

第6章 层次结构存储系统

存储器概述

主存与CPU的连接及其读写操作

磁盘存储器

高速缓冲存储器(cache)

虚拟存储器

IA-32/Linux中的地址转换

层次结构存储系统

- **主要教学目标**

- 理解CPU执行指令过程中为何要访存
- 理解访存操作的大致过程及涉及到的部件
- 了解层次化存储器系统的由来及构成
- 了解CPU与主存储器之间的连接及读写操作
- 掌握Cache机制并理解其对程序性能的影响
- 理解程序局部性的重要性并能开发局部性好的程序
- 了解虚拟存储管理的基本概念和实现原理
- 理解访存操作完整过程以及所涉及到的部件之间的关联
 地址转换（查TLB、查页表）、访问Cache、访问主存、读写磁盘
- 理解访存过程中硬件和操作系统之间的协调关系

层次结构存储系统

◦ 分以下六个部分介绍

- 第一讲：存储器概述
- 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：磁盘存储器
- 第四讲：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
- 第五讲：虚拟存储器
 - 虚拟地址空间、虚拟存储器的实现
- 第六讲：IA-32/Linux中的地址转换
 - 逻辑地址到线性地址的转换
 - 线性地址到物理地址的转换

回顾：程序及指令的执行过程

- ° 在内存存放的指令实际上是机器代码（0/1序列）

08048394 <add>:

1	8048394:	55
2	8048395:	89 e5
3	8048397:	8b 45 0c
4	804839a:	03 45 08
5	804839d:	5d
6	804839e:	c3

push	%ebp
mov	%esp, %ebp
mov	0xc(%ebp), %eax
add	0x8(%ebp), %eax
pop	%ebp
ret	

栈是主存中的一个区域！

- ° 对于add函数的执行，何时需要访存？

- ✓ 每条指令都需从主存单元取到CPU执行 取指
- ✓ PUSH指令需把寄存器内容压入栈中 存数 POP指令则相反 取数
- ✓ 第3条mov指令需要从主存中取数后送到寄存器 取数
- ✓ 第4条add指令需要从主存取操作数到ALU中进行运算 取数
- ✓ ret指令需要从栈中取出返回地址，以能正确回到调用程序执行 取数

访存是指令执行过程中一个非常重要的环节！ 取指、取数、存数

基本术语

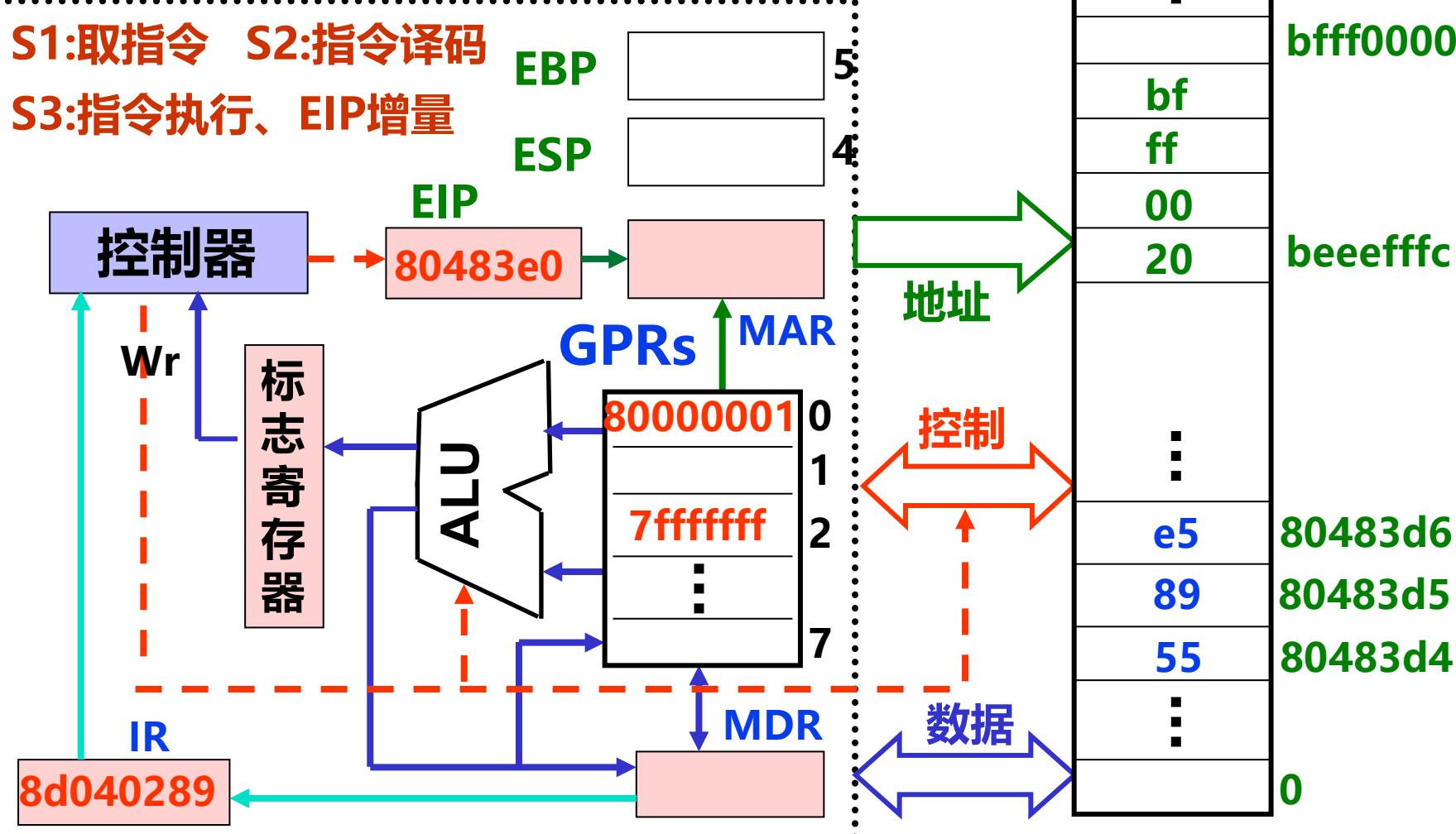
- 记忆单元（存储基元 / 存储元 / 位元）(Cell)
 - 具有两种稳态的能够表示二进制数码0和1的物理器件
- 存储单元 / 编址单位 (Addressing Unit)
 - 具有相同地址的位构成一个存储单元，也称为一个编址单位
- 存储体/ 存储矩阵 / 存储阵列 (Bank)
 - 所有存储单元构成一个存储阵列
- 编址方式 (Addressing Mode)
 - 字节编址、按字编址
- 存储器地址寄存器 (Memory Address Register - MAR)
 - 用于存放主存单元地址的寄存器
- 存储器数据寄存器 (Memory Data Register-MDR (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

回顾：冯.诺依曼计算机模型

80483da: 8b 45 0c mov 0xc(%ebp), %eax
80483dd: 8b 55 08 mov 0x8(%ebp), %edx
→ 80483e0: 8d 04 02 lea (%edx,%eax,1), %eax

S1:取指令 S2:指令译码

S3:指令执行、EIP增量



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、U盘、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）

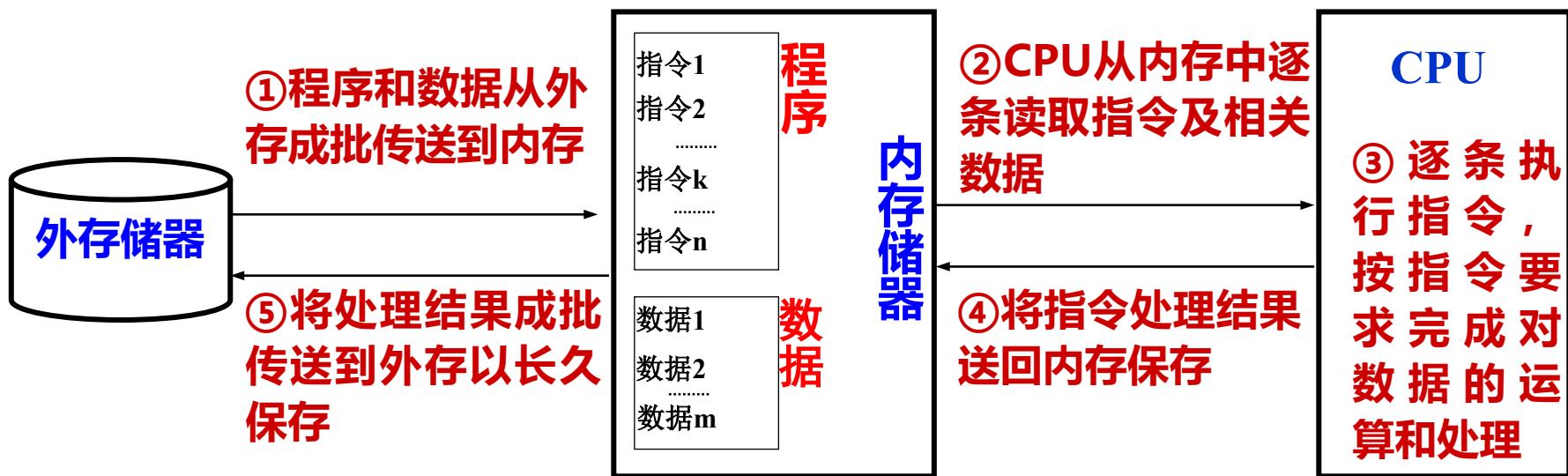
- 内存储器MM (主存储器Main (Primary) Memory)

- 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)

- 位于主机之外，用来存放暂不运行的程序、数据或存档文件
- 用磁盘、SSD等实现，容量大而速度慢

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

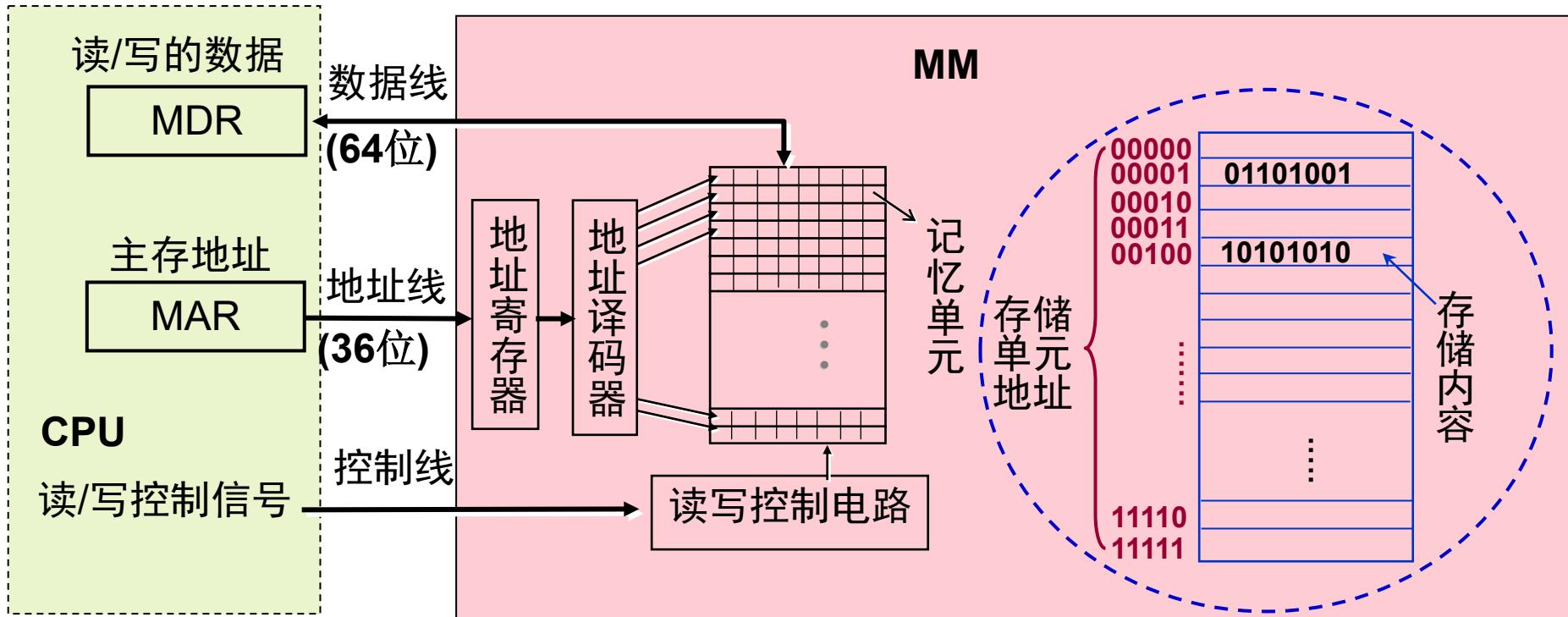
问题：主存中存放的是什么信息？CPU何时会访问主存？

指令及其数据！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？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）。

可寻址范围为 $0 \sim 2^{36}-1$ ，即主存地址空间为64GB（按字节编址时）。



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。

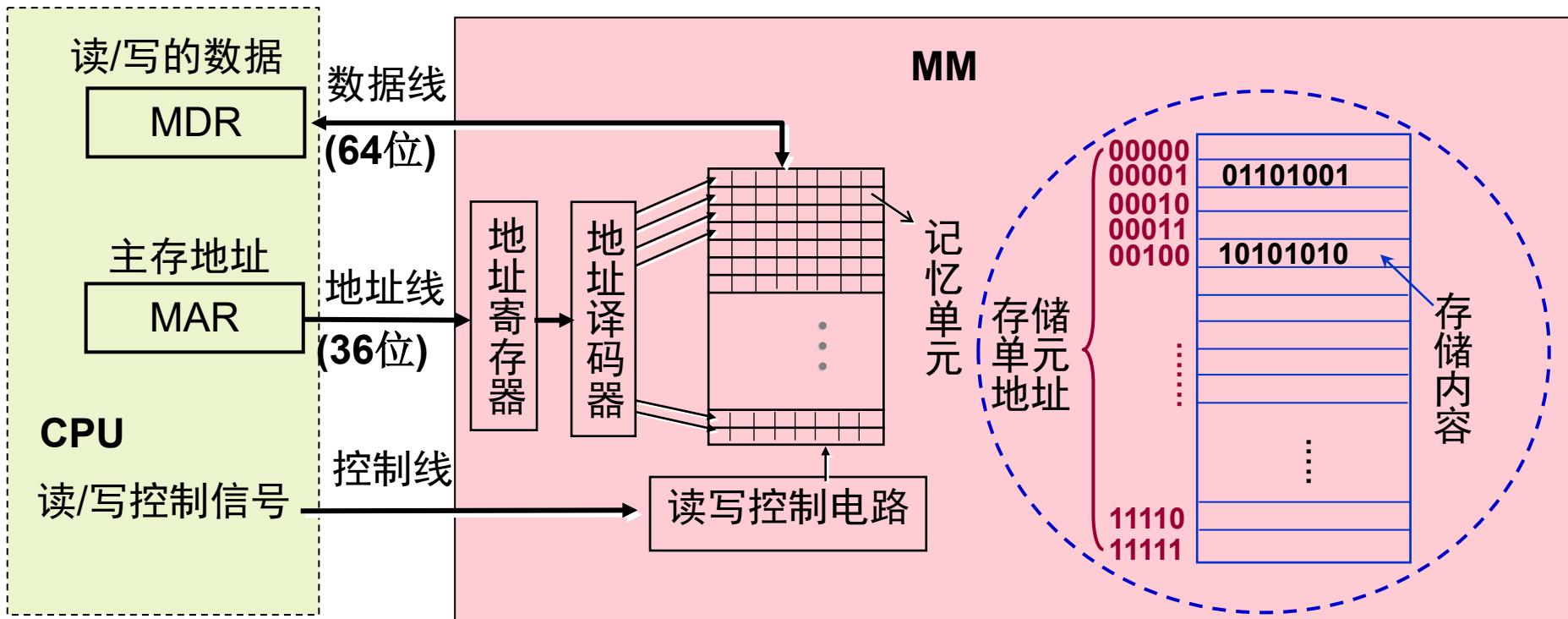
主存的主要性能指标

- ° 性能指标：

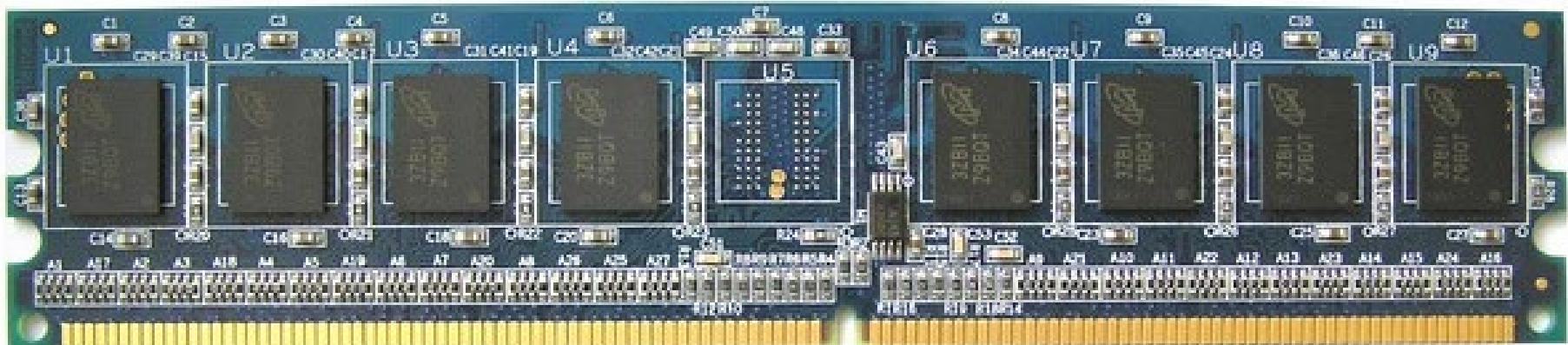
按字节连续编址，每个存储单元为1个字节（8个二进位）

- 存储容量：所包含的存储单元的总数（单位：MB或GB）
- 存取时间 T_A ：从CPU送出内存单元的地址码开始，到主存读出数据并送到CPU（或者是把CPU数据写入主存）所需要的时间（单位： ns , $1 \text{ ns} = 10^{-9} \text{ s}$ ），分读取时间和写入时间
- 存储周期 T_{MC} ：连读两次访问存储器所需的最短时间间隔，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。）
(就像一趟货车运货时间和发车周期是两个不同概念一样。)

主存的结构



主存地址空间大小不等于主存容量（实际安装的主存大小）！



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}

为避免混淆，Patterson和Hennessy将2的幂次和10的幂次进行了区分。

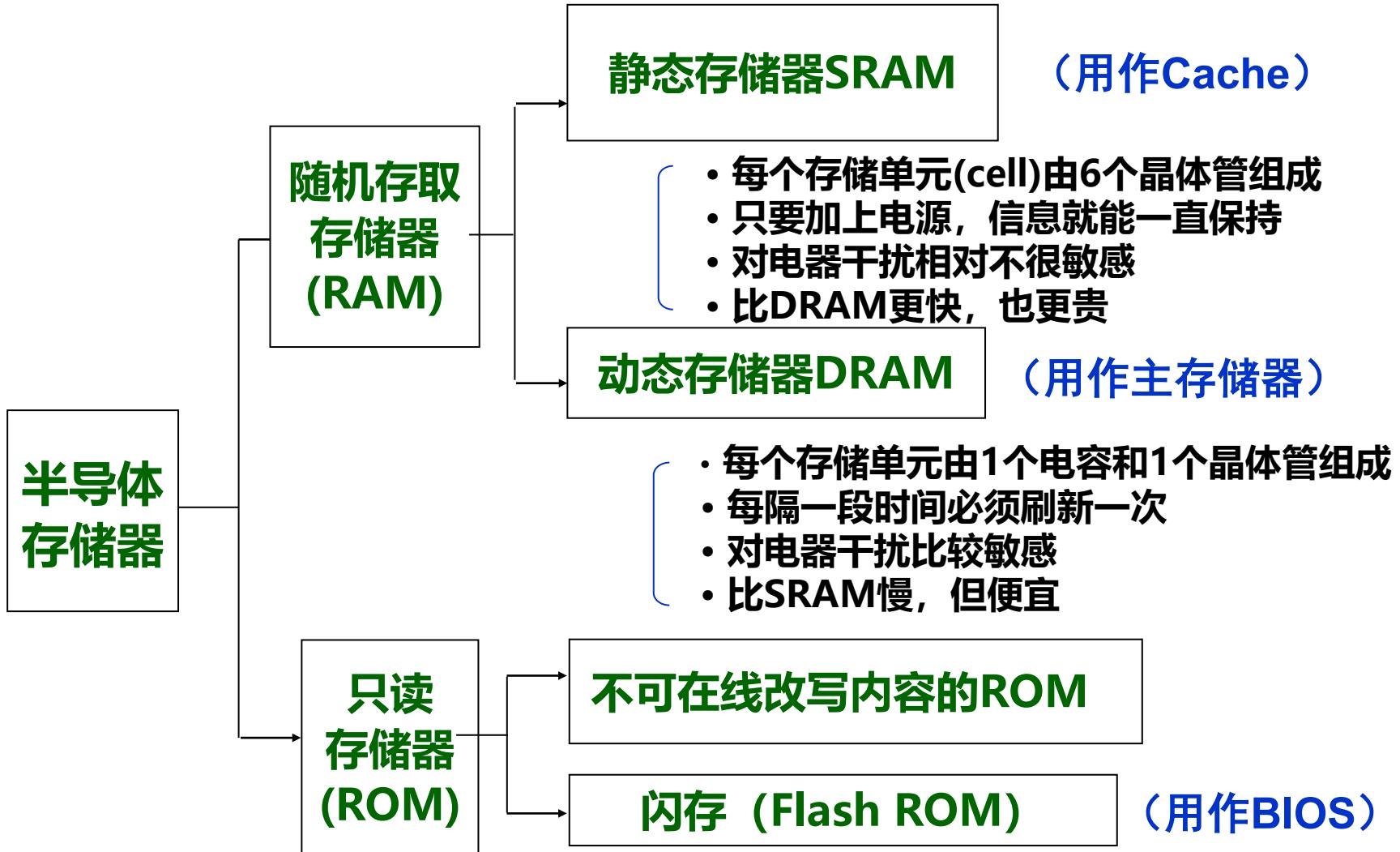
2的幂次和10的幂次单位

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

摘自《Computer Organization and Design RISC-V》

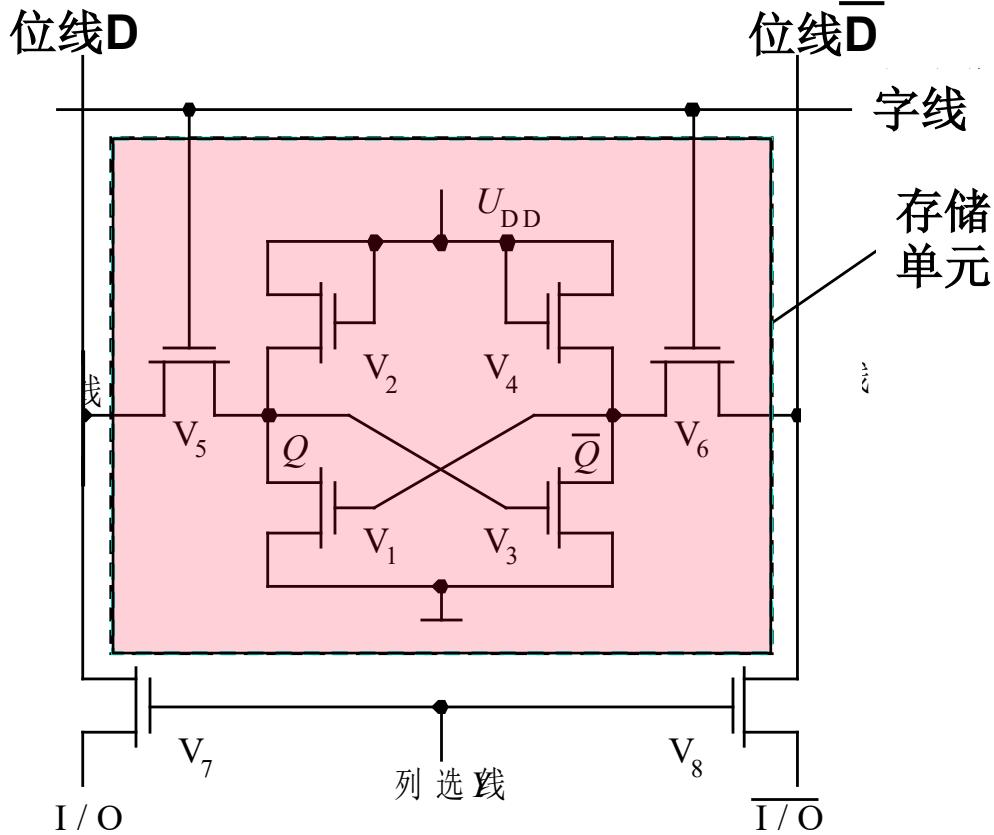
内部存储器的分类及应用

- 内部存储器由半导体存储器芯片组成，芯片有多种类型：



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，不是破坏性读出，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

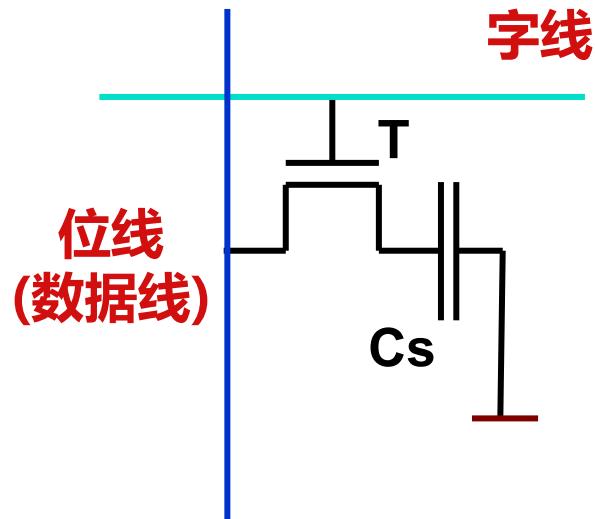
动态单管记忆单元电路（不作要求）

读写原理：字线上加高电平，使T管导通。

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

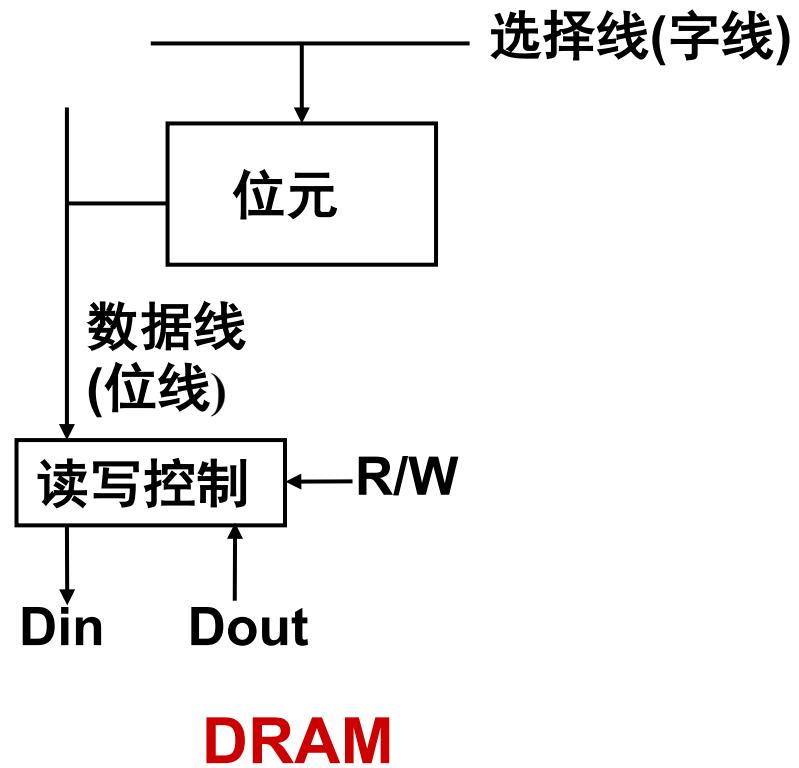
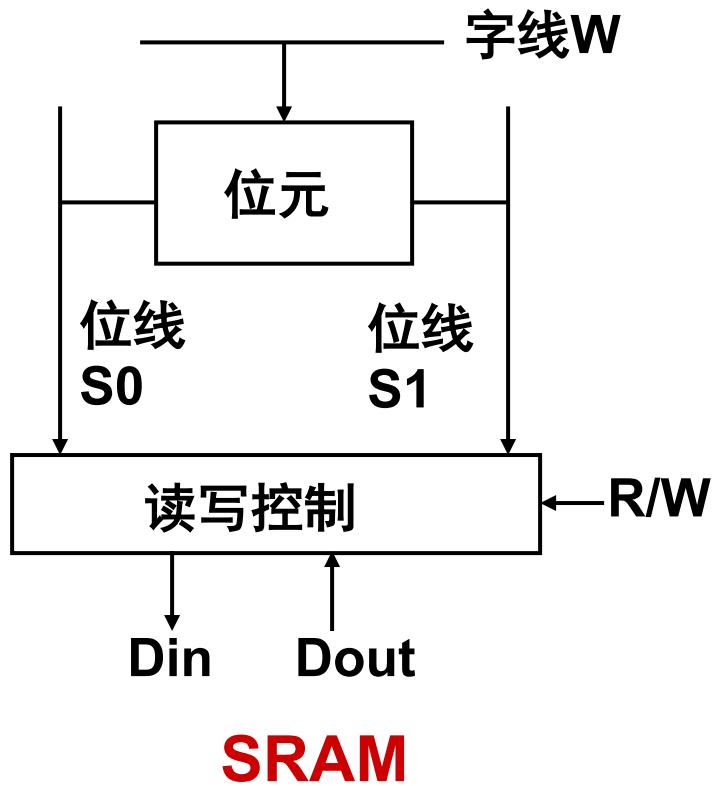
刷新：DRAM的一个重要特点是，数据以电荷的形式保存在电容中，电容的放电使得电荷通常只能维持几十个毫秒左右，相当于1M个时钟周期左右，因此要定期进行刷新（读出后重新写回），按行进行（所有芯片中的同一行一起进行），刷新操作所需时间通常只占1%~2%左右。

半导体RAM的组织

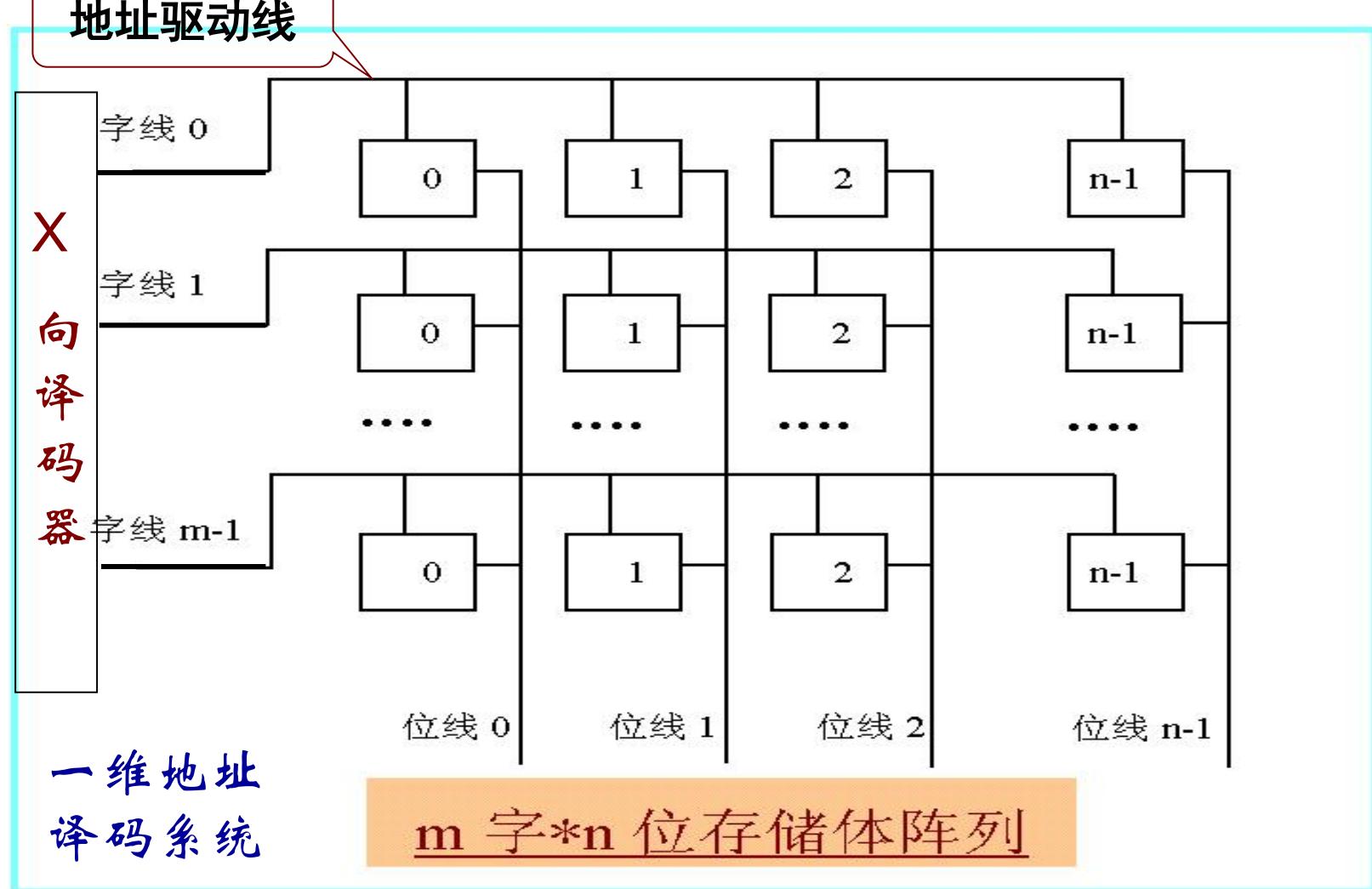
记忆单元(Cell)→存储器芯片(Chip)→ 内存条 (存储器模块)

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:

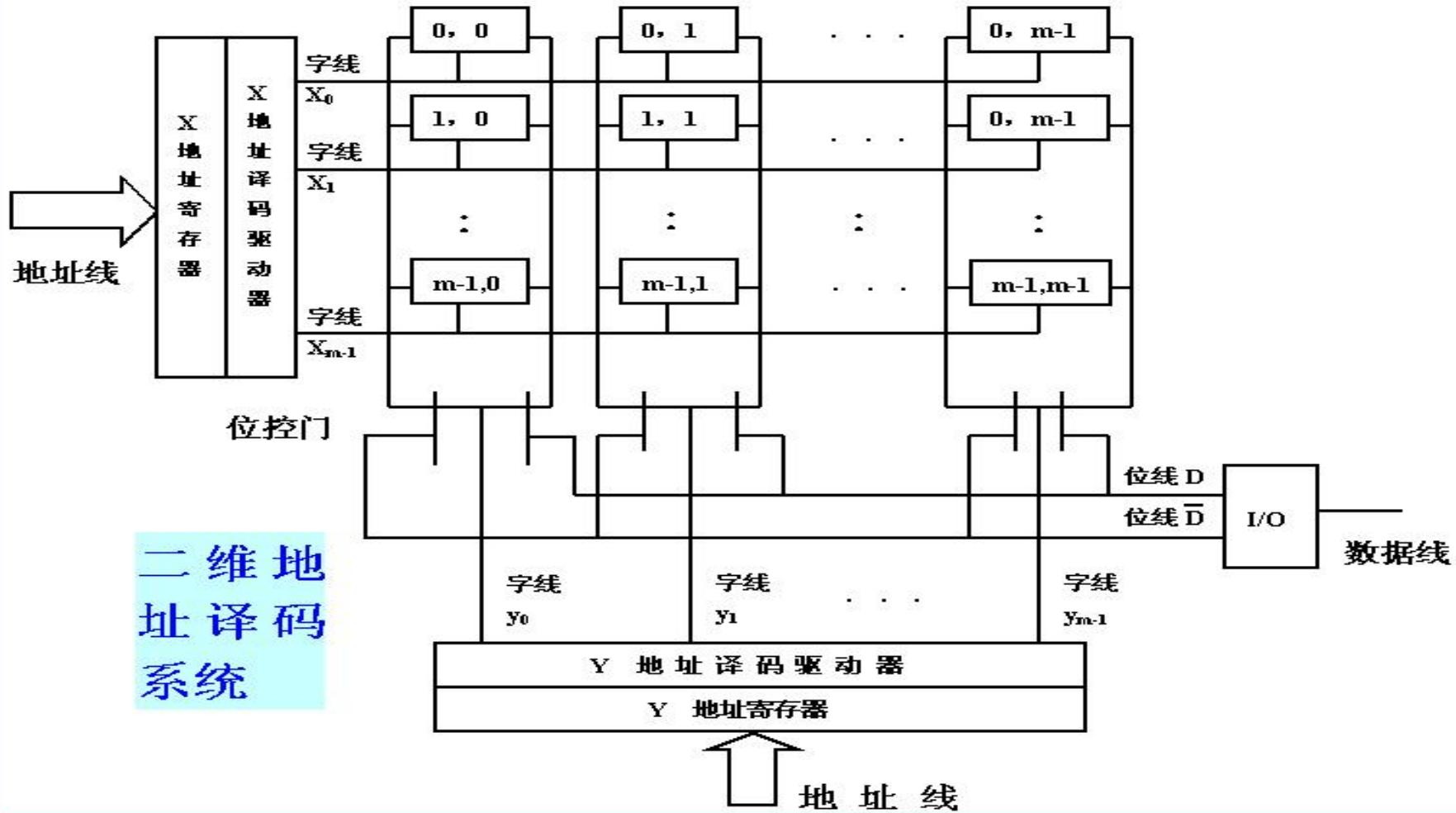


字片式存储体阵列组织（不作要求）



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

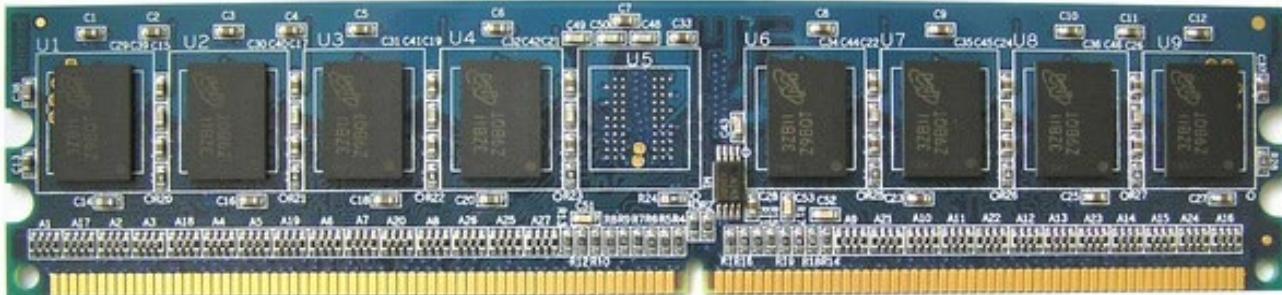
层次结构存储系统

◦ 分以下六个部分介绍

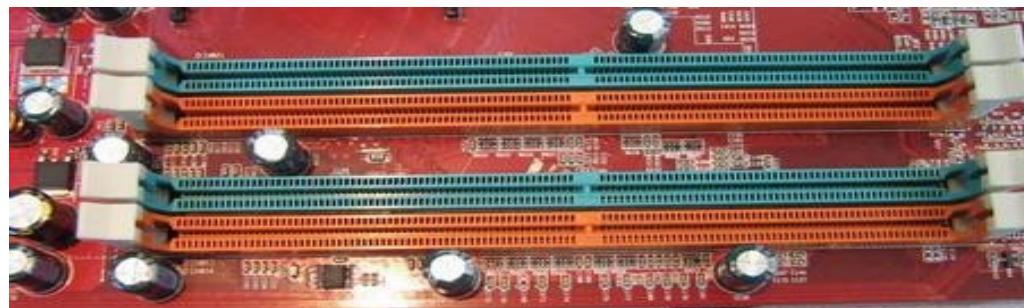
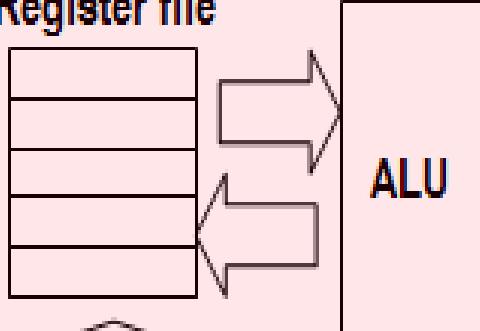
- 第一讲：存储器概述
- 第二讲：主存与CPU的连接及其读写操作
 - 主存模块的连接和读写操作
 - “装入”指令和“存储”指令操作过程
- 第三讲：磁盘存储器
- 第四讲：高速缓冲存储器(cache)
 - 程序访问的局部性、cache的基本工作原理
 - cache行和主存块之间的映射方式
 - cache和程序性能
- 第五讲：虚拟存储器
 - 虚拟地址空间、虚拟存储器的实现
- 第六讲：IA-32/Linux中的地址转换
 - 逻辑地址到线性地址的转换
 - 线性地址到物理地址的转换

主存模块的连接和读写操作

CPU chip



Register file



前端总线

存储器总线

Bus interface

I/O
bridge

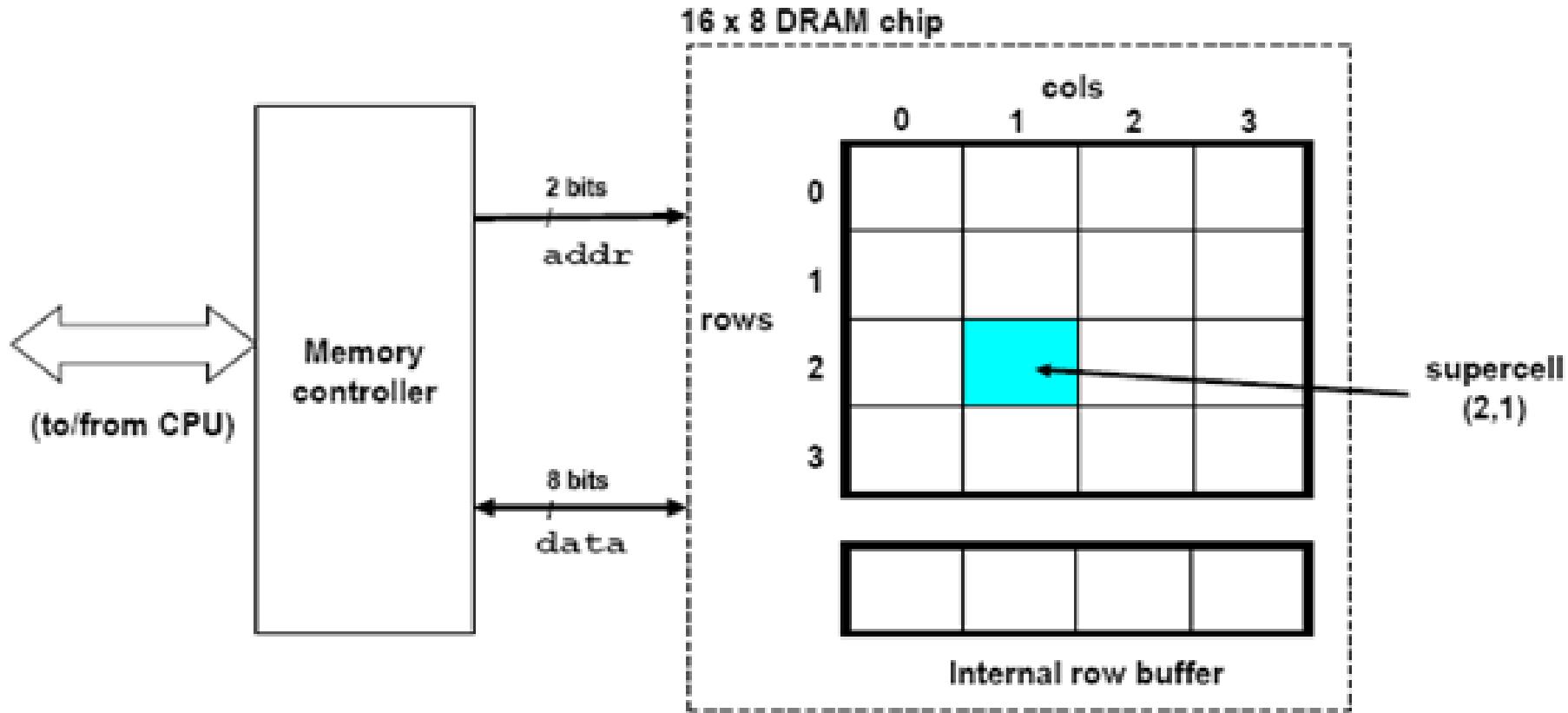
Main
memory

总线中有哪三类传输线?
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

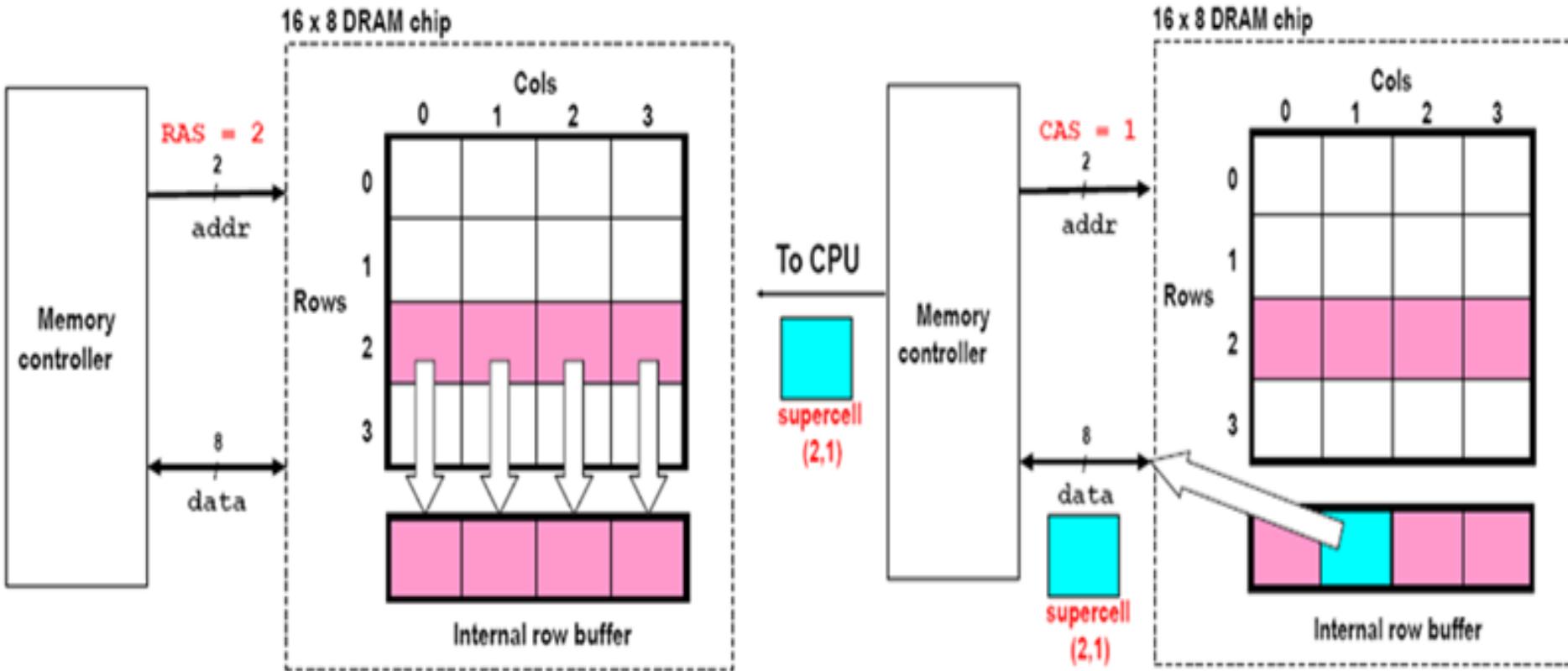
同时有多个芯片进行读写



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

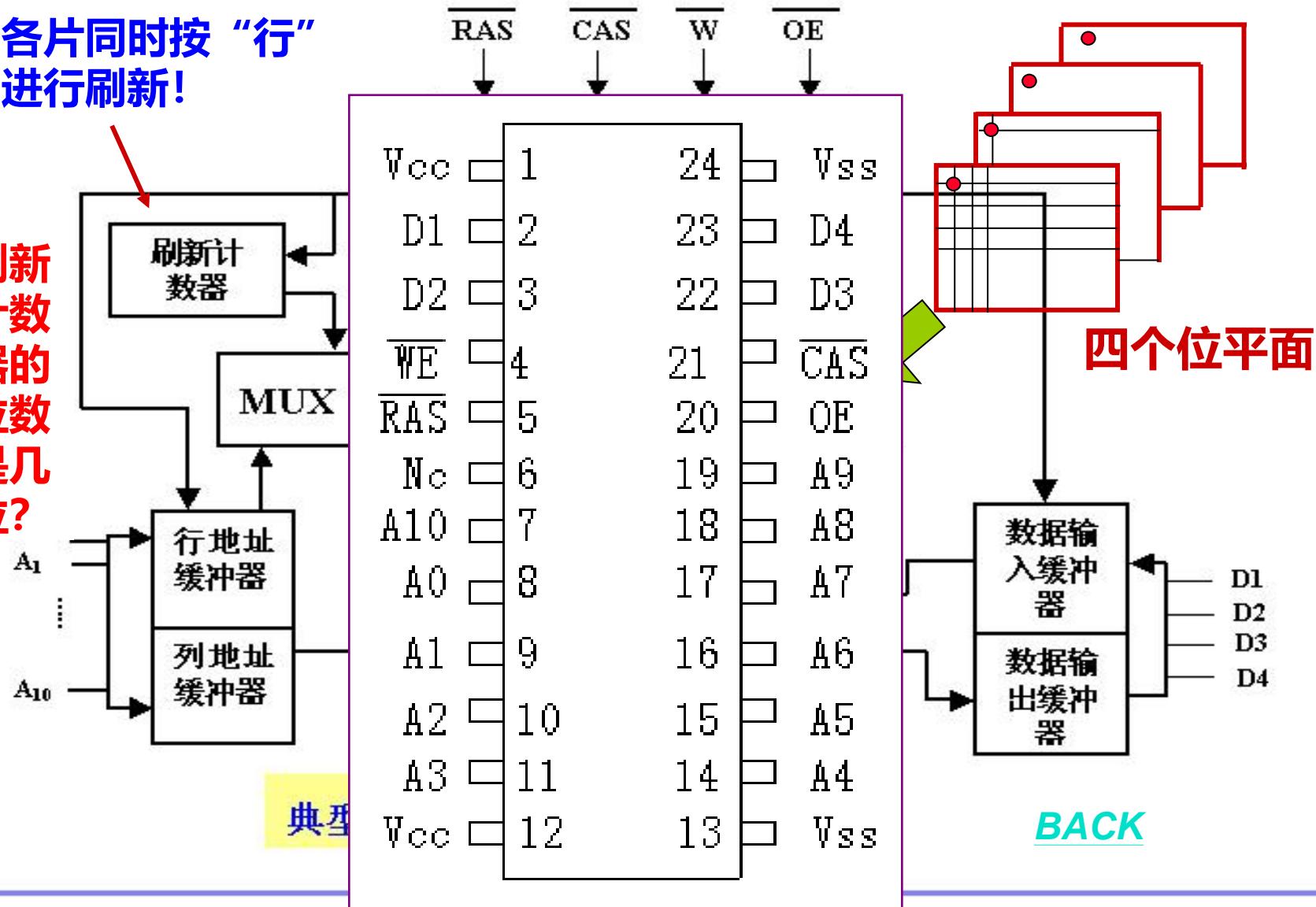


首先，存储控制器将行地址“2”送行译码器，选中第“2”行，此时，整个一行数据被送行缓冲。然后，存储控制器将列地址“1”送列译码器，选中第“1”列，此时，将行缓冲第“1”列的8位数据supercell(2,1)读到数据线，并继续送往CPU。

