

Lecture 10: 知识 - Cache

程序访问的局部性原理

时空局部性

概念：时间局部性原理

- 概念：
 - 时间局部性是指最近的**未来要用到的信息**，很可能是**现在正在使用的信息**；
 - 讲人话，就是现在正在用的东西，马上就还要用到
- 来源：
 - 1. 循环；
 - 2. 需要多次重复执行的子程序段；
 - 3. 对数组的存储和访问操作；

概念：空间局部性原理

- 空间局部性是指**最近的未来要用到的信息**，很可能与现在正在使用的信息**在存储空间上是邻近的**，因为指令通常是顺序存放、顺序执行，数据一般也是以向量、数组等形式成簇的存储的；

分析：程序的访问命中率

分析：数组

- 当数组的存放顺序和访问顺序一致时，程序的空间局部性好；
- 比如当二维数组是行优先存储时，此时如图在 `for` 的外层控制列、内层控制行的索引，则数组空间局部性好；

分析：`for` 循环体

- 对于 `for` 循环体，程序 A 和程序 B 中的访问局部性是一样的；
- 因为循环体内指令按序连续存放，所以空间局部性好；
- 内循环体被连续重复执行 `2048 * 2048` 次，因此时间局部性也好；
- 图示：

程序 A:

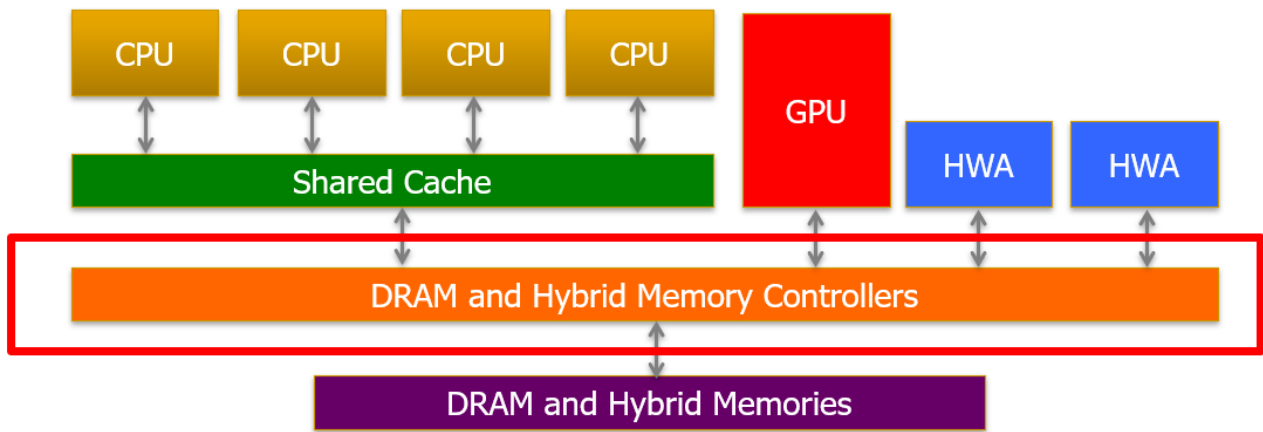
```
1  int sumarrayrows(int a[M][N])
2  {
3      int i, j, sum = 0;
4      for (i = 0; i < M; i++)
5          for (j = 0; j < N; j++)
6              sum += a[i][j];
7      return sum;
8  }
```

程序 B:

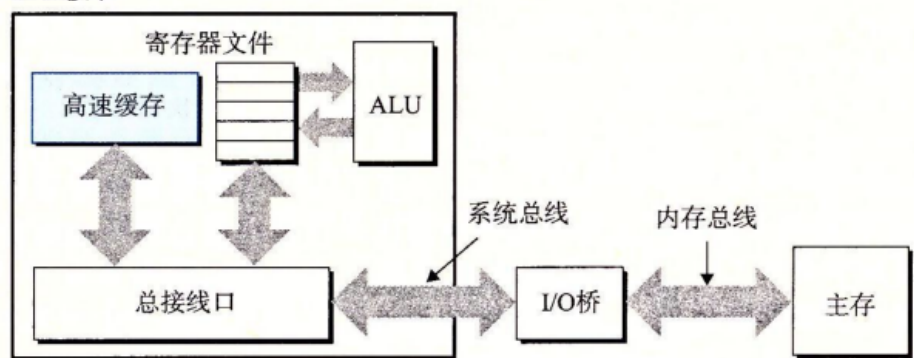
```
1  int sumarraycols(int a[M][N])
2  {
3      int i, j, sum = 0;
4      for (j = 0; j < N; j++)
5          for (i = 0; i < M; i++)
6              sum += a[i][j];
7      return sum;
8  }
```

