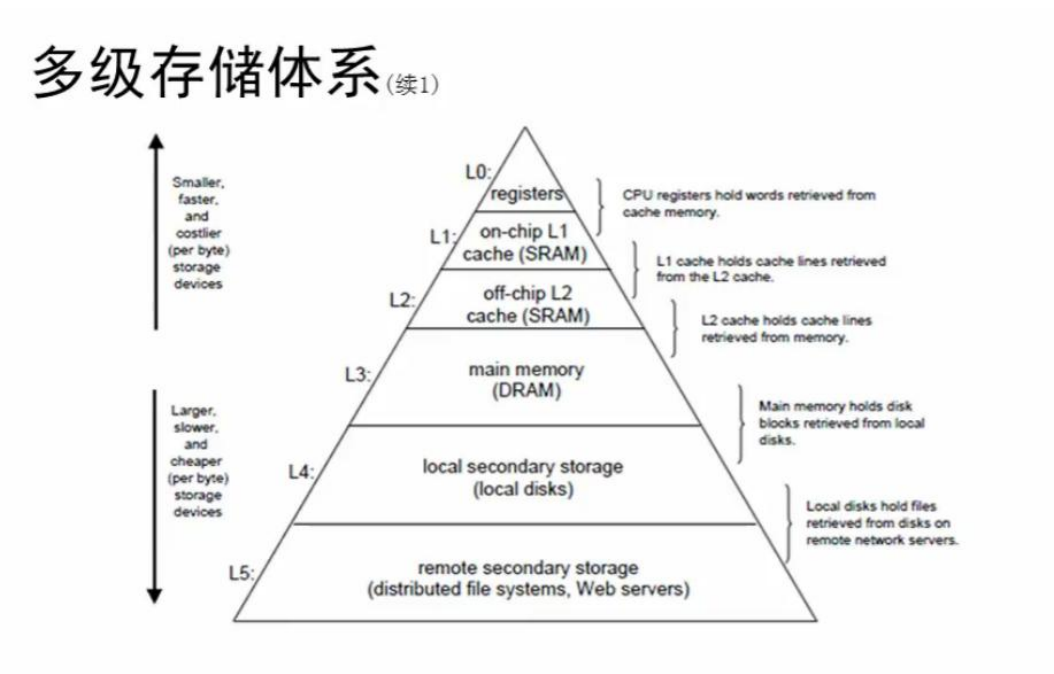
csapp

笔记本： 考研专业课
创建时间： 2018/10/10 17:31 **更新时间：** 2018/12/5 19:23
作者： 15801850335@163.com



寄存器——>缓存（1级缓存->2级缓存）——>主存——>硬盘——>远端服务器

MIPS 包括：算术运算指令（Arithmetic instructions），数据传输指令（Data transfer instructions），条件转移指令（Conional instructions）和无条件转移指令（Not conditional instructions）.

指令	功能	应用实例
LB	从存储器中读取一个字节的数据到寄存器中	LB R1, 0(R2)
LH	从存储器中读取半个字的数据到寄存器中	LH R1, 0(R2)
LW	从存储器中读取一个字的数据到寄存器中	LW R1, 0(R2)
LD	从存储器中读取双字的数据到寄存器中	LD R1, 0(R2)
L.S	从存储器中读取单精度浮点数到寄存器中	L.S R1, 0(R2)
L.D	从存储器中读取双精度浮点数到寄存器中	L.D R1, 0(R2)
LBU	功能与LB指令相同，但读出的是不带符号的数据	LBU R1, 0(R2)
LHU	功能与LH指令相同，但读出的是不带符号的数据	LHU R1, 0(R2)
LWU	功能与LW指令相同，但读出的是不带符号的数据	LWU R1, 0(R2)
SB	把一个字节的数据从寄存器存储到存储器中	SB R1, 0(R2)
SH	把半个字节的数据从寄存器存储到存储器中	SH R1，0(R2)
SW	把一个字的数据从寄存器存储到存储器中	SW R1, 0(R2)
SD	把两个字节的数据从寄存器存储到存储器中	SD R1, 0(R2)
S.S	把单精度浮点数从寄存器存储到存储器中	S.S R1, 0(R2)
S.D	把双精度数据从存储器存储到存储器中	S.D R1, 0(R2)
DADD	把两个定点寄存器的内容相加，也就是定点加	DADD R1,R2,R3
DADDI	把一个寄存器的内容加上一个立即数	DADDI R1,R2,#3
DADDU	不带符号的加	DADDU R1,R2,R3
DADDIU	把一个寄存器的内容加上一个无符号的立即数	DADDIU R1,R2,#3
ADD.S	把一个单精度浮点数加上一个双精度浮点数，结果是单精度浮点数	ADD.S F0,F1,F2

左手指令、右手数据

Amdahl定律

- Amdahl定律可以阐述为：系统中某一部件由于采用某种更快的执行方式后所获得系统性能的提高，与这种执行方式的使用频率或占总执行时间的比例有关。

Amdahl定律定义了一台计算机系统采用某种改进措施所取得的加速比。

$$\text{加速比}=\frac{\text{采用改进措施后计算机的性能}}{\text{没有采用改进措施时计算机的性能}}$$

$$=\frac{\text{没有采用改进措施时某任务的执行时间}}{\text{采用改进措施后某任务的执行时间}}$$

加速比反映了使用改进措施后完成一个任务比不使用改进措施完成同一任务加快的比率。

- Amdahl定律中，加速比与两个因素有关：

a. 在原有的计算机上，能被改进的部分在总执行时间中所占的比例。即

$$\frac{\text{可改进部分的执行时间}}{\text{改进前整个任务的执行时间}}$$

这个值称为改进比例，记为Fe，它总小于等于1。例如，如果一个任务在原来机器上的执行时间为60s，其中20s的执行时间可以使用改进措施，那么Fe就是20 / 60。

