存储器层次结构（1）

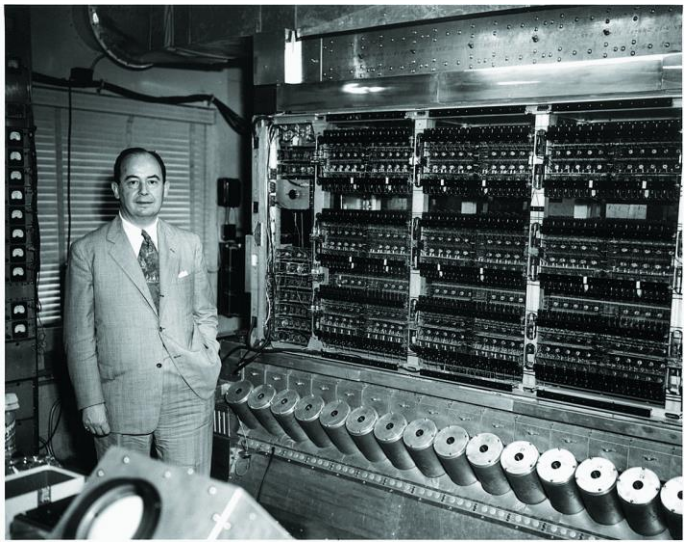
王晶

jwang@ruc.edu.cn，信息楼124

2024年11月

计算机系统

- 三个关键部件
- 计算
- 存储
- 互联



Burks, Goldstein, von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," 1946.

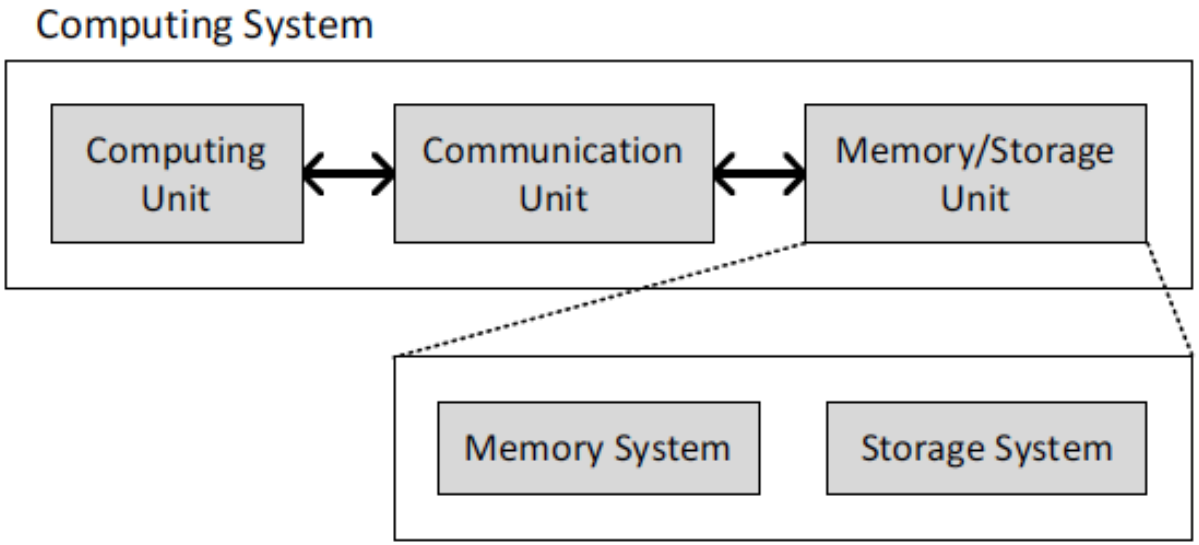
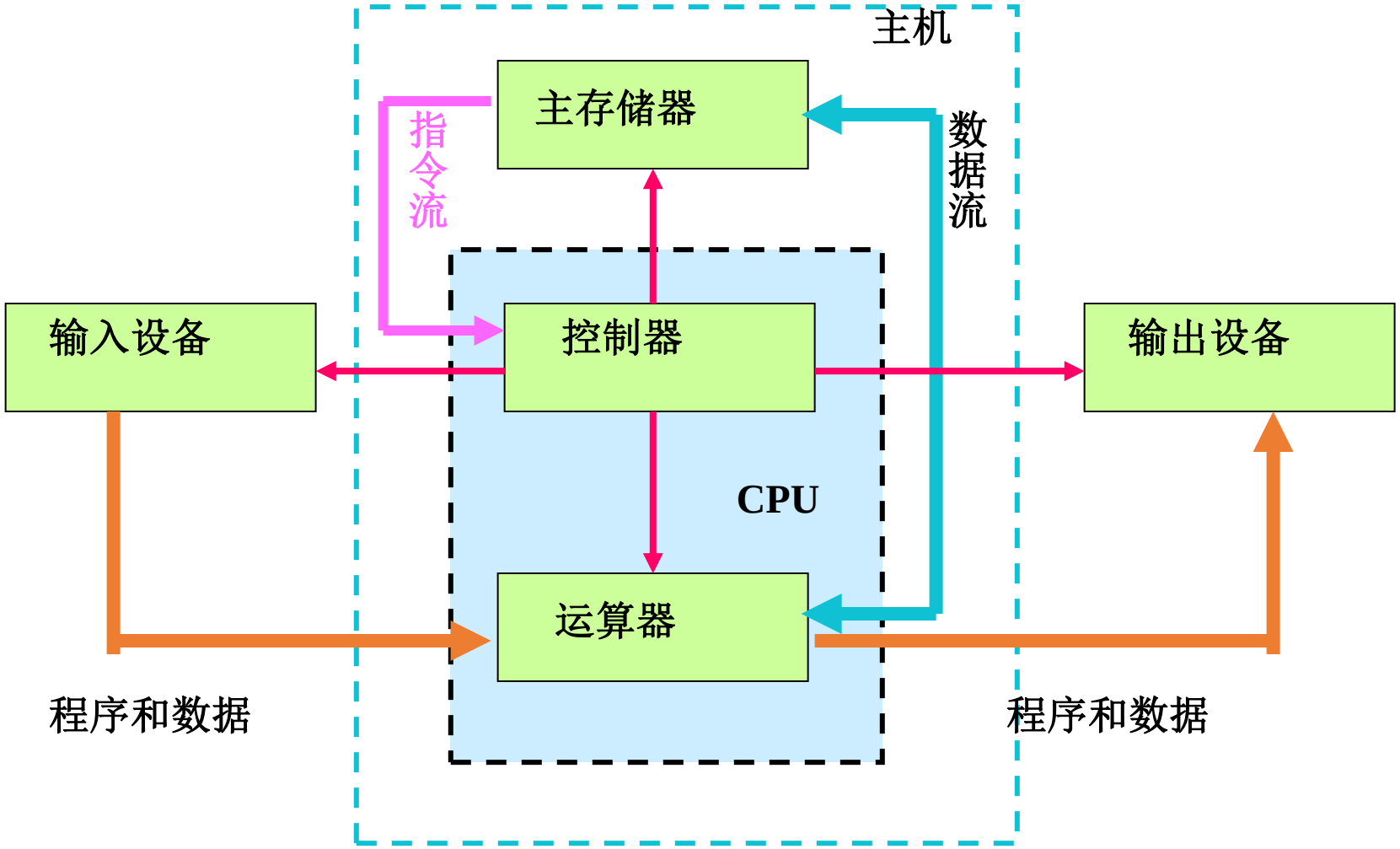
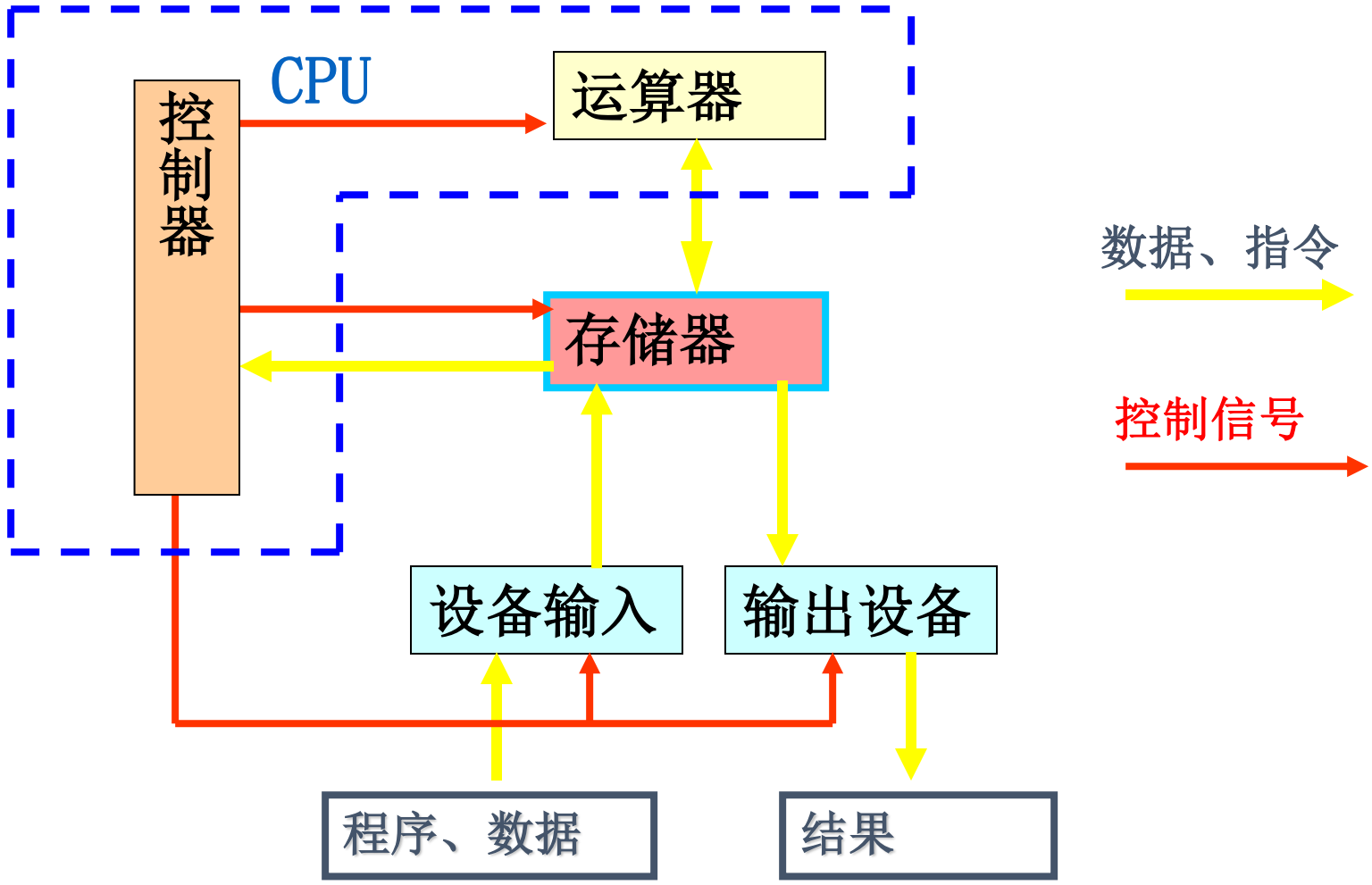


Image source: https://lbsitbytes2010.wordpress.com/2013/03/29/john-von-neumann-roll-no-15/

以运算器为中心的硬件结构



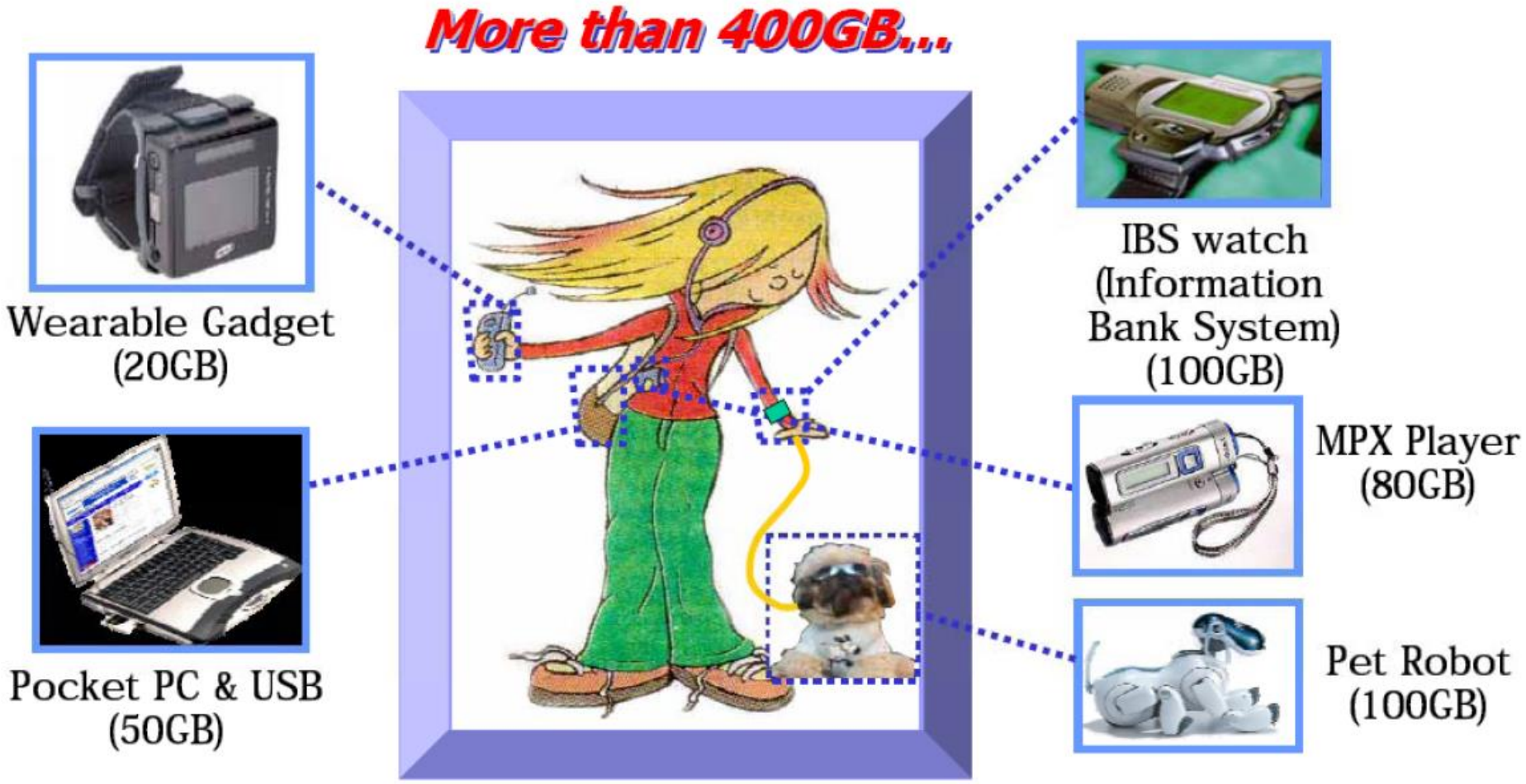
以存储器为中心的结构



存储器

- 存储器在计算机中存放程序和数据。在现代计算机中处于全机的中心地位。
  - ✓ 存放当前正在运行的程序和数据。
  - ✓ 实现主存与I/O设备之间的信息传送。
  - ✓ 在多处理机系统中，存放共享数据。
- 设计一个容量大、速度快、成本低的存储系统是计算机发展的一个重要课题。

Memory in Your Hands



Phone, Data, GPS, Game, Entertainment, …

C. Kim, "Future Memory Technology: Trends and Challenges," Plenary paper, *ISQED* (2006).

你认识它们吗？

（1）

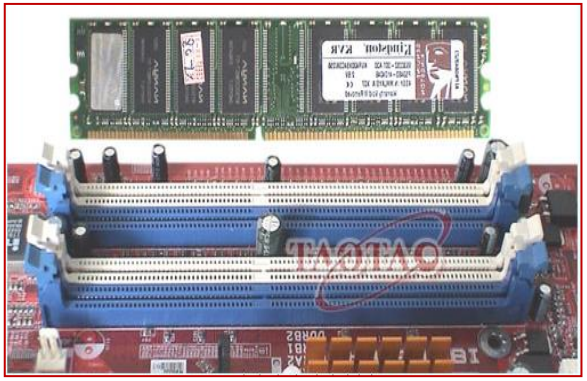
（2）

（3）

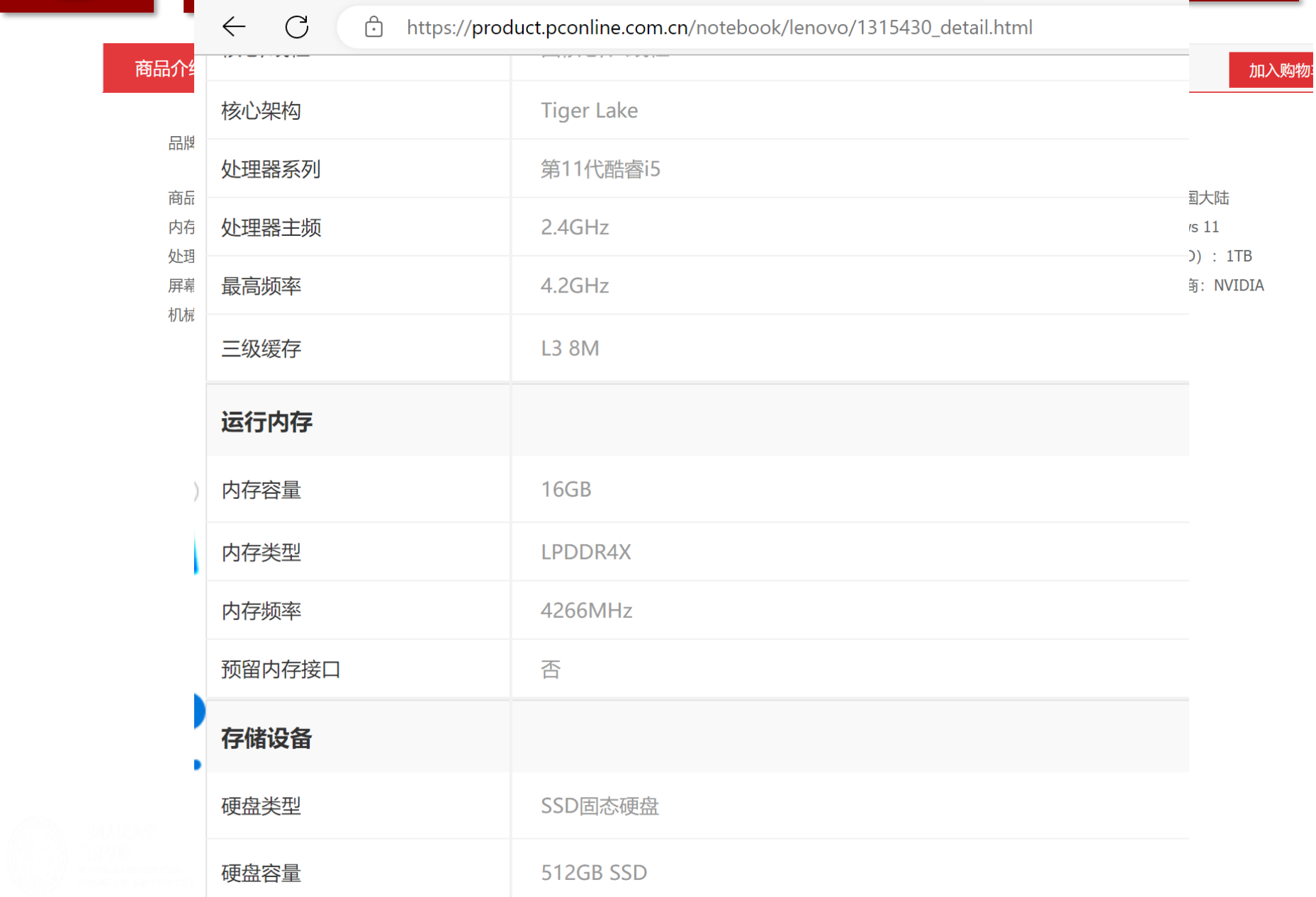
（4）

（5）

（6）



你关注过这些参数吗？


https://product.pconline.com.cn/notebook/lenovo/1315430_detail.html
商品介绍
加入购物
核心架构 Tiger Lake
处理器系列 第11代酷睿i5
处理器主频 2.4GHz
最高频率 4.2GHz
三级缓存 L3 8M
运行内存
内存容量 16GB
内存类型 LPDDR4X
内存频率 4266MHz
预留内存接口 否
存储设备
硬盘类型 SSD固态硬盘
硬盘容量 512GB SSD

Storage Trends

SRAM

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/MB	2,900	320	256	100	75	60	*25*	*116*
access (ns)	150	35	15	3	2	1.5	*1.3*	*115*

DRAM

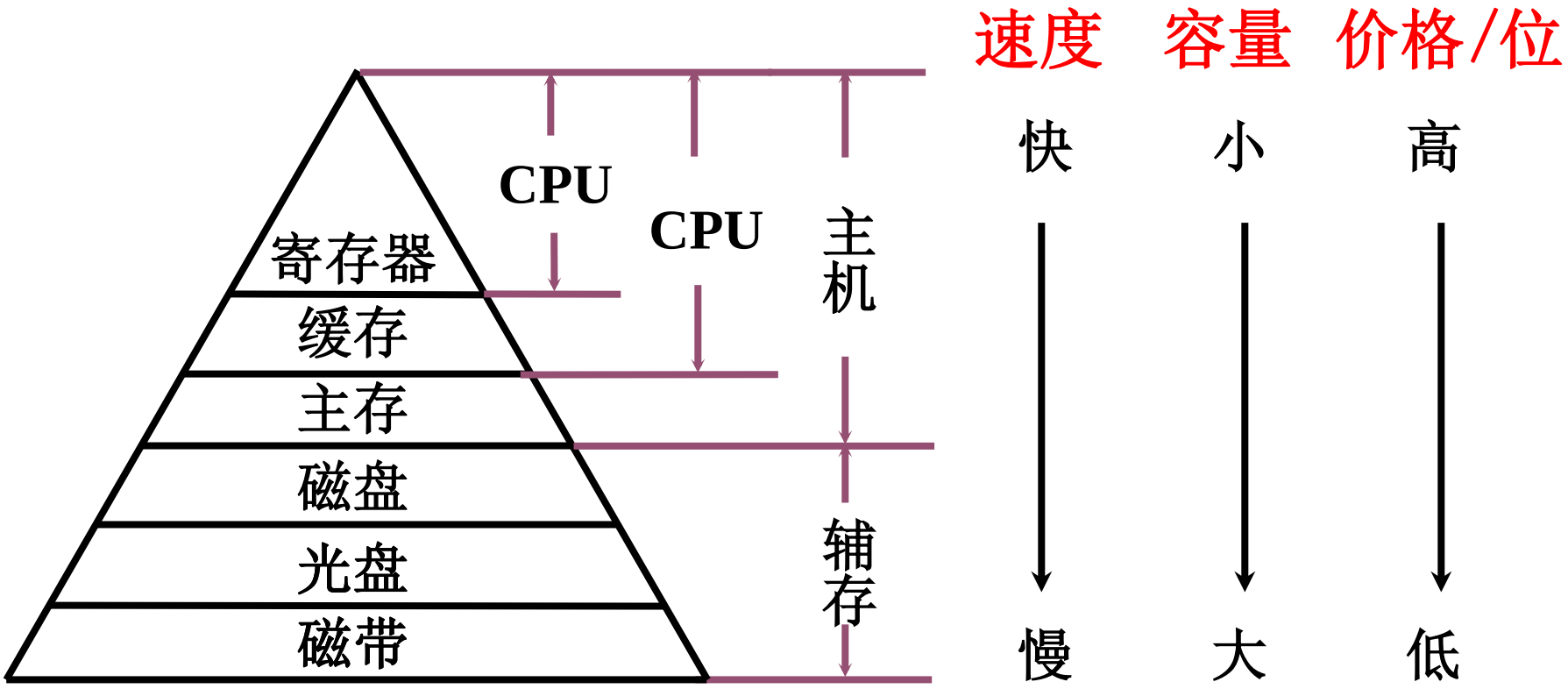
Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/MB	880	100	30	1	0.1	0.06	0.02	*44,000*
access (ns)	200	100	70	60	50	40	20	*10*
typical size (MB)	0.256	4	16	64	2,000	8,000	16.000	*62,500*

Disk

Metric	1985	1990	1995	2000	2005	2010	2015	*2015:1985*
$/GB	100,000	8,000	300	10	5	0.3	0.03	*3,333,333*
access (ms)	75	28	10	8	*5*	*3*	*3*	*25*
typical size (GB)	0.01	0.16	1	20	160	1,500	3,000	*300,000*

存储器的层次结构

存储器三个主要特性的关系



对其要求是：尽可能快的读写速度，尽可能大的存储容量，尽可能低的成本费用。

缓存--主存层次和主存 --辅存层次

结论：采用单一的存储模式很难满足速度快、容量大、价格低的要求！

根据以上存储器的分层情况，如何将它们组织成一个存储系统呢？

为解决主存速度与CPU不匹配的问题，借助于硬件在二者之间增加了—CACHE。

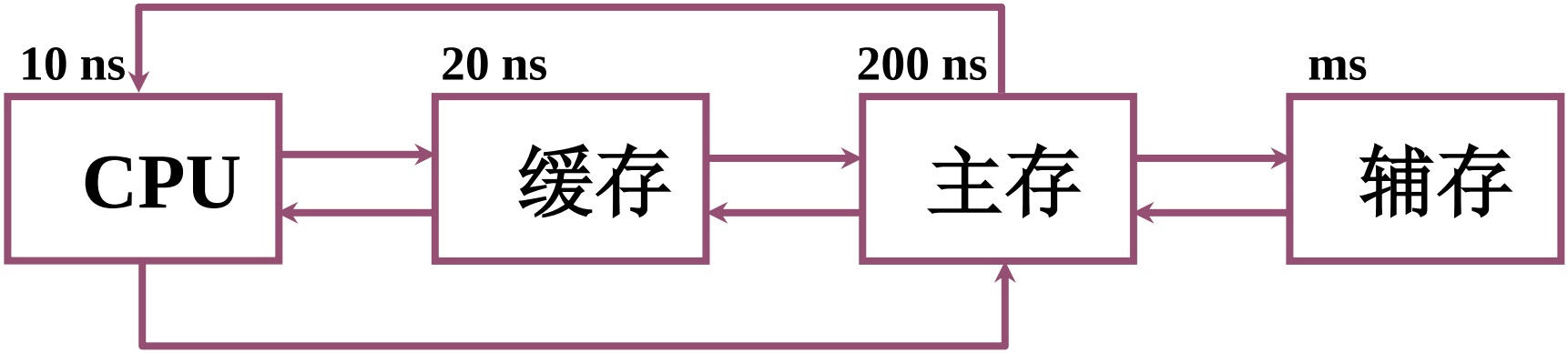
为解决主存容量不足的问题，借助于软硬件在高速硬盘上开辟了—虚拟存储空间。

构成了一个统一管理、统一调度，并且对用户来说透明的一体化的存储器系统，即三级结构的存储器系统！

缓存--主存层次和主存 --辅存层次

存储层次 / 比较项目	“Cache - 主存”层次	“主存 - 辅存”层次
目　　的	为了弥补主存速度的不足	为了弥补主存容量的不足
存储管理实现	主要由专用硬件实现	由硬件和软件实现
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

采用三级存储的体系结构!



- 若能使 CPU大部分时间访问高速缓存CACHE速度最快；
- 仅在从缓存中读不到数据时才去读主存，速度略慢但容量较大；
- 当从主存中还读不到时才去成批量读虚存，速度很慢容量极大；
- 很好地同时解决了对速度、容量、成本三个方面的需求。

为什么这个方案是可行的？

访存的局部性原理

局部性原理表现在：时间上和空间上

- 时间局部性：在一小段时间内，最近被访问过的存储单元很可能再次被访问。
- 空间局部性：在空间上，被访问的存储单元往往集中在一小片连续存储区。

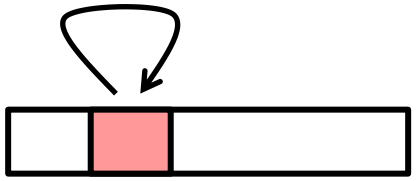
因此：可以把程序和数据合理地分配在不同存储介质中。

```
for （i=0； i<1000; i++)
    for （j=0； j<1000; j++)
        sum+= a[i][j];
```

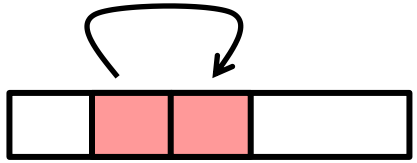
经典程序片段

局部性原理

- 局部性(locality)：倾向于引用邻近于其他最近引用过的数据项的数据项，或者最近引用过的数据项本身
- 局部性原理(principle of locality)：数据局部性的倾向性称为局部性原理
- 局部性的两种形式：
  - 时间局部性(temporal locality)：被引用过的内存位置很可能在不远的将来再被多次引用



  - 空间局部性(spatial locality)：一个内存位置被引用了一次，程序很可能在不远的将来引用附近的一个内存位置



局部性的例子

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- 数据访问
  - 连续引用数组元素 （stride-1 reference pattern）。 空间局部性
  - 引用变量 sum 每次迭代。 时间局部性
- 指令访问
  - 按顺序访问指令。 空间局部性
  - 重复执行循环指令。 时间局部性

Qualitative Estimates of Locality

- Claim: 能够查看代码并对其局部性有定性感觉是专业程序员的一项关键技能。

- Question: 这个函数相对于数组 a 有很好的局部性吗？

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

- Question:这个函数相对于数组 a 有很好的局部性吗？

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Answer: no, unless…

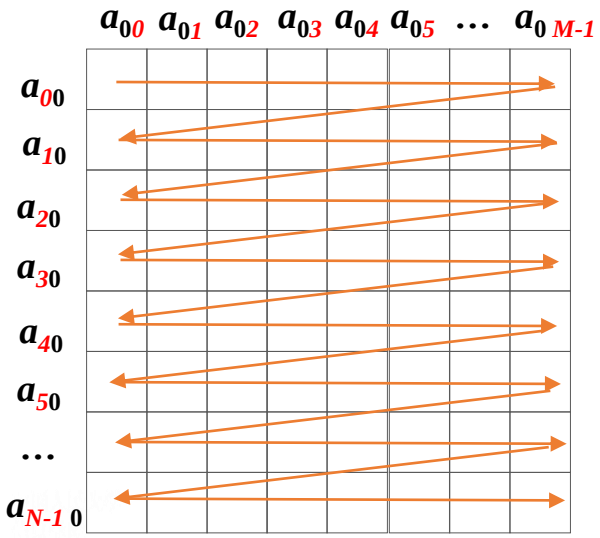
M is very small

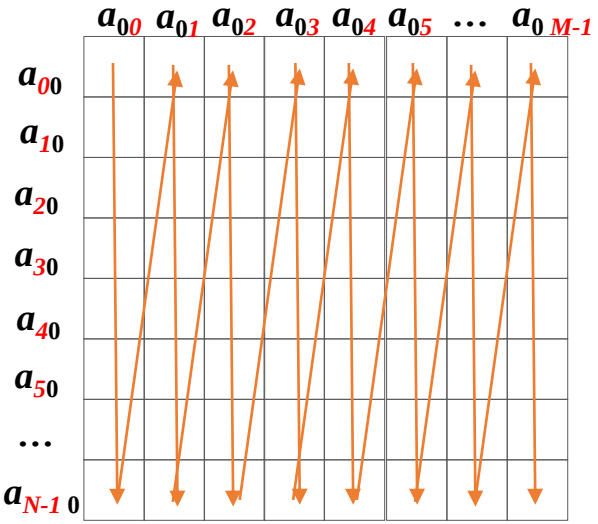
多维数组引用局部性

- 多维数组行访问顺序与列访问顺序具有不同的局部性

```
int sumarrayrows(int a[M][N]){
        int i,j,sum=0;
        for (i=0;i<M;i++)
                for (j=0;j<N;j++)
                        sum+=a[i][j];
        return sum;
}
```

```
int sumarraycols(int a[M][N]){
        int i,j,sum=0;
        for (j=0;j<N;j++)
                for (i=0;i<M;i++)
                        sum+=a[i][j];
        return sum;
}
```





