# Ch4: Memory Hierarchy

## 存储器层次结构

第一讲 基本概念和主存储器
第二讲 高速缓冲存储器（Cache）
第三讲 虚拟存储器（Virtual Memory）



## 第一讲 基本概念和主存储器

主 要 内 容

- 信息的存储、传送、处理单位的含义
  - 记忆单元 / 编址单位 / 存储单位 / 传输单位 / 机器字长
- 存储器分类
  - 可按存取方式 / 易失性 / 可更改性 / 元器件 / 功能来分
- 半导体存储器随机访问存储器
  - SRAM的原理和特点
  - DRAM的原理和特点
- RAM芯片组织
  - 如何由记忆单元构成存储阵列
  - 如何读写存储阵列中的信息
  - 如何由芯片构成存储器
- 提高存储器速度的措施：
  - 芯片内采用行缓存，同行内数据直接从缓存中取
  - 采用多模块存储器，多个存储器交叉存取
  - 引入Cache（下一讲的主要内容）



## 回顾：存储器基本术语

- 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）
  - 主存中具有相同地址的那些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）
  - 对存储体中各存储单元进行编号的方式
    - 按字节编址（基本上现代计算机都按字节编址）
    - 按字编址（早期有机器按字编址）
- 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(MBR) ）
  - 用于存放主存单元中的数据的寄存器



## 回顾：存储器基本术语

- 机器字长
  - 运算器中参加运算的寄存器的位数，即：数据通路的宽度
- 存储字
  - 存储芯片中的一个读写单位，一般等于芯片的数据线宽度

  （注：最好存储器按机器字长组织成一个“自然”单位。它的长度一般应等于一个数或指令的位数。但很多机器的数据和指令都是变长的。）
- 编址单位
  - 一个存储单元的位数。现在都按字节编址，即编址单位为8位
- 传输单位
  - 对主存而言，指一次从主存读出或写入的数的位数，它可以不等于存储字的长度，也可不等于编址单位。
  - 对外存而言，数据通常按块传输，传输单位为块。

（例如：386/486等，其编址单位为字节，字长为32位，单字位数为16位，但传输单位可以是8/16/24/32位。）



## 回顾：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)
  - 每个单元的读写时间一样，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)
  - 利用一个共享读写机制，直接定位到要读写的数据块，在读写某个数据块时按顺序进行。例如：磁盘。
- 相联存储器Associate Memory or Content Addressed Memory (CAM)
  - 按内容检索到存储位置进行读写。例如：快表。



## 回顾：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）

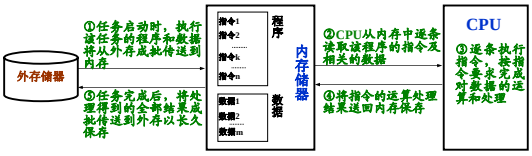
## 回顾：存储器分类

（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
  - 用磁表面或光存储器实现，容量大而速度慢



## 回顾：内存与外存的关系及比较

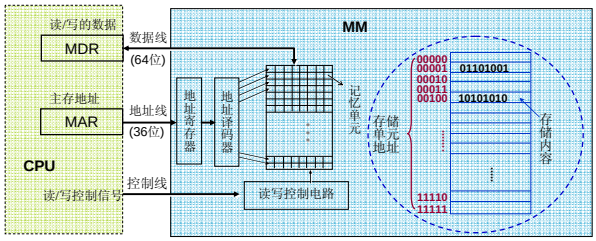



- 外存储器(简称外存或辅存)
  - 存取速度慢
  - 成本低、容量很大
  - 不与CPU直接连接，计算机运行程序时，外存中的程序及相关数据必须先传送到内存，然后才能被CPU使用。
  - 属于非易失性存储器(Nonvolatile)，用于长久存放系统中几乎所有的信息

- 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(指令)可以对内存中的指令及数据进行读、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



问题：主存中存放的是什么信息？CPU何时会访问主存？




- 主存是CPU可直接访问的存储器，用于存放供CPU处理的指令和数据
- 性能指标：
  - 以字节为单位进行连续编址，每个存储单元为1个字节（8个二进位）
  - 存储容量：主存储器中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 ns = $10^{-9}$ s）
  - 存储周期$T_{MC}$：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。 ）

## 存储容量和速度的单位

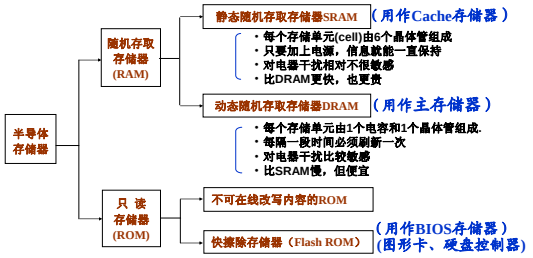
Notations and Conventions for Numbers

| 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* $2^{60}$ |



## 回顾：内存储器的分类及应用

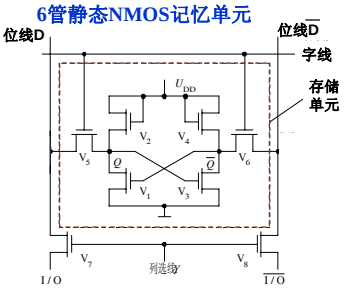
- 内存由半导体存储器芯片组成，芯片有多种类型：






## 回顾：六管静态MOS管电路

6管静态NMOS记忆单元



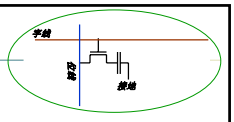

信息存储原理： 看作带时钟的RS触发器

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
- 根据存储单元的状态改变位线的输出电平

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，所以不是破坏性读出，也不需要重写数据来保持数据不变。即：无需刷新！

## 回顾：记忆单元的基本原理




- 动态单管MOS记忆单元电路

• 构造和表示：

数据记忆在电容$C_S$上，T为门控管，控制数据的进出。其栅极接读/写选择线(字线)，漏和源分别接数据线(位线)和记忆电容$C_S$ 。数据1或0以电容$C_S$上电荷量的有无来判别。

• 读写原理：在选择（字）线上加高电平，使T管导通。

写“0”时，在数据线上加低电平，使$C_S$上电荷对数据线放电；

写“1”时，在数据线上加高电平，使数据线对$C_S$充电；

读出时，在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，所以被广泛应用于大容量存储器中

• 缺点：

速度慢、是破坏性读出 (读后状态被改变，需读后再生)、需定时刷新

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2毫秒左右，相当于1000 000个时钟左右，因此要定期在2ms内刷新（读出后重新写回），按行进行刷新（所有芯片中的同一行一起进行），所以刷新操作所需时间只占1%~2%左右。



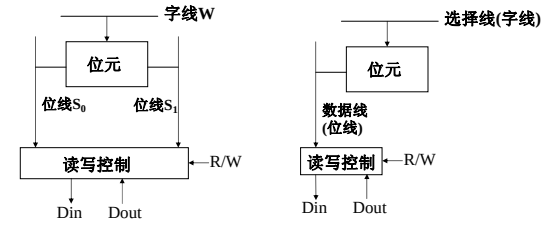
## 回顾：半导体RAM的组织

记忆单元(Cell)→存储器芯片(Chip)→ 内存条（存储器模块）

存储器芯片：存储体+外围电路(地址译码和读写控制)

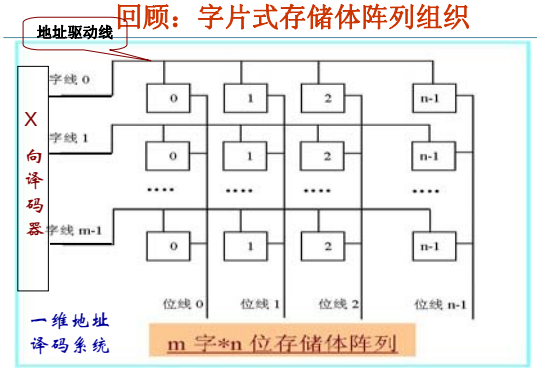
存储体(Memory Bank)： 由记忆单元(位元)构成的存储阵列

记忆单元的组织：






## 回顾：字片式存储体阵列组织

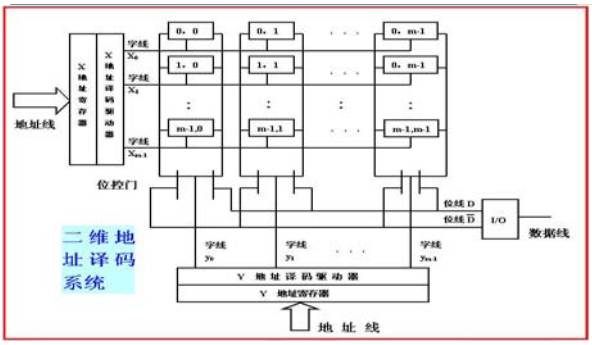



假定有m位地址，则地址译码驱动（选择）线的条数为多少？ 有$2^m$条！

一般SRAM为字片式芯片，只在单方向上译码，同时读出一条字线上的所有位！



## 回顾：位片式存储体阵列组织

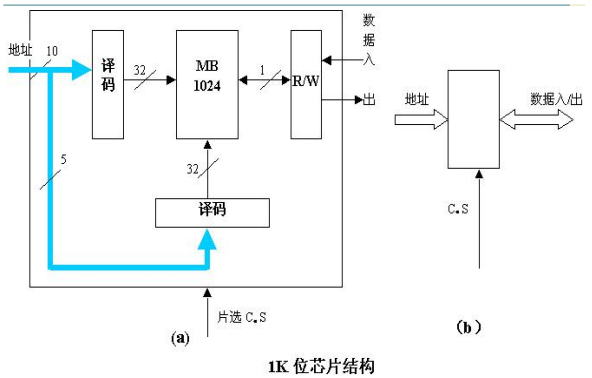



假定有m位地址，其地址译码驱动（选择）线的条数为多少？有$2^{m/2+1}$

位片式可在字方向和位方向扩充，需要有片选信号！



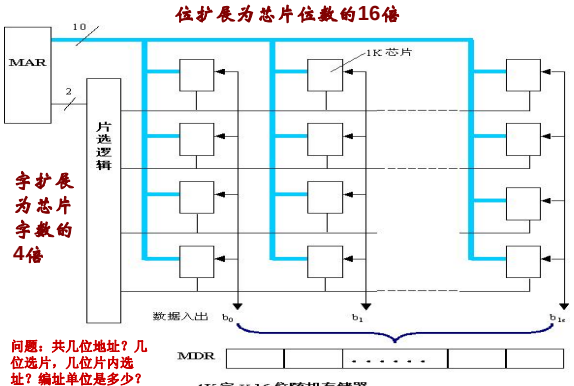
## 回顾：位片式芯片框图




1K 位芯片结构



## 位扩展为芯片位数的16倍

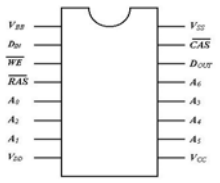



问题：共几位地址？几位选片，几位片内选址？编址单位是多少？

共12位地址，高两位用来选片，低10位用于片内选址

## 举例：TMS4116动态MOS存储器芯片

- 总体性能：
  - 存储容量：16K x 1位
  - 7根地址线复用（2x7=14）
- 芯片引脚
- 芯片框图
- 地址缓冲器和地址译码器
- 存储阵列
- 读出再生放大器
- 基本时序、读写操作定时
- 由4116芯片构成64K字X16位的存储器



TMS4116 引脚图

问题：CPU送出的地址有几位？送到4116芯片的地址有几位？ CPU送出的地址位数由物理空间大小决定；送4116有14位地址。

问题：7个地址引脚何时送行地址？何时送列地址？

$\overline{RAS}$有效时送行地址

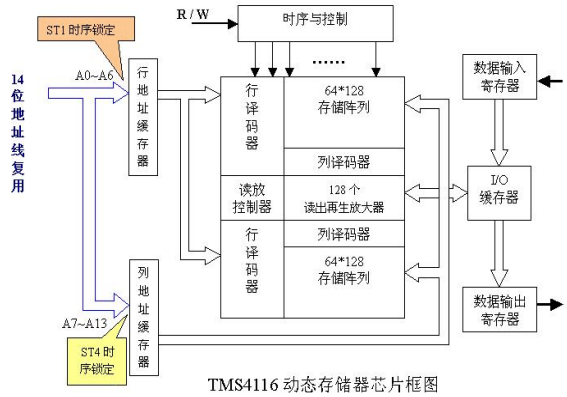
$\overline{CAS}$有效时送列地址

问题：$\overline{WE}$的含义是什么？

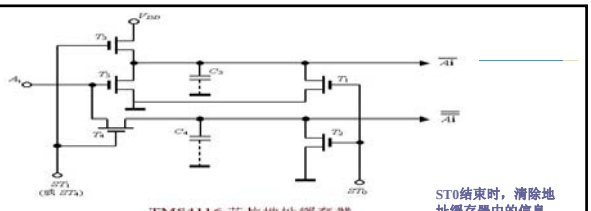
$\overline{WE}$低时写操作，高时读操作

| | | | |
|---|---|---|---|
| $A_0$~$A_6$ | 地址 | $D_{IN}$ | 数据输入端 |
| $V_{DD}$ | -12V | $D_{out}$ | 数据输出端 |
| $V_{CC}$ | +5V | $\overline{WE}$ | 写允许 |
| $V_{BB}$ | -5V | $\overline{RAS}$ | 行地址选通 |
| $V_{SS}$ | 地 | $\overline{CAS}$ | 列地址选通 |