b. 改进部分采用改进措施后，相比没有采用改进措施前性能提高的倍数，即

$$\frac{\text{改进前改进部分的执行时间}}{\text{改进后改进部分的执行时间}}$$

这个值称为改进加速比，记为Se，它总大于1。例如，在原来的条件下程序的某一部分执行时间为5s，而改进后这一部分只需要2s，那么Se就是5 / 2。

CPU性能公式

* 大多数计算机都有一个产生周期性定时信号的时钟。

* 时钟的长度可以用时钟周期（如2ns）或其频率（如500MHz）来度量。一个程序执行时所花费的CPU时间可以表示为：

CPU时间 = 一个程序的CPU时钟周期数×时钟周期

或

CPU时间 = 一个程序的CPU时钟周期数 / 频率

* 此外，还可以用一个程序的指令条数（用IC表示）和执行所需的时钟周期数，来计算出执行一条指令所需的平均时钟周期数CPI（Clock cycles Per Instruction）：

CPI = 一个程序的CPU时钟周期数 / IC

* CPI是衡量不同指令和不同实现方法的一个处理器性能指标。

流水线 吞吐率：

某指令流水线由5段组成，各段所需要的时间如下图所示。

--> t --> 3t --> t --> 2t --> t -->

连续输入10条指令时的**吞吐率**为（）。

A.10/70t B.10/49t C.10/35t D.10/30t

解答：

第一条指令 - （ - - - ） - （ - - ） -

第二条指令 - （ - - - ） - （ - - ） -

第三条指令 - （ - - - ） - （ - - ） -

因为 是流水线，所以时间为3t的指令不能重叠，所以每隔3t时间开始一条指令，当第一条指令花费8t时间后，每隔3t完成一条指令，第10条指令完成的时间是：8+3*9=35t.

吞吐率为:10条指令/花费时间35t = 10/35

弄懂两个概念就好做了：流水线时间和**吞吐率**

流水线时间计算有个公式：一条指令所需时间+（指令条数-1）*时间最长的指令的一段 // 8t+9*3t=35t

吞吐率也有个公式：指令条数除以流水线时间 // 10/35t

经典的五级流水线

特殊的，对于计算机体系结构来说，原理和汽车流水线一样，将一条指令的完成分成若干部分（流水节拍或流水段），然后指令依次流过这些流水段，就完成了这条指令。对于经典的5级流水线，如下：

1，IF（指令取得）

需要完成的工作：

Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 (since each instruction is 4 bytes) to the PC.

根据PC（程序计数器）指示的地址，从存储器中取指令，并装入到IR（指令寄存器），同时PC+4。当然假设每条指令占4个字节。

2，ID（指令译码）

需要完成的工作：

Decode the instruction and read the registers corresponding to register source specifiers from the register file。

对指令进行译码，并且访问寄存器堆读出相应寄存器的内容。

3，EX（执行）

需要完成的工作：

The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.

Memory reference

Register-Register ALU instruction

Register-Immediate ALU instruction

ALU对上一个周期准备好的操作数进行运算，根据指令类型执行下面三种操作之一。

访问存储器

寄存器-寄存器ALU指令

寄存器-立即数ALU指令

4，MEM（访问存储器）

需要完成的工作：

If the instruction is a load, memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.