局部性的实例

- 问题:您能否排列循环，以便该函数使用 stride-1 参考模式扫描 3-d 数组 a（从而具有良好的空间局部性）？

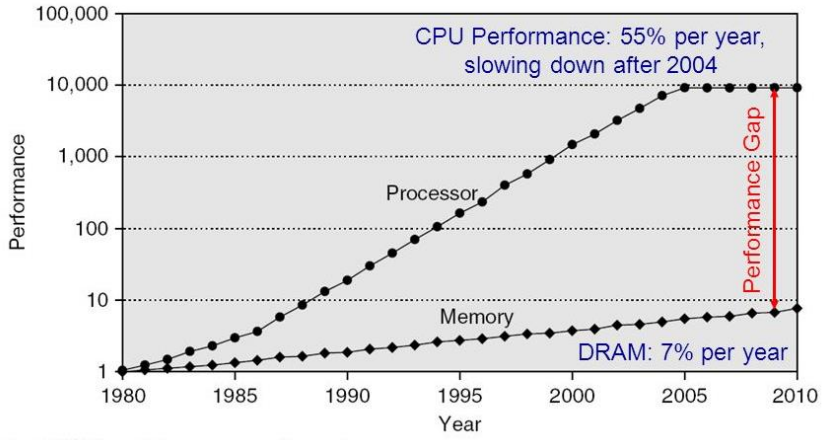
```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

Answer: make j the inner loop

存储层次结构

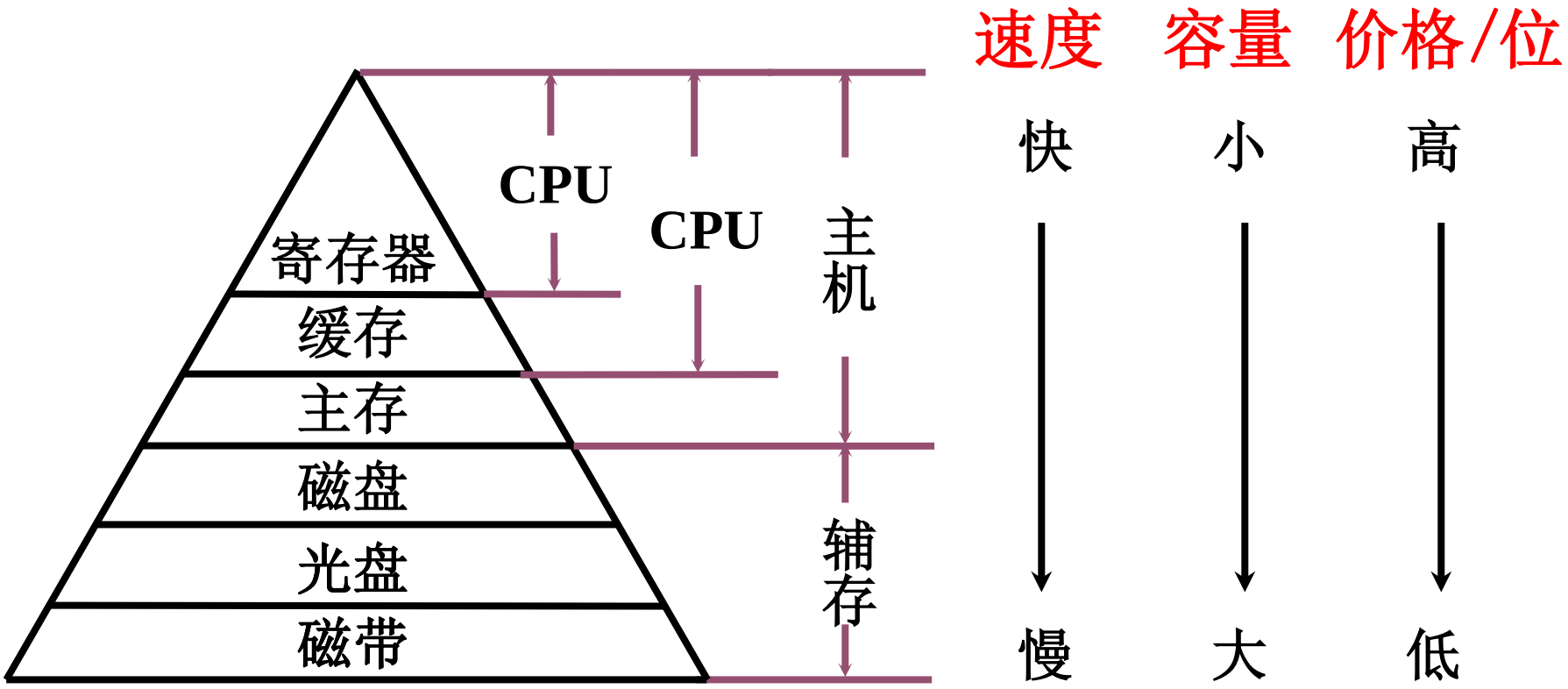
- 硬件和软件的一些基本特性：
  - 快速存储技术每字节成本更高，容量更小，并且需要更多的功率（热量！）
  - CPU 和主内存速度之间的差距正在扩大。
  - 编写良好的程序往往表现出良好的局部性。
- 这些基本特性完美地相辅相成。
- 提出了一种组织内存和存储系统的方法，称为内存层次结构。



- 1980 – No cache in microprocessor
- 1995 – Two-level cache on microprocessor

存储器的层次结构

存储器三个主要特性的关系



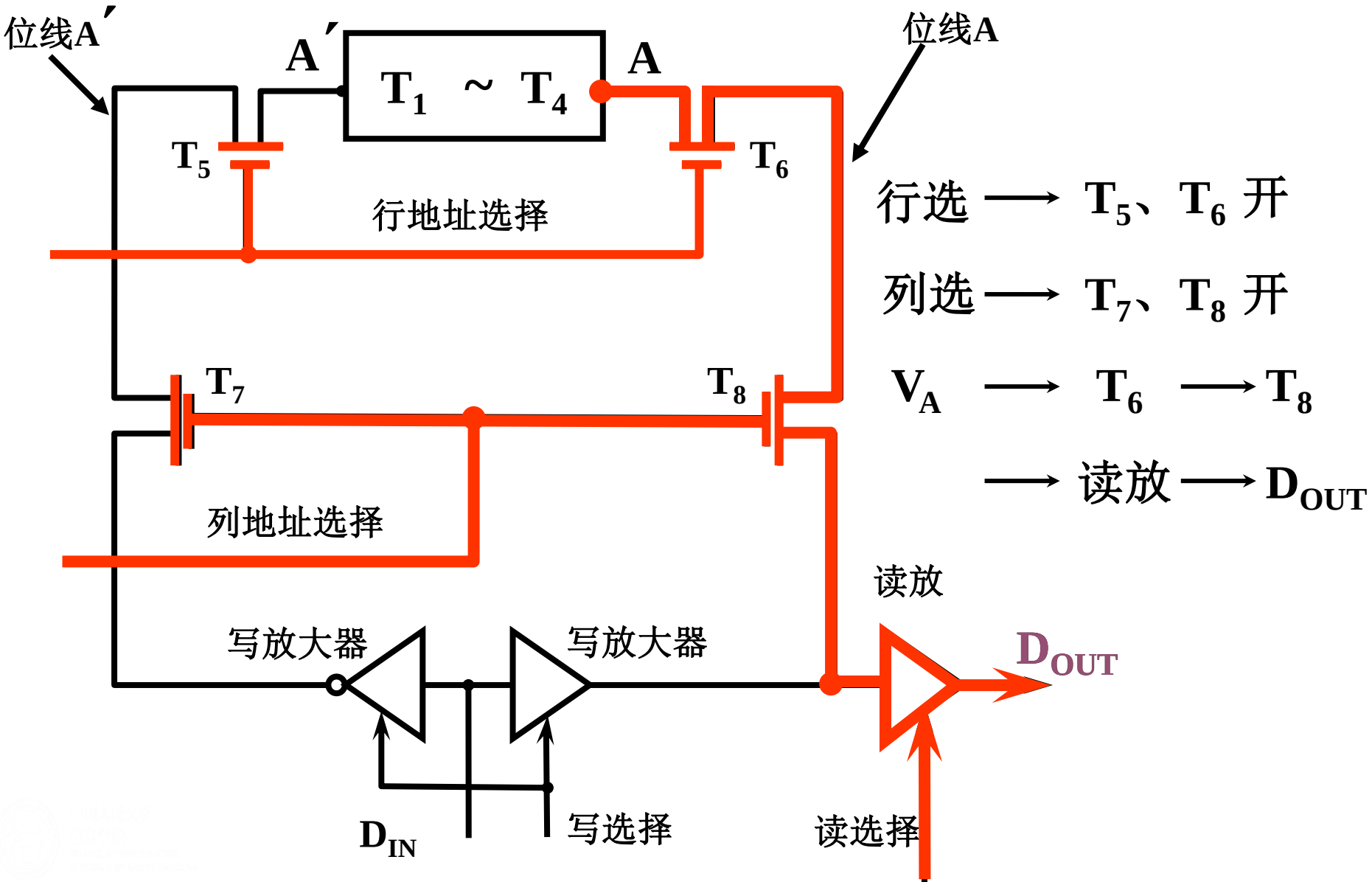
对其要求是：尽可能快的读写速度，尽可能大的存储容量，尽可能低的成本费用。

随机访问存储器(RAM)

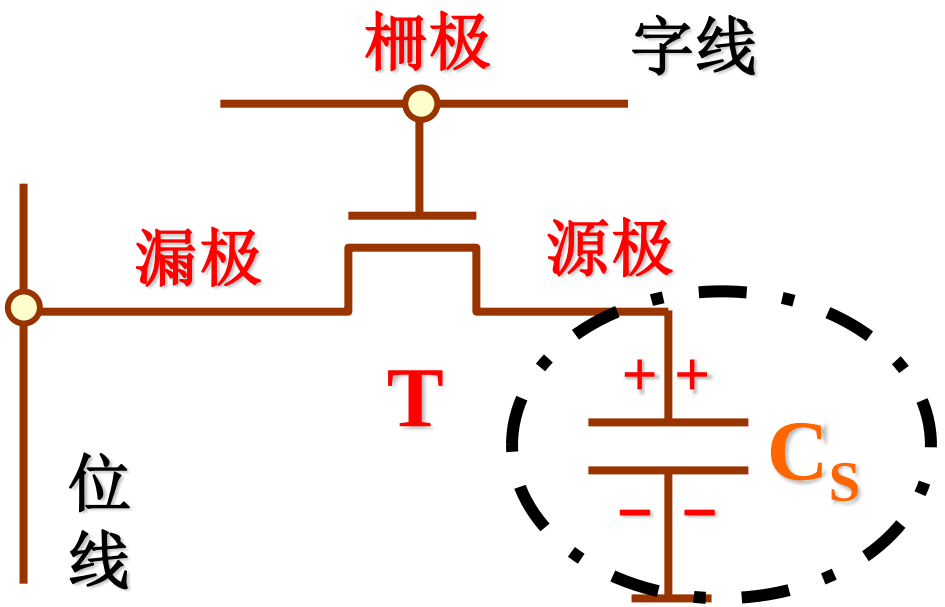
- 特性
  - cell 是基本存储单元(one bit per cell).
  - RAM 制造成芯片.
  - 多个芯片封装为存储器.

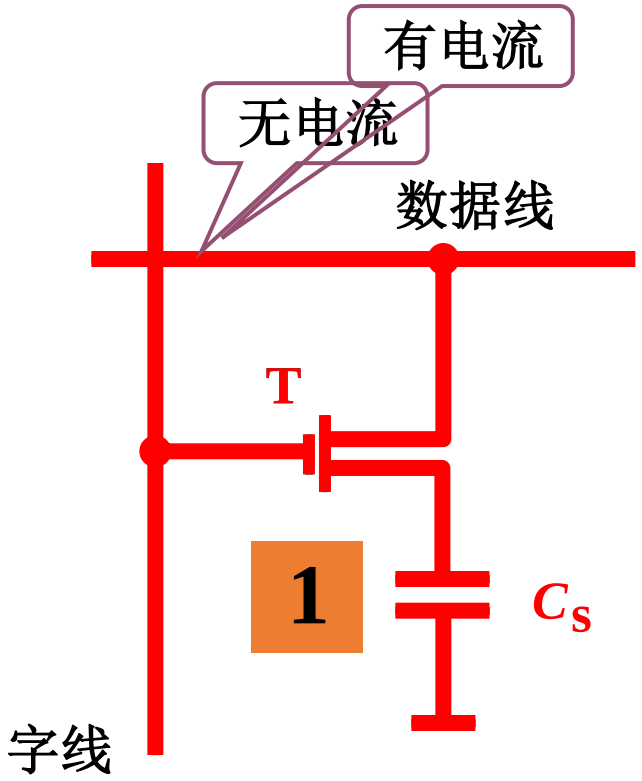
- RAM 有两种类型:
  - SRAM (Static RAM)
  - DRAM (Dynamic RAM)

静态 RAM 基本电路的 读 操作



动态RAM（DRAM）存储元电路





电容 C_S 有无存电荷区分信号 0、1

读出时数据线有电流 为 “1”

写入时 C_S 充电 为 “1” 放电 为 “0”

DRAM的问题

1) 读出重写：读操作时，CS的放电过程会引起信息的丢失，称为破坏性读出。因此必须有一个重写操作。而在写回操作未完成时，不能开始下一次的读操作。

2)定时刷新：由于CS的值很小，又因电容的电荷泄漏需要定时对CS进行充电，以补充电荷，通常间隔2ms。动态存储器也由此而得名。

刷新过程的实质是将原有信息读出，再由刷新放大器形成原信息并重新写入的过程。刷新一次的时间等于一次读写操作的时间。

注意：

动态存储器的刷新方式

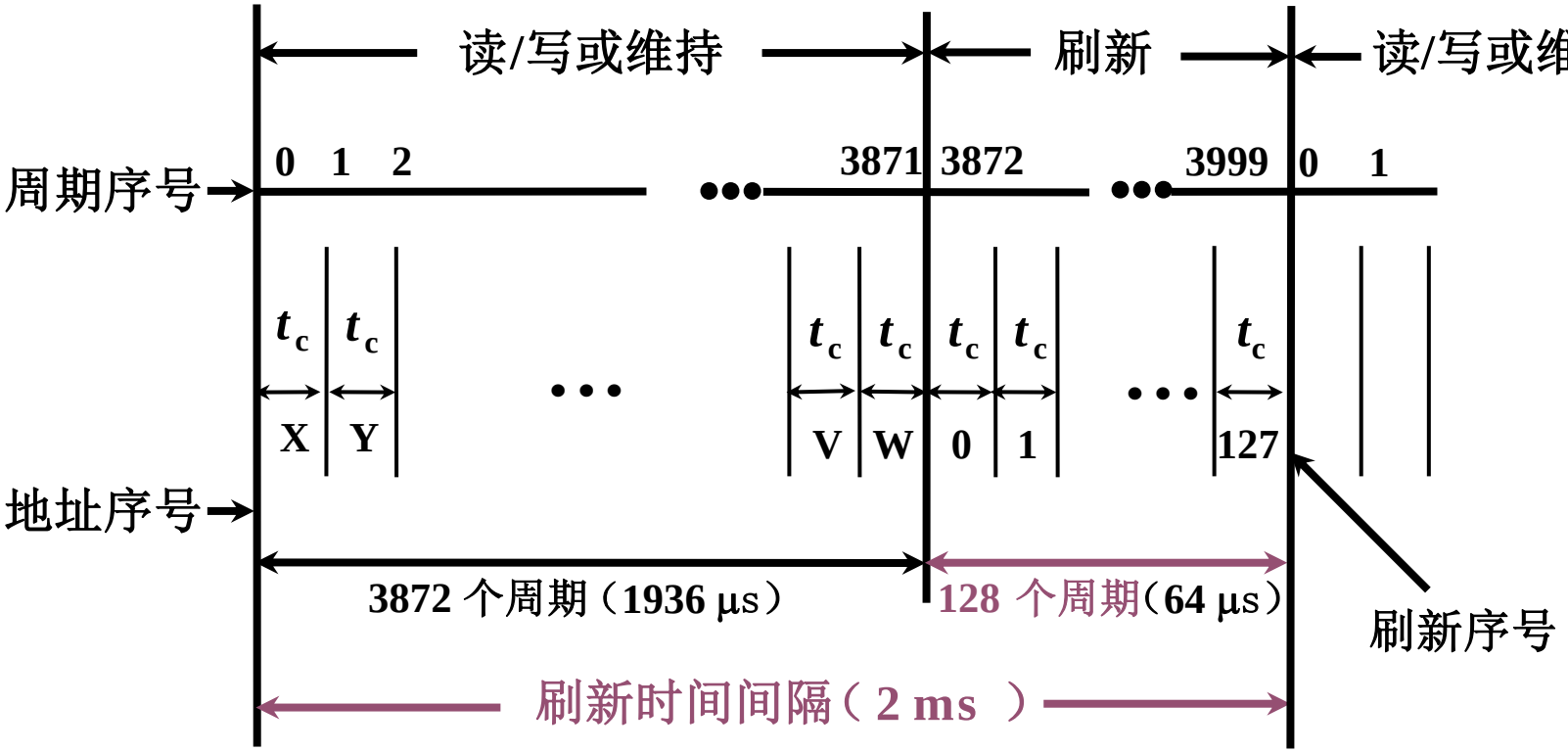
刷新时间 = 存储矩阵行数×存储周期

假设存取周期为500ns(0.5us),则在2ms内共可以安排4000个存取周期。

刷新与行地址有关

集中刷新

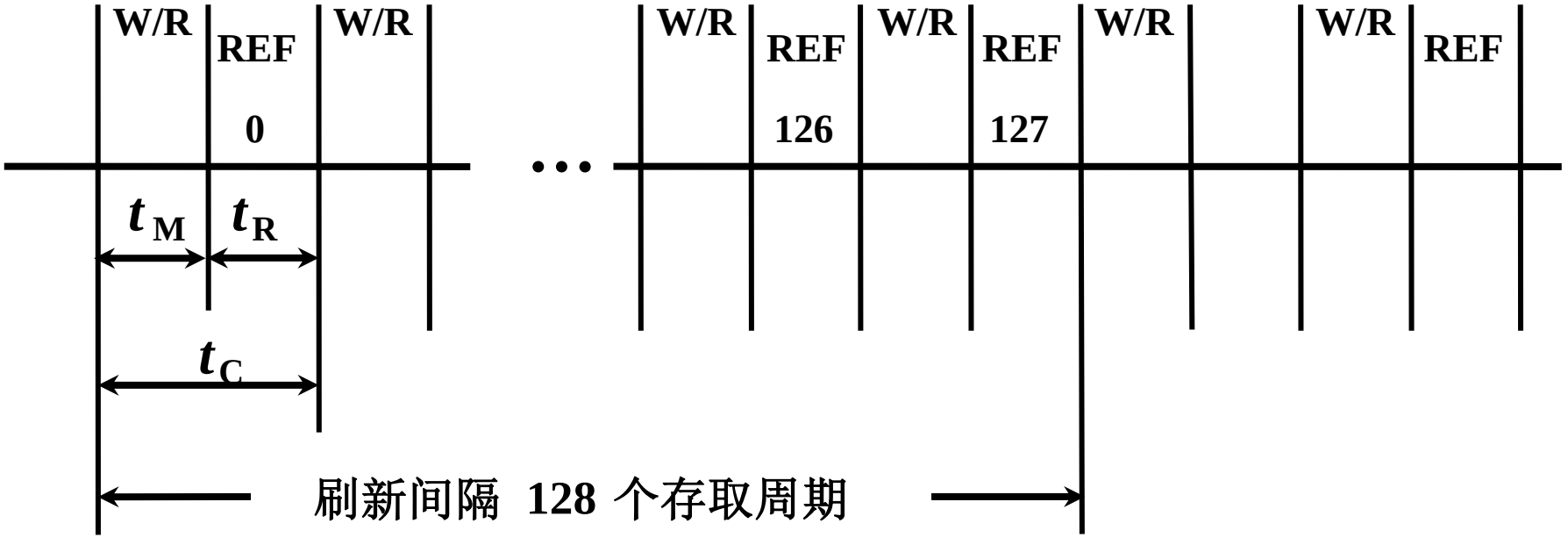
(存取周期为0.5 μs ）　　以128 × 128 矩阵为例



“死区”为　0.5 μs ×128 = 64 μs

“死时间率”为　128/4 000 ×100% = 3.2%

分散刷新



$t_C = t_M + t_R$

读写　刷新

无“死区”

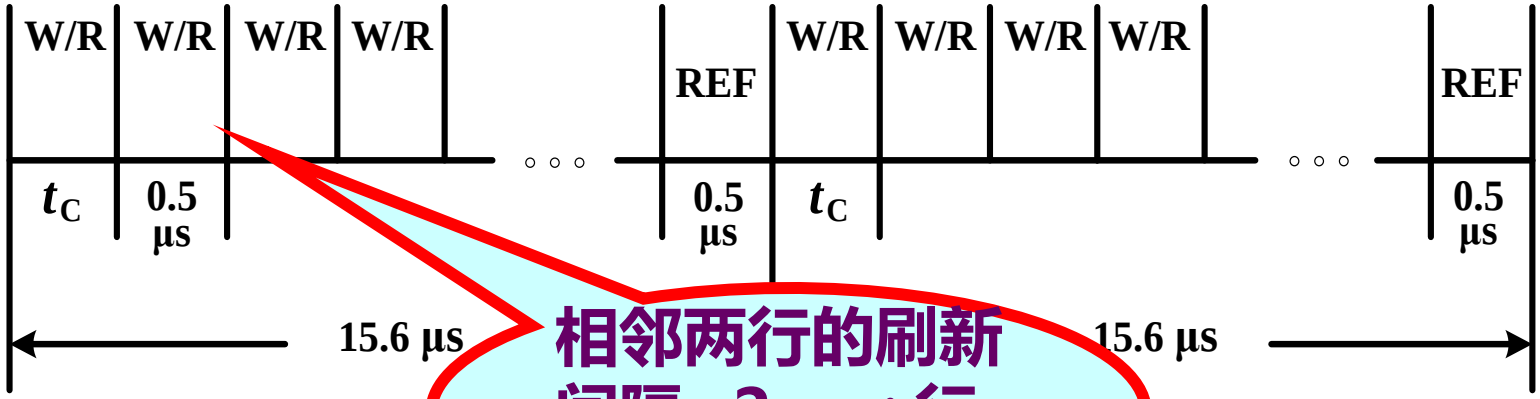
(存取周期为 0.5 μs + 0.5 μs)

这种方式增加了系统的存取时间，假定存储周期为500ns,就要再增加500ns用于刷新。另外是刷新过于频繁，没有利用最大的刷新周期。

分散刷新与集中刷新相结合（异步刷新）

对于 128 ×128 的存储芯片（存取周期为 0.5 μs ）

若每隔 15.6 μs 刷新一行



相邻两行的刷新间隔=2ms÷行数

每行每隔一个“死区”为 0.5 μs

将刷新安排在指令译码阶段，不会出现“死区”

异步刷新方式可避免CPU连续长时间的等待，其用于刷新总的时间开销和集中式刷新一样。是一种比较实用的刷新方式。

静态和动态存储器芯片特性

	SRAM	DRAM
存储信息	触发器	电容
破坏性读出	非	是
需要刷新	不要	需要
送行列地址	同时送	分两次送
运行速度	快	慢
集成度	低	高
发热量	大	小
存储成本	高	低

SRAM vs DRAM

	每比特晶体管	访问时间	是否刷新	需要EDC?	成本	作用
SRAM	6 or 8	1x	No	Maybe	100x	高速缓存
DRAM	1	10x	Yes	Yes	1x	主存

EDC: Error detection and correction

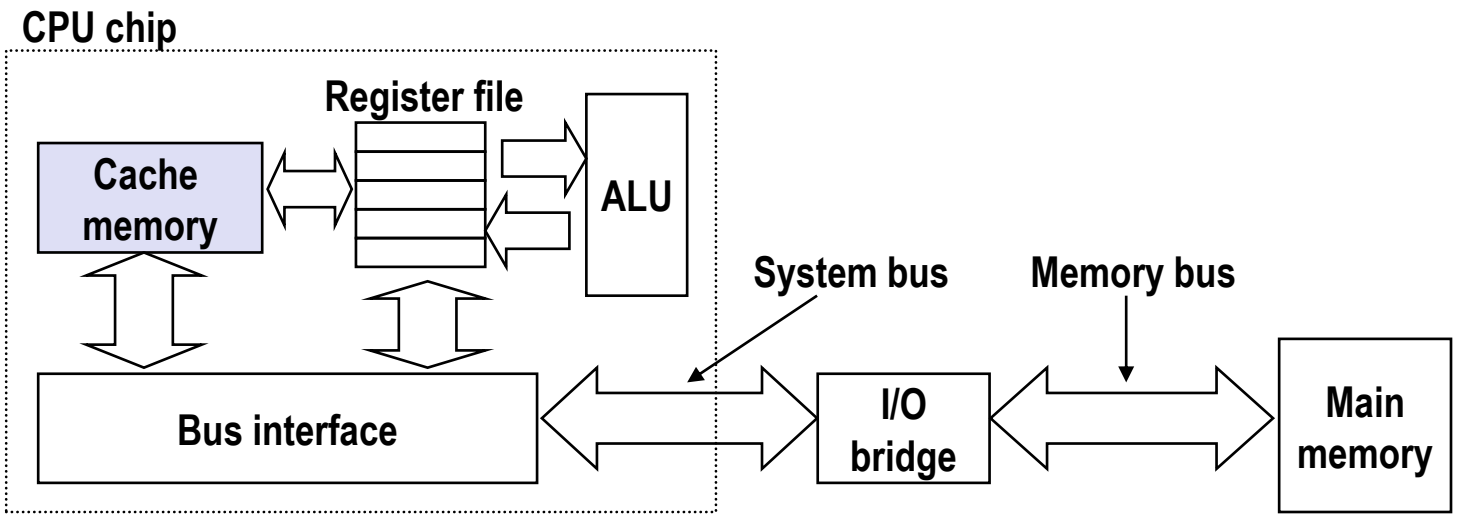
- 趋势
  - SRAM 随着半导体工艺发展
    - 接近极限
  - DRAM 扩展受最小电容的限制
    - 纵横比限制了电容器的深度
    - 也达到了极限

缓冲存储

- **存储器层次的中心思想：**
  - 对于每个*k*，位于*k*层的更快更小的存储设备作为位于*k*+1层的更大更慢存储设备的缓存
- **存储层次的工作原理：**
  - Power law(80/20原则)， *k*+1层数据中部分数据属于频繁访问的热数据，大部分数据属于不频繁访问的冷数据
  - 程序局部性特点倾向于访问*k*层的数据，速度快于*k*+1层数据访问
  - *k*+1层数据访问更慢，但可以提供更大的容量
- ***重要思想:*** **存储层次创建一个大的存储池，成本接近底层存储，性能接近上层存储**

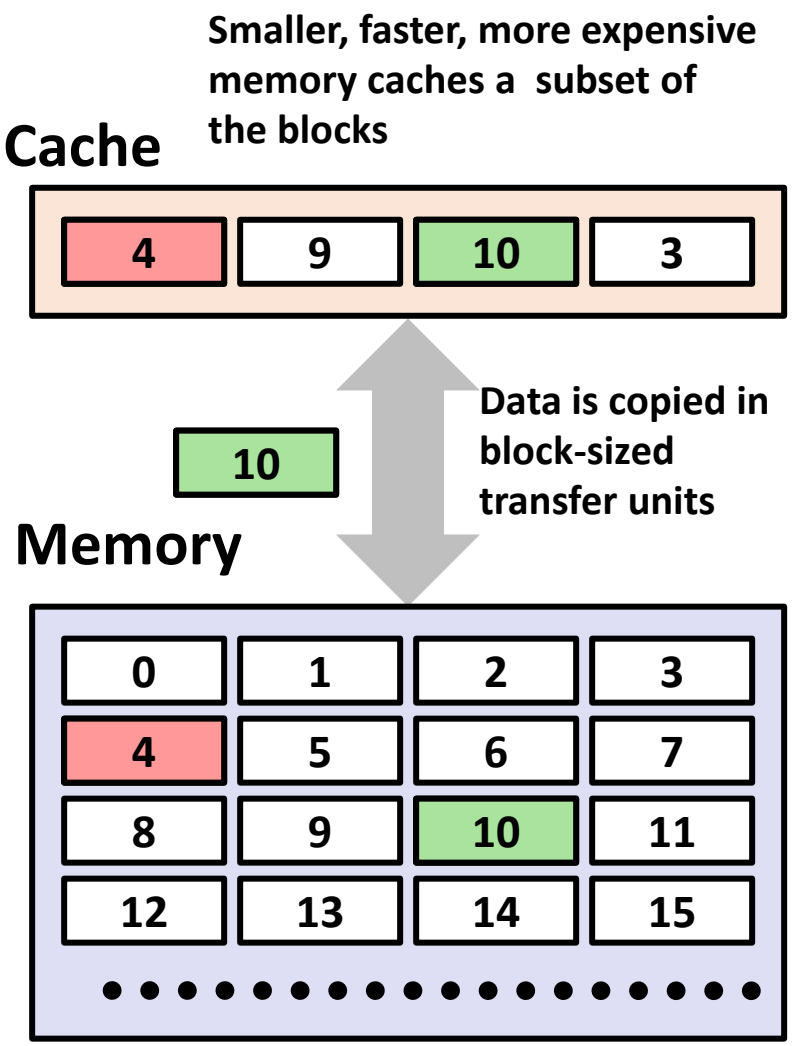
Cache Memories

- 高速缓存存储器是在硬件中自动管理的小型、快速的基于 SRAM 的存储器
- 保存经常访问的主内存块
- CPU 首先在缓存中查找数据
- 典型系统结构：

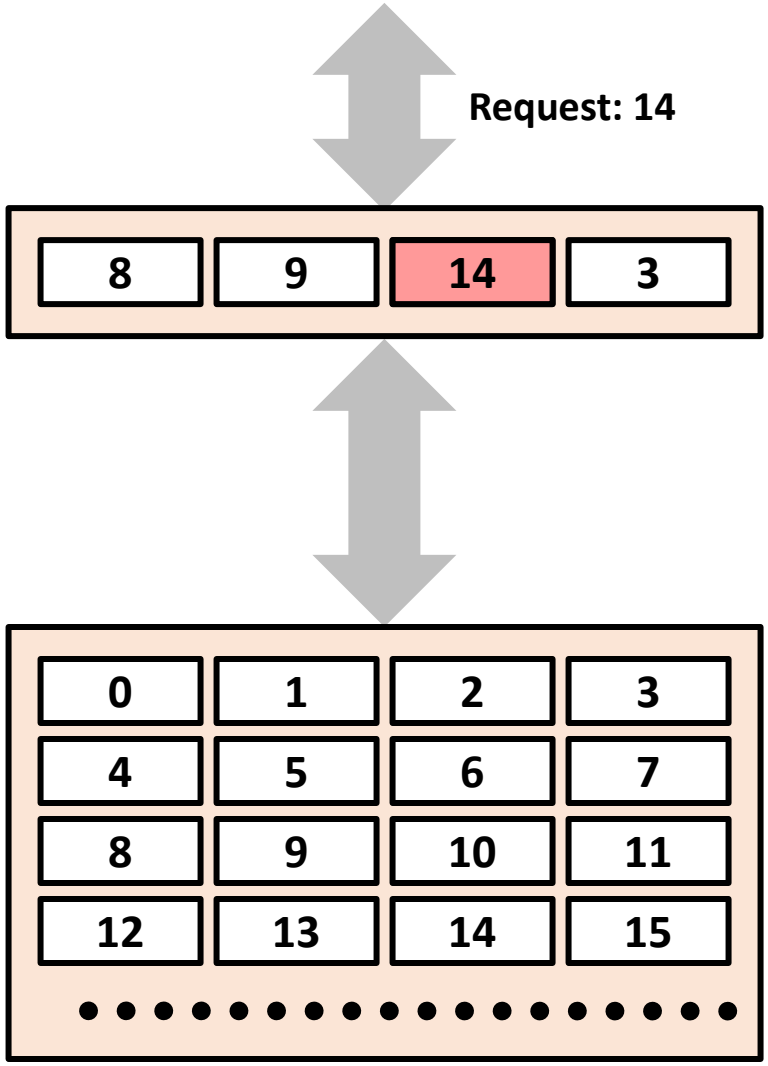


General Cache Concepts

- Cache是一种小容量高速缓冲存储器，它由SRAM组成。
- Cache直接制作在CPU芯片内，速度几乎与CPU一样快。
- 程序运行时，CPU使用的一部分数据/指令会预先成批拷贝在Cache中，Cache的内容是主存储器中部分内容的映象。
- 当CPU需要从内存读(写)数据或指令时，先检查Cache，若有，就直接从Cache中读取，而不用访问主存储器。



General Cache Concepts: Hit



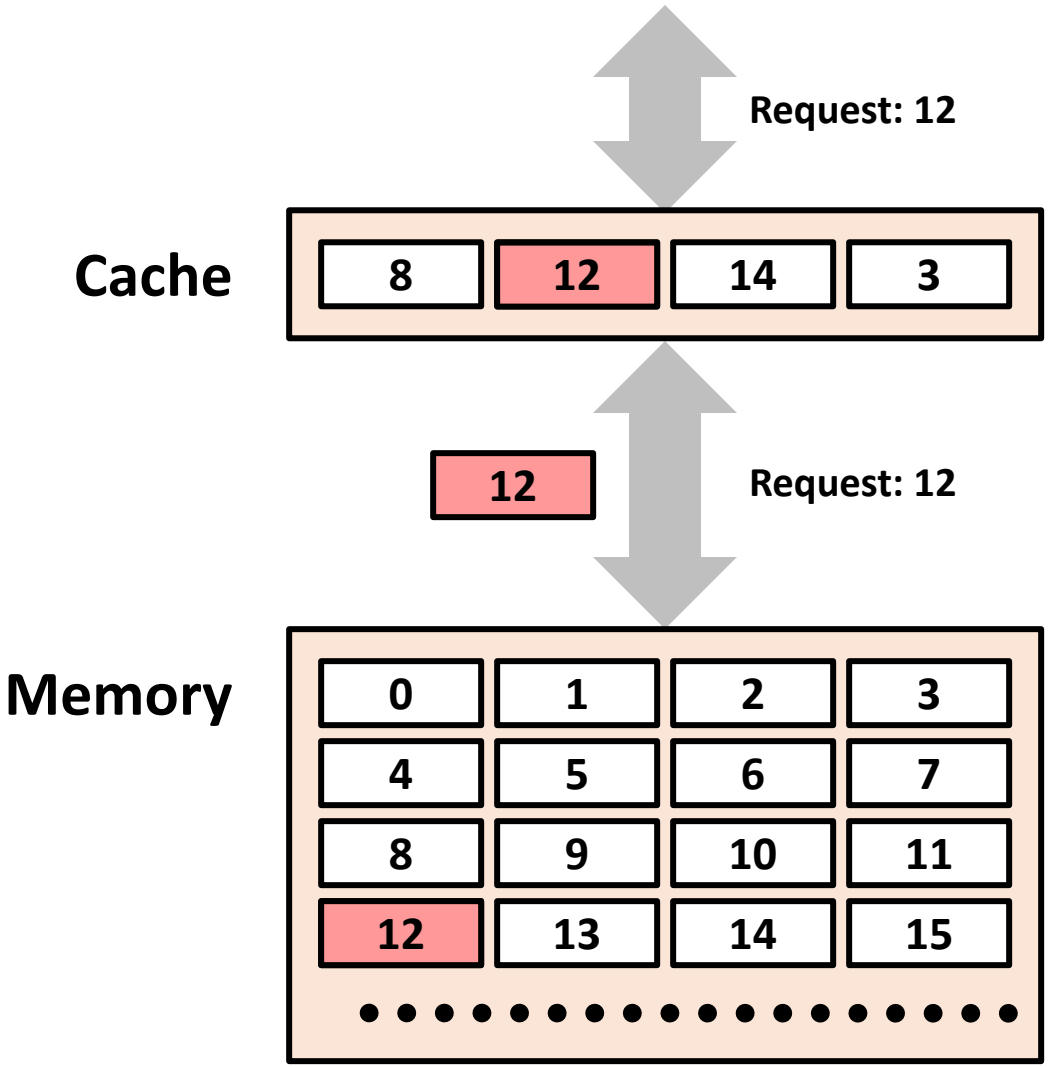
Data in block b is needed

Block b is in cache:
Hit!（命中）

问：若主存的访问时间为60n秒，CACHE的访问时间为10n秒，在命中率为90％时，CPU访存的平均时间为多少?

解：10ns×90％ + 60ns×10％
= 15ns

General Cache Concepts: Miss



Data in block b is needed

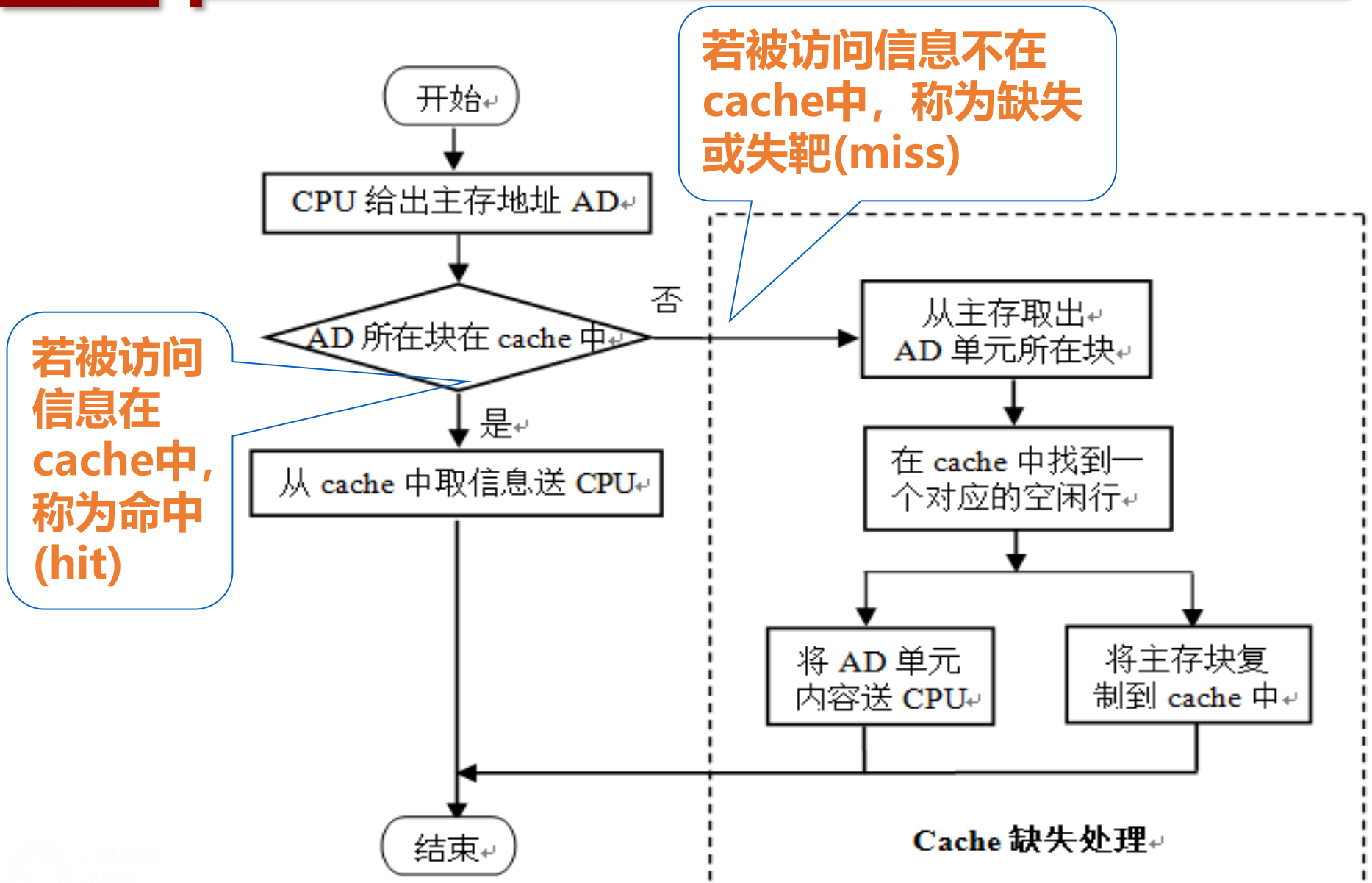
Block b is not in cache:
Miss!（缺失）

Block b is fetched from memory

Block b is stored in cache

- Placement policy放置: determines where b goes
