

计算机组织结构

# 8 高速缓冲存储器 ( Cache )

刘博涵

2023年10月26日



南京大學  
NANJING UNIVERSITY

# 教材对应章节

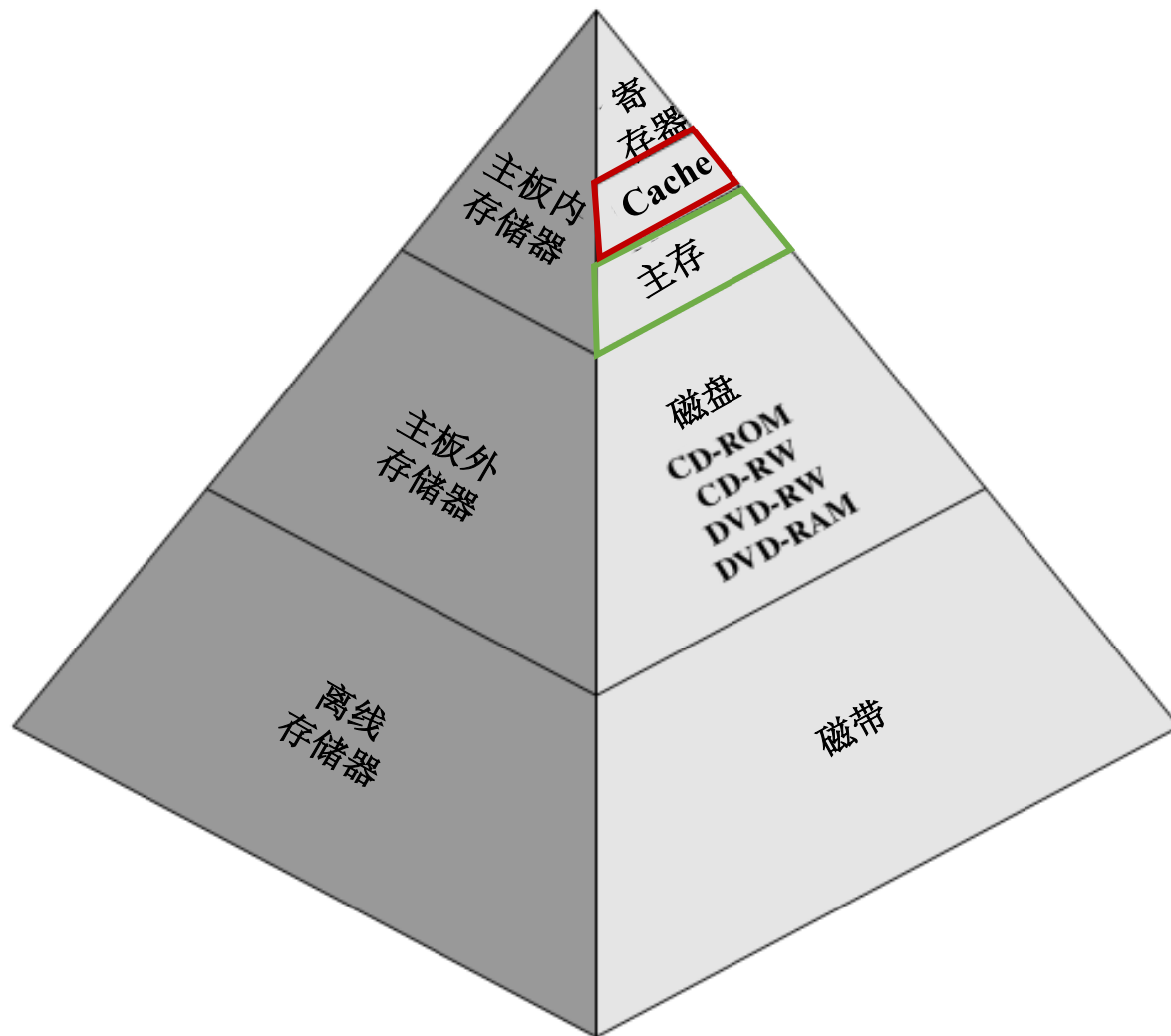


## 第7章 存储器分层体系结构



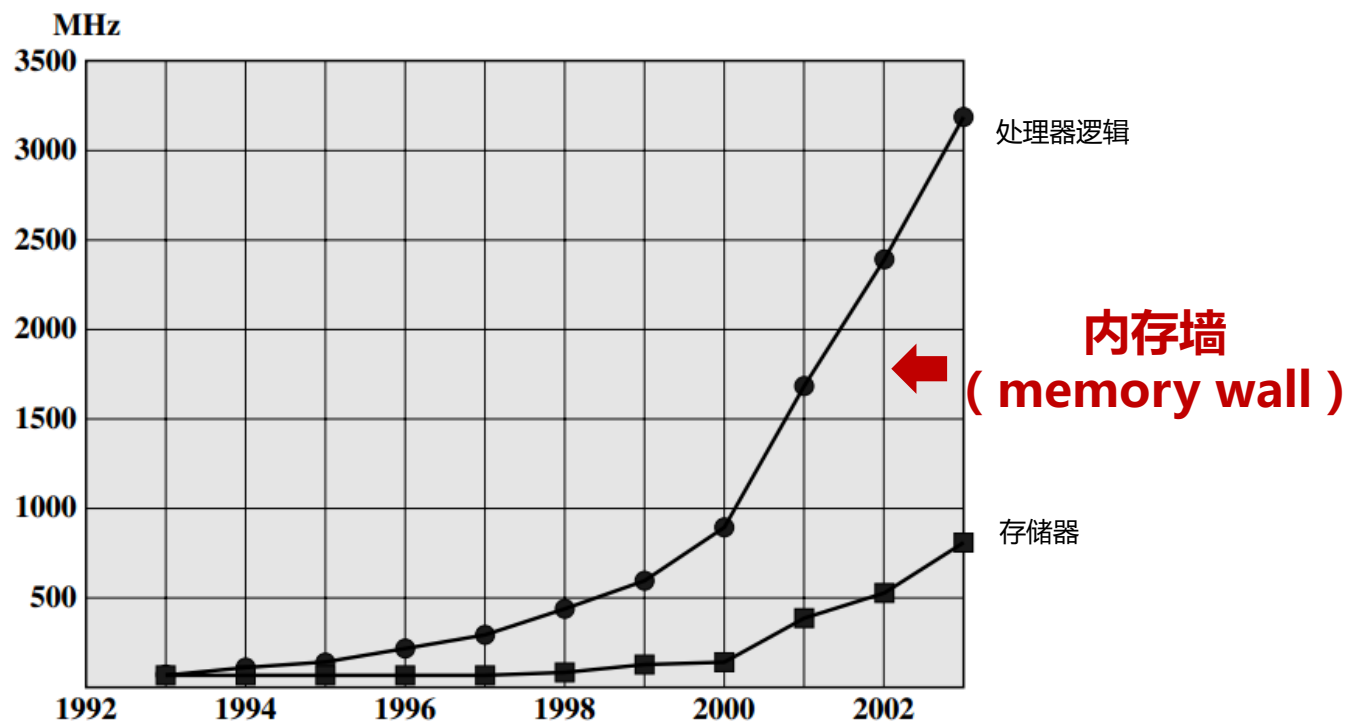
## 第4章 cache存储器

# 存储器层次结构






# 回顾：内存墙

- **问题**：CPU的速度比内存的速度快，且两者差距不断扩大



# 内存墙

- 解决：CPU和内存之间增加Cache

	Read 	Write 	Copy 	Latency
Memory	22861 MB/s	24157 MB/s	21629 MB/s	77.0 ns
L1 Cache	870.95 GB/s	442.51 GB/s	859.23 GB/s	1.2 ns
L2 Cache	183.62 GB/s	126.17 GB/s	160.59 GB/s	3.7 ns
L3 Cache	180.31 GB/s	128.01 GB/s	146.89 GB/s	16.2 ns
CPU Type	QuadCore Intel Core i7-4770 (Haswell-DT, LGA1150)			
CPU Stepping	C0			
CPU Clock	3793.6 MHz (original: 3400 MHz, overclock: 11%)			
CPU FSB	97.3 MHz (original: 100 MHz)			
CPU Multiplier	39x	North Bridge Clock		3793.6 MHz
Memory Bus	778.2 MHz	DRAM:FSB Ratio		24:3
Memory Type	Dual Channel DDR3-1556 SDRAM (11-11-11-28 CR2)			
Chipset	Intel Lynx Point B85, Intel Haswell			
Motherboard	Gigabyte GA-B85-HD3			
BIOS Version	F11			

