北京大学
PEKING UNIVERSITY

人工智能的硬件基石
从物理器件到计算架构

第八讲：多级缓存与缓存一致性

主讲：陶耀宇

2025年春季

注意事项

北京大学 PEKING UNIVERSITY

- 课程作业情况

  - **第3次作业时间：4月7号 – 4月21号**
    - 6次作业可以使用总计6个Late day
    - Late Day耗尽后，每晚交1天扣除20%当次作业分数
  - **第1次lab时间：3月10晚上线 - 4月10晚11:59**
    - **2个基础任务（50%+50%）+ 1个Bonus（可2选1，50%）**
  - **第2次lab时间：4月11日 – 6月11日**
    - **2个基础任务（50%+50%）+ 1个Bonus（50%）**

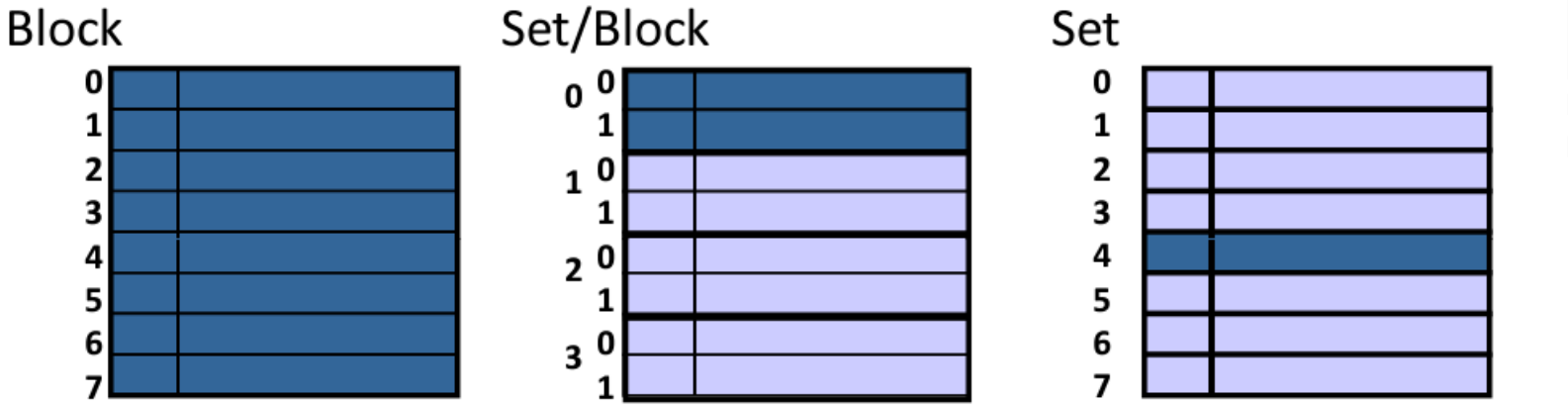
目 录

CONTENTS

缓存Cache的基本思路

- **Cache Block Placement**

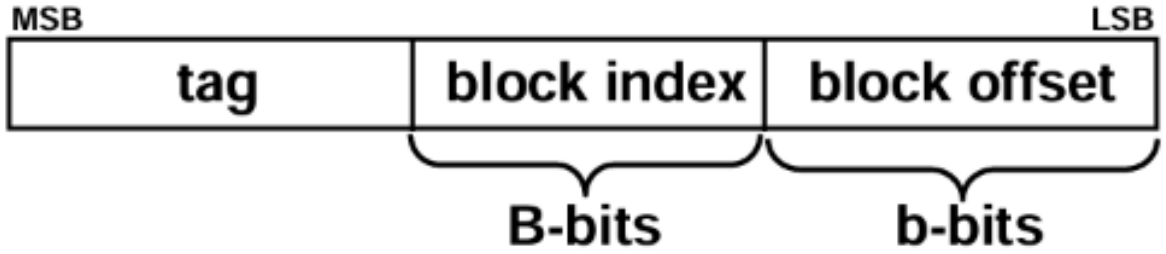
Where does block 12 (b'1100) go?



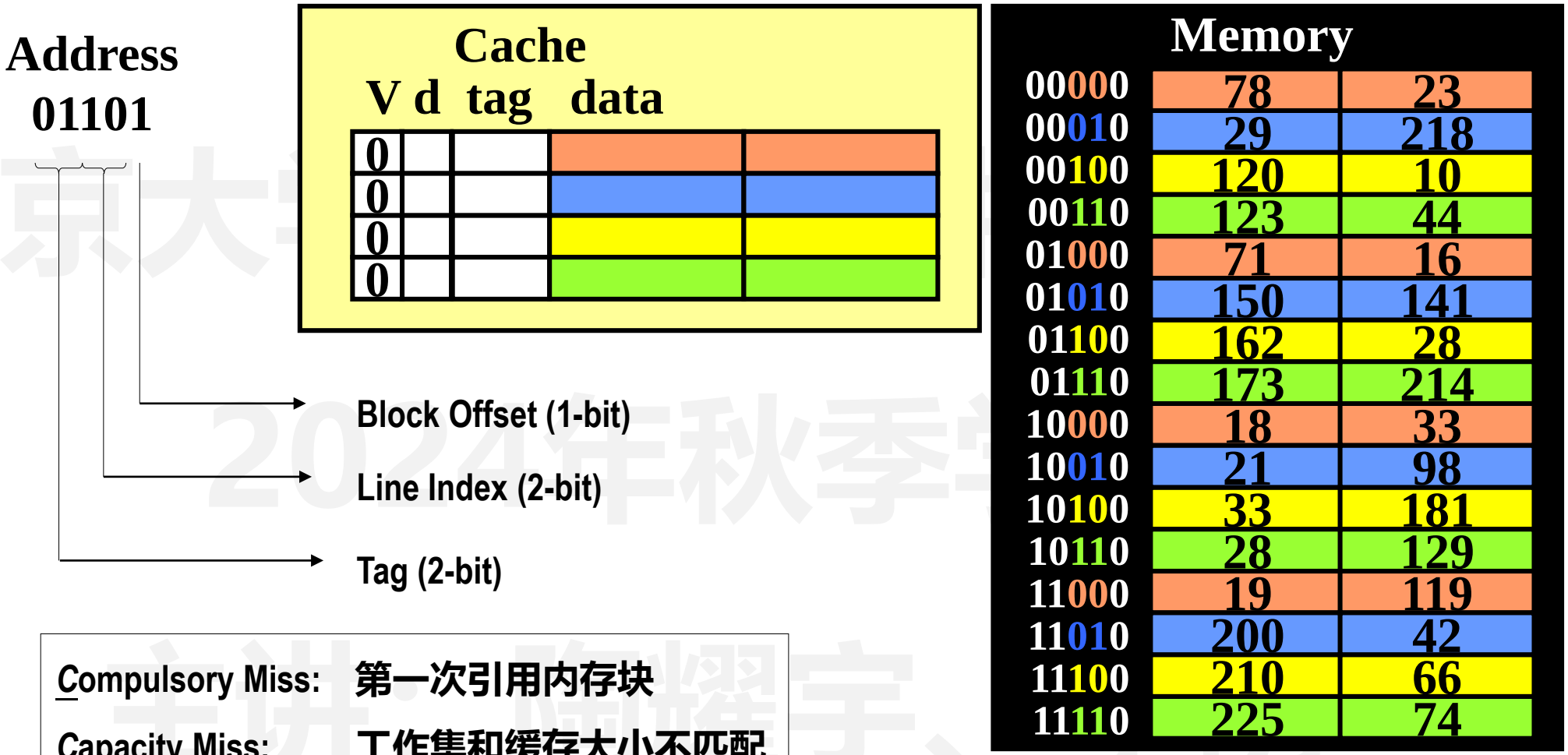
Fully-associative	Set-associative	Direct-mapped
block goes in any frame	a block goes in any frame in exactly one set	block goes in exactly one frame
(think all frames in 1 set)	(frames grouped into sets)	(think 1 frame per set)

Cache Block Size的概念

- Each cache block frame or (cache line) has only one tag but can hold multiple “chunks” of data
  - **Reduce tag storage overhead**
    - In 32-bit addressing, an 1-MB direct-mapped cache has 12 bits of tags
      - 4-byte cache block ⇒ 256K blocks ⇒ ~384KB of tag
      - 128-byte cache block ⇒ 8K blocks ⇒ ~12KB of tag
  - **The entire cache block is transferred to and from memory all at once**
    - good for spatial locality because if you access address i you will probably want i+1 as well (prefetching effect)
- Block size = 2^b; Direct Mapped Cache Size = 2^(B+b)



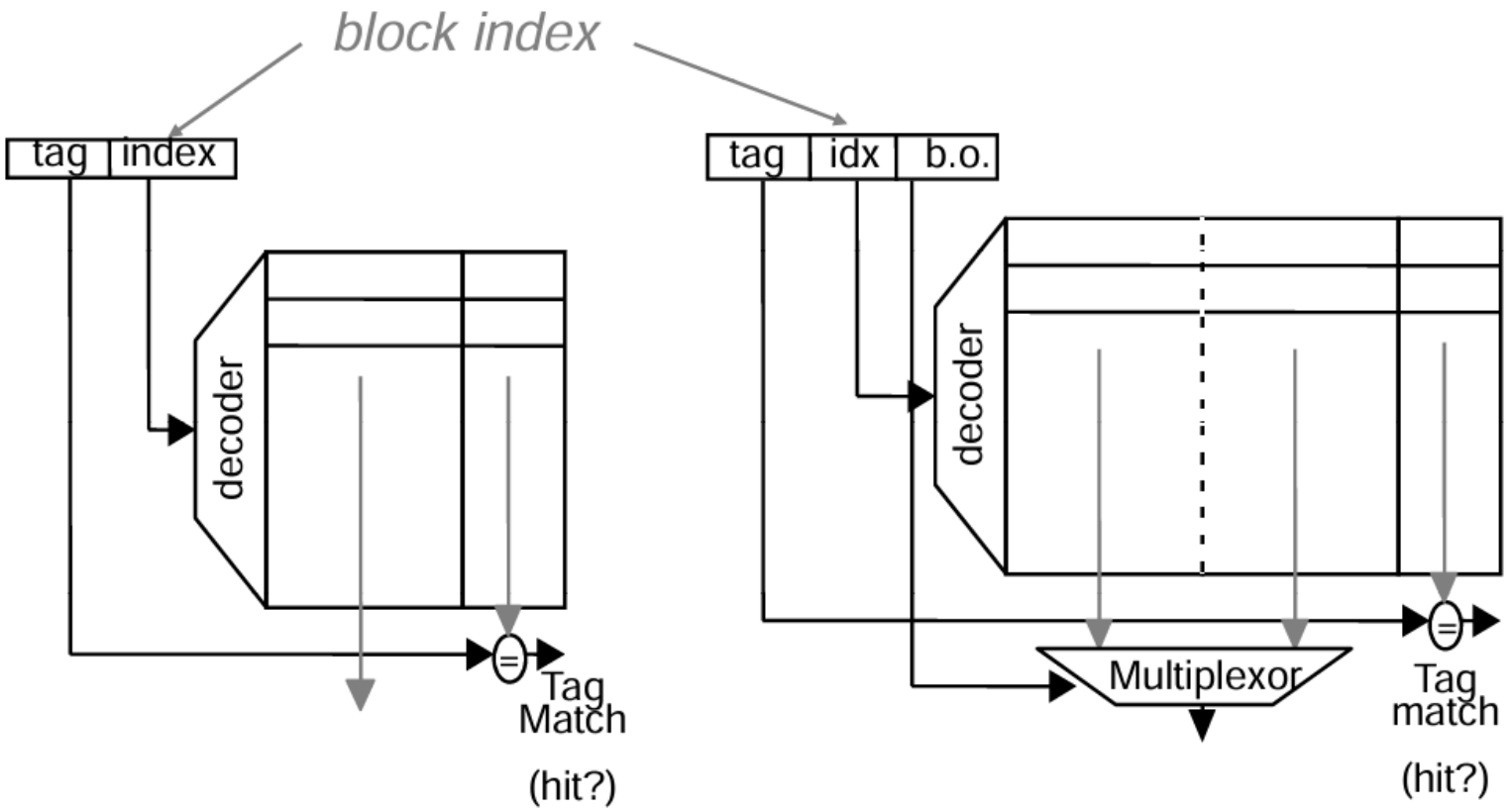
Direct-Mapped Cache设计



3-C's

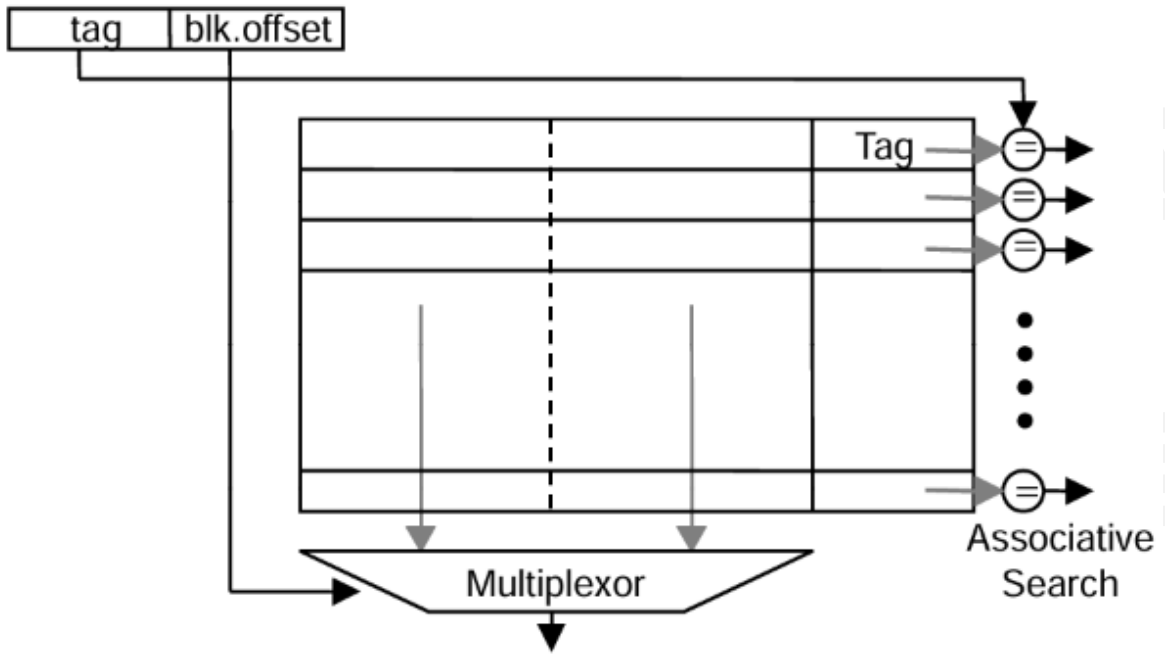
Compulsory Miss:	第一次引用内存块
Capacity Miss:	工作集和缓存大小不匹配
Conflict Miss:	工作集映射到同一缓存行

Direct-Mapped Cache设计



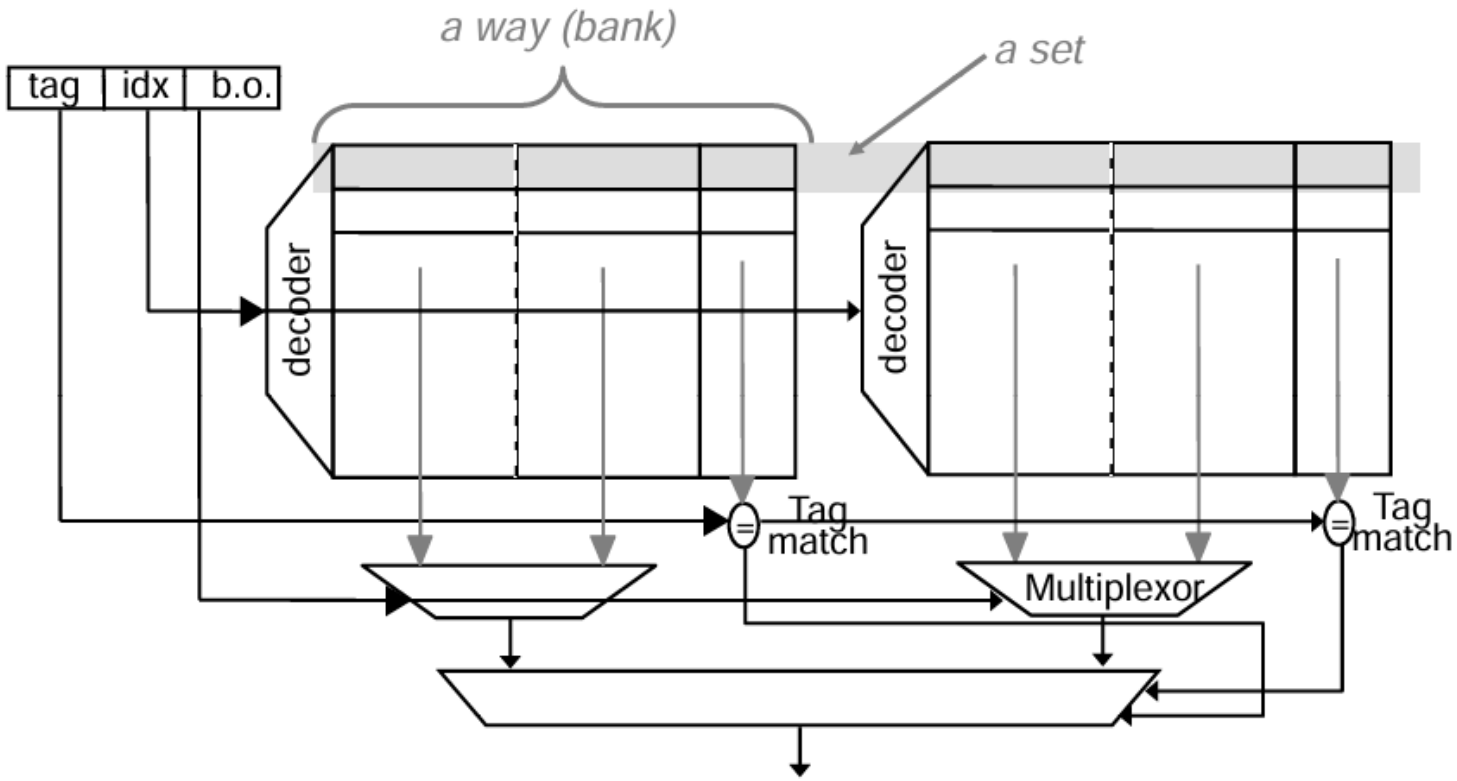
Don't forget to check the valid/state bits

Fully-Associative Cache设计



没有block index

N-Way Set Associative Cache设计



Cache Size = N x 2^{B+b}

N-Way Set Associative Cache设计

北京大学 PEKING UNIVERSITY

- **Associative Block Replacement**
  - 当发生未命中时要替换set中的哪一个block?
    - 理想情况下 — — 替换未来最晚被访问的block
      - 如何实现?
    - Approximations:
      - **Least recently used — LRU**
        - 针对时间局部性进行了优化（假设有的话），对于超过两路的情况，成本较高
      - **Not most recently used — NMRU**
        - 跟踪 MRU，从其他block中随机选择，很好的折衷方案
      - **Random**
        - **几乎和 LRU 一样好，更简单（通常是伪随机）**
        - 区块替换政策有多重要?

