7.3 高速缓冲存储器cache

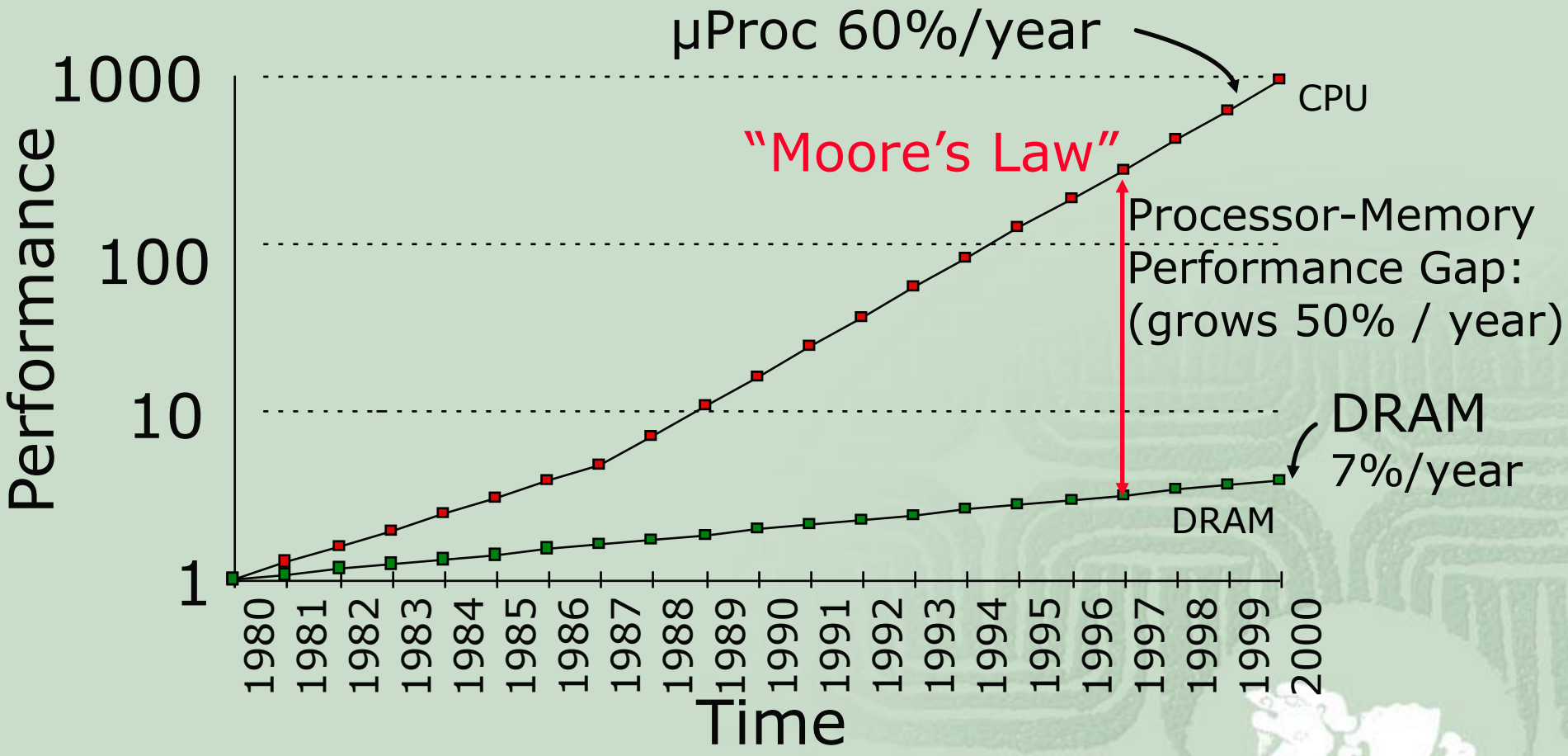
存储系统的多层次结构

cache-主存-辅存三级存储体系：三级存储层次

主存-辅存存储层次：　大容量、低成本；
软硬件结合完成

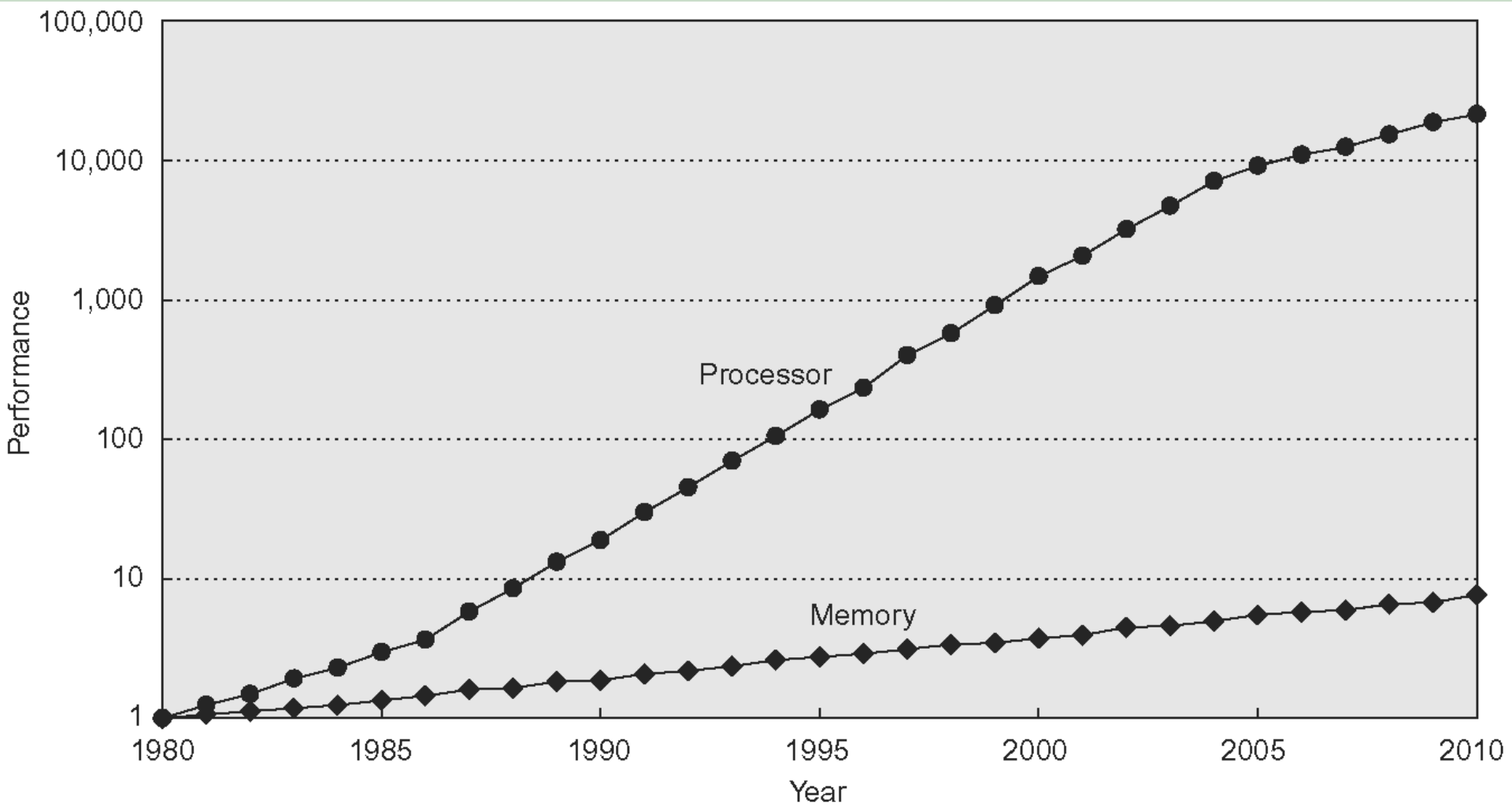
cache-主存存储层次：　高速度、低成本；
纯硬件完成

解决存储系统三个指标：容量、速度、价格/位的矛盾



Four-issue 2GHz superscalar accessing 100ns DRAM could execute 800 instructions during time for one memory access!

处理器与主存的速度差距



处理器与主存的速度差距

7.3.1 cache工作原理

主要依据：程序的局部性原理

两种可预测的主存访问性质（Two predictable properties of memory references）：

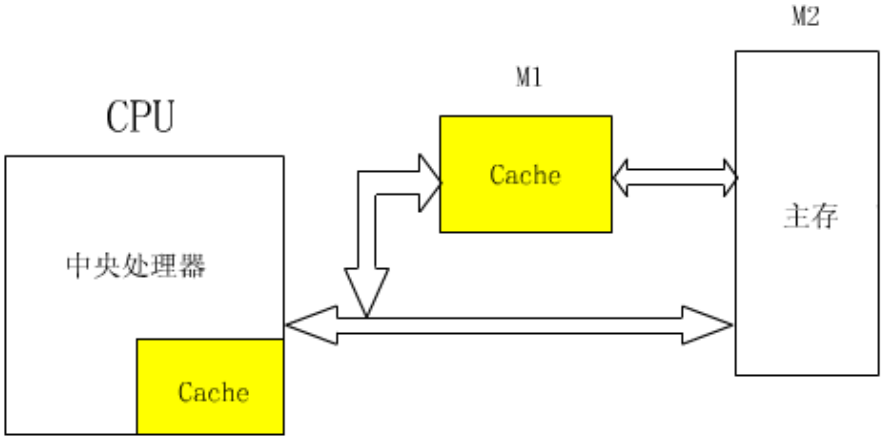
- 时间局部性：某位置若被访问，则近期内很可能又被访问。
  Temporal Locality: If a location is referenced it is likely to be referenced again in the near future.

- 空间局部性：某位置若被访问，则临近的位置也可能被访问。
  Spatial Locality: If a location is referenced it is likely that locations near it will be referenced in the near future.

同样，对数据的访问也会存在局部性现象。

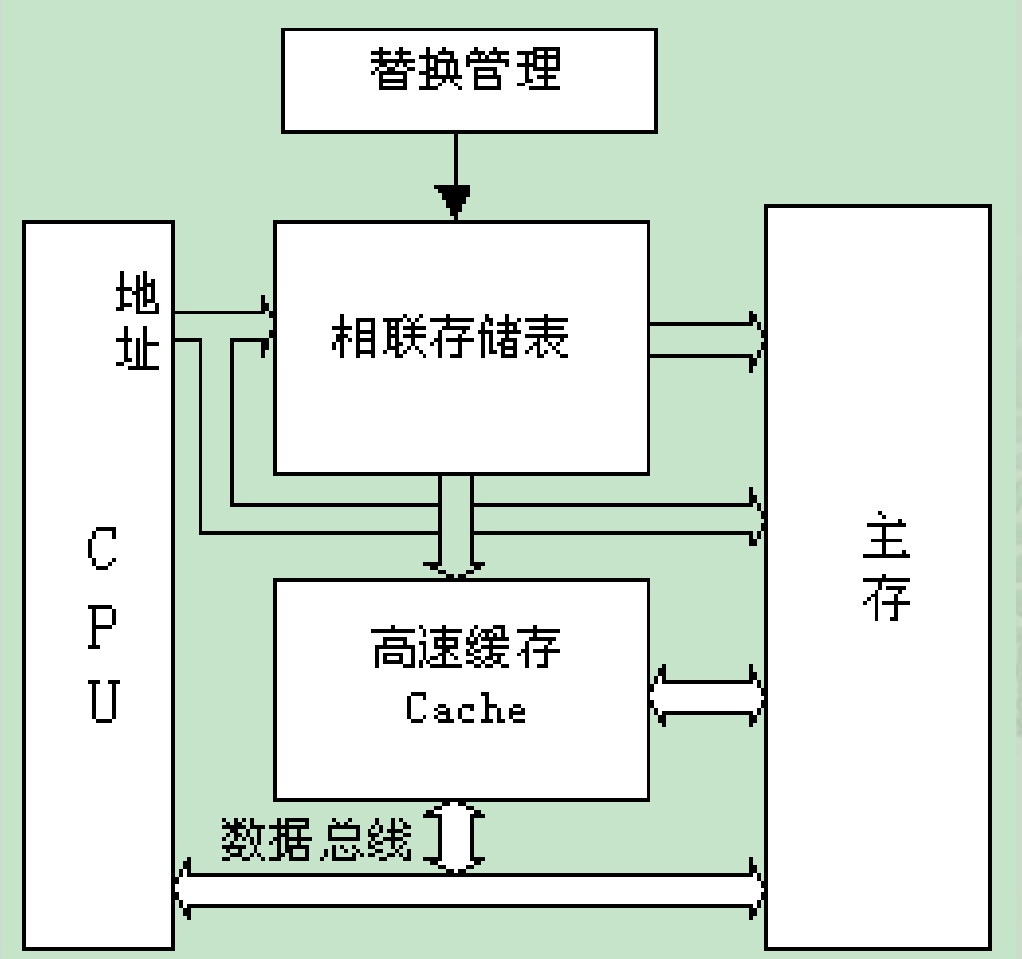
1、概述

（1）cache的位置

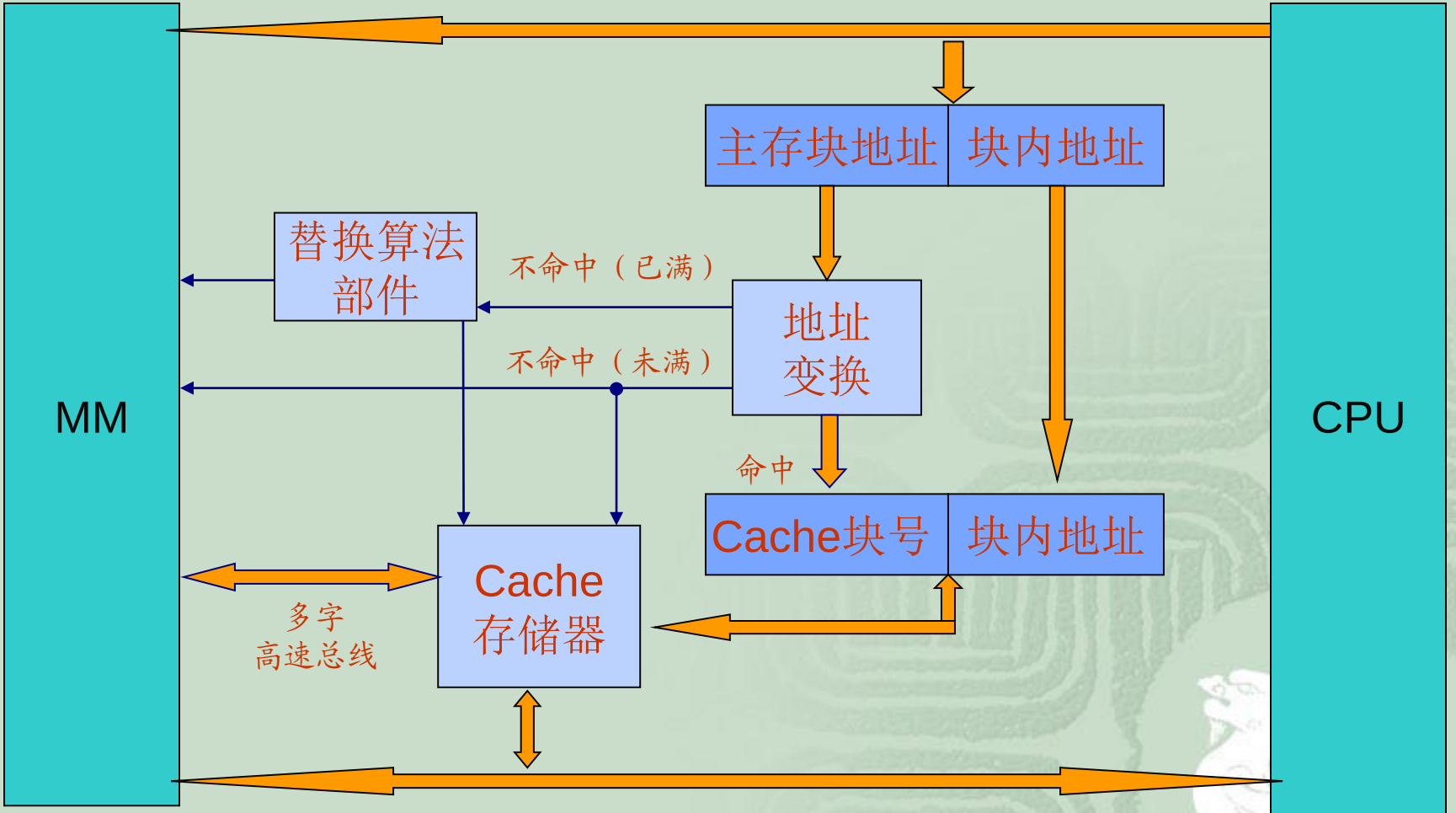


- 基于程序及数据访问的局部性原理，在CPU和主存之间，尽量靠近CPU的地方设置一种容量比较小而速度高的存储器，将当前正在执行的程序和正在访问的数据放在其中。
- 在程序运行时，不需要从慢速的主存中取指令和数据，而是直接访问这种高速小容量的存储器，从而可以提高CPU的程序执行速度，这种存储器就称为高速缓冲存储器（Cache）。

（2）cache的构成



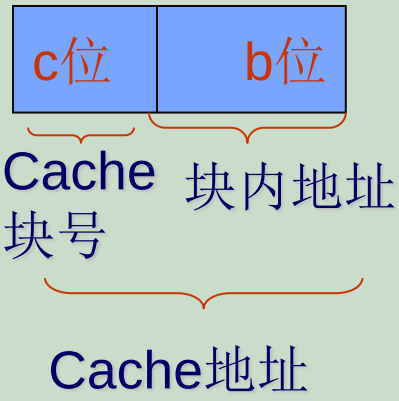
Cache的工作过程：


MM
CPU
主存块地址
块内地址
替换算法
部件
不命中（已满）
地址
变换
不命中（未满）
命中
Cache块号
块内地址
Cache
存储器
多字
高速总线

