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

第九讲：缓存微架构与AI芯片基础

主讲：陶耀宇

2026年春季

注意事项

北京大学 PEKING UNIVERSITY

- 课程作业情况

- **作业将在3月底-4月中旬、4月中旬-5月初、5月中旬-6月初**

  第二次作业时间：5.11-5.25

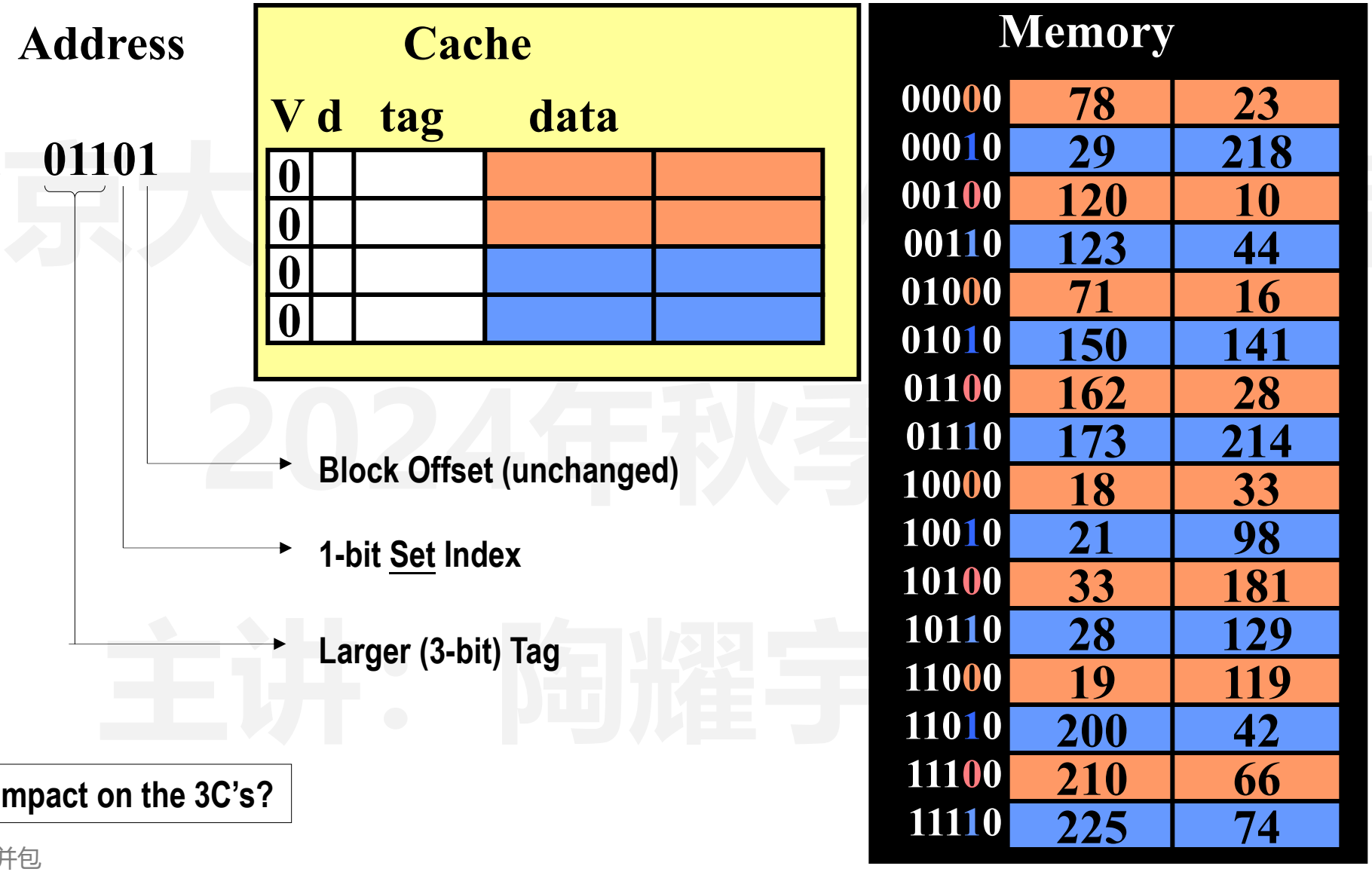
  第三次作业时间：5.28-6.13

- **第1次lab时间：4月13日-5月13日**
- **第2次lab时间：5月13日-6月13日**
- **助教安排硬件Verilog/SystemVerilog编写及设计、验证全流程入门介绍，有兴趣的同学请积极参与！**

