



北京大學
PEKING UNIVERSITY

人工智能的硬件基石

从物理器件到计算架构

第七讲：指令乱序执行及其微架构

主讲：陶耀宇

2026年春季

• 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**

第一次作业时间：3.30-4.20（今天截止）

第二次作业时间：4.25-5.15

- **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

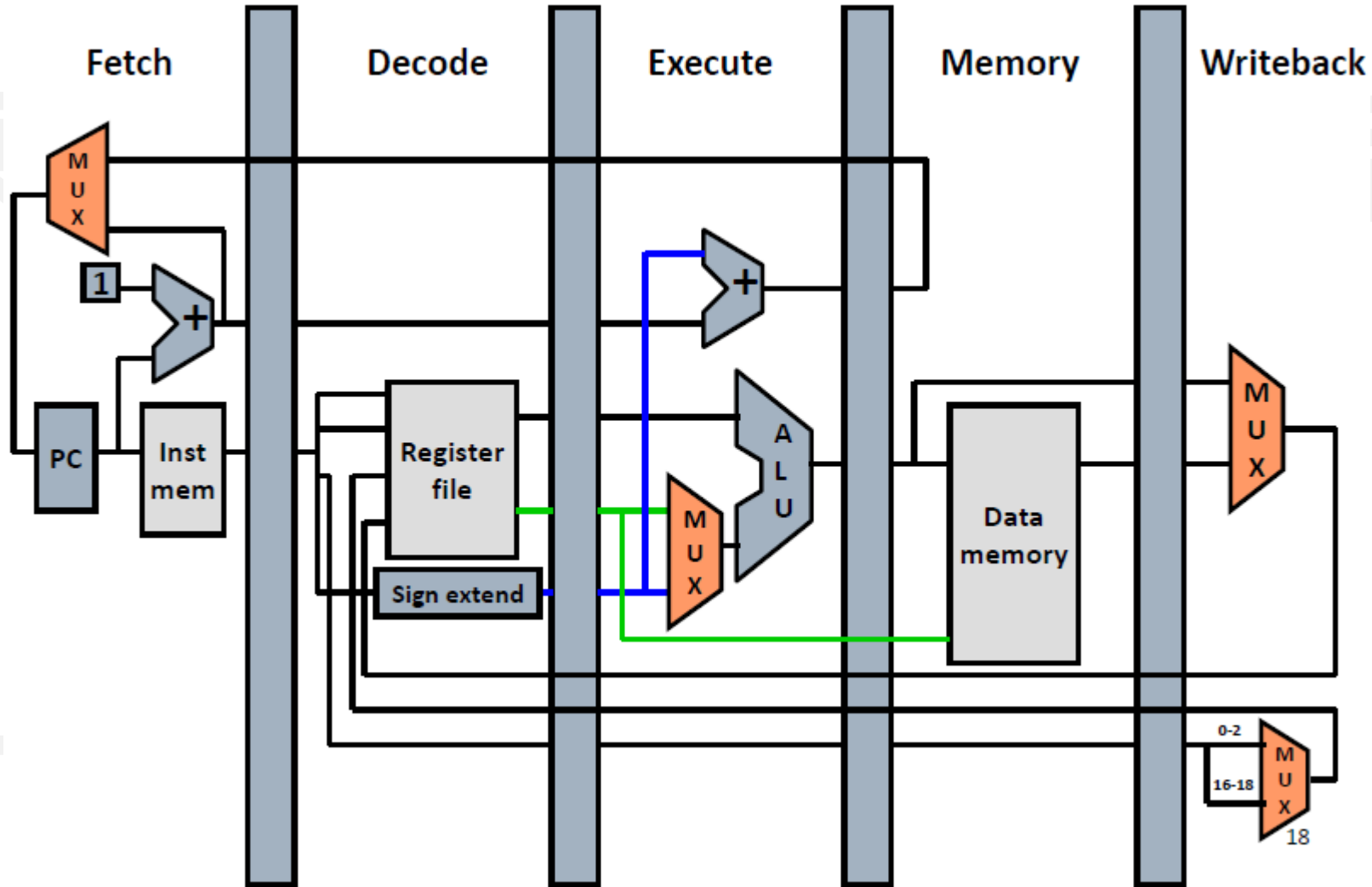
CONTENTS



01. 指令集架构基础
02. 指令集设计基础
03. 流水线架构基础
04. 流水线架构优化

最基本的单条流水线设计示意图

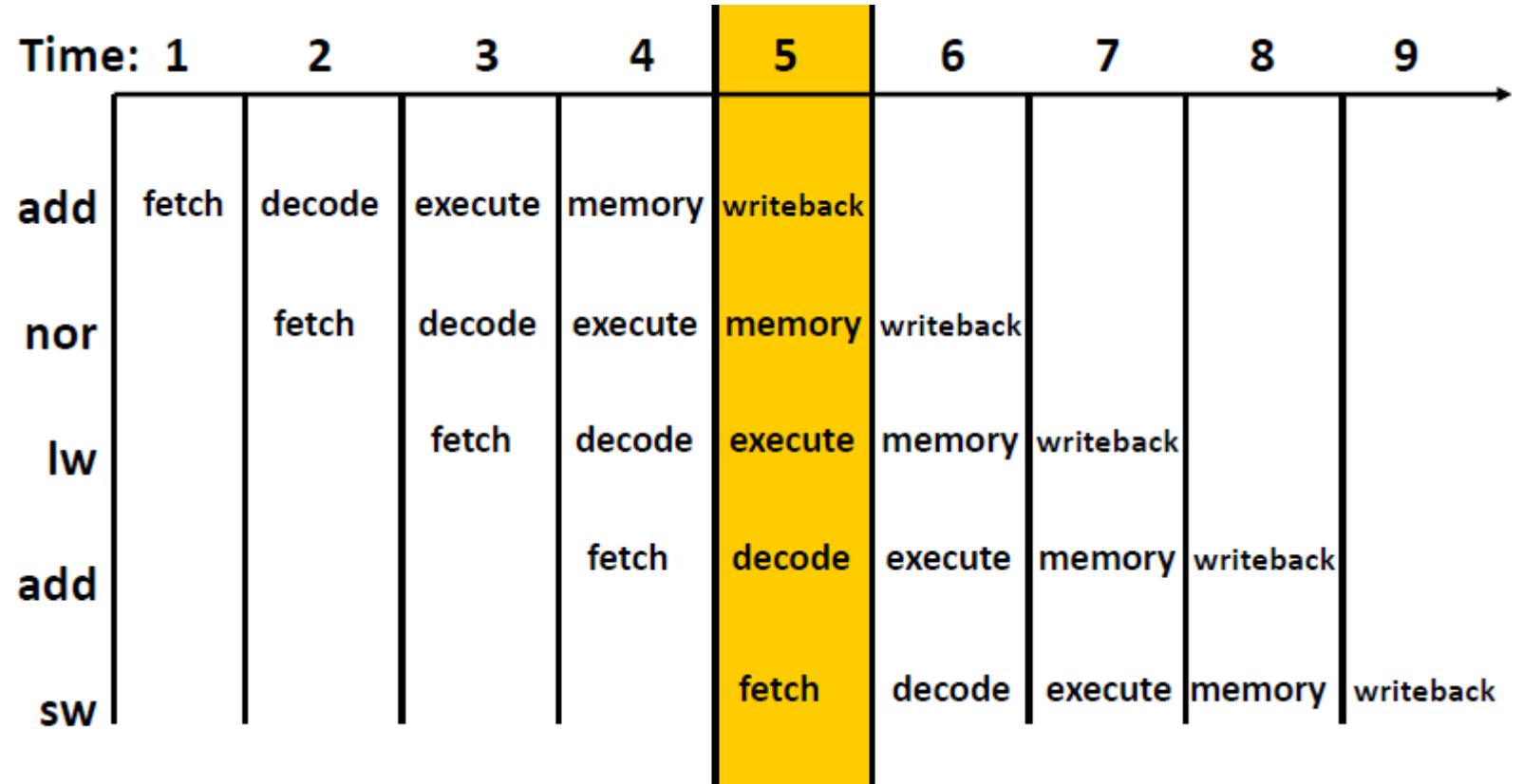
- 5级流水线设计: Fetch、Decode、Execute、Memory、Writeback



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10]=reg 7



目录

CONTENTS



01. 指令集架构基础
02. 指令集设计基础
03. 流水线架构基础
04. 流水线架构优化

简单5级流水线可能存在问题？

- **Data hazards** : since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read the wrong value if it is about to be written.
- **Control hazards** : A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- **Exceptions**: Sometimes we need to pause execution, switch to another task (maybe the OS), and then resume execution... how to we make sure we resume at the right spot

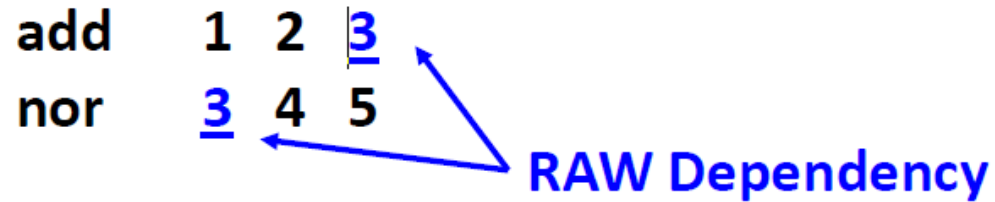
主讲：陶耀宇、李萌

问题1: Data Hazards

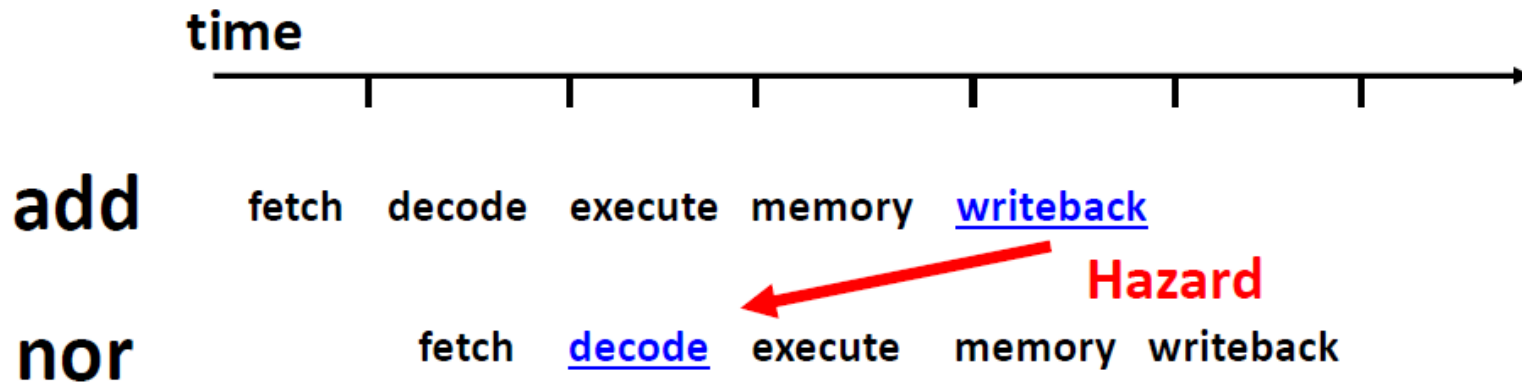
- RAW问题: Read After Write 数据冲突

北京

Recall: registers are read /sourced in the "decode" stage



构

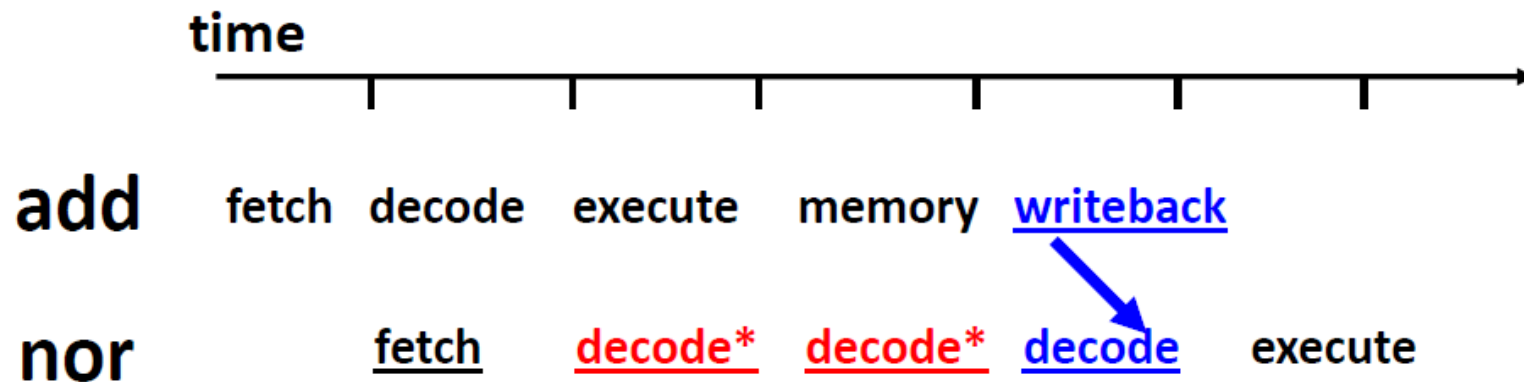


If not careful, nor will read a stale value of **register 3**

问题1: Data Hazards

- RAW问题: Read After Write 数据冲突

add	1	2	<u>3</u>
nor	<u>3</u>	4	5

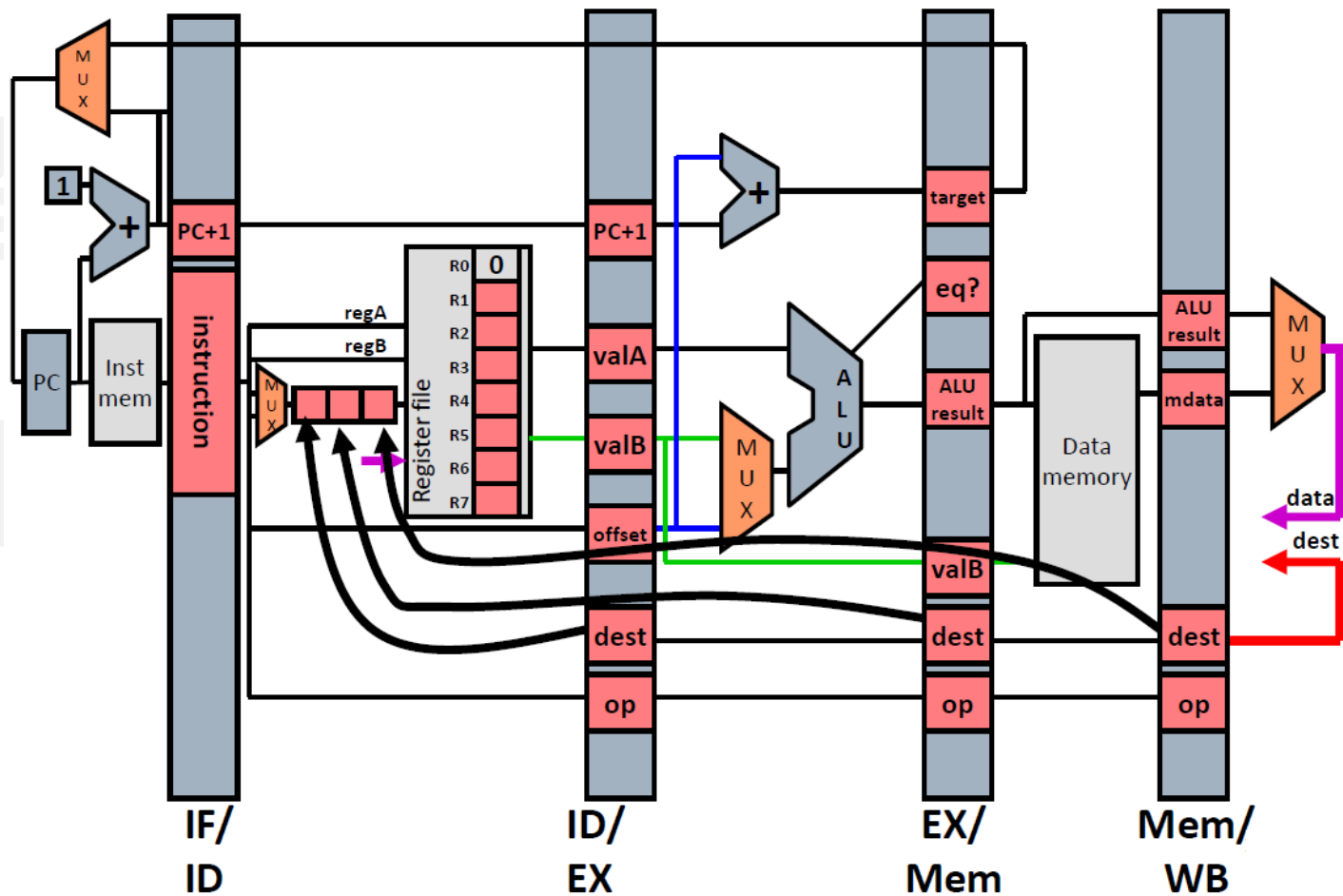
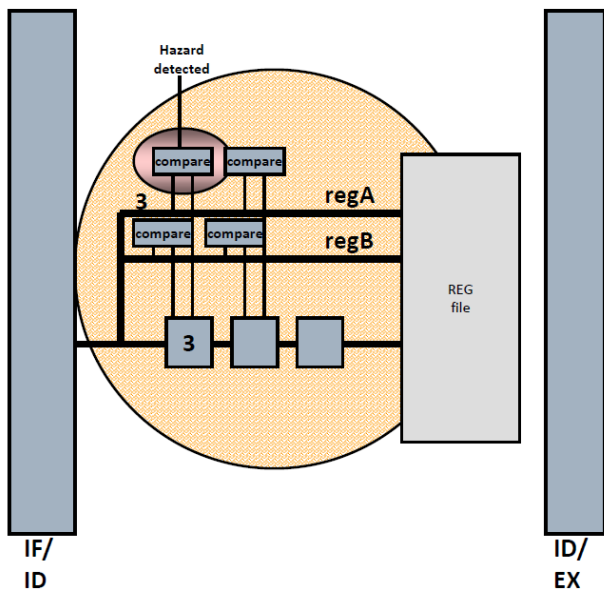


