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

第七讲：分支处理与缓存设计

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2次作业时间：3月20号 – 4月4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%）**

目 录

CONTENTS

01. 动态发射与乱序执行设计

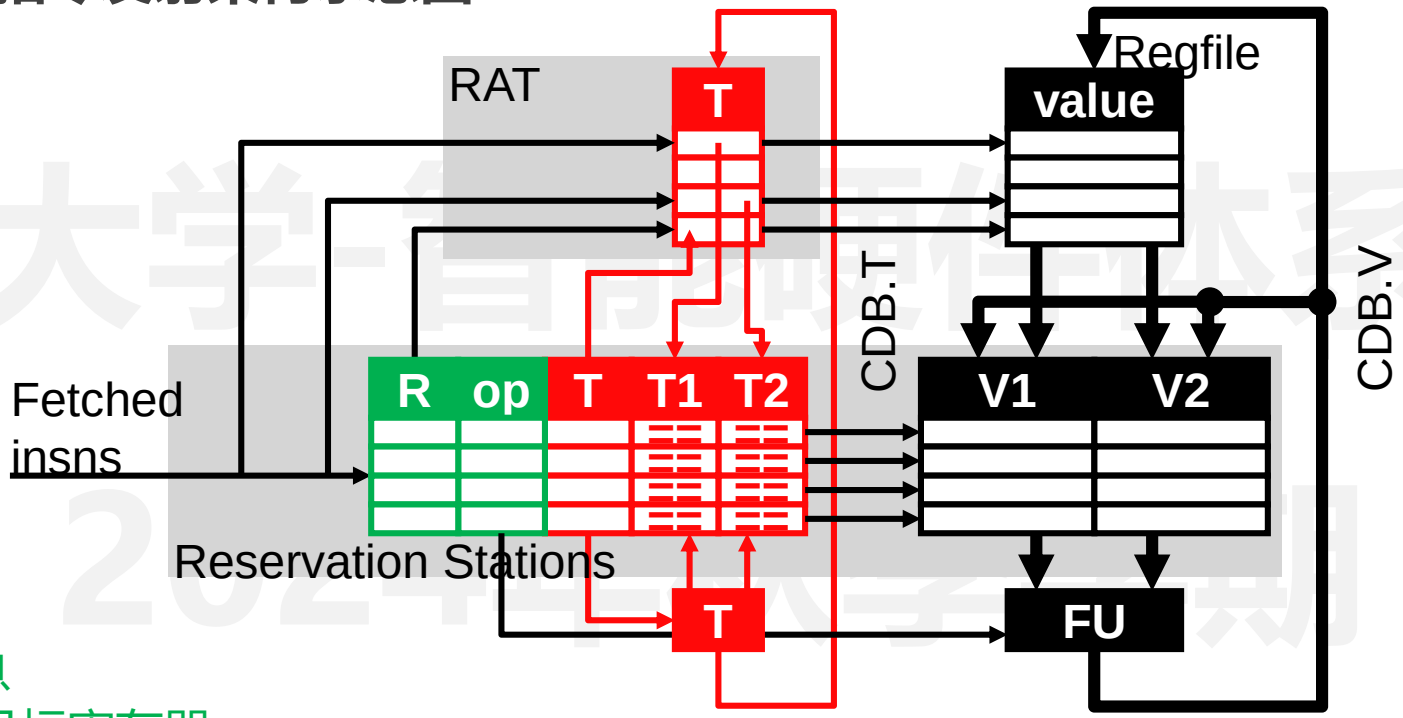
02. 分支处理机制与地址预测

03. 经典的MIPS架构实例分析

04. 多级缓存微架构与一致性

Tomasulo动态指令发射算法

- Tomasulo动态指令发射架构示意图



- RS:
  - 状态信息
    - R: 目标寄存器
    - op：操作数（加法等）
  - 标签
    - T1、T2：源操作数标签
  - 值
    - V1、V2：源操作数值

- 映射表（又称 RAT：寄存器别名表）
  - 将寄存器映射到标签
- 寄存器堆（又叫ARF：架构寄存器堆）
  - 如果 RS 中没有值，则保留寄存器的值

分支处理机制与地址预测

北京大学 PEKING UNIVERSITY

- Tomasulo动态发射的潜在问题

  - When can Tomasulo go wrong?
    - 分支
      - 如果分支在较新的指令（分支之后出现）完成后发生会怎样
    - Exceptions!!
      - 无法确定 RS 中指令的相对顺序
      - **我们需要一种预测分支结果的机制**
      - **我们需要一种机制来确保按顺序完成**

分支预测

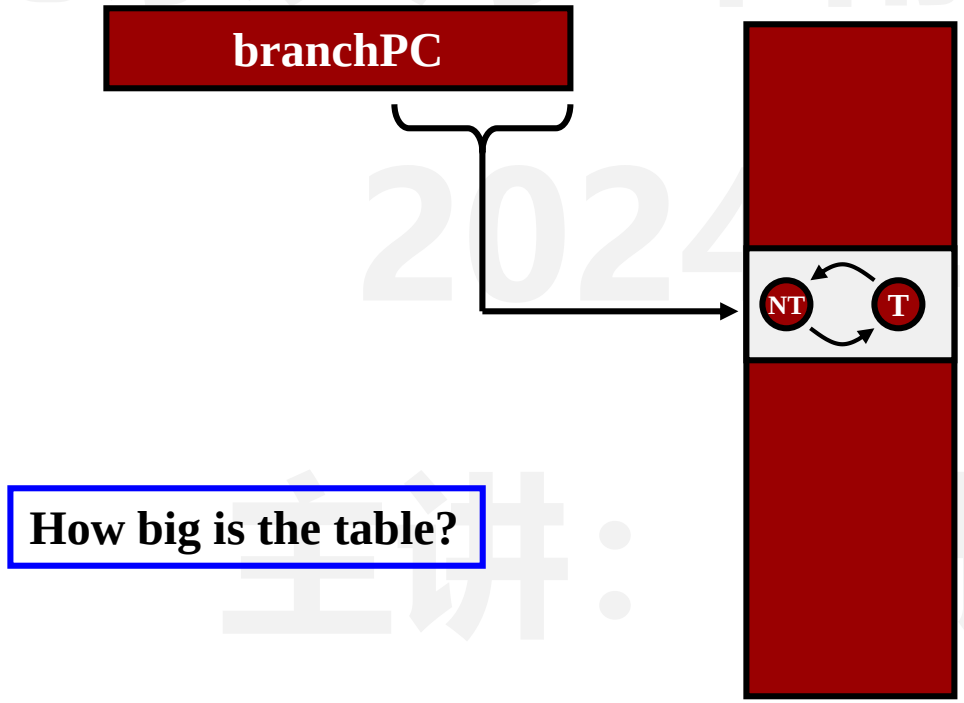
- 包括方向预测、地址预测与恢复机制

- 方向预测器
  - 对于条件分支
    - **预测分支是否会被执行**
  - 例子：
    - 总是分支跳跃；总是向后分支跳跃
- 地址预测器
  - 预测目标地址（预计需要时使用）
  - 示例：
    - **BTB；Return Address Stack; Precomputed Branch**
- 恢复逻辑

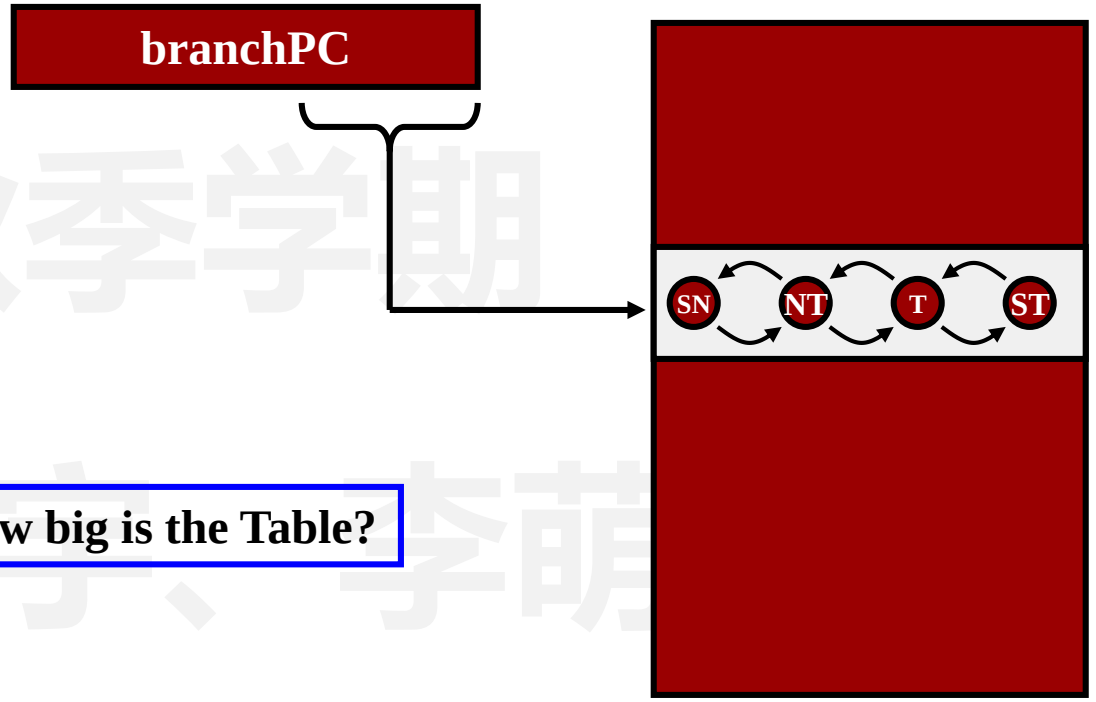
分支预测

- 方向预测 – 基于历史的简单状态机FSM

- 1 位历史记录（方向预测器）
  - 记住分支的最后方向



- 2 位历史记录（方向预测器）



分支预测

- 方向预测 – 基于历史的简单状态机FSM

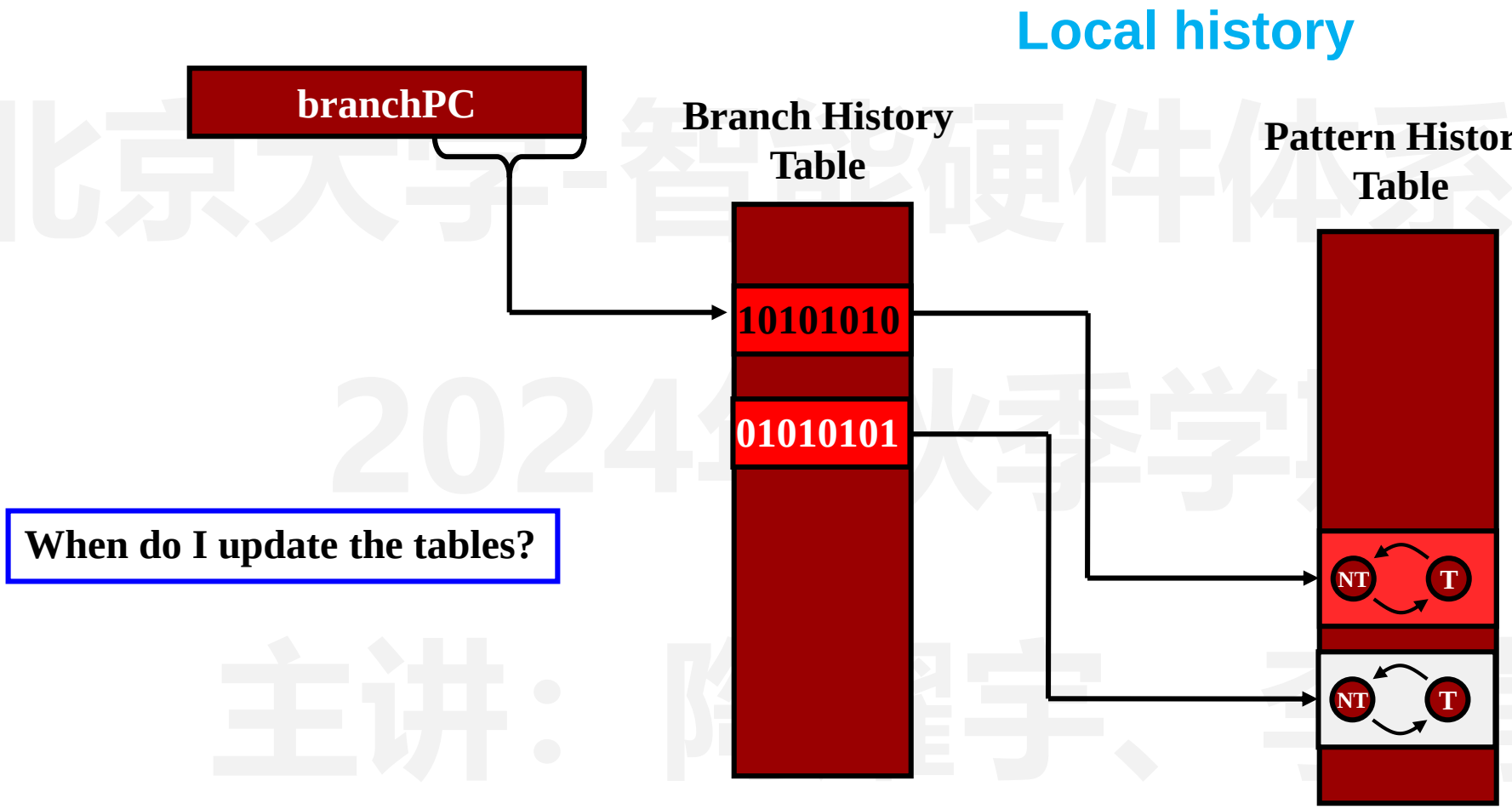
- **约 80% 的分支要么大量被采用，要么大量未被采用**
- 对于剩下的 20%，我们需要查看分支具体模式，确定是否可以使用更复杂的**预测器**进行预测
- Example: **gcc** has a branch that flips each time

