计算机组成原理

Cache存储器

王浩宇,教授

haoyuwang@hust.edu.cn

https://howiepku.github.io/

本节目录

- **Cache存储器**
  - Cache的基本原理
  - Cache的命中率
  - 主存与Cache的地址映射
  - Cache替换策略
  - Cache写操作策略

回顾：这两个程序性能是否一样？

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

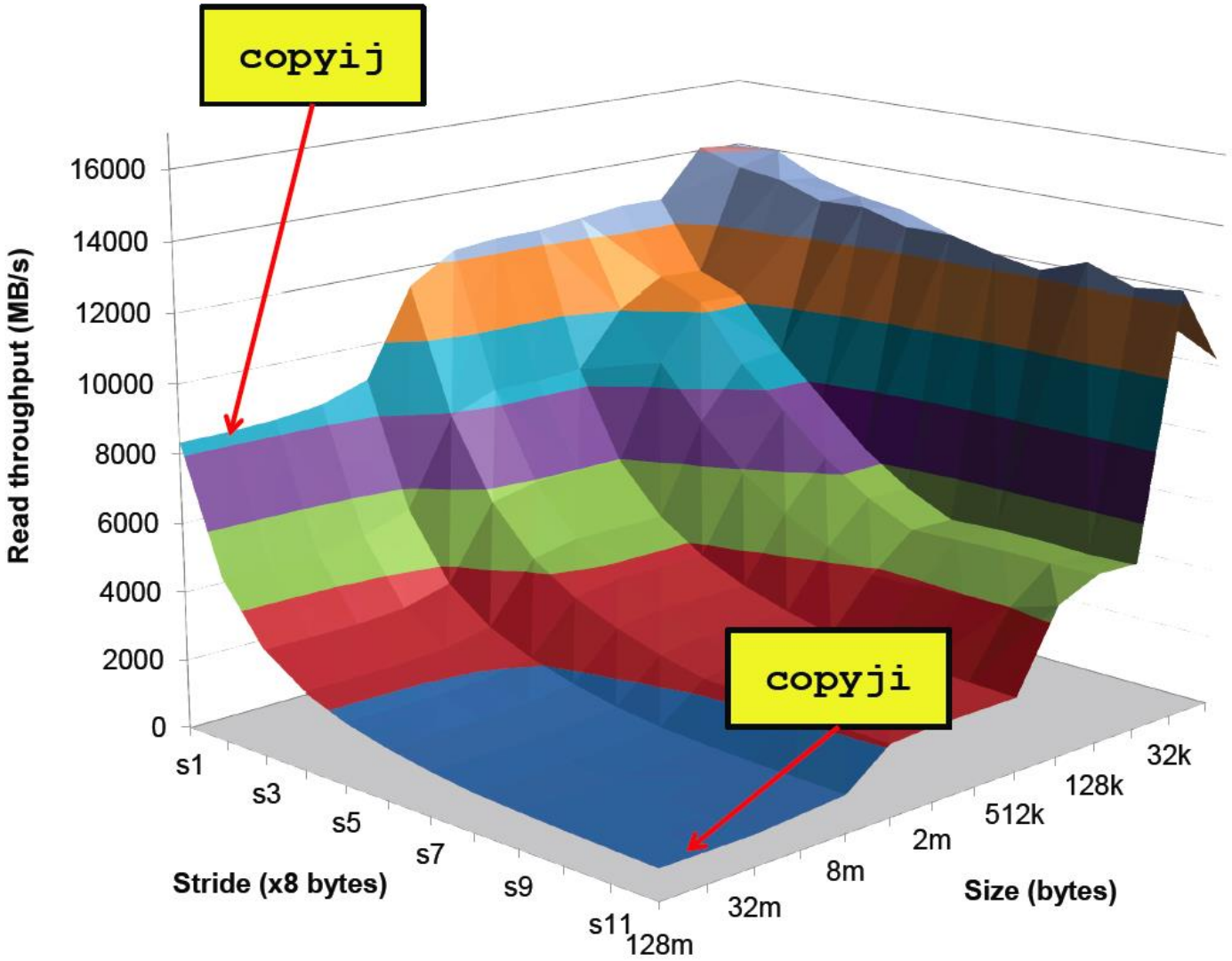
```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

4.3ms　　81.8ms

2.0 GHz Intel Core i7 Haswell

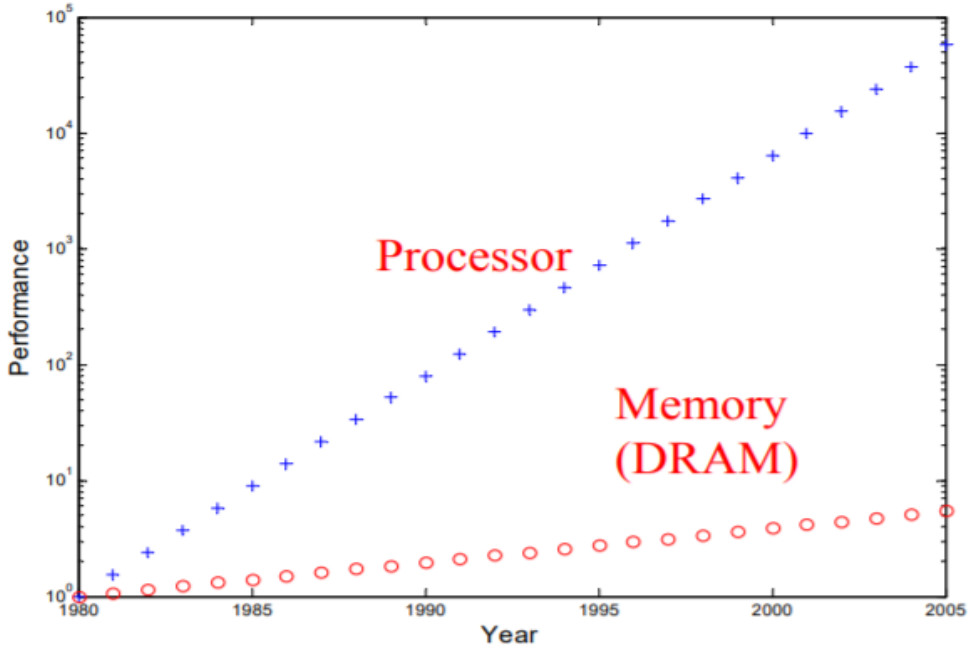
- **计算机系统中的分级存储结构**
  - CPU的Cache机制
- **程序的性能与对数据的访问模式十分相关**
  - 空间局部性

为什么性能差别这么大?


copyij
copyji
Read throughput (MB/s)
16000
14000
12000
10000
8000
6000
4000
2000
0
s1
s3
s5
s7
s9
s11
Stride (x8 bytes)
32k
128k
512k
2m
8m
32m
128m
Size (bytes)

为什么要引入Cache?

