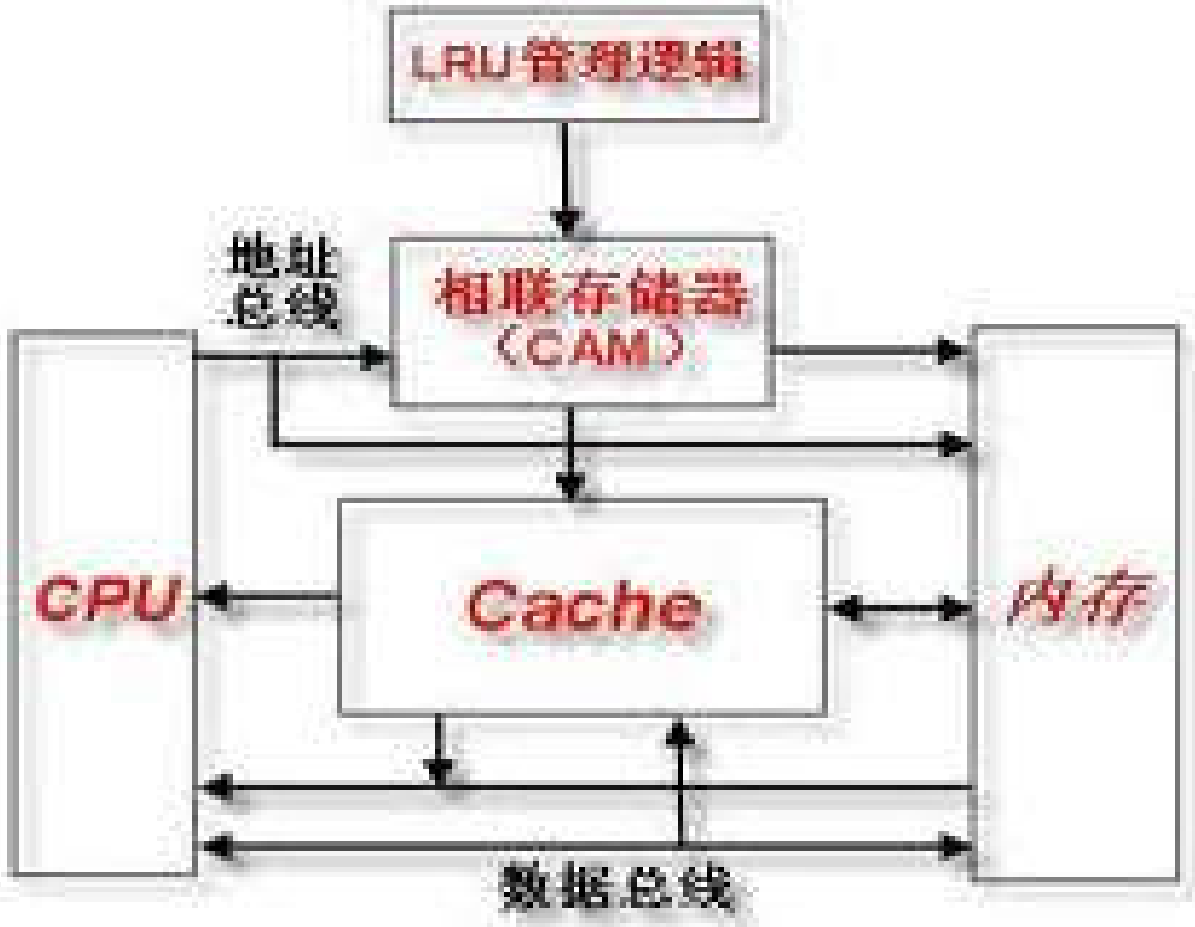
•按内容访问的存储器（CAM）

Associative Memory Structure



每行需要一个比较器！

全相联Cache



Cache工作原理图

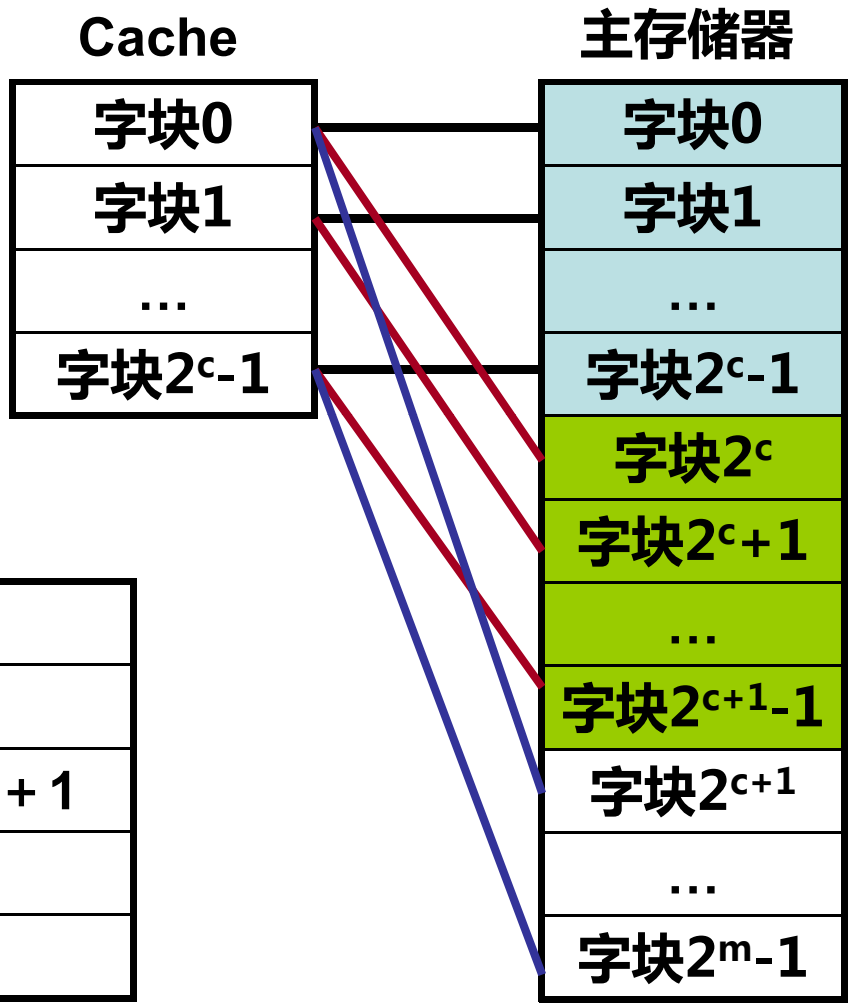
2. 直接映象：主存按Cache大小分段

Cache字块数为：$C=2^c$

主存字块数为：$M=2^m$

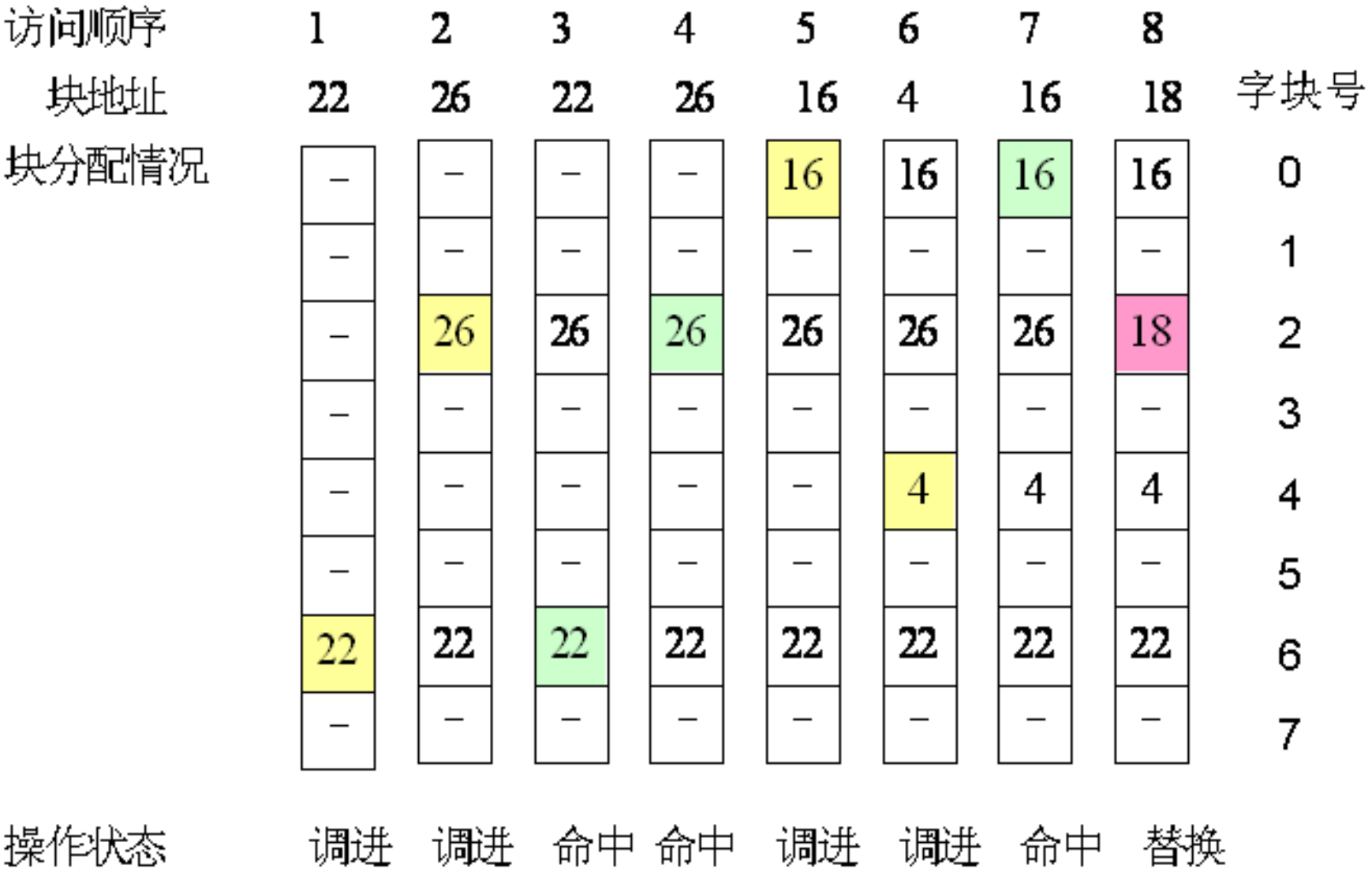
映射关系式：$i=j \bmod C$

或 $i=j \bmod 2^c$



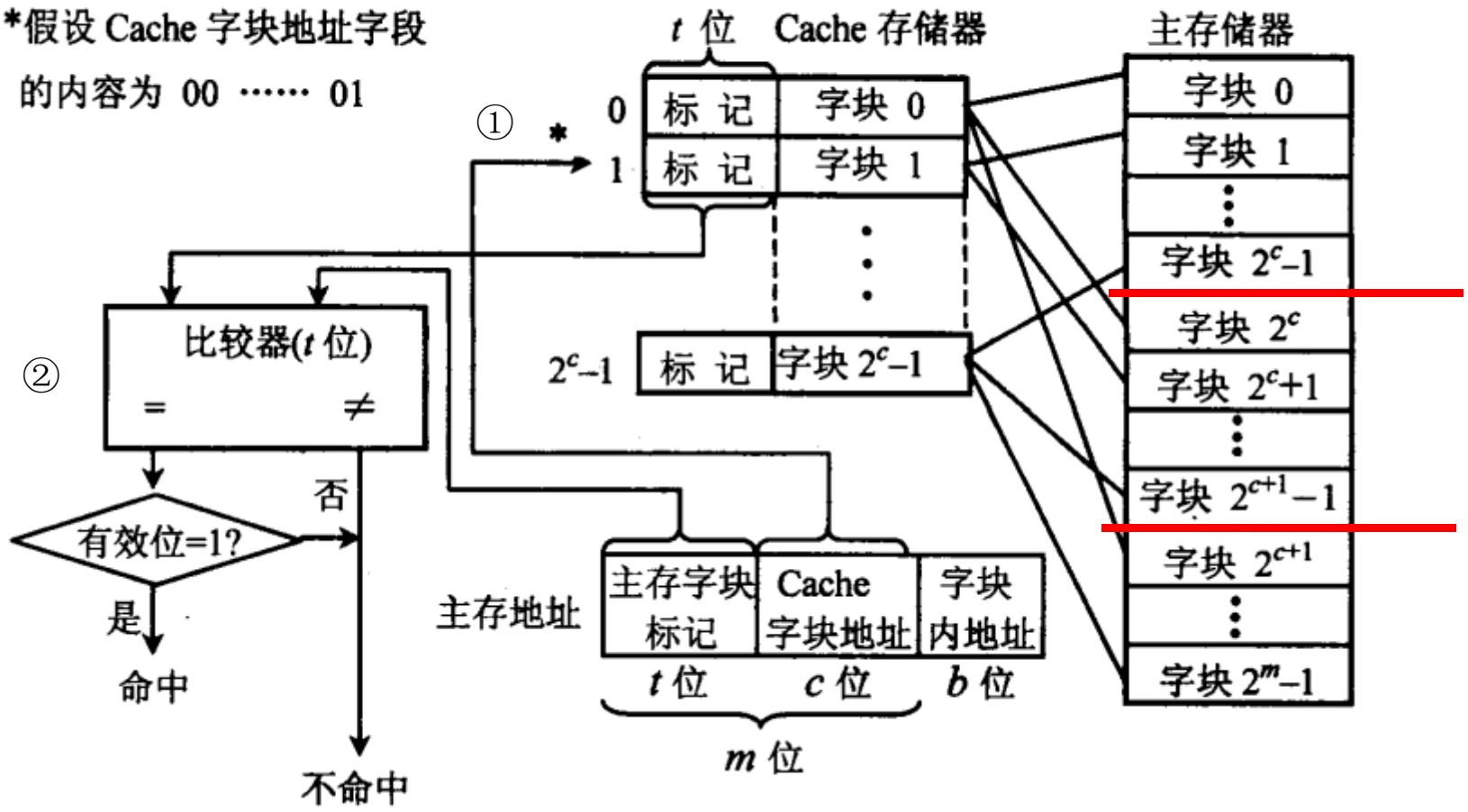
缓存块号i	主存块号j
0	0，C，……，2^m-C
1	1，C+1，……，2^m-C+1
……	……
C-1	C-1，2C-1，……，2^m-1