简单解决办法: 流水线停顿 (Pipeline Stall)

问题1: Data Hazards

- RAW问题: Read After Write数据冲突

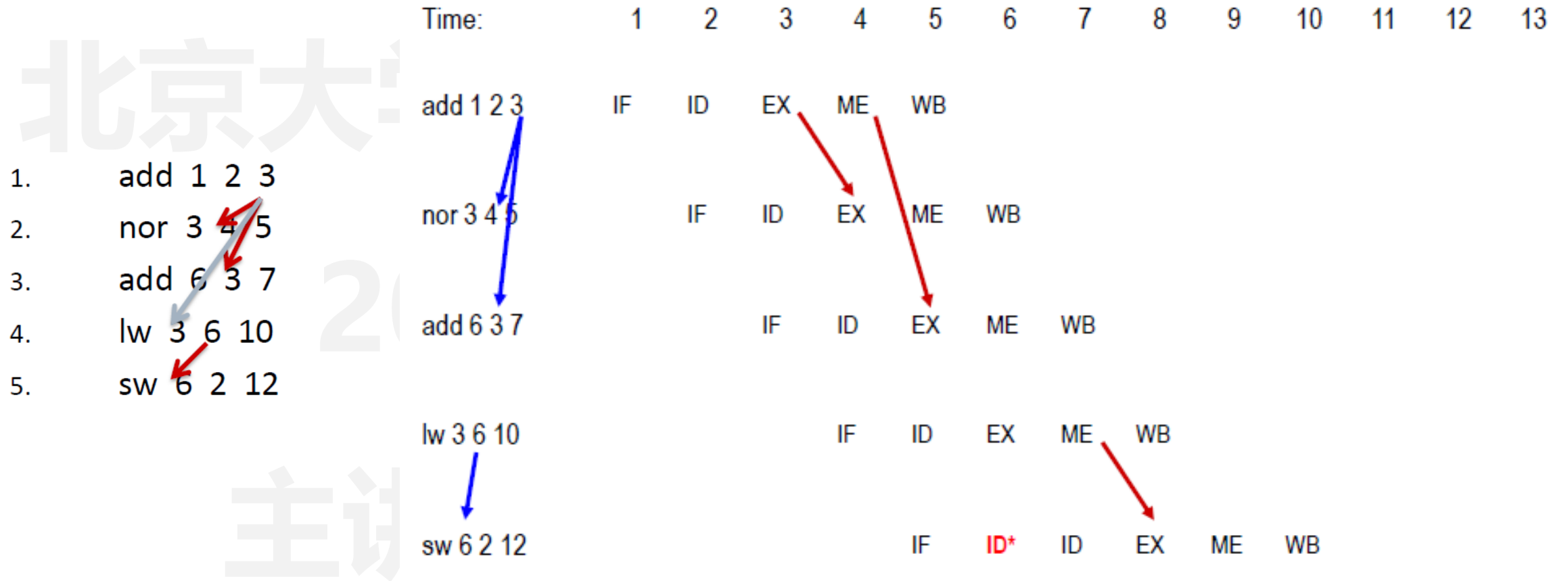
```
1. add 1 2 3
2. nor 3 4 5
3. add 6 3 7
4. lw 3 6 10
5. sw 6 2 12
```



进阶解决办法: Detect and Forward

问题1: Data Hazards

- RAW问题: Read After Write数据冲突



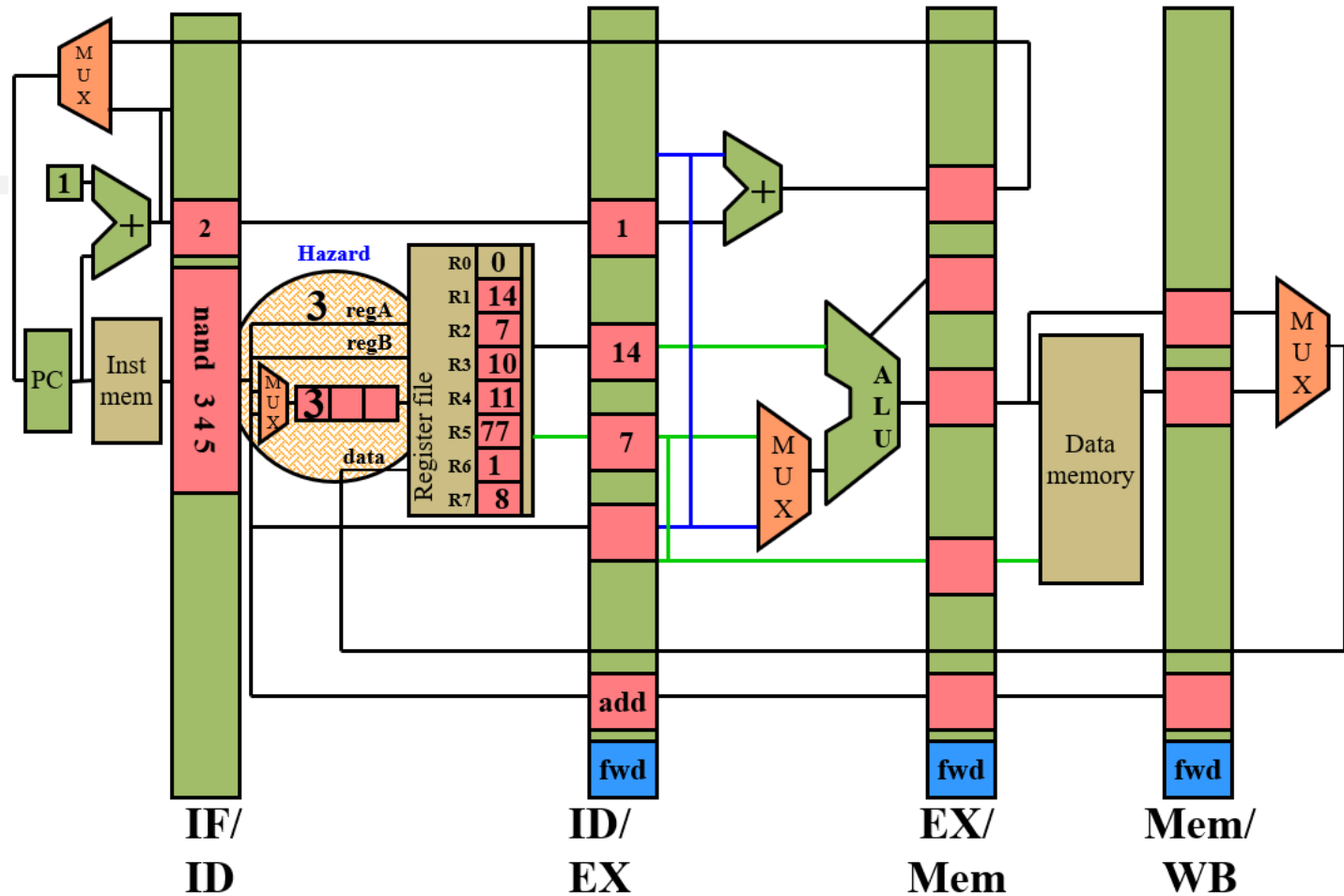
进阶解决办法: Detect and Forward

问题1: Data Hazards

• Detect and Forward案例

add 1 2 3 // r3 = r1 + r2
nand 3 4 5 // r5 = r3 NAND r4
add 6 3 7 // r7 = r3 + r6
lw 3 6 10 // r6 = MEM[r3+10]
sw 6 2 12 // MEM[r6+12]=r2

Cycle 3前半段



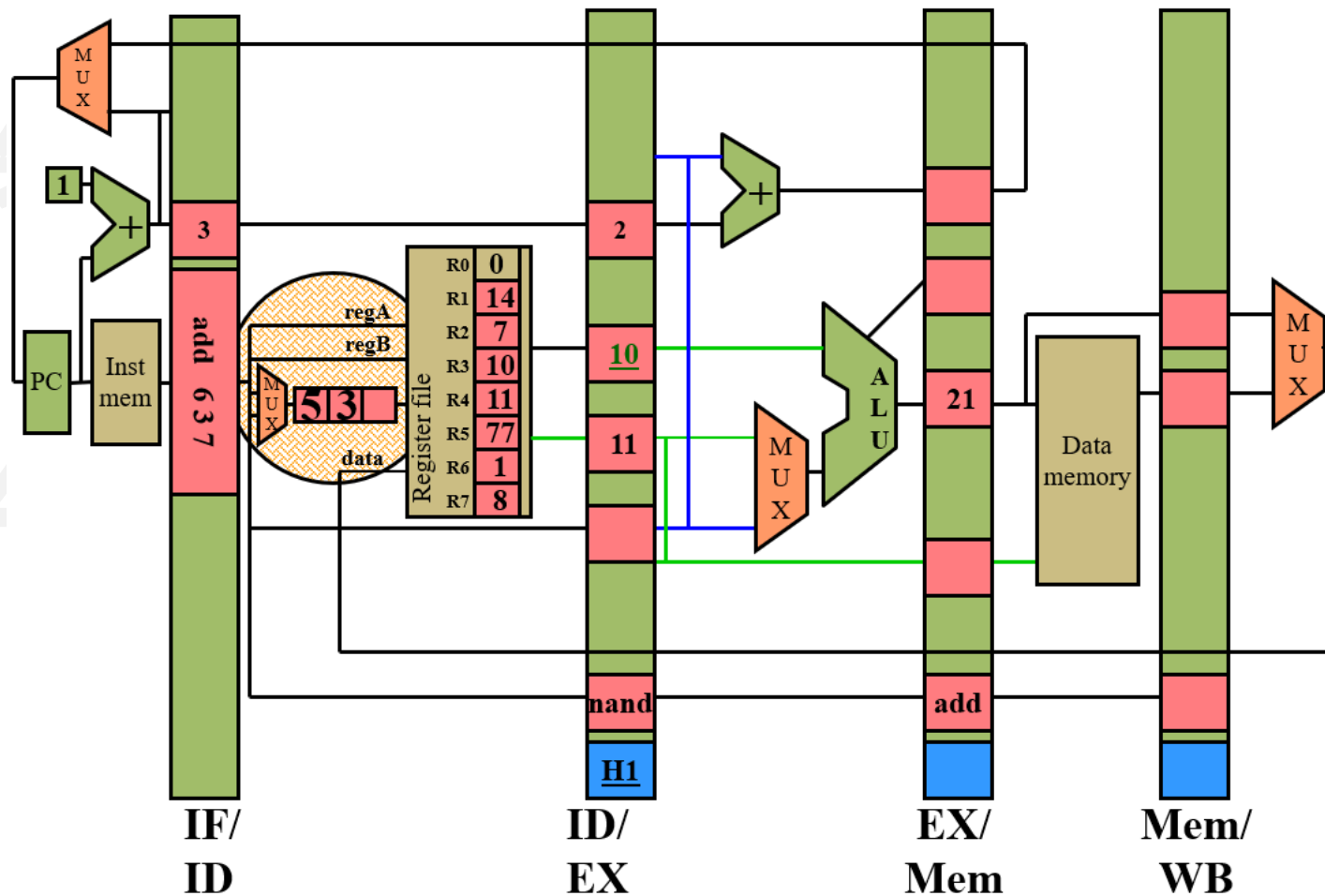
主讲:

问题1: Data Hazards

• Detect and Forward案例

add 1 2 3 // r3 = r1 + r2
nand 3 4 5 // r5 = r3 NAND r4
add 6 3 7 // r7 = r3 + r6
lw 3 6 10 // r6 = MEM[r3+10]
sw 6 2 12 // MEM[r6+12]=r2

Cycle 3后半段



问题1: Data Hazards

• Detect and Forward案例

Cycle 4前半段 (forwarding)