- （1） CPU和主存之间的性能差距

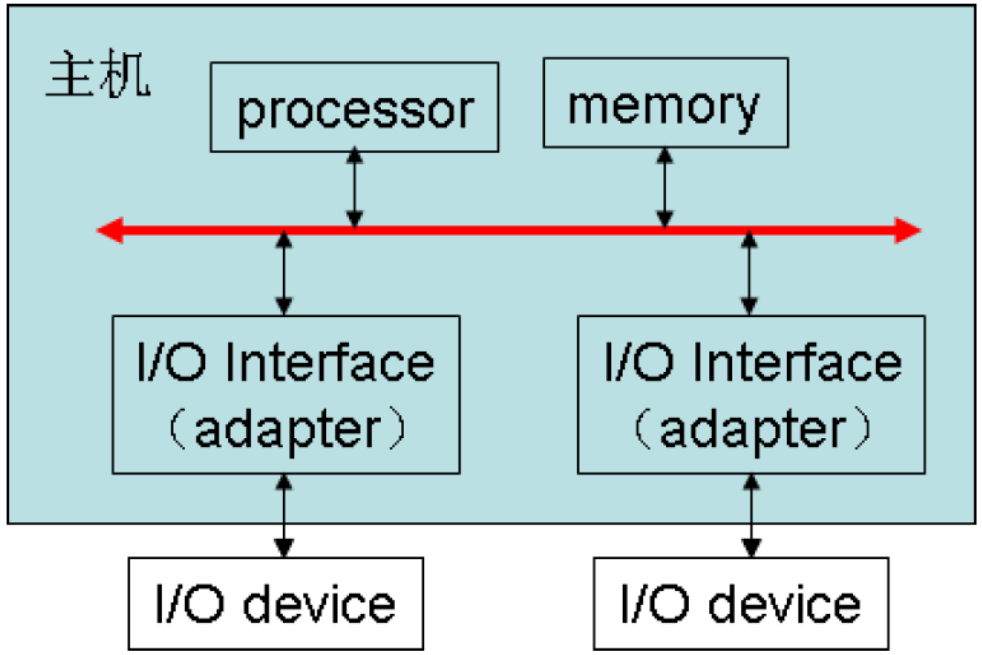


CPU的性能和主存的性能之间，存在差异，且越来越大
如何解决?

（1） 并行技术（双端口存储器，多体交叉存储器）
（2） 采用更高速的技术来缩短读出时间 → Cache

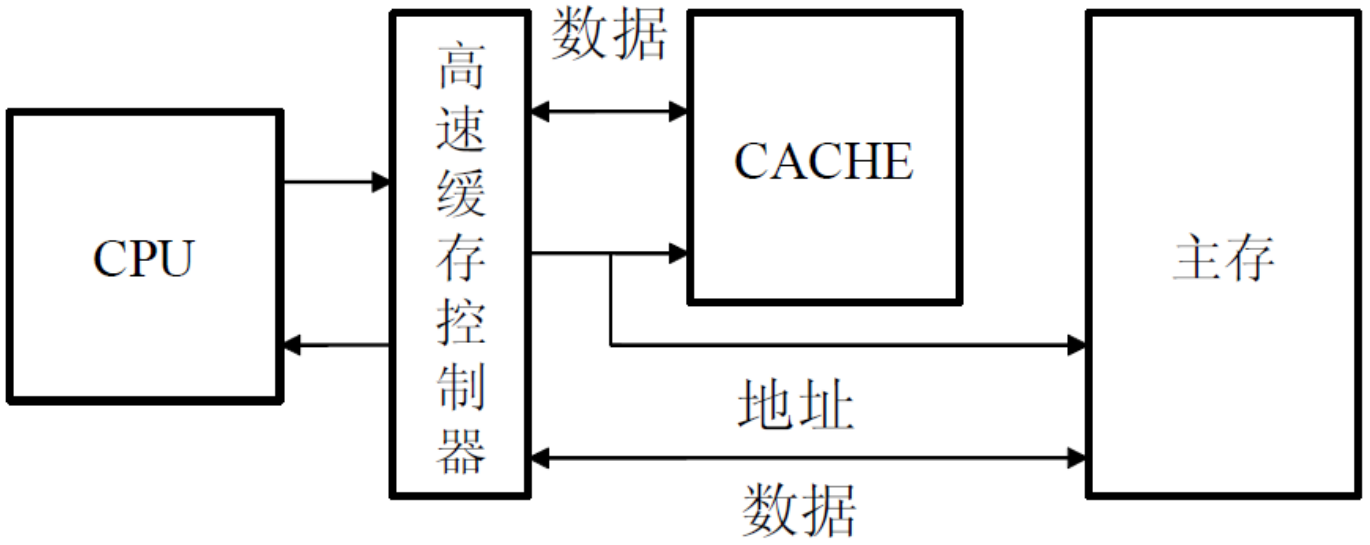
为什么要引入Cache?

- **（2）结构冲突**
  - 总线占用：CPU和I/O争抢访问主存
  - 访存冲突：指令预取和数据读写（后续流水CPU涉及到）

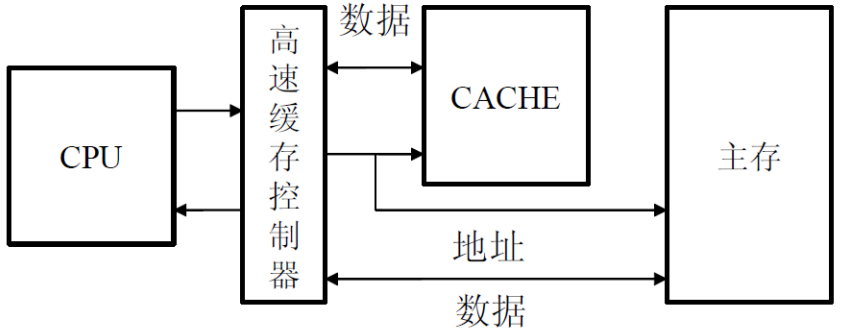


Cache的作用

- **CPU不直接访问主存，只与高速Cache交换信息**
  - Cache的速度比主存快
  - 让出总线



Cache基本原理（1/3）



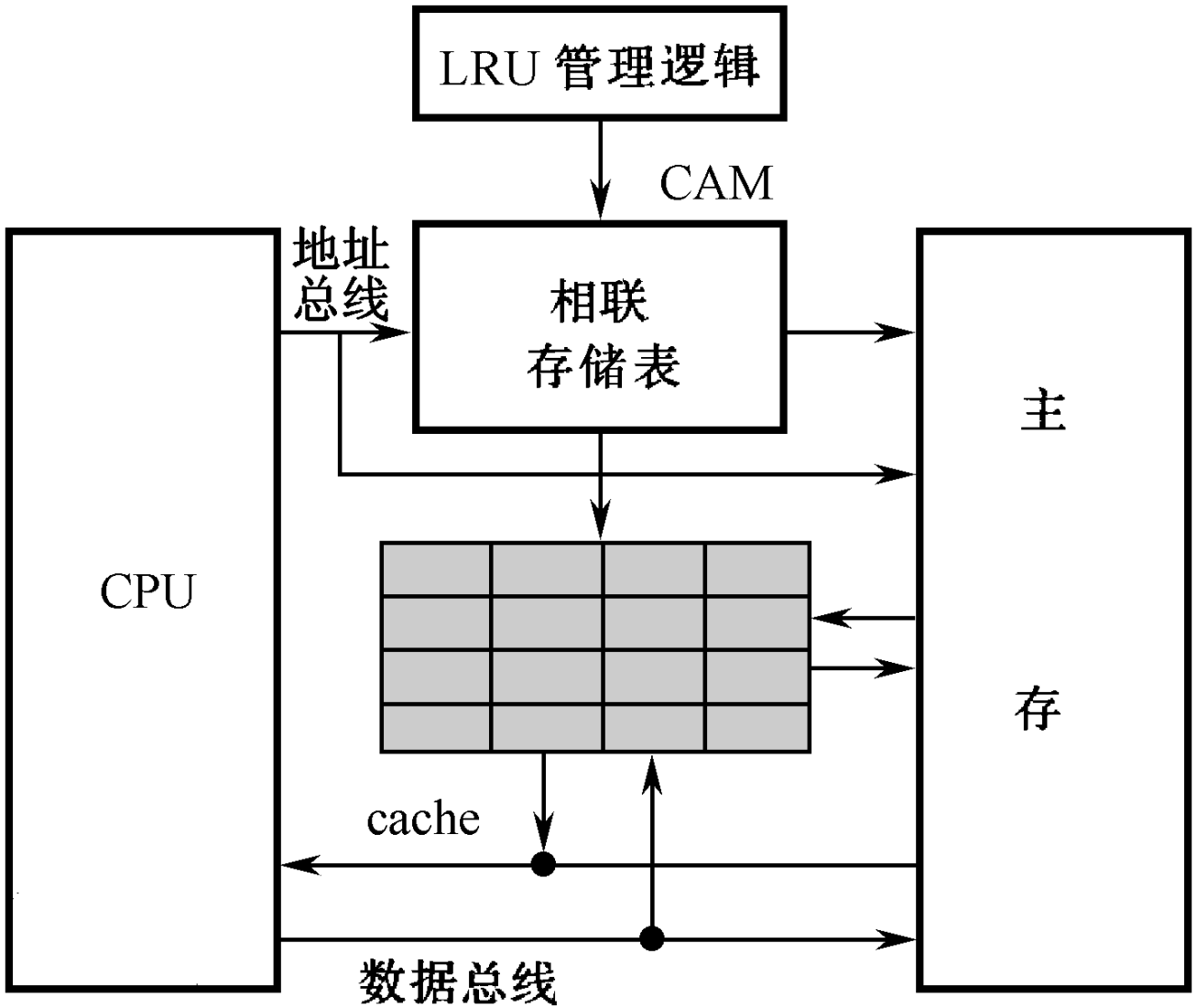
- CPU与Cache的数据传送以字为单位
- 主存与Cache的数据传送以块为单位
- 一个块由若干字组成，是定长的

- CPU读主存的一个字时，把此字的内存地址同时送给Cache和主存，Cache控制逻辑依据地址判断是否在Cache中：
  - 若在：此字立即传送给CPU
  - 否不在

  （1） 用主存读周期把此字从主存读出送到CPU

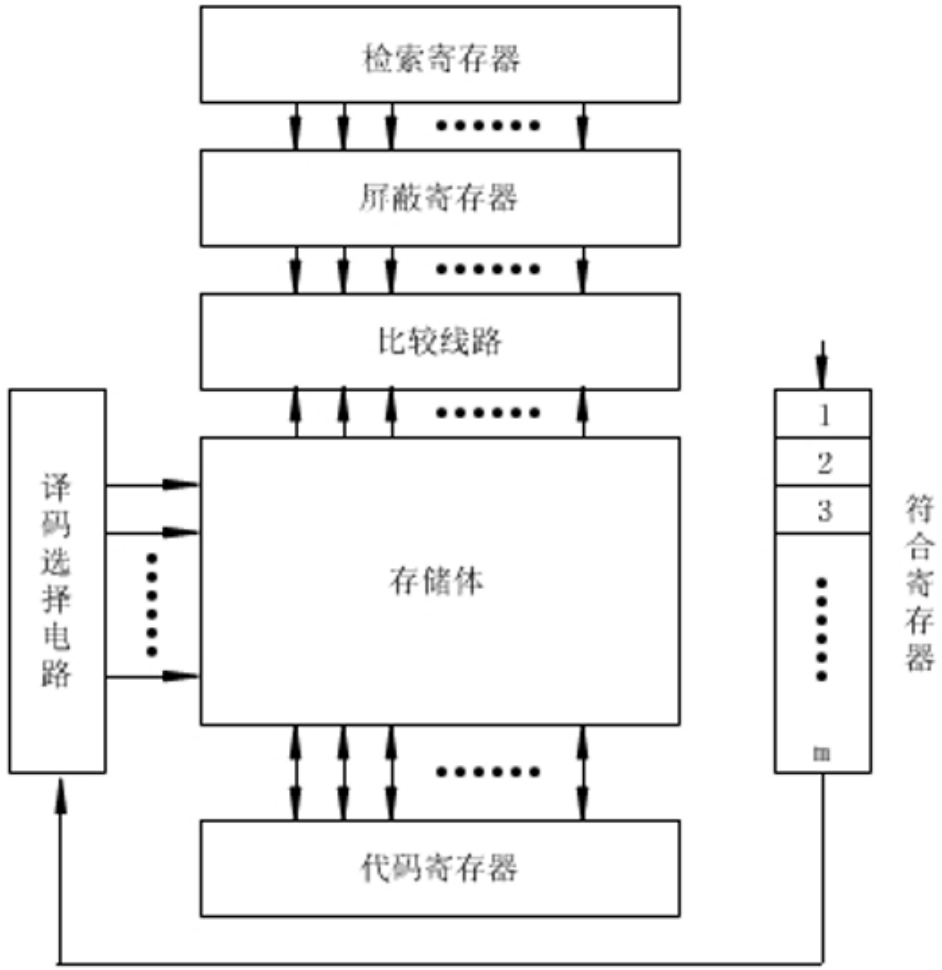
  （2） 把含有这个字的整个数据块从主存读出送到cache中

Cache基本原理（2/3）



Cache基本原理（3/3）

- **相联存储器/CAM**
  - **原理：**按内容存取的存储器，可以选择关键字的一个字段作为地址
  - **写入时：**按顺序写入，不需要地址