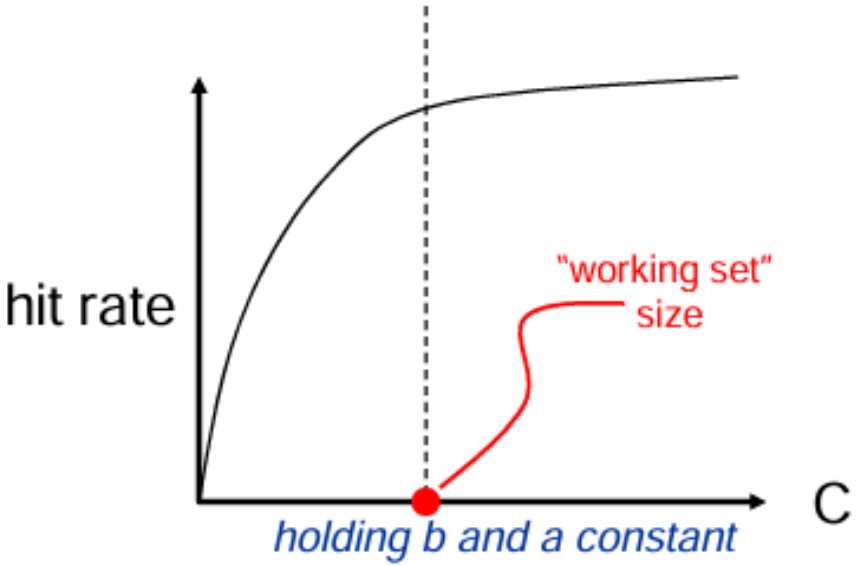
Cache Miss种类

- Miss Classification (3+1 C’s)
  - **Compulsory Miss**
    - **首次访问某个块时出现“cold miss”**
      - —defined as: miss in infinite cache
  - **Capacity Miss**
    - **由于缓存不够大而发生未命中—**
      defined as: miss in fully-associative cache
  - **Conflict Miss**
    - **由于限制性映射策略而发生未命中**
    - 仅在组相联或直接映射缓存中
      - —defined as: not attributable to compulsory or capacity
  - **Coherence Miss**
    - **由于多处理器之间的共享而发生未命中**

Cache参数的选择

- Cache Size（C），Block Size（b），Associativity（a）

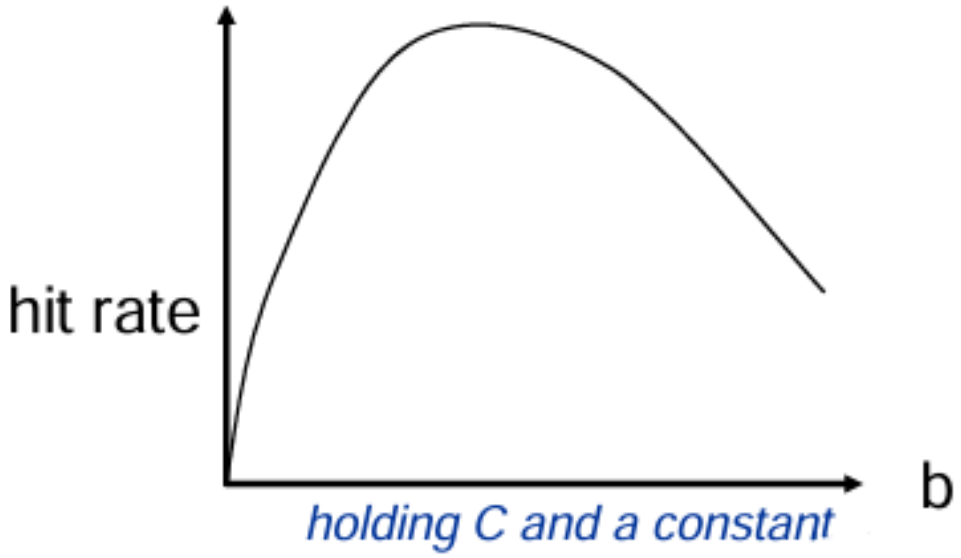
- **缓存大小是总数据（不包括标签）容量**
  - 更大可以更好地利用时间局部性
  - not ALWAYS better
- **缓存太大**
  - 越小越快 => 越大越慢
  - 访问时间可能会降低关键路径
- **缓存太小**
  - 没有很好地利用时间局部性
  - 有用的数据不断被替换



Cache参数的选择

- Block Size（b）

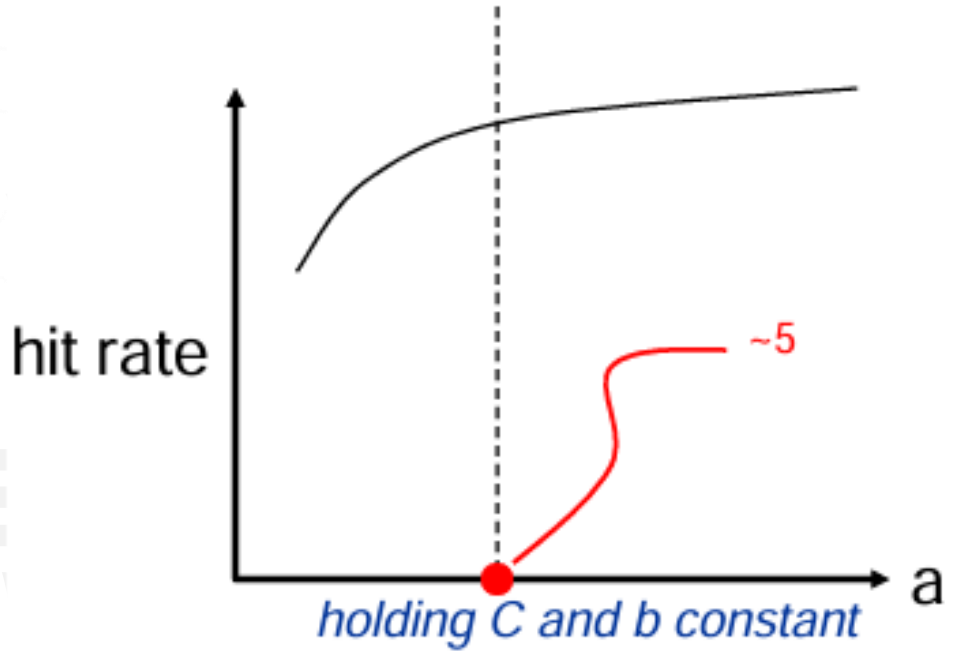
- **块大小是指**
  - 与地址标签关联
  - 不一定是层次结构之间的传输单位（remember sub-blocking）
- **区块太小**
  - 没有很好地利用空间局部性
  - 有过多的标签开销
- **块太大**
  - 传输了无用的数据
  - 有用数据被频繁替换
  - 总区块数太少



Cache参数的选择

- Associativity（a）

  - **Partition cache frames into**
    - equivalence classes of frames called sets
  - **Typical values for associativity**
    - 1, 2-, 4-, 8-路相关
  - **Larger associativity**
    - 更低的miss rate可以减小程序的变动
    - 仅对小“C/b”重要
  - **Smaller associativity**
    - 成本更低，命中时间更快



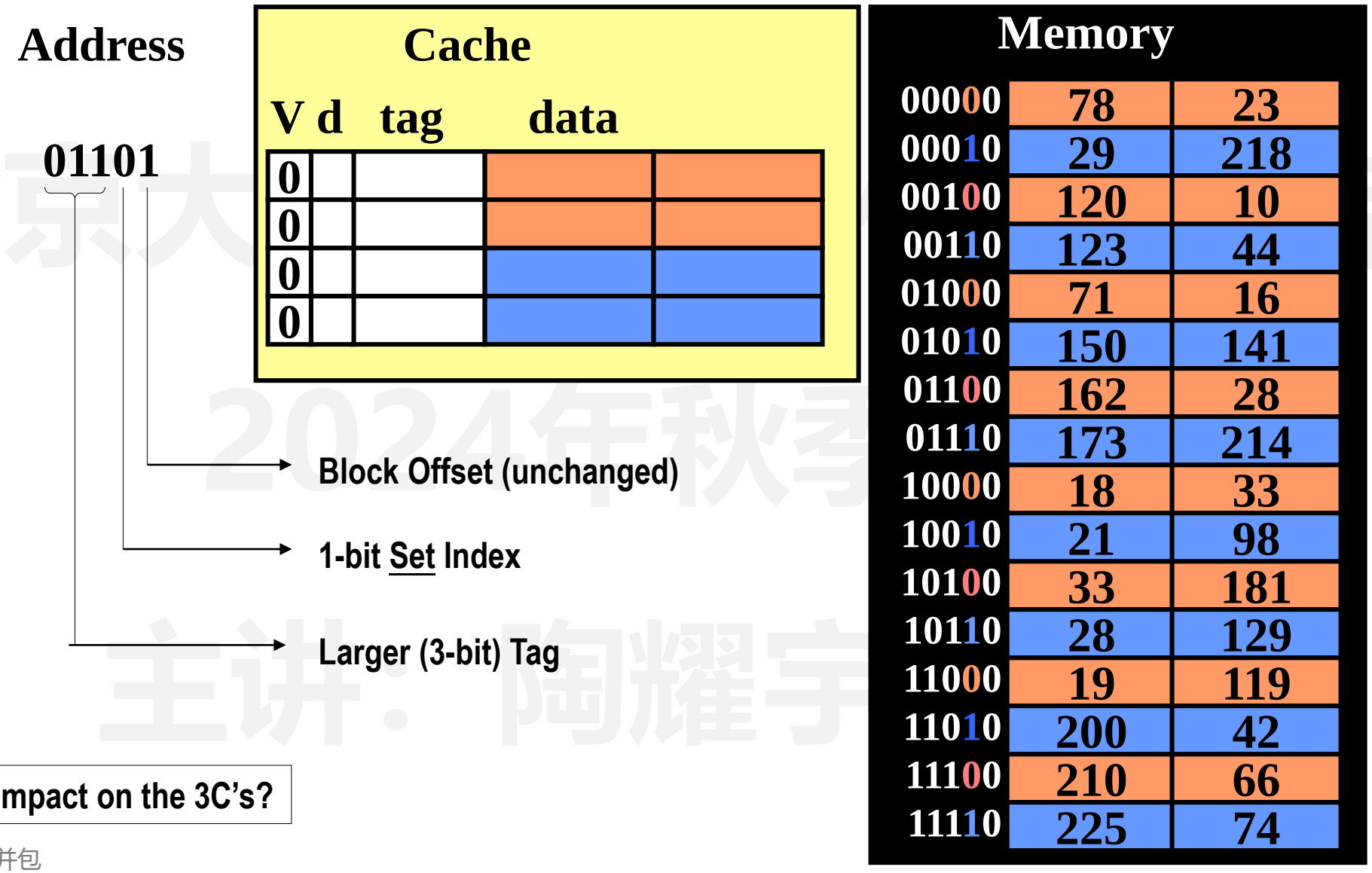
Cache设计选择的影响

北京大学 PEKING UNIVERSITY

- Cache写入和Miss处理策略

- 写入策略：如何处理写入未命中?
  - ***Write-through / no-allocate***
    - 每次写入时更新内存
    - 保持内存更新
  - ***Write-back / write-allocate***
    - 仅在块替换时更新内存
    - 许多缓存行只读，从不写入
    - 将“dirty”位添加到状态字
      - 替换后原先清除
      - 当块帧被写入时set
      - 仅写回dirty块，并清除clean块而不进行内存更新

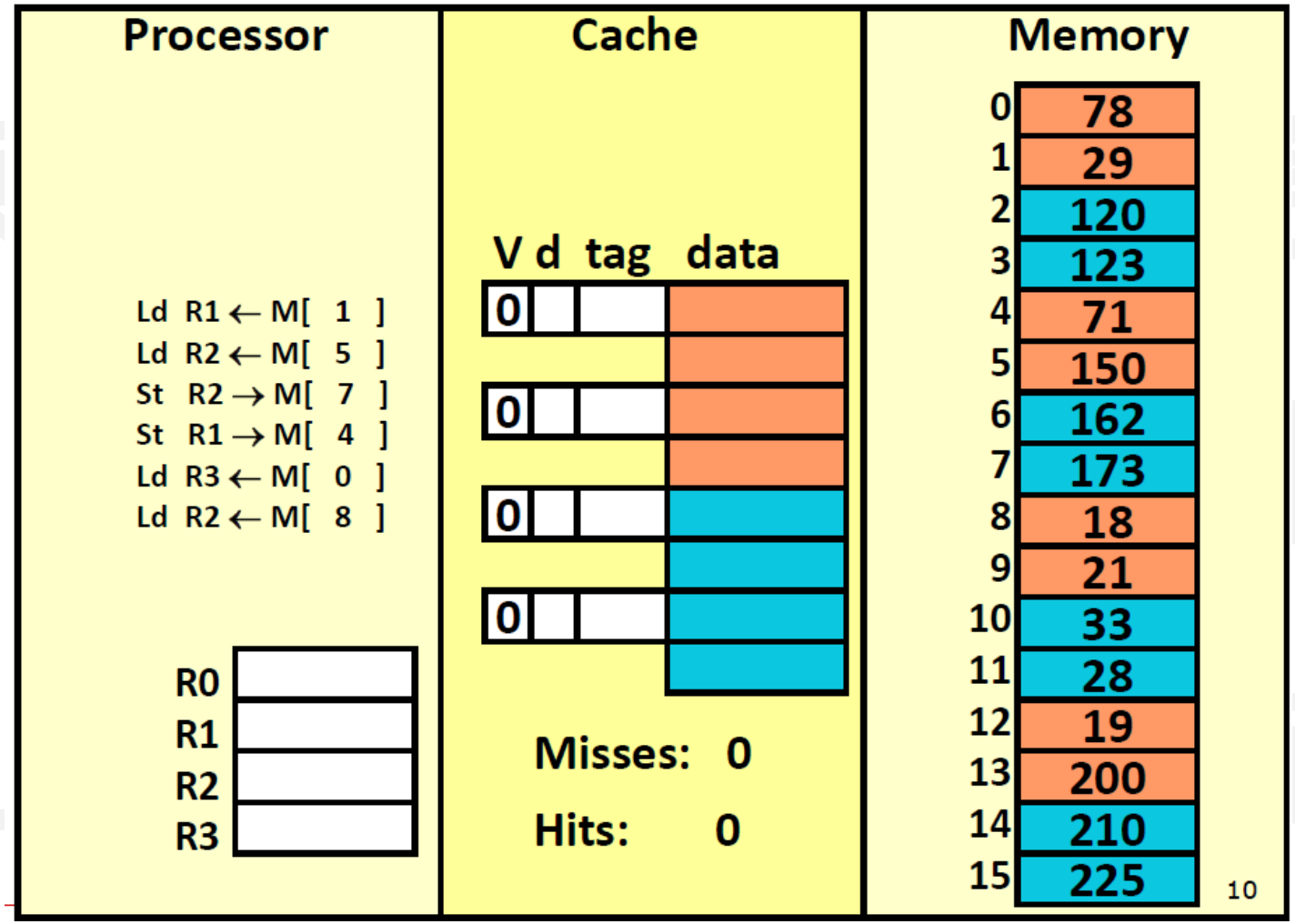
2-Way Set Associative Cache设计



Memory		
00000	78	23
00010	29	218
00100	120	10
00110	123	44
01000	71	16
01010	150	141
01100	162	28
01110	173	214
10000	18	33
10010	21	98
10100	33	181
10110	28	129
11000	19	119
11010	200	42
11100	210	66
11110	225	74

