超星学习通 QR Code

- 作业会放到里面发布

缓存 (Cache)

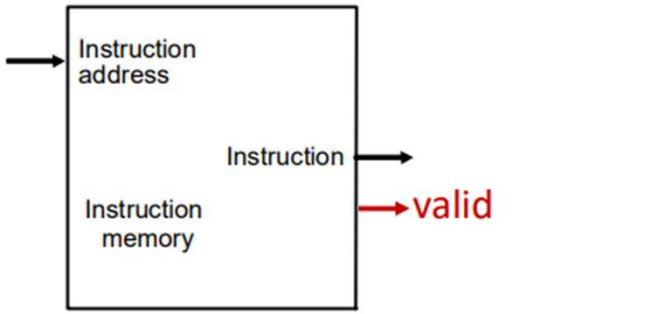
COA: 计算机系统与结构
Sep. 2, 2025

在计算机领域，缓存是一种通用概念：任何能够“记住”经常重复计算结果的结构，都可以称作缓存。通过存储这些结果，我们就可以避免每次都从零开始重新计算，从而节省时间和资源。例如：网页缓存（web cache）。

计算机体系结构中的缓存（Cache）

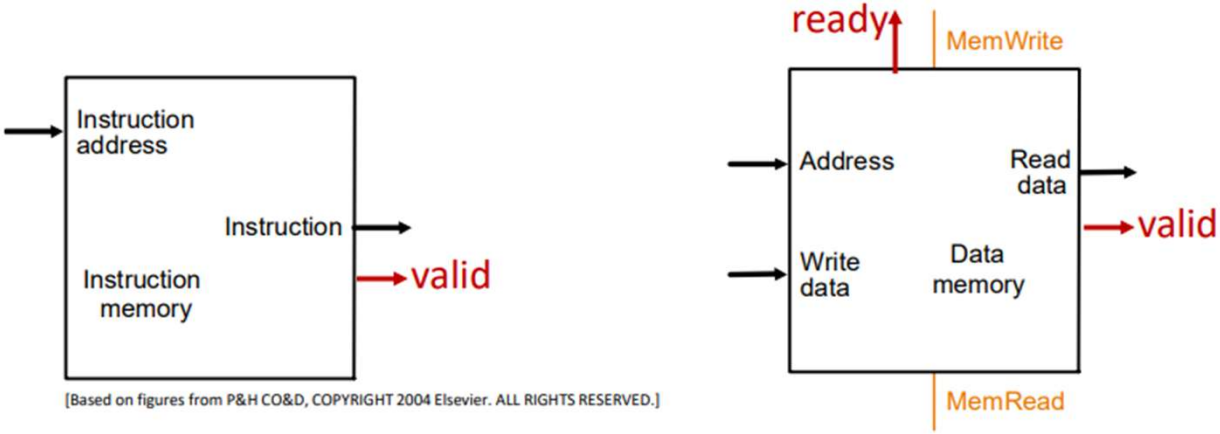
- 缓存是一个“不可见的”、自动管理的内存层次结构
- CPU 首先查找缓存中的数据
- 缓存的核心功能：
  - 缓存会在一块小而快的存储器中，保存一些频繁访问的 DRAM 数据副本
- 程序的预期：
  - 当你在代码中写 x = M[A] 时，你只关心能不能读到数据，而不需要知道它是从 L1 缓存、L2 缓存还是 DRAM 中取出来的。
- 程序一般不需要为缓存做特别处理，但是
  - 多核处理器访问同一块内存；
  - DMA（直接内存访问）绕过缓存直接修改数据；

缓存类型：指令缓存（I-Cache）



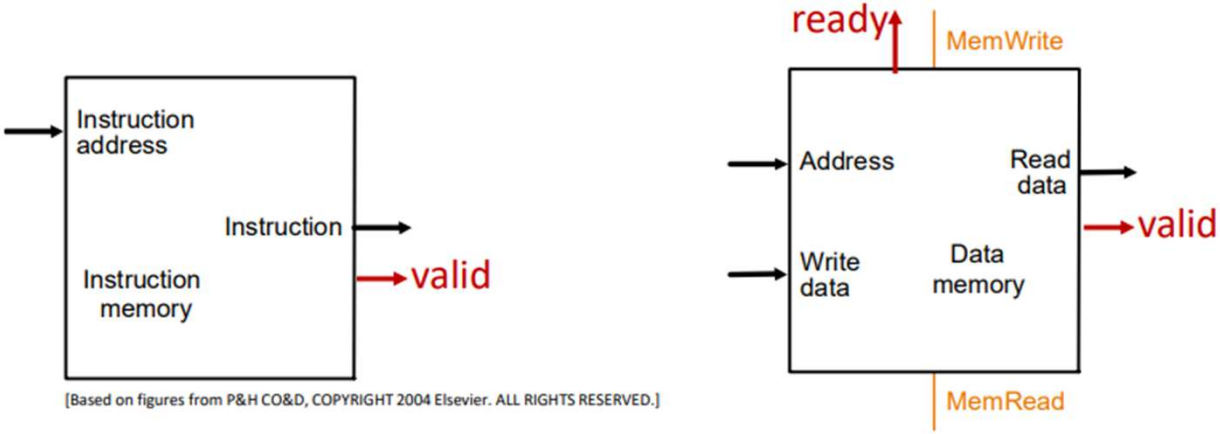
- 指令缓存（Instruction Cache，I-Cache）
  - 专门用来缓存程序指令的小型高速存储器。
  - 当 CPU 需要执行一条指令时，它会先从 I-Cache 中查找对应的指令，而不是直接去主存取。
  - 左边的箭头：CPU → 指令缓存，CPU 把想要取的指令地址发给指令缓存。
  - 右边的箭头 ：指令缓存 → CPU，如果缓存命中，指令缓存会把对应的指令内容返回给 CPU。
  - “valid” 标签：与右边箭头绑定，表示返回的指令是否有效：
    - valid = 1 → 指令已经准备好；
    - valid = 0 → 指令缓存还在等待数据（比如需要去更低级缓存或内存取）。

缓存类型：数据缓存（D-Cache）



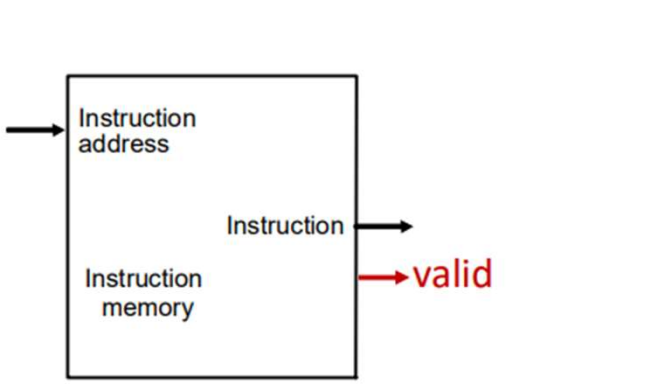
- 数据缓存（Data Cache，D-Cache）
  - 专门用来缓存程序运行时访问的数据的小型高速存储器。
  - 当 CPU 需要读取或写入变量、数组、对象等数据时，会先从 D-Cache 查找。
  - 只存放数据，不存放指令。既有读取（load）操作，又有写入（store）操作。
  - 左边 → Address：CPU 提供要访问的**数据地址**。
  - 左下角 → Write data：如果 CPU 要写数据，就把要写入的内容传给缓存。
  - 上/下方控制箭头 → MemWrite/Read：告诉缓存“我要写/读数据”。
  - 右边 → Read data：如果缓存命中且是读操作，把对应的数据返回给 CPU。
  - “valid” 标签：表示**返回的数据是否准备好**。
  - “ready” 标签：表示缓存是否准备好接收新的读写请求：

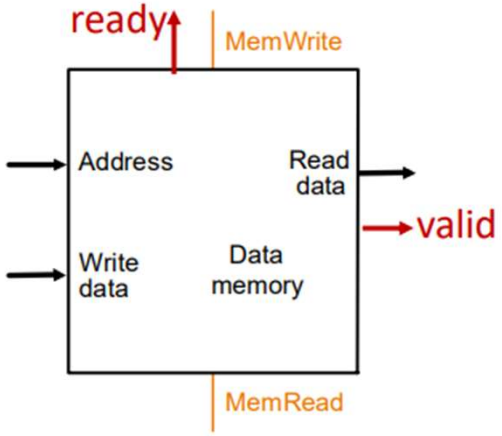
为什么要分开设计 I-Cache 和 D-Cache



- 现代 CPU 通常采用 Harvard 架构（哈佛架构）：
  - I-Cache 和 D-Cache 分开设计，独立访问。
  - 优点：
    - 并行性更高：取指令和取数据可以同时进行。
    - 优化更灵活：指令缓存只读优化，数据缓存读写优化。
    - 减少冲突：避免指令和数据互相抢缓存空间。
- 如果把两者放在一起（统一缓存，Von Neumann 架构），当 CPU 同时需要取指令和读数据时，发生排队，会降低性能。

缓存接口：简单抽象





- **缓存接口：**
  - CPU 给出地址、读/写命令等请求
  - 缓存会在极短且固定的延迟后返回结果或完成更新
- **但有时候会遇到例外，当缓存未命中或者需要额外时间处理时：**
  - 数据最终会变得 valid/ready（数据总能在更低层级找到，缓存处理任务总会完成）
  - 但是在此期间，CPU 的流水线**必须等待**
  - 这种等待现象被称为 **流水线停顿（stall）**

缓存的核心问题

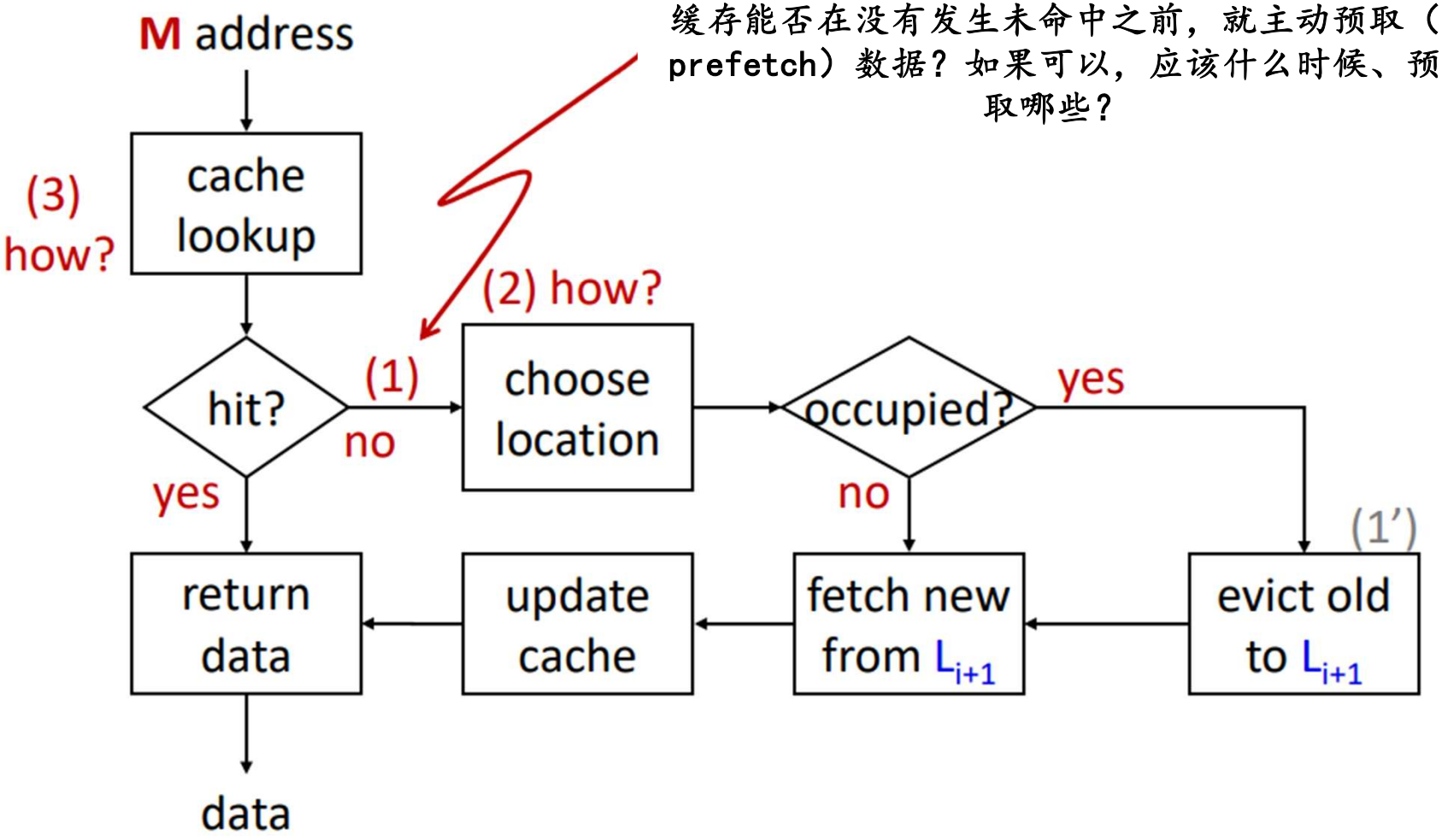
- 缓假设系统有 M = 2^m 字节的主存储器，我们需要在只有 C 字节的小型高速存储器（缓存）中，保存最常用数据的副本，且满足 C << M（缓存容量远小于主存）。

- 缓存设计的三大基本问题（相互关联）
  1. 何时缓存（When）
     - 什么时候应该把某个内存位置的数据副本放到缓存里？
  2. 缓存位置（Where）
     - 在有限的高速缓存空间里，应该把这个副本放在哪一块？
  3. 如何找到（How）
     - 当再次访问该数据时，如何快速在缓存中找到对应的副本？

- 必须既高效又快速，否则缓存的存在就失去了意义。

答案（1）：按需驱动（demand-driven）

缓存能否在没有发生未命中之前，就主动预取（prefetch）数据？如果可以，应该什么时候、预取哪些？



缓存的基本参数

- M = 2^m ：地址空间大小（以字节为单位）
  - 例如：2^{32}（4GB）、2^{64}（16EB）
- G = 2^g ：缓存访问粒度（以字节为单位）
  - 例如：4 字节、8 字节
- C ：缓存容量（以字节为单位）
  - 例如：L1 缓存 16KB，L2 缓存 1MB
- B = 2^b ：缓存块大小（以字节计）
  - 示例：L1 缓存常见 16B，L2 缓存常见 ≥64B
- a ：缓存的相联度（associativity）
  - 示例：1、2、4、，，“C/B”
  - 直接映射、全相联映射和组相联映射

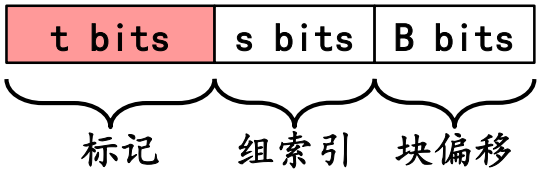
内存块与缓存寻址

- 内存会被逻辑上划分为固定大小的块:内存块（memory block）

- 每个内存块会映射到缓存中的某个位置，这个位置由内存地址中的索引位（index bits）决定
  - 用于定位标记（tag）和数据（data）

内存块与缓存寻址

CPU 想访问某个数据时，它会给出一个内存地址：



- 假设我们有一个8 位的内存地址，地址会被拆成三部分：

地址部分	作用	类比
标记位 Tag	确认缓存里存的数据是不是我们要的	书名
索引位 Index	告诉我们数据在哪个缓存槽位	书架号
块内偏移 Offset	定位块内的具体字节	页码

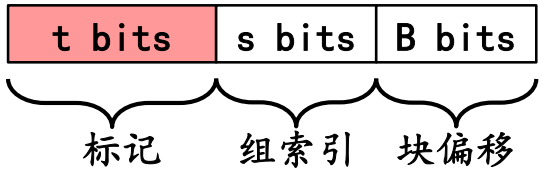
- 地址总长：8 位，标记位 Tag：2位，索引位 Index：3位，块内偏移 Offset：3位
- 举个例子，如果地址是 101 011 00：
  - 前 2 位是 Tag → 确认是不是我们想要的那本书
  - 中间 3 位是 Index → 去找第几个缓存槽
  - 后 3 位是 Offset → 这个块里的第几个字节

内存块与缓存寻址

- 内存会被逻辑上划分为固定大小的块:内存块（memory block）

- 每个内存块会映射到缓存中的某个位置，这个位置由内存地址中的索引位（index bits）决定
  - 用于定位标记（tag）和数据（data）

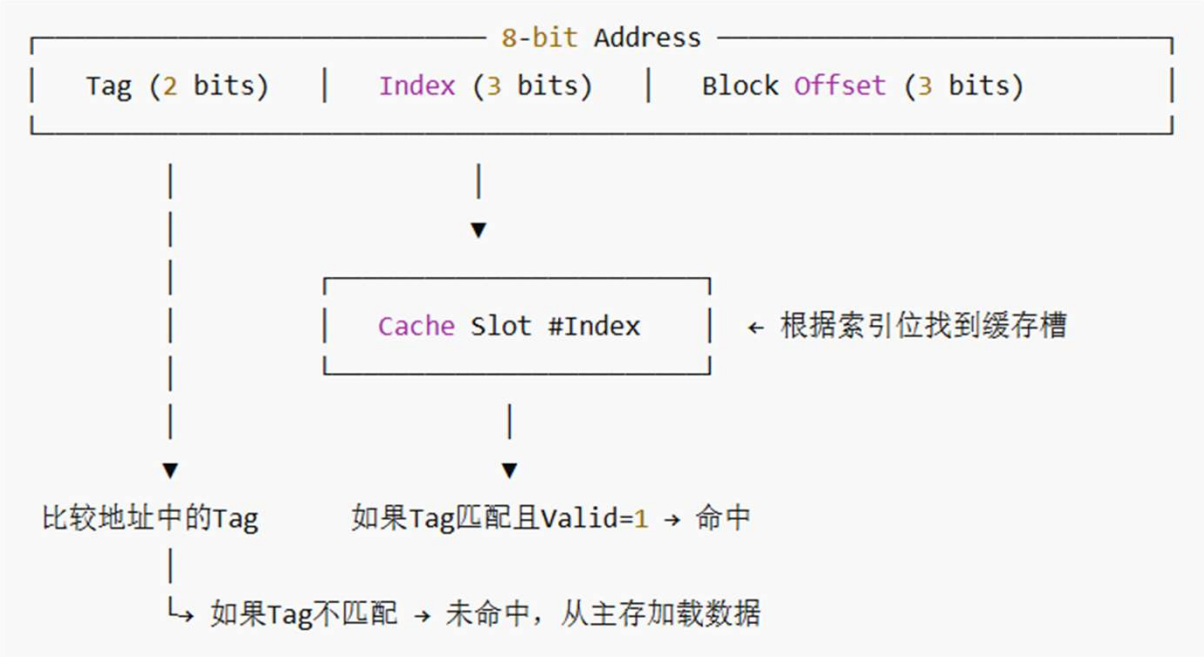
CPU 想访问某个数据时，它会给出一个内存地址:



