

第4章

存储体系

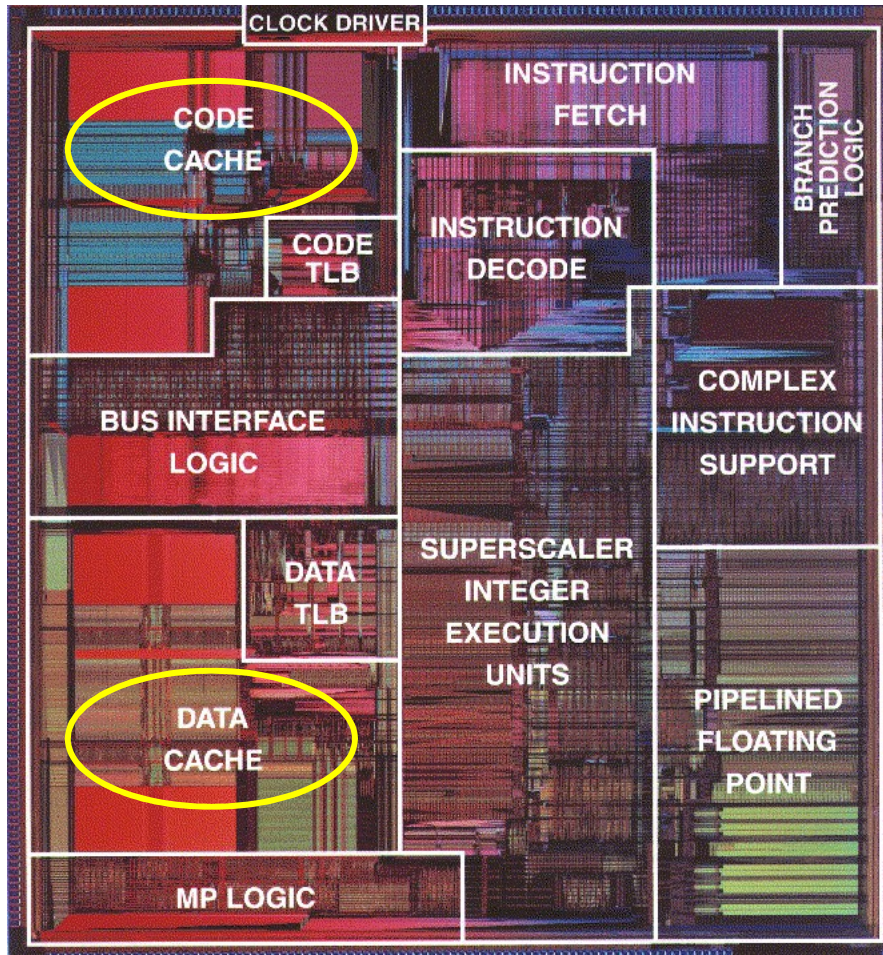
郑宏 副教授
计算机学院
北京理工大学

第四章 存储体系

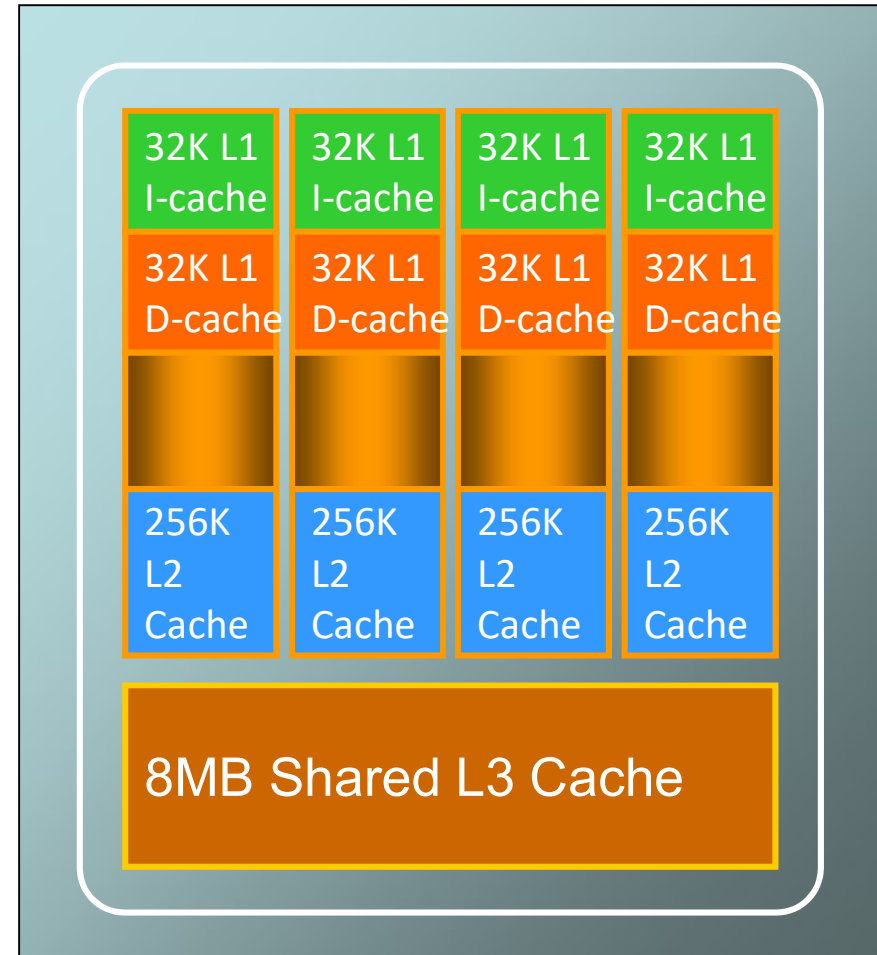
学习内容：

- 4.1 存储体系概念和并行存储系统
- 4.2 虚拟存储系统
- 4.3 高速缓冲存储器（Cache）
- 4.4 Cache - 主存 - 辅存三级层次
- ARM存储系统

高速缓冲存储器(Cache)

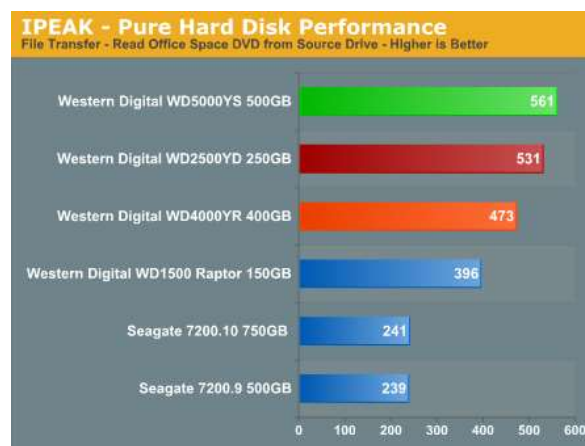
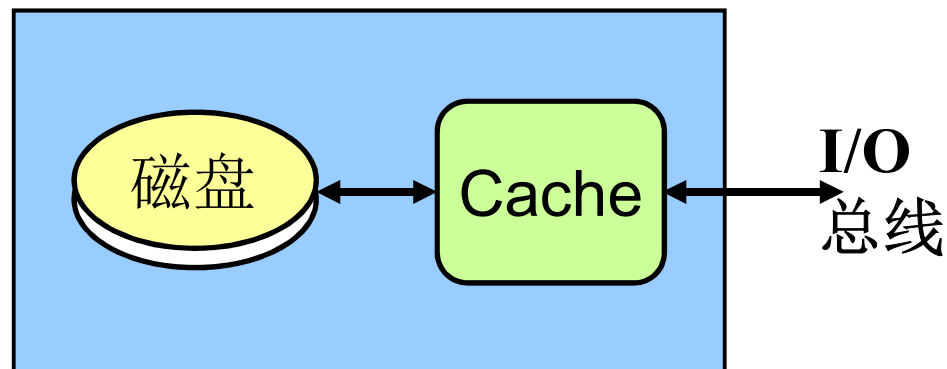


Intel Pentium 的Cache



Core i7 的Cache 结构

高速缓冲存储器(Cache)



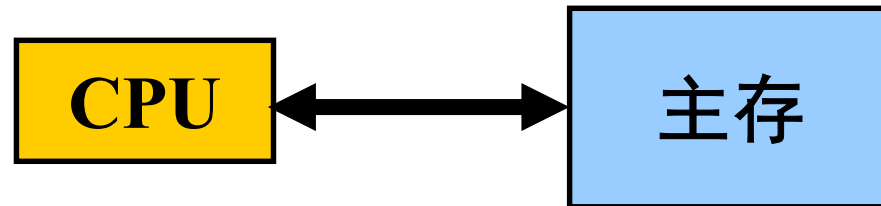
硬盘上的cache

主要内容

- 基本结构与工作原理
- 地址映像规则与地址变换
- 替换算法与实现
- 透明性与性能
- Cache层次

CPU-主存瓶颈

- 高速计算机的性能通常受到主存带宽和响应时间的限制。



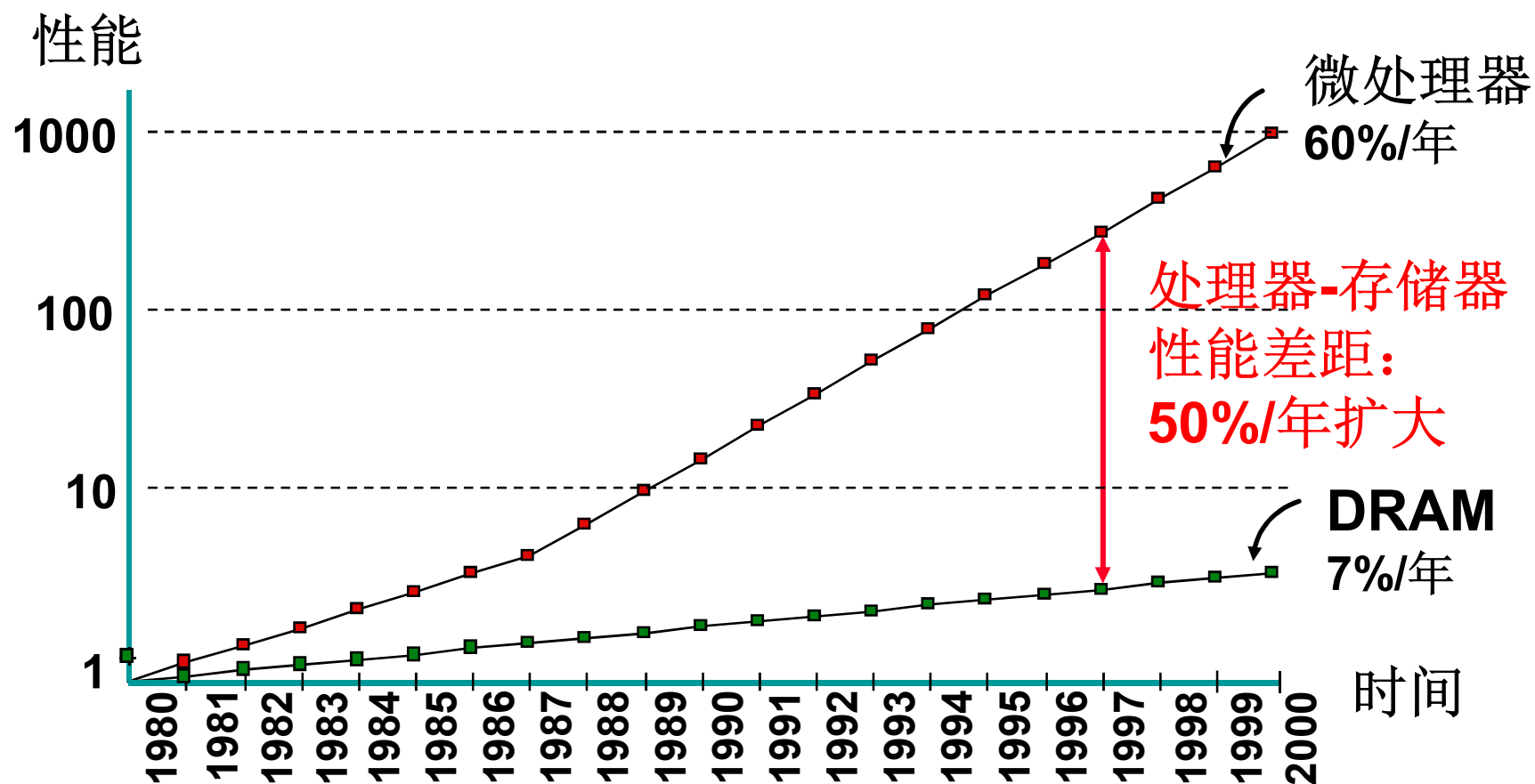
- 响应时间(Latency): 一次访存所需要的时间。

主存访存时间 \gg 处理器机器周期

- 带宽(bandwidth): 单位时间内的访存次数。

假设每条指令需要一个机器周期，一条指令需要访问主存 m 次，意味着每个机器周期需要访存 m 次。

CPU-主存瓶颈



CPU-主存瓶颈

解决速度问题方法：

■ 1. 改进工艺和设计，提高存储器的性能；

- DDR2

- ◆ Lower power (2.5 V -> 1.8 V)
- ◆ Higher clock rates (266 MHz, 333 MHz, 400 MHz)

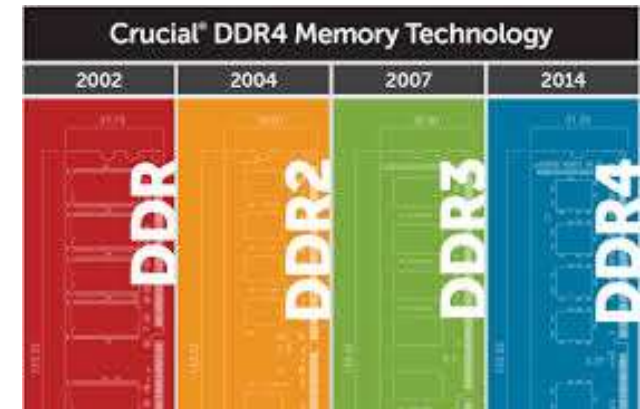
- DDR3

- ◆ 1.5 V
- ◆ 800 MHz

- DDR4

- ◆ 1-1.2 V
- ◆ 1600 MHz

- GDDR5 is graphics memory based on DDR3

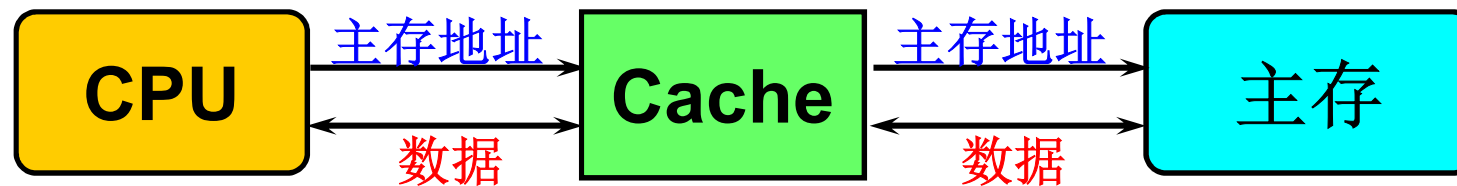


Key Features of Crucial DDR4					
Product	Clock Rate		Data Rate		Density
	Max	Min	Min	Max	
DDR3	2.5ns	1.25ns	800 Mb/s	1600 Mb/s	1-8Gb
DDR4	1.25ns	0.625ns	1600 Mb/s	3200 Mb/s	4-16Gb

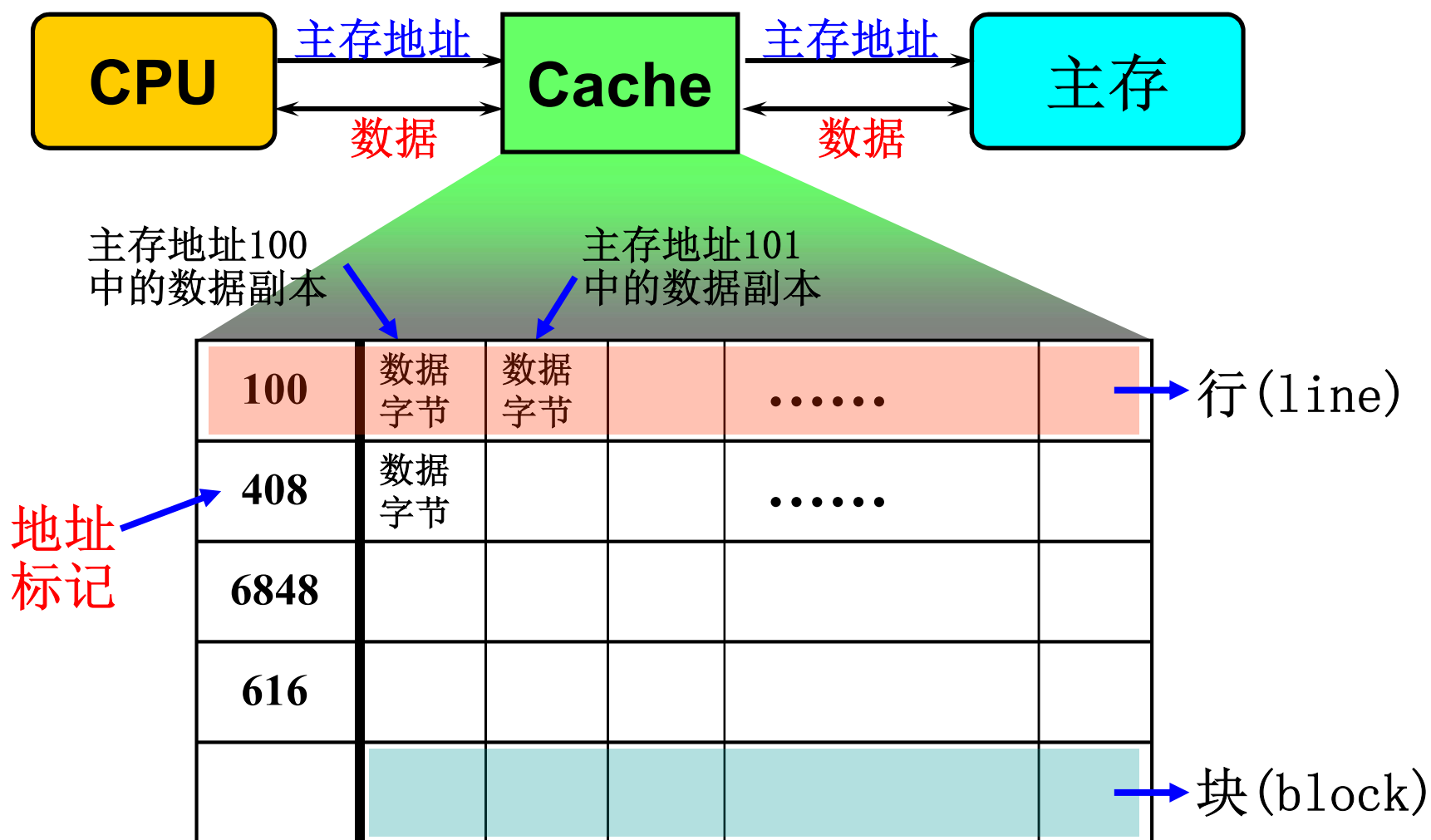
CPU-主存瓶颈

解决速度问题方法：

- 2. 在CPU和主存之间设置速度快、容量小的高速缓冲存储器(cache)。依据程序局部性原理，将未来要用到的指令或数据从低速主存预取到高速cache中，从而减少平均响应时间，提高平均访问速度。



基本结构和工作原理



Cache中缓存数据的基本结构

基本结构和工作原理

Cache Array Showing full Tag

Tag	Data	Data	Data	Data
1234	from 1234	from 1235	from 1236	from 1237
2458	from 2458	from 2459	from 245A	from 245B
17B0	from 17B0	from 17B1	from 17B2	from 17B3
5244	from 5244	from 5245	from 5246	from 5247

- **16** 位地址，用 **4** 个 **16** 进制数表示
- 本例中，每行包含 **4** 字节
- 这 **4** 个字节的地址的前 **14** 位相同
- 标记为第 **1** 个字节的地址

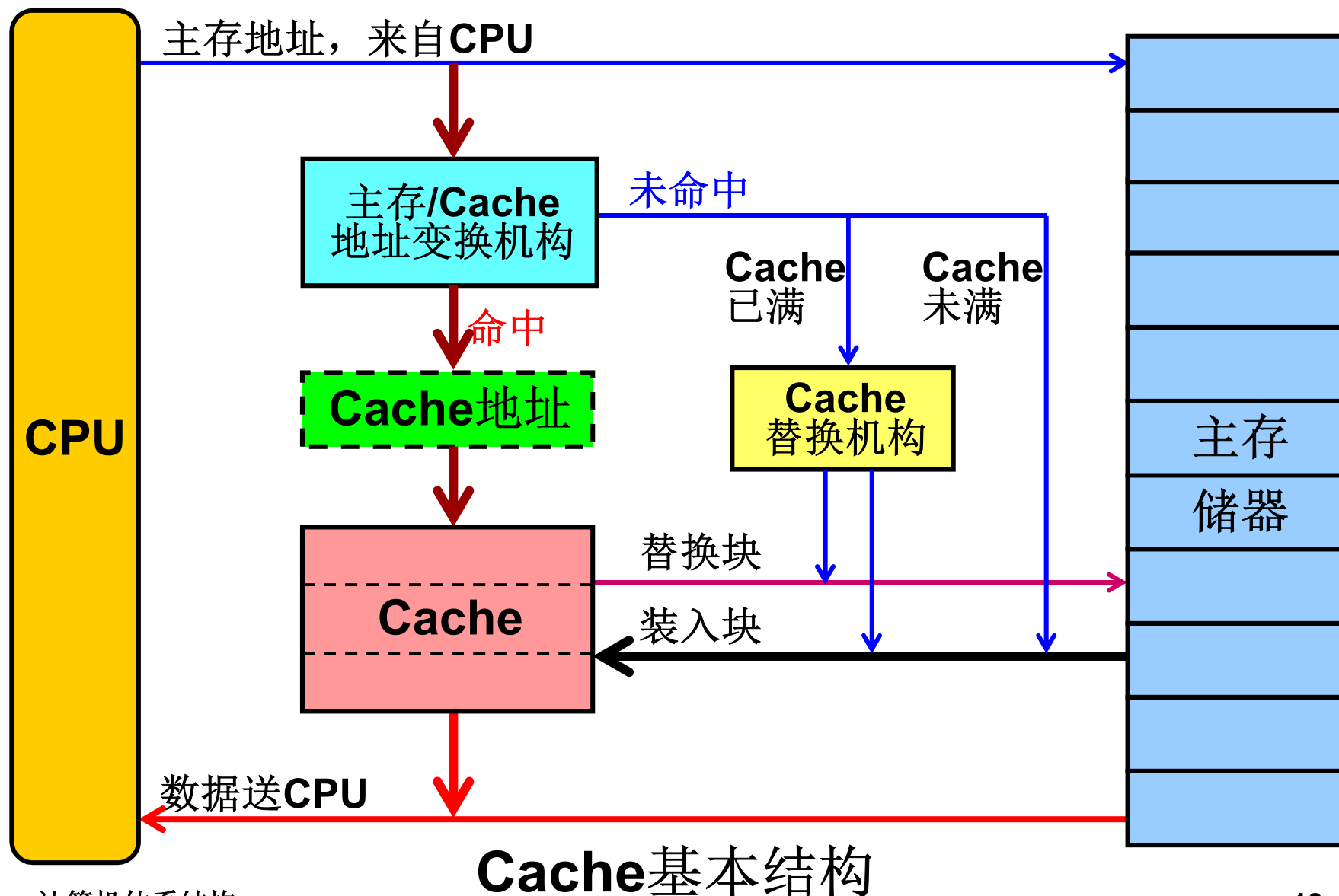
基本结构和工作原理

Cache Array Showing Tag

Tag	Data	Data	Data	Data
48D	from 1234	from 1235	from 1236	from 1237
916	from 2458	form 2459	from 245A	from 245B
5EC	from 17B0	from 17B1	from 17B2	from 17B3
1491	from 5244	from 5245	from 5246	from 5247

- 16 位地址，用 4 个 16 进制数表示
- 本例中，每行包含 4 字节
- 这 4 个字节的地址的前 14 位相同
- 标记为地址的前 14 位 (0x1234 = 0001 0010 0011 0100, 前 14 位 = 00 0100 1000 1101 = 0x48D)

基本结构和工作原理



基本结构和工作原理

Cache工作原理：读取

将**CPU**给出的主存地址
变换为**Cache**地址，搜索**Cache**

在**Cache**中找到
(命中)

访问**Cache**，
向**CPU**返回
Cache中的数据副本。

在**Cache**中未找到
(未命中)

1. 从主存中读取数据块
2. 等待...
3. 向 **CPU** 返回数据，
更新**cache**（满时替换）

只要**Cache**的命中率足够高，
就能以接近于**Cache**的速度访问主存。

基本结构和工作原理

Cache工作原理：写入

将**CPU**给出的主存地址
变换为**Cache**地址，搜索**Cache**

在**Cache**中找到
(命中)

写**Cache**，写主存
(存在一致性问题)

在**Cache**中未找到
(未命中)

写主存
(与**Cache**无关)

为保持**Cache**与主存中的内容一致，
采取以下方法：

- 写直达法
- 写回法

Cache特点

- Cache与CPU采用**相同工艺**；
- 地址映象、变换、替换、调度等**由专门的硬件实现**；
- Cache靠近CPU或就放在CPU中，以减少与CPU之间的传输延迟；
- Cache—主存之间的信息交换**对所有程序员都透明**；
- Cache访问主存的**优先级高于**其他系统访问主存的优先级；
- 除了Cache和CPU有直接的通路外，主存和CPU也有直接的通路，可以实现**读直达和写直达**；

Cache与虚拟存储器的区别

存储层次 比较项目	Cache	虚拟存储器
目的	弥补主存速度的	弥补主存容量的不足
存储管理实现	由专用硬件实现	软件、硬件实现
访问速度的比值 (第一级和第二级)	几比一	几百比一
典型的块(页)大小	几十个字节	几百到几千个字节
CPU对第二级的 访问方式	可直接访问	均通过第一级
失效时CPU是否切换	不切换	切换到其他进程
透明性	对所有程序员透明	仅对应用程序员透明

地址映像规则与地址变换

■ Cache系统须解决三个问题：

1. 定位问题

- 将主存中的数据装入cache的哪个位置；
- 如何知道主存中的数据已经装入cache（即是否命中）；
- 如果命中，如何形成Cache地址并访问主存。

2. 替换问题

- 若未命中或失效，需将数据从主存调入Cache；
- 若Cache满，则按何种算法将Cache中的数据替换出去。

3. 数据一致性问题

- 如何保证Cache内容与主存内容的一致。

地址映像规则与地址变换

■ 地址映像：

把主存中的数据按照某种规则装入Cache中，并建立主存地址与Cache地址之间的对应关系，进而根据主存地址，判断Cache有无命中并变换为Cache的地址。

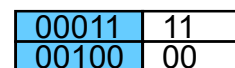
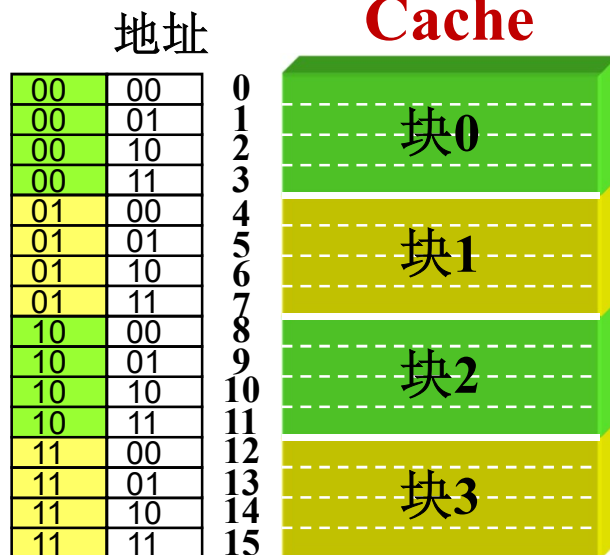
■ 为便于进行地址的映象和变换，也便于替换和管理，把Cache和主存**等分成相同大小的块**，这样，Cache—主存地址映像就演变为**主存中的块如何与Cache中的块相对应**。