T(1) NT(0) 1010101010101010101010101010101010101010

Using History Patterns

分支预测

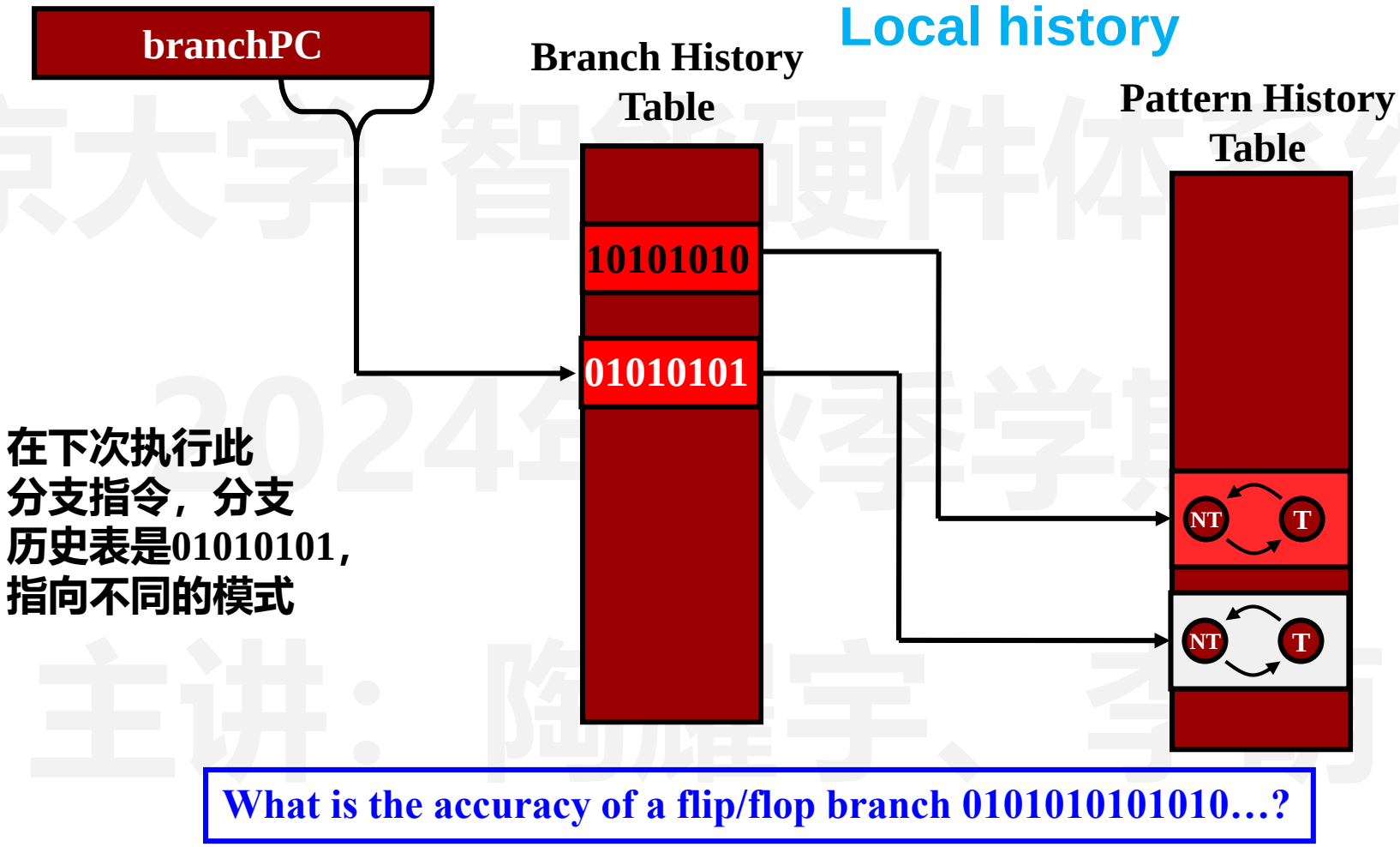
- 方向预测 – 基于历史的简单状态机FSM



Using History Patterns

分支预测

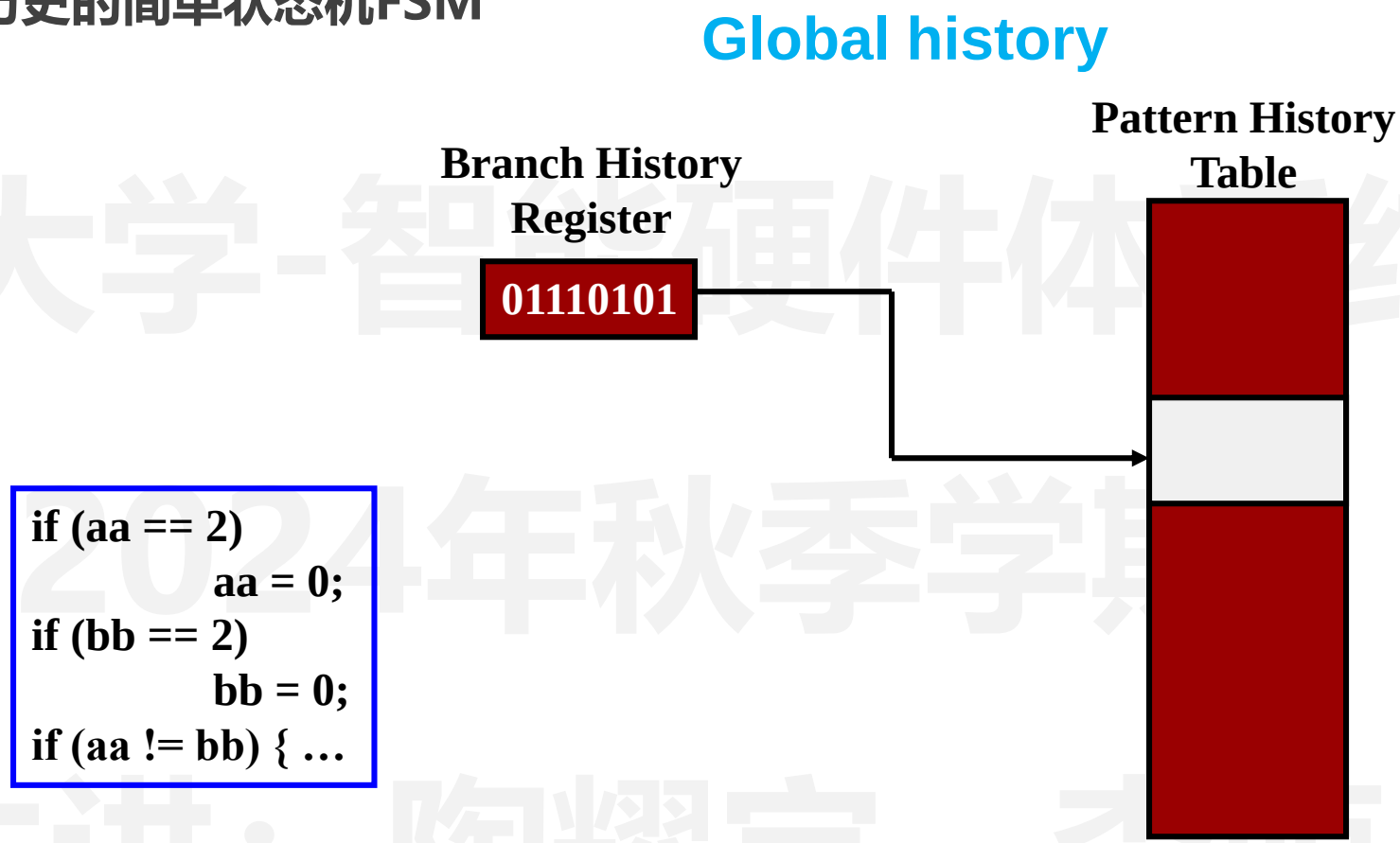
- 方向预测 – 基于历史的简单状态机FSM



Using History Patterns

分支预测

- 方向预测 – 基于历史的简单状态机FSM

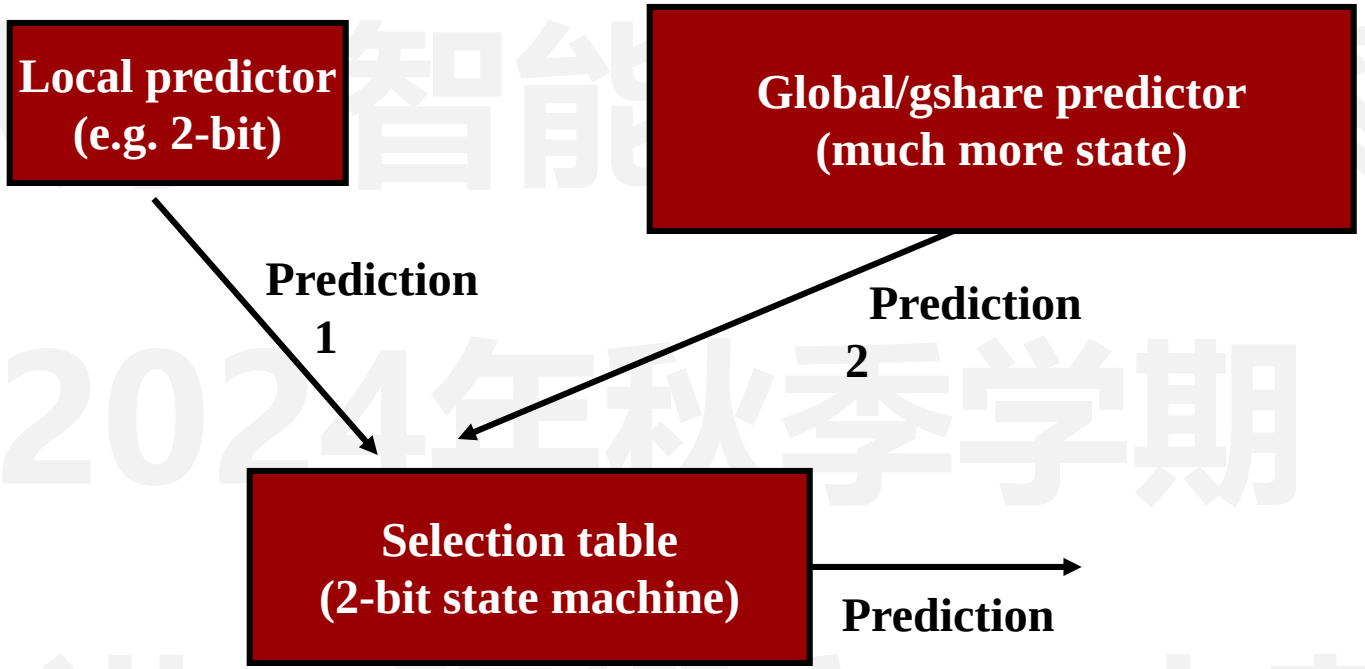


Using History Patterns

分支预测

- 方向预测 – 基于历史的简单状态机FSM

Hybrid predictors



如何选择使用哪个预测因子?
如何更新各种预测器/选择器

Using History Patterns

分支预测

- 地址预测 – Branch Target Buffer

- **按当前 PC 索引的 BTB**
  - 如果条目位于 BTB 中，则下一步获取目标地址
- 通常设置关联（速度较慢）
- 通常由分支预测器限定

Branch PC	Target address
0x05360AF0 →	0x05360000
…	…
…	…
…	…
…	…
…	…

分支流水线处理机制

- 对于顺序多级流水线比较简单，对乱序执行需要额外的机制

顺序多级流水线

- Squash 并使用正确的地址重新启动获取
  - 只需确保尚未有任何指令使用其状态
- 在我们的 5 级流水线中，状态仅在 MEM（存储）和 WB（寄存器写回）期间提交完成

Tamosulo

- 恢复似乎很难
  - **如果在我们发现分支错误之前分支后的指令已经完成，该怎么办?**
    - 这是有可能发生的。想象一下

      R1=MEM[R2+0]

      BEQ R1, R3 DONE ← Predicted not taken

      R4=R5+R6
- 因此，我们不能对分支进行猜测，也不能让任何东西通过分支
  - 这实际上是同一件事。
  - **分支成为顺序执行指令。**
    - 请注意，一旦分支解决，就可以在分支之前和之后执行一些操作。

目 录

CONTENTS

01. 动态发射与乱序执行设计

02. 分支处理机制与地址预测

03. 经典的MIPS架构实例分析

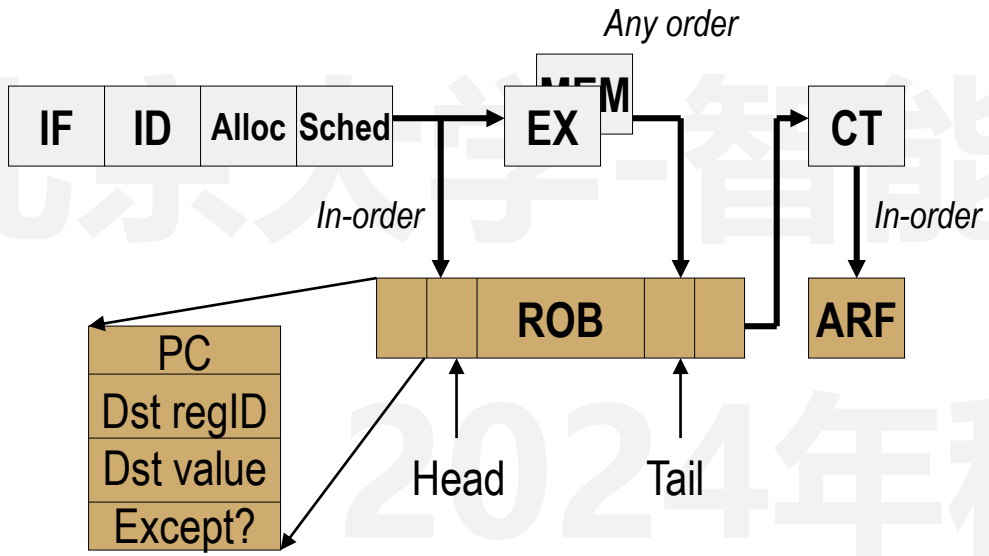
04. 多级缓存微架构与一致性

MIPS R10K：超标量+动态指令发射

- Adding a Reorder Buffer, aka ROB
  - 为什么需要 Reorder Buffer
    - ROB 是一个*顺序*放置指令的队列.
    - **分支指令按顺序完成**
    - **其他指令仍然乱序*执行***
    - 还是用RS
      - **同时向RS和ROB发出指令**
      - **重命名为 ROB 条目，而不是 RS。**
      - 什么时候*执行*完成指令离开 RS
    - 仅当程序顺序中该分支指令之前的所有指令都完成后，该分支指令才会退出

MIPS R10K：超标量+动态指令发射

- Adding a Reorder Buffer, aka ROB

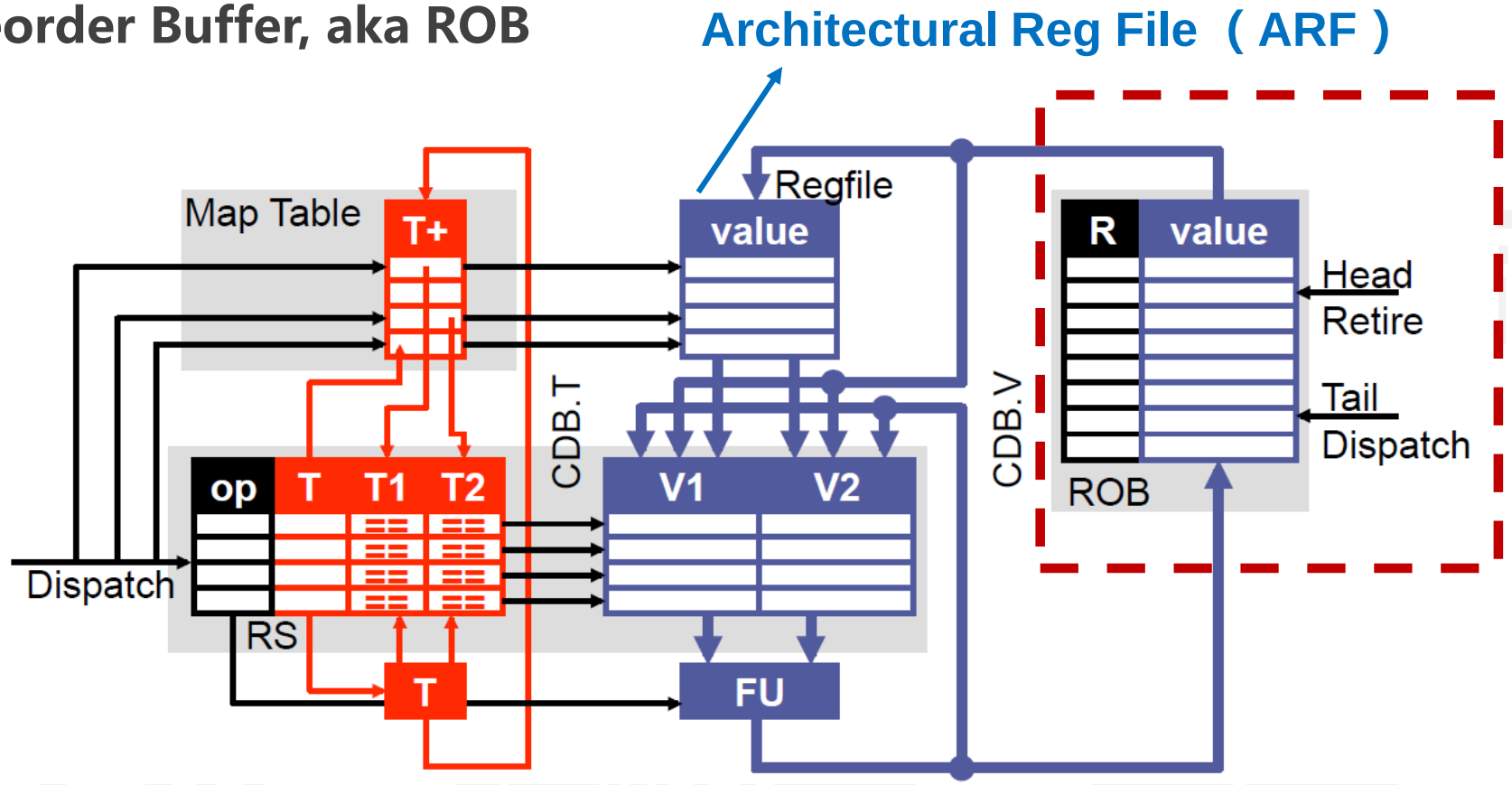


- Reorder Buffer (ROB)
  - spec 状态的循环队列
  - 同一个寄存器可能存在多个定义

- @ **Alloc**
  - 在ROB尾部存下进入的指令
- @ **Sched**
  - 获取输入(ROB T-to-H then ARF)
  - 等到所有输入准备就绪
- @ **WB**
  - 将结果写入 ROB/RS
  - 表示结果已准备好
- @ **CT**
  - Wait until inst @ Head is done
  - 如果发生错误，启动处理程序
  - 否则，将结果写入 ARF
  - 从 ROB 中释放存储空间

MIPS：超标量+动态指令发射

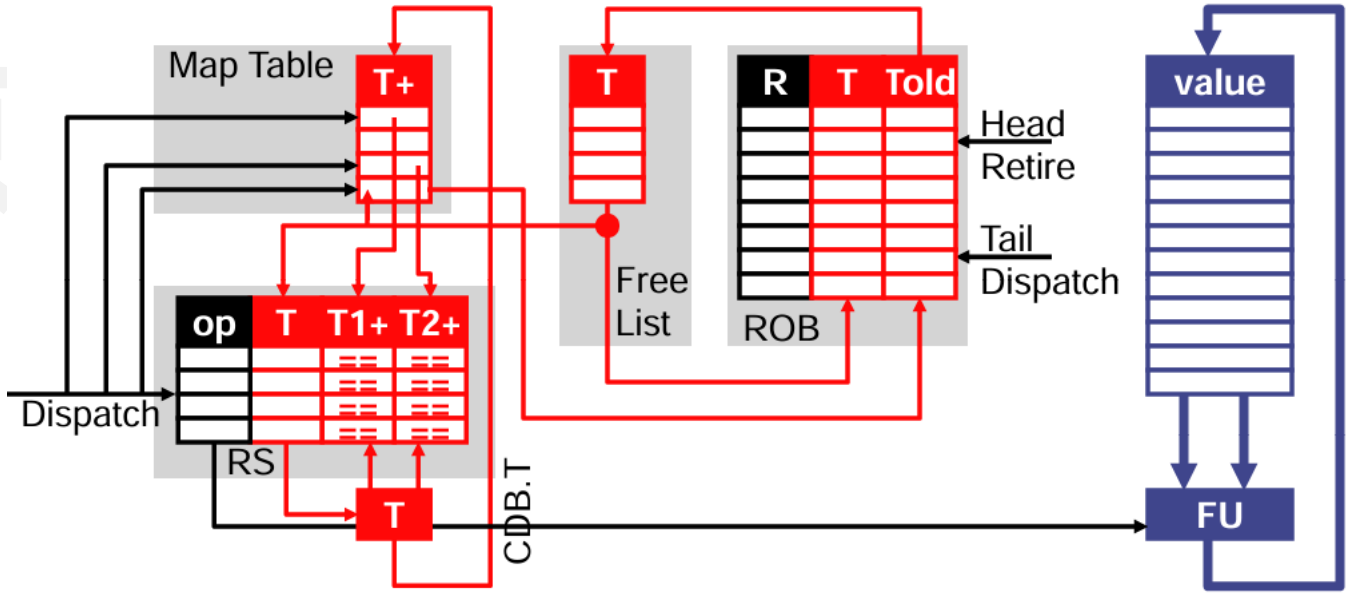
- Adding a Reorder Buffer, aka ROB



- **Simple Tomasulo+ROB**
  - 数据移动(regfile/ROB→RS→ROB→regfile)
  - 多输入多路复用器长总线使走线复杂化并且使时钟速度变慢

MIPS：超标量+动态指令发射

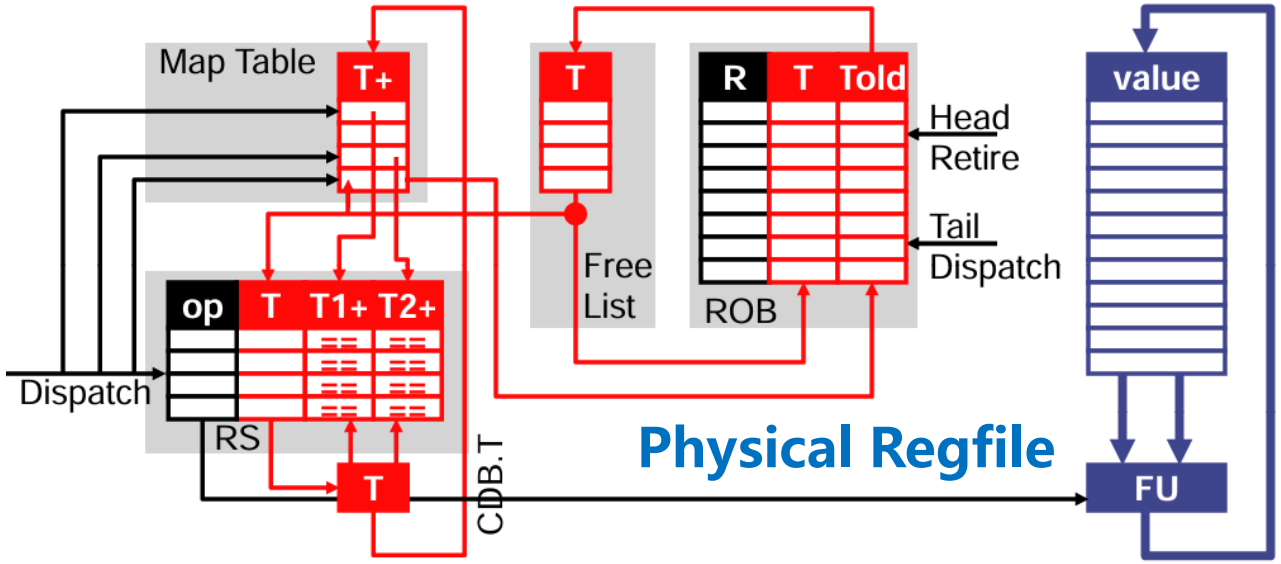
- MIPS：An alternative implementation



- **一个大的物理寄存器堆保存所有数据-无需复制**
  - + 靠近 FU 的寄存器文件 → 小型快速数据路径 ROB
  - ROB and RS “on the side” used only for control and tags