LOAD：从存储器中读取数据。STOR：把寄存器内容写到存储器。

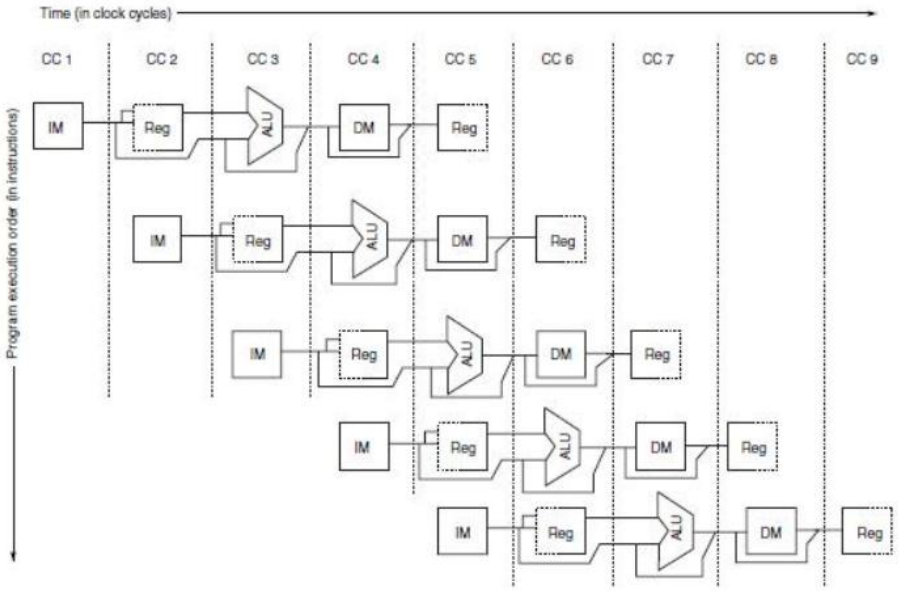
5，WB（写回）

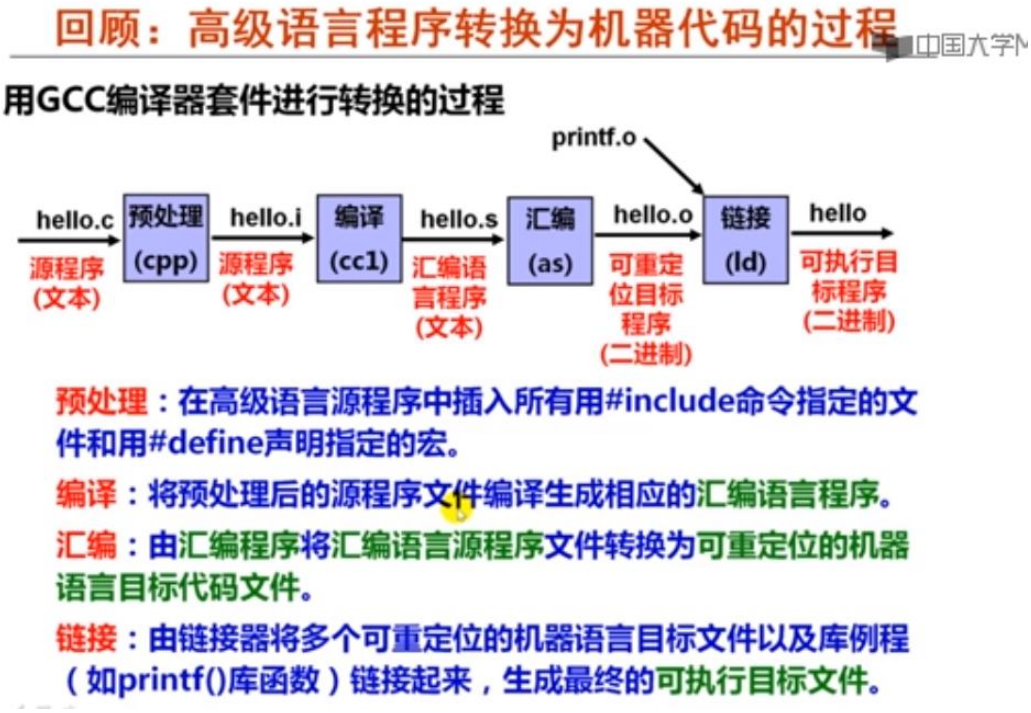
需要完成的工作：

寄存器-寄存器ALU指令，LOAD指令会经过这个流水段。

Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

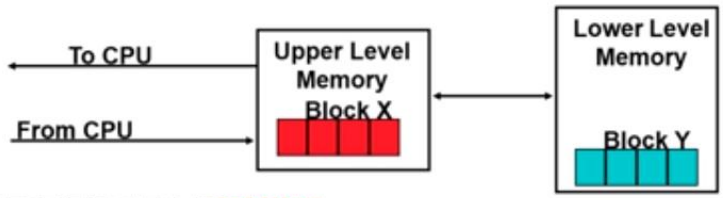
将结果（来自LOAD指令或来自ALU）写入寄存器堆。





局部性原理

层次化存储器结构（Memory Hierarchy）



数据总是在相邻两层之间复制传送

Upper Level: 上层更靠CPU

Lower Level: 下层更远离CPU

Block: 最小传送单位是定长块，互为副本

相当于工厂中设置了多级仓库！

问题：为什么这种层次化结构是有效的？

程序访问局部化特点！
例如，写论文时图书馆借参考书：欲借书附近的书也是欲借书！

- 时间局部性（Temporal Locality）
  - 含义：刚被访问过的单元很可能不久又被访问
  - 做法：让最近被访问过的信息保留在靠近CPU的存储器中
- 空间局部性（Spatial Locality）
  - 含义：刚被访问过的单元的邻近单元很可能不久被访问
  - 做法：将刚被访问过的单元的邻近单元调到靠近CPU的存储器中

加快访存速度措施之三：引入Cache

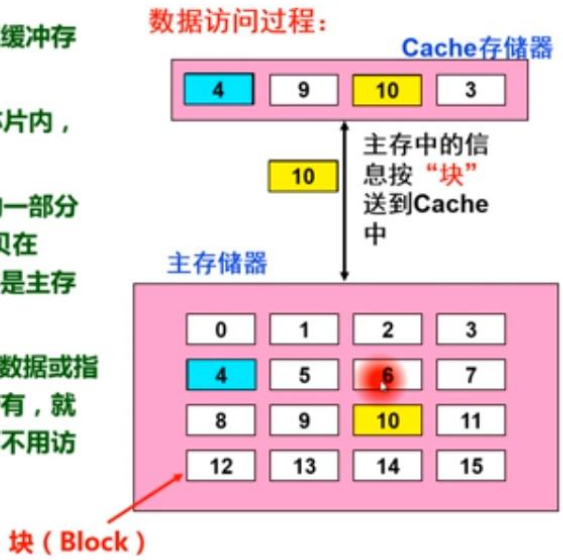
- 大量典型程序的运行情况分析结果表明
  - 在较短时间间隔内，程序产生的地址往往集中在一个很小范围内

  这种现象称为程序访问的局部性：空间局部性、时间局部性
- 程序具有访问局部性特征的原因
  - 指令：指令按序存放，地址连续，循环程序段或子程序段重复执行
  - 数据：连续存放，数组元素重复、按序访问
- 为什么引入Cache会加快访存速度？
  - 在CPU和主存之间设置一个快速小容量的存储器，其中总是存放最活跃（被频繁访问）的程序和数据，由于程序访问的局部性特征，大多数情况下，CPU能直接从这个高速缓存中取得指令和数据，而不必访问主存。