Cache 工作原理

概述：关系图

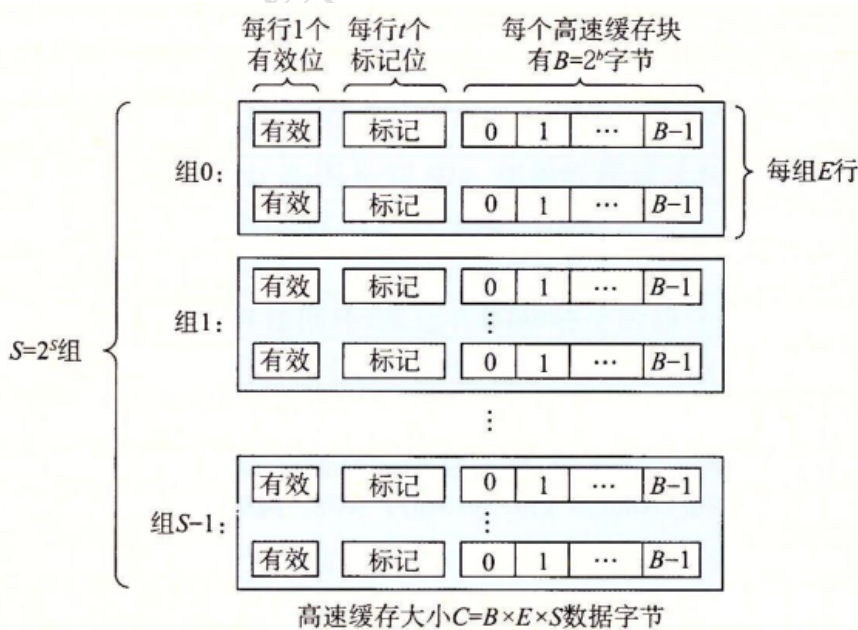


CPU芯片



概述：逻辑存储图（组相联映射）

- 解释：
 - 标记：当前块的内存地址的位的一个子集；
 - 有效：一位 bit 位，标记当前行（也可以称之为“块”）是否包含有效数；
- 图示：



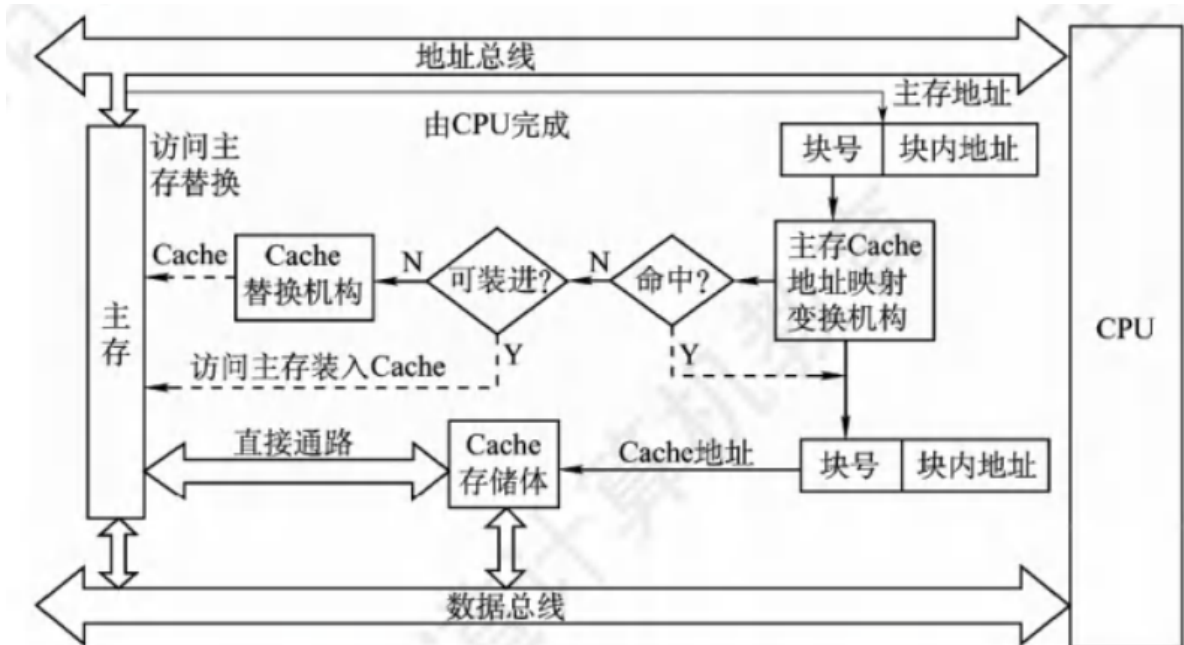
基本工作原理

概念：Cache 的基本工作原理

工作原理:

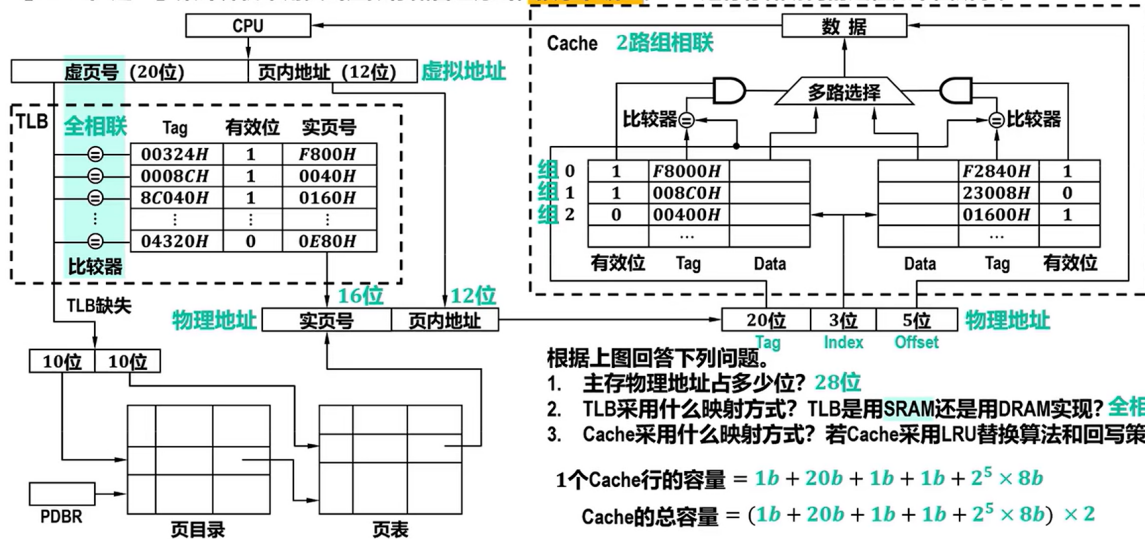
- 为便于 Cache 与主存交换信息, Cache 和主存都被划分为大小相等的块, Cache 块也称 Cache 行;
- 每块由若干字节组成, 块的长度称为块长 (也称行长);
- 因为 Cache 的容量远小于主存的容量, 所以 Cache 中的块数要远少于主存中的块数, Cache 中仅保存主存中最活跃的若干块的副本;
- 因此需要按照某种策略, 预测 CPU 在未来一段时间内欲访存的数据, 将其装入 Cache;

基本结构:



例题:

【2018年 题44】某计算机采用页式虚拟存储管理方式, 按字节编址。CPU进行存储访问的过程如下图所示。



Cache 命中分析

分析: Cache 命中对 CPU 执行时间的影响

概念:

- 当 CPU 发出读请求时:
 - 若访存地址在 Cache 中命中:

- 就将此地址转换成 **Cache** 地址，直接对 **Cache** 进行读操作，与主存无关；
- 若 **Cache** 不命中：
 - 则仍需访问主存，并把此字所在的块一次性地从主存调入 **Cache**；
 - 若此时 **Cache** 已满，则需根据某种替换算法，用这个块替换 **Cache** 中原来的某块信息；
- 当 **CPU** 发出写请求时：
 - 若 **Cache** 命中：
 - 有可能会遇到 **Cache** 与主存中的内容不一致的问题；
 - 例如：由于 **CPU** 写 **Cache**，把 **Cache** 某单元中的内容从 **a** 修改成 **b**，而主存对应单元中的内容仍然是 **a**，没有改变；
 - 因此若 **Cache** 命中，需要按照一定的写策略处理，常见的处理方法有**全写法**和**回写法**
- 总结：**TLB**、**Cache**、主存
 - 真题：2010年
 - 1. 不存在：**TLB** 命中，页表未命中，**Cache** 未命中；
 - **TLB** 命中时，数据肯定在主存中，页表肯定能命中，所以错误；
 - 2. 不存在：**TLB** 命中，页表未命中，**Cache** 命中；
 - **TLB** 命中时，数据肯定在主存中；
 - 3. 不存在：**TLB** 未命中，页表未命中，**Cache** 命中；
 - **Cache** 命中时，数据肯定在主存中，只是如果是写命中，可以会出现数据不一致情况，需要用修改位标识；
 - 即：只要 **Cache** 或者 **TLB** 能命中，则此时页表肯定也能命中；
- 注意：
 - 1. 访问 **Cache** 整个过程全部由**硬件实现**；
 - 真题：2020年
 - **TLB** 和 **Cache** 的缺失处理都可以由硬件实现，缺失后都需要去访问主存，**TLB** 缺失访存是为了访问页表；
 - 2. **CPU** 与 **Cache** 之间的数据交换以**字**为单位（数据通路）；
 - 3. **Cache** 与主存之间的数据交换则以 **Cache** 块为单位；
- 分析：
 - 真题：2020年
 - 题目：
 - 假定主存地址为 **32** 位，按字节编址，指令 **Cache** 和数据 **Cache** 与主存之间均采用 8 路组相联映射方式直写 (Write Through)写策略和 LRU 替换算法，主存块大小为 64 B，数据区容量各为 32 KB。开始时 **Cache** 均为空请回答下列问题；
 - 有如下C语言程序段：


```
for (k = 0; k < 1024; k++)
  s[k] = 2 * s[k];
```
 - 若数组s及其变量k均为int型、int型数据占4B，变量k分配在寄存器中，数组s在主存中的起始地址为0080 00C0H，
 - 则在该程序段执行过程中，访问数组s的数据Cache缺失次数是多少？
 - 解析：

- 由于主存块大小为 64 B，因此块内偏移为 6 位；
- 数据区容量为 32KB，而且一组中有 8 个 Cache 行，因此一共有 $32KB/(64B*8)=64$ 组，因此组号为 6 位
- 因此 Tag 位占 $32-12=20$ 位；
- 由于数组 S 的起始地址为 0080 00C0H，因此前 20 位为 00800，组号为 3，块内偏移为 0；

数组s在主存中的起始地址为0080 00C0H

↓ 二进制

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

主存分区号

cache组号 块内偏移
主存数据块号

$$2^{11} = 2048$$

3

0

- 由此可知数组 S 的一共占 64 个数据块：

数组s在主存中的起始位置：主存分区号2048中的主存数据块3的起始处

for (k = 0; k < 1024; k++)

s[k] = 2 * s[k];

数组s的数据类型为int

循环次数

$$\text{数组s所占主存数据块的数量} = \frac{4B \times 1024}{64B} = 64$$

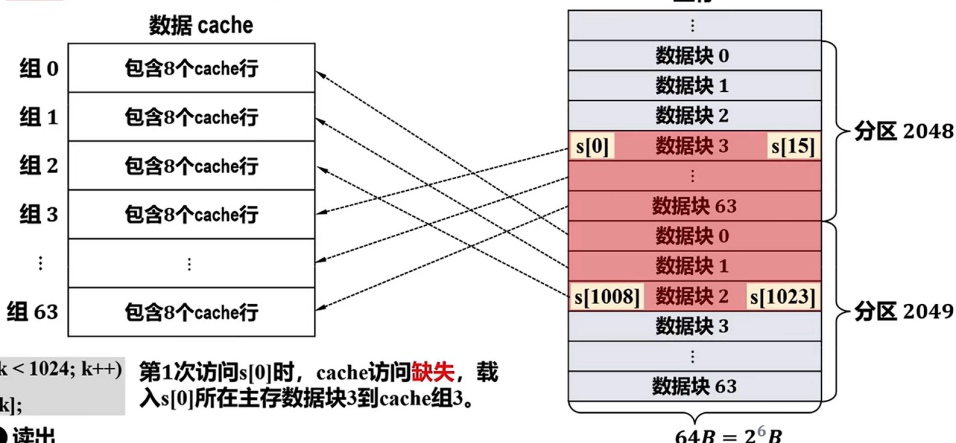
主存数据块大小

- 由数组 S 的 VA，可知其 Cache 块分布情况，其分布在 Tag 号为 2048 和 Tag 号为 2049 的两个组内；

解析

数组s在主存中的起始位置：主存分区号2048中的主存数据块3的起始处

数组s所占主存数据块的数量 = 64



for (k = 0; k < 1024; k++)

s[k] = 2 * s[k];

第1次访问s[0]时，cache访问缺失，载入s[0]所在主存数据块3到cache组3。

- ② 写入 ① 读出

- 每访问缺失一次时，会调入一块的内容，而一块中有 16 个元素。因此第一个元素的写入、后续 15 个元素的读出、写入都可以成功访问；
- 因为一共 64 块，因此一共会产生 64 次读 Cache 缺失；

分析：Cache 命中率计算

- 概念：

- CPU 欲访问的信息已在 Cache 中的比率称为 Cache 的命中率 (H)
- H 越靠近 1 越好
- 公式:
 - Cache 总命中次数: N_c
 - 访问主存的总次数: N_m
 - 命中率: H
 - 公式:

$$H = \frac{N_c}{(N_c + N_m)}$$

分析: Cache 缺失率对主存带宽的影响

- 公式:
 - $T_{\text{平均访问时间}} = T_{\text{Cache命中时的访问时间}} * \text{命中率} + T_{\text{Cache缺失时的访问时间}} * (1 - \text{命中率})$
- 举例:
 - 问题:
 - 假设 Cache 的速度是主存的 5 倍, 且 Cache 的命中率为 95%, 则采用 Cache 后, 存储器性能提高多少 (假设采用先访问 Cache, Cache 不命中时, 才采用访问主存的方式)
 - 解答:
 - 已知 $5T_{\text{Cache}} = T_{\text{主存}}$
 - $T_{\text{使用Cache后的平均访问时间}} = 0.95 * T_{\text{Cache}} + 0.05 * (5 * T_{\text{Cache}} + T_{\text{Cache}}) = 1.25T_{\text{Cache}}$
 - 因为 $5T_{\text{Cache}} = T_{\text{主存}}$
 - 所以使用 Cache 之和的效率为 $\frac{5T_{\text{Cache}}}{1.25T_{\text{Cache}}} \sim 4\text{倍}$

