计算机组成 PART2

CHAPTER 5: MEMORY HIERARCHY

Memory Hierarchy Introduction

Principle of Locality 局部性原理

一般情况下，程序只会访问地址空间中相对较小的一部分，并且访问的内容具有局部性：不久前刚被访问的和附近被访问过的数据更容易被访问

- **Temporal Locality**：时间局部性，即近期访问的项目很有可能会在短时间内再次被访问。例如循环中的指令、induction variables (循环中用来计数的变量) 等
- **Spatial Locality**：空间局部性，即近期访问项目附近的项目也有可能会在短时间内再次被访问。例如连续的指令执行，或者数组变量等

Memory Hierarchy

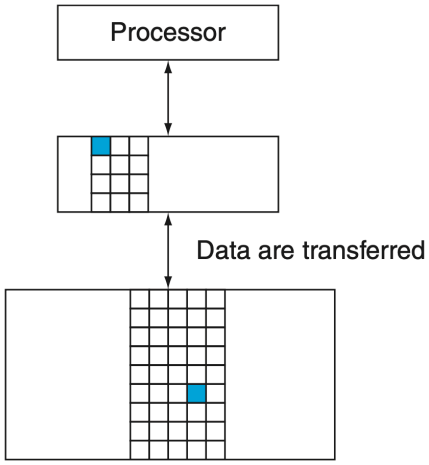
根据上述的局部性原理，我们可以设计层次化存储系统，将近期访问的内存单元及其附近的内容放在更小但速度更快的硬件中。越接近CPU，速度越快，成本越高，容量越小

$$\begin{gathered}\text{Registers} \\ \bigtriangledown \\ \text{L1} - \text{Cache} \\ (\text{On Chip}) \\ \bigtriangledown \\ \text{L2} - \text{Cache (SRAM)} \\ \bigtriangledown \\ \text{Main Memory (DRAM)} \\ \bigtriangledown \\ \text{Disk, Tape, etc.}\end{gathered} \tag{1}$$

一些概念

1. **Block**：Unit of copying，Cache和主存间传输信息的单元，一般为2^nwords
2. **Words**：Cache和处理器间传输信息的单元
3. **Hit**：If accessed data is present in upper level, CPU accesses the upper level and succeeds
    - **Hit Time**：接触上层存储需要的时间，包括hit和miss的判断时间
    - **Hit Ratio**：Hits/Accesses
4. **Miss**：If accessed data is absent, CPU accesses the upper level and fails. The block is needed to be copied from lower level

- **Miss Penalty**：从低层存储复制所需要的block到上层存储的时间 + 将该block传输到处理器的时间
- **Miss Ratio**：Misses/Accesses



存储技术

SRAM

靠近处理器的部分一般采用SRAM，其速度更快，但成本也更高。SRAM通常被分为多个层级的cache，现今的SRAM都已集成入处理器，因此独立的SRAM芯片已经消失

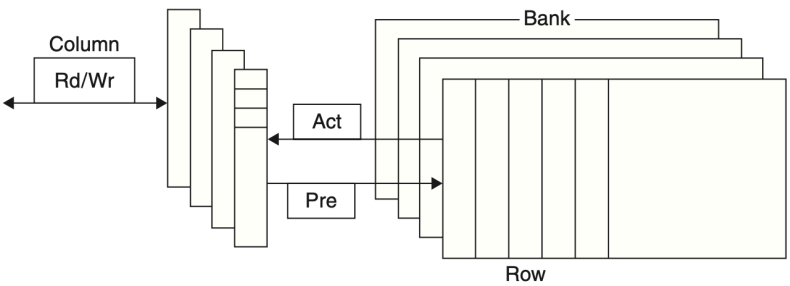
SRAM只有一个端口，要么读要么写。SRAM读写操作需要的时间不同，但对于不同位置的数据访问时间是相同的。其使用多个晶体管（transistor）来存储电荷，实现自锁，因此只需要最低功率就可以确保数据的稳定存储

DRAM

> DRAM和SRAM技术在数逻中涉及较多，计组中应该是不要求的，所以笔记白写了，看看就好

主存由DRAM实现，它用单个晶体管来存储电荷，因此相比DRAM数据密度更高，成本更低。但需要进行周期性刷新（数逻中涉及，每次刷新都将其中整行的数据读出，并再次写回），为了实现这一点，DRAM采用2-level decoding structure，以在刷新存储的同时保证数据的正常访问

地址可以定位到DRAM中的任意一个bit，直到换行。为了解决和处理器时钟冲突的问题，给DRAM添加了时钟，称为Synchronous DRAM（或SDRAM，同步DRAM）。SDRAM进行burst transfer时无需指定额外的地址（简单理解一下，就是只需要给一个起始地址，后面的数据都会通过地址递增的形式被传出）。速度最快的版本是Double Data Rate (DDR) SDRAM，在时钟的上升沿和下降沿都传输数据



Flash & Disk

1. **Flash**：速度介于Disk和DRAM之间，价格也是

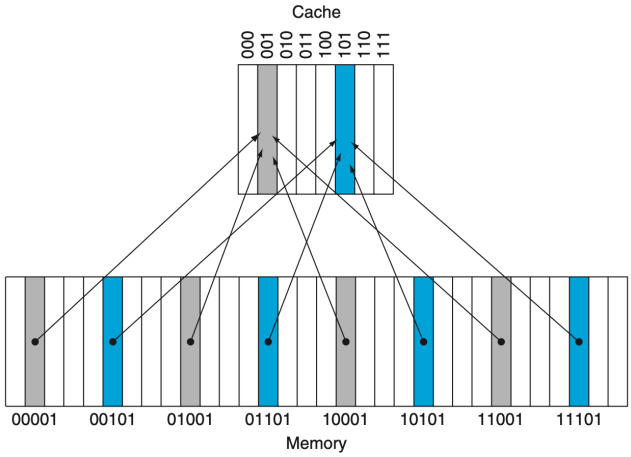
2. **Disk**：量大管饱

Basics of Cache

Direct Mapped Cache

对于直接映射cache，每一个主存地址对应唯一一个cache中的地址。处理器给出一个主存地址，我们将其转化为cache中的地址，并进入cache中寻找。具体的映射方式就是取余，因为cache有2^n个block，所以取余等效于保留后n位

$$(\text{Block address}) \mod (\text{Number of blocks in the cache}) \tag{2}$$



> Consider a cache with 64 blocks and a block size of 16bytes. What block number does byte address 1215 map to?
>
> 因为地址是以byte为单位的，先将其转化为block：$\lfloor 1215/16 \rfloor = 75$（注意除不尽是正常的，因为一个block本身就含有很多byte，并不是每一个byte都是一个block的起始的）
>
> 然后按block数取余：$75 \mod 64 = 11$

1. **主存地址的拆分**：Tag + Index + Offset
    - **Byte Offset**：block的偏移，大小和block的大小相同（以byte为单位，故主存地址定位到的是block中的一个byte）
    - **Index**：用于定位到cache的某个block，因此大小和cache的大小相同
    - **Tag**：主存地址除去Offset和Index剩下的部分，用于和cache中的Tag进行比对。如果一致，说明cache中存储的就是需要寻找的那个block

> 假设我们有一个32bit的address，cache的大小为256block，block的大小为4word。则Offset为4（因为4word相当于是16个byte，需要4位才能表示），Index为8（因为cache一共有256个block，因此需要8位定位到一个具体的block），Tag就是剩下的20位了

Tag	Index	Offset
20	8	4

2. **Cache的结构**：Tag + Data + Valid Bit
    - **Tag**：就是主存地址里面的Tag，计算Index剩下的部分
    - **Data**：存储的内容
    - **Valid Bit**：为0时表示该block为空，数据不在其中，为1时有效

