

Chapter 2



并行体系结构

Outline

- 弗林分类法
- 共享内存系统和分布式内存系统
- 互联网络
- 存储器层次结构和缓存一致性

弗林分类法(Flynn's Classification)

classic von Neumann

<p>SISD</p> <p>Single instruction stream</p> <p>Single data stream</p>	<p> (SIMD)</p> <p>Single instruction stream</p> <p>Multiple data stream</p>
<p>MISD</p> <p>Multiple instruction stream</p> <p>Single data stream</p>	<p> (MIMD)</p> <p>Multiple instruction stream</p> <p>Multiple data stream</p>

not covered

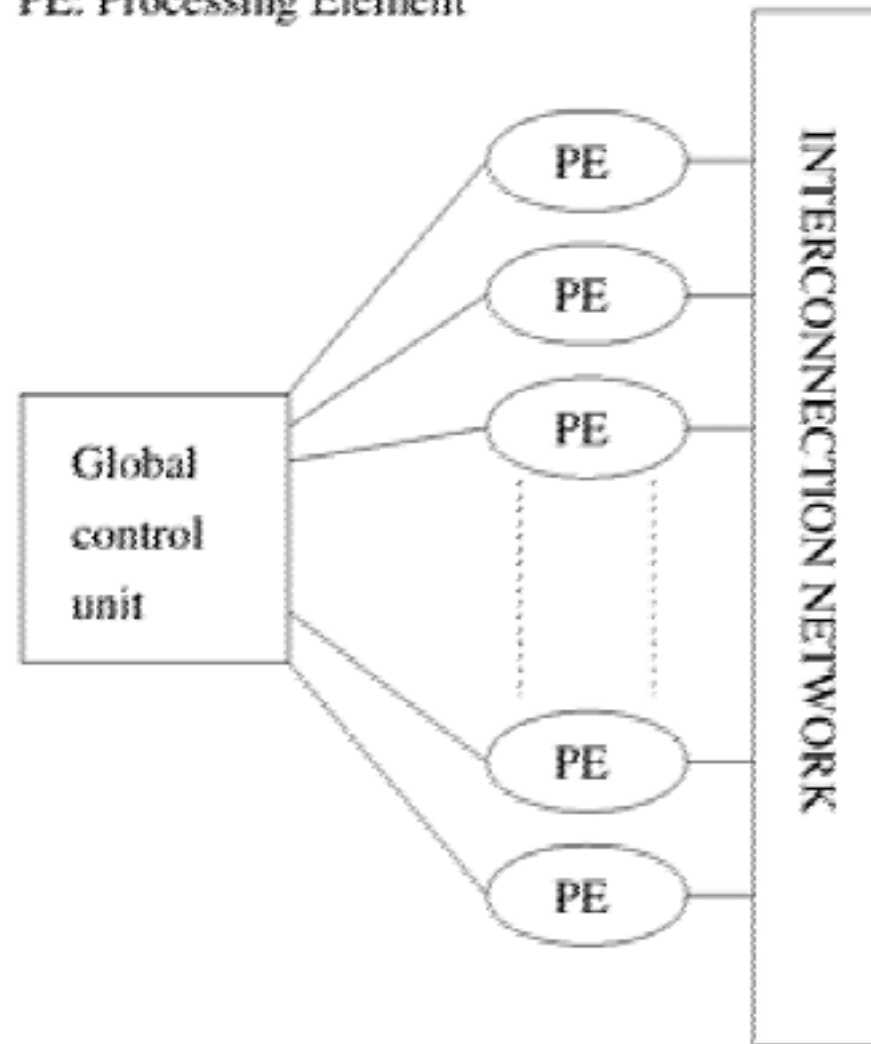


SIMD

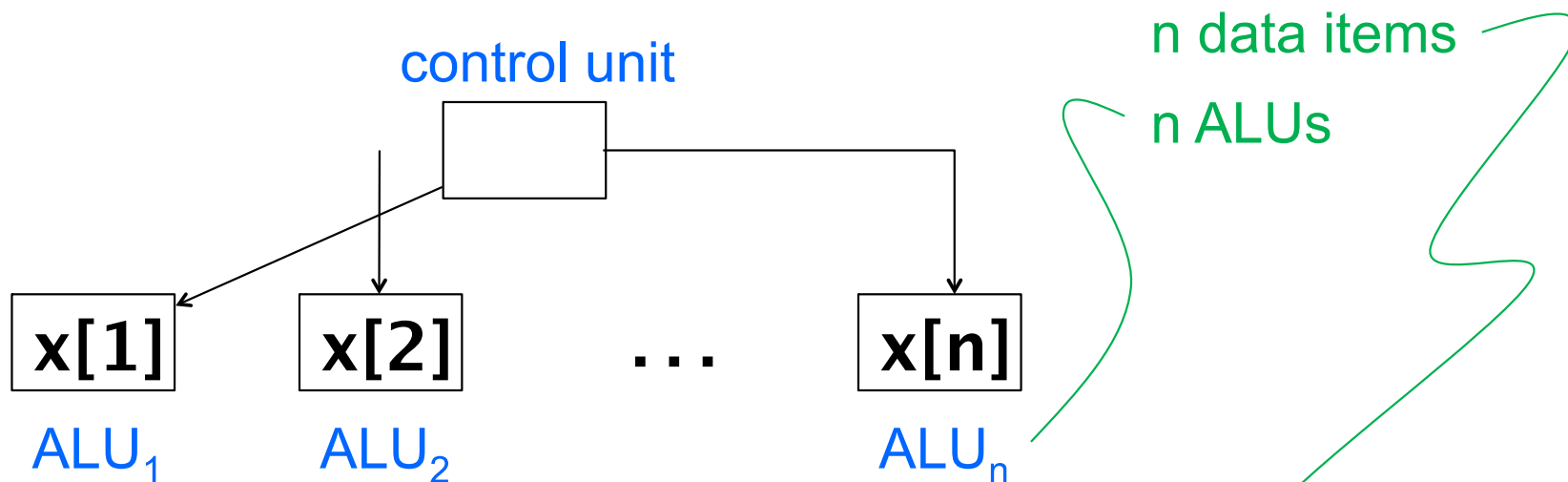
- 将同一指令应用于多个数据项
- 多个处理器以同步方式执行同一程序
 - 在一个 "经典 "的SIMD系统中，每个ALU都必须等待下一条指令的广播，然后再进行操作
 - 所有的ALU执行相同的指令或处于空闲状态
- 通过在处理器之间划分数据来实现并行性。每个处理器所 “看到的” 数据可能是不同的

SIMD

PE: Processing Element



SIMD Example(1)



```
for (i = 0; i < n; i++)  
    x[i] += y[i];
```



SIMD Example(2)

- 如果我们没有那么多的ALU和数据项怎么办?
- 切分计算, 迭代执行
- Ex. $m = 4$ ALUs and $n = 15$ data items.

Round3	ALU ₁	ALU ₂	ALU ₃	ALU ₄
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

SIMD

- 所有的ALU都执行相同的指令或处于空闲状态。这可能会降低SIMD系统的整体性能

```
for (i = 0; i < n; i++)  
    if (y[i] > 0.0) x[i] += y[i];
```

- 对大规模数据并行问题有效，但对复杂控制类并行问题无效
- 典型案例：向量机(Vector processors)

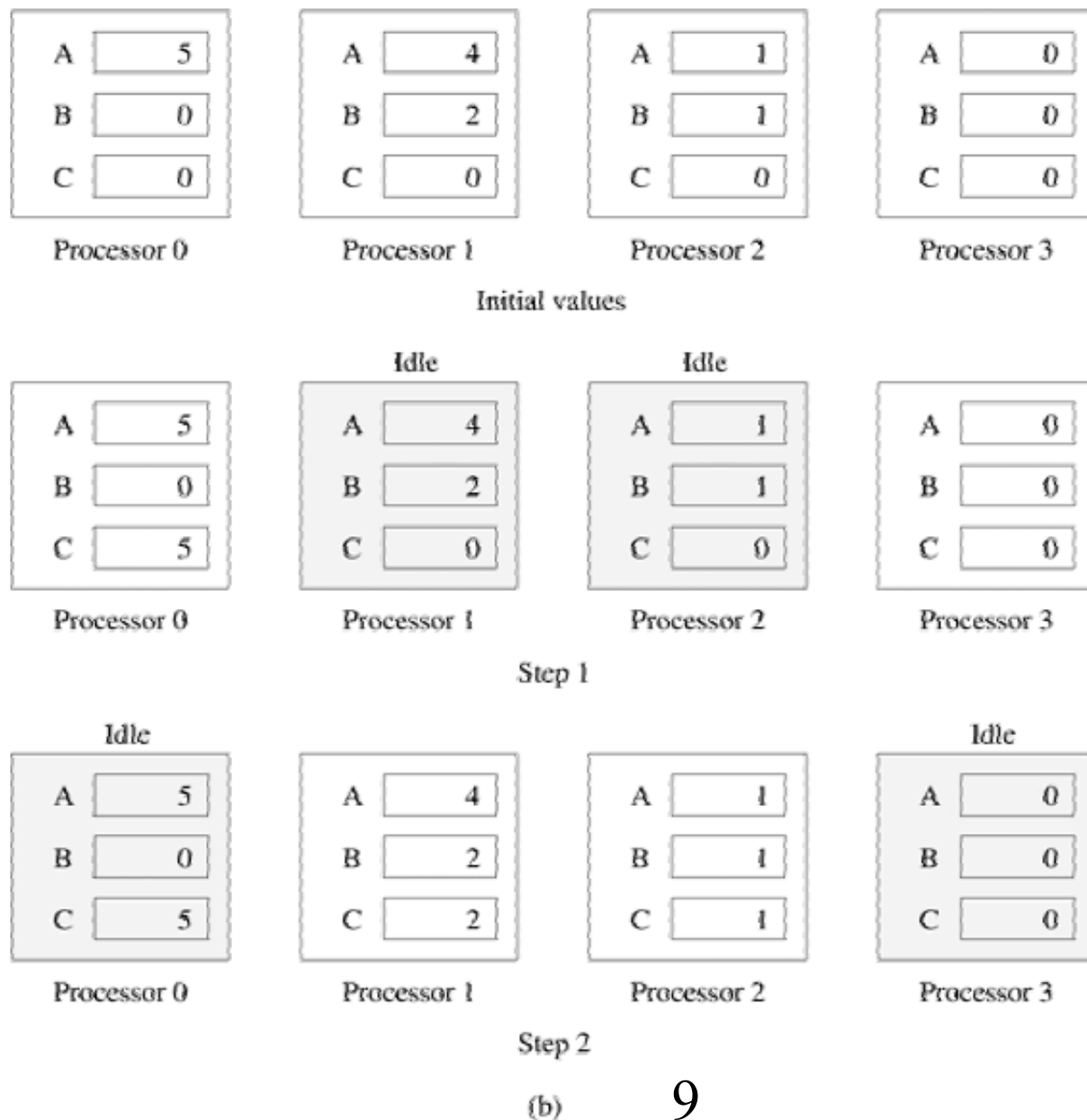
在SIMD架构上的执行

```
if (B == 0)
    C = A;
else
    C = A/B;
```

(a)

(a) the conditional statement;

(b) the execution of the statement in two steps.



向量机(1)

- 对数组或向量进行操作（传统的CPU则对单个数据元素或标量进行操作）

- 典型体系架构特点

(1)向量寄存器

- 能够存储一个操作数的向量，并同时对其内容进行操作

(2)向量化和流水线功能单元

- 同样的操作适用于向量中的每个元素（或成对的元素）

(3)矢量化指令

- 对向量而非标量进行操作

```
e.g:  for (i = 0; i < n; i++)  
        x[i] += y[i];
```

向量机(2)

(4) 交错式存储器

- 多个内存“库” (Memory bank) , 可以独立访问
- 访问一个库后, 在重新访问该库之前会有一个延迟
- 如果将一个向量的元素分布在多个库中, 可以减少或消除加载/存储连续元素的延迟

(5) 交叉内存访问 (Strided memory) , 硬件散射/聚集

- 该程序访问位于固定间隔的向量的元素
- 散射/聚集 (在这里是指写入 (散射) 或读取 (聚集) 位于不规则间隔的矢量的元素)

向量机(3)

• 优点

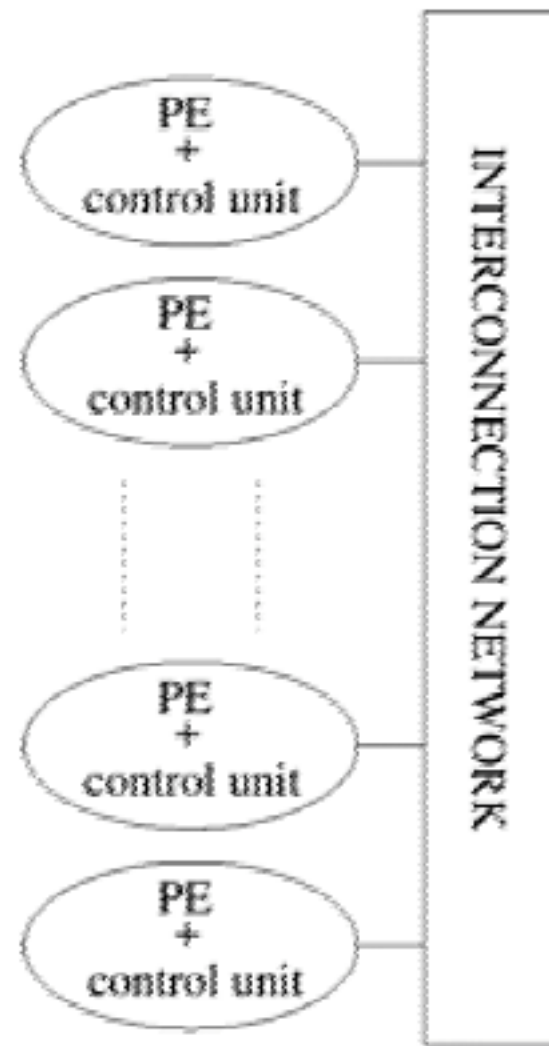
- 快，简单
- 矢量化编译器善于识别可被矢量化的代码
- 编译器还可以提供不能被矢量化的循环的信息，并提供原因。还能帮助用户重新评估代码
- 高内存带宽
- 每个加载的数据项都被使用

• 缺点

- 难以处理不规则的数据结构
- 更大规模问题扩展性

MIMD

- 支持多个指令流同时在多个数据流上运行
- 通常由完全独立的处理单元或核心的集合组成，每个单元都有自己的控制单元和自己的ALU
- MIMD系统通常是异步的
- 许多MIMD系统有独立的时钟
- 典型案例：
 - 共享内存和分布式内存机器

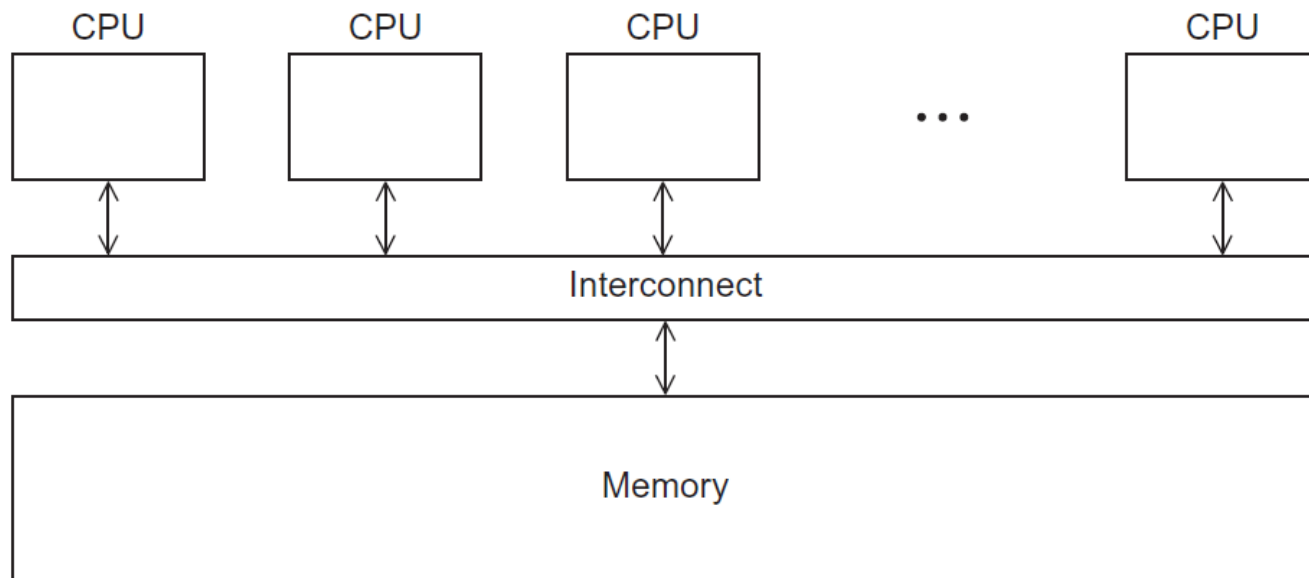


Outline

- 弗林分类法
- 共享内存系统和分布式内存系统
- 互联网络
- 存储器层次结构和缓存一致性

共享内存系统(Shared Memory System)

- 一组处理器通过互连网络与存储系统相连
- 处理器可以直接访问系统中的所有数据
- 处理器通常是通过访问共享数据结构进行隐式通信的

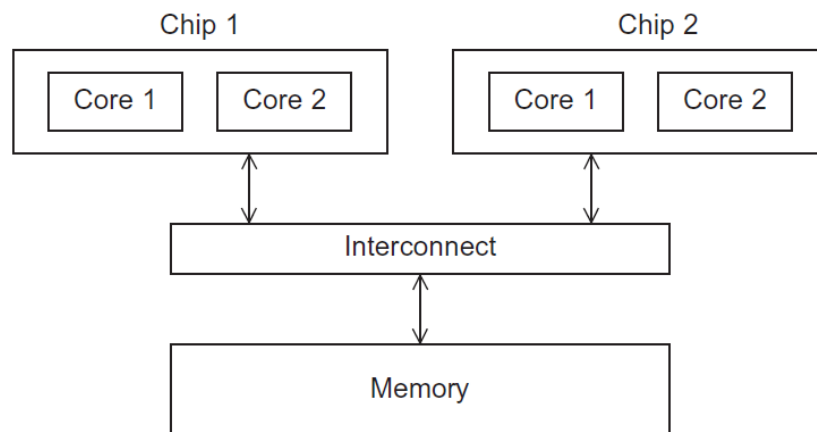


共享内存系统

- 多核处理器在一个芯片上有多个CPU或内核
 - 核心有私有的一级缓存，而其他的缓存可能会或不可能在核心之间共享
- 在具有多个多核处理器的共享内存系统中，互连可以将所有处理器直接连接到主内存（UMA），或者每个处理器可以直接连接到一个主内存块，而处理器可以通过处理器内置的特殊硬件访问彼此的主内存块（NUMA）

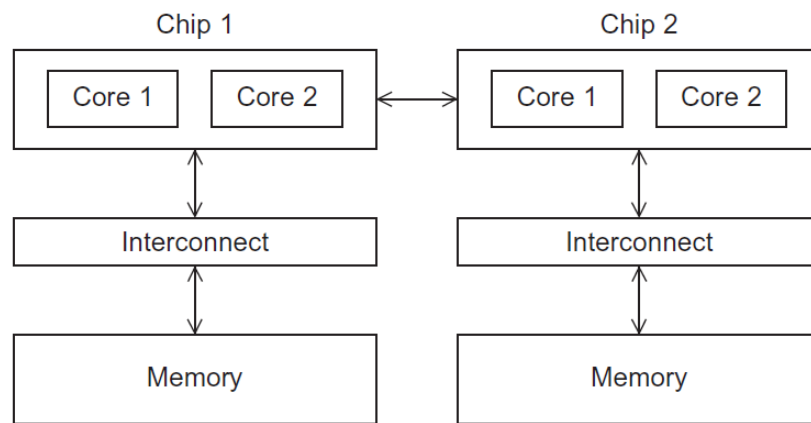
(Uniform Memory Access)均匀存储访问

- 物理存储器被所有处理器**均匀共享**;
- 所有处理器访问任何存储字**取相同的时间**;
- 每台处理器可带**私有高速缓存**;
- 外围设备也可以一定形式**共享**



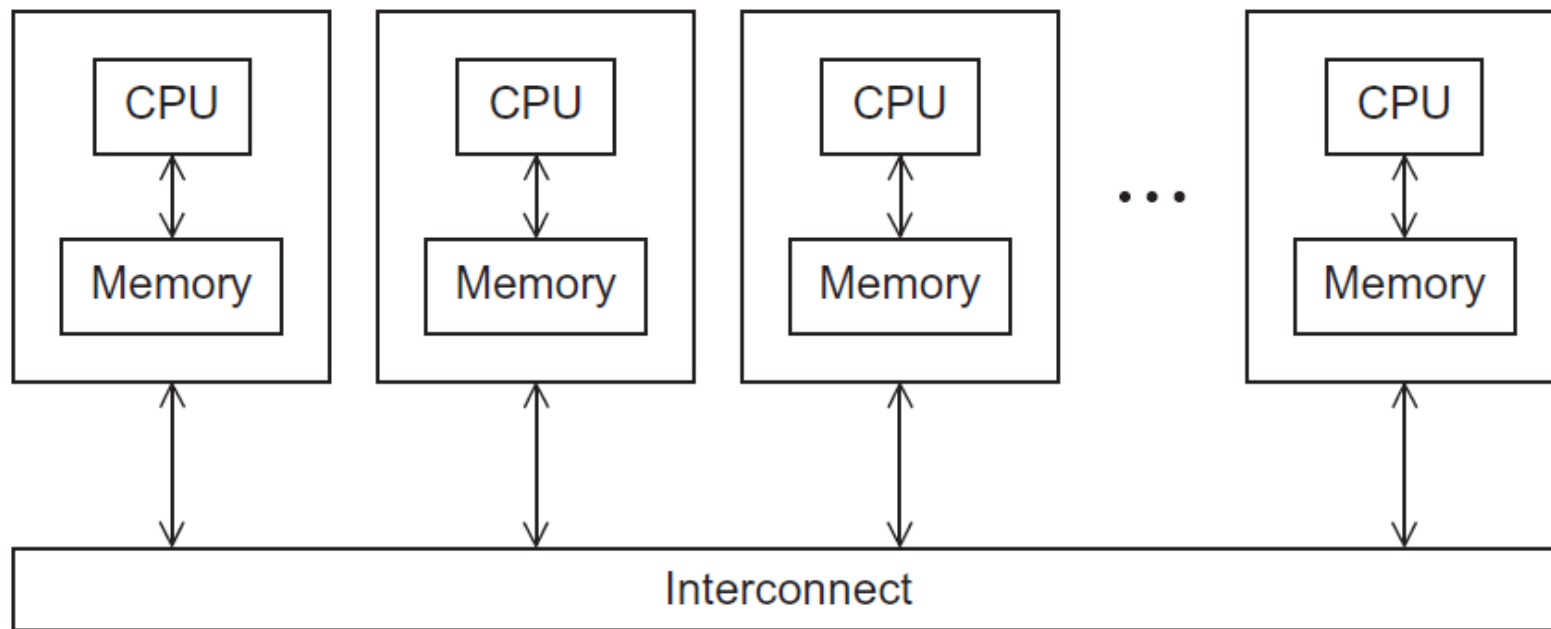
NUMA(Nonuniform Memory Access)非均匀存储访问

- 共享存储器分布在所有的处理器中;
- 处理器访问存储器的**时间不均匀**;
- 每台处理器照例可带**私有高速缓存**;
- 外设也可以某种形式共享。



分布内存系统

- 每个处理器都有自己的私有内存
- 处理器通过明确的消息传递或特殊函数共享数据



Outline

- 并行硬件
 - 弗林分类法
 - 共享内存系统和分布式内存系统
- 互连网络
- 存储器层次结构和缓存一致性

互联网络 (Inter-connection)

• 两种分类:

- 共享内存互连
- 分布式内存互连

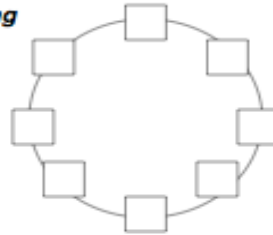
Bus



Linear array



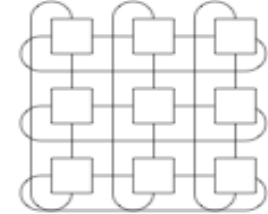
Ring



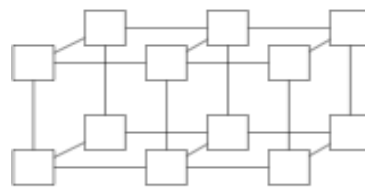
2-D Mesh



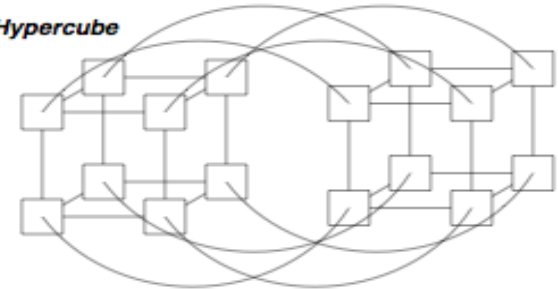
2-D Torus



Cube



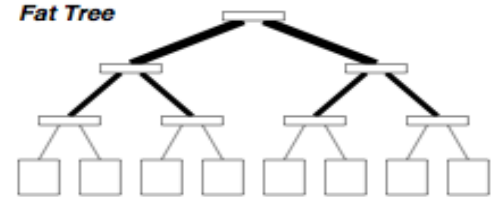
Hypercube



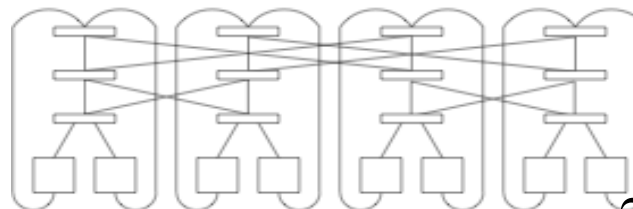
Tree



Fat Tree



Butterfly



Legend

- Switch-only node
- Node



• 总线互连 BUS

- 通信线，加上一些控制访问总线的硬件
- 通信线被连接到它的设备所共享
- 低成本和灵活性
- 随连接到总线上的设备数量的增加，总线竞争加剧，性能下降

• 交叉开关矩阵 Crossbar

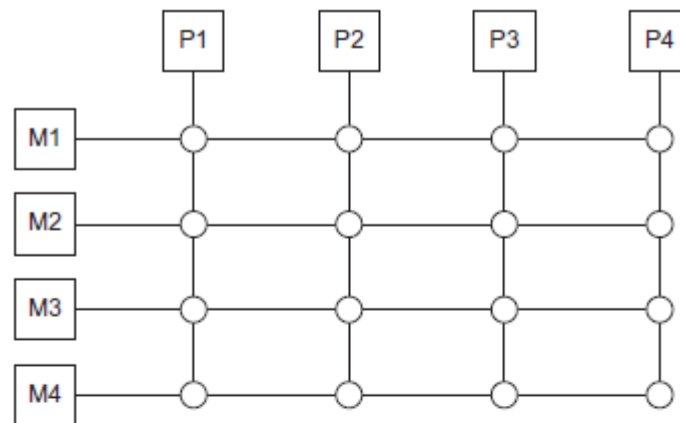
- 用开关来控制连接设备之间的数据路由
- 允许不同设备之间同时进行通信
- 更快