  - **读出时**：CPU给出一个相联关键字，用它和存储器中所有单元中的一部分信息进行比较，若它们相等，则将此单元中余下的信息读出。
  - **用途**
    - 在虚拟存储器中存放段表、页表和快表
    - Cache的行地址



Cache的命中率（1/4）

- **增加Cache的目标**
  - 主存的**平均读出时间**尽可能接近**Cache的读出时间**
- **如何达到这个目标？**
  - 在所有的存储器访问中，由cache满足CPU需要的部分应占很高的比例

    → 即cache的命中率应接近于1
- **实现这个目标可行么？**
  - 程序访问的局部性

Cache的命中率（2/4）

- **Cache命中（hit）**
  - CPU欲访问的数据已经在缓存中，即可直接访问Cache
- **Cache不命中（miss）**
  - CPU欲访问的数据不在Cache内，此时需要将该数据所在的主存整个子块一次调入Cache中
- **命中率（hit rate）**
  - CPU要访问的信息已在Cache内的比率
  - 在一个程序执行期间，设N_c表示cache完成存取的总次数，N_m表示主存完成存取的总次数，h定义为命中率，则有

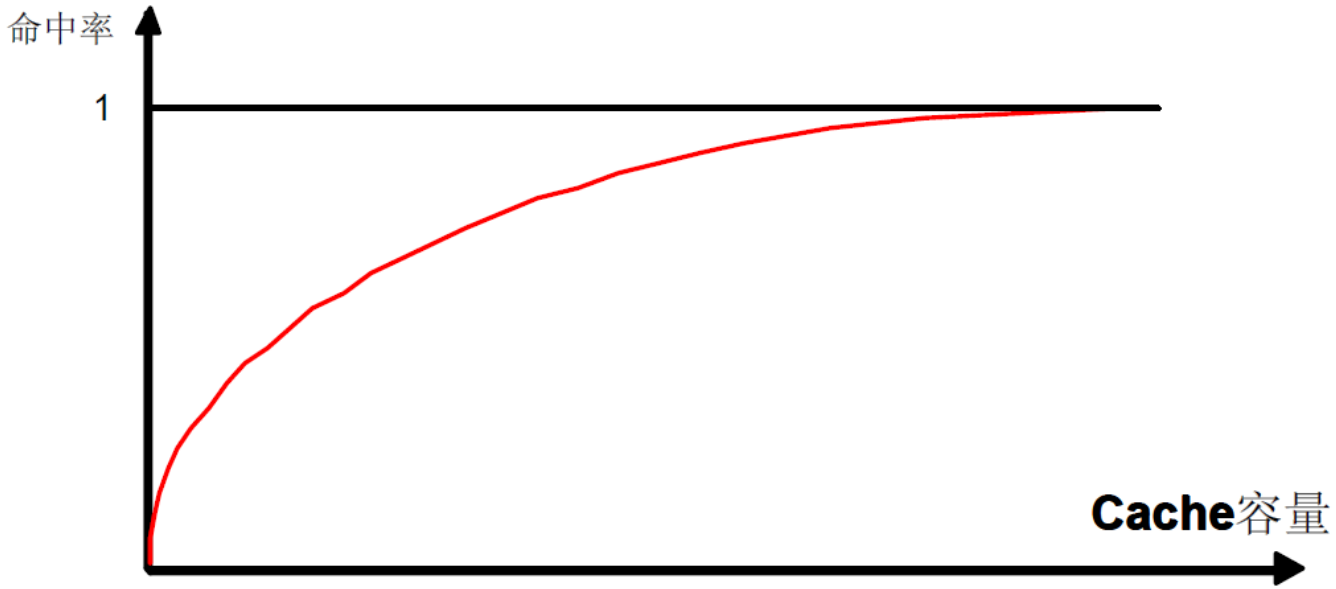
$$h = N_c /(N_c + N_m)$$

- **平均访问时间**
  - 若t_c表示命中时的Cache访问时间，t_m表示未命中时的主存访问时间，1-h表示未命中率，则cache/主存系统的平均访问时间t_a为：

$$t_a = h * t_c + (1-h) * t_m$$

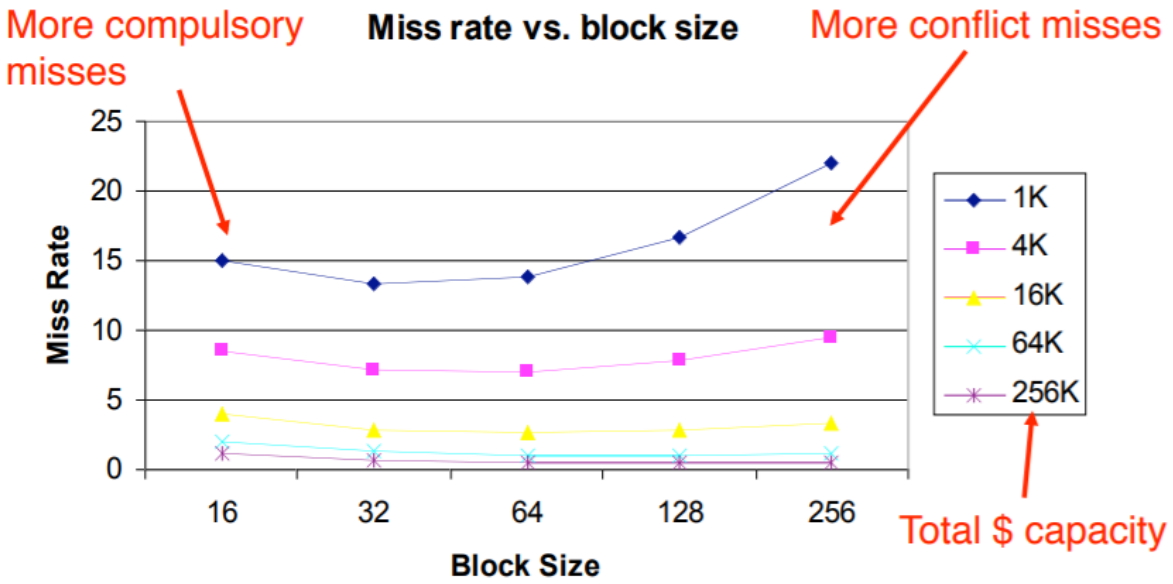
Cache的命中率（3/4）

- **Cache的命中率与什么因素相关?**
