北京大学
PEKING UNIVERSITY

智能硬件体系结构

第九讲：MIPS架构与缓存设计

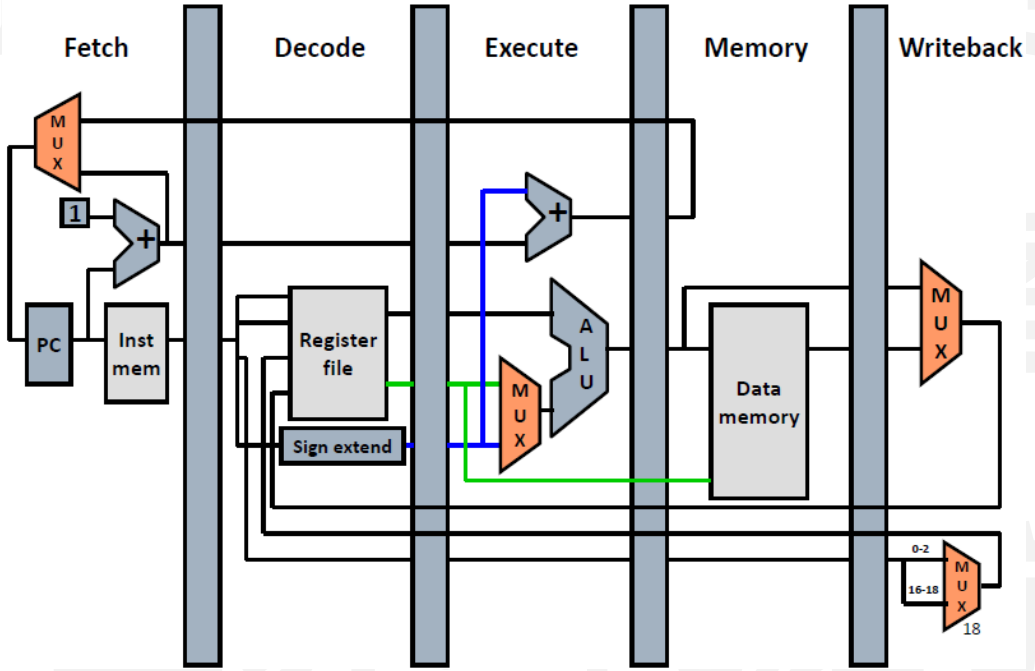
主讲：陶耀宇、李萌

2025年秋季

注意事项

- 课程作业情况

- **第2次lab时间：11月14日-12月10日晚23:59**




```
├── Makefile
├── run_tests.sh              # 自动运行测试用例脚本
├── setup_env.sh              # 配置环境脚本
├── testbench/                # 测试平台及可视化
│   ├── testbench.v           # 主测试平台
│   └── ...
├── test_progs/               # 测试指令程序（汇编 .s 文件）
│   └── *.s                   # 示例程序文件
├── verilog/                  # 你将完成的核心 Verilog 文件
│   ├── pipeline.v            # 顶层模块
│   ├── if_stage.v            # IF 阶段实现
│   ├── id_stage.v            # ID 阶段实现
│   ├── ex_stage.v            # EX 阶段实现
│   ├── mem_stage.v           # MEM 阶段实现
│   ├── wb_stage.v            # WB 阶段实现
│   ├── regfile.v             # 通用寄存器文件
│   ├── icache.v              # 指令缓存（可作为黑盒使用）
│   └── cachemem.v            # 数据缓存（可作为黑盒使用）
└── vs-asm                    # 汇编工具，用于生成测试程序二进制
```

- **Lab2的目标：1、进一步熟悉硬件设计、验证与优化流程；2、了解流水线的基本运行模式**

注意事项

- 课程作业情况

```
├── Makefile
├── run_tests.sh              # 自动运行测试用例脚本
├── setup_env.sh              # 配置环境脚本
├── testbench/                # 测试平台及可视化
│   ├── testbench.v           # 主测试平台
│   └── ...
├── test_progs/               # 测试指令程序（汇编 .s 文件）
│   └── *.s                   # 示例程序文件
├── verilog/                  # 你将完成的核心 Verilog 文件
│   ├── pipeline.v            # 顶层模块
│   ├── if_stage.v            # IF 阶段实现
│   ├── id_stage.v            # ID 阶段实现
│   ├── ex_stage.v            # EX 阶段实现
│   ├── mem_stage.v           # MEM 阶段实现
│   ├── wb_stage.v            # WB 阶段实现
│   ├── regfile.v             # 通用寄存器文件
│   ├── icache.v              # 指令缓存（可作为黑盒使用）
│   └── cachemem.v            # 数据缓存（可作为黑盒使用）
└── vs-asm                    # 汇编工具，用于生成测试程序二进制
```

1. if_stage.v
   ~5行代码
2. id_stage.v
   ~15行代码
3. ex_stage.v
   ~15行代码
4. mem_stage.v
   <5行代码
5. wb_stage.v
   ~5行代码
6. pipeline.v
   ~10行代码

```
// 在此处插入你的代码
// 这里有任务详情的注释
// Insert your code below

... Your code here ...

// Insert your code above
```

注意事项

北京大学 PEKING UNIVERSITY

- **课程作业情况**

关于 Debug

在你完成 CPU 实现后，运行测试程序将自动生成多个调试输出文件。每个文件提供不同层次的信息，帮助你理解 CPU 在每个周期做了什么，哪里可能出错了。

Tips：假如你运行测试程序很久还没有输出，那很有可能是CPU死循环了，需要排查是否错误地发生了stall；

输出文件说明

文件名格式	内容简介
<test_program.s>_detailed.csv	**每个周期各阶段的详细信号变化**（包括寄存器值、ALU输出、控制信号、PC等），适合深入单步追踪执行流程。
<test_program.s>_memory.out	程序运行结束后，**内存中数据的最终状态**。
<test_program.s>_pipeline.out	**流水线五个阶段在每个周期的执行情况**（即哪条指令在哪个阶段），尽管本 Lab 中是顺序执行，也可用于观察执行是否正确。
<test_program.s>_program.out	仿真器的输出：辅助验证整体行为。
<test_program.s>_writeback.out	每次进入 WB 阶段时，**写回的寄存器编号、值和对应的 PC**。

你可以在 ./golden_result 里面找到期望的输出，并用于参考；

注意事项

- **课程作业情况**

关于 Debug

在你完成 CPU 实现后，运行测试程序将自动生成多个调试输出文件。每个文件提供不同层次的信息，帮助你理解 CPU 在每个周期做了什么，哪里可能出错了。

Tips：假如你运行测试程序很久还没有输出，那很有可能是CPU死循环了，需要排查是否错误地发生了stall；

输出文件说明

文件名格式	内容简介
<test_program.s>_detailed.csv	**每个周期各阶段的详细信号变化**（包括寄存器值、ALU输出、控制信号、PC等），适合深入单步追踪执行流程。
<test_program.s>_memory.out	程序运行结束后，**内存中数据的最终状态**。
<test_program.s>_pipeline.out	**流水线五个阶段在每个周期的执行情况**（即哪条指令在哪个阶段），尽管本 Lab 中是顺序执行，也可用于观察执行是否正确。
<test_program.s>_program.out	仿真器的输出：辅助验证整体行为。
<test_program.s>_writeback.out	每次进入 WB 阶段时，**写回的寄存器编号、值和对应的 PC**。

你可以在 ./golden_result 里面找到期望的输出，并用于参考；

注意事项

北京大学 PEKING UNIVERSITY

- **课程作业情况**

1. 压缩为zip格式的实现的 Verilog 文件（提交整个 verilog/ 目录）

   你可以通过 `make submit` 来快速生成一个zip文件；

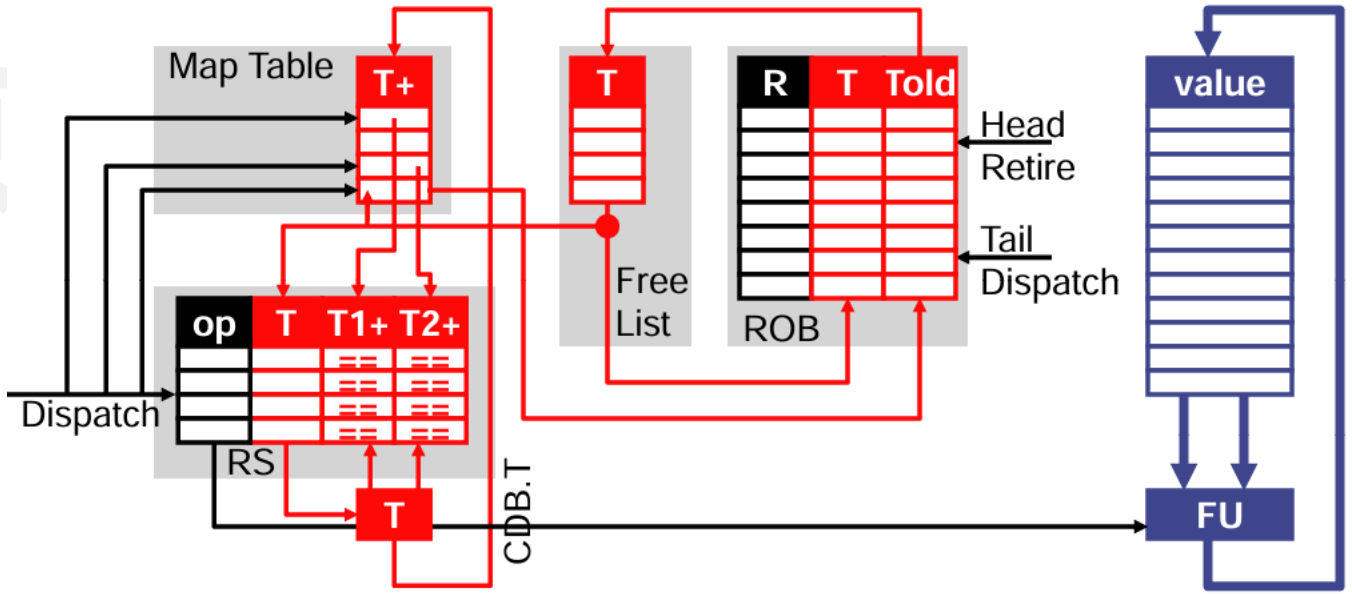
2. 简略的实验报告，包含：
   - 实现思路
   - 你调试时遇到的关键问题及解决方案
   - **注意：报告不超过 2 页**

目 录

CONTENTS

MIPS：超标量+动态指令发射

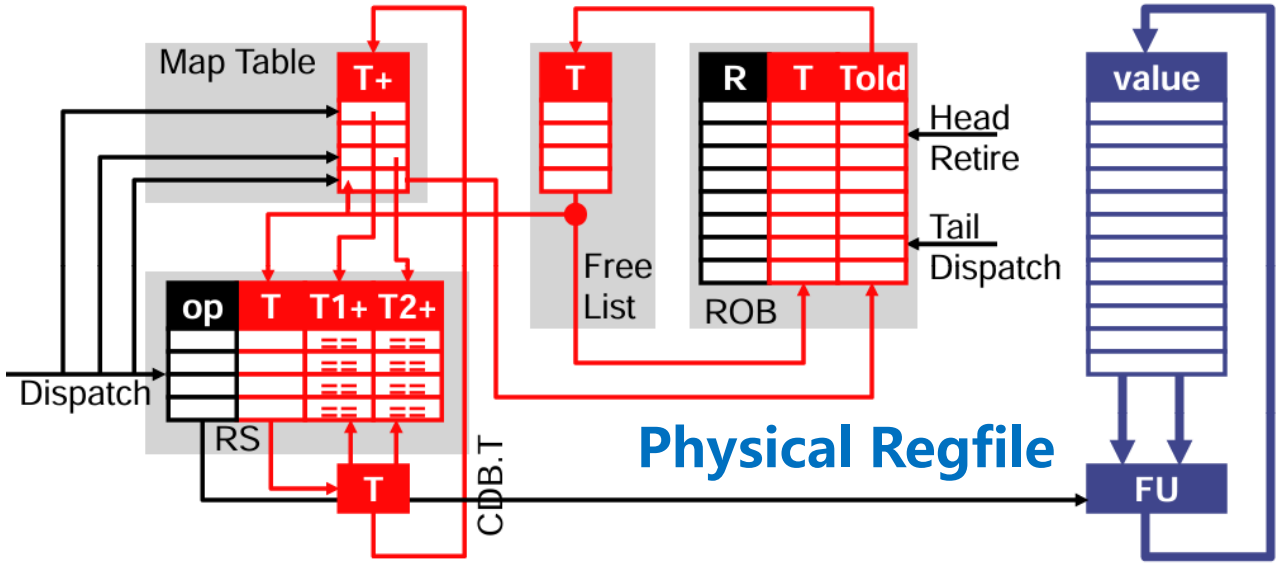
- MIPS：An alternative implementation



- **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

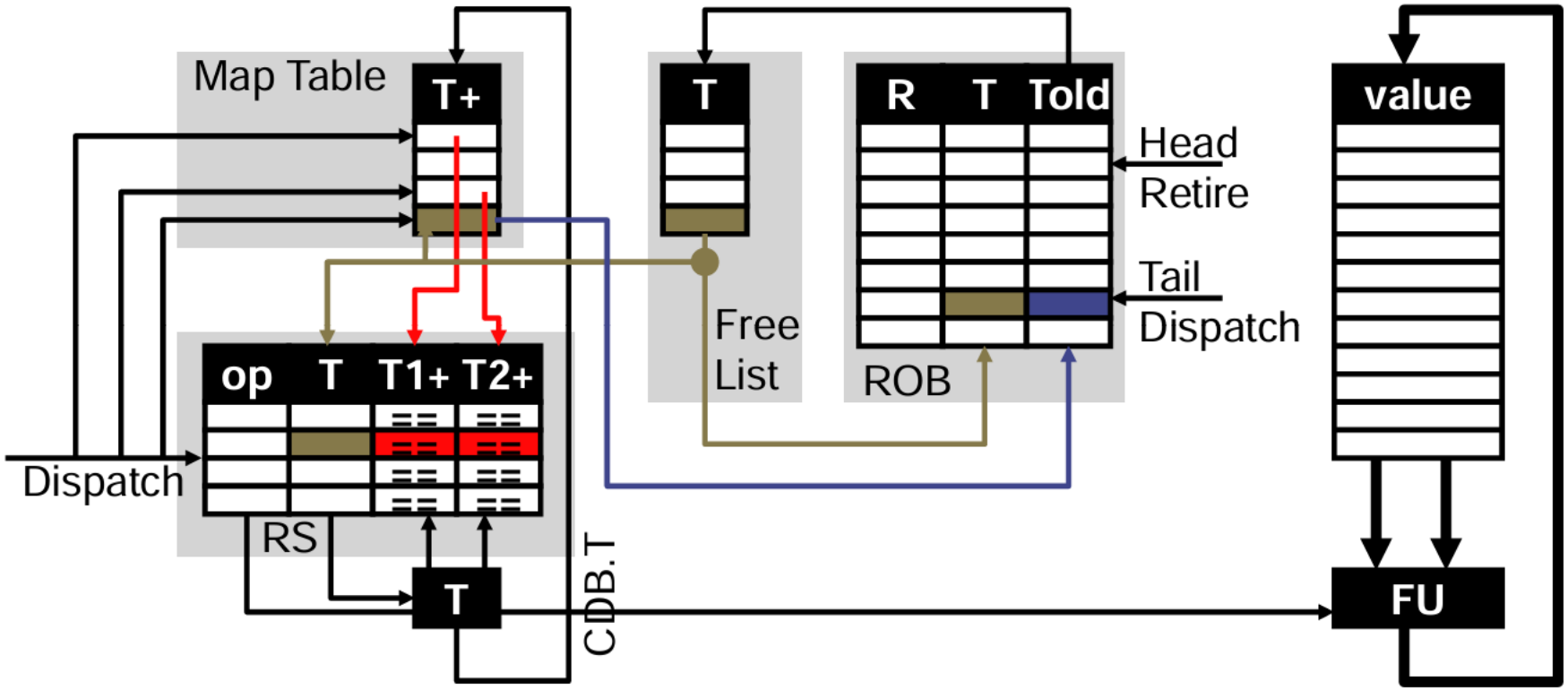


- **架构寄存器文件？ Gone**
- **物理寄存器文件保存所有值**
  - #物理寄存器 = #架构寄存器 + #ROB entries
  - 将架构寄存器映射到物理寄存器
  - 消除数据冲突（物理寄存器取代 RS 副本）