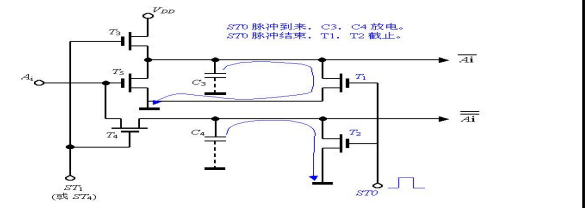
TMS4116 芯片引脚名称



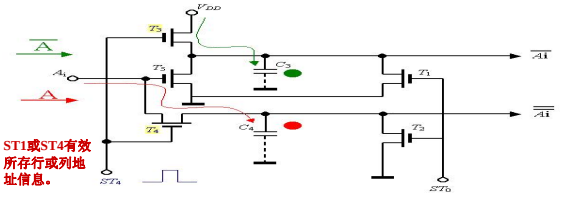
TMS4116 动态存储器芯片框图


TMS4116 芯片地址缓存器

ST0结束时，清除地址缓存器中的信息。



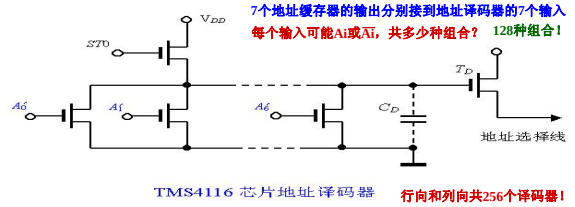
TMS4116 地址缓存器操作：清除上一次锁存的地址信息


ST1或ST4有效所存行或列地址信息。

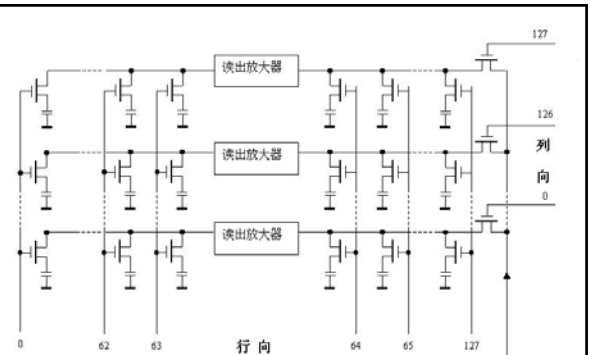
TMS4116 地址缓存器：锁存输入进来的列地址信息

7个地址缓存器的输出分别接到地址译码器的7个输入

每个输入可能Ai或$\overline{Ai}$，共多少种组合？ 128种组合！

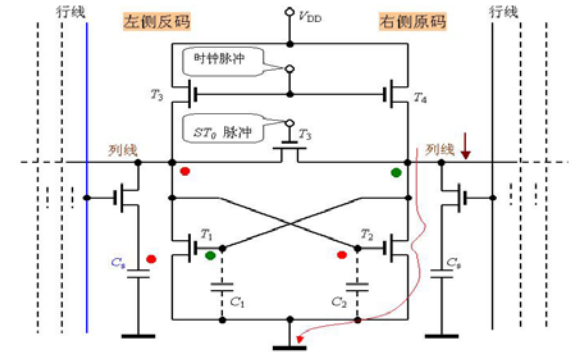


TMS4116 芯片地址译码器 行向和列向共256个译码器！

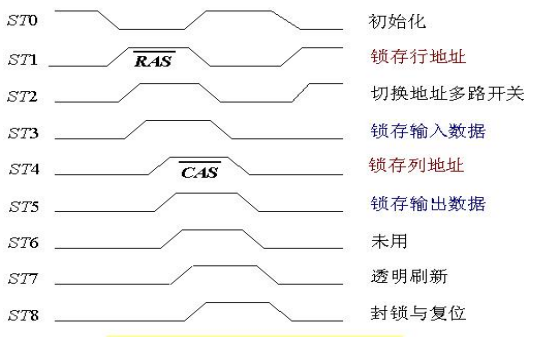


每个译码器输出连到一根行线或列线上。

1 位 16K 存储阵列

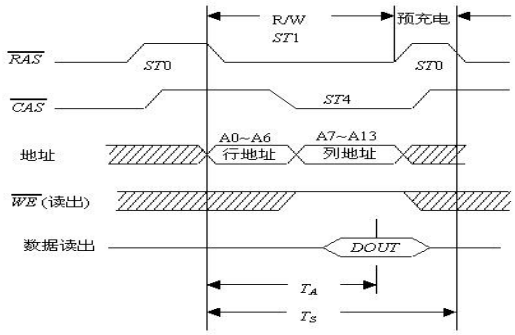


TMS4116 存储器芯片的读出再生操作



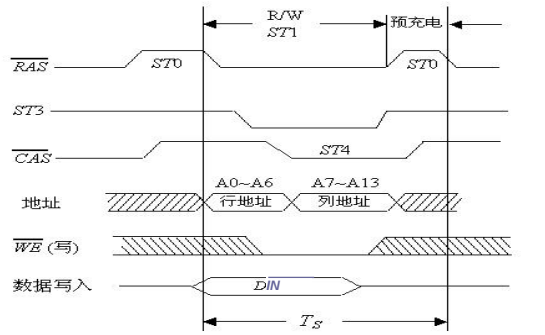

TMS4116 芯片基本时序简图

memory.25




TMS4116 芯片读出操作定时

memory.26




TMS4116 芯片写入操作定时

memory.27

## 举例：典型的16M位DRAM（4MX4）

- 16M位=4Mb x 4=2048 x 2048 x 4=$2^{11}x2^{11}x4$

(1) 地址线：11根分时复用,由RAS和CAS提供控制时序。

(2) 存储字是4位，需四个位平面，对相同行、列交叉点的四个位一起读/写

(3) 内部结构框图

问题：
为什么每出现新一代存储器芯片，容量至少提高四倍？

行地址和列地址分时复用, 每出现新一代存储器芯片，至少要增加一根地址线

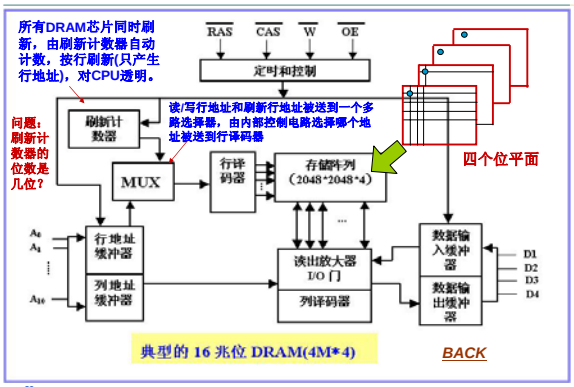
每加一根地址线，则行地址和列地址各增加一位，所以行数和列数各增加一倍

因而: 容量至少提高四倍

SKIP

memory.28

## 举例：典型的16M位DRAM（4MX4）




典型的 16 兆位 DRAM(4M*4)

BACK

memory.29

## 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，一般为2ms。即：2ms内必须对每个单元刷新一遍。
- 有四种刷新方式：集中式、分散式、异步和透明刷新。

① 集中刷新：

前一段时间正常读/写，后一段时间停止读/写，集中逐行刷新。

特点：集中刷新时间长，不能正常读/写（死区），很少使用该方法。

② 分散刷新：

一个存储周期分为两段: 前一段用于正常读/写操作，后一段用于刷新操作。

特点：不存在死区，但每个存储周期加长。很少使用。

③ 异步刷新：

结合上述两种方式。以128行为例，在2ms时间内必须轮流对每一行刷新一次，即每隔2ms/128=15.5μs刷新一行。这时假定读/写与刷新操作时间都为0.5μs，则可用前15μs进行CPU读/写，最后0.5μs完成刷新操作。

特点：结合前两种，效率高，用得较多。

④ 透明刷新（或称隐含式刷新）：

在存储器空闲时由存储控制器控制插入刷新操作，对CPU透明。

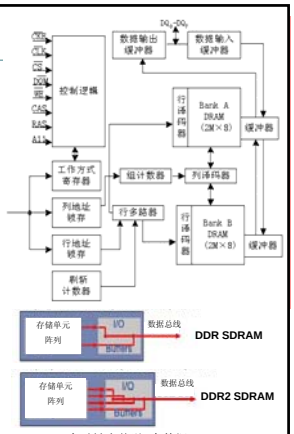
特点：不占用CPU时间，对CPU而言透明的。目前高档微机中大部分采用这种方式。

memory.30

## SDRAM芯片技术

- 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
    - CPU送地址到地址线，主存译码
    - CPU发读命令，然后等待存储器发回“完成”信号
    - 主存接受到读命令后，读数据送至数据线，然后发“完成”信号给CPU
    - CPU接受到“完成”信号，从数据线取数据

    写操作过程类似
  - 同步方式的特点
    - CPU和主存在同一个时钟信号控制下工作，不需要应答信号（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，总是在确定的几个时钟周期后去取数据
    - 存储器芯片必须支持同步方式
- SDRAM是同步存储器芯片
  - 每步操作都在系统时钟控制下进行
  - 成组（突发/猝发）传送方式
  - 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




DDR3：一个时钟内传送8个数据



## 只读存储器和Flash存储器

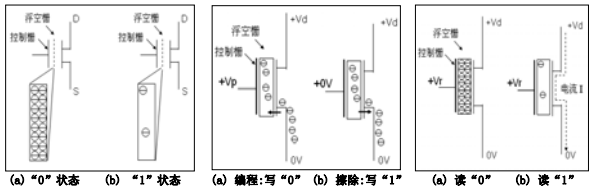
- 特点：
  - ★ 信息只能读不能写。
  - ★ 非破坏性读出，无需再生。
  - ★ 也以随机存取方式工作。
  - ★ 信息用特殊方式写入，一经写入，就可长久保存，不受断电影响。故是非易失性存储器。
- 用途：
  - ★ 用来存放一些固定程序。如监控程序、启动程序等。只要一接通电源，这些程序就能自动地运行；
  - ★ 可作为控制存储器，存放微程序。
  - ★ 还可作为函数发生器和代码转换器。
  - ★ 在输入/出设备中，被用作字符发生器，汉字库等。
  - ★ 在嵌入式设备中用来存放固化的程序。



## 回顾：只读存储器(Read Only Memory)

MROM：掩膜只读存储器（Mask ROM）
PROM：可编程只读存储器（Programmable ROM）
EPROM：可擦除可编程只读存储器（Erasable PROM）
EEPROM ($E^2$PROM)：电可擦除可编程只读存储器 （Electrically EPROM）
flash memory：闪存

闪存的读取速度与DRAM相近，是磁盘的100倍左右；写数据（快擦－编程）则与硬盘相近




控制栅加足够正电压时，浮空栅储存大量负电荷，为 “0”
控制栅不加正电压时，浮空栅少带或不带负电荷，为 “1”

有三种操作：编程、读取、擦除

最初都是1；编程：写0；擦除：写1
读出：控制栅加正电压，若为0，则读出电路检测不到电流；若为1，则检测到电流。



## 存储器芯片的扩展

- 字扩展
  - 位数不变、扩充容量

    例如，用16K×8位芯片扩展成64K×8位存储器，需几个芯片？地址范围各是多少？

    字方向扩展4倍，即4个芯片。0000-3FFFH，4000-7FFFH，8000-BFFFH，C000- FFFFH

    地址共16位，高两位由外部译码器译码生成4个输出，分别连到4个片选信号，片内地址有14位
  - 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
- 位扩展
  - 字数不变，位数扩展

    例如，用4096×1位芯片构成4K×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，没有外部译码器
- 字位同时扩展
  - 字和位同时扩展

    例如，用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？

    字向4倍、位向2倍，8个芯片。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



## 举例：128MB的DRAM存储器

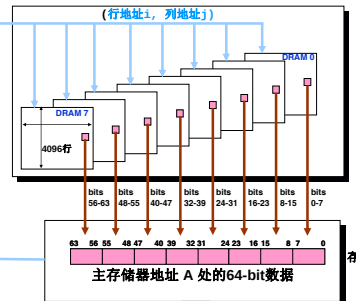
问题：芯片内地址是否连续？

不连续，是交叉编址方式！可同时读写所有芯片。

地址A

问题：如果高位地址用作片选，则情况如何？

连续编址，不可同时读写多个芯片。




- 总容量128 MB
- 由8片DRAM芯片构成
- 每片 16Mx8 bits
- 行地址、列地址各12位

为什么呢？

- 每1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

□ 行、列地址为(i,j)的8个单元

问题：地址范围？

存储器地址位数27位，片内地址24位，与高24位存储器地址相同。

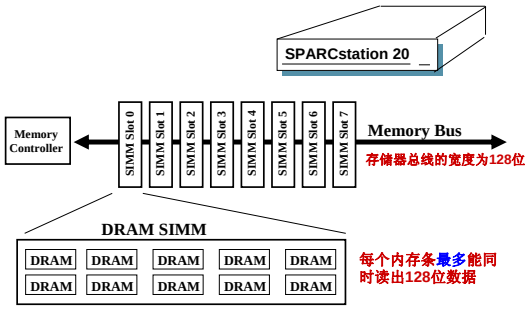
范围：000000-FFFFFFH

最多读64位

低3位地址的作用是什么？确定8个字节中的哪个，用来进行片选。



## 举例：SPARCstation 20’s Memory Module

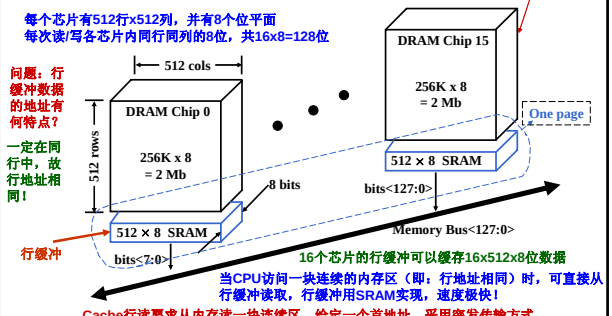



每次访存操作总是在某一个内存条内进行！

## 举例：SPARCstation 20's Memory Module

- one memory module （内存条）
  - Smallest: 4 MB = 16x 2Mb DRAM chips, 8 KB of Page Mode SRAM
  - Biggest: 64 MB = 32x 16Mb chips, 16 KB of Page Mode SRAM

每个芯片有512行x512列，并有8个位平面
每次读/写各芯片内同行同列的8位，共16x8=128位




问题：行缓冲数据的地址有何特点？
一定在同行中，故行地址相同！

16个芯片的行缓冲可以缓存16x512x8位数据
当CPU访问一块连续的内存区（即：行地址相同）时，可直接从行缓冲读取，行缓冲用SRAM实现，速度极快！

Cache行读要求从内存读一块连续区，给定一个首地址，采用突发传输方式



## PC机主存储器的物理结构

- 主存储器由若干内存条组成
- 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DDR和DDR2内存条：
  - 均采用双列直插式，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



## 存储器芯片和CPU的连接

- 通过总线连接
  - 地址线的连接
    - CPU地址线数决定了整个主存空间的寻址范围，故比存储芯片地址引脚线多。通常将CPU地址线的低位和存储芯片地址线相连，高位部分用作字扩展时的片选信号的译码。
  - 数据线的连接
    - CPU数据线数决定了一次可读写的最大数据宽度，故比存储芯片数据引脚线多，通常将CPU数据线连到多个进行位扩展的芯片中，使扩展后的位数与CPU数据线数相等
  - 控制线的连接
    - 如果CPU读/写命令线和存储芯片的读/写控制线都是一根，且电平信号一致，则可直接相连。若CPU读/写命令线分开，则需要分别进行连接。
- ROM区和RAM区的划分
  - 主存空间包括ROM和RAM区
  - ROM区用来存放系统程序、标准子程序等，选用ROM芯片构造；RAM区用来存放用户程序，选用RAM芯片构造
  - 选择存储芯片的类型和数量时，必须先确定好ROM区和RAM区的地址范围



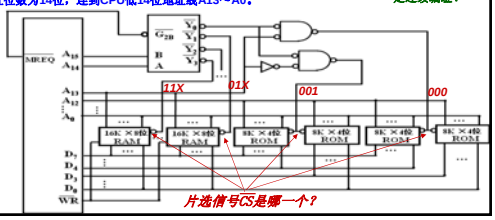
## 举例：CPU和主存的连接

CPU地址线A15～A0，数据线D7～D0，$\overline{WR}$为读/写信号，$\overline{MREQ}$为访存请求信号。0000H～3FFFH为系统程序区，4000H～FFFFH为用户程序区。用8K×4位ROM芯片和16K×8位RAM芯片构成该存储器，要求说明地址译码方案，并将ROM芯片、RAM芯片与CPU连接。