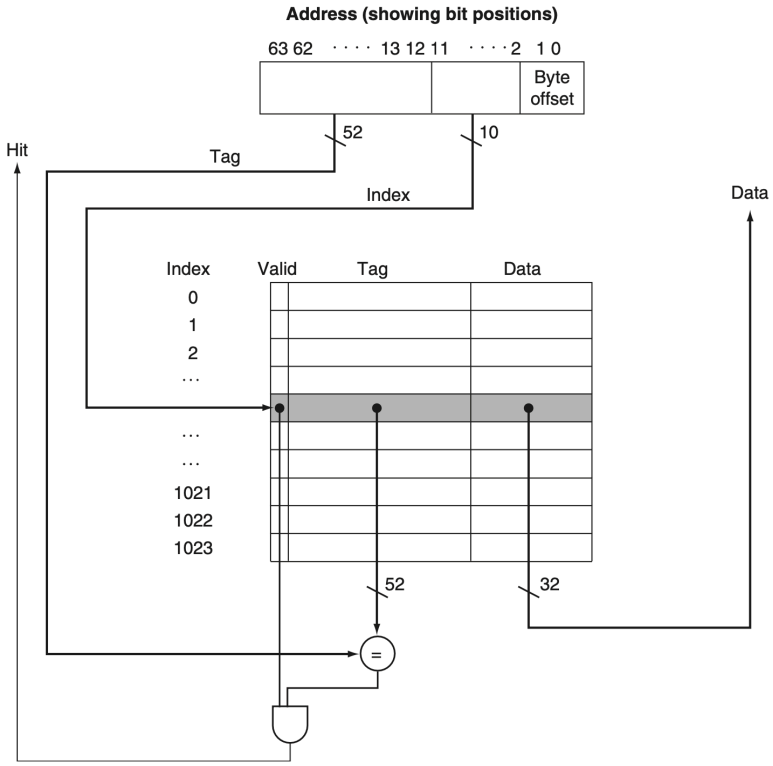
> How many total bits are required for a direct mapped cache 16KB of data and 4word blocks, assuming a 32bit address?
>
> 先计算Tag需要的位数。因为1个block有4个word，因此Offset为4
>
> 因为一共有16KB的数据，每个block存4word，因此一共需要$2^4 \times 2^{10}/16 = 2^{10}$个block，Index需要10位。因此Tag为18位
>
> 最后，一个block存4word的数据，因此Data需要$4 \times 32 = 128\text{bit}$（注意这里是实打实的数据，不是地址）

Valid Bit	Tag	Data
1	18	128

$$\text{Total Bits} = 2^{10} \times (1 + 18 + 128) = 147\text{Kbits} \tag{3}$$

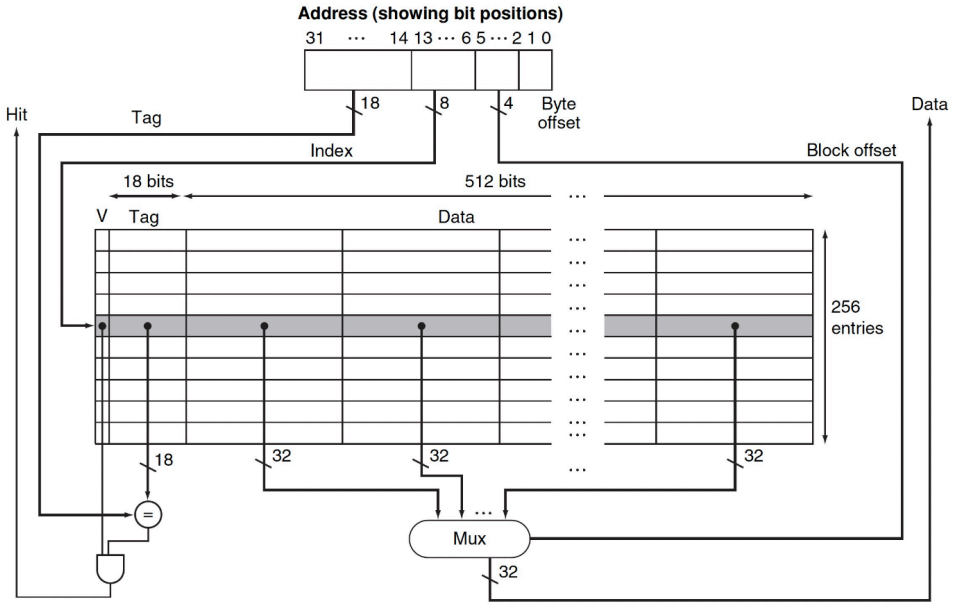
3. **连线方式**

   从主存地址中剥离出Index，在cache中找到对应的block，将其中的Tag与主存地址中的Tag比较，判断是否Hit，并返回Data



   一个block可能含有多个word，但事实上每次访问只会获得一个word，因此我们还需要在block中选择我们需要的那个word

   下图是一个Intrinsity RastMATH处理器的cache，Index有8位，因此共有$2^8 = 256$个cache entry；每个block都有$2^4 = 16$个word，因此在Index的右边有4位Block Offset；剩下2位的Byte Offset用于在word中定位到特定的byte



Write

1. **Write Hits Strategy**
   - **Write Back**：CPU写入数据时只写入cache，主存不更新，仍保留旧数据

     因为不涉及主存，所以速度快。但因为主存中数据未更新，会导致inconsistent，因此cache中除valid位之外还需要1位dirty位。如果cache更改但主存未更改，dirty = 1

     $$\text{CPU} \xrightarrow{\text{写入字}} \text{Cache} \dashrightarrow \text{Main Memory} \tag{4}$$

   - **Write Through**：总是同时更新cache和主存

     保证了consistent，但因为需要访问主存，速度较慢。因此需要增加write buffer，用于暂存需要写入的数据，避免写入的排队等待

     $$\text{CPU} \xrightarrow{\text{写入字}} \text{Cache} \xrightarrow{\text{写入块}} \text{Main Memory} \tag{5}$$

2. **Write Misses**
   - **Write Allocate**：将主存中的整个block拿到cache中，并写入

     如采用write allocate和write back的策略，且dirty = 1，说明即将被覆盖的cache中的数据还未更新入主存中，因此需要先将cache中的数据存入主存，再从主存中读取miss的block入cache，再写入cache

     对于write back策略，通常采取write allocate策略，保证cache中的数据始终是最新的；对于write through，allocate和around都有选择

   - **Write Around**：直接改写主存，不将block拿到cache中

Read

1. **Read Hits**：这就是构建cache的目的，直接读就好

   $$\text{CPU} \xleftarrow{\text{读出字}} \text{Cache} \dashleftarrow \text{Main Memory} \tag{6}$$

2. **Read Misses**：暂停CPU运行，等待主存中的block读入到cache中，然后将其中需要的word给到处理器

   $$\text{CPU} \xleftarrow{\text{读出字}} \text{Cache} \xleftarrow{\text{读出块}} \text{Main Memory} \tag{7}$$

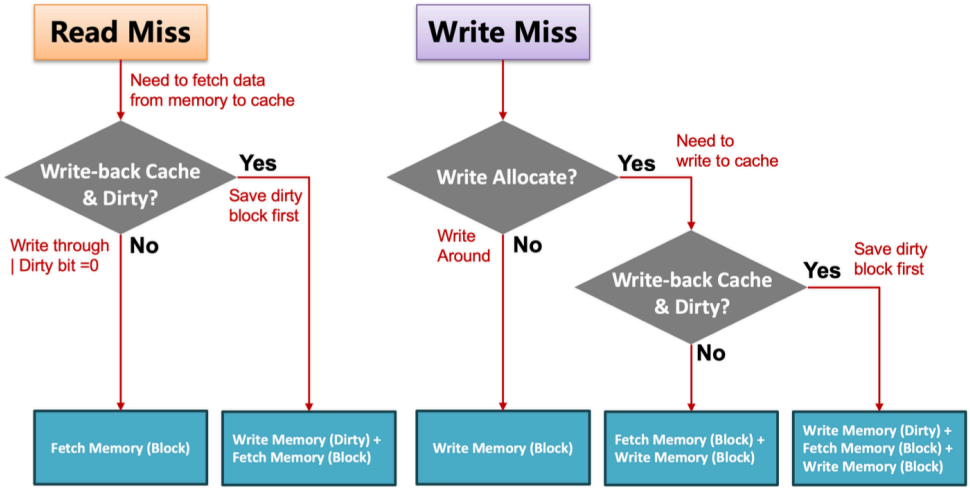
- **Data Cache Misses**：如果是需要用到的数据不在cache中，暂停CPU运行，将memory中的block拿上来，然后让CPU重新开始运行
- **Instruction Cache Misses**：如果发生了意料之外的跳转，可能会出现目标指令不在cache中的情况，这就是instruction cache misses

