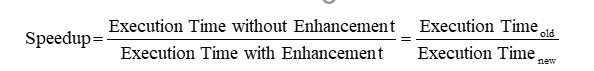
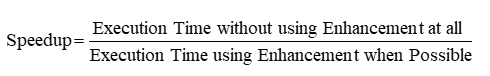
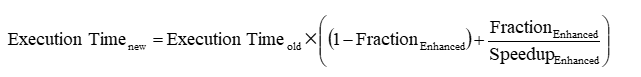
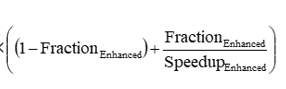
辅助理解、**加粗的感觉是重点、重点之重、不确定的**

# 01 Introduction（并行、分布式计算定义+并行五维度+并行计算要素+两个定律

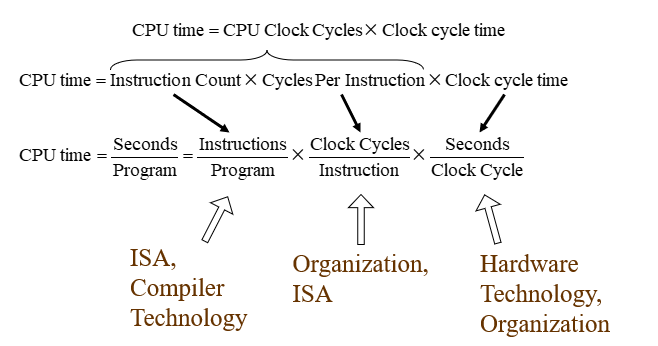
* 并行性的五个维度/方面（degrees of parallelism）
  + 比特级并行性Bit-level：比如ALU有32 bit，变多了)
    - ALU：算术逻辑单元，用于实现多组算术运算和逻辑运算的组合（逻辑电路）
  + 指令级并行性Instruction Level：pipeline, superscalar, VLIW
  + 数据级并行性Data Level: SIMD
  + 线程级并行性Thread Level: Chip Multiprocessors（单芯片多处理器/多核）
  + 作业级并行性Job Level: Cloud Computing
* 计算类型术语：
  + Centralized computing中心计算：所有计算机资源集中在一个物理系统中
  + **Parallel Computing并行计算：所有处理器要么与集中式共享内存（centralized shared memory）紧密耦合，要么与分布式内存（with distributed memory）松散耦合**
  + **Distributed Computing分布式计算：由多个独立的计算机组成（**地理位置是分布的**）**
  + Cloud Computing云计算:虚拟资源（计算资源虚拟化）
* 计算关键要素（比如评价并行系统的好坏主要考虑哪几个方面）SEDPPP
  + Scalability可扩展性
  + Efficiency效率
  + Dependability可靠性、依赖性
  + Programming Models编程模型（能否快速写出自定义的软件）
  + Power Consume功耗
  + Performance性能：大应用考虑速度，小应用考虑吞吐量
    - 区分带宽和吞吐量：
      * 传输速率：两个设备之间数据流动的物理速度（单位：bit/s || bps）
      * 带宽：单位时间内从一端传送到另一端的**最大数据量**，即最大传输速率。比如一个水管AB，平时2s从AB可以传4L水，传输速率2L/s，但是如果水管全满，则2s8L，最大传输速率4L/s（带宽）
      * 吞吐量：实际的带宽（不仅仅衡量带宽，还衡量CPU的处理能力，网络拥堵程度及报文中数据字段的占有份额）
* **Amdahl's law阿姆达尔定律（计算加速比speedup）**
  + 本质：旧的执行时间/经过提升之后的执行时间，为了计算速度提高了几倍
  + 通用公式（适用于整体加速的情况）
    - 
  + 提升部分情景：如果只提升了整体中的某部分Fraction（比如5%），而非全部
    - 1. 计算提升部分的加速比：
      * 
    - 2. 计算新的运行时间
      * 
        + 没提升部分的百分比：（1-FractionEnhanced）
        + 整体加速比例：
    - 3. 计算最终的总体加速比

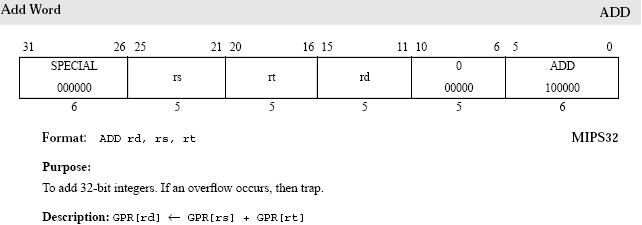
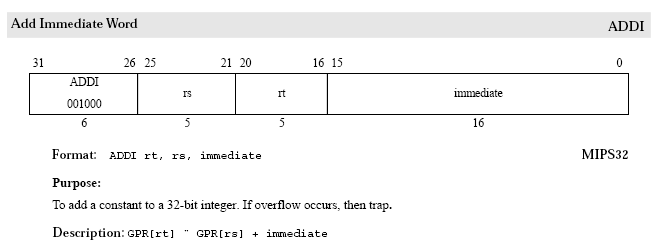
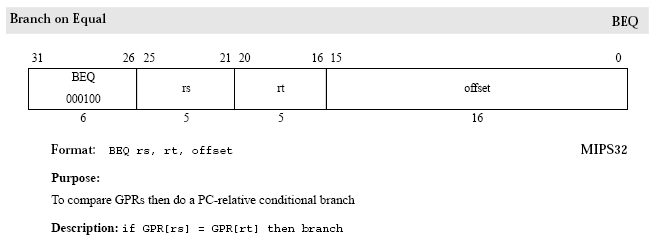
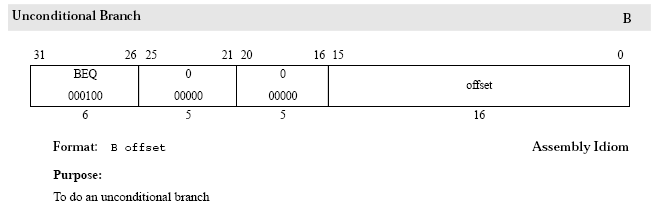
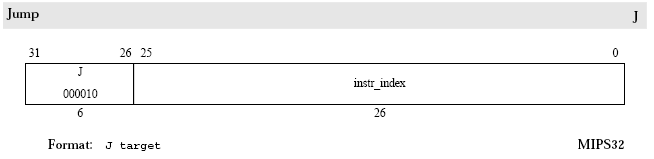
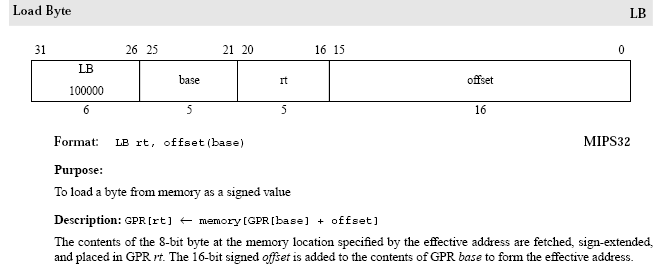
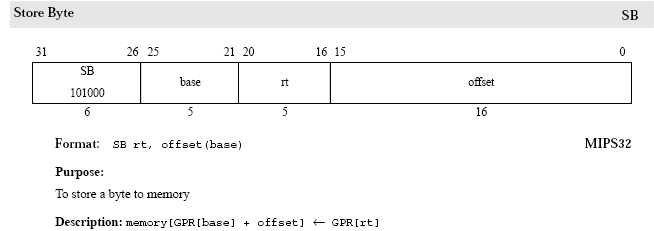
* + - * 
  + 例子：A对10%部分加速20倍，B对90%部分加速1.2倍，A的最终加速比为1.105，B的最终加速比为1.176，两者相差不大，是因为**“局部性原理”（Principle of locality），**对于B来说，大约90%的时间都花在了10%的代码上。
  + 并行提升情境：假设（1-P）部分不并行，P部分并行,并行比例n（并行处理器数量）
    - 提升后的运行时间
      * 加速比 = T（old）/T（new）
      * 比如n为2，则整体加速比例为（（1-P）+P/2），当n趋向于无穷大的时候，整体加速比例为（（1-P）+P/∞）≈ （1-P），则最终加速比=T（old）/T（new）=1/（1-P）
      * **结论：串行代码的比例（P）决定了最大的加速比**
* **摩尔定律Moore’s Law**
  + 集成电路（integrated circuits）上每平方英寸的晶体管（transistors）数量每年都翻一番
    - 处理器性能（processor performance）大概18个月翻两倍
    - Memory capacity大概每两年内翻两倍
    - **不过现在好像已经不是这个规律了**
* 发展趋势：晶体管数量上升、操作速度上升、物理电路尺寸下降、耗能上升、能量密度（power density）上升

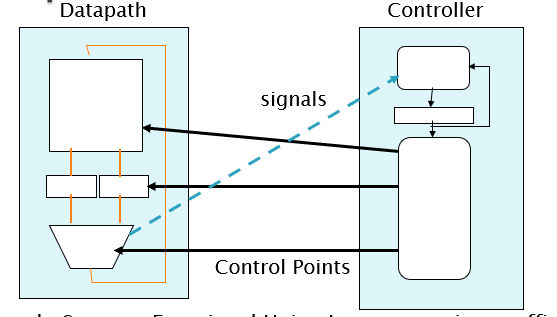
# 02-1. Instruction Level Parallelism 指令级并行pipeline

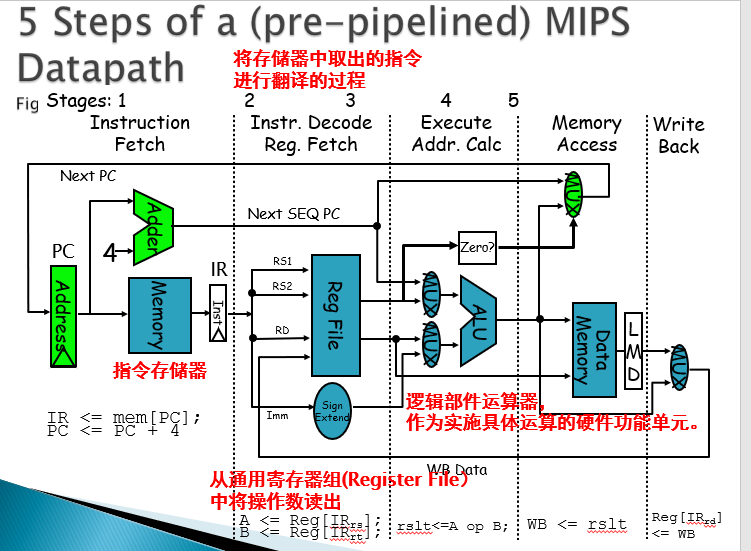
【性能测试因素CPUtime🡪为了减少CPUtime🡪**小指令（RISC ISA**）/大IPC（superscalar 、VLIW）/大CPU f(**pipeline**)】