- 缓存访问流程:
  - 当 CPU 给出一个内存地址，缓存访问分三步走:
    1. 根据索引位定位缓存槽位
       - 就像先找到哪一排书架。
    2. 检查有效位（valid bit）
       - 确认这个槽里是否有最新数据。
    3. 比较标记位
       - 如果缓存里存的 Tag 和地址中的 Tag 相同 → 缓存命中（Cache Hit），直接取数据;
       - 如果不一样 → 缓存未命中（Cache Miss），需要去主存加载数据。

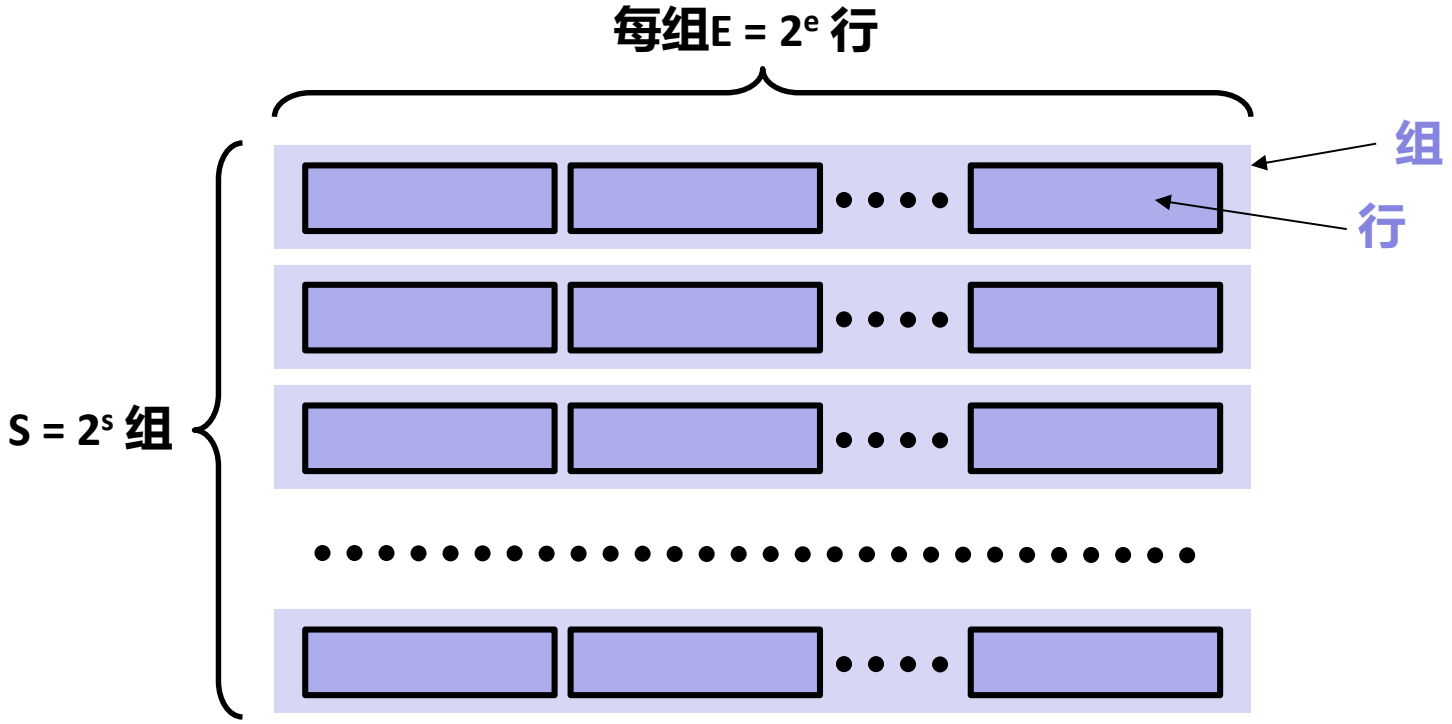
内存块与缓存寻址

- 内存会被逻辑上划分为固定大小的块:内存块（memory block）
- 每个内存块会映射到缓存中的某个位置，这个位置由内存地址中的索引位（index bits）决定
  - 用于定位标记（tag）和数据（data）
- 缓存访问流程:

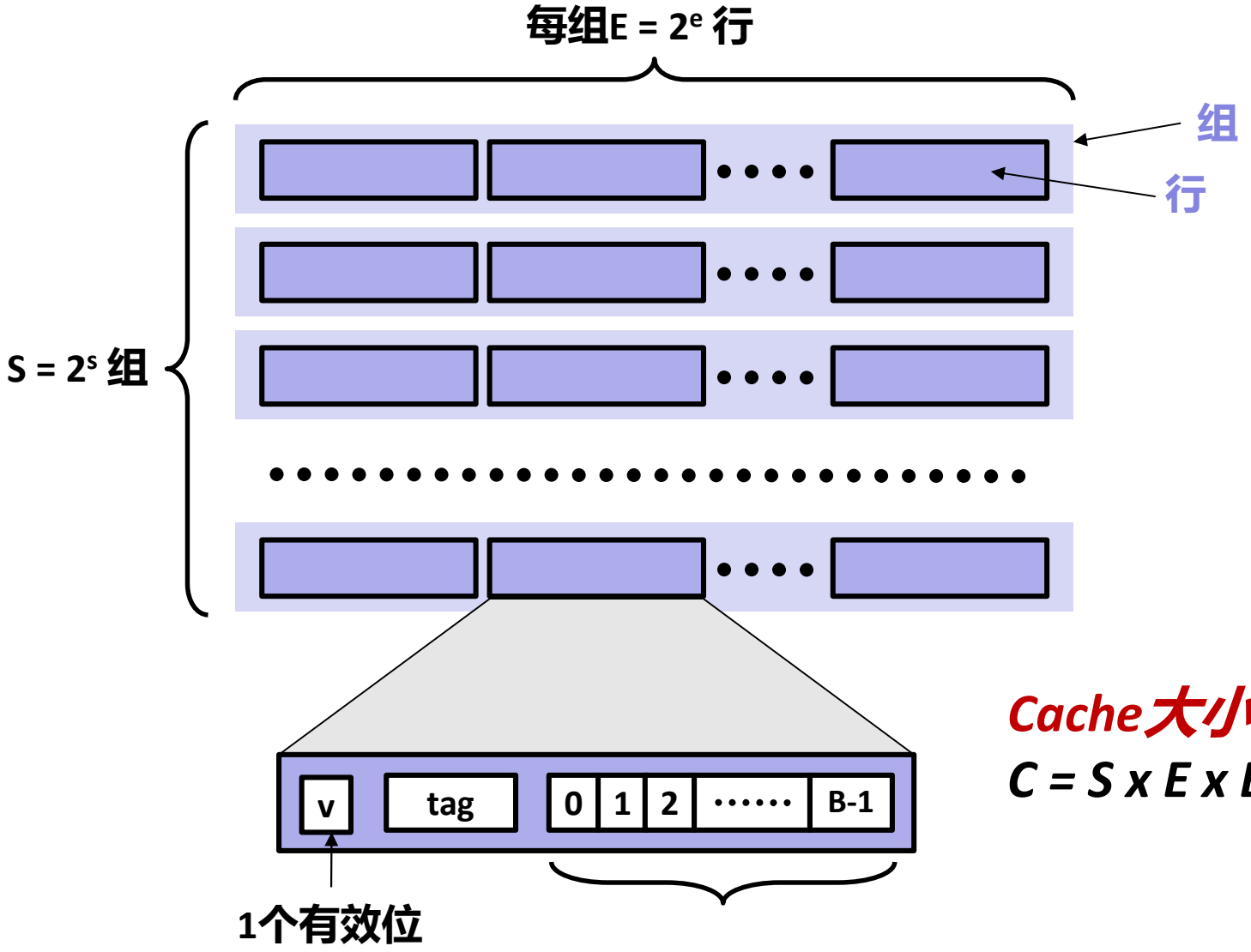


1. 根据索引位定位缓存槽
2. 地址中的 Tag 和 缓存中对应槽位的 Tag 比较，确认是不是我们要的数据
3. 根据块内偏移找到具体字节

一般Cache组织结构(S, E, B)



一般Cache组织结构(S, E, B)



Cache大小:

C = S x E x B 字节

Cache 的三大核心参数：S、E、B

- 1. S：组的数量（Sets）
  - 缓存被分成了 S 个组。
  - 每个组可以看作是一排“书架”。
  - CPU 地址中的**索引位（index bits）**会告诉我们，数据在哪一个组中。
  - 例如，如果 S = 8，那么缓存就有 8 个组，索引位就需要 3 位（因为 $2^3 = 8$）。
- 2. E：每组的行数（Lines per Set）
  - 每个组内部有 E 行（也称 E 个缓存槽位）。
  - E 决定了同一组里能存放多少不同的数据块。
    - 如果 E = 1 → 每组只有一行 → 直接映射缓存（Direct-Mapped Cache）。
    - 如果 E 很大，甚至能容纳所有数据块 → 全相联缓存（Fully Associative Cache）。
    - 如果 1 < E < 全部 → 组相联缓存（Set-Associative Cache）。
- 3. B：每行的数据块大小（Block Size）
  - 每一行缓存存放的数据量是 B 字节。
  - CPU 地址中的块内偏移（block offset）用来找到块内的具体字节。
  - 例如，如果 B = 8，那么每行缓存存 8 个字节，块内偏移就需要 3 位（因为 $2^3 = 8$）。

Cache 的三大核心参数：S、E、B

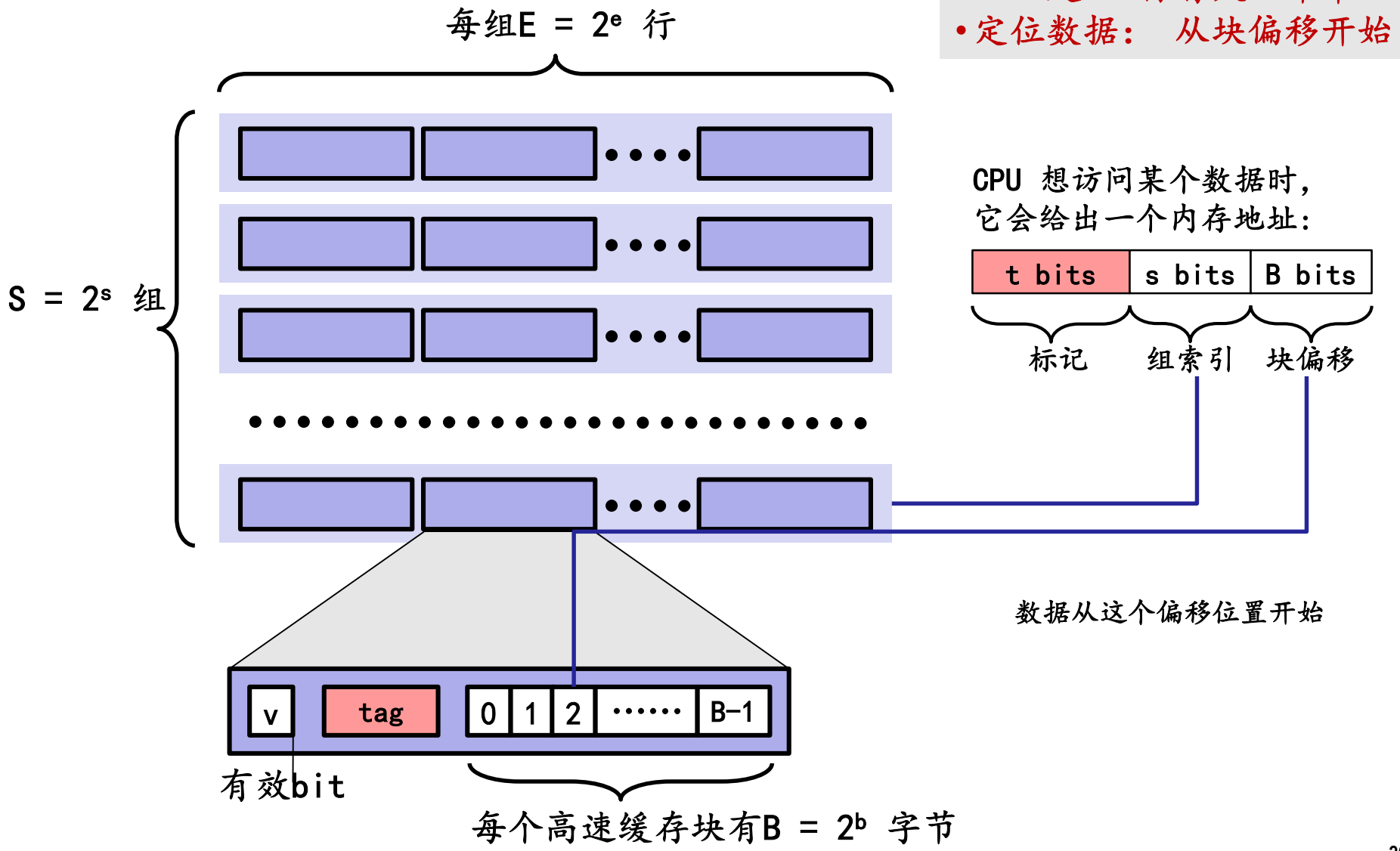
符号	含义	解释
S	组的总数（Number of Sets）	缓存被分成多少组
s	组索引位数（Index Bits）	用于定位缓存组的比特位数
E	每组的行数（Lines per Set）	每组里有多少缓存行（Cache Lines）
e	行索引位数（Line Index Bits）	如果 E>1，用多少比特区分组内行
B	每行数据块大小（Block Size in Bytes）	每行缓存能存多少字节
b	块内偏移位数（Block Offset Bits）	用于确定块内具体字节的比特数

Cache 的三大核心参数：S、E、B

- 每一行缓存包含三个关键部分：
  - 有效位 v
    - 只有当 v=1 时，表示这一行的数据是有效的。
    - 如果 v=0，说明这一行的数据是垃圾，需要去主存取数据。
  - 标记位 tag
    - 标记这行缓存对应的是主存中的哪一个数据块。
    - 当我们找到组后，要通过比较 Tag 判断数据是否匹配。
  - 数据块（B 字节）
    - 真正存储的数据。

读取Cache：可视化

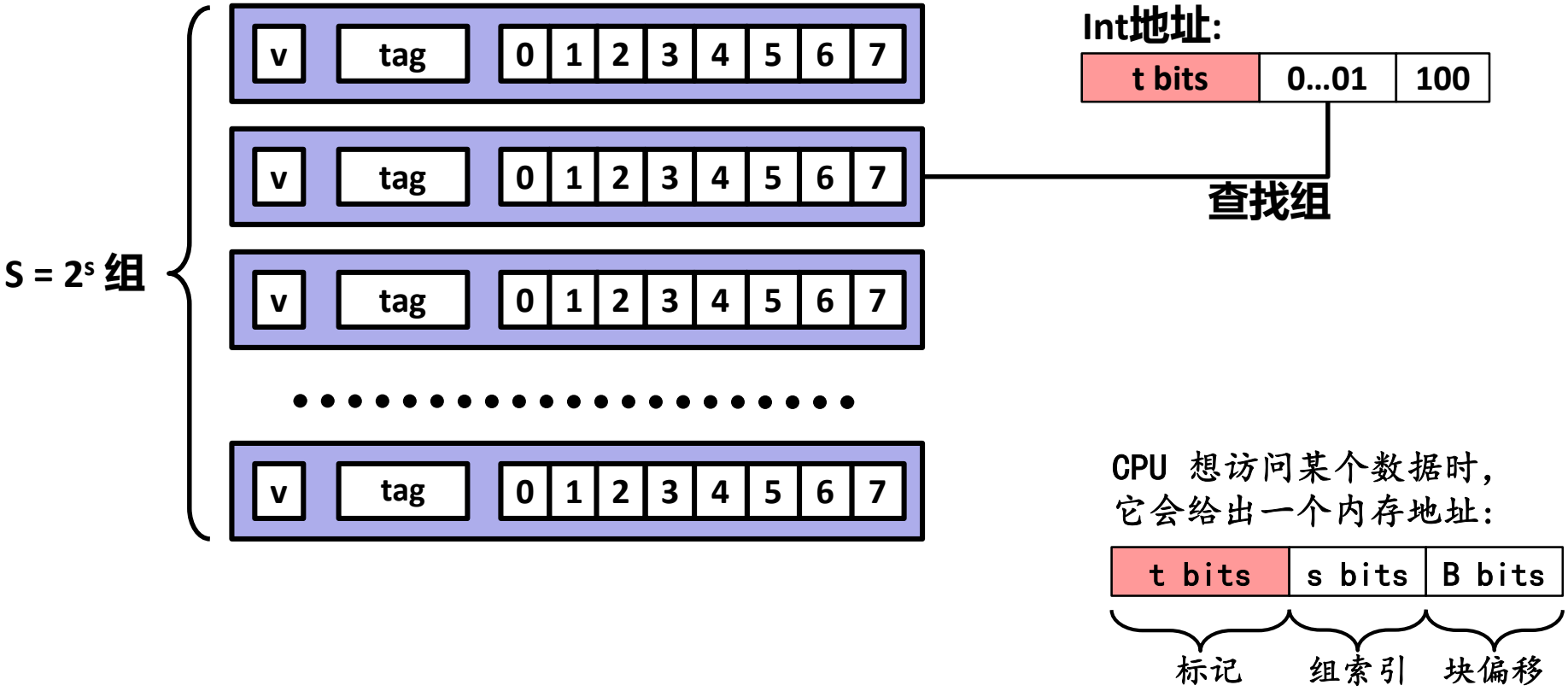
- 定位组
- 定位行
  - 是 + 行有效：命中
- 定位数据：从块偏移开始