目 录

CONTENTS

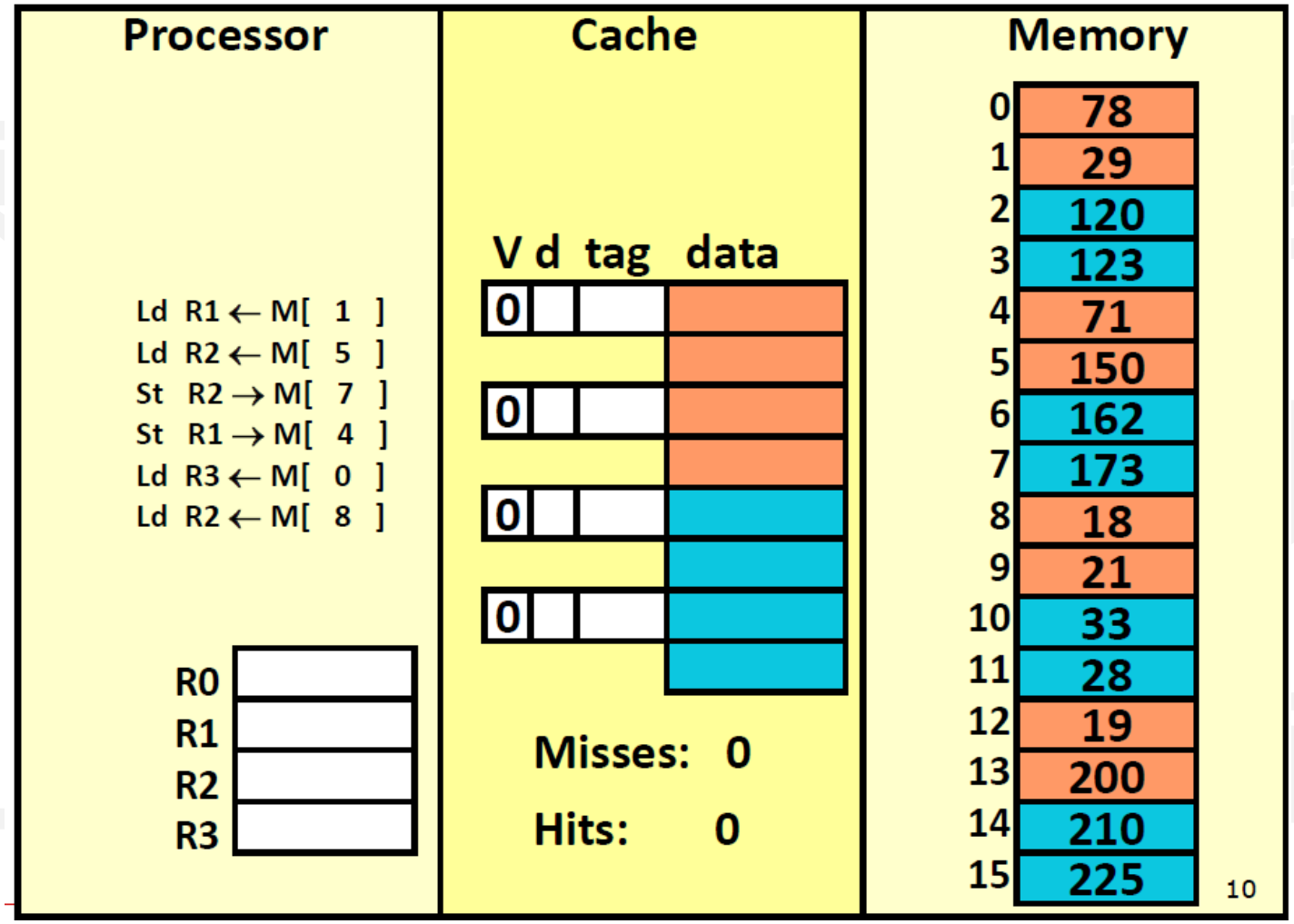
2-Way Set Associative Cache设计



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

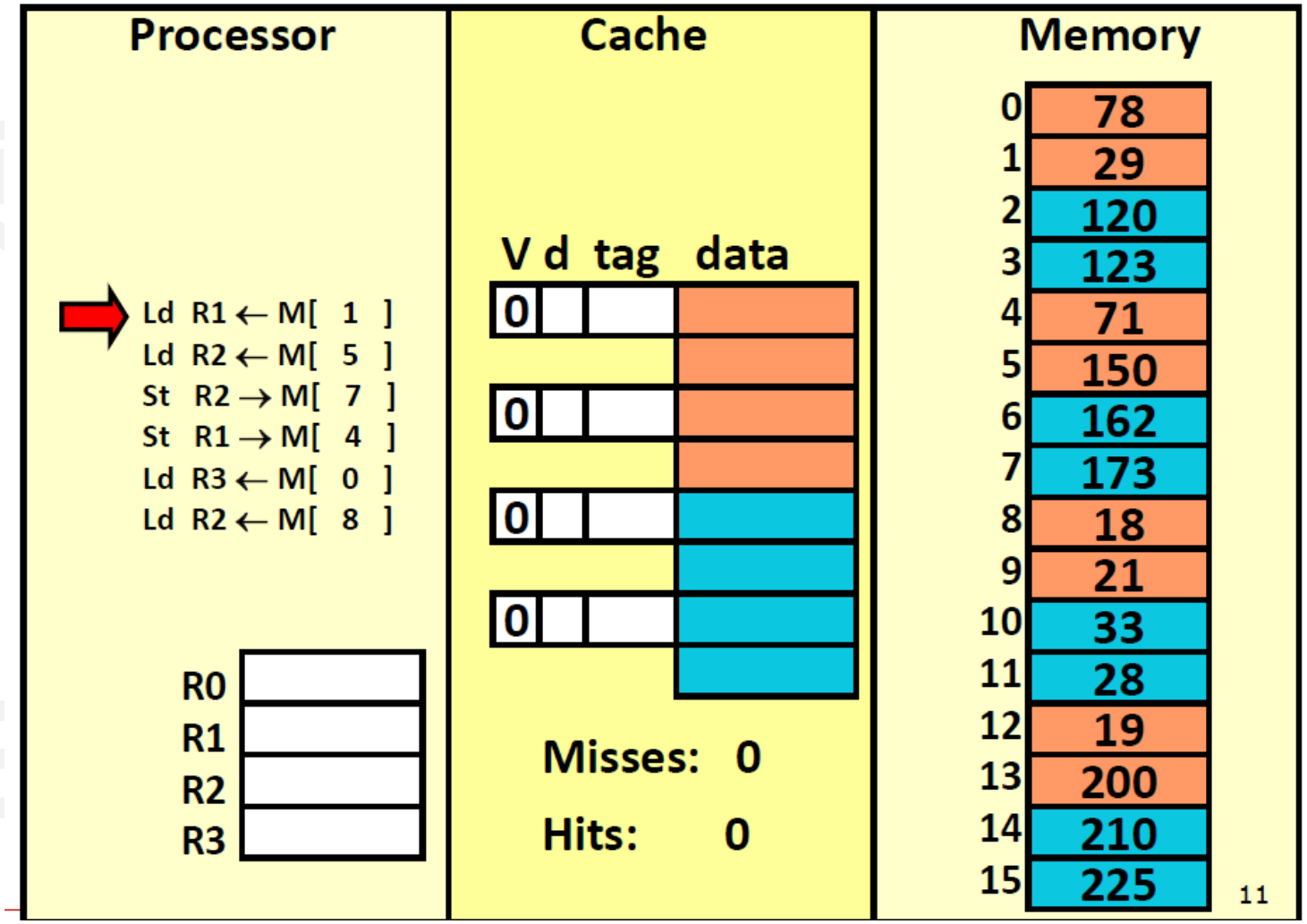
2-Way Set Associative Cache实例

- Write-back & Write-allocate



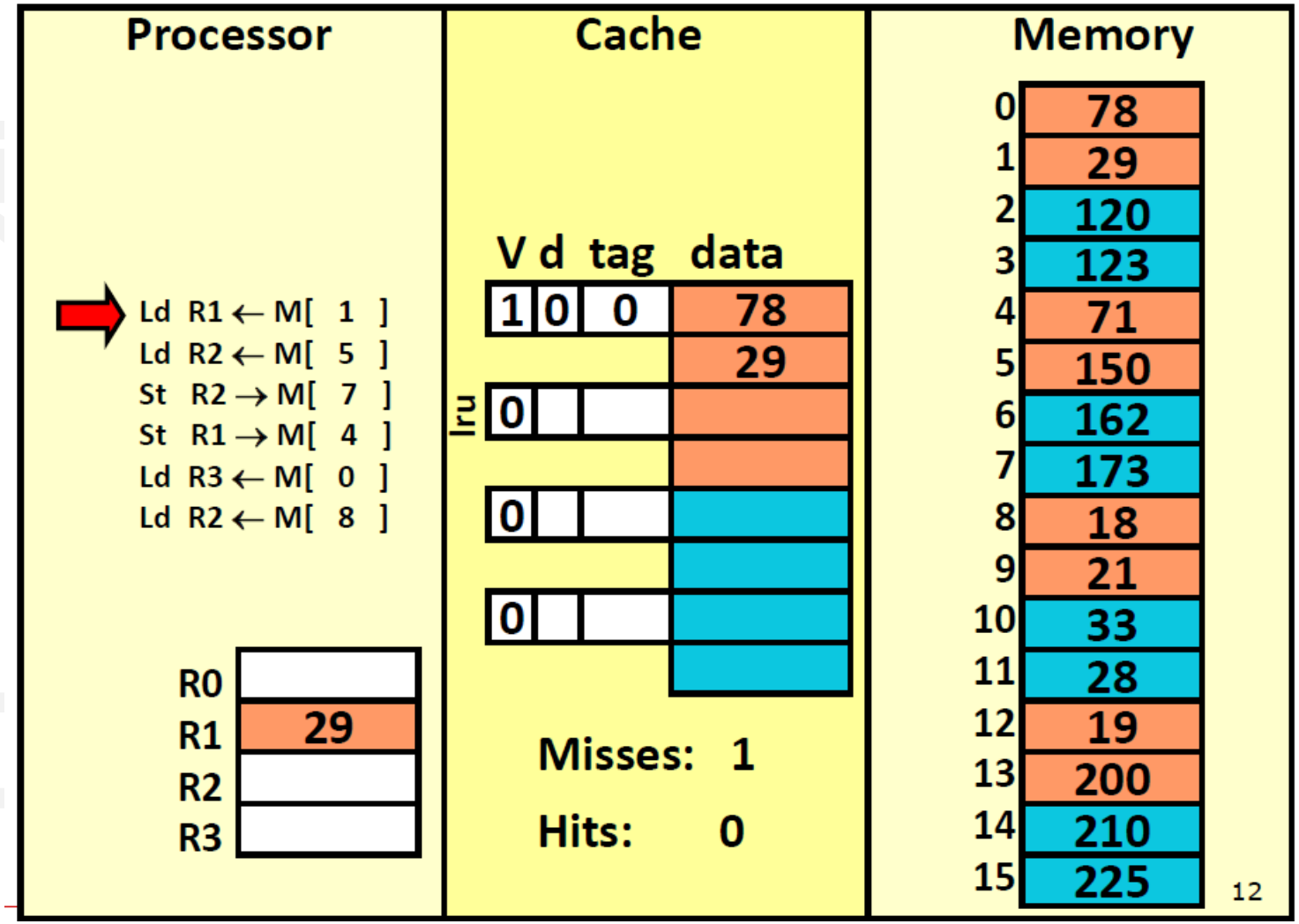
2-Way Set Associative Cache实例

- Write-back & Write-allocate



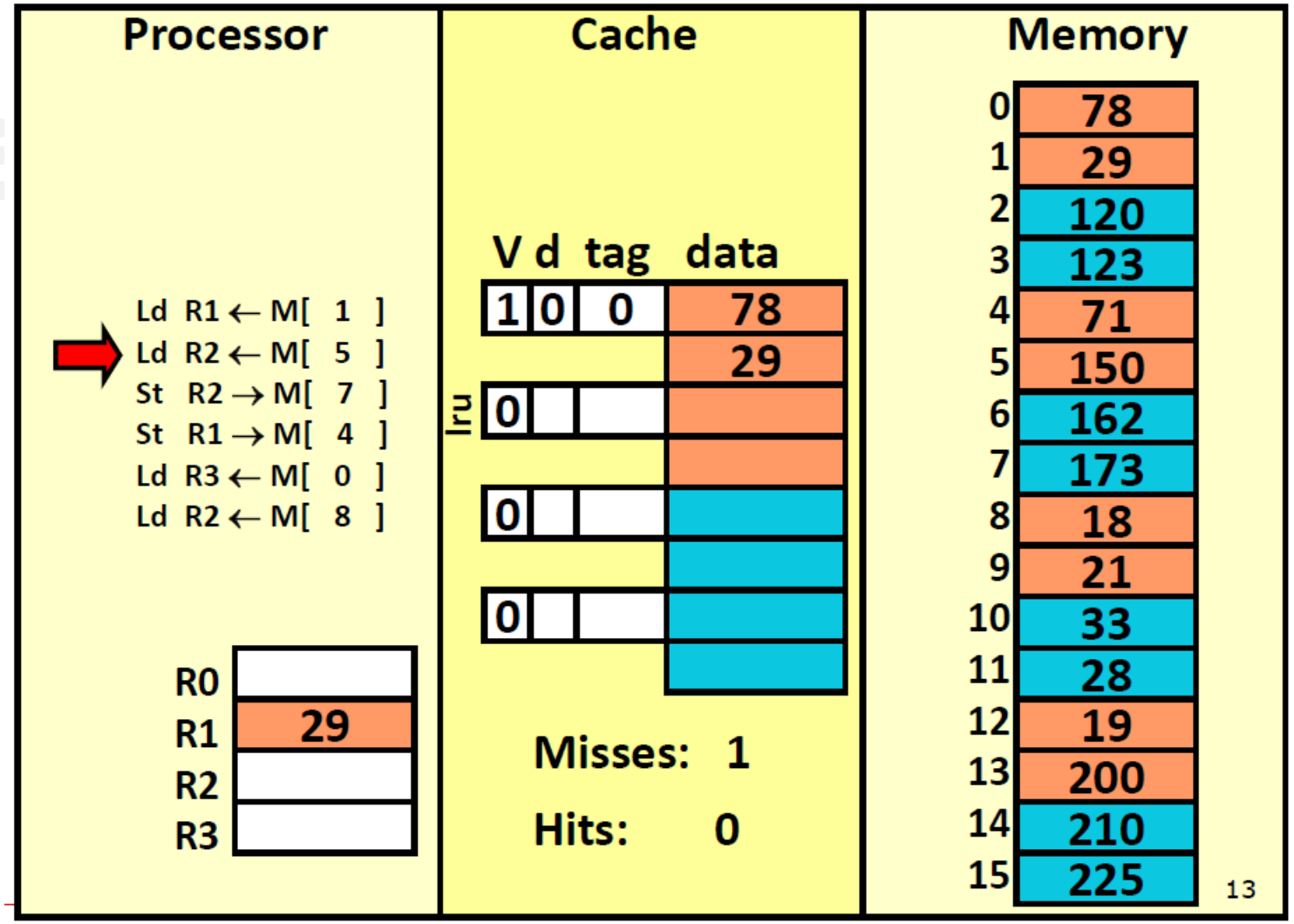
2-Way Set Associative Cache实例

- Write-back & Write-allocate



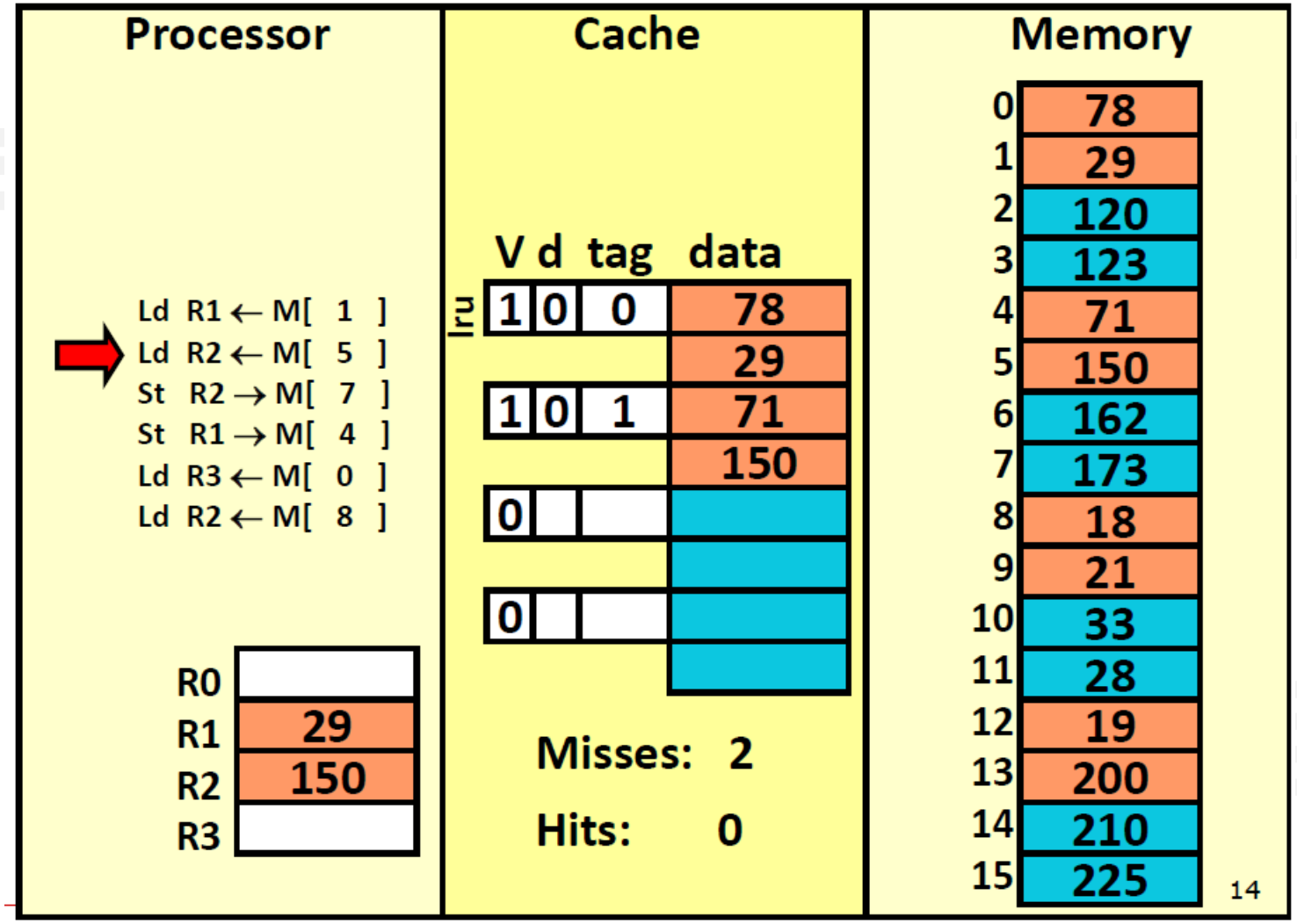
2-Way Set Associative Cache实例

- Write-back & Write-allocate



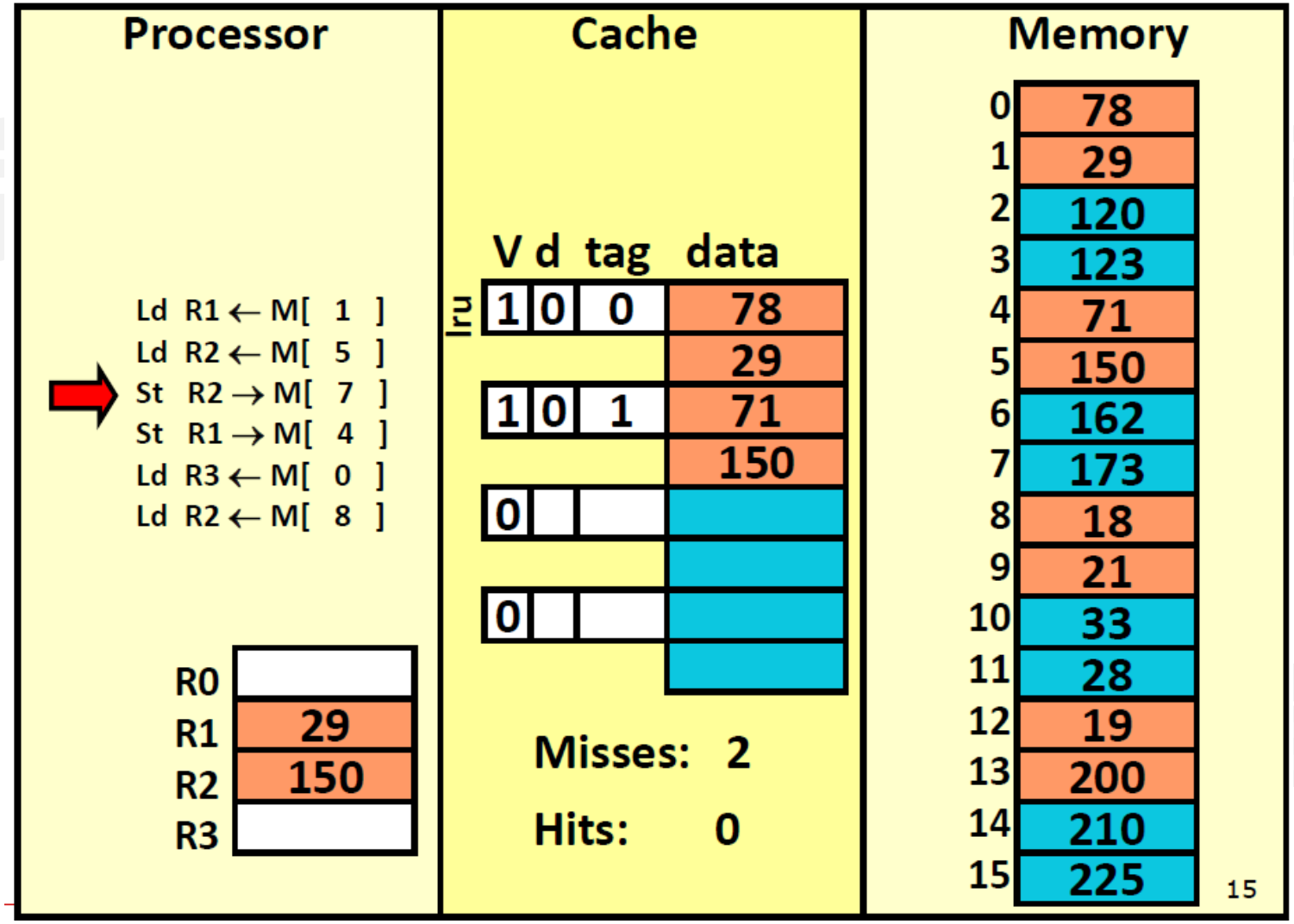
2-Way Set Associative Cache实例

- Write-back & Write-allocate



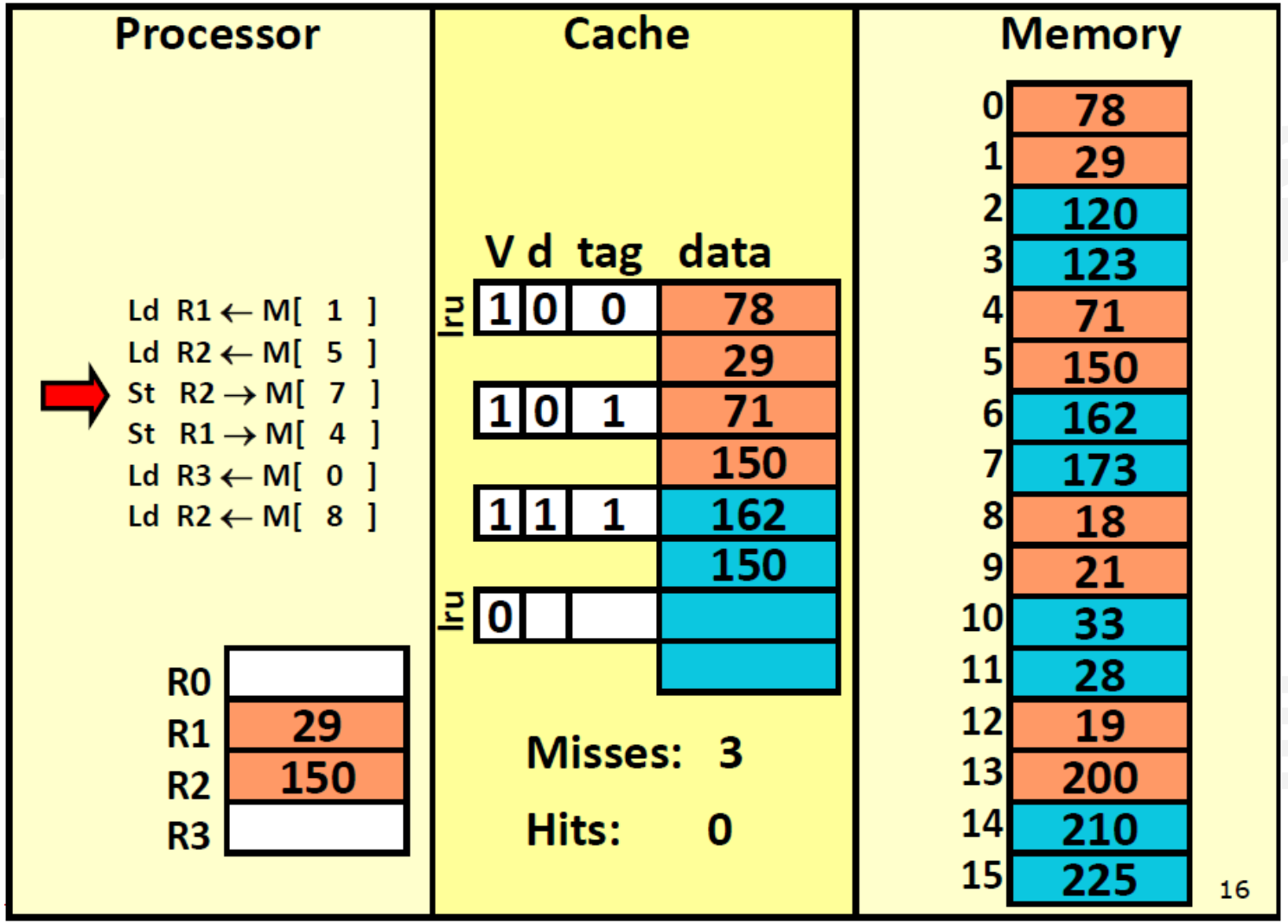
2-Way Set Associative Cache实例

- Write-back & Write-allocate



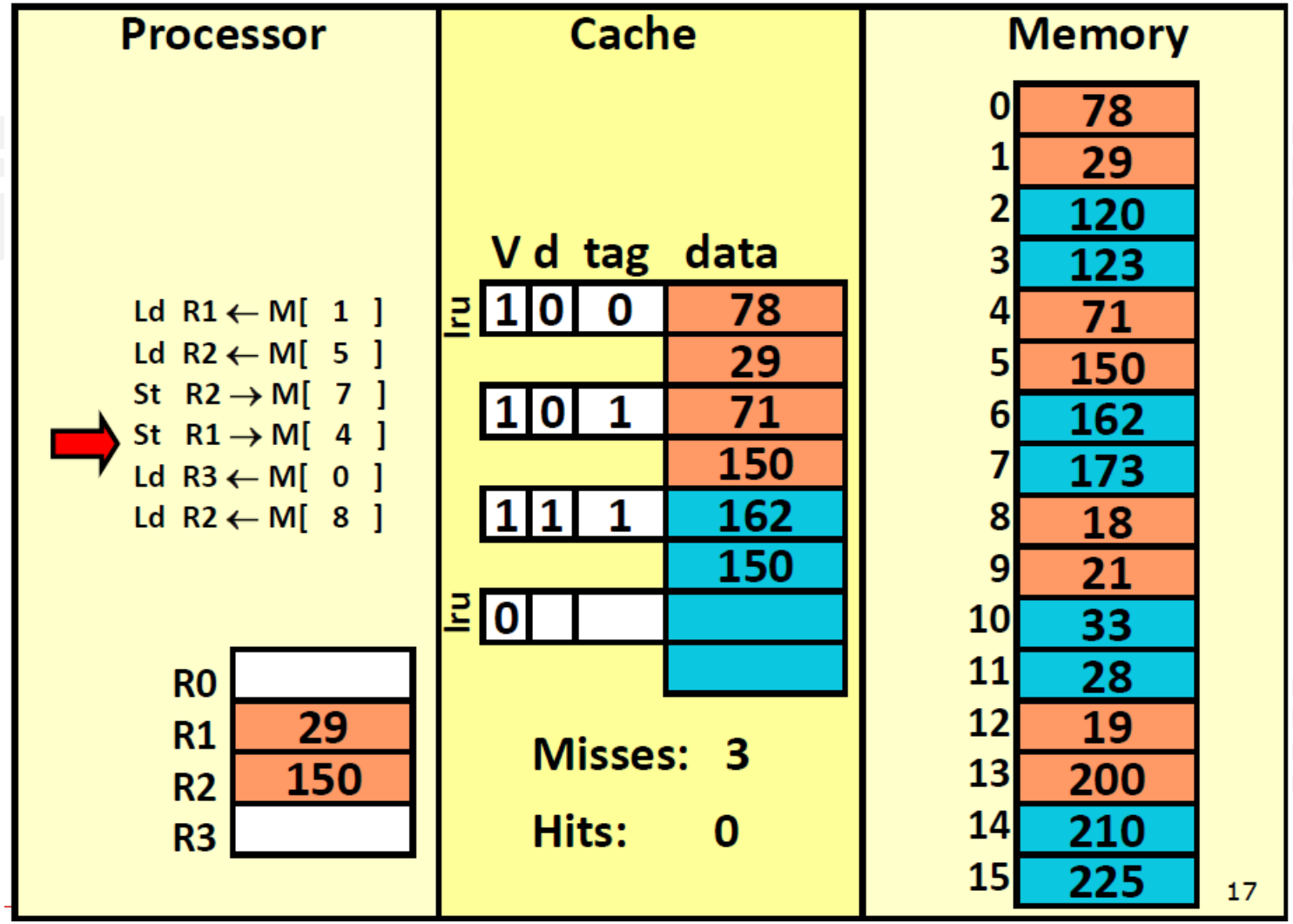
2-Way Set Associative Cache实例

- Write-back & Write-allocate



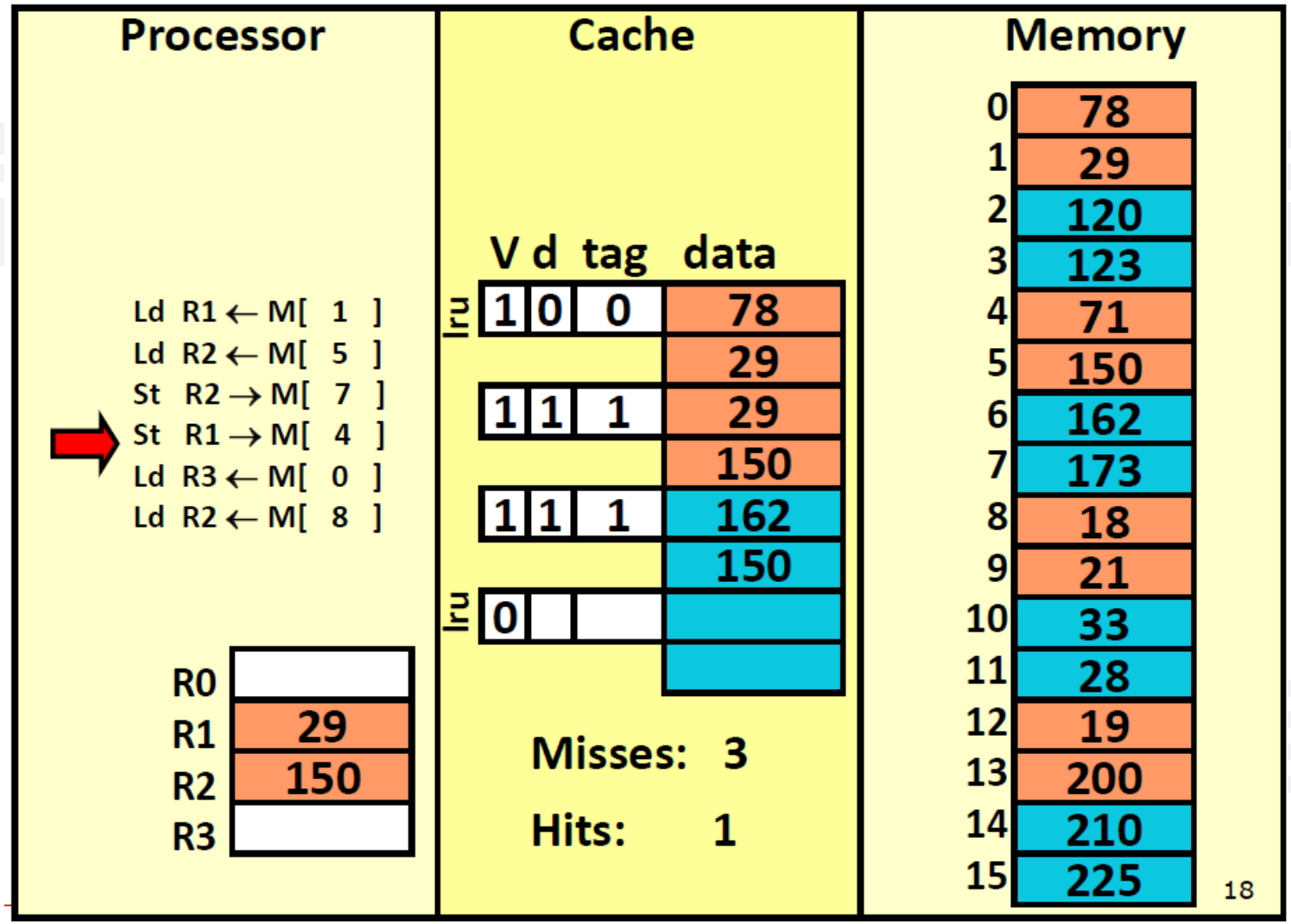
2-Way Set Associative Cache实例

- Write-back & Write-allocate



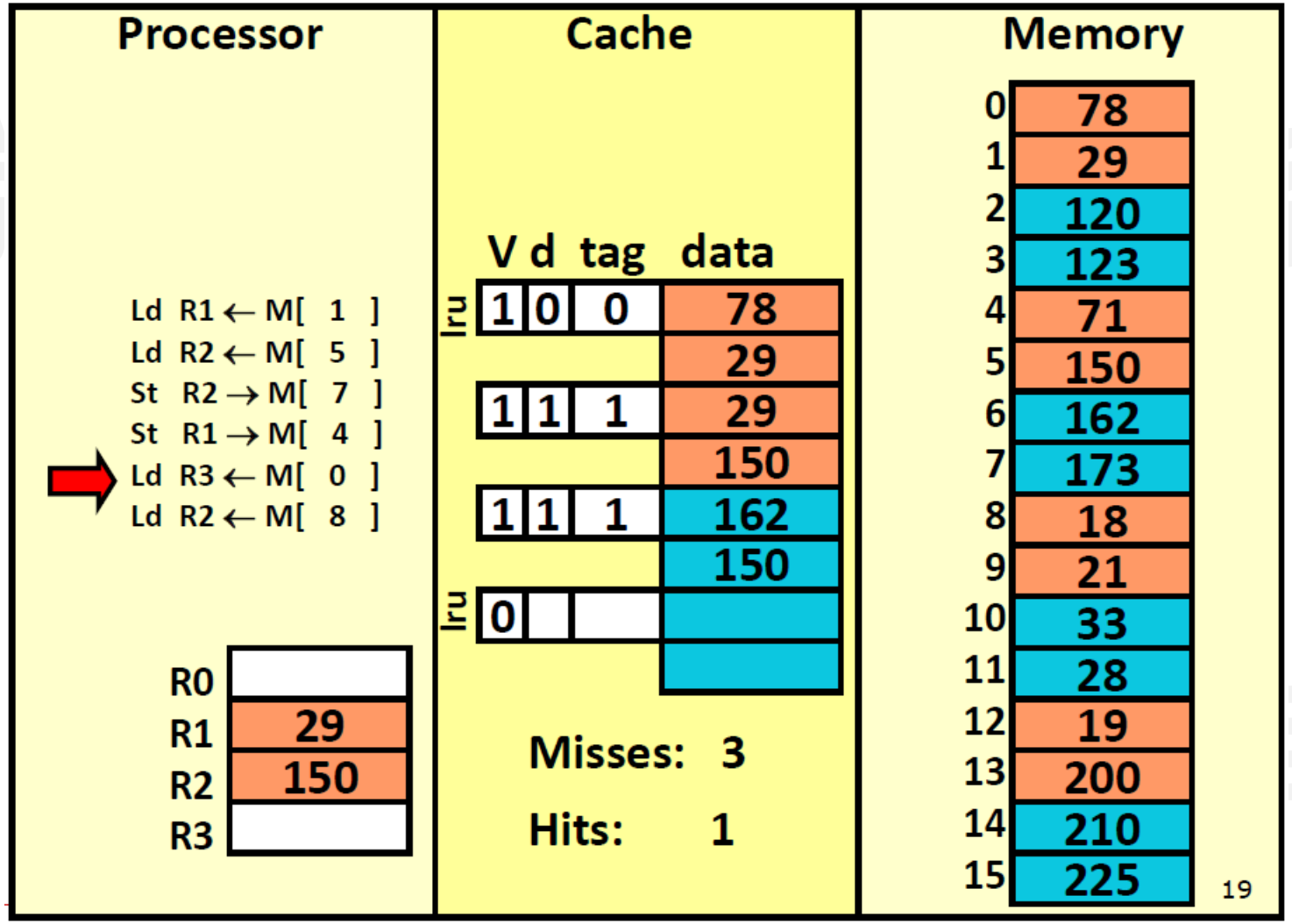
2-Way Set Associative Cache实例

- Write-back & Write-allocate



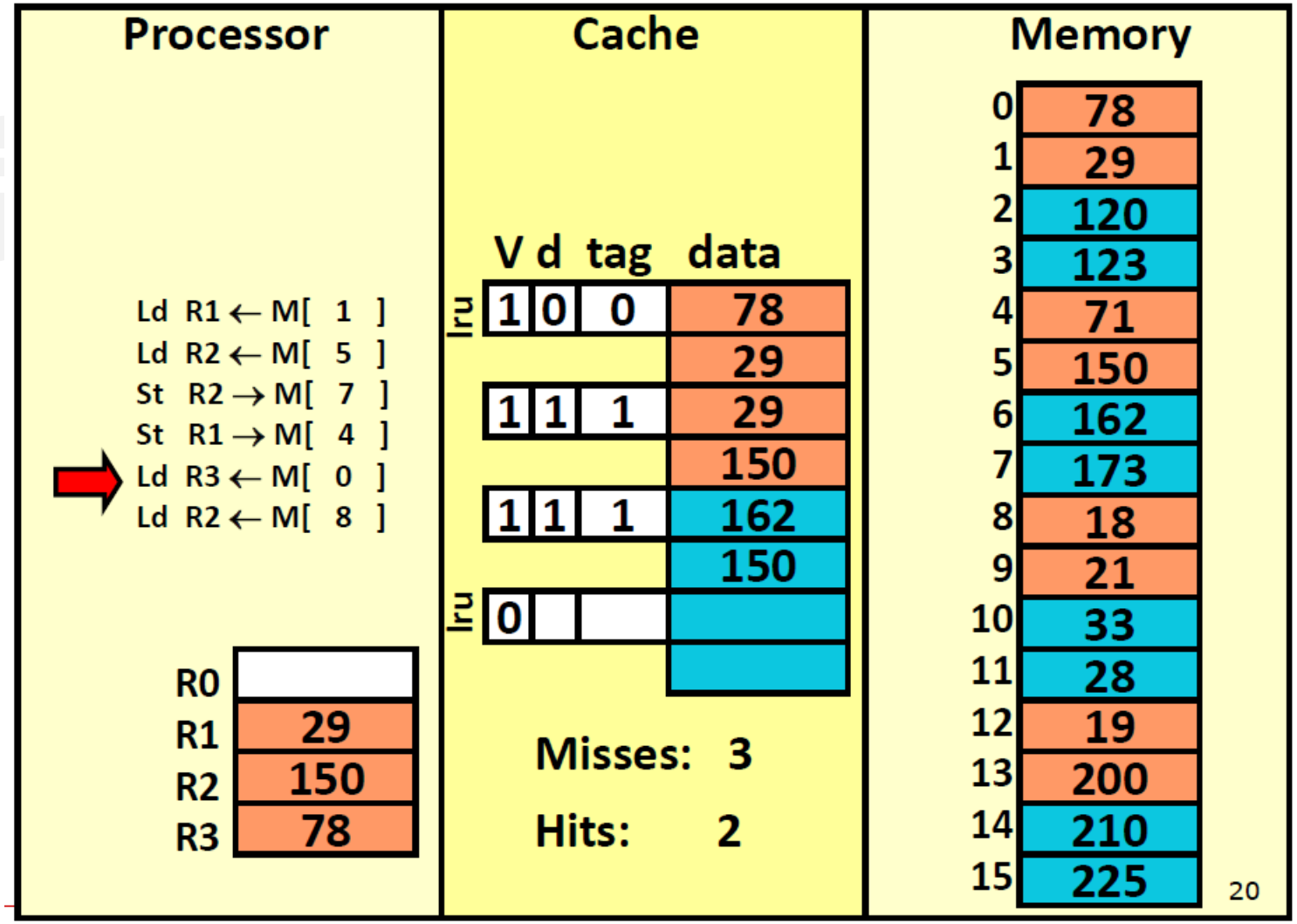
2-Way Set Associative Cache实例

- Write-back & Write-allocate



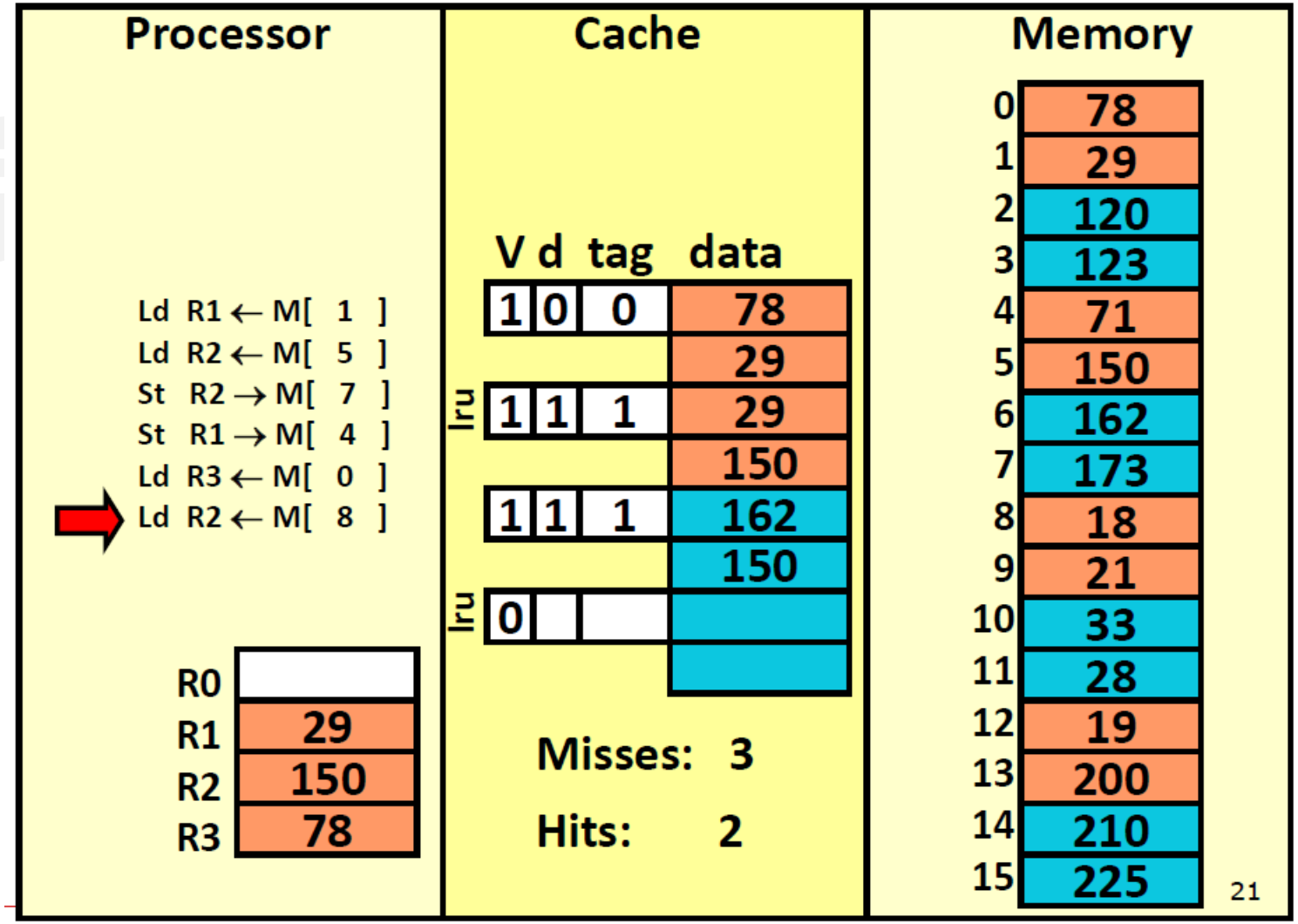
2-Way Set Associative Cache实例

- Write-back & Write-allocate



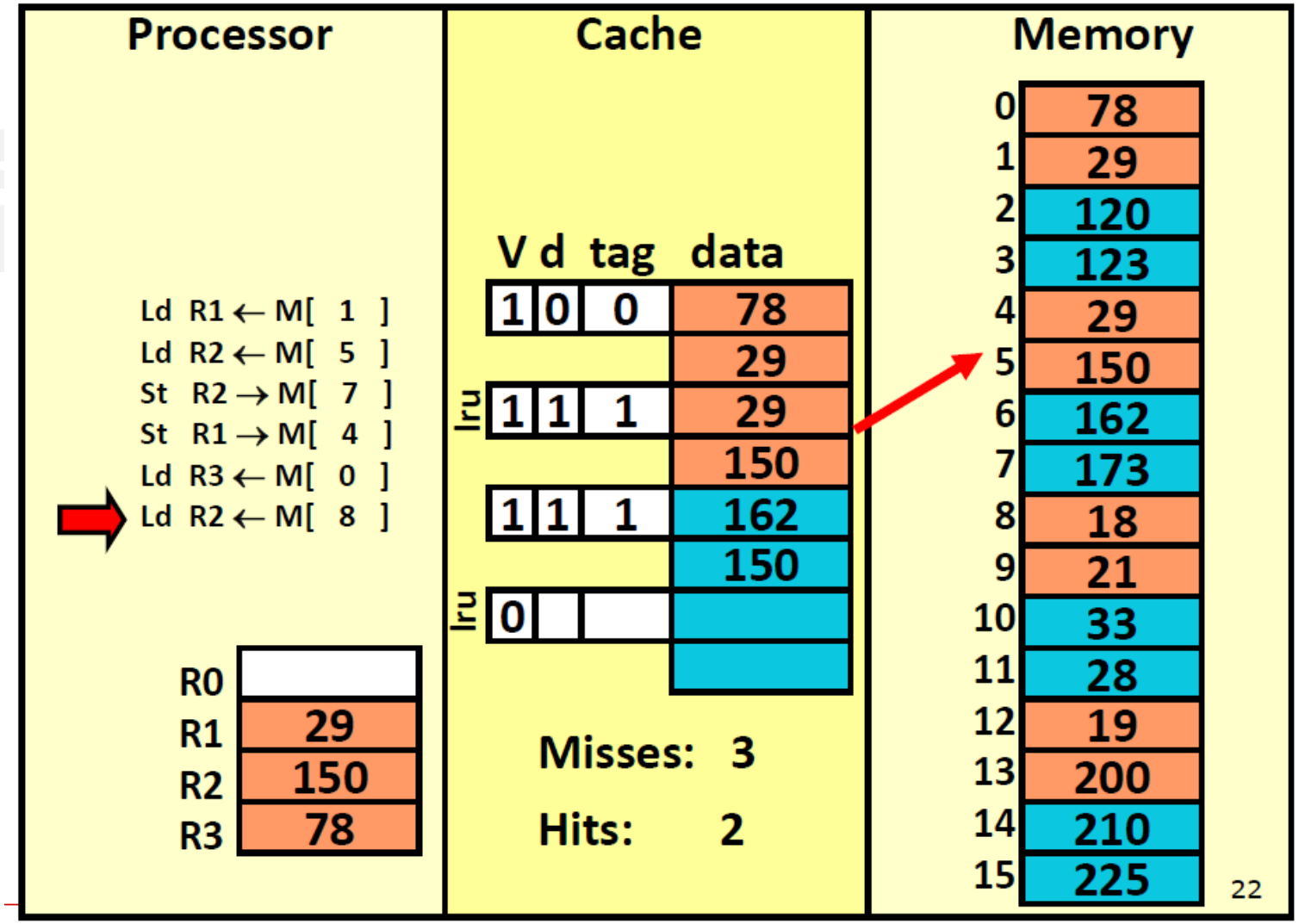
2-Way Set Associative Cache实例

- Write-back & Write-allocate



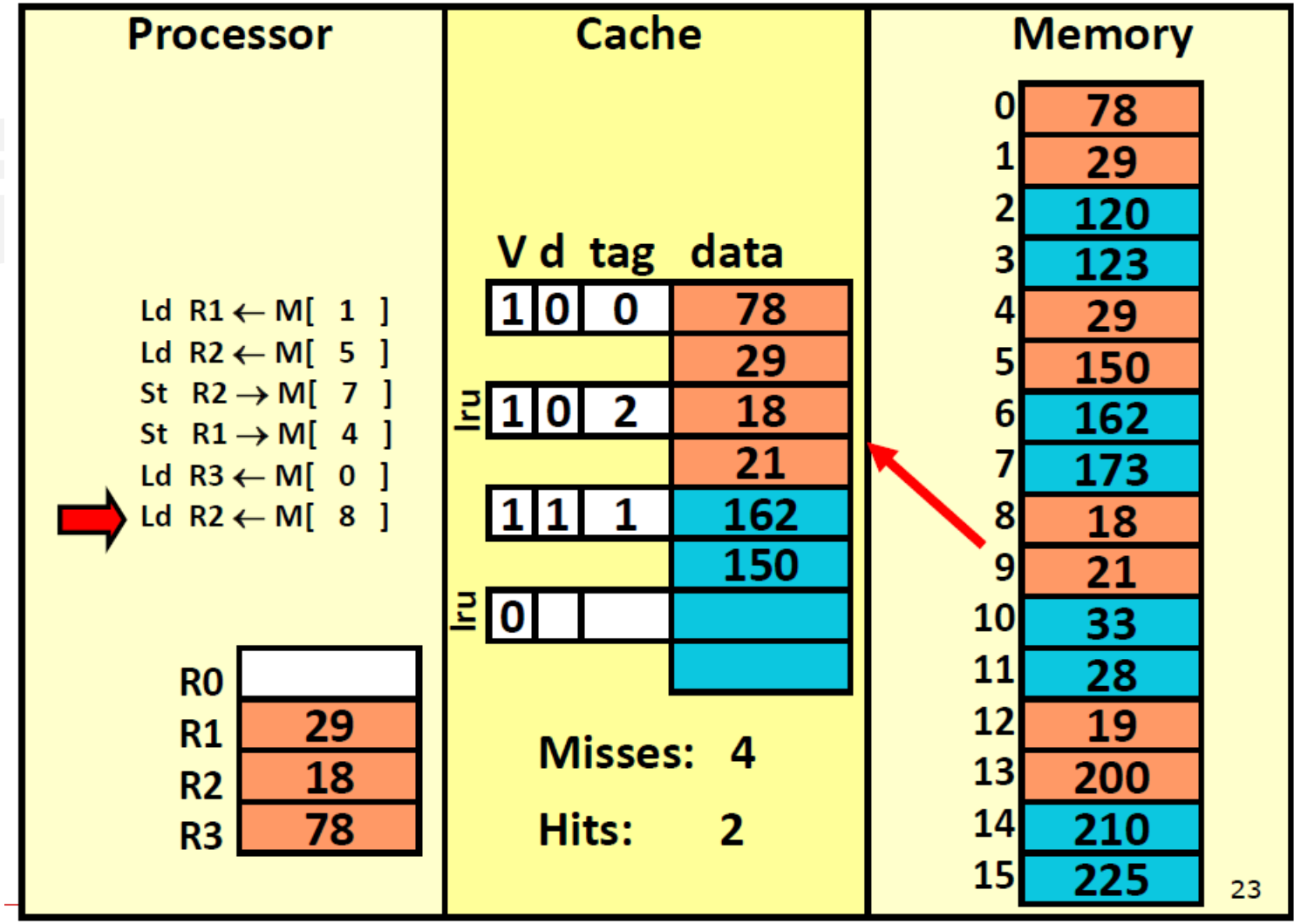
2-Way Set Associative Cache实例

- 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

For a 32-bit address and 16KB cache with 64-byte blocks, show the breakdown of the address for the following cache configuration:

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

- Cache Access时间分析

- **T_avg = T_hit + miss_ratio x T_miss**