2. 直接映象（续）



如果连续访问26和18？

2. 直接映象命中比较过程

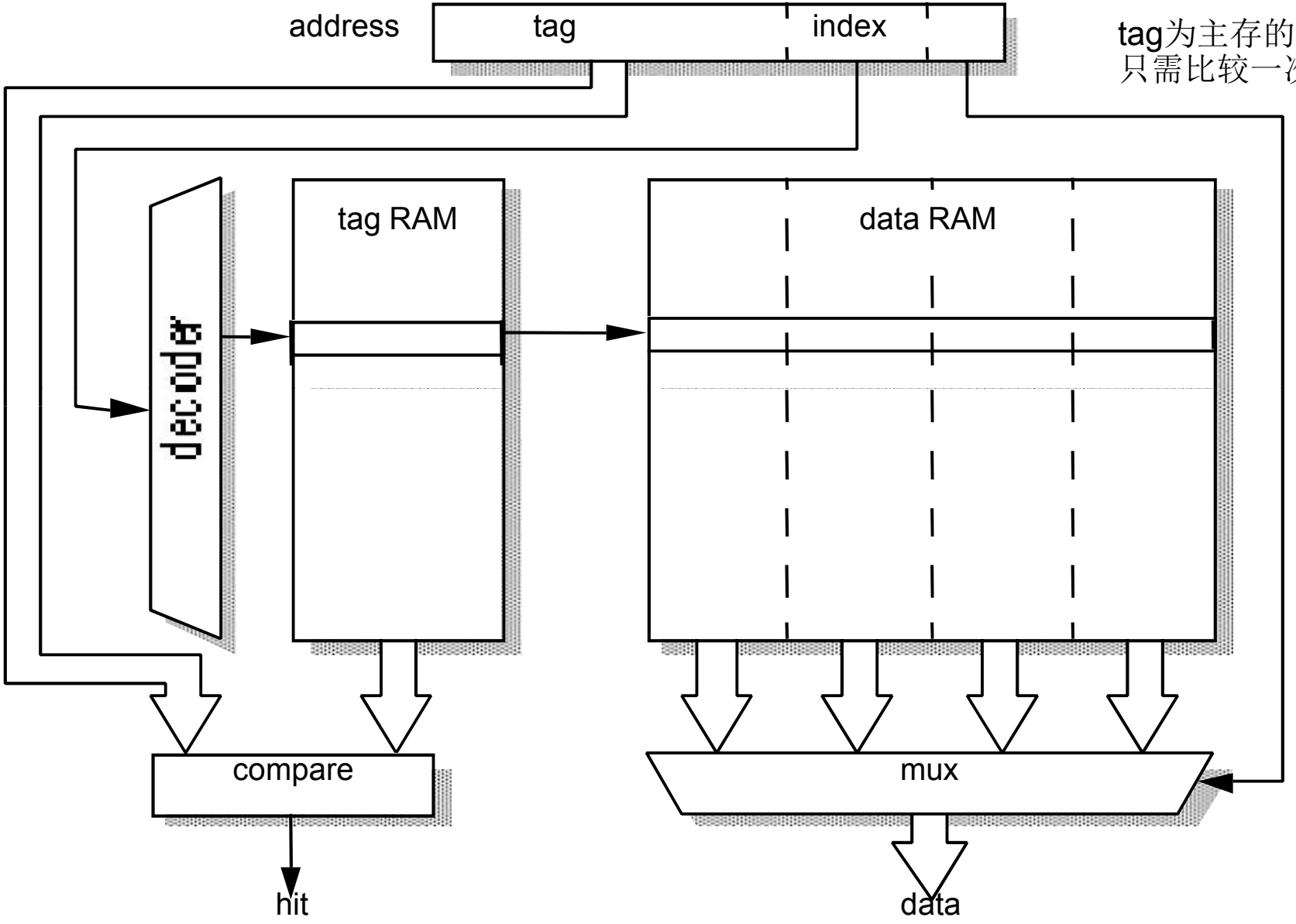


主存分段，段大小=Cache大小。段内块号与Cache块号（Cache字块地址c）一一对应。

根据c位找到对应的cache块（按c位index）。

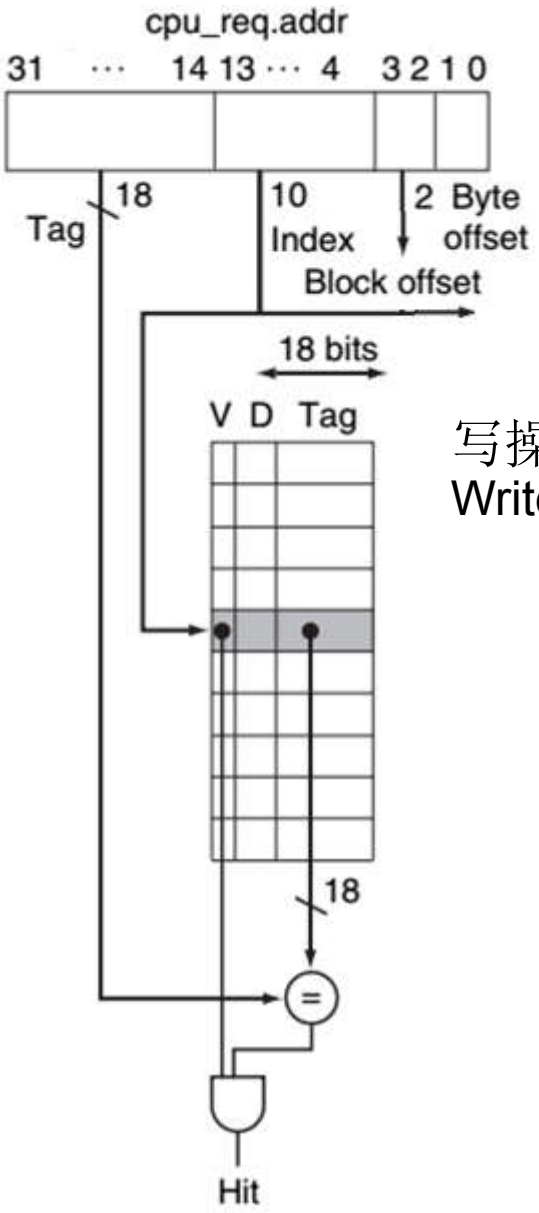
主存字块标记t为主存的段号。比较t位与tag（只需比较一次）！

直接映射cache结构



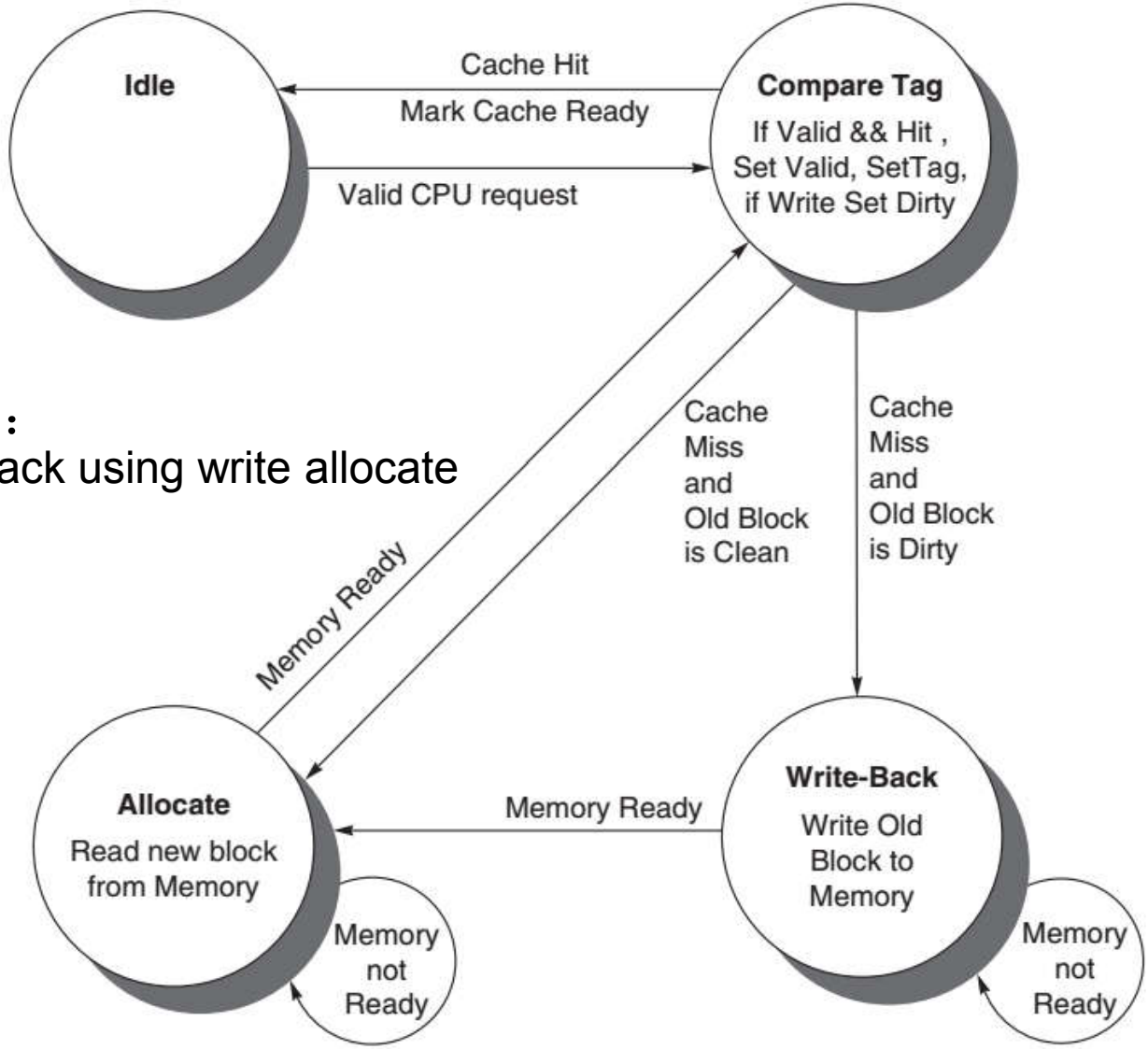
tag为主存的段号。
只需比较一次！

FSM for a Direct-mapped Cache Controller

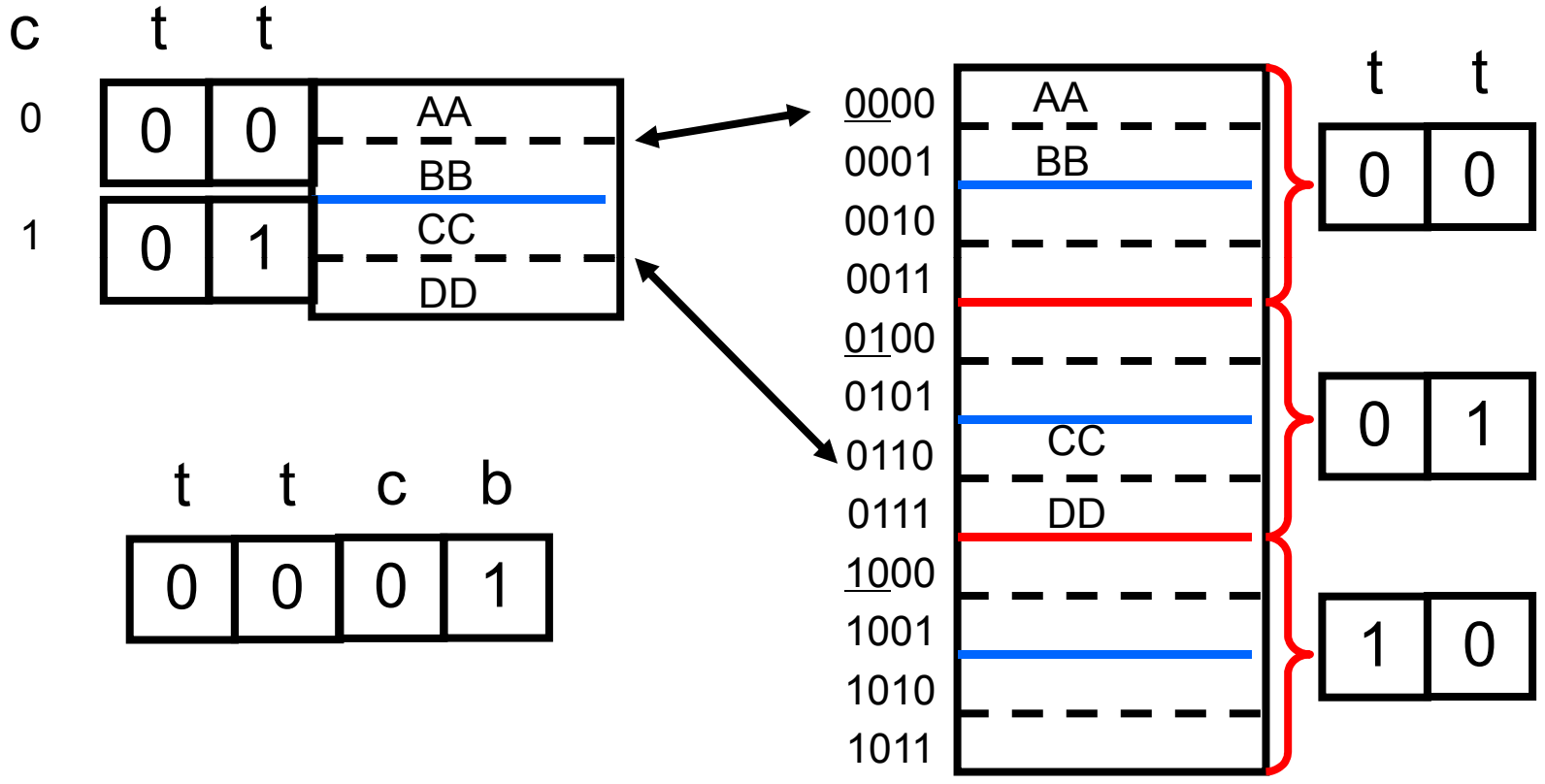


写操作：

Write-back using write allocate



- 块容量2B，cache容量4B，主存容量12B
  - 主存地址4位，其中b=1，c = 1，t = 2
  - 设访存地址为0001（内容为BB），命中否？



由m中的c位找到对应的cache块，比较其t位与tag，判断是否命中！

2. 直接映象（续）

- 优点：实现简单
  - 只需利用主存地址的某些位直接判断，就可确定所需字块是否在缓存中。

- 缺点：效率低。