- **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

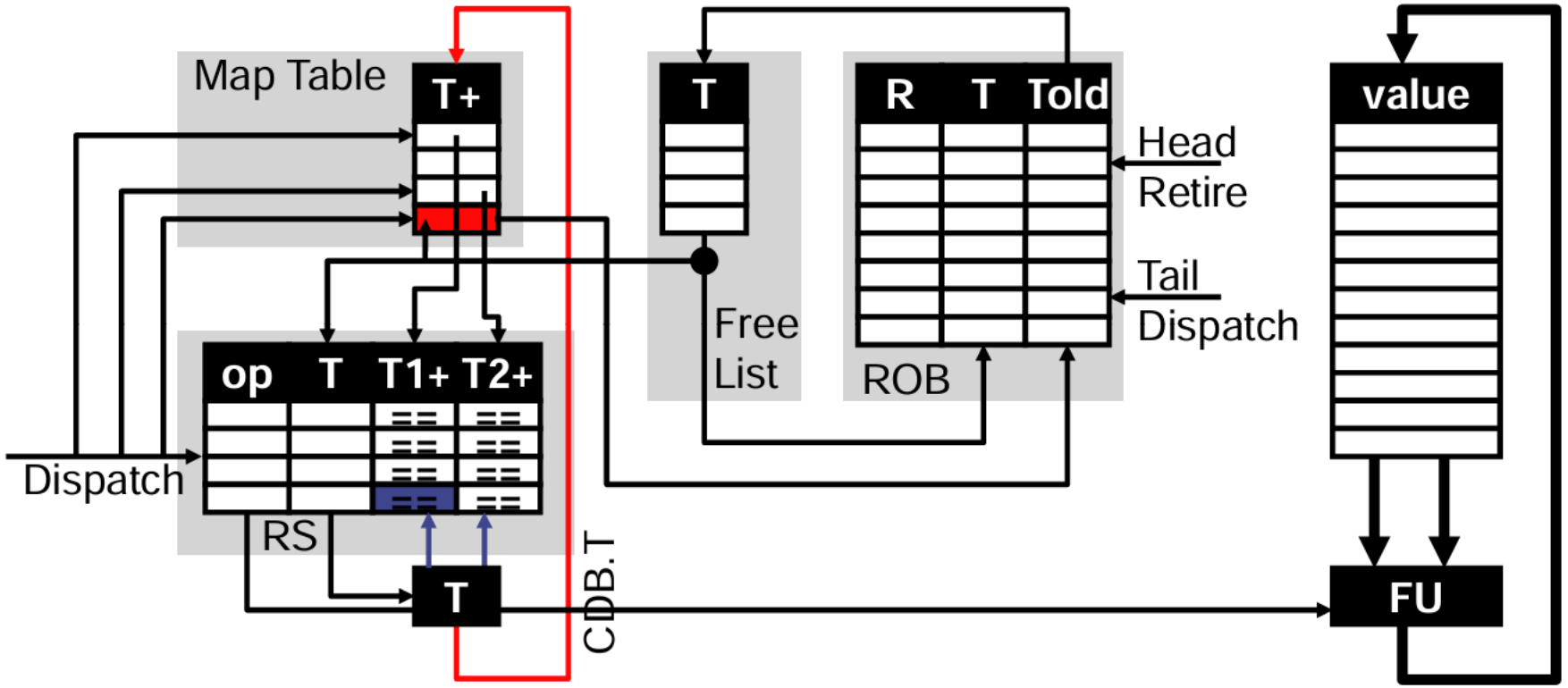
- MIPS R10K Dispatch步骤



- **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

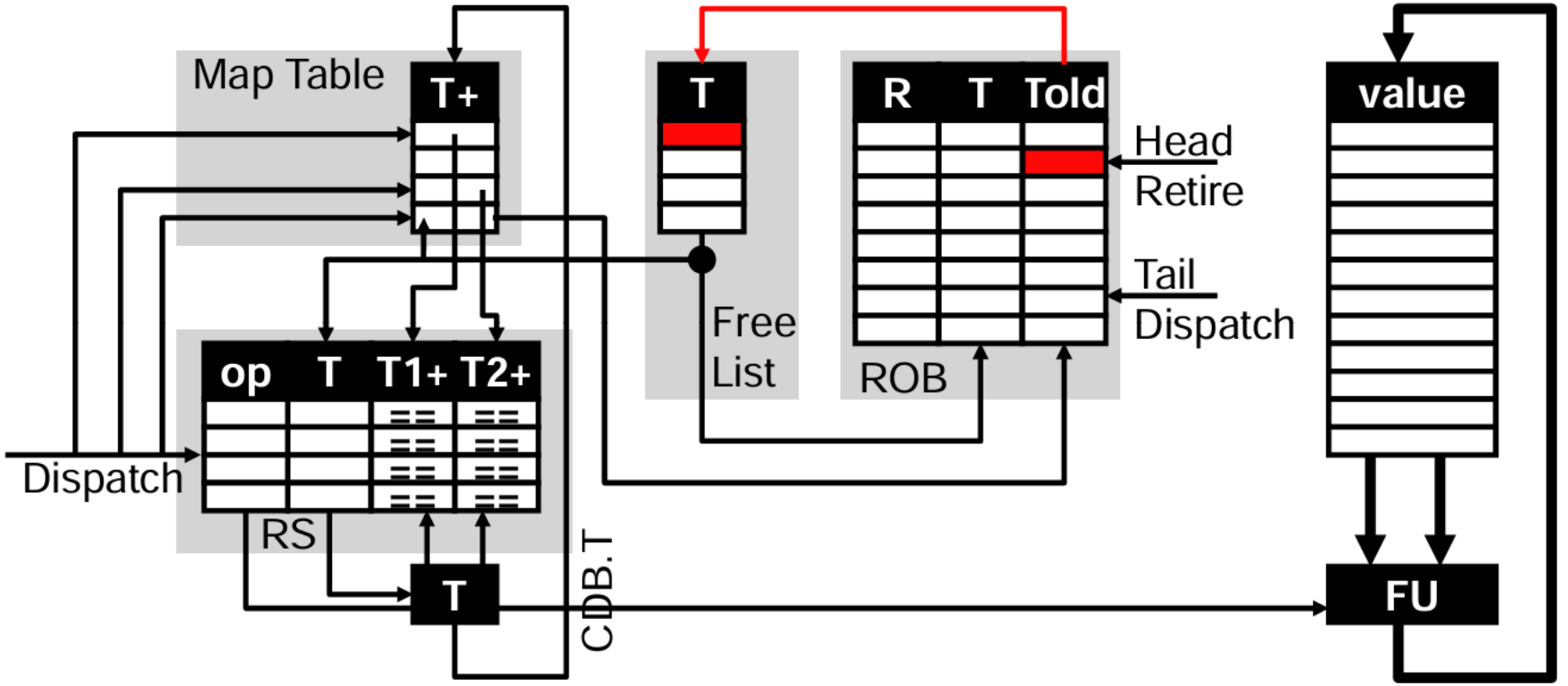
- MIPS R10K Complete步骤



- **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

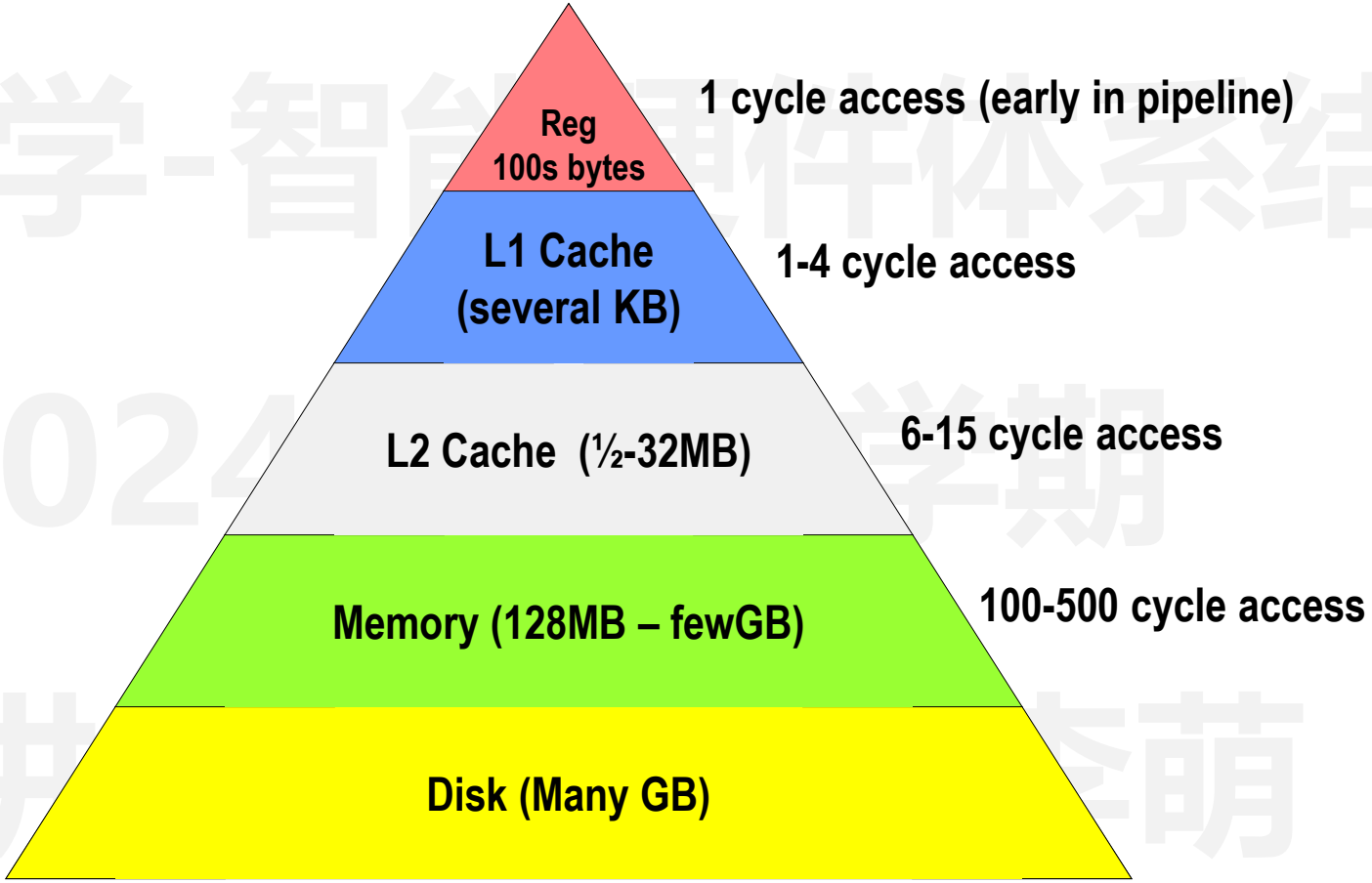
4.D$的写入如何撤销?

