Chapter 5

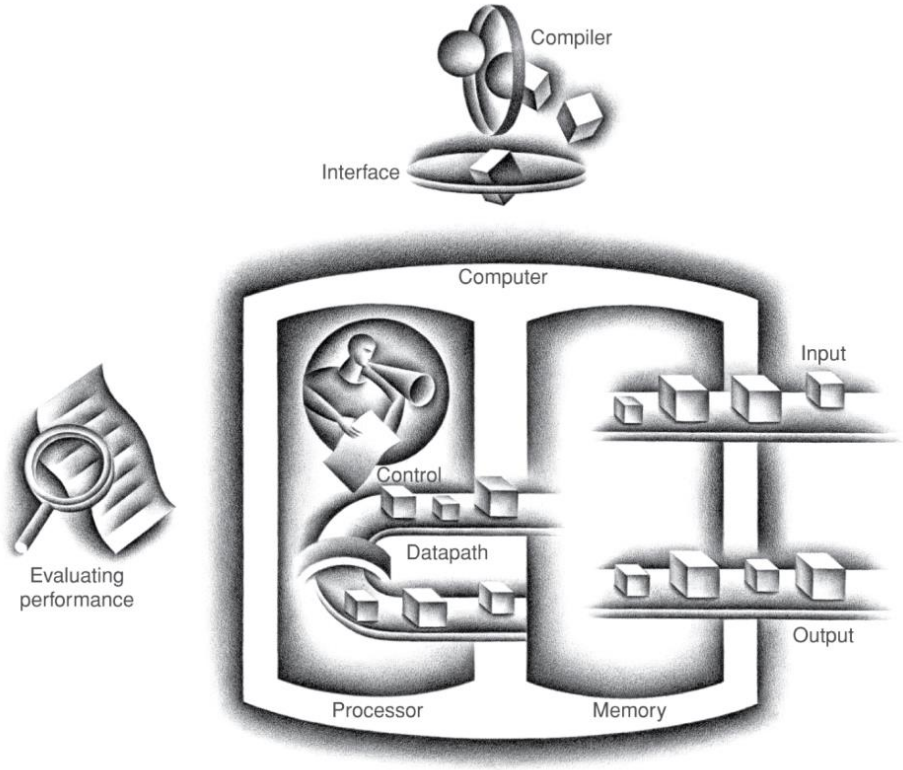
大容量、高性能: 利用内存层次结构

Outline

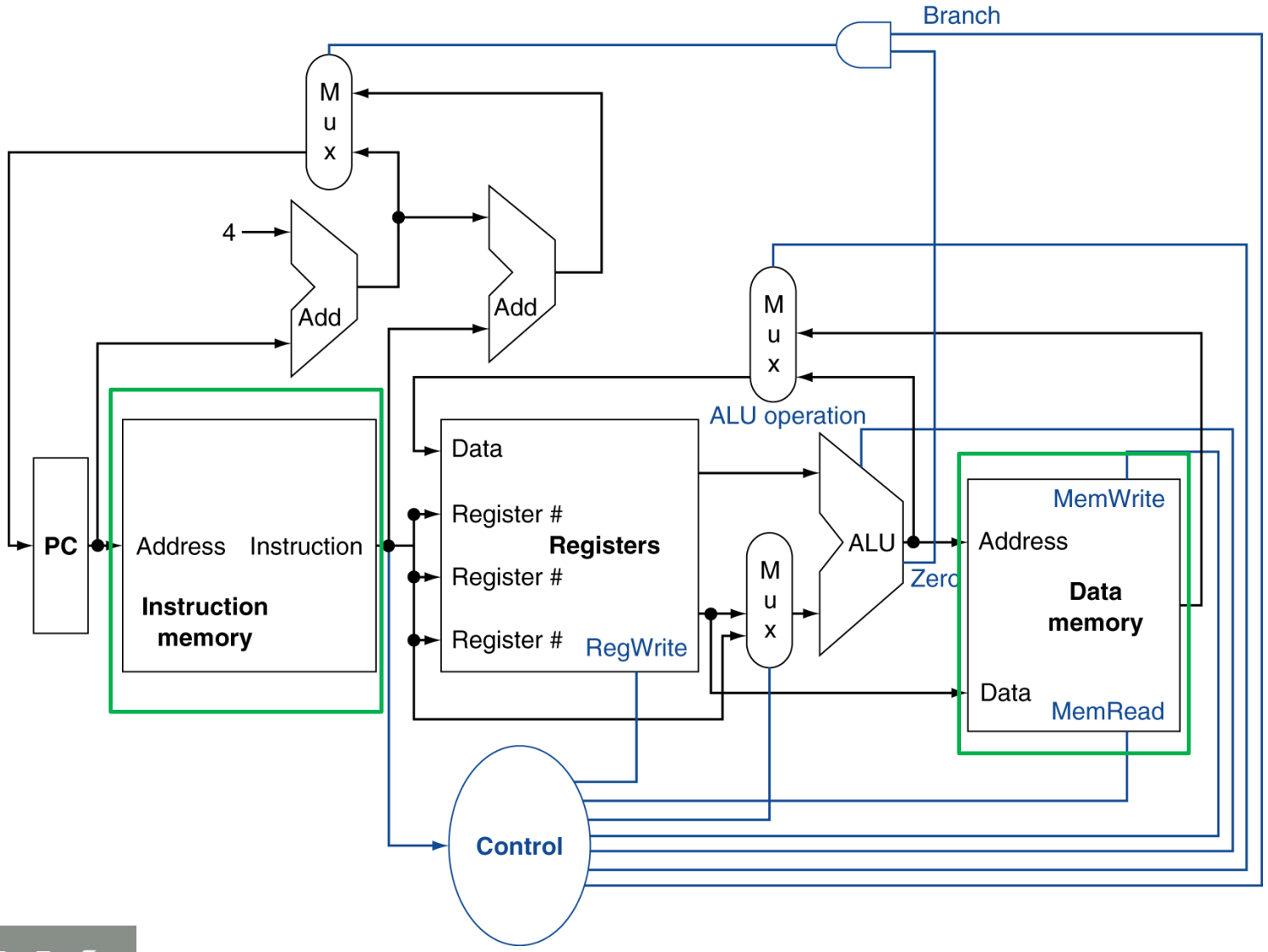
- **内存概述**
- 缓存基础
- 缓存的性能
- 可靠的内存技术
- 虚拟内存
- 一个通用框架

内存

- 计算机的组成
  - CPU
    - datapath & control
  - **memory内存**
  - I/O
- 内存在哪里?

