

计算机体系结构基础

胡伟武、苏孟豪

第11章：多核处理结构

- 多核处理器的发展演化
- 多核处理器的访存结构
 - 片上Cache结构
 - 存储一致性模型
 - Cache一致性协议
- 多核处理器的互连结构
- 多核处理器的同步机制
- 典型多核处理器
 - 多核龙芯3号、Intel Sandybridge、IBM CELL、NVIDIA GPU

多核处理器的发展演化

多核处理器

- 多核处理器历史
 - 美国斯坦福大学的多核处理器项目Hydra（1994年，学术界最早）
 - IBM Power4双核处理器（2001年IBM公司）
 - AMD于2005年推出第一款x86架构双核处理器；
 - Intel于2006年推出第一款酷睿2双核处理器
 - 国内2009年推出四核龙芯3A处理器
- 多核处理器与多路服务器/工作站
 - 多核处理器和早期的多路服务器/工作站在结构上没有本质不同
 - 多核处理器共享LLC、多路服务器共享内存
 - 多核处理器的核间通信比多路服务器快

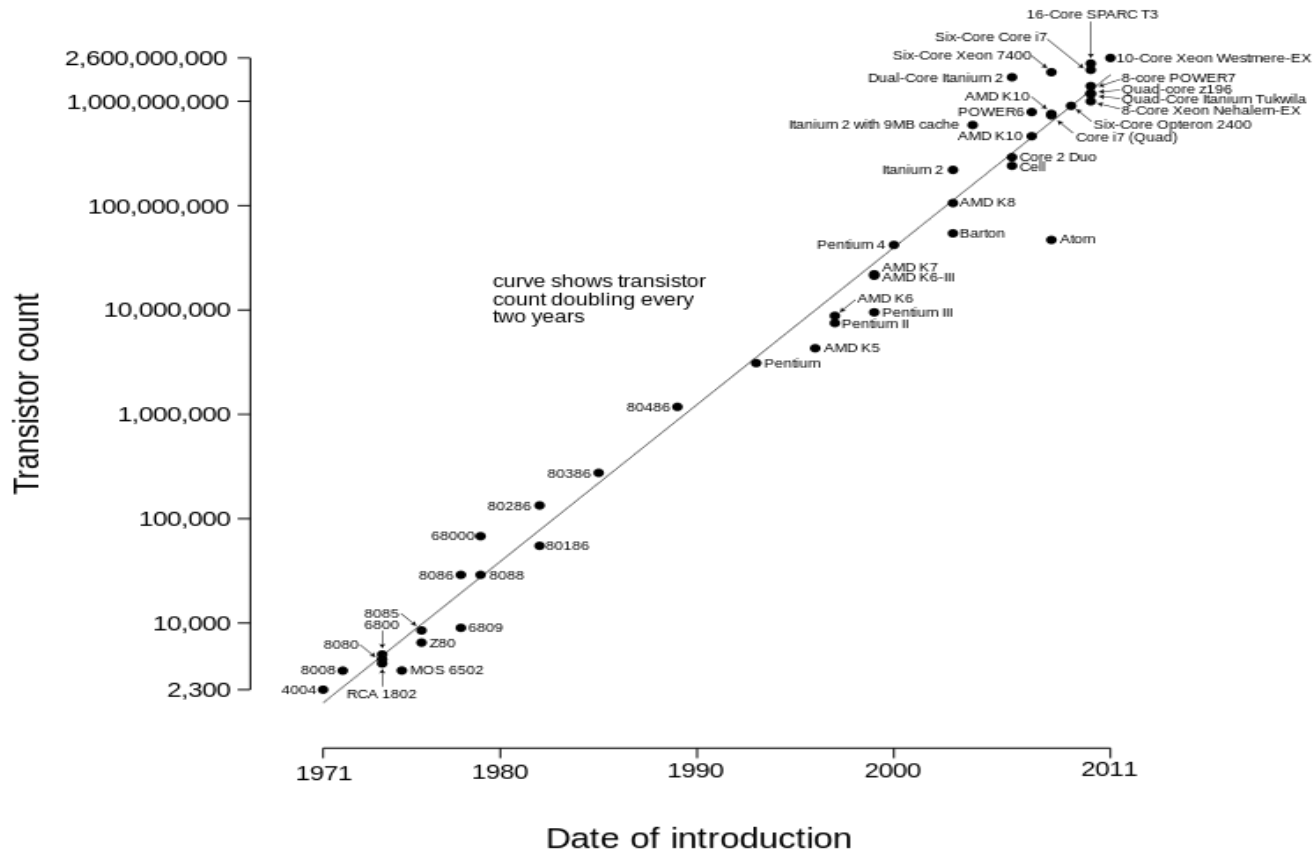
多核处理器动力一：半导体工艺

- 摩尔定律：不是自然规律，而是经验法则
 - 1965年Gordon Moore（Intel 联合创始人）提出：半导体芯片上集成的晶体管和电阻数量将每年增加一倍
 - 1975年对摩尔定律进行了修正，把“每年增加一倍”改为“每两年增加一倍”
 - 摩尔定律流行的表述：集成电路芯片上所集成的晶体管数目，每隔18个月就翻一倍
- 2020年前后晶体管尺寸难以进一步缩小已经成为共识
 - 太难造：面临量子与原子机制边界，新工艺研发投入巨大
 - 不好用：对性能油水不大，功耗密度等问题突出
 - 用不起：一次性成本太高，晶体管成本不降低，7nm是小众工艺

工艺技术支撑计算机发展

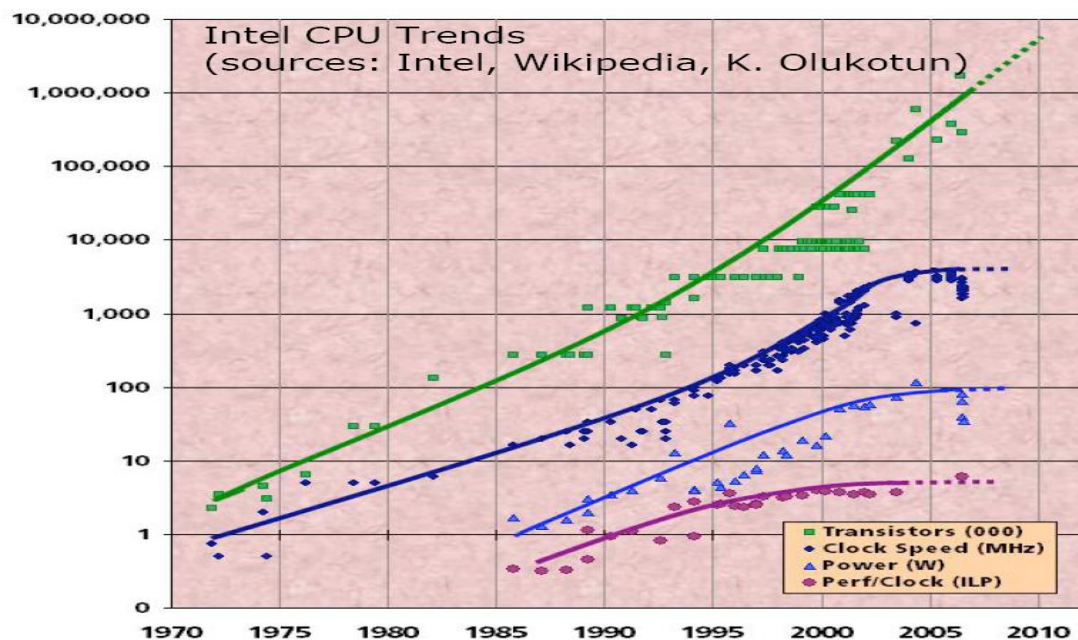
- CPU芯片上集成的晶体管数达到几十亿到上百亿量级
- 买一颗大米的钱可以买100-1000只晶体管

Microprocessor Transistor Counts 1971-2011 & Moore's Law



单核性能难以进一步提高

- 2005年以前，性能提高伴随着功耗和主频的提高
- 2005年以后，功耗墙问题突出，多核设计成为主流
 - 以Intel放弃4GHz的Pentium IV为标志，终止高主频设计
 - 2010年以前：继续通过微结构优化提高单核性能，3-5倍
 - 以Intel的Haswell主要降低功耗为标志，单核的性能难以进一步提高



多核处理器动力二：功耗墙

- 单片设计复杂的单处理器核 vs 单芯片设计多个处理器核
 - 后者的性能功耗比收益大
- 芯片功耗=动态功耗+静态功耗，其中动态功耗为主
 - 动态功耗=开关功耗+短路功耗

$$P_{\text{switch}} = \frac{1}{2} C_{\text{load}} V^2 f$$

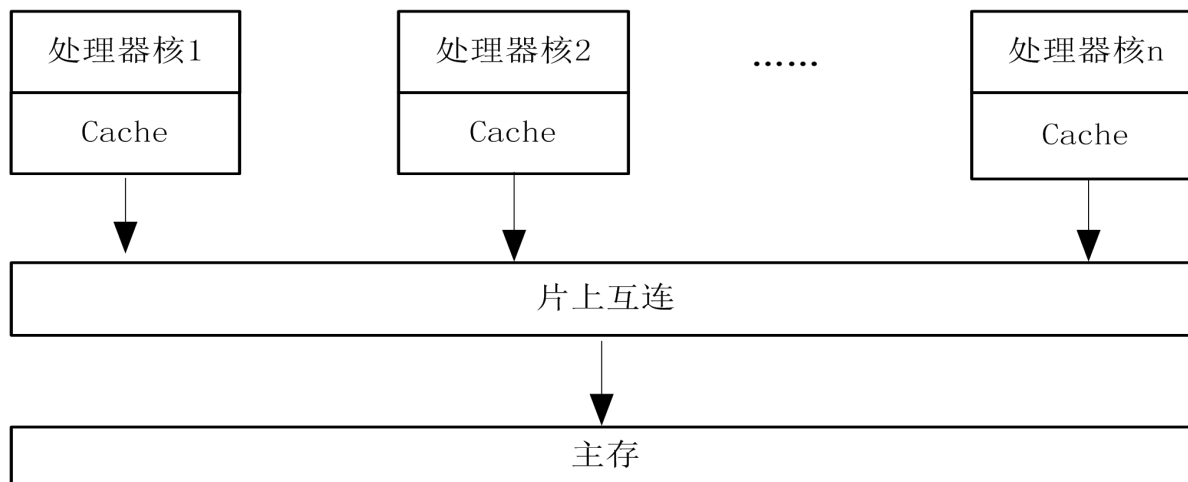
- 提高单核性能开销比多核大
 - 提高主频：进一步细分流水线、提高电压
 - 通过微结构优化提高IPC：四发射后难以提高
 - 例：采用多核结构功耗和性能成线性关系；在一定范围内通过提升电压来提升主频来，功耗与性能成三次方关系