例子: 直接映射缓存 (E = 1)

直接映射: 每一组只有一行

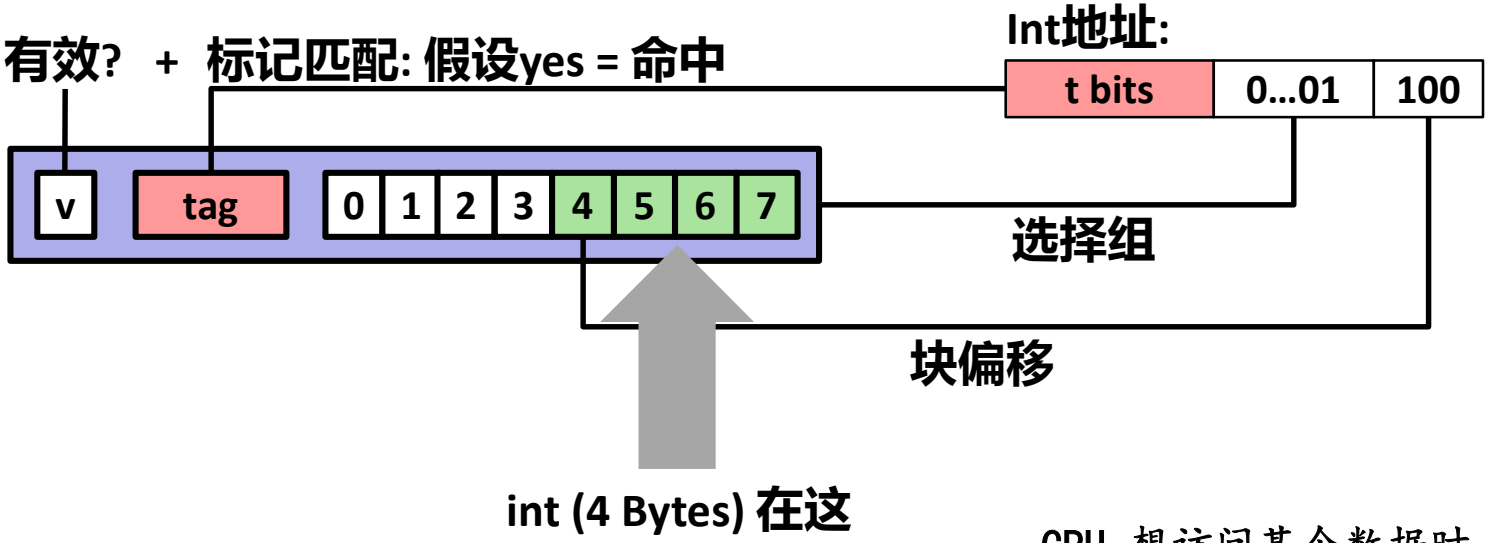
假设: 缓存块大小为8字节:



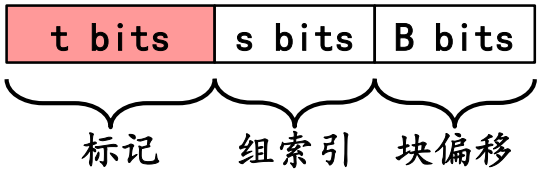
例子: 直接映射高速缓存 (E = 1)

直接映射: 每一组只有一行

假设: 缓存块大小为8字节:



CPU 想访问某个数据时，
它会给出一个内存地址:



如果标记不匹配: 旧的行被驱逐、替换

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 0 | set 00 | off 0]
- 地址1 = 0001_2 → [tag 0 | set 00 | off 1]
- 地址7 = 0111_2 → [tag 0 | set 11 | off 1]
- 地址8 = 1000_2 → [tag 1 | set 00 | off 0]
- 地址0 = 0000_2 → [tag 0 | set 00 | off 0]

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

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 0 | set 00 | off 0] 未命中
- 地址1 = 0001_2 → [tag 0 | set 00 | off 1]
- 地址7 = 0111_2 → [tag 0 | set 11 | off 1]
- 地址8 = 1000_2 → [tag 1 | set 00 | off 0]
- 地址0 = 0000_2 → [tag 0 | set 00 | off 0]

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

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 0 | set 00 | off 0] 未命中
- 地址1 = 0001_2 → [tag 0 | set 00 | off 1]
- 地址7 = 0111_2 → [tag 0 | set 11 | off 1]
- 地址8 = 1000_2 → [tag 1 | set 00 | off 0]
- 地址0 = 0000_2 → [tag 0 | set 00 | off 0]

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

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 0 | set 00 | off 0] 未命中
- 地址1 = 0001_2 → [tag 0 | set 00 | off 1] 命中
- 地址7 = 0111_2 → [tag 0 | set 11 | off 1]
- 地址8 = 1000_2 → [tag 1 | set 00 | off 0]
- 地址0 = 0000_2 → [tag 0 | set 00 | off 0]

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

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 0 | set 00 | off 0] 未命中
- 地址1 = 0001_2 → [tag 0 | set 00 | off 1] 命中
- 地址7 = 0111_2 → [tag 0 | set 11 | off 1] 未命中
- 地址8 = 1000_2 → [tag 1 | set 00 | off 0]
- 地址0 = 0000_2 → [tag 0 | set 00 | off 0]

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

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 0 | set 00 | off 0] 未命中
- 地址1 = 0001_2 → [tag 0 | set 00 | off 1] 命中
- 地址7 = 0111_2 → [tag 0 | set 11 | off 1] 未命中
- 地址8 = 1000_2 → [tag 1 | set 00 | off 0]
- 地址0 = 0000_2 → [tag 0 | set 00 | off 0]

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

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 0 | set 00 | off 0] 未命中
- 地址1 = 0001_2 → [tag 0 | set 00 | off 1] 命中
- 地址7 = 0111_2 → [tag 0 | set 11 | off 1] 未命中
- 地址8 = 1000_2 → [tag 1 | set 00 | off 0] 未命中
- 地址0 = 0000_2 → [tag 0 | set 00 | off 0]

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

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 0 | set 00 | off 0] 未命中
- 地址1 = 0001_2 → [tag 0 | set 00 | off 1] 命中
- 地址7 = 0111_2 → [tag 0 | set 11 | off 1] 未命中
- 地址8 = 1000_2 → [tag 1 | set 00 | off 0] 未命中
- 地址0 = 0000_2 → [tag 0 | set 00 | off 0]

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

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 0 | set 00 | off 0] 未命中
- 地址1 = 0001_2 → [tag 0 | set 00 | off 1] 命中
- 地址7 = 0111_2 → [tag 0 | set 11 | off 1] 未命中
- 地址8 = 1000_2 → [tag 1 | set 00 | off 0] 未命中
- 地址0 = 0000_2 → [tag 0 | set 00 | off 0]

	v	Tag	Block
Set 0	1	1	M[8-9]
Set 1	0	?	?
Set 2	0	?	?
Set 3	1	0	M[6-7]

直接映射高速缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 2 位）
t:（标记 1 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 4 个 set
  - E=1 块/组 ⇒ 每组 1 行（直接映射）

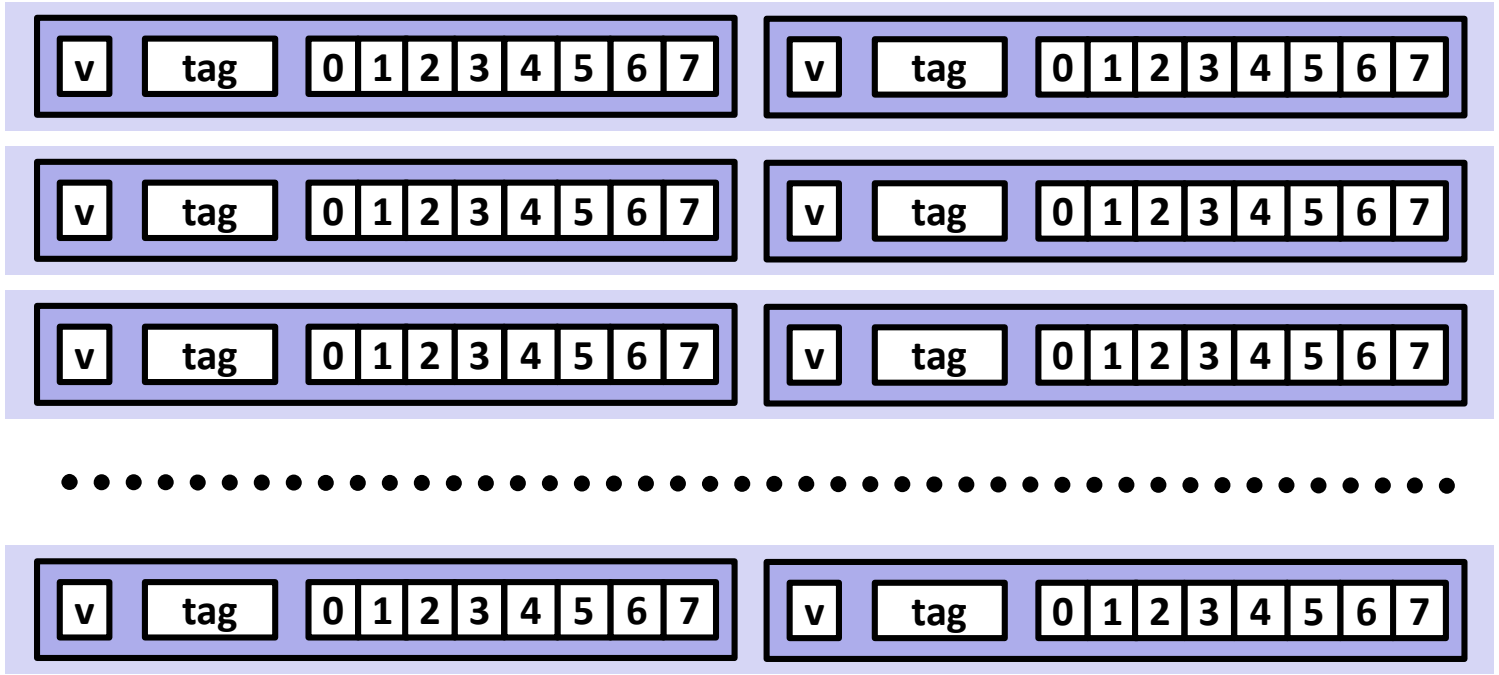
假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 0 | set 00 | off 0] 未命中
- 地址1 = 0001_2 → [tag 0 | set 00 | off 1] 命中
- 地址7 = 0111_2 → [tag 0 | set 11 | off 1] 未命中
- 地址8 = 1000_2 → [tag 1 | set 00 | off 0] 未命中
- 地址0 = 0000_2 → [tag 0 | set 00 | off 0] 未命中

	v	Tag	Block
Set 0	1	1	M[8-9]
Set 1	0	?	?
Set 2	0	?	?
Set 3	1	0	M[6-7]

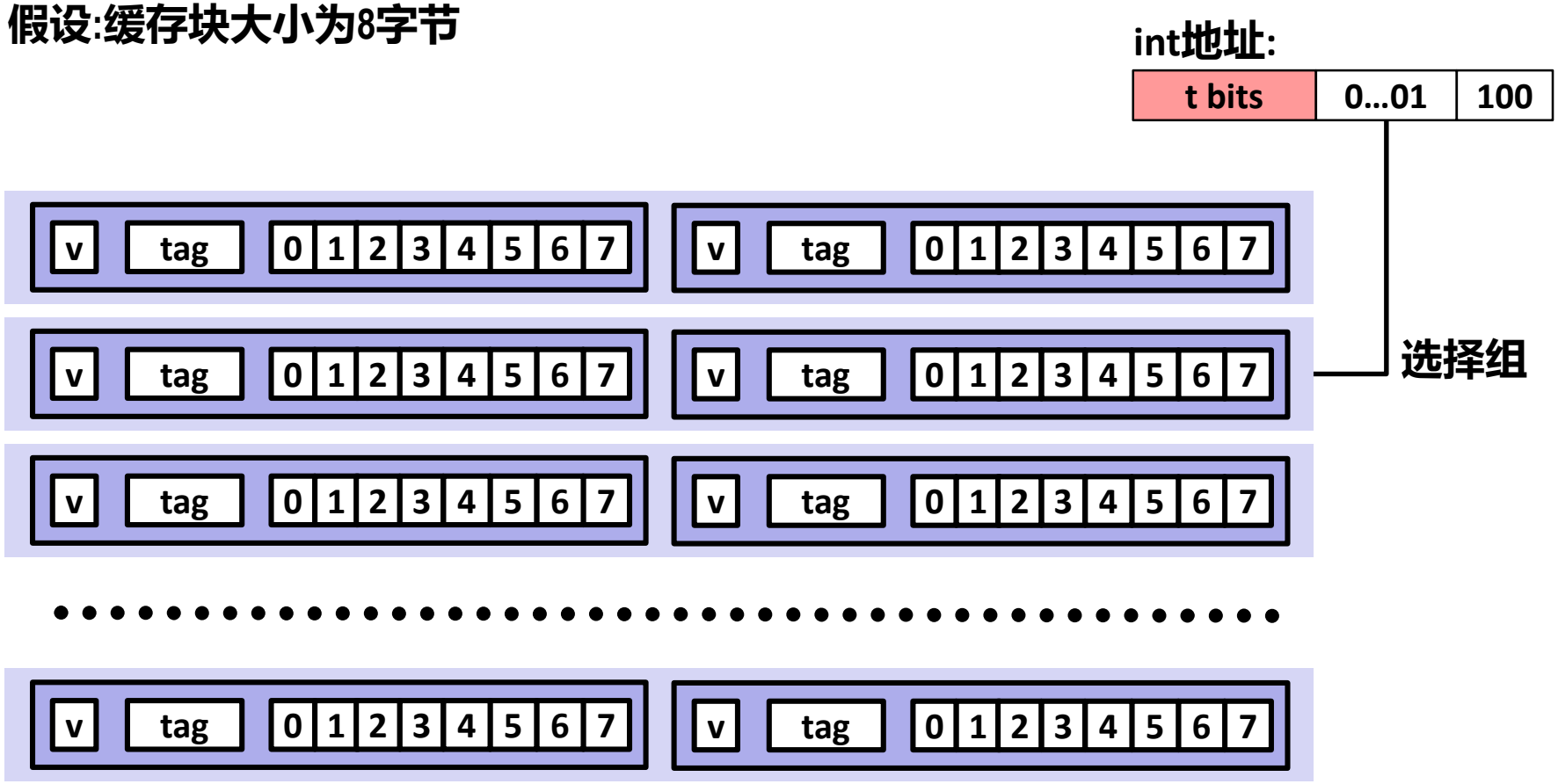
E-路组相联高速缓存 (E = 2)

E = 2: 每组两行
假设:缓存块大小为8字节



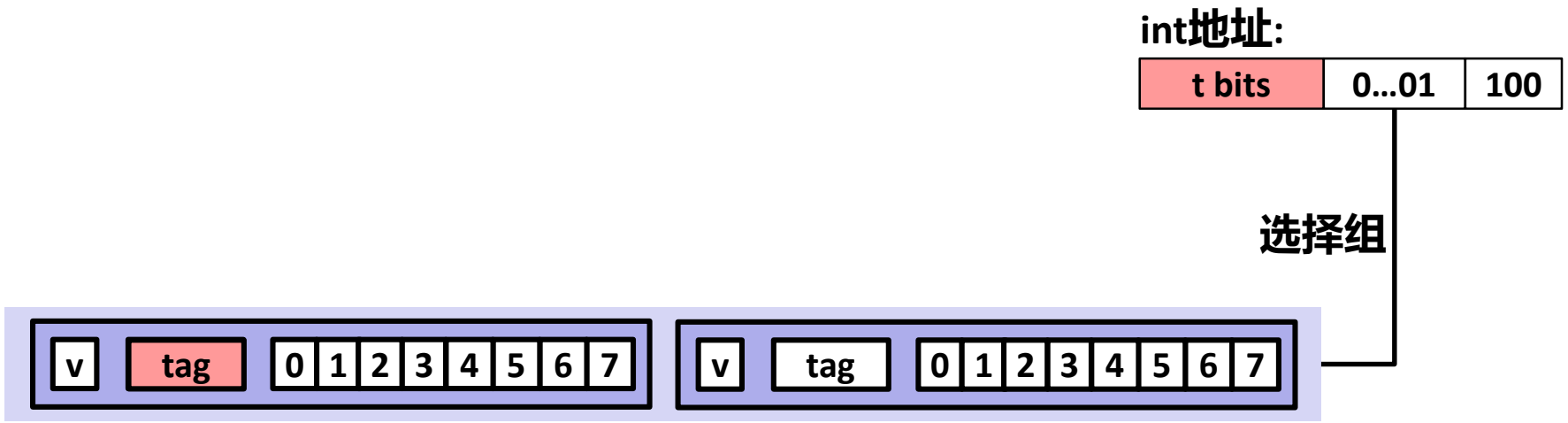
E-路组相联高速缓存 (E = 2)