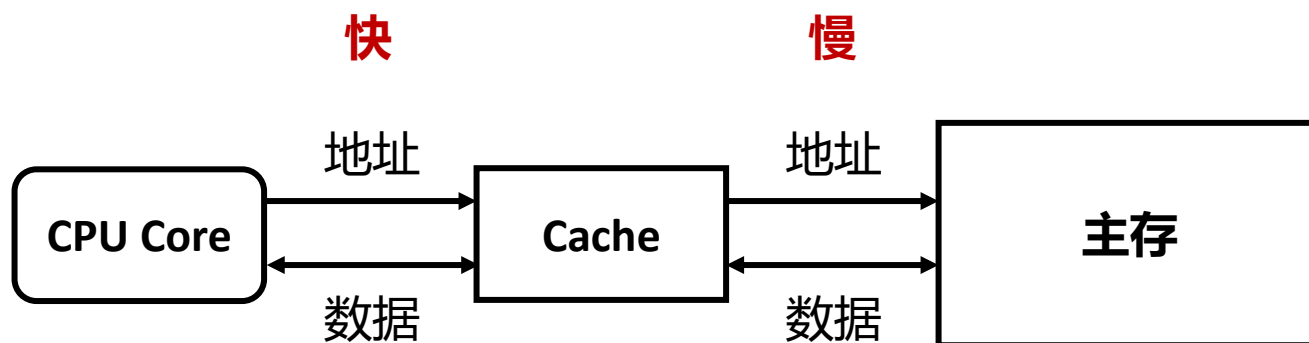
AIDA64 v6.25.5400 / BenchDLL 4.5.816.8-x64 (c) 1995-2020 FinalWire Ltd.

CPU时钟周期  
约等于0.26ns



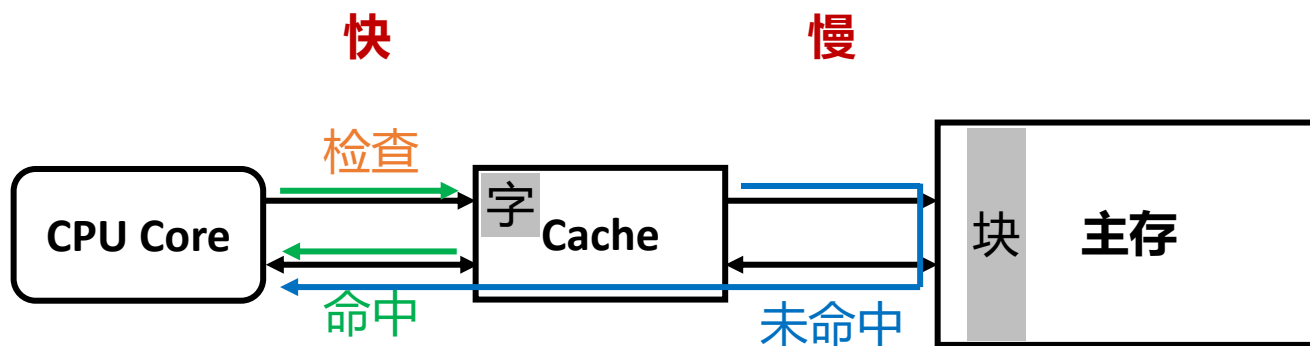
# Cache的基本思路

- 解决**内存墙**带来的CPU和主存协作问题
  - 在使用主存（相对大而慢）之余，添加一块小而快的cache
  - Cache位于CPU和主存之间，可以集成在CPU内部或作为主板上的一个模块
  - Cache中存放了主存中的部分信息的“副本”



# Cache的工作流程

- **检查 ( Check )** : 当CPU试图访问主存中的某个字时, **首先**检查这个字是否在cache中
- 检查后分两种情况处理 :
  - **命中 ( Hit )** : 如果在cache中, 则把**这个字**传送给CPU
  - **未命中 ( Miss )** : 如果不在cache中, 则将主存中包含这个字**固定大小的块 ( block )** 读入cache中, 然后再从cache**传送该字**给CPU



# 问题

- 如何判断是命中还是未命中?
- 如果未命中,为什么不直接把所需要的字从内存传送到CPU?
- 如果未命中,为什么从内存中读入一个块而不只读入一个字?
- 使用Cache后需要更多的操作,为什么还可以节省时间?





# 命中和未命中的判断

- 冯·诺伊曼体系的设计
  - CPU通过**位置**对主存中的内容进行寻址，不关心存储在其中的内容
- Cache通过**标记 ( tags )**来标识其内容在主存中的对应**位置**



# 问题

- 如何判断是命中还是未命中?
- 如果未命中,为什么不直接把所需要的字从内存传送到CPU?
- 如果未命中,为什么从内存中读入一个块而不只读入一个字?
- 使用Cache后需要更多的操作,为什么还可以节省时间?



# 程序访问的局部性原理

- 定义：
  - 处理器频繁访问主存中相同位置或者相邻存储位置的现象（ 维基百科 ）
- 类型
  - **时间局部性**：在相对较短的时间周期内，重复访问特定的信息（ 也就是访问**相同存储位置**的信息 ）
  - **空间局部性**：在相对较短的时间周期内，访问**相邻存储位置**的数据
    - **顺序局部性**：当数据被线性排列和访问时，出现的空间局部性的一种特殊情况
      - 例如：遍历一维数组中的元素



# 局部性原理的示例

- 时间局部性

```
int factorial = 1;  
for (int i = 2; i <= n; i++) {  
    factorial = factorial * i;  
}
```