解：因为0000H～3FFFH为系统程序区，ROM区高两位总是00，低14位为全译码。
ROM区大小为：$2^{14}$×8位=16K×8位=16KB
ROM芯片数为：16K×8位 / 8K×4位 = 2×2 = 8，字方向扩展2倍，位方向扩展2倍
ROM芯片内地址位数为13位，连到CPU低13位地址线A12～A0
因为4000H～FFFFH为用户程序区，RAM区高两位是01、10、11，低14位为全译码。
RAM区大小为：3×$2^{14}$×8位=3×16K×8位= 48KB
RAM芯片数为：48K×8位 / 16K×8位 = 3×1 = 3，字方向上扩展3倍，位方向上不扩展。
RAM芯片内地址位数为14位，连到CPU低14位地址线A13～A0。

ROM芯片的片选信号由最高三位确定。
RAM芯片的片选信号由最高两位确定。
问题：交叉/连续编址方案？
高位地址译码作为片选信号，故是连续编址！

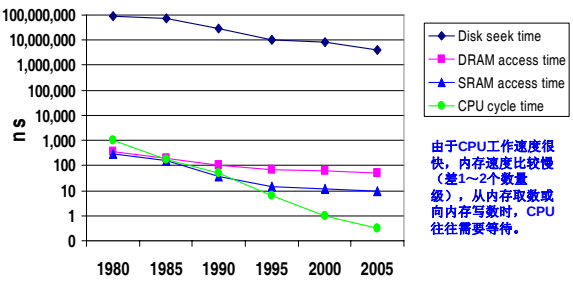
问题：为什么WR不连到ROM芯片上？
因为ROM芯片只能读，不能写，只要选中就是读，无需读写信号。
问题：$\overline{MREQ}$信号的作用是什么？
为有效(低电平)时，表示选中主存读写。




片选信号CS是哪一个？



## 回顾：存储器与CPU速度差距愈来愈大




由于CPU工作速度很快，内存速度比较慢（差1～2个数量级），从内存取数或向内存写数时，CPU往往需要等待。

从上图可以看出什么？
DRAM、硬盘与CPU 之间的速度差距愈来愈大！

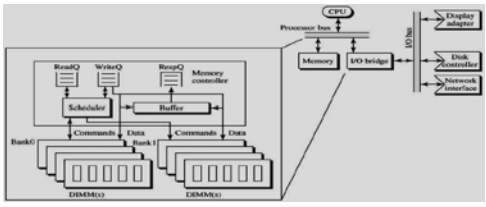
解决内存访问速度慢的措施有三个：
- 提高主存芯片本身的速度
- 采用多模块存储器技术
- 在主存和CPU之间加入Cache



## 提高DRAM存储器速度的措施

- 之一：采用芯片内部行缓冲，以提高芯片本身的速度
  反复多次使用芯片内部缓冲器中的内容，不要每次都重复地进行“行访问”
  DDR SDRAM (Double data-rate synchronous DRAM)和DDR2 SDRAM

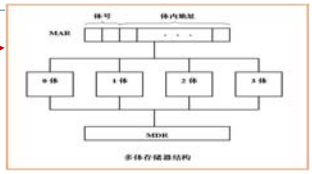

- 之二：多模块技术
  2个、4个或多个存储器同时工作

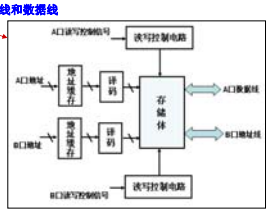


## 加快访存速度措施之二：多模块存储器

- 多体存储器和多模块存储器
  - 多体存储器
    - 由若干个小体组成
    - 共用地址寄存器MAR和数据寄存器MDR
    - 不能提高数据访问速度
  - 双口存储器
    - 通常作为双口RAM或指令预取部件
    - 两套独立的读/写控制电路、地址缓存、地址译码及地址线和数据线
    - 能同时进行两个数据的读/写
  - 多模块存储器
    - 也包含多个小体
    - 每个体有其自己的MAR、MDR和读写电路
    - 可独立组成一个存储模块
    - 能提高数据访问速度

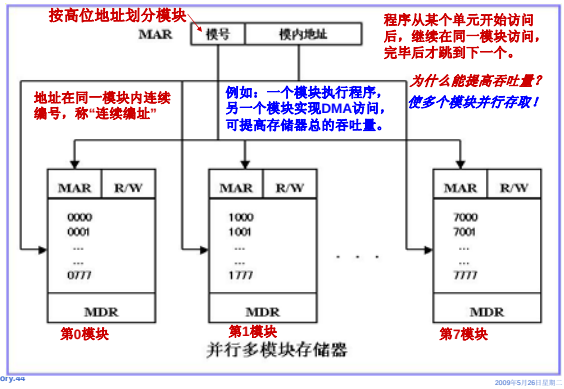
根据不同的编址方式可分为
连续编址和交叉编址





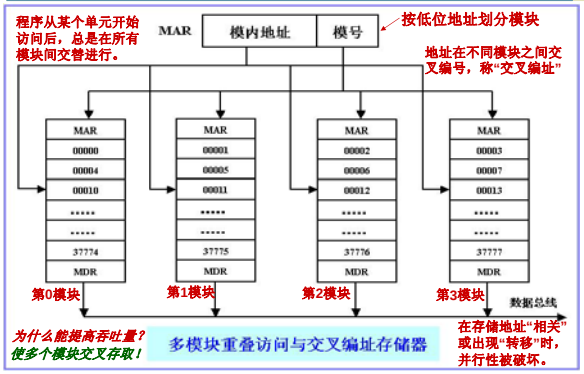


## 连续编址多模块存储器





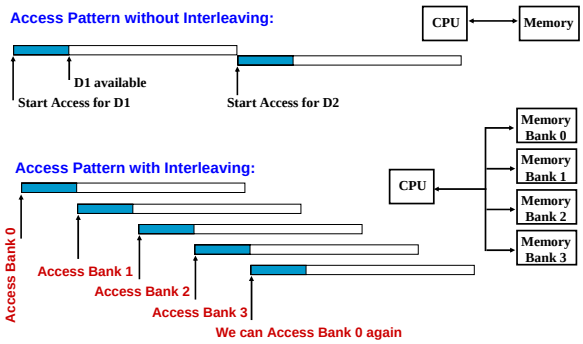

## 交叉编址多模块存储器




每隔1/4周期在数据总线上得到一个信息，使主存吞吐量提高4倍!



## Increasing Bandwidth – Interleaving(交叉)






## 第一讲小结

- 信息的存储、传送、处理单位
  - 记忆单元 / 编址单位 / 存储单位 / 传输单位 / 机器字长
- 存储器分类
  - 可按存取方式 / 易失性 / 可更改性 / 元器件 / 功能来分
- 半导体存储器随机访问存储器
  - SRAM ：速度快，容量小，可做快速小容量存储器
  - DRAM：速度慢，容量大，用作主存
- RAM芯片组织：
  - 存储阵列：按行、列排，分别由行地址和列地址指出位置
  - 地址译码器：分行、列地址译码器
  - 读写逻辑：可控制多个位平面的同一位数据一起读/写
- 提高存储器速度的措施：
  - 芯片内采用行缓存，行内数据直接从缓存中取
  - 采用多模块存储器，多个存储器交叉存取
  - 引入Cache（下一讲的主要内容）



## 第二讲 高速缓冲存储器(Cache)

主 要 内 容

- 什么是程序访问的局部化特性
- 具有Cache机制的CPU的基本访存过程
- Cache和主存之间的映射方式
  - 直接映射 / 全相联映射 / 组相联映射
- cache容量和块大小的选择
- Cache替换算法
- cache-friendly的程序
- Cache的写策略
  - Write Back 和Write Through
- Cache失靶处理
- Cache性能评估

## What we want in a memory

到目前为止，已经了解到有以下几种存储器：

Register，SRAM，DRAM， Hard Disk，Magnetic Tape and Optical Disk

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

* non-volatile

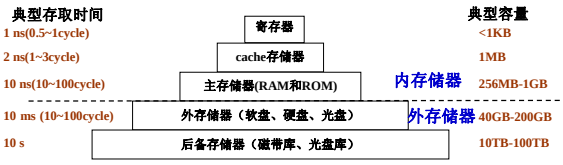
问题：你认为哪一种最适合做计算机的存储器呢？

单独用某一种存储器，都不能满足我们的需要！

考虑结合各种存储器的特点，采用分层存储器结构来构建计算机的存储体系！



## 计算机中存储器的层次结构




- 分析：速度越快，成本较高
- 为提高性能/价格，各存储器组成一个层状塔式结构，取长补短，协调工作
- 工作过程：
  1）CPU运行时，需要的操作数大部分来自寄存器
  2）如需要从(向)存储器中取(存) 数据时，先访问cache，如在，取自cache
  3）如操作数不在cache，则访问RAM，如在RAM中，则取自RAM
  4）如操作数不在RAM，则访问硬盘，操作数从硬盘中读出→RAM →cache



## 回顾：传统存储器分级体系结构

五层金字塔形分层系统从上到下的特点：

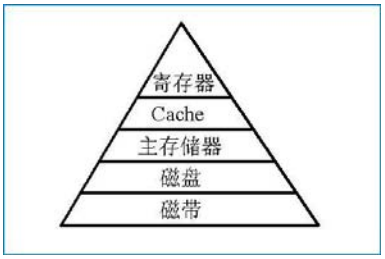
1，每位价格降低

2，容量增大

3，存取时间增大

4，访问频度降低

传统结构




Traditional Memory Hierarchy



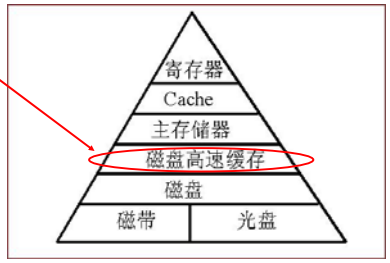
## 现代存储器分级体系结构

开辟一部分内存区，用作“Disk Cache”，用于存放将被送到磁盘上的数据。

引入“Disk Cache”的好处：

(1)写盘时按“簇”进行，以避免频繁地小块数据写盘。

(2)有些中间结果数据在写回盘之前可被快速地再次使用。

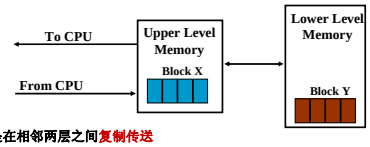



Contemporary Memory Hierarchy

（现代结构）



## 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
  Smaller, faster, and uses more expensive technology
  Lower Level: 下层更远离CPU
  Bigger, slower, and uses less expensive technology
- Block: 最小传送单位是一个定长块，互为副本

问题：为什么这种层次化结构是有效的？ *主要是基于“程序访问局部化”特点！*

- 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



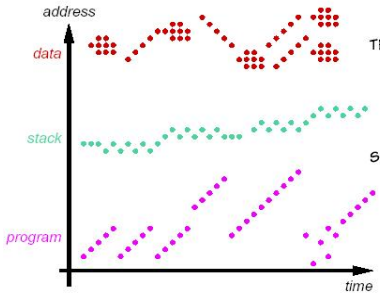
## 加快访存速度措施之三：引入Cache

- 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存储器的一个很小范围内
  - 这种现象称为程序访问的局部性
- 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
- 程序访问局部性分为空间局部性和时间局部性
- 基于程序访问的局部性使访存要求能快速得到响应
  - 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）的程序块和数据，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！

SKIP

## Typical Memory Reference Patterns




TEMPORAL LOCALITY –
If an item is referenced, it will tend to be referenced again soon

SPATIAL LOCALITY –
If an item is referenced, nearby items will tend to be referenced soon.

下面用一个例子来说明！



## 程序的局部性原理举例1

高级语言源程序

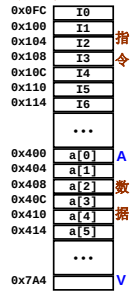
```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

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    累计在sum中
I6:         ap   <-- ap + 4     计算下1个数组元素的地址
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:




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）→0x120（I9）

若n足够大，在一段时间内就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，故程序的空间局部性好！

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，也按顺序访问，所以，空间局部性好；
每个数组元素都被访问1次，所以没有时间局部性。

BACK



## 程序的局部性原理举例2

以下程序A和B中，哪一个对数组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:

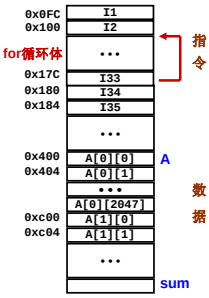
```
int sumarrayrows(int A[M][N])
{
  int i, j, sum=0;
    for (i=0; i<M, i++)
        for (j=0; j<N, j++)
            sum+=A[i][j];
    return sum;
}
```

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;
}
```

假定数组在存储器中按行优先顺序存放

M=N=2048时主存的布局:

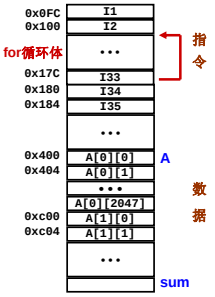





## 程序的局部性原理举例2

程序段A:

```
……
int i, j, sum=0;
for (i=0; i<2048, i++)
    for (j=0; j<2048, j++)
        sum+=A[i][j];
return sum;
……
```




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] ,……, 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][]都只被访问一次，所以时间局部性差！

（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2048x2048次，所以时间局部性好！

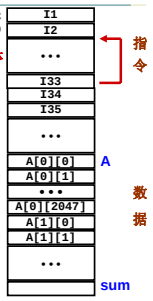
实际上 优化的编译器使循环中的sum分配在寄存器中，最后才写回存储器！



## 程序的局部性原理举例2

程序段B:

```
……
int i, j, sum=0;
for (j=0; j<2048, j++)
    for (i=0; i<2048, i++)
        sum+=A[i][j];
return sum;
}
```




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？

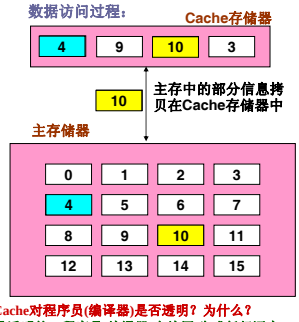
- Cache是一种小容量高速缓冲存储器，它由SRAM组成
- Cache直接制作在CPU芯片内，速度几乎与CPU一样快
- 程序运行时，CPU使用的一部分数据/指令会预先成批拷贝在Cache中，Cache的内容是主存储器中部分内容的映象
- 当CPU需要从内存读(写)数据或指令时，先检查Cache，若有，就直接从Cache中读取，而不用访问主存储器