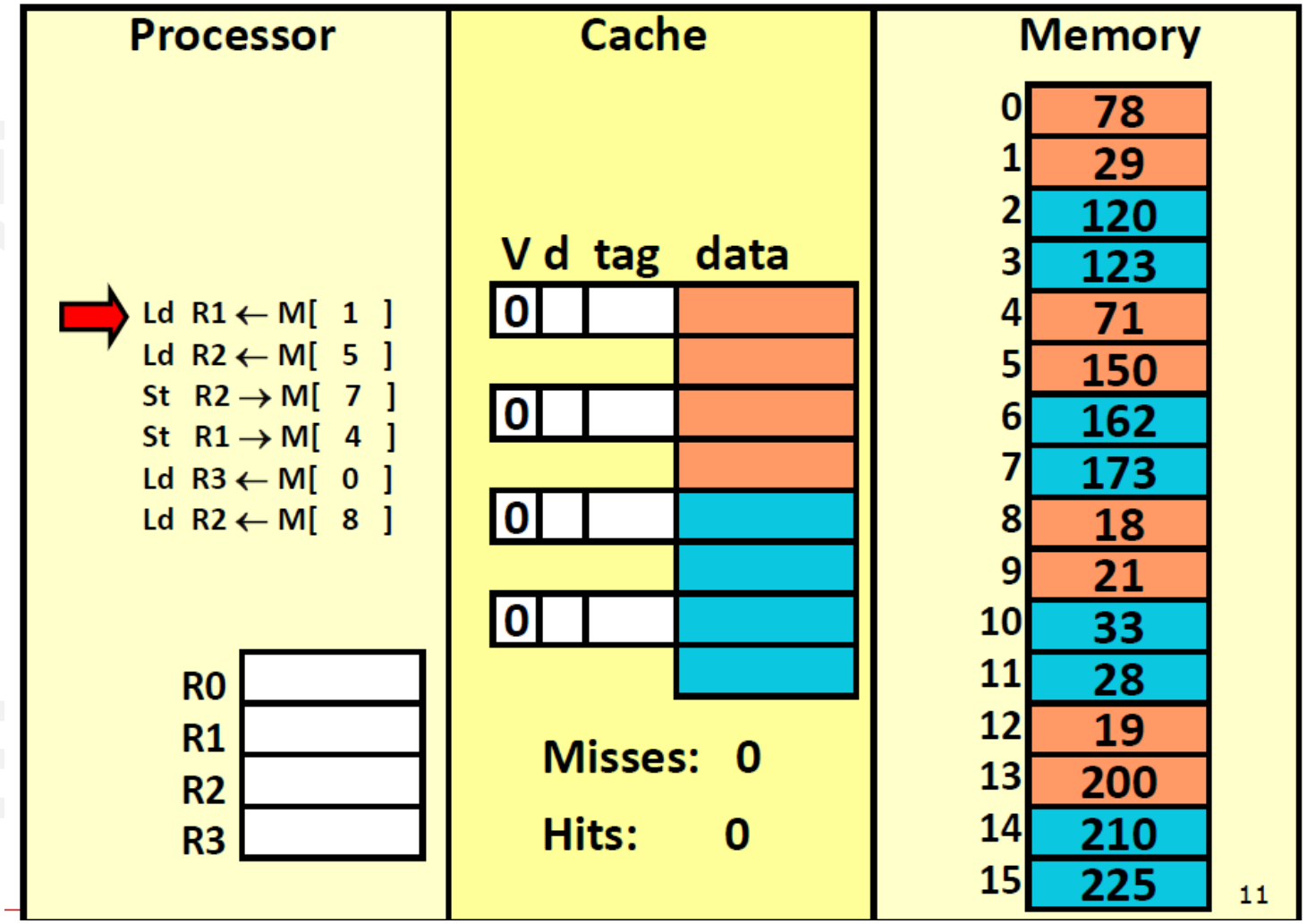
2-Way Set Associative Cache实例

- Write-back & Write-allocate



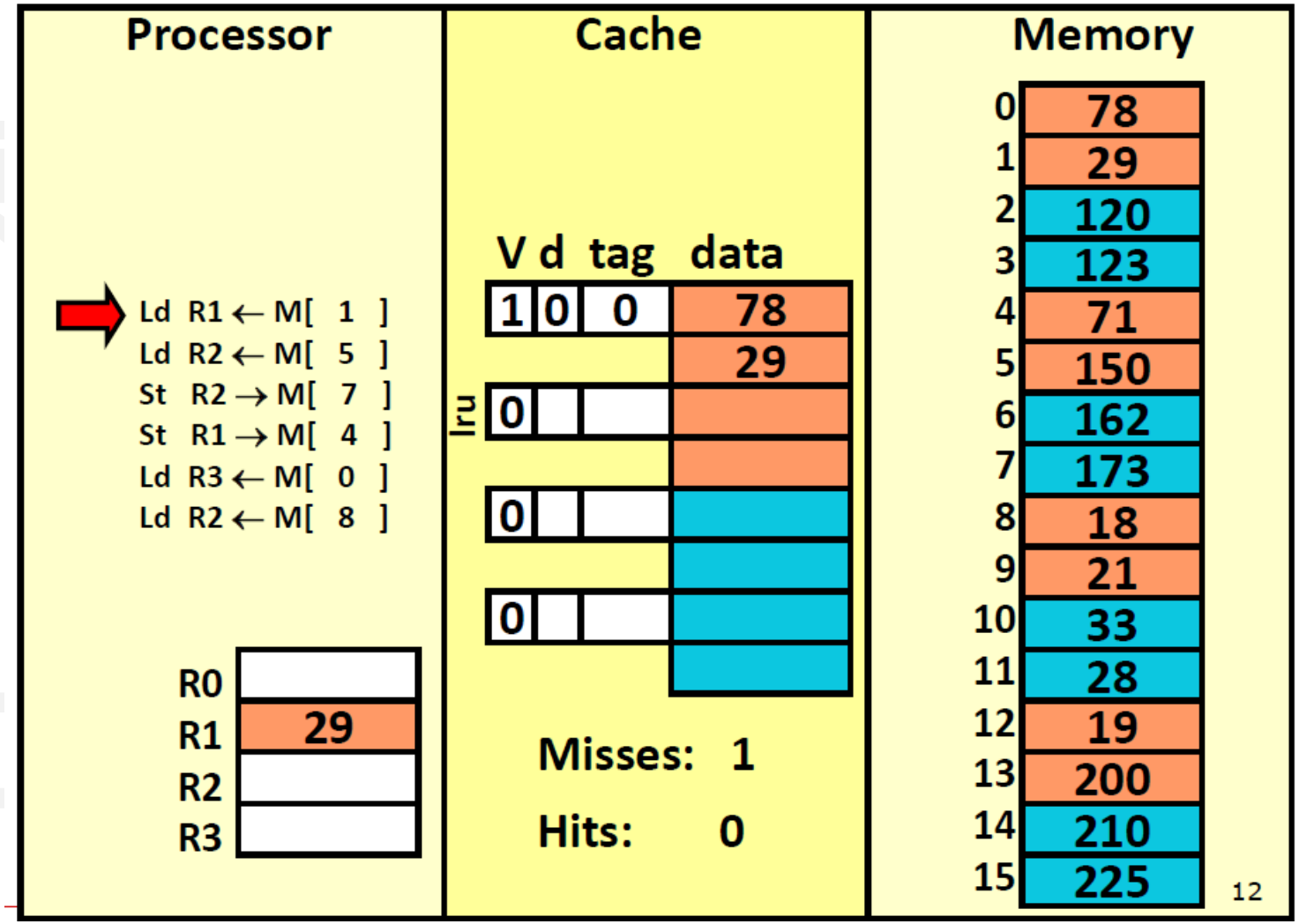
2-Way Set Associative Cache实例

- Write-back & Write-allocate



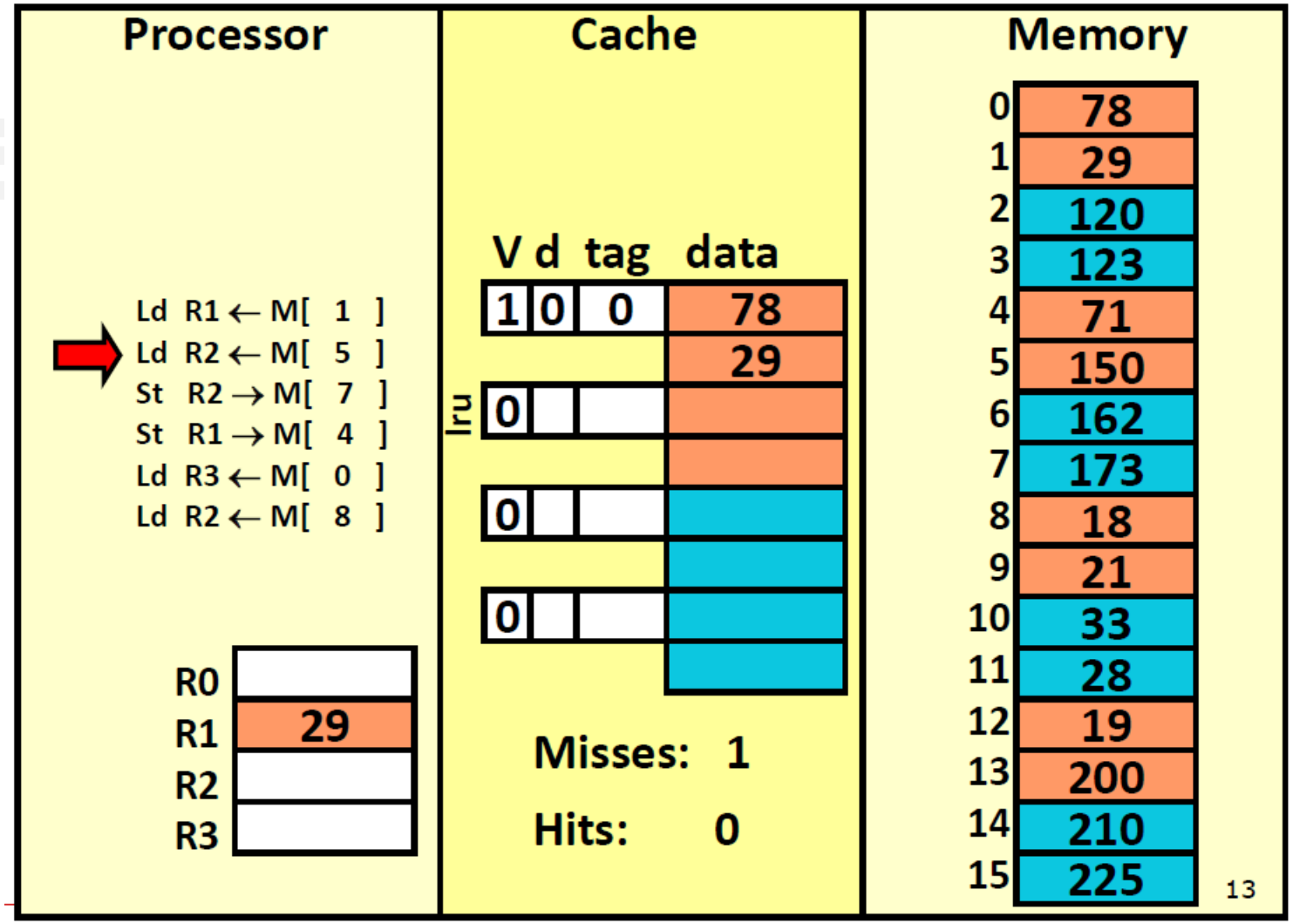
2-Way Set Associative Cache实例

- Write-back & Write-allocate



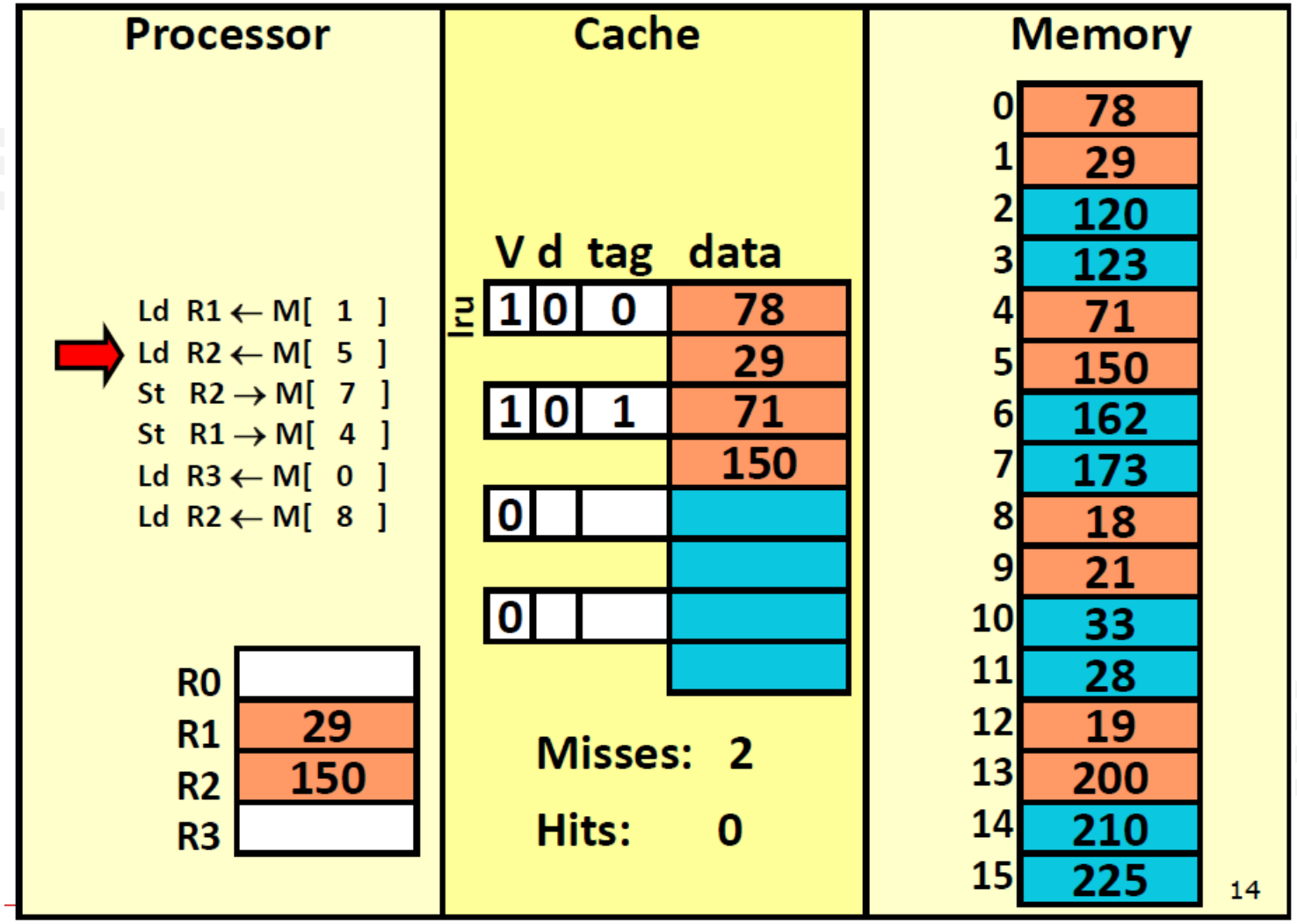
2-Way Set Associative Cache实例

- Write-back & Write-allocate



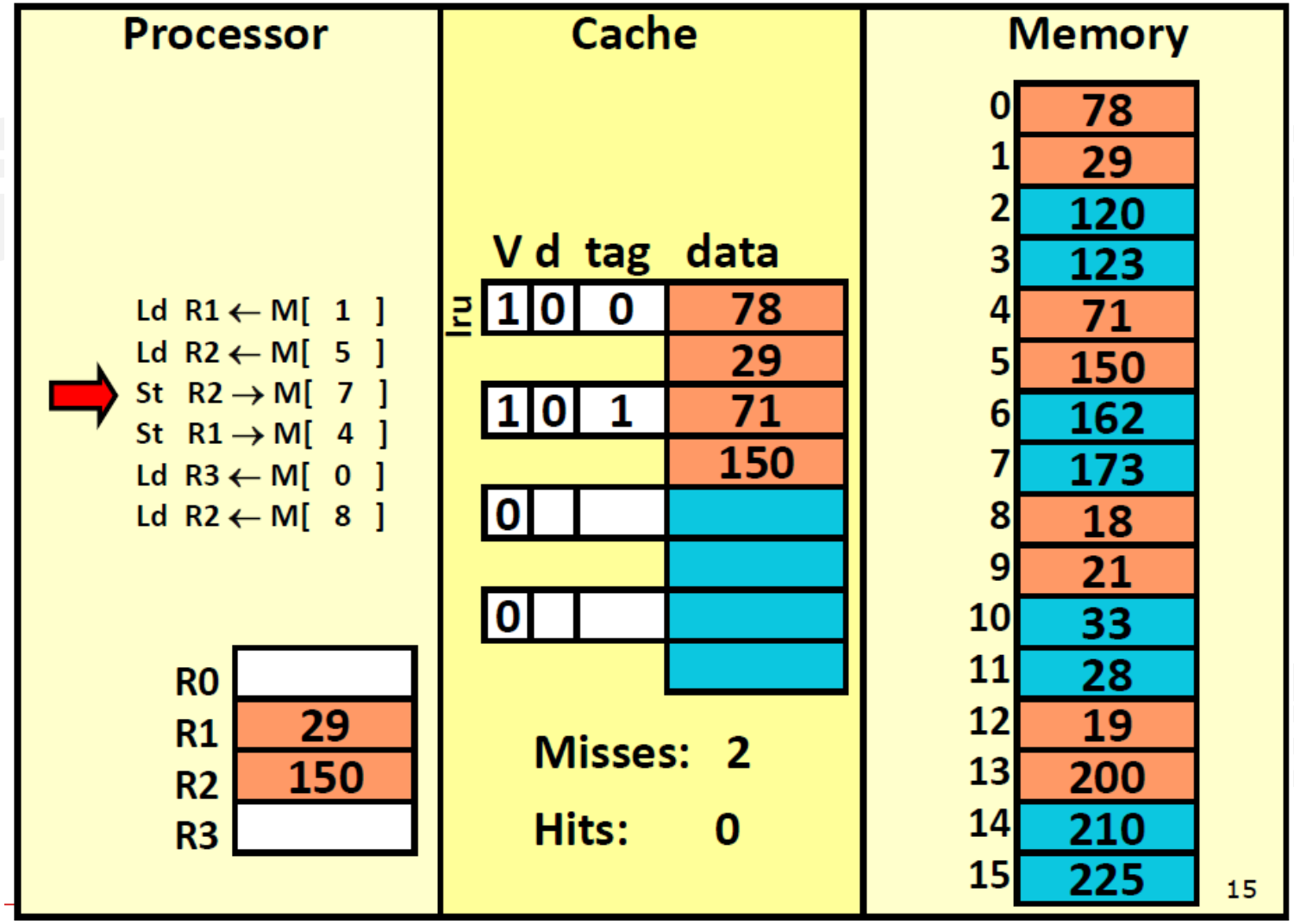
2-Way Set Associative Cache实例

- Write-back & Write-allocate



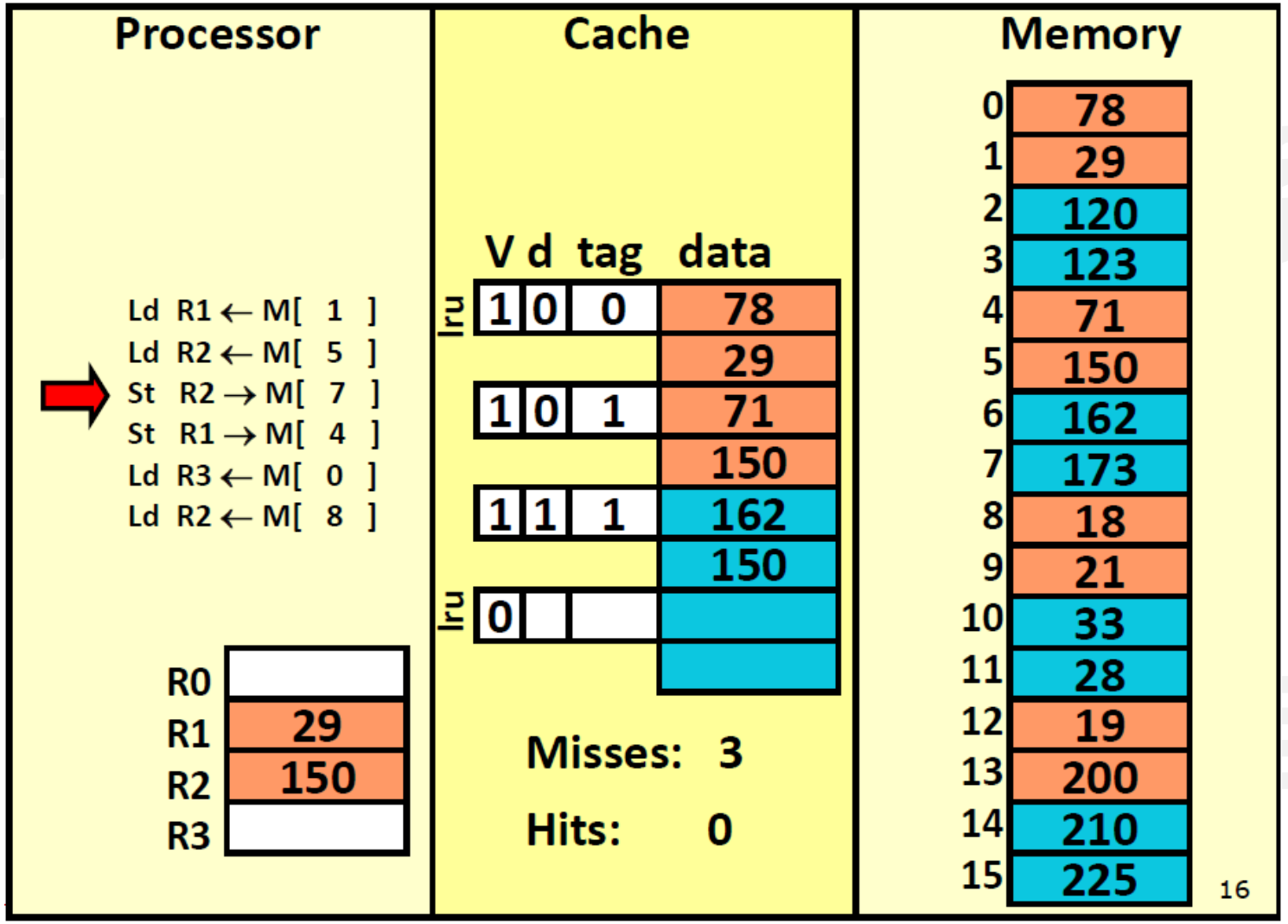
2-Way Set Associative Cache实例

- Write-back & Write-allocate



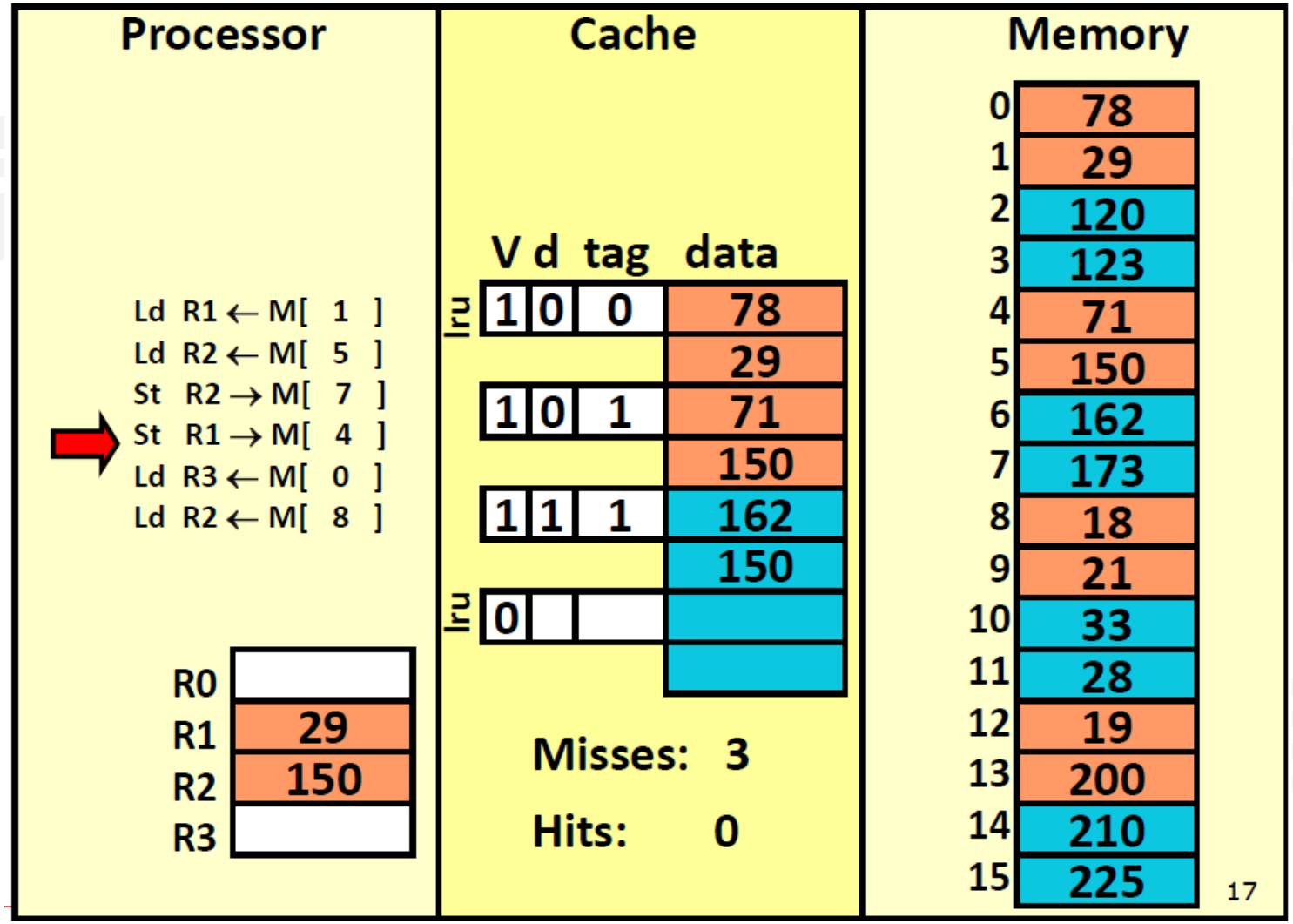
2-Way Set Associative Cache实例

- Write-back & Write-allocate



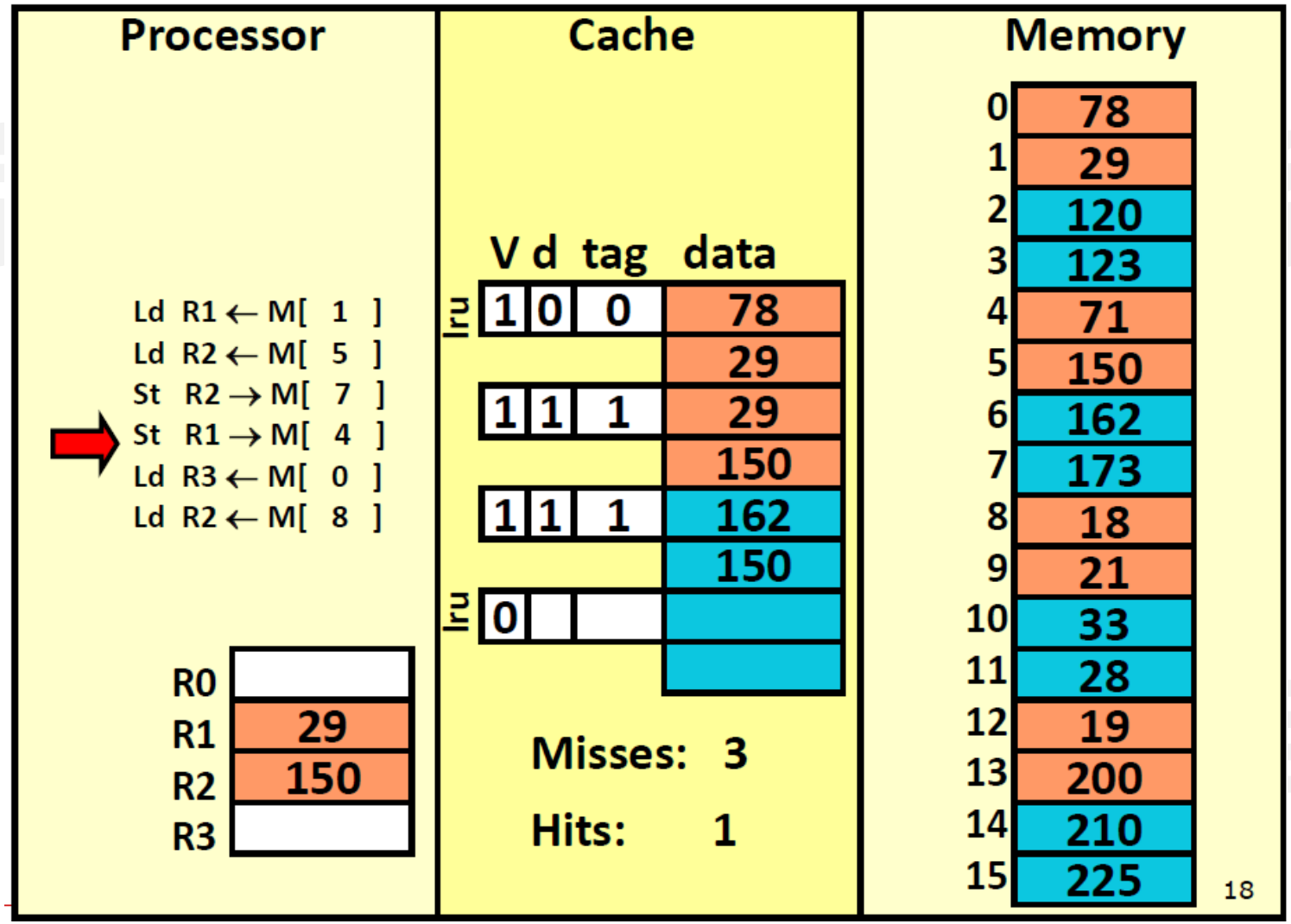
2-Way Set Associative Cache实例

- Write-back & Write-allocate



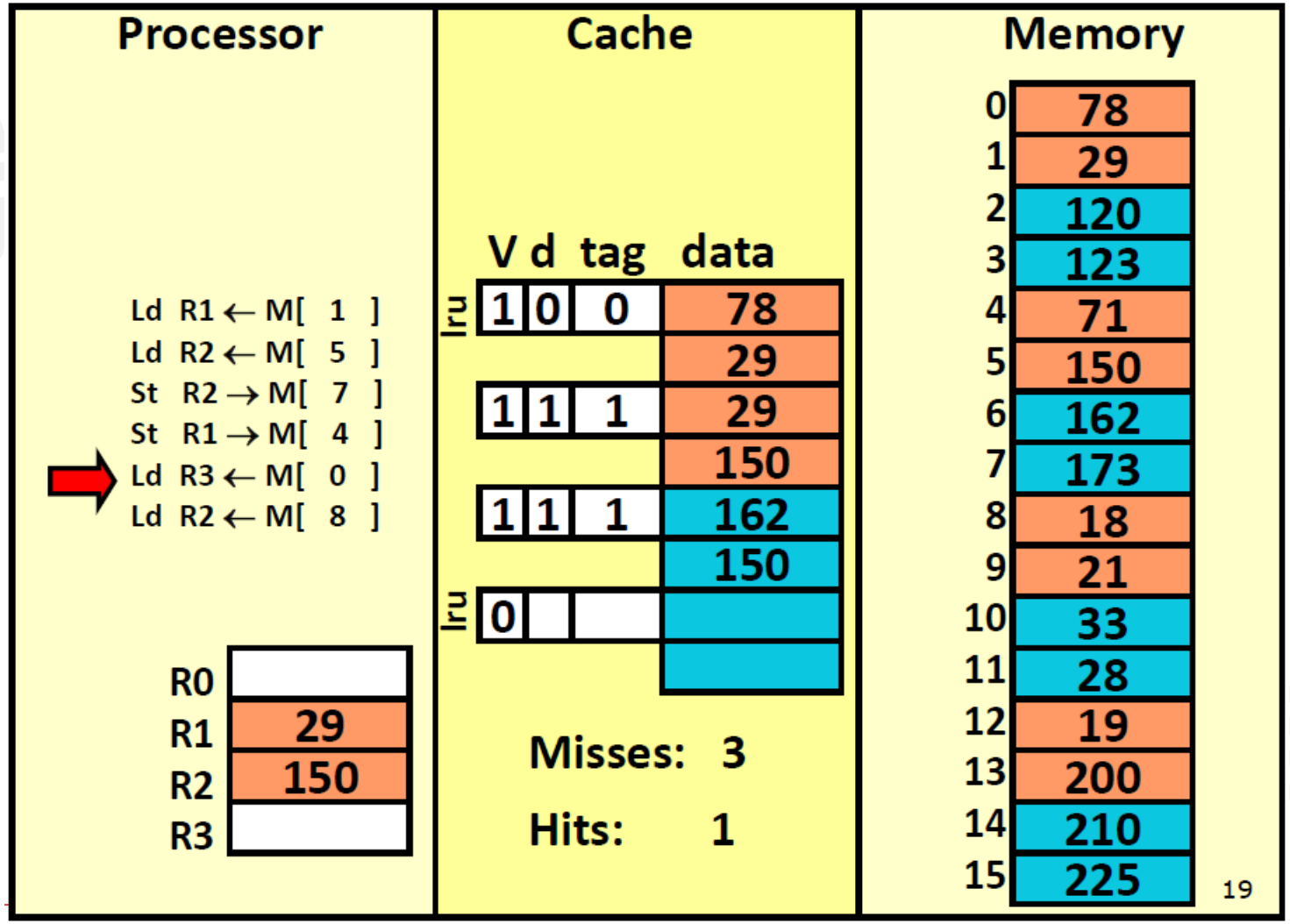
2-Way Set Associative Cache实例

- Write-back & Write-allocate



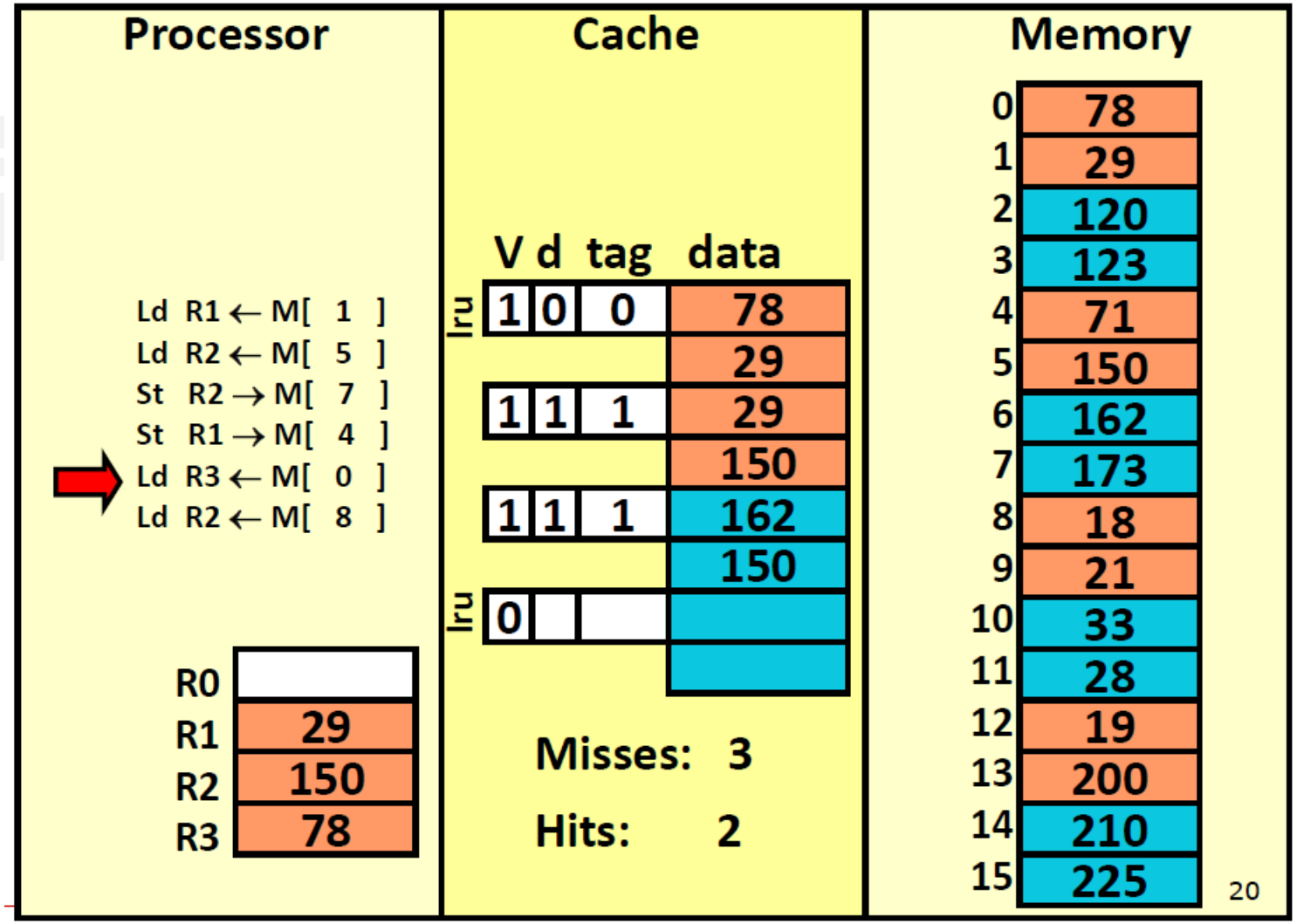