  - comparable DM and SA caches with same T_miss
  - but, associativity that minimizes T_avg often smaller than associativity t hat minimizes miss_ratio

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

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

e.g.,

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

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

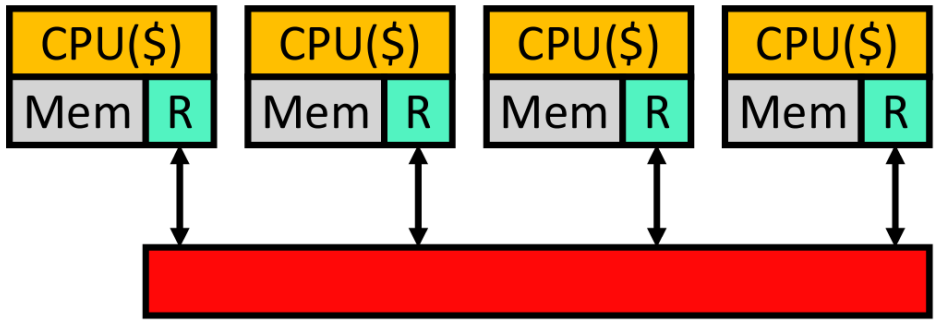
$\Rightarrow$ if $diff(t_{cache}) > 0.2$ cycle then SA loses

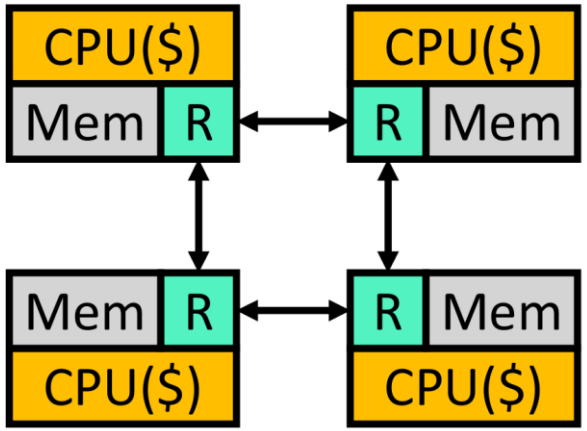
目 录

CONTENTS

多核架构的组织形式

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

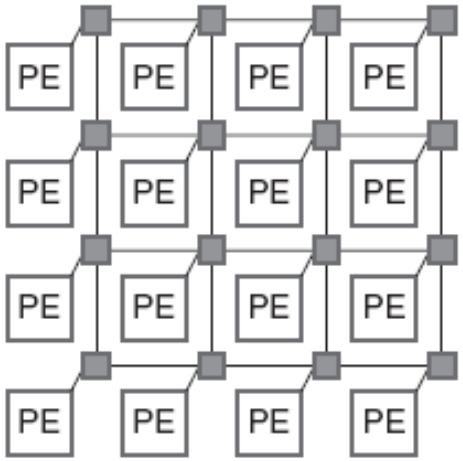


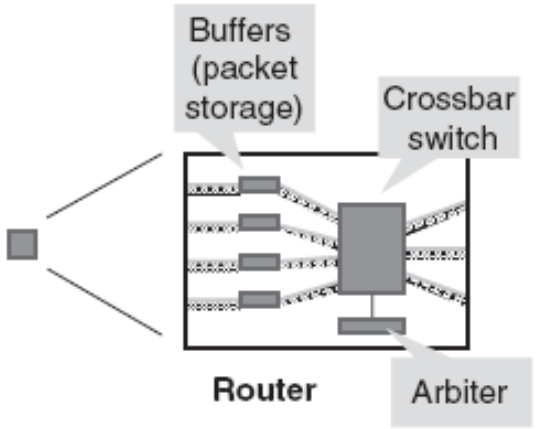


- 共享网络: e.g., bus
  - 低延迟
  - 低带宽：无法扩展到超过 16 个处理器
  - 共享属性简化了缓存一致性协议（后面会提到）
- 点对点网络: e.g., mesh or ring
  - 更长的延迟：可能需要多个“hops”才能进行通信
  - 更高的带宽：可扩展至数千个处理器
  - 缓存一致性协议很复杂

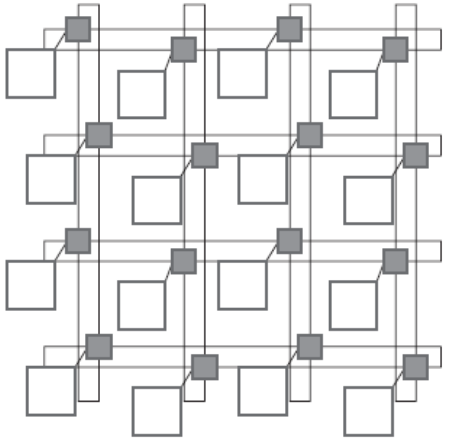
多核架构的组织形式

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

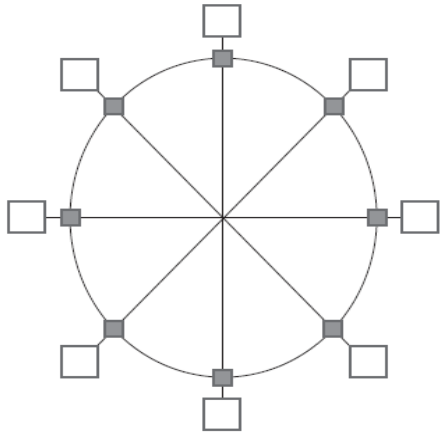




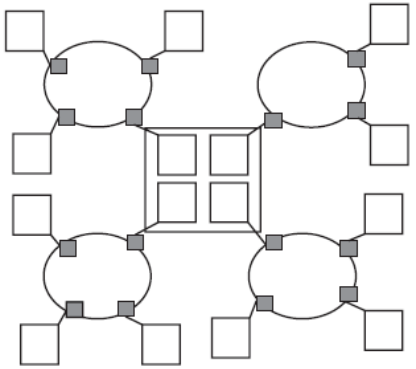
Mesh Topology



2-D torus topology



Octagon topology



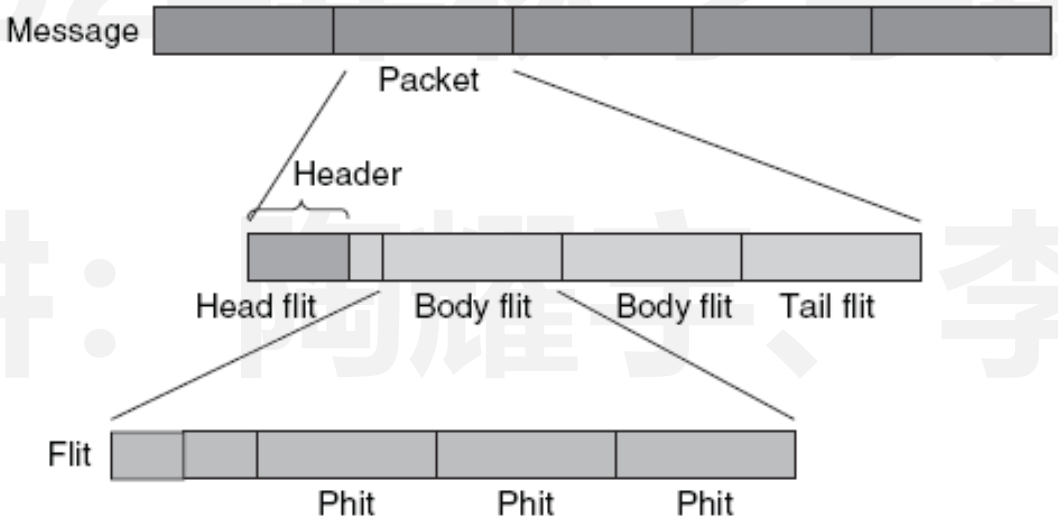
Irregular or ad hoc network topologies

多核架构的组织形式

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

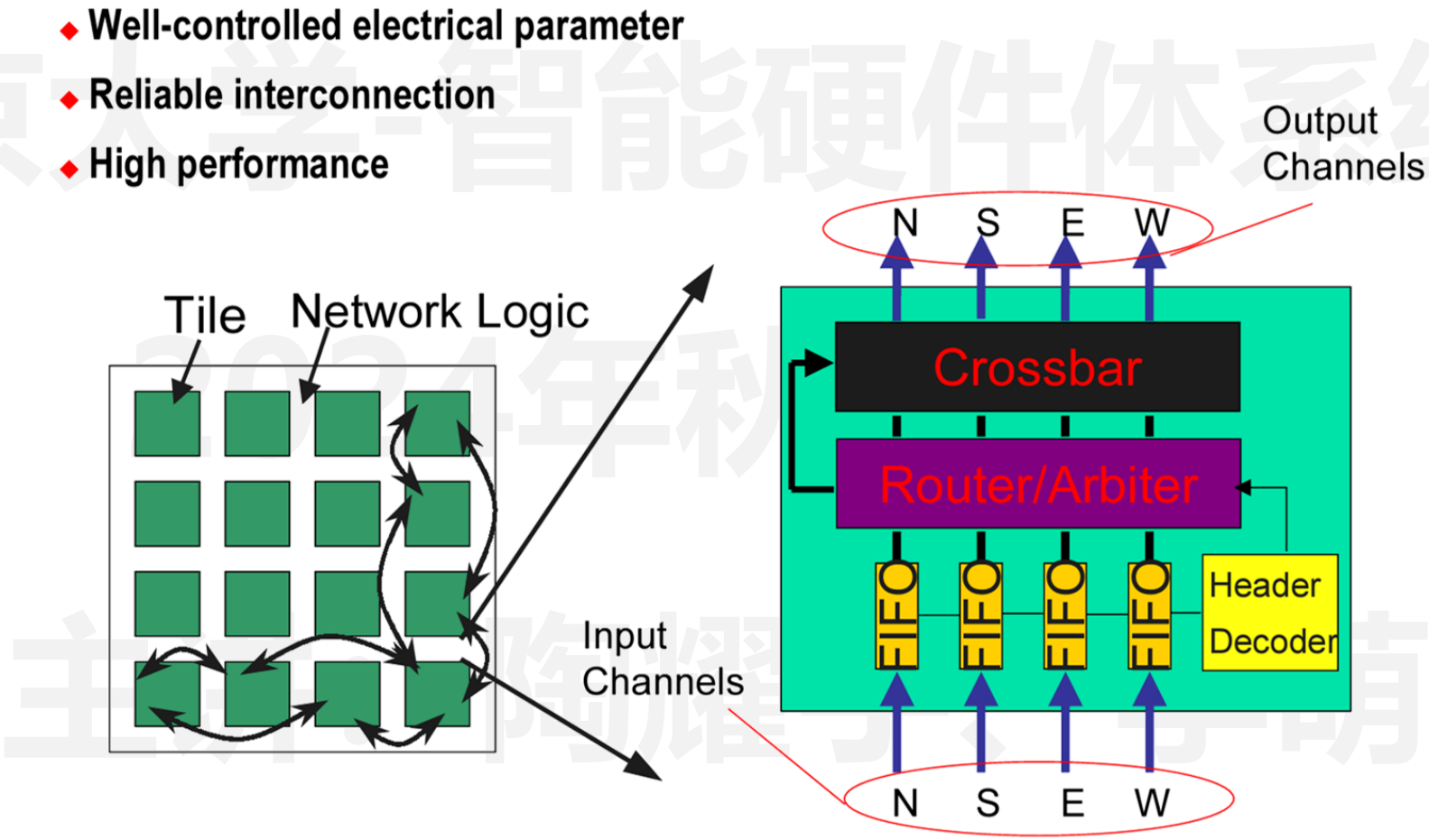
Switching strategies

- 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)**

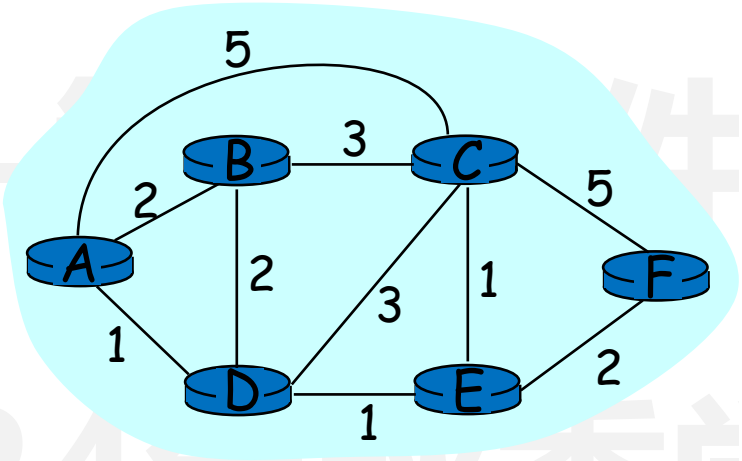
- Static and dynamic routing
  - **static routing**: 固定路径用于在特定源和目标之间传输数据
    - 没有考虑网络的当前状态
  - static routing优点:
    - 易于实现，因为几乎不需要额外的路由器逻辑
    - 如果使用单路径，则按顺序传送数据包
  - **dynamic routing**: 根据网络的当前状态做出路由决策
    - 考虑可用性和链接负载等因素
  - 源和目标之间的路径可能会随时间而改变
    - 随着**traffic**状况和应用要求的变化
  - 需要更多资源来监控网络状态并动态改变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