目 录

CONTENTS

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：

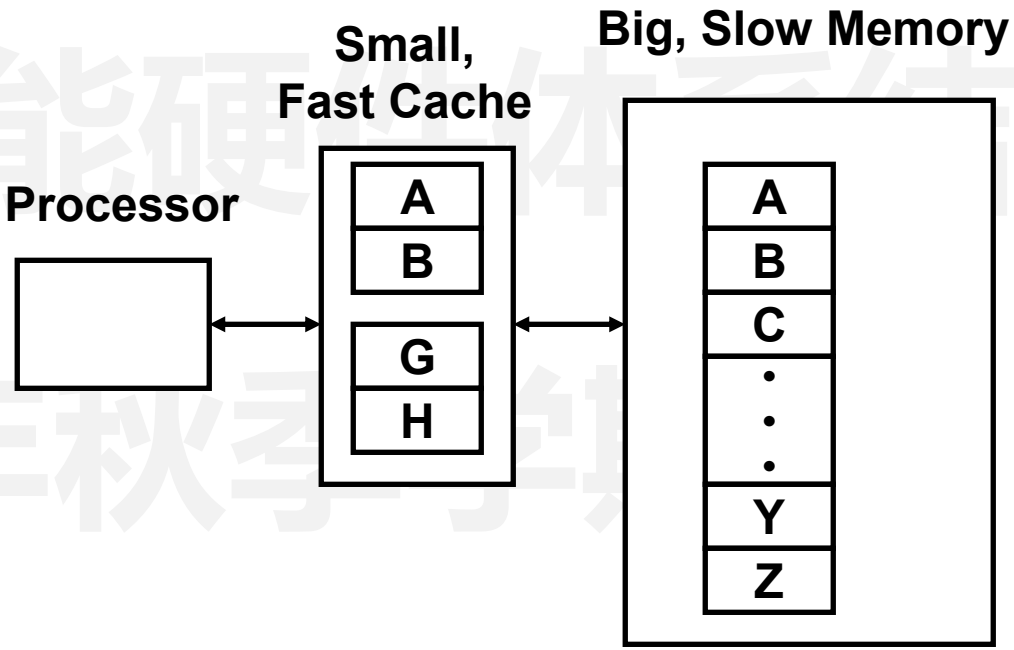
- **数据**
  - 连续引用数组元素（空间）
- **指令**
  - 按顺序（空间）引用指令
  - 反复循环（时间）

```
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

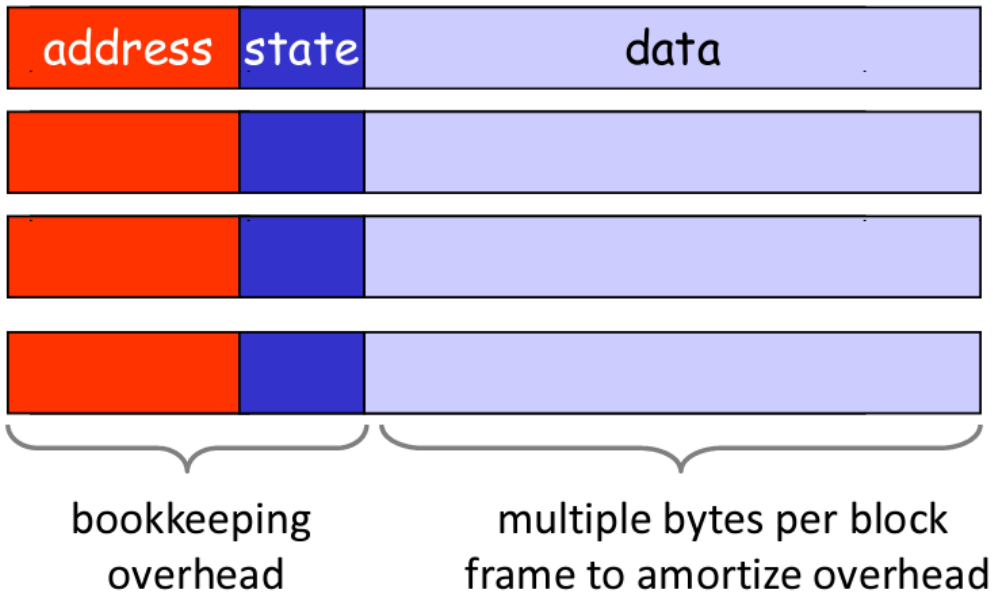


缓存Cache的基本思路

- 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

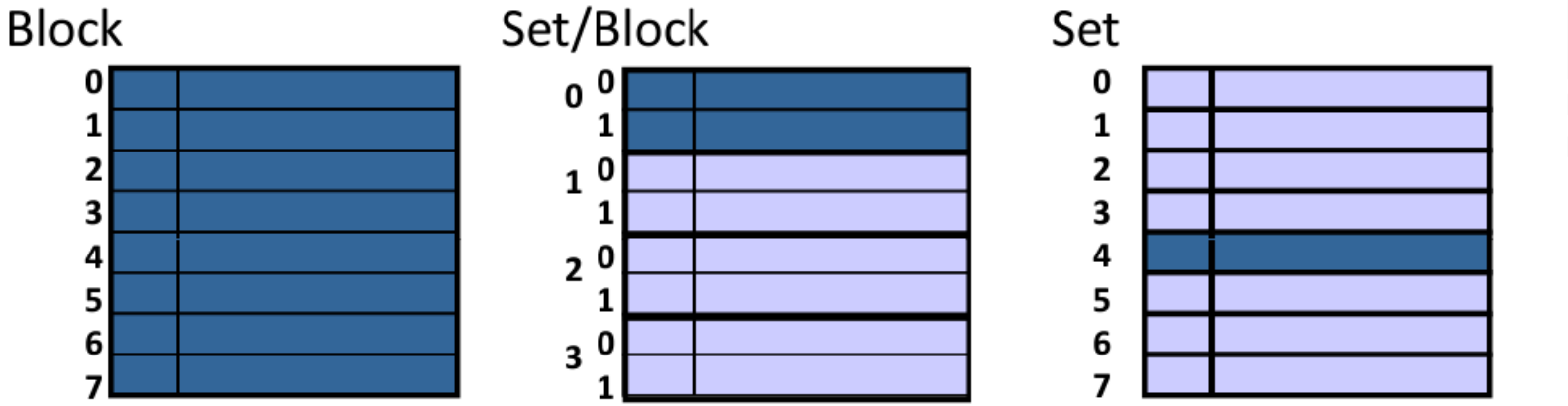
北京大学 PEKING UNIVERSITY

- 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**

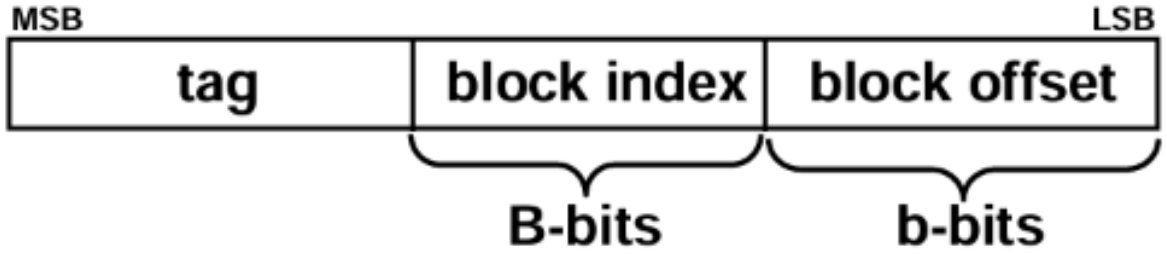
Where does block 12 (b'1100) go?



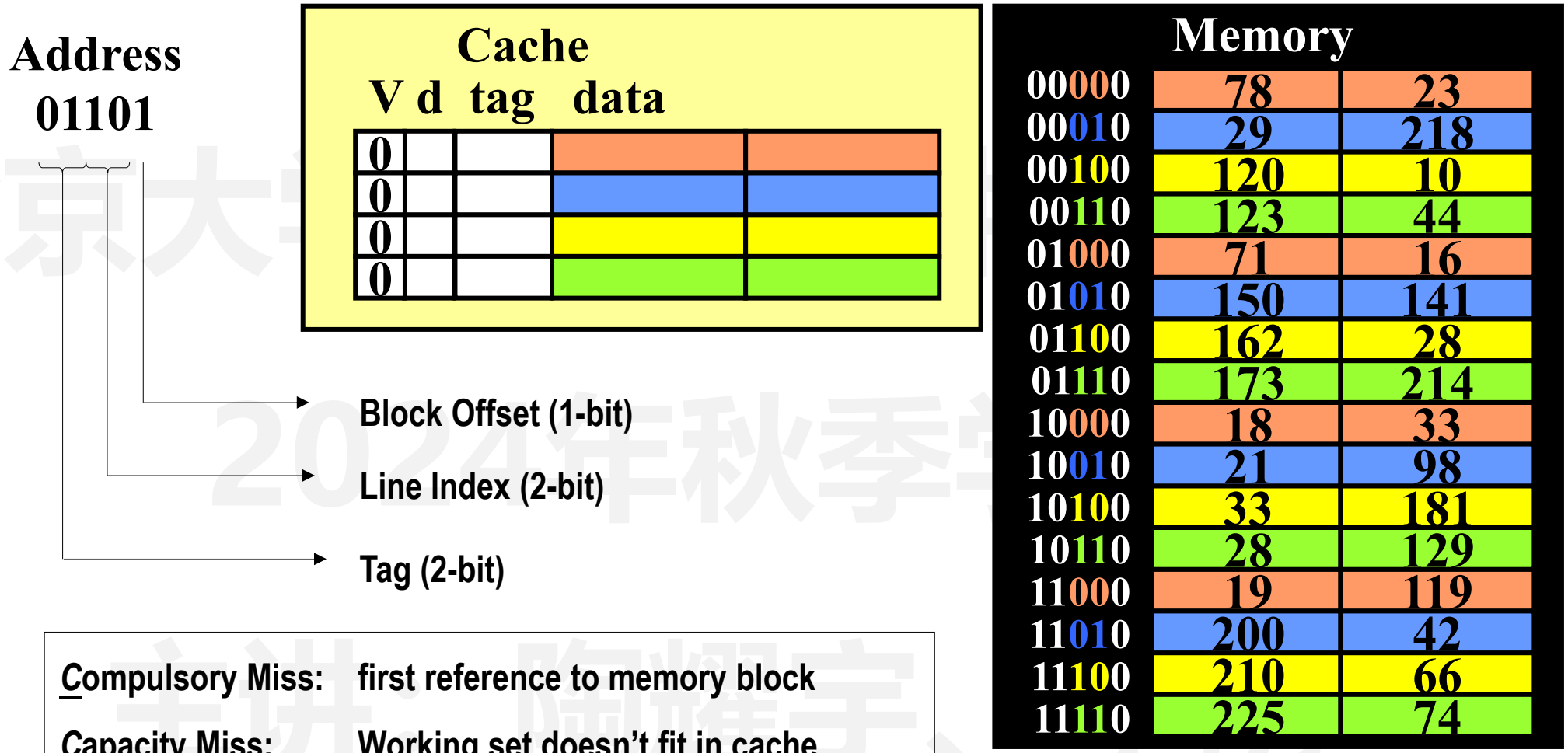
Fully-associative	Set-associative	Direct-mapped
block goes in any frame	a block goes in any frame in exactly one set	block goes in exactly one frame