地址映像规则与地址变换

Cache字节数=16
块内字节数= $2^2=4$

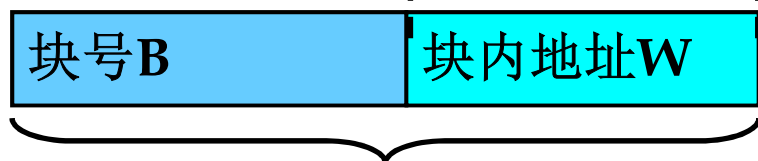
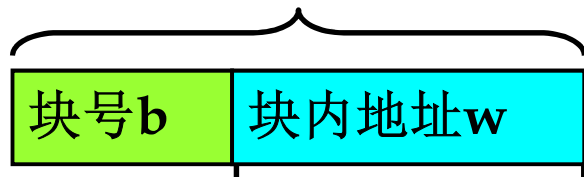
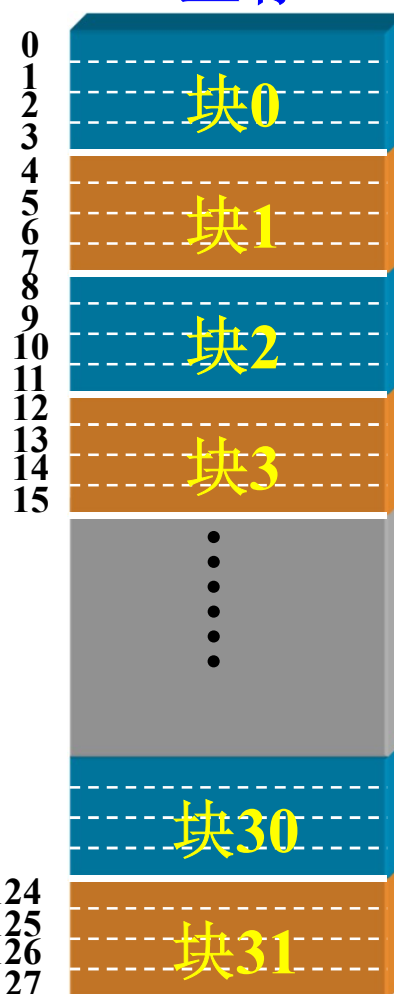
Cache

当Cache和主存的字节数，以及块数和块内字节数都是2的幂次时，地址好划分。

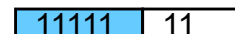


块内字节数= $2^2=4$
主存字节数= $2^7=128$
主存块数= $2^{7-2}=32$

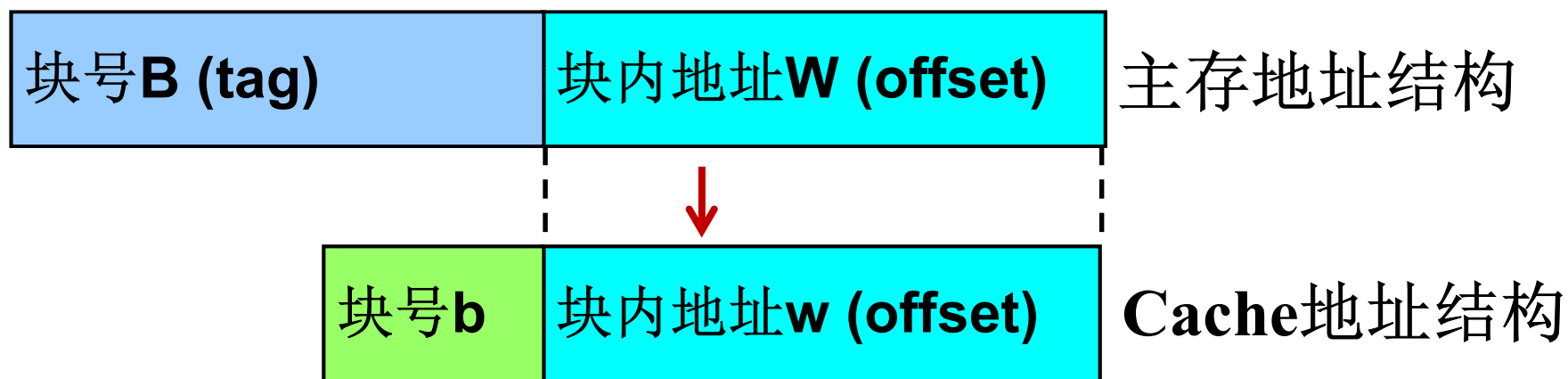
主存



计算机体系结构



地址映像规则与地址变换



■ 主存地址 到 Cache地址的变换：

Cache 块内地址 w = 主存内地址 W （因块大小相同）

Cache 块号 $b = f$ (主存块号 B 或 tag)

地址映像规则与地址变换

- 可以采用的地址映像方法有很多。选择依据：
 - 地址映像和变换硬件的速度是否高，价格是否低，实现是否容易；
 - Cache空间的利用率是否高；
 - 块冲突概率是否低；

■ 块冲突：

主存中的块要调入Cache中的某个位置，但该位置已经被其他主存块所占用。

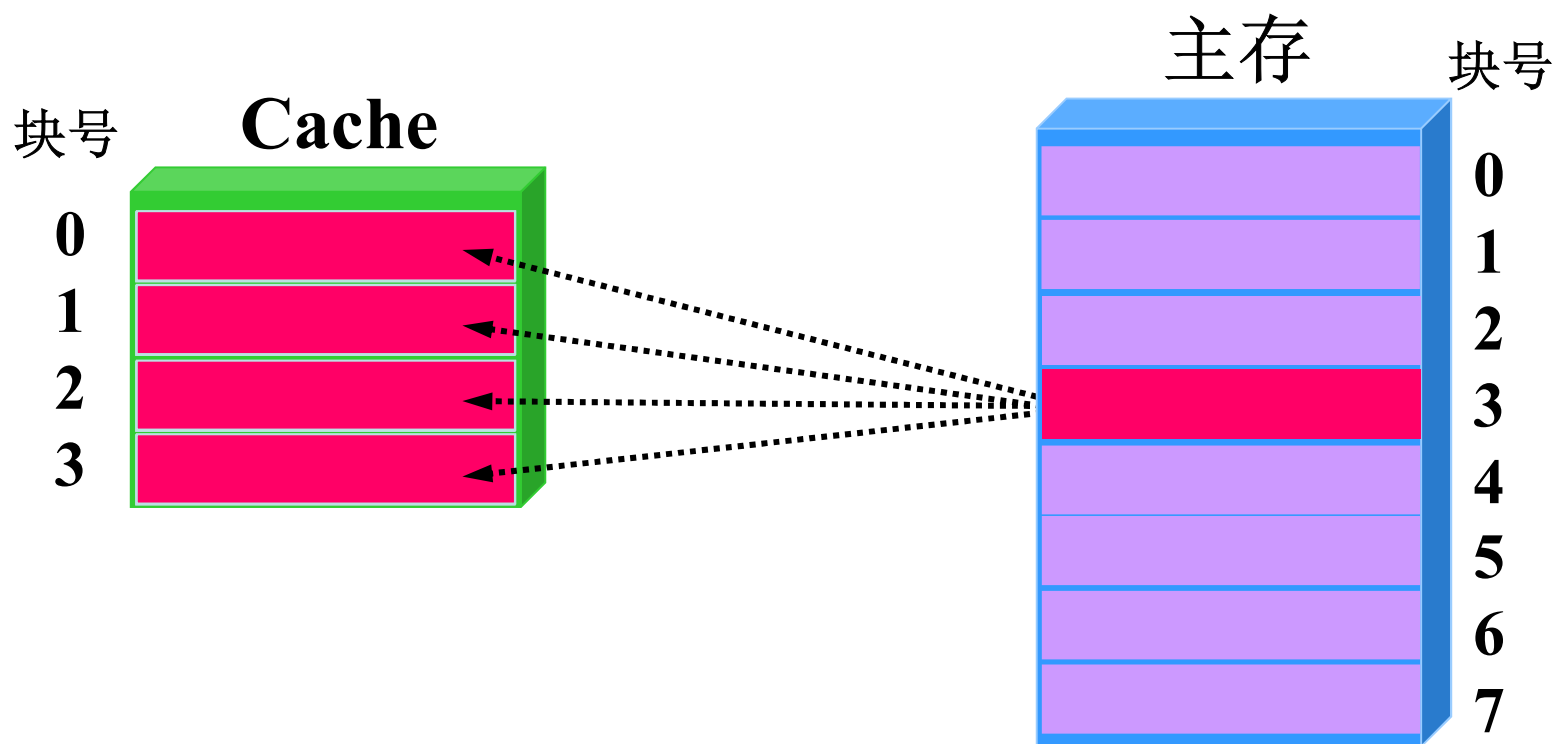
地址映像规则与地址变换

■ 典型的地址映像与变换方法主要有：

1. 全相联映像与变换
2. 直接映像与变换
3. 组相联映像与变换

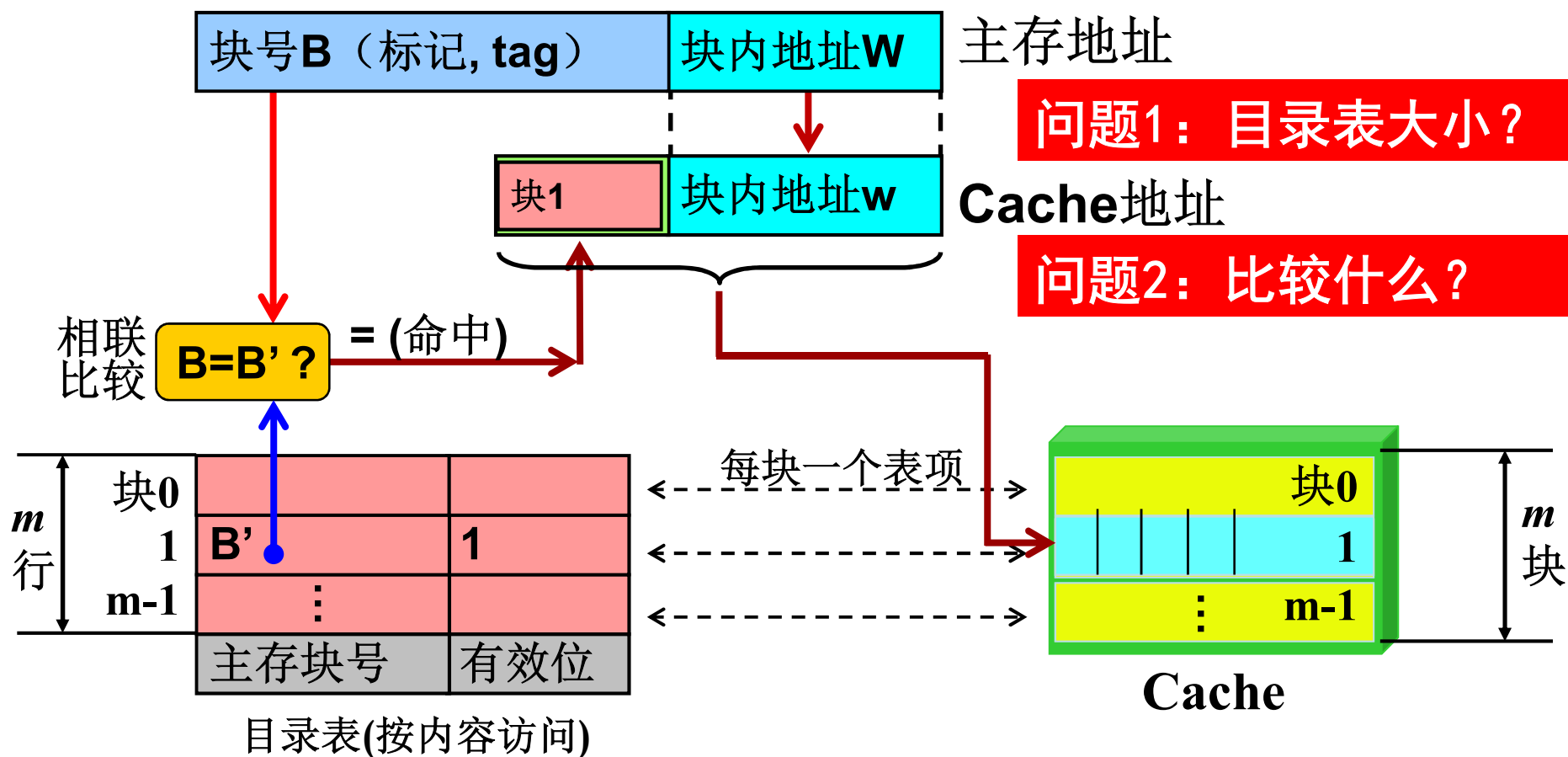
1. 全相联映像规则与变换

映像规则：主存中的任意一块都可以装入到Cache中的任意一个块位置。



1. 全相联映像规则与变换

地址变换：采用相联存储器构成的**目录表**，以硬件方式实现。

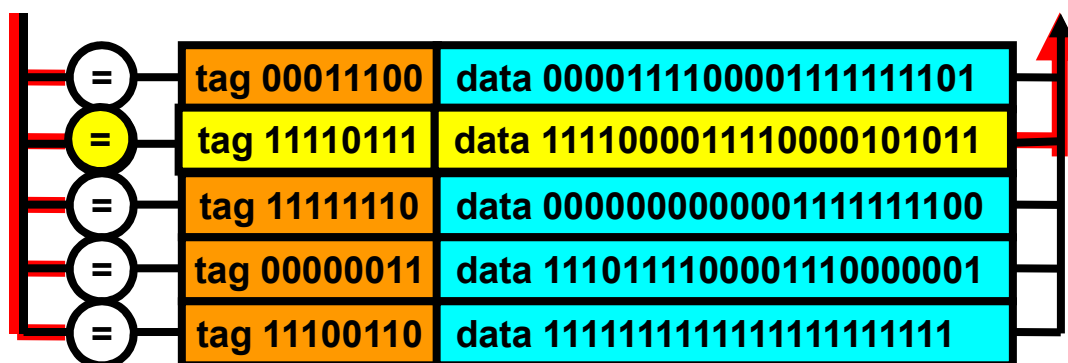


全相联Cache查找过程

问题： 逐行比较还是所有行同时比较？

tag in 11110111

data out 1111000011110000101011



所有行同时比较

Key idea:

- 1 comparator required for each block
- Practical only for small caches due to hardware demands

1. 全相联映像规则与变换

■ 优点：

- 块冲突概率最低；
- Cache空间利用率最高。

■ 缺点：

- 所需容量的相联存储器代价较高；
- Cache容量已经很大，相联查表速度难以提高。

1. 全相联映像规则与变换

How many bits are in the tag and offset fields?

24 bit addresses,

128K bytes of cache,

64-byte cache line (block).

A. tag=20, offset=4

B. tag=19, offset=5

C. tag=18, offset=6

D. tag=16, offset=8

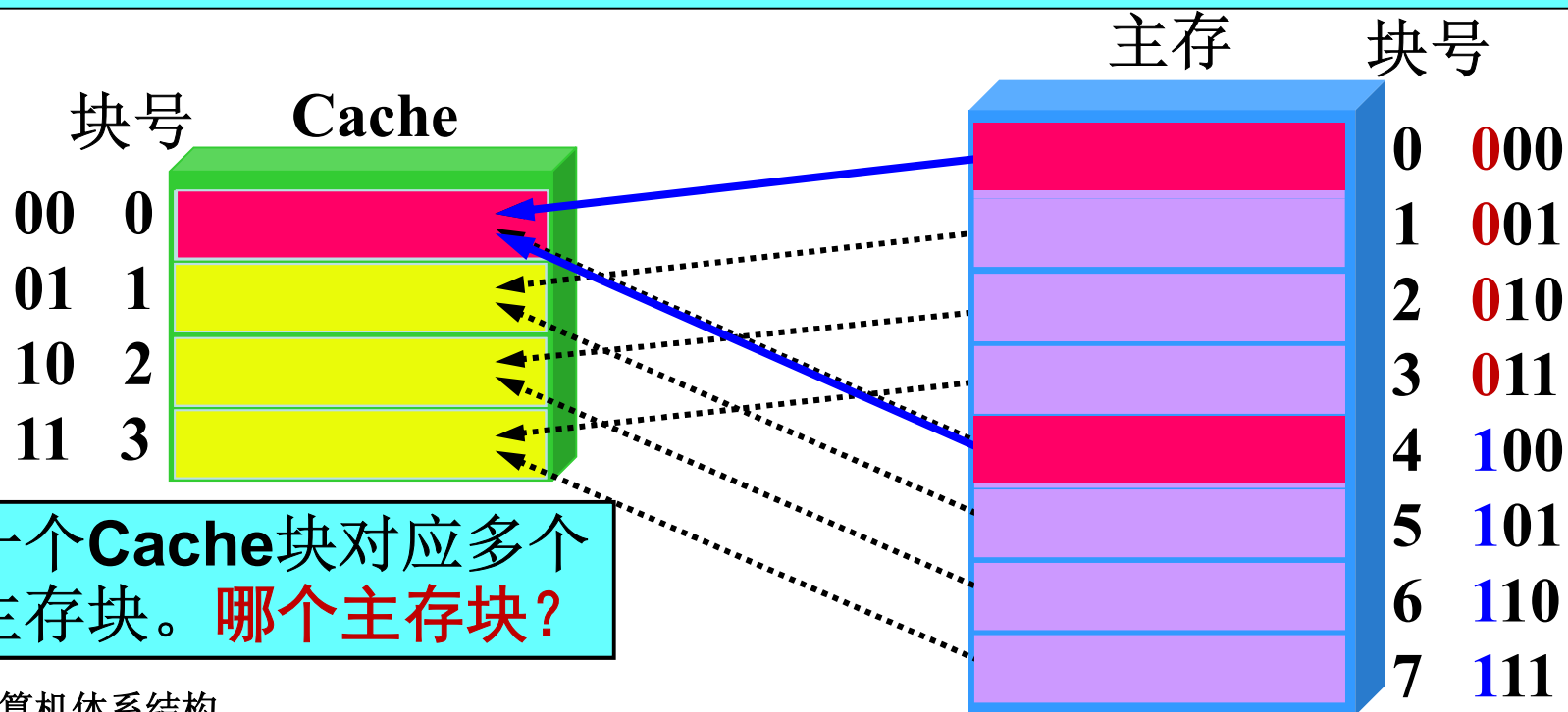
2. 直接映像规则与变换

映像规则：主存中的每一块只能装入到Cache内**唯一**一个指定的块位置。

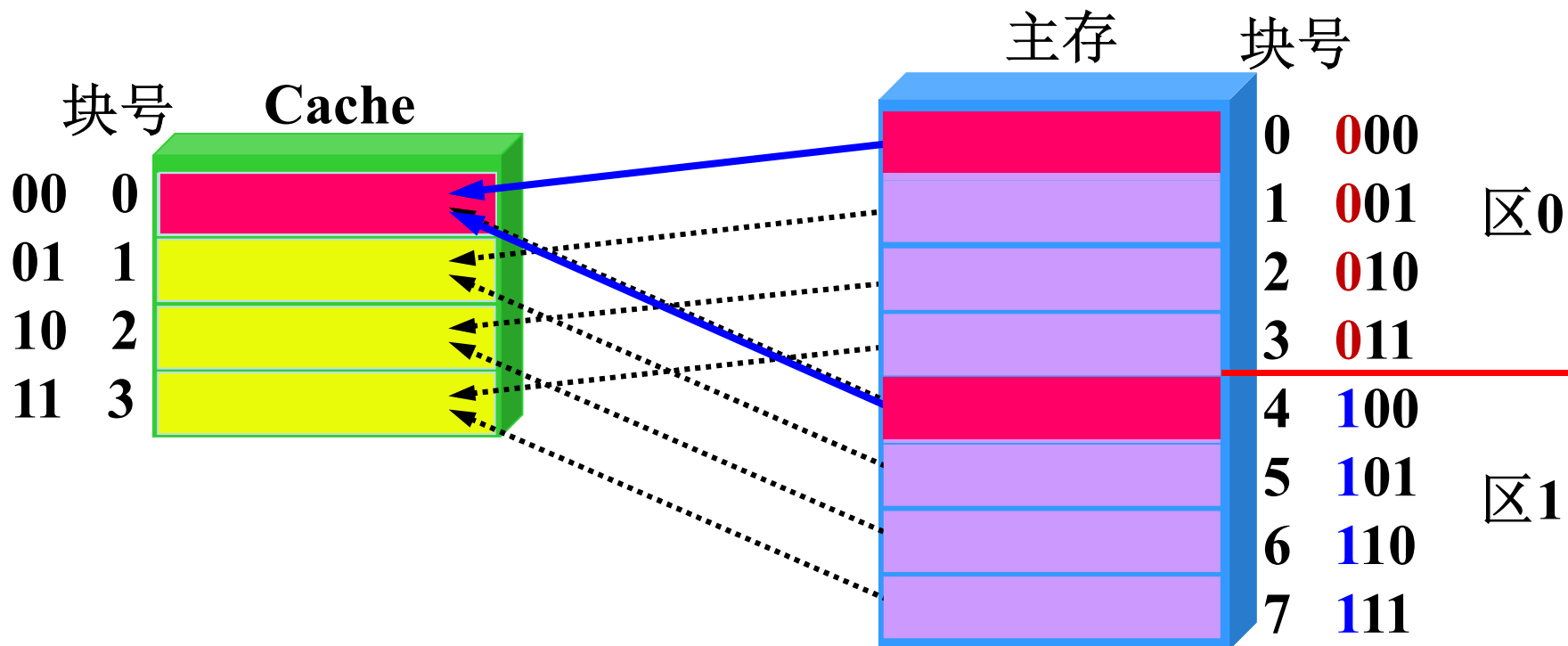
哪个位置？ 为便于地址变换，设：

Cache块号 $b = (\text{主存块号} B) \bmod (\text{Cache块数})$

则： $0 = 0 \mid 4 \bmod 4$, $1 = 1 \mid 5 \bmod 4$, $2 = 2 \mid 6 \bmod 4$, $3 = 3 \mid 7 \bmod 4$



2. 直接映像规则与变换



地址变换：

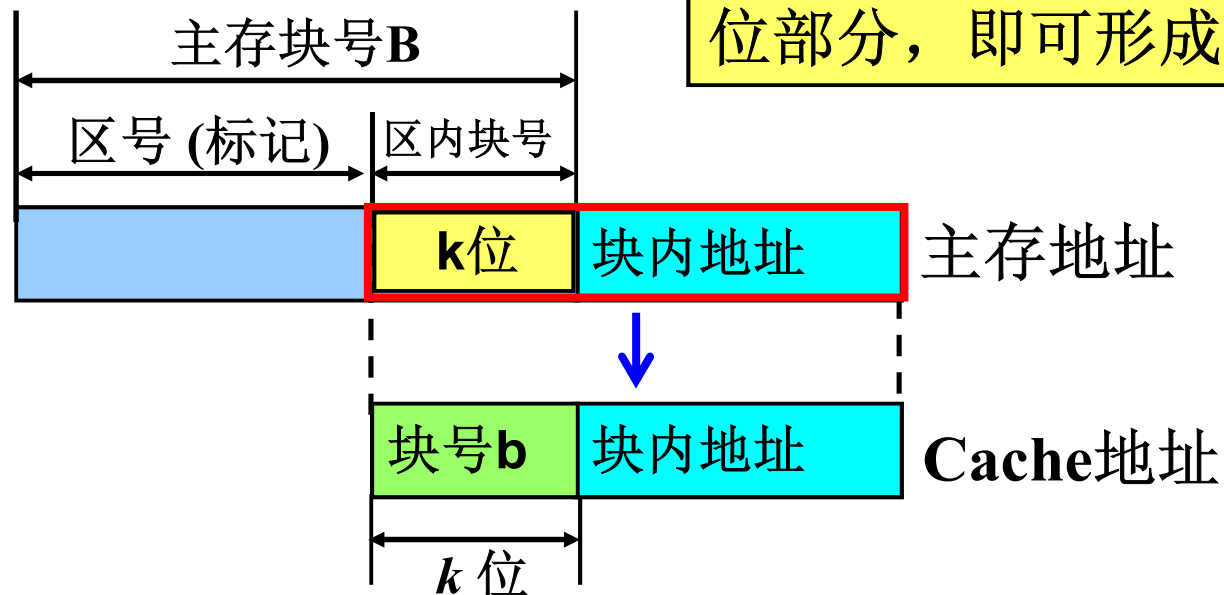
$00 = f(000 \mid 100)$ ，等于低 2 位

2. 直接映像规则与变换

地址变换:

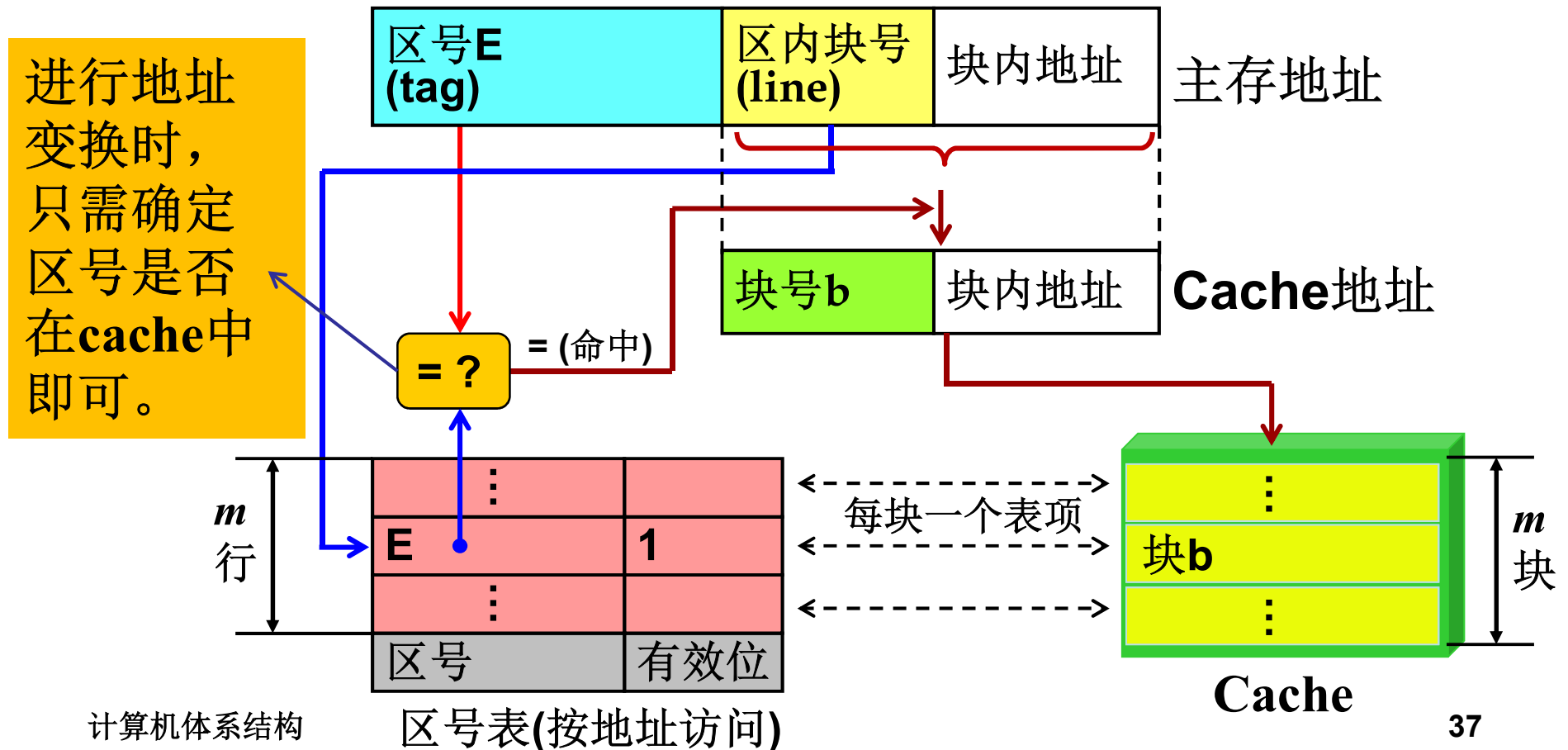
若Cache块号 $b = (\text{主存块号} B) \bmod (\text{Cache块数})$,
设Cache块数 $= 2^k$, 当用二进制数表示时, Cache块号 b 与主存块号 B 的低 k 位完全相同。