多核处理器动力三：并行结构的发展

- **SIMD结构**
 - 早期Cray向量机，现代CPU的SIMD短向量指令（128、256、512位）
- **SMP结构：多核共享存储**
 - 早期多路服务器/工作站（DEC、SUN、SGI），现在片内多核
- **CC-NUMA结构：更多核共享存储**
 - SGI、IBM、HP高端服务器，几十到上千路共享内存服务器
 - 片内核数增加导致CC-NUMA
- **MPP或机群结构**
 - HPC，如曙光高性能机、太湖之光，主要用于科学计算
 - 云计算，机群数据库
- **GPU（上千核）采用什么结构？**

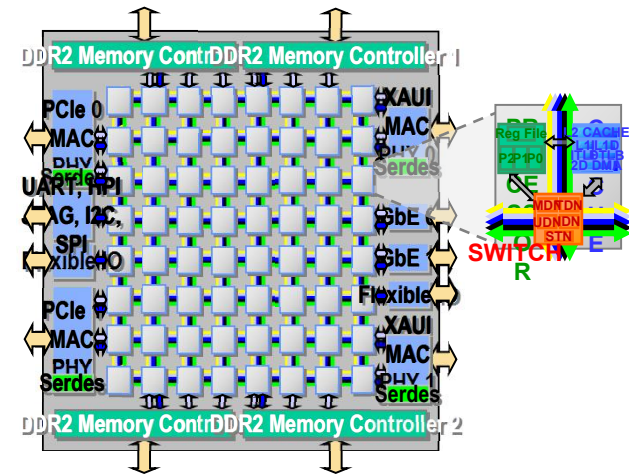
多核处理器的组成

- 多核处理器
 - 单个芯片集成多个处理器核
- 多核处理器分类
 - 应用角度：通用（面向桌面、服务器）、专用（如GPU、HPC等）
 - 处理器核结构：同构、异构多核处理器
 - 从核数角度：多核处理器、众核处理器（64核以上）



比较流行的CPU结构

- 多核 + 向量处理：商业主流结构
 - 典型：Haswell, Power8,
 - 向量的位宽：64 / 128 / 256 /
- 众核：同构基于分片（tile based）
 - 典型：GPU, Tile64
 - 处理器核的个数：64 / 128 / 512 / 1024
- 带有协处理器的异构多核
 - 典型：CPU+GPU、CPU+众核、CELL
 - 通用处理器+专用的协处理器（如GPU、智能计算、安全处理器）
- 不同的结构的存储模型很不一样



共享存储多核处理器的关键问题

- 通用多核处理器一般采用共享存储结构
- 多个处理器核发出的访存指令次序如何约定？
 - 存储一致性模型：如顺序一致性、处理器一致性等
- 多个处理器核间共享片上Cache如何组织及维护一致性？
 - Cache一致性协议：片上Cache结构及Cache一致性协议
- 多个核处理器核间如何实现通信？
 - 片上互连结构
- 多个处理器核间如何实现同步？
 - 多核同步机制：互斥锁操作（lock）、路障操作（barrier）

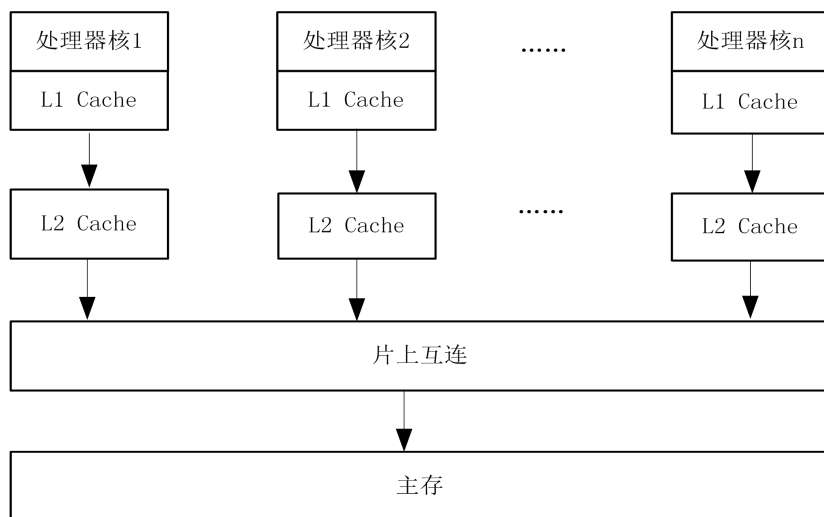
多核处理器的访存结构

通用多核处理器的片上Cache结构

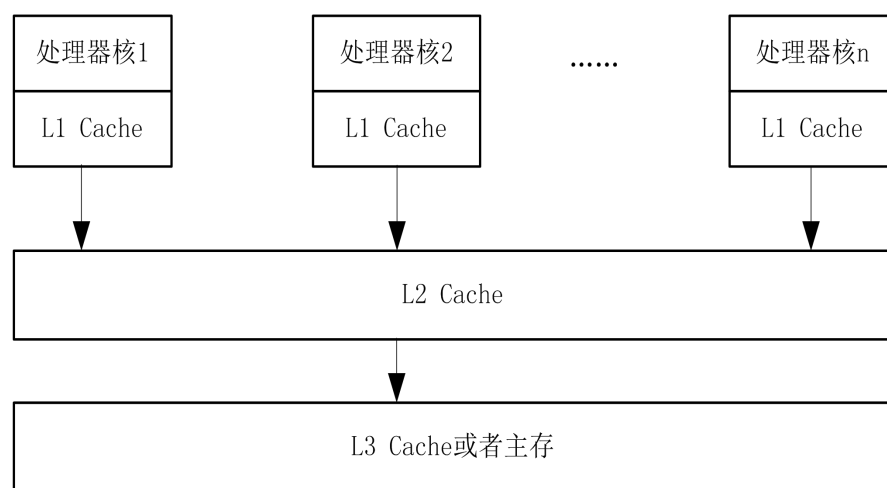
- 片内Cache种类

- 私有Cache: 处理器核包含私有Cache, 速度快、失效率高
- 片内共享Cache: 多个处理器核共享Cache, 速度稍慢, 失效率低
- 片间共享Cache: 多片共享Cache, 速度慢, 失效率高

- 片内共享LLC, 片间共享内存应是一种典型结构



(a) 私有Cache结构



(b) 共享Cache结构

主流多核处理器Cache结构

- 主流商用多核处理器一般采用共享最后一级Cache结构
 - 一级及二级Cache私有，三级Cache（最后一级Cache）共享
 - 有些处理器有片外L4 Cache

表：商用多核处理器主要参数示例



	IBM Power8	Intel Haswell	Oracle SPARC T5	龙芯3A3000
Cores per chip	12	4	16	4
Threads per core	8	2	8	1
L1 ICache per core	32KB	32KB	16KB	64KB
L1 DCache per core	64KB	32KB	16KB	64KB
L2 Cache per core	512KB	256KB	128KB	256KB
On-chip shared LLC	96MB	8MB	8MB	8MB

共享LLC结构

- **UCA (Uniform Cache Access) 结构**
 - 集中式共享结构，多个处理器核通过总线或者交叉开关连接最后一级Cache，所有处理器核对二级Cache的访问延迟相同。
 - 扩展性受限，适用于多核
- **NUCA (Non-Uniform Cache Access) 结构**
 - 分布式共享结构，每个处理器核拥有本地的二级Cache，通过可扩展的片上互连（如MESH等）访问其他处理器核的二级Cache。
 - 有较好扩展性，一般采用目录Cache一致性协议，适用于众核

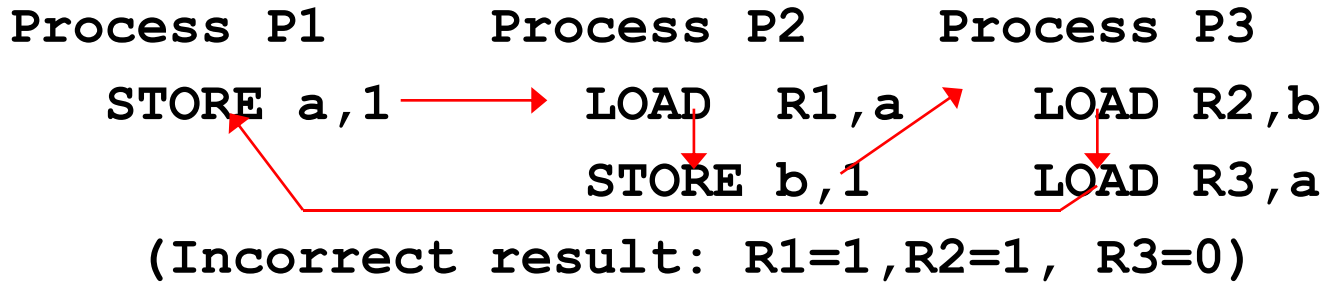
一致性问题

- 在单机系统中，只要保持程序中的数据相关性，就可以保证执行正确。在多处理机系统中，不仅要考虑单机内的数据相关，而且要考虑多机之间的数据相关
- 为了执行的正确，每个处理器核必须根据程序序来执行指令

Process P1		Process P2
STORE a, 1		STORE b, 1
LOAD R1, b		LOAD R2, a
(Incorrect Result: R1=0, R2=0)		

一致性问题 (cont.)

- 即使每个处理器核都根据程序序执行指令，仍可能导致错误



- 起因：从Write Atomic到Write Non-atomic
- 什么是写可分割系统中正确的执行？
 - 存储一致性模型和Cache一致性协议

存储一致性模型 (Memory Consistency)

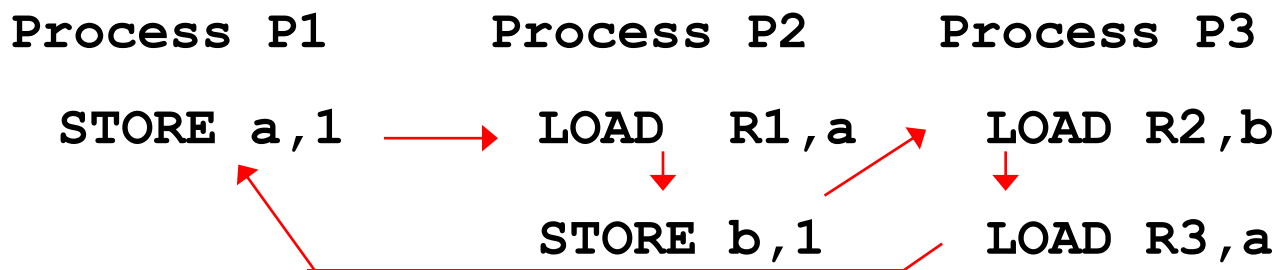
- 结构设计者与应用程序员之间的一种约定
 - 给出正确程序的标准
 - 程序员不用考虑访存次序
 - 系统设计者有更多的提高性能的空间
- 常见的存储一致性模型
 - 顺序一致性模型(Sequential Consistency)
 - 处理器一致性模型(Processor Consistency)
 - 弱一致性模型(Weak Ordering)
 - 释放一致性模型(Release Consistency)
- 从某种意义上说，存储一致性模型对共享存储系统中多处理器的访存次序作了限制，从而影响了性能