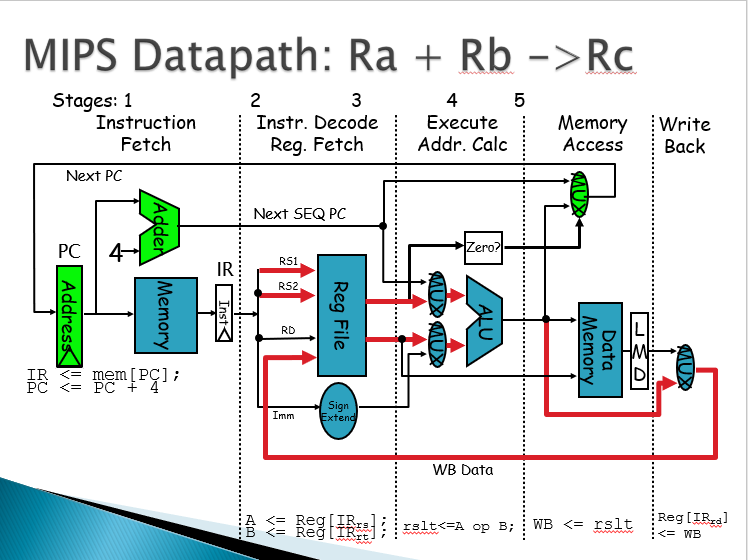
* **CPU性能计算🡪计算CPU time**

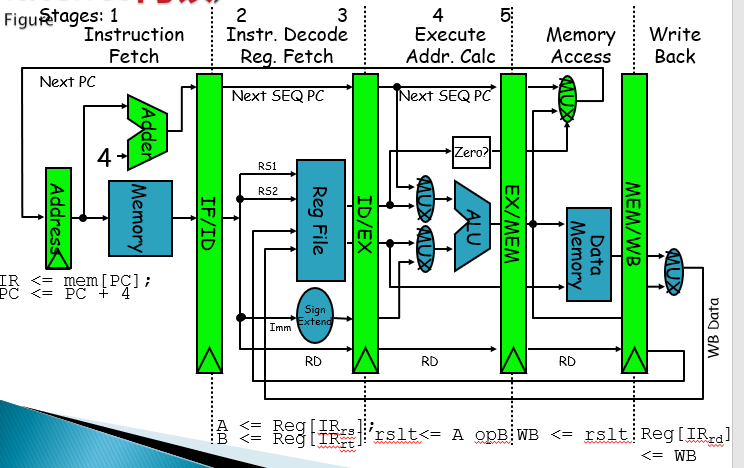
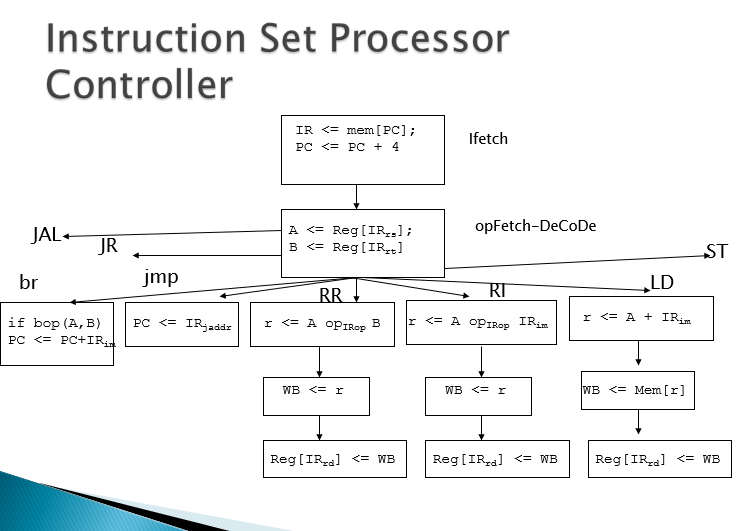
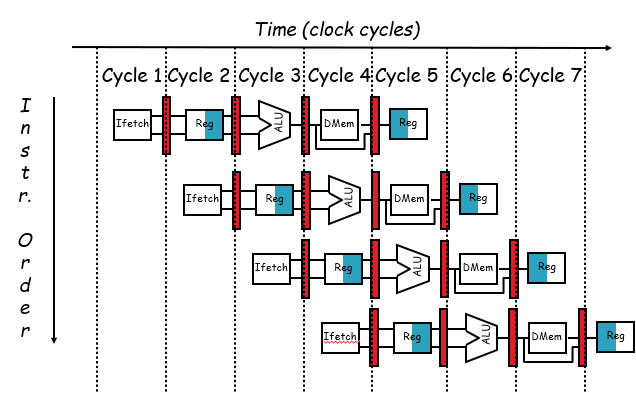
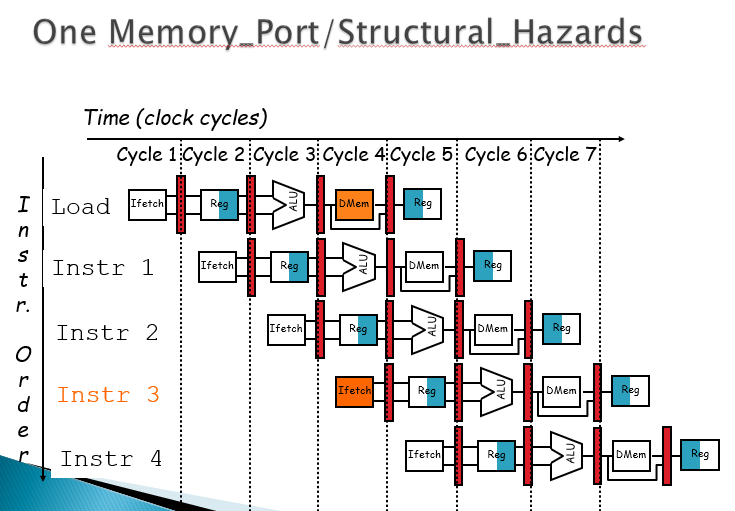
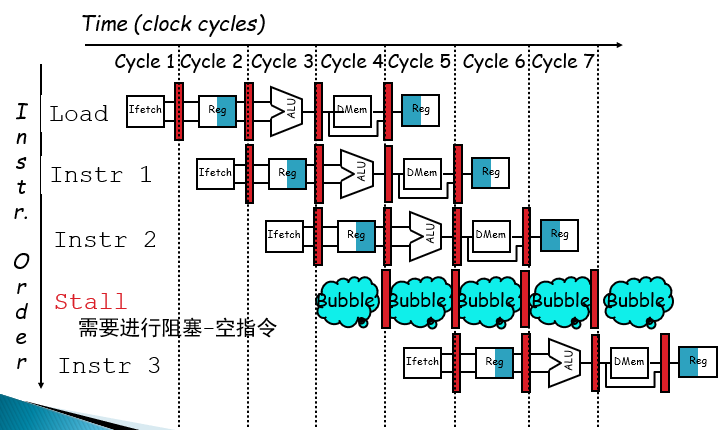
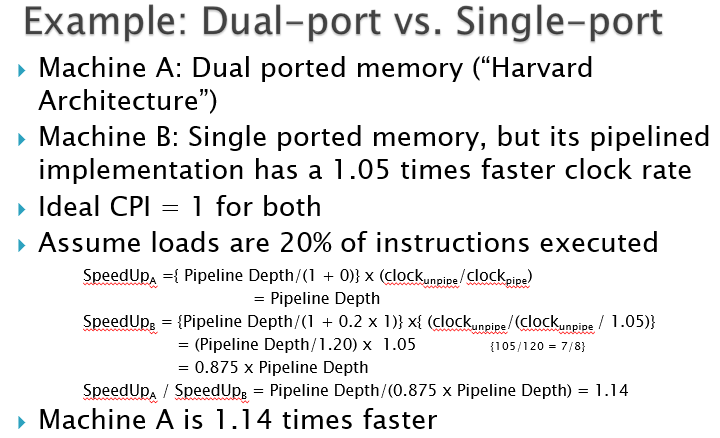
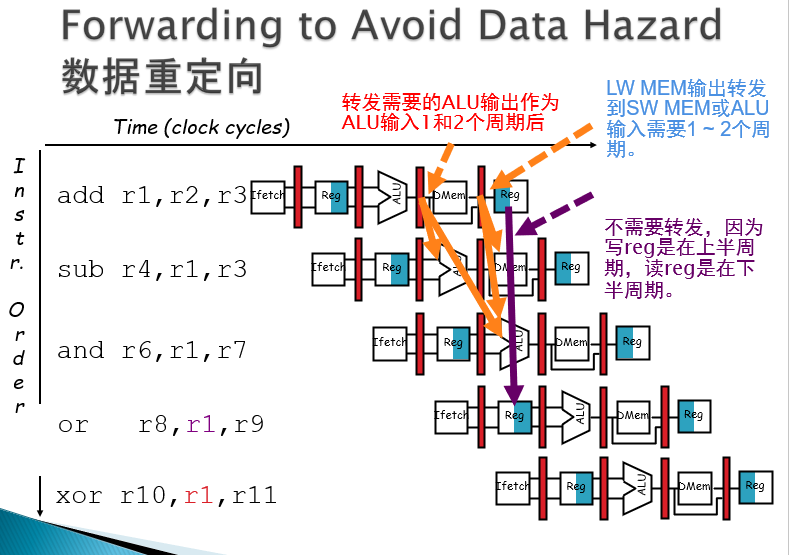
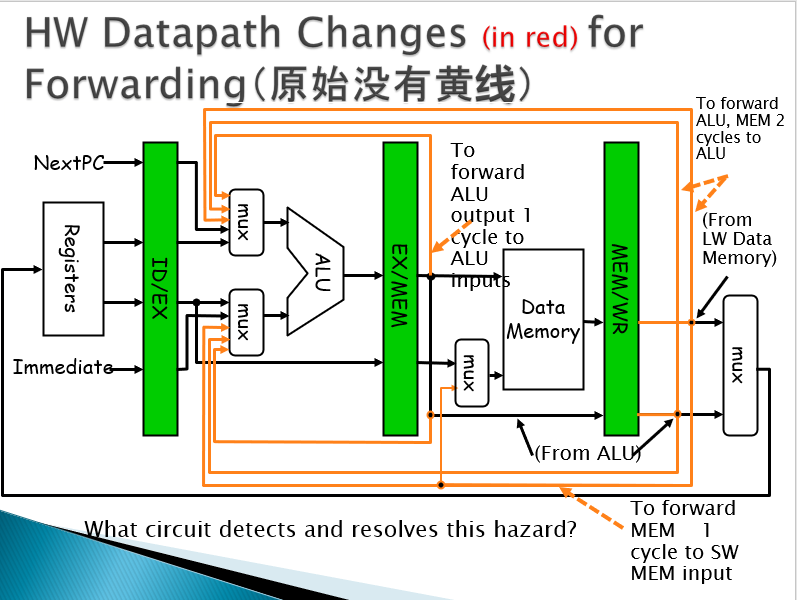
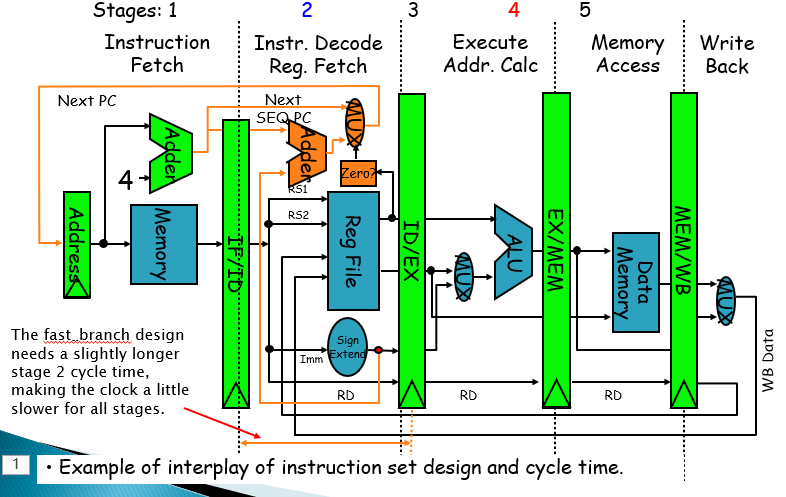


* + CPU time = 指令数 \* 每个指令有几个cycles \* 每个cycle要多少second
  + 指令数=Instruction Count = Instructions/Program: 每个程序需要执行多少个指令，由ISA和编译技术决定每个语句需要几个指令
  + Cycles per Instruction = Clock Cycles/Instruction = 每个指令需要几个Clock cycles， 由ISA决定
  + Clock cycle time = seconds / clock cycle = 每个clock cycle需要几秒，硬件技术决定
* **减少CPU时间(知道三个维度对应的技术)**
  + 降低Instruction count🡪complex instruction & simple instruction
  + 增大IPC（Instructions/cycle）：每个指令用更少的cycles完成，单位数量的cycle可以执行更多指令🡪superscalar & VLIW
  + 增大CPU频率（减少clock cycle time）🡪pipeline
* 降低Instruction count
  + RISC ISA: 精简指令集计算机（执行较少类型计算机指令的微处理器）
    - 是一个32-bit由三个部分R I J组成的指令
    - 对load/store为单地址模式：base + displacement
    - 简单分支条件
    - MIPS是一种具有RISC架构的指令
  + CPU定义的数据格式：Bit (b)、Byte (8 bits, B)、Halfword (16 bits, H)、Word (32 bits, W)、Doubleword (64 bits, D)
  + MIPS指令组（instruction groups）每秒百万条指令
    - Load and store 、Computational、Jump and branch、Miscellaneous 其他、Coprocessor 协处理器
  + MIPS指令编码（6+5+5+5+5+6））（GPR= General purpose register）
    - 算数操作（比如c=a+b）：Register-Register - R format
      * 
    - 立即数操作（比如b=a+1）：Register-Immediate - I Format
      * 
    - 相对中等距离条件分支：Branch - I Format PC=PC+offset
      * 
      * 
    - 长距离跳转/调用：Jump / Call - J Format
      * 
    - Load操作：I format
      * 
    - Store操作：I format
      * 
* Datapath & Control （指令怎么被获取和执行）

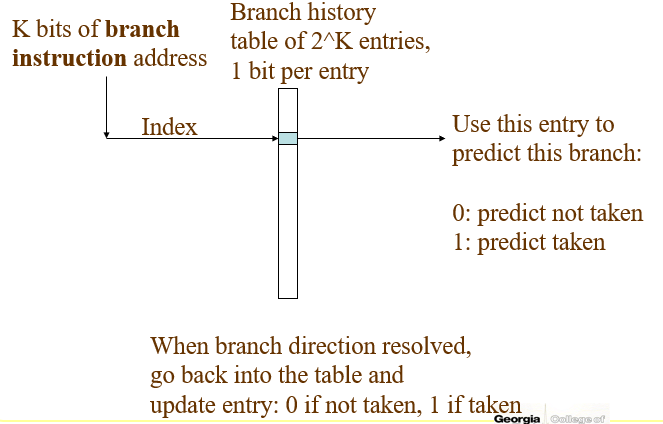
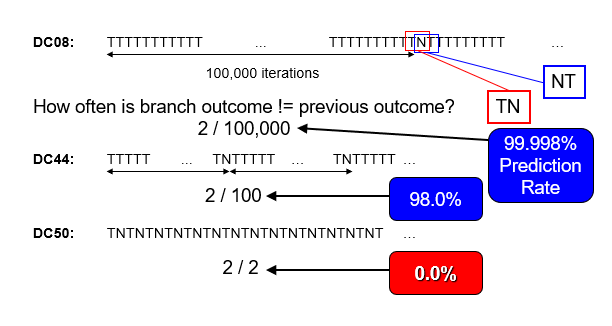
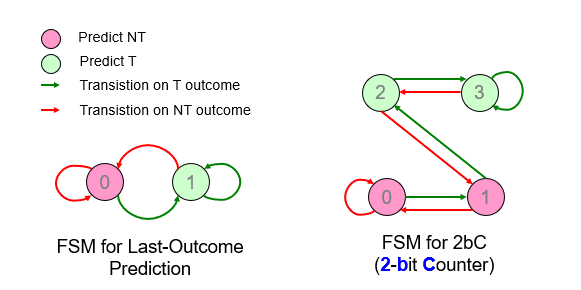
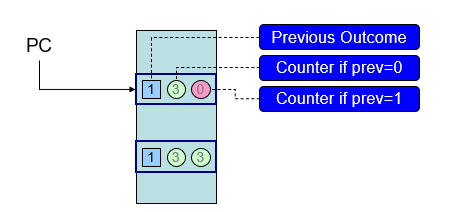
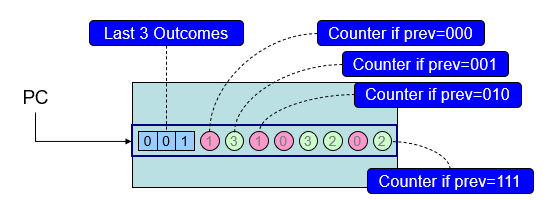
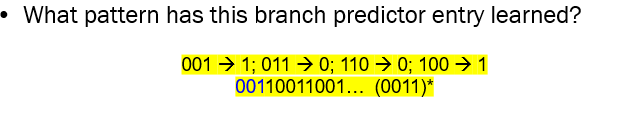
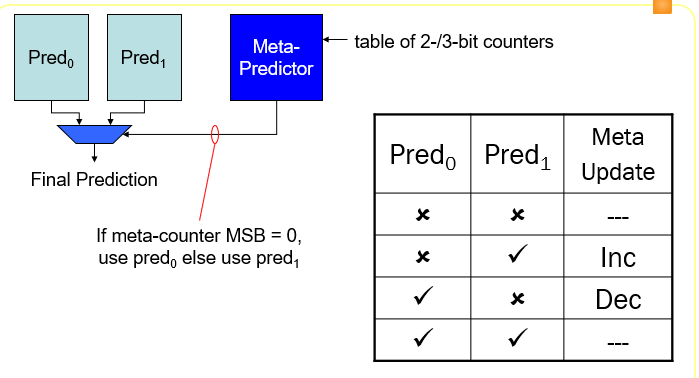
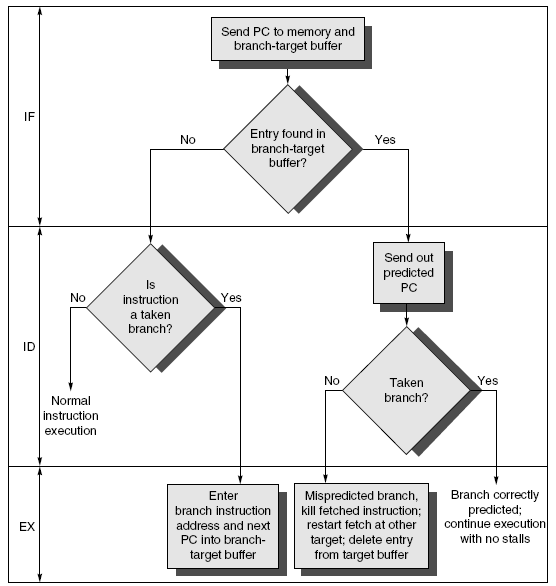
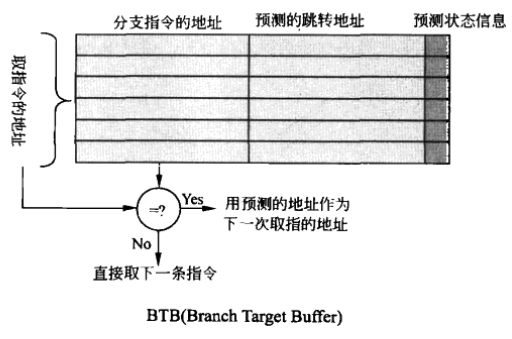