举例：典型的16M位DRAM (4Mx4)

各片同时按“行”
进行刷新！

刷新计数器的位数是几位？



举例：典型的16M位DRAM（4Mx4）

$$16\text{M位} = 4\text{Mb} \times 4 = 2048 \times 2048 \times 4 = 2^{11} \times 2^{11} \times 4$$

- (1) 地址线：11根线分时复用，由RAS和CAS提供控制时序。
- (2) 需4个位平面，对相同行、列交叉点的4位一起读/写
- (3) 内部结构框图

问题：

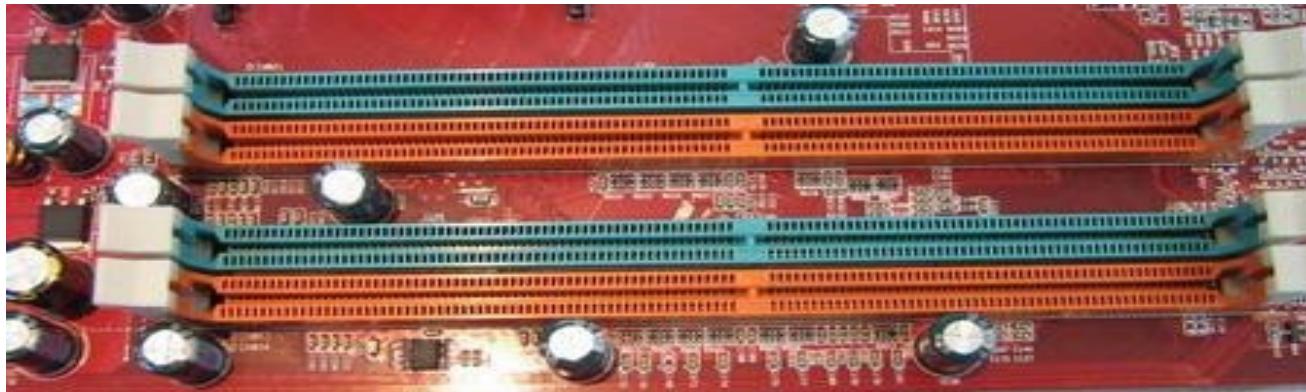
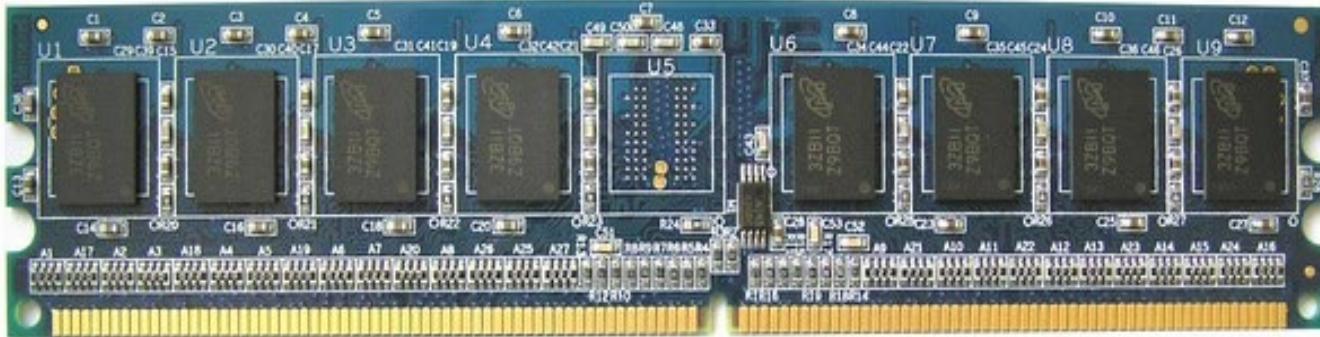
为什么每出现新一代DRAM芯片，容量至少提高到4倍？

行地址和列地址分时复用，每出现新一代DRAM芯片，至少要增加一根地址线。每加一根地址线，则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4倍。

SKIP .

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

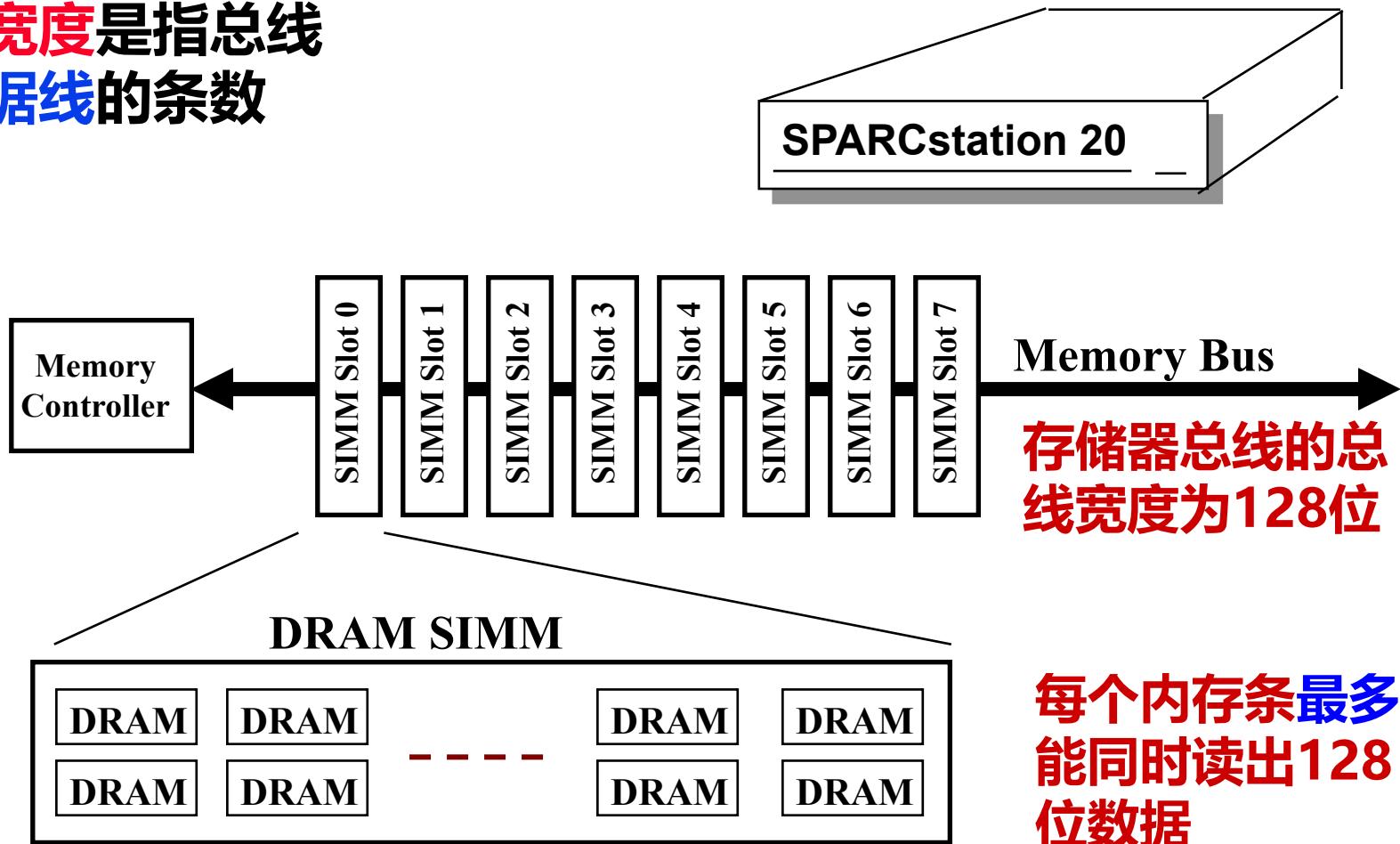


目前流行的是DDR2、DDR3内存条：

- 采用双列直插式，其触点分布在内存条的两面
- DDR条有184个引脚，DDR2有240个引脚
- PC机主板中一般都配备有2个或4个DIMM插槽

举例： SPARCstation 20's Memory Module

总线宽度是指总线
中数据线的条数



每次访存操作总是在某一个内存条内进行！

举例：SPARCstation 20's内存条(模块)

最小配置: 4 MB = 16x 2Mb DRAM chips, 8 KB of Page SRAM

最大配置: 64 MB = 32x 16Mb chips, 16 KB of Page SRAM

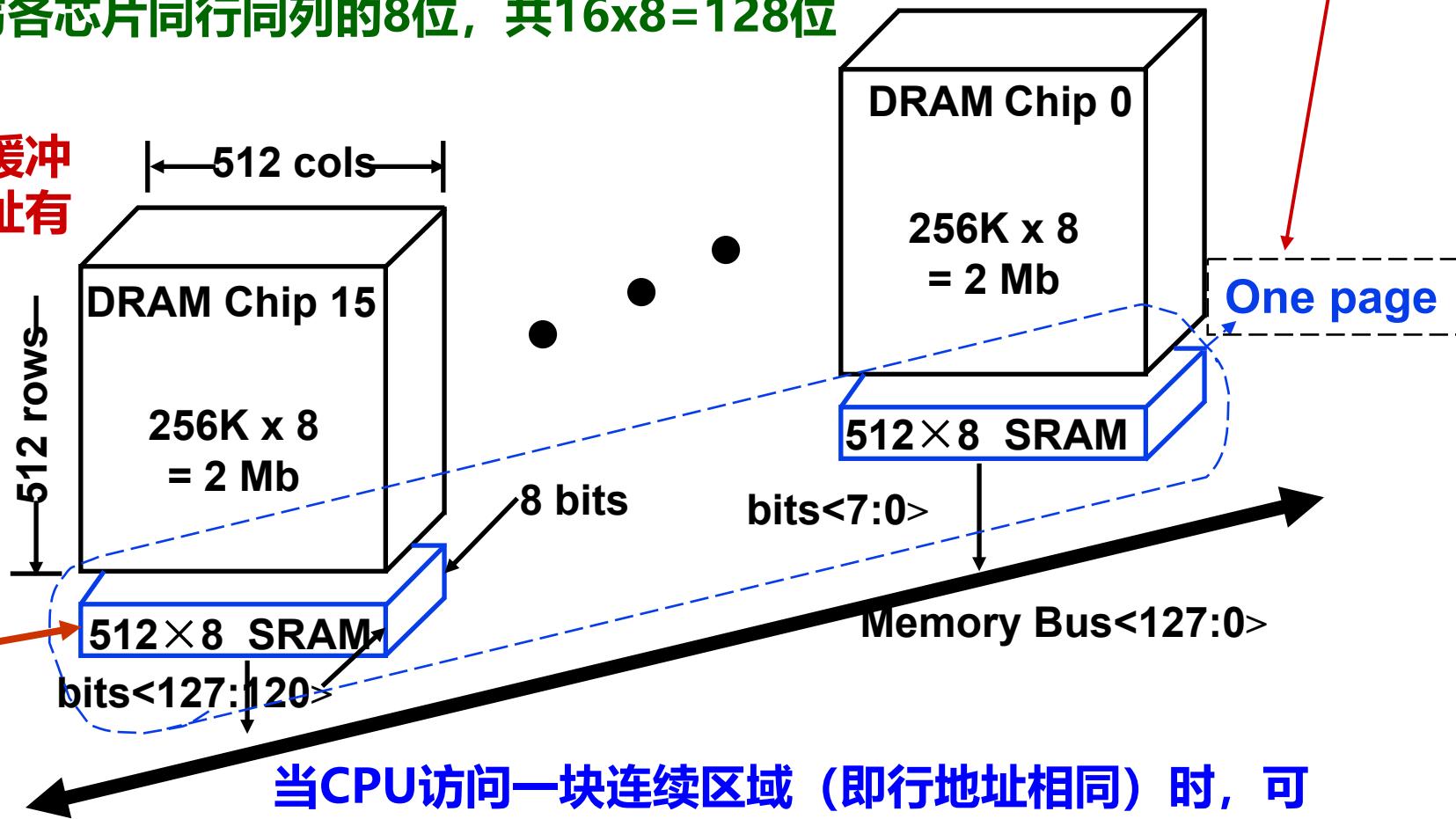
最小: $2\text{Mb} = 2^9 \times 2^9 \times 8\text{b} = 512\text{行} \times 512\text{列}$ 并有8个位平面

每次读/写各芯片同行同列的8位, 共 $16 \times 8 = 128$ 位

问题: 行缓冲
数据的地址有何特点?

一定在同一行中!
即地址是连续的!

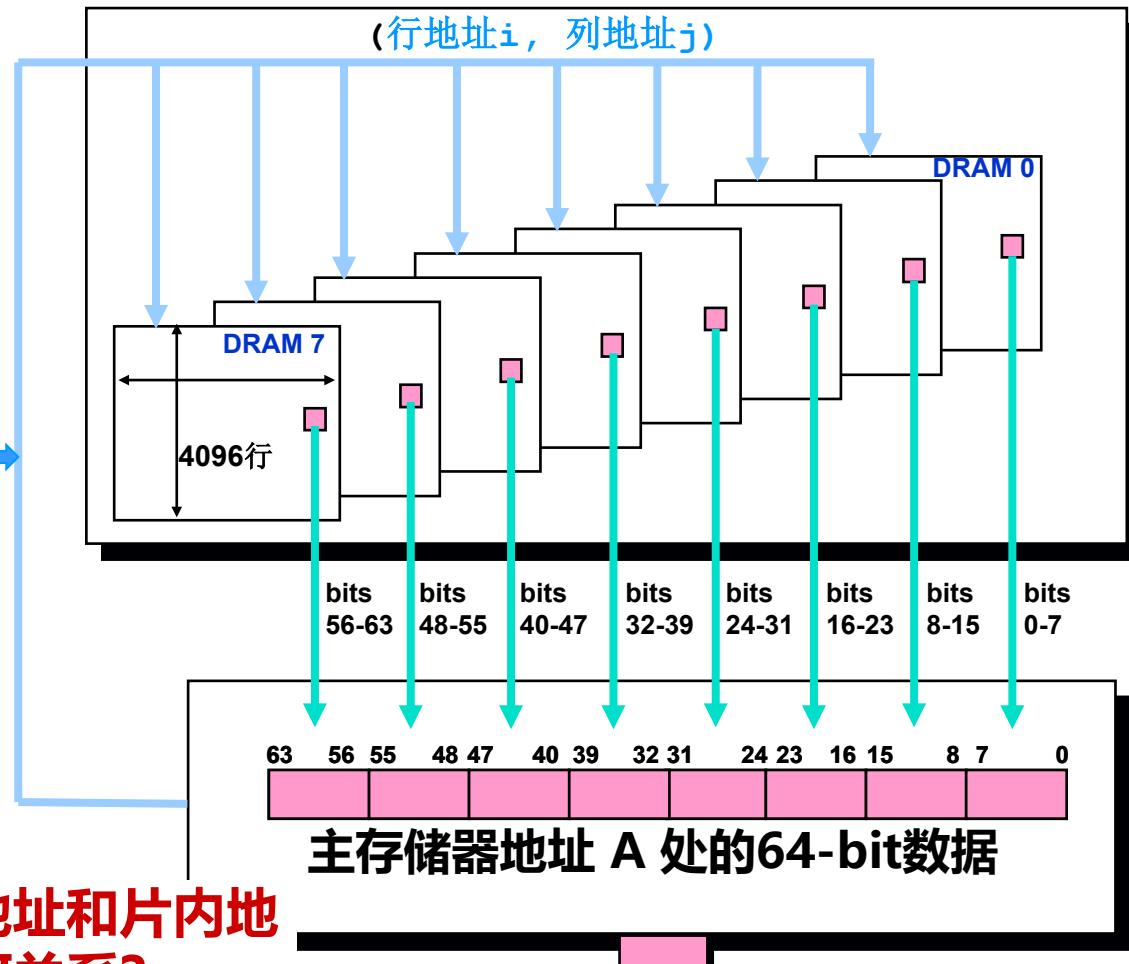
行缓冲



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, 它用SRAM实现, 速度极快!

举例：128MB的DRAM存储器

地址A →



主存地址和片内地
址有何关系？

主存地址27位，片内地址24位，
与高24位主存地址相同。

主存低3位地址的作用是什么？

确定8个字节中的哪个，即用来选片。

- 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

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。

存储控制器

■ 行、列地址为
(i,j)的8个单元

回顾： Alignment(对齐)

如： int i, short k, double x, char c, short j,.....

按字节编址

每次只能读写
某个字地址开
始的4个单元中
连续的1个、2
个、3个或4个
字节

虽节省了空间，
但增加了访存次
数！

需要权衡，目前
来看，浪费一点
存储空间没有关
系！

按边界对齐

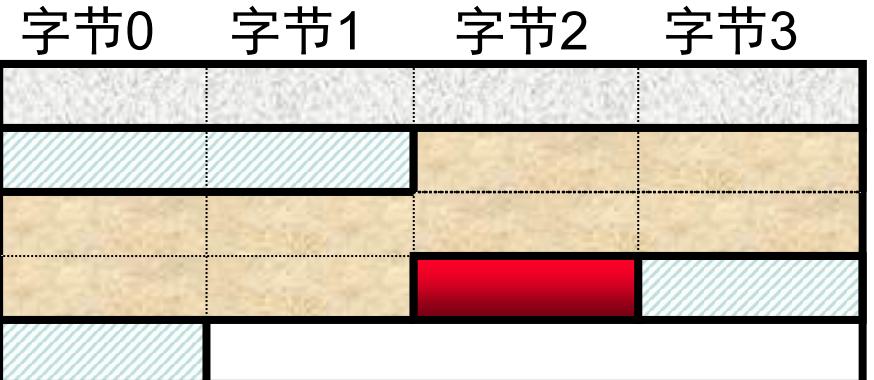
x: 2个周期
j: 1个周期



则： &i=0; &k=4; &x=8; &c=16; &j=18;.....

边界不对齐

x: 3个周期
j: 2个周期

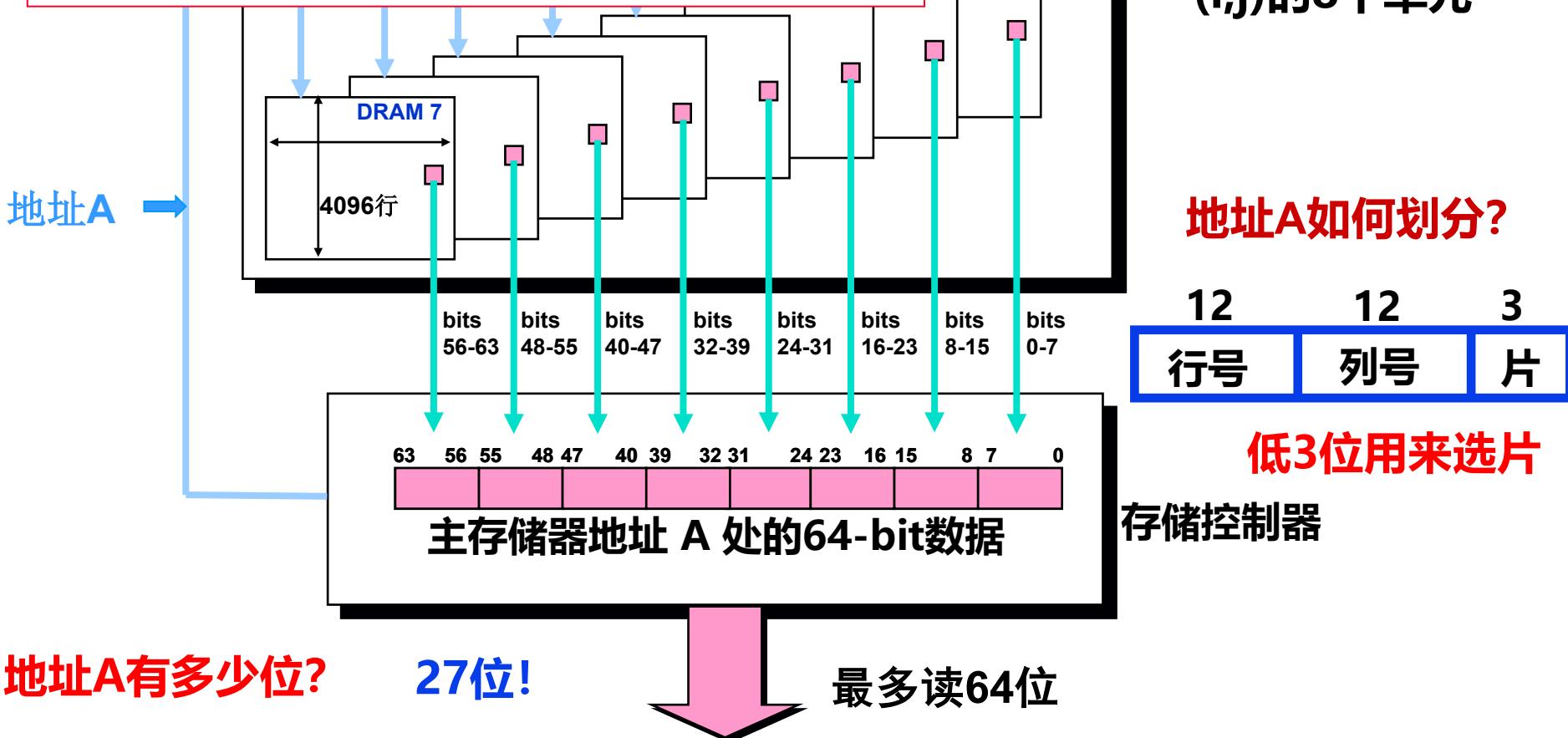


则： &i=0; &k=4; &x=6; &c=14; &j=15;.....

复习：128MB的DRAM存储器

从该存储器结构可理解为什么规定数据对齐存放。

例如，一个32位int型数据若存放在第8、9、10、11这4个单元，则需要访问几次内存？若存放在6、7、8、9这4个单元，则需要访问几次内存？



复习：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 ($i=32768*k$, k 为行号) , 则地址分布如下:

	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 $8*4096B=2^{15}B=32768$ 个单元

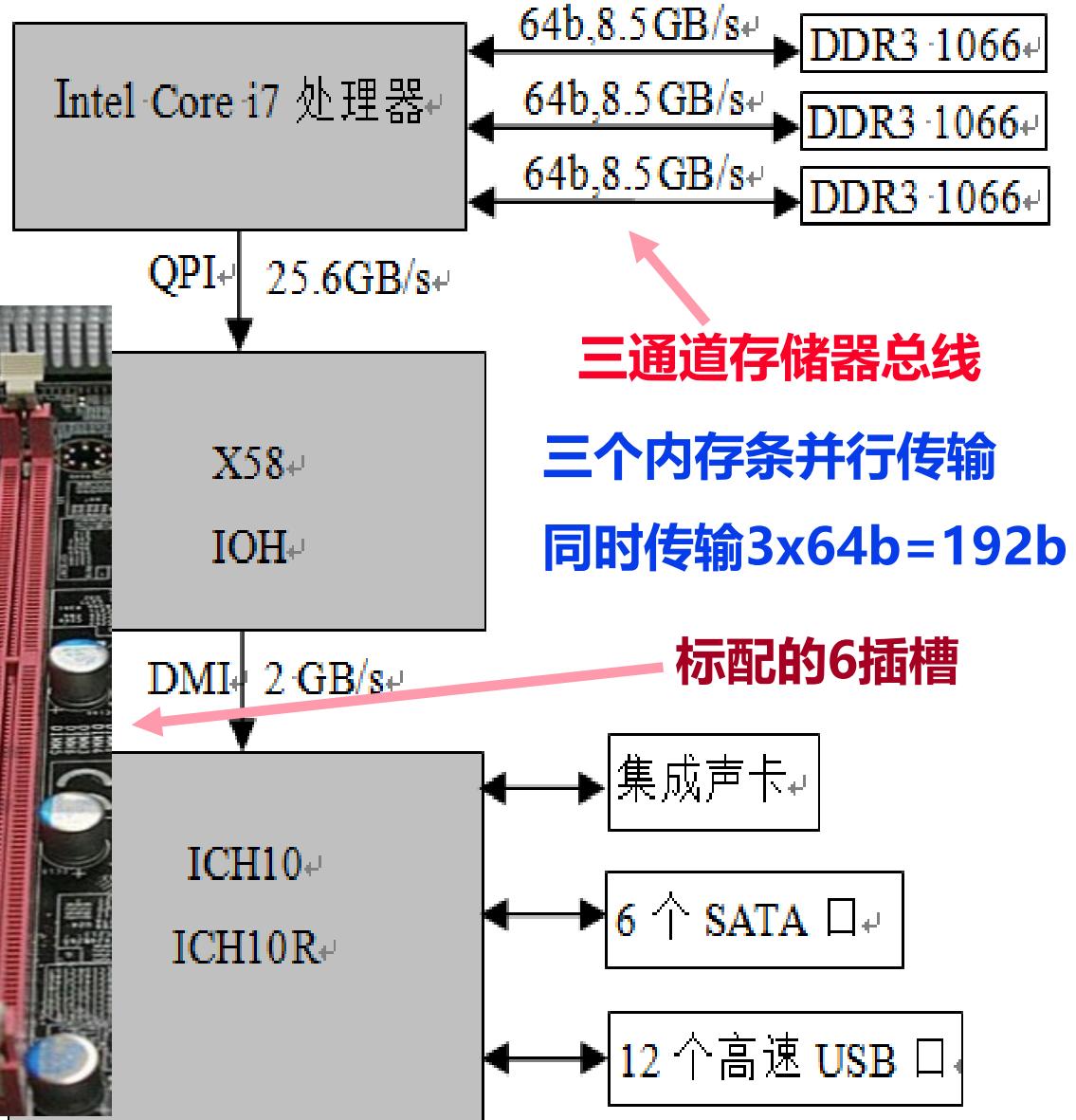
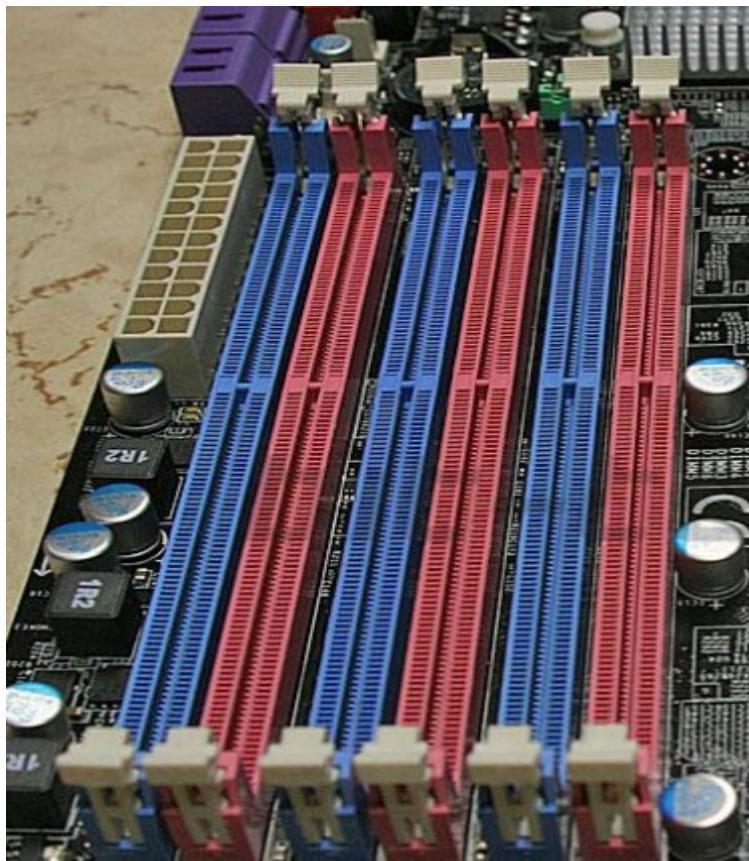
通常，一个主存块包含在行缓冲中
可降低Cache缺失损失

如果片内地址连续，则
地址A如何划分？



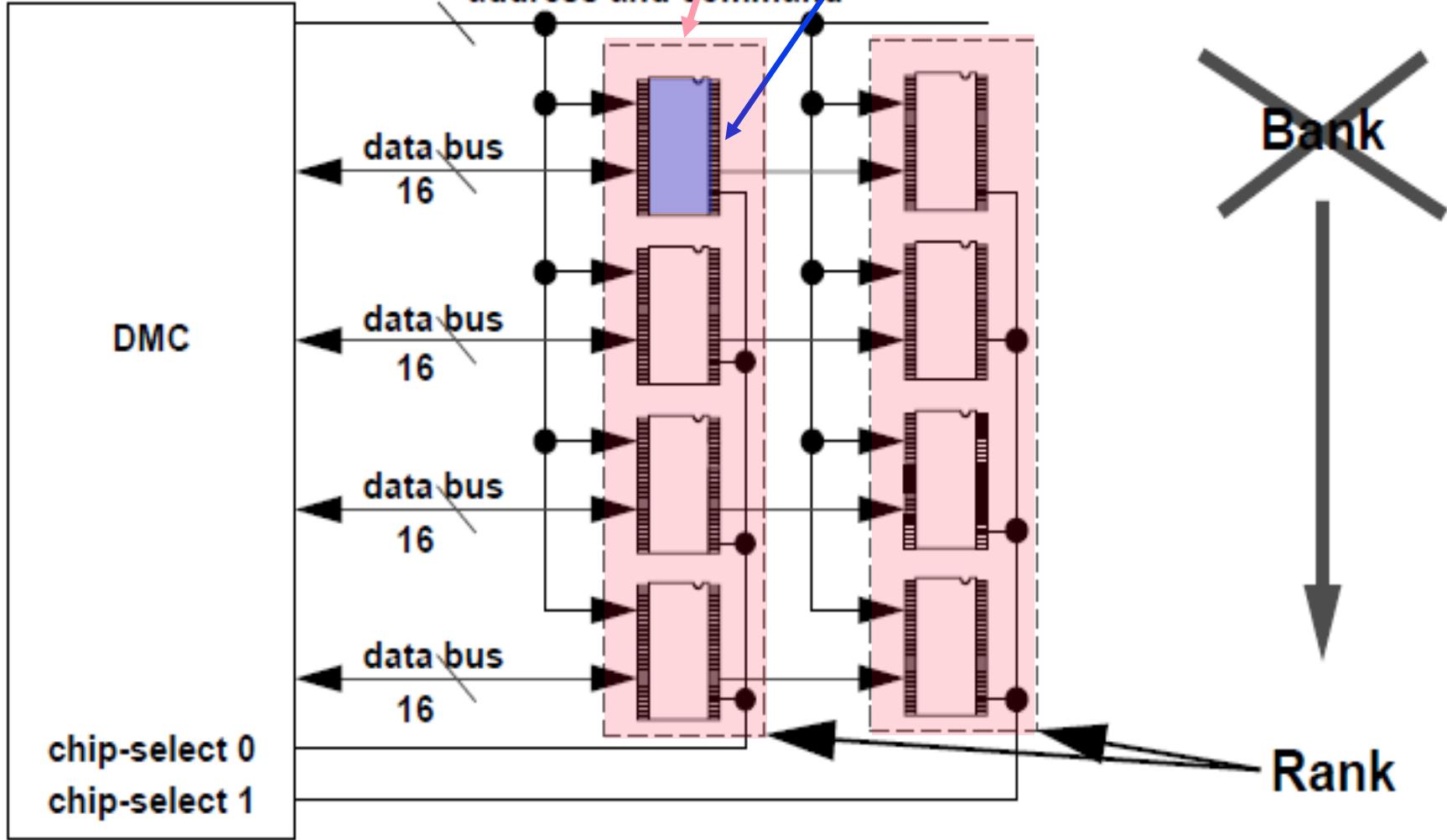
计算机系统互连

只要将同色的三个内存插槽插上内存条，系统便会自动识别进入三通道模式



DRAM内存条结构

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

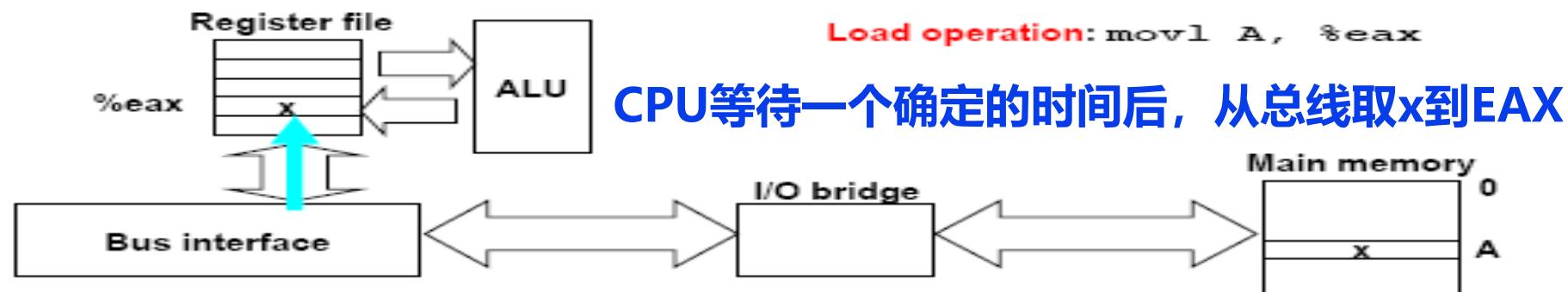
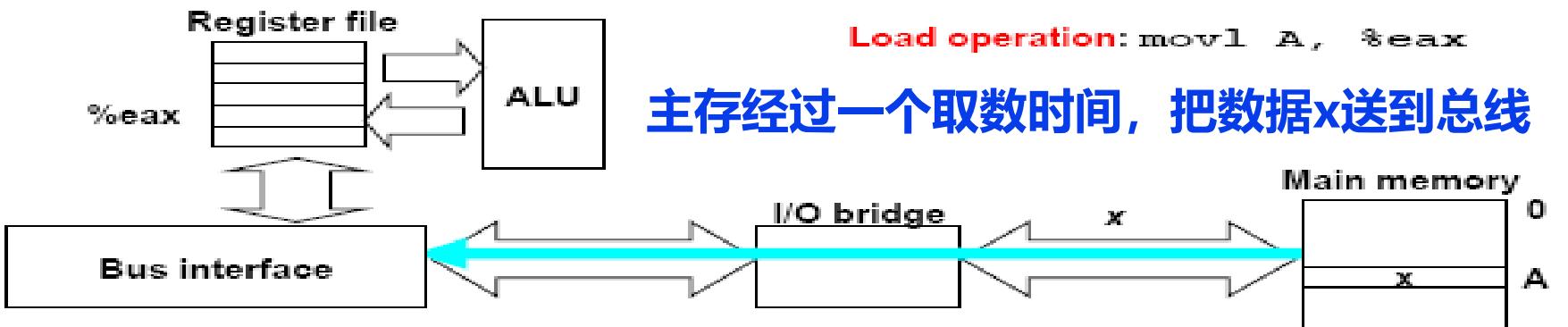
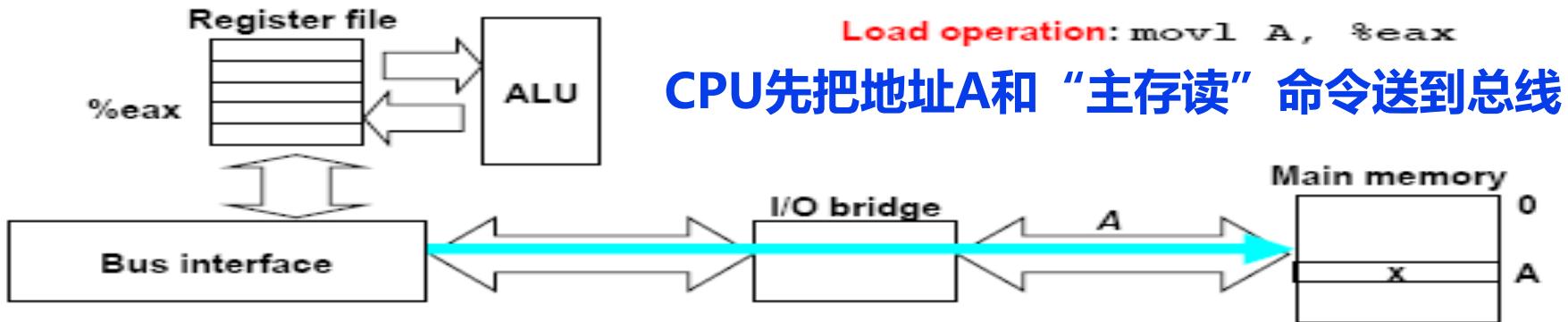


DRAM芯片的规格

- 若一个 $2^n \times b$ 位DRAM芯片的存储阵列是 r 行 $\times c$ 列，则该芯片容量为 $2^n \times b$ 位且 $2^n = r \times c$ 。如：16K×8位DRAM，则 $r=c=128$ 。
- 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，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 $r \leq c$ 且 $|r-c|$ 最小。
 - 例如，对于8K×8位DRAM芯片，其存储阵列设置为2⁶行 \times 2⁷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 $A_{12} A_{11} \dots A_1 A_0$ 中，行地址为 $A_{12} A_{11} \dots A_7$ ，列地址为 $A_6 \dots A_1 A_0$ 。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

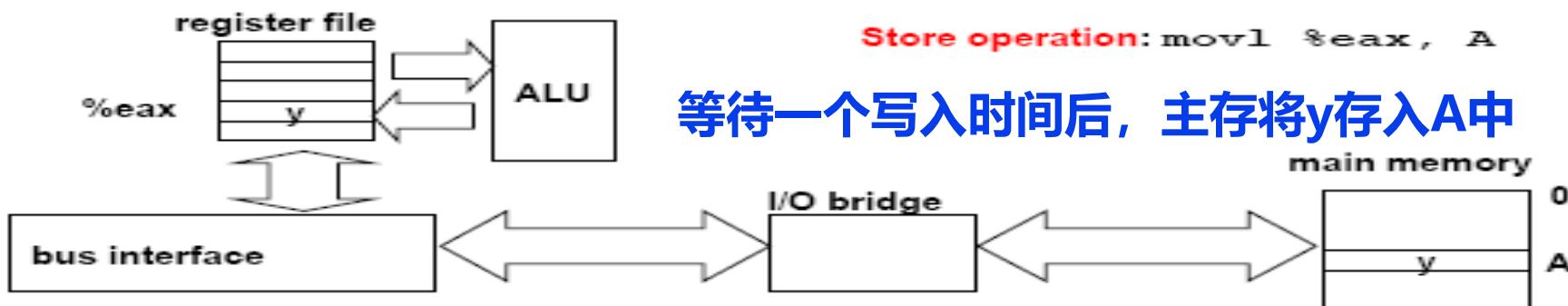
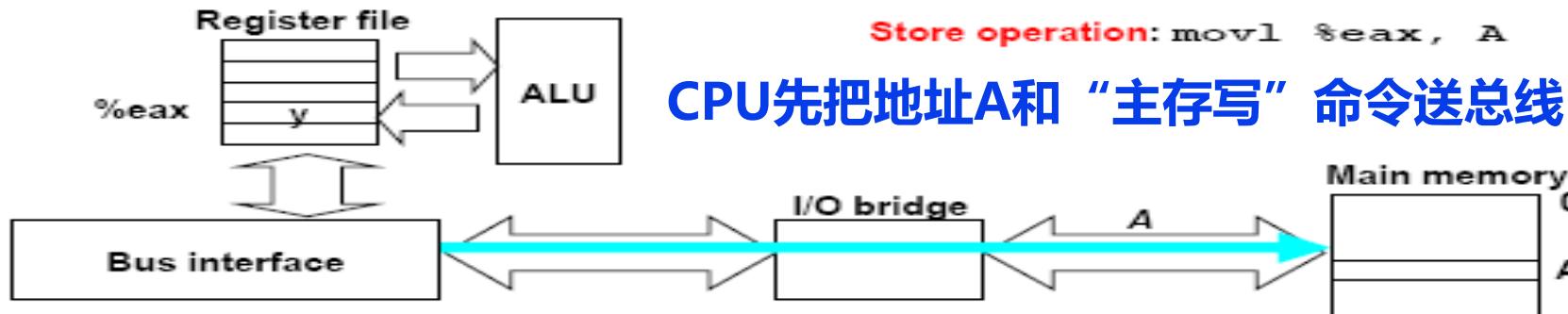
指令“`movl 8(%ebp), %eax`”操作过程