(think all frames in 1 set)	(frames grouped into sets)	(think 1 frame per set)

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)



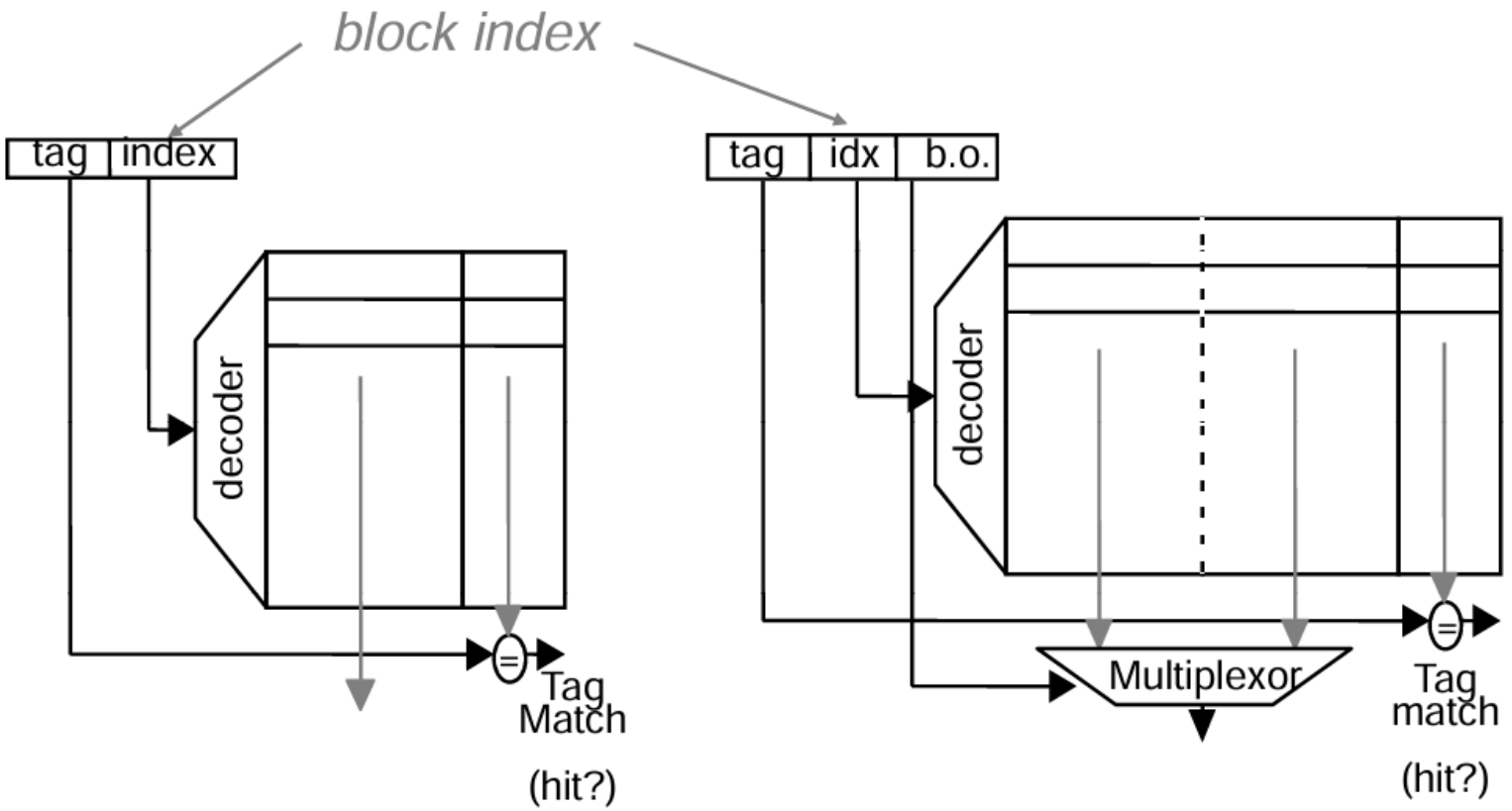
Direct-Mapped Cache设计



3-C's

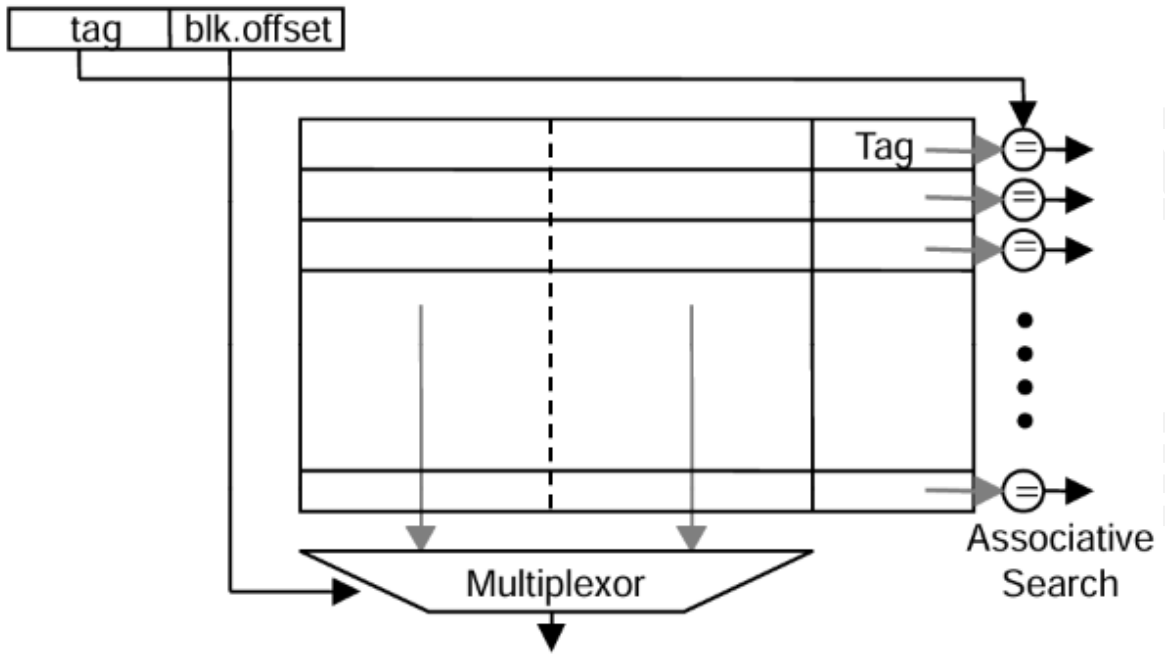
Compulsory Miss:	first reference to memory block
Capacity Miss:	Working set doesn't fit in cache
Conflict Miss:	Working set maps to same cache line

Direct-Mapped Cache设计



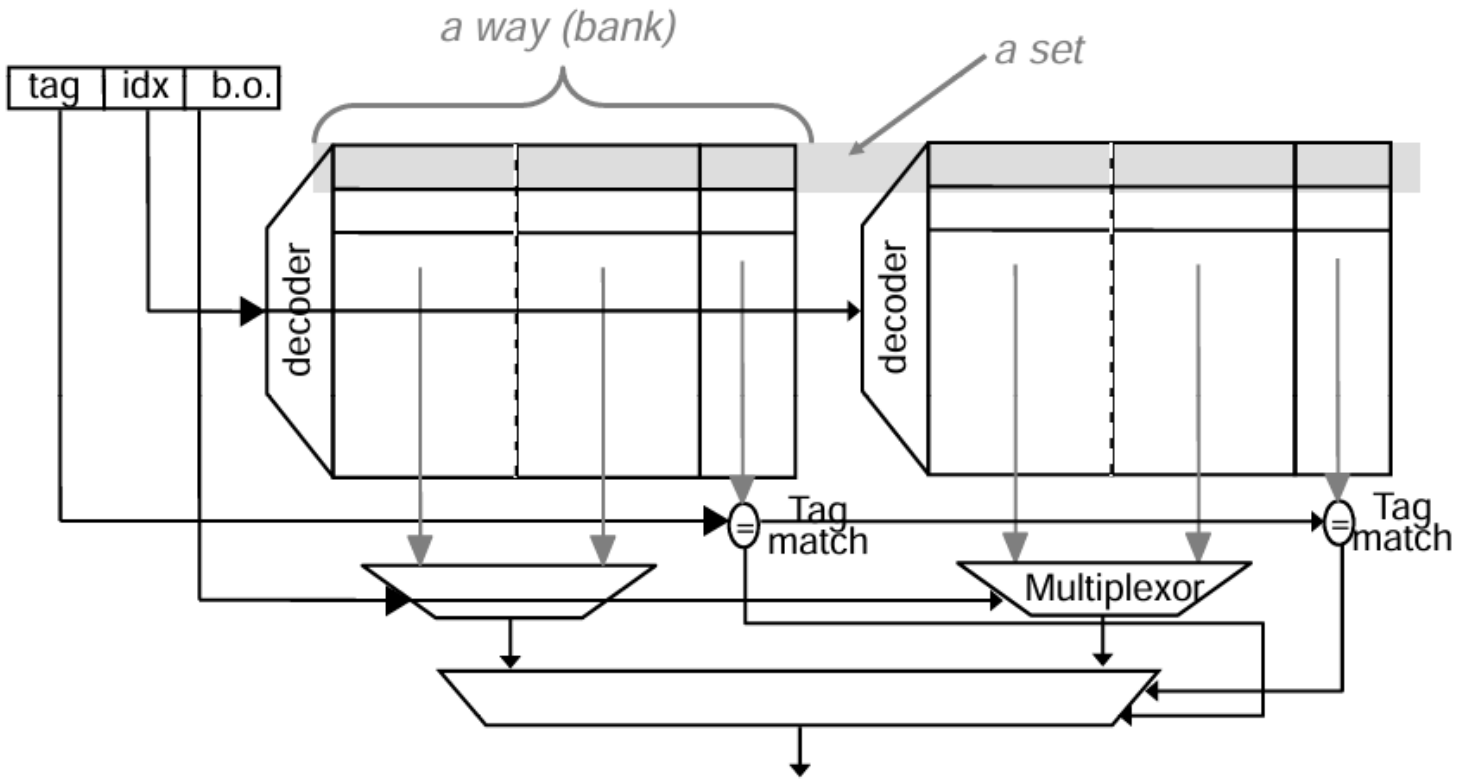
Don't forget to check the valid/state bits

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**
  - 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?

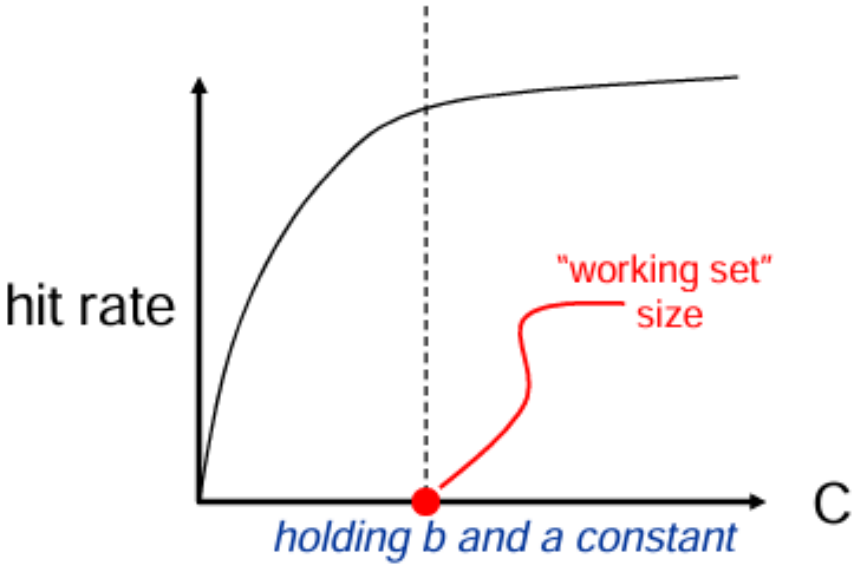
Cache Miss种类

- 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）