顺序一致性

- 正确性标准：符合程序员的直觉
- 正确性规范：顺序一致性
 - **Sequential consistency defines a correct execution as the one “whose result is the same as if the operations of each individual processor appear in this sequence in the order specified by the program”.**
 - 如果在多处理机环境下的一个并行执行的结果等于同一程序在单处理机多进程环境下的一个执行的结果，则此并行程序执行正确

满足顺序一致的访存事件次序

- 顺序一致的一个充分条件（**GPPO**条件）
 - 在共享存储多处理机中，若任一处理机都严格按照访存指令在进程中出现的次序执行，且在当前访存指令彻底执行完之前不能开始执行下一条访存指令，则此共享存储系统是顺序一致的
 - 一个存数操作“彻底完成”指的是它所引起的值的变化已被所有处理机所接受，
 - 一个取数操作彻底完成是指它取回的值已确定，且写此值的存数操作已“彻底完成”



处理器一致性

- 由 Goodman 提出的处理器一致性(Processor Consistency)比顺序一致性弱，处理器一致性对访存事件发生次序施加的限制：
 - 在任一取数操作 **LOAD** 允许被执行之前，所有在同一处理器中先于这一 **LOAD** 的取数操作都已完成；
 - 在任一存数操作 **STORE** 允许被执行之前，所有在同一处理器中先于这一 **STORE** 的访存操作(包括 **LOAD** 和 **STORE**)都已完成。
- 上述条件允许 **STORE** 之后的 **LOAD** 越过 **STORE** 而执行，放松了顺序一致性模型对访存次序的限制。
- 实际上是把 **Write Buffer** 变得让用户可见
 - 如：Store提交后在Write Buffer，还没有写Cache/内存，后面的Load已经从Cache取回数据，此时收到对Load访问Cache行的一个无效请求

共享存储并行程序举例-Pthread积分求 π

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4 //假设线程数目为4
int num_steps = 1000000;
double step = 0.0, sum = 0.0;
pthread_mutex_t mutex;
void *countPI(void *id) {
    int index = (int ) id;
    int start = index*(num_steps/NUM_THREADS);
    int end; double x = 0.0, y = 0.0;
    if (index == NUM_THREADS-1)
        end = num_steps;
    else
        end = start+(num_steps/NUM_THREADS);
    for (int i=start; i<end; i++) {
        x=(i+0.5)*step;
        y +=4.0/(1.0+x*x); }
    pthread_mutex_lock(&mutex);
    sum += y;
    pthread_mutex_unlock(&mutex);
}
```

```
int main() {
    int i;
    double pi;
    step = 1.0 / num_steps;
    sum = 0.0;
    pthread_t tids[NUM_THREADS];