  - 程序访问模式、Cache容量、块的大小均相关
- **Cache容量越大，命中率越高**
  - 当Cache容量达到一定值时，命中率不会因为容量的增大而明显提高



（补充内容）思考题

- **在Cache容量一定的情况下，Cache块的大小（块长），与命中率的关系? 是否越大越好?**
- 补充内容
  - Cache的缺失损失(miss penalty), Cache的缺失类型



https://www3.nd.edu/~mniemier/teaching/2010_B_Fall/lectures/lec_20_slides.pdf

主存与Cache的地址映射

- 回顾CPU访问数据的过程
  - CPU要取一个字，首先要给出这个字**所在主存的地址**，**然后将该地址发往Cache**，判断所要取的字是否在Cache中，在就取走，不在就去内存找
- 问题引入
  - 如何用主存地址去访问Cache存储器？

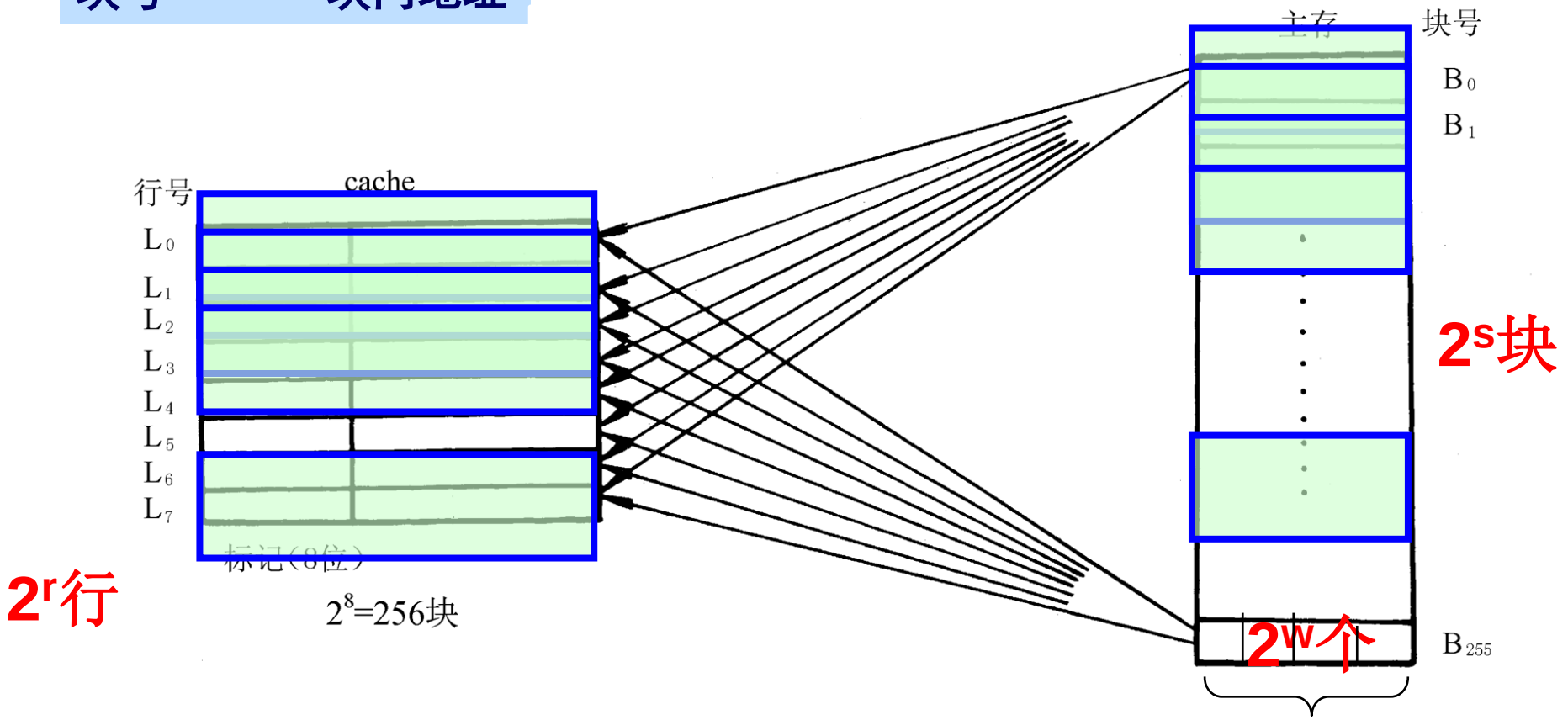
→主存与Cache的地址映射！

全相联的映射方式

内存地址：

XXXX XXXX	XX
块号	块内地址

- **映射方法（多对多）**
  - 主存内容可以拷贝到Cache任意行
- **地址变换**
  - 标记实际上构成了一个目录表



全相联的映射方式

- 转换公式

  块大小＝行大小＝2^w个字

  主存的块数＝2^s

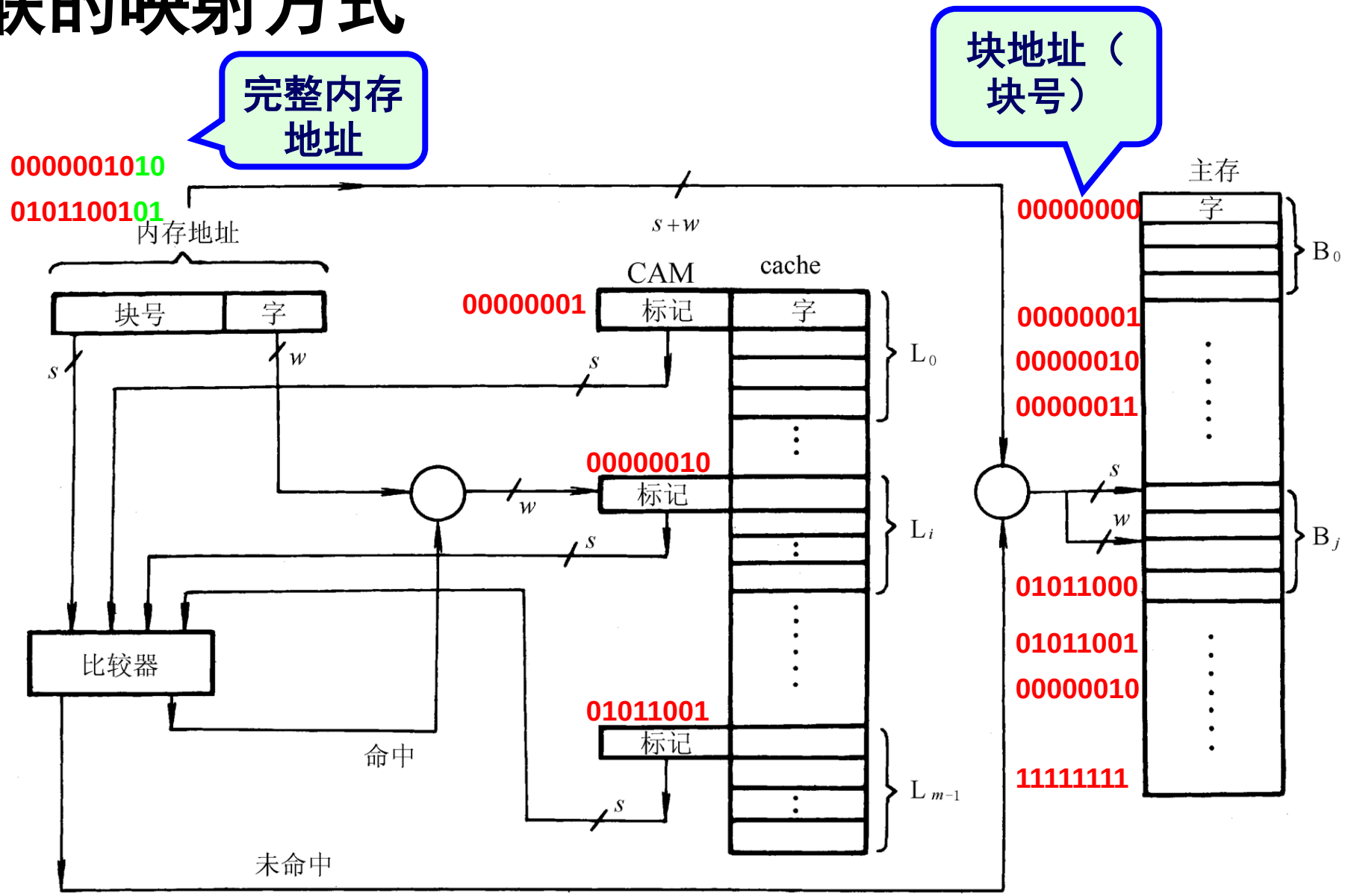
  主存地址长度＝(s+w)位

➔

标记大小＝块号的位数=s位

cache的行数：不由地址格式确定

全相联的映射方式



（b） 全相联cache的检索过程

全相联的映射方式

特点：

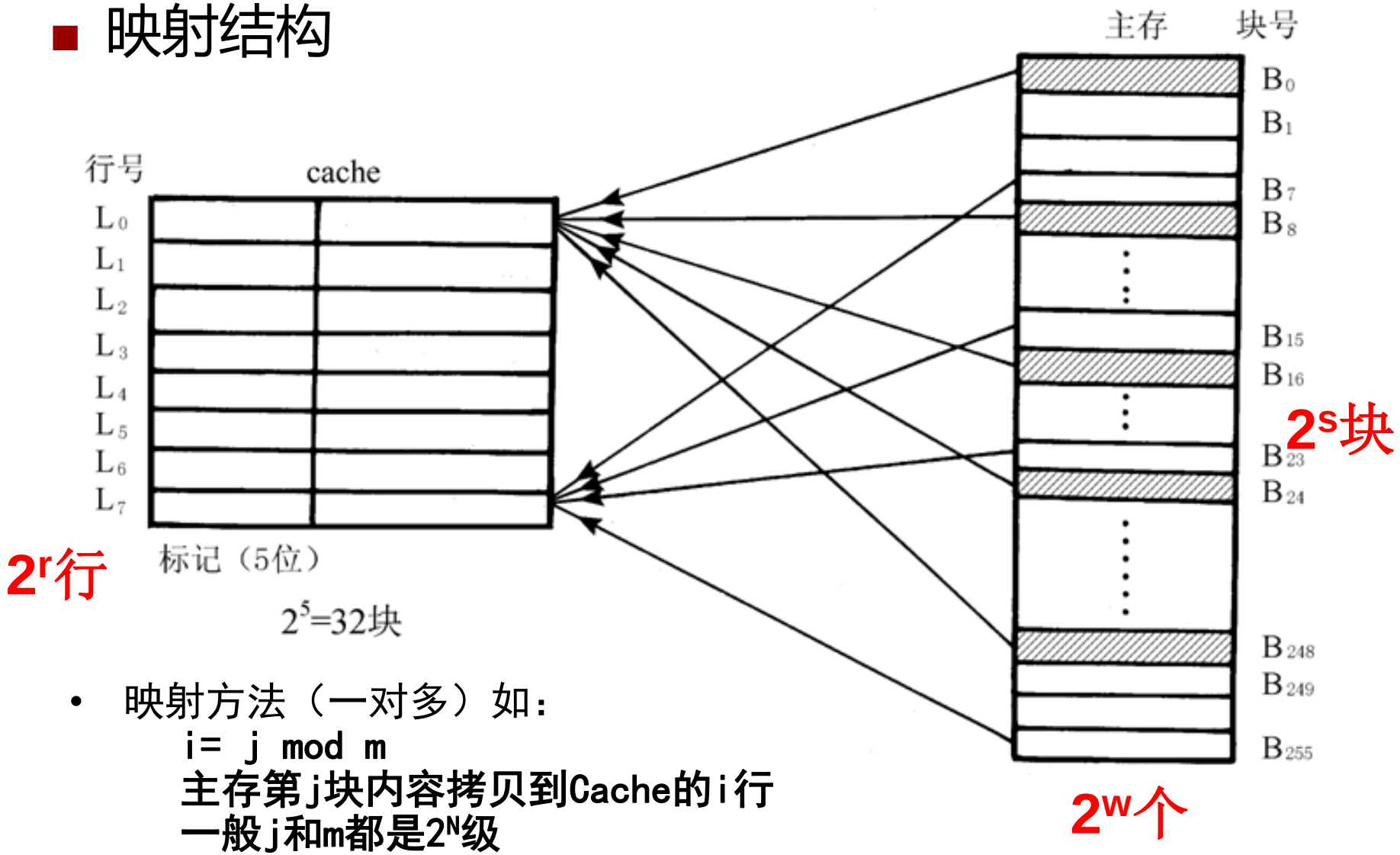