* + Datapath数据路径：存储模块、功能模块相互连接以运行想要功能。
    - 输入Control points，输出signal
  + Controller：状态机，根据signal和所要完成的功能协调data path中操作
* ISA：定义了一组指令格式、操作、硬件支持数据类型、命名存储、寻址模式、排序
  + Approaching an ISA:
    - 1. 给定技术约束，绑定对等的数据通路
      * 架构定义的存储如何映射到真实存储中【存储关系】
      * 功能单元（FUs）如何完成所需要的功能【功能执行】
      * 可能的额外存储（例如内存地址寄存器，内存数据寄存器，指令寄存器）
      * 如何在寄存器和功能单元中移动信息，两者如何互连
    - 2. 把指令映射到一个RTL操作序列
      * 架构寄存器（architected registers）和内存（memory）中的RTL(Register transfer language)描述了每个指令的意思
    - 3. 将RTL操作序列整理成符号控制器状态转换图(STD)
    - 4. 降低符号控制器状态转换图（STD）到control point
    - 5. 实现controller
* MIPS DataPaths的五步实现/五级流水线
  + Instruction Fetch(取指): 从存储器 中取出指令
    - IR <= mem[PC];
    - PC <= PC + 4
  + Instruction Decode（解码），Register Fetch(取操作数)：对指令进行翻译，从通用机存储器将操作数读出
  + Instruction Execute(执行)，Address calculate（地址计算）：
    - ALU（逻辑部件运算器）：作为实施具体运算的硬件功能单元
  + Memory Access(访存)：存储器访问指令将数据从存储器中读出，或写入存储器
  + Write Back(写回)：将指令执行结果写回通用寄存器组的过程。如果是普通运算指令，该结果值来自于“执行”阶段计算的结果; 如果是存储器读指令，该结果来自于“访存”阶段从存储器中读取出来的数据。



* MIPS DataPath有pipeline latches
  + 
* 指令集处理器控制器
  + 
* DataPath流水线
  + 
* **使用pipeline会存在的三种hazard：**
  + Structural hazards：硬件并不支持这种指令的合并（比如一个人同时叠衣服和收衣服）
    - 比如只有一个memory port，存储器是单输入单输出
    - Dual-port vs. Single-port：dual-port有两个关口可以一个读指令，一个读数据
    - 解决办法：需要进行阻塞stall-控制零，使用bubble
    - （出现hazard）
    - （需要stall）
    - **计算pipeline下的加速比（structural harazds的情况）**
      * CPIpipeline=Ideal CPI + Averages stall cycles per Instruction
      * 
      * For simple RISC pipeline, Ideal CPI = 1:
      * 
      * 例子
        + 
  + **Data hazards: Instruction depends on result of prior instruction still in the pipeline**
    - 三种数据冲突的类型
      * Read after write (RAW)：先写后读，还没把处理的数据存入存储器又被都出来了“(True) Dependence
      * Write after Read (WAR)：先读后写，（在顺序执行的指令执行时不会发生冲突，但其如果在乱序处理中，可能会发生冲突，（顺序变化））anti-dependence
      * Write After Write (WAW)：同时都要写，output dependence
    - Data hazard解决方法
      * 调整程序顺序
        + 编译器优化表现，硬件保证安全
        + 但是原始代码和执行顺序并不一致
      * Forwarding to Avoid Data Hazard 数据重定向
        + ALU放前面，L\S放后面没问题
        + 如果L\S放前面，ALU放后面，forward无效，只能放bubble
        + 
        + 
  + Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps). 由获取指令和关于控制流(分支和跳转)更改的决策之间的延迟引起。
  + Control hazard原因与解决方法
    - 在原始的5级流水线设计中，branch指令的完成在步骤4（Memory Access），前分支需要执行完才知道后面一条指令是哪个（branch🡪jump）
    - 两个因素决定浪费的影响：有多少概率（P）代码会发生跳转 &每次跳转浪费3条指令🡪CPI=CPI+3\*p
    - 问题解决方法
      * **预测发生跳转的概率（先去拿预测的分支地址**）-分支预测
      * 早期1986年MIPS提出在ID&RF环节先增加额外的Program Counter地址空间（早点确定是否需要跳转，并存放分支指令的pc）
      * 

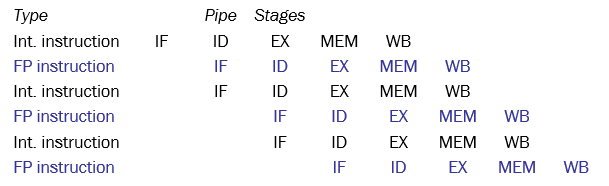
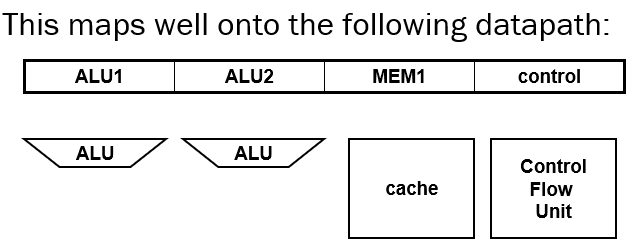
**02-1-1解决Control hazard方法：Branch Prediction**

* 预测分支：
  + 对于预测分支需要知道a）分支是否发生（下一步的direction）b）如果发生的target address是什么（target）
  + 预测path主要是事先Fetch，decode指令，对于是否execute指令 a) 当分支处理结束后执行 b）不论如何都先执行；
  + 如果预测错，需要对correct path重新fetch
* 三种分支类型
  + Direct jump (函数调用)：肯定发生+地址容易计算
  + Conditional branch 条件分支(一般为PC-relative)：不知道是否需要跳转+地址容易计算
  + Indirect jump(函数返回)：肯定发生+地址不容易计算
* 对于a）分支是否发生（direction）的预测方法
  + 静态预测：固定的规则，比如BEQL（branch if equal likely，一般if的条件比较容易发生）🡪总是预测T或者NT或则BTFNT（总是预测这个loop会发生）
  + 动态预测：通常基于历史（硬件预测）
* One-bit branch predictor：根据该指令上一次是否跳转来预测当前时刻是否跳转。如果该跳转指令上次发生跳转,就预测这一次也会跳转,如果上一次没有跳转,就预测这一次也不会跳转。
  + 
  + 例子
    - 
* 
* Two-bit branch predictor：每个跳转指令的预测状态信息从1bit增加到2bit计数器,如果这个跳转执行了,就加1,加到3就不加了,如果这个跳转不执行,就减1,减到0就不减了,当计数器值为0和1时,就预测这个分支不执行,当计数器值为2和3时,就预测这个分支执行。**2位的计数器比1位的计数器拥有更好的稳定性。**
* 使用1bit或2bit的预测效果并没有很好，可能需要考虑branch correlation
  + Branch correlation：The outcome of a branch is often related to previous outcome(s)
  + 需要track the history of a branch,记录：
    - previous outcome；ounter if prev = 0；counter if prev = 1
    - 
    - 也可以记录deeper history covers more patterns
    - 
    - （看不懂）
* Local branch history – local behavior
  + 根据分支A以前实例的结果，预测分支A的direction（发不发生）是什么
* Global branch history – global behavior
  + 考虑到之前所有的分支A, B…的结果（分支结果的记录长度受到history length），预测分支Z的方向是什么
  + global correlation存在的情景：
    - related branch condition：比如有两个互斥条件A和B，条件A发生了，条件B一定不发生（互斥）
    - Testing same/similar conditions
      * partial correlations: one branch could test for cond1, and another branch could test for cond1 && cond2 (if cond1 is F, then the second branch can be predicted as F)
      * multiple correlations: one branch tests cond1, a second tests cond2, and a third tests cond1 + cond2 (which can always be predicted if the first two branches are known).
* Tournament Predictors：同时有多个predictor，哪个predictor预测效果好就用哪个
  + 
* 有一些predictor方法的combination
  + Global history + Local history
  + “easy” branches（1 bit 或 2bit）+ global history
  + short history + long history
* Target Address Prediction：分支指令在执行后,会将这条指令的地址及它的跳转信息记录在BTB中。BTB buffer没有太大,不能将所有的分支指令都存进去,通常采用Hash表的方式存入。在取指时,先将PC(程序指针)和BTB中的分支指令的地址进行比较,如果找到了,说明这条指令是分支指令,并且在BTB中有记录,就使用BTB预测出来的跳转地址。如果没有记录,就不能使用BTB的信息了,取指下一条指令。（需要高指令带宽）
  + 
* Branch folding
  + 大的BTB，将目标指令添加到缓冲区中，以处理较大的缓冲区所需的较长的解码时间
* Return Address Stack (RAS)：函数调用在程序中大量出现,函数调用与返回也都是通过跳转来实现的。例如,有3个函数调用了printf函数,pinf函数地址固定,调用时知道地方,但是在返回时,并不知道该返回到哪个地方, Retum Stack(返回栈)可以用于解决这个问题。在函数调用时,将函数的返回地址压栈到 Retum Stack中,当遇到函数返回指令时,就从 Retum Stack中取出地址。

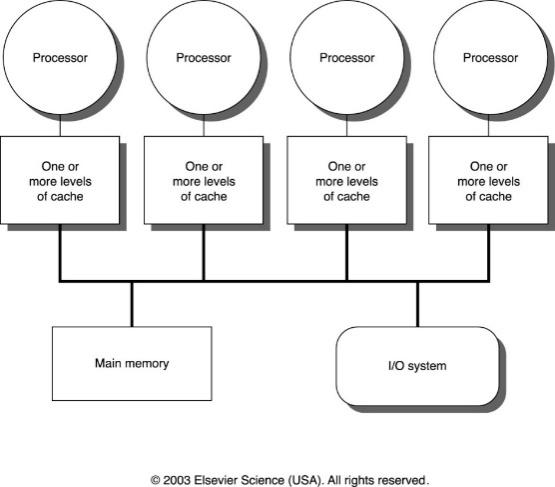
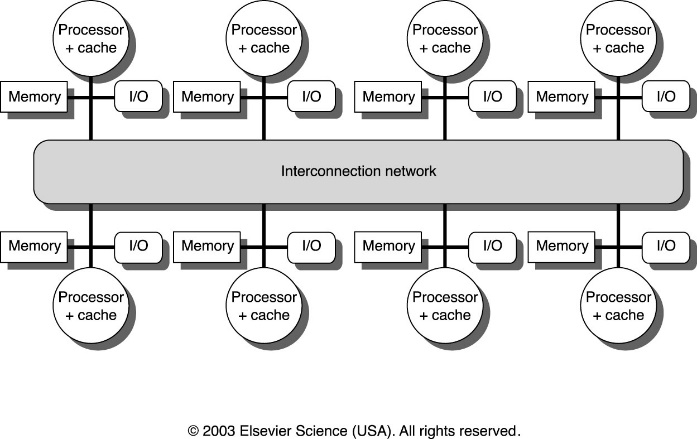
# 02-2. Instruction Level Parallelism 指令级并行 superscalar, VLIW