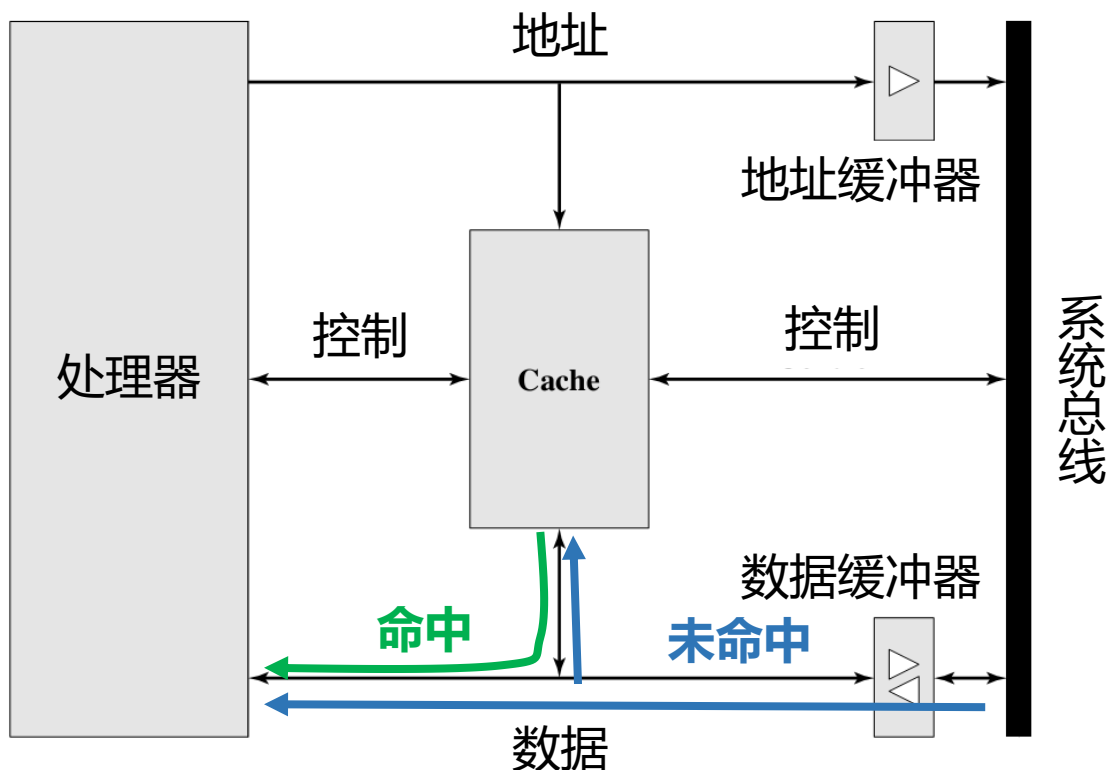


- 空间局部性

```
for (int i = 0; i < num; i++) {  
    score[i] = final[i] * 0.4 + midterm[i] * 0.3 + assign[i] * 0.3;  
}
```

# 向Cache传送内容

- 利用“时间局部性”
  - 将未命中的数据在返回给CPU的同时存放在Cache中，以便再次访问时命中

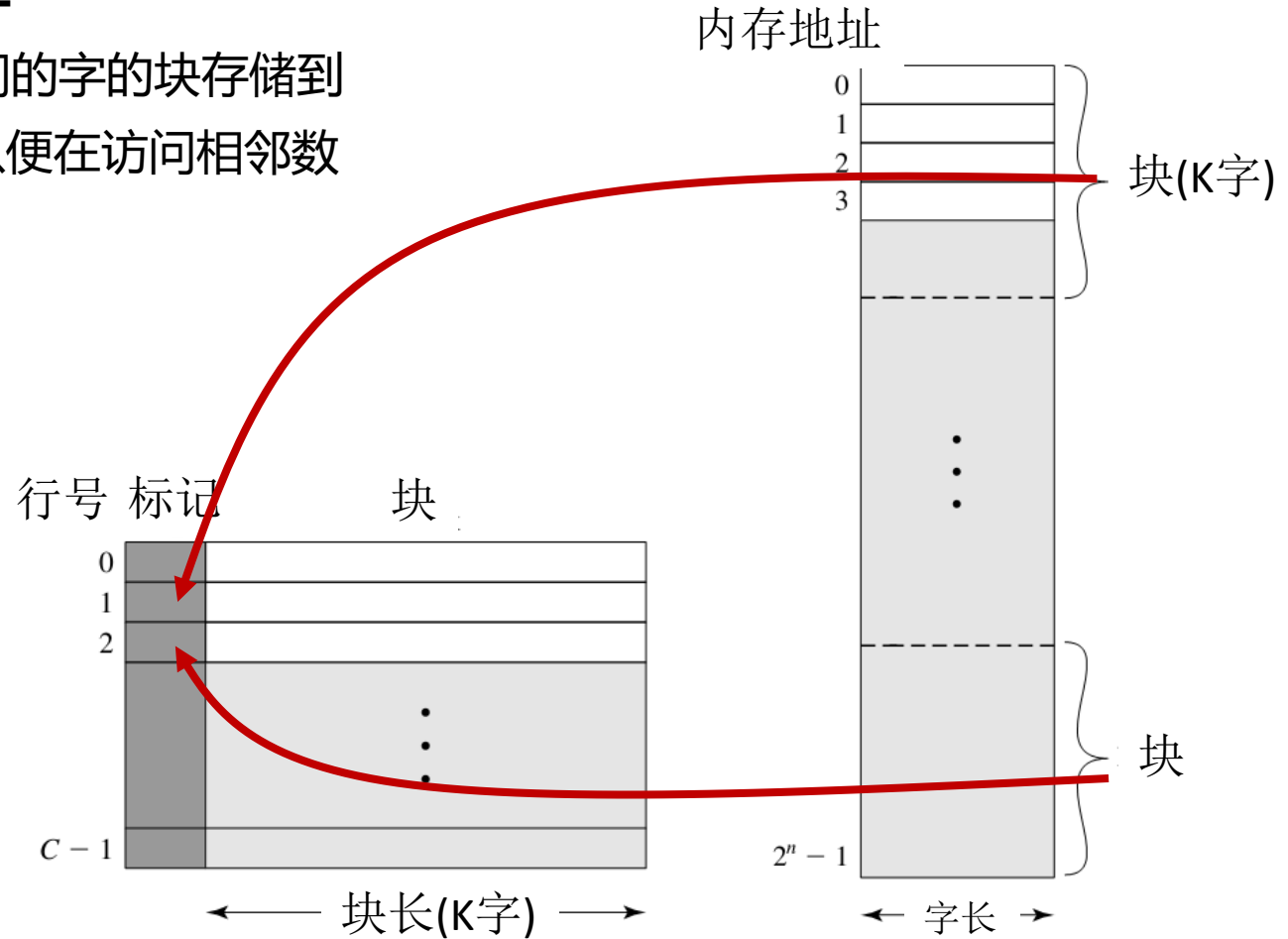


- 不同计算机结构是不同的
  - 数据缓冲器 -> Cache, 处理器
  - 数据缓冲器 -> Cache -> 处理器



# 传送块而不是传送字

- 利用“空间局部性”
  - 将包含所访问的字的块存储到Cache中，以便在访问相邻数据时命中



块在Cache中  $\rightarrow$  字在Cache中  $\Leftarrow$  块是预先划分好的



# 问题

- 如何判断是命中还是未命中?
- 如果未命中,为什么不直接把所需要的字从内存传送到CPU?
- 如果未命中,为什么从内存中读入一个块而不只读入一个字?
- 使用Cache后需要更多的操作,为什么还可以节省时间?



# 平均访问时间

- 假设 $p$ 是**命中率**， $T_C$ 是cache的访问时间， $T_M$ 是主存的访问时间，使用cache时的**平均访问时间** $T_A$ 为

$$\begin{aligned} T_A &= p \times T_C + (1 - p) \times (T_C + T_M) \\ &= T_C + (1 - p) \times T_M \end{aligned}$$

Cache返回数据的时间  
忽略不计

降低Cache  
访问时间

**矛盾**

提高  
命中率

降低主存  
访问时间  
(即降低未命中惩罚)

**很难**

- 命中率 $p$ 越大， $T_C$ 越小，效果越好
- 如果想要  $T_A < T_M$ ，必须要求

$$p > T_C / T_M$$

- 难点：cache的容量远远小于主存的容量（随机访问时 $p = \text{Cache容量} / \text{主存容量}$ ）





# Cache未命中原因

- **义务失效 ( Compulsory Miss ) / 冷启动失效 ( Cold Start Miss )**
  - 第一次访问一个块时
  - 例如：第一次访问一个数组，会发生义务失效
- **容量失效 ( Capacity Miss )**
  - Cache无法保存程序访问所需的所有数据块，则当某数据块被替换后，又重新被访问，则发生失效
  - 例如：cache大小为8KB，如果需要重复访问一个16KB大小的数组，数组无法全部放入cache，会发生容量失效
- **冲突失效 ( Conflict Miss )**
  - 多个存储器位置映射到同一Cache位置
  - 例如：有两个4KB大小的数组都映射到了相同的地址，需要来回访问，则发生冲突失效