2、地址映射

- **块**

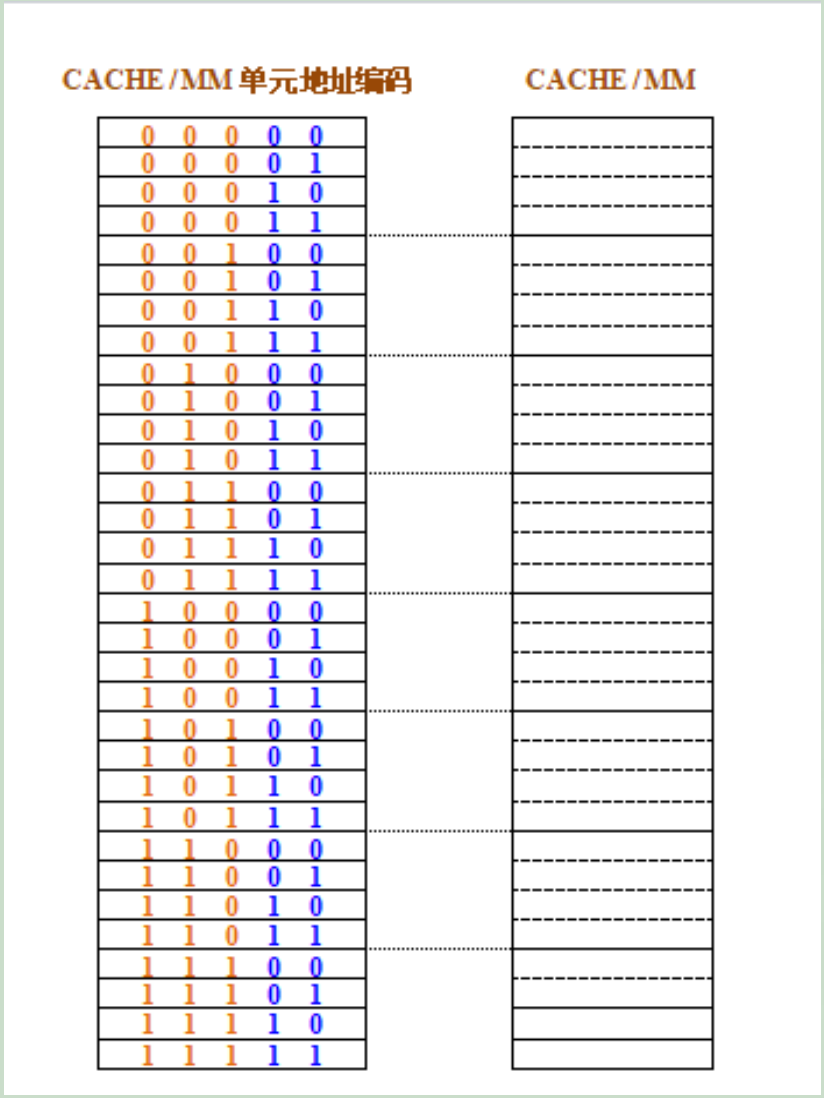
  （块号、块内地址）

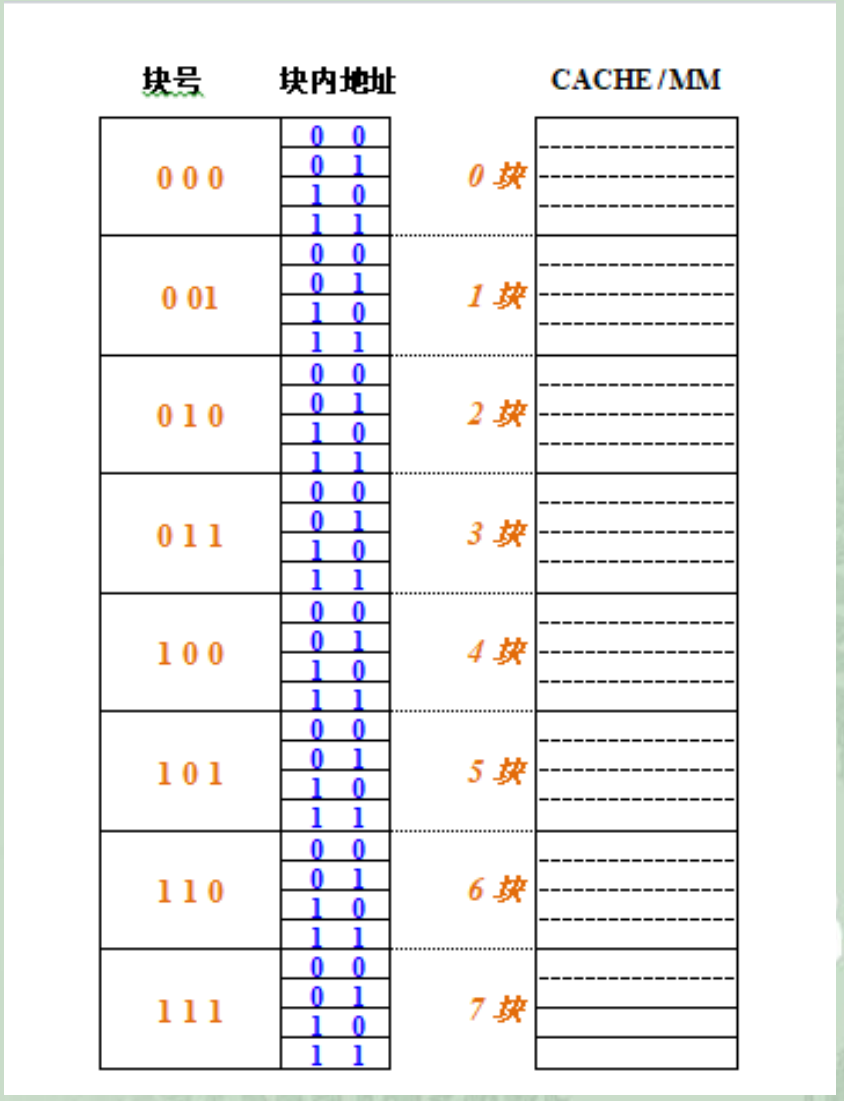


CACHE

$2^c = C$

理解：地址字段划分示意图

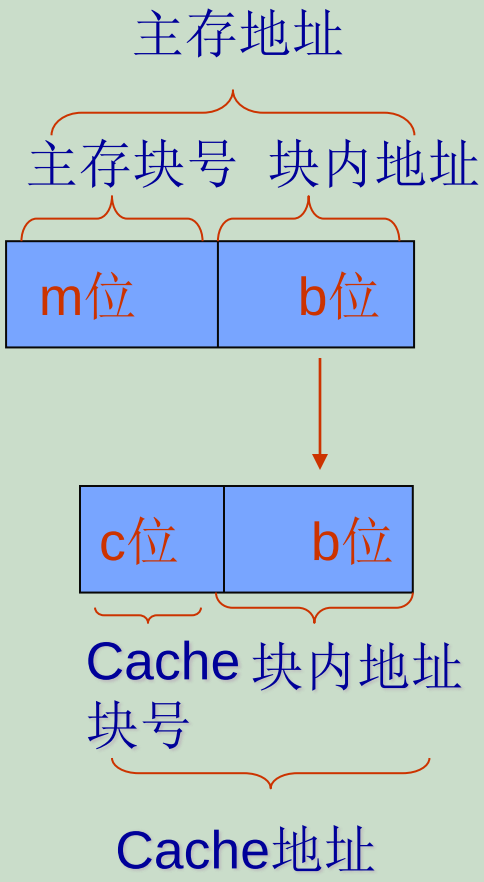




2、地址映射

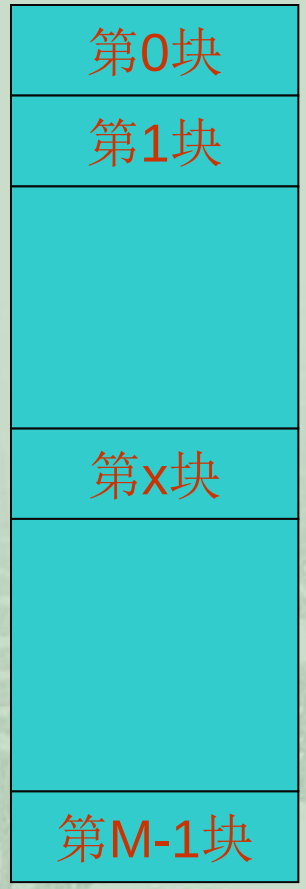
- **块**

  （块号、块内地址）



CACHE

$2^c = C$

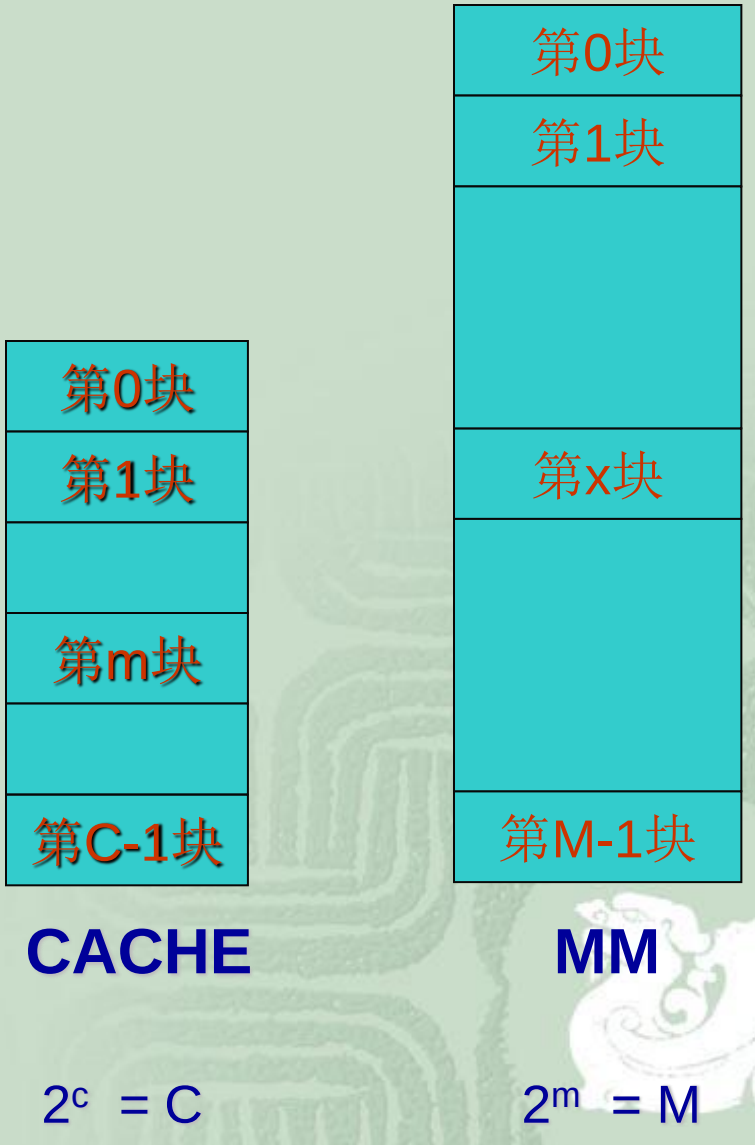


MM

$2^m = M$

2、地址映射

- 块
- 命中/不命中

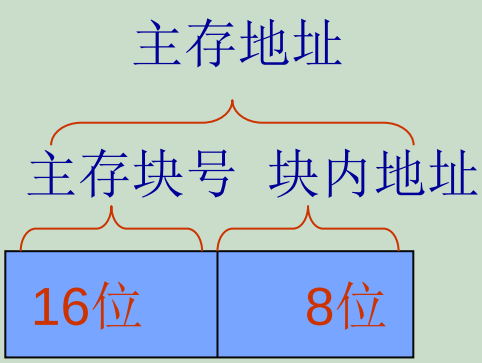