- Replacement policy替换: determines which block gets evicted (victim)

Cache 的操作过程


若被访问信息不在cache中，称为缺失或失靶(miss)
若被访问信息在cache中，称为命中(hit)
开始
CPU 给出主存地址 AD
AD 所在块在 cache 中
否
是
从 cache 中取信息送 CPU
结束
从主存取出AD 单元所在块
在 cache 中找到一个对应的空闲行
将 AD 单元内容送 CPU
将主存块复制到 cache 中
Cache 缺失处理

Cache的块

在存储系统中,把Cache和主存储器都划分成相同大小的块。

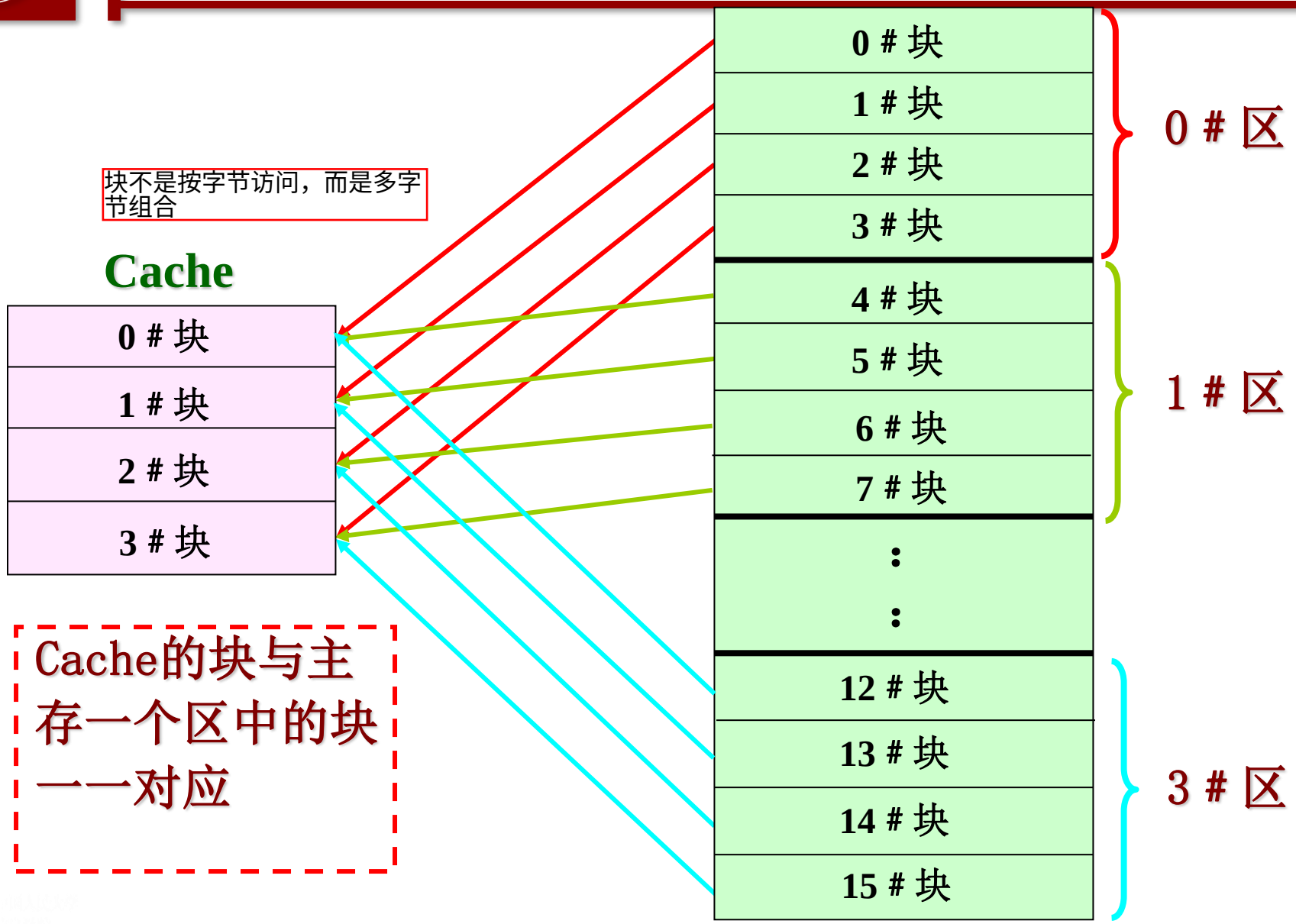
例如：设主存：64KB　　Cache：1KB　　块大小：32B

所以：主存分为2K个块

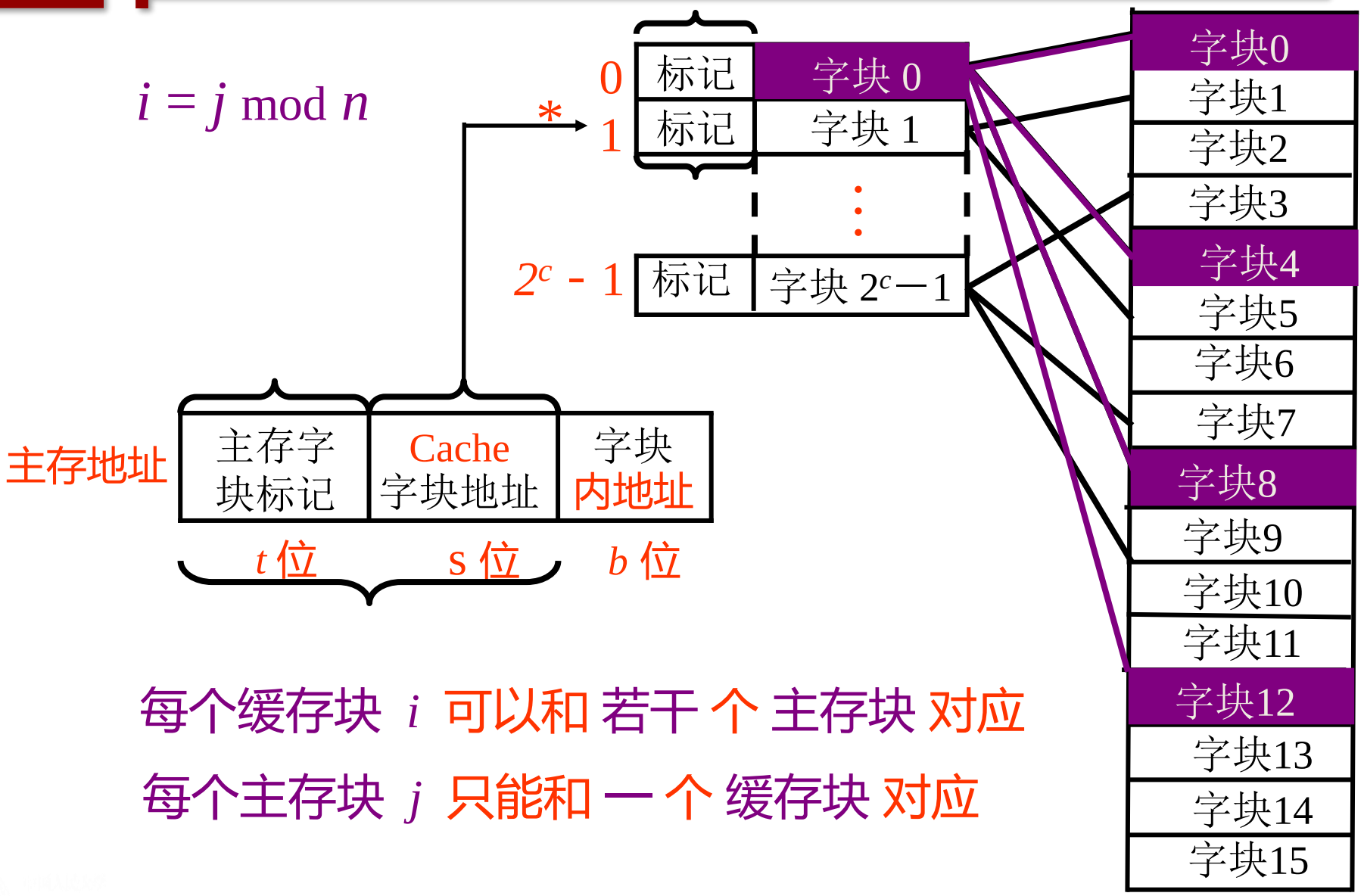
　　CACHE可容纳32个块

- 映像方式
  - 直接映象及其变换
  - 全相联映象及其变换
  - 组相联映象及其变换

直接映象方式示意图


主存
0 # 块
1 # 块
2 # 块
3 # 块
0 # 区
4 # 块
5 # 块
6 # 块
7 # 块
1 # 区
:
:
12 # 块
13 # 块
14 # 块
15 # 块
3 # 区
块不是按字节访问，而是多字节组合
Cache
0 # 块
1 # 块
2 # 块
3 # 块
Cache的块与主存一个区中的块一一对应

直接映象方式



每个缓存块 i 可以和 若干 个 主存块 对应

每个主存块 j 只能和 一 个 缓存块 对应

直接映射高速缓存

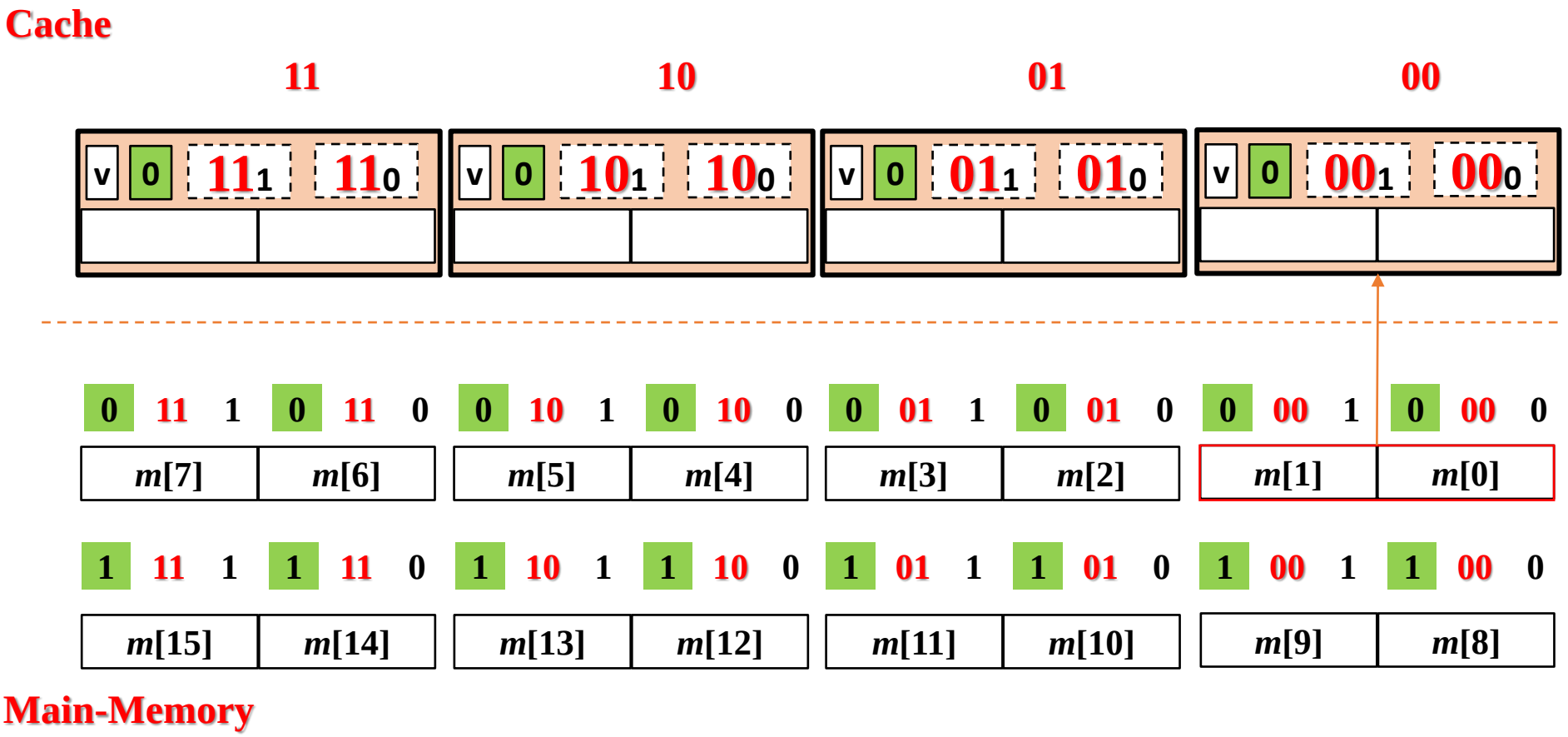
- 假设直接映射高速缓存 **(*S*,*E*,*B*,*m*)=(4,1,2,4)**：
  - 高速缓存有4个组
  - 每组1行
  - 每个块2字节
  - 地址位是4位
- 地址映射
  - 标记位和索引位连起来唯一标识内存中的每个块
  - 有8个内存块，4个高速缓存组，有相同组索引的多个块映射到一个高速缓存组
  - 映射到同一个高速缓存组的块由标记唯一地址标识

cache为8字节，主存为16字节，所以黄色分为两组

地址(十进制)	地址位			块号(十进制)
	标记(t=1)	索引(s=2)	偏移(b=1)	
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Cache缓存示例

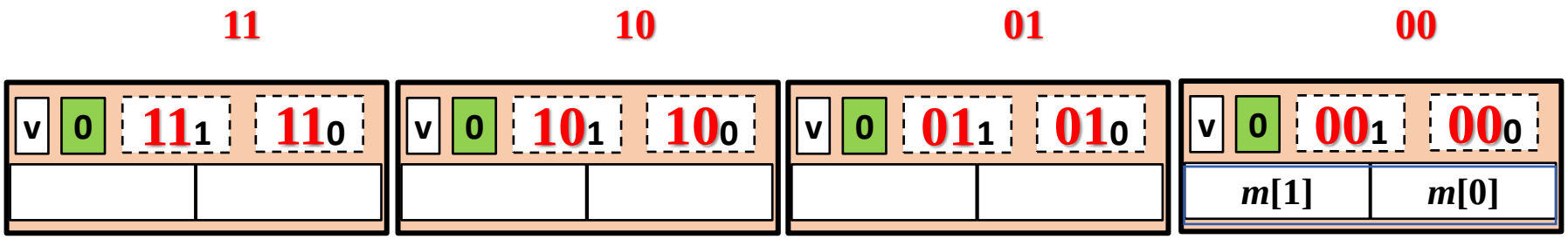
1.读地址0的字

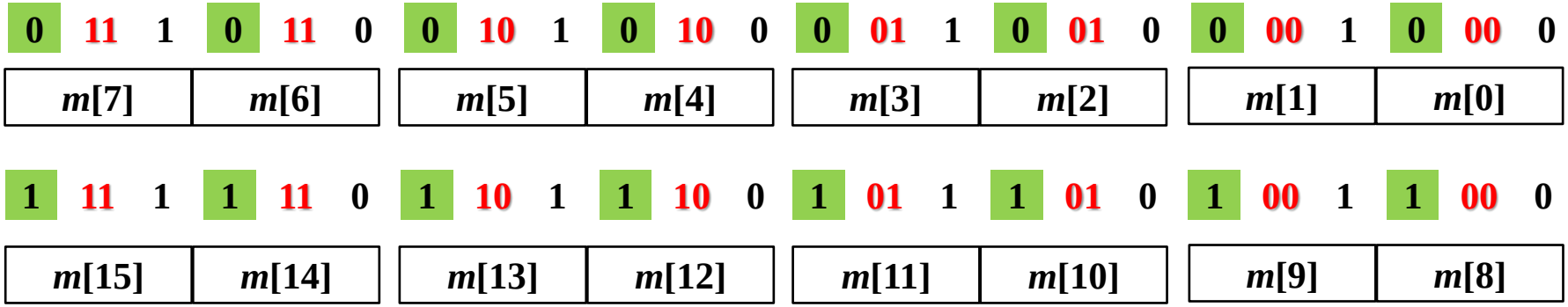


Cache缓存示例

1.读地址0的字

Cache



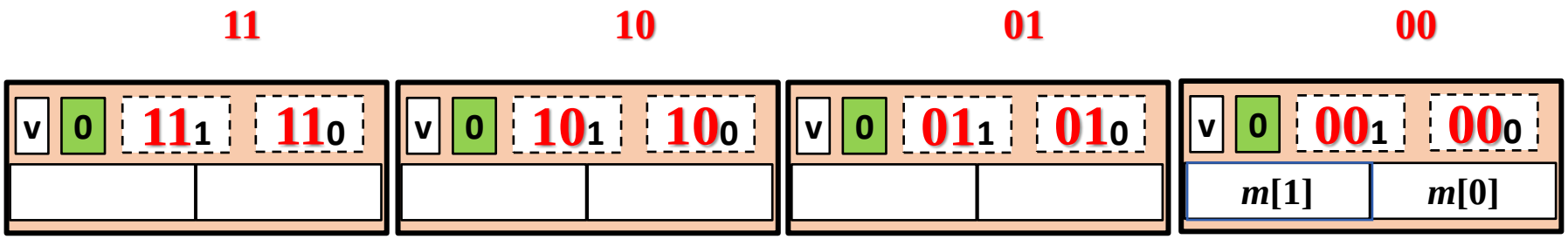


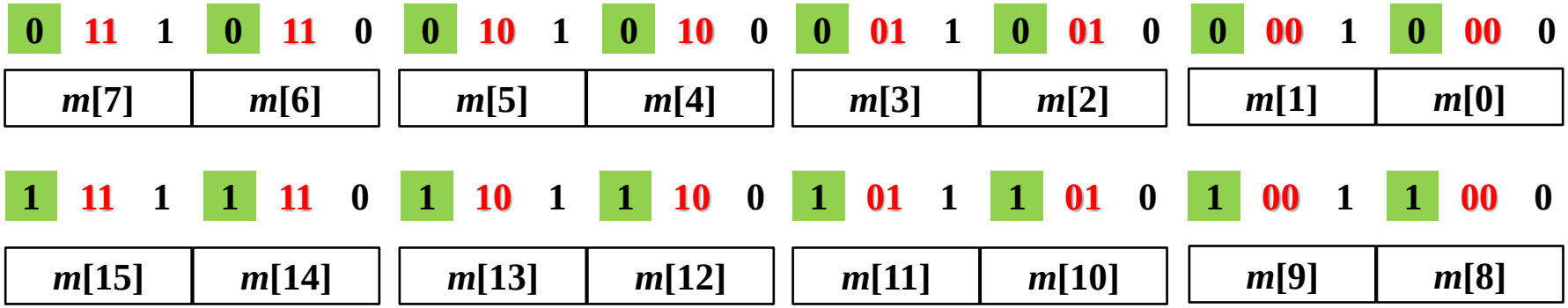
Main-Memory

Cache缓存示例

2.读地址1的字

Cache

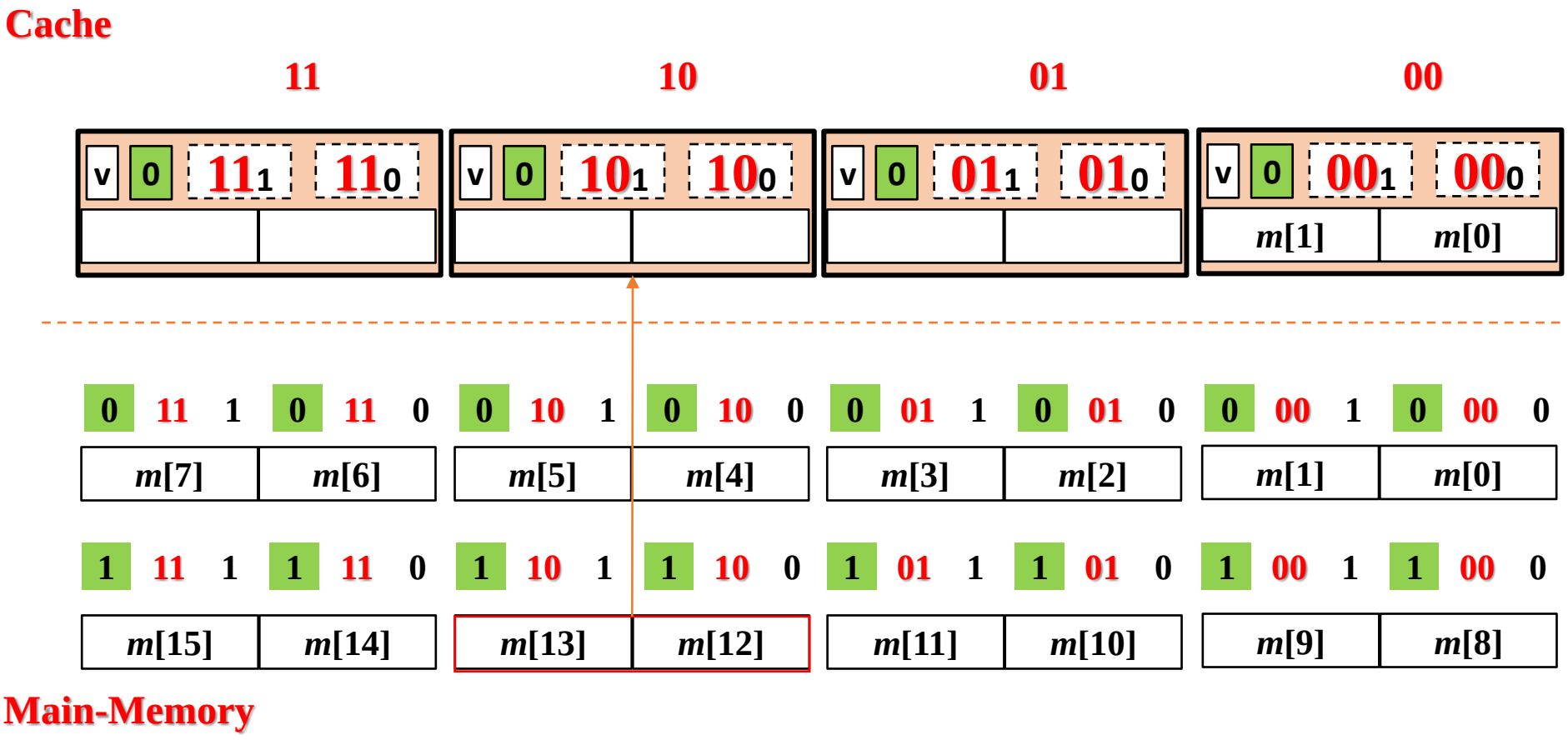




Main-Memory

Cache缓存示例

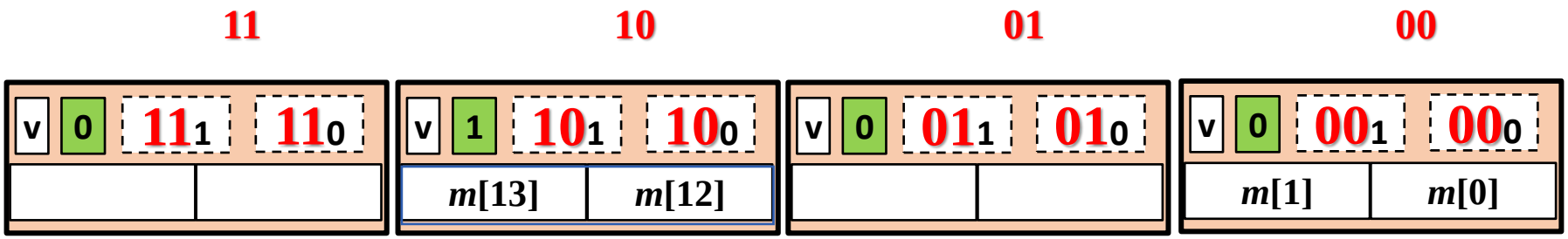
3.读地址13的字

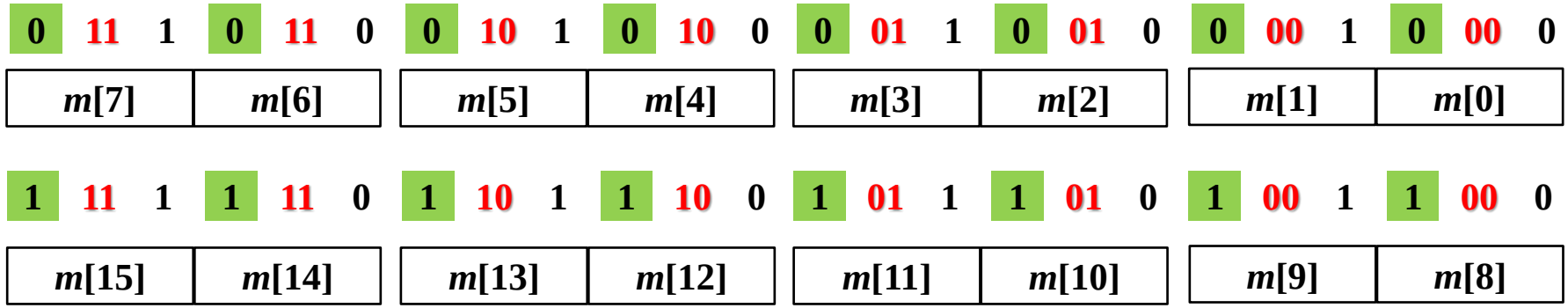


Cache缓存示例

3.读地址13的字

Cache

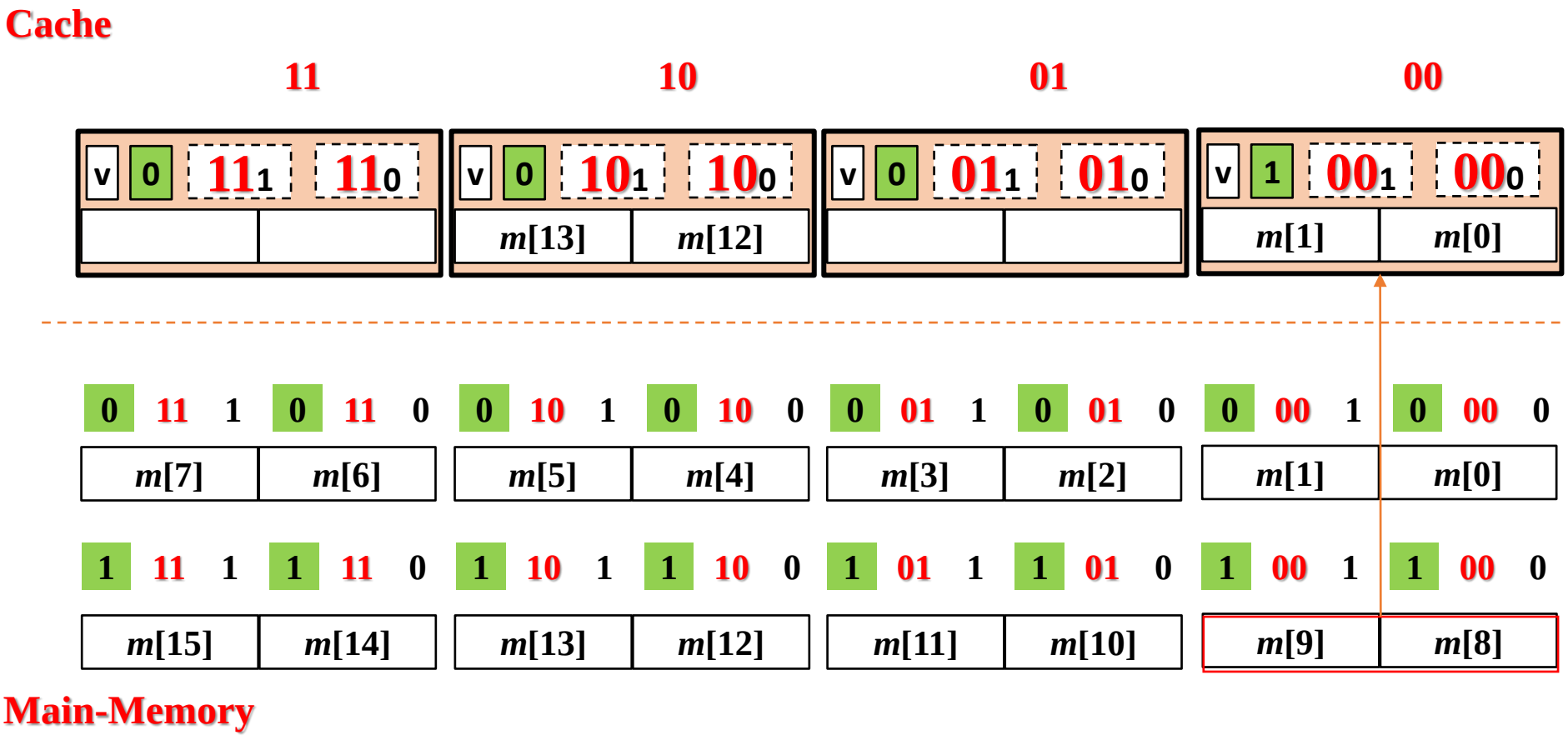




Main-Memory

Cache缓存示例

4.读地址8的字

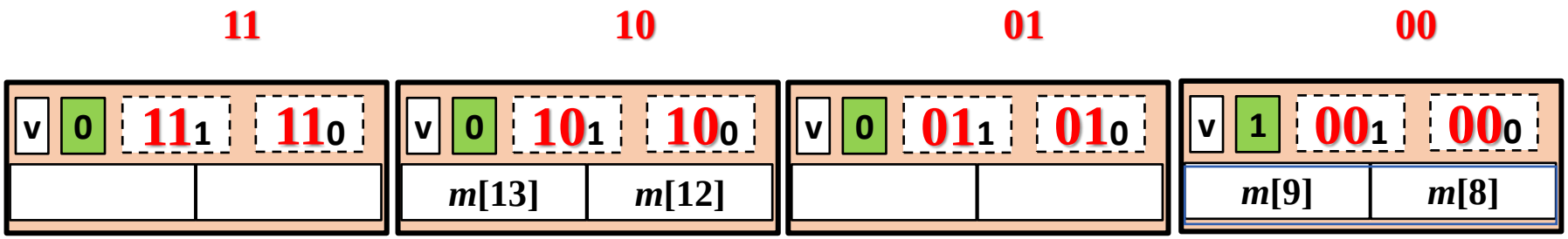


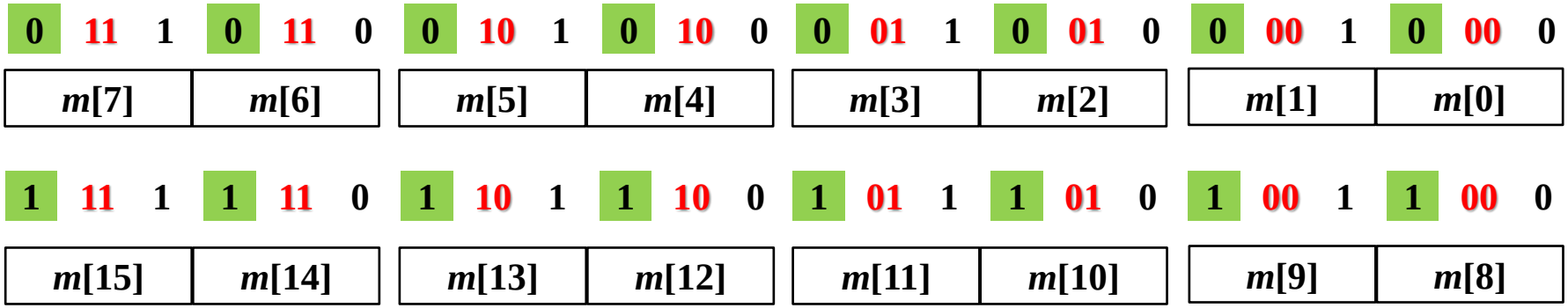
Cache缓存示例

4.读地址8的字

按cathe的块去连续处理主存

Cache

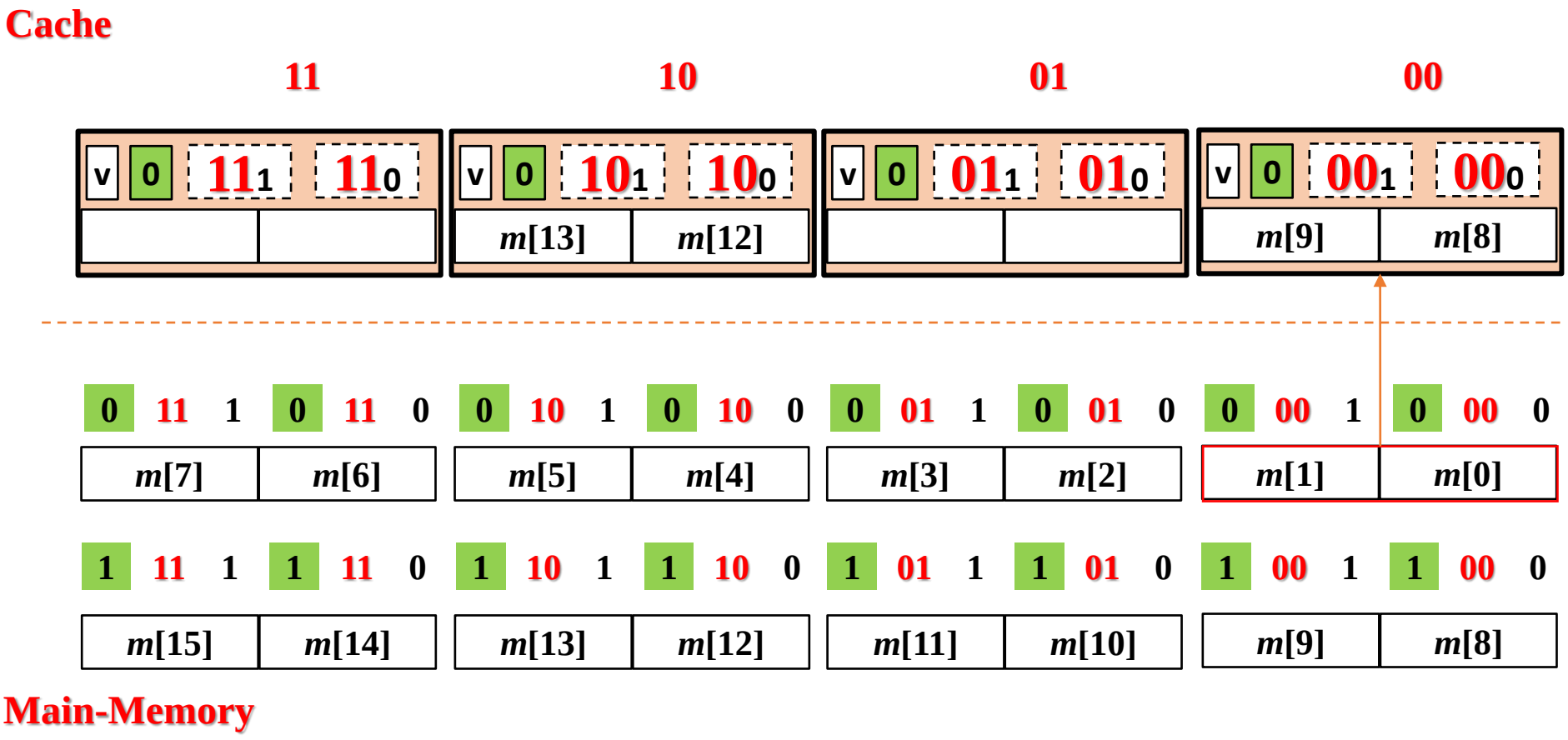




Main-Memory

Cache缓存示例

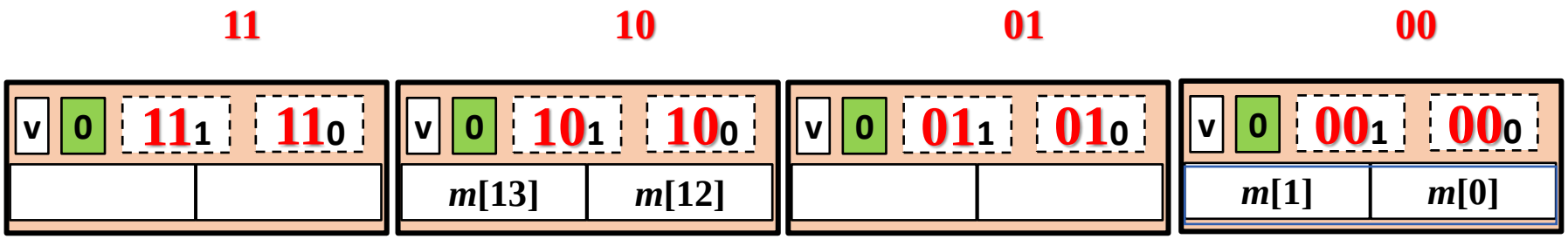
5.读地址0的字

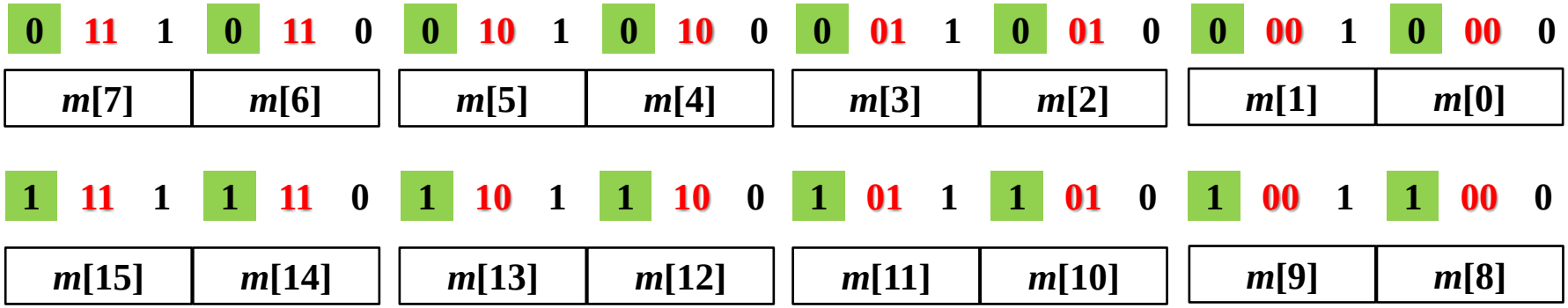


Cache缓存示例

5.读地址0的字

Cache

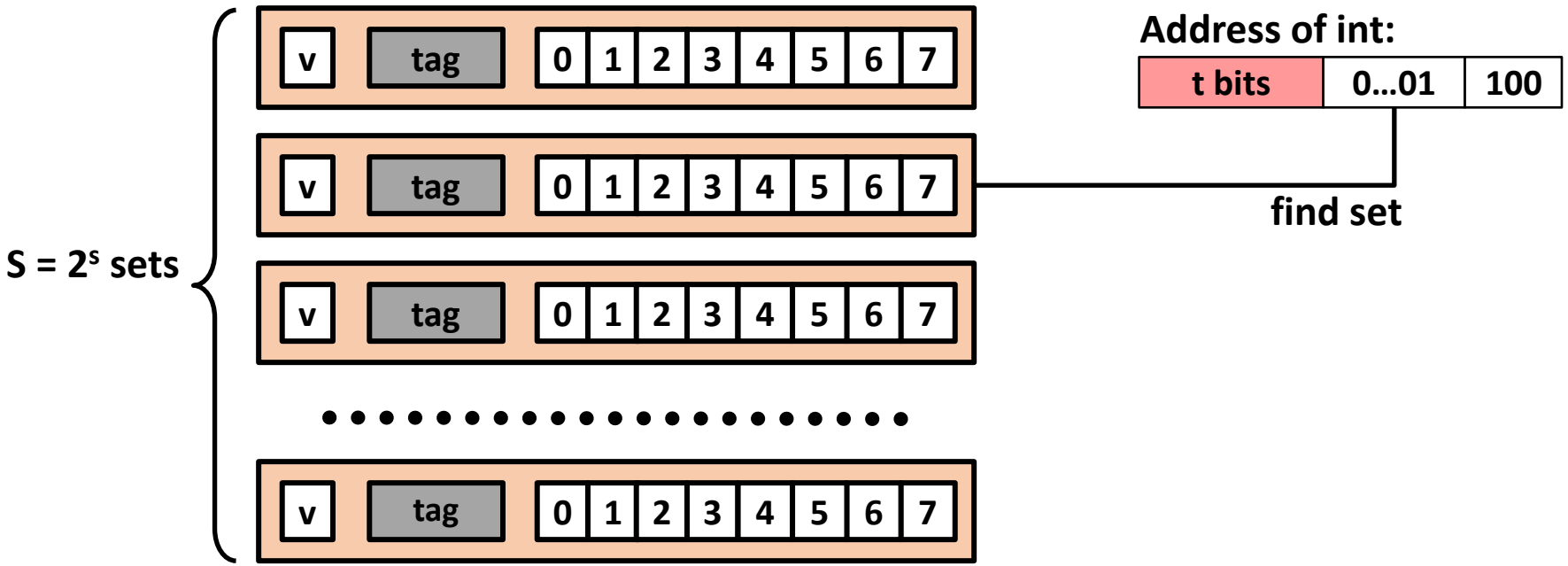




Main-Memory

Example: Direct Mapped Cache (E = 1)

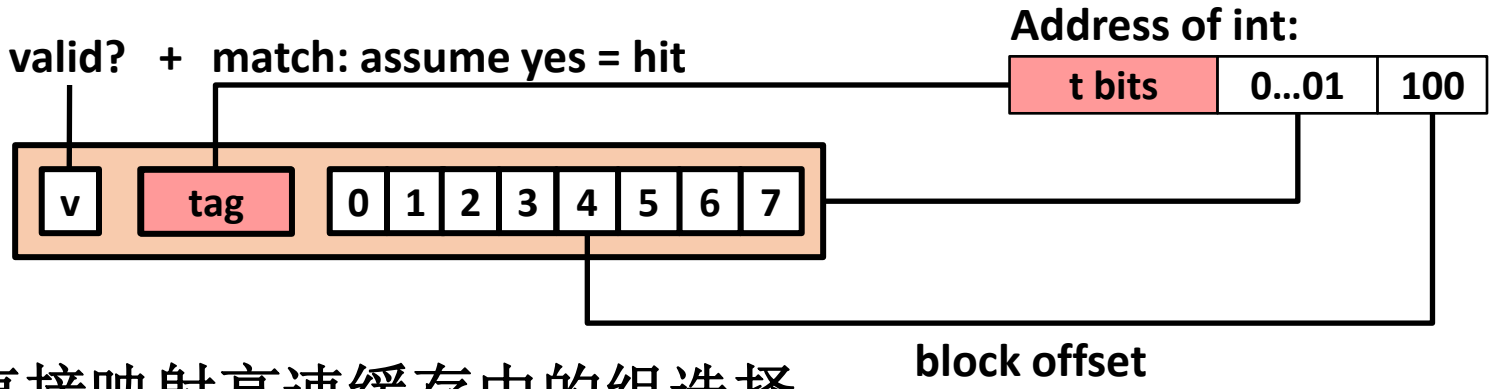
直接映射：每组一行
假设：缓存块大小 8 字节



- 1.直接映射高速缓存中的组选择
  - 通过组索引位(0…01)选择高速缓存中的组

Example: Direct Mapped Cache (E = 1)

直接映射：每组一行
假设：缓存块大小 8 字节

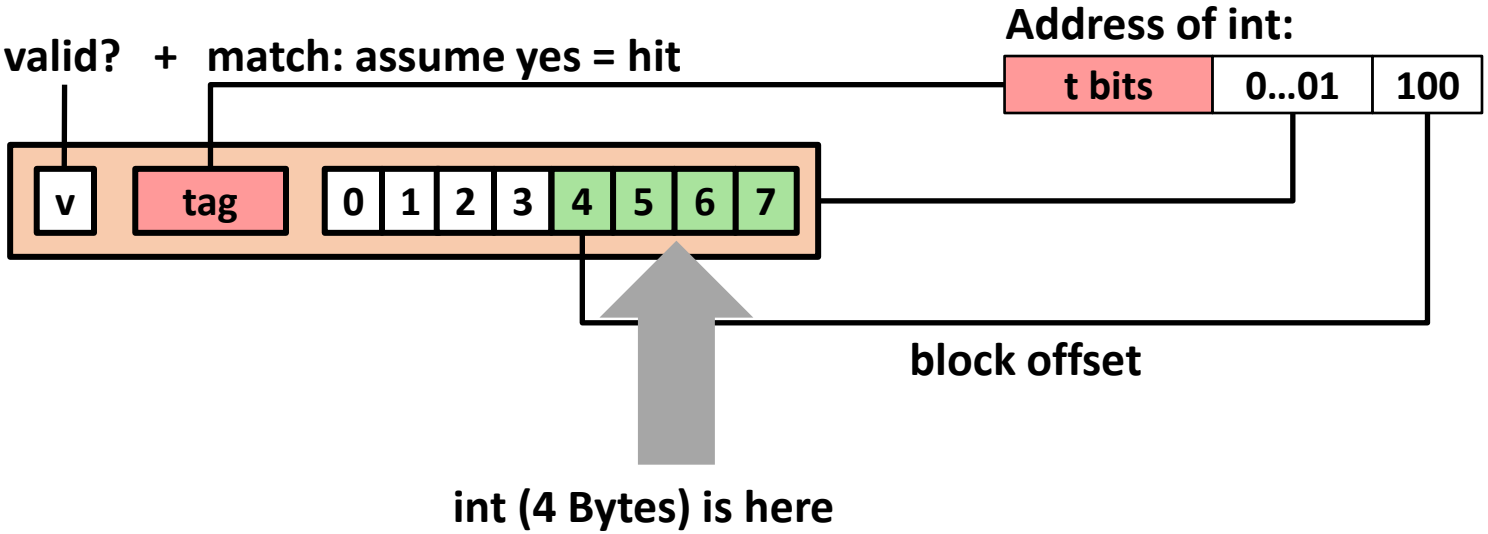


- **1.直接映射高速缓存中的组选择**
  - 通过组索引位(0…01)选择高速缓存中的组
- **2.直接映射高速缓存中的行匹配**
  - 标记位为1，并且tag标记位与地址中的标记位相匹配
- **3.直接映射高速缓存中的字选择**
  - 块偏移位(100)提供了所需要的字的第一个字节在cache block中的偏移地址

Example: Direct Mapped Cache (E = 1)

直接映射：每组一行
假设：缓存块大小 8 字节

- 4.直接映射高速缓存不命中时的行替换
  - 如果不命中，从下一级存储器中取出请求的块，替换当前行



如果 tag 不匹配：旧行被驱逐并替换

填空题 5分

设置

下面的访问如果命中填H，缺失填M

t=1	s=2	b=1
x	xx	x

4-bit addresses (address space size M=16 bytes)
S=4 sets, E=1 Blocks/set, B=2 bytes/block

Address trace (reads, one byte per read):

0	[0000_2],	[填空1]
1	[0001_2],	[填空2]
7	[0111_2],	[填空3]
8	[1000_2],	[填空4]
0	[0000_2]	[填空5]

	v	Tag	Block
Set 0	0		
Set 1	0		
Set 2	0		
Set 3	0		

作答

直接映射Cache

t=1	s=2	b=1
x	xx	x

4-bit addresses (address space size M=16 bytes)
S=4 sets, E=1 Blocks/set, B=2 bytes/block

Address trace (reads, one byte per read):

0	**[0000_2],**	miss
1	**[0001_2],**	hit
7	**[0111_2],**	miss
8	**[1000_2],**	miss
0	**[0000_2]**	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1	0		
Set 2	0		
Set 3	1	0	M[6-7]

直接映射高速缓存中的冲突不命中

- 两个向量点积函数
  - 具有良好的空间局部性
  - Cache命中率仍然较低