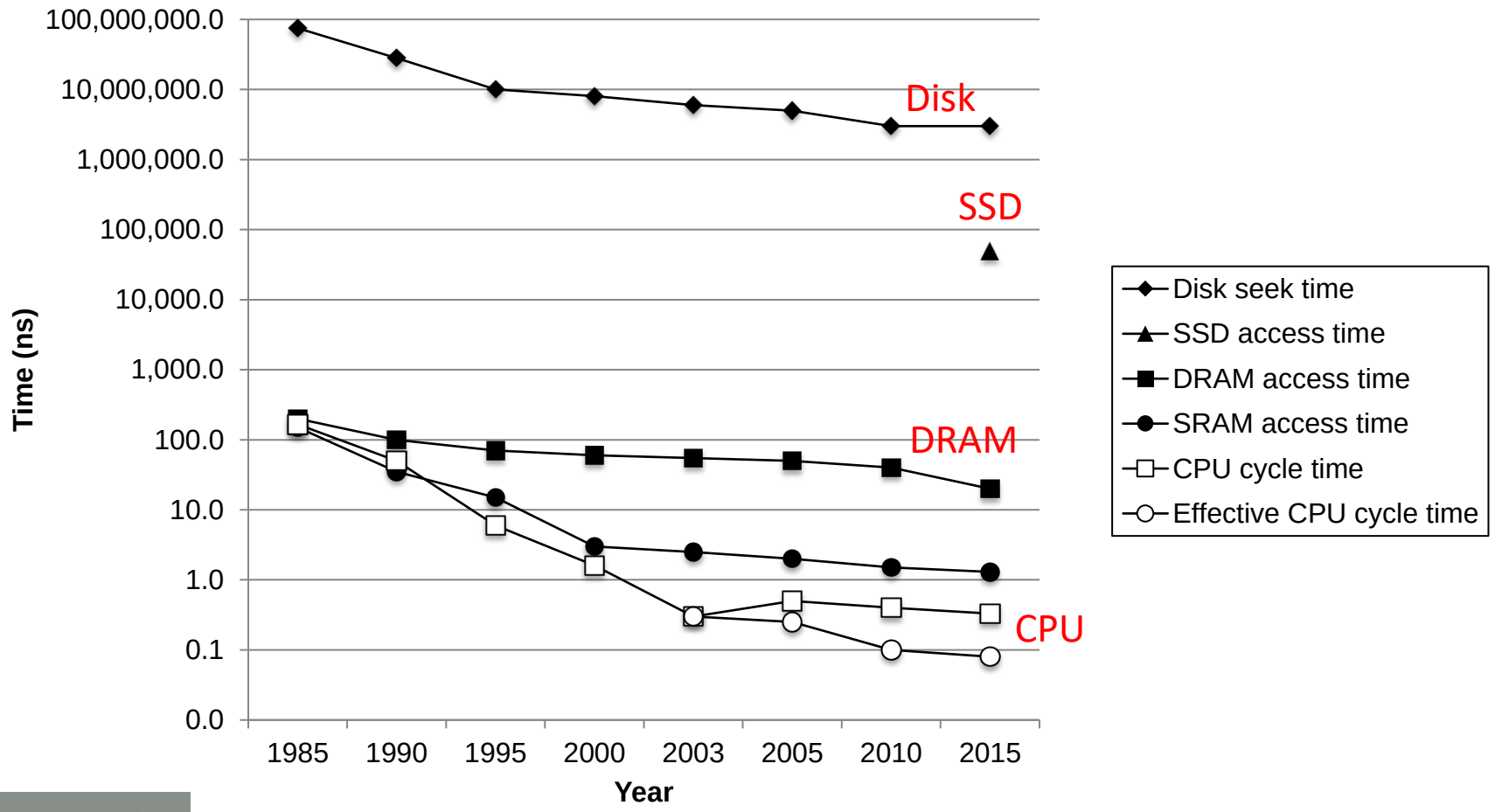


内存在哪里？


Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

处理器-内存的性能差距

CPU、内存、磁盘之间的性能差距，越来越大



内存墙

- 关于存储的挑战
  - 容量越大，速度越慢
  - memory速度远慢于CPU
- 成为性能瓶颈

度量标准	1985	1990	1995	2000	2005	2010	2015	2015:1985
美元/MB	2900	320	256	100	75	60	25	116
访问时间（ns）	150	35	15	3	2	1.5	1.3	115

a）SRAM趋势

度量标准	1985	1990	1995	2000	2005	2010	2015	2015:1985
美元/MB	880	100	30	1	0.1	0.06	0.02	44 000
访问时间（ns）	200	100	70	60	50	40	20	10
典型的大小（MB）	0.256	4	16	64	2000	8000	16 000	62 500

b）DRAM趋势

度量标准	1985	1990	1995	2000	2005	2010	2015	2015:1985
美元/GB	100 000	8000	300	10	5	0.3	0.03	3 333 333
最小寻道时间（ms）	75	28	10	8	5	3	3	25
典型的大小（GB）	0.01	0.16	1	20	160	1500	3000	300 000

c）旋转磁盘趋势

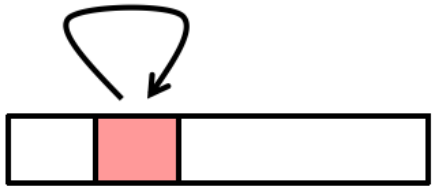
度量标准	1985	1990	1995	2000	2003	2005	2010	2015	2015:1985
Intel CPU	80 286	80 386	Pent.	P-Ⅲ	Pent.4	Core 2	Core i7 (n)	Core i7 (h)	—
时钟频率（MHz）	6	20	150	600	3300	2000	2500	3000	500
时钟周期（ns）	166	50	6	1.6	0.3	0.5	0.4	0.33	500
核数	1	1	1	1	1	2	4	4	4
有效周期时间（ns）	166	50	6	1.6	0.30	0.25	0.10	0.08	2075

克服内存墙：局部性!

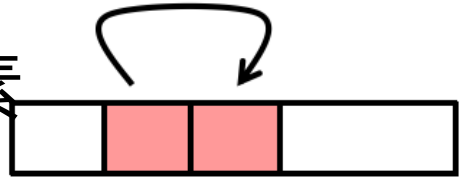
弥合CPU与内存之间差距的关键在于计算机程序的一个基本属性，即**局部性原理**

局部性原理

- 任何时候，程序只访问内存空间的一小部分
- 时间局部性（相同地址）
  - 最近访问的数据，很可能在不久的将来被再次访问
  - 比如，循环内部的指令和索引变量



- 空间局部性（附近地址）
  - 最近访问的数据旁边的数据，很可能在不久的将来被访问
  - 比如，顺序的指令访问、数组元素



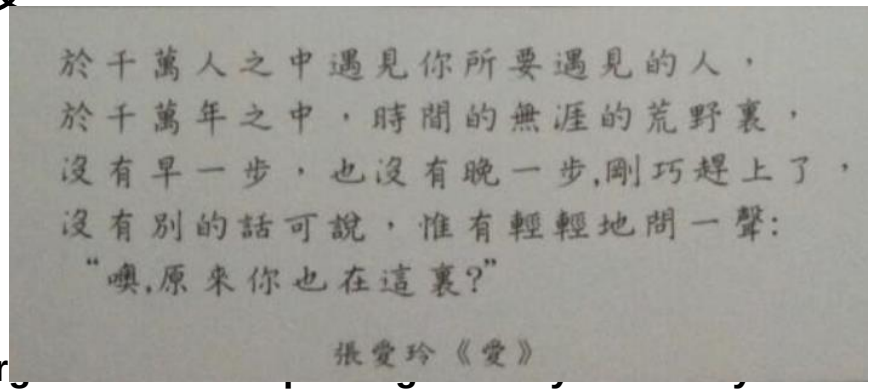
例子中的局部性分析

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

- 数据访问中的局部性
  - 时间局部性: ?
  - 空间局部性
    - array, structure, object, streaming data
  - 数据相关的构造:
    - 表量类型: char, int ,float, double
    - 聚合类型：
- 指令访问中的局部性:
  - 时间局部性： ?
  - 空间局部性： ?
  - 指令相关构造:
    - 循环, 分支、顺序

局部性的来源

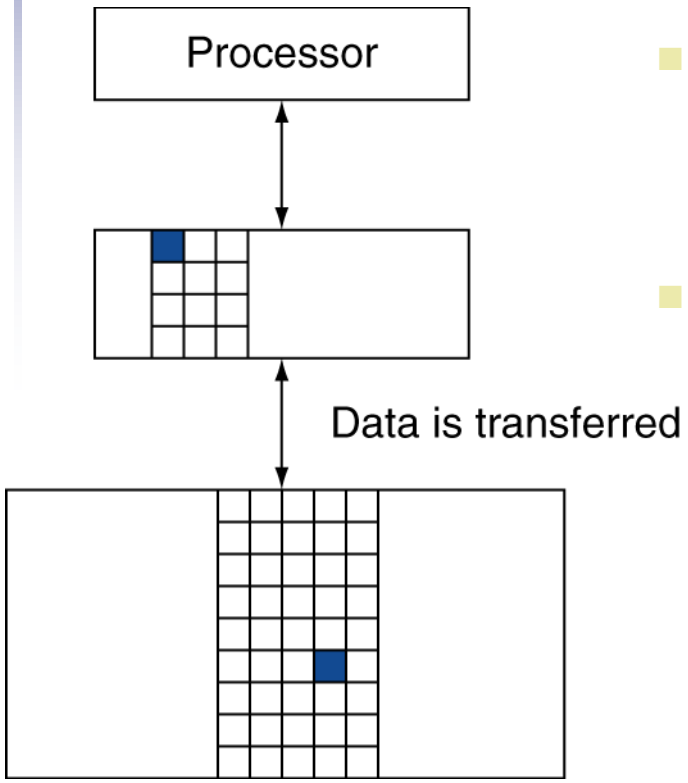
- 局部性从何而来——客观世界 or 主观大脑?
  - 客观上
    - 相邻时间发生的事件，具有更强的相关关系
      - 包括因果关系、顺序关系、同类关系等
    - 相邻空间发生的事件，也具有更强的相关关系
      - 包括同类关系、聚集关系等
  - 主观上
    - 希望提取相关关系，来解释客观现象；利用局部性，成本低
    - 时间上：记忆曲线；今日事今日毕；活在当下
    - 空间上：唯一的朋友是室友
- 一种优化思路：
  - 利用原有局部性 —cache
  - 改善原有局部性