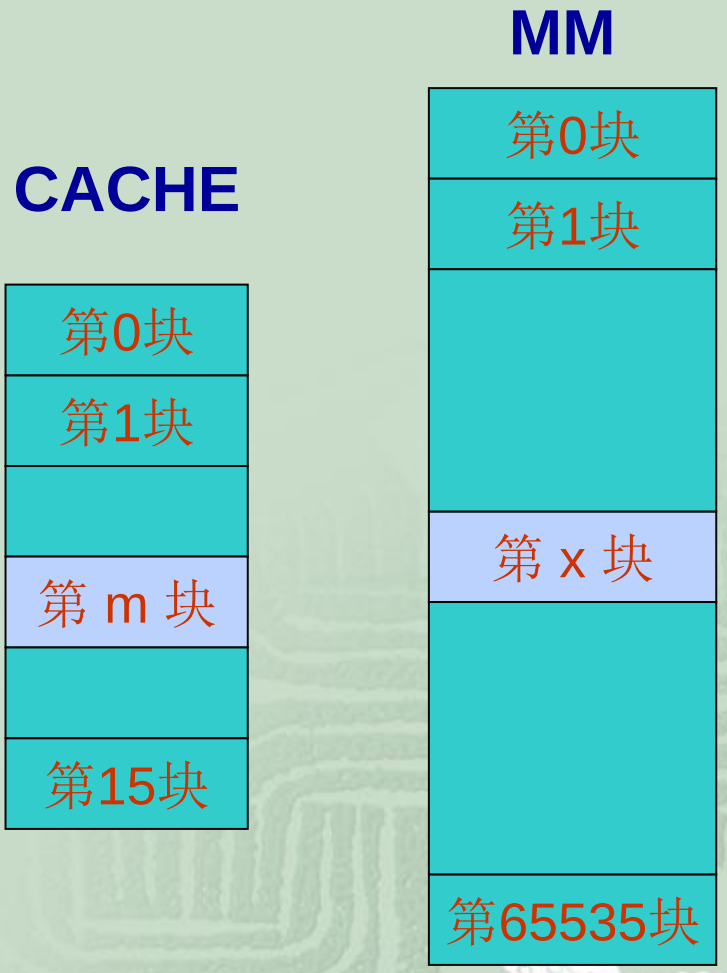
地址映像、地址变换

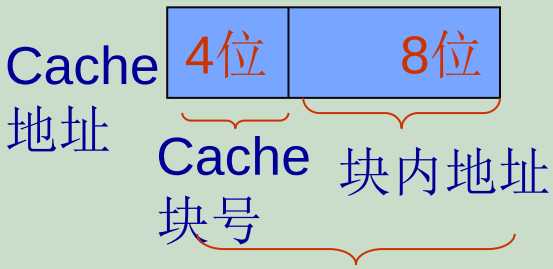
（1）全相连映射方式

（2）直接映射方式

（3）组相连映射方式

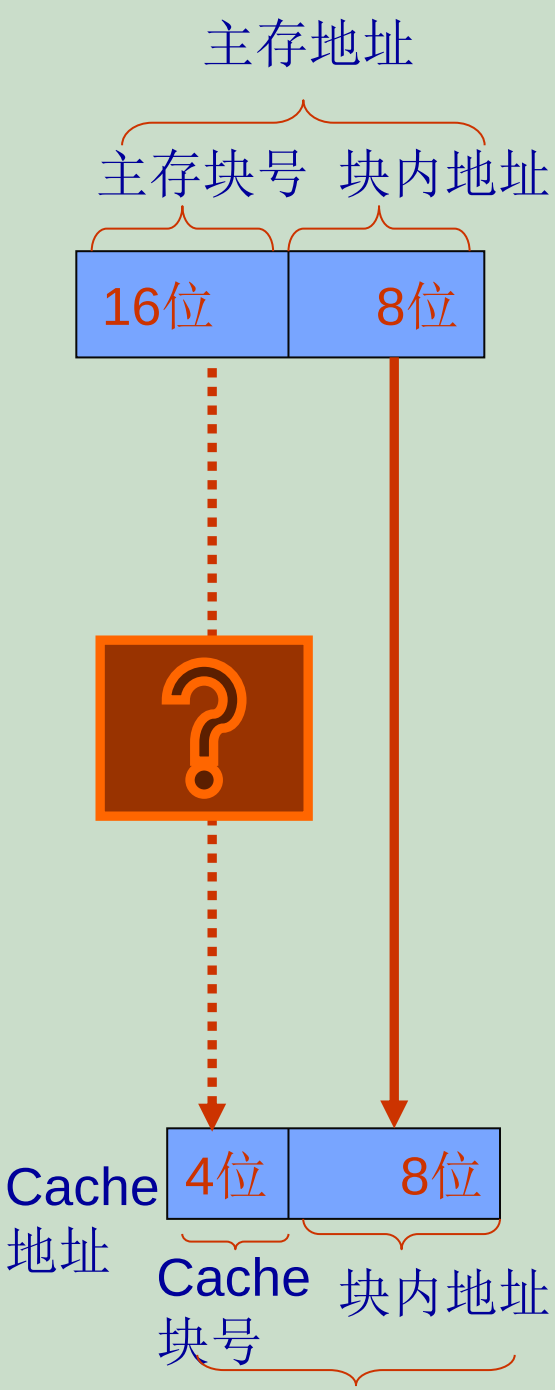


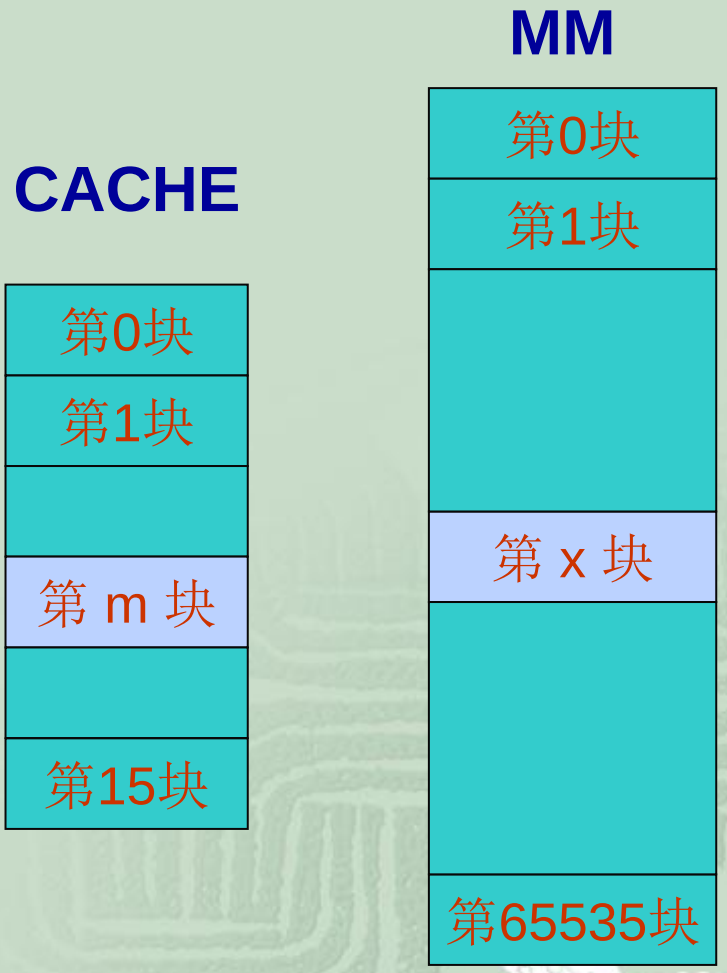




CACHE　共4KB，　每块256B，划分为16块
（块内地址8位，块地址4位）

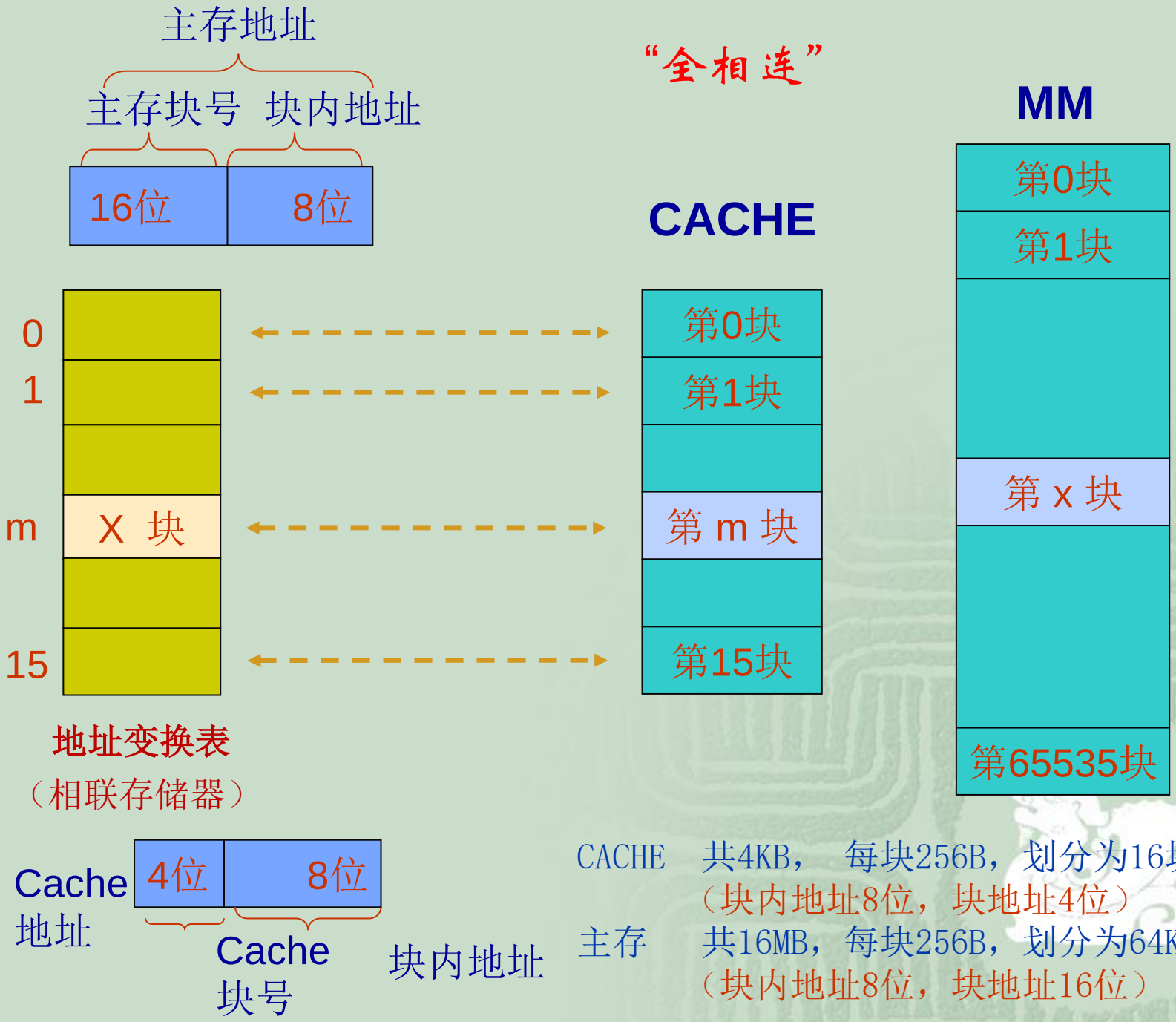
主存　共16MB，每块256B，划分为64K块
（块内地址8位，块地址16位）





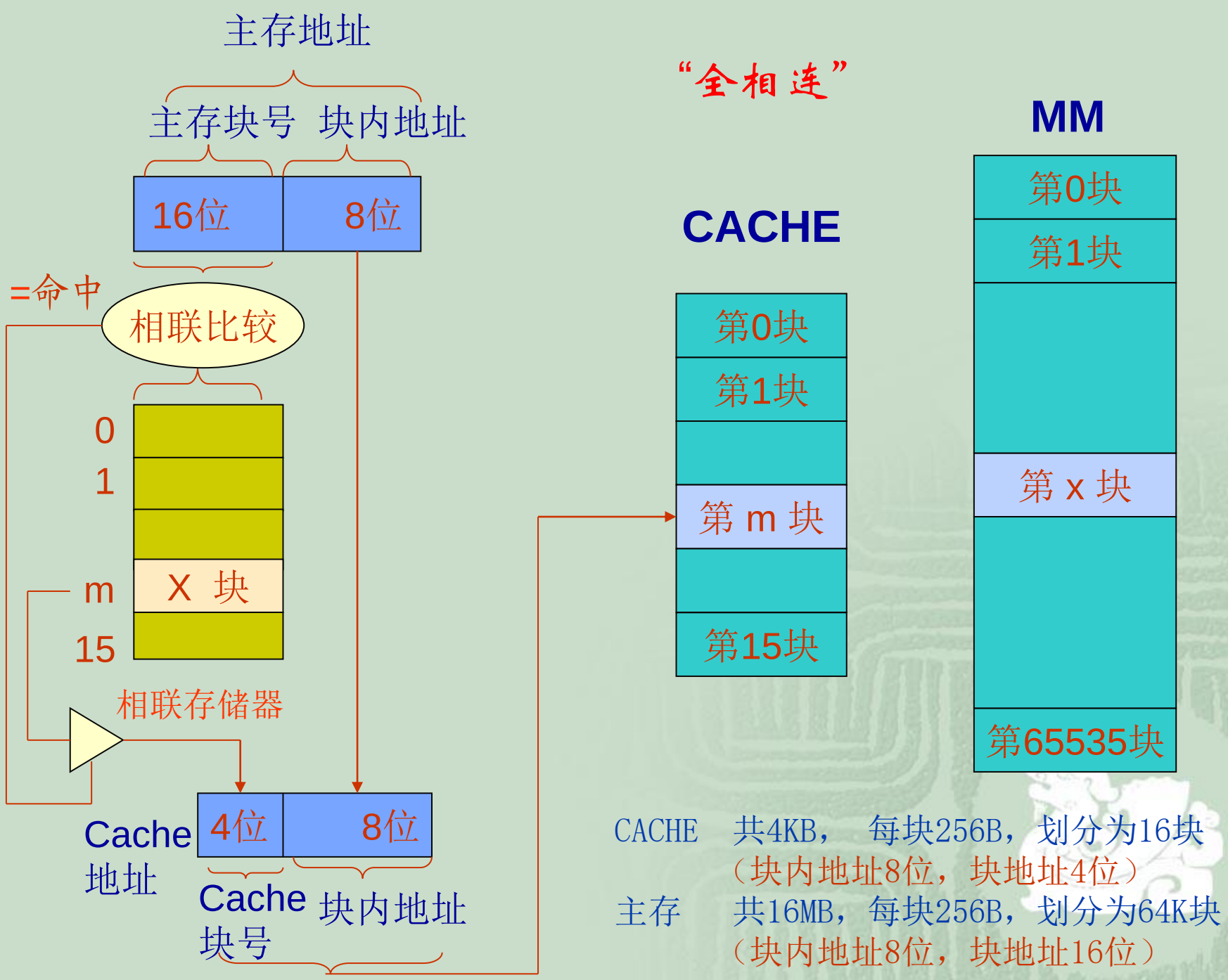
CACHE　共4KB，　每块256B，划分为16块
（块内地址8位，块地址4位）