MIPS：超标量+动态指令发射

- MIPS：An alternative implementation



- **架构寄存器文件？ Gone**
- **物理寄存器文件保存所有值**
  - #物理寄存器 = #架构寄存器 + #ROB entries
  - 将架构寄存器映射到物理寄存器
  - 消除数据冲突（物理寄存器取代 RS 副本）

- **根本上改变 map table/RAT**
  - 映射不能为 0（没有架构寄存器文件）
- **空闲列表跟踪未分配的物理寄存器**
  - ROB 负责将物理寄存器返回到空闲列表
- 从概念上讲，这是“真正的寄存器重命名”，因为没有寄存器值搬运

MIPS：超标量+动态指令发射

- MIPS：An alternative implementation

Parameters

- Names: r1 , r2 , r3
- Locations: p1 , p2 , p3 , p4 , p5 , p6 , p7
- Original mapping: r1→p1, r2→p2, r3→p3, p4–p7 are “free”

MapTable

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p6	p2	p5

FreeList

p4 , p5 , p6 , p7
p5 , p6 , p7
p6 , p7
p7

Raw insns	Renamed insns
add r2,r3,r1	add p2,p3,p4
sub r2,r1,r3	sub p2,p4,p5
mul r2,r3,r1	mul p2,p5,p6
div r1,r3,r2	div p4,p5,p7

- **问题：div之后的指令该如何进行重命名**
  - 物理寄存器都被占用了
  - **现实的问题：物理寄存器应当在什么时候释放**

MIPS：超标量+动态指令发射

- MIPS实例分析

New tags (again)

- Tomasulo+ROB：ROB# → MIPS：PR#

ROB

- T:对应指令逻辑输出的物理寄存器
- Told：之前映射到指令逻辑输出的物理寄存器

RS

- **T**, **T1**, **T2**: output, input physical registers

Map Table

- **T+**: PR# (never empty) + “ready” bit

Free List

- **T**: PR#

No values in ROB, RS, or on CDB

MIPS：超标量+动态指令发射

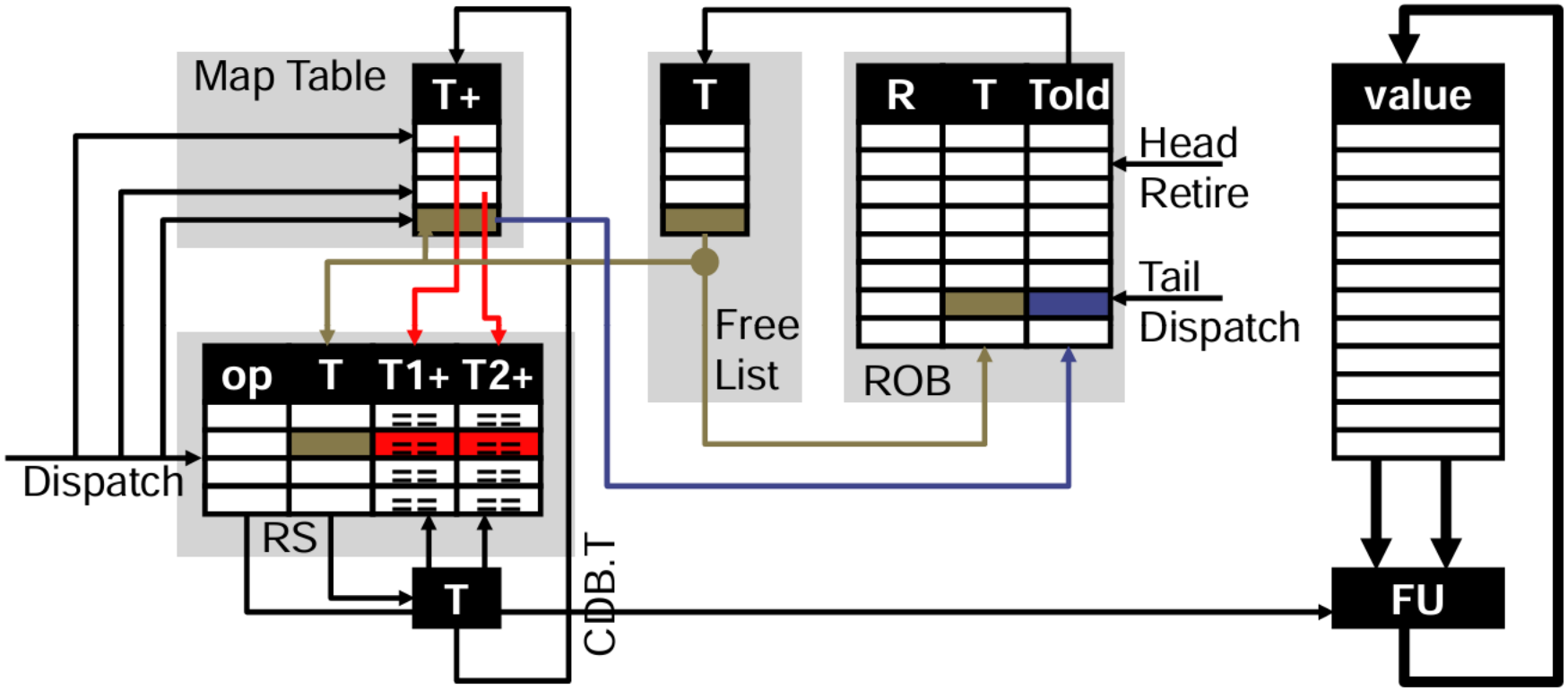
- MIPS实例分析

R10K pipeline structure: F, **D**, S, X, **C**, **R**

- **D (dispatch)**
  - Structural hazard (RS, ROB, LSQ, **physical registers**) ? stall
  - Allocate RS, ROB, LSQ entries and new physical register (T)
  - **Record previously mapped physical register (Told)**
- **C (complete)**
  - Write destination physical register
- **R (retire)**
  - ROB head not complete ? Stall
  - Handle any exceptions
  - Store write LSQ head to D$
  - Free ROB, LSQ entries
  - **Free previous physical register (Told)**

MIPS：超标量+动态指令发射

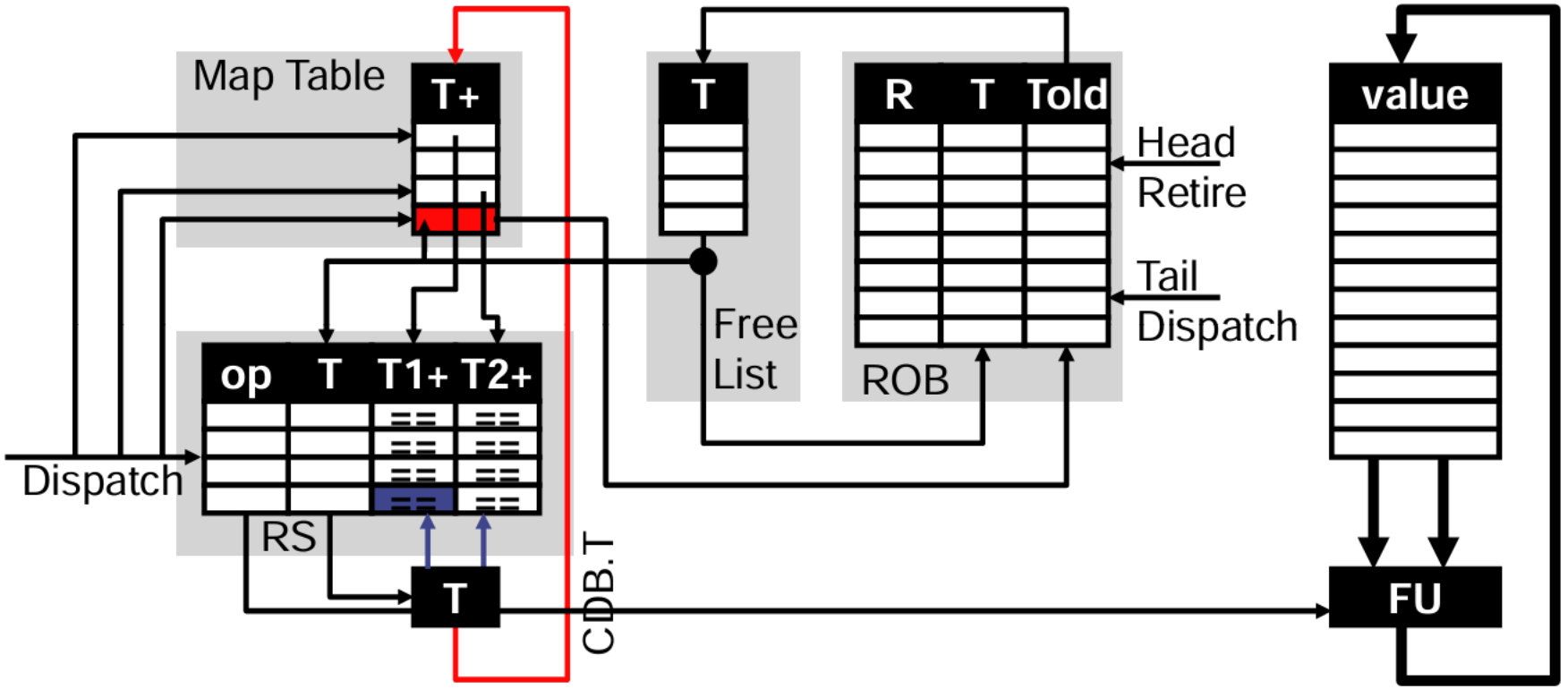
- MIPS R10K Dispatch步骤



- **T 读取输入寄存器对应preg（物理寄存器）标签，存在RS**
- Told：之前映射到指令逻辑输出的物理寄存器读取输出寄存器的preg标签，**存在ROB（Told）**
- 给输出寄存器分配新的preg（free list），存在RS，ROB，Map tabe

MIPS：超标量+动态指令发射

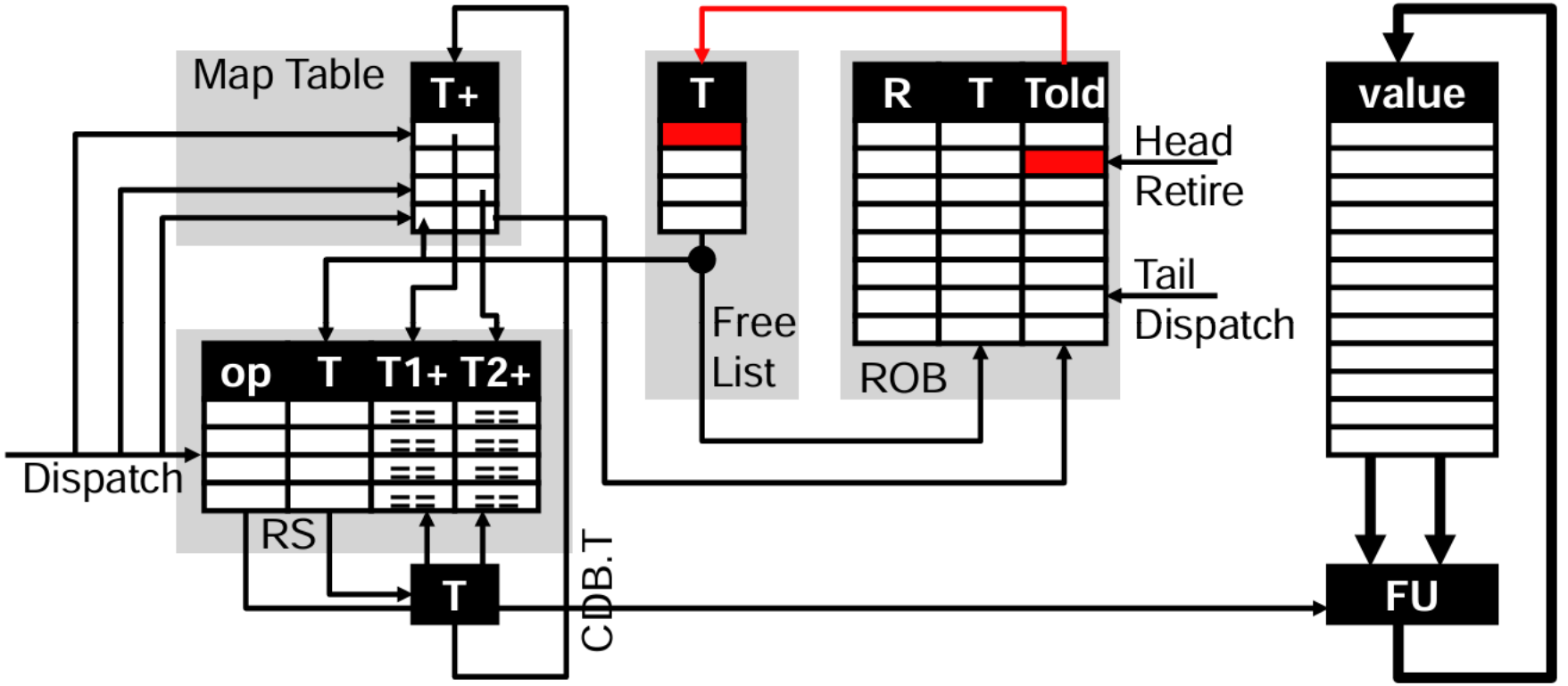
- MIPS R10K Complete步骤



- **在map table中设定指令输出寄存器的ready bit**
- 在RS中设定匹配输入标签的ready bits

MIPS：超标量+动态指令发射

- MIPS R10K Retire步骤



- Return Told of ROB head to free list

MIPS：超标量+动态指令发射

- MIPS实例分析

ROB							
ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1					
	2	mulf f0,f1,f2					
	3	stf f2,Z(r1)					
	4	addi r1,4,r1					
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#2+
f2	PR#3+
r1	PR#4+

CDB
T

Free List
PR#5,PR#6,
PR#7,PR#8

+：Ready bit

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	no				
3	ST	no				
4	FP1	no				
5	FP2	no				

Notice I: **任何地方都不存储寄存器values**

Notice II: **Map Table不为空**

MIPS：超标量+动态指令发射

- MIPS实例分析

MIPS:

Cycle 1

ROB							
ht	#	Insn	T	Told	S	X	C
ht	1	ldf X(r1),f1	PR#5	PR#2			
	2	mulf f0,f1,f2					
	3	stf f2,Z(r1)					
	4	addi r1,4,r1					
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#5
f2	PR#3+
r1	PR#4+

CDB
T

Free List
PR#5, PR#6, PR#7, PR#8

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	yes	ldf	PR#5		PR#4+
3	ST	no				
4	FP1	no				
5	FP2	no				

给f1分配新的preg位置（PR#5）

同时在ROB存储f1旧preg（PR#2）

MIPS：超标量+动态指令发射

- MIPS实例分析

MIPS:

Cycle 2

ROB

ht	#	Insn	T	Told	S	X	C
h	1	ldf X(r1),f1	PR#5	PR#2	c2		
t	2	mulf f0,f1,f2	PR#6	PR#3			
	3	stf f2,Z(r1)					
	4	addi r1,4,r1					
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table

Reg	T+
f0	PR#1+
f1	PR#5
f2	PR#6
r1	PR#4+

CDB

T

Free List

PR#6, PR#7, PR#8

Reservation Stations

#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	yes	ldf	PR#5		PR#4+
3	ST	no				
4	FP1	yes	mulf	PR#6	PR#1+	PR#5
5	FP2	no				

给f2分配新的preg位置（PR#6）

同时在ROB存储f2旧preg（PR#3）

MIPS：超标量+动态指令发射

- MIPS实例分析

MIPS:

Cycle 3

ROB

ht	#	Insn	T	Told	S	X	C
h	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	
	2	mulf f0,f1,f2	PR#6	PR#3			
t	3	stf f2,Z(r1)					
	4	addi r1,4,r1					
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table

Reg	T+
f0	PR#1+
f1	PR#5
f2	PR#6
r1	PR#4+

CDB

T

Free List

PR#7 , PR#8

Reservation Stations

#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	no				
3	ST	yes	stf		PR#6	PR#4+
4	FP1	yes	mulf	PR#6	PR#1+	PR#5
5	FP2	no				

Store指令不分配preg

Free 在X阶段即Free掉RS entry

MIPS：超标量+动态指令发射

- MIPS实例分析

MIPS:

Cycle 4

ROB

ht	#	Insn	T	Told	S	X	C
h	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
	2	mulf f0,f1,f2	PR#6	PR#3	c4		
	3	stf f2,Z(r1)					
t	4	addi r1,4,r1	PR#7	PR#4			
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table

Reg	T+
f0	PR#1+
f1	PR#5+
f2	PR#6
r1	PR#7

CDB

T
PR#5

Free List

PR#7 , PR#8

Reservation Stations

#	FU	busy	op	T	T1	T2
1	ALU	yes	addi	PR#7	PR#4+	
2	LD	no				
3	ST	yes	stf		PR#6	PR#4+
4	FP1	yes	mulf	PR#6	PR#1+	PR#5+
5	FP2	no				

ldf completes
set MapTable ready bit

Match PR#5 tag from CDB & issue

MIPS：超标量+动态指令发射

- MIPS实例分析

MIPS:

Cycle 5

ROB							
ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
h	2	mulf f0,f1,f2	PR#6	PR#3	c4	c5	
	3	stf f2,Z(r1)					
	4	addi r1,4,r1	PR#7	PR#4	c5		
t	5	ldf X(r1),f1	PR#8	PR#5			
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#8
f2	PR#6
r1	PR#7

CDB
T

Free List
PR#8, PR#2

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	yes	addi	PR#7	PR#4+	
2	LD	yes	ldf	PR#8		PR#7
3	ST	yes	stf		PR#6	PR#4+
4	FP1	no				
5	FP2	no				

ldf retires
Return PR#2 to free list

Free 在X阶段即Free掉RS entry

MIPS：超标量+动态指令发射

- MIPS实例分析

- Problem with R10K design? Precise state is more difficult

-- 物理寄存器乱序写入

- 是可以的，没有架构寄存器文件
- **我们可以“free”被写入的寄存器并“restore”旧的**
- 通过控制Map Table和Free List来完成，而不是寄存器文件

- 两种恢复Map Table 和Free List的方式
- **方式1：通过用ROB中的T，Told串行恢复（速度慢但简单）**
- **方式2：从一些检查点单周期恢复（速度快，但检查点是昂贵的）**
- 现代处理器的折中方案：让普遍情况快速运行
  - 需要频繁rollback的跳转使用检查点
  - 较少rollback的Page-fault和interrupt进行串行恢复

MIPS：超标量+动态指令发射

- MIPS实例分析

Replace with a taken branch

MIPS:

Cycle 5

Precise

State

ROB

ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
h	2	jmp f0 f1 f2	PR#6	PR#3	c4	c5	
	3	stf f2,Z(r1)					
	4	addi r1,4,r1	PR#7	PR#4	c5		
t	5	ldf X(r1),f1	PR#8	PR#5			
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table