  先将PC的原始值（当前PC - 4）发送给memory，将主存中的数据读出并写入到cache中，完成后重新取指运行

读写操作总结

如果write miss时write buffer需要的stall很小，可以忽略write memory penalty

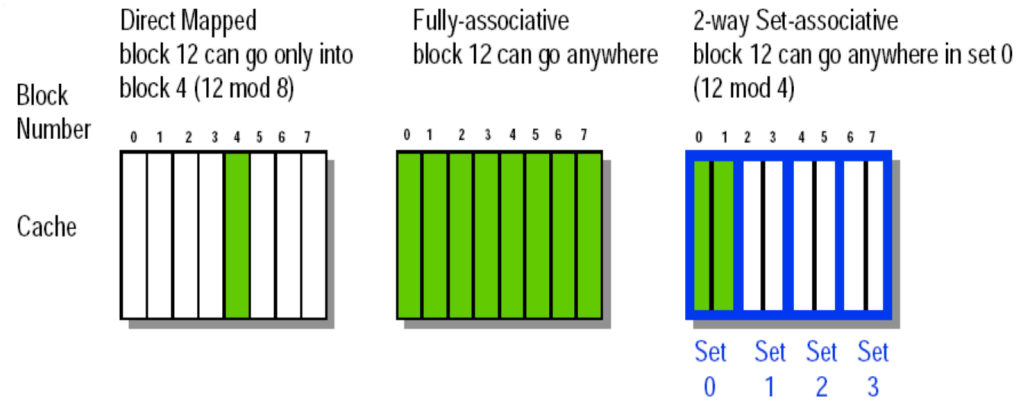
如果block大小为1word，write miss penalty是0，除非dirty = 1。因为CPU给出的这个word就是一个完整的block，直接写入cache中，正好覆盖掉了之前的数据。不需要主存提供额外的block数据（但是如果CPU给出的word少于一个block，则该block中的其他数据只能从主存中获得）



- *If the write buffer stalls are small, we can safely ignore them. (No penalty on Write Memory)*
- *If the cache block size is one word, the write miss penalty is 0. (Except the block is dirty)*

其他映射机制

直接映射虽然思路简单，但因为块冲突率高，导致cache利用率低，适用于大容量的cache。因此我们还有如下2种映射方式



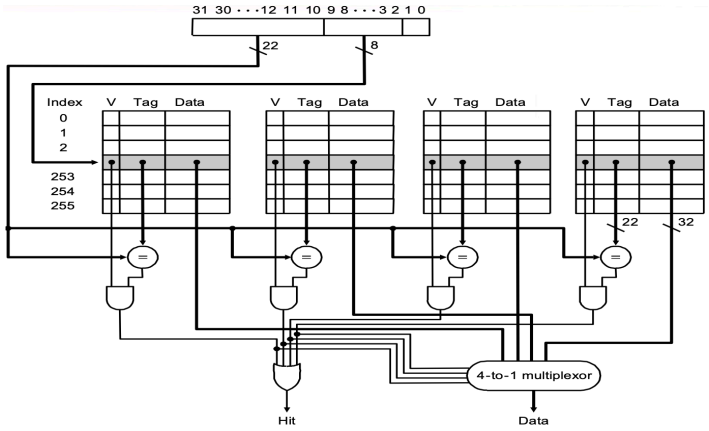
1. **Fully Associative**：Block可以去到cache的任意一个地方

   对于全相连来说，不存在Index，除了Offset剩下的全是Tag了，因为哪里都有可能，只能一个个找

2. **Set Associative**：将cache分为几个set，对于一个k-way set associative的cache来说，每个set中有k个block。每个主存中的block对应一个set，可以进入到该set中的任意位置

我们称1个set中含有的block数为associativity。其实direct mapped就是associativity = 1的特例，而fully associative就是associativity = n的特例

对于set associative而言，Index不再用于定位到block，而是set。因此Index的最大值也就是set的个数。上图中的Index应为2位



Block替换策略

1. **Random Replacement**：随机替换一个
2. **LRU**：Least Recently Used，基于时间局部性。此方法需要使用额外的存储单元，记录上一次访问本block的时间
3. **FIFO**：First in First out

Measuring and Improving Cache Performance

这部分比较无聊，考察点单一，就在这儿放个例题吧

Measuring Cache Performance

In this exercise, we will look at the different ways capacity affects overall performance. In general, cache access time is proportional to capacity. Assume that main memory accesses take 70 ns and that 36% of all instructions access data memory. The following table shows data for L1 caches attached to each of two processors, P1 and P2.

	L1 Size	L1 Miss Rate	L1 Hit Time
P1	2 KiB	8.0%	0.66 ns
P2	4 KiB	6.0%	0.90 ns

1. What is the Average Memory Access Time for P1 and P2 (in cycles)?

> **注意点**
>
> - 不管hit还是miss，hit time都是需要的，因此1不需要乘以系数；
> - $70/0.66$不是整数，要向上取整，因为处理器不可能在半个周期的时候开始下一步

$$
\begin{aligned}
\text{Mem Access}_1 &= \left\lceil \frac{70}{0.90} \right\rceil = 107 \\
\text{Mem Access}_2 &= \left\lceil \frac{70}{0.90} \right\rceil = 78 \\
\overline{t}_1 &= 1 + 8\% \times 107 = 9.56 \\
\overline{t}_2 &= 1 + 6\% \times 78 = 5.68
\end{aligned} \tag{8}
$$

> 如果想要提升cache性能，只提高CPI without any memory stalls（也就是那个常数项）是不够的

2. Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 and P2? Which processor is faster? (When we say a "base CPI of 1.0", we mean that instructions complete in one cycle, unless either the instruction access or the data access causes a cache miss.)

> 所有指令都需要IF，其中有8%会miss；此外，还有36%的指令需要进行data access，这36%中又有8%会miss
>
> $$\begin{aligned}\mathrm{CPI}_1 &= 1 + 8\% \times 107 + 36\% \times 8\% \times 107 = 12.64\\ t_1 &= \mathrm{CPI}_1 \cdot \text{Clock Time}_1 = 8.34\text{ns}\\ \mathrm{CPI}_2 &= 1 + 6\% \times 78 + 36\% \times 6\% \times 78 = 7.36\\ t_2 &= \mathrm{CPI}_2 \cdot \text{Clock Time}_2 = 6.63\text{ns}\end{aligned} \tag{9}$$
>
> The second processor is faster.

For the next three problems, we will consider the addition of an L2 cache to P1 (to presumably make up for its limited L1 cache capacity). Use the L1 cache capacities and hit times from the previous table when solving these problems. The L2 miss rate indicated is its local miss rate.

L2 Size	L2 Miss Rate	L2 Hit Time
1MiB	5%	5.65ns

4. What is the AMAT for P1 with the addition of an L2 cache? Is the AMAT better or worse with the L2 cache?

> 这里L2 hit time也是始终需要的，只要L1没命中就需要这个时间，因此括号里也不需要乘以系数
>
> $$\begin{aligned}\mathrm{AMAT} &= 1 + 8\% \times \left(\left\lceil \frac{5.65}{0.66} \right\rceil + 5\% \times 107\right)\\ &= 1.292\end{aligned} \tag{10}$$

5. Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 with the addition of an L2 cache?