```
Float dotprod(float x[8], float y[8])
{
    float sum=0.0;
    int i;

    for(i=0;i<8;i++)
        sum+=x[i]*y[i];
    return sum;
}
```

x				*y*			
元素	地址	组索引	二进制地址	元素	地址	组索引	二进制地址
x[0]	0	0	000000	*y*[0]	32	0	100000
x[1]	4	0	000100	*y*[1]	36	0	100100
x[2]	8	0	001000	*y*[2]	40	0	101000
x[3]	12	0	001100	*y*[3]	44	0	101100
x[4]	16	1	010000	*y*[4]	48	1	110000
x[5]	20	1	010100	*y*[5]	52	1	110100
x[6]	24	1	011000	*y*[6]	56	1	111000
x[7]	28	1	011100	*y*[7]	60	1	111100

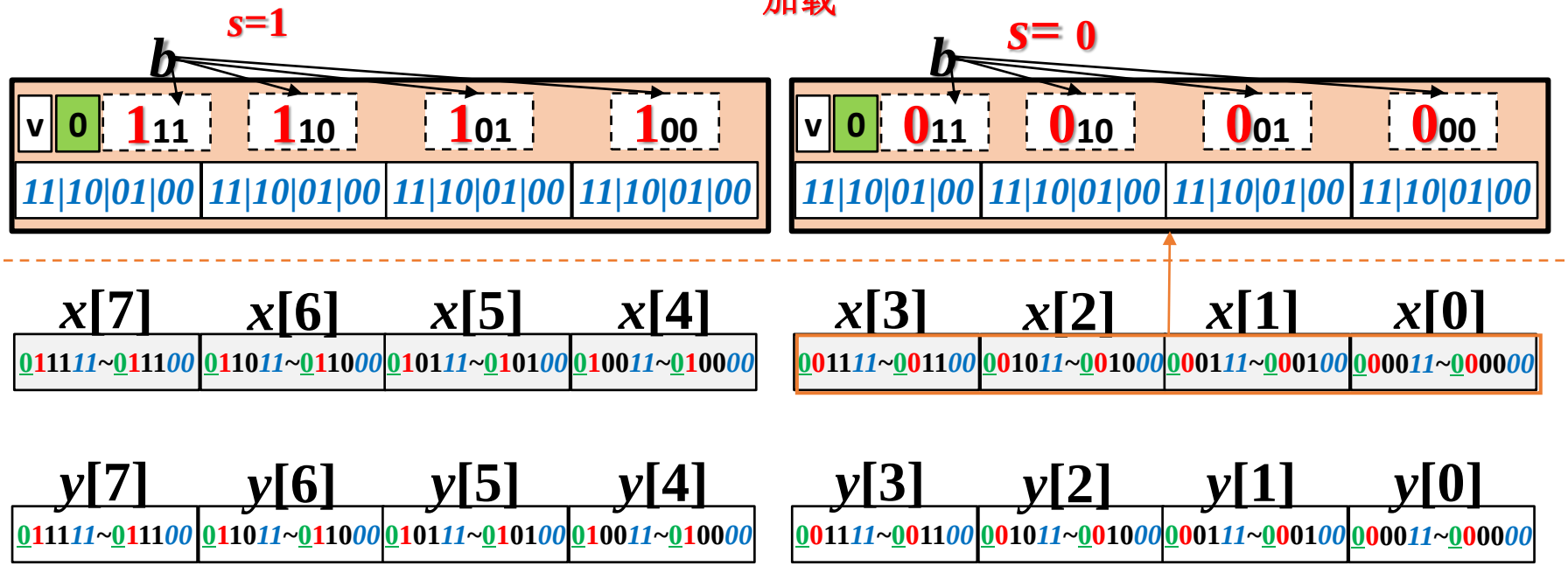
- 参数设置：
  - 浮点数4字节
  - X加载到从0到32个字节连续内存
  - Y加载到从32开始的连续内存
  - 一个块为16字节

内存与cache地址分布

```
sum+=x[0]*y[0];
```

加载

Cache



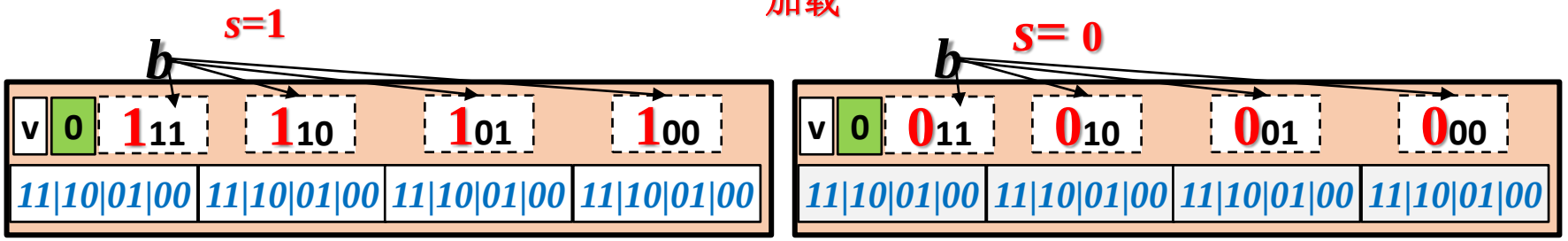
Main-Memory

内存与cache地址分布

```
sum+=x[0]*y[0];
```

加载

Cache

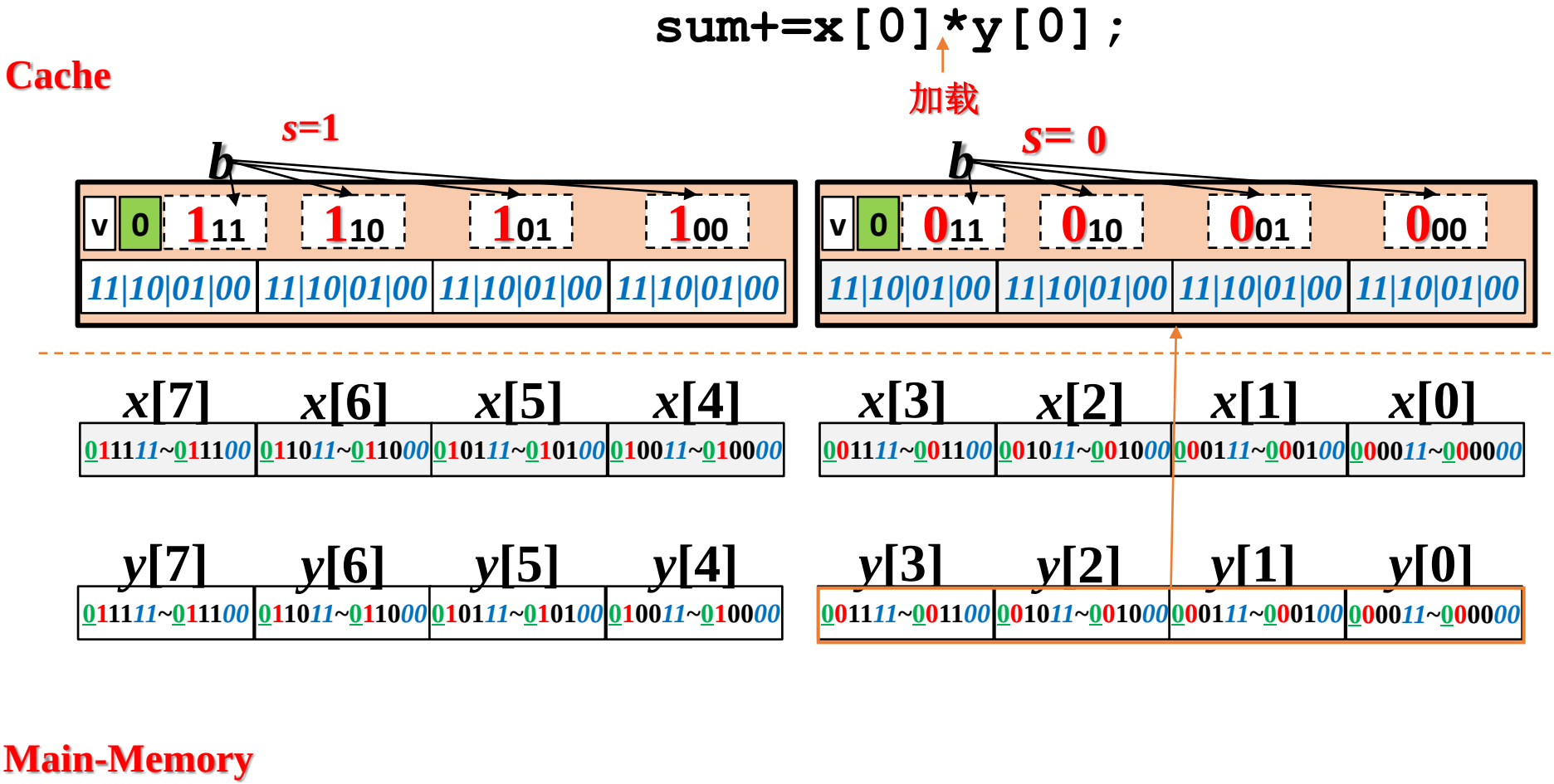


x[7]	x[6]	x[5]	x[4]	x[3]	x[2]	x[1]	x[0]
01111*11*~01111*00*	01101*11*~01100*00*	01011*11*~01010*00*	01001*11*~01000*00*	00111*11*~00111*00*	00101*11*~00100*00*	00011*11*~00010*00*	00001*11*~00000*00*

y[7]	y[6]	y[5]	y[4]	y[3]	y[2]	y[1]	y[0]
01111*11*~01111*00*	01101*11*~01100*00*	01011*11*~01010*00*	01001*11*~01000*00*	00111*11*~00111*00*	00101*11*~00100*00*	00011*11*~00010*00*	00001*11*~00000*00*

Main-Memory

内存与cache地址分布

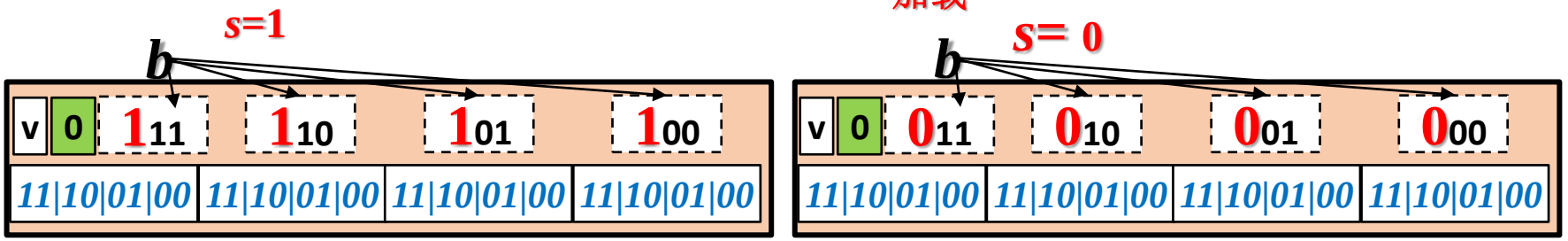


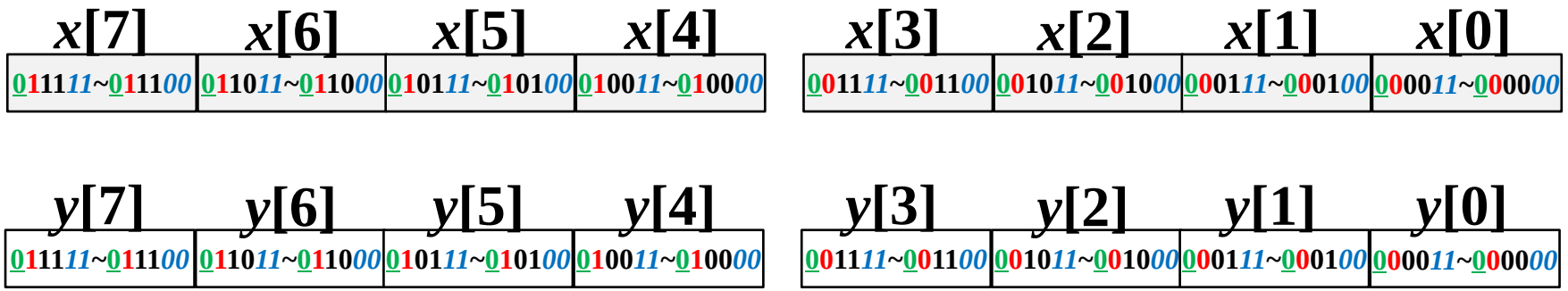
内存与cache地址分布

```
sum+=x[0]*y[0];
```

加载

Cache





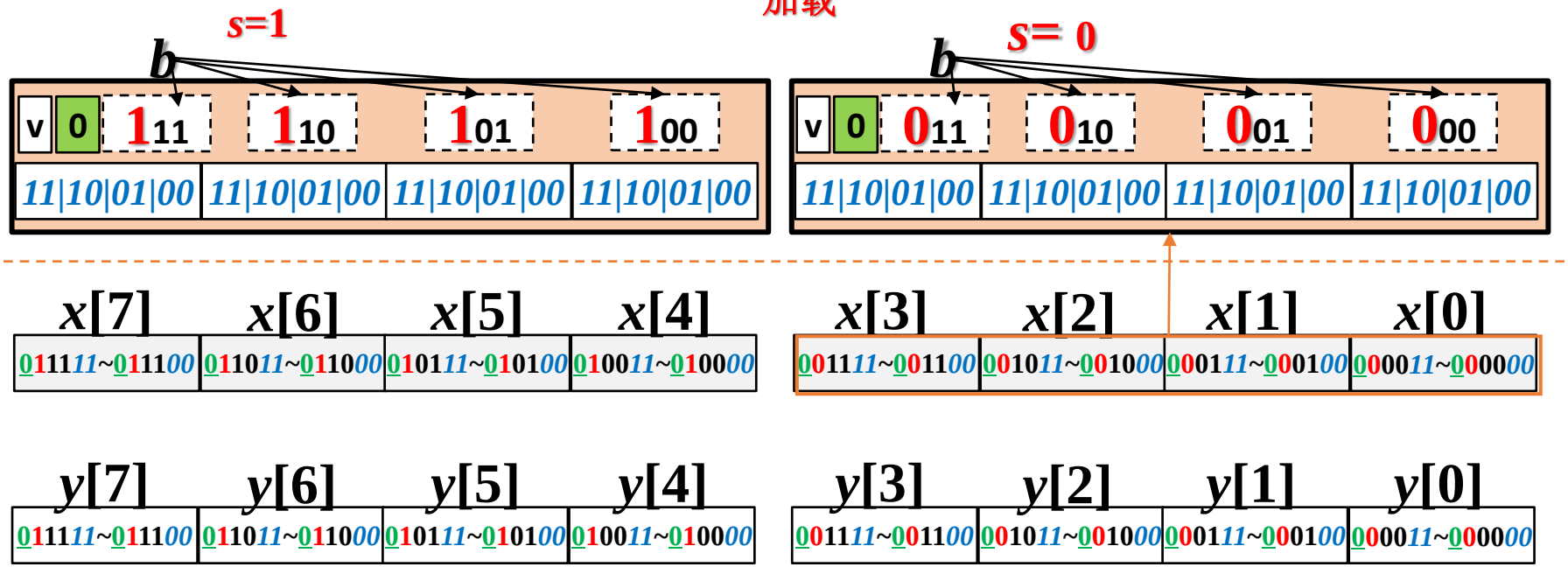
Main-Memory

内存与cache地址分布

```
sum+=x[1]*y[1];
```

加载

Cache



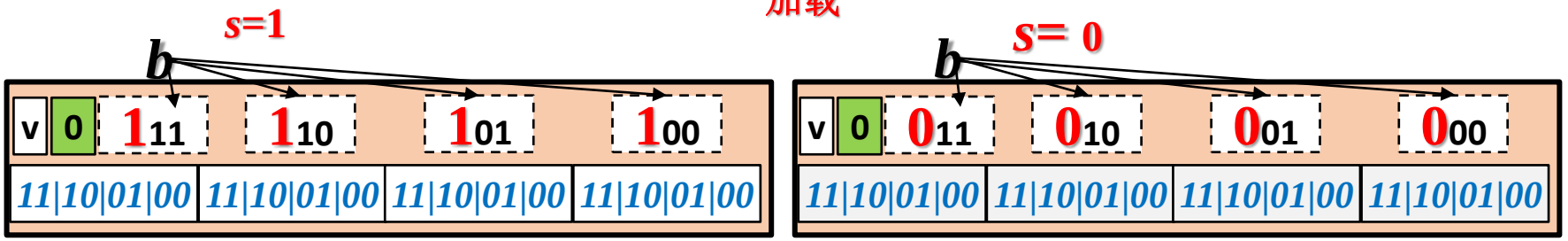
Main-Memory

内存与cache地址分布

```
sum+=x[1]*y[1];
```

加载

Cache

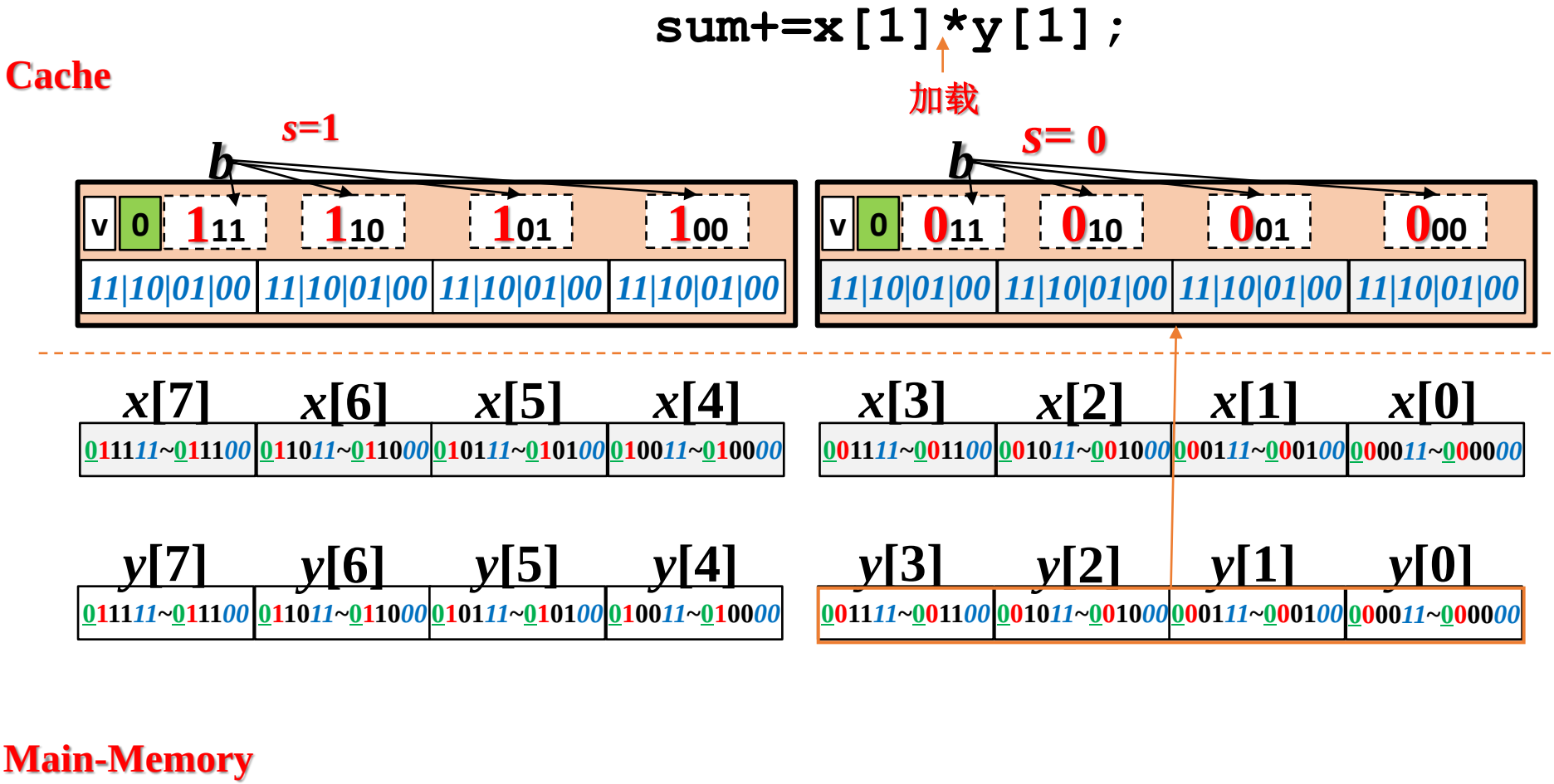


x[7]	x[6]	x[5]	x[4]	x[3]	x[2]	x[1]	x[0]
01111100	01101100	01011100	01001100	00111100	00101100	00011100	00001100

y[7]	y[6]	y[5]	y[4]	y[3]	y[2]	y[1]	y[0]
01111100	01101100	01011100	01001100	00111100	00101100	00011100	00001100

Main-Memory

内存与cache地址分布

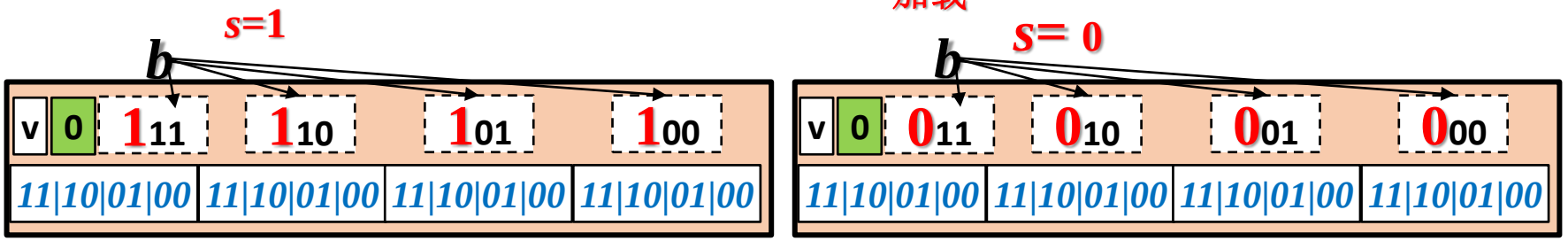


内存与cache地址分布