利用局部性

- 内存层次
  - **磁盘**存放所有数据
  - 将**最近访问**（及**相邻**）的数据，从磁盘拷贝到更小、更快的**DRAM内存**
    - **Main memory**
  - 将**更加最近访问**（及**相邻**）的数据，从DRAM内存拷贝到更小、更快的SRAM内存
    - **SRAM**内存，通常指附属在**CPU**的**缓存**

内存层次的分层



- **块**(aka **行**): 拷贝的单位
  - 可能包括多个字
- 如果要访问的数据在更高的层次
  - **命中**: 更高的层次能够满足这次访问
    - **命中率**: 命中次数/访问次数
- 如果要访问的数据没在更高的层次
  1. 未命中: 从更低层次拷贝一**块到该层次**
     - 需要更多时间: 未命中的惩罚
     - **未命中率**: 未命中次数/访问次数
       = 1 – **命中率**
  2. 然后，可以从更高层次访问该数据

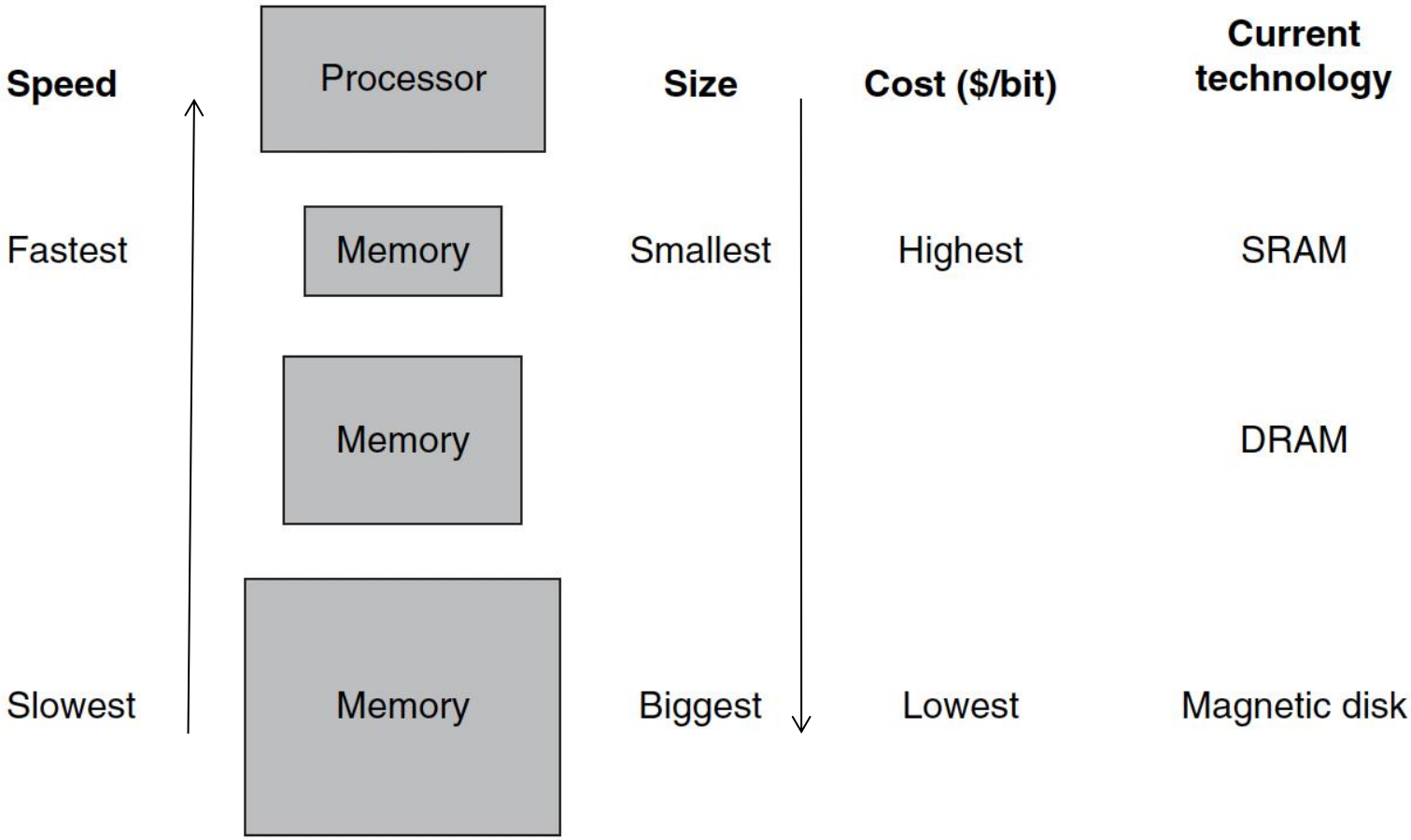


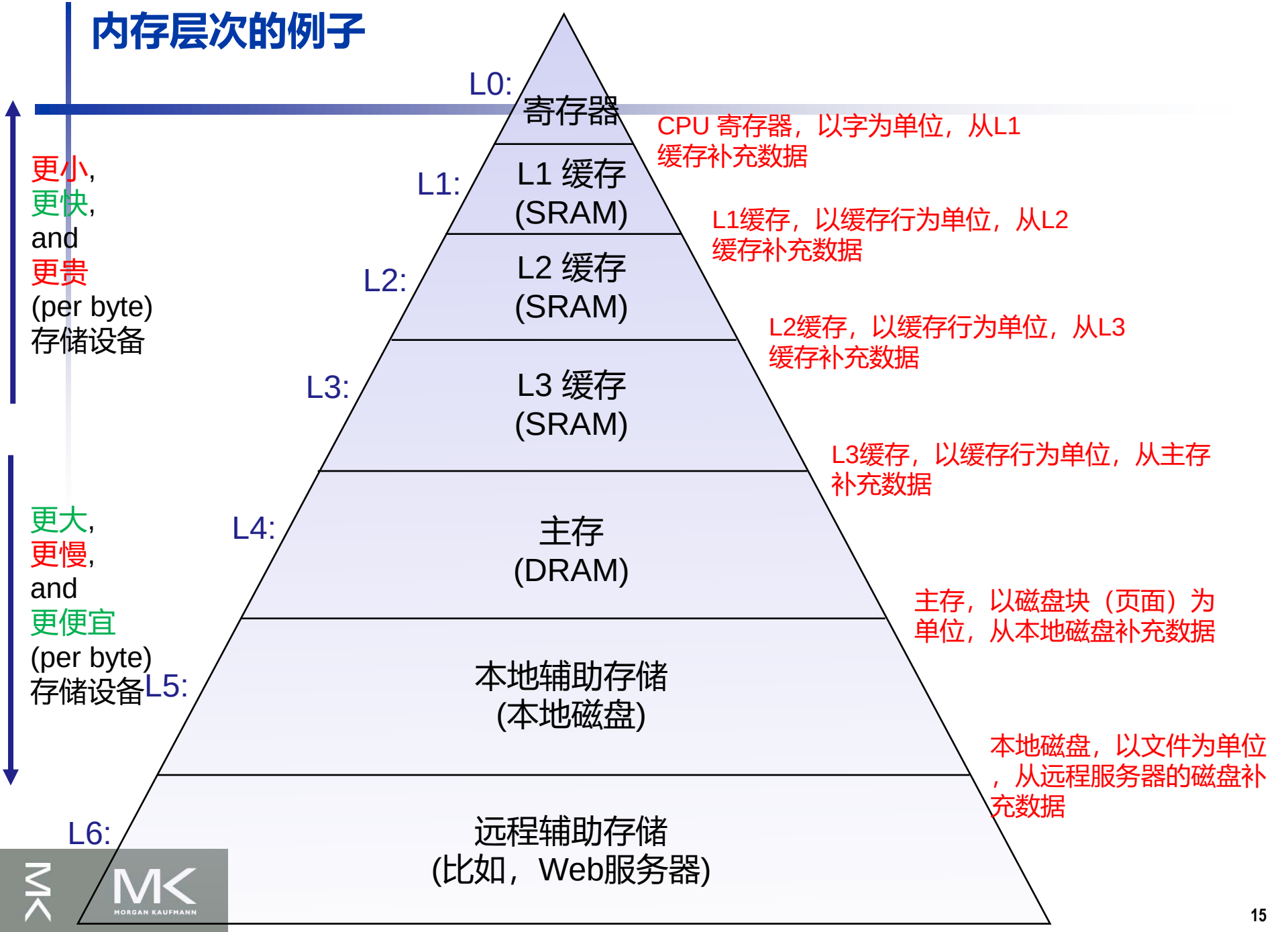
FIGURE 5.1 **The basic structure of a memory hierarchy.** By implementing the memory system as