> $$\begin{aligned}\mathrm{CPI} &= 1 + 8\% \times (9 + 5\% \times 107) + 36\% \times 8\% \times (9 + 5\% \times 107)\\ &= 2.56\end{aligned} \tag{11}$$

6. What would the L2 miss rate need to be in order for P1 with an L2 cache to be faster than P1 without an L2 cache?

> $$\begin{aligned}9 + \text{Miss Rate} \times 107 &\le 107\\ \text{Miss Rate} &\le 91.59\%\end{aligned} \tag{12}$$

7. What would the L2 miss rate need to be in order for P1 with an L2 cache to be faster than P2 without an L2 cache?

> 别忘了两个处理器的时钟周期是不一样的
>
> $$\begin{aligned}[1 + 8\% \times (9 + 107 \cdot \text{Miss Rate}) + 36\% \times 8\% \times (9 + 107 \cdot \text{Miss Rate})] \times 0.66 &\le 6.62832\\ \text{Miss Rate} &\le 69.27\%\end{aligned} \tag{13}$$

Improving Cache Performance

1. **提升处理器速度**

- Instruction cache miss rate = 2%
- Data cache miss rate = 4%
- CPI without any memory stalls = 2
- Miss penalty = 100 cycles
- The frequency of all loads and stores in gcc = 36%

1. How faster a processor would run with a perfect cache?

$$\begin{aligned}\text{CPI} &= \text{CPI with Perfect Cache} + \text{Instruction Miss Cycle} + \text{Data Miss Cycles} \\ &= 2 + 2\% \times 100 + 36\% \times 4\% \times 100 = 5.44 \\ \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} &= \frac{5.44}{2} = 2.72\end{aligned} \tag{14}$$

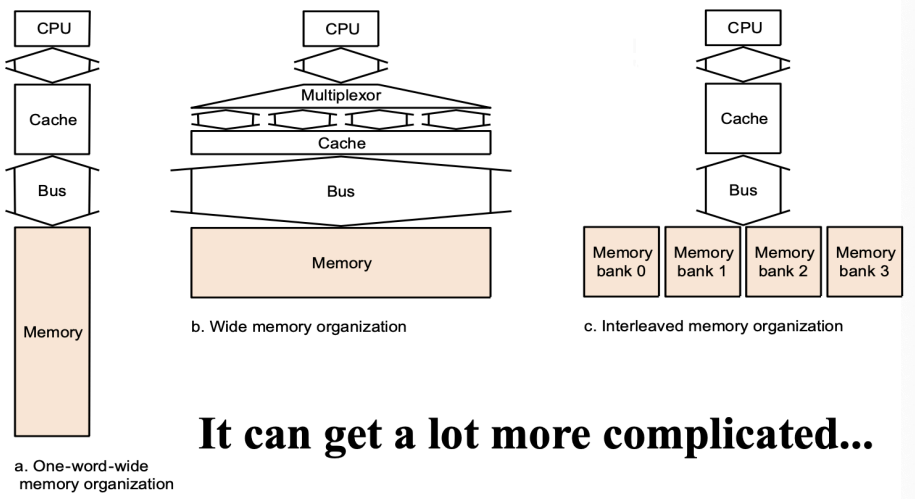
2. What if the processor is made 2 times faster?

> 注意，这里提到的是处理器变快了，而不是memory变快了，也不是时钟变快了，因此事实上stall的周期是一样的，只有$\text{CPI}_{\text{perfect}}$从2来到了1
>
> $$\text{CPI}_{\text{perfect}} = 1 + 3.44 = 4.44 \tag{15}$$

3. How much faster will the computer be with a 2-times faster clock rate?

> 虽然时钟周期变快了一倍，但注意miss penalty的时间是不变的（而非周期不变），因此miss penalty变为200 cycles；同时，CPI不变，因为执行同一条指令所需要的周期不变（时间变快，但周期相同）
>
> $$\begin{aligned}\text{CPI} &= 2 + 2\% \times 200 + 36\% \times 4\% \times 200 \\ &= 8.88\end{aligned} \tag{16}$$

2. **加快读写速度**（降低missing penalty）



在下面的分析中，我们假设：传输地址需要1个周期，启动DRAM需要15个周期（15 memory bus clock cycles for each DRAM access initiated），传输一个word的数据需要1个周期；每个block含有4个word

Bandwidth：每个周期可以传输的byte数

- **One-word-wide Memory Organization**

> 每次只能传1个word，bus需要启动并传输4次，因此传输1个block的数据需要周期为：
>
> $$1 + 4 \times (15 + 1) = 65 \tag{17}$$

> Bandwidth大小为：
>
> $$\frac{4\times 4}{65}\approx\frac{1}{4} \tag{18}$$

- **Wide Memory Organization**

> 假设主存的宽度为4w，那只需要一次传输就可以拿出整个block，因此需要的周期为：
>
> $$1+1\times(1+15)=17 \tag{19}$$

- **Bank**：并行存储

> 假设有4个bank，CRAM只需要启动1次，但数据还是要传4次：
>
> $$1+15+4\times 1=20 \tag{20}$$

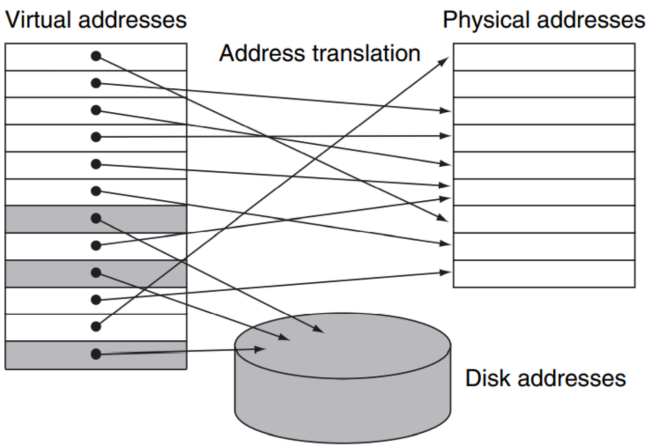
3. **增加存储层次**

增加二级cache也可以提升性能。假设有一个CPI为1.0、时钟频率为5GHz、miss rate为2%、DRAM访问时间为100ns；假设我们增加一个二级cache，访问时间为5ns，并将miss rate降为0.5%

> 先计算miss penalty
>
> $$\text{Miss Penalty}=\frac{100\times 10^{-9}}{1/(5\times 10^{9})}=500 \tag{21}$$
>
> 所以单层cache的总CPI为：
>
> $$\text{Total CPI}_1=1+2\%\times 500=11 \tag{22}$$
>
> 如果一级miss了，则访问二级，所需要的时间为：
>
> $$\text{Miss Penalty without Accessing Mem}=\frac{5\times 10^{-9}}{1/(5\times 10^{9})}=25 \tag{23}$$
>
> 因此，二级cache的总CPI为：
>
> $$\text{Total CPI}_2=1+25\times 2\%+500\times 0.5\%=4 \tag{24}$$

Virtual Memory

因为不同的进程可能会用到同一个物理地址，为了让多个程序可以高效、安全地共享内存，并且允许单个程序使用超过内存容量的内存（超过的部分实际上并不在内存中，在disk中）。因此，引入虚拟内存技术。给出一个虚拟地址，通过page table映射到其物理地址，进入DRAM（主存）查找；如果没找到（page fault），就进入disk查找



- **Physical Address**：主存中使用
- **Virtual Address**：程序内部使用，是一个软件层面的技术
- **Page Table**：Virtual page number到Physical page number的映射表，按虚拟地址排列，存储main memory中

虚拟到物理地址的映射

在disk与main memory中，地址传输是以page为单位进行的（地址依旧定位到一个byte，因此计算Offset还是按照byte为基本单位）。不妨假设page size为4KiB，则Page Offset有12位，剩下的就是Virtual page number

根据Virtual page number进入page table查找Physical page number，然后把Physical page number和Offset拼起来就是physical address了