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？
给出的主存地址怎样转换为Cache地址？……

主存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）

数据访问过程：



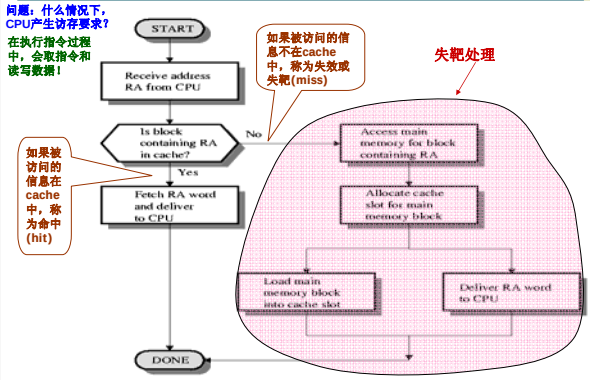

Cache对程序员(编译器)是否透明？为什么？
是透明的，程序员(编译器)在编写/生成低级语言程序时无需了解Cache是否存在或如何设置。

但是，对Cache深入了解有助于编写出高效的程序！

## Cache 的操作过程

问题：什么情况下，CPU产生访存要求？

在执行指令过程中，会取指令和读写数据！






## Cache映射(Cache Mapping)

- 什么是Cache的映射功能？
  - 把访问的局部主存区域取到Cache中时，该放到Cache的何处？
  - Cache槽比主存块少，多个主存块映射到一个Cache槽中
- 如何进行映射？
  - 把主存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）或项（Entry）或块（Block）
  - 将主存块和Cache槽按照以下三种方式进行映射
    - 直接(Direct)：每个主存块映射到Cache的固定槽中
    - 全相联(Full Associate)：每个主存块映射到Cache的任意槽中
    - 组相联(Set Associate)：每个主存块映射到Cache的固定组中的任意槽中



## The Simplest Cache: Direct Mapped Cache

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

    Cache槽号=主存块号 mod Cache槽数

    举例：4=100 mod 16 （假定Cache共有16槽）

    (说明：主存第100块应映射到Cache的第4槽中。)
- 特点：
  - 容易实现，命中时间短
  - 无需考虑淘汰（替换）问题
  - 但不够灵活，Cache存储空间得不到充分利用，命中率低

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

SKIP



## 直接映射Cache组织示意图

假定：

数据在主存和Cache之间按块传送的单位为512字。

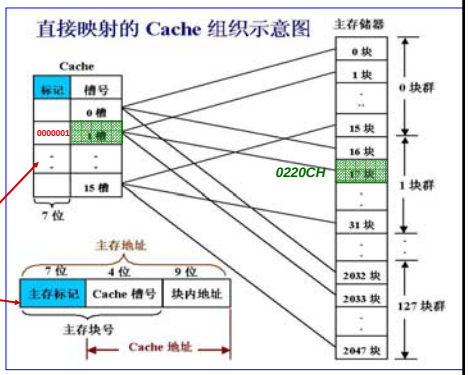
Cache大小：$2^{13}$字=8K字=16槽 x 512字/ 槽

主存大小：$2^{20}$字=1024K字=2048块 x 512字/ 块

Cache标记（tag）指出对应槽取自哪个主存块群

主存tag指出对应地址位于哪个块群

举例：假定Cache为空，如何对0220CH单元进行访问？



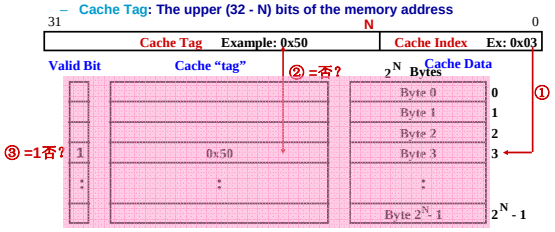



0000 0010 0010 0000 1100B第1块群中的0001块（即第17块）中第12个单元！

BACK

## Cache Organization: Cache Tag and Cache Index

- 假定主存地址为32位, 按字节编址
- 假定Cache是块大小为1B的直接映射Cache
  - Cache Index: The lower N bits of the memory address
  - Cache Tag: The upper (32 - N) bits of the memory address




- 利用了时间局部性：某字节不久又可能被用
- 没有利用空间局部性：某字节的邻近字节不久可能也被用，但没有被调到Cache（每次调入的一块信息只有一个字节！）
- 发生冲突概率增大，因为块小使映射到同一个Cache行的主存块增加

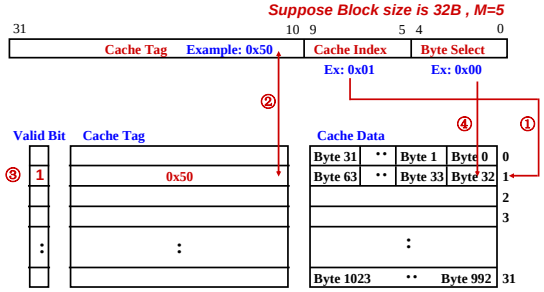
可以通过增大块的大小来利用空间局部性



## Ex2: 1 KB Direct Mapped Cache with 32 B Blocks

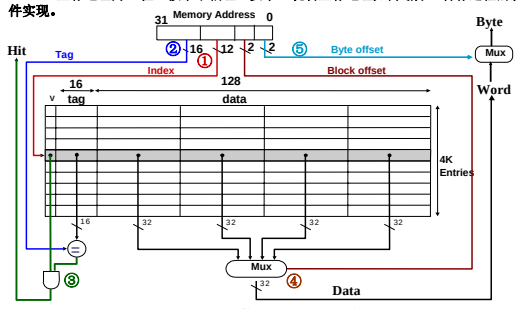
- For a $2^N$ byte cache (N=10 in this example):
  - The uppermost (32 - N) address bits are always the Cache Tag
  - The lowest M address bits are the Byte Select (Block Size = $2^M$)

*Suppose Block size is 32B , M=5*



## Ex3: 64 KB Direct Mapped Cache with 16B Blocks

- 假定主存和Cache之间采用直接映射方式，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，访存过程的硬件实现。




问题：Cache有多少行？容量多大？ 共 (64K / 16)= 4K 行

容量 4Kx(1+16)+64Kx8=580Kbits=72.5KB,数据占64KB / 72.5KB = 88.3%



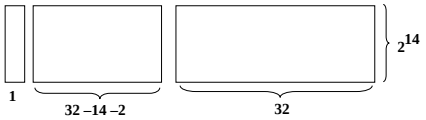
## 如何计算Cache的容量？

Consider a cache with 64 Blocks and a block size of 16 bytes.
What block number does byte address 1200 map to?

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

How many total bits are required for a directed mapped cache with 16K Entries of data and 1-word blocks, assuming a 32-bit address?
(Cache :直接映射方式 、16K项数据、块大小为1个32位字、32位地址)

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：$2^{14} \times (32 + (32-14-2)+1) = 2^{14} \times 49 = 784$ Kbits

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

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



## 全相联映射Cache组织示意图

各主存块可装到Cache任一槽Slot（行Line或项Entry）中。称为全映射或全相联映射

假定：

数据在主存和Cache之间块传送的单位为512字。

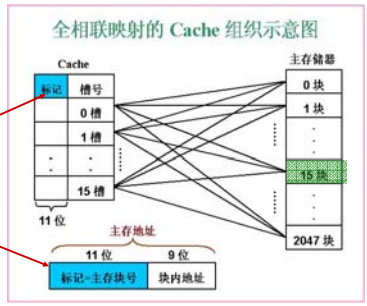
Cache大小：$2^{13}$字=8K字=16槽 x 512字/槽

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

Cache标记（tag）指出对应槽取自哪个主存块

主存tag指出对应地址位于哪个主存块

两个标记相等时，说明要找的地址在对应槽中。比较所有标记都不等，则失靶。



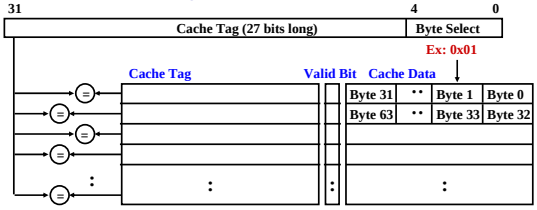

举例：假定Cache为空，如何对0220CH单元进行访问？

0000 0010 0010 0000 1100B 第17块中的第12个单元！可映射到任意cache槽中



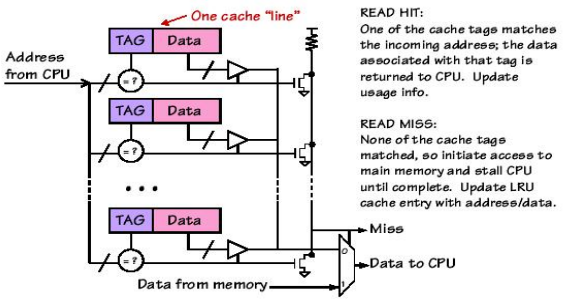
## 举例：Fully Associative

- Fully Associative Cache
  - 无需Cache索引，为什么？
    因为同时比较所有Cache项的标志
- By definition: Conflict Miss = 0
  - (全相联映射没有冲突失靶，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






## Fully Associative Cache




READ HIT:
One of the cache tags matches the incoming address; the data associated with that tag is returned to CPU. Update usage info.

READ MISS:
None of the cache tags matched, so initiate access to main memory and stall CPU until complete. Update LRU cache entry with address/data.

- 比直接映射灵活，命中率高。
- 采用相联存取技术（按内容访问），实现复杂、速度慢。
- Cache标志位数增加，比较逻辑成本随之增加。

} 不适合大容量Cache



## 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所有槽分组，把主存块映射到Cache固定组的任一槽中。
  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槽/组x512字/槽
  4=100 mod 8
  (主存第100块应映射到Cache的第4组的任意槽中。)

- 特点：
  - 结合了直接映射和全相联映射的优点。当Cache的组数为1时，则变为相联映射；当每组只有一个槽时，则变为直接映射。
  - 每组两个槽（称为2路组相联）较常用。一般每组4个槽以上的情况很少用。在较大容量的L2 Cahce和L3 Cahce中使用4-路以上。

## 组相联映射的Cache组织图

假定：

数据在主存和Cache之间按块传送的单位为512字。

Cache大小：$2^{13}$字=8K字=16槽 x 512字/ 槽=8组x2槽/ 组x512字/ 槽

主存大小：$2^{20}$字=1024K字=2048块 x 512字/ 块

Cache标记（tag）指出对应槽取自哪个主存组群

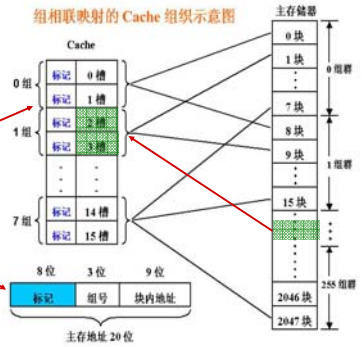
主存tag指出对应地址位于哪个主存组群中

两个标记相等时，说明要找的地址在对应槽中

举例：假定Cache为空，如何对0220CH单元进行访问？

0000 0010 0010 0000 1100B第2组群中的001块（即第17块）中第12个单元。

所以，映射到第一组中。

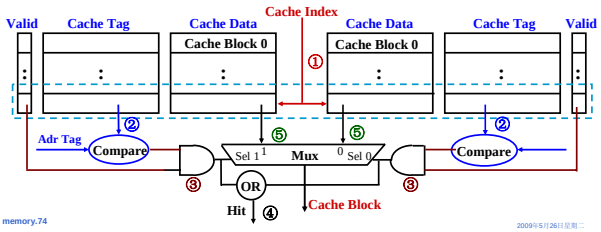



将主存地址标记和固定Cache组中每个Cache标记进行比较



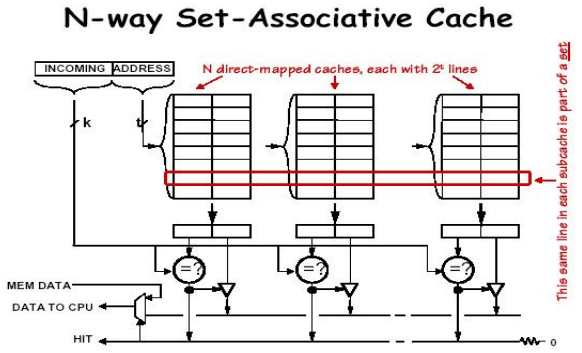
## 例1：A Two-way Set Associative Cache

- N-way set associative
  - N entries for each Cache Index (每个Cache组有N槽 or 行 )
  - 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（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






## 例2：A N-way Set Associative Cache

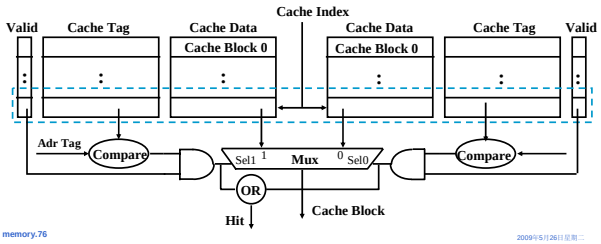





## Disadvantage of Set Associative Cache

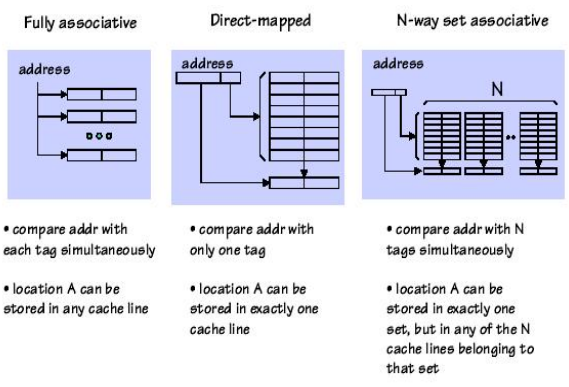
- N-way Set Associative Cache相对于 Direct Mapped Cache:
  - 比较器的个数为：N:1
  - 需要额外的MUX延时
  - 数据在判断是否命中后才被得到，而直接映射可“投机”预取

    在直接映射中，可认为总是命中, 所以可先把数据取来，若失靶则再恢复






## 三种映射方式总结




## 命中率、失靶率、失靶损失

- 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 (失靶率/失效率) = 1 - (Hit Rate)
  - Miss Penalty (失靶损失): 从主存替换到Cache ，并送CPU的时间

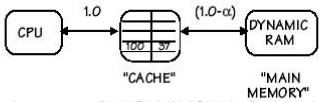
    Time to replace a block in Cache +

    Time to deliver the block to the processor

    (即：Cache替换 + 送CPU 时间)
- Hit Time << Miss Penalty （Why?）
- Average Access Time = Hit Rate × Hit Time + Miss Rate × Miss Penalty

## 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-α) MISS RATIO: Remaining references.

*Hit Time* *Miss Penalty*

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

2) Transparency (compatibility, programming ease)

Cache对程序员是透明的：程序不用管信息在主存还是在Cache！

完全由硬件完成在主存和Cache之间的信息交换

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



## 看看命中率对平均访问时间的影响

- 设平均访问时间T：

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

$= T_C + (1 - H)T_M$

- 例1. 若命中率H=0.85, Tc=1 ns, $T_M$=20ns，则平均访问时间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

### 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?

访问速度与命中率的关系非常大！



## 高速缓存的失靶率和关联度

- 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，可能存放的位置个数
    - 直接映射：关联度最低，为1
    - 全相联映射：关联度最高，为Cache行数
    - N-路组相联映射：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）
    - 失靶率：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



## 关联度示例

One-way set associative (direct mapped)

| Block | Tag | Data |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |

关联度为多少？ 1

Two-way set associative

| Set | Tag | Data | Tag | Data |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