cache

Cache(高速缓存)是什么样的？

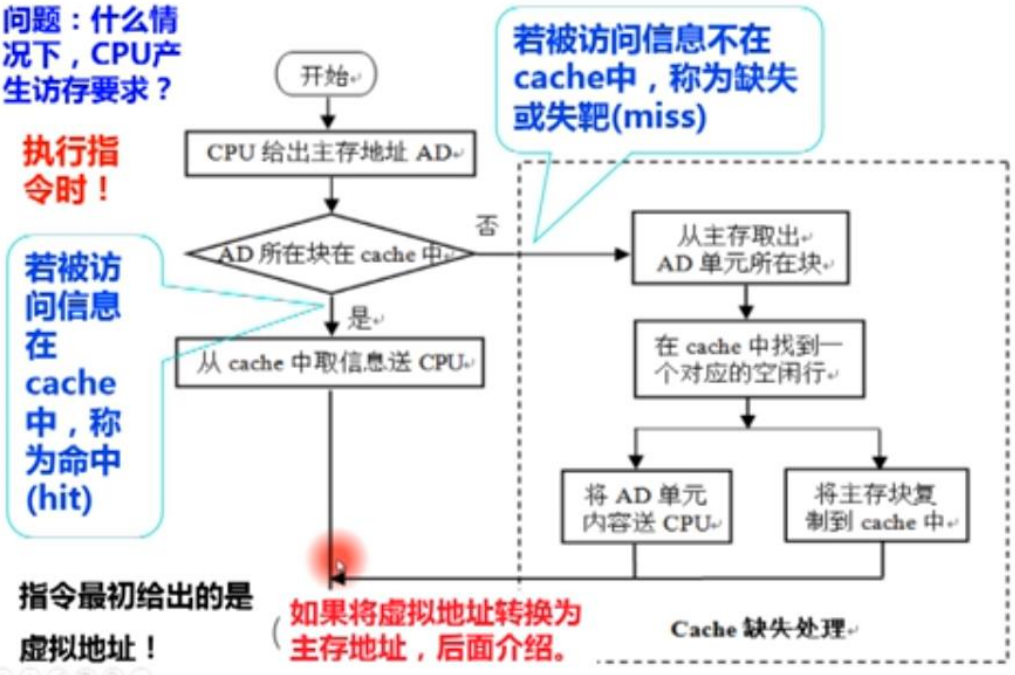
- Cache是一种小容量高速缓冲存储器，它由SRAM组成。
- Cache直接制作在CPU芯片内，速度几乎与CPU一样快。
- 程序运行时，CPU使用的一部分数据/指令会预先成批拷贝在Cache中，Cache的内容是主存储器中部分内容的映象。
- 当CPU需要从内存读(写)数据或指令时，先检查Cache，若有，就直接从Cache中读取，而不用访问主存储器。



Cache 的操作过程

问题：什么情况下，CPU产生访存要求？

执行指令时！



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)

指令最初给出的是虚拟地址！

如果将虚拟地址转换为主存地址，后面介绍。

Cache（高速缓存）的实现

问题：要实现Cache机制需要解决哪些问题？

如何分块？

主存块和Cache之间如何映射?

Cache已满时，怎么办？

写数据时怎样保证Cache和MM的一致性？

如何根据主存地址访问到cache中的数据？……

主存被分成若干大小相同的块，称为主存块(Block)，Cache也被分成相同大小的块，称为Cache行（line）或槽（Slot）。

问题：Cache对程序员(编译器)是否透明？为什么？

是透明的，程序员(编译器)在编写/生成高级或低级语言程序时无需了解Cache是否存在或如何设置，感觉不到cache的存在。

但是，对Cache深入了解有助于编写出高效的程序！

Cache映射(Cache Mapping)

- 什么是Cache的映射功能？
  - 把访问的局部主存区域取到Cache中时，该放到Cache的何处？
  - Cache行比主存块少，多个主存块映射到一个Cache行中
- 如何进行映射？
  - 把主存空间划分成大小相等的主存块（Block）
  - Cache中存放一个主存块的对应单位称为槽（Slot）或行（line）

    有书中也称之为块（Block），有书称之为页（page）（不妥！）
  - 将主存块和Cache行按照以下三种方式进行映射
    - 直接(Direct)：每个主存块映射到Cache的固定行
    - 全相联(Full Associate)：每个主存块映射到Cache的任一行
    - 组相联(Set Associate)：每个主存块映射到Cache固定组中任一行