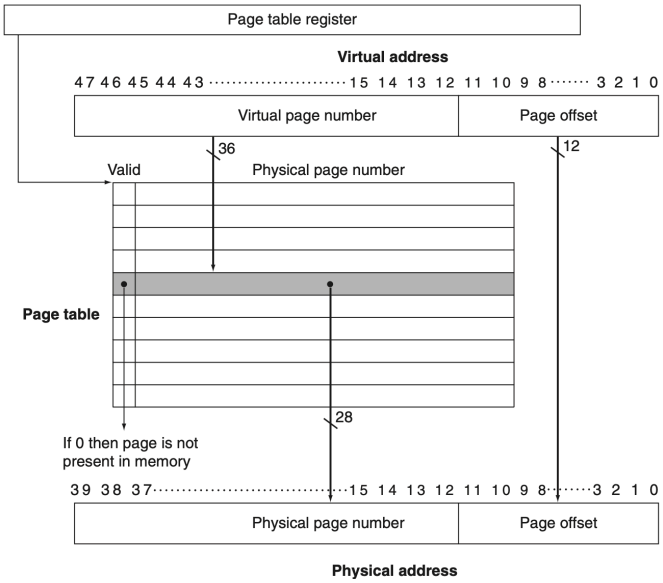
- **Fully-associative**：因为page fault需要去磁盘中查找，开销非常大，因此需要尽可能提高命中率。主存没有分组，因此没有Index，除了Offset就是Tag（page number）
- **Write Back**：write througe策略需要频繁访问disk，过于昂贵，因此采用write back；page fault时仅写入主存，不写入disk

Page Table

页表用于实现虚拟页数到物理页数的映射，存储在主存中，每个进程都有一个独立的page table。Page table可以理解为一个数组，`page_table[i]`就代表第i个virtual page对应的physical page number

Page table的每一行就是一个page table entry。如果Valid位为0，说明该虚拟地址对应的数据不在主存中，发生page fault（此时可以利用本来存储physical page number的字段映射到对应的disk位置，如上图所示；或者也可以采用其他索引结构映射到disk）

因为采用的是write back策略，所以通常来说还需要一位Dirty位



假设虚拟地址是32位的，page size为4KB，entry size为4B

> 因为page size为4KB，所以Offset一共是12位，则剩下的20位为Index，因此一共有2^{20}个entry
>
> 因为每个entry大小为4B，因此page table的大小为$4\mathrm{B} \times 2^{20} = 4\mathrm{MB}$

Page Fault

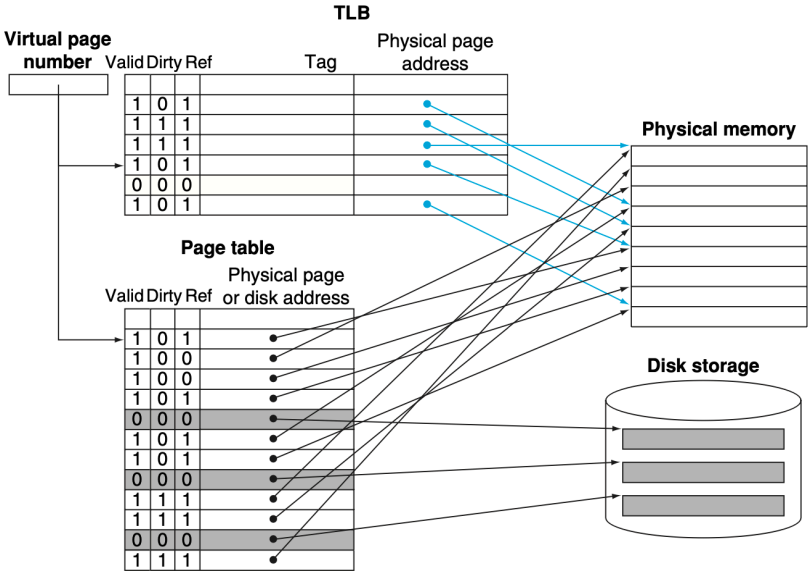
当操作系统新建一个进程时，其将会在disk中创建一个空间（swap space），大小和虚拟地址一致。当发生page fault时，引发exception，操作系统接管，从disk中找到对应的page，放到主存中（如果主存已满，可能会引发替换，按照LRU原则），然后更新page table

Translation Look-aside Buffer：缩短映射时间

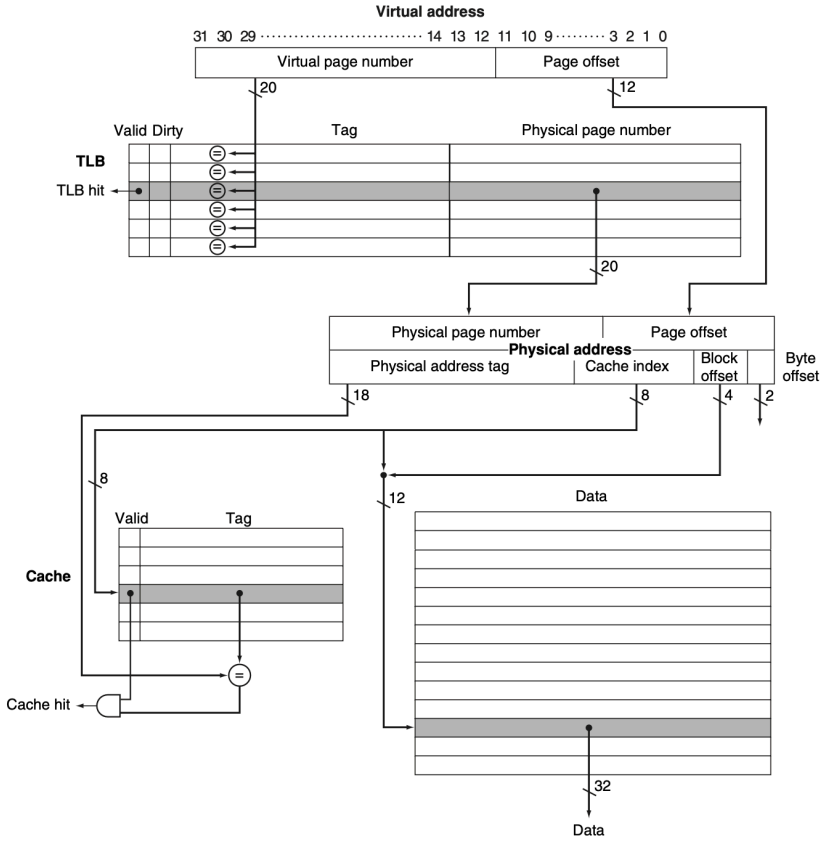
按照之前的逻辑，我们访问一个字节，需要先访问page table以将虚拟地址转化为物理地址，涉及一次内存访问；然后我们要根据求得的物理地址访问该字节，又涉及一次内存访问。也就是说，读取一个字节需要访问两次内存，开销较大

因此，我们为page table设计一个cache，也就是TLB。TLB一般存储在处理器内部，访问速度很快。和cache一样，其associativity可以视具体需要决定

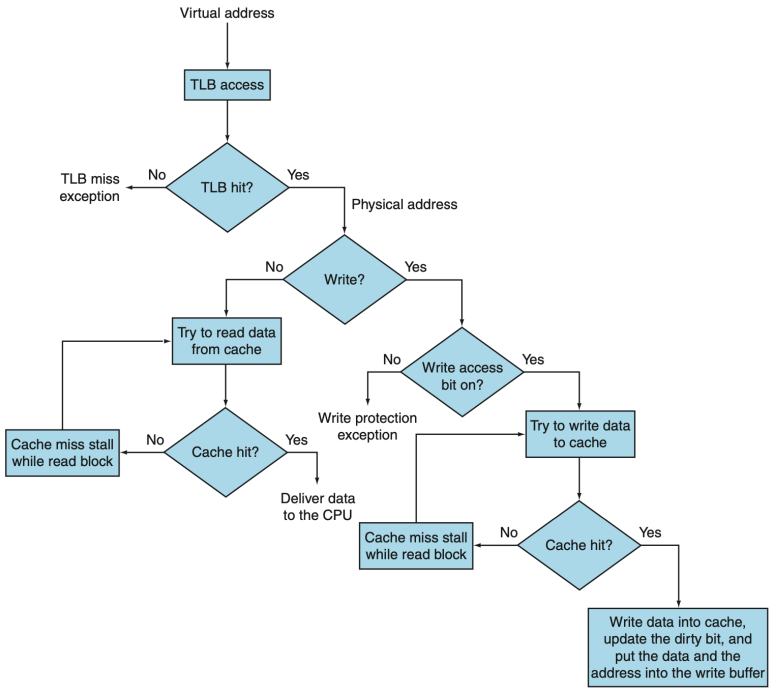
1. **TLB Miss**：给出一个虚拟地址，访问TLB，如果没有命中，就去page table中查找。如果page table有效，则将其拿到TLB中（此时如果替换掉的TLB entry是dirty的，需要先写回page table再覆盖）。下面是一个全相连的TLB