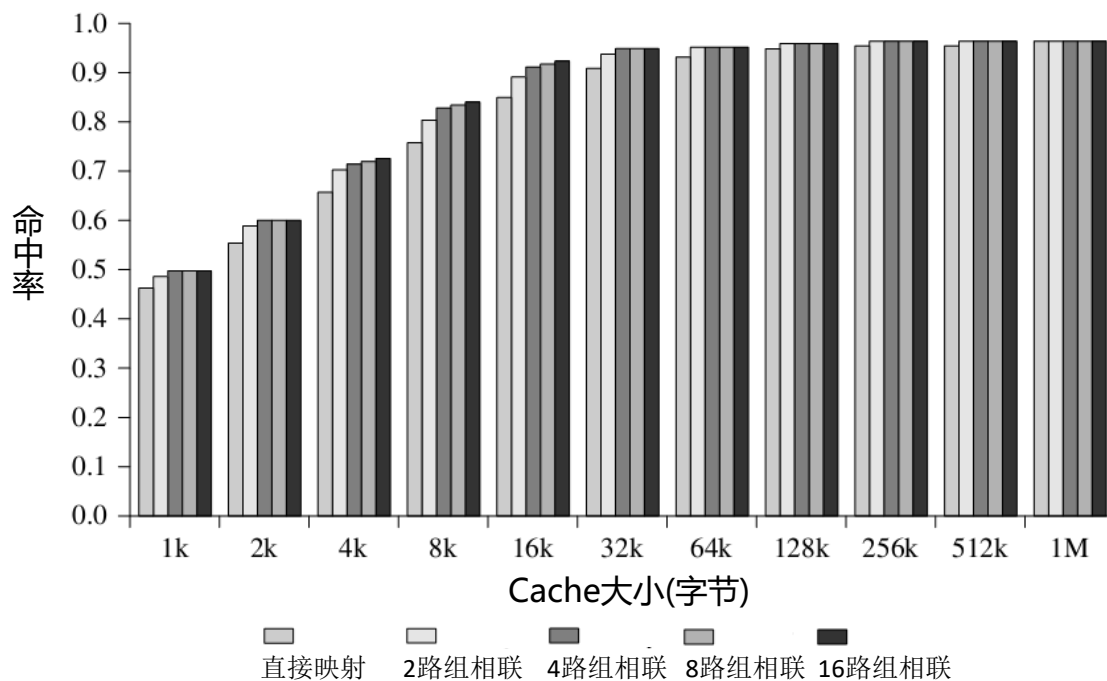
# Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- Cache行大小（即主存的块大小）
- Cache数目



# Cache容量

- 扩大cache容量带来的结果：
  - 增大了命中率 $p$
  - 增加了cache的开销和访问时间  $T_C$



**增大容量，避免容量失效**



# Cache容量

- 一些处理器的cache容量

处理器	类型	推出年份	L1 cache <sup>①</sup>	L2 cache	L3 cache
IBM 360/85	大型机	1968	16 ~ 32kB	—	—
PDP-11/70	小型机	1975	1kB	—	—
VAX 11/780	小型机	1978	16kB	—	—
IBM 3033	大型机	1978	64kB	—	—
IBM 3090	大型机	1985	128 ~ 256kB	—	—
Intel 80486	PC	1989	8kB	—	—
Pentium	PC	1993	8kB/8kB	256 ~ 512KB	—
PowerPC 601	PC	1993	32kB	—	—
PowerPC 620	PC	1996	32kB/32kB	—	—
PowerPC G4	PC/服务器	1999	32kB/32kB	256KB ~ 1MB	2MB
IBM S/390 G4	大型机	1997	32kB	256KB	2MB
IBM S/390 G6	大型机	1999	256kB	8MB	—
Pentium 4	PC/服务器	2000	8kB/8kB	256KB	—
IBM SP	高端服务器/超级计算机	2000	64kB/32kB	8MB	—
CRAY MTA <sup>②</sup>	超级计算机	2000	8kB	2MB	—
Itanium	PC/服务器	2001	16kB/16kB	96KB	4MB



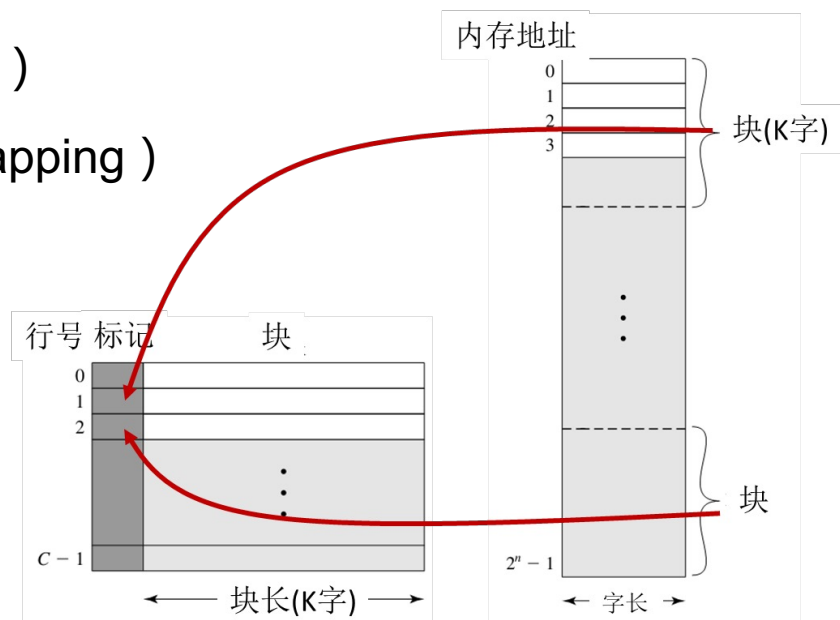
# Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目



# 映射功能 ( Mapping Function )

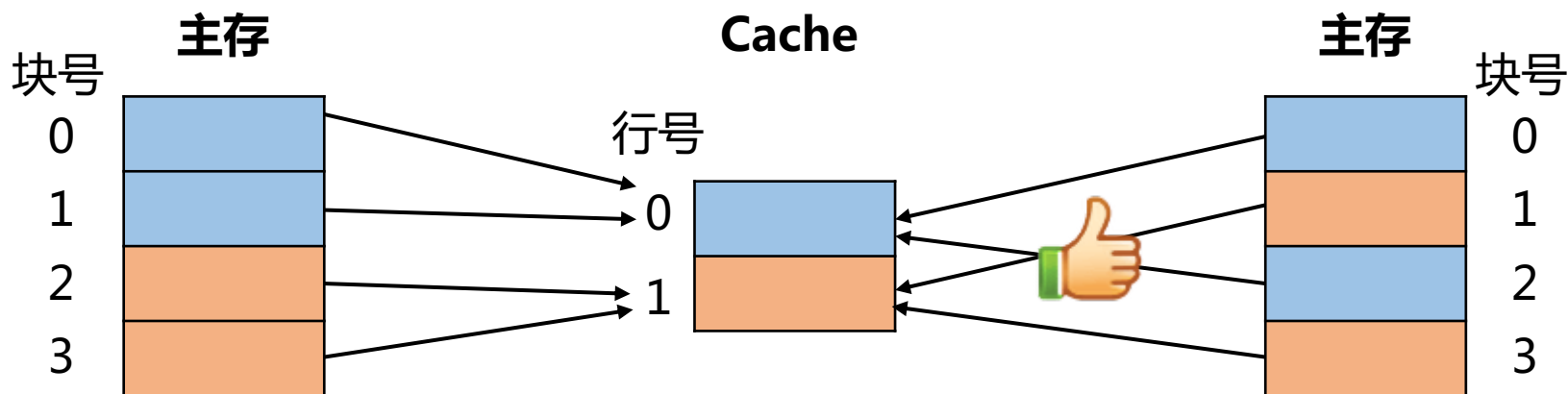
- 实现主存块到cache行的映射
- 块号，块内地址
- 映射方式的选择会影响cache的组织结构
  - 直接映射 ( Direct mapping )
  - 关联映射 ( Associative mapping )
  - 组关联映射 ( Set associative mapping )