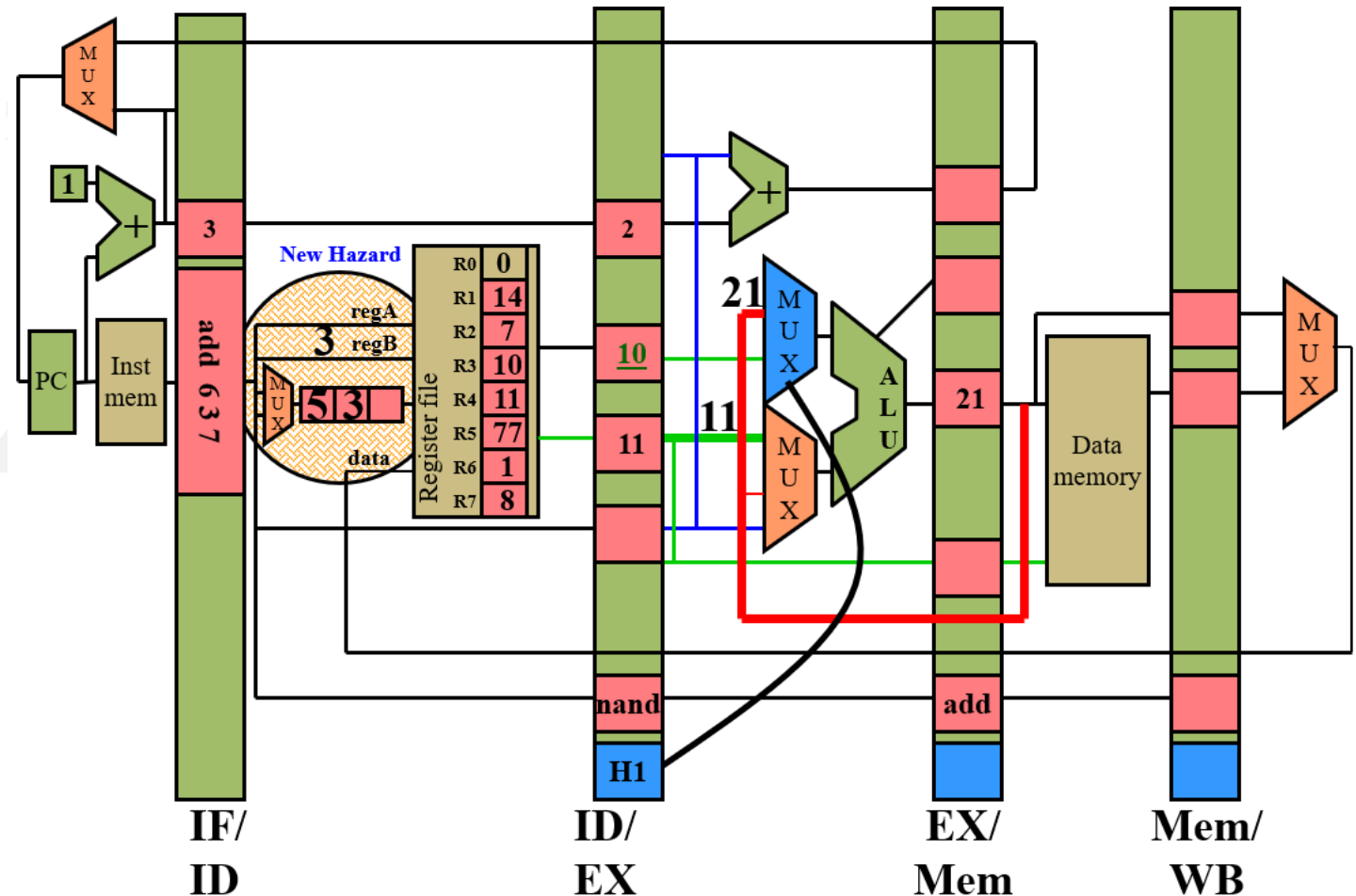
add 1 2 3 // r3 = r1 + r2

nand 3 4 5 // r5 = r3 NAND r4

add 6 3 7 // r7 = r3 + r6

lw 3 6 10 // r6 = MEM[r3+10]

sw 6 2 12 // MEM[r6+12]=r2



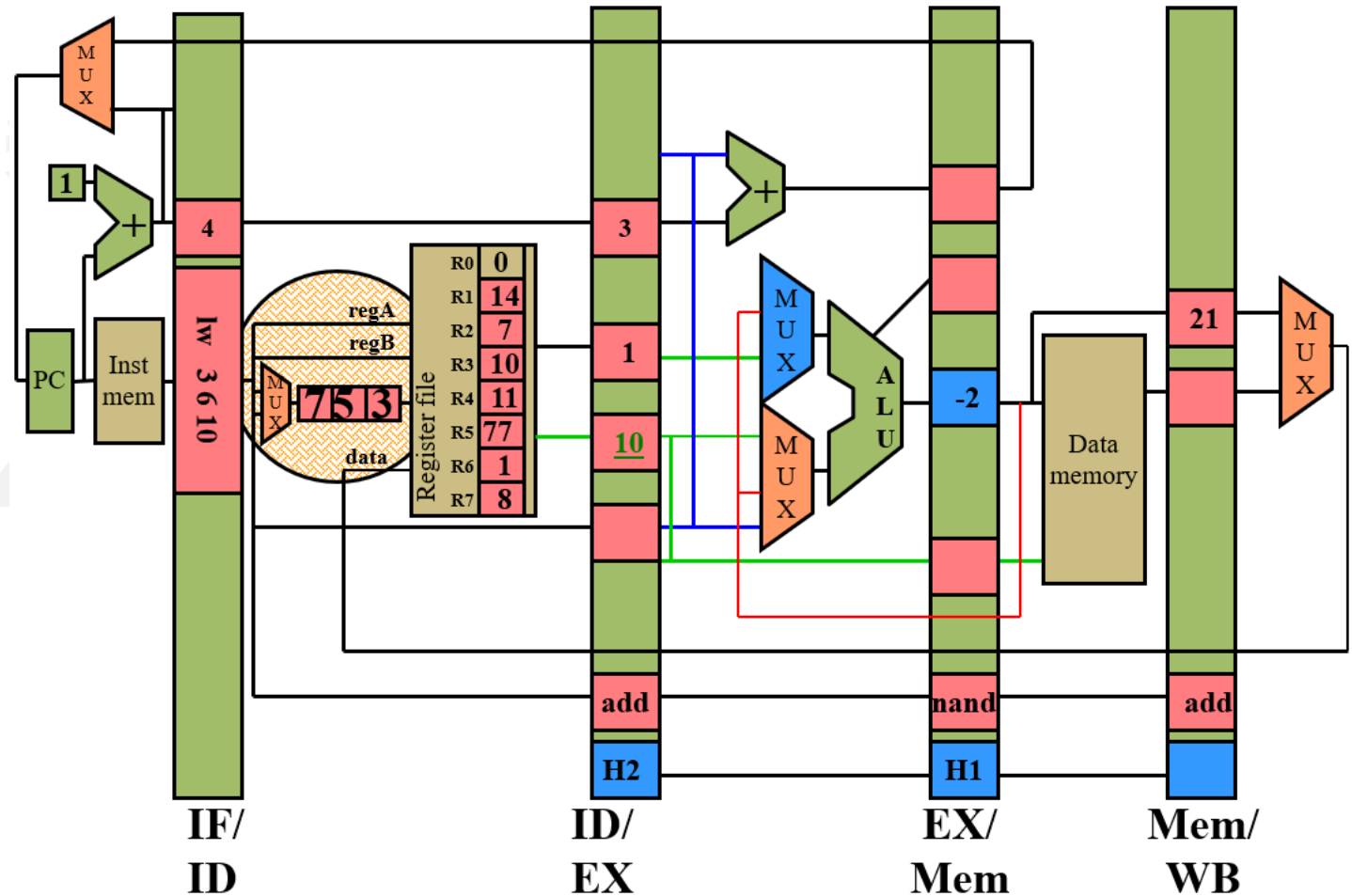
主讲:

问题1: Data Hazards

• Detect and Forward案例

Cycle 4后半段

add 1 2 3 // r3 = r1 + r2 = 21
nand 3 4 5 // r5 = r3 NAND r4
add 6 3 7 // r7 = r3 + r6 = 22
lw 3 6 10 // r6 = MEM[r3+10]
sw 6 2 12 // MEM[r6+12]=r2



问题1: Data Hazards

• Detect and Forward案例

Cycle 5前半段

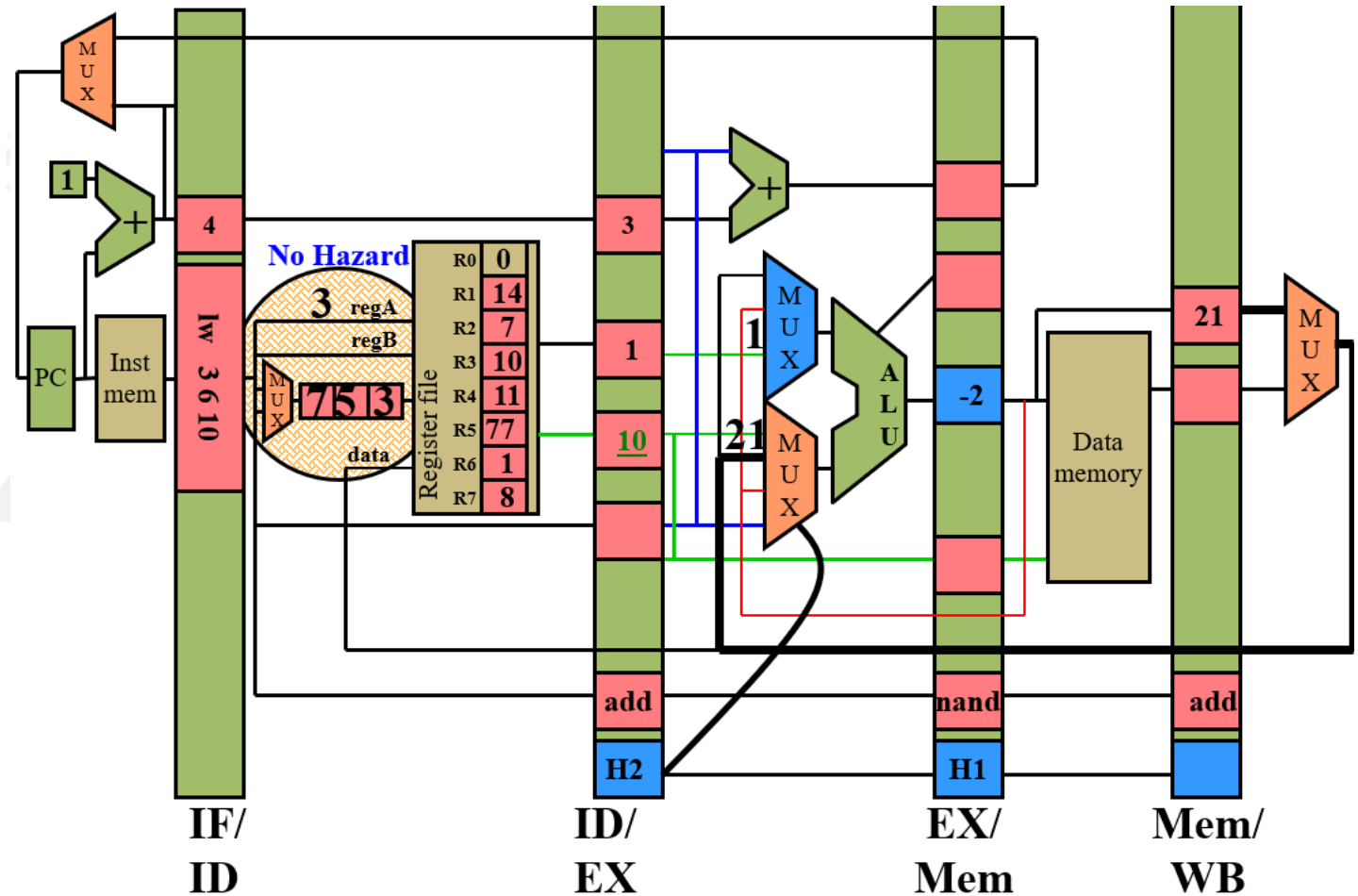
add 1 2 3 // r3 = r1 + r2

nand 3 4 5 // r5 = r3 NAND r4

add 6 3 7 // r7 = r3 + r6

lw 3 6 10 // r6 = MEM[r3+10]

sw 6 2 12 // MEM[r6+12]=r2



主讲:

问题1: Data Hazards

• Detect and Forward案例

Cycle 5后半段

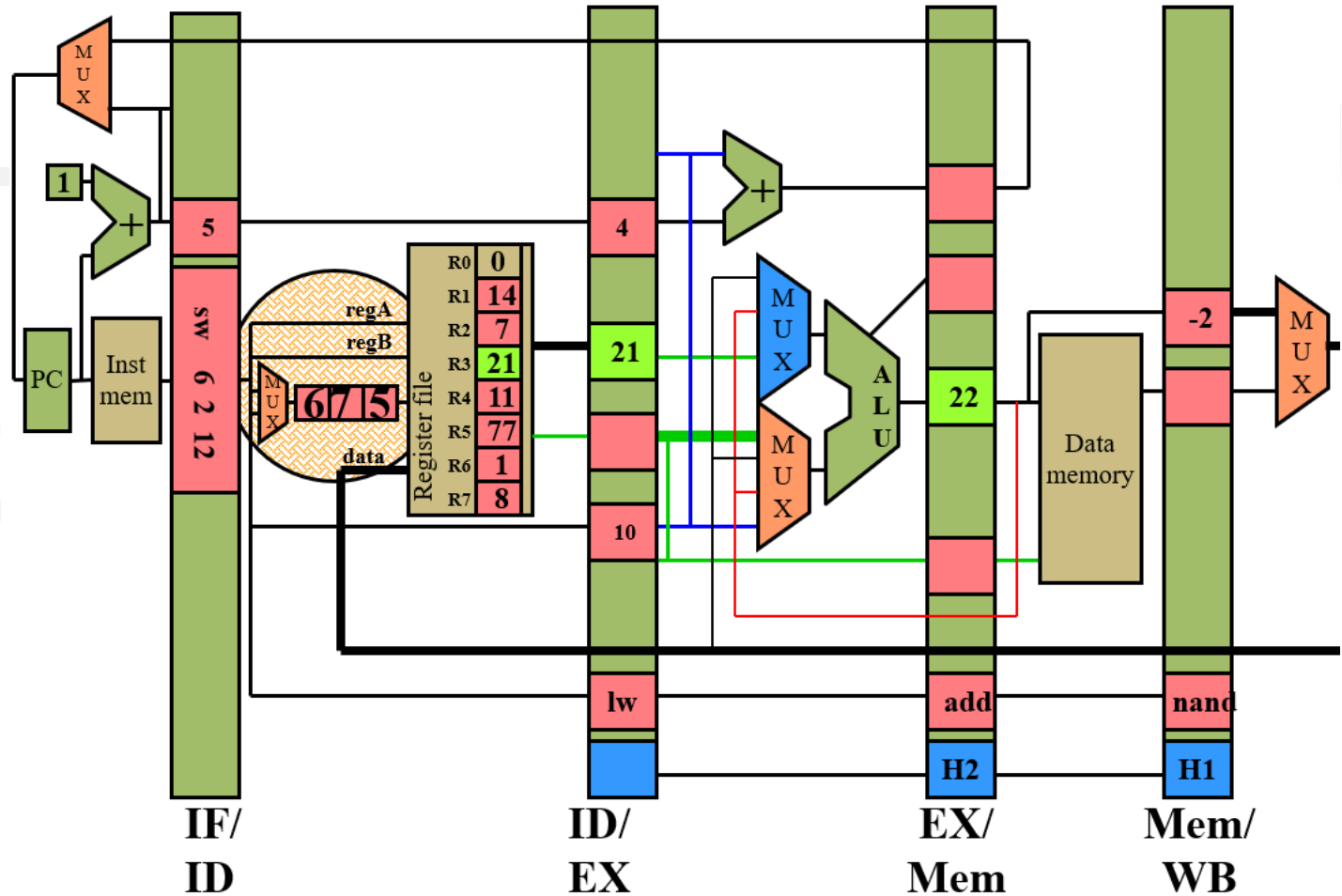
add 1 2 3 // r3 = r1 + r2

nand 3 4 5 // r5 = r3 NAND r4

add 6 3 7 // r7 = r3 + r6

lw 3 6 10 // r6 = MEM[r3+10]

sw 6 2 12 // MEM[r6+12]=r2



主讲:

问题1: Data Hazards

- WAW和WAR问题: Write After Write和Write After Read

- False or Name dependencies

- WAW – Write after Write

$$R1=R2+R3$$

$$R1=R4+R5$$

- WAR – Write after Read

$$R2=R1+R3$$

$$R1=R4+R5$$

- 在顺序的单条5级流水线上不会出现问题
- **指令乱序执行则会出现问题, 可利用Register重命名解决 (后续乱序执行深入讲解)**

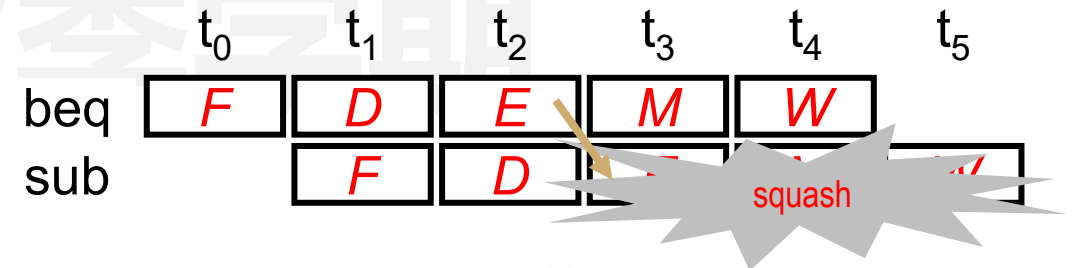
问题2: Control Hazards

- Branch类指令

- Fetch: read instruction from memory
- Decode: read source operands from reg
- Execute: calculate target address and test for equality
- Memory: Send target to PC if test is equal
- Writeback: Nothing left to do

```

beq  1 1 10
sub  3 4 5
  