主存　共16MB，每块256B，划分为64K块
（块内地址8位，块地址16位）



CACHE 共4KB， 每块256B，划分为16块
（块内地址8位，块地址4位）

主存 共16MB，每块256B，划分为64K块
（块内地址8位，块地址16位）



CACHE 共4KB， 每块256B，划分为16块
（块内地址8位，块地址4位）

主存 共16MB，每块256B，划分为64K块
（块内地址8位，块地址16位）

问题：

上面的例子中，cache采用全相连映射方式

问：

进行相联比较的相联存储器的容量？

问题：

Cache与主存间采用全相联地址映像方式， Cache容量为4KB，分为4块每块1KB，主存容量为1MB。

问：（1）Cache与主存地址的各字段如何划分？

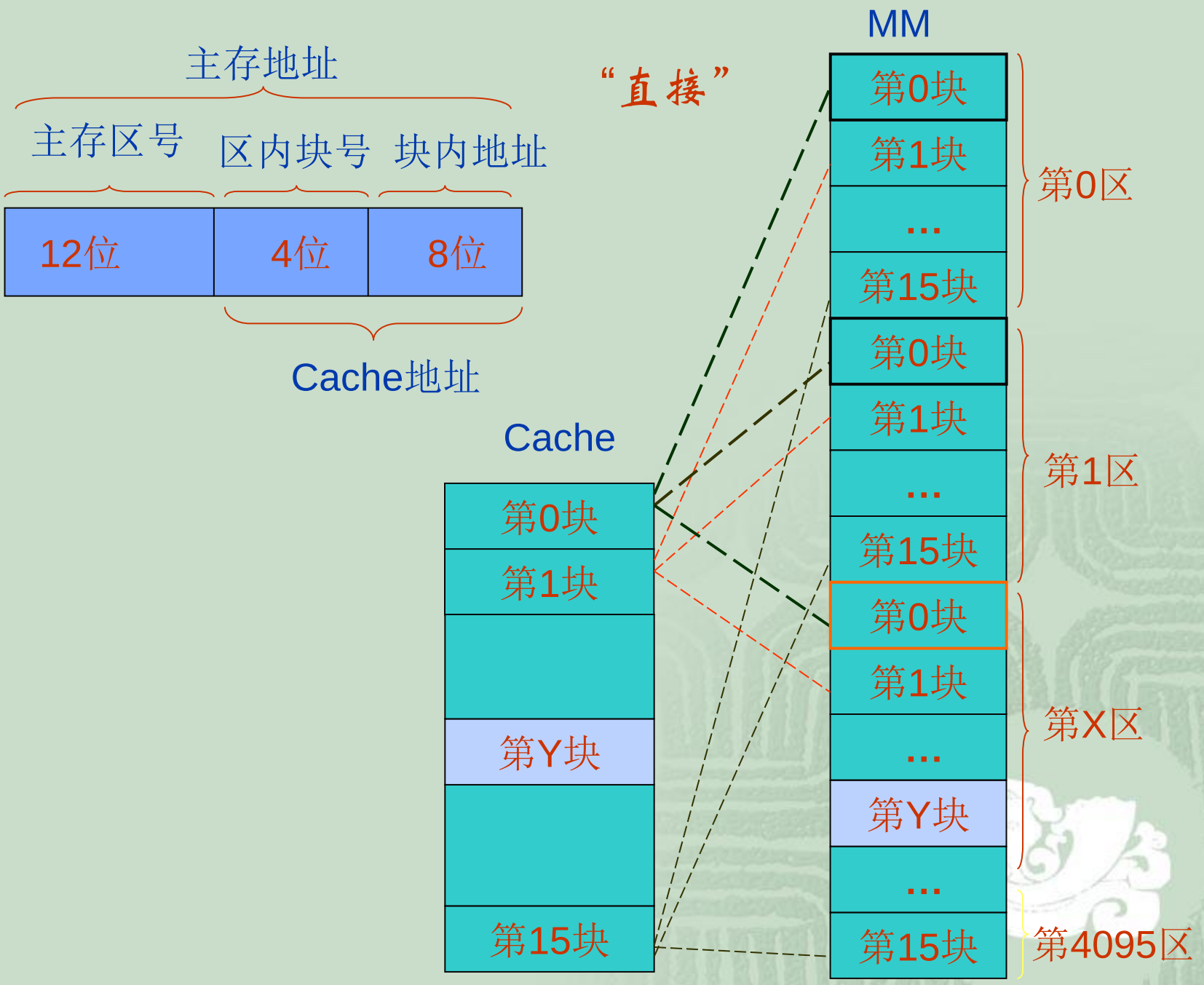
（2）若地址变换表如下，

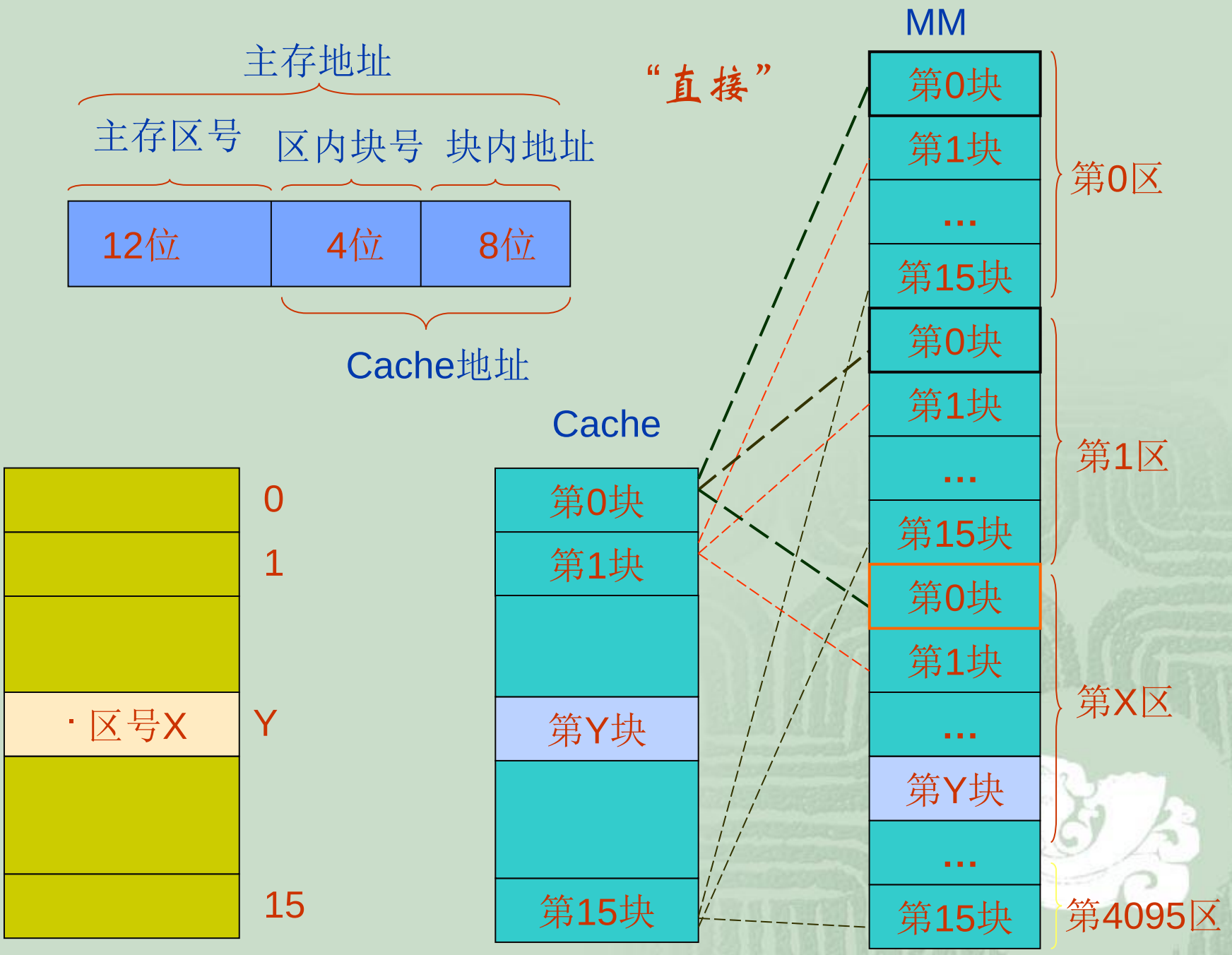
地址变换表

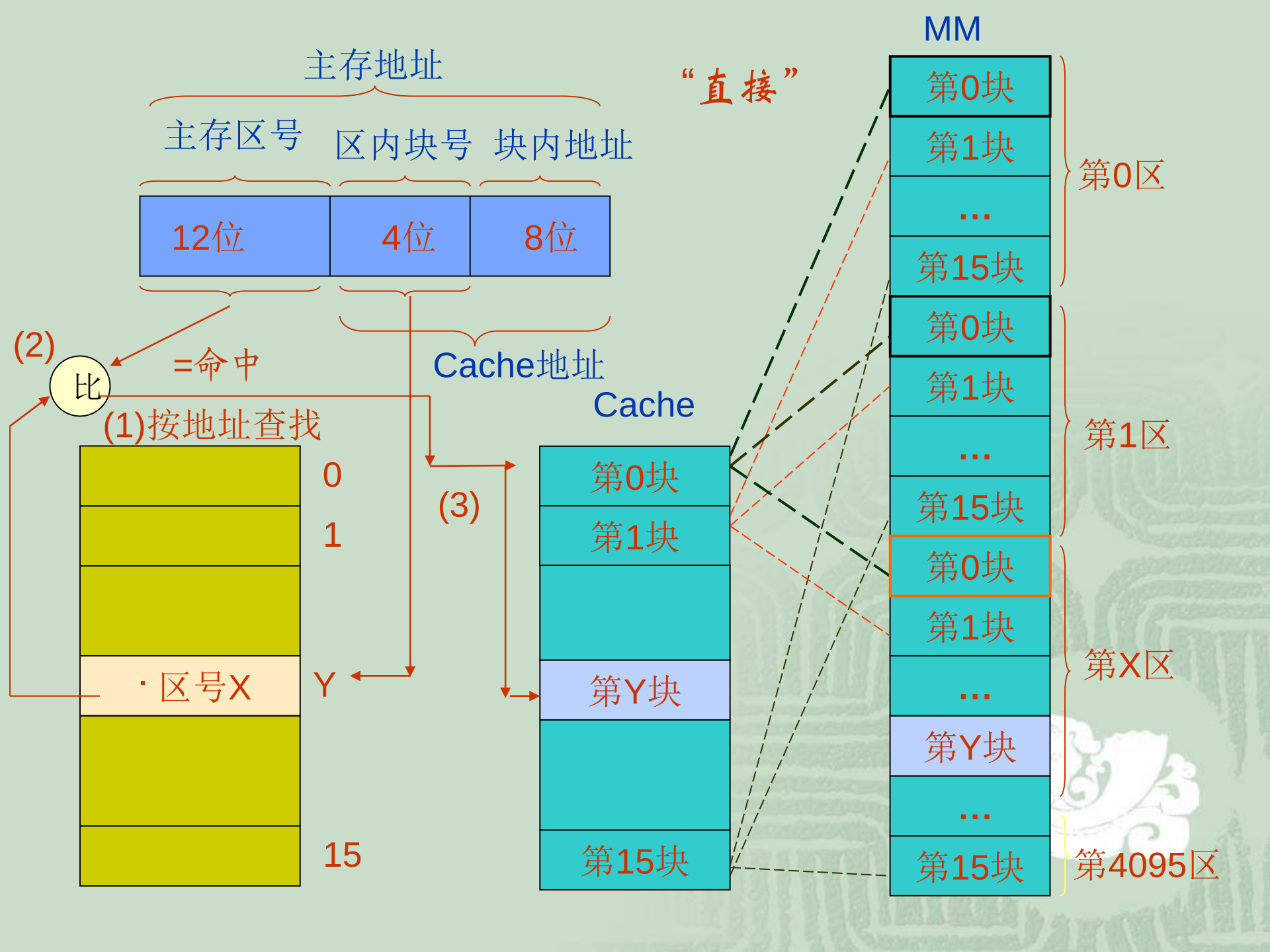
0	367H
1	222H
2	195H
3	388H

试根据主存地址确定变换后的Cache地址。

1. 主存地址为 654E2H 时，高速缓存地址为_____H。
2. 主存地址为 D9D9DH 时，高速缓存地址为_____H。


主存地址
主存区号
区内块号
块内地址
12位
4位
8位
Cache地址
“直接”
MM
第0块
第1块
…
第15块
第0区
第0块
第1块
…
第15块
第1区
第0块
第1块
…
第Y块
…
第15块
第X区
Cache
第0块
第1块
第Y块
第15块
第4095区


主存地址
主存区号
区内块号
块内地址
12位
4位
8位
Cache地址
“直接”
MM
第0块
第1块
…
第15块
第0区
第0块
第1块
…
第15块
第1区
第0块
第1块
…
第15块
第X区
第Y块
…
第15块
第4095区
Cache
第0块
第1块
第Y块
第15块
0
1
区号X
Y
15


主存地址
主存区号
区内块号
块内地址
12位
4位
8位
(2)
比
=命中
Cache地址
(1)按地址查找
0
1
区号X
Y
15
(3)
Cache
第0块
第1块
第Y块
第15块
“直接”
MM
第0块
第1块
...
第15块
第0区
第0块
第1块
...
第15块
第1区
第0块
第1块
...
第Y块
...
第15块
第X区
第4095区

问题：

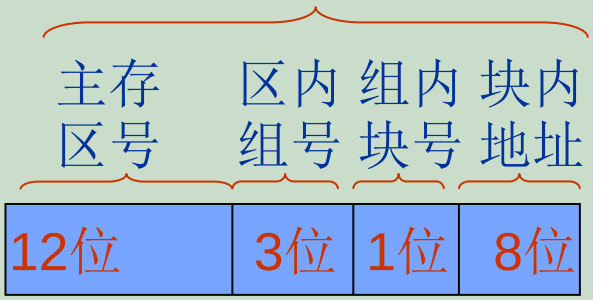
一个具有16KB直接相连映射cache的32位微处理器，cache的块为4个字（字长32位），内存为256MB

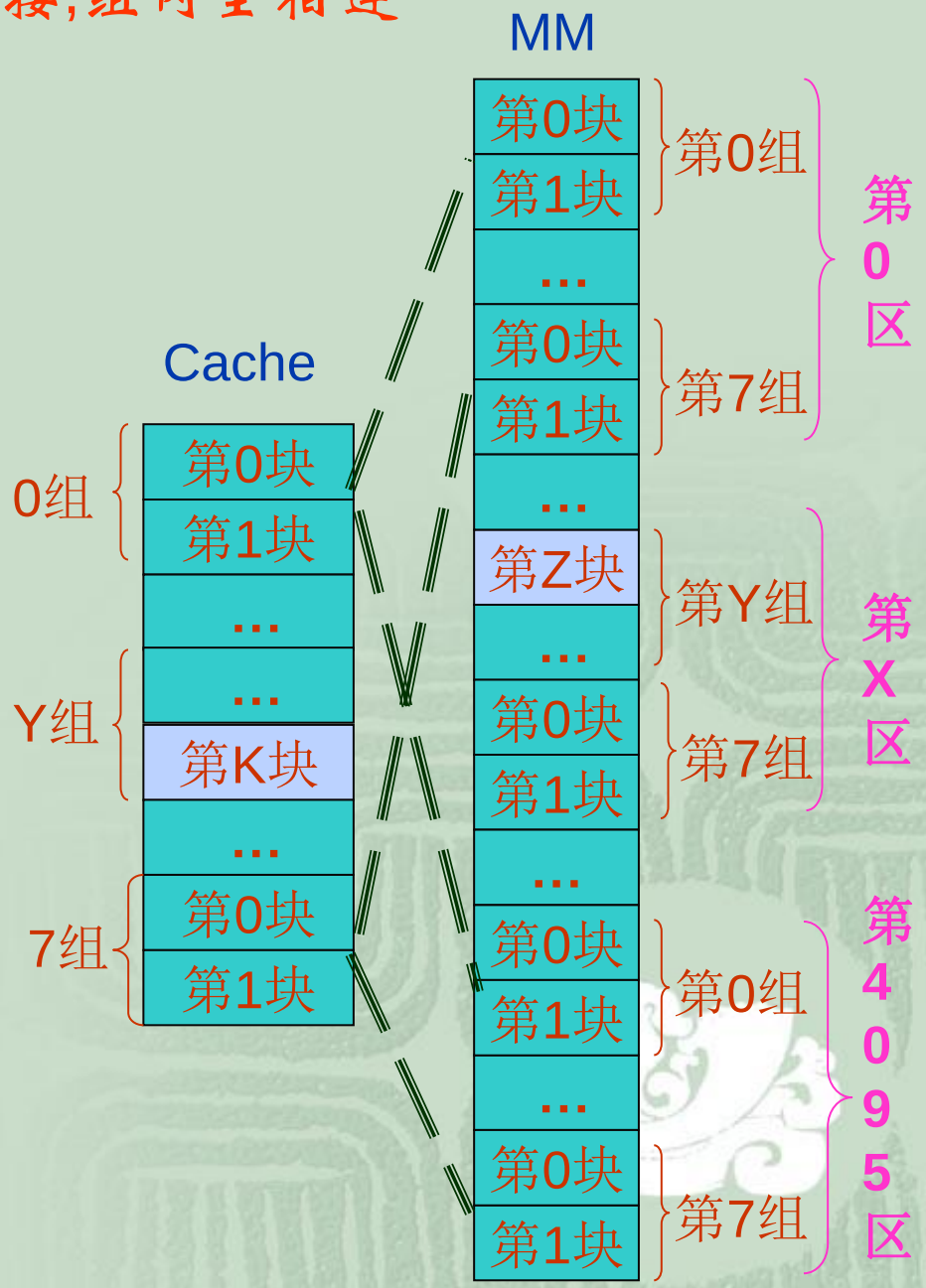
问：（1）主存地址为多少位（按字节编址），各字段如何划分？

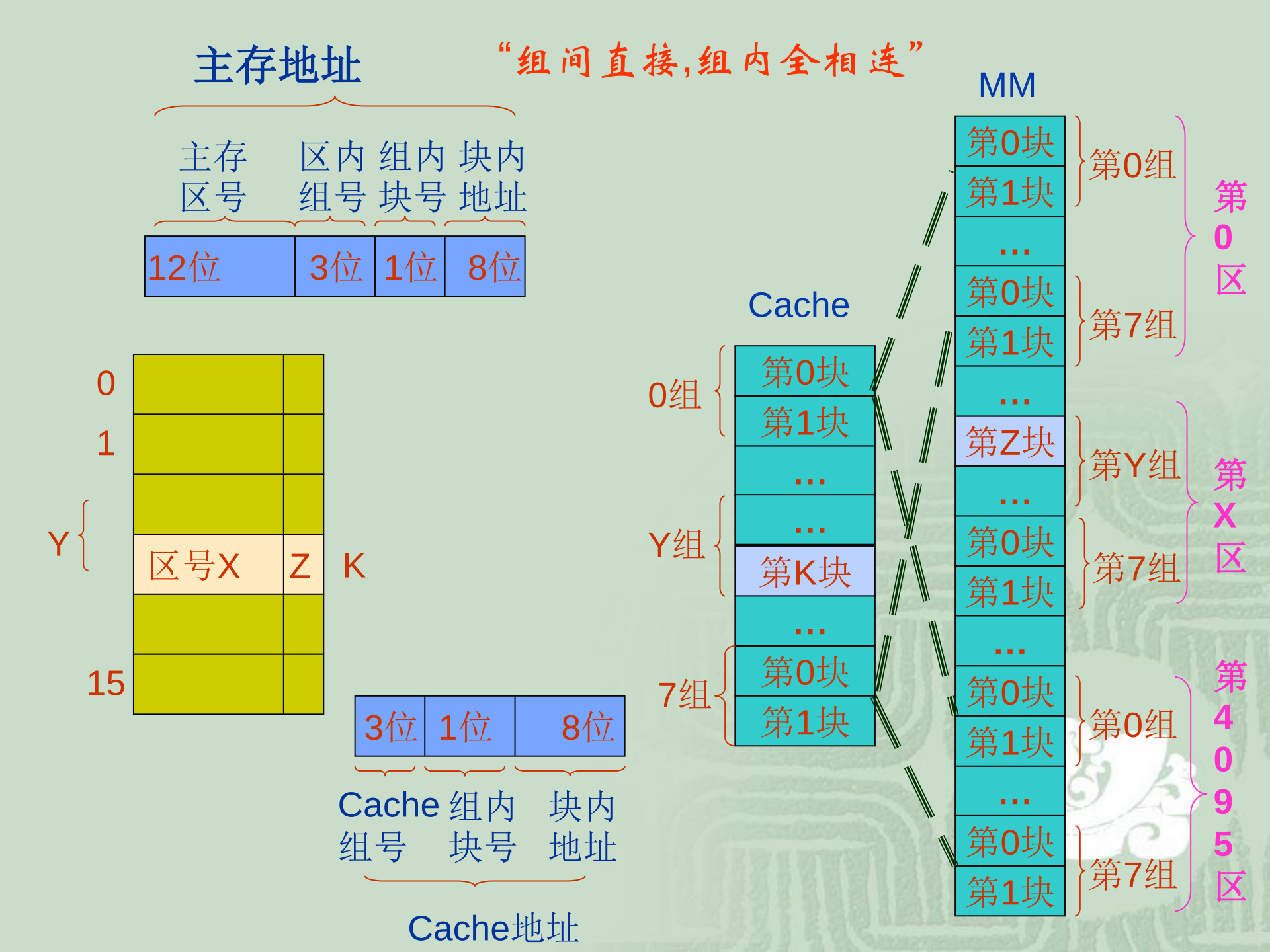
（2）主存地址为ABCDEF8H的单元调入cache中的位置？

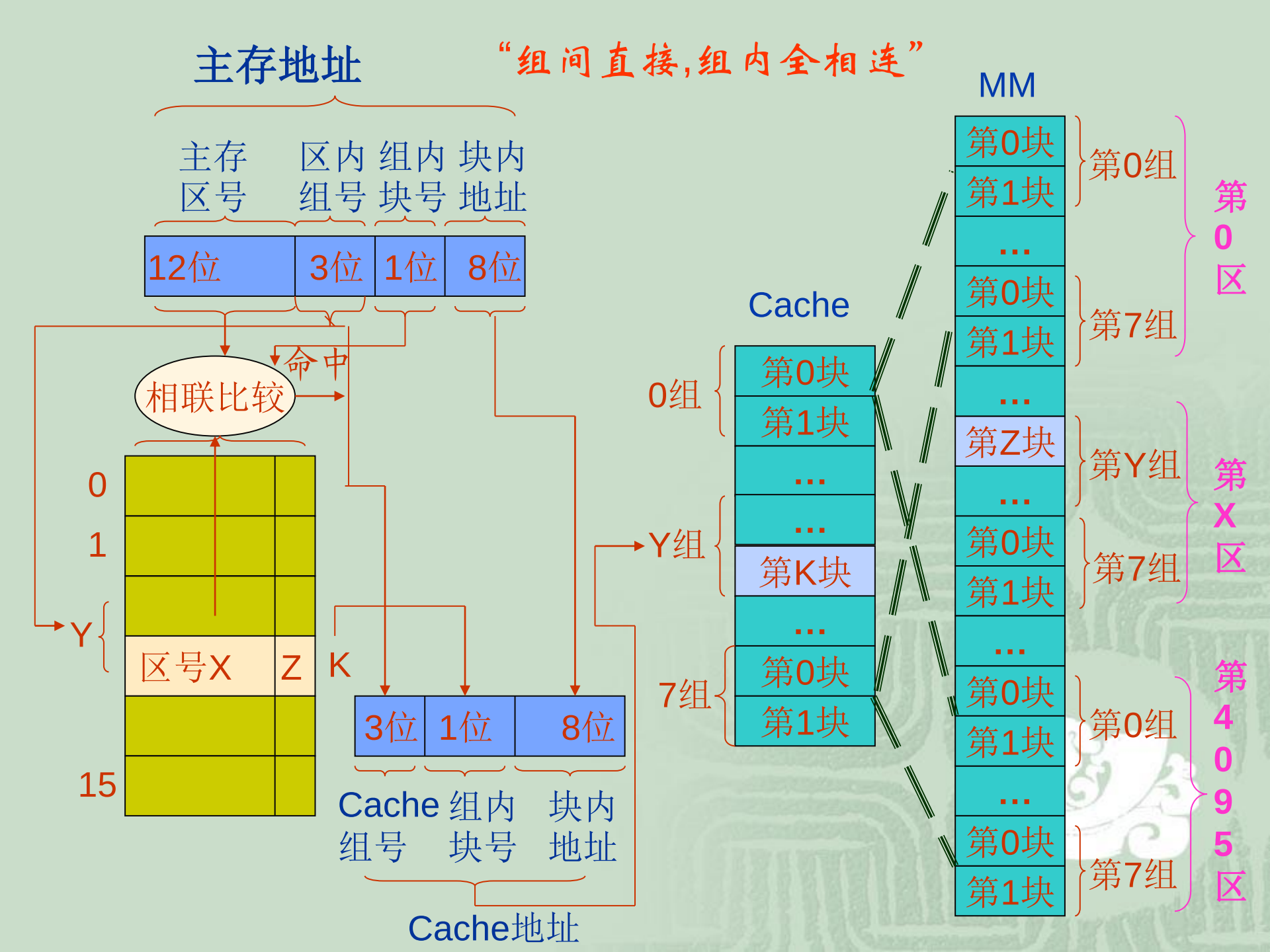
“组间直接,组内全相连”