    pthread_mutex_init(&mutex, NULL);
    for(i=0; i<NUM_THREADS; i++) {
        pthread_create(&tids[i], NULL, countPI,
        (void *) i);
    }
    for(i=0; i<NUM_THREADS; i++)
        pthread_join(tids[i], NULL);
    pthread_mutex_destroy(&mutex);
    pi = step*sum;
    printf("pi %1f\n", pi);
    return 0;
}
```

弱存储一致性(Weak Consistency)

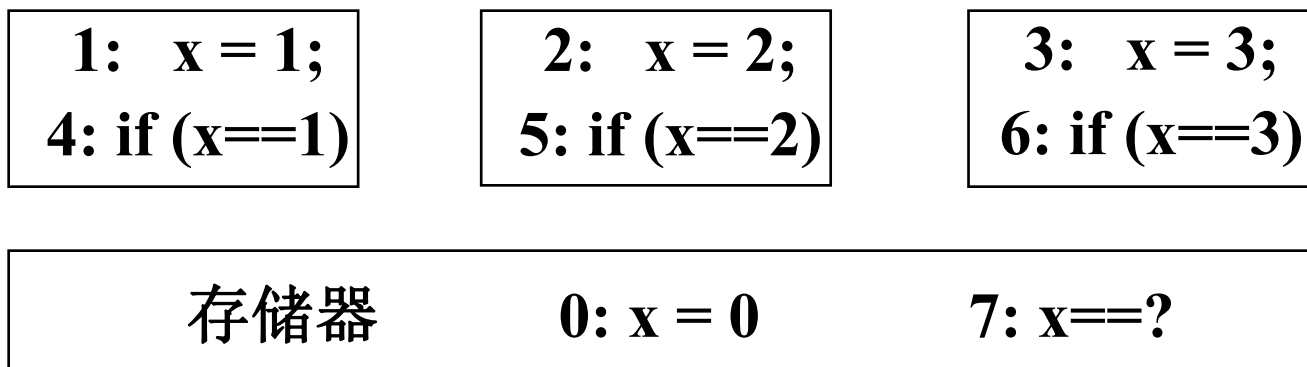
- 把同步操作和普通访存操作区分开来，只在同步点维护一致性
- 冲突访问必须用同步操作保护
 - 如果两个访存操作访问的是同一单元且其中至少有一个是存数操作, 则称这两个访存操作是冲突的
- 访存次序
 - 同步操作的执行满足顺序一致性条件;
 - 在任一普通访存操作允许被执行之前, 所有在同一处理机（或处理器核）中先于这一访存操作的同步操作都已完成;
 - 在任一同步操作允许被执行之前, 所有在同一处理机（或处理器核）中先于这一同步操作的普通访存操作都已完成。

释放一致性模型 (Release Consistency)

- 把同步操作进一步分成获取操作**ACQUIRE**和释放操作**RELEASE**
- 冲突访问必须用**REL**->**ACQ**对隔开
- 访存次序
 - 同步操作的执行满足顺序一致性条件;
 - 在任一普通访存操作允许被执行之前, 所有在同一处理器 (处理器核) 中先于这一访存操作的**ACQUIRE**操作都已完成;
 - 在任一**RELEASE**操作允许被执行之前, 所有在同一处理器 (处理器核) 中先于这一**RELEASE**的普通访存操作都已完成。

共享存储多核中的Cache一致性问题

- **Cache**在共享存储系统中的作用
 - 弥补CPU与主存间的速度差距
 - 减少访存冲突以及对互连网络带宽的需求
- **Cache一致性问题**
 - 如何保持数据在**Cache**及主存中的多个备份的一致性



Cache一致性协议

- **Cache一致性协议（Cache Coherence Protocol）**：一种把新写的值传播到其他处理器核的机制
 - 如何传播新值：Write Invalidate vs. Write Update
 - 向何处传播新值：Snoopy vs. Directory Protocol
- **Cache一致性协议**决定系统为维护一致性所做的具体动作，因而直接影响系统性能

Write-Invalidate和Write Update

- **Write-Invalidate**

- 当一个处理机更新某共享单位（如存储行或存储页）时（之前或之后），通过某种机制使该共享单位的其它备份无效，当其它处理机访问该共享单位时，访问失效，再取得有效备份

- **Write-Update**

- 当一个处理机更新某共享单位时，把更新的内容传播给所有拥有该共享单位备份的处理机

- **比较**

- **Write Update:** 重复更新已不再使用的行
- **Write Invalidate:** 假共享导致乒乓问题

侦听协议

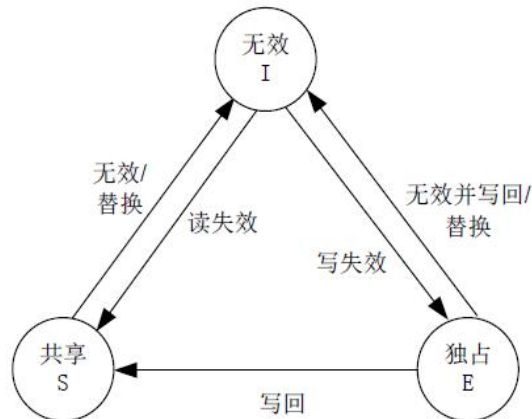
- 通过广播维护一致性
 - 写数的处理机把新写的值或所需的存储行地址广播出去
 - 其他处理机侦听广播，当广播中的内容与自己有关时，接受新值或提供数据
 - 存储器和每个处理机的Cache只维护状态信息
- 适合于总线结构的SMP系统中
 - 总线是一种廉价而有效的广播工具
- 可伸缩性有限
 - 总线是一种独占性资源
 - 总线延迟随处理机数的增加而增加：仲裁、总线长度、总线阻抗

基于目录的协议

- 每个存储行对应一个目录项
 - 记录拥有该行的一个副本的那些处理机
 - 当某个处理机写该行时,根据目录项的内容传播数据
 - 在COMA结构中, 记录该行的Owner
- 由于避免了广播, 目录协议有一定的伸缩性
- 目录需要大量存储空间, 需要动态维护
 - 位向量目录: $O(MN)$, 存储开销大
 - 有限指针目录: $O(M \log N)$, 指针溢出
 - 链目录: $O(M \log N)$, 串行更新

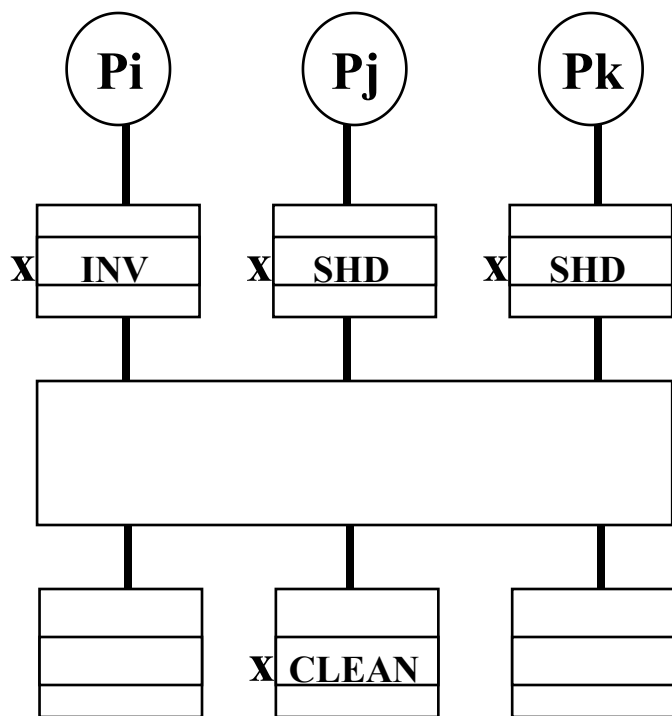
ESI协议的状态

- ESI 是指Cache 行的三种一致性状态：
 - E（Exclusive，独占）：表明对应Cache 行被当前处理器核独占，当前处理器核可以随意读写，其他处理器核如果想读写这个cache 行需要请求占有这个cache 块的处理器核释放该Cache 行
 - S（Shared，共享）：表明当前Cache 行可能被多个处理器核共享，只能读取，不能写入
 - I（Invalid，无效）：表明当前Cache 块是无效的

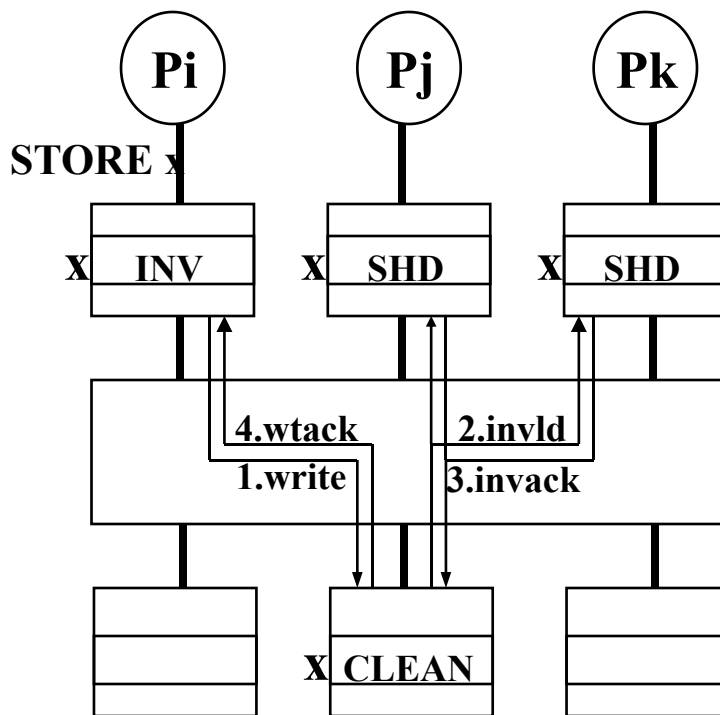


协议举例

- 基于目录，单写，写使无效

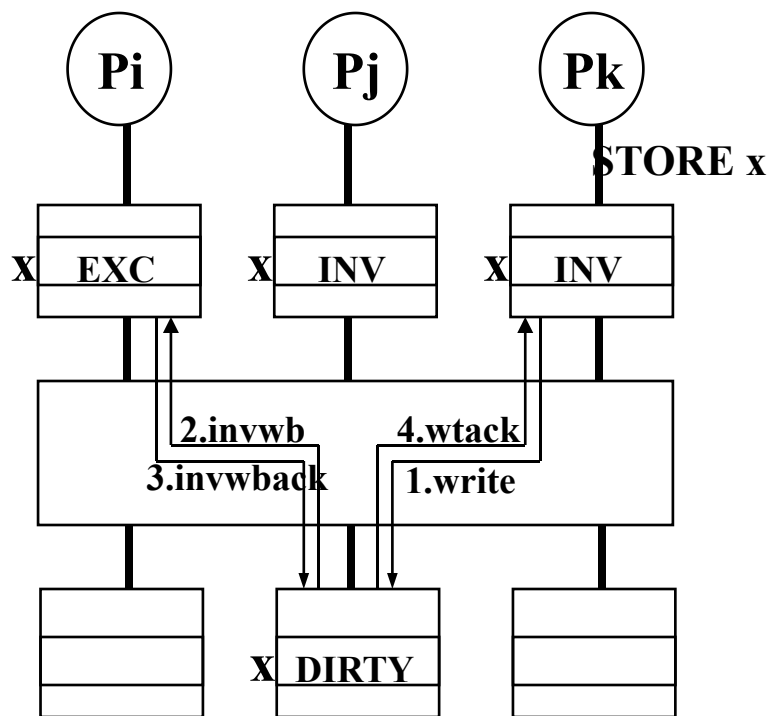


(a)初始状态

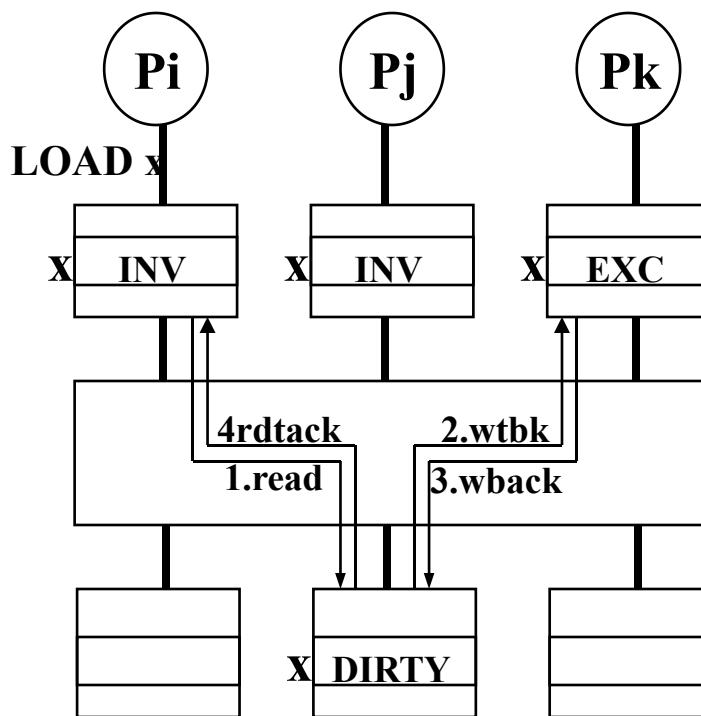


(b) P_i 发出存数操作

协议举例 (cont.)



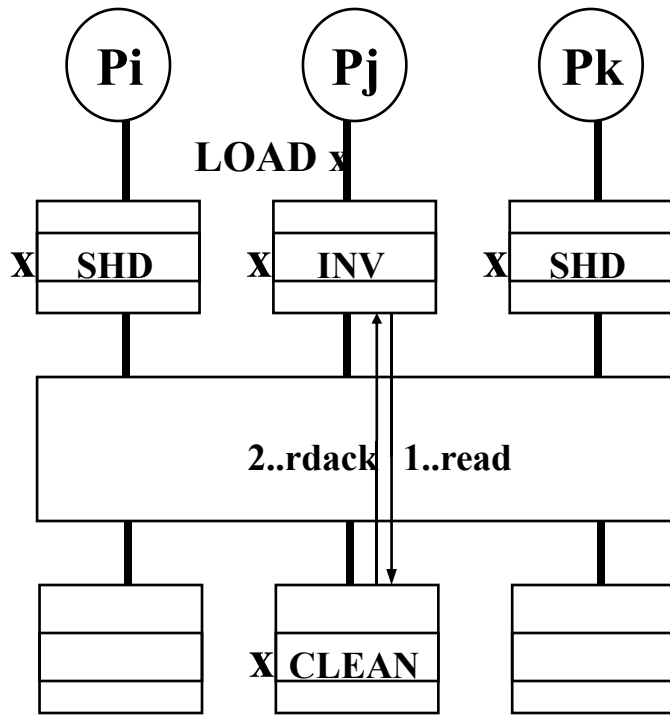
(c) P_k 发出存数操作



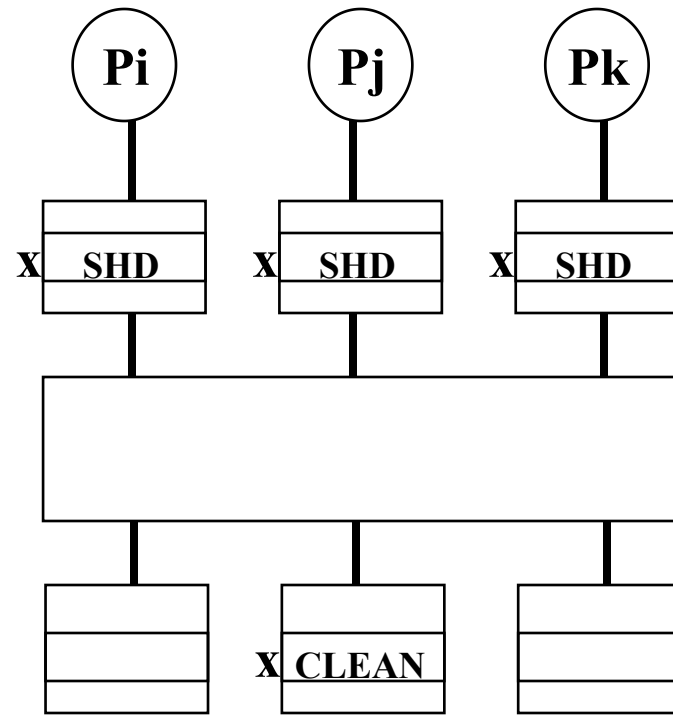
(d) P_i 发出存数操作

- 若支持SC, P_i 需等到 P_k 的“STORE x ”执行完
- 若支持RC, 无限制

协议举例 (cont.)



(e) P_j 发出取数操作



(e) 最终状态

存储一致性模型与Cache一致性协议

- 存储一致性模型对Cache一致性协议的制约作用
 - Cache一致性协议都是针对某种存储一致性模型而设计的
 - 存储一致性模型为Cache一致性协议规定了“一致性”的目标，即，什么是“一致性”
 - 如，顺序一致性模型要求对某处理机所写的值立即进行传播，在确保该值已经被所有处理机接受后才能继续其它指令的执行
 - 如，释放一致性模型允许将某处理机所写的值延迟到释放锁时进行传播

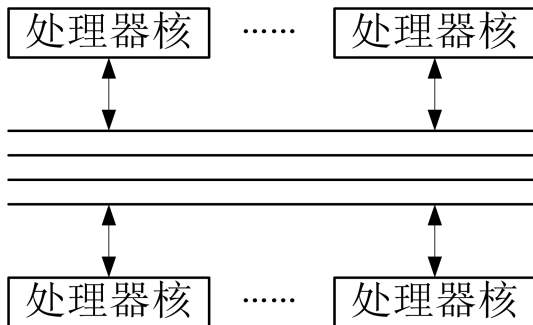
访存事件次序在微结构中的实现

- 指令在保留站中等待发射时检查并等待
 - 等到前面所有指令都提交
 - Store操作写到Cache、访存失效队列空
 - 同步操作如sync、LL、SC一般在发射时控制
- Load操作能不能越过未完成（地址未确定或尚在Store Buffer中）的Store操作执行
 - 如果Load数据返回到寄存器并被使用了，取消起来很麻烦
 - 外部来的invalidate操作要查看Store Buffer也很麻烦
- 为了性能和实现的简洁常使用弱一致性模型

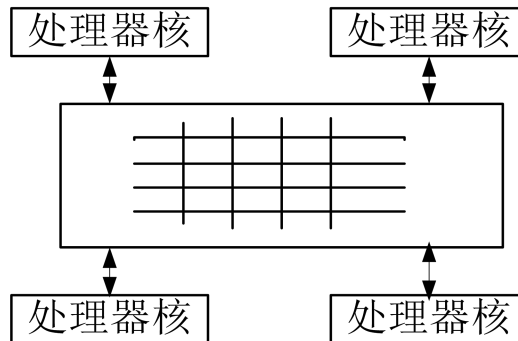