内存技术

- 理想的内存
  - 访问速度媲美SRAM
  - 存储容量 & 单位价格，媲美磁盘

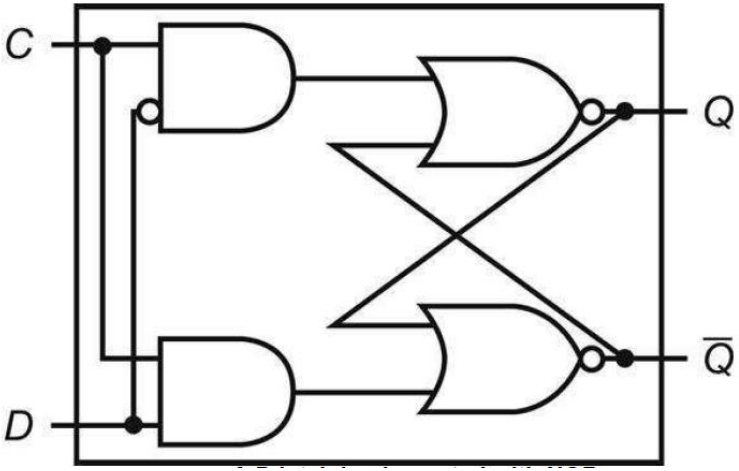
Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

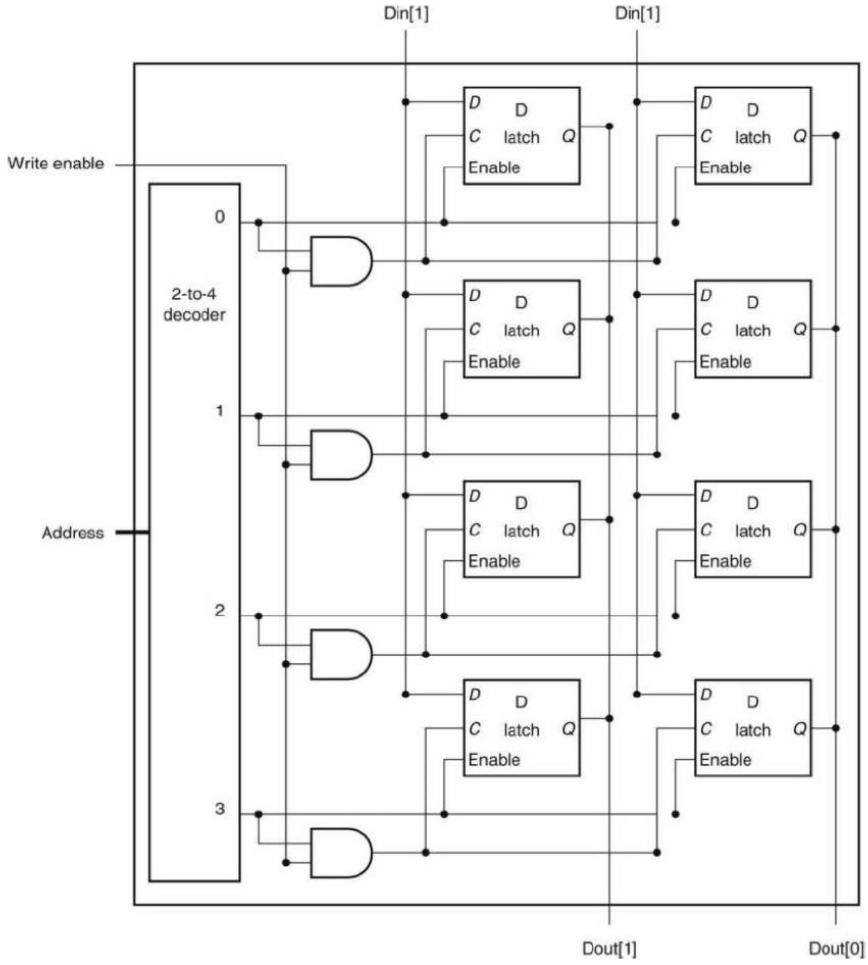
内存层次的例子



SRAM

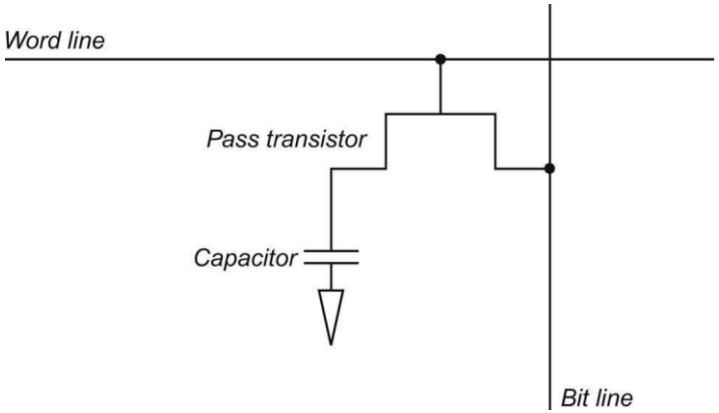
- 每个bit需6-8个逻辑门
- 贵、快，容量小

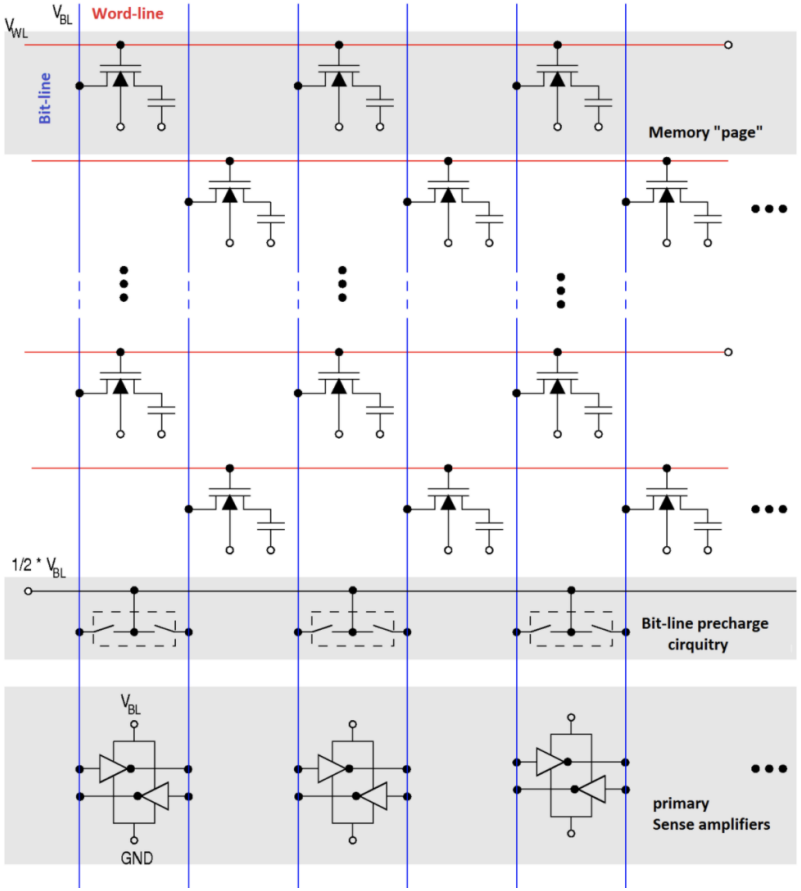




DRAM

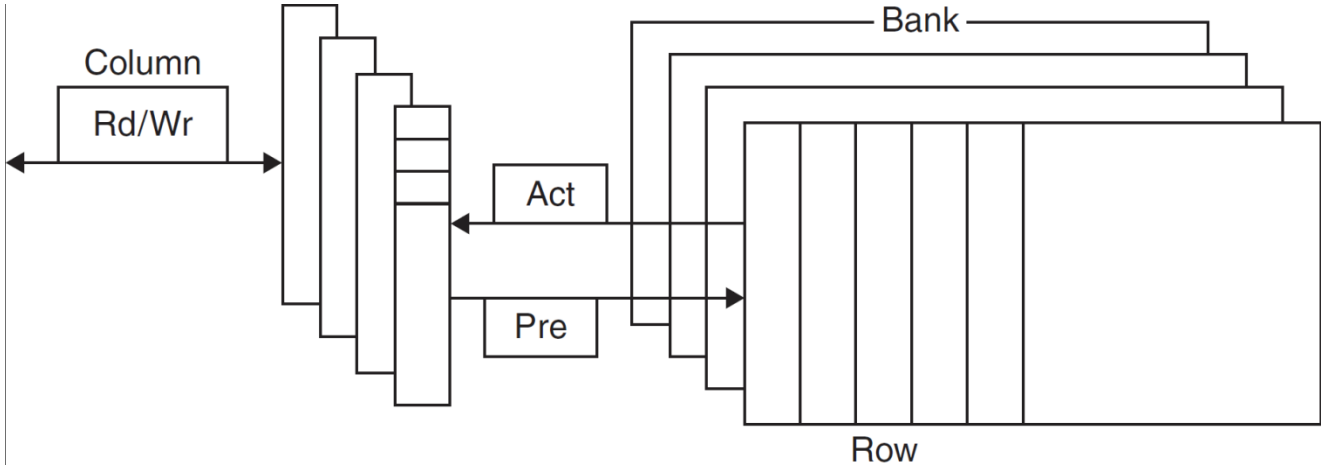
- 用**电容上**的电荷来存储0/1
- 需要一个**晶体管**来访问电容
- 便宜、慢，但容量大