2-Way Set Associative Cache实例

- Write-back & Write-allocate



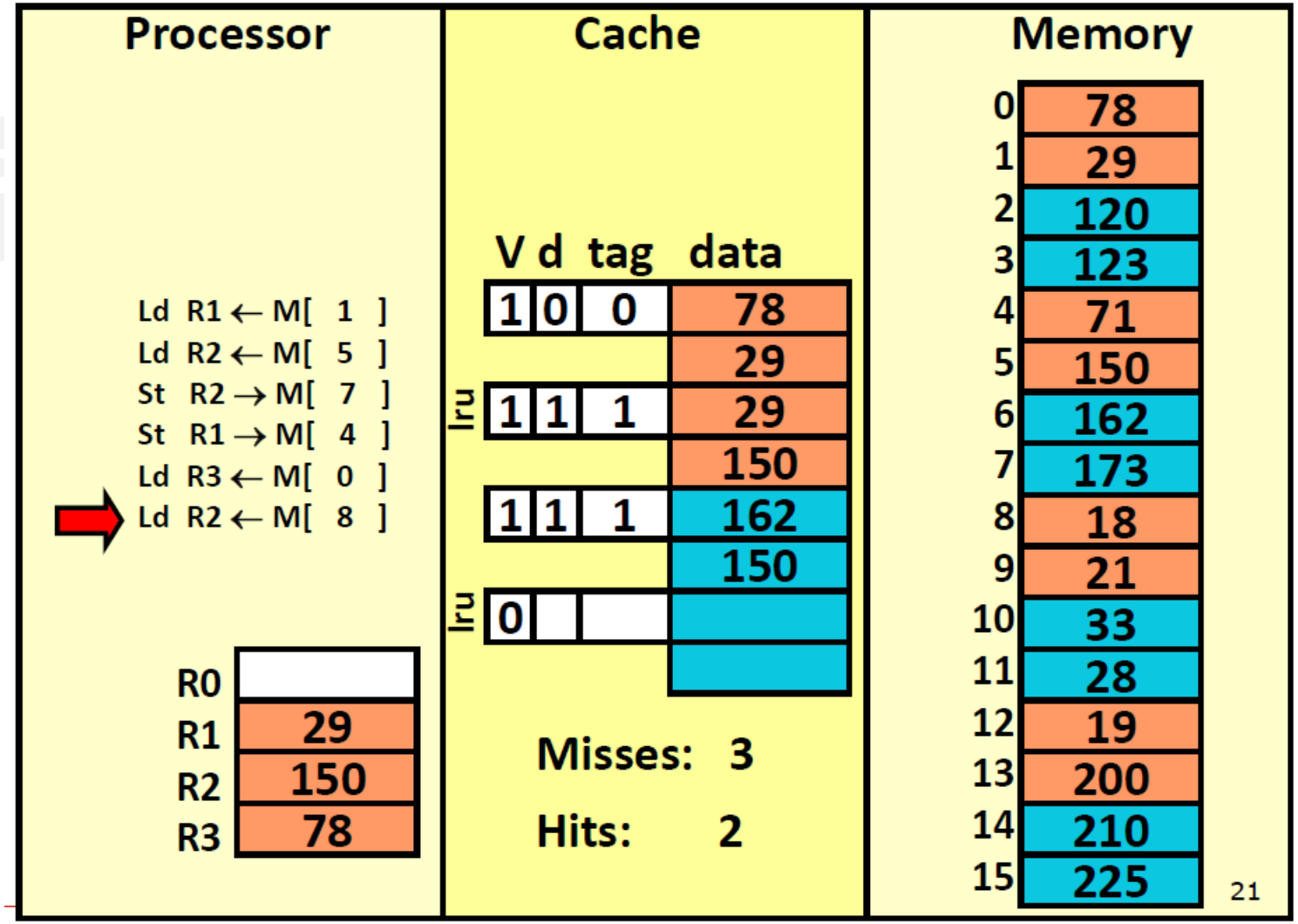
2-Way Set Associative Cache实例

- Write-back & Write-allocate



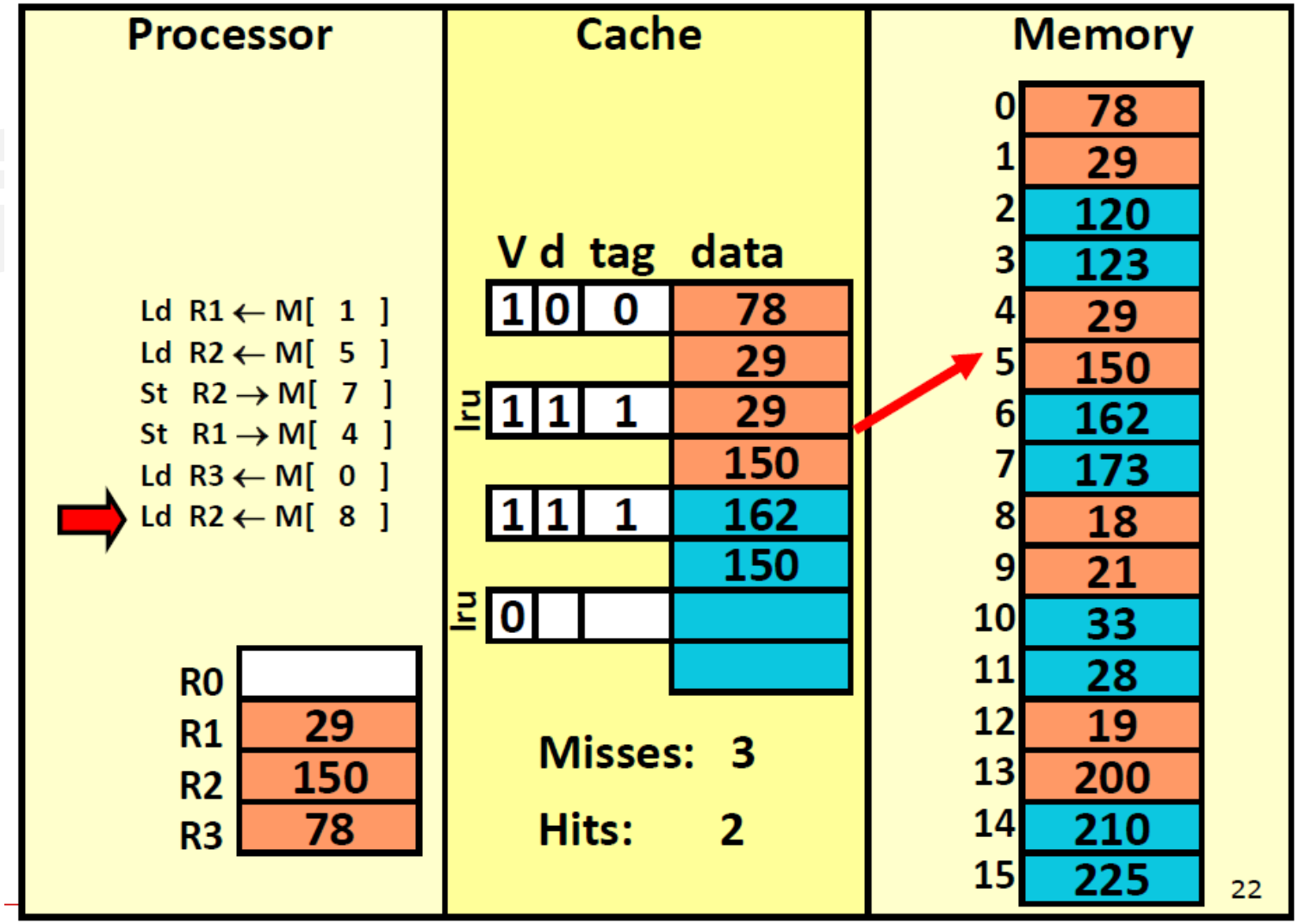
2-Way Set Associative Cache实例

- Write-back & Write-allocate



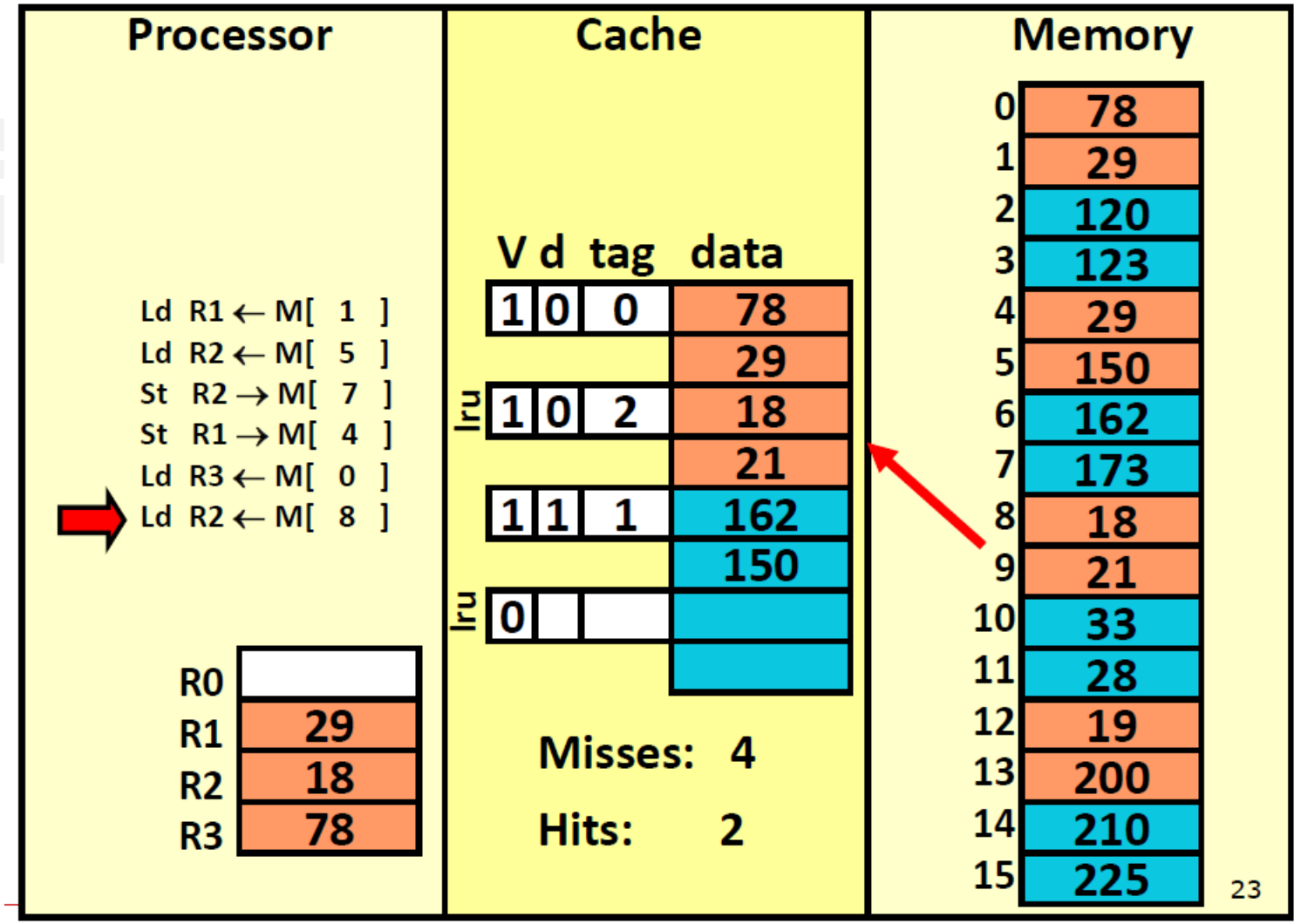
2-Way Set Associative Cache实例

- Write-back & Write-allocate



2-Way Set Associative Cache实例

- Write-back & Write-allocate



案例分析

- Cache地址的比特分区映射

对于 32 位地址和具有 64 字节block的 16KB 缓存，不同缓存配置的地址breakdown：

A) fully associative cache

Block Offset = 6 bits
Tag = 32 - 6 = 26 bits

B) 4-way set associative cache

Block Offset = 6 bits
#sets = #lines / ways = 64
Set Index = 6 bits
Tag = 32 - 6 - 6 = 20 bits

C) Direct-mapped cache

Block Offset = 6 bits
#lines = 256 Line Index = 8 bits
Tag = 32 – 6 – 8 = 18 bits

案例分析

- Cache Access时间分析

  - **T_avg = T_hit + miss_ratio x T_miss**
    - comparable DM and SA caches with same T_miss
    - 但是最小化T_avg的associativity一般比最小化miss_ratio 的associativity要小