主存地址






主存地址
“组间直接,组内全相连”
主存区号
区内组号
组内块号
块内地址
12位
3位
1位
8位
MM
Cache
0
1
Y
区号X
Z
K
15
3位
1位
8位
Cache组号
组内块号
块内地址
Cache地址
0组
Y组
7组
第0块
第1块
第K块
第Z块
第0组
第7组
第Y组
第0区
第X区
第4095区


主存地址
“组间直接,组内全相连”
主存区号
区内组号
组内块号
块内地址
12位
3位
1位
8位
命中
相联比较
0
1
Y
15
区号X
Z
K
3位
1位
8位
Cache 组内块号
Cache 块内地址
Cache地址
Cache
0组
第0块
第1块
...
Y组
第K块
7组
MM
第0组
第7组
第Y组
第0块
第1块
第Z块
第0块
第1块
第0组
第7组
第0块
第1块
第0块
第1块
第0块
第1块
第0块
第1块
第0区
第X区
第4095区

问题：

上面的例子中，cache采用组相连映射方式

问：

其中相联存储器的容量？

进行相联比较的存储单元的容量？

问题：

高速缓存Cache与主存间采用组相联地址映像方式（组间直接，组内全相联），高速缓存每组包含4块，每块为8个字，每个字为32位。若主存容量为2MB，Cache的容量为16KB。

问：

（1）请分析cache地址有多少位，各字段如何划分？

（2）请分析主存地址有多少位，试说明主存区号、区内组号、组内块号、块内字号、字内地址号各用多少位表示？

问题： Cache-MM两级存储器采用组相联映像（组间直接，组内全相联）。若Cache容量为512B，64个字节为一块，且共分为2个组。主存容量是Cache容量2048倍。

(1) 主存区号___位，区内组号___位，组内块号___位，块内地址___位。每次进行MM→Cache的地址变换时，需要参与相联比较的位数是____位。

(2) 若Cache-MM地址变换表的内容如下表，当CPU访问主存的地址分别为91118H和0EDCBAH时，问是否能命中Cache，若能命中，指出相应的Cache地址。

	主存区号	组内块号
000	0C9H	00B
001	574H	01B
010	244H	10B
011	76EH	11B
100	76EH	10B
101	373H	10B
110	0C9H	00B
111	488H	00B

问题：

（3）若主存以Cache的64个字节为一块，从0块开始顺序分块并编号，试决定515块应放在Cache的哪一组中？

参考答案：

(1) 区号为11位；组号1位；块号2位；块内地址6位；
相联比较13位。

(2) 主存地址91118H命中，其Cache地址为1D8H；
主存地址0EDCBAH未命中。

(3) 515块应放在Cache的0组中。

总结：地址映像、地址变换

（1）全相连映射方式

规则：主存中的任意块均可以装入Cache中的任意一个块。

特点：主存块装入位置无限制，仅当Cache全部装满后才会出现冲突，即块冲突概率最低，但是相联存储器的容量最大，变换机构复杂。

（2）直接映射方式

规则：主存中的任意块只能装入Cache中的固定一个块。

特点：主存块装入位置限制，主存不同区的相同块将无法同时出现在Cache中，块冲突概率最高，但是相联存储器的容量最小，变换简单，命中时可直接由主存地址中提取到Cache地址。

（3）N路-组相连映射方式

规则：组间直接、组内全相联

特点:是前两种方式的折中，即块冲突概率和变换复杂度均处于中间位置。

总结：地址映像、地址变换

地址变换表(相联存储器)还在每一个存储单元设定：

1. **设置1位有效位**

   规定该位为1时，该块有效；

   该位为0时，该块无效。

2. **在数据Cache的相联存储器中，对应着Cache的一块设置1位修改位**

   当在该块在使用中数据被修改时，用该位标志。

3. **为替换方便，可设置计数器。**

7.3.2 替换算法

若 Cache未命中，而 Cache已装满，则需替换Cache中的块。

- **随机替换算法（RAND）**

  这种算法是用随机函数发生器产生需替换的块号，将其替换。

  这种方法没有考虑信息的历史及使用情况，故其命中率比较低。目前已不再使用。

- **先进先出算法（FIFO）**

  该算法是将最先装入Cache的那一块替换出去。

  这种方法只考虑信息的历史情况而没有考虑其使用情况，也许最先装入的那一块正在频繁使用。因此，该算法也有一定局限性，命中率也不是很高。

- **例：** 假定程序在主存为5块，Cache为3块。CPU执行程序的顺序为：P2、P3、P2、P1、P5、P2、P4、P5、P3、P2、P5、P2。画出FIFO算法命中情况如图所示。

P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2
2	2	2	2	5	5	5	5	3	3	3	3
	3	3	3	3	2	2	2	2	2	5	5
			1	1	1	4	4	4	4	4	2
		H					H		H		

图中用 H 表示“命中”，

在利用FIFO算法的情况下，命中率为 3/12=25%

7.3.2 替换算法

- 近期最少使用算法（LRU）

  对每块设置一个计数器，某块每命中一次，就将其计数器清0而其他块的计数器加1，记录Cache中各块的使用情况。

  当需要替换时，便将计数值最大的块替换出去。

  由于Cache的工作是建立在程序执行及数据访问的局部性原理。因此，该算法较前两种算法的命中率要高一些。

- 最不经常使用算法（LFU）

  对每块设置一个计数器，且开始调入时计数为0。每被访问一次，被访问块的计数器加1。

  当需要替换时，便将计数值最小的块替换出去，同时将所有各块的计数器清0。

  将计数周期限定在两次替换的时间间隔内。不能完全反映近期的访问情况。

- **例：** 假定程序在主存为5块，Cache为3块。CPU执行程序的顺序为：P2、P3、P2、P1、P5、P2、P4、P5、P3、P2、P5、P2。画出LRU算法命中情况如图所示。

P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2
2	2	2	2	2	2	2	2	3	3	3	3
	3	3	3	5	5	5	5	5	5	5	5
			1	1	1	4	4	4	2	2	2
		H			H		H			H	H

图中用 H 表示“命中”，

在利用LRU算法的情况下，命中率为 5/12=42%

7.3.2 替换算法

- 最优替换算法（OTP）

  要实现这种算法程序需执行两次。

  执行第一遍时，记录各块地址的使用情况。根据第一遍的记录就能找出需要替换出去的该是哪块。有了先验的替换信息，在第二次执行时一定能将命中率达到最高。

  非实用算法，仅作为算法评价的基准。

7.3.3 主存—cache内容的一致性

指令 Cache 不存在一致性问题

- **写回法（Write Back）**
  - ✓ 当CPU写Cache命中时，只将数据写入Cache而不立即写入主存。只有当被CPU写入修改的块被替换出去时才写回到主存中。

    这种方法减少了访问主存的次数,但是存在不一致性的隐患。
  - ✓如果CPU写Cache未命中，则写修改是将相应主存块调入Cache之后，在Cache中进行。对主存的修改仍留待该块替换出去时进行。

    实现这种方法时，每个cache行必须配置一个修改位，以反映此行是否被CPU修改过。

- **全写法（写直达法 Write Through）**

  ✓当CPU写Cache命中时，在将数据写入修改Cache的同时写入修改主存，较好地保证了主存与Cache内容的一致性。

  当写cache命中时，cache与主存同时发生写修改，因而较好地维护了cache与主存的内容的一致性。

  Cache中每块无需设置一个修改位以及相应的判断逻辑。缺点是Cache对CPU向主存的写操作无高速缓冲功能，降低了Cache的功效。

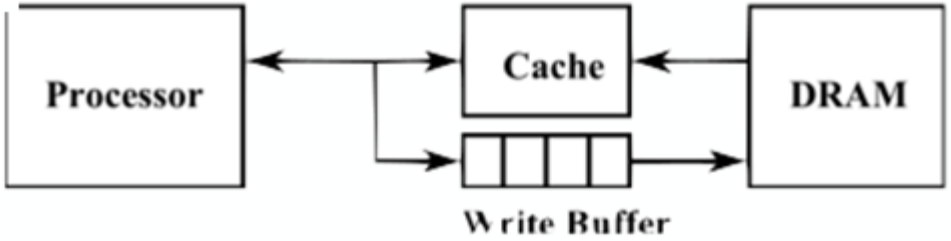
  当写cache未命中时，直接向主存进行写入。这时可以采用两种方式进行处理:

  WTWA: （Write Through With Write Allocate）取主存块到Cache并为其分配一个行位置;

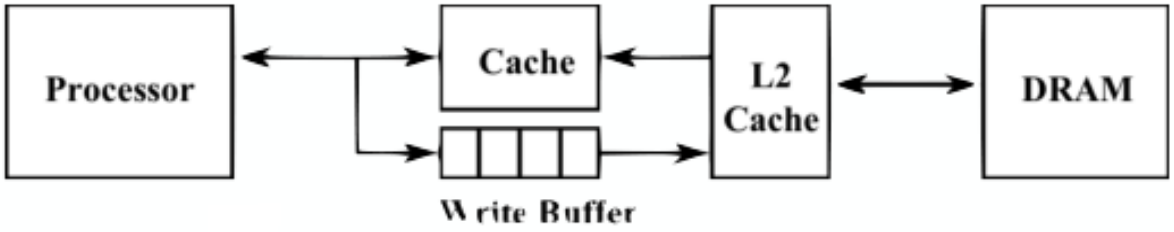
  WTNWA: （Write Through With No Write Allocate）不取主存块到Cache。

全写法（写直达法 Write Through）

- Cache 和内存速度不一致如何解决?
  - 速度不够：Buffer



- 谁把Buffer数据搬到内存?
  - 加个控制器让内存主动取回来...
- 加2级Cache!



7.3.4 cache的性能分析

- Cache的命中率 (Hit Rate)

  在一个程序执行期间，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，H定义为命中率，则有

$$H = \frac{N_C}{N_C + N_m}$$

- 非命中率(Miss Rate)

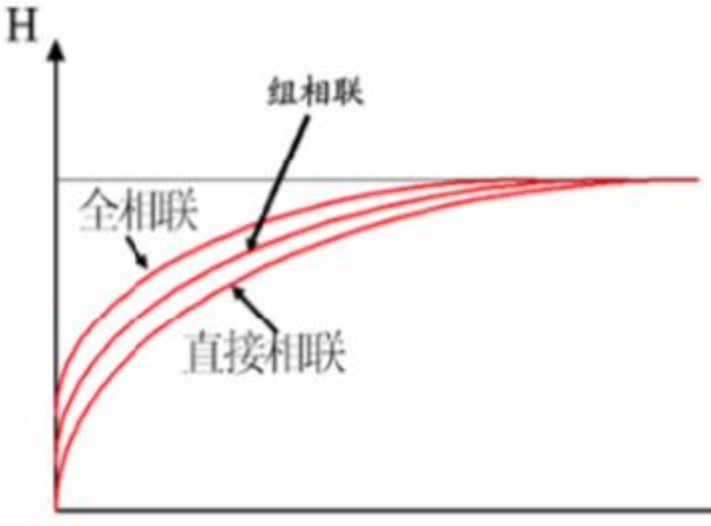
  Cache非命中(Miss)

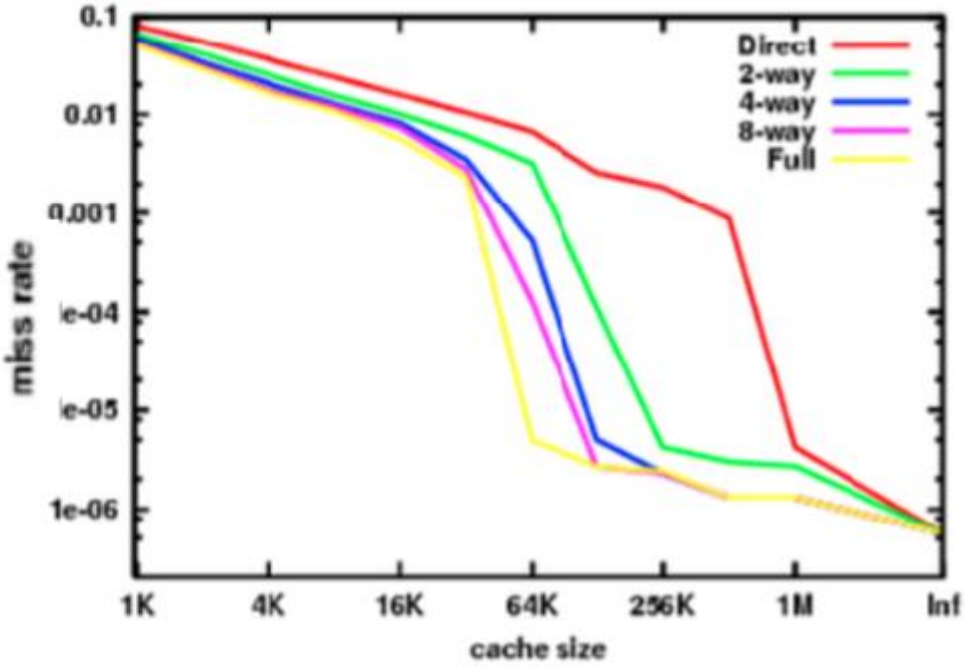
  1 - H

映射方式比较：

全相联最优；

组相联方式4路以上效果减弱





- 加速比

假设Cache的访问周期为T_C，主存的访问周期为T_M，数据块调入Cache的块传输时间（或称块访问时间）为T_B，Cache的命中率为H，则Cache系统的平均访问时间T(average access time)可如下表示：

$$T = H \times T_C + (1 - H) \times T_M$$

或 $$= H \times T_C + (1 - H) \times (T_B + T_C) = T_C + (1 - H) \times T_B$$

或 $$= H \times T_C + (1 - H) \times (T_M + T_C) = T_C + (1 - H) \times T_M$$

- $H \times T_C$: time to directly access cache
- $(1 - H) \times (T_M + T_C)$ in time of miss, time to load the data into cache, and access it. $T_M >> T_C$

Cache系统的加速比S_P定义为：

$$S_P = T_M / T$$

访存时间的有关说明：与Cache的访问和控制机制有关

例如，下表列出了针对采用不同写策略的Cache，在不同操作状况下产生的对存储器的访问时间。

采用Cache的访存操作

Cache类型	操作	访存操作	访存时间
写直达Cache	读命中	只读Cache	T_C
	写命中	写Cache，同时写内存	T_C（隐藏T_M）
	读不命中	调入Cache块，再读Cache	$T_B + T_C$
	写不命中	只写内存	T_M
写回Cache	读命中	只读Cache	T_C
	写命中	只写Cache	T_C
	读不命中	调入Cache块，再读Cache	$T_B + T_C$
	写不命中	调入Cache块，再写Cache	$T_B + T_C$

例如，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=*t+0.05*5t=1.25t*

性能提升=*5t/1.25t=4*倍

例1 假设我们能够轻松地创建一个带有4ns访问时间的片上静态存储器，但是，我们能够为主存储器买到的最快的动态存储器的平均访问时间是40ns。如果我们必须保持5ns的平均访问时间，问需要多高的命中率？

解：

$$h = 1 - \frac{t_{avg} - t_c}{t_m} = 1 - \frac{5-4}{40} = 97.5\%$$

- 成本

  假设计算机中的主存与Cache的容量分别为S1和S2。显然，S1＞＞S2。同时，若主存与Cache的单位价格分别为C1和C2，而且C1一定是低的。则存储器的平均价格C由下式决定：

  $$C=(C1\times S1+C2\times S2)/(S1+S2)$$

Cache系统速度与价格分析：

存储系统成本

- $C = (C1 \times S1 + C2 \times S2)/(S1 + S2)$

S1>>S2

- 尽管Cache的价格比主存高，但是当其容量很小时，存储器的平均价格C接近于主存的价格。
- 由于设置了Cache使CPU的访存速度接近Cache的速度，而使存储器的成本接近于主存的价格。