Cache 和主存的映射方式

引入: 映射方式

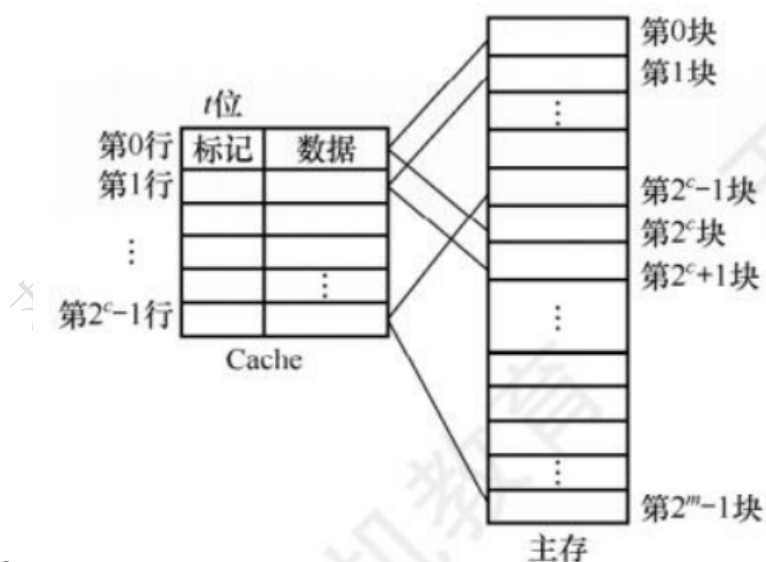
- 原因:
 - 由于 Cache 行数比主存块数少得多, 因此主存中只有一部分块的信息可放在 Cache 中;
- 标记:
 - 需要在 Cache 中要为每块加一个**标记位**, 指明它是主存中哪一块的副本;
 - 该标记的内容相当于**主存中块的编号**;
- 有效位:
 - 为了说明 Cache 行中的信息是否有效, 每个 Cache 行需要一个有效位;
 - 用有效位来标记此 Cache 行是否有效;
- 信息:
 - Cache 行中的信息是主存中某个**块**的副本;
 - 即: 主存中一块和 Cache 的一行数据对应;

- 地址映射：
 - 地址映射是指把主存地址空间映射到 **Cache** 地址空间，即把存放在主存中的信息按照某种规则装入 **Cache**；

直接映射

概念：直接映射

- 实现：
 - 主存中的每一块只能装入 **Cache** 中的唯一位置；
 - 若这个位置已有内容，则产生块冲突，原来的块将无条件地被替换出去（无须使用替换算法）；
- 特点：
 - 优点：
 - 直接映射实现简单；
 - 缺点：
 - 不够灵活，即使 **Cache** 的其他许多地址空着也不能占用；
 - 使得直接映射的**块冲突概率最高，空间利用率最低**；
- 映射关系：
 - 主存块号的低 **c** 位正好是它要装入的 **Cache** 行号，给每个 **Cache** 行设置一个长为 **t=m-c** 的标记；
- $$\text{Cache行号} = \text{主存块号} \bmod \text{Cache总行数}$$
- 图示：
 - 映射方式



- 直接映射的地址结构：

- 结构：

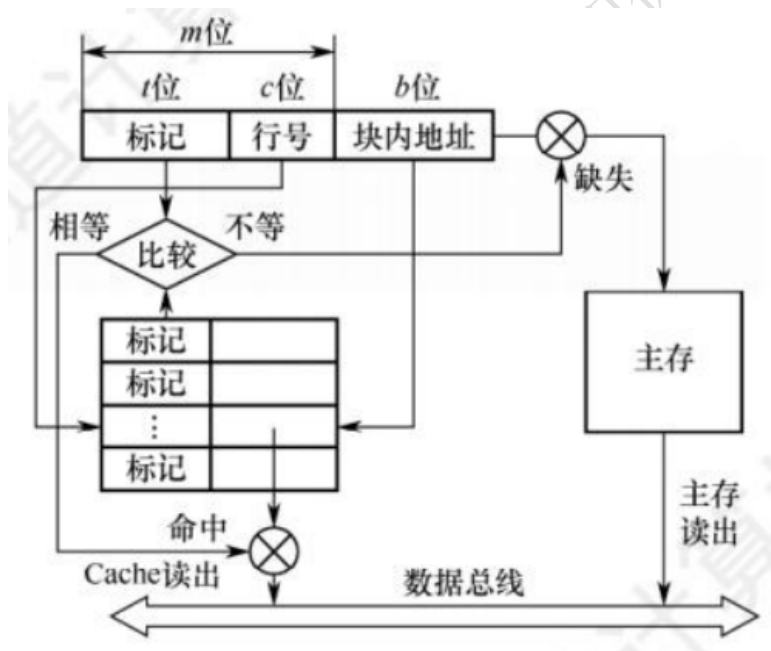
标记	Cache 行号	块内地址
----	----------	------

- 举例：

主存块号		块内地址
22位		6位
19位 标记	3位 行号	6位块内 地址

分析：直接映射的访问过程

- 概念：
 - 标记比较：
 - 首先根据访存地址中间的 c 位，找到对应的 Cache 行，将对应 Cache 行中的标记和主存地址的高 t 位标记进行比较；
 - 访问成功：
 - 若 Cache 行中的标记和主存地址的高 t 位标记比较后相等，且有效位为 1，则访问 Cache “命中”；
 - 此时根据主存地址中低位的块内地址，在对应的 Cache 行中存取信息；
 - 访问失败：
 - 若不相等或有效位为 0，则“不命中”；
 - 此时 CPU 从主存中读出该地址所在的一块信息，送到对应的 Cache 行中，将有效位置 1，并将标记设置为地址中的高位，同时将该地址中的内容送 CPU；
- 图示：