  - 因为每个主存块固定地对应某个缓存块（有2^t个主存字块对应同一个Cache字块），如果这2^t个字块中有两个或两个以上的主存字块要调入Cache，必然会发生冲突。这时，即使Cache中还有很多空闲块也无法占用，使缓存的空间得不到充分的利用。

3. 组相联映象（2 way-set-associated）

Cache分组，

主存分段，

段大小＝组数

r = 组内块数－1

r = 0？直

r = c？全

Cache（r = 1）

标记	字块0
标记	字块1
标记	字块2
标记	字块3
……	……
标记	字块2^c-2
标记	字块2^c-1

第0组

第1组

第2^{c-r}-1组

主存储器

字块0

字块1

……

字块2^{c-r}-1

字块2^{c-r}

字块2^{c-r}+1

……

字块2^{c-r+1}

……

字块2^m-1

段

主存地址

主存子块标记	组地址	子块内地址
s=t+r位	q=c-r位	b位

m位

3. 组相联映象（续）

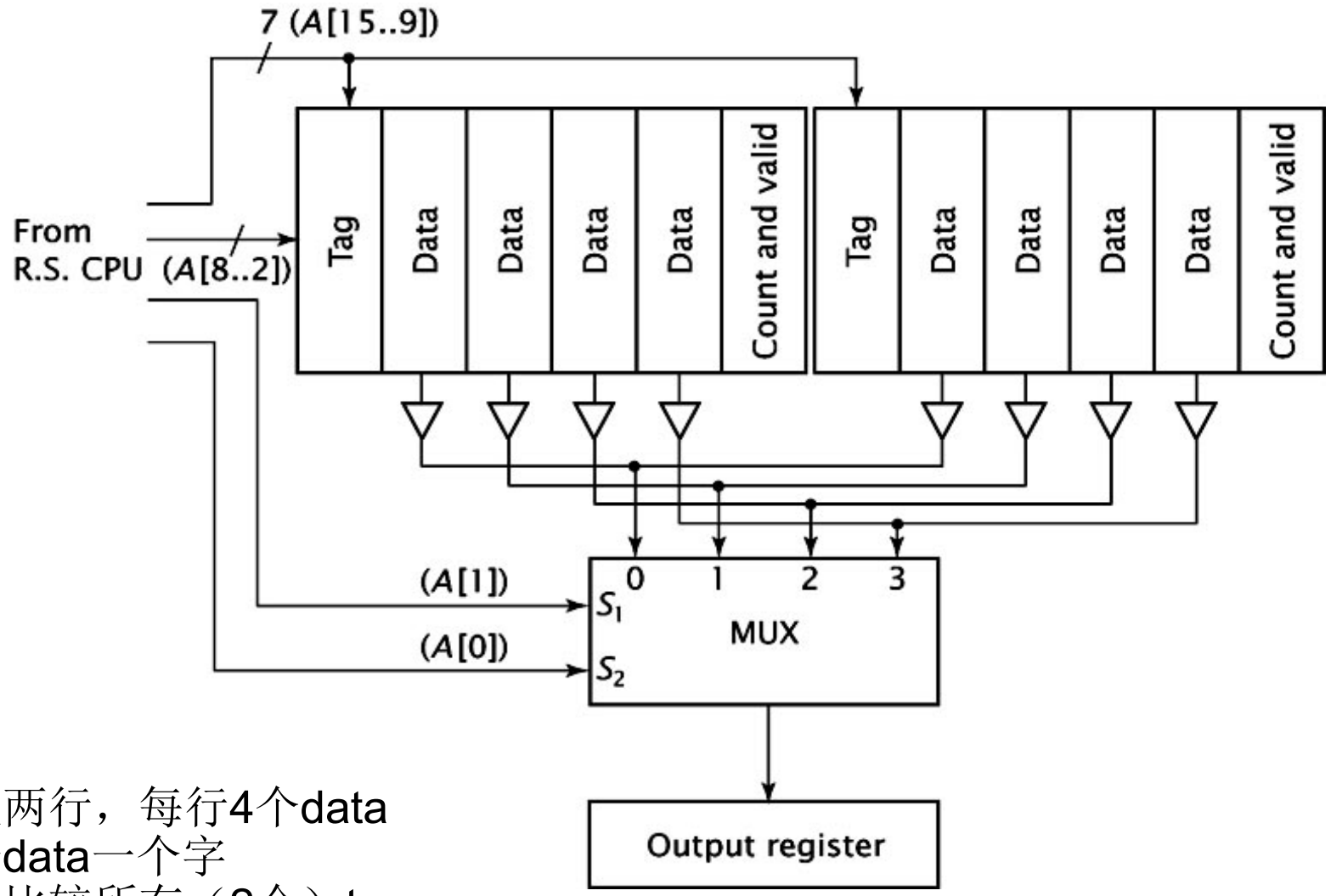
- 原理：把Cache分为Q(=2^q)组，每组有R(=2^r)块，且

$$i=j \bmod Q$$

- 其中，i为缓存的组号，j为主存的块号
  - r＝0，直接相联；r＝c，全相联。

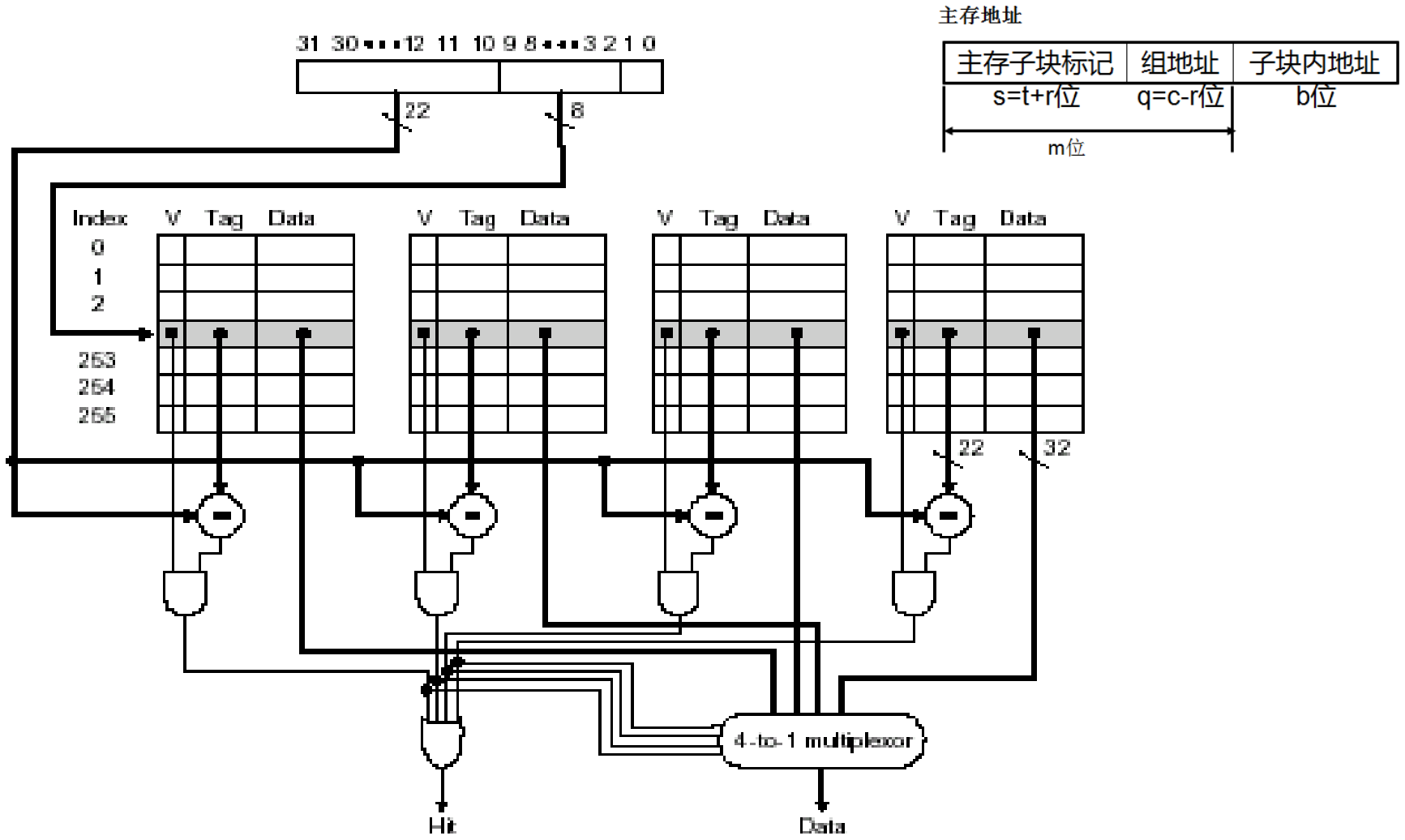
- **在主存块和Cache的各组之间，属于直接映象关系；而主存块可以映射到对应组内的任何一块，这又体现出了全相联映象的关系。**