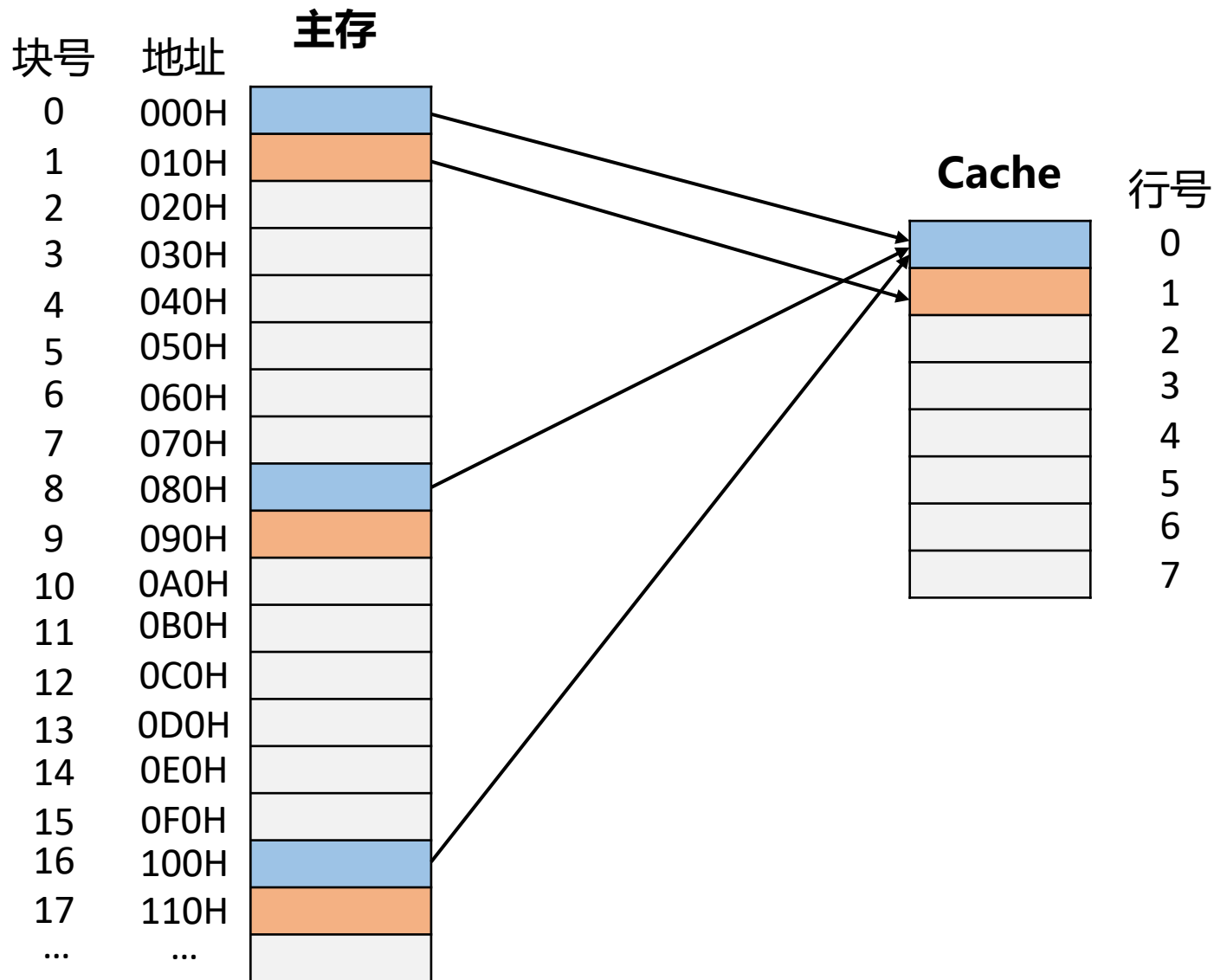
# 直接映射

- 将主存中的每个块映射到**一个固定可用的**cache行中
- 假设 $i$ 是cache行号,  $j$ 是主存储器的块号,  $C$ 是cache的行数

$$i = j \bmod C$$



# 直接映射：示例





# 直接映射

- (块) 标记

- 地址中最高 $n$ 位,  $n = \log_2 M - \log_2 C$   $i = j \bmod C$   
块的数量                  行的数量

同行号则低  
 $\log_2 C$  位相同

主存地址

标记	Cache行号	块内地址
----	---------	------

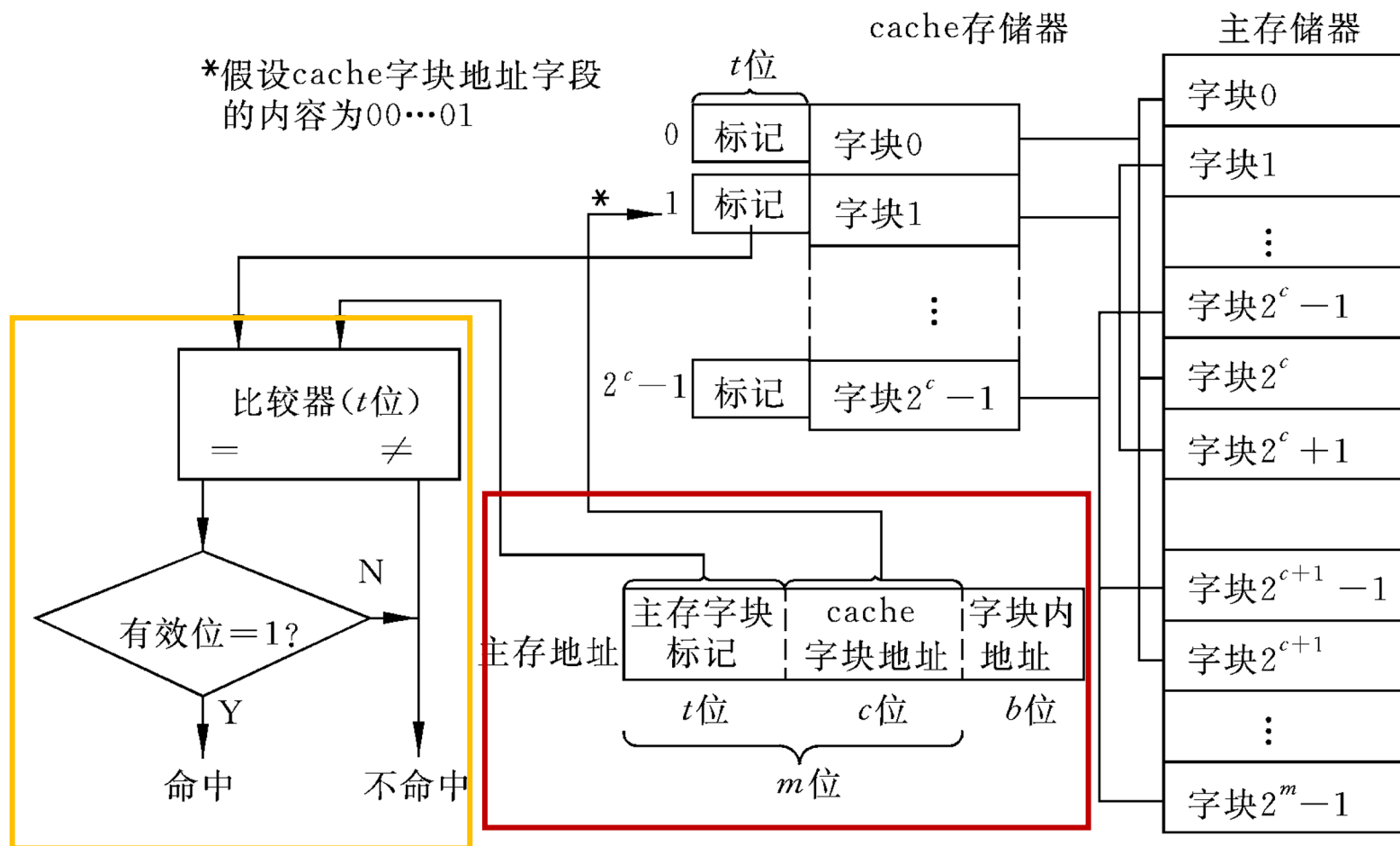
- 例

- 假设cache有4行，每行包含8个字；主存中包含128个字。访问主存的地址长度为7位，则：
  - 最低的3位：块内地址
  - 中间的2位：映射时所对应的Cache行号
  - 最高的2位：区分映射到同一行的不同块，记录为Cache标记