```
sum+=x[1]*y[1];
```

加载

Cache



x[7]	x[6]	x[5]	x[4]	x[3]	x[2]	x[1]	x[0]
011111~011100	011011~011000	010111~010100	010011~010000	001111~001100	001011~001000	000111~000100	000011~000000

y[7]	y[6]	y[5]	y[4]	y[3]	y[2]	y[1]	y[0]
011111~011100	011011~011000	010111~010100	010011~010000	001111~001100	001011~001000	000111~000100	000011~000000

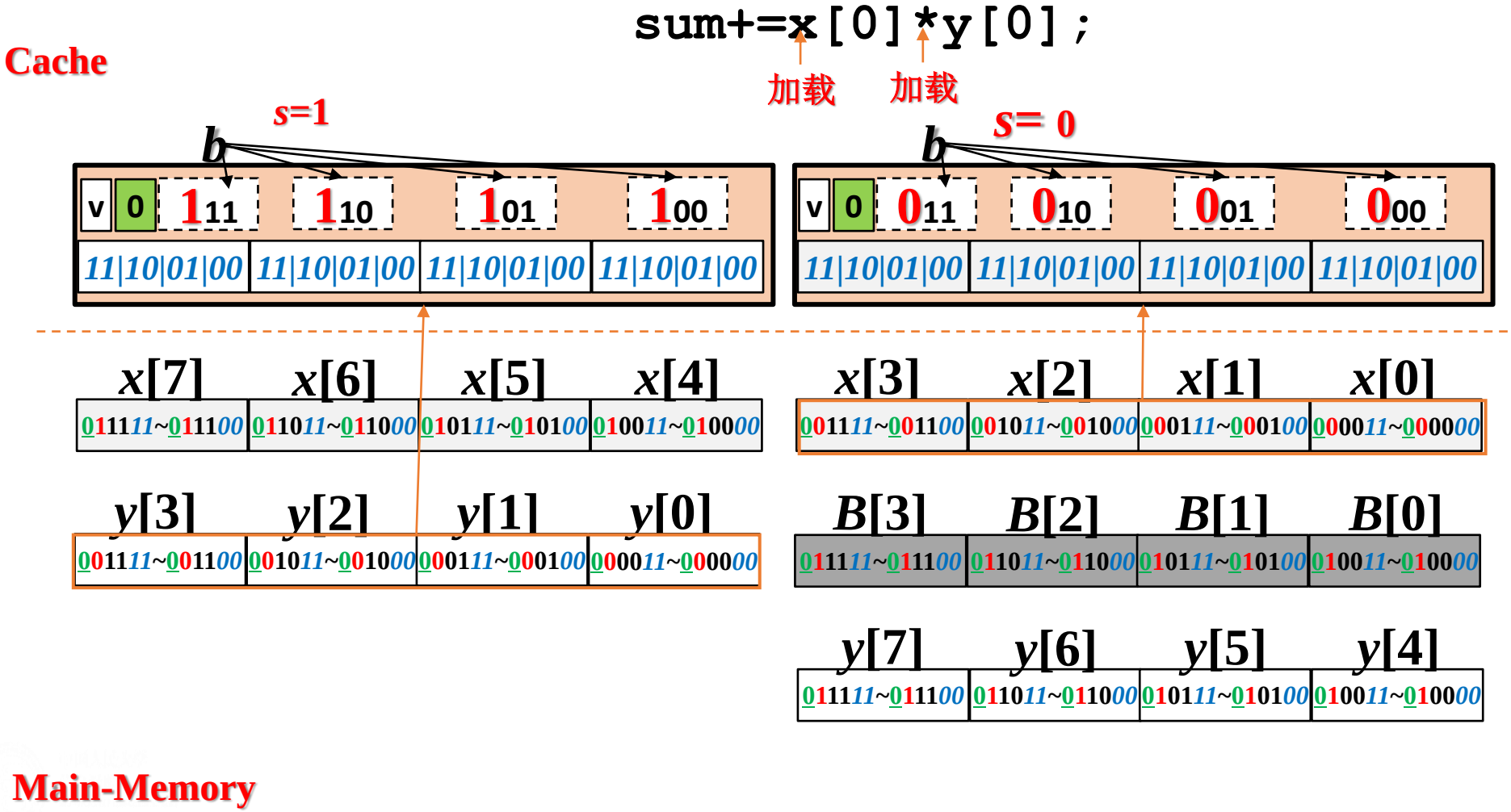
Main-Memory

Cache thrashing原因分析

- 抖动(trash)：高速缓存反复地加载和驱逐相同的高速缓存块组
- 原因：
  - 循环迭代处理语句中*x*[*i*]和*y*[*i*]内存块对应相同的cache组
  - 读取*x*[*i*]加载4个连续的浮点数到cache组
  - 读取*y*[*i*]加载4个连续的浮点数到相同的cache组，原*x*[*i*]数据块被cache驱逐
  - 读取*x*[*i*+*1*]需要重新加载4个连续的浮点数到cache组
  - 读取*y*[*i*+*1*] 重新驱逐已加载的4个连续的*x*[*i*+*1*]对应的浮点数
  - 周而复始…
  - 导致每个*x*[*i*]和*y*[*i*]产生一个cache miss

填充解决Cache thrashing

- 在数组尾部增加B字节填充，使$x[i]$和$y[i]$内存块对应不同的cache组
- 每4个浮点数只加载一次，命中3次，命中率为75%

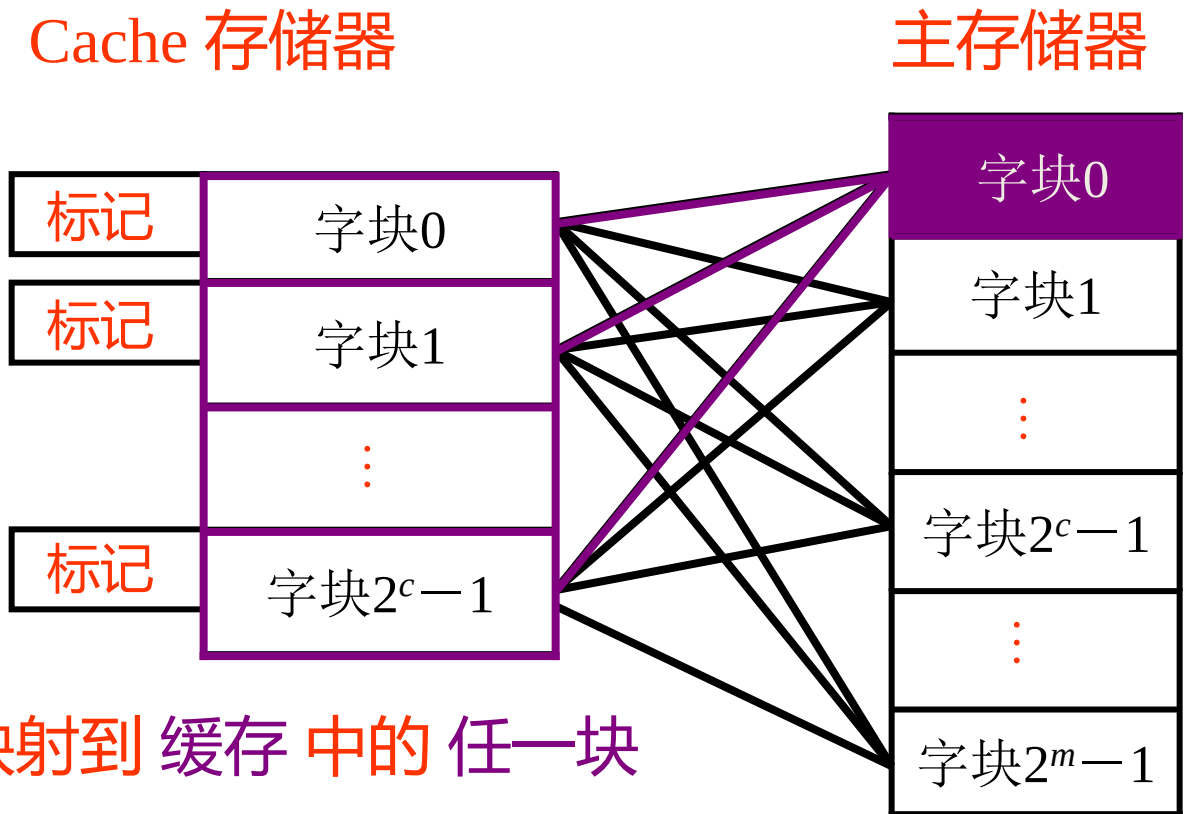


Types of Cache Misses

- **强制缺失(compulsory miss)或冷缺失(cold miss)**
  - 缓存为空时产生的缺失，是暂时事件，缓存热身(warm up)之后消失
- **容量缺失(Capacity miss)**
  - 当工作集大小超过缓存大小时产生的不命中
- **冲突缺失(Conflict miss)**
  - 缓存放置策略(placement policy)：通常将第k+1层的某个块限制放在第k层块的一个小的子集中(有时是一个块)
    - 例：第k+1层的块i必须放置在第k层的块(i mod 4)中
  - 冲突缺失(Conflict misses)指在限制性的放置策略中缓存足够 大，能够保存被引用的数据对象，但因为这些对象会映射到同一个缓存块，缓存会一直不命中
    - 例：程序请求块0, 8, 0, 8, 0, 8, ... ，在第k层的缓存中，对这两个块的每次引用都会失效

优缺点分析

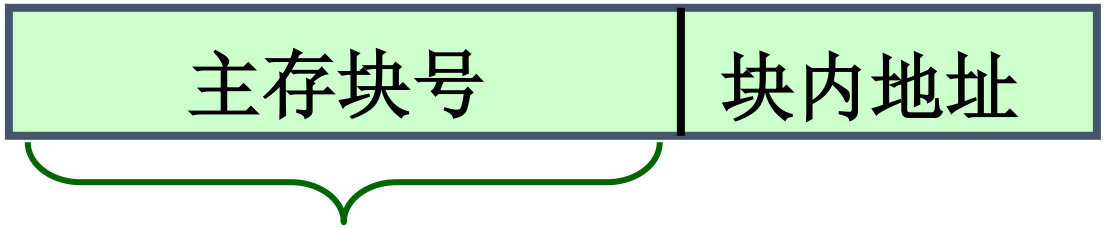
- 直接映射
  - 优点:所需硬件简单,成本低； 地址变换速度较快。
  - 缺点:块冲突概率很高； Cache利用率很低
- 全相连？
  - 随着Cache容量的增加,查表速度很难提高且电路复杂，成本高



主存 中的 任一块 可以映射到 缓存 中的 任一块

全相联地址划分

在这种方式中主存地址划分为二个部分



例如：设主存：64KB

Cache ：2KB

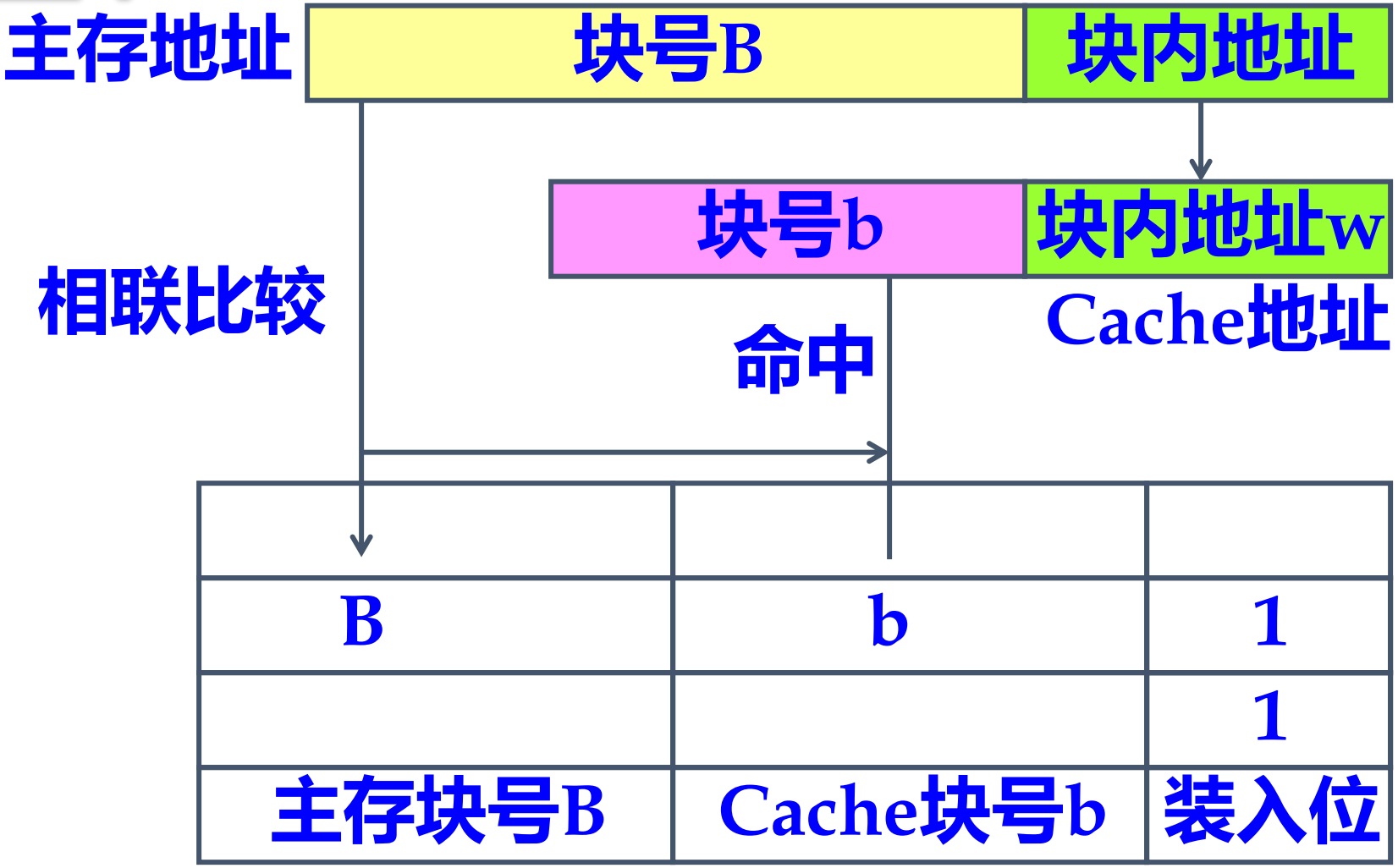
块大小：128B = 2^7B

则 主存 = $2^{16}/2^7$ = 512（块）

块号是多少位？9位，作为标记

Cache= $2^{11}/2^7$ = 16（块）

目录表结构



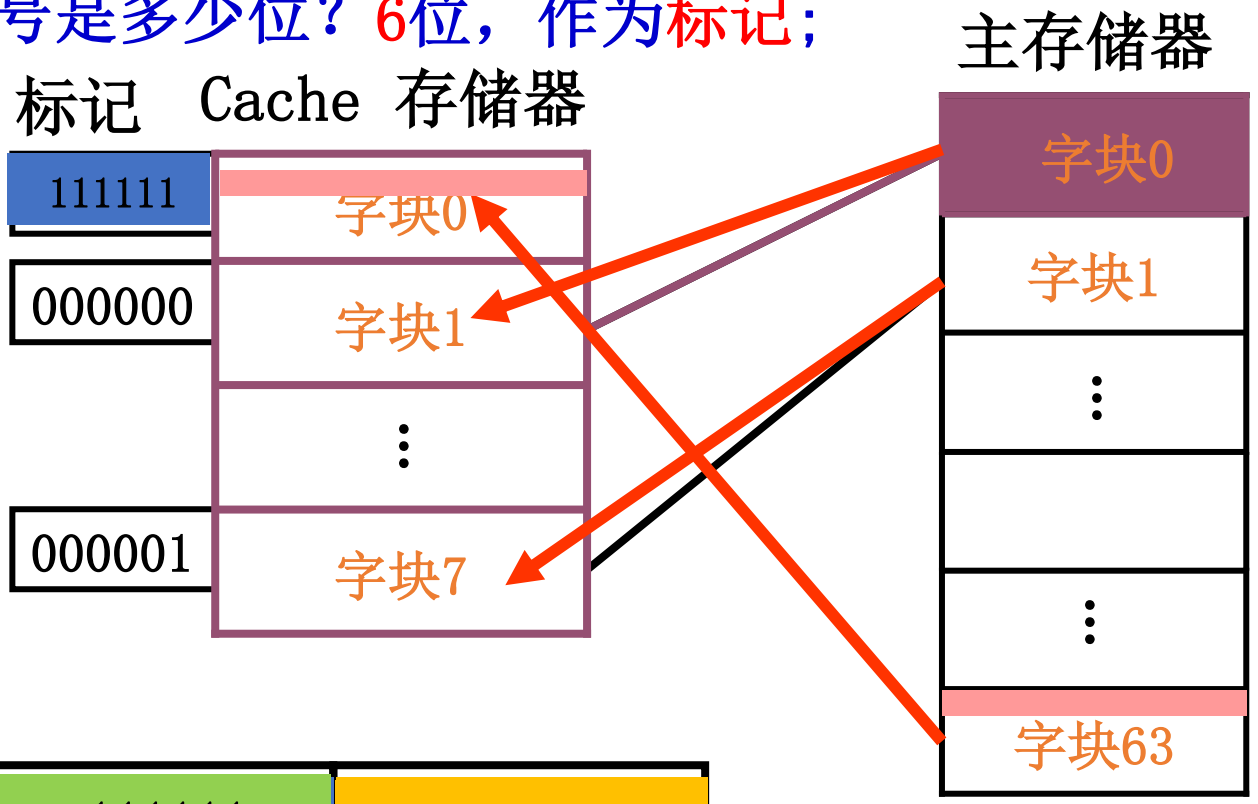
目录表，其行数与CACHE的块数相同

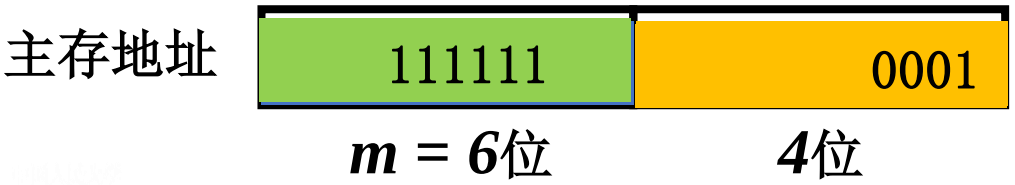
例题

例如：设 主存 1KB，Cache128B，块大小 16B。

则： 主存 = $2^{10}/2^4$ = 64（块） ； Cache= $2^7/2^4$ = 8（块）

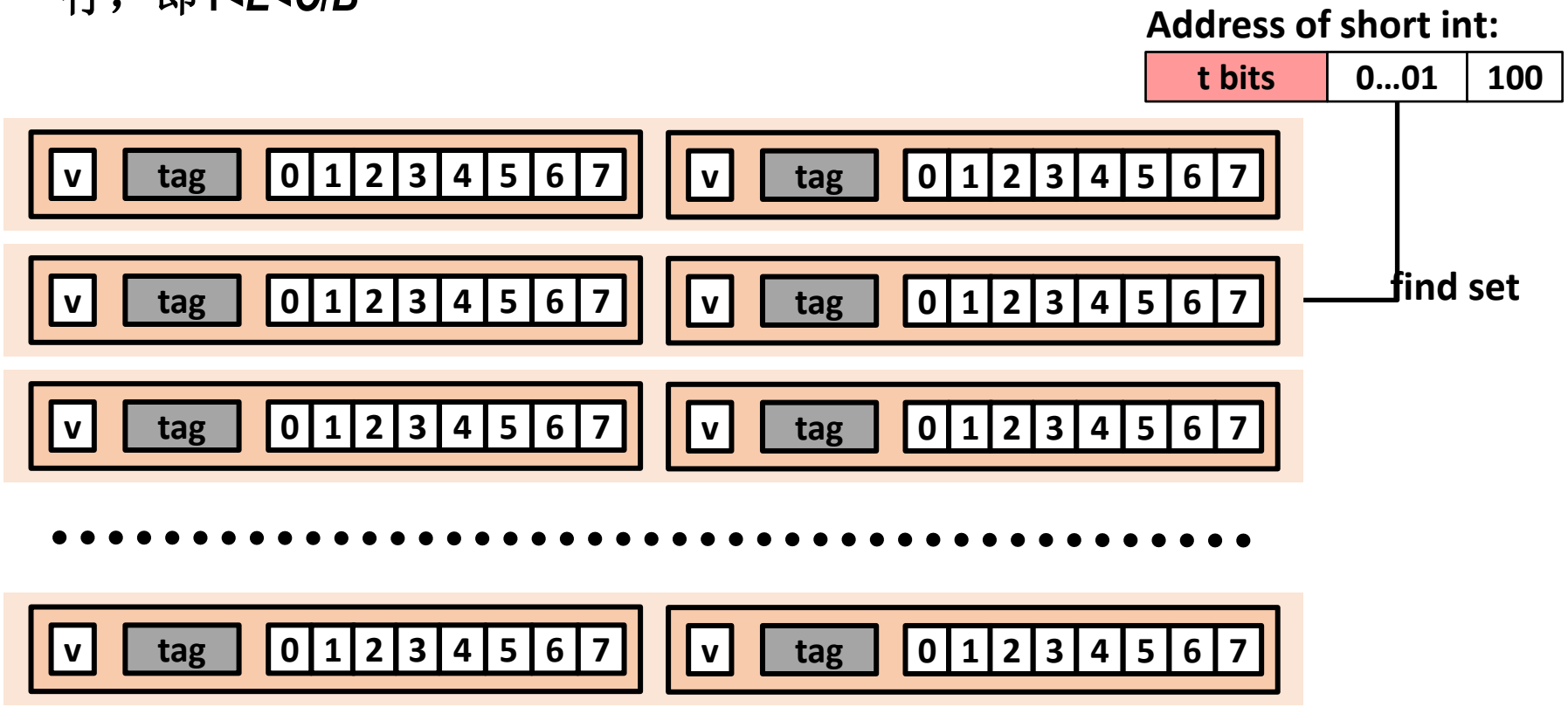
主存块号是多少位？6位，作为标记；





E路组相联Cache (E = 2)

- 直接映射cache冲突不命中的根源为每个组只有一行(*E*=1)
- 组相关高速缓存(set associative cache)每个组都保存多于一个的高速缓存行，即1<*E*<*C*/*B*



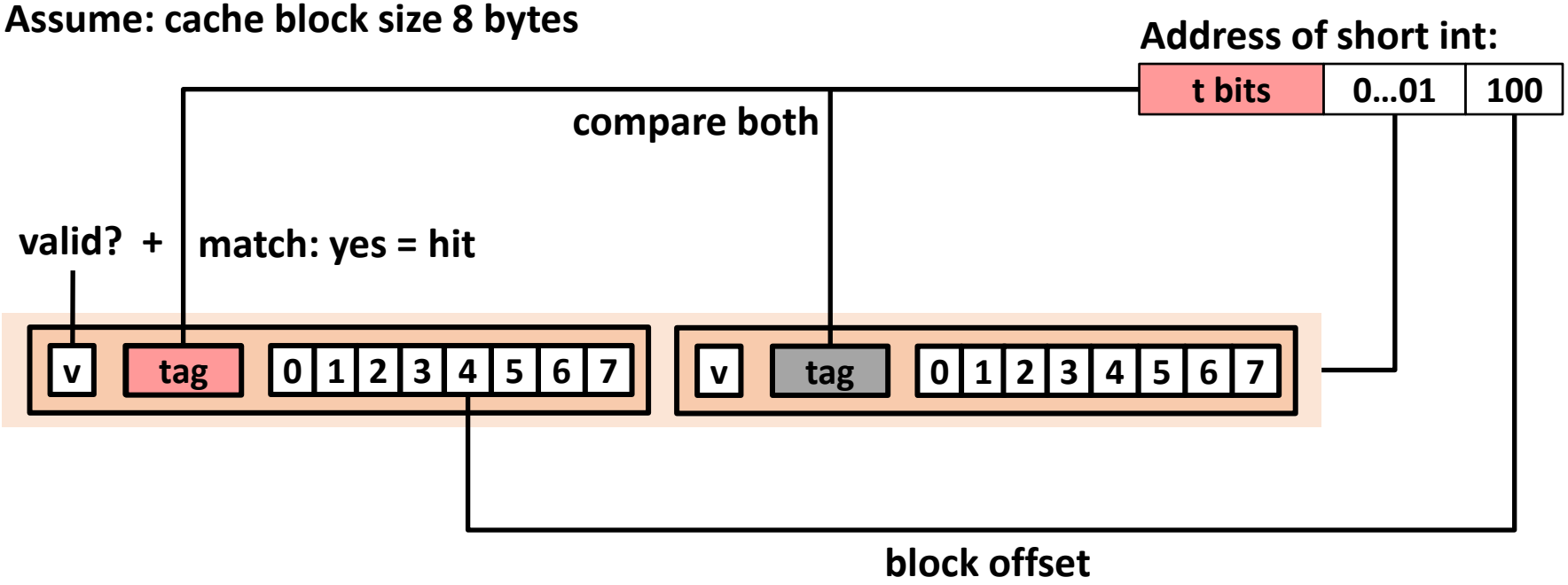
E = 2: Two lines per set

Assume: cache block size 8 bytes

E路组相联Cache (E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

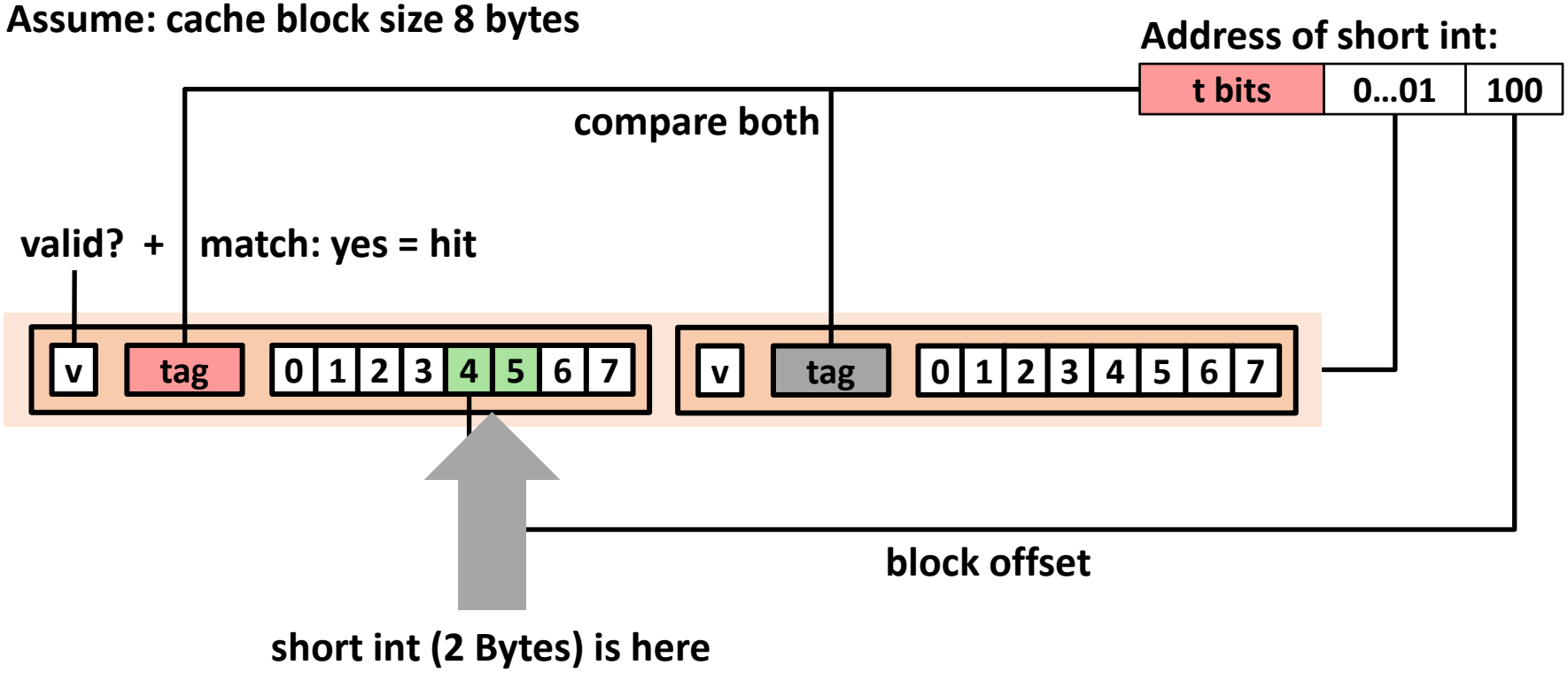


- **1.组相联高速缓存中的组选择**
  - 组索引位标识cache组
- **2.组相联高速缓存中的行匹配和字选择**
  - 检查多个行的有效位和标记位，确定所请求的字是否在集合中

E路组相联Cache (E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

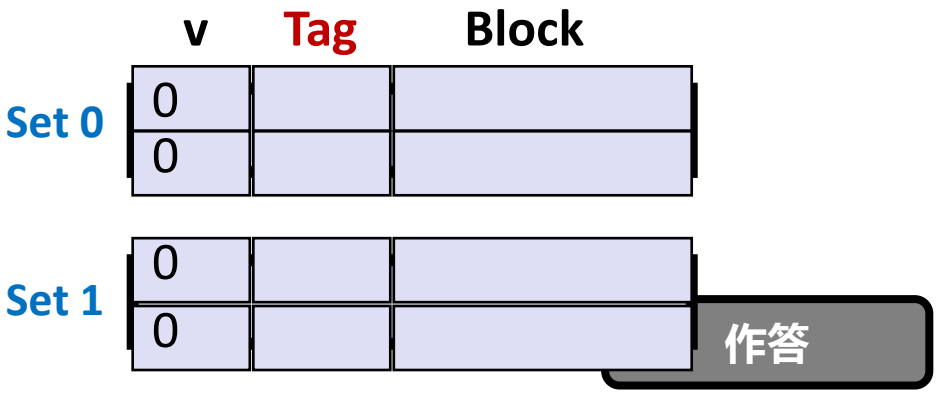
2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

4-bit addresses (M=16 bytes)
S=2 sets, E=2 blocks/set, B=2 bytes/block

Address trace (reads, one byte per read):

0	[0000_2],	[填空1]
1	[0001_2],	[填空2]
7	[0111_2],	[填空3]
8	[1000_2],	[填空4]
0	[0000_2]	[填空5]



2路组相联Cache

t=2	s=1	b=1
xx	x	x

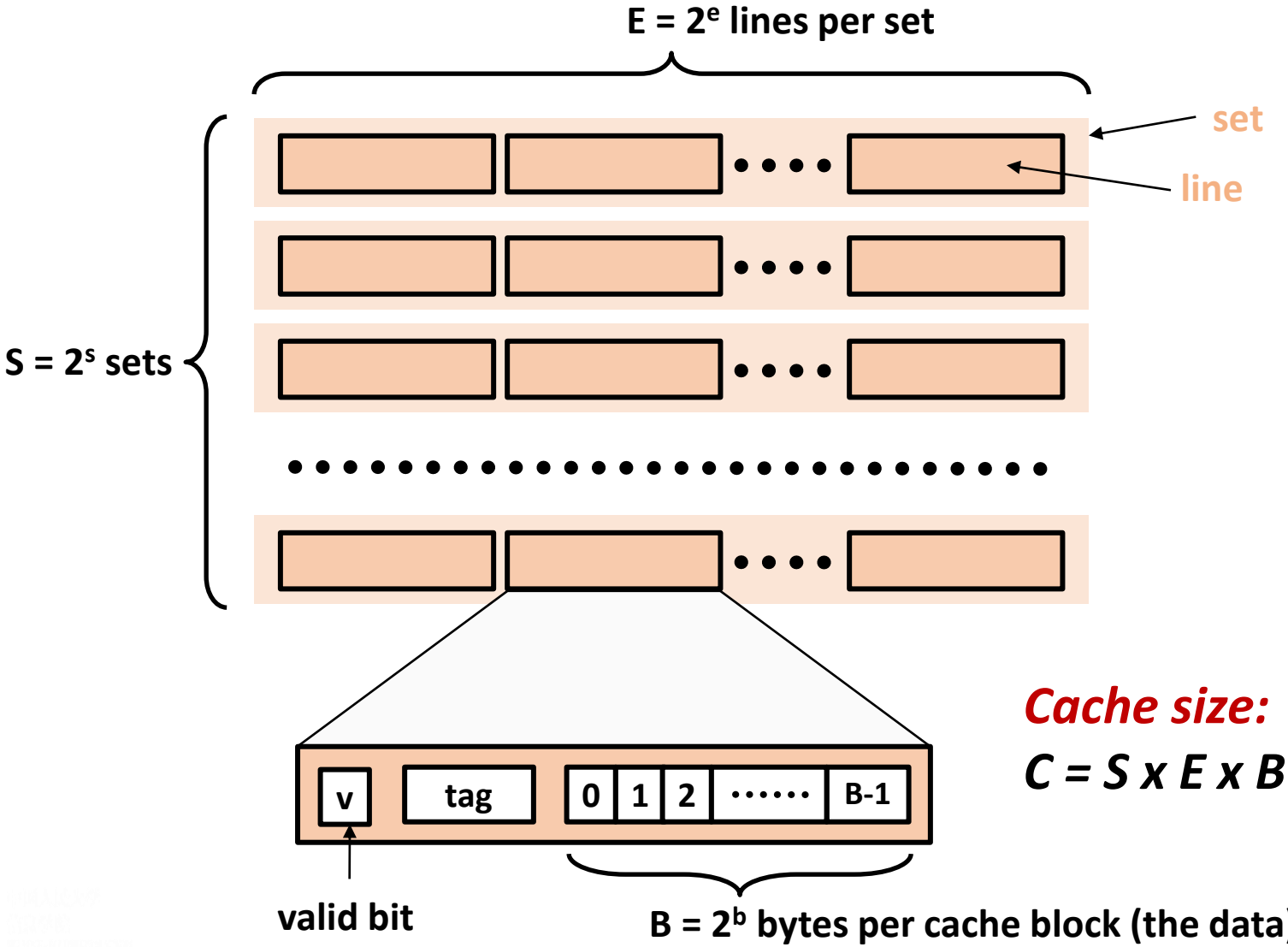
4-bit addresses (M=16 bytes)
S=2 sets, E=2 blocks/set, B=2 bytes/block

Address trace (reads, one byte per read):

0	**[0000_2],**	miss
1	**[0001_2],**	hit
7	**[0111_2],**	miss
8	**[1000_2],**	miss
0	**[0000_2]**	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

General Cache Organization (S, E, B)

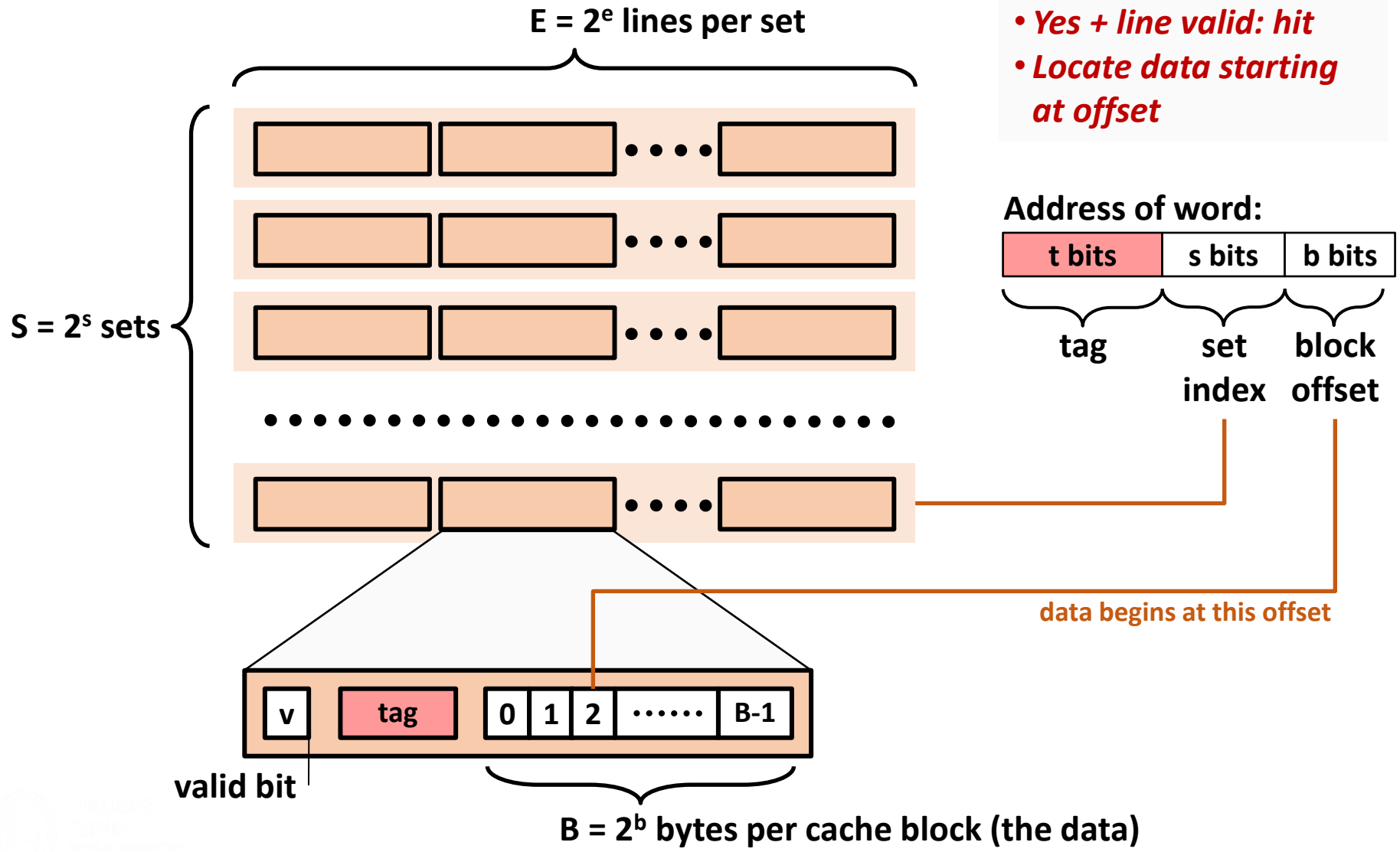


Cache size:

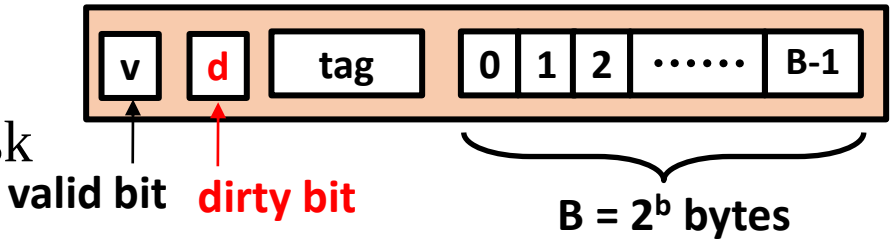
C = S x E x B data bytes

Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

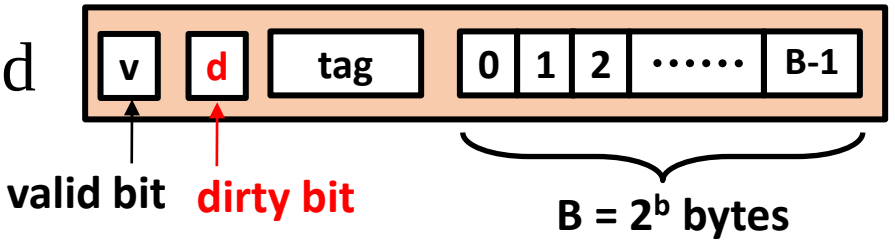


What about writes?



- Multiple copies of data exist:
  - L1, L2, L3, Main Memory, Disk
- What to do on a write-hit?
  - Write-through 写直达(write immediately to memory)
  - Write-back 写回(defer write to memory until replacement of line)
    - Each cache line needs a dirty bit (set if data has been written to)
- What to do on a write-miss?
  - Write-allocate写分配 (load into cache, update line in cache)
    - Good if more writes to the location will follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- Typical
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate（写回+写分配）**

Practical Write-back Write-allocate



- A write to address X is issued
- If it is a hit
  - Update the contents of block
  - Set dirty bit to 1 (bit is sticky and only cleared on eviction)
- If it is a miss
  - Fetch block from memory (per a read miss)
  - The perform the write operations (per a write hit)
- If a line is evicted and dirty bit is set to 1
  - The entire block of 2^b bytes are written back to memory
  - Dirty bit is cleared (set to 0)
  - Line is replaced by new contents

Why Index Using Middle Bits?

Direct mapped: One line per set
Assume: cache block size 8 bytes

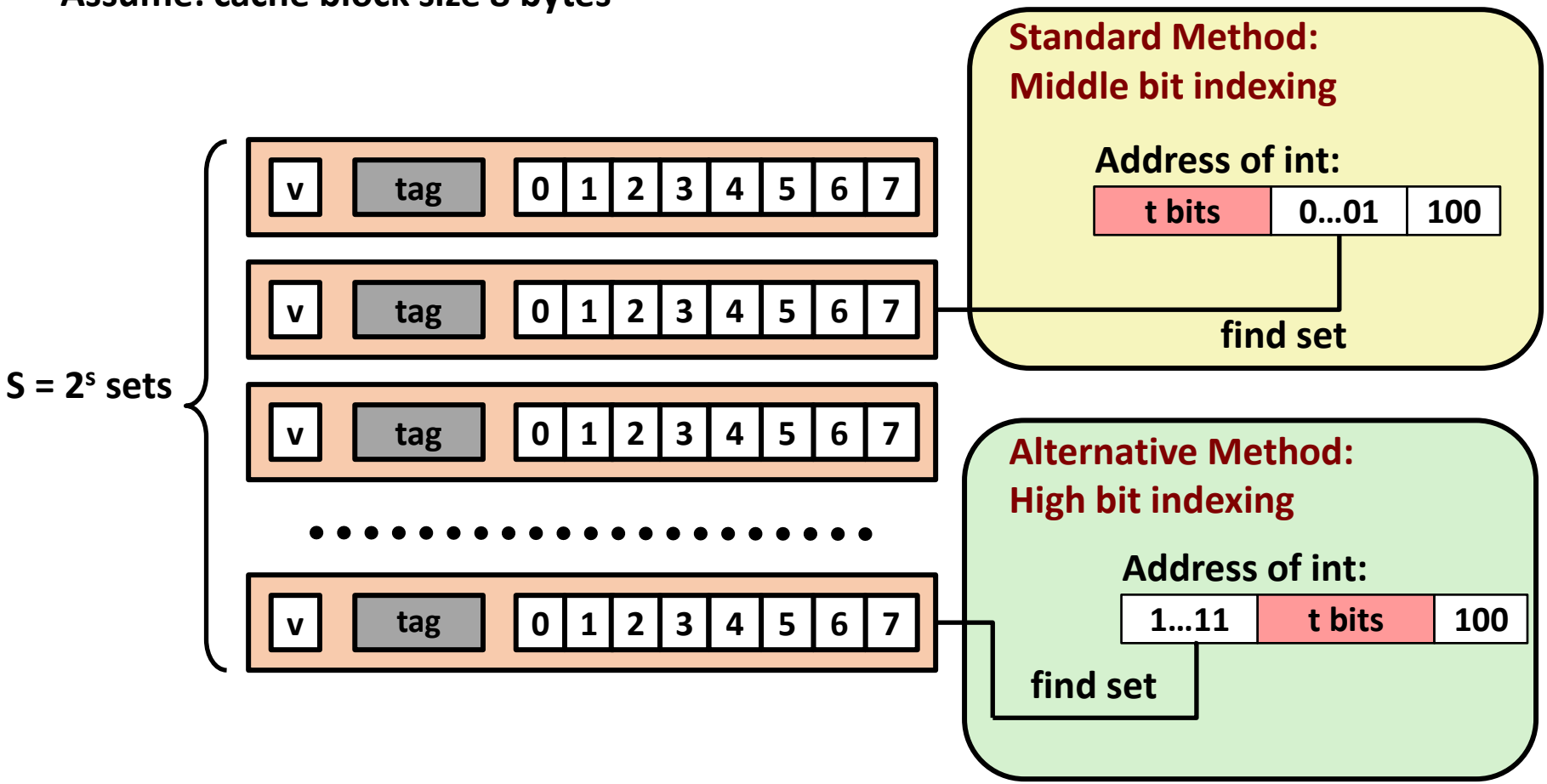
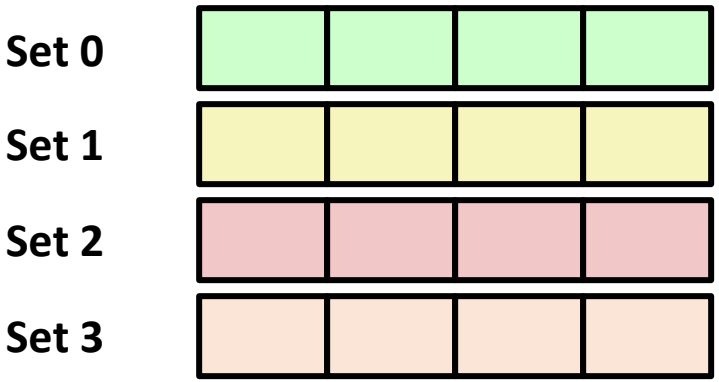
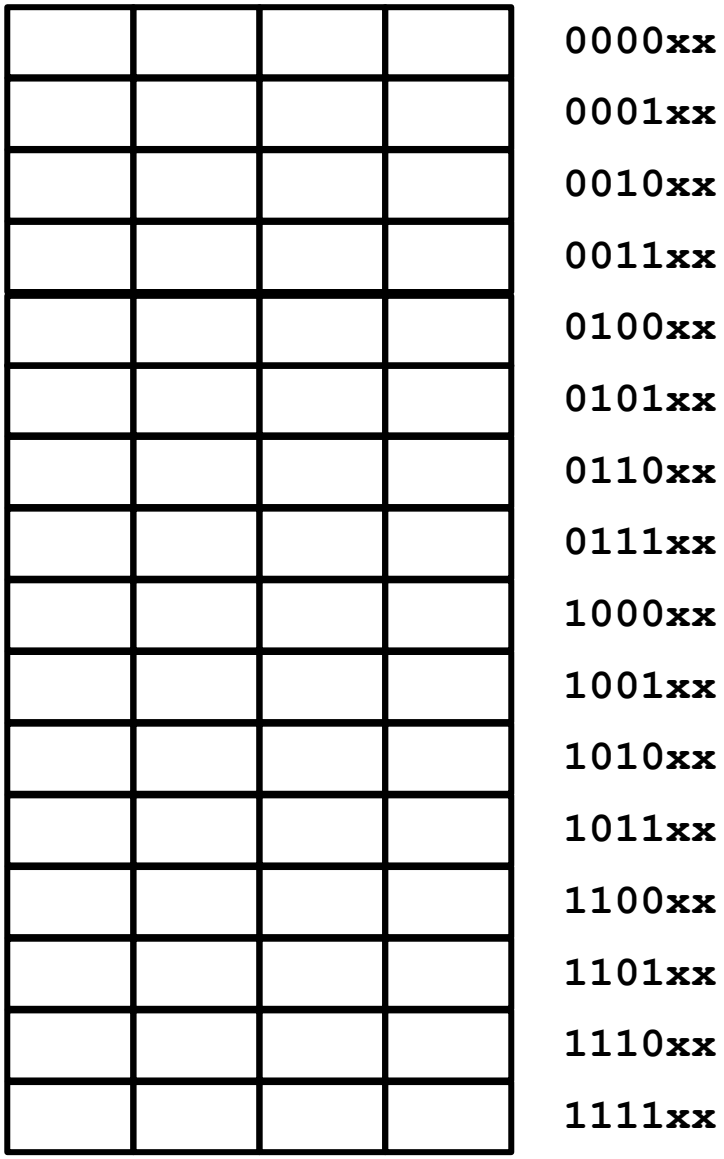


Illustration of Indexing Approaches

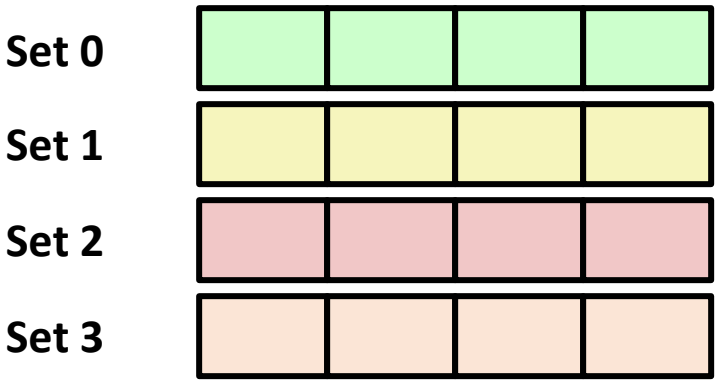
- 64-byte memory
  - 6-bit addresses
- 16 byte, direct-mapped cache
- Block size = 4. (Thus, 4 sets; why?)
- 2 bits tag, 2 bits index, 2 bits offset

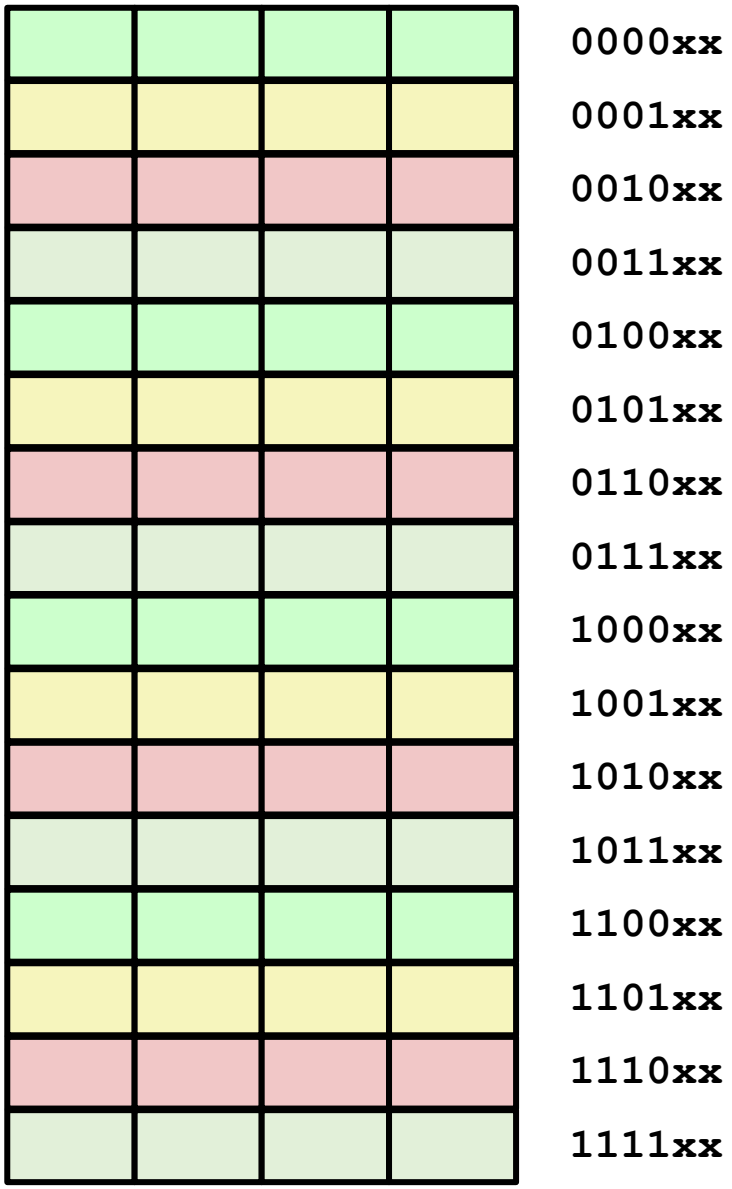




Middle Bit Indexing

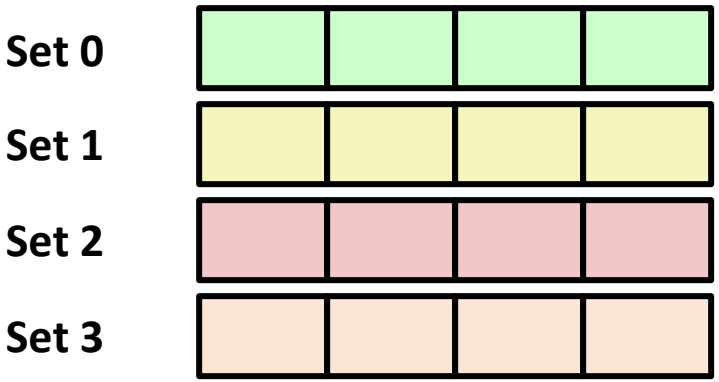
- Addresses of form TTSSBB
  - **TT** Tag bits
  - **SS** Set index bits
  - **BB** Offset bits
- Makes good use of spatial locality

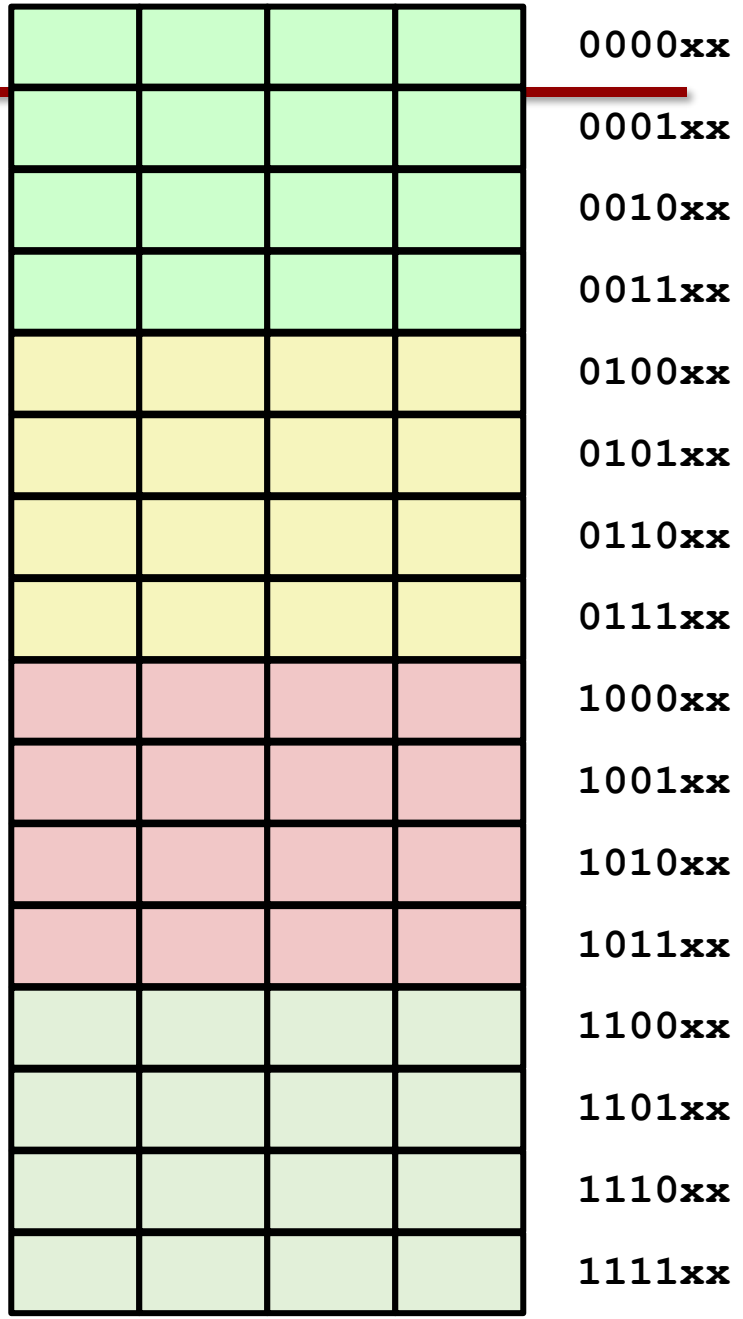




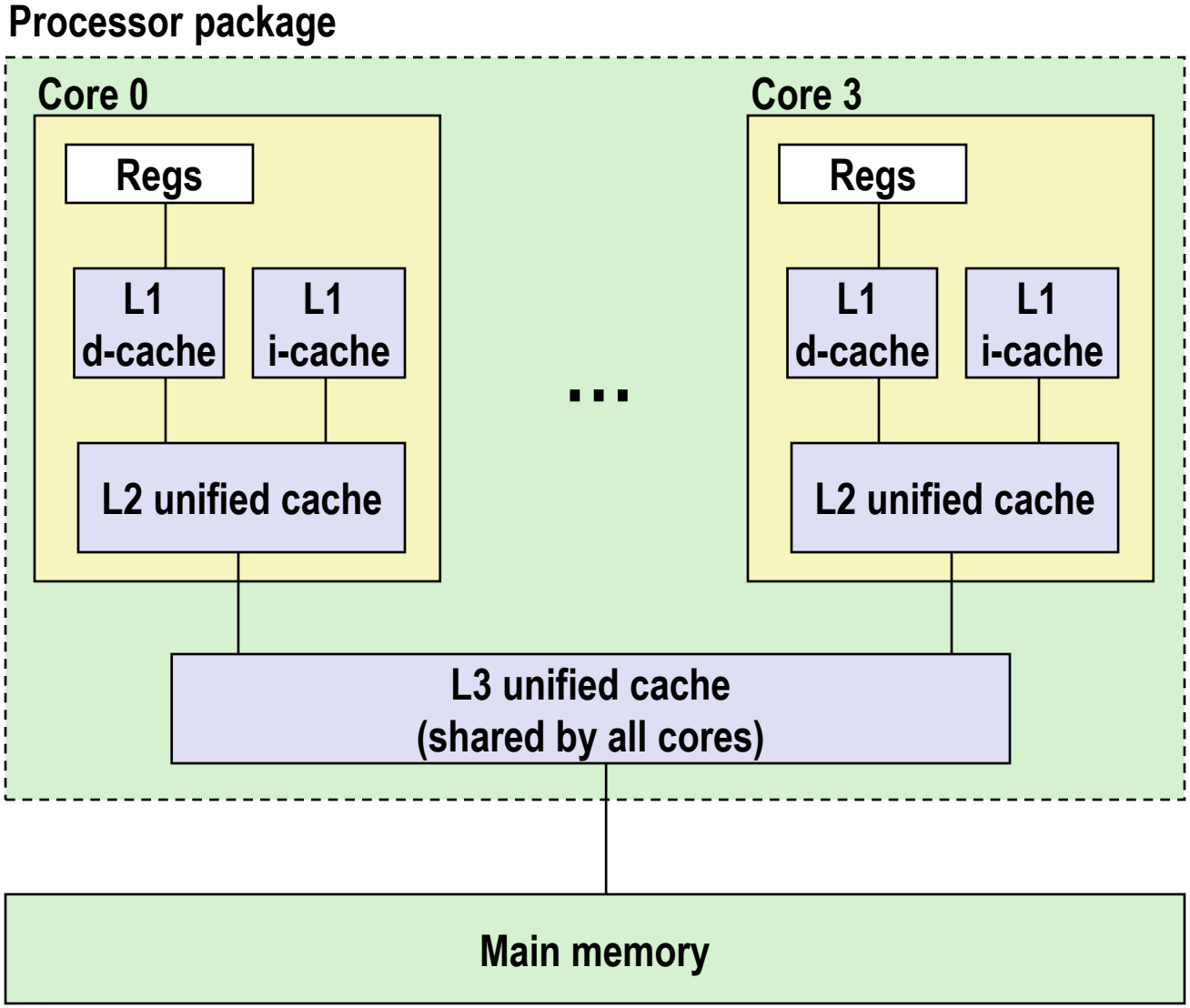
High Bit Indexing

- Addresses of form SSTTBB
  - **SS** Set index bits
  - **TT** Tag bits
  - **BB** Offset bits
- Program with high spatial locality would generate lots of conflicts





Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 kB, 8-way,
Access: 4 cycles

L2 unified cache:
256 kB, 8-way,
Access: 10 cycles

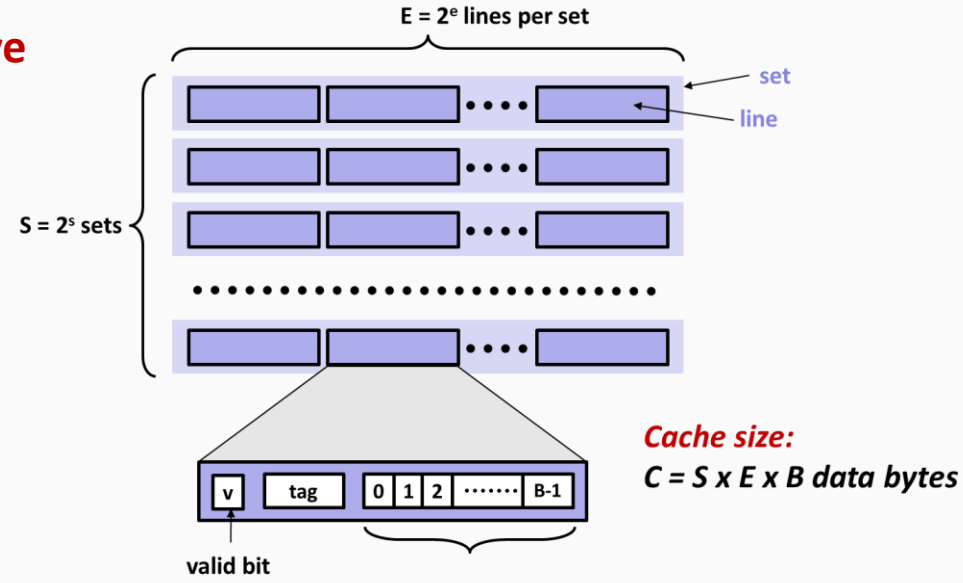
L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

Example: Core i7 L1 Data Cache

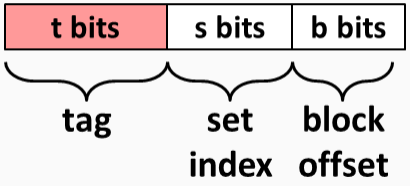
32 kB 8-way set associative
64 bytes/block
47 bit address range

B = , b =
S = , s =
E = , e =
C =



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



Block offset: . bits
Set index: . bits
Tag: . bits

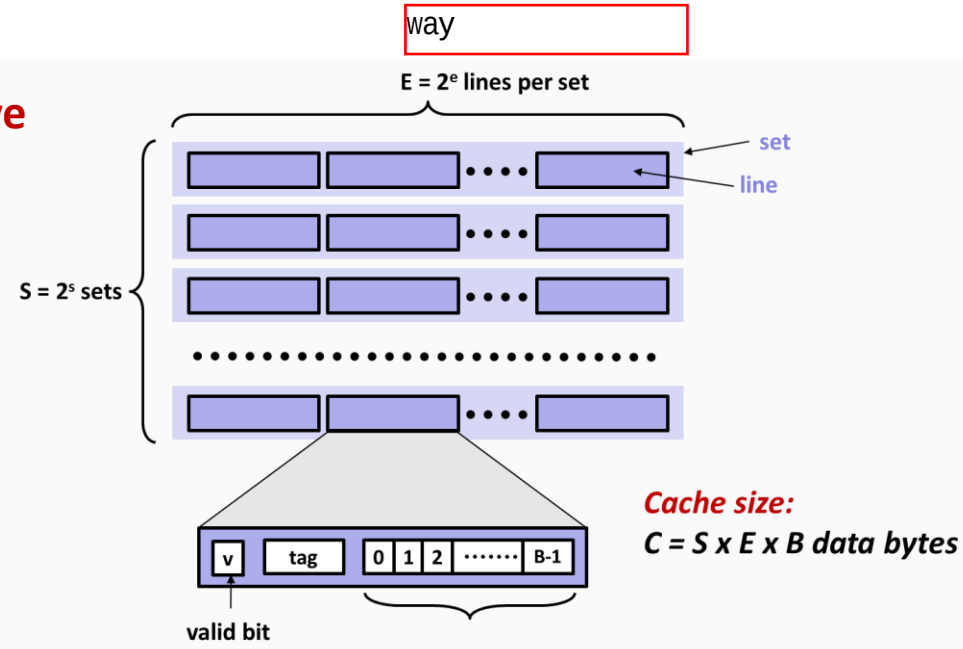
Stack Address:
0x00007f7262a1e010

Block offset: 0x??
Set index: 0x??
Tag: 0x??

Example: Core i7 L1 Data Cache

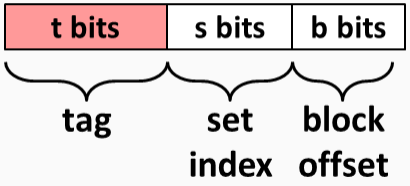
32 kB 8-way set associative
64 bytes/block
47 bit address range

B = 64, b=6
S = 64, s = 6
E = 8, e = 3
C = 64 x 64 x 8 = 32,768



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



Block offset: 6 bits
Set index: 6 bits
Tag: 35 bits

Stack Address:
0x00007f7262a1e010

0000 0001 0000

Block offset: 0x10
Set index: 0x0
Tag: 0x7f7262a1e

Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses)
    = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- Hit Time
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 4 clock cycle for L1
    - 10 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Average access time:
    97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

- This is why “miss rate” is used instead of “hit rate”

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions
- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)重复引用相同变量的程序有良好的时间局部性
  - 频繁访问数据集的缓存级别：寄存器>L1 cache>L2 cache-L3 cache>内存
  - Stride-1 reference patterns are good (spatial locality)步长为k的引用模式程序，步长越小，空间局部性越好
  - CPU内存访问单位cache line(64字节)，步长越小，效率越高
  - 对于取指令而言，循环有好的时间和空间局部性，循环越小，循环迭代次数越多，局部性越好,简化循环体内代码结构

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

填空题 1分

设置

空间局部性排序