E = 2: 每组两行
假设:缓存块大小为8字节



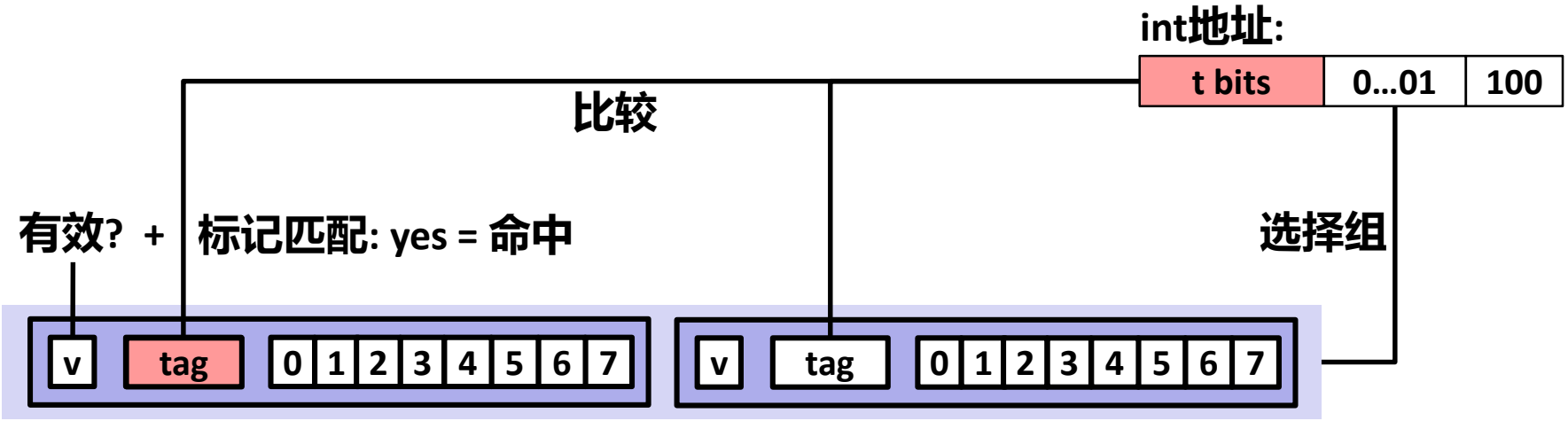
E-路组相联高速缓存 (E = 2)

E = 2: 每组两行
假设:缓存块大小为8字节



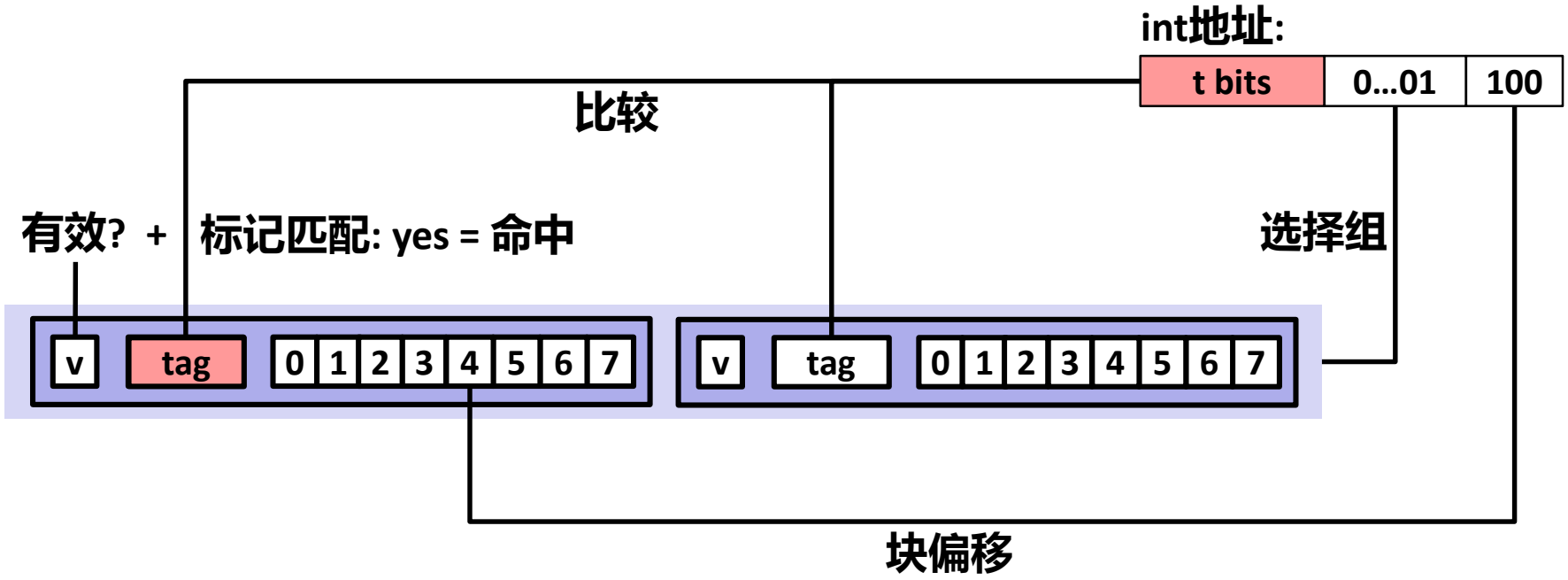
E-路组相联高速缓存 (E = 2)

E = 2: 每组两行
假设:缓存块大小为8字节



E-路组相联高速缓存 (E = 2)

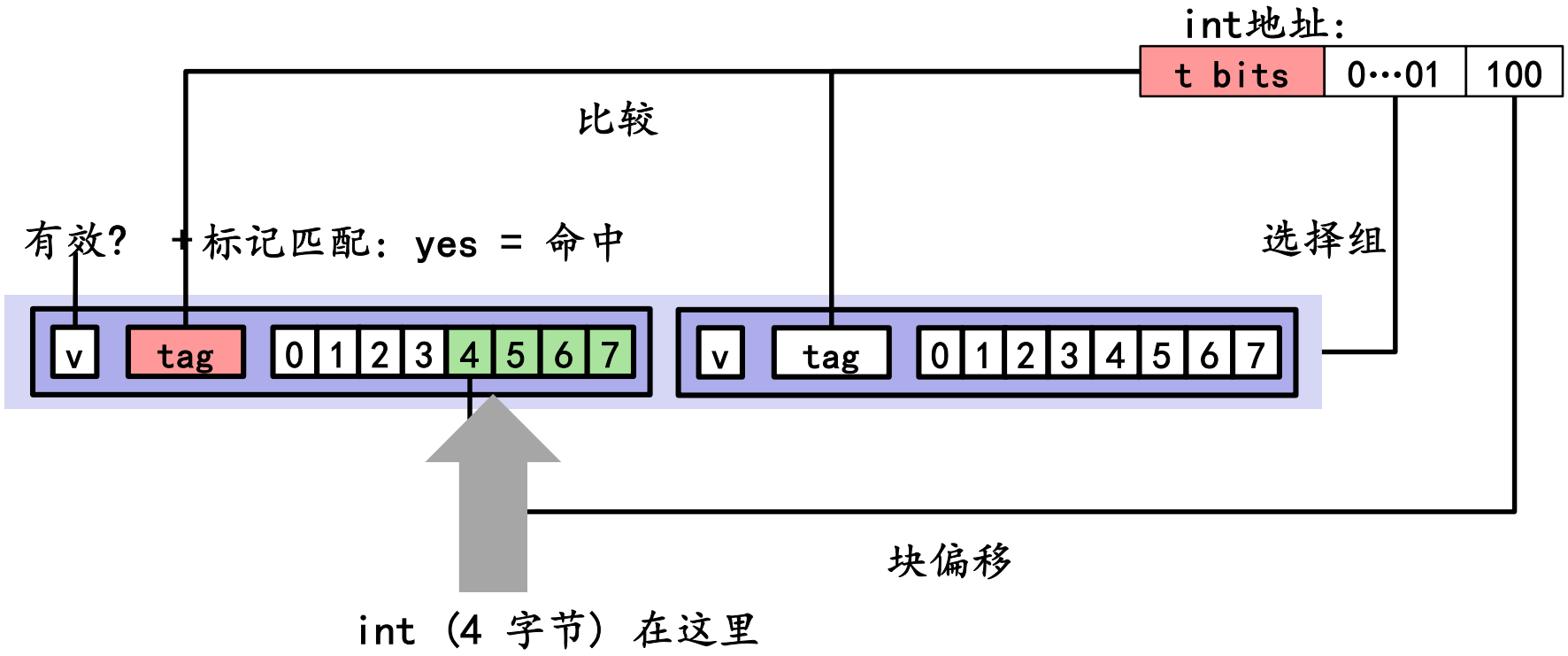
E = 2: 每组两行
假设:缓存块大小为8字节



E-路组相联高速缓存 (E = 2)

E = 2: 每组两行

假设:缓存块大小为8字节



如果不匹配:

- 在组中选择1行用于驱逐和替换
- 替换策略: 随机、最近使用(LRU), …

2-路组相联缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 1 位）
t:（标记 2 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)，2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 2 个 set
  - E=2 块/组 ⇒ 每组 2 行

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 00 | set 0 | off 0]
- 地址1 = 0001_2 → [tag 00 | set 0 | off 1]
- 地址7 = 0111_2 → [tag 01 | set 1 | off 1]
- 地址8 = 1000_2 → [tag 10 | set 0 | off 0]
- 地址0 = 0000_2 → [tag 00 | set 0 | off 0]

	v	Tag	Block
Set 0	0	?	?
	0	?	?
Set 1	0	?	?
	0	?	?

2-路组相联缓存模拟

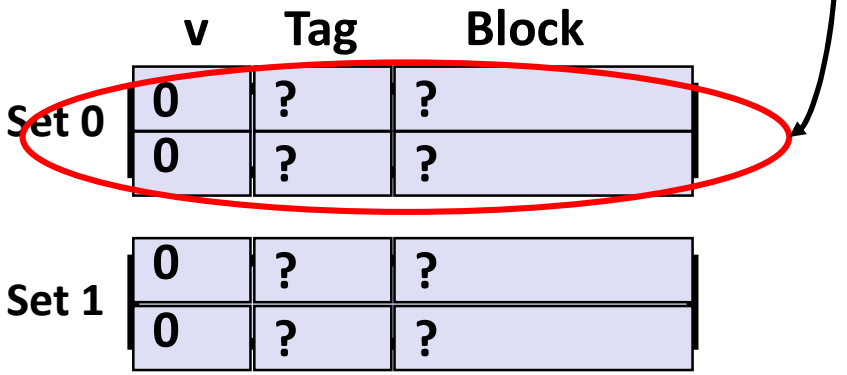
t=2	s=1	b=1
xx	xx	x

b:（块内偏移 1 位）
s:（组索引 1 位）
t:（标记 2 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)， 2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 2 个 set
  - E=2 块/组 ⇒ 每组 2 行

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 未命中
- 地址1 = 0001_2 → [tag 00 | set 0 | off 1]
- 地址7 = 0111_2 → [tag 01 | set 1 | off 1]
- 地址8 = 1000_2 → [tag 10 | set 0 | off 0]
- 地址0 = 0000_2 → [tag 00 | set 0 | off 0]



2-路组相联缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 1 位）
t:（标记 2 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)，2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 2 个 set
  - E=2 块/组 ⇒ 每组 2 行

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 未命中
- 地址1 = 0001_2 → [tag 00 | set 0 | off 1] 命中
- 地址7 = 0111_2 → [tag 01 | set 1 | off 1] 未命中
- 地址8 = 1000_2 → [tag 10 | set 0 | off 0] 未命中
- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 命中

	v	Tag	Block
Set 0	1	00	M[0-1]
	0	?	?
Set 1	0	?	?
	0	?	?

2-路组相联缓存模拟

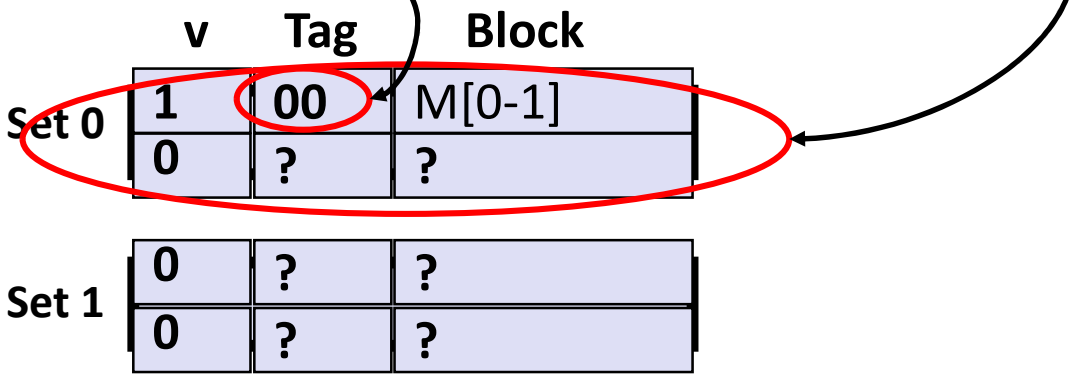
t=2	s=1	b=1
xx	xx	x

b:（块内偏移 1 位）
s:（组索引 1 位）
t:（标记 2 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)，2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 2 个 set
  - E=2 块/组 ⇒ 每组 2 行

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 未命中
- 地址1 = 0001_2 → [tag 00 | set 0 | off 1] 命中
- 地址7 = 0111_2 → [tag 01 | set 1 | off 1] 未命中
- 地址8 = 1000_2 → [tag 10 | set 0 | off 0] 未命中
- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 命中



2-路组相联缓存模拟

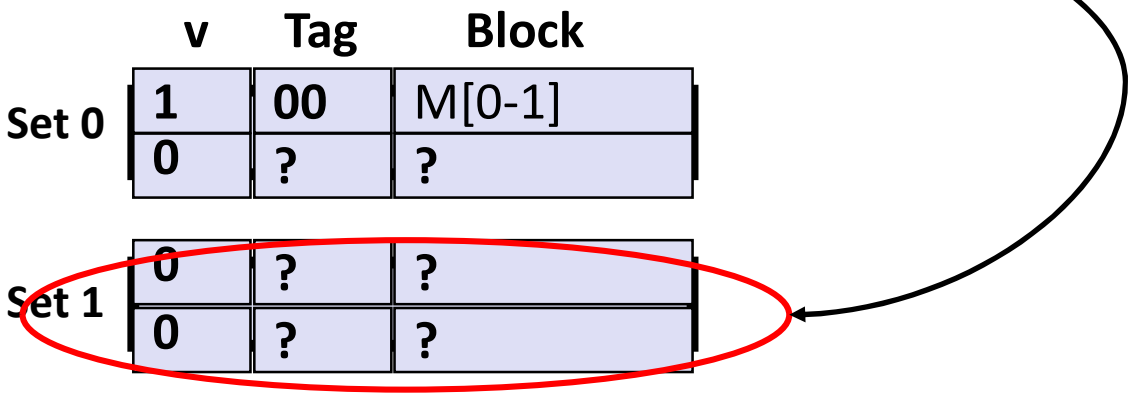
t=2	s=1	b=1
xx	xx	x

b:（块内偏移 1 位）
s:（组索引 1 位）
t:（标记 2 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)，2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 2 个 set
  - E=2 块/组 ⇒ 每组 2 行

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 未命中
- 地址1 = 0001_2 → [tag 00 | set 0 | off 1] 命中
- 地址7 = 0111_2 → [tag 01 | set 1 | off 1] 未命中
- 地址8 = 1000_2 → [tag 10 | set 0 | off 0] 未命中
- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 命中



2-路组相联缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 1 位）
t:（标记 2 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)，2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 2 个 set
  - E=2 块/组 ⇒ 每组 2 行

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 未命中
- 地址1 = 0001_2 → [tag 00 | set 0 | off 1] 命中
- 地址7 = 0111_2 → [tag 01 | set 1 | off 1] 未命中
- 地址8 = 1000_2 → [tag 10 | set 0 | off 0] 未命中
- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 命中

	v	Tag	Block
Set 0	1	00	M[0-1]
	0	?	?
Set 1	1	01	M[6-7]
	0	?	?