2路组相联cache



- 每组两行，每行4个data
- 每个data一个字
- 需要比较所有（2个）tag

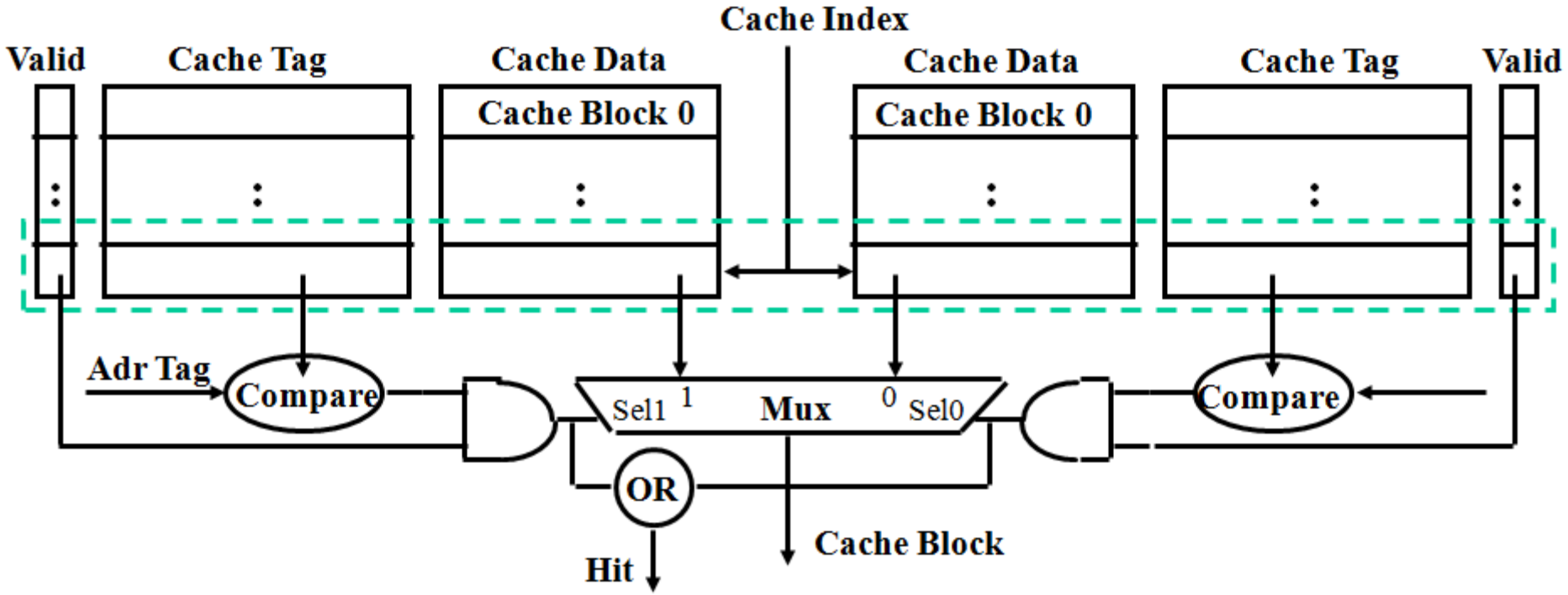
Example: 4-way set associative Cache



•块大小，组大小，Cache大小，内存段大小，内存大小？
•Tag比较次数？

Disadvantages of Set Associative Cache

- N-way Set Associative Cache vs. Direct Mapped Cache
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss decision and set selection
    - In a direct mapped cache, Cache Block is available BEFORE Hit/Miss



例：

- 某PC主存容量分2048块,每块512B,Cache容量8KB,分为16块,每块512B。
  - 用直接映象时，主存应被分几段？Cache标记几位？
  - 用全相联映象，Cache标记几位？
  - 用组相联映象，Cache每组2块（即：两路组相联），主存应划分为几段？每段几块？Cache标记几位？