全相联映射

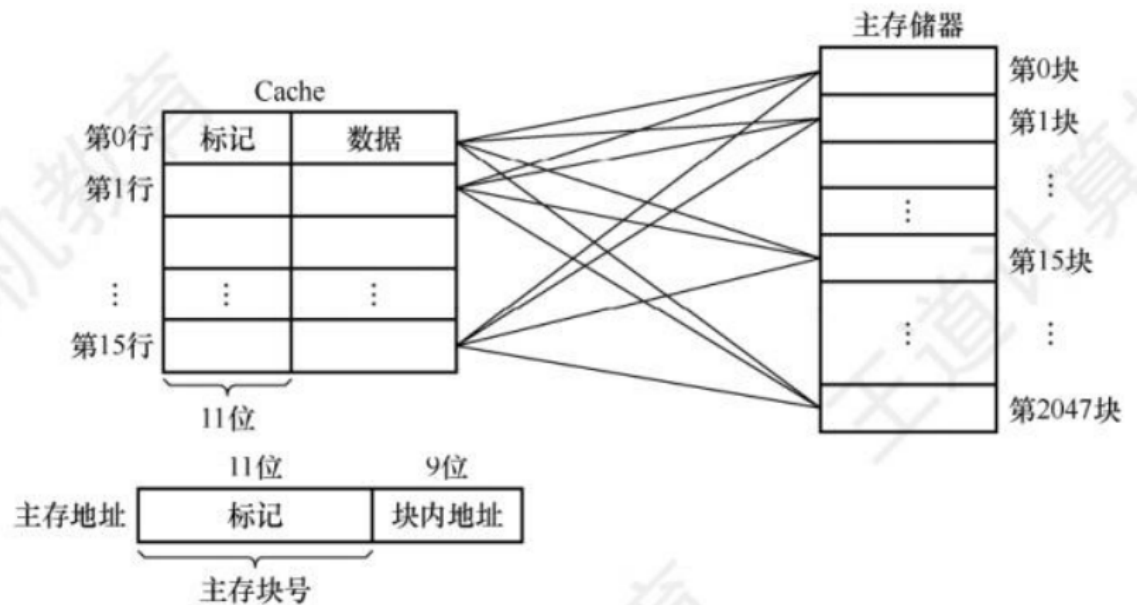
概念：全相联映射

- 实现：
 - 主存中的每一块可以装入 Cache 中的任何位置；
 - 每行的标记用于指出该行来自主存的哪一块，因此 CPU 访存时需要与所有 Cache 行的标记进行比较；
- 特点：

- 优点：
 - 1. Cache 块的冲突概率低，只要有空闲 Cache 行，就不会发生冲突；
 - 2. 空间利用率高；
 - 3. 命中率高；
- 缺点：
 - 1. 标记的比较速度较慢；
 - 2. 实现成本较高，通常需采用按内容寻址的相联存储器；

• 图示：

- 全相联映射方式：



- 地址结构：



分析：全相联的访问过程

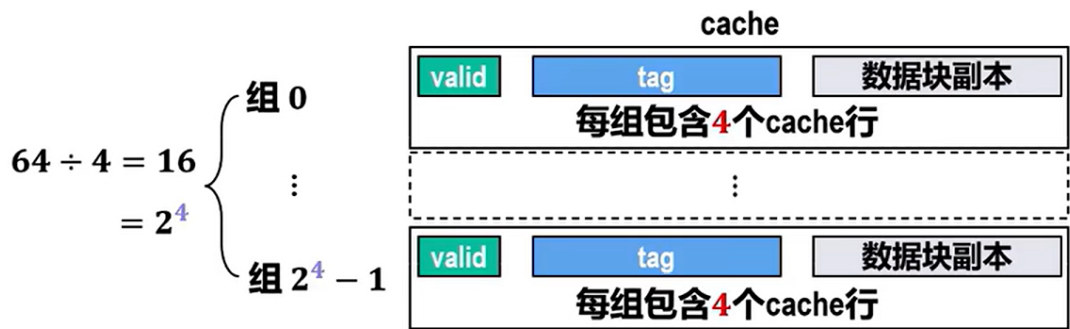
- 概念：
 - 缓存标记比较：
 - 首先将主存地址的高位标记与 Cache 各行的标记进行比较；
 - 访问成功：
 - 若有一个标记相等，且对应有有效位为 1 -> 则命中；
 - 此时根据块内地址从该 Cache 行中取出信息；
 - 访问不成功：
 - 若两个里面有一个不相等（标记、有效位） -> 不命中；
 - 此时 CPU 从主存中读出该地址所在的一块信息，送到 Cache 的任意一个空闲行中，将有效位置 1，并设置标记，同时将该地址中的内容送 CPU；
- 比较器数量：
 - 数量：
 - 通常为每个 Cache 行都设置一个比较器；
 - 比较器位数等于标记字段的位数；

- 比较器的数量等于 Cache 的行数；
- 访存时：
 - 根据标记字段的内容来访问 Cache 行中的主存块，因而其查找过程是一种“按内容访问”的存取方式；
 - 所以是一种“相联存储器”；
- 特点：
 - 这种方式的时间开销和硬件开销都较大，不适合大容量 Cache；

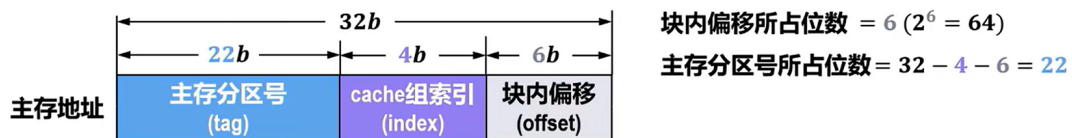
组相联映射

概念：组相联映射

- 概念：
 - 将 Cache 分成 Q 个大小相等的组，每个主存块可以装入固定组中的任意一行；
 - 假设每组有 r 个 Cache 行，则称为 r 路组相联；
- 核心：
 - 组间采用直接映射、而组内采用全相联映射的方式；
 - 它是对直接映射和全相联映射的一种折中，选定适当的数量，可使组相联映射的成本接近直接映射，而性能上仍接近全相联映射；
- 分析：
 - 当 $Q=1$ 时 $\rightarrow$ 变为全相联映射；
 - 当 $Q=\text{Cache 行数}$ $\rightarrow$ 变为直接映射；
 - 路数越大、每组 Cache 行的数量越大 $\rightarrow$ 发生块冲突的概率越低、相联比较电路也越复杂；
- 映射关系：
 - $$\text{Cache组号} = \text{主存块号} \bmod \text{Cache组数}(Q)$$
 - 注意：
 - (1) 是主存块号和组数相 mod，而不是地址号和组数相 mod，因此如果给出的是主存地址，还需要先转换为主存块号后、再进行映射；
 - (2) 主存块从 0 开始计数；
- 分析：
 - 真题：2019年
 - 题目：
 - 若计算机 M 的主存地址为 32 位，指令 Cache 有 64 行，采用 4 路组相联映射方式，主存块大小为 64 B，则 32 位主存地址中，哪几位表示块内地址？哪几位表示 Cache 组号？哪几位表示标记 (tag) 信息？
 - 解析：
 - 由于当前一共有 64 个 Cache 行，而 4 行为一个组，因此一共 16 组，如图；