2-路组相联缓存模拟

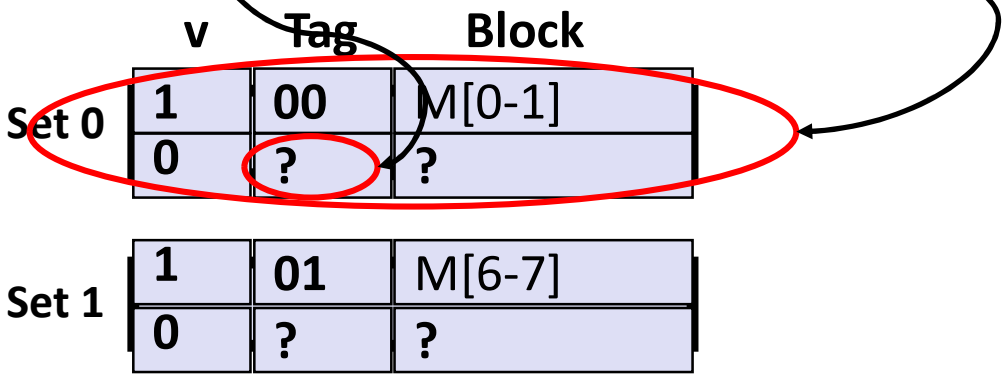
t=2	s=1	b=1
xx	xx	x

b:（块内偏移 1 位）
s:（组索引 1 位）
t:（标记 2 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)，2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 2 个 set
  - E=2 块/组 ⇒ 每组 2 行

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 未命中
- 地址1 = 0001_2 → [tag 00 | set 0 | off 1] 命中
- 地址7 = 0111_2 → [tag 01 | set 1 | off 1] 未命中
- 地址8 = 1000_2 → [tag 10 | set 0 | off 0] 未命中
- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 命中



2-路组相联缓存模拟

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

b:（块内偏移 1 位）
s:（组索引 1 位）
t:（标记 2 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)，2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 2 个 set
  - E=2 块/组 ⇒ 每组 2 行

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 未命中
- 地址1 = 0001_2 → [tag 00 | set 0 | off 1] 命中
- 地址7 = 0111_2 → [tag 01 | set 1 | off 1] 未命中
- 地址8 = 1000_2 → [tag 10 | set 0 | off 0] 未命中
- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 命中

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

2-路组相联缓存模拟

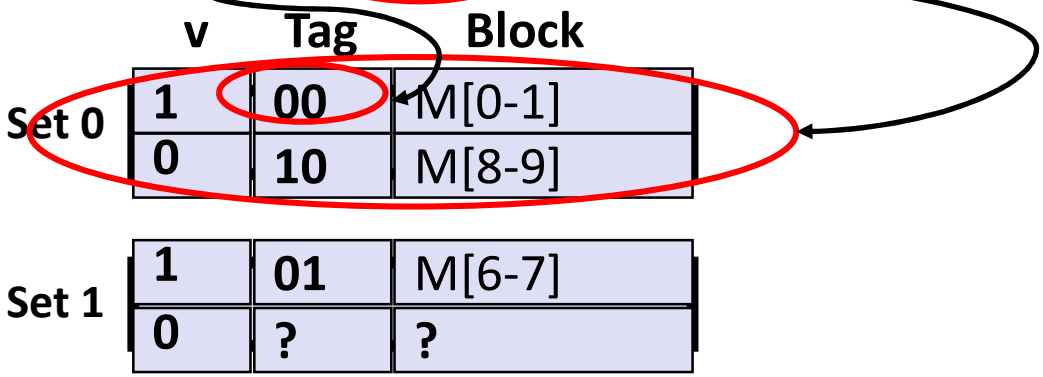
t=2	s=1	b=1
xx	xx	x

b:（块内偏移 1 位）
s:（组索引 1 位）
t:（标记 2 位）

- 假设主存容量M=16 字节 (4-位地址,地址从 0-15)，2 字节/块
- 假设缓存容量为8 字节
  - B=2 字节/块 ⇒ 每行存 2 字节(一次从内存取2字节)
  - S=4 组 ⇒ 有 2 个 set
  - E=2 块/组 ⇒ 每组 2 行

假设CPU开始尝试顺序读取以下内存地址数据：

- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 未命中
- 地址1 = 0001_2 → [tag 00 | set 0 | off 1] 命中
- 地址7 = 0111_2 → [tag 01 | set 1 | off 1] 未命中
- 地址8 = 1000_2 → [tag 10 | set 0 | off 0] 未命中
- 地址0 = 0000_2 → [tag 00 | set 0 | off 0] 命中



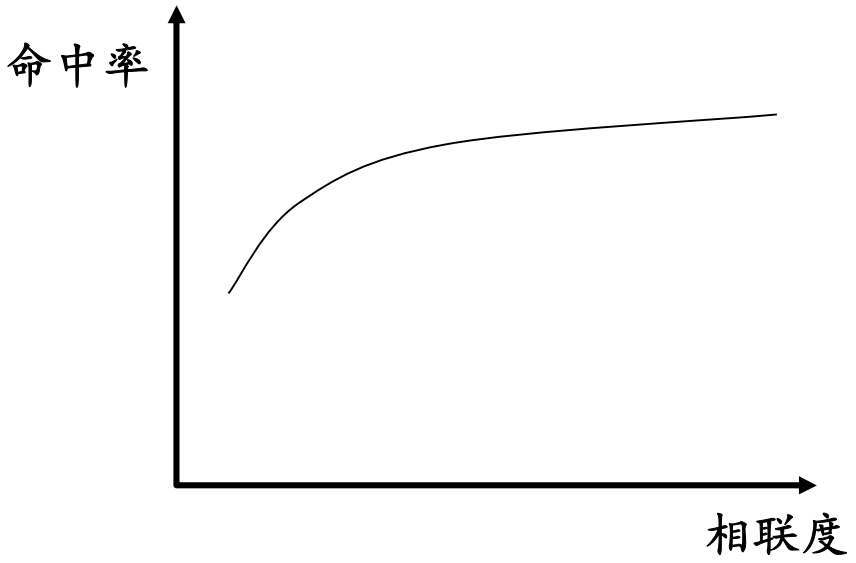
Cache的种类

- **直接映射缓存（Direct-Mapped Cache）：E = 1**
  - 主存中的一个块只能映射到 Cache 中的某一个**特定块**。
  - 类比：就像宿舍床位固定分配，每个人只能睡自己的床位。如果别人偶尔需要住一晚，就必须赶走你 → 容易打架。
- **全相联映射缓存（Fully Associative Cache）：S = 1**
  - 主存中的任何一块数据都可以映射到 Cache 中的**任意一块**。
  - 类比：就像宿舍没有床位固定分配，大家可以随便睡哪张床。虽然灵活，但每次都得走遍整个宿舍找空床 → 成本高。
- **组相联映射缓存（Set-Associative Cache）：E > 1 且 S > 1**
  - 是前两种方法的折中方案。
  - **兼顾两者优点，尽量避免两者缺点**，因此在现代计算机中最常用。
  - 类比：宿舍被分成多个寝室（组），每个寝室里有多张床（行）。学号告诉你住哪个寝室，但寝室里的床可以灵活选择 → 冲突少，灵活高效。

相联度（Associativity）及其权衡

- **相联度的定义：**
  - 指有多少个主存块可以映射到同一个缓存索引（或组）。
- **更高的相联度：**
  - 优点：缓存命中率更高（Higher hit rate）
  - 缺点：
    - 缓存访问时间变慢（包括命中延迟和数据访问延迟）
    - 硬件成本更高（需要更多比较器 Comparator）
- **高相联度的边际收益递减：**
  - 当相联度不断提高时，

命中率提升越来越有限。



组相联缓存中的问题

- 在组相联缓存中，可以认为每个缓存块都有一个“优先级”
  - 表示在缓存中保留该块的重要程度
- 核心问题：如何确定或调整这些缓存块的优先级？
- 在一组（Set）中有三个关键决策：
  - 插入（Insertion）提升（Promotion）驱逐/替换（Replacement）
- 插入：当缓存填充新块时，优先级如何变化？
  - 插入新块的位置在哪里？
  - 是否需要插入该块？
- 提升：当缓存命中（Cache Hit）时，优先级如何变化？
  - 是否需要调整该块的优先级？
  - 如果需要，怎么调整？
- 驱逐/替换：当缓存未命中（Cache Miss）时，优先级如何变化？
  - 选择哪个块被驱逐？
  - 驱逐后如何重新调整组内块的优先级？

缓存的驱逐 / 替换策略

- 缓存未命中（cache miss）时：应该替换同一组中的哪一块？
  - 优先替换无效块（invalid block）。
  - 如果所有块都有效，则遵循替换策略（replacement policy）。
- 常见替换策略：
  - 随机替换（Random）
  - 先进先出（FIFO, First In First Out）
  - 最近最少使用（LRU, Least Recently Used）
  - 非最近使用（NRU, Not Most Recently Used）
  - 最少使用次数（LFU, Least Frequently Used）
  - 最优替换策略（Optimal replacement policy） —— 理论上最优，但实现困难
  - 。。。。。。

LRU（Least Recently Used）

- 实现 LRU（Least Recently Used，最近最少使用）
  - 核心思想：驱逐最近最久未被访问的缓存块
  - 实现问题：需要记录每个缓存块的访问顺序
  - 问题 1：两路组相联缓存（2-way set associative cache）
    - 如何才能**完美实现 LRU**?
  - 问题 2：四路组相联缓存（4-way set associative cache）
    - 如何才能**完美实现 LRU**?
    - 对于 4 个块，一共有多少种可能的访问顺序？
    - 需要多少位（bits）才能编码每个块的 LRU 顺序？
    - 如何设计逻辑电路来确定 LRU 替换的“牺牲者”（victim）？

LRU（Least Recently Used）

- 实现 LRU（Least Recently Used，最近最少使用）
  - 核心思想：驱逐最近最久未被访问的缓存块
  - 实现问题：需要记录每个缓存块的访问顺序
  - 问题 1：两路组相联缓存（2-way set associative cache）
    - 如何才能完美实现 LRU?
    - 每组只需 1 位状态位：访问某一行时，把这 1 位设置为“另一行是最久未用”。当需要驱逐时，直接看这 1 位，驱逐标记的那一行。
  - 问题 2：四路组相联缓存（4-way set associative cache）
    - 如何才能完美实现 LRU?
      - 每次缓存命中或插入新块时，都要更新 4 行的“最近使用顺序”。
    - 对于 4 个块，一共有多少种可能的访问顺序？
      - 对 4 个块，可能的访问顺序有 4! = 24 种。
    - 需要多少位（bits）才能编码每个块的 LRU 顺序？
      - 为了区分 24 种顺序，至少需要 $\lceil \log_2(24) \rceil = 5$ 位状态信息。

LRU 的近似实现

- 大多数现代处理器在高相联缓存（highly-associative caches）中并不会实现真正的 LRU（也称“完美 LRU”）。
- 原因：
  - 完美 LRU 实现非常复杂
  - LRU 本身就是一种近似策略，用于预测数据局部性，本质上并不是最优的缓存管理策略
- 常见的 LRU 近似实现方法：
  - Not MRU（非最近使用）：替换掉不是最近访问过的块
  - 层次化 LRU（Hierarchical LRU）：
    - 把 4 路组相联缓存划分为两个 2 路“小组”
    - 分别跟踪最常用组（MRU group）以及组内的 MRU 行
  - Victim-NextVictim 替换策略：
    - 只记录当前被替换的块（victim）和下一个要被替换的块（next victim）

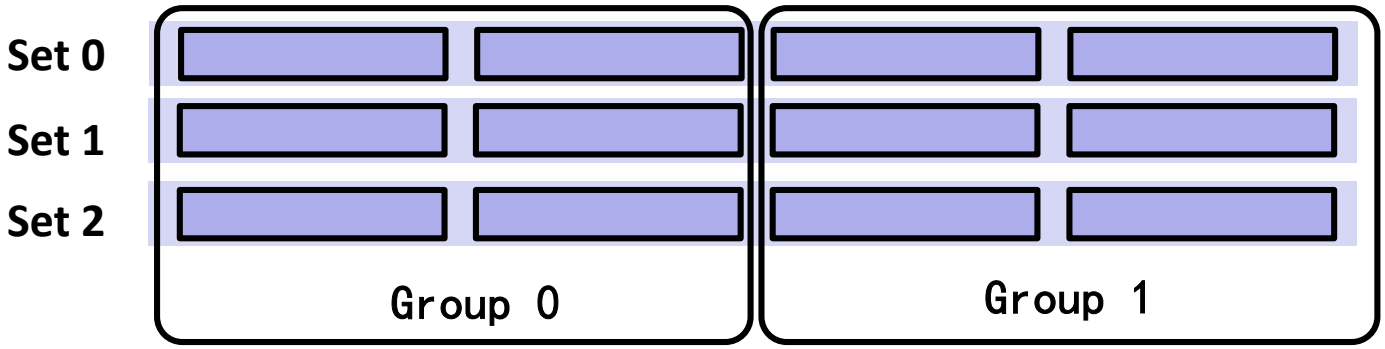
层次化LRU（Hierarchical LRU, not MRU）

- 将一个组（set）划分为多个小组（groups）
- 只跟踪最近最常用的小组（MRU group）
- 只跟踪每个小组内的最近最常用块（MRU block）

- 当需要替换缓存块时：
  - 在非 MRU 小组中，选择一个非 MRU 块作为替换目标
  - 如果有多个可选，随机选一个块或小组进行替换

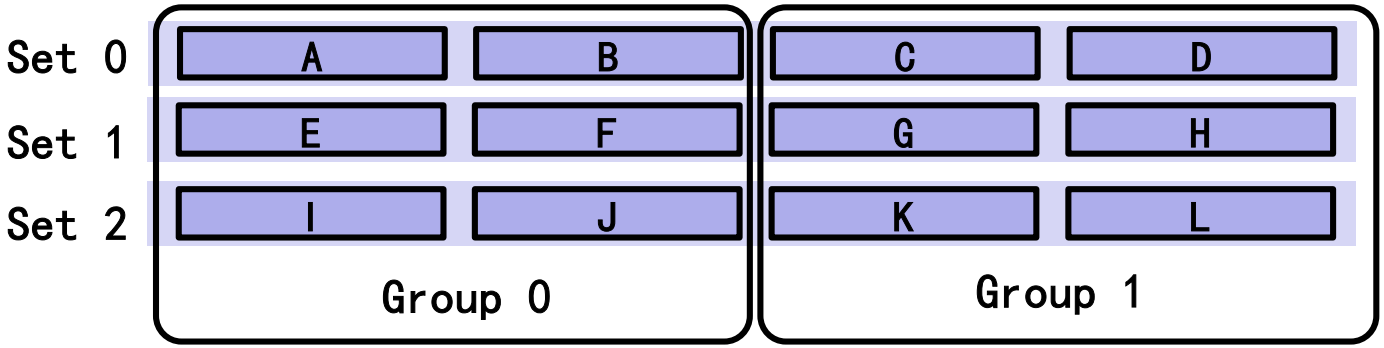
层次化LRU例子

- 4 路组相联缓存（4-way cache）
  - 每组只需 2 位状态位来记录替换信息：
    - 其中 1 位用于标记 MRU 小组（最近最常用的小组）
    - 另一位用于标记该小组内的 MRU 块（最近最常用的块）
  - 替换策略：
    - 首先判断：当前小组是不是 MRU 小组？
    - 再判断：在小组内部，这个块是不是 MRU 块？
    - Victim（被替换者）：选择既不是 MRU 小组、又不是 MRU 块的缓存块。



层次化LRU例子

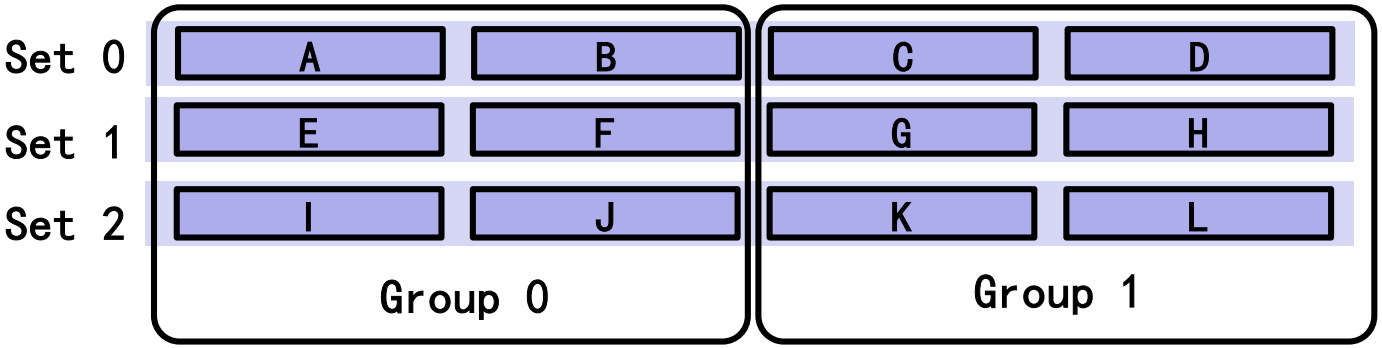
- 4路组相联使用层次化LRU
- 每组有4行，每组再分成两个2个小组(Group)
- 每组有：
  - 1 位组间 MRU → 记录哪个小组最近被访问
  - 每小组 1 位组内 MRU → 记录小组内哪个块最近使用
  - 初始状态如下
  - MRU 状态随访问更新访问序列：B → D → G → K → X（X 是新数据）



组间 MRU 位
Set 0 = 0 Set 1 = 0
Set 2 = 0
组内 MRU 位
[A=0, B=0] [C=0, D=0]
[E=0, F=0] [G=0, H=0]
[I=0, J=0] [K=0, L=0]

层次化LRU例子

- 4路组相联使用层次化LRU
- 每组有4行，每组再分成两个2个小组(Group)
- 每组有：
  - 1 位组间 MRU → 记录哪个小组最近被访问
  - 每小组 1 位组内 MRU → 记录小组内哪个块最近使用
  - 初始状态：所有块有效
  - MRU 状态随访问更新访问序列：B → D → G → K → X（X 是新数据）

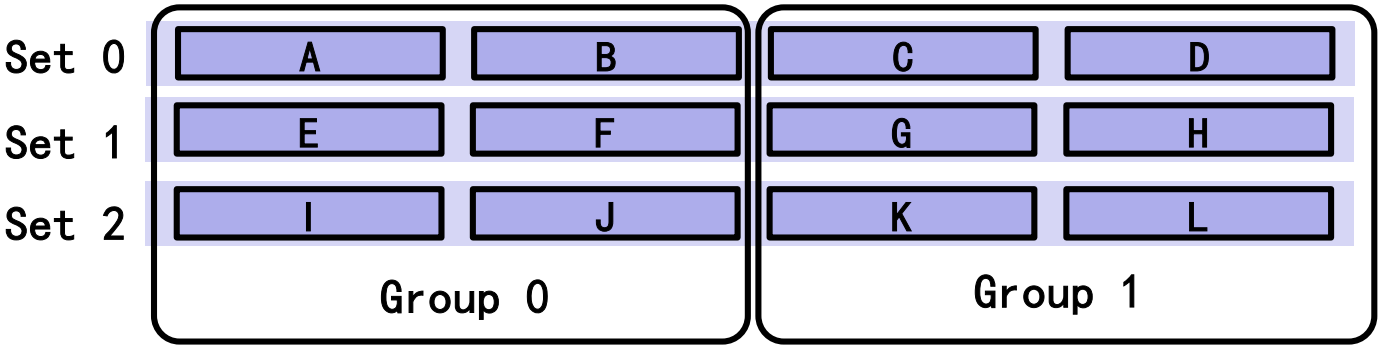


组间 MRU 位
Set 0 = 0 Set 1 = 0
Set 2 = 0
组内 MRU 位
[A=0,B=1] [C=0,D=0]
[E=0,F=0] [G=0,H=0]
[I=0,J=0] [K=0,L=0]

1.访问B,命中,属于组0，组间MRU不变,更新组内MRU B=1

层次化LRU例子

- 4路组相联使用层次化LRU
- 每组有4行，每组再分成两个2个小组(Group)
- 每组有：
  - 1 位组间 MRU → 记录哪个小组最近被访问
  - 每小组 1 位组内 MRU → 记录小组内哪个块最近使用
  - 初始状态：所有块有效
  - MRU 状态随访问更新访问序列：B → D → G → K → X（X 是新数据）

