Ch7: Memory Hierarchy
存储器层次结构

第一讲 存储器概述和RAM芯片

第二讲 主存与CPU的连接及其读写操作

第三讲 高速缓冲存储器(cache)

第四讲 虚拟存储器

层次结构存储系统

- 分以下四个部分介绍
  - 第一讲：存储器概述和RAM芯片
  - 第二讲：主存与CPU的连接及其读写操作
    - 主存模块的连接和读写操作
    - “装入”指令和“存储”指令操作过程
  - 第三讲：高速缓冲存储器(cache)
    - 程序访问的局部性、cache的基本工作原理
    - cache行和主存块之间的映射方式
    - cache和程序性能
  - 第四讲：虚拟存储器
    - 虚拟地址空间
    - 虚拟存储器的实现
    - 存储保护

基本术语

- 记忆单元（存储基元 / 存储元 / 位元）（Cell）
  - 具有两种稳态的能够表示二进制数码0和1的物理器件
- 存储单元 / 编址单位（Addressing Unit）
  - 具有相同地址的位构成一个存储单元，也称为一个编址单位
- 存储体/ 存储矩阵 / 存储阵列（Bank）
  - 所有存储单元构成一个存储阵列
- 编址方式（Addressing Mode）
  - 字节编址、按字编址
- 存储器地址寄存器（Memory Address Register - MAR）
  - 用于存放主存单元地址的寄存器
- 存储器数据寄存器（ Memory Data Register-MDR (或MBR) ）
  - 用于存放主存单元中的数据的寄存器

存储器分类

依据不同的特性有多种分类方法

（1）按工作性质/存取方式分类

- 随机存取存储器 Random Access Memory (RAM)
  - 每个单元读写时间一样，且与各单元所在位置无关。如：内存。

  （注：原意主要强调地址译码时间相同。现在的DRAM芯片采用行缓冲，因而可能因为位置不同而使访问时间有所差别。）
- 顺序存取存储器 Sequential Access Memory (SAM)
  - 数据按顺序从存储载体的始端读出或写入，因而存取时间的长短与信息所在位置有关。例如：磁带。
- 直接存取存储器 Direct Access Memory(DAM)
  - 直接定位到读写数据块，在读写数据块时按顺序进行。如磁盘。
- 相联存储器 Associate Memory（AM）
  Content Addressed Memory (CAM)
  - 按内容检索到存储位置进行读写。例如：快表。

存储器分类

（2）按存储介质分类

半导体存储器：双极型，静态MOS型，动态MOS型

磁表面存储器：磁盘（Disk）、磁带（Tape）

光存储器：CD，CD-ROM，DVD

（3）按信息的可更改性分类

读写存储器（Read / Write Memory)：可读可写

只读存储器（Read Only Memory)：只能读不能写

（4）按断电后信息的可保存性分类

非易失（不挥发）性存储器(Nonvolatile Memory)

信息可一直保留， 不需电源维持。

（如 ：ROM、磁表面存储器、光存储器等）

易失（挥发）性存储器(Volatile Memory)

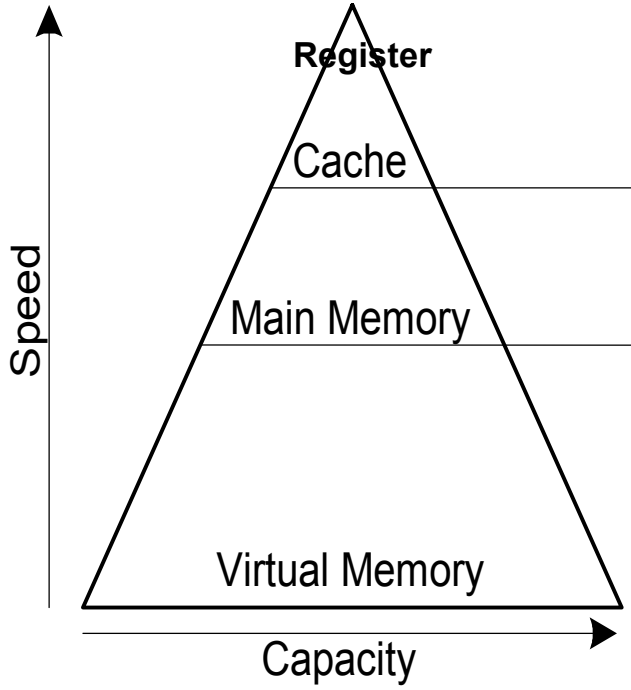
电源关闭时信息自动丢失。（如：RAM、Cache等）

存储器分类

（5）按功能/容量/速度/所在位置分类

- 寄存器(Register)
  - 封装在CPU内，用于存放当前正在执行的指令和使用的数据
  - 用触发器实现，速度快，容量小（几~几十个）
- 高速缓存(Cache)
  - 位于CPU内部或附近，用来存放当前要执行的局部程序段和数据
  - 用SRAM实现，速度可与CPU匹配，容量小（几MB）
- 内存储器MM（主存储器Main (Primary) Memory）
  - 位于CPU之外，用来存放已被启动的程序及所用的数据
  - 用DRAM实现，速度较快，容量较大（几GB）
- 外存储器AM (辅助存储器Auxiliary / Secondary Storage, 虚拟存储Virtual memory)
  - 位于主机之外，用来存放暂不运行的程序、数据或存档文件
  - 用磁表面或光存储器实现，容量大而速度慢

存储器架构（Memory Hierarchy）



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1-2	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

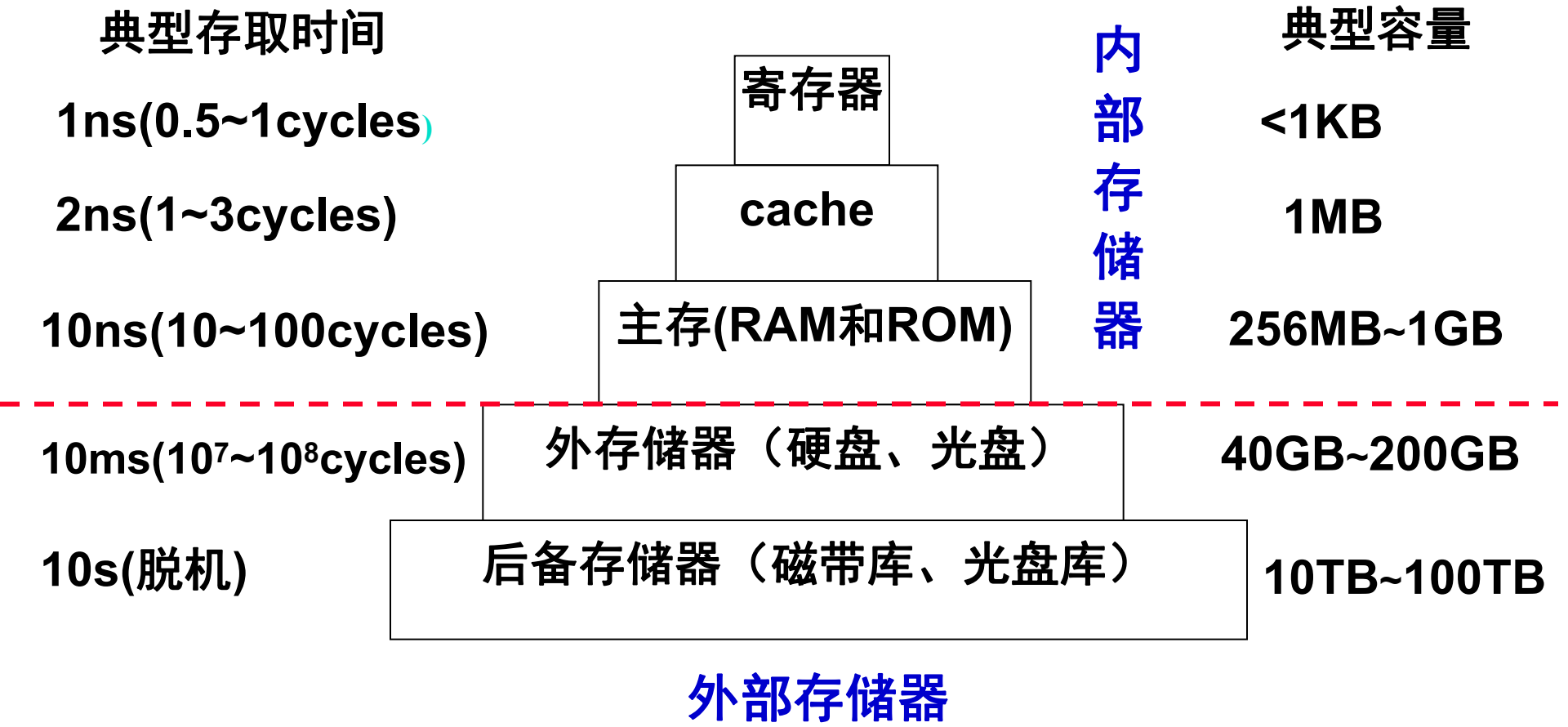
Static Random Access Memory (SRAM), 静态随机存取存储器

Dynamic Random Access Memory (DRAM), 动态随机存取存储器

Solid State Drive (SSD), 固态硬盘

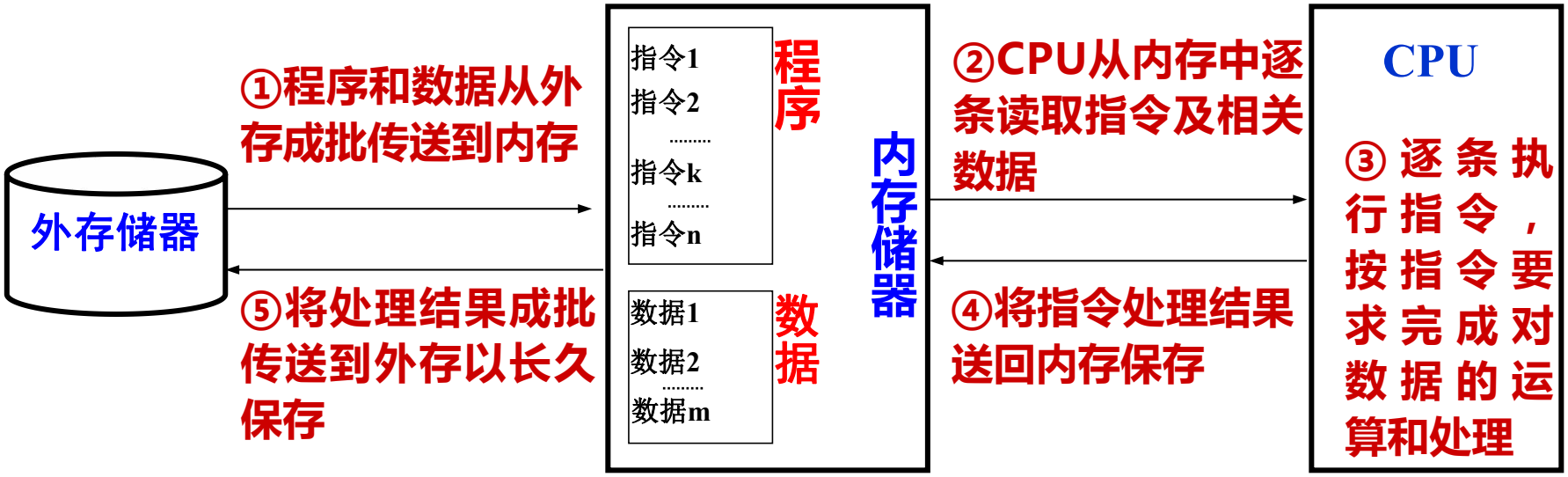
Hard Disk Drive (HDD), 硬盘驱动器

存储器的层次结构



列出的时间和容量会随时间变化，但数量级相对关系不变。

内存与外存的关系及比较



✓ 外存储器（简称外存或辅存）

– 存取速度慢

– 成本低、容量很大

– 不与CPU直接连接，先传送到内存，然后才能被CPU使用。

– 属于非易失性存储器，用于长久存放系统中几乎所有的信息

✓ 内存储器（简称内存或主存）

- 存取速度快
- 成本高、容量相对较小
- 直接与CPU连接，CPU对内存中可直接进行读、写操作
- 属于易失性存储器(volatile)，用于临时存放正在运行的程序和数据

主存的结构

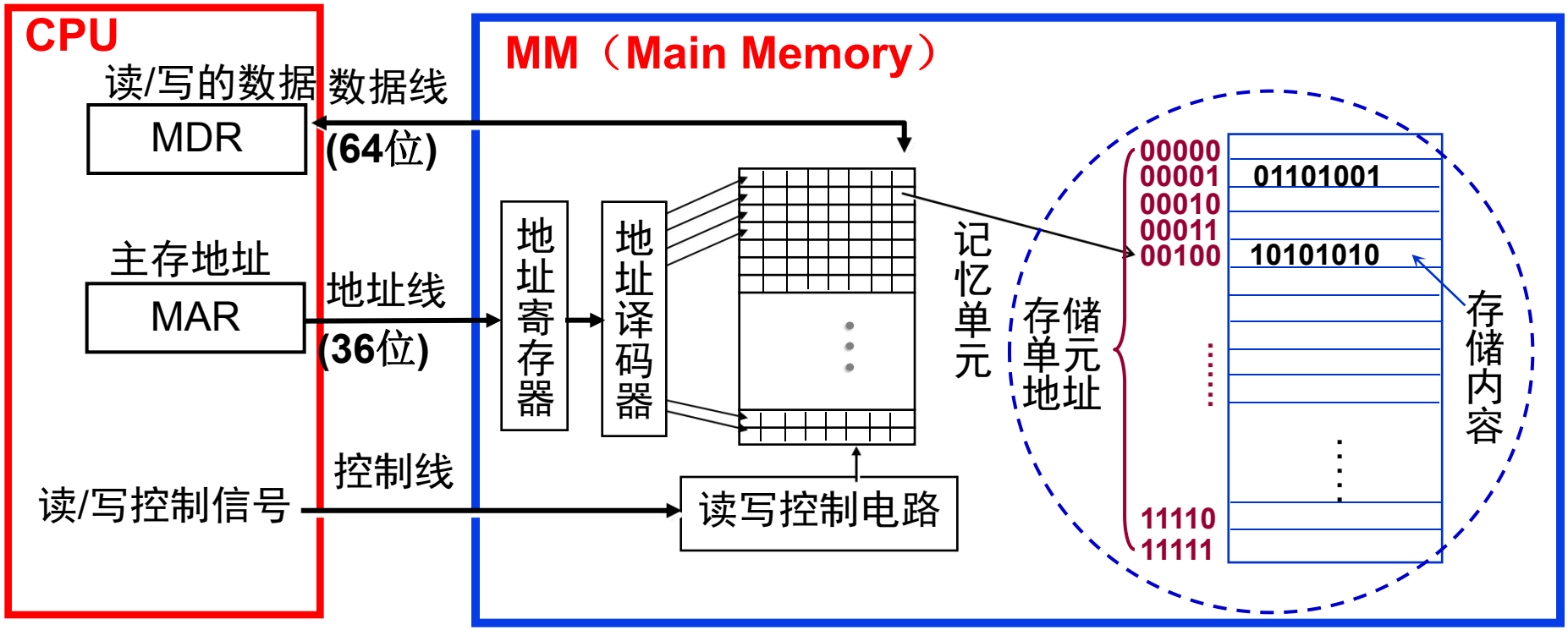
问题：主存中存放的是什么信息？指令及其数据！

CPU何时会访问主存？CPU执行指令时需要取指令、取数据、存数据！

问题：地址译码器的输入是什么？输出是什么？36位地址可寻址范围多少？

输入是地址，输出是地址驱动信号（只有一根地址驱动线被选中）。

36位地址可寻址范围为0~2^{36}-1，按字节编址时，主存地址空间为64GB



主存地址空间大小不等于主存容量（实际安装的主存大小）！

若是字节编址，则每次最多可读/写8个单元，给出的是首(最小)地址.

主存的主要性能指标

- 性能指标：
  - 按字节连续编址，每个存储单元为1个字节（8个二进位）
  - 存储容量：所包含的存储单元的容量的总和（单位：MB或GB）
  - 存取时间$T_{A(Access)}$：从CPU送出内存单元的地址码开始，主存到该地址读出数据并送到CPU所需要的时间（单位:ns，1ns = 10^{-9}s），分读取时间和(或者内存把CPU数据写入该地址所需）写入时间
  - 存储周期$T_{MC(Memory\ Cycle)}$：连读两次访问存储器所需的最小时间间隔，它应等于存取时间加上下一次存取开始前所要求的附加时间，因此，T_{MC}比T_A大（ 因为存储器由于读出放大器、驱动电路等都有一段稳定恢复时间，所以读出后不能立即进行下一次访问。 ）

（就像一趟火车运行时间和发车周期是两个不同概念一样。）

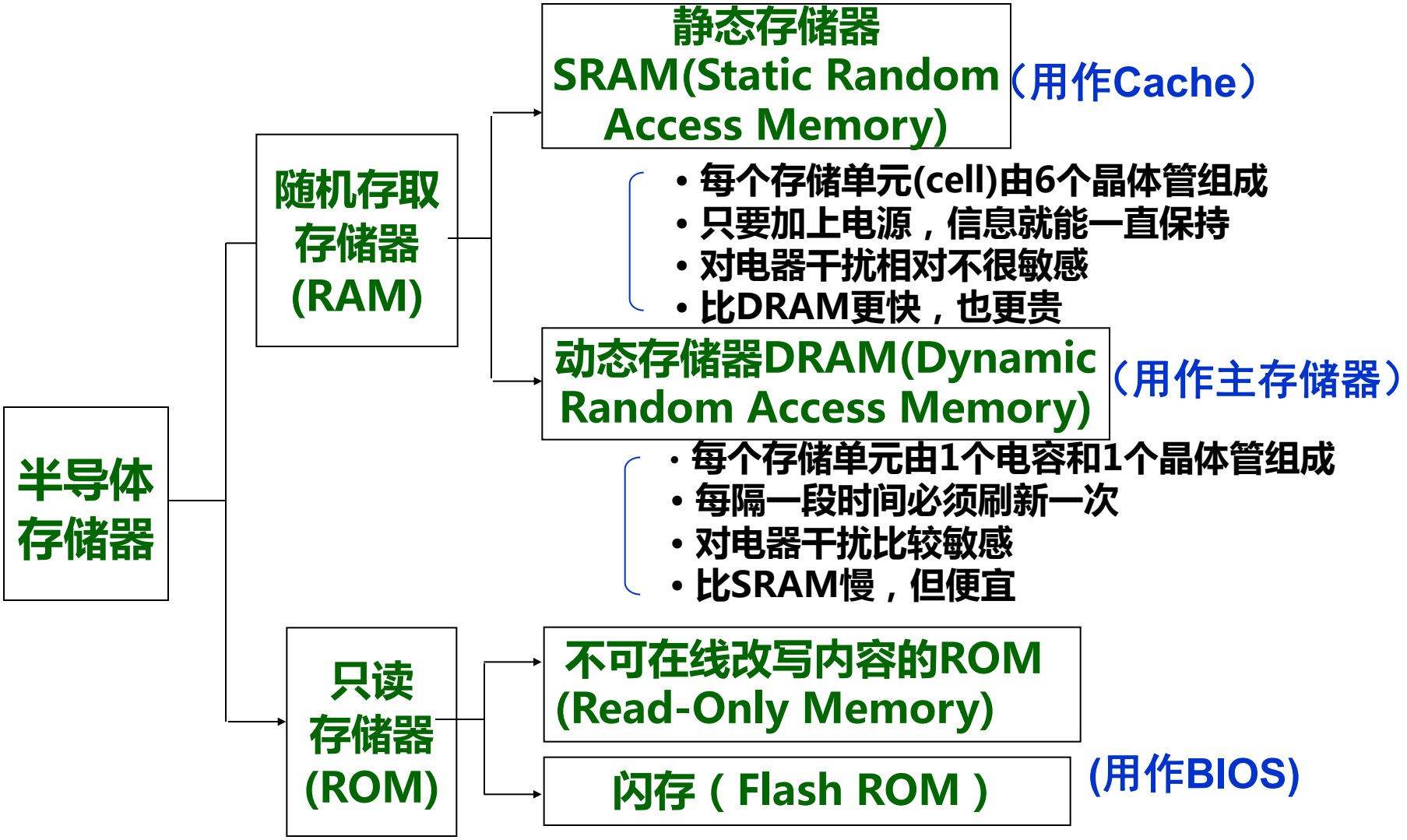
时间、存储容量（或带宽）的单位

Notations and Conventions for Numbers

Prefix	Abbreviation	Meaning	Numeric Value
mill	m	One thousandth	10^{-3}
micro	μ	One millionth	10^{-6}
nano	n	One billionth	10^{-9}
pico	p	One trillionth	10^{-12}
femto	f	One quadrillionth	10^{-15}
atta	a	One quintillionth	10^{-18}
kilo	K (or k)	Thousand	10^{3} or 2^{10}
mega	M	Million	10^{6} or 2^{20}
giga	G	Billion	10^{9} or 2^{30}
tera	T	Trillion	10^{12} or 2^{40}
peta	P	Quadrillion	10^{15} or 2^{50}
exa	E	Quintillion	10^{18} or 2^{60}

内存储器的分类及应用

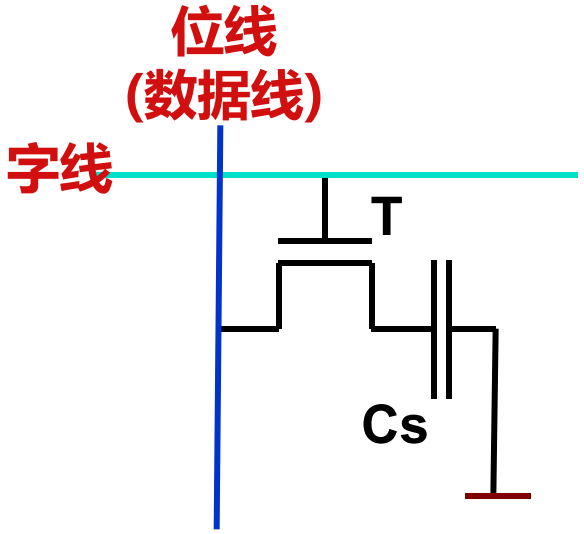
- 内存由半导体存储器芯片组成，芯片有多种类型：



动态单管记忆单元电路（不作要求）

读写原理：字线上加高电平，使T管导

- **写“0”时，数据线加低电平，使C_S上电荷对数据线放电；**
- **写“1”时，数据线加高电平，使数据线对C_S充电；**
- **读出时，数据线上有一读出电压。它与C_S上电荷量成正比。**



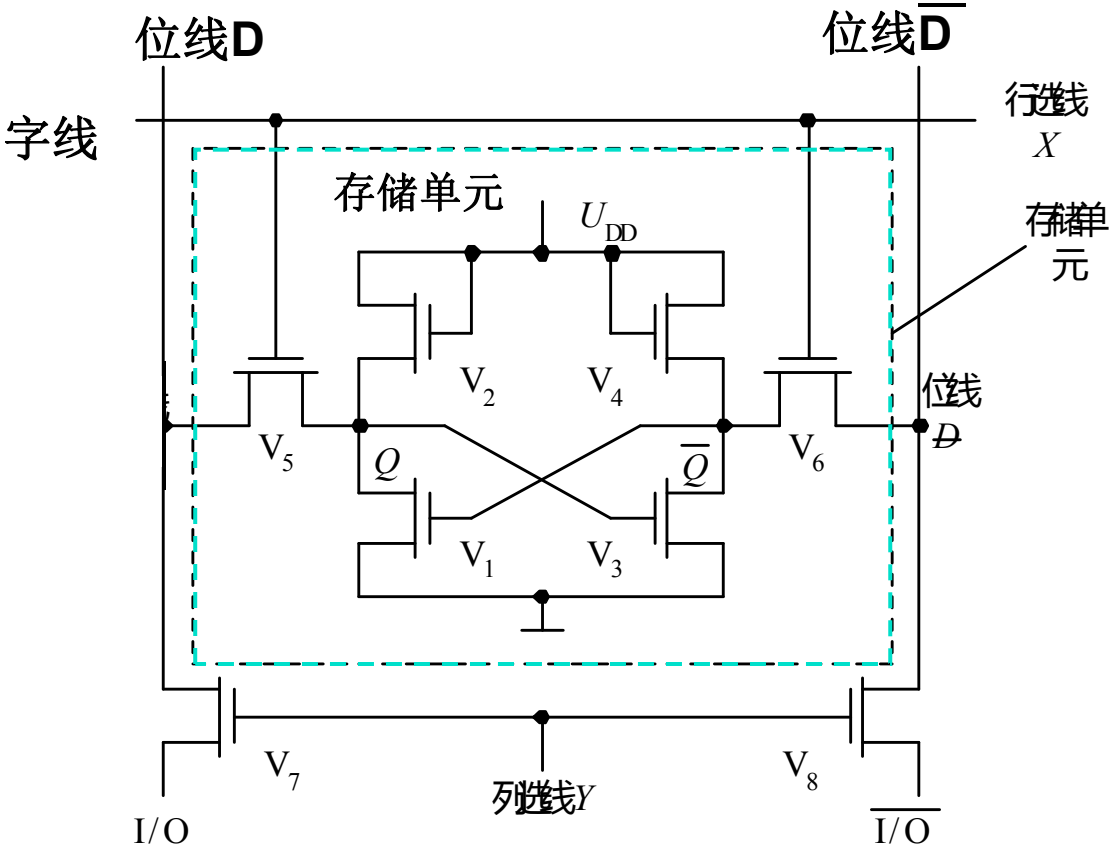
优点：电路元件少，功耗小，集成度高，用于构建主存储器

缺点：速度慢、是破坏性读出（需读后再生）、需定时刷新

刷新：DRAM的一个重要特点是，数据以电荷的形式保存在电容中，电容的放电使得电荷通常只能维持几十个毫秒左右，相当于1M个时钟周期左右，因此要定期进行刷新（读出后重新写回），按行进行（所有芯片中的同一行一起进行），刷新操作所需时间通常只占1%~2%左右。

六管静态MOS管电路（不作要求）

6管静态NMOS记忆单元



SRAM中数据保存在一对正负反馈门电路中，只要供电，数据就一直保持，不是破环性读出，也无需重写，即无需刷新！

信息存储原理： 看作带时钟的RS触发器

保持时：
- **字线为0（低电平）**

写入时：
- **位线上是被写入的二进位信息0或1**
- **置字线为1**
- **存储单元(触发器)按位线的状态设置成0或1**

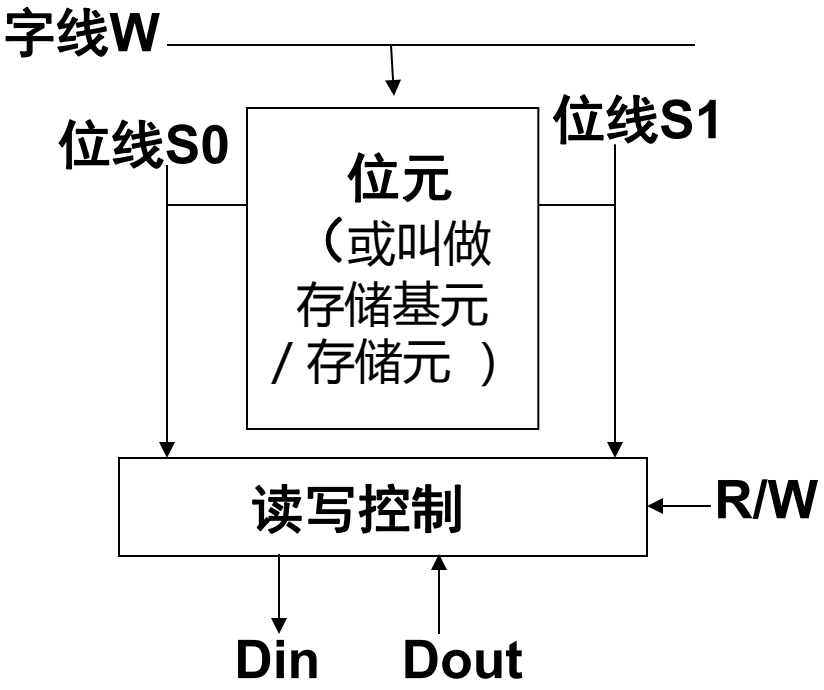
读出时：
- **置2个位线为高电平**
- **置字线为1**
- **存储单元状态不同，位线的输出不同**

半导体RAM的组织

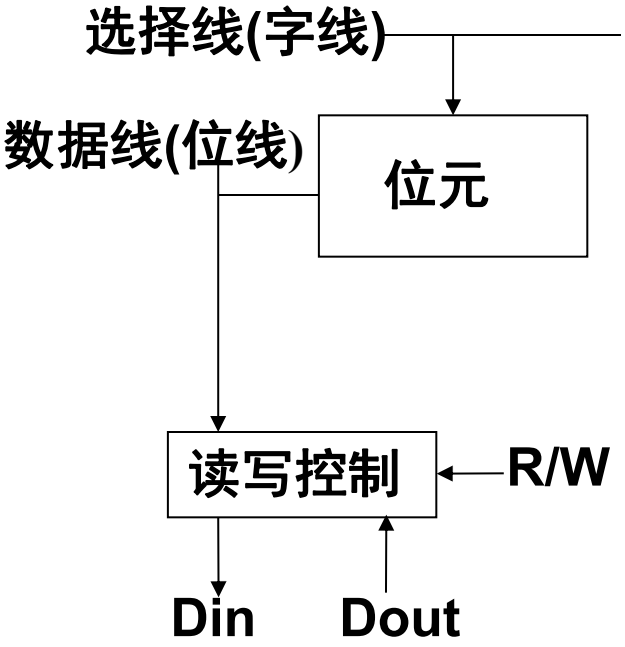
记忆单元(Cell)→存储器芯片(Chip)→ 内存条（存储器模块）

存储体(Memory Bank)： 由记忆单元(位元)构成的存储阵列

记忆单元的组织：

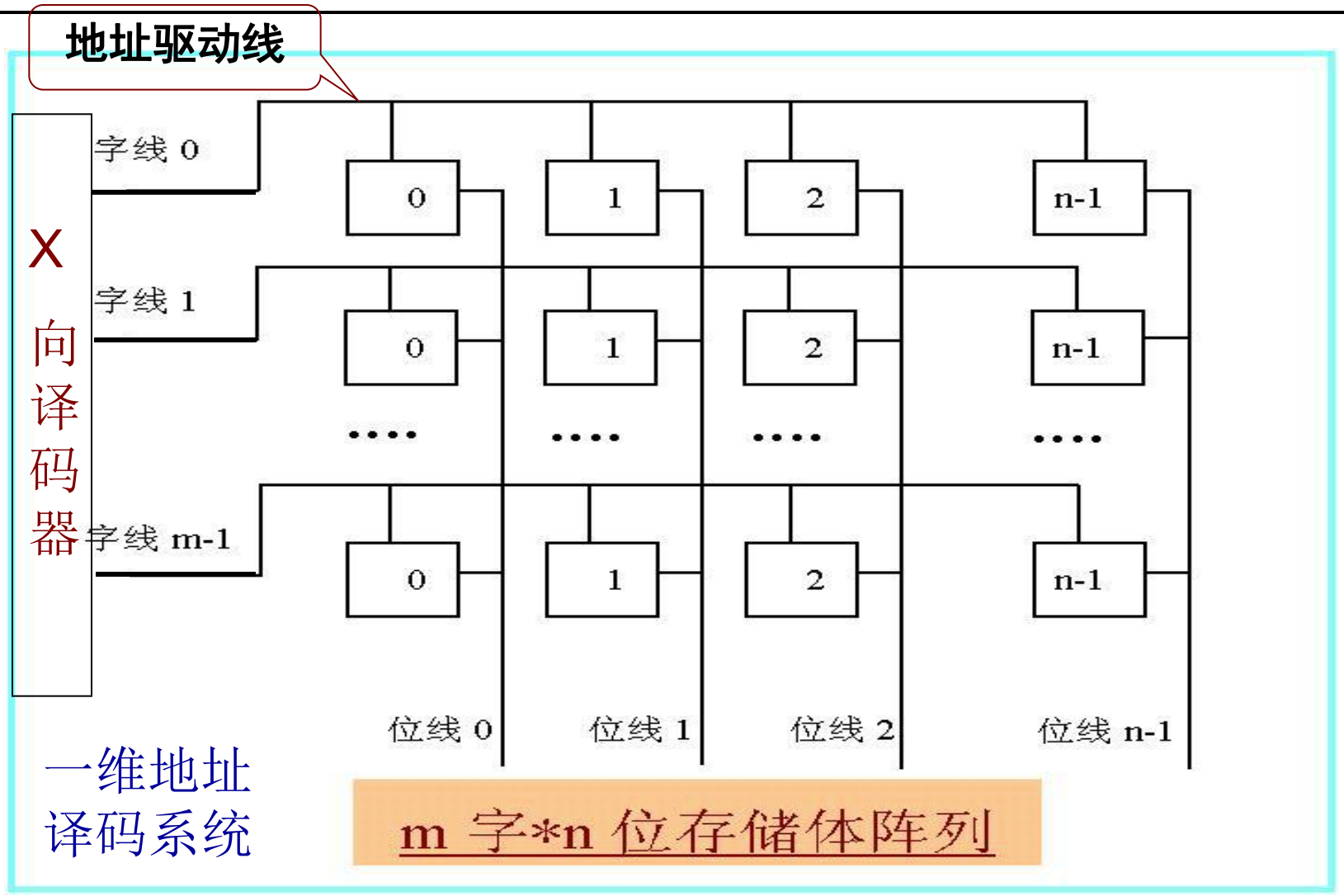


SRAM



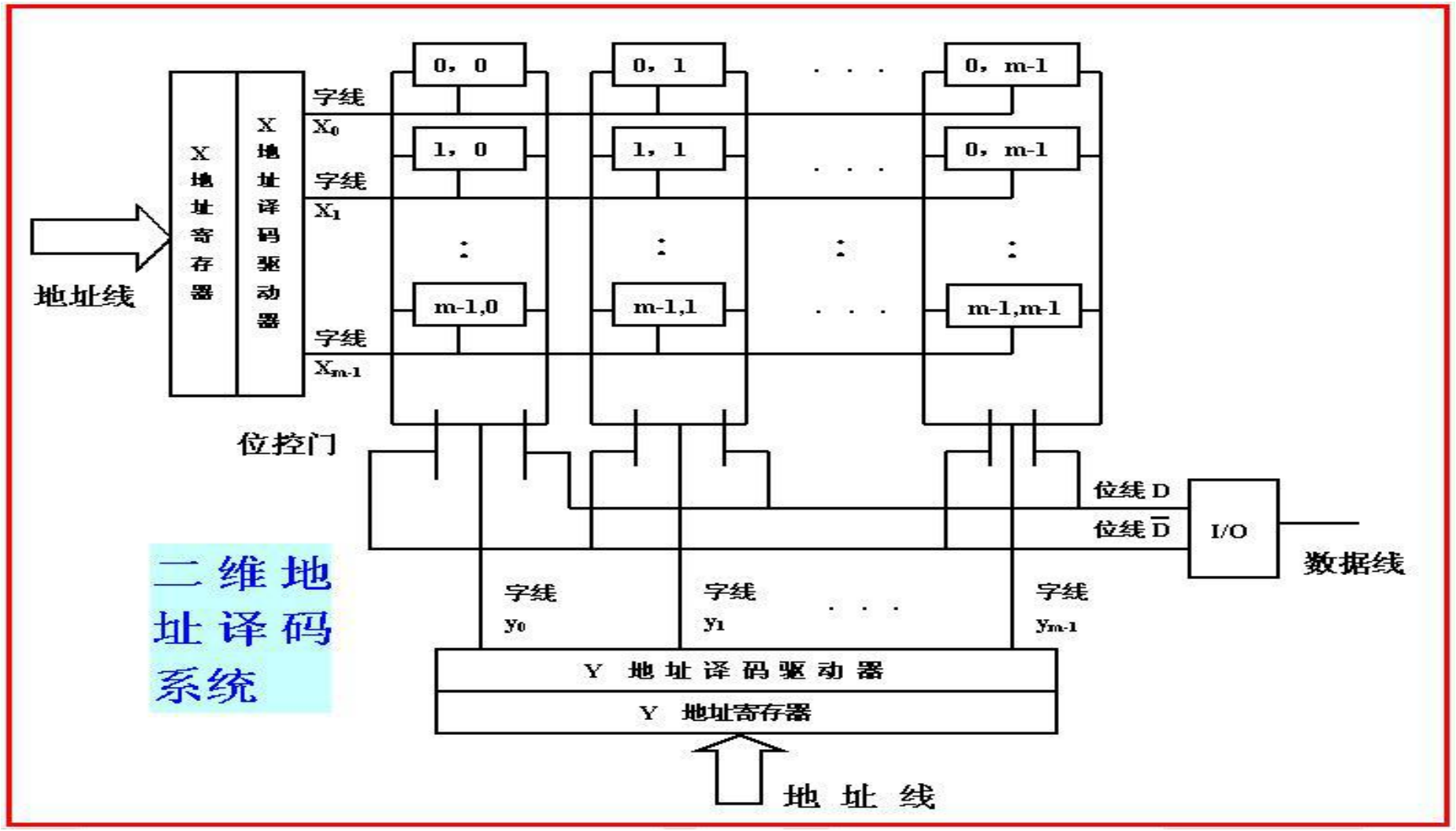
DRAM

字片式存储体阵列组织（不作要求）



一般SRAM为字片式芯片，只在x向上译码，同时读出字线上所有位！

位片式存储体阵列组织（不作要求）



位片式在字方向和位方向扩充，需要有片选信号

DRAM芯片都是位片式

举例：典型的16M位（4Mx4位）DRAM

4Mx4(简称16M位)DRAM表示的意思:由4M个不同的内存地址，每个地址提供4位数据

$2^n \times b$位DRAM芯片的存储阵列容量是[r行(地址)×c列(地址)]×b位数据

4Mb(地址)x4(位)=2^{22}(地址)*4(位)=2^{11}(地址)*2^{11}(地址)*4 (位)

= (2^{11} 行)x (2^{11}列)x4(位)=2048行x2048列x4 (位)

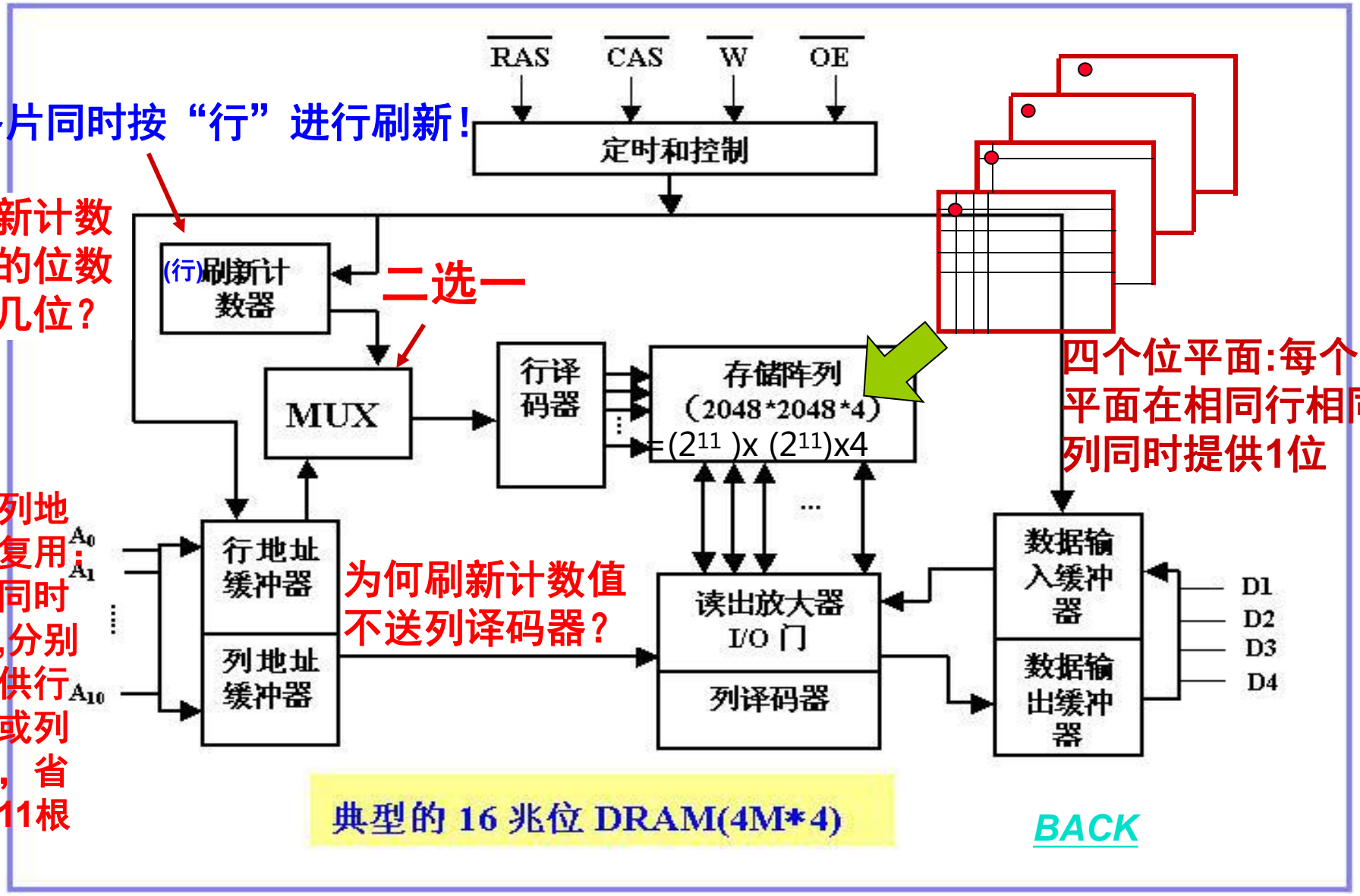
地址线：(一般不用22根线，只用)11根线，它们分时复用，分别表示2048行及2048列, 由(行)RAS和(列)CAS提供控制时序，决定何时选行,何时选列。