但交换机和链接的成本相对较高

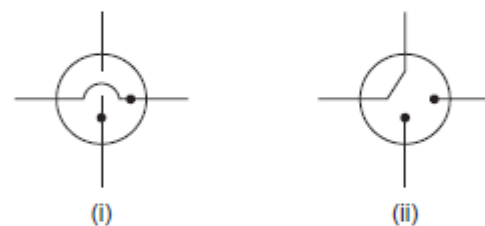
交叉开关矩阵

(a) 一个连接4个处理器(P_i)和4个内存模块(M_j)的crossbar



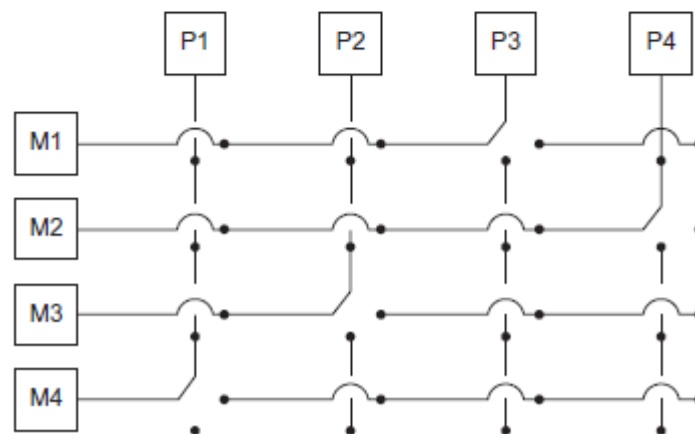
(a)

(b) 配置内部开关



(b)

(c) 处理器同时访问内存



(c)



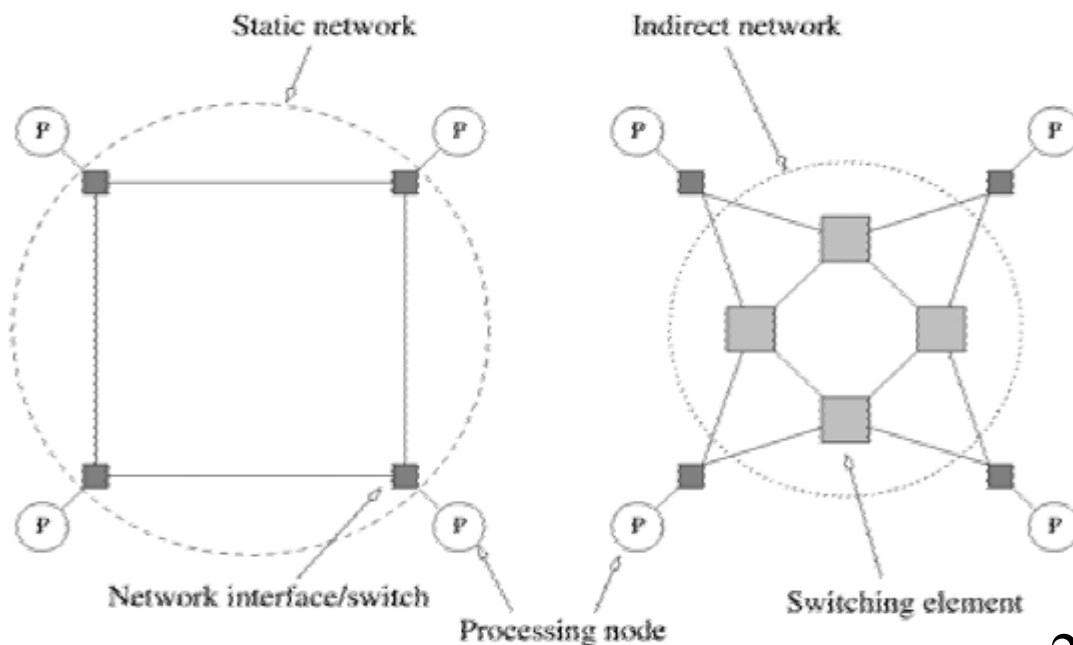
互联网络——分布内存

- 直接互连（静态网络）

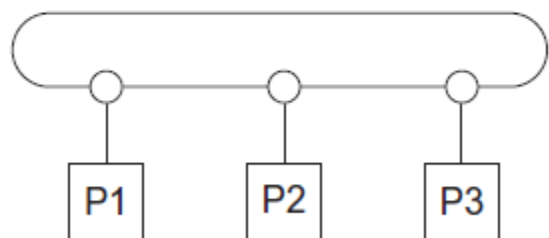
- 每个开关都直接连接到一个处理器-内存对，而开关之间是相互连接的
- 它由处理节点之间的点对点通信链接组成

- 间接互连（动态网络）

- 开关不能直接连接到一个处理器上
- 它是使用交换机和通信链路建立的

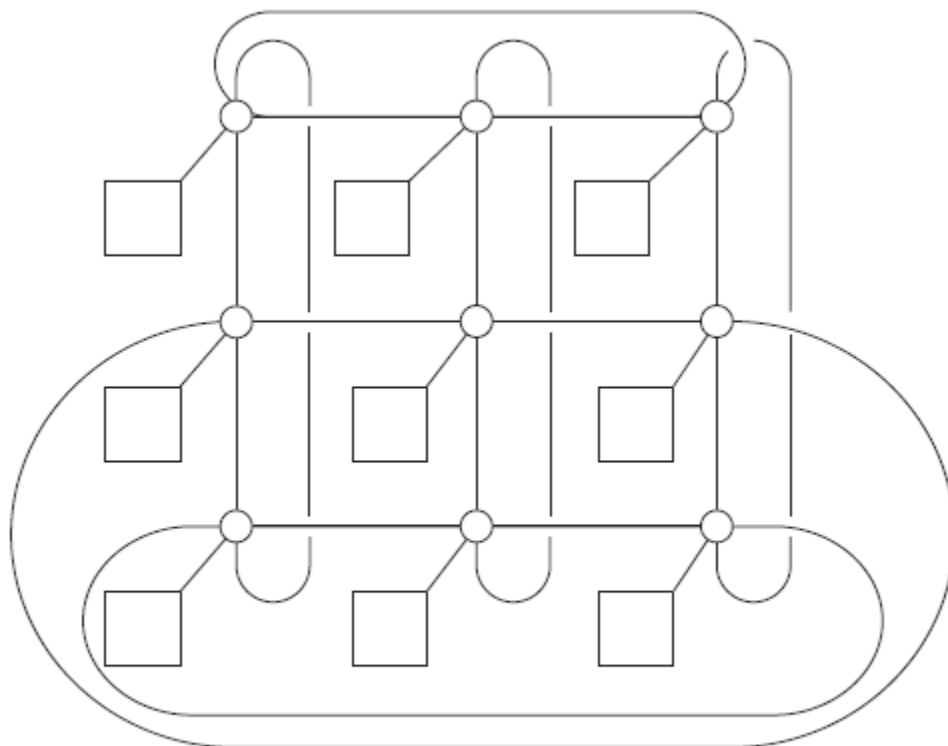


分布内存互联网络——直接互联



(a)

环ring



(b)

环面网格toroidal mesh



度量指标

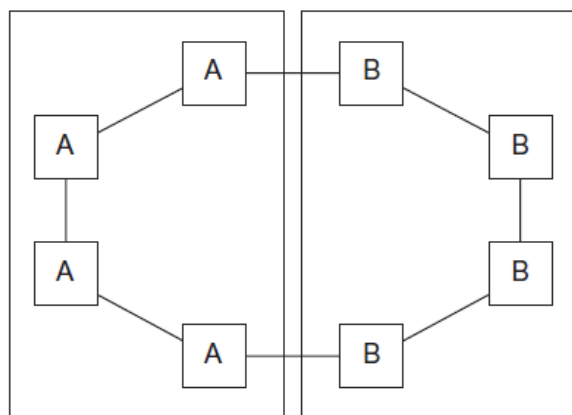
- 节点度(Node Degree)
- 网络直径(Network Diameter)
- 对剖宽度(Bisection Width)
- 对剖带宽(Bisection Bandwidth)
- 是否恒定边长

度量指标 (一)

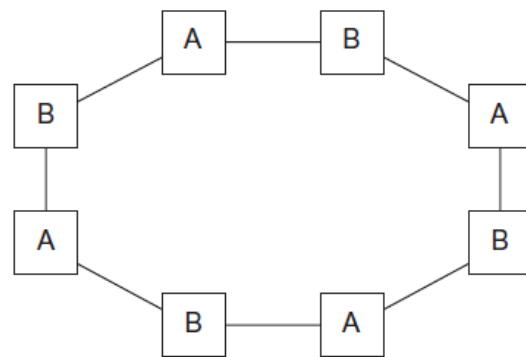
• 对剖宽度 (Bisection width)

— 衡量 "同时通信的数量" 或 "连接性" 的标准
有多少种同时进行的通信可以在两部分之间发生
对分宽度提供一个 "最坏情况" 的估计

— 计算方法: 将节点集分成大小相同的两部分所需去除的最小边数



(a)



(b)



度量指标 (三)

• 网络直径

- 两个节点之间的最大距离
- 越短越好
- 任意点对之间通信复杂性的下限

• 节点度

- 如果边/节点的最大数量是一个与网络规模无关的常数，那是最好的，因为这可以使处理器组织更容易扩展到更多的节点数量
- 每个节点的最大边数

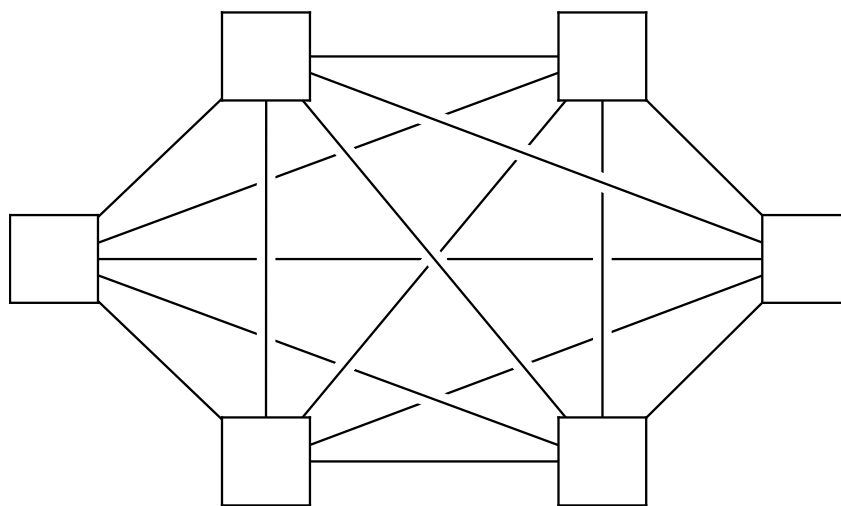
• 是否恒定边长

- 将节点和边布置在三维空间中，最大的边长是一个与网络大小无关的常数



分布内存互联——全连接网络

- 每个节点都直接连接到其他每个节点



成本太高，不现实



分布内存互联——全连接网络

- 直径

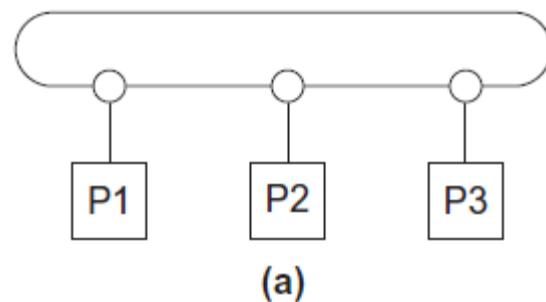
- 最远的两个节点之间的距离
- K_n : $d = O(1)$
- but #edges $m = O(n^2)$;

- 对分宽度

- 将网络大致分为两半的切割中的最小边数 - 决定了网络的最小带宽
- K_n 的分切宽度是 $O(n^2)$

分布内存互联——线状网络

- 交换机排列成一个一维网状
- 变体，环形网络——终端的交换机之间进行环绕式连接
 - 本质上支持两个方向的管道

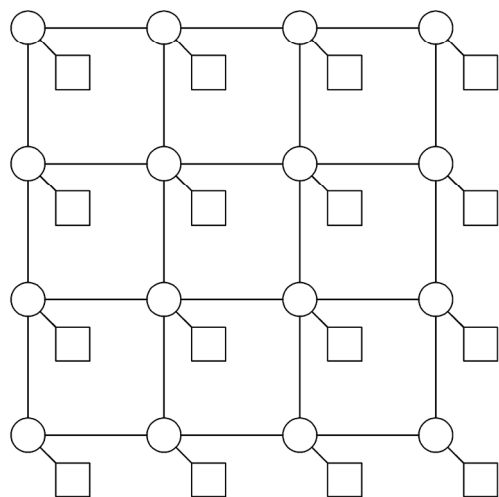


线状网络、环状网络评估

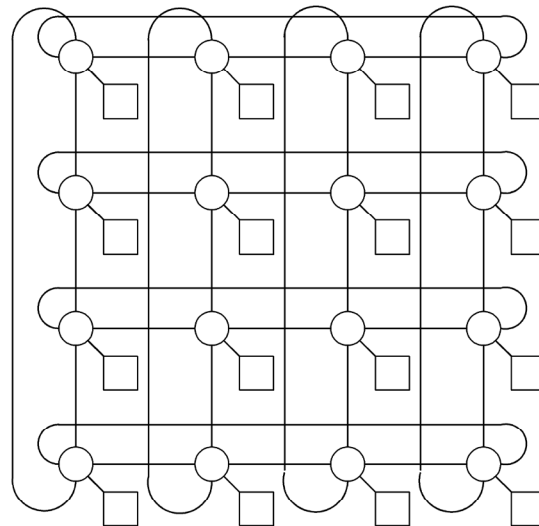
- 直径
 - Linear : $n-1$ or $\Theta(n)$
 - Ring: $\lfloor n/2 \rfloor$ or $\Theta(n)$
- 分割带宽:
 - Linear: 1 or $\Theta(1)$
 - Ring: 2 or $\Theta(1)$
- 节点的度:
 - 2
- 恒定边长?

分布内存互联——2D 网格

- 直接拓扑结构
- 节点排列成一个二维的格子或网格
- 只允许在相邻的交换机之间进行通信
- 包括网状边缘的节点之间的环绕连接的变体



(a)



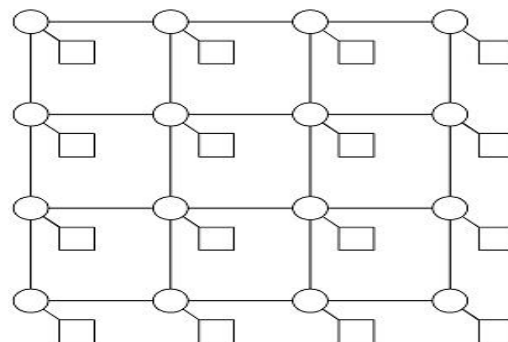
(b)



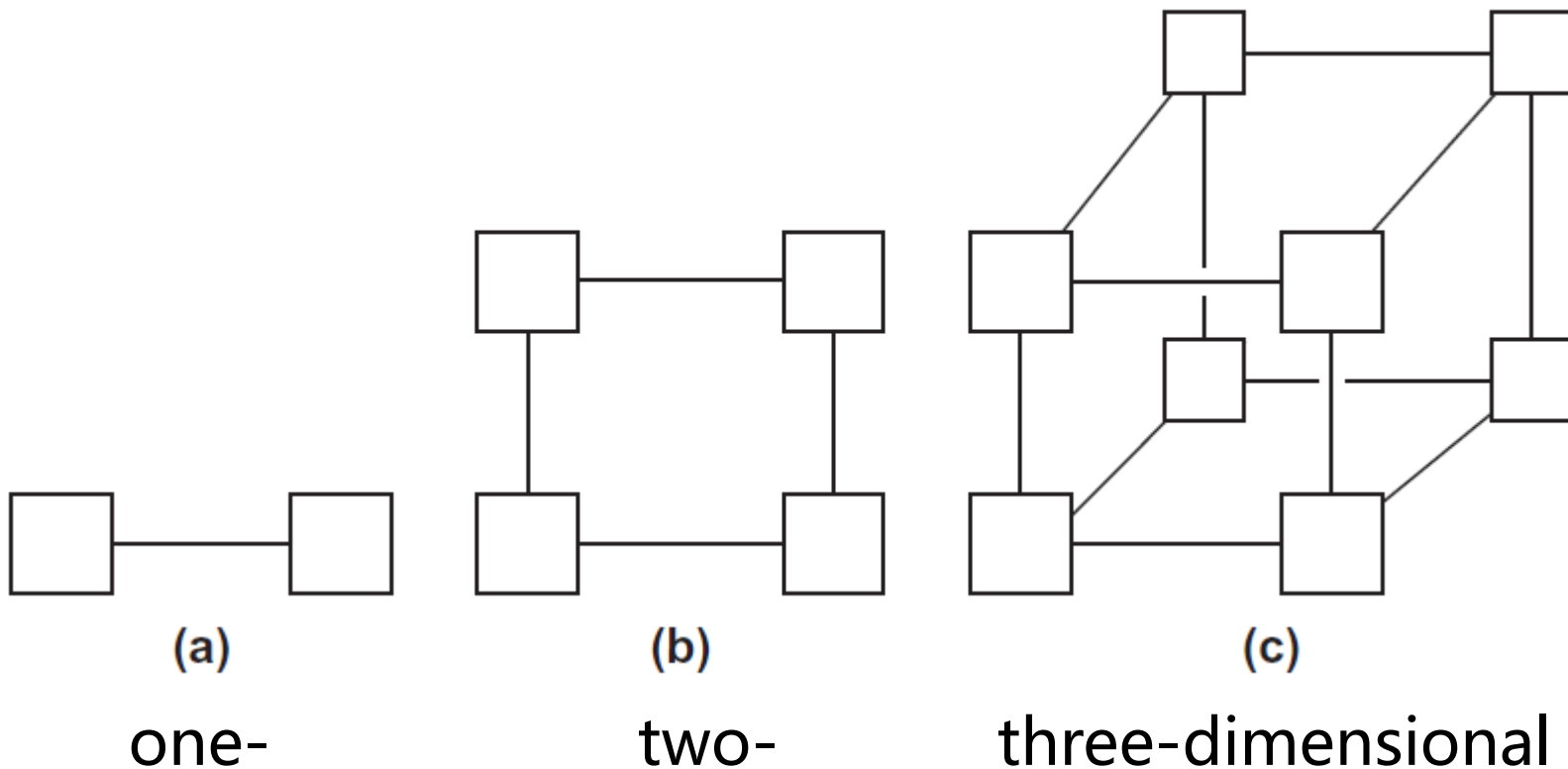
2D 网格评估

n :处理器数量(Assumes mesh is a square)

- 直径
 - $\Theta(n^{1/2})$
 - 对需要与共享数据的任意节点进行处理的算法设置了下限
- 对分宽带
 - $\Theta(n^{1/2})$
 - 为需要向所有节点分配数据的算法设置了下限
- 每个节点的最大边数:
 - 4 is the degree
- 恒定边长?
 - Yes



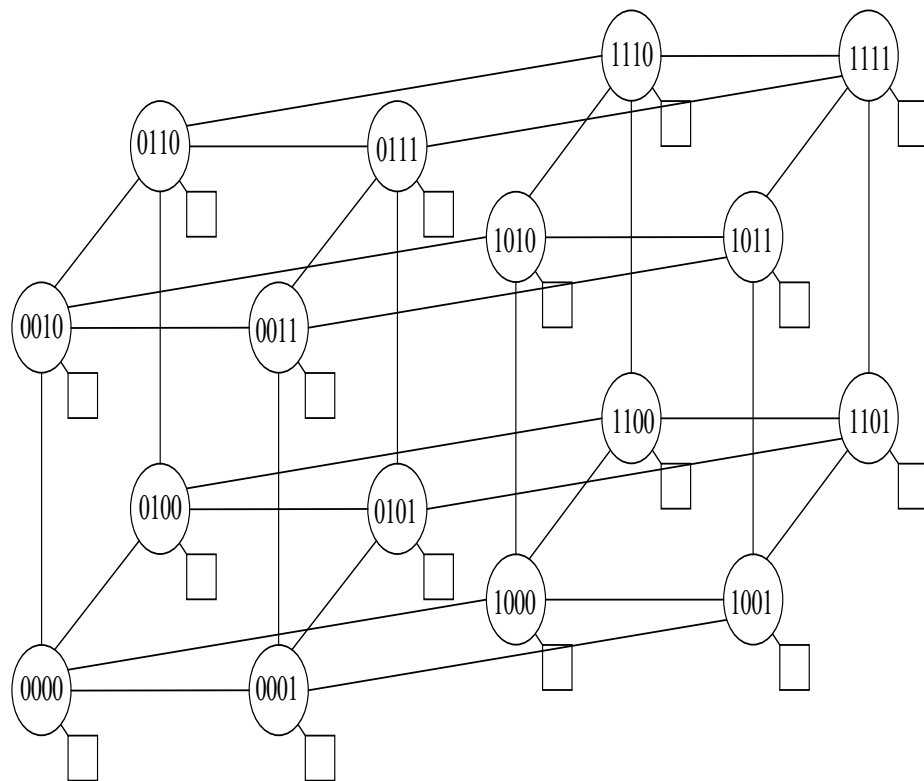
分布内存互联——超立方网络



分布内存互联——超立方网络

$n = 2^d$ Processors

- 直接拓扑结构
- $2 \times 2 \times \dots \times 2$ mesh
- 节点的数量是2的幂
- 节点*i*与*k*个节点相连, 这些节点的地址与*i*正好相差一个比特位置
- Example: $k = 0111$ is connected to 1111 , 0011 , 0101 , and 0110

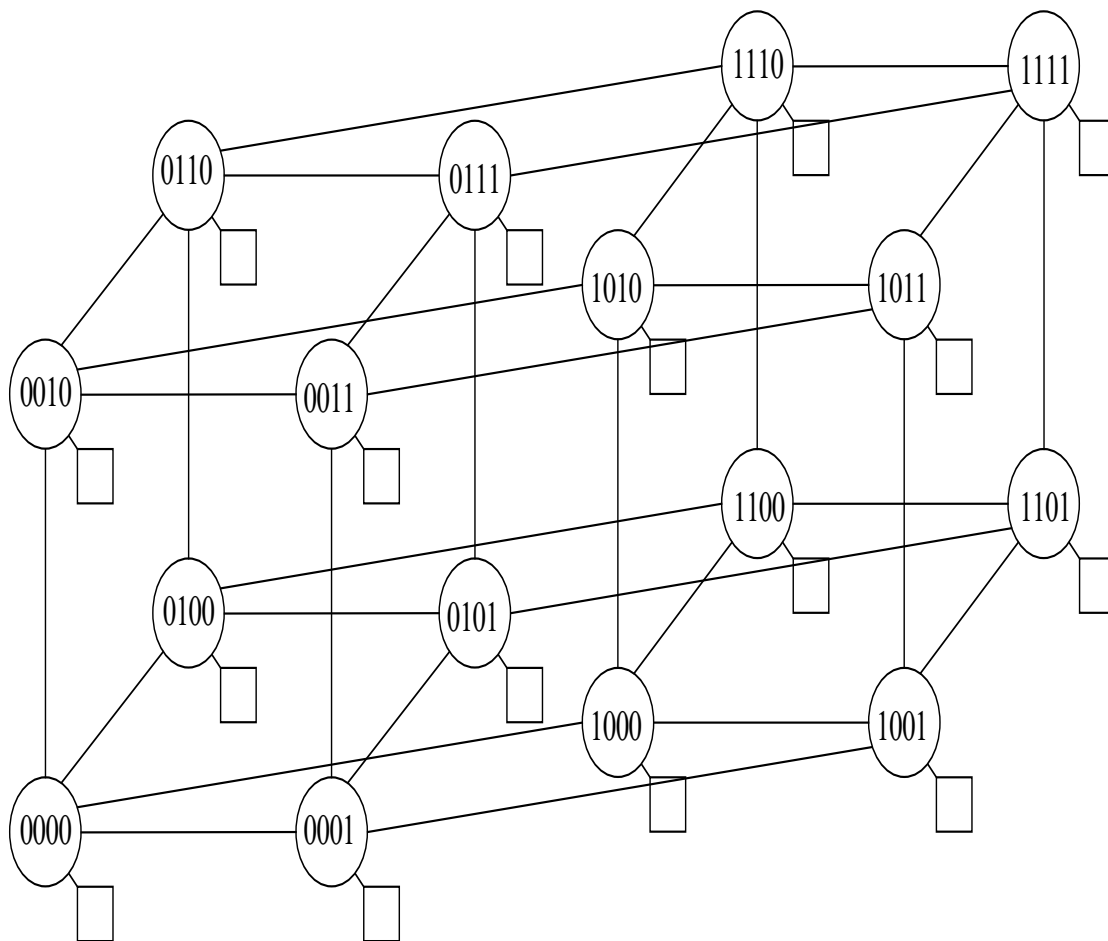


超立方网络—评估

$n = 2^d$ nodes

- 直径:
 - $d = \log n$
- 分割带宽:
 - $n / 2$
- 每个节点的边数:
 - $\log n$
- 恒定边长?
 - No.