<https://max.book118.com/html/2018/0308/156371180.shtm>

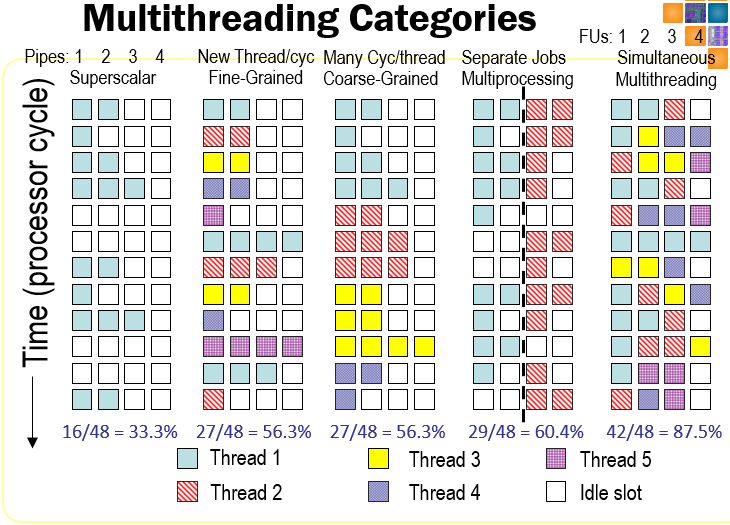
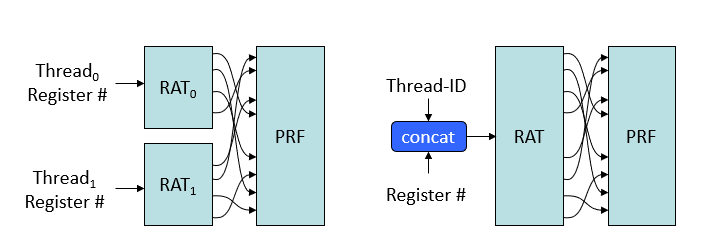
减少CPU time：增大IPC（Instructions/cycle）🡪每个指令需要更少的cycles，单位cycle可以执行更多的指令（也就是降低CPI，每个指令需要的cycles）

* 方法：多条指令/cycles 🡪 （CPI<1）
  + VLIW（very long instruction words）:fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates；将多操作打包成单个指令（超长指令集架构）
  + Superscalar：不同的IPC（每个cycle可执行多个instruction），由编译器或硬件安排
  + Vector processing：独立循环的显式编码作为数字的大向量处理(Explicit coding of independent loops as operations on large vectors of numbers); Multimedia instructions被添加到many processors
* Superscalar MIPS：64bit上有2个指令：一个FP浮点指令，一个非浮点（算数）指令（每个周期可以执行一个以上的指令），由编译器或硬件安排🡪本质：加入更多的流水线
  + 格式：左边：integer/load、右边：FP算数
  + 第一个指令发出，第二个指令才发出
  + 需要更多的寄存器端口
  + 
    - 只看int instruction就是正常的5步流水线方法
    - 每一组的FP instruction会与该组的int instruction差一个stage
* Multiple-Issue processors的设计复杂性(多重启动处理器) （控制、性能、结构）
  + Issue packet: 1 clock内属于fetch unit的一组指令
    - 如果指令导致structural hazard或data hazard（由于还在执行中的较早指令或还在issue packet中的较早指令），那么指令不应该被issue
    - 对于N-issue，每个clock cycle发出0到N个指令
  + 对Issue的检查可能需要耗费1 cycle（:O(N2) comparisons）
    - issue stage通常分裂并流水线化：1st stage决定packet内issue多少条指令，, 2nd stage检查选定instruction和已经发出的instruction中的hazards
  + 如果对与branch没有预测准确，会有很高的branch延误
  + 对于硬件来说，区分出integer/FP很容易，但程序中需要准确的一般FP操作和无hazard才有可能达到CPI=0.5
  + Decode的困难
* Dynamic Scheduling in Superscalar（如何保证指令按顺序执行，load/store可能造成依赖）
  + 将load reservation station替换为load queue
  + 操作数必须按照Load被fetch的顺序读取
  + Load queue检查store队列中的地址，以避免RAW violation
  + Store queue检查in Load & Store队列中的地址，以避免WAR和WAW
* 寄存器重命名，虚拟寄存器vs缓冲区重新排序（reorder buffer）
  + reorder buffer的替代方法是使用一个更大的虚拟寄存器和寄存器重命名
  + 虚拟寄存器有架构上可见的寄存器和临时值：replace functions of reorder buffer and reservation stations
  + Renaming process将 体系结构寄存器 的名称 映射到 虚拟寄存器集中的寄存器
  + 简化指令提交:将寄存器标记为不再speculative，用旧值释放寄存器
* 静态调度和动态调度
  + 静态调度：依靠编译器对代码进行静态调度，以减少相关和冲突。
    - 它不是在程序执行的过程中、而是在编译期间进行代码调度和优化对相关的处理方法。在程序执行过程中始终不变。通过把相关的指令拉开距离来减少可能产生的停顿。
  + 动态调度：在程序的执行过程中，依靠专门硬件对代码进行调度，减少数据相关导致的停顿。
* VLIW Processors：将多操作打包成单个指令（超长指令集架构）🡪本质：多个指令一起做
  + 有多个独立的功能单元
  + 单个指令包含多个独立的指令，并且每个独立的指令对应一个功能单元
  + 具有固定的latency
  + 编译器将指令打包到VLIW中，同时也调度所有的硬件资源
  + 整个VLIW问题作为一个单一的单元
  + 是一种用简单硬件实现的ILP（更快的硬件控制，更快的clock）
* VLIW资源分配
  + VLIW可以表示为资源的分配
  + 
* VLIW的关键技术：**编译器**判断这种map
  + Predicated Execution
  + Trace Scheduling：If-Conversion
  + Software Pipelining
* VLIW处理器的缺点
  + 静态发现并行性
  + 代码大小，指令空间可能浪费
  + 无危险检测硬件
  + 二进制代码的兼容性
* ILP的limit
  + Application benchmarks（程序中FP和integer的指令数量不相等）
  + 硬件复杂性
  + 编译器复杂性

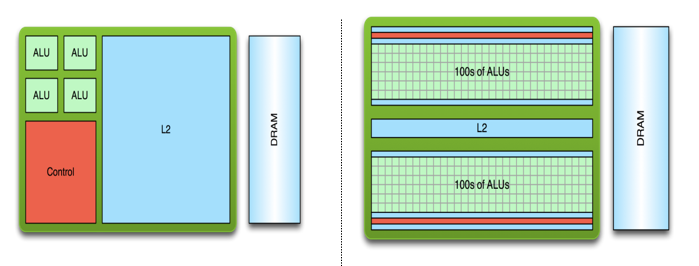
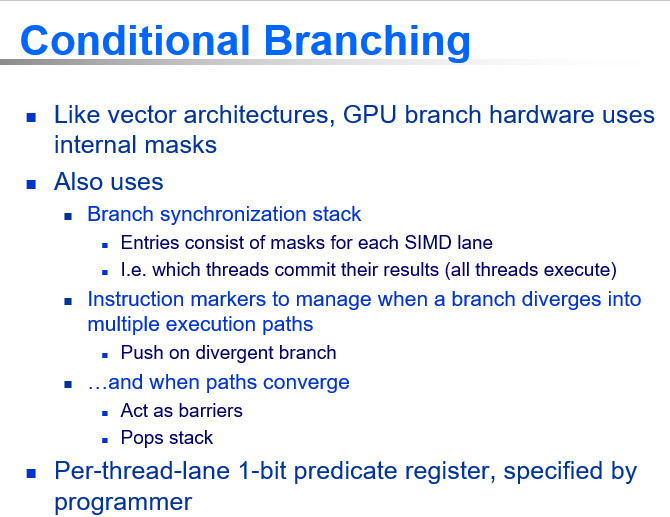
**03-1 Multiprocessors**

* Flynn’s Taxonomy（考虑指令流、考虑数据流）
  + SISD：uniprocessor单处理器
  + SIMD：比如vector processor/MMX
    - 每个“处理器”在自己的数据上运行，但他们都同步执行同样的指令
  + MISD：不经常被使用
  + MIMD
    - 每个处理器执行自己的指令，操作自己的数据
    - 是一种多处理器（multiprocessor），包括多核处理器
* Multiprocessor
  + 主要是为了得到更快的处理速度
  + 问题：多数代码都要求顺序化（for uniprocessor），不好并行处理
* MIMD multiprocessor memory
  + Centralized shared memory(对称的多处理器（SMP）)
    - 统一内存访问（UMA）
      * 所有的内存地址都有相似的延迟
      * 通过内存读写 共享数据
    - 问题-内存（memory content）：所有的处理器共享单个内存，内存带宽成为限制瓶颈，所以适用于小机器（比如有2，4，8处理器的）
  + Distributed memory
    - 两种类型：Distributed Shared-Memory (DSM) & Message-Passing
    - Distributed Shared-Memory)(DSM)：
      * 所有处理器可以在所有memory location中进行寻址
      * 类似Centralized shared memory的数据共享（通过内存读写），也称为UUMA（非统一内存访问）
      * 不同的内存地址的延迟不同（本地会比远程快）
    - Message-Passing（multicomputer/cluster）
      * 单个处理器只能在local memory中直接寻址
      * 为了与其他处理器通讯，必须显式的发送/接收消息
    - 相比Centralized shared memory，其处理器更多访问local memory，所以有更少的memory contention问题

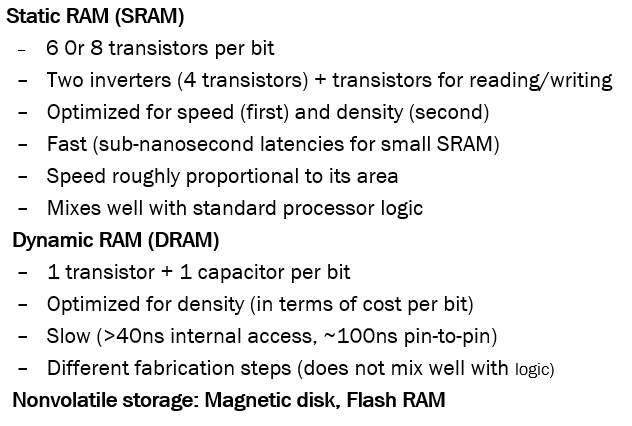
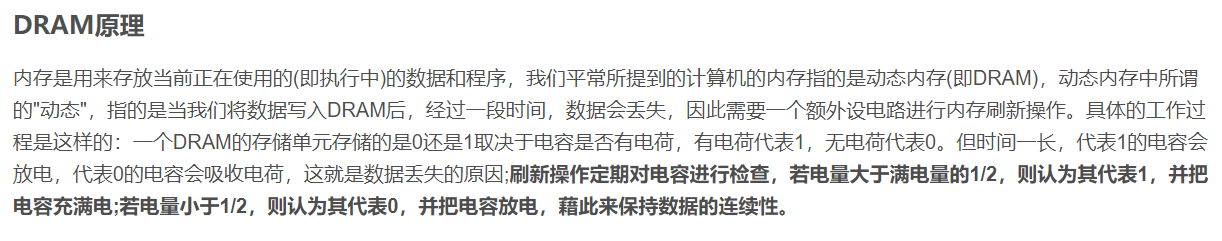
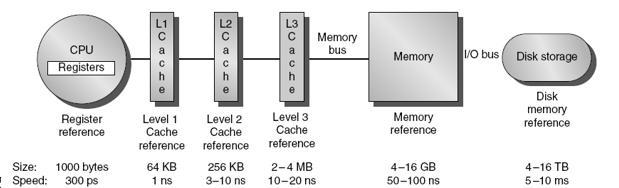
# Thread Level Parallelism （multicore process）

* Thread Level Parallelism为什么出现：针对任意代码的ILP现在被限制为3到6个指令/周期，但在一些应用程序(如数据库或科学代码)中可以有更高的自然并行性。
* Thread定义：a process with its own instructions and data（has all the state necessary to allow it to execute）
* Thread Level Parallelism：有多个并行执行的线程（TLP explicitly represented by the use of multiple threads of execution that are inherently parallel），目的是得到运行多程序的高吞吐量和减少多线程程序的运行时间
* Multithreading定义：多个线程通过重叠执行 共享一个处理器的功能单元
  + 处理器必须复制每个线程的独立状态
  + 通过 支持多进程的虚拟内存机制 共享内存
  + HW中，快速线程切换（0.1 to 10 clocks） 比 需要复制状态(state = registers and memory)的完整进程切换(100s to 1000s of clocks) 快得多。
* **Multithreading的不同粒度（线程切换时间）**
  + Fine Grained：每个cycle切换线程
  + Coarse Grained：每几个cycle切换线程（切换时机可能是发生cache miss）
  + Simultaneous Multithreading（SMT）（hyperthreading）：每个周期可以执行不同的线程，也可以从多个pc同时取指。
  + 
* **Fine Grained Multithreading细粒度多线程**
  + 定义：在每个指令周期（cycle）都进行线程交换，使得多个线程交替执行
  + 技术：通常使用round-robin的方式，跳过stalled的指令；并且CPU要能够在每个cycle切换线程
  + 优点：能够隐藏短和长的stall，因为当thread stall时，来自其他线程的指令被执行
  + 缺点：减慢了单个线程的执行速度，因为一个准备执行而没有stall的线程会被其他线程的指令切换从而延迟
* **Course Grained Multithreading粗粒度多线程**
  + 定义：发生stall时切换线程，比如cache miss或data memory reference
  + 优点：
    - Relieves need to have very fast thread-switching (if use caches).
    - 不会减慢线程，因为其他线程的指令只有在active thread遇到costly stall时开始被执行
  + 缺点：遇到很短的stall，会浪费pipeline的启动开销
    - 因为CPU通常只在一个线程上执行指令,发生stall时,pipeline必须清空或冷冻，而在指令完成之前，新线程必须填充管道
  + 所以根据start-up overhead，该多线程更适用于stall time大于pipeline refill time的high cost stall
* **Simultaneous Multithreading（SMT）（hyperthreading）：**
  + 定义：每个周期可执行不同的线程，也可以从多个pc同时取指。相当于把一个大的单处理器看出多处理器
  + 对于N-way SMT，需要
    - Ability to **fetch** from N threads
    - N sets of **registers** (including PCs)
    - N **rename tables** (RATs)
    - N virtual **memory spaces**
  + SMT Fetch（多个fetch logic）：可以在活动线程之间进行简单的循环，或者根据每个线程相对于其他线程的停顿程度来优先考虑某些线程
  + SMT rename：独立的name space，需要消除同名歧义（使用重命名）
    - 
  + SMT Cache：每个进程都有独立的虚拟地址空间
    - TLB是线程层面的，可以把线程id，虚拟page翻译成物理page
    - Virtual portion of caches是线程层面的
  + SMT Commit：
    - Register File Management：ARF/PRF organization，need one ARF per thread
    - 在每个线程的层面上，保留interrupts, exceptions, faults
  + SMT Performance：用其他线程的工作填充空闲的“槽”，但可能会造成performance degradation（因为要一起完成两个事情，对于单个事情的完成时间就变长了）
* **Cache thrashing 抖动**
  + Cache足够大到可以装下一个thread需要的data，但装不下两个thread的data，所以执行两个thread的时候都会有更高的cache miss rate
* **CMP：**单芯片多处理器，也指多核心。其思想是将大规模并行处理器中的SMP（对称多处理器）集成到同一芯片内，单个处理器并行执行不同的进程。
* **感谢后面的具体实例不同公司的产品应该不需要看了**