直接映射Cache组织示意图

假定数据在主存和Cache间的传送单位为512B。

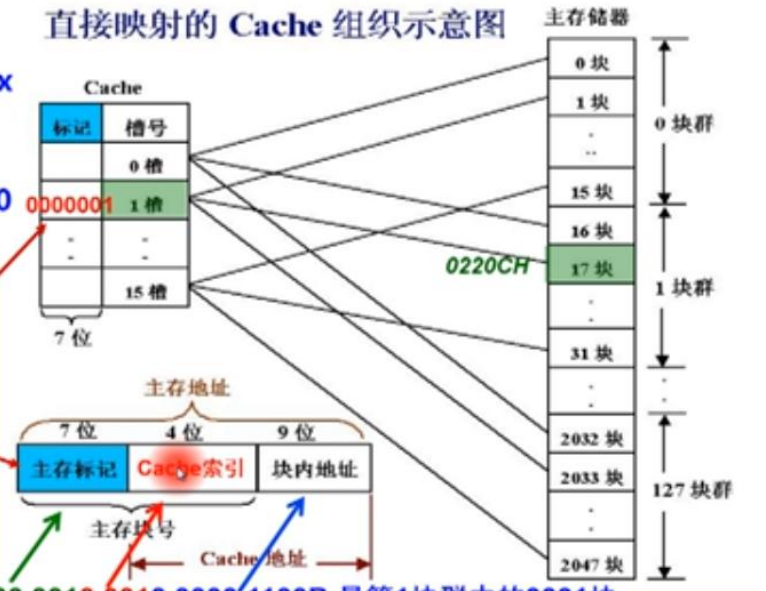
Cache大小：
2^{13}B=8KB=16行 x 512B/ 行

主存大小：
2^{20}B=1024KB=2048块 x 512B/块

Cache标记(tag)指出对应行取自哪个主存块群

指出对应地址位于哪个块群

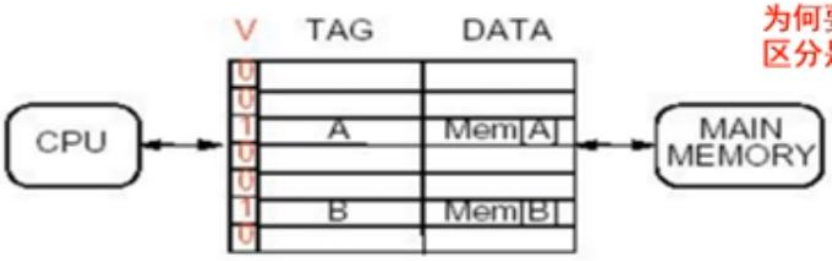
例：如何对0220CH单元进行访问？



0000 0010 0010 0000 1100B 是第1块群中的0001块（即第17块）中第12个单元！

有效位（Valid Bit）

为何要用有效位来区分是否有效？



- V为有效位，为1表示信息有效，为0表示信息无效
- 开机或复位时，使所有行的有效位V=0
- 某行被替换后使其V=1
- 某行装入新块时 使其V=1
- 通过使V=0来冲刷Cache（例如：进程切换时，DMA传送时）
- 通常为操作系统设置“cache冲刷”指令，因此，cache对操作系统程序员不是透明的！

The Simplest Cache: Direct Mapped Cache

° Direct Mapped Cache (直接映射Cache举例)

- 把主存的每一块映射到一个固定的Cache行（槽）
- 也称模映射(Module Mapping)
- 映射关系为：

块（行）都从0开始编号

Cache行号=主存块号 mod Cache行数

举例：4=100 mod 16 （假定Cache共有16行）

(说明：主存第100块应映射到Cache的第4行中。)

◆ 特点：

- 容易实现，命中时间短
- 无需考虑淘汰（替换）问题
- 但不够灵活，Cache存储空间得不到充分利用，命中率低

例如，需将主存第0块与第16块同时复制到Cache中时，由于它们都只能复制到Cache第0行，即使Cache其它行空闲，也有一个主存块不能写入Cache。这样就会产生频繁的 Cache装入。

全相联映射Cache组织示意图

假定数据在主存和Cache间的传送单位为512字。

Cache大小：2^{13}字=8K字=16行 x 512字/ 行

主存大小：2^{20}字=1024K字=2048块 x 512字/ 块

Cache标记（tag）指出对应行取自哪个主存块

主存tag指出对应地址位于哪个主存块

如何对01E0CH单元进行访问？

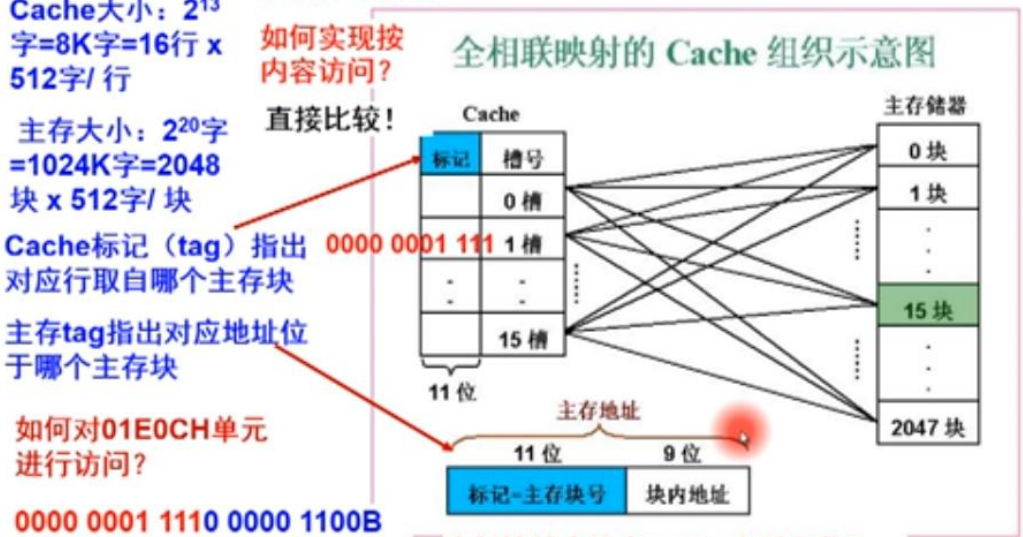
0000 0001 1110 0000 1100B
是第15块中的第12个单元！

按内容访问，是相联存取方式！

每个主存块可装到Cache任一行中。

如何实现按内容访问？

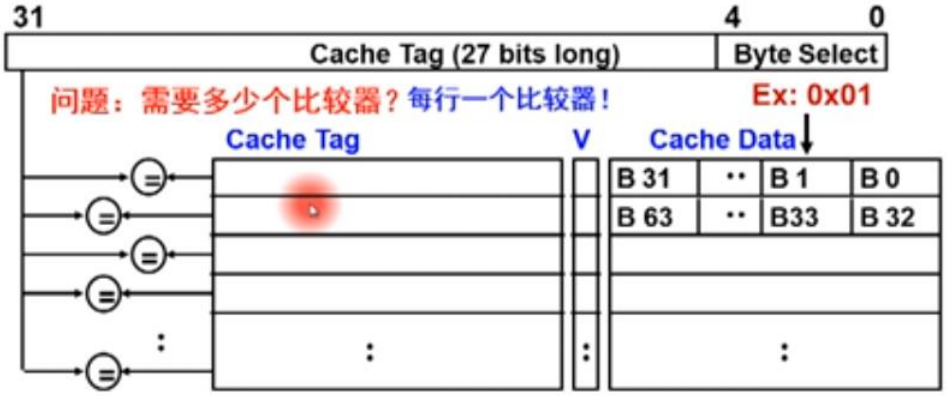
直接比较！



为何地址中没有cache索引字段？
因为可映射到任意一个cache行中！

举例：Fully Associative

- Fully Associative Cache
  - 无需Cache索引，为什么？ 因为同时比较所有Cache项的标志
- By definition: Conflict Miss = 0
  - (没有冲突缺失，因为只要有空闲Cache块，都不会发生冲突)
- Example: 32bits memory address, 32 B blocks. 比较器位数多长？
  - we need N 27-bit comparators