(2)需4个位平面,分别提供(相同)行及(相同)列交叉点的1到4位(最多4位同时)读/写 (3) 内部结构框图 SKIP、

问题： 为什么每出现新一代DRAM芯片，容量至少提高到4倍？

因为行地址和列地址分时复用,新一代DRAM芯片，至少要增加一根地址线，该地址线给行地址和列地址各增加一位，所以行数和列数各增加一倍。因而容量至少提高到2*2=4倍。

举例：典型的16M位DRAM(4Mx4)[=[(2^11)x(2^11)]x4]



DRAM芯片的刷新

- 刷新周期：从上次对整个存储器刷新结束到下次对整个存储器全部刷新一遍为止的时间间隔，也就是相邻两次对某个特定行进行刷新的时间间隔。

为电容数据有效保存期的上限10ms～100ms(毫秒)，目前公认是64ms。

多采用异步刷新方式：

将一个刷新周期分配给所有行，使得在一个刷新周期内每行至少刷新一次，且仅刷新一次。

例如: 以4096行为例，在64ms时间内必须完成对4096行中的每一行(轮流)刷新一次，即

每隔64ms/4096=15.625μs刷新一行。

CPU与存储器之间的通信方式

- CPU和主存之间有同步和异步两种通信方式
  - 异步方式（读操作）过程（需握手信号）
    - CPU送地址到地址线，主存进行地址译码
    - CPU发读命令，然后等待存储器发回“完成”信号
    - 主存收到读命令后开始读数，完成后发“完成”信号给CPU
    - CPU接收到“完成”信号，从数据线取数

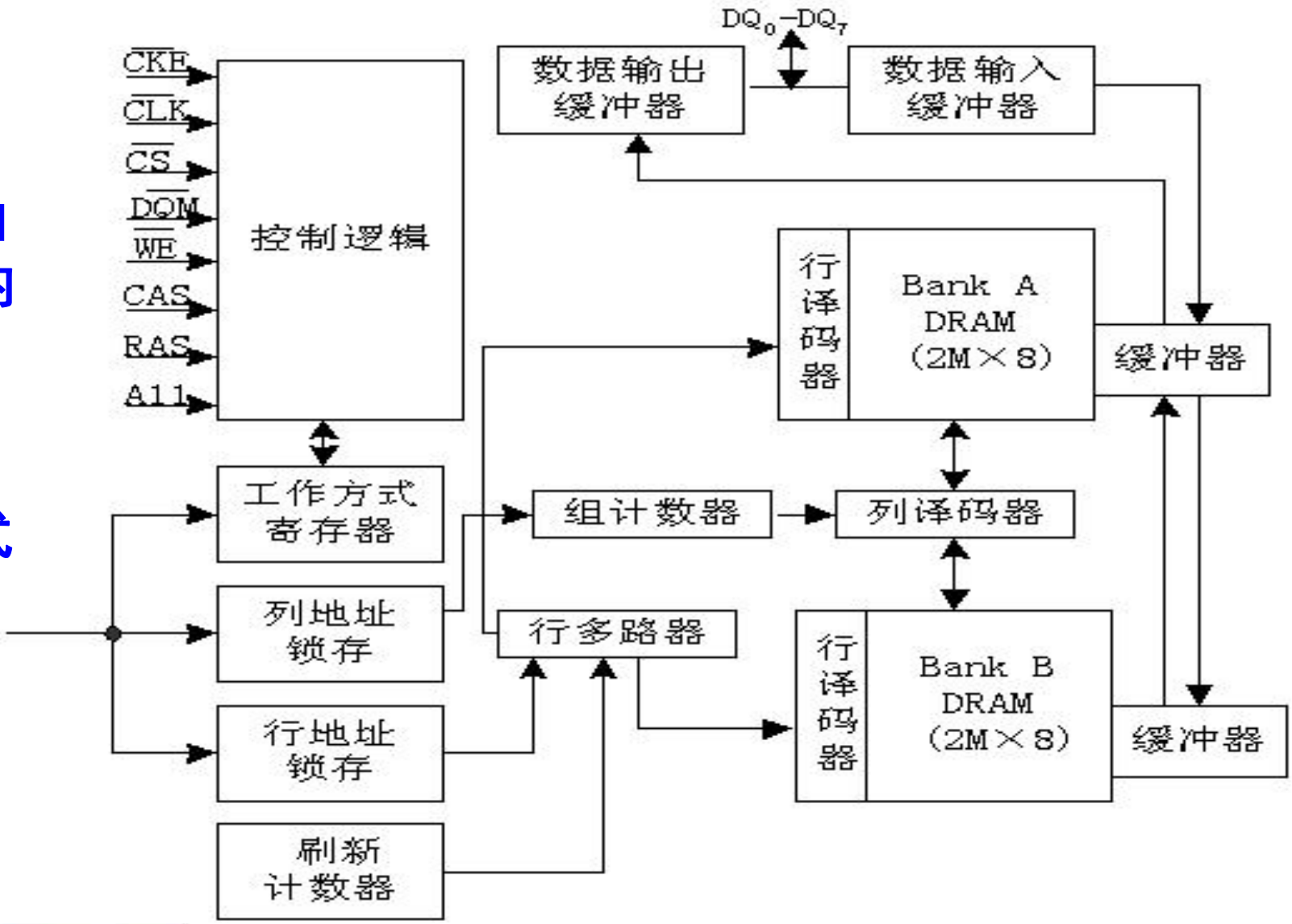
    写操作过程类似
  - 同步方式的特点
    - CPU和主存由统一时钟信号控制，无需应答信号（如“完成”）
    - 主存总是在确定的时间内准备好数据
    - CPU送出地址和读命令后，总是在确定的时间取数据
    - 存储器芯片必须支持同步方式

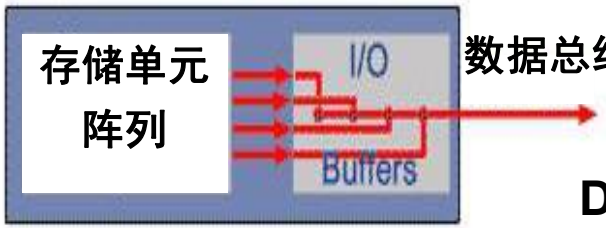
SDRAM芯片技术

- 同步动态随机存取内存(synchronous dynamic random-access memory，SDRAM）
  - 每步操作都在系统时钟控制下进行
  - 有确定的等待时间（读命令开始到数据线有效的时间, 称为CAS潜伏期(Column Address Strobe 或者Column Address Select **Latency, CAS Latency，CL),**CAS意为列地址选通脉冲），例如 CL=2 clks,
  - 连续传送（Burst）数据个数 BL=1 / 2 / 4 / 8
  - 多体(缓冲器)交叉存取
  - 利用总线时钟上升沿与下降沿同步传送

SDRAM芯片的内部结构

同步方式





数据总线

一个时钟内传送4个数据

DDR2 SDRAM

DDR3：一个时钟内传送8个数据

只读存储器

- 特点：
  - 信息只能读不能（在线）写。
  - 非破坏性读出，无需再生。
  - 也以随机存取方式工作。
  - 信息用特殊方式写入，一经写入，就可长久保存，不受断电影响。故是非易失性存储器。
- 用途：
  - 用来存放一些固定程序。如监控程序、启动程序等。只要一接通电源，这些程序就能自动地运行；
  - 可作为控制存储器，存放微程序。
  - 还可作为函数发生器和代码转换器。
  - 在输入/出设备中，被用作字符发生器，汉字库等。
  - 在嵌入式设备中用来存放固化的程序。

只读存储器(Read Only Memory)

MROM（Mask ROM）：掩膜只读存储器

PROM（Programmable ROM）：可编程只读存储器

EPROM（Erasable PROM）：可擦除可编程只读存储器

EEPROM（E^2PROM，Electrically EPROM）：

电可擦除可编程只读存储器

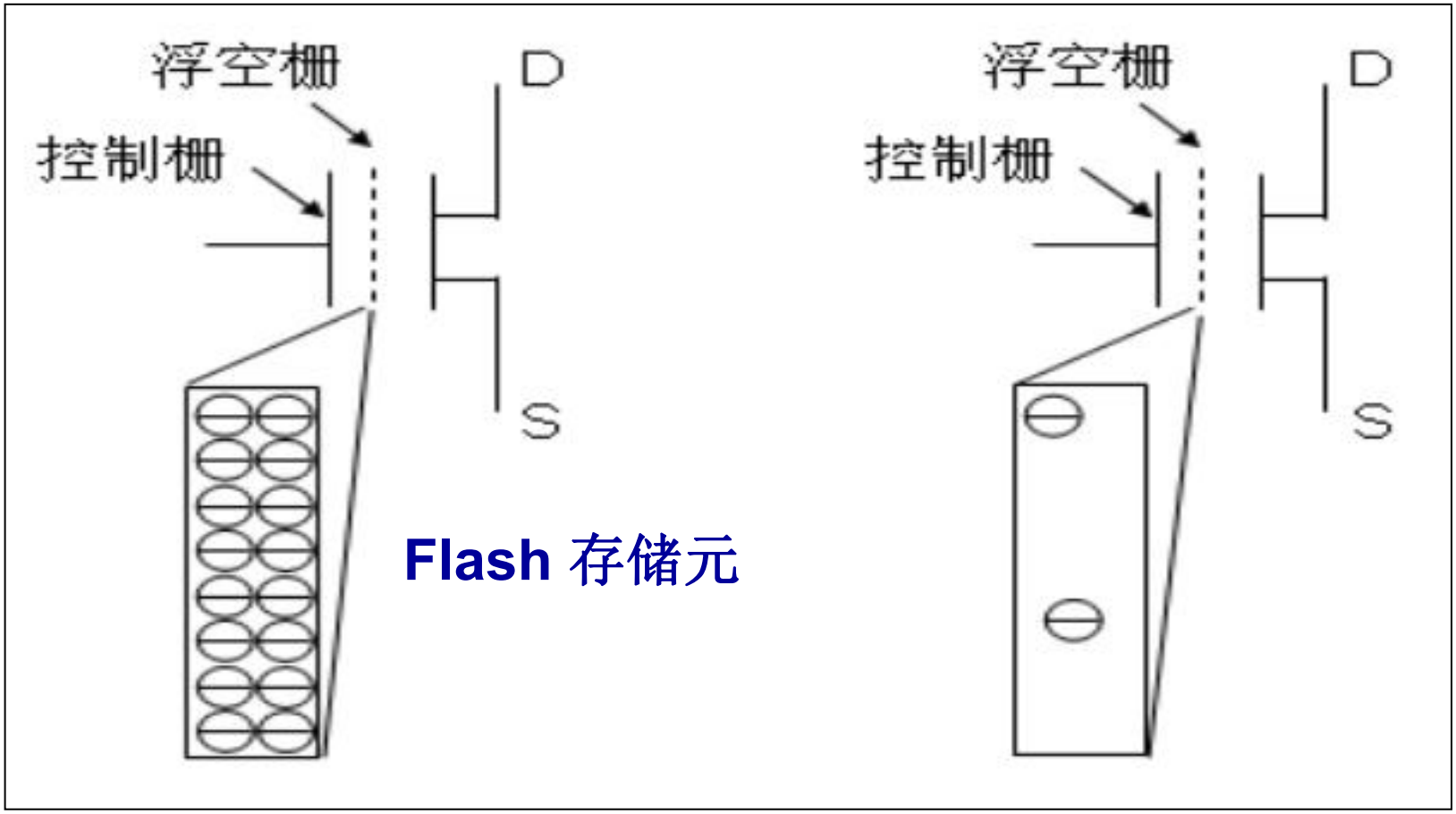
flash memory：闪存（快擦存储器）：

快擦型电可擦除重编程ROM

闪存（Flash Memory）

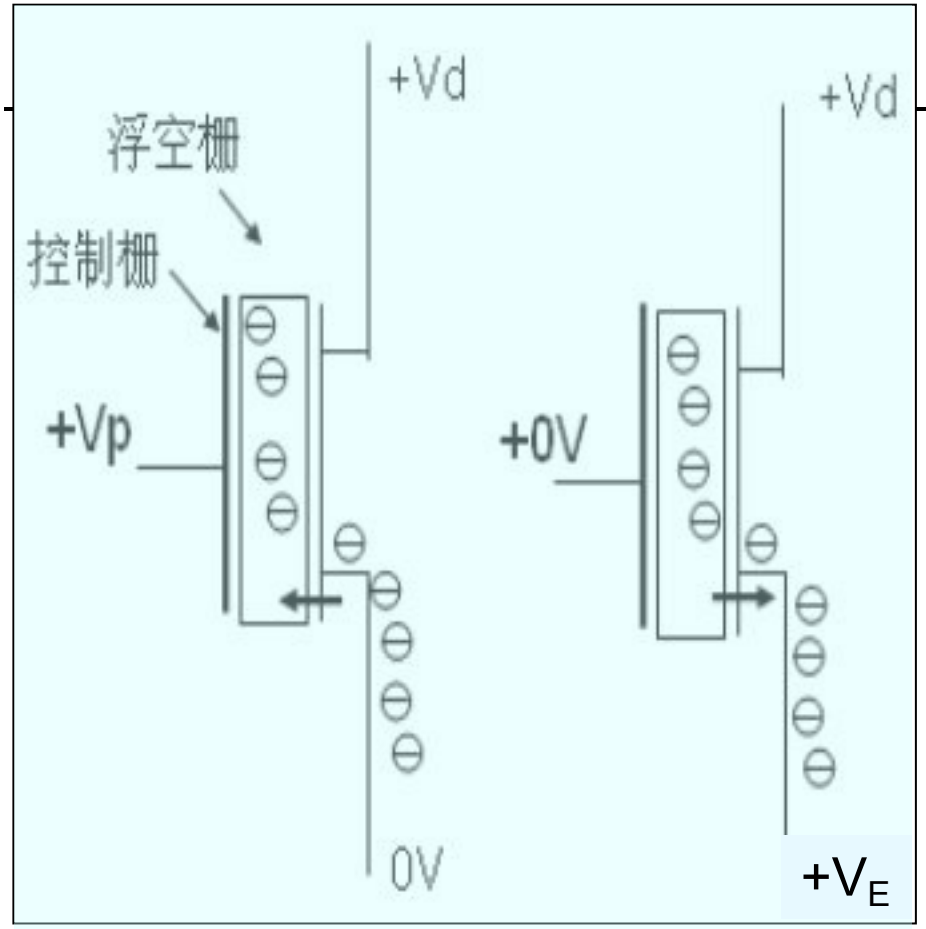
控制栅加足够正电压时，
浮空栅储存大量负电荷，
为“0”态；

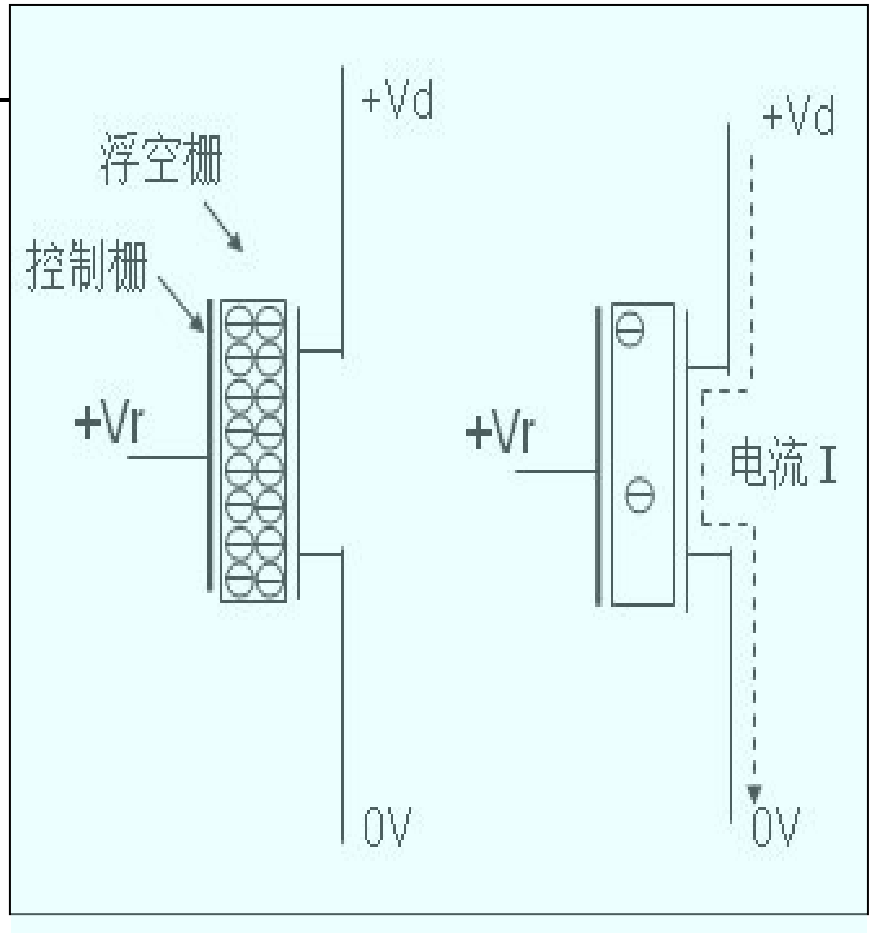
控制栅不加正电压时，
浮空栅少带或不带负电荷，
为“1”态。



(a) Flash 存储元“0”状态

(b) Flash 存储元“1”状态





(a) 编程:写“0”　(b) 擦除:写“1”　(a) 读“0”　(b) 读“1”

有三种操作：擦除、编程、读取　　读快、写慢！

写入：快擦（所有单元为1）-- 编程（需要之处写0）

读出：控制栅加正电压，若状态为0，则读出电路检测不到电流；
若状态为1，则能检测到电流。

层次结构存储系统

- 分以下四个部分介绍
  - 第一讲：存储器概述和RAM芯片
  - 第二讲：主存与CPU的连接及其读写操作
    - 主存模块的连接和读写操作
    - “装入”指令和“存储”指令操作过程
  - 第三讲：高速缓冲存储器(cache)
    - 程序访问的局部性、cache的基本工作原理
    - cache行和主存块之间的映射方式
    - cache和程序性能
  - 第四讲：虚拟存储器
    - 虚拟地址空间、虚拟存储器的实现

存储器芯片的扩展

- 字(地址字)扩展（位数不变、扩充容量）

  DRAM芯片引脚

  用16K×8位芯片扩成64K×8位存储器需几个芯片？地址范围各为什么？

  (地址)字方向扩展4倍，即4个芯片,分布包含地址:0000-3FFFH， 4000-7FFFH，8000-BFFFH， C000- FFFFH， 共16位地址，其中高两位用来选择4个芯片中的某一个,余下14位做片内地址. 由外部译码器译码生成4个输出，分别连到4个片选信号;

  - 地址线、读/写控制线等对应相接，片选信号连译码输出

- 位扩展（[地址]字数不变, 总地址数不变，位数扩展）

  用(每个容量为)4096(即4K地址)×1位(每个地址上有1位)的芯片构成4K(地址)×8位(每个地址上有8位)存储器需几个芯片？地址范围各是多少？

  位方向扩展8倍，字方向无需扩展。即8个芯片，地址范围都一样：000- FFFH，地址共12位，全部作为片内地址

  - 芯片的地址线及读/写控制线对应相接，而数据线单独引出

- 字位同时扩展（字和位同时扩展）

  用(每个容量为)(16K×4位)芯片构成64K×8位存储器需几个芯片，地址范围各是多少？

  (地址)字向[扩大]4倍、而且位向[同时扩大] 2倍，共需要4*2=8个芯片。这些芯片表示的地址范围分别为: 0000-3FFFH， 4000-7FFFH， 8000-BFFFH， C000- FFFFH

  - 地址线、读/写控制线等对应相接，片选信号则分别与外部译码器各个译码输出端相连

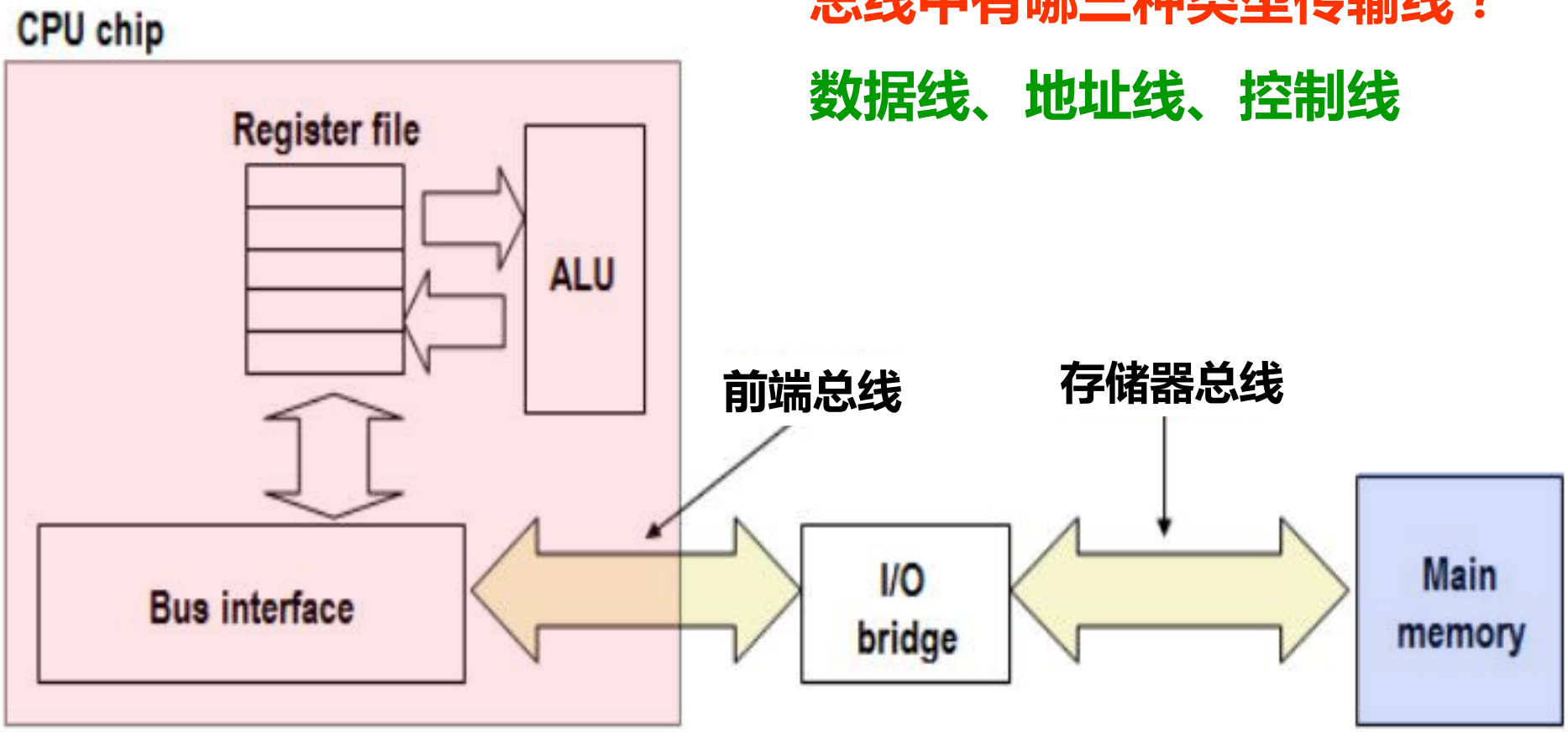
有两种容量扩展方式：交叉编址和连续编址。上述例子都是何种编址方式？连续编址！

主存模块的连接和读写操作

- 主存与CPU的连接

总线中有哪三种类型传输线？

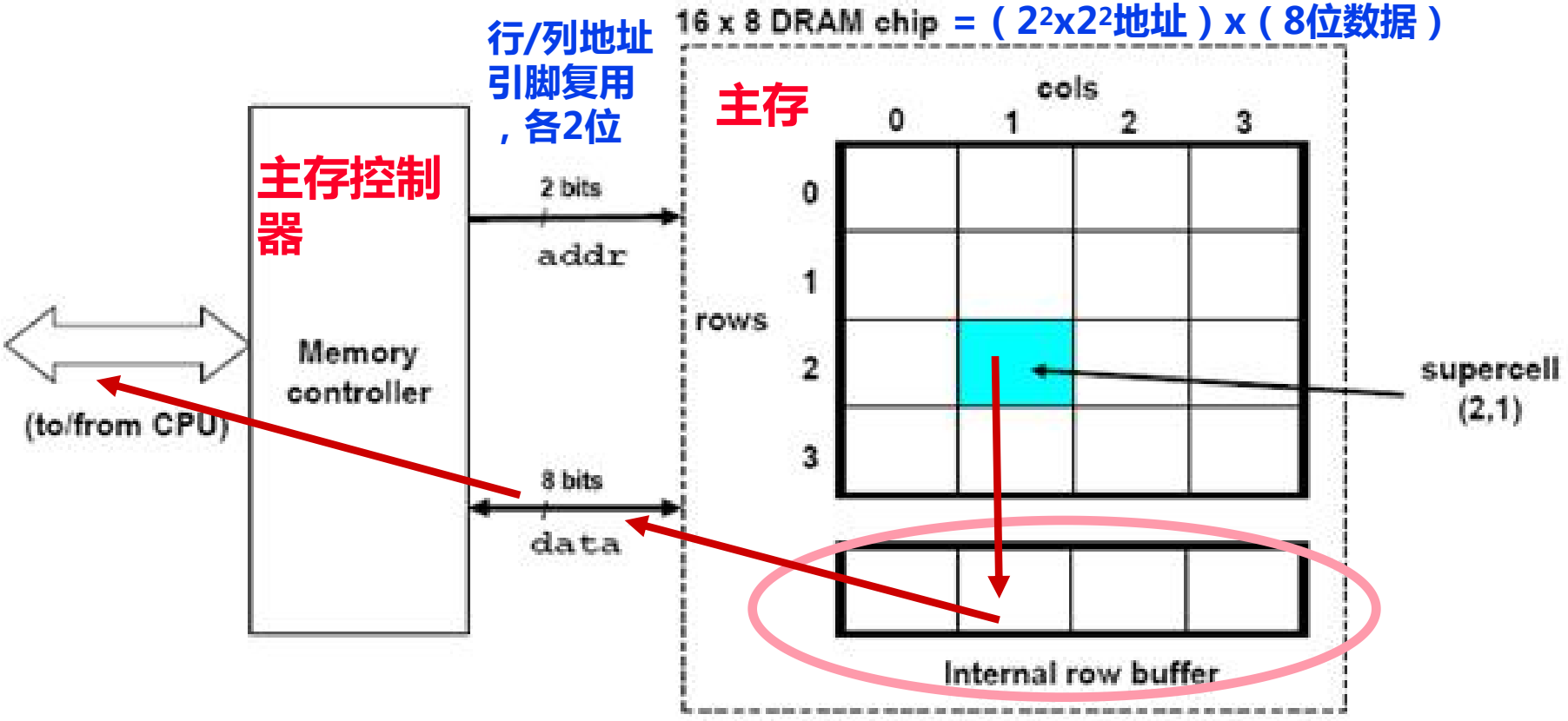
数据线、地址线、控制线



主存模块的连接和读写操作

° DRAM芯片内部结构示意图

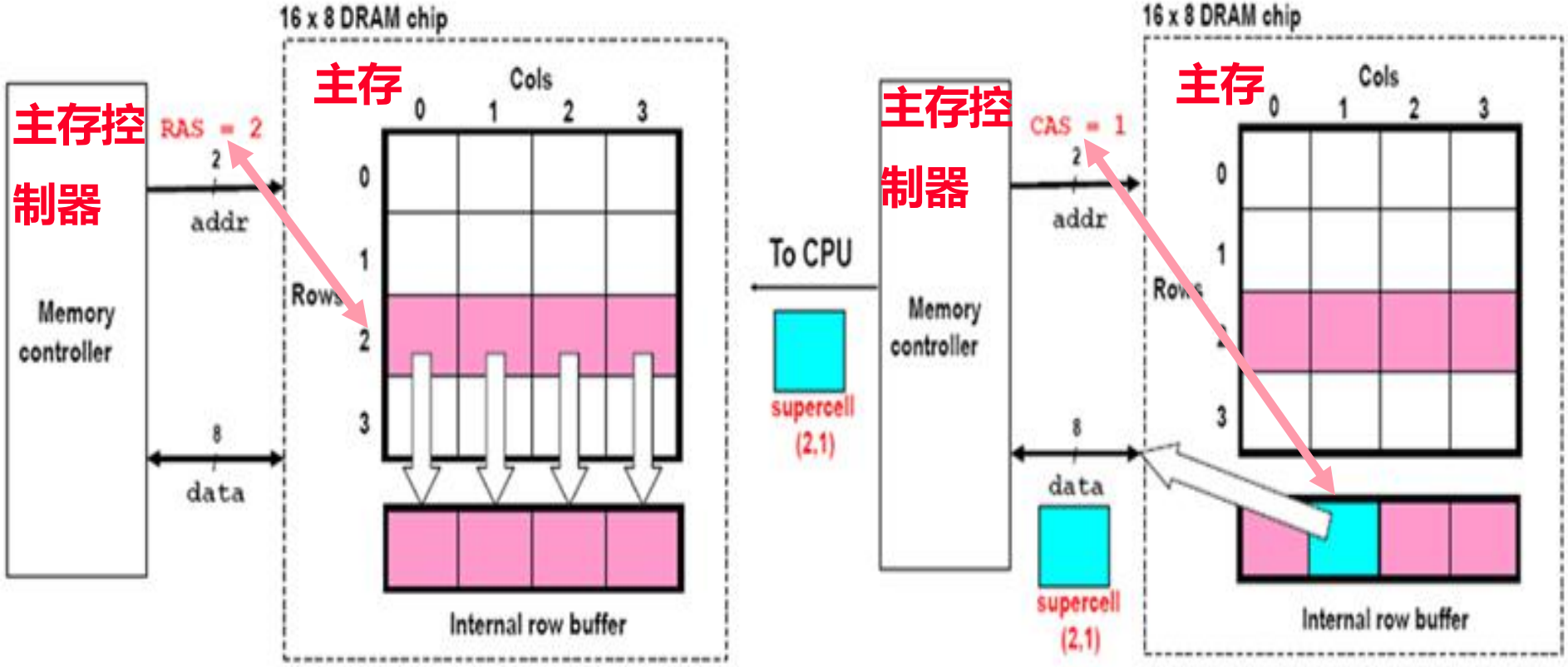
同时有多个芯片进行读写



图中芯片容量为16×8位，存储阵列为4行×4列，地址引脚采用复用方式，因而仅需2根地址引脚，每个超元（supercell）有8位，需8根数据引脚，有一个内部的行缓冲（row buffer），通常用SRAM元件实现。

主存模块的连接和读写操作

- DRAM芯片读写原理示意图

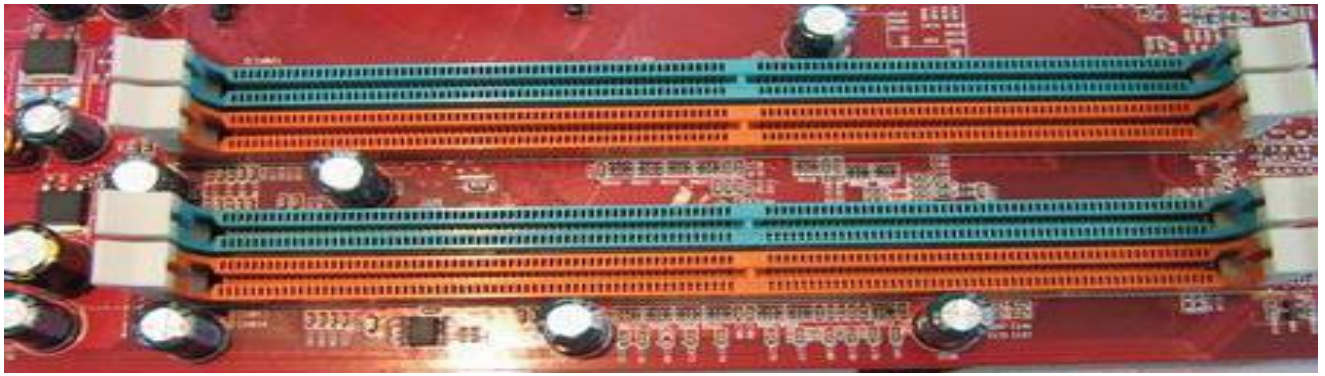


首先，存储控制器将行地址“2”送行译码器，选中第“2”行，此时，整个一行数据被送行缓冲。然后，存储控制器将列地址“1”送列译码器，选中第“1”列，此时，将行缓冲第“1”列的8位数据supercell(2,1)读到数据线，并继续送往CPU。

PC机主存储器的物理结构

- 由若干内存条组成
- 内存条的组成：

  把若干片DRAM芯片(颗粒)焊装在一小条印制电路板上制成
- 内存条必须插在主板上的内存条插槽中才能使用



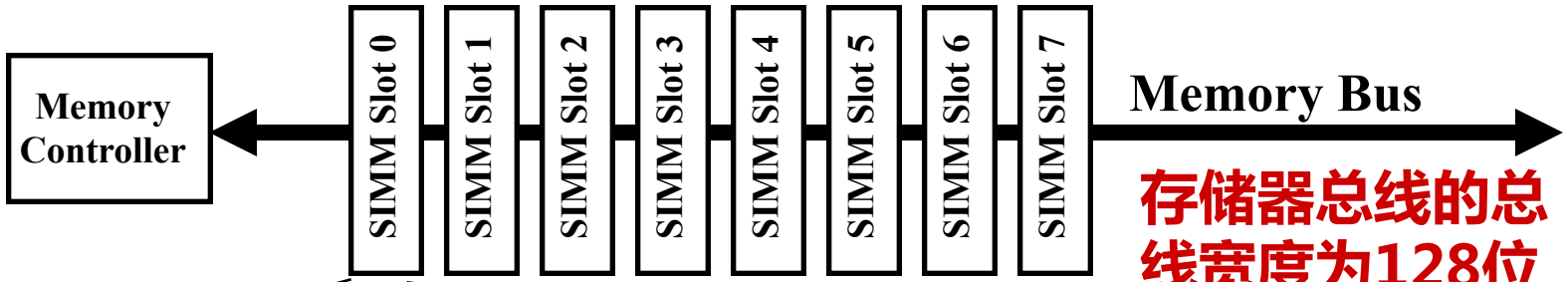
目前流行的是DDR4内存条：

- Double Date Rate SDRSM,双倍速率
- **采用双列直插式，其触点分布在内存条的两面,PC机主板中一般都配备有2个或4个DIMM插槽(**Dual-Inline-Memory-Modules，中文名叫双列直插式存储模块)

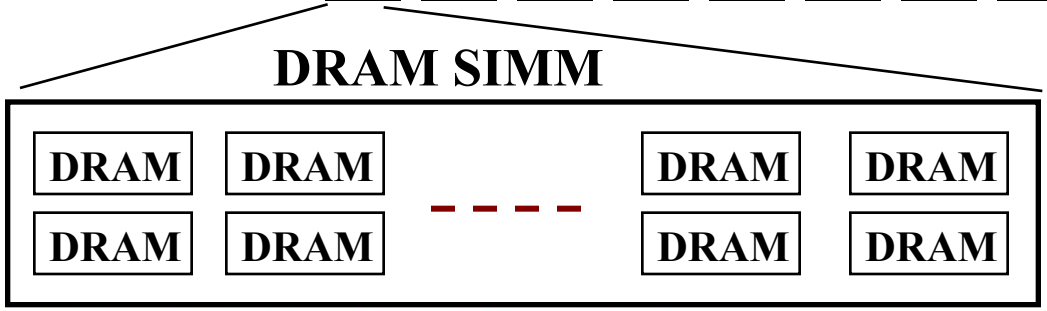
举例：SPARCstation 20’s Memory Module

总线宽度是指总线中数据线的条数

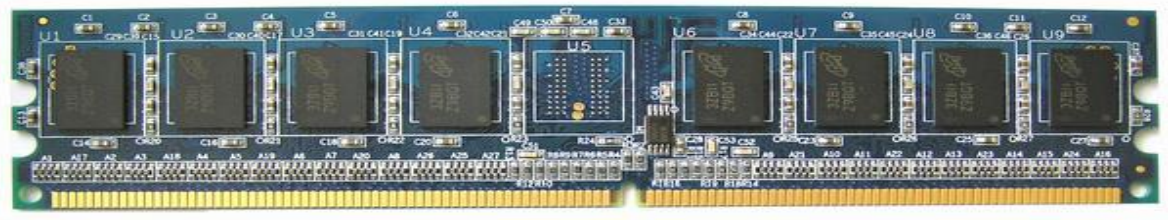
(SIMM)Single-In-line-Memory-Modules,单列直插内存模块, 由若干个 内存芯片(颗粒)集成在一小块电路板上，然后通过 **SIMM插槽与主板相连**