```



主讲: 陶耀宇、李萌

问题2: Control Hazards

• 如何解决Control Hazards

Avoidance (static)

- No branches?
- Convert branches to predication
 - Control dependence becomes data dependence

Detect and Stall (dynamic)

- Stop fetch until branch resolves

Speculate and squash (dynamic)

- Keep going past branch, throw away instructions if wrong

问题2: Control Hazards

• Detection and Stall: 检测-停顿机制

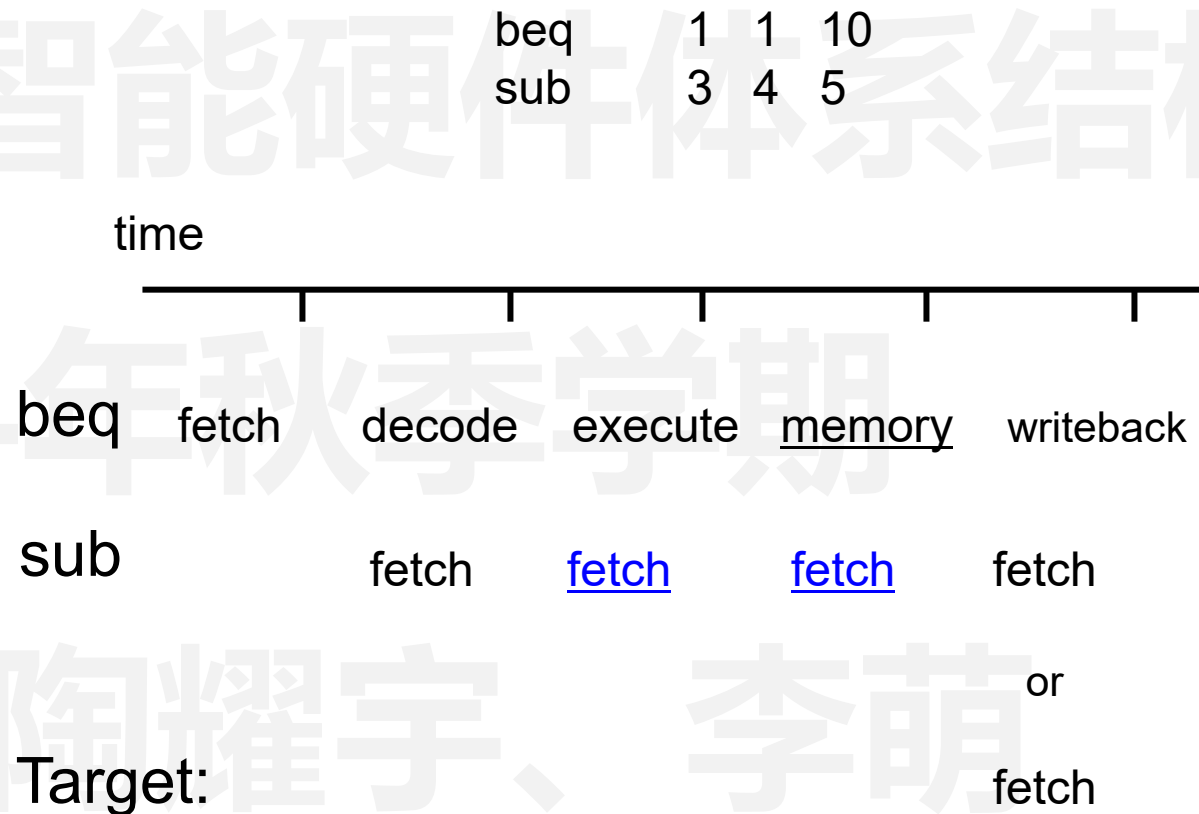
Detection

- In decode, check if opcode is branch or jump

Stall

- Hold next instruction in Fetch
- Pass noop to Decode

- CPI increases on every branch
- Are these stalls necessary? Not always!
 - Assume branch is NOT taken
 - Keep fetching, treat branch as noop
 - If wrong, make sure bad instructions don't complete



问题2: Control Hazards

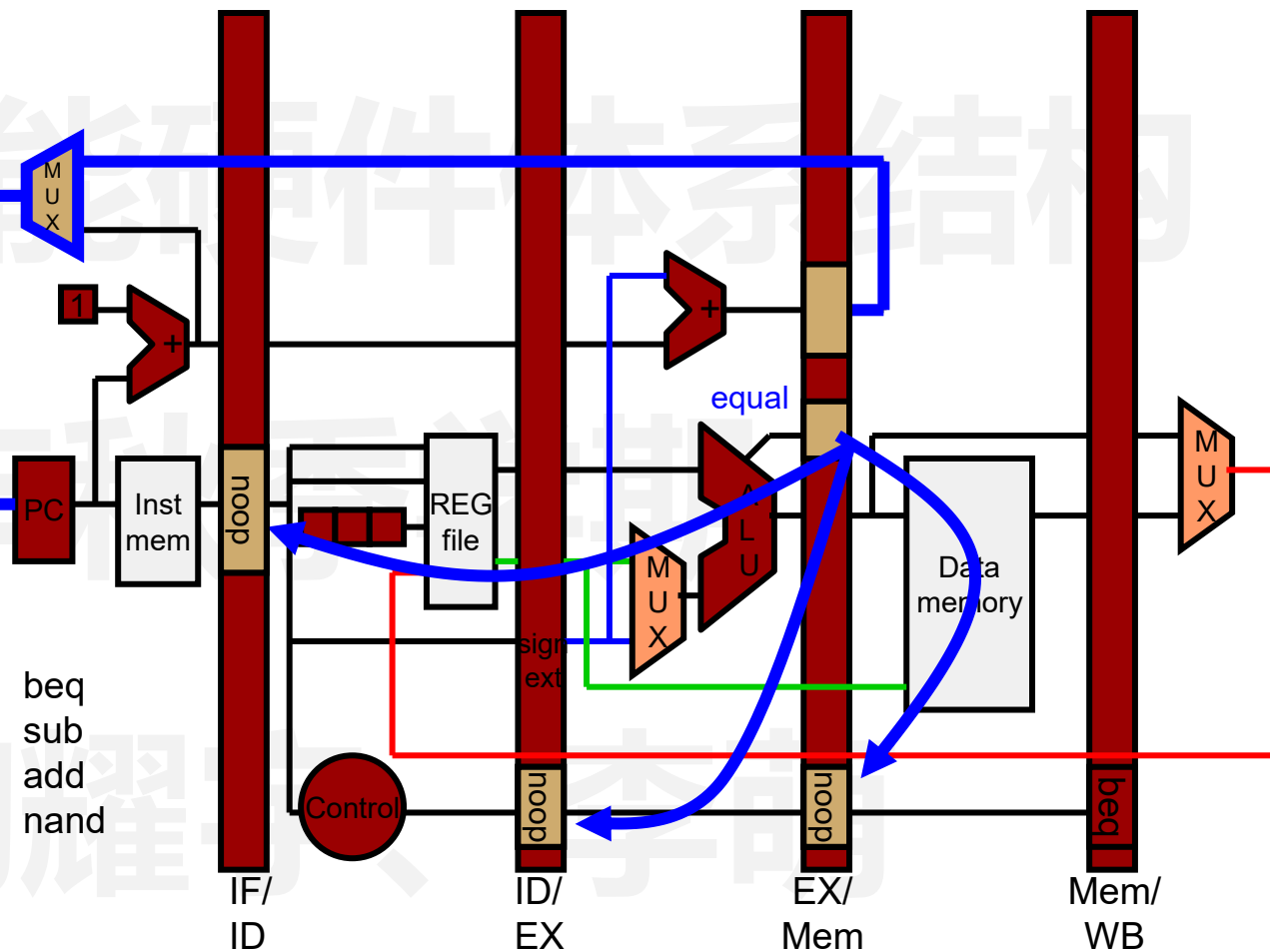
• Speculate and Squash: 投机-制止机制

Speculate "Not-Taken"

- Assume branch is not taken

Squash

- Overwrite opcodes in Fetch, Decode, Execute with noop
- Pass target to Fetch



问题2: Control Hazards

- Speculate and Squash: 投机-制止机制的问题

Always assumes branch is not taken

Can we do better? Yes.

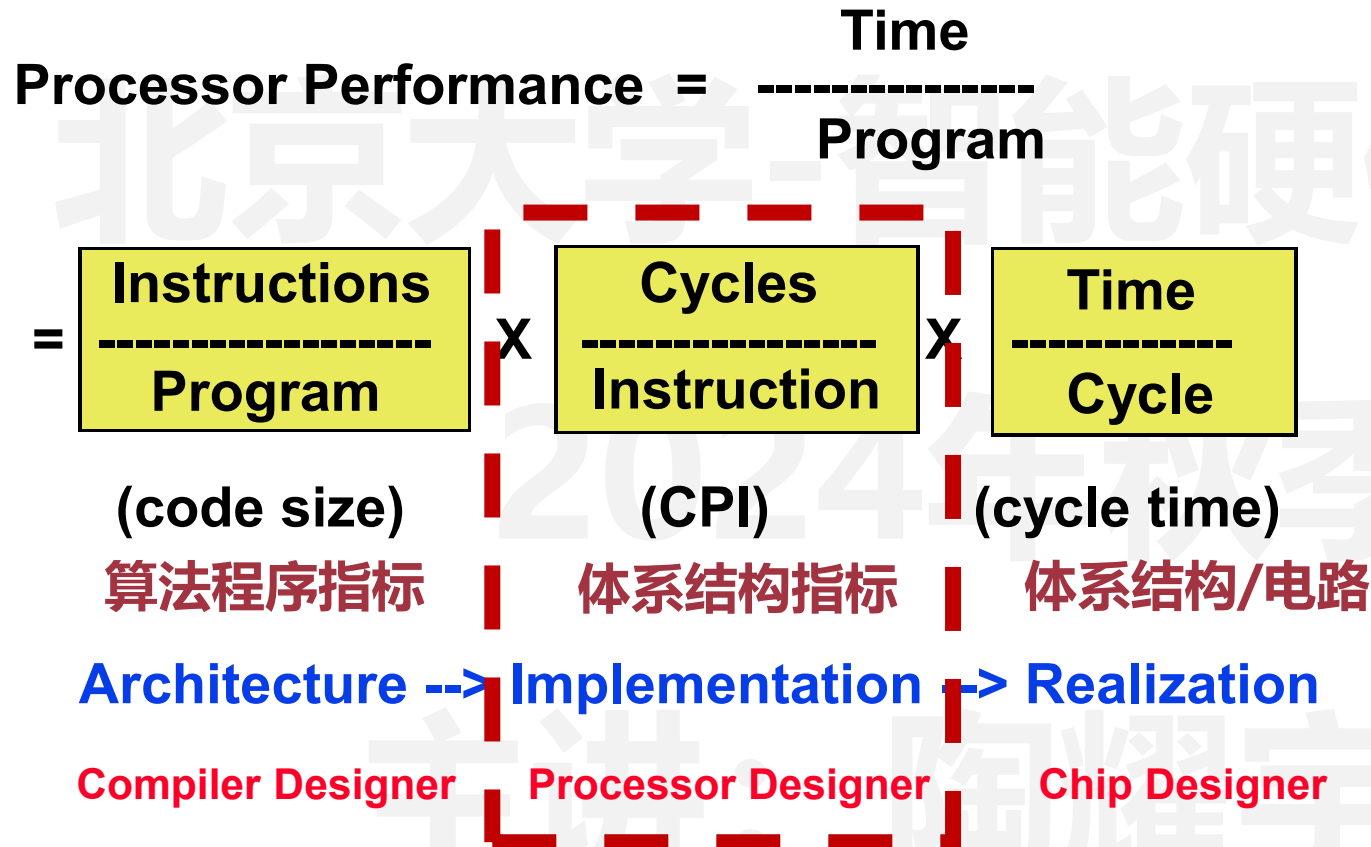
- **Predict branch direction and target!**
- Why possible? Program behavior repeats.

More on branch prediction to come...

主讲：陶耀宇、李萌

如何提高指令运行的并行度?

- Instruction-level parallelism



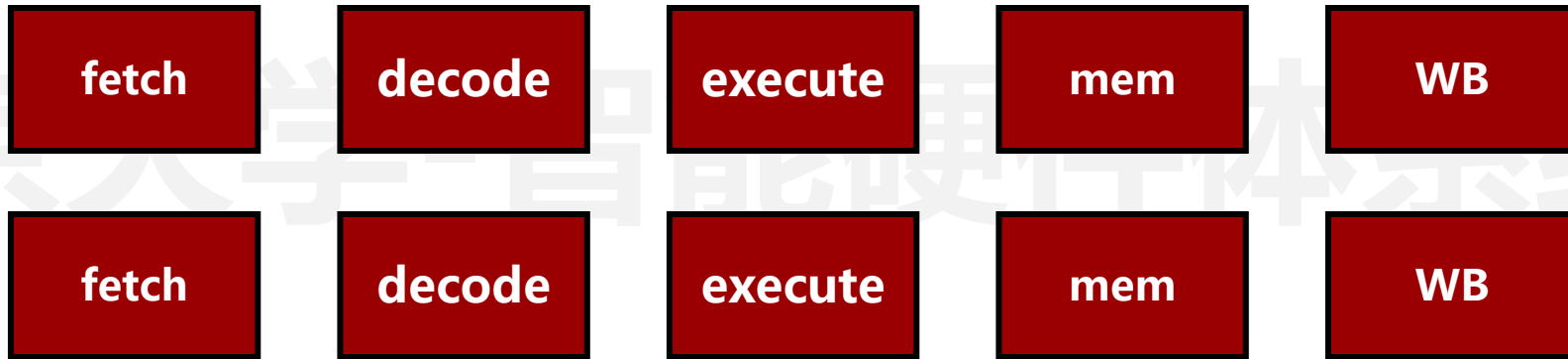
两大限制:

- 1、标量流水吞吐带来的上界
- 2、严格顺序流水执行带来的性能损失

Unnecessary stalls

并行度的来源

- 简单并行流水线



更加复杂的冒险检测

- 2X 流水线寄存器用于前馈转发
- 2X 指令检查
- 2X more destinations (MUXes)
- 需要考虑同一级的多个指令是否独立的问题

Superscalar: 超标量的概念

• Instruction-level parallelism

指令并行度

同时执行的指令个数

Peak IPC

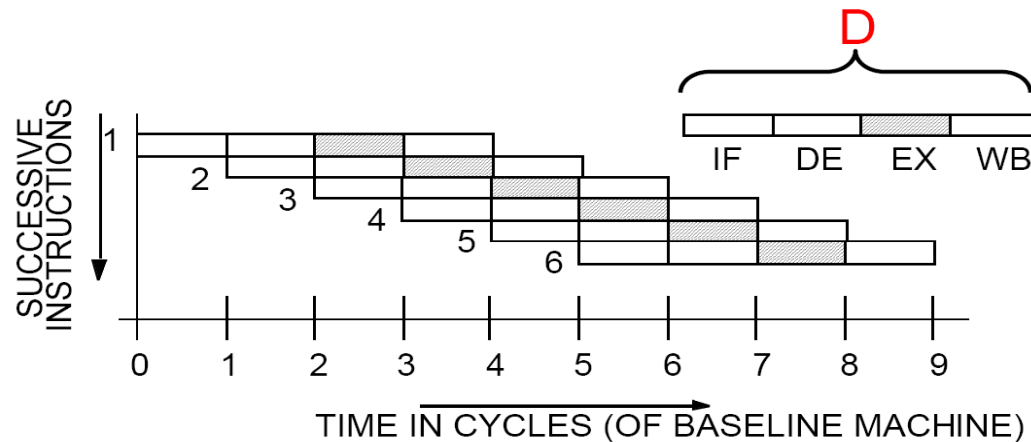
可以在一个时钟周期内运行的指令的最大可持续数量

Scalar Pipeline (baseline)