由`8(%ebp)`得到地址A的过程较复杂，涉及MMU、TLB、页表等重要概念！



指令“`movl %eax,8(%ebp)`”操作过程

由`8(%ebp)`得到地址A的过程较复杂，涉及MMU、TLB、页表等重要概念！

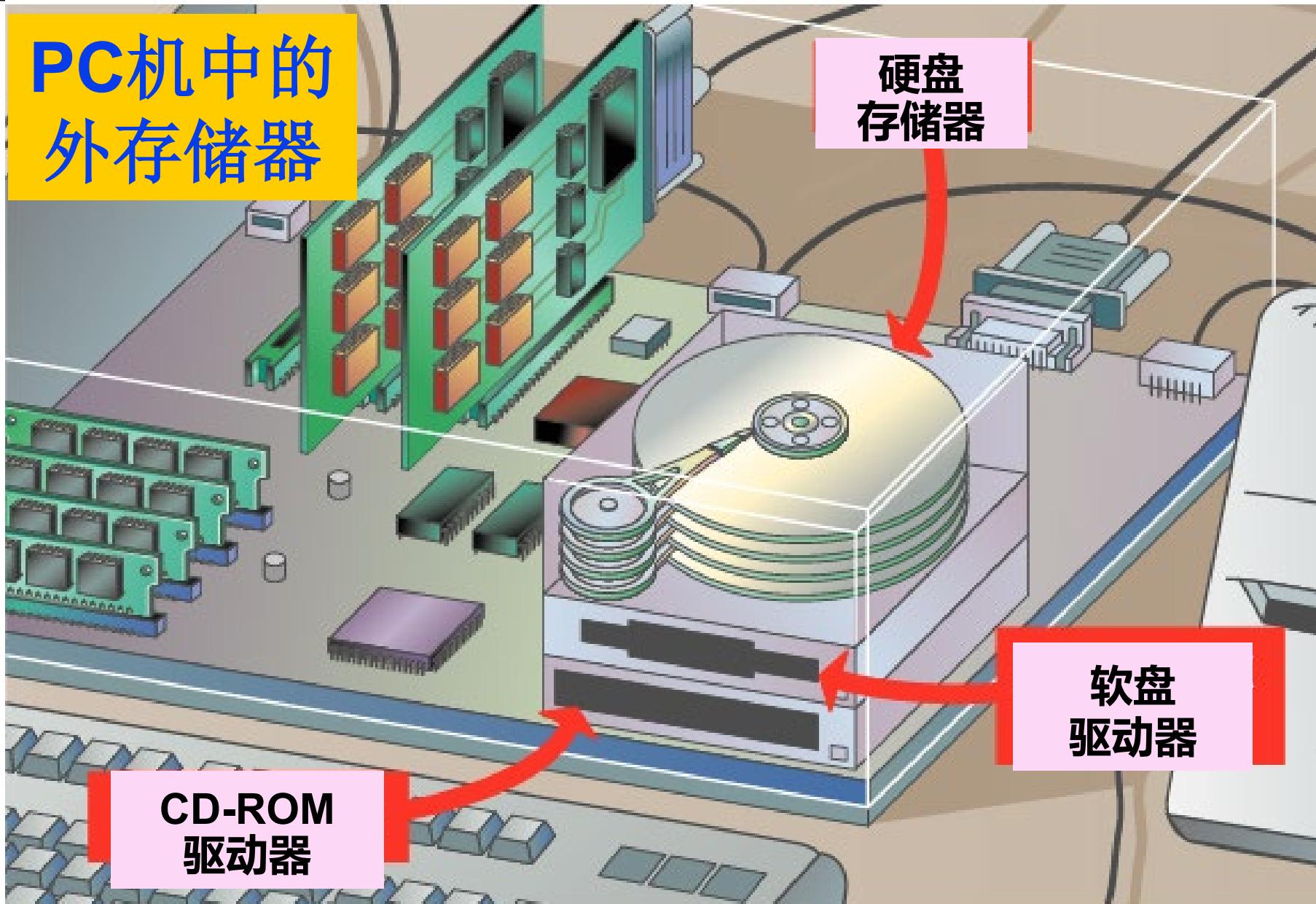


层次结构存储系统

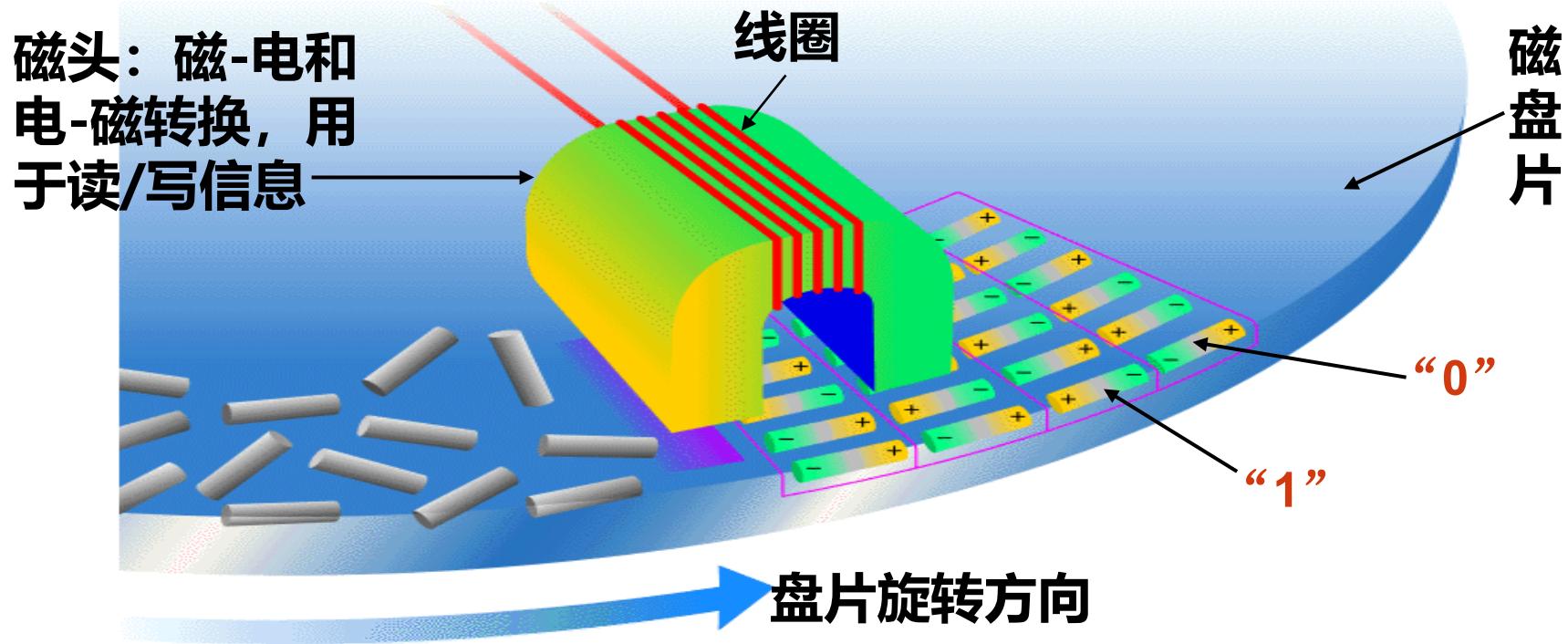
◦ 分以下六个部分介绍

- 第一讲：存储器概述
- 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：磁盘存储器
- 第四讲：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
- 第五讲：虚拟存储器
 - 虚拟地址空间、虚拟存储器的实现
- 第六讲：IA-32/Linux中的地址转换
 - 逻辑地址到线性地址的转换
 - 线性地址到物理地址的转换

PC中的外存储器



磁盘存储器的信息存储原理



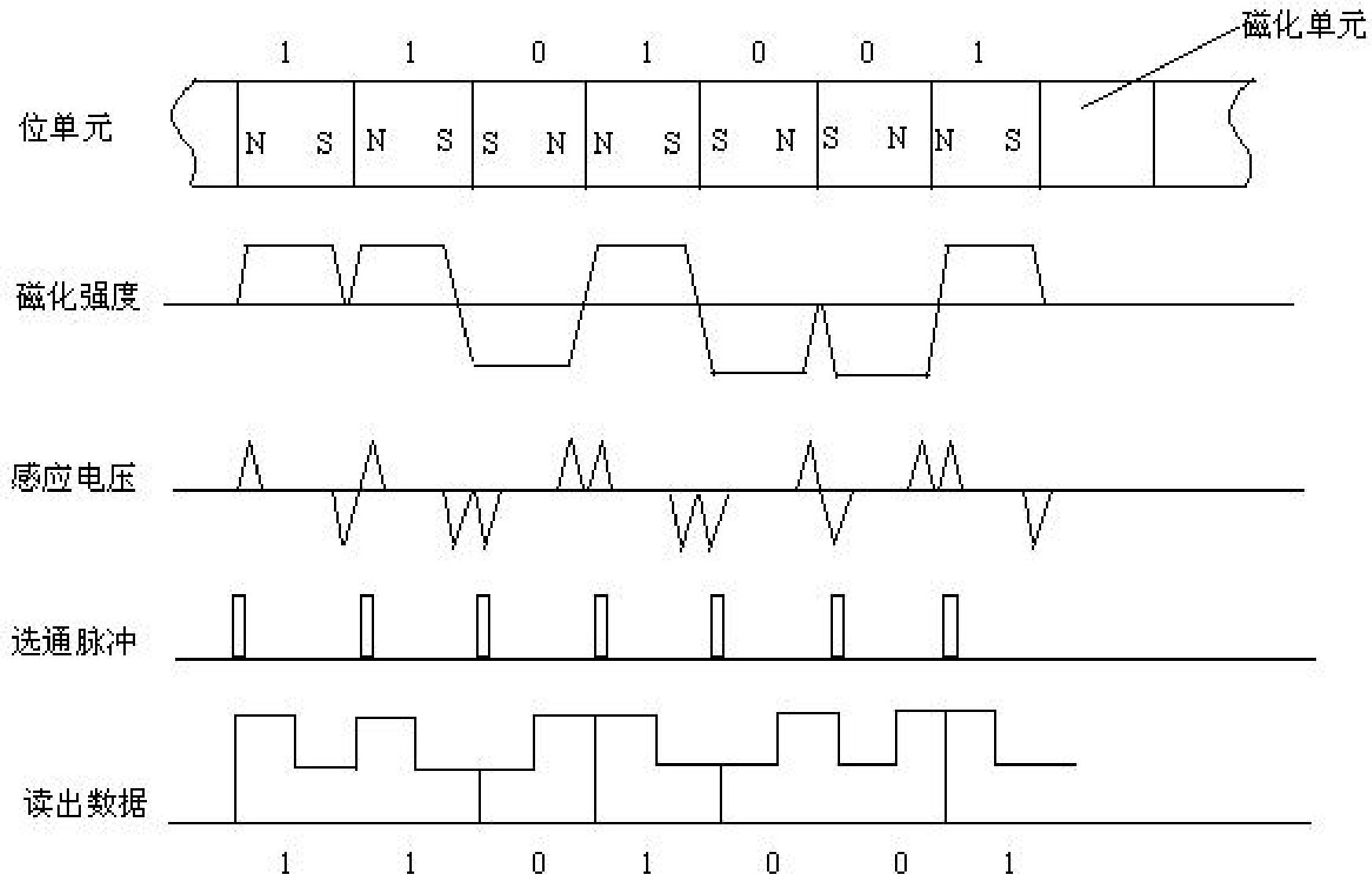
写1：线圈通以正向电流，使呈N-S状态

写0：线圈通以反向电流，使呈S-N状态

读时：磁头固定不动，载体运动。因为载体上小的磁化单元外部的磁力线通过磁头铁芯形成闭合回路，在铁芯线圈两端得到感应电压。根据感应电压的不同的极性，可确定读出为0或1。

} 不同的磁化状态被
记录在磁盘表面

磁表面信息读出过程



磁盘的磁道和扇区

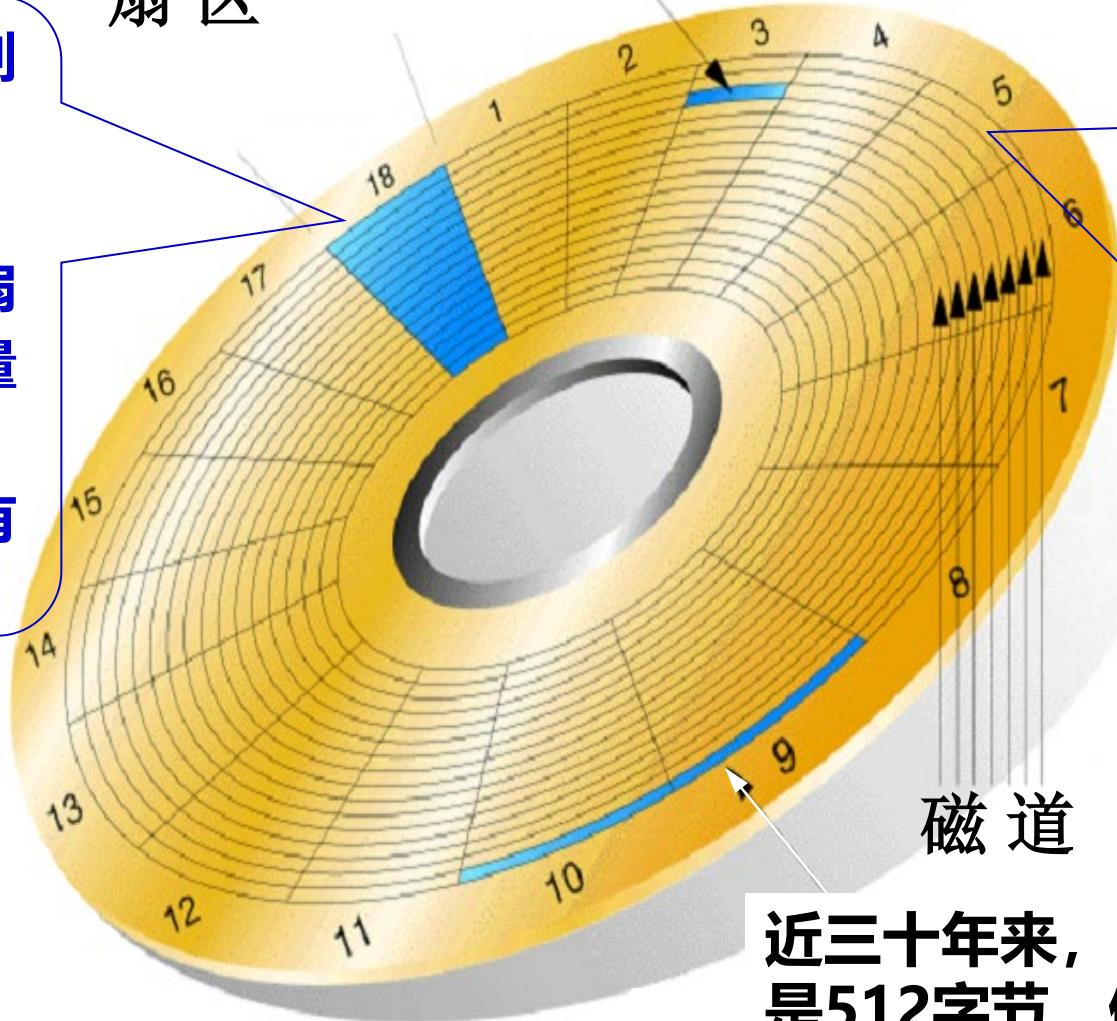
每个磁道被划分为若干段（段又叫扇区），每个扇区的存储容量为**512字节**。每个扇区都有一个编号

扇区

磁盘表面被分为许多同心圆，每个同心圆称为一个磁道。每个磁道都有一个编号，最外面的是**0磁道**

磁道

近三十年来，扇区大小一直是512字节。但最近几年正迁移到更大、更高效的4096字节扇区，通常称为**4K扇区**。



磁盘磁道的格式

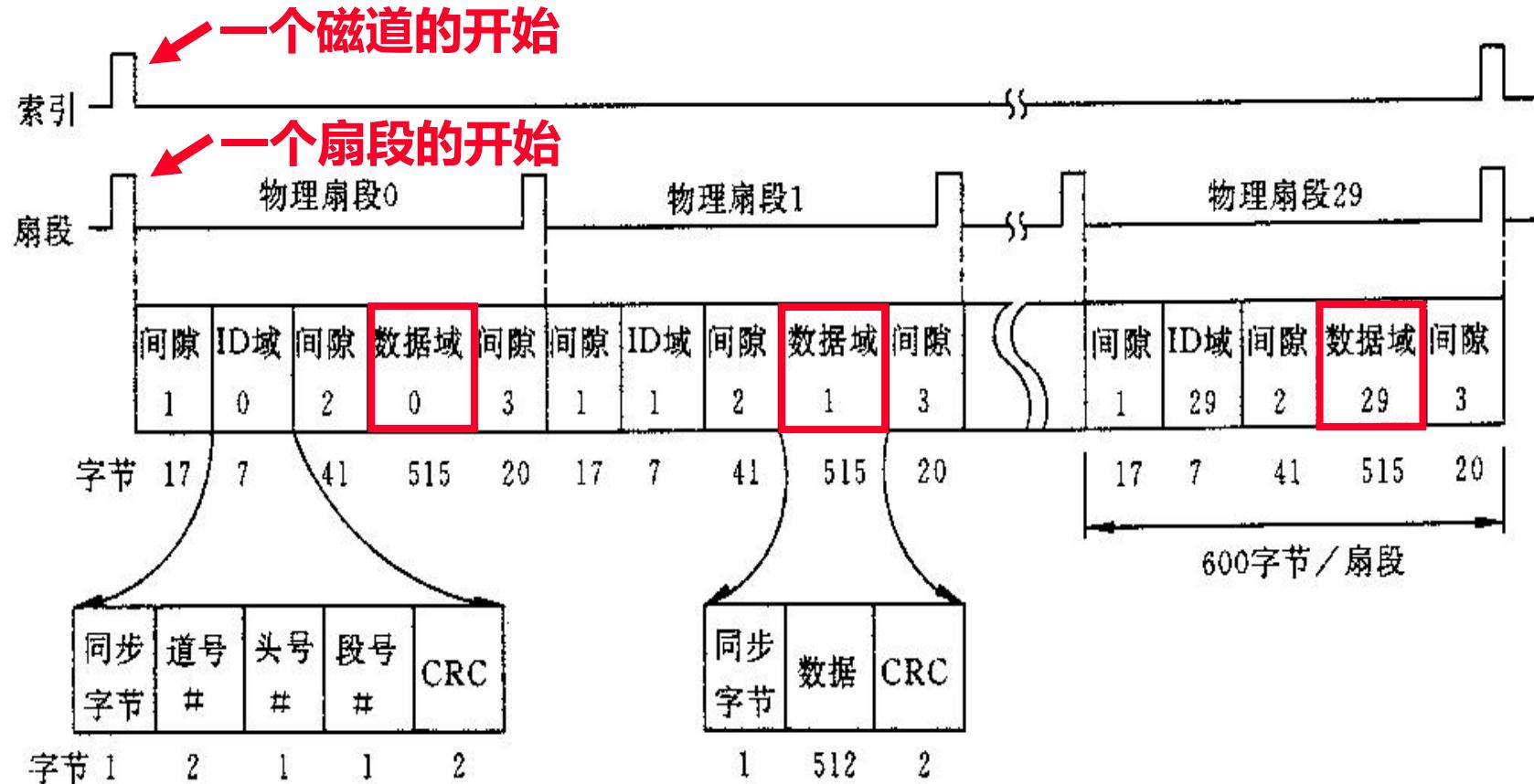
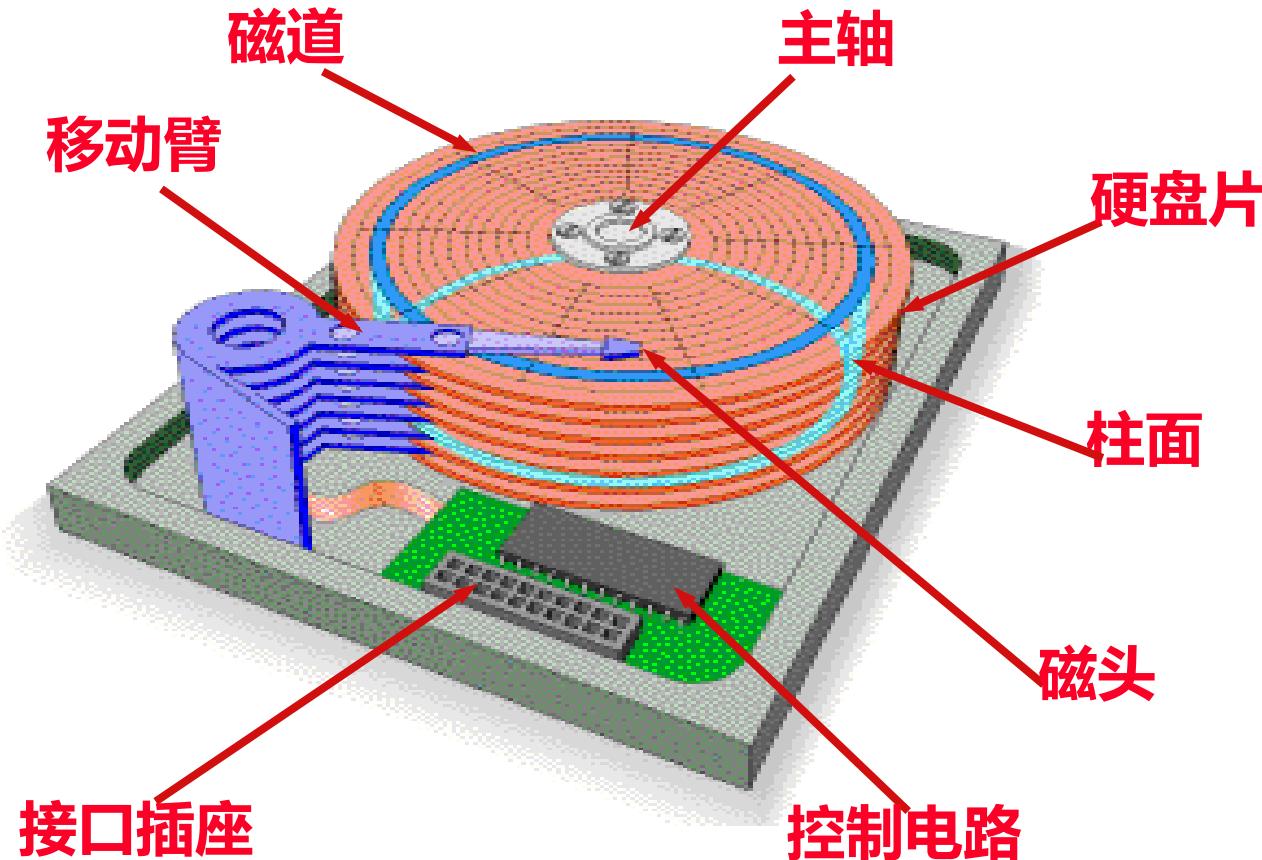


图 5.2 温彻斯特磁盘磁道格式(Seagate ST506)

磁盘格式化操作指在盘面上划分磁道和扇区，并在扇区中填写ID域信息的过程

在此例中，每个磁道包含30个固定长度的扇段，每个扇段有600个字节 $(17+7+41+515+20=600)$ 。

磁盘驱动器



假定有5片、1000个同心圆，每个圆分2000个扇区

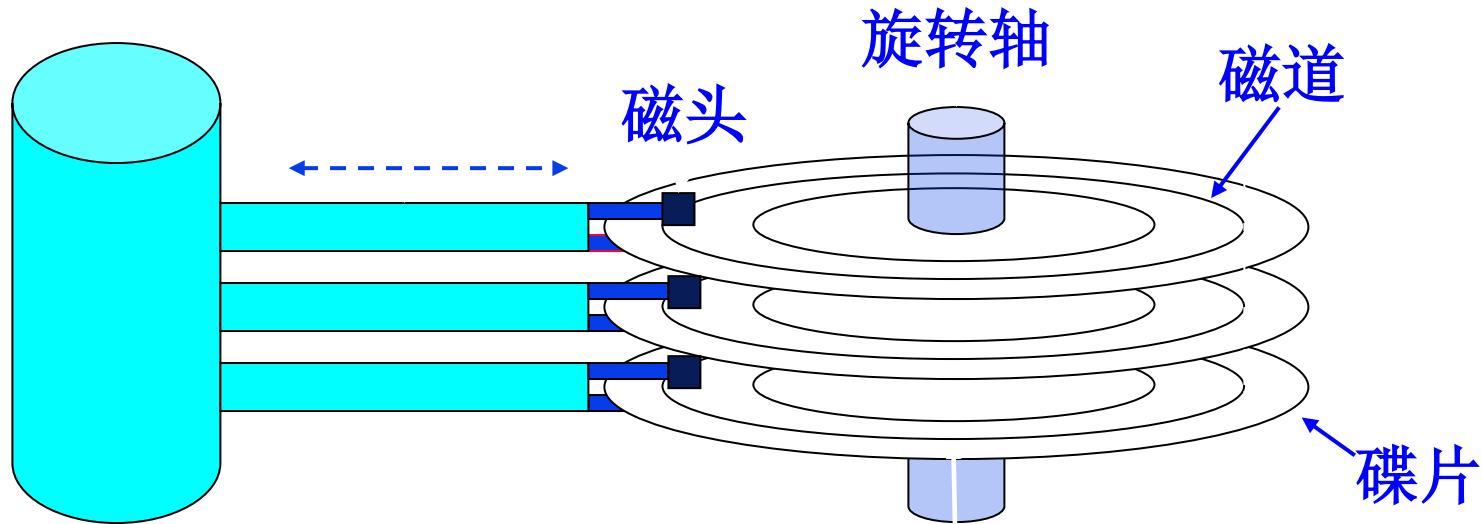
磁盘空间划分为1000个柱面、每柱面10个磁头，每个磁头形成一个磁道，每个磁道2000个扇区，每扇区512B

磁道号就是柱面号、磁头号就是盘面号，每片有两个面，每面一个磁头

磁盘数据地址：

10	4	11
柱面号	磁头号	扇区号

平均存取时间



硬盘的操作流程如下：

所有磁头同步寻道（由柱面号控制）→ 选择磁头（由磁头号控制）→
被选中磁头等待扇区到达磁头下方（由扇区号控制）→ 读写该扇区中数据

◦ 磁盘信息以扇区为单位进行读写，平均存取时间为：

$$T = \text{平均寻道时间} + \text{平均旋转等待时间} + \text{数据传输时间} \quad (\text{忽略不计})$$

• 平均寻道时间——磁头寻找到指定磁道所需平均时间 (约5ms)

• 平均旋转等待时间——指定扇区旋转到磁头下方所需平均时间

(约4 ~ 6ms) (转速: 4200 / 5400 / 7200 / 10000rpm)

• 数据传输时间——(大约0.01ms / 扇区)

磁盘响应时间计算举例

- 假定每个扇区512字节，磁盘转速为5400 RPM，声称寻道时间（最大寻道时间的一半）为12 ms，数据传输率为4 MB/s，磁盘控制器开销为1 ms，不考虑排队时间，则磁盘响应时间为多少？

Disk Response Time = Queuing Delay + Controller Time +
Seek time + Rotational Latency + Transfer time

$$\begin{aligned} &= 0 + 1 \text{ ms} + 12 \text{ ms} + 0.5 / 5400 \text{ RPM} + 0.5 \text{ KB} / 4 \text{ MB/s} \\ &= 0 + 1 \text{ ms} + 12 \text{ ms} + 0.5 / 90 \text{ RPS} + 0.128 / 1000 \text{ s} \\ &= 1 \text{ ms} + 12 \text{ ms} + 5.5 \text{ ms} + 0.1 \text{ ms} \\ &= 18.6 \text{ ms} \end{aligned}$$

如果实际的寻道时间只有1/3的话，则总时间变为10.6ms，这样旋转等待时间就占了近50%！

$$12/3+5.5+0.1+1=10.6\text{ms}$$

为什么实际的寻道时间可能只有1/3？ 磁盘转速非常重要！

访问局部性使得每次磁盘访问大多在局部几个磁道，实际寻道时间变少！

每道有多少扇区？

$(4\text{MB} \times 60 / 5400) / 512\text{B} \approx 87$ 个扇区

硬盘存储器的组成

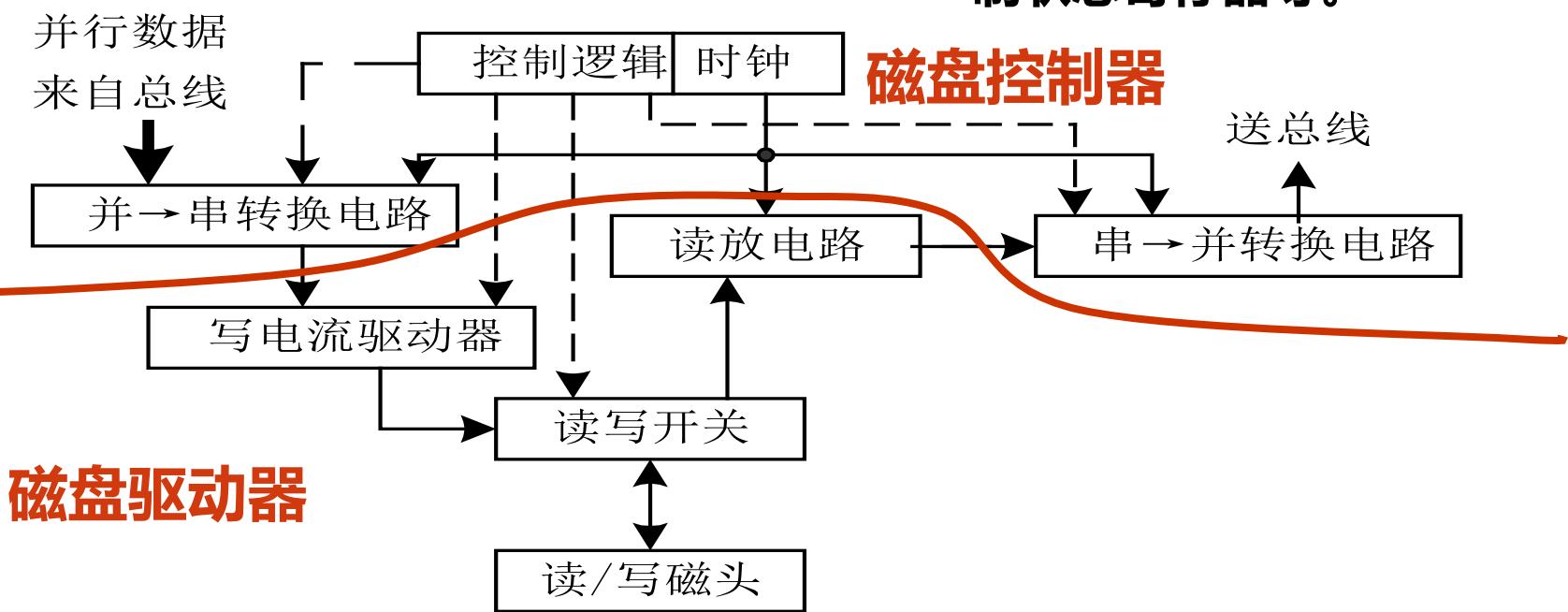
- 硬盘存储器的基本组成

磁记录介质：用来保存信息

磁盘驱动器：包括读写电路、读\写转换开关、磁头与磁头定位伺服系统等

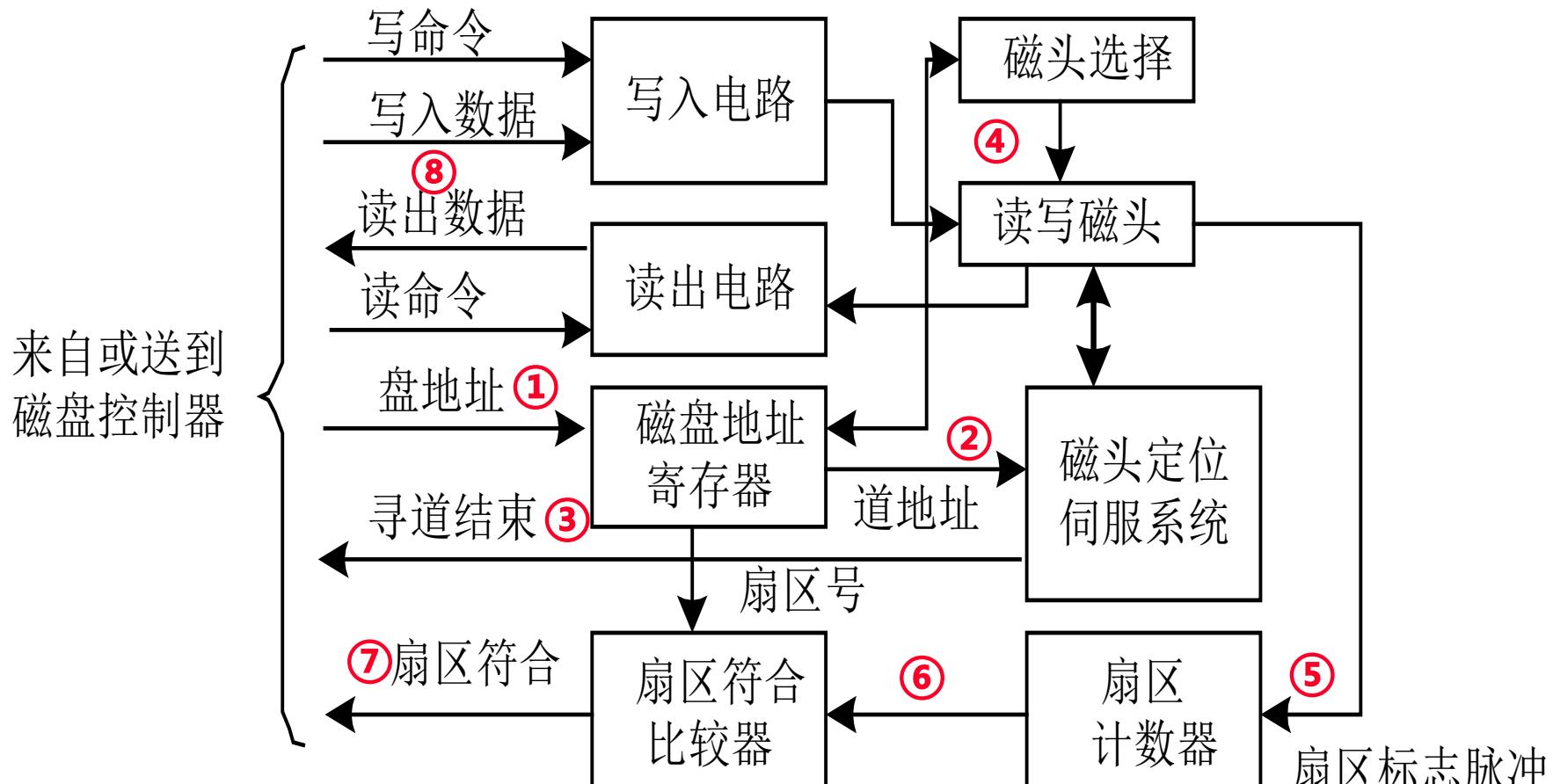
磁盘控制器：包括控制逻辑、时序电路、“并→串”转换和“串→并”转换电路等。（用于连接主机与盘驱动器）

还包括数据缓存器、控制状态寄存器等。



硬盘存储器的简化逻辑结构

硬盘驱动器的逻辑结构



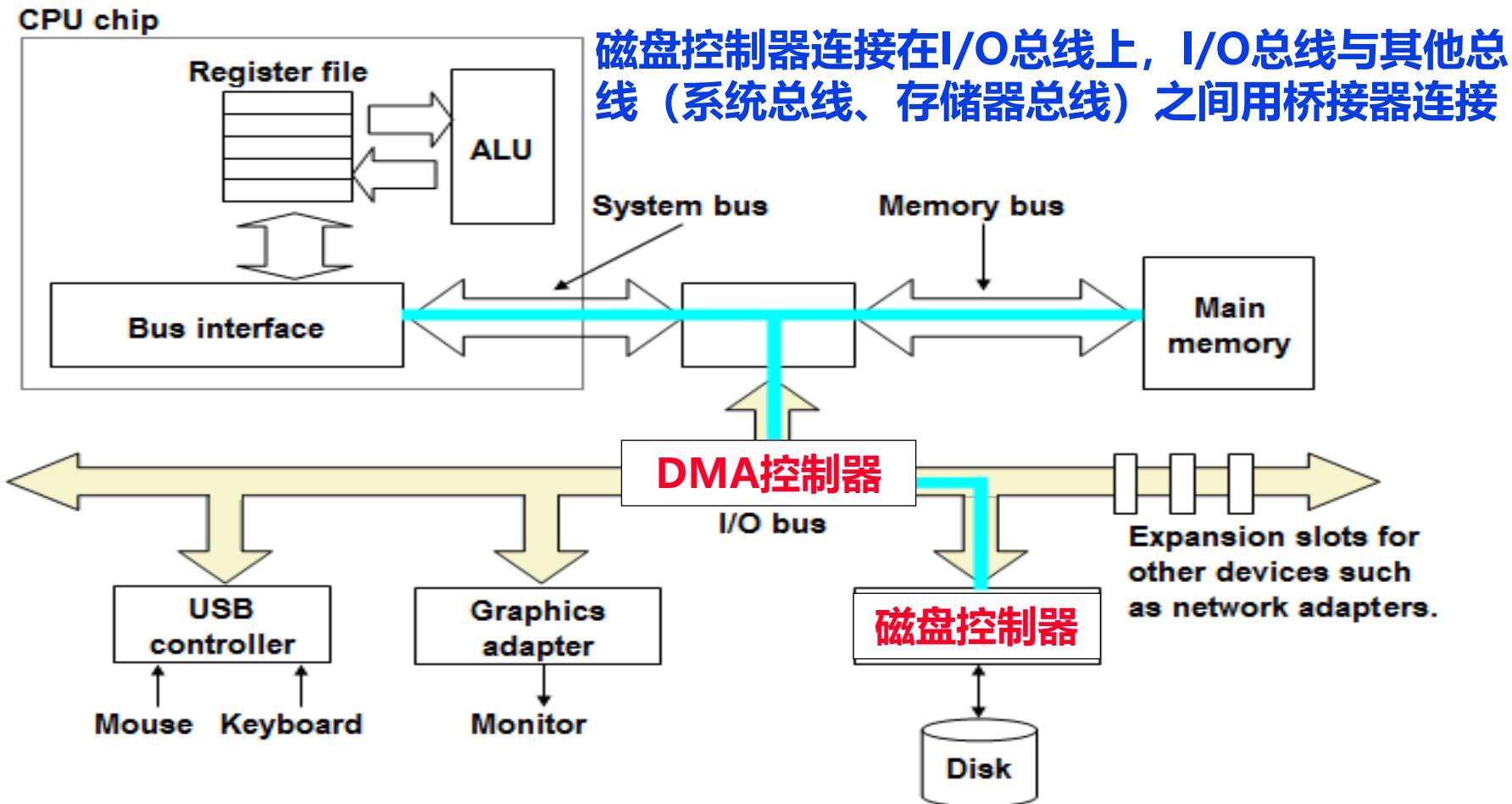
与磁盘控制器之间的接口

如何定位磁盘上的数据 (磁盘地址格式) ?

柱面(磁道)号、磁头 (盘面) 号、扇区号

操作过程? 寻道、旋转、读/写

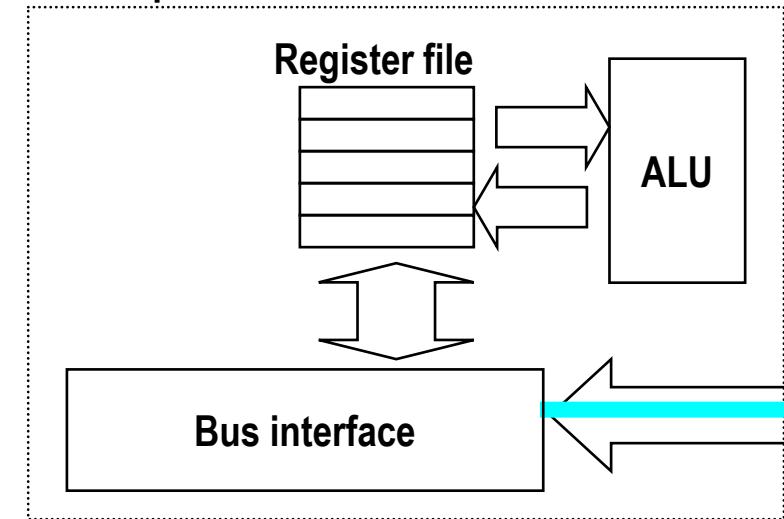
磁盘存储器的连接



磁盘的最小读写单位是扇区，因此，磁盘按成批数据交换方式进行读写，采用直接存储器存取（DMA, Direct Memory Access）方式进行数据输入输出，需用专门的DMA接口来控制外设与主存间直接数据交换，数据不通过CPU。通常把专门用来控制总线进行DMA传送的接口硬件称为**DMA控制器**

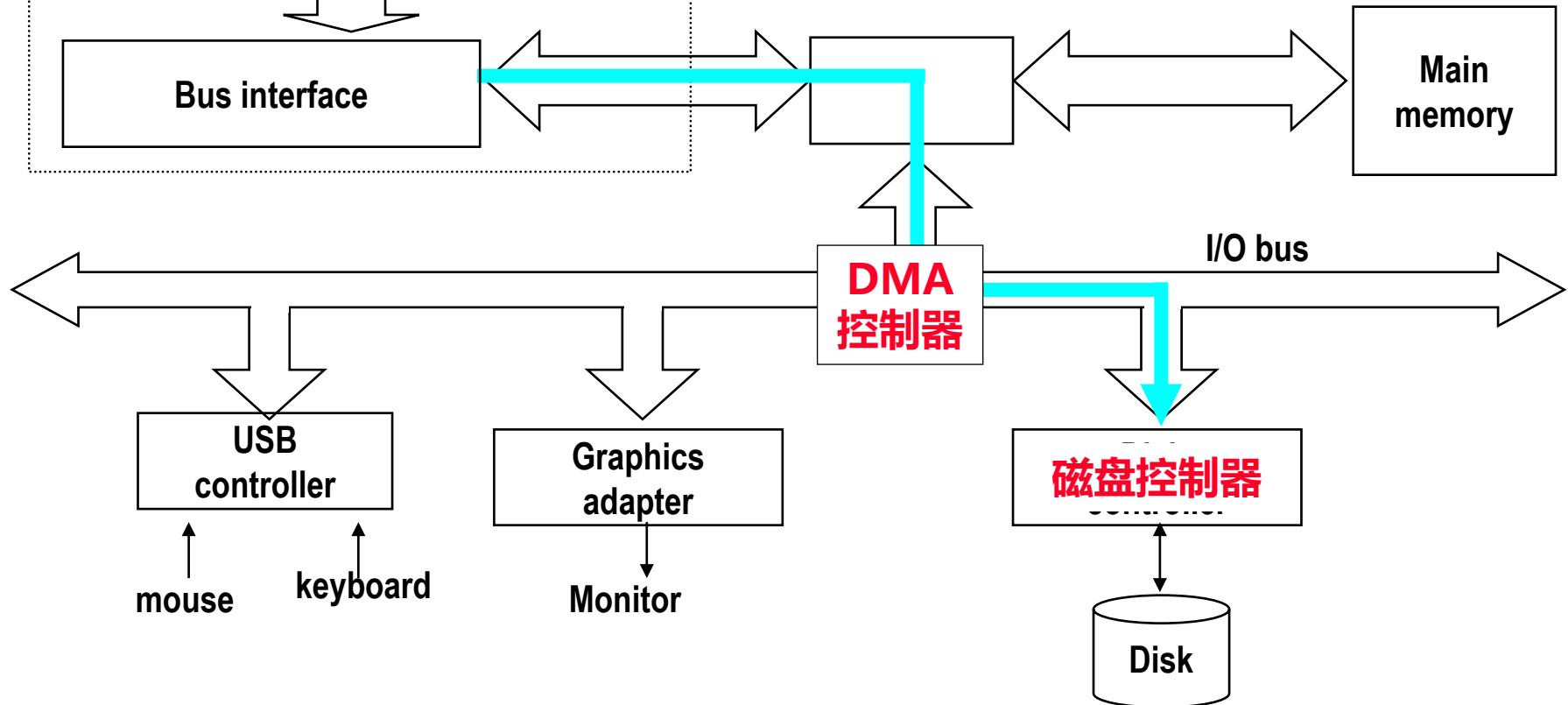
读一个磁盘扇区 - 第一步

CPU chip



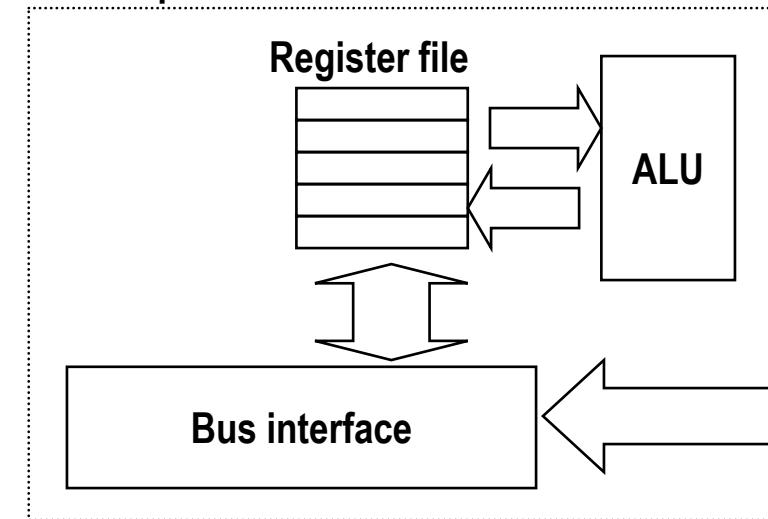
CPU对DMA控制器初始化：

读命令、磁盘逻辑块号、
主存起始地址、数据块大小
然后启动磁盘驱动器工作

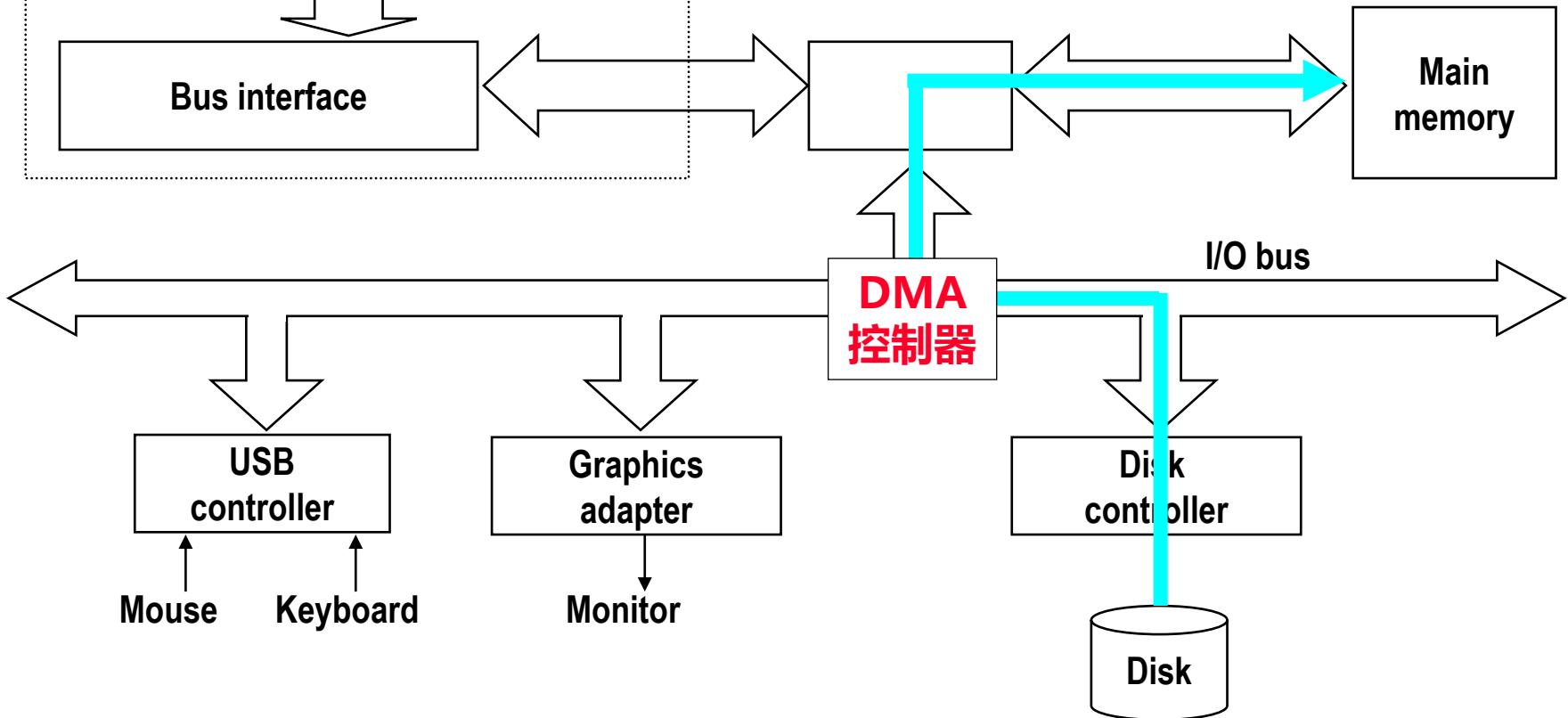


读一个磁盘扇区 - 第二步

CPU chip

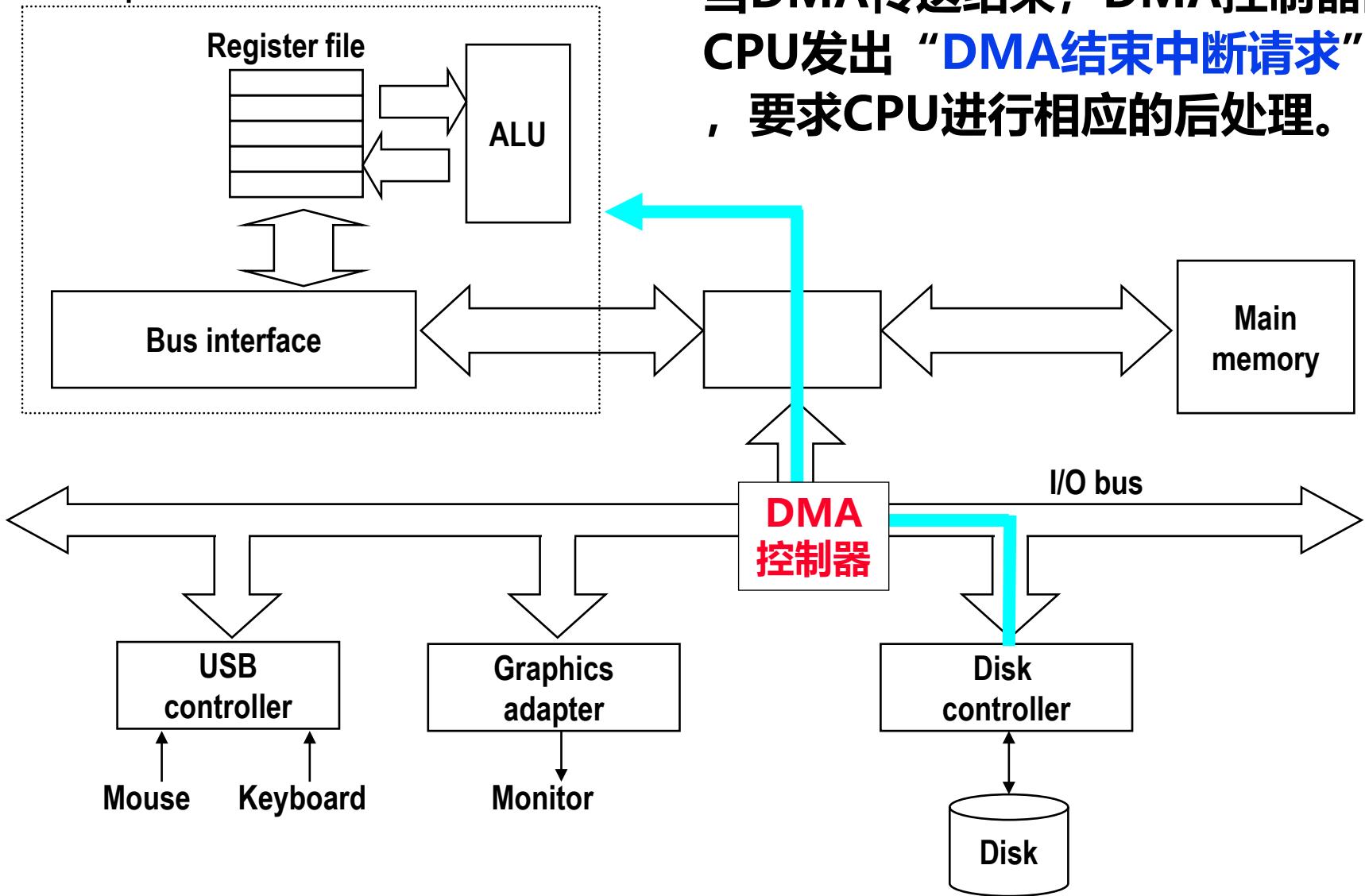


磁盘控制器读相应的扇区，并按
DMA方式把数据送主存



读一个磁盘扇区 - 第三步

CPU chip



固态硬盘（不作要求）

- 固态硬盘（Solid State Disk，简称SSD）也被称为**电子硬盘**。
- 它并不是一种磁表面存储器，而是一种使用**NAND闪存**组成的外部存储系统，与U盘并没有本质差别，只是容量更大，存取性能更好。
- 电信号的控制使得固态硬盘的内部**传输速率远远高于常规硬盘**。
- 其接口规范和定义、功能及使用方法与传统硬盘完全相同，在产品外形和尺寸上也与普通硬盘一致。目前接口标准上使用USB、SATA和IDE，因此SSD是通过**标准磁盘接口与I/O总线互连的**。
- 在SSD中有一个**闪存翻译层**，它将来自CPU的逻辑磁盘块读写请求翻译成对底层SSD物理设备的读写控制信号。因此，这个**闪存翻译层**相当于**磁盘控制器**。
- 闪存的**擦写次数有限**，所以频繁擦写会降低其写入使用寿命。