To Node \ From 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

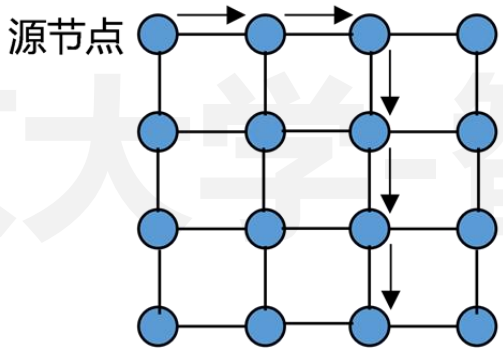
Destination	Next Node
1	4
2	4
3	3
4	4
6	6

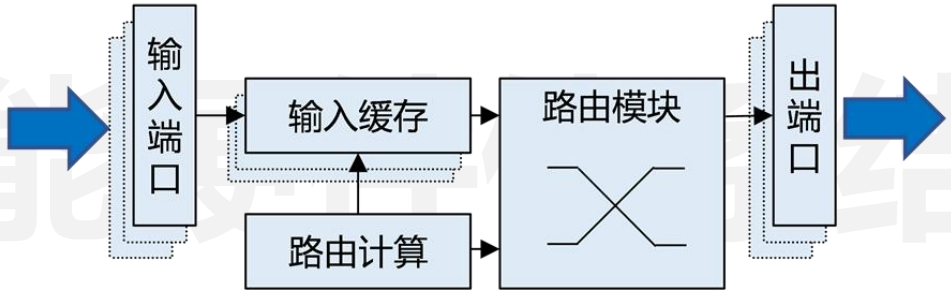
Node 6 Directory

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

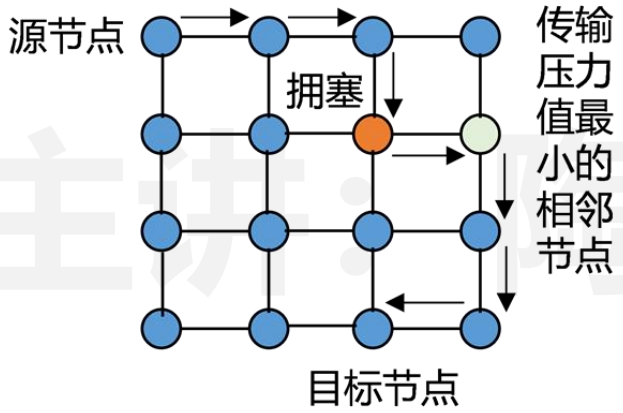
多核架构的组织形式

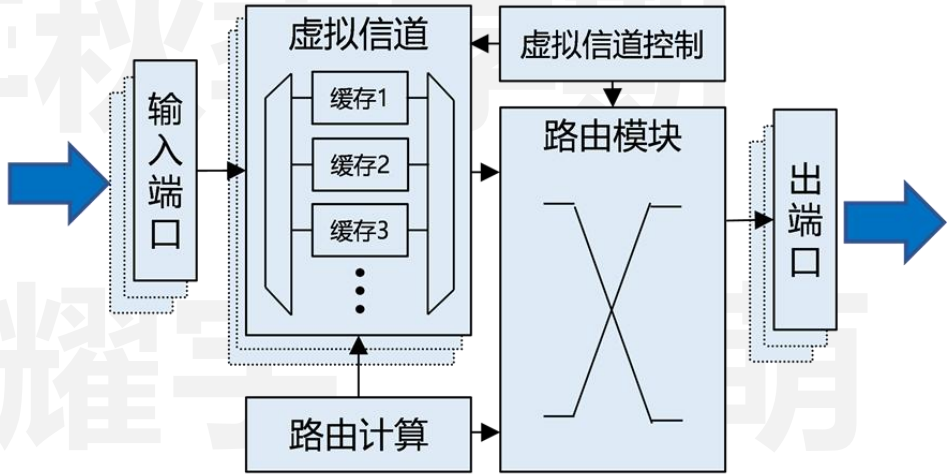
- Dynamic Routing





传统路由模块





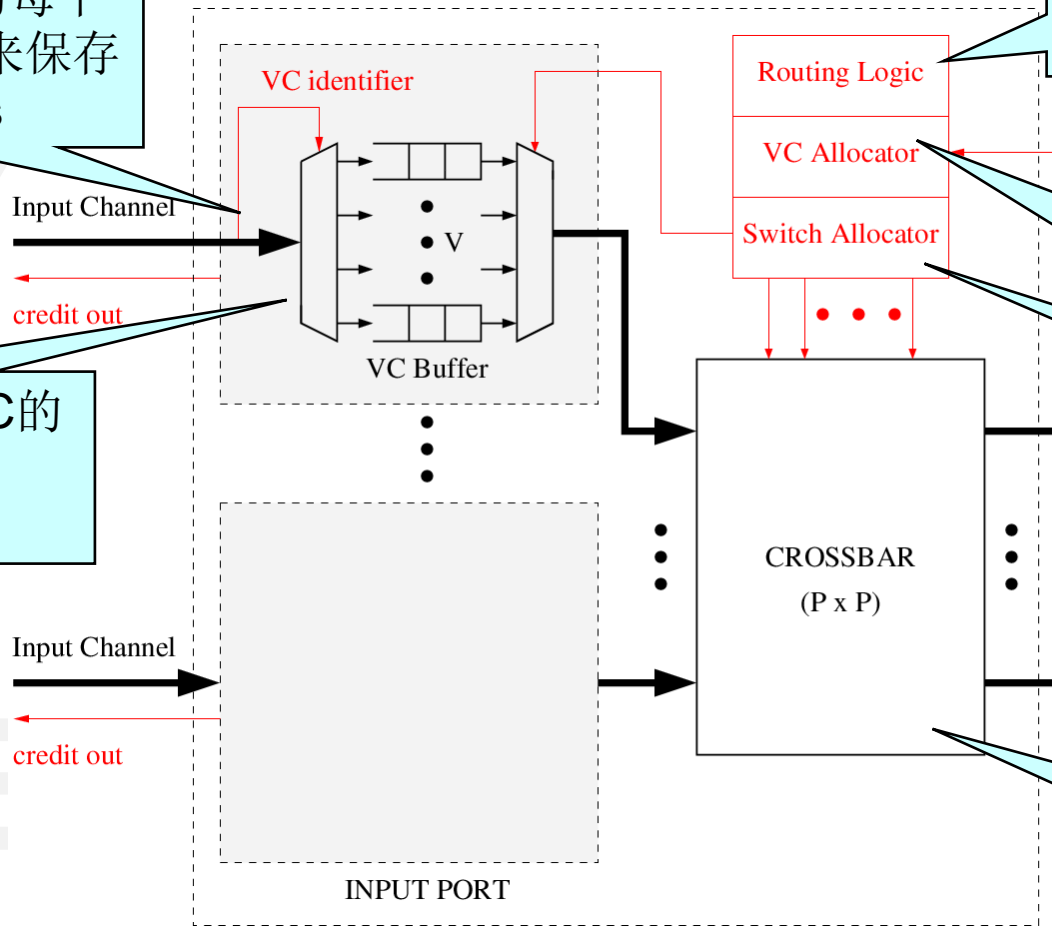
虚拟信道路由模块

多核架构的组织形式

- 虚拟信道路由

每个输入端口的每个 VC 都有缓冲区来保存到达的 flits

路由逻辑分配一组传出的 VC，以便 flit 可以继续传输



Arbitrates between competing input VC & allocates output VC

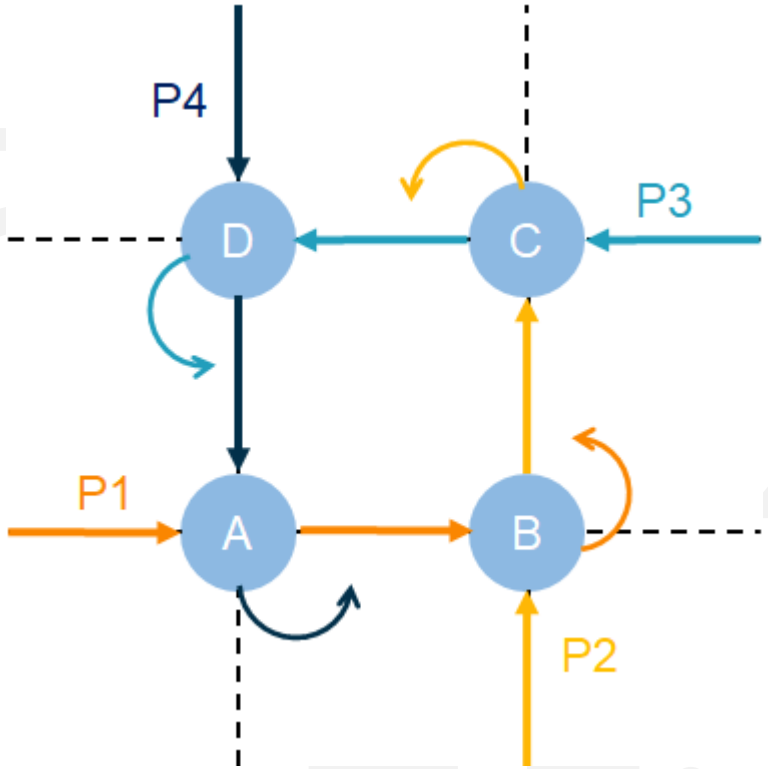
到达的flit被放入相应VC的缓冲区中

将成功的输入端口（分配的 VC）与输出端口匹配

输入 VC 处获得授权的 Flits 被传递到输出 VC

多核架构的组织形式

- 路由死锁Deadlock

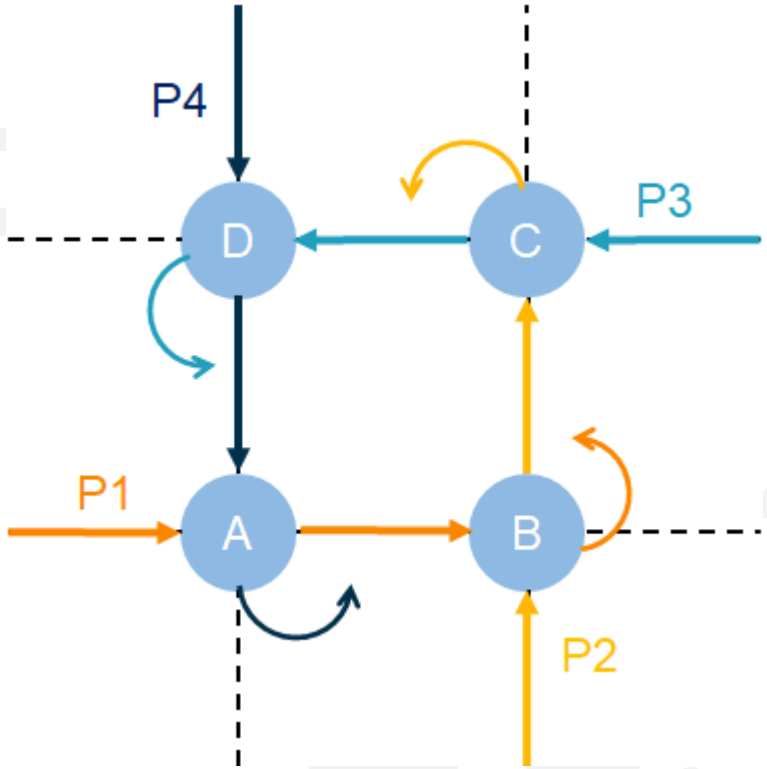


假设路由规则要求全路径资源空闲才可发送数据包

- **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

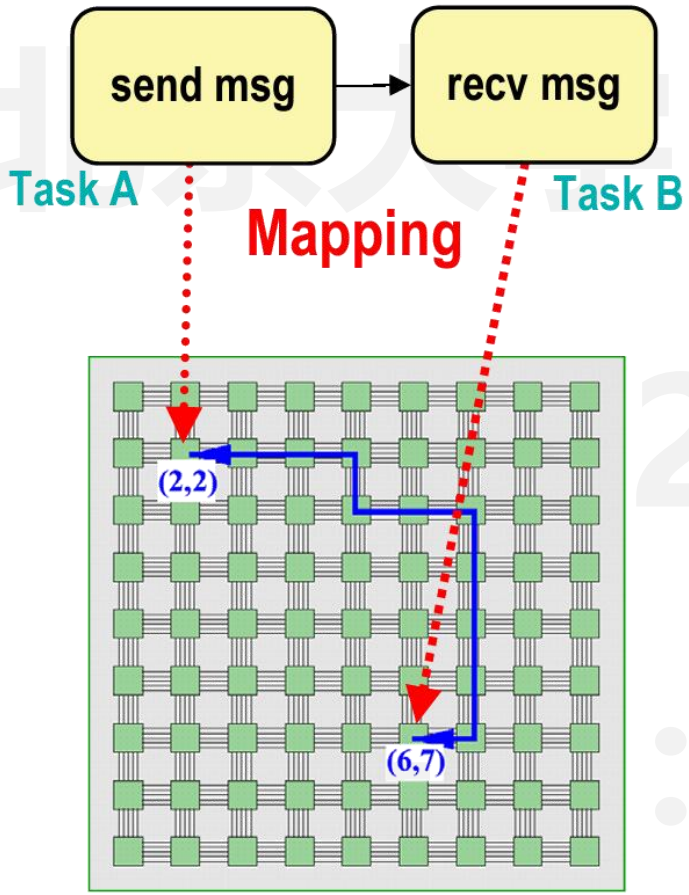
1.提供更多资源：该方法对应解决死锁条件中的互斥，主要采用两种策略：增加虚拟通道和消费通道。前者通过在路由端口增加额外缓存，避免单一通道资源占用导致的死锁，但是并不增加额外的物理传输通道；后者与前者相反，通过增加物理传输通道，从而在转发过程中提供更多资源

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.取消任务对资源的全链路占用：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

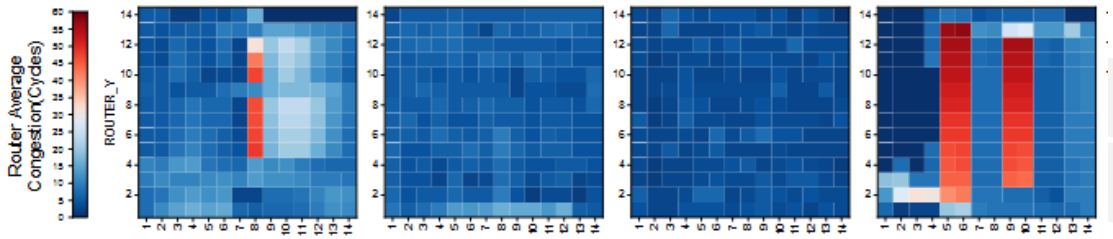
多核架构的组织形式

- 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？

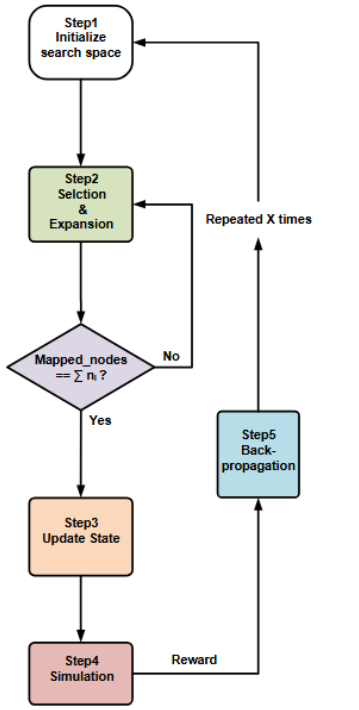
多核架构的组织形式

- Workload Mapping – 利用AI模型进行映射和片上网络优化



NoC的Traffic实时状态热图

- 将NoC映射问题抽象化为复杂棋子在复杂棋盘上的下棋问题
  - 利用RL、大模型等工具进行优化
  - 特别适用于神经网络的NoC映射

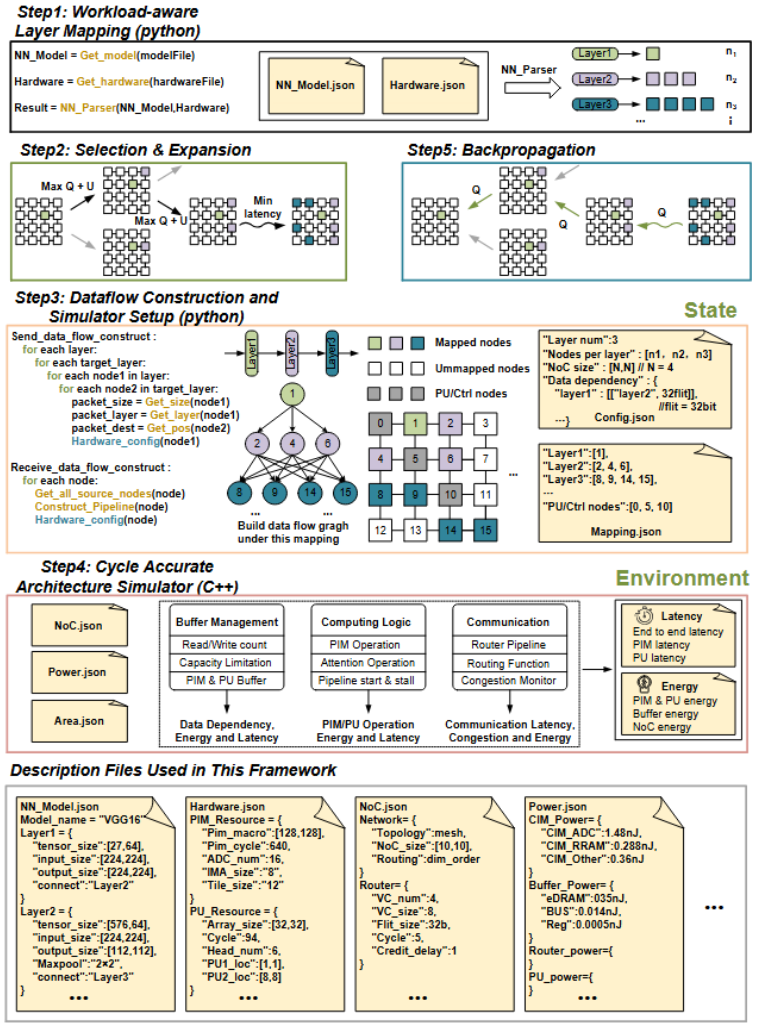


*Selection & Expansion: Incremental node placement guided by value function(Q) and evaluation(U).

*Update State: Update configuration file based on the new mapping state.

*Backpropagation: Backpropagate and update the value function of the selected node.

*Reward: Get reward from the environment based on current state.



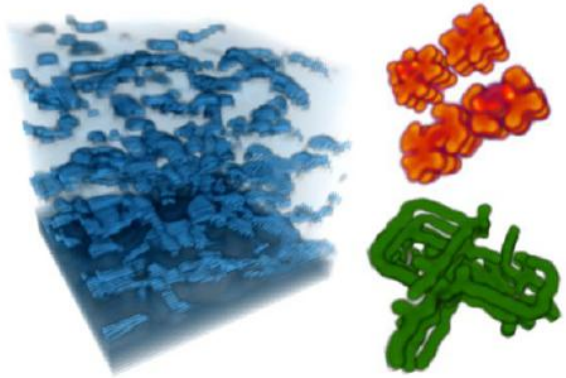
目 录

CONTENTS

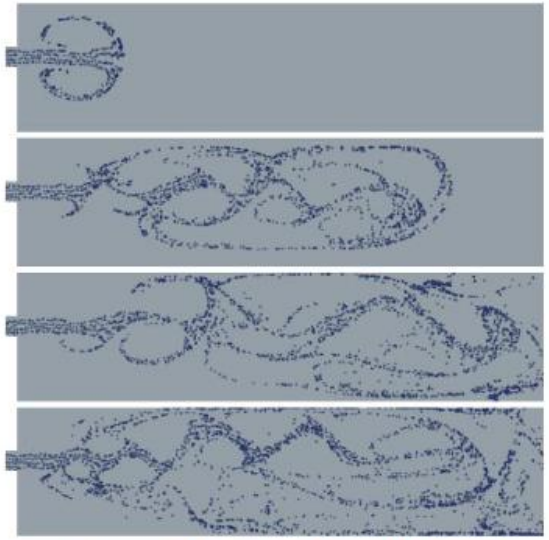
GPU: Overview

- Application

Great diversity of materials and lights in the world!