# 04. Data Level Parallelism 数据级并行性（SIMD+SIMT）

* <https://www.zhihu.com/question/273989958/answer/1080662354>
* Data Level Parallelism: 对数据执行相同的((lock-step))操作（有大量数据）
* SIMD（单指令流多数据流）
  + 能复制多个操作数，并把它们打包在大型寄存器的一组指令集。
  + 比MIMD更有效率，因为SIMD在每个data operation只需一个指令
  + 以加法指令为例，单指令单数据（SISD）的CPU对加法指令译码后，执行部件先访问内存，取得第一个操作数；之后再一次访问内存，取得第二个操作数；随后才能进行求和运算。而在SIMD型的CPU中，指令译码后几个执行部件同时访问内存，一次性获得所有操作数进行运算。这个特点使SIMD特别适合于多媒体应用等数据密集型运算
* SIMD 并行性的实现
  + Vector architecture
  + SIMD extension
  + GPU
* SIMD extension与Vector instruction相比的limitation
  + Number of data operands encoded into op code
  + 无复杂寻址模式(strided, scatter-gather)
  + No mask registers
* GPU（graphical processing units）
  + 使用异构的执行模型：CPU是host，GPU是device
  + 开发一种类似于c语言的GPU编程语言
  + 统一所有形式的GPU并行 作为CUDA线程
  + 编程模型是“单指令多线程”
  + GPU Computing = Throughput Computing
  + CPU & GPU
    - GPU是一种大规模的并行处理器，拥有数百个简单的内核，旨在解决并行问题。擅长处理大量类似的数据，因为这个问题可以被分割成数千个部分并同时计算。
    - GPU有很多很多的ALU，CPU比较多空间给cache（latency vs throughput）（异构）
      * 
    - 让我们使用Photoshop来演示不同类型的工作负载。当您打开应用程序时，这是一个非常串行的任务。操作系统从内存或磁盘读取数据，并连续运行指令，使应用程序启动并运行。然而，当你开始编辑照片时，比如去模糊或改变红眼…这是一个高度并行的任务。这是因为照片中有数千个像素需要并行计算才能执行这些编辑更改。
    - CPU是更擅长于基于串行的任务的处理器，比如运行操作系统和组织数据。
    - 对逻辑部分使用延迟（Latency）优化的处理器（CPU），计算部分使用吞吐量（Throughput）优化的处理器（GPU）
* SIMT（单指令多线程）
  + 每个Block分为多个warp（里面有32个thread）
  + Warp是最小的调度单元，并总是执行同样的指令（SIMT）
  + 每个thread都可以指令自己的code path
  + 具有自由的上下文切换（Context switching is free）
  + 大量的warp就可以掩饰memory的latency🡪让人觉得很快
* Threads of SIMD instructions
  + 每个都有自己的PC
  + 线程调度器使用scoreboard去分派
  + 线程之间没有数据依赖
  + 保持对SIMD指令多大48个线程的跟踪🡪掩盖memory latency
* Fermi Architecture Innovations
  + Each SIMD processor has 2 SIMD thread schedulers, + two instruction dispatch units
  + Thus, two threads of SIMD instructions are scheduled every two clock cycles
* Roofline Performance Model
  + 将浮点峰值吞吐量作为算术强度的函数绘制出来，将目标机器的浮点性能和内存性能捆绑在一起
  + 算术强度：每个字节读取的浮点操作
* 

**05 Cache system**

* 存储器类型：SRAM, DRAM, 非易失存储器(disk)
  + Cache追求的是速度所以选择SRAM，而内存则追求容量所以选择能够在相同空间中存放更多内容并且造价相对低廉的DRAM
  + 主存储器是DRAM
    - 因为其需要周期性的被刷新
    - 地址被分为两部分（memory as a 2D matrix）
      * RAS or Row Access Strobe
      * CAS or Column Access Strobe
  + Cache是SRAM
    - 不需要刷新
  + 
  + (辅助理解)
* 随着多核处理器的出现，memory hierarchy的设计越来越重要（memory hierarchy）
  + 
* 数据局部性原理（基于数据局部性原理，cache主要存储这两类数据）
  + 时空(temporal)：如果现在需要某数据，在不久的将来它很可能再次被需要
  + 空间(spatial)：如果现在需要数据项，在不久的将来可能需要附近的数据
* 术语解释：
  + Latency：因为cache没有命中产生的penalty（延迟）
  + Access time：request和word arrival之间的时间
  + Cycle time：两个request之间最小的时间
  + bandwidth带宽: I/0或large block没有命中的延迟
* Cache: 靠近处理器的快（但小）的内存，主要是为了得到“最优的平均内存访问延迟”
  + 作用：查找数据时，如果cache有，就用cache的不用内存的；如果cache没有，就把数据（entire block of data）放入cache（需要踢出去一些block）
  + Cache数据块内容变化的时候需要考虑
    - Placement：Block可以放在cache哪个line（三种策略）
      * Direct mapped (block can go to only one line)
      * Fully Associative (block can go to any line)
      * Set-associative (block can go to one of N lines)
    - Identification：我们怎么找在cache里面的block
      * 先判断data是否在cache里面，如果在找出cache在哪里-🡪cache lookup
      * Cache中的每个line都有一个valid bit（1表示该行有数据，0表示没有），有个tag表示哪个block在该line中
    - Replacement: 当我们要把新的block放进cache时，需要把哪个block一出去
      * Random (randomly selected line)
      * FIFO (line that has been in cache the longest)
      * LRU (least recently used line)
      * LRU Approximations
      * NMRU
      * LFU
    - Write policy：我们怎么处理“存储”操作
      * cache写miss时是否考虑修改cache
        + Write-allocate

A write miss brings block into cache

* + - * + No-write-allocate

A write miss leaves cache as it was

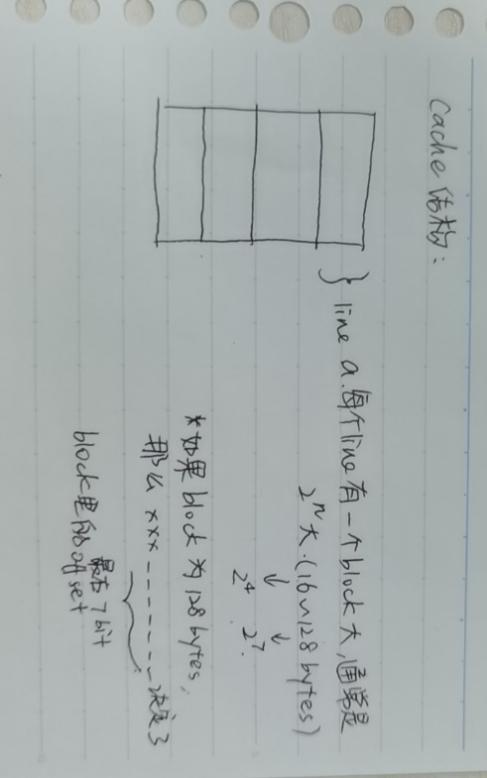
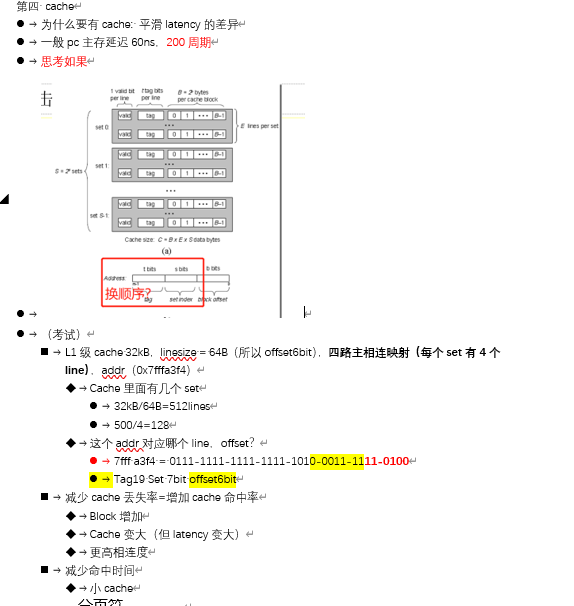
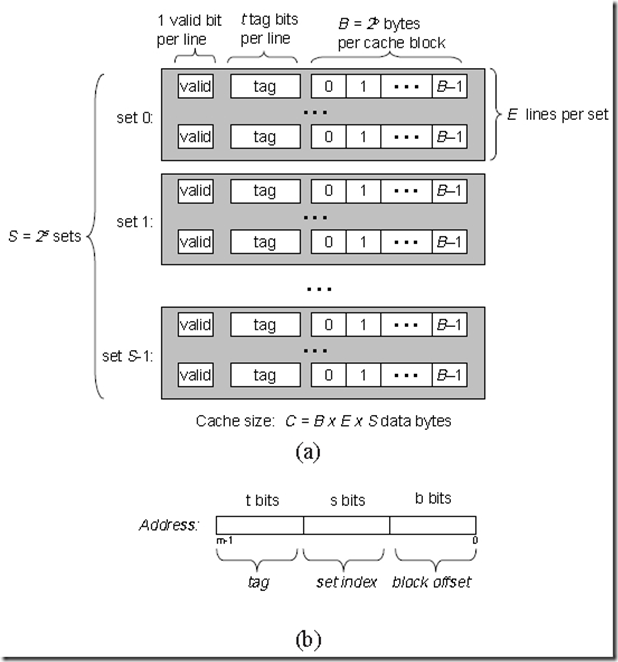
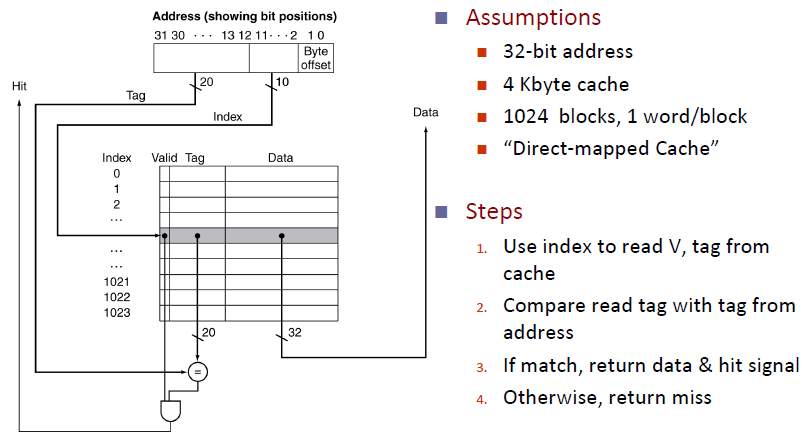
* + - * cache写时是否更新memory
        + Write-through

Memory immediately updated on each write

* + - * + Write-back

Memory updated when line replaced

每个line有一个dirty bit，line在一开始是clean的，当第一次有write的时候变成dirty，只有当需要替换dirty line的时候，才需要将数据写回memory

* + **Cache organization**
    - 
    - 
    - 
    - 
  + **Cache performance**
    - Miss rate：在cache中没有命中的概率
    - Hit rate=1-miss rate
    - Average memory access time（AMAT）== hit time + miss rate \* miss penalty
    - CPUtime = CycleTime x (CyclesExec + CyclesMemoryStall)
    - CyclesMemoryStall = CacheMisses x (MissLatencyTotal – MissLatencyOverlapped)
  + Cache miss的类型
    - Compulsory: have to have these：每个block第一次被访问的时候miss
    - Capacity: due to limited cache capacity，如果cache size是无穷大，就不会
    - Conflict: due to limited associativity，如果是fully associative就不会
  + **如何提高cache表现**
    - Reduce miss penalty
      * 使用multilevel caches
      * Early restart & critical word first
      * Early restart（提前重启）
        + 当需要到的word到时，先让处理器使用它，再继续block transfer以填满cache line
      * critical word first（关键字优先）
        + 先transfer loaded word，再transfer剩下的block
        + 使用wrap-around得到整个block
        + 再使用“Early restart”让处理器快点得到
      * Increase Load Miss Priority
        + Load有自己独立的指令；如果同时有load和store，但是load miss，让load miss go first，所以也需要一个write buffer记住store
      * Merging Write Buffer
        + 如果多个write在同个block中miss，把他们合并到write buffer
        + 用block write 替代很多的small writes
      * Victim cache
        + 最近被kicked-out的block被保存在一个小的cache（victim cache）里面；如果在cache中没有找到block，可以从victim cache中找回来

\*很多经常被用到的block会想去到same entry里，叫做“conflict miss“（一般发生在direct-mapped和n-way set-association cache）