DRAM 技术

- 用电容上的电荷表示数据
  - 使用单个晶体管来访问电荷
  - 必须周期性地刷新
    - 刷新：读取内容然后写回去
    - 以DRAM“行”为单位进行刷新，否则就太忙了



先进DRAM的组成

- DRAM中的位被组织成一个矩形数组
  - DRAM内部以行为单位进行访问
  - 突发模式: 从一个行中连续供应多个字，以便提升吞吐量
    - 缓存、磁盘中是否也有突发模式?
  - Double data rate (DDR) DRAM 双倍数据速率
    - 在时钟的上升/下降边沿都进行传送
  - Quad data rate (QDR) DRAM 四倍数据速率
    - DDR的输入和输出独立进行

影响DRAM性能的因素

- Row buffer行缓冲区
  - 允许**并行地**读和刷新多个字
- 同步DRAM
  - DRAM使用跟CPU不一样的时钟
  - 发送一次地址后，允许以突发的模式连续访问相邻的地址
  - 提升贷款
- DRAM banking
  - 允许同时访问多个DRAM
  - 提升吞吐量

MacOS下的内存条信息

内存插槽	大小	类型	速度	状态
▼内存插槽				
BANK 0/ChannelA-DIMM0	8 GB	LPDDR3	2133 MHz	好
BANK 2/ChannelB-DIMM0	8 GB	LPDDR3	2133 MHz	好

内存插槽：

ECC：　已停用
可升级内存：　否

BANK 0/ChannelA-DIMM0：

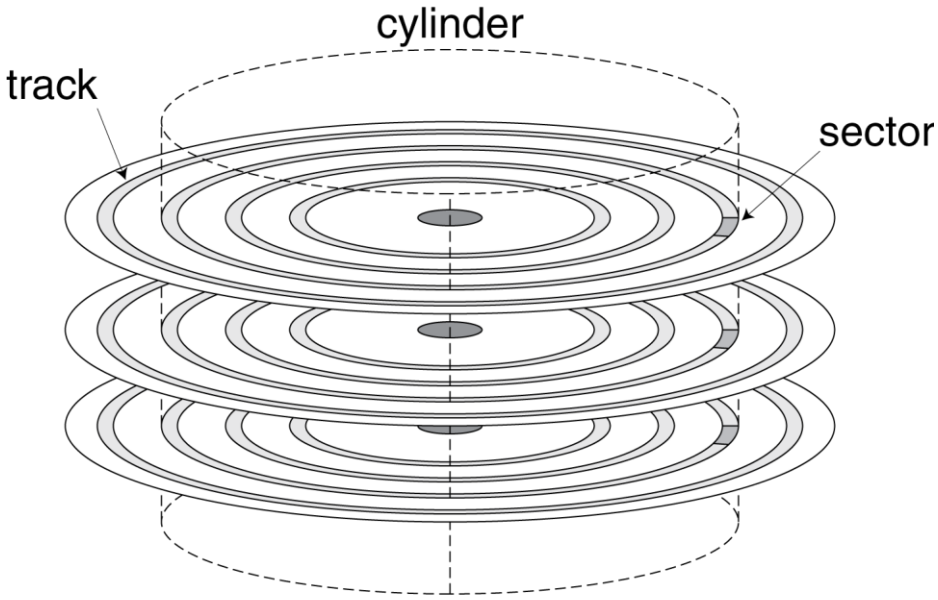
大小：　8 GB
类型：　LPDDR3
速度：　2133 MHz
状态：　好
生产企业：　SK Hynix
部件号：　-
序列号：　-

BANK 2/ChannelB-DIMM0：

大小：　8 GB

磁盘存储

- 非易失，旋转的磁存储



磁盘扇区的访问

- 每个扇区记录了：
  - 扇区的ID
  - 数据 (512 字节, 建议使用4096字节)
  - 纠错码 (ECC)
    - 用来隐藏缺陷和记录错误
  - 扇区间的同步字段和间隙
- 访问一个扇区的时间包括：
  - 排队延迟（如果其他访问正在等待）
  - 寻道: 移动磁头到正确的柱面上
  - 旋转时间：旋转磁盘，以便磁头定位到正确的扇区
  - 数据传输时间
  - 控制器开销

磁盘访问的例子

- 假设
  - 扇区大小512字节, 转速15,000 rpm(圈/分钟) , **平均**寻道时间为4ms, 数据传输速度100MB/s , 控制器开销为0.2ms
- **平均**的读一个扇区的时间
  - 4ms 寻道时间
    + ½ / (15,000/60) = 2ms 旋转时间
    + 512 / 100MB/s = 0.005ms 数据传输时间
    + 0.2ms 控制器开销
    = 6.2ms
- 如果实际的寻道时间为 1ms
  - **平均**的读一个扇区的时间= 3.2ms
- 为什么突发模式比较好?

磁盘性能的问题

- 制造商标注的平均寻道时间
  - 基于所有可能的寻道方式
  - 局部性和OS调度，可以实现更短的实际平均寻道时间

- 磁盘通常包括缓存
  - 支持预取一些扇区，以便后续访问
  - 避免了寻道和旋转时间

Flash存储

- 非易失的半导体存储
  - 比磁盘块100倍 – 1000倍
  - 更小、更低功耗、更鲁棒
  - 但是单价更贵$/GB（介于磁盘与DRAM之间）
  - SSD，SD卡，SIM卡，BIOS系统，U盘

Flash的类型

- NOR flash: bit cell like a NOR gate
  - 支持随机读写
  - 用于嵌入式系统中的指令内存
- NAND flash: bit cell like a NAND gate
  - 存储密度更高，但是每次访问以块为单位
  - 单价更低
  - 用于U盘, media storage, …
- Flash单元在1000次访问之后，会损坏
  - 不适合用直接替换主存，也不适合直接替换磁盘
  - 磨损均衡：将数据重新映射到更少使用的物理块

回顾: 内存技术

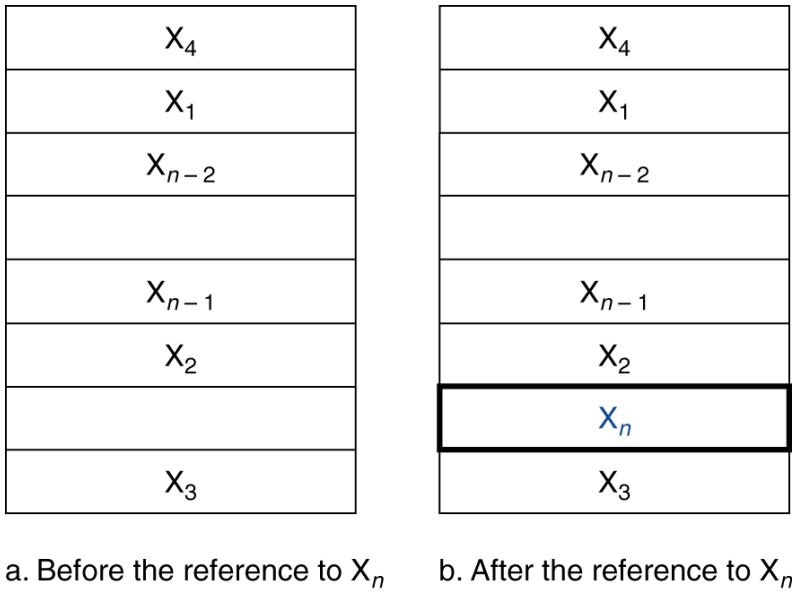
- 理想的内存
  - 访问速度媲美SRAM
  - 存储容量 & 单位价格，媲美磁盘