关联度为多少？ 2

Four-way set associative

| Set | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | |
| 1 | | | | | | | | |

关联度为多少？ 4

Eight-way set associative (fully associative)

| Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |

关联度为多少？ 8

BACK



## 例子：Cache缺失和关联度

- 设有三个大小相等的Cache，都有四行，每行一个字。
  - Cache1: 全相联
  - Cache2: 2-路组相联
  - Cache3: 直接映射

按以下主存块地址顺序访问，其缺失次数各为多少？

0, 8, 0, 6, 8

| Address of memory block accessed | Hit or miss | Contents of cache blocks after reference | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| 0 | miss | Memory[0] | | | |
| 8 | miss | Memory[8] | | | |
| 0 | miss | Memory[0] | | | |
| 6 | miss | Memory[0] | | Memory[6] | |
| 8 | miss | Memory[8] | | Memory[6] | |

*Cache块号=主存块号 mod 4*

| Address of memory block accessed | Hit or miss | Contents of cache blocks after reference | | | |
|---|---|---|---|---|---|
| | | Set 0 | Set 0 | Set 1 | Set 1 |
| 0 | miss | Memory[0] | | | |
| 8 | miss | Memory[0] | Memory[8] | | |
| 0 | hit | Memory[0] | Memory[8] | | |
| 6 | miss | Memory[0] | Memory[6] | | |
| 8 | miss | Memory[8] | Memory[6] | | |

*Cache组号=主存块号 mod 2*

| Address of memory block accessed | Hit or miss | Contents of cache blocks after reference | | | |
|---|---|---|---|---|---|
| | | Block 0 | Block 1 | Block 2 | Block 3 |
| 0 | miss | Memory[0] | | | |
| 8 | miss | Memory[0] | Memory[8] | | |
| 0 | hit | Memory[0] | Memory[8] | | |
| 6 | miss | Memory[0] | Memory[8] | Memory[6] | |
| 8 | hit | Memory[0] | Memory[8] | Memory[6] | |

右边三种情况各对应哪种Cache？

直接映射

2路组相联

全相联

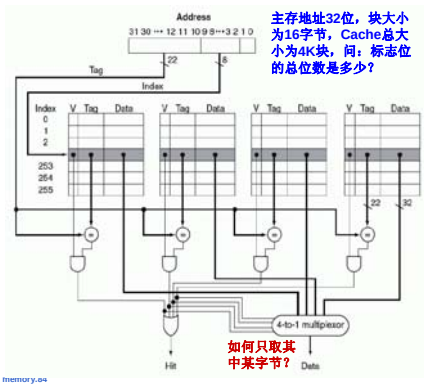
全相联的缺失率最低

直接映射最高！

相联度高，则缺失率低



## 关联度与标记位大小




主存地址32位，块大小为16字节，Cache总大小为4K块，问：标志位的总位数是多少？

块内地址占几位？ 4位

直接映射方式下：

相当于每组1块，共4K组

标志占32-4-12=16位

总位数占4Kx16=64K位

关联度增加一倍(2-way)

每组2块，共2K组

标志占32-4-11=17位

总位数占4Kx17=68K位

关联度增加2倍(4-way)

每组4块，共1K组

标志占32-4-10=18位

总位数占4Kx18=72K位

全相联时：

整个为1组，每组4K块

标志占32-4=28位

总位数占4Kx28=112K位

如何只取其中某字节？

## The Need to Replace! (何时需要替换？)

- Direct Mapped Cache:
  - 映射唯一，无需考虑替换，毫无选择地用新信息替换老信息
- N-way Set Associative Cache:
  - 每个主存数据有N个Cache槽可选择，需考虑替换
- Fully Associative Cache:
  - 每个主存数据可存放到Cache任意槽中，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槽
- 对应的Cache槽已被占满而需要调入新的主存块时，必须考虑从cache槽中调出一个主存块



## 替换(Replacement)算法

- 问题举例：

  组相联映射时，假定第0组的两个槽分别被主存第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第一组，因此，第一组中必须调出一块，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

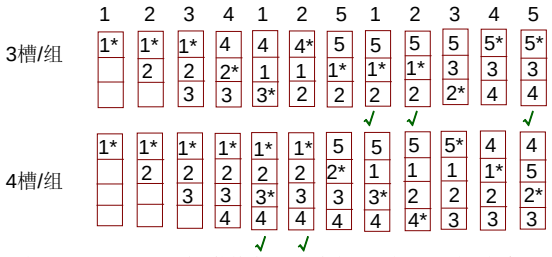
这里的替换策略和后面的虚拟存储器所用的替换策略类似，将是以后操作系统课程的重要内容，本课程只做简单介绍。有兴趣的同学可以自学。



## （自学）替换算法-先进先出（FIFO）

- 总是把最先进入的那一块淘汰掉。

  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槽/组、 4槽/组的情况。



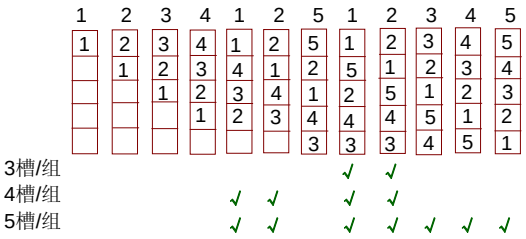

由此可见，FIFO不是一种堆栈算法，即命中率并不随组的增大而提高。



## （自学）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。

  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槽/组、 4槽/组、 5槽/组的情况。






## （自学）替换算法-最近最少用

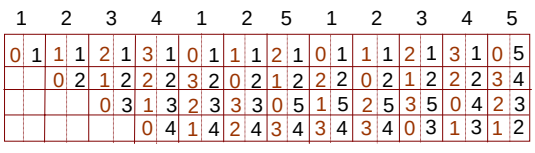
- 是一种堆栈算法，它的命中率随组的增大而提高。
- 当分块局部化范围(即：某段时间集中访问的存储区)超过了Cache存储容量时，命中率变得很低。极端情况下，假设地址流是1,2,3,4,1 2,3,4,1,……，而Cache每组只有3槽，那么，不管是FIFO，还是LRU算法，其命中率都为0。这种现象称为颠簸(Thrashing / PingPong)。
- 该算法具体实现时，并不是通过移动块来实现的，而是通过给每槽设定一个计数器，根据计数值来记录这些主存块的使用情况。这个计数值称为LRU位。

  具体实现



## （自学）替换算法-最近最少用

- 计数器变化规则：
  - ★ 每组4槽时，计数器有2位。计数值越小则说明越被常用。
  - ★ 命中时，被访问的槽的计数器置0，比其低的计数器加1，其余不变。
  - ★ 未命中且该组未满时，新槽计数器置为0，其余全加1。
  - ★ 未命中且该组已满时，计数值为3的那一槽中的主存块被淘汰，新槽计数器置为0，其余加1。



## （自学）替换算法-其他算法

- 最不经常用（LFU）算法：

  替换掉Cache中引用次数最少的块。LFU也用与每个槽相关的计数器来实现。

  （这种算法与LRU有点类似，但不完全相同。）

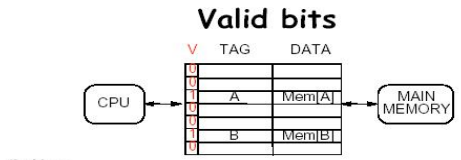
- 随机算法：

  随机地从候选的槽中选取一个淘汰，与使用情况无关。

  （模拟试验表明，随机替换算法在性能上只稍逊于基于使用情况的算法。而且代价低！）



## 有效位（Valid Bit）




Problem:

Ignoring cache lines that don't contain anything of value... e.g., on
- start-up
- "Back door" changes to memory (eg loading program from disk)

Solution:

Extend each TAG with VALID bit.

- Valid bit must be set for cache line to l装入新块时 使V=1
- At power-up / reset : clear all valid bits 开机或复位时使V=0
- Set valid bit when cache line is first replaced. 第一次被替换时使V=1
- Cache Control Feature: Flush cache by clearing all valid bits, Under program/external control. 通过使V=0冲刷Cache



## 举例

- 假定计算机系统有一个容量为32Kx16位的主存，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 x 64字 / 块

  Cache：4K字=16组 x 4槽 / 组 x 64 字 / 槽

  主存地址划分为：

| 标志位 | 组号 | 字号 |
|---|---|---|
| 5 | 4 | 6 |

4352/64=68，所以处理器的访问过程是对前68块连续访问10次。



## 举例

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

LRU算法：第一次循环,对于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/(ptc+(1-p)tm)=10/(p+10x(1-p))=9.5倍



## 举例

| | 0 槽 | 1 槽 | 2 槽 | 3 槽 |
|---|---|---|---|---|
| 0组 | 0/16/32/48 | 16/32/48/64 | 32/48/64/0 | 48/64/0/16 |
| 1组 | 1/17/33/49 | 17/33/49/65 | 33/49/65/1 | 49/65/1/17 |
| 2组 | 2/18/34/50 | 18/34/50/66 | 34/50/66/2 | 50/66/2/18 |
| 3组 | 3/19/35/52 | 19/35/51/67 | 35/51/67/3 | 51/67/3/19 |
| 4组 | 4 | 20 | 36 | 52 |
| … | … | … | … | … |
| … | … | … | … | … |
| 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 (43520-68-7x4-2 x8)/43520=99.74%
速度提高：tm/ta=tm/(ptc+(1-p)tm)=10/(p+10x(1-p))=9.77倍



## 写策略（Cache一致性问题）

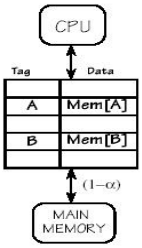
- 为何要保持在Cache和主存中数据的一致？
  - 因为Cache中的内容是主存块副本，当对Cache中的内容进行更新时，就存在Cache和主存如何保持一致的问题。
  - 以下情况也会出现"Cache一致性问题"
    - 当多个设备都允许访问主存时

      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，则对应Cache槽中的内容无效。
    - 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Cache槽的内容都变为无效。
- 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

## 基本的Cache处理算法

### Basic Cache Algorithm




ON REFERENCE TO Mem[X]: Look for X among cache 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] (写分配方式)

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]



## Write Policy: Write Through versus Write Back

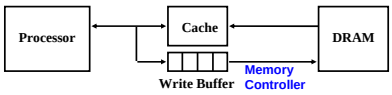
- 处理Cache读比Cache写更容易，指令Cache比数据Cache容易设计
- 对于写命中，有Two options
  - Write Through (通过式写、写直达、直写)　　你会如何翻译？
    - 同时写Cache和主存单元
    - What!!! How can this be? Memory is too slow(>100Cycles)?
      10%的存储指令使CPI增加到：1.0+100x10%=11
    - 使用写缓冲（Write Buffer）
  - Write Back (一次性写、写回、回写)
    - 在失靶时一次写回Cache块，每块有个修改位（“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Two options
  - 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，不装入主存块到Cache

直写Cache可用非写分配或写分配
写回Cache通常用写分配　　为什么？

SKIP



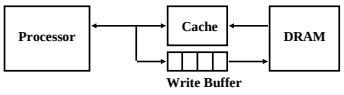
## Write Through中的Write Buffer



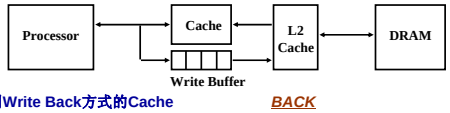

- 在 Cache 和 Memory之间加一个Write Buffer
  - Processor: 同时写数据到Cache和Write Buffer
  - Memory controller: 将缓冲内容写主存
- Write buffer (写缓冲) 是一个FIFO队列
  - 一般有4项
  - 在存数频率<<DRAM写（周期）频率情况下，效果好
- 最棘手的问题
  - Store frequency > 1 / DRAM write cycle(频繁写)时，使 Write buffer 饱和(溢出)，会发生阻塞



## Write Buffer Saturation（写缓冲饱和）




- 发生写缓冲饱和的可能性
  - The CPU Cycle Time < DRAM Write Cycle Time (客观上如此)
  - Store frequency >> 1/ DRAM write cycle(又发生频繁写)
    即：如果CPU 时钟宽度远远小于DRAM写周期，并且一段时间内发生大量的写操作，则不管写缓冲多大，都会发生写缓冲溢出（饱和）
- 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一致性问题）

*问题1：以下算法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]]

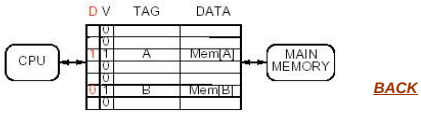
READ: Read Mem[X]
Set TAG[k] = X, DATA[k] = Mem[X]

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（“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性能评估与改善

- CPU时间：CPU执行时间+等待内存访问时间。即：
  - CPU时间=（CPU执行时钟数+Cache失靶引起阻塞的时钟数）X 时钟周期
  - Cache失靶引起阻塞的时钟数=读操作阻塞时钟数+写操作阻塞时钟数
  - 读操作阻塞时钟数=(读的次数 / 程序) x 读失靶率 x 读失靶损失
  - 写操作的情况较复杂：
    - 回写（write back）：
      替换时，需要一次性回写一个块，故会产生一些附加回写阻塞
      写操作阻塞时钟数=(写次数/ 程序) x 写失靶率 x 写失靶损失+回写阻塞
    - 直写（write through）：
      包括写失靶和write buffer阻塞两部分
      写操作阻塞时钟数=(写次数/ 程序) x 写失靶率 x 写失靶损失+写缓冲阻塞
  - 假定回写阻塞或写缓冲阻塞可以忽略不计，则可将读和写综合考虑：
    - 内存阻塞时钟数=(访存次数 / 程序) x 失靶率 x 失靶损失
    - 内存阻塞时钟数=(指令条数 / 程序) x (失靶数/ 指令) x 失靶损失



## 举例: 失靶带来的损失到底多大？

设Code Cache的失靶率为2%，Data Cache的失靶率为4%。假定一个处理器在没有任何存储器阻塞时的CPI为2，miss penalty为100个时钟。如果用SPECint2000来衡量，则使用完全没有失靶的完美Cache，处理器的速度会快多少？

分析过程如下：

指令的失靶时钟数为：Ix2%x100=2.0xI

SPECint2000的访存指令(Load和Store)频度为：36%，所以

数据的失靶时钟数为：Ix36%x4%x100=1.44xI

指令和数据总的失靶时钟数为：2xI+1.44xI=3.44I，也即：

平均每条指令要有3.44个时钟处在存储器阻塞状态

因此，因为存储器阻塞而使得CPI数增大到2+3.44=5.44. 故：

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{\text{IxCPIstallxClock cycle}}{\text{IxCPIperfectxClock cycle}} = \frac{5.44}{2}$$

如果Cache不发生失靶，则处理器速度会快2.72倍。



## 举例：处理器速度提高而存储器不变时的情况

- 例1：假定上例中CPI减为1，时钟宽度不变，则：
  因为存储器阻塞而使得CPI数增大到1+3.44=4.44. 故：

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{\text{IxCPIstallxClock cycle}}{\text{IxCPIperfectxClock cycle}} = \frac{4.44}{1}$$