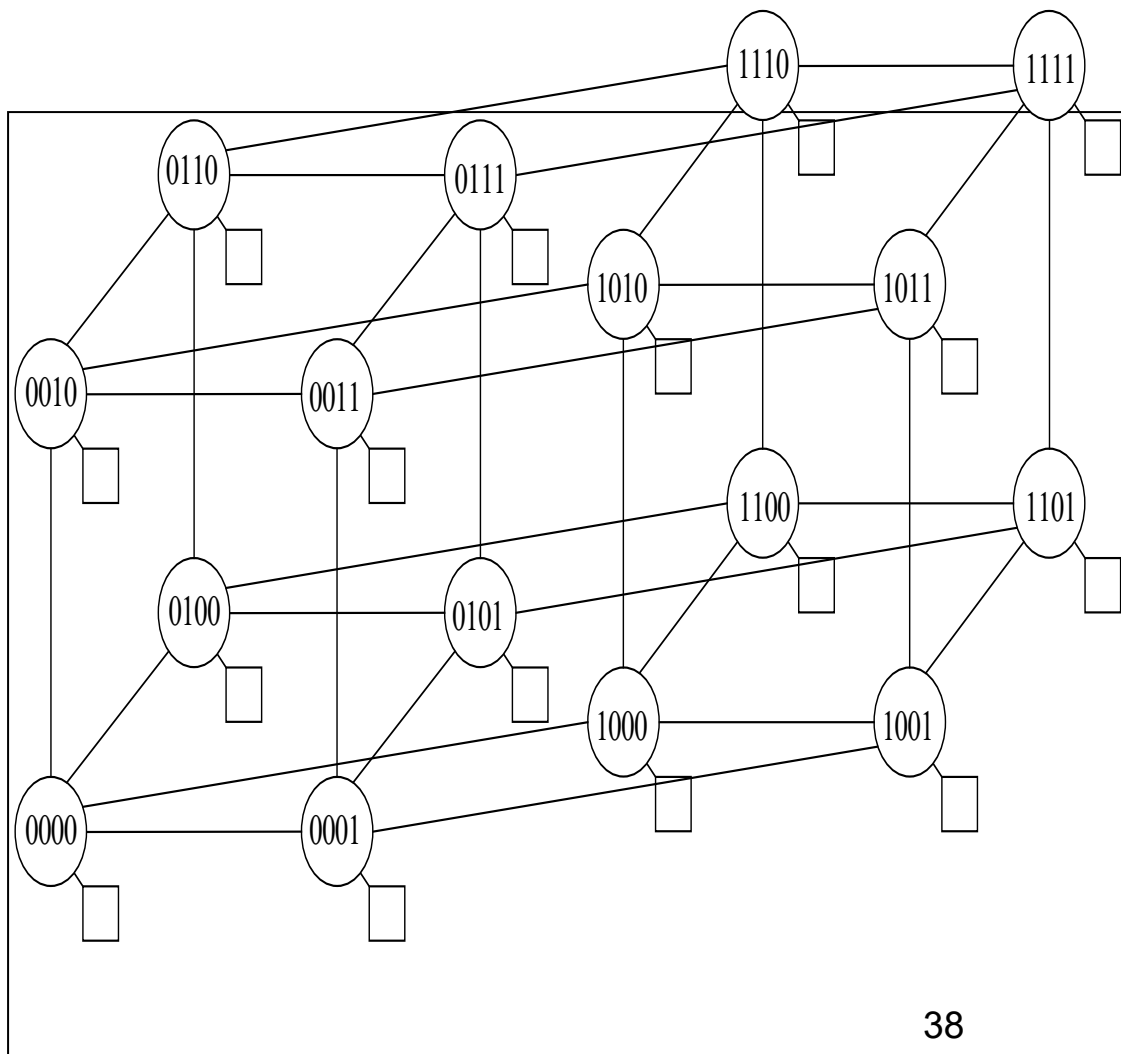
最长边的长度随着 n 的增加而增加



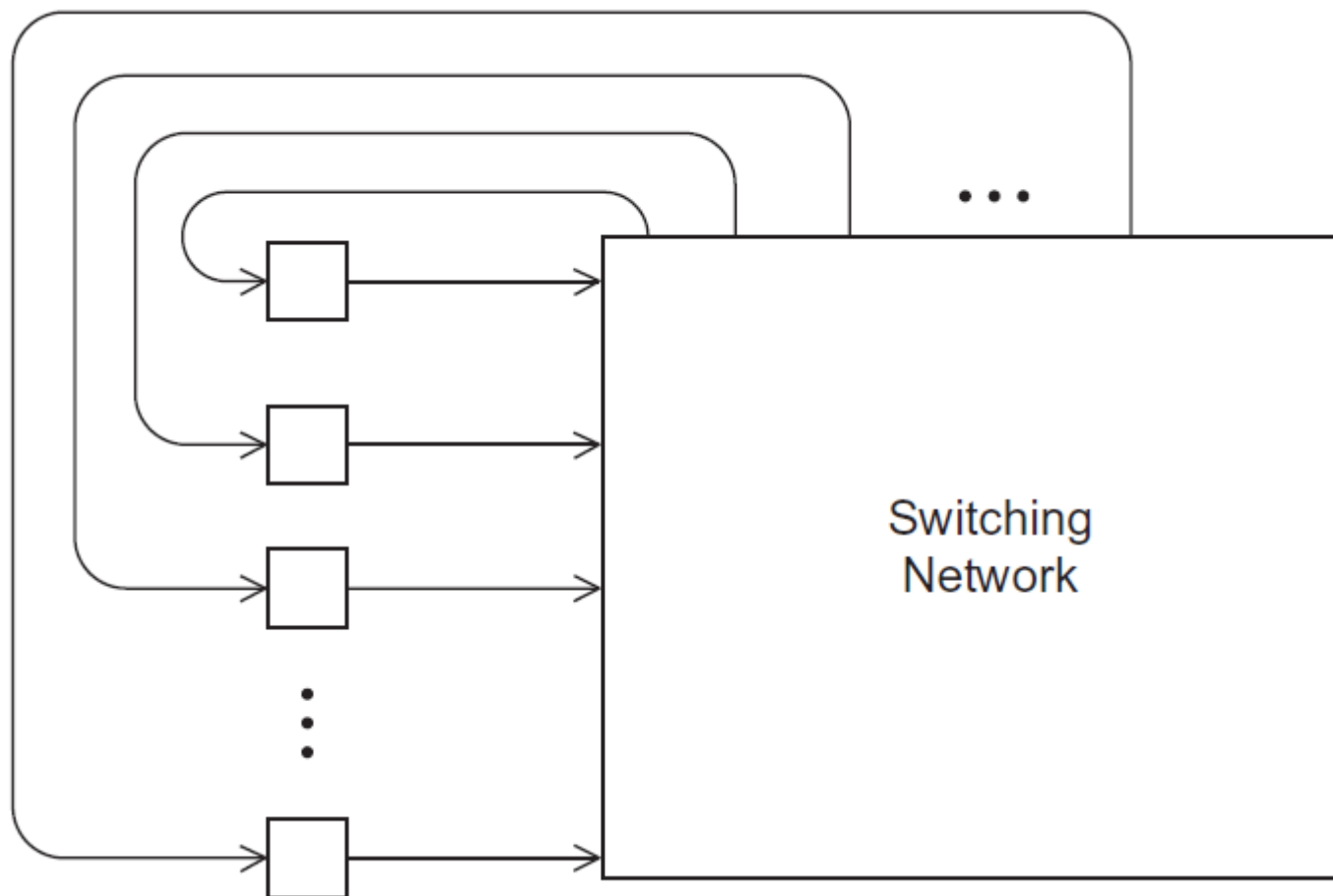
超立方网络—路由

• 比特翻转有助于设计网络路由

- Example: 从节点 2=0010 向节点 5=0101 发送一个信息
- 节点相差3位, 所以最短路径的长度为3.
 - 一条路径是 0010 → 0110 → 0100 → 0101
 - 通过在每一步翻转一个不同的位来获得

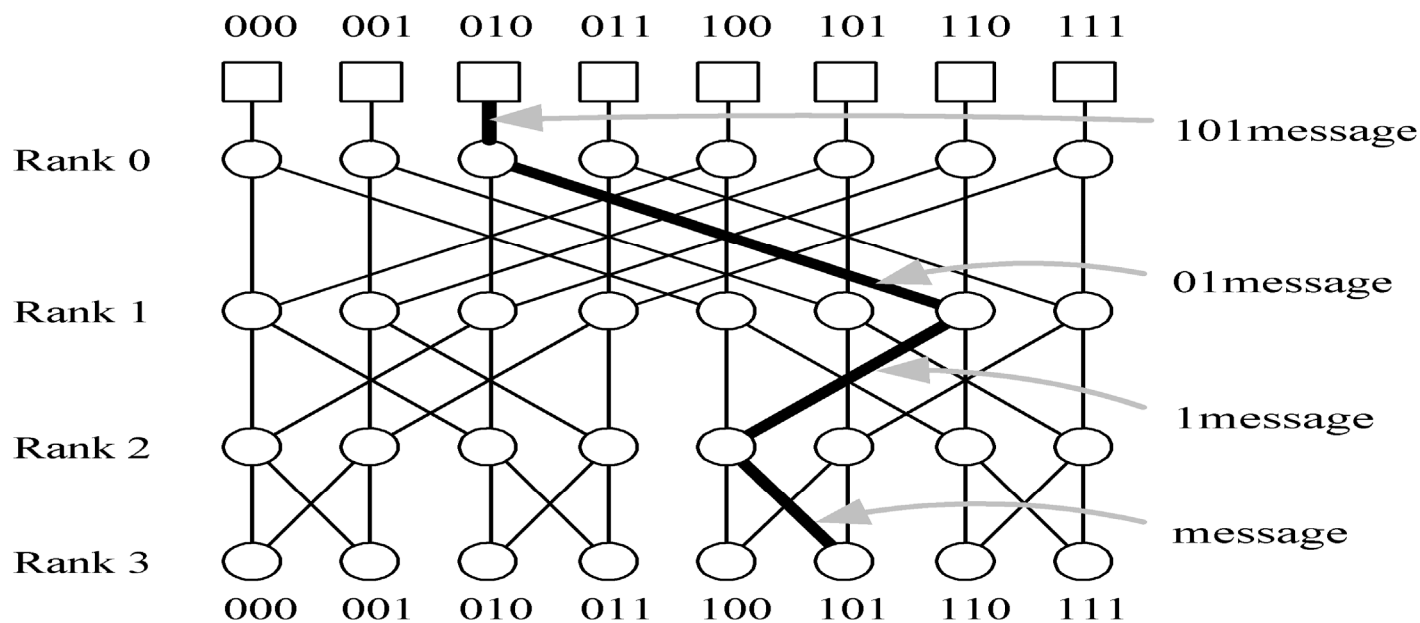


互联网络——间接互联



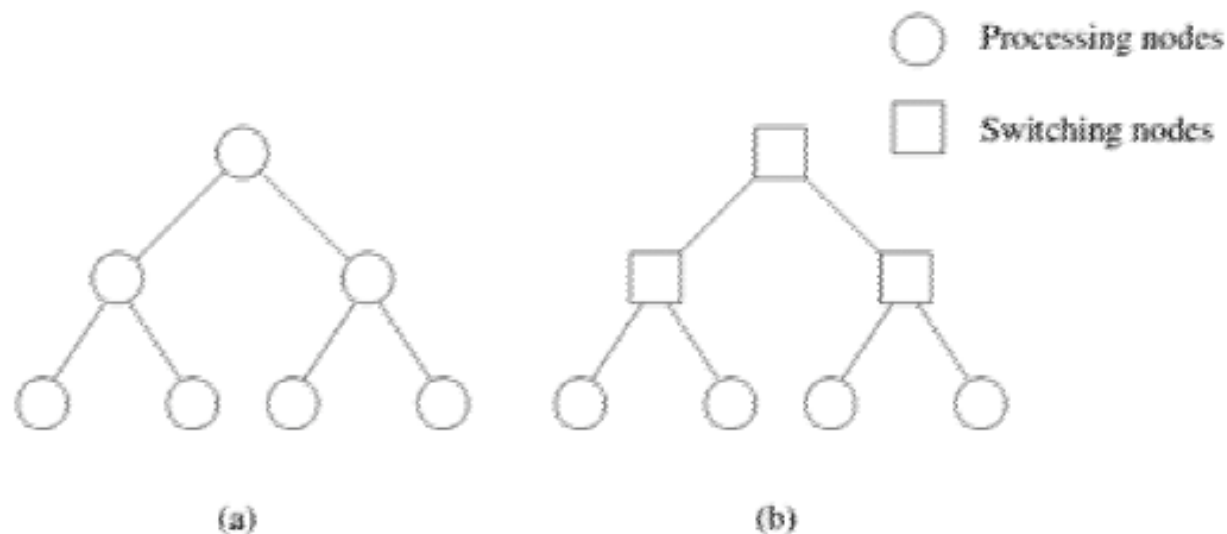
间接互联——蝶形网络

- 间接拓扑结构
- $n = 2^d$ 个进程节点, 由 $n(\log n + 1)$ 个交换节点连接而成



间接互联——树形网络

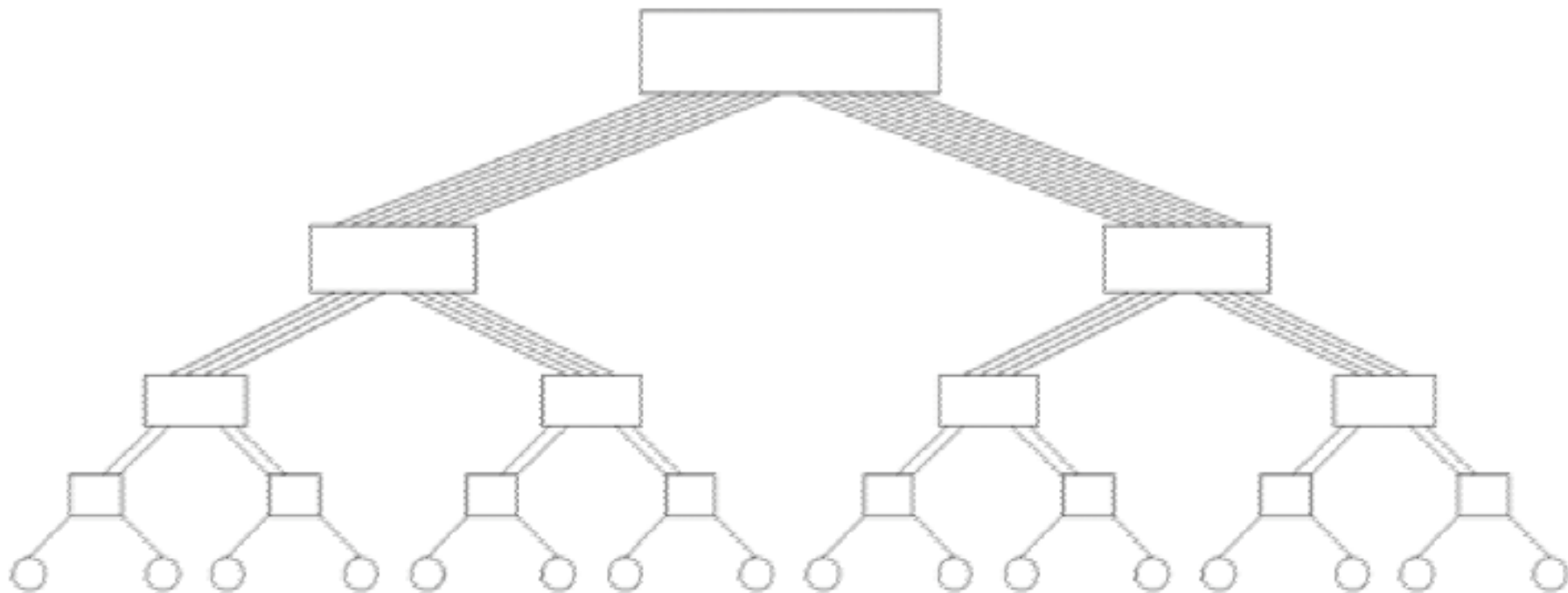
- 树状网络是指任何一对节点之间只有一条路径
- 静态树状网络在树的每个节点都有一个处理器
- 在动态树状网络中，中间层的节点是交换节点，叶子节点是处理器



完整的二叉树网络：(a) 静态树网络。和(b) 一个动态树状网络

Fat Tree Network

- 间接拓扑结构
- 在树的较高层次上使用更广泛的链接



Fat tree networks for 16 processors

More Definitions

$$\text{Message transmission time} = l + n / b$$

latency (seconds)

length of message (bytes)

bandwidth (bytes per second)

- 时延

- 从源头开始传输数据到目的地开始接收第一个字节之间所经过的时间

- 带宽

- 目的地在开始接收第一个字节后接收数据的速率

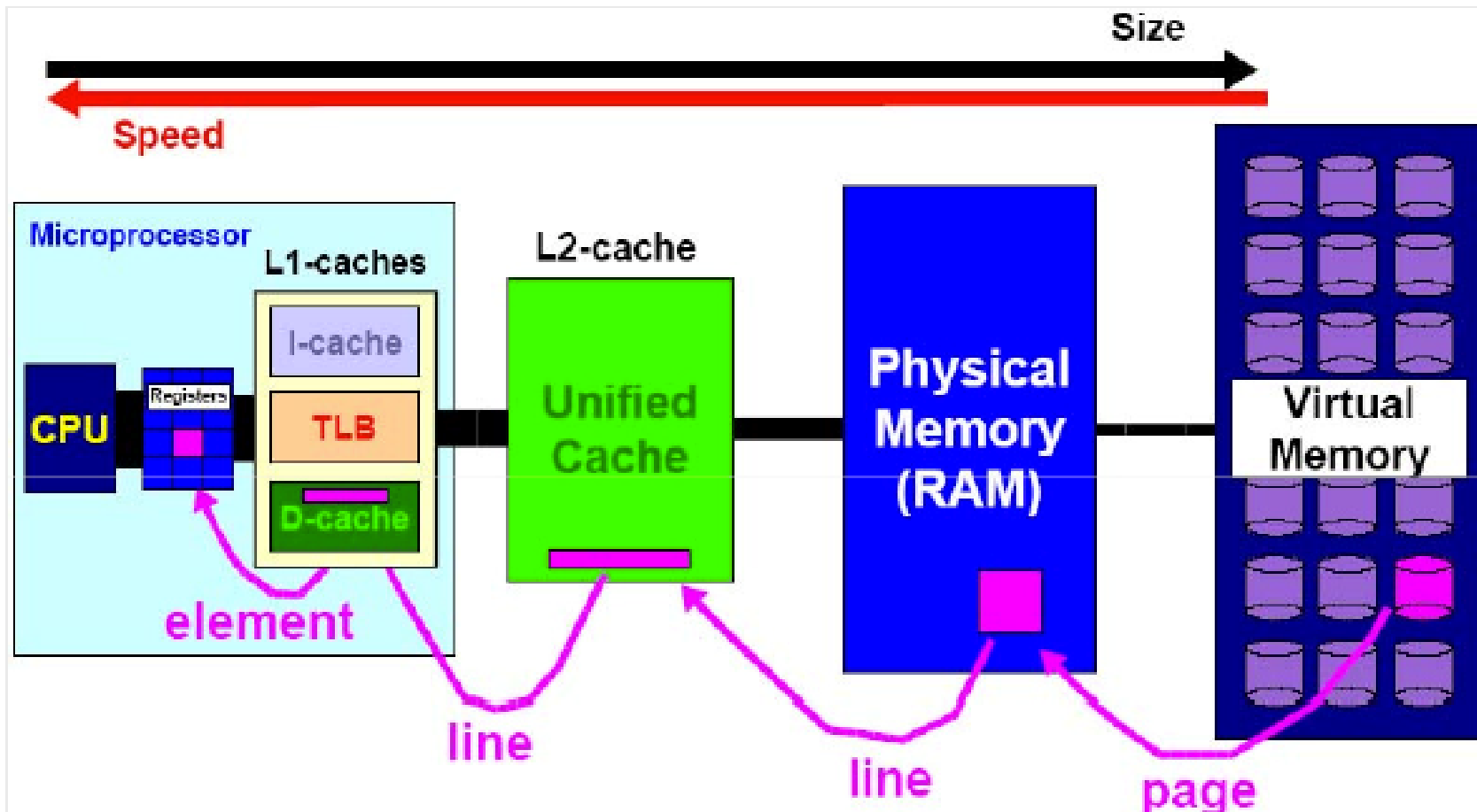


网络名称	网络规模	节点度	网络直径	对剖宽度	链路数	固定边长
线性	N	2	N-1	1	N-1	是
环形	N	2	$[N/2]$	2	N	是
2Dmesh	$N^{1/2} * N^{1/2}$	4	$2(N^{1/2}-1)$	$N^{1/2}$	$2(N-N^{1/2})$	是
2D环绕	$N^{1/2} * N^{1/2}$	4	$2[N^{1/2}/2]$	$2N^{1/2}$	2N	否
超立方	$N=2^n$	n	n	N/2	$n * N/2$	否
二叉树	N	3	$2(\log N - 1)$	1	N-1	是

Outline

- 弗林分类法
- 共享内存系统和分布式内存系统
- 互联网络
- 存储器层次结构和缓存一致性

存储器层次结构



- 多层次的内存层次结构
- 离CPU更近的层次访问速度更快
- 如果数据访问之间存在空间和时间上的定位，缓存存储器就能很好地工作

Memory Latency

Hierarchy	Processor clocks
register	1
L1 cache	2-3
L2 cache	6-12
L3 cache	14-40
Near memory	100-300
Far memory	300-900
Remote memory	$O(10^3)$
Message passing	$O(10^3)$ - $O(10^4)$



缓存的一致性

- 共享内存的一个重要问题
- 处理器可能会对同一位置进行缓存
- 如果一个处理器向该位置写入，所有其他处理器最终必须看到该写入

$X:=1$

Memory



缓存的一致性

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

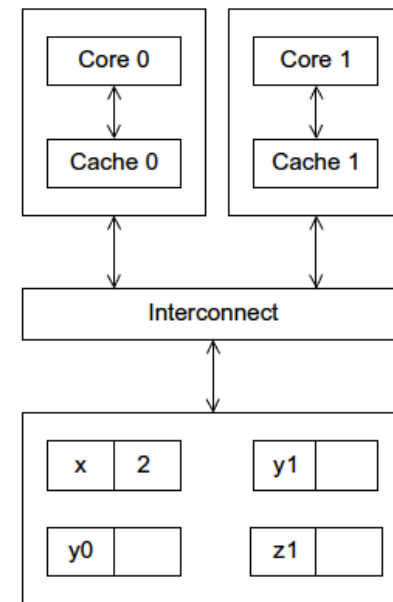
Time	Core 0	Core 1
0	y0 = x;	y1 = 3*x;
1	x = 7;	Statement(s) not involving x
2	Statement(s) not involving x	z1 = 4*x;

y0 eventually ends up = 2

y1 eventually ends up = 6

z1 = ???

- 程序员无法控制缓存以及它们何时被更新



A shared memory system with two cores and two caches

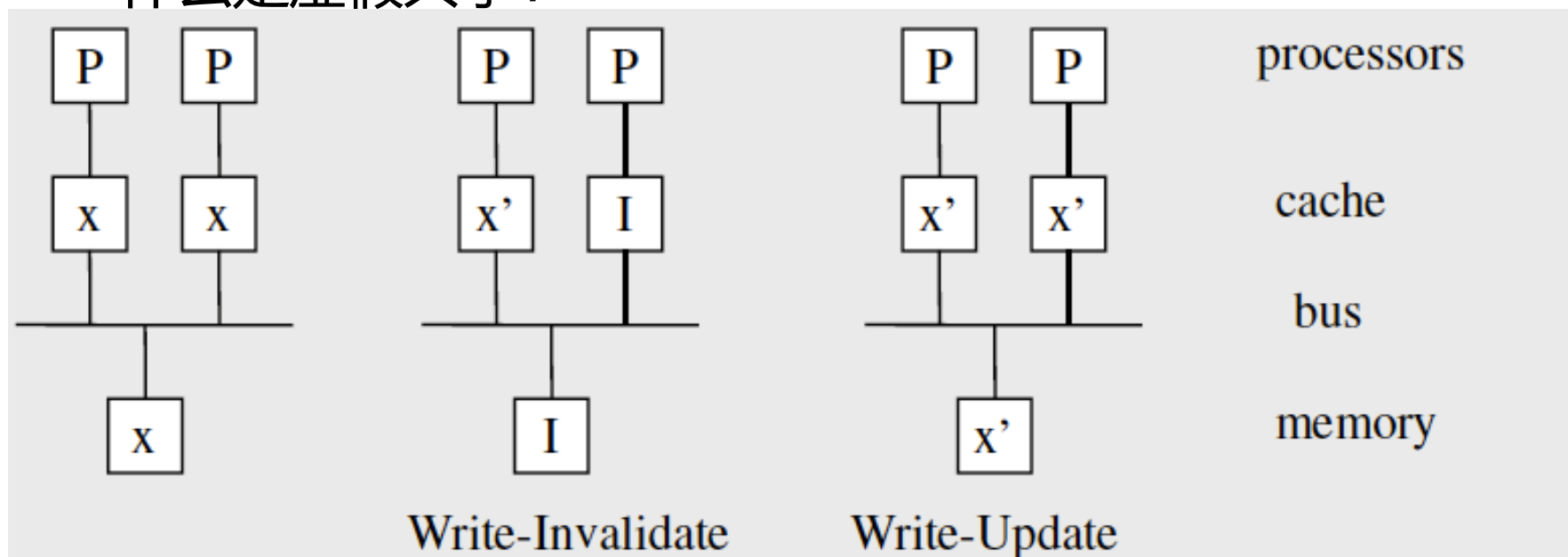
共享内存系统——缓存一致性问题

- 在多个缓存中复制数据可以减少处理器之间对共享数据值的争夺
- 但如何确保不同的处理器对同一地址有相同的值?
- 缓存一致性问题是指一个过时的值仍然存储在一个处理器的缓存中

缓存一致性协议

- 当改变变量的值时：使所有副本失效 (invalidate) 或 (update)
- 这两种协议都有虚假共享的开销

— 什么是虚假共享？



虚假共享

- 虚假共享，是指不同的处理器更新同一缓存线的不同部分的情)
- CPU的缓存是在缓存线上操作的，而不是单个变量
- Serial Version:

```
int i, j, m, n;  
double y[m];  
  
/* Assign y = 0 */  
.  
.  
.  
  
for (i = 0; i < m; i++)  
    for (j = 0; j < n; j++)  
        y[i] += f(i,j);
```



False Sharing

- Parallel Version:

```
/* Private variables */
int i, j, iter_count;

/* Shared variables initialized by one core */
int m, n, core_count
double y[m];

iter_count = m/core_count

/* Core 0 does this */
for (i = 0; i < iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i,j);

/* Core 1 does this */
for (i = iter_count+1; i < 2*iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i,j);
```

- 假设共享内存系统有2个核心， $m=8$ ， $y[0]$ 存储在一个缓存行的开头，一个缓存行可以存储8个双倍数， y 需要一个完整的缓存行。
- 当核心0和核心1同时执行它们的代码时会发生什么)

False Sharing

- 虚假的共享并不会导致不正确的结果。
- 然而，它可能会破坏一个程序的性能，因为它导致对内存的访问比必要的多得多)

• 内存连贯性/一致性

- 写直达 (Write-through) : 当该行被写入高速缓存时, 会被写入主内存中
- 写回 (Write-back) : 数据不会立即写入。缓存中的更新数据被标记为脏, 当缓存行被来自内存的新缓存行替换时, 脏行被写入内存



缓存一致性——策略比较

- **写失效 (Write Invalidate)** : 当一个处理器向其本地缓存写入时, 其他缓存的副本必须被无效化
 - 如果是写直达, 它也会更新内存中的副本
 - 如果是写回, 它使内存中的副本失效
- **写更新 (Write Invalidate)** : 当一个处理器写到它的本地缓存时, 其他缓存的副本必须立即更新
 - 如果是写直达, 它也会更新内存中的副本
 - 如果是写回, 数据并不立即写入。缓存中更新的数据被标记为脏, 当缓存行被来自内存的新缓存行替换时, 脏行被写入内存

问题：哪种策略更好？

- 取决于流量模式
 - A. 对一个地址的突发性写操作
 - B. 生产者-消费者
- 1. Invalidate is worse when _____?
- 2. Update is worse when _____?

现代机器使用无效协议作为默认值(Why?)