```
#define N 1024
#define M 256
typedef struct{
        int vel[M];
        int acc[M];
}point;
point p[N];
```

```
void clear1(point *p, int n){
        int i,j;
        for (i=0;i<n;i++){
                for (j=0;j<M;j++)
                        p[i].vel[j]=0;
                for (j=0;j<M;j++)
                        p[i].acc[j]=0;
        }
        return;}
```

```
void clear2(point *p, int n){
        int i,j;
        for (i=0;i<n;i++){
                for (j=0;j<M;j++){
                        p[i].vel[j]=0;
                        p[i].acc[j]=0;
                }       }
        return;}
```

```
void clear3(point *p, int n){
        int i,j;
        for (j=0;j<M;j++){
                for (i=0;i<n;i++)
                        p[i].vel[j]=0;
                for (i=0;i<n;i++)
                        p[i].acc[j]=0;
        }return;}
```

[填空1]

练习题:分析三个函数的空间局部性

```
#define N 1024
#define M 256
typedef struct{
        int vel[M];
        int acc[M];
}point;
point p[N];
```

```
void clear1(point *p, int n){
        int i,j;
        for (i=0;i<n;i++){
                for (j=0;j<M;j++)
                        p[i].vel[j]=0;
                for (j=0;j<M;j++)
                        p[i].acc[j]=0;
        }
        return;}
```

```
void clear2(point *p, int n){
        int i,j;
        for (i=0;i<n;i++){
                for (j=0;j<M;j++){
                        p[i].vel[j]=0;
                        p[i].acc[j]=0;
                }       }
        return;}
```