Instruction Parallelism = D

Operation Latency = 1

Peak IPC = 1

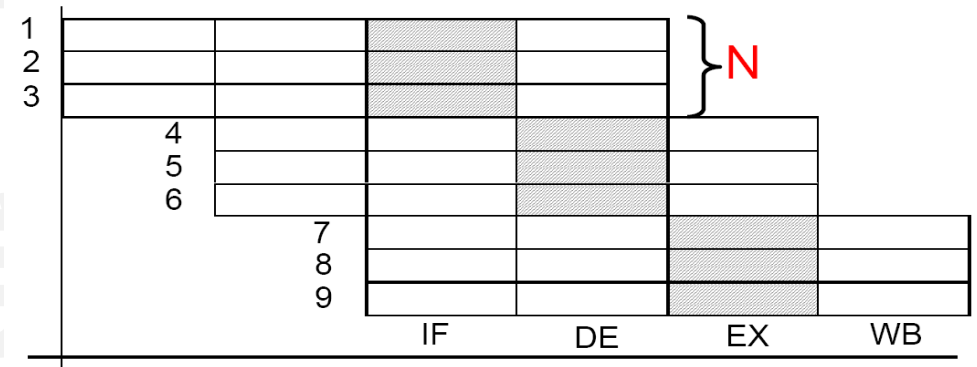


Superscalar (Pipelined) Execution

IP = $D \times N$

OL = 1 baseline cycles

Peak IPC = N per baseline cycle



Out-Of-Order: 乱序执行的概念

• Missed Speedup in In-Order Pipelines

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
<code>addf f0, f1, f2</code>	F	D	E+	E+	E+	W										
<code>mulf f2, f3, f2</code>		F	D	d*	d*	E*	E*	E*	E*	E*	W					
<code>subf f0, f1, f4</code>			F	p*	p*	D	E+	E+	E+	W						

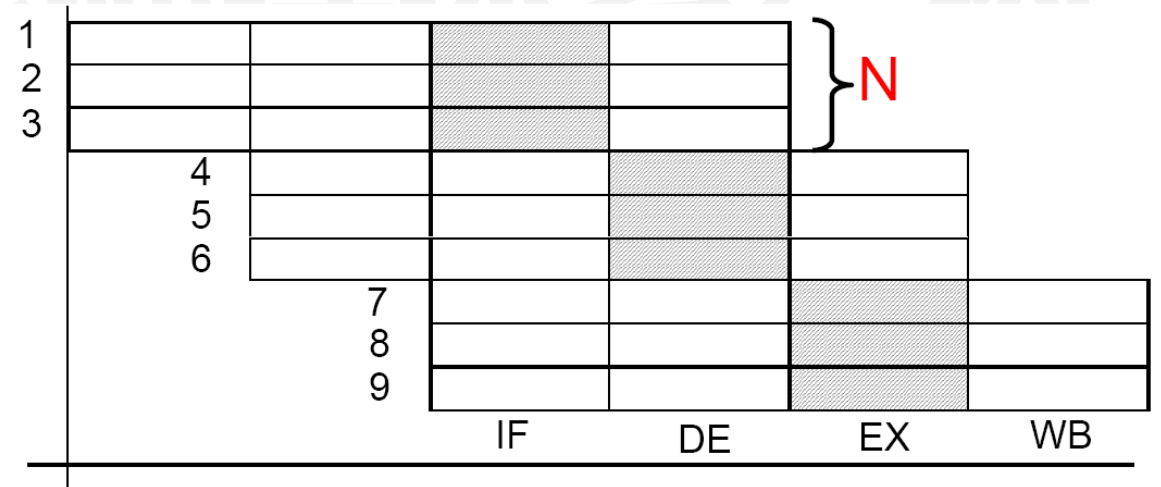
在cycle 4出现的问题

- `mulf` 因为 **RAW hazard** 而需要stall
 - 这里的stall是必要的
- `subf` 因 **pipeline hazard** 而需要stall
 - `subf` 不能进入Decode, 因为`mulf`在Decode那里stall
 - `subf`的stall是不必要的

所以为什么不`subf`在cycle4就进入Decode呢?

Out-Of-Order: 乱序执行的概念

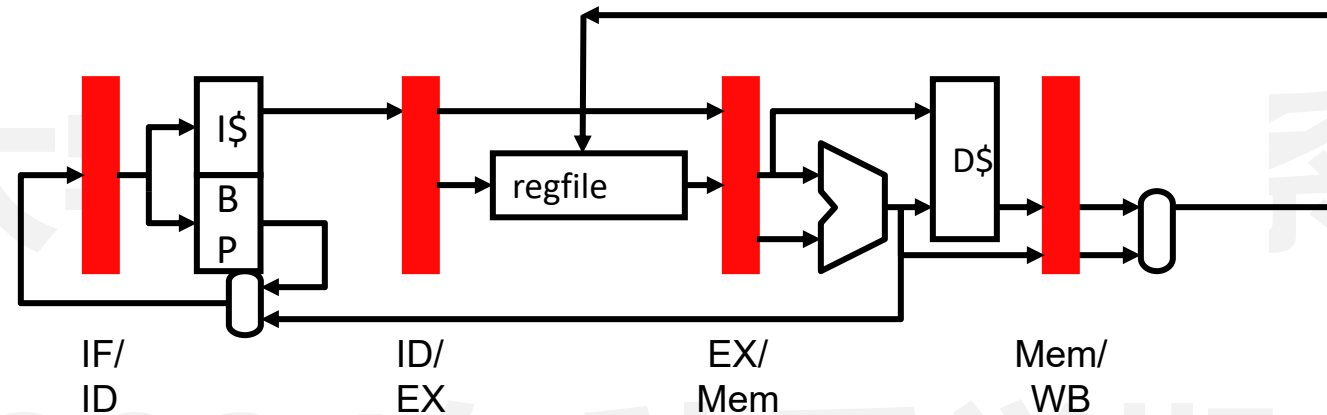
- Instruction-level parallelism
 - 如果并行度超过一定的阈值，顺序执行流水线的CPI将会严重下降
 - 当相关指令的平均距接近为 $N \times M$
 - 前馈不再有效
 - 因为频繁的stall，流水线将会一直是空闲的



耀宇、李萌

Out-Of-Order: 乱序执行的概念

• The Problem With In-Order Pipelines



• In-order pipeline

- **Structural hazard:** 流水线每级只有一个指令寄存器
 - 每周期每级只有一个指令（除非流水线复制多份）
 - 后级指令不能越过前级指令 without “clobbering” it

• Out-of-order pipeline

- 通过去除**structural hazard**的方法来实现指令的跳跃执行

Out-Of-Order: 乱序执行的概念

- 乱序执行完全在硬件实现
- 动态调度
 - 完全在硬件中
 - 也称为“乱序执行” (OoO)
- 将许多指令提取到指令窗口中
 - 使用分支预测推测过去 (多个) 分支
 - 在分支错误预测时刷新流水线
- 重命名以避免错误的依赖关系 (WAW 和 WAR)
- 尽可能快的执行指令
 - 寄存器依赖项是已知的
 - 处理内存依赖关系更棘手 (稍后会详细介绍)
- 按顺序提交指令
 - 任何奇怪的事情都会在 commit 之前发生, 只需 flush pipeline
- 当前机器: 100+ 指令调度窗口

乱序执行

以非顺序执行指令...

减少 RAW 停顿

提高流水线和功能单元 (FU) 利用率

最初的动机是提高 FP 单位的利用率

为并行执行行 (ILP) 提供更多机会

不按顺序-可以并行

...但要让它看起来像顺序执行

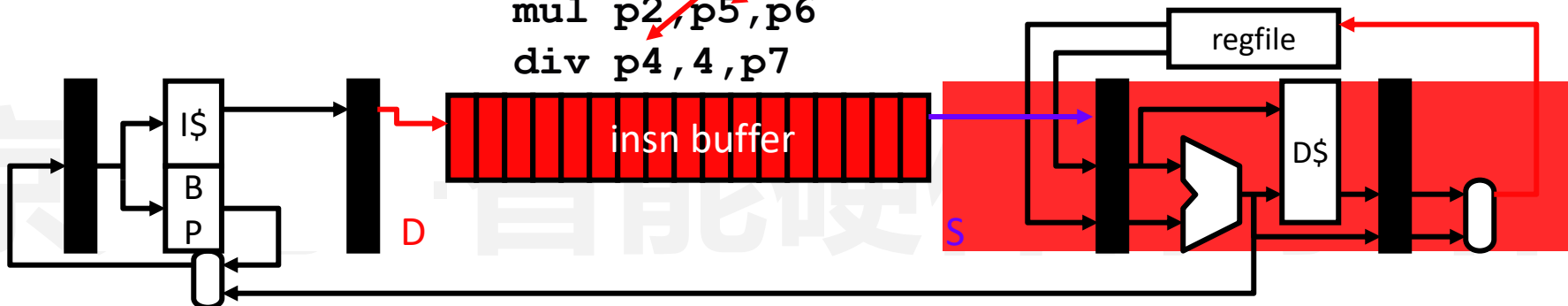
重要但困难, 接下来会讲到

Out-Of-Order: 乱序执行的概念

- 乱序执行如何实现?

```

add p2, p3, p4
sub p2, p4, p5
mul p2, p5, p6
div p4, 4, p7
    
```



Ready Table

	P2	P3	P4	P5	P6	P7
	Yes	Yes				
	Yes	Yes	Yes			
	Yes	Yes	Yes	Yes		Yes
t ↓	Yes	Yes	Yes	Yes	Yes	Yes

```

add p2, p3, p4
sub p2, p4, p5 and div p4, 4, p7
mul p2, p5, p6
    
```

- 指令获取/解码/重命名后进入 *Instruction Buffer*
 - 也叫做 “instruction window” 或 “instruction scheduler”
- 指令每周期检查是否就绪
 - 就绪后执行

- 数据依赖存在于原始任务逻辑，与硬件体系结构如何设计无关

- 依赖项独立于硬件而存在

- 如果 Inst #1000 需要 Inst #1 的结果，则存在依赖关系

- 只有当硬件必须处理它时，它才是一种冲突hazard

- 当硬件不需要处理的时候，hazard不存在

主讲：陶耀宇、李萌

- True/False Data Dependencies

- 真数据依赖

- RAW – Read after Write

$$R1 = R2 + R3$$

$$R4 = R1 + R5$$

- True dependencies prevent reordering
 - 大部分不可避免

- 假数据依赖

- WAW – Write after Write

$$R1 = R2 + R3$$



$$R1 = R4 + R5$$

- WAR – Write after Read

$$R2 = R1 + R3$$



$$R1 = R4 + R5$$

- False dependencies prevent reordering
 - 通过renaming可以被消除

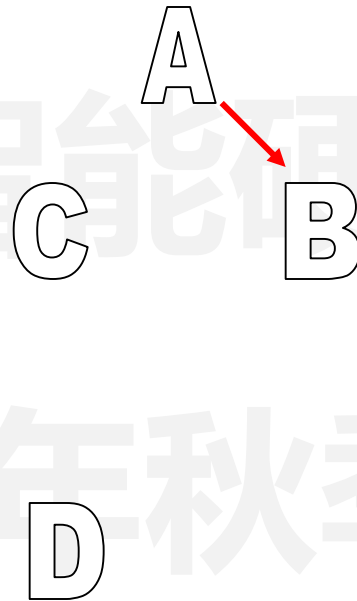
- True/False Data Dependencies

$R1 = \text{MEM}[R2 + 0]$ // A

$R2 = R2 + 4$ // B

$R3 = R1 + R4$ // C

$\text{MEM}[R2 + 0] = R3$ // D



主讲：陶耀宇、李萌

■ RAW ■ WAW ■ WAR

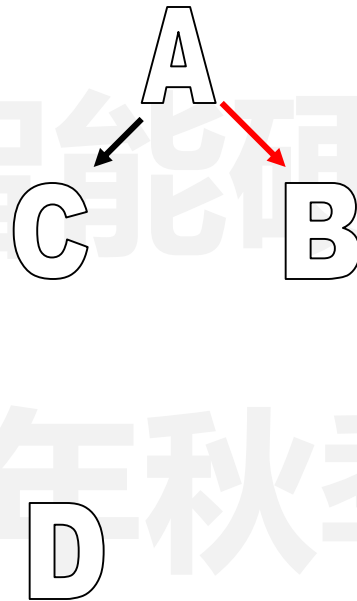
- True/False Data Dependencies

$R1 = \text{MEM}[R2 + 0]$ // A

$R2 = R2 + 4$ // B

$R3 = R1 + R4$ // C

$\text{MEM}[R2 + 0] = R3$ // D



主讲：陶耀宇、李萌

■ RAW ■ WAW ■ WAR

数据依赖图

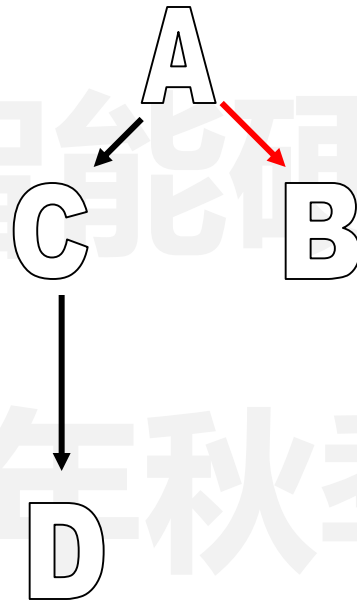
- True/False Data Dependencies

$R1 = \text{MEM}[R2 + 0]$ // A

$R2 = R2 + 4$ // B

$R3 = R1 + R4$ // C

$\text{MEM}[R2 + 0] = R3$ // D



主讲：陶耀宇、李萌

■ RAW

■ WAW

■ WAR

数据依赖图

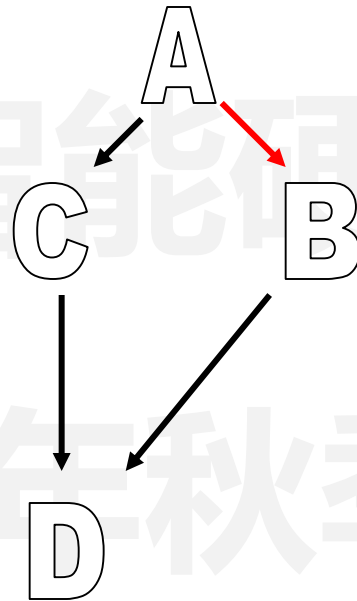
- True/False Data Dependencies

$R1 = \text{MEM}[R2 + 0]$ // A

$R2 = R2 + 4$ // B

$R3 = R1 + R4$ // C

$\text{MEM}[R2 + 0] = R3$ // D



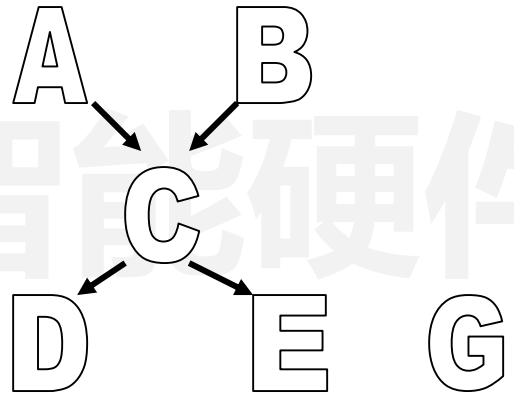
主讲：陶耀宇、李萌

■ RAW ■ WAW ■ WAR

数据依赖图

- True/False Data Dependencies

```
R1=MEM[R3+4] // A
R2=MEM[R3+8] // B
R1=R1*R2 // C
MEM[R3+4]=R1 // D
MEM[R3+8]=R1 // E
```



```
R1=MEM[R3+12] // F
R2=MEM[R3+16] // G
R1=R1*R2 // H
MEM[R3+12]=R1 // I
MEM[R3+16]=R1 // J
```