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

- **内存概述**
- **缓存基础**
- 缓存的性能
- 可靠的内存技术
- 虚拟内存
- 一个通用框架

缓存

- 缓存
  - 内存层次中最靠近CPU
- 假设要访问数据： $X_1, \ldots, X_{n-1}, X_n$

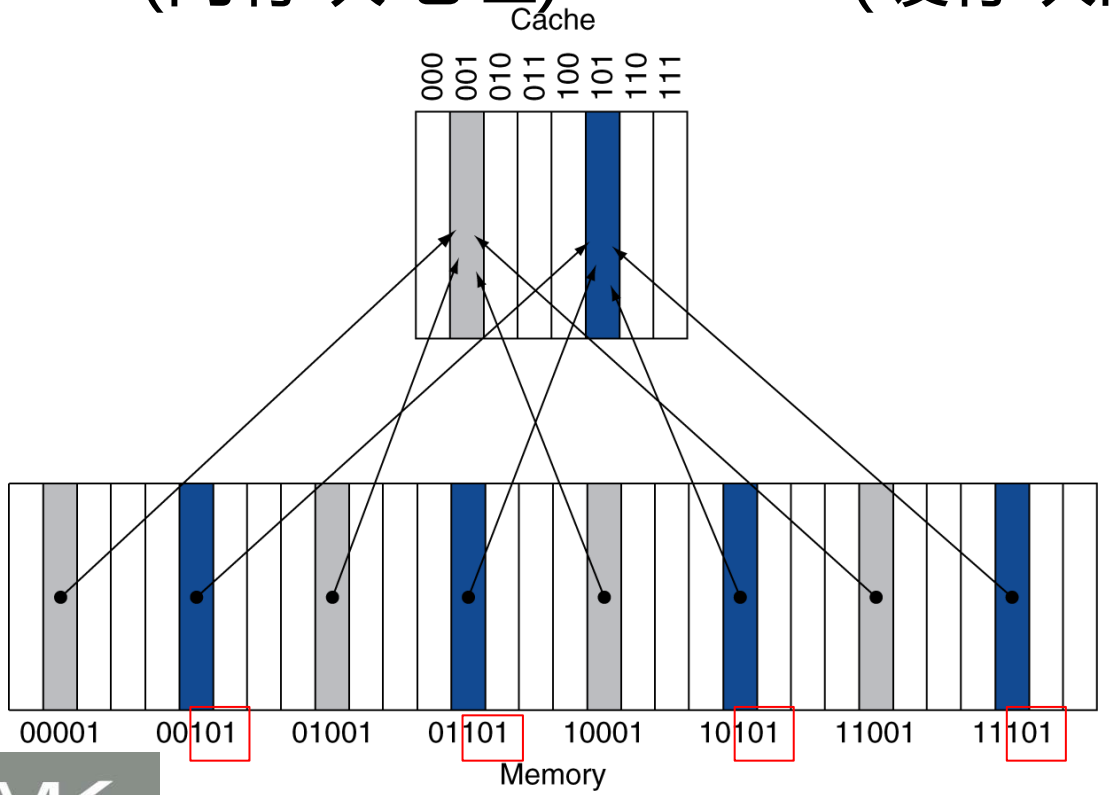


a. Before the reference to X_n　　b. After the reference to X_n

- 如何确定数据是否在缓存?
- 去缓存的哪个位置寻找?

直接映射的缓存

- 根据**内存地址**，直接确定**缓存位置**
- **直接映射**: 唯一映射
  - (内存块地址) modulo (缓存块的数目)



- 缓存块的数目是 2的幂
- 使用块地址的**低位**来映射
  - 多对1映射

标签 & 有效位

- 多对一映射中，如何知道具体哪个内存块存放在缓存中？
  - 缓存中存放数据的同时，也存放**块地址**
  - 但是，只需要存放块地址的**高位部分**，why?
    - 称为：标签（tag）
- What if there is no data in a location?
- 如果对应的缓存块中没有存放有效数据呢？
  - 有效位：1 表示有效, 0 表示无效
  - 初始化值 0

缓存的例子

- 8个块, 每块1个字, 直接映射
- 初始状态

索引不需要存储

索引	有效位	标签	数据
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

缓存的例子

字地址	二进制地址	是否命中	缓存块
22	10 110	未命中	110

索引不需要存储

索引	有效位	标签	数据
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	**10**	Mem[10110]
111	N		

缓存的例子

字地址	二进制地址	是否命中	缓存块
22	10 110	未命中	110
26	11 010	未命中	010

索引	有效位	标签	数据
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

缓存的例子

字地址	二进制地址	是否命中	缓存块
22	10 110	Hit	110
26	11 010	Hit	010

不需要存储

⇩

索引	有效位	标签	数据
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

缓存的例子

字地址	二进制地址	是否命中	缓存块
22	10 110	未命中	110
26	11 010	未命中	010
16	**10 000**	**未命中**	**000**
3	**00 011**	**未命中**	**011**
16	**10 000**	**命中**	**000**

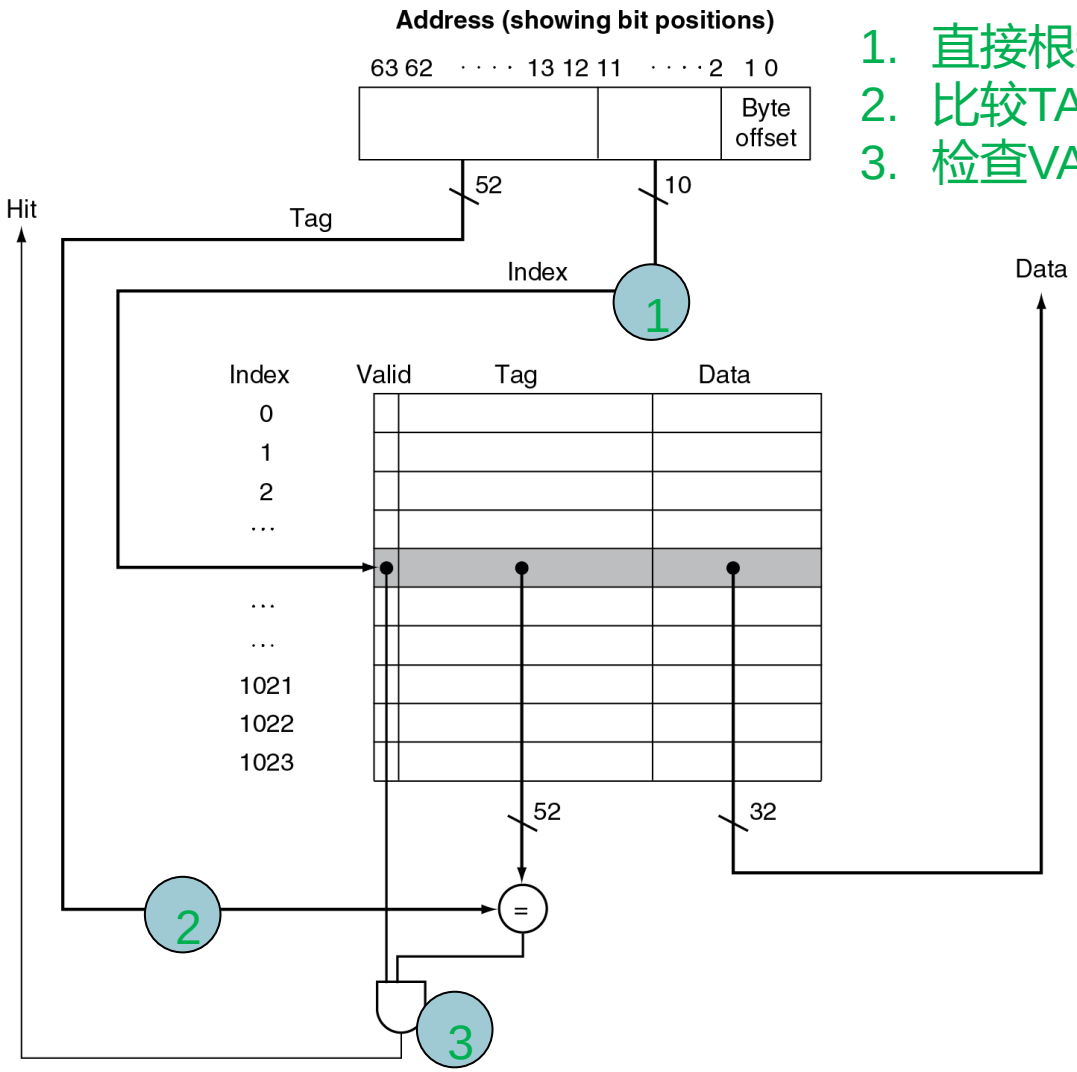
索引	有效位	标签	数据
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