存储器总线的总线宽度为128位

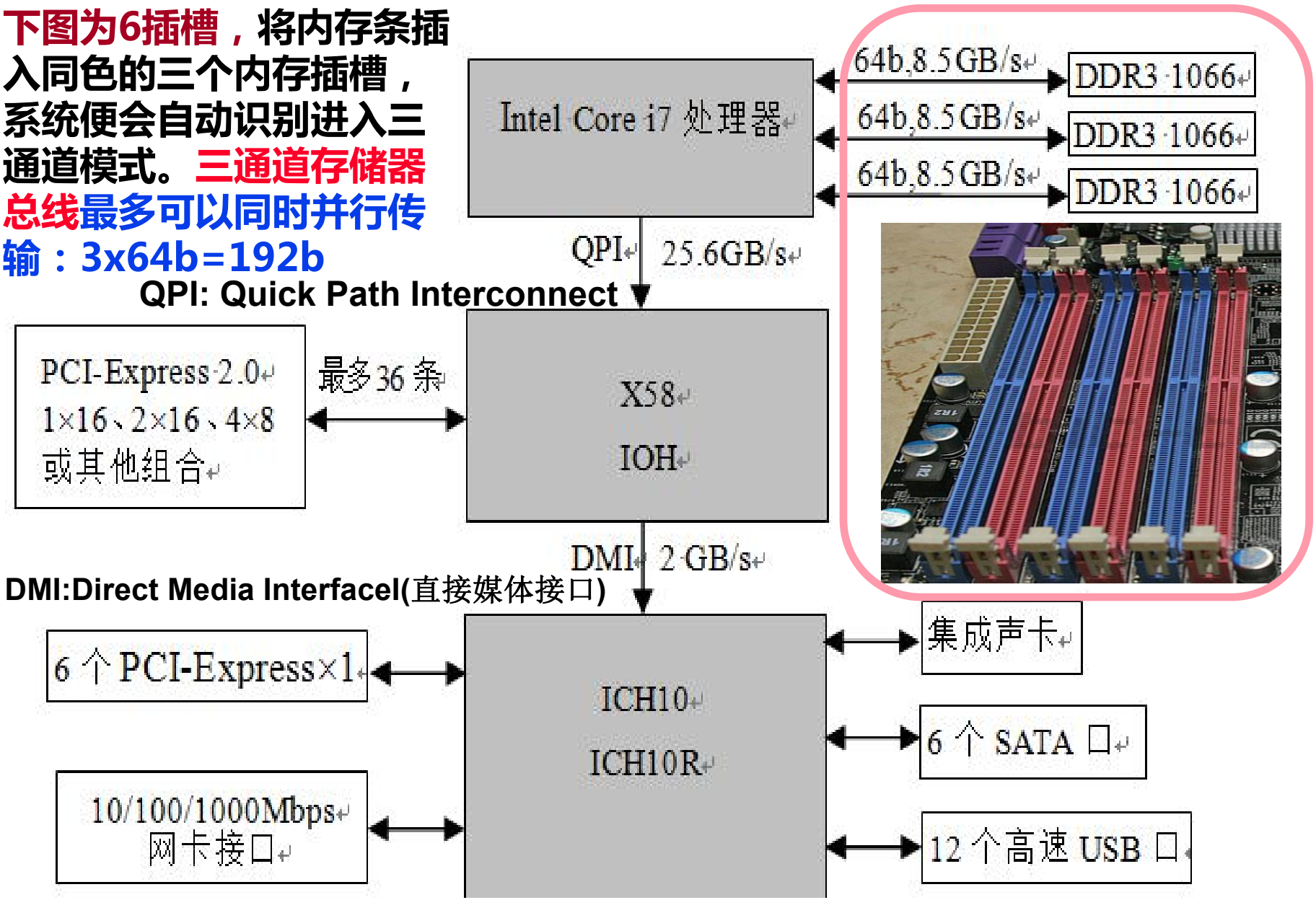


每个内存条最多能同时读出128位数据



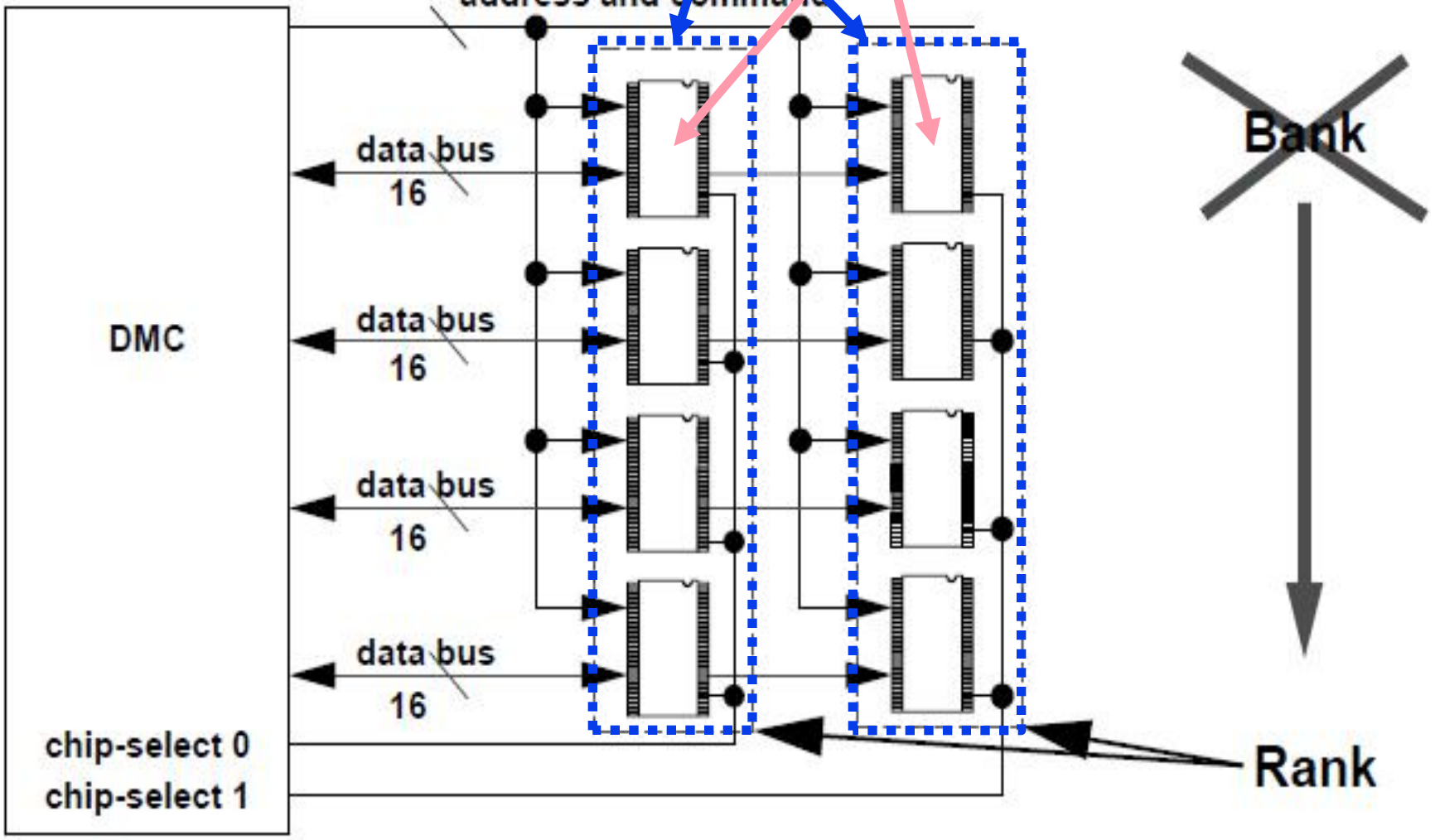
计算机系统互连

下图为6插槽，将内存条插入同色的三个内存插槽，系统便会自动识别进入三通道模式。三通道存储器总线最多可以同时并行传输：3x64b=192b

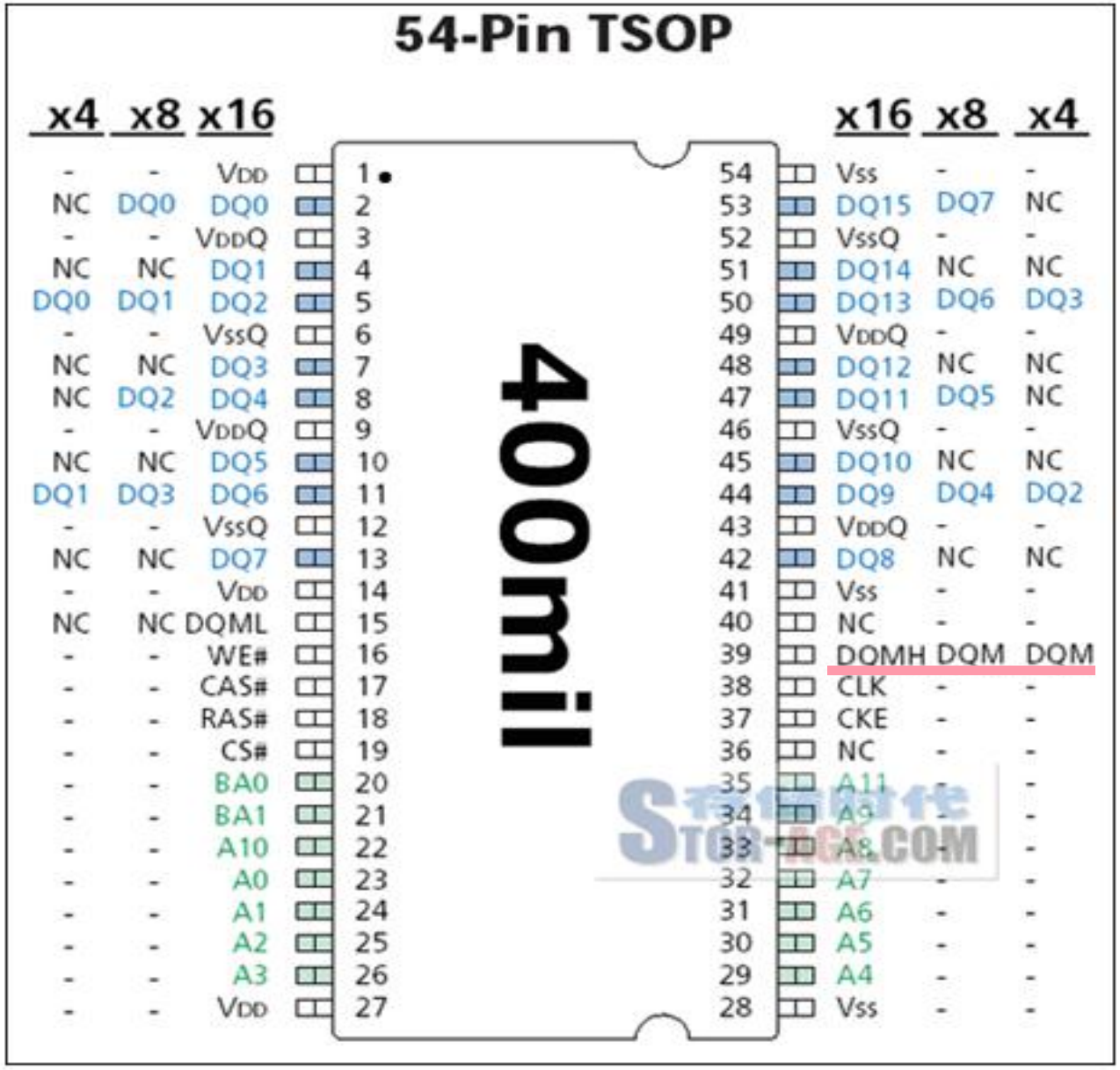


DRAM内存条结构

存控给出的地址包括：Channel、Rank、Bank、Row、Coloum



SDRAM芯片的引脚



DQM（数据掩码信号）：用于选择Burst传输中的哪个数据，比如，burst长度是4时，则表示需要传输4个64-bit，此时DQM选择需要传输哪几个64bit。

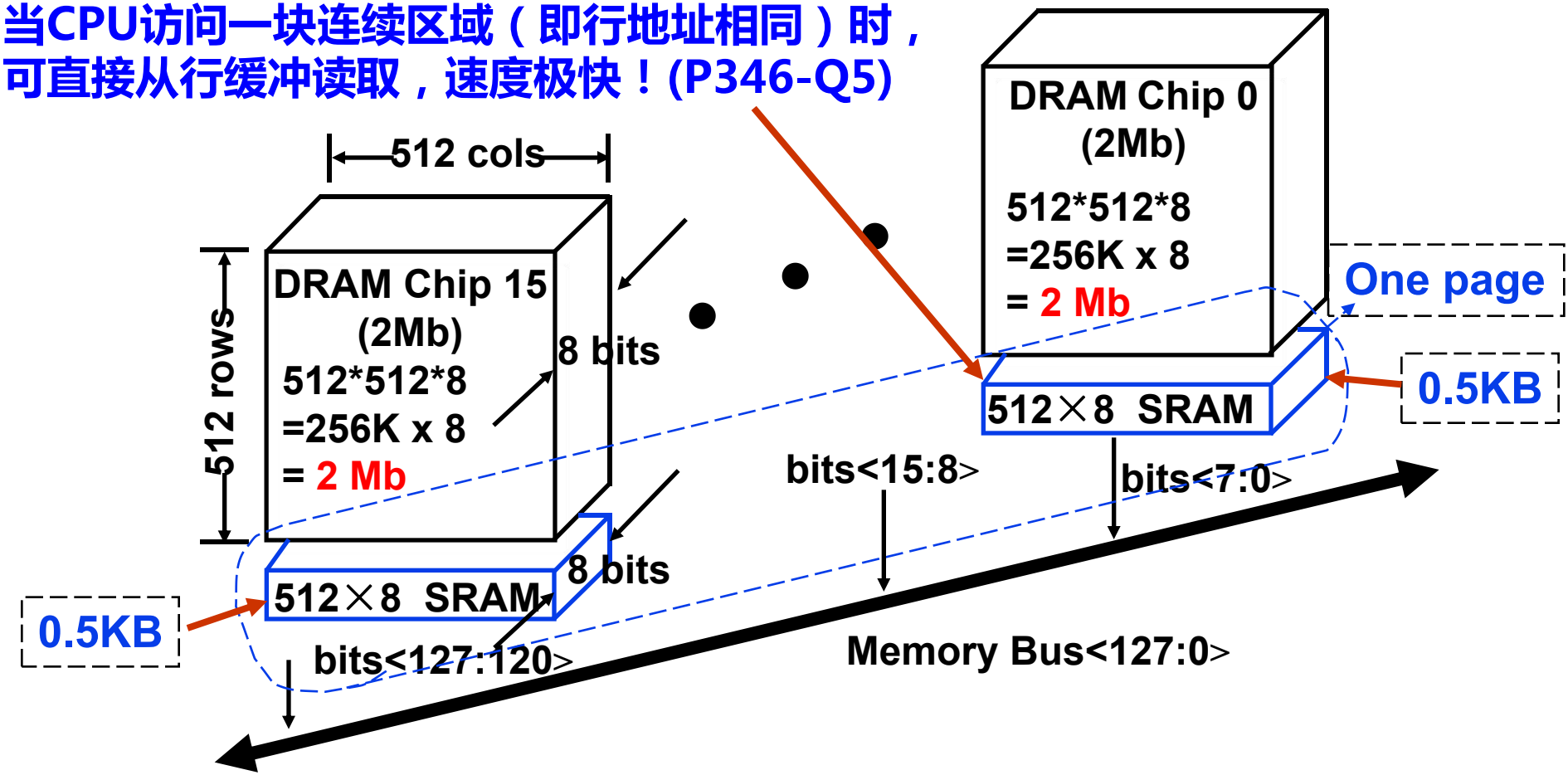
举例：SPARCstation 20’s内存条(模块)

某个4 MB的(DRAM)内存条含有： 16个内存芯片 x(2Mb/每个内存芯片）,每个芯片含有0.5KB的SRAM，一共有8 KB的 SRAM， 叫做 Page SRAM

每个内存芯片有512行x512列，并有8个位平面。每次读/写每个芯片的同行同列的8位，一共可以读出 (16个芯片）x（8位/每个芯片）=128位

问:行缓冲用SRAM实现,其数据的地址有何特点？在同一行中！

当CPU访问一块连续区域（即行地址相同）时，可直接从行缓冲读取，速度极快！(P346-Q5)

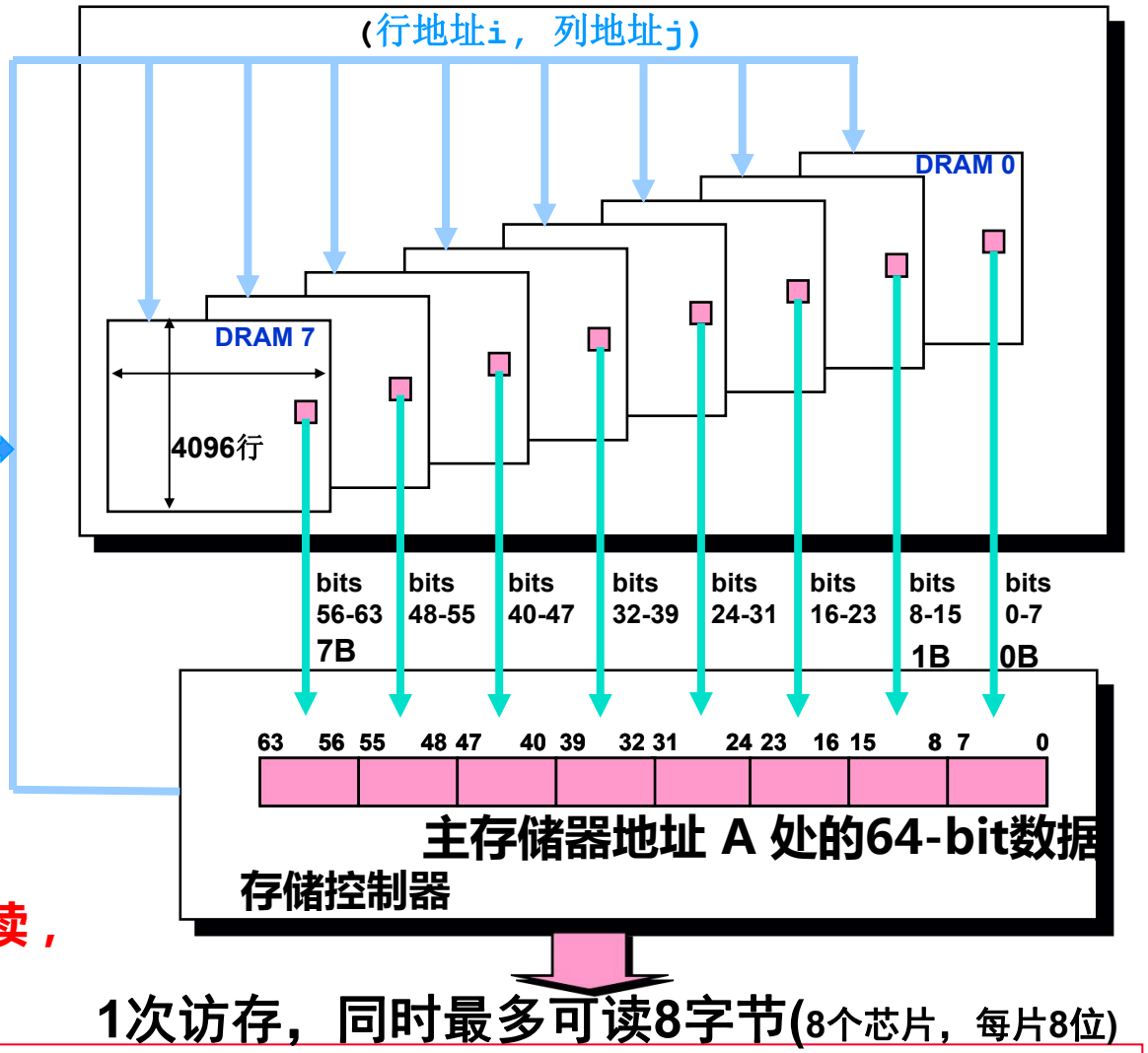


128MB的DRAM存储器

由8片DRAM芯片构成

- 每片 16Mx8 bits
- 行地址、列地址各12位
- 每行共4096列(8位/列)
- 选中某一行并读出之后再由列地址选择其中的一列(8个二进位) 送出，即: 行、列地址为(i,j)的8个单元
- 主存地址和片内地址有何关系？主存地址27位,其高24位为(每)片内地址，低3位为选片用(来确定哪一个芯片)

每个芯片内地址是否连续？不连续，交叉编址，可同时读写所有芯片。



1次访存，同时最多可读8字节(8个芯片，每片8位)

该存储器结构显示为何规定数据对齐存放，如一个int型数据若存放在第 8、9、10、11(字节序号) 这4个单元，需要几次访存？若存放在 6、7、8、9(字节序号) 这4个单元，则需要几次访内？分别访问1次和2次

128MB的DRAM存储器

128MB的存储器(其每个地址提供1个字节).**地址A有多少位？**

=4K*4K*8个字节

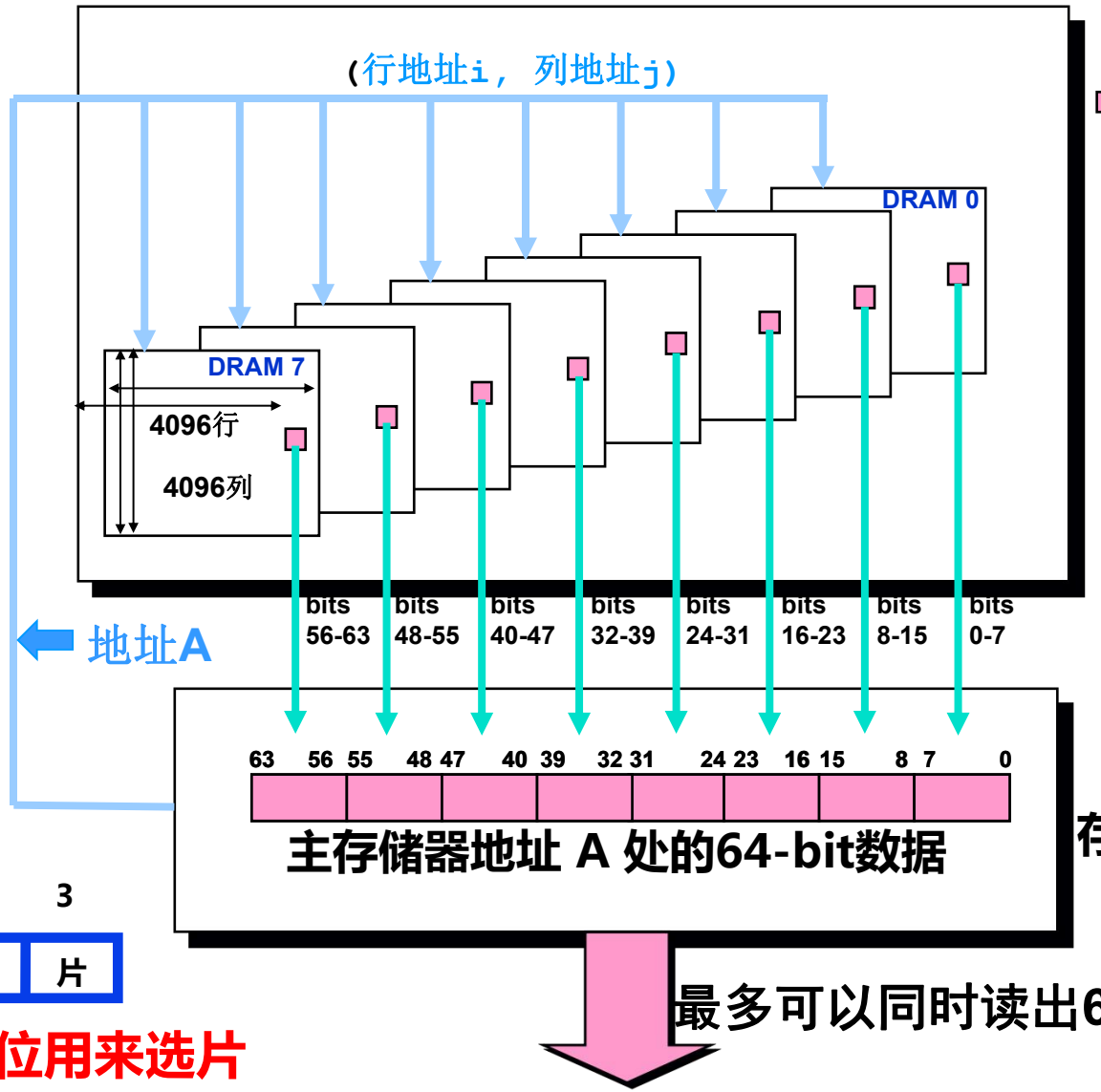
=4094*4096*8B

=2^{12} *2^{12}* 2^3个不同地址,每个地址提供一个字节(B)的数据.

地址位数=

$\log_2(2^{12+12+3})$

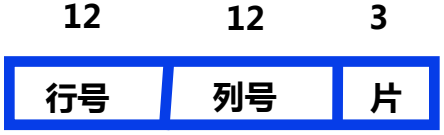
=27



: 不同芯片上相同行相同列上的不同个8位组成的行、列地址为(i,j)的8个单元

最多可以同时读出64位

如何划分地址？

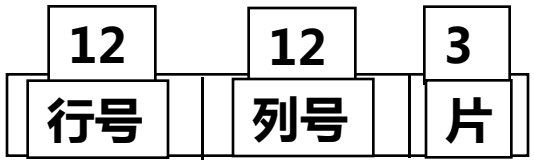


共27位！低3位用来选片

在DRAM行缓冲中数据的地址特点？不同芯片的相同行中的4096个列中的相同列里面的数据的地址连续，(每行共4096列)*(8个芯片）=2^{15}=32768个单元

128MB的DRAM存储器

地址A如何划分？　低3位用来选片



在DRAM行缓冲中数据的地址有何特点？

假定首地址为i，则地址分布如下：

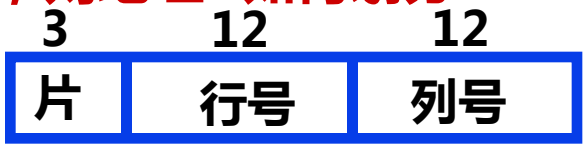
不同芯片的相同列的地址连续

	Chip0	Chip1	…	Chip7
第0列	i	i+1		i+7
第1列	i+8	i+9		i+15
…				
第4095列	i+8*4095	i+1+8*4095		i+7+8*4095

地址连续，共$8*4096=2^{15}=32768$个单元，每单元存8位数据

通常，一个主存块包含在行缓冲中可降低Cache缺失损失

如果片内地址连续，则地址A如何划分？



DRAM芯片的规格

- 若一个$2^n \times b$位(即（2^n)×(b位))DRAM芯片的存储阵列是r行×c列，则该芯片容量为$2^n \times b$位且$2^n = r \times c$。如：16K×8位DRAM，$16K = 2^{14} = r \times c$，则$r = c = 2^7 = 128$，$\log_2 r = \log_2 c = \log_2(2^7) = 7$。
- 芯片内的地址位数为n，其中行地址位数为$\log_2 r$，列地址位数为$\log_2 c$。如：16K×8位DRAM，$2^n = 16K$,则n=14，行、列地址各占7位。
- n位地址中高位部分为行地址，低位部分为列地址
- 为提高DRAM芯片的性价比，通常设置的r和c满足r≤c且|r-c|最小。
  - 例如，对于8K×8位DRAM芯片，其存储阵列设置为2^6行×2^7列，因此行地址和列地址的位数分别为6位和7位，13位芯片内地址$A_{12}\ A_{11}...A_1A_0$中，行地址为$A_{12}\ A_{11}...A_7$，列地址为$A_6...A_1A_0$。因按行刷新，为尽量减少刷新次数，故行数越少越好，但是，为了减少地址引脚，应尽量使行、列地址位数一致

层次结构存储系统

- 分以下四个部分介绍
  - 第一讲：存储器概述和RAM芯片
  - 第二讲：主存与CPU的连接及其读写操作
    - 主存模块的连接和读写操作
    - “装入”指令和“存储”指令操作过程
  - 第三讲：高速缓冲存储器(cache)
    - 程序访问的局部性、cache的基本工作原理
    - cache行和主存块之间的映射方式
    - cache和程序性能
  - 第四讲：虚拟存储器
    - 虚拟地址空间、虚拟存储器的实现

希望的理想存储器

到目前为止，已经了解到有以下几种存储器：

寄存器，SRAM，DRAM， 硬盘

	Capacity	Latency	Cost
Register	<1KB	1ns	$$$$
SRAM	1MB	2ns	$$$
DRAM	1GB	10ns	$
Hard disk*	1000GB	10ms	¢
Want	100GB	1ns	cheap

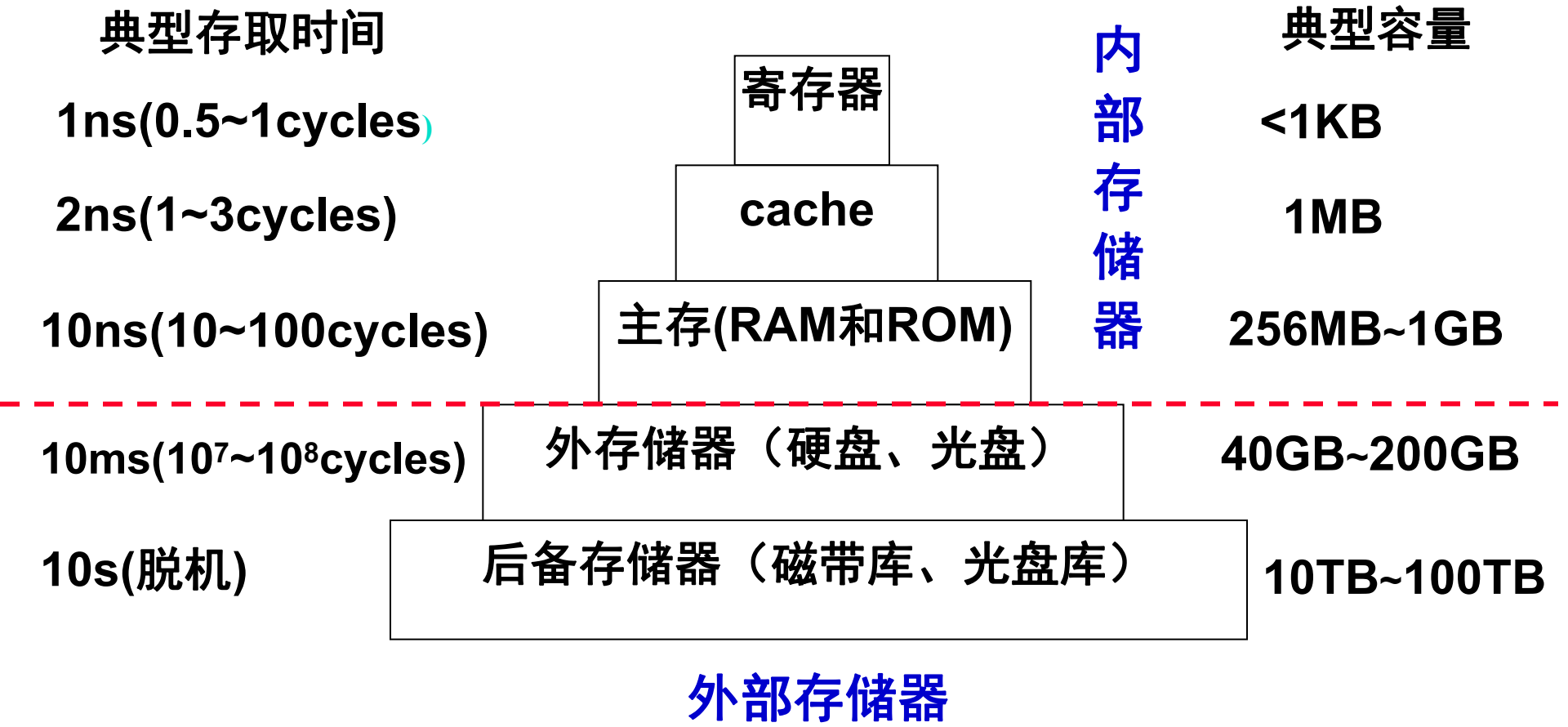
* non-volatile

问题：你认为哪一种最适合做计算机的存储器呢？

单独用某一种存储器，都不能满足我们的需要！

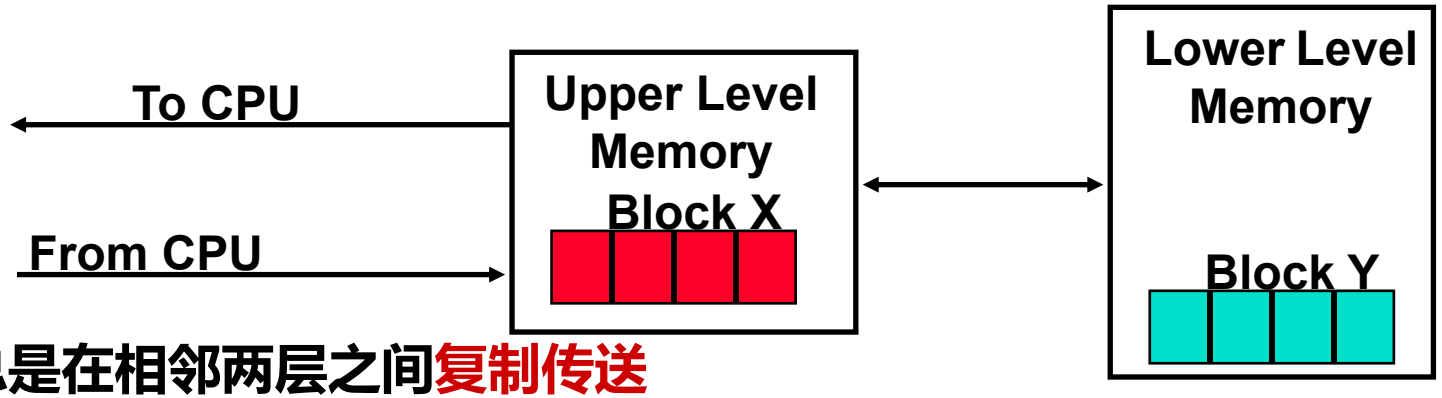
采用分层存储结构来构建计算机的存储体系！

存储器的层次结构



列出的时间和容量会随时间变化，但数量级相对关系不变。

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

相当于工厂中设置了多级仓库！

Block: 最小传送单位是定长块，互为副本

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。

这个高速缓存就是位于主存和CPU之间的Cache！

程序的局部性原理举例1

高级语言源程序

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

每个数组元素4字节

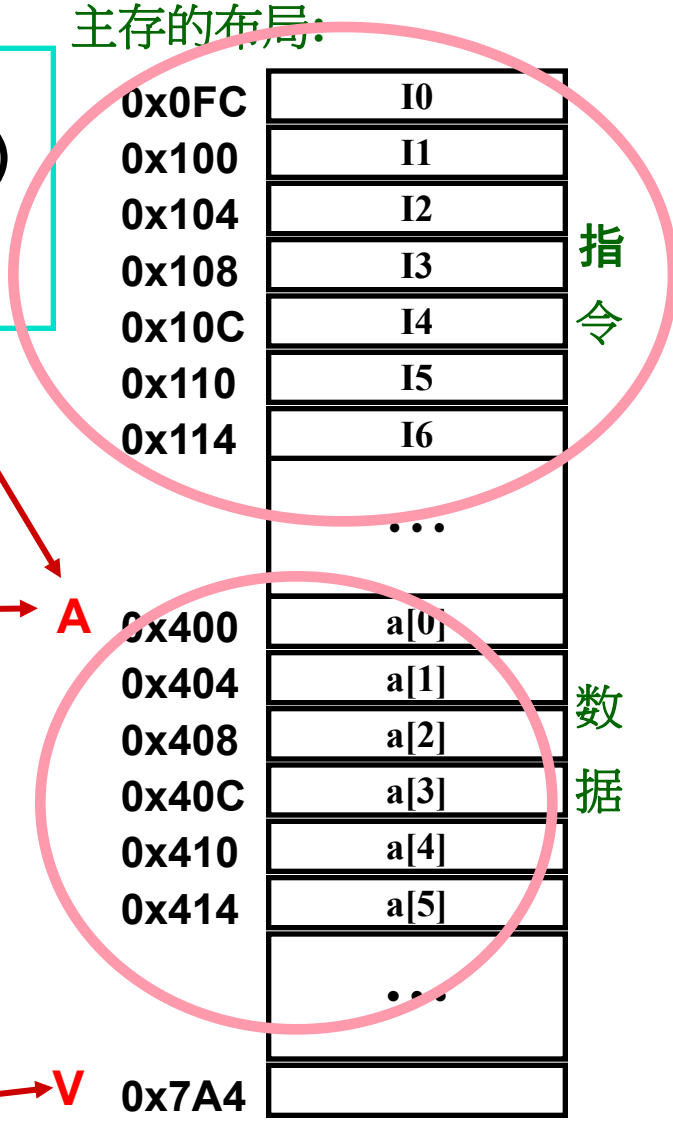
每条指令4个字节；

指令和数组元素在内存中分别连续存放

sum, ap ,i, t 均为通用寄存器；A，V为内存地址

对应的汇编语言程序

```
I0:              sum  <-- 0
I1:              ap  <-- A          //A是数组a的起始地址
I2:              i   <-- 0
I3:              if (i >= n) goto done
I4:    loop:     t   <-- (ap)       //数组元素a[i]的值
I5:              sum <-- sum + t    //累加结果在sum中
I6:              ap  <-- ap + 4     //计算下个数组元素地址
I7:              i   <-- i + 1
I8:              if (i < n) goto loop
I9:    done:     V  <-- sum         // 累加结果保存至地址v
```



程序的局部性原理举例1

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

问题：指令和数据的时间局部性和空间局部性各自体现在哪里？

指令：

0x0FC（I0）

…

→0x108（I3）

→0x10C（I4）

…

→0x11C（I8）

循环n次

→0x120（I9）

指令：**若n足够大，则在一段时间内一直在局部区域内执行指令，故循环内指令的时间局部性好；按顺序执行，故程序空间局部性好！**

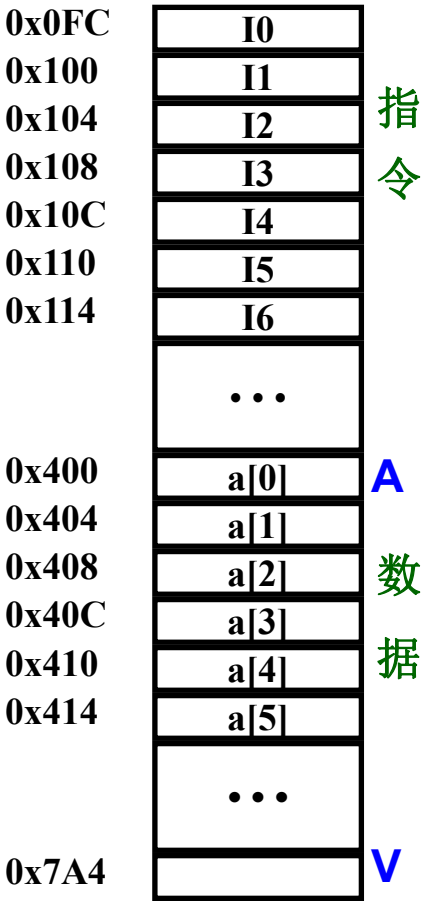
数据：只有数组在主存中：

0x400→0x404→0x408

→0x40C→……→0x7A4

数据:**数组元素按顺序存放，按顺序访问，故空间局部性好；每个数组元素都只被访问1次，故没有时间局部性。**

主存的布局：



程序的局部性原理举例2

以下哪个对数组a引用的空间局部性更好？时间局部性呢？变量sum的空间局部性和时间局部性如何？对于指令来说，for循环体的空间局部性和时间局部性如何？

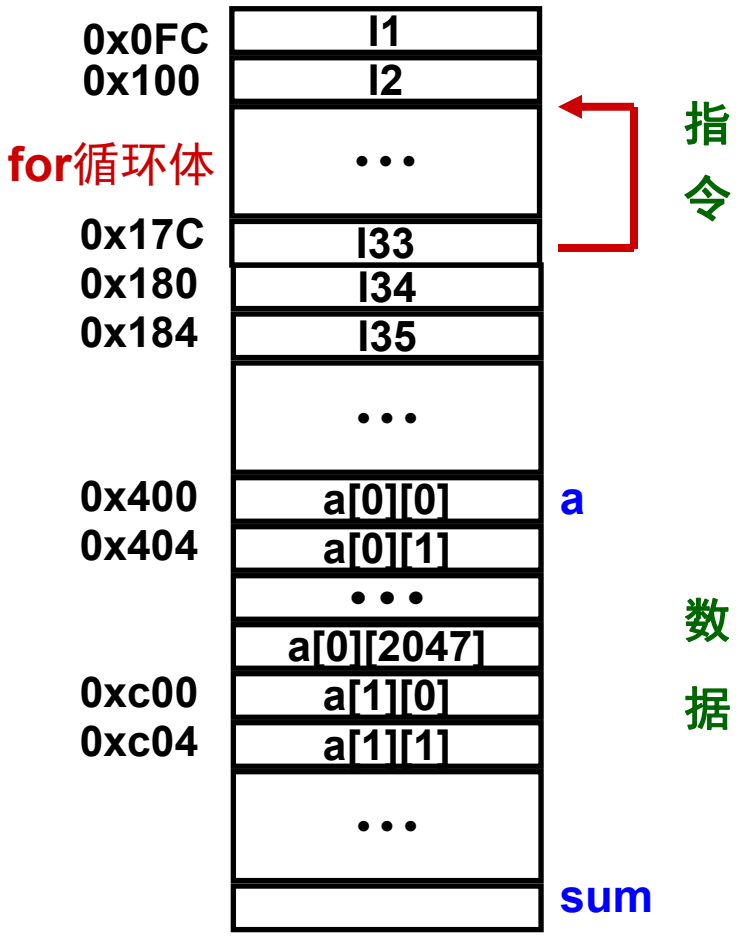
程序段A:

```
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;
}
```

程序段B:

```
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;
}
```

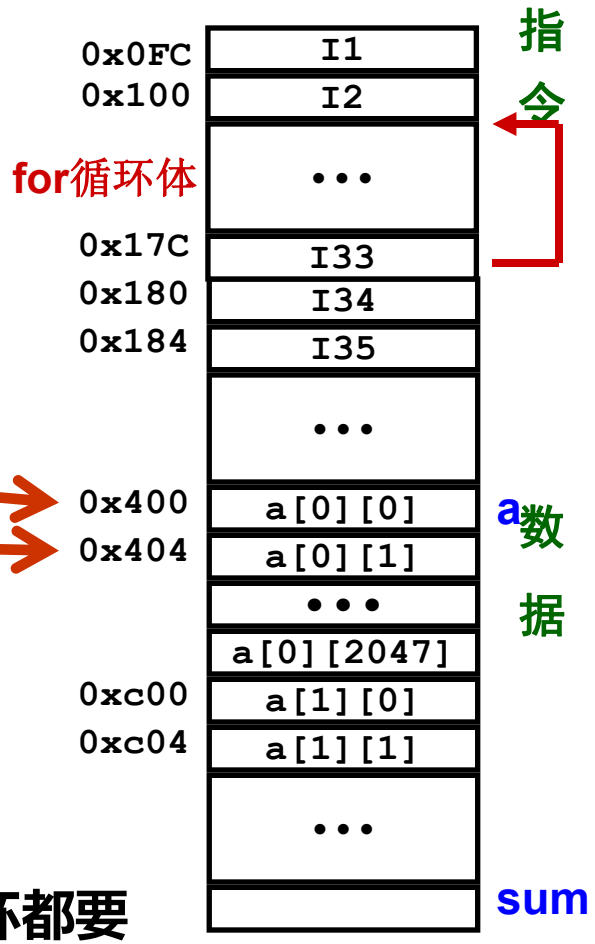
M=N=2048时主存的布局:



数组在存储器中按行优先顺序存放

程序的时间局部性和空间局部性分析-程序段A

```
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;
}
```



（1）**数组a**：访问顺序为a[0][0], a[0][1]

……, a[0][2047]; a[1][0], a[1][1],…… ,a[1][2047];

……，与存放顺序一致，故空间局部性好！

因为每个a[i][j]只被访问一次，故时间局部性差！

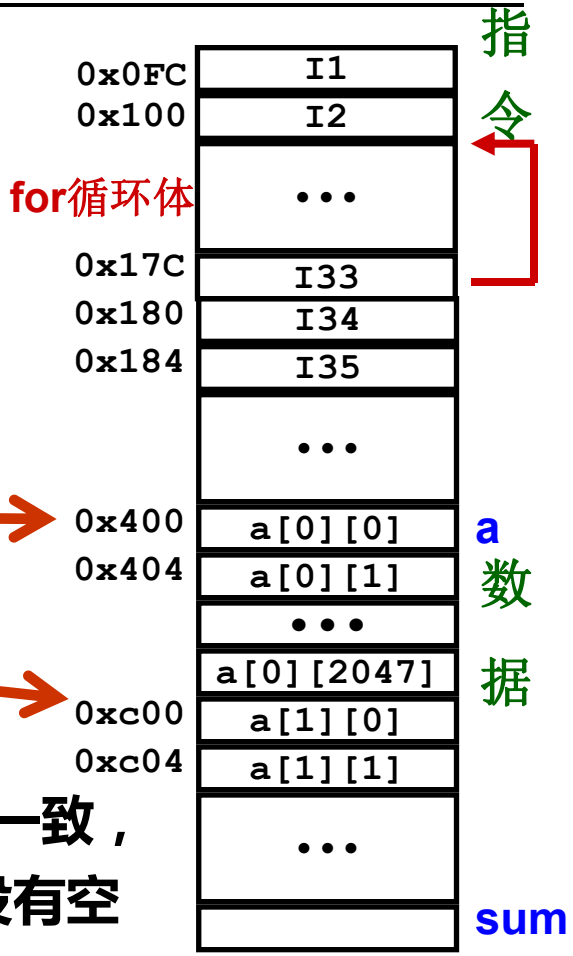
（2）**变量sum**：单个变量不考虑空间局部性；每次循环都要访问sum，所以其时间局部性较好！

（3）**for循环体指令**：循环体内指令按序连续存放，所以(既有)空间局部性好！循环体被连续重复执行2048x2048次，所以(又有)时间局部性好(的特点)！

实际上 优化的编译器使循环中的sum分配在寄存器中，最后写回存储器！

程序的时间局部性和空间局部性分析-程序段B

```
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;
}
```



（1）数组a：访问顺序为a[0][0],

a[1][0] ,

....., a[2047][0];

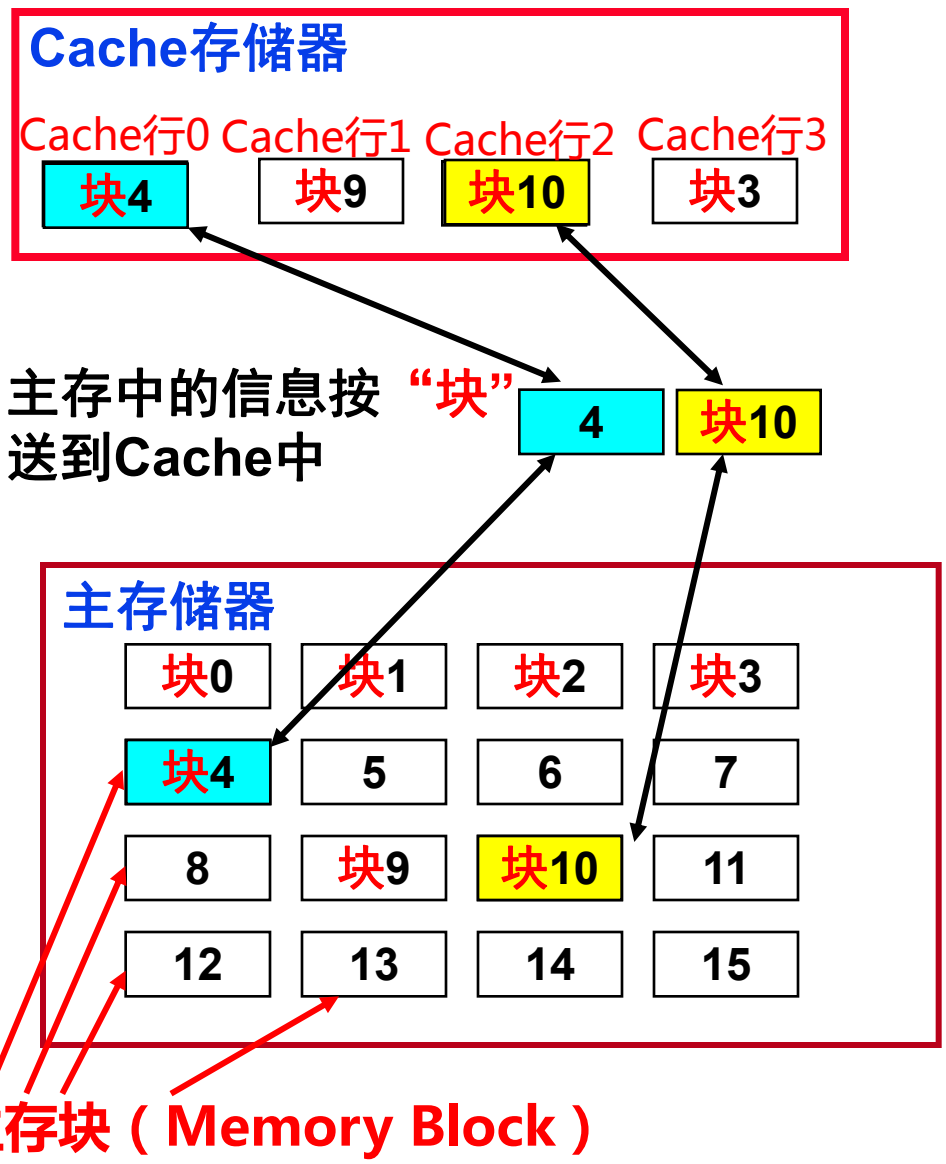
a[0][1], a[1][1],..... ,a[2047][1];.....，与存放顺序不一致，每次跳过2048(2K)个字节，若交换单位小于2KB，则没有空间局部性！时间局部性差(这点同程序A).