意味着: 只要从主存地址中取出低位部分, 即可形成Cache的地址。

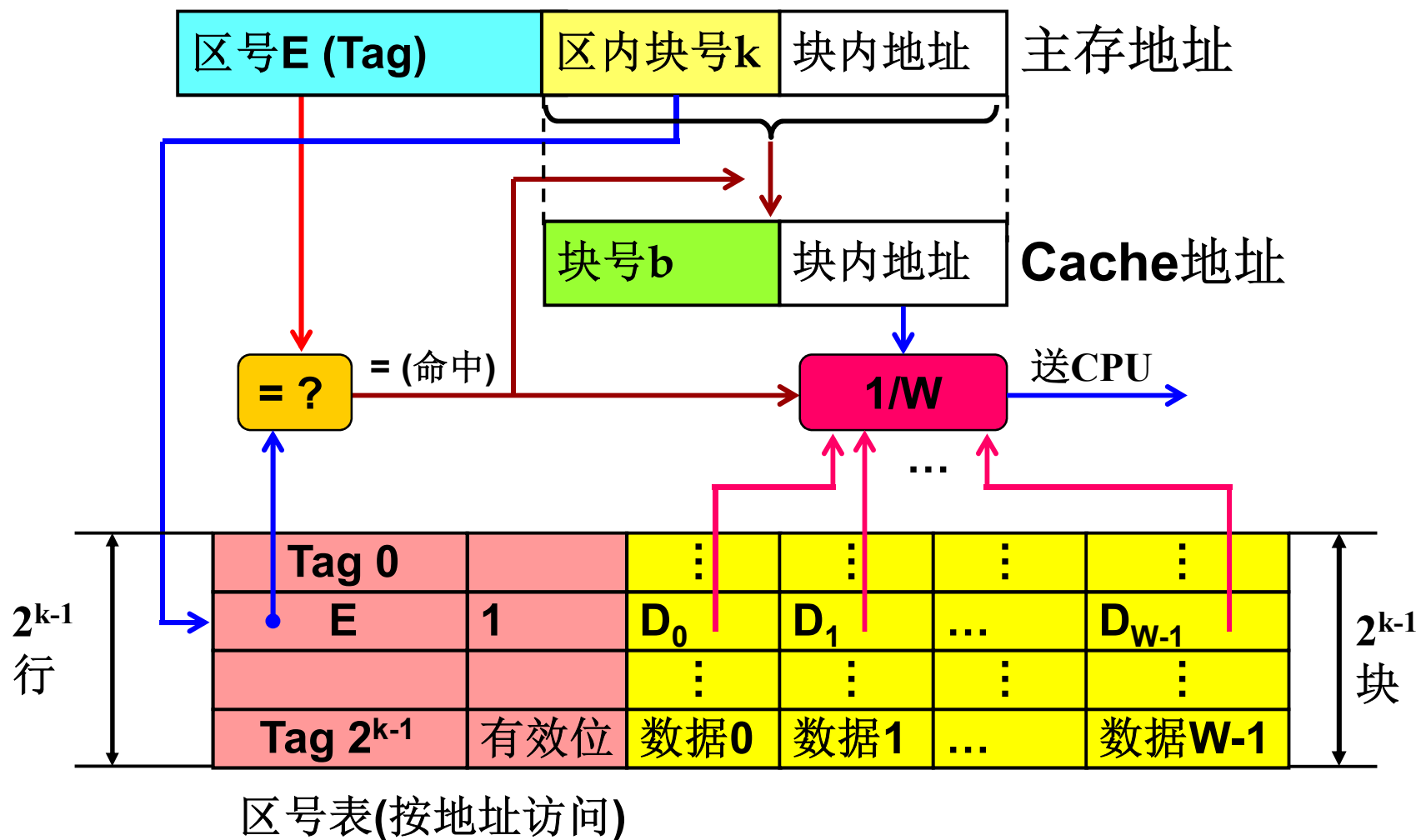


2. 直接映像规则与变换

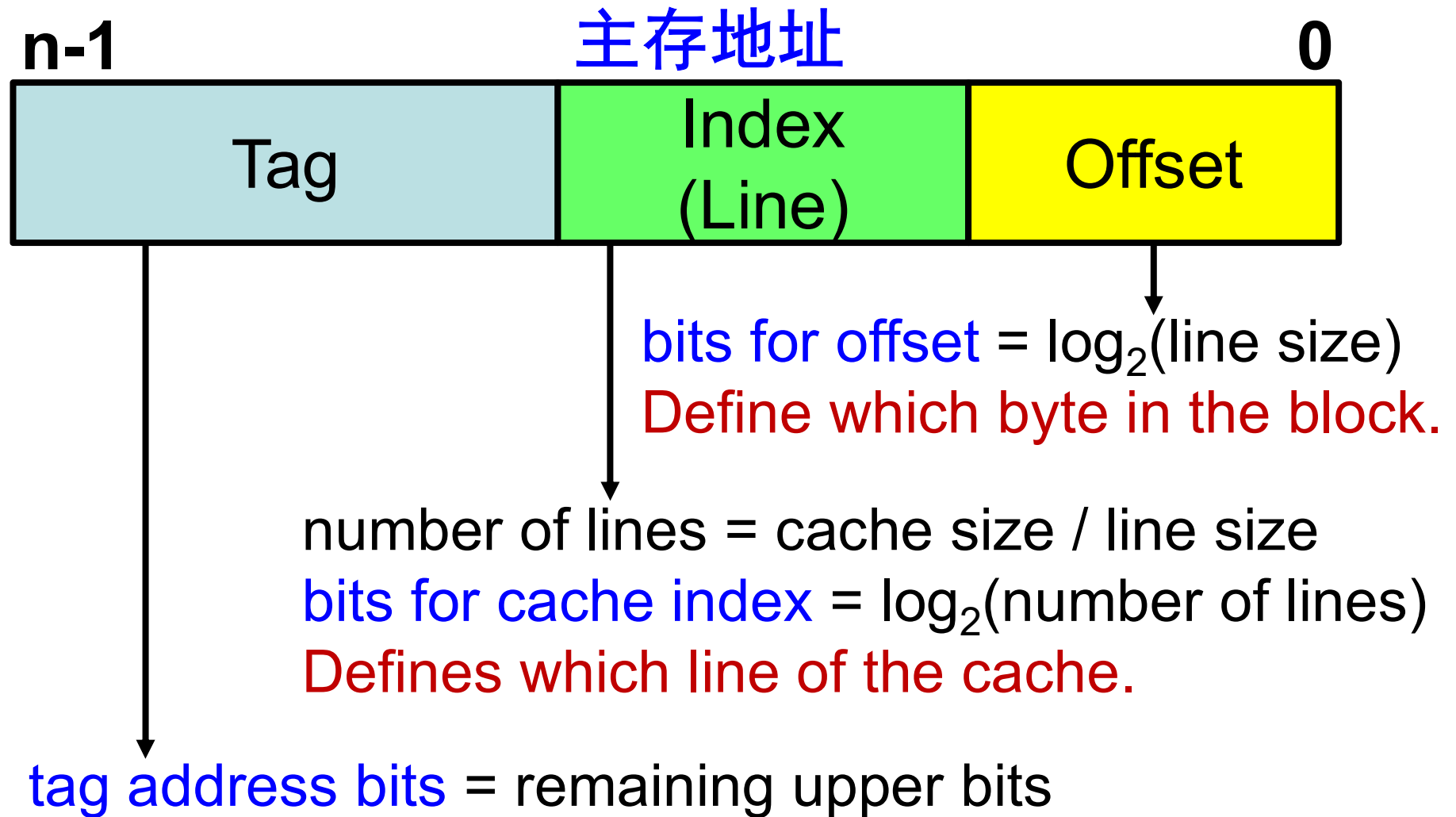
地址变换： 设置一个**按地址访问**的**区表**存储器（称之为**区号表**），存放**Cache**中每一块目前被主存中哪个区的对应块所占用。



2. 直接映像规则与变换



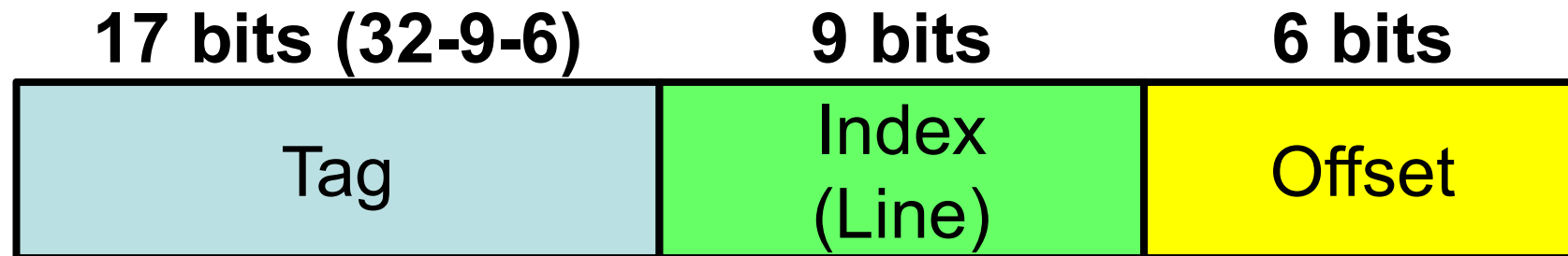
Cache Constants



Example: Direct Address

■ Assume you have

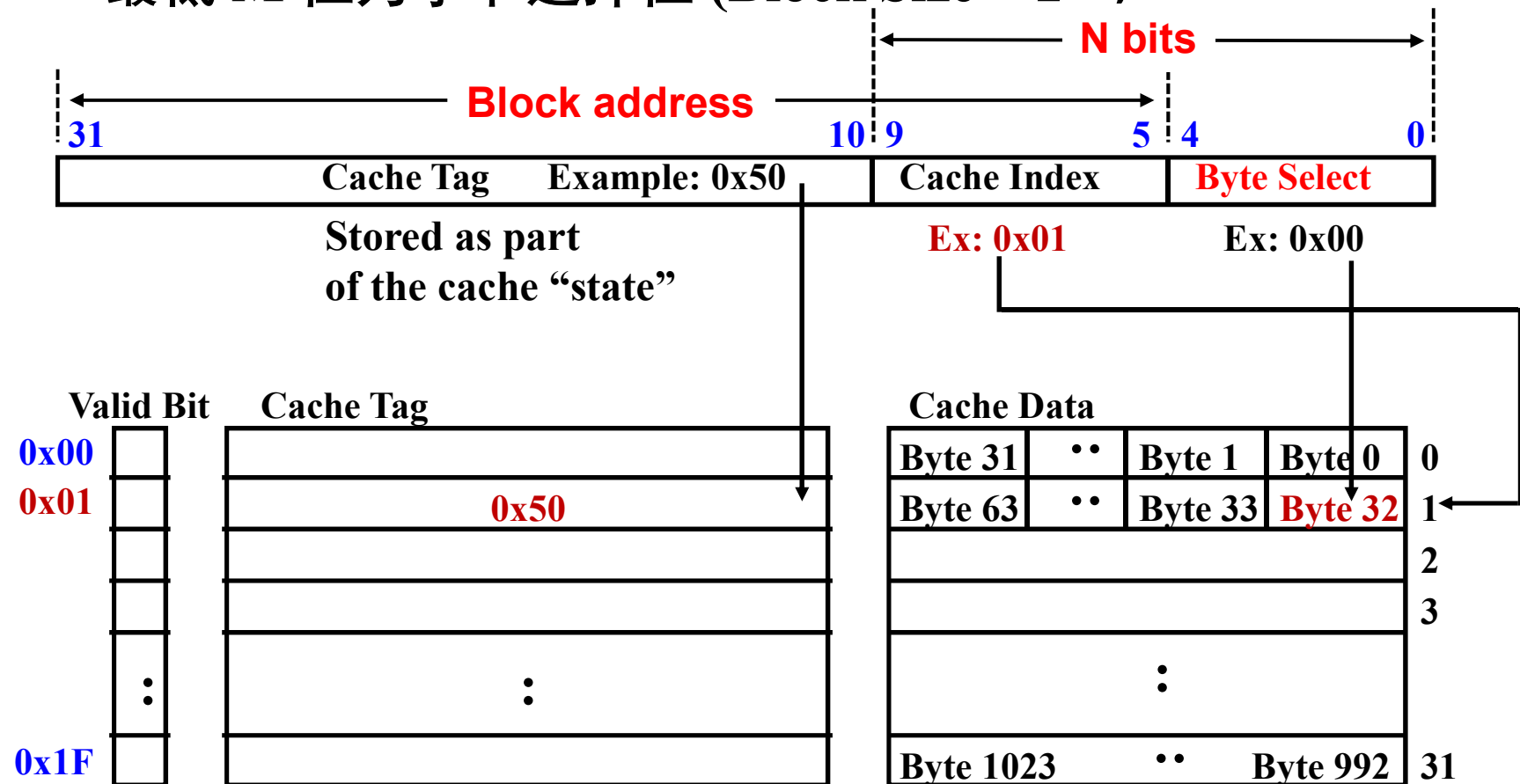
- 32 bit addresses (can address 4 GB)
- 64 byte lines (2^6) (offset is 6 bits)
- 32 KB of cache (2^{15})
- Number of lines = 32 KB / 64B = 512
- Bits to specify which line = $\log_2(512) = 9$



Example: 1 KB Direct Mapped Cache with 32 B Blocks

■ 对于容量为 2^N 字节的 Cache:

- 最高 (32-N) 位部分为 Cache Tag
- 最低 M 位为字节选择位 (Block Size = 2^M)



2. 直接映像规则与变换

■ 优点：

- 所需硬件简单，成本较低；
- 访问Cache可与访问区号表、比较区号等操作同时进行，节省了地址变换时间。

■ 缺点：

- 块冲突概率很高；
- Cache空间利用率很低。因此已经很少使用。

■ 提高Cache速度的一种方法：

- 把区号存储器与Cache合并成一个存储器。

2. 直接映像规则与变换

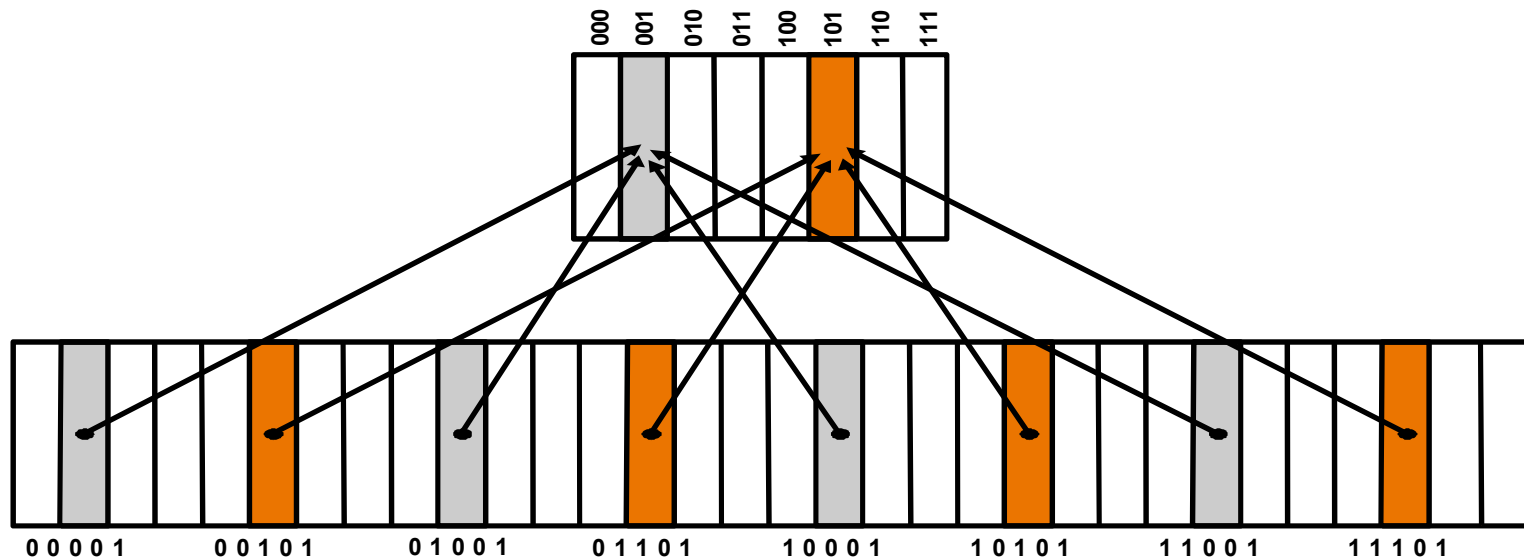
How many bits are in the tag, line(index) and offset fields?

**24 bit addresses,
64K bytes of cache,
16-byte cache lines.**

- A. tag=4, line=16, offset=4**
- B. tag=4, line=14, offset=6**
- C. tag=8, line=12, offset=4**
- D. tag=6, line=12, offset=6**

Direct-mapped cache Example

- Cache: 8 blocks, 1 word/block.
- Memory: 32 blocks.
- Block Placement :
 - Address mapping: modulo number of blocks.
 - For each item of data at the lower level, there is exactly one location in cache where it might be.



Tags and Valid Bits: Block Finding

- **How do we know which particular block is stored in a cache location?**
 - Store block address as well as the data
 - Actually, only need the high-order bits
 - Called the **tag**
- **What if there is no data in a location?**
 - Valid bit: 1 = present, 0 = not present
 - Initially 0

Direct-mapped cache Example

■ Initial state

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Direct-mapped cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-mapped cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-mapped cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-mapped cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

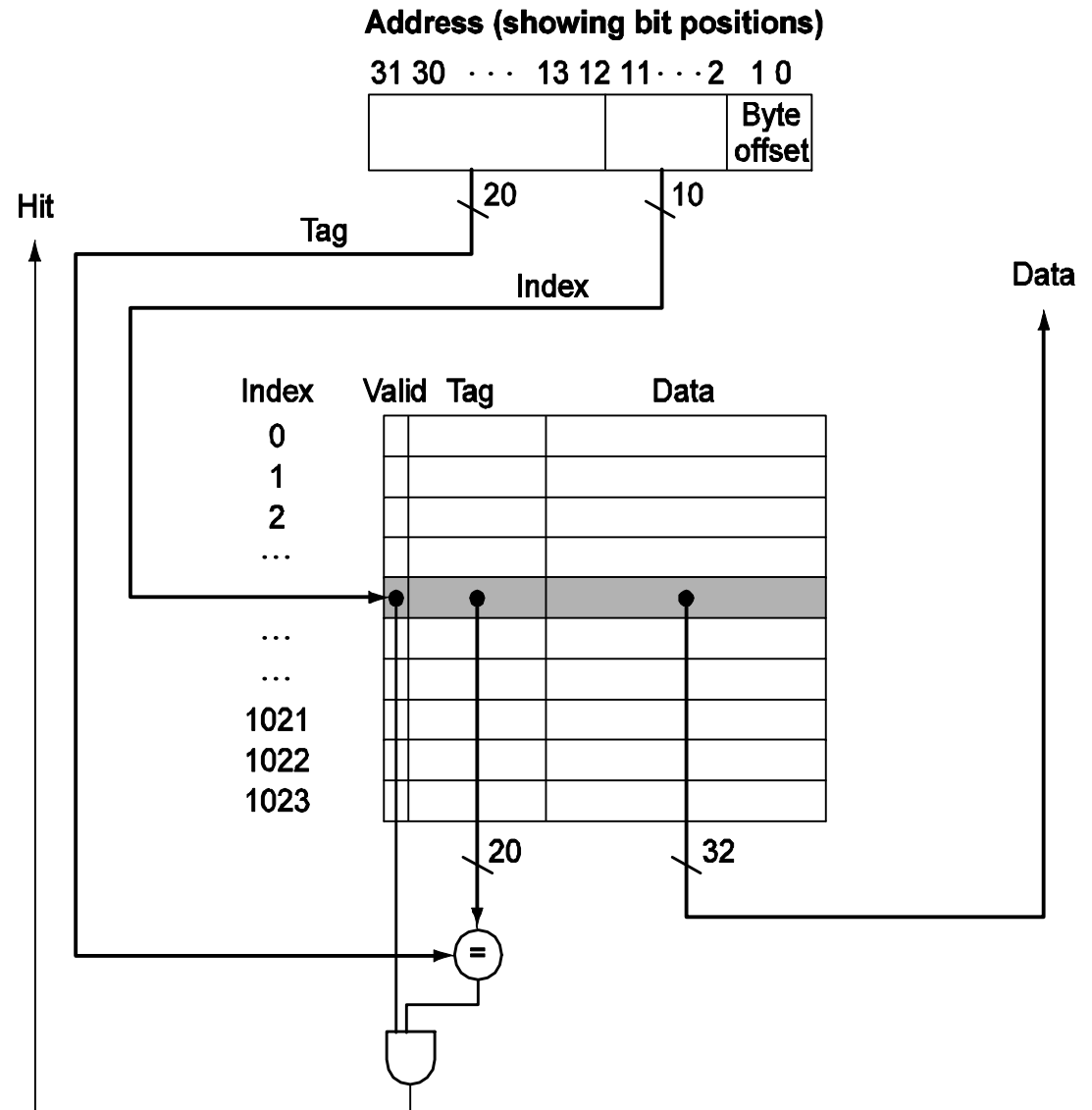
Direct-mapped cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11->10	Mem[10010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

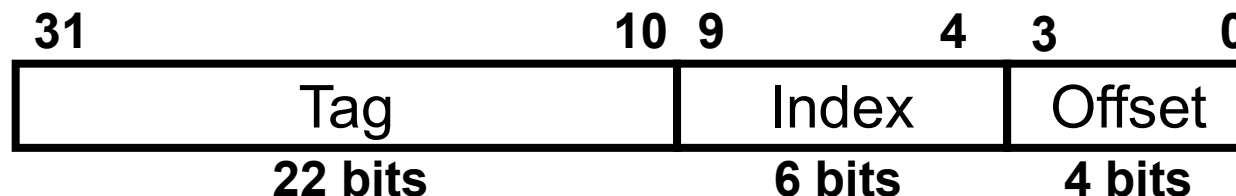
Direct-mapped cache Address Subdivision

- 1K words,
1-word block:
 - Cache index:
lower 10 bits
 - Cache tag:
upper 20 bits
 - Valid bit (When
start up, valid is 0)



Example: Larger Block Size

- 64 blocks, 16 bytes/block. To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor = 75$
- Block number = $75 \bmod 64 = 11$
- $1200 = 10010110000_2 / 10000_2 \Rightarrow 1001011_2$
- $1001011_2 \Rightarrow 001011_2$



3. 组相联映像规则与变换

- 组相联映象和变换方式有很多种变型，它们各有不同的特点。
- 下面，介绍一种常用的组相联映象及变换方式（**书上的**），称为：

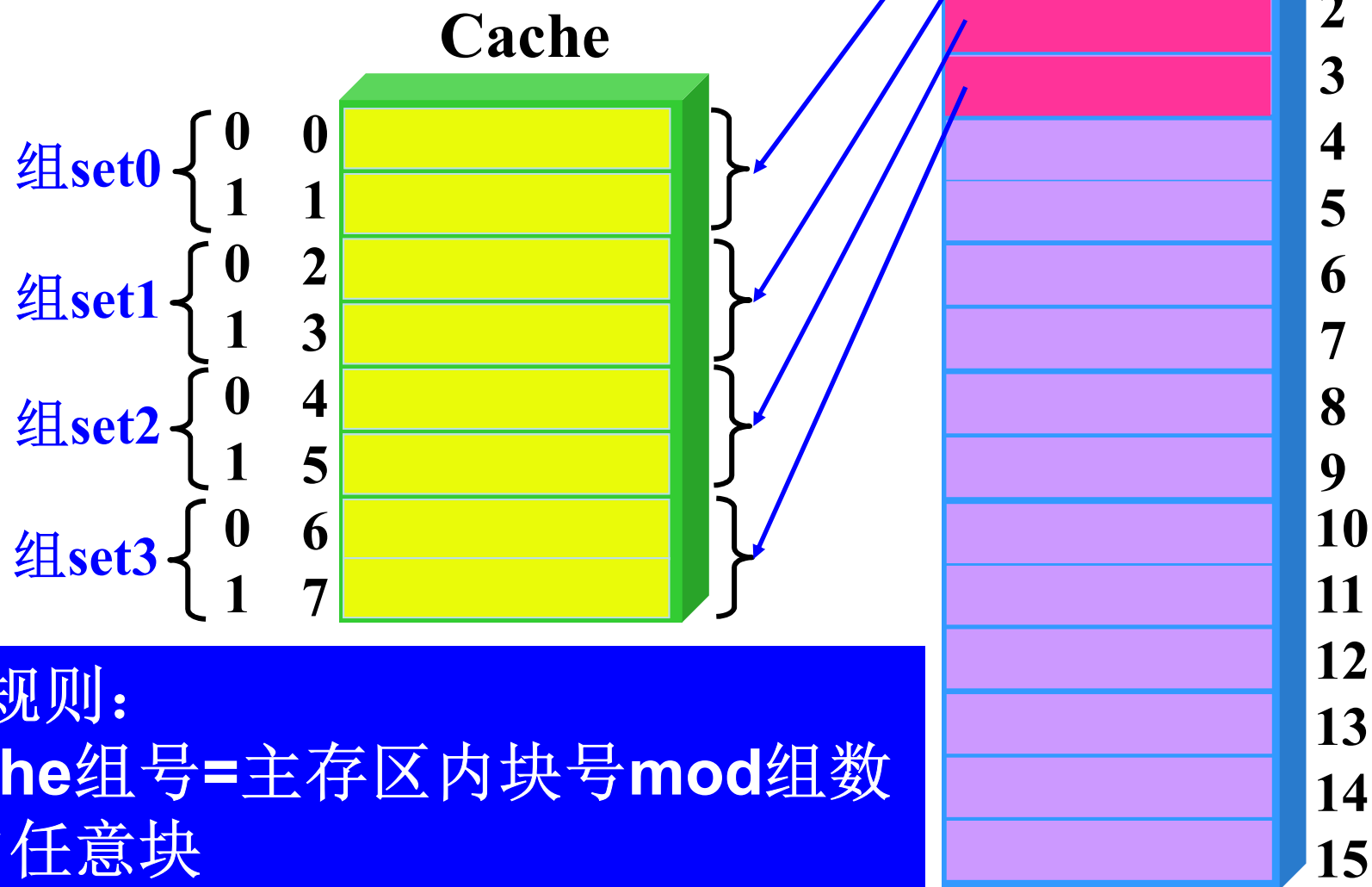
位选择组相联映象及变换方式 或

位选择算法组相联映象及变换方式 或

N-way 组相联 (N-way Set Associative Cache)

位选择组相联映像规则与变换

将 **Cache** 分组(set), 主存不分组。



映像规则:

- **Cache组号**=主存区内块号**mod**组数
- 组内任意块

位选择组相联映像规则与变换

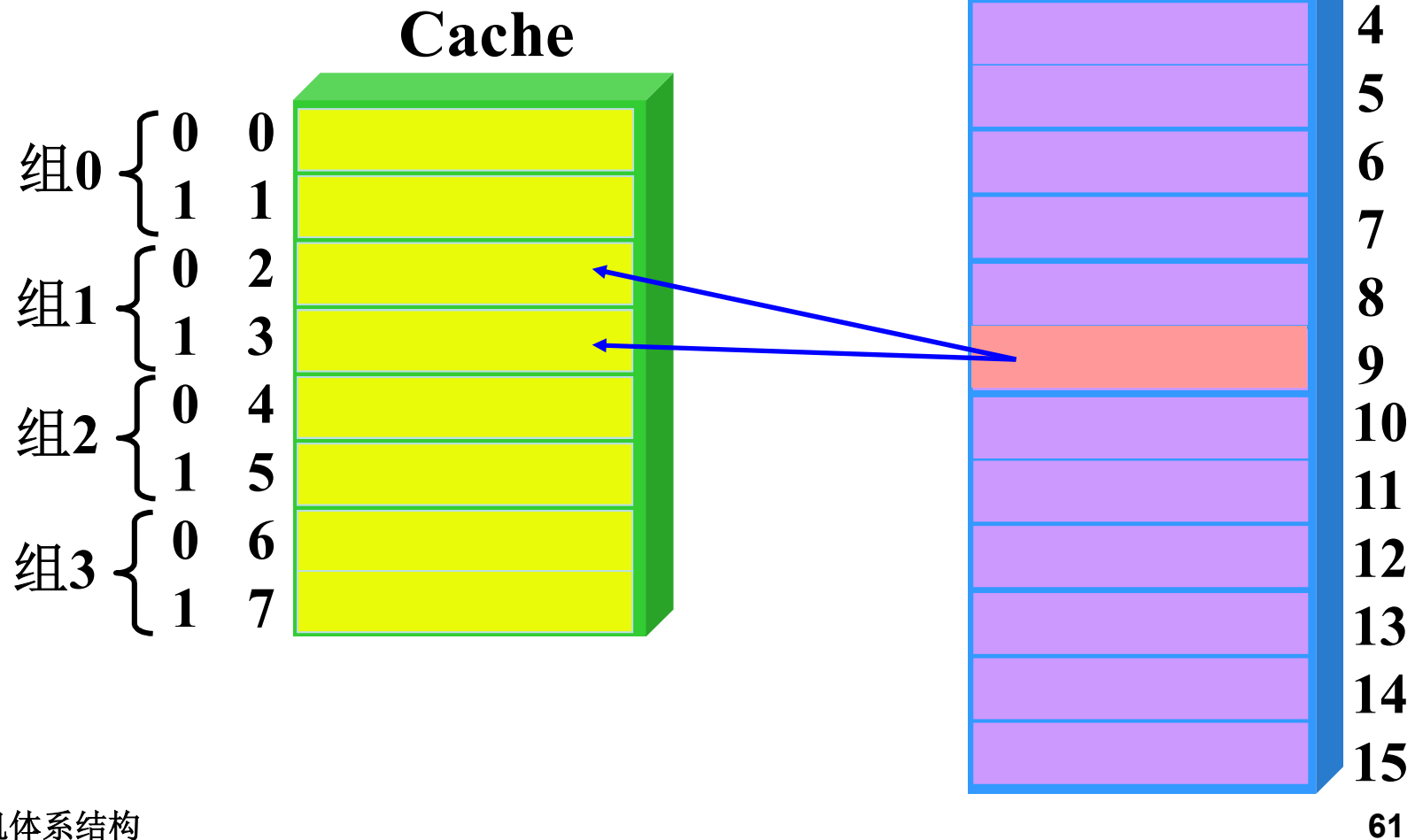
■ 映像规则：

- 主存块到 Cache 组 set 之间采用直接映像方式（即 1 个主存块只能映像到 1 个 Cache 组）；所对应的Cache组号=主存块号 mod Cache组数。
- 在对应的组内部采用全相联映像方式（即 1 个主存块可以放入指定 Cache 组内任何 1 个块位置——组内随便放）。