2024年秋季学期

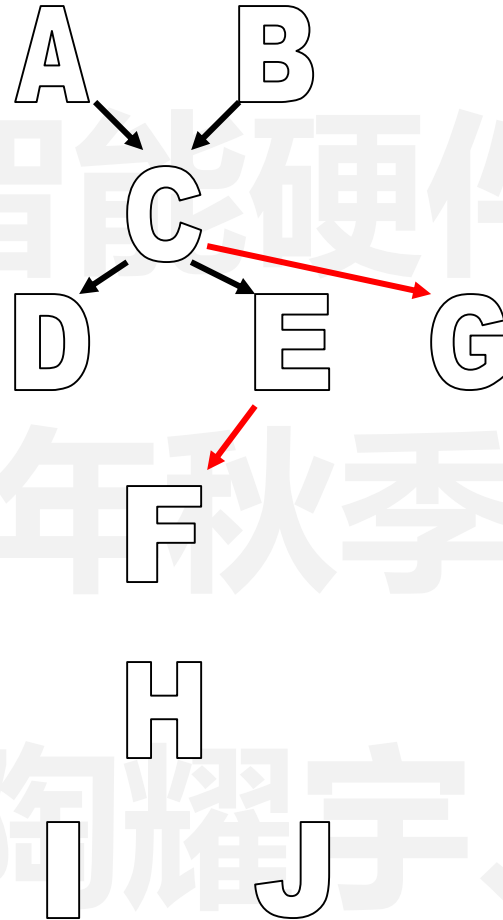
主讲：陶耀宇、李萌

■ RAW ■ WAW ■ WAR

数据依赖图

• True/False Data Dependencies

```
R1=MEM[R3+4] // A
R2=MEM[R3+8] // B
R1=R1*R2 // C
MEM[R3+4]=R1 // D
MEM[R3+8]=R1 // E
R1=MEM[R3+12] // F
R2=MEM[R3+16] // G
R1=R1*R2 // H
MEM[R3+12]=R1 // I
MEM[R3+16]=R1 // J
```

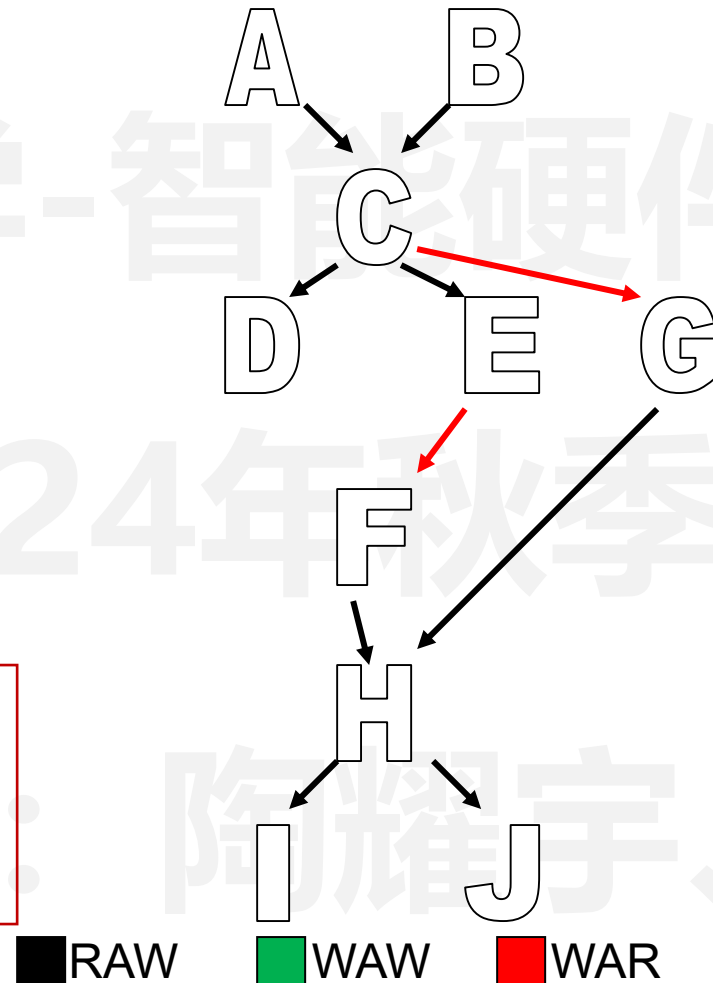


■ RAW ■ WAW ■ WAR

数据依赖图

• True/False Data Dependencies

```
R1=MEM[R3+4] // A
R2=MEM[R3+8] // B
R1=R1*R2 // C
MEM[R3+4]=R1 // D
MEM[R3+8]=R1 // E
R1=MEM[R3+12] // F
R2=MEM[R3+16] // G
R1=R1*R2 // H
MEM[R3+12]=R1 // I
MEM[R3+16]=R1 // J
```



- 从逻辑上讲, F-J 没有理由依赖于 A-E. So.....
 - ABFG
 - CH
 - DEIJ
 - Should be possible.
- 但这会导致 C 或 H 具有错误的 reg 输入
- 如何解决?
 - Remember, the dependency is really on the *name* of the register
 - So... 改变寄存器名称即可!

- 寄存器Register重命名概念

- 寄存器名称是任意的
- 寄存器名称只需要在写入之间保持一致

R1 =

.... = R1

.... = ... R1

R1 =

R1中的值是有效的，在写入到最后一次读取期间；
在下一次写R1之前，值都是对的

主讲：陶耀宇、李萌

寄存器重命名

寄存器Register重命名机制 – 将实际电路寄存器与虚拟架构寄存器解耦

- 每次写入架构寄存器时，我们都会将其分配给物理寄存器

- 在再次写入架构寄存器之前，我们将继续将其转换为物理寄存器号
保持 RAW 依赖项不变

- 很简单，让我们看一个例子:

- Architecture Regs:** r1, r2, r3

- Physical Regs:** p1, p2, p3, p4, p5, p6, p7

- Original mapping: r1→p1, r2→p2, r3→p3, p4-p7 are "free"

Architecture register

虚拟的架构寄存器 – 汇编代码中

Physical register

实际的电路寄存器 – 硬件架构中

RAT (Alias T)

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

FreeList

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

Orig. insns

```

add r2, r3, r1
sub r2, r1, r3
mul r2, r3, r3
div r1, 4, r1
    
```

Renamed insns

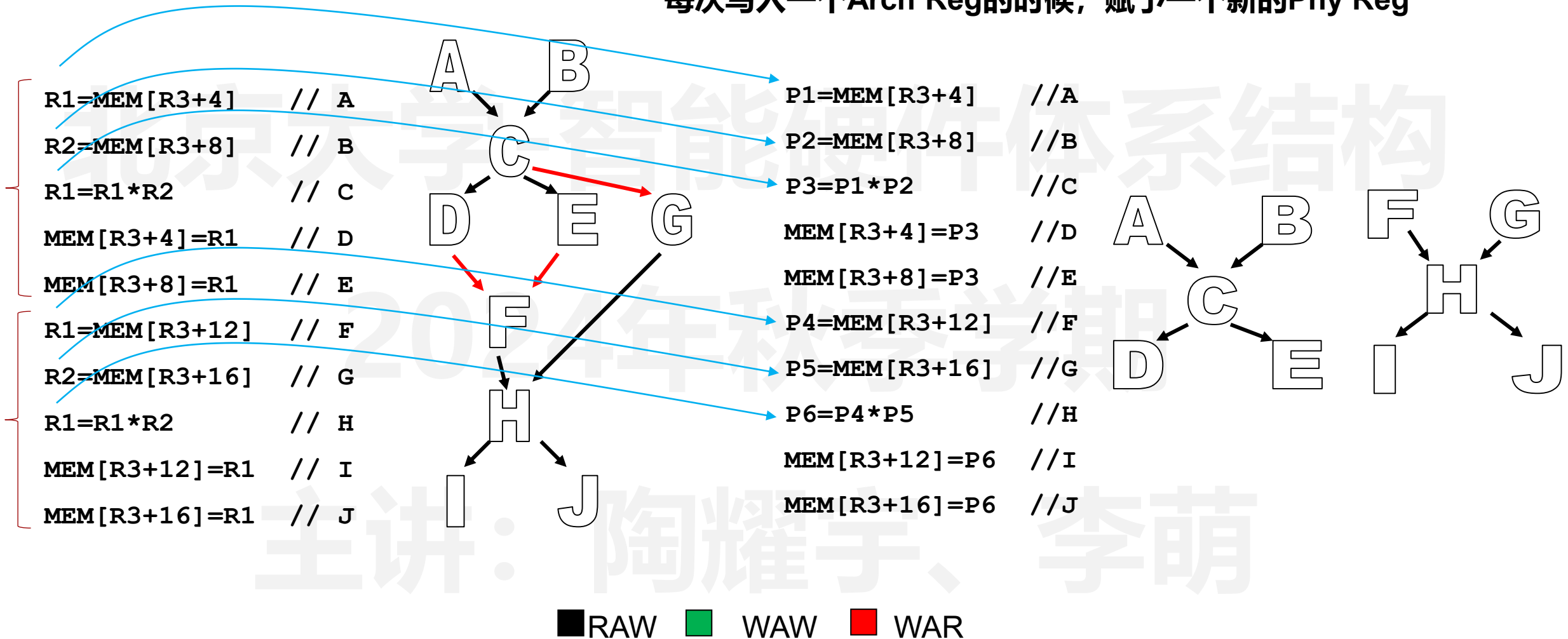
```

add p2, p3, p4
sub p2, p4, p5
mul p2, p5, p6
div p4, 4, p7
    
```

寄存器重命名

• 寄存器Register重命名的效果

每次写入一个Arch Reg的时候, 赋予一个新的Phy Reg



- 寄存器Register重命名硬件设计

- Really simple table (Reg Alias Table, RAT)
 - **每次一个指令写入寄存器冲突时为其分配一个新的物理寄存器编号**
- 但存在一些复杂性
 - 什么时候可以清空物理寄存器?
 - 下一章乱序执行中会讲

主讲：陶耀宇、李萌

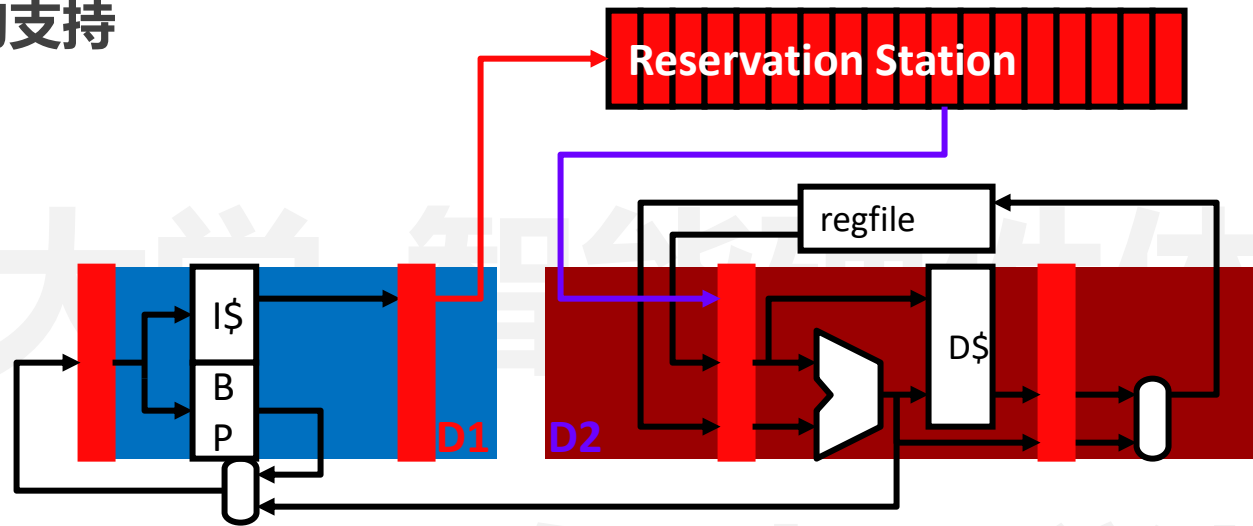
目录

CONTENTS



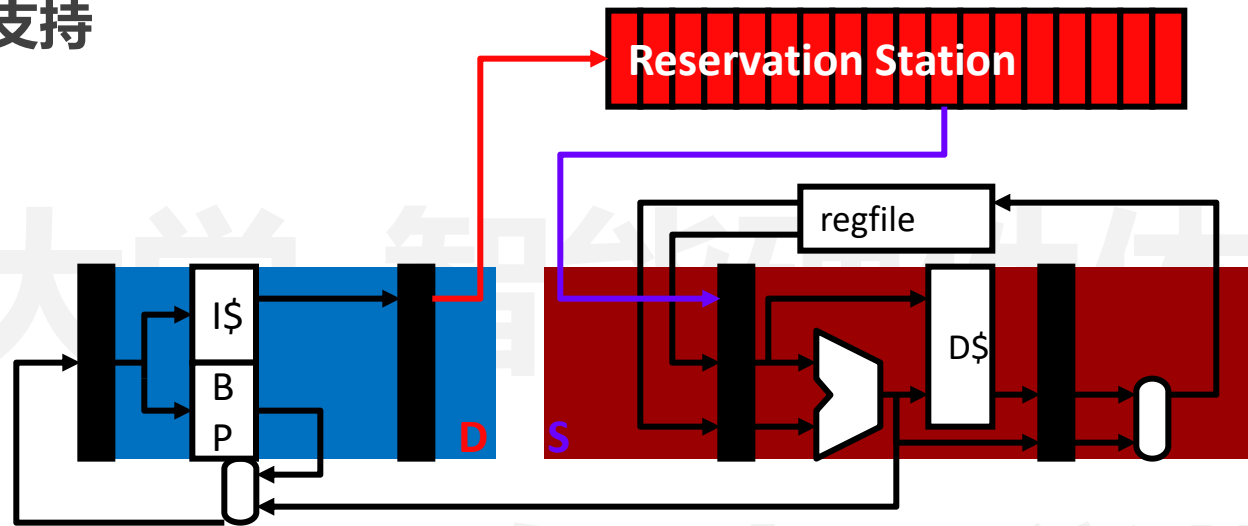
01. 超标量架构数据控制冲突
02. 动态发射与乱序执行设计
03. 分支处理机制与地址预测
04. 经典的MIPS架构实例分析

- 乱序执行的架构支持



- 指令缓冲区或保留区 Insn buffer or Reservation station (RS) (有很多名字)
 - 基本单元: 保持指令的锁存器
 - 指令候选池
- 将 ID 拆分为两部分
 - 缓冲区**顺序**获取解码的指令
 - RS将指令送往下行流水线**乱序**执行

• 乱序执行的架构支持



- **Dispatch (D):** 解码的第一部分 (原来的ID分解为2步)
 - 在RS指令缓冲中分配位置
 - 新型结构冲突Structure Hazard (指令缓冲区已满)
 - 顺序执行: **stall** back-propagates to younger insns
- **Issue (S):** 解码的第二部分 (原来的ID分解为2步)
 - 将指令从RS缓冲送到执行单元
 - + 乱序执行: **wait** doesn't back-propagate to younger insns

动态发射与乱序执行设计

• 指令动态发射算法

- **调度算法**: 根据寄存器依赖关系进行调度

• 两种基本调度算法

- Scoreboard: 无寄存器重命名 → 有限的调度灵活性
- Tomasulo: 寄存器重命名 → 更灵活, 性能更佳
- 我们着重介绍Tomasulo算法
- Scoreboard算法没有测试问题
 - 注意在特定GPU中它会被用到