Coupled Map Lattice Simulation [Harris 02]

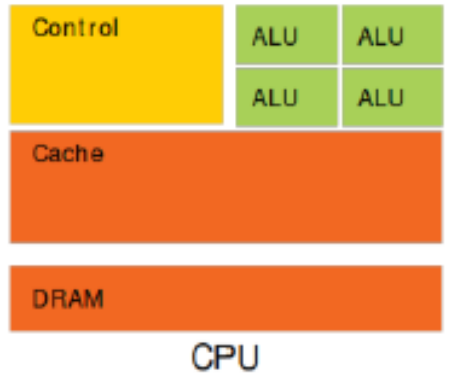


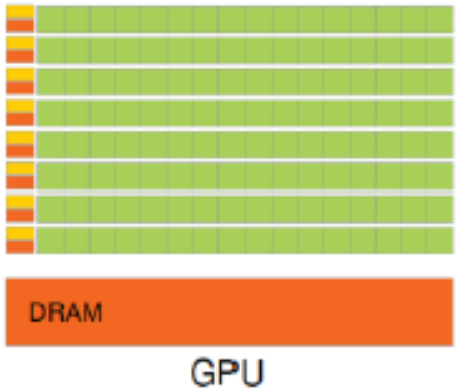
Sparse Matrix Solvers [Bolz 03]

GPU: Overview

- **Highly Parallel Coprocessor**

  - GPU特点
    - 有其专有的DRAM内存
    - 多个简单计算核心
    - 非常小的Cache
    - 并行运行多个线程
  - GPU Threads
    - GPU Threads十分轻量（几乎不需要creation/context switch）
    - 为了达到full efficiency，GPU需要至少几千个threads





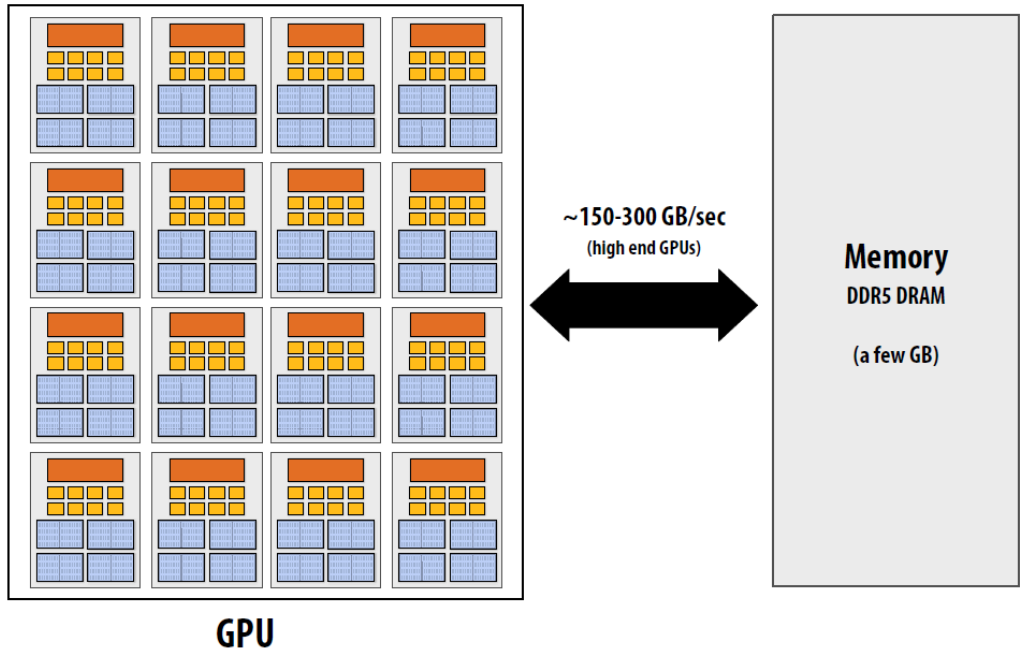
GPU: Overview

- What is GPU Good at

  - **GPU is good at data-parallel processing**
    - 对多个data element并行进行相同的计算操作——low control flow overhead
  - **High SP floating point arithmetic intensity**
    - Many calculations per memory access
    - 更多针对浮点数的运算而非整形计算
  - 更高的浮点数计算intensity和更多的data element意味着**memory access的延迟和计算的延时相比可以忽略不计**（不需要大的cache）

GPU: Overview

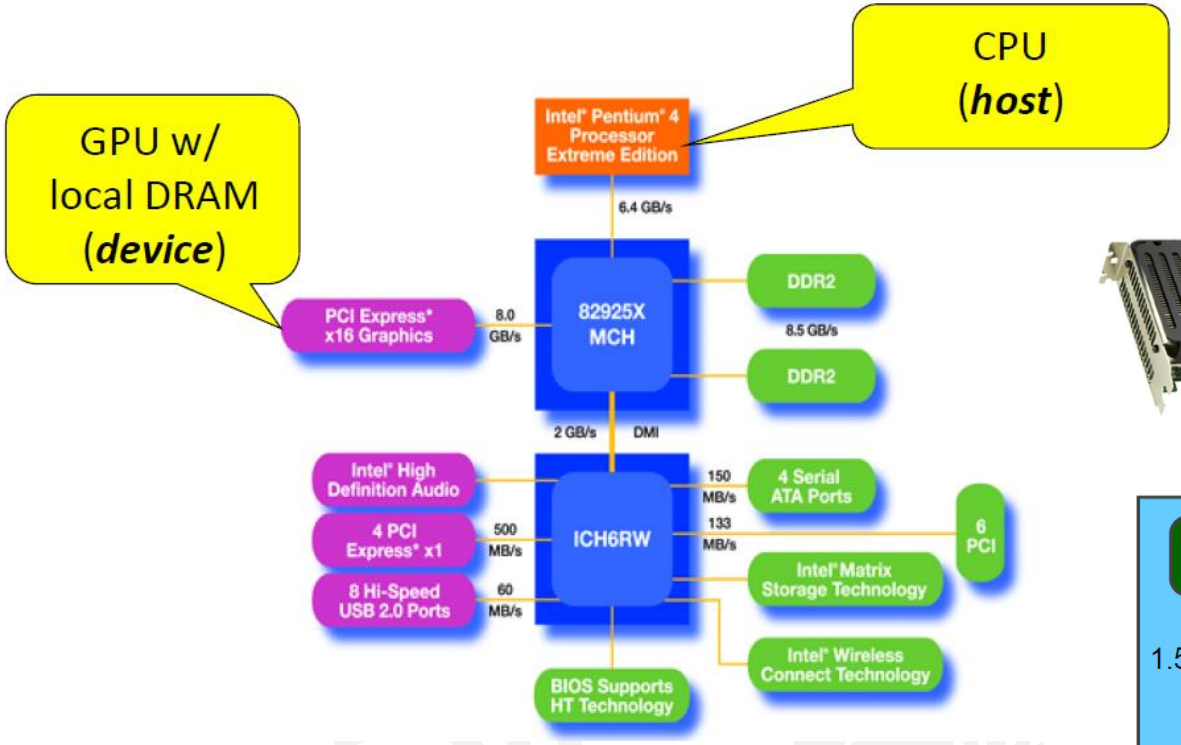
- General Purpose GPU

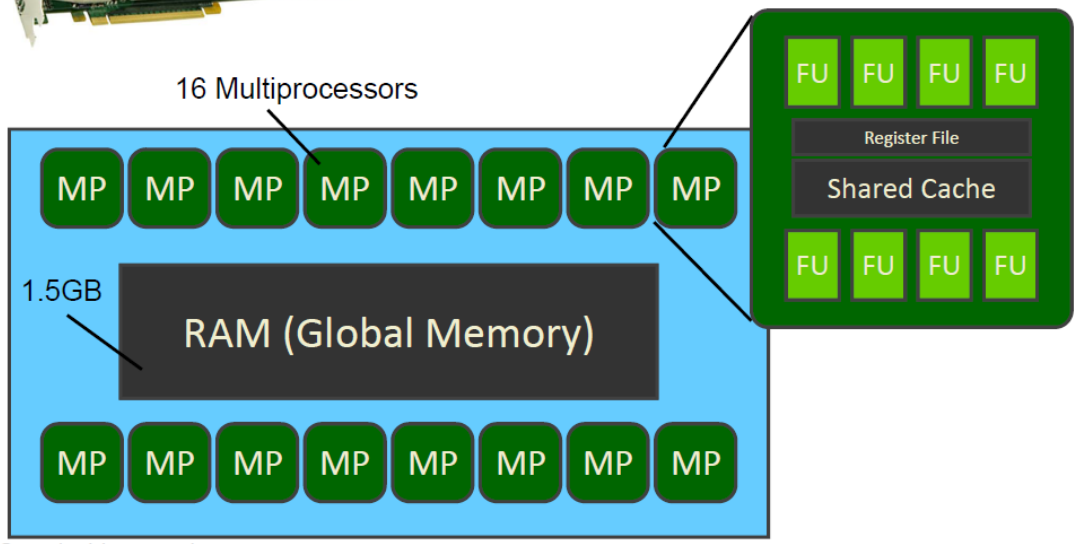


- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: **CUDA**
  - "Compute Unified Device Architecture"
  - Subsequently, broader industry pushing for OpenCL, a vendor -neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general -purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GPGPU for execution

GPU: Overview

- **Example GGPGPU System & Example GPU**





© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

7

GPU: Architecture

北京大学 PEKING UNIVERSITY

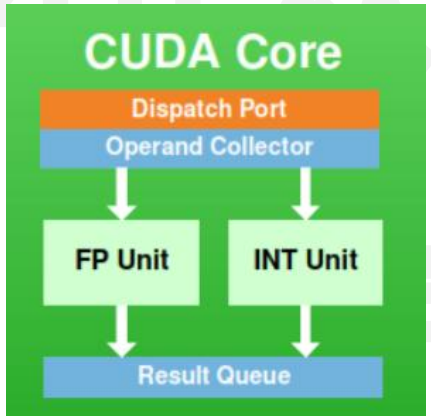
- **Example：NVIDIA Fermi Architecture**

Streaming Processor (SM)

GPU: Architecture

- **Example：NVIDIA Fermi Architecture**

Streaming Processor (SM)

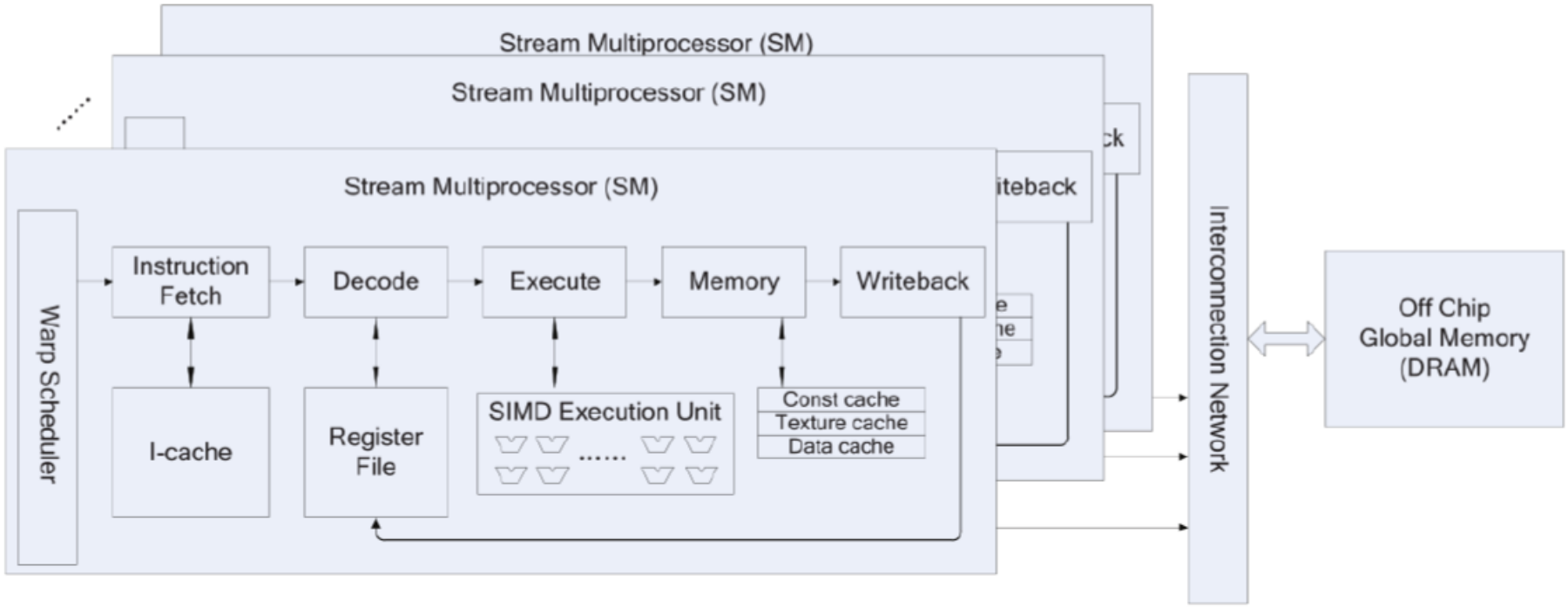


- 每个SM内部有32个Streaming Processor (SP)
- 单个芯片集成多至512个SP
- 峰值算力~200GOPS

GPU: Architecture

- **Warp**

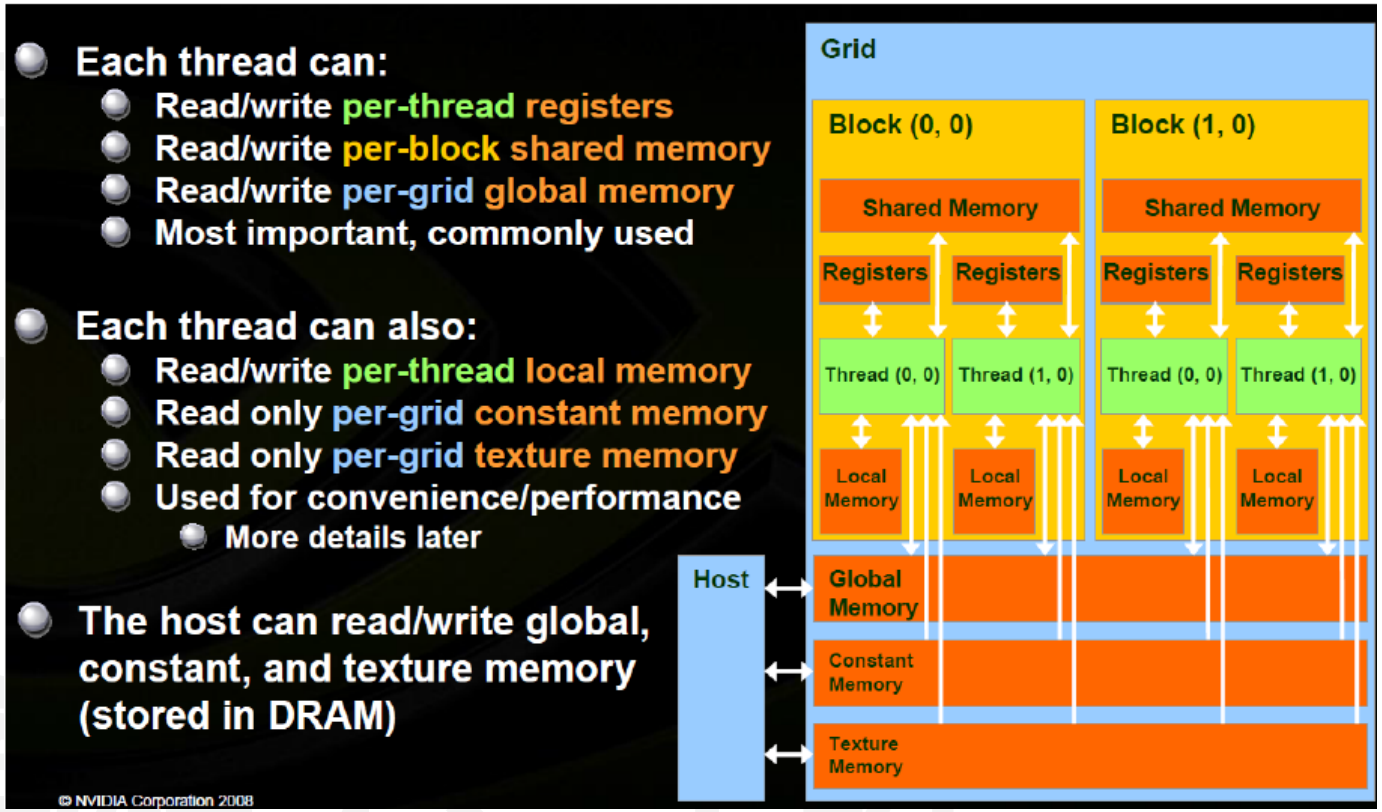
最小thread scheduling unit （对于NVIDIA GPU为32 threads）



GPU: Architecture

- **Memory Hierarchy**

- Shared memory/L1 cache: ~50 cycles
- L2 cache: ~150 cycles
- Global memory (GDDR5): ~500 cycles



GPU: H100 Overview

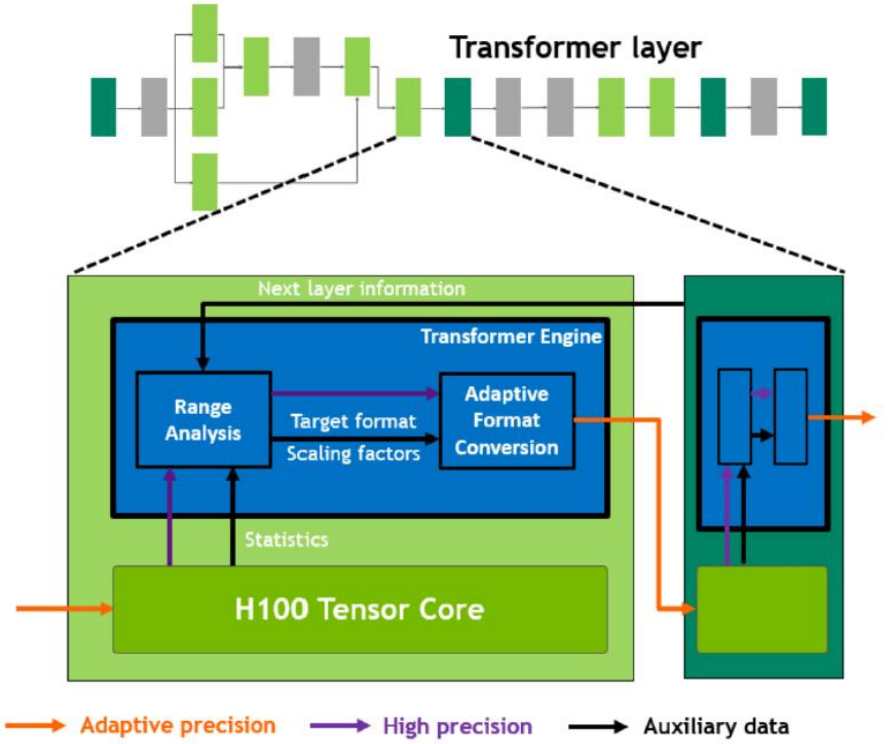
北京大学 PEKING UNIVERSITY

- **Overview**
  - 性能带宽相对前一代提升
    - 增加FP32 Core、FP64 Core、Tensor Core、SM Core
    - 采用HBM3、 HBM2e增加访存带宽

GPU Features	NVIDIA A100	NVIDIA H100 SXM5[1]	NVIDIA H100 PCIe[1]
GPU Architecture	NVIDIA Ampere	NVIDIA Hopper	NVIDIA Hopper
GPU Board Form Factor	SXM4	SXM5	PCIe Gen 5
SMs	108	132	114
TPCs	54	62	57
FP32 Cores / SM	64	128	128
FP32 Cores / GPU	6912	16896	14592
FP64 Cores / SM (excl. Tensor)	32	64	64
FP64 Cores / GPU (excl. Tensor)	3456	8448	7296
INT32 Cores / SM	64	64	64
INT32 Cores / GPU	6912	8448	7296
Tensor Cores / SM	4	4	4
Tensor Cores / GPU	432	528	456
GPU Boost Clock (Not Finalized for H100) [3]	1410 MHz	Not Finalized	Not Finalized
Peak FP8 Tensor TFLOPS with FP16 Accumulate[1]	NA	2000/4000[2]	1600/3200[2]
Peak FP8 Tensor TFLOPS with FP32 Accumulate[1]	NA	2000/4000[2]	1600/3200[2]
Peak FP16 Tensor TFLOPS with FP16 Accumulate[1]	312/624[2]	1000/2000[2]	800/1600[2]
Peak FP16 Tensor TFLOPS with FP32 Accumulate[1]	312/624[2]	1000/2000[2]	800/1600[2]
Peak BF16 Tensor TFLOPS with FP32 Accumulate[1]	312/624[2]	1000/2000[2]	800/1600[2]
Peak TF32 Tensor TFLOPS[1]	156/312[2]	500/1000[2]	400/800[2]
Peak FP64 Tensor TFLOPS[1]	19.5	60	48
Peak INT8 Tensor TOPS[1]	624/1248[2]	2000/4000[2]	1600/3200[2]
Peak FP16 TFLOPS (non-Tensor)[1]	78	120	96
Peak BF16 TFLOPS (non-Tensor)[1]	39	120	96
Peak FP32 TFLOPS (non-Tensor)[1]	19.5	60	48
Peak FP64 TFLOPS (non-Tensor)[1]	9.7	30	24
Peak INT32 TOPS[1]	19.5	30	24
Texture Units	432	528	456

GPU: H100 Improvements

- Transformer Engine
  - 支持混合精度
  - 根据硬件统计分布动态调整scaling factor
    - 硬件统计每层的output range
      - 考虑下一层的range
      - 计算得到高精度的结果
      - scale回下一层的range