Set 1 = 9 Mod 4

主存块9可以放入Cache组 1
的块 0 和块 1 中。

如何知道放入哪个Cache块？

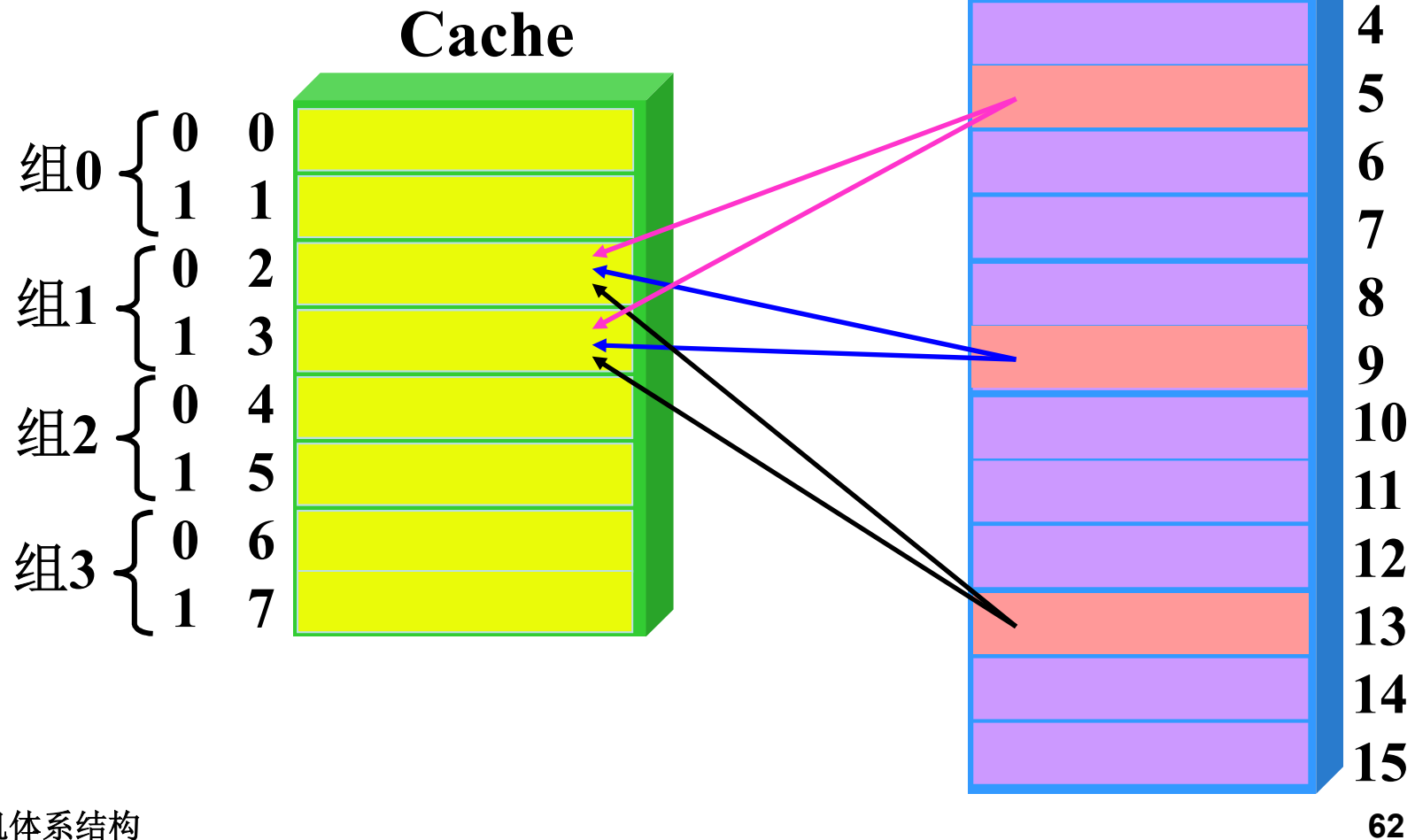


Set 1 = 5|9|13 Mod 4

主存块5、9、13 都可以放入

Cache组 1 的块 0 和块 1 中。

如何知道放入的是哪个主存块？



位选择组相联映像规则与变换

- 将Cache分成G组 ($G=2^g$)，每个组S块 ($S=2^s$)

Cache地址

组号(g位) Set	组内块号(s位)	块内地址
------------	----------	------

组内块数 S 称为**相联度**。

每组有 S 块的组相联称为 **S 路组相联**。

- 将主存按同样大小划分成块，所对应的Cache组号=主存块号 mod Cache组数

主存地址

标记 (Tag)	组号(g位) Set	块内地址
----------	------------	------

地址变换：

根据主存地址中的**标记+组号**找到所对应的Cache组内块号

位选择组相联映像规则与变换

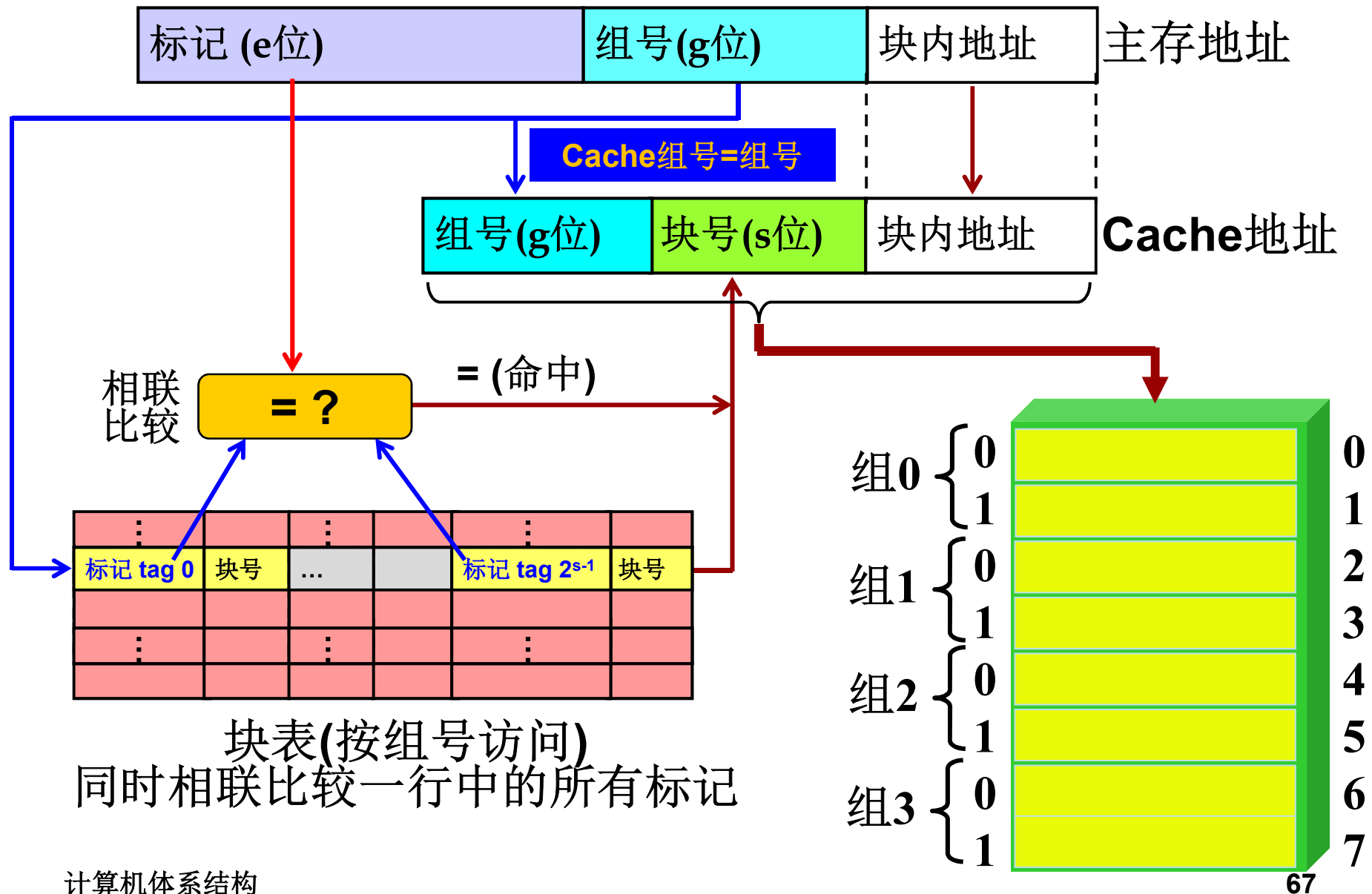
地址变换：根据组号按地址访问块表，取得目录表地址，按标记访问目录表，取得Cache块号。

块表

⋮		⋮		⋮		共 $G=2^g$ 行 按地址(组号)访问
标记 tag 0	组内块号	...		标记 tag 2^s-1	组内块号	
⋮		⋮		⋮		
← 目录表，每组一行。 →						

每行共 $S=2^s$ 项（ 2^s 路）。按内容访问，同时比较。

位选择组相联映像规则与变换



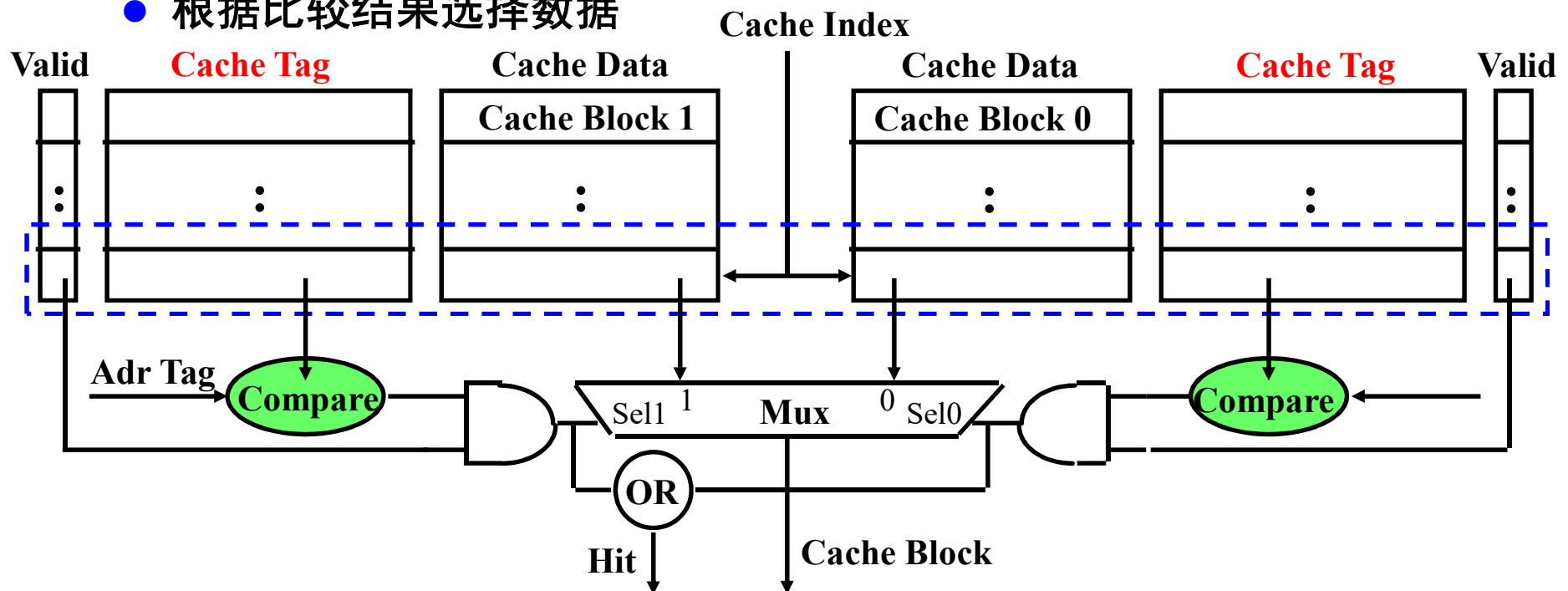
N-way Set Associative Cache

■ **N-way set associative:** 每一个cache set 对应 N 个cache entries

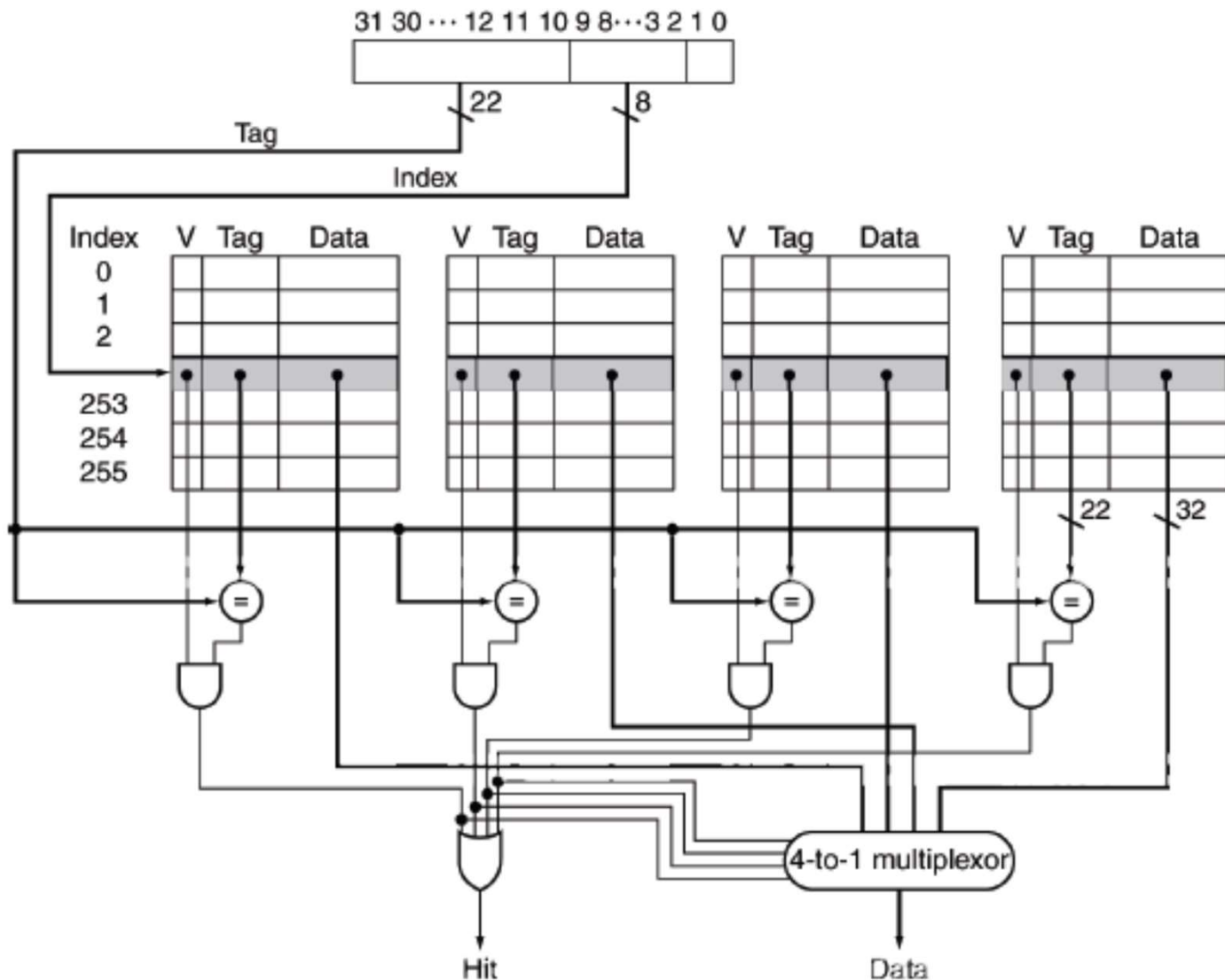
- 这 N 个 cache 项并行操作

■ **Example: Two-way set associative cache**

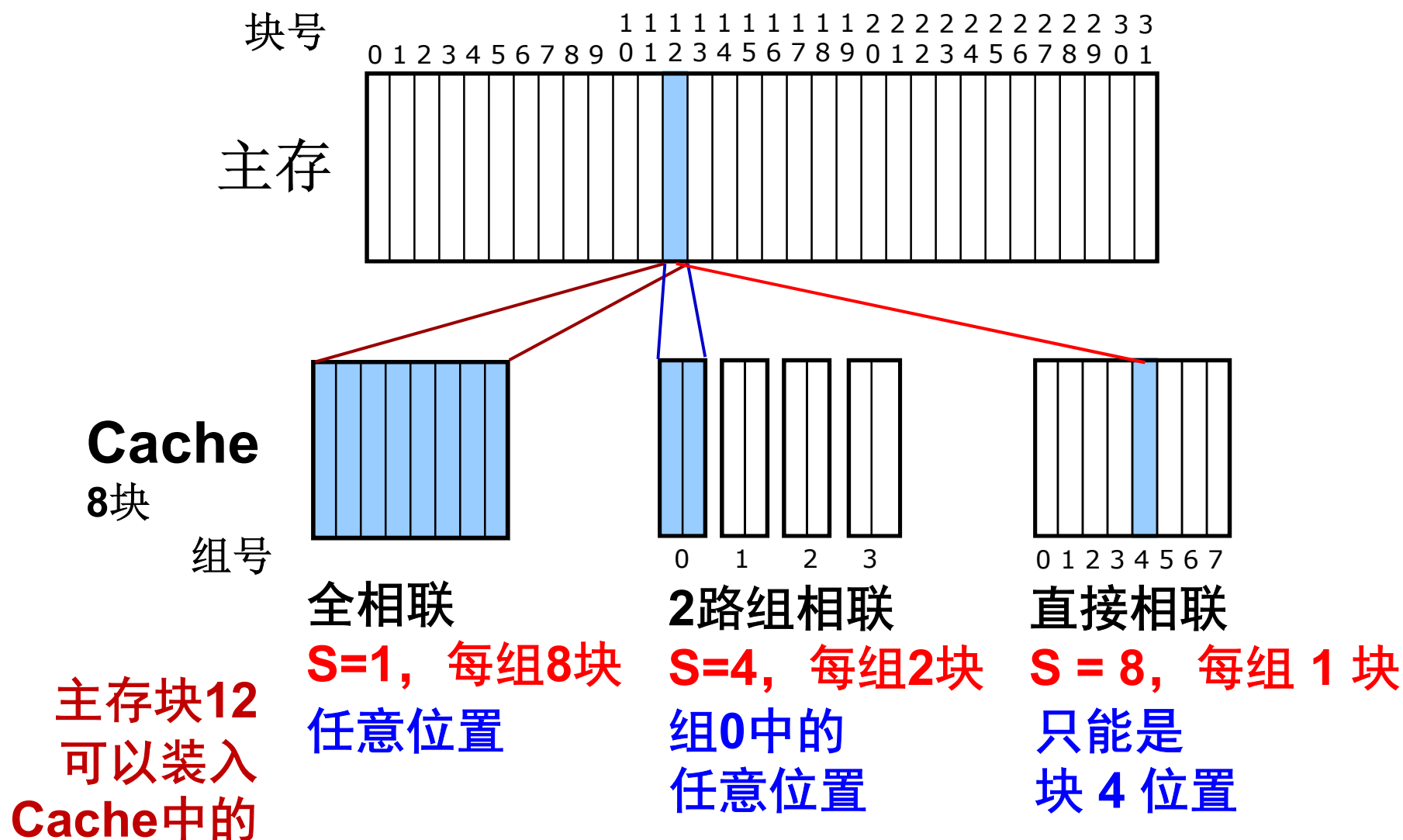
- Cache index (Set) 选择 cache 中的一组
- 这一组中的 2 块对应的 Tags 与输入的地址同时比较
- 根据比较结果选择数据



Example: 4-way Set Associative Cache



组相联映像规则与变换



Associativity with 8-block cache

0	tag	Data
1	tag	Data
2	tag	Data
3	tag	Data
4	tag	Data
5	tag	Data
6	tag	Data
7	tag	Data

***One Way Associative
(Direct Mapped)***

0	tag	Data	tag	Data
1	tag	Data	tag	Data
2	tag	Data	tag	Data
3	tag	Data	tag	Data

Two Way Associative

0	tag	Data	tag	Data	tag	Data	tag	Data
1	tag	Data	tag	Data	tag	Data	tag	Data

Four Way Associative

tag	Data	tag	Data	tag	Data	tag	Data	tag	Data	tag	Data	tag	Data	tag	Data
-----	------	-----	------	-----	------	-----	------	-----	------	-----	------	-----	------	-----	------

***Eight Way Associative
(Fully Associative)***

N-Way组相联

- **N-Way组相联：**如果每组由N个块构成，cache的块数为M，则cache的组数G为 M/N 。
- 不同相联度下的路数和组数

	路数	组数
全相联	M	1
直接相联	1	M
其他组相联	$1 < N < M$	$1 < G < M$

- 相联度越高，cache空间利用率就越高，块冲突概率就越小，失效率就越低
- N值越大，失效率就越低，但Cache的实现就越复杂，代价越大
- 现代大多数计算机都采用直接映象，两路或四路组相联。

位选择组相联映像规则与变换

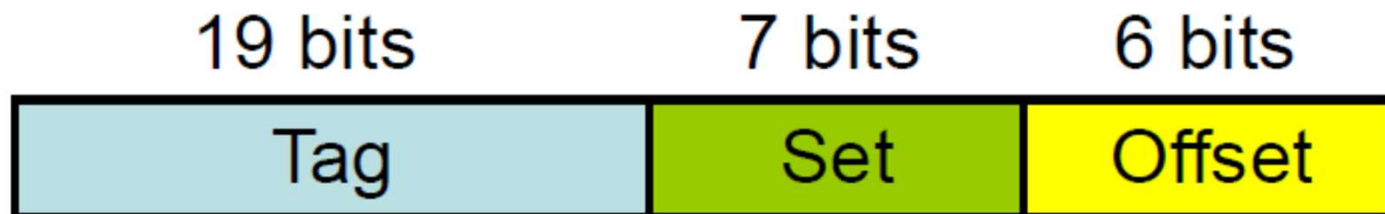
- 例：某计算机的Cache共有16块，采用2路组相联映像方式（即每组2块）。每个主存块大小为32字节，按字节编址。主存129号单元所在主存块应装入到的Cache组号是多少？
- 解：由于每个主存块大小为32字节，按字节编址。根据计算主存块号的公式，主存块号=
$$\lfloor \text{主存地址} / \text{块大小} \rfloor = \lfloor \frac{129}{32} \rfloor = 4$$
，所以主存129号单元所在的主存块应为第4块。若Cache共有16块，采用2路组相联映像方式，可分为8组。根据组相联映像的映像关系，主存第4块转入Cache第4组。

Set Associative Mapping

- **When the processor wants an address, it indexes to the set and then searches the tag fields of all lines in the set for the desired address**
 - **$n = \text{cache size} / \text{line size} = \text{number of lines}$**
 - **$b = \log_2(\text{line size}) = \text{bit for offset}$**
 - **$w = \text{number of lines} / \text{set}$**
 - **$s = n / w = \text{number of sets}$**

Example Set Associative

- Assume you have: 32 bit addresses, 32 KB of cache of 64-byte lines, 4-way set associative.
 - Number of lines = $32 \text{ KB} / 64\text{B} = 512$
 - Number of sets = $512 / 4 = 128$
 - Set bits = $\log_2(128) = 7$
 - Offset bits = $\log_2(64) = 6$



组相联映像规则与变换

■ 优点：

- 块冲突概率比直接映像低得多；
- Cache空间利用率也比直接映像提高；
- 实现成本比全相联映像要低得多；
- 性能接近于全相联映像。

■ 缺点：

- 实现难度和造价要比直接映像方式高。

Calculating Bits in Cache

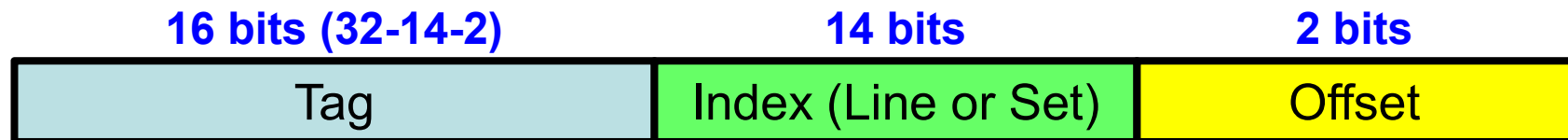
- **Ex1:** How many total bits are needed for a **direct-mapped cache** with **64 Kbytes** of data and **one word** blocks, assuming a **32-bit address**?

#block = 64 Kbytes = 16 K words = 2^{14} words = 2^{14} blocks

block size = 4 bytes \Rightarrow offset size = 2 bits

#sets = **#blocks/1** = $2^{14} \Rightarrow$ **index size** = 14 bits

tag size = address size - index size - offset size = $32 - 14 - 2 = 16$ bits



Cache大小:

bits/block = tag bits + valid bit + data bits + = $16 + 1 + 32 = 49$ bits

bits in cache = **#blocks** x **bits/block** = $2^{14} \times 49 = 98$ Kbytes

Calculating Bits in Cache

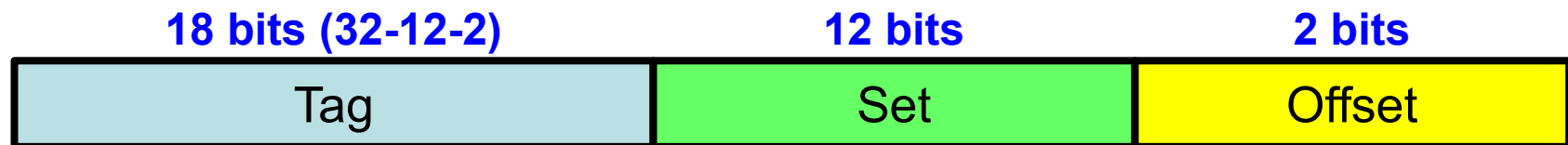
- **Ex2:** How many total bits would be needed for a **4-way set associative** cache to store the same amount of data as in Ex1?

#block = 64 Kbytes = 16 K words = 2^{14} words = 2^{14} blocks

block size = 4 bytes \Rightarrow offset size = 2 bits

#sets = $\#blocks/4 = (2^{14})/4 = 2^{12} \Rightarrow$ **index size** = 12 bits

tag size = address size - index size - offset = 32 - 12 - 2 = 18 bits



Cache大小: Increase associativity \Rightarrow increase bits in cache

bits/block = tag bits + valid bit + data bits = 18 + 1 + 32 = 51 bits

bits in cache = $\#blocks \times bits/block = 2^{14} \times 51 = 102 \text{ Kbytes}$

Calculating Bits in Cache

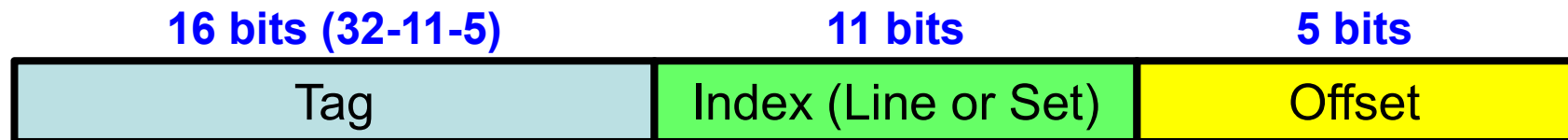
- Ex3: How many total bits are needed for a **direct-mapped** cache with **64 Kbytes** of data and **8 word** blocks, assuming a **32-bit address**?