多核处理器的互连结构

多核处理器的片上互连网络

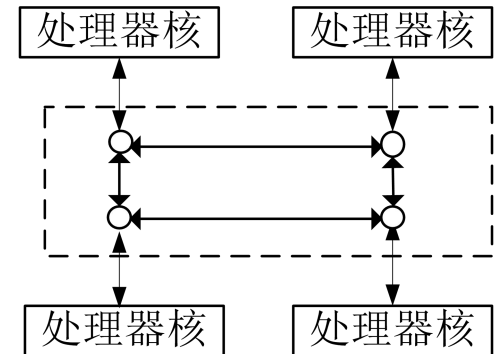
- 多核处理器通过片上互连将处理器核、Cache、内存控制器、IO接口等模块连接起来
 - 常见的片上互连结构：总线、交叉开关和片上网络



图(a) 共享总线



图(b) 交叉开关

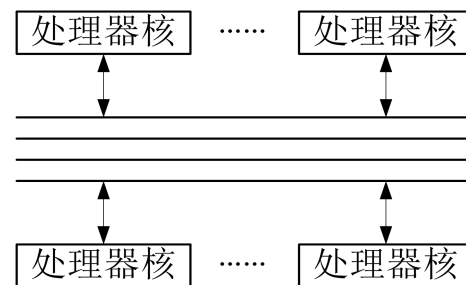


图(c) 片上网络

片上总线 and 交叉开关

• 片上总线

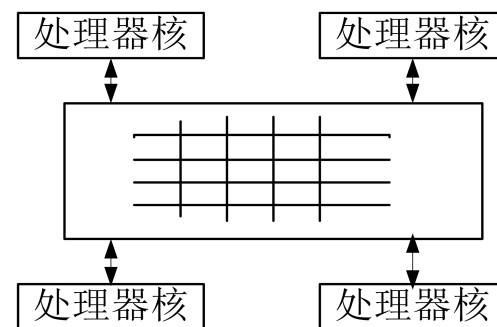
- 片上各个部件间通信的公共通路，由一组线组成
- 类型：数据总线、地址总线、控制总线等
- 优点：简单，有利于广播通信
- 缺点：独占性资源，可伸缩性差



图(a) 共享总线

• 交叉开关

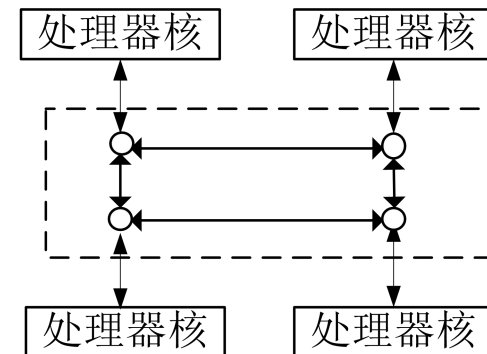
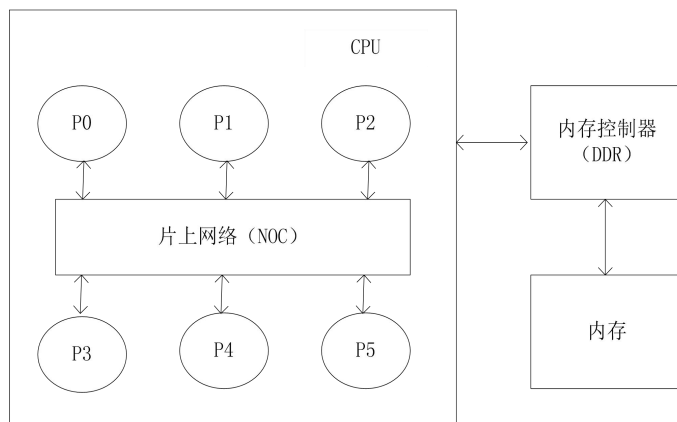
- 在一个M个输入，N个输出的交叉开关中，每个输出端口都可以接任意输入端口
- 优点：高带宽：并行通信，高带宽
- 缺点：复杂度为 $M*N$ ，可伸缩性有限
- 常用于可伸缩网络的结点内部



图(b) 交叉开关

片上网络

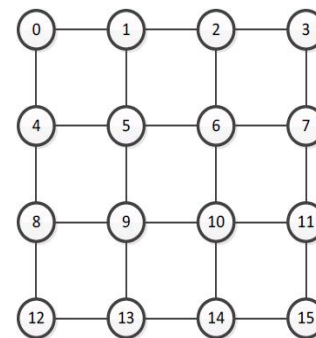
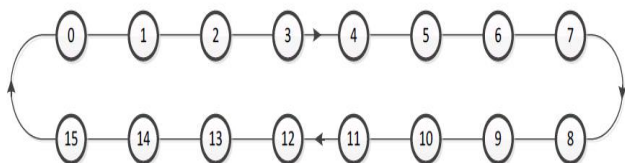
- C. Seitz和W. Dally在本世纪初首先提出了片上网络的概念
 - 处理器核抽象成节点，互连网络用来在处理器核节点间传输消息数据
 - 数据封装成数据包，通过路由器之间的分组交换和对应的存储-转发机制来实现处理器核间的通信
 - 研究内容包括：拓扑结构、路由算法、流量控制（Flow Control）、服务质量



图(c) 片上网络

片上网络—拓扑结构

- 片上网络由节点和传输信道的集合构成，片上网络的拓扑指网络中节点和信道的排列方式
- 环（Ring）、网格（Mesh）结构为最常见两种片上网络拓扑结构
 - IBM CELL、Intel处理器采用环结构
 - Tiler处理器采用MESH结构

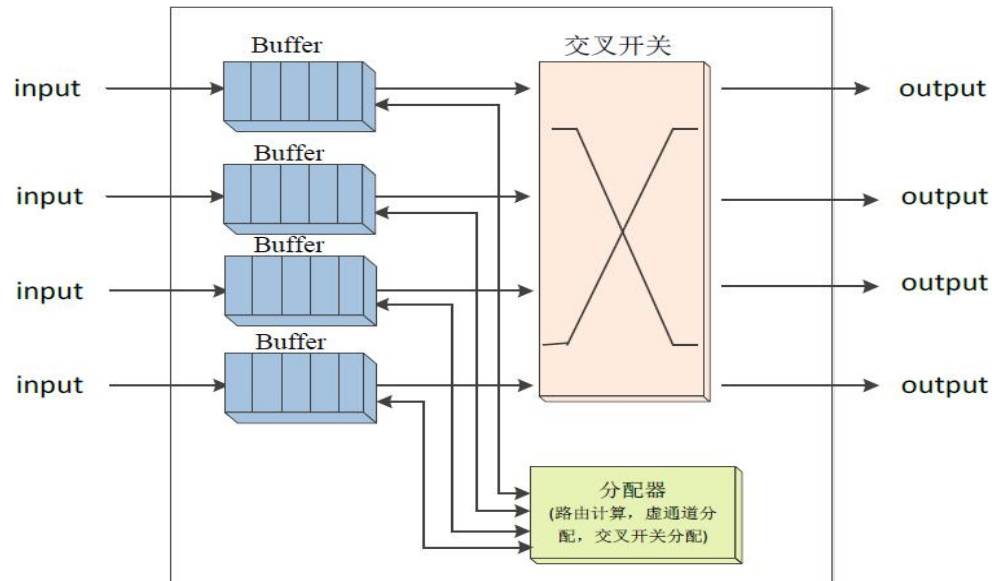


片上网络—路由算法

- 路由算法决定了数据包从源节点到其目的节点传输的路径
- 某些片上网络拓扑结构中（如环），从某个源节点出发到其目的节点的路径只有唯一的一条
- **MESH网络的寻路算法**
 - 维序路由（**Dimension-Order Routing**）：先选择一个维度的方向传输，当此维度走到目的地址相同维度方向后，再改变到其他维度，如先X方向，再Y方向
 - 自适应路由（**Adaptive Routing**）：在每个节点有多种方向选择时，优先选择负载较轻的那一个节点方向作为路径，可以解决局部负载不均衡的情况。

片上网络—路由器

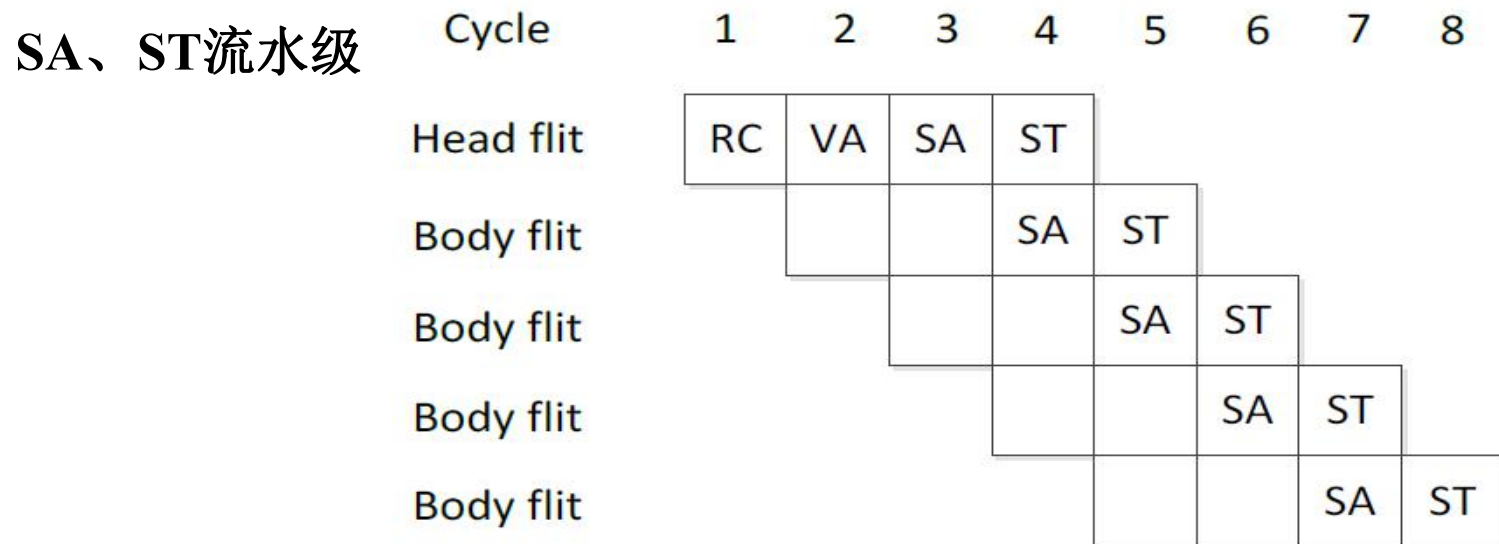
- 路由器将数据包从输入端口存储转发到输出端口发送出去
 - 路由器组成：缓冲区、交叉开关、功能单元和控制逻辑
- 例：适合MESH结构的路由器
 - 每个输入端口有缓冲区来存储数据包、交叉开关连接输入端缓冲区和输出端口、分配器负责路由计算及虚通道分配和交叉开关分配



片上网络—路由器（Cont.）

- 数据包传输的流水级

- 数据包切分为：head flit（flow unit，流控单元），body flit
- 路由器的四个流水级：路由计算（Routing Computation, RC）、虚通道分配（Virtual-channel allocation, VA）、交叉开关分配（switch allocation, SA）、交叉开关上传输（switch traversal, ST）
- head flit必须依次经历RC、VA、SA、ST四个流水级；body flit依次经历

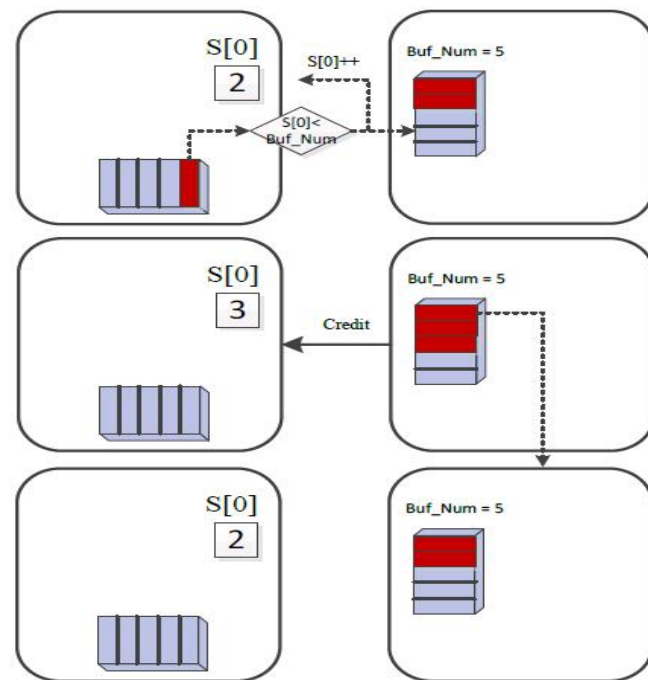
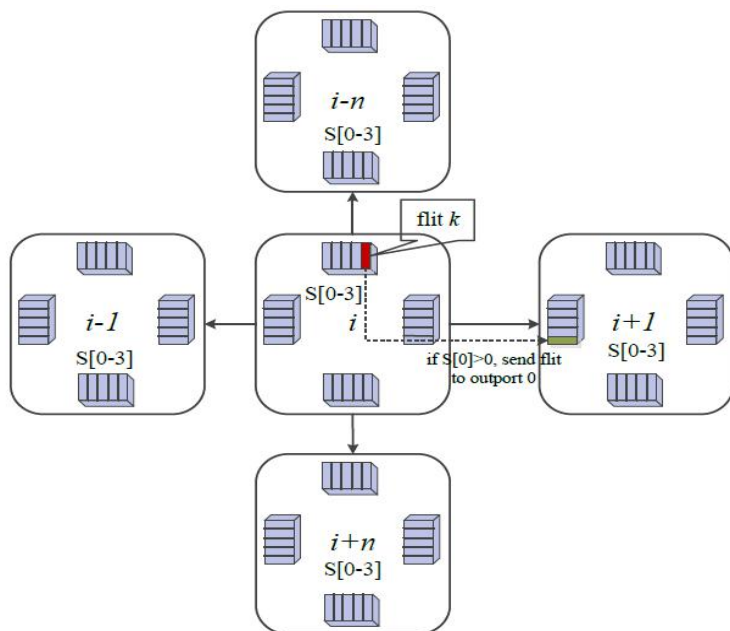


片上网络—流量控制

- 流量控制用来组织每个处理器核节点中有限的共享资源，尽量避免资源冲突的发生
- 片上网络的主要资源是信道（**channels**）和缓冲区（**buffers**）
 - 信道主要用来传输节点之间的数据包
 - 缓冲区是节点上的存储装置，比如寄存器，内存等，用来临时存储经过节点的数据包
- 好的流量控制策略要求它保持公平性和无死锁
 - 不公平的流控制极端情况会导致某些数据包陷入无限等待状态
 - 死锁是当一些数据包互相等待彼此释放资源而造成的无限阻塞的情况

片上网络—流量控制 (Cont.)

- 基于信号量的流量控制方法



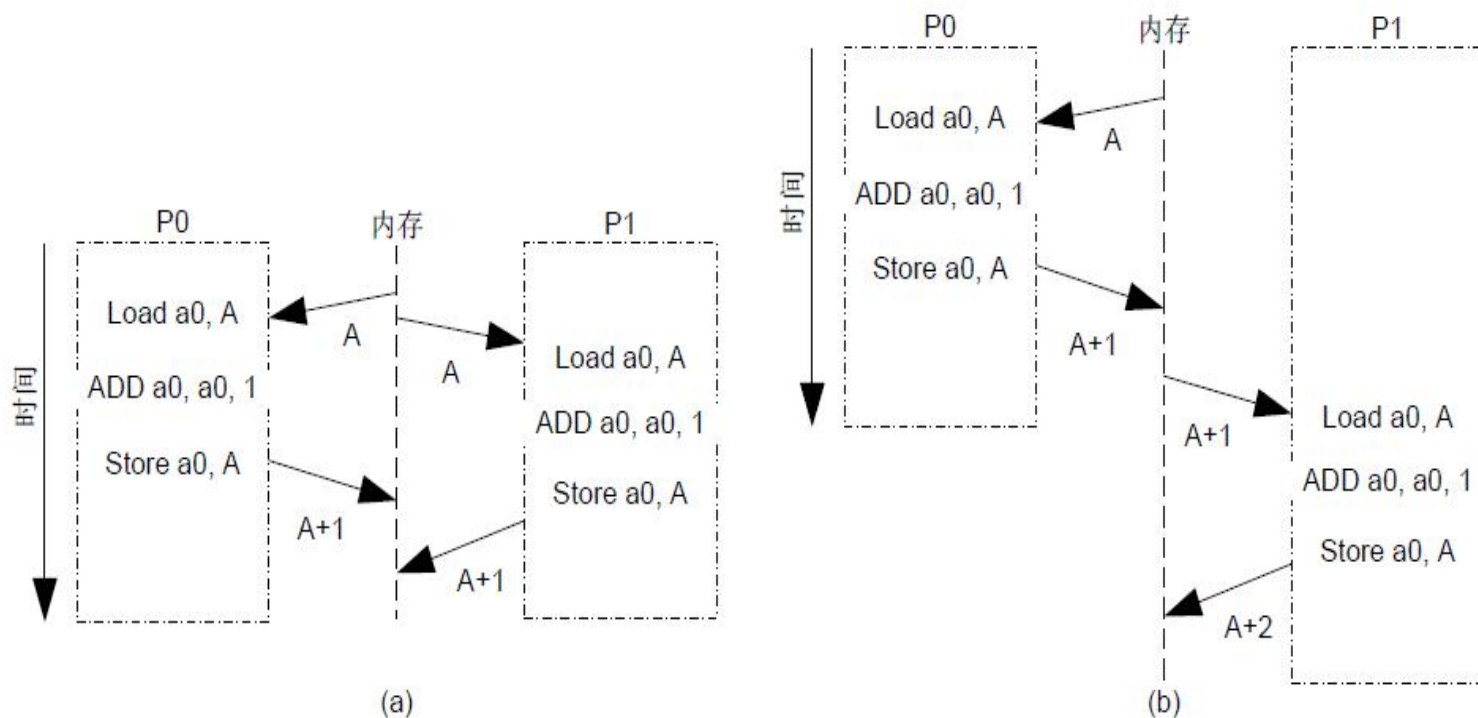
- 节点输入端口有自己的缓冲区队列，存取来自上一跳节点的包，如 $i+1$ 节点最左侧的Buffer用来存储来自 i 号节点的包
- 每个节点对应的相邻节点都有一个计数器， $S[0]$ 到 $S[3]$ ，记录相邻节点内Buffer使用情况

- 对于 $i+1$ 号节点，当它左侧的Buffer送走一个flit时，它就向其左侧的节点发送一个Credit信号，通知左侧节点此Buffer已多出一个空余位置，当左侧节点收到此Credit信号后，则会更新对应的 $S[0]$ 减1。

多核处理器的同步机制

多核处理器的同步问题

- 一个并行程序产生不同的执行结果



不正确：内存变量A的值可能为A+1或者A+2

解决方法：对共享变量A的访问需要同步机制

同步机制

- 三种常见同步机制
 - 锁操作（**Lock**）：临界区
 - 路障操作（**Barrier**）：全局同步
 - 事务内存（**Transaction Memory**）：临界区
- 常见的实现方式
 - 硬件同步指令：“读-改-写”指令
 - 硬件同步指令：“**LL/SC**”指令
 - 用户级软件例程（**routine**）来实现

“读-改-写” 原子指令

- **Test_and_Set 指令**
 - 取出内存中对应地址的值，同时对该内存地址赋予一个新的值
- **Swap指令**
 - 交换两个内存位置的值
- **Compare_and_Swap指令**
 - 取出内存中对应地址的值和另一个值进行比较，如果相等，则交互两个位置的值，否则不进行任何操作
- **Fetch_and_Op指令**
 - 在读取内存对应地址值的同时将该地址的值进行一定的运算再写回
 - 根据运算操作不同，有多种不同的实现形式，如Fetch_and_Increment

LL/SC原子指令对

- 以MIPS的LL和SC指令 为例
 - LL (Load Linked) 取数且置系统中LLbit为1
 - LL为1时，处理器检查相应单元是否被修改，如果其它处理器或设备访问了相应单元或执行了ERET操作，LLbit置为0
 - 执行SC (Store Conditional) 时若Llbit为1，则成功，目标寄存器为1；否则存数不成功，目标寄存器为0
 - 如Fetch_and_increment及swap