  由此可知：存储器阻塞所花时间占整个执行时间的比例从：
  3.44 / 5.44=63% 上升到 3.44 / 4.44=77%
  结论：CPI越小，Cache阻塞的影响越大
- 例2：假定上例中时钟频率加倍， CPI不变，则：
  主存速度不太可能改变，故绝对时间不变，所以miss损失为200个时钟。
  每条指令发生的总失靶时钟数为：(2%x200)+36%x(4%x200)=6.88
  故：存储器阻塞使得CPI数增大到2+6.88=8.88

$$\frac{\text{时钟快的机器的性能}}{\text{时钟慢的机器的性能}} = \frac{\text{IxCPIstallofslow xClock cycle}}{\text{IxCPIstalloffast xClock cycle/2}} = \frac{5.44}{8.88/2} = 1.23$$

由此可知：时钟快的机器的性能只是较慢时钟机器的1.2倍。
如果没有Cache失靶的话，应该是2倍！
结论：CPU时钟频率越高，Cache缺失损失就越大

更正：
P.121例11(2)有错，
5.88/(10.72/2)=1.1倍

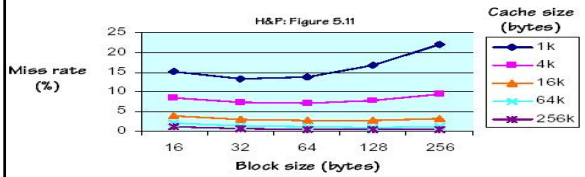
*上述两个例子说明：处理器性能越高，高速缓存的性能就越重要！*



## Cache大小、Block大小和失靶率的关系

Cache性能由失靶率确定，而失靶率与Cache大小、Block大小、Cache级数等有关

### Block size vs. miss rate




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

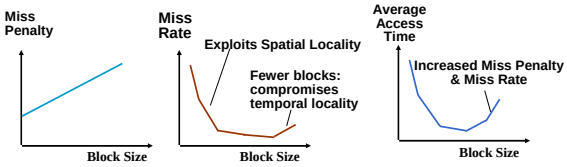
Cache大小：Cache越大，Miss率越低，但成本越高！

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



## Block Size Tradeoff (块大小的选择)

- 块大能很好利用 spatial locality， BUT:
  - 块大，则需花更多时间读块，失效损失变大
  - 块大，则Cache项变少，失效率上升
- Average Access Time:
  - = Hit Time x (1 - Miss Rate) + Miss Penalty x Miss Rate




所以，块大小必须适中！

## 系统中的Cache数目

- 刚引入Cache技术时只有一个Cache。近年来多Cache系统成为主流
- 多Cache系统中，主要有两个考虑因素：

[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CPU，其速度比L2快，其容量比L2大

[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？

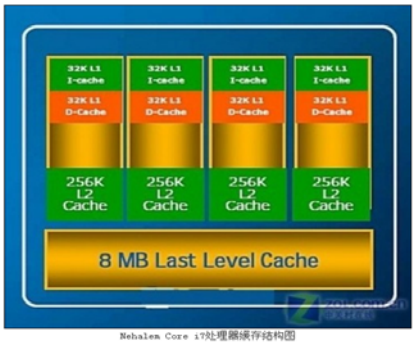
联合：指数据和指令都放在一个Cache中

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？



## 多核处理器中的多级Cache




Per core:
-32KB, 4-way L1 $I
-32KB, 8-way L1 $D
-256KB, 8-way L2

Shared
- 8 MB, 16-way L3

Nehalem Core i7处理器缓存结构图



## 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的访问时间
  - 否则，要访问主存，并取到L1 Cache和L2 Cache（缺失损失更大）
- 例子：有一处理器的CPI为1，所有访问能在L1 Cache中命中，时钟频率为5GHz。假定访问一次主存的时间为100ns，包括所有的缺失处理。设平均每条指令在L1 Cache中的缺失率为2%，若增加一个L2 Cache，访问时间为5ns，而且容量大到使L2 Cache缺失率减为0.5%，问处理器速率提高了多少？

解：如果只有一级Cache，则缺失只有一种：

即：L1缺失(访问主存)，其缺失损失为：100nsx5GHz=500个时钟

CPI=1+500x2%=11.0

如果有二级Cache，则有两种缺失：

即：L1缺失(访问L2Cache)：5nsx5GHz=25个时钟

L2缺失(访问主存)：500个时钟

CPI=1+25x2%+500x0.5%=4.0

因此，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

- 指令执行过程中，发生Cache失靶时，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（miss penalty最小）呢？

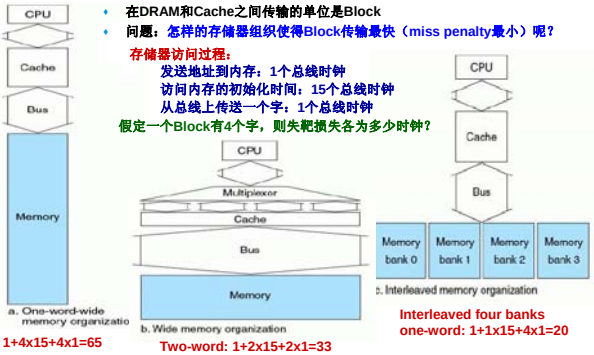
存储器访问过程：

发送地址到内存：1个总线时钟

访问内存的初始化时间：15个总线时钟

从总线上传送一个字：1个总线时钟

假定一个Block有4个字，则失靶损失各为多少时钟？




1+4x15+4x1=65

Two-word: 1+2x15+2x1=33

Four-word: 1+15+1=17

Interleaved four banks
one-word: 1+1x15+4x1=20

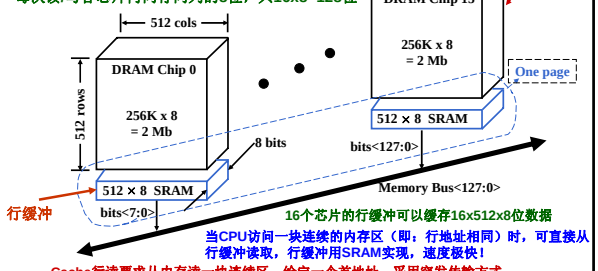


## 复习：SPARCstation 20's Memory Module

- one memory module （内存条） “页模式”：一行为一页（块）
  - Smallest: 4 MB = 16x 2Mb DRAM chips, 8 KB of Page Mode SRAM
  - Biggest: 64 MB = 32x 16Mb chips, 16 KB of Page Mode SRAM

每个芯片有512行x512列，并有8个位平面

每次读/写各芯片内同行同列的8位，共16x8=128位



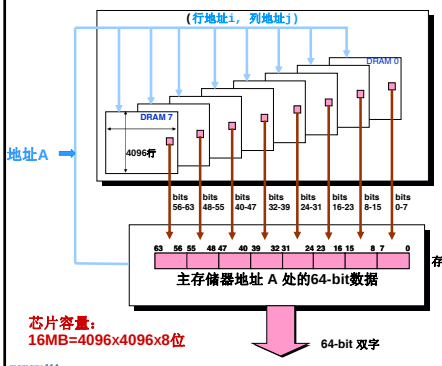

16个芯片的行缓冲可以缓存16x512x8位数据

当CPU访问一块连续的内存区（即：行地址相同）时，可直接从行缓冲读取，行缓冲用SRAM实现，速度极快！

Cache行读要求从内存读一块连续区，给定一个首地址，采用突发传输方式



## 复习：128MB的DRAM存储器




芯片容量：
16MB=4096x4096x8位

- 总容量=128 MB
- 由8片DRAM芯片构成
- 每片 16Mx8 bits
- 行地址、列地址各12位
- 每1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

在主存和CPU之间传送的是一个“Cache行”

一个Cache行通常位于同一个行缓冲中，第一次访存时间长，以后很快

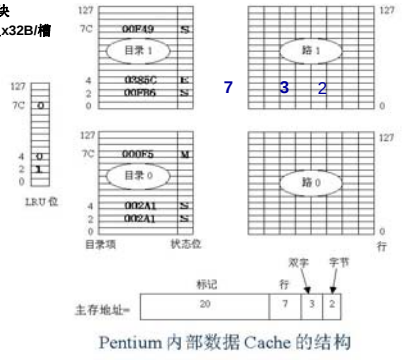
## 实例：奔腾机的Cache组织

主存：4GB=$2^{20}$x $2^{7}$块x $2^{5}$B/块
Cache：8KB=128组x2槽/组x32B/槽

**替换算法：**
采用LRU，每组有一位LRU位，用于记录该组2个槽的使用情况。可以这样规定：该位为0，则下次将淘汰第0槽。该位为1，则下次淘汰第1槽。

**写策略：**
默认方式为一次性写（Write Back），也可动态设置为通过式写（Write Through）。

**Cache一致性：**
支持MESI协议(在"计算机系统结构"课程详细介绍)。




Pentium 内部数据 Cache 的结构

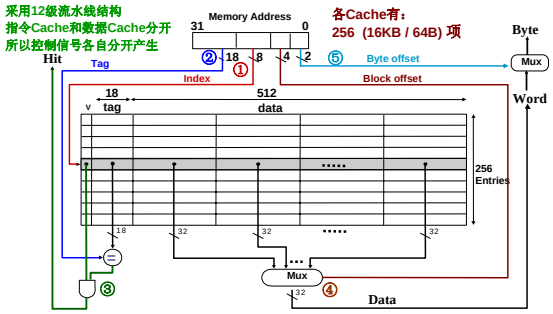


## 实例：内置FastMATH处理器

- FastMATH处理器是MIPS结构的嵌入式微处理器

采用12级流水线结构
指令Cache和数据Cache分开
所以控制信号各自分开产生

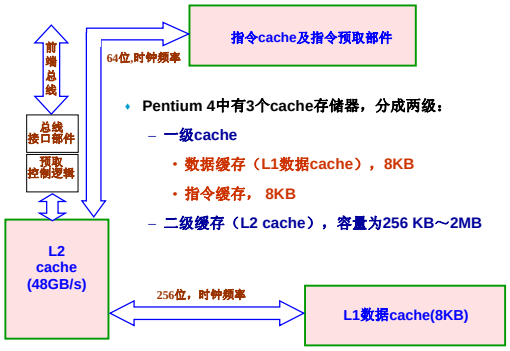
各Cache有：
256（16KB / 64B）项




写操作比读操作复杂! 处理器提供了写通过和写回两种方式，由OS决定采用何种策略
SPEC2000int的指令、数据和综合缺失率分别为：0.4%, 11.4%, 3.2%



## 实例：Pentium 4的cache存储器




- Pentium 4中有3个cache存储器，分成两级：
  - 一级cache
    - 数据缓存（L1数据cache），8KB
    - 指令缓存， 8KB
  - 二级缓存（L2 cache），容量为256 KB～2MB



## 缓存在现代计算机中无处不在

- 问题：缓存技术可以应用在哪些方面？

例如：

| 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 |
| *Disk* 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 |

问题：缓存技术的实现手段和目的各是什么？

*将大容量慢速存储器中当前刚用过的局部数据复制或暂存在小容量快速存储器中，由于信息访问的局部性特点，可提高总体访问效率。*



## 第二讲小结

- 引入Cache的基础是程序访问的局部化特性
  - 时间局部性和空间局部性
- 引入Cache减少了对内存的访问，CPU能在快速的Cache中得到信息
- Cache和主存之间的映射方式
  - 直接映射（模映射）：地址=标志 | cache行索引 | 块内地址
  - 全相联映射（全映射）：地址=标志 | 块内地址
  - 组相联映射（组间模映射，组内全映射）：地址=标志 | cache组索引 | 块内地址
- 如何提高cache的命中率？
  - 增大cache容量，适中的块大小
  - 采用多级cache技术(2级或3级等)
  - 采用快速查找算法，并采用并行判定是否命中
  - 不能命中时，采用有效的算法将读入的内容替换cache中暂时不使用的内容
  - 编译器优化目标程序
  - 程序员写出cache-friendly的程序
- Cache的写策略
  - Write Back 和Write Through



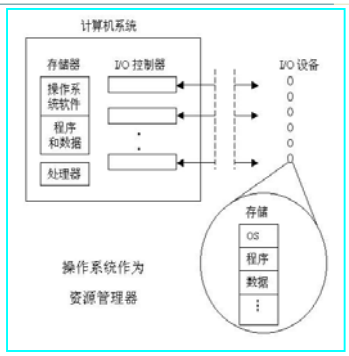
## 第三讲 虚拟存储器

**主 要 内 容**

- 存储管理技术的发展过程
- 虚拟存储器的基本概念
  - 按需调页
  - 虚拟地址空间
- 虚拟存储器方式
  - 三种方式：页式、段式、段页式
  - 逻辑地址--物理地址的转换
  - 页表
  - 缺页处理
- 替换策略
- 快表
- 存储保护
  - 地址越界检查
  - 存取权限检查

## 存储器资源的管理由操作系统来实现

- 操作系统(OS)通过合理地管理、调度计算机的硬件资源，使其高效被利用。
- 存储器作为一种空间资源也由OS来管理
- CPU执行的程序总是在操作系统和用户程序之间切换。
- 主存中同时要存储OS和用户程序。磁盘中也要存储OS和用户程序
- CPU执行指令时，涉及到存储器操作，因此，CPU中专门有一个存储器管理部件MMU（Memory Management Unit）协助OS完成存储器访问




OS为“进程”分配存储器资源，所以，先了解一下进程的概念。



## 复习：一个典型程序的转换处理过程

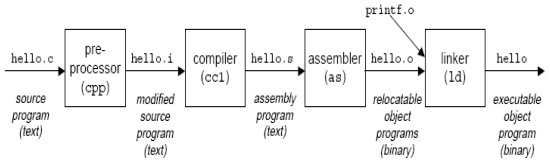
经典的“ hello.c ”C-源程序

```
1 #include <stdio.h>
2
3 int main()
4 {
5 printf("hello, world\n");
6 }
```