#block = 64 Kbytes = 16 K words = 2^{14} words = $(2^{14})/8 = 2^{11}$ blocks

block size = 8 * 4 bytes \Rightarrow offset size = 5 bits

#sets = **#blocks/1** = $2^{11} \Rightarrow$ **index size** = 11 bits

tag size = address size - index size - offset size = 32 - 11 - 5 = 16 bits



Cache大小: Increase block size \Rightarrow decrease bits in cache

bits/block = tag bits + valid bit + data bits = 16 + 1 + 8*32 = 273 bits

bits in cache = #blocks x bits/block = $2^{11} \times 273 = 68.25$ Kbytes

Associativity Example

■ Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

time

$$\text{命中率} = 0/5 = 0$$

Associativity Example

■ 2-way set associative


time ↓

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	Mem[0]	Mem[8]		
6	0	miss	Mem[0]	Mem[6]		
8	0	miss	Mem[8]	Mem[6]		

$$\text{命中率} = 1/5 = 20\%$$

Associativity Example

■ Fully associative



Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	Mem[8]	Mem[6]	

time

$$\text{命中率} = 2/5 = 40\%$$

Cache替换算法与实现

- **Cache**通常使用组相联或直接映像，而不采用全相联映像。
- 当所要访问的块不在**Cache**中时，则发生**块失效**。
- 当所能装入的**Cache**块都已被装满时，则出现**块冲突**，必须进行**块替换**。
- **替换算法**：确定被替换的主存块。

Cache替换算法与实现

■ 典型替换算法：

1. **随机算法**：随机选取一块进行替换。
2. **FIFO算法**：选取最早装入的块进行替换。
3. **LRU算法**：选取近期被CPU访问次数最少（最久未使用）的主存块进行替换。**优点：**
比较正确地反映了程序的局部性，失效率低。
→ **常用**

Cache替换算法与实现

表 LRU和随机算法的失效率的比较

Cache 容量	相联度					
	2路		4路		8路	
	LRU	随机	LRU	随机	LRU	随机
16KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

测试条件：块大小=16字节，地址流=VAX流

从表中数据可以看出：

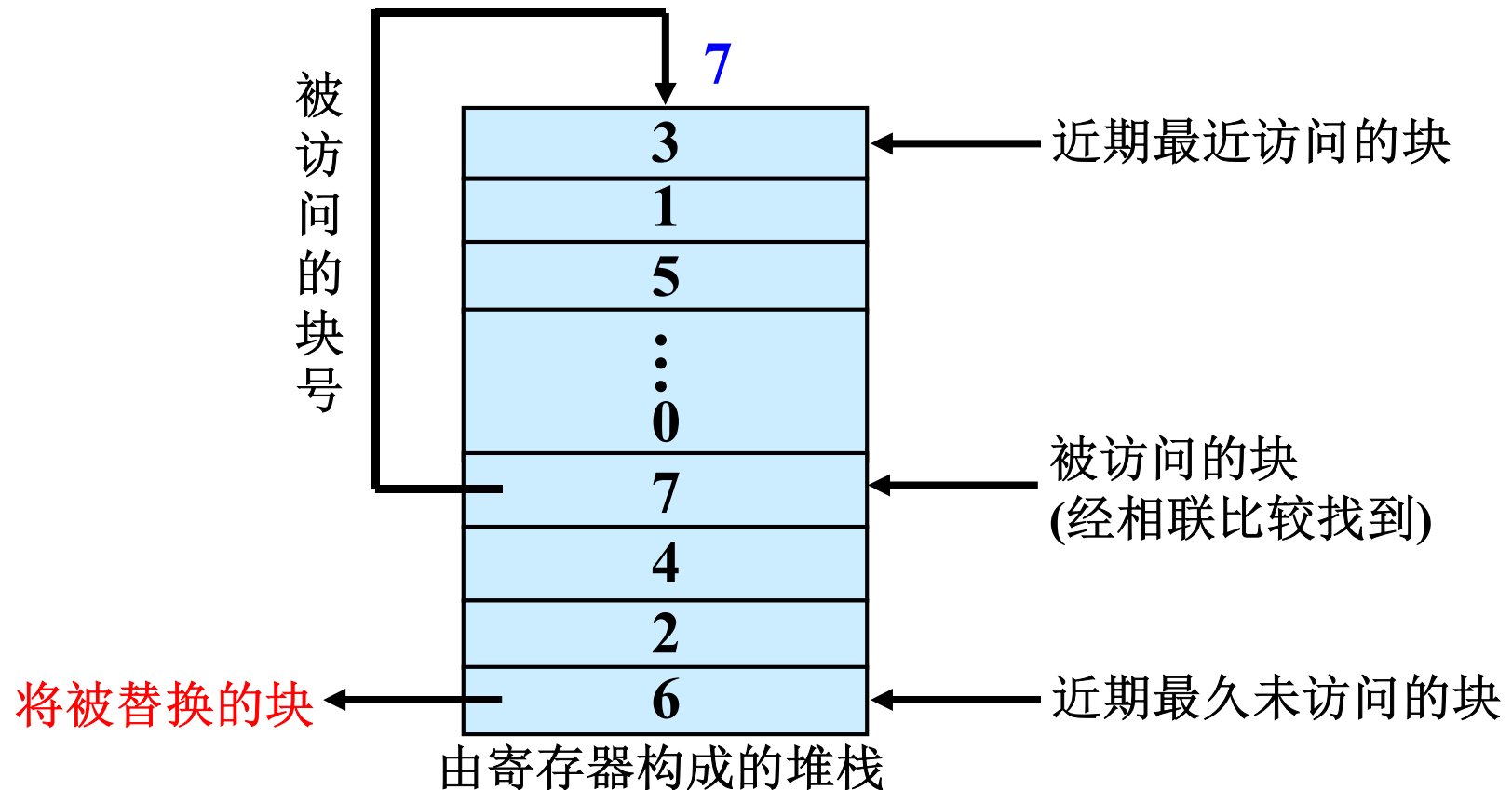
- LRU算法的失效率低于随机算法。
- 随着Cache容量的增加，失效率降低。
- 当Cache容量较大时，LRU算法和随机算法的失效率几乎无差别。

Cache替换算法与实现

- 实现：全部用硬件
- 两种方法：
 - 堆栈法：使用硬堆栈
 - 比较对法：使用逻辑电路、触发器等

堆栈法

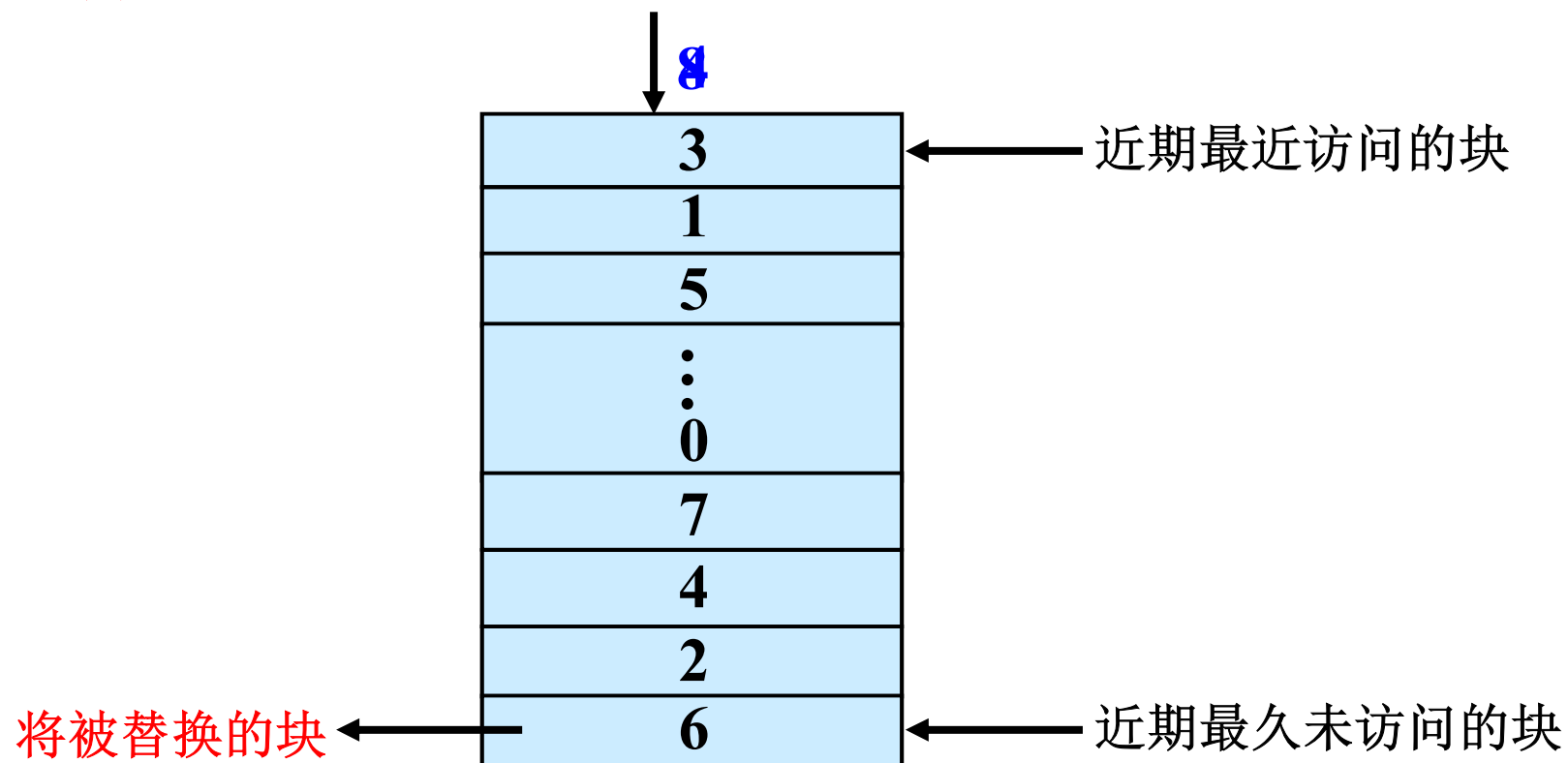
用硬件堆栈实现。



要访问的主存块7 已在**Cache**中时的情况

堆栈法

用硬件堆栈实现。



由寄存器构成的堆栈

要访问的主存块8不在**Cache**中时的情况

比较对法

- 用触发器（硬联逻辑）实现。
- 基本思路：
 - 让各个块**成对**组合，用一个触发器的状态来表示该比较对内两块访问的远近次序，再经门电路就可找到 LRU 块。

比较对法

- 例如：有A、B、C 三块，之间的组合共有 $C_3^2 = 3$ 组合：AB、AC、BC
- 各对内块的访问顺序分别用两态“触发器”表示。
 - T_{AB} 为 “1”：表示A比B更近被访问过；
 - T_{AB} 为 “0”，表示B比A更近被访问过。

比较对法

- 如果C为最久未被访问过的块，三个块的排列顺序有两种可能：

块A、块B、块C

块B、块A、块C

- 根据逻辑关系，很容易写出块C最久没有被访问过表达式：

$$C_{LRU} = T_{AB} \cdot T_{AC} \cdot T_{BC} + \overline{T_{AB}} \cdot T_{AC} \cdot T_{BC} = T_{AC} \cdot T_{BC}$$

$$B_{LRU} = T_{AB} \cdot \overline{T_{BC}}$$

$$A_{LRU} = \overline{T_{AB}} \cdot \overline{T_{AC}}$$

比较对法

3个块时:

- 3个触发器,
- 3个与门,
- 每个与门需要两个输入端。

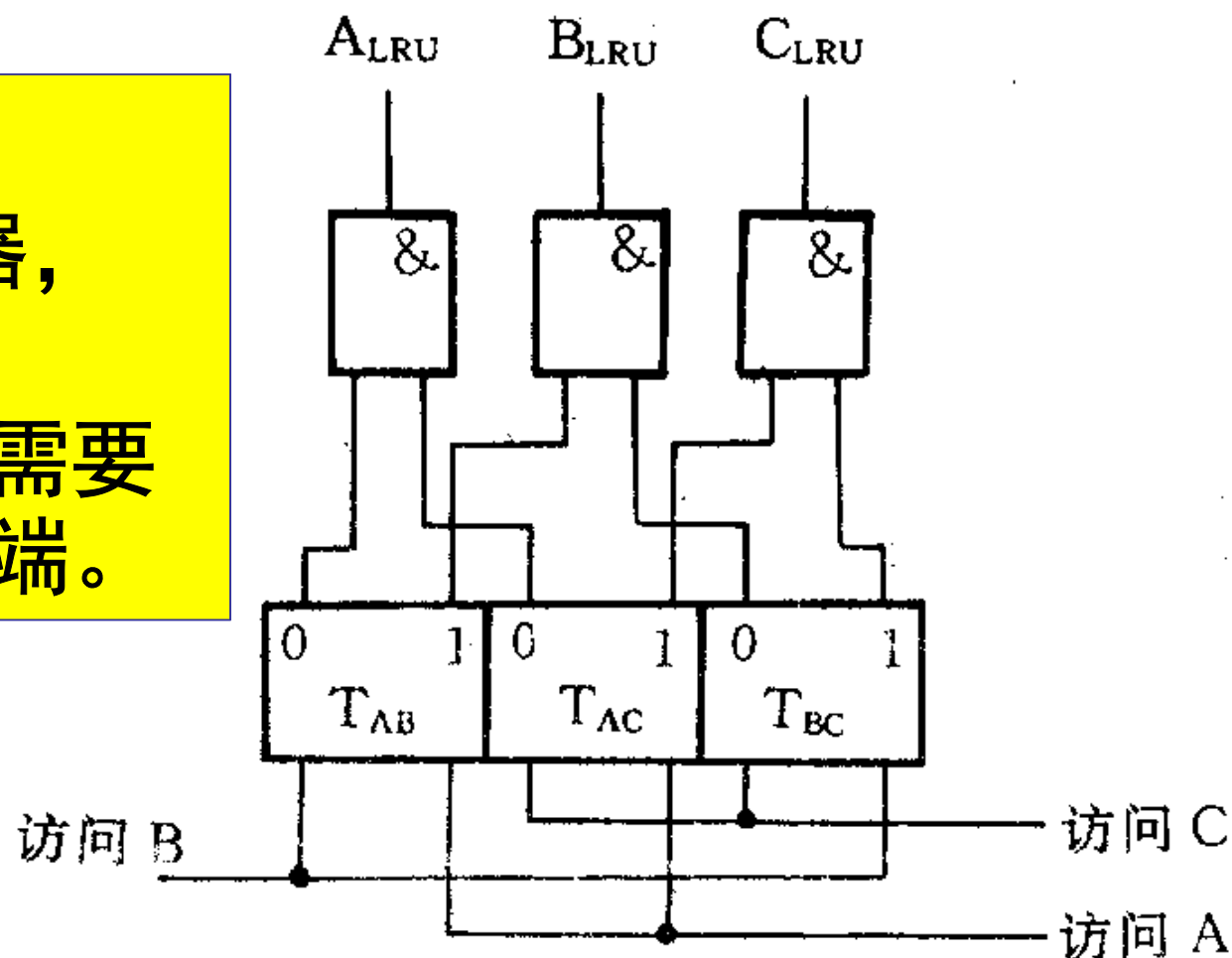


图 用比较对法实现LRU算法

比较对法

表 比较对触发器数、门数、门的输入端数与块数的关系

块数	3	4	8	16	64	256	...	p
比较对触发器数	3	6	28	120	2016	32 640	...	$p(p-1)/2$
门数	3	4	8	16	64	256	...	p
门输入端数	2	3	7	15	63	255	...	$p-1$

当每组的块容量为**8块或8块以上**时，所要的触发器个数及与门输入端个数很多，硬件实现的成本很高。

随着每组中的块容量增加，所需要的触发器的个数及与门的个数成平方关系增加。

堆栈法与比较对法

	堆栈法	比较对法
速度	速度比较低，因为它需要进行相联比。因此，当每一组的块容量比较大时，不宜采用堆栈法。	工作速度比较高，组合逻辑简单。
硬件实现	除了必须的寄存器之外，其它控制逻辑很简单。	相对比较复杂。需要比较多的触发器，特别是当每组的块容量比较大时。

比较对法所用触发器与堆栈的比例关系是：

$$\frac{G_b(G_b - 1)}{2} : G_b \log_2 G_b$$

其中， G_b 是Cache每一组的块容量。在 G_b 比较小时，两者的差别不大，当 G_b 大于8时，堆栈法所用的器件明显少于比较对法。

Cache替换算法与实现

■ 替换算法实现的设计是围绕下述**两点**来考虑的：

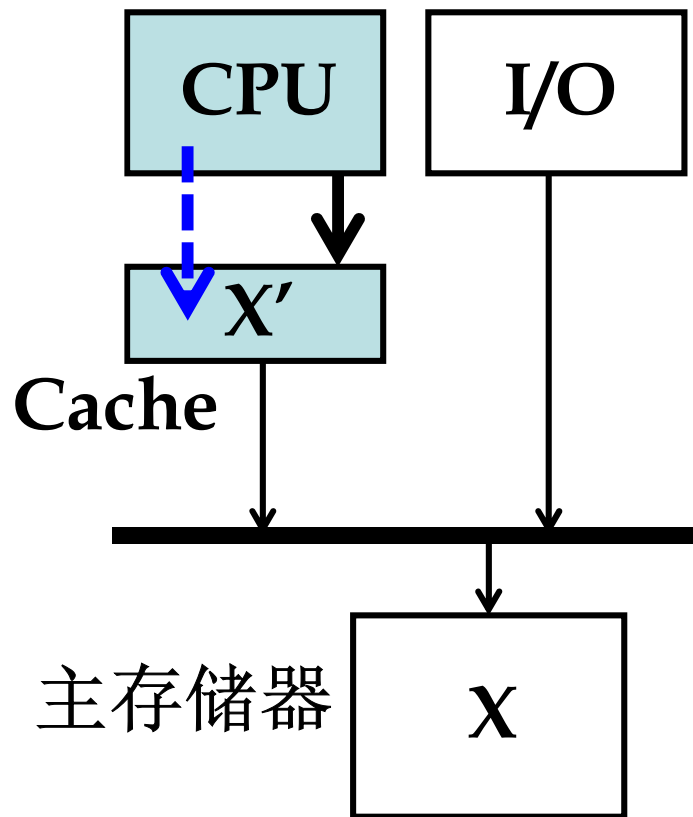
- 1. 如何对每次访问进行记录(使用位法、堆栈法、比较对法所用的记录方法都不同)；
- 2. 如何根据所记录的信息来判定近期内哪一块是最久没有被访问过的。

■ 实现方法和所用的映像方法密切相关。

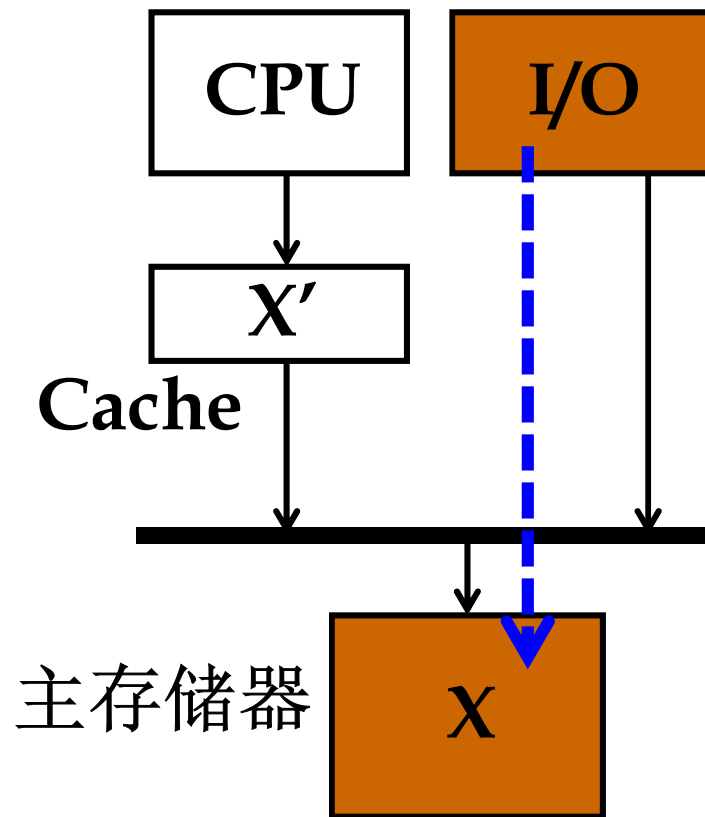
4.3.4 Cache的透明性及性能分析

- Cache—主存存贮层次对所有程序员透明
- Cache内容是主存内容的一小部分副本
- 但Cache内容有可能与主存内容不一致：
 - CPU更新（写）了Cache而未更新主存；
 - I/O更新了主存而未更新Cache；

1. Cache透明性分析



(a) CPU 写 Cache



(b) I/O 写主存

Cache与主存不一致的两种情况

1. Cache透明性分析

- 必须解决Cache的一致性问题
- **解决问题的关键：** 写Cache时如何更新主存的内容。
- “读写”操作所占的比例
 - **Load指令：26%**
 - **Store指令：9%**
- “写”在访问Cache操作中所占的比例：
 - **$9\% / (26\% + 9\%) \approx 25\%$**

1. Cache透明性分析

- **大概率事件优先原则**：优化Cache读操作
- **Amdahl定律**：不可忽视“写”速度
- **“写”问题**
 - 读出标识，确认命中后，对Cache写（串行操作）
 - Cache与主存内容的一致性问题
- **写策略就是要解决：何时更新主存问题**

Cache一致性算法：2 种写策略

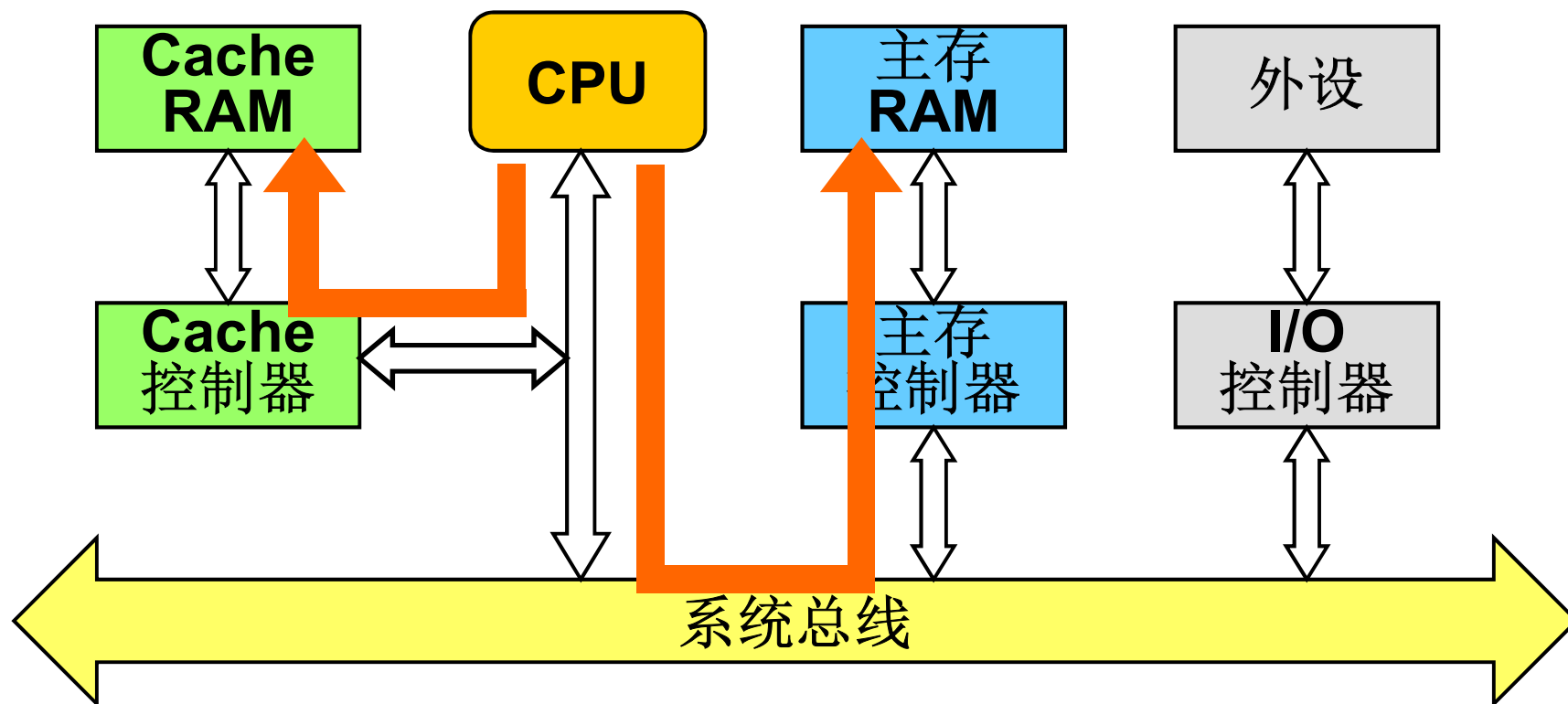
■ 写直达法（Write-through）

- CPU在执行写操作时，利用直接通路，把数据同时写入Cache和主存

■ 写回法（Write-Back）

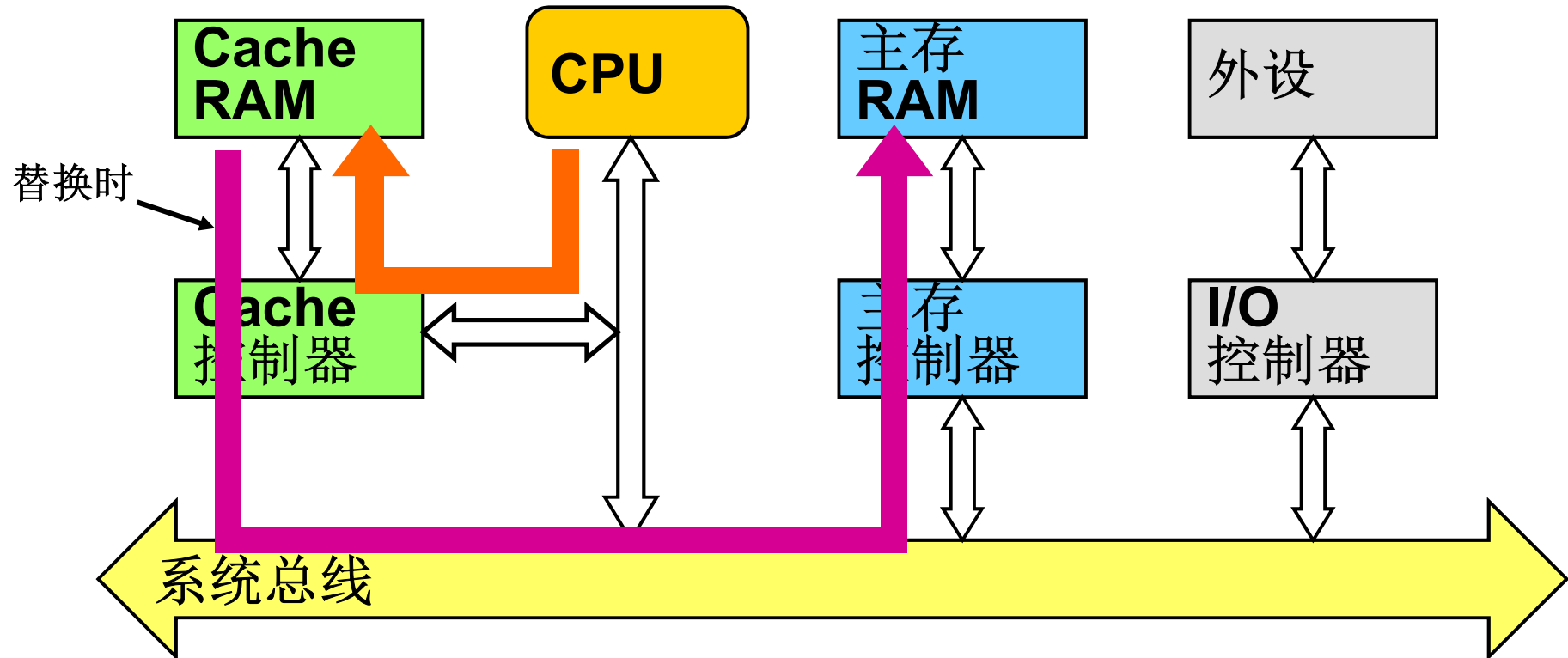
- 也称为抵触修改法
- CPU数据只写入Cache，不写入主存
- 为每个Cache块设置“修改位”
- 仅当替换时，才把修改过的Cache块写回到主存

Cache一致性算法：2种写策略



写直达法：数据**同时**写入 **Cache** 和主存

Cache一致性算法：2种写策略



写回法：数据只写入Cache，**替换时**才写入主存

两种“写”策略的比较

	写直达法	写回法
可靠性	好于写回法	块替换前仍存在一致性问题
与主存的通信量		少于写直达法达10多倍
控制复杂性	比写回法简单	
硬件实现代价		比写直达法低得多

“写”调块

- “写”操作必须在确认是命中后才可进行
- **问题：**当出现写不命中时，是否需要将主存块调入Cache？
- 两种方法：
 - **不按写分配法：**在写Cache不命中时，把所要写的字直接写入主存，不调块；
 - **按写分配法：**在写Cache不命中时，写入主存，并把单元所在块调入Cache；

写策略与调块

■ 一般：

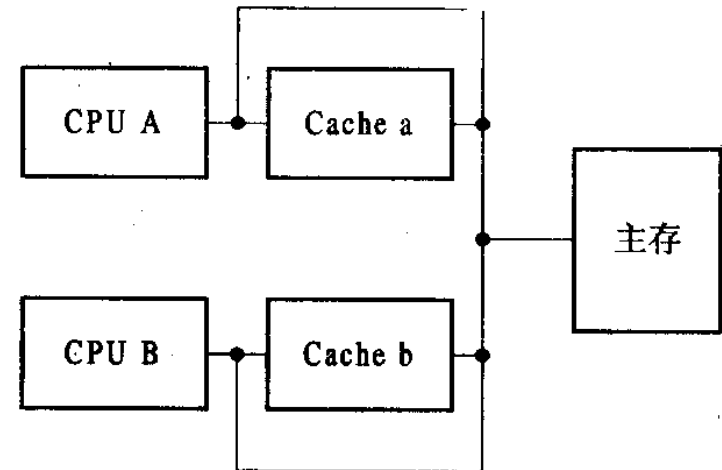
- 写直达法 + 不按写分配
- 写回法 + 按写分配

■ 单处理机系统

- 大多采用写回法

■ 共享主存的多处理机

- 为保证各处理机经主存交换信息时不出错，较多采用写直达法
- 多**Cache**一致性算法



“写” 缓冲器

- 写回法和写直达法都需要少量缓冲器
- 写回法中存放将要写回的块，不必等待写回主存后才进行**Cache**取
- 写直达法中存放要写回主存的内容，减少**CPU**等待写主存的时间
- 缓冲器对**Cache**—主存透明

两种写策略

■ 写直达法 (Write through)