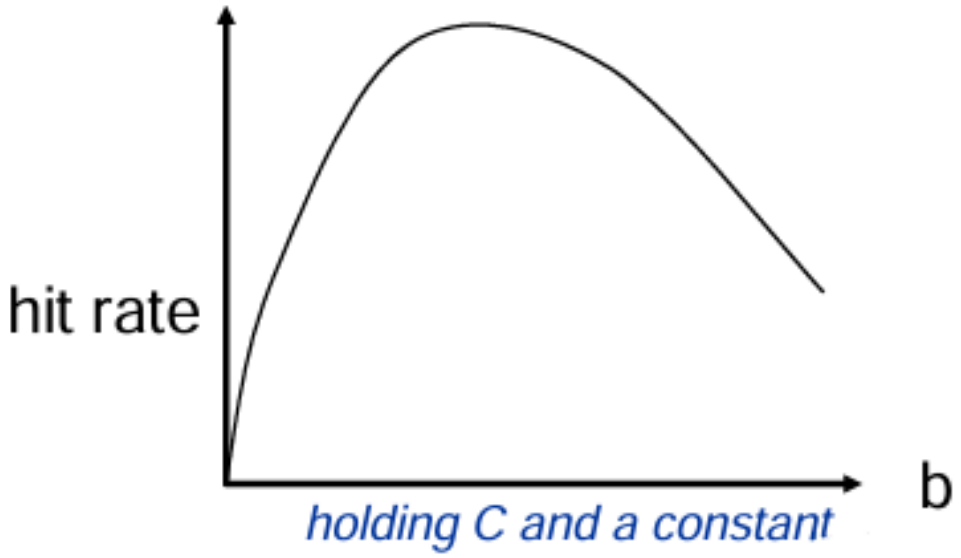
- **缓存大小是总数据（不包括标签）容量**
  - 更大可以更好地利用时间局部性
  - not ALWAYS better
- **缓存太大**
  - 越小越快 => 越大越慢
  - 访问时间可能会降低关键路径
- **缓存太小**
  - 没有很好地利用时间局部性
  - 有用的数据不断被替换



Cache参数的选择

- Block Size（b）

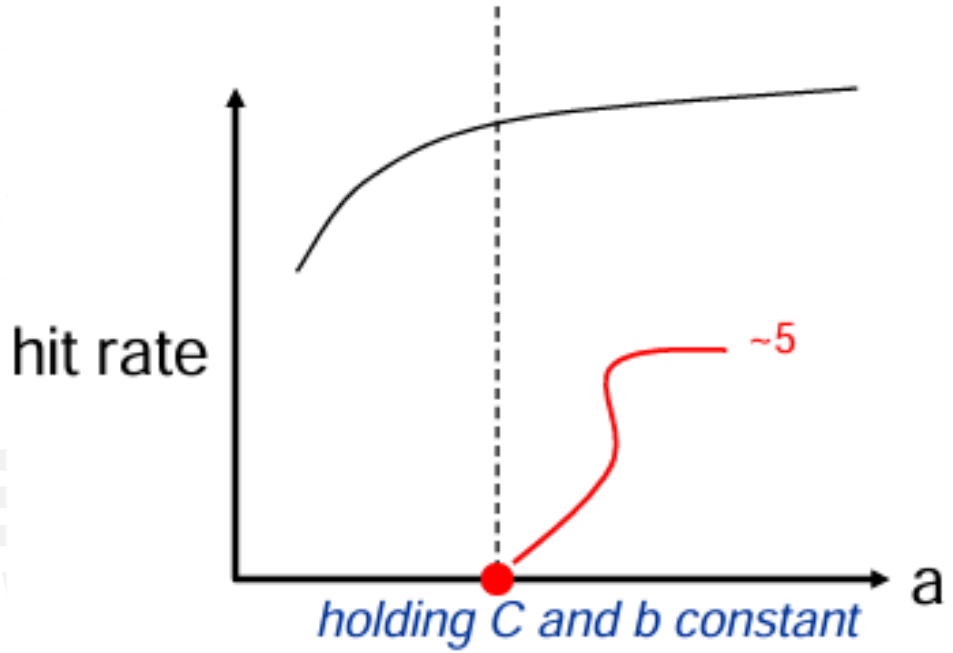
- **块大小是指**
  - 与地址标签关联
  - 不一定是层次结构之间的传输单位（remember sub-blocking）
- **区块太小**
  - 没有很好地利用空间局部性
  - 有过多的标签开销
- **块太大**
  - 传输了无用的数据
  - 有用数据被频繁替换
  - 总区块数太少



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



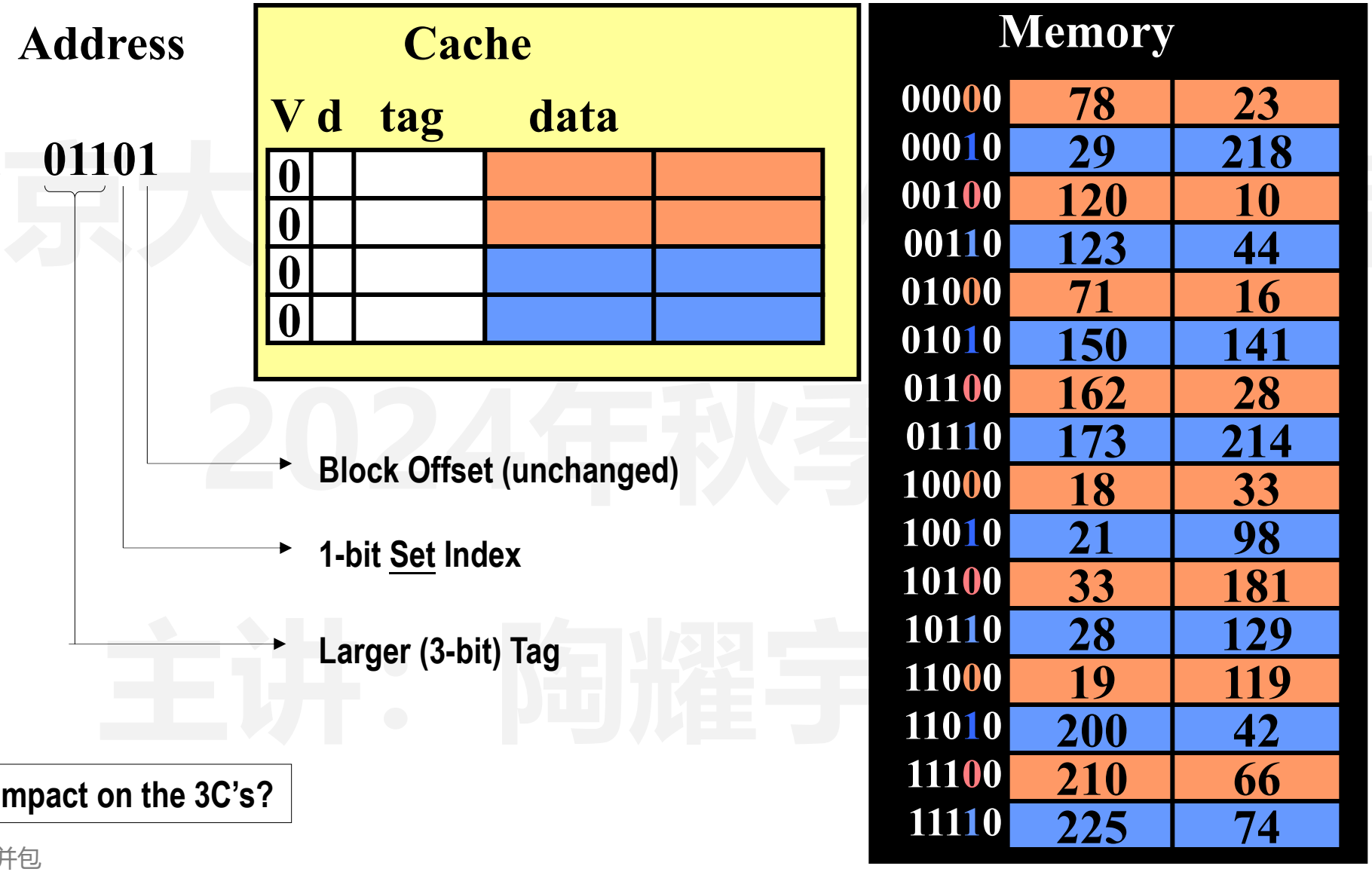
Cache设计选择的影响

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

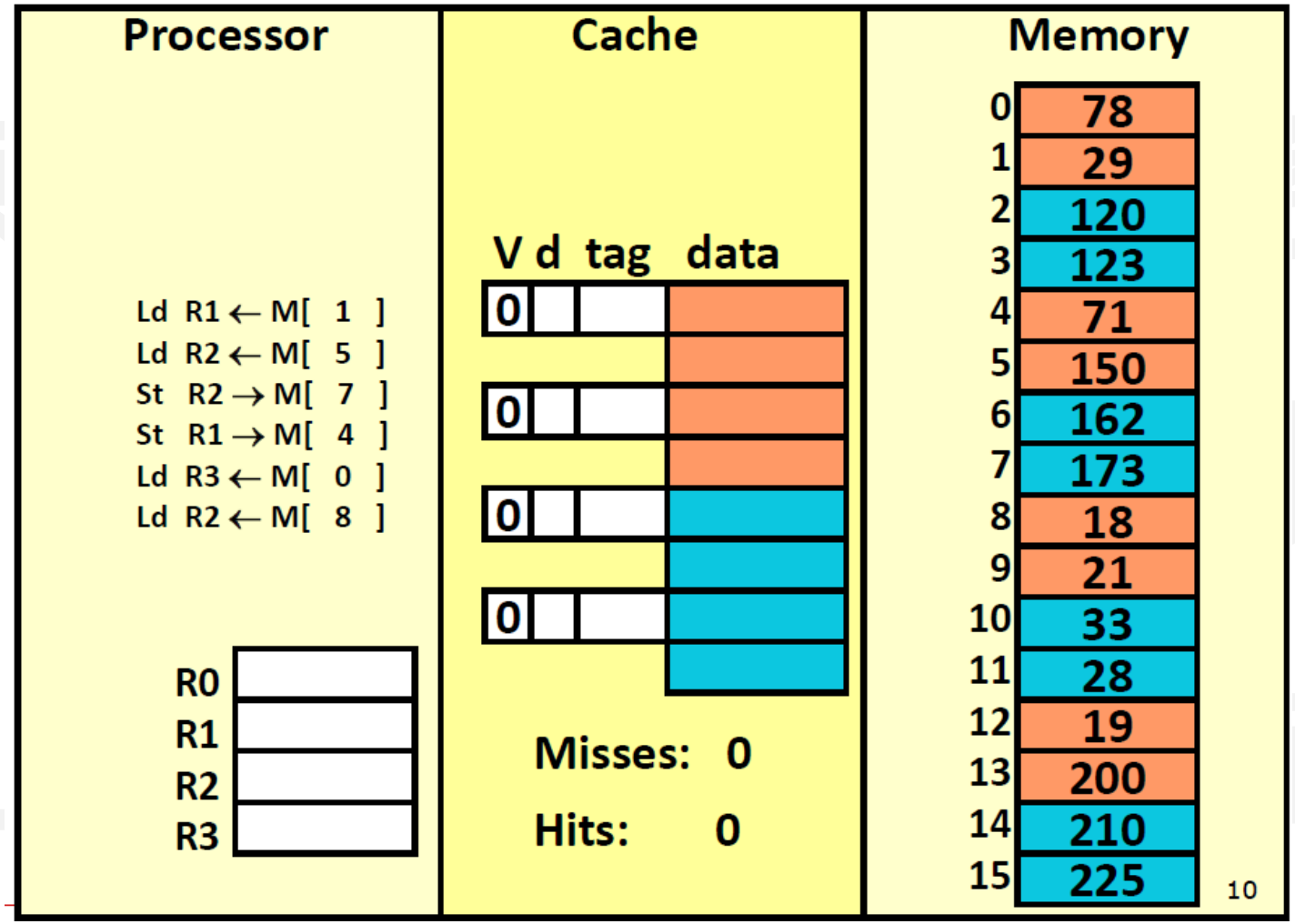
2-Way Set Associative Cache设计



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

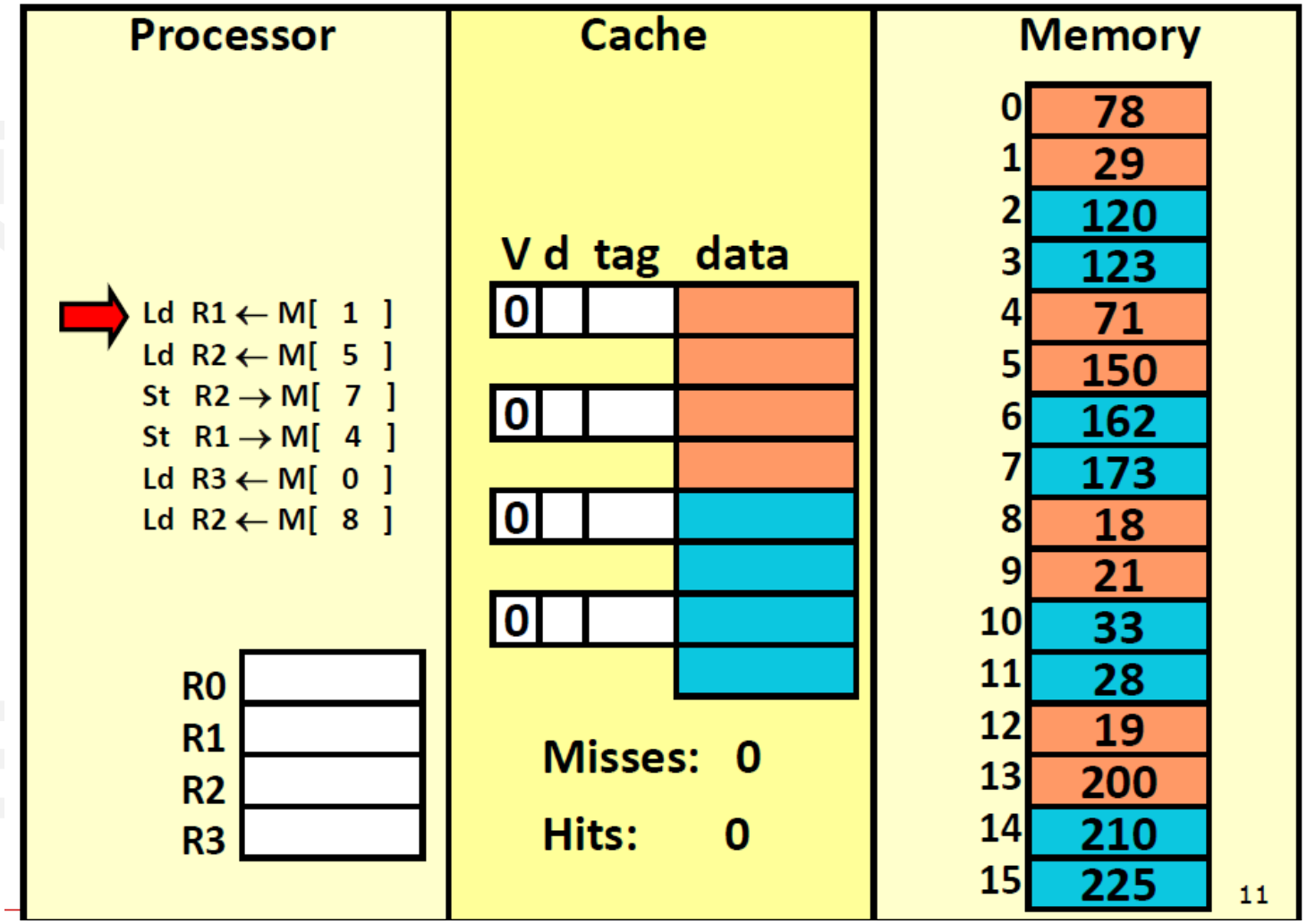