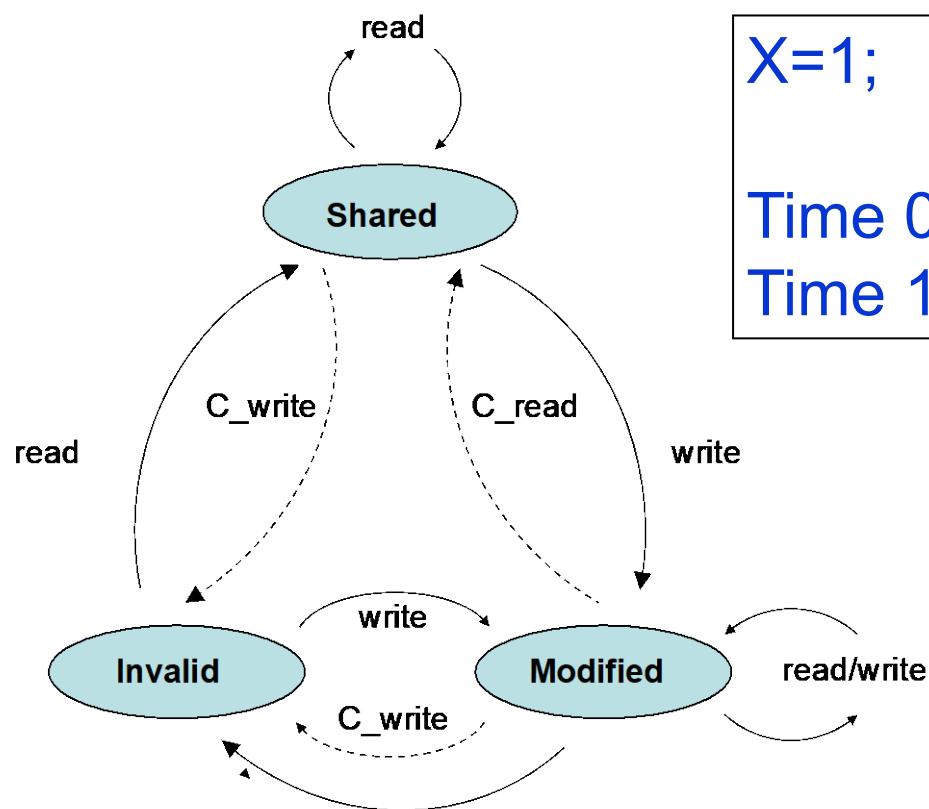
写无效协议

- 一个数据项的每个副本都与一个状态相关联
- 典型状态MSI：共享、无效或修改（**S**hared, **I**nvalid, or **M**odified）
 - **S**hared: 缓存中存在多个有效的副本
 - **I**nvalid: 数据副本无效
 - **M**odified : 只存在一个有效副本

Note: 状态集不是唯一的。也可以定义其他状态 (例如: MESI)

写无效协议

- 一个简单的MSI一致性协议的状态图
- 实线表示处理器动作，虚线表示一致性动作



$X=1;$

P0

P1

Time 0

load x

load x

Time 1

write #3, x

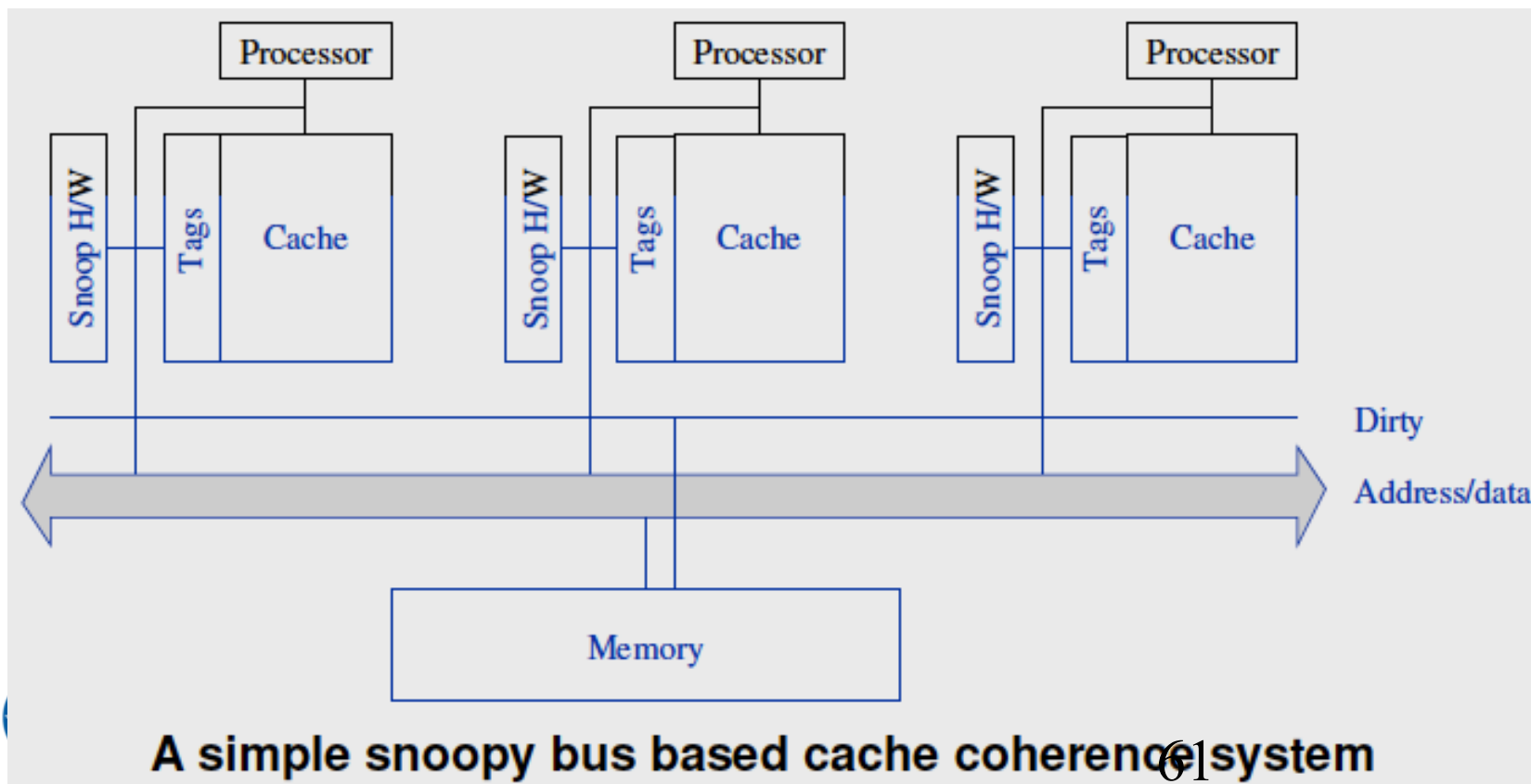


基于监听的缓存一致性

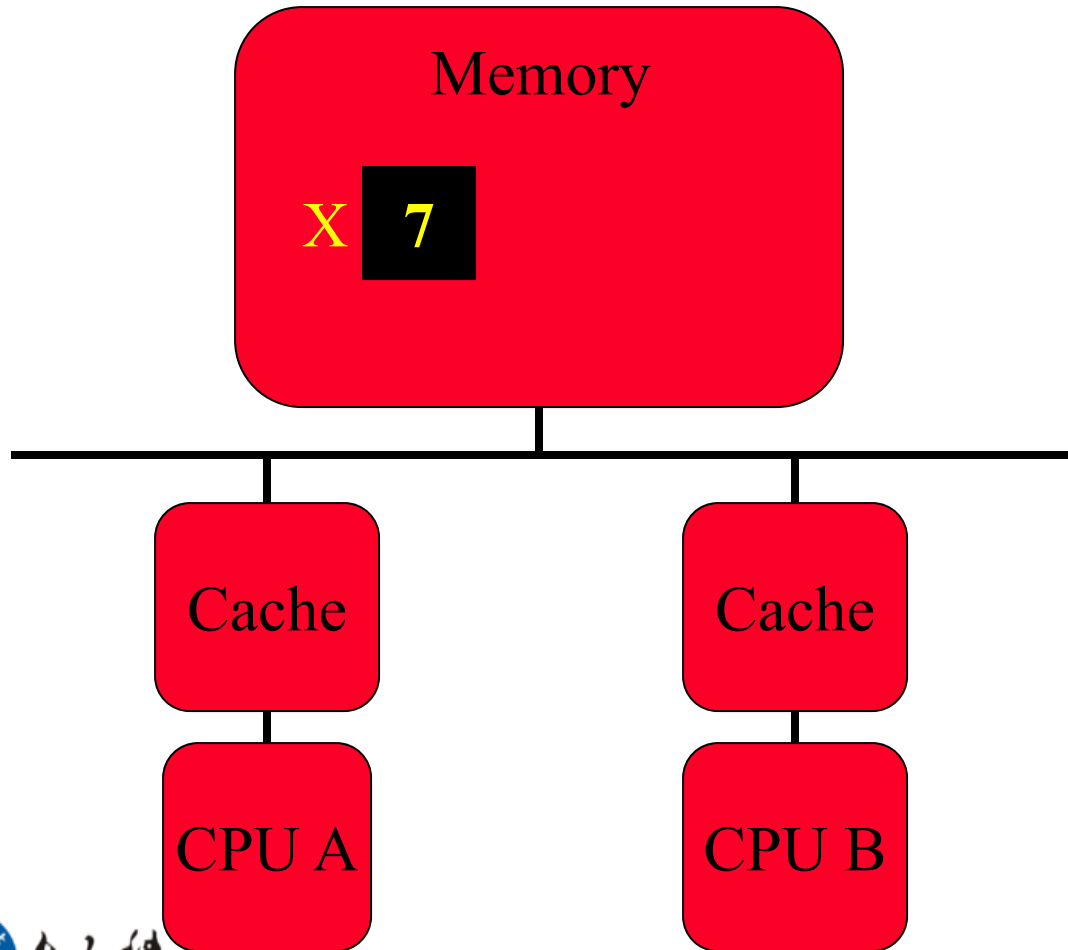
- 核心共享一条总线
- 在总线上传输的任何信号都可以被连接到总线上的所有内核 "看到"
- 当内核0更新存储在其缓存中的x的副本时，它也在总线上广播这一信息
- 如果核1在 "监听" 总线，它就会看到x已经被更新了，它可以把它的x的副本标记为无效
- 缓存一致性的最常见的解决方案

Snoopy Cache Systems

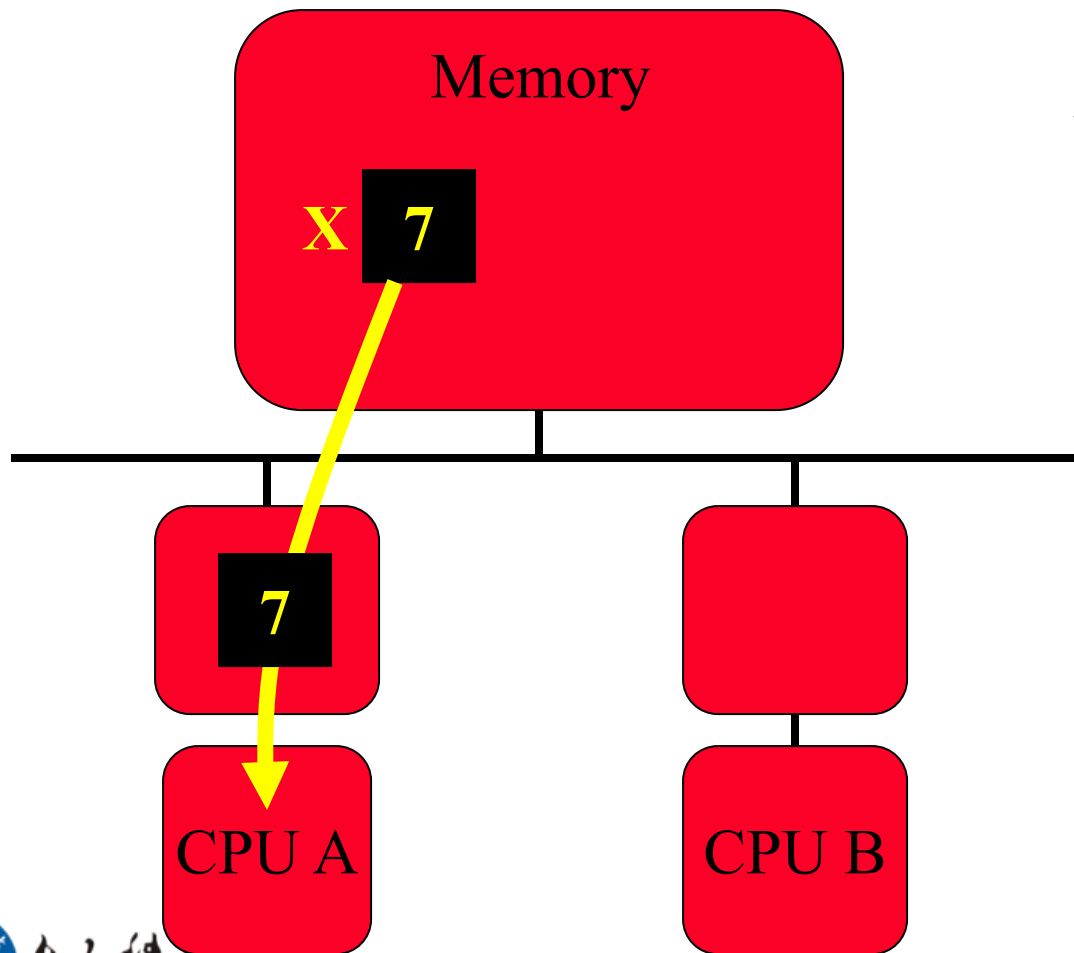
- 如何将invalidates信息发送到正确的处理器上？
 - 广播所有的invalidates数据和read请求
 - Snoopy缓存监听并在本地执行一致性操作