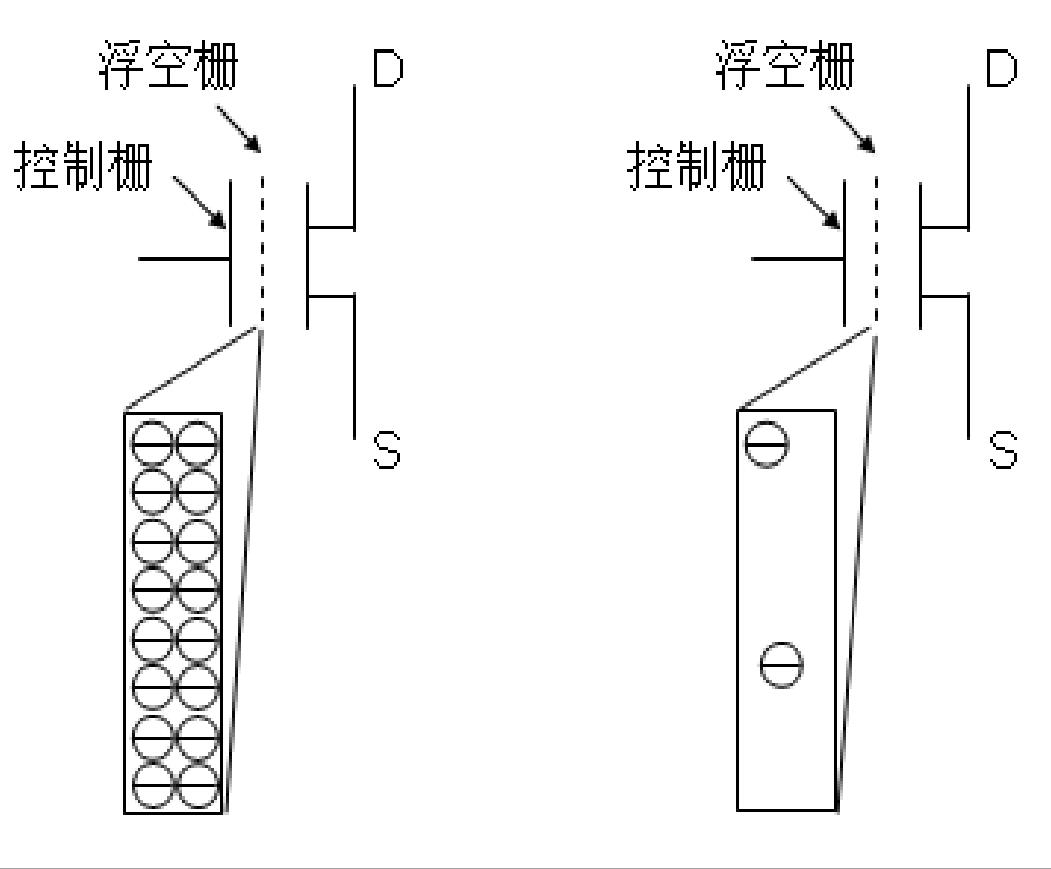
固态硬盘（不作要求）

- 它用闪存颗粒代替了磁盘作为存储介质，利用闪存的特点，以块写入和抹除的方式进行数据的写入。
- 写操作比读操作慢。顺序读比顺序写大致快一倍，随机读比随机写大致快10倍。
- 随机读写时延比硬盘低两个数量级（随机读约几十微秒，随机写约几百微秒）。
- 一个闪存芯片由若干个区块组成，每个区块由若干页组成。通常，页大小为512B~4KB，每个区块由32~128个页组成，因而区块大小为16KB~512KB，数据可以按页为单位进行读写。
- 当需要写某页信息时，必须先对该页所在的区块进行擦除操作。一旦一个区块被擦除过，区块中的每一页就可以直接再写一次。若某一区块进行了大约100 000次重复写之后，就会被磨损而变成坏的区块，不能再被使用。因此，闪存翻译层中有一个专门的均化磨损（wear leveling）逻辑电路，试图将擦除操作平均分布在所有区块上，以最大限度地延长SSD的使用寿命。由此可见，对于物理区块的写优化是由SSD中的硬件实现的，无需软件进行写优化。

Skip

闪存 (Flash Memory)

Flash 存储元：

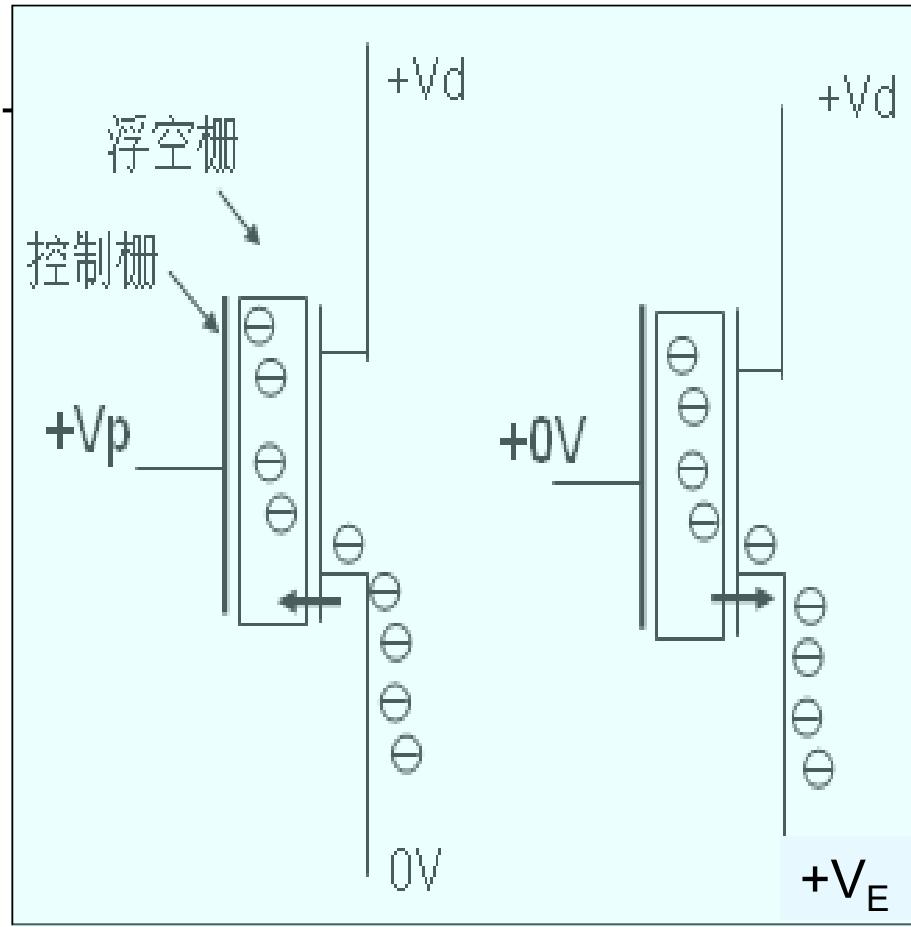


(a)“0”状态

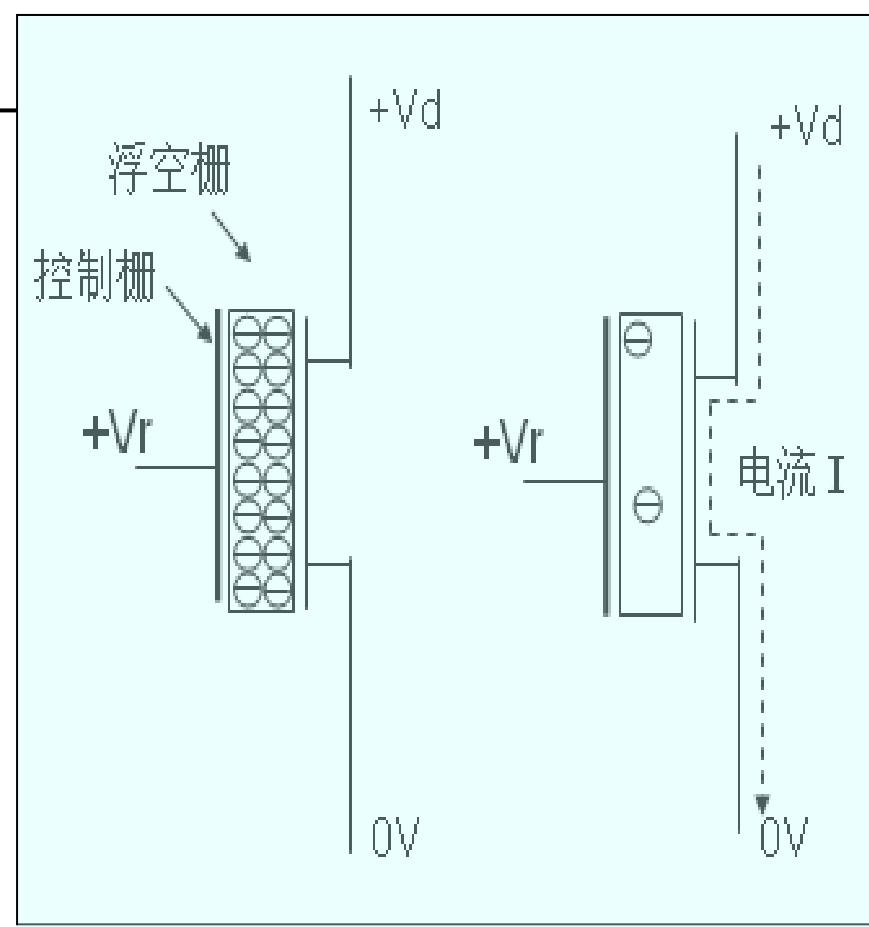
(b) “1”状态

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) 编程:写“0” (b) 擦除:写“1”



(a) 读“0” (b) 读“1”

有三种操作: 擦除 (写1) 、编程 (写0) 、读取

读快、写慢!

{ 写入: 快擦 (所有单元为1) -- 编程 (需要之处写0)

[BACK](#)

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

◦ 分以下六个部分介绍

- 第一讲：存储器概述
- 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：磁盘存储器
- 第四讲：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
- 第五讲：虚拟存储器
 - 虚拟地址空间、虚拟存储器的实现
- 第六讲：IA-32/Linux中的地址转换
 - 逻辑地址到线性地址的转换
 - 线性地址到物理地址的转换

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，SSD、硬盘

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

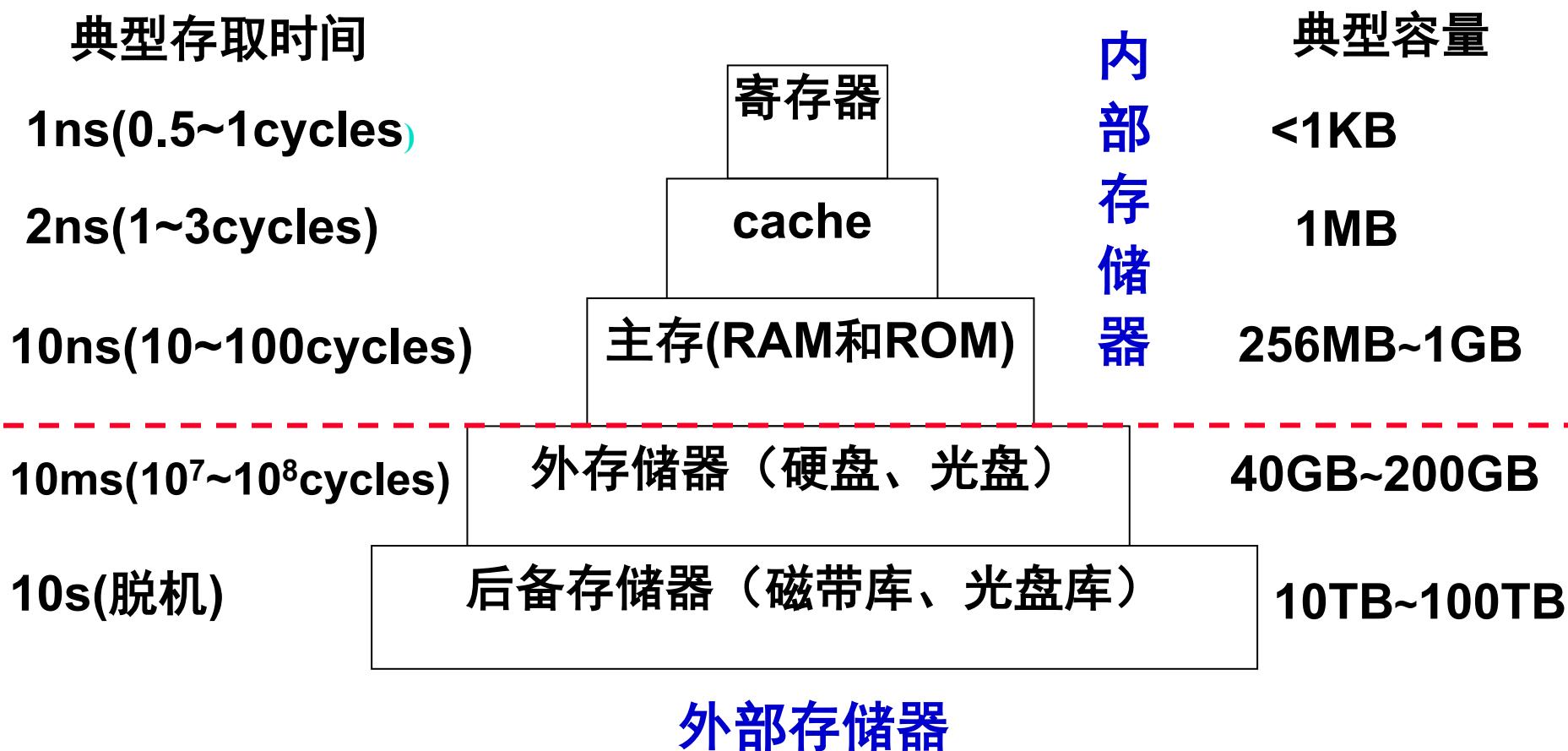
* non-volatile

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！

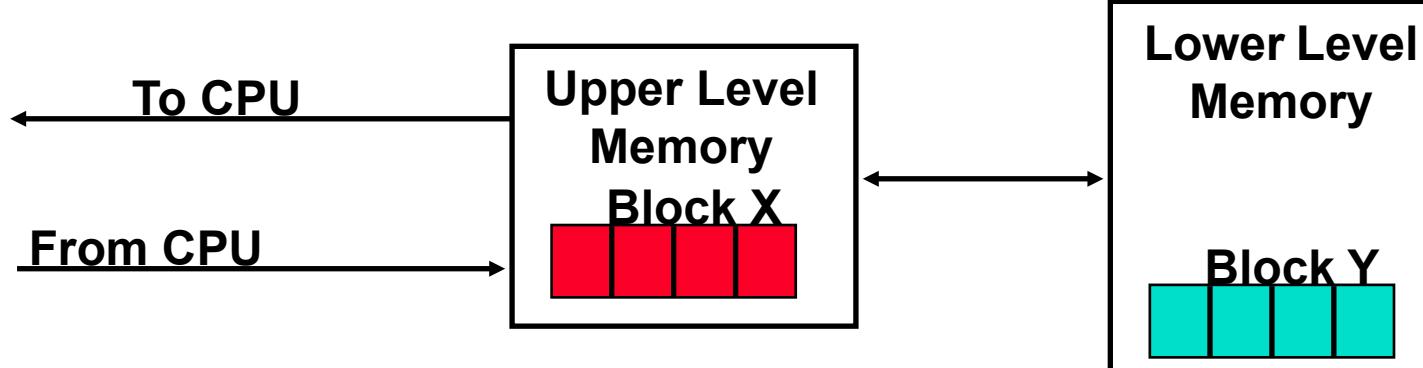
采用分层存储结构来构建计算机的存储体系！

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

Block: 传送单位, 比所需数据块大得多, 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?

° **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

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!

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

对应的汇编语言程序

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

```
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 累计在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 累计结果保存至地址v
```

每条指令4个字节；每个数组元素4字节

指令和数组元素在内存中均连续存放

sum, ap ,i, t 均为通用寄存器；A, V为内存地址

主存的布局:

0x0FC	I0	指
0x100	I1	令
0x104	I2	
0x108	I3	
0x10C	I4	
0x110	I5	
0x114	I6	
	...	
0x400	a[0]	A
0x404	a[1]	数
0x408	a[2]	据
0x40C	a[3]	
0x410	a[4]	
0x414	a[5]	
	...	
0x7A4		V

程序的局部性原理举例1

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
→0x120 (I9) n次

数据：只有数组在主存中：

0x400→0x404→0x408
→0x40C→.....→0x7A4

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。

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

主存的布局：

0x0FC	I0	指令
0x100	I1	
0x104	I2	
0x108	I3	
0x10C	I4	
0x110	I5	
0x114	I6	
...		
0x400	a[0]	A
0x404	a[1]	
0x408	a[2]	数
0x40C	a[3]	
0x410	a[4]	据
0x414	a[5]	
...		
0x7A4		V