例：设有一个cache的容量为2K字，每个块为16字，求(1) 该cache可容纳多少个块？

(2) 如果主存的容量是256K字，则有多少个块？

(3) 主存的地址有多少位？cache地址有多少位？

(4) 在直接映像方式下，主存中的第*i*块映像到cache中哪一个块中？

(5) 进行地址映像时，存储器的地址分成哪几段？各段分别有多少位

解：(1) cache中有2048/16=128个块。

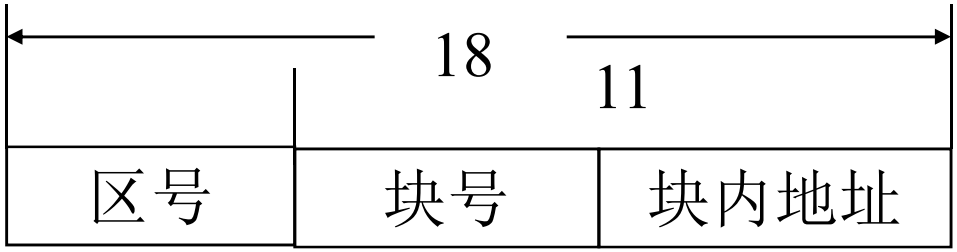
(2) 主存有256K/16=16384个块。

(3) 主存容量为256K=2^{18}字，字地址有18位。

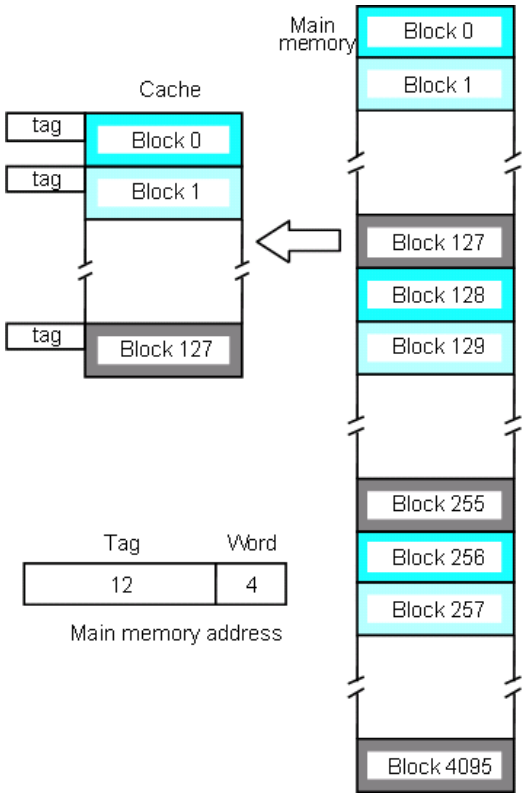
cache容量为2K=2^{11}字，字地址为11位。

(4) 在直接映像方式下，主存中的第*i*块映像到cache中第 $i \bmod 128$个块中。

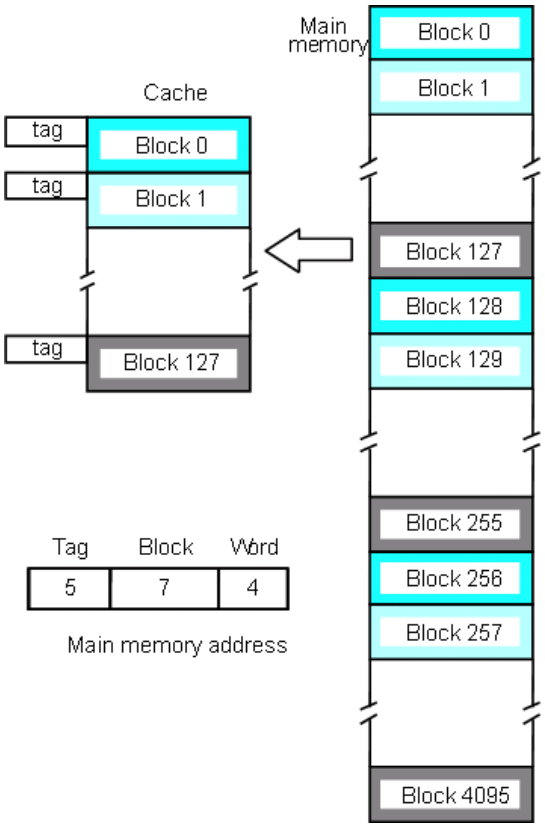
(5) 区号7位，块号为7位，块内字地址为4位。



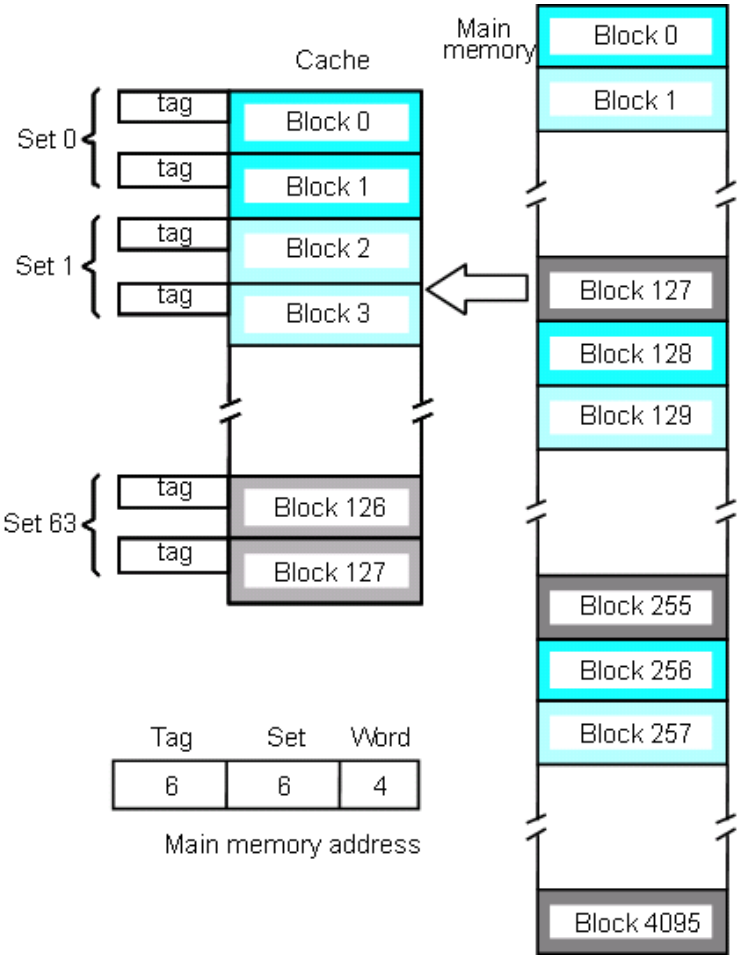
映象方式比较



•Associative mapping

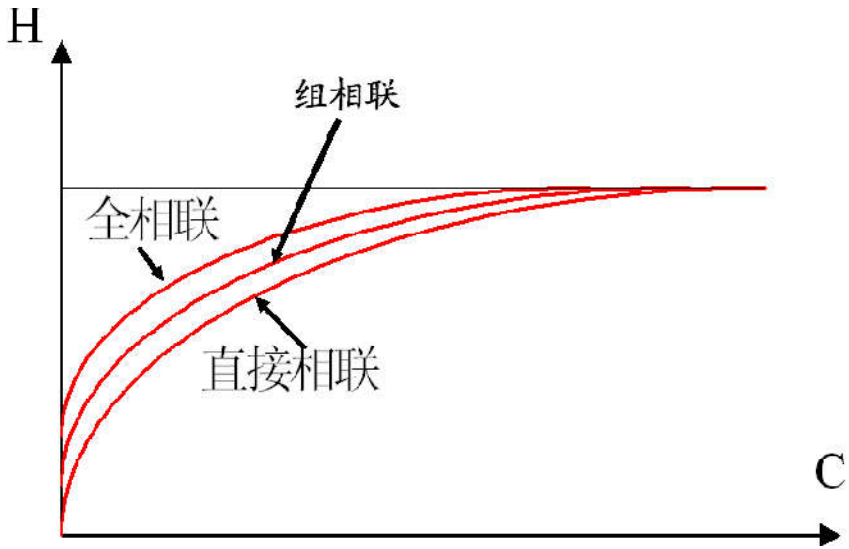


•Direct mapping

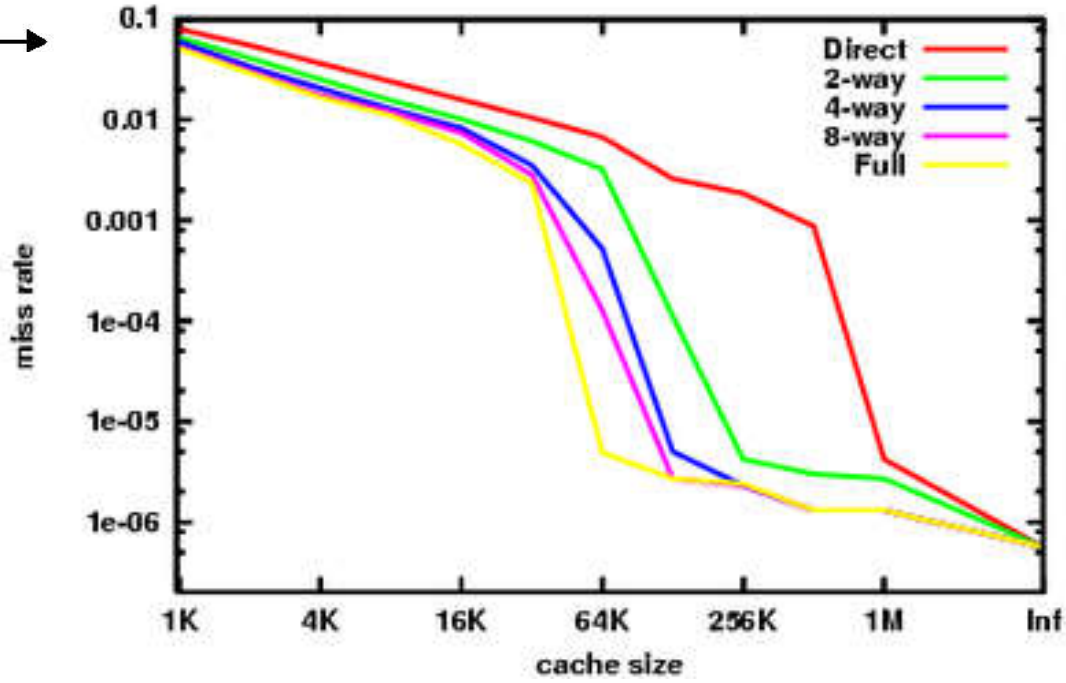


•2 ways Set-Associative mapping

Cache的性能和类型— 地址映象与命中率



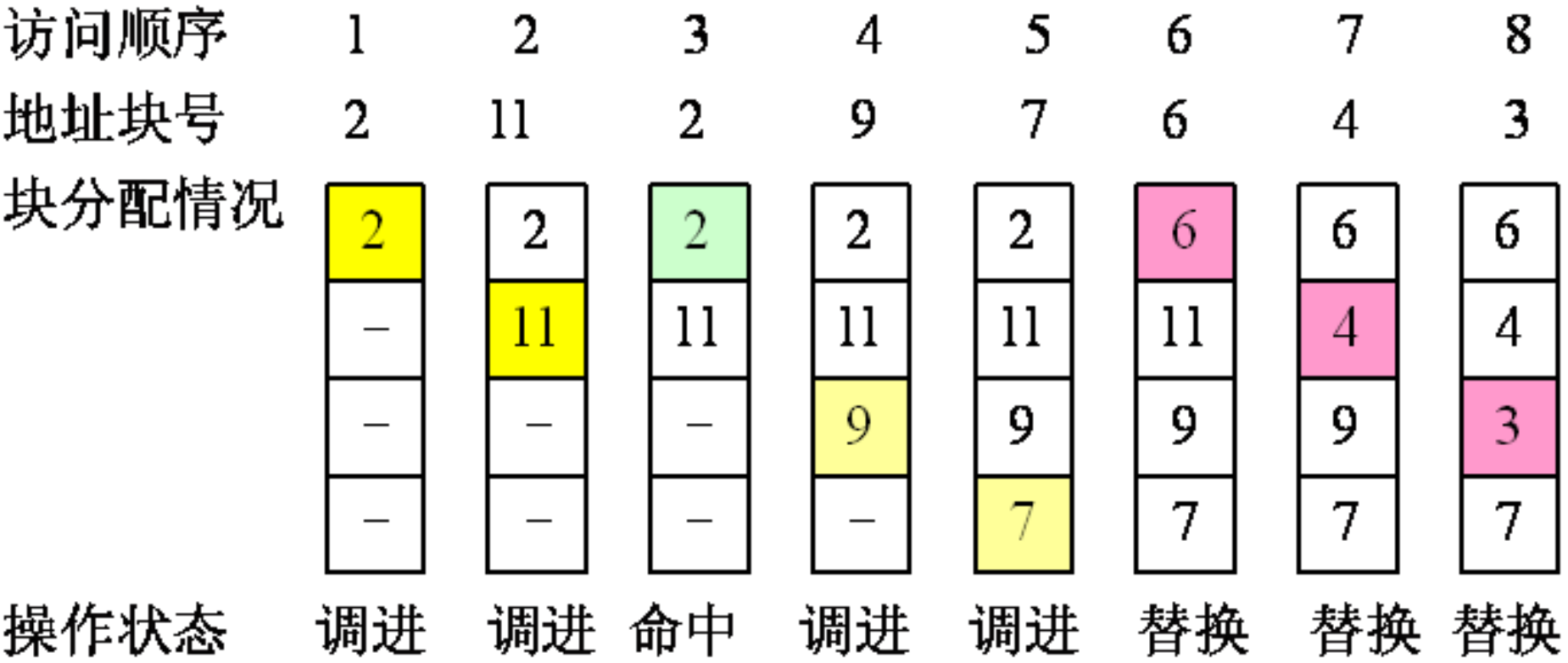
4路以上效果不显著



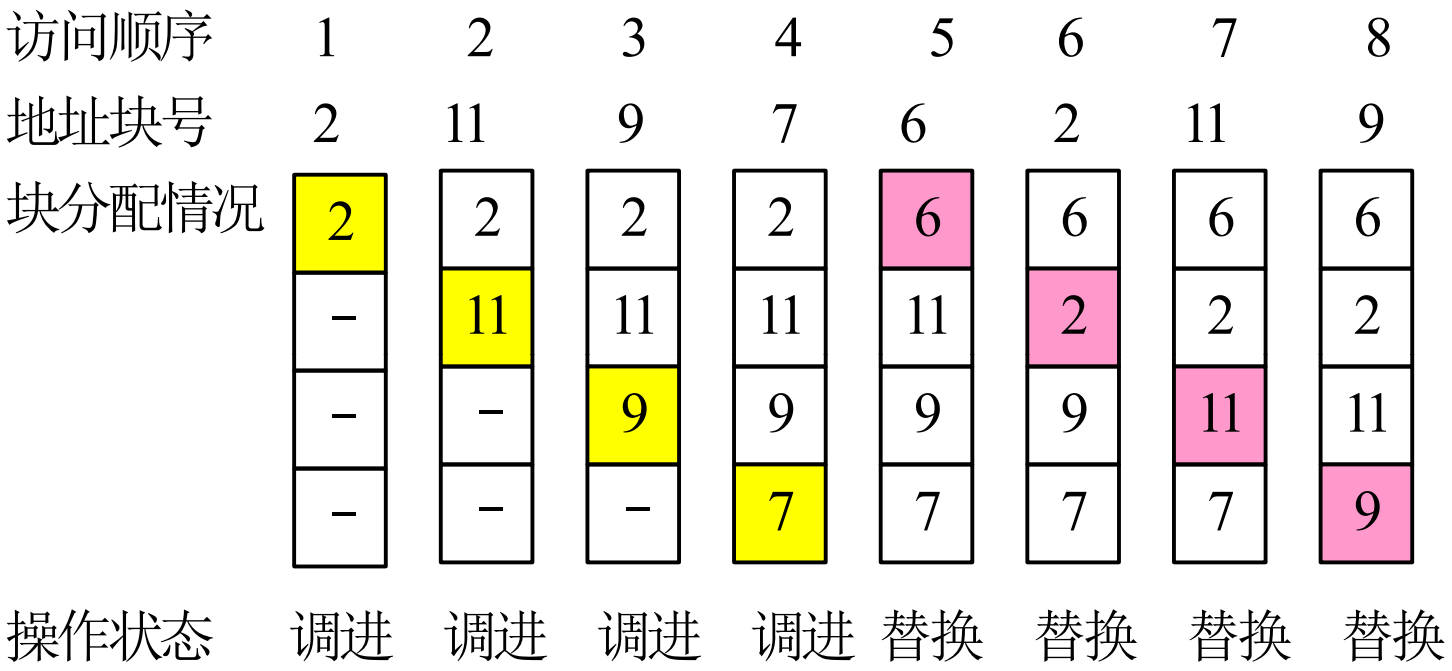
替换算法

- **随机替换**
- **基于局部性原理**
  - **最优替换算法（OPT）：未来最不可能使用者**
    - 置换最长时间中不会被使用的页：预知工作集（work sets）
  - **FIFO：实现方便，但不能正确反映程序的局部性**
    - **最先进入的字块也可能是目前经常要用的字块。**
  - **LRU，最少近期使用，**最近最久未使用
    - **能比较正确反映程序的局部性，因为当前最少使用的块一般来说也是未来最少被访问的块。**
    - **开销：具体实现比FIFO复杂**
      - **计时法（绝对）：替换计时最长的cache line**
      - **计数法（近似）：替换计数最大的cache line**
        - » **两位标志位；一位标志位（NRU，**最近未使用**）**
      - **堆栈法**
  - **直接映射？**

FIFO替换算法


访问顺序 1 2 3 4 5 6 7 8
地址块号 2 11 2 9 7 6 4 3
块分配情况
2 2 2 2 2 6 6 6
– 11 11 11 11 11 4 4
– – – 9 9 9 9 3
– – – – 7 7 7 7
操作状态 调进 调进 命中 调进 调进 替换 替换 替换

颠簸现象



先进先出替换方式下的 cache 内容颠簸情况

近期最少使用算法LRU

例：选最近4次访问期间最少使用Cache块作为被替换的块。

访问顺序	1	2	3	4	5	6	7	8
地址块号	2	11	2	9	7	6	4	3
块分配情况	2	2	2	2	2	2*	4	4
	–	11	11	11	11*	6	6	6
	–	–	–	9	9	9	9*	3
	–	–	–	–	7	7	7	7*
操作状态	调进	调进	命中	调进	调进	替换	替换	替换

*表示将要被替换者。

计数法实现：各块的LRU位变化？

例子：

设程序有5个信息块， Cache空间为3块，地址流为：

P1, P2, P1, P5, P5, P1, P3, P4, P3, P4

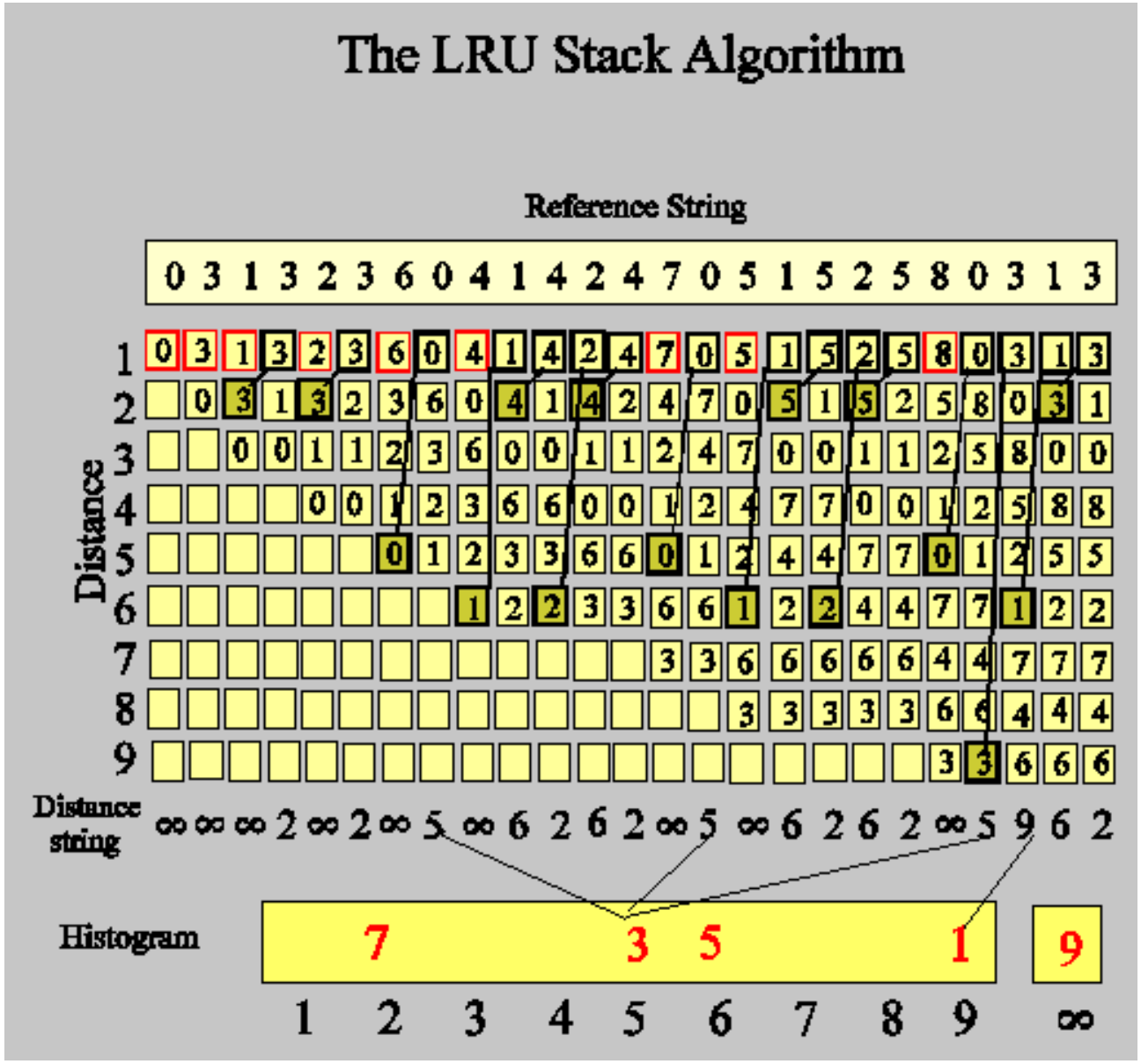
给出FIFO、LRU两种页面替换算法对这3块Cache的使用情况，包括调入、替换和命中等。