# 直接映射

\*假设cache字块地址字段的内容为00...01



# 直接映射：读数据示例

- MOV AL, [2011H] 未命中，读主存地址2010H，映射Cache行1，读出A1H
- MOV AL, [3012H]
- MOV AL, [301FH]

行号	有效位	标记	数据					
0	0							
1	1	20H	A0H	A1H	A2H	A3H	...	AFH
3	0							
4	0							
...	0							
15	0							



# 直接映射：读数据示例

- MOV AL, [2011H]
- MOV AL, [3012H] 未命中，读主存地址3010H，映射Cache行1，读出C2H
- MOV AL, [301FH]

行号	有效位	标记	数据					
0	0							
1	1	30H	C0H	C1H	C2H	C3H	...	CFH
3	0							
4	0							
...	0							
15	0							



# 直接映射：读数据示例

- MOV AL, [2011H]
- MOV AL, [3012H]
- MOV AL, [301FH] 命中, 读出CFH

行号	有效位	标记	数据					
0	0							
1	1	30H	C0H	C1H	C2H	C3H	...	CFH
3	0							
4	0							
...	0							
15	0							



# 直接映射

- **优点**

- 简单
- 快速映射
- 快速检查

- **缺点**

- 抖动现象（Thrashing）：如果一个程序重复访问两个需要映射到同一行中且来自不同块的字，则这两个块不断地被交换到cache中，cache的命中率将会降低，即发生冲突失效

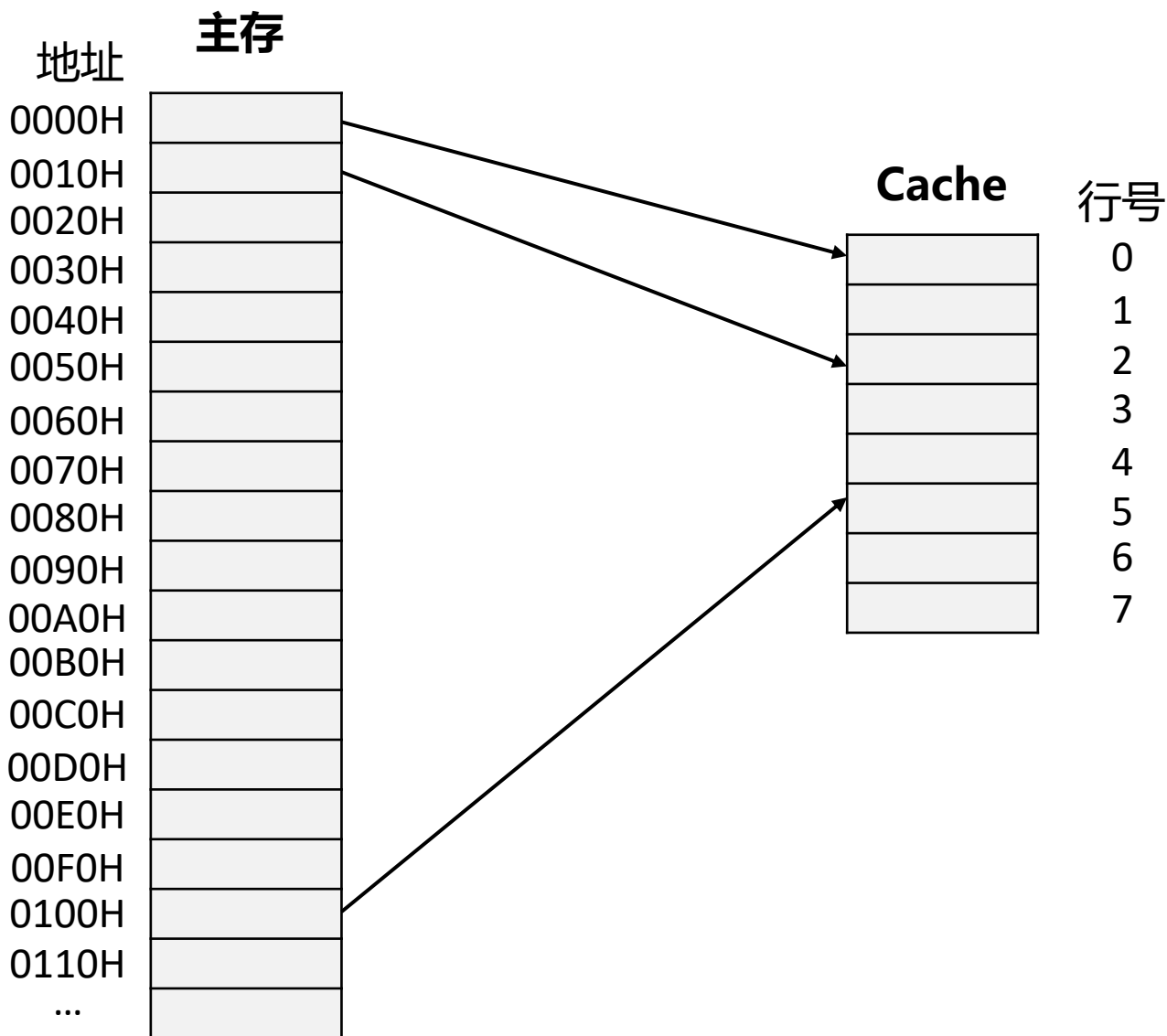
- **适合大容量的cache**

- 行数变多，发生冲突失效的概率降低
- 硬件电路简单，增大容量对 $T_c$ 的影响不明显



# 关联映射（全相联映射）

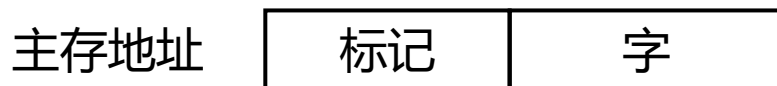
- 一个主存块可以装入cache**任意一行**



# 关联映射

- (块) 标记

- 地址中最高 $n$ 位,  $n = \log_2 M$



- 示例

- 假设cache有4行，每行包含8个字；主存中包含128个字。访问主存的地址长度为7位，则：
  - 最低的3位：块内地址
  - 最高的4位：块号，记录为Cache标记





# 关联映射

- 优点
  - 避免抖动
- 缺点
  - 实现起来比较复杂
  - Cache搜索代价很大，即在检查的时候需要去访问cache的每一行
- 适合容量较小的cache
  - 小容量更容易发生冲突失效
  - 小容量检查的时间短