Cache性能分析与优化

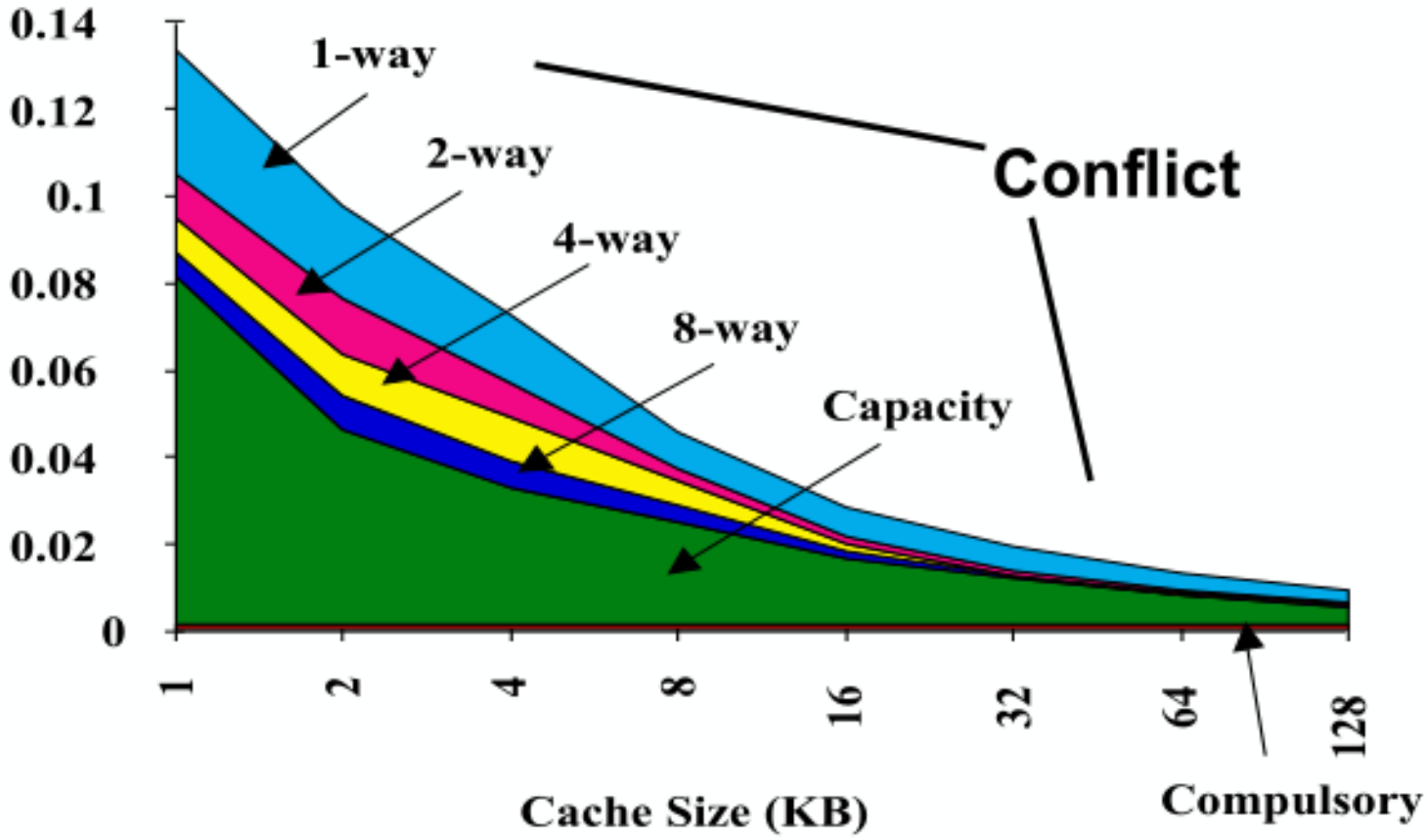
- Cache命中(Hit)
  - CPU欲访问的数据已在缓存中，即可直接访问 Cache
- Cache非命中(Miss)
  - 数据不在Cache，需要将主存数据调入Cache
  - CPU被阻塞(Blocking)，等！
- 命中率(Hit Rate)
- 非命中率(Miss Rate)

Cache性能分析：Miss的4C类型

- Miss的4种类型(4C):
  - Compulsory（必然）
    - 冷启动、过程转移、首次引用等不可避免的miss…
  - Capacity（容量）
    - 容量不够造成的
  - Conflict（冲突）
    - Cache块冲突
      - 方案1：增加容量
      - 方案2：提升相联度
  - Coherence（一致性）
    - 其它操作带来的结构相关（I/O操作等）

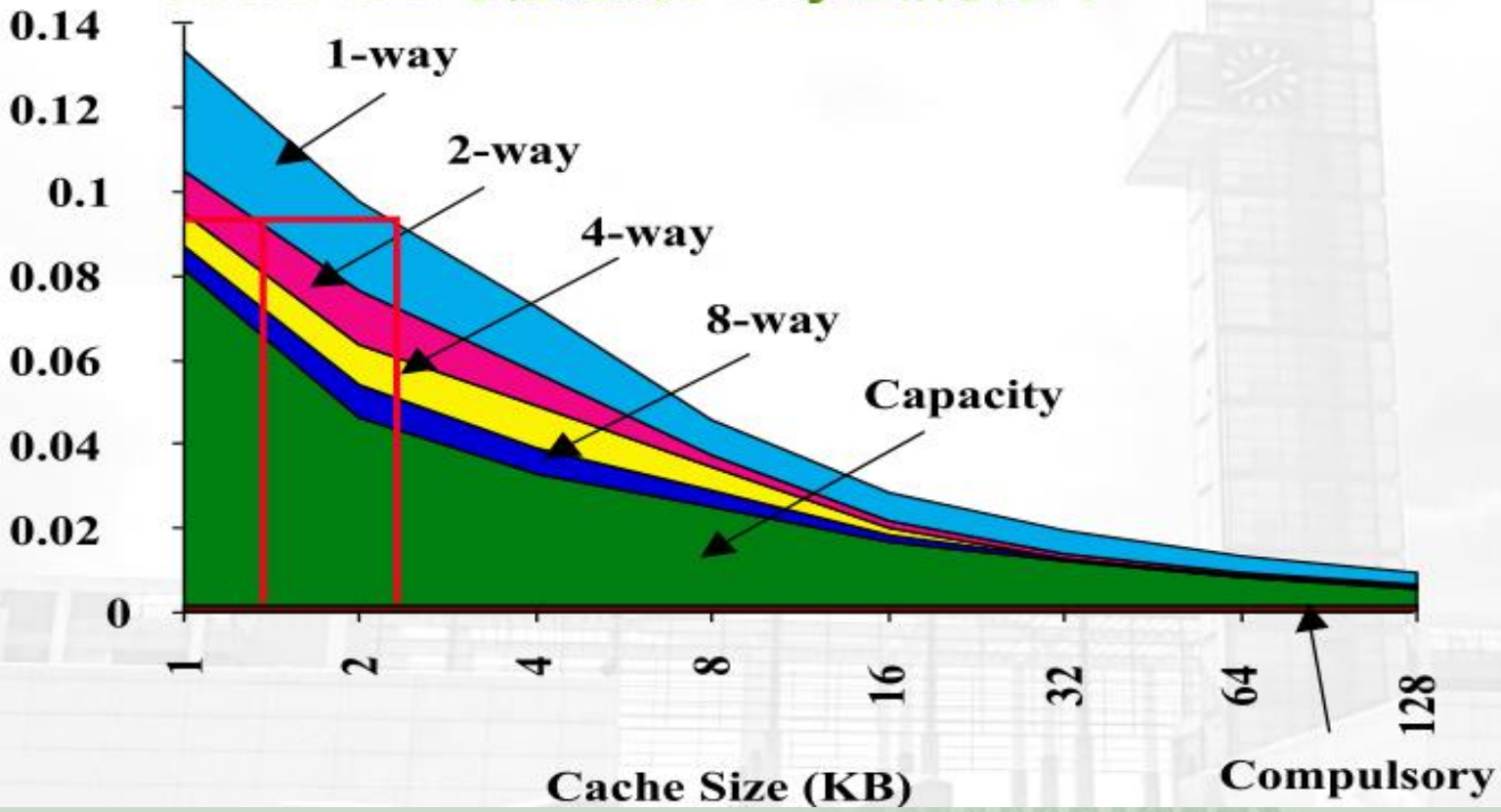
Cache性能分析：Miss的4C类型

- Capacity和Conflict Miss 随容量提升而降低



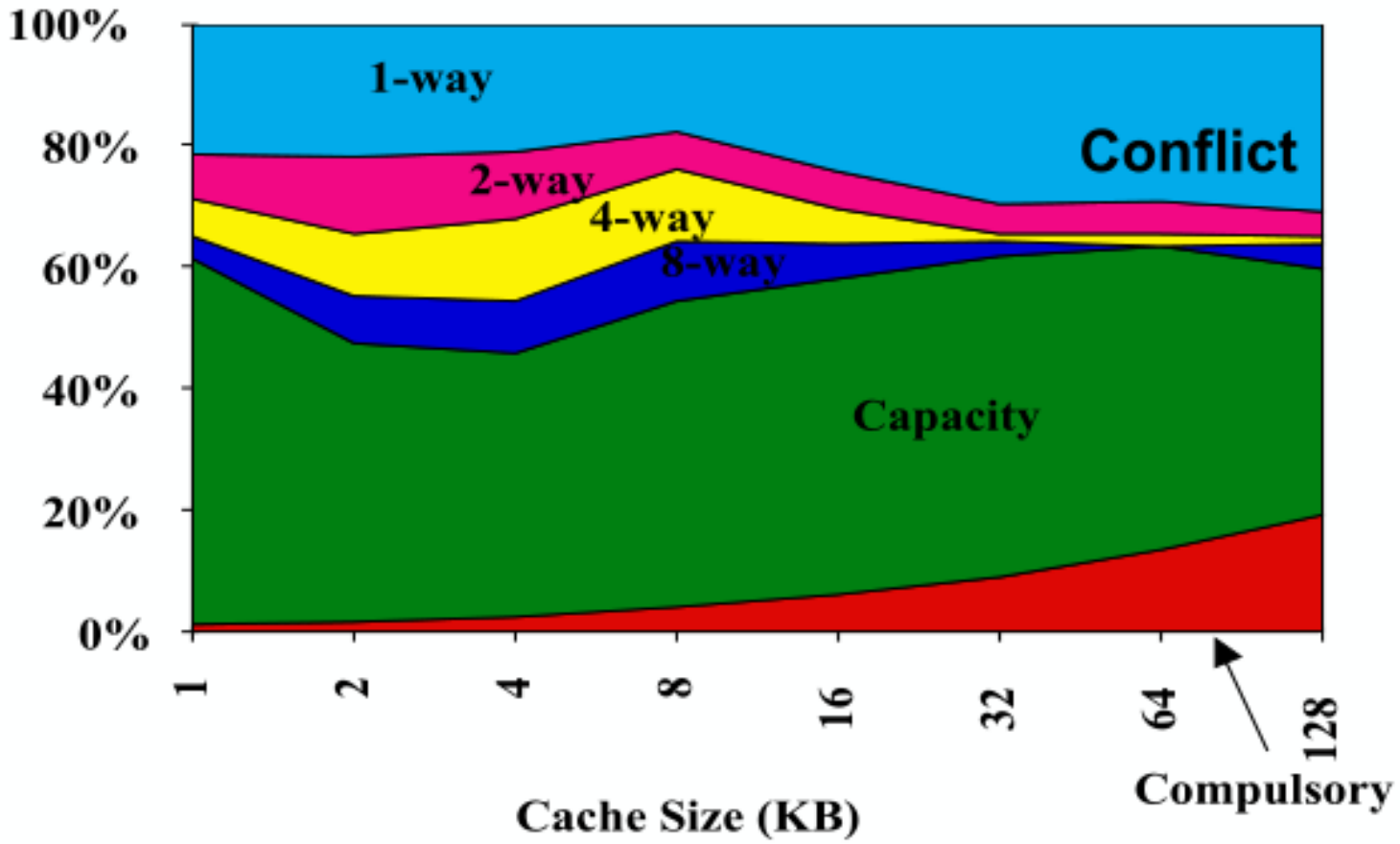
Cache性能分析：Miss的4C类型

- 多路组相联明显降低miss rate
  - 相同的miss rate，1-way组相联(直接映射)需要2倍的Cache size才能达到2-way组相联水平