$$diff(t_{cache}) = t_{cache}(SA) - t_{cache}(DM) \geq 0$$

$$diff(miss) = miss(SA) - miss\ (DM) \leq 0$$

e.g.,

assuming $diff(t_{cache}) = 0$ => SA better

assuming $diff(miss) = -1\%$, $t_{miss} = 20$

$\Rightarrow$ if $diff(t_{cache}) > 0.2$ cycle then SA loses

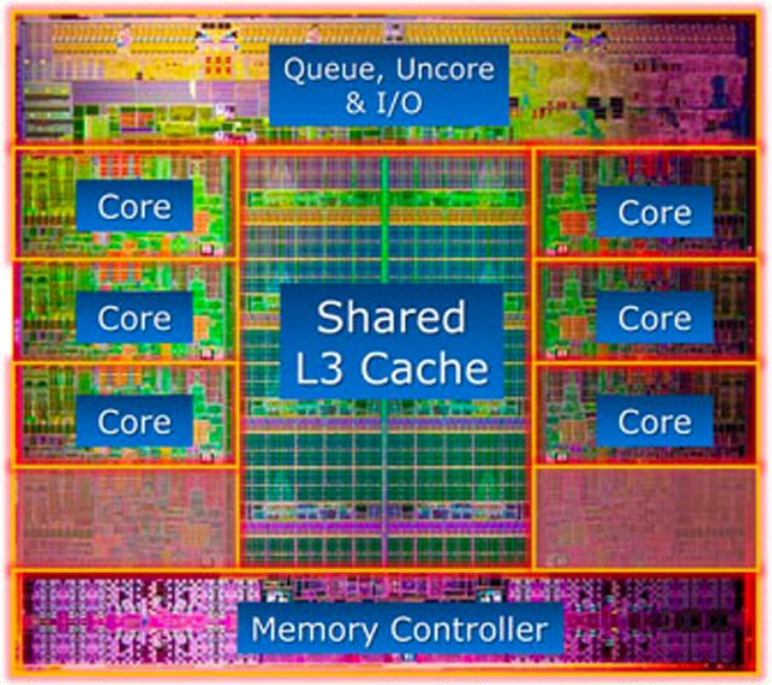
目 录

CONTENTS

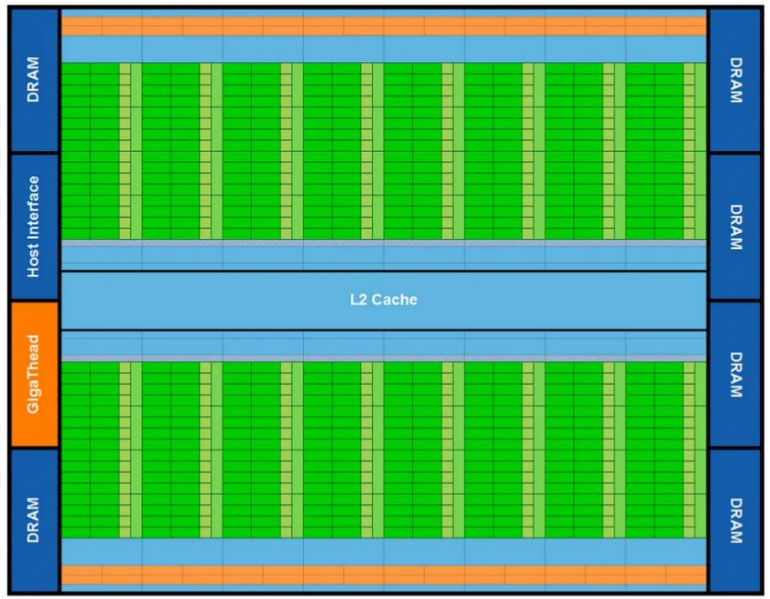
缓存一致性的来源

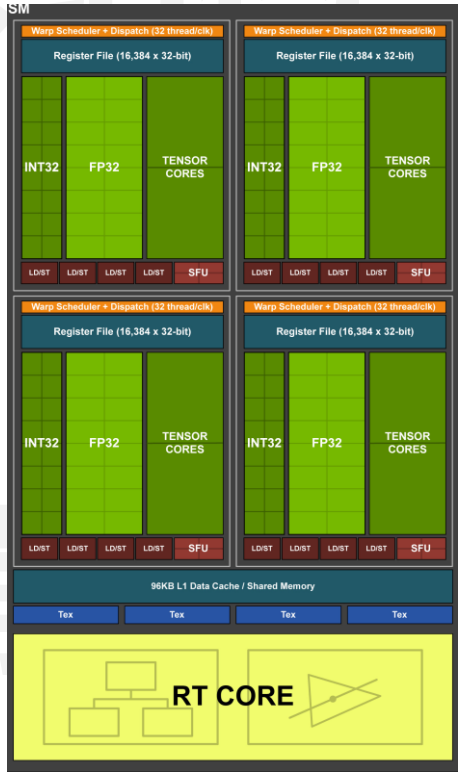
- 多核共享cache

Intel® Core™ i7-3960X Processor Die Detail



- 30% of the die area is cache





Nvidia GPU Architecture

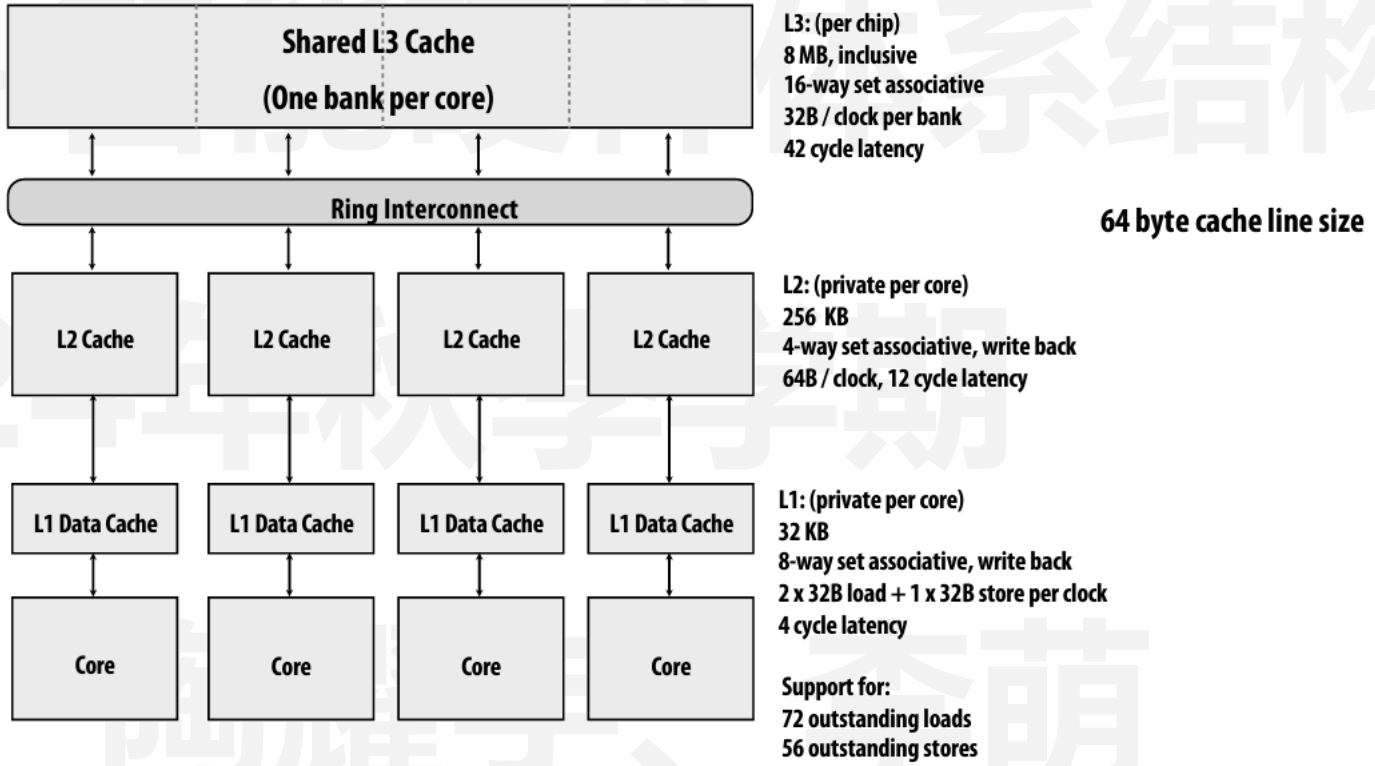
缓存一致性的来源

- 多核共享cache

3 Cs cache miss model

- Cold
- Capacity
- Conflict

Caches exploit locality

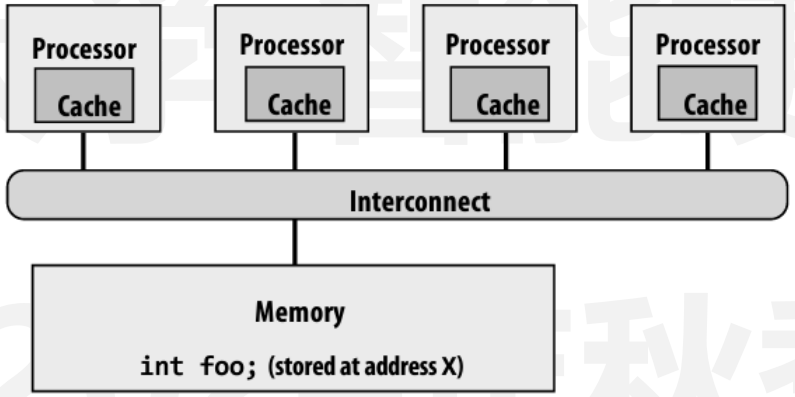


Source: Intel 64 and IA-32 Architectures Optimization Reference Manual (June 2016)

缓存一致性的来源

- 多核共享cache

现代处理器在局部缓存中复制存储器的内容
问题：不同的处理器在同样内存位置可能会观察到不同的值



The chart at right shows the value of variable foo (stored at address X) in main memory and in each processor's cache

Assume the initial value stored at address X is 0

Assume write-back cache behavior

Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (assume this load causes eviction of X)		0	2		1

缓存一致性的来源

- 缓存一致性协议

- **每个处理器的缓存控制器应该响应以下操作:**
  - -本地处理器Loads and stores
  - -来自总线上其他缓存的消息
- **如果所有缓存控制器都按照所述协议运行，那么将保持一致性**
  - - 缓存“合作”以确保保持一致性

缓存一致性的来源

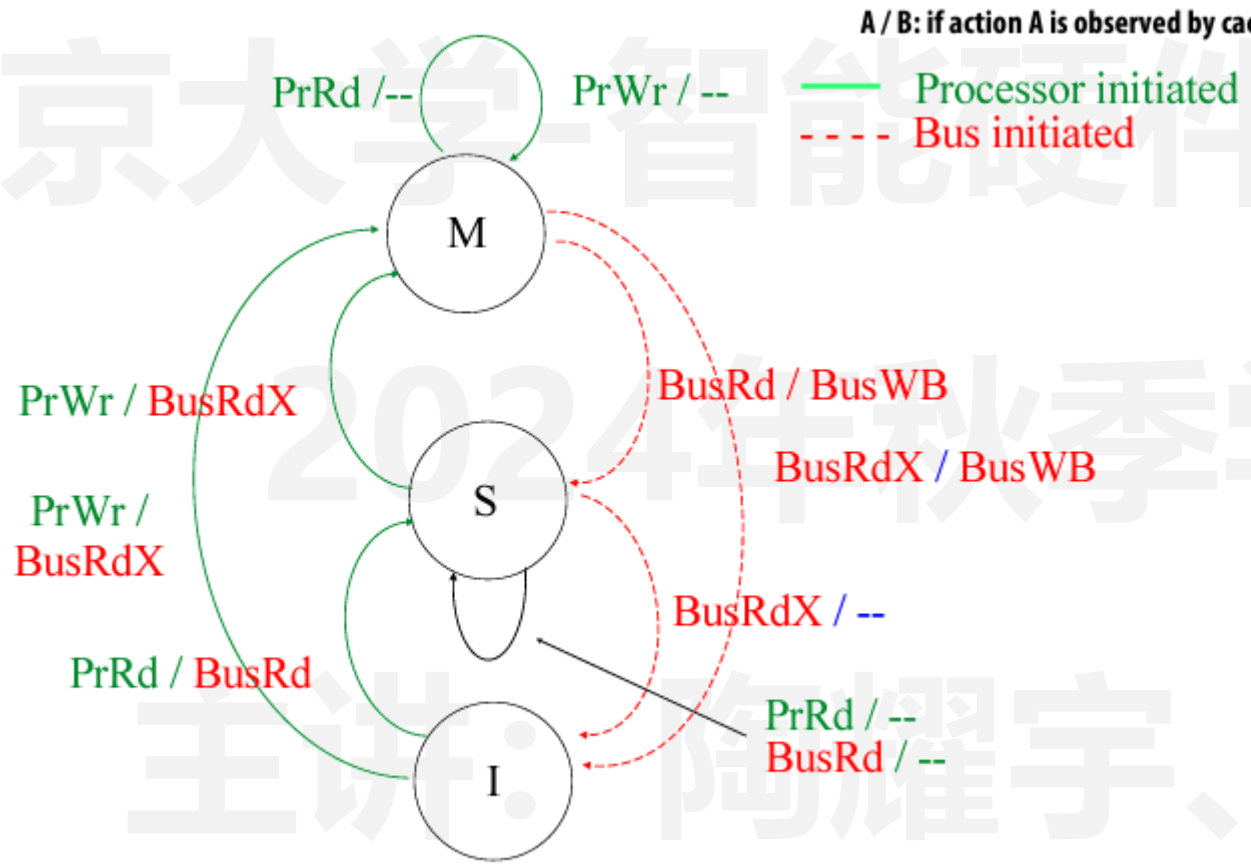
- Invalidation-based write-back protocol
  - **关键思想：**
    - 处于“已修改”状态的行可以在不通知其他缓存的情况下进行修改
  - **处理器只能写入已修改状态的行**
    - 需要一种方法来告诉其他缓存，处理器想要独占访问该行
    - 我们通过向所有其他缓存发送消息来实现这一点
  - **当缓存控制器看到对其所包含的行进行修改访问的请求时**
    - 它必须使缓存中的行无效

缓存一致性

- MSI write-back invalidation protocol
  - **协议的主要任务**
    - 确保处理器获得写入的独占访问权限
    - 在缓存未命中时查找缓存行数据的最新副本
  - **三种缓存行状态**
    - 无效（I）：与单处理器缓存中的无效含义相同
    - 共享 (S)：行在一个或多个缓存中有效，内存是最新的
    - 已修改 (M)：仅在一个缓存中有效的行（又称“dirty”或“exclusive”状态）
  - **两个处理器操作（由本地 CPU 触发）**
    - PrRd(读取)
    - PrWr(写入)
  - **三个与一致性相关的总线操作（来自远程缓存）**
    - BusRd：获取缓存行副本，无意修改
    - BusRdX：获取要修改的缓存行副本
    - BusWB：将dirty行写入内存

缓存一致性

- Cache Coherence Protocol: MSI 状态图



Abbreviation	Action
PrRd	Processor Read
PrWr	Processor Write
BusRd	Bus Read
BusRdX	Bus Read Exclusive
BusWB	Bus Writeback

缓存一致性

- Cache Coherence Protocol: MSI State Diagram

Proc Action	P1 State	P2 state	P3 state	Bus Act	Data from
1. P1 read x	S	--	--	BusRd	Memory
2. P3 read x	S	--	S	BusRd	Memory
3. P3 write x	I	--	M	BusRdX	Memory
4. P1 read x	S	--	S	BusRd	P3's cache
5. P2 read x	S	S	S	BusRd	Memory
6. P2 write x	I	M	I	BusRdX	Memory

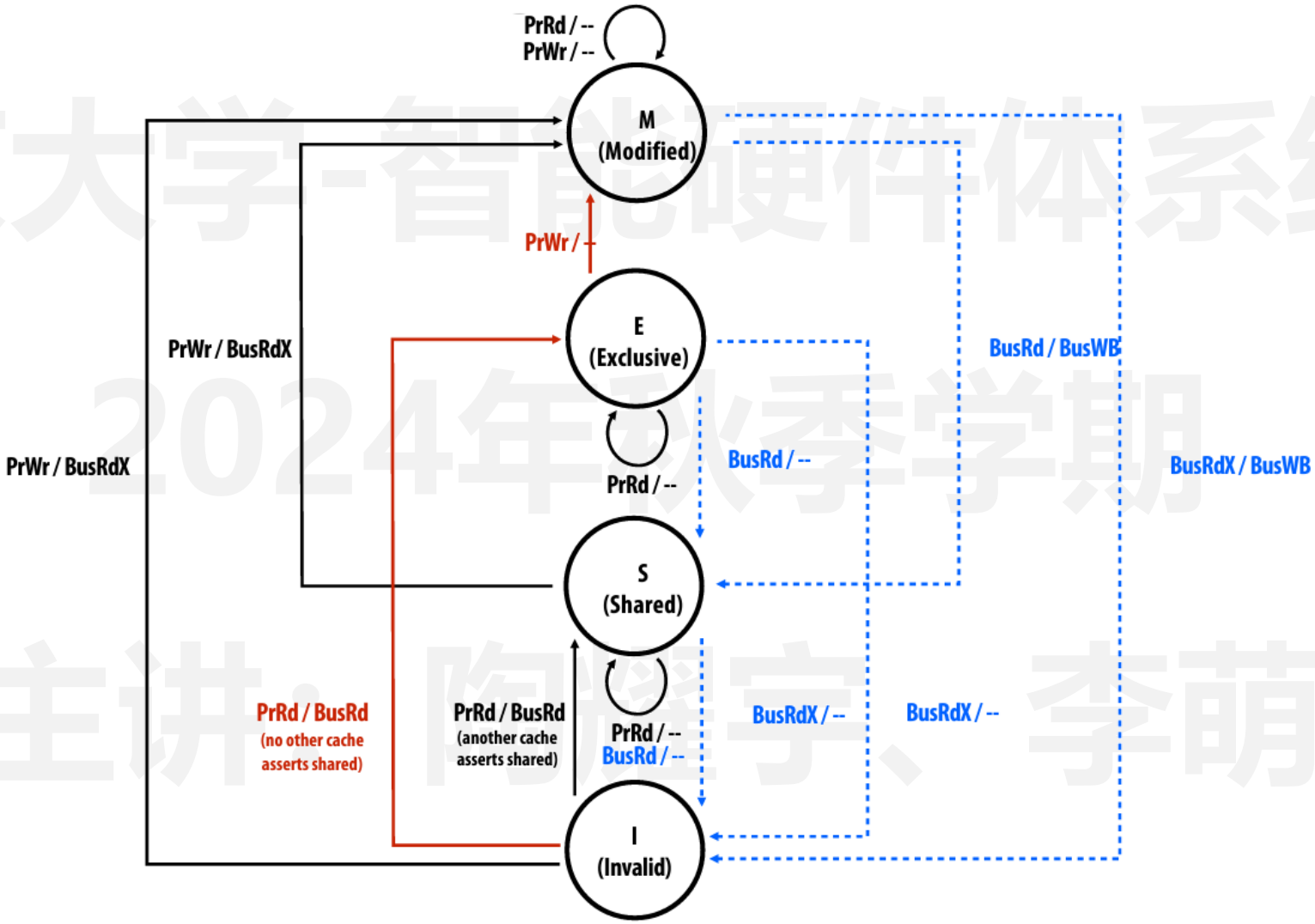
缓存一致性

- Cache Coherence Protocol: MESI invalidation protocol

- **对于读取地址然后写入地址的常见情况，MSI 需要两个互联操作**
  - 操作 1：BusRd 把 I 状态转为 S 状态
  - 操作 2：BusRdX 把 S 状态转为 M 状态
- **即使应用程序根本没有共享，这种低效率仍然存在**
- **解决方案：添加附加状态 E(“exclusive clean”)**
  - 行未被修改，但只有此缓存有该行的副本
  - 将exclusivity与行所有权分离（线路不脏，因此内存中的副本是数据的有效副本）
  - 从 E 升级到 M 不需要总线操作

缓存一致性

- Cache Coherence Protocol: MESI invalidation protocol

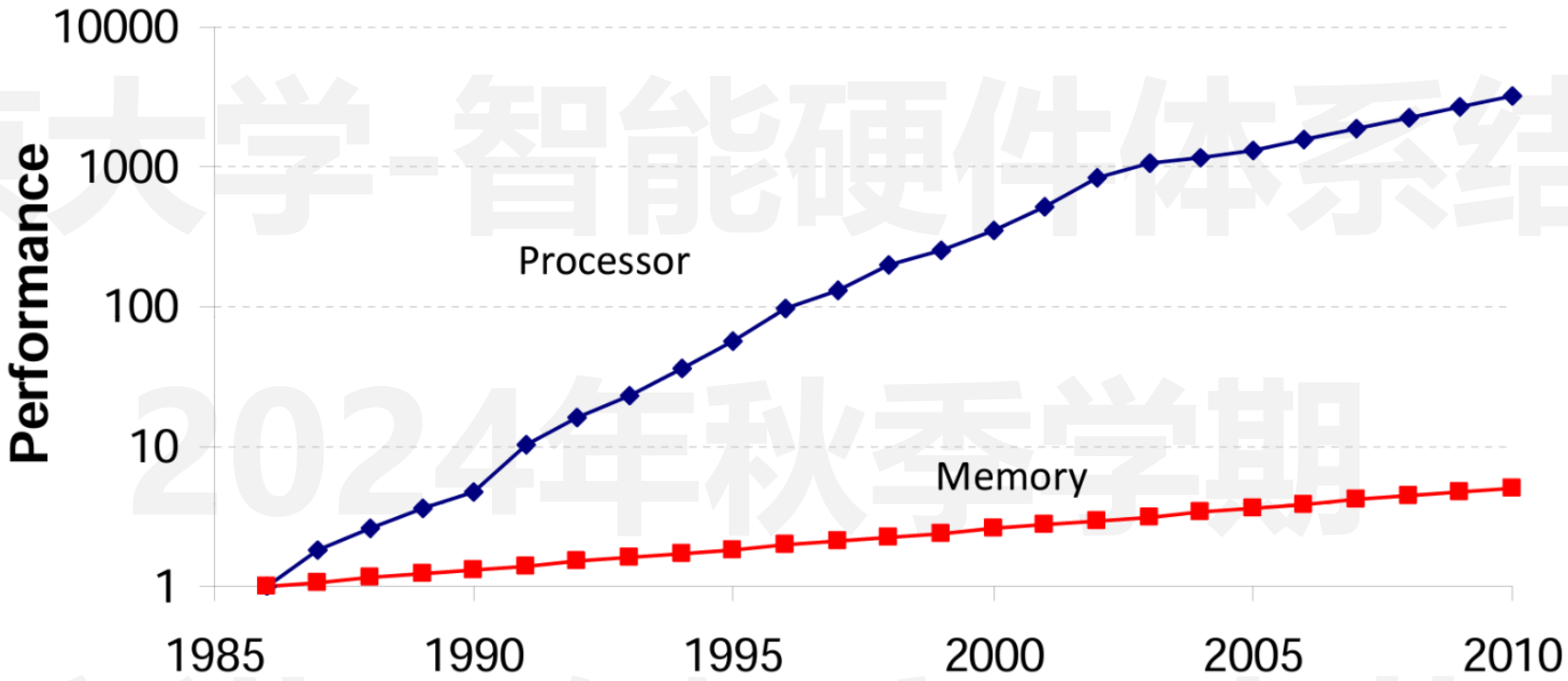


目 录

CONTENTS

缓存预读取优化机制

- Memory Wall



Source: Hennessy & Patterson, *Computer Architecture: A Quantitative Approach, 4th ed.*

Today: 1 mem access ≈ 500 arithmetic ops

How to reduce memory stalls for existing SW?