（2）变量sum，时间局部性较好！for循环体指令: 时间和空间局部性都好！

P4: Intel Pentium 4 2GHz，P4机器上运行结果：程序A：59,393,288 时钟周期, 程序B：1,277,877,876 时钟周期.程序A比程序B快21.5 倍!!

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，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;若没有，就访问主存储器。
- 主存被分成大小相同的单元，称为主存块(Block)，Cache也被分成大小相同的单元，称为Cache行（line）或槽（Slot）

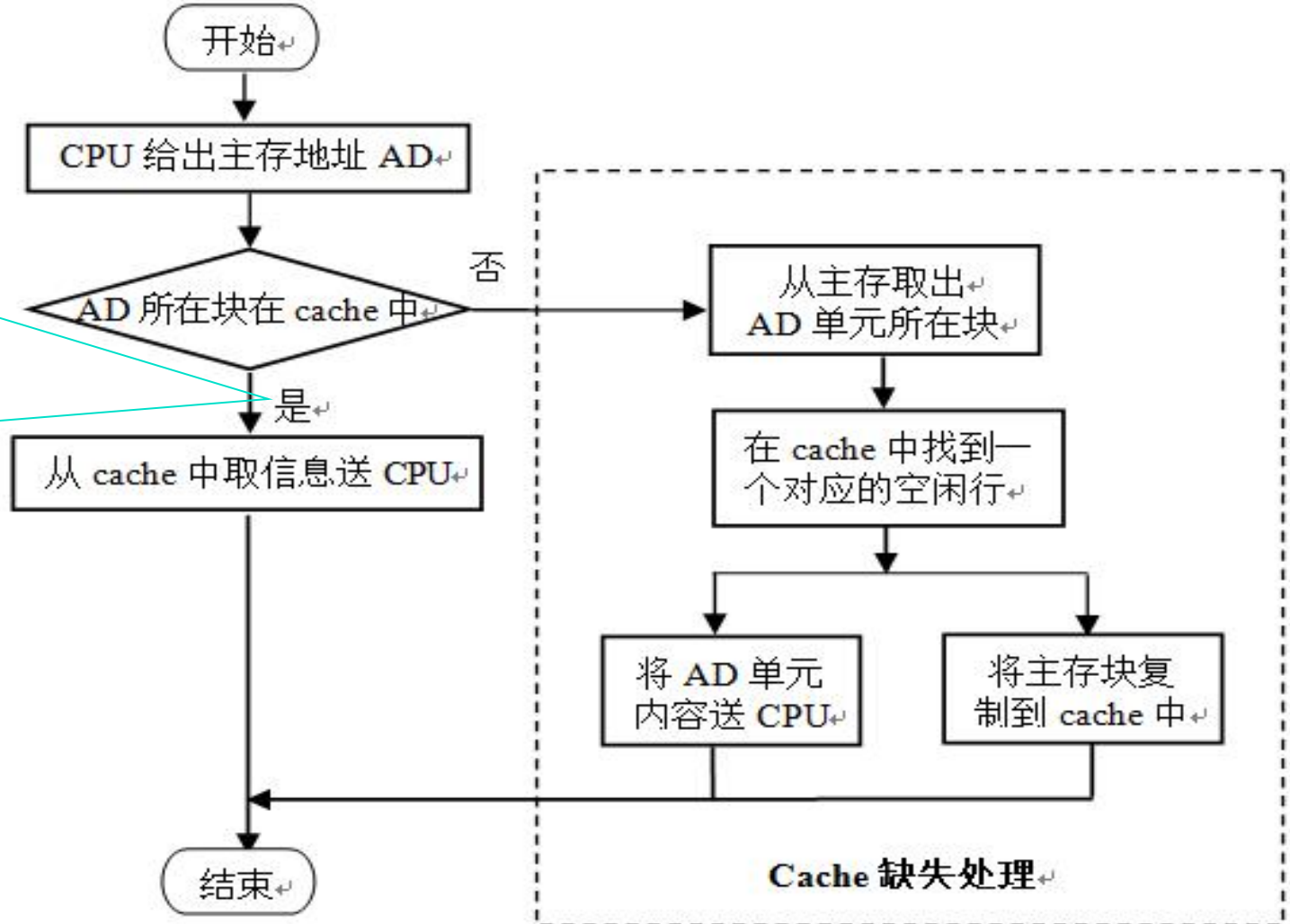


Cache 的操作过程

什么情况下，CPU产生访存要求？执行指令时！指令最初给出的是虚拟地址！

如何将虚拟地址转换为主存地址，后面介绍

若被访问信息在cache中，称为命中(hit)；如果不在,称为缺失或失靶(miss)



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？……

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？
  - 把将要访问的主存中的局部区域中的数据拷贝到Cache中时，放到Cache的何处？
  - Cache行数比主存块数目少，可能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(可以选择的多个行中的)任一行(e.g. 4-way, 就是4行中选1行）

直接(Direct)映射：每个主存块映射到Cache的固定行

- 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),块(行)都从0开始编号
  - 也称模映射(Module Mapping)
  - 映射关系为：

    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

SKIP

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。

直接映射Cache组织示意图

设数据在主存和Cache间的传送单位为512B,即512B /块。

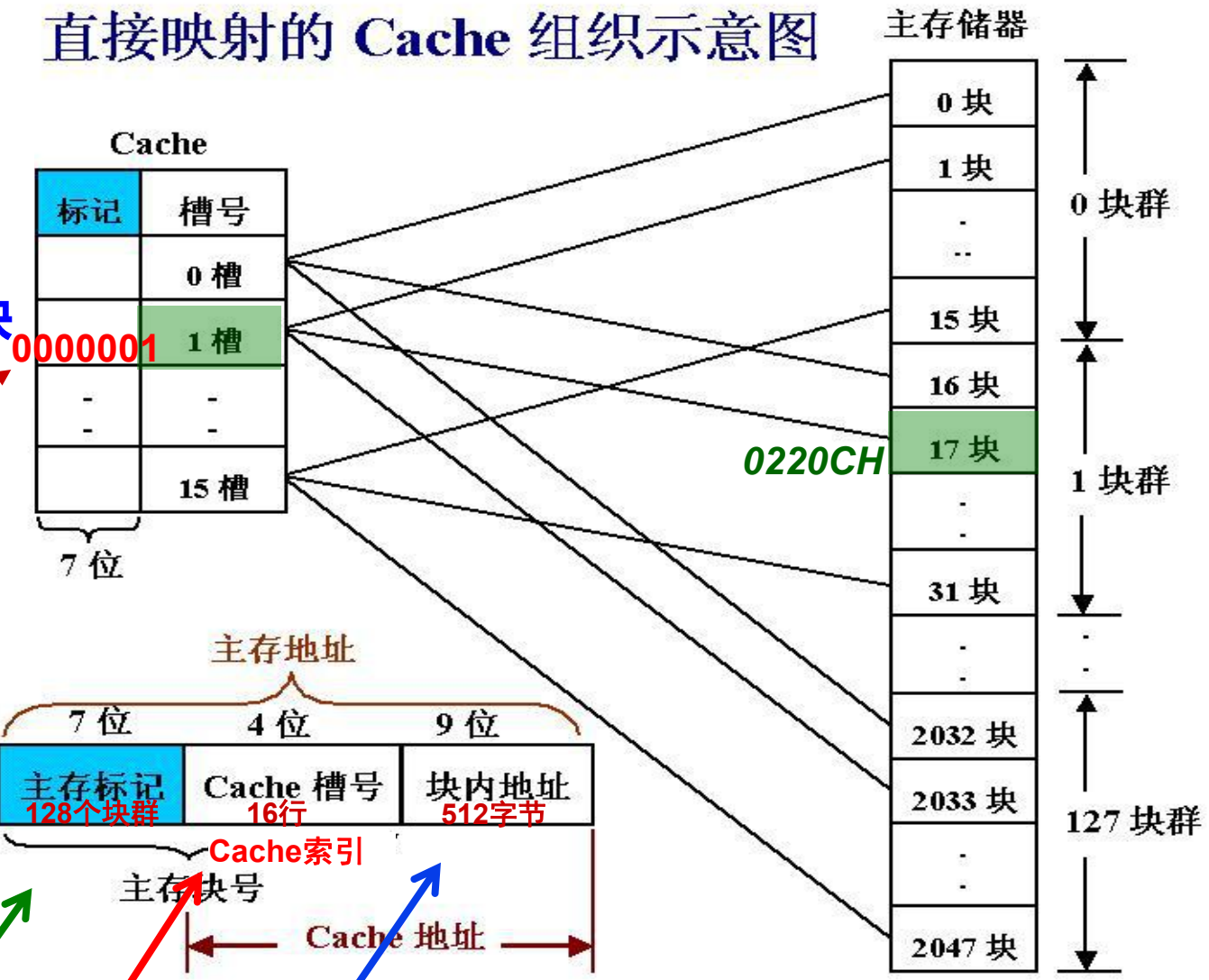
Cache大小：
2^{13}B=8KB=16行 x (512B/ 行)

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

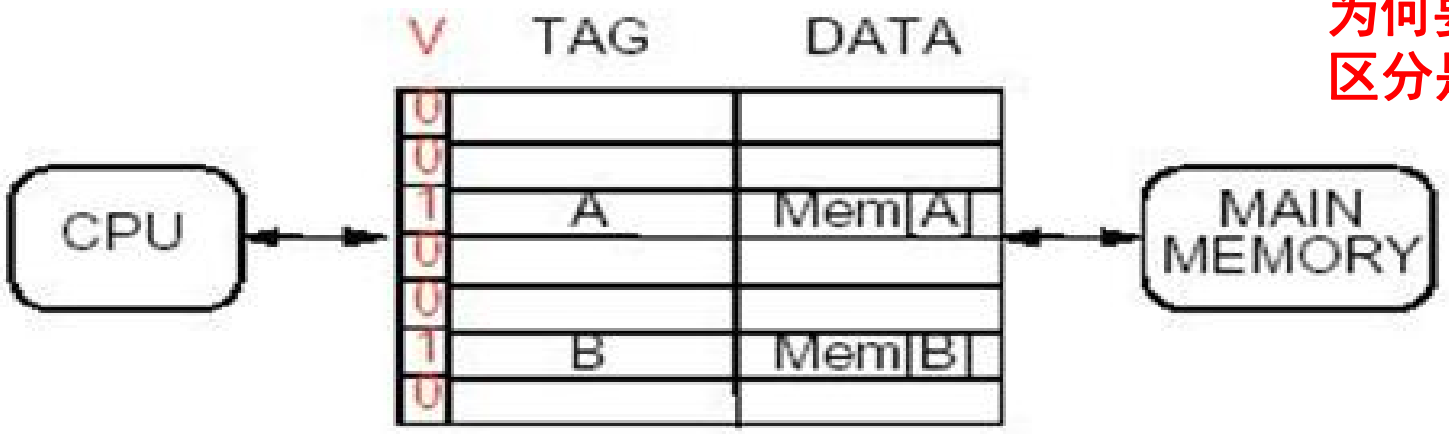
例：如何对0220CH单元进行访问？



0000-001 0-001 0-0000 1100B 是第1块群中的0001块（即主存中的第17块）中第12个字节单元！

有效位（Valid Bit）

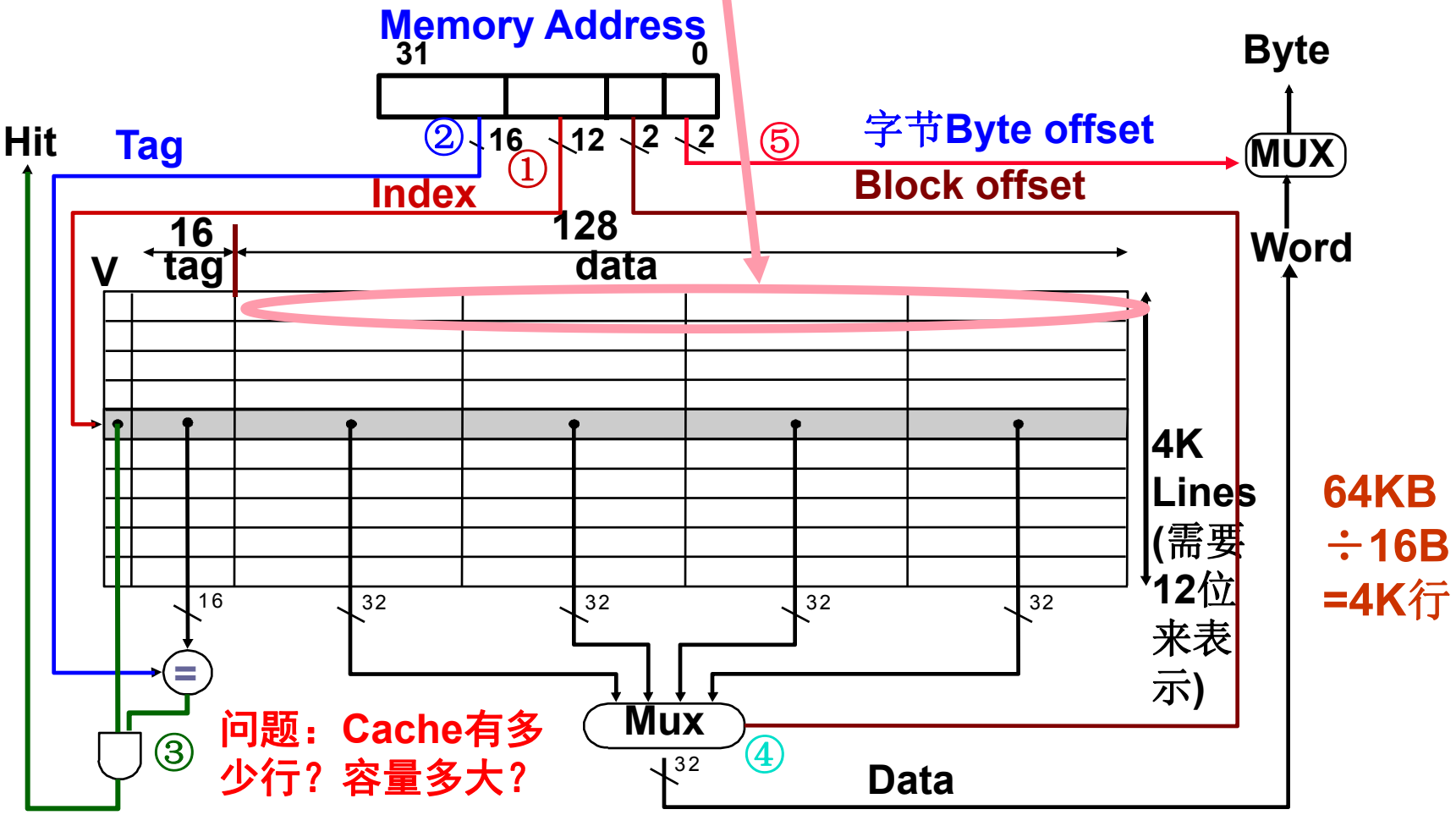
为何要用有效位来区分是否有效？



- 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！

64 KB Direct Mapped Cache with 16B Blocks

主存和Cache之间直接映射，块大小为16B。Cache的数据区容量为64KB，主存地址为32位，按字节编址。 要求：说明主存地址如何划分和访存过程。



容量 4Kx(1+16) （位） +4Kx16（字节）x8（位）=580Kbits=72.5KB,
数据占64KB / 72.5KB = 88.3%

如何计算Cache的容量？

Consider a cache with 64 Lines and a block size of 16 bytes.

What line number does byte address 1200 map to?

地址1200对应存放在第11行。因为： [1200/16=75] module 64 = 11

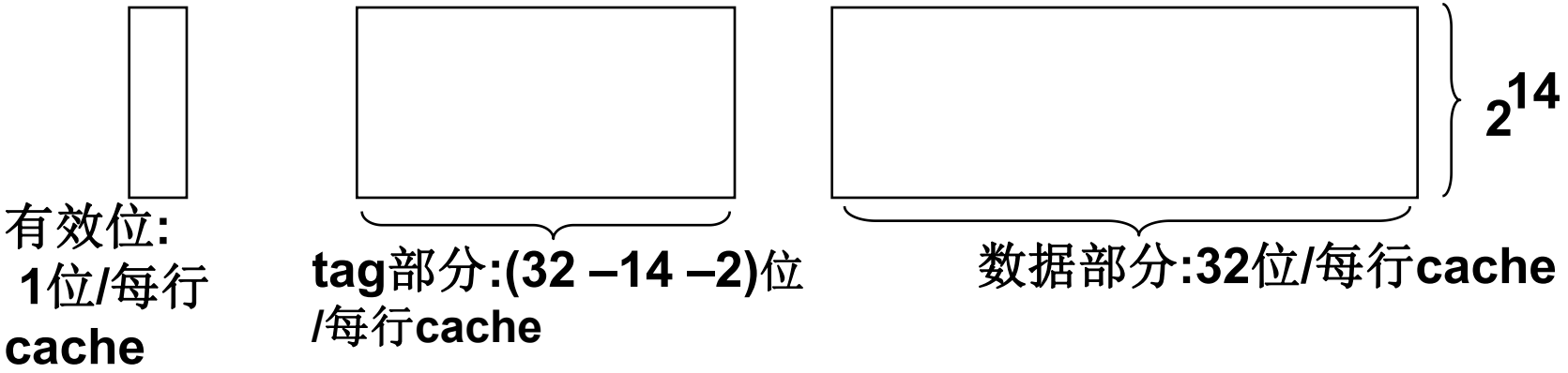
1200 = 1024+128+32+16 = 0…01 001011 0000 B

实现以下cache需要多少位容量？

Cache：直接映射 、16K行数据、块大小为1个字(4B)、32位主存地址

答：Cache的存储布局如下：

Cache共有16K x 4B= 64KB数据



所以，Cache的大小为：2^{14} ×(32 + (32-14-2)+1) = 2^{14}×49 = 784 Kbits

若块大小为4个字呢？ 2^{14} ×(4×32 + (32-14-2-2)+1) = 2^{14}×143 = 2288 Kbits

若块大小为2^m个字呢？ 2^{14} ×(2^m×32 + (32-14-2- m)+1)

BACK

全相联映射Cache组织示意

假定数据在主存和Cache间的传送单位为512字(2^9个字)[不是字节，不必精确到字节]。

Cache(数据区)大小：8K字=2^{13}字= 2^4*2^9 =16行 x 512字/ 行

主存大小：1M(个)字= 2^{20}字=(2^{11} (块)) *(2^9字/ 块) =2048块 x 512字/ 块

Cache标记（tag）指出对应行取自哪个主存块

主存地址中的tag部分指出对应地址位于哪个主存块

如何对01E0CH单元进行访问？

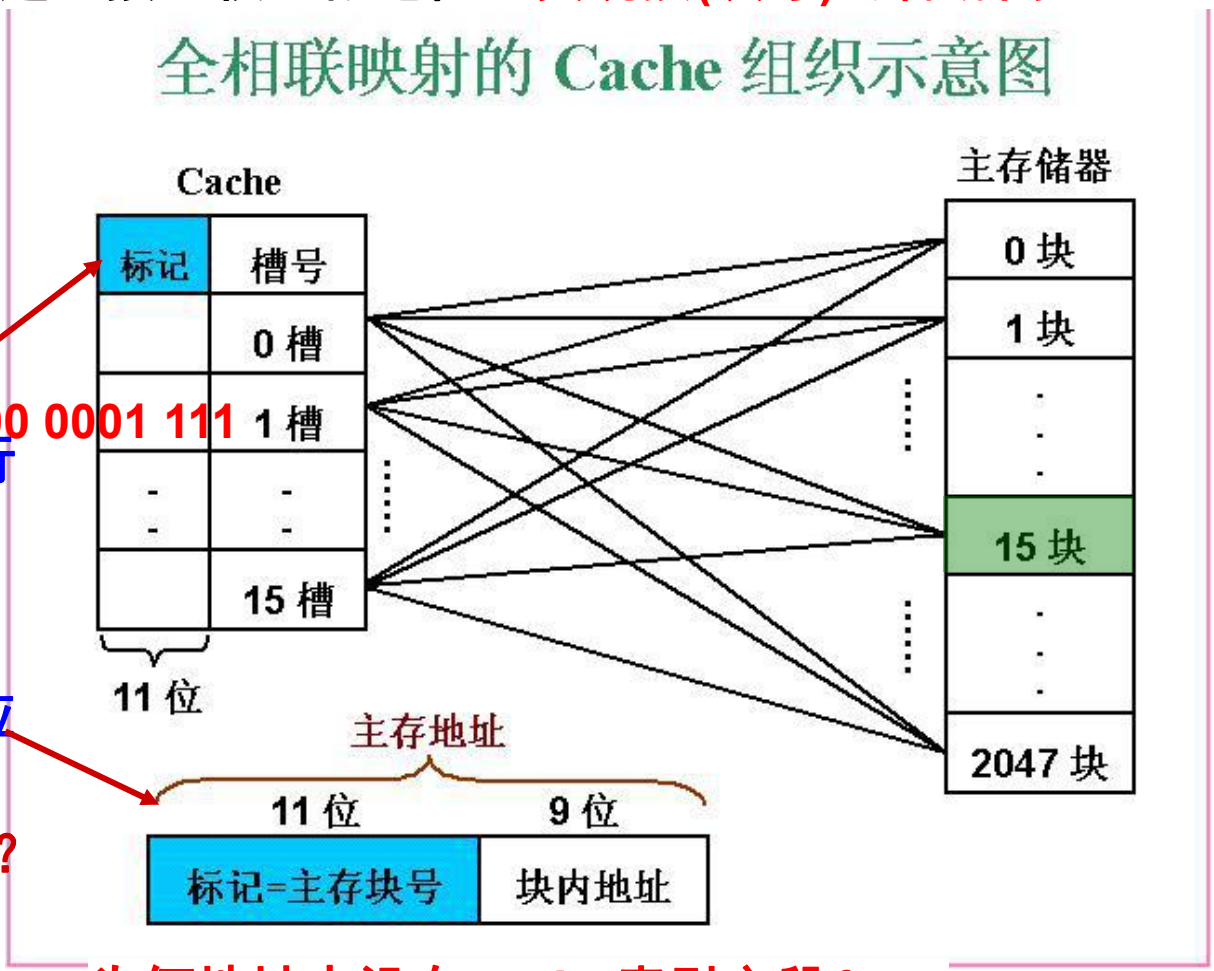
0000 0001 1110 0000 1100b

是第(0000 0001 111)15块中的第12个单元！

全相联映射:每个主存块可装到Cache任一行中。

按内容访问，是相联存取方式！

通过直接比较（标记位）实现按(块号)内容访问

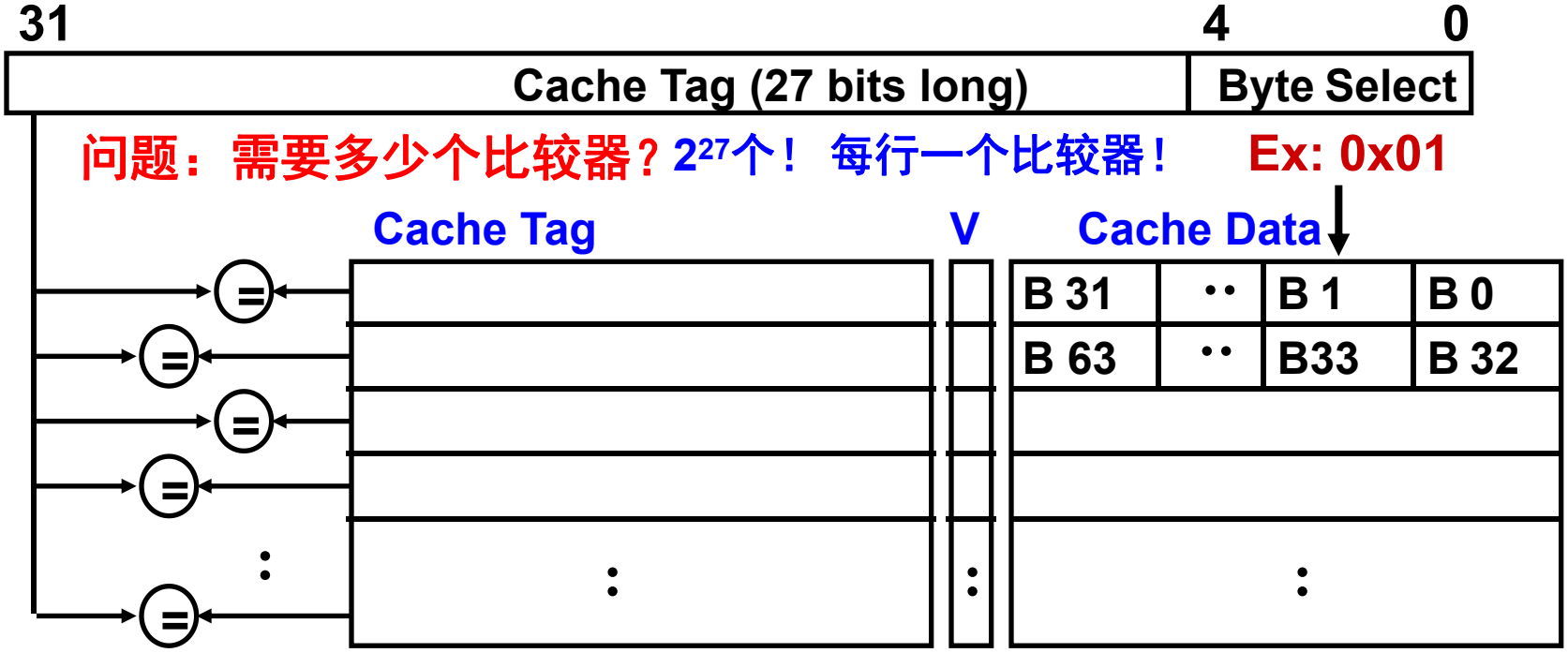


为何地址中没有cache索引字段？

因为可映射到任意一个cache行中！

举例：全相联映射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, 32B(Bytes) per block(i.e. 32 B(Bytes) per cache line). 比较器(比较Tag部分是否相等)位数多长？
  - we need N 27-bit comparators



组相联映射（Set Associative）

- 组相联映射结合直接映射和全相联映射的特点
- 将Cache所有行分组，把主存块映射到Cache固定组的任一行中。也即：组间模映射、组内全映射。映射关系为：

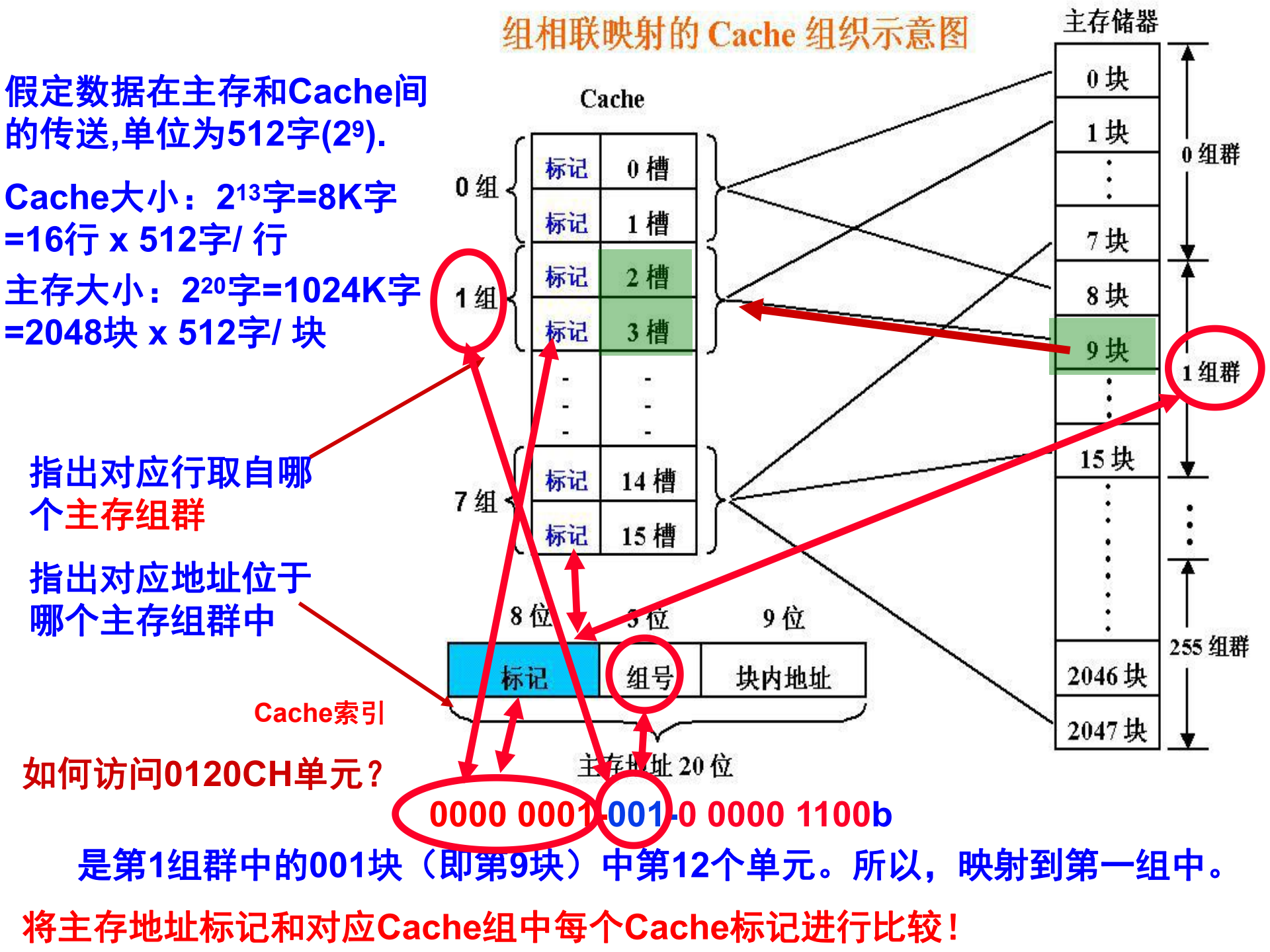
Cache组号=主存块号 mod Cache组数

举例：假定Cache划分为：8K字=(8组)x(2行/组)x(512字/行)

主存第100块应映射到Cache的第几组的哪一行？

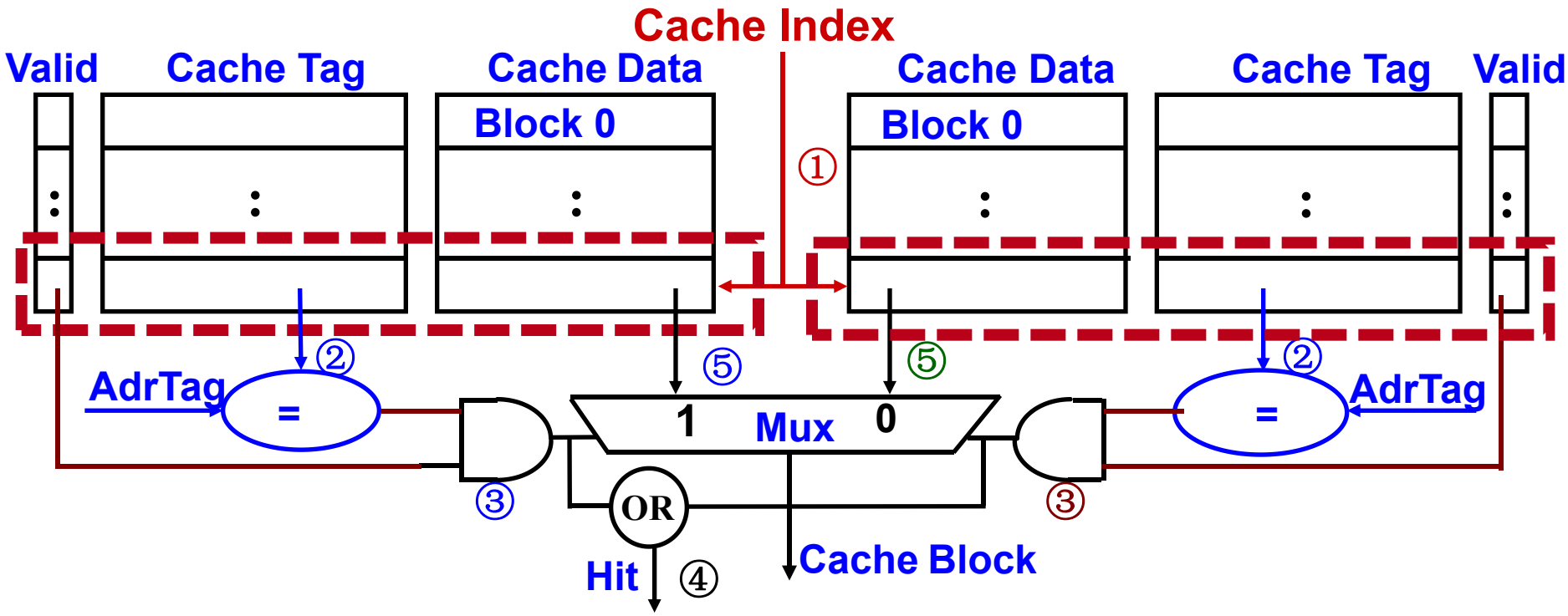
100 mod 8 = 4　答：第4组的任意行中。

- 特点：
  - 结合直接映射和全相联映射的优点。当Cache组数为1时，变为全相联映射；当每组只有一个槽时，变为直接映射。
  - 每组2行或4行（称为2-路或4-路组相联）。在较大容量的（ L1, L2和L3) Cahce中使用更多路。


组相联映射的 Cache 组织示意图
假定数据在主存和Cache间的传送,单位为512字(2⁹).
Cache大小：2¹³字=8K字
=16行 x 512字/ 行
主存大小：2²⁰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
Cache索引
如何访问0120CH单元？
Cache
0 组
1 组
7 组
标记
0 槽
1 槽
2 槽
3 槽
14 槽
15 槽
主存储器
0 块
1 块
7 块
8 块
9 块
15 块
2046 块
2047 块
0 组群
1 组群
255 组群
8 位
3 位
9 位
标记
组号
块内地址
主存地址 20 位
0000 0001 001 0 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！

2路组相联(A Two-way Set Associative) Cache

- N-way set associative
  - N 个直接映射的行并行操作
- Example: Two-way set associative cache
  - Cache Index 选择其中的一个Cache行集合（共2行）
  - 对这个集合中的两个Cache行的Tag并行进行比较
  - 根据比较结果确定信息在哪个行，或不在Cache中



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?）

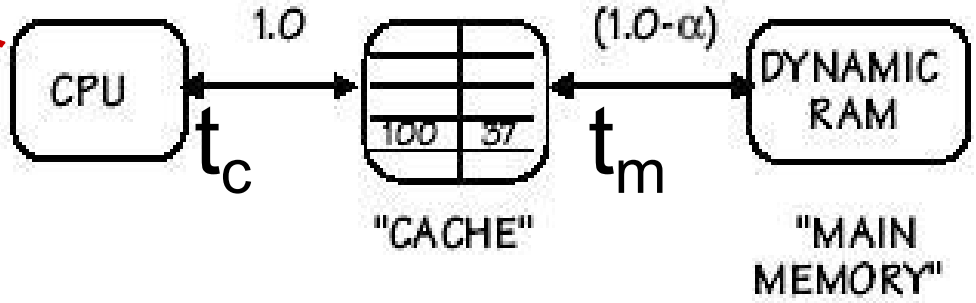
Average access time(平均访问时间)

CPU都必须先访问cache,无论cache命中或不命中！

t_c访问cache的时间

t_m访问主存的时间

Program-Transparent Memory Hierarchy



Cache contains TEMPORARY COPIES of selected main memory locations... eg. Mem[100] = 37

GOALS:

1) Improve the average access time

要提高平均访问速度，必须提高命中率！

α HIT RATIO: Fraction of refs found in CACHE.

(1-α) MISS RATIO: Remaining references.

$$t_{ave} = \alpha t_c + (1-\alpha)(t_c + t_m) = t_c + (1-\alpha)t_m$$

2) Transparency (compatibility, programming ease)

Challenge:
To make the hit ratio as high as possible.

Cache对程序员来说是透明的，以方便编程！

cache命中率到底应该有多大?

How high of a hit ratio?

Suppose we can easily build an on -chip static memory with a 4 nS access time, but the fastest dynamic memories that we can buy for main memory have an average access time of 40 nS. How high of a hit rate do we need to sustain an average access time of 5 nS?

$$\alpha = 1 - \frac{t_{ave} - t_c}{t_m} = 1 - \frac{5-4}{40} = 97.5\%$$

WOW, a cache really needs to be good?