目 录

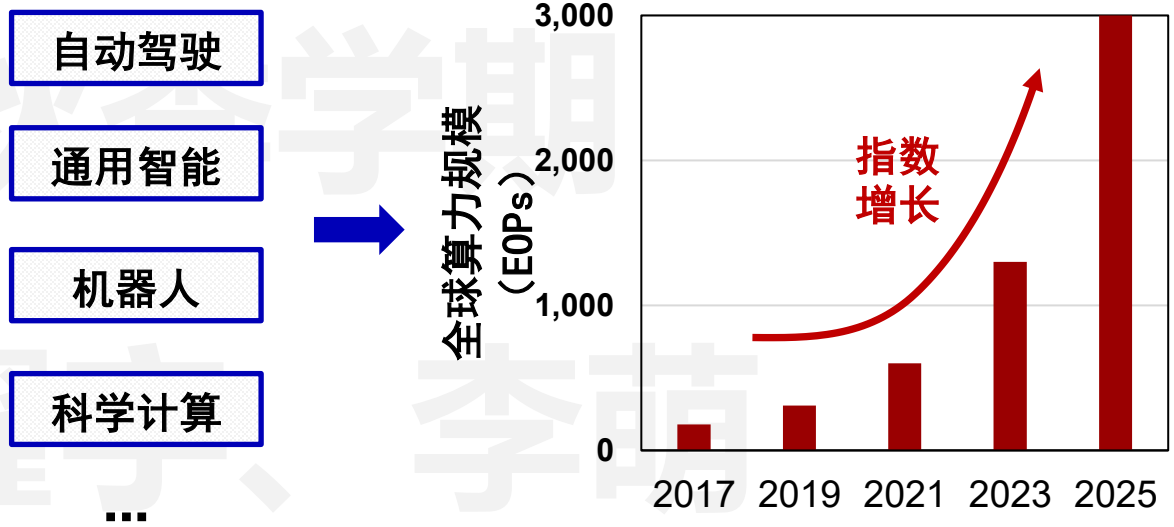
CONTENTS

目标应用 —— 深度学习与深度神经网络

- 过去10年，得益于**算力的指数级**提升，以深度学习为代表的人工智能快速发展
  - 从全连接网络（MLP）到卷积神经网络（CNN）、循环神经网络（RNN）再到注意力网络（Transformer）和大模型
- 人工智能已经成为驱动半导体和硬件体系结构发展的重要驱动力

时间	模型	模型参数量 (GB)	模型算力需求 (TFLOPs-day)
2012	AlexNet	10^{-2}	10^{-3}
2015	ResNets	10^{-1}	10^{-1}
2017	Transformer	1	10^{1}
2020	GPT-3	10^{2}	10^{6}
2023	ChatGPT	10^{3}	10^{7}

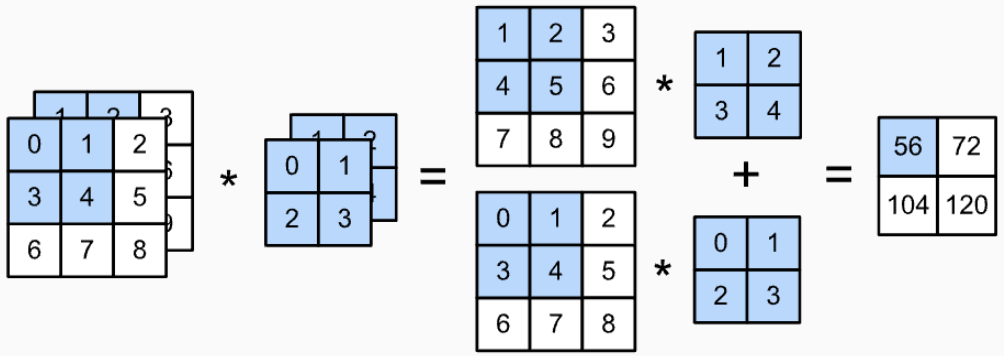
AI算法参数量与算力需求增长迅猛

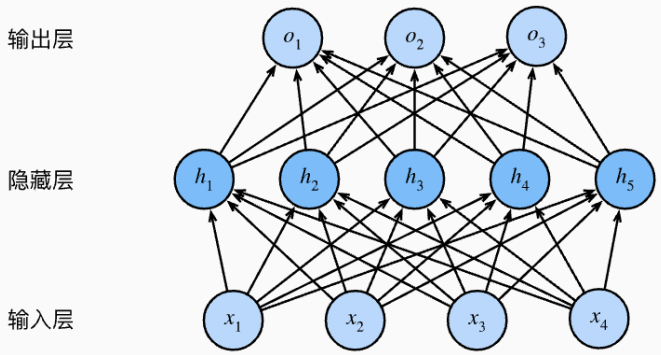


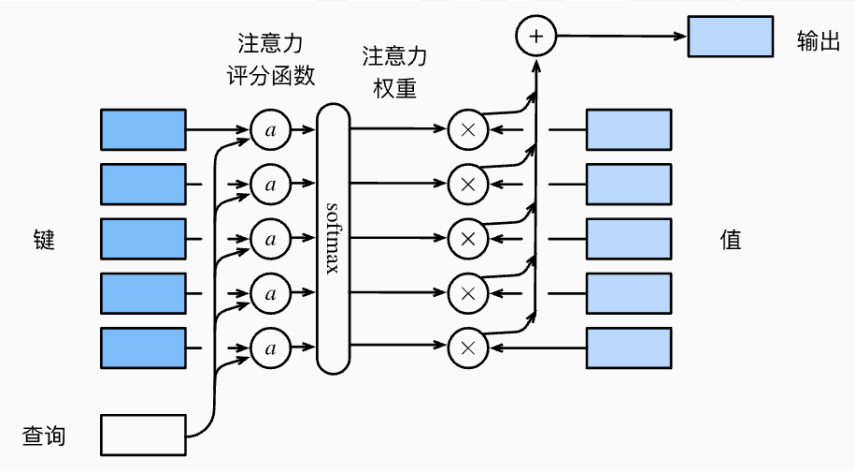
算力需求驱动硬件体系结构发展

目标应用 —— 深度学习与深度神经网络

- 典型神经网络**算子**和**拓扑**结构
  - **算子**：卷积、全连接、注意力机制、归一化函数、非线性函数等，主要操作对象为**张量、向量**等

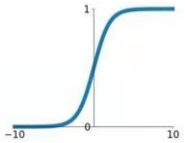






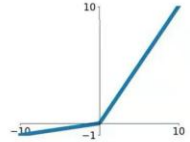
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



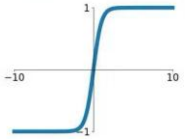
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

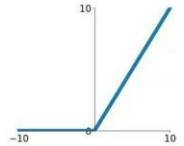


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

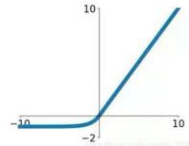
ReLU

$$\max(0, x)$$



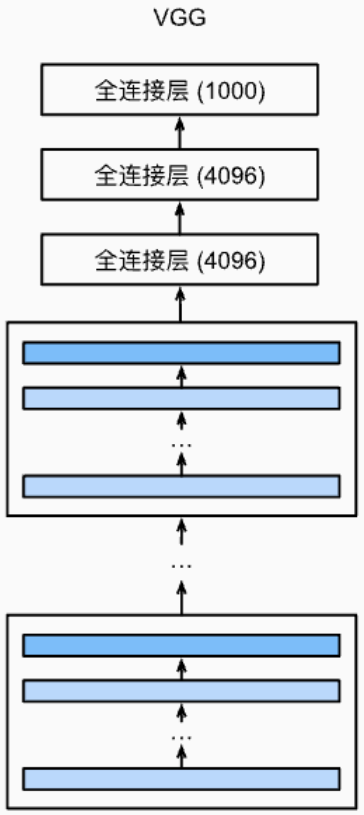
ELU

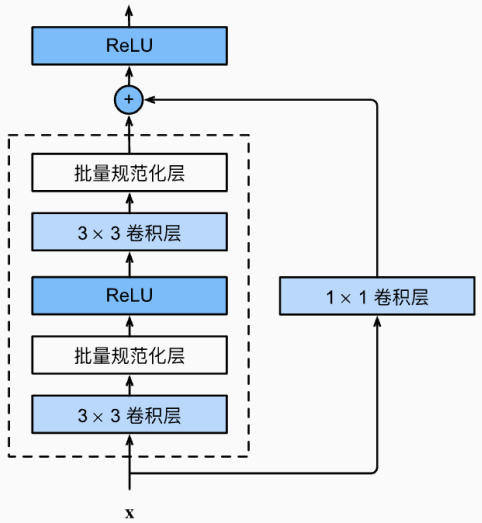
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

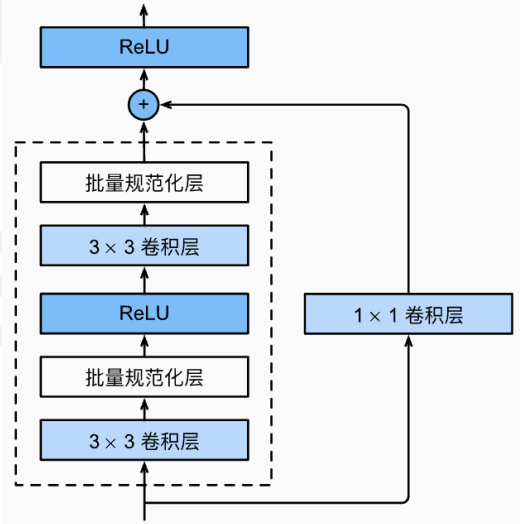


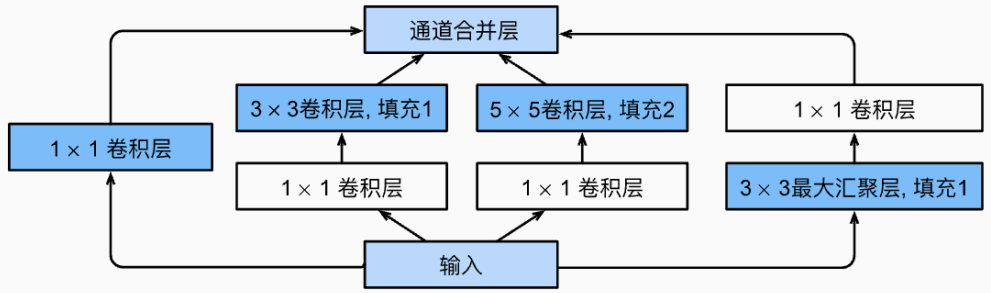
目标应用 —— 深度学习与深度神经网络

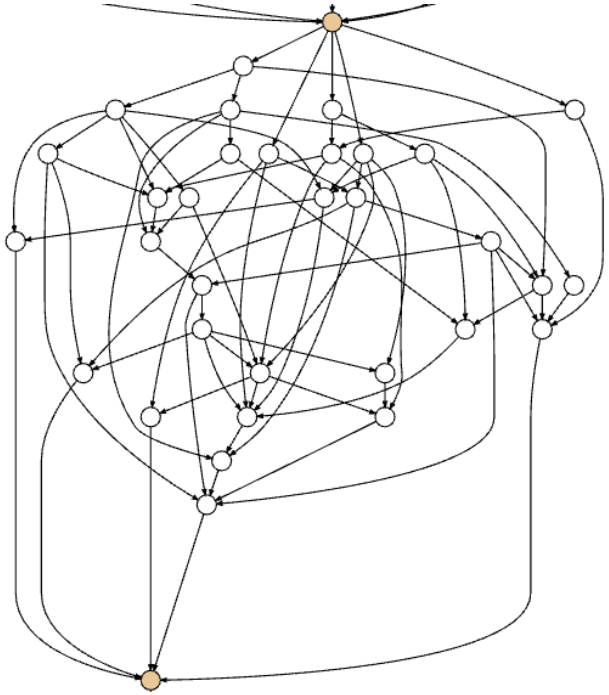
- 典型神经网络**算子**和**拓扑**结构
  - **算子**：卷积、全连接、注意力机制、归一化函数、非线性函数等，主要操作对象为张量、向量等
  - **拓扑**：线性结构、残差结构、多分支结构、随机结构











北京大学 PEKING UNIVERSITY

“没关系，AI芯片都是在加速线性算子，而线性算子都可以视为矩阵乘法”

—— 常见误区

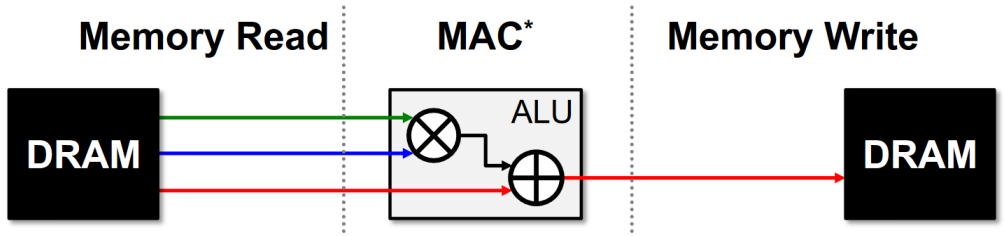
$$\begin{bmatrix} 1 & 0 & 2 \\ 3 & 1 & 0 \\ 5 & -1 & 2 \end{bmatrix} \times \begin{bmatrix} 2 & -1 & 0 \\ 5 & 1 & -1 \\ -2 & 0 & 0 \end{bmatrix} = \begin{bmatrix} -2 & -1 & 0 \\ 11 & -2 & -1 \\ 1 & -6 & 1 \end{bmatrix}$$

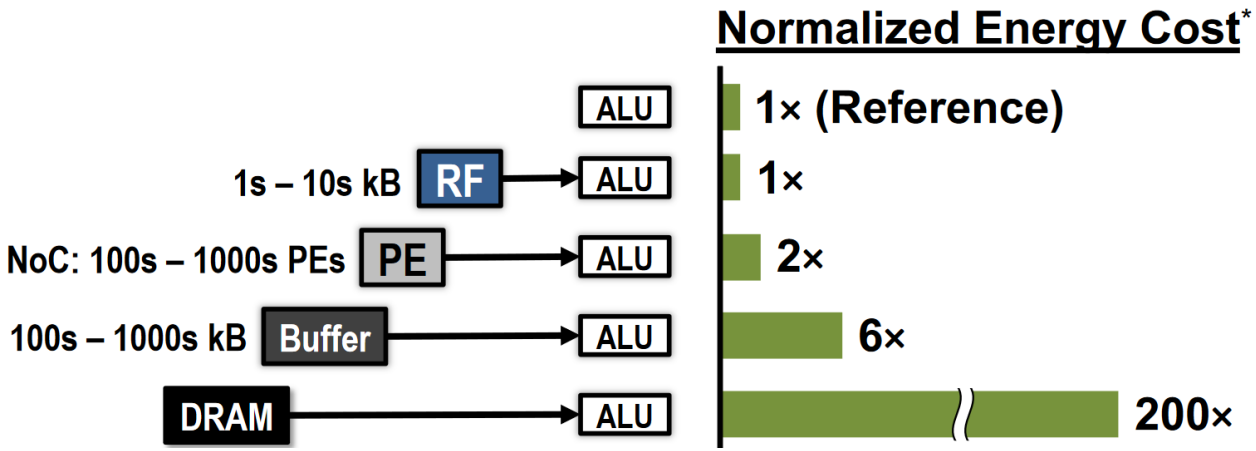
- 与CPU中的很多负载相比，深度神经网络计算的核心差别有哪些?
- 是否还有其他？对于硬件的影响又是什么样子的呢?

	CPU常见负载	神经网络
操作对象	数据种类多样	**张量、向量为主**
控制流	控制流复杂灵活	**静态固定数据流为主**
数据流	复杂不规则	**具有较为固定形式**