Reg	T+
f0	PR#1+
f1	PR#8
f2	PR#6
r1	PR#7

CDB

T

Free List

PR#8, PR#2

Reservation Stations

#	FU	busy	op	T	T1	T2
1	ALU	yes	addi	PR#7	PR#4+	
2	LD	yes	ldf	PR#8		PR#7
3	ST	yes	stf		PR#6	PR#4+
4	FP1	no				
5	FP2	no				

如果指令2采取跳转，则通过rollback撤销指令3-5

MIPS：超标量+动态指令发射

- MIPS实例分析

MIPS:

Cycle 6

ROB							
ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
h	2	jmp f0 f1 f2	PR#6	PR#3	c4	c5	
	3	stf f2,Z(r1)					
t	4	addi r1,4,r1	PR#7	PR#4	c5		
	5	ldf X(r1),f1	~~PR#8~~	~~PR#5~~			
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#5+
f2	PR#6
r1	PR#7

CDB
T

Free List
PR#2 , PR#8

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	yes	addi	PR#7	PR#4+	
2	LD	no				
3	ST	yes	stf		PR#6	PR#4+
4	FP1	no				
5	FP2	no				

撤销ldf（ROB#5）

1.释放RS

2.释放T（PR#8），返回Freelist

3.将MT[f1]重新存储到Told（PR#5）

4.释放ROB#5

指令可以rollback执行

MIPS：超标量+动态指令发射

- MIPS实例分析

MIPS:

Cycle 7

ROB

ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
h	2	jmp f0 f1 f2	PR#6	PR#3	c4	c5	
t	3	stf f2,Z(r1)					
	4	addi r1,4,r1	PR#7	PR#4	c5		
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table

Reg	T+
f0	PR#1+
f1	PR#5+
f2	PR#6
r1	PR#4+

CDB

T

Free List

PR#2, PR#8, PR#7

Reservation Stations

#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	no				
3	ST	yes	stf		PR#6	PR#4+
4	FP1	no				
5	FP2	no				

撤销ldf（ROB#4）

1.释放RS

2.释放T（PR#7），返回Freelist

3.将MT[r1]重新存储到Told（PR#4）

4.释放ROB#4

MIPS：超标量+动态指令发射

- MIPS实例分析

MIPS:

Cycle 8

ROB							
ht	#	Insn	T	Told	S	X	C
	1	ldf X(r1),f1	PR#5	PR#2	c2	c3	c4
ht	2	jmp f0 f1 f2	PR#6	PR#3	c4	c5	
	3	stf f2,Z(r1)					
	4	addi r1,4,r1					
	5	ldf X(r1),f1					
	6	mulf f0,f1,f2					
	7	stf f2,Z(r1)					

Map Table	
Reg	T+
f0	PR#1+
f1	PR#5+
f2	PR#6
r1	PR#4+

CDB
T

Free List
PR#2,PR#8, PR#7

Reservation Stations						
#	FU	busy	op	T	T1	T2
1	ALU	no				
2	LD	no				
3	ST	no				
4	FP1	no				
5	FP2	no				

撤销ldf（ROB#3）

1.释放RS

2.释放ROB#3

3.没有寄存器需要恢复/释放

4.D$的写入如何撤销?

目 录

CONTENTS

01. 动态发射与乱序执行设计

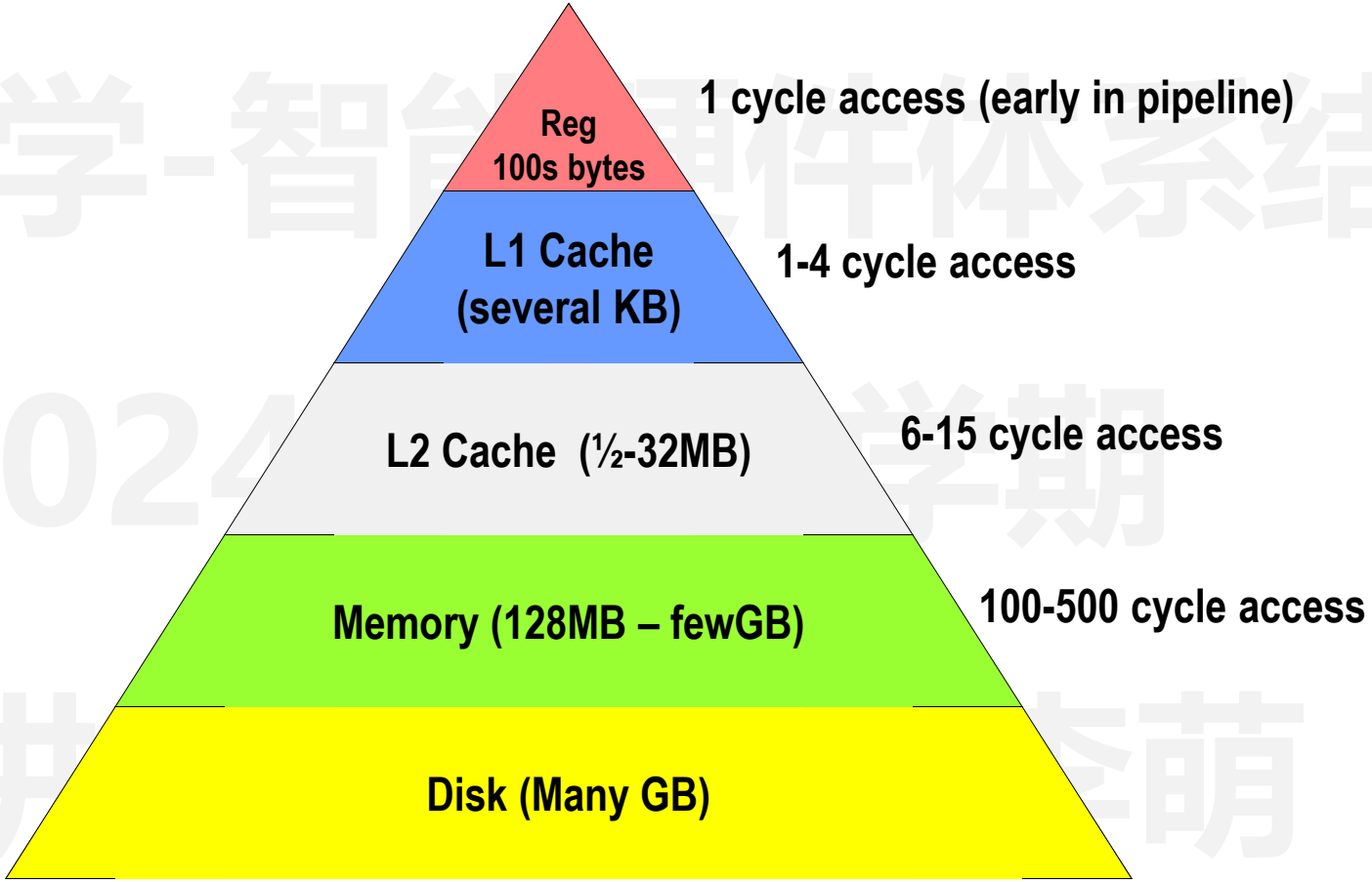
02. 分支处理机制与地址预测

03. 经典的MIPS架构实例分析

04. 多级缓存微架构与一致性

缓存Cache设计基础

- 当前芯片架构中的缓存层级



缓存Cache设计中的局部性

- **局部性原理 – 时间局部性与空间局部性**

- 局部性原理：
  - 程序倾向于重复使用最近使用过的数据和指令。
  - *时间局部性：* 最近引用的项目很可能在不久的将来被引用。
  - *空间局部性：* 地址相近的物品往往会在时间上被紧密引用。

示例中的局部性：

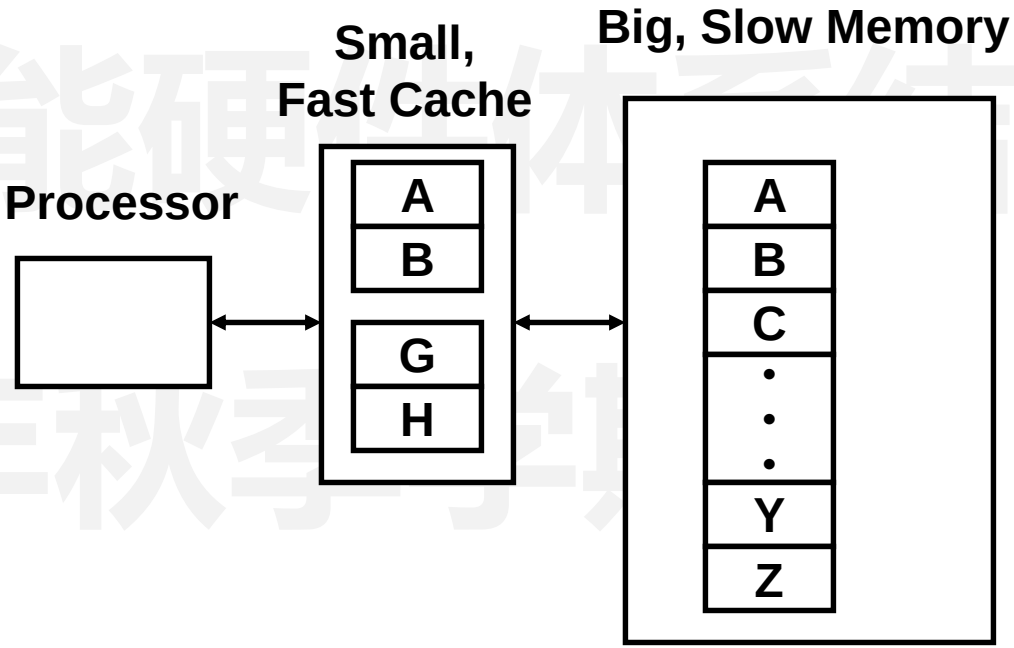
- **数据**
  - 连续引用数组元素（空间）
- **指令**
  - 按顺序（空间）引用指令
  - 反复循环（时间）

```
sum = 0;
for (i = 0; i < n; i++)
        sum += a[i];
*v = sum;
```

缓存Cache的基本思路

- 加快访存速度

- **Main Memory**
  - Stores words
    A–Z in example
- **Cache**
  - Stores subset of the words
    4 in example
  - Organized in lines
    - Multiple words
    - To exploit spatial locality
- **Access**
  - Word must be in cache for processor to access

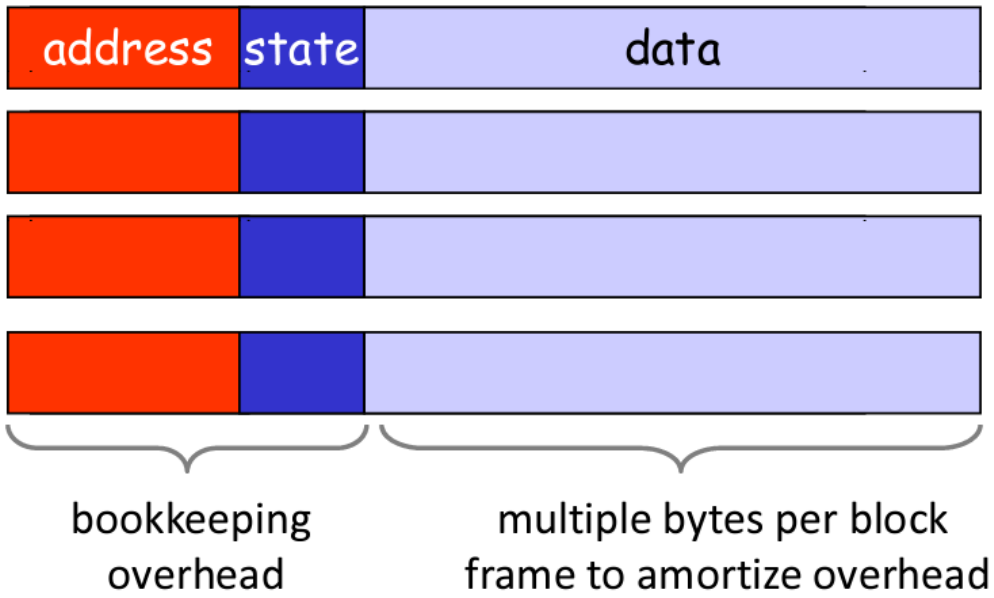


缓存Cache的基本思路

- Cache的抽象模型

Keep recently accessed block in “block frame”

- state (e.g., valid)
- address tag
- data



On memory read

- if incoming address corresponds to on e of the stored address tag then
  - **HIT**
  - return data
- else
  - **MISS**
  - Choose and displace a current block in use
  - fetch new (referenced) block from memory into frame
  - return data

缓存Cache的基本思路

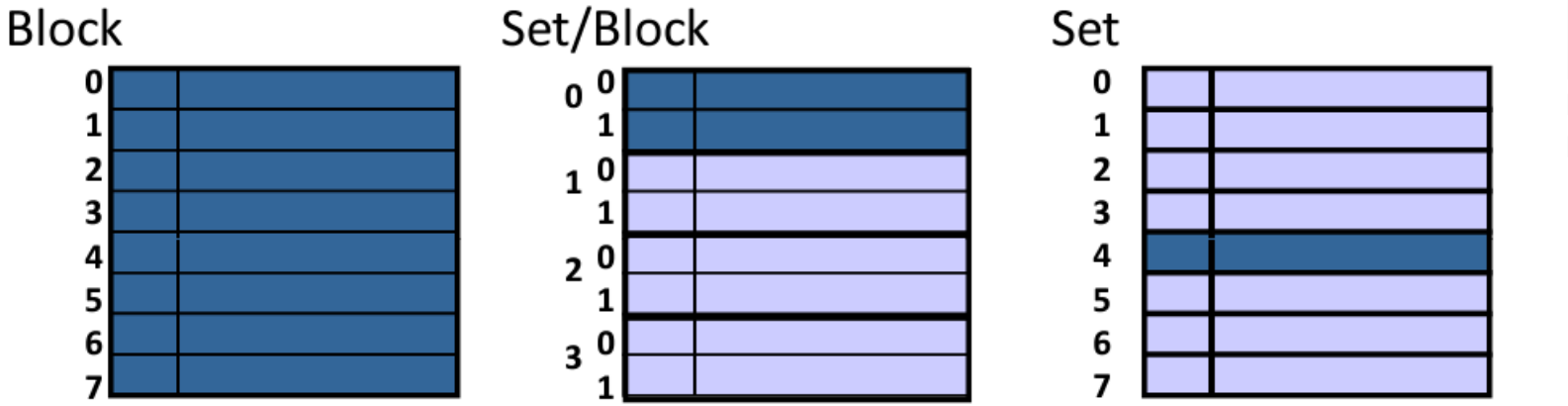
北京大学 PEKING UNIVERSITY

- Cache使用术语
  - **block (cache line)** — minimum unit that may be present
  - **hit** — block is found in the cache
  - **miss** —block is not found in the cache
  - **miss ratio** — fraction of references that miss
  - **hit time** —time to access the cache miss penalty
  - **miss penalty**
    - time to replace block in the cache + deliver to upper level
    - access time — time to get first word
    - transfer time — time for remaining words

缓存Cache的基本思路

- **Cache Block Placement**

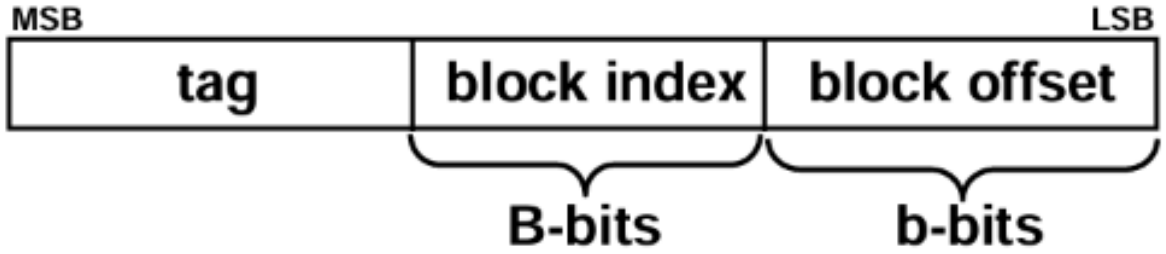
Where does block 12 (b'1100) go?



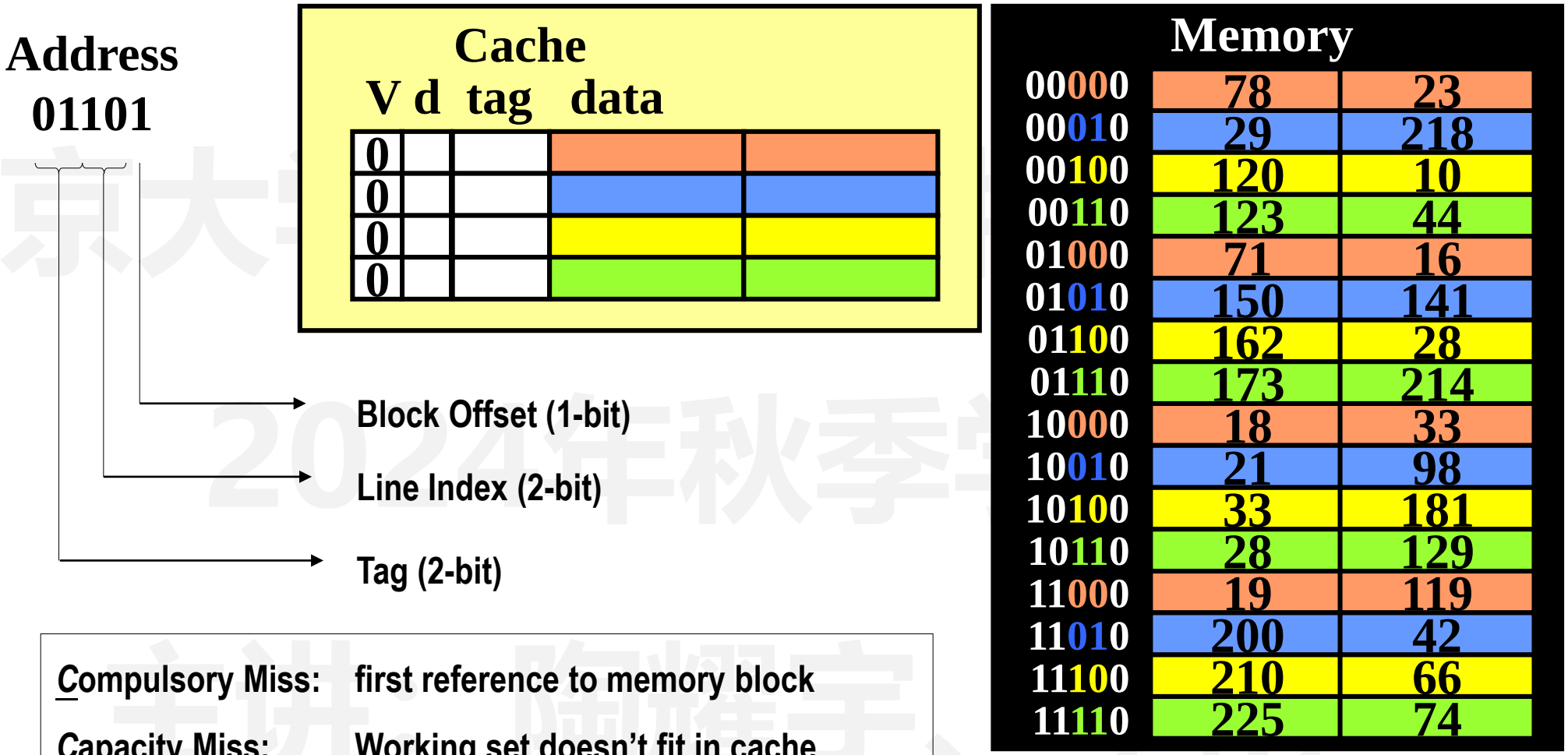
Fully-associative	Set-associative	Direct-mapped
block goes in any frame	a block goes in any frame in exactly one set	block goes in exactly one frame
(think all frames in 1 set)	(frames grouped into sets)	(think 1 frame per set)