程序的功能是：
输出“hello,world”

hello.c的ASCII文本表示

```
# i n c l u d e <sp> < s t d i o .
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46
h > \n \n i n t <sp> m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n <sp> <sp> <sp> <sp> p r i n t f ( " h e l
10 32 32 32 32 112 114 105 110 116 102 40 34 104 101 108
l o , <sp> w o r l d \ n " ) ; \n }
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```






## 复习：Hello程序的执行过程

Unix系统启动可执行程序hello的shell命令行：

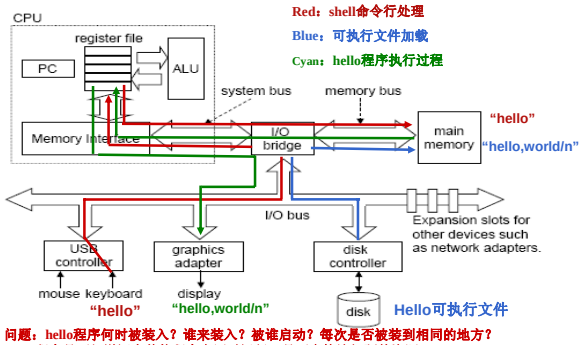
```
unix> ./hello [Enter]
hello, world
unix>
```

Hello程序被启动后，计算机的动作过程如下：

1. Shell程序读取字符串“./hello”中各字符到寄存器，然后存放到主存；
2. “Enter”键输入后，shell调用驻留在内存的“加载器”程序，由加载器根据主存中的字符串“hello”到磁盘上找到特定的hello目标文件，将其包含的指令代码和数据（“hello, world\n”）从磁盘读到主存；
3. 处理器从hello主程序的指令代码开始执行；
4. Hello程序将“hello, world\n”串中的字节从主存读到寄存器，再从寄存器输出到显示器上



## 复习：Hello程序的数据流动过程

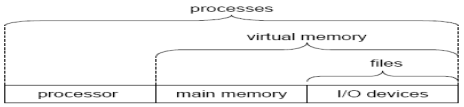



问题：hello程序何时被装入？谁来装入？被谁启动？每次是否被装到相同的地方？
Hello程序是否知道还有其他程序在同时运行？是否直接访问硬件资源？



## 操作系统在程序执行过程中的作用

- 在Hello程序执行过程中，Hello程序本身没有直接访问键盘、显示器、磁盘和主存储器这些硬件资源，而是依靠操作系统提供的服务来间接访问。 例如，调用printf函数访问硬件。
- 操作系统是在应用程序和硬件之间插入的一个中间软件层。
- 操作系统的两个主要的作用：
  - 硬件资源管理，以达到以下两个目的：
    - 统筹安排和调度硬件资源，以防止硬件资源被用户程序滥用
    - 对于广泛使用的复杂低级设备，为用户程序提供一个简单一致的使用接口
  - 为用户使用系统提供一个操作接口
- 操作系统通过三个基本的抽象概念（进程、虚拟存储器、文件）实现硬件资源管理
  - 文件（files）是对I/O设备的抽象表示
  - 虚拟存储器（Virtual Memory）是对主存和磁盘I/O的抽象表示
  - 进程（processes）是对处理器、主存和I/O设备的抽象表示

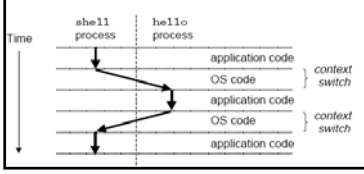





## 进程（processes）