* + - Reduce miss rate
      * Larger block（考虑空间局部性）
      * Larger caches（但会导致hit latency变长）
      * Higher Associativity（降低conflict miss，但会导致hit latency变长）
      * Way Prediction
        + 加速在Set-associative cache中查找block

先预测数据可能在哪个N-way Set，fast access as direct-mapped cache，如果预测错误，把他作为set-assoc cache再次访问

* + - * Pseudo Associative Caches
        + 与way prediction相似，先使用direct mapped cache，If miss on “primary” entry, try another entry
      * Compiler optimizations
        + Loop interchange
        + Blocking
    - Reduce hit time
      * Small & Simple Caches are faster
      * Avoid address translation on cache hits（需要把虚拟地址（软件使用）转化为物理地址（硬件使用）
      * Pipelined Caches🡪提高bandwidth，让L1有很高的f
      * Trace cache（对于instruction cache）
    - Hiding miss latency：Cache缺失的时候，cache依然可以为其他指令服务
      * Prefetching：预测哪个要先被拿
        + 如果预测的access没有发生，叫做cache pollution
        + 为了防止cache pollution，我们安排一个prefetch buffers（small separate）

When we do access it, put data in cache

If we don’t access it, cache not polluted

* + - * Simple sequential prefetch
        + 有一个cache miss，取两(N)个sequential memory blocks (“Adjacent Cache Line Prefetch” or “Spatial Prefetch”)
        + Stream prefetching: prefetching的连续版本

Stream buffer can fit N cache lines

On a miss, start fetching N sequential cache lines

On a stream buffer hit: Move cache line to cache, start fetching line (N+1)

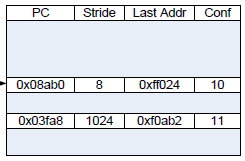
* + - * Strided Prefetch 跳预取，检测和预取跨步访问
        + 使用PC-based table进行stride detected
        + Stride detection

Remember the last address used for this PC

Compare to currently used address for this PC

* + - * + 使用2 bit saturating counter

如果stride正确则增加，如果错误则减少



* + - * Software prefetching（预取和运行工作同步进行）
        + 两种类型Register prefetch和cache prefetch
        + 每个类型可以是faulting或non-faulting（是否会create exception）
        + Faulting register prefetch是binding（normal load, address must be OK, uses register）
        + Not Faulting register prefetch是non-binding\*

如果地址错误🡪变为空指令，不影响register state

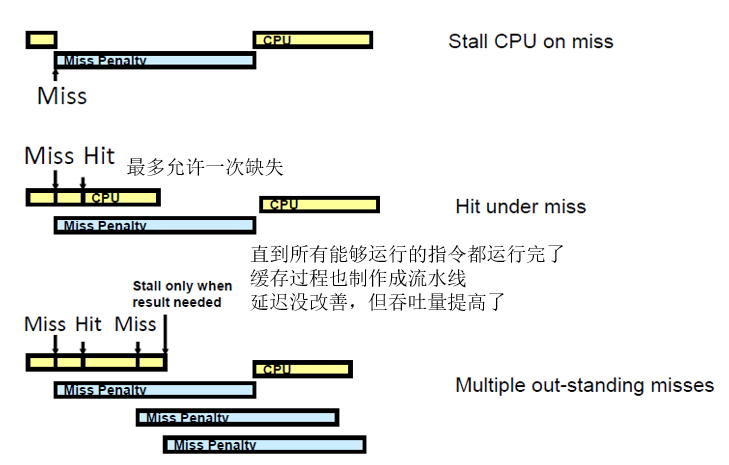
Has more overhead (load still there),

ISA change (prefetch instruction),

complicates cache (prefetches and loads different)

* + - * Non-blocking caches
        + A blocking cache services一次一个访问，当miss时，其他访问被阻塞
        + Non-blocking caches消除了这个限制，当miss时，可以处理其他请求

内存系统允许pending多个request，即当有多个进行中的miss情况发生时，依然允许hit和miss 🡪 Hit under（multiple）miss / miss under miss



# 06 Cache coherence

* Cache coherence属性
  + 程序顺序保护（preserves program order）：无共享情况下，每个处理器都像单处理器
  + 每个处理器最终都可以看到对某个address的写操作
  + 保留因果关系（前后处理顺序）
* cache coherence的硬件方法
  + Shared caches
    - 简单的执行一致性，并且不可scalable（L1缓存成为瓶颈）
  + Snooping
    - * 有broadcast network加强coherence
      * 每个cache有个block用于跟踪自己的共享状态
      * 两种方式write-update & write-invalidate
        + Write-update：

A write to shared data is broadcast to update all copies

All subsequent reads will return the new written value (property 2)

All see the writes in the order of broadcasts

One bus == one order seen by all (property 3)

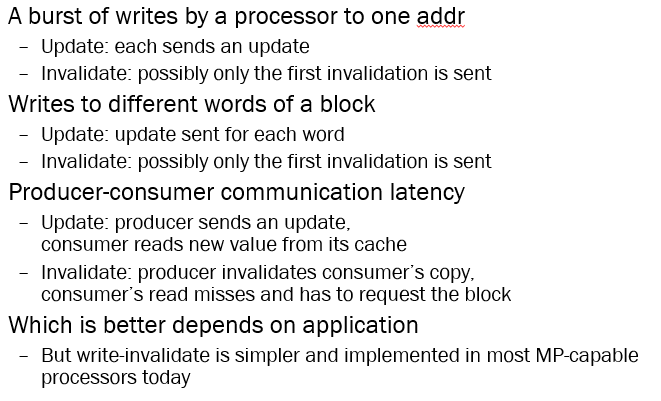
* + - * + Write-invalidate

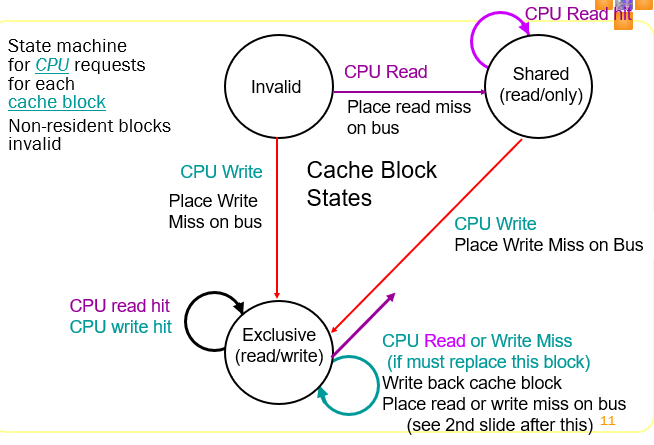
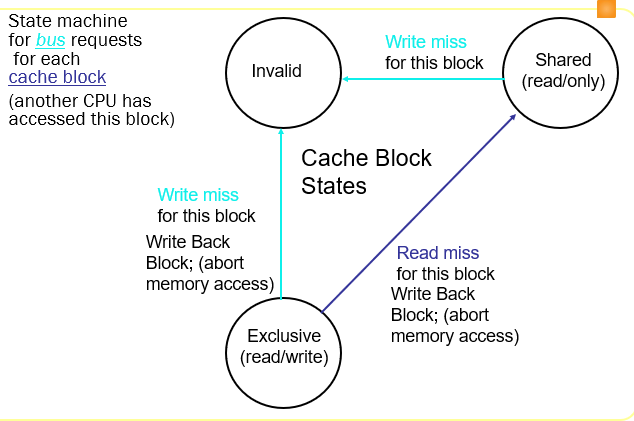
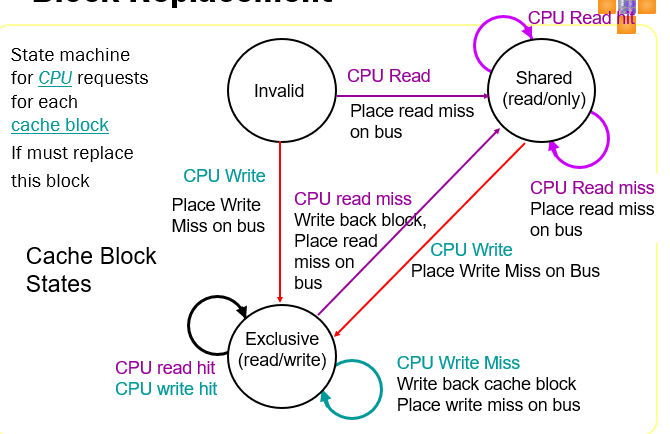
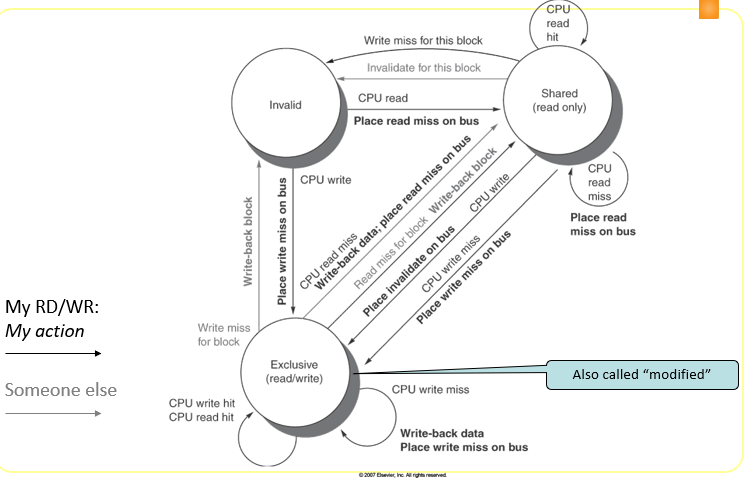
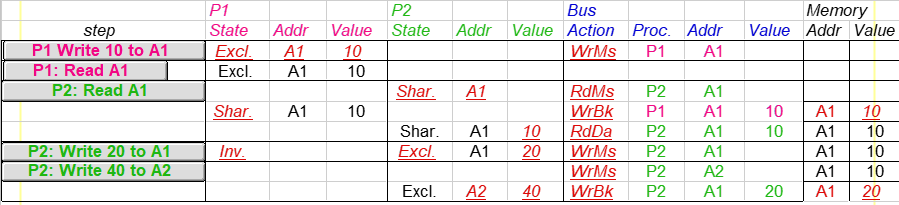
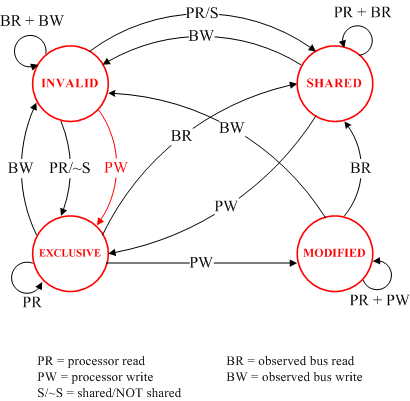
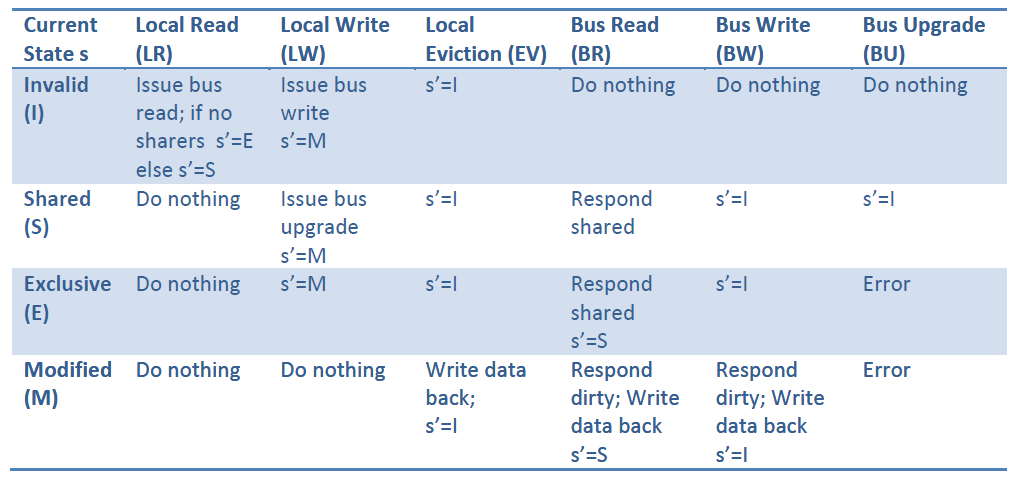
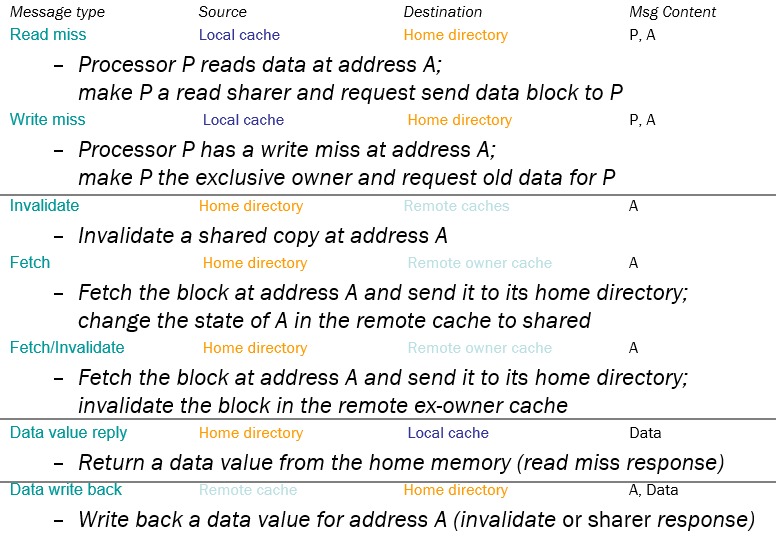
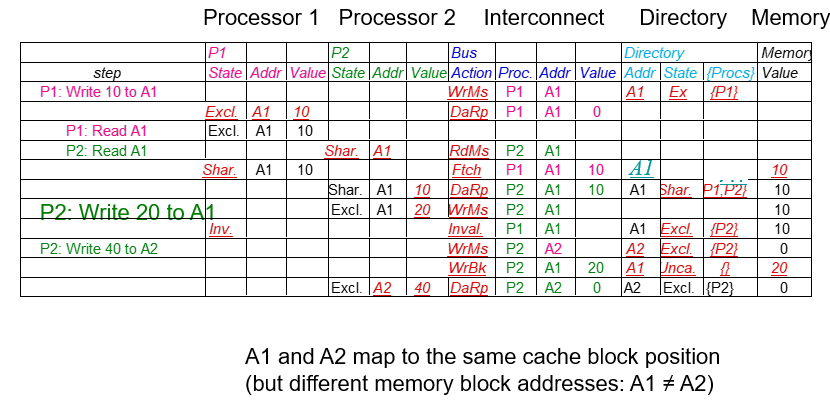
Write to shared data forces invalidation of all other cached copies