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？

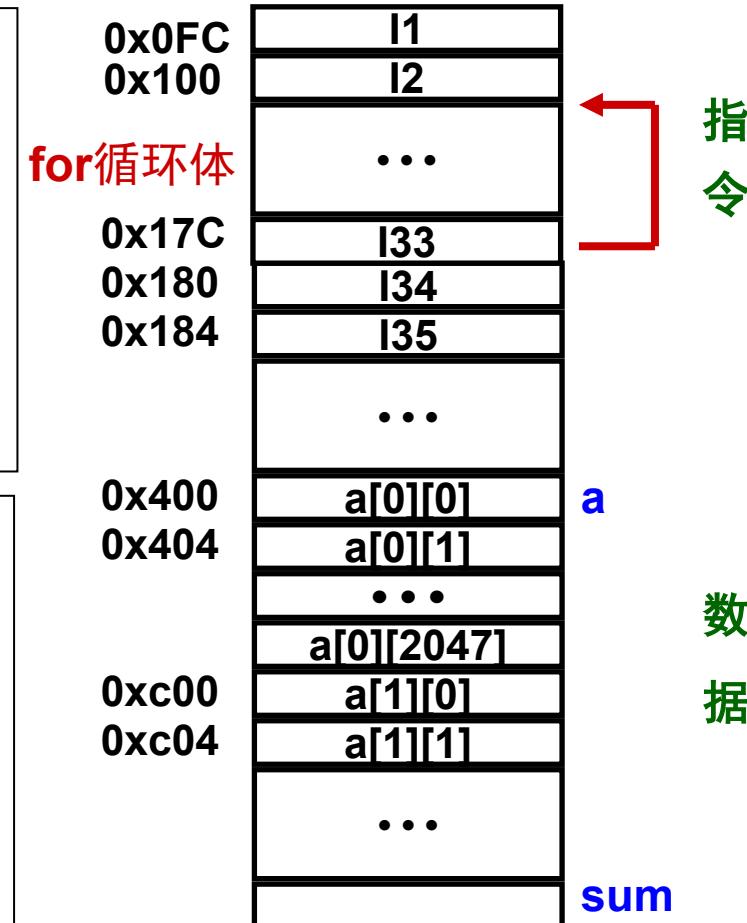
程序段A：

```
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

程序的局部性原理举例2

程序段A的时间局部性和空间局部性分析

(1) 数组a: 访问顺序为 $a[0][0], a[0][1], \dots, a[0][2047]; a[1][0], a[1][1], \dots, a[1][2047]$;

....., 与存放顺序一致, 故空间局部性好!

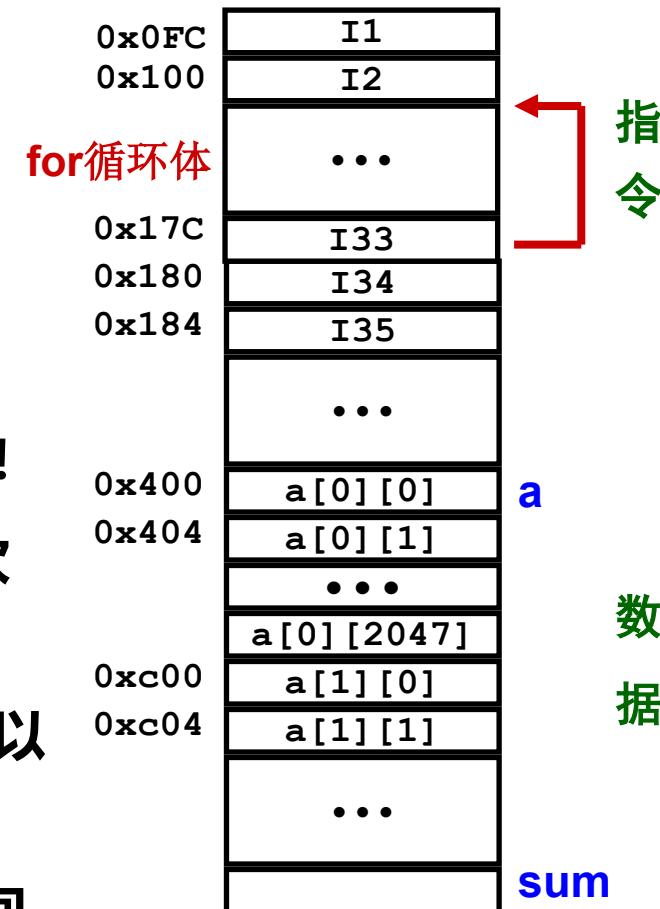
因为每个 $a[i][j]$ 只被访问一次, 故时间局部性差!

(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: 循环体内指令按序连续存放, 所以空间局部性好!

循环体被连续重复执行 2048×2048 次, 所以时间局部性好!

实际上 优化的编译器使循环中的sum分配在寄存器中, 最后才写回存储器!



程序的局部性原理举例2

程序段B的时间局部性和空间局部性分析

(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个单元，若交换单位小于2KB，则没有空间局部性！

(时间局部性差，同程序A)

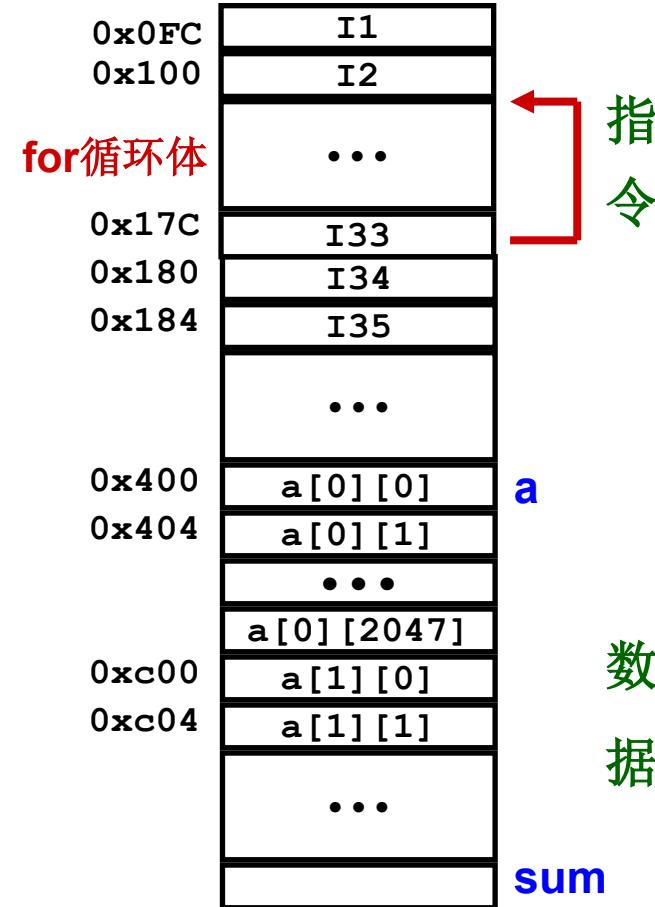
(2) 变量sum：(同程序A)

(3) for循环体：(同程序A)

实际运行结果(2GHz Intel Pentium 4):

程序A：59,393,288 时钟周期

程序B：1,277,877,876 时钟周期

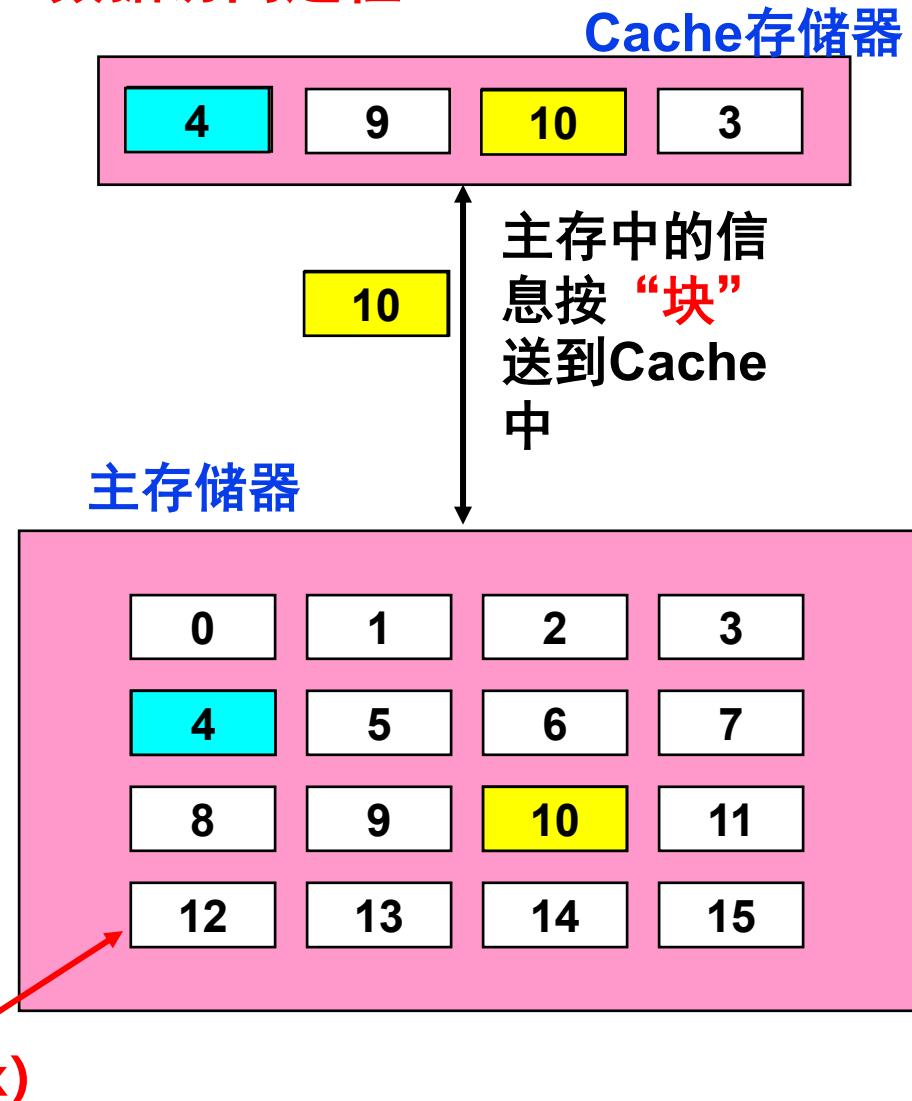


程序A比程序B快
21.5 倍!!

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。
- 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。

数据访问过程：

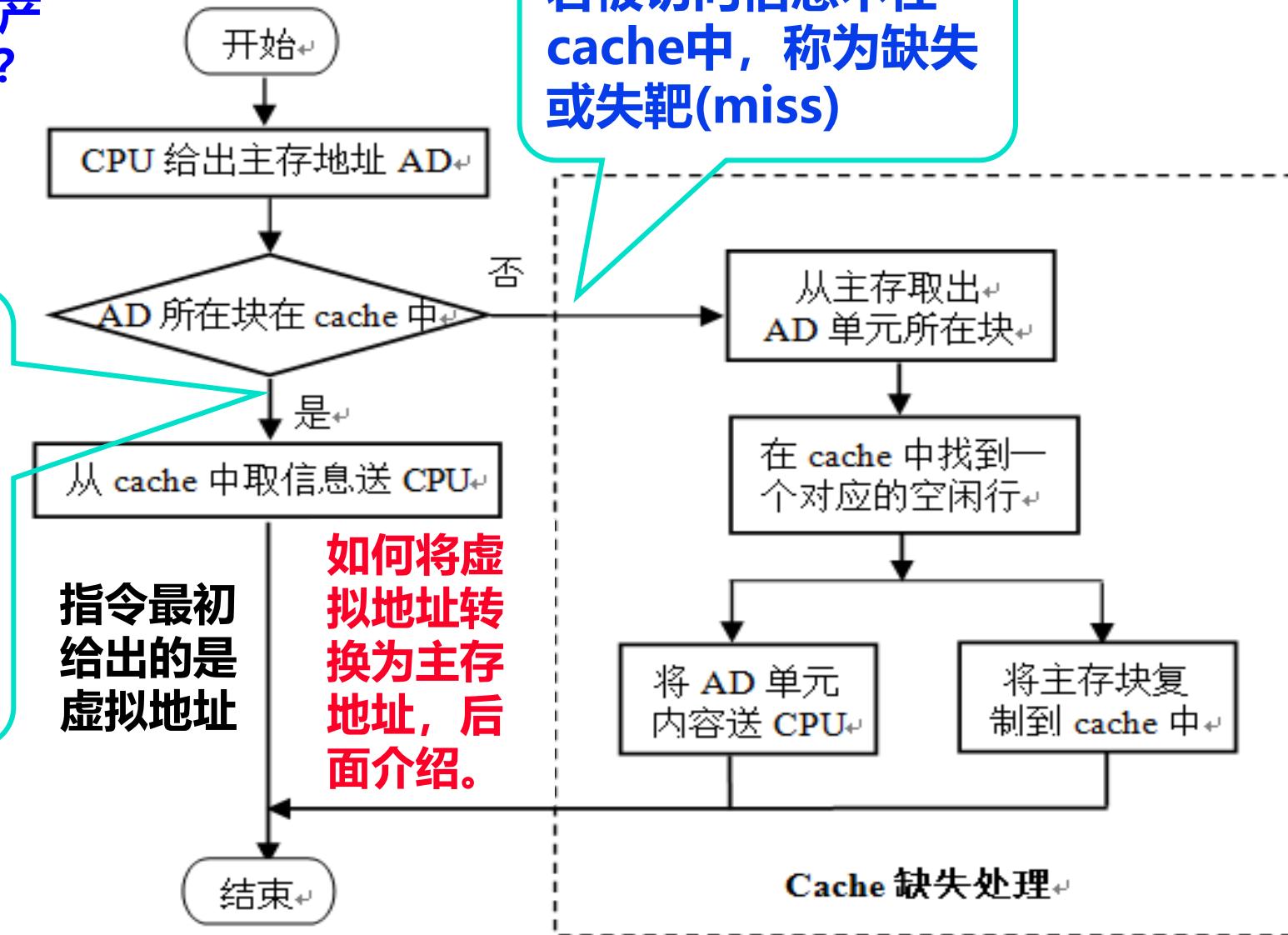


Cache 的操作过程

问题：什么情况下，CPU产生访存要求？

执行指令时！

若被访问信息在 cache 中，称为命中 (hit)



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?

- 把访问的主存块取到Cache中时，该放到Cache的何处?
- Cache行比主存块少，因而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

The Simplest Cache: Direct Mapped Cache

◦ Direct Mapped Cache (直接映射Cache举例)

- 主存每一块只能映射到固定的Cache行（槽）

- 也称模映射(Module Mapping)

- 映射关系为：

$\text{Cache行号} = \text{主存块号} \bmod \text{Cache行数}$

举例： $4 = 100 \bmod 16$ (假定Cache共有16行)

(说明：主存第100块应映射到Cache的第4行中。)

举例：书和书架的关系

块（行）都从0开始编号

◆ 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。

假定主存块为512B。

直接映射Cache组织示意图

Cache大小：

$$2^{13}B = 8KB = 16\text{行} \times 512B/\text{行}$$

主存大小：

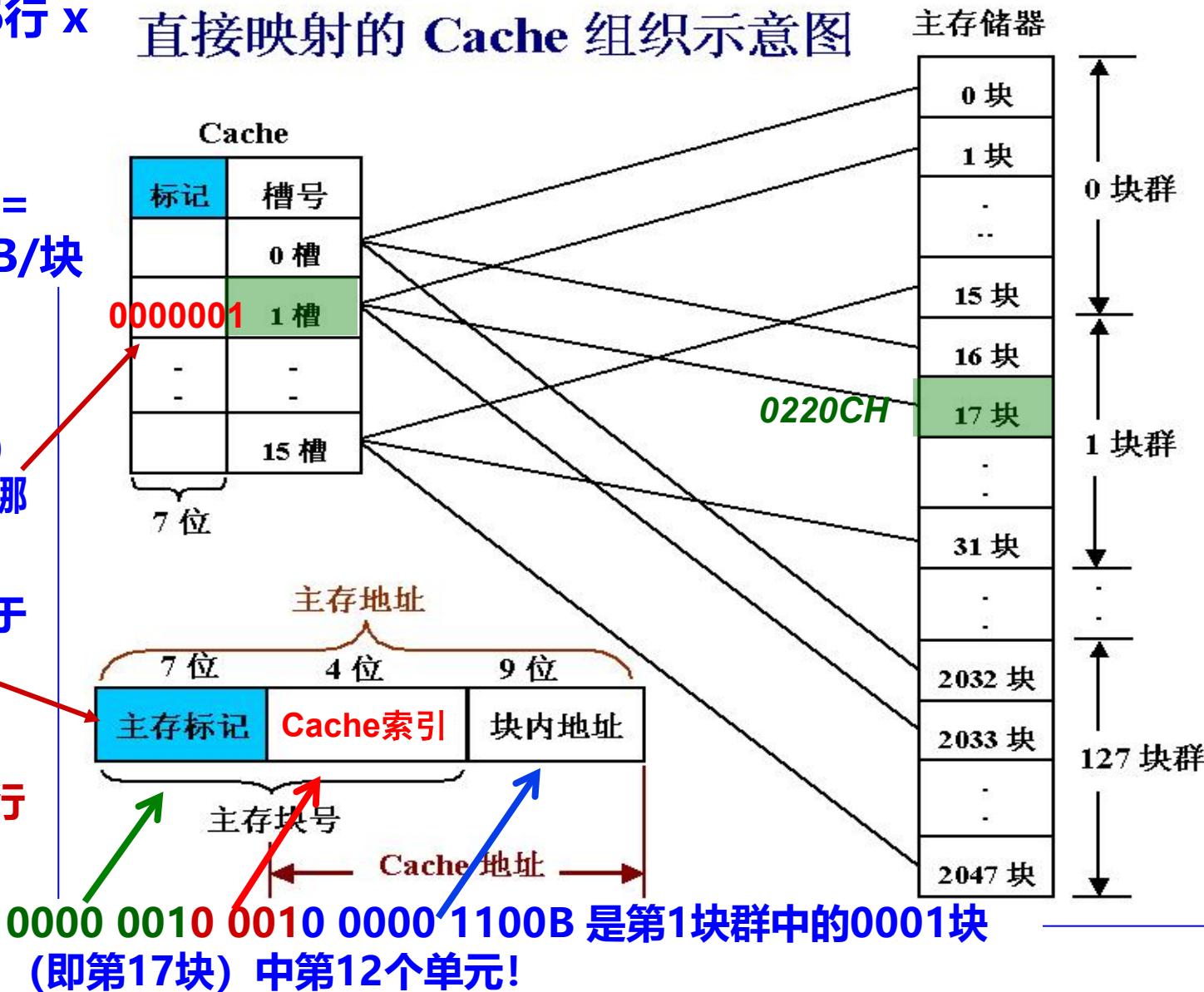
$$2^{20}B = 1024KB = 2048\text{块} \times 512B/\text{块}$$

Cache标记(tag)
指出对应行取自哪
个主存块群

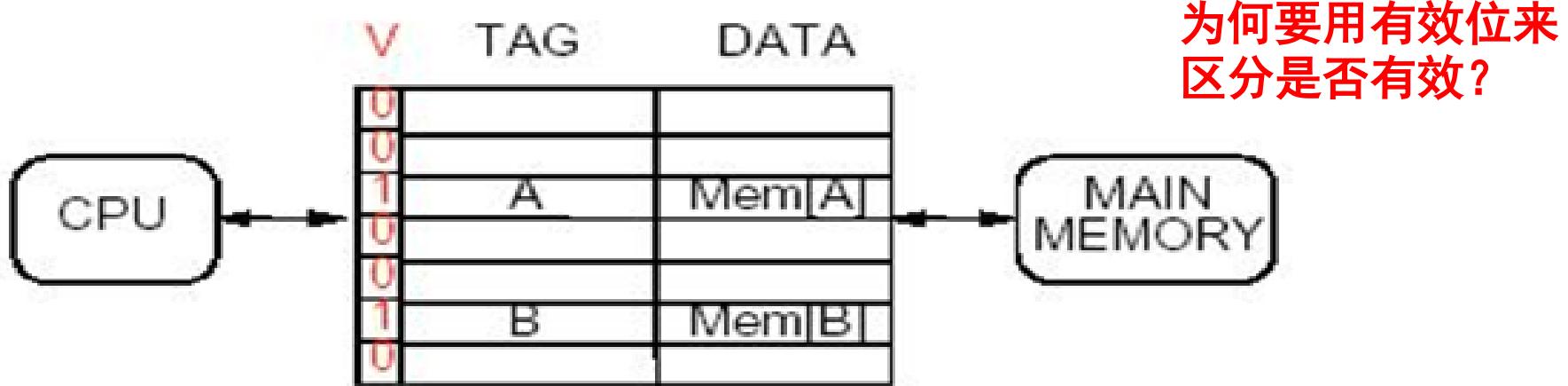
指出对应地址位于
哪个块群

例：如何对
0220CH单元进行
访问？

直接映射的 Cache 组织示意图



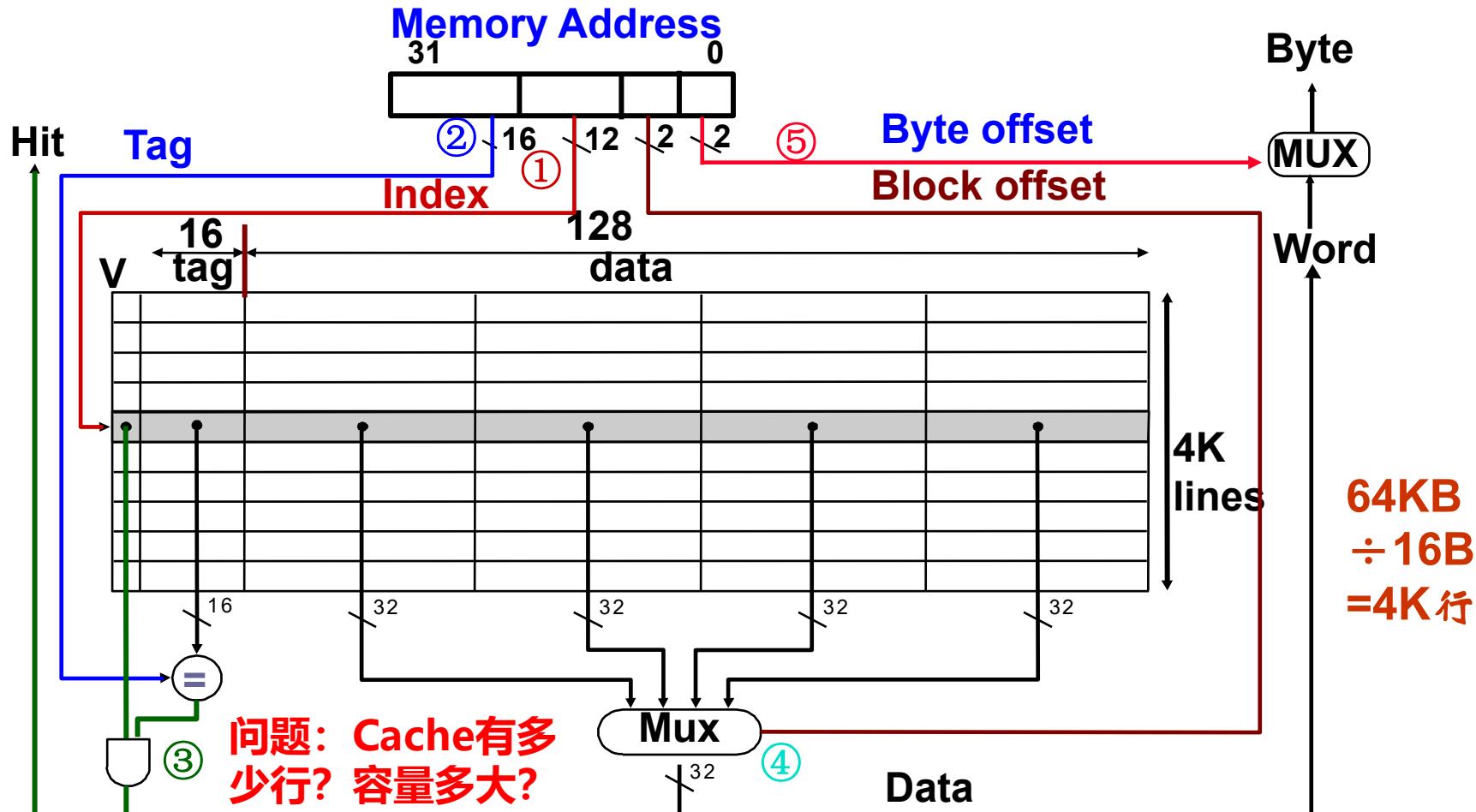
有效位 (Valid Bit)



- 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。



$$\begin{aligned} & 64KB \\ & \div 16B \\ & = 4K \text{ 行} \end{aligned}$$

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?

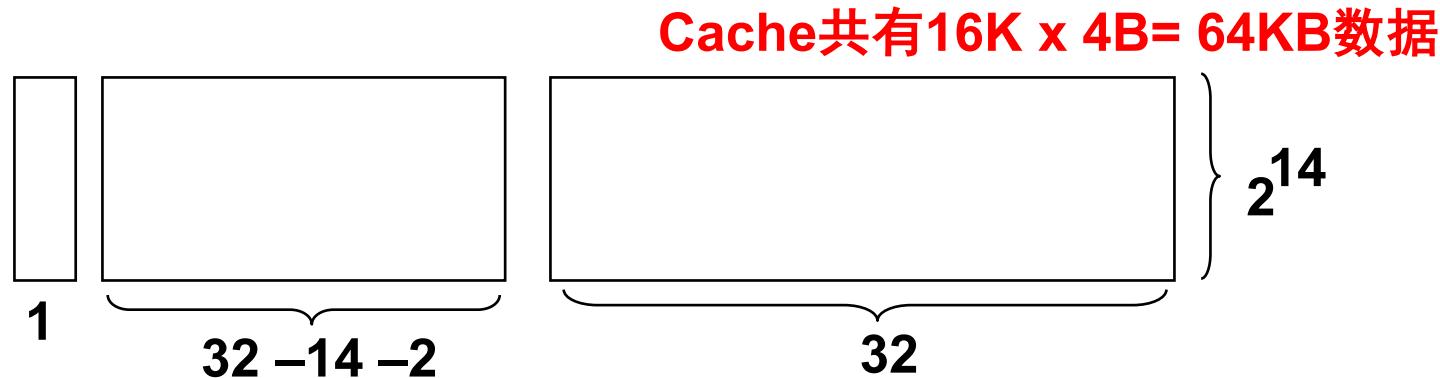
地址1200对应存放在第11行。因为： $[1200/16=75] \text{ module } 64 = 11$

$$1200 = 1024 + 128 + 32 + 16 = 0\dots01 \boxed{001011} 0000 \text{ B}$$

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：



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

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

若块大小为 2^m 个字呢？ $2^{14} \times (2^m \times 32 + (32-14-2-m)+1)$ [BACK](#)

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 \times 512 字 / 行

主存大小: 2^{20} 字
= 1024K 字 = 2048 块 \times 512 字 / 块

Cache 标记 (tag) 指出对应行应取自哪个主存块

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!

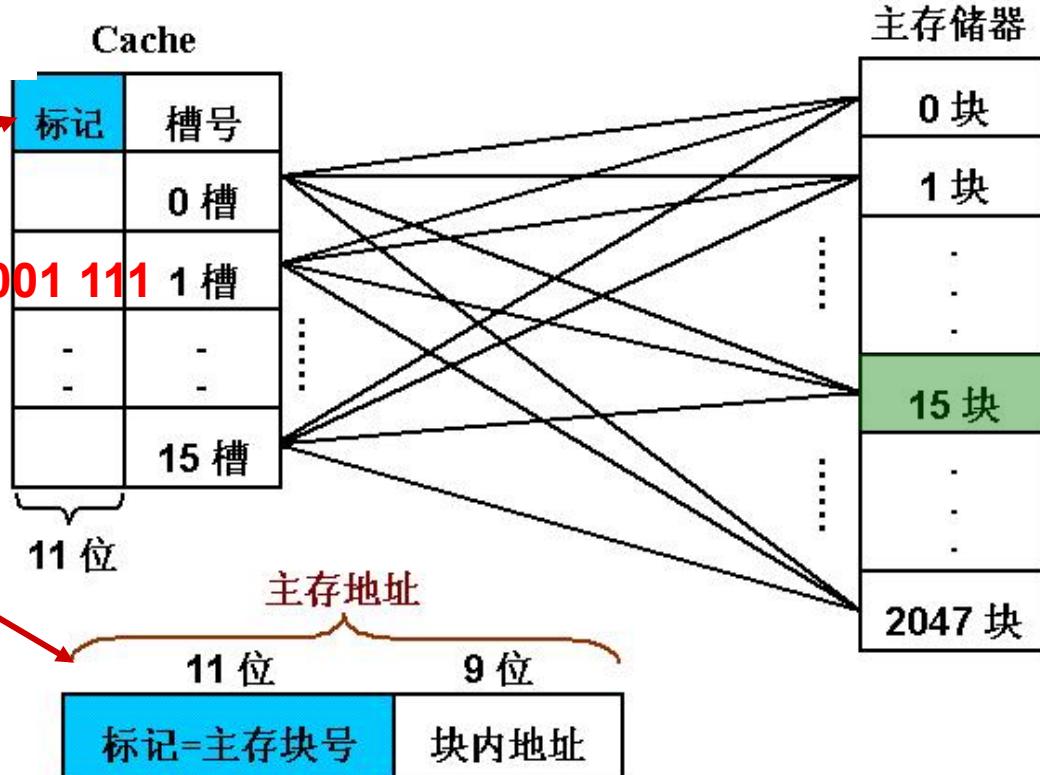
全相联映射 Cache 组织示意图

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!

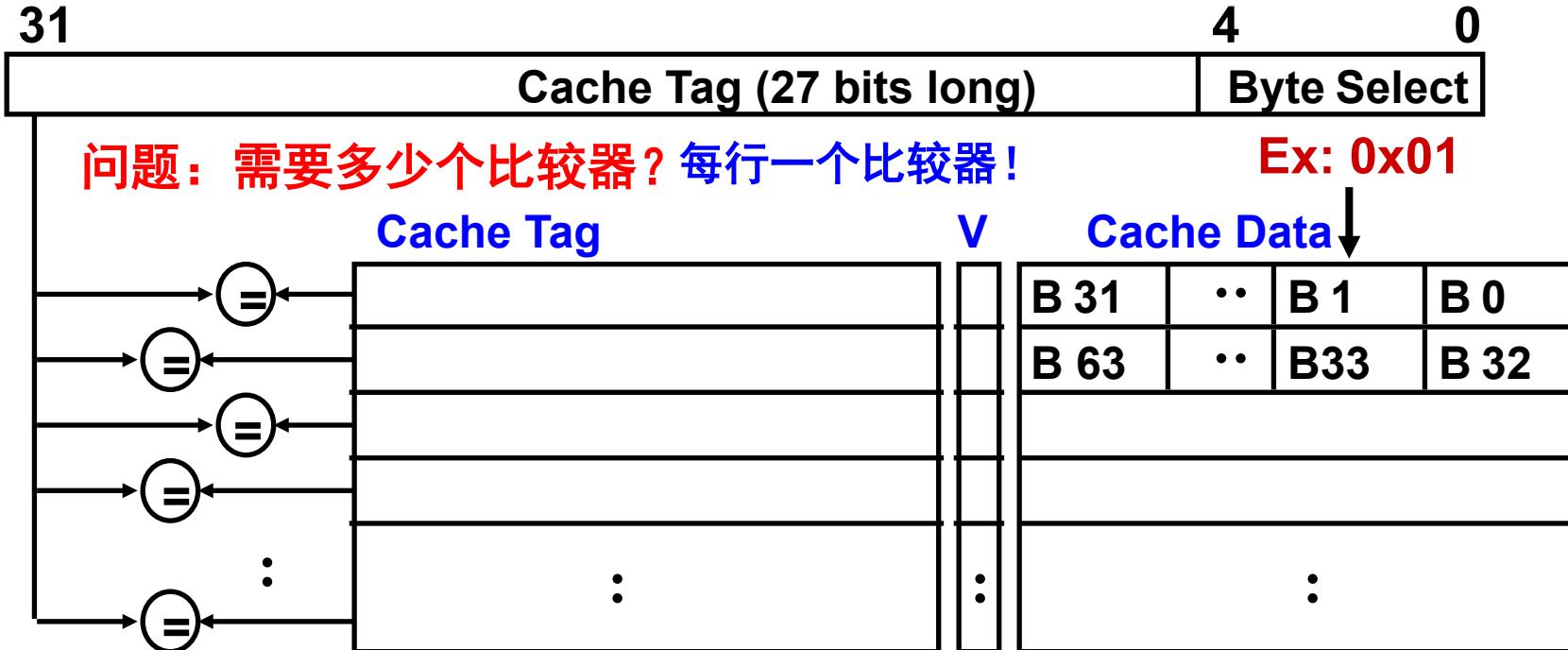
全相联映射的 Cache 组织示意图



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行的标志
 - ° 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
- 缺点：比较器个数多，比较器位数多！



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x2行/组x512字/行

$$4 = 100 \bmod 8$$

(主存第100块应映射到Cache的第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，变为相联映射；当每组只有一个槽时，变为直接映射。
- 每组2或4行（称为2-路或4-路组相联）较常用。通常每组4行以上很少用。在较大容量的L2 Cache和L3 Cache中使用4-路以上。

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

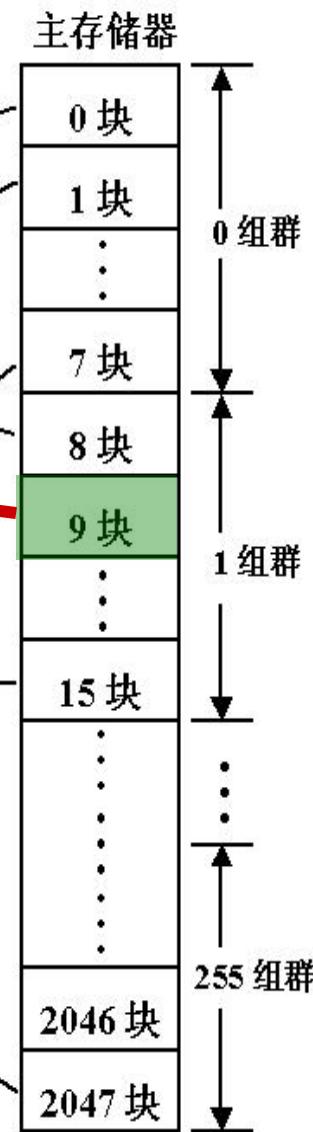
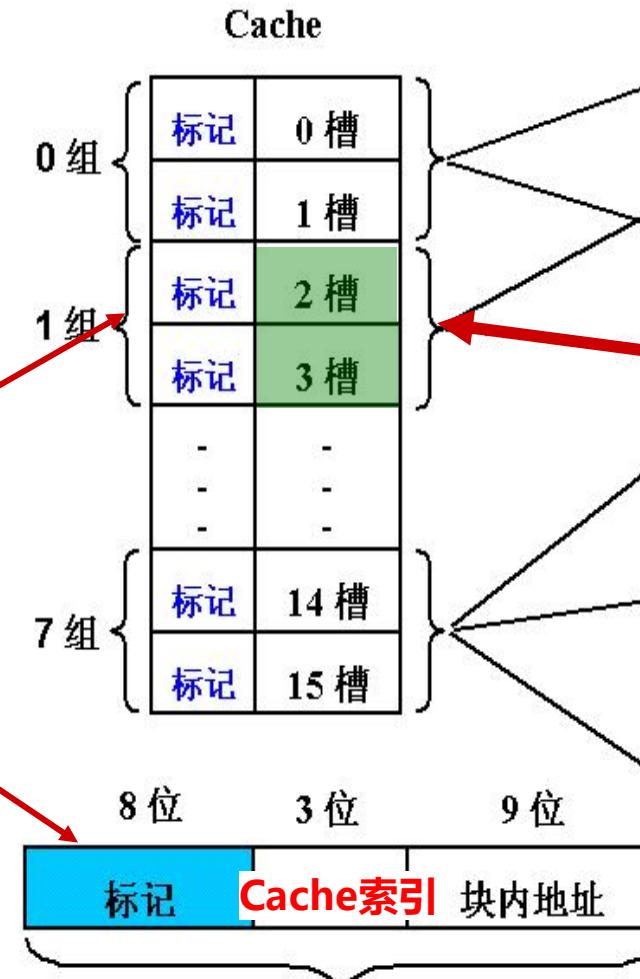
指出对应地址位于哪
个主存组群中

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。

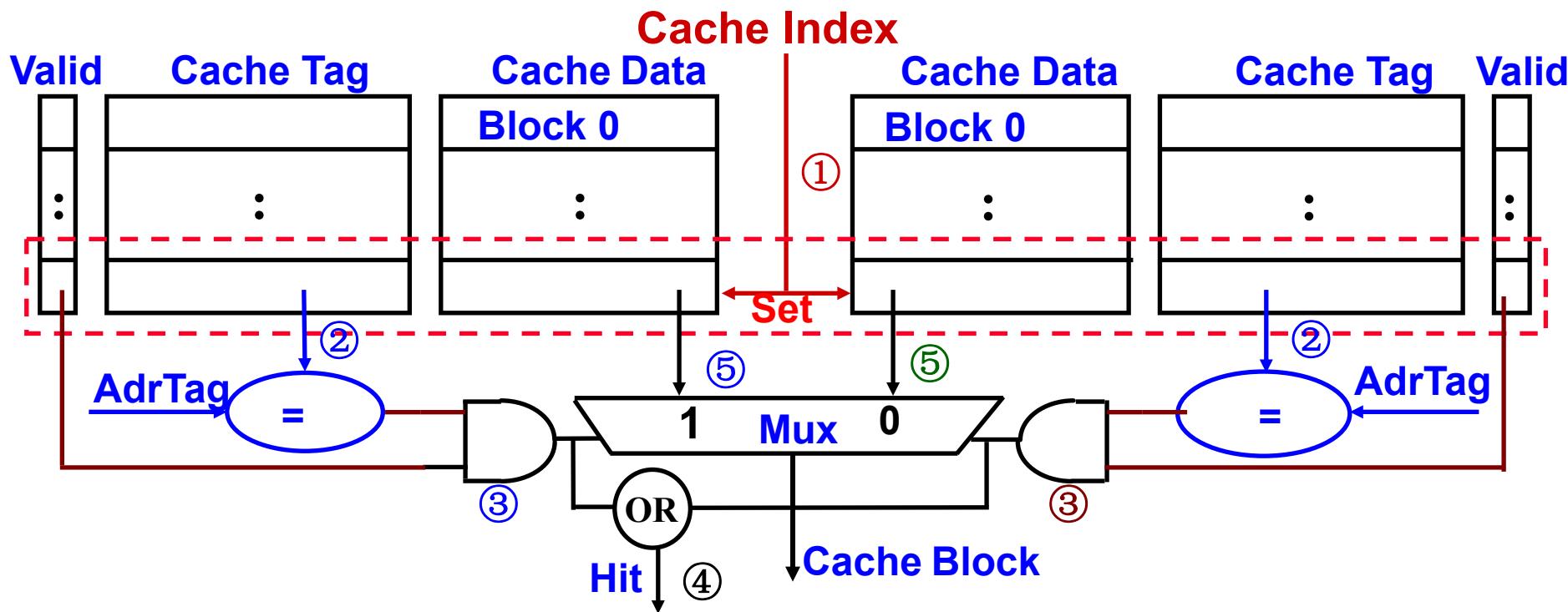
组相联映射的 Cache 组织示意图



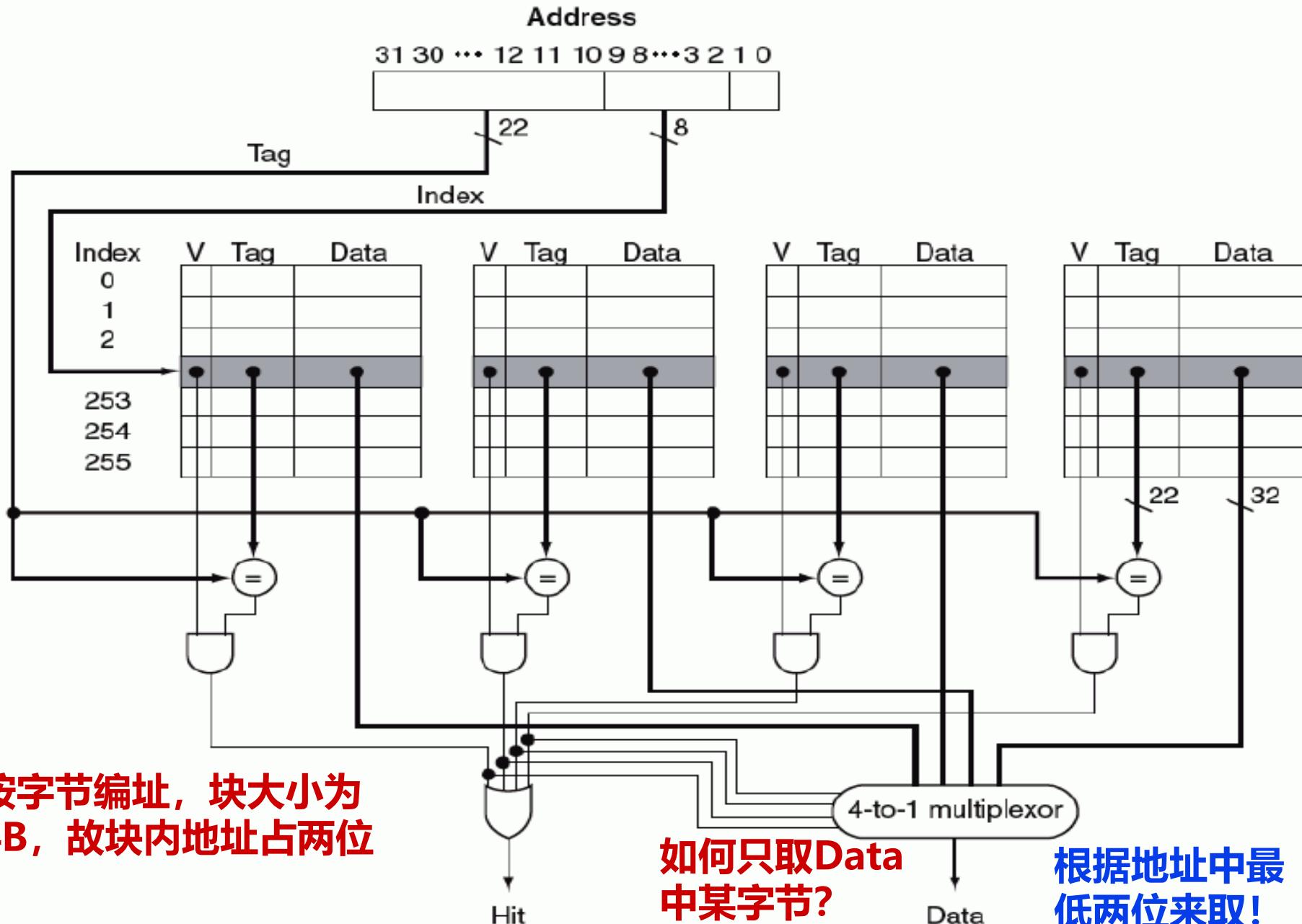
将主存地址标记和对应Cache组中每个
Cache标记进行比较!

例1：A Two-way Set Associative Cache

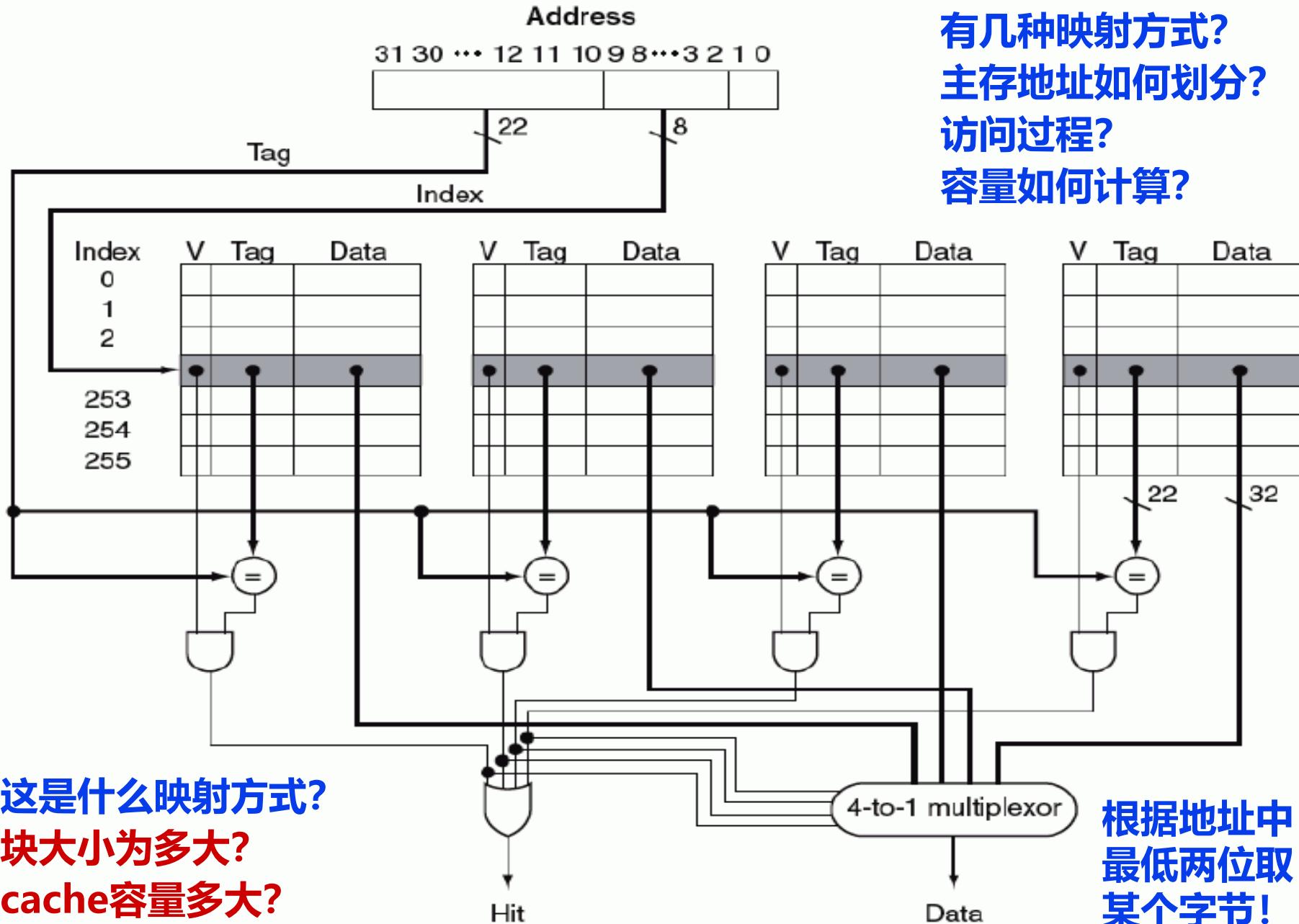
- N-way set associative
 - N 个直接映射的行并行操作
- Example: Two-way set associative cache
 - Cache Index 选择一个Cache行集合Set (共2行)
 - 对集合中的两个Cache行的Tag并行进行比较
 - 根据比较结果确定信息在哪个行，或不在Cache中



例2：4-路组相联方式



复习：cache和主存映射方式

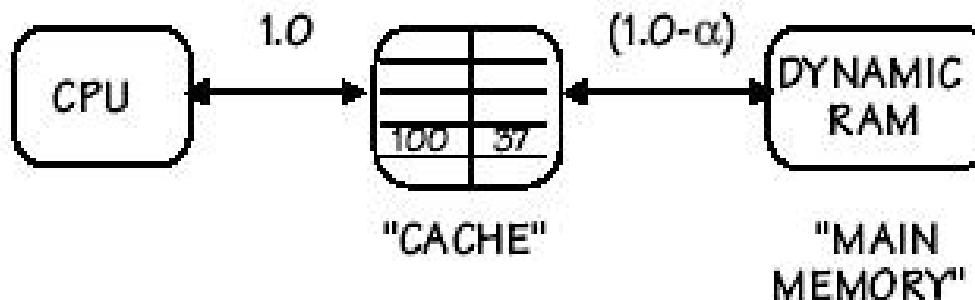


命中率、缺失率、缺失损失

- ° 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)

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.

$(1-\alpha)$ MISS RATIO: Remaining references.

$$t_{\text{avg}} = \underline{\alpha t_c} + (1-\alpha)(\underline{t_c + t_m}) = t_c + (1-\alpha)t_m$$

- 2) Transparency (compatibility, programming ease)

Cache对程序员来说是透明的，以方便编程！

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

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?

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)

SKIP

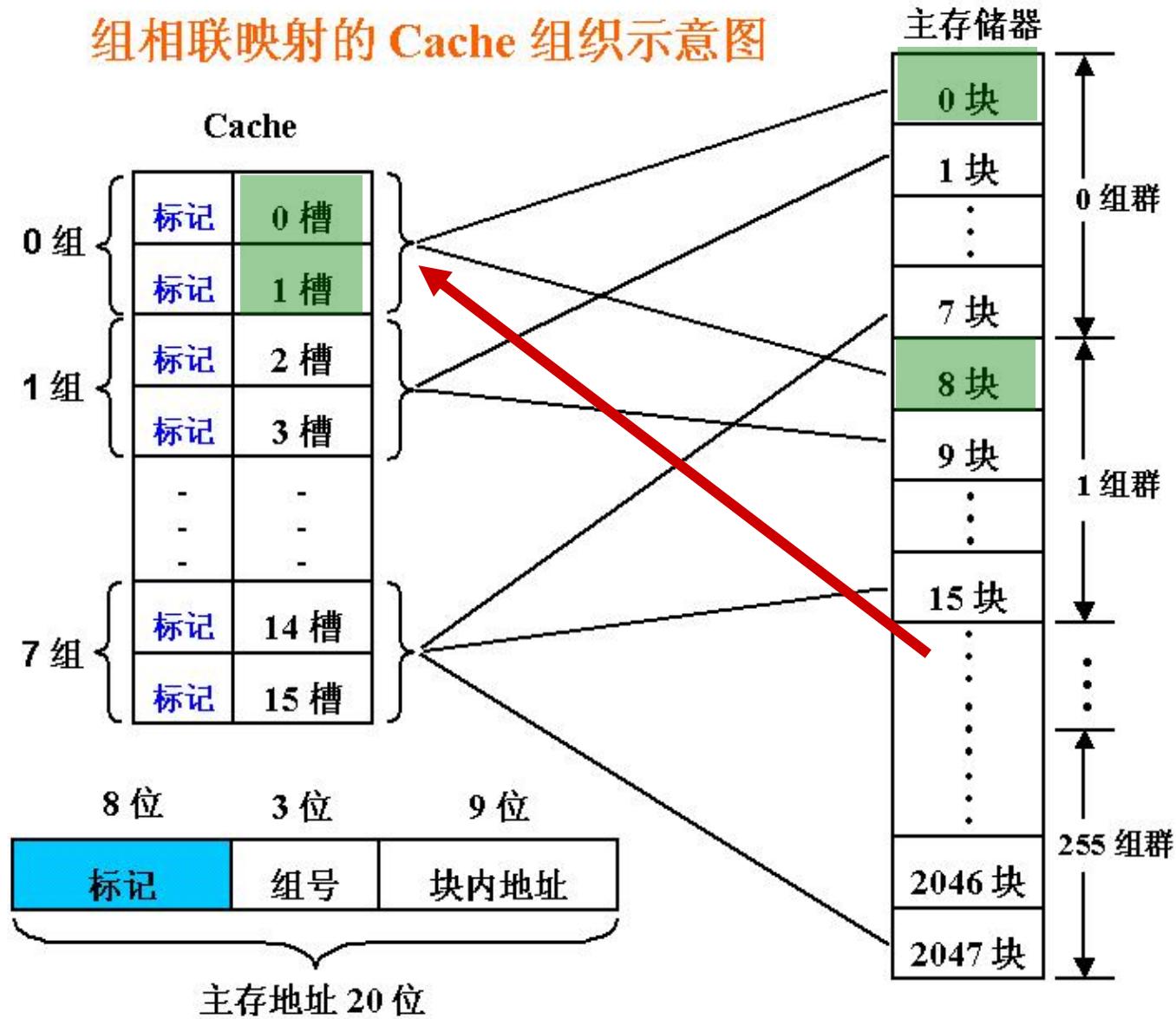
等等

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块和第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？

组相联映射的 Cache 组织示意图



[BACK](#)

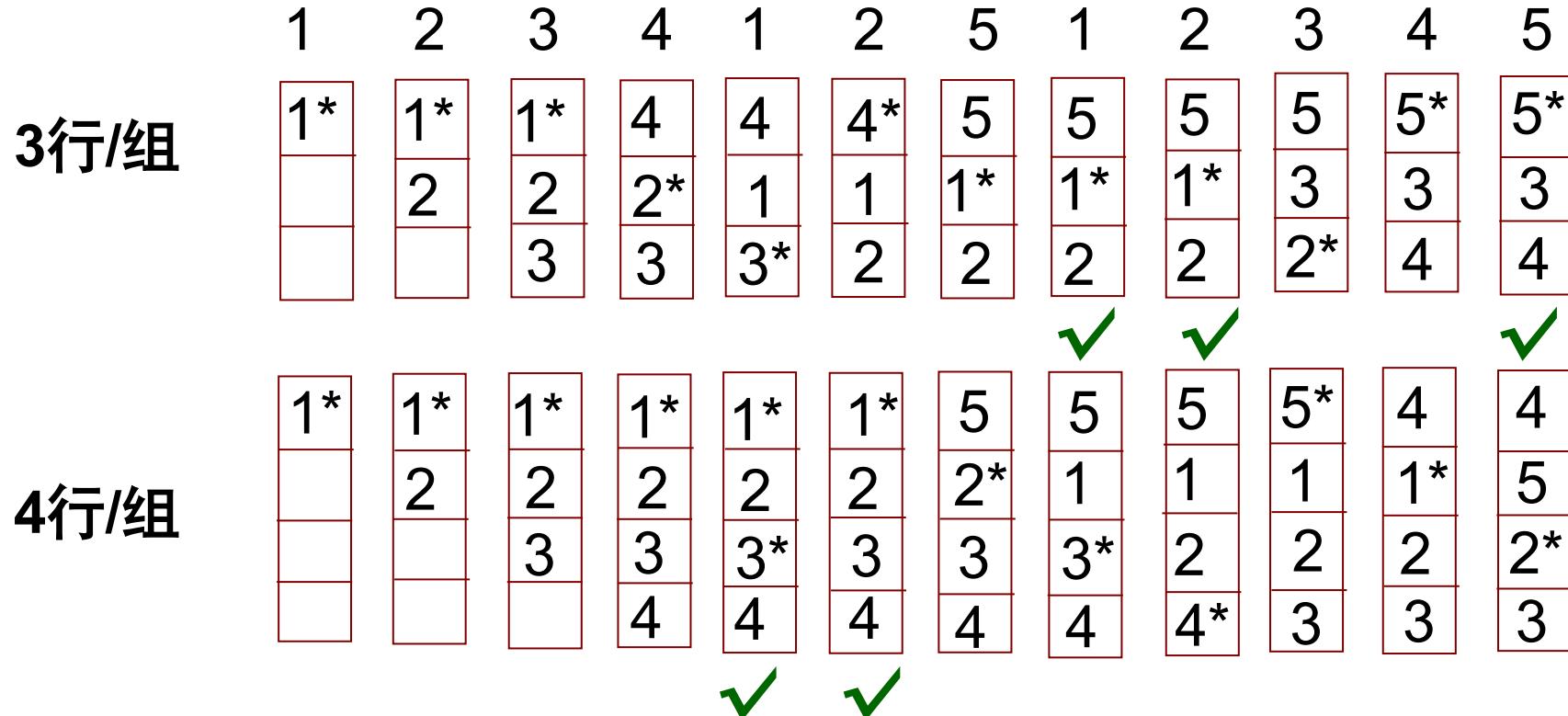
替换算法-先进先出 (FIFO)

- 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



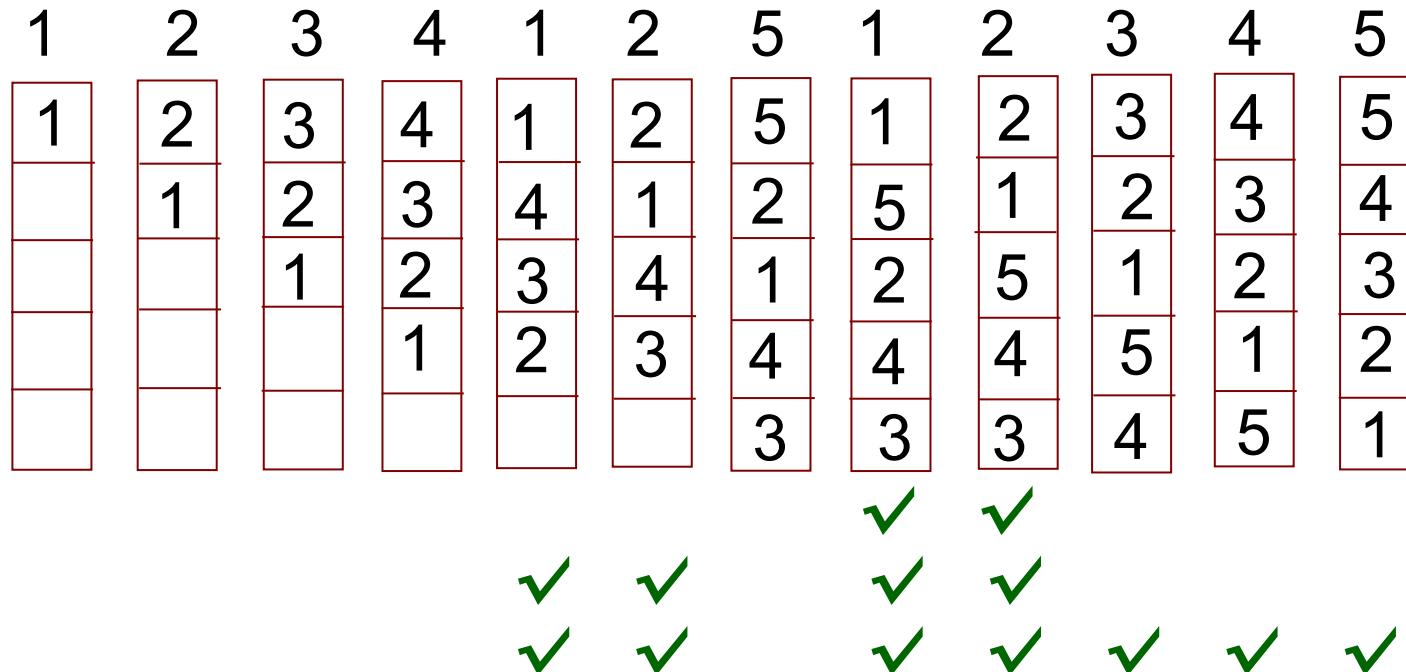
由此可见，FIFO不是一种栈算法，即命中率并不随组的增大而提高。

替换算法-最近最少用(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。



LRU是一种**栈算法**其命中率随组的增大而提高

LRU实现时不能移动块，而是给每个cache行设定一个计数器，以记录块的使用情况，称**LRU位**

替换算法-最近最少用

通过计数值 (LRU位) 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。

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

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32Kx16位，且有一个数据区为4K字的4路组相联Cache，主存块大小为64字。假定Cache开始为空，处理器顺序地从存储单元0、1、...、4351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 × 64字 / 块

Cache：4K字=16组 × 4行 / 组 × 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。

举例

	第0行	第1行	第2行	第3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 $(43520 - 68 - 9 \times 20) / 43520 = 99.43\%$
速度提高: $tm/ta = tm/(tc + (1-p)tm) = 10 / (1 + 10 \times (1-p)) = 9.5$ 倍

举例

	第0行	第1行	第2行	第3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 $(43520 - 68 - 7 \times 4 - 2 \times 8) / 43520 = 99.74\%$

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

写策略（Cache一致性问题）

- 为什么要保持在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

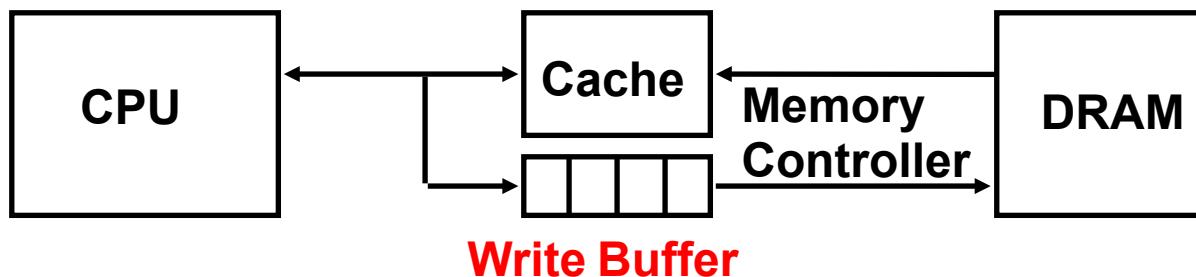
例如：DMA控制器读写主存时，如果对应Cache行中被修改，则DMA读出的内容无效；若DMA修改了主存单元的内容，则对应Cache行中内容无效。
 - 当多个CPU都带有各自的Cache而共享主存时

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Cache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： $1.0 + 100 \times 10\% = 11$
 - 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，但控制较复杂
- 对于写不命中，有两种处理方式
 - **Write Allocate (写分配)** 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?
 - 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)** Skip
 - 直接写主存单元，不把主存块装入到Cache

Write Through中的Write Buffer



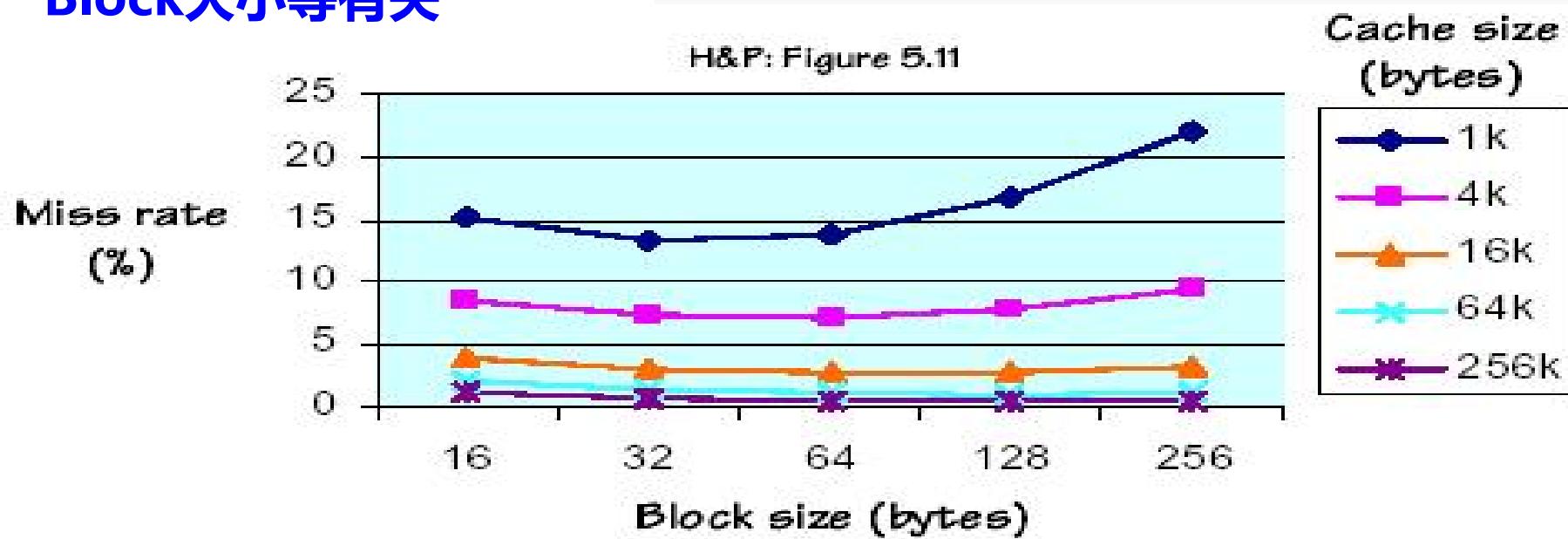
- 在 Cache 和 Memory之间加一个Write Buffer
 - CPU同时写数据到Cache和Write Buffer
 - Memory controller (存控) 将缓冲内容写主存
- Write buffer (写缓冲) 是一个FIFO队列
 - 一般有4项
 - 在存数频率不 高时效果好
- 最棘手的问题
 - 当频繁写时，易使写缓存饱和，发生阻塞
- 如何解决写缓冲饱和?
 - 加一个二级Cache
 - 使用Write Back方式的Cache

[BACK](#)

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

书架越大，在书架上找到书的概率就越大！
书架容量一定时，每排放的书越多，排数就越少。
每排的大小会影响找到书的概率



- 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，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L1Cache都是分立Cache，为什么？

两级Cache时L1Cache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L2Cache都是联合Cache，为什么？

L2Cache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： $4\text{GB} = 2^{20} \times 2^7 \text{块} \times 2^5 \text{B/块}$

Cache： $8\text{KB} = 128\text{组} \times 2\text{行/组}$

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：

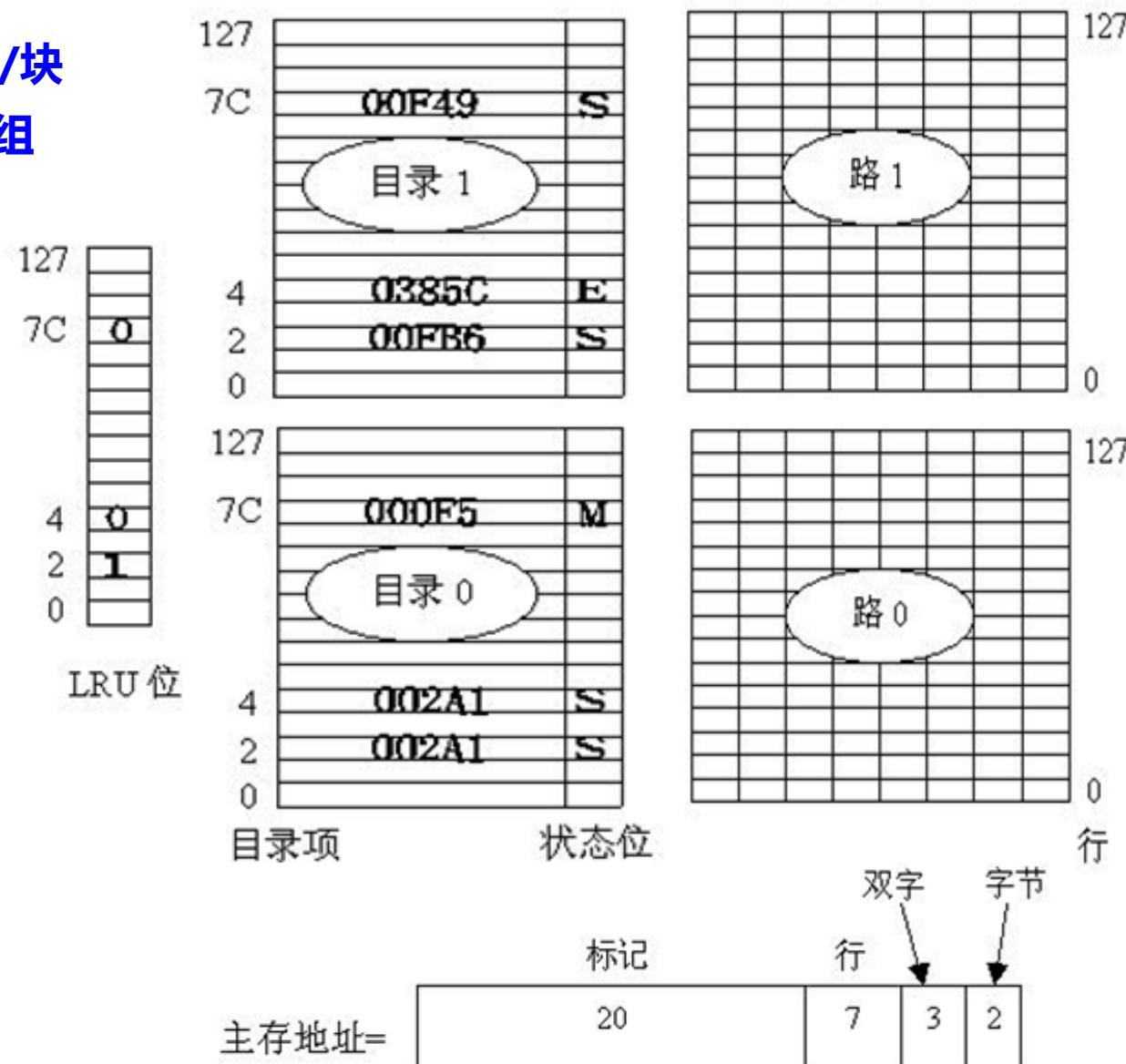
支持MESI协议

M：修改

E：互斥