时间 t	1	2	3	4	5	6	7	8	9	10	实际命中次数
地址流	P1	P2	P1	P5	P4	P1	P3	P4	P2	P4	
先进先出算法（FIFO 算法）	1	1	1	1*	4	4	4*	4*	2	2	
		2	2	2	2*	1	1	1	1*	4	
				5	5	5*	3	3	3	3*	
	调入	调入	命中	调入	替换	替换	替换	命中	替换	替换	2 次
最近最少使用算法（LRU 算法）	1	1	1	1	1	1	1	1*	2	2	
		2	2	2*	4	4	4*	4	4	4	
				5	5*	5*	3	3	3*	3*	
	调入	调入	命中	调入	替换	命中	替换	命中	替换	命中	4 次
最优替换算法（OPT 算法）	1	1	1	1	1	1*	3*	3*	3	3	
		2	2	2	2*	2	2	2	2	2	
				5*	4	4	4	4	4	4	
	调入	调入	命中	调入	替换	命中	替换	命中	命中	命中	5 次

三种页面替换算法对同一个页地址流的调度过程

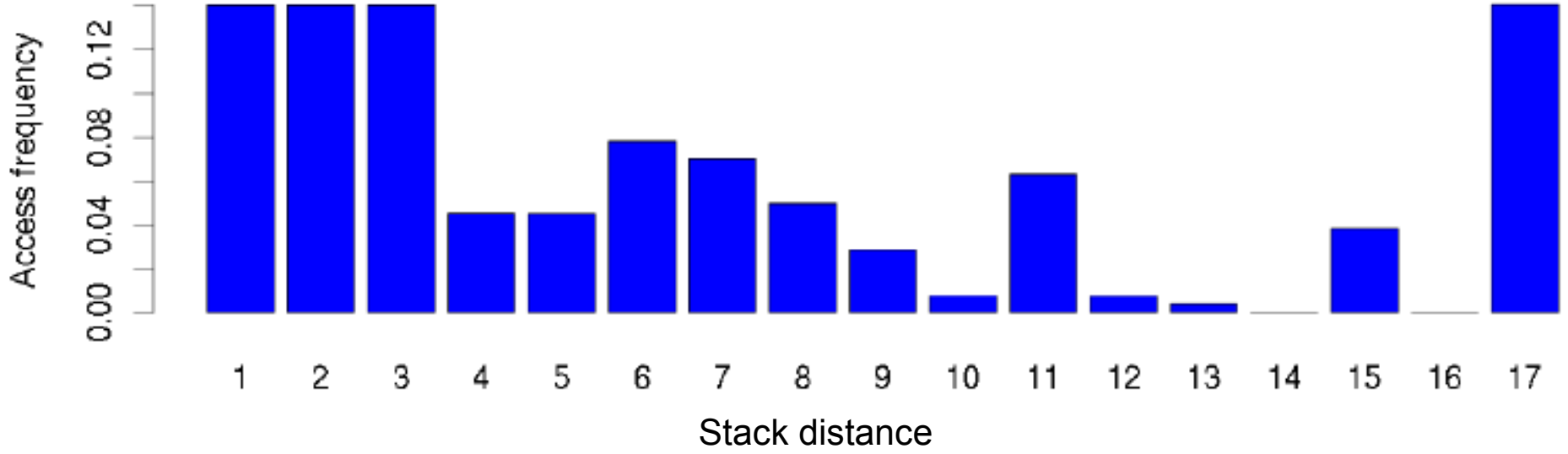
LRU stack算法

- stack distance
  - 访问某line时其在栈中的位置
  - 第一次访问时 = ∞
- 替换哪一行？
- histogram
  - 局部性



Stack distance profiles： histogram

- Stack distance of an access in a cache set
  - the position of the line in the LRU stack when this line is accessed（定义：访问此line时它在LRU栈中的位置）
  - = number of distinct cache lines accessed since previous access to the same line（访问同一line的间隔次数）
- Stack distance profile (histogram) of a cache set
  - counts (frequencies) of accesses depending on their stack distance（各距离的总数）



Cache Misses：4种（COD为“3C”）

- Compulsory
  - cold start or process migration, first reference
  - “Cold” fact of life: not a whole lot you can do about it
  - Note: If you are going to run “billions” of instruction, Compulsory Misses are insignificant
- Capacity
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- *Coherence (Invalidation):*
  - *other process (e.g., I/O) updates memory*

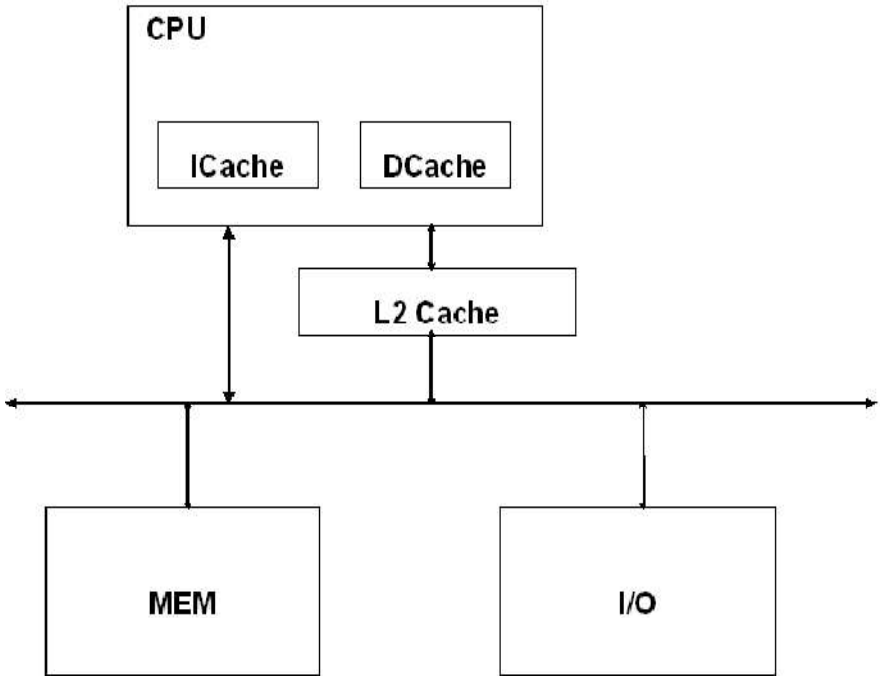
Blocking（阻塞式）Cache

- 有效位（Valid）
  - cold start or process migration, first reference
  - 写操作的使无效法（Invalidated）
- 重写位（Dirty）：
  - 数据块被换出时是否需要写回mem

- Cache miss时需要处理器与Cache控制器协同
  - Cache miss时，处理器（流水线）被阻塞等待
  - 流水线指令cache缺失时的处理步骤：
    - PC – 4
    - 通知主存执行一次读操作，等待主存访问完成
    - 写cache项，设置有效位
    - 重新取指
  - 流水线数据cache缺失时的处理：类似

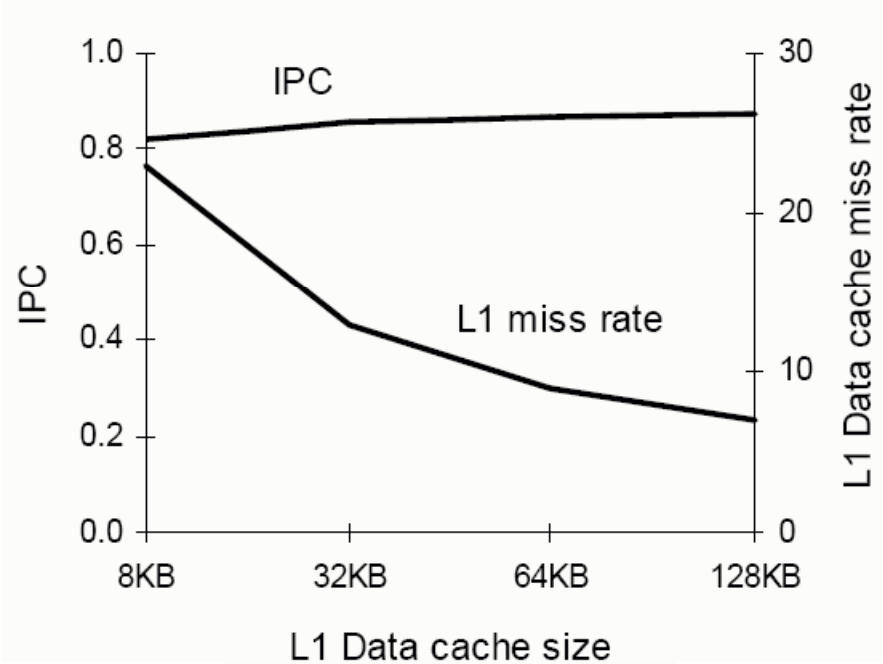
多级Cache与哈佛结构

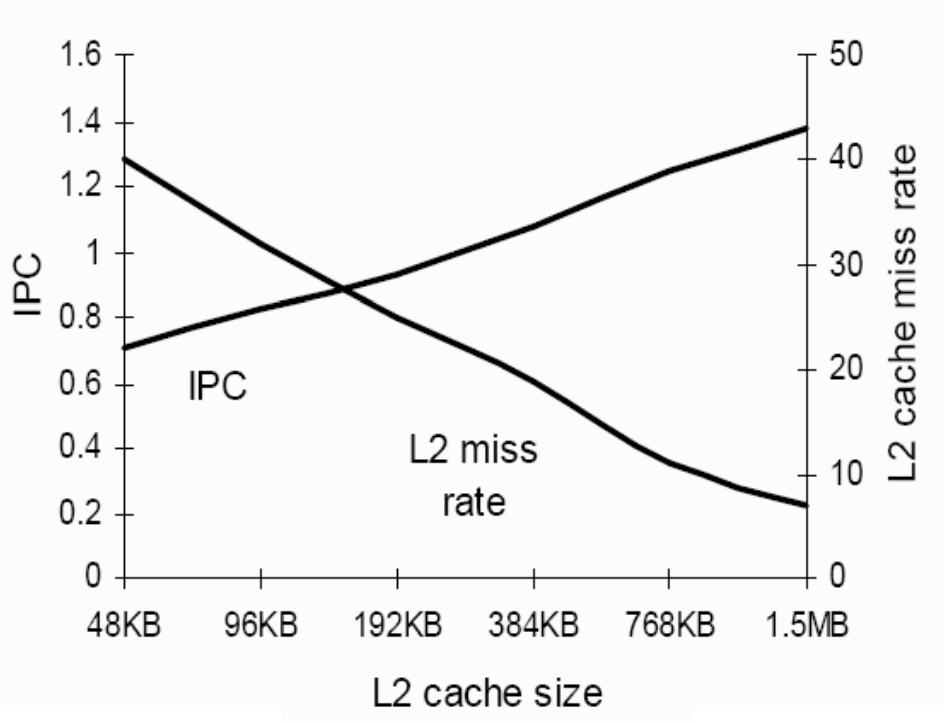
- 两层存储结构的存储访问时间
  - H为Cache命中率，T1和T2分别为两层存储器的访问时间，则系统访问时间Ts
    - Ts = T1 × H + （1 - H）×（T1 + T2）