```
void clear3(point *p, int n){
        int i,j;
        for (j=0;j<M;j++){
                for (i=0;i<n;i++)
                        p[i].vel[j]=0;
                for (i=0;i<n;i++)
                        p[i].acc[j]=0;
        }return;}
```

Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

The Memory Mountain

- Read throughput (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

- Memory mountain: Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.

Memory Mountain Test Function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *      array “data” with stride of "stride", using
 *      using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

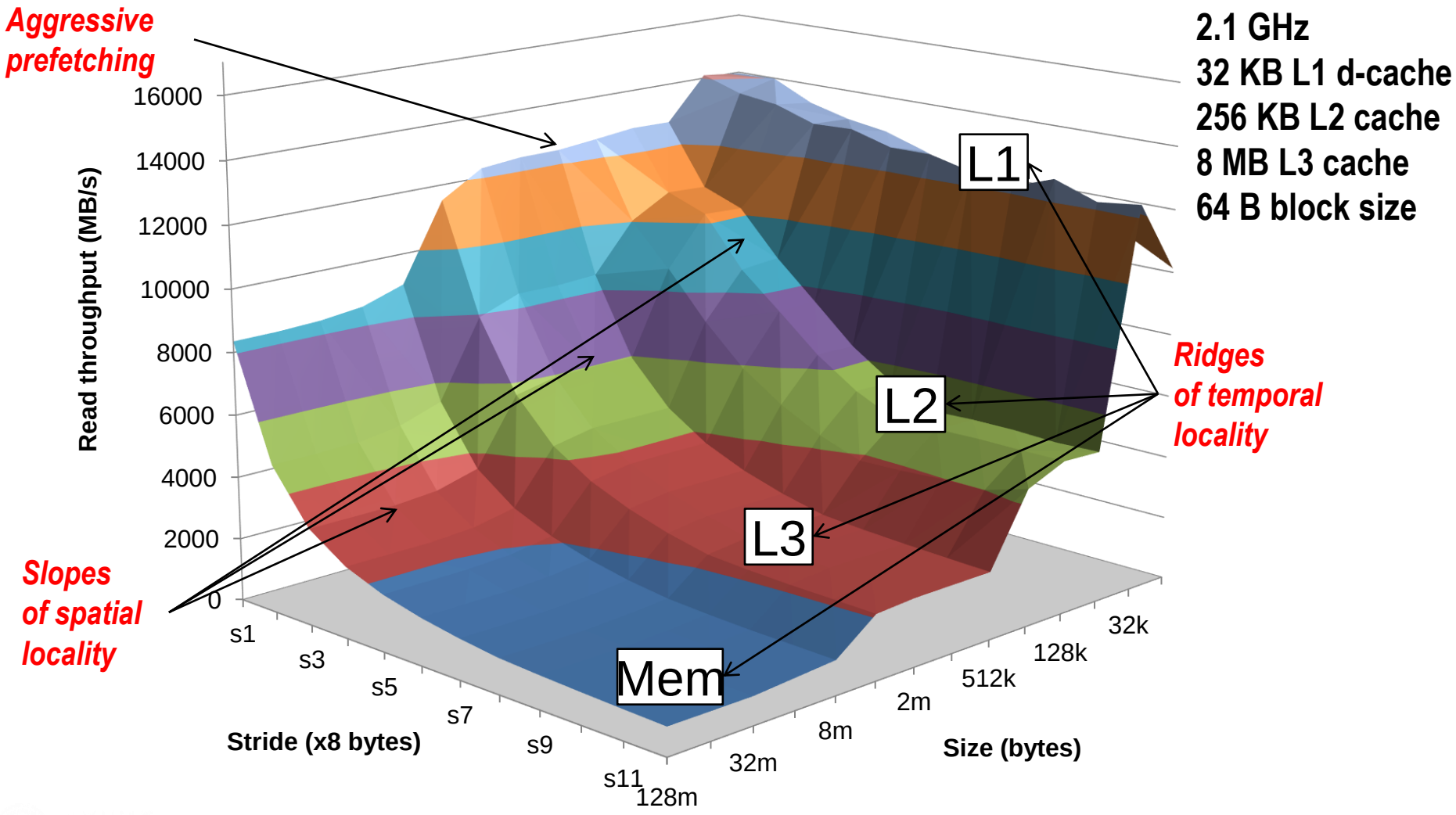
Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

2. Call `test()` again and measure the read throughput(MB/s)

The Memory Mountain



Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Matrix Multiplication Example

- Description:
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

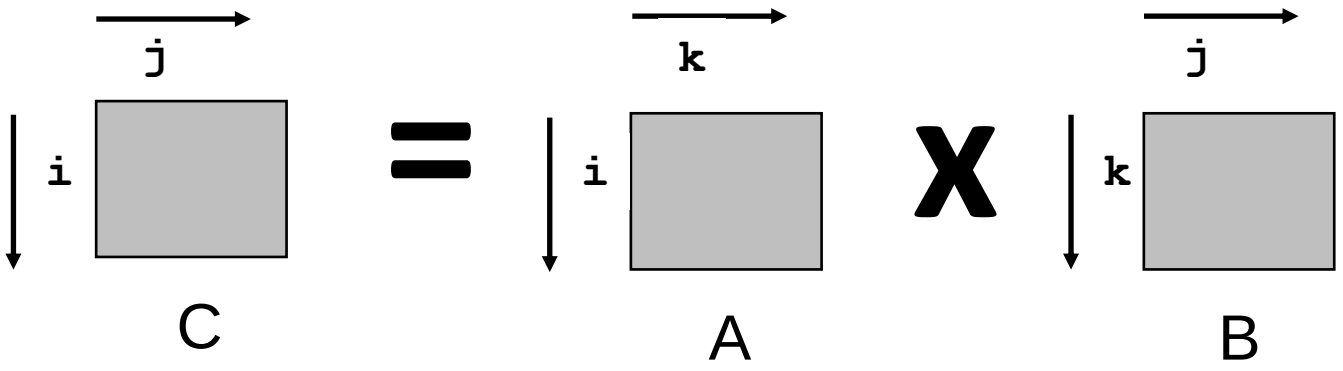
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable sum held in register

matmult/mm.c

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop



Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`

    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`

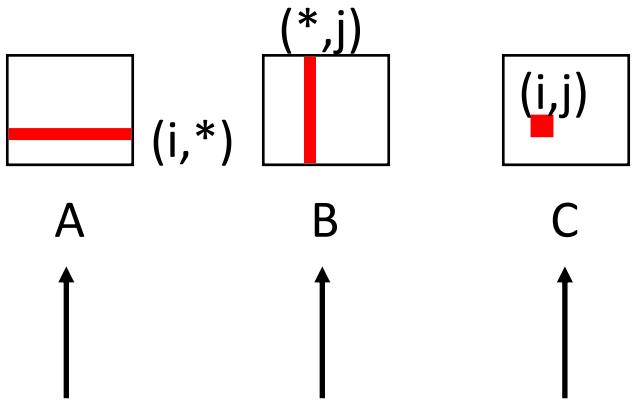
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

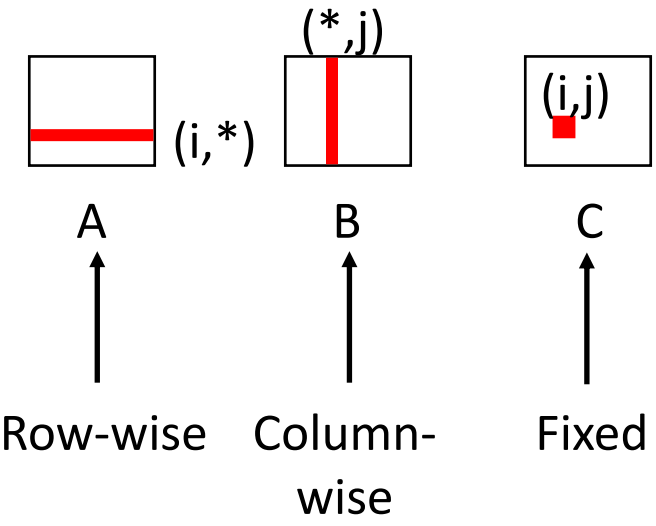
A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

matmult/mm.c

Inner loop:



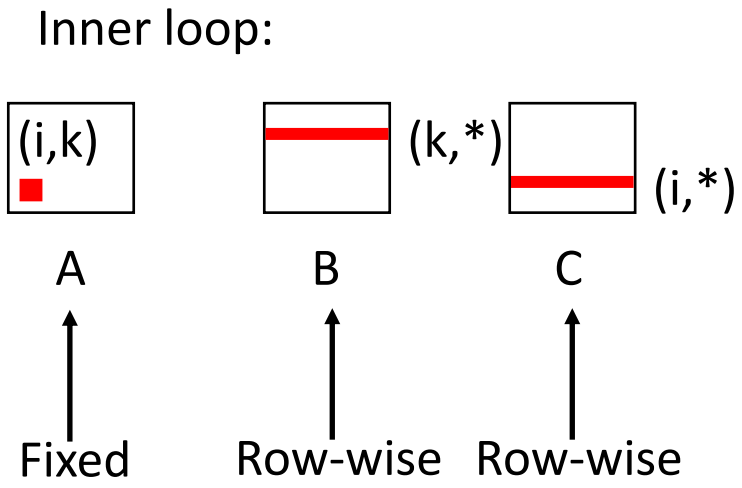
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c



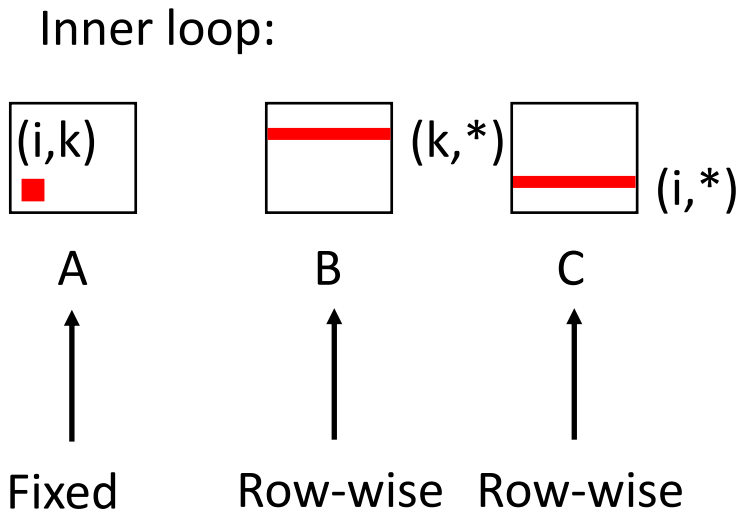
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c



Misses per inner loop iteration:

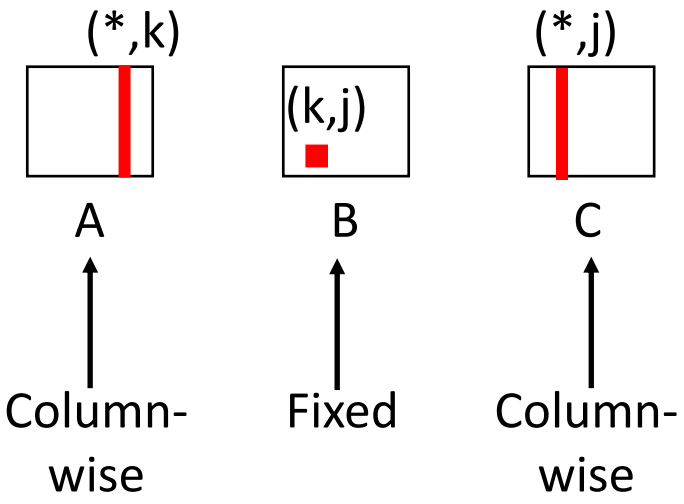
A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

Inner loop:



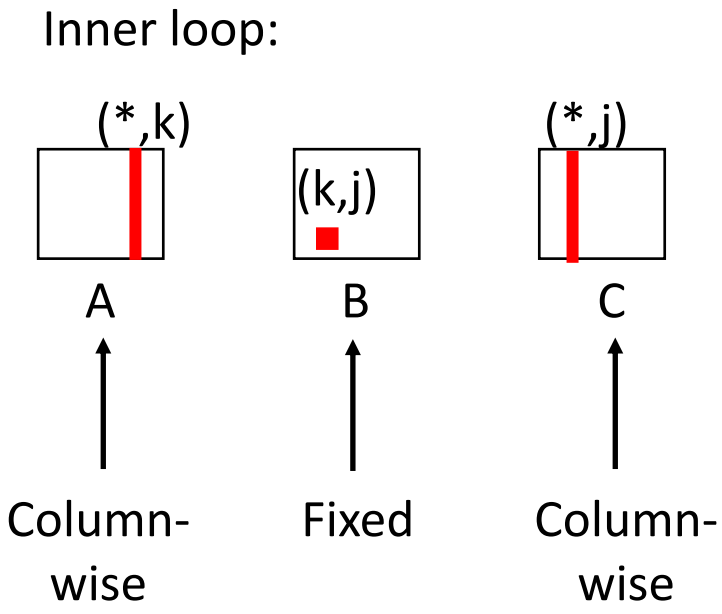
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

kij (& ikj):

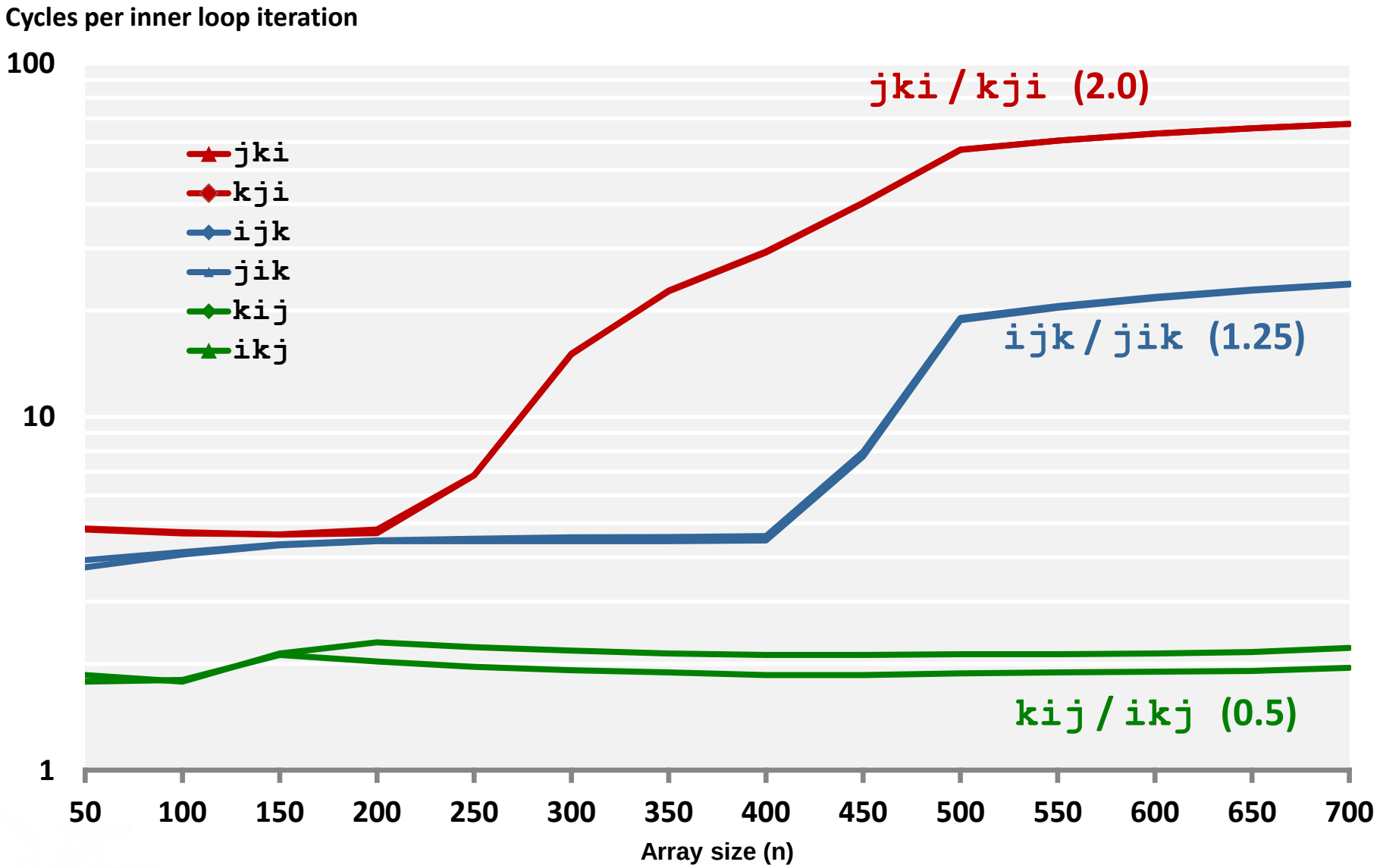
- 2 loads, 1 store
- misses/iter = **0.5**

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

jki (& kji):

- 2 loads, 1 store
- misses/iter = **2.0**

Core i7 Matrix Multiply Performance



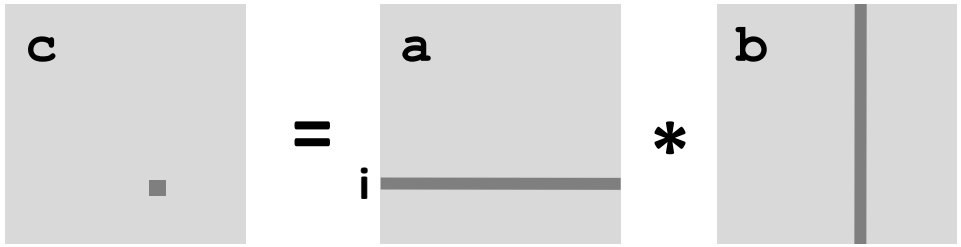
Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

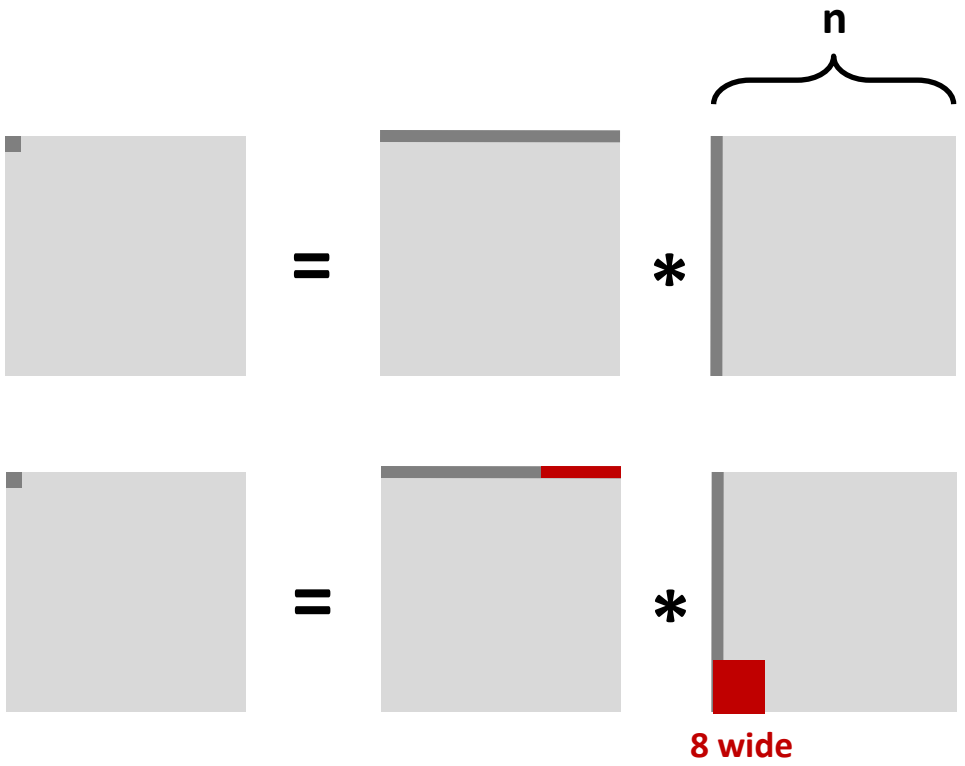


Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- First iteration:
  - n/8 + n = 9n/8 misses

  - Afterwards in cache: (schematic)

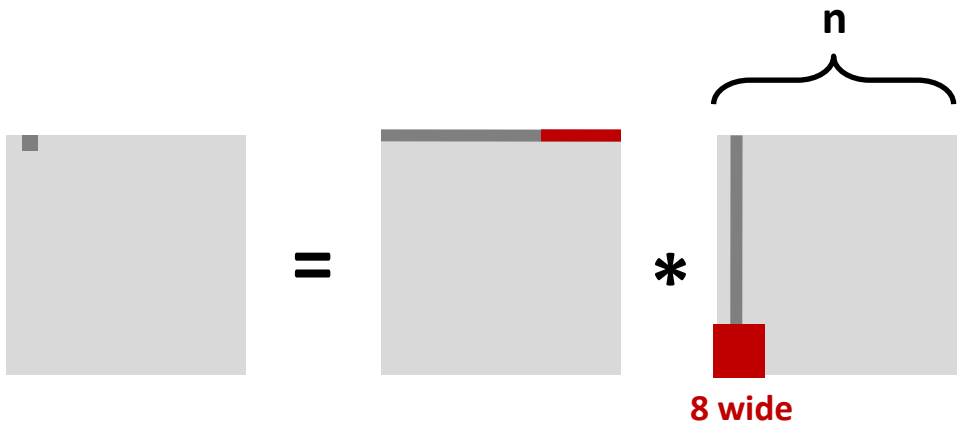


Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- Second iteration:
  - Again:
    n/8 + n = 9n/8 misses

- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$

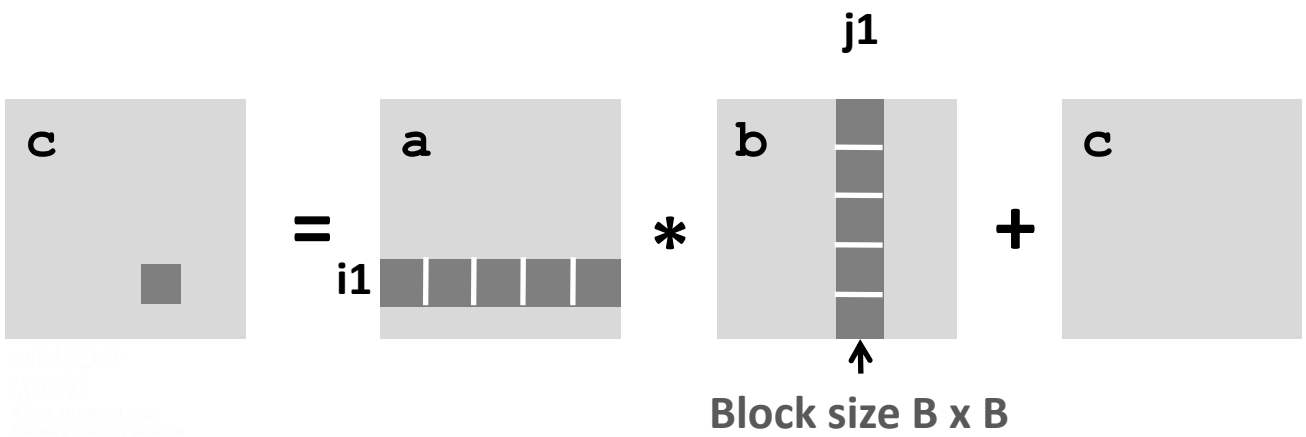


Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c

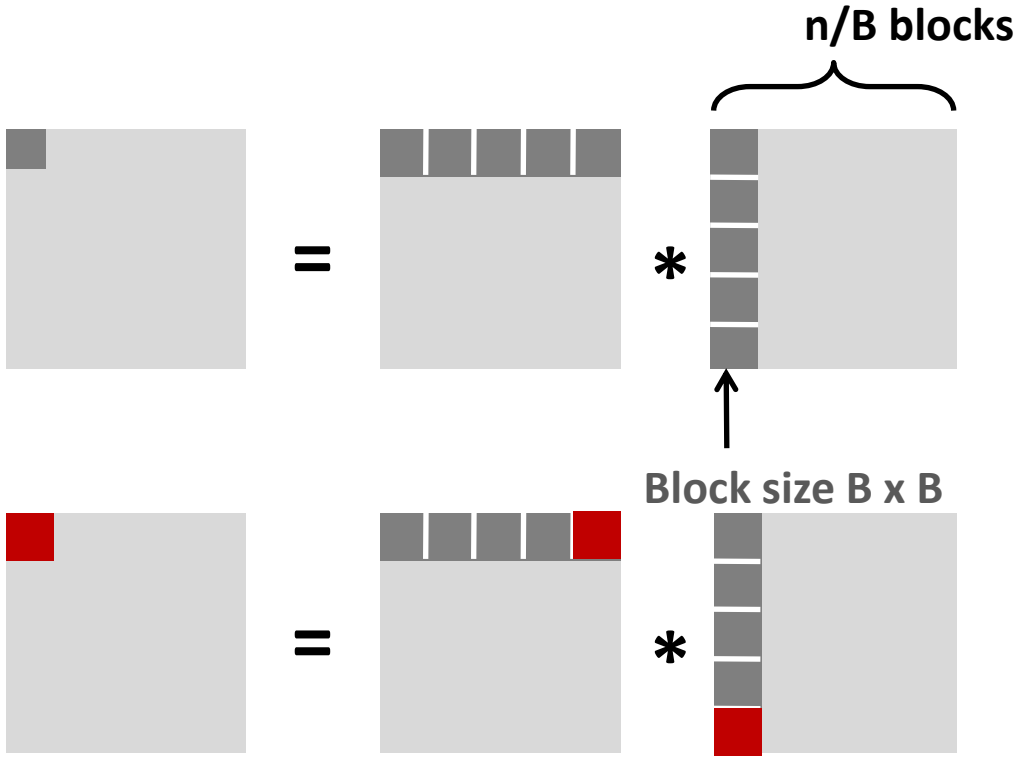


Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- First (block) iteration.
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$
    (omitting matrix c)

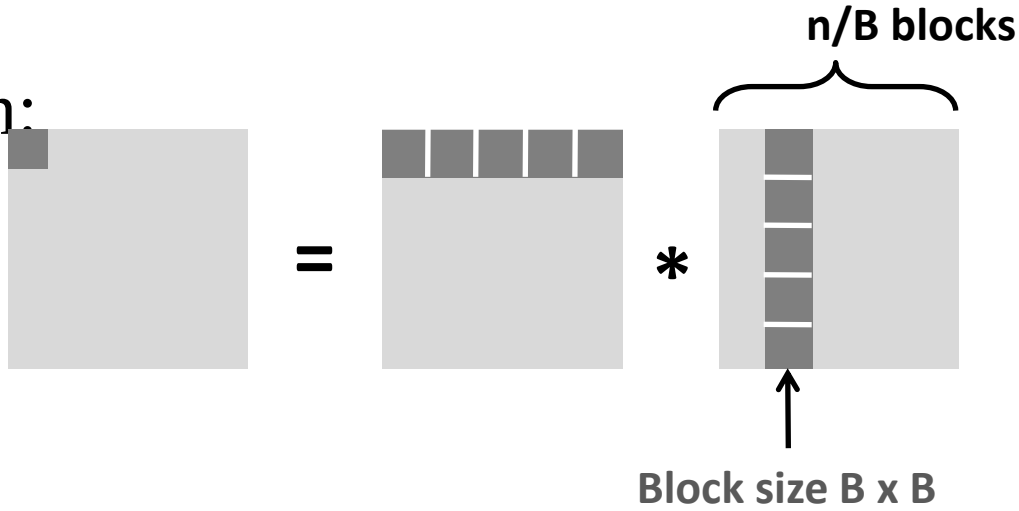
  - Afterwards in cache
    (schematic)



Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- Second (block) iteration:
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$



- Total misses:
  - $nB/4 * (n/B)^2 = n^3/(4B)$

Blocking Summary

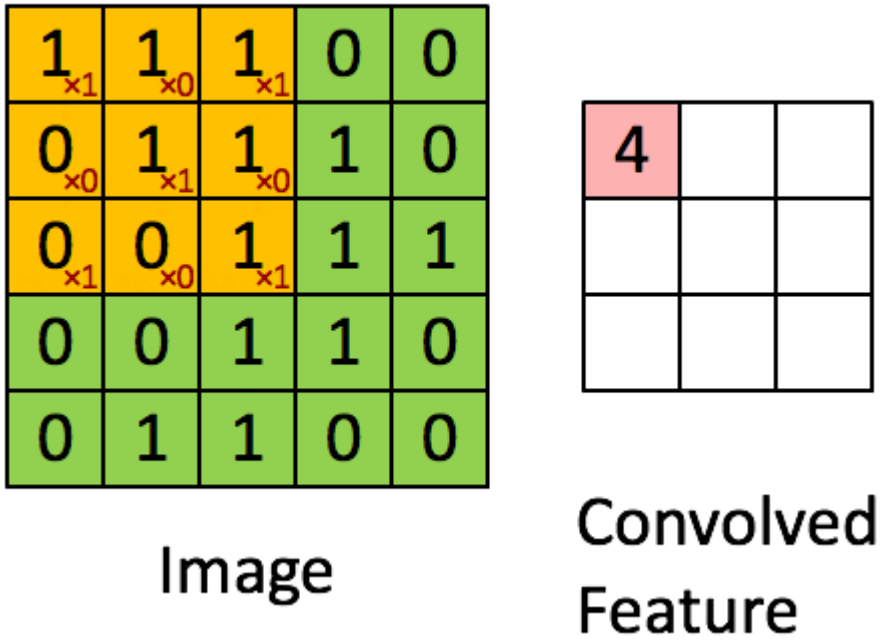
- No blocking: (9/8) * n^3
- Blocking: 1/(4B) * n^3

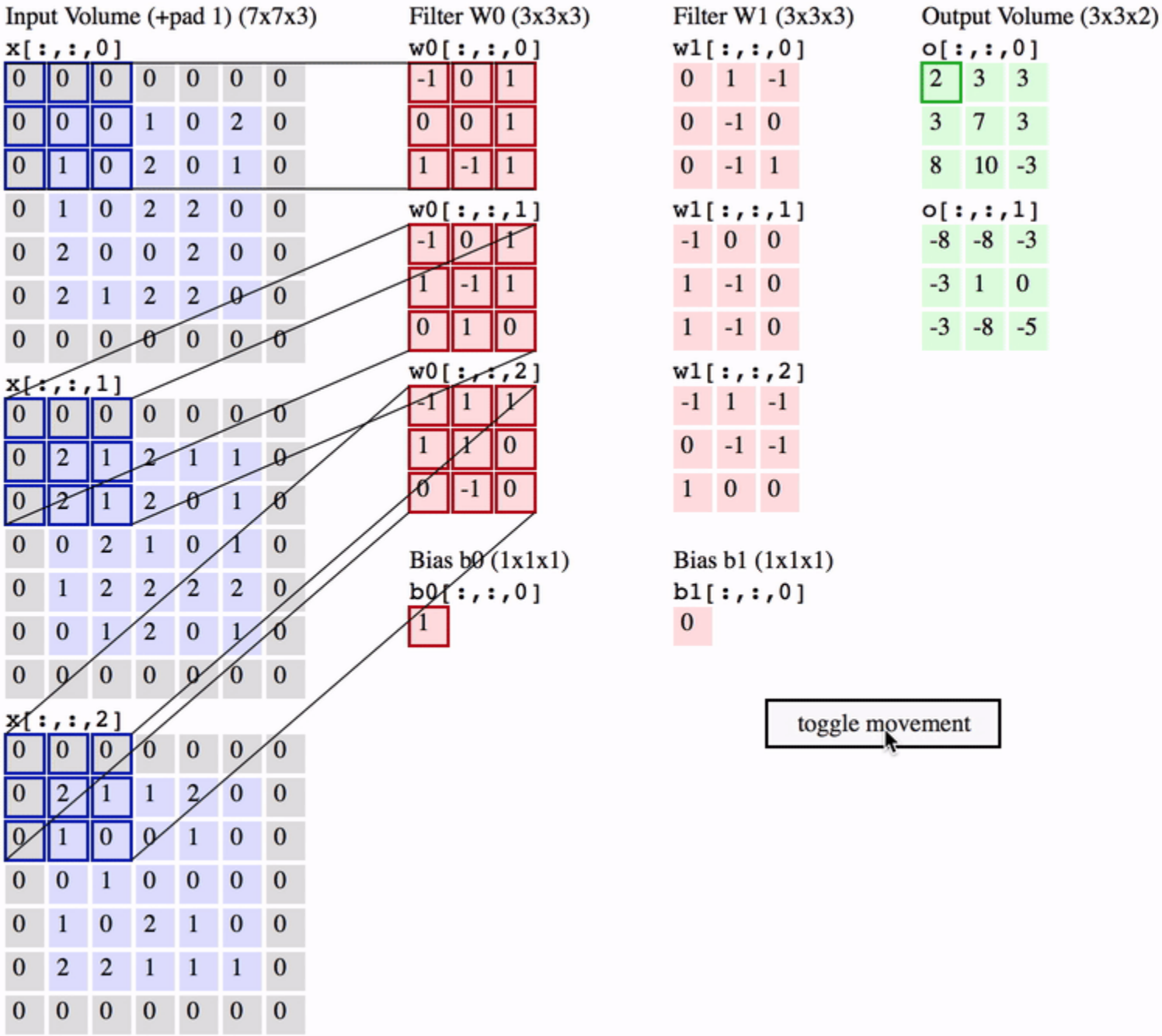
- Suggest largest possible block size B, but limit $3B^2$ < C!

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:

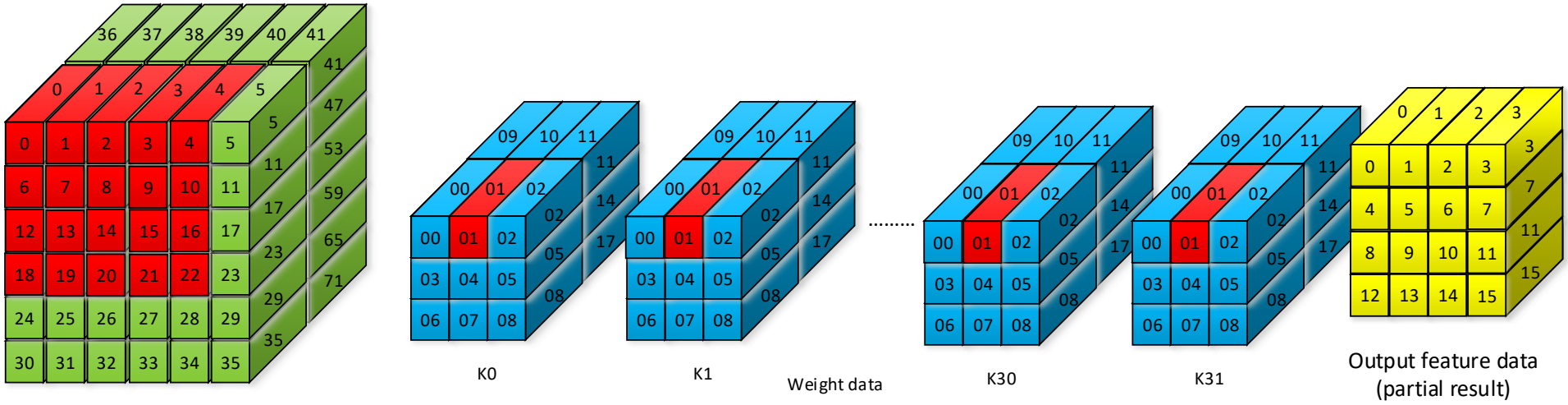
  - But program has to be written properly

卷积神经网络


1 1 1 0 0
0 1 1 1 0
0 0 1 1 1
0 0 1 1 0
0 1 1 0 0
×1 ×0 ×1
×0 ×1 ×0
×1 ×0 ×1
4
Image
Convolved Feature


Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 0 0 1 0 2 0
0 1 0 2 0 1 0
0 1 0 2 2 0 0
0 2 0 0 2 0 0
0 2 1 2 2 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 2 1 2 1 1 0
0 2 1 2 0 1 0
0 0 2 1 0 1 0
0 1 2 2 2 2 0
0 0 1 2 0 1 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 2 1 1 2 0 0
0 1 0 0 1 0 0
0 0 1 0 0 0 0
0 1 0 2 1 0 0
0 2 2 1 1 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
-1 0 1
0 0 1
1 -1 1
w0[:,:,1]
-1 0 1
1 -1 1
0 1 0
w0[:,:,2]
-1 1 1
1 1 0
0 -1 0
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
0 1 -1
0 -1 0
0 -1 1
w1[:,:,1]
-1 0 0
1 -1 0
1 -1 0
w1[:,:,2]
-1 1 -1
0 -1 -1
1 0 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
2 3 3
3 7 3
8 10 -3
o[:,:,1]
-8 -8 -3
-3 1 0
-3 -8 -5
toggle movement

英伟达开源AI加速器NVDLA

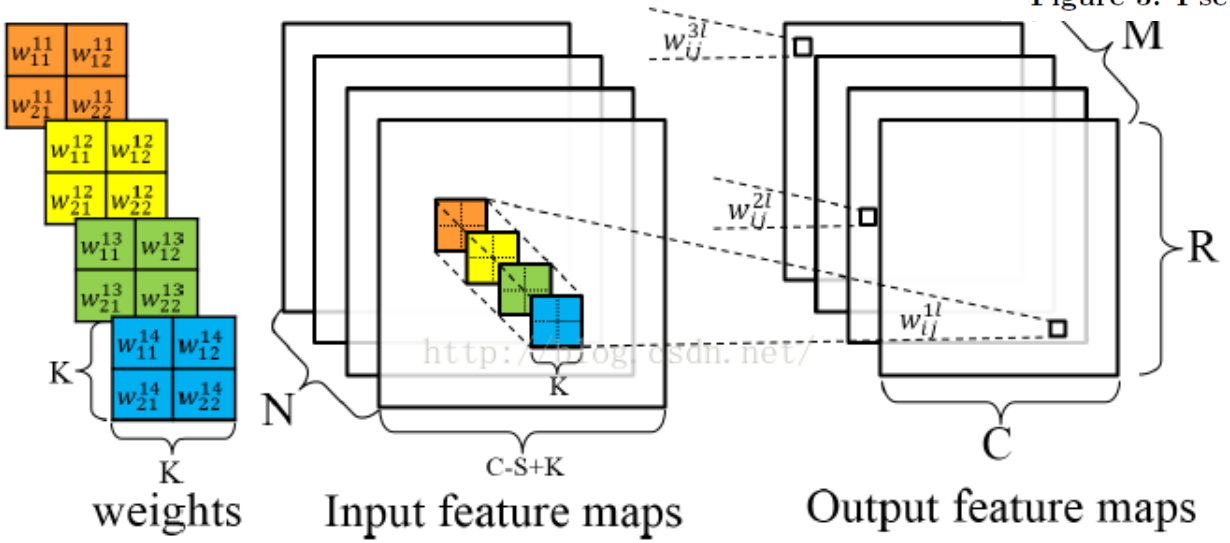


缓存友好的卷积

- 2015年UCLA
- 引用2010次

```
for(row=0; row<R; row+=Tr) {                    External data transfer
 for(col=0; col<C; col+=Tc) {                   To be discussed in Section 3.2
  for(to=0; to<M; to+=Tm) {
   for(ti=0; ti<N; ti+=Tn) {
    //load output feature maps
    //load weights
    //load input feature maps
                                                On-chip data computation
                                                To be discussed in Section 3.1
    for(trr=row; trr<min(row+Tr,R); trr++){
     for(tcc=col; tcc<min(col+Tc,C); tcc++){
      for(too=to; too<min(to+Tm,M); too++){
       for(tii=ti; tii<min(ti+Tn,N); tii++){
        for(i=0; i<K; i++) {
         for(j=0; j<K; j++) {
          L: output_fm[too][trr][tcc] +=
               weights[too][tii][i][j]*
               input_fm[tii][S*trr+i][S*tcc+j];
    } } } } } }
    //store output feature maps
} } } }
```

Figure 5: Pseudo code of a tiled convolutional layer



练习题

主观题 10分

设置

- 根据参数确定每个高速缓存的高速缓存组数(*S*)、标记位数(*t*)、组索引位数(*s*)以及块偏移位数(*b*)，系统地址位宽（m），高速缓存容量（C）

高速缓存	*m*	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1	32	1024	4	1				
2	32	1024	8	4				
3	32	1024	32	32				

作答

练习题

- 根据参数确定每个高速缓存的高速缓存组数(S)、标记位数(t)、组索引位数(s)以及块偏移位数(b)，系统地址位宽（m），高速缓存容量（C）

高速缓存	*m*	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1	32	1024	4	1	256	22	8	2
2	32	1024	8	4	32	24	5	3
3	32	1024	32	32	1	27	0	5

- 参数计算(以1为例)：
  - $S=C/B/E=1024/4/1=256$
  - $s=\log_2(S)=\log_2 256=8$
  - $b=\log_2(B)=\log_2 4=2$
  - $t=m-(s+b)=32-(8+2)=22$

练习题

- 假设高速缓存使用地址的高s位作组索引，内存块连续的片(chunk)会被映射到同一个高速缓存组
  - **A. 每个这样的连续的数组片中有多少个块？**
  - 标记位数量$t=m-(s+b)$，每个连续的数组片由2^t个块组成，数组前2^t个连续的块都会映射到组0，接下来的2^t个块会映射到组1，以此类推
  - **B. 代码运行在高速缓存形式为$(S,E,B,m)=(512,1,32,32)$系统上，任意时刻，存储在高速缓存中的数组块的最大数量是多少？**