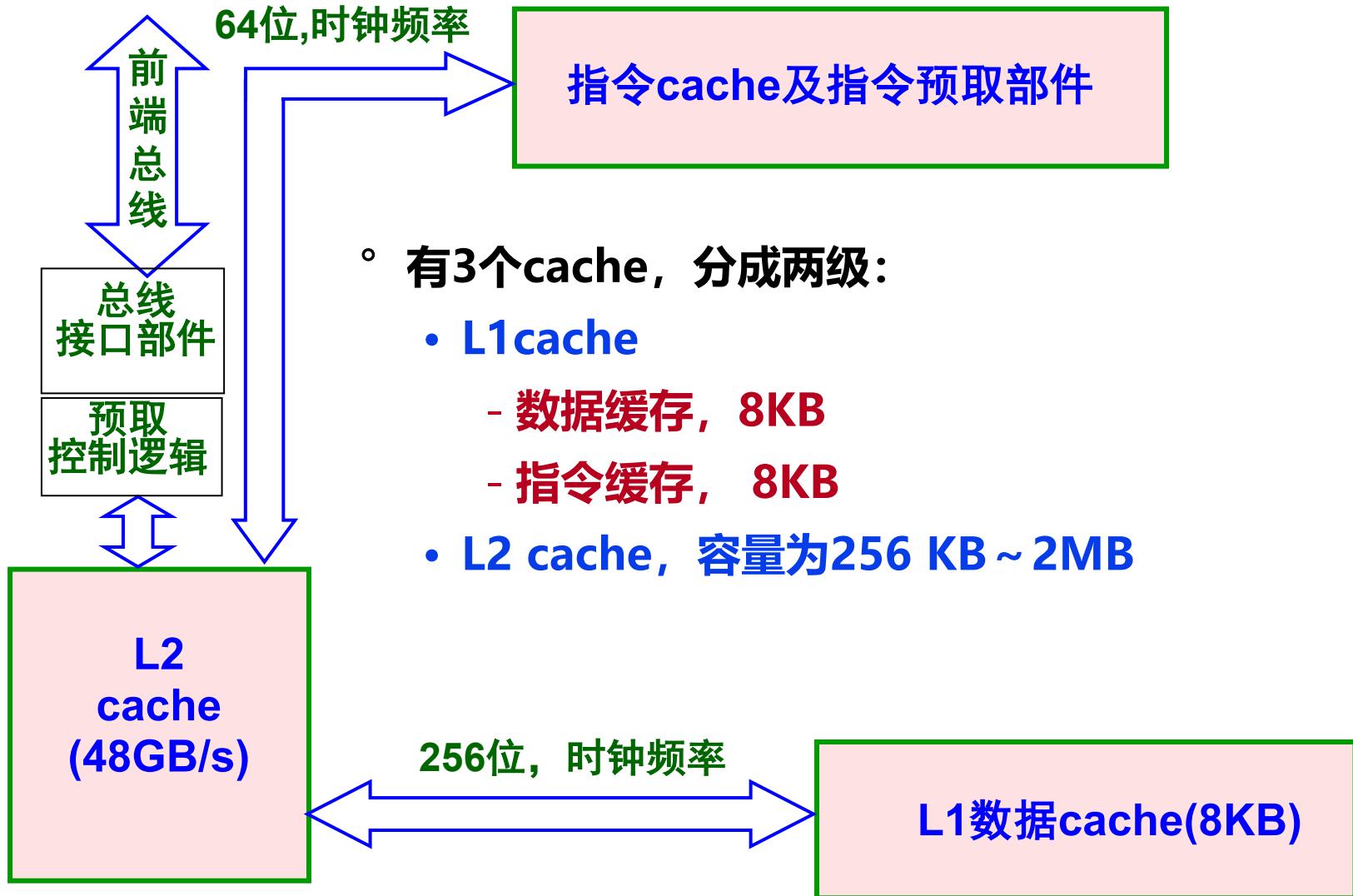
S：共享

I：无效

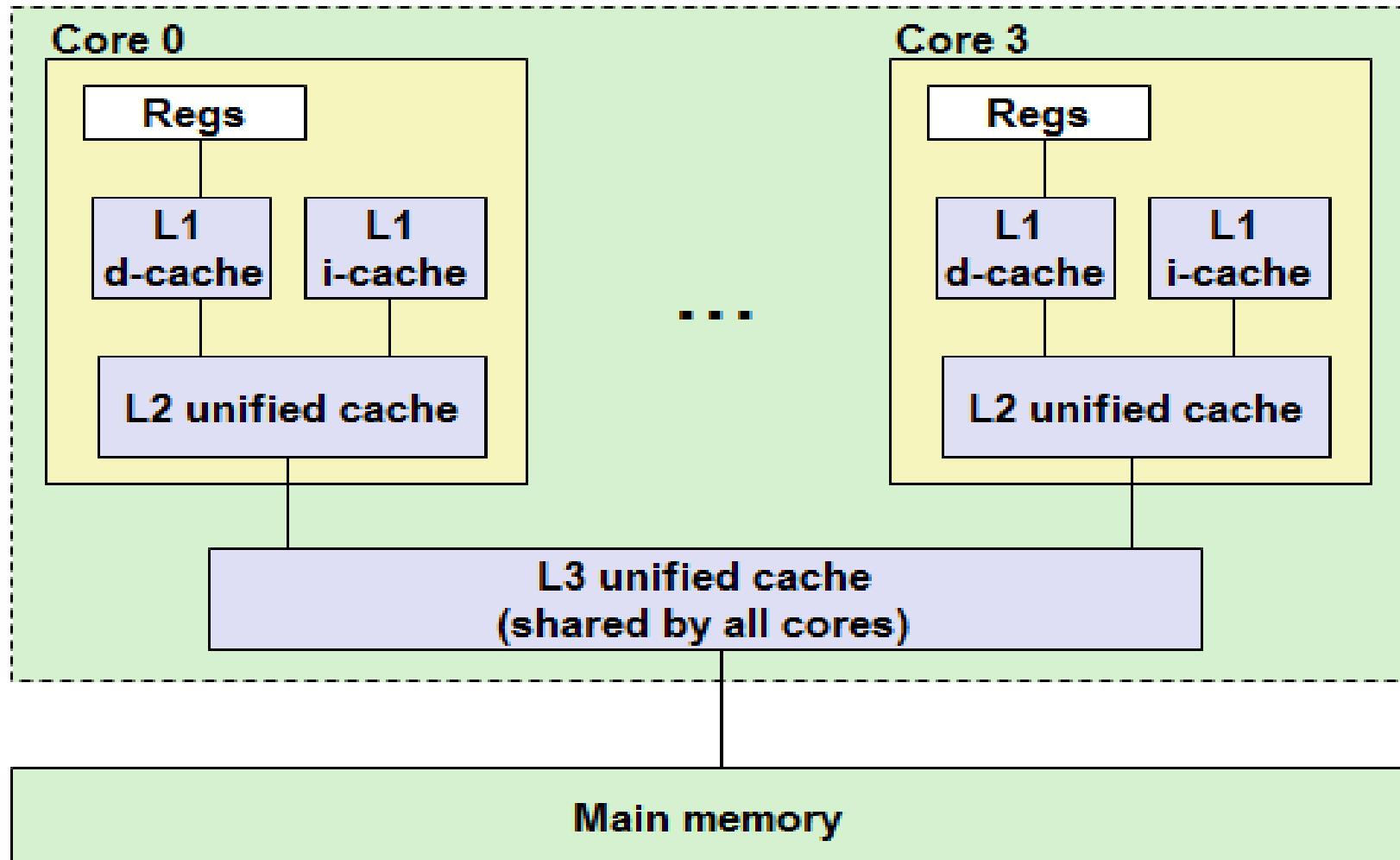


Pentium 内部数据 Cache 的结构

实例：Pentium 4的cache存储器



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

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，故主存地址为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：
 $8 \times (19+1+64 \times 8) = 4256$ 位 = 532字节。

(2) a[0][31]的地址为 $320 + 4 \times 31 = 444$, $[444/64] = 6$ (取整)，因此a[0][31]对应的主存块号为6。 $6 \bmod 8 = 6$ ，对应cache行号为6。
或: $444 = 0000\ 0000\ 0000\ 0000\ 110\ 111100$ B, 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：

$$[(320 + 4 \times (1 \times 256 + 1)) / 64] \bmod 8 = 5.$$

Cache和程序性能举例

a[0][0]所在主存块号
为: $320/64=5$

一个主存块占
 $64B/4B=16$ 个元素

总访问次数为:
 $256 \times 256 = 64K$

总块数(缺失次数)为
 $64K \times 4B / 64B = 4K$

缺失率为:
 $4K / 64K = 1/16$

命中率为:
 $1 - 4K / 64K = 15/16$

- 程序A对数组元素的访问过程: 直接映射
- 5#: a[0][0], a[0][1], ..., a[0][15] → → → 第5行
- 6#: a[0][16], a[0][17], ..., a[0][31] → → → 第6行
- 7#: a[0][32], a[0][33], ..., a[0][47] → → → 第7行
- 8#: a[0][48], a[0][49], ..., a[0][63] → → → 第0行
-
-, a[255][255]

每块都是第一个不命中，可以仅考虑一个主存块的情况：
第1次不命中，以后15次都命中，故命中率为 $15/16$

(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 $15/16 = 93.75\%$ 。

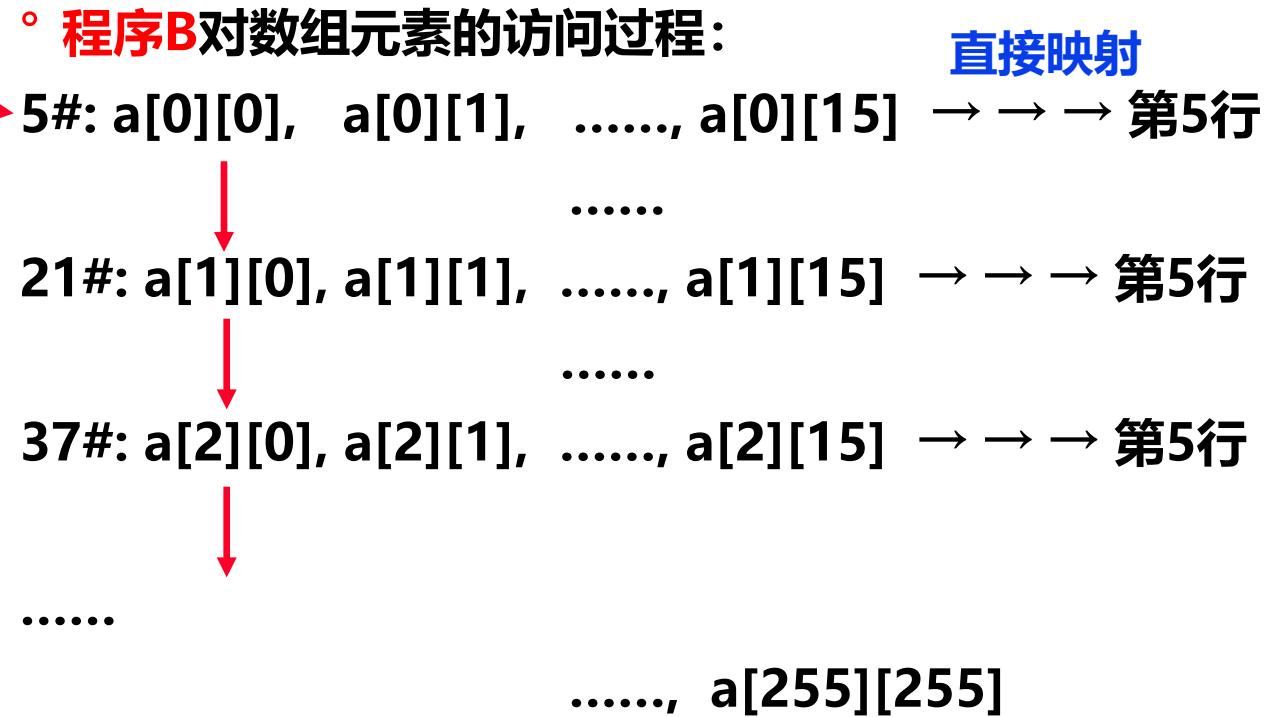
Cache和程序性能举例

a[0][0]所在主存块号
为: $320/64=5$

一个主存块占
 $64B/4B=16$ 个元素

每行数组元素占
 $256 \times 4B = 1024B$
即 $1024B/64B = 16$ 块

a[i][0]和a[i+1][0]之
间相差1024B, 即16
块, 因为 $16 \bmod 8 = 0$
因此, 被映射到cache
同一行中!



访问后面数组元素时, 总是把上一次装入到cache中的主
存块覆盖掉!

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
 $256 \times 4 = 1024$ 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。

层次结构存储系统

◦ 分以下六个部分介绍

- 第一讲：存储器概述
- 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：磁盘存储器
- 第四讲：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
- 第五讲：虚拟存储器 (Virtual Memory)
 - 虚拟地址空间、虚拟存储器的实现
- 第六讲：IA-32/Linux中的地址转换
 - 逻辑地址到线性地址的转换
 - 线性地址到物理地址的转换

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

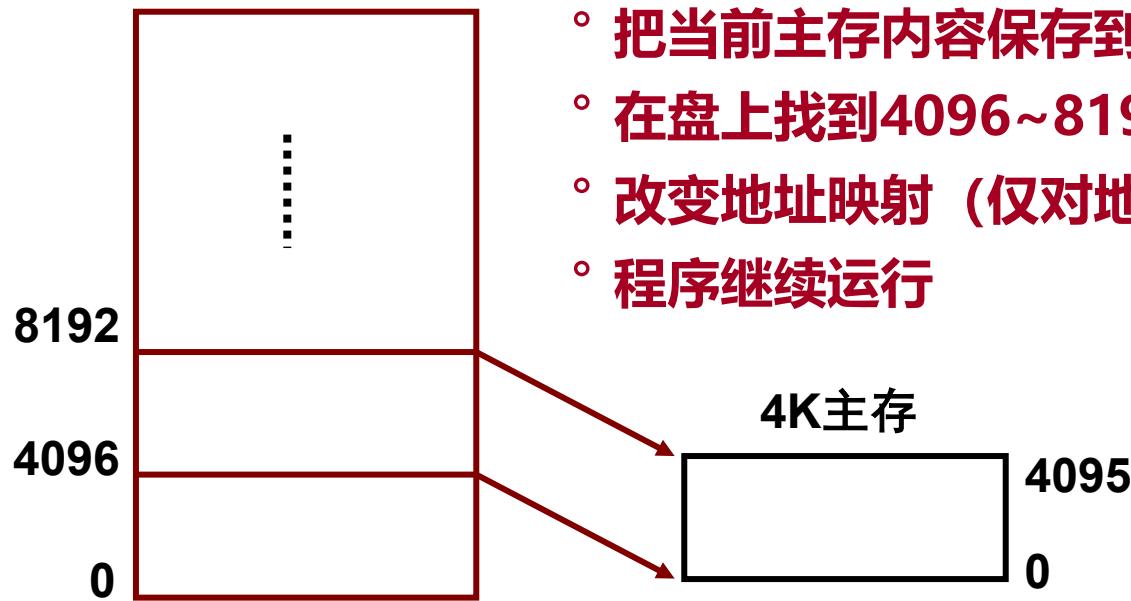
- 1961年，英国曼切斯特研究人员提出一种**自动执行overlay**的方式。
- 动机：把程序员从大量繁琐的存储管理工作中解放出来，**使得程序员编程时不用管主存容量的大小**。
- 基本思想：把**地址空间**和**主存容量**的概念区分开来。程序员在地址空间里编写程序，而程序则在真正的内存中运行。由一个**专门的机制**实现地址空间和实际主存之间的**映射**。
- 举例说明：

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，而主存容量只有4K字，则指令可寻址范围是多少？

地址空间为0、1、2...、65535组成的地址集合，即地址空间大小为 2^{16} 。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

地址空间



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（仅对地址取模即可） 模为多少？
- 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：
 - $4096 \rightarrow 0, 4097 \rightarrow 1, \dots, 8191 \rightarrow 4095$ 如何映射？
- 程程序员在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！

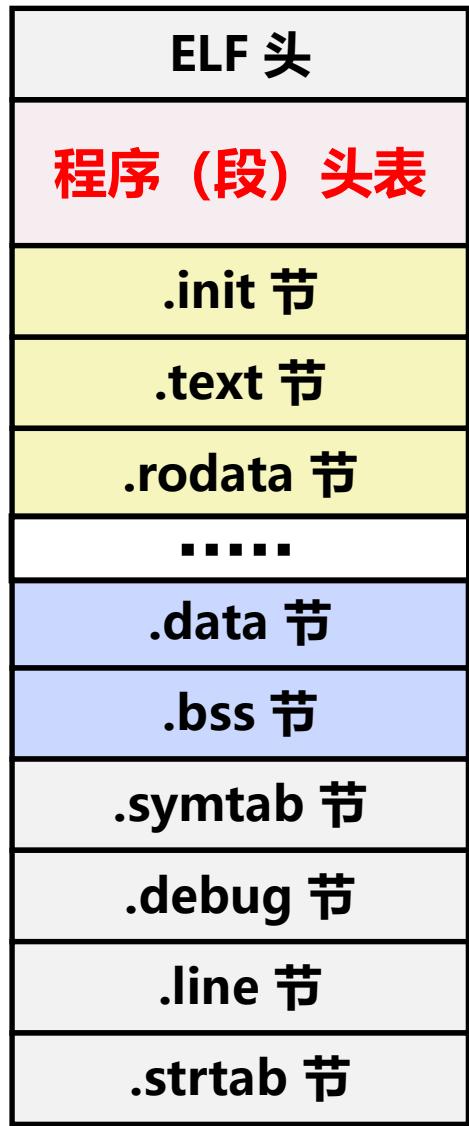
分页 (Paging)

- 基本思想：
 - 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)
 - 程序块可装入主存页框中
 - 无需用连续页框存放一个进程
 - 操作系统为每个进程生成一个页表
 - 通过页表(page table)实现逻辑地址向物理地址转换
 - 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主存地址, 也称为实地址、主存地址。
- 如：页大小=页框大小=4KB
- 程序块装入主存前存放在哪里?
- 每页的起始地址是什么形式?
- 磁盘和虚拟地址空间如何关联?

回顾：可执行文件的存储器映像

程序(段)头表描述如何映射

00000



0xC00000000

004d3

00f0c

01014

0101c

0x08049000

0x08048000

0

内核虚存区

用户栈 (User stack)
动态生成

共享库区域

堆 (heap)
(由 malloc 动态生成)

读写数据段
(.data, .bss)

只读代码段
(.init, .text, .rodata)

未使用

1GB

%esp
(栈顶)

空洞
页面

brk

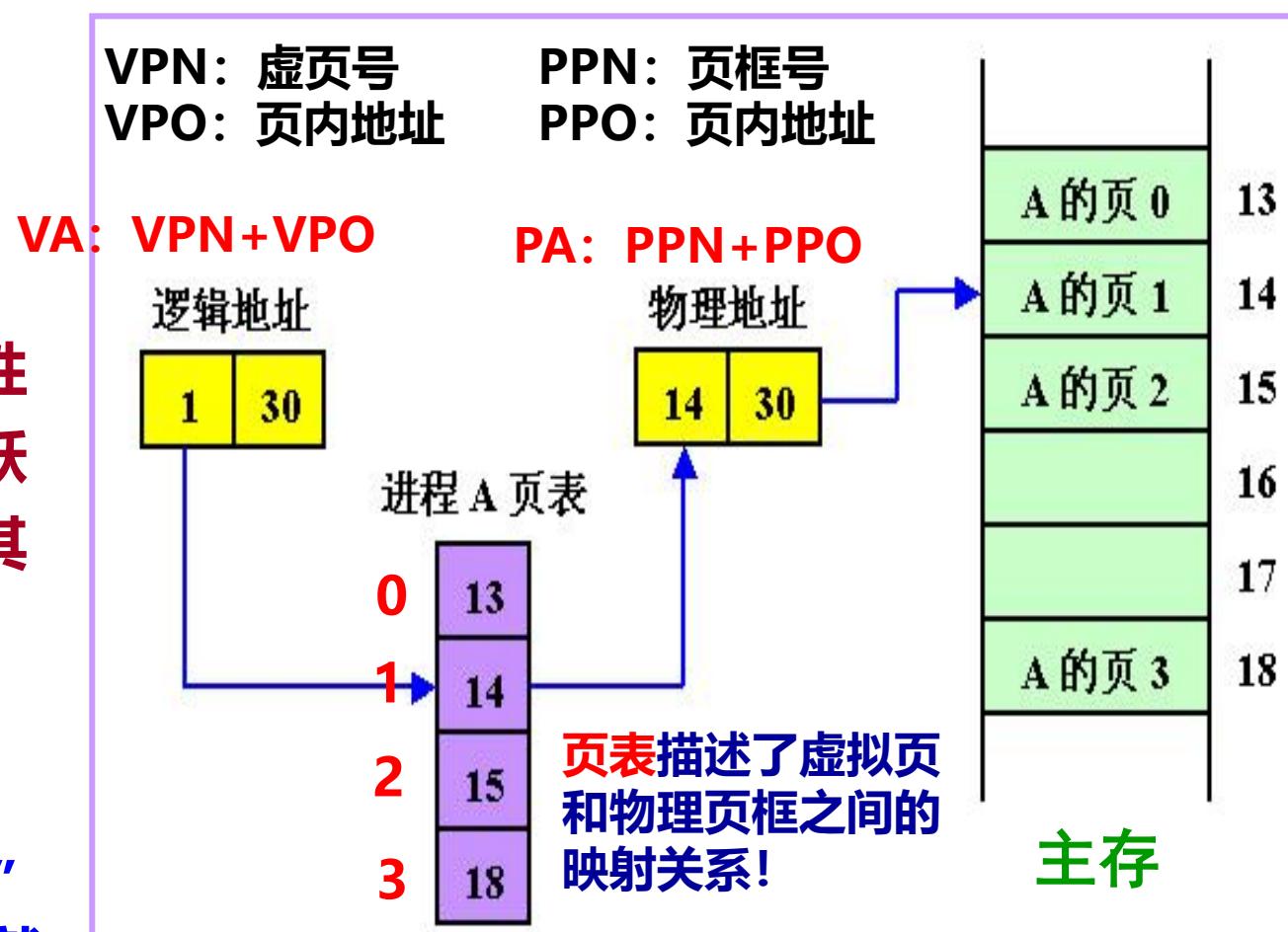
从可
执行
文件
装入

分页 (Paging)

问题：是否需要将一个进程的全部都装入内存？为什么？

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



页大小有什么特点？和页内地址位数有什么关系？
每个页和每个页框的起始地址一定是固定的吗？
地址转换由硬件还是软件实现？为什么？

虚拟存储系统的基本概念

- 虚拟存储技术的最初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](#)

虚拟地址空间

虚拟地址空间

页表1

主存物理空间

页表k

操作系统程序

用户程序k页面

用户程序1页面

.....

用户程序2页面

仅装入当前所需代码和数据

存储全部

磁盘物理空间

用户程序1

用户程序k

用户程序1

用户程序2

发生缺页时、调入新页

虚拟地址空间

0xC00000000

- Linux在X86上的虚拟地址空间
(其他Unix系统的设计类此)

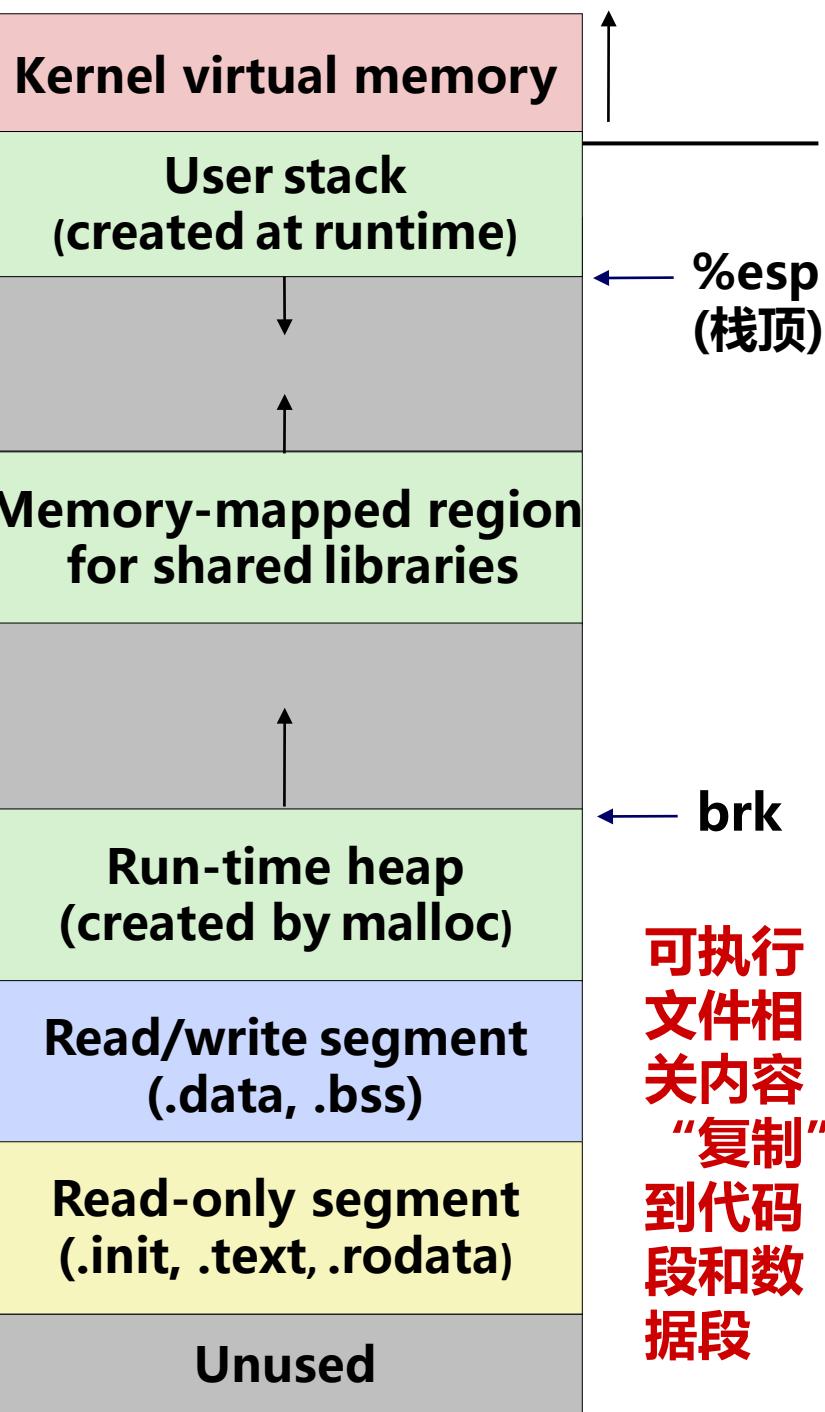
- 内核空间 (Kernel)
- 用户栈 (User Stack)
- 共享库 (Shared Libraries)
- 堆 (heap)
- 可读写数据 (Read/Write Data)
- 只读数据 (Read-only Data)
- 代码 (Code)

问题：加载时是否真正从磁盘调入信息到主存？

不会，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”。
mmap系统调用！

0x08048000

0



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？

主存与虚存的空间如何分区管理？

程序块（页） / 存储块（页框）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？

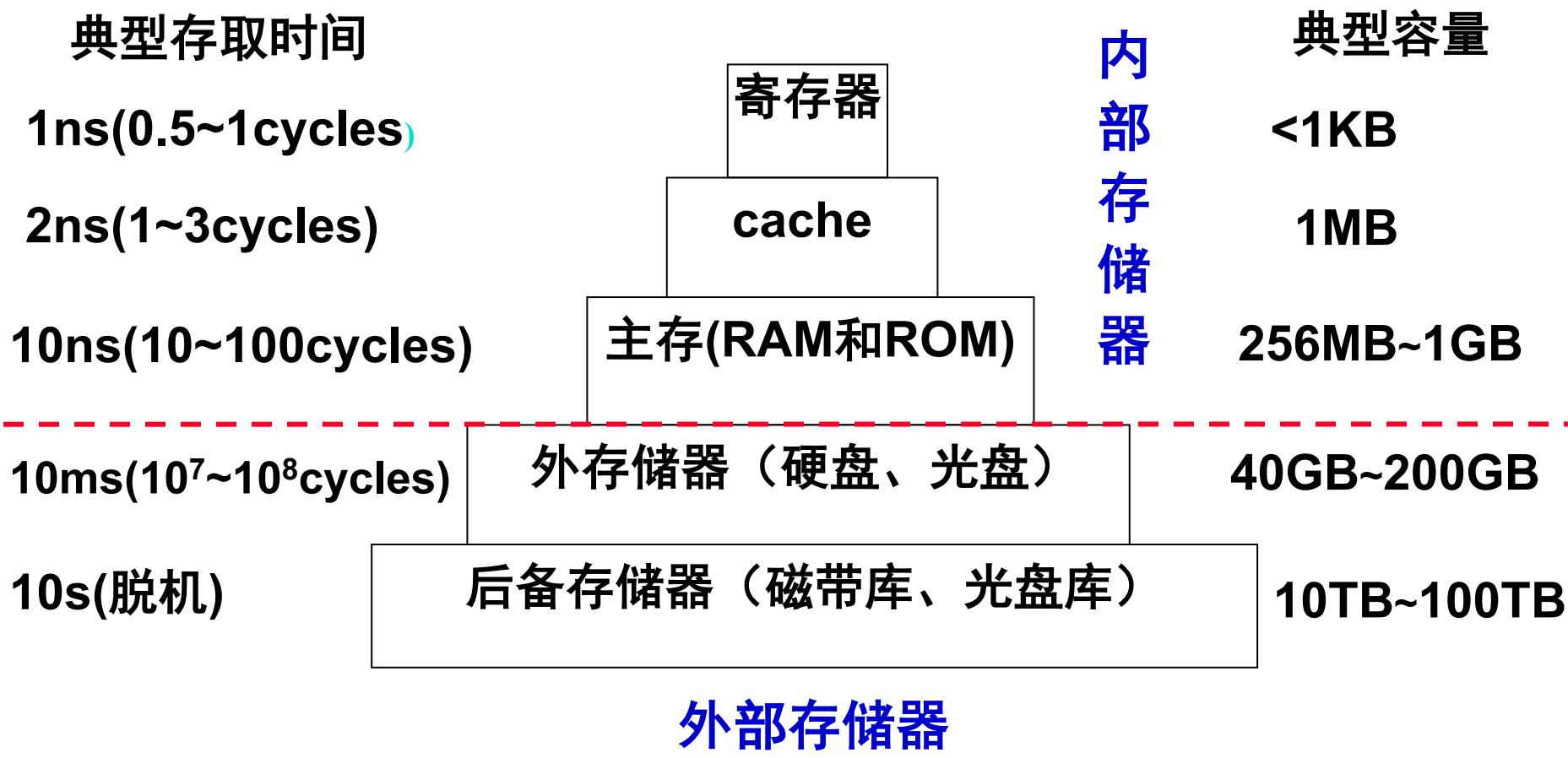
如何保护进程各自的存储区不被其他进程访问？

有三种虚拟存储器实现方式：

分页式、分段式、段页式

这些问题是由硬件和OS
共同协调解决的！

回顾：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?

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?

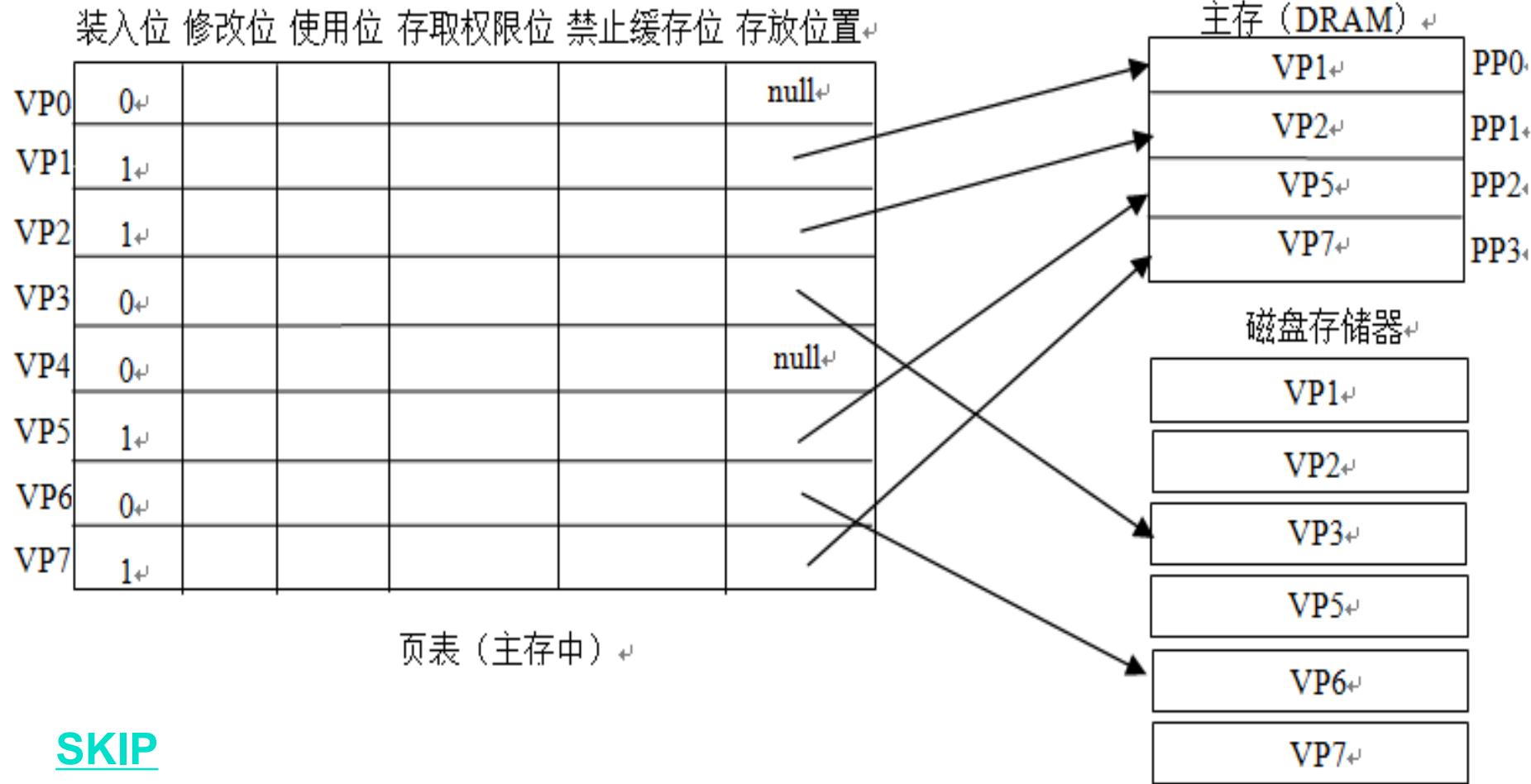
加快指令执行

页表结构



- 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



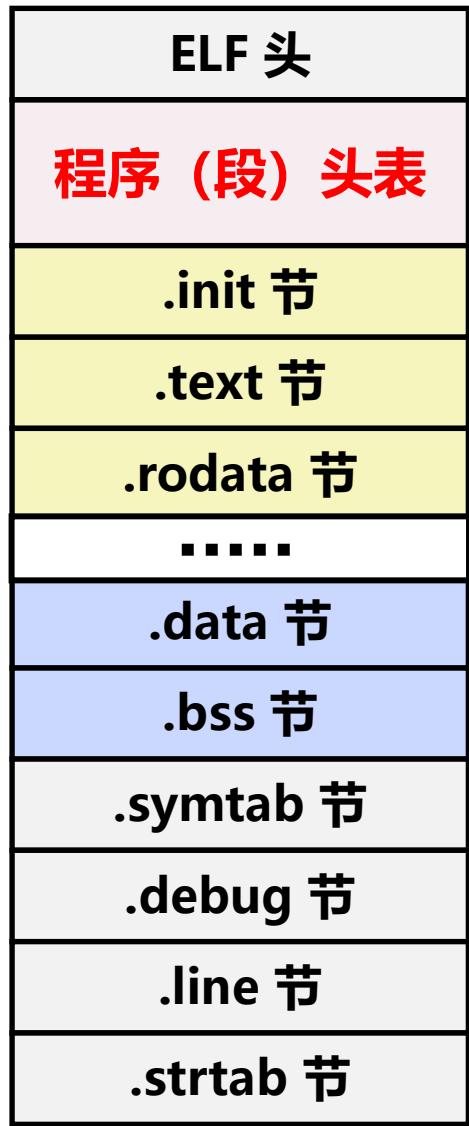
SKIP

- ◆ **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)

回顾：可执行文件的存储器映像

程序(段)头表描述如何映射

00000



0xC00000000

0x08049000

0x08048000

0

内核虚存区

用户栈 (User stack)
动态生成

共享库区域

堆 (heap)
(由 malloc 动态生成)

读写数据段
(.data, .bss)

只读代码段
(.init, .text, .rodata)

未使用

1GB

%esp
(栈顶)

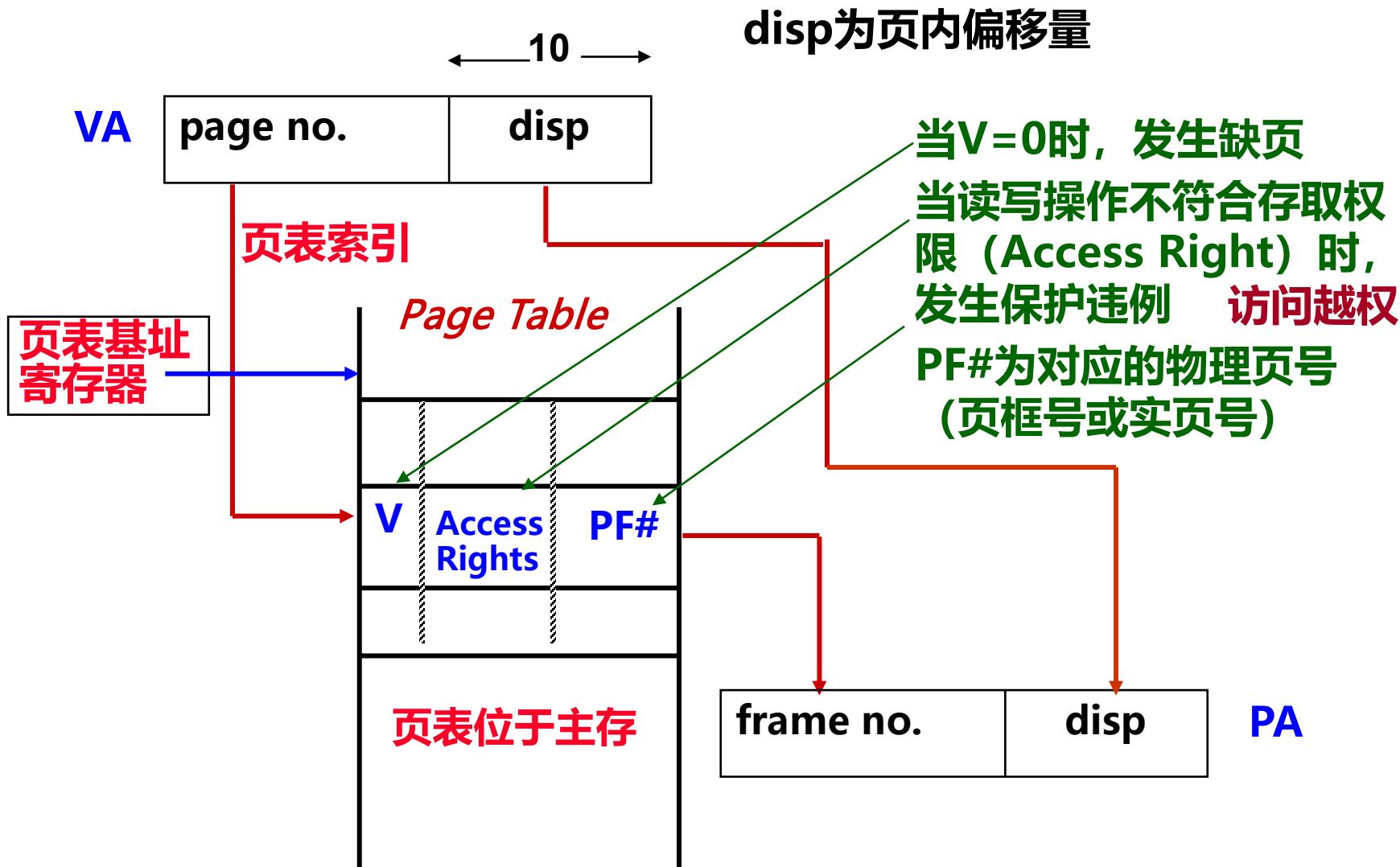
空洞
页面

BACK

brk

从可
执行
文件
装入

逻辑地址转换为物理地址的过程



问题：什么情况下，不能成功进行地址转换呢？

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：从磁盘读页面到主存，若主存没有空间，则从主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 (protectionViolationFault) 或访问违例

产生条件：当Access Rights (存取权限)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？

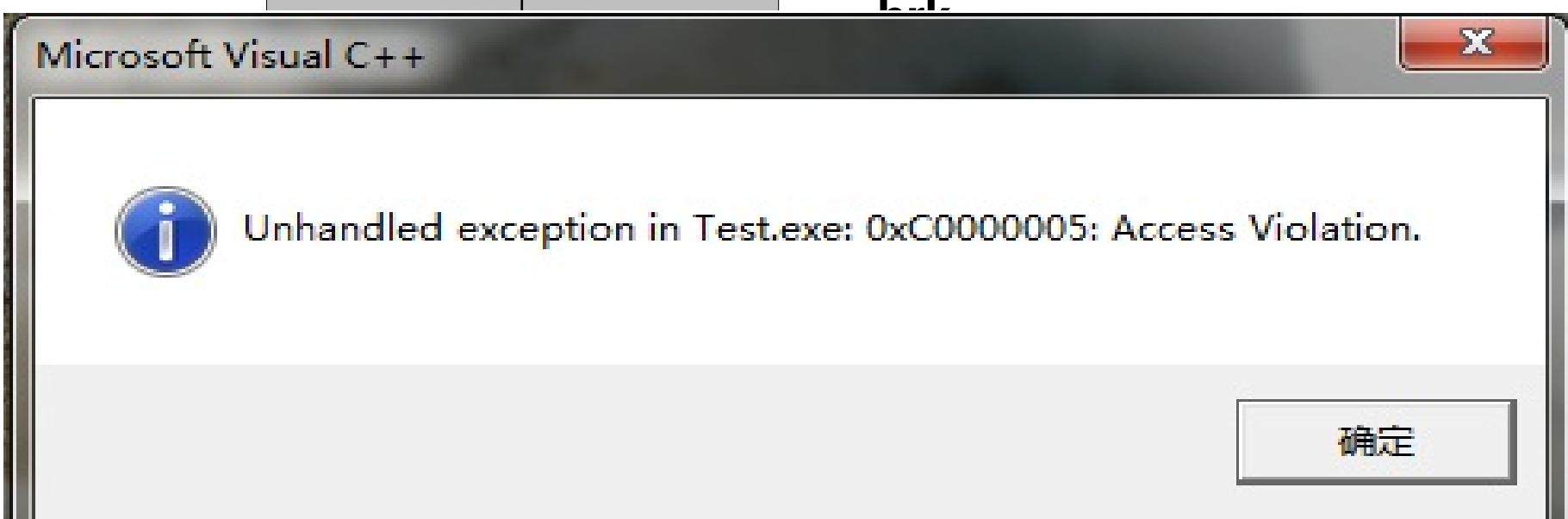
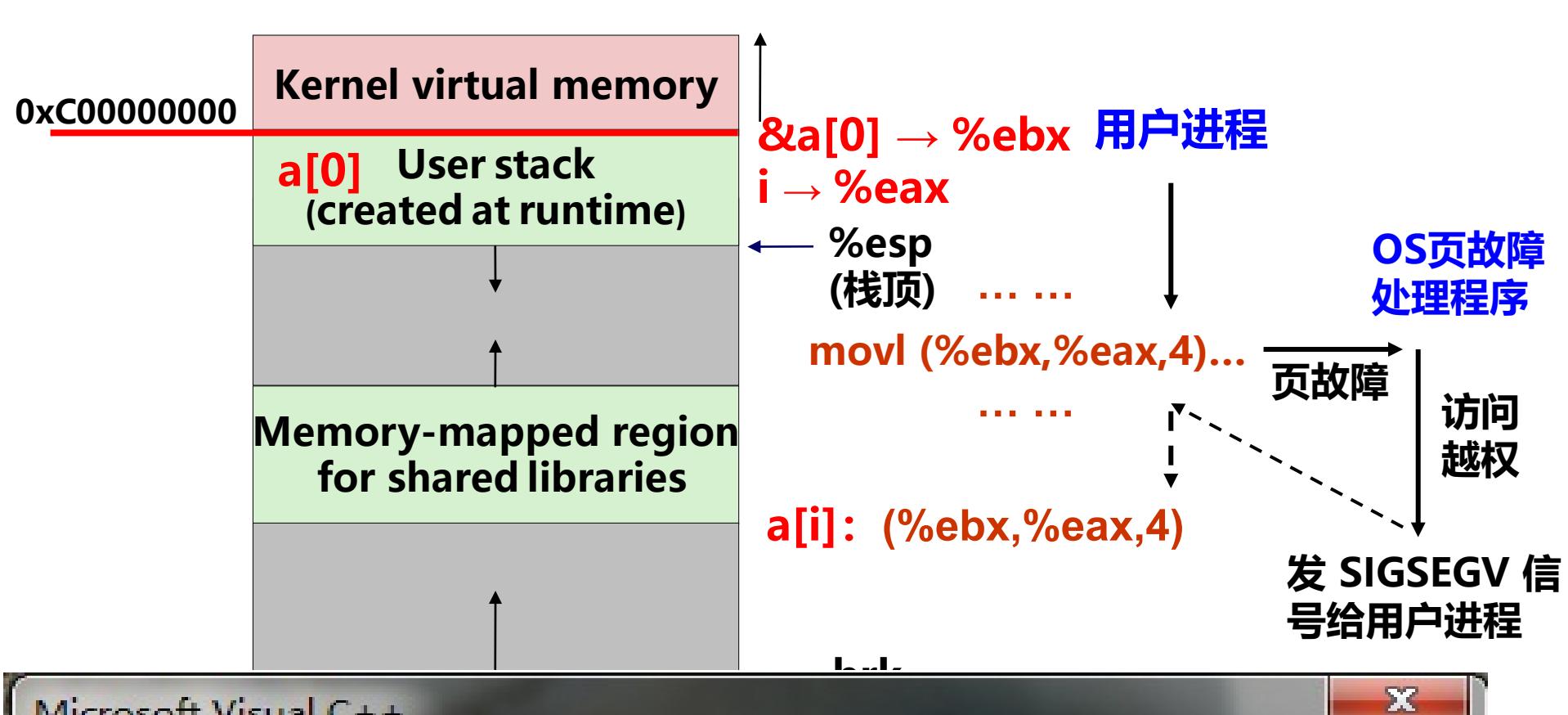
R = Read-only, R/W = read/write, X = execute only

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?





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)

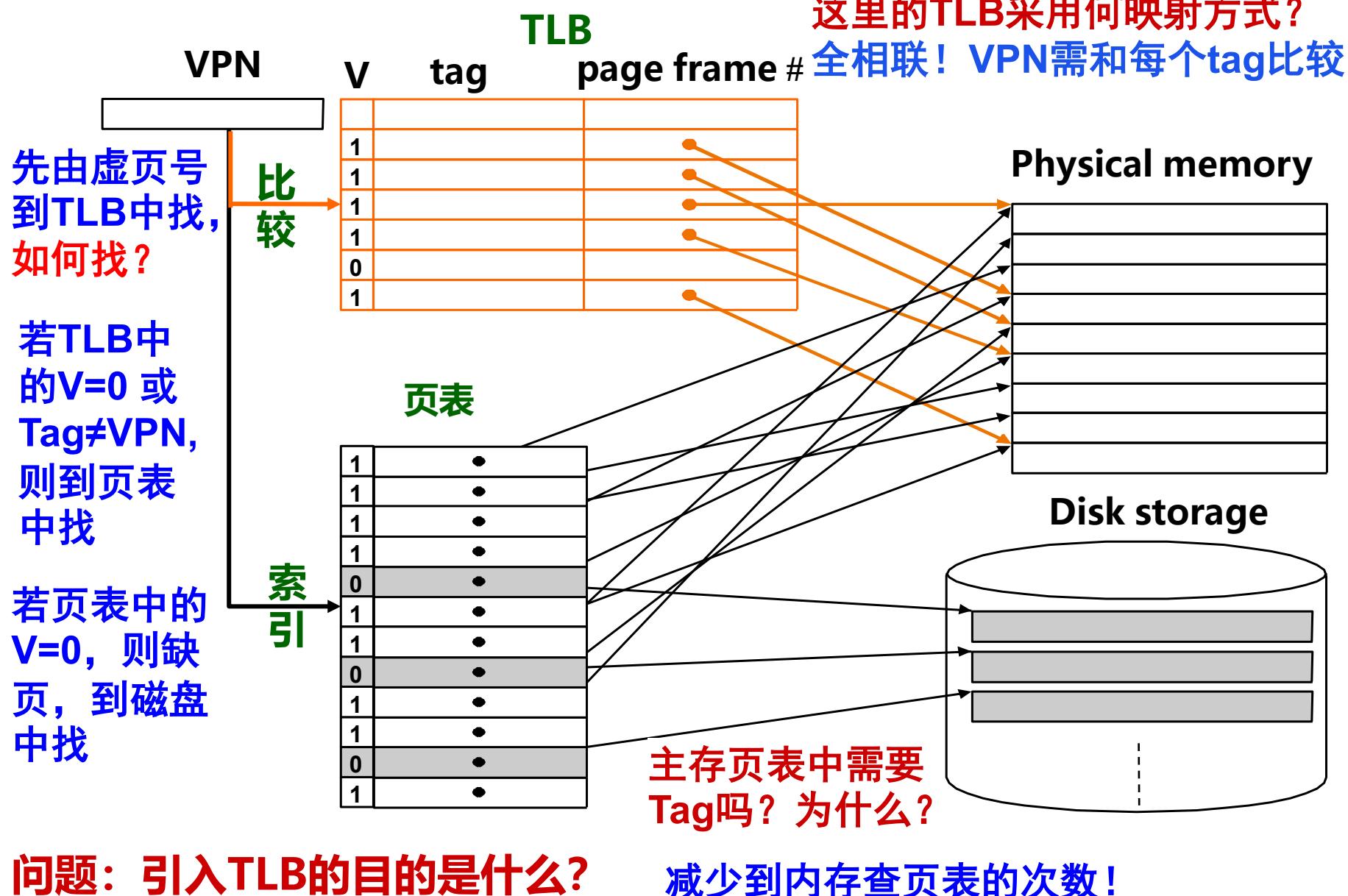
TLB中的页表项： tag + 主存页表项

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

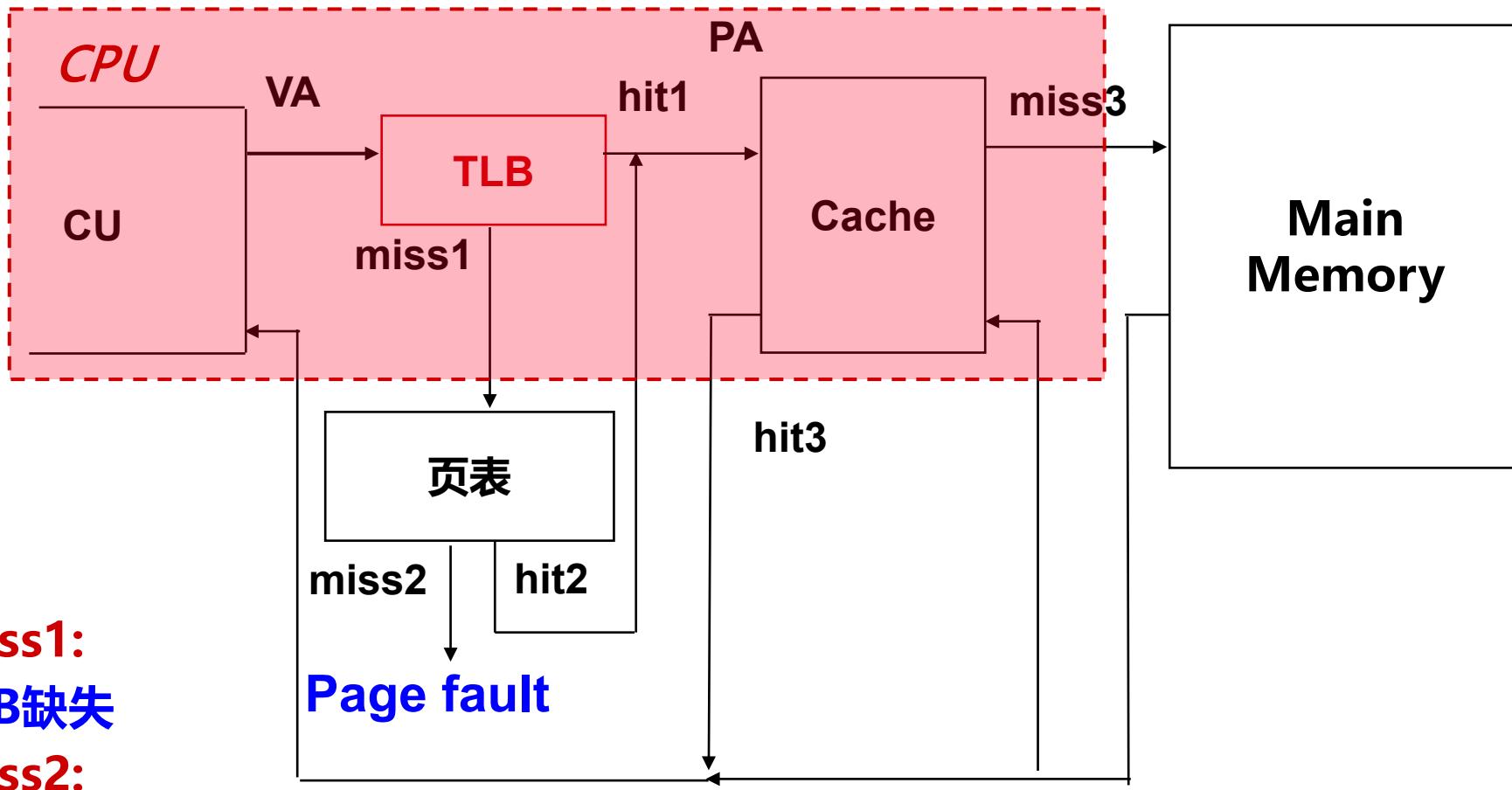
CPU访存时，地址中虚页号被分成tag + index， tag用于和TLB页表项中的tag比较， index用于定位需比较的表项

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 --- Making Address Translation Fast



Translation Look-Aside Buffers



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

P268图6.33

虚拟地址

TLB

页表

缺页
处理

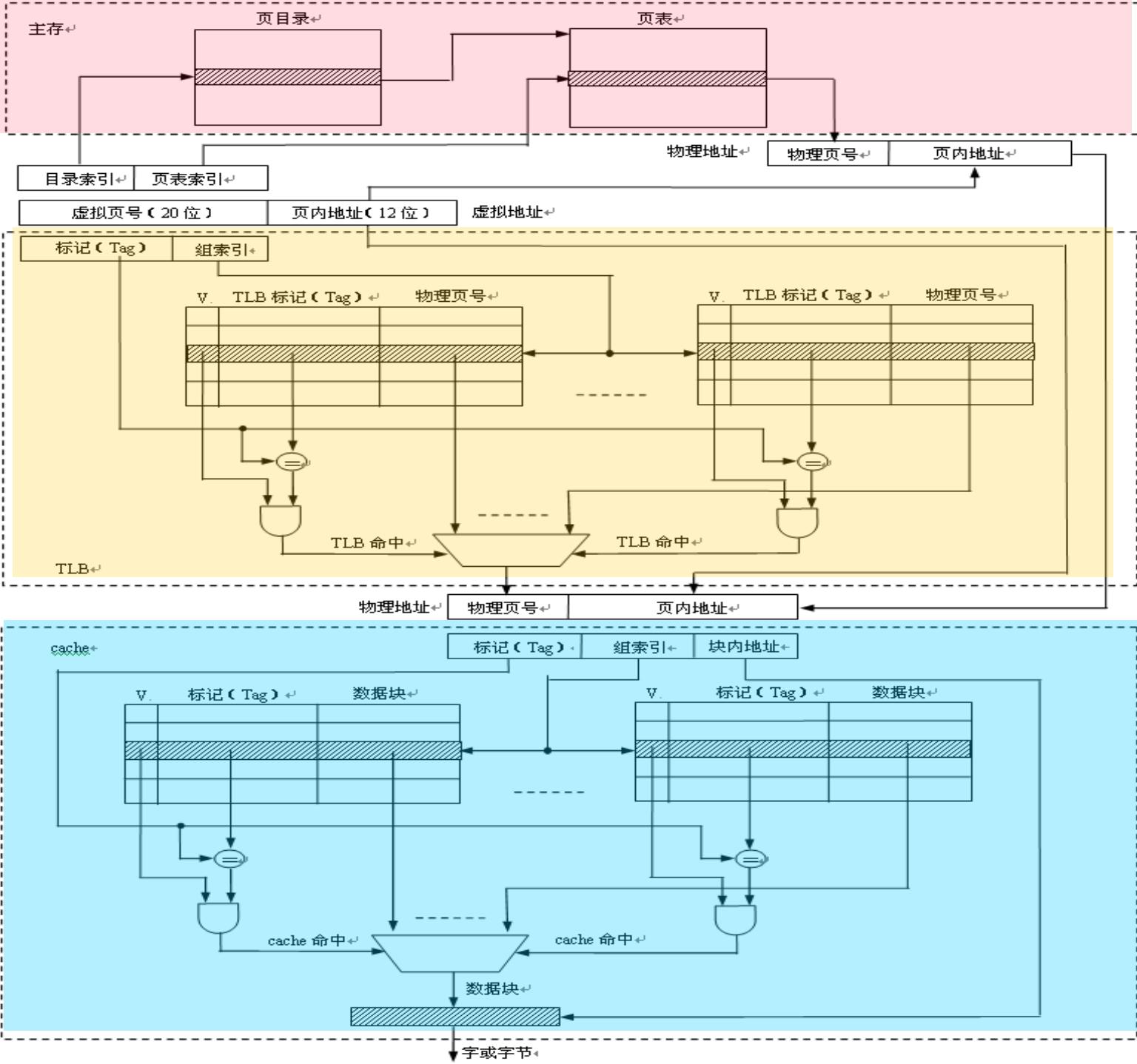
物理地址

命中

cache

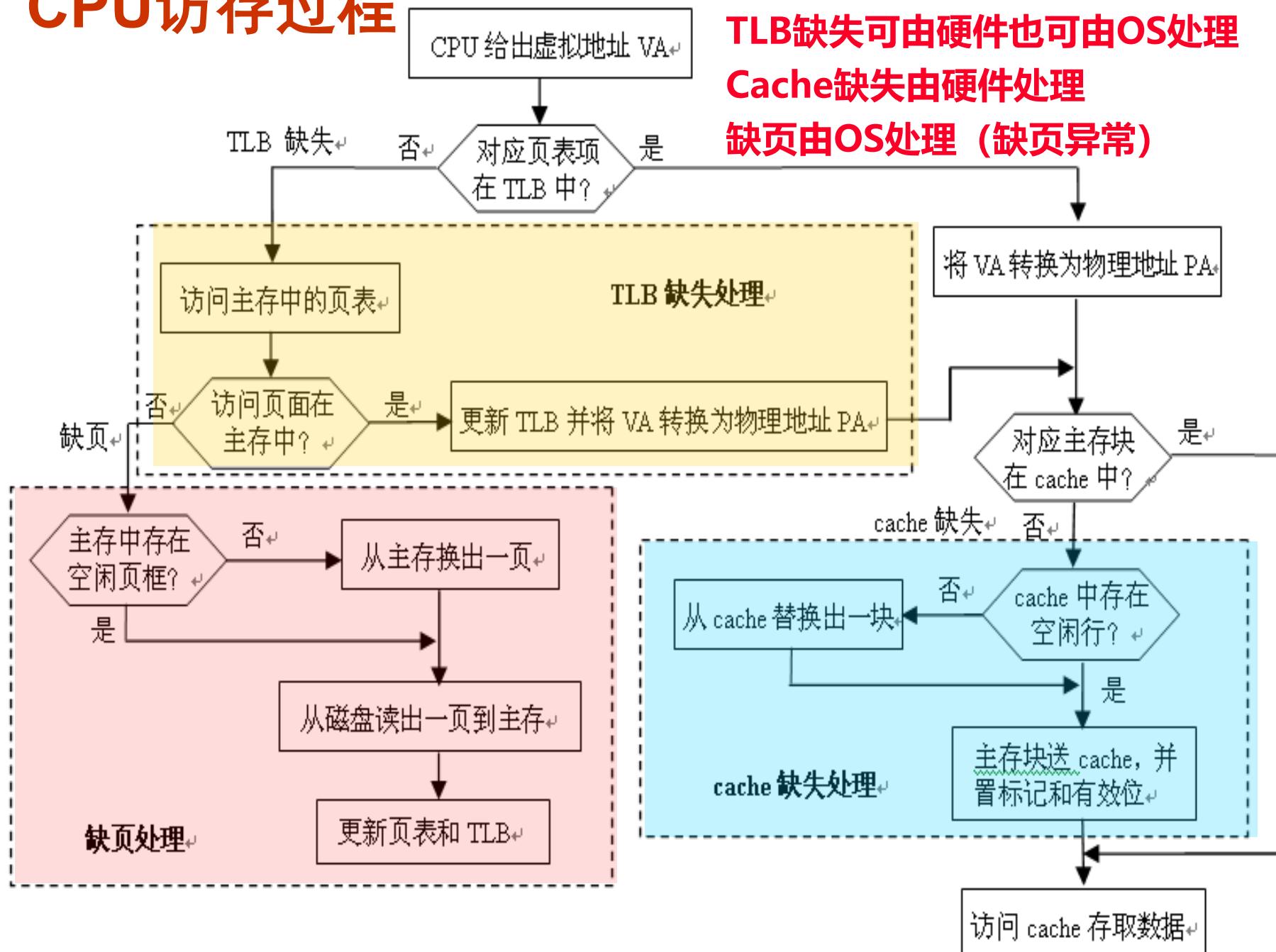
缺失

主存



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

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

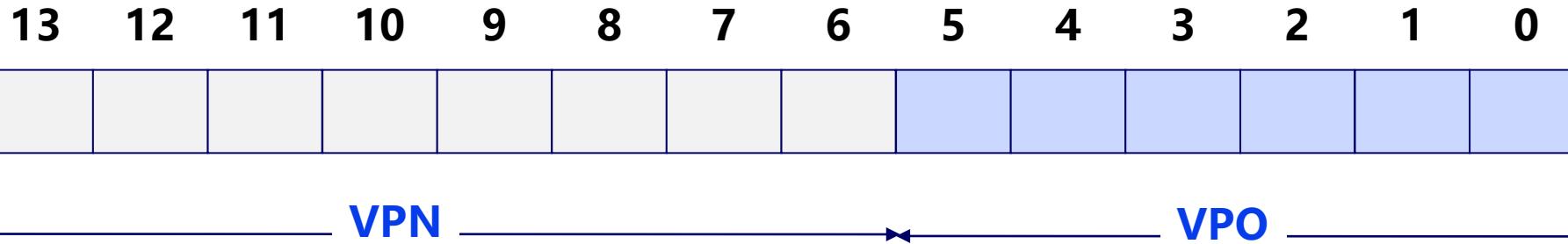
- PPO: Physical page offset (页内偏移地址)
- PPN: Physical page number (物理页号)
- CO: Byte offset within cache line (块内偏移地址)
- CI: Cache index (cache索引)
- CT: Cache tag (cache标记)

一个简化的存储系统举例

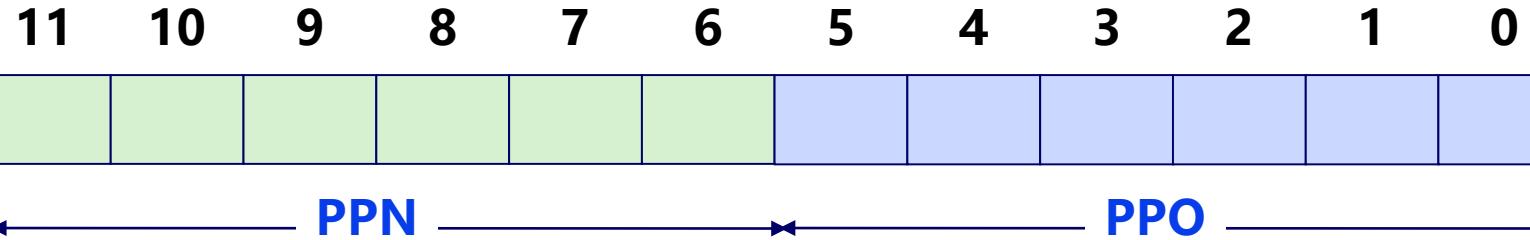
° 假定以下参数，则虚拟地址和物理地址如何划分？共多少页表项？

- 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：
 $2^{14-6} = 256$



Virtual Page Number Virtual Page Offset



Physical Page Number

Physical Page Offset

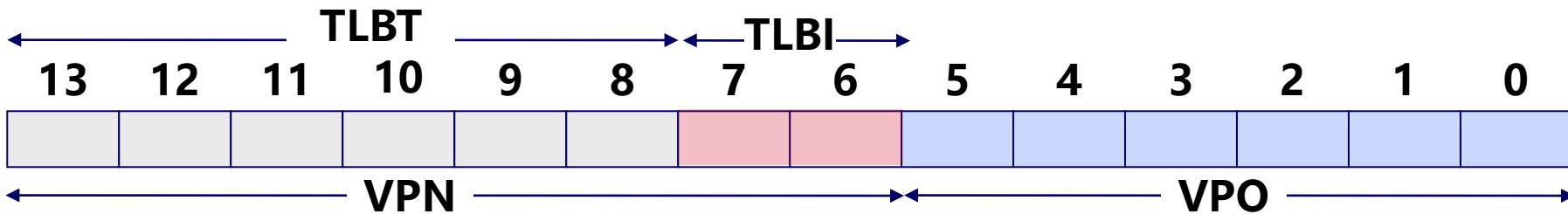
一个简化的存储系统举例（续）

假定部分页表项内容（
十六进制表示）如右：

思考题：红色数字处
存在什么问题？

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

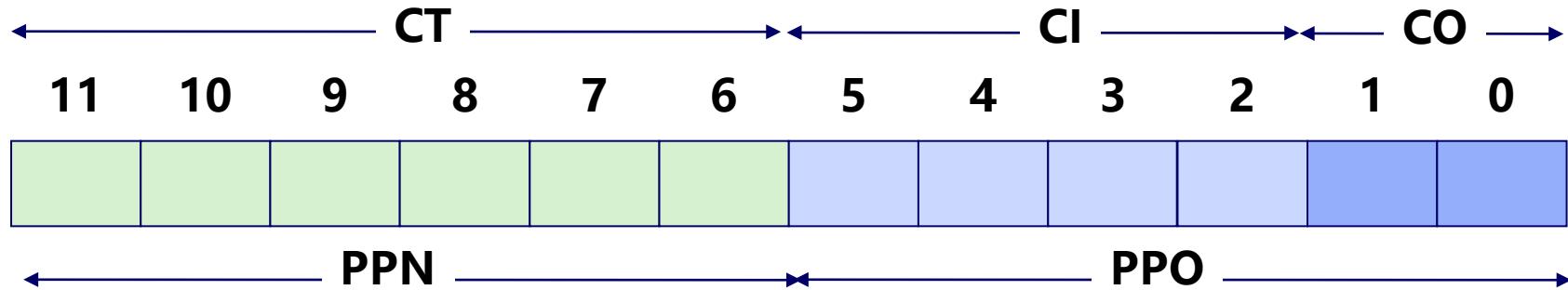
<i>VPN</i>	<i>PPN</i>	<i>Valid</i>	<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
000	28	1	028	13	1
001	-	0	029	17	1
002	33	1	02A	09	1
003	02	1	02B	-	0
004	-	0	02C	-	0
005	16	1	02D	2D	1
006	-	0	02E	11	1
007	-	0	02F	0D	1



<i>Set</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>									
0	03	-	0	09	08	0	00	-	0	07	02	1
1	0B	2D	1	0A	17	1	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	0B	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？

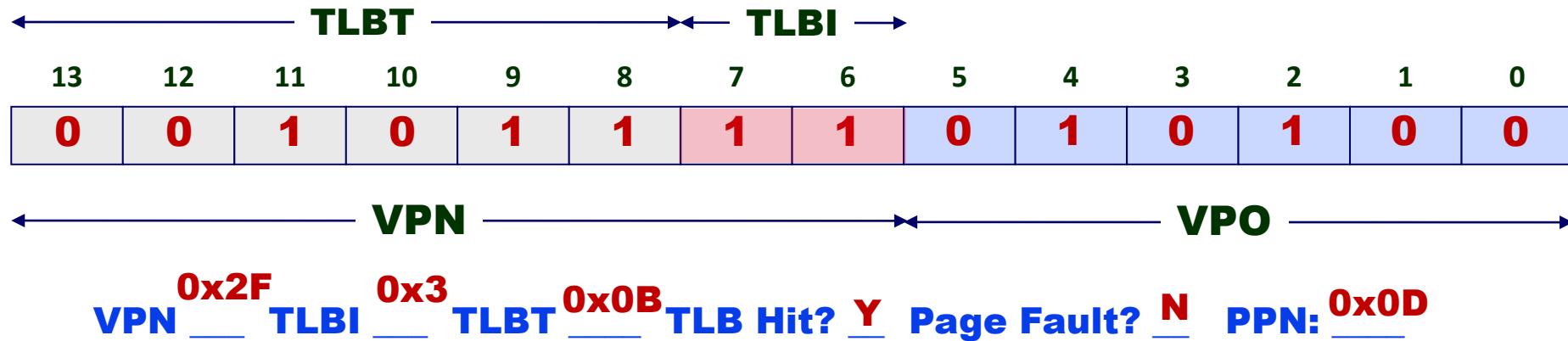


<i>Idx</i>	<i>Tag</i>	<i>V</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	1B	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03

<i>Idx</i>	<i>Tag</i>	<i>V</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
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	-	-	-	-