AI加速器架构基础

- 为什么传统CPU进行深度神经网络计算是不高效的？CPU的向量指令呢?
- 需要开发专用AI加速器，提升神经网络的计算效率

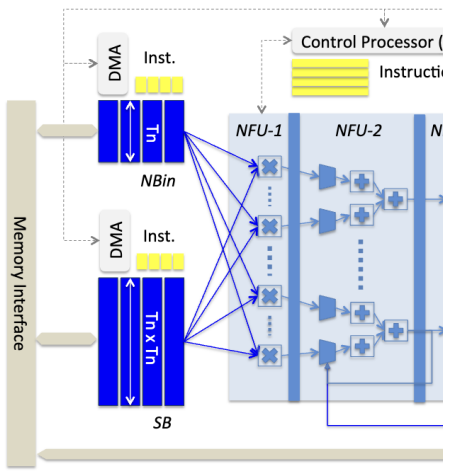




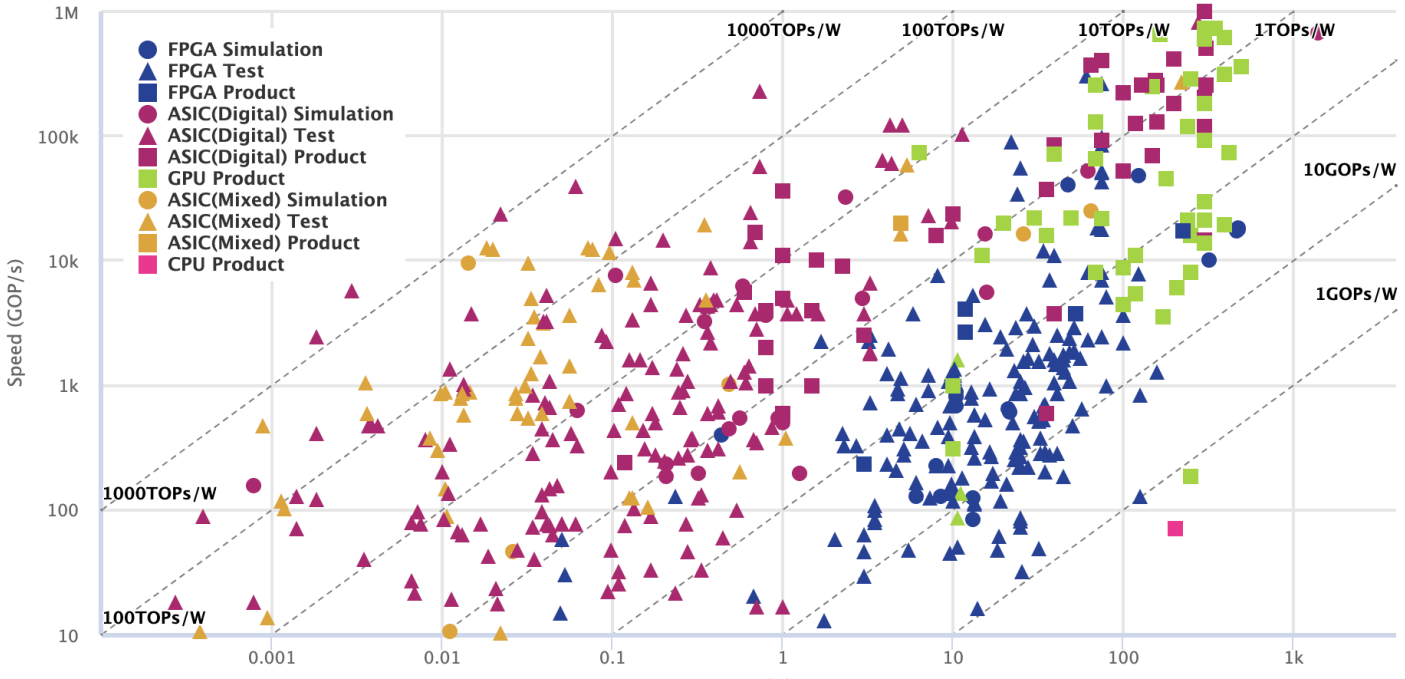
AI加速器架构基础

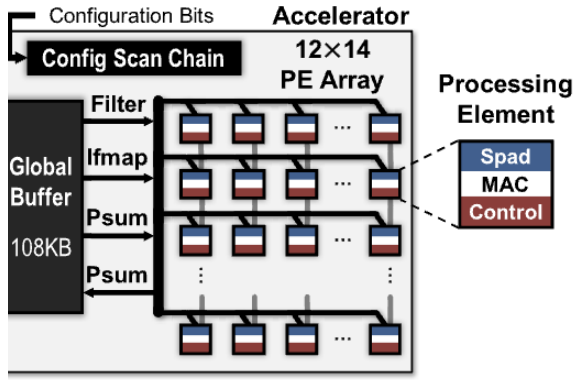
北京大学 PEKING UNIVERSITY

- 在深度神经网络的计算需求驱动下，学术界和业界都在专用AI加速器领域开展了广泛的研究



DianNao



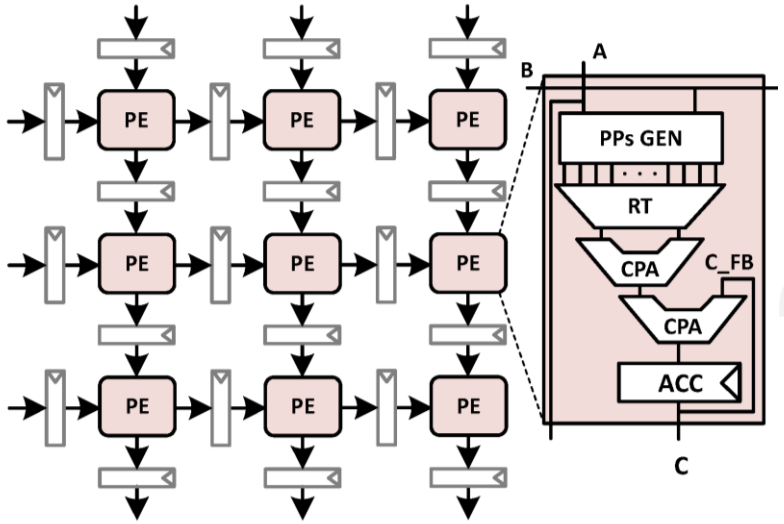


yeriss

Meta Google amazon TESLA 理想 Alibaba

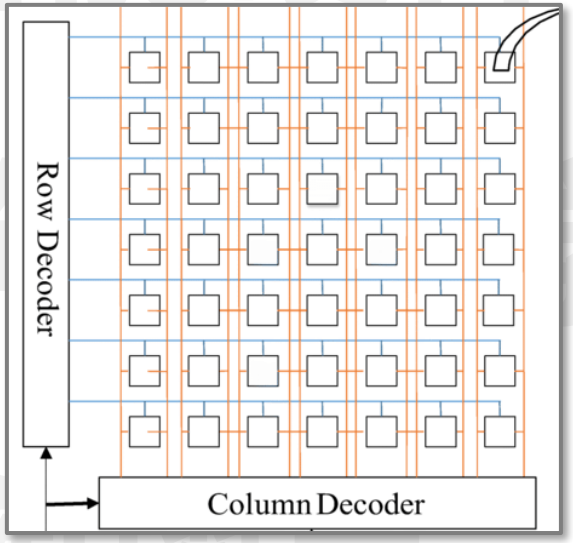
AI加速器架构基础

- AI加速器整体架构



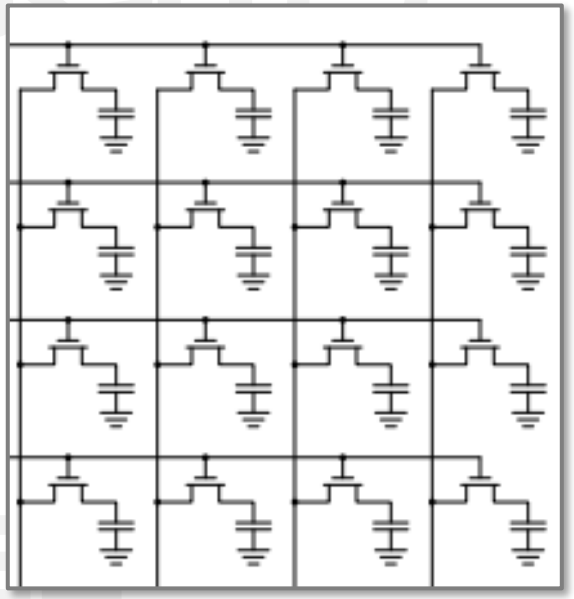
计算

On-chip SRAM



访存

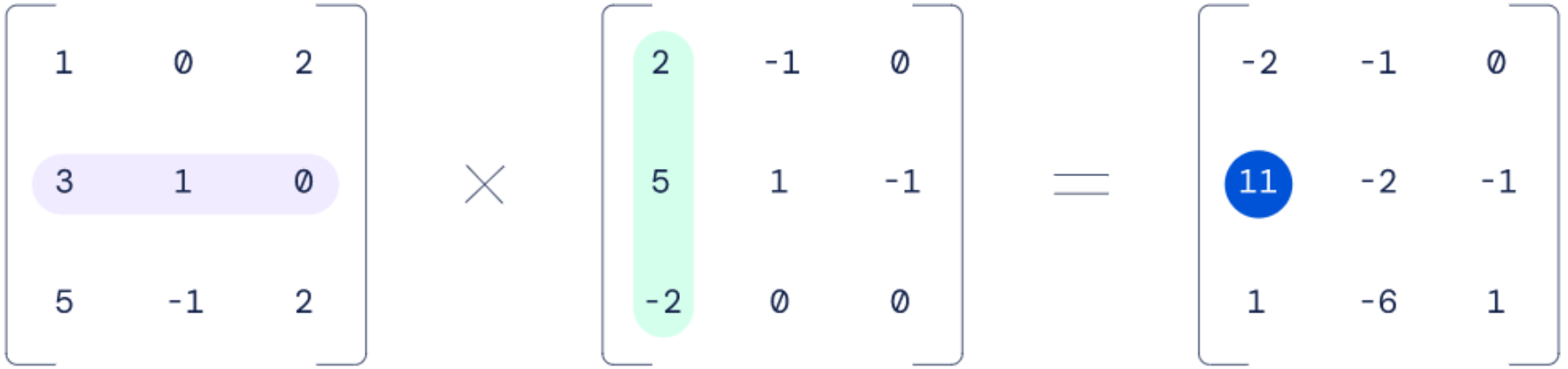
Off-chip DRAM



通信

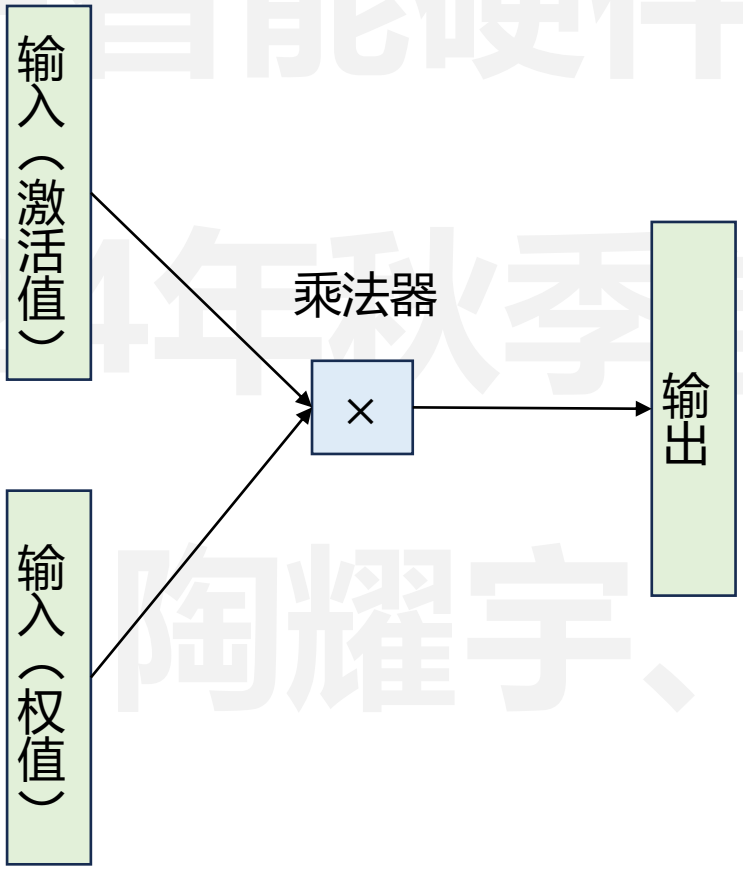
AI加速器架构基础——计算单元

- 计算单元：主要包含面向矩阵、向量和标量三种
  - 分别对应神经网络计算过程中矩阵、向量和标量计算算子
- 针对计算单元，我们经常关心的指标是**计算访存比**，高计算访存比，通常能带来更高的能效
- 计算访存比同时受到**计算负载**和**硬件架构**的影响