```
Try:  ll R2, 0(R1)      ;load linked
      addi R2, R2, 1    ;加1
      sc R2, 0(R1)      ;store cond.
      beqz R2, try      ;写失败，跳转
      nop
```

```
Try:  mov R2, R1        ;送交换值
      ll R4, 0(R3)      ;load linked
      sc R2, 0(R3)      ;store cond.
      beqz R2, try      ;写失败，跳转
      mov R1, R4        ;将读出值送给R1
```

LL/SC原子指令对的优缺点

- **LL/SC原子指令对优点**
 - 设计简单，每条指令只需和内存交互一次，而“读-修改-写”指令需要和内存进行两次交互
 - 在LL和SC之间可以加入任意运算指令，可灵活实现类似“读-改-写”的复杂原子操作
- **LL/SC原子指令对缺点**
 - 密集共享时导致SC不容易成功
 - LL访问要把相应Cache行置为EXC状态，而不是SHD状态，以提高SC成功的概率

实现锁机制

- 自旋锁（Spin lock）
 - 最基本的锁实现方式，处理器核环绕一个锁不停地旋转而请求获得该锁
- 自旋锁缺点：
 - 一个处理器核获得锁后，其他处理器核循环执行Test_and_Set指令访问锁变量，试图获取锁，从而在片上互连上产生大量的访存通信
 - 一种简单改进措施：在Test_and_Set指令之间加入一定的延迟，减少等待阶段Test_and_Set指令自旋执行的次数以减轻访存的压力