缓存的例子

字地址	二进制地址	是否命中	缓存块
22	10 110	未命中	110
26	11 010	未命中	010
16	10 000	未命中	000
3	00 011	未命中	011
16	10 000	命中	000
18	**10 010**	**未命中**	**010**

索引	有效位	标签	数据
000	Y	10	Mem[10000]
001	N		
010	**Y**	**11 -> 10**	Mem[11010] -> **Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

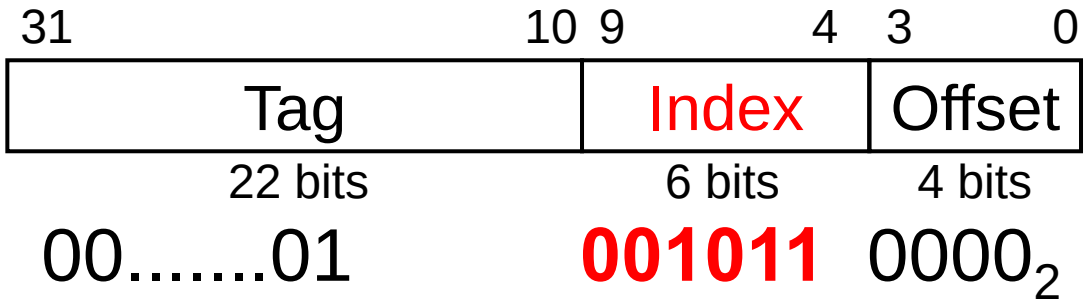
地址的划分



1. 直接根据INDEX找到缓存块
2. 比较TAG是否相等
3. 检查VALID 是否有效

例子: 更大的缓存块

- 64个块, 每块16字节
  - 字节地址1200，会映射到哪个缓存块？
- 内存块地址：⌊1200/16⌋= 75
  - 1200_{10} = 00...01 001011 0000_2
  - 00...01 00 1011_2=$4B_{16}$ = 75
- 映射的缓存块的编号：75 modulo 64 = 11



缓存块大小的影响

- 更大的缓存块可以**提升命中率**
  - 提升了空间局部性
- 但是，固定缓存总容量的话
  - 更大缓存块 ⇒ 更少的缓存块数目
    - 缓存容量 = 缓存块大小 * 缓存块数目
    - 更多的缓存块竞争 ⇒ 降低命中率 -> 不能充分利用时间局部性?
  - 更大的未命中惩罚
    - 加载大的缓存块需要更长时间
  - 更大的缓存块 ⇒ 更容易被污染（写回成本高）

缓存未命中

- 缓存命中时，CPU可以正常往后执行
- 缓存未命中时——类似于可恢复的异常处理
  1. 停顿CPU流水线，然后从下一级缓存加载对应的缓存块
     - 更新当前级缓存
  2. 指令缓存未命中
     - 重新从当前级缓存读取指令
  3. 数据缓存未命中
     - 重新在当前级缓存完成数据操作

练习1

- 一个直接映射的Cache有128个块，主内存有16K个块，每个块有16个字，访问Cache的时间是10 ns，填充一个Cache块的时间是200 ns。Cache初始状态为空。
  - 如果按字寻址，请给出主内存的地址字段格式。
  - CPU从主存中一次读取位置为16～210的字，循环读取10次，访问Cache的命中率多少？
  - 10次循环中，CPU平均每次循环读取的时间是多少？

练习1——缓存hit ratio

- 一个直接映射的Cache有128个块，主内存有16K个块，每个块有16个字，访问Cache的时间是10 ns，填充一个Cache块的时间是200 ns。Cache初始状态为空。
  - 如果按字寻址，请给出主内存的地址字段格式。
  - CPU从主存中一次读取位置为16～210的字，循环读取10次，访问Cache的命中率多少？
  - 10次循环中，CPU平均每次循环读取的时间是多少？
- 分析
  - 内存地址：16K * 16 = 256K，需要18-bit，其中块内地址需要4-bit
  - 开始块号：16/16=1；结束块号：210/16=13。由于直接映射的Cache有128块，所以不会有Cache块之间的竞争；但是每个块第一次访问需要从主内存填充Cache。所以，每次循环
    - 访问次数为：210-15=195
    - 其中，第一次循环Cache失效13次，后面的循环不会失效
    - 命中率为：195*10-13/195*10
    - （1950*10+210*13）/10

缓存写操作——策略1:写直达

- 在**数据写**命中时，直接更新缓存中对应的块
  - 但是缓存和下一级内存的内容会不一致！
- **写直达**策略: 同时更新下级存储
  - 但是写操作需要更长时间
    - 比如，假设基础CPI为1，10%的指令为store指令，写入主存需要100个周期
    - 实际的CPI = 1 + 0.1×100 = 11
- 加速写直达: 写缓冲区 (why does buffer work?)
  - 保存等待写到主存的数据
  - CPU 立即往后执行
    - 只有写缓存区满了，才会阻塞store指令
    - 利用写缓冲区加速写直达策略

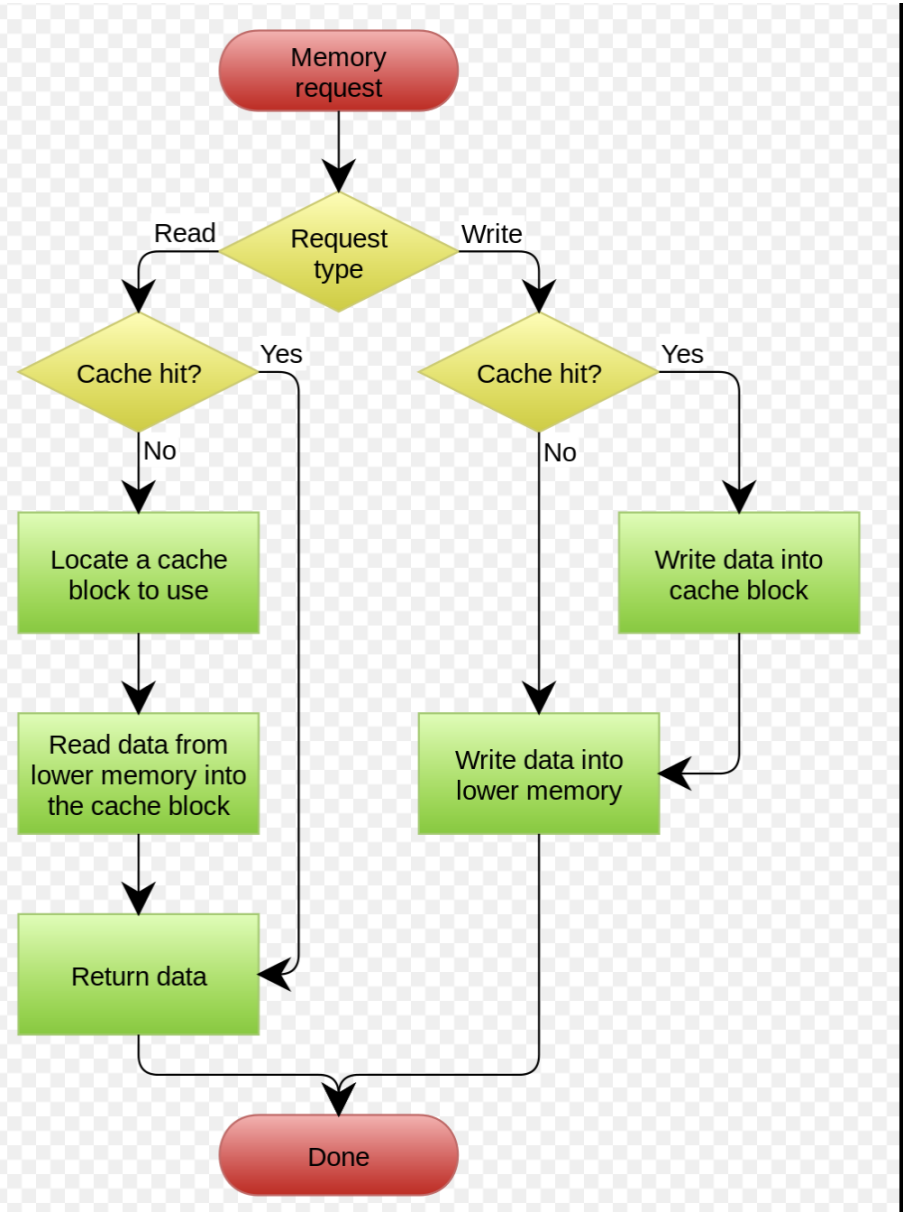
缓存写操作——策略2:写回

- 在数据写命中时，只更新缓存中的块
  - 记录每个缓存块是否dirty（是否被写过）
- 后面当一个脏块被替换时（write+read）
  - 将该块的旧内容写回到主存（write），同时从主存加载新的内存块到缓存
    - 可以利用一个写缓冲区，这样写操作不会阻塞，可以立即开始read操作
    - 利用写缓冲区加速写回策略