# 组关联映射（组相联映射）

- Cache分为若干组，每一组包含相同数量的行，每个主存块被映射到**固定组的任意一行**

- 假设 $s$ 是cache组号， $j$ 是主存块号， $S$ 是组数

$$s = j \bmod S$$

- K-路**组关联映射

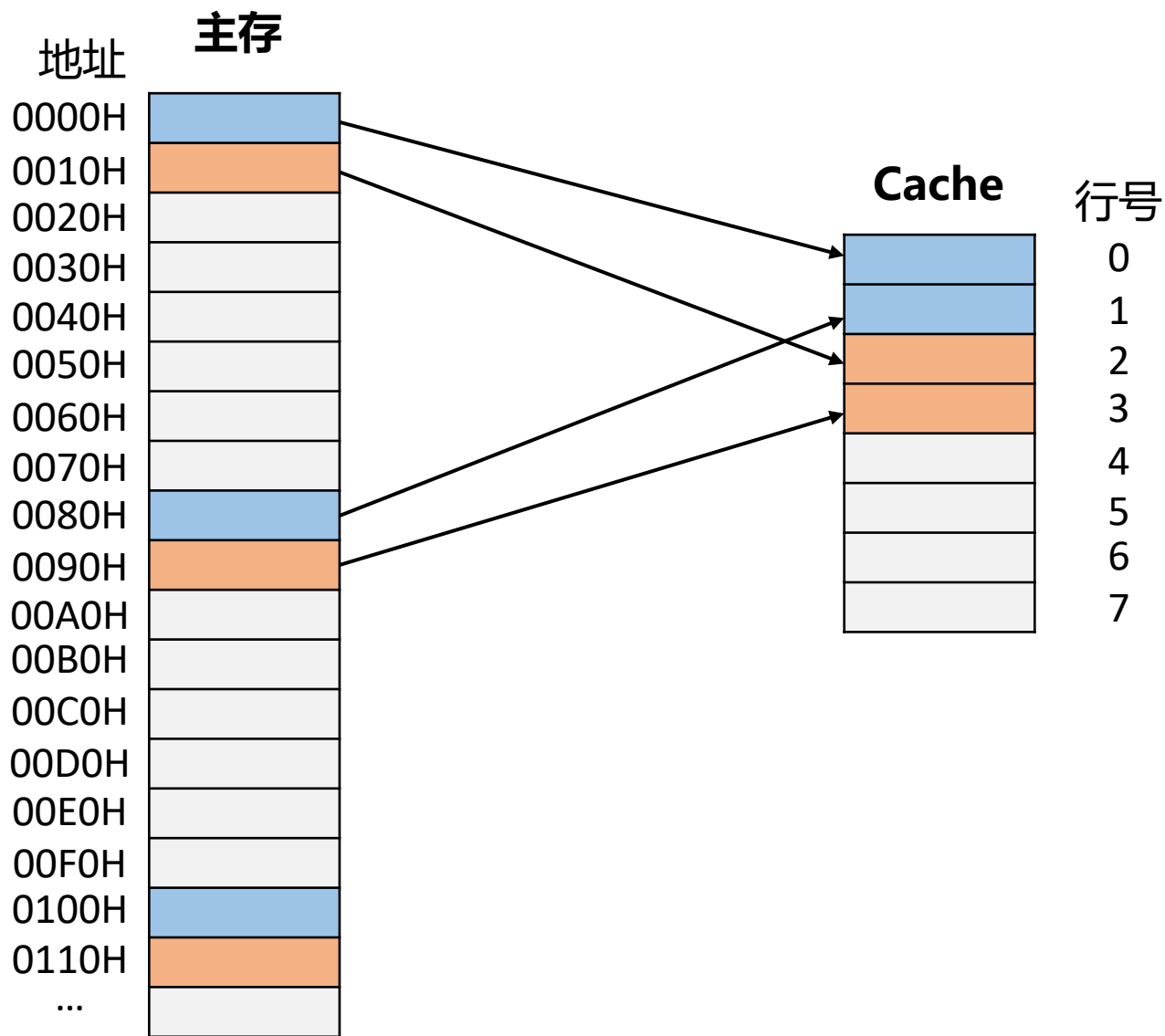
$$K = C/S$$

↓      ↓

行数   组数



# 组关联映射：示例



# 组关联映射

- 标记
  - 地址中最高 $n$ 位,  $n = \log_2 M - \log_2 S$



- 示例
  - 假设cache有4行, 每行包含8个字, 分成2个组; 主存中包含128个字。访问主存的地址长度为7位, 则:
    - 最低的3位: 块内地址
    - 中间的1位: 映射时所对应的Cache中的组
    - 最高的3位: 区分映射到同一组的不同块, 记录为Cache标记



# 组关联映射

- **优点**
  - 结合了直接映射和关联映射的优点
- **缺点**
  - 结合了直接映射和关联映射的缺点
- **面向不同容量的cache做了折中**



# 三种映射方式比较

- 三种方式的相关性
  - 如果  $K = 1$  , 组关联映射等同于直接映射
  - 如果  $K = C$  , 组关联映射等同于关联映射



# 三种映射方式比较

- 关联度 ( Correlation ) : 一个主存块映射到cache中可能存放的位置个数
  - 直接映射 : 1
  - 关联映射 :  $C$
  - 组关联映射 :  $K$
- 关联度越低, 命中率越低
  - 直接映射的命中率最低, 关联映射的命中率最高
- 关联度越低, 判断是否命中的时间越短
  - 直接映射的命中时间最短, 关联映射的命中时间最长
- 关联度越低, 标记所占额外空间开销越小
  - 直接映射的标记最短, 关联映射的标记最长



# Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目





# 替换算法

- 一旦cache行被占用，当新的数据块装入cache中时，原先存放的数据块将会被替换掉
- 对于直接映射，每个数据块都只有唯一对应的行可以放置，没有选择的机会
- 对于关联映射和组关联映射，每个数据块被允许在多个行中选择一个进行放置，就需要替换算法来决定替换哪一行中的数据块
  - 采用替换算法的情景是，多行可选且多行均被占
  - 替换算法通过硬件来实现
  - 设计替换算法的目标是提高命中率



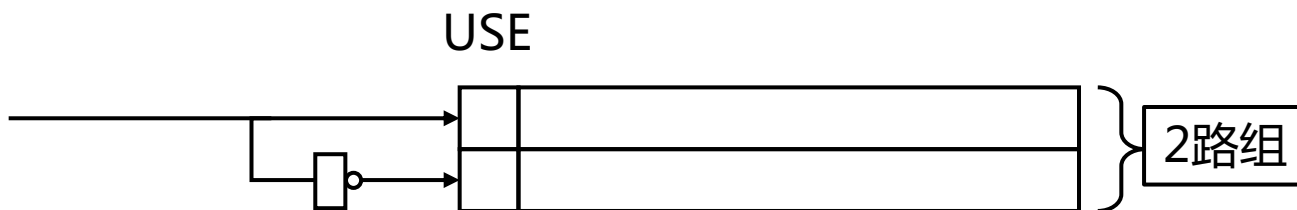
# 常用的替换算法

- 最近最少使用算法 ( Least Recently Used, LRU )
- 先进先出算法 ( First In First Out, FIFO )
- 最不经常使用算法 ( Least Frequently Used, LFU )
- 随机替换算法 ( Random )



# 最近最少使用算法（LRU）

- **假设**：最近使用过的数据块更有可能会被再次使用
- **策略**：替换掉在cache中**最长时间未被访问**的数据块
- **实现**：对于2路组关联映射
  - 每行包含一个USE位
  - 当同一组中的某行被**访问**时，将其USE位设为1，同时将另一行的USE位设为0
  - 当将新的数据块读入该组时，替换掉USE位为0的行中的数据块



# 最近最少使用算法（LRU）：示例

例：假设cache有2组，每组4行，每行8个字，主存地址长度为8位

主存地址    00000000    10000010    01000001    11000100    00000000    11100011

Cache

LRU	标记
0	0000

未命中  
载入

LRU	标记
1	0000
0	1000

未命中  
载入

LRU	标记
2	0000
1	1000
0	0100

未命中  
载入

LRU	标记
3	0000
2	1000
1	0100
0	1100

未命中  
载入

LRU	标记
0	0000
3	1000
2	0100
1	1100

命中

LRU	标记
1	0000
0	1110
3	0100
2	1100

未命中  
替换

- LRU位需要额外的**硬件实现**

- $K$ 路需要 $\log(K!) \approx K \cdot \log(K)$ 位，例如4路时每行需要2位，共需8位

- LRU会增加cache**访问时间**



# 最近最少使用算法（LRU）：示例

例（续）：抖动现象

访问顺序 4H , 12H , 0H , 14H , 10H , 4H , 12H , 0H , 14H

主存地址 11100011 10100010 01000001 11000100 00000000 11100011

Cache

LRU	标记
1	0000
0	1110
3	0100
2	1100

未命中  
替换

LRU	标记
2	0000
1	1110
0	1010
3	1100

未命中  
替换

LRU	标记
3	0000
2	1110
1	1010
0	0100

未命中  
替换

LRU	标记
0	1100
3	1110
2	1010
1	0100

未命中  
替换

LRU	标记
1	1100
0	0000
3	1010
2	0100

未命中  
替换

LRU	标记
2	1100
1	0000
0	1110
3	0100

未命中  
替换

- LRU的命中率可能也并没有想象中那么高



# 先进先出算法 ( FIFO )

- **假设**：最近由主存载入Cache的数据块更有可能被使用
- **策略**：替换掉在Cache中**停留时间**最长的块
- **实现**：时间片轮转法 或 环形缓冲技术
  - 每行包含一个标识位
  - 当同一组中的某行被**替换**时，将其标识位设为1，同时将其**下一行**的标识位设为0
    - 如果被**替换**的是该组中的最后一行，则将该组中的第一行的标识位设为0
  - 当将新的数据块读入该组时，替换掉标识位为0的行中的数据块