假定数据在主存和Cache间的传送单位为512字。

Cache大小：2^{13}字=8K字=16行 x 512字/ 行

主存大小：2^{20}字=1024K字=2048块 x 512字/ 块

指出对应行取自哪个主存组群

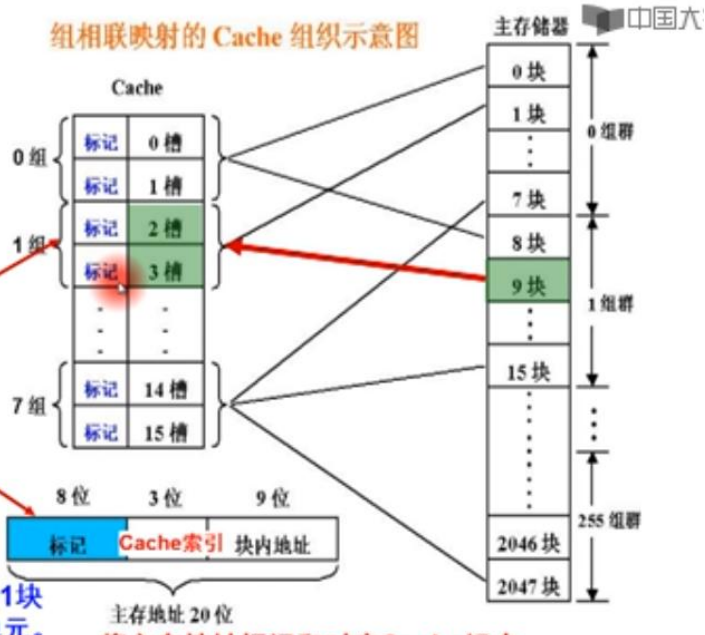
指出对应地址位于哪个主存组群中

例：如何对0120CH单元进行访问？

0000 0001 0010 0000

1100B是第1组群中的001块（即第9块）中第12个单元。

所以，映射到第一组中。



将主存地址标记和对应Cache组中每个Cache标记进行比较！

命中率、缺失率、缺失损失

° Hit: 要访问的信息在Cache中
  • Hit Rate(命中率)：在Cache中的概率
  • Hit Time (命中时间) ：在Cache中的访问时间，包括：
    Time to determine hit/miss + Cache access time
    (即： 判断时间 + Cache访问)
° Miss: 要找的信息不在Cache中
  • Miss Rate (缺失率) = 1 - (Hit Rate)
  • Miss Penalty (缺失损失)：访问一个主存块所花时间
° Hit Time << Miss Penalty （Why?）

SRAM　　DRAM，且要访问一整块

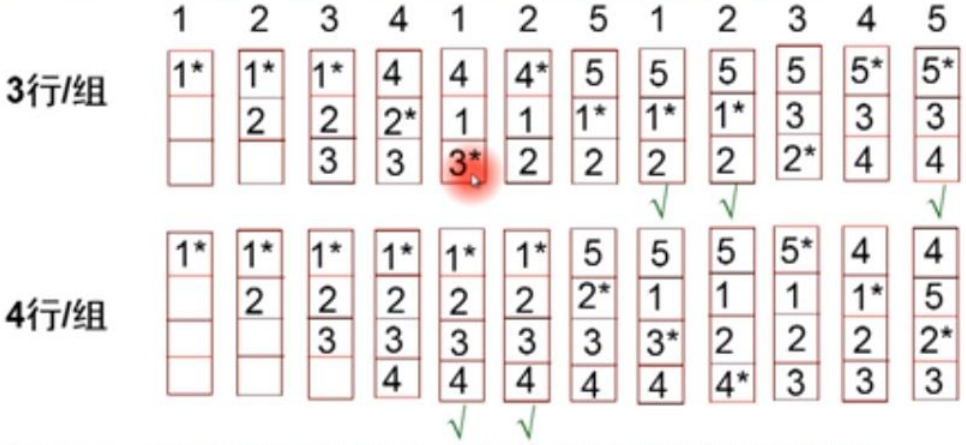
替换算法-先进先出（FIFO）

° 总是把最先进入的那一块淘汰掉。

总是把最先从图书馆搬来的书还回去！

例：假定主存中的5块{1,2,3,4,5}同时映射到Cache同一组中，对于同一地址流，考察3行/组、 4行/组的情况。

注：通常一组中含有2^k行，这里3行/组主要为了简化问题而假设



由此可见，FIFO不是一种栈算法，即命中率并不随组的增大而提高。

替换算法-最近最少用

即：计数值为0的行中的主存块最常被访问，计数值为3的行中的主存块最不经常被访问，先被淘汰！

° 计数器变化规则：

- 每组4行时，计数器有2位。计数值越小则说明越被常用。
- 命中时，被访问行的计数器置0，比其低的计数器加1，其余不变。
- 未命中且该组未满时，新行计数器置为0，其余全加1。
- 未命中且该组已满时，计数值为3的那一行中的主存块被淘汰，新行计数器置为0，其余加1。