缓存预读取优化机制

北京大学 PEKING UNIVERSITY

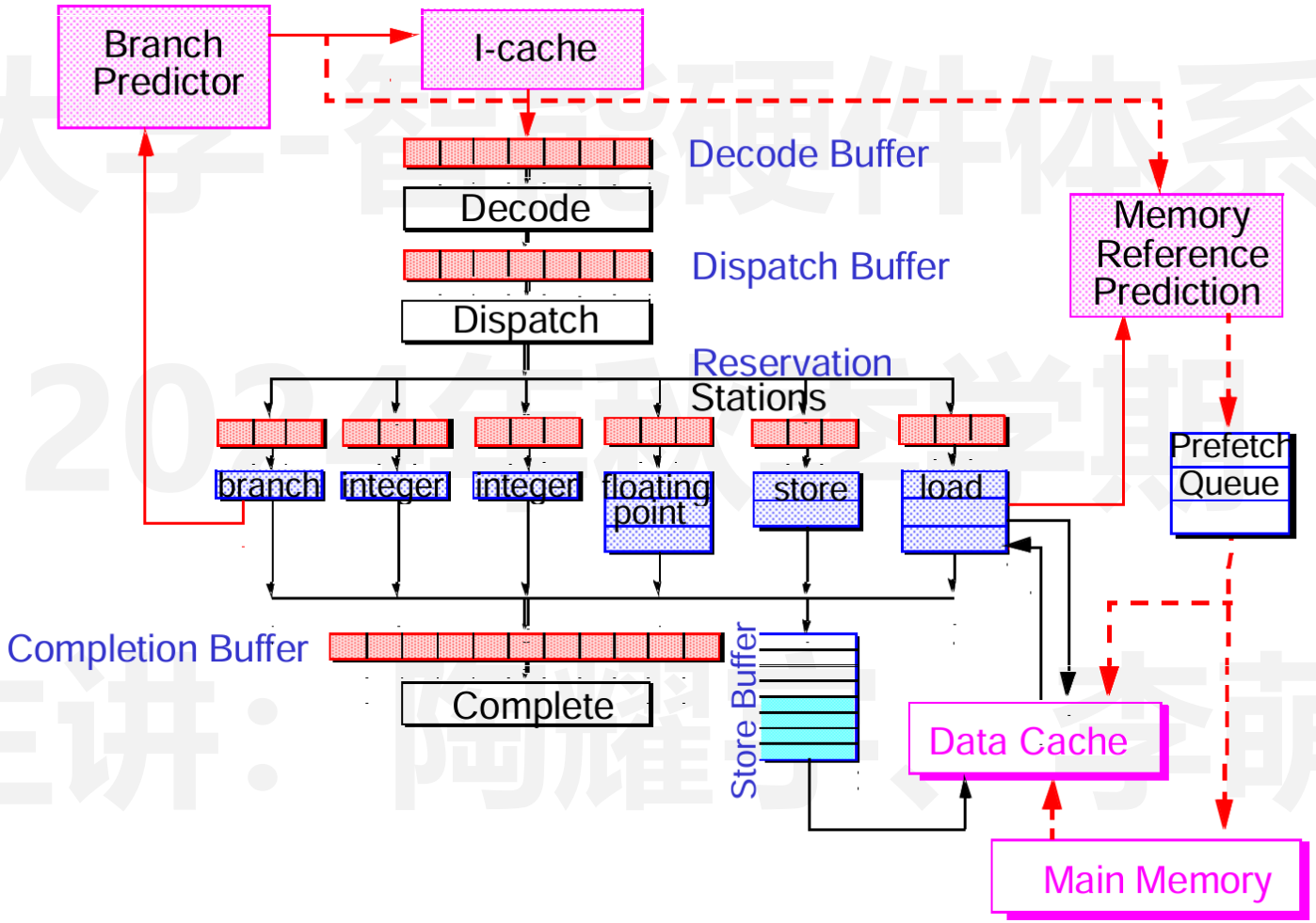
- 什么是预读取（Prefetch）

  - 提前获取内存
  - 面向解决compulsory, capacity, & coherence misses

  - 挑战：
    - **知道要取什么**
      - 获取无用的信息会浪费宝贵的资源
    - **“何时”获取**
      - 过早获取会导致存储混乱
      - 太晚获取会违背“预”获取的目的

缓存预读取优化机制

- 预读取（Prefetch）缓存设计示意图



基于软件的预读取

- 需要编译器和指令集的支持

- Compiler/programmer places prefetch instructions
  - Requires ISA support
  - Why not use regular loads?
  - Found in ISA’s such as SPARC V-9
- Prefetch into
  - register (binding)
  - caches (non-binding): preferred in multiprocessors

```
for (I = 1; I < rows; I++)
    for (J = 1; J < columns; J++)
    {
        prefetch(&x[I+1,J]);
        sum = sum + x[I,J];
    }
```

基于硬件的预读取

- 无需要编译器和指令集的支持

  - **预取什么?**
    - 空间上领先一个block?
    - 使用地址预测器->适用于常规模式(e.g., x, x+8, …)
  - **什么时候预取?**
    - 每次引用
    - 每次miss
    - 当引用先前预取的数据时
    - 最后一次处理器引用时
  - **预取的数据放在哪里?**
    - Buffers、caches

基于硬件的预读取

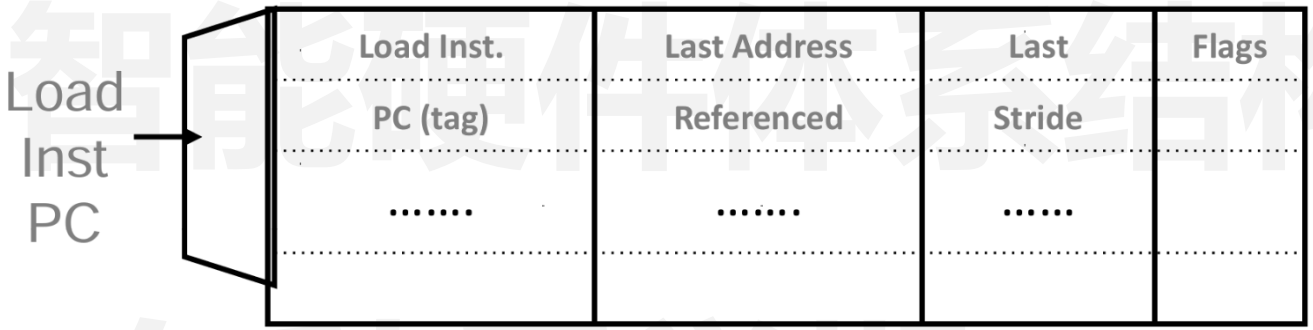
- 利用空间局部性的顺序预读取
  - **适用于 I-cache**
    - 指令获取倾向于顺序访问内存
  - **对于 D-cache 来说效果不太好**
    - 更不规则的访问模式
    - 常规模式可能具有非单位步长（例如矩阵代码）
  - **相对容易实现**
    - 较大的缓存块大小已经起到预取的作用
    - 加载一行缓存后，如果该行不在缓存中且总线不忙，则自动开始加载下一行
  - **如果你在错误的时间预取会怎么样 ….**
    - 分支, etc…

基于硬件的预读取

- Stride Prefetchers
  - **特定静态负载的访问模式更可预测**

Reference Prediction Table

(**RPT**)



- 记住先前执行的加载、它们的 PC、最后引用的地址、在最后两个引用之间的步幅
- 执行加载时，查找**RPT**并计算当前数据地址和最后一个地址之间的距离
  - **如果新的距离与旧的步幅匹配** ⇒

    找到一个模式，继续预取“当前地址+步幅”
    - 更新“last addr”和“last stride”以供下次查找

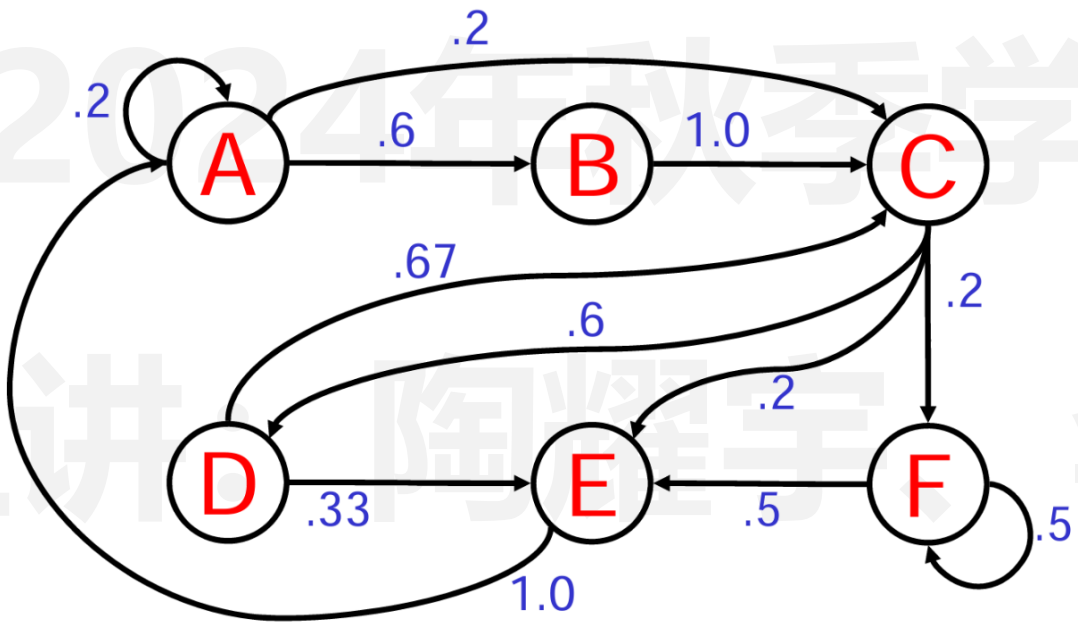
基于硬件的预读取

- Correlation-Based Prefetchers

考虑以下处理器发出的加载地址的历史记录

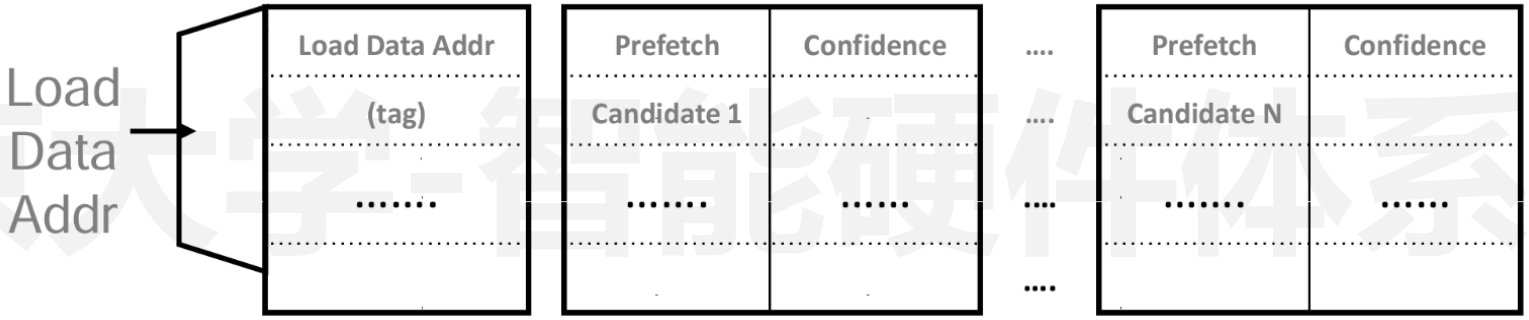
A, B, C, D, C, E, **A, C**, F, F, E, **A, A**, B, C, D, E, **A ,B** C, D, C

在引用特定地址（例如 A 或 E）后，某些地址是否接下来更可能会被引用



基于硬件的预读取

- Correlation-Based Prefetchers



- **看到特定地址后跟踪可能的下一个地址。**
- 预取精度通常较低，因此预取最多*N*个next地址以增加覆盖范围（但这会浪费带宽）
- 通过使用更长的历史记录可以提高预取准确性
- 通过查看最后K个加载地址来决定下一个预取的地址，而不仅仅是当前地址
  - 例如，使用最后 K 个负载的数据地址的 XOR 进行索引
  - 使用几次加载的历史记录可以显著提高准确性
  - 此技术也可应用于仅加载未命中stream

基于硬件的预读取

- 更多的预读取机制

Miss 地址是相关的

- Intuition: Miss sequences repeat
  - Because code/data traversals repeat

- Temporal Address Correlation
  - Prior evidence: [Joseph 97][Luk 99][Chilimbi 01][Lai 01]
  - Contrast: temporal locality

stream = ordered address sequence

目 录

CONTENTS

数据并行

- **Data-level parallelism (DLP)**

Clock rate and IPC are at odds with each other

- Pipelining
  - Fast clock
  - Increased hazards lower IPC
- Wide issue
  - Higher IPC
  - N^2 bypassing slows down clock

- 可以同时获得高频时钟和发射宽度吗？
  - 可以，但需要使用不如ILP普遍的并行模型
- 数据级并行（DLP）
  - 对多数据元素重复运行单个操作
  - 不如ILP普遍：并行指令是同样的操作

数据并行

- Data-level parallelism (DLP)

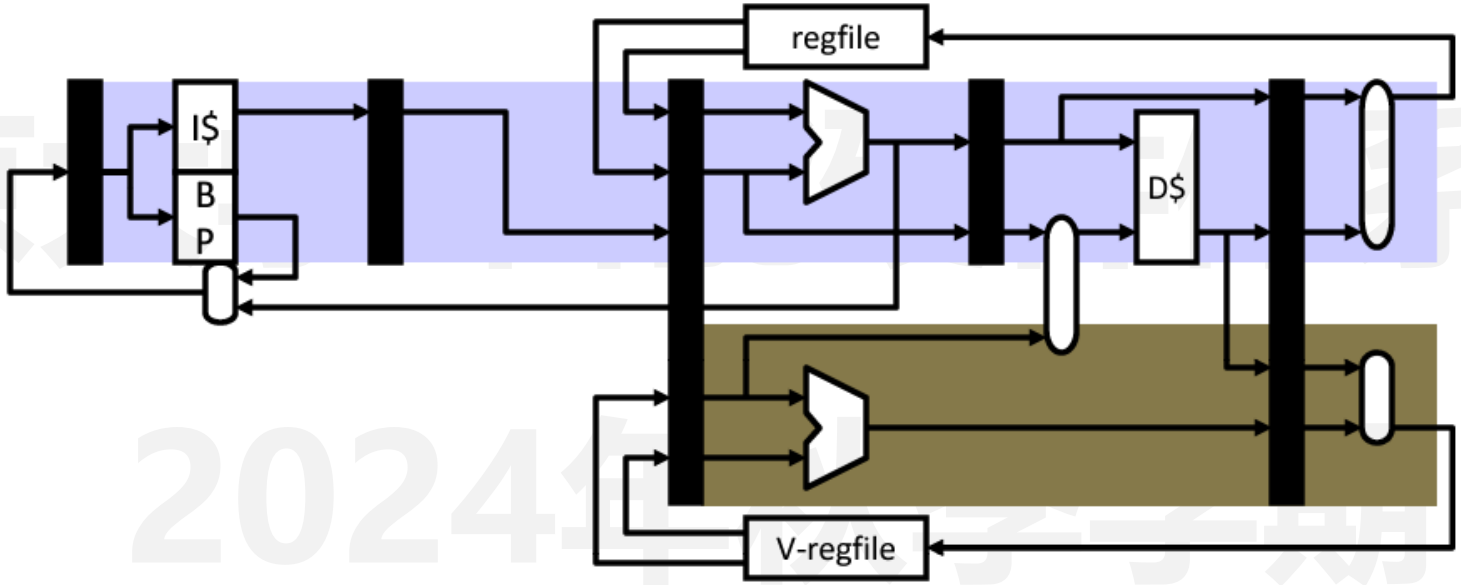
```
for (I = 0; I < 100; I++)
    Z[I] = A*X[I] + Y[I];

L0: ldf X(r1),f1          // I is in r1
    mulf f0,f1,f2         // A is in f0
    ldf Y(r1),f3
    addf f2,f3,f4
    stf f4,Z(r1)
    addi r1,4,r1
    blti r1,400,L0
```

One example of DLP: **inner loop-level parallelism**

- Iterations can be performed in parallel

- Exploiting DLP With Vectors

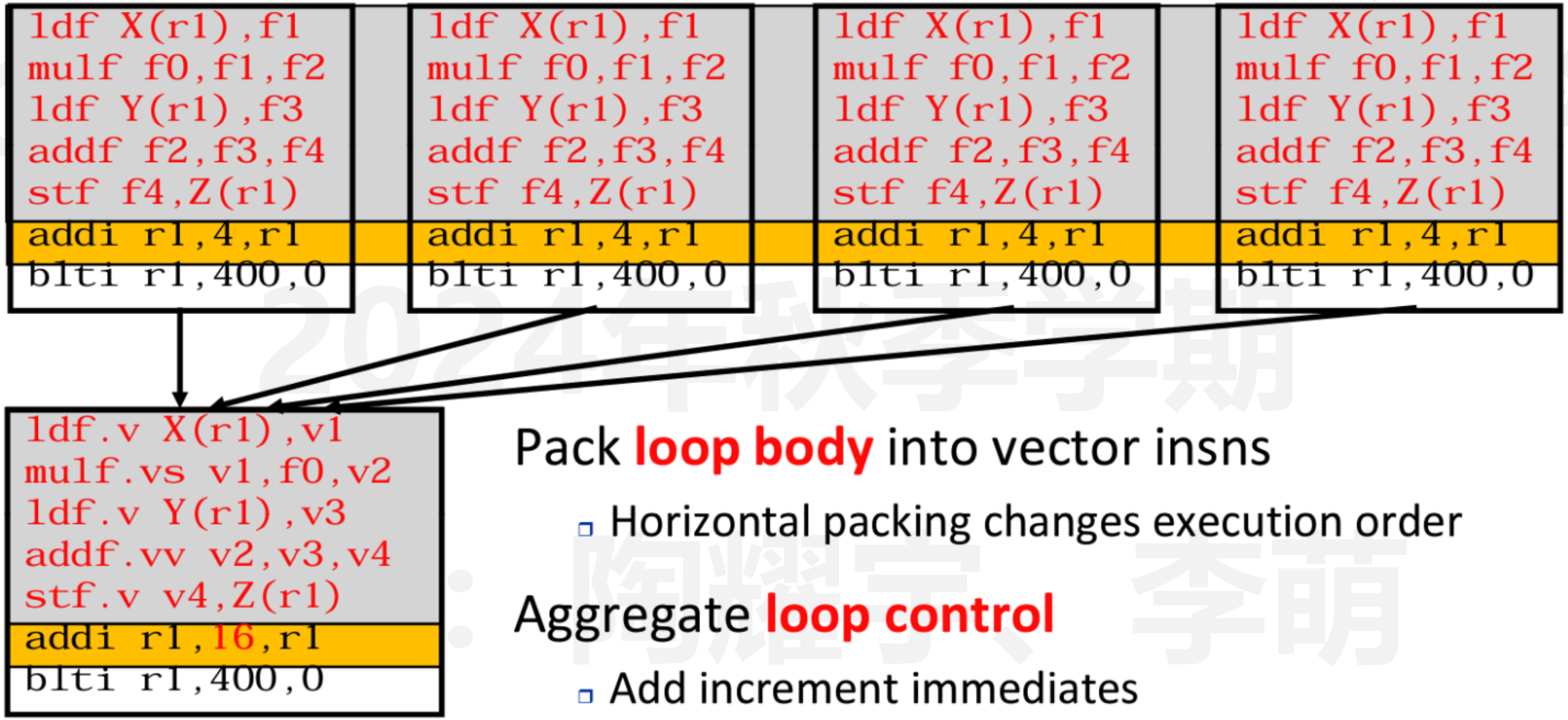


One way to exploit DLP: **vectors**

- Extend processor with **vector "data type"**
- Vector: array of MVL 32-bit FP numbers
  - **Maximum vector length (MVL)**: typically 8–64
- **Vector register file**: 8–16 vector registers (v0–v15)

数据并行

- **Exploiting DLP With Vectors**



Pack **loop body** into vector insns

- Horizontal packing changes execution order

Aggregate **loop control**

- Add increment immediates

数据并行

北京大学 PEKING UNIVERSITY

- **MIPS-V Instructions**

Vector-vector instructions
- operate on two vectors
- produce a third vector
- addv v1, v2, v3

Vector-scalar instructions
- operate on one vector and one scalar
- addv v1, f0, v3

Vector ld/st instructions
- ld/st a vector from memory into a vector register
- operates on contiguous addresses
- lv [r1], v1 ; v[I] = M[r1+I]
- sv v1, [r1] ; M[r1+I] = v[I]

ld/st vector with stride
- vectors are not always contiguous in memory
- add non-unit stride on each access
- lvws [r1,r2], v1 ; v[I] = M[r1+I*r2]
- svws v1, [r1, r2] ; M[r1+I*r2] = v[I]

ld/st indexed
- indirect accesses through an index vector
- lvws [r1,v2], v1 ; v[I] = M[r1+v2[I]]
- svws v1, [r1, v2] ; M[r1+v2[I]] = v[I]

- MIPS-V Instructions

DAXPY: double-precision a * x + y

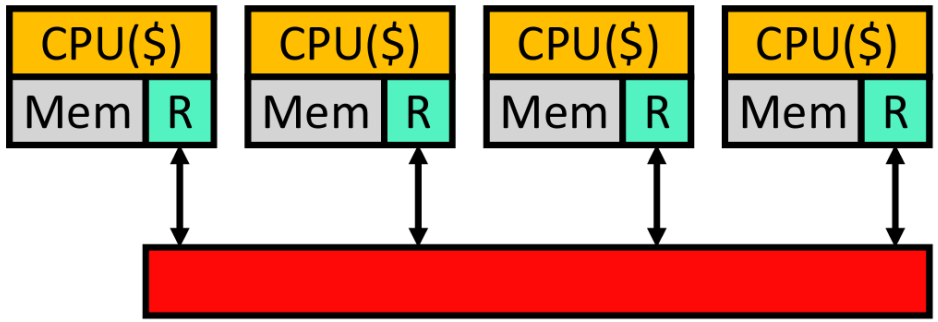
```
for (I=1; I<=64;I++)
  y[I] = a*x[I]+y[I]
```

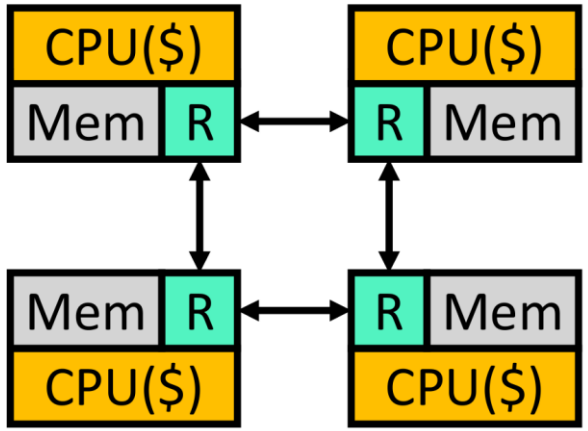
```
VLR 64
ld     [a], f0
lv     [rx], v1
multv v1, f0, v2
lv     [ry], v3
addv v2, v3, v4
sv     v4, [ry]
```

6 instructions total as compared to 600 MIPS instructions!