# 先进先出算法（FIFO）：示例

例：假设cache有2组，4行，每行8个字，主存地址长度为8位

主存地址	当前	11100010	01000001	00000000	10000000	10100011
Cache	标识	标识	标识	标识	标识	标识
	标记	标记	标记	标记	标记	标记
	1	0000	1	1110	1	1110
	1	1000	0	1000	1	0000
	1	0100	1	0100	1	1000
	1	1100	1	1100	0	1100
			未命中 替换	命中	未命中 替换	未命中 替换

- FIFO每行1位标识位，需要额外的**硬件实现**
- FIFO仅当替换时，需要改变标识位



# 最不经常使用算法 ( LFU )

- 假设：访问越频繁的数据块越有可能被再次使用
- 策略：替换掉cache中被访问次数最少的数据块
- 实现：为每一行设置计数器





# 随机替换算法 ( Random )

- 假设：每个数据块被再次使用的可能性是相同的
- 策略：随机替换cache中的数据块
- 实现：随机替换
- 随机替换算法在性能上只稍逊于使用其它替换算法



# Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目



# 缓存命中时的写策略

- **主存和cache的一致性**
  - 当cache中的某个数据块被替换时，需要考虑该数据块是否被修改
- **替换时面临的两种情况**
  - 如果没被修改，则该数据块可以直接被替换掉
  - 如果被修改，则在替换掉该数据块之前，必须将修改后的数据块写回到主存中对应位置
- **写策略**
  - 写直达 ( Write Through )
  - 写回法 ( Write Back )



# 缓存命中时的写策略：写直达

- 所有写操作都**同时**对cache和主存进行
- 优点
  - 确保主存中的数据总是和cache中的数据一致，总是最新的（例如多CPU同步的场景）
- 缺点
  - 产生大量的主存访问，减慢写操作



# 缓存命中时的写策略：写回法

- 先更新cache中的数据，当cache中某个数据块被替换时，如果它被修改了，才被写回主存
- 利用一个脏位 ( dirty bit ) 或者使用位 ( use bit ) 来表示块是否被修改
- 优点
  - 减少了访问主存的次数
- 缺点
  - 部分主存数据可能不是最新的 ( 例如未发生替换但需要读主存的场景 )
    - I/O模块存取时可能无法获得最新的数据，为解决该问题会使得电路设计更加复杂且有可能带来性能瓶颈



# 缓存未命中时的写策略

- **写不分配 ( Write Non-Allocate )** : 直接将数据写入主存, 无需读入cache
  - 优点: 避免cache和主存中的数据不一致
  - 通常搭配: 写直达
- **写分配 ( Write Allocate )** : 将数据所在的块读入cache后, 在cache中更新内容
  - 优点: 利用了cache的高速特性, 减少写内存次数
  - 通常搭配: 写回法



# Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目



# 行大小

- 假设从行的大小为一个字开始，随着**行大小的逐步增大**，则**Cache命中率会增加**
  - 数据块中包含了更多周围的数据，每次会有更多的数据作为一个块装入cache中
  - 利用了**空间局部性**
- 当行大小变得较大之后，**继续增加行大小**，则**Cache命中率会下降**
  - 当Cache容量一定的前提下，较大的行会导致Cache中的**行数变少**，导致装入cache中的数据块数量减少，进而造成数据块被频繁替换
  - 每个数据块中包含的数据在主存中位置变远，被使用的可能性减小
- 行大小与命中率之间的关系较为复杂
  - 行太小，**行数太多反时间局部性**
  - **行太大**，**行数太少反空间局部性**





# Cache的设计要素

- Cache容量
- 映射功能
- 替换算法
- 写策略
- 行大小
- Cache数目



# Cache数目：一级 vs. 多级

- **一级**

- 将cache与处理器置于同一芯片（片内cache）
- 减少处理器在外部总线上的活动，从而减少了执行时间

- **多级**

- 当L1未命中时，减少处理器对总线上DRAM或ROM的访问
- 使用单独的数据路径，代替系统总线在L2缓存和处理器之间传输数据，部分处理器将L2 cache结合到处理器芯片上



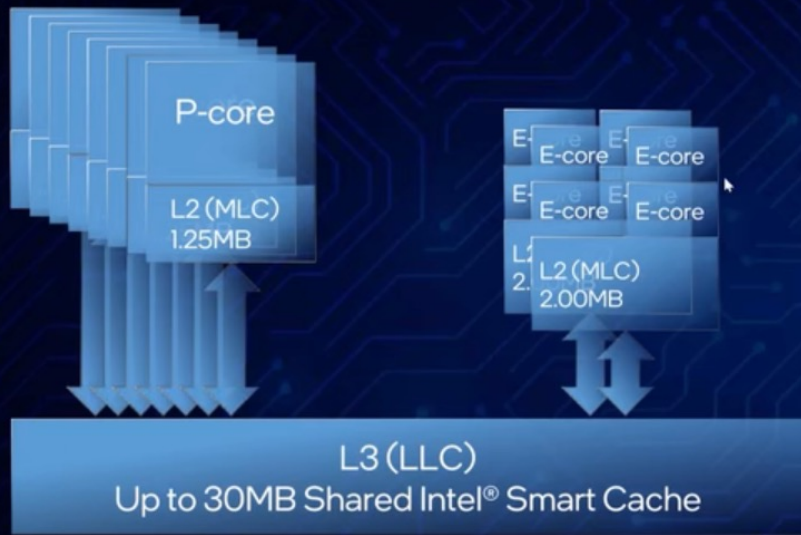
# Cache数目：统一 vs. 分立

- 统一
  - 更高的命中率，在获取指令和数据的负载之间自动进行平衡
  - 只需要设计和实现一个cache **指令和数据的局部性会相互影响**
- 分立
  - 消除cache在指令的取值/译码单元和执行单元之间的竞争，在任何基于指令流水线的设计中都是重要的



# 最新的Cache架构

## 12th Gen Intel® Core™ Cache Architecture



### NEW Cache Architecture

- Common Intel® Smart Cache shared among P-cores, E-cores and processor graphics
- Increased L2 Cache per P-Core
- L2 Cache shared amongst each E-Core cluster

- 8个性能核 ( P-Core ) + 8个能效核 ( E-Core )
- 每个P-Core独享一个L2 , 每4个E-Core共享一个L2
- 所有核共享一个L3

# 总结

- Cache的目的、基本思路、工作流程
- Cache的若干问题
  - 命中 vs. 未命中
  - 未命中时将数据块传送到Cache中
  - 平均访问时间
- Cache的设计要素
  - Cache容量
  - 映射功能：直接映射，关联映射，组关联映射
  - 替换算法：LRU，FIFO，LFU，随机
  - 写策略：写直达，写回法，写分配，写不分配
  - 行大小
  - Cache数目：一级vs.多级，统一vs.分立



# 谢谢

bohanliu@nju.edu.cn



南京大學  
NANJING UNIVERSITY