举例



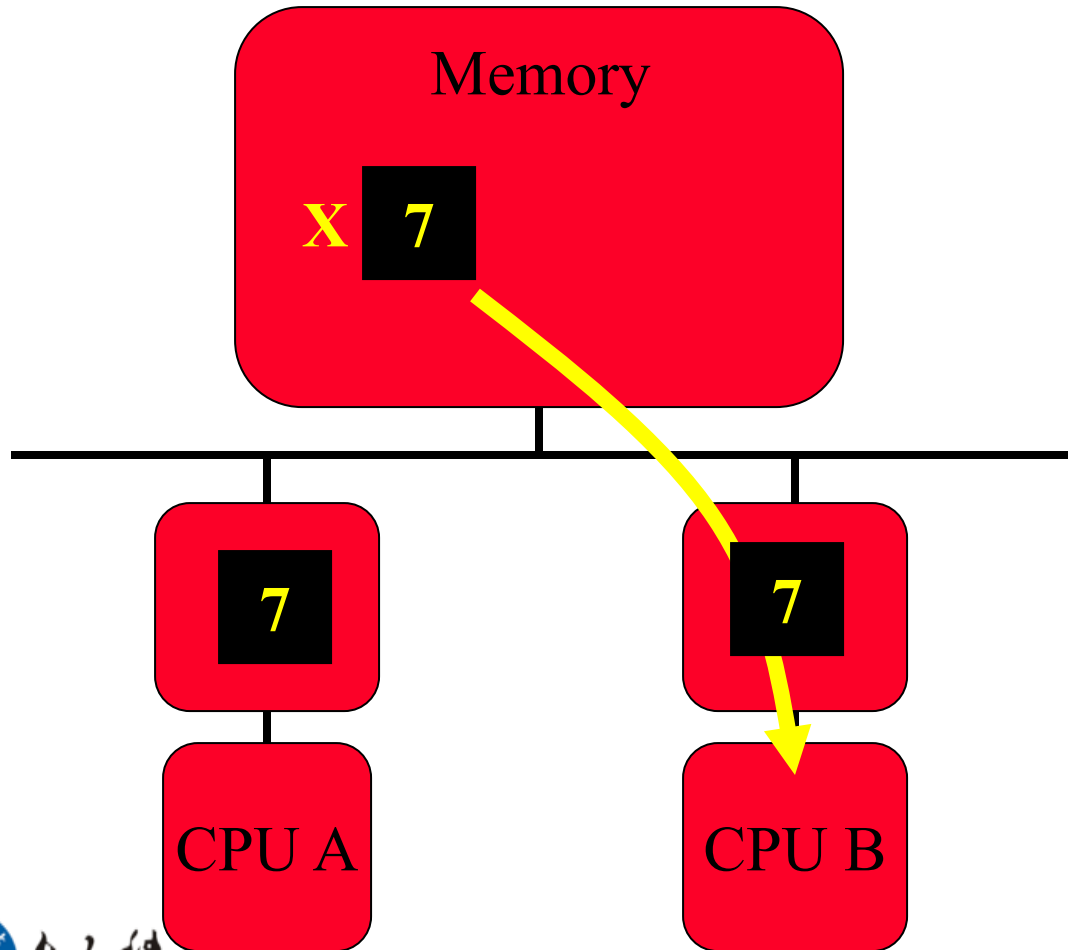
举例



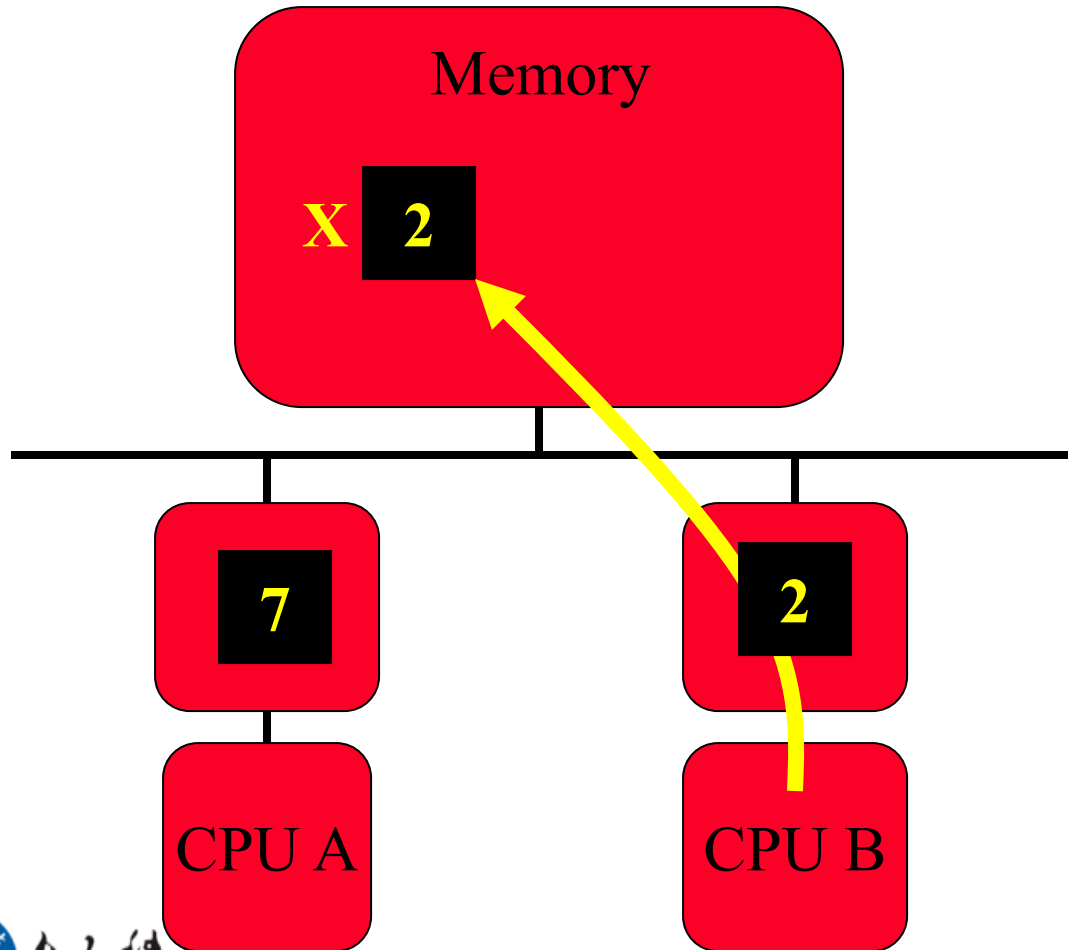
从内存中读取



举例

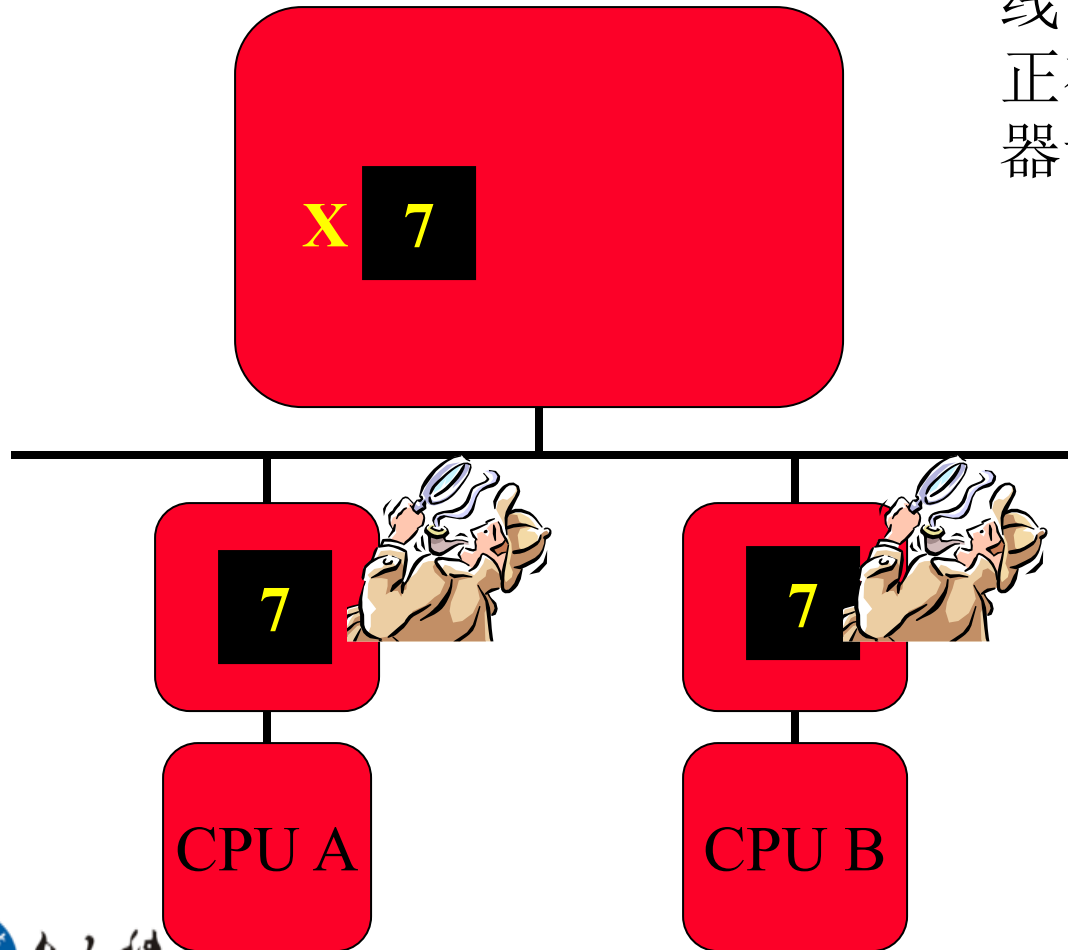


举例

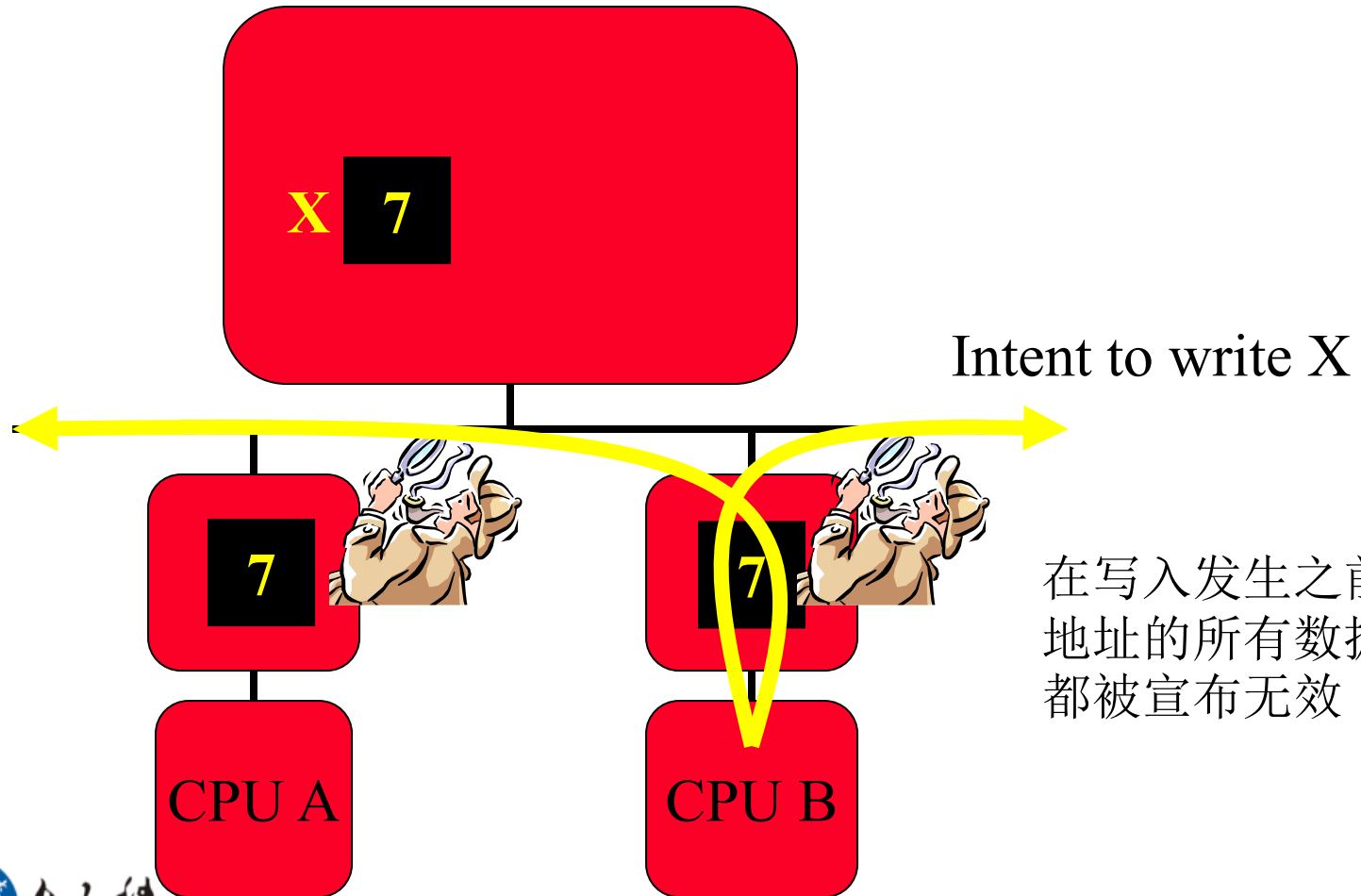


Write Invalidate Protocol

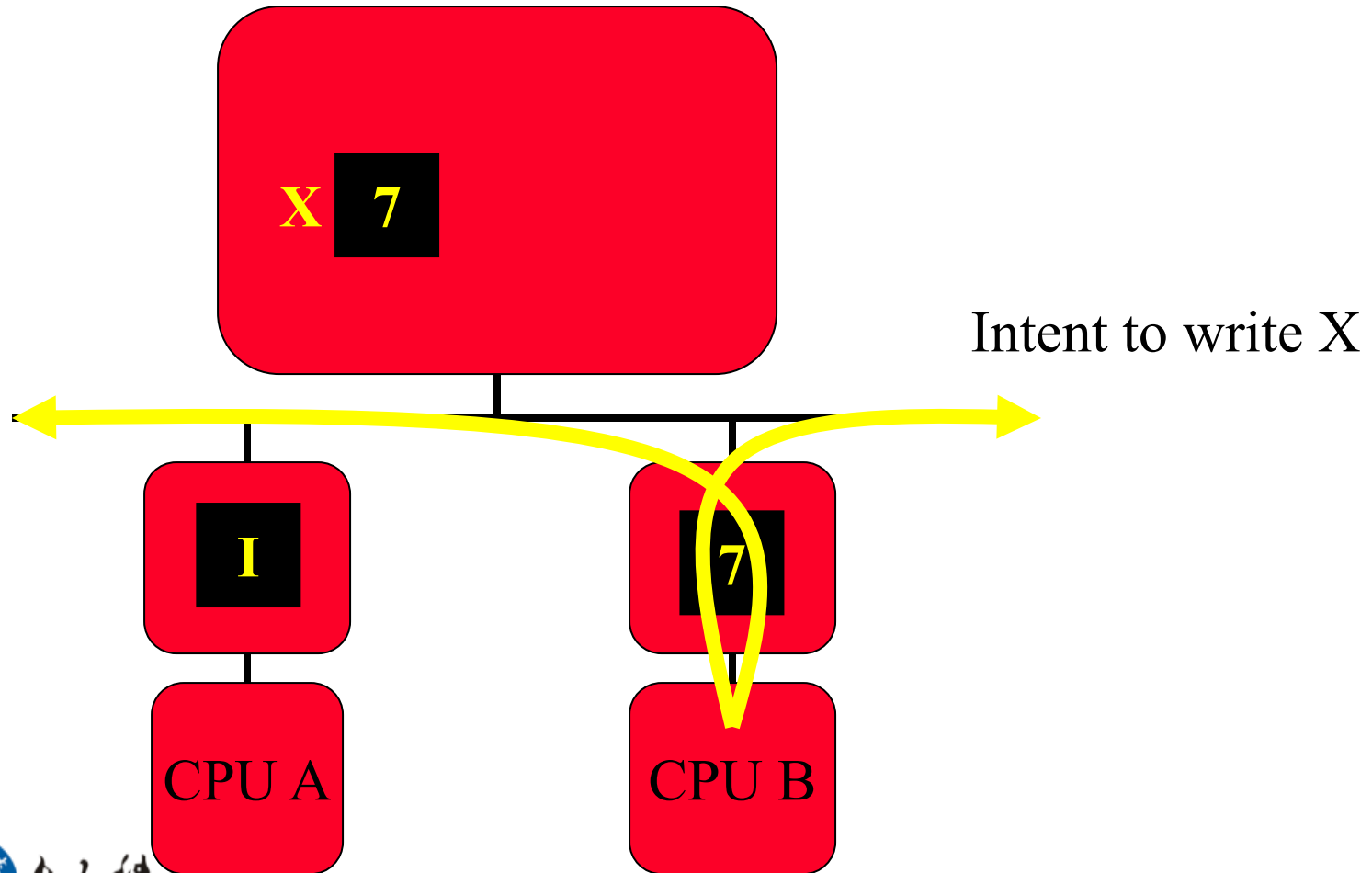
缓存控制监视器监听总线，以查看哪个缓存块正在被正在被其他处理器请求



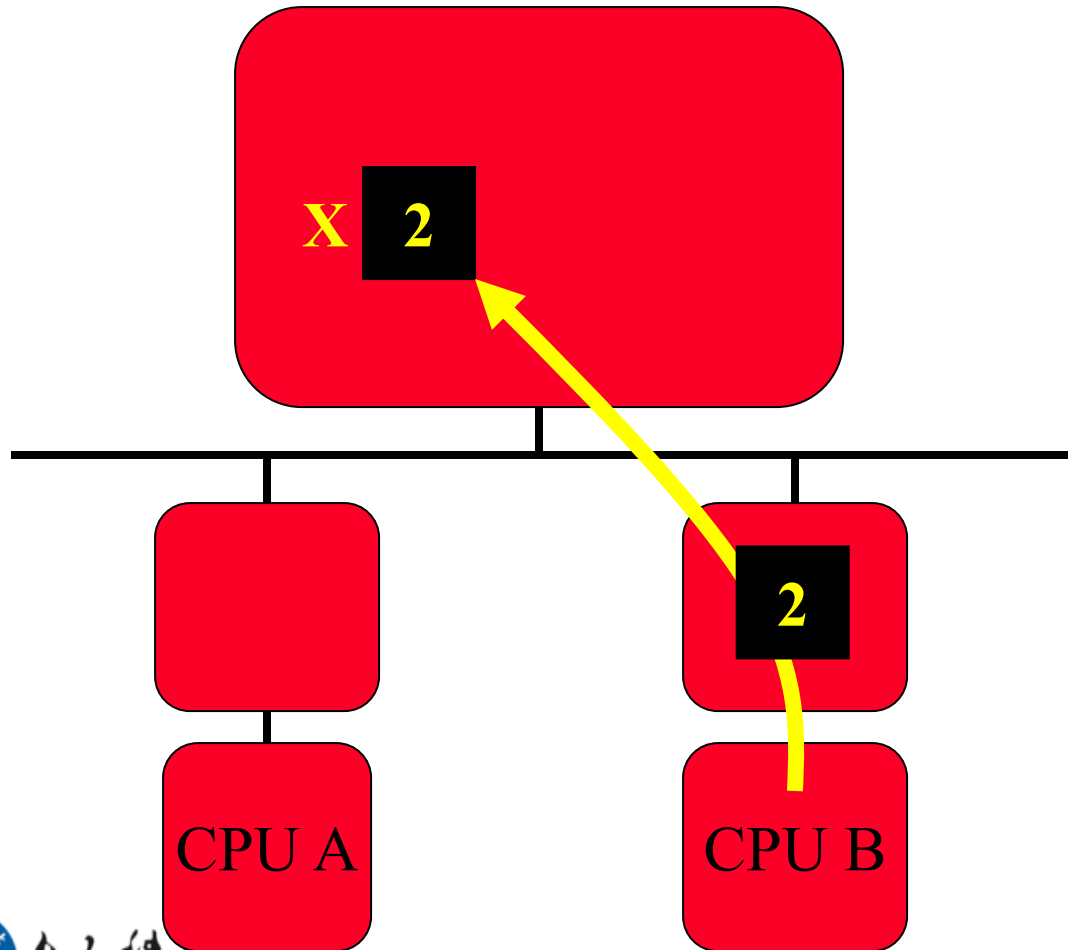
Write Invalidate Protocol



Write Invalidate Protocol



Write Invalidate Protocol



当另一个处理器试图从缓存中的这个位置读取时，它将收到一个缓存缺失错误，并将不得不从主内存中刷新

Operation of Snoopy Caches

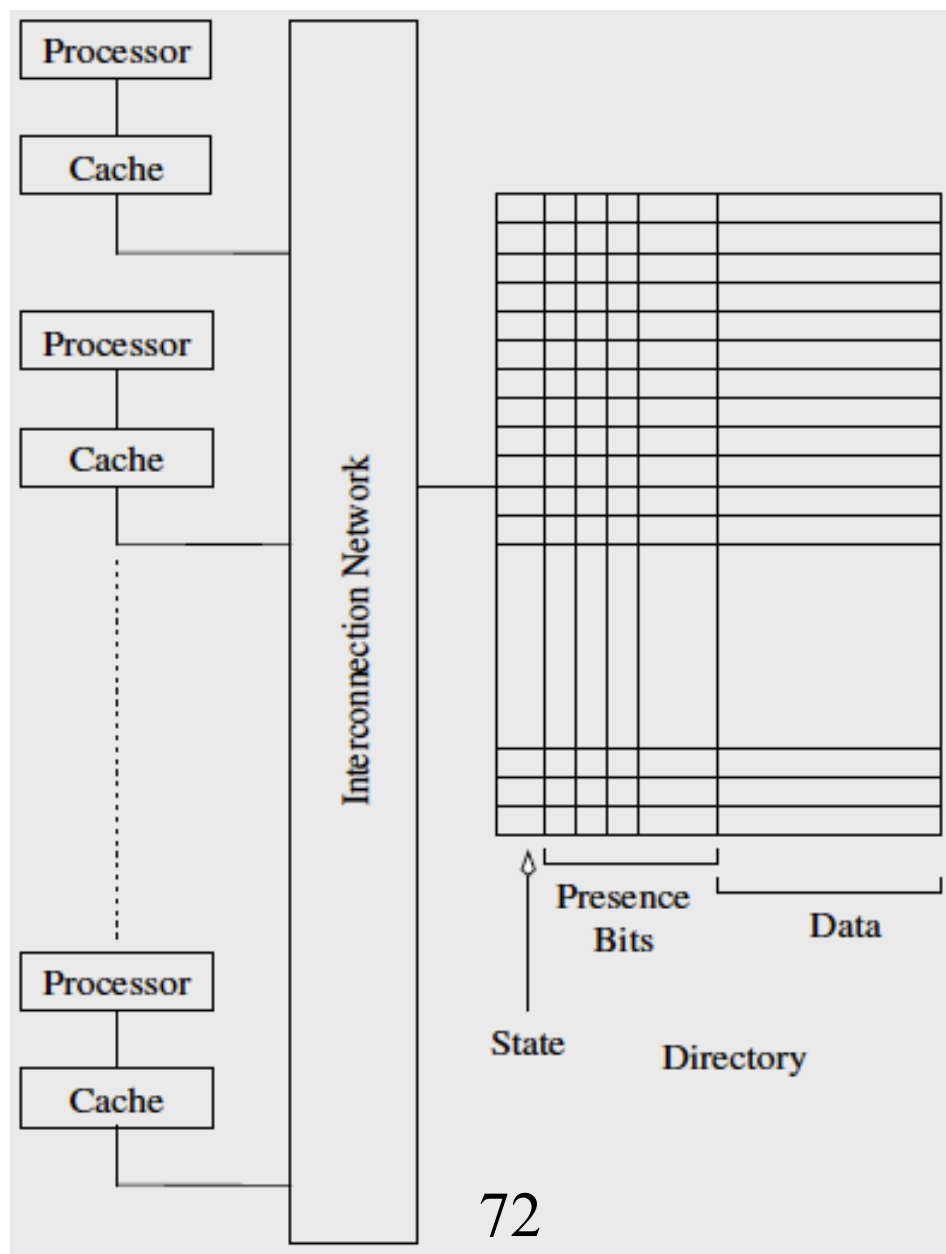
- 一旦一个数据点被标记为脏
 - 所有后续操作都可以在本地缓存中进行
 - 不需要外部流量
- 过渡到所有缓存中的共享状态
 - 所有后续的读取操作都变成了本地操作
- 如果多个处理器读取和更新相同的数据，就会出现一个基本的瓶颈
 - 在总线上产生一致性请求
 - 总线的BW限制: 对每秒的更新施加了限制

The Cost of Coherence

- Snoopy caches
 - 每个相干操作被发送到所有处理器
 - 这有什么问题?
- 为什么不把一致性请求只发给那些需要被通知的处理器?

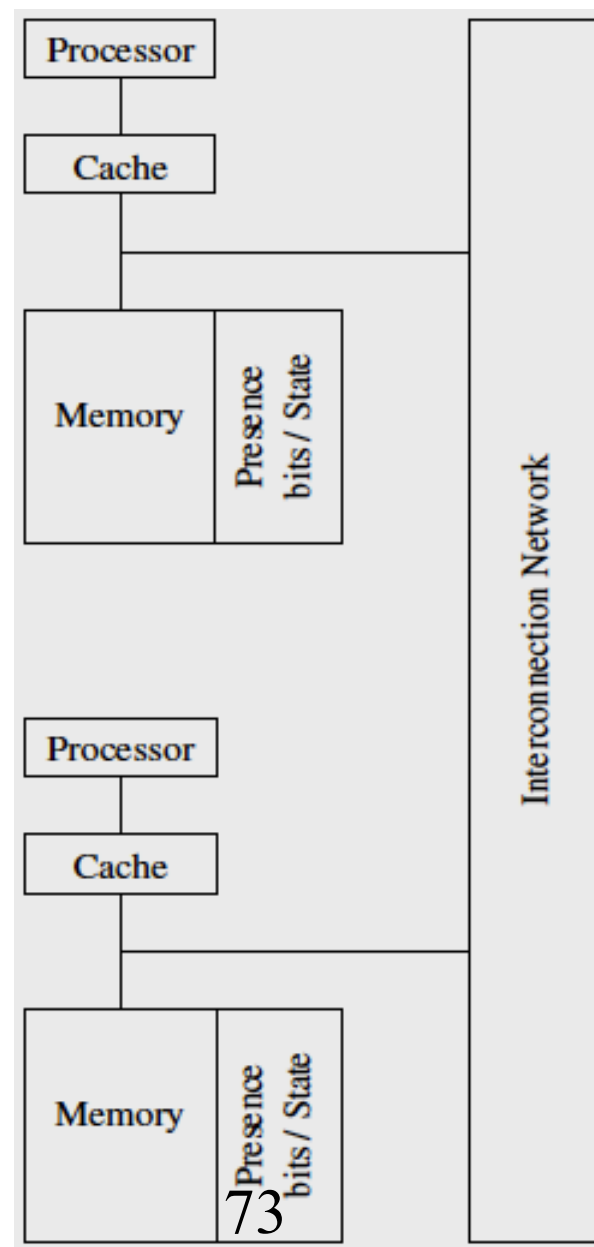
基于目录的缓存一致性

- 共享状态被保存在目录中
- 代表缓存块和它们被缓存的处理器位图



分布式目录

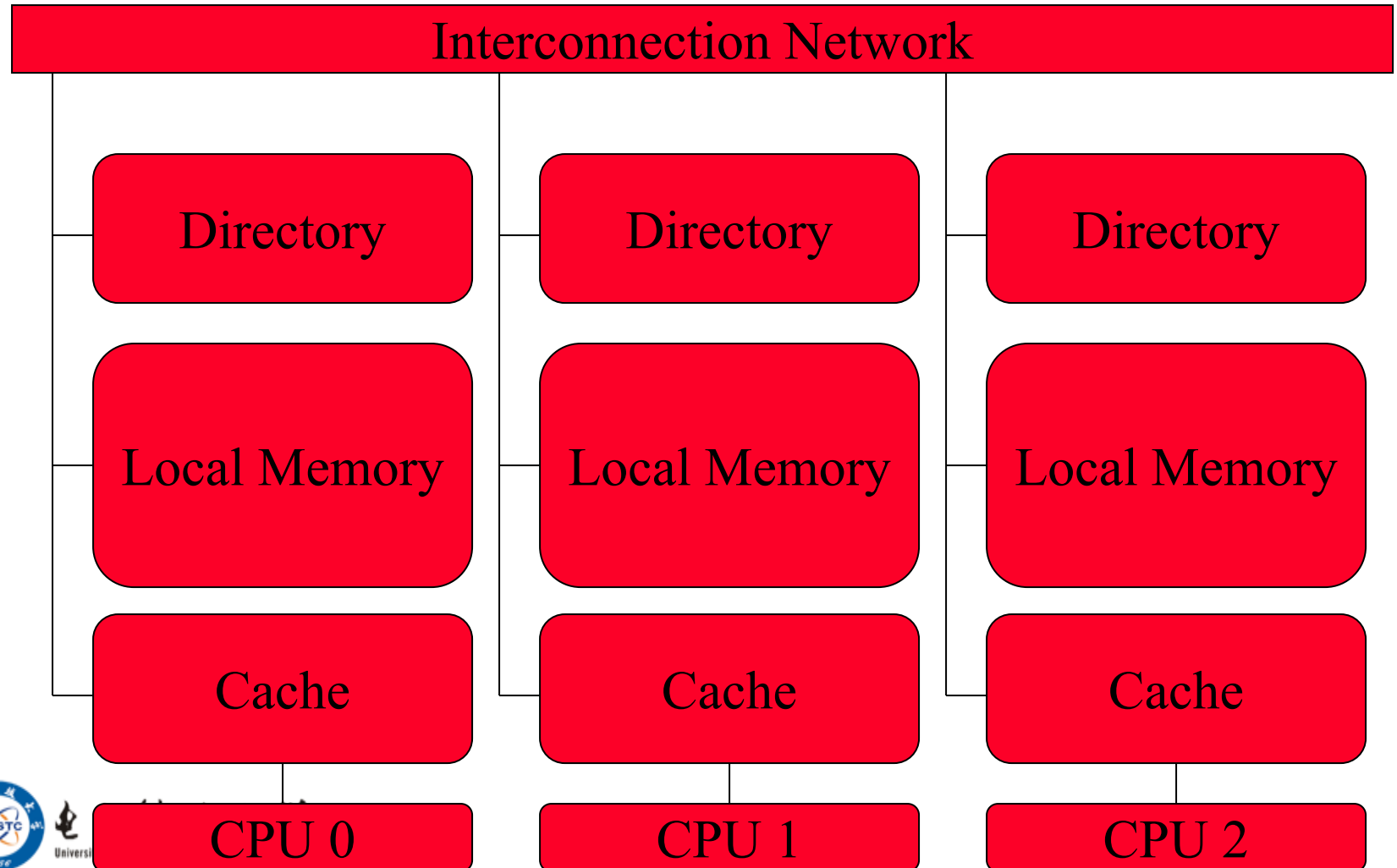
- 分布式目录计划的性能:
 - 更具可扩展性
 - 允许 $O(p)$ 同时进行一致性操作
 - 底层网络必须承载所有的一致性请求
- 网络的时延和带宽成为主要的性能瓶颈



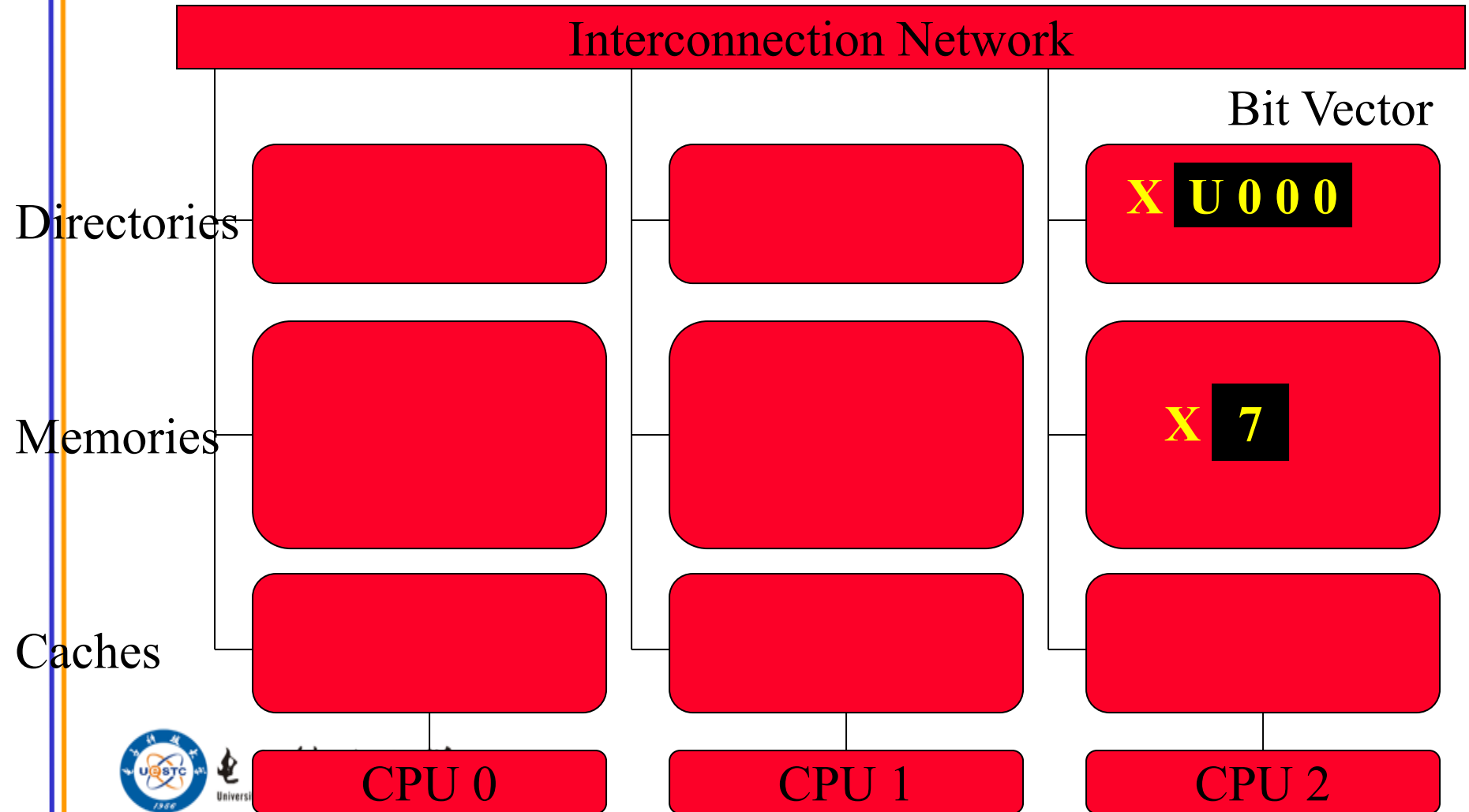
Sharing Status

- Uncached -- (用“U”表示)
 - 不在任何处理器的缓存中的块
- Shared – (用“S”表示)
 - 由一个或多个处理器缓存
 - 只读
- Exclusive – (用“E”表示)
 - 恰好由一个处理器缓存
 - 处理器已经写入块中
 - 内存中的拷贝已经过时