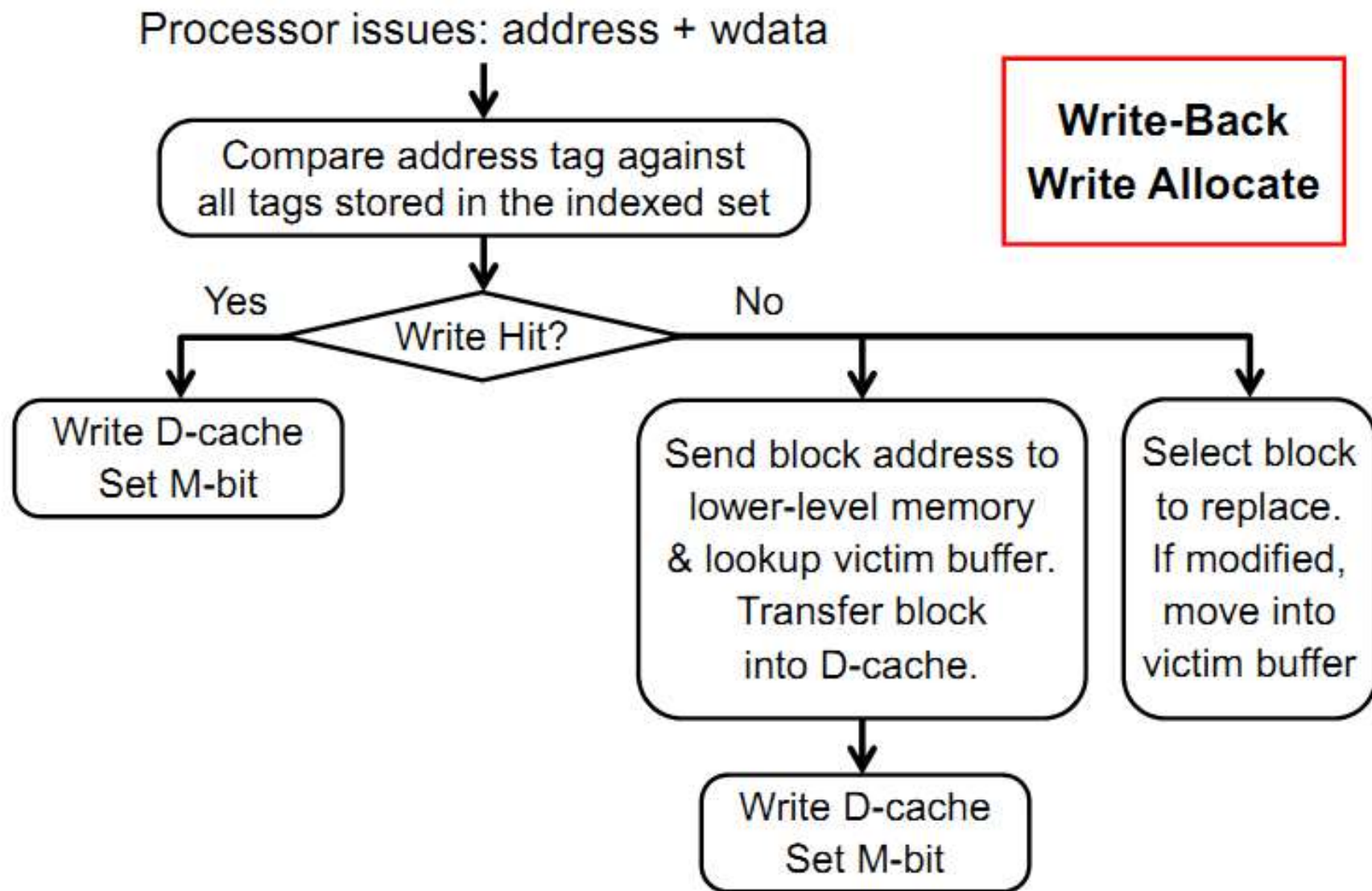
- 优点：易于实现，容易保持不同层次间的一致性
- 缺点：速度较慢

■ 写回法

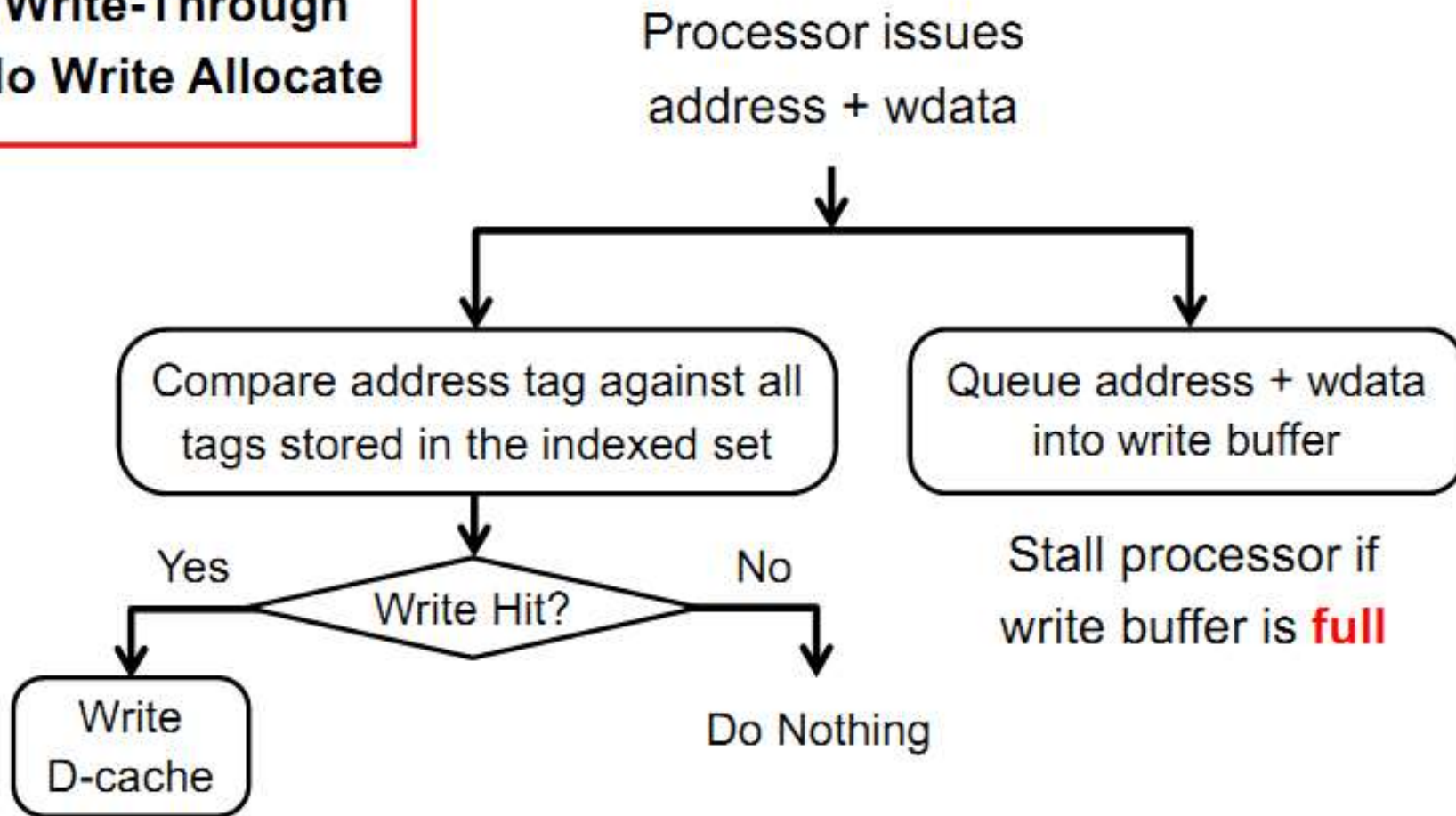
- 优点：速度快，减少访存次数
- 缺点：一致性问题

■ 当发生写失效时的两种策略

- 按写分配法(Write allocate)：写失效时，先把所写单元所在块调入Cache，然后再进行写入，也称写时取 (Fetch on Write)方法
- 不按写分配法 (no-write allocate)：写失效时，直接写入下一级存储器，而不将相应块调入Cache，也称绕写法 (Write around)
- 原则上以上两种方法都可以应用于写直达法和写回法，**一般情况下**
 - ◆ Write Back 用 Write allocate
 - ◆ Write through 用 no-write allocate



**Write-Through
No Write Allocate**



2. Cache的预取算法

- Cache的命中率对性能影响很大
- 如何预取提高Cache的命中率？
- 命中率与很多因素有关：
 - 预取算法
 - 块大小
 - 预取开销等

2. Cache的预取算法

■ 预取算法有如下几种：

- **按需取**：在出现**Cache**不命中时，把一个块取到**Cache**中来
- **恒预取**：无论**Cache**是否命中，都把下一块取到**Cache**中
- **不命中预取**：当**Cache**不命中，把本块和下一块一起取到**Cache**中

主要考虑因素：

- 命中率的提高；
- **Cache**与主存之间通信量的增加。

2. Cache的预取算法

■ 从模拟实验的结果看：

- 采用恒预取能使**Cache**的不命中率降低**75~85%**
- 采用不命中预取能使**Cache**的不命中率降低**30~40%**

■ **Cache**所用的取算法基本上仍是**按需取进法**，即在出现**Cache**块失效时，才将要访问的字所在的块(行)取进。由于程序存在局部性，只要适当选择好**Cache**的容量、块的大小、组相联的组数和组内块数，是可以保证有较高的命中率的。

2. Cache的预取算法

- **注意：采用预取法并非一定能提高命中率，它还和其他因素有关。**
 - **一是块的大小。**若每块的字节数过少，预取的效果不明显。从预取需要出发，希望块尽可能增大。但若每块的字节数过多，一方面可能会预取进不需要的信息，另一方面由于Cache的容量有限，又可能把正在使用或近期内就要用到的信息给替换出去，反而降低了命中率。从已有模拟结果来看，每块的字节数如果超过256，就会出现这种情况。
 - **二是预取开销。**要预取就要有访主存开销和将它取进Cache的访Cache开销，还要加上把被替换块写回主存的开销。这些开销会增加主存和Cache的负担，干扰和延缓程序的执行。

3. 任务切换对失效率的影响

- 受限于Cache的容量，多个进程的工作区很难同时保留在Cache内。
- 因此，造成Cache失效的一个**重要原因**是任务切换。
- 失效率的高低当然就和任务切换的频度有关，或者说与任务切换的平均时间间隔 Q_{sw} 的大小有关。

3. 任务切换对失效率的影响

■ 解决办法：

- 增大Cache容量；
- 修改调度算法，使任务切换回来之前，有用的信息仍能保留在Cache中不被破坏；
- 设置多个Cache，例如设置两个Cache，一个专用于管理程序，一个专用于用户程序。这样，在管态和目态之间切换时，不会破坏各自Cache中的内容。
- 此外，对于某些操作，例如长的向量运算、长的字符行运算等，可以不经Cache直接进行，以避免这些操作由于使用Cache，而从Cache中置换出大量更有希望被重用的数据。

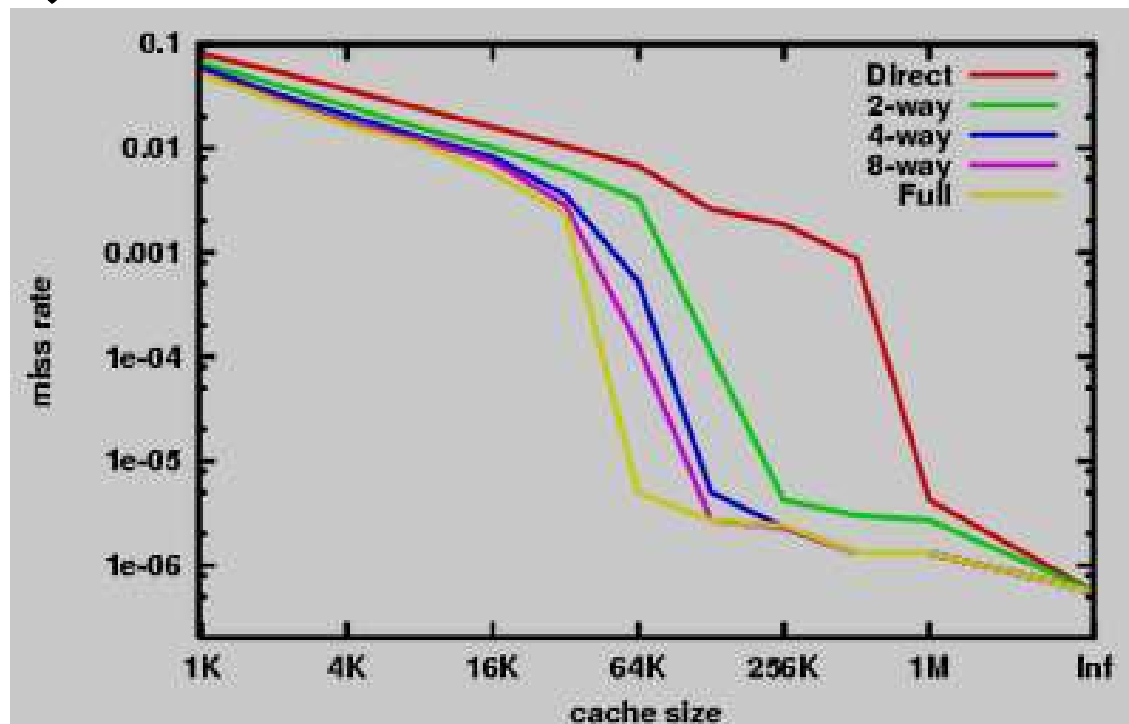
4. 影响Cache存储器性能的因素

■ 影响Cache性能的因素很多：

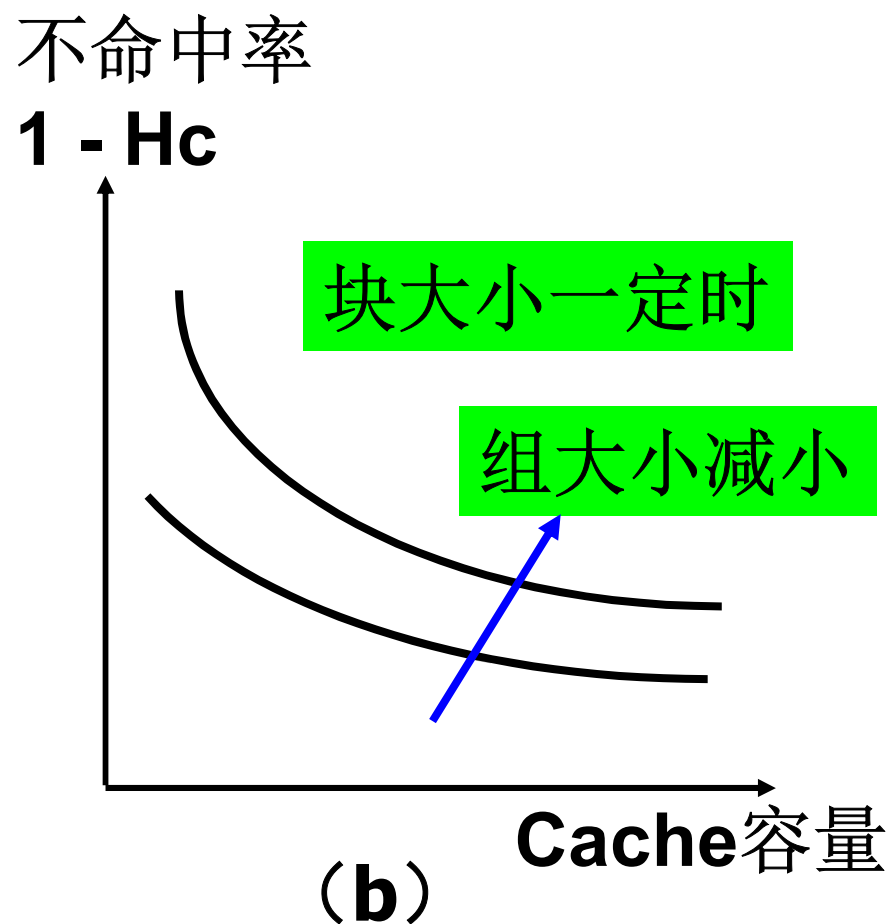
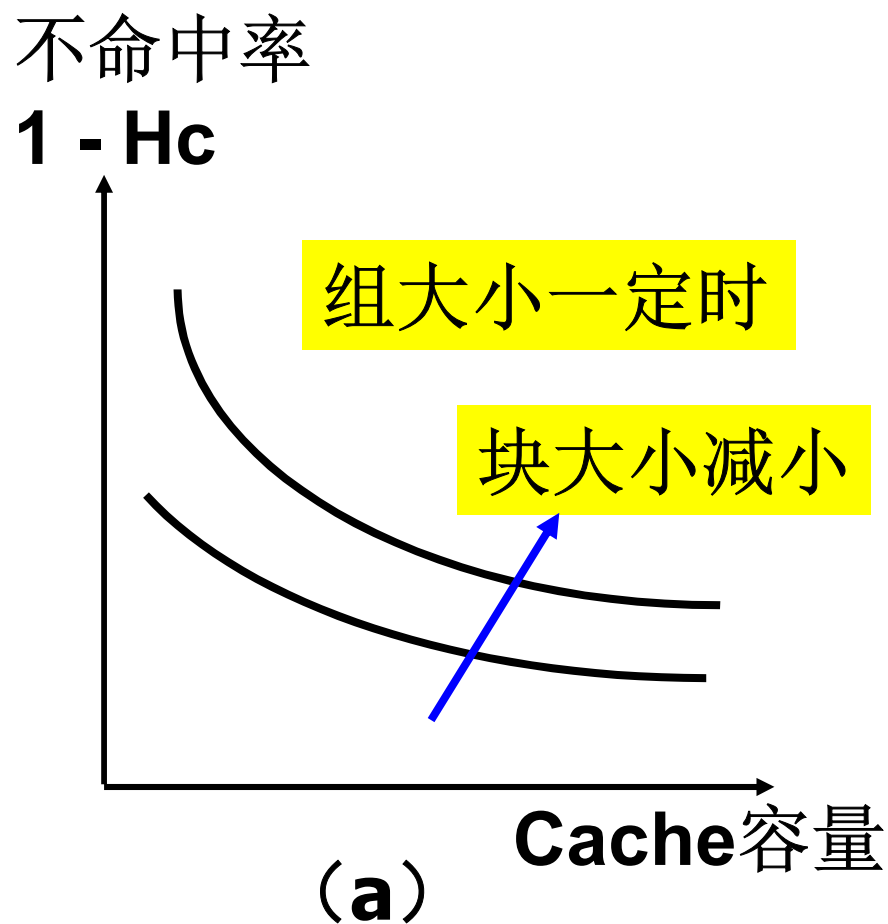
- 命中率
- 块大小，块数量
- 采用组相联时，组内块数和组数
- 替换算法
- 地址流
- **Cache预取算法等**

4. 影响Cache存储器性能的因素

- 相对来说，Cache 容量越大，命中率相对来说也就越高，但是在达到一定的容量之后，命中率的增长与改善则会趋于0.



4. 影响Cache存储器性能的因素



5. Cache性能分析

■ Cache的等效访问时间

$$t_a = H_c t_c + (1 - H_c) t_m$$

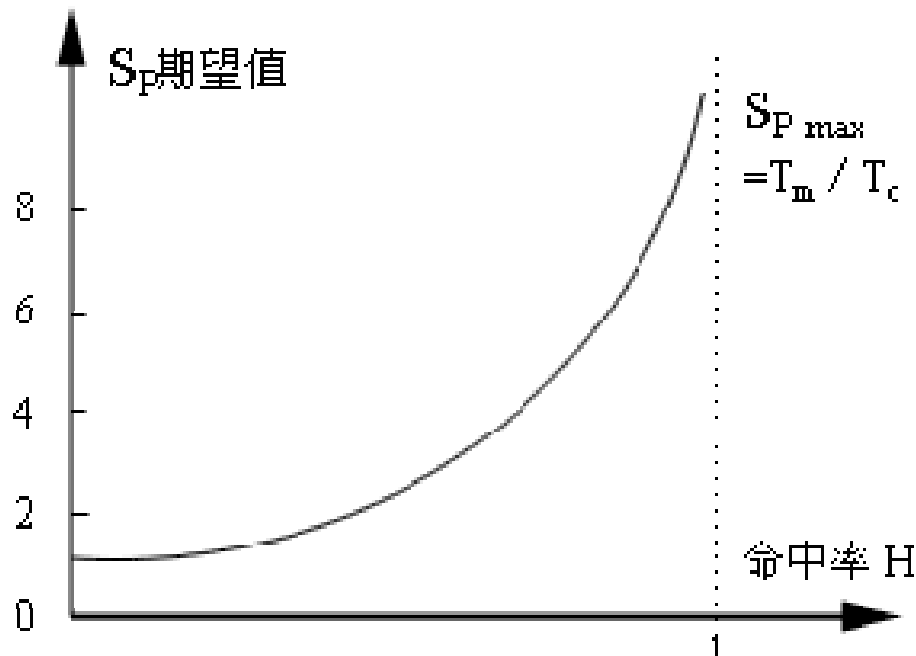
■ 由于Cache与主存之间有直接通路，所以加速比为：

$$S_p = \frac{t_m}{t_a} = \frac{t_m}{H_c + (1 - H_c) t_m} = \frac{1}{1 - (1 - t_c / t_m) H_c} < \frac{1}{1 - H_c}$$

当 $t_c \ll t_m$ 时

5. Cache性能分析

- 在Cache系统中，主存储器的访问周期 T_m 和Cache的访问周期 T_c 由于受所用器件的限制通常是一定的。因此，**提高Cache系统的加速比 S_p 最好的途径是提高命中率 H 。**



由于命中率 H 的值一般都大于**0.9**，能达到**0.99**以上，因此，实际上**Cache**的加速比 **S_p** 能够接近于它的最大值 T_m/T_c 。

6. Cache层次

■ 1. 统一Cache和分离Cache

- **统一Cache:** 只有一个Cache, 指令和数据混放。
- **分离Cache:** 分为指令Cache和数据Cache。

它消除了流水线中指令处理器和执行单元间的竞争, 因此特别适用于超标量流水线; 是Cache结构发展的趋势。

■ 2. 单级Cache与多级Cache

- **L1 Cache, L2 Cache, L3 Cache**
采用多级Cache结构可以提高性能。

Improving Cache Performance

- **AMAT = hit time + miss rate * miss penalty**
 - Reduce hit time
 - Reduce miss rate
 - Reduce miss penalty

Reducing Cache Miss Penalty

■ Multilevel caches

- Very Fast, small Level 1 (L1) cache
- Fast, not so small Level 2 (L2) cache
- May also have slower, large L3 cache, etc.

■ Why does this help?

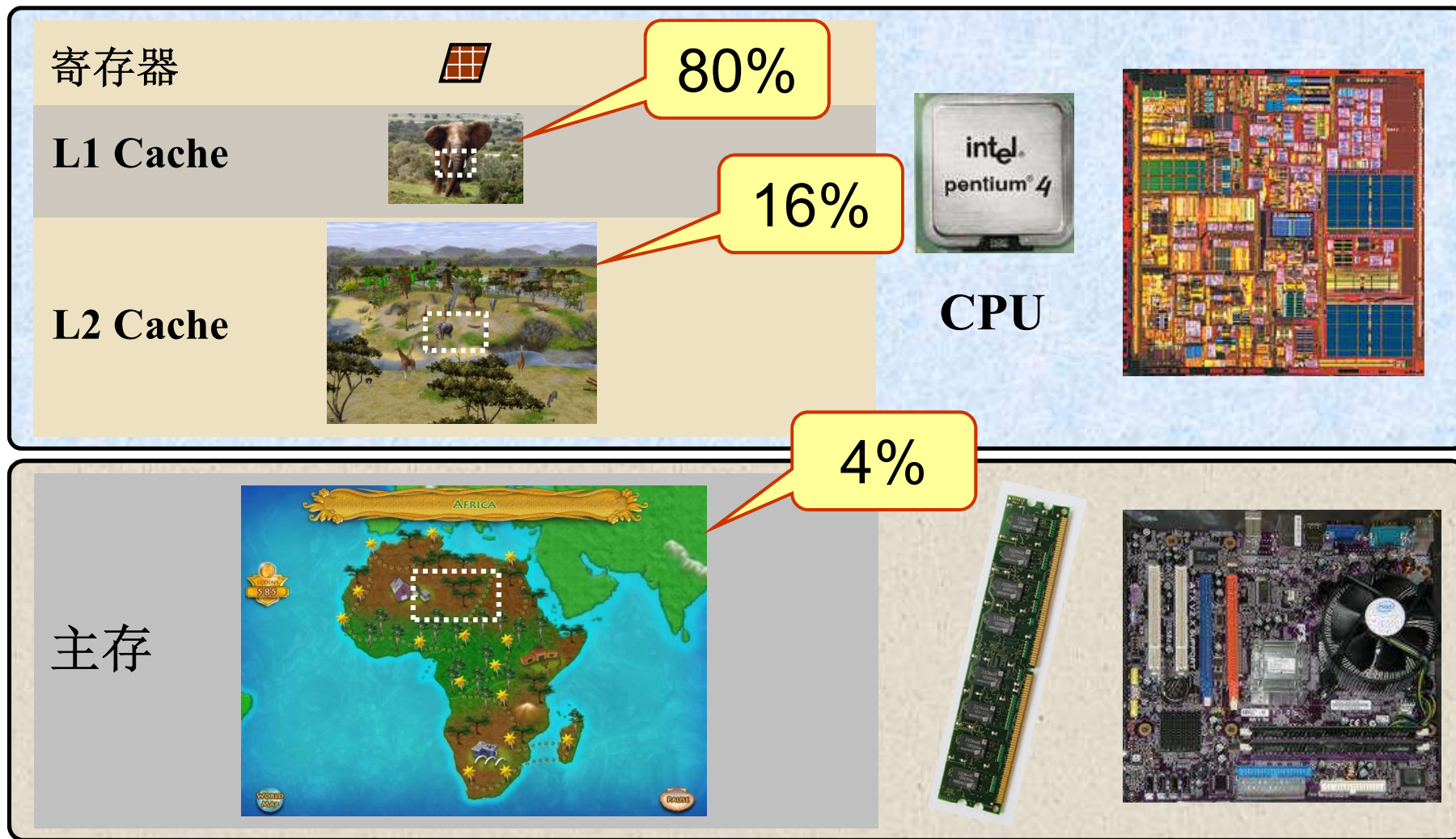
- Miss in L1 cache can hit in L2 cache, etc.

$$AMAT = HitTime_{L1} + MissRate_{L1} MissPenalty_{L1}$$

$$MissPenalty_{L1} = HitTime_{L2} + MissRate_{L2} MissPenalty_{L2}$$

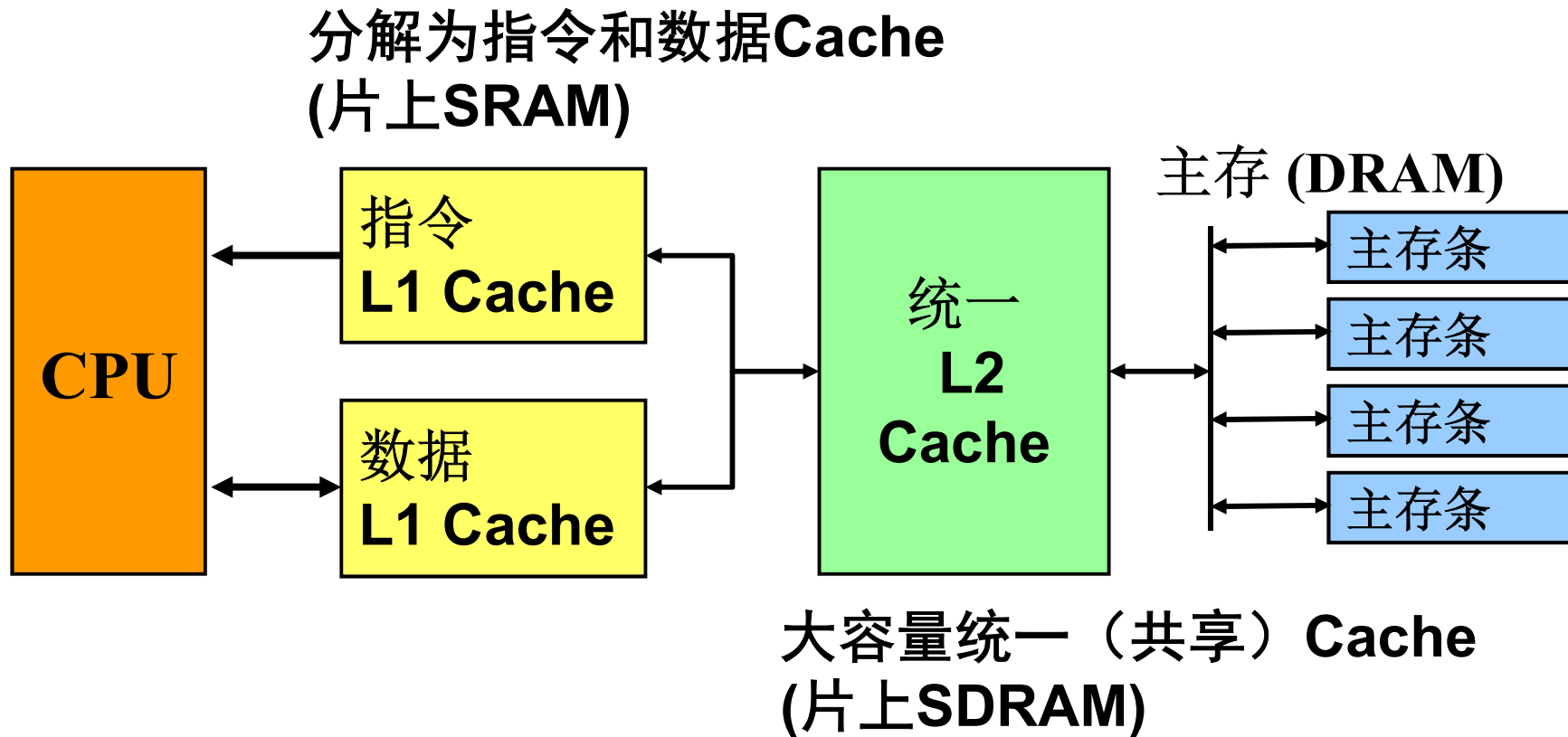
$$MissPenalty_{L2} = HitTime_{L3} + MissRate_{L3} MissPenalty_{L3}$$