1	2	3	4	1	2	5	1	2	3	4	5
0 1	1 1	2 1	3 1	0 1	1 1	2 1	0 1	1 1	2 1	3 1	0 5
	0 2	1 2	2 2	3 2	0 2	1 2	2 2	0 2	1 2	2 2	3 4
		0 3	1 3	2 3	3 3	0 5	1 5	2 5	3 5	0 4	2 3
			0 4	1 4	2 4	3 4	3 4	3 4	0 3	1 3	1 2

The Need to Replace! (何时需要替换？)

° Direct Mapped Cache:

- 映射唯一，毫无选择，无需考虑替换

° N-way Set Associative Cache:

- 每个主存数据有N个Cache行可选择，需考虑替换

° Fully Associative Cache:

- 每个主存数据可存放到Cache任意行中，需考虑替换

结论：若Cache miss in a N-way Set Associative or Fully Associative Cache，则可能需要替换。其过程为：

- 从主存取出一个新块
- 选择一个有映射关系的空Cache行
- 若对应Cache行被占满时又需调入新主存块，则必须考虑从Cache行中替换出一个主存块

写策略（Cache一致性问题）

- 处理Cache读比Cache写更容易，故指令Cache比数据Cache容易设计
- 对于写命中，有两种处理方式
  - Write Through (通过式写、写直达、直写)
    - 同时写Cache和主存单元
    - What!!! How can this be? Memory is too slow(>100Cycles)?
      10%的存储指令使CPI增加到：1.0+100x10%=11
    - 使用写缓冲（Write Buffer）
  - Write Back (一次性写、写回、回写)
    - 只写cache不写主存，缺失时一次写回，每行有个修改位（“dirty bit-脏位”），大大降低主存带宽需求，控制可能很复杂
- 对于写不命中，有两种处理方式
  - Write Allocate (写分配)
    - 将主存块装入Cache，然后更新相应单元
    - 试图利用空间局部性，但每次都要从主存读一个块
  - Not Write Allocate (非写分配)
    - 直接写主存单元，不把主存块装入到Cache

直写Cache可用非写分配或写分配
回写Cache通常用写分配

流水线处理器性能分析
https://www.cnblogs.com/doctorx/p/9735883.html

流水线处理器性能分析

- 流水线中的各个处理部件可并行工作，从而可使整个程序的执行时间缩短
- 流水线并不会缩短单条指令的执行时间（甚至会增加时间），而是提高了**指令的吞吐率**

虚拟存储系统的基本概念

- 虚拟存储技术的引入用来解决一对矛盾
  - 一方面，由于技术和成本等原因，主存容量受到限制
  - 另一方面，系统程序和应用程序要求主存容量越来越大
- 虚拟存储技术的实质
  - 程序员在比实际主存空间大得多的逻辑地址空间中编写程序
  - 程序执行时，把当前需要的程序段和相应的数据块调入主存，其他暂不用的部分存放在磁盘上
  - 指令执行时，通过硬件将逻辑地址（也称虚拟地址或虚地址）转化为物理地址（也称主存地址或实地址）
  - 在发生程序或数据访问失效(缺页)时，由操作系统实现主存和磁盘之间的信息交换
- 虚拟存储器机制由硬件与操作系统共同协作实现，涉及到操作系统中的许多概念，如进程、进程的上下文切换、存储器分配、虚拟地址空间、缺页处理等。

“主存--磁盘”层次

与“Cache--主存”层次相比：

页大小（2KB~64KB）比Cache中的Block大得多！ Why？

采用全相联映射！Why?

因为缺页的开销比Cache缺失开销大的多！缺页时需要访问磁盘（约几百万个时钟周期），而cache缺失时，访问主存仅需几十到几百个时钟周期！因此，页命中率比cache命中率更重要！“大页面”和“全相联”可提高页命中率。

通过软件来处理“缺页”！Why?