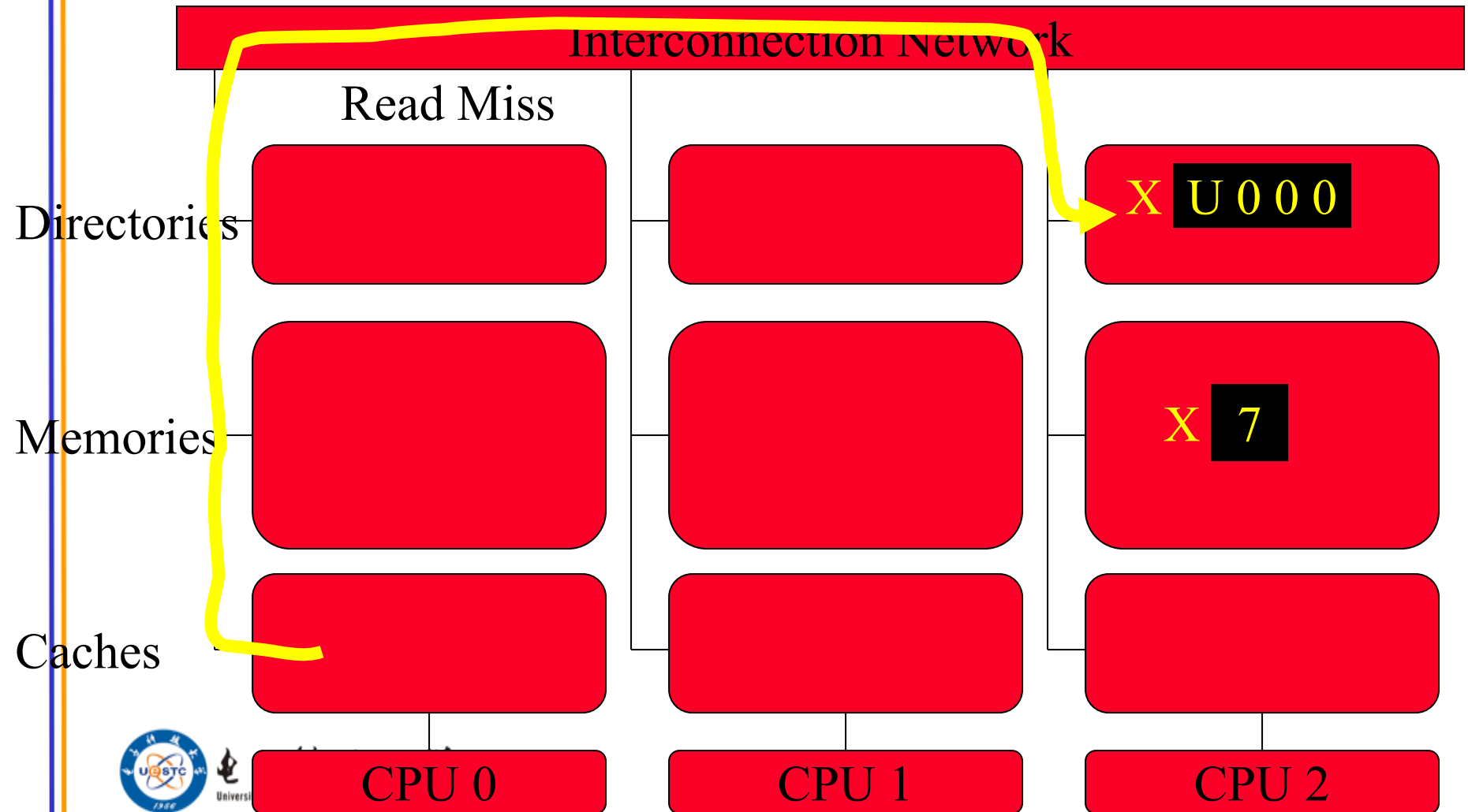
Directory-based Protocol - step1



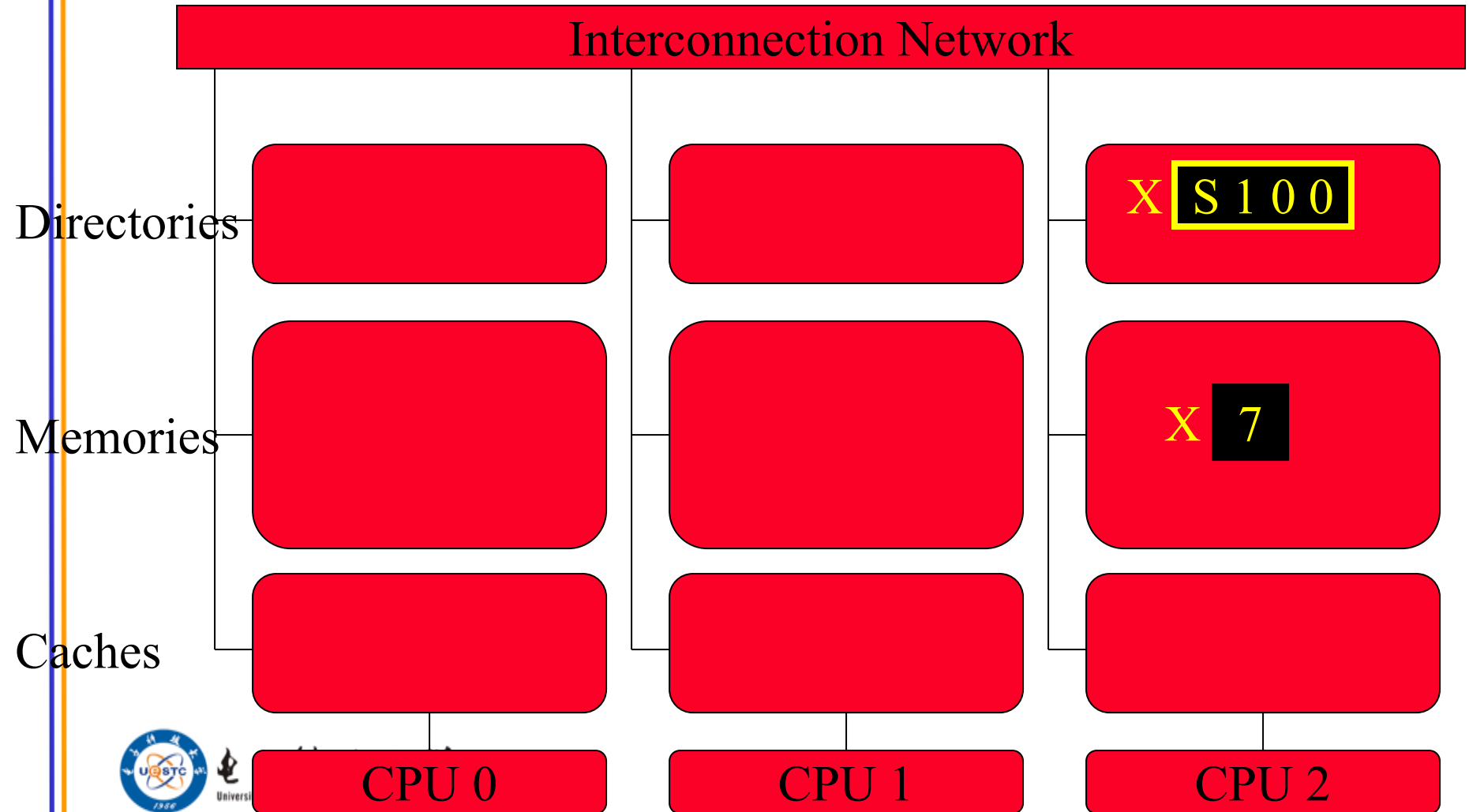
X has value 7 – step 2



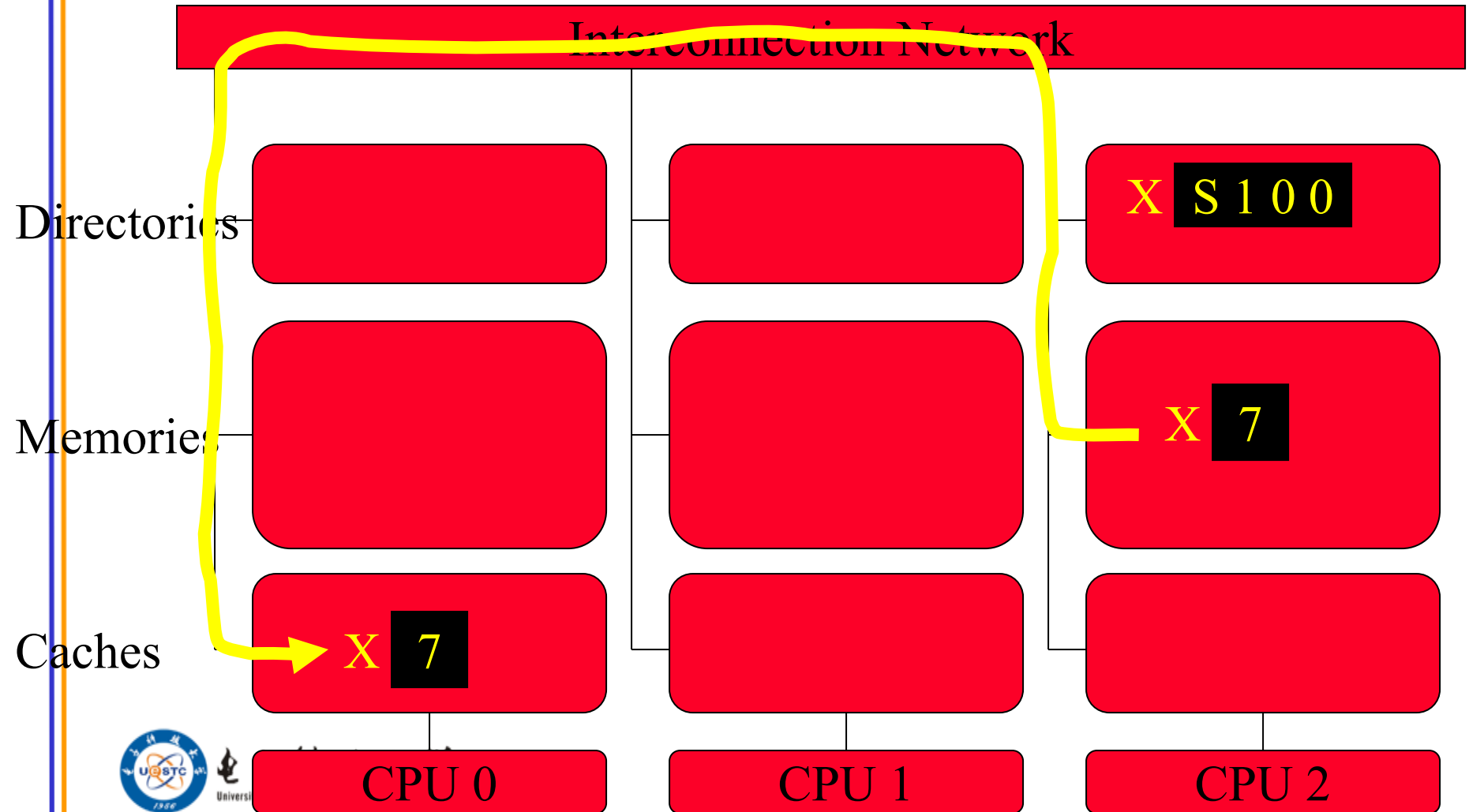
CPU 0 Reads X – step 3



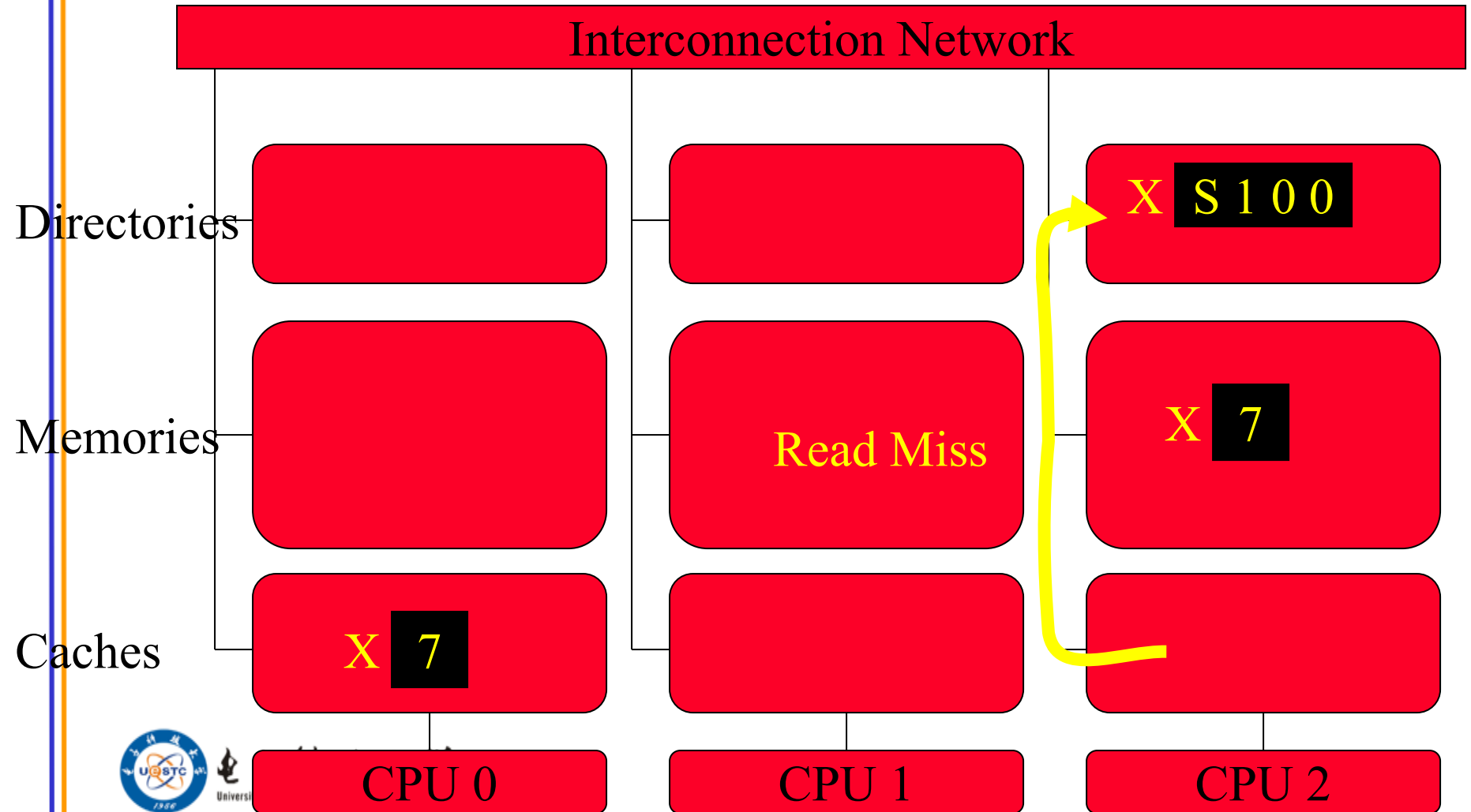
CPU 0 Reads X –step 4



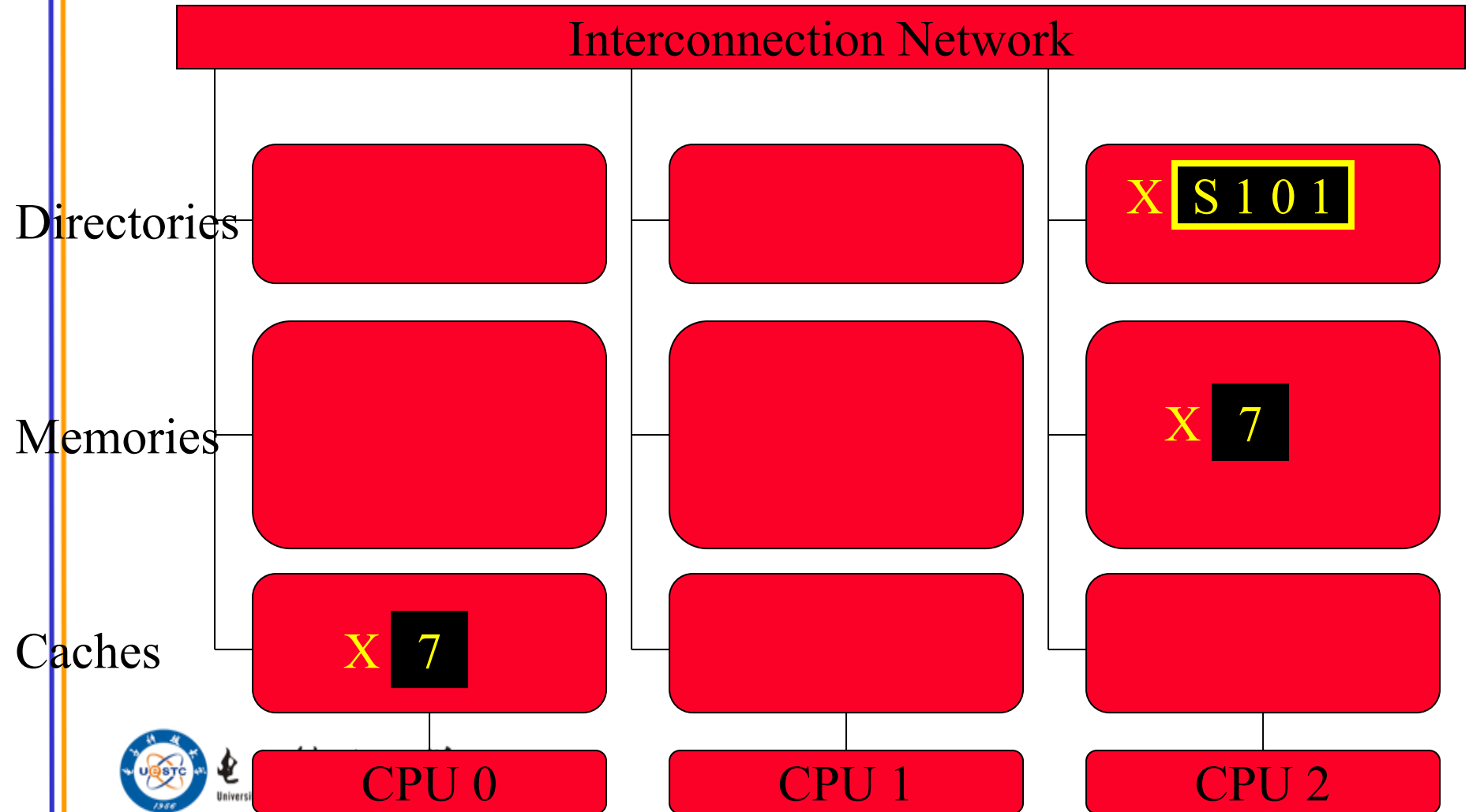
CPU 0 Reads X –step 5



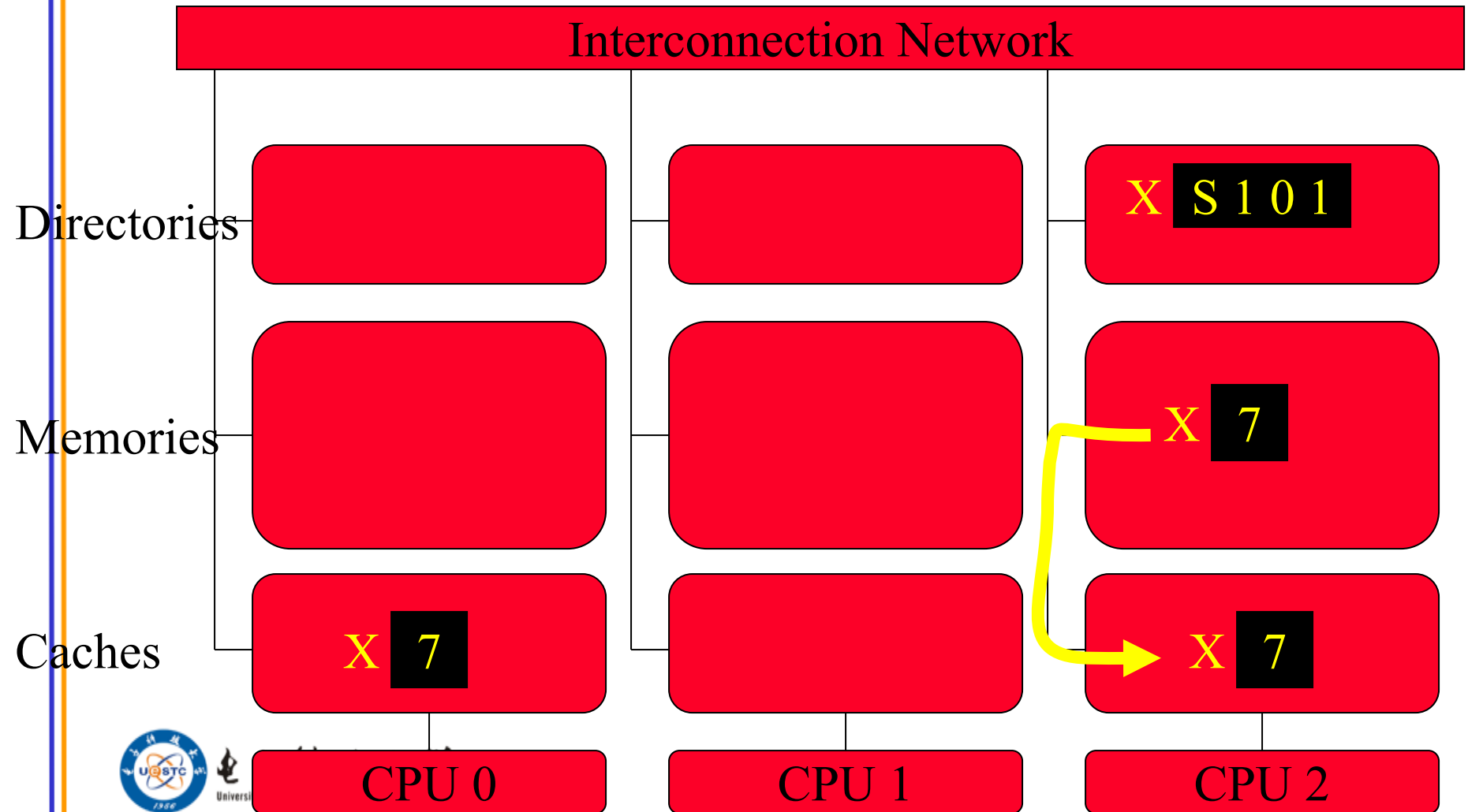
CPU 2 Reads X – step 6



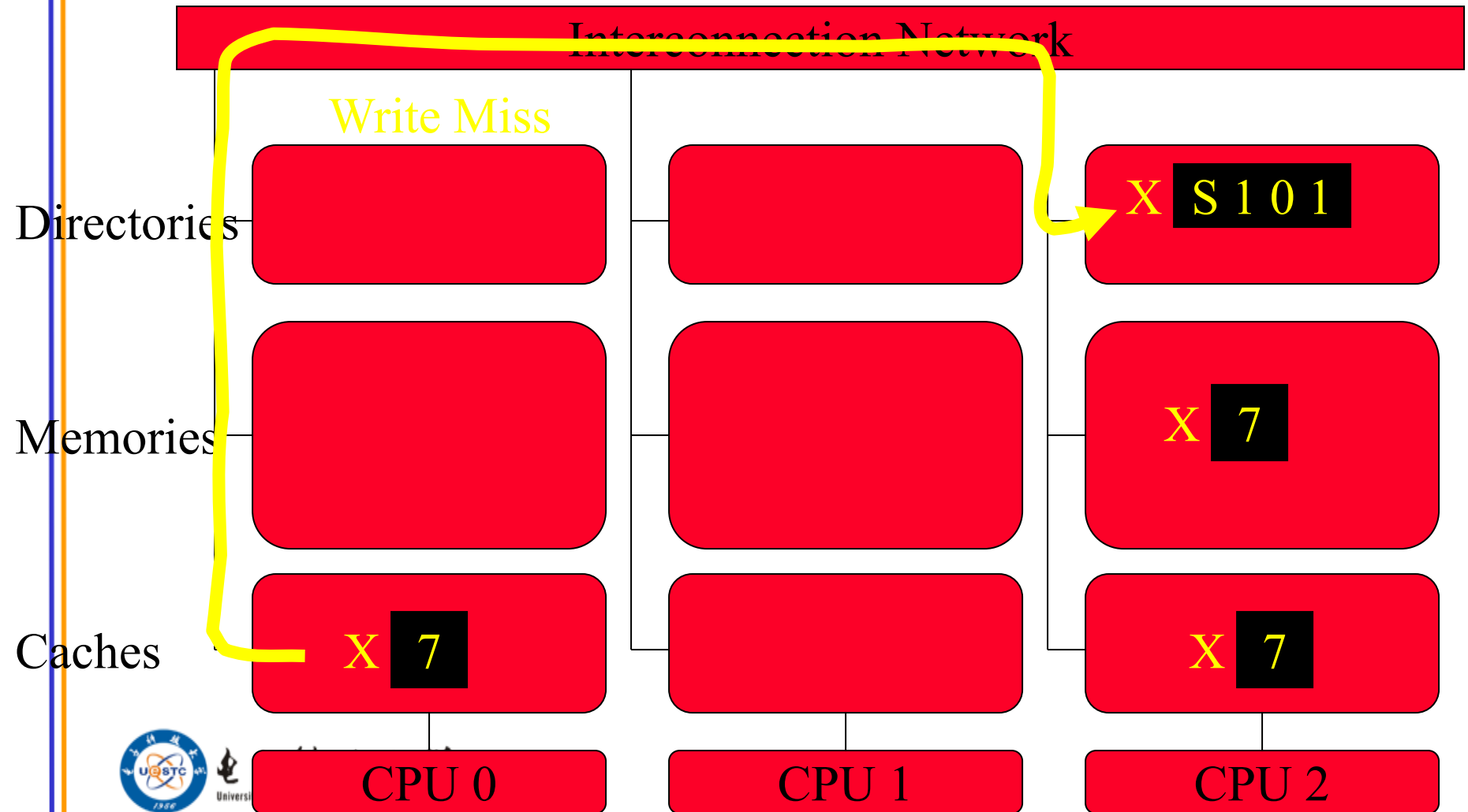
CPU 2 Reads X – step 7



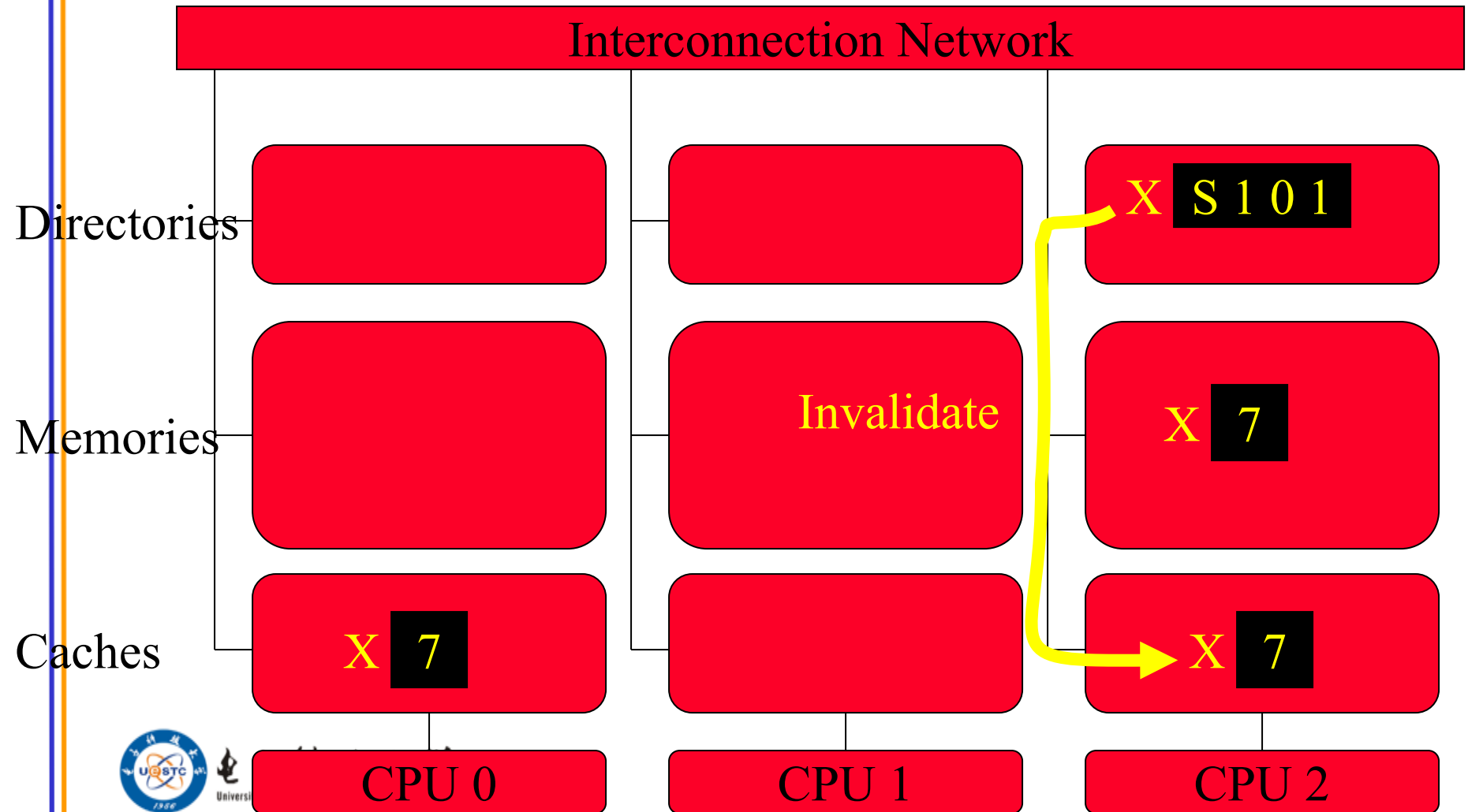
CPU 2 Reads X – step 8



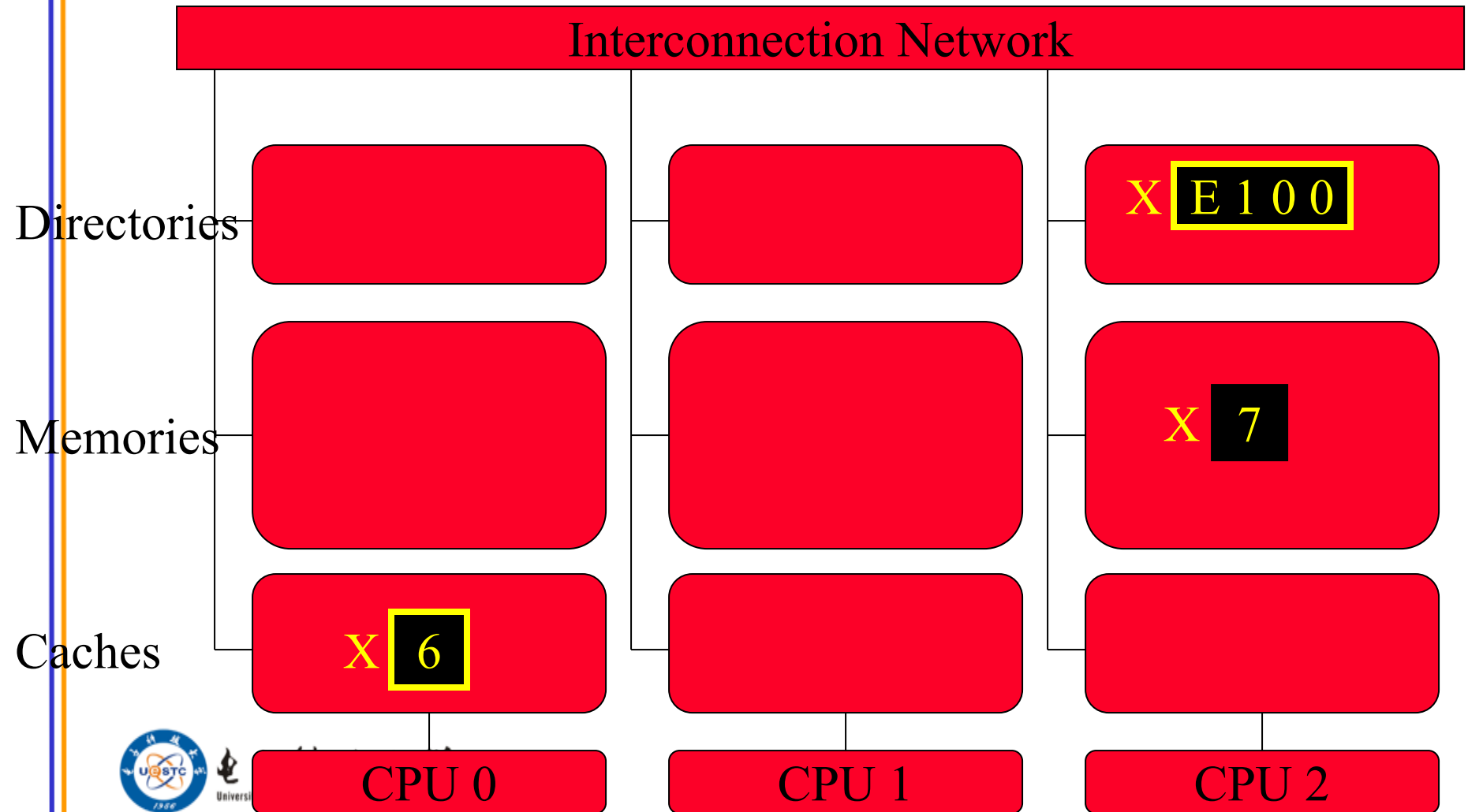
CPU 0 Writes 6 to X – step 9



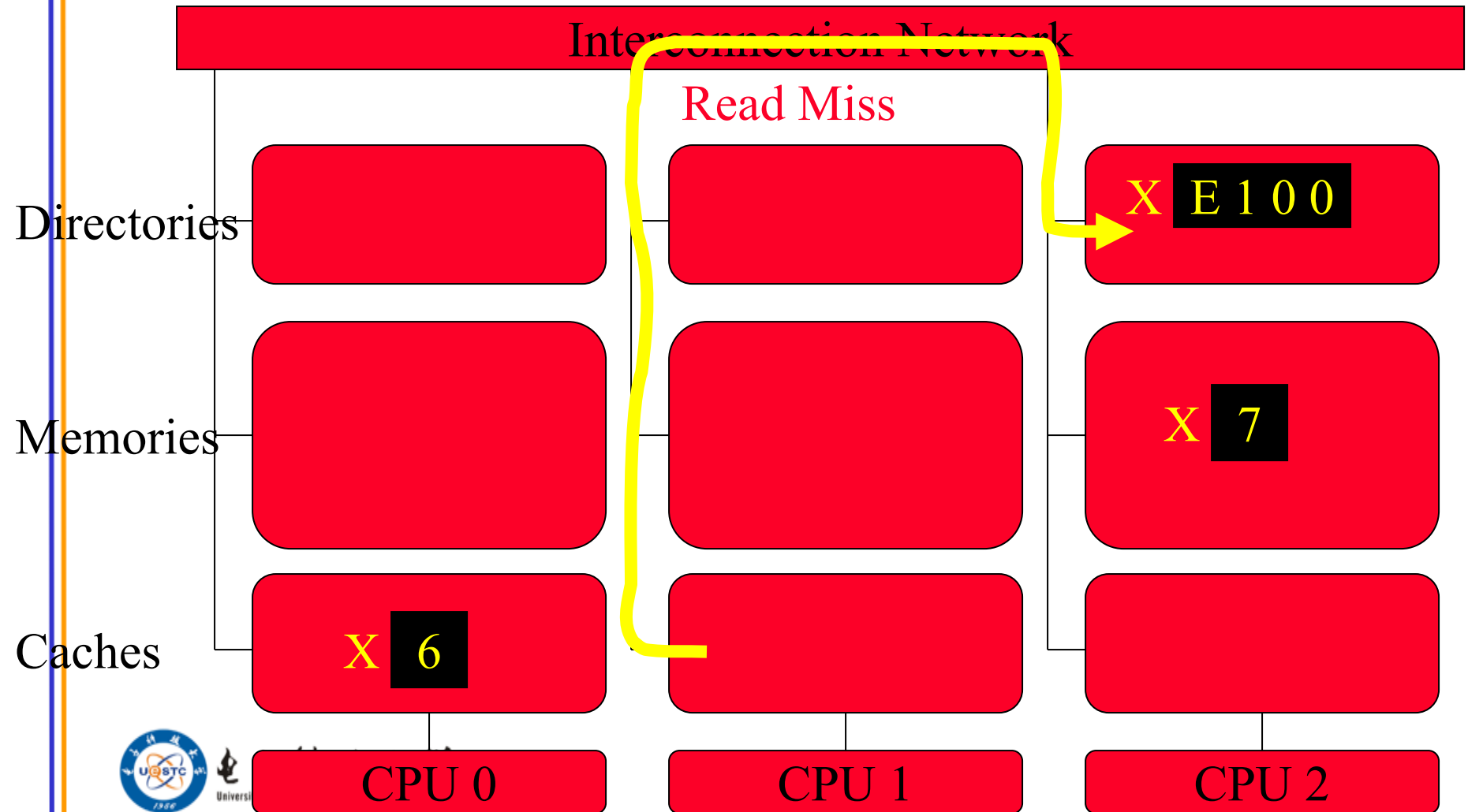
CPU 0 Writes 6 to X – step 10



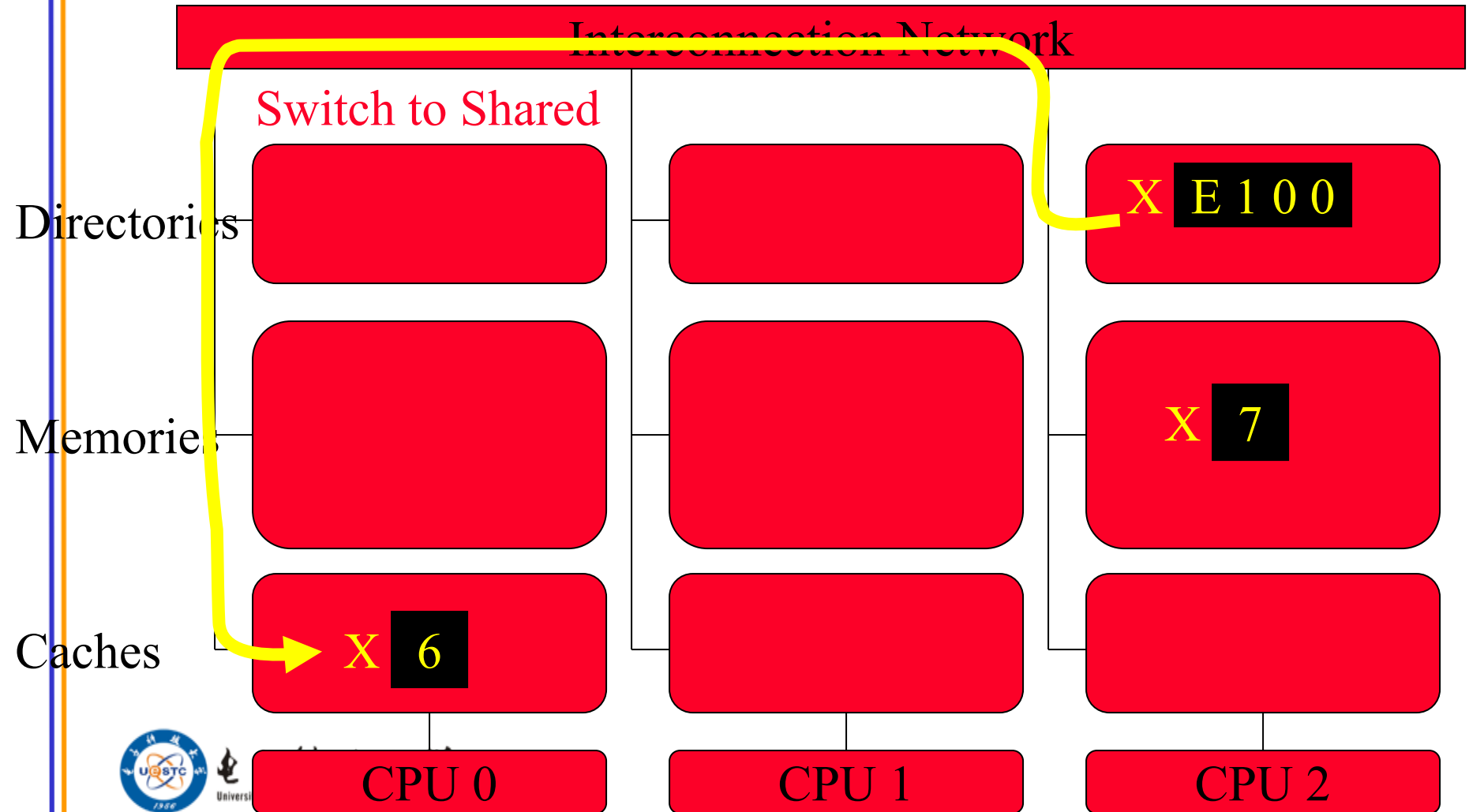
CPU 0 Writes 6 to X – step 11