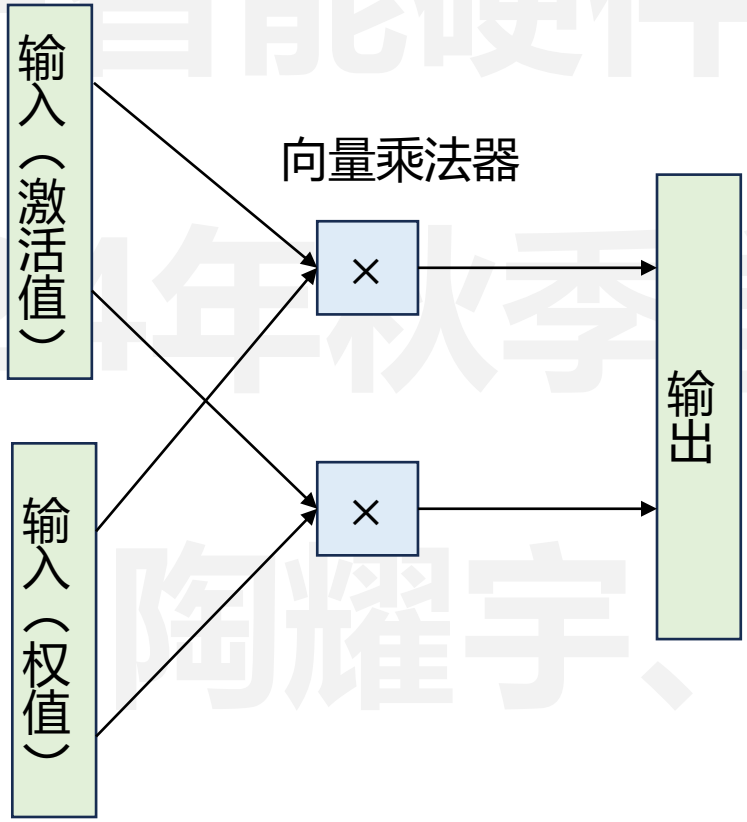


AI加速器架构基础——计算单元

- **矩阵计算单元：通过内积计算，最大化数据复用**

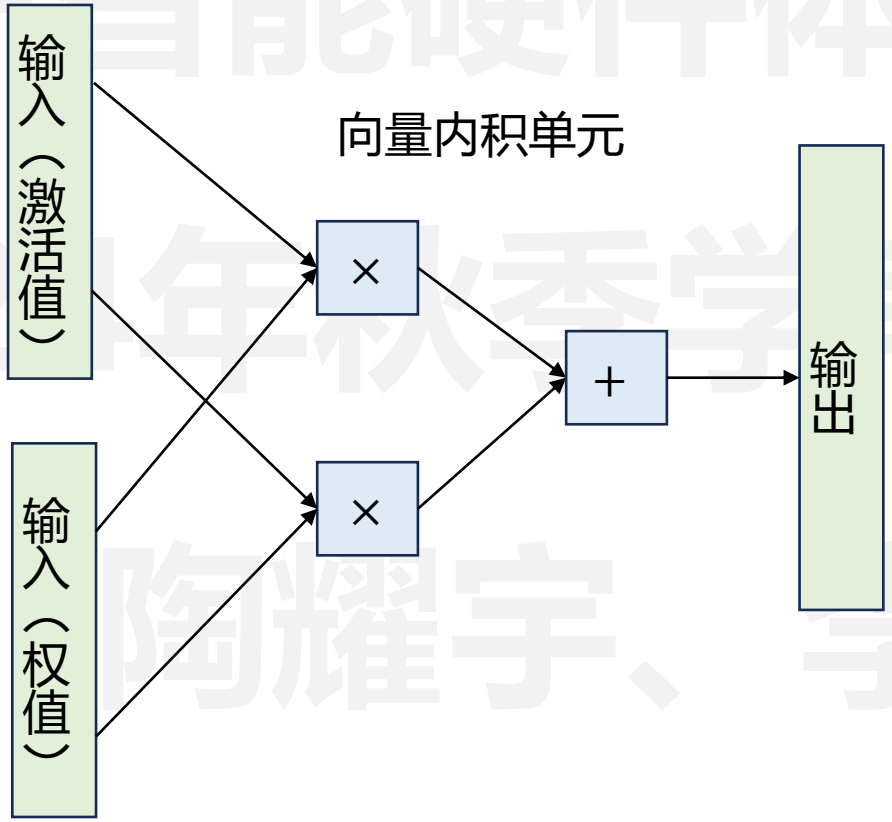


- 矩阵计算单元：通过内积计算，最大化数据复用

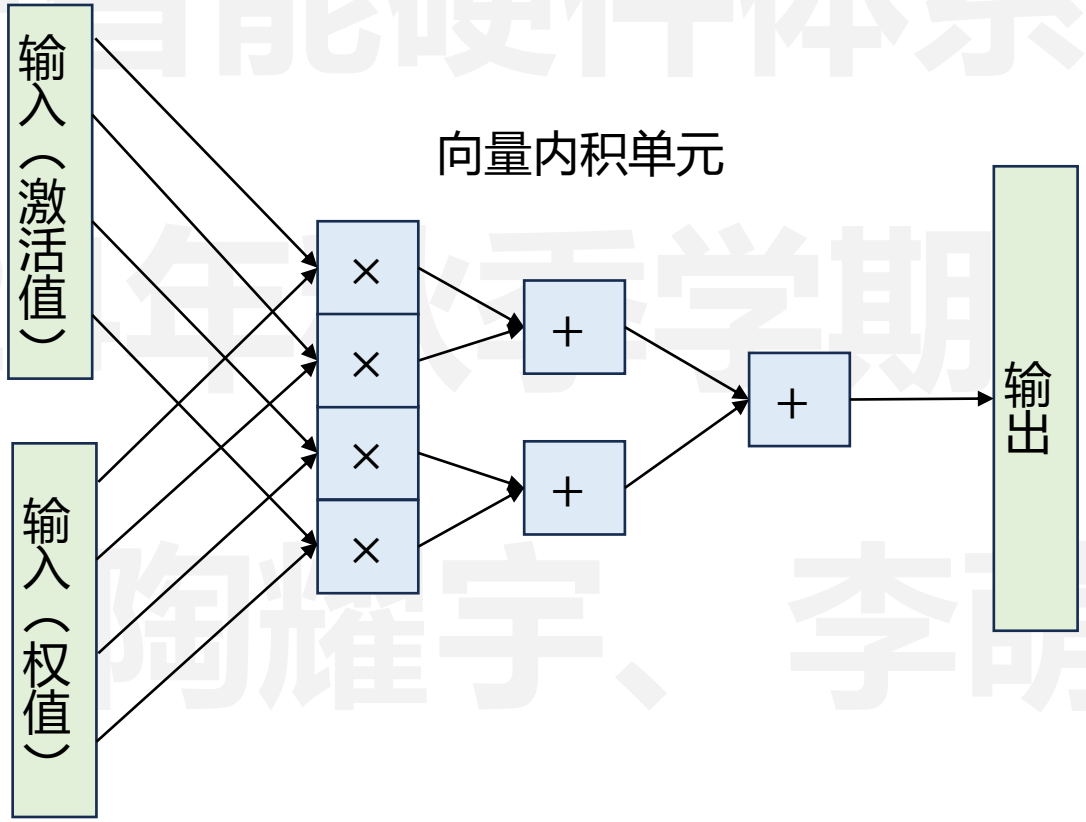


AI加速器架构基础

- **矩阵计算单元：通过内积计算，最大化数据复用**

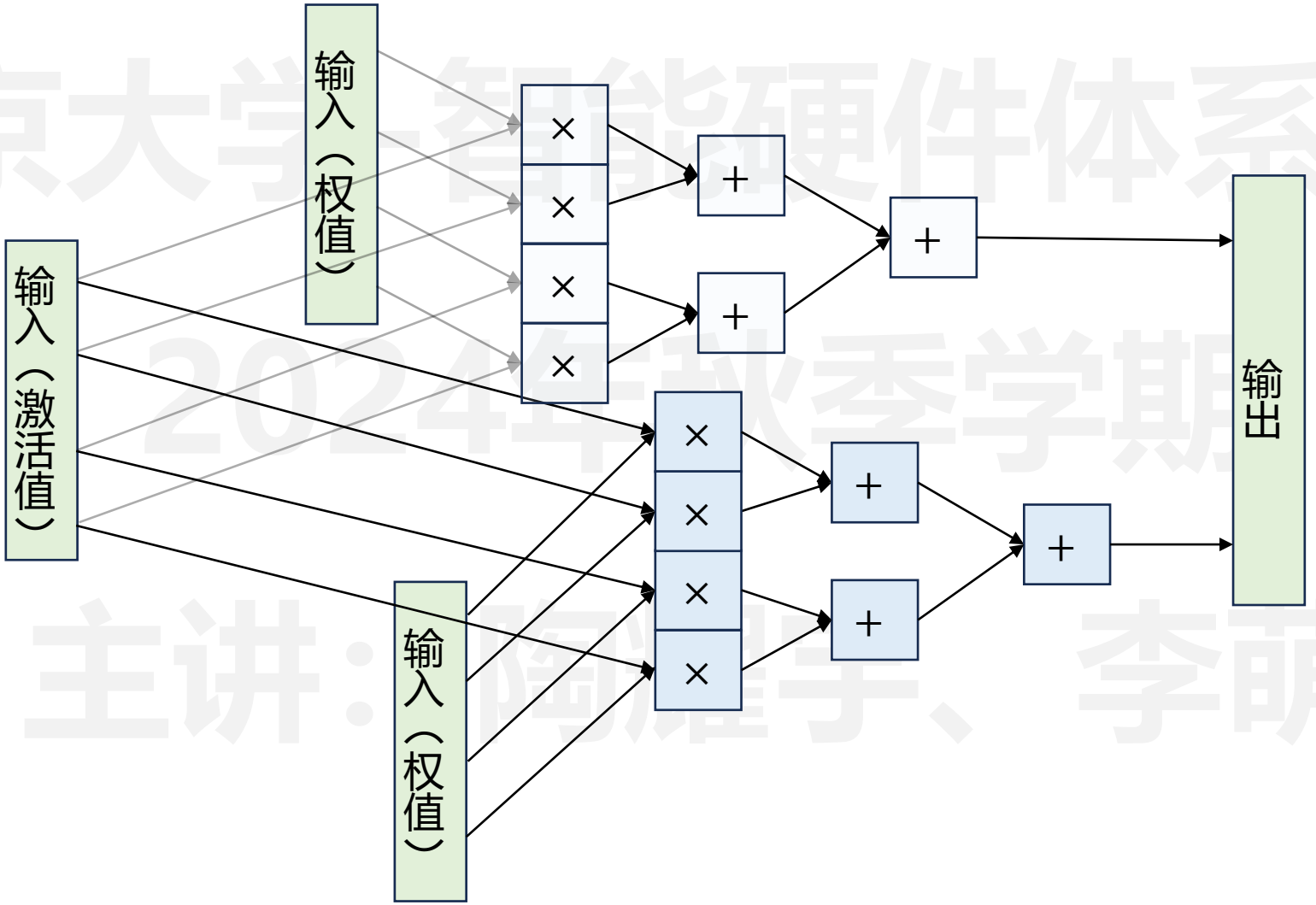


- 矩阵计算单元：通过内积计算，最大化数据复用



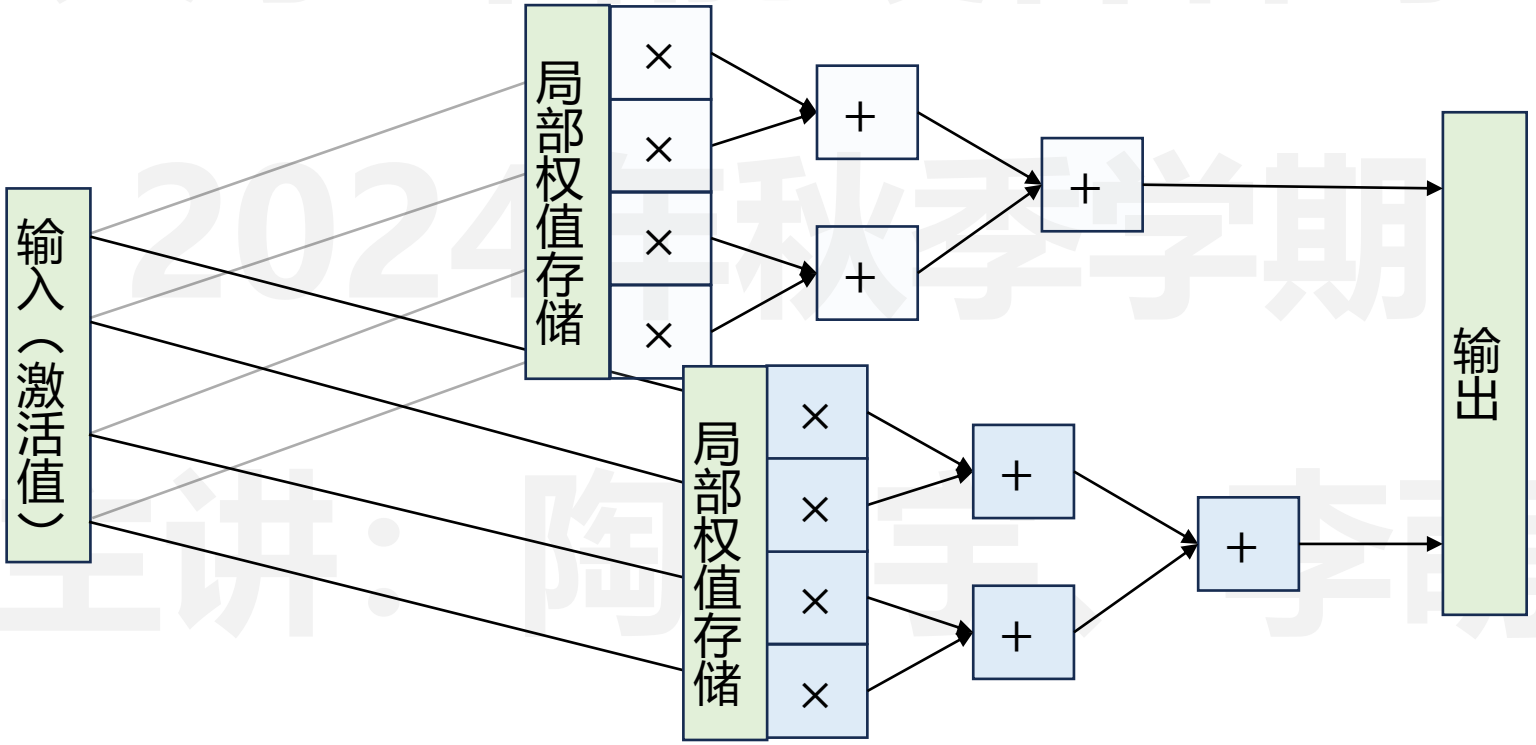
AI加速器架构基础

- 矩阵计算单元：通过内积计算，最大化数据复用



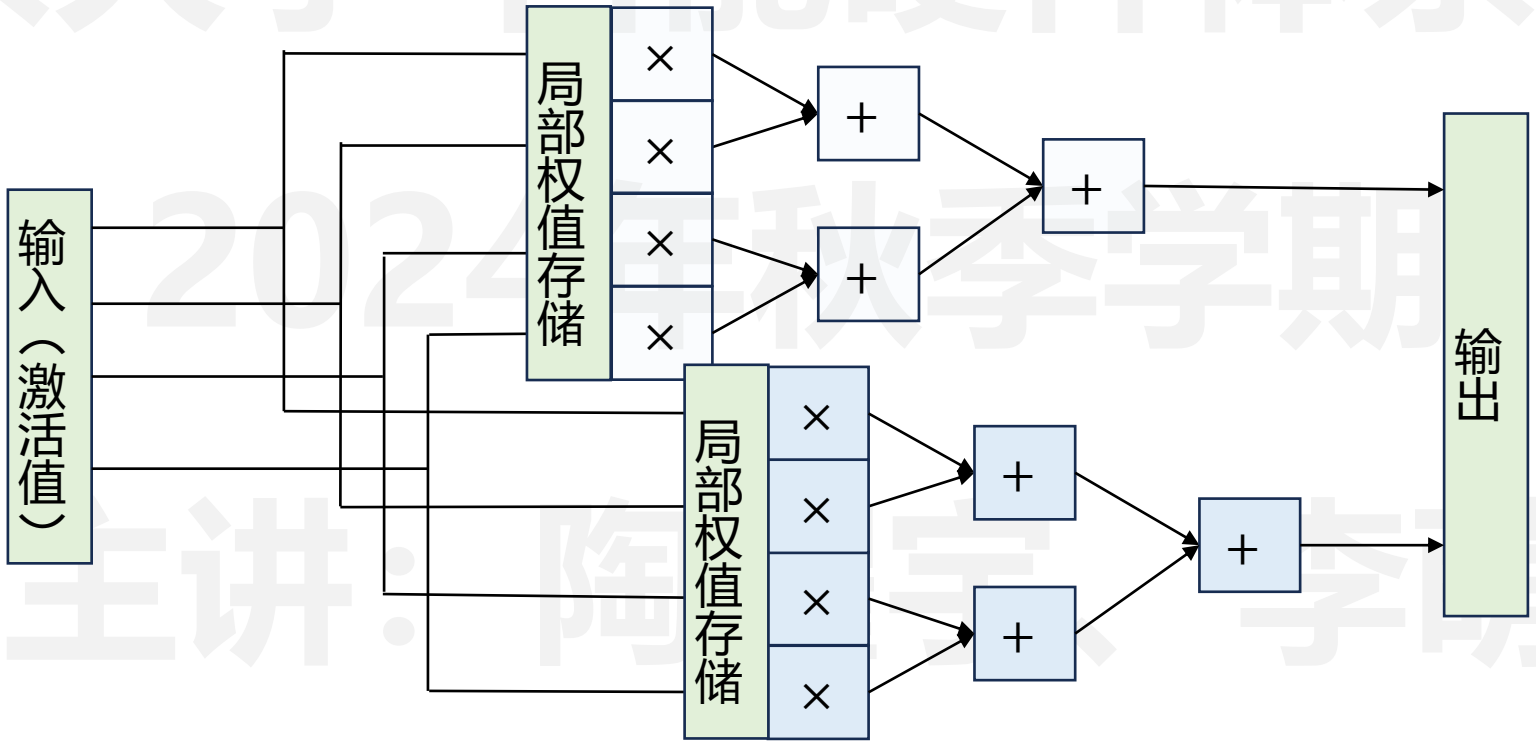
AI加速器架构基础

- **矩阵计算单元：通过内积计算，最大化数据复用**
- **权重采用局部小而快的存储器，直接存储在内积单元附近的电路中**

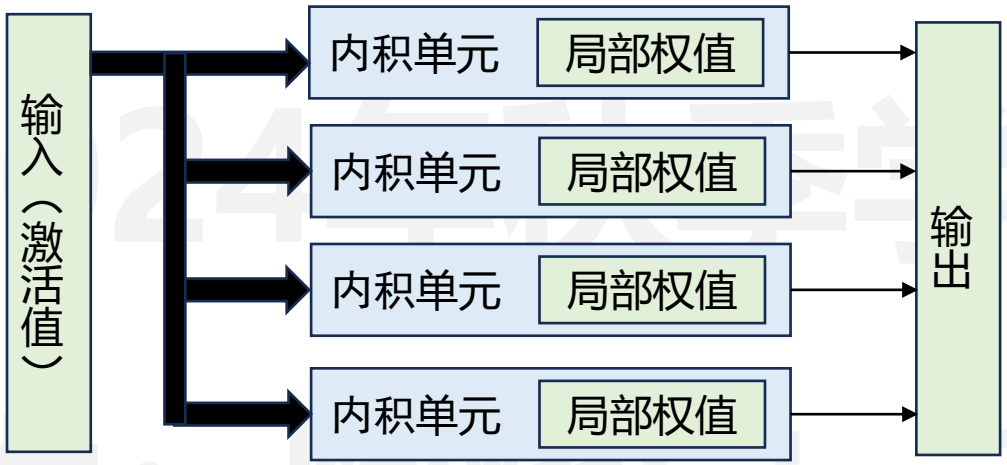


AI加速器架构基础

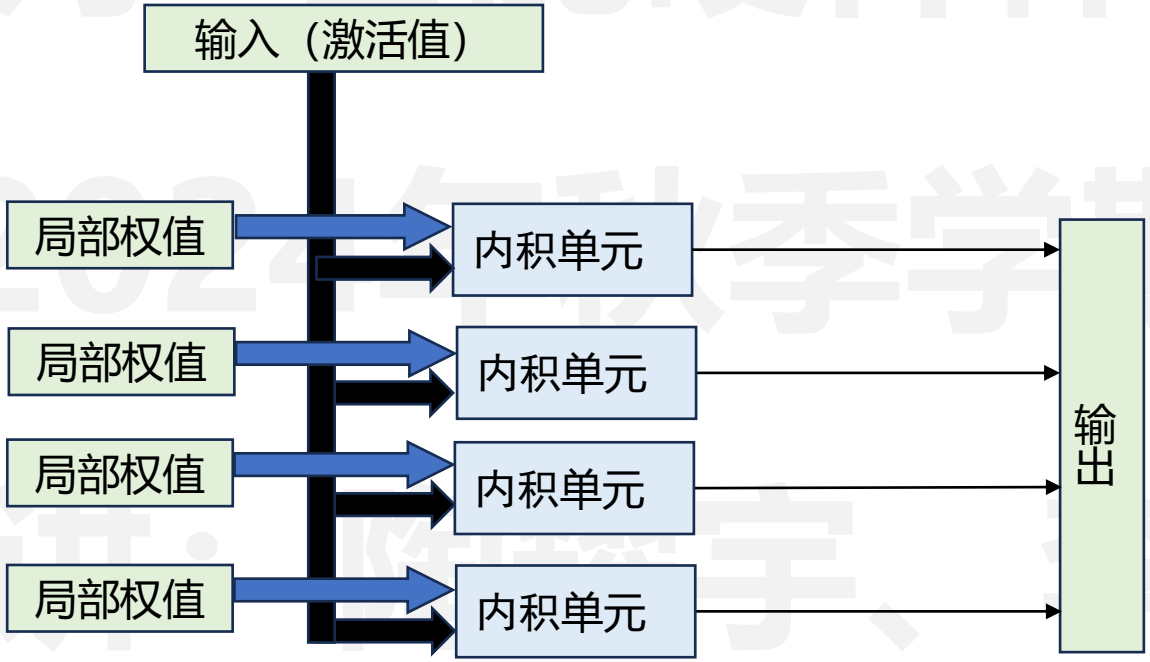
- **矩阵计算单元：通过内积计算，最大化数据复用**
- **所有内积单元共享激活值，采用广播的形式**



- **矩阵计算单元：通过内积计算，最大化数据复用**
- **所有内积单元共享激活值，采用广播的形式**

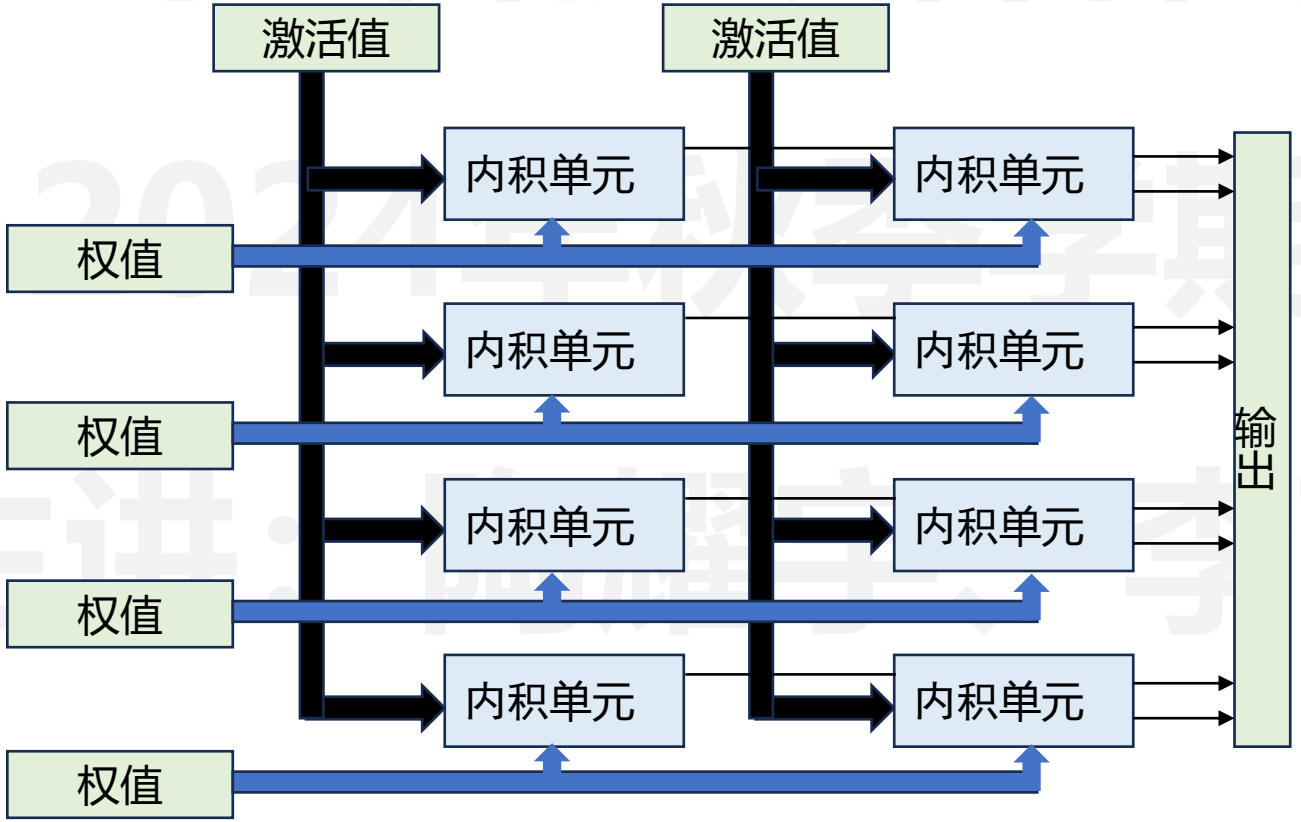


- **矩阵计算单元：通过内积计算，最大化数据复用**
- **如果我们把权值提取出来**



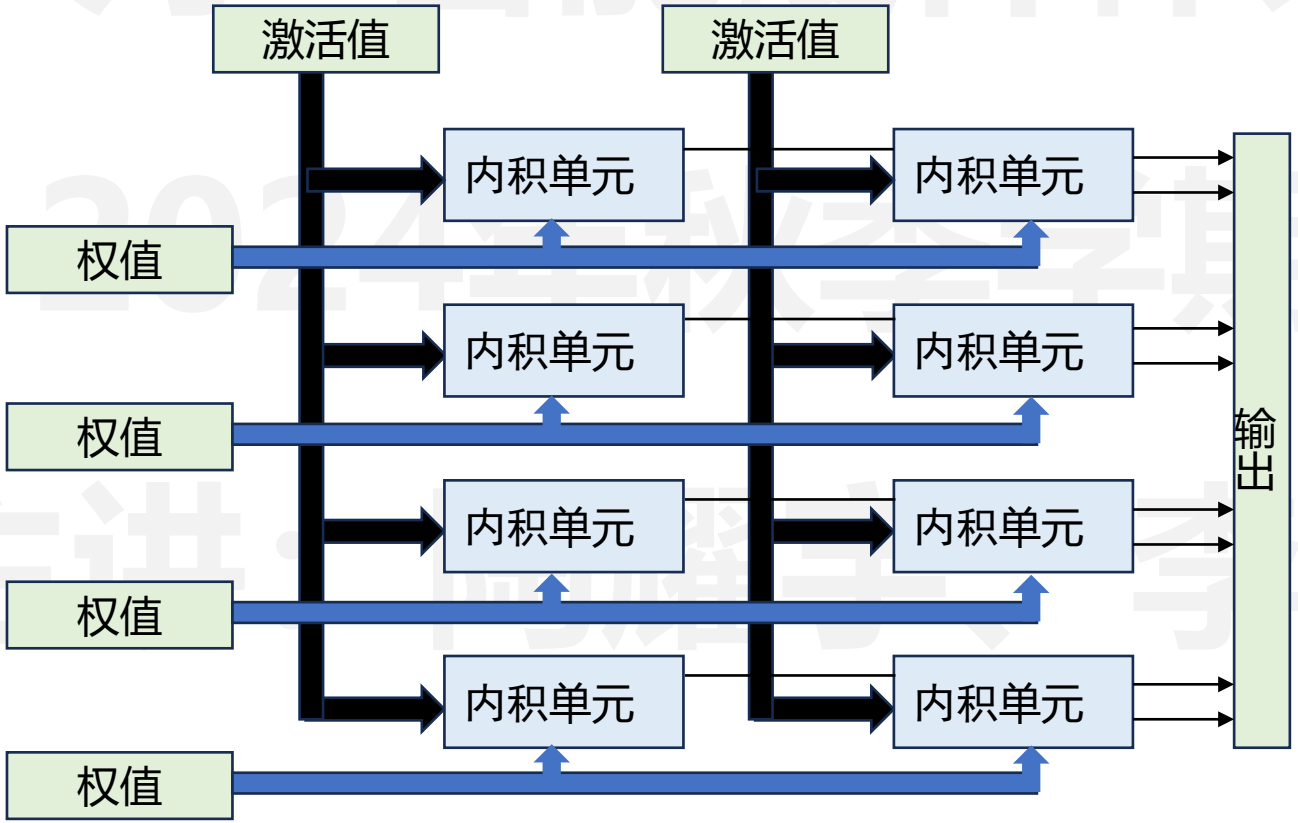
AI加速器架构基础

- **矩阵计算单元：通过内积计算，最大化数据复用**
- **如果我们把权值提取出来，并进一步扩大规模，能够显著增加数据复用**



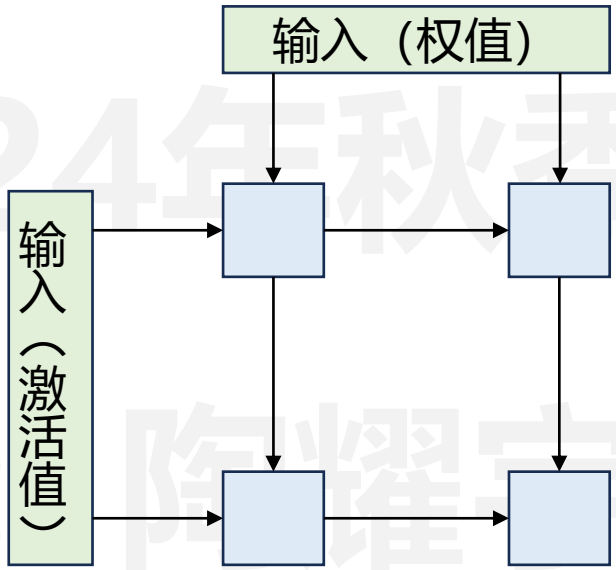
AI加速器架构基础

- 矩阵计算单元：通过内积计算，最大化数据复用
- 矩阵乘法单元在规模大时，能够实现很大的计算访存比
- 但是，面临连线复杂、距离远、扇出（Fanout）多等问题



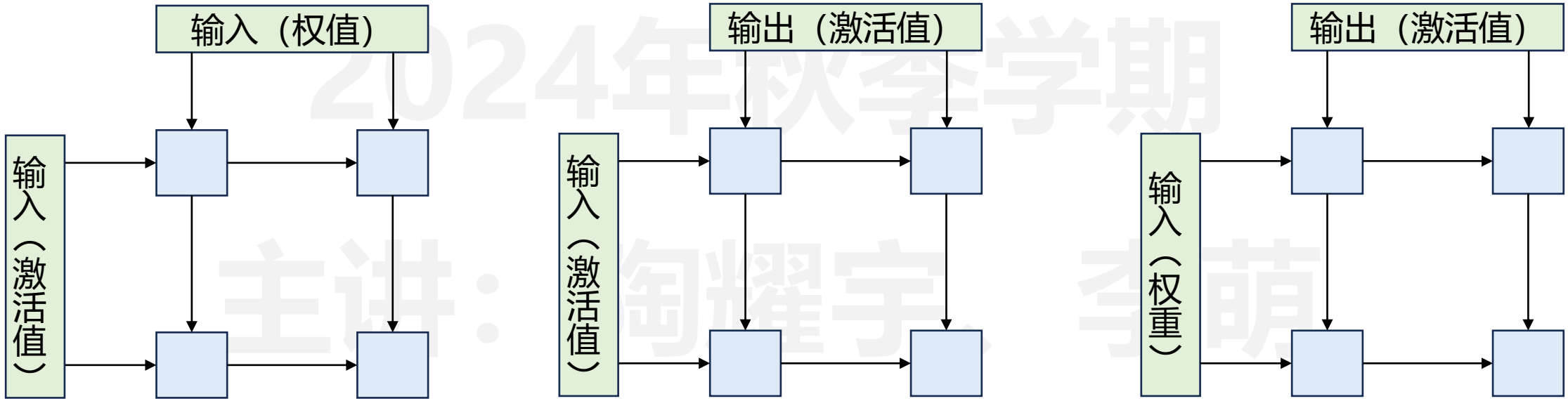
AI加速器架构基础

- 针对连线距离复杂、距离远、扇出多的问题，提出**脉动阵列**进行改进
- 只存在相邻计算单元之间的数据传输，连线短、扇出少
- 但是，**脉动阵列的延迟高（需要等待启动/排空）**，专用性更强，难以改造支持其他功能



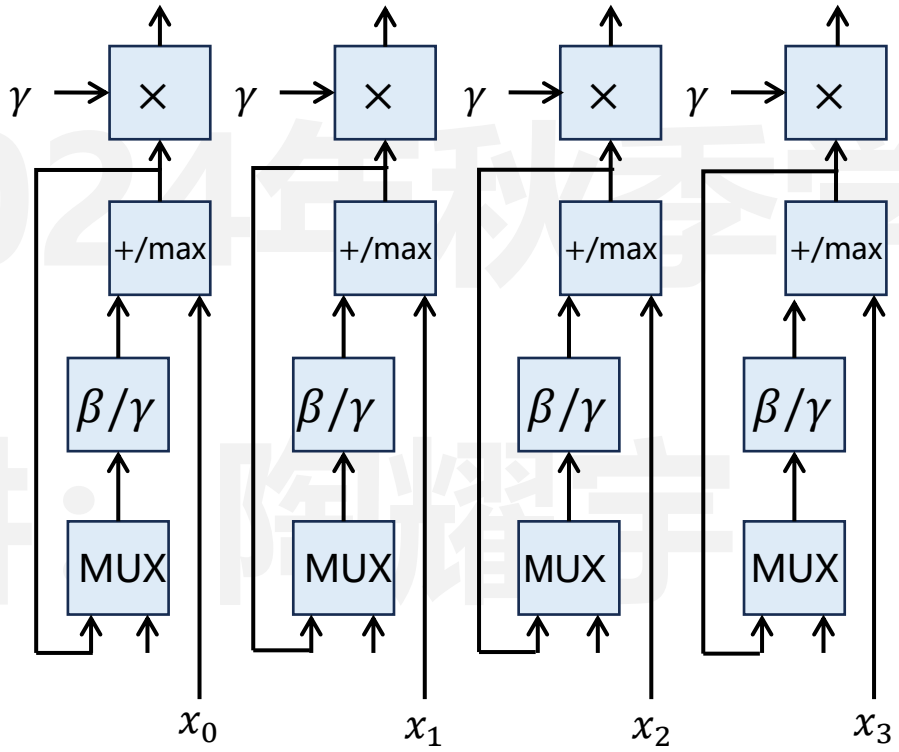
AI加速器架构基础——张量单元

- 矩阵乘法阵列设计核心思想：设计数据流架构，利用计算模式本身的数据复用，提升计算效率
- 针对数据流架构，核心是明确哪些数据是**移动**的，哪些数据是**固定不动**
- **典型数据流**：output stationary、weight stationary、input stationary
  - 不同的数据流，具有不同的神经网络算子适用性



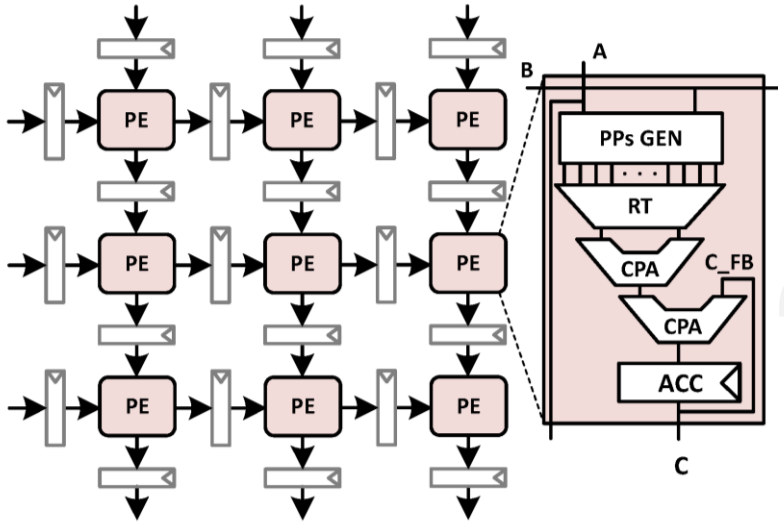
AI加速器架构基础——向量和标量单元

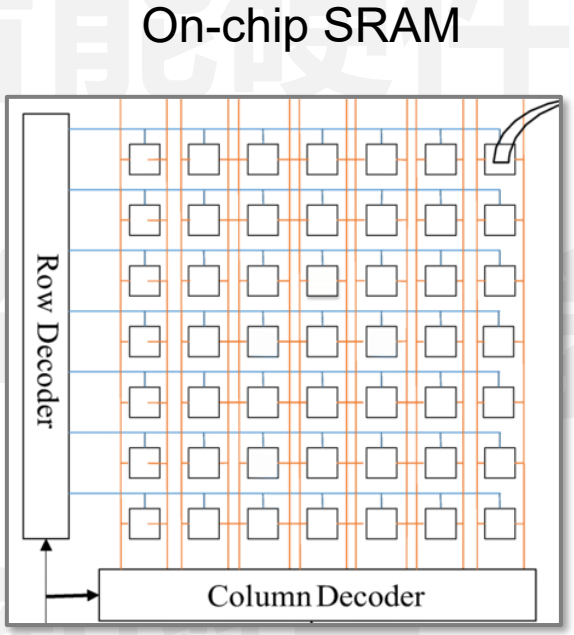
- 向量计算单元主要用于计算池化、归一化、ReLU、Sigmod、Softmax等特殊函数
- 尽管向量运算占神经网络总运算量并不大，但是，如果没有合适的加速硬件，仍然可能成为瓶颈
- 思考：向量计算单元和矩阵计算单元都有哪些区别？

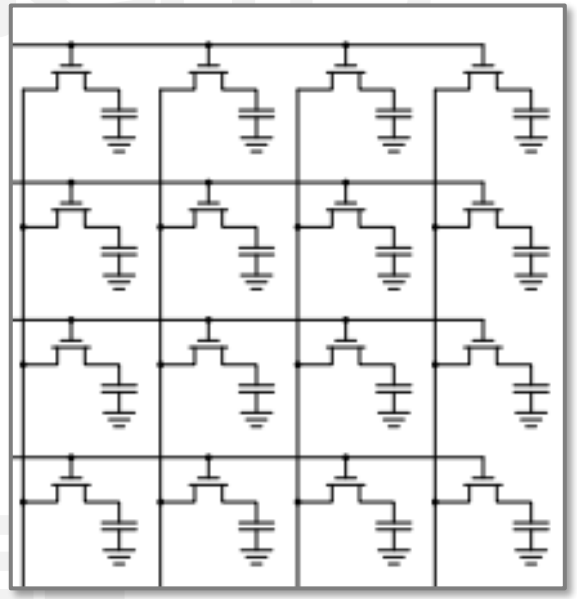


AI加速器架构基础

- AI加速器整体架构







计算 **访存** **通信**