```
Void acquire_lock(){  
    while (test_and_set(lock)!=0;  
}
```

```
Void unlock(){  
    lock=0;           //释放锁  
}
```

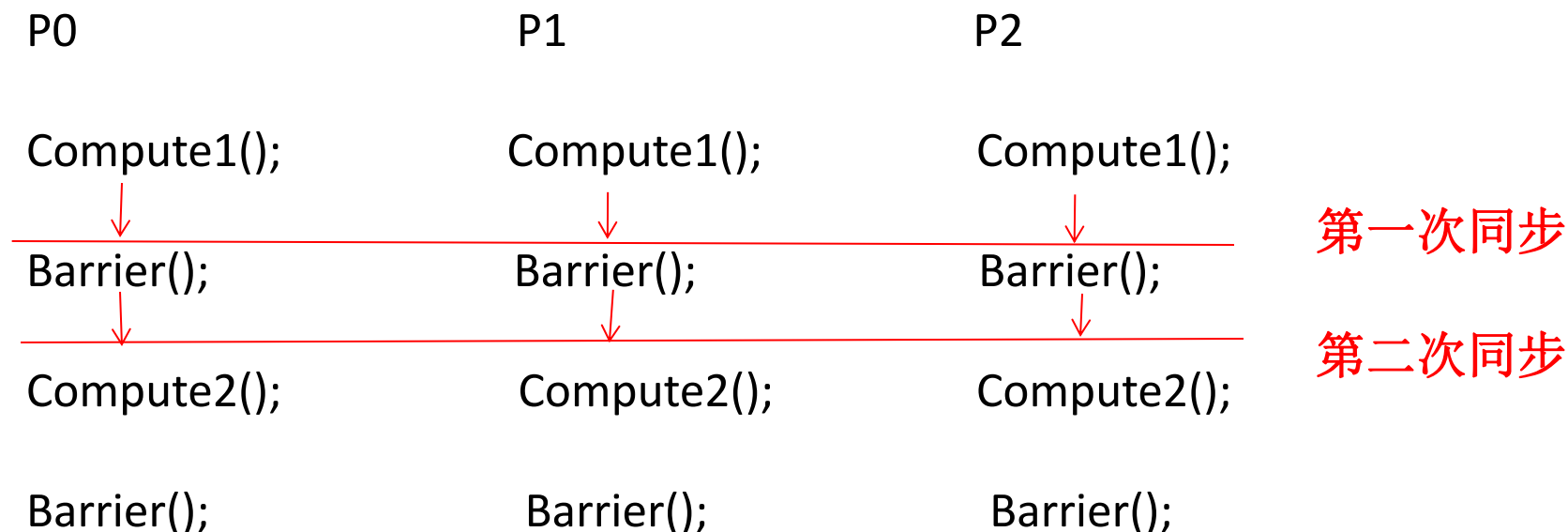
栅障实现

- 栅障（**barrier**）功能
 - 栅障强制要求所有处理器核上执行的线程（进程）等待，一直到所有处理器核上的线程（进程）都到达栅障后，才能释放所有处理器核上的线程（进程），继续执行
- 集中式栅障的实现
 - 设置一个共享的栅障计数变量。当一个处理器核到达栅障，就使用原子指令将栅障的值加1，然后对该栅障值进行自旋等待
 - 优点是简单，缺点是已到达栅障的处理器核，对栅障计数变量的自旋访问，造成很大的访存通信开销

```
Barrier(){  
    fetch_and_inc(count); //到达栅障  
    while(count!=max);    //自旋等待  
}
```

逆向栅障 (Sense Reversal Barrier)

- 栅障作为并行程序不同阶段的分界线可能被多次反复使用
 - 隐含两次同步：第一次保证所有处理器核都到达栅障；第二次保证所有处理器核都离开栅障



上图为并行政程序的代码片段示意图。假设当P0已经达到第二个栅障，P1还未离开第一个栅障，之前的栅障算法就会出错

逆向栅障 (Cont.)

- 逆向栅障的实现

- 当前栅障完成的标志信号与上次不同，只需要一次同步完成栅障