Subsequent reads miss and fetch new value (property 2)

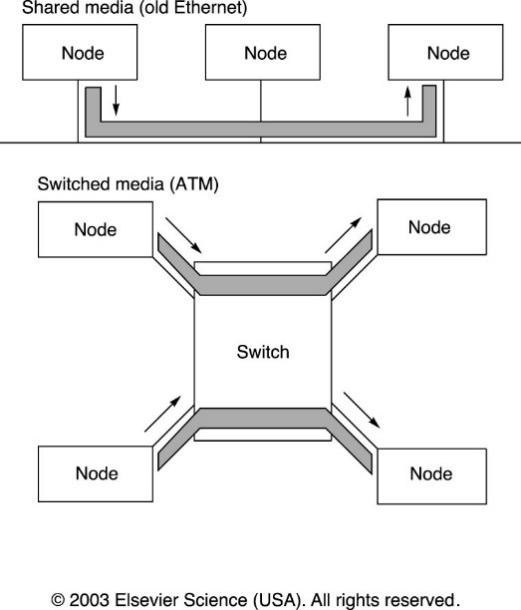
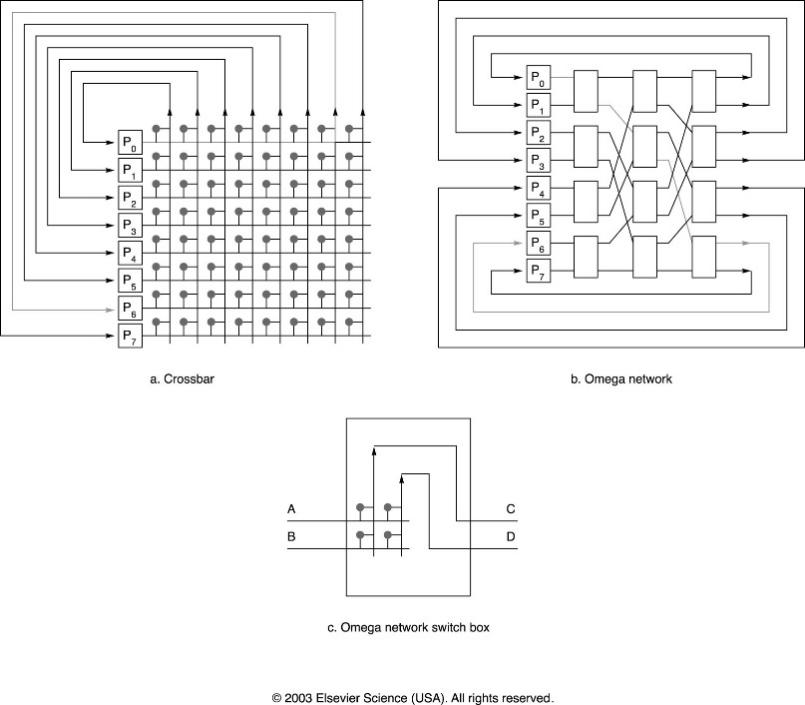
Writes ordered by invalidations on the bus (property 3)

* + - * + 两种方式的对比

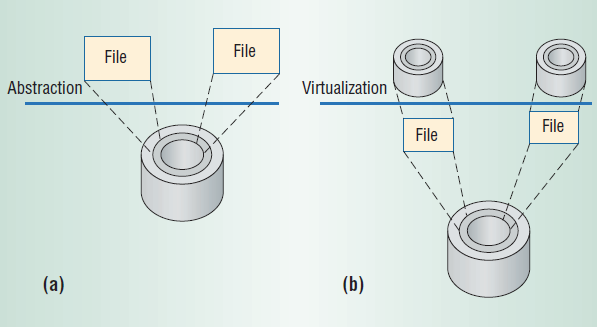
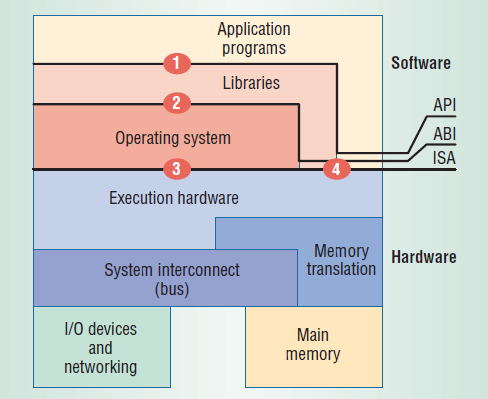
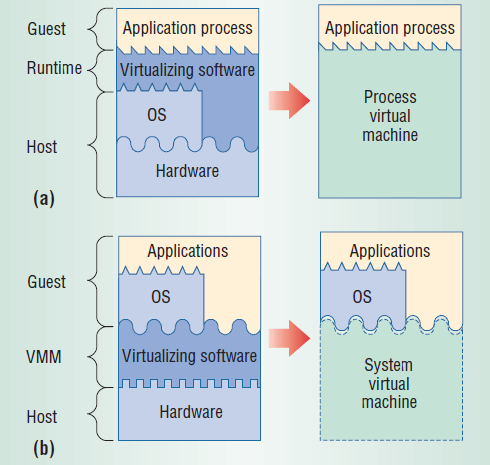
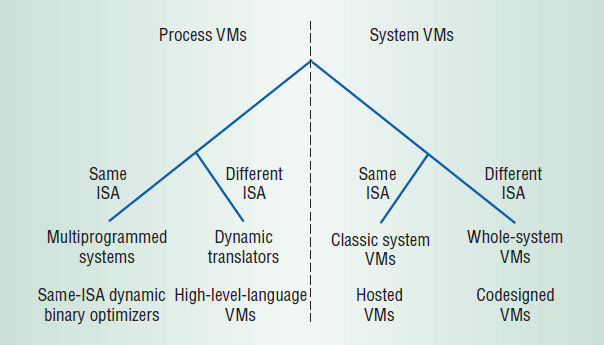
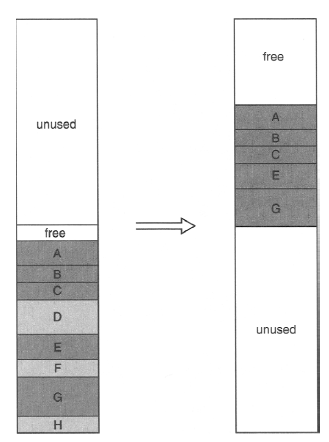
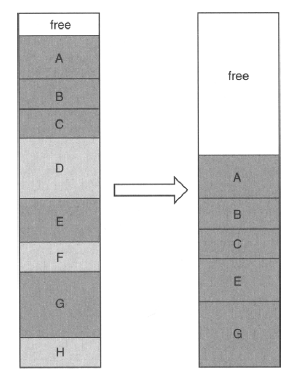
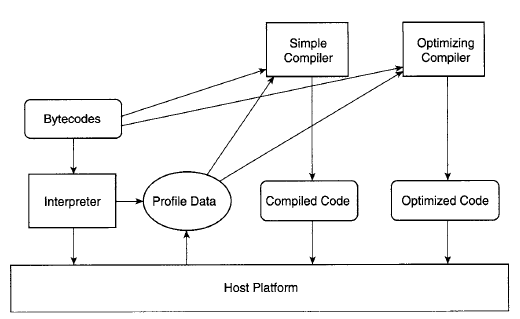
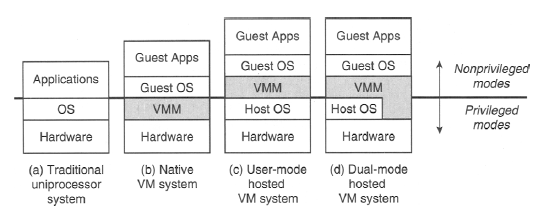
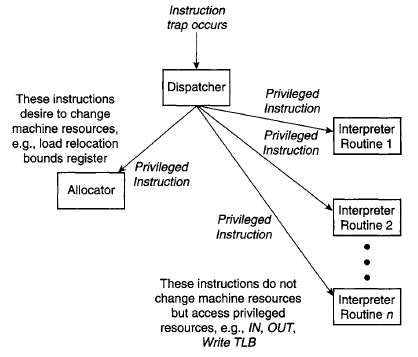
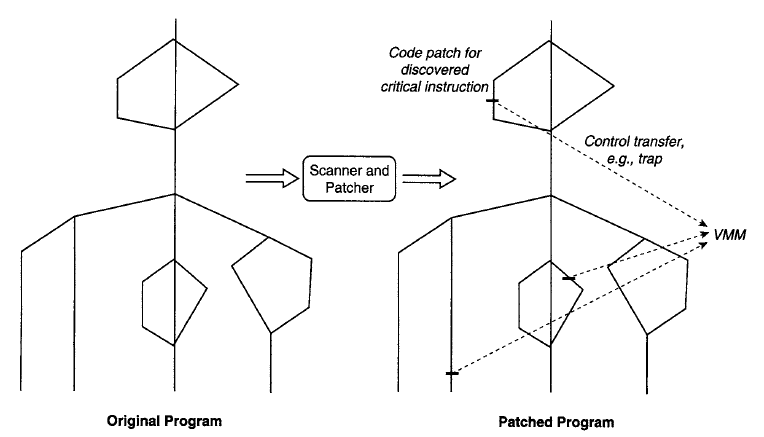


* + Directory
    - Can enforce coherence even with a point-to-point network
    - A block has just one place where its full sharing state is kept
* （snoopy）MSI Protocol
  + 三种state：invalid、shared（clean）、modified（dirty）
    - Invalid: B（block）is not cached in C(cache)
    - Modified: B is dirty in C（b在c中是最新的，如果要替换b时需要在内存更新b）
    - Shared：B is clean in C（如果要替换b时不需要在内存更新b）
      * To write, must send **an upgrade request** to the bus
  + Write-Back State Machine – CPU（作为向总线发出请求的主体）
    - 
  + Write-Back State Machine - Bus Requests（对于总线发出的请求，其他cache状态变化）
    - 情景：P1修改了A（A🡪M），并向总线发出write miss，P2修改了A（A🡪I）
    - 
  + Block-Replacement
    - 
  + 总图
    - 
    - 在这个protocol中，一个valid的cache block（最新的）只有两个状态“exclusive（modify）”和“shared”；在最最开始的时候，所有的cache block状态都是“invalid。，
      * 当CPU1第一次写入block并向bus发出“write miss”使得其他副本block都变为“invalid”，其自己的状态变为“exclusive（modify）”；如果CPU2是第一次（新）read某个block，向bus中发出read miss，并认为该block状态“shared”（但该block中可能没有value）；然后bus会write-back给memory更新value，并且CPU1中的dirty cache block（状态为exclusive（modify））会被更改为“shared”；再然后CPU2会准备从memory中read data，（向bus发出Read Data），得到value
    - 例子（A1、A2在不同的memory block，但在cache中是同样的location）
    - 
* cache to cache transfers情景：
  + - 如果P1有最新的数据（Memory没有），此时P2需要read，两种做法：
      * Solution 1: abort/retry
        + P1 cancels P2’s request, issues a write back
        + P2 later retries RdReq and gets data from memory
        + Too slow (two memory latencies to move data from P1 to P2)
      * Solution 2: intervention（如果某个cache的data处于M）
        + P1 indicates it will supply the data (“intervention” bus signal)
        + Memory sees that, does not supply the data, and waits for P1’s data
        + P1 starts sending the data on the bus, memory is updated
        + P2 snoops the transfer during the write-back and gets the block
      * 如果cache要求的data处在S状态
        + 1：让memory提供data
        + 2：谁赢了仲裁，谁就提供数据
        + 3：有一个类似S的独立状态，表明可能有其他的block处于S状态，但如果有CPU需要数据，我们应该提供它
* （snoopy）MESI Protocol
  + 总图
    - 
  + 新状态exclusive
    - Data是clean，但有唯一的（最新数据的copy）copy（除memory）
    - 好处：减少仅存在一个cache中的block被写入时造成的交通堵塞
  + 规则：
    - 只要本地写入，s🡪M
    - 除了s=M时，bus write🡪I
    - S=M时，数据被bus要求更改，都需要write back
    - 
  + 情景：Detecting Other Sharers（如果P1想要读B，向总线发送读请求，P1要怎么知道B是否在S或E状态）
    - Solution（有bus signal “share”）
      * 该信号总是low
      * 如果P2检测到P1请求，将信号变high，如果信号high🡪（为S），信号low🡪（为E）
* Directory-based coherence 用于分布式共享存储器（所有cpu都可以看到存储器）
  + 对于每个local memory block，local directory都有一个entry
    - Directory entry显示谁有自己block的cached copy & 那个block的state
    - 每个entry有1 dirty（modify） bit（dirty cached copy）🡪D bit
    - A Presence vector（每个node 1 bit）🡪告诉哪个node有cached copy
  + 情景：Read miss (Processor Pk has a read miss on block B, sends request to home node of the block)
    - Directory controller
      * Finds entry for B, checks D bit
      * If D=0: Read memory and send data back, set P[k]
      * If D=1: Request block from processor whose P bit is 1; When block arrives, update memory, clear D bit, send block to Pk and set P[k]
  + Directory operation
    - Network controller连接着每个bus，对remote cache和memory都有自己的代理
      * Requests for remote addresses forwarded to home, responses from home placed on the bus
      * Requests from home placed on the bus, cache responses sent back to home node
    - 每个cache仍具有自己的一致性状态，directory可以避免broadcast并要求按照order访问每个location
    - 所有的miss被发送回block的home node
    - Directory respond with data
  + Directory protocol messages
    - 
  + 例子
  + 
* Shared Memory Performance
  + 两种coherence misses
    - True sharing：Different processors access the same data
    - False sharing: Different processors access different data, but they happen to be in the same block
* Synchronization(for shared memory MPs)
  + 为什么需要同步性：需要知道什么时候能够安全的让不同的处理器访问shared data
  + 同步性存在问题：
    - 需要不会被打断的指令来获取和更新内存(原子操作)
      * 原子交换：将寄存器中的值与内存中的值交换；如果0表示同步变量是free的，如果1表示同步变量被锁住并且不可得
      * Test-and-set：设置一个锁值(=>1)，并测试prior lock value，查看PU是否可以控制锁定的数据(或代码)
      * Fetch-and-increment：返回该内存位置的前一个值&在内存中原子地增加它；用于为任务队列中的作业提供PU唯一指针
      * Load linked (or “load locked”) + store conditional
        + Load linked (*ll*) 返回初始值并设置一个“addr. not stored” flag；如果成功，存储条件返回1到“新值reg”，否则返回0。
    - 简单的用户级同步操作;
      * Spin locks:处理器不断尝试获取锁，并绕着一个循环，试图找到没有锁的(=0)
    - 对于大规模的MPs，同步可能是一个瓶颈;需要技术减少多个处理器争用同一个锁引起的系统开销和同步的延迟