2-Way Set Associative Cache实例

- Write-back & Write-allocate



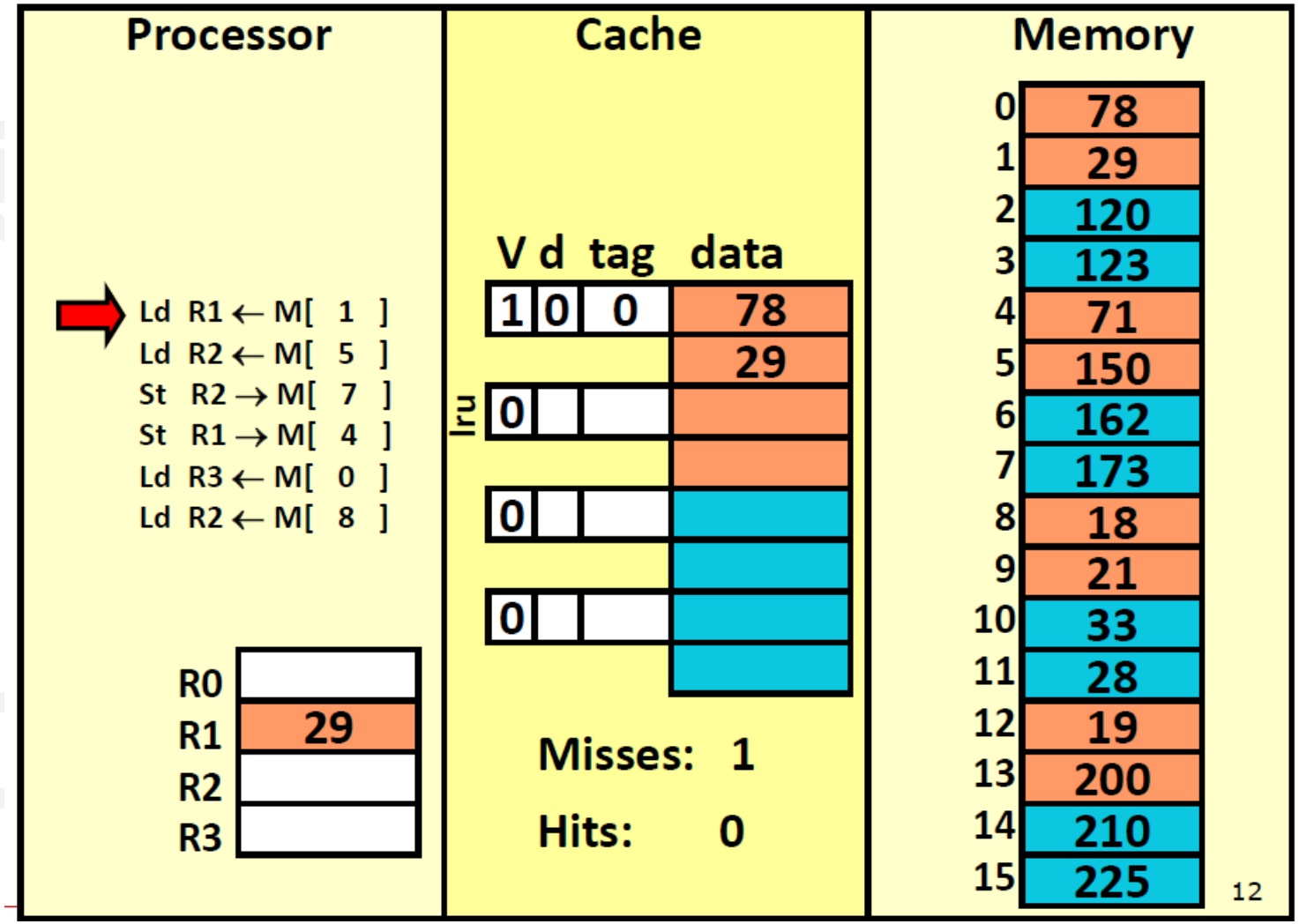
2-Way Set Associative Cache实例

- Write-back & Write-allocate



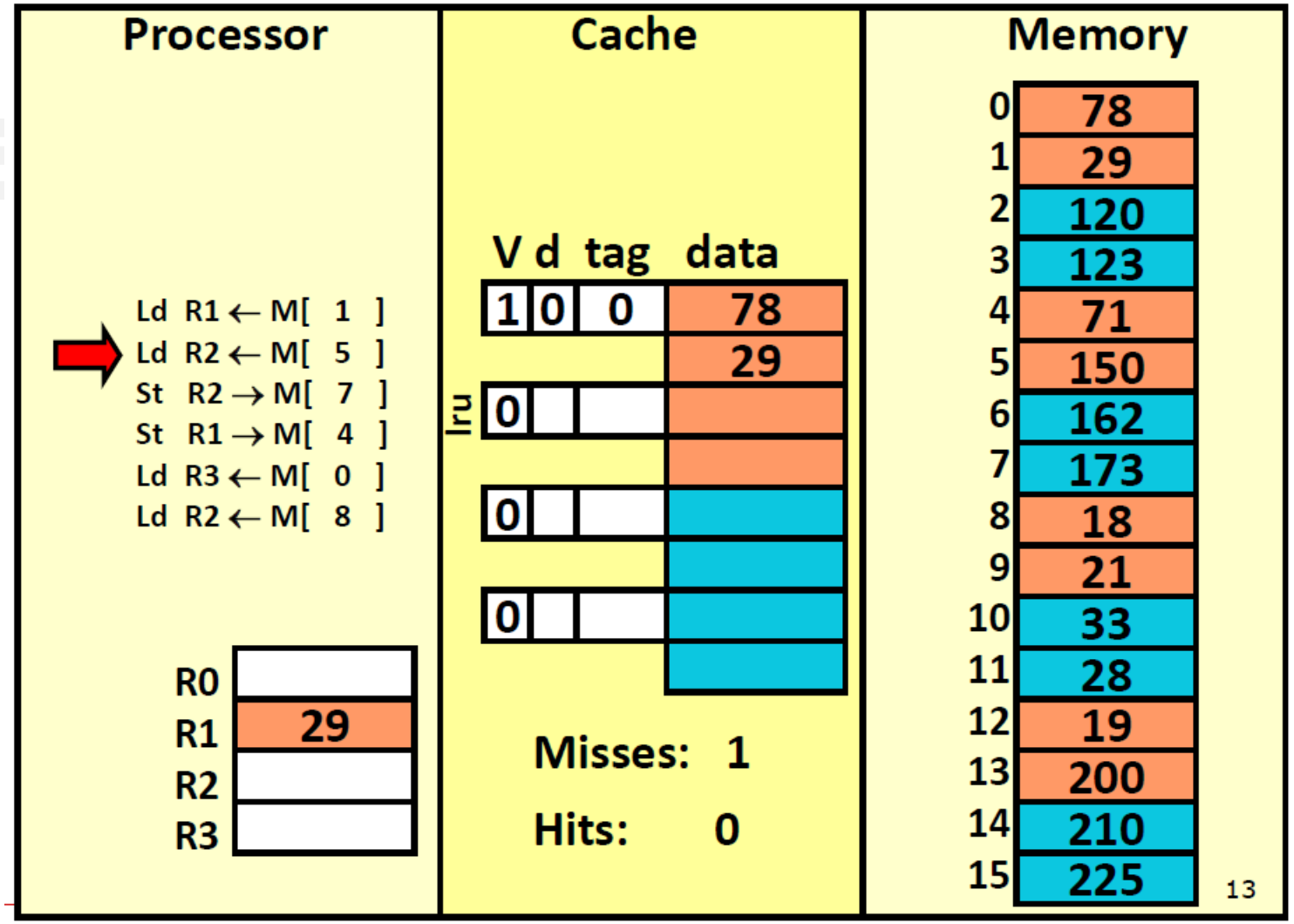
2-Way Set Associative Cache实例

- Write-back & Write-allocate



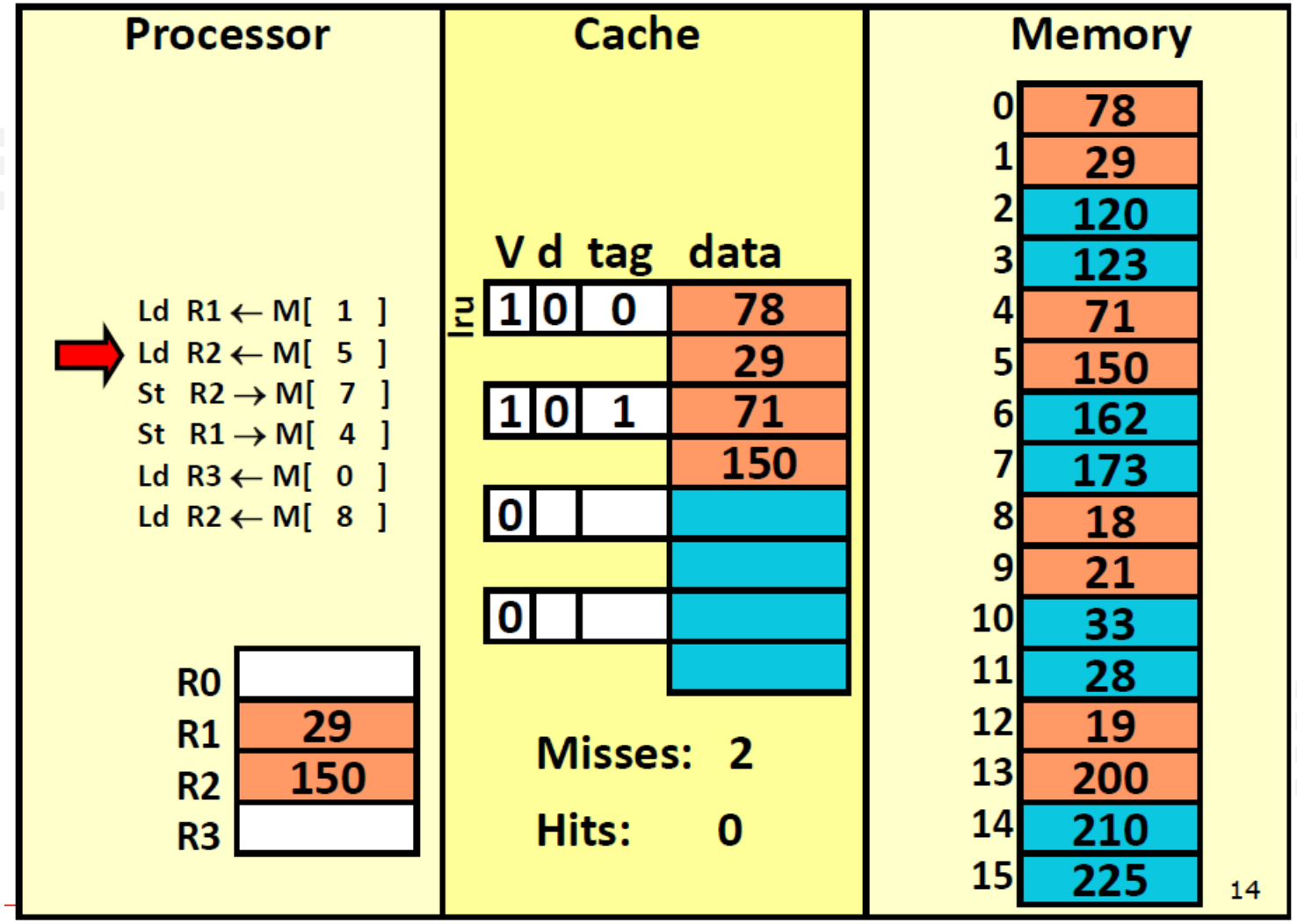
2-Way Set Associative Cache实例

- Write-back & Write-allocate



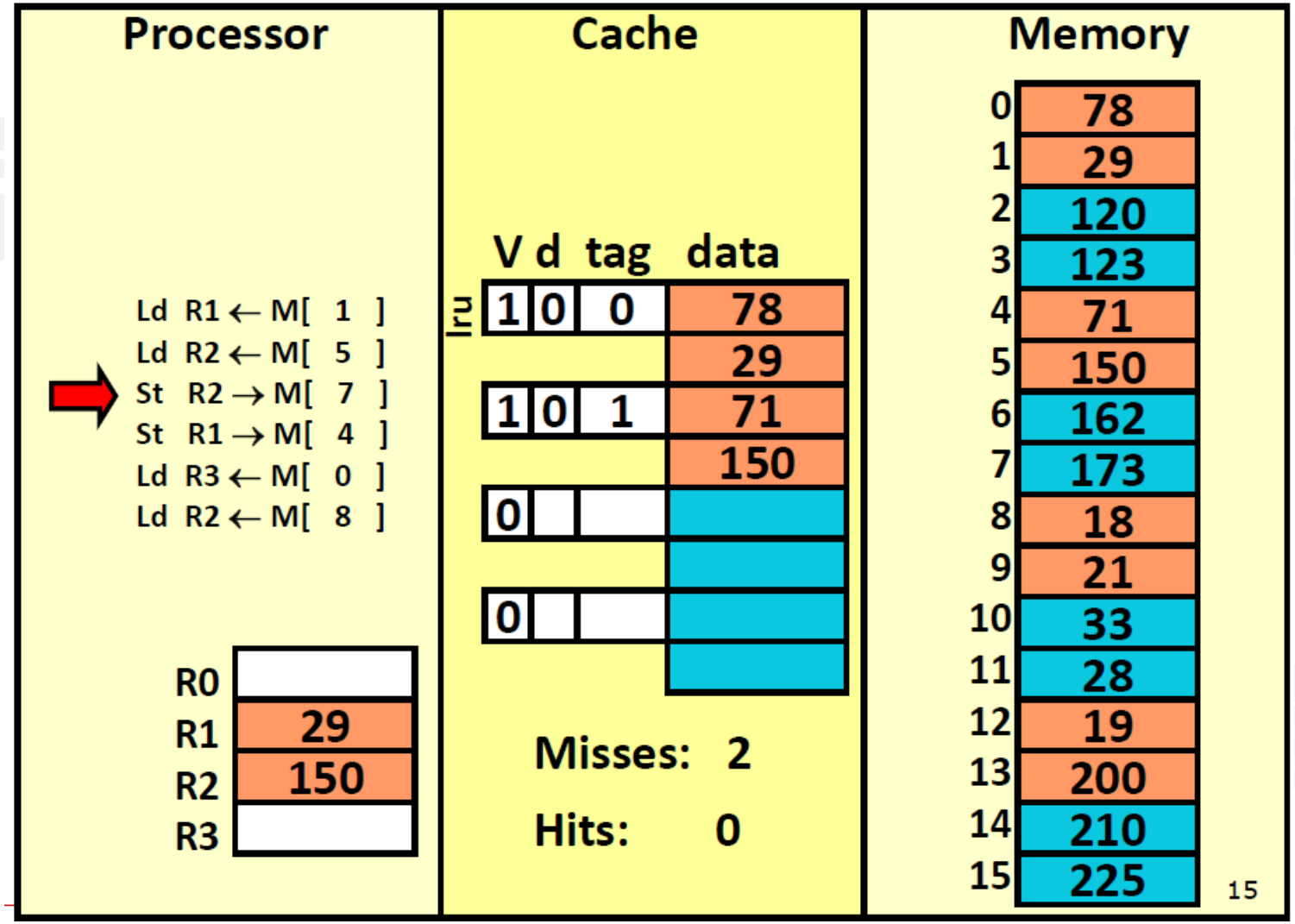
2-Way Set Associative Cache实例

- Write-back & Write-allocate



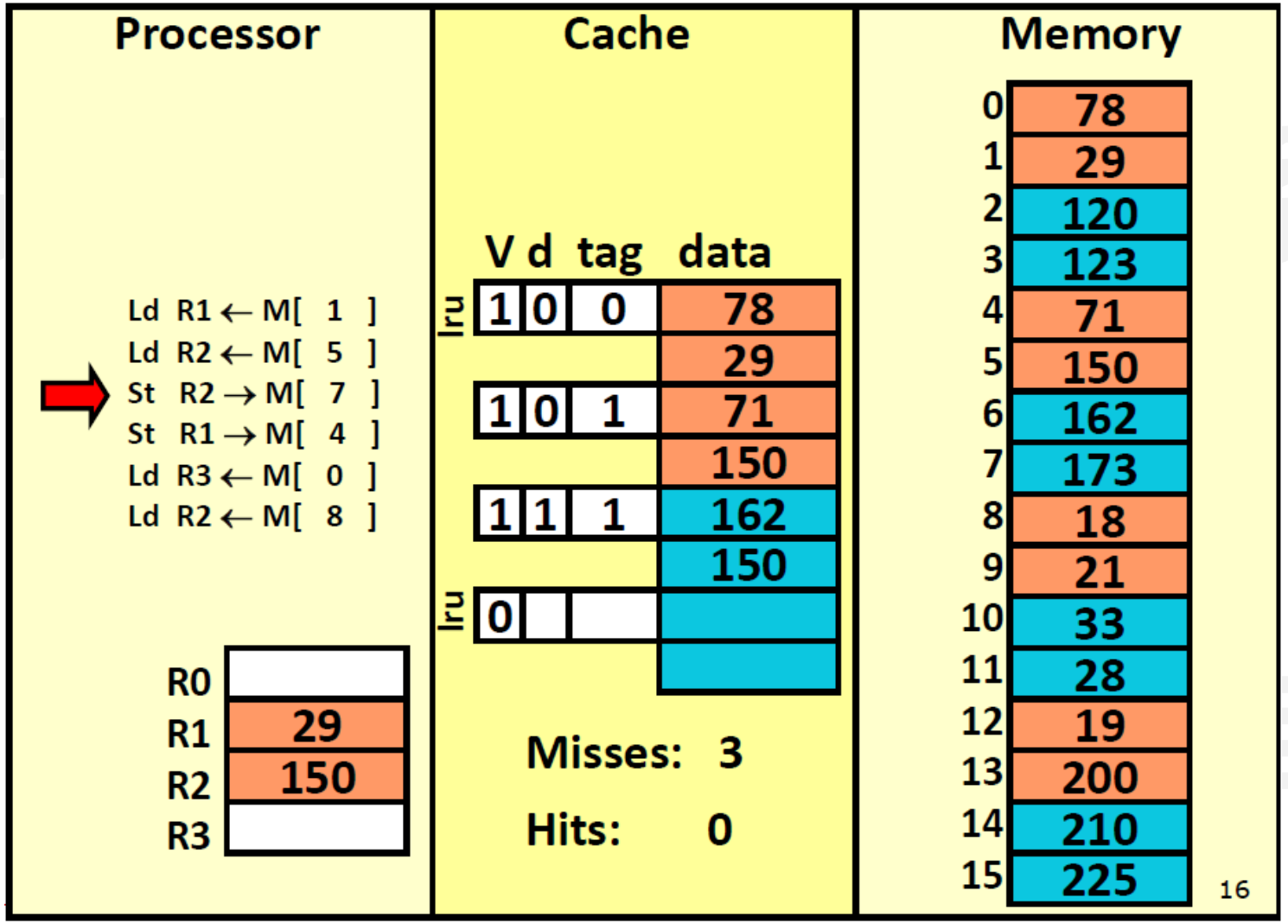
2-Way Set Associative Cache实例

- Write-back & Write-allocate



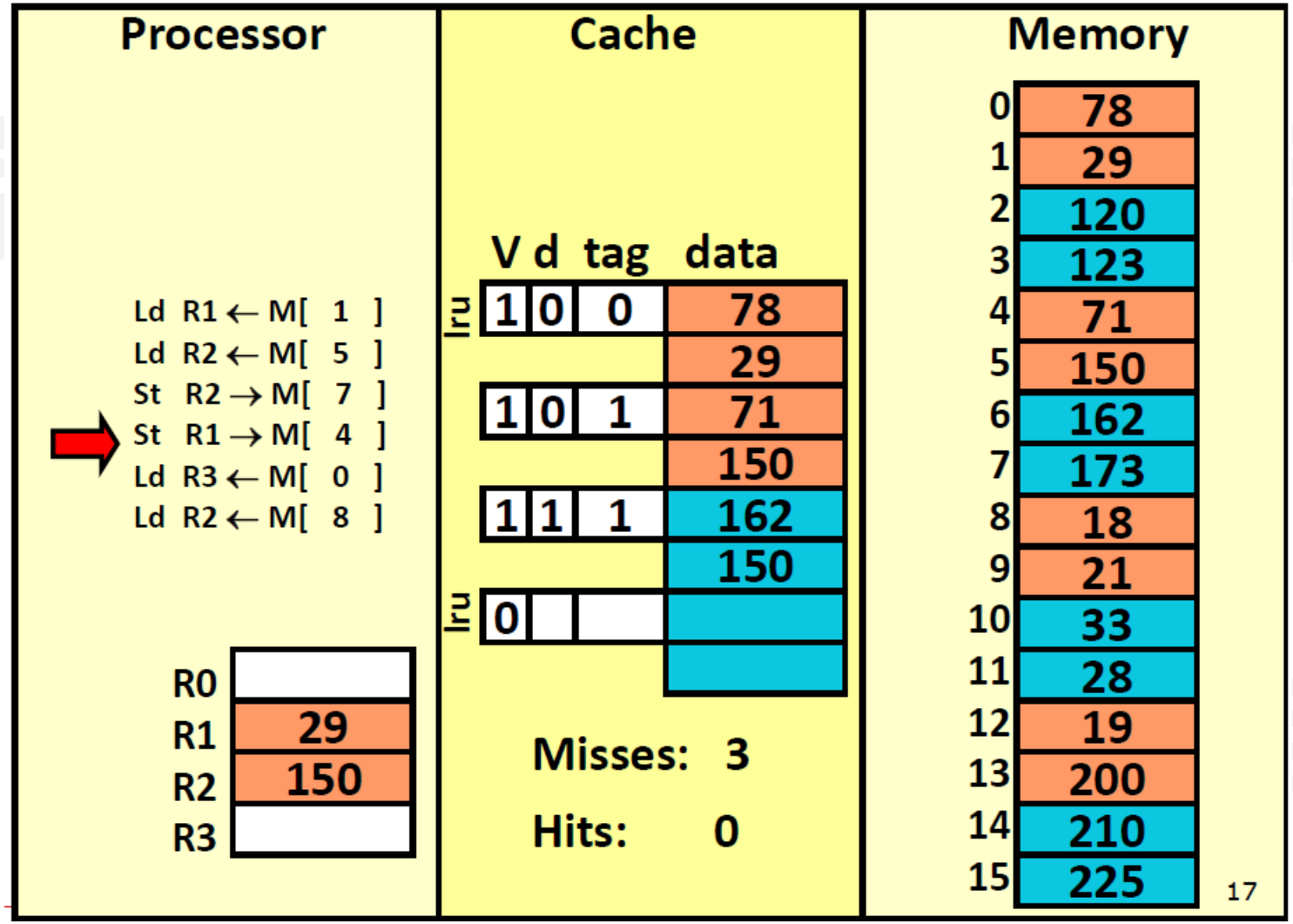
2-Way Set Associative Cache实例

- Write-back & Write-allocate



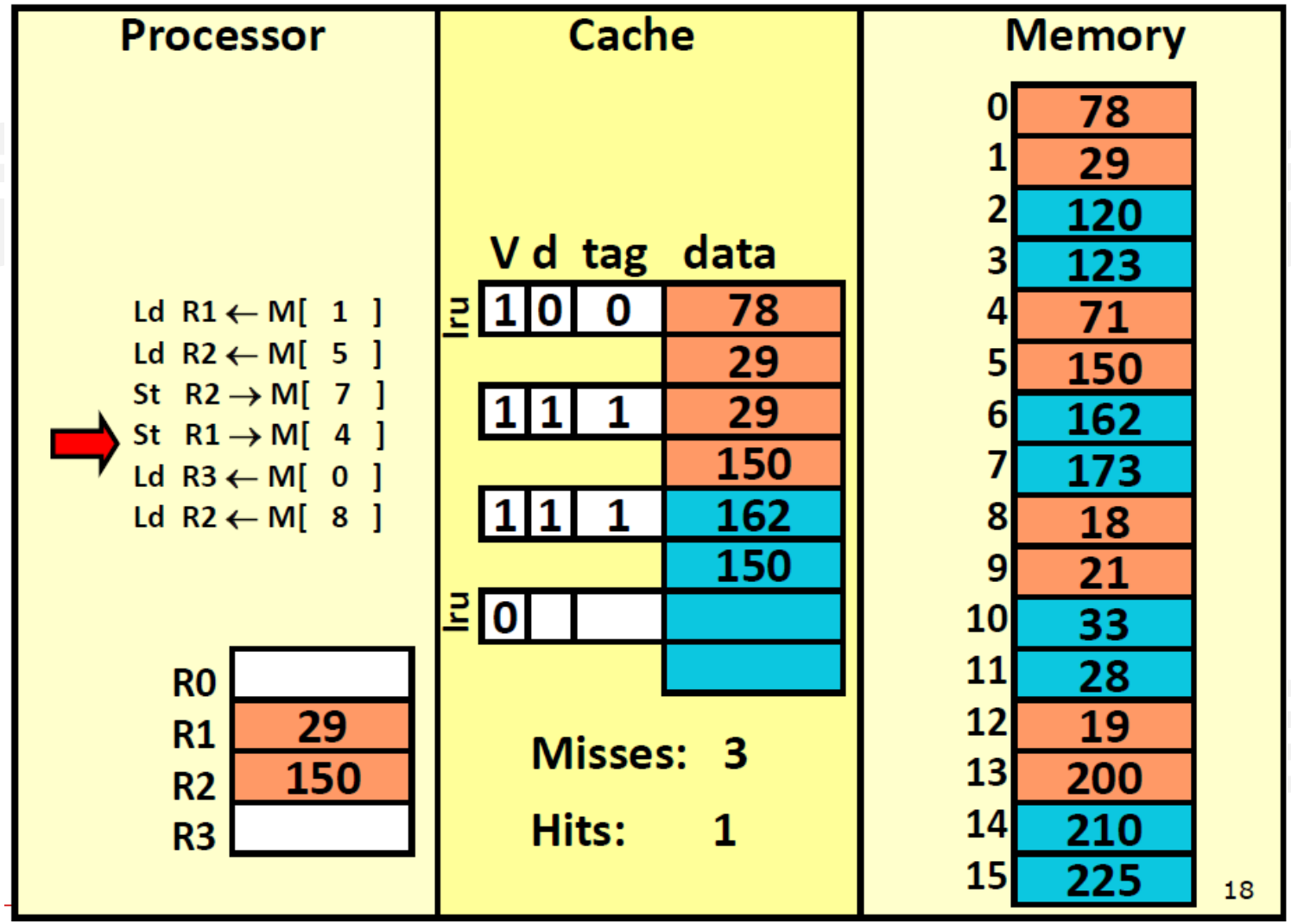