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)



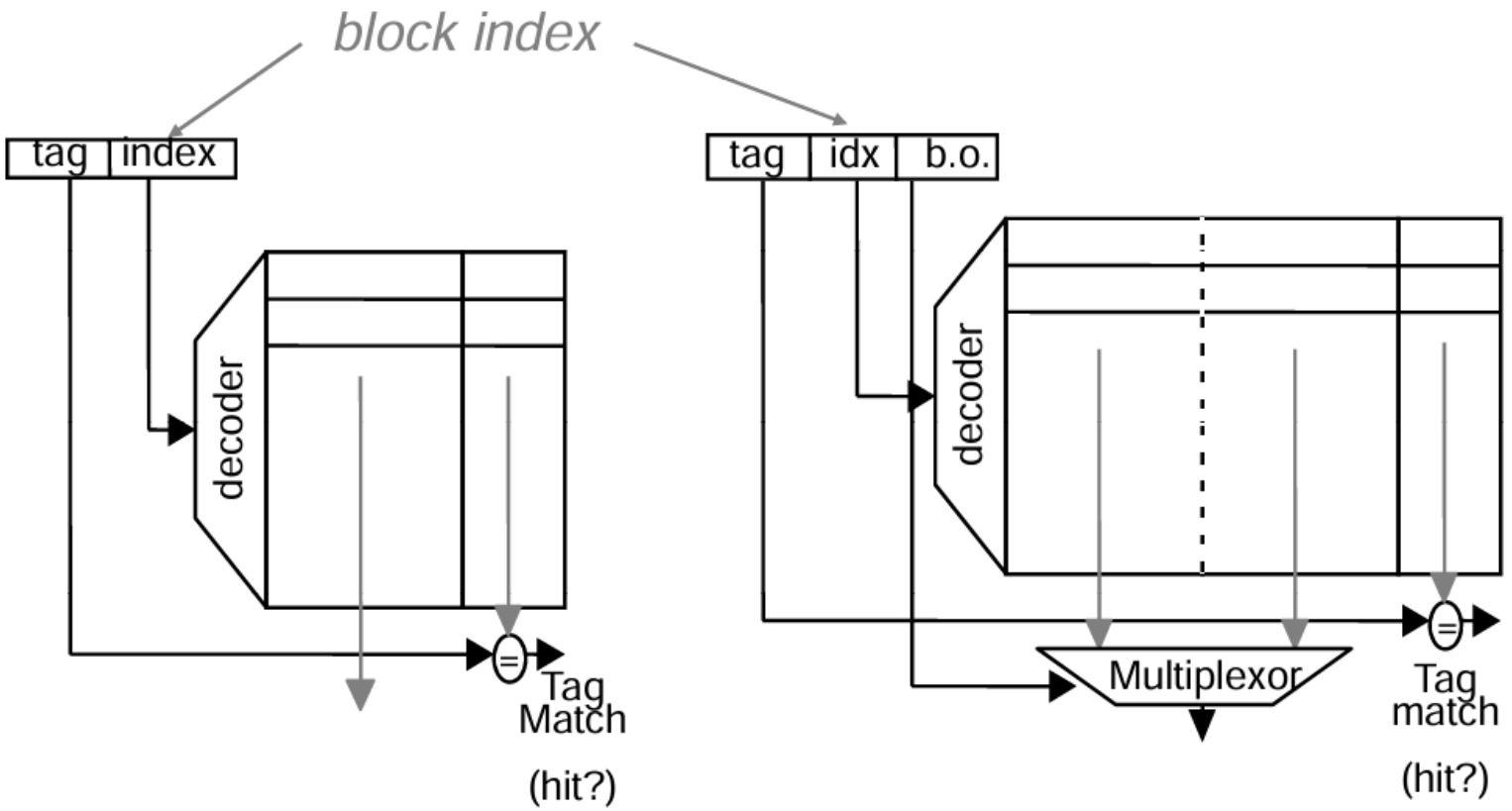
Direct-Mapped Cache设计



3-C's

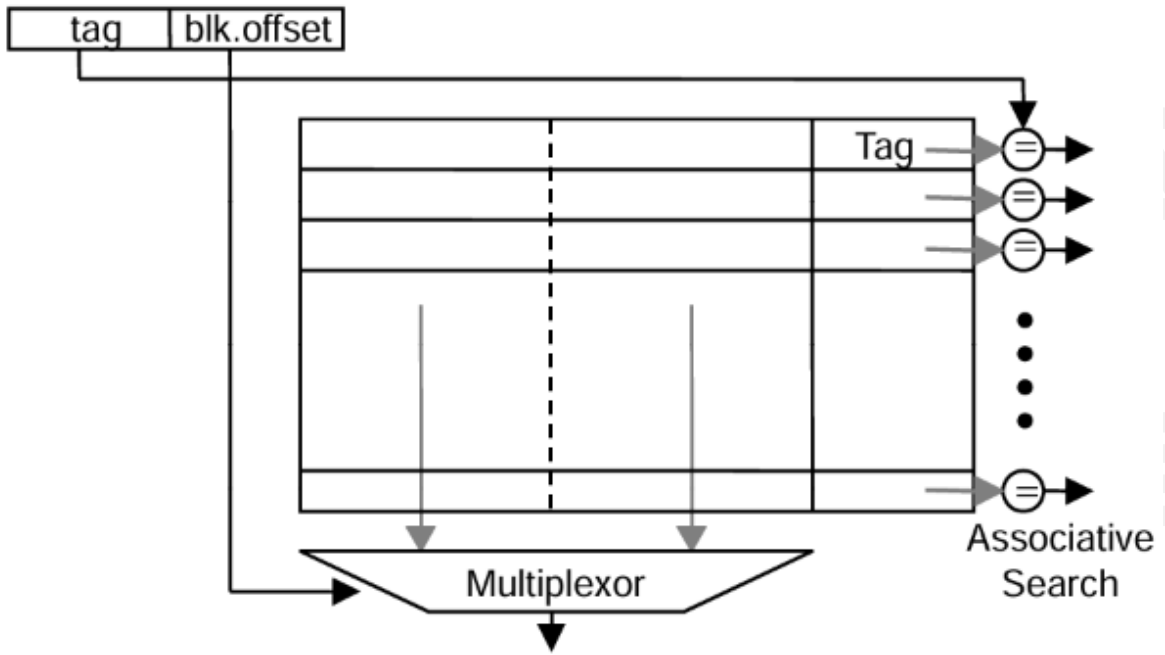
Compulsory Miss:	first reference to memory block
Capacity Miss:	Working set doesn't fit in cache
Conflict Miss:	Working set maps to same cache line

Direct-Mapped Cache设计



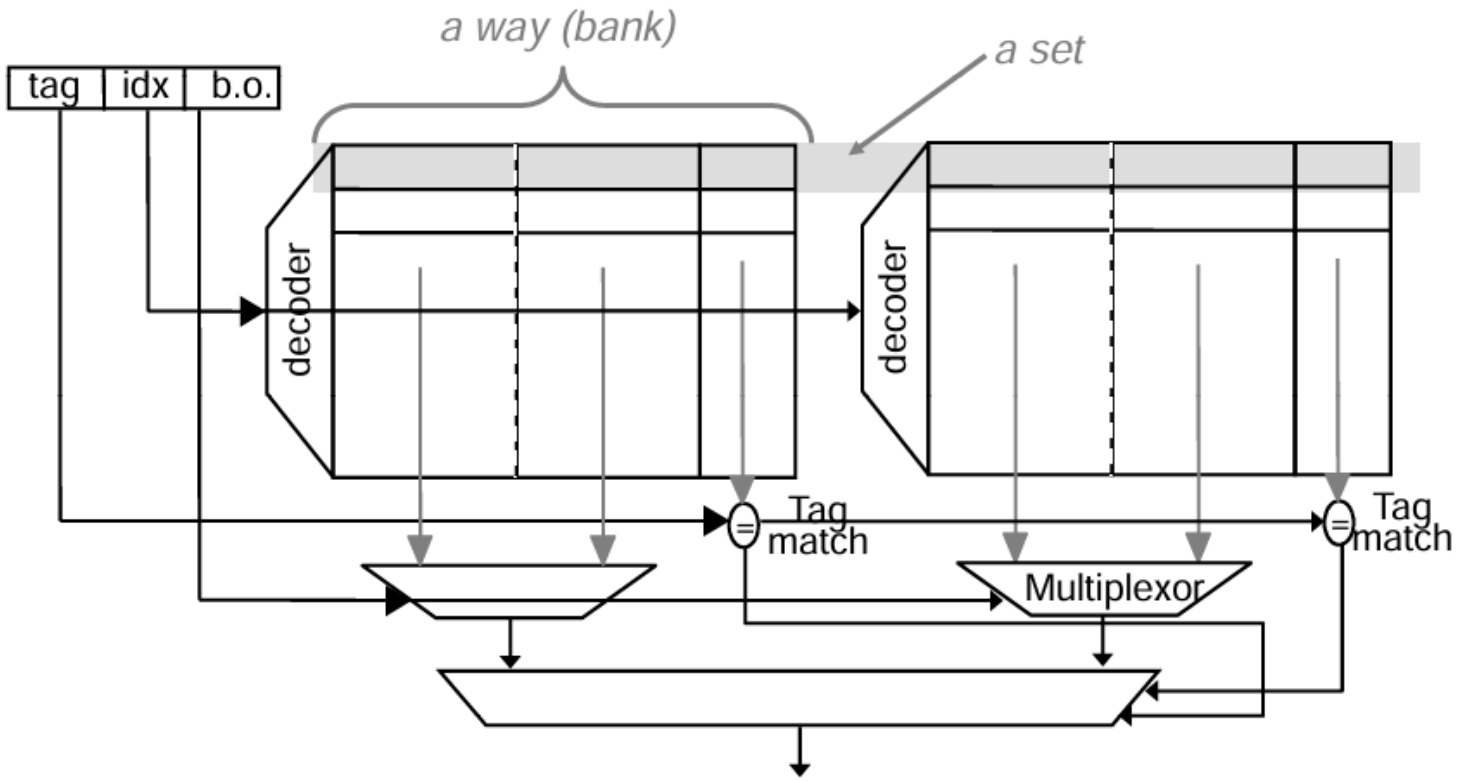
Don't forget to check the valid/state bits

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**
  - Which block in a set to replace on a miss?
    - Ideally — replace the block that "will" be accessed the furthest in the future
      - How do you implement it?
    - Approximations:
      - **Least recently used — LRU**
        - optimized (assume) for temporal locality (expensive for more than 2-way)
      - **Not most recently used — NMRU**
        - track MRU, random select from others, good compromise
      - **Random**
        - **nearly as good as LRU, simpler (usually pseudo-random)**
        - How much can block replacement policy matter?

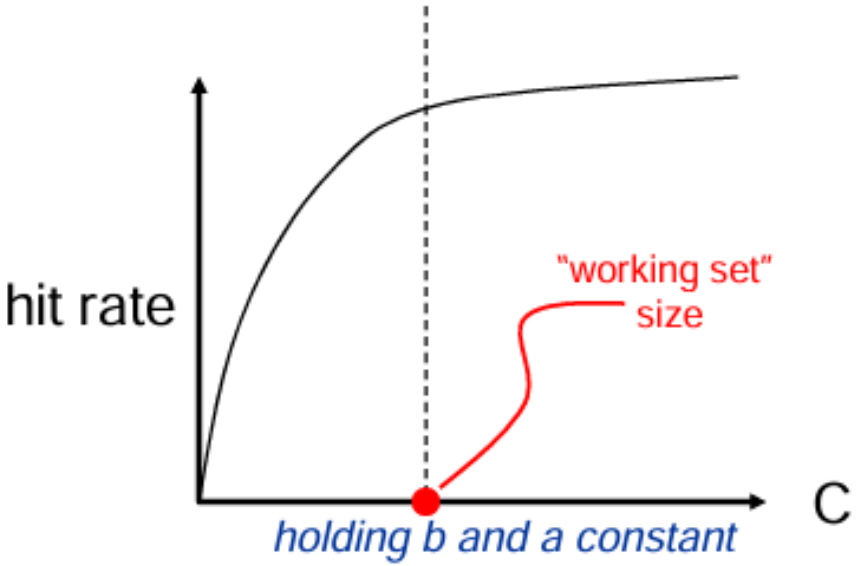
Cache Miss种类

- Miss Classification (3+1 C’s)
  - **Compulsory Miss**
    - **“cold miss” on first access to a block**
      - —defined as: miss in infinite cache
  - **Capacity Miss**
    - **misses occur because cache not large enough** — defined as: miss in fully-associative cache
  - **Conflict Miss**
    - **misses occur because of restrictive mapping strategy**
    - only in set-associative or direct-mapped cache
      - —defined as: not attributable to compulsory or capacity
  - **Coherence Miss**
    - **misses occur because of sharing among multiprocessors**

Cache参数的选择

- Cache Size（C）

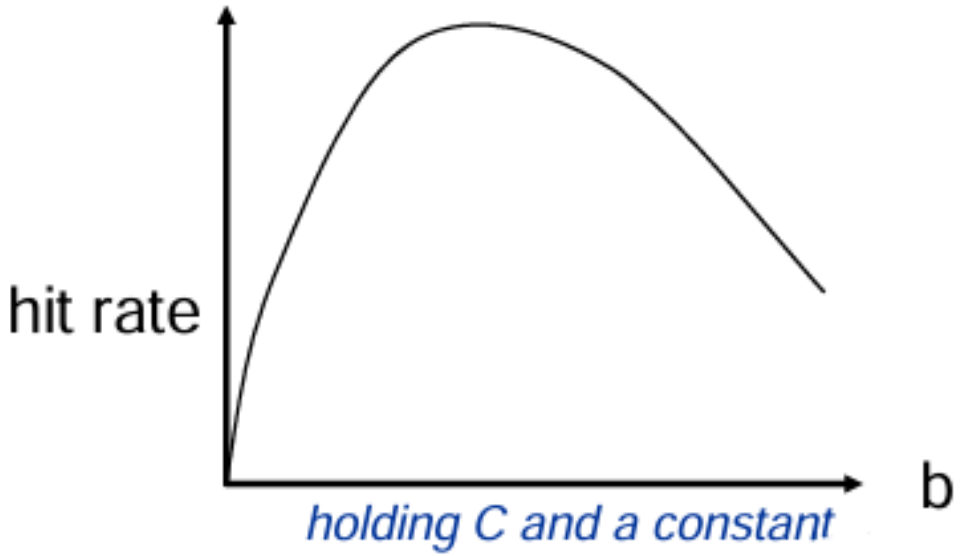
- **Cache size is the total data (not including tag) capacity**
  - bigger can exploit temporal locality better
  - not ALWAYS better
- **Too large a cache**
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- **Too small a cache**
  - don’t exploit temporal locality well
  - useful data constantly replaced



Cache参数的选择

- Block Size（b）

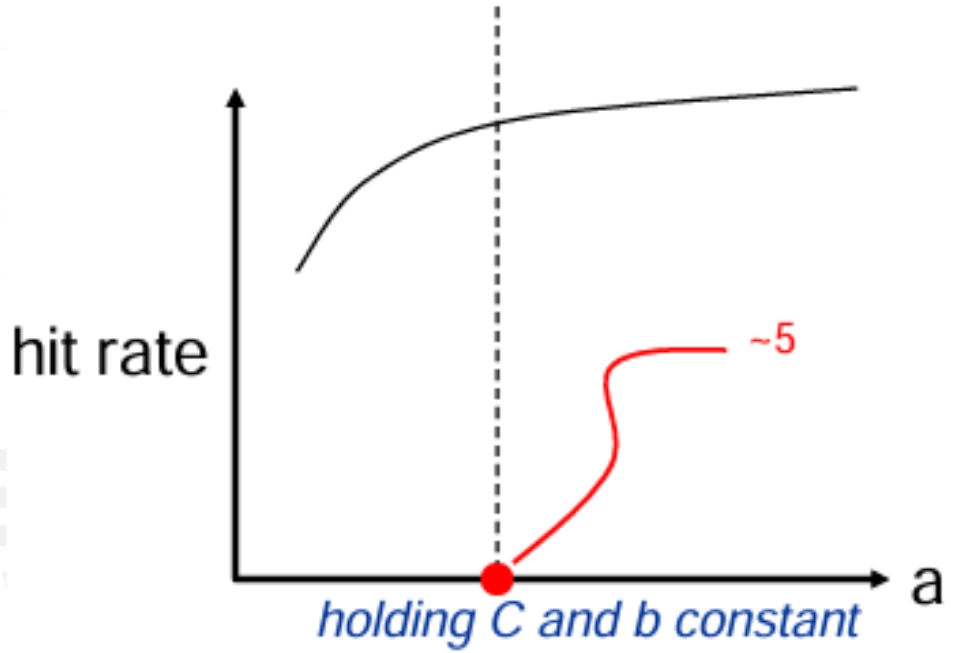
- **Block size is the data that is**
  - associated with an address tag
  - not necessarily the unit of transfer between hierarchies (remember sub-blocking)
- **Too small blocks**
  - don’t exploit spatial locality well
  - have inordinate tag overhead
- **Too large blocks**
  - useless data transferred
  - useful data permanently replaced
  - —too few total # blocks



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-way associative
  - **Larger associativity**
    - lower miss rate less variation among programs
    - only important for small “C/b”
  - **Smaller associativity**
    - lower cost, faster hit time