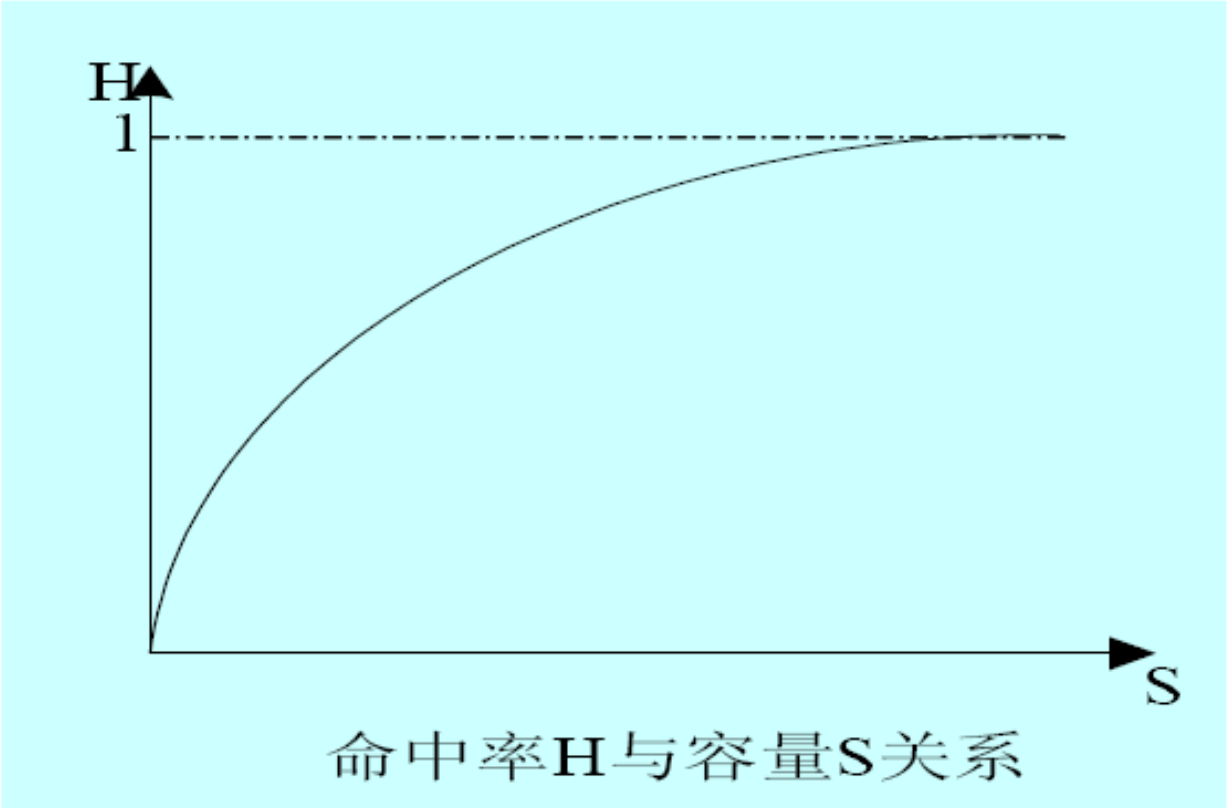
Cache性能分析：Miss的4C类型

- 3种miss的比例



Cache性能分析与优化1：

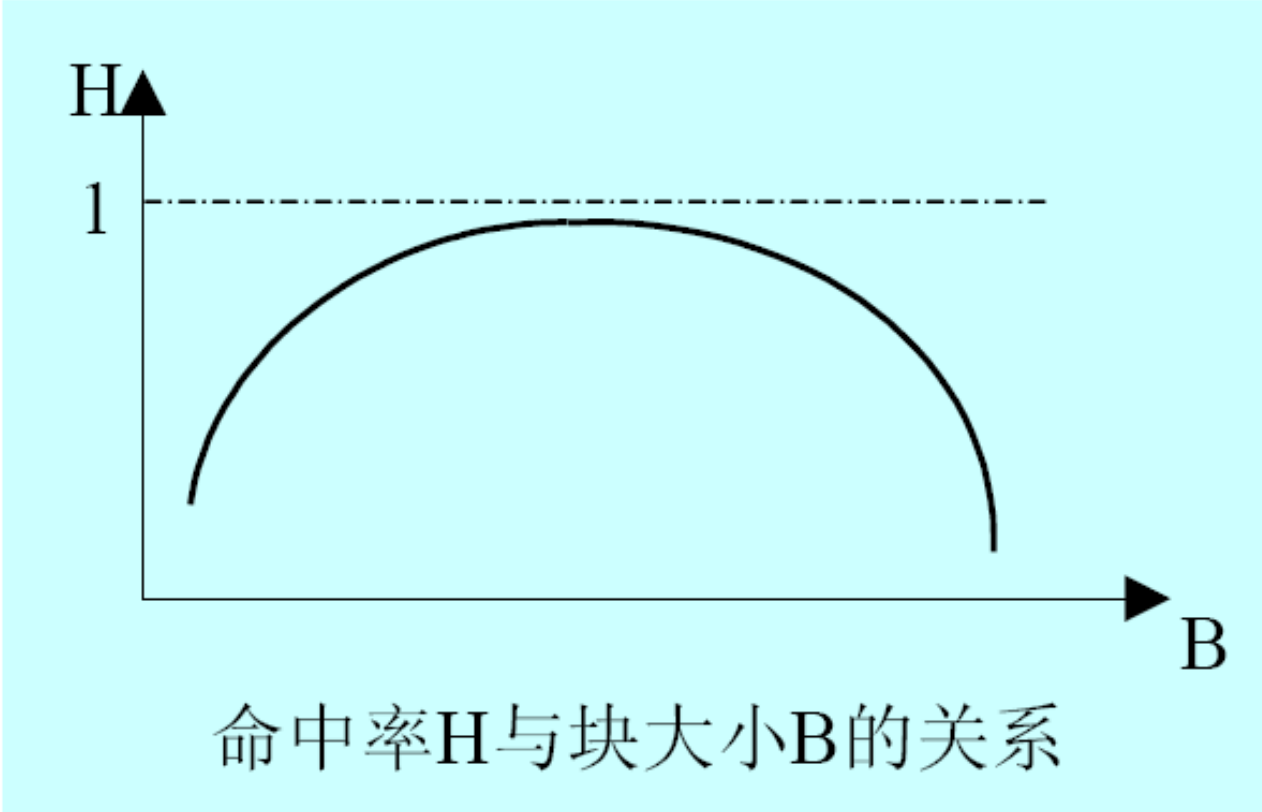
- 命中率与Cache容量的关系



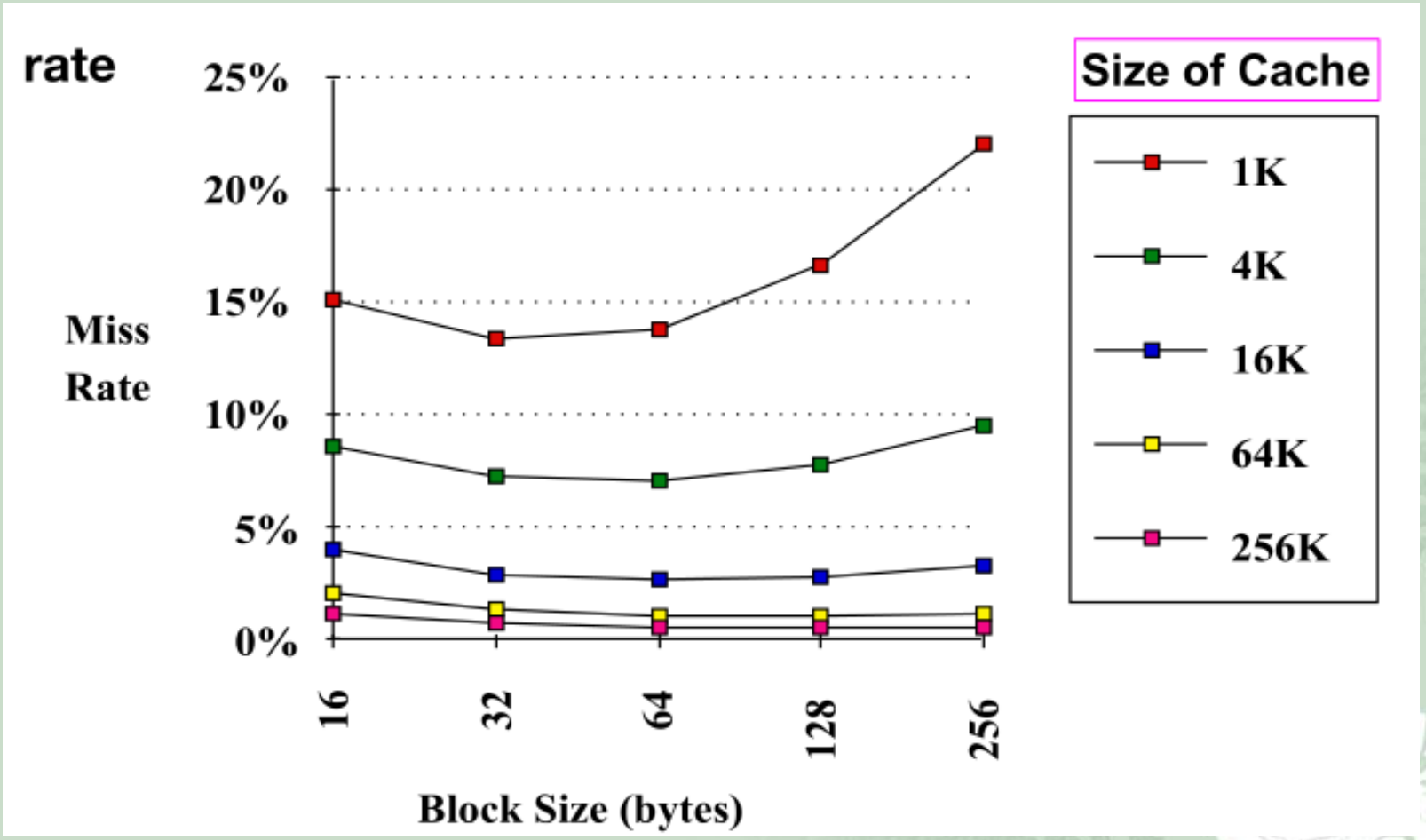
命中率H与容量S关系

Cache性能分析与优化2：

- 命中率与块大小的关系

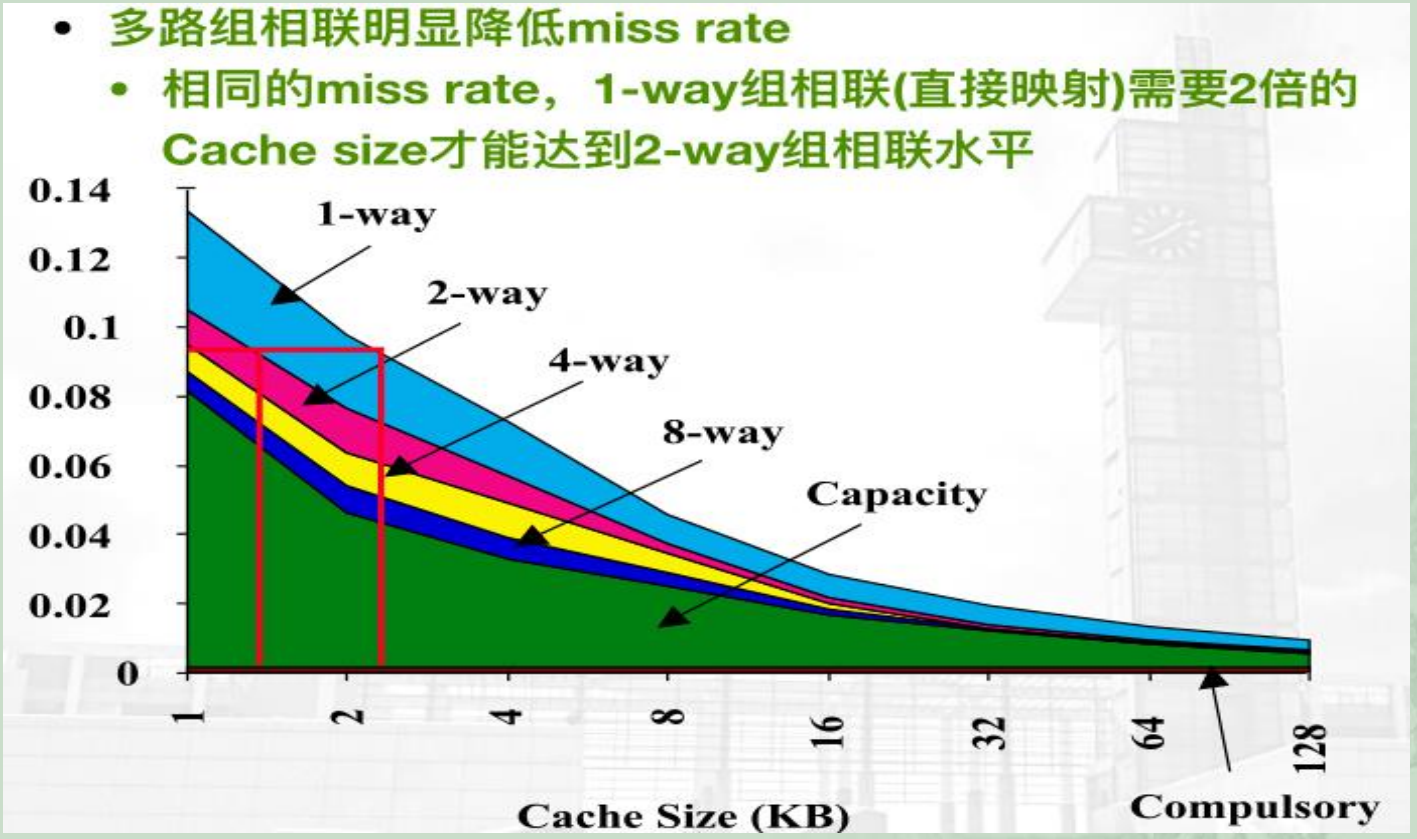


命中率H与块大小B的关系


rate
25%
20%
15%
10%
5%
0%
Miss
Rate
16
32
64
128
256
Block Size (bytes)
Size of Cache
1K
4K
16K
64K
256K

Cache性能分析与优化3：

- 多路组相联明显降低miss rate
  - 相同的miss rate，1-way组相联(直接映射)需要2倍的Cache size才能达到2-way组相联水平



- 提升组相联级别
- 级别也不要太高：级别提升 的边际效率在下降
  - 相联比较复杂
  - 相联存取复杂—>提高TC时间

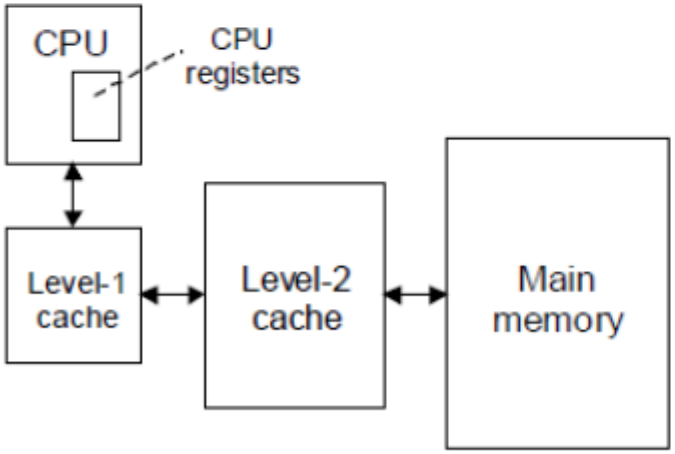
Cache性能分析与优化4：

- **两级Cache**

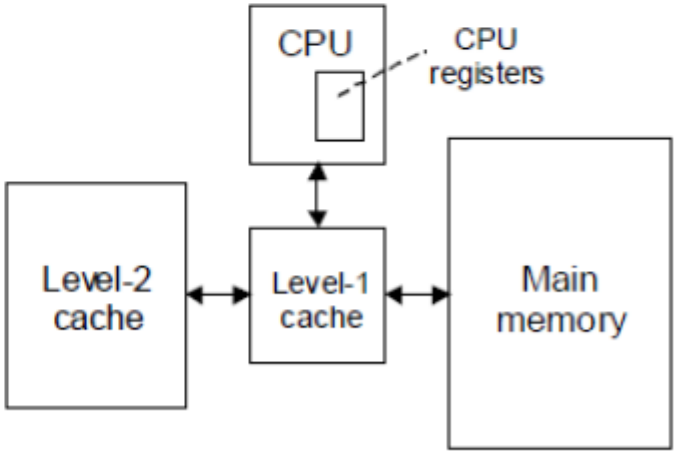
为了克服**CPU**和主存间的性能差距，使两者更好的匹配，目前均在原有的**Cache**和主存之间增加另一级**Cache**，构成两、三级**Cache**系统。总失效率=(失效率)$_{第一级}$ × (失效率)$_{第二级}$

One level of cache with hit rate h

$$C_{eff} = hC_{fast} + (1 - h)(C_{slow} + C_{fast}) = C_{fast} + (1 - h)C_{slow}$$



(a) Level 2 between level 1 and main



(b) Level 2 connected to "backside" bus

- 两级Cache的总未命中率(总失效率):

  总失效率 = $(\text{失效率})_{\text{第一级}} \times (\text{失效率})_{\text{第二级}} \ldots\ldots \times (\text{失效率})_{\text{第N级}}$

- 【例】10000次访存，第一级Cache失效400次，第二级Cache失效4次。
  - $(\text{失效率})_{\text{第一级}} = 400/10000 = 4\%$
  - $(\text{失效率})_{\text{第二级}} = 4/400 = 1\%$
  - 利用两级Cache后总的失效率 = 0.04%。

多级Cache的命中率

【例】访问内存需50ns, L1 1ns 10%失效率，L2 5ns 1%失效率，L3 10ns 0.2%失效率，求L1， L1+L2，L1+L2+L3构架下的平均访问时间。

L1： $T = 1\ \text{ns} + (0.1 \times 50\ \text{ns}) = 6\ \text{ns}$

L1+2： $T = 1\ \text{ns} + (0.1 \times [5\ \text{ns} + (0.01 \times 50\ \text{ns})]) = 1.55\ \text{ns}$

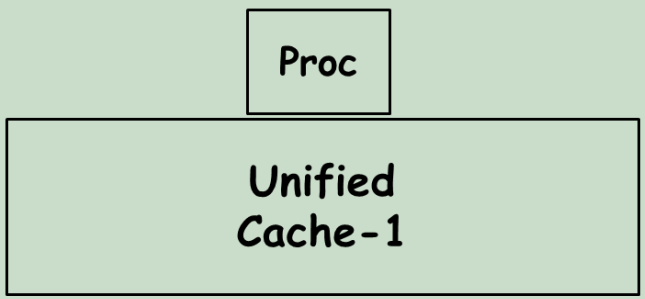
L1+2+3： $T = 1\ \text{ns} + (0.1 \times [5\ \text{ns} + (0.01 \times [10\ \text{ns} + (0.002 \times 50\ \text{ns})])]) = 1.5001\ \text{ns}$

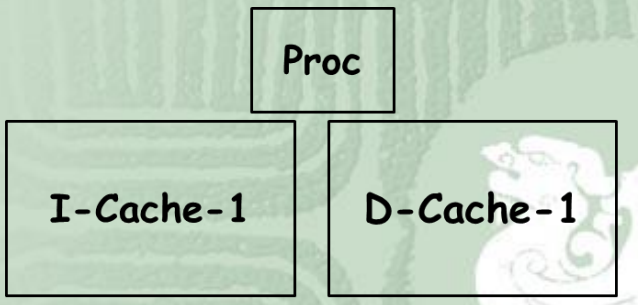
Cache性能分析与优化5：

拆分出指令/数据独立Cache

- **例如**
  - **16KB 指令/数据独立Cache**
    - **指令 miss rate=0.64%, 数据 miss rate=6.47%**
  - **32KB 统一Cache: miss rate=2.99%**
  - **假设75%指令,25%数据，Hit time = 1, Miss time = 50**

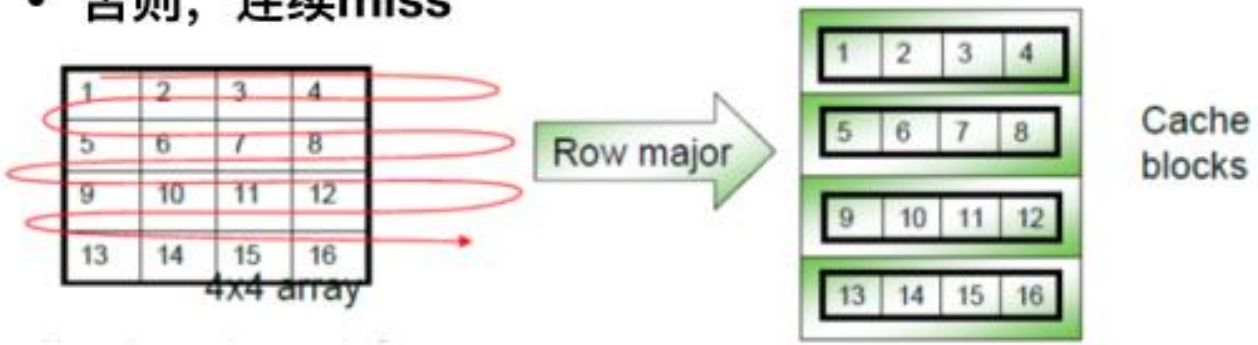
T独立=75%x(1+0.64%x50)+25%x(1+6.47%x50) = 2.05
T统一=75%x(1+2.99%x50)+25%x(1+2.99%x50)= 2.50





Cache性能分析与优化6：

- 代码层面优化　：1) Row Major
- 注意循环：大部分计算和访存都在循环中发生
  - 按照数据对象在存储器中的存储顺序读取，最优
  - 否则，连续miss



- 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
  }
}
```

- 优化后(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]
  }
}
```

Cache性能分析与优化6：

2) 循环合并

```
for(i := 0; i < N; i := i+1)
    for(j := 0; j < N; j := j+1)
        a[i][j] := 1 / b[i][j] * c[i][j];
for(i := 0; i < N; i := i+1)
    for(j := 0; j < N; j := j+1)
        d[i][j] := a[i][j] + c[i][j];
```

- 原程序中a和c访问两次
- 有可能不在cache中

优化后a和c第2次访问都在cache里

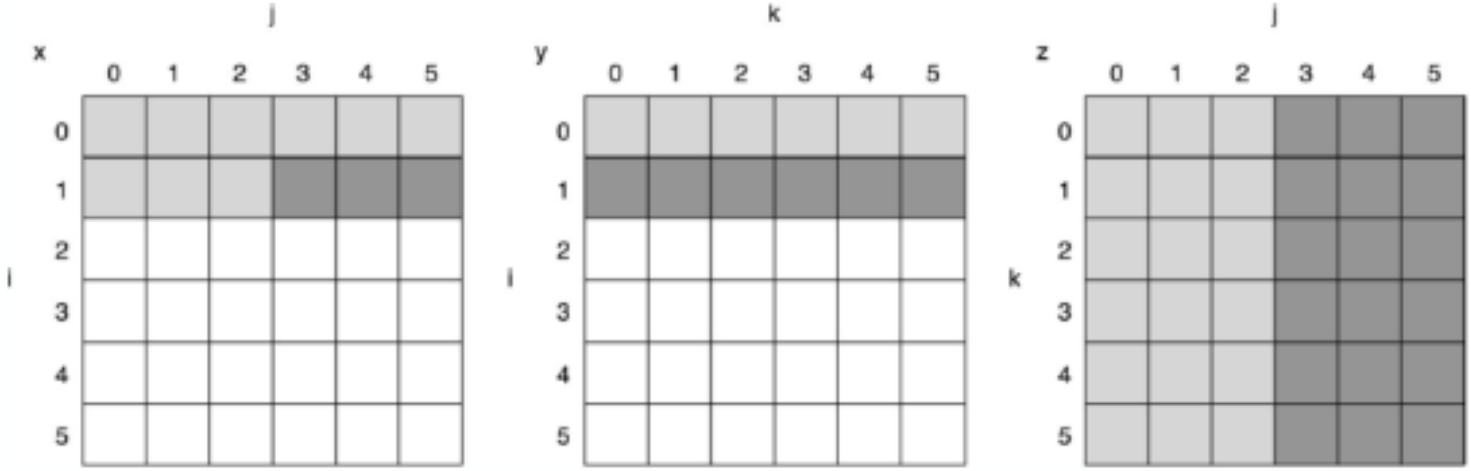
```
for(i := 0; i < N; i := i+1)
    for(j := 0; j < N; j := j+1){
        a[i][j] := 1 / b[i][j] * c[i][j];
        d[i][j] := a[i][j] + c[i][j];}
```