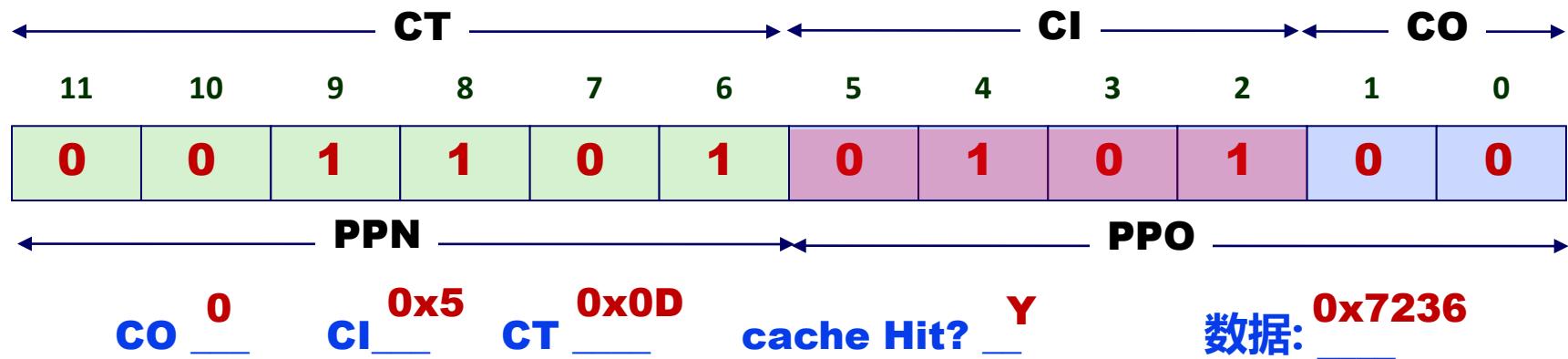
一个简化的存储系统举例（续）

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0x0BD4，说明访存过程。



物理地址为

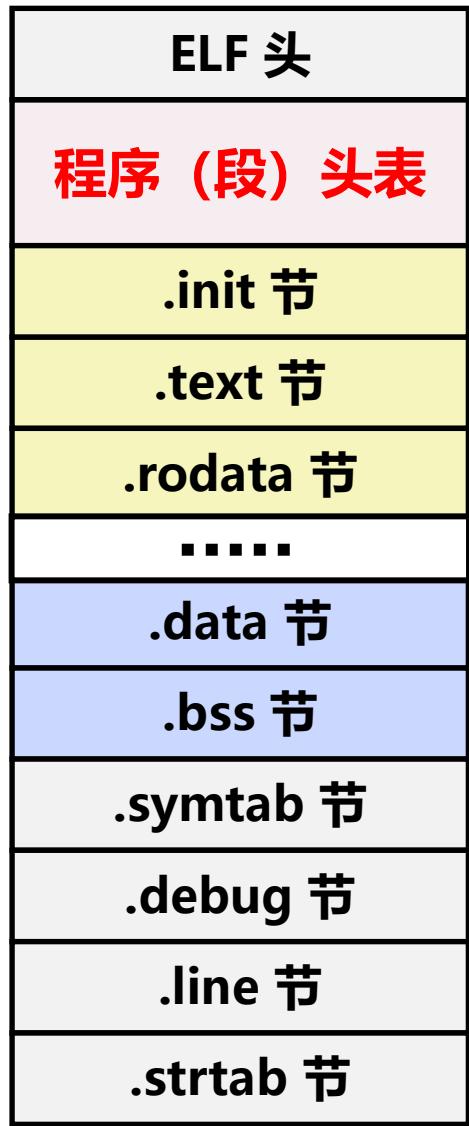
问题：逻辑地址为0x0A7A、0x0507时的访存过程如何？



回顾：可执行文件的存储器映像

程序(段)头表描述如何映射

00000



0xC00000000

004d3

00f0c

01014

0101c

0x08049000

0x08048000

0

内核虚存区

用户栈 (User stack)
动态生成

共享库区域

堆 (heap)
(由 malloc 动态生成)

读写数据段
(.data, .bss)

只读代码段
(.init, .text, .rodata)

未使用

1GB

%esp
(栈顶)

空洞
页面

brk

从可
执行
文件
装入

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存储段，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

- 分页、分段方式各自的优缺点

- 分页：浪费空间 或 只读代码和可读写数据在同一页（无法设置权限）
- 分段：不同段分开易管理，但内存会形成大量碎片

- 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，进入主存后仍以页为基本单位管理
- 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
 - 先分段：根据段地址到段表中找到该段对应的页表首地址
 - 再分页：根据页地址从页表中找到对应页框地址，形成物理地址

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

内存访问时的异常信息



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执行非内核的用户程序时，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：这部分信息包括：页表、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）来读写这些信息
 - 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“中断”使CPU从用户态转到内核态；异常/中断处理中的“返回”指令使CPU从内核态转到用户态
- 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

层次结构存储系统

◦ 分以下六个部分介绍

- 第一讲：存储器概述
- 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：磁盘存储器
- 第四讲：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
- 第五讲：虚拟存储器 (Virtual Memory)
 - 虚拟地址空间、虚拟存储器的实现
- 第六讲：IA-32+Linux中的地址转换
 - 逻辑地址到线性地址的转换
 - 线性地址到物理地址的转换

IA-32的存储管理

- 按字节编址（通用计算机大多是）
- 在**保护模式**下，IA-32采用**段页式**虚拟存储管理方式
- 存储地址采用**逻辑地址**、**线性地址**和**物理地址**来进行描述，其中，逻辑地址和线性地址是虚拟地址的两种不同表示形式，描述的都是4GB虚拟地址空间中的一个存储地址
 - ✓ 逻辑地址由48位组成，包含16位段选择符和32位段内偏移量（即有效地址） `movw 8(%ebp,%edx,4), %ax`
 - ✓ 线性地址32位（其位数由虚拟地址空间大小决定）
 - ✓ 物理地址32位（其位数由存储器总线中的地址线条数决定）
- 分段过程实现将逻辑地址转换为线性地址 ←———— 以下介绍分段机制
- 分页过程实现将线性地址转换为物理地址

IA-32处理器的寻址方式

IA-32指令举例：

`movw 8(%ebp,%edx,4), %ax // R[ax]←M[R[ebp]+R[edx]×4+8]`

操作数的来源：

32位有效地址

- 立即数(立即寻址)：直接来自指令
- 寄存器(寄存器寻址)：来自32位 / 16位 / 8位通用寄存器
- 存储单元(其他寻址)：需进行地址转换

逻辑地址 => 线性地址LA (=> 内存地址)

即采用段页式！

分段

分页

指令中的信息：

- (1) 段寄存器SR (隐含或显式给出)
- (2) 8/16/32位偏移量A (显式给出)
- (2) 基址寄存器B (明显给出，任意通用寄存器皆可)
- (3) 变址寄存器I (明显给出，除ESP外的任意通用寄存器皆可。)

➤ 有比例变址和非比例变址

➤ 比例变址时要乘以比例因子S (1:8位 / 2:16位 / 4:32位 / 8:64位)

IA-32处理器寻址方式

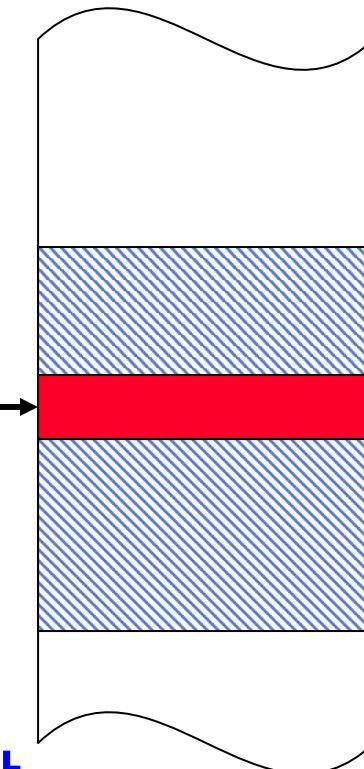
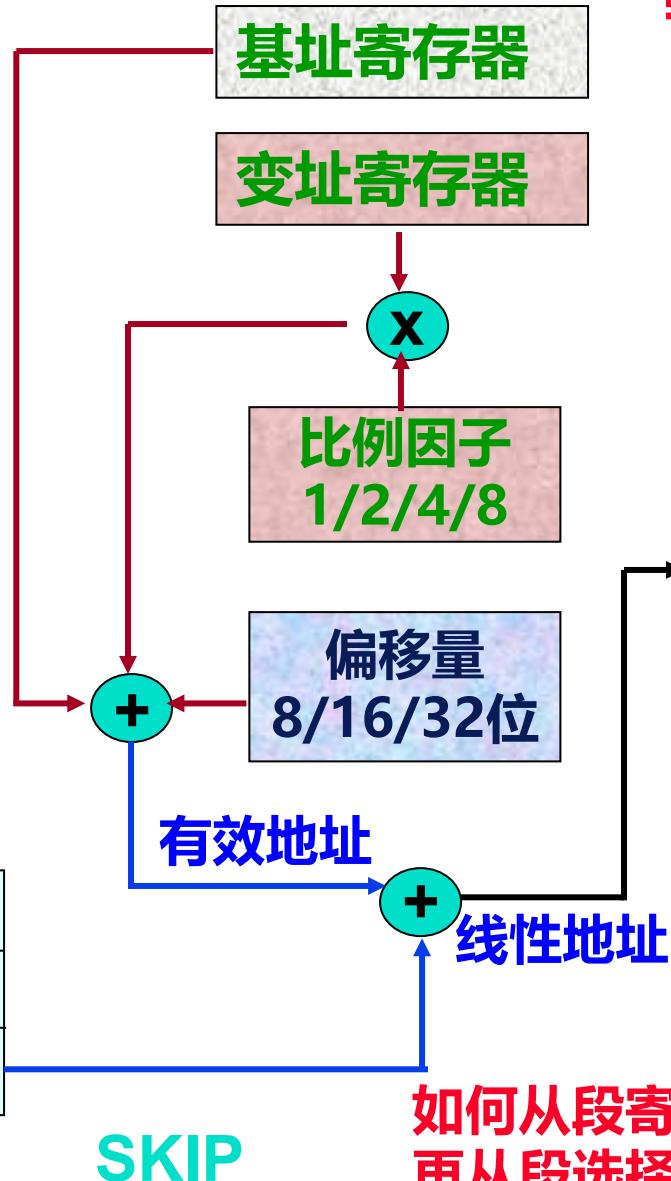
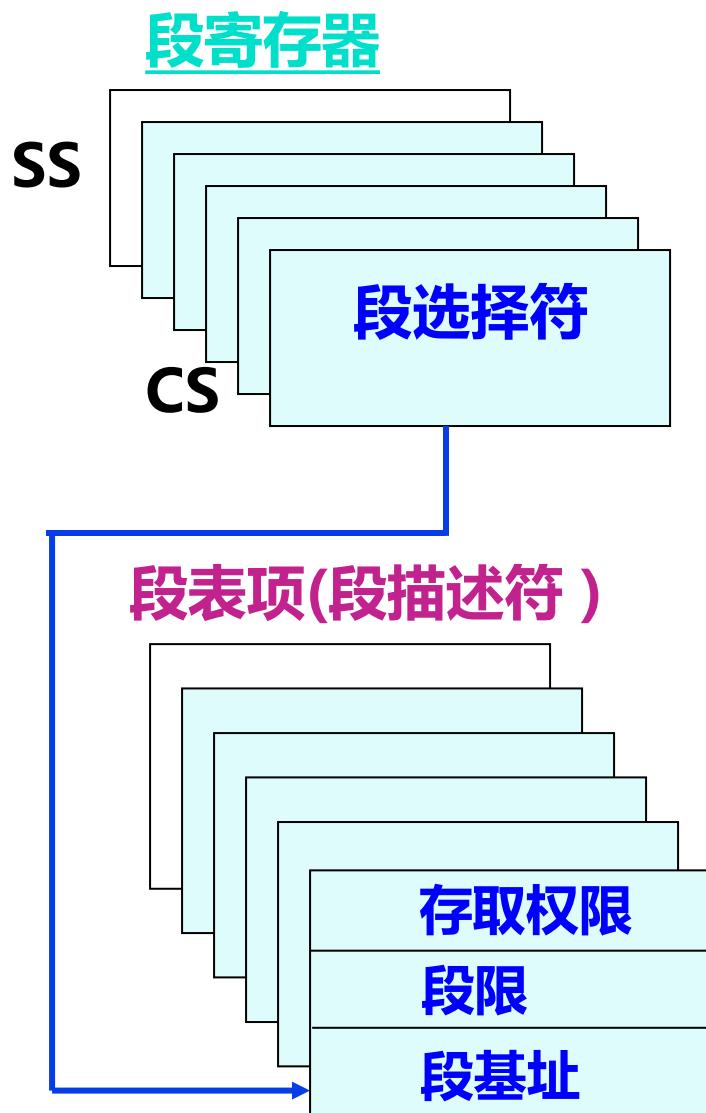
寻址方式	算法
立即(地址码A本身为操作数)	$\text{操作数} = A$
寄存器(通用寄存器的内容为操作数)	$\text{操作数} = (R)$
偏移量(地址码A给出8/16/32位偏移量)	$LA = (SR) + A$
基址(地址码B给出基址器编号)	$LA = (SR) + (B)$
基址带偏移量(一维表访问)	$LA = (SR) + (B) + A$
比例变址带偏移量(一维表访问)	$LA = (SR) + (I) \times S + A$
基址带变址和偏移量(二维表访问)	$LA = (SR) + (B) + (I) + A$
基址带比例变址和偏移量(二维表访问)	$LA = (SR) + (B) + (I) \times S + A$
相对(给出下一指令的地址, 转移控制)	$\text{转移地址} = (PC) + A$

有效地址EA

IA-32指令举例:

`movw 8(%ebp,%edx,4), %ax // R[ax]←M[R[ebp]+R[edx]×4+8]`

IA-32处理器的存储器寻址



IA-32的寄存器组织

	31	16	15	8	7	0	
EAX			AH	(AX)	AL		累加器
EBX			BH	(BX)	BL		基址寄存器
ECX			CH	(CX)	CL		计数寄存器
EDX			DH	(DX)	DL		数据寄存器
ESP				SP			堆栈指针
EBP				BP			基址指针
ESI				SI			源变址寄存器
EDI				DI			目标变址寄存器
EIP				IP			
EFLAGS				FLAGS			
8个通用寄存器		CS		代码段			
两个专用寄存器		SS		堆栈段			
6个段寄存器		DS		数据段			
		ES		附加段			
		FS		附加段			
		GS		附加段			

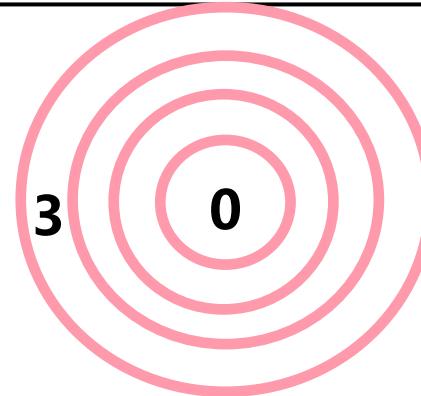
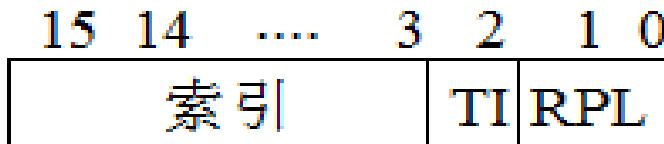
[BACK](#)

段选择符和段寄存器

◦ **段寄存器** (16位) , 用于存放段选择符

- CS(代码段): 程序代码所在段
- SS(栈段): 栈区所在段
- DS(数据段): 全局静态数据区所在段
- 其他3个段寄存器ES、GS和FS可指向任意数据段

◦ 段选择符各字段含义:



环保保护: 内核工作在0环, 用户工作在3环, 中间环留给中间软件用。Linux仅用第0和第3环。

CS寄存器中的RPL字段表示CPU的**当前特权级** (Current Privilege Level, CPL)

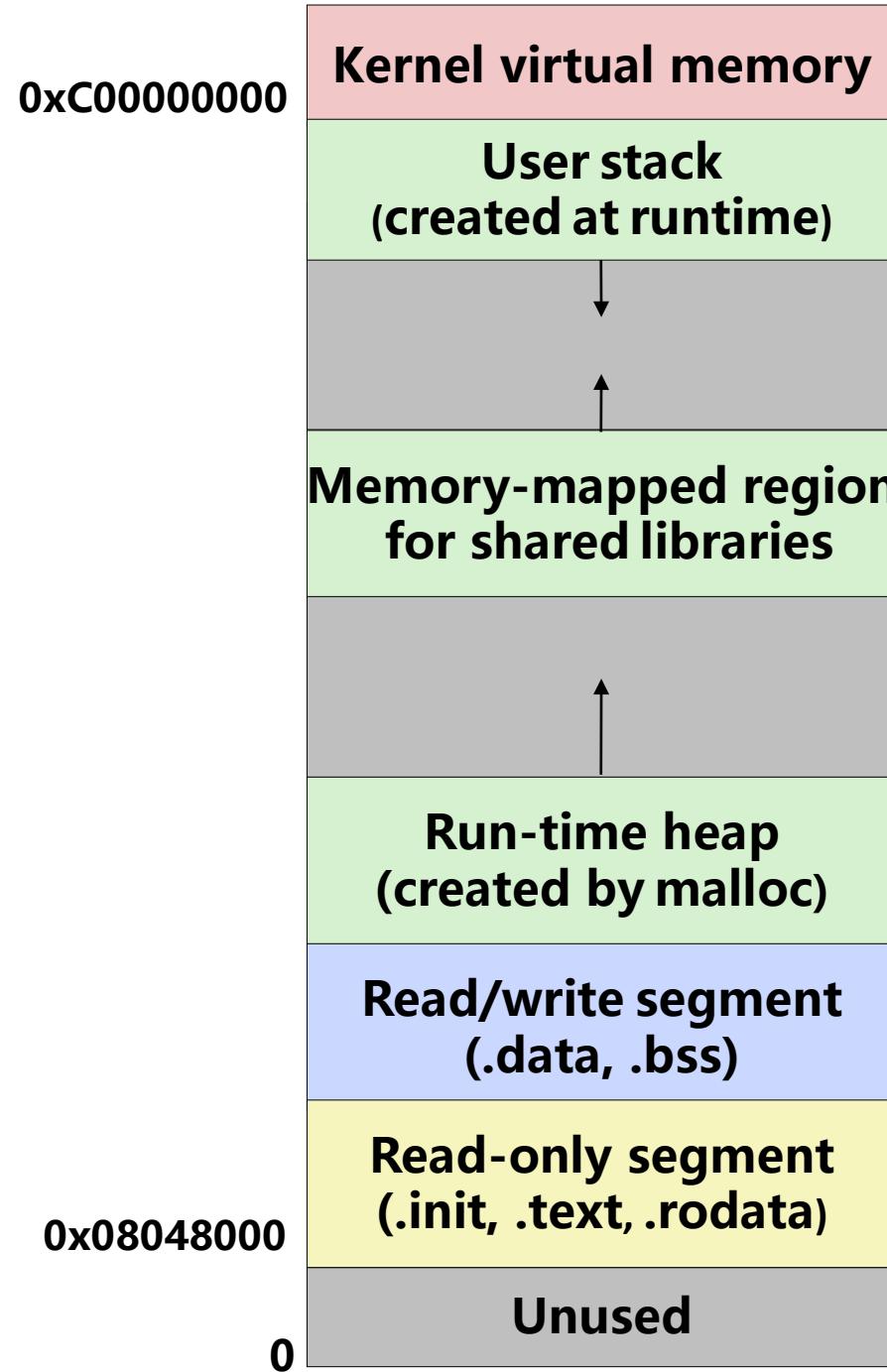
- TI=0, 选择**全局描述符表(GDT)**, TI=1, 选择**局部描述符表(LDT)**
- RPL=00, 为第0级, 位于最高级的内核态, RPL=11, 为第3级, 位于最低级的用户态, 第0级高于第3级
- 高13位索引用来确定当前使用的**段描述符**在描述符表中的位置

你认为什么是段描述符?

段表项一段的描述信息

SKIP

段寄存器的含义



%esp
(栈顶)

SS (栈段寄存器)

段内再分页

ES/GS/FS (辅助段寄存器)

brk

DS (数据段寄存器)

CS (代码段寄存器)

BACK

段式虚拟存储器的地址映像



段描述符和段描述符表

- 段描述符是一种数据结构，实际上就是段表项，分两类：
 - 普通段：用户/内核的代码段和数据段描述符
 - 系统控制段，又分两种：
 - 特殊系统控制段描述符，包括：局部描述符表（LDT）描述符和任务状态段（TSS）描述符
 - 控制转移类描述符，包括：调用门描述符、任务门描述符、中断门描述符和陷阱门描述符
- 描述符表实际上就是段表，由段描述符（段表项）组成。有三种类型：
 - 全局描述符表GDT：只有一个，用来存放系统内每个任务共用的描述符，例如，内核代码段、内核数据段以及TSS（任务状态段）等都属于GDT中描述的段
 - 局部描述符表LDT：存放某任务（即用户进程）专用的描述符
 - 中断描述符表IDT：包含256个中断门、陷阱门和任务门描述符

IDT将在第7章介绍

段描述符 (8B) 的定义

15	8	7	6	5	4	3	2	1	0
7	段地址 (B31-B24)	G	D	0	AVL	限界 (L19-L16)	6	4	2
5	P DPL S TYPE A	段地址 (B23-B16)	3	段地址 (B15-B0)	因CPL和DPL 的数值越大，等 级越低	1	限界 (L15-L0)	0	0
当CPL>DPL时，说明当前特权级比所要求的最低等级更低，故访问越级									

- B31~B0: 32位地址； L19~L0: 20位限界，表示段中最大页号
- G: 粒度。G=1以页 (4KB) 为单位； G=0以字节为单位。因为界限为20位，故当G=0时最大的段为1MB；当G=1时，最大段为 $4KB \times 2^{20} = 4GB$
- D: D=1表示段内偏移量为32位宽，D=0表示段内偏移量为16位宽
- P: P=1表示存在，P=0表示不存在。Linux总把P置1，不会以段为单位淘汰
- DPL: 访问段时对当前特权级的最低等级要求。因此，只有CPL为0 (内核态) 时才可访问DPL为0的段，任何进程都可访问DPL为3的段 (0最高、3最低)
- S: S=0系统控制描述符，S=1普通的代码段或数据段描述符
- TYPE: 段的访问权限或系统控制描述符类型
- A: A=1已被访问过，A=0未被访问过。 (通常A包含在TYPE字段中)

用户不可见寄存器

- 为支持分段机制，CPU中有多个用户不可访问的内部寄存器，操作系统通过特权指令可对寄存器TR、LDTR、GDTR和IDTR进行读写

段寄存器

CS	0x60/73
SS	
DS	0x68/7B
ES	
FS	
GS	

描述符 cache

基地址	界限	访问权限

每次段寄存器装入新选择符时，新描述符装入描述符cache，在逻辑地址到线性地址转换时，MMU直接用描述符cache中的信息，不必访问主存段表

TR(任务寄存器)存放TSS描述符的段选择符

LDTR(LDT寄存器)存放LDT描述符的段选择符

TSS描述符和LDT描述符在GDT中

TR	0x80
LDTR	0x88

基础TSS首地址	界限	访问权限
LDT首地址		

GDTR	GDT首地址
IDTR	

界限

GDT和IDT只有一个，GDTR和IDTR指向各自起始处。例如，根据TR取GDT中的TSS描述符时，GDTR给出首址

Linux的全局描述符表 (GDT)

Linux 全局描述符表

null	段选择符 0x0
reserved	
reserved	
0000 0000 1000 0000	
说明TR所指段TSS处于第0环，其描述符在GDT中，索引值为0x0010	
0000 0000 1000 1000	
说明LDTR所指段LDT处于第0环，其描述符在GDT中，索引值为0x0011	
TL3 #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (<u> KERNEL_CS</u>)
kernel data	0x68 (<u> KERNEL_DS</u>)
user code	0x73 (<u> USER_CS</u>)
user data	0x7b (<u> USER_DS</u>)

段选择符

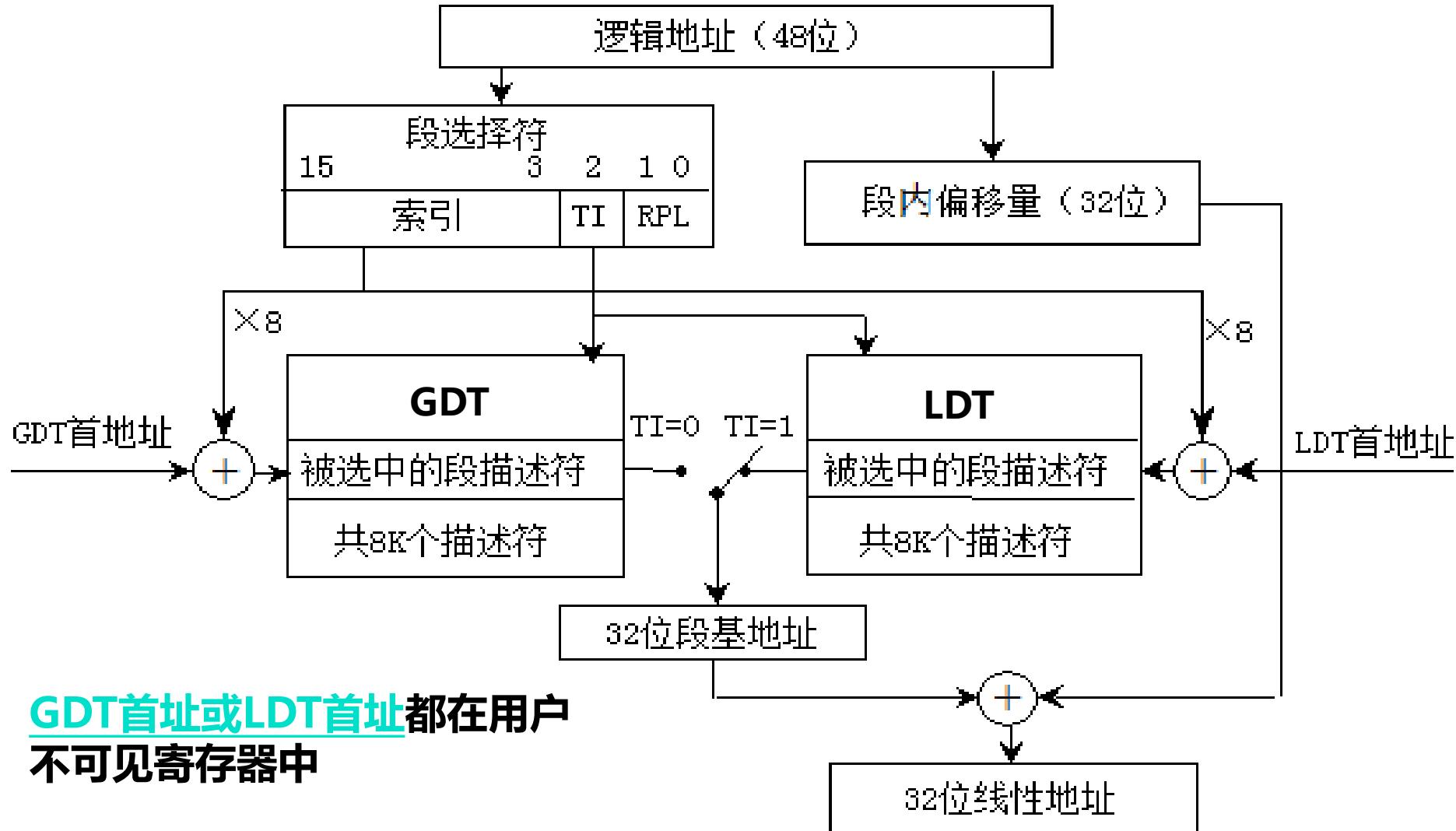
TR中为80H
LDTR中为88H

Linux 全局描述符表

TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
double fault TSS	0xf8

逻辑地址向线性地址转换

- 被选中的段描述符先被送至描述符cache，每次从描述符cache中取32位段基址，与32位段内偏移量（有效地址）相加得到线性地址



IA-32/Linux中的分段机制

- 为使能移植到绝大多数流行处理器平台，Linux简化了分段机制
- RISC对分段支持非常有限，因此Linux仅使用IA-32的分页机制，而对于分段，则通过在初始化时将所有段描述符的基址设为0来简化
- 若把运行在用户态的所有Linux进程的代码段和数据段分别称为用户代码段和用户数据段；把运行在内核态的内核代码段和数据段分别称为内核代码段和内核数据段，则Linux初始化时，将上述4个段的段描述符中各字段设置成下表中的信息：

段	基地址	G	界限	S	TYPE	DPL	D	P
用户代码段	0x0000 0000	1	0xFFFFFFF	1	10	3	1	1
用户数据段	0x0000 0000	1	0xFFFFFFF	1	2	3	1	1
内核代码段	0x0000 0000	1	0xFFFFFFF	1	10	0	1	1
内核数据段	0x0000 0000	1	0xFFFFFFF	1	2	0	1	1

每个段都被初始化在
0~4GB的线性地址空间中 初始化时，上述4个段描述符被存放在GDT中

Linux的全局描述符表（GDT）

[BACK](#)

Linux 全局描述符表 段选择符

0000 0000 0110 0000

说明内核代码段处于第0环，其描述符在GDT中，索引值为0x000C

0000 0000 0110 1000

说明内核数据段处于第0环，其描述符在GDT中，索引值为0x000D

0000 0000 0111 0011

说明用户代码段处于第3环，其描述符在GDT中，索引值为0x000E

0000 0000 0111 1011

说明用户数据段处于第3环，其描述符在GDT中，索引值为0x000F

reserved

kernel code 0x60 (__KERNEL_CS)

kernel data 0x68 (__KERNEL_DS)

user code 0x73 (__USER_CS)

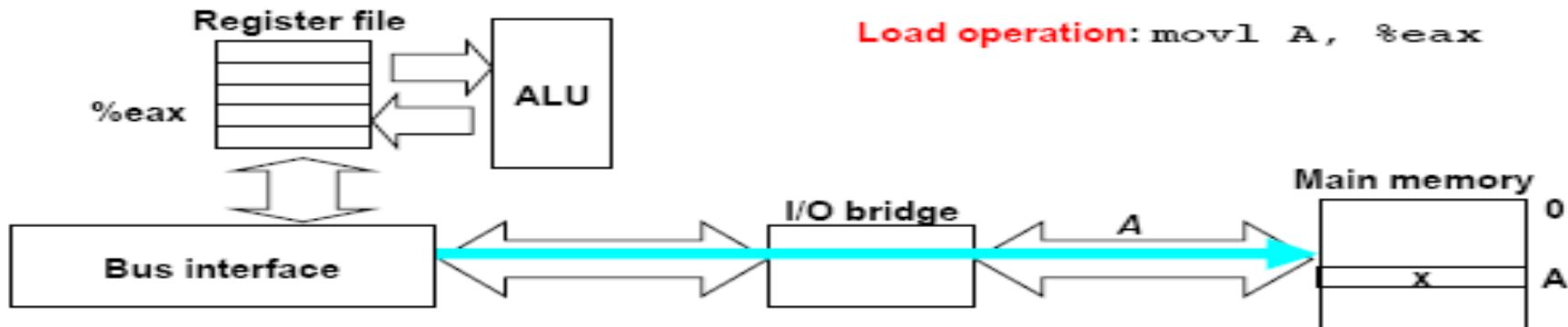
user data 0x7b (__USER_DS)

Linux 全局描述符表 段选择符

TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
double fault TSS	0xf8

回顾：指令“`movl 8(%ebp), %eax`”操作过程

由`8(%ebp)`得到主存地址A的过程较复杂，涉及MMU、TLB、页表等许多重要概念！



- IA-32/Linux中，执行“`movl 8(%ebp), %eax`”时，源操作数的逻辑地址向线性地址转换的过程如下：
 - 计算有效地址 $EA = R[ebp] + 0 \times 0 + 8$
 - 取出段寄存器 DS 对应的描述符 cache 中的段基址（Linux 中段基址为 0）
 - 线性地址 $LA = \text{有效地址 } EA$ ！

逻辑地址向线性地址转换举例

- 已知变量y和数组a都是int型，a的首地址为0x8048a00。假设编译器将a的首地址分配在ECX中，数组的下标变量i分配在EDX中，y分配在EAX中，C语言赋值语句“y=a[i];”被编译为指令“movl (%ecx, %edx, 4), %eax”。若在IA-32/Linux环境下执行指令地址为0x80483c8的该指令时，CS段寄存器对应的描述符cache中存放的是表6.2中所示的用户代码段信息且CPL=3，DS段寄存器对应的描述符cache中存放的是表6.2中所示的用户数据段信息，则当i=100时，取指令操作过程中MMU得到的指令的线性地址是多少？取数操作过程中MMU得到的操作数的线性地址是多少？