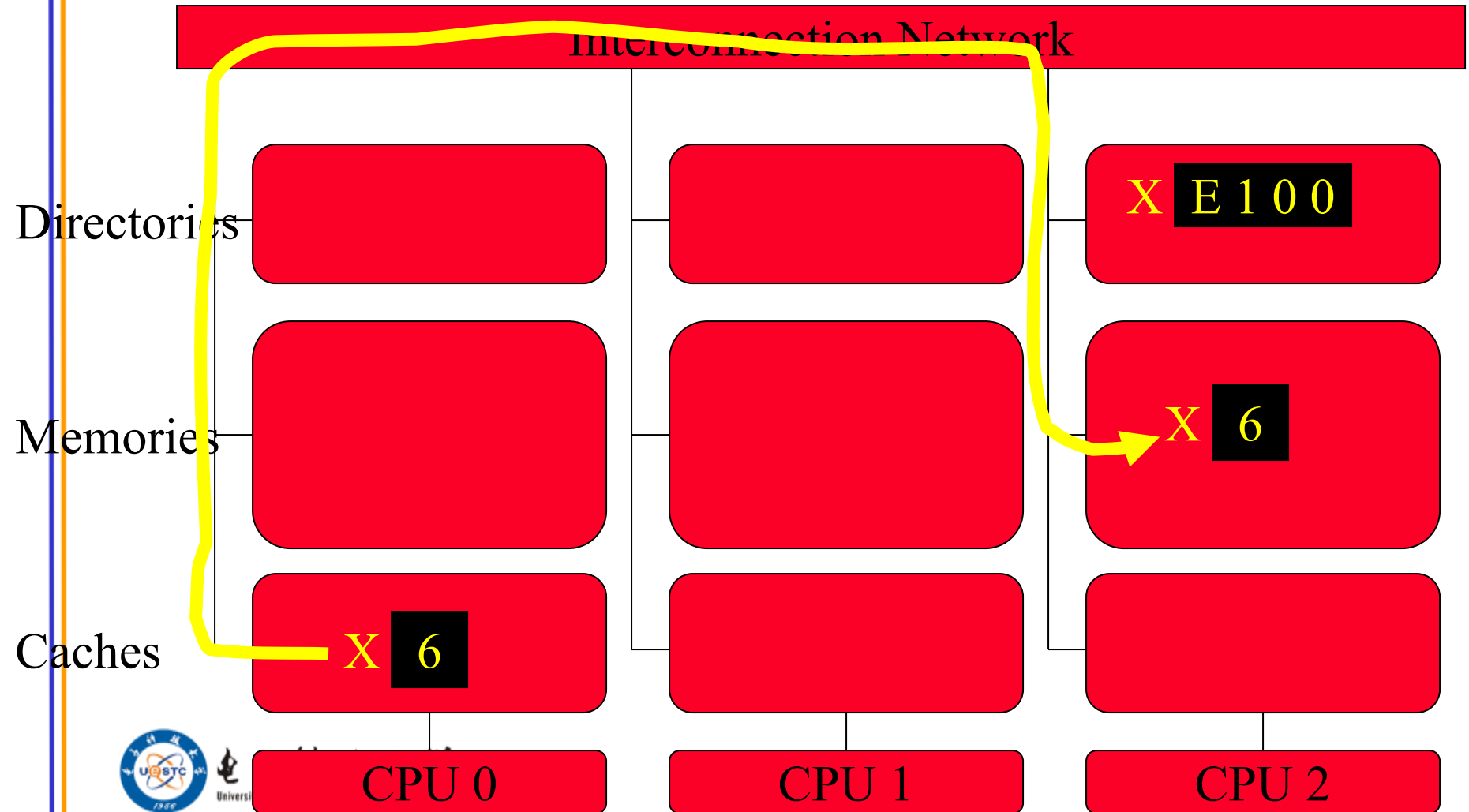
CPU 1 Reads X – step 12



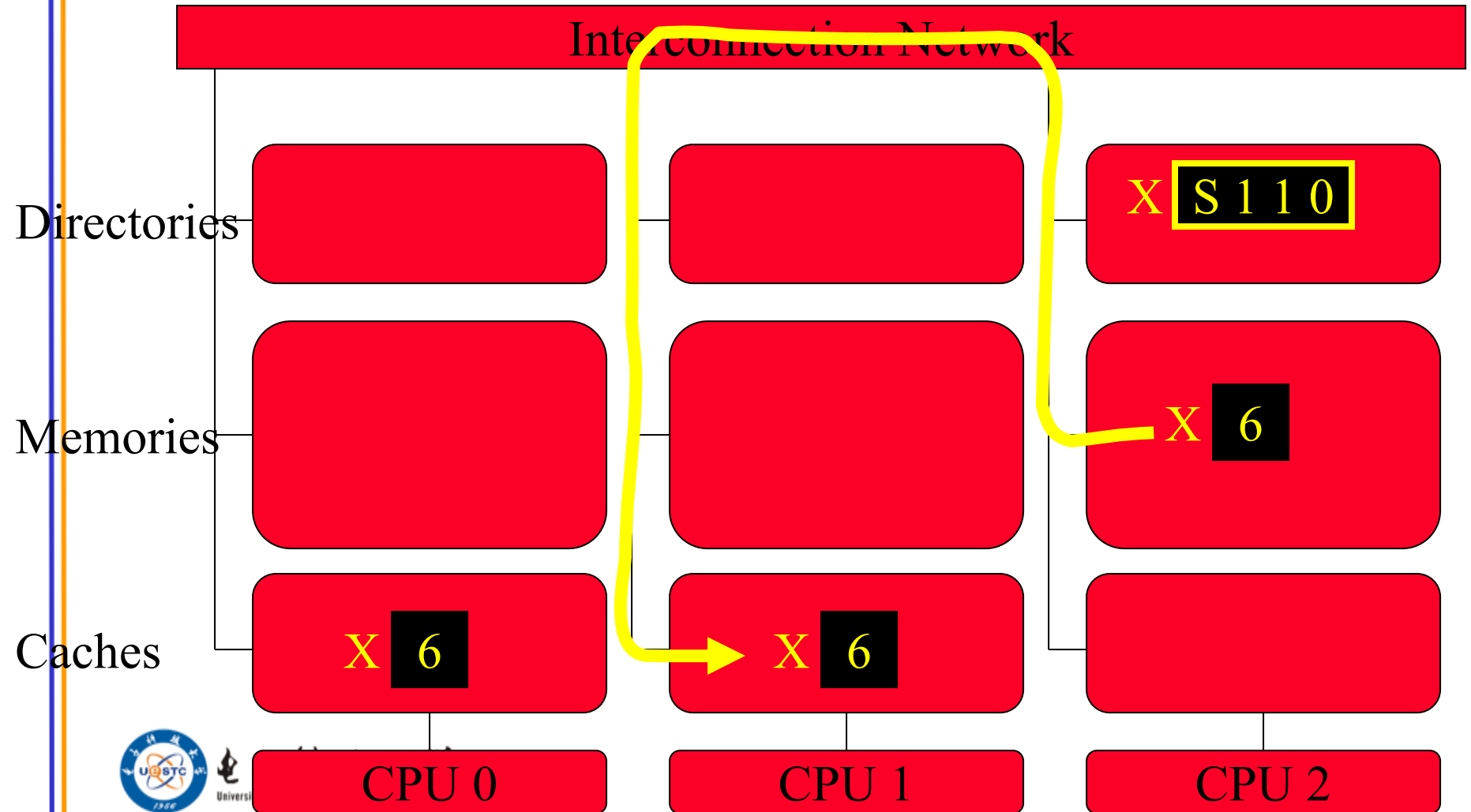
CPU 1 Reads X – step 13



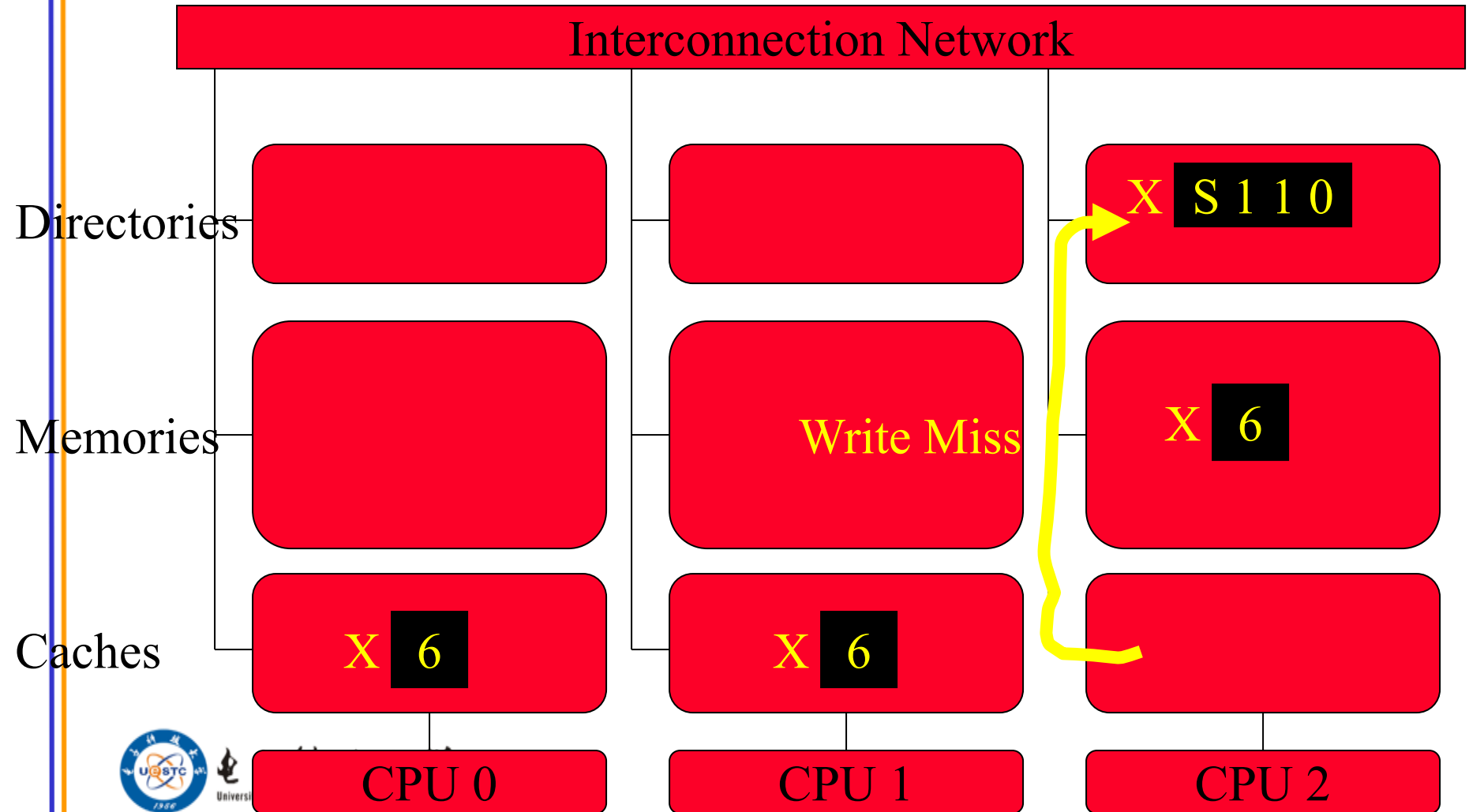
CPU 1 Reads X – step 14



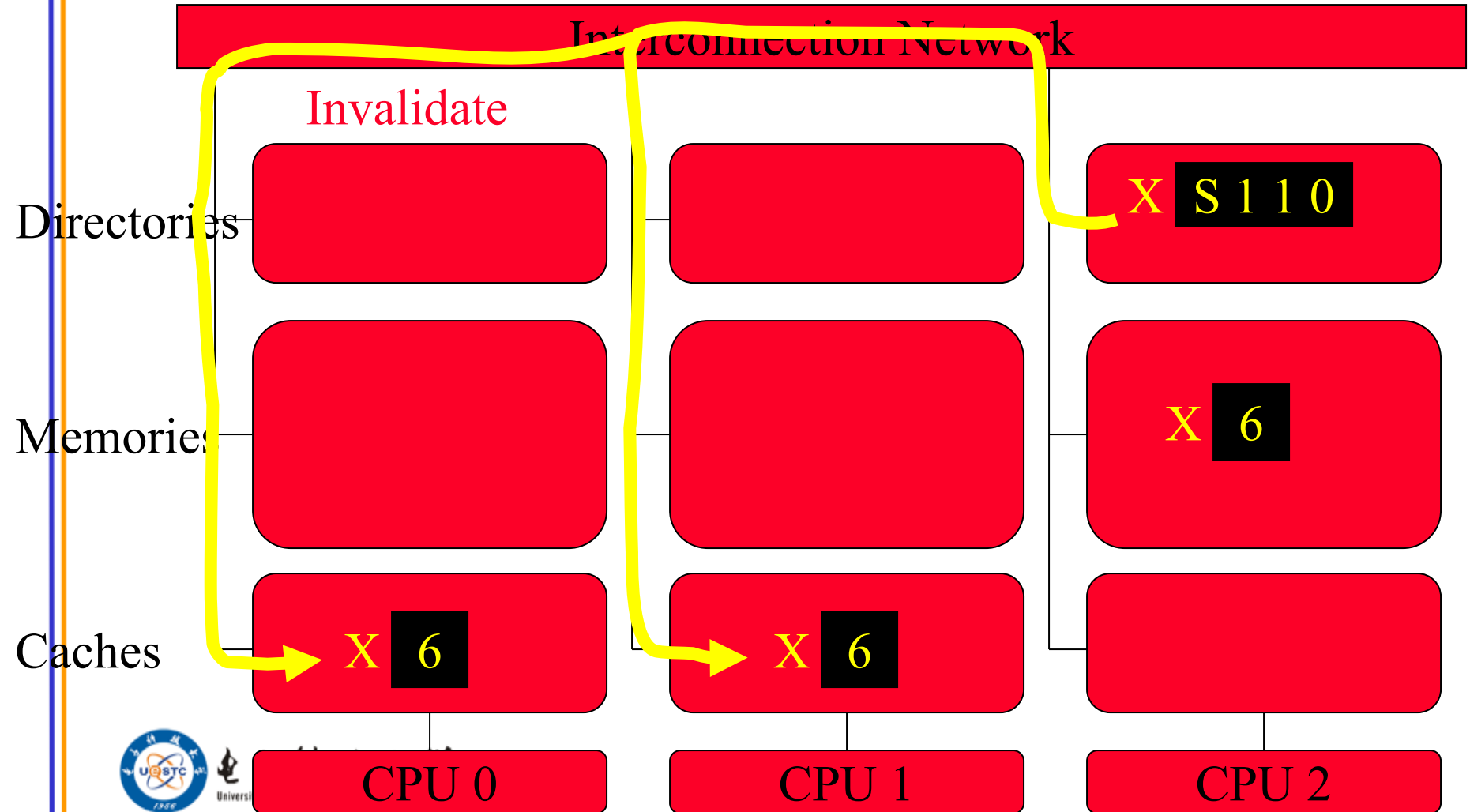
CPU 1 Reads X – step 15



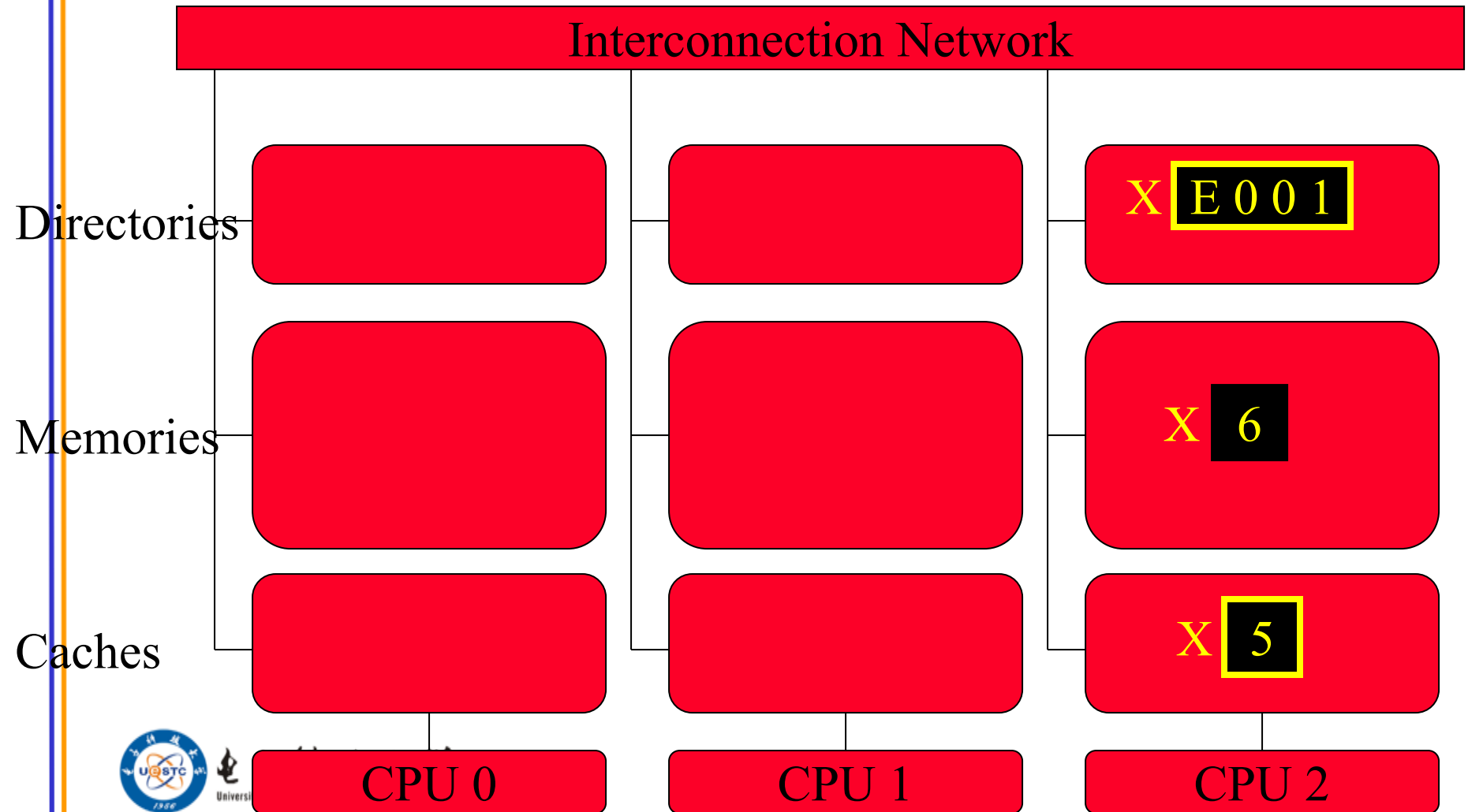
CPU 2 Writes 5 to X – step 16



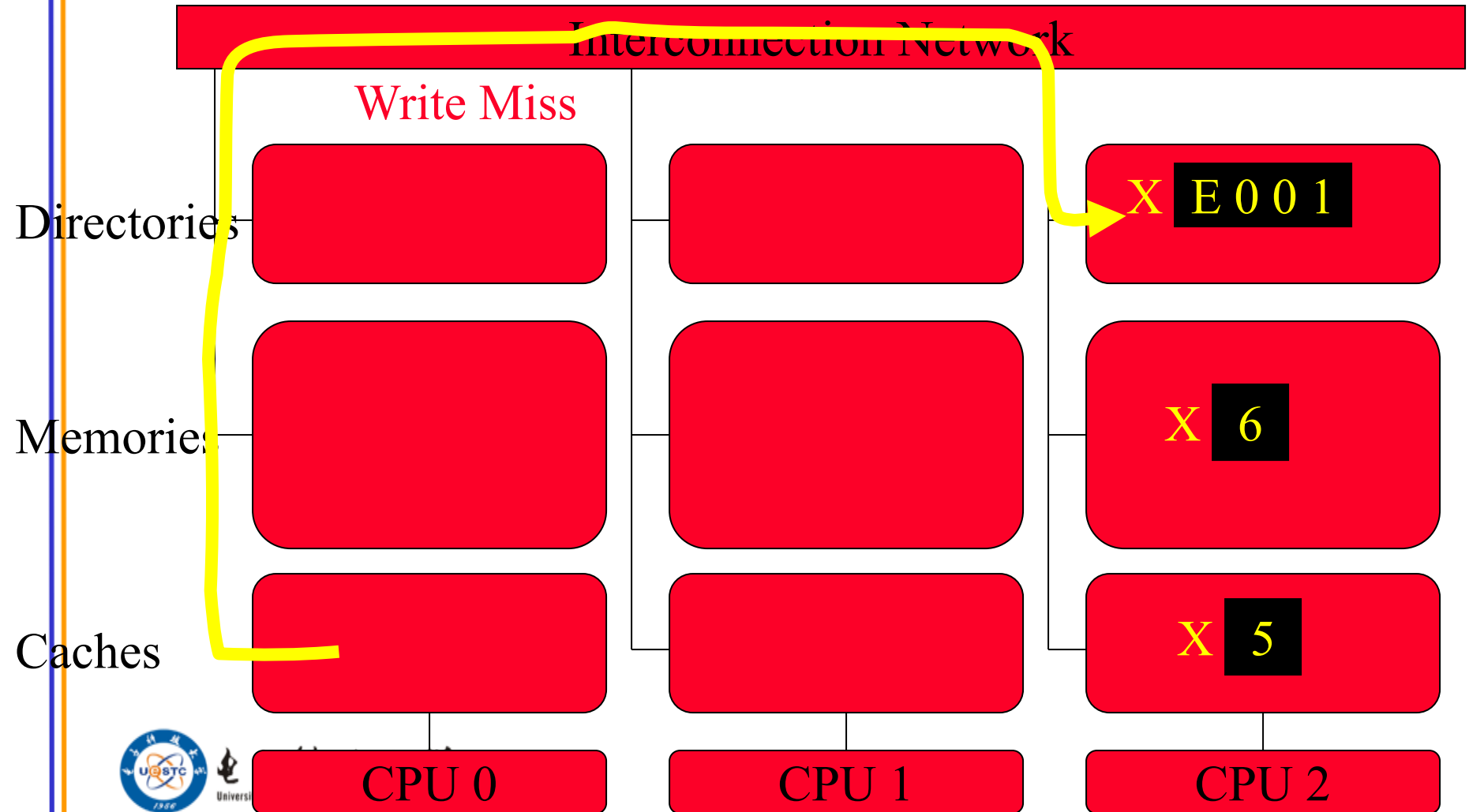
CPU 2 Writes 5 to X - step 17



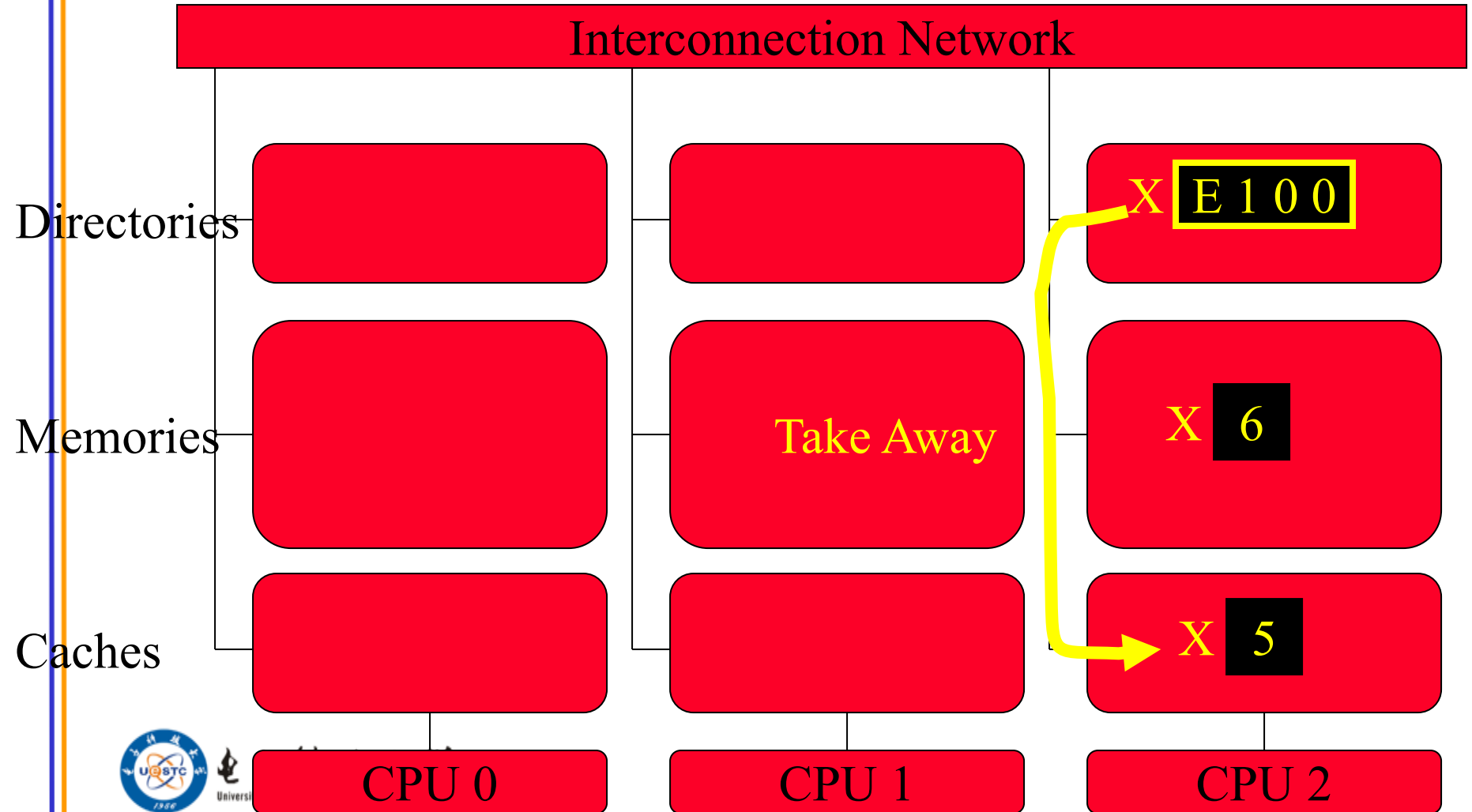
CPU 2 Writes 5 to X –step 18



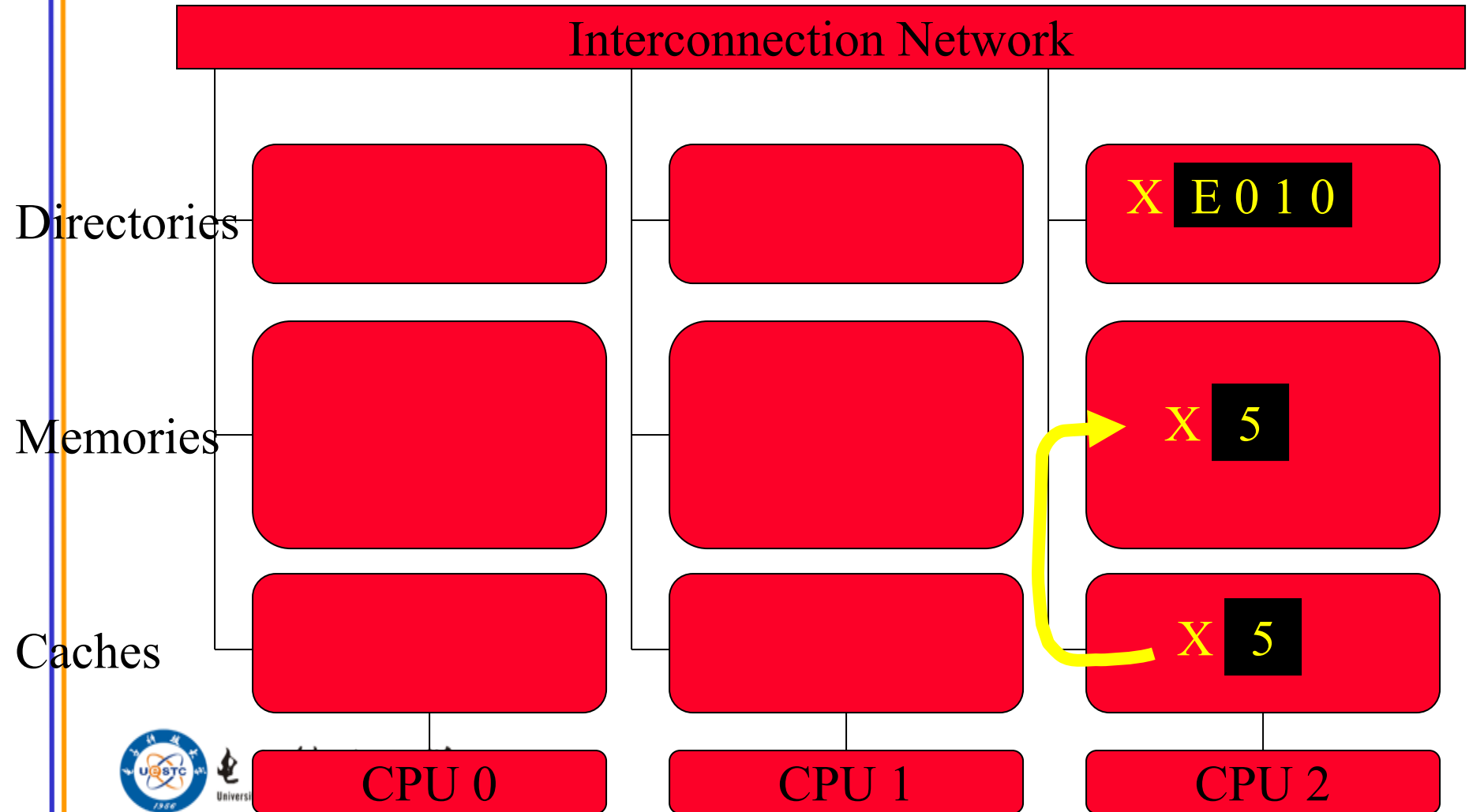
CPU 0 Writes 4 to X – step 19



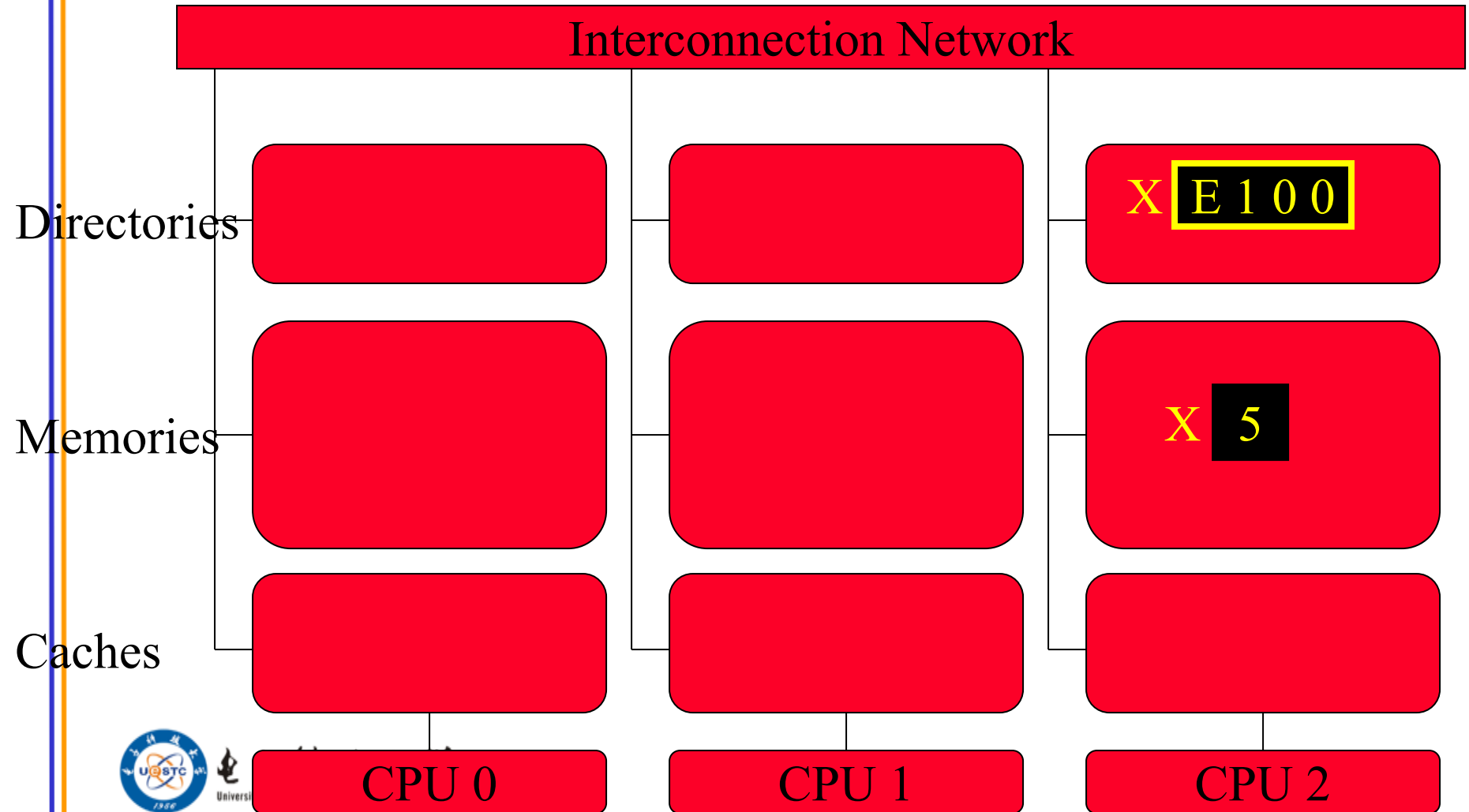
CPU 0 Writes 4 to X – step 20



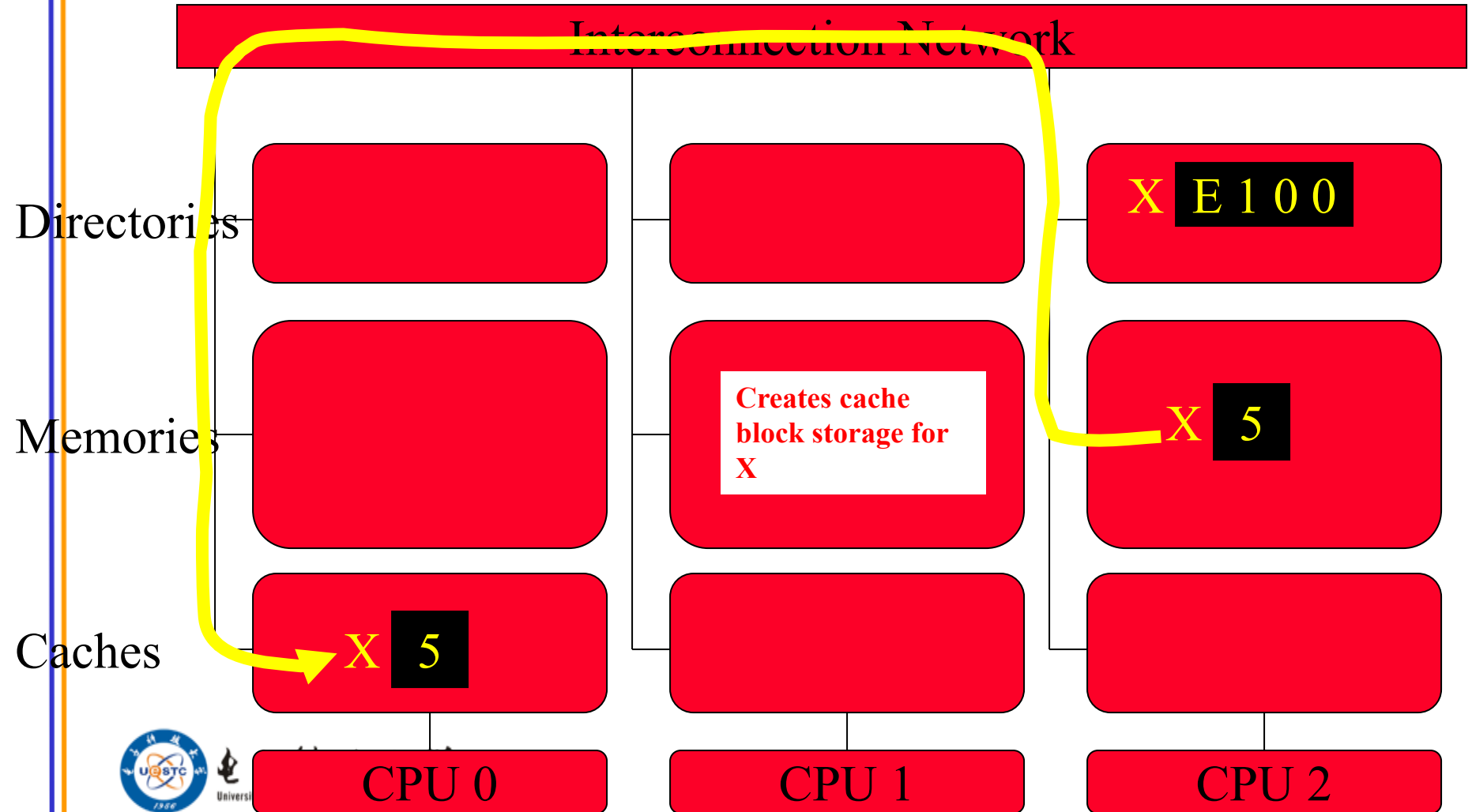
CPU 0 Writes 4 to X – step 21



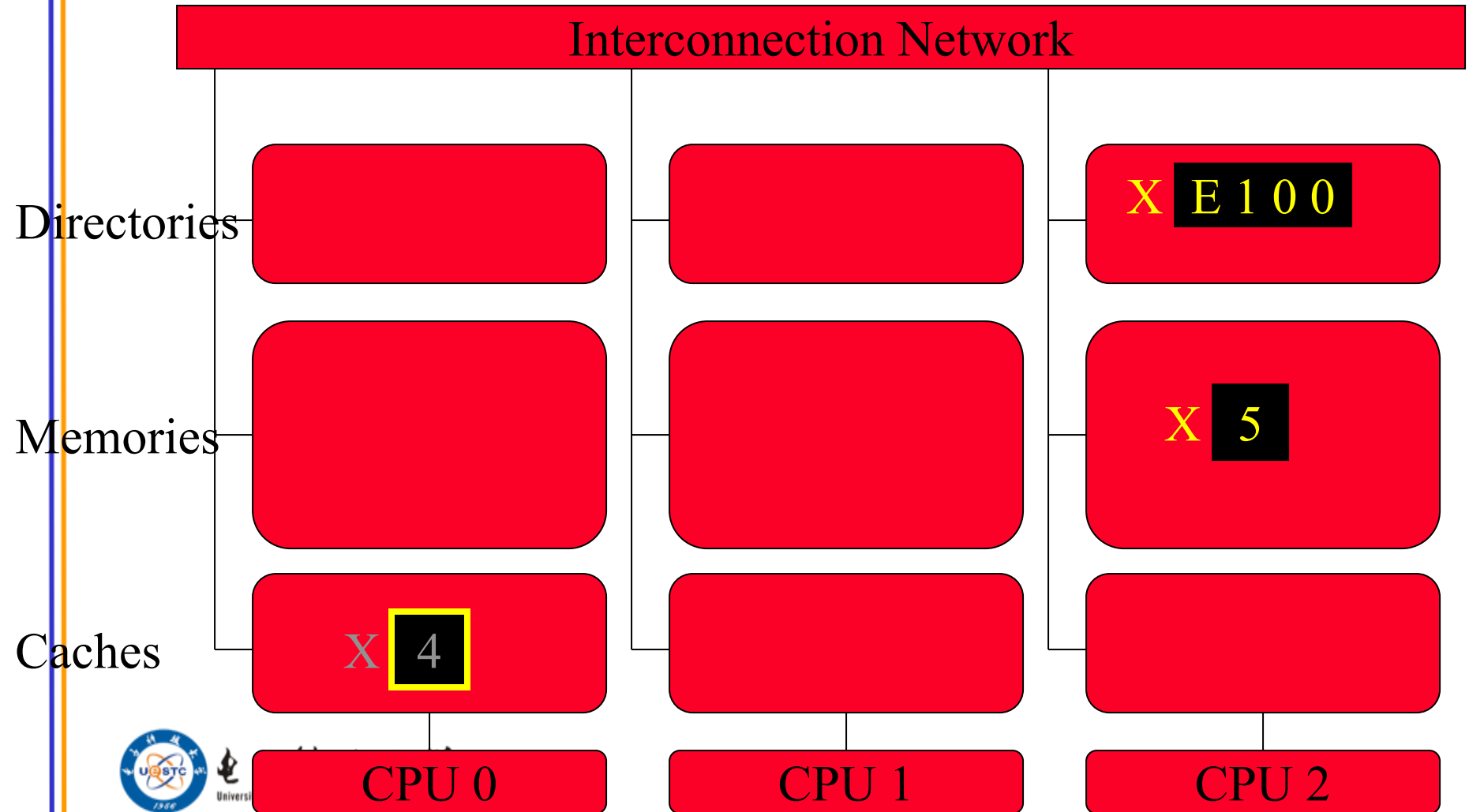
CPU 0 Writes 4 to X – step 22