```
int array[4096];
for (i=0; i<4096; i++)
    sum+=array[i];
```

  - 标记位数量$t=m-(s+b)=32-(\log_2 512+\log_2 32)=18$
  - 数组中前2^{18}个块映射到组1
  - 数组长度为4096，由(4096*4)/32=512个块组成，数组中所有的块都映射到组0，任何时刻，高速缓存至多保存一个数组块，即使数组能够全部容纳在cache中

练习题： 理解cache工作原理

- 假设：
  - 内存是字节寻址的
  - 内存访问是1字节的字，不是4字节的字
  - 地址的宽度为13位
  - Cache是2路组相联的(E=2)，块大小为4字节(B=4)，有8个组(S=8)

	行0						行1					
组索引	标记位	有效位	字节0	字节1	字节2	字节3	标记位	有效位	字节0	字节1	字节2	字节3
0	09	1	86	30	3F	10	00	0	——	——	——	——
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	——	——	——	——	0B	0	——	——	——	——
3	06	0	——	——	——	——	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	——	——	——	——
6	91	1	A0	B7	26	2D	F0	0	——	——	——	——
7	46	0	——	——	——	——	DE	1	12	C0	88	37

- 标出下列内容字段：
  - CO:cache块偏移，CI:cache组索引；CT:cache标记

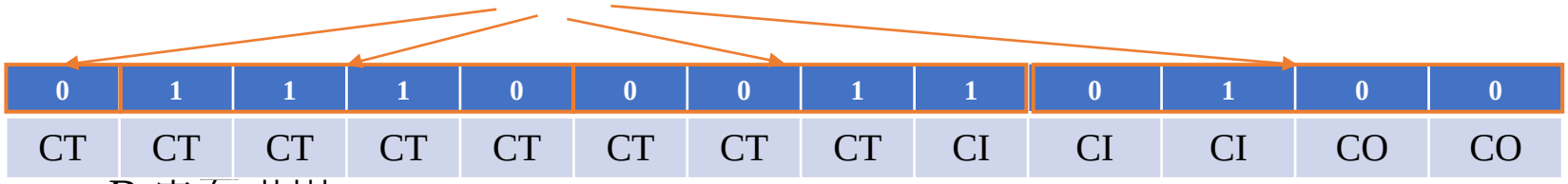
$\log_2 8$

$\log_2 4$

12	11	10	9	8	7	6	5	4	3	2	1	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

练习题

- 程序引用0x0E34处1字节的字，指出访问的cache条目和十六进制表示的返回的cache字节值，指出是否发生cache miss，如果发生cache miss，用—表示返回的cache缓存字节
  - A.地址格式： 0x0E34



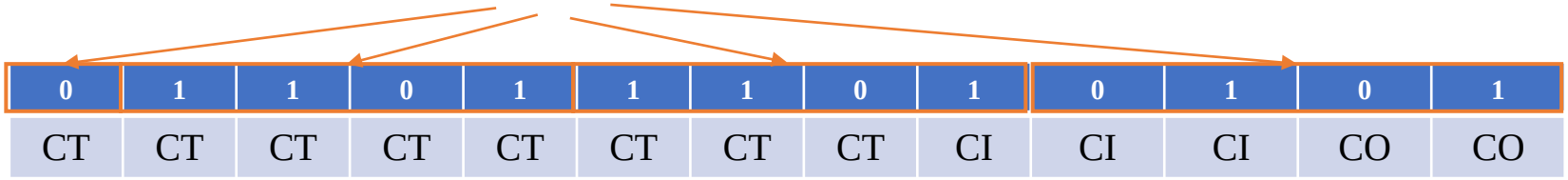
  - B.内存引用：

参数	值
Cache块偏移(CO)	0x 0
Cache组索引(CI)	0x 5
Cache标记(CT)	0x 71
Cache hits?	是
返回cache字节	0x B

	行0						行1					
组索引	标记位	有效位	字节0	字节1	字节2	字节3	标记位	有效位	字节0	字节1	字节2	字节3
0	09	1	86	30	3F	10	00	0	——	——	——	——
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	——	——	——	——	0B	0	——	——	——	——
3	06	0	——	——	——	——	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	——	——	——	——
6	91	1	A0	B7	26	2D	F0	0	——	——	——	——
7	46	0	——	——	——	——	DE	1	12	C0	88	37

练习题

- 程序引用0x0DD5处1字节的字，指出访问的cache条目和十六进制表示的返回的cache字节值，指出是否发生cache miss，如果发生cache miss，用—表示返回的cache缓存字节
  - A.地址格式： 0x0DD5



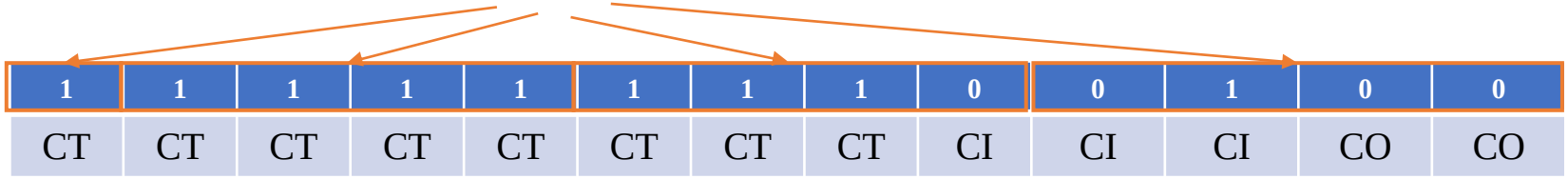
  - B.内存引用：

参数	值
Cache块偏移(CO)	0x 1
Cache组索引(CI)	0x 5
Cache标记(CT)	0x 6E
valid?	否
Cache hits?	否
返回cache字节	—

	行0						行1					
组索引	标记位	有效位	字节0	字节1	字节2	字节3	标记位	有效位	字节0	字节1	字节2	字节3
0	09	1	86	30	3F	10	00	0	——	——	——	——
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	——	——	——	——	0B	0	——	——	——	——
3	06	0	——	——	——	——	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	——	——	——	——
6	91	1	A0	B7	26	2D	F0	0	——	——	——	——
7	46	0	——	——	——	——	DE	1	12	C0	88	37

练习题

- 程序引用0x1FE4处1字节的字，指出访问的cache条目和十六进制表示的返回的cache字节值，指出是否发生cache miss，如果发生cache miss，用—表示返回的cache缓存字节
  - A.地址格式：0x1FE4



  - B.内存引用：

参数	值
Cache块偏移(CO)	0x 0
Cache组索引(CI)	0x 1
Cache标记(CT)	0x FF
Cache hits?	否
返回cache字节	—

	行0						行1					
组索引	标记位	有效位	字节0	字节1	字节2	字节3	标记位	有效位	字节0	字节1	字节2	字节3
0	09	1	86	30	3F	10	00	0	——	——	——	——
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	——	——	——	——	0B	0	——	——	——	——
3	06	0	——	——	——	——	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	——	——	——	——
6	91	1	A0	B7	26	2D	F0	0	——	——	——	——
7	46	0	——	——	——	——	DE	1	12	C0	88	37

练习题

- 列出所有的在组3中会命中的十六进制内存地址

	行0						行1					
组索引	标记位	有效位	字节0	字节1	字节2	字节3	标记位	有效位	字节0	字节1	字节2	字节3
3	06	0	——	——	——	——	32	1	12	08	7B	AD

- 组3包含一个有效行，标记为0x32，有效位为1，4个地址会命中，这些地址的有效形式为：

0	0	1	1	0	0	1	0	0	1	1		
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

-

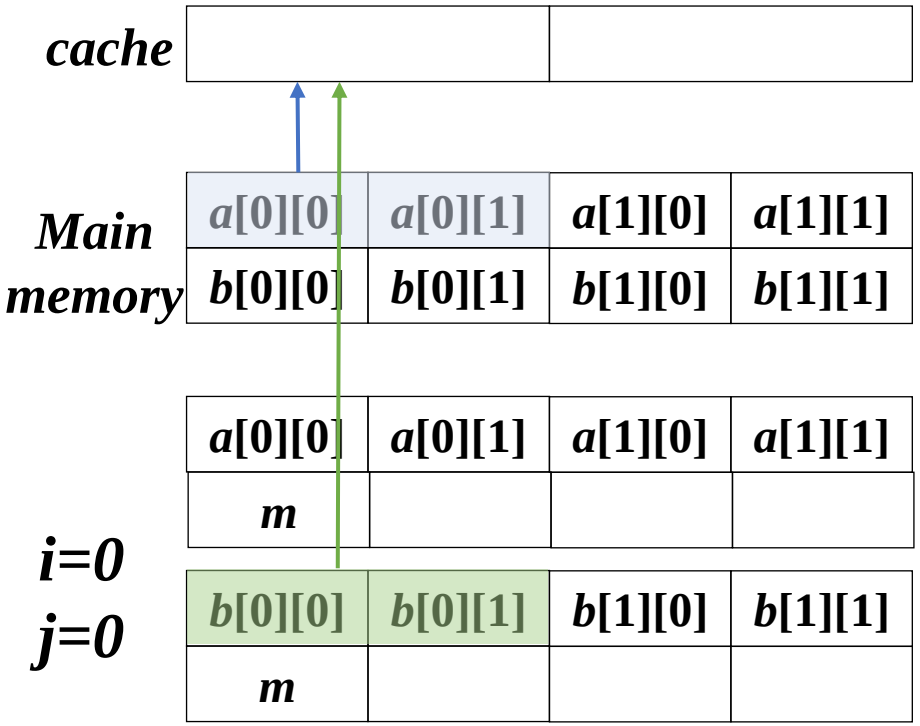
0	0	1	1	0	0	1	0	0	1	1	0	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO
0	0	1	1	0	0	1	0	0	1	1	1	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO
0	0	1	1	0	0	1	0	0	1	1	1	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO
0	0	1	1	0	0	1	0	0	1	1	1	1
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

  - 0x 064C, 0x 064D, 0x 064E, 0x 064F

练习题： 转置矩阵

- 假设：
  - Sizeof(int)=4
  - Src数组从地址0开始，dst数组从地址16开始
  - 只有一个L1 cache，直接映射、直写和写分配，块大小为8字节
  - Cache大小为16字节，开始为空
  - Src和dst数组访问分别是读和写不命中的唯一来源

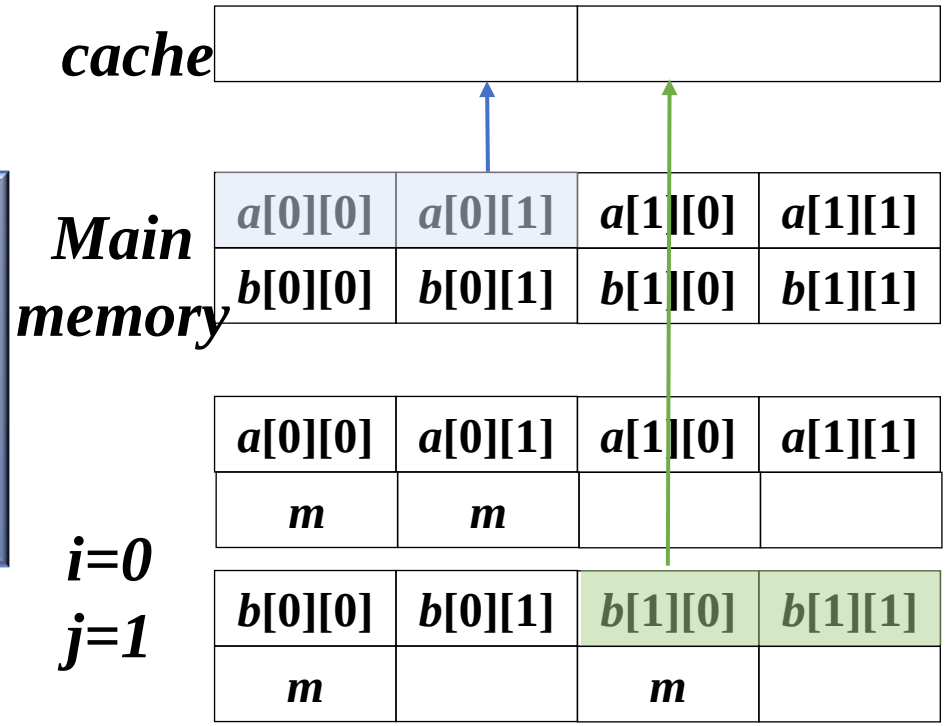
```
Typedef int array[2][2]
Void transpose1(array dst, array src)
{
      int I,j;
      for(i=0;i<2;i++){
            for(j=0;j<2;j++){
                dst[j][i]=src[i][j];
            }
      }
}
```



练习题：转置矩阵

- 假设：
  - Sizeof(int)=4
  - Src数组从地址0开始，dst数组从地址16开始
  - 只有一个L1 cache，直接映射、直写和写分配，块大小为8字节
  - Cache大小为16字节，开始为空
  - Src和dst数组访问分别是读和写不命中的唯一来源

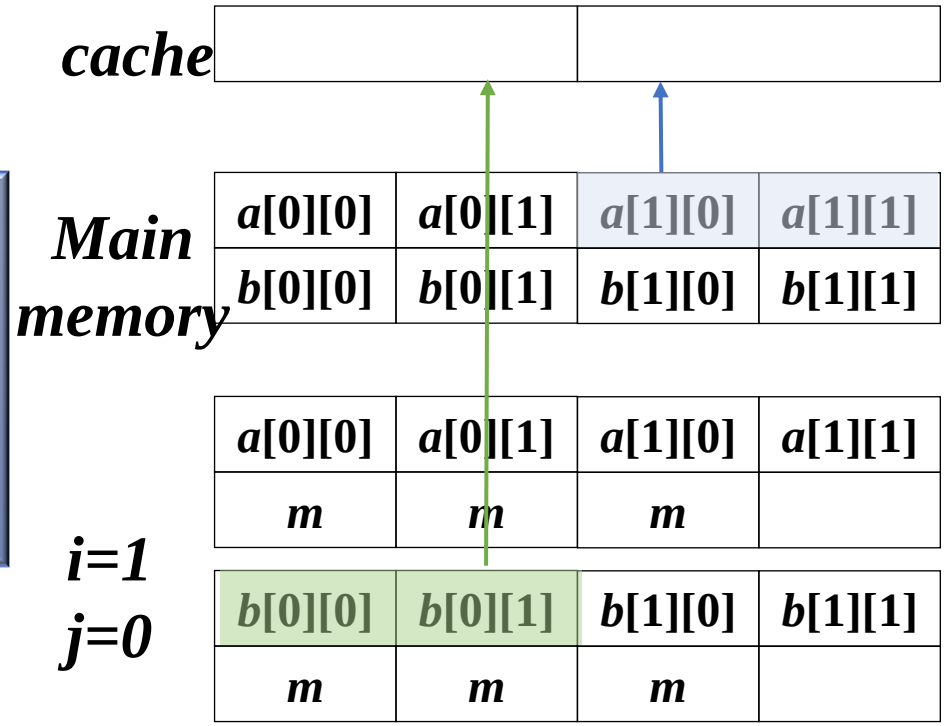
```
Typedef int array[2][2]
Void transpose1(array dst, array src)
{
      int I,j;
      for(i=0;i<2;i++){
            for(j=0;j<2;j++){
                dst[j][i]=src[i][j];
            }
      }
}
```



练习题： 转置矩阵

- 假设：
  - Sizeof(int)=4
  - Src数组从地址0开始，dst数组从地址16开始
  - 只有一个L1 cache，直接映射、直写和写分配，块大小为8字节
  - Cache大小为16字节，开始为空
  - Src和dst数组访问分别是读和写不命中的唯一来源

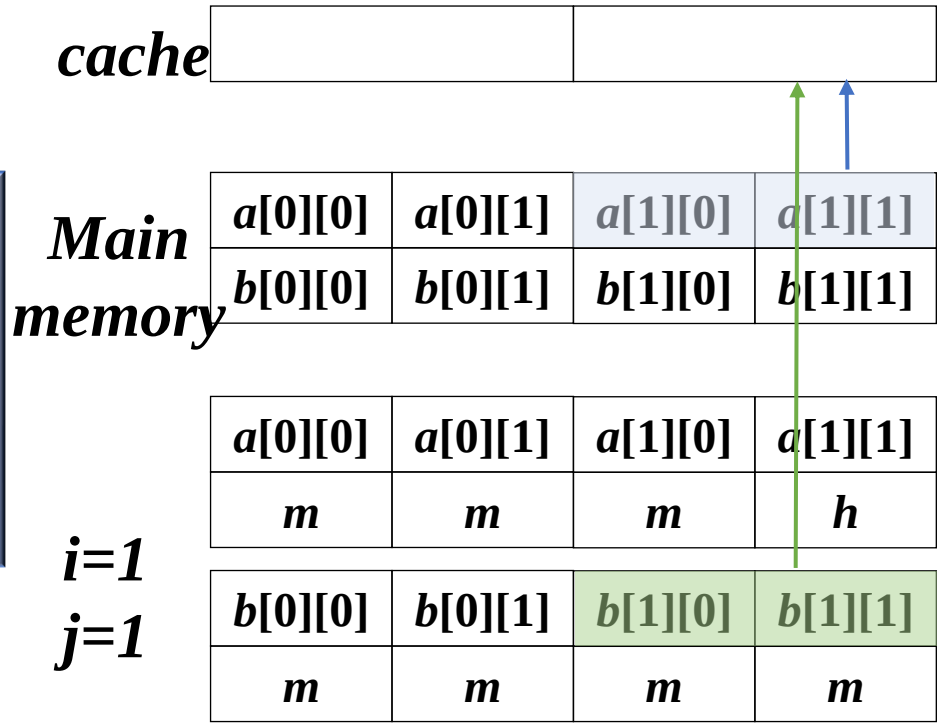
```
Typedef int array[2][2]
Void transpose1(array dst, array src)
{
      int I,j;
      for(i=0;i<2;i++){
            for(j=0;j<2;j++){
                dst[j][i]=src[i][j];
            }
      }
}
```



练习题： 转置矩阵

- 假设：
  - Sizeof(int)=4
  - Src数组从地址0开始，dst数组从地址16开始
  - 只有一个L1 cache，直接映射、直写和写分配，块大小为8字节
  - Cache大小为16字节，开始为空
  - Src和dst数组访问分别是读和写不命中的唯一来源

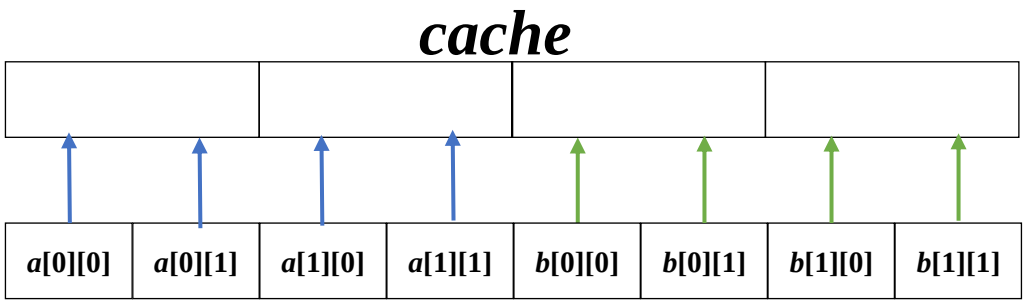
```
Typedef int array[2][2]
Void transpose1(array dst, array src)
{
      int I,j;
      for(i=0;i<2;i++){
            for(j=0;j<2;j++){
                dst[j][i]=src[i][j];
            }
      }
}
```



练习题：转置矩阵

- 假设：
  - Sizeof(int)=4
  - Src数组从地址0开始，dst数组从地址16开始
  - 只有一个L1 cache，直接映射、直写和写分配，块大小为8字节
  - Cache大小为32字节，开始为空
  - Src和dst数组访问分别是读和写不命中的唯一来源

```
Typedef int array[2][2]
Void transpose1(array dst, array src)
{
      int I,j;
      for(i=0;i<2;i++){
            for(j=0;j<2;j++){
                dst[j][i]=src[i][j];
            }
      }
}
```

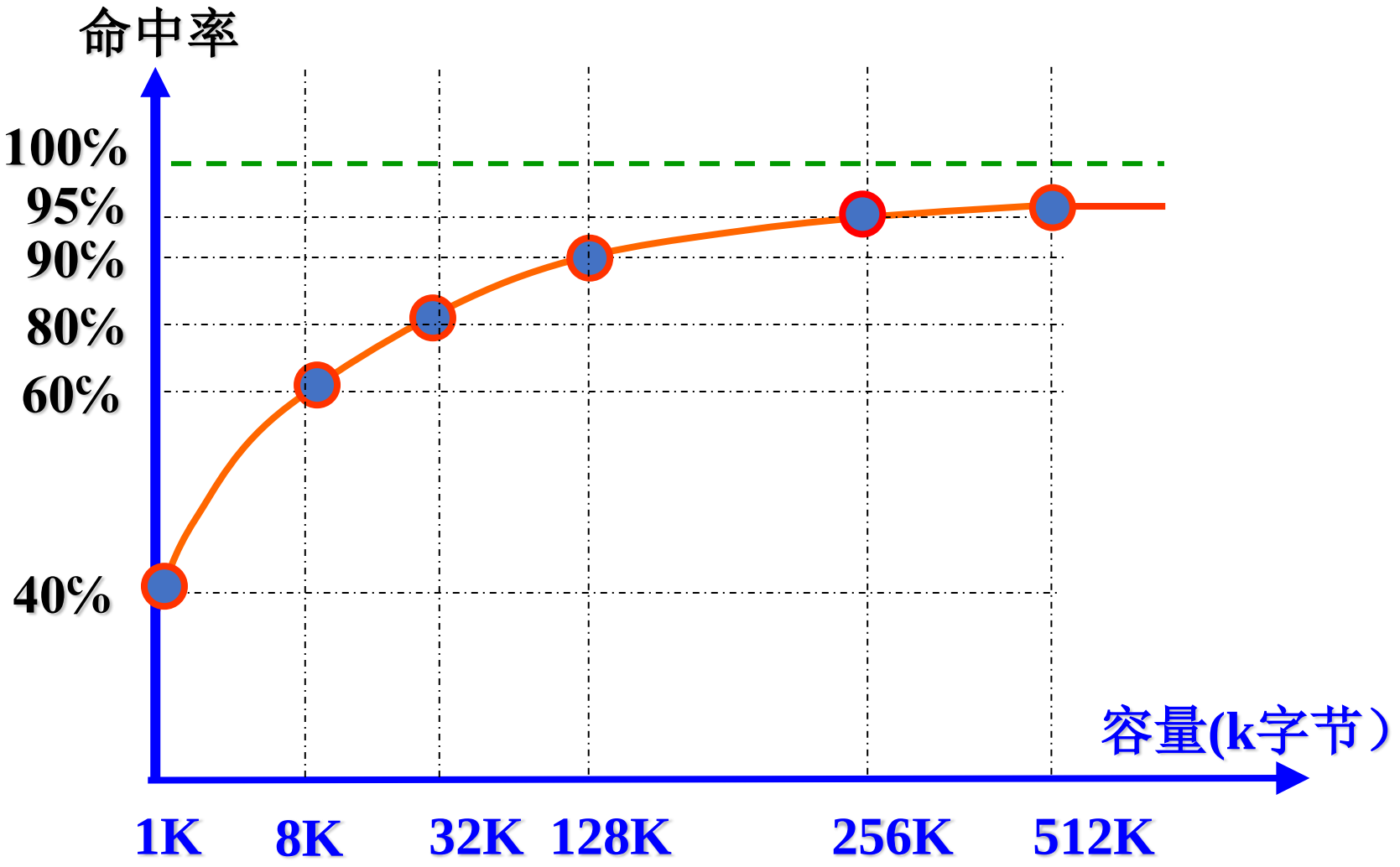


Main memory

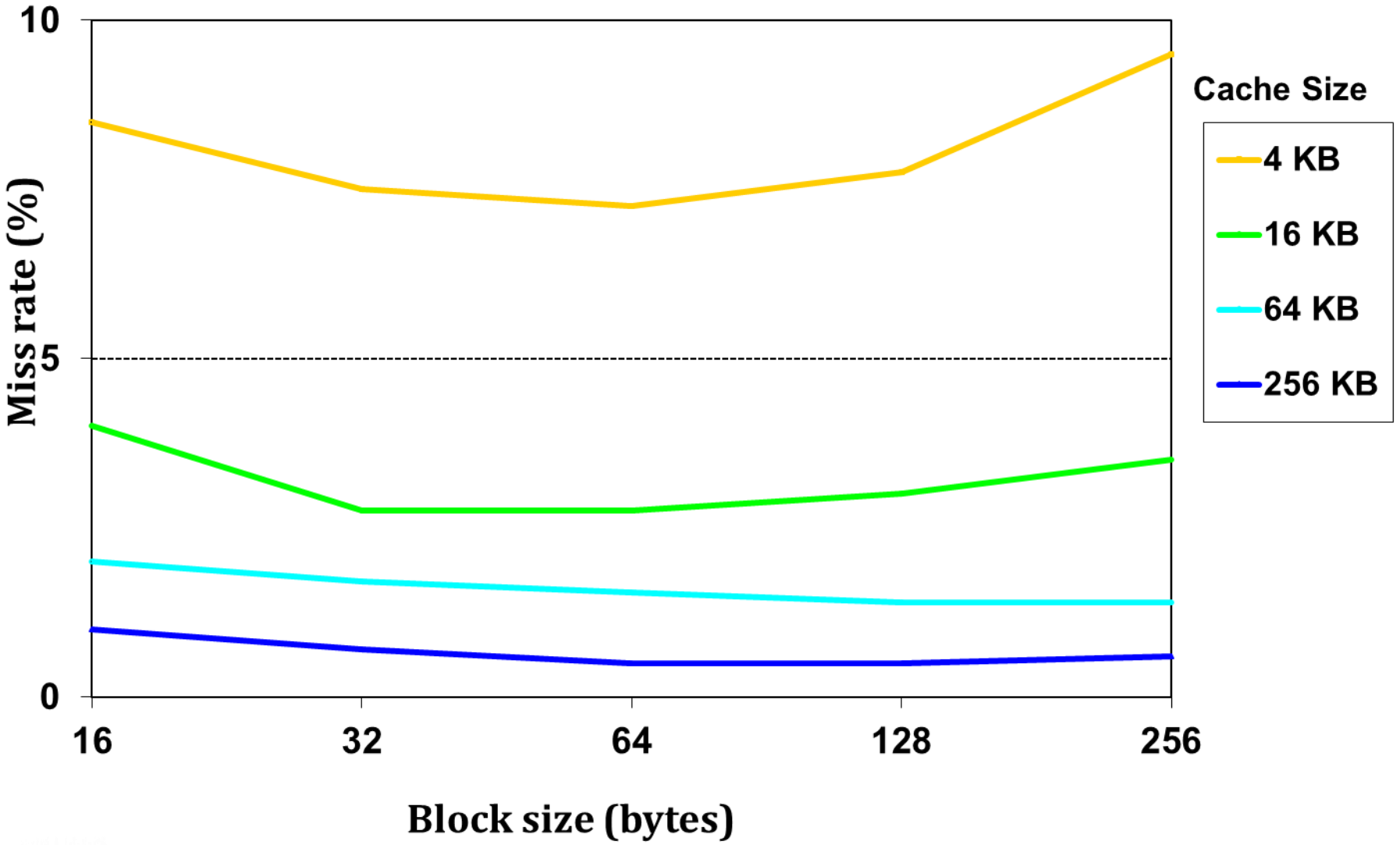
a[0][0]	a[0][1]	a[1][0]	a[1][1]
m	h	m	h

b[0][0]	b[0][1]	b[1][0]	b[1][1]
m	h	m	h

Cache命中率与容量的关系



Cache命中率与块大小的关系



直接映象Cache命中率与块大小分析

- 考虑如下的访问序列

0 1 2 3 4 3 4 15

0000 0001 0010 0011 0100 0011 0100 1111

0 miss

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

01 4

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

15 miss

11 15

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

- **8 次请求, 6次失效 (命中率 = 0.25,失效率 = 0.75)**

直接映象Cache命中率与块大小分析

- 让每个cache块多于1个字节

0	1	2	3	4	3	4	15
0000	0001	0010	0011	0100	0011	0100	1111

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

4 miss

01

~~00~~	~~Mem(1)~~ 5	~~Mem(0)~~ 4
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

11

01	Mem(5)	Mem(4)
~~00~~	~~Mem(3)~~ 15	~~Mem(2)~~ 14

- **8 次请求, 4次失效 (命中率= 0.5, 失效率= 0.5)**

替换策略

- （1）随机数替换算法
  - 当需要找替换块时，产生一个随机数，它就是被替换的块号。
  - 这种算法完全不反映程序的局部性特点，只是算法简单、实现容易。
  - 结论：不是一个好的算法。
- （2）FIFO算法（先进先出）
  - 总是把一组中最先调入 Cache存储器的字块替换出去，它不需要随时记录各个字块的使用情况，所以实现容易，开销小。
  - 缺点：它没有根据访存局部性原理，最早调入的块可能是以后还要用到的，或者是经常用到的。
- （3）LRU算法（近期最少使用法）
  - 把Cache中近期最少使用的块替换出去。

练习题 紧密循环cache命中率计算

- 硬件假设：
  - 块大小：16字节
  - Sizeof(int)=4
  - Cache大小1024字节
- 计算：
  - A. 读总数
  - B. Cache不命中的读总数
  - C. 不命中率

```
Struct algae_position{
      int x,y;
}
Struct algae_position grid[16][16];
Int total_x=0, total_y=0;
Int I,j;

For(i=0;i<16;i++){
      for(j=0;j<16;j++){
          total_x+=grid[i][j].x;
      }
}
For(i=0;i<16;i++){
      for(j=0;j<16;j++){
          total_y+=grid[i][j].y;
      }
}
```

作答

练习题： 紧密循环cache命中率计算

- 硬件假设：
  - 块大小：16字节
  - Sizeof(int)=4
  - Cache大小1024字节
- 计算：
  A. 读总数
  B. Cache不命中的读总数
  C. 不命中率

```
Struct algae_position{
      int x,y;
}
Struct algae_position grid[16][16];
Int total_x=0, total_y=0;
Int I,j;

For(i=0;i<16;i++){
      for(j=0;j<16;j++){
          total_x+=grid[i][j].x;
      }
}
For(i=0;i<16;i++){
      for(j=0;j<16;j++){
          total_y+=grid[i][j].y;
      }
}
```

A. 读总数为：512
B. 缓存不命中的读总数：256
C. 不命中率：256/512=50%

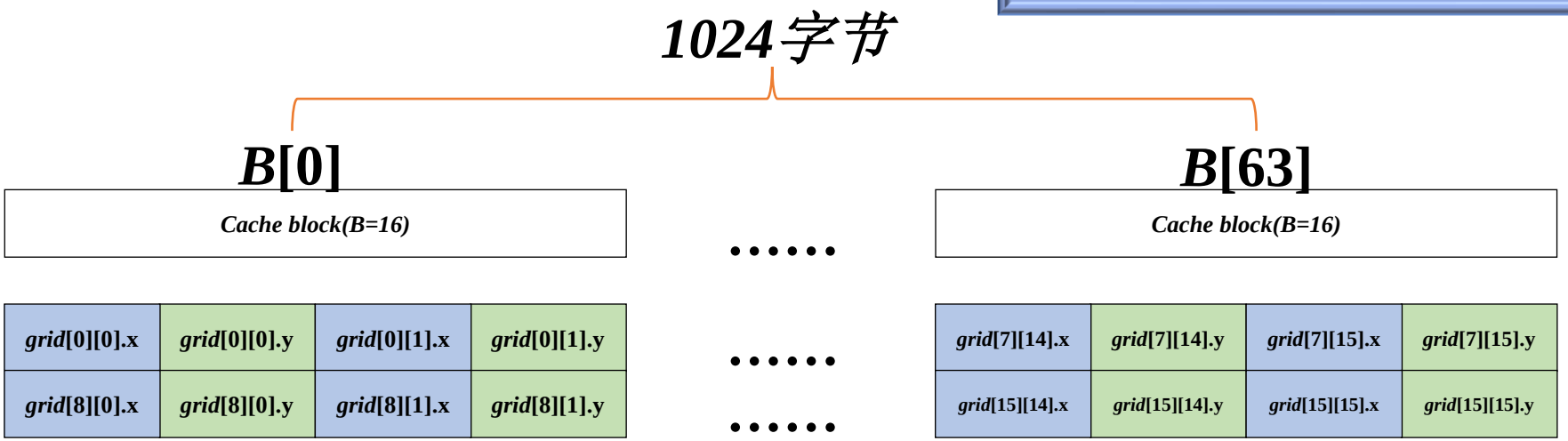
练习题

A. 读总数为：512

B. 缓存不命中的读总数：
- 每次j循环前1024字节产生8个不命中
- 后1024字节与前一半循环映射冲突，产生8个不命中
- 16次循环共16*16=256次不命中

C. 不命中率：256/512=50%

```
For(i=0;i<16;i++){
      for(j=0;j<16;j++){
          total_x+=grid[j][i].x;
          total_y+=grid[j][i].y;
      }
}
```



练习题

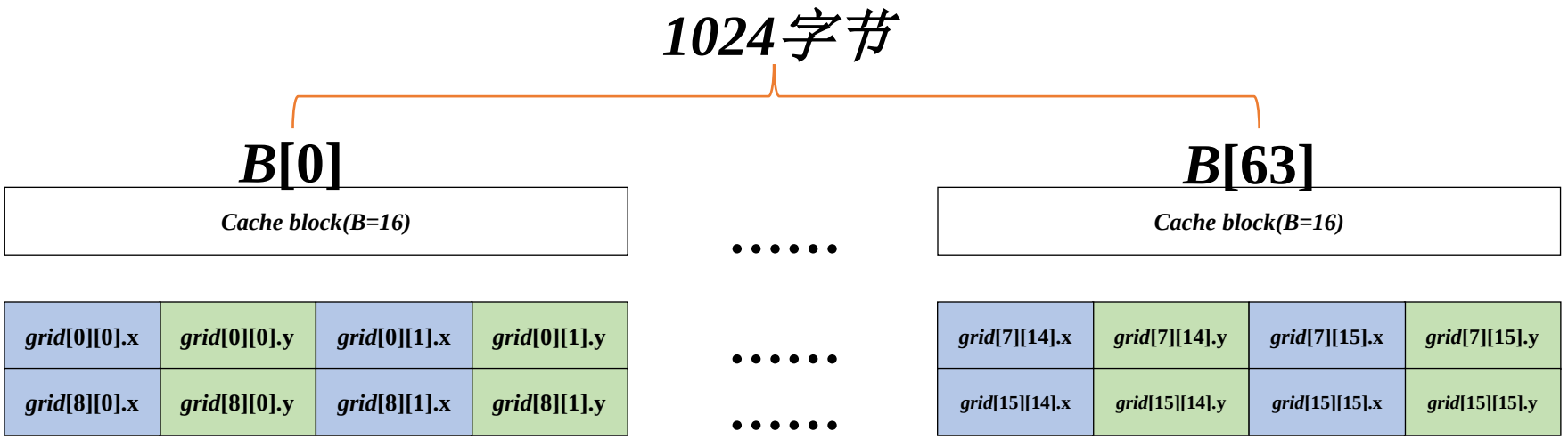
A. 读总数为：512

B. 缓存不命中的读总数：
- 顺序步长为1访问，不命中为强制不命中
- 不命中数为64*2=128

C. 不命中率：128/512=25%

D. 缓存加倍，与数据相同时，强制不命中数量不会改变

```
For(i=0;i<16;i++){
      for(j=0;j<16;j++){
          total_x+=grid[i][j].x;
          total_y+=grid[i][j].y;
      }
}
```



Cache Summary

- Cache memories can have significant performance impact

- You can write your programs to exploit this!
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.