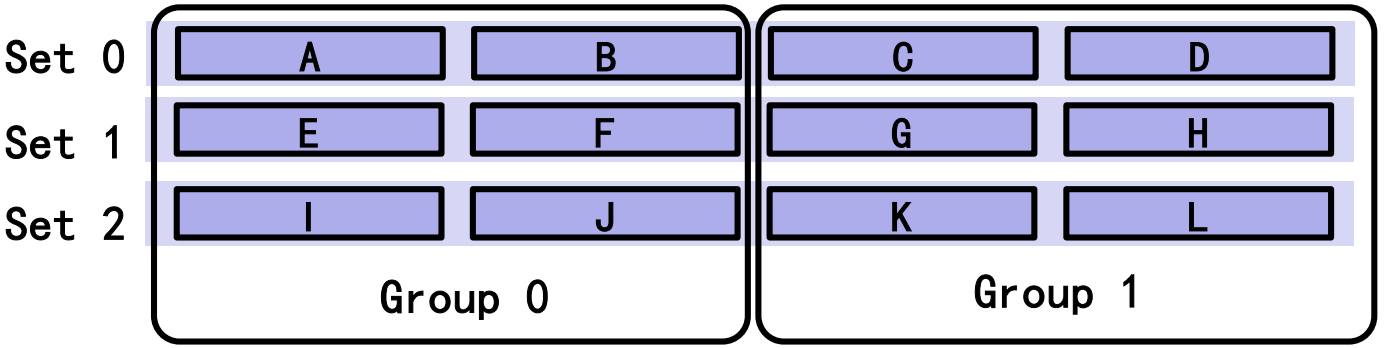


组间 MRU 位
Set 0 = 1 Set 1 = 0
Set 2 = 0
组内 MRU 位
[A=0,B=1] [C=0,D=1]
[E=0,F=0] [G=0,H=0]
[I=0,J=0] [K=0,L=0]

2.访问D,命中,属于组1，更新组间MRU Set 0 = 1,更新组内MRU D=1

层次化LRU例子

- 4路组相联使用层次化LRU
- 每组有4行，每组再分成两个2个小组(Group)
- 每组有：
  - 1 位组间 MRU → 记录哪个小组最近被访问
  - 每小组 1 位组内 MRU → 记录小组内哪个块最近使用
  - 初始状态：所有块有效
  - MRU 状态随访问更新访问序列：B → D → G → K → X（X 是新数据）

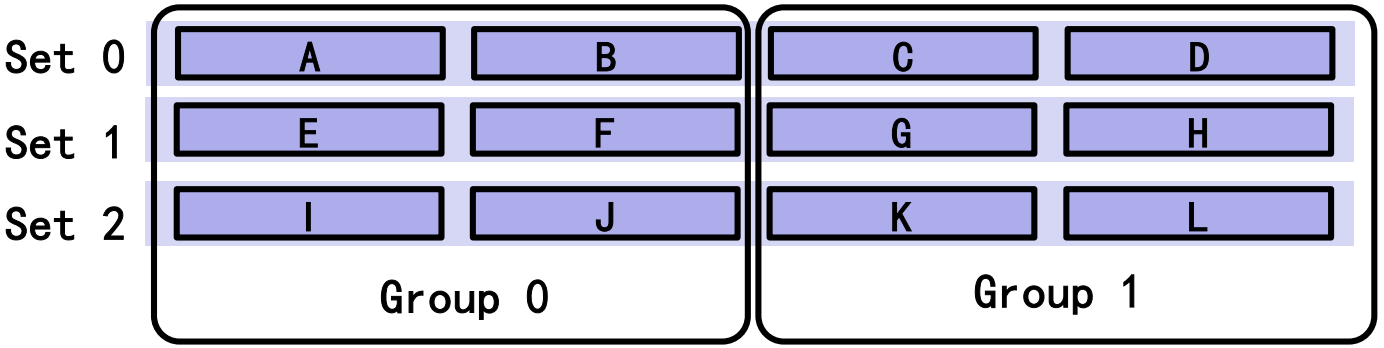


组间 MRU 位
Set 0 = 1 Set 1 = 1
Set 2 = 0
组内 MRU 位
[A=0,B=1] [C=0,D=1]
[E=0,F=0] [G=1,H=0]
[I=0,J=0] [K=0,L=0]

3.访问G,命中,属于组1，组间MRU Set 1 = 1,更新组内MRU G = 1

层次化LRU例子

- 4路组相联使用层次化LRU
- 每组有4行，每组再分成两个2个小组(Group)
- 每组有：
  - 1 位组间 MRU → 记录哪个小组最近被访问
  - 每小组 1 位组内 MRU → 记录小组内哪个块最近使用
  - 初始状态：所有块有效
  - MRU 状态随访问更新访问序列：B → D → G → K → X（X 是新数据）

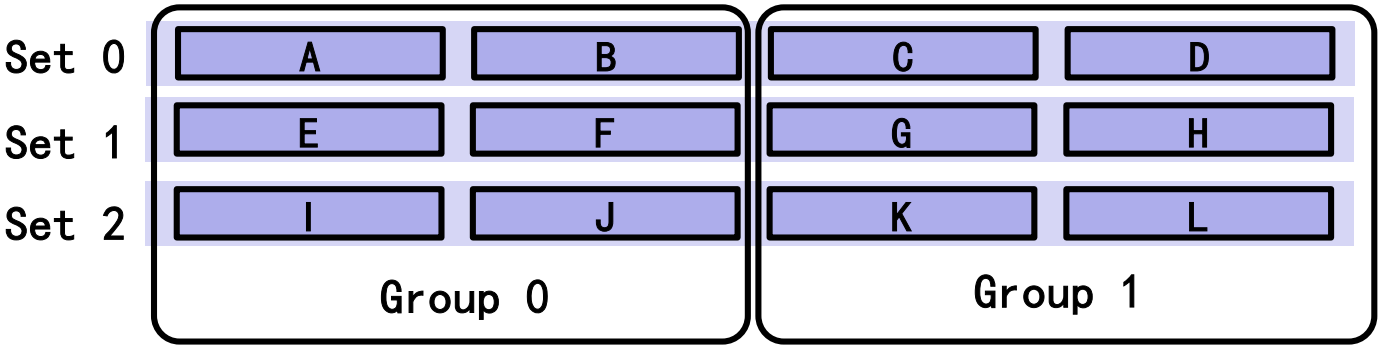


组间 MRU 位
Set 0 = 1 Set 1 = 1
Set 2 = 1
组内 MRU 位
[A=0,B=1] [C=0,D=1]
[E=0,F=0] [G=1,H=0]
[I=0,J=0] [K=1,L=0]

3.访问K,命中,属于组1，组间MRU Set 2 = 1,更新组内MRU K=1

层次化LRU例子

- 4路组相联使用层次化LRU
- 每组有4行，每组再分成两个2个小组(Group)
- 每组有：
  - 1 位组间 MRU → 记录哪个小组最近被访问
  - 每小组 1 位组内 MRU → 记录小组内哪个块最近使用
  - 初始状态：所有块有效
  - MRU 状态随访问更新访问序列：B → D → G → K → X（X 是新数据）

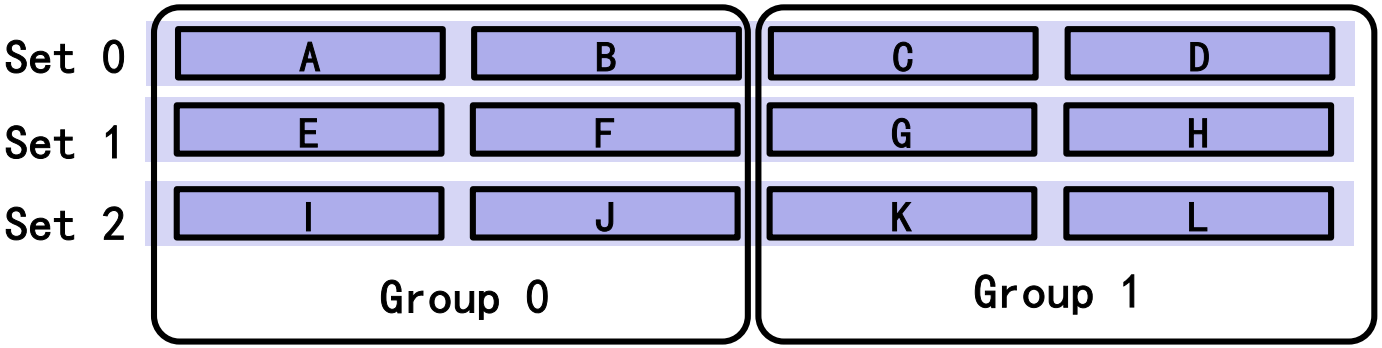


组间 MRU 位
Set 0 = 1 Set 1 = 1
Set 2 = 1
组内 MRU 位
[A=0,B=1] [C=0,D=1]
[E=0,F=0] [G=1,H=0]
[I=0,J=0] [K=1,L=0]

3.访问M,假设 M 的地址映射到 Set2，未命中,首先跳过 MRU 小组，在非 MRU 小组中选择victim，I 或 J（随机选一个）

层次化LRU例子

- 4路组相联使用层次化LRU
- 每组有4行，每组再分成两个2个小组(Group)
- 每组有：
  - 1 位组间 MRU → 记录哪个小组最近被访问
  - 每小组 1 位组内 MRU → 记录小组内哪个块最近使用
  - 初始状态：所有块有效
  - MRU 状态随访问更新访问序列：B → D → G → K → X（X 是新数据）



组间 MRU 位
Set 0 = 1 Set 1 = 1
Set 2 = 0
组内 MRU 位
[A=0,B=1] [C=0,D=1]
[E=0,F=0] [G=1,H=0]
[I=0,J=1] [K=1,L=0]

3.访问M,假设 M 的地址映射到 Set2，未命中,首先跳过 MRU 小组，在非 MRU 小组中选择victim，I 或 J（随机选一个）

Victim / Next-Victim 策略

- 每个 set 只跟踪 2 个块的状态：
  - Victim (V)：当前候选的被替换块
  - Next Victim (NV)：下一个可能被替换的块
  - 其他所有块统一标记为 O（普通块，Ordinary block）

- 在缓存未命中（Cache Miss）时：
  - 用新数据替换 V，新数据直接放到当前的 Victim 位置上。
  - 把 NV 降级为 V
  - 在剩余的普通块（O）中随机选择一个作为新的 NV

- 在访问命中 V 时：
  - 把 NV 降级为 V
  - 在普通块（O）中随机选择一个作为新的 NV
  - 原本的 V 变成普通块（O）

Victim / Next-Victim 策略

- 如果缓存命中在 NV（Next Victim）上：
  - 在剩余的普通块（O）中随机选一个作为新的 NV
  - 当前的 NV 块被降级为普通块（O）

- 如果缓存命中在 O（普通块）上：
  - 不做任何操作

Victim/Next-Victim策略例子

- 如果缓存命中在 NV（Next Victim）上：
  - 在剩余的普通块（O）中随机选一个作为新的 NV
  - 当前的 NV 块被降级为普通块（O）

- 如果缓存命中在 O（普通块）上：
  - 不做任何操作

Set	块 0	块 1	块 2	块 3	V	NV
Set0	A (O)	B (O)	C (V)	D (NV)	C	D
Set1	E (O)	F (O)	G (V)	H (NV)	G	H
Set2	I (V)	J (NV)	K (O)	L (O)	I	J

- 每个 set 只标记 2 个特殊块（V 和 NV），其余为 O。
- 替换优先顺序：先换 V，如果 V 被命中，更新状态；NV 被命中则重新挑选新 NV；O 命中不更新。

Victim/Next-Victim策略例子

- 假设我们设计一个访问序列：C → D → B → X
- 假设 X 映射到 Set0，且不在缓存中 → 缓存未命中

Set	块 0	块 1	块 2	块 3	V	NV
Set0	A (O)	B (O)	C (V)	D (NV)	C	D
Set1	E (O)	F (O)	G (V)	H (NV)	G	H
Set2	I (V)	J (NV)	K (O)	L (O)	I	J

- 每个 set 只标记 2 个特殊块（V 和 NV），其余为 O。
- 替换优先顺序：先换 V，如果 V 被命中，更新状态；NV 被命中则重新挑选新 NV；O 命中不更新。

Victim/Next-Victim策略例子

- 假设我们设计一个访问序列：C → D → B → X
- 假设 X 映射到 Set0，且不在缓存中 → 缓存未命中
  - 步骤 ①：访问 C（Set0 的 V）
  - Set0：C 被命中，原 V → O，NV D 降级为 V，在剩余 O（A、B）中随机选一个新 NV

Set	块 0	块 1	块 2	块 3	V	NV
Set0	A (O)	B (O)	C (V)	D (NV)	C	D
Set1	E (O)	F (O)	G (V)	H (NV)	G	H
Set2	I (V)	J (NV)	K (O)	L (O)	I	J

- 每个 set 只标记 2 个特殊块（V 和 NV），其余为 O。
- 替换优先顺序：先换 V，如果 V 被命中，更新状态；NV 被命中则重新挑选新 NV；O 命中不更新。

Victim/Next-Victim策略例子

- 假设我们设计一个访问序列：C → D → B → X
- 假设 X 映射到 Set0，且不在缓存中 → 缓存未命中
  - 步骤 ①：访问 C（Set0 的 V）
  - Set0：C 被命中，原 V → O，NV D 降级为 V，在剩余 O（A、B）中随机选一个新 NV
  - 更新后的 Set0 状态：

Set	块 0	块 1	块 2	块 3	V	NV
Set0	A (O)	B (NV)	C (O)	D (V)	C	D
Set1	E (O)	F (O)	G (V)	H (NV)	G	H
Set2	I (V)	J (NV)	K (O)	L (O)	I	J

- 每个 set 只标记 2 个特殊块（V 和 NV），其余为 O。
- 替换优先顺序：先换 V，如果 V 被命中，更新状态；NV 被命中则重新挑选新 NV；O 命中不更新。

Victim/Next-Victim策略例子

- 假设我们设计一个访问序列：C → D → B → X
- 假设 X 映射到 Set0，且不在缓存中 → 缓存未命中
  - 步骤 ②：访问 D（Set0 的 V）
  - Set0：D 被命中 → D 变 O，NV B 降级为 V，在 O（A、C）中随机选一个新 NV
  - 更新后的 Set0 状态：

Set	块 0	块 1	块 2	块 3	V	NV
Set0	A (NV)	B (V)	C (O)	D (O)	C	D
Set1	E (O)	F (O)	G (V)	H (NV)	G	H
Set2	I (V)	J (NV)	K (O)	L (O)	I	J

- 每个 set 只标记 2 个特殊块（V 和 NV），其余为 O。
- 替换优先顺序：先换 V，如果 V 被命中，更新状态；NV 被命中则重新挑选新 NV；O 命中不更新。

Victim/Next-Victim策略例子

- 假设我们设计一个访问序列：C → D → B → X
- 假设 X 映射到 Set0，且不在缓存中 → 缓存未命中
  - 步骤 ③：访问 B（Set0 的 V）
  - Set0：B 被命中 → B 变 O，NV A 降级为 V，在 O（C、D）中随机选一个新 NV
  - 更新后的 Set0 状态：

Set	块 0	块 1	块 2	块 3	V	NV
Set0	A (V)	B (O)	C (NV)	D (O)	C	D
Set1	E (O)	F (O)	G (V)	H (NV)	G	H
Set2	I (V)	J (NV)	K (O)	L (O)	I	J

- 每个 set 只标记 2 个特殊块（V 和 NV），其余为 O。
- 替换优先顺序：先换 V，如果 V 被命中，更新状态；NV 被命中则重新挑选新 NV；O 命中不更新。

Victim/Next-Victim策略例子

- 假设我们设计一个访问序列：C → D → B → X
- 假设 X 映射到 Set0，且不在缓存中 → 缓存未命中
  - 步骤 ④：访问 X（Set0，未命中）
  - Set0：缓存 miss → 用新块 X 替换 V A，NV C 降级为 V，从 O（B、D）中随机选一个新 NV
  - 更新后的 Set0 状态：

Set	块 0	块 1	块 2	块 3	V	NV
Set0	X (O)	B (NV)	C (V)	D (O)	C	D
Set1	E (O)	F (O)	G (V)	H (NV)	G	H
Set2	I (V)	J (NV)	K (O)	L (O)	I	J

- 每个 set 只标记 2 个特殊块（V 和 NV），其余为 O。
- 替换优先顺序：先换 V，如果 V 被命中，更新状态；NV 被命中则重新挑选新 NV；O 命中不更新。

缓存替换策略：LRU或随机（Random）

- LRU vs. Random：哪一个更好？
  - 示例：4 路组相联缓存，循环访问 A、B、C、D、E
  - A → B → C → D → E → A → B → ...
  - 使用 LRU 策略 → 命中率可能是 0%
- 集合抖动（Set Thrashing）
  - 当程序的“工作集”大小超过当前 set 的相联度时，会频繁替换，导致命中率低
  - 在这种情况下，随机替换策略通常比 LRU 更好
- 在实际应用中
  - 策略效果取决于具体工作负载
  - 在大多数情况下，LRU 与 Random 的平均命中率差别并不大
- 结合两者的优势：LRU + Random 混合策略
  - 如何动态选择最优策略 → 集合采样（Set Sampling）
  - 参考文献：Qureshi 等人，《A Case for MLP-Aware Cache Replacement》，ISCA 2006

什么是最优的缓存替换策略？

- Belady 的最优替换策略（OPT）
  - 原则：替换未来最晚会被再次访问的块
  - 提出者：Belady，《A Study of Replacement Algorithms for a Virtual-Storage Computer》，IBM Systems Journal，1966
  - 如何实现？只能通过模拟未来访问，实际硬件上无法直接实现
- OPT 是否在最小化未命中率上最优？
  - 是的，OPT 保证未命中次数最少
- OPT 是否在最小化程序执行时间上最优？不一定！
  - 因为 缓存未命中的代价（延迟/时间）会因数据块不同而变化
  - 原因主要有两点：
    - 远程缓存 vs 本地缓存 → 不同层次的缓存访问代价不同
    - 未命中重叠 → 多个未命中可以并行，不一定所有未命中都影响执行时间
  - 参考论文：Qureshi 等人，《A Case for MLP-Aware Cache Replacement》，ISCA 2006