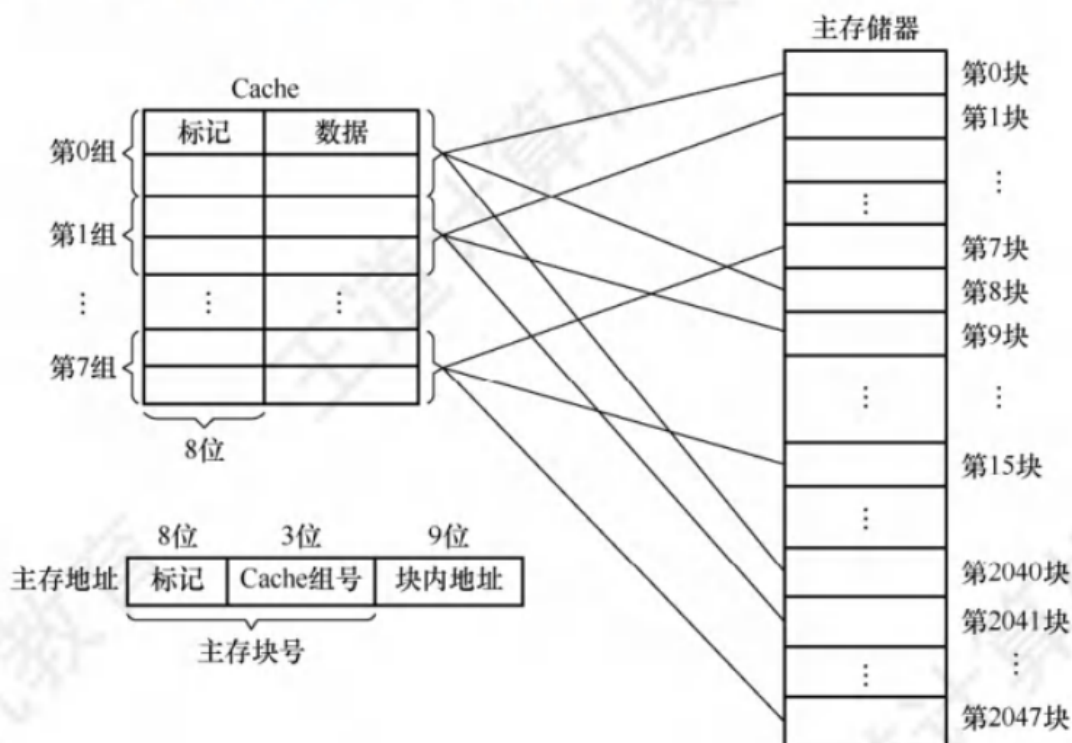
- 并且由于 Cache 块为 64 B, 因此需要 6 位表示块内偏移:



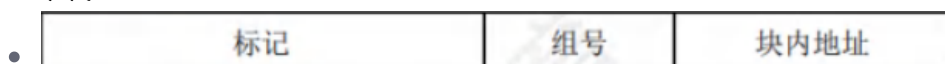
- 由此可得 Tag 位数;

图示:

- 映射方式:



- 地址结构:



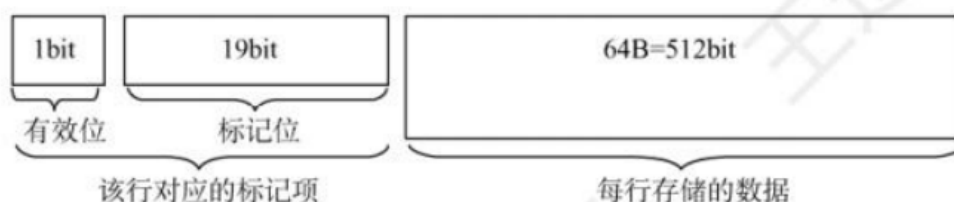
分析: 组相联映射的访存过程及 cache 缺失处理过程

- 第 1 步:
 - 根据访存地址中间的组号 -> 找到对应的 Cache 组;
- 第 2 步:
 - 将对应 Cache 组中每个行的标记与主存地址的高位标记进行比较;
 - 因为 r 路组相联映射中每一组 cache 有 r 行, 因此需要有 r 个比较器;
- 第 3.1 步:

- 若有一个相等且有效位为 1，则访问 Cache 命中；
- 此时根据主存地址中的块内地址，在对应 Cache 行中存取信息；
- 第 3.2 步：
 - 若都不相等或虽相等但有效位为 0，则不命中；
 - 此时 CPU 从主存中读出该地址所在的一块信息送到对应 Cache 组的任意一个空闲行中，将有效位置 1，并设置标记；
 - 同时将该地址中的内容送 CPU；

分析：直接映射、组相联映射相关标记位及总容量的分析

- 前提：
 - 假设某个计算机的主存地址空间大小为 256 MB，按字节编址；
 - 其数据 Cache 有 8 个 Cache 行，行长为 64B；
- 问题一：求 Cache 总容量
 - 描述：
 - 若不考虑用于 Cache 的一致维护性位 (脏位) 和替换算法控制位，并且采用直接映射方式，则该数据 Cache 的总容量为多少？
 - 解答：
 - 分析：
 - 因为 Cache 包括了可以对 Cache 中所包含的存储器地址进行跟踪的硬件，即：Cache 的总容量=存储容量+标记阵列容量（有效位、标记位）
 - 标记阵列：



- 下面是 Cache 行的存储容量示意图
- 主存地址大小：256MB = $2^8 * 2^{20} = 2^{28}$ ，即 28b
- 有效位大小：1b
- 块内地址大小：64B = 2^6 ，所以为 6b
- 行号：8 = 2^3 ，即 3b
- 标记位大小：主存地址位数 - 块内地址位数 - 有效位位数 = 标记位位数
- 标记位位数 = 28 - 6 - 3
- 存储容量：
 - 64B = 64 * 8 = 512b
 - 总容量Cache = 行数 * 每行大小 = 8 * (512b + 1b + 19b) = 4256b
- 问题二：求某主存地址对应的 Cache 行号
 - 描述：
 - 若该 Cache 采用直接映射方式，则主存地址为 3200 (十进制) 的主存块对应的 Cache 行号是多少？采用二路组相联映射时又是多少？
 - 解答：