**07 Interconnect（多个计算机相互连接）**

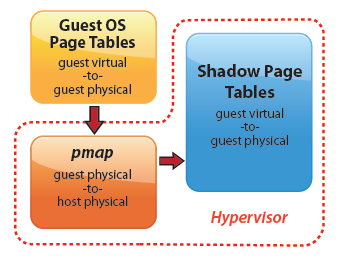
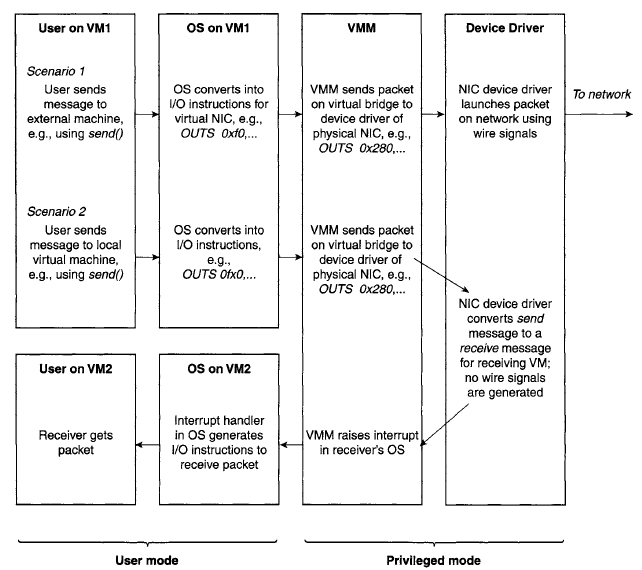
* Interconnection network
  + Shared medium
  + Switched
  + 
* Shared （media）medium network
  + 需要仲裁哪个node可以发言（发送消息）
  + 仲裁可以是centralized或distributed
    - Centralized不太适用于network，需要使用专用仲裁设备（Special arbiter device）或必须使用仲裁，并且仲裁者不能太远，否则表现很差
    - Distributed仲裁：检查media是否已经使用（carrier sensing）：如果media没有使用，开始发送；如果另一个已经发送（检测到冲突），随机等待一段时间后再尝试
      * 指数回退以避免在冲突上浪费带宽
* Switched network
  + 需要使用switch，包括switching overhead
  + 没有时间浪费在仲裁和冲突中
  + 如果使用不同的link，可以同时进行多个传输
  + 对于端到端的circuit switching，为其连接预留link（例如电话线路）；对于每个packet都有独立路径的packet switching，仅在数据传输时使用的link(例如互联网协议)
* Routing
  + 在shared medium network，有一条不重要的routing（broadcast）
  + 在switched network
    - 分类
      * Source-based (source specifies route)
      * Virtual circuits (end-to-end route created)：当连接建立，创建路径，switch在该路径上转发packet
      * Destination-based (source specifies destination)：switch route包去此目的地
    - 另一种分类
      * Deterministic (一条从起点到终点的路径)
      * Adaptive (different routes can be used)
  + 对于switch的routing method
    - Store-and-Forward: 交换机接收整个数据包（存储），然后转发，如果转发发生错误，交换机可以重新发送
    - Wormhole routing：数据包由flit组成(每个有一些字节数)。第一个flit header有目的地址，交换器得到此header，决定向何处转发packet，其他的flit在到达时也跟此一样向前转发，如果在此过程中发生错误，发送方必须重新发送，因为没有交换机拥有重新发送的整个数据包。
  + 对于routing的网络延迟
    - Switch delay：交换机中从incoming到outgoing link的时间
    - Switches：沿路上switch的数量
    - Transfer time：Time to send the packet through a link。数据从发送到对方完全接受到所有数据的总时间，包括发送时间，存储转发时间，传播时间。
      * vs 传播时间：数据在收，发两端的线路间传输所用得时间。
    - Store-and-Forward end-to-end transfer time： （不知道怎么算出来得）
      * (Switches\*SwitchDelay)+(TransferTime\*(Switches+1))
    - Wormhole or Cut-Through end-to-end transfer time：
      * (Switches\*SwitchDelay) + TransferTime
  + （switch技术）我们想要的switch是：有很多输入输出link，内部低冲突，短switch延迟
    - Crossbar：很低的switching延迟，没有内部冲突，但随着link的上升复杂度上升
    - Omega network：switch使用更小的crossbar有更多的port，每个link的复杂度小，但高延迟和高冲突
    - 
    - 也可以是mesh，torus或hypercube
* Interconnects for Datacenters
  + Multi-tier Model（比如web server、application server、database server层）
  + Server cluster model
  + Google的tree like network

**Virtual machine**

* 抽象和虚拟化
  + 抽象abstraction： 抽象的目的是把细节隐藏，使其变得简单
  + 虚拟化virtualization：虚拟化将一个特殊的接口映射到底层（或不同的、真实的系统）的一个接口
  + 
* 体系架构接口
  + ISA：包括system ISA和user ISA
  + ABI - application binary interface：不包括系统指令，通过系统调用接口调用OS
  + API：用户指令，HLL（高级语言接口）
  + 
* VM定义：a software implemented abstraction of the underlying hardware, which is presented to the application layer of the system.（底层硬件的一个软件实现抽象，并被系统的应用层呈现）
  + Process VM：VMM模拟user level instruction和任意的系统或库调用。在主机操作系统内作为普通应用程序运行,并支持单个进程.它是在该进程启动时创建的,并在退出时销毁.其目的是**提供独立于平台的编程环境**,抽象出底层硬件或操作系统的细节,并允许程序在任何平台上以相同的方式执行.
  + System VM：VMM模拟整个ISA，包括用户和系统指令。其提供了一个完整的系统平台,支持完整的操作系统(OS)的执行, 比如virtualbox。
  + 
* VM的分类
  + 
* 不同层次的VM
  + 应用层application level：（jvm）
  + Library level
  + Operation system level操作系统层次的虚拟机：动态复制操作系统环境，此类虚拟机能够创建多个虚拟运行容易，所以对于运行在每个容器上的软件来说，这类虚拟机提供了完整的操作系统运行环境，而它与上层软件的接口即为系统调用接口
  + Hardware abstraction level硬件抽象层的虚拟机：（vmware）对于上层软件（客户操作系统）而言，该类虚拟机构造了一个完整的计算机硬件系统，此虚拟机与客户操作系统的接口是处理器指令
  + ISA level：Crusoe
* 全虚拟full virtualization
  + 向VM提供与底层物理硬件相同的功能，所以guest os（运行在虚拟环境中的os）不需要修改任何东西就可以在vm中运行，但这可能会导致一些性能降低
* 部分虚拟para virtualization
  + 向VM提供与实际底层物理硬件相似的虚拟的硬件抽象，但不是完全一样，所以guest os（运行在虚拟环境中的os）需要修改一些部分去配合VMM，使得自己能够在VM中运行。通过这样，它们会有一些更低的开销，以得到更好的性能
* Instruction set emulation指令集模拟
  + Guest的source isa会被host的target isa模拟（针对不同的isa）
  + 指令翻译过程：取到source指令，进行分析，执行操作
  + 二进制翻译：将source指令的block翻译到target上
  + Depending on the repeat times of source instruction block execution
* 12-18、20、21看不懂
* 指令块
  + 动态指令块（Dynamic Instruction Block）由程序的实际流程决定，它从分支或跳转后执行的指令开始，跟随顺序指令流，并以下一个分支或跳转结束
* High level VM
  + ISA通常只为用户模式程序定义，而不是为真正的硬件处理器设计的：jvm
  + Security and Protection: Sandbox
  + Robustness: Object Oriented, Garbage Collection
  + Networking: dynamic linking
  + Performance: JIT
* Jvm
  + 23-24看不懂
* 垃圾回收
  + 垃圾：objects that are no longer accessible
  + Mark-and-sweep collectors：从根引用开始，跟踪所有可到达的对象，将每个对象标记为reached。未标记的对象被确定为垃圾，可以reused
  + Compacting and Copying Collectors：将live objects移动到堆的底部（顶部）；在查找live object时，将其复制到相邻位置的未使用区域
    - 
  + Generational Collectors：Objects 出生在nursery heap中。如果它在nursery,中保存了一定时间的collection，它将被移动到 tenured subheap.
    - Tenured终生 heap, infrequently GC 不容易被回收
    - Nursery heap, frequently GC
  + Concurrent Collectors：While the collection, the application is still running，当标记对象中的引用被更新时，指向的对象也被标记
* 模拟器：
  + 从interpretation开始，当达到使用阈值时，采用最小优化方法进行编译；根据使用级别，选定的代码部分利用hot method进一步优化
  + 
* System VM：
  + 
  + 特权指令：Traps in user mode and does not trap in system mode。系统中有一些操作和管理关键系统资源的指令，这些指令只有在最高特权级上能够正确运行。如果在非最高特权级上运行，特权指令会引发一个异常，处理器会陷入到最高特权级，交由系统软件处理了。
  + 敏感指令：控制敏感-更改资源的配置；行为敏感：取决于资源的配置.操作特权资源的指令，包括修改虚拟机的运行模式或者下面物理机的状态；读写时钟、中断等寄存器；访问存储保护系统、地址重定位系统及所有的I/O指令。
  + 对于任何传统的第三代计算机，如果该计算机的敏感指令集是特权指令的子集，则可以构造虚拟机监视器VMM
  + 虚拟化场景下，要求将GuestOS内核的特权解除，从原来的0降低到1或者3。这部分特权指令在GuestOS中发生的时候，就会产生Trap，被VMM捕获，从而由VMM完成。这就是虚拟的本质方法，特权解除和陷入模拟(Privilege deprivileging/Trap-and-Emulation)。虚拟化场景中敏感指令必须被VMM捕获并完成。对于一般 RISC 处理器，如 MIPS，PowerPC 以及 SPARC，敏感指令肯定是特权指令，但是x86 例外，但是x86绝大多数的敏感指令是特权指令，但是由部分敏感指令不是特权指令，执行这些指令的时候不会自动trap被VMM捕获
  + （architecture）
  + Patching Code
    - * + Critical Instructions: sensitive but not in system mode
        + Scan the guest code, discovery all critical instructions, replace them with a trap or a jump to the VMM
        + 
      * Patching Code Dynamically
        + Before the block execution：scan the block; replace the critical instructions with trap; replace the last branch instruction with the trap;

if all successor blocks have been patched：replace the last trap to the original branch instruction

else：replace the last branch instruction with the trap;

* + - * + Execute the block including critical instructions and last branch
        + The last trap：Check next block
      * 但是
  + Special instruction：POPF(Intel IA-32)
    - Pop Stack into Flags Register
    - Interrupt Enable Bit in Flags Register
    - In system mode, the Bit will be changed
    - In user mode, the Bit will not be changed
* Virtual box 看不懂
* Memory virtualization
  + Hypervisor（VMM）会对每台VM的guest physical地址到host physical地址建立映射（pmap）【客物🡪主物】
  + Shadow page tables包含了Guest Virtual-> guest Physical映射关系【客虚🡪客物】
  + Hypervisor（VMM）拦截所有操作硬件TLB或页表的guest instruction(Guest Virtual->Guest Physical)，Guest TLB指令用Shadow页表更新TLB
  + 当guest memory access instruction执行时，Guest Virtual Address直接映射为Global Physical Address
  + 
* Virtualization I/0
  + Dedicated Devices专用设备：中断首先由VMM处理（它专用于一个特定的VM）
  + Partitioned Devices分区设备：VMM将参数(例如，轨迹和扇区位置)转换为主机操作系统的参数
  + Spooled Devices共享设备：guest VM的请求通过虚拟设备驱动程序被VM转换为对physical device的请求
  + Spooled Devices后台打印设备：Two level spool table approach，不存在的物理设备
* Network Virtualization
  + 
* Hardware virtualization
  + 在根模式下，有4个特权级别(“rings”)，并且支持相同的指令集，还添加了几个虚拟化特定指令；在非根模式下，仍然有4个特权环和相同的指令集，但一个名为VMCS(虚拟机控制结构)的新结构现在控制CPU操作并决定某些指令的行为。