看看cache命中率对平均访问时间的影响

设H是cache命中率，即（1-H）是不命中率，则平均访问时间

T = (cache命中情况下的时间) + (cache不命中情况下的时间)

= (cache命中情况下访问cache的时间) + (cache不命中情况下访问cache及主存的时间)

$= HT_C + (1 - H)(T_C + T_M)$

$= T_C + (1 - H)T_M$

例1. 若H=0.85, Tc=1ns, T_M=20ns，则T为多少？ 答：T = 4ns

例2. 若命中率H提高到0.95，则结果又如何？ 答： T = 2ns

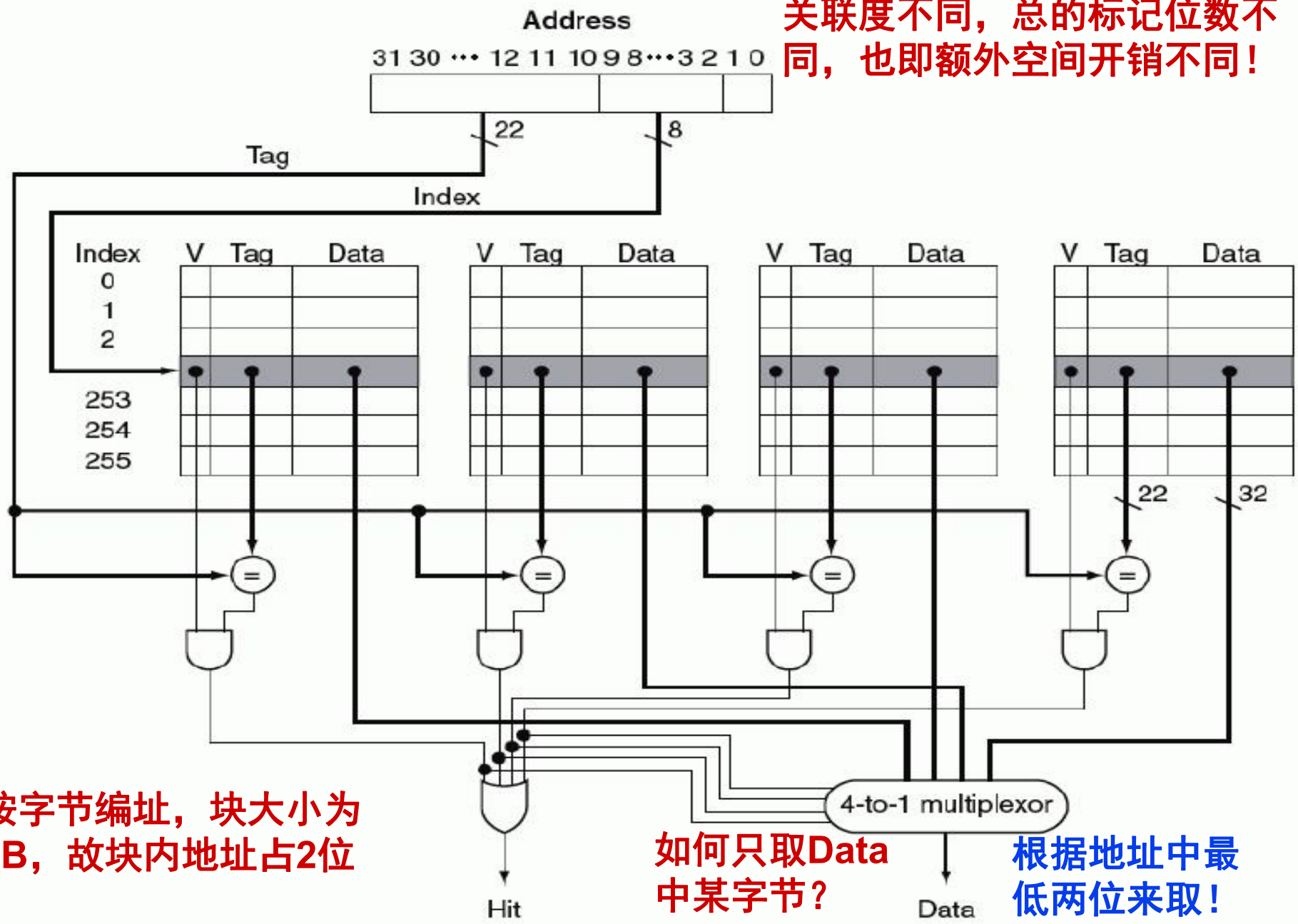
例3. 若命中率为0.99呢？ 答: T = 1.2ns

访存速度与命中率的关系非常大！

高速缓存的缺失率和关联度

- 三种映射方式
  - 直接映射：唯一映射（只有一个可能的位置）
  - 全相联映射：任意映射（每个位置都可能）
  - N-路组相联映射：N-路映射（有N个可能的位置）
- 什么叫关联度？一个主存块映射到Cache中时，(最多)可能(选择的能够)存放的cache行的位置的个数
  - 直接映射关联度？一个主存块只能存放到1个cache行,关联度最低，为1
  - 全相联映射关联度？一个主存块(最多)可存放到任意个cache行,关联度最高，为Cache行数
  - N-路组相联映射关联度？一个主存块(最多)可存放到N个cache行中任意一行,关联度居中，为N
- 关联度和miss rate有什么关系呢？和命中时间的关系呢？
  - 直观上，你的结论是什么？（ Cache大小和块大小一定时 ）
    - 缺失率：直接映射最高，全相联映射最低
    - 命中时间：直接映射最小，全相联映射最大
  - 用例子来说明

标记位大小与关联度



关联度不同，总的标记位数不同，也即额外空间开销不同！

按字节编址，块大小为4B，故块内地址占2位

如何只取Data中某字节？

根据地址中最低两位来取！

标记位大小与关联度

问题：若主存地址32位，块大小为16字节(2^4)，Cache总大小为4K行，问：需要多少位来表示这4K行? 标记位的总位数是多少？

4K行= 2^{12}行， $\log_2{}^{4k}$= 12, 需要12位来表示这4K行

需要多少位来表示行内或块内偏移?$\log_2{}^{16}$= 4

直接映射方式下：相当于每组1行，共4K组，标志占32-12-4=16位
标志位的总位数占 4Kx16=64K位

关联度增加到2倍(2-way)：每组2行，共2K组(2^{11})(为表示这2K组需要$\log_2{}^{2k}$=11位), 标志占32-4(块大小为16字节(2^4))-11(共2K组(2^{11}))=17位, 标志位的总位数占4Kx17=68K位

关联度增加到4倍(4-way)：每组4行，共1K组(2^{10})(为表示这1K组需要$\log_2{}^{1k}$=10位)，，标志占32-4-10=18位, 标志位的总位数占4Kx18=72K位

全相联时：整个为1组，每组4K行，标志占32-4=28位
标志位的总位数占4Kx28=112K位

关联度(2->4->全相联)越高，总的标记位数越多，(除了存储数据外的)额外（的存储非数据信息的）空间开销越大！

替换(Replacement)算法

- 问题举例：

  组相联映射时，假定第0组的两行分别被主存第0和8块占满，此时若需调入主存第16块，根据映射关系，它只能放到Cache第0组，因此，第0组中必须调出一块，那么调出哪一块呢？

  这就是淘汰策略问题，也称替换算法。

- 常用替换算法有：
  - 先进先出FIFO（first-in-first-out）
  - 最近最少用LRU（ least-recently used）
  - 最不经常用LFU（ least-frequently used）
  - 随机替换算法（Random）

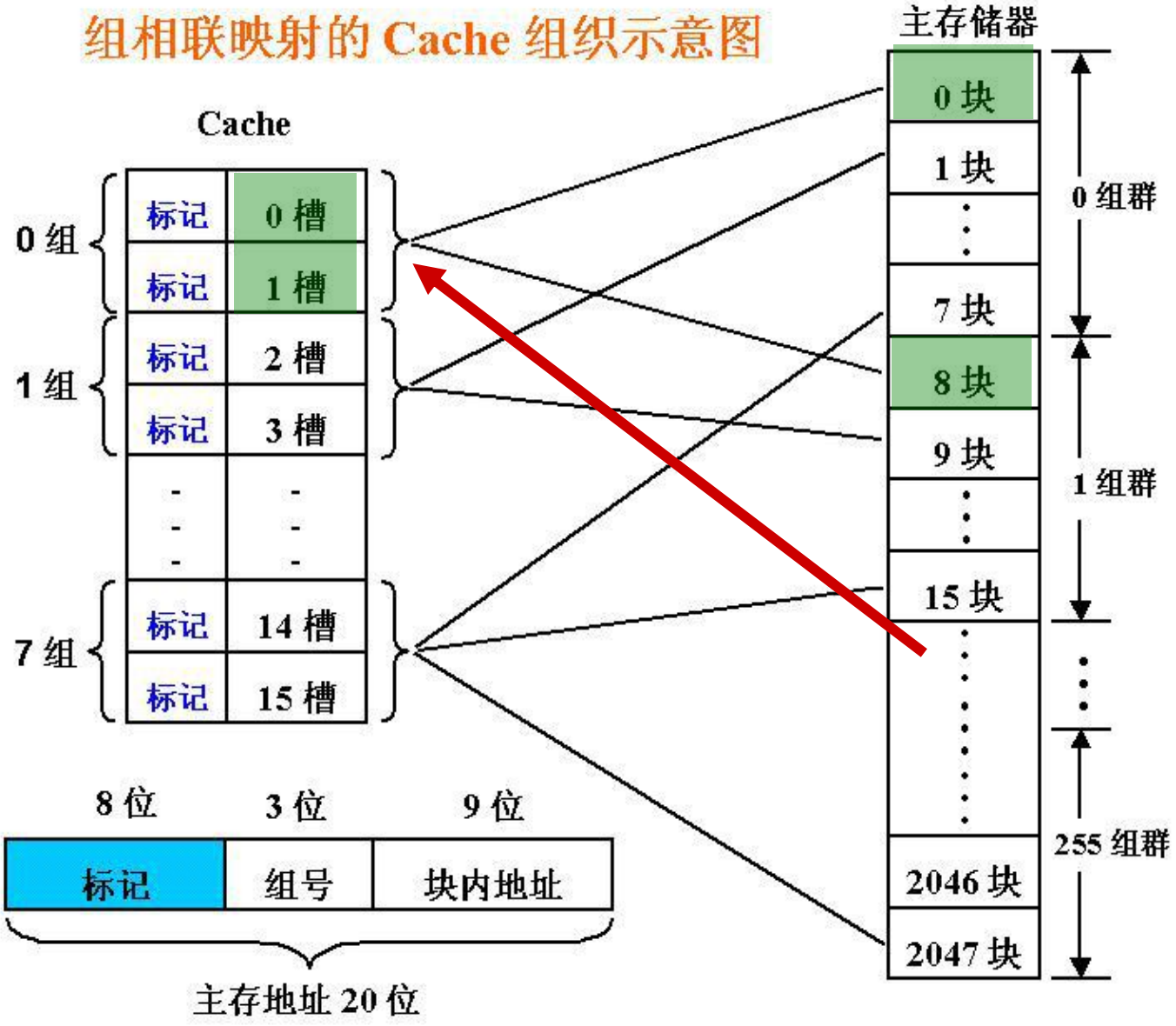
  等等

SKIP

这里的替换策略和后面的虚拟存储器所用的替换策略类似，将是以后操作系统课程的重要内容，本课程只做简单介绍。有兴趣的同学可以自学。

假定第0组的两行分别被主存第0和8块占满，此时若需调入主存第16块
该怎么办？

第0组中必须调出一块，那么，调出哪一块呢？



BACK

cache行替换算法-先进先出(FIFO)

- 总是把最先进入的那一块淘汰掉。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，考察下面cache中3行/组、 4行/组的情况。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；

带*的表示最先进入的那一内存块（相对来说），

CPU访问的块:1-> 2 -> 3 ->4 ->1 ->2 ->5->1-> 2-> 3-> 4-> 5

3行/组的cache

1*	1*	1*	4	4	4*	5	5	5	5	5*	5*
	2	2	2*	1	1	1*	1*	1*	3	3	3
		3	3	3*	2	2	2	2	2*	4	4

√表示命中　　命中3次　　√　√　√

CPU访问的块:1 ->2-> 3 -> 4 -> 1->2-> 5-> 1 -> 2-> 3-> 4-> 5

4行/组的cache

1*	1*	1*	1*	1*	1*	5	5	5	5*	4	4
	2	2	2	2	2	2*	1	1	1	1*	5
		3	3	3	3	3	3*	2	2	2	2*
			4	4	4	4	4	4*	3	3	3

命中2次　　√　√

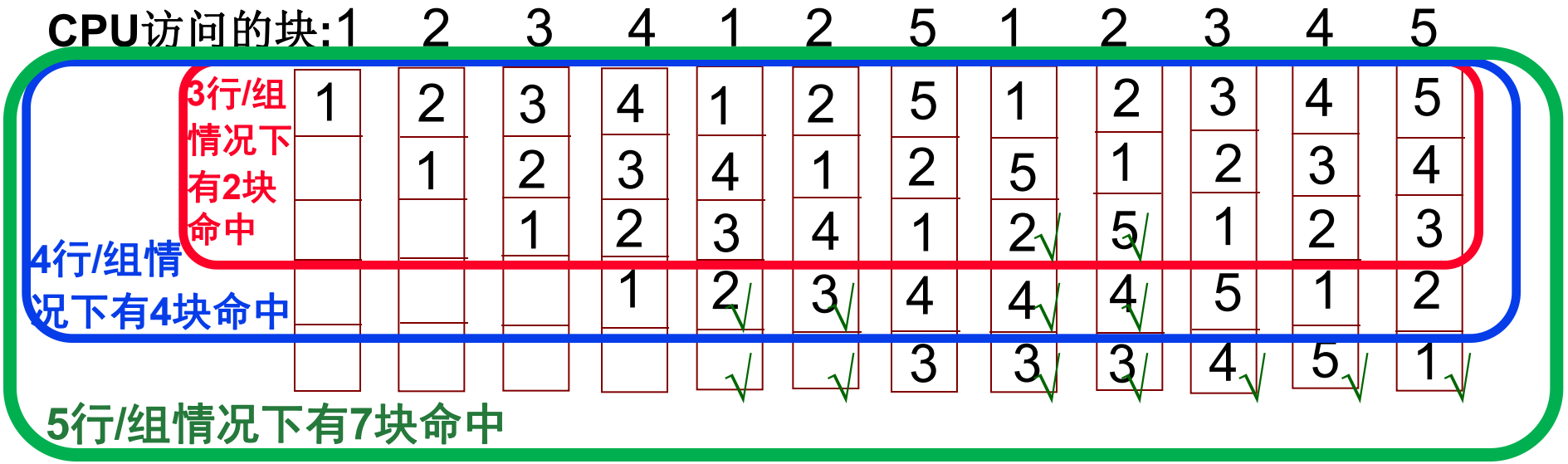
由此可见，FIFO不是一种栈算法，即命中率并不随组的增大而提高。

cache行替换算法-最近最少用(LRU)

- 总是把最近最少用的那一块淘汰掉。

总是把最长时间不看的书还回去！

- 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、 5行/组的情况。在下图中, 若某一组中(分别在3行/组、4行/组、5行/组的情况下)命中,则分别打勾 √



3行/组cache

4行/组cache

5行/组cache

cache行替换算法-最近最少用

- LRU是一种栈算法，它的命中率随组的增大而提高。
- 当分块局部化范围(即：某段时间集中访问的存储区)超过了Cache存储容量时，命中率变得很低。极端情况下，假设地址流是1,2,3,4,1 2,3,4,1,……，而Cache每组只有3行，那么，不管是FIFO，还是LRU算法，其命中率都为0。这种现象称为颠簸(Thrashing / PingPong)。
- LRU具体实现时，并不是通过移动块来实现的，而是通过给每个cache行设定一个计数器，根据计数值来记录这些主存块的使用情况。这个计数值称为**LRU位**。

具体实现

cache行替换算法-最近最少用

通过计数值来确定cache行中主存块的使用情况,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。**

CPU访问的块: 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

命中时，被访问行的计数器置0，未命中且该组未满时，新行计数器置为0，

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

举例

- 假定计算机系统主存空间大小为(32K字＝)32Kx16位，且有一个4K字的4路组相联Cache，主存和Cache之间的数据交换块的大小为64字。假定Cache开始为空，处理器顺序地从存储单元（字序号）0、1、…、4351(字)中取数，一共重复10次。设Cache比主存快10倍。采用LRU算法。试分析Cache的结构和主存地址的划分。说明采用Cache后速度提高了多少？采用MRU算法后呢？
- 答：假定主存按字编址。每字16位。
- 主存：32K字=512块 x 64字 / 块
- 主存总地址位数＝$\log_2{}^{32k}$= 15，行内偏移量位数＝字号＝$\log_2{}^{64}$= 6

Cache：4K字=（16组）x（4行 / 组）x（64 字 / 行）

主存地址划分为：

标志位	组号	字号
5	4	6

4352/64=68，所以访问过程实际上是对前68块连续访问10次。

LRU算法举例-第一次循环时块出现在cache中的情况

	第0 cache行	第1 cache行	第2 cache行	第3cache行
第0组	0/64(64替换出0)	16	32	48
第1组	1/65(65替换出1)	17	33	49
第2组	2/66(66替换出2)	18	34	50
第3组	3/67(67替换出3)	19	35	51
第4组	4	20	36	52
……	……	……	……	……
……	……	……	……	……
第15组	15	31	47	63

LRU算法：第一次循环,每一块只有第一字未命中,其余都命中;
以后9次循环,有20块的第一字未命中,其余都命中.
所以,命中率p为 (43520-68-9x20)/43520=99.43%
速度提高：tm/ta=tm/(tc+(1-p)tm)=10/(1+10x(1-p))=9.5倍

LRU算法举例-第二次循环时块出现在cache中及替换情况

	第0 cache行	第1 cache行	第2 cache行	第3cache行
第0组	64 /48	16 /0/64	32 /16	48 /32
第1组	65 /49	17 /1/65	33 /17	49 /33
第2组	66 /50	18 /2/66	34 /18	50 /34
第3组	67 /51	19 /3/67	35 /19	51 /35
第4组	4	20	36	52
……	……	……	……	……
……	……	……	……	……
第15组	15	31	47	63

第2次循环需要替换的4组

第1次循环留下的块

第1次循环留下的块

LRU算法：第2次循环,有0-3组的共20块需要新存入cache，替换出去

上表中“/”左边的块，这20块中的每块的第一个字未命中,其余都命中;

以后8次循环,情况类似，各有20块的第一字未命中,其余都命中.

所以,命中率p为 (43520-68-9x20)/43520=99.43%

速度提高：tm/ta=tm/(tc+(1-p)tm)=10/(1+10x(1-p))=9.5倍

举例（书中的表表示相同的意思）

	第0 cache行	第1 cache行	第2 cache行	第3cache行
第0组	0/64/48	16/0/64	32/16	48/32
第1组	1/65/49	17/1/65	33/17	49/33
第2组	2/66/50	18/2/66	34/18	50/34
第3组	3/67/51	19/3/67	35/19	51/35
第4组	4	20	36	52
……	……	……	……	……
……	……	……	……	……
第15组	15	31	47	63

LRU算法：第一次循环,每一块只有第一字未命中,其余都命中;
以后9次循环,有20块的第一字未命中,其余都命中.
所以,命中率p为 (43520-68-9x20)/43520=99.43%
速度提高：tm/ta=tm/(tc+(1-p)tm)=10/(1+10x(1-p))=9.5倍

举例

	第0 cache行	第1 cache行	第2 cache行	第3cache行
第0组	0/16/32/48	16/32/48/64	32/48/64/0	48/64/0/16
第1组	1/17/33/49	17/33/49/65	33/49/65/1	49/65/1/17
第2组	2/18/34/50	18/34/50/66	34/50/66/2	50/66/2/18
第3组	3/19/35/52	19/35/51/67	35/51/67/3	51/67/3/19
第4组	4	20	36	52
……	…	…	…	…
……	…	…	…	…
第15组	15组	31	47	63

MRU(最近最常用的块被替换出去)算法：第一次68字未命中；第2,3,4,6,7,8,10次各有4字未命中；第5,9次各有8字未命中；其余都命中.所以,命中率p为 (43520-68-7x4-2 x8)/43520=99.74%

速度提高：tm/ta=tm/(tc+(1-p)tm)=10/(1+10x(1-p))=9.77倍

写策略（cache-主存间存储一致性问题）

- 为何要保持在Cache和主存中数据的一致？
  - 因为Cache中的内容是主存块副本，当对Cache中的内容进行更新时，就存在Cache和主存如何保持一致的问题。
  - 以下情况也会出现“Cache一致性问题”
    - 当多个设备都允许访问主存时

      例如：I/O设备可直接读写内存时，如果Cache中的内容被修改，则I/O设备读出的对应主存单元的内容无效；若I/O设备修改了主存单元的内容，则Cache中对应的内容无效。
    - 当多个CPU都带有各自的Cache而共享主存时

      某个CPU修改了自身Cache中的内容，则对应的主存单元和其他CPU中对应的内容都变为无效。
- 写操作有两种情况
  - 写命中（Write Hit）：要写的单元已经在Cache中
  - 写不命中（Write Miss）：要写的单元不在Cache中

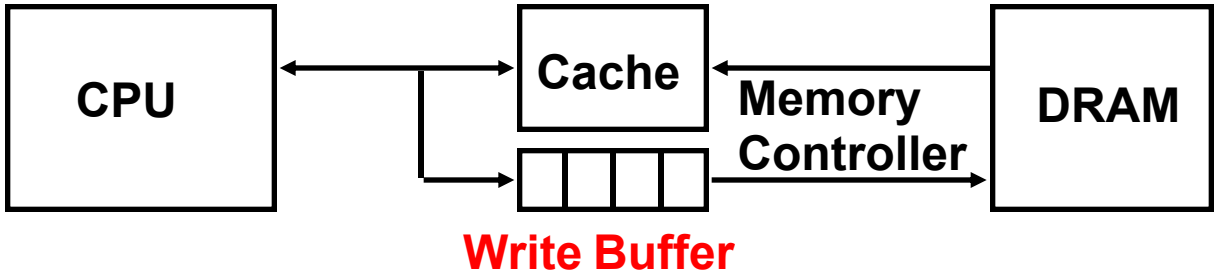
写策略（cache-主存间存储一致性问题）

- 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 为什么？

SKIP

Write Through中的Write Buffer



- 在 Cache 和 Memory之间加一个Write Buffer
  - CPU同时写数据到Cache和Write Buffer
  - Memory controller（存控）将缓冲内容写主存
- Write buffer (写缓冲) 是一个FIFO队列
  - 一般有4项
  - 在存数频率不 高时效果好
- 最棘手的问题
  - 当频繁写时，易使写缓存饱和，发生阻塞
- 如何解决写缓冲饱和？
  - 加一个二级Cache
  - 使用Write Back方式的Cache

BACK

写策略（cache-主存间存储一致性问题）

问题1：以下描述的是哪种写策略？

Write Through 、Write Allocate !

问题2：如果用非写分配，则如何修改算法？

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X == TAG(i) , for some cache line i*

READ: return DATA[I]
WRITE: change DATA[I]; Start Write to Mem[X]

MISS: *X not found in TAG of any cache line*

REPLACEMENT SELECTION:
Select some line k to hold Mem[X]

READ: Read Mem[X]
Set TAG[k] = X, DATA[k] = Mem[X]

WRITE: Start Write to Mem[X]
~~Set TAG[k] = X, DATA[k] = new Mem[X]~~

BACK

写策略2：Write Back算法

问题：以下算法描述的是哪种写策略？

Write Back 、Write Allocate !

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X = TAG(i) , for some cache line I*

READ: return DATA(i)
WRITE: change DATA(i); ~~Start Write to Mem[X]~~

MISS: *X not found in TAG of any cache line*

REPLACEMENT SELECTION:
Select some line k to hold Mem[X]
Write Back: Write Data(k) to Mem[Tag[k]]

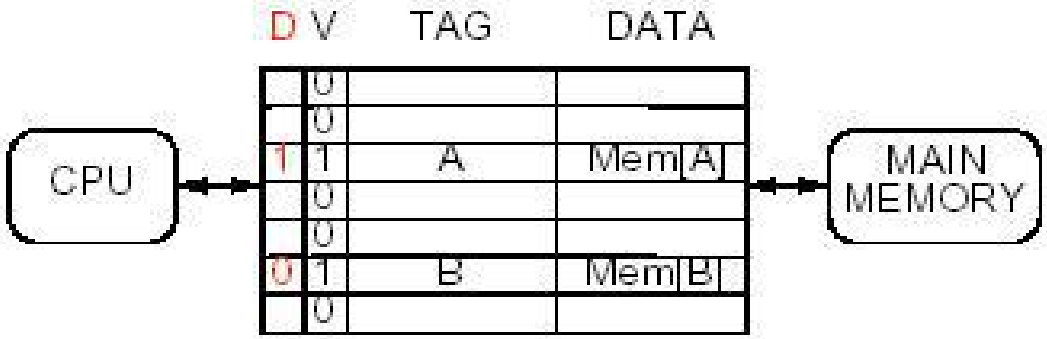
READ: Read Mem[X]
Set TAG[k] = X, DATA[k] = Mem[X]

WRITE: ~~Start Write to Mem[X]~~
Set TAG[k] = X, DATA[k] = new Mem[X]

Is write-back worth the trouble? Depends on (1) cost of write; (2) consistency issues.

写策略2：Write Back中的修改（D“脏”）位

Write-back w/ "Dirty" bits



BACK

ON REFERENCE TO Mem[X]: Look for X among tags...

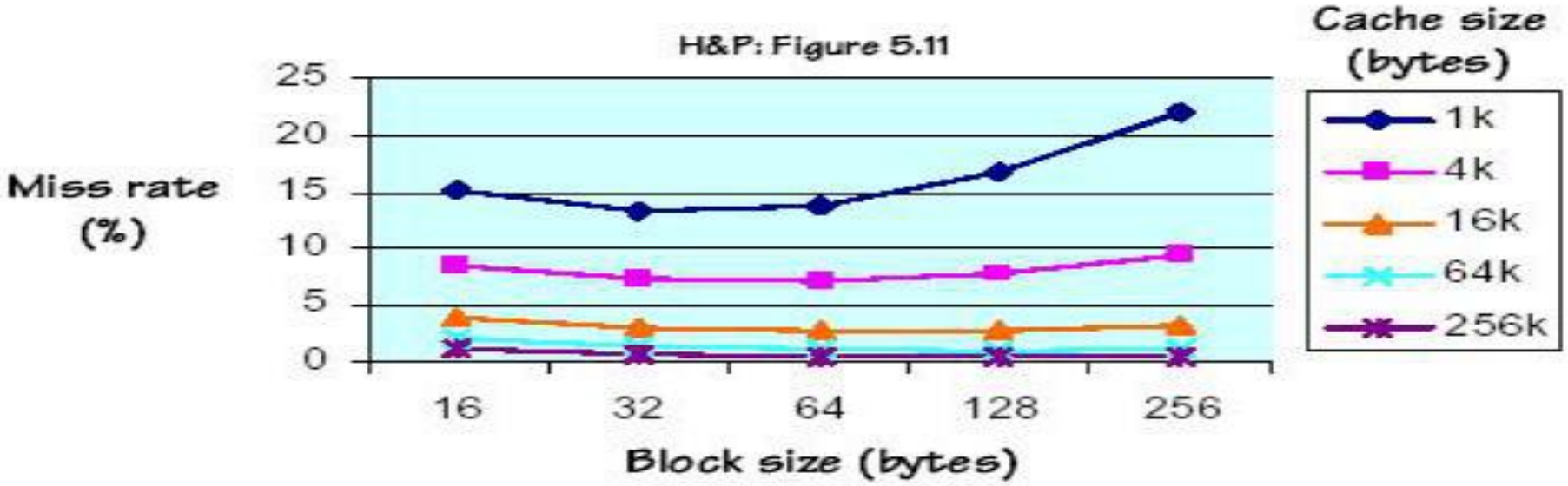
HIT: *X = TAG(i) , for some cache line I*
READ: return DATA(i)
WRITE: change DATA(i); ~~Start Write to Mem[X]~~ D[i]=1

MISS: *X not found in TAG of any cache line*
REPLACEMENT SELECTION:
Select some line k to hold Mem[X]
If D[k] == 1 (Write Back) Write Data(k) to Mem[Tag[k]]
READ: Read Mem[X]; Set TAG[k] = X, DATA[k] = Mem[X], D[k]=0
WRITE: ~~Start Write to Mem[X]~~ D[k]=1
Set TAG[k] = X, DATA[k] = new Mem[X]

Cache大小、Block大小和缺失率的关系

Cache性能由缺失率确定

而缺失率与Cache大小、Block大小等有关



- spatial locality: larger blocks → reduce miss rate
- fixed cache size: larger blocks
  - → fewer lines in cache
  - → higher miss rate, especially in small caches

Cache大小：Cache越大，Miss率越低，但成本越高！

Block大小：Block大小与Cache大小有关，且不能太大，也不能太小！

系统中的Cache数目

- 刚引入Cache时只有一个Cache。近年来多Cache系统成为主流
- 多Cache系统中，需考虑两个方面：

  [1] 单级/多级？

  外部(Off-chip)Cache:不做在CPU内而是独立设置一个Cache

  片内(On-chip)Cache: 将Cache和CPU作在一个芯片上

  单级Cache：只用一个片内Cache

  多级Cache：同时使用L1 Cache和L2 Cache，甚至有L3 Cache，L1 Cache更靠近CPU，其速度比L2快，其容量比L2小

  [2] 联合/分立？

  分立：指数据和指令分开存放在各自的数据和指令Cache中

  一般L1 Cache都是分立Cache，为什么？

  L1 Cache的命中时间比命中率更重要！为什么？

  联合：指数据和指令都放在一个Cache中

  一般L2 Cache都是联合Cache，为什么？

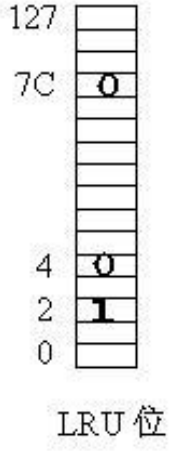
  L2 Cache的命中率比命中时间更重要！为什么？

  因为缺失时需从主存取数，并要送L1和L2cache，损失大！

实例：奔腾机的Cache组织

主存：4GB=2^{20}x 2^{7}块x 2^{5}B/块

Cache：8KB=128组x2行/组



替换算法：

LRU，每组一位LRU位

0：下次淘汰第0路

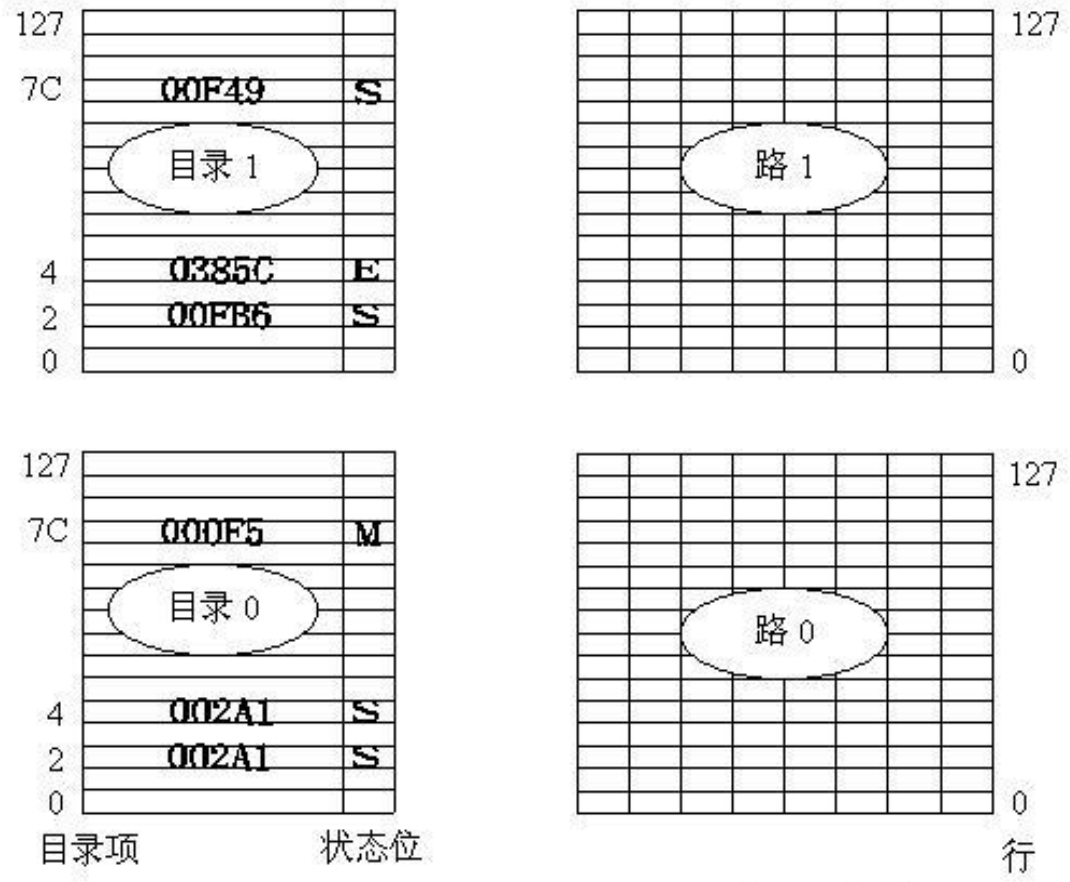
1：下次淘汰第1路

写策略：

默认为Write Back，可动态设置为Write Through。

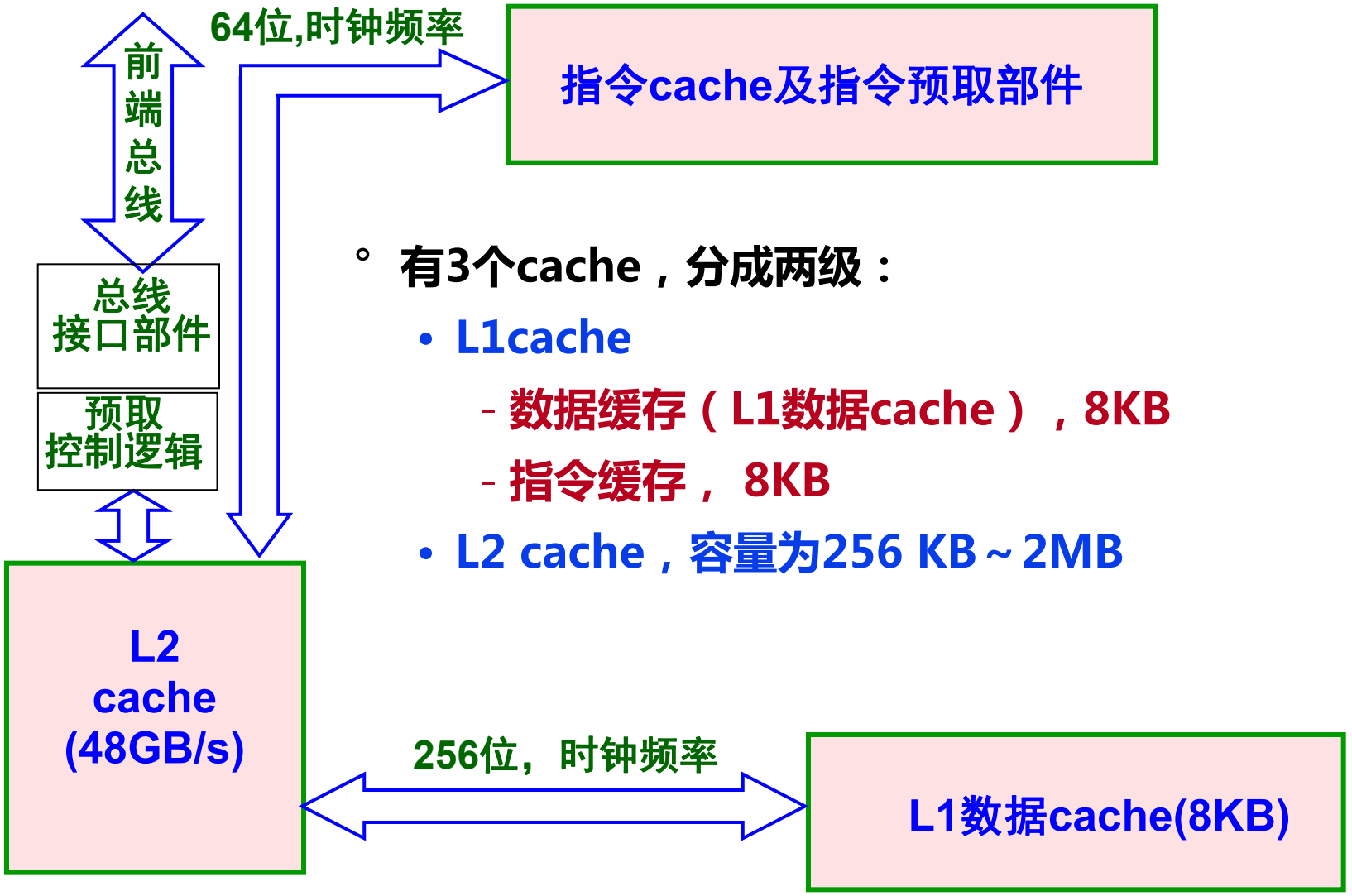
Cache一致性：

支持MESI协议



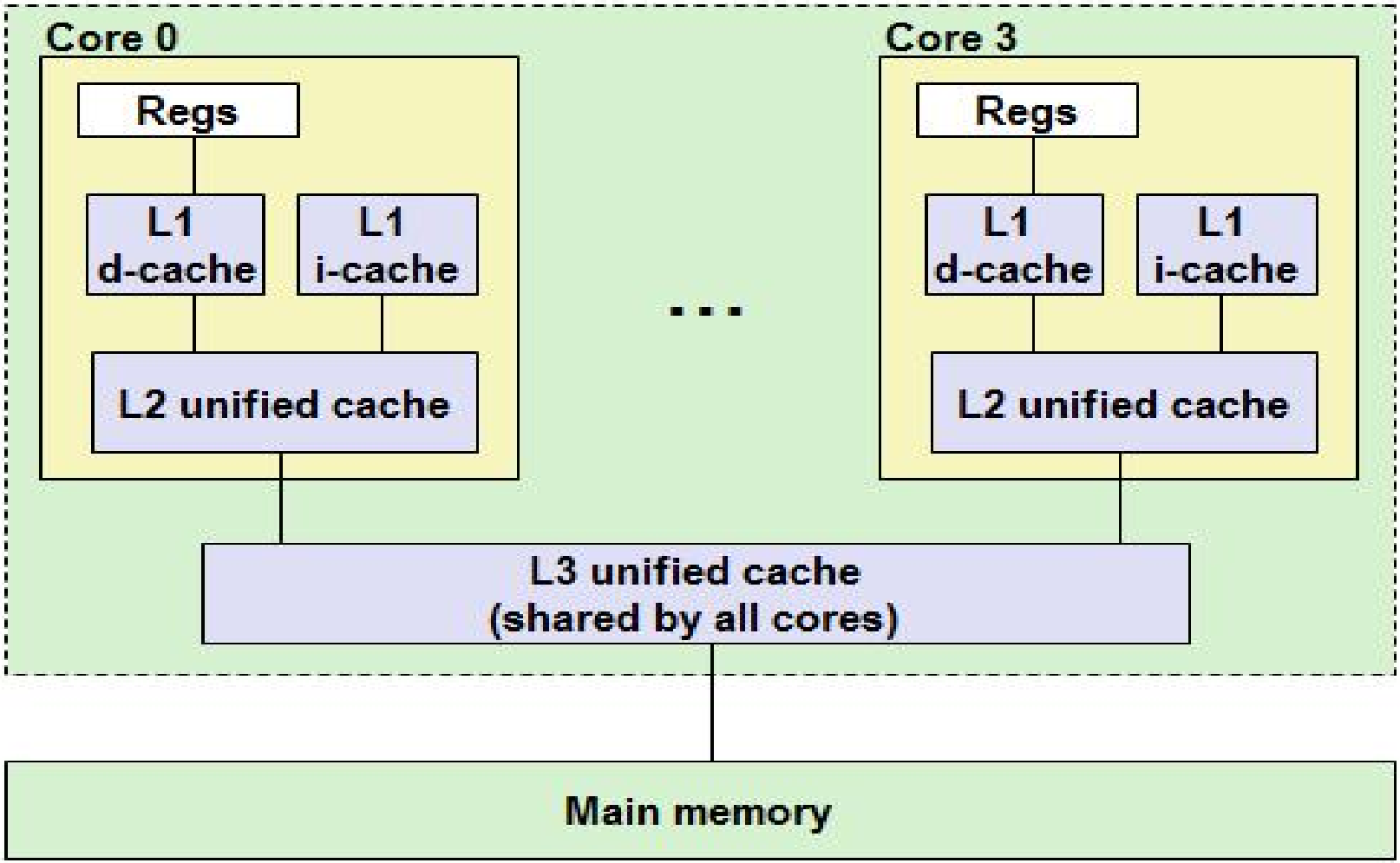
Pentium 内部数据 Cache 的结构

实例：Pentium 4的cache存储器



- 有3个cache，分成两级：
  - L1cache
    - 数据缓存（L1数据cache），8KB
    - 指令缓存， 8KB
  - L2 cache，容量为256 KB～2MB

实例：Intel Core i7处理器的cache结构



i-cache和d-cache都是32KB、8路、4个时钟周期；L2 cache：256KB、8路、11个时钟周期。所有核共享的L3 cache：8MB、16路、30~40个时钟周期。Core i7中所有cache的块大小都是64B