2-Way Set Associative Cache实例

- Write-back & Write-allocate



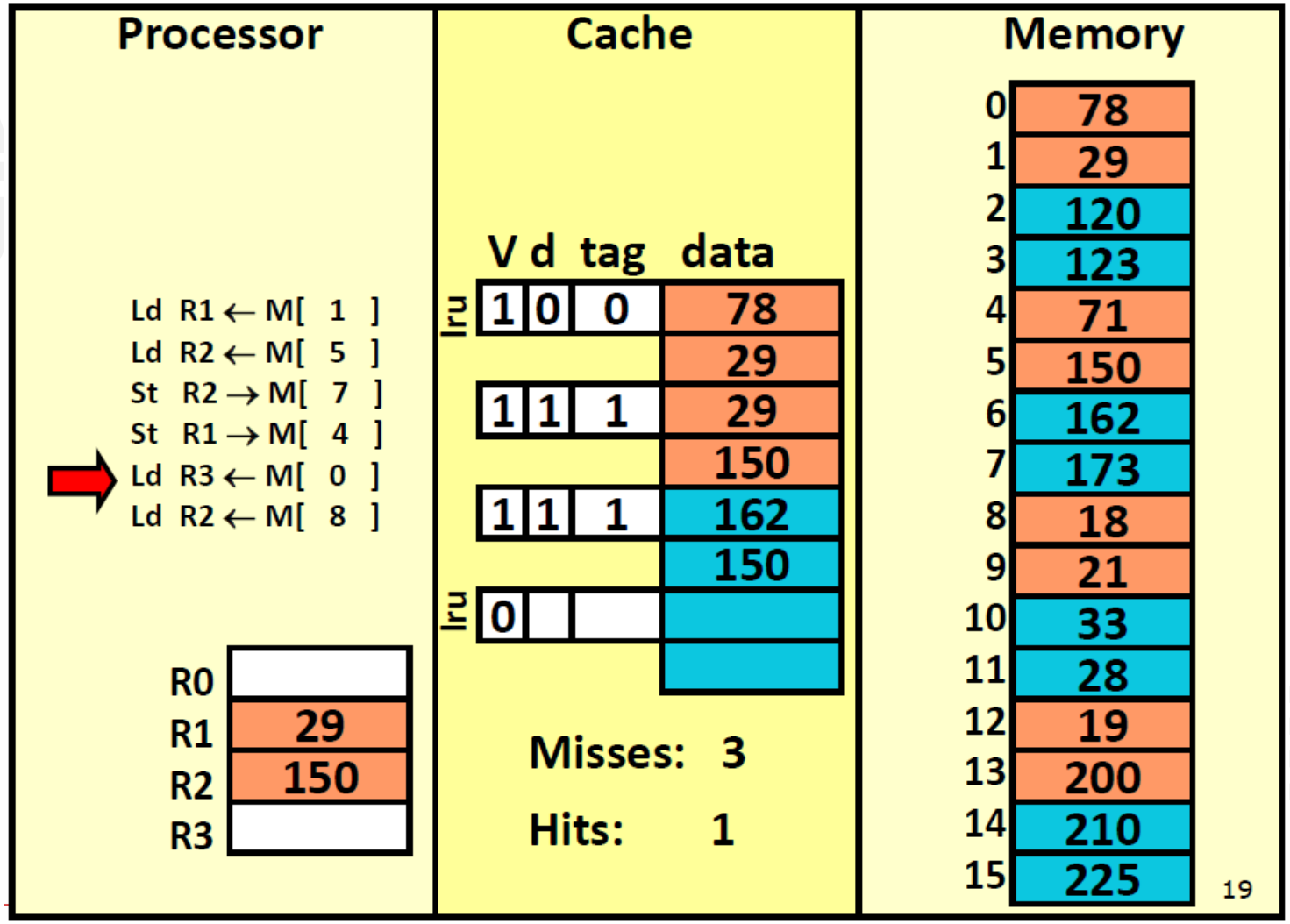
2-Way Set Associative Cache实例

- Write-back & Write-allocate



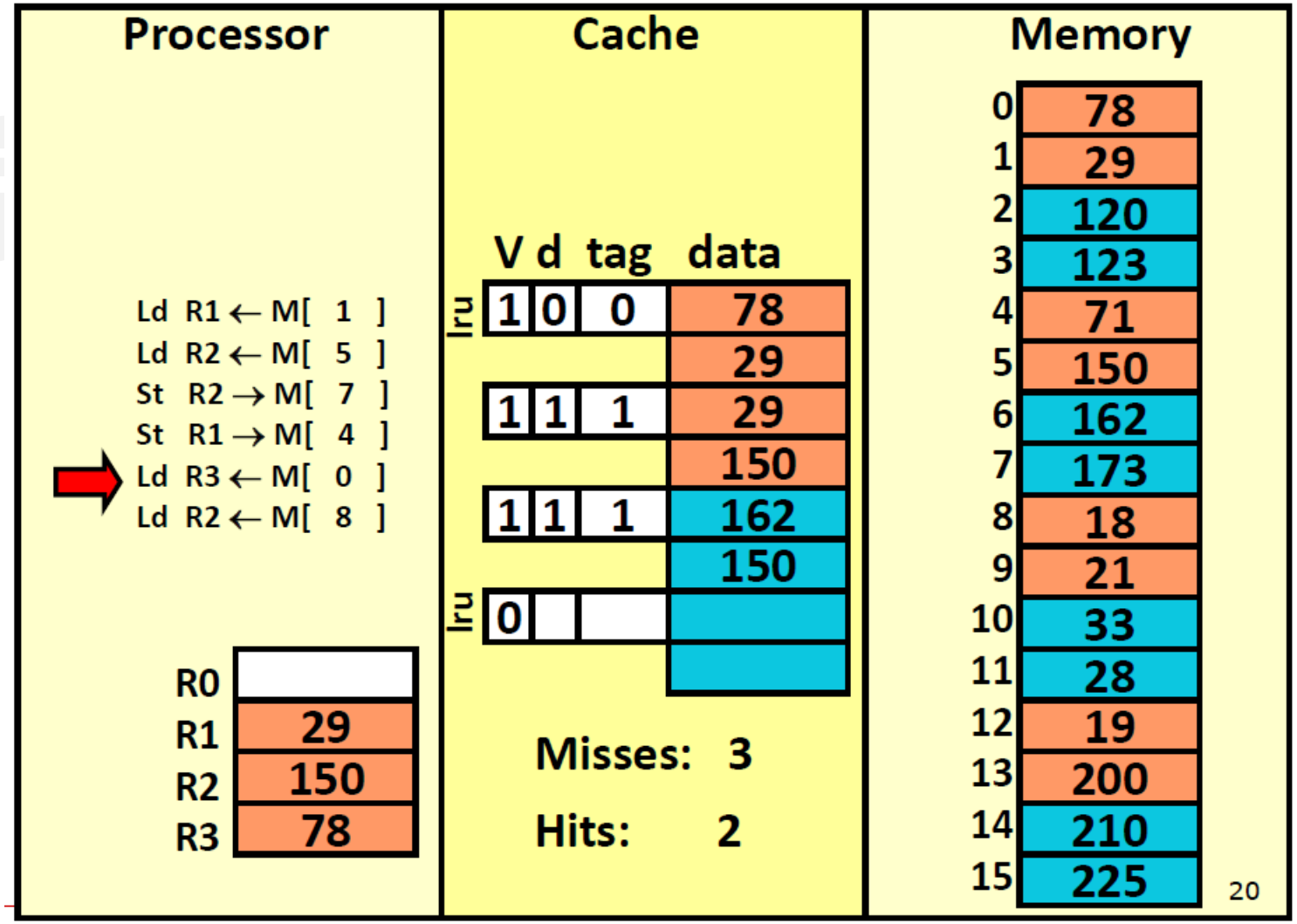
2-Way Set Associative Cache实例

- Write-back & Write-allocate



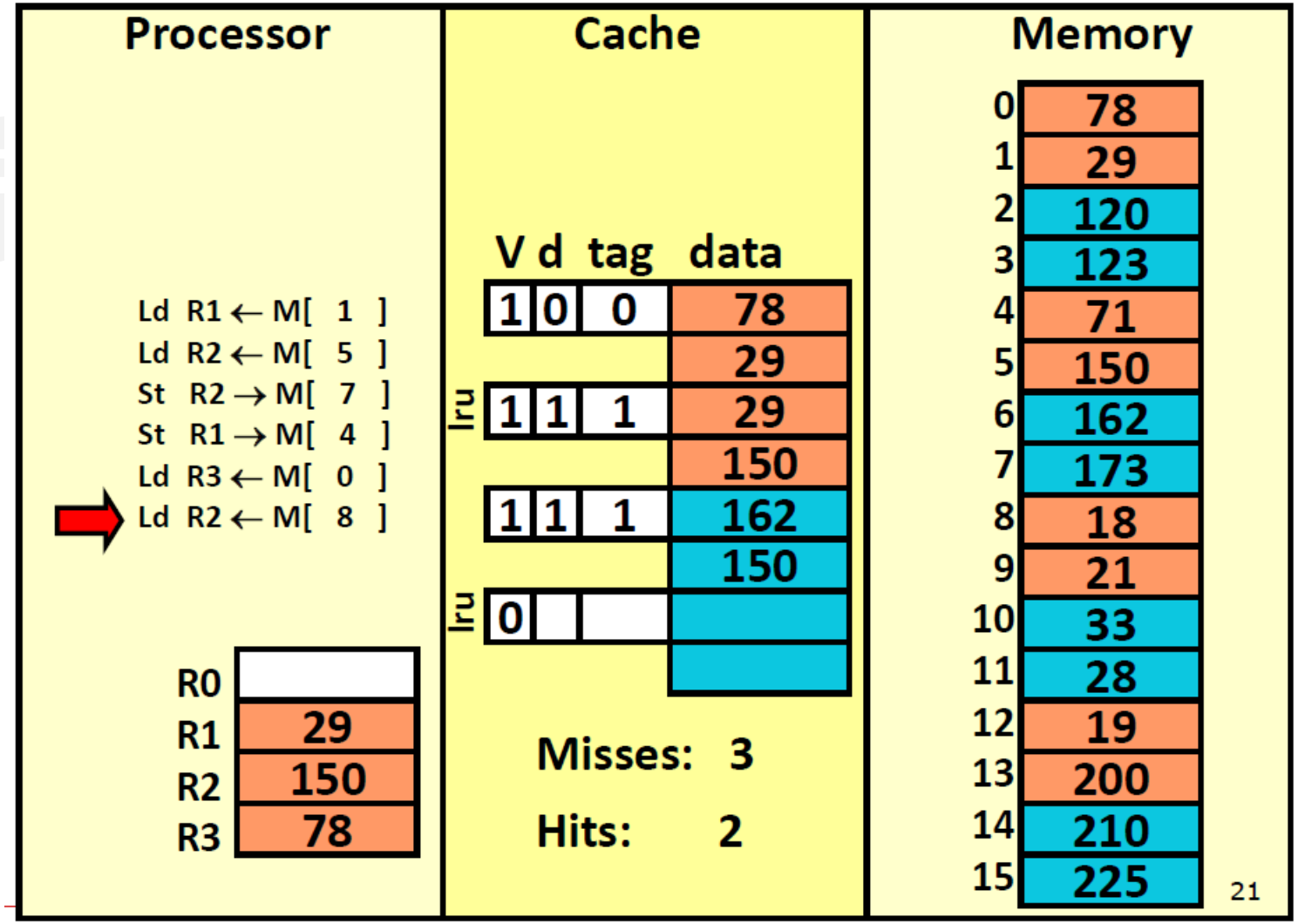
2-Way Set Associative Cache实例

- Write-back & Write-allocate



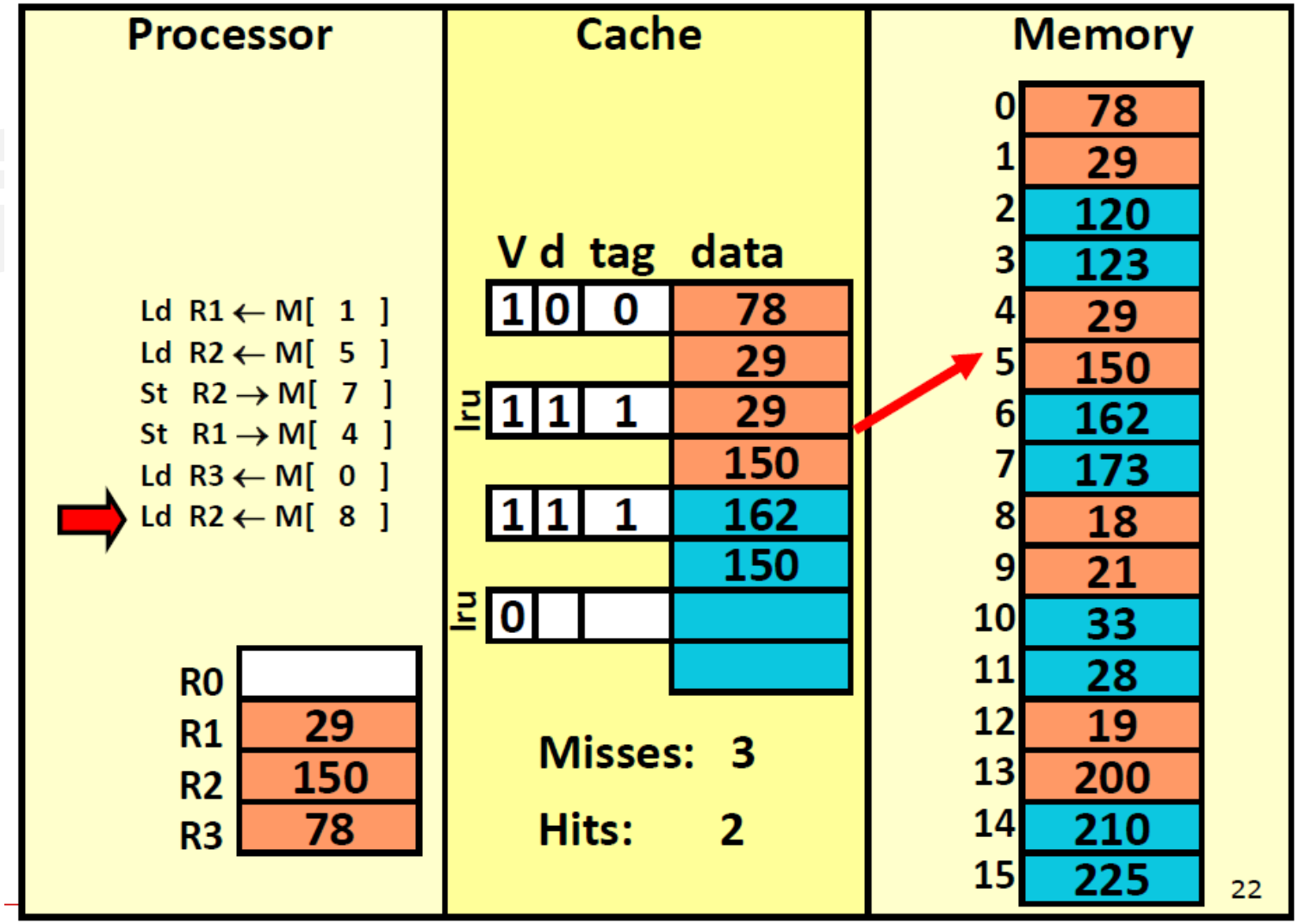
2-Way Set Associative Cache实例

- Write-back & Write-allocate



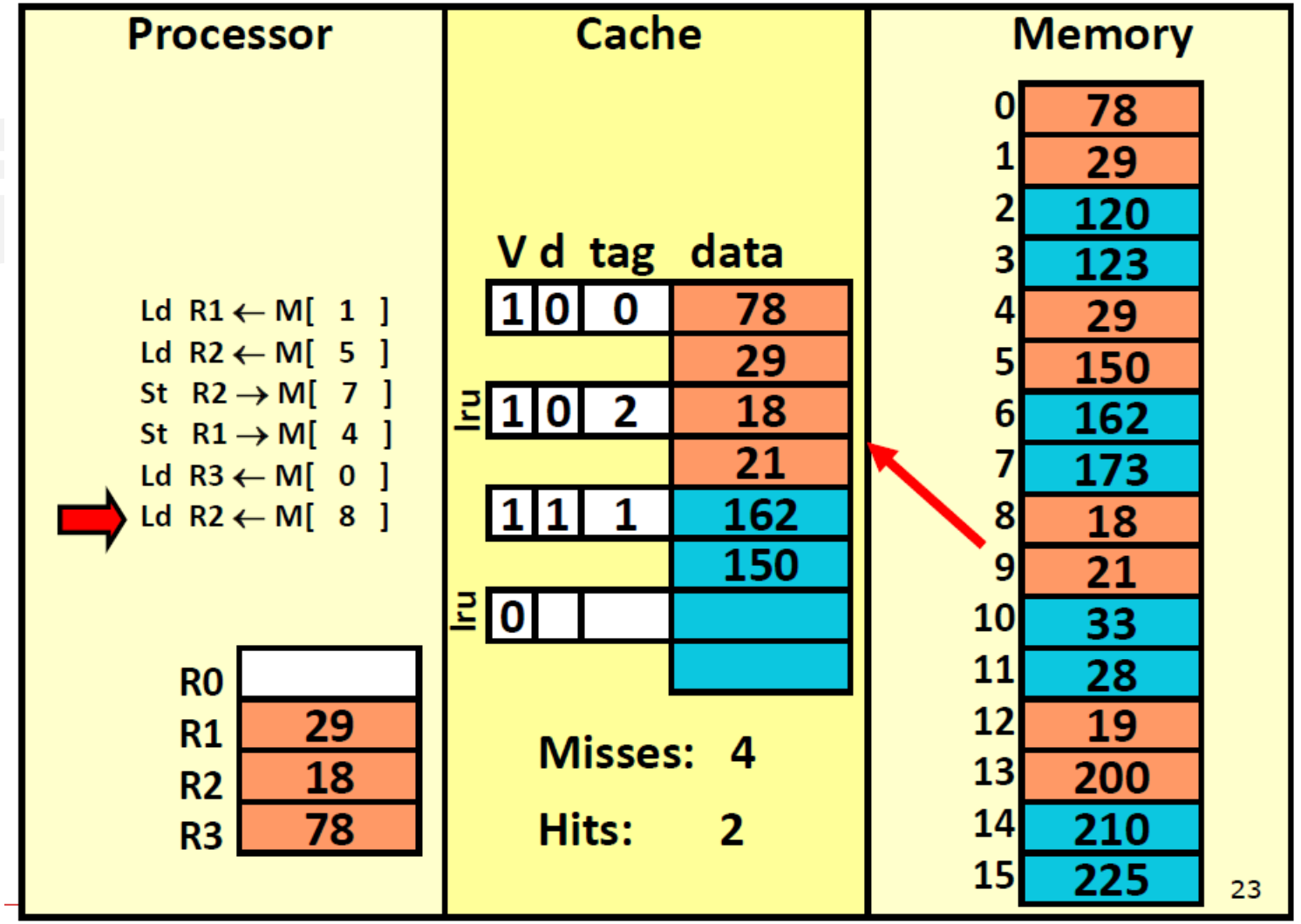
2-Way Set Associative Cache实例

- 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

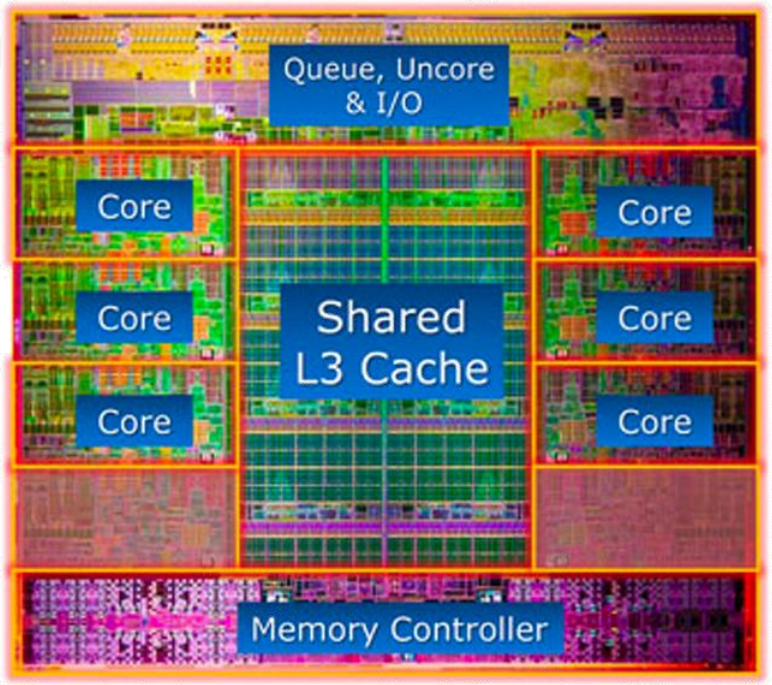
目 录

CONTENTS

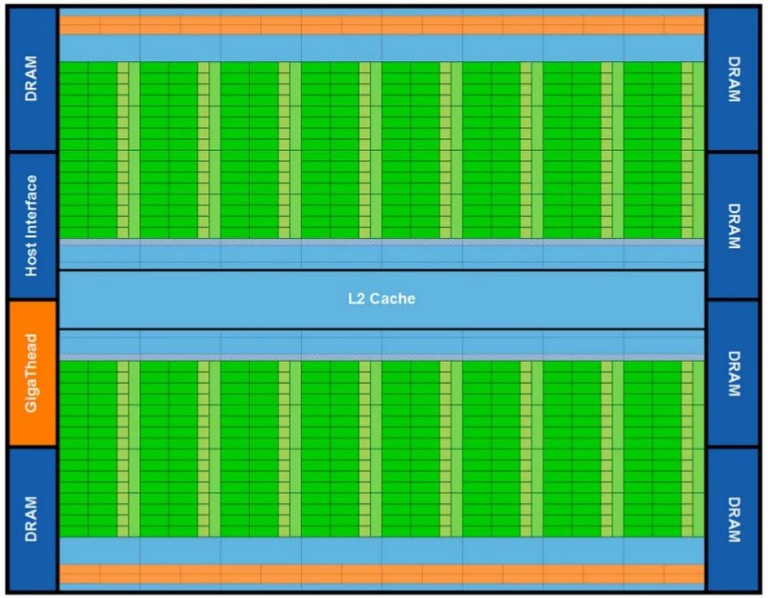
缓存一致性的来源

- 多核共享cache

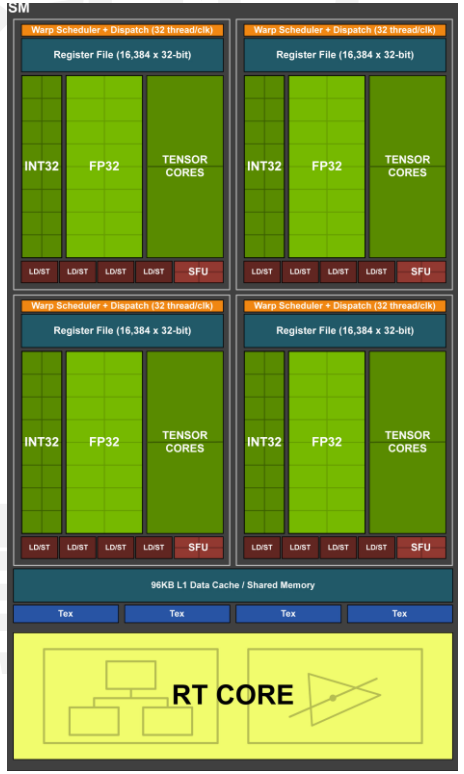