多核架构的组织形式

- **Bus-based Multi-core 和 Network-on-chip：Chip Multiprocessor (CMP)**

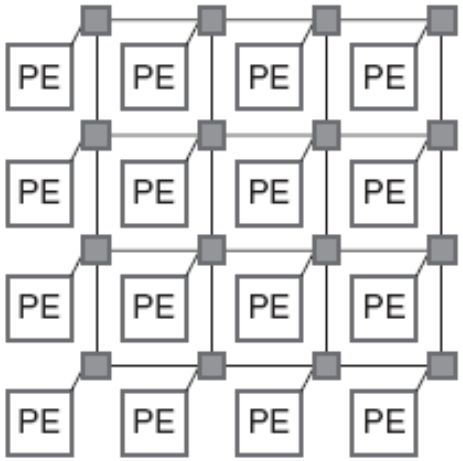


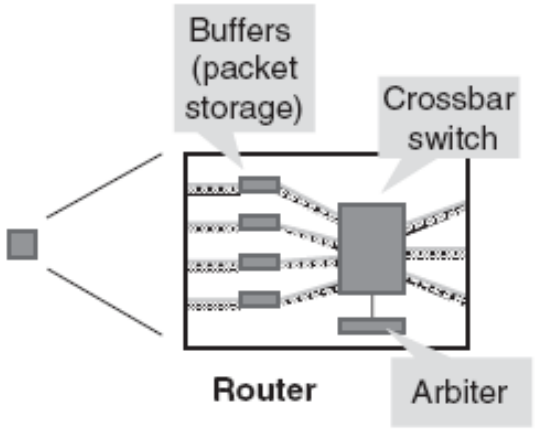


- 共享网络: e.g., bus
  - 低延迟
  - 低带宽：无法扩展到超过 16 个处理器
  - 共享属性简化了缓存一致性协议（后面会提到）
- 点对点网络: e.g., mesh or ring
  - 更长的延迟：可能需要多个“hops”才能进行通信
  - 更高的带宽：可扩展至数千个处理器
  - 缓存一致性协议很复杂

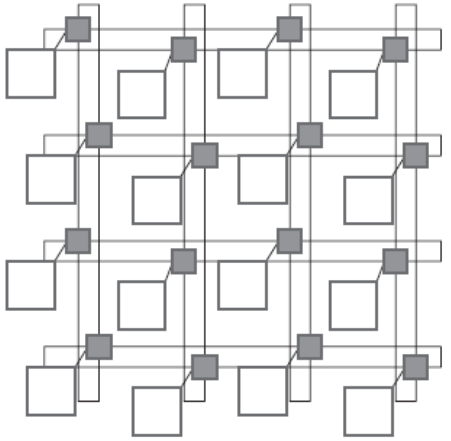
多核架构的组织形式

- Network-on-chip：Chip Multiprocessor (CMP)

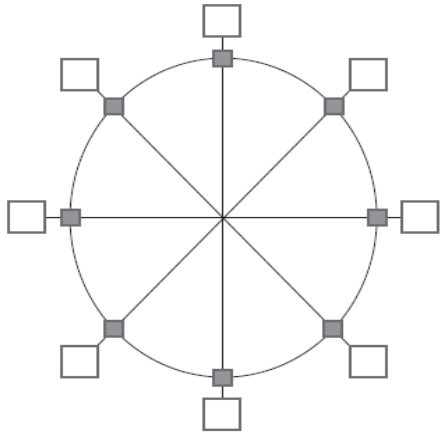




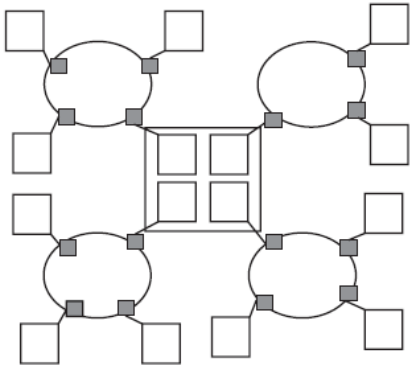
Mesh Topology



2-D torus topology



Octagon topology



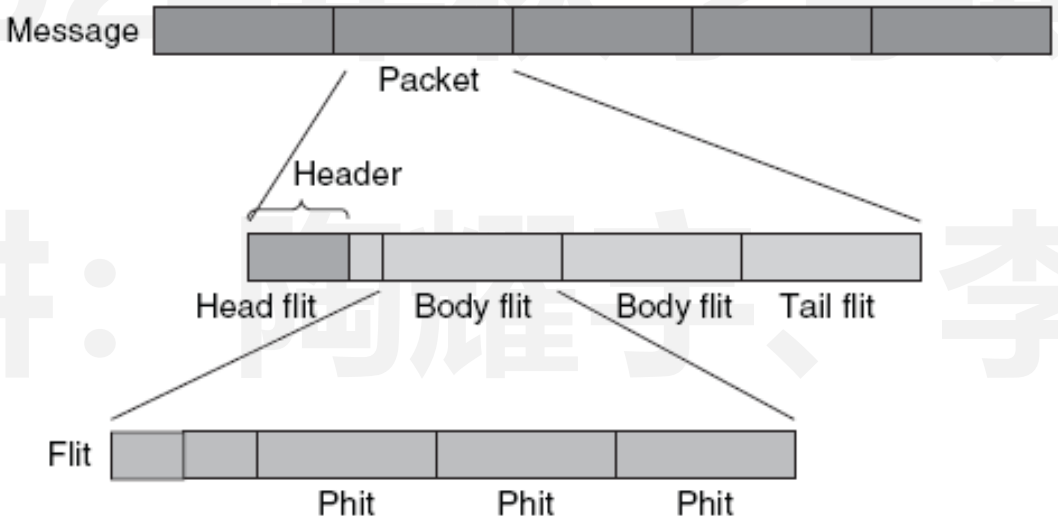
Irregular or ad hoc network topologies

多核架构的组织形式

- **Network-on-chip：Chip Multiprocessor (CMP)**

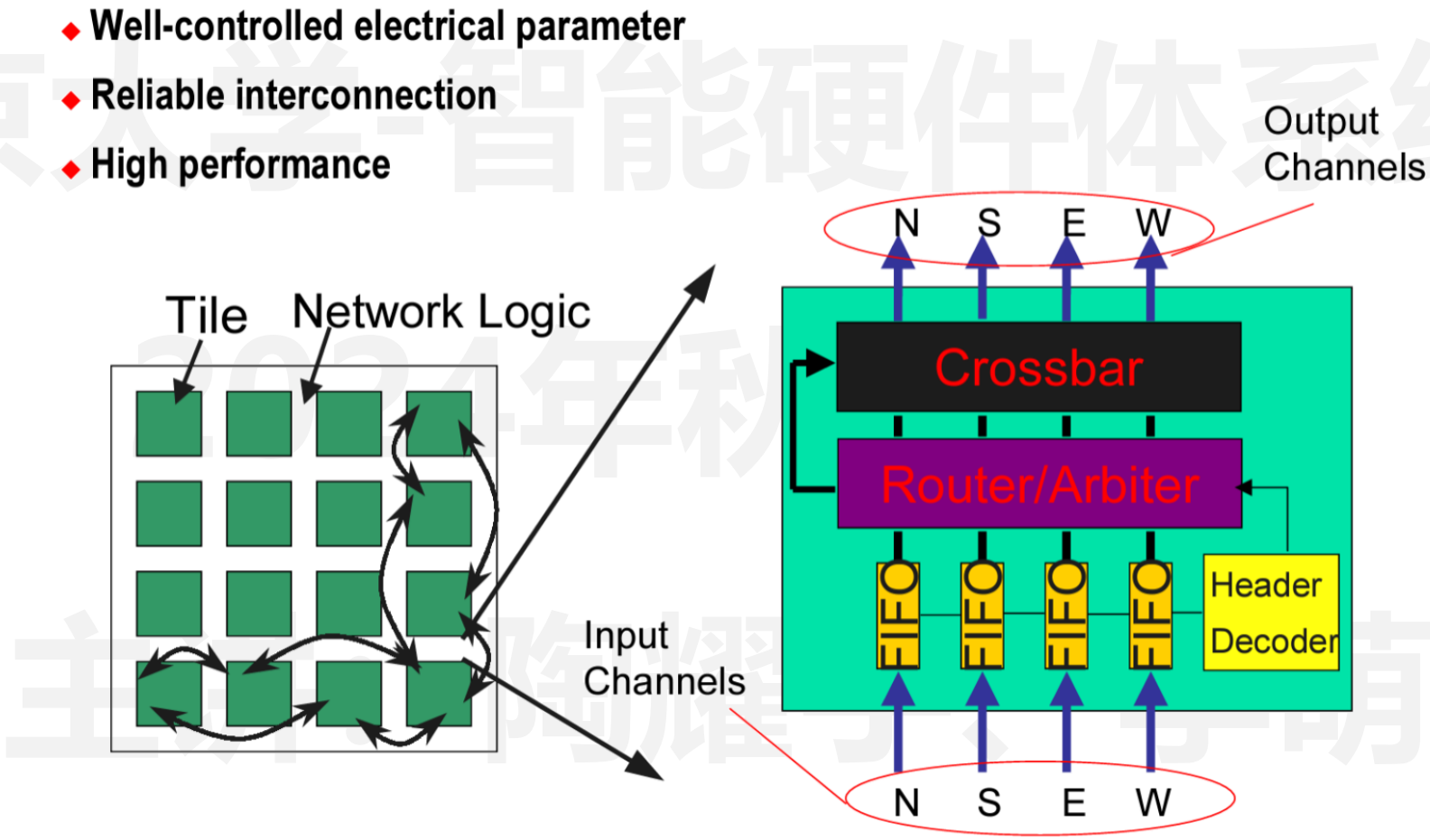
Switching strategies

- 确定数据如何通过网络中的routers
- 定义数据传输的粒度和应用的交换技术
  - phit 是在单个周期内通过链路传输的数据单位
  - typically, phit size = flit size



多核架构的组织形式

- Network-on-chip：Chip Multiprocessor (CMP)



多核架构的组织形式

北京大学 PEKING UNIVERSITY

- **Network-on-chip：Chip Multiprocessor (CMP)**

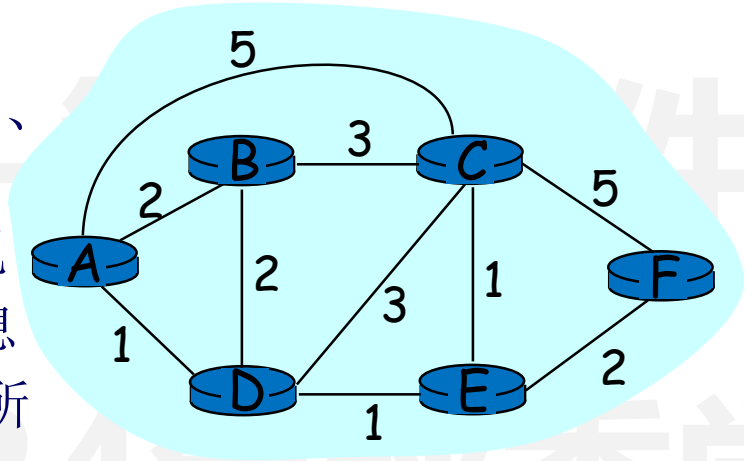
- Static and dynamic routing
  - **static routing**: 固定路径用于在特定源和目标之间传输数据
    - 没有考虑网络的当前状态
  - static routing优点:
    - 易于实现，因为几乎不需要额外的路由器逻辑
    - 如果使用单路径，则按顺序传送数据包
  - **dynamic routing**: 根据网络的当前状态做出路由决策
    - 考虑可用性和链接负载等因素
  - 源和目标之间的路径可能会随时间而改变
    - 随着traffic状况和应用要求的变化
  - 需要更多资源来监控网络状态并动态改变路由路径
  - 能够更好地分配网络中的traffic

多核架构的组织形式

- **Static Routing Tables**

Dijkstra's algorithm

- 所有节点都知道的 网络拓扑、链路成本
  - 通过"链路状态广播"实现
  - 所有节点都有相同的信息
- 计算从一个节点（"源"）到所有其他节点的最低成本路径
  - 给出对于该节点的路由表
- 迭代：经过 k 次迭代，可以知道到达目标的最小成本路径



CENTRAL ROUTING DIRECTORY

From Node

To Node	1	2	3	4	5	6
1	—	1	5	2	4	5
2	2	—	5	2	4	5
3	4	3	—	5	3	5
4	4	4	5	—	4	5
5	4	4	5	5	—	5
6	4	4	5	5	6	—

Node 1 Directory

Destination	Next Node
2	2
3	4
4	4
5	4
6	4

Node 2 Directory

Destination	Next Node
1	1
3	3
4	4
5	4
6	4

Node 3 Directory

Destination	Next Node
1	5
2	5
4	5
5	5
6	5

Node 4 Directory

Destination	Next Node
1	2
2	2
3	5
5	5
6	5

Node 5 Directory

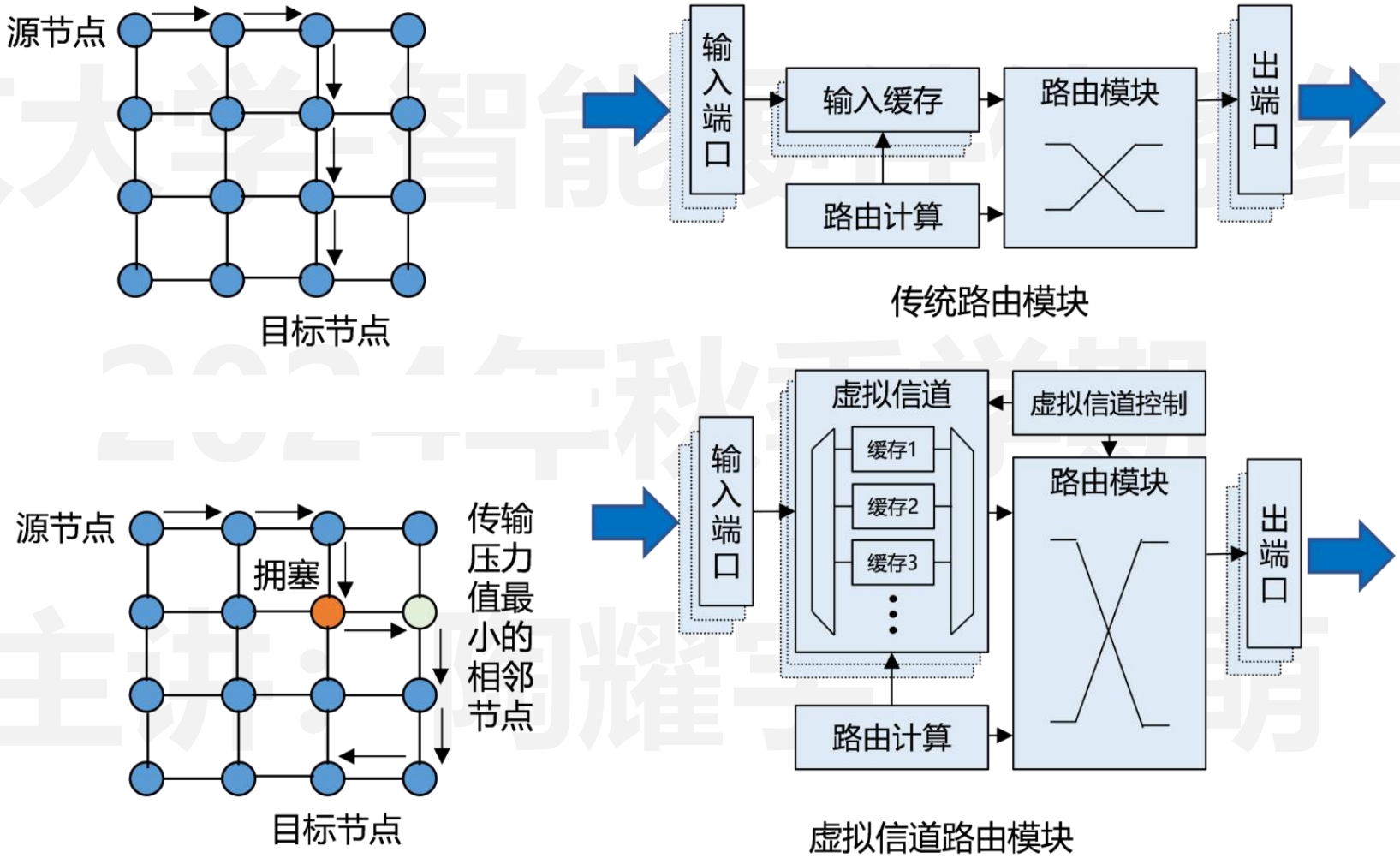
Destination	Next Node
1	4
2	4
3	3
4	4
6	6

Node 6 Directory

Destination	Next Node
1	5
2	5
3	5
4	5
5	5

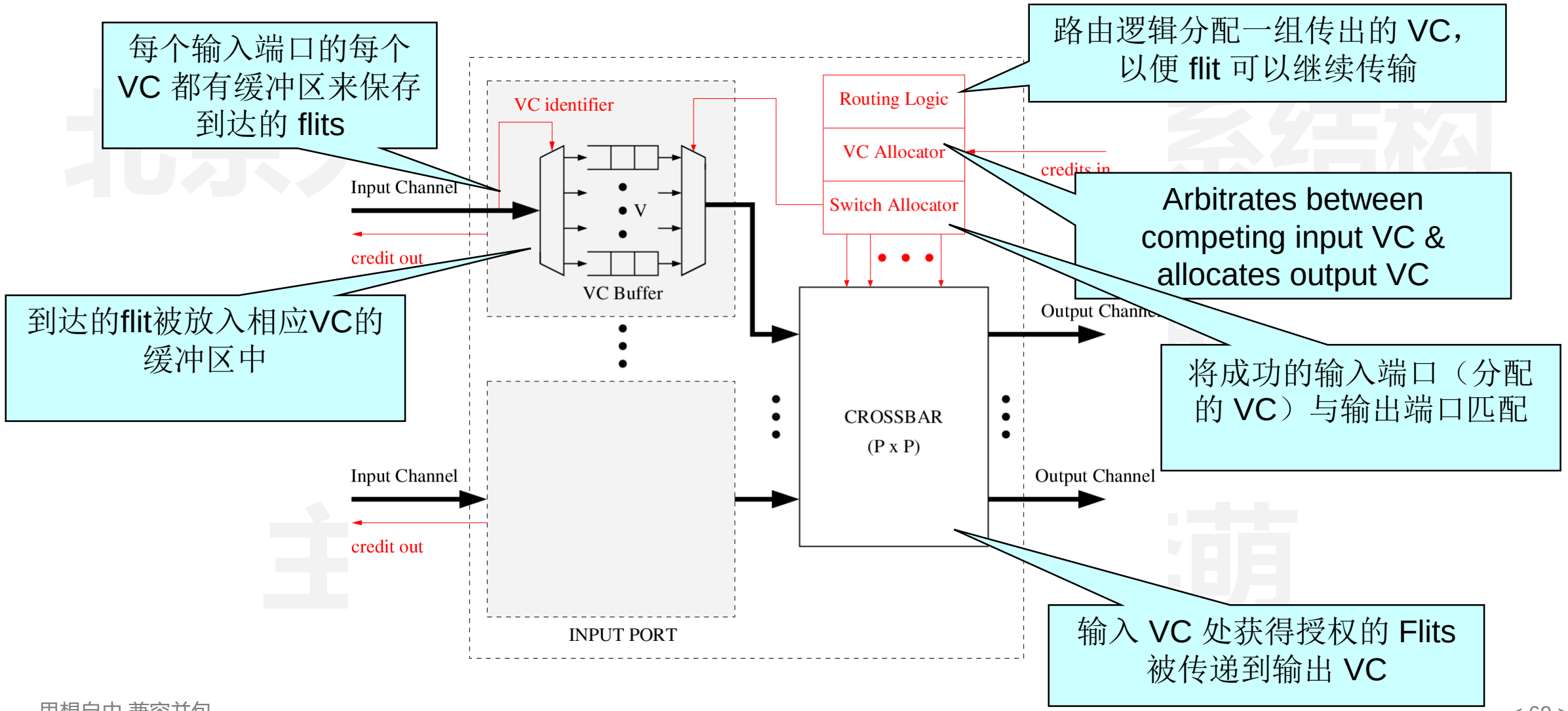
多核架构的组织形式

- Dynamic Routing



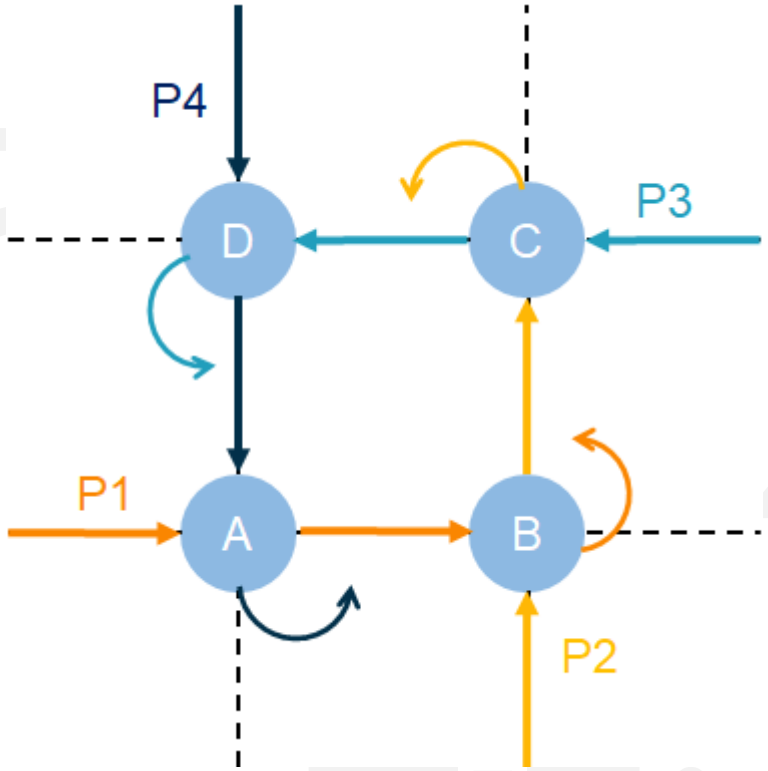
多核架构的组织形式

- 虚拟信道路由



多核架构的组织形式

- **路由死锁Deadlock**

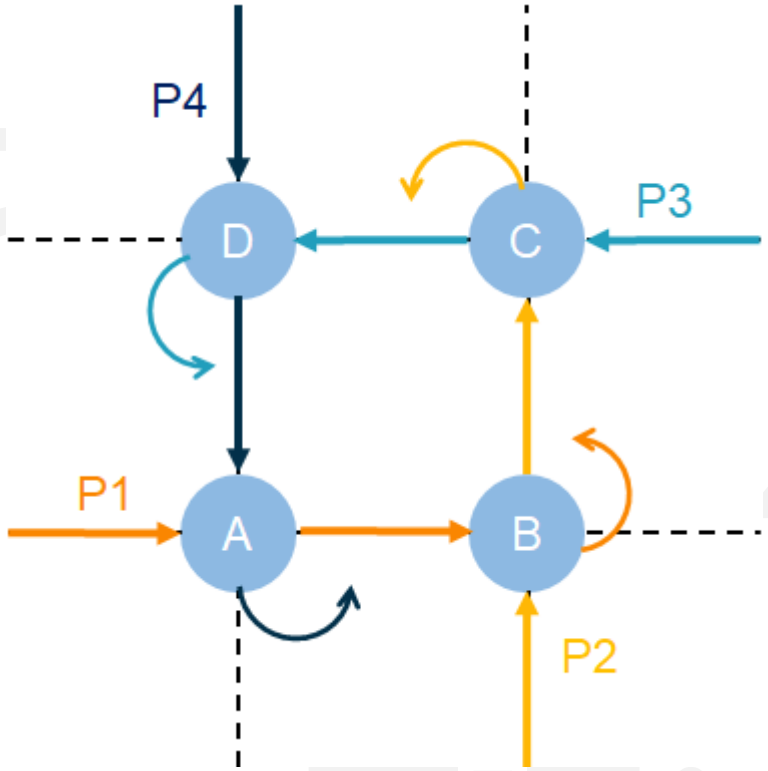


假设路由规则要求全路径资源空闲才可发送数据包

- 死锁是指多个数据包或任务在网络中争夺资源导致的路由间的永久阻塞，从而导致数据或消息任务在没有外界干扰的情况下永远无法转发到目的地。
- P1、P2、P3和P4四个数据包依次占用了一条路径资源，并请求额外的资源，但请求的资源又被相邻数据包占据
- 以P1和P2数据包为例，P1 数据包从路由器A发往B并最终期望达到路由器C
- P2 数据包从路由器B发往C并最终期望达到路由器D。
- P1、P2分别占用AB、BC链路，P1请求的BC链路目前没有得到释放，原因在于P2请求的CD链路没有得到释放，导致P1、P2数据包无法进行后续转发。因循环的数据路径以及相互的资源占用导致四个数据包产生死锁