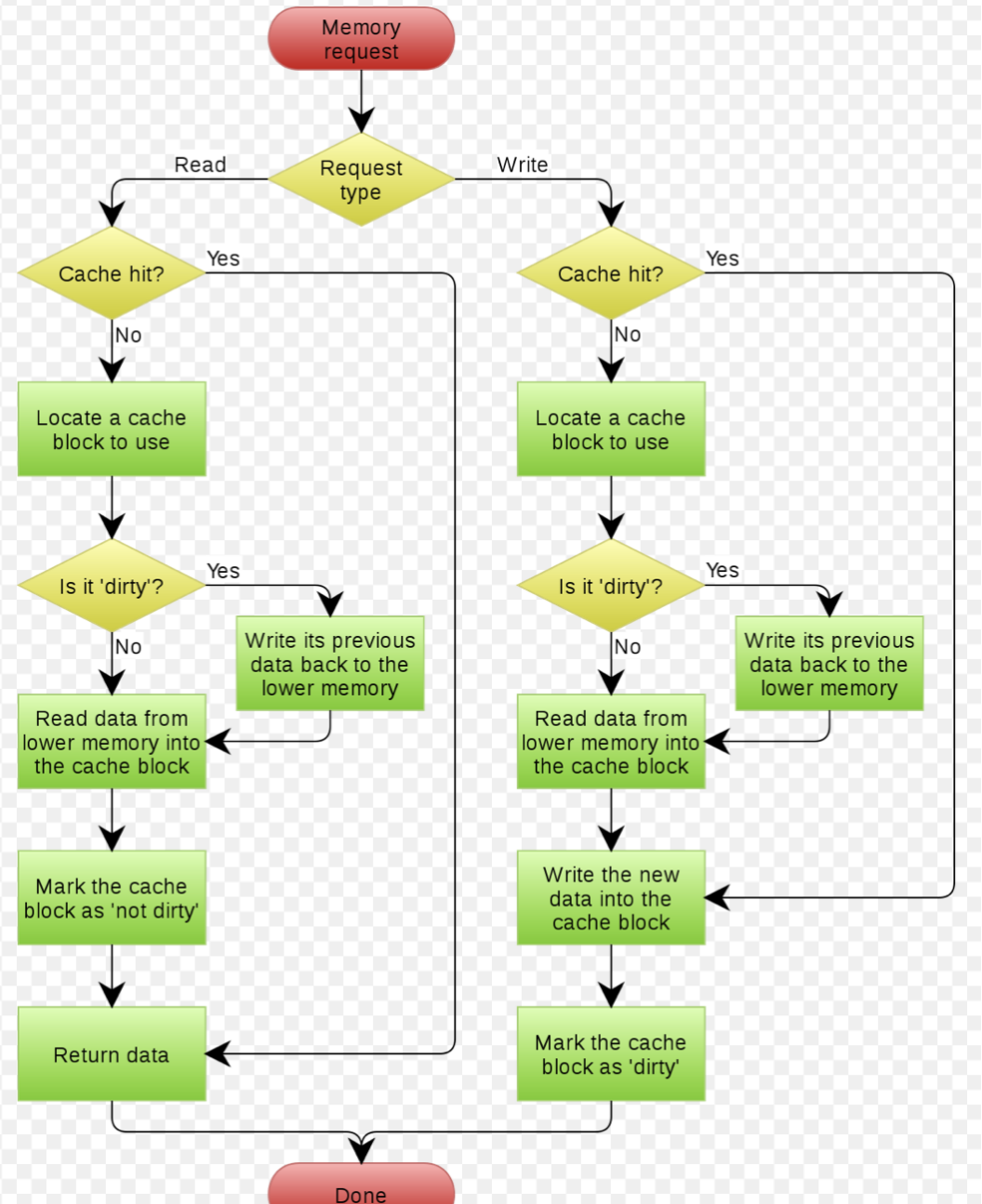
写未命中

- 写操作未命中时，怎么做?
- 写分配：+写回策略
  - 立即从下级存储加载缓存块
- 写不分配：+写直达策略
  - 直接在内存中写，不加载缓存块
    - 因为程序有时写完一个内存块之后**很久**，才去读这个内存块（比如，初始化数据），因此没必要写失效时就加载到缓存

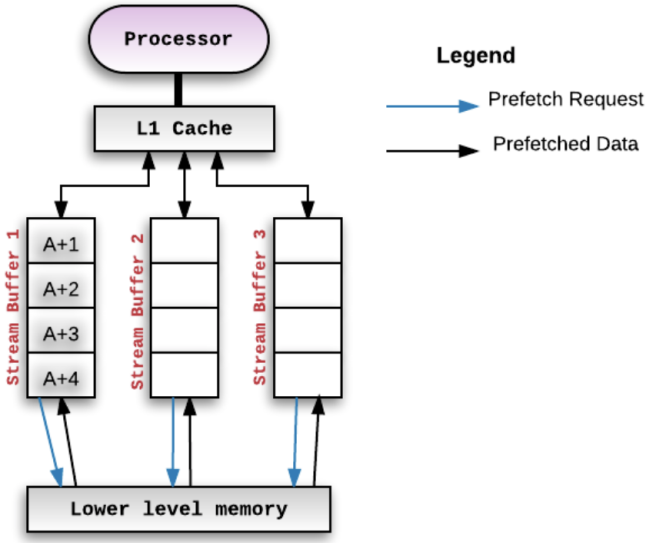
- 写直达
- 写不分配


Memory request
Request type
Read
Write
Cache hit?
Yes
No
Locate a cache block to use
Read data from lower memory into the cache block
Return data
Cache hit?
Yes
No
Write data into cache block
Write data into lower memory
Done

- 写回
- 写分配


Memory request
Request type
Read
Write
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Mark the cache block as 'not dirty'
Return data
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Write the new data into the cache block
Mark the cache block as 'dirty'
Done

- 写缓冲区可以隐藏内存写操作
  怎样的设计可以隐藏内存读操作呢?
  - 预取



[5] A typical stream buffer setup as originally proposed by Norman Jouppi in 1990

```
for (int i=0; i<1024; i++) {
    prefetch (array1 [i + k]);
    array1[i] = 2 * array1[i];
}
```

Bonus: 缓存模拟器

- Intel Pin 工具
  - 缓存模拟器
  - 也可以搜集指令序列.
  - https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html
- https://github.com/NVlabs/NVBit

Bonus: 缓存模拟器

- Valgrind 工具
  - http://csapp.cs.cmu.edu/3e/labs.html

```
$ ../../../pin -t obj-ia32/malloctrace.so -- /bin/cp makefile obj-ia32/malloctrace.so.makefile.copy
$ head malloctrace.out
malloc(0x24d)
  returns 0x6504f8
malloc(0x57)
  returns 0x650748
malloc(0xc)
  returns 0x6507a0
malloc(0x3c0)
  returns 0x6507b0
malloc(0xc)
  returns 0x650b70
```

- *Cache Lab* **[Updated 5/2/16]** (README, Writeup, Release Notes, Self-Study Handout)

  At CMU we use this lab in place of the Performance Lab. Students write a general-purpose cache simulator, and then optimize a small matrix transpose kernel to minimize the number of misses on a simulated cache. This lab uses the Valgrind tool to generate address traces.

  Note: This lab must be run on a 64-bit x86-64 system.

例子: Intrinsity FastMATH

- 嵌入式系统中的MIPS 处理器
  - 12-级流水
  - 每时钟完成指令和数据访问
- 分开的缓存：分开的指令缓存和数据缓存
  - 每个容量为16KB：256个块 * 16个字/每块
  - 数据缓存: 写直达或写回策略（可配置）
- SPEC2000 失效率
  - 指令缓存: 0.4%
  - 数据缓存: 11.4%
  - 加权平均: 3.2%

例子: Intrinsity FastMATH

16KB: 256个块 * 16个字

$2^{14} = 2^8 * 2^4 * 2^2$

1. 直接根据INDEX找到缓存块
2. 比较TAG是否相等
3. 检查VALID 是否有效
4. load对应的单字