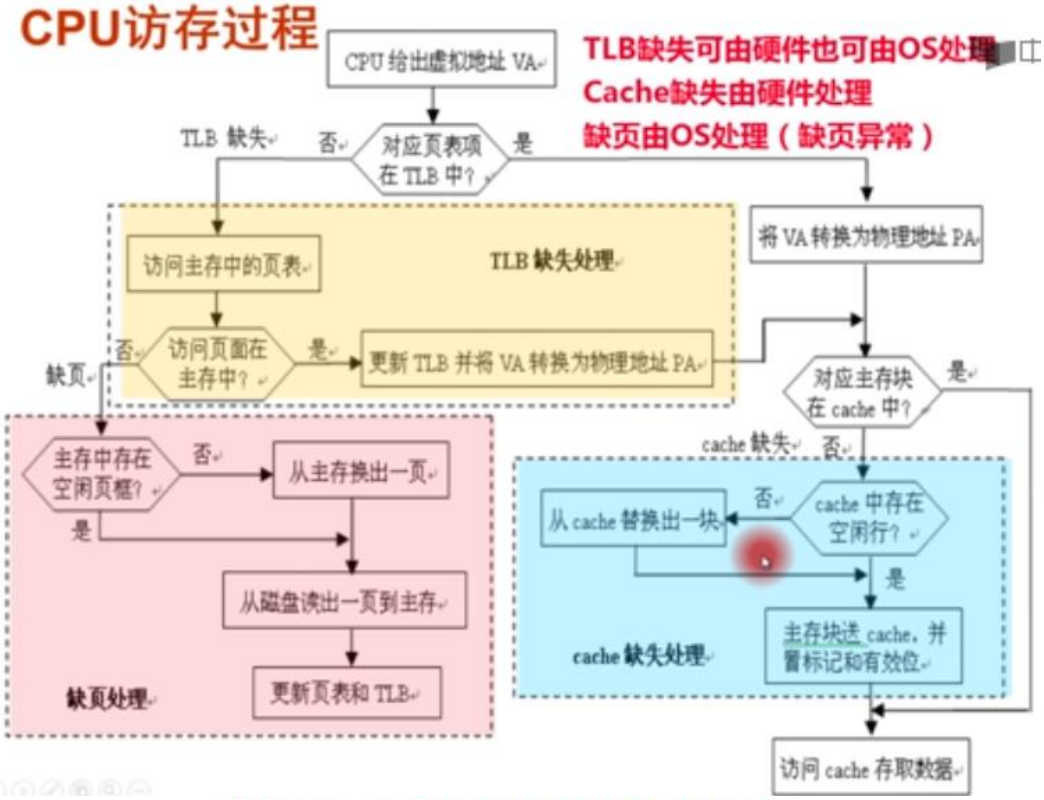
缺页时需要访问磁盘（约几百万个时钟周期），慢！不能用硬件实现。

采用Write Back写策略！ Why?

避免频繁的慢速磁盘访问操作。

地址转换用硬件实现！Why?

CPU访存过程



TLB缺失可由硬件也可由OS处理

Cache缺失由硬件处理

缺页由OS处理（缺页异常）

举例：三种不同缺失的组合

TLB	Page table	Cache	Possible? If so, under what circumstance?
hit	hit	miss	可能，TLB命中则页表一定命中，但实际上不会查页表
miss	hit	hit	可能，TLB缺失但页表命中，信息在主存，就可能在Cache
miss	hit	miss	可能，TLB缺失但页表命中，信息在主存，但可能不在Cache
miss	miss	miss	可能，TLB缺失页表缺失，信息不在主存，一定也不在Cache
hit	miss	miss	不可能，页表缺失，信息不在主存，TLB中一定没有该页表项
hit	miss	hit	同上
miss	miss	hit	不可能，页表缺失，信息不在主存，Cache中一定也无该信息

最好的情况是hit、hit、hit，此时，访问主存几次？ 不需要访问主存！

以上组合中，最好的情况是？ hit、hit、miss和miss、hit、hit 访存1次

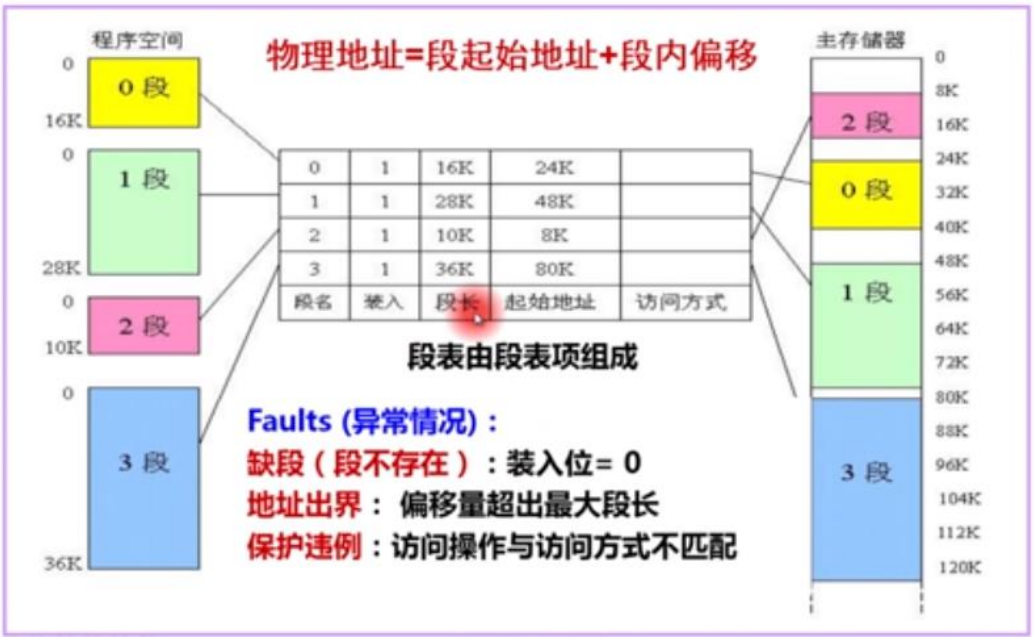
以上组合中，最坏的情况是？ miss、miss、miss 需访问磁盘、并访存至少2次

介于最坏和最好之间的是？ miss、hit、miss 不需访问磁盘、但访存至少2次

分段式虚拟存储器

- 分段系统的实现
  - 程序员或OS将程序模块或数据模块分配给不同的主存段，一个大程序有多个代码段和多个数据段构成，是按照程序的逻辑结构划分而成的多个相对独立的部分。
    （例如，代码段、只读数据段、可读写数据段等）
  - 段通常带有段名或基地址，便于编写程序、编译器优化和操作系统调度管理
  - 分段系统将主存空间按实际程序中的段来划分，每个段在主存中的位置记录在段表中，并附以“段长”项
  - 段表由段表项组成，段表本身也是主存中的一个可再定位段

段式虚拟存储器的地址映像



段页式存储器

- 段页式系统基本思想
  - 段、页式结合：
    - 程序的虚拟地址空间按模块分段、段内再分页，进入主存仍以页为基本单位
  - 逻辑地址由段地址、页地址和偏移量三个字段构成
  - 用段表和页表（每段一个）进行两级定位管理
  - 根据段地址到段表中查阅与该段相应的页表首地址，转向页表，然后根据页地址从页表中查到该页在主存中的页框地址，由此再访问到页内某数据