```
unix> ./hello [Enter]
hello, world
unix>
```

- Hello程序运行时，Hello程序会以（错觉）：
  - 所有系统资源都被自己独占使用
  - 处理器始终在执行本程序的一条条指令
- 进程是操作系统对运行程序的一种抽象
  - 一个系统上可以同时运行很多进程，但每个进程都好像自己是独占使用系统
  - 实际上，操作系统让处理器交替执行很多进程中的指令
  - 操作系统实现交替指令执行的机制称为“上下文切换（context switching）”
- 进程的上下文
  - 指进程运行所需的所有状态信息，例如：PC、寄存器堆的当前值、主存的内容、段/页表
  - 系统中有一套状态单元存放当前运行进程的上下文
- 上下文切换过程（任何时刻，系统中只有一个进程正在运行）
  - 上下文切换指把正在运行的进程换下，换一个新进程到处理器执行，上下文切换时，必须保存换下进程的上下文，恢复换上进程的上下文




开始，Shell进程等待命令行输入
输入“Hello”后Shell进行系统调用
OS保存shell上下文，创建并换入hello进程
Hello进程中止时，进行系统调用
OS恢复shell进程上下文，并换入shell进程

由于在一个进程的整个生命周期中，有不同进程在处理器交替运行，所以运行时间很难准确、重复测量。

## 存储器管理(Memory Management)

- 早期采用单道程序，系统的主存中包含：
  - 操作系统（常驻监控程序）
  - 正在执行的一个用户程序

  所以无需进行存储管理，即使有也很简单。
- 现在都采用多道程序，系统的存储器中包含：
  - 操作系统
  - 若干个用户程序

  如果在存储器中进程数很少，则由于进程花费很多时间来等待I/O，常使处理机处于空闲状态。因此，存储器需要进行合理分配，尽可能让更多进程进入存储器。
- 在多道程序系统中，存储器的“用户”部分须进一步划分以适应多个进程。划分的任务由OS动态执行，这被称为存储器管理(memory management)。



## Memory Management Schema

- 使系统中尽量多地存储用户程序的解决办法有：

  (1) 扩大主存(程序越来越长、主存贵，不是根本办法)

  (2) 采用交换(Exchange)方式和覆盖(Overlap)技术

  存储器中无处于就绪状态的进程（例如：某一时刻所有进程都在等待I/O）时，处理器将一些进程调出写回到磁盘，然后OS再调入其他进程执行，或新的作业直接覆盖老作业的存储区。

  分区(Partitioning)和分页(Paging)是交换的两种实现方式

  “交换”和“覆盖”技术的缺点：对程序员不透明、空间利用率差

  (3) 虚拟存储器(Virtual Memory)

  类似上述分页方式，但不是把所有页面一起调到主存，而是采用“按需调页Demand Paging”，在外存和主存间以固定页面进行调度。

  虚拟存储器方式下，引入了虚拟地址空间的概念。

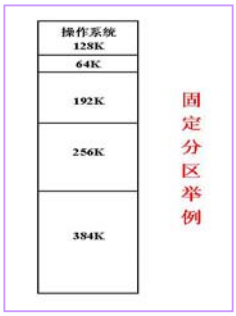
SKIP



## 简单分区（Partitioning）

- 主存分配：
  - 操作系统：固定
  - 用户区：分区
- 简单分区方案：使用长度不等的固定长分区(fixed-size partition)。
- 当一个进程调入主存时，分配给它一个能容纳它的最小的分区。

  例如，对于需196K的进程可分配256K的分区。
- 简单分区方式的缺点：
  - 因为是固定长度的分区，故可能会浪费主存空间。多数情况下，进程对分区大小的需求不可能和提供的分区大小一样。



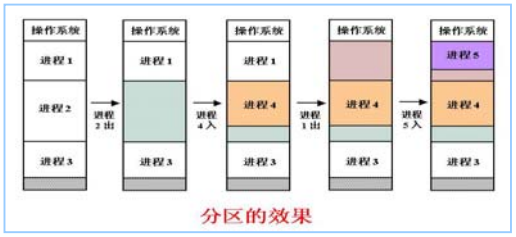

问题：如何生成物理地址？

可以采用更有效的可变长分区的方式！



## 可变长分区（Partitioning）

- 更有效的方式----可变长分区　(variable-length partition)
  - 分配的分区大小与进程所需大小一样。
  - 特点：开始较好，但到最后在存储器中会有许多小空块出现。时间越长，存储器中的碎片就会越来越多，因而存储器的利用率下降。

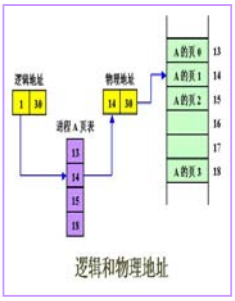



更有效的方式是分页！



## 分页（Paging）

- 基本思想：
  - 把内存分成固定长且比较小的存储块，每个进程也被划分成固定长的程序块
  - 程序块（页/page）可装到存储器可用的存储块(页框/page frame)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
  - 通过页表(page table)实现逻辑地址向物理地址转换（Address Mapping ）
- 逻辑地址（Logical Address）：
  - 程序中的指令所用的地址
- 物理地址（physical或Memory Address）：
  - 存放指令或数据的实际内存地址




问题：是否需要将一个进程的全部都装入到内存？

根据程序访问的局部性可知：可以仅把当前活跃的页面调入主存，其余留在磁盘上！

浪费的空间最多是最后一页的部分！

采用 “按需调页 / Demand Paging”方式对主存进行分配！

BACK

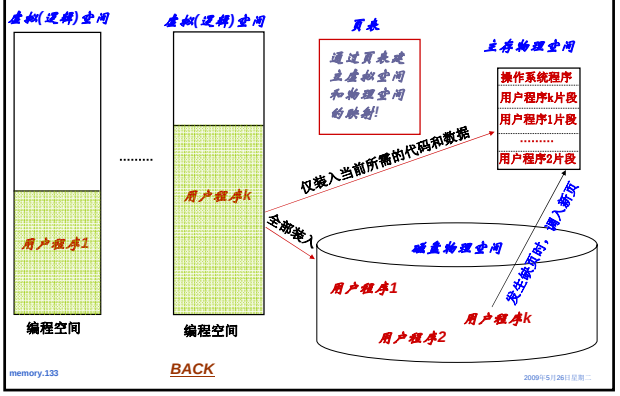


## 虚拟存储系统的基本概念

- 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。

下面先介绍操作系统中一些有关概念。

## 虚拟存储技术的实质



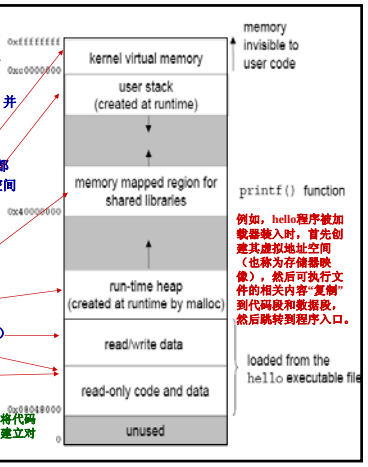



## 虚拟地址空间

- 虚存为每个进程提供了一个假象
  - 好像每个进程都独占使用主存，并且主存空间极大
- 虚存是主存和磁盘I/O设备的抽象
  - OS使每个进程看到的存储空间都一致，称为虚拟（逻辑）地址空间
- Linux操作系统的虚拟地址空间（其他Unix系统的设计类此）
  - 内核（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？

实际上不会从磁盘调入，只是将代码和数据按“组块”与虚拟空间建立对应关系，称为“映射”。






## MIPS程序和数据的存储器分配

- 每个MIPS程序都按如下规定进行存储器分配
- 每个可执行文件都按如下规定给出代码和数据的地址

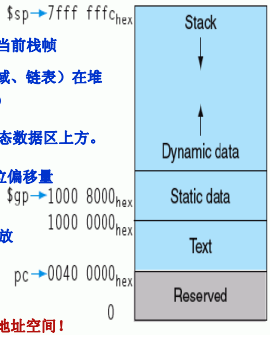
堆栈在高地址区，从高到低增长。
过程调用时，生成当前“栈帧”，返回后退回当前栈帧

程序的动态数据（如：C中的malloc申请区域、链表）在堆(heap)中从低向高进行存放和释放（free时）

栈区位于堆栈高端，堆区位于堆栈低端，静态数据区上方。

全局指针$gp固定设为0x1000 8000，其16位偏移量的访问范围为0x1000 0000 到0x1000 ffff

静态数据区从固定的0x1000 0000处 开始存放
程序代码从固定的0x0040 0000处开始存放
故PC的初始值为0x0040 0000




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：

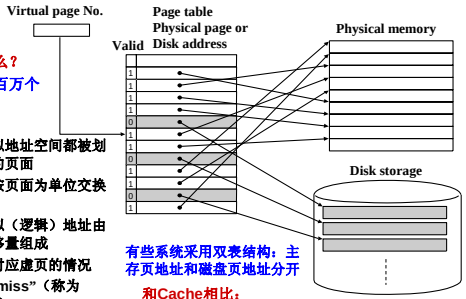
块大小（在虚拟存储器中“块”被称为“页 / Page”）应多大?
主存与辅存的空间如何分区管理?
程序块 / 存储块之间如何映像?
逻辑地址和物理地址如何转换，转换速度如何提高?
主存与辅存之间如何进行内容调换（与Cache所用策略相似）？
页表如何实现，页表项中要记录哪些信息?
如何加快访问页表的速度?
如果要找的内容不在主存，怎么办?
如何保护进程各自的存储区不被其他进程访问?

- 有三种虚拟存储器实现方式：

分页式、分段式、段页式



## 分页式系统




缺页的代价是什么？
读磁盘（需花几百万个时钟周期！）

- 物理存储器和虚拟地址空间都被划分成大小物相等的页面
- 磁盘和主存之间按页面为单位交换信息
- 指令中给出的虚拟（逻辑）地址由虚页号和页内偏移量组成
- 每个页表项记录对应虚页的情况
- Valid为“0”说明“miss”（称为page fault / 缺页）
- CPU执行指令时，首先需要将逻辑地址转换为主存的物理地址
- 地址转换由CPU中的MMU实现

有些系统采用双表结构：主存页地址和磁盘页地址分开

和Cache相比：
页大小比Cache中Block大得多！32KB~64KB
采用全相联映射！ Why?
通过软件来处理“缺页”! Why?
采用Write Back写策略！ Why?



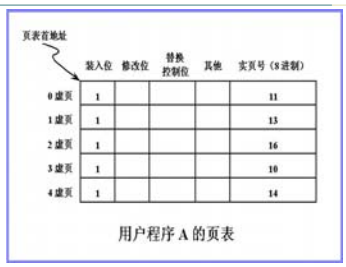
## 页表结构

- 每个进程有一个页表
- 典型页表中有装入位、修改（Dirt）位、替换控制位、访问权限位、禁止缓存位、实页号
- 页表项数由进程大小决定
- 页表在主存中的首址记录在页表基址寄存器中
- 必须解决页表占空间过大的问题

（1）页表可能很大，为了让一个进程具有很大的虚拟编程空间，系统必须允许页表的项数足够多

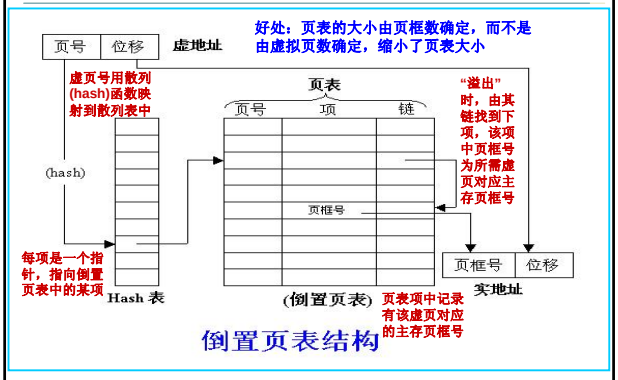
例如：在VAX系统中，每个进程能拥有高达$2^{31}=2G$字节的虚拟存储器，按512字节/页进行分页，则每个进程最多可达$2^{22}$个页表项。显然，这么大的页表全部放在主存中是不适合的

（2）系统中有许多进程，每个进程有一个页表



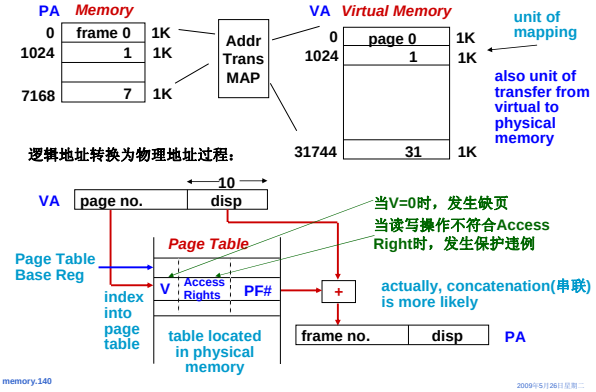

- 解决页表过大的方法
  - 一级页表：动态扩充，限制大小
  - 一级页表：分两个独立的段
  - 二级或多级页表：一级为段、二级为页
  - 将页表分页，当前使用的页的页表项所在页表在内存，其他在外存，页表也要调进调出
  - 反向（倒置）页表

## 倒置页表(inverted page table)结构






## 逻辑地址转换为物理地址的过程






## 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：从磁盘中读信息到内存，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，页面淘汰时，根据"dirty"位确定是否要写磁盘

异常处理结束后：当前指令的执行被阻塞，当前进程被挂起，处理结束后回到原指令继续执行

2）保护违例（ protection_violation_fault ）

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"信息

异常处理结束后：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



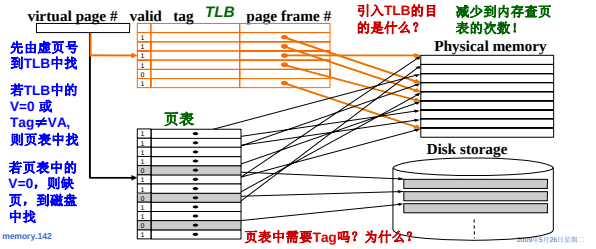
## TLBs --- Making Address Translation Fast

问题：一次内存引用要访问几次内存？ 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全相联时，则没有index，只有Tag，虚页号需与每个Tag比；若组相联时，则虚页号高位为Tag，低位为index，用作组索引。

| Virtual Address (tag+index) | Physical Address | Dirty | Ref | Valid | Access |
|---|---|---|---|---|---|
| 虚页号分成tag+Index | 对应物理页框号 | | | | |






## Translation Look-Aside Buffers

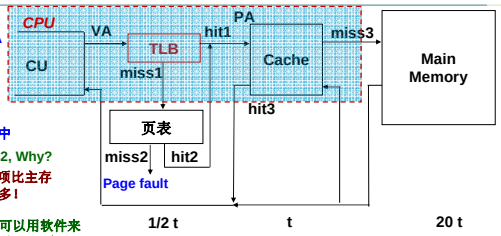
Miss1: TLB中没有VA

Miss2: 页面不在主存

Miss3: PA 在主存中，但不在Cache中

miss1>>miss2, Why?

因为TLB中的项比主存中的页数少得多！




可以用硬件也可以用软件来处理TLB失靶（miss1）

Miss1的处理：查内存页表，若发生hit2，则将表项装入TLB中，并进行地址转换；否则"缺页"，引起相应的异常处理

大多用全相联：命中率高，因为小,故成本不高

采用随机替换策略：降低替换算法的开销

采用回写策略：减少访问内存的次数

TLB的一些典型指标：

TLB大小：16~512项

块大小：1~2项(每个表项4-8B)

命中时间：0.5~1个时钟周期

失靶损失：10~100个时钟周期

命中率：90~99%



## 举例：三种不同缺失的组合

- 三种不同缺失：TLB缺失、Cache缺失、缺页
- 三种缺失的组合情况的可能性分析

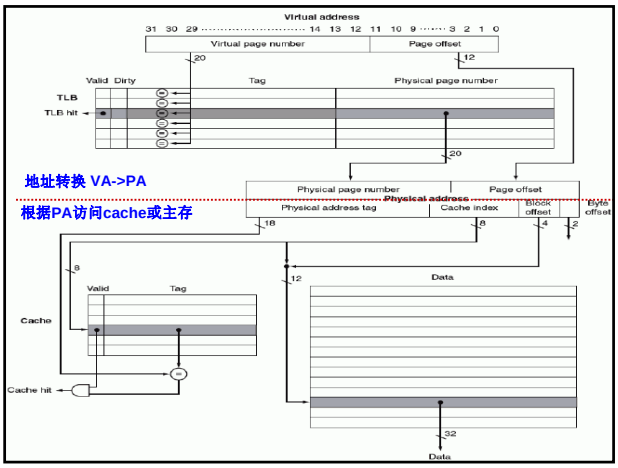
| TLB | Page table | Cache | Possible? If so, under what circumstance? |
|---|---|---|---|
| hit | hit | miss | 可能，TLB命中则页表一定命中，但实际上不会查页表 |
| miss | hit | hit | 可能，TLB缺失但页表可能命中，信息在主存，就可能在Cache |
| miss | hit | miss | 可能，TLB缺失但页表可能命中，信息在主存，但可能不在Cache |
| miss | miss | miss | 可能，TLB缺失页表可能缺失，信息不在主存，一定也不在Cache |
| hit | miss | miss | 不可能，页表缺失，说明信息不在主存，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，说明信息不在主存，Cache中一定也没有该信息 |

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

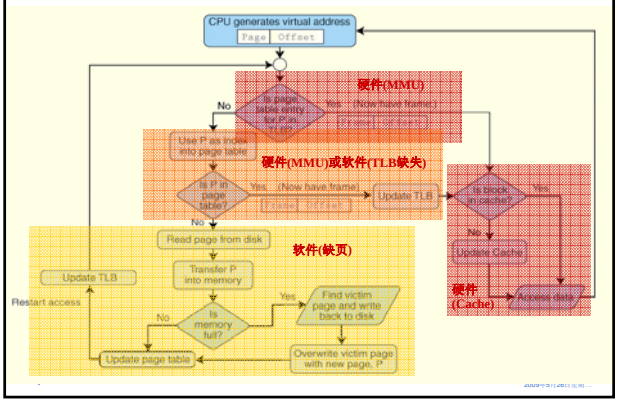
以上组合中，最好的情况是什么？hit、hit、miss和miss、hit、hit 只需访问主存1次

以上组合中，最坏的情况是什么？ 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什么？ miss、hit、miss 不需访问磁盘、但访存至少2次




## Putting VM、TLB、Cache all together

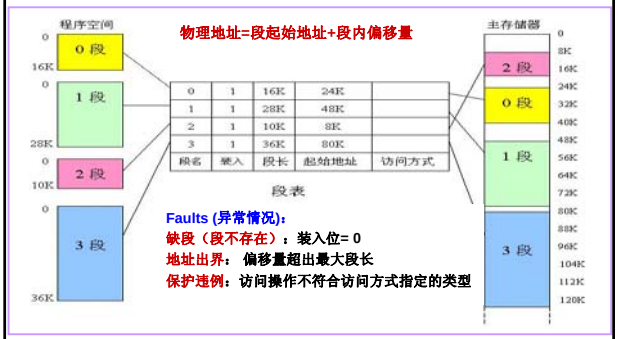



## 分页式和分段式的比较

- 分页系统的特点
  - 优点：实现简单，开销少。因为只有进程的最后一个零头（内部碎片）不能利用，故浪费很小
  - 缺点：由于页不是逻辑上独立的实体，因此可能会出现如“一条指令跨页”等问题，使处理、管理、保护和共享等都不方便
- 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，过程、子程序、数据表、阵列等）
  - 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
  - 段可作为独立逻辑单位被其他程序调用，以形成段间连接，产生规模较大的程序
  - 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
  - 段表本身也是主存中的一个可再定位段
  - 因为段本身是程序的逻辑结构所决定的一些独立部分，因而分段对程序员来说是不透明的（而分页对程序员来说是透明的）

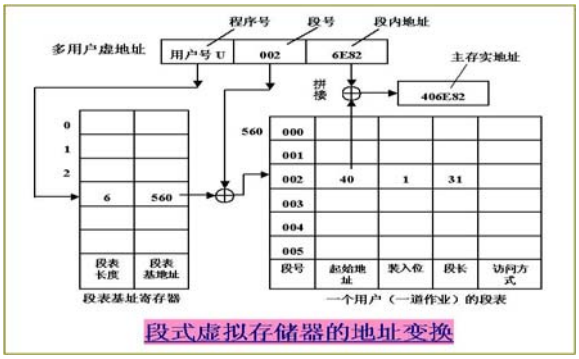


## 段式虚拟存储器的地址映像






## 段式虚拟存储器的地址变换图

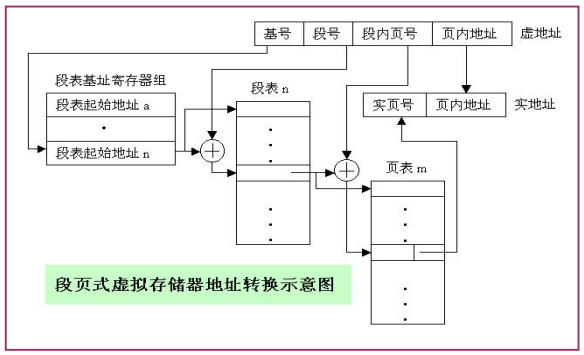





## 分段式和段页式的比较

- 分段式系统的特点
  - 优点：段的分界与程序的自然分界对应，故段具逻辑独立性，易于编译、管理、修改和保护，便于多道程序共享；某些类型的段（堆栈、队列）具有动态可变长度，允许自由调度以有效利用主存空间
  - 缺点：段长各不相同，起、终点不定，变化很大，给主存分配带来麻烦，且易在段间留下许多空余的零碎空间，不好利用，造成浪费
    （例如：一个长段调出后，调进一个短段就会造成碎区）
- 段页式系统基本思想
  - 段、页式结合。程序按模块分段，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指针，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

## 段页式虚拟存储器的地址变换






## （自学）Pentium处理器的寻址方式

**操作数的来源：**

- 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)

  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)

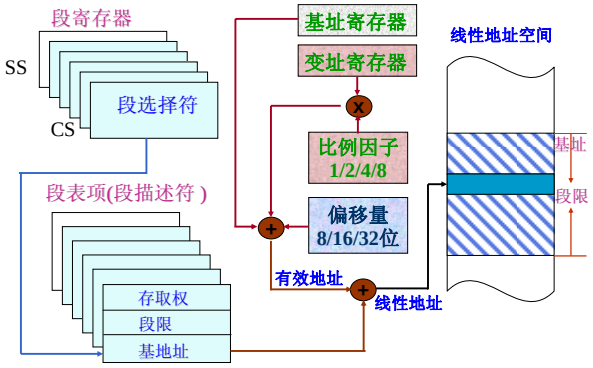


## （自学） Pentium处理器寻址方式

| 寻址方式 | 算法 |
|---|---|
| 立即(地址码A本身为操作数) | 操作数=A |
| 寄存器(通用寄存器的内容为操作数) | 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)xS+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)xS+A |
| 相对(给出下一指令的地址，转移控制) | 转移地址=(PC)+A |



## （ 自学）Pentium处理器的存储器寻址






## 存储保护的基本概念

- 什么是存储保护？
  - 为避免主存中多道程序相互干扰，防止某程序出错而破坏其他程序的正确性，或某程序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）
    - 访问重定位、键保护、环保护
  - 访问越权（访问操作与所拥有的访问权限不符）
    - 页表中设定权限
- 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
  - 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

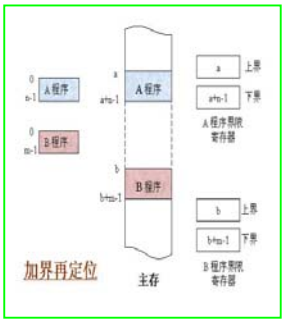
- 为了对操作系统的存储保护提供支持，硬件必须具有以下三种基本功能：
  - 支持至少两种运行模式：
    - 管理模式(Supervisor Mode)

      用于完成操作系统各种功能的进程为系统进程，也称为内核(Kernel)进程、管理(Supervisor)进程。执行系统进程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
    - 用户模式(User Mode)

      完成非操作系统功能的进程称为用户进程，当系统运行用户进程时，处理器模式就是用户模式，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状态只能由系统进程写而不能由用户进程写（只能读）：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）来写这些状态
  - 提供让CPU在管理模式和用户模式相互转换的机制：“异常”和“陷阱”（系统调用）使CPU从用户模式转到管理模式；异常处理中的“返回”指令（return from exception）使CPU从管理状态转到用户状态
- 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，并防止用户进程改变页表，确保用户进程只能访问由OS分配给的存储空间

## 通过程序重定位进行存储区域保护

- 把逻辑地址转换为实际的物理地址的过程称为“地址转换”或“程序重定位”
- 重定位方式：
  - **静态**：在装入前将所有地址全部转换为物理地址。
  - **动态**：靠硬件的地址转换机构来实现，在程序执行过程中动态进行地址转换。动态定位可实现程序在主存中的浮动。
- 程序重定位是通过逻辑地址加界（即加基准地址）来实现的
- 通过对程序生成的地址进行越界判断，可实现程序的保护




对程序生成的地址进行判断，若在界限内，则说明没有越界，否则访问越界



## 键保护和环保护方式进行存储区域保护

- 键保护方式
  - 系统为每道作业设置一个保护键
  - 为某作业分配主存时，根据它的保护键在页表中建立键标志
  - 进程运行时，将程序状态寄存器中的键(程序键)和访问页的键（存储键）进行核对，相符时才可访问该块，如同“锁”与“钥匙”的关系
  - 为使某块能被各进程访问，或某个进程可访问任何一块，规定键标志为0，此时不进行核对工作。例如，操作系统有权访问所有块，所以可让OS的程序状态字中的键为0
- 环保护方式
  - 主存中各进程按其重要性分为多个保护级，各级别构成同心环
  - 最内环的进程保护级别最高，向外逐次降低
  - 内环进程可以访问外环和同环进程的空间，而外环不得访问内环进程空间
  - 系统进程的保护级别高，环号小，而用户进程大都处于外环
  - Pentium采用该方案



## 第三讲小结

- 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，把一部分程序调到主存，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



## 本章总结1

- 存储器的分类
  - 按存取方式分：随机、顺序、直接、相联
  - 按存储介质分：半导体、磁表面、激光盘
  - 按信息可更改性：可读可写、只读
  - 按断电后可否保存：易失、非易失
  - 按功能/容量/速度分：寄存器、Cache、主存（内存）、辅存（外存）
- 存储器的分层结构：
  - 速度从快到慢、容量从小到大、价格从贵到便宜，按与CPU连接的距离由近到远的顺序，构成的分层次结构为：

    寄存器→Cache→主存→磁盘→光盘、磁带



## 本章总结2

- 半导体随机存取存储器的组织
  - 存储元（记忆单元）→存储芯片→存储模块（内存条）→存储器
- 存储器芯片与CPU的连接
  - **地址线的连接**：考虑芯片在字方向上扩展，低位用于芯片内地址、高位用于片选逻辑，送到片选信号译码器，译码输出连到芯片的片选信号引脚上。
  - **数据线的连接**：考虑芯片在位方向上扩展，分别连到位扩展的芯片上
  - **控制线的连接**：读/写信号、主存/IO访问信号等经过组合连到芯片相应的引脚。
- 只读存储器：MROM、PROM、EPROM、EEPROM、Flash ROM
- 多体交叉编址存储器
  - **连续编址**：按高位地址划分模块
  - **交叉编址**：按低位地址划分模块



## 本章作业

- 2(4), 2(8), 2(9), 2(10), 2(11)
- 3，4，6，7，11，12，14，17，18，20，22，23，24

已经学过的内容可以先做起来。4月14号交作业！