补充：缓存替换与页面替换的关系

- 物理内存（DRAM）本质上是磁盘的缓存
  - 通常由操作系统通过虚拟内存子系统进行管理
- 页面替换（Page Replacement）与缓存替换类似
- 页表（Page Table）相当于物理内存的数据“标记存储区”（tag store）
- 两者的主要区别：
  - 访问速度要求不同：缓存访问速度要求比内存更快
  - 块数量不同：缓存中的块数量远少于物理内存
  - 可容忍的替换延迟不同：
    - 磁盘访问慢 → 页面替换可以容忍较长延迟
    - 缓存访问快 → 缓存替换必须非常迅速
  - 硬件与软件的角色不同：
    - 缓存管理主要依赖硬件
    - 页面替换主要由操作系统软件管理

写操作处理（Handling Writes I）

- 什么时候将缓存中被修改的数据写回到下一级存储？
  - 直写（Write-through）：在写入发生的同时立即写回主存
  - 回写（Write-back）：只在缓存块被替换（驱逐）时才写回主存

写操作处理（Handling Writes I）

- 什么时候将缓存中被修改的数据写回到下一级存储？
  - 直写（Write-through）：在写入发生的同时立即写回主存
  - 回写（Write-back）：只在缓存块被替换（驱逐）时才写回主存

- 回写（Write-back）
  - 优点：可以将对同一个缓存块的多次写入合并，只在被驱逐时写回一次
    - 这样能减少缓存与主存之间的带宽消耗，还能降低能耗
  - 缺点：需要在标签存储（Tag Store）中额外维护一个脏位（Dirty bit），用来标识该块是否被修改过

写操作处理（Handling Writes I）

- 什么时候将缓存中被修改的数据写回到下一级存储？
  - 直写（Write-through）：在写入发生的同时立即写回主存
  - 回写（Write-back）：只在缓存块被替换（驱逐）时才写回主存

- 回写（Write-back）
  - 优点：可以将对同一个缓存块的多次写入合并，只在被驱逐时写回一次
    - 这样能减少缓存与主存之间的带宽消耗，还能降低能耗
  - 缺点：需要在标签存储（Tag Store）中额外维护一个脏位（Dirty bit），用来标识该块是否被修改过

- 直写（Write-through）
  - 优点：实现更简单
  - 所有缓存层级（L1、L2、L3）与主存总是保持最新数据
  - 一致性（Consistency）更容易保证，不需要检查低层缓存的数据是否有效
  - 缺点：写操作会更频繁，对带宽消耗大，无法像回写一样合并多次写入，能效较低

写操作处理（Handling Writes II）

- 在写未命中（write miss）时，我们是否要为该地址分配一个缓存块？
  - 写分配（Allocate on write miss），也叫 Write-allocate:
    - 是
    - 同时把数据块从下一级存储加载到缓存中，供后续访问使用
  - 非写分配（No-allocate on write miss）：
    - 否
- 写分配（Write-allocate）
  - 优点：
    - 可以合并多次写操作，而不是每次写都立即写到下一级存储
    - 实现更简单：写未命中和读未命中可以统一处理
    - 适合写入局部性较高的场景
  - 缺点：
    - 可能需要把整个缓存块从下一级存储加载到缓存中，增加一次额外的数据传输
- 非写分配（写未分配，No-allocate）
  - 优点：如果写操作的局部性很低，这种策略节省缓存空间
    - 潜在地**提高缓存的总体命中率**
  - **缺点：**每次写都访问下一级存储

写操作处理（Handling Writes III）

- 如果处理器在很短的时间内会把整个缓存块都写一遍，该怎么办？（CPU默认Write-back + Write-allocate）‘
  - Write-back → 数据先写缓存，替换时再写回内存
  - Write-allocate → 如果写未命中，先在缓存中分配一个块

- 这种情况下，是否有必要先把这个块从内存加载到缓存呢？

写操作处理（Handling Writes III）

- 场景一：写满整个缓存块
- 例子：假设缓存块大小为 64 字节，现在处理器要在短时间内写满这 64 字节：
- 传统写分配 Write-allocate 策略的流程：
  1. CPU 第一次写入 buffer[0]，发现缓存未命中
  2. 从内存加载整个 64 字节缓存块到缓存
  3. 写入缓存 buffer[0]
  4. 后续写入 buffer[1] ~ buffer[63] → 全部在缓存中完成
  5. 如果缓存块被替换，再回写内存

写操作处理（Handling Writes III）

- 场景一：写满整个缓存块
- 例子：假设缓存块大小为 64 字节，现在处理器要在短时间内写满这 64 字节：
- 传统写分配 Write-allocate 策略的流程：
  1. CPU 第一次写入 buffer[0]，发现缓存未命中
  2. 从内存加载整个 64 字节缓存块到缓存
  3. 写入缓存 buffer[0]
  4. 后续写入 buffer[1] ~ buffer[63] → 全部在缓存中完成
  5. 如果缓存块被替换，再回写内存

- 问题：我们其实不需要原始数据，因为整个 64 字节会被覆盖，但缓存还是白白加载了 64 字节 → 浪费：
  - 时间：额外一次 64 字节加载
  - 带宽：没用的数据被传输
  - 能耗：无意义的访问

写操作处理（Handling Writes III）

- 场景一：写满整个缓存块
- 例子：假设缓存块大小为 64 字节，现在处理器要在短时间内写满这 64 字节：
- 传统写分配 Write-allocate 策略的流程：
  1. CPU 第一次写入 buffer[0]，发现缓存未命中
  2. 从内存加载整个 64 字节缓存块到缓存
  3. 写入缓存 buffer[0]
  4. 后续写入 buffer[1] ~ buffer[63] → 全部在缓存中完成
  5. 如果缓存块被替换，再回写内存

- 问题：我们其实不需要原始数据，因为整个 64 字节会被覆盖，但缓存还是白白加载了 64 字节 → 浪费：
  - 时间：额外一次 64 字节加载
  - 带宽：没用的数据被传输
  - 能耗：无意义的访问

- 优化方案：绕过缓存（Write-around）
  - 如果 CPU 知道自己会写满整个块，就可以直接：不加载内存数据到缓存
  - CPU如何知道？（编译器优化或者硬件预测机制）

写操作处理（Handling Writes III）

- 场景二：只写部分缓存块（Subblock Writes）
- 例子：同样的 64 字节缓存块，处理器只写其中 4 字节：

写操作处理（Handling Writes III）

- 场景二：只写部分缓存块（Subblock Writes）
- 例子：同样的 64 字节缓存块，处理器只写其中 4 字节：

- 两种情况：
  - 情况①：新值不依赖旧数据
  - 如果程序只写 4 字节，且不需要读取旧值：
    - 我们完全可以不加载整个 64 字节，只写那 4 字节到内存
    - 避免浪费带宽
    - “依赖旧数据” → 属于程序逻辑问题：要不要用旧值参与运算？

写操作处理（Handling Writes III）

- 场景二：只写部分缓存块（Subblock Writes）
- 例子：同样的 64 字节缓存块，处理器只写其中 4 字节：

- 两种情况：
  - 情况①：新值不依赖旧数据
  - 如果程序只写 4 字节，且不需要读取旧值：
    - 我们完全可以不加载整个 64 字节，只写那 4 字节到内存
    - 避免浪费带宽
    - “依赖旧数据” → 属于程序逻辑问题：要不要用旧值参与运算？
  - 情况②：新值需要和旧值合并
  - 如果程序写 4 字节，但后续还需要访问同一块的其他数据：
    - 就必须加载整个 64 字节到缓存
    - 更新 4 字节
    - 保证缓存与内存一致
    - “需要合并” → 属于硬件实现问题：写的数据覆盖范围小于缓存块时，要不要把没覆盖的部分保留下来？

写操作处理（Handling Writes III）

- CPU 怎么知道“是否依赖旧数据”
  - 取决于指令的语义，也就是看处理器在执行写操作的时候，是否需要读取旧数据。
  - 这种判断是由解码单元（Instruction Decoder）完成的，编译器在汇编时已经告诉 CPU 具体操作类型
  - 会在后续课程讲述

写操作处理（Handling Writes III）

- CPU 怎么知道“是否依赖旧数据”
  - 取决于指令的语义，也就是看处理器在执行写操作的时候，是否需要读取旧数据。
  - 这种判断是由解码单元（Instruction Decoder）完成的，编译器在汇编时已经告诉 CPU 具体操作类型
  - 会在后续课程讲述
  - (1) 两种写操作类型：
  - ① 完全覆盖式写入（不依赖旧数据）
    - 程序直接把一个新值写到内存位置上，完全覆盖原来的值
    - 指令形式例如：
      - mov [addr], eax ; 把寄存器 eax 的值直接写到 addr，不看旧值

写操作处理（Handling Writes III）

- CPU 怎么知道“是否依赖旧数据”
  - 取决于指令的语义，也就是看处理器在执行写操作的时候，是否需要读取旧数据。
  - 这种判断是由解码单元（Instruction Decoder）完成的，编译器在汇编时已经告诉 CPU 具体操作类型
  - 会在后续课程讲述
  - (1) 两种写操作类型：
  - ① 完全覆盖式写入（不依赖旧数据）
    - 程序直接把一个新值写到内存位置上，完全覆盖原来的值
    - 指令形式例如：
      - mov [addr], eax ; 把寄存器 eax 的值直接写到 addr，不看旧值
  - ② 读-改-写（依赖旧数据）读-改-写
    - 如果程序需要先读出旧值，再根据旧值计算出新值，然后写回去，就必须先知道旧数据
    - 指令形式例如：
      - add [addr], 5 ; 把 addr 的值 +5 再写回去

写操作处理（Handling Writes III）

- CPU 怎么知道“是否需要和旧数据合并”
  - 如果缓存块大小 64B，而 CPU 只写其中 4B，是否需要加载剩下的 60B？

写操作处理（Handling Writes III）

- **CPU 怎么知道“是否需要和旧数据合并”**
  - 如果缓存块大小 64B，而 CPU 只写其中 4B，是否需要加载剩下的 60B？
- **(1) 如果不依赖旧值**
  - 如果这 4B 完全覆盖了目标位置，而且程序不关心其他 60B 的内容：
    - 可以**只写这 4B**到下一级存储
    - 不需要加载整块 → 子块写（Subblock Write） 或 直写缓存（Write-around）

写操作处理（Handling Writes III）

- **CPU 怎么知道“是否需要和旧数据合并”**
  - 如果缓存块大小 64B，而 CPU 只写其中 4B，是否需要加载剩下的 60B？
- **(1) 如果不依赖旧值**
  - 如果这 4B 完全覆盖了目标位置，而且程序不关心其他 60B 的内容：
    - 可以**只写这 4B**到下一级存储
    - 不需要加载整块 → 子块写（Subblock Write） 或 直写缓存（Write-around）
- **(2) 如果需要旧值或合并数据**
  - 如果缓存块的剩余部分在后续可能会被用到，或者这次写入和旧数据相关：
    - 必须**先把整个缓存块加载到缓存**
    - 把 4B 的新值写到缓存块里
    - 缓存块的剩余 60B 依然保留原始数据
    - 最后根据策略（Write-back / Write-through）决定是否写回

写操作处理（Handling Writes III）

- **CPU 怎么知道“是否需要和旧数据合并”**
  - 如果缓存块大小 64B，而 CPU 只写其中 4B，是否需要加载剩下的 60B？
- **(1) 如果不依赖旧值**
  - 如果这 4B 完全覆盖了目标位置，而且程序不关心其他 60B 的内容：
    - 可以**只写这 4B**到下一级存储
    - 不需要加载整块 → 子块写（Subblock Write） 或 直写缓存（Write-around）
- **(2) 如果需要旧值或合并数据**
  - 如果缓存块的剩余部分在后续可能会被用到，或者这次写入和旧数据相关：
    - 必须**先把整个缓存块加载到缓存**
    - 把 4B 的新值写到缓存块里
    - 缓存块的剩余 60B 依然保留原始数据
    - 最后根据策略（Write-back / Write-through）决定是否写回
- **(3) 硬件如何判断是否要合并**
  - CPU 的写掩码（Write Mask）和指令信息会告诉缓存控制器：
    - 这次写入的**字节范围**是多少
    - 其他字节是否会被覆盖
    - 程序是否需要缓存块的其他部分
- **现代 CPU 的缓存控制器会利用这些信息自动决定：**
  - 只写必要的字节（不加载整块）
  - 或者先加载整块再写

写掩码（Write Mask）：是 CPU 在执行写操作时，用于标识哪些字节需要被更新的一组控制信号。

分区缓存（Sectored Caches）

- 核心思想：把一个缓存块（cache block）再细分为更小的子块（subblock）或扇区（sector），共享Tag但
  - 为每个子块单独维护 有效位（valid bit） 和 脏位（dirty bit）
    - 有效位（valid bit）：这个子块是否有效
    - 脏位（dirty bit）：子块是否被修改过
  - 这种方法在写操作场景下尤其有用

分区缓存（Sectored Caches）

- 核心思想：把一个缓存块（cache block）再细分为更小的子块（subblock）或扇区（sector），共享Tag但
  - 为每个子块单独维护 有效位（valid bit） 和 脏位（dirty bit）
    - 有效位（valid bit）：这个子块是否有效
    - 脏位（dirty bit）：子块是否被修改过
  - 这种方法在写操作场景下尤其有用
- 优点
  - 不需要每次都把整个缓存块加载进缓存
    - 写操作只需更新对应的子块，并设置其有效位即可
  - 在缓存块与主存之间传输数据更灵活
    - 只传输需要的子块，不必一次性加载整个缓存块
    - 例如：读操作时，可以只加载部分子块，而不是整个缓存块

分区缓存（Sectored Caches）

- 核心思想：把一个缓存块（cache block）再细分为更小的子块（subblock）或扇区（sector），共享Tag但
  - 为每个子块单独维护 有效位（valid bit） 和 脏位（dirty bit）
    - 有效位（valid bit）：这个子块是否有效
    - 脏位（dirty bit）：子块是否被修改过
  - 这种方法在写操作场景下尤其有用
- 优点
  - 不需要每次都把整个缓存块加载进缓存
    - 写操作只需更新对应的子块，并设置其有效位即可
  - 在缓存块与主存之间传输数据更灵活
    - 只传输需要的子块，不必一次性加载整个缓存块
    - 例如：读操作时，可以只加载部分子块，而不是整个缓存块
- 缺点
  - 设计更复杂，硬件实现成本更高
  - 对于读操作来说，不能充分利用空间局部性
    - 因为数据被拆得更细，连续访问的效率可能不如整块加载

脏位（Dirty Bit）是什么

- 脏位（Dirty Bit）是缓存中用于标记缓存块是否被修改过的一个状态位。
  - Dirty Bit = 1 → 缓存块被 CPU 修改过，缓存中的数据与主存不一致
  - Dirty Bit = 0 → 缓存块没有被修改，缓存与主存数据一致

脏位（Dirty Bit）是什么

- **脏位（Dirty Bit）是缓存中用于标记缓存块是否被修改过的一个状态位。**
  - Dirty Bit = 1 → 缓存块被 CPU 修改过，缓存中的数据与主存不一致
  - Dirty Bit = 0 → 缓存块没有被修改，缓存与主存数据一致
- **为什么需要脏位**
  - 缓存是主存的“加速器”，但缓存里的数据可能被 CPU 修改过。如果不加标记：
    - 当缓存块被替换（evicted）时，我们不知道是否需要把它写回主存。
    - 如果一律写回 → 会浪费带宽，即使数据根本没改。
    - 如果一律不写回 → 会丢失数据，主存就不再正确。
  - 解决方案：引入脏位
    - 被修改过 → 驱逐时写回主存
    - 没修改过 → 驱逐时直接丢弃缓存块

脏位（Dirty Bit）是什么

- **脏位（Dirty Bit）是缓存中用于标记缓存块是否被修改过的一个状态位。**
  - Dirty Bit = 1 → 缓存块被 CPU 修改过，缓存中的数据与主存不一致
  - Dirty Bit = 0 → 缓存块没有被修改，缓存与主存数据一致
- **为什么需要脏位**
  - 缓存是主存的“加速器”，但缓存里的数据可能被 CPU 修改过。如果不加标记：
    - 当缓存块被替换（evicted）时，我们不知道是否需要把它写回主存。
    - 如果一律写回 → 会浪费带宽，即使数据根本没改。
    - 如果一律不写回 → 会丢失数据，主存就不再正确。
  - 解决方案：引入脏位
    - 被修改过 → 驱逐时写回主存
    - 没修改过 → 驱逐时直接丢弃缓存块
- **脏位与 Subblock（子块）的关系：**
  - 在 sectored caches（分区缓存） 或 subblock caches（子块缓存） 中，每个缓存块会被拆成多个子块，每个子块有独立的脏位：
    - 如果只修改一个子块，只设置该子块的脏位为 1
    - 驱逐缓存块时，只写回脏位为 1 的子块
    - 节省带宽，避免写回整个缓存块
  - 例：缓存块 64B → 4 个 16B 子块
    - 修改子块 1 → 只把 子块 1 的脏位置 1
    - 驱逐时只写回 16B，而不是整块 64B。

缓存是分离的还是统一的好？

- 1. 统一缓存（Unified Cache）
- 优点：动态共享缓存空间
  - 指令（Instruction）和数据（Data）共用同一片缓存
  - 不会因为静态划分导致某一侧浪费空间
  - 例如，如果分离 I-cache 和 D-cache，其中一个满了而另一个空闲，就会造成资源浪费；统一缓存能避免这种问题。
- 缺点：
  - 指令与数据可能互相干扰
    - 如果程序同时大量访问指令和数据，可能造成缓存争用（thrashing）
    - 导致指令或数据都无法保证缓存空间
  - 流水线访问冲突
    - CPU 流水线中，**取指令**（fetch）和**访存数据**（load/store）发生在不同阶段
    - 如果只有一个统一缓存，硬件需要额外设计解决冲突，否则取指和访存不能并行