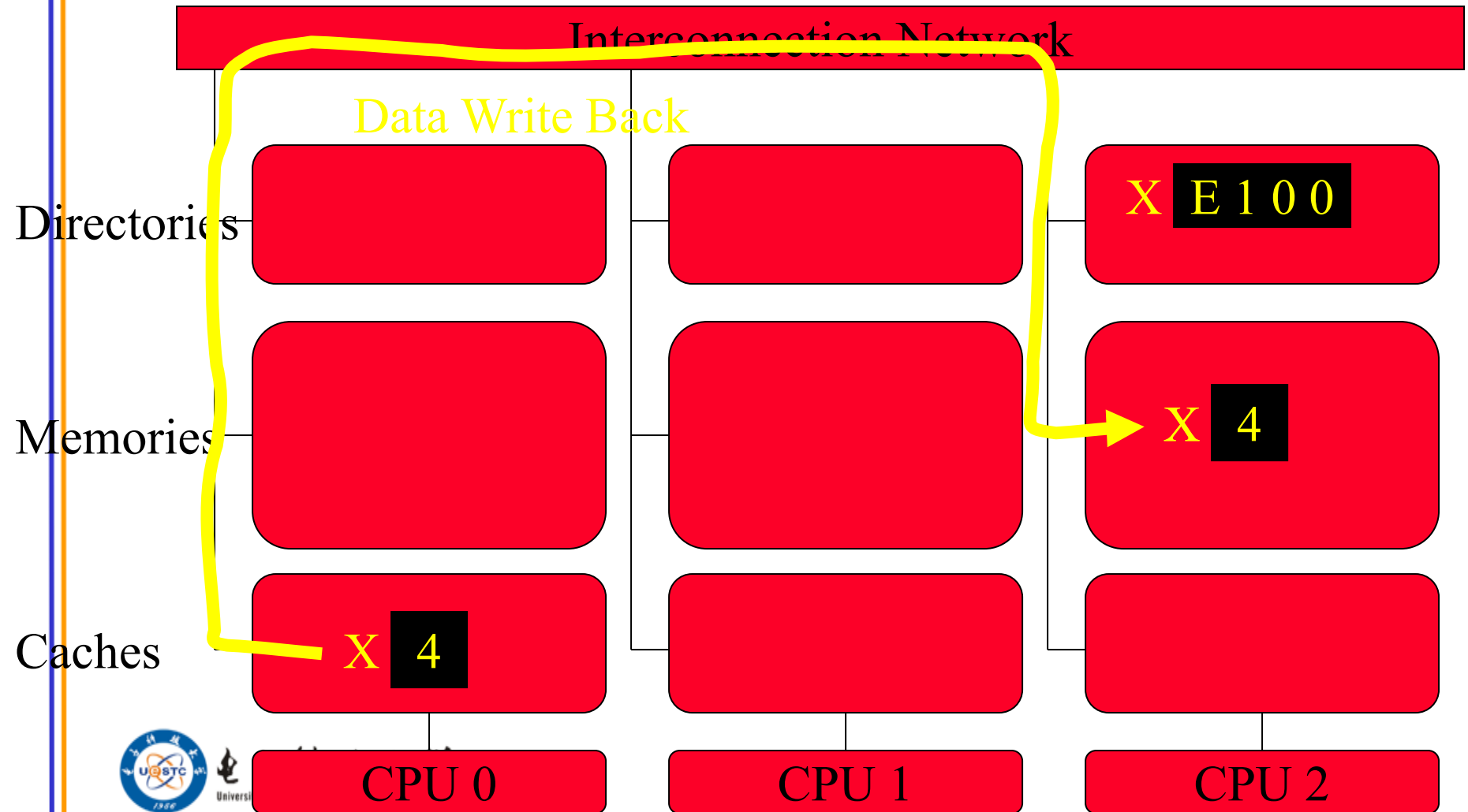
CPU 0 Writes 4 to X – step 23



CPU 0 Writes 4 to X – step 24

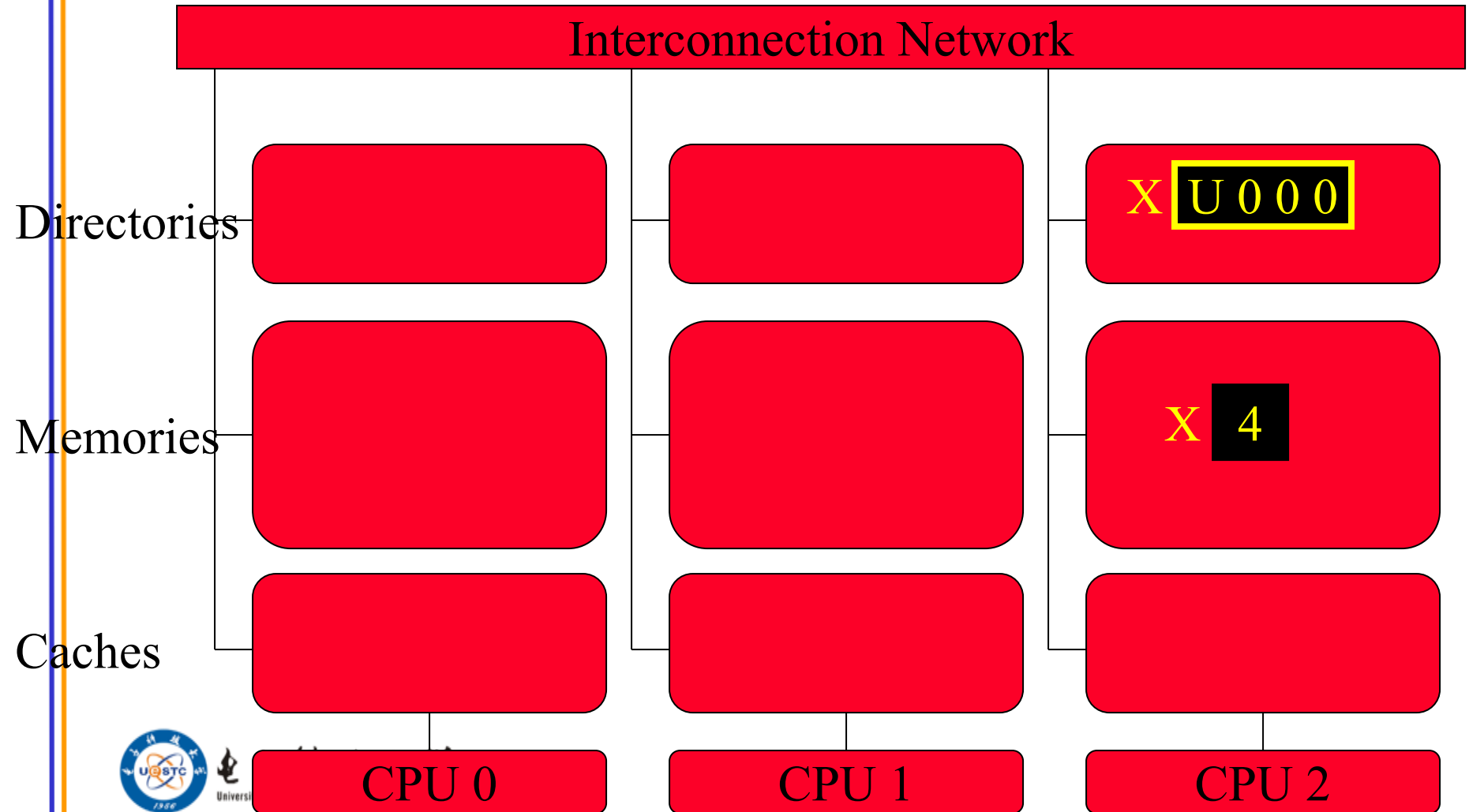


CPU 0 Writes Back X Block – step 25



CPU 0 flushes cache block X

step 26



基于目录的缓存一致性

- Overhead: 状态更新的传播（通信开销）；从目录中生成状态信息
- 如果一个并行程序需要大量的一致性动作（大量的读/写共享数据块），目录将最终约束其并行性能) Why?
- 存储目录的比特可能会增加大量的开销
 - 考虑到扩展到许多处理器
- 目录成为一个争论点
 - 分布式目录方案是必要的

Partitioned Global Address Space Languages

- 允许用户使用一些共享内存技术对分布式内存硬件进行编程

```
shared int n = ... ;
shared double x [ n ] , y [ n ] ;
private int i , my_first_element , my_last_element ;
my_first_element = ... ;
my_last_element = ... ;
/* Initialize x and y */
...
for ( i = my_first_element ; i <= my_last_element ; i++)
    x [ i ] += y [ i ] ;
```

- 私有变量被分配在进程执行的核心的本地内存中，而共享数据结构中的数据分配由程序员控制