6. Cache层次



Cache层次(20/80规则)

6. Cache层次



一种典型的Cache存储层次

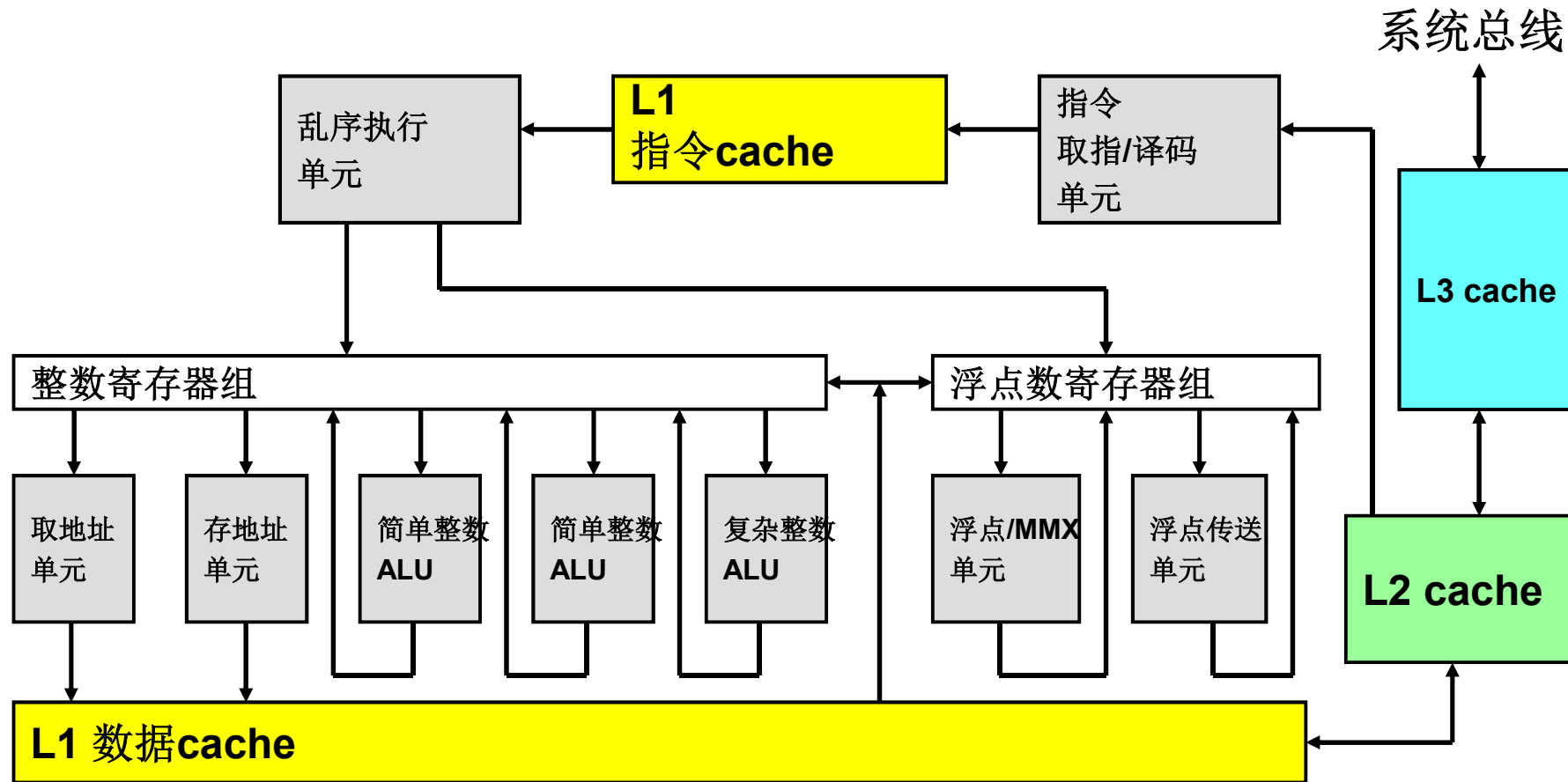
PC中的Cache

CPU	L1 cache	L2 cache	L3 cache
8088	无	无	无
80286	无	无	无
80386dx	片外	无	无
80486dx	片内8KB	片外	无
Pentium	片内8KB+8KB	片外	无
Pentium MMX	片内16KB+16KB	片外	无
Pentium II/III	片内16KB+16KB	卡上512KB~1MB	无
AMD毒龙	片内128KB	片内64KB	无
AMD雷鸟	片内128KB	片内256KB	无
Pentium IV	片内8KB数据 +2KB指令	片内256KB	片内1MB

Intel Pentium 4 Cache

- Pentium (所有版本): 2个片上L1 cache, 数据和指令
- Pentium III: 增加了片外L3 cache
- Pentium 4
 - L1 cache
 - ◆ 容量: 8K 字节
 - ◆ 块大小 (一行): 64 字节
 - ◆ 映像规则: 4路组相联
 - L2 cache: 为数据和指令cache提供输入
 - ◆ 容量: 256K字节
 - ◆ 块大小 (一行): 128 字节
 - ◆ 映像规则: 8路组相联
 - 片内L3 cache, 1MB

Intel Pentium 4 Cache



Measuring Cache Performance

■ Components of CPU time

- Program execution cycles

 - ◆ Includes cache hit time

- Memory stall cycles

 - ◆ Mainly from cache misses

■ With simplifying assumptions:

Memory stall cycles

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instructions}} \times \text{Miss penalty}$$

Cache Performance Example

■ Given

- I-cache miss rate = 2%
- D-cache miss rate = 4%
- Miss penalty = 100 cycles
- Base CPI (ideal cache) = 2
- Load & stores are 36% of instructions

■ Miss cycles per instruction

- I-cache: $0.02 \times 100 = 2$
- D-cache: $0.36 \times 0.04 \times 100 = 1.44$

■ Actual CPI = $2 + 2 + 1.44 = 5.44$

- Ideal CPU is $5.44/2 = 2.72$ times faster

Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
 - $\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$
- Example
 - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
 - $\text{AMAT} = 1 + 0.05 \times 20 = 2\text{ns}$
 - ◆ 2 cycles per instruction

Performance Summary

- **When CPU performance increased**
 - Miss penalty becomes more significant
- **Decreasing base CPI**
 - Greater proportion of time spent on memory stalls
- **Increasing clock rate**
 - Memory stalls account for more CPU cycles
- **Can't neglect cache behavior when evaluating system performance**

Improving Cache Performance

- **AMAT = hit time + miss rate * miss penalty**
 - Reduce hit time
 - Reduce miss penalty
 - Reduce miss rate

基本Cache优化方法

■ 降低缺失率

- 1、增加Cache块的大小
- 2、增大Cache容量
- 3、提高相联度

■ 减少缺失开销

- 4、多级Cache
- 5、使读失效优先于写失效

■ 缩短命中时间

- 6、避免在索引缓存期间进行地址转换

降低失效率

Cache 缺失的原因 可分为三类（3C）

■ 强制性失效 (Compulsory)

- 第一次访问某一块，只能从下一级Load，也称为冷启动或首次访问失效。

■ 容量失效 (Capacity)

- 如果程序执行时，所需块由于容量不足，不能全部调入Cache，则当某些块被替换后，若又重新被访问，就会发生失效。
- 可能会发生“抖动”现象。

■ 冲突失效 (Conflict (collision))

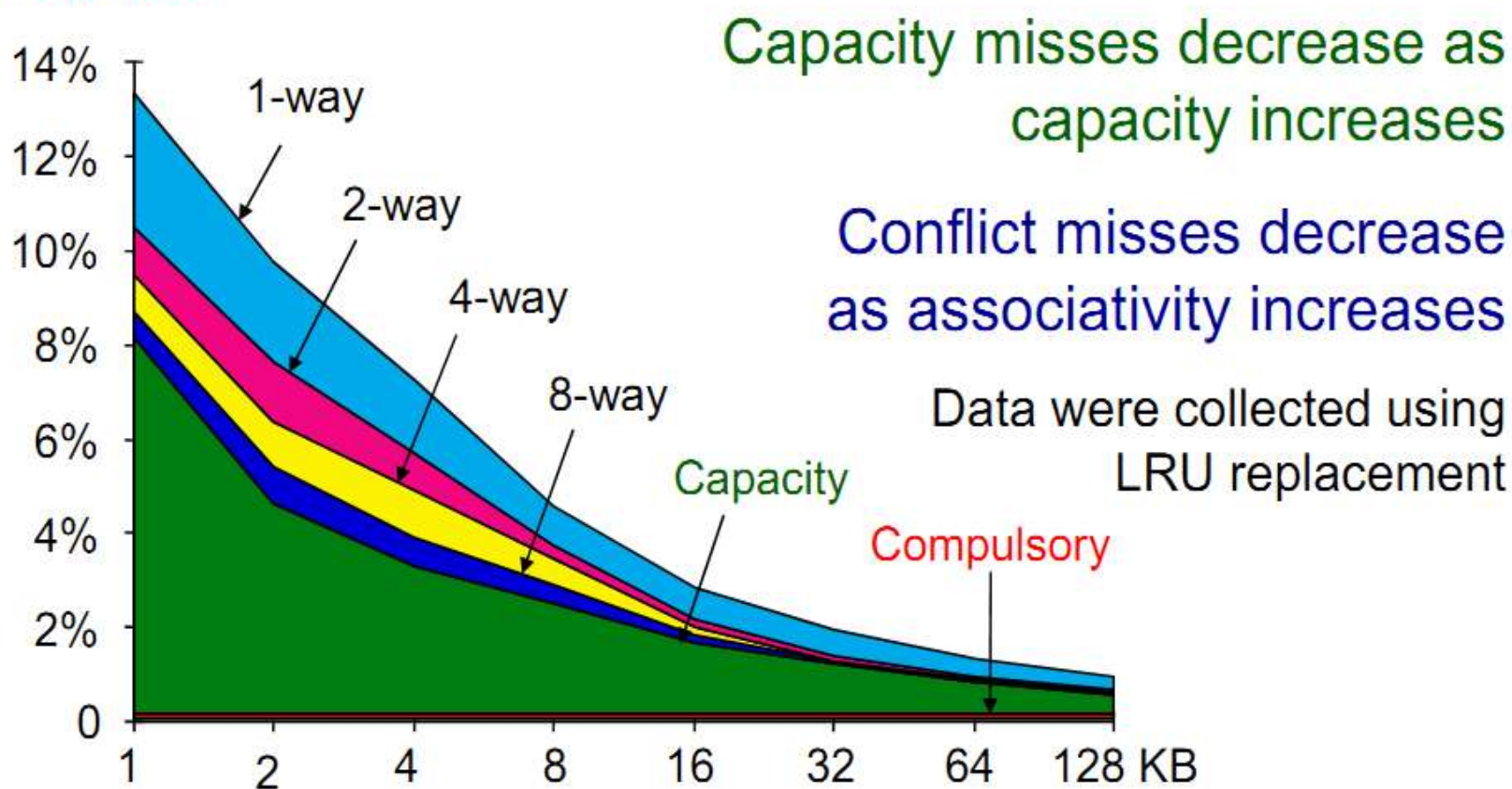
- 组相联和直接相联的副作用。
- 若太多的块映象到同一组（块）中，则会出现该组中某个块被别的块替换（即使别的组或块有空闲位置），然后又被重新访问的情况，这就属于冲突失效。

各种类型的失效率

Compulsory misses are independent of cache size

Very small for long-running programs

Miss Rate



从统计规律中得到的一些结果

- 相联度越高，冲突失效就越小
- 强制性失效和容量失效不受相联度的影响
- 强制性失效不受Cache容量的影响
- 容量失效随着容量的增加而减少
- 符合2:1 Cache经验规则
 - 即大小为 N 的直接映象Cache的失效率约等于大小为 $N/2$ 的两路组相联的Cache失效率。

减少3C的方法

从统计规律可知：

■ 增大Cache容量

- 对冲突和容量失效的减少有利

■ 增大块

- 减缓强制性失效
- 可能会增加冲突失效（因为在容量不变的情况下，块的数目减少了）

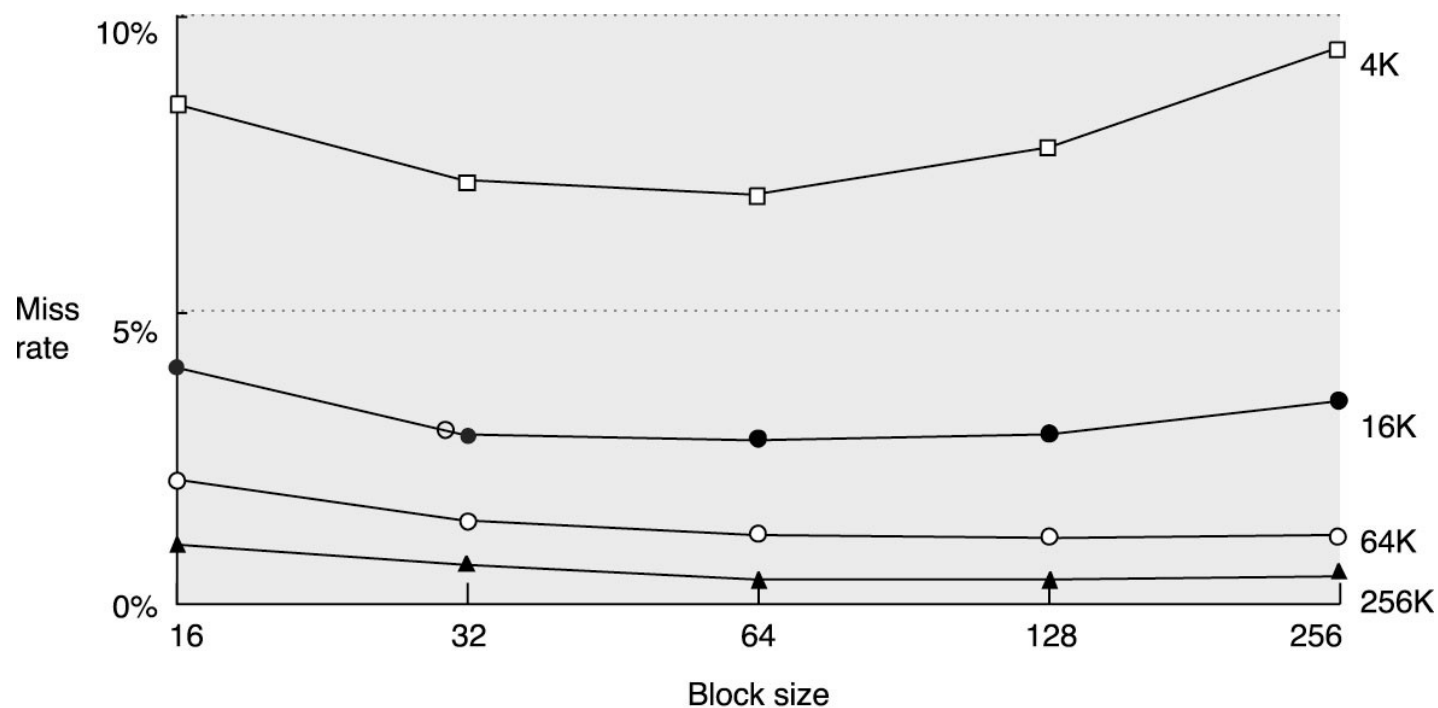
■ 通过预取可帮助减少强制性失效

- 必须小心不要把你需要的东西换出去
- 需要预测比较准确（对数据较困难，对指令相对容易）

降低失效率

■ Larger blocks

- Helps if there is more spatial locality



增加Cache块大小

■ 从统计数据可得到如下结论

- 对于给定Cache容量，块大小增加时，失效率开始是下降，但后来反而上升
- Cache容量越大，使失效率达到最低的块大小就越大

■ 分析

- 块大小增加，可使强制性失效减少（空间局部性原理）
- 块大小增加，可使冲突失效增加（因为Cache中块数量减少）
- 失效开销增大（上下层间移动，数据传输时间变大）

■ 设计块大小的原则，不能仅看失效率

- 原因：平均访存时间 = 命中时间 + 失效率 × 失效开销

降低缺失率

■ Larger caches

- Fewer capacity misses, but longer hit latency!

■ Higher Associativity

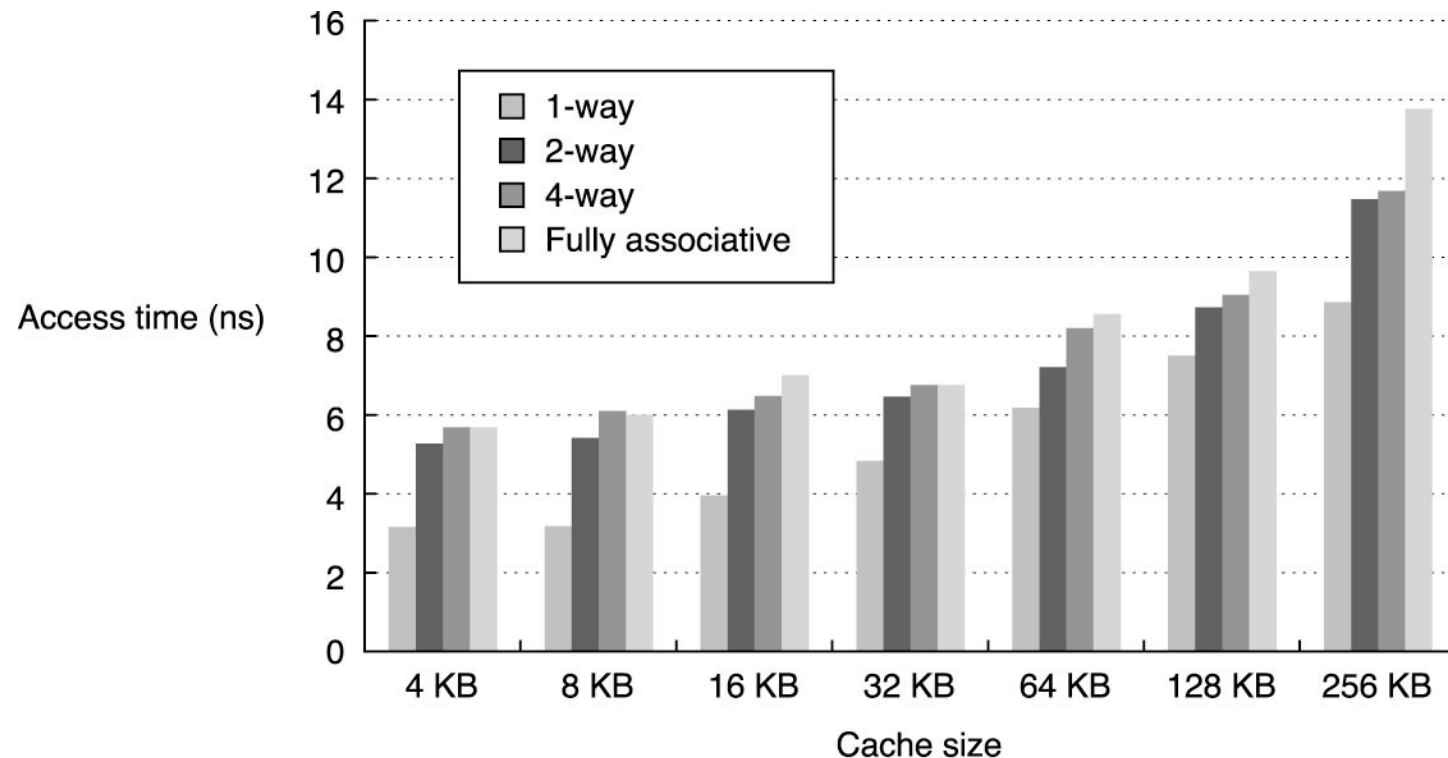
- Fewer conflict misses, but longer hit latency

■ Way Prediction

- Speeds up set-associative caches
- Predict which of N ways has our data, fast access as direct-mapped cache
- If mispredicted, access again as set-assoc cache

Reducing Hit Time

■ Small & Simple Caches are faster



减少缺失开销

■ 减少CPU与存储器间性能差异的重要手段

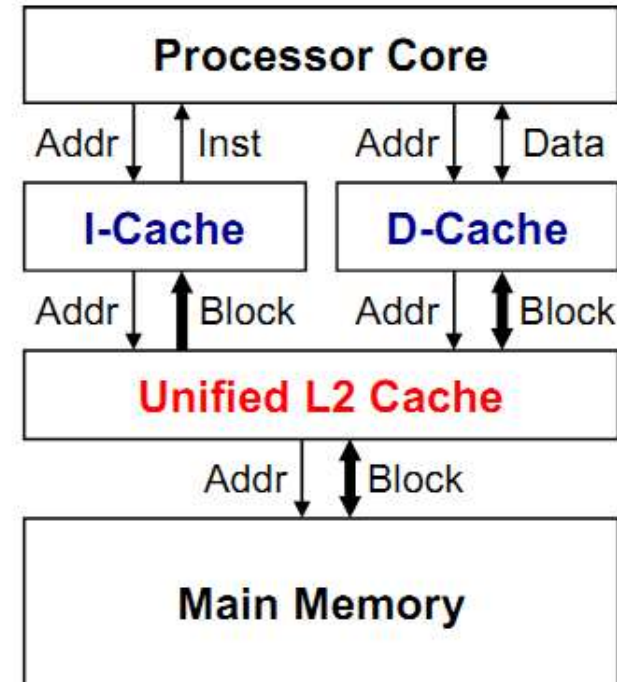
- $\text{平均访存时间} = \text{命中时间} + \text{失效率} \times \text{失效开销}$

■ 基本手段：

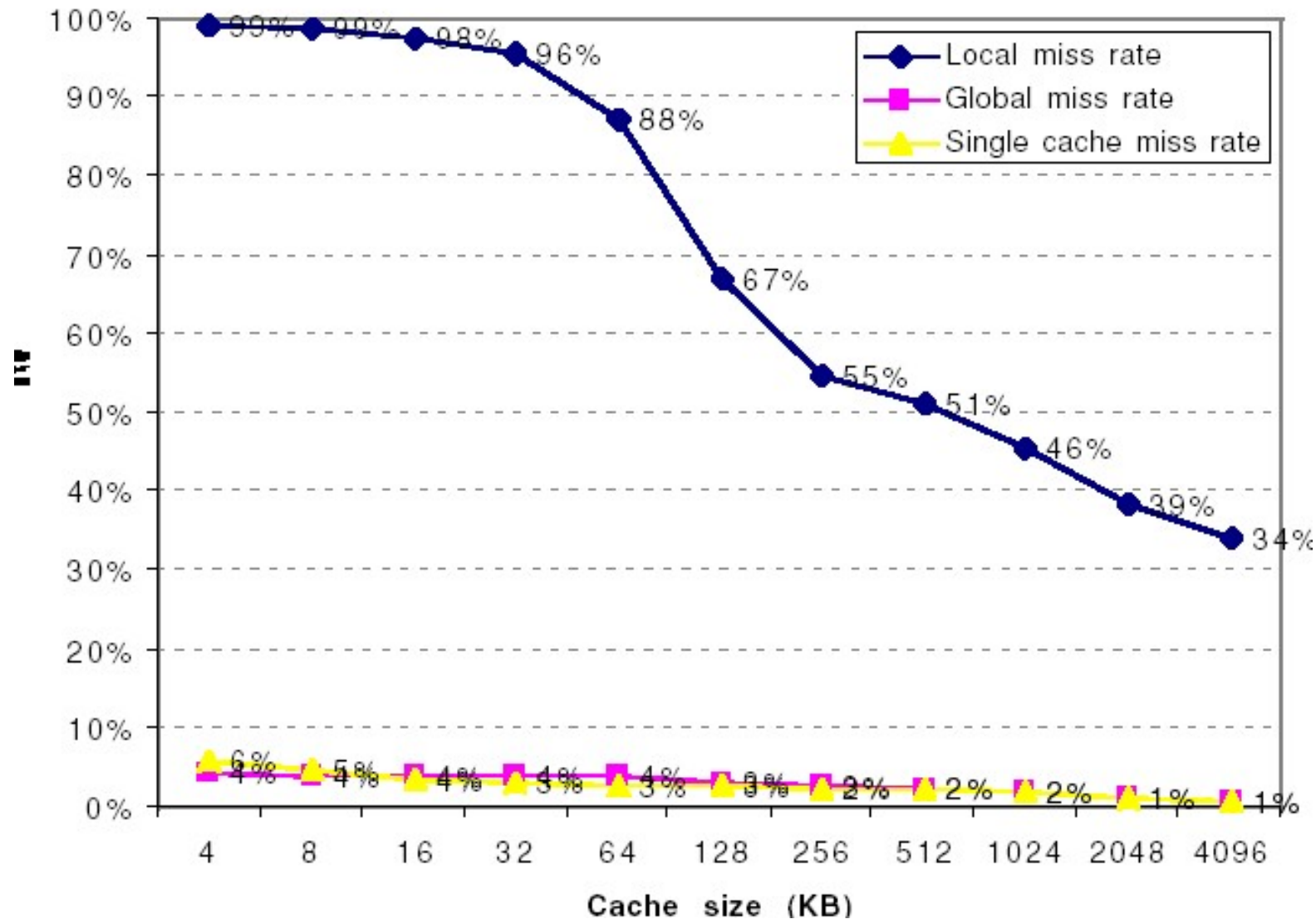
- 1、多级 Cache 技术 (Multilevel Caches)
- 2、让读优先于写 (Giving Priority to Read Misses over Writes)

采用多级Cache

- 与CPU无关，重点是Cache与Memory之间的接口
- **问题：**为了使Memory-CPU性能匹配，到底应该把Cache做的更快，还是应该把Cache做的更大
- **答案：**两者兼顾。二级Cache –降低失效开销（减少访问存储器的次数。
- 带来的复杂性：性能分析问题
- 性能参数
 - $AMAT = Hit\ Time_{L1} + Miss\ rate_{L1} \times Miss\ penalty_{L1}$
 $Miss\ penalty_{L1} = Hit\ Time_{L2} + Miss\ rate_{L2} \times Miss\ penalty_{L2}$
 - $AMAT = Hit\ Time_{L1} + Miss\ rate_{L1} \times$
 $(Hit\ Time_{L2} + Miss\ rate_{L2} \times Miss\ penalty_{L2})$
 - 对第二级Cache，系统所采用的术语
 - ◆ 局部失效率：该级Cache的失效次数 / 到达该级Cache的访存次数
 - ◆ 全局失效率：该级Cache的失效次数 / CPU发出的访存总次数



Miss rates versus cache size for multilevel caches



两级Cache的一些研究结论

- 在 L2 比 L1 大得多得情况下，两级Cache的全局失效率 and 容量与第二级Cache相同的单级Cache的失效率接近
- 局部失效率不是衡量第二级Cache的好指标
 - 它是第一级Cache失效率的函数
 - 不能全面反映两级Cache体系的性能
- 第二级Cache设计需考虑的问题
 - 容量：一般很大，可能没有容量失效，只有强制性失效和冲突失效
 - ◆ 相联度对第二级Cache的作用
 - ◆ Cache可以较大，以减少失效次数
 - 多级包容性问题：第一级Cache中的数据是否总是同时存在于第二级Cache中。
 - ◆ 如果L1和L2的块大小不同，增加了多级包容性实现的复杂性

多级包容性(multilevel inclusive)

- L1 cache 的块总是存在于L2 cache中
 - 浪费了L2 cache 空间, L2 还应当有存放其他块的空间
- L1中miss, 但在L2中命中, 则从L2拷贝相应的块到L1
- 在L1和L2中均miss, 则从更低级拷贝相应的块到L1和L2
- 对L1写操作导致将数据同时写到L1和L2
- Write-through 策略用于L1到L2
- Write-back 策略可用于L2 到更低级存储器, 以降低存储总线的数据传输压力
- L2的替换动作 (或无效)对L1可见
 - 即L2的一块被替换出去, 那么其在L1中对应的块也要被替换出去。

多级不包容 (Multilevel Exclusive)

- L1 cache 中的块不会在L2 cache中，以避免浪费空间
- 在L1中miss, 但在L2中命中，将导致Cache间块的互换
- 在L1和L2中均miss, 将仅仅从更低层拷贝相应的块到L1
- L1的被替换的块移至L2
 - L2 存储L1抛弃的块，以防后续L1还需要使用
- L1到L2的写策略为 Write-Back
- L2到更低级cache的写策略为 Write-Back
- L1和L2的块大小可以相同也可以不同
 - Pentium 4 had 64-byte blocks in L1 but 128-byte blocks in L2
 - Core i7 uses 64-byte blocks at all cache levels (simpler)

让读失效优先于写

■ Write Buffer (写缓冲), 特别对写直达法更有效

- CPU不必等待写操作完成, 即将要写的数据和地址送到Write Buffer后, CPU继续作其他操作。

■ 写缓冲导致对存储器访问的复杂化

- 在读失效时写缓冲中可能保存有所读单元的最新值, 还没有写回
- 例如, 直接映射、写直达、512 和 1024 映射到同一块。则

SW R3, 512(R0)

LW R1, 1024(R0) 失效

LW R2, 512(R0) 失效

■ 解决问题的方法

1. 推迟对读失效的处理, 直到写缓冲器清空, 导致新的问题——读失效开销增大。
2. 在读失效时, 检查写缓冲的内容, 如果没有冲突, 而且存储器可访问, 就可以继续处理读失效

■ 由于读操作为大概率事件, 需要读失效优先, 以提高性能

■ 写回法时, 也可以利用写缓冲器来提高性能

- 把脏块放入缓冲区, 然后读存储器, 最后写存储器

高级Cache优化方法

- 缩短命中时间
 - 1、小而简单的第一级Cache
 - 2、路预测方法
- 增加Cache带宽
 - 3、Cache访问流水化
 - 4、无阻塞Cache
- 减小失效开销
 - 5、多体Cache
 - 6、关键字优先和提前重启
 - 7、合并写
- 降低失效率
 - 8、编译优化
- 通过并行降低失效代价或失效率
 - 9、硬件预取
 - 10、编译器控制的预取

Data Placement Policy

■ Direct mapped cache:

- Each memory block mapped to one location
- No need to make any decision
- Current item replaces previous one in location

■ N-way set associative cache:

- Each memory block has choice of N locations

■ Fully associative cache:

- Each memory block can be placed in ANY cache location

■ Misses in N-way set-associative or fully associative cache:

- Bring in new block from memory
- Throw out a block to make room for new block
- Need to decide on which block to throw out

Cache Block Replacement

- **Direct mapped:** no choice

- **Set associative or fully associative:**

- **Random**

- **LRU (Least Recently Used):**

- ◆ Hardware keeps track of the access history and replace the block that has not been used for the longest time

- **An example of a pseudo LRU (for a two-way set associative) :**

- ◆ use a pointer pointing at each block in turn

- ◆ whenever an access to the block the pointer is pointing at, move the pointer to the next block

- ◆ when need to replace, replace the block currently pointed at

Comparing the Structures

■ **N-way set-associative cache**

- **N comparators vs. 1**
- **Extra MUX delay for the data**
- **Data comes AFTER Hit/Miss decision and set selection**

■ **Direct mapped cache**

- **Cache block is available BEFORE Hit/Miss:**
- **Possible to assume a hit and continue, recover later if miss**

Multilevel Caches

- **Primary cache** attached to CPU
 - Small, but fast
- **Level-2 cache** services misses from primary cache
 - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

■ Given

- CPU base CPI = 1, clock rate = 4GHz
- Miss rate/instruction = 2%
- Main memory access time = 100ns

■ With just primary cache

- Miss penalty = $100\text{ns}/0.25\text{ns} = 400$ cycles
- Effective CPI = $1 + 0.02 \times 400 = 9$

Example (cont.)