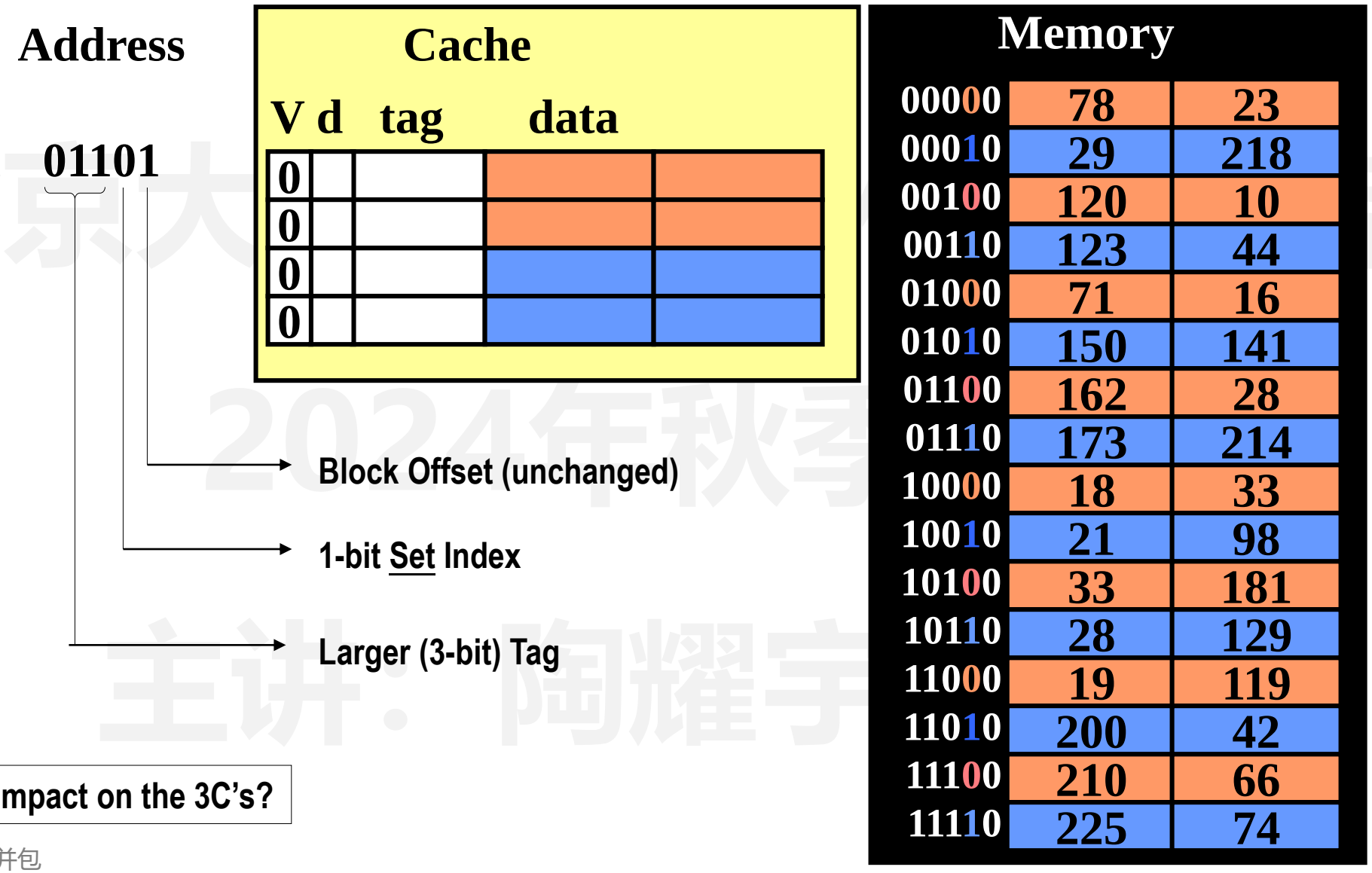


Cache设计选择的影响

- Cache写入和Miss处理策略

- Write Policy: How to deal with write misses?
  - ***Write-through / no-allocate***
    - update memory on each write
    - keeps memory up-to-date
  - ***Write-back / write-allocate***
    - update memory only on block replacement
    - Many cache lines are only read and never written to
    - add “dirty” bit to status word
      - originally cleared after replacement
      - set when a block frame is written to
      - only write back a dirty block, and “drop” clean blocks w/o memory update