2. **TLB Hit**：如果TLB命中，直接获得物理地址。为了进一步缩短时间，在cache中进行读写，而不是进入主存进行查找。在cache中hit和miss的操作和之前内容一致
    - **Read**：尝试从cache中读取
    - **Write**：先检测write access bit，如果允许写，就进入cache尝试写入



TLB的操作总结如下。不难发现，TLB实质上就是page table的cache（与其说"像是"，不如说"就是"）



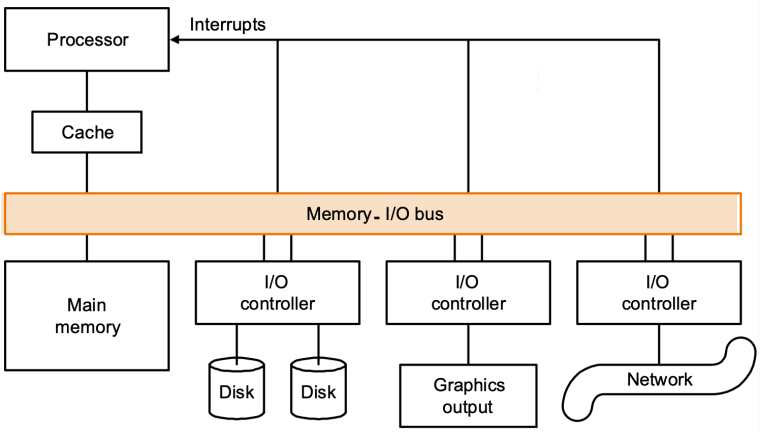
值得注意的是，如果TLB hit，则page table一定hit；如果TLB和page table均miss，则cache也一定miss，因为page table都miss了，只能去disk中寻找，自然不会出现在cache中

CHAPTER 6：I/O

IO设备的设计需要考虑很多因素，比如说可拓展性（expandability）、可还原性（resilience）以及性能（performance）

评价IO设备的性能非常复杂，在不同的场景下往往采用不同的评价方式。一个IO设备的性能和设备与系统间的连接、存储层次以及操作系统都有关系

主存和IO设备与处理器都是通过memory-I/O bus进行通讯的。示意图如下：



I/O Introduction

3 Characters of I/O Devices

1. **Behavior**：IO设备支持的行为（只能读、只能写，或者存储）

   Input只能读一次（read once。比如，键盘只能一次读入，无法复现之前写入的内容），output只支持写出，而storage可以重复读写（reread and usually rewritten）

2. **Partner**：IO设备另一端是面向机器（feeding data）还是面向人类（input or reading data on output，人类需要对着输出设备进行输入，比如说要看着屏幕打字，但机器不用）

3. **Data Rate**：IO设备和主存间传输数据的峰值功率

Performance

衡量IO设备的性能有很多维度，在不同的应用场景中需要选取不同的指标

- **Throughput**：吞吐量。在这样的情况下，IO设备的bandwidth是最重要的。带宽可以有两种衡量方式：一是测量单位时间内IO设备可以传输的数据（一次传输大量数据），二是单位时间内IO设备可以进行的操作次数（频繁传输小规模数据）
- **Response Time**
- **Both**

Amdahl's Law

系统性能提升的幅度是由无法提升的部分决定的，因此在提升系统性能的时候不能忽略IO的开销。因此，虽然IO经常被忽略，但实际上也是非常重要的

将I/O设备Interface到存储、处理器和操作系统中

1. **Memory-mapped I/O**：内存映射IO的指令。一部分的memory address被分配到IO设备，`lw`和`sw`指令可以用于IO设备的读写（还有其他IO指令，但不需要知道）
2. **Communication with Processor**：IO设备和处理器的3种交流方式
   - **Polling**：处理器每隔一段时间就检查一下是否可以进行下一个IO操作。这样的交流方式处理器会被硬控，因为只要IO没做完它就要一直检查

> 假设一次polling需要400个时钟周期（注意fraction指的是"占比"，如果题目中要求计算别漏了）
>
> 1. The mouse must be polled 30 times per second to ensure that we do not miss any movement made by the user.
>
> $$30 \times 400 = 12000\text{cycle/s} \tag{25}$$
>
> 2. The floppy disk transfers data to the processor in 16-bit units and has a data rate of 50KB/sec. No data transfer can be missed.
>
> $$\frac{50\text{KB/s}}{2\text{B}} \times 400 = 1 \times 10^7\text{cycle/s} \tag{26}$$

- **Interrupt**：当某个IO完成了某操作或发生异常需要处理器关注时，给处理器发送中断信号

> 我们将上面的例子换成interrupt策略。假设访问软盘的时间是5%，先算出100%访问所需要的开销（则，interrupt的次数和polling次数相同），然后乘以5%即可

- **DMA**：Direct Memory Access，设备直接和memory进行数据读写，绕开处理器

> 将上面的例子换为DMA策略。DMA需要setup，假设setup需要1000周期，DMA处理完毕后处理器需要500周期来interrupt，硬盘传输速率为4MB/s，average transfer from disk is 8KB（也就是说平均每次传8KB的数据），100%的时间都在传输
>
> 每秒传输的次数为：
>
> $$\frac{4\text{MB}}{8\text{KB}} = 500\text{s}^{-1} \tag{27}$$
>
> 每次消耗的周期为：
>
> $$1000 + 500 = 1500\text{cycles} \tag{28}$$
>
> 每秒消耗的时钟周期就是上面两个乘起来

设计IO系统

给一道例题：A CPU sustains 3 billion instructions per second and it takes average 100,000 instructions in the operating system per I/O operation.（CPU支持3亿条指令，平均每个IO操作需要10,000条指令） A memory backplane bus is capable of sustaining a transfer rate of 1000MB/sec.（总线每秒1000MB）SCSI-Ultra320 controllers with a transfer rate of 320MB/sec and accommodating up to 7 disks.（最多支持7个磁盘，传输速度为329MB/s） Disk drives with a read/write bandwidth of 75MB/sec and an average seek plus rotational latency of 6ms.（磁盘平均读写速度为75MB/s，平均seek和旋转延迟时间为6ms）

If the workload consists of 64-KB reads (assuming the the data block is sequential on a track), and the user program need 200,000 instructions per I/O operation, please find the maximum sustainable I/O rate and the number of disks and SCSI controllers required.