```
Barrier(){  
    local_flag = ~local_flag;           //逆向标志  
    if(fetch_and_inc(count)==max-1) {    //到达栅障  
        count = 0;                       //重置栅障  
        flag = local_flag;               //栅障完成通知  
    } else {  
        while(flag!=local_flag);         //自旋等待  
    }  
}
```


事务内存 (Transaction Memory)

- 传统的同步方法同步开销大，适合于传统并行系统；多核处理器中同步次数多、粒度小，需要新的并行同步方案
- 事务定义：
 - 访问共享变量的代码区域声明为一个事务 (transaction)。事务具有原子性 (Atomicity)，即事务中的所有指令要么执行要么不执行；一致性 (Consistency)，即任何时刻内存处于一致的状态；隔离性 (Isolation)，即事务不能看见其它未提交事务涉及的内部对象状态
 - 事务执行并原子地提交所有结果到内存（如果事务成功），或中止并取消所有的结果（如果事务失败）
 - 事务内存类似于乱序执行中的重命名寄存器，先猜测执行，可以取消

事务内存 (Cont.)

- 事务内存实现方式
 - 软件事务内存（以运行时库或者编程语言形式）
 - 硬件事务内存（如Intel TSX (Haswell)、IBM Power8）
- Intel TSX (Transactional Synchronization Extensions)
 - 提供3条新指令：XBEGIN、XEND和XABORT
 - XBEGIN指令启动一个事务，并提供了回退地址信息
 - XEND指令表示事务的结束
 - XABORT指令立刻触发一个中止，类似于事务提交不成功
 - 硬件以Cache行为单位，跟踪事务的读集（read-set）和写集（write-set）。如果事务读集中的一个Cache行被另一个线程写入，或者事务的写集 中的一个Cache行被另一个线程读取或写入，则事务就遇到冲突（conflict）。冲突通常导致事务中止

典型多核处理器

分析多核CPU的要素

- 处理器核
 - 异构、同构；通用、专用；重核、轻核；多核、众核
- 互连结构
 - 可伸缩程度，是否支持片间互连
- 访存结构
 - 所有核同一地址空间、不同核不同地址空间
 - Cache一致性协议：是否可伸缩
- 峰值性能和访存带宽及IO带宽
 - 避免茶壶里面倒饺子
 - 通用多核处理器访存带宽和峰值性能差距不能太大

龙芯3号多核处理器

- 面向桌面/服务器应用

- 可扩展互连，节点内交叉开关连接四核，节点间通过可伸缩网络互连
- 支持片间直连实现多路共享存储
- 片内/片间多核共享LLC（片间共享LLC不如片间共享内存），基于目录的Cache一致性协议
- 四核龙芯3A3000的峰值性能为24Gflops（1.5GHz）；理论访存带宽为24GBps，stream实测访存带宽为13GBps

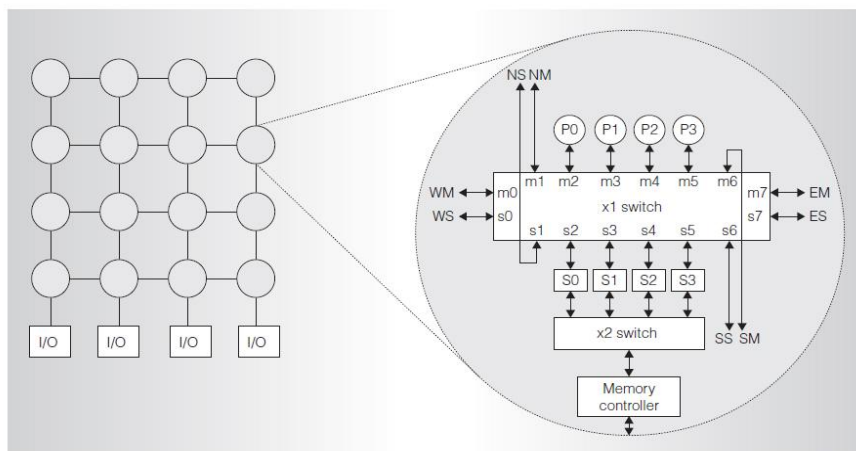


Figure 7. Godson-3 interconnection topology. A 2D mesh connects nodes, and each node connects cores and L2 modules through a crossbar; memory controllers are connected in nodes, and I/O controllers are connected in the 2D mesh boundary.

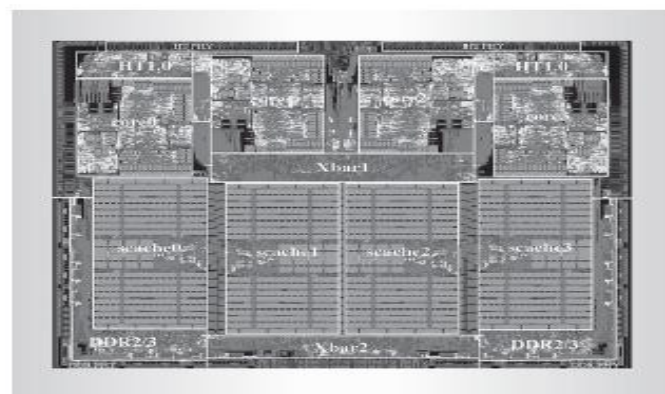
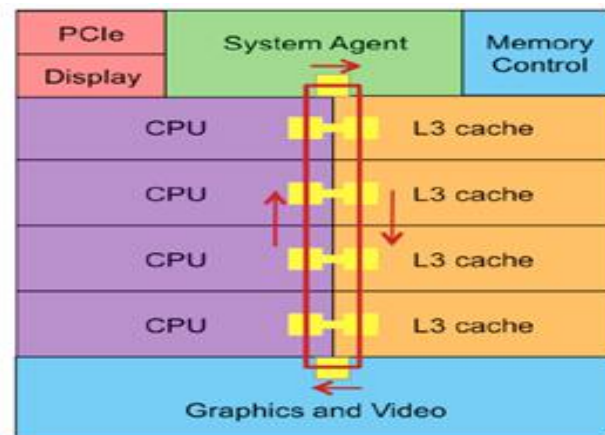


Figure 10. Layout of the four-core Godson-3. The HyperTransport links are at the top, the CPU cores and L2 caches are in the center,

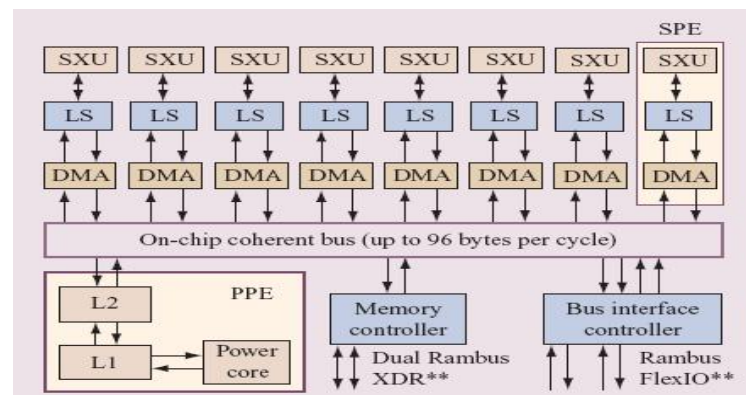
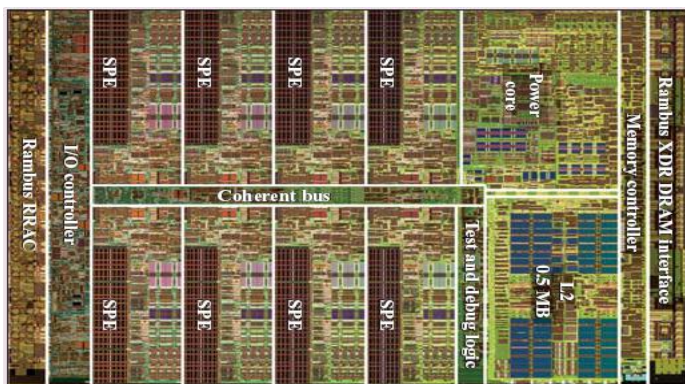
Intel Sandybridge架构

- **Sandybridge架构于2011年推出，32纳米的tock**
 - 组成部分：处理器核、环连接、共享L3、System Agent和GPU
 - 处理器采用乱序执行技术、支持双线程、支持256位AVX向量指令集扩展
 - System Agent包括内存控制、PCIE接口、显示引擎、功耗控制单元等
 - L3（LLC）在处理器核、图形核心和系统代理之间共享
 - 采用侦听一致性协议，随后的IvyBridge开始采用目录协议
 - 峰值性能为96GFlops（4核、3GHz、128位向量），理论访存带宽25.6GBps，stream实测访存带宽14~16GBps



IBM CELL

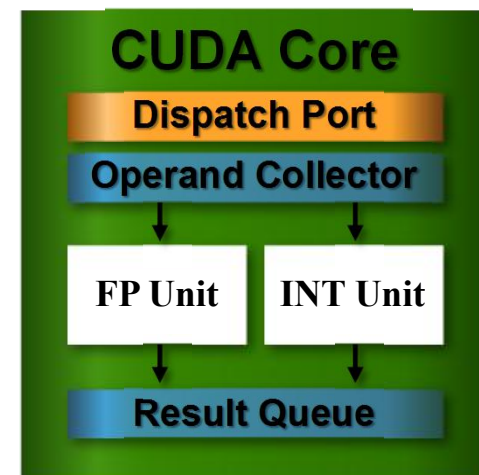
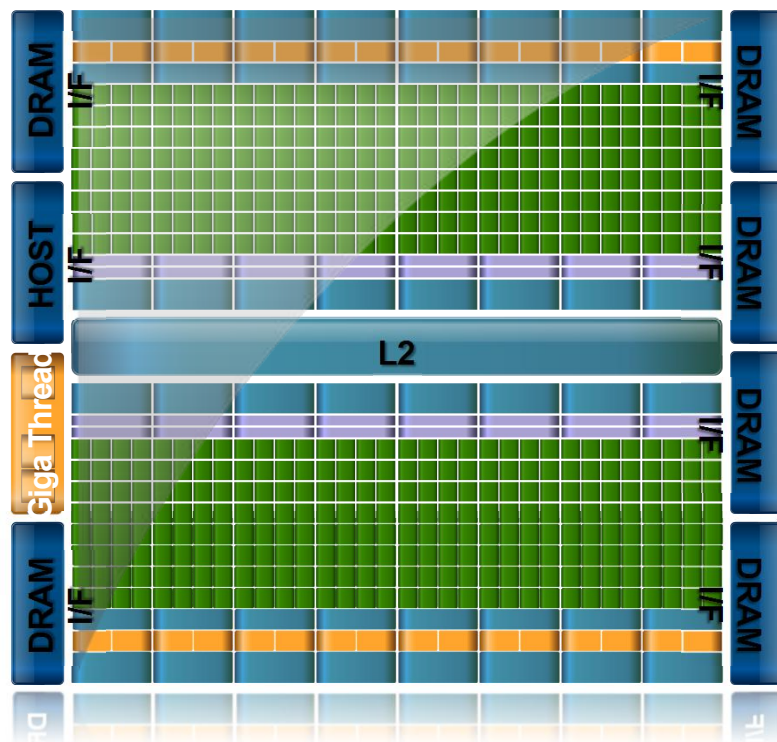
- 由IBM、索尼和东芝联合研发，在ISSCC2005上首次公开
 - 峰值性能256GFlops@4GHz，理论访存带宽25.6GBps，面向游戏、超算等
- 采用异构多核结构
 - 1个双发射64位PowerPC核（PPE）和8个SIMD向量协处理器（SPE）
 - 高带宽的环状高速总线（EIB）把PPE、SPE及RAMBUS内存接口控制器（MIC）、FlexI/O外部总线接口控制器（BIC）连接起来
 - 每个SPE有256KB的LS（Local Storage），LS与内存间通信必须通过DMA进行，LS对PPE可见（PPE任务是运行OS）



NVIDIA Fermi GPU

- 现代GPU包括数百个并行浮点运算单元，是典型众核处理器
- 例：NVIDIA公司的Fermi GPU体系结构
 - 第一个基于Fermi体系结构的GPU芯片有30亿个晶体管，支持512个CUDA核心，组织成16个流多处理器（Stream Multiprocessor，SM）
 - 每个SM包含32个CUDA核心、16个Load/store单元、4个特殊处理单元（Special Function Unit，SPU）、64KB的片上高速存储
 - CUDA核心支持一个全流水定点ALU和浮点单元（FPU）
 - 64KB片上高速存储可配置（48KB共享存储+16KB L1 Cache或16KB共享存储+48KB L1 Cache）
 - 768KB统一的片上二级Cache，在16个SM间共享
 - 采用CUDA（Compute Unified Device Architecture）编程环境，可以采用类C语言开发程序，编译器和硬件可以在GPU上将上千个CUDA线程并行执行
 - GeForce GTX 480包含480核，主频700MHz，单精度浮点峰值性能为1.536TFlops，访存带宽177.4GBps
 - 不仅用作GPU，还用于高性能计算

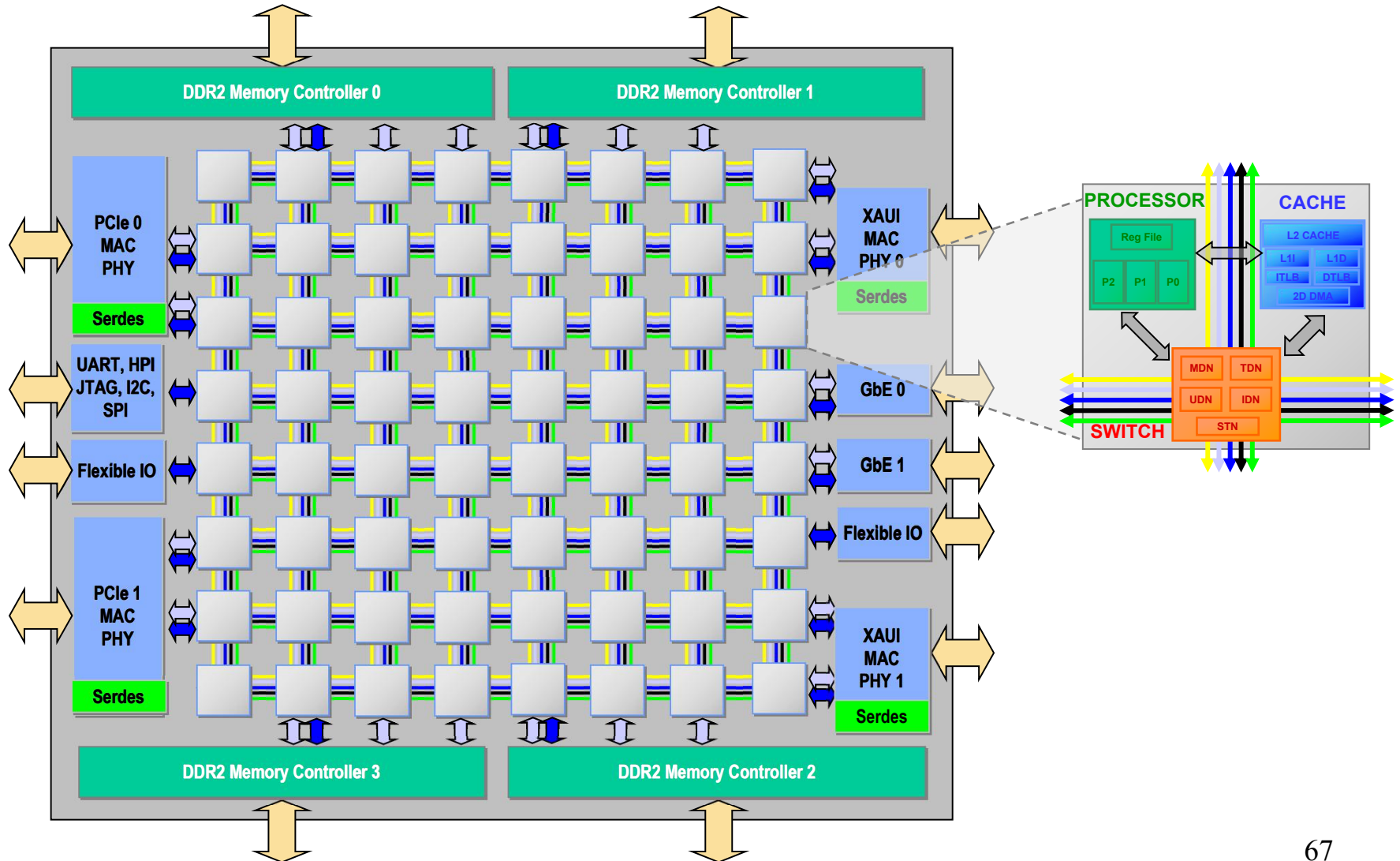
NVIDIA Fermi结构



TILE64众核处理器

- Tiler公司于2007推出，面向网络和视频处理等领域
- 支持64个tile（瓦片），组成8*8的MESH结构，每个tile包含通用CPU核、Cache和路由器
 - CPU核：支持MIPS类VLIW指令集、三发射、按序、短流水
 - 每个Tile有私有L1（16KB）和私有L2 Cache（64KB）、虚拟的L3 Cache（所有tile的L2 Cache聚合）
 - 路由器实现tile间的MESH结构
- 采用Neighborhood缓存机制实现片上分布式共享Cache
 - 每个虚拟地址对应一个Home tile，先访问该Home tile的私有Cache，如果不命中则访问内存；Home tile负责维护一致性
- 峰值性能为每秒192G个32位运算（主频1GHz），理论访存带宽为25GBps

TILE64众核处理器 (cont.)



作业