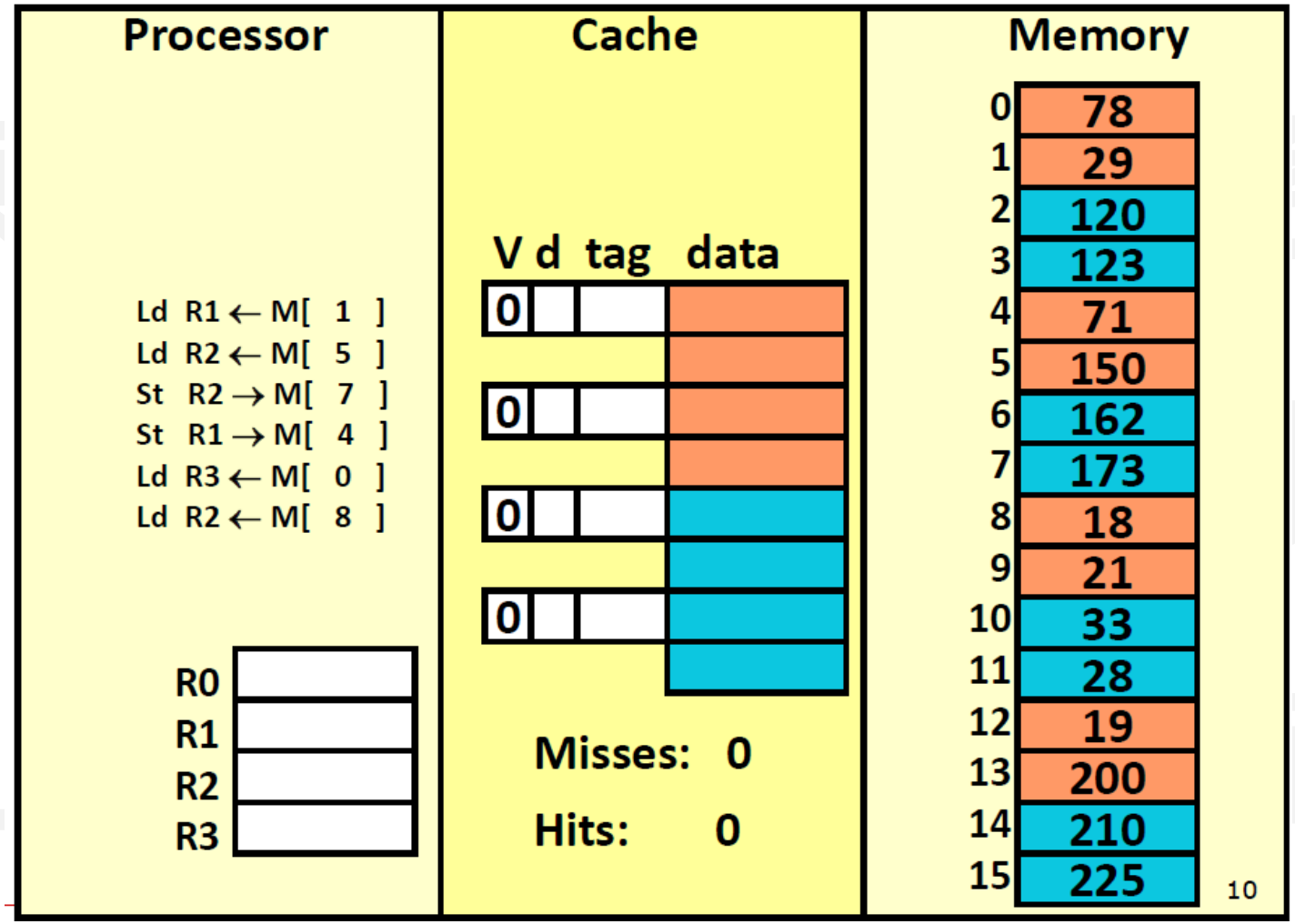
2-Way Set Associative Cache设计



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

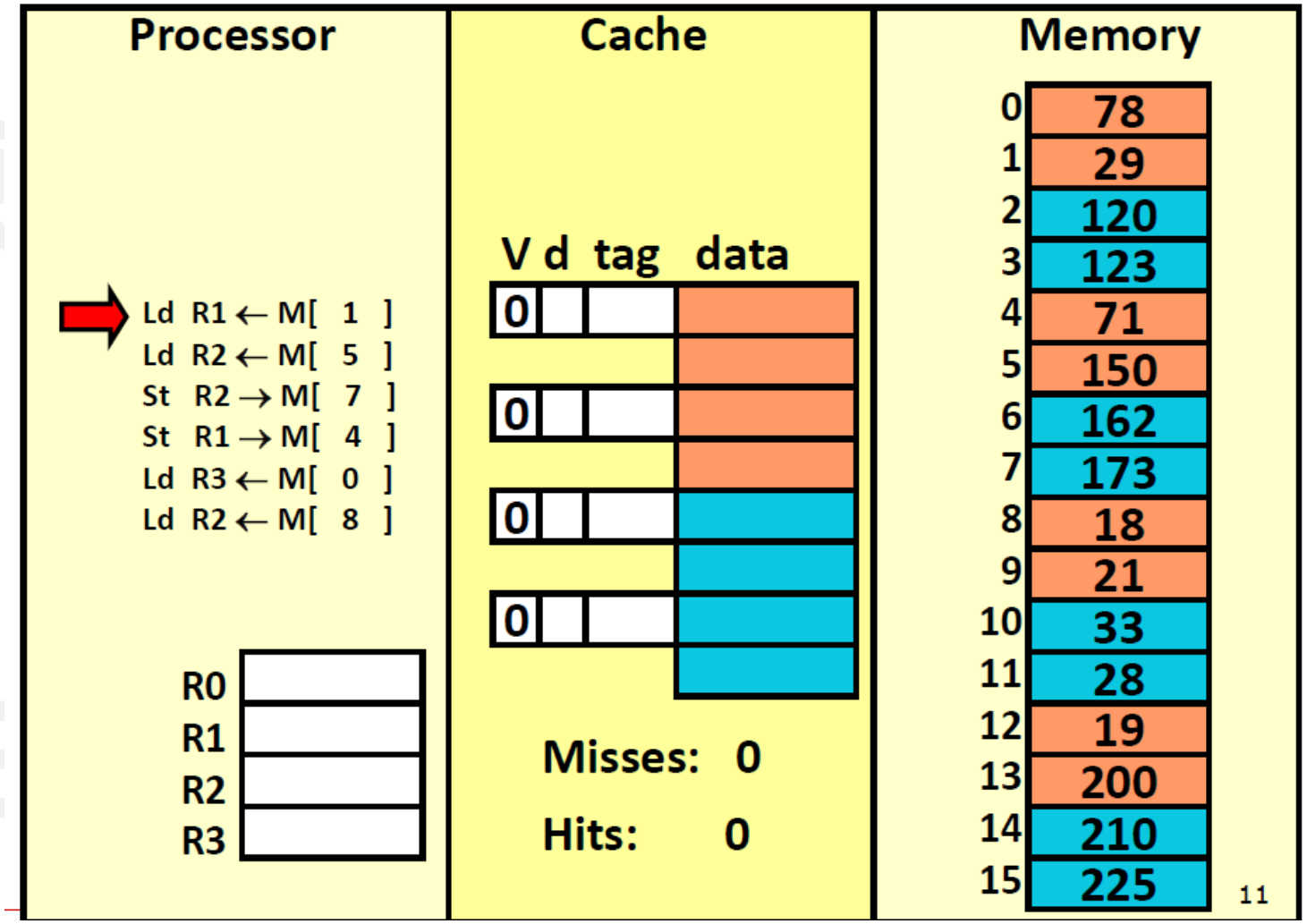
2-Way Set Associative Cache实例

- Write-back & Write-allocate



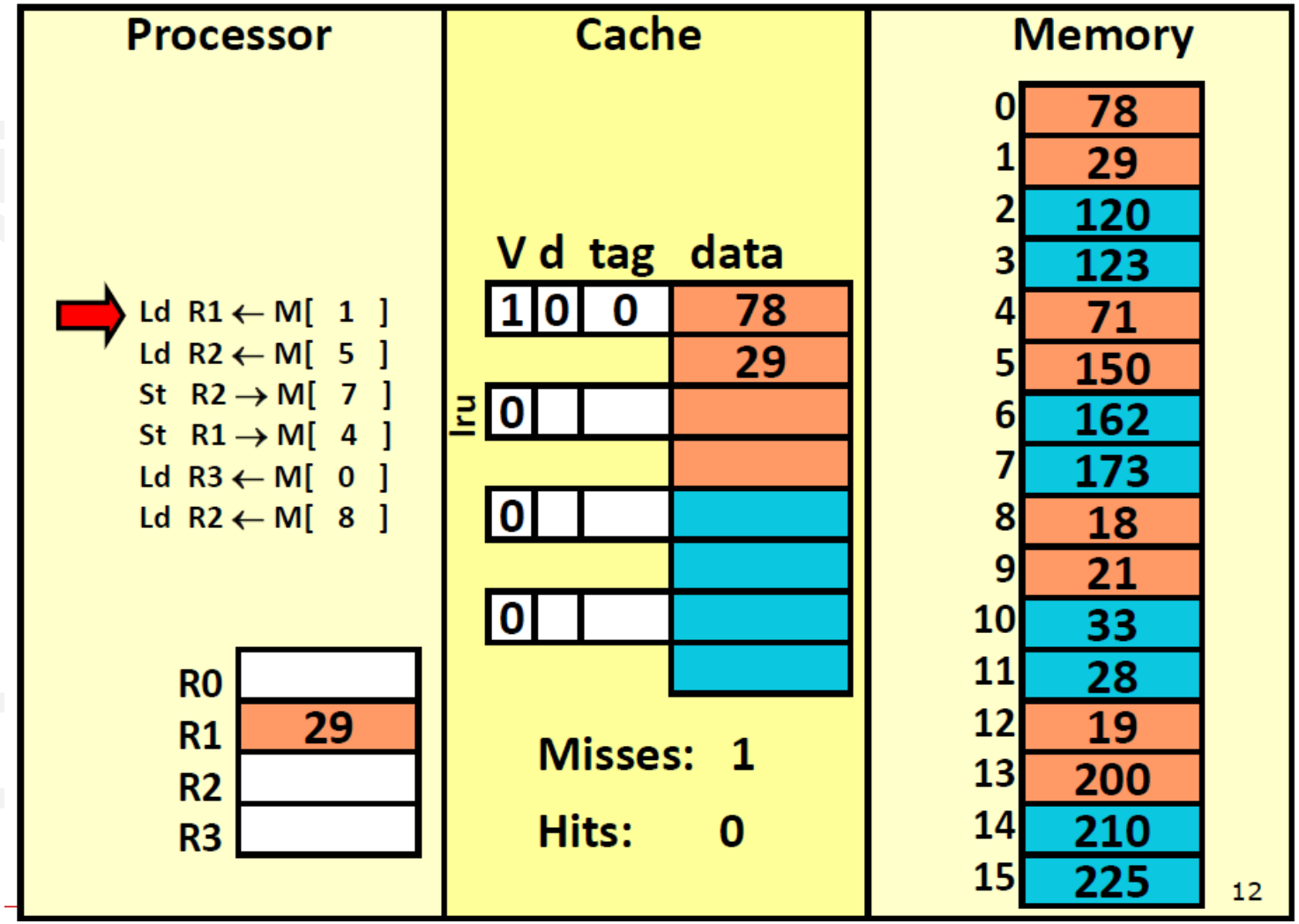
2-Way Set Associative Cache实例

- Write-back & Write-allocate



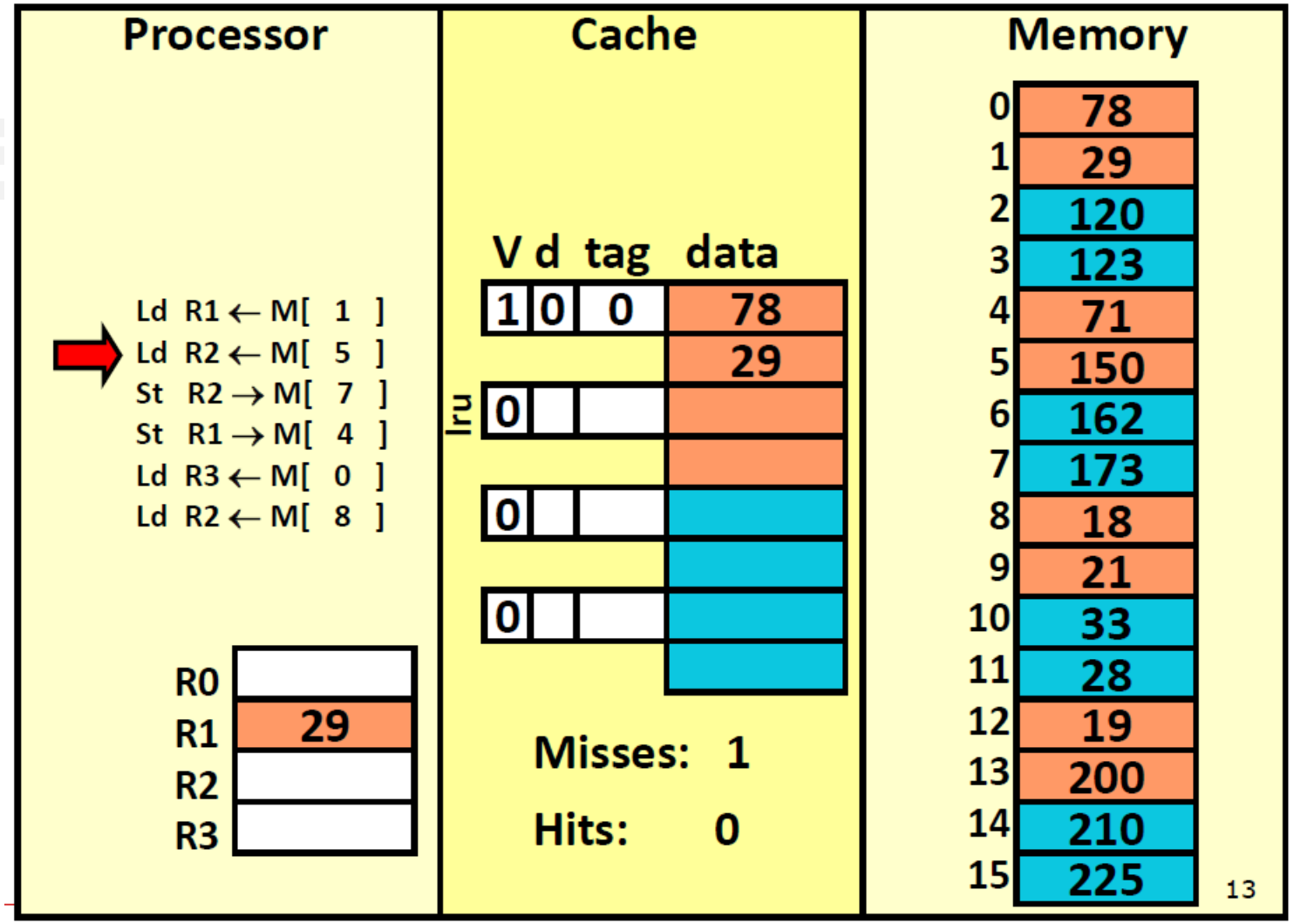
2-Way Set Associative Cache实例

- Write-back & Write-allocate



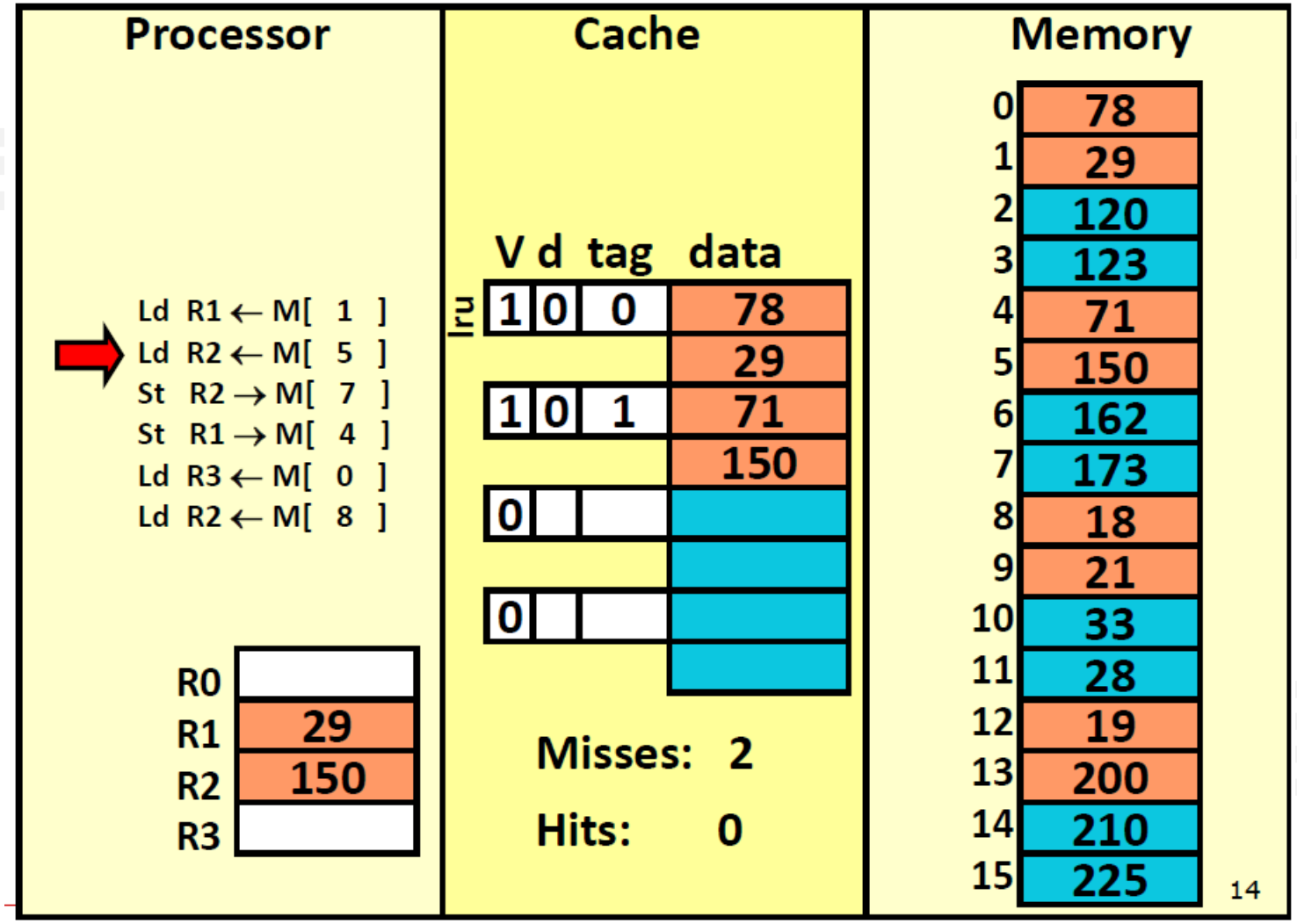
2-Way Set Associative Cache实例

- Write-back & Write-allocate



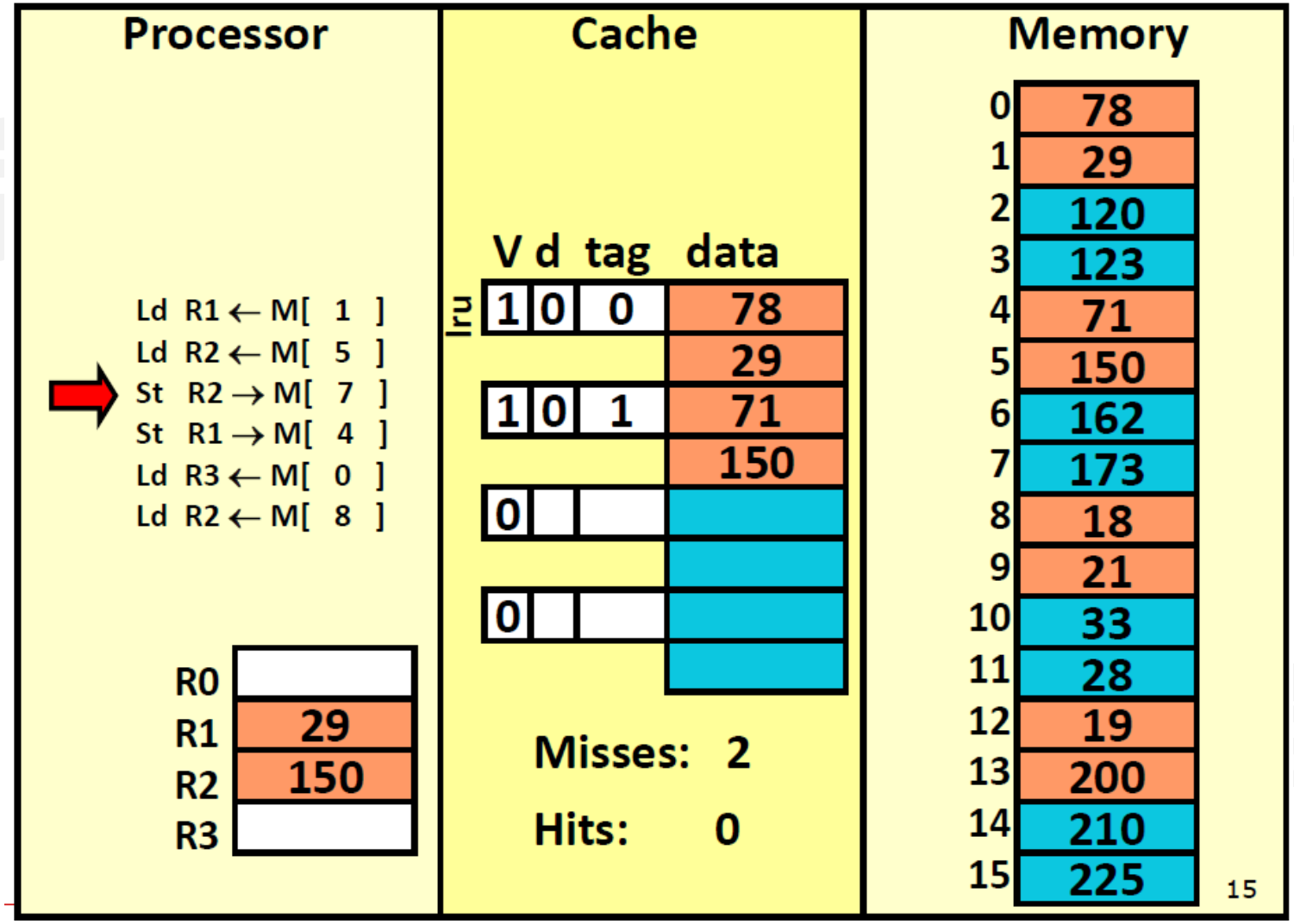
2-Way Set Associative Cache实例

- Write-back & Write-allocate



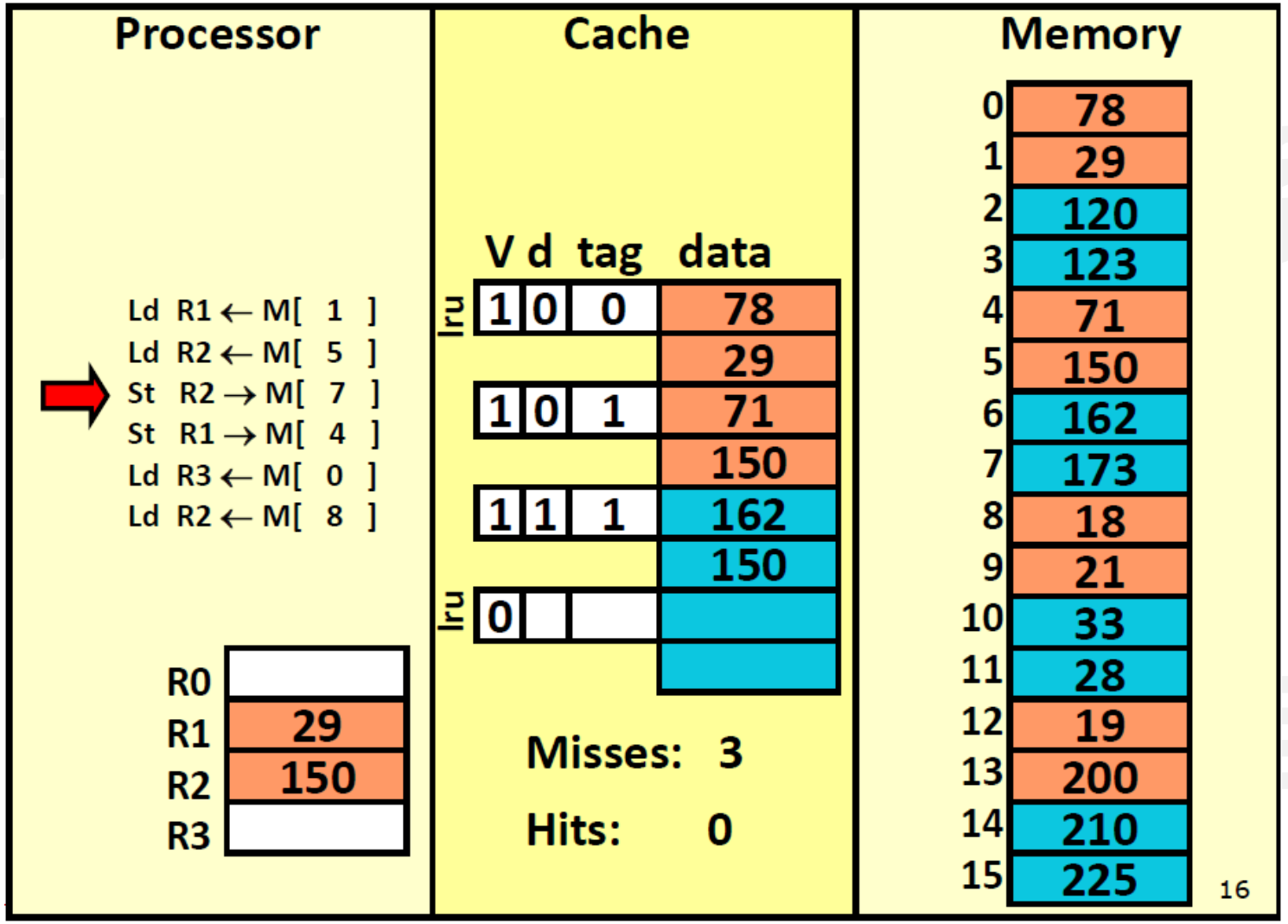
2-Way Set Associative Cache实例

- Write-back & Write-allocate



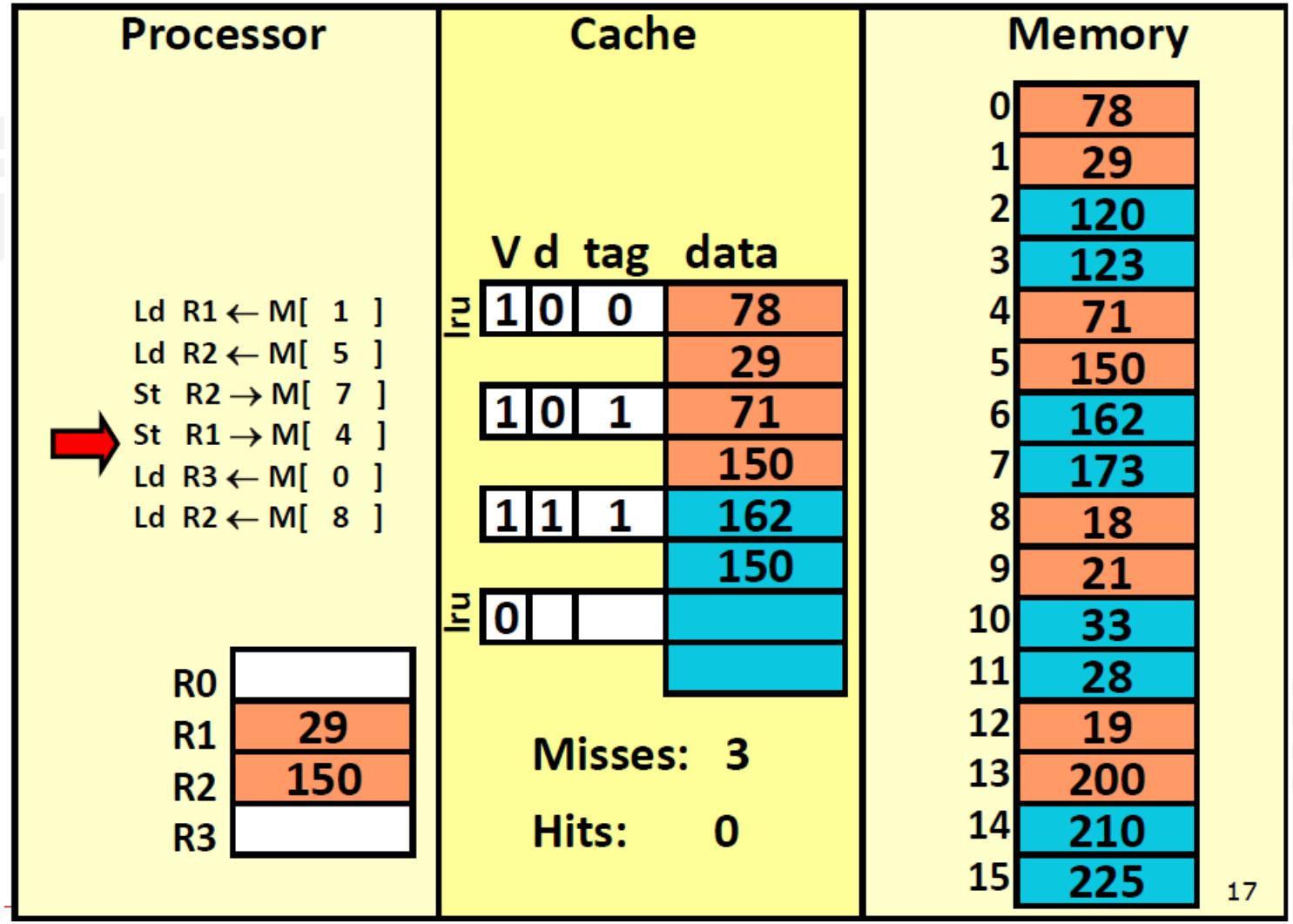
2-Way Set Associative Cache实例