缓存是分离的还是统一的好？

- 1. 统一缓存（Unified Cache）
- 优点：动态共享缓存空间
  - 指令（Instruction）和数据（Data）共用同一片缓存
  - 不会因为静态划分导致某一侧浪费空间
  - 例如，如果分离 I-cache 和 D-cache，其中一个满了而另一个空闲，就会造成资源浪费；统一缓存能避免这种问题。
- 缺点：
  - 指令与数据可能互相干扰
    - 如果程序同时大量访问指令和数据，可能造成缓存争用（thrashing）
    - 导致指令或数据都无法保证缓存空间
  - 流水线访问冲突
    - CPU 流水线中，**取指令**（fetch）和**访存数据**（load/store）发生在不同阶段
    - 如果只有一个统一缓存，硬件需要额外设计解决冲突，否则取指和访存不能并行
- 2. 分离缓存（Split Cache）
  - L1 缓存通常是分离的（I-cache + D-cache）
    - 主要原因是流水线需要同时取指和访问数据
  - L2 缓存及以上通常是统一的
    - 因为高层缓存延迟更大，且带宽更高，争用冲突较少

CPU流水线设计中的多级缓存

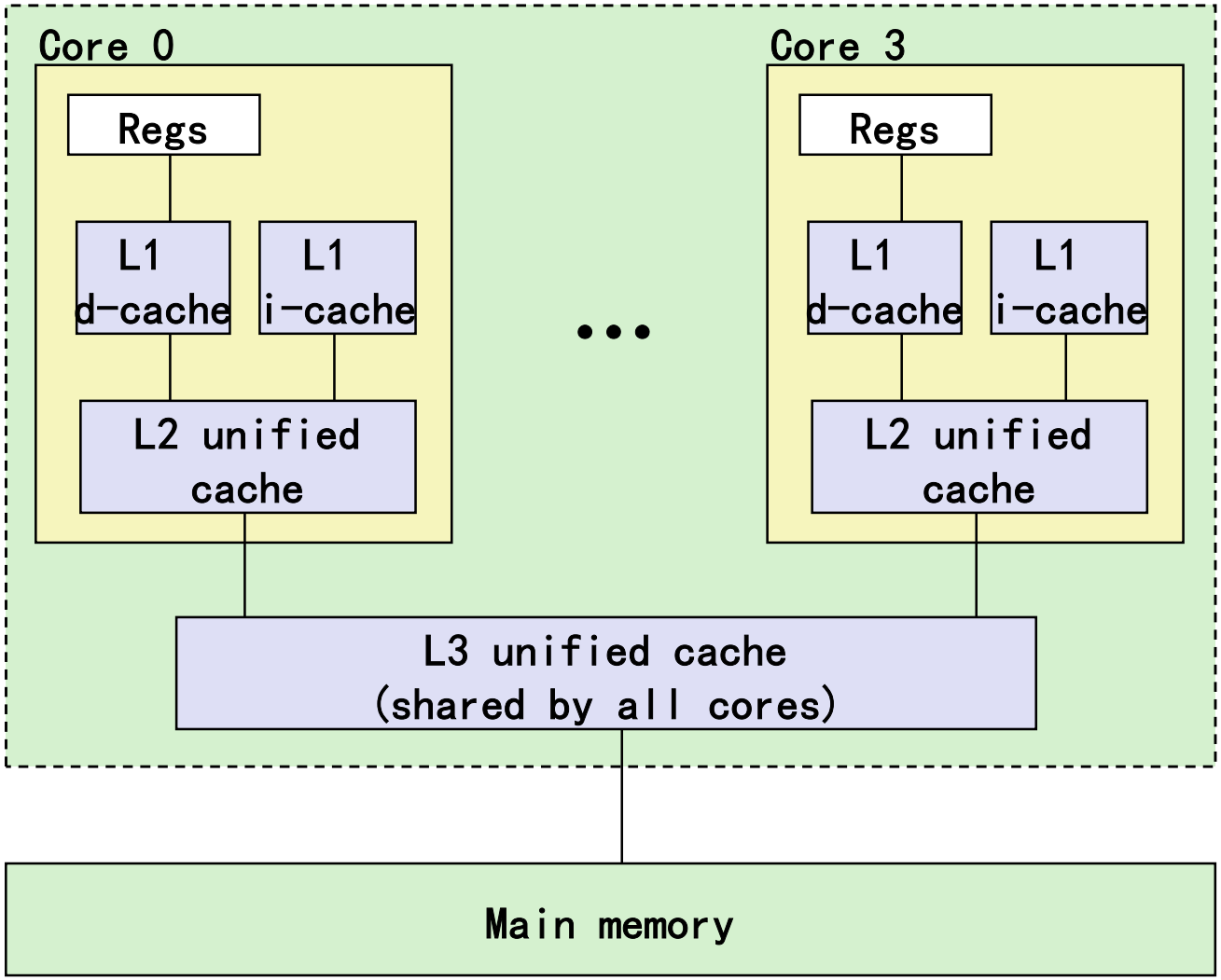
- 1. 一级缓存（L1 Cache，指令与数据）
  - 一级缓存的设计高度受限于 CPU 的时钟周期
  - 容量较小，通常采用低相联度（low associativity）
  - 标签存储（tag store）和数据存储（data store）通常并行访问
    - 提高速度，满足 CPU 每周期取指或读写数据的高性能需求
- 2. 二级缓存（L2 Cache）
  - 设计目标：在命中率和访问延迟之间找到平衡
  - 容量较大，通常采用高相联度（high associativity）
  - 标签存储和数据存储通常串行访问（serially）
    - 先查标签，确认命中后再取数据

CPU流水线设计中的多级缓存

- **1. 一级缓存（L1 Cache，指令与数据）**
  - 一级缓存的设计高度受限于 CPU 的时钟周期
  - 容量较小，通常采用低相联度（low associativity）
  - 标签存储（tag store）和数据存储（data store）通常并行访问
    - 提高速度，满足 CPU 每周期取指或读写数据的高性能需求
- **2. 二级缓存（L2 Cache）**
  - 设计目标：在命中率和访问延迟之间找到平衡
  - 容量较大，通常采用高相联度（high associativity）
  - 标签存储和数据存储通常串行访问（serially）
    - 先查标签，确认命中后再取数据
- **3. 串行 vs. 并行访问多级缓存**
  - 串行访问（Serial Access）：
    - 只有在 L1 缓存未命中时，才会去访问 L2 缓存
    - **L2 缓存不会看到所有访问请求**，因为 L1 已经过滤掉大多数
  - 并行访问（Parallel Access）：
    - 某些设计会让 L1 和 L2 同时访问，提高命中率，但增加能耗

Intel Core i7高速缓存层次结构

处理器封装



L1 指令高速缓存 和
数 据高速缓存:
32 KB, 8-way,
访问时间: 4周期

L2 统一的高速缓存:
256 KB, 8-way,
访问时间: 10 周期

L3 统一的高速缓存:
8 MB, 16-way,
访问时间: 40-75 周期

块大小:
所有缓存都是64 字节

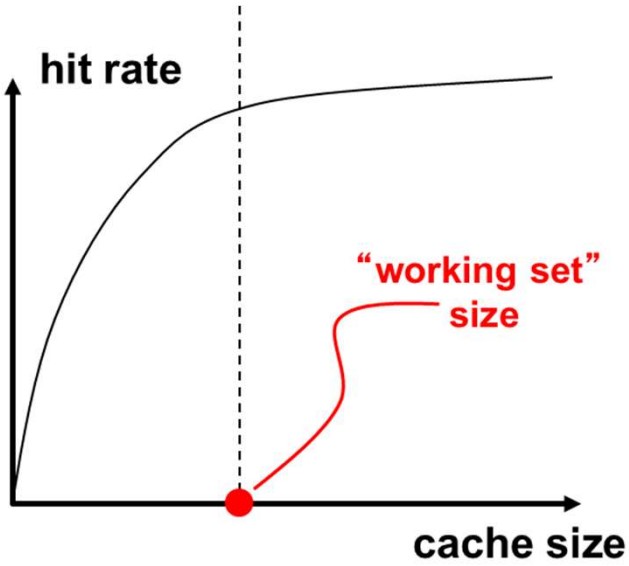
缓存性能

缓存参数与命中率/未命中率的关系

- 缓存大小（Cache size）
- 块大小（Block size）
- 相联度（Associativity）
- 替换策略（Replacement policy）
- 插入/放置策略（Insertion/Placement policy）

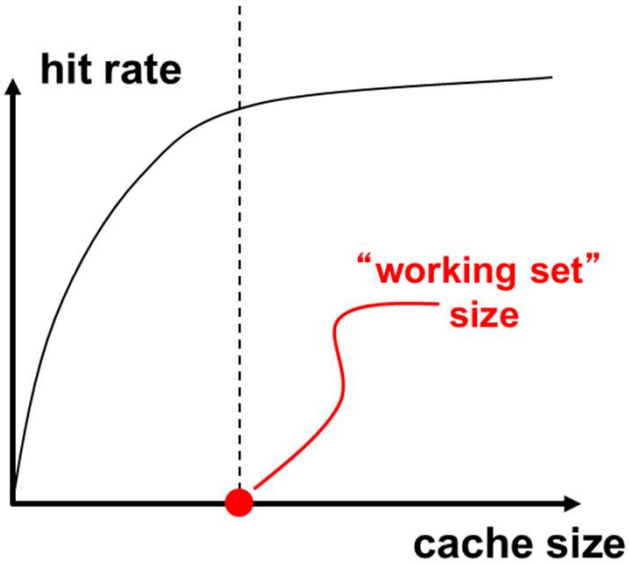
缓存大小（Cache Size）

- **缓存大小 ：缓存容量，仅指存储数据的大小**
  - 缓存越大，通常更容易利用时间局部性（temporal locality）
  - 但缓存并不是越大越好，有性能平衡点
- **更大的缓存往往访问速度更慢**
  - 因为缓存越大，查找和选择数据的电路更长
- **缓存太小，无法充分利用时间局部性**
  - 热点数据无法长期保留
  - 有用的数据会频繁被替换
    - 导致缓存未命中率上升，增加主存访问次数



缓存大小（Cache Size）

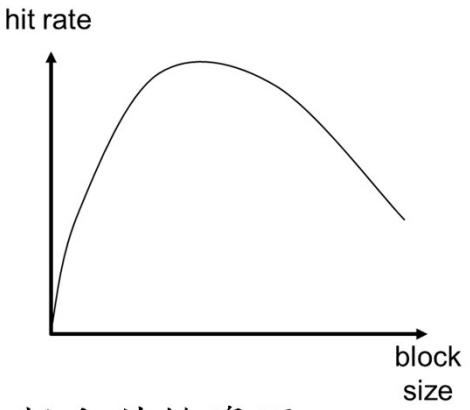
- **缓存大小 ：缓存容量，仅指存储数据的大小**
  - 缓存越大，通常更容易利用时间局部性（temporal locality）
  - 但缓存并不是越大越好，有性能平衡点
- **更大的缓存往往访问速度更慢**
  - 因为缓存越大，查找和选择数据的电路更长
- **缓存太小，无法充分利用时间局部性**
  - 热点数据无法长期保留
  - 有用的数据会频繁被替换
    - 导致缓存未命中率上升，增加主存访问次数



- **工作集：在某一段时间内，程序实际使用的全部数据集合**
  - 如果缓存大小 ≥ 工作集大小 → 高缓存命中率
  - 如果缓存大小 < 工作集大小 → 缓存不断抖动，命中率显著下降

回顾：缓存块大小（Block Size）

- 缓存块（Block size）：与一个地址标签（tag）关联的一段数据
  - 缓存块不一定是不同存储层级之间的最小传输单位
    - 子块化（Sub-blocking）
      - 将一个缓存块划分为多个子块，每个子块有独立的有效位
- 缓存块太小
  - 无法充分利用空间局部性
    - 一次加载的数据少，如果后续访问邻近数据，又会发生未命中
  - 标签存储开销大
    - 块数量增多，每个块都需要标签，硬件存储成本上升
- 缓存块太大
  - 块数量减少 → 降低时间局部性利用
    - 如果总块数太少，有用的数据更容易被替换掉
  - 浪费缓存空间与带宽
    - 如果程序只用到一小部分数据，却要加载整块，浪费能耗和传输资源
  - 空间局部性不足时，收益有限
    - 如果数据访问是随机的，块太大只会浪费



大缓存块：关键字优先与子块化

1. 大缓存块的问题：填充时间长

- 当缓存块很大时，从主存把整个块加载到缓存需要较长时间
- 关键字优先（Critical Word First）：
  - 优先加载 CPU 当前需要的那一部分数据（关键字）
  - 不必等整个缓存块加载完成，就能先返回所需数据，缩短等待时间
- 提前重启缓存访问（Early Restart）：
  - 当关键字加载完成后，CPU 可立即使用缓存，不必等待剩余部分

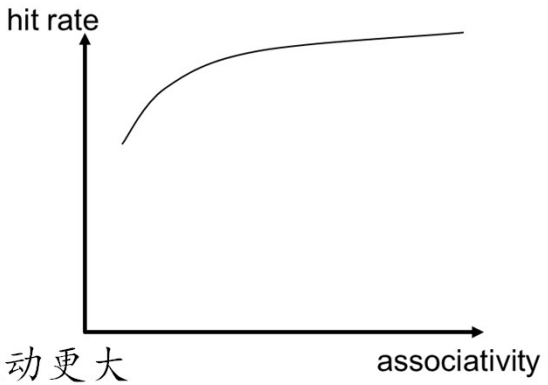
2. 大缓存块的问题：浪费总线带宽

- 如果缓存块很大，而程序只用到其中一部分数据，加载整个块会浪费带宽
- 子块化（Subblocking）：
- 适用场景：
  - 程序只访问缓存块中的部分数据
  - 写操作频繁但写入范围小
  - 内存带宽紧张，需要减少不必要的数据传输

回顾：相联度（Associativity）

- **相联度：有多少缓存块可以映射到同一个索引（set）中**

- **大相联度（Larger Associativity）**
  - 优点：
    - 降低未命中率（miss rate），因为更多候选位置可用
  - 缺点：
    - **收益递减（Diminishing Returns）**：相联度超过某个值后，命中率提升有限
    - **命中延迟更高**：需要比较更多标签，硬件路径更长
- **小相联度（Smaller Associativity）**
  - 优点：
    - 成本更低，硬件结构更简单
    - **命中延迟更低**，尤其对于 L1 缓存非常重要
  - 缺点：
    - 可能导致冲突未命中率（conflict miss）上升，性能波动更大



- **相联度必须是 2 的幂吗？**
  - 通常缓存的设计会让相联度是 2 的幂（例如 2 路、4 路、8 路组相联）
  - 但理论上，不一定非得是 2 的幂，只是会增加实现复杂度

回顾：缓存未命中的分类

- 1. 强制未命中（Compulsory Miss）
  - 定义：
    第一次访问某个内存地址（或缓存块）时，缓存中肯定没有该数据 → 必然未命中
  - 特征：
    - 后续访问同一地址通常会命中
    - 除非该块由于其他原因被替换掉
- 2. 容量未命中（Capacity Miss）
  - 定义：
    缓存容量太小，无法同时容纳程序运行所需的所有数据
  - 即使缓存是全相联（fully associative）、采用最优替换策略，如果容量不足，也会发生这种未命中
  - 本质上，容量未命中取决于缓存大小与工作集大小的关系
- 3. 冲突未命中（Conflict Miss）
  - 定义：
    除了强制未命中和容量未命中以外的所有未命中，称为冲突未命中
  - 发生原因：
    - 多个不同的内存块映射到了同一个缓存组（set）
    - 当相联度较低时，冲突未命中更常见

如何降低不同类型的缓存未命中

- **1. 强制未命中（Compulsory Miss）**
  - 缓存机制本身无能为力（第一次访问必然 miss）
  - 通过（Prefetching）可以降低影响：
    - 在程序需要数据之前，提前将数据加载到缓存中
- **2. 冲突未命中（Conflict Miss）**
  - 提高缓存相联度（Associativity）：
  - 不用增加相联度也能降低冲突的方法：
    - **Victim Cache（牺牲缓存）**
      用一个小型全相联缓存存放最近被替换的块
    - **软件提示（Software Hints）**
      编译器或操作系统根据数据访问模式提前优化缓存分布
- **3. 容量未命中（Capacity Miss）**
  - 更高效地利用缓存空间：
    - 优先保留会被频繁访问的数据块，避免无用数据占位
  - 软件优化（Software Management）：
    - 将工作集划分为不同的“阶段（phase）”
    - 确保每一阶段的数据都能放得下缓存，从而减少换入换出

提升缓存性能

- 1. 记住一个核心指标：AMAT
  - 平均内存访问时间（AMAT, Average Memory Access Time）
  - $AMAT = (hit-rate \times hit-latency) + (miss-rate \times miss-latency)$
- 2. 降低未命中率（Reducing Miss Rate）
  - 但需要注意：
    - 如果为了降低 miss rate 而频繁替换那些重新加载代价高的数据块
    - 反而可能导致性能下降
- 3. 降低未命中延迟/成本（Reducing Miss Latency/Cost）
- 4. 降低命中延迟/成本（Reducing Hit Latency/Cost）

提升缓存性能

- 1. 降低缓存未命中率（Reducing Miss Rate）
  - 提高相联度（Associativity）
    - 从直接映射 → 组相联 → 全相联，降低冲突未命中
  - 更优的替换/插入策略
    - 改进 LRU、随机替换等策略，提高有效缓存利用率
  - 软件优化
    - 通过编译器优化、数据布局调整、循环重排等方式减少缓存冲突

提升缓存性能

- **1. 降低缓存未命中率（Reducing Miss Rate）**
  - 提高相联度（Associativity）
    - 从直接映射 → 组相联 → 全相联，降低冲突未命中
  - 更优的替换/插入策略
    - 改进 LRU、随机替换等策略，提高有效缓存利用率
  - 软件优化
    - 通过编译器优化、数据布局调整、循环重排等方式减少缓存冲突
- **2. 降低缓存未命中延迟或代价（Reducing Miss Latency/Cost）**
  - 多级缓存（Multi-level Caches）
    - 用 L1、L2、L3 多级设计，减少访问主存的延迟
  - 关键字优先（Critical Word First）
    - 先返回 CPU 当前需要的字，而不是等整块数据都加载完
  - 子块化/分区缓存（Subblocking/Sectoring）
    - 把缓存块拆成子块，减少无关数据加载
  - 更优的替换/插入策略
    - 提高缓存命中率，间接减少延迟
  - 软件方法
    - 编译器和操作系统配合缓存预取（prefetching），降低等待时间