Intel® Core™ i7-3960X Processor Die Detail



- 30% of the die area is cache



Nvidia GPU Architecture

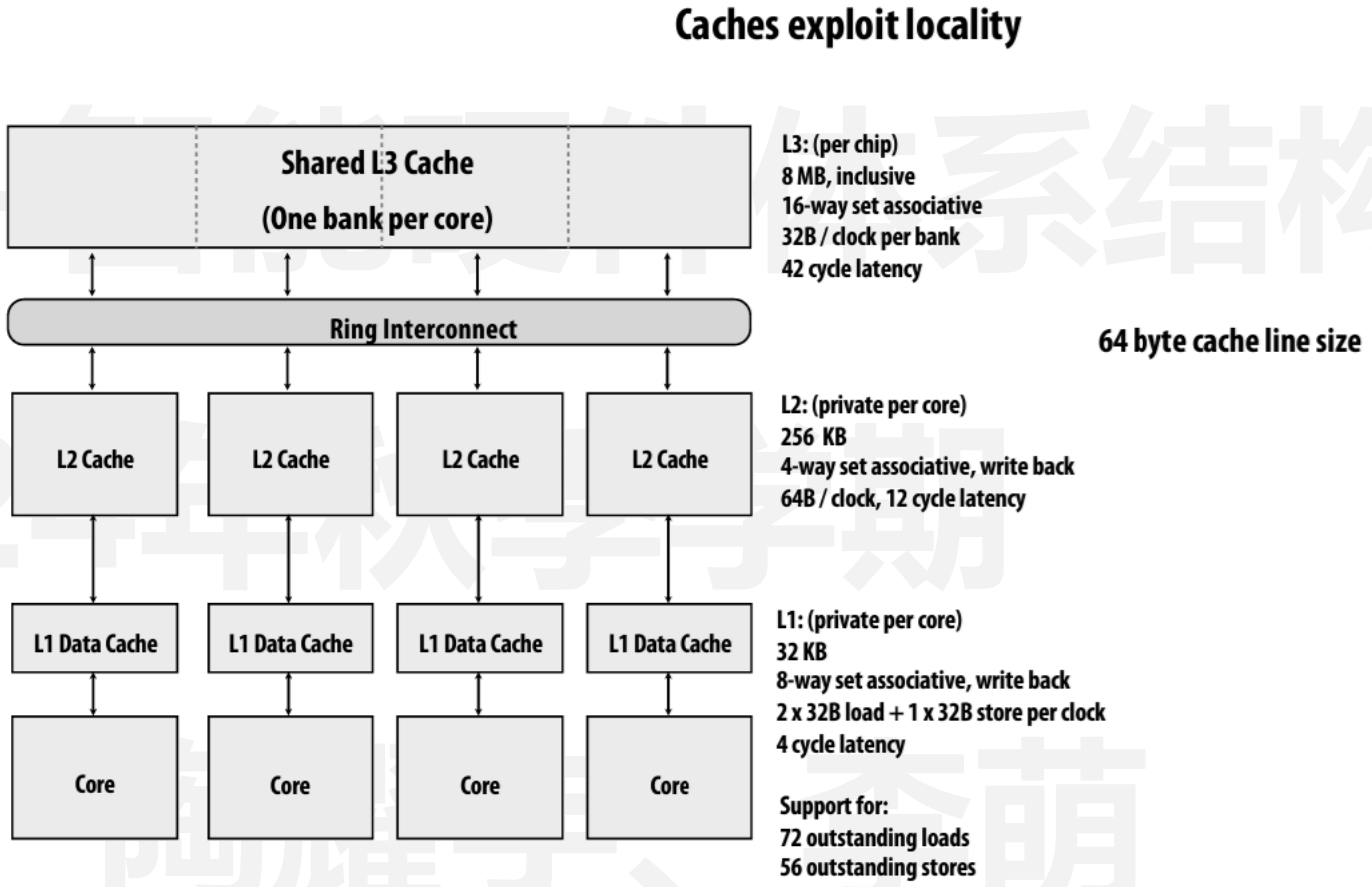


缓存一致性的来源

- 多核共享cache

3 Cs cache miss model

- Cold
- Capacity
- Conflict

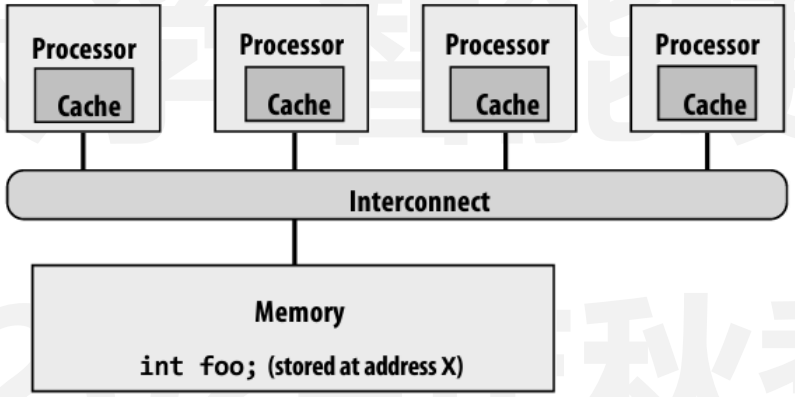


Source: Intel 64 and IA-32 Architectures Optimization Reference Manual (June 2016)

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

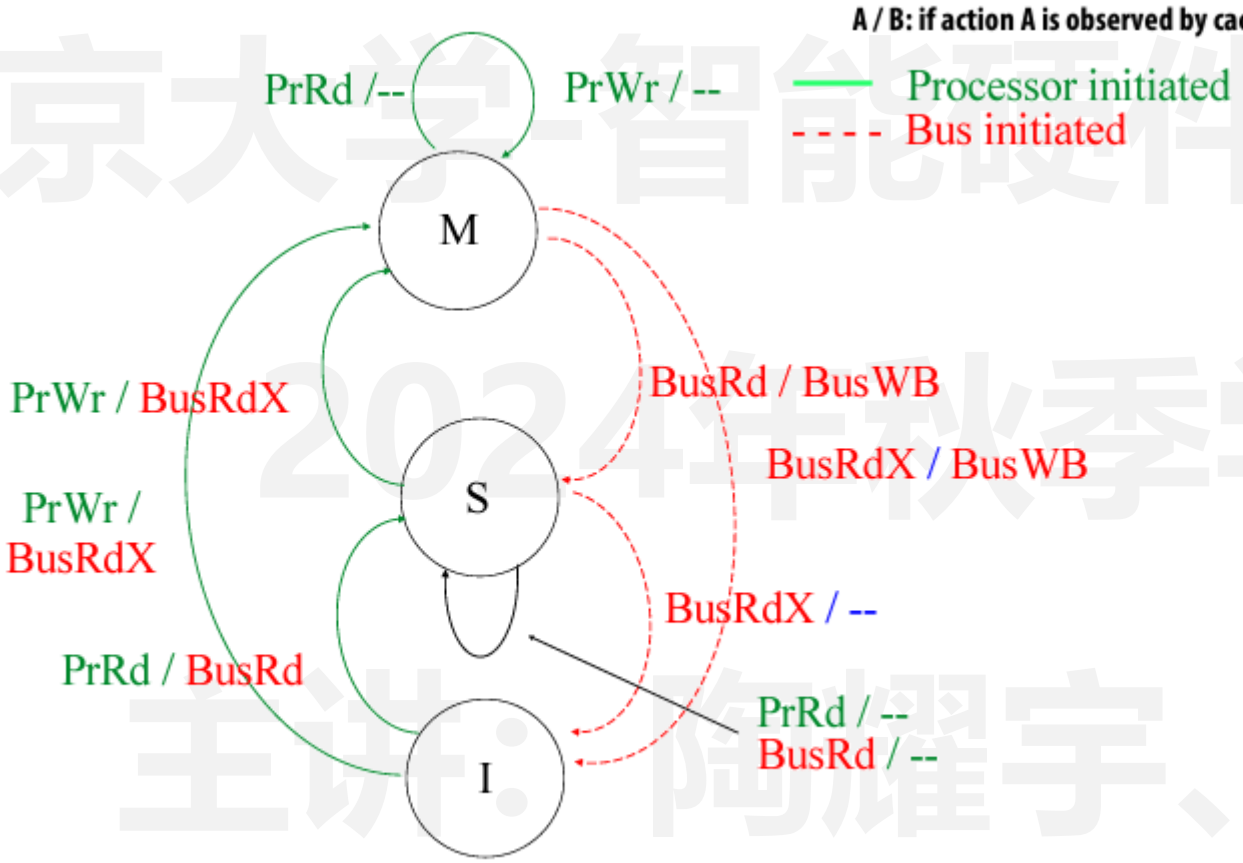
- Invalidation-based write-back protocol
  - **关键思想：**
    - 处于“已修改”状态的行可以在不通知其他缓存的情况下进行修改
  - **处理器只能写入已修改状态的行**
    - 需要一种方法来告诉其他缓存，处理器想要独占访问该行
    - 我们通过向所有其他缓存发送消息来实现这一点
  - **当缓存控制器看到对其所包含的行进行修改访问的请求时**
    - 它必须使缓存中的行无效

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