系统中的Cache数目

- 刚引入Cache时只有一个Cache。近年来多Cache系统成为主流
- 多Cache系统中，需考虑两个方面：

[1] 单级/多级？

片内(On-chip)Cache: 将Cache和CPU作在一个芯片上

外部(Off-chip)Cache:不做在CPU内而是独立设置一个Cache

单级Cache：只用一个片内Cache

多级Cache：同时使用L1 Cache和L2 Cache，有些高端系统甚至有L3 Cache，
L1 Cache更靠近执行部件，速度比L2快，容量比L2小

[2] 联合/分立？

分立：指数据和指令分开存放在各自的数据Cache和指令Cache中

一般L1 Cache都是分立Cache，为什么？

L1 Cache的命中时间比命中率更重要！为什么？

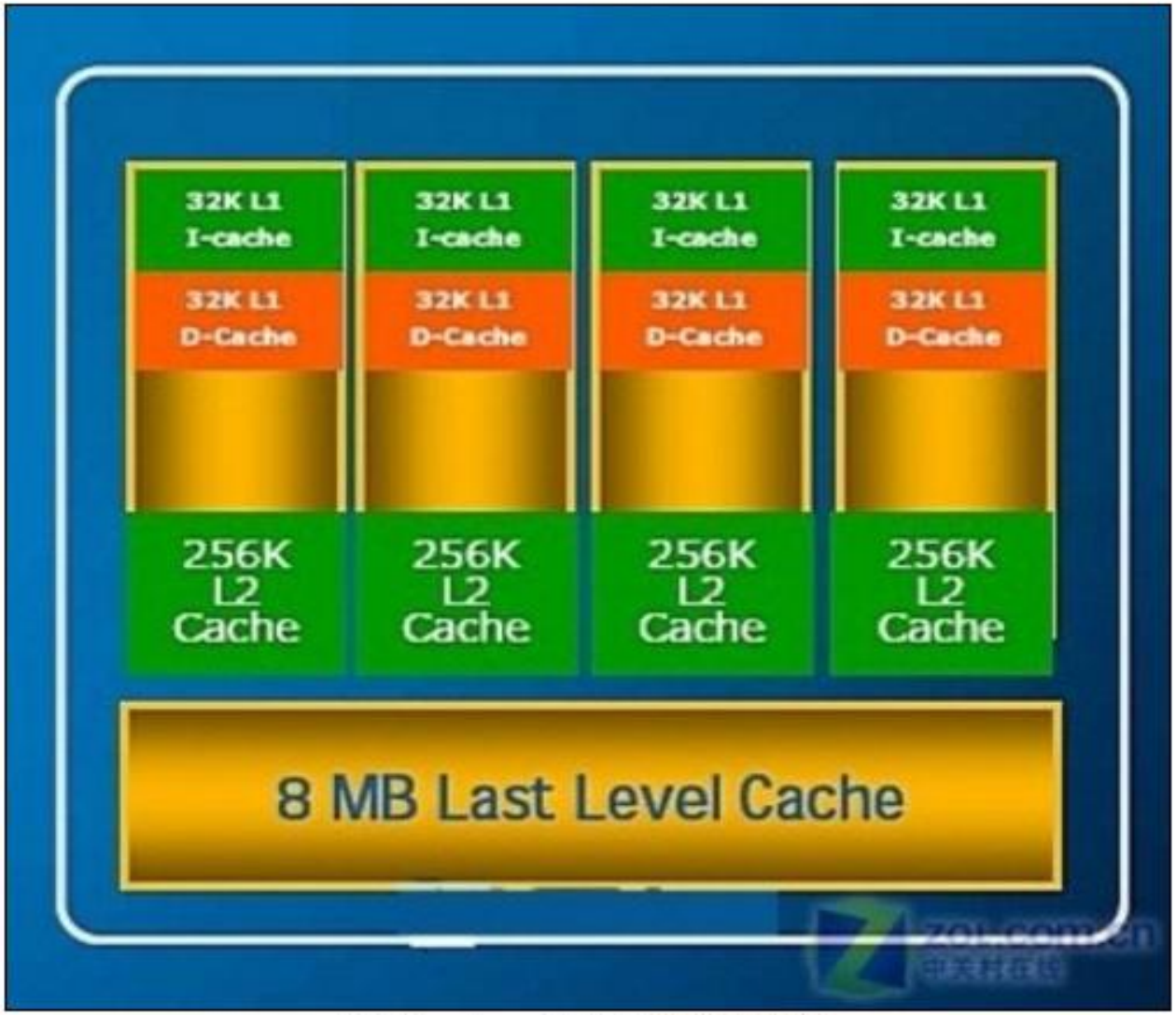
联合：指数据和指令都放在一个（联合的）Cache中

一般L2 Cache都是联合Cache，为什么？

L2 Cache的命中率比命中时间更重要！为什么？

因为缺失时需从主存取数，并要送L1和L2cache，损失大！

多核处理器中的多级Cache



Nehalem Core i7处理器缓存结构图

Per core:

-32KB, 4-way L1 $I

-32KB, 8-way L1 $D

-256KB, 8-way L2

Shared

- 8 MB, 16-way L3

多级cache的性能

- 采用L2 Cache的系统，其缺失损失的计算如下：
  - 若L2 Cache包含所请求信息，则缺失损失为L2 Cache访问时间
  - 否则访问主存，并取到L1 Cache和L2 Cache（缺失损失更大）
- 例：某处理器在无cache缺失时CPI为1，时钟频率为5GHz。假定访问一次主存的时间（包括所有的缺失处理）为100ns，平均每条指令在L1 Cache中的缺失率为2%。若增加一个L2 Cache，其访问时间为5ns，而且容量足够大到使全局缺失率减为0.5%，问处理器执行指令的速度提高了多少？

  解：如果只有一级Cache，则缺失只有一种。即L1缺失(需访问主存)，其缺失损失为：100nsx5GHz=500个时钟，

  CPI=(无cache缺失时CPI为1)1+(有L1cache缺失时)1500x2%=11.0。

  如果有二级Cache，则有两种缺失：

  L1缺失(需访问L2 Cache)：5nsx5GHz=25个时钟

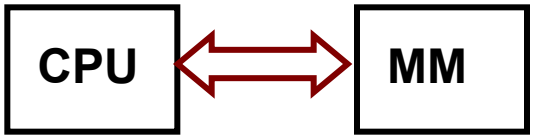
  L1和L2都缺失(需访问主存)：500个时钟

  CPI=无cache缺失时CPI+有L1cache缺失时Cyc+有L2cache缺失时Cyc

  因此，CPI=1+25x2%+500x0.5%=4.0,二者的性能比为11.0/4.0=2.8倍！

设计支持Cache的存储器系统

- 指令执行若发生Cache缺失，必须到DRAM中取数据或指令
- 在DRAM和Cache之间传输的单位是Block
- 问题：怎样的存储器组织使得Block传输最快（缺失损失最小）？



假定存储器访问过程：

CPU发送地址到内存：1个总线时钟

访问内存的初始化时间：10个总线时钟

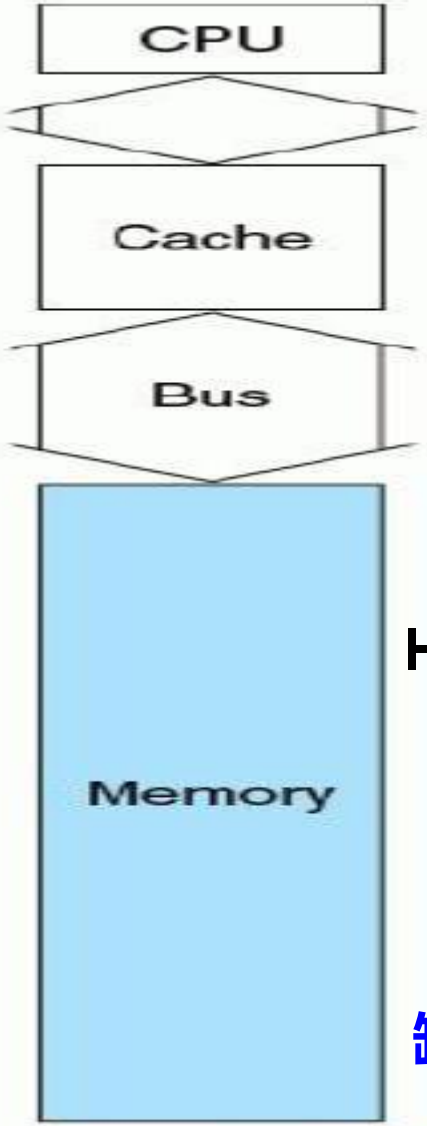
从总线上传送一个字：1个总线时钟

相加=(1+10+1)=12个总线时钟

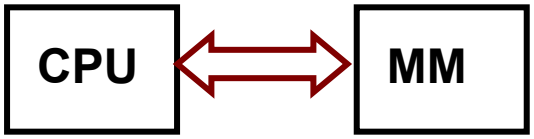
假定一个Block有4个字，则缺失损失各为多少时钟？

可以有三种不同的组织形式！（下页）

设计支持Cache的存储器系统



a. One-word-wide memory organization

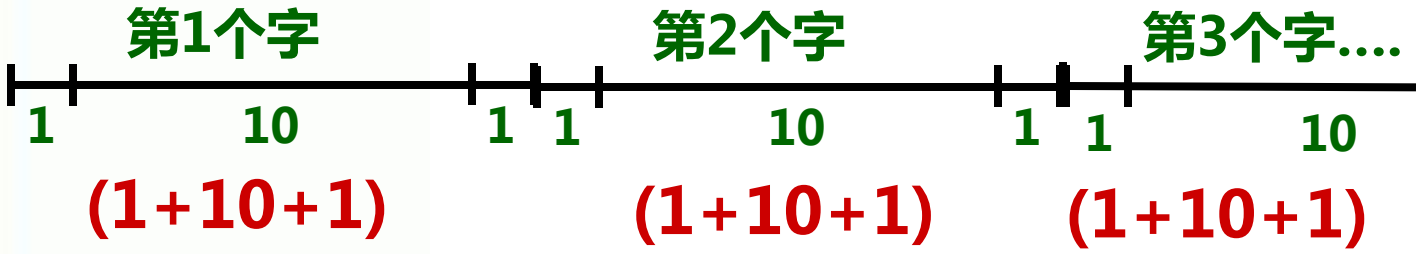


假定存储器访问过程：

CPU发送地址到内存：1个总线时钟

内存访问时间：10个总线时钟

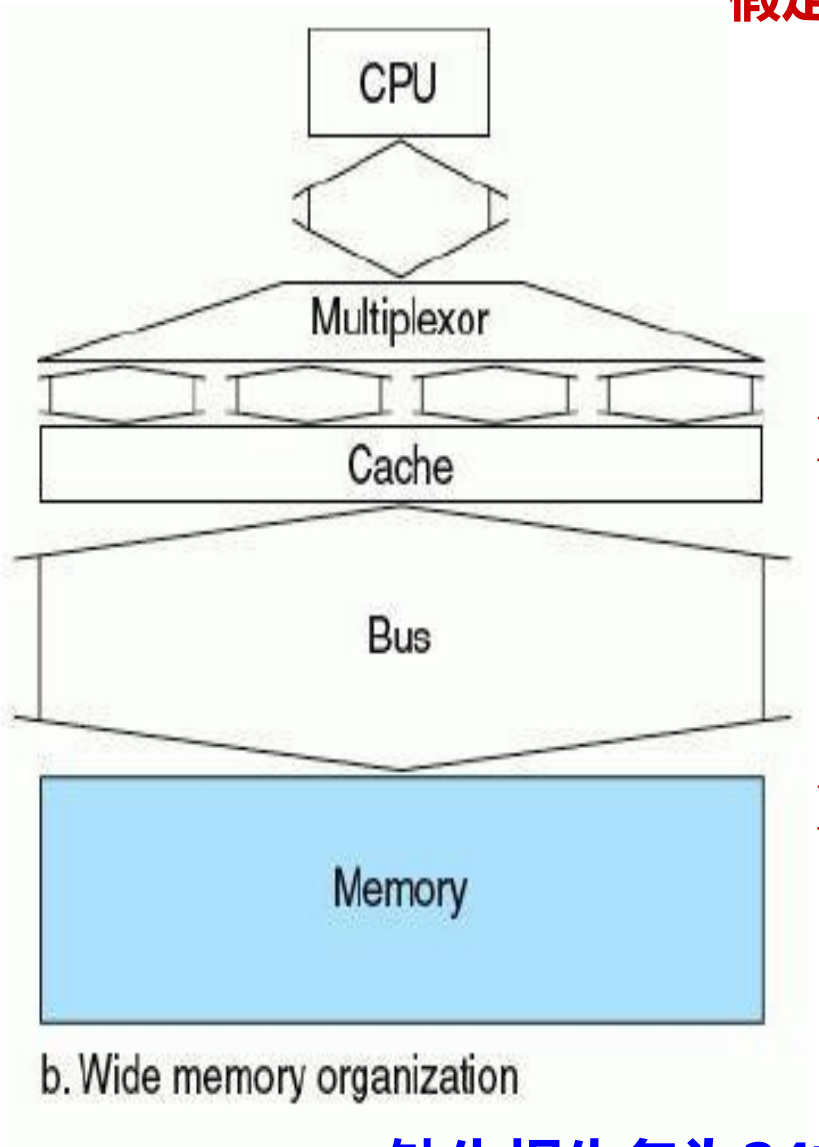
从总线上传送一个字：1个总线时钟



需要4x(1+10+1)=48个总线时钟

缺失损失为48个时钟周期。硬件代价小，但速度慢！

设计支持Cache的存储器系统



b. Wide memory organization

假定存储器访问过程：

CPU发送地址到内存：1个总线时钟

内存访问时间：10个总线时钟

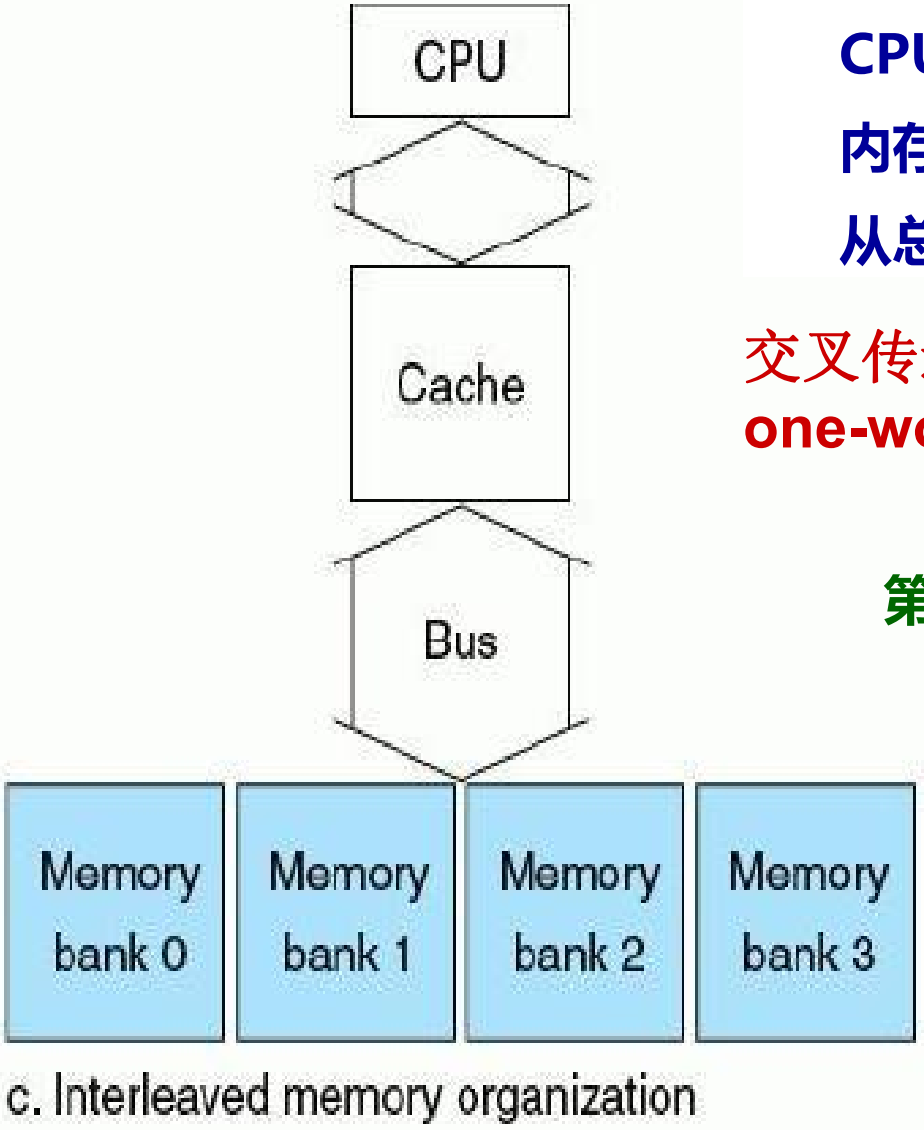
从总线上传送一个字：1个总线时钟

如果每次能同时发送2个字(Two-word)，
只需要2次发送，就能发送4个字
需要：2x(1+10+1)=24个总线时钟

如果每次能同时发送4个字(Four-word)，
只需要1次发送，就能发送4个字
需要: 1+10+1=12个总线时钟

缺失损失各为24或12个时钟周期速度快，但硬件代价大！

设计支持Cache的存储器系统



c. Interleaved memory organization

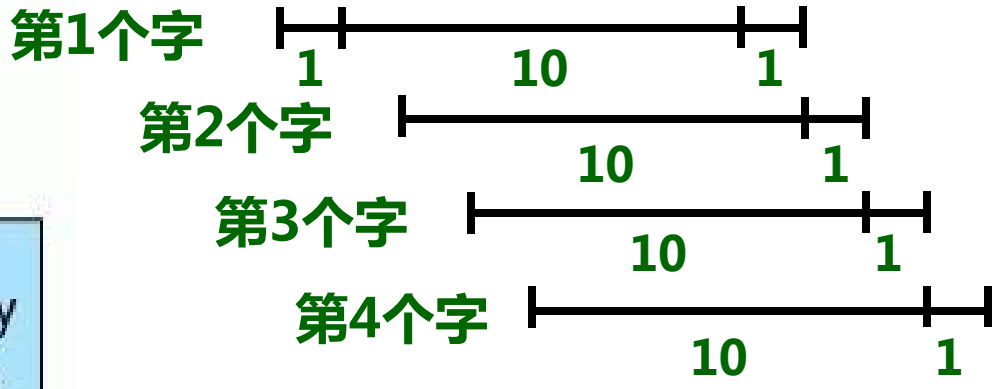
假定存储器访问过程：

CPU发送地址到内存：1个总线时钟

内存访问时间：10个总线时钟

从总线上传送一个字：1个总线时钟

交叉传送Interleaved four banks
one-word: 1+1x10+4x1=15



缺失损失为15个时钟周期

代价小，而且速度快！

缓存在现代计算机中无处不在

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte word	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware
L1 cache	32-byte block	On-chip L1 cache	1	Hardware
L2 cache	32-byte block	Off-chip L2 cache	10	Hardware
Virtual memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Cache和程序性能

- 程序的性能指执行程序所用的时间
- 程序执行所用时间与程序执行时访问指令和数据所用的时间有很大关系，而指令和数据的访问时间与cache命中率、命中时间和缺失损失有关
- 对于给定的计算机系统而言，命中时间和缺失损失是确定的，因此，指令和数据的访存时间主要由cache命中率决定
- cache命中率主要由程序的空间局部性和时间局部性决定。因此，为了提高程序的性能，程序员须编写出具有良好访问局部性的程序
- 考虑程序的访问局部性通常在数据的访问局部性上下工夫
- 数据的访问局部性主要是指数组、结构等类型数据访问时的局部性，这些数据结构的数据元素访问通常是通过循环语句进行的，所以，如何合理地处理循环对于数据访问局部性来说是非常重要的。

Cache和程序性能举例

- 举例：某32位机器主存地址空间大小为256 MB，按字节编址。指令cache和数据cache均有8行，主存块为64B，数据cache采用直接映射。假定编译时i, j, sum均分配在寄存器中，数组a按行优先方式存放，其首址为320。

程序 A：

```
int a[256][256];
……
int sum_array1 ( )
{
        int i, j, sum = 0;
        for ( i = 0; i < 256; i++)
                for (j = 0; j < 256; j++)
                        sum += a[i] [j];
        return sum;
}
```

程序 B：

```
int a[256][256];
……
int sum_array2 ( )
{
        int i, j, sum = 0;
        for ( j = 0; j < 256; j++)
                for ( i = 0; i < 256; i++)
                        sum += a[i] [j];
        return sum;
}
```

（1）不考虑用于一致性和替换的控制位，数据cache的总容量为多少？

（2）a[0][31]和a[1][1]各自所在主存块对应的cache行号分别是多少？

（3）程序A和B的数据访问命中率各是多少？哪个程序的执行时间更短？

Cache和程序性能举例

- 举例：某32位机器主存地址空间大小为256 MB，按字节编址。指令cache和数据cache均有8行，主存块为64B，数据cache采用直接映射。假定编译时i, j, sum均分配在寄存器中，数组a按行优先方式存放，其首址为320。

（1）主存地址空间大小为256MB，因而主存地址为28位，其中6位为块内地址，3位为cache行号（行索引），标志信息有28-6-3=19位。在不考虑用于cache一致性维护和替换算法的控制位的情况下，数据cache的总容量为：8×(19+1+64×8)=4256位=532字节 。

（2）a[0][31]的地址为320+4×31=444，[444/64]=6（取整），因此a[0][31]对应的主存块号为6。6 mod 8=6，对应cache行号为6。

或: 444=0000 0000 0000 0000 000 110 111100B，中间3位110为行号（行索引），因此，对应的cache行号为6。 a[1][1]对应的cache行号为：[(320+4×(1×256+1))/64] mod 8=5。

（3）A中数组访问顺序与存放顺序相同，共访问64K次，占4K个主存块；首地址位于一个主存块开始，故每个主存块总是第一个元素缺失，其他都命中，共缺失4K次，命中率为1-4K/64K=93.75%。

方法二：每个主存块的命中情况一样。对于一个主存块，包含16个元素，需访存16次，其中第一次不命中，因而命中率为15/16=93.75%。

B中访问顺序与存放顺序不同，依次访问的元素分布在相隔256×4=1024的单元处，它们都不在同一个主存块中,cache共8行，一次内循环访问16块，故再次访问同一块时，已被调出cache，因而每次都缺失，命中率为0。

第四讲：虚拟存储器（Virtual Memory）

虚拟地址空间

虚拟存储器的实现

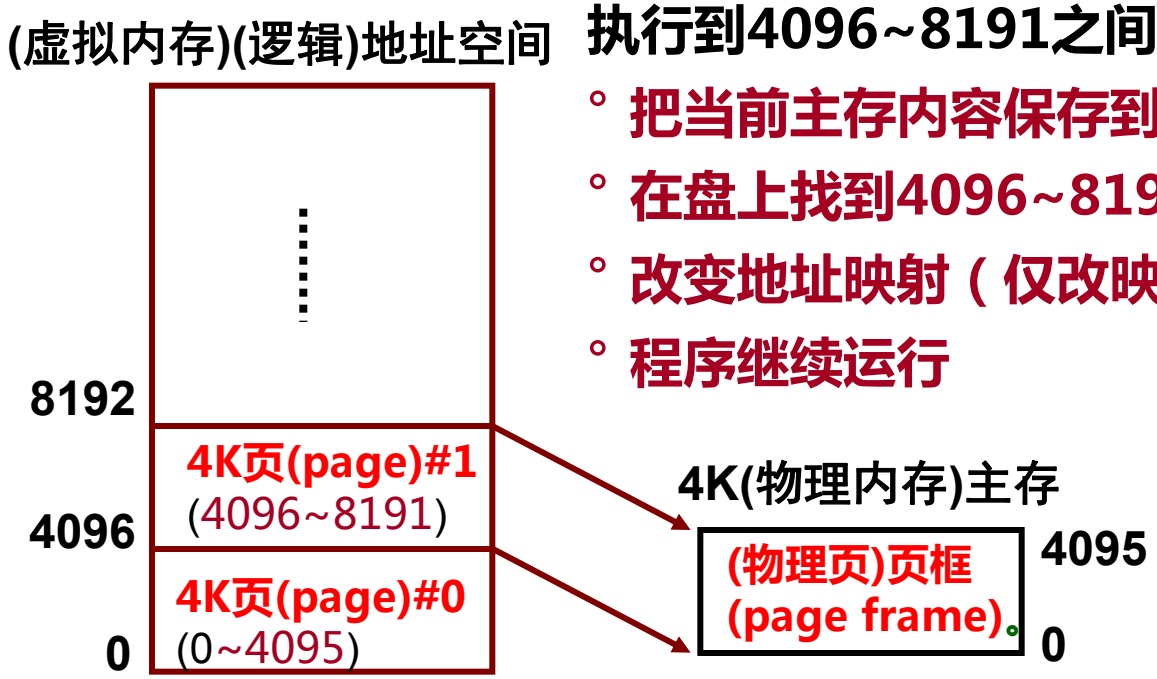
存储保护

早期分页方式的概念

早期：程序员自己管理主存，通过分解程序并覆盖主存的方式执行程序

- 1961年(英)曼切斯特研究人员提出一种自动执行(覆盖)overlay的方式
- 动机：把程序员从大量繁琐的存储管理工作中解放出来，使得程序员编程时不用管主存容量的大小。
- 基本思想：把地址空间和主存容量的概念区分开来。程序员在地址空间里编写程序，而程序则在真正的内存中运行。由一个专门的机制实现地址空间和实际主存之间的映射。举例说明:
- 例如，当时的一种典型计算机，其指令中给出的主存地址为16位,地址空间为0、1、2...、65535(64K)组成的地址集合，即地址空间大小为2^{16}=64K
- 而物理主存容量只有4K字，则指令可寻址范围是多少？程序员编写程序的空间（地址空间，可寻址空间）比执行程序的空间（主存容量）大得多，怎么自动执行程序呢？

早期分页方式的实现



执行到4096~8191之间的程序段时，自动做：

- 把当前主存内容保存到磁盘上；
- 在盘上找到4096~8191之间的程序段并读入主存
- 改变地址映射（仅改映射区间号（页号））
- 程序继续运行

后来把区间称为页(page)，主存中存放页的区域称为页框(page frame)。

早期主存只有一个页框！

- 将地址空间划分成4K大小的区间，装入内存的总是其中的一个区间
- 执行到某个区间时，把该区间的地址自动映射到0~4095之间，例如：
  - (0~4095)→0, (4096~8191) →1, ……,
- 程序员在0~65535范围内写程序，完全不用管在多大的主存空间上执行，所以，这种方式对程序员来说，是透明的！
- 可寻址的地址空间是一种虚拟内存！

分页（Paging）

- 基本思想：
  - (物理)内存被分成固定长且比较小的存储块（页框、实页、物理页）
  - 每个进程也被划分成固定长的程序块（页、虚页、逻辑页）
  - 程序块可装到(物理)存储器中可用的存储块中
  - 无需用连续页框来存放一个进程
  - 操作系统为每个进程生成一个页表(page table)
  - 通过页表实现逻辑地址向物理地址转换（Address Mapping ）
- 逻辑地址（Logical Address）：
  - 程序中指令所用地址(进程所在地址空间)，也称为虚拟地址（Virtual Address，简称VA）
- 物理地址（Physical Address，简称PA）：
  - 存放指令或数据的实际()内存地址，也称为实地址、主存地址。

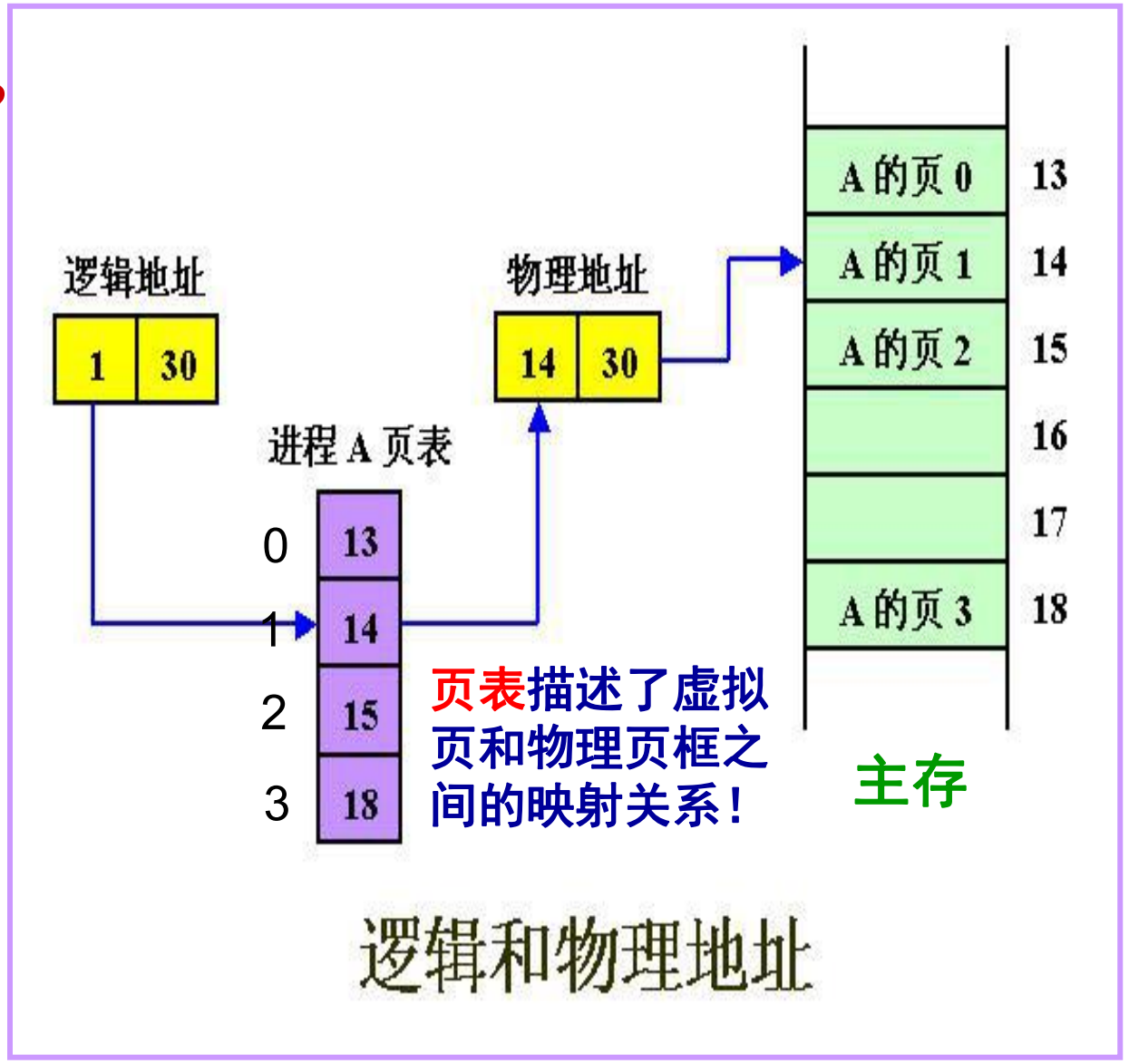
分页（Paging）

问题：是否需要将一个进程的全部都装入内存？

根据程序访问局部性可知：可把当前活跃的页面调入主存，其余留在磁盘上！

采用 “按需调页 Demand Paging” 方式分配主存！这就是虚拟存储管理概念

优点：浪费的空间最多是最后一页的部分！



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，由操作系统进行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。

SKIP

虚拟地址空间

- Linux在X86上的虚拟地址空间

（其他Unix系统的设计类此）

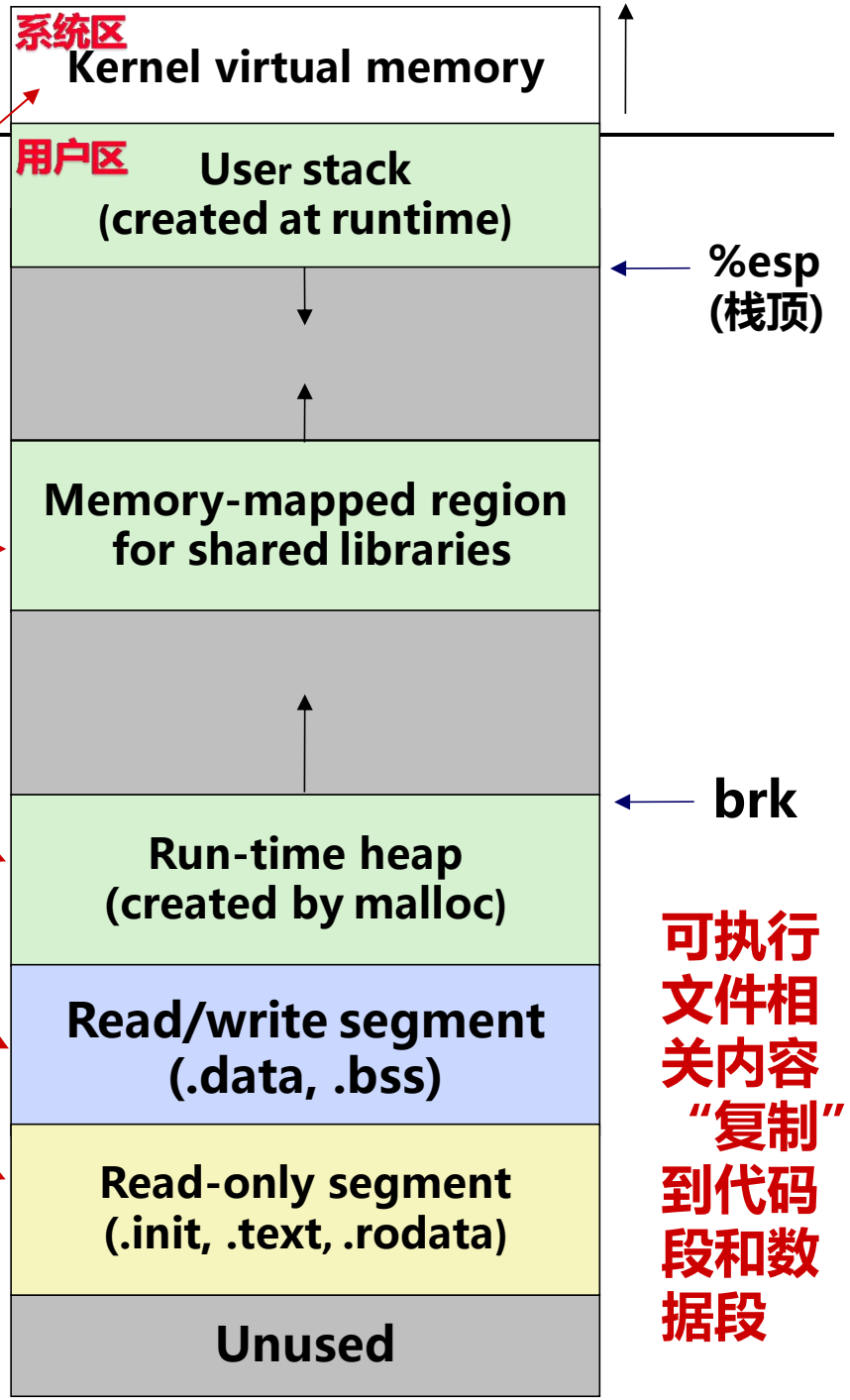
系统区：内核空间（Kernel）

用户区：

- 用户栈（User Stack）
- 共享库（Shared Libraries）
- 堆（heap）
- 可读写数据（Read/Write Data）
- 只读数据（Read-only Data）
- 代码（Code）

问题：加载时是否真正从磁盘调入信息到主存？

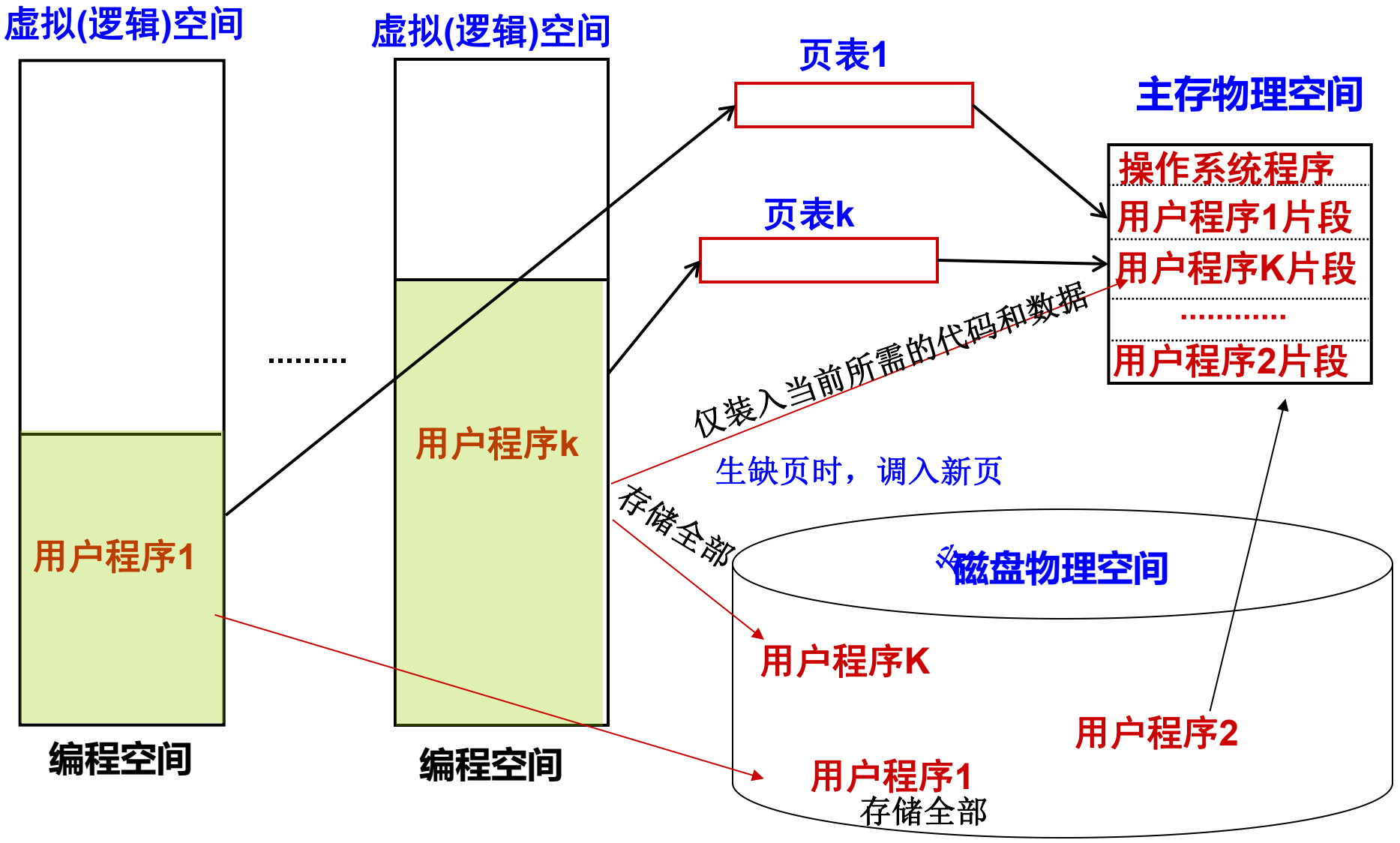
实际上不会从磁盘调入，只是将虚拟页和磁盘上的数据/代码建立对应关系，称为“映射”。



可执行文件相关内容“复制”到代码段和数据段

虚拟存储技术的实质

通过页表建立虚拟空间和物理空间之间的映射!