多级Cache的影响

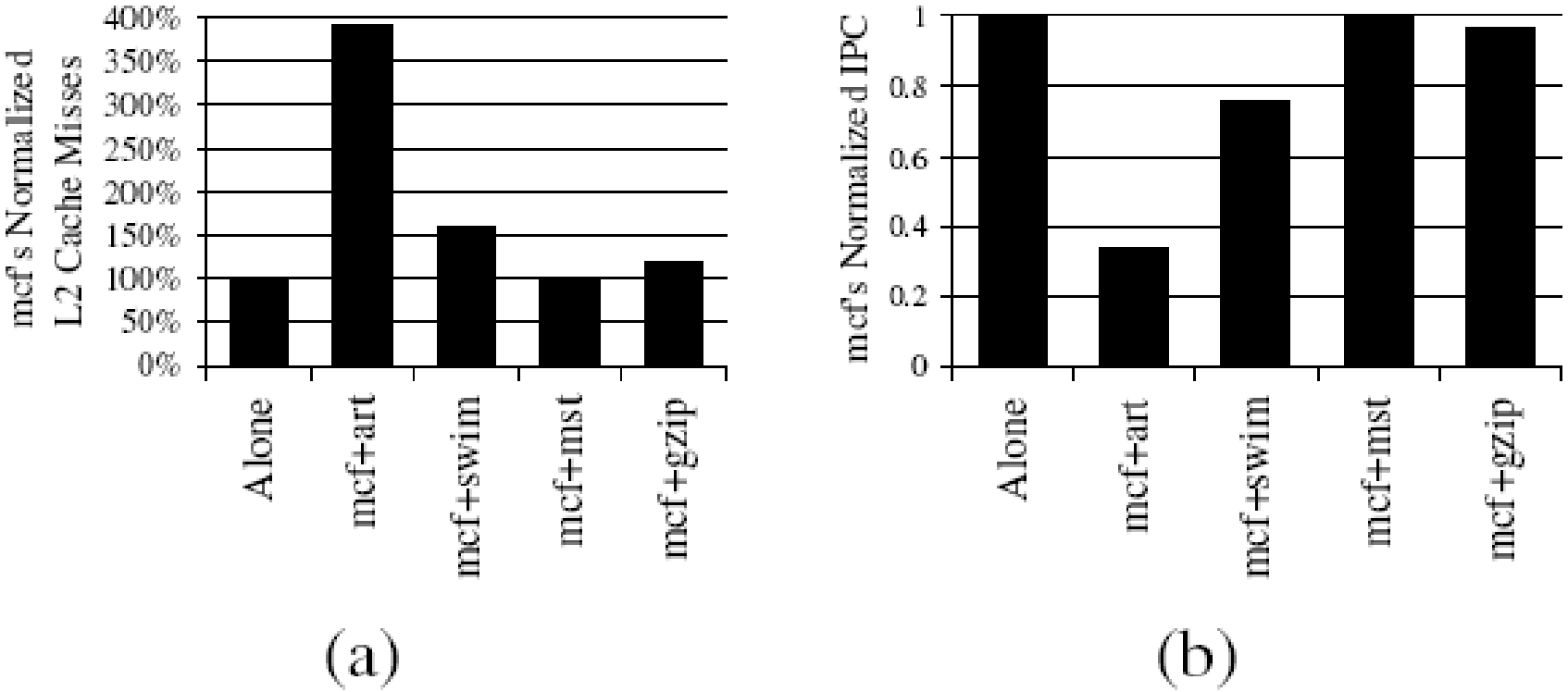
- L1 Cache几乎对IPC没有影响，L2影响很大





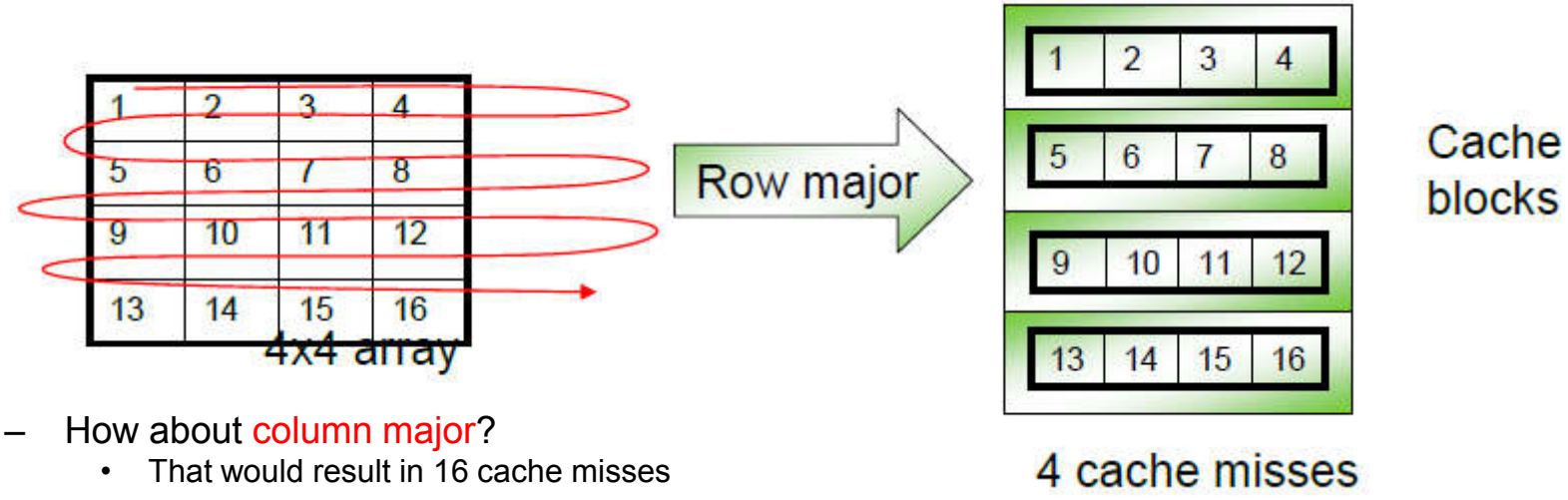
多线程共享Cache的效果

- 不同线程组合
  - 运行于不同处理器核，但共享L2 Cache



Cache Effects

- Cache affinity ： 冷热、预热
  - 尽可能多的对已读取的数据进行操作，最大限度的发挥时间局部性；
- 注意循环：大部分计算和访存都发生在这里
  - 按照数据对象在存储器中的存放顺序读取数据，从而最大限度的发挥空间局部性；
  - Suppose
    - storing multidimensional arrays in linear memory
    - a program accesses the array one row at a time.



  - How about column major?
    - That would result in 16 cache misses

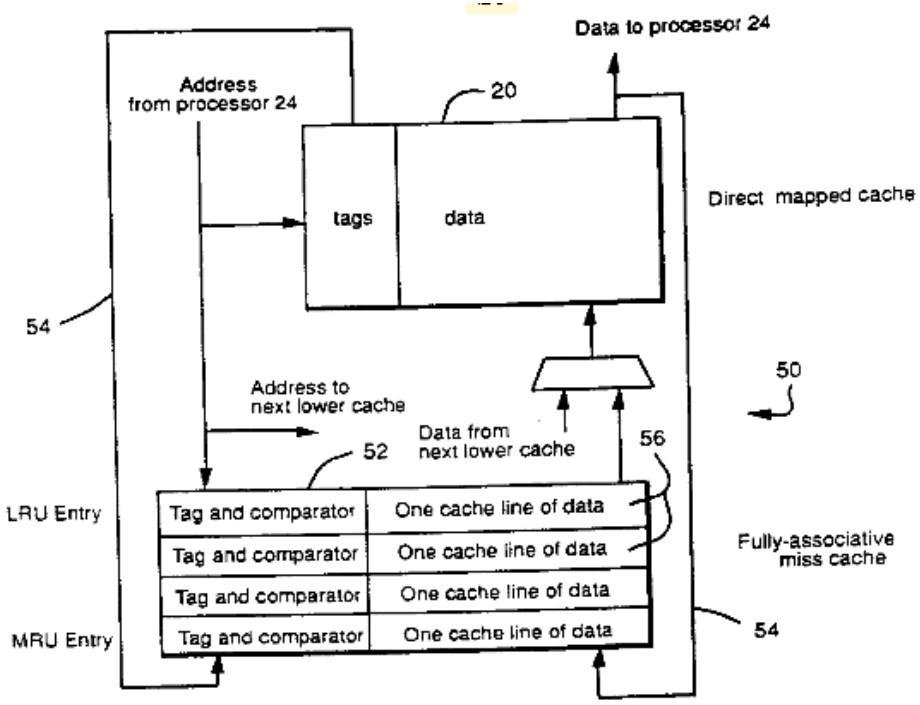
注意：cache的容量为一行！

基于Cache的代码优化

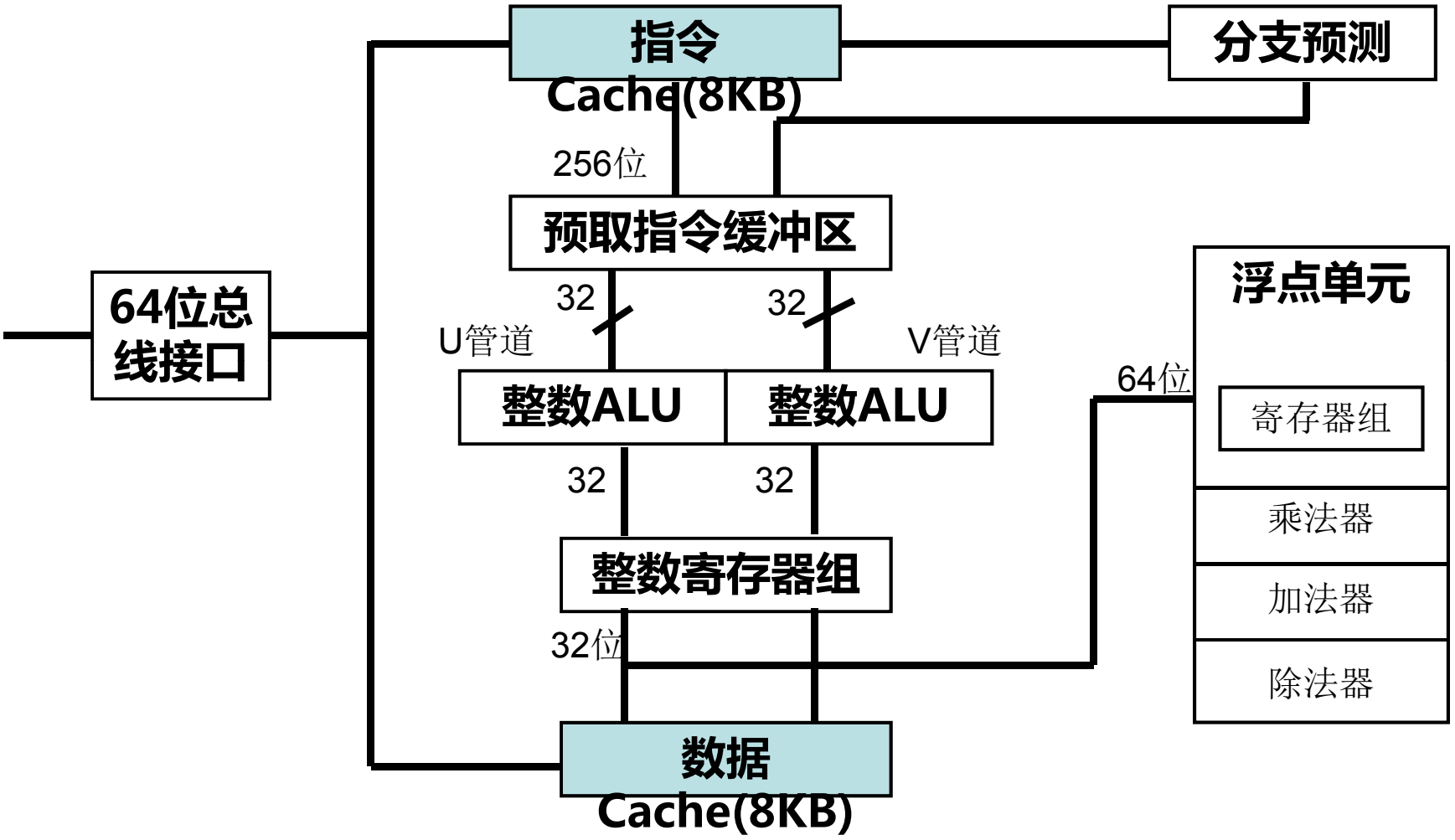
- *循环交换*（Loop Interchange）
  - 原程序(column major)
    - a[100][5000]=...//初始化
      for(j=0; j<5000; j=j+1) {
        for(i=0; i<100; i=i+1) {
          a[i][j] = 2 * a[i][j]; 每次都不命中
        }
      }
  - 改进 (row major)
    - a[100][5000]=...//初始化
      for(i=0; i<100; i=i+1) {
        for(j=0; j<5000; j=j+1) {
          a[i][j] = 2 * a[i][j]; 可连续命中若干次[cache行大小]
        }
      }
- *循环合并*（Loop fusion）
- *循环分块*（Blocking）
- **Cache-oblivious algorithm：与cache结构无关的算法**