- 优点：冲突概率小，Cache的利用高。
- 缺点：比较电路实现成本高，需要一个访问速度很快代价高的相联存储器

应用场合：

- 适用于小容量的Cache

直接映射方式

- 映射结构



- 映射方法（一对多）如：
  i= j mod m
  主存第j块内容拷贝到Cache的i行
  一般j和m都是2N级

直接映射方式

- 转换公式

  块大小＝行大小＝2^w个字

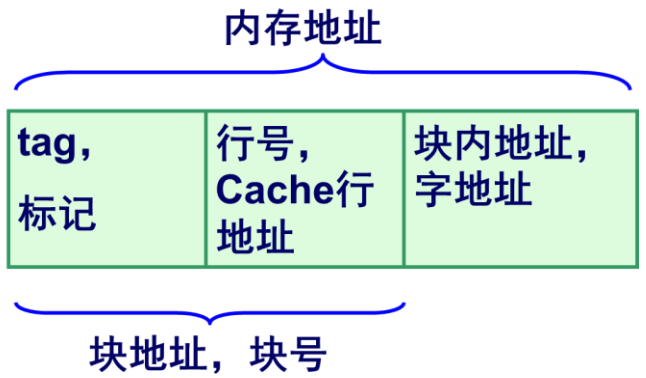
  主存的块数＝2^s

  主存地址长度＝(s+w)位

  Cache的行数＝m＝2^r

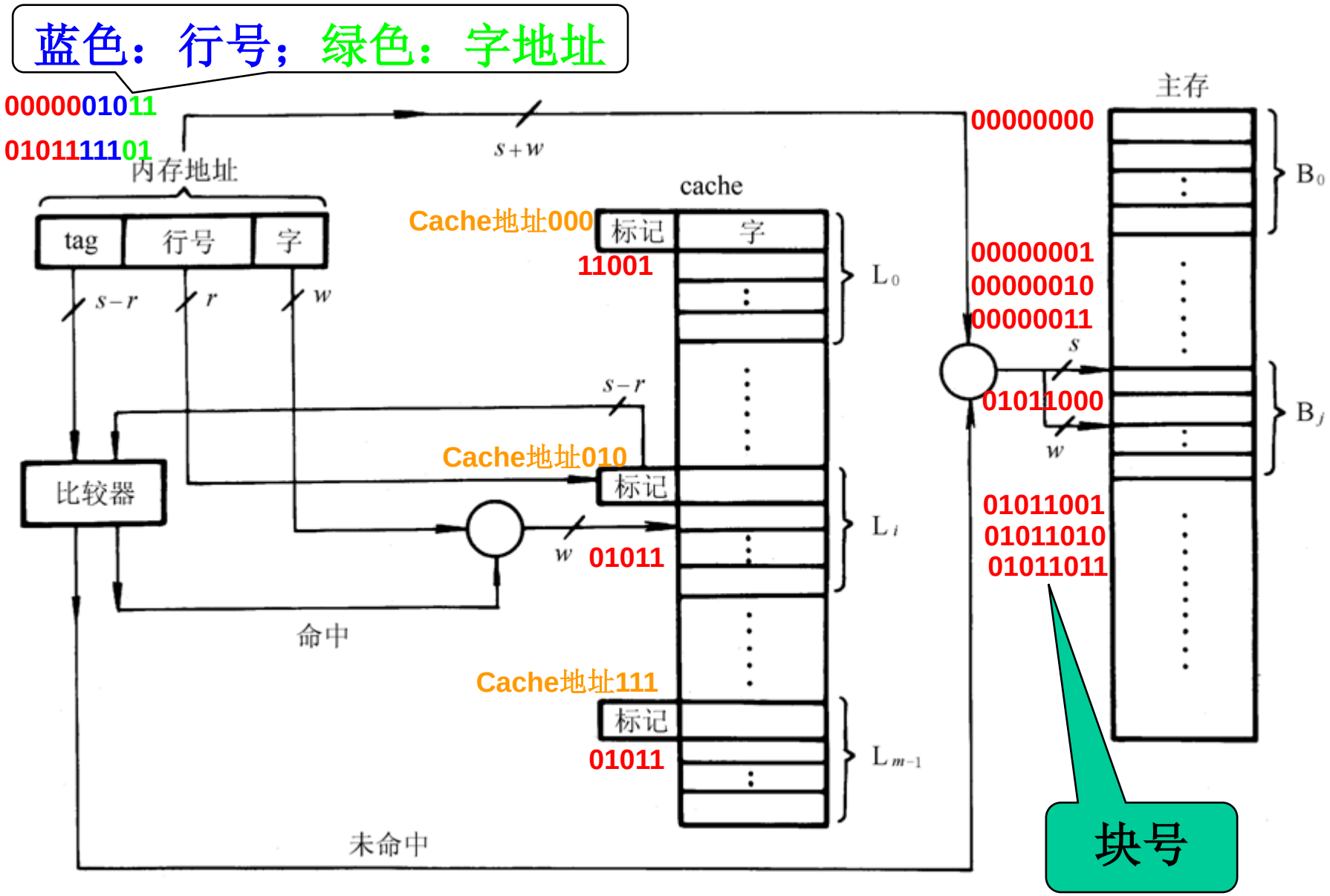
➔

  标记大小＝(s-r)位



直接映射方式

蓝色：行号；绿色：字地址



直接映射方式

特点

- 优点：比较电路少，所以硬件实现简单
- 缺点：冲突概率高（易抖动，频繁交换）

应用场合

- 适合大容量Cache

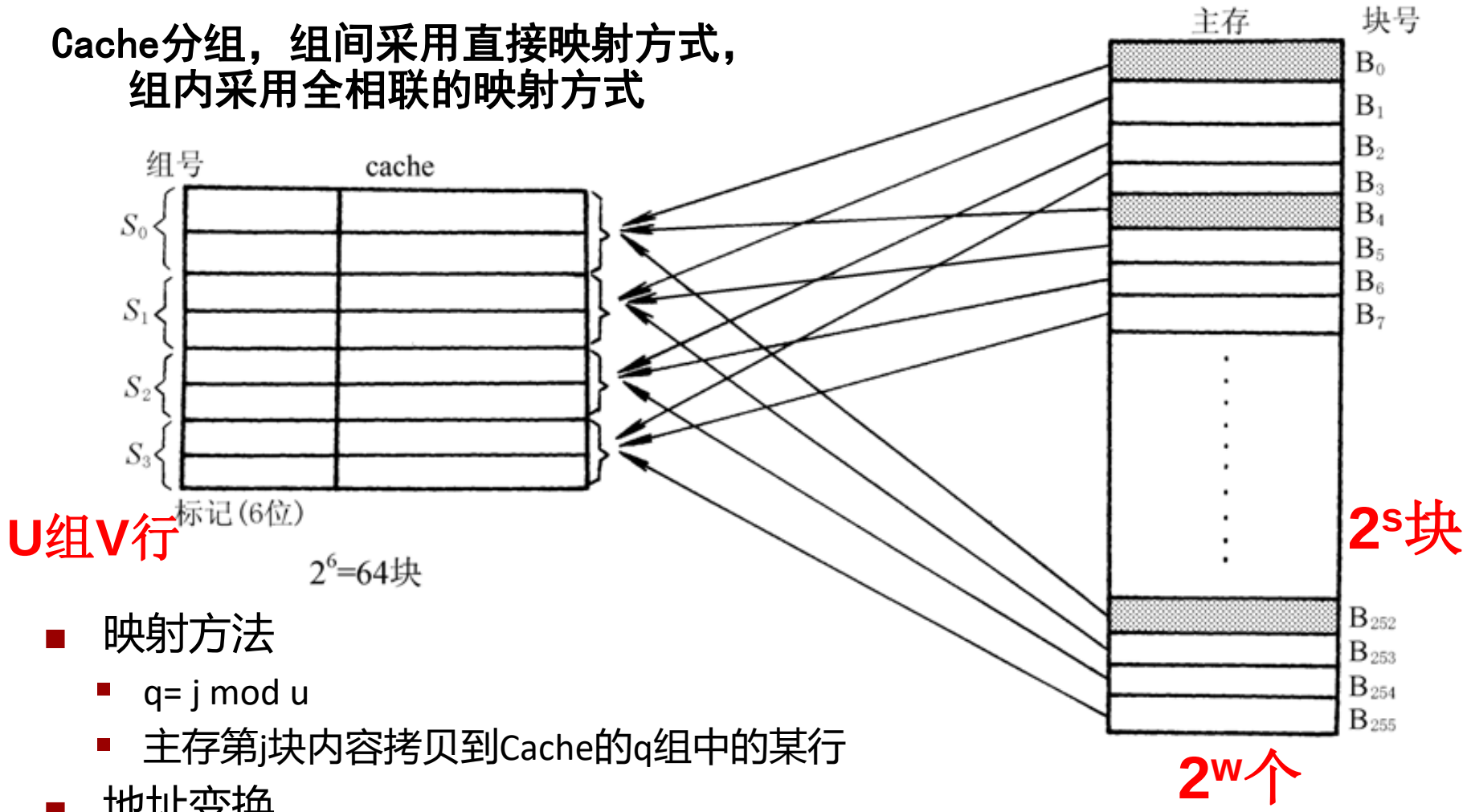
思考两种映射方式的优缺点

- 全相联检索成本高
- 直接映射换出换入机制有缺陷

- **能否折中？**

组相联映射方式

Cache分组，组间采用直接映射方式，
组内采用全相联的映射方式



- 映射方法
  - q= j mod u
  - 主存第j块内容拷贝到Cache的q组中的某行
- 地址变换
  - 设主存内块地址x，看是不是在cache中，
    先计算组号y= x mod u，则在y组中查找

组相联映射方式

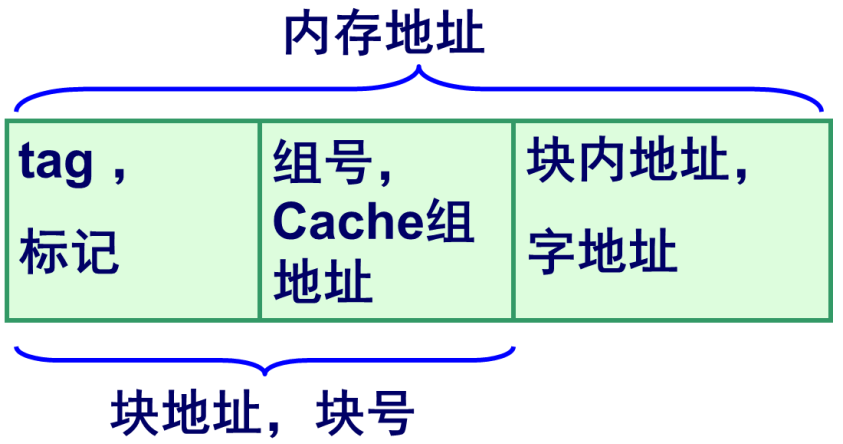
- 分析：比全相联容易实现，冲突低
- v=1，则为直接相联映射方式
- u=1，则为全相联映射方式