多核架构的组织形式

- 解决路由死锁Deadlock的方法



假设路由规则要求全路径资源空闲才可发送数据包

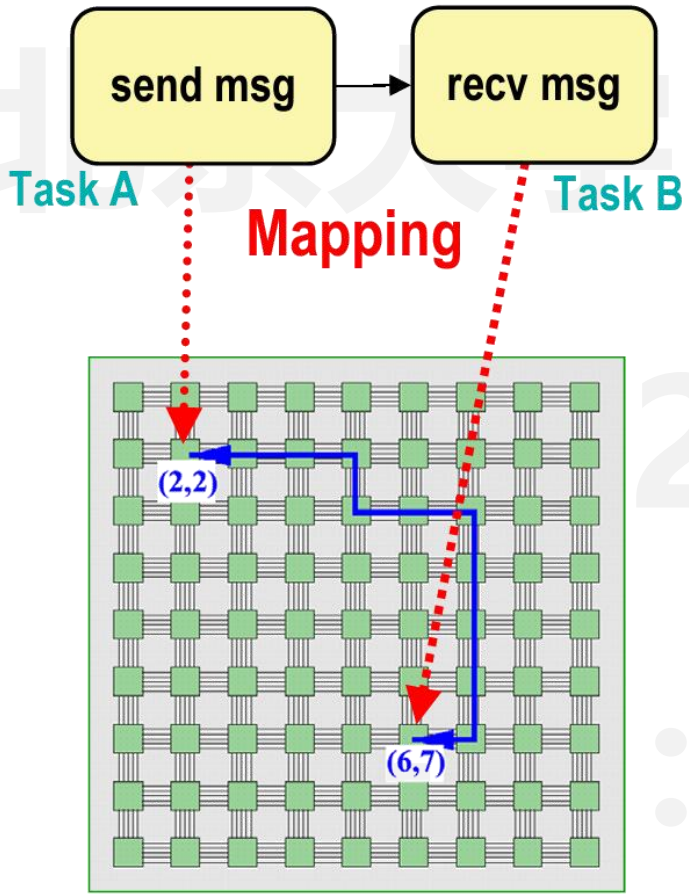
1.提供更多资源：该方法对应解决死锁条件中的互斥，主要采用两种策略：增加虚拟通道和消费通道。前者通过在路由端口增加额外缓存，避免单一通道资源占用导致的死锁，但是并不增加额外的物理传输通道；后者与前者相反，通过增加物理传输通道，从而在转发过程中提供更多资源

2.避免数据环路：该方法对应解决死锁条件中的数据环路，主要通过设计合理的路由算法，来避免环路的发生

3.取消任务对资源的占用：该方法对应解决死锁条件中的资源占用，即取消阻塞数据包对通道的占用请求。一般考虑一个确定的等待时间，如果数据包在路由缓存中长期未转发，则网络考虑出现死锁，取消其占用请求，并利用额外的死锁缓存通道来进行缓存，使得路由资源重新得到释放

多核架构的组织形式

- Workload Mapping



- **映射问题：**
  - 将任务分配给资源
  - 将任务地址转换为资源/任务地址
  - 建立和关闭channels
  - 静态分配与动态分配
- **系统软件：NoC OS**
  - OS、RTOS、run-time调度器
  - 取决于组件和网络
  - 应用程序

多核架构的组织形式

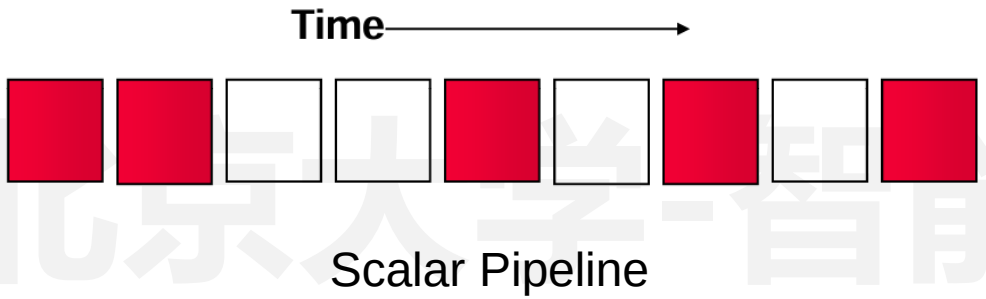
北京大学 PEKING UNIVERSITY

- NoC设计需要注意的问题

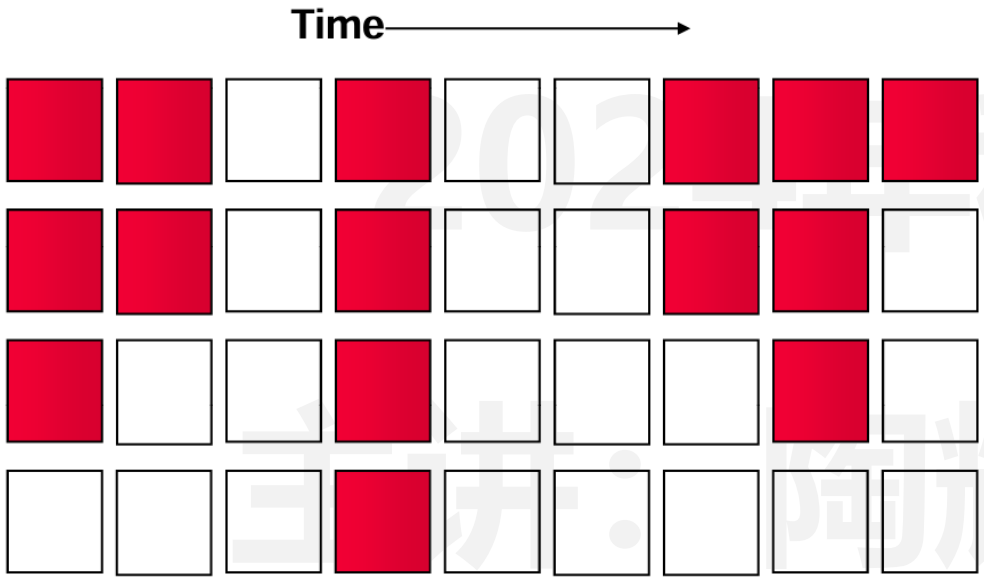
- **扩展问题**
  - 需要多大的NoC？应用面积要求是什么？
- **区域定义问题**
  - 需要什么样的区域？区域之间有什么样的接口？
  - 各区域的容量要求有哪些？
- **资源设计问题**
  - 资源内部需要什么？内部计算类型和内部通信？
- **应用程序映射流程问题**
  - 必须支持什么样的语言、模型和工具？
  - 如何验证和测试最终产品？

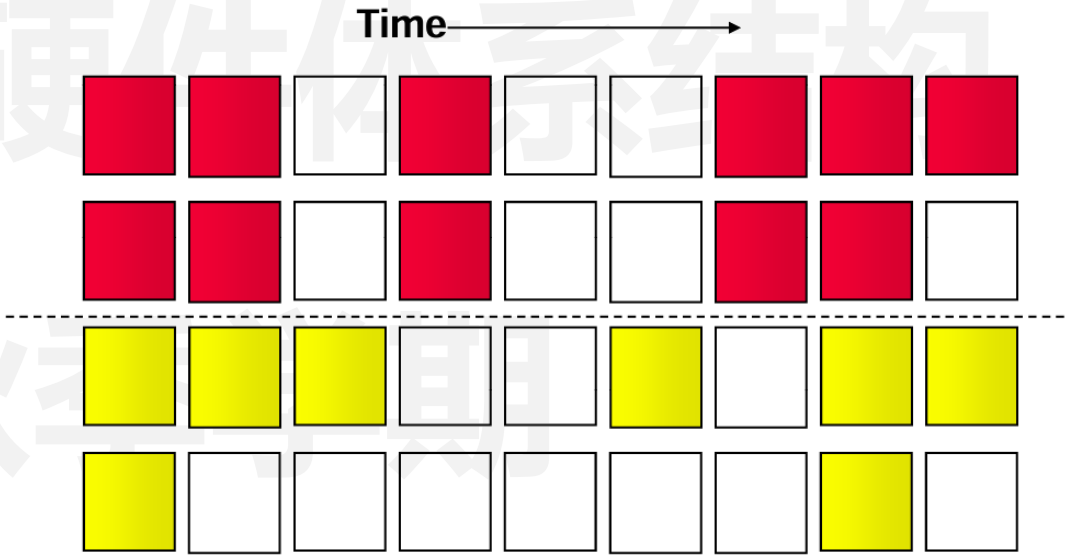
多线程组织形式

- Instruction Issue



Scalar Pipeline



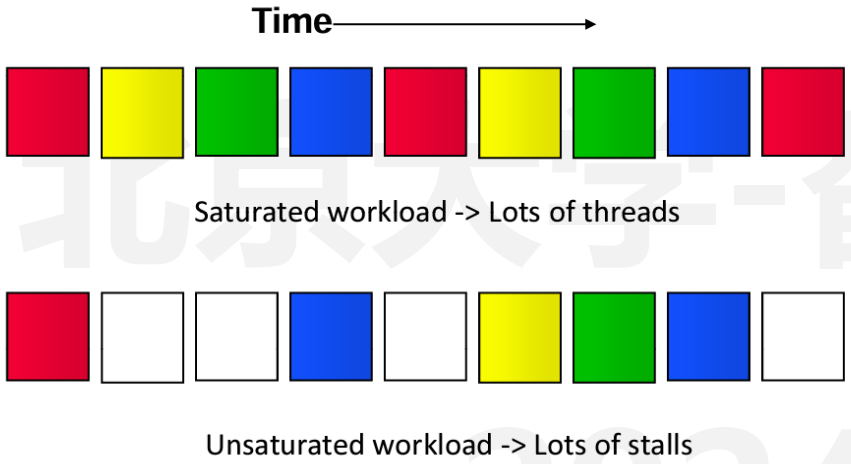


Limited utilization when only running one thread

Superscalar leads to more performance, but lower utilization

多线程组织形式

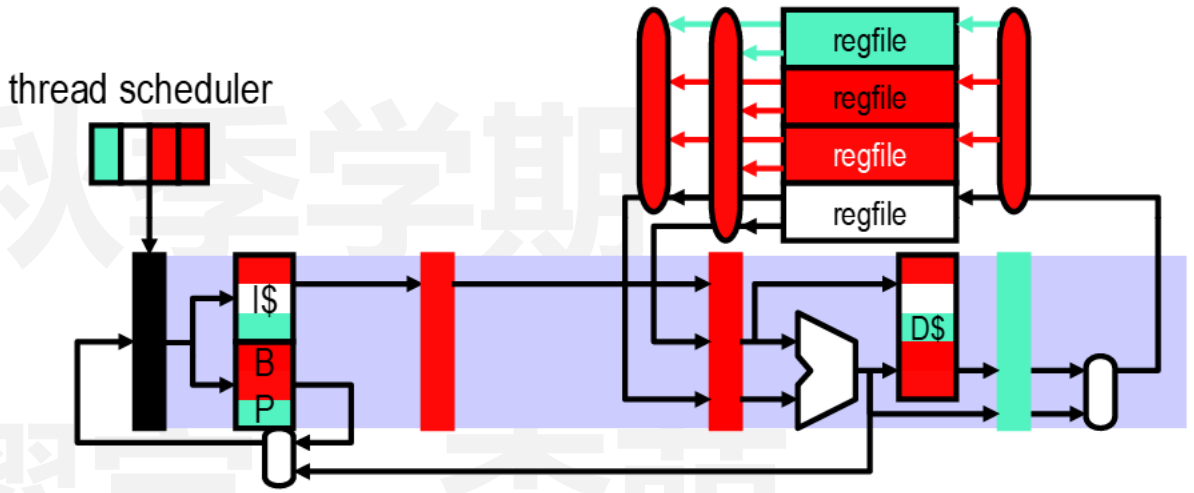
- **Fine Grained Multithreading (FGMT)**



- FGMT
  - (Many) more threads
  - Multiple threads in pipeline at once

Intra-thread dependencies still limit performance

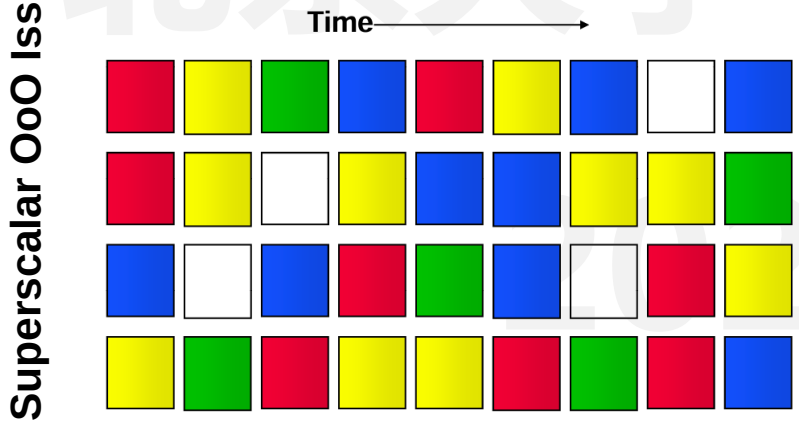
- 每个Thread（线程）个线程都可以单独执行一个运行任务
- 线程包含于进程当中，进程是线程的集合
- 当程序运行时，至少有一个进程会启动，而这个进程中往往包含了多个线程



Many threads → many register files

多线程组织形式

- **Simultaneous Multithreading (SMT)**

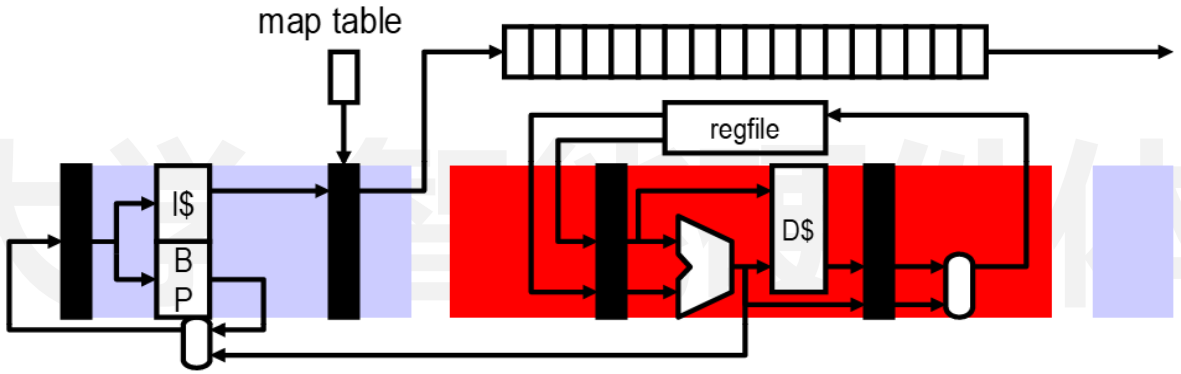


Maximum utilization of function units by independent operations

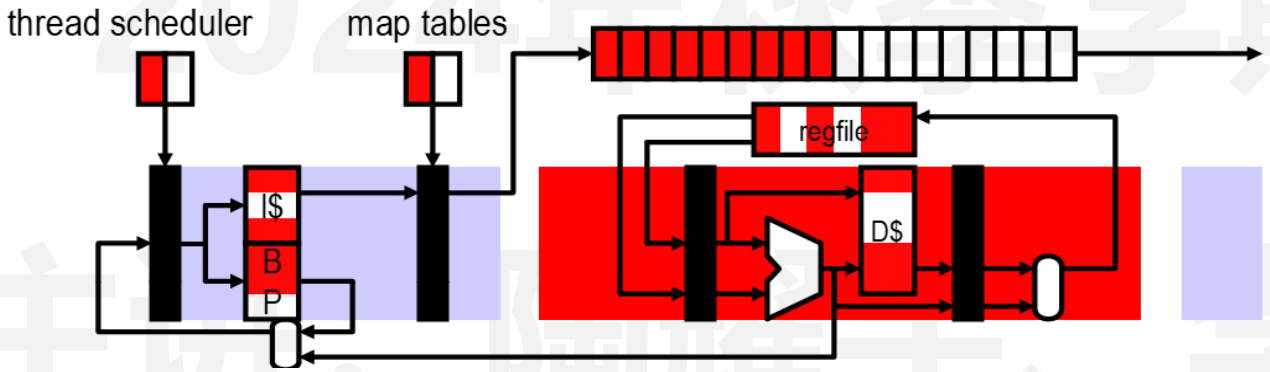
- Can we multithread an out-of-order machine?
  - Don't want to give up performance benefits
  - Don't want to give up natural tolerance of D$ (L1) miss latency
- **Simultaneous multithreading (SMT)**
  - Tolerates all latencies (e.g., L2 misses, mispredicted branches)
  - Sacrifices some single thread performance
  - Thread scheduling policy
    - Round-robin (just like FGMT)
  - Pipeline partitioning
    - Dynamic, hmmm…
  - Example: Pentium4 (hyper-threading): 5-way issue, 2 threads
  - Another example: Alpha 21464: 8-way issue, 4 threads (canceled)

多线程组织形式

- **Simultaneous Multithreading (SMT)**



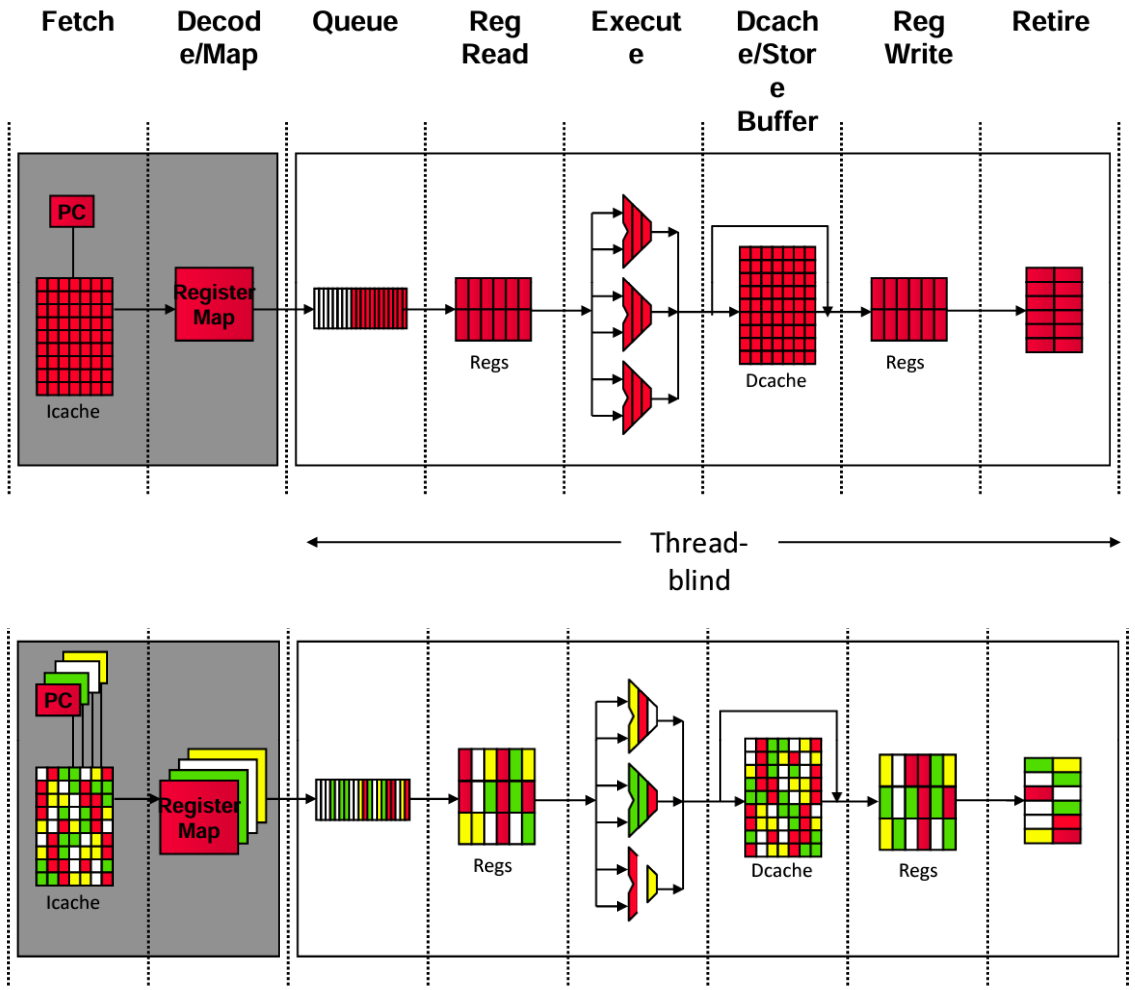
- SMT
  - Replicate map table, share physical register file



- Large map table and physical register file
  - #mt-entries = (**#threads** * #arch-regs)
  - #phys-regs = (**#threads** * #arch-regs) + #in-flight insns

多线程组织形式

- Simultaneous Multithreading (SMT) Pipeline



SMT Changes

Basic pipeline – unchanged

Replicated resources
- Program counters
- Register maps

Shared resources
- Register file (size increased)
- Instruction queue
- First and second level caches
- Translation buffers
- Branch predictor

多线程组织形式

北京大学 PEKING UNIVERSITY

- **Simultaneous Multithreading (SMT) vs Chip Multiprocessor (CMP)**

  - 如果你想多线程运行你会构建…..
    - 多处理器芯片（CMP）：多个分离流水线?
    - 一个多线程处理器（SMT）：单个更大流水线?

  - **Both will get you throughput on multiple threads**
    - CMP will be simpler, possibly faster clock
    - SMT will get you better performance (IPC) on a single thread
      - SMT is basically an ILP engine that converts TLP to ILP
      - CMP is mainly a TLP engine

  - **Again, do both**
    - Sun's Niagara (UltraSPARC T1)
    - 8 processors, each with 4-threads (coarse-grained threading)
    - 1Ghz clock, in-order, short pipeline (6 stages or so)
    - Designed for power-efficient "throughput computing"