- Now add L-2 cache

- Access time = 5ns (to M: 100ns)

- Global miss rate to main memory = 0.5% (to M 2%)

- Primary miss with L-2 hit

- Penalty = $5\text{ns}/0.25\text{ns} = 20$ cycles

- Primary miss with L-2 miss (0.5%)

- Extra penalty = 400 cycles

- $\text{CPI} = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$

- Performance ratio = $9/3.4 = 2.6$

Multilevel Cache Considerations

■ Primary cache

- Focus on minimal hit time

■ L-2 cache

- Focus on low miss rate to avoid main memory access
- Hit time has less overall impact

■ Results

- L-1 cache usually smaller than a single-level cache
- L-1 block size smaller than L-2 block size

第四章 存储体系

学习内容：

- 4.1 存储体系概念和并行存储系统
- 4.2 虚拟存储系统
- 4.3 高速缓冲存储器（Cache）
- 4.4 Cache - 主存 - 辅存三级层次
- ARM存储系统

4.4 Cache - 主存 - 辅存三级层次

- 目前的大部分计算机系统中，既有虚拟存储器，也有Cache。
- 程序员使用且只关心一个存储器：
 - 访问方式 = 按地址随机访问
 - 等效速度 = **Cache**
 - 等效容量 = 虚拟空间容量

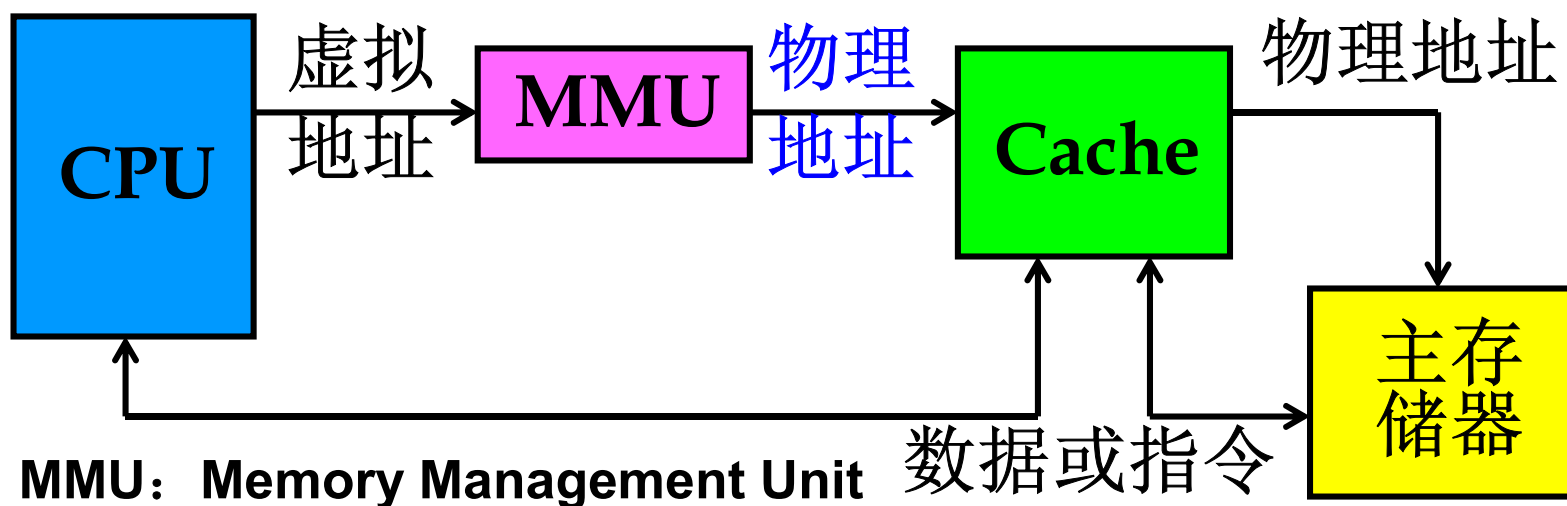
4.4 Cache - 主存 - 辅存三级层次

■ Cache、主存、磁盘这三个存储器可以分别构成：

1. 两个存储系统：“Cache—主存”和“主存—磁盘”
2. 一个存储系统：“Cache—主存—磁盘”
3. 全 Cache 系统

两个存储系统

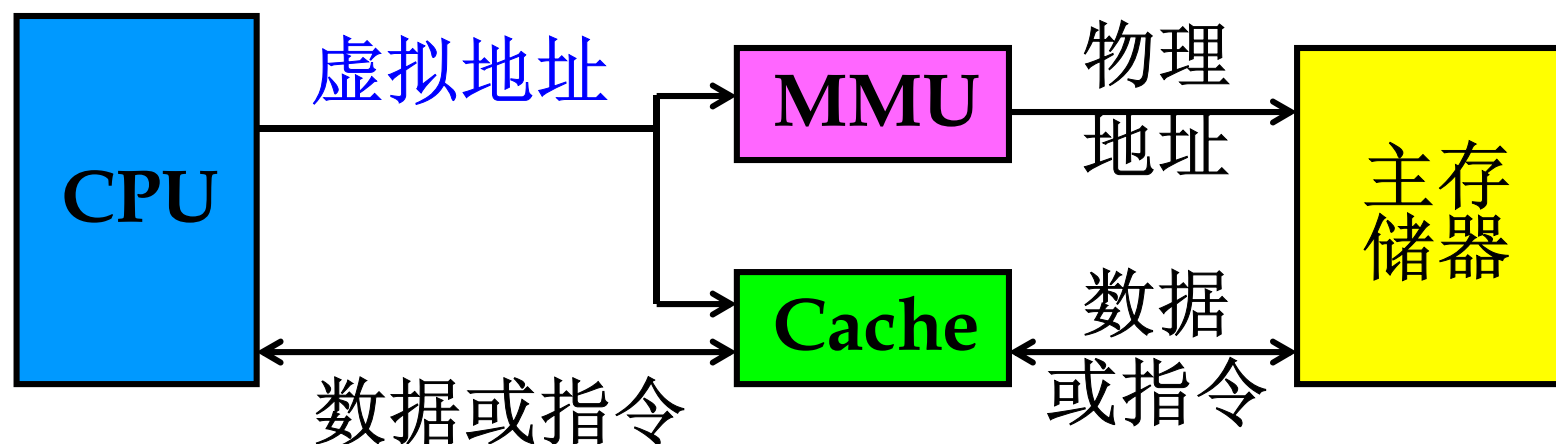
- 有“Cache—主存”和“主存—磁盘”两个独立的存储系统。这种结构在有些资料上也称为**物理地址 Cache**。



Intel公司的i486和DEC公司的VAX 8600等处理机均采用这种两级存储系统。

一个存储系统

- 把Cache、主存和磁盘三个存储器组织在一起构成一个“Cache—主存—磁盘”存储系统。有些资料上把这种组织方式称为**虚拟地址Cache**。

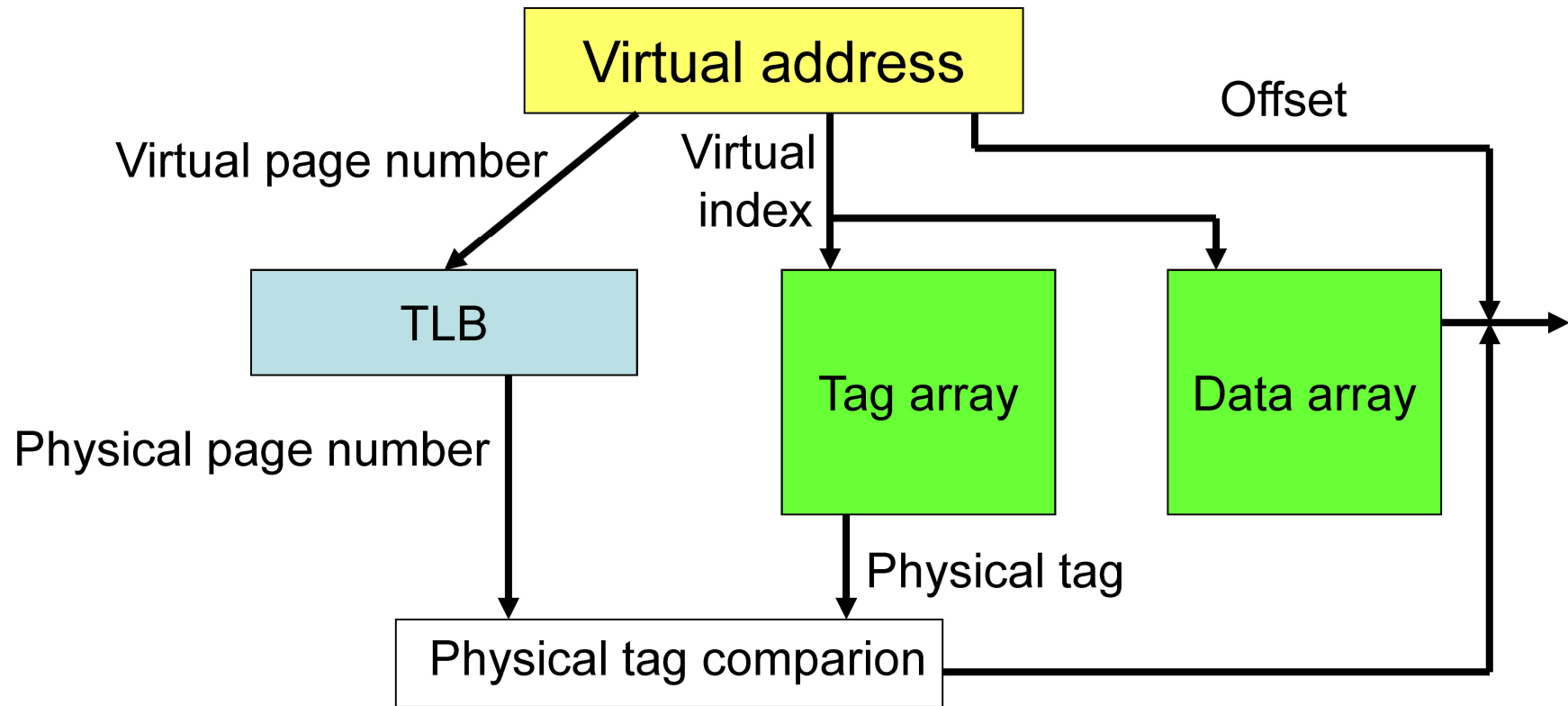


Intel公司的i860等处理机采用这种组织方式。

一个存储系统

- 在既有Cache，又有虚拟存储器的处理机中，如果对Cache的访问仍采用主存实地址，就要把虚拟地址首先变换成主存实地址，然后才能访问Cache，这样必然增加访问Cache所花费的时间，至少要增加一个查主存快表的时间。
- 因此，在许多系统中，采用**直接用虚拟地址访问 Cache 方法**。

Cache and TLB Pipeline

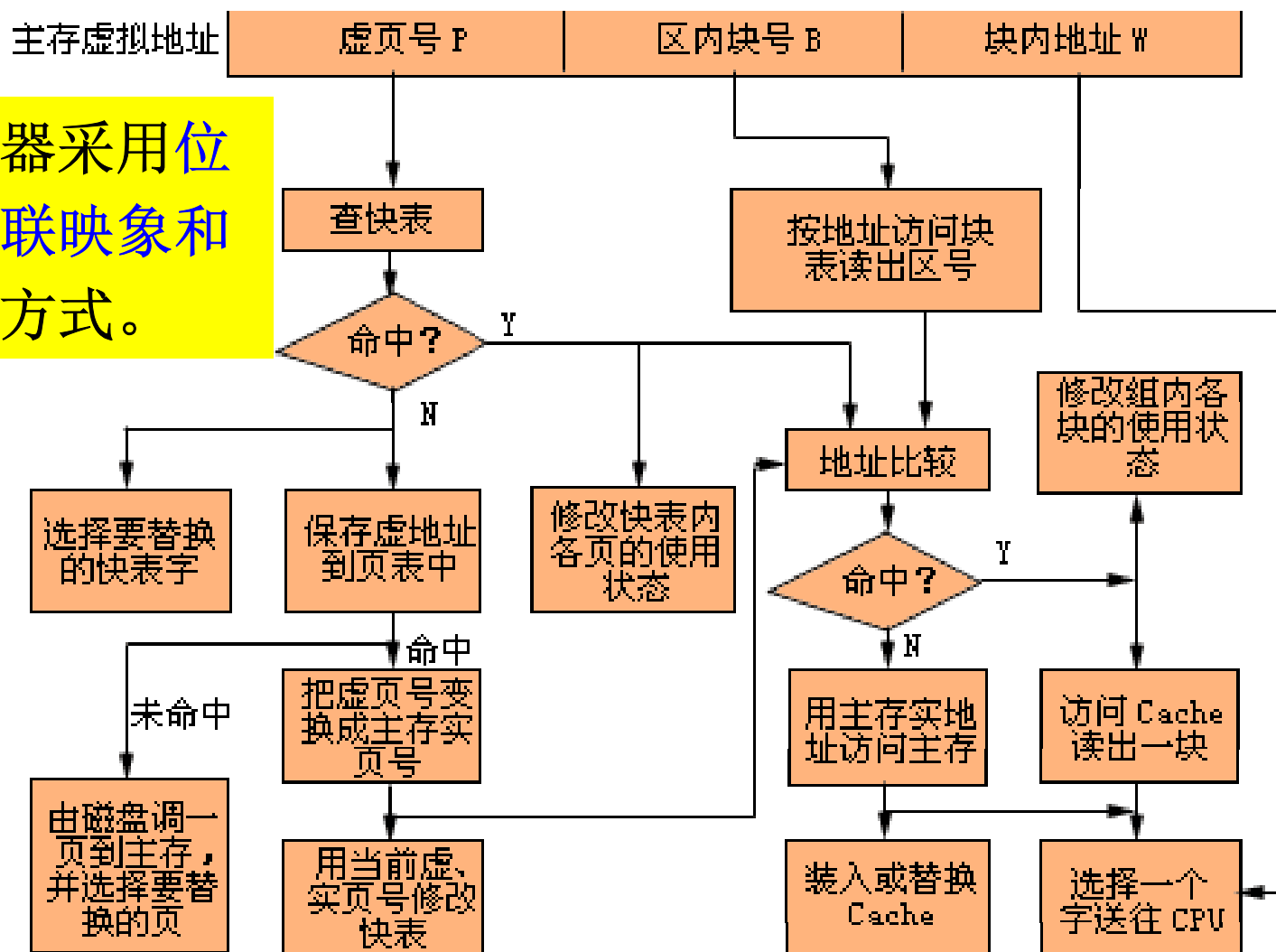


Virtually Indexed; Physically Tagged Cache

一个存储系统

使虚拟存储器中的一页恰好就是主存储器的一个区。可以直接用虚拟地址中的区内块号B按地址访问Cache的块表。区号实际上也就是页号。

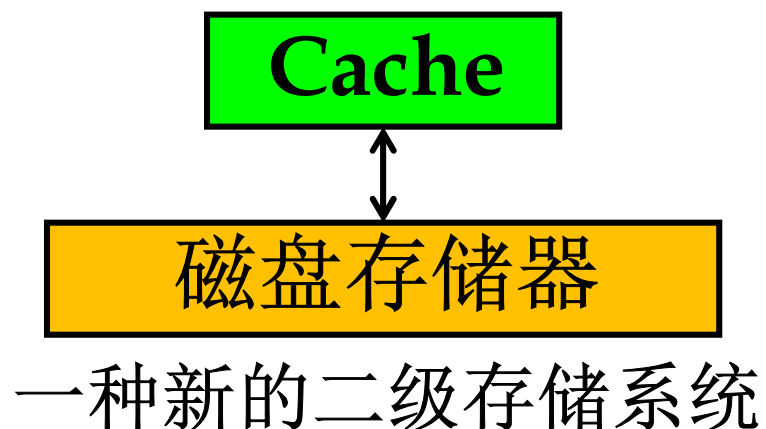
虚拟存储器采用位选择组相联映象和地址变换方式。



一种虚拟地址Cache的地址变换过程

全Cache系统

- 一种新的存储器组织方式。
- all-Cache
- 没有主存储器，只用Cache和磁盘（实际上只是磁盘中的一部分）两个存储器构成“Cache - 磁盘”存储系统。



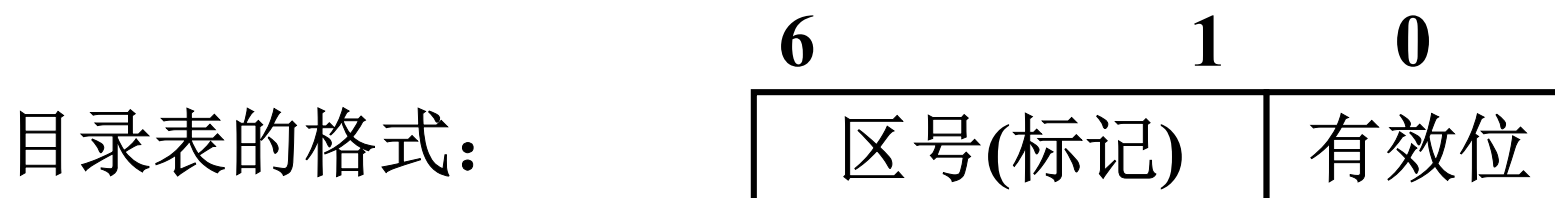
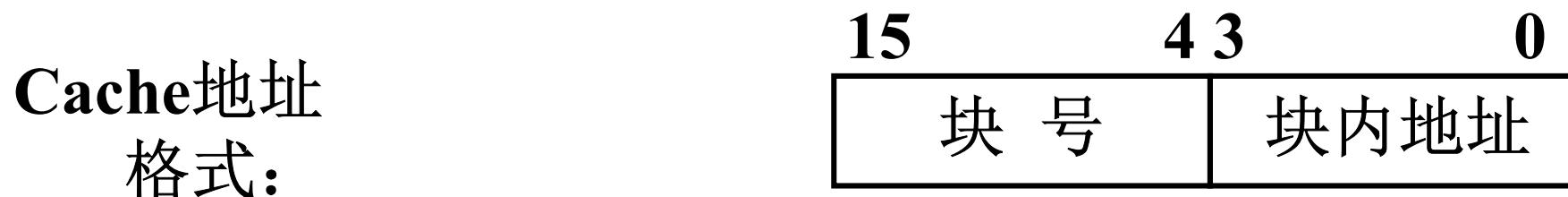
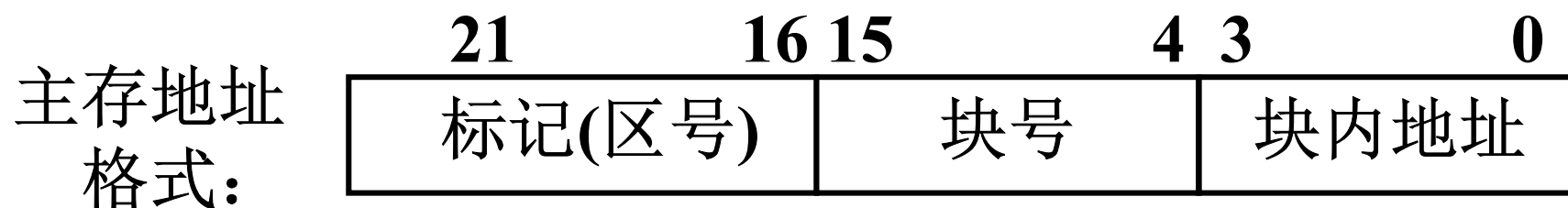
地址映象举例

■ 例1：假设某个计算机系统Cache容量为64K字节，数据块大小是16个字节，主存容量是4M，地址映象为直接映象方式。问：

- (1) 主存地址多少位？如何分配？
- (2) Cache地址多少位？如何分配？
- (3) 目录表的格式和容量？

地址映象举例

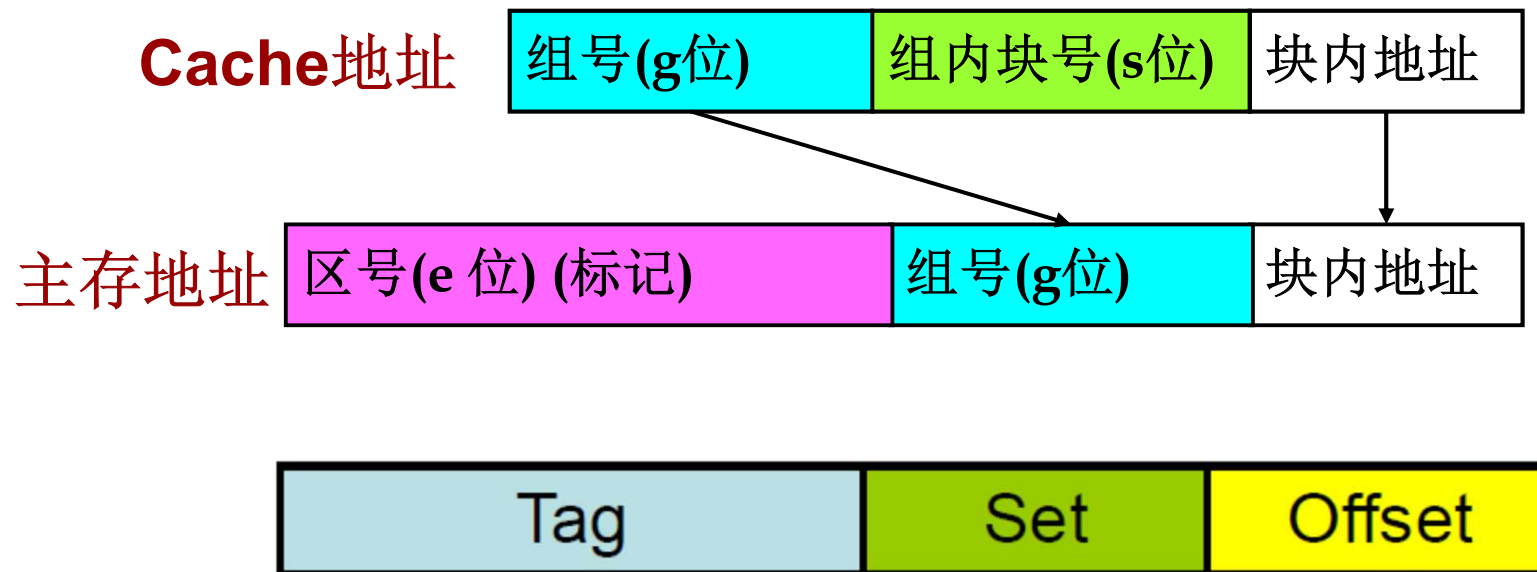
■ 例1：解：



目录表容量：与Cache块数量相同，即 $2^{12} = 4096$

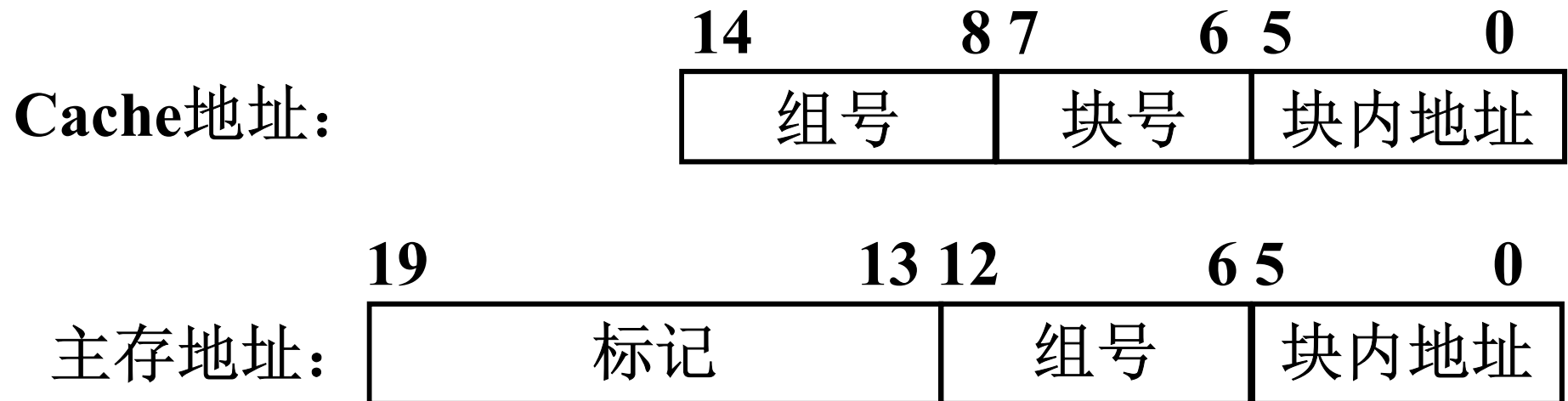
地址映象举例

- 例2：主存容量为1MB，Cache容量为32KB，每块为64个字节，共分128组。请写出主存与Cache的格式。



地址映象举例

■ 例2 解:



本章重点

- 并行存储器和交叉访问存储器的工作原理；
- 存储系统的定义；
- 存储系统的性能参数；
- 段式、页式、段页式虚拟存储管理的特点；
- 页式虚拟存储系统的工作原理；
- 虚拟存储系统的页面替换算法；
- 虚拟存储系统中加快地址变换的方法；
- Cache存储系统地址映像及变换方法；
- Cache存储系统的块替换算法；
- Cache存储系统的一致性问题；

本章重点

- 原理，特点，优缺点；
- 地址映像与变换（页式，Cache）；
- 失效（故障）与冲突；替换算法及实现（页式，Cache）；
- 命中率计算（页式，Cache）（地址流，映像，替换）；
- 加快地址变换的方法（页式，Cache）；
- 影响命中率的因素，提高命中率的方法（页式，Cache）；
- Cache一致性算法；

本章作业