```
int func(int a[ ], int c)
{
    int i, y = 0;
    for(i = 0; i < c; i++) {
        y = a[i];
    }
    .....
}
```

$$\begin{aligned}400 &= 511 - 111 = 511 - (64 + 32 + 15) \\&= 1\ 1111\ 1111B - (0110\ 1111B) \\&= 1\ 1001\ 0000B = 190H\end{aligned}$$

…代码和数据段DPL都为3，即CPL最低应为3，而CPL=3，故访问未越级
指令的线性地址：代码段基地址+EA=0+0x80483c8=0x80483c8

操作数的线性地址：数据段基地址+EA=0+R[ecx]+R[edx]×4

$0x8048b90 = 0x8048a00 + 100 \times 4 = \underline{\underline{0x8048e00}}$ 对吗？

IA-32的存储管理

- 按字节编址（通用计算机大都是）
- 在保护模式下，IA-32采用**段页式**虚拟存储管理方式
- 存储地址采用逻辑地址、线性地址和物理地址来进行描述，其中，
逻辑地址和线性地址是虚拟地址的两种不同表示形式，描述的都是
4GB虚拟地址空间中的一个存储地址
 - ✓ 逻辑地址由48位组成，包含16位段选择符和32位段内偏移量（即有效地址）
 - ✓ 线性地址32位（其位数由虚拟地址空间大小决定）
 - ✓ 物理地址32位（其位数由存储器总线中的地址线条数决定）
- 分段过程实现将逻辑地址转换为线性地址
- 分页过程实现将线性地址转换为物理地址 ←———— 以下介绍分页机制

若页大小为4KB，每个页表项占4B，则理论上一个页表有多大？

因为 $2^{32}/2^{12}=2^{20}$ ，故页表大小为4MB，比页还大！故采用多级页表方式

IA-32中的控制寄存器

- 控制寄存器保存机器的各种控制和状态信息，它们将影响系统所有任务的运行，操作系统进行任务控制或存储管理时使用这些控制和状态信息。

- CR0：控制寄存器

- ① PE: 1-保护模式；0-实地址模式。② PG: 1-启用分页；0-禁止分页，此时线性地址被直接作为物理地址使用。若要启用分页机制，则PE和PG都要置1。③任务切换位TS：任务切换时将其置1，切换完毕则清0，可用CLTS指令将其清0。④ 对齐屏蔽位AM。⑤ cache功能控制位NW（(Not Write-through) 和CD (Cache Disable)）。只有当NW和CD均为0时，cache才能工作。

- CR2：页故障 (page fault) 线性地址寄存器

- 存放引起页故障的线性地址。只有在CR0中的PG=1时，CR2才有效。

- CR3：页目录基址寄存器

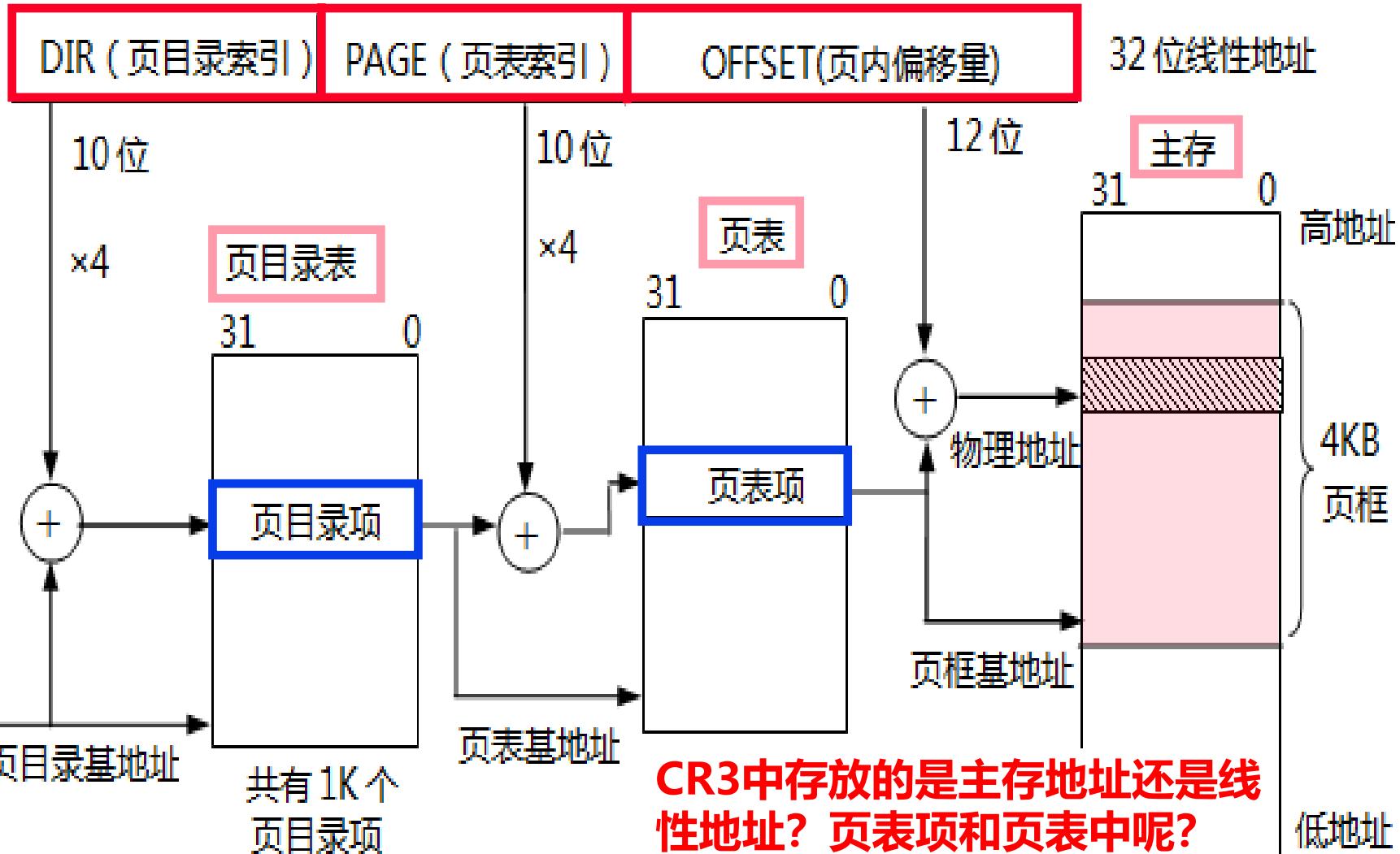
- 保存页目录表的起始物理地址。只有CR0中的PG=1时，CR3才有效。

IA-32采用两级页表方式，第一级页表称为页目录表

线性地址向物理地址转换

线性地址空间划分: $4GB = 1K\text{个子空间} * 1K\text{个页/子空间} * 4KB/\text{页}$

- ° 页目录项和页表项格式一样, 有32位 (4B)



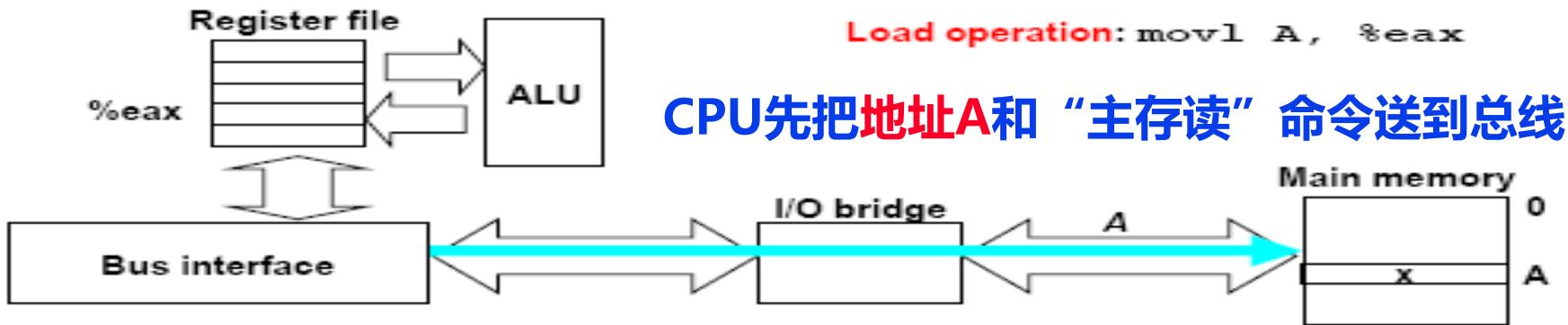
IA-32的页目录项和页表项

31	12	11	10	09	8	7	6	5	4	3	2	1	0
基地址	AVL	0	0	D	A	PCD	PWT	U/S	R/W	P			

- P: 1表示页表或页在主存中; P=0表示页表或页不在主存, 即缺页, 此时需将页故障线性地址保存到CR2。
- R/W: 0表示页表或页只能读不能写; 1表示可读可写。
- U/S: 0表示用户进程不能访问; 1表示允许访问。
- PWT: 控制页表或页的cache写策略是全写还是回写 (Write Back) 。
- PCD: 控制页表或页能否被缓存到cache中。
- A: 1表示指定页表或页被访问过, 初始化时OS将其清0。利用该标志, OS可清楚了解哪些页表或页正在使用, 一般选择长期未用的页或近来最少使用的页调出主存。由MMU在进行地址转换时将该位置1。
- D: 修改位(脏位dirty bit)。页目录项中无意义, 只在页表项中有意义。初始化时OS将其清0, 由MMU在进行写操作的地址转换时将该位置1。
- 高20位是页表或页在主存中的首地址对应的页框号, 即首地址的高20位。
每个页表的起始位置都按4KB对齐。

回顾：指令“`movl 8(%ebp), %eax`”操作过程

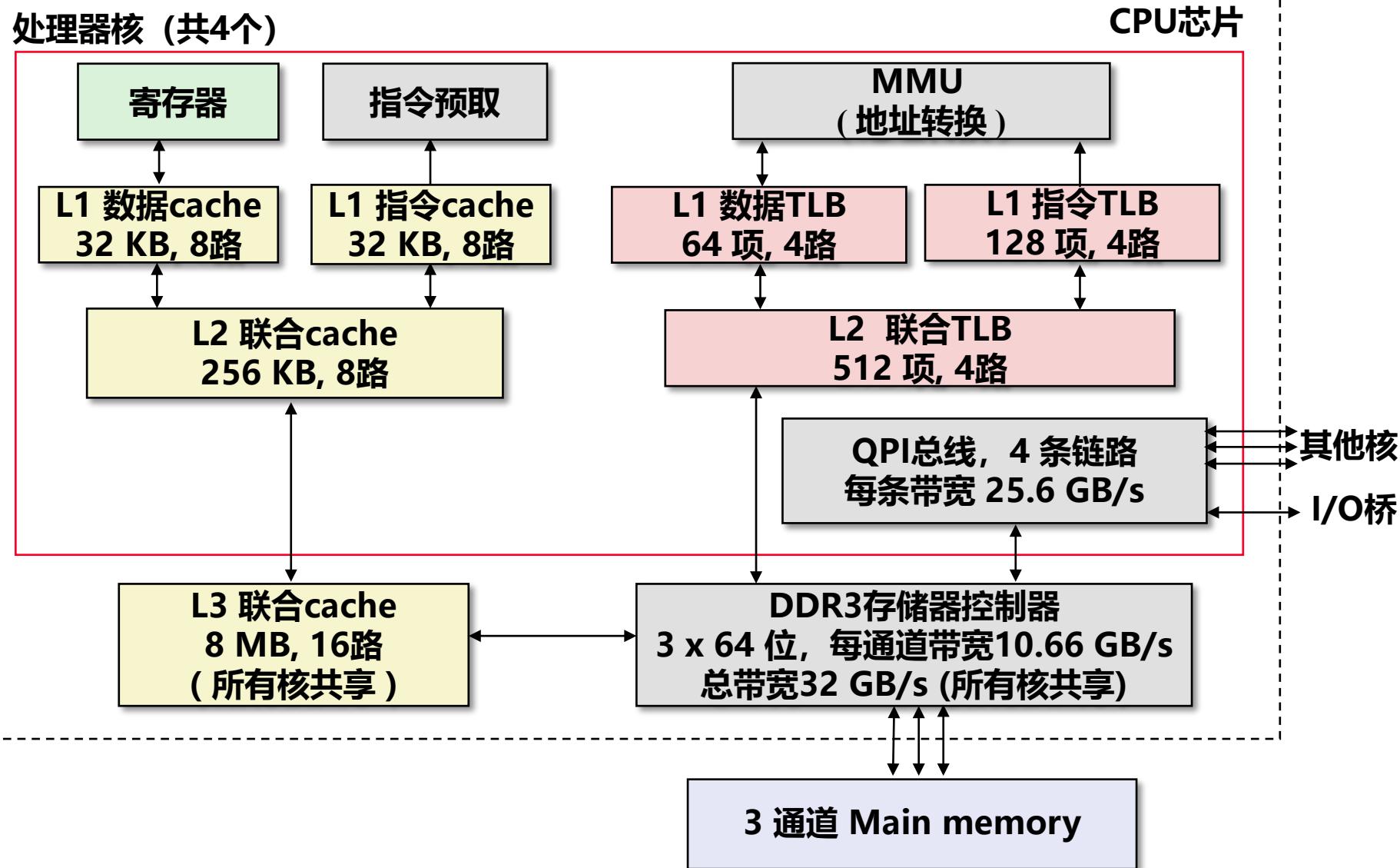
由`8(%ebp)`得到主存地址A的过程较复杂，涉及MMU、TLB、页表等许多重要概念！



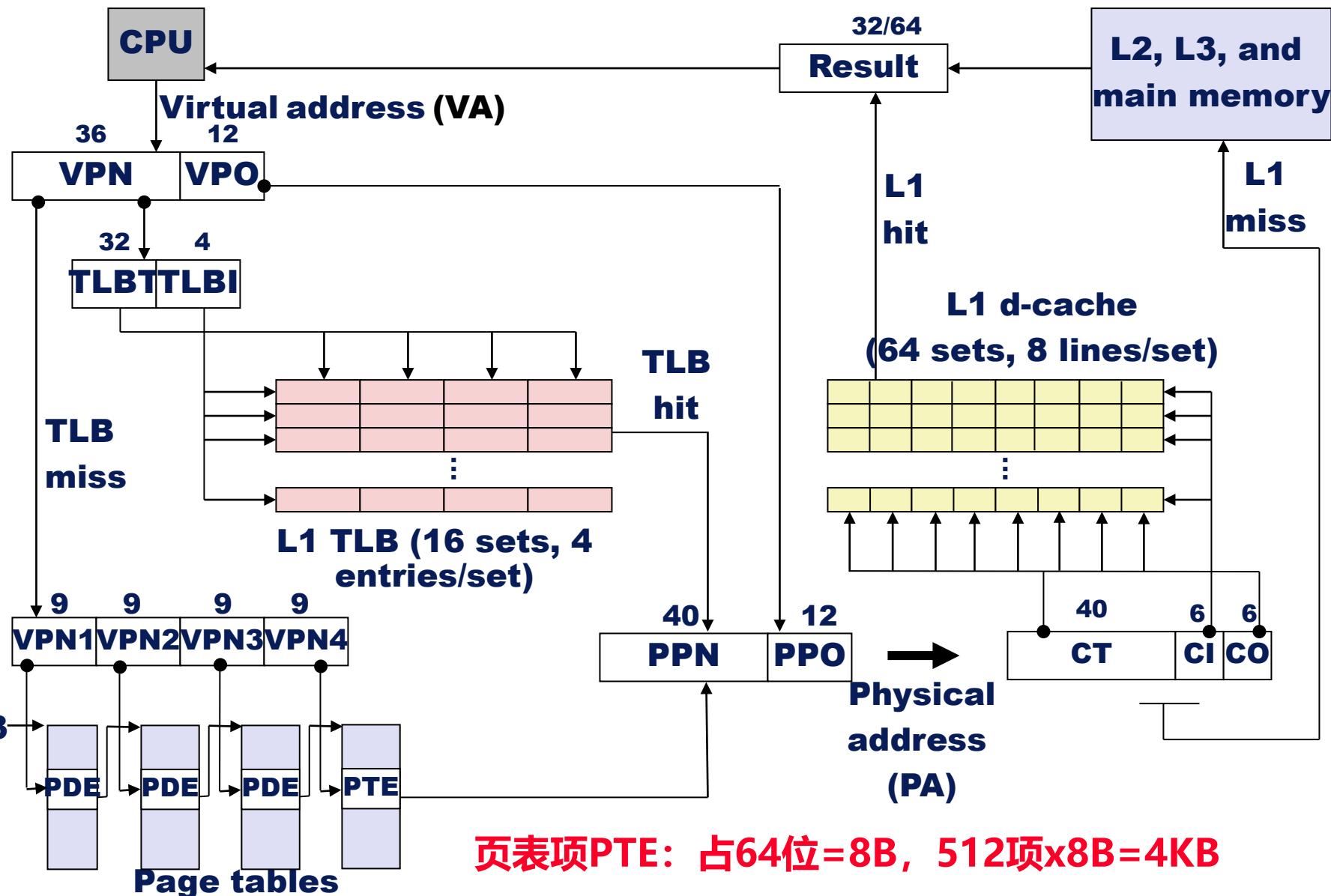
- IA-32中，执行“`movl 8(%ebp), %eax`”中**取数操作**的大致过程如下：
 - 若**CPL>DPL**则**越级**，否则计算**有效地址EA=R[ebp]+0×0+8**
 - 通过段寄存器找到段描述符以获得**段基址**，线性地址**LA=段基址+EA**
 - 若“**LA>段限**”则**越界**，否则将**LA转换为主存地址A**
 - 若访问**TLB命中**则**地址转换**得到**A**；否则处理**TLB缺失**（硬件/OS）
 - 若**缺页或越权(R/W不符)**则调出**OS内核**；否则**地址转换**得到**A**
 - 根据**A**先到**Cache**中找，若**命中**则取出**A**在**Cache**中的**副本**
 - 若**Cache不命中**，则再到**主存**取**A**所在**主存块**送对应**Cache行**

实例：Intel Core i7+Linux存储系统

处理器核 (共4个)



End-to-end Core i7 Address Translation



Core i7 Level 1-3 Page Table Entries

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	未使用	下级页表的主存物理基址			未使用	G	PS		A	CD	WT	U/S	R/W	P=1	
OS使用 (下级页表在硬盘上的位置)															P=0

Each entry references a 4KB child page table

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

G: Global page (don't evict from TLB on task switch)

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

Core i7 Level 4 Page Table Entries

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	未使用	虚页的主存物理基址(页框号)			未使用	G		D	A	CD	WT	U/S	R/W	P=1	
OS使用 (虚拟页在硬盘上的位置)															P=0

Each entry references a 4KB child page

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

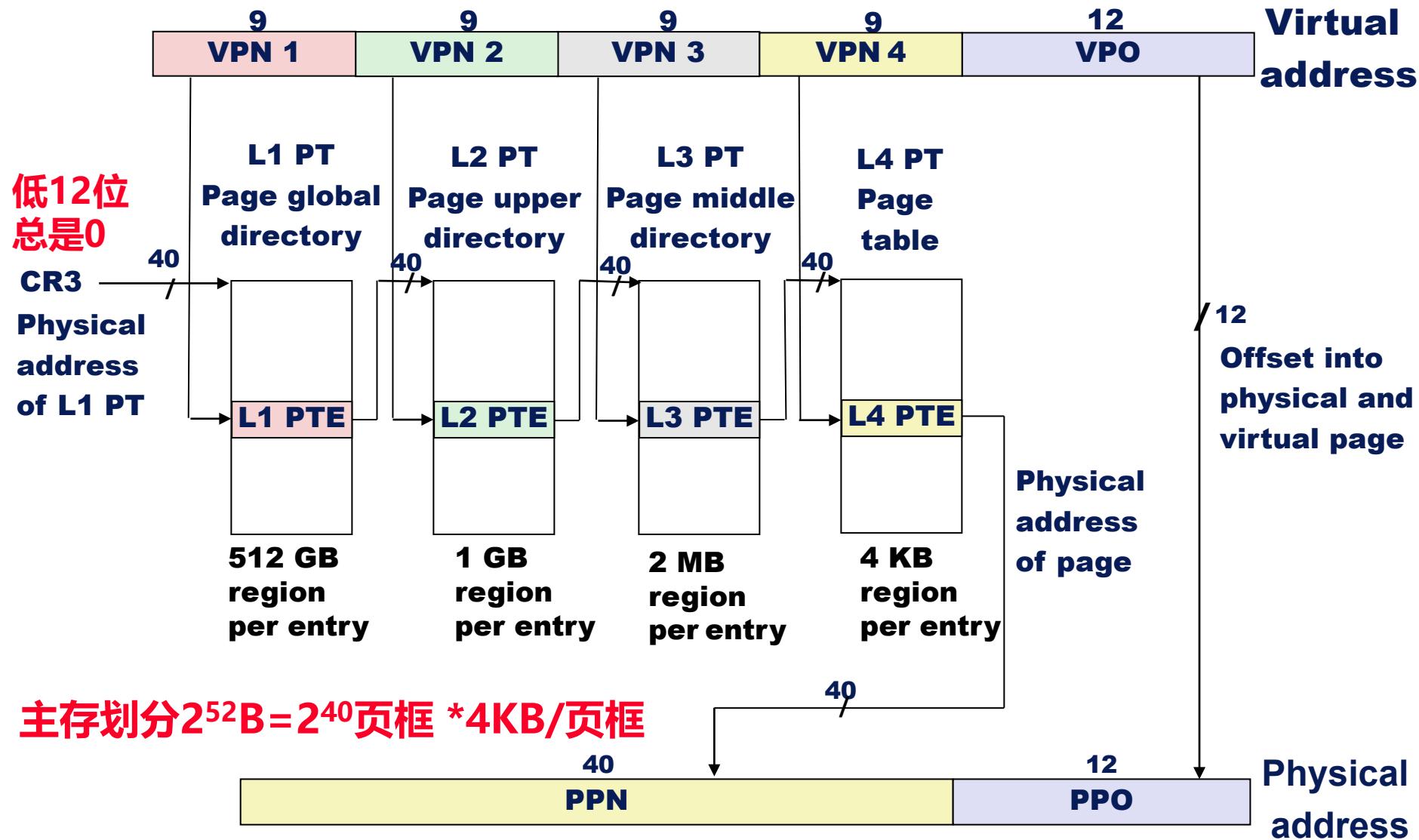
D: Dirty bit (set by MMU on writes, cleared by software)

G: Global page (don't evict from TLB on task switch)

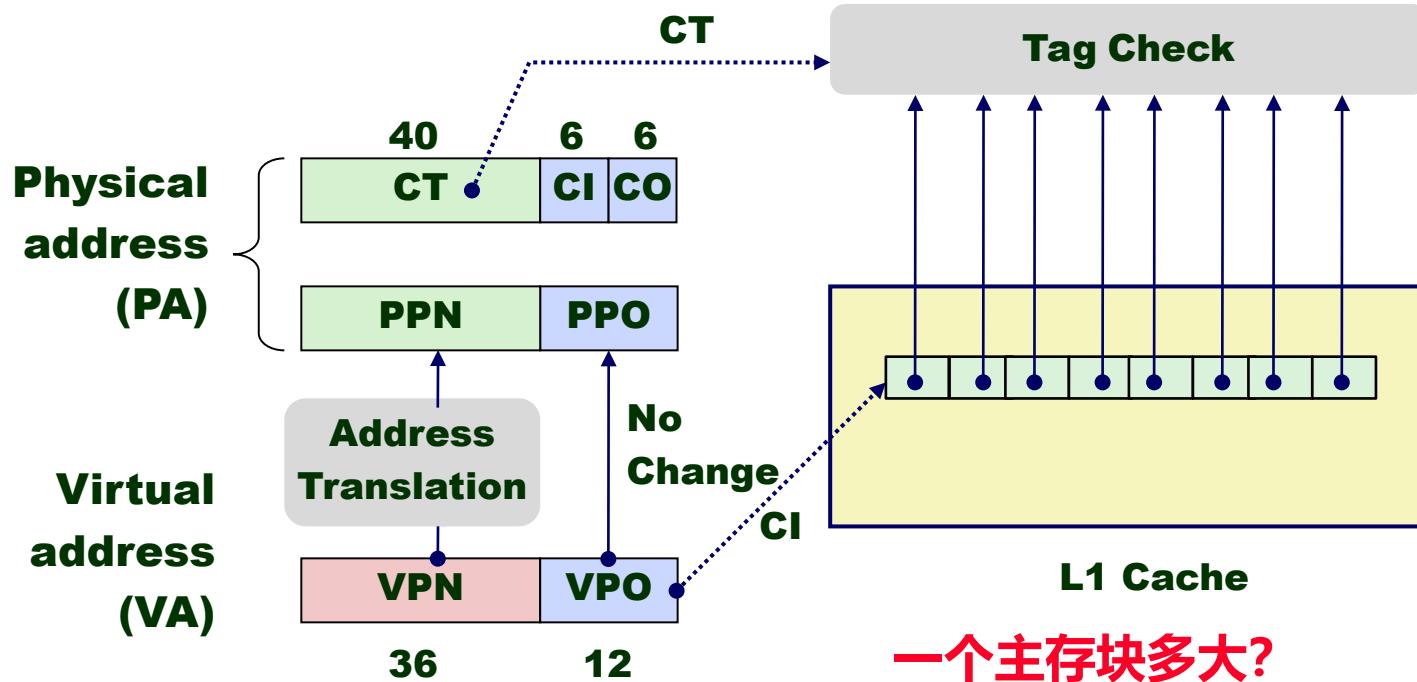
Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

Core i7 Page Table Translation

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



Cute Trick for Speeding Up L1 Access



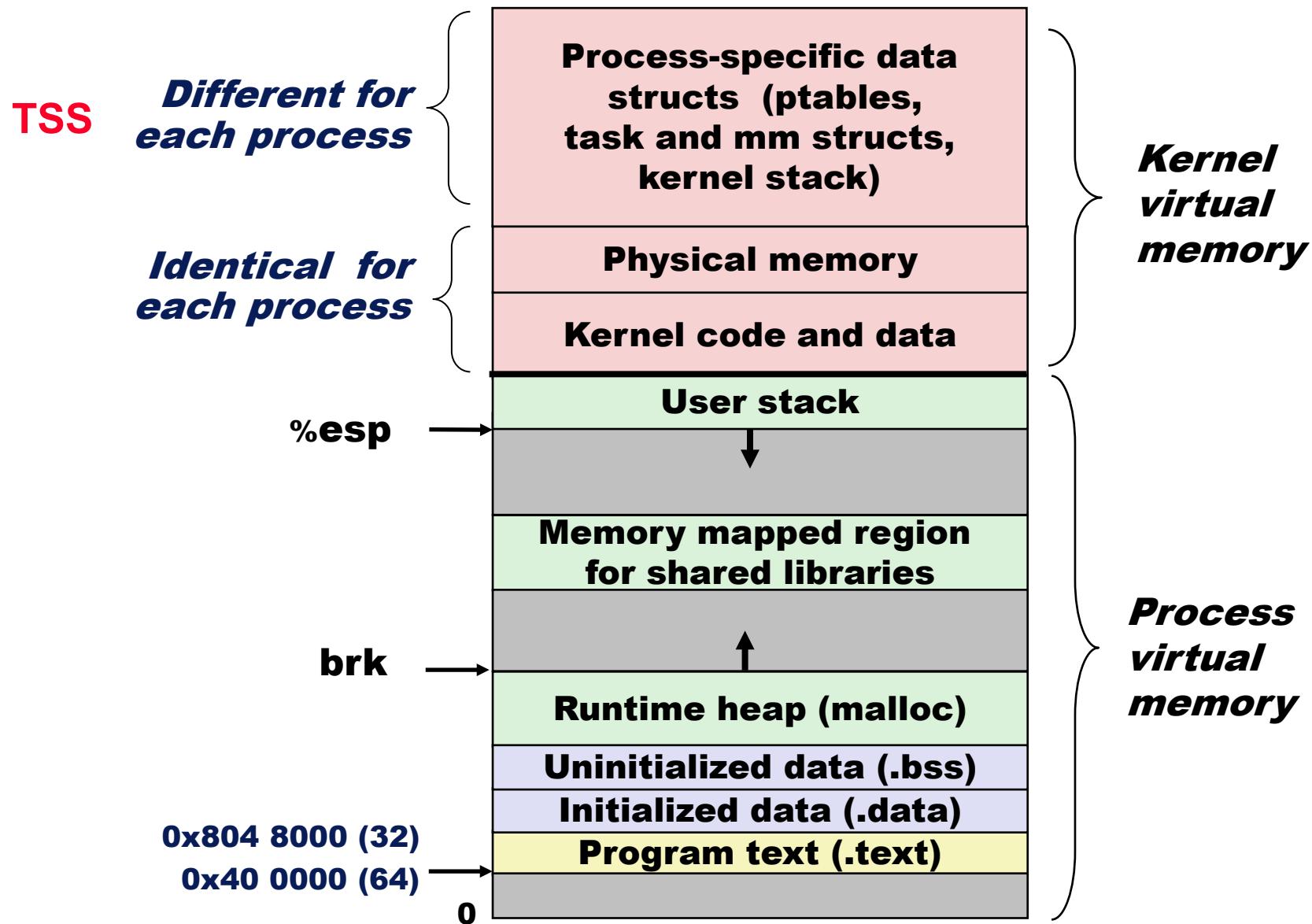
一个主存块多大?

64B

- **Observation**

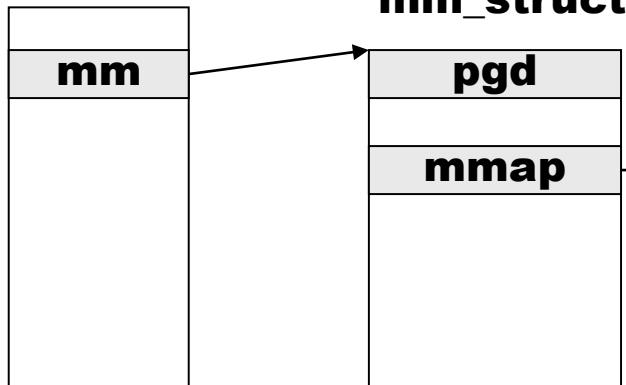
- Bits that determine **CI** identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (**CT** bits) available next
- “Virtually indexed, physically tagged”
- Cache carefully sized to make this possible

Virtual Memory of a Linux Process

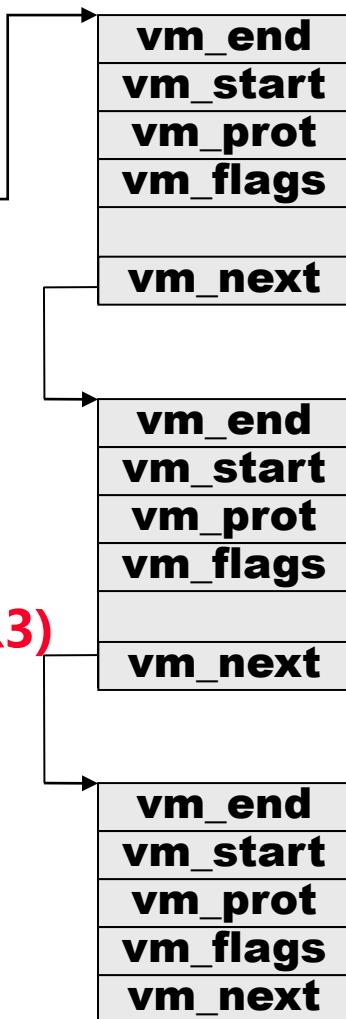


Linux将虚存空间组织成“区域(area)”

`task_struct`



`vm_area_struct`



进程虚拟地址空间

◦ **pgd:** 全局页目录地址

- Page global directory address 指向L1页表(装入CR3)
- Points to L1 page table

◦ **vm_prot:**

- Read/write permissions for this area 访问权限

◦ **vm_flags**

- Pages shared with other processes or private to this process 是共享还是本进程私有

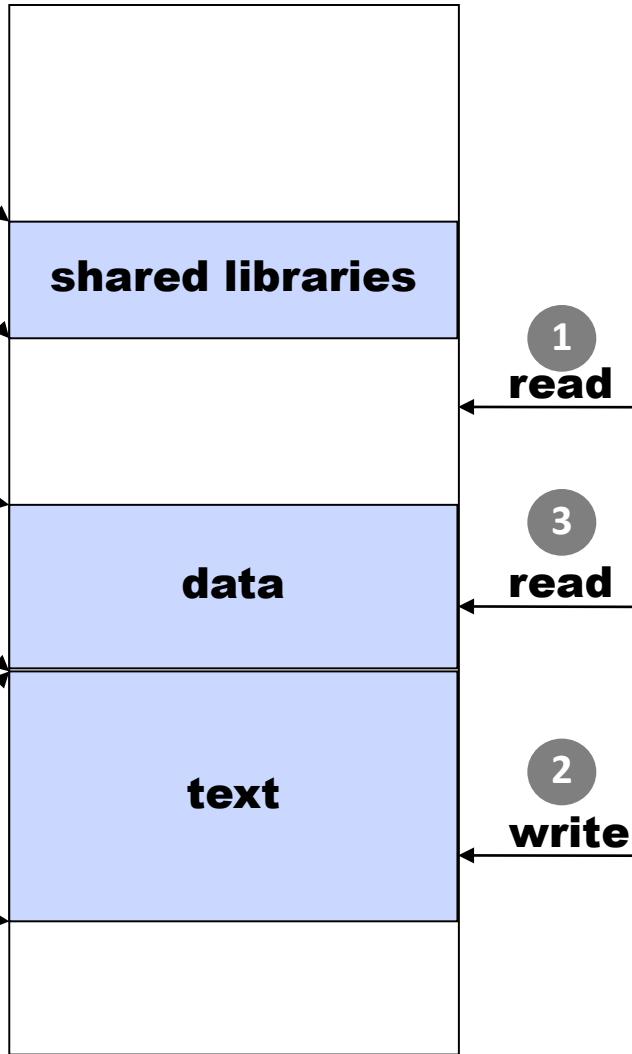
每个区域可被划分成若干个大小相等的虚拟页

Linux Page Fault Handling

vm_area_struct

vm_end
vm_start
vm_prot
vm_flags
vm_end
vm_start
vm_prot
vm_flags
vm_end
vm_start
vm_prot
vm_flags
vm_next

Process virtual memory



Linux页故障类型

1
read

3
read

2
write

Segmentation fault:
accessing a non-existing
page

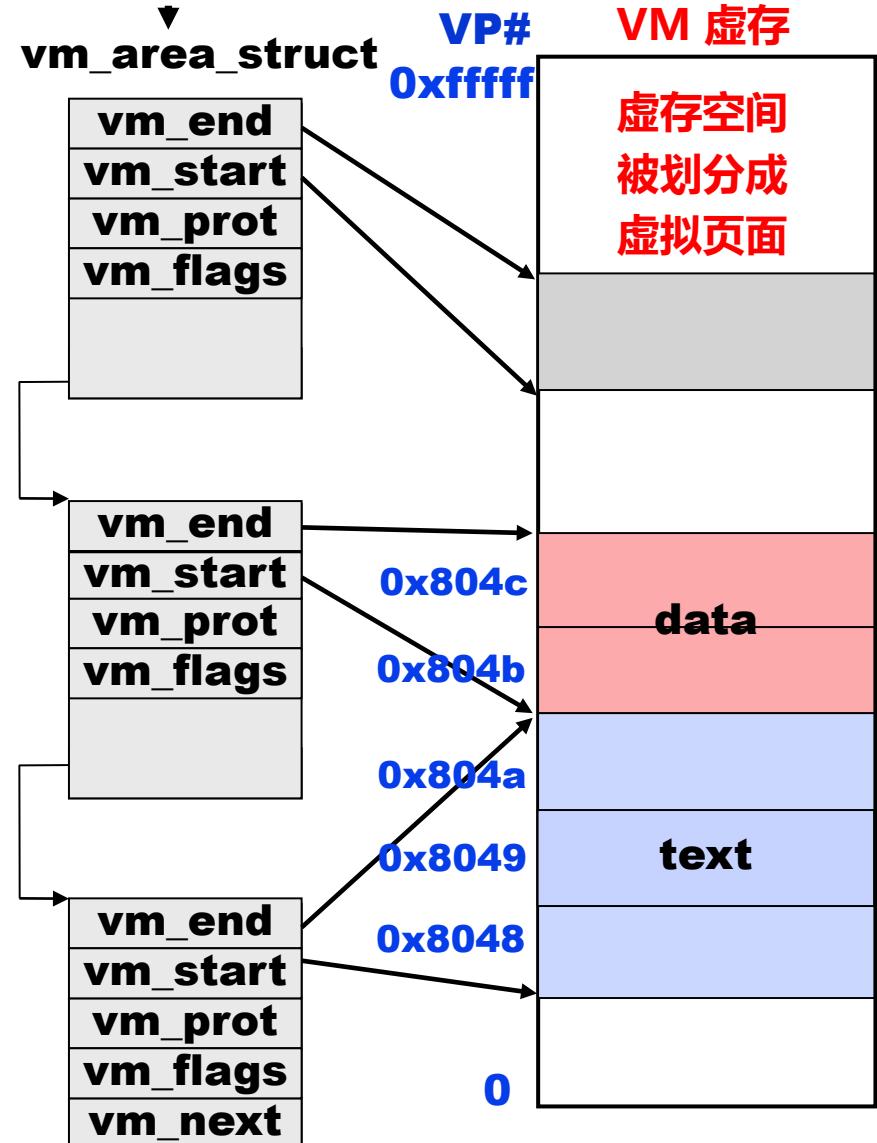
Normal page fault
not in memory

Protection exception:
e.g., violating permission
by writing to a read-only
page (Linux reports as
Segmentation fault)

存储管理全局图

task_struct 进程控制块

页大小: 4KB



可执行目标文件

0000

ELF 头

程序头表

.text 节

.rodata 节

....

.data 节

.bss 节

....

程序头表描述
可执行文件与
虚拟地址空间
之间的映射

页表描述虚拟
地址空间与主
存地址空间之
间的映射

MM 主存 PF#

0

1

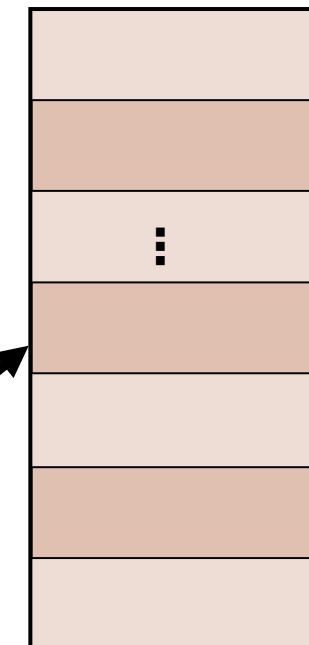
...

120

...

	VP#	P	PF#
0	0		Null
0x1			
0x2			
1			120

页表



本章小结

◦ 分以下六个部分介绍

- 第一讲：存储器概述
- 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：磁盘存储器
- 第四讲：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
- 第五讲：虚拟存储器 (Virtual Memory)
 - 虚拟地址空间、虚拟存储器的实现
- 第六讲：IA-32/Linux中的地址转换
 - 逻辑地址到线性地址的转换
 - 线性地址到物理地址的转换

设计支持Cache的存储器系统

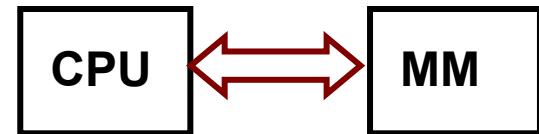
- 指令执行若发生Cache缺失，必须到DRAM中取数据或指令
- 在DRAM和Cache之间传输的单位是Block
- 问题：怎样的存储器组织使得Block传输最快（缺失损失最小）？

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

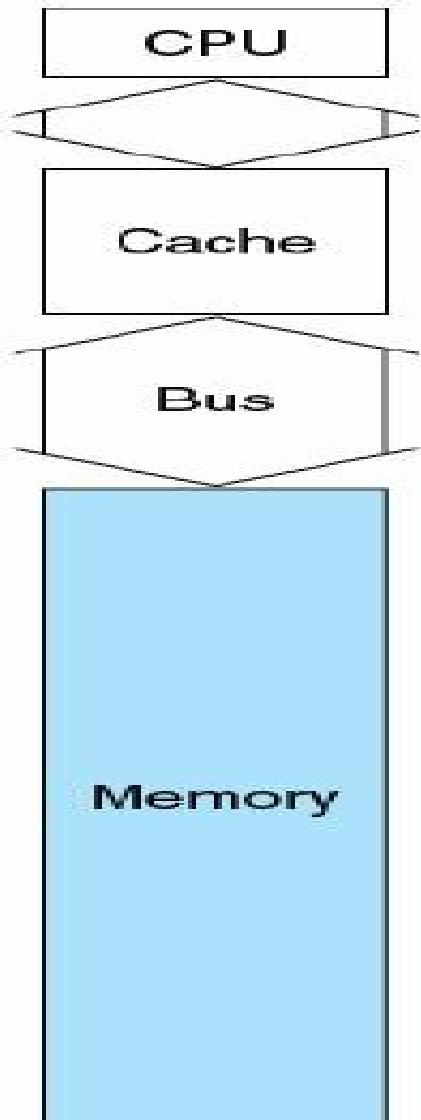
从总线上传送一个字：1个总线时钟



可以有三种不同的组织形式！

假定一个Block有4个字，则缺失损失各为多少时钟？

设计支持Cache的存储器系统



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

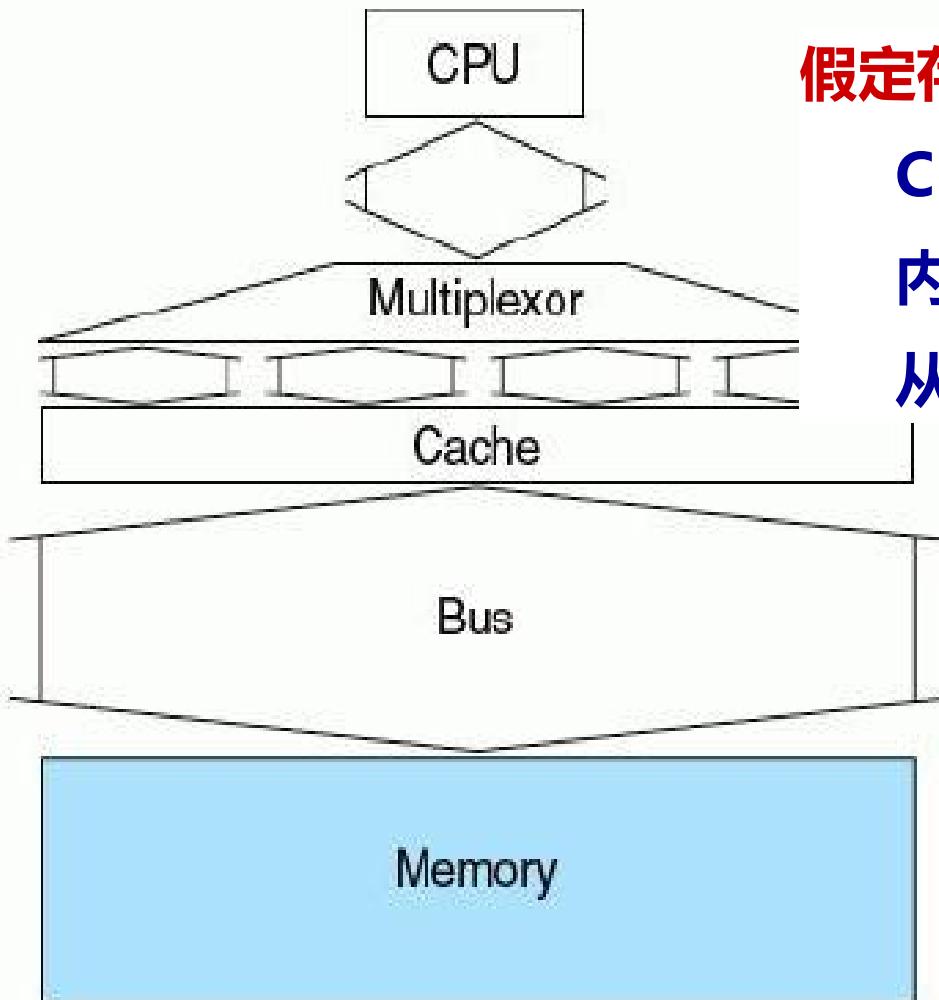
$$4 \times (1 + 10 + 1) = 48$$

缺失损失为48个时钟周期

代价小，但速度慢！

a. One-word-wide
memory organization

设计支持Cache的存储器系统



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

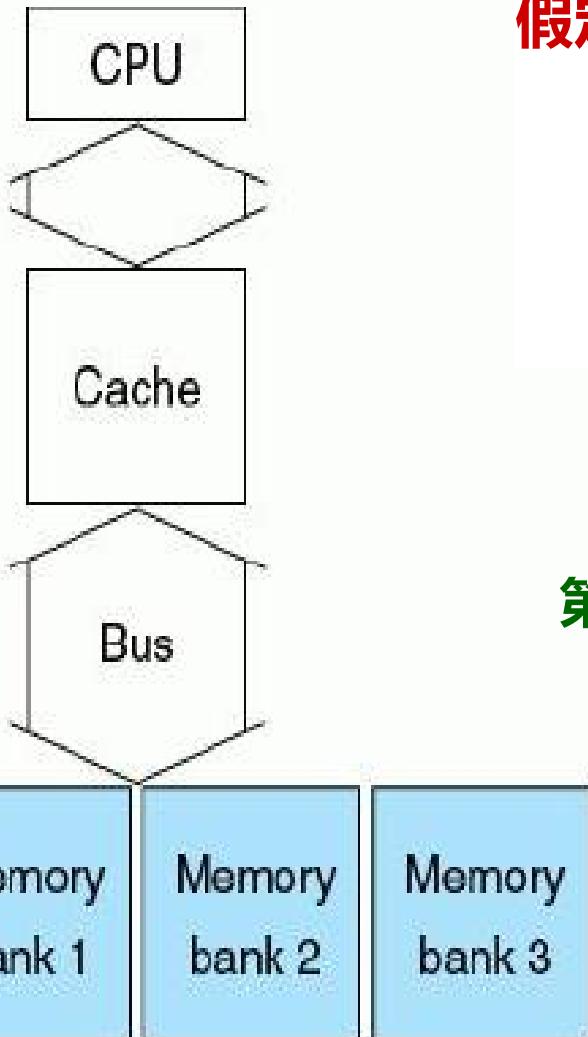
Two-word: $2 \times (1 + 10 + 1) = 24$

Four-word: $1 + 10 + 1 = 12$

**缺失损失各为24或12个时钟周期
速度快，但代价大！**

b. Wide memory organization

设计支持Cache的存储器系统



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

Interleaved four banks

one-word: $1+1\times10+4\times1=15$

第1个字



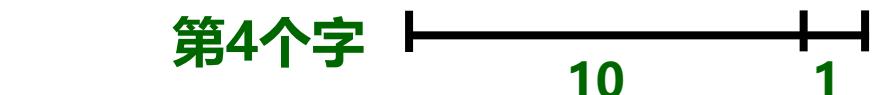
第2个字



第3个字



第4个字



缺失损失为15个时钟周期

代价小，而且速度快！

c. Interleaved memory organization