> 对于CPU而言，最大的IO速度为指令执行速率与每个IO操作需要的指令数之比，而每个IO操作都含有200,000个用户指令和100,000个OS指令：
>
> $$\frac{3 \times 10^9}{(100 + 200) \times 10^3} = 10000\text{操作数/秒} \tag{29}$$
>
> 对于总线而言，每次IO读取64KB的数据，而总线每秒可以传输1000MB数据

$$\frac{1000 \times 10^6}{64 \times 10^3} = 14525\text{操作数/秒} \tag{30}$$

因此系统每秒能执行的IO操作数为10,000个。

然后我们计算需要多少个磁盘才能满足每秒10,000次IO。我们先计算磁盘每次IO的时间：

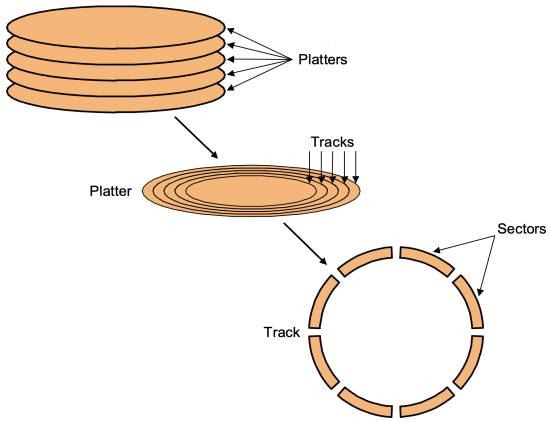
$$6 + \frac{64 \times 10^3}{75 \times 10^6} = 6.9\text{ms} \tag{31}$$

也就是说每个硬盘可以满足每秒进行$1000/6.9 = 146$次IO，为了实现10,000次IO，需要$10000/146 = 69$个硬盘。因为每个controller能塞7个，因此需要10个controller

Disk

Introduction

1. **磁盘的种类**
    - **Floppy Disk**：软盘。容量小，速度慢
    - **Hard Disk**：硬盘。容量更大，存储密度更高，速度更快
2. **Organization of Hard Disk**



    - **Platters**：每个磁盘都含有多个光盘，光盘的两面均可存储
    - **Tracks**：每个光盘上都有很多同心圆，也就是轨道
    - **Sectors**：轨道被分为很多扇区，扇区是硬盘上最小的可读写单元
3. **Access Data of Disk**：想要从硬盘中读出数据，首先需要找到对应的platter，将读写头移动到对应的track上，然后旋转磁盘到指定的sector
    - **Seek**：将read/write head放到正确的轨道上（有minimum seek time、maximum seek time、average seek time）
    - **Rotational Latency**：将磁盘旋转到对应的sector需要的时间（注意磁盘只能一个方向旋转，因此平均的rotational latency就是磁盘旋转半圈的时间）

        > 假设磁盘的转速为5400RPM（Rotation per Minute），则average rotational time为$0.5/5400 \times 60\text{s} = 5.6\text{ms}$

    - **Transfer**：移动到指定的sector后，将整个sector读出。Transfer time取决于sector的大小以及transfer rate

        > 假设sector的大小为0.5KB/sector，transfer rate为50MB/sec，则transfer time（或disk read time）为$0.5/(50 \times 10^3) = 0.01\text{ms}$（注意一个是KB，一个是MB）

- **Disk Controller**：控制disk和memory间的传输

Disk access time就是上面四个时间的总和

Flash Storage

闪存比disk成本更高，但速度也更快。同时，闪存的寿命也更短，经历有限次数读写后可能会损坏（wear out），因此无法用作RAM或disk的完全替代品

闪存有如下两个种类：

- **NOR**：随机读写访问（random read/write access），在instruction memory和植入式系统中使用
- **NAND**：密度更大、成本更低，但逐块访问（block-at-a-time access），在U盘、media storage（存储卡、固态硬盘等）中使用

Disk Performance Issues 影响disk性能的因素

- **Average Seek Time**：纸面上的平均查找时间是基于理论计算的平均值，但locality & OS scheduling可以让查询变得更聪明
- **Smart Disk Controller**：在disk上合理分配sector
- **在disk上增加cache**：减少seek和rotational的延时

Dependability, Reliability and Availability

> MTTF、MTTR是期末考试第六章的重点之一

Dependability：可依赖性。指的是计算机系统的服务质量达到有理由信赖的水平（The quality of delivered service that reliance can justifiably be placed on this service. 按理想情况完成称为service accomplishment）。每个模块都有其指定的理想行为，当实际行为与理想行为发生偏离时，系统发生错误（A system failure occurs when the actual behavior deviates from the specified behavior, service interruption）

对于dependability和reliability有如下这些衡量标准：

- **MTTF**：Mean Time to Failure，平均无故障时间。提升MTTF有三种方式：
  1. **Fault Avoidance**：通过construction预防错误
  2. **Fault Tolerance**：通过redundancy以保证即便错误发生最终也可以给出符合理想行为的结果（主要针对硬件错误）
  3. **Fault Forecasting**：预测错误（软硬件实现都有）
- **MTTR**：Mean Time to Repair，平均修复时间
- **MTBF**：Mean Time between Failure，平均故障间隔时间
- **Availability**：MTTF/MTBF。即，系统中有多少时间比例是在无故障运行的

廉价磁盘冗余阵列：RAID

> 这也是期末考第六章的重点之一，不过总体来说第六章就没什么重要的

Introduction

传统的磁盘如果想要增加容量，只能做一整块更大的磁盘；成本高，只能顺序访问，且如果其中的一部分出现损坏，整块磁盘都会失效。那能不能将多块小磁盘拼在一起，做成一个磁盘阵列呢？

这就是redundant arrays of disks，其成本更低，可以实现并行访问，并且具有更高的冗余性

但是这样做也有缺点，会降低disk的reliability，N块硬盘中的任意一块出现故障的概率显然比一块硬盘出现故障的概率要高。定义（不要试图从概率论的角度理解这个计算，不然你会发现这个式子是错误的，我们就简单地认为MTTF变为原来的$1/N$）：

$$\text{reliability of } N \text{ disks} = \frac{\text{reliability of 1 disk}}{N} \tag{32}$$

这也就是为什么我们提出了冗余磁盘阵列（redundant arrays of disk）

RAID的不同种类

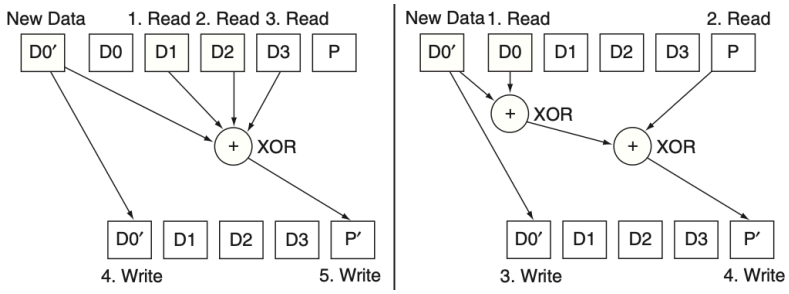
同一个文件存储在不同的小磁盘中（而不是在同一个磁盘中顺序存储），同时，增加冗余空间以提升availability（需要占用一定的额外空间，有capacility penalty；同时，为了更新redundant info，会产生一定的bandwidth penalty）。当然，RAID也还是有可能fail的

在下表中，我们给出几个不同的RAID冗余存储方式，假设我们一共有8个盘用于存储数据

	最多允许损坏几个盘	Check Disk	校验方式
RAID 0	0	0	无redundancy
RAID 1	1	8	复制，可靠性高但成本高
RAID 2	1	4	（不重要）
RAID 3	1	1	第1个bit存在disk1中，第2个bit存在disk2中，依此类推 最后一个盘中为校验位，是前面每个盘的加和对2取模
RAID 4	1	1	第1个block存在disk1中，第2个block存在disk2中，依此类推 最后一个盘中为校验block，同RAID 3奇偶校验（parity）
RAID 5	1	1	（最常用）校验盘分布式存放
RAID 6	2	2	P + Q redundancy，原理类似RAID 5，增加一个校验盘

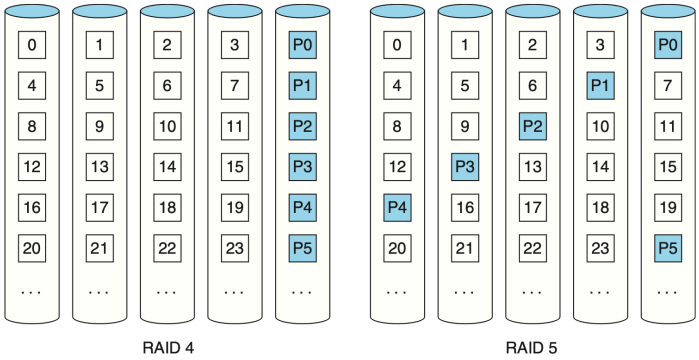
1. **RAID 4 vs. RAID 3**：小数据读

   在频繁的小数据读写操作中，RAID 4要显著优于RAID 3。RAID 3会频繁调用校验盘，导致其更容易损坏；同时，RAID 3涉及更多盘的读取，开销更大



2. **RAID 5 vs. RAID 4**：小数据写

   RAID 4在小数据读取上运行良好，但因为小数据的写涉及到P盘（校验盘）的访问，因此不同行之间的小数据写无法同时进行（比如想要写下图中的0和5，看上去不冲突，但实际上因为0和5的写都涉及到P盘的更新，但同一个盘只能顺序访问，因此无法并行），但RAID 5就解决了这个问题



注意，对于大的数据的写入，RAID 3、4和5的吞吐量是一样的。For large writes, RAID 3, 4, and 5 have the same throughput.