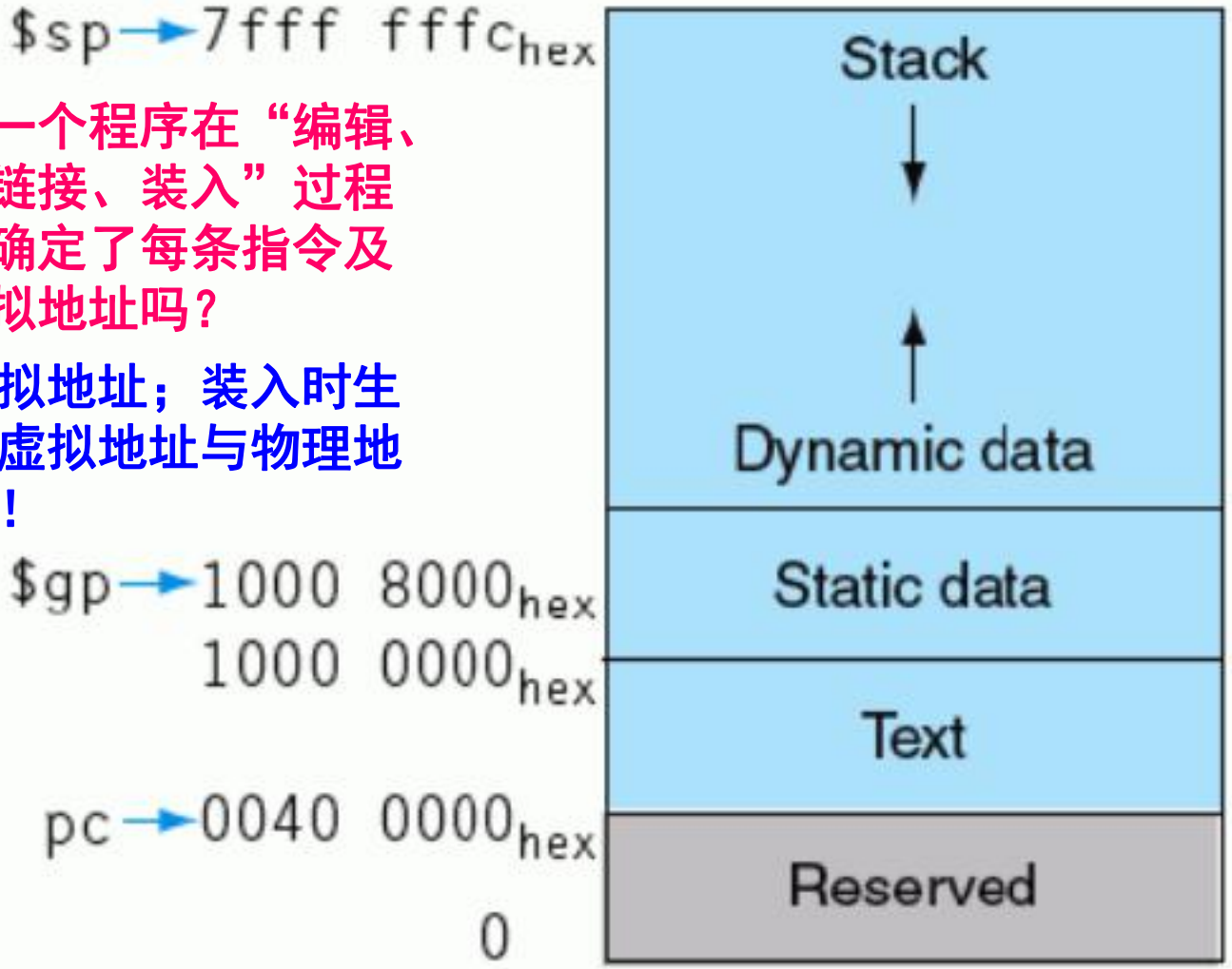
BACK

MIPS程序和数据的存储器分配

问题：你知道一个程序在“编辑、编译、汇编、链接、装入”过程中的哪个环节确定了每条指令及其操作数的虚拟地址吗？

链接时确定虚拟地址；装入时生成页表以建立虚拟地址与物理地址之间的映射！

请参考《深入理解计算机系统》和有关Linux内核分析方面的资料。



每个用户程序都有相同的虚拟地址空间！

这就是每个进程的虚拟（逻辑）地址空间！

虚拟存储器管理

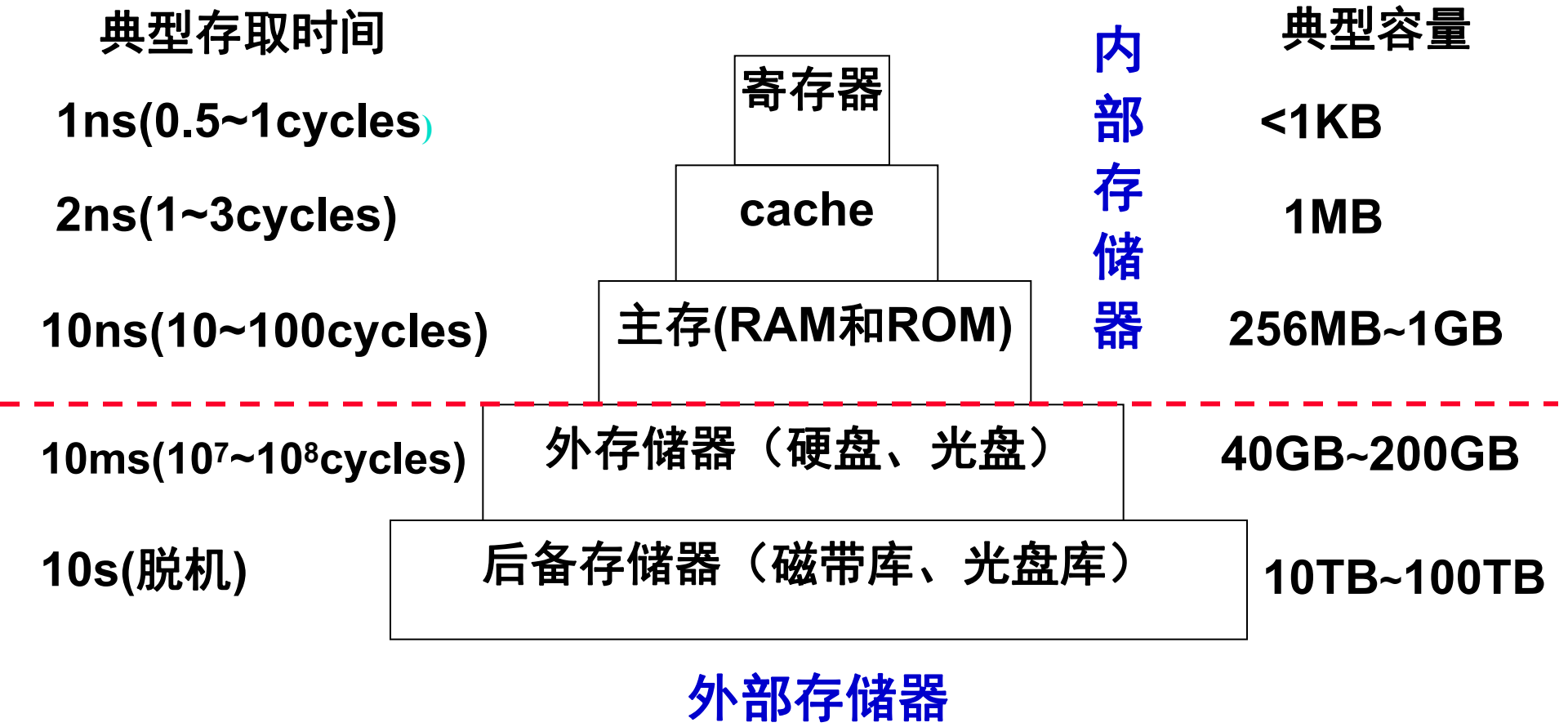
实现虚拟存储器管理，需考虑：

- **“块”大小（在虚拟存储器中”块”被称为“页 / Page”)应多大？**
- **主存与辅存的空间如何分区管理?**
- **程序块 / 存储块之间如何映像?**
- **逻辑地址和物理地址如何转换，转换速度如何提高?**
- **主存与辅存之间如何进行替换（与Cache所用策略相似）?**
- **页表如何实现，页表项中要记录哪些信息?**
- **如何加快访问页表的速度?**
- **如果要找的内容不在主存，怎么办?**
- **如何保护进程各自的存储区不被其他进程访问?**

这些问题是由硬件和OS共同协调解决的！

有三种虚拟存储器实现方式： 分页式、分段式、段页式

存储器的层次结构



列出的时间和容量会随时间变化，但数量级相对关系不变。

“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? 加快指令执行

页表结构

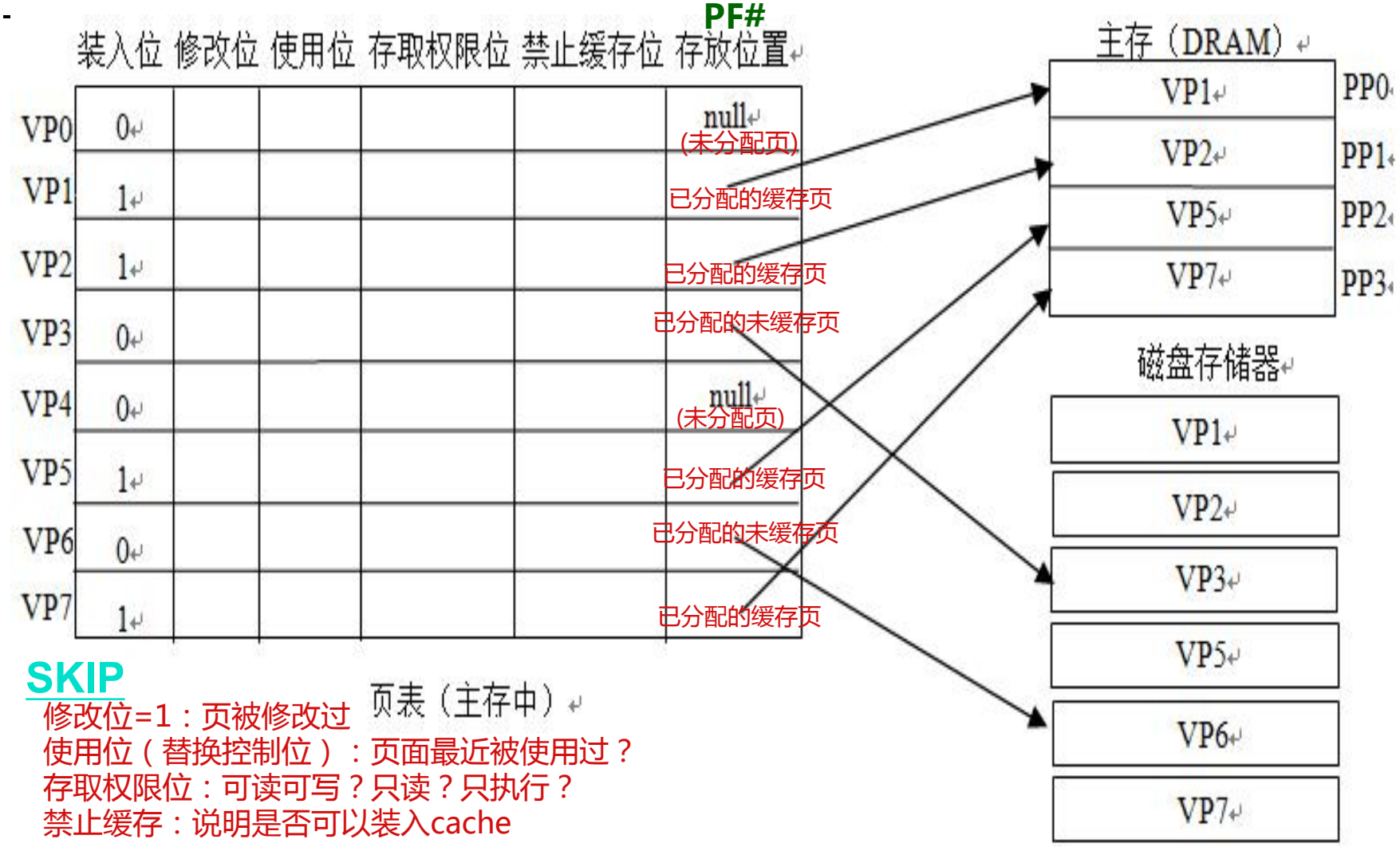
◆ 页表首址记录在页表基址寄存器中

页表首地址

	装入位	修改位	替换控制位	其他	实页号（8 进制）
0 虚页	1				11
1 虚页	1				13
2 虚页	1				16
3 虚页	1				10
4 虚页	1				14

- 每个进程有一个页表，其中有装入位、修改（Dirt）位、替换控制位、访问权限位、禁止缓存位、实页号。
- 一个页表的项数由什么决定？理论上由虚拟地址空间大小决定。
- 每个进程的页表大小一样吗？ 各进程有相同虚拟空间，故理论上一样。实际大小看具体实现方式，如“空洞”页面如何处理等

主存中的页表示例



SKIP

修改位=1：页被修改过

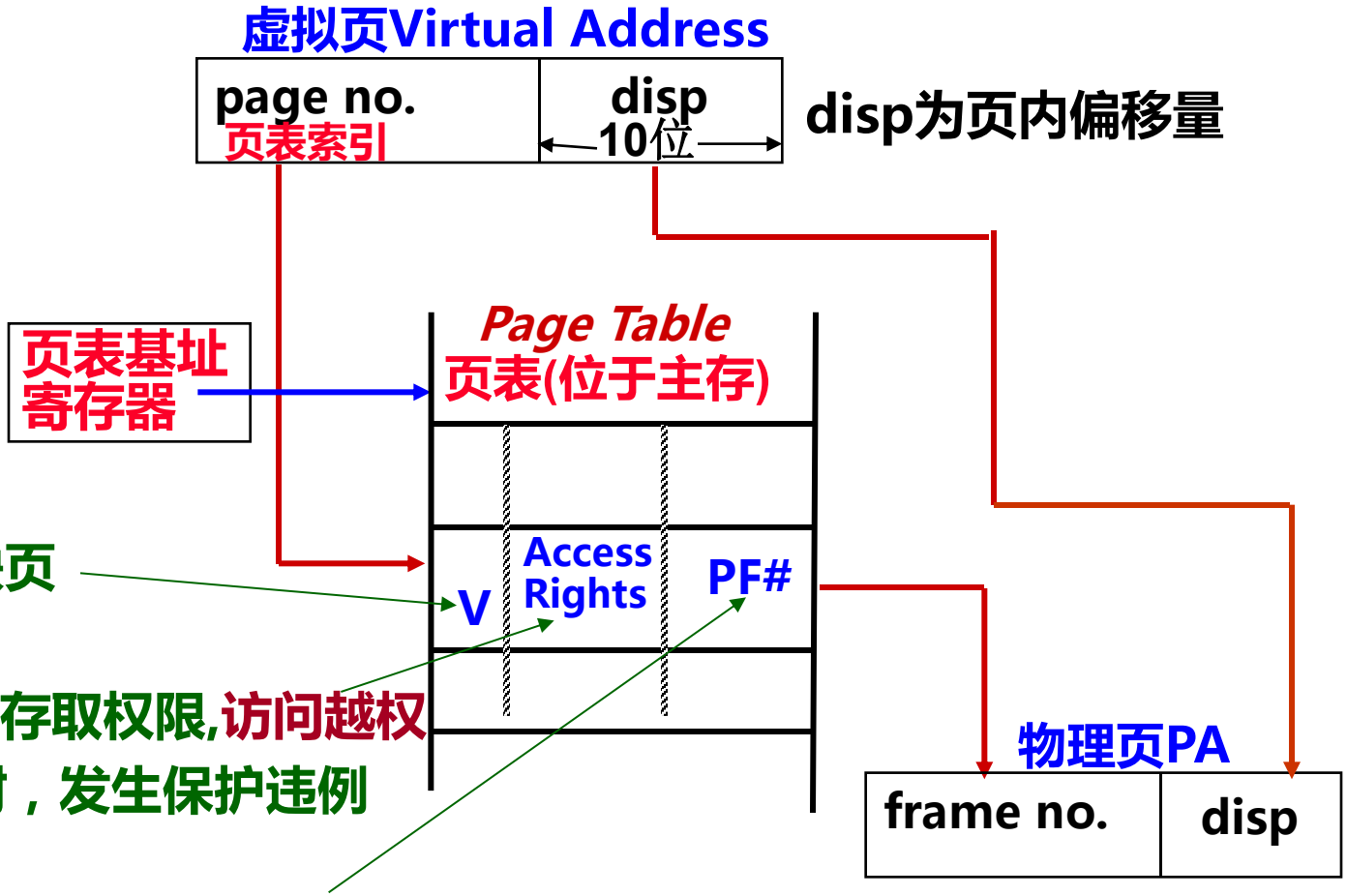
使用位（替换控制位）：页面最近被使用过？

存取权限位：可读可写？只读？只执行？

禁止缓存：说明是否可以装入cache

- **未分配页**：进程的虚拟地址空间中“空洞”对应的页（如VP0、VP4）
- **已分配的缓存页**：有内容对应的已装入主存的页（如VP1、VP2、VP5等）
- **已分配的未缓存页**：有内容对应但未装入主存的页（如VP3、VP6）

逻辑地址转换为物理地址的过程



问题：虚拟页与主存页框之间采用全相联方式进行映射，为何不像全相联Caceh那样（高位地址是Tag），而高位地址是索引呢？

信息访问中可能出现的异常情况

可能有两种异常情况：

1）缺页（ page fault）

产生条件：当Valid（有效位 / 装入位）为 0 时

相应处理：从磁盘读到内存，若内存没有空间，则还要从内存选择一页替换到磁盘上，替换算法类似于Cache，采用回写法，淘汰时，根据“dirty”位确定是否要写磁盘

当前指令执行被阻塞，当前进程被挂起，处理结束回到原指令继续执行

2）保护违例（ protection_violation_fault ）或访问违例

产生条件： 当存取权限(Access Rights)与所指定的具体操作不相符时

相应处理：在屏幕上显示“内存保护错”或“访问违例”信息

当前指令的执行被阻塞，当前进程被终止

Access Rights (存取权限)可能的取值有哪些？

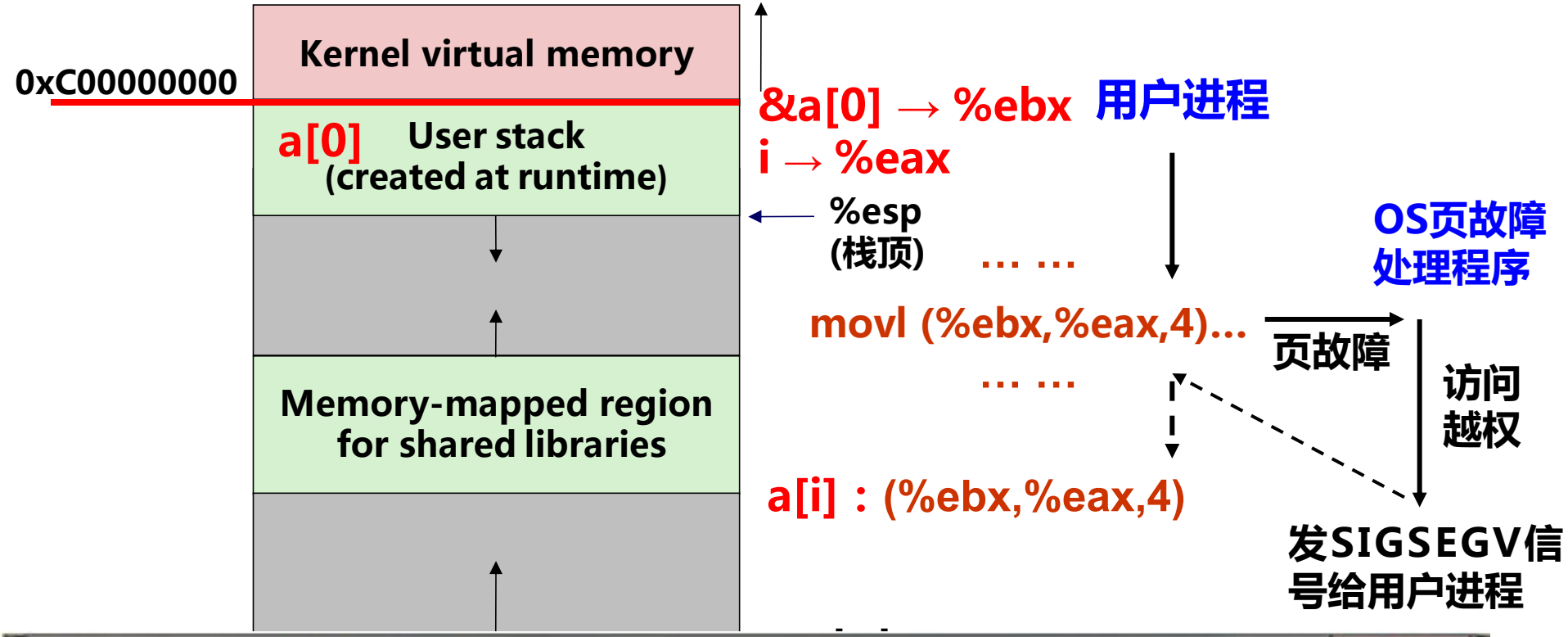
R = Read-only, R/W = read/write, X = execute only

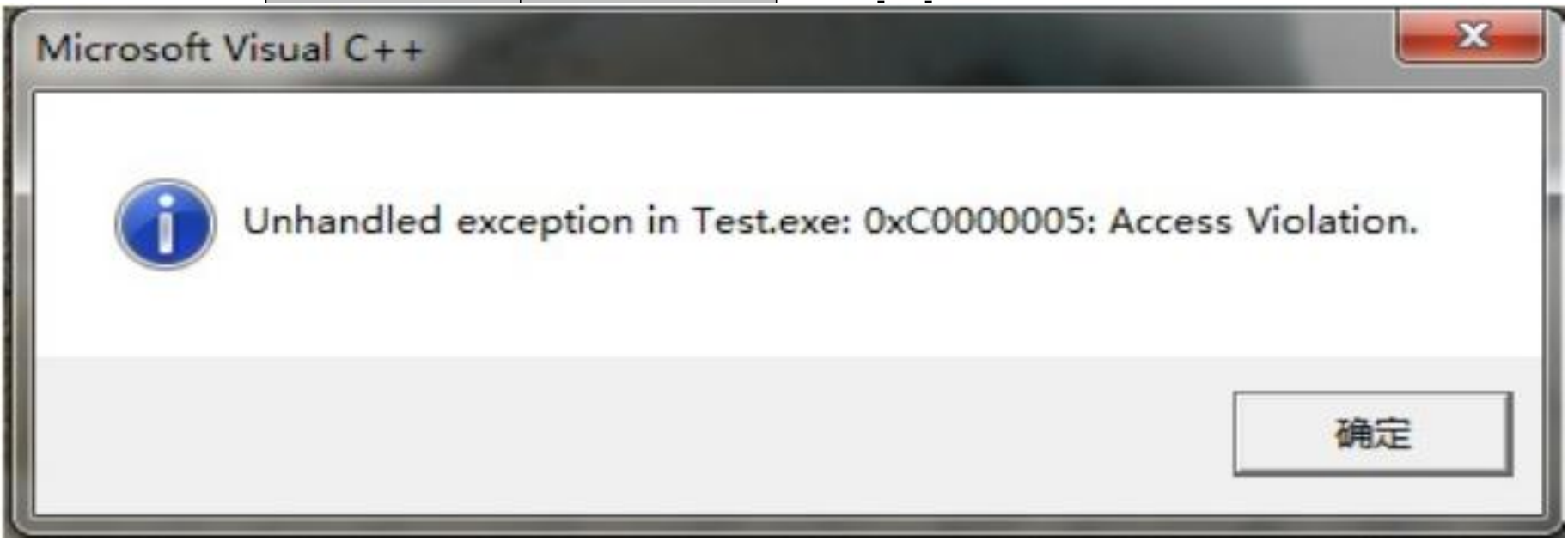
回顾：用“系统思维”分析问题

```
int sum(int a[ ], unsigned len)
{
  int   i , sum = 0;
  for   (i = 0; i <= len–1; i++)
        sum += a[i];
  return sum;
}
```

当参数len为0时，返回值应该是0，但是在机器上执行时，却发生访存异常。但当len为int型时则正常

Why?


0xC00000000
Kernel virtual memory
a[0]
User stack
(created at runtime)
Memory-mapped region
for shared libraries
&a[0] → %ebx
i → %eax
%esp
(栈顶)
用户进程
… …
movl (%ebx,%eax,4)…
… …
a[i] : (%ebx,%eax,4)
OS页故障
处理程序
页故障
访问
越权
发SIGSEGV信
号给用户进程


Microsoft Visual C++
Unhandled exception in Test.exe: 0xC0000005: Access Violation.
确定

TLBs --- Making Address Translation Fast

问题：一次存储器引用要访问几次主存？ 0 / 1 / 2 / 3次？

把经常要查的页表项放到Cache中，这种在Cache中的页表项组成的页表称为 *Translation Lookaside Buffer* or *TLB（快表）*

TLB中的页表项：tag+主存页表项

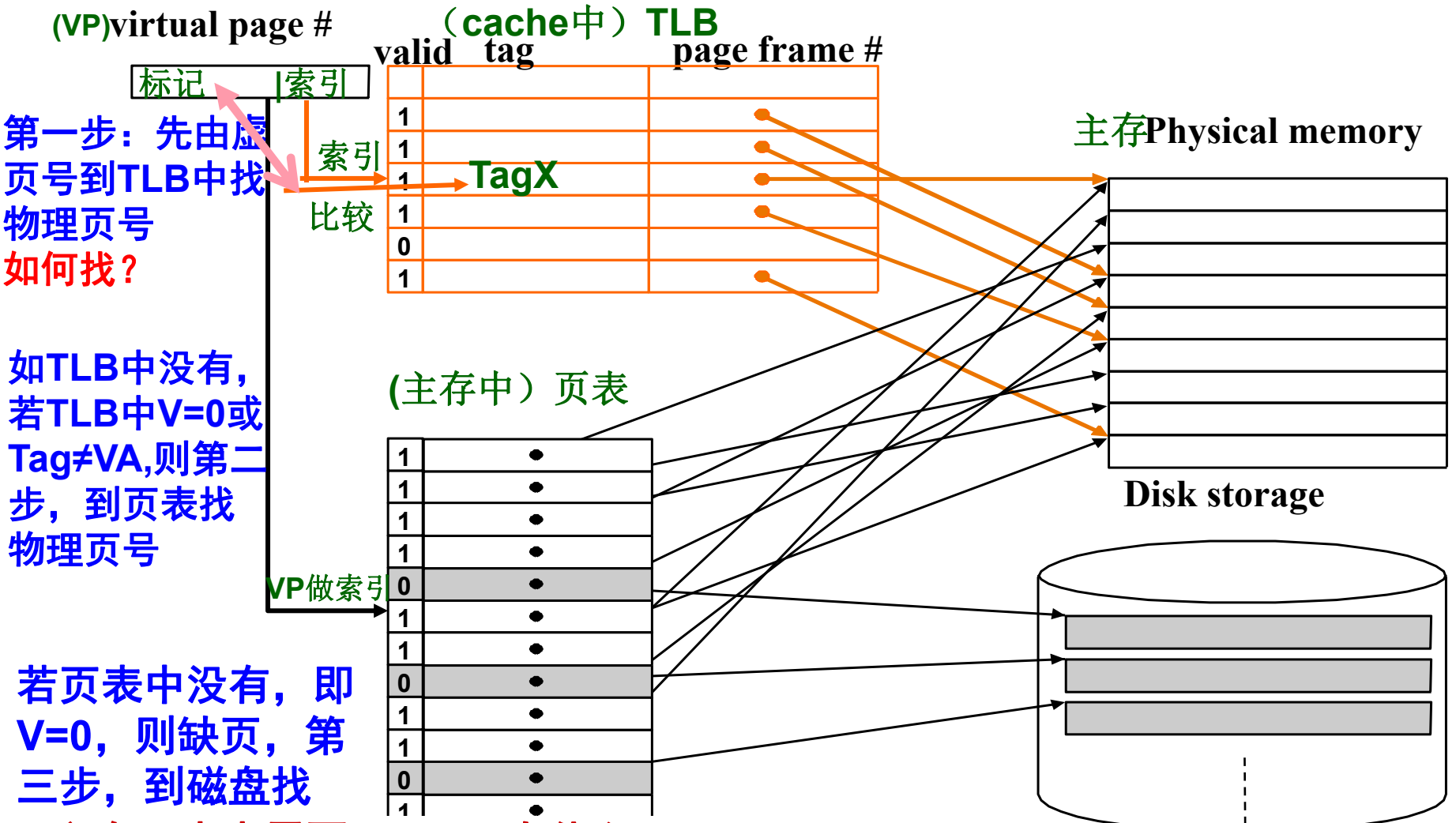
Virtual Address (tag)	Physical Address	Dirty	Ref	Valid	Access
	对应物理页框号				

CPU访存时，地址中虚页号被分成tag+index，tag用于和TLB页表项中的tag比较，index用于定位需要比较的表项

TLB全相联时，没有index，只有Tag，虚页号需与每个Tag比较；
TLB组相联时，则虚页号高位为Tag，低位为index，用作组索引。

TLBs(快表)Making Address Translation Fast

如果用组相连方式构建TLB

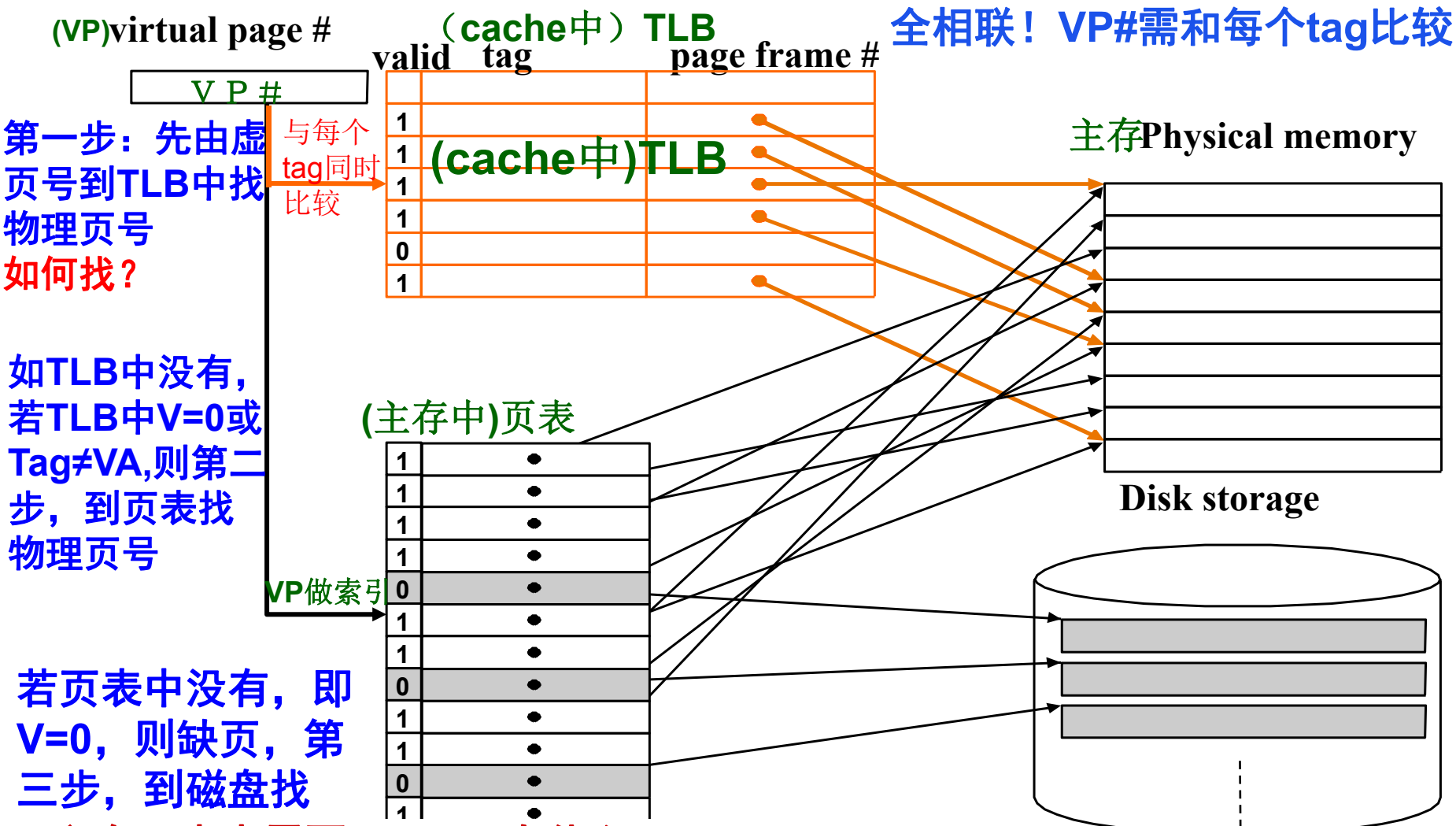


问题：引入TLB的目的是什么？减少到内存查页表的次数！

TLBs(快表)Making Address Translation Fast

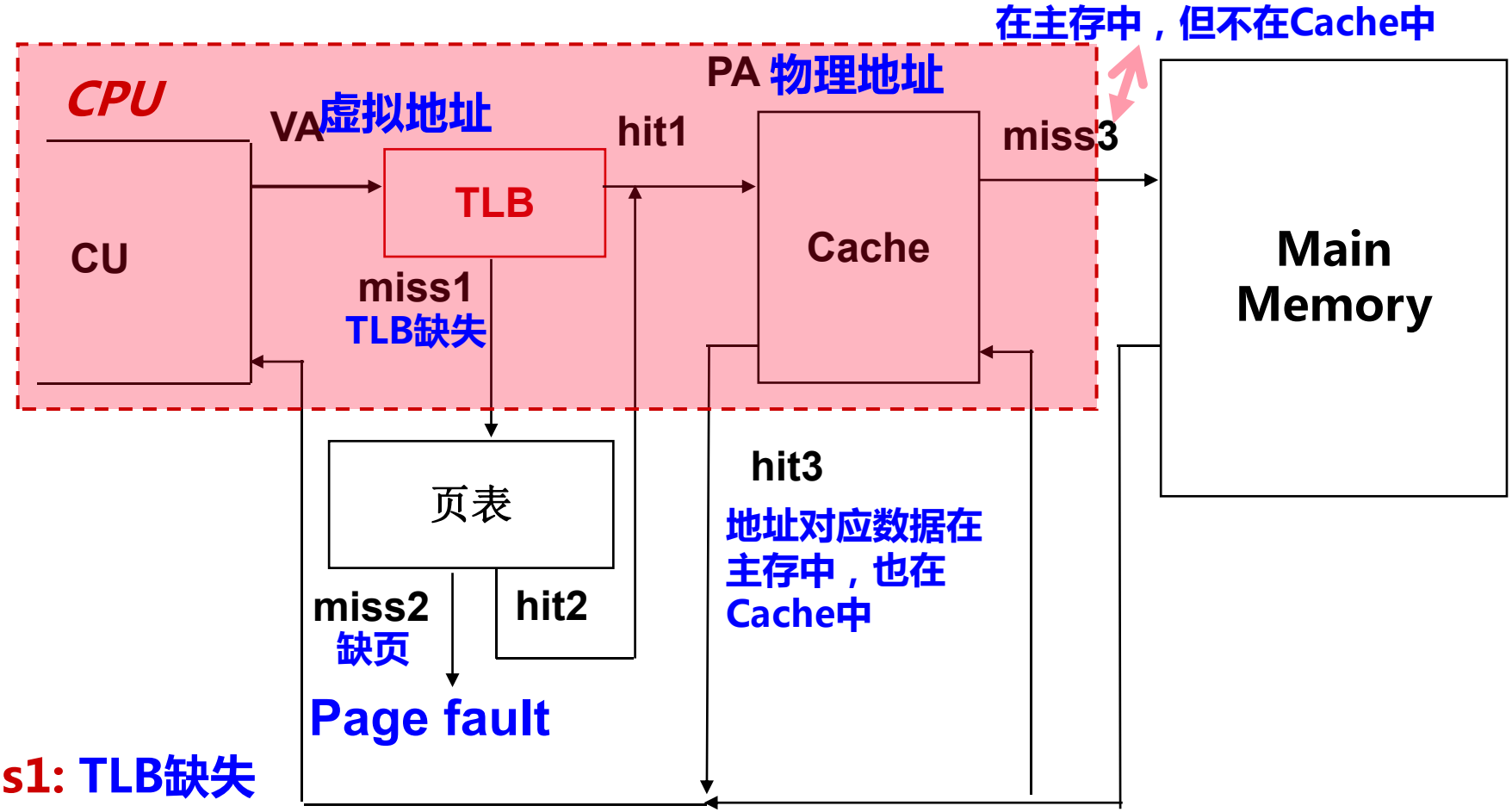
如果用全相联方式构建TLB

这里的TLB采用何映射方式？
全相联！ VP#需和每个tag比较



问题：引入TLB的目的是什么？减少到内存查页表的次数！

Translation Look-Aside Buffers(*TLB,快表*)



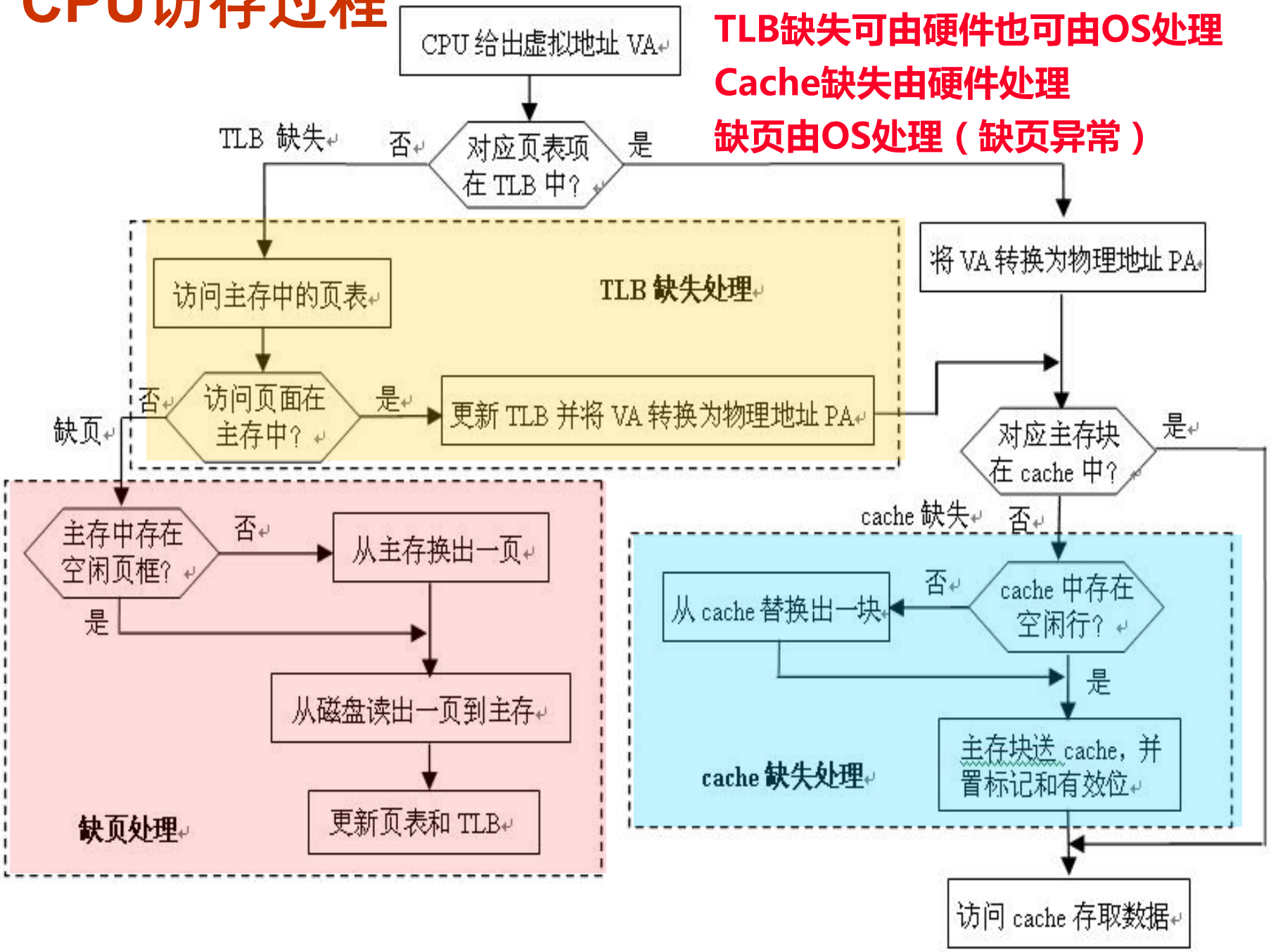
Miss1: TLB缺失

Miss2: 缺页

Miss3: PA 在主存中，但不在Cache中

TLB冲刷指令和Cache冲刷指令都是操作系统使用的特权指令

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

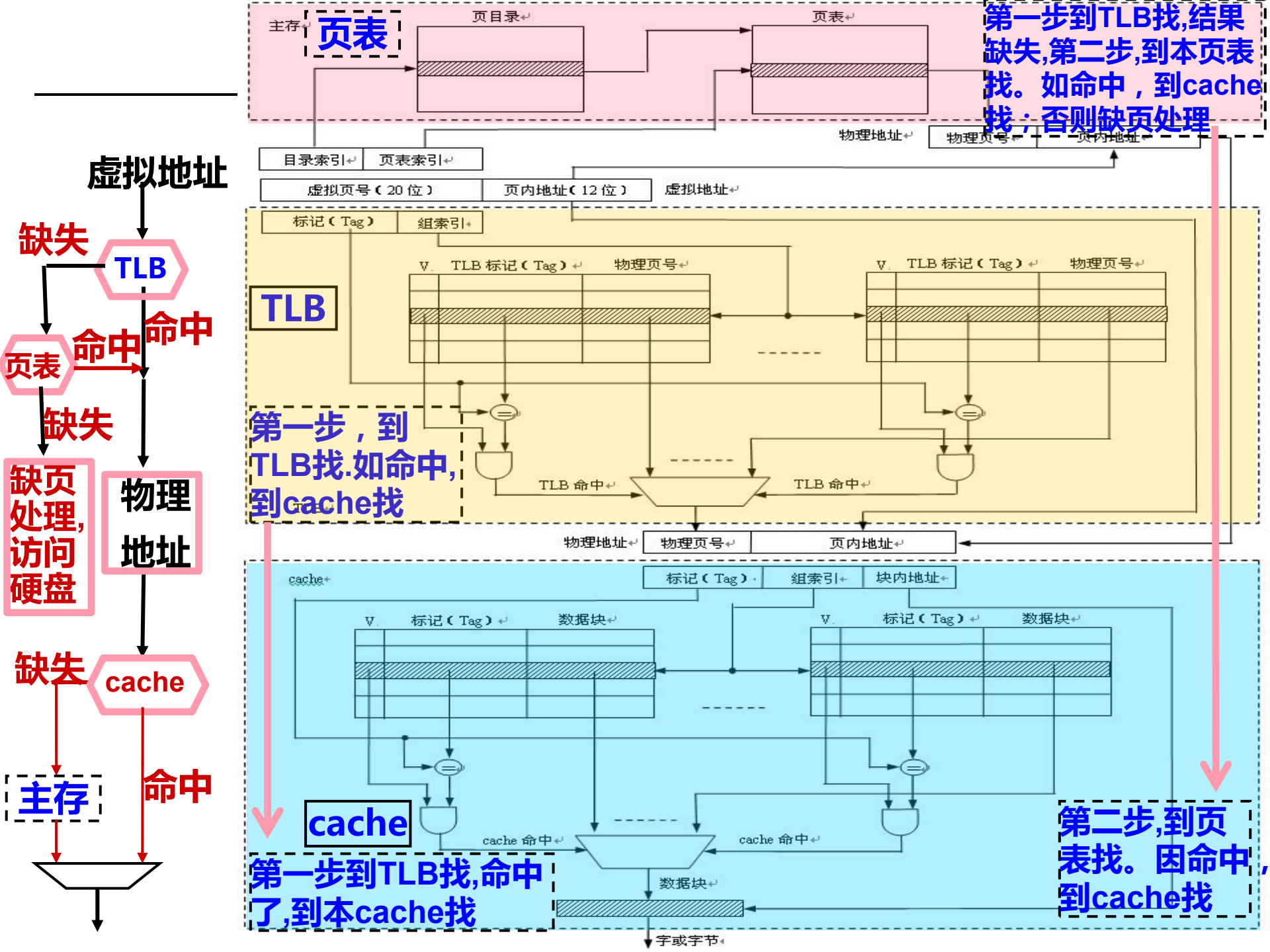
TLB	Page table	Cache	Possible? If so, under what circumstance?
hit	hit	miss	可能，TLB命中则页表一定命中，但实际上不会查页表
miss	hit	hit	可能，TLB缺失但页表命中，信息在主存，就可能在Cache
miss	hit	miss	可能，TLB缺失但页表命中，信息在主存，但可能不在Cache
miss	miss	miss	可能，TLB缺失页表缺失，信息不在主存，一定也不在Cache
hit	miss	miss	不可能，页表缺失，信息不在主存，TLB中一定没有该页表项
hit	miss	hit	同上
miss	miss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