- v的取值一般比较小， 一般是2的幂，称之为v路组相联cache

组相联映射方式

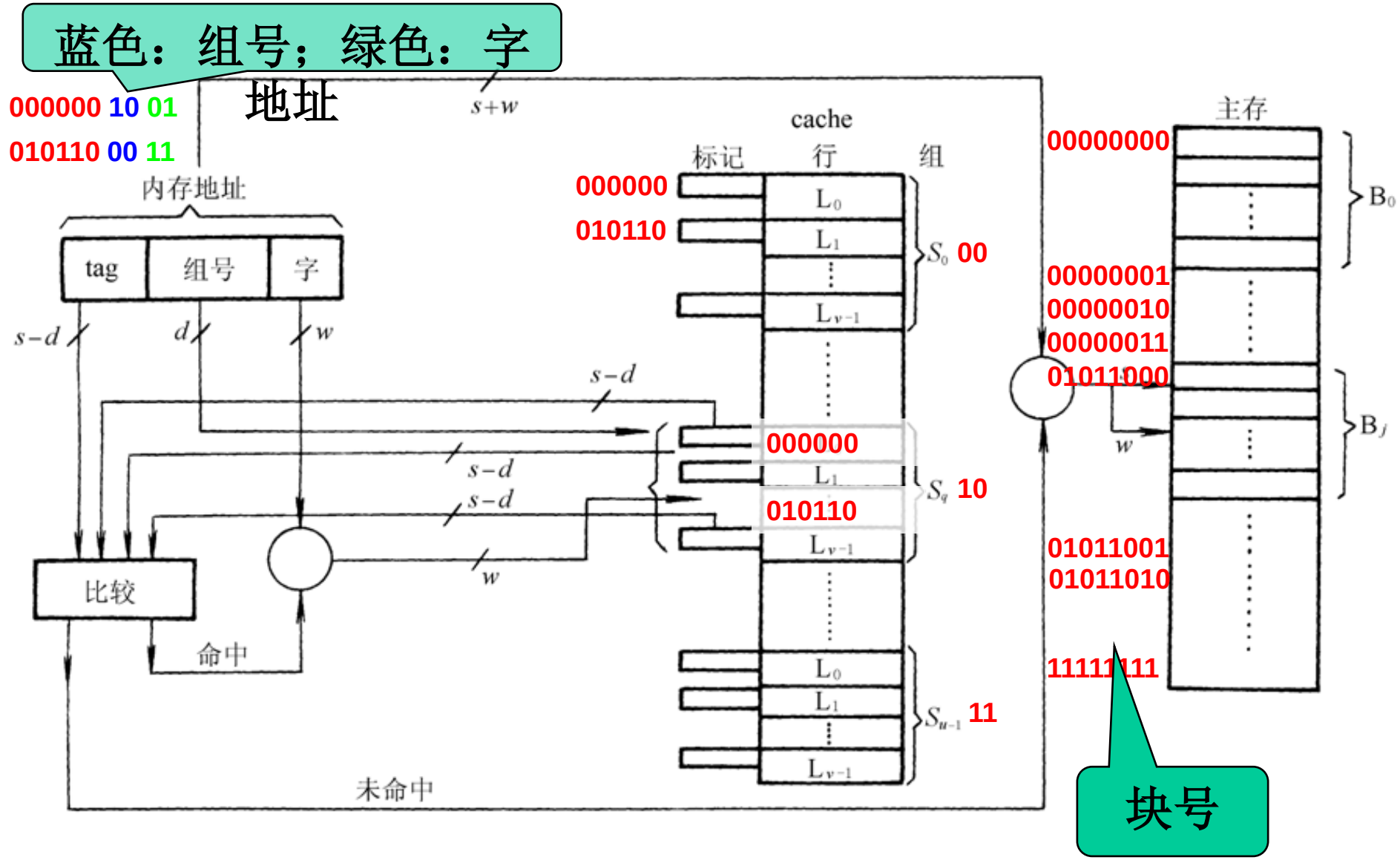
- 转换公式
  - 主存地址长度＝(s+w)位
    块大小＝行大小＝2^w个字
    主存的块数＝2^s

组数 ＝ u
每组的行数＝v
cache的行数＝uv

组号的位数为d　　2^d＝u
标记大小＝(s−d)位



组相联映射方式



(b) 组相联cache的检索过程

组相联映射方式

- 特点
  - V路比较器较易实现
  - 块在组中存放有一定灵活性，冲突较少
  - 是全相联映射和直接映射方法的折衷
- 应用场合
  - 因兼顾了二者优点又尽量避免了缺点，被普遍使用

练习题

- 有一个存储体系，主存容量1MB，字长1B，块大小16B，Cache容量64KB。若Cache采用全相联映射，对内存地址（F0010）H给出相应的标记和字号。

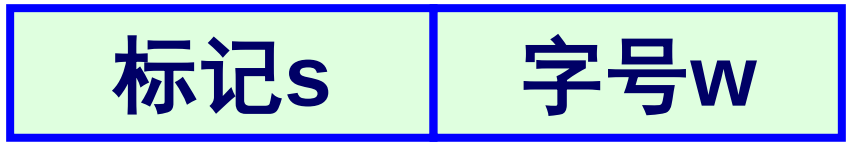
块大小=行大小=2^4字节　w=4

主存寻址单元数1M=2^{20}

S+w=20位，s=16位

主存的块数2^{16}，标记大小s=16位

主存格式



主存地址（F0010）$_{16}$=（1111 0000 0000 0001 0000）$_2$

对应的标记=1111 0000 0000 0001　字号=0000

Cache

- **仍然未被解决的问题：**
  - CPU访问Cache未命中发生后，要把内存中的块放入Cache，多次放入导致cache满了怎么办？
- **需要新的块覆盖旧的块，即替换**
  - 覆盖掉哪个块？依照什么策略替换？

Cache 替换策略

- LFU（最不经常使用，Least Frequently Used）：