- 求主存地址对应的主存字块号：主存块号 = $\frac{\text{主存地址}}{\text{块大小}} = 3200\text{B}/64\text{B} = 50$
- 求直接映射下的行号：
 - 因为直接映射是根据模运算来得到主存与 Cache 的映射，因此 Cache 行号 = $50 \bmod 8 = 2$
- 求组相连映射下的组号：
 - 因为组相联映射中，组和主存的映射关系是直接映射关系。组内是全映射关系，因此通过主存块号可以知道其对应的 Cache 组号：
 - Cache 组号 = $50 \bmod \frac{8}{2} = 50 \bmod 4 = 2$
- 求组相连映射下的行号：
 - 知道了组号为 2 后，可以知道其对应的行号为 $2 \times 2 = 4$ ，或者是 $4 + 1 = 5$ ；
- 问题三：简述直接映射的访存
 - 描述：
 - 以直接映射方式为例，简述访存过程（设访存的地址为 0123456_{H} ）
 - 解答：
 - 因为当前的主存大小为 2^{28} ，所以想要描述主存的地址，主存地址必须要用 28b 的大小去存储，因此访存的地址都是 28b
 - 根据 cache 的信息，可以知道 28 位主存地址分为 19 位的主存标记位，3 位的块号，6 位的块内地址；
 - 拆解访存地址：
 - $012, 3456_{\text{H}} = 0000, 0001, 0010, 0011, 0100, 0101, 0110$
 - 高 19 位为主存标记位：0000, 0001, 0010, 0011, 010
 - 中 3 位为块号：001
 - 后 6 位为块内地址：01, 0110
 - 首先根据块号，查 Cache（即 001 号 Cache 行）中对应的主存标记位，看是否相同；
 - 若相同，再看 Cache 行中的有效位是否为 1；
 - 若是，称此访问命中，按块内地址 010110 读出 Cache 行所对应的单元并送入 CPU 中，完成访存；
 - 若出现标记位不相等或有效位为 0 的情况，则不命中，访问主存将数据取出并送往 CPU 和 Cache 对应块中，把主存地址的高 19 位写入 001 行的标记位，并将有效位置 1；

关联度

概念：映射方式与关联度

- 真题：2024年
- 概念：关联度
 - 对于一个主存块来说，3 种映射方式下对应 Cache 行的个数不同；
 - 直接映射是唯一映射，每个主存块只有一个行与之对应；全相联映射是任意映射，每个行都可对应；N 路组相联映射有 N 行对应；
 - 这种特性可用“关联度”来度量，关联度指一个主存块映射到 Cache 中时、可能存放的位置个数；

- 因此，直接映射的关联度最低，为 1；
- 全相联映射的关联最高，为 Cache 的总行数；
- N 路组相联映射的关联度居中，为 N；
- 概念：关联度与性能
 - (1) 关联度越低，命中率越低；
 - 因此直接映射命中率最低，全相联映射命中率最高；
 - 真题：2024年
 - Cache - 主存 层次间可以采用直接映射，因为 Cache-主存 系统缺失时所付出的代价没有那么大，因此可以采用直接相连映射，提高映射的速度；
 - 主存 - 外存 层次间不能使用直接映射，只能使用全相联映射。因为访问外存的代价大，如果使用直接映射，则需要频繁的对外存进行 IO，代价太高。并且相比于外存 IO 时间而言，判断是否命中所花的时间可以忽略，因此 主存-外存 间应当使用全相联映射；
 - (2) 关联度越低，判断是否命中的开销越小，命中时间越短；
 - 因此，直接映射的命中时间最短，全相联映射的命中时间最长；
 - (3) 关联度越低，标记所占额外空间开销越少；
 - 因此，直接映射额外空间开销最少，全相联映射额外空间开销最大；
- 分析：关联度相关计算
 - 题目：
 - 假定主存地址为 32 位，按字节编址，主存块大小为 16B，cache 共有 4K 行；
 - 则在关联度分别为 1、2、4 和全相联方式下标记所占总位数是多少？
 - 解答：
 - 关联度为 1 (即直接映射) 时，每组 1 行，共 4K 组，标记 (行号) 占 $32-4-12=16$ 位，总位数占 $4\text{K} \times 16=64\text{K}$ 位；
 - 关联度为 2 (即 2 路组相联) 时，每组 2 行，共 2K 组，标记 (组号) 占 $32-4-11=17$ 位，总位数占 $4\text{K} \times 17=68\text{K}$ 位；
 - 关联度为 4 (即 4 路组相联) 时，每组 4 行，共 1K 组，标记 (组号) 占 $32-4-10=18$ 位，总位数占 $4\text{K} \times 18=72\text{K}$ 位；
 - 全相联时，整个为 1 组，每组有 4K 行，标记占 $32-4=28$ 位，总位数占 $4\text{K} \times 28=112\text{K}$ 位；

Cache 替换算法

全相联映射：替换算法

- Cache 完全满了，才需要替换需要在全局选择替换哪一块；

直接映射：不需要替换算法

- 如果对应位置非空，则毫无选择地直接替换；

组相连映射：替换算法

- 分组内满了，才需要替换需要在分组内选择替换哪一块；

随机算法 RA

概念

- 概念：
 - 若 Cache 已满，则随机选择一块替换；
- 特点：
 - 实现简单，但完全没考虑局部性原理，命中率低，实际效果很不稳定；
- 图示：

访问主存块	1	2	3	4	1	2	5	1	2	3	4	5
Cache #0	1	1	1	1	1	1	1	1	1	1	4	4
Cache #1		2	2	2	2	2	2	2	2	2	2	2
Cache #2			3	3	3	3	5	5	5	5	5	5
Cache #3				4	4	4	4	4	4	3	3	3
Cache命中?	否	否	否	否	是	是	否	是	是	否	否	是
Cache替换?	否	否	否	否	否	否	是	否	否	是	是	否

先进先出算法 FIFO

概念

- 概念：
 - 若 Cache 已满，则替换最先被调入 Cache 的块；
- 特点：
 - 先进先出算法—实现简单，最开始按 #0 #1 #2 #3 放入 Cache ，之后轮流替换 #0 #1 #2 #3
 - FIFO 依然没考虑局部性原理，最先被调入 Cache 的块也有可能是被频繁访问的；
- 抖动现象：
 - 刚被替换的块很快又被调入 -> 频繁的换入换出现象；
- 图示：