Cache性能分析与优化6：

3) 分块运算

- 例如，矩阵.*运算
  - 当3个矩阵无法全放在cache时，严重的Capacity Miss

```
for(i := 0; i < N; i := i+1)
    for(j := 0; j < N; j := j+1){
        r := 0;
        for(k := 0; k < N; k := k+1)
            r := r + y[i][k] * z[k][j];
        x[i][j] := r;}
```



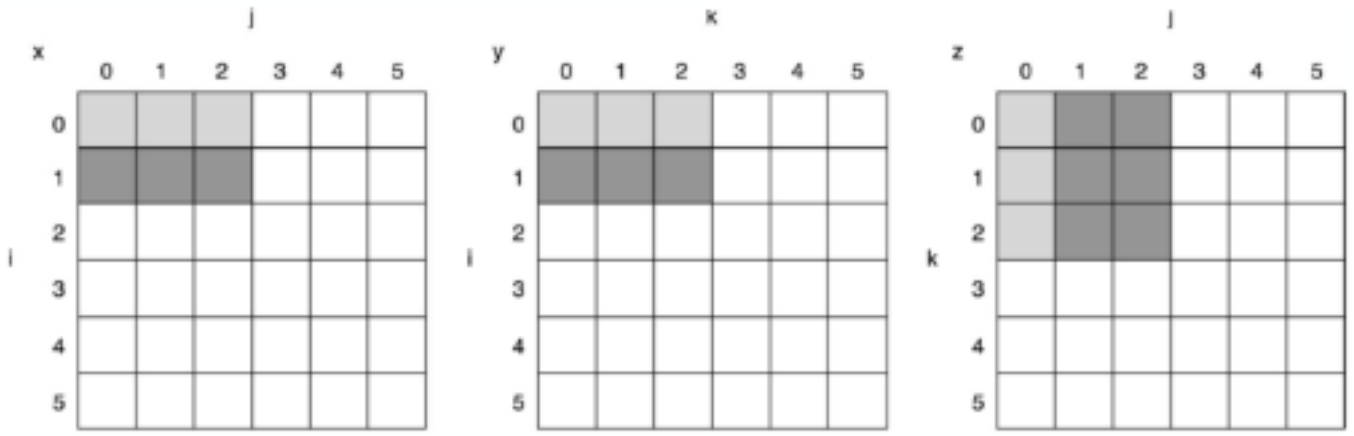
Cache性能分析与优化6：

3) 分块运算

- 例如，矩阵.*运算
  - 当3个矩阵无法全放在cache时，严重的Capacity Miss
  - 把矩阵分成小块执行，充分避免Capacity Miss

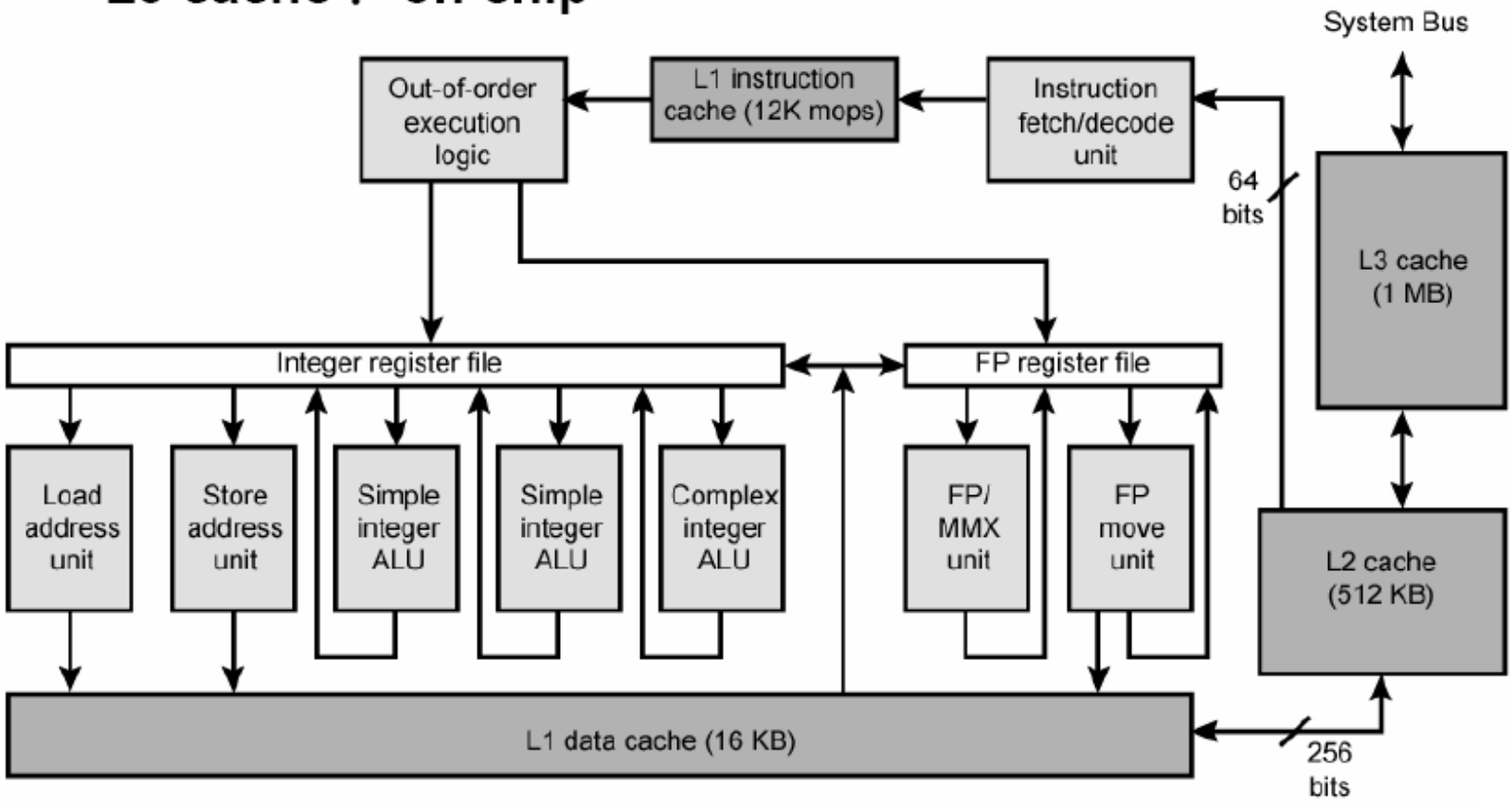
After:

```
for(jj := 0; jj < N; jj := jj+B)
for(kk := 0; kk < N; kk := kk+B)
for(i := 0; i < N; i := i+1)
    for(j := jj; j < min(jj+B-1,N); j := j+1){
        r := 0;
        for(k := kk; k < min(kk+B-1,N); k := k+1)
            r := r + y[i][k] * z[k][j];
        x[i][j] := x[i][j] + r;}
```



实例：Pentium的Cache

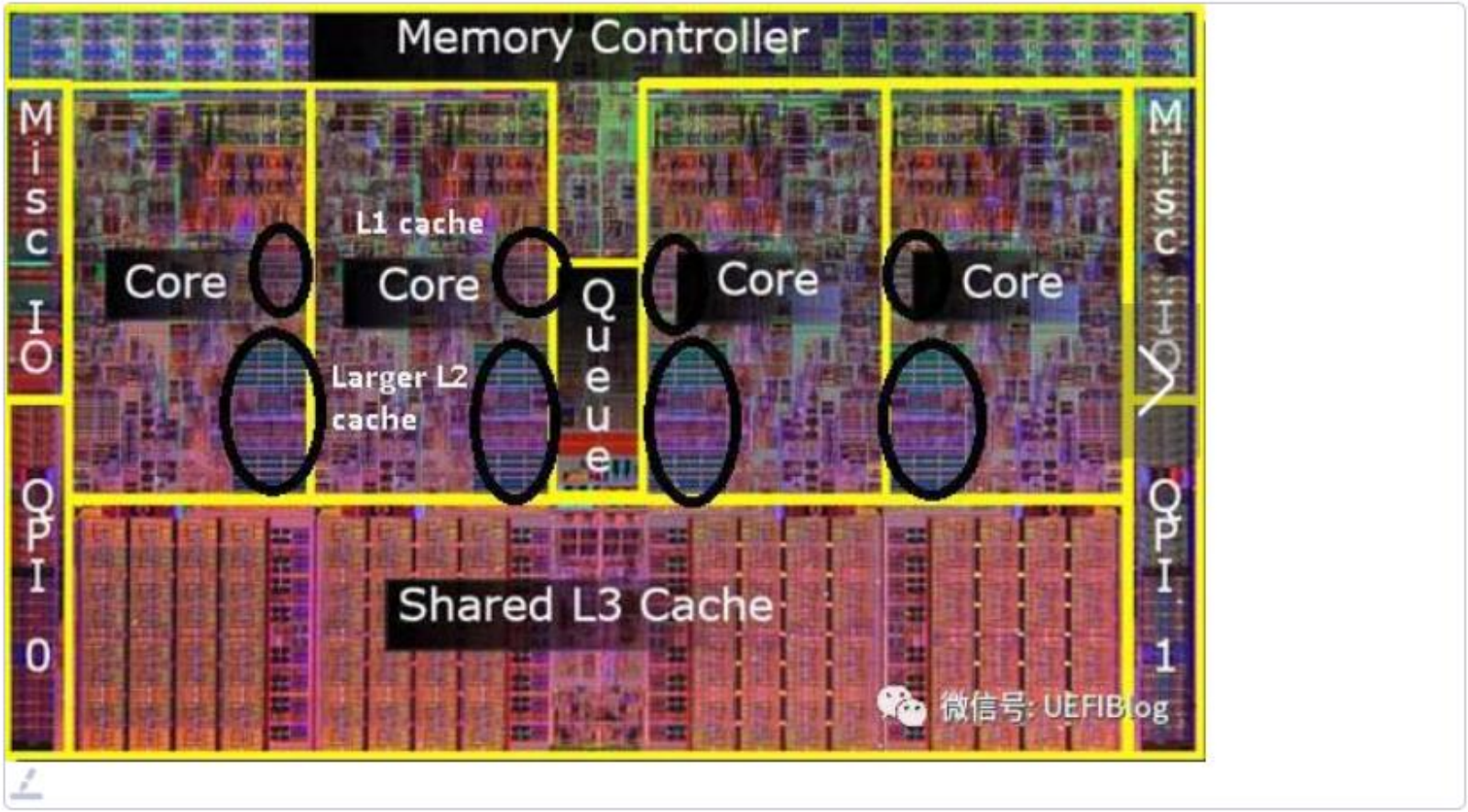
- **Pentium 4 Cache**
  - L1 caches: 64 byte lines，4 way set associative
  - L2 cache: 128 byte lines，8 way set associative
  - L3 cache : on chip



- “64-byte line size” 和“128-byte line size”
  表示Cache的Line长为64字节和128字节。
  Line为Cache每次向下级存储设备读取数据的大小。

- 4 way set associative， 8 way set associative
  表示4路组相联、8路组相联

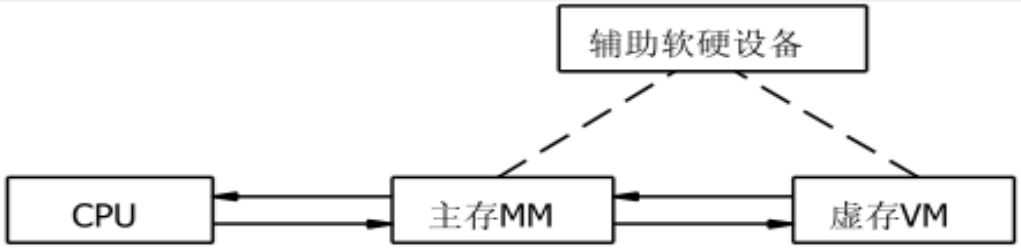
- 现在：L3被加入到CPU Die中，它在逻辑上是共享模式。而L2则被每个Core单独占据。这时L2也常被指做**MLC**（Middle Level Cache），而L3也被叫做**LLC**（Last Level Cache）：



7.4 虚拟存储器（VM）

7.4.1 虚拟存储器的概念

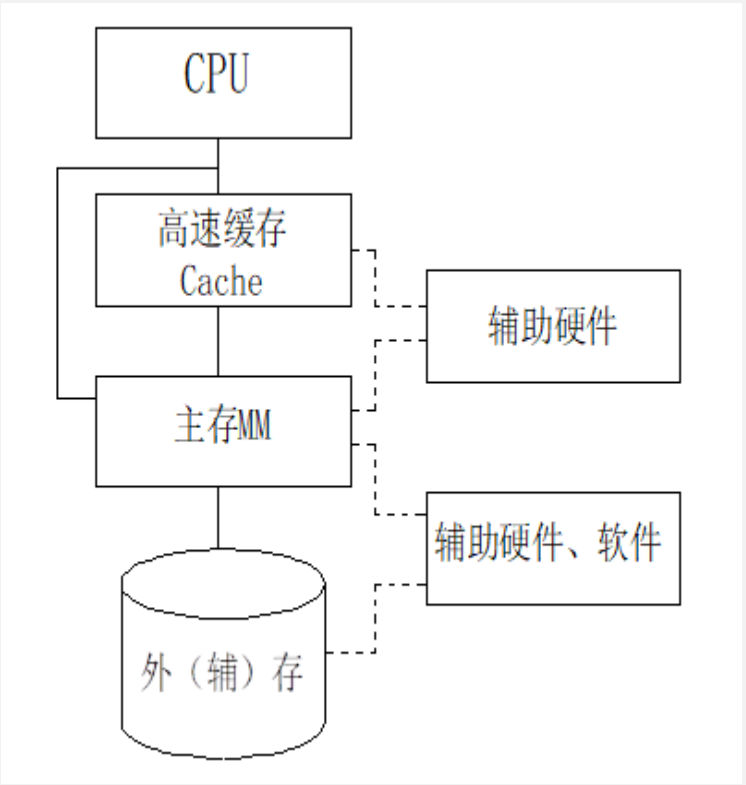
- 虚拟存储技术是在主存与辅存之间，增加软件及必要的硬件，使主、辅存之间的信息交换，程序的再定位，地址的转换都能自动进行，使两者形成一个有机的整体。以透明的方式给用户提供了一个比实际主存空间大得多的程序地址空间。
- 由于程序员可以用到的空间远远大于主存的实际空间，但实际并不存在这么大的主存，故称“虚拟存储器”（Virtual Memory）简称 VM 。



虚拟存储器原理

7.4.1 虚拟存储器的概念

- 虚拟存储器是由价格比较贵、容量不太大、速度相对比较高的 主存储器 和价格很低、容量非常大、速度慢的 外部（辅助）存储器，在 操作系统 及 辅助硬件 的管理下，构成了像一个单一的、可直接访问的超大容量的主存储器。
- 解决主存容量与价格的矛盾，使速度接近主存速度而容量和价格又接近外存.



7.4.2 虚拟存储器的管理

主存-辅存的工作原理 （略）

总结：Cache-主存和主存-辅存两个存储层次的比较

相同点

- 主存-外存层次和cache-主存层次用的地址变换映射方法和替换策略的思想是相同的，都基于程序局部性原理。

- 遵循的原则都是：
  - ✓ 把程序中最近常用的部分驻留在高速的存储器中。
  - ✓ 一旦这部分变得不常用了，把它们送回到低速的存储器中。
  - ✓ 这种换入换出是由硬件或操作系统完成的，对用户是透明的。
  - ✓ 力图使存储系统的性能接近高速存储器，价格接近低速存储器。

总结：Cache-主存和主存-辅存两个存储层次的比较

区别

- 目的不尽相同。 cache主要解决主存与 CPU的速度差异问题；而辅存主要是解决存储容量的问题。
- 数据通路不同。CPU与cache和主存之间均有直接访问通路，cache不命中时可直接访问主存；而辅存与CPU之间不存在直接的数据通路，当主存不命中时只能通过调页解决，CPU最终还是要访问主存。
- 透明性不同。 cache的管理完全由硬件完成，对系统程序和应用程序均透明；而虚存管理由软件(操作系统)和硬件共同完成，对系统程序不透明，对应用程序透明(段式和段页式管理对应用程序“半透明”).
- 未命中时的损失不同。由于主存的存取时间是 cache 的5~10倍，而辅存的存取时间通常是主存的上千倍，故虚存未命中时系统的性能损失罗远大于 cache未命中时的损失。

本章作业-2

第22、

23（1）（2）（3）、

24、

27题

注意：各题要求写出过程，包括23、24题