  - 被访问的行计数器增加1，换值小的行
  - 不能反映近期cache的访问情况
- LRU（近期最少使用，Least Recently Used）
  - 被访问的行计数器置0，其他的计数器增加1
  - 换值大的行，符合cache的工作原理
- 随机替换：
  - 从特定的行位置中随机地选取一行换出
  - 在硬件上容易实现，且速度也比前两种策略快
  - 缺点是随意换出的数据很可能马上又要使用，从而降低命中率和cache工作效率
  - 但这个不足随着cache容量增大而减小
  - 随机替换策略的功效只是稍逊于前两种策略

Cache替换策略

- 例：设cache有1、2、3、4共4个行，a、b、c、d等为主存中的块，访问顺序一次如下：a、b、c、d、b、b、c、c、d、d、a ，下次若要再访问e块。
  问，采用LFU和LRU算法替换结果是不是相同？

	LFU（最不经常使用）					LRU（近期最少使用）				
	说明	1行	2行	3行	4行	说明	1行	2行	3行	4行
a	a进入	1	0	0	0	a进入	0	1	1	1
b	b进入	1	1	0	0	b进入	1	0	2	2
c	c进入	1	1	1	0	c进入	2	1	0	3
d	d进入	1	1	1	1	d进入	3	2	1	0
b	命中	1	2	1	1	命中	4	0	2	1
b	命中	1	3	1	1	命中	5	0	3	2
c	命中	1	3	2	1	命中	6	1	0	3
c	命中	1	3	3	1	命中	7	2	0	4
d	命中	1	3	3	2	命中	8	3	1	0
d	命中	1	3	3	3	命中	9	4	2	0
a	命中	2	3	3	3	命中	0	5	3	1
e	替换a	1	0	0	0	替换b	1	0	4	2

Cache写操作策略

- 前面讲cache的作用时多用的例子是CPU从内存读数据的情形
- 试想，如果是CPU要给内存写数据是什么情况？