访问主存块	1	2	3	4	1	2	5	1	2	3	4	5
Cache #0	1	1	1	1	1	1	5	5	5	5	4	4
Cache #1		2	2	2	2	2	2	1	1	1	1	5
Cache #2			3	3	3	3	3	3	2	2	2	2
Cache #3				4	4	4	4	4	4	3	3	3
Cache命中?	否	否	否	否	是	是	否	否	否	否	否	否
Cache替换?	否	否	否	否	否	否	是	是	是	是	是	是

近期最少使用 LRU

概念

- 概念：

- 为每一个 Cache 块设置一个“计数器”，用于记录每个 Cache 块已经有多久没被访问了；
- 当 Cache 满后替换“计数器”最大的；
- LRU 算法可能会有抖动现象；
- 实现：
 - 命中时：
 - 所命中的行的计数器清零，比其低的计数器加 1，其余不变；
 - 未命中且还有空闲行时：
 - 新装入的行的计数器置 0，其余非空闲行全加 1；
 - 未命中且无空闲行时：
 - 计数值最大的行的信息块被淘汰，新装行的块的计数器置 0，其余全加 1；
- 特点：
 - LRU 算法是基于“局部性原理”的 -> 近期被访问过的主存块，在不久的将来也很有可能被再次访问；
 - 因此：淘汰最久没被访问过的块是合理的；
 - LRU 算法的实际运行效果优秀，Cache 命中率高；

最不经常使用算法 LFU

概念

- 概念：
 - 为每一个 Cache 块设置一个计数器，记录每个 Cache 块被访问过几次；
 - 当 Cache 满后，将“计数器”值最小的替换掉；
 - 即：新调入的块计数器=0，之后每被访问一次计数器+1。需要替换时，选择计数器最小的一行；
 - 若有多个计数器最小的行可按行号递增或 FIFO 策略进行选择；
- 特点：
 - 曾经被经常访问的主存块在未来不一定会用到 (如：微信视频聊天相关的块)；
 - 并没有很好地遵循局部性原理，因此实际运行效果不如 LRU；
- 图示：

计数器	访问主存块	1	2	3	4	1	2	5	1	2	3	4	5
2	Cache #0	1	1	1	1	1	1	1	1	1	1	1	1
2	Cache #1		2	2	2	2	2	2	2	2	2	2	2
0	Cache #2			3	3	3	3	5	5	5	3	3	5
1	Cache #3				4	4	4	4	4	4	4	4	4
	Cache命中?	否	否	否	否	是	是	否	是	是	否	是	否
	Cache替换?	否	否	否	否	否	否	是	否	否	是	否	是

Cache 写策略

写命中

概念：回写法 -> Write Back

- 原理：
 - 当 CPU 对 Cache 写命中时，只修改 Cache 的内容，而不立即写入主存，只有当此块被换出时才写回主存；
 - 同时：未被修改的块不必写回；
- 脏位：
 - 为了能够区分数据是否被 CPU 写回过，因此需要一个脏位来表示是否被修改过；
- 特点：
 - 减少了访存次数，但存在数据不一致的隐患；

概念：全写法 -> Write-Through

- 原理：
 - 当 CPU 对 Cache 写命中时，必须把数据同时写入 Cache 和主存，一般使用写缓冲 (write buffer)；
 - Cache 块被替换时无需写回；
- 写缓冲：
 - 使用 SRAM 实现的 FIFO 队列；
 - 对主存的数据写同步、会先放到写缓存当中，有一个专门的控制电路、逐一进行写回；
 - 因为 CPU 写的速度很快，若写操作不频繁，则效果很好；
 - 若写操作频繁，则可能发生阻塞；
- 特点：
 - 因为需要在每次 cache 写中时对主存进行修改，因此访存次数增加，速度变慢；
 - 但更能保证数据一致性；

写不命中

概念：写分配法 (write-allocate)

- 更新主存单元，然后把这个主存块调入 Cache；
- 它试图利用程序的空间局部性，缺点是每次写不命中都要从主存读一个块到 Cache 中；

概念：非写分配法 (not-write-allocate)

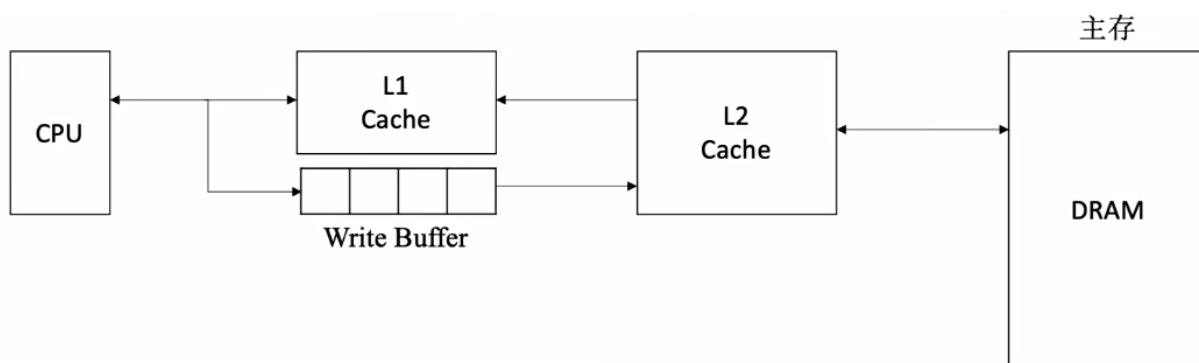
- 只更新主存单元，而不把主存块调入 Cache；
- 非写分配法通常与全写法合用，写分配法通常和回写法合用；

多级 Cache 结构与写回

概念

- 概念：
 - 现代计算机常采用多级 Cache 离 CPU 越近的速度越快，容量越小离 CPU 越远的速度越慢，容量越大
 - 各级 Cache 之间常采用“全写法+非写分配法；
 - Cache - 主存之间常采用“写回法+写分配法；
- 解释：

- 现代计算机的 Cache 通常设立多级 Cache;
- 假定设 2 级 Cache, 按离 CPU 的远近可各自命名为 L1 Cache、L2 Cache;
- 离 CPU 越远, 访问速度越慢, 容量越大, 指令 Cache 与数据 Cache 分离一般在 L1 级, 此时通常为写分配法与回写法合用;
- 下图是一个含有两级 Cache 的系统, L1 Cache 对 L2 Cache 使用全写法, L2 Cache 对主存使用回写法;
- 由于 L2 Cache 的存在, 其访问速度大于主存, 因此避免了因频繁写时造成的写缓冲饱和和溢出;
- 真题: 2014年
 - 采用指令缓存和数据缓存分离的主要目的, 是减少指令流水线资源的冲突;
- 图示:



获取笔记持续更新可加微信