- Write-back & Write-allocate



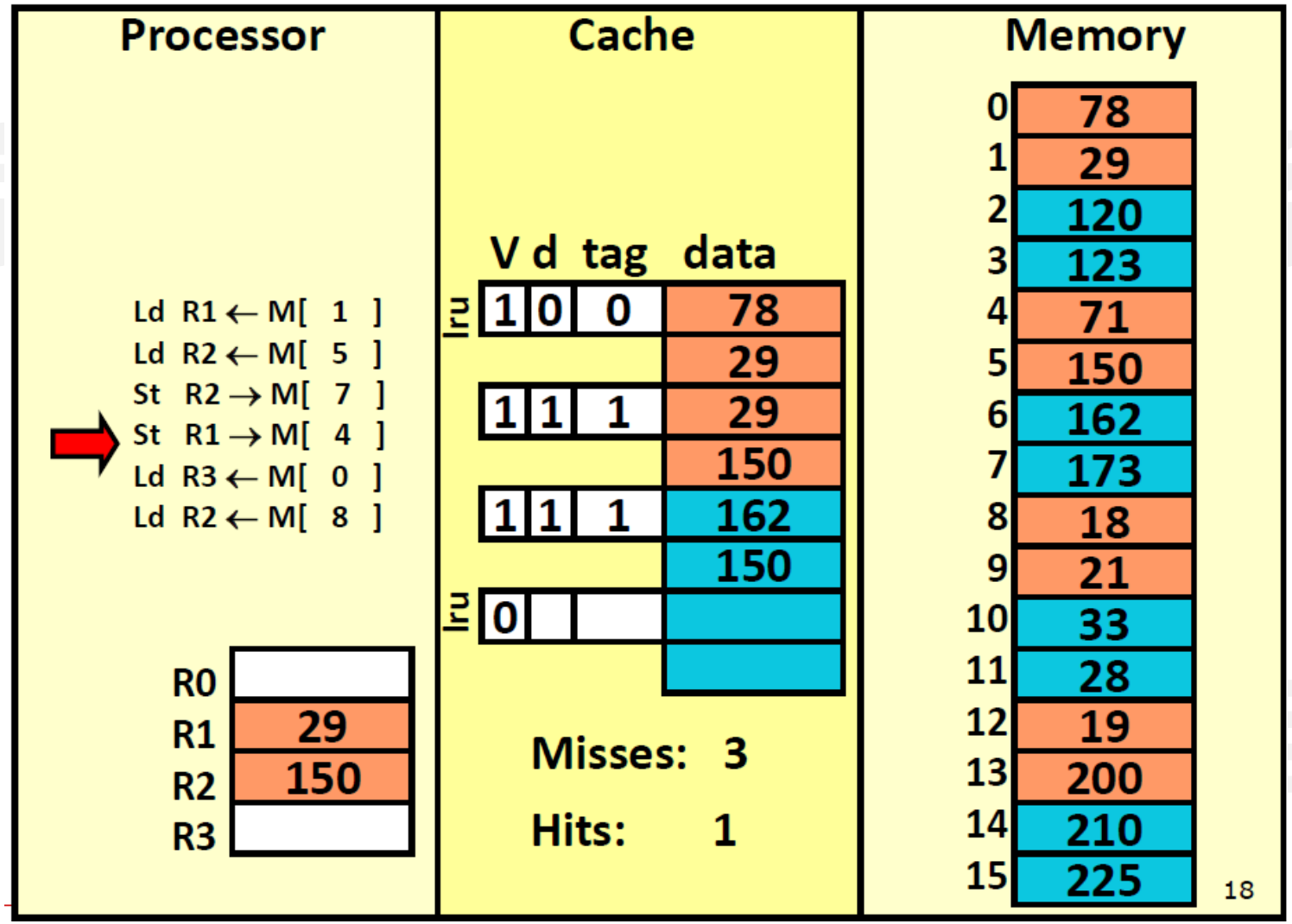
2-Way Set Associative Cache实例

- Write-back & Write-allocate



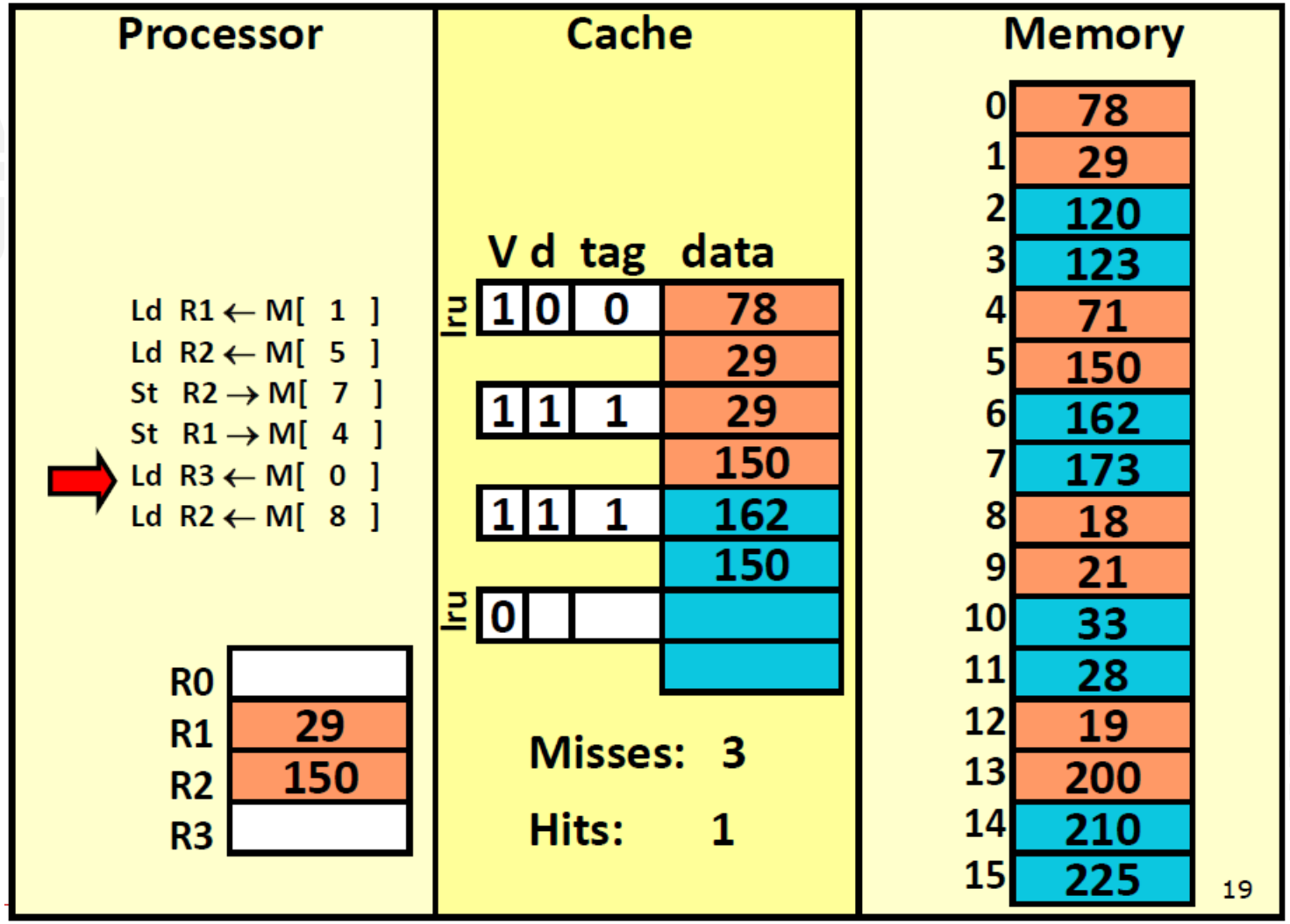
2-Way Set Associative Cache实例

- Write-back & Write-allocate



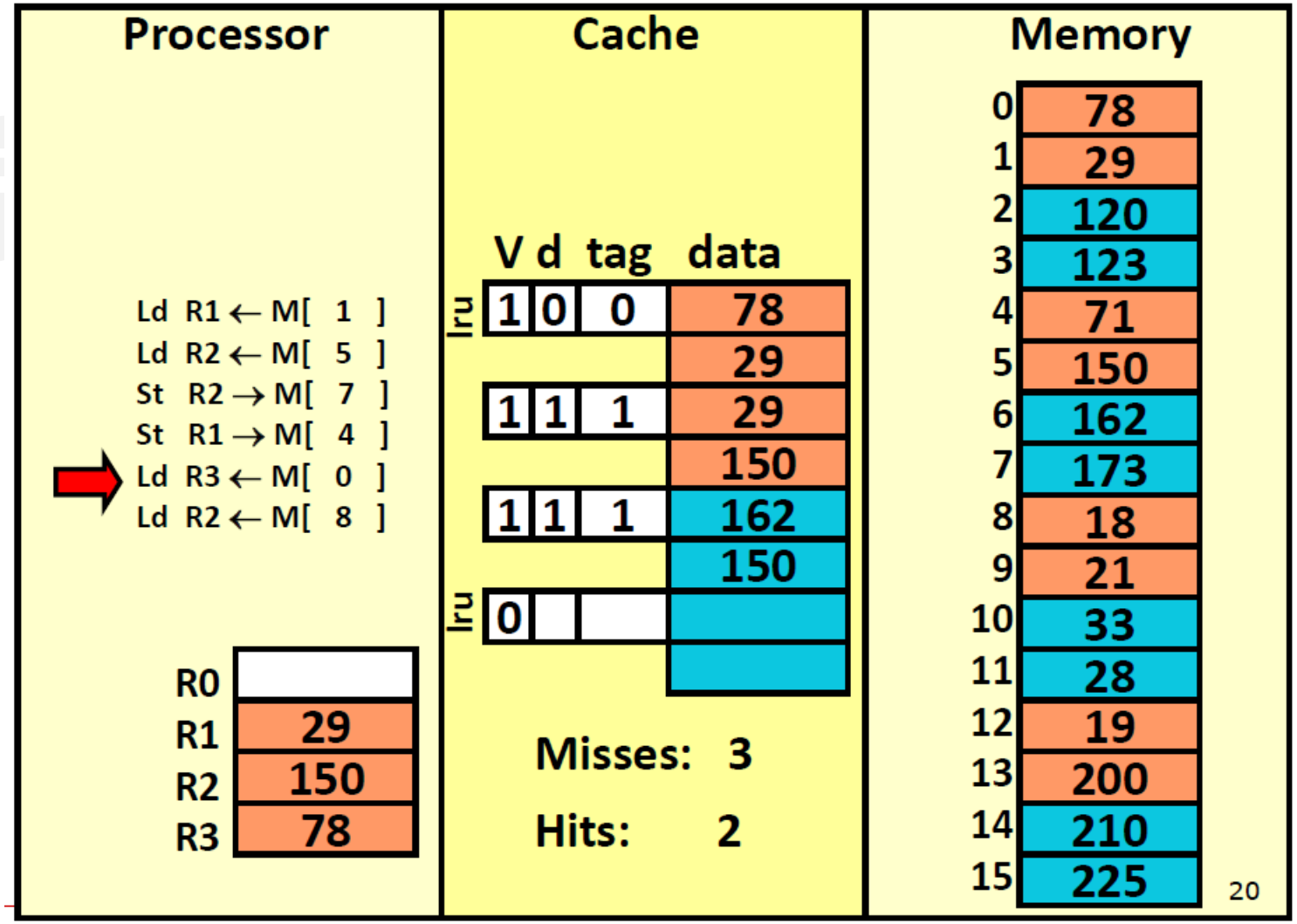
2-Way Set Associative Cache实例

- Write-back & Write-allocate



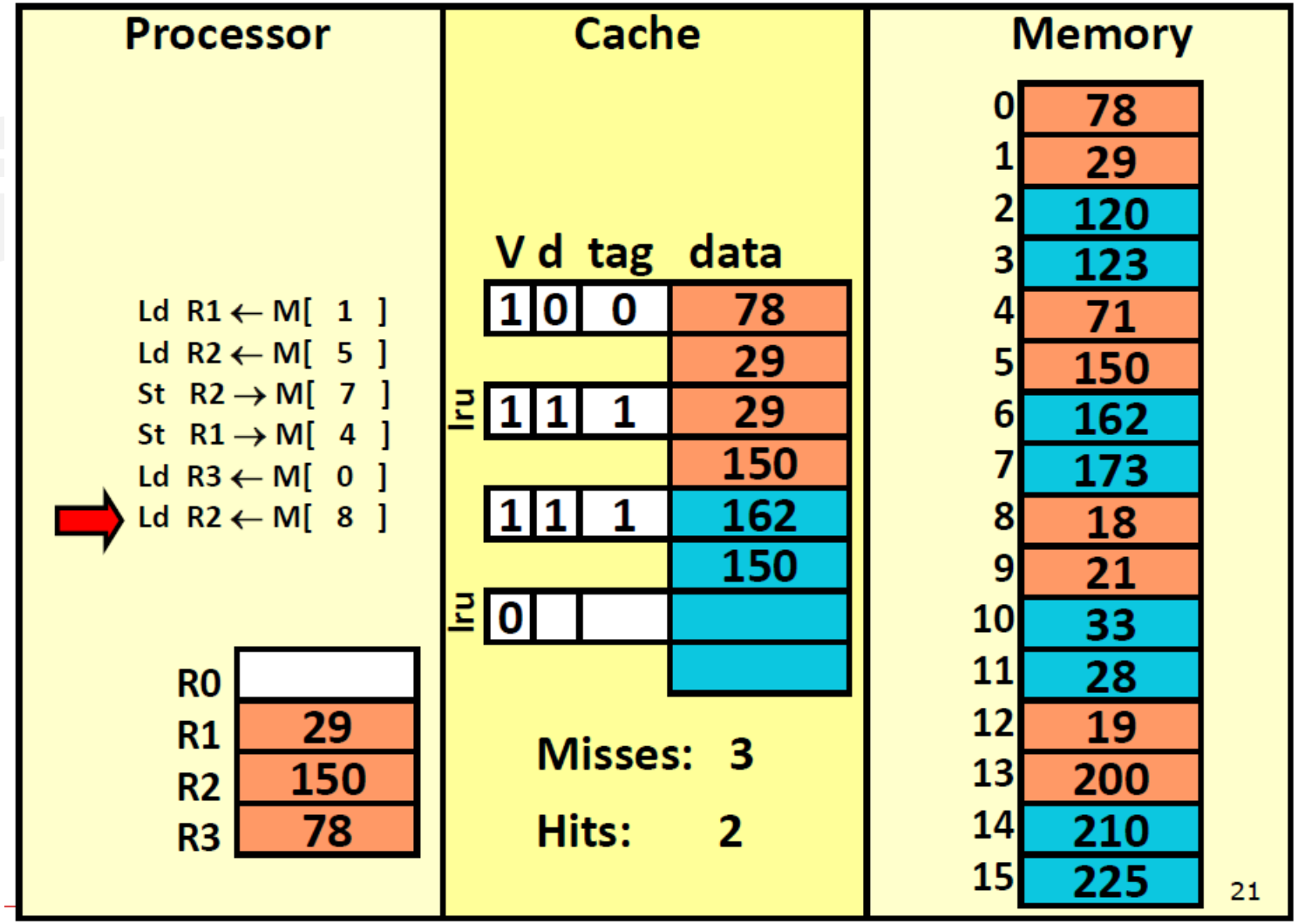
2-Way Set Associative Cache实例

- Write-back & Write-allocate



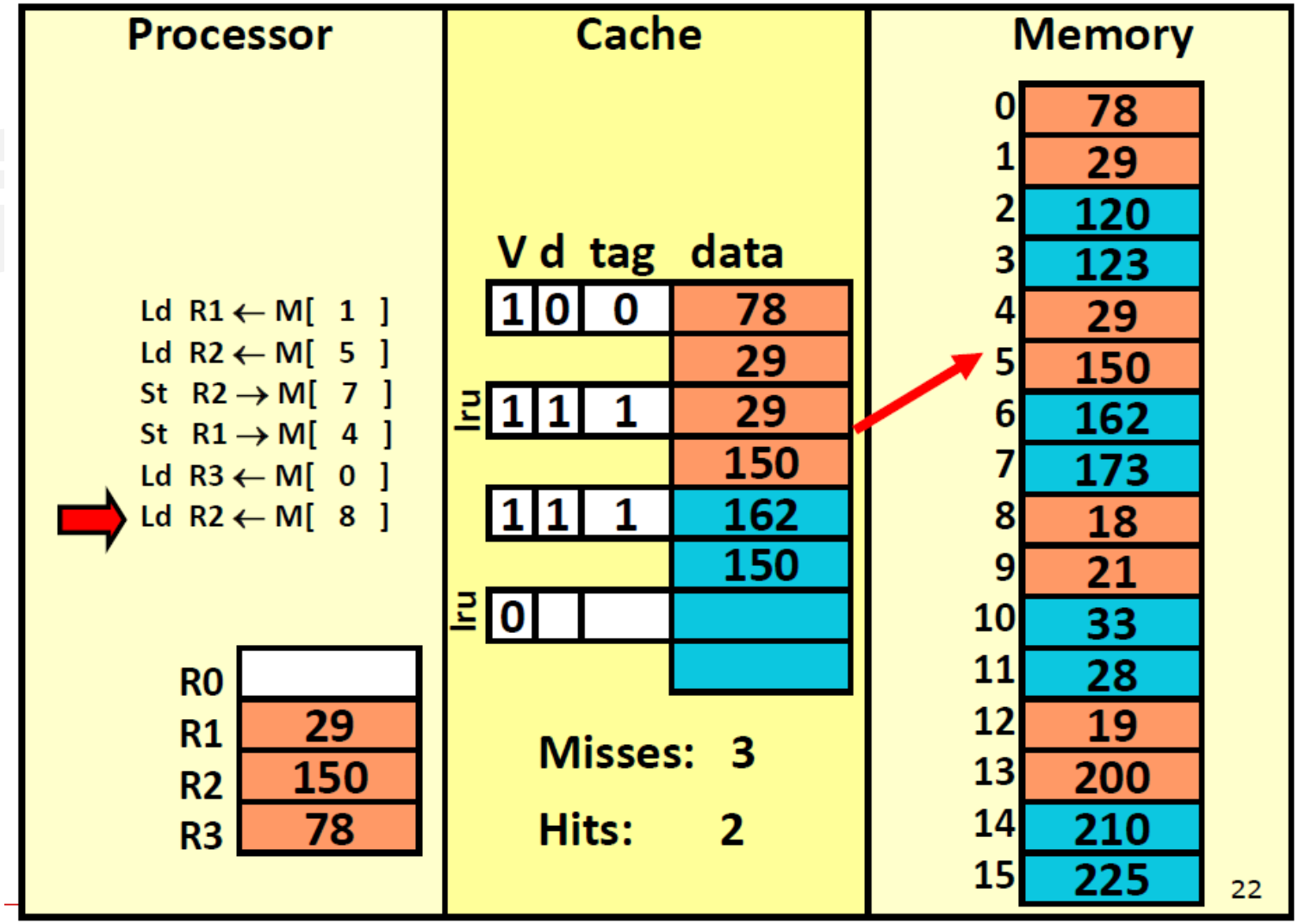
2-Way Set Associative Cache实例

- Write-back & Write-allocate



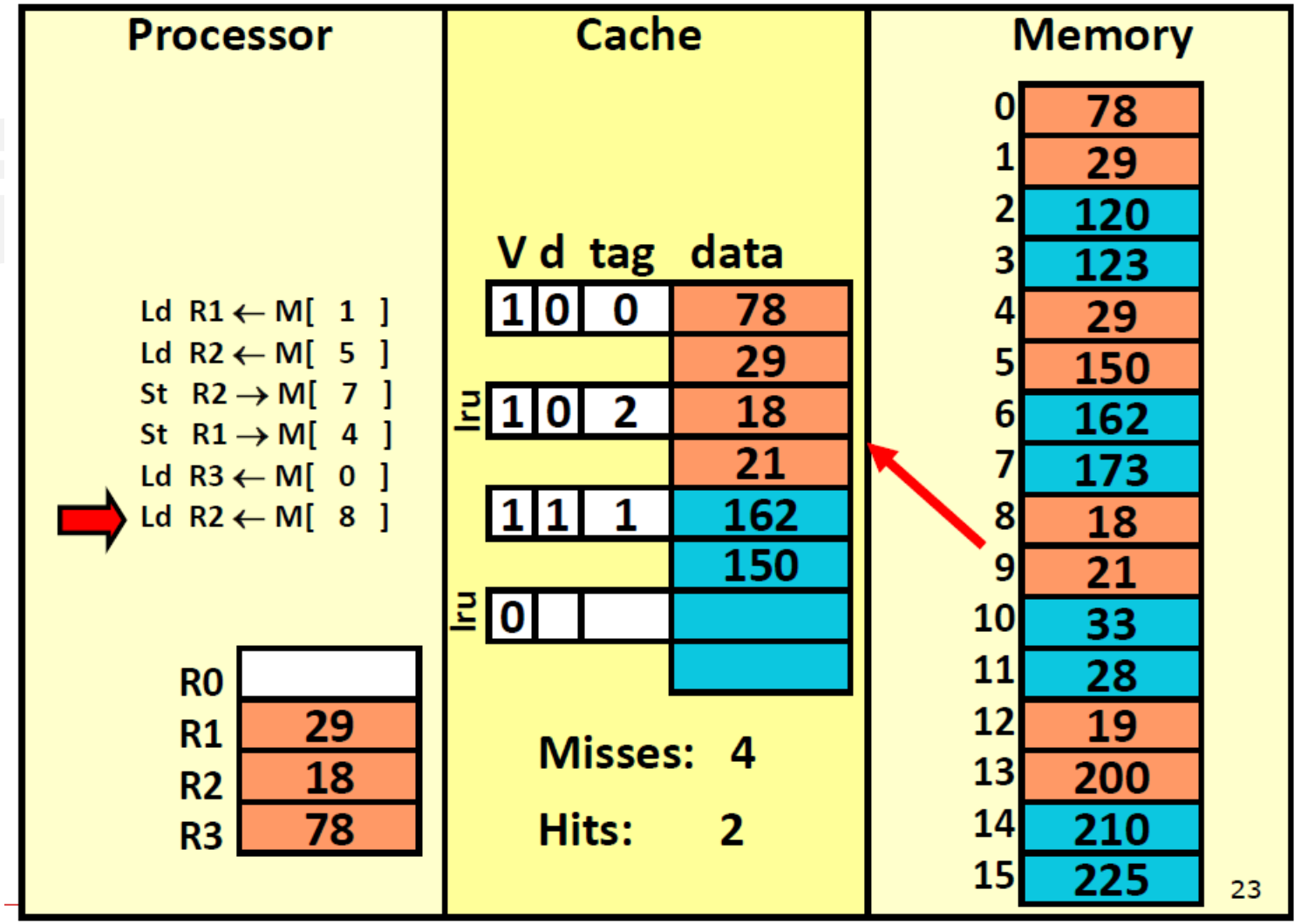
2-Way Set Associative Cache实例

- 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

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
  - 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

下一次课 – 第8次课

- 缓存一致性与虚拟内存技术

北京大学-智能硬件体系结构

2024年秋季学期

主讲：陶耀宇、李萌