提高Cache性能：Victim Cache

- 对于RISC处理器，缺失时损失100多个周期
- Victim Cache：介于L1 Cache和下一级存储器之间
  - L1 Cache采用直接相连，较大
  - Victim Cache采用全相连，较小，存放由于失效而被替换出的块
  - 含有4个块的缺失cache可以使一个4KB的直接映像数据cache的冲突失效减少20～90%

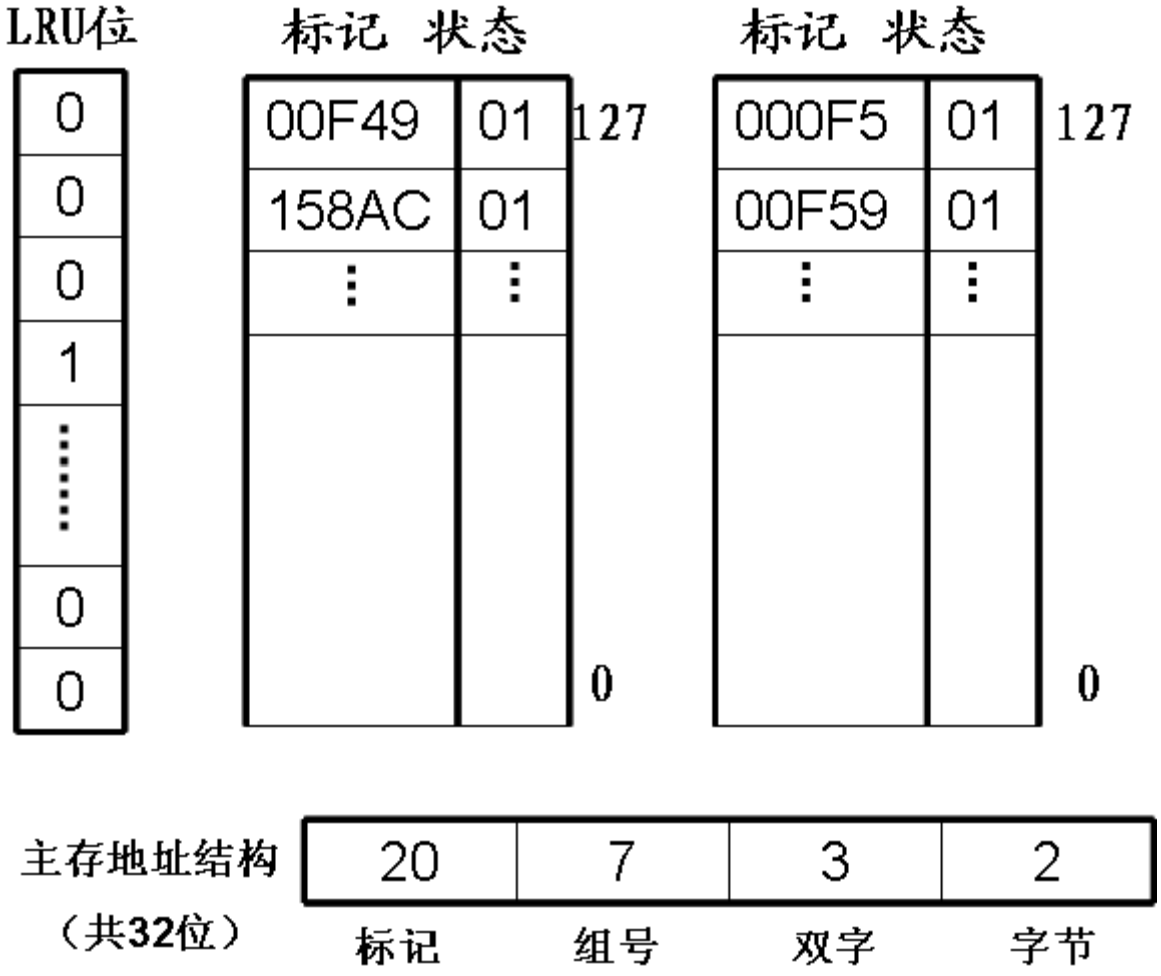


Pentium处理器框图



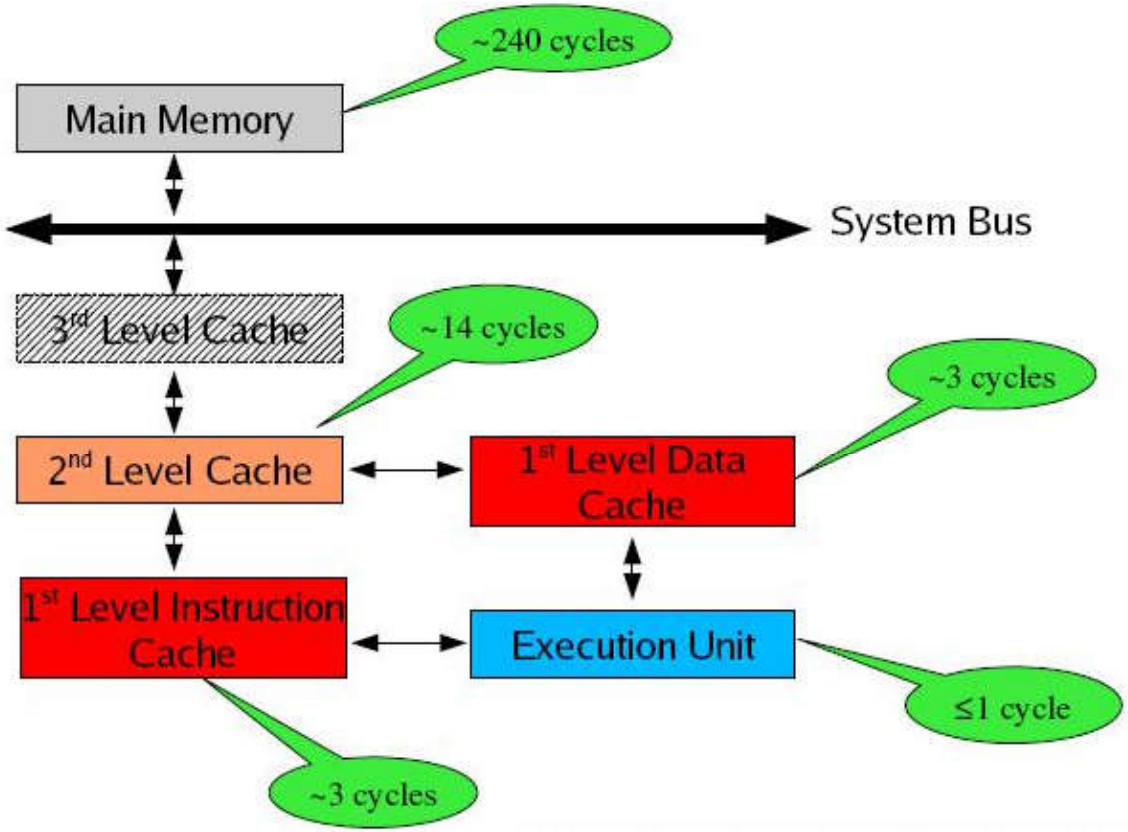
Pentium处理器的片内Cache

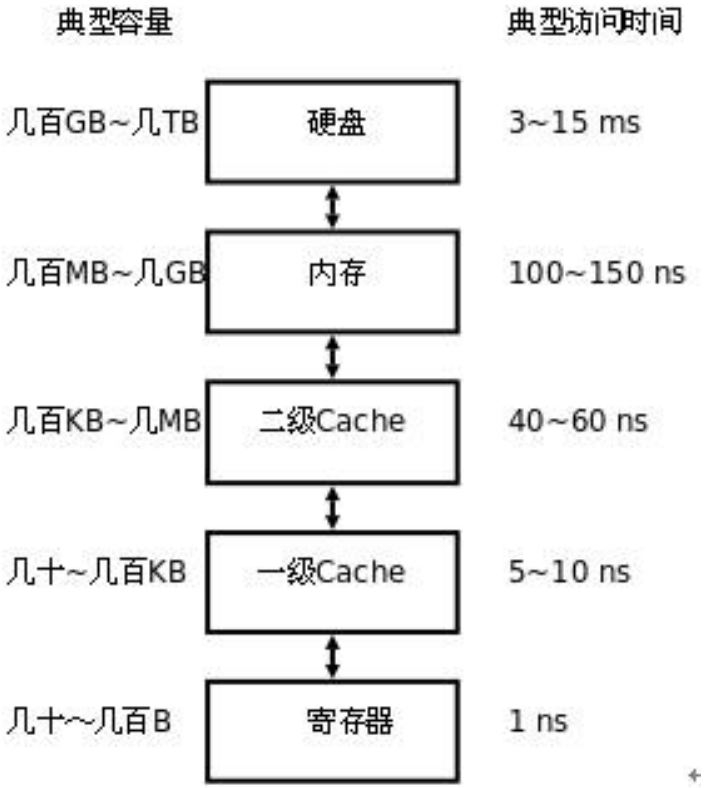
- 两路组相连: 共128组，每组2页(line)，每页8个双字（4X8＝32B）。
- 采用“写回”策略(可以动态重构支持“写直达法”)
- 有两条单独的指令来清除或回写Cache

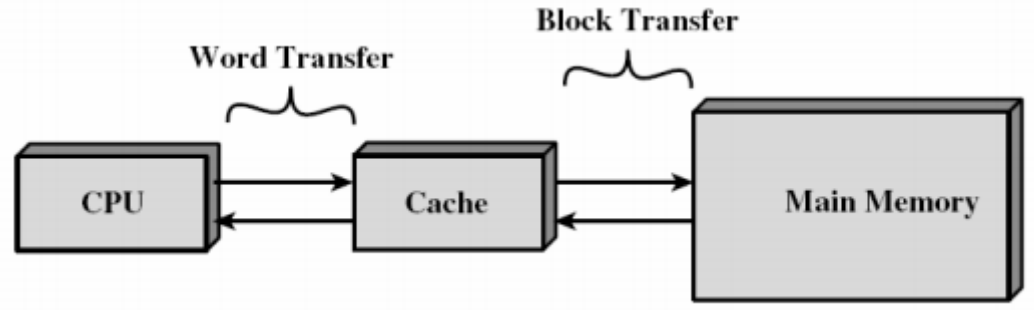


- **状态位：**共有4种不同状态，在Cache一致性协议（MESI）中使用。

Cache对系统性能的影响







Intel® Processor the CPUID Instruction

```
#include<stdio.h>
#include<stdlib.h>

int d_eax;
int d_ebx;
int d_ecx;
int d_edx;

int parse_cache()
{

        asm
          (
          "movl $2,%eax\n\t"
          "cpuid\n\t"
          "mov  %eax,d_eax\n\t"
          "mov  %ebx,d_ebx\n\t"
          "mov  %ecx,d_ecx\n\t"
          "mov  %edx,d_edx\n\t"
          );

        printf("d_eax : %x\nd_ebx : %x\nd_ecx : %x\nd_edx : %x\n",
                d_eax,d_ebx,d_ecx,d_edx);
        return 0;

}
int main()
{
        parse_cache();
        return 0;
}
```

小结

- Cache有效性
  - Cache效率和局部性的度量指标？
  - Cache miss的原因？
  - 发挥Cache的作用：利用局部性
  - Cache的Side effect
    - 一致性：如何使Cache与主存内容保持一致。
      - 单CPU，多CPU，DMA
    - 实时性：访存时间的确定性
- Cache组织结构，读写过程，映射机制，替换策略
  - 为何需要不同的映射模式？
    - 全相连：“多次比较或多个比较器”，可使用CAM
    - 直接映射：“一次比较，一个比较器”
    - N-way set: “一次比较，N个比较器”
  - 三种映射方式各自需要哪种置换算法？
  - 编程实现LRU？
- 作业：5.2.1～3，5.3.1，5.8.1～2


Thank You