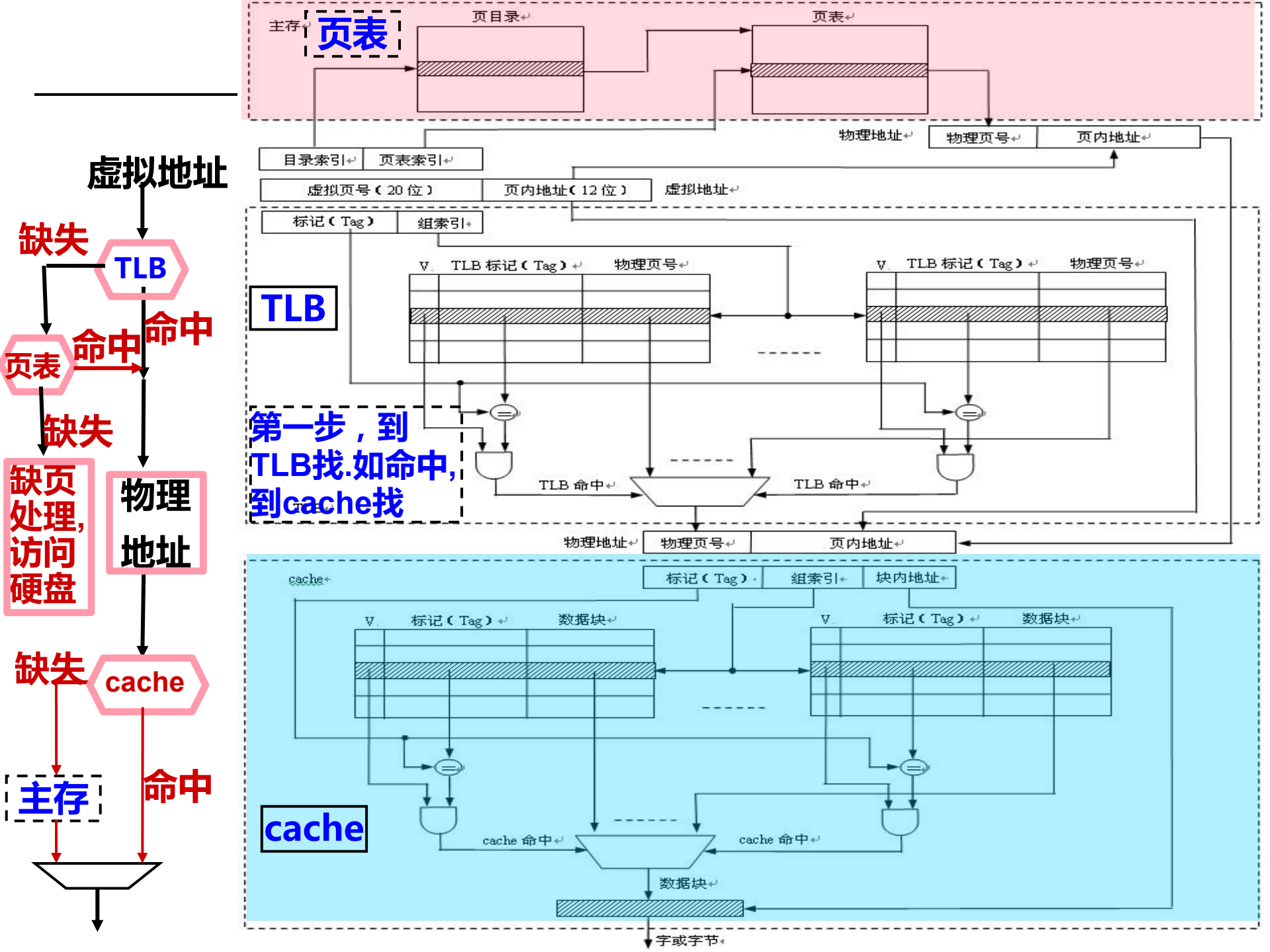
最好的情况是hit、hit、hit，此时，访问主存几次？ 不需要访问主存！

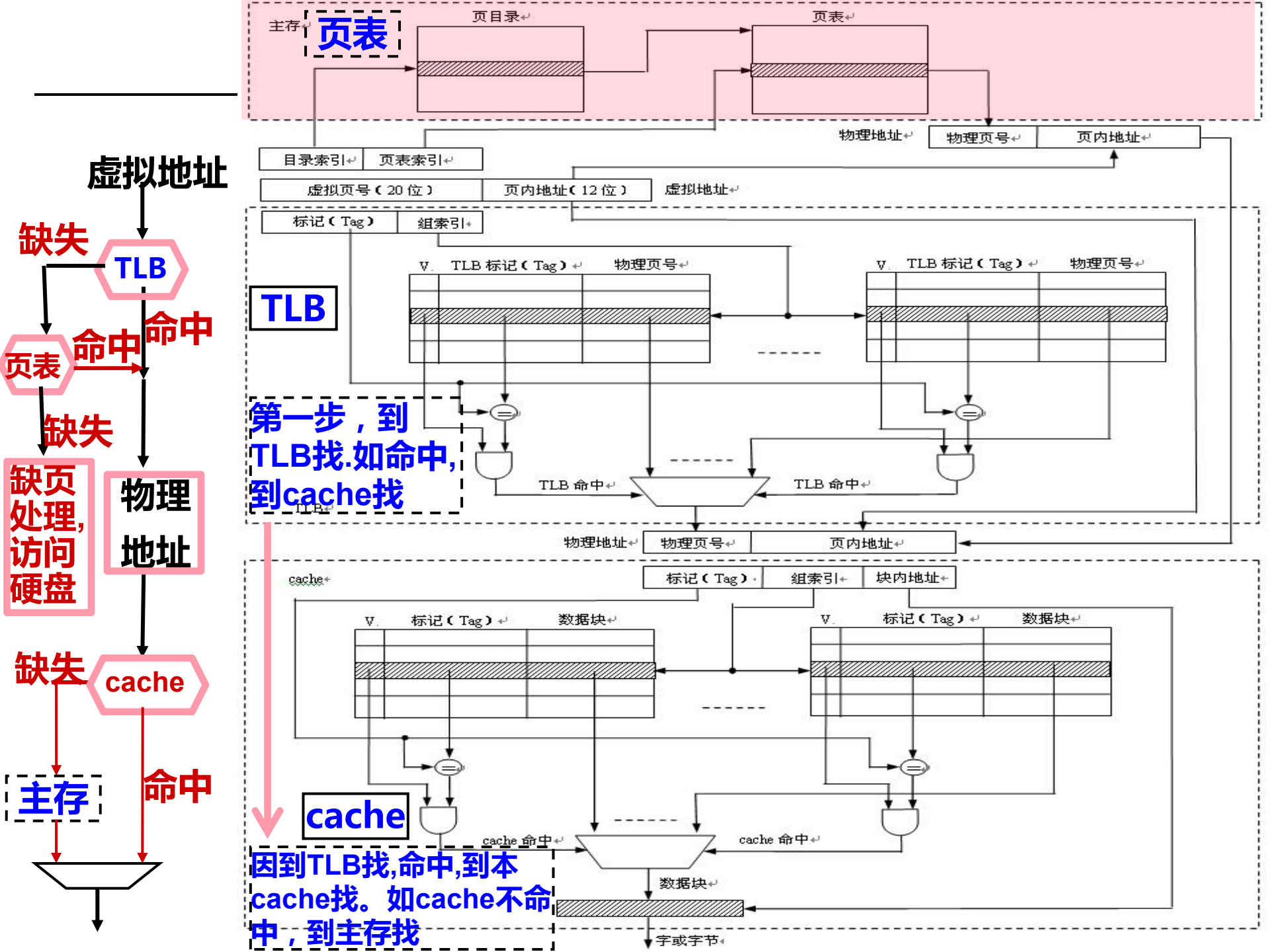
以上组合中，最好的情况是？ hit、hit、miss和miss、hit、hit 访存1次

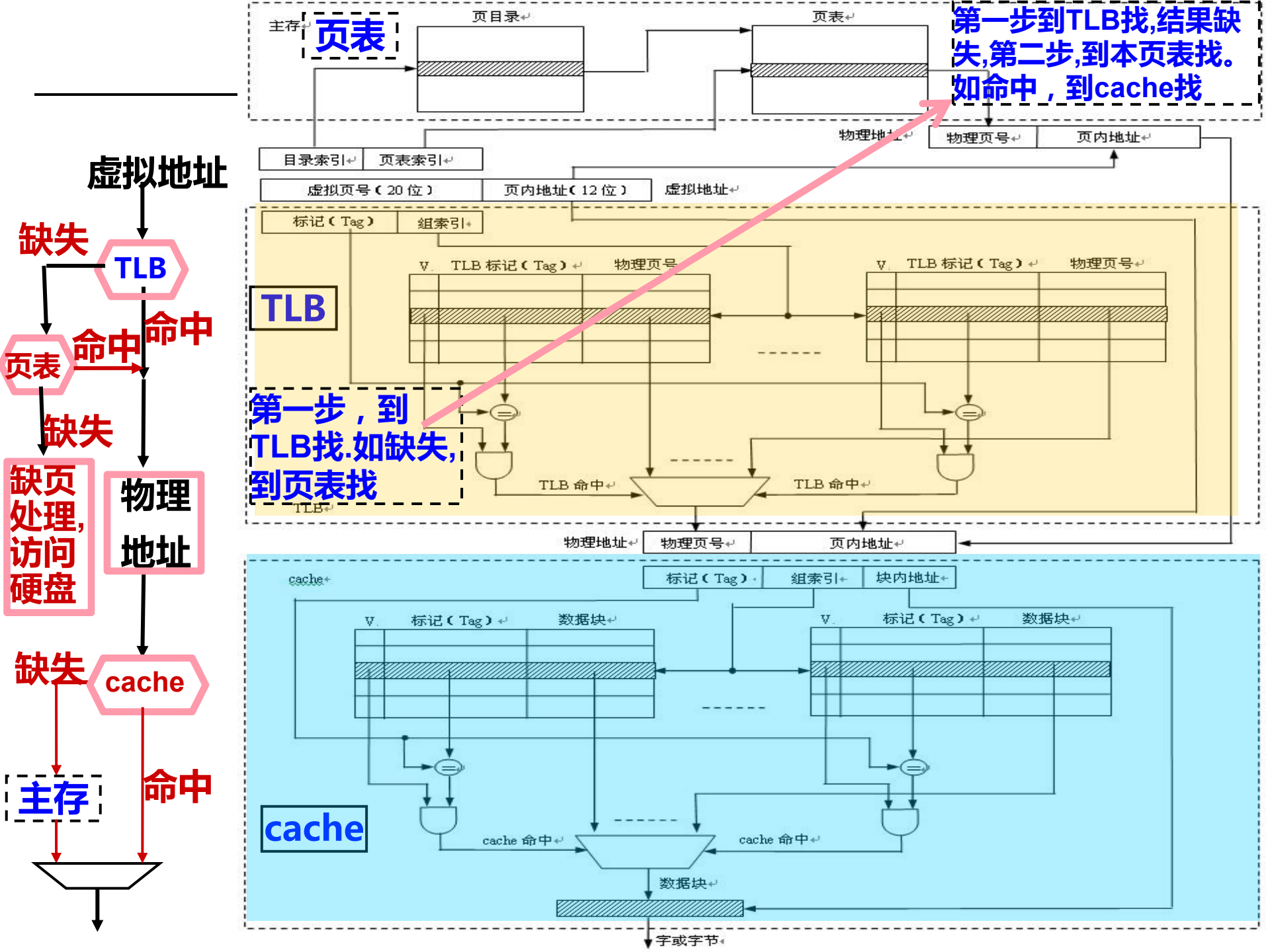
以上组合中，最坏的情况是？ miss、miss、miss 需访问磁盘、并访存至少2次

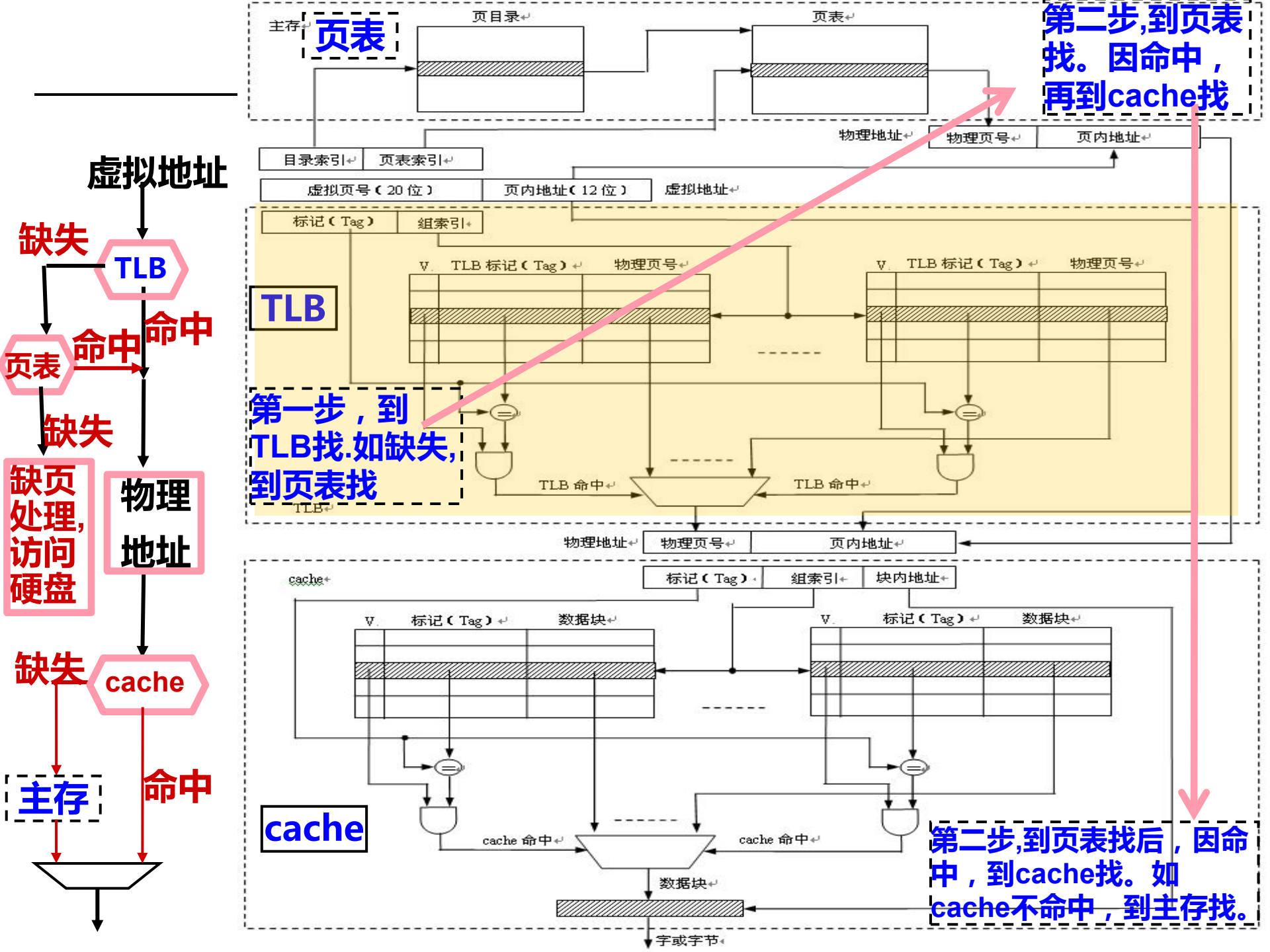
介于最坏和最好之间的是？ miss、hit、miss 不需访问磁盘、但访存至少2次


虚拟地址
缺失
TLB
命中
页表
命中
缺失
缺页处理,访问硬盘
物理地址
缺失
cache
主存
命中
主存
页表
页目录
页表
第一步到TLB找,结果缺失,第二步,到本页表找。如命中，到cache找；否则缺页处理
物理地址
物理页号
页内地址
目录索引
页表索引
虚拟页号（20 位）
页内地址（12 位）
虚拟地址
标记（Tag）
组索引
TLB
V
TLB 标记（Tag）
物理页号
TLB 命中
第一步，到TLB找.如命中,到cache找
物理地址
物理页号
页内地址
cache
标记（Tag）
组索引
块内地址
V
标记（Tag）
数据块
cache 命中
cache
第一步到TLB找,命中了,到本cache找
数据块
字或字节
第二步,到页表找。因命中,到cache找


虚拟地址
缺失
TLB
命中
页表
命中
缺失
缺页处理,访问硬盘
物理地址
缺失
cache
主存
命中
主存
页表
页目录
页表
物理地址
物理页号
页内地址
目录索引
页表索引
虚拟页号（20 位）
页内地址（12 位）
虚拟地址
标记（Tag）
组索引
TLB
V
TLB 标记（Tag）
物理页号
第一步，到TLB找.如命中,到cache找
TLB 命中
TLB 命中
物理地址
物理页号
页内地址
cache
标记（Tag）
组索引
块内地址
V
标记（Tag）
数据块
cache 命中
cache 命中
数据块
字或字节


虚拟地址
缺失
TLB
命中
页表
命中
缺失
缺页处理,访问硬盘
物理地址
缺失
cache
主存
命中
主存
页表
页目录
页表
物理地址
物理页号
页内地址
目录索引
页表索引
虚拟页号（20 位）
页内地址（12 位）
虚拟地址
标记（Tag）
组索引
TLB
V
TLB 标记（Tag）
物理页号
第一步，到TLB找.如命中,到cache找
TLB 命中
TLB 命中
物理地址
物理页号
页内地址
cache
标记（Tag）
组索引
块内地址
V
标记（Tag）
数据块
cache
因到TLB找,命中,到本cache找。如cache不命中，到主存找
cache 命中
cache 命中
数据块
字或字节


虚拟地址
缺失
TLB
命中
页表
命中
缺失
缺页处理,访问硬盘
物理地址
缺失
cache
主存
命中
主存
页表
页目录
页表
第一步到TLB找,结果缺失,第二步,到本页表找。如命中，到cache找
物理地址
物理页号
页内地址
目录索引
页表索引
虚拟页号（20 位）
页内地址（12 位）
虚拟地址
标记（Tag）
组索引
TLB
V
TLB 标记（Tag）
物理页号
第一步，到TLB找.如缺失,到页表找
TLB 命中
TLB 命中
物理地址
物理页号
页内地址
cache
标记（Tag）
组索引
块内地址
V
标记（Tag）
数据块
cache 命中
cache 命中
数据块
字或字节


虚拟地址
缺失
TLB
命中
页表
命中
缺失
缺页处理，访问硬盘
物理地址
缺失
cache
主存
命中
主存
页表
页目录
页表
第二步,到页表找。因命中，再到cache找
物理地址
物理页号
页内地址
目录索引
页表索引
虚拟页号（20 位）
页内地址（12 位）
虚拟地址
标记（Tag）
组索引
TLB
V
TLB 标记（Tag）
物理页号
第一步，到TLB找.如缺失，到页表找
TLB 命中
TLB 命中
TLB
物理地址
物理页号
页内地址
cache
标记（Tag）
组索引
块内地址
V
标记（Tag）
数据块
cache
cache 命中
cache 命中
数据块
字或字节
第二步,到页表找后，因命中，到cache找。如cache不命中，到主存找。

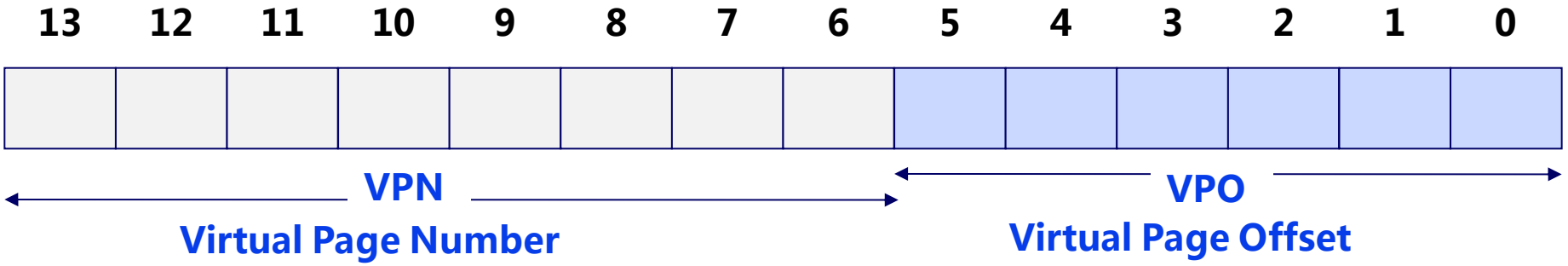
缩写的含义

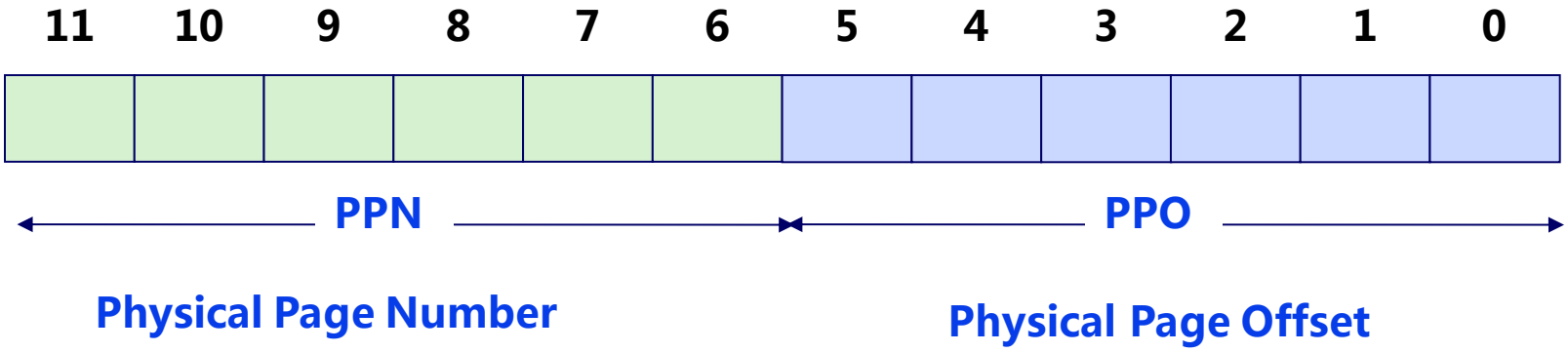
- 基本参数（按字节编址）
  - $N = 2^n$：虚拟地址空间大小
  - $M = 2^m$：物理地址空间大小
  - $P = 2^p$：页大小
- 虚拟地址 (VA)中的各字段
  - TLBI: TLB index（TLB索引）
  - TLBT: TLB tag（TLB标记）
  - VPO: Virtual page offset （页内偏移地址）
  - VPN: Virtual page number （虚拟页号）
- 物理地址(PA)中的各字段
  - PPO: Physical page offset (页内偏移地址)
  - PPN: Physical page number（物理页号）
  - CO: Byte offset within cache line（块内偏移地址）
  - CI: Cache index（cache索引）
  - CT: Cache tag（cache标记）

一个简化的存储系统举例

- 假定以下参数，则虚拟地址和物理地址如何划分？共多少页表项？
  - 14-bit virtual addresses（虚拟地址14位）
  - 12-bit physical address（物理地址12位）
  - Page size = 64 bytes（页大小64B）

页表项数应为：
$2^{14-6}=256$





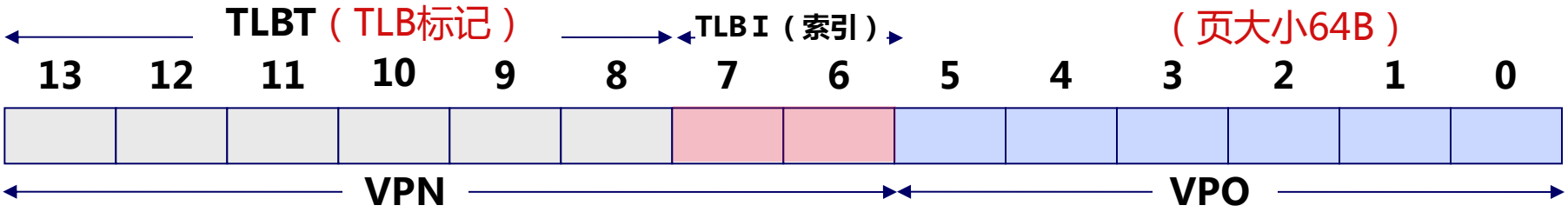
一个简化的存储系统举例（续）

假定部分页表项内容（十六进制表示）如右：

VPN	PPN	Valid
000	28	1
001	–	0
002	33	1
003	02	1
004	–	0
005	16	1
006	–	0
007	–	0

VPN	PPN	Valid
028	13	1
029	17	1
02A	09	1
02B	–	0
02C	–	0
02D	2D	1
02E	11	1
02F	0D	1

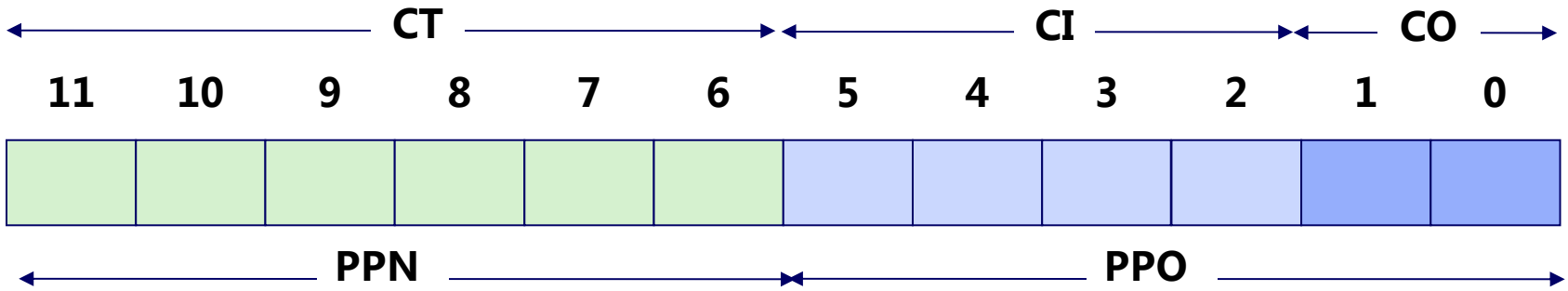
假定TLB如下：16 个TLB项，4路组相联，则TLBT和TLBI各占几位？



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

一个简化的存储系统举例（续）

假定Cache的参数和内容（十六进制）如下：16 行，主存块大小为4B，直接映射，则主存地址如何划分？

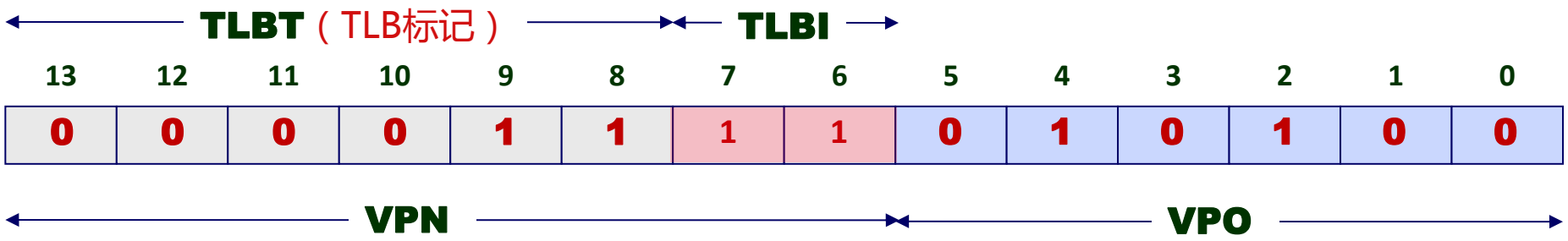


Idx	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	V	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

一个简化的存储系统举例（续）

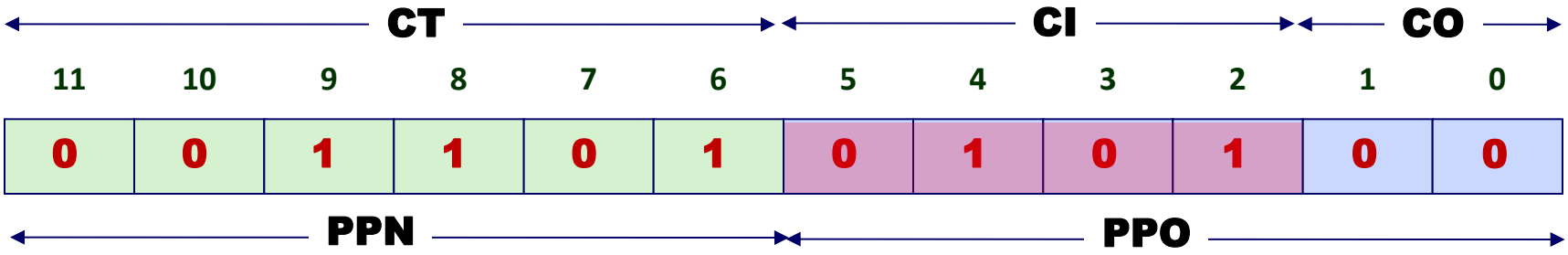
假设该存储系统所在计算机采用小端方式， CPU执行某指令过程中要求访问一个16位数据，给出的逻辑地址为0x03D4，说明访存过程。



VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

物理地址为

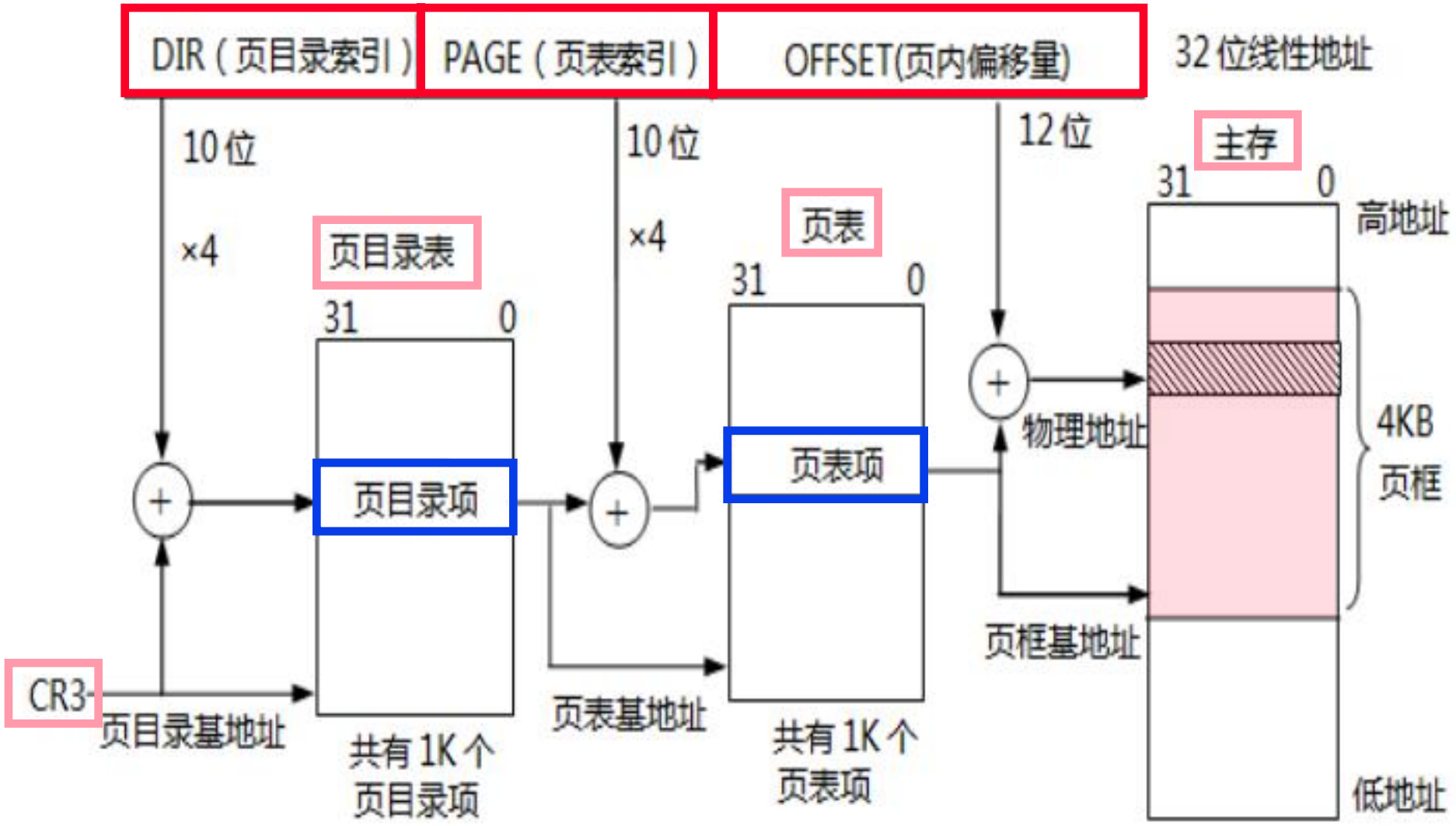
问题：逻辑地址为0x0A7A、0x0507时的访存过程如何？
TLB缺失/cache缺失、TLB缺失/缺页



CO 0 CI 0x5 CT 0x0D cache Hit? Y 数据: 0x7236

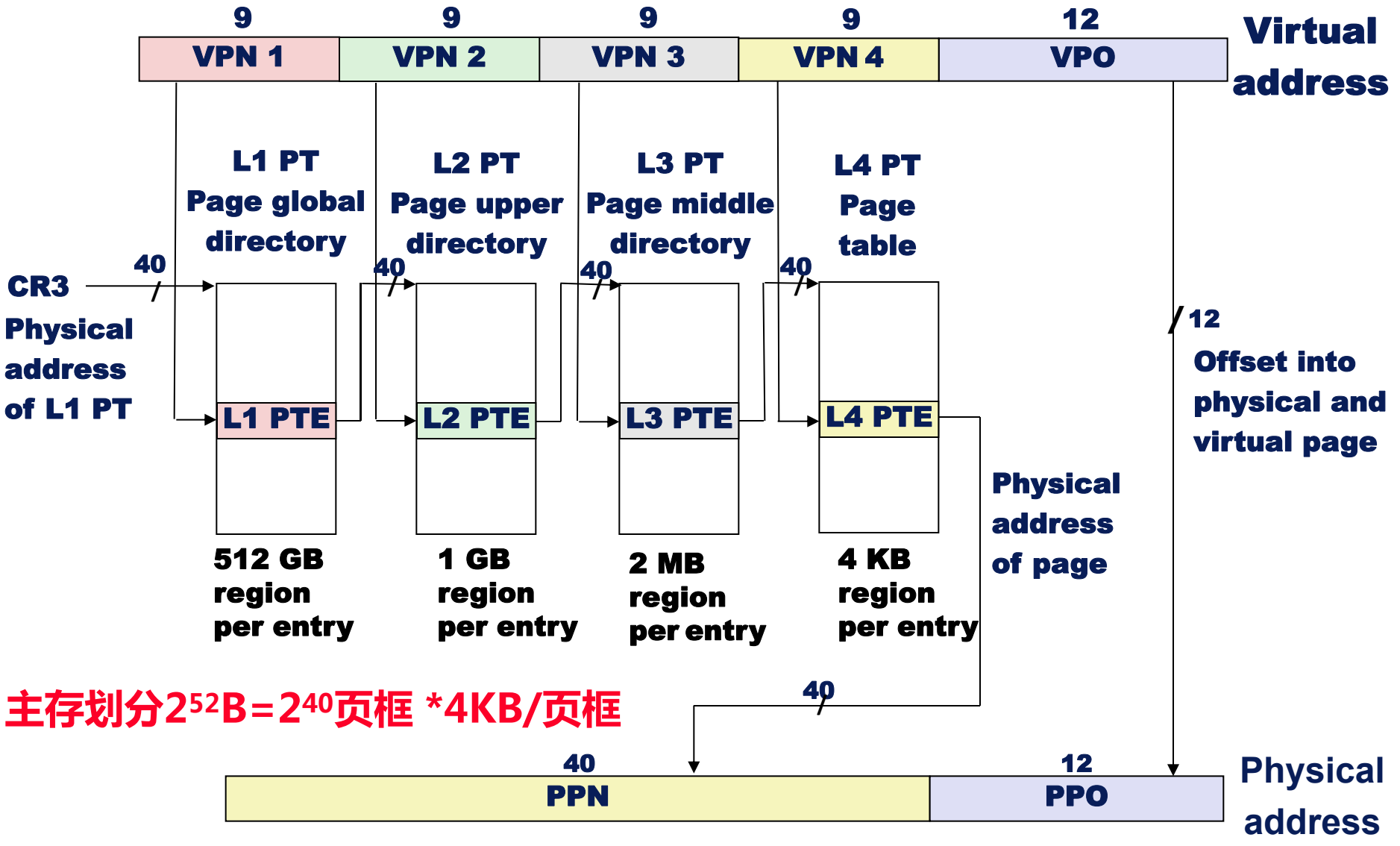
IA-31中线性地址向物理地址转换

线性地址空间划分：4GB=1K个子空间 * 1K个页面/子空间 * 4KB/页



Core i7 Page Table Translation

线性地址空间划分：2^{48}B=512* 512 *512 *512 * 4KB/页



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

内存访问时的异常信息


_360DesktopSwitcherControlWnd_MsgOnly_Wnd: 360DesktopSwitch64.e...
0x772440e8 指令引用的 0x191c17d8 内存。该内存不能为 read。
要终止程序，请单击“确定”。
确定


Microsoft Visual C++
Unhandled exception in Test.exe: 0xC0000005: Access Violation.
0xc0000005为出错ID
确定

存储保护的基本概念

- 什么是存储保护？
  - 为避免多道程序相互干扰，防止某程序出错而破坏其他程序的正确性或不合法地访问其他程序或数据区，应对每个程序进行存储保护
- 操作系统程序和用户程序都需要保护
- 以下情况发生存储保护错
  - 地址越界（转换得到的物理地址不属于可访问范围）
  - 访问越权（访问操作与所拥有的访问权限不符）
    - 页表中设定访问（存取）权限
- 访问属性的设定
  - 数据段可指定R/W或RO；程序段可指定R/E或RO
- 最基本的保护措施：

  规定各道程序只能访问属于自己所在的存储区和共享区
  - 对于属自己存储区的信息：可读可写，只读/只可执行
  - 对共享区或已获授权的其他用户信息：可读不可写
  - 对未获授权的信息（如OS内核、页表等）：不可访问

存储保护的硬件支持

- 为了对操作系统的存储保护提供支持，硬件必须具有以下三种基本功能：
  - 支持至少两种运行模式：
    - 管理模式(Supervisor Mode)

      执行系统程序（内核）时处理器所处的模式称为管理模式(Supervisor Mode)，或称管理程序状态，简称管态、管理态、核心态、内核态
    - 用户模式(User Mode)

      CPU执行非操作系统的用户程序时，处理器所处的模式就是用户模式，或称用户状态、目标程序状态，简称为目态或用户态
  - 使一部分CPU状态只能由内核程序读写而不能由用户程序读写：这部分状态包括：User/Supervisor模式位、页表首地址、TLB等。OS内核可以用特殊的指令（一般称为管态指令或特权指令）来读写这些状态
  - 提供让CPU在管理模式（内核态）和用户模式（用户态）相互转换的机制：“异常”和“陷阱”（系统调用）使CPU从用户态转到内核态；异常处理中的“返回”指令使CPU从内核态态转到用户态
- 通过上述三个功能并把页表保存在OS的地址空间，OS就可以更新页表，并防止用户程序改变页表，确保用户程序只能访问由OS分配给的存储空间

第四讲小结

- 虚拟存储器是磁盘和主存之间的缓存管理机制，而不是一种物理存储器
- 引入虚拟存储器，使程序员可以在一个极大的存储空间写程序，无需知道运行程序的物理存储器有多大
- 虚拟存储器采用“按需调页”技术，一部分程序调入主存，一部分存放在磁盘上
- 交换的块（称为页）比Cache-MM层次的块要大得多
- 采用全相联映射，通过页表实现逻辑地址和物理地址转换，由硬件实现
- 缺页处理由OS完成（cache miss处理由硬件实现）
- 采用Write Back写策略
- 页表中记录装入位、访问方式、使用情况、修改位、磁盘地址或页框号
- 经常使用的页表项放到特殊的Cache中，称为快表（TLB）
- 有分页式、分段式、段页式三种管理模式
- 两类存储保护形式
  - 可利用程序重定位或其他存储保护方式进行地址越界判断
  - 可利用访问方式进行存取权限的判断

第7章作业

2（4），2（5），2（9），2（10）

6， 17，23，

24（其中表中数据都是16进制）