• Issue

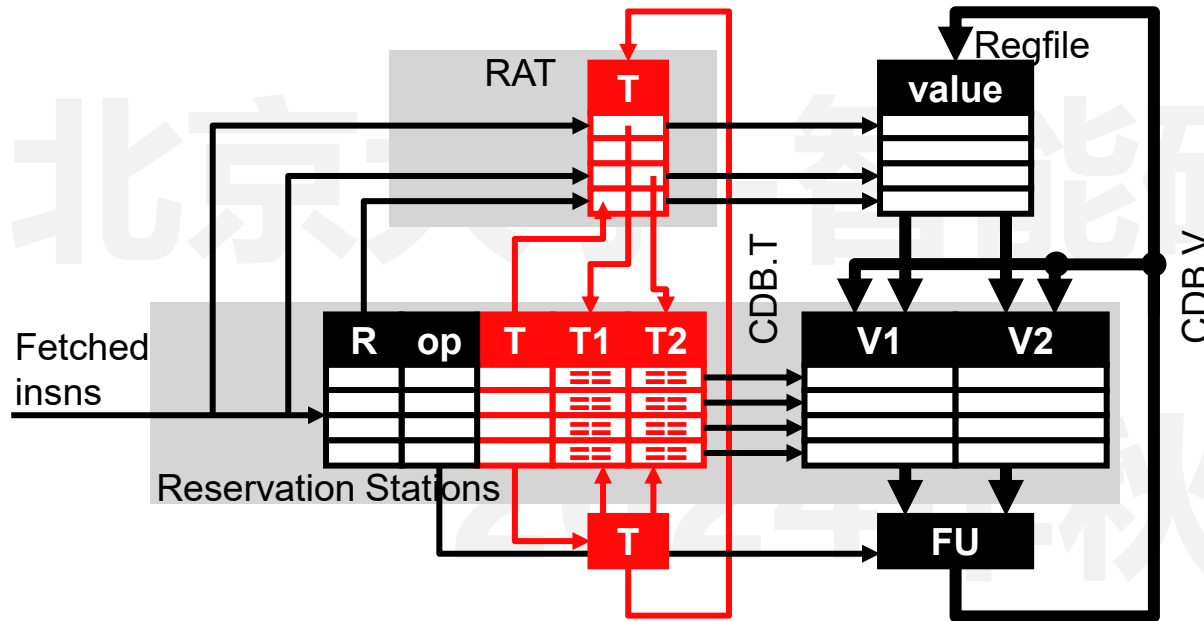
- 如果有多条指令就绪, 该选择哪一条? **Issue policy**
 - 最靠前的先执行? 安全
 - 最长延迟优先执行? 可能带来更好的性能
- **Select logic**: implements issue policy
 - 大多数芯片使用随机或优先级编码器

Tomasulo动态指令发射算法

- 指令动态发射算法
- **Tomasulo' s algorithm**
 - **预留站 Reservation stations (RS)**: 指令缓冲区
 - **通用数据总线 Common data bus (CDB)**: 将结果广播到 RS
 - **寄存器重命名 Register renaming**: 消除 WAR/WAW 数据依赖
- 首次实现: IBM 360/91 -> Modern x86 CPU -> GPU -> ASIC **仍在使用!**
 - 适用于针对多计算单元的动态调度
- 我们的简单示例: “Simple Tomasulo”
 - 对一切进行动态调度, 包括加载/存储
 - 5 RS entry: 1 ALU, 1 load, 1 store, 2 FP (3-cycle, pipelined)

Tomasulo动态指令发射算法

• Tomasulo算法的基础结构



- Insn fields and status bits
- Tags
- Values

- Reservation Stations (RS#)
 - **FU, busy, op, R**: destination register name
 - **T**: destination register tag (RS# of this RS)
 - **T1, T2**: source register tags (RS# of RS that will produce value)
 - **V1, V2**: source register values
- Rename Table/Map Table/RAT
 - **T**: tag (RS#) that will write this register
- Common Data Bus (CDB)
 - Broadcasts $\langle RS\#, value \rangle$ of completed insns
- Tags interpreted as ready-bits++
 - $T=0 \rightarrow$ Value is ready somewhere
 - $T \neq 0 \rightarrow$ Value is not ready, wait until CDB broadcasts T

Tomasulo动态指令发射算法

• Tomasulo算法的基础结构

- Reservation Stations (RS#)
 - **R**: 目标寄存器名称, Busy: 表明RS是否空闲, Op: 存储指令操作类型
 - **T**: 目标寄存器标签 (RS# of this RS)
 - **T1, T2**: 源寄存器标签 (RS# of RS that will produce value)
 - **V1, V2**: 源寄存器值
- Rename Table/Map Table/RAT
 - **T**: 将重命名该寄存器的标签 (RS#)
- Common Data Bus (CDB)
 - 广播已完成指令的 <RS#, value>
- Tags interpreted as ready-bits++
 - $T=0$ → 价值在某处已经准备好
 - $T \neq 0$ → 值尚未准备好, 等待 CDB 广播 T

- Tomasulo算法的新增步骤

- 新的流水线结构: F, **D**, **S**, X, **W**

- **D (dispatch)**

- **Structural** hazard ? **stall** : 分配RS空间

- **S (issue)**

- **RAW** hazard ? **wait** (monitor CDB) : go to execute

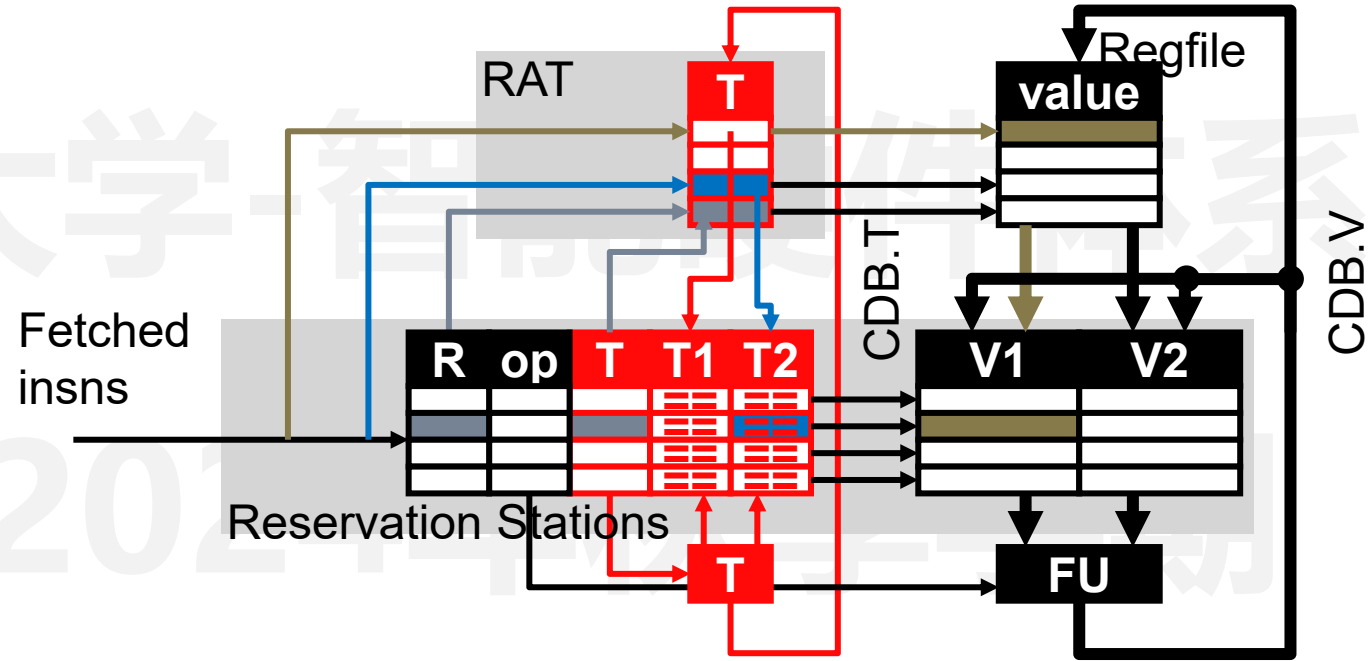
- **W (writeback)**

- 写入寄存器 (sometimes...), 释放RS空间
 - W与具有RAW依赖的S在同一周期完成
 - W与具有结构依赖的D在同一周期完成

Tomasulo动态指令发射算法

• Tomasulo算法步骤

Tomasulo Dispatch (D)

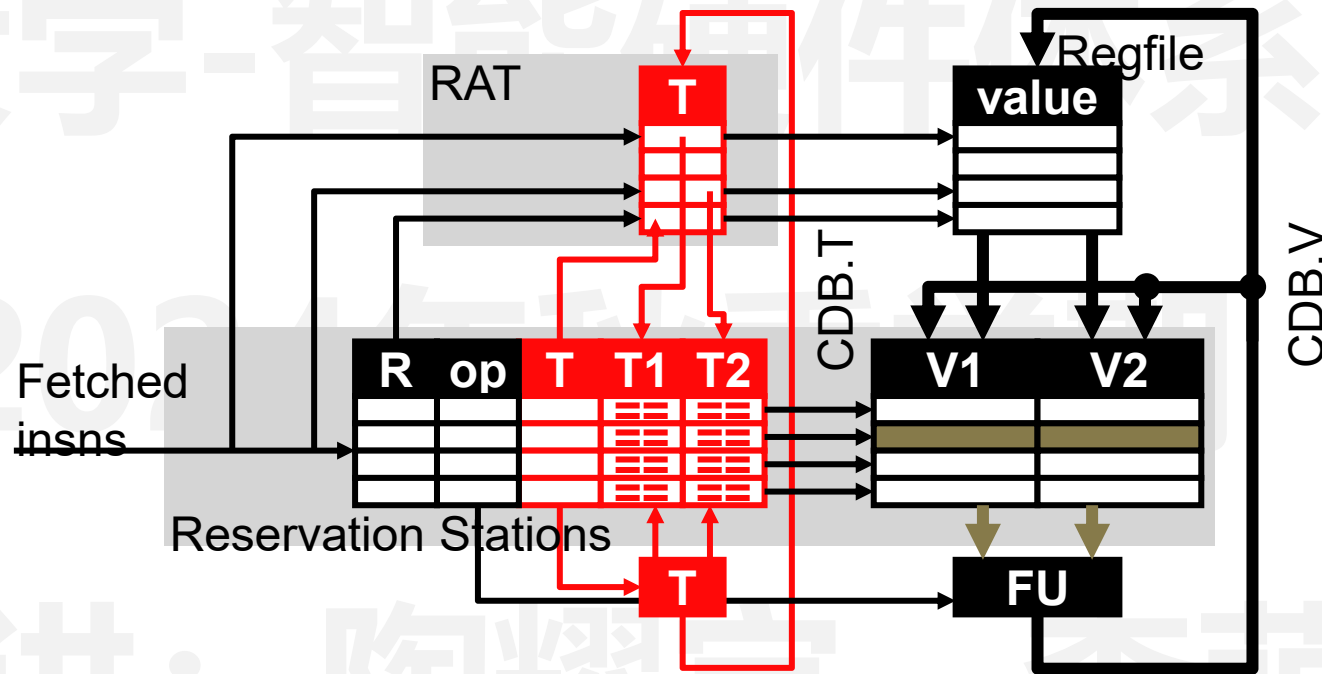


• RS结构冲突可能带来的Stall

- 分配 RS 空间
- 输入寄存器准备好了吗? 将值读入RS: 将标签读入RS
- 将输出寄存器重命名为 RS# (代表唯一值的“名称”)

- Tomasulo算法步骤

Tomasulo Issue (S)