Cache写操作策略

- 思考CPU写入内存时，cache命中的情况？
- 由于cache的内容只是主存部分内容的拷贝，它应当与主存内容保持一致
- 而CPU对cache的写入更改了cache的内容。如何与主存内容保持一致？
- 可选用三种写操作策略

Cache写操作策略

- 写回法
  - 当CPU写cache命中时，只修改cache的内容，而不立即写入主存
  - 只有此行被换出时才写回主存
  - 这种方法使cache真正在CPU-主存之间读写两方面都起到了高速缓存作用
  - 对一个cache行的多次写命中都能够在cache中快速完成，只是需要替换时才写回速度较慢的主存，减少访问主存的次数
  - 每个cache行配一个修改位，换出时，对行的修改位进行判断，决定是写回还是舍掉
  - 显著减少写主存次数，但存在不一致性隐患

Cache写操作策略

- 全写法
  - 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与主存内容的一致性
  - 当写cache未命中时，只能直接向主存进行写入
    - 是否将修改后的主存块取到cache，有两种选择方法：
      - WTWA：取主存块到cache并为它分配一个行位置
      - WTNWA：不取主存块到cache

Cache写操作策略

- 写一次法
  - 基于写回法并结合全写法
  - 对称多处理器系统中（每个处理器地位相同，且都有内置的Cache）/多级Cache
  - 写命中与写未命中的处理方法与写回法基本相同
  - **第一次写命中时要同时写入主存**
    - 因为第一次写cache命中时，CPU要在总线上启动一个存储写周期， 其他cache监听到此主存块地址及写信号后，即可拷贝该块或及时作废，以便维护系统全部cache的一致性