Buses and Other Connections：总线

Introduction

总线将processor memory和I/O设备连接起来。一个bus有两种line：

- **Control Line**：控制线，还表明了data line上数据的种类
- **Data Line**：用于在不同设备间搬运数据（地址数据也通过data line传输）

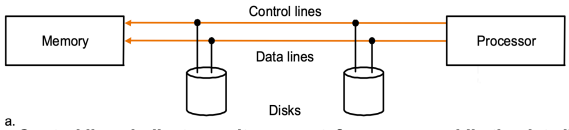
Bus在传输过程中，包含两个过程：

1. **Sending Address**
2. **Receiving or Sending Data**

2 Operations of Bus

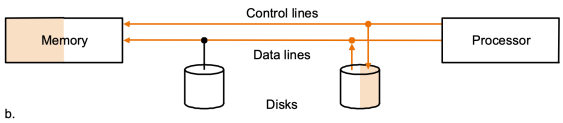
对于数据流向的不同，bus有两种operation：

- **Input**：从device到memory。处理器先从发出control和data（data line用于传输地址）信号，当mem准备好后发送信号给device，device收到指令后将数据给到mem



a.

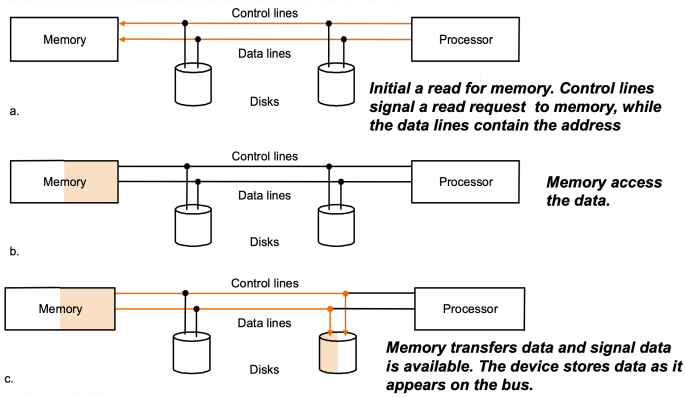
Control lines indicate a write request for memory, while the data lines contain the address



b.

When the memory is ready, it signals the device, which then transfers the data. The memory will store the data as it receives it . The device need not wait for the store to be completed.

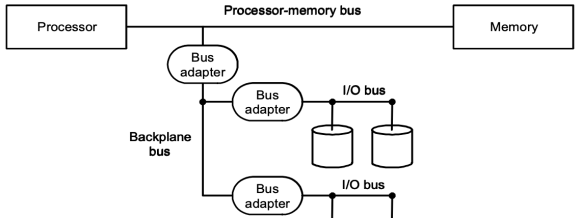
- **Output**：从memory到device。处理器发出control信号，并将地址从data line中传给mem；mem进行读取，并将结果从data line返回到设备中（此时control line也需要用到，因为需要区分操作和数据种类等）



Types of Buses

早先的计算机只有一条总线，处理器和主存以及IO设备的通信全都通过这一条总线来实现

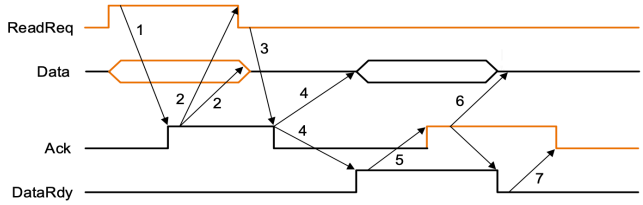
- **Processor-memory Bus**：连接处理器和主存的总线。短，高速
- **Backplane Bus**：一串IO设备通过I/O bus相连，一串I/O bus通过backplane相连。常标准化，高速
- **I/O Bus**：长，标准化。连接不同的I/O设备



.....

Synchronous vs. Asynchronous

- **同步总线**：通过时钟和固定的协议（fixed prorocol），让所有总线上的设备都采用相同的工作频率。为了减少时钟偏差（clock skew）导致的异常情况的发生，同步总线不能太长
- **异步总线**：通过握手协议（handshaking prorocol）实现不同设备间的数据在总线上的传输。具体有如下7步，我认为考试不会涉及具体的步骤，因此只要会做题就可以。而会做题只需要知道2、3、4和mem的读取是同步进行的，因此其消耗的时间为$\max\{3 \times \text{handshake}, \text{mem access}\}$



这个部分可能会考计算题，不过一般来说题目都比较直观，只要知道每个量是什么意思就可以做出来

> **Assume**: The synchronous bus has a clock cycle time of 50ns, and each bus transmission takes 1 clock cycle. The asynchronous bus requires 40ns per handshake. The data portion of both buses is 32bits wide.
>
> **Question**: Find the bandwidth for each bus when reading one word from a 200-ns memory.
>
> - **Synchronous Bus**：时钟周期为50ns，其读取数据需要如下三步：
>
> 1. 将控制信号和地址传给mem：50ns

2. 等待mem读取：200ns
3. 将数据传回设备：50ns

$$\Rightarrow \text{bandwidth} = 32\text{bit}/300\text{ns} = 13.3\text{MB/s} \tag{33}$$

- **Asynchronous Bus**：1、5、6、7需要4个handshake，2、3、4与mem read所需要的时间取大者

$$\Rightarrow \text{bandwidth} = 32\text{bit}/(4\text{ns} \times 40 + \max\{40\text{ns} \times 3, 200\text{ns}\}) = 11.1\text{MB/s} \tag{34}$$

Increasing the Bus Bandwidth: 时钟频率为200MHz，每传输64位需要1个时钟周期，传输地址也需要1个周期，每次bus传输间都需要间隔2个周期。A memory access time for the first four words of 200ns; each additional set of four words can be read in 20ns. Assume that a bus transfer of the most recently read data and a read of the next four words can be overlapped.

Find the sustained bandwidth and the latency for a read of 256words for transfers that use 4-word blocks and for transfers that use 16-word blocks. Also compute effective number of bus transactions per second for each case.

- **4-word**：每次传1个block，128位数据。地址一次传输，读出4word，然后分两次传回数据，然后间隔2个周期。每次读4block，一共需要读64次，上面的结果乘以64
- **16-word**：每次传1个block，512位数据。地址一次传输，读出16word。因为每读出4word都可以直接传回，不需要等待16word都读完了才分批传回。因此实际上传输数据的时间还是2个周期（只有最后4word需要额外的时间传回），然后间隔2个周期

Bus Arbitration：总线仲裁

Bus Master：显然不能让bus上的设备想读就读想写就写，因此需要一个bus master来初始化并控制所有的bus使用请求。但是，很多设备本身就含有bus master，仅通过bus master无法完全解决总线冲突的问题。在这之间进行协调就涉及到bus arbitration

Increase the Bus Bandwidth

- 提升data bus width，一次可以传输更多数据
- 采用分离的data address和data line，提升并行处理的能力
- Transfer multiple words（参考上面的例题）