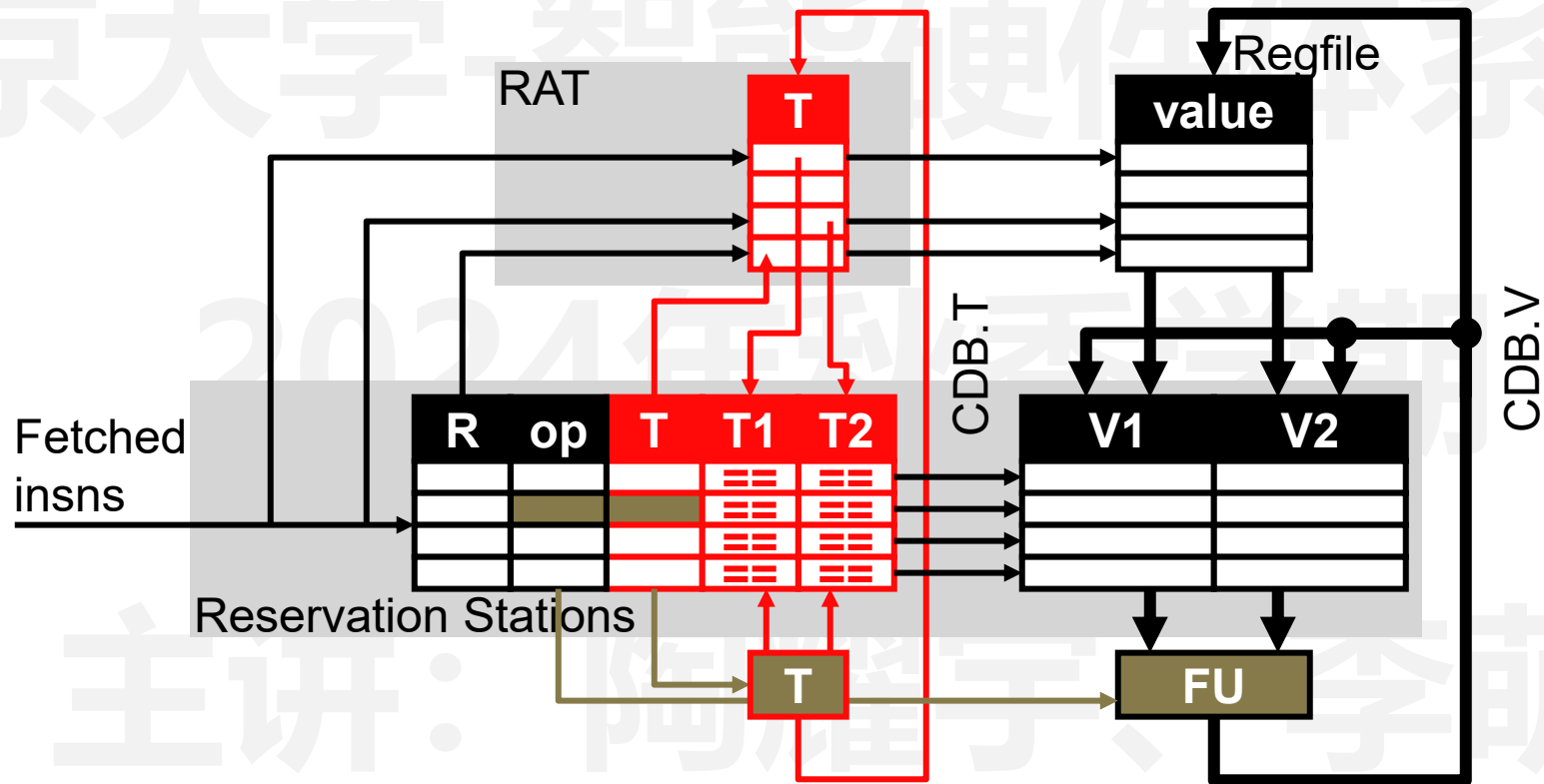


- 因RAW冲突而等待
- 从RS读取寄存器值

Tomasulo动态指令发射算法

- Tomasulo算法步骤

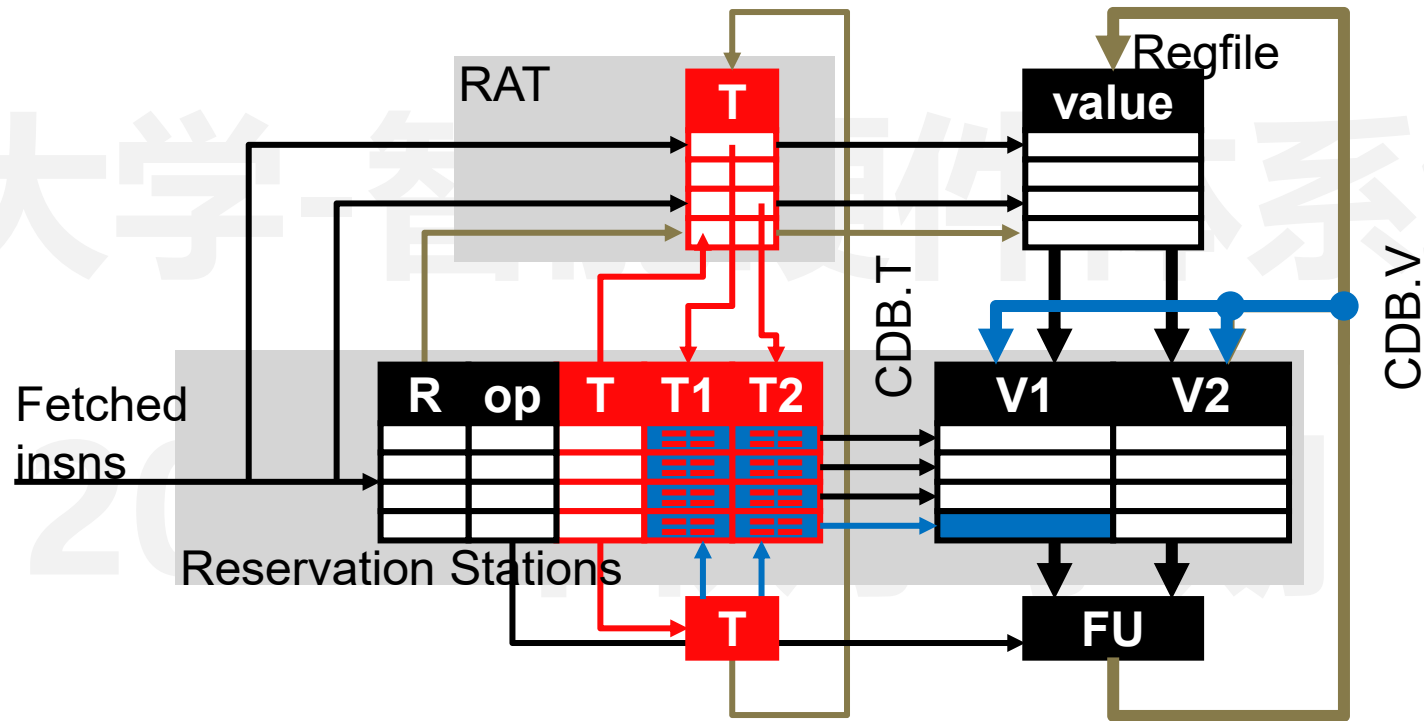
Tomasulo Execute (X)



Tomasulo动态指令发射算法

• Tomasulo算法步骤

Tomasulo Writeback (W)

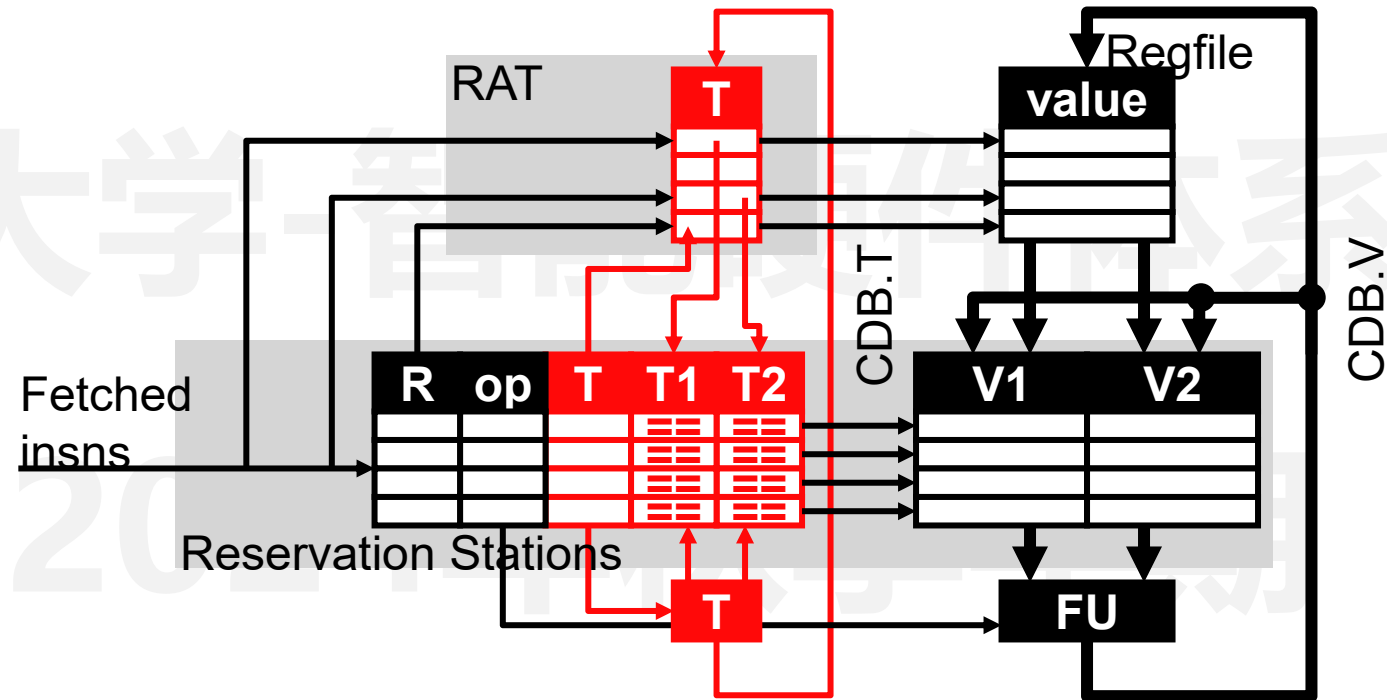


- 因CDB结构冲突而等待
 - 如果RAT 重命名仍然匹配? 清除映射, 将结果写入regfile
 - CDB 广播到 RS: 标签匹配? 清除标签, 复制值
 - 清除RS中相应存储

Tomasulo动态指令发射算法

• Tomasulo算法步骤

Tomasulo Register Renaming



- Tomasulo 的寄存器重命名中做了什么？
 - RS 中的值复制 (V1、V2)
 - Insn 在其自己的 RS 位置中存储正确的输入值
 - + Future insns can overwrite master copy in regfile, doesn' t matter

Tomasulo动态指令发射算法

- Value-based / Copy-based Register Renaming

- Tomasulo-style register renaming

- Called “**value-based**” or “**copy-based**”

- **Names:** 架构寄存器

- **存储位置: 寄存器堆或RS**

- 值可以并确实存储在两者之中

- **寄存器堆保存主 (即最新) 值**

- + **RS 副本消除了 WAR 危险**

- RS的标签表明了存储位置

- **Register table 将名称转换为标签**

- Tag == 0 的值位于寄存器文件中

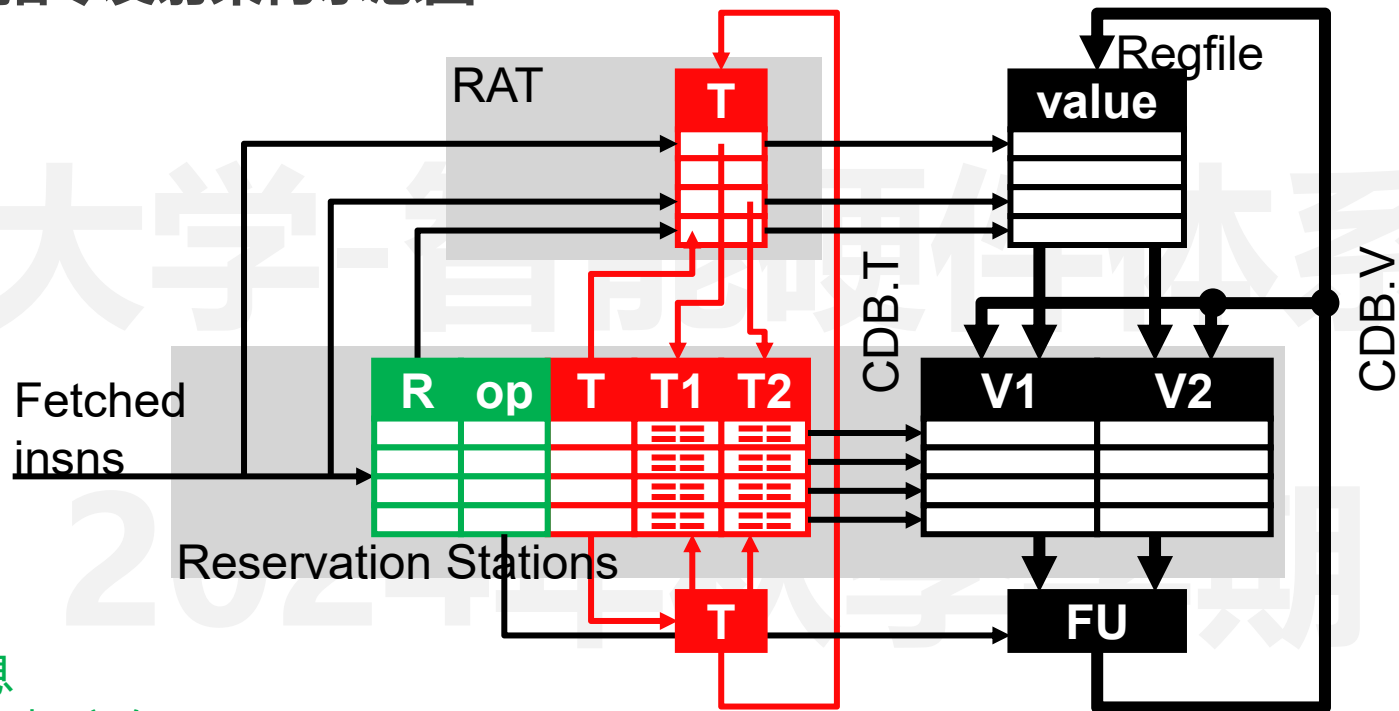
- Tag != 0 的值尚未准备好, 正在由 RS# 计算

- CDB 广播带有标签的值

- 所以指令知道他们正在寻找什么值

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

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

Insn Status				
Insn	D	S	X	W
ldf X(r1), f1				
mulf f0, f1, f2				
stf f2, Z(r1)				
addi r1, 4, r1				
ldf X(r1), f1				
mulf f0, f1, f2				
stf f2, Z(r1)				

Map Table	
Reg	T
f0	
f1	
f2	
r1	

CDB	
T	V

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	no						
4	FP1	no						
5	FP2	no						

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

Insn Status				
Insn	D	S	X	W
ldf X(r1), f1	c1			
mulf f0, f1, f2				
stf f2, Z(r1)				
addi r1, 4, r1				
ldf X(r1), f1				
mulf f0, f1, f2				
stf f2, Z(r1)				

Map Table	
Reg	T
f0	
f1	RS#2
f2	
r1	

CDB	
T	V

Tomasulo:

Cycle 1

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	no						
2	LD	yes	ldf	f1	-	-	-	[r1] allocate
3	ST	no						
4	FP1	no						
5	FP2	no						

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

Insn Status				
Insn	D	S	X	W
ldf X(r1), f1	c1	c2		
mulf f0, f1, f2	c2			
stf f2, Z(r1)				
addi r1, 4, r1				
ldf X(r1), f1				
mulf f0, f1, f2				
stf f2, Z(r1)				

Map Table	
Reg	T
f0	
f1	RS#2
f2	RS#4
r1	

CDB	
T	V

Tomasulo:

Cycle 2

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	no						
2	LD	yes	ldf	f1	-	-	-	[r1]
3	ST	no						
4	FP1	yes	mulf	f2	-	RS#2	[f0]	-
5	FP2	no						

allocate

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

Insn Status				
Insn	D	S	X	W
ldf X(r1), f1	c1	c2	c3	
mulf f0, f1, f2	c2			
stf f2, Z(r1)	c3			
addi r1, 4, r1				
ldf X(r1), f1				
mulf f0, f1, f2				
stf f2, Z(r1)				

Map Table	
Reg	T
f0	
f1	RS#2
f2	RS#4
r1	

CDB	
T	V

Tomasulo:

Cycle 3

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	no						
2	LD	yes	ldf	f1	-	-	-	[r1]
3	ST	yes	stf	-	RS#4	-	-	[r1]
4	FP1	yes	mulf	f2	-	RS#2	[f0]	-
5	FP2	no						

allocate

等待 insn #1的结果

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

Tomasulo:
Cycle 4

Insn Status				
Insn	D	S	X	W
ldf X(r1), f1	c1	c2	c3	c4
mulf f0, f1, f2	c2	c4		
stf f2, Z(r1)	c3			
addi r1, 4, r1	c4			
ldf X(r1), f1				
mulf f0, f1, f2				
stf f2, Z(r1)				

Map Table	
Reg	T
f0	
f1	RS#2
f2	RS#4
r1	RS#1

CDB	
T	V
RS#2	[f1]

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	yes	addi	r1	-	-	[r1]	-
2	LD	no						
3	ST	yes	stf	-	RS#4	-	-	[r1]
4	FP1	yes	mulf	f2	-	RS#2	[f0]	CDB.V
5	FP2	no						

Ldf指令完成(W)
通过CDB广播f1的寄存状态

allocate
free

RS#2 ready →
获取CDB的值

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

Insn Status				
Insn	D	S	X	W
ldf X(r1), f1	c1	c2	c3	c4
mulf f0, f1, f2	c2	c4	c5	
stf f2, Z(r1)	c3			
addi r1, 4, r1	c4	c5		
ldf X(r1), f1	c5			
mulf f0, f1, f2				
stf f2, Z(r1)				

Map Table	
Reg	T
f0	
f1	RS#2
f2	RS#4
r1	RS#1

CDB	
T	V

Tomasulo:

Cycle 5

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	yes	addi	r1	-	-	[r1]	-
2	LD	yes	ldf	f1	-	RS#1	-	-
3	ST	yes	stf	-	RS#4	-	-	[r1]
4	FP1	yes	mulf	f2	-	-	[f0]	[f1]
5	FP2	no						

allocate

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

假设 `multf` 需要3个cycle完成

Insn Status				
Insn	D	S	X	W
<code>ldf X(r1), f1</code>	c1	c2	c3	c4
<code>multf f0, f1, f2</code>	c2	c4	c5+	
<code>stf f2, Z(r1)</code>	c3			
<code>addi r1, 4, r1</code>	c4	c5	c6	
<code>ldf X(r1), f1</code>	c5			
<code>multf f0, f1, f2</code>	c6			
<code>stf f2, Z(r1)</code>				

Map Table	
Reg	T
f0	
f1	
f2	RS#4RS#5
r1	RS#1

CDB	
T	V

Tomasulo:

Cycle 6

针对WAW不需要D处stall: scoreboard将覆盖f2的寄存状态, 如果需要旧f2值可以从tag获取

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	yes	addi	r1	-	-	[r1]	-
2	LD	yes	ldf	f1	-	RS#1	-	-
3	ST	yes	stf	-	RS#4	-	-	[r1]
4	FP1	yes	multf	f2	-	-	[f0]	[f1]
5	FP2	yes	multf	f2	-	RS#2	[f0]	-

allocate

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

假设 `multf` 需要3个cycle完成

Insn Status				
Insn	D	S	X	W
<code>ldf X(r1), f1</code>	c1	c2	c3	c4
<code>multf f0, f1, f2</code>	c2	c4	c5+	
<code>stf f2, Z(r1)</code>	c3			
<code>addi r1, 4, r1</code>	c4	c5	c6	c7
<code>ldf X(r1), f1</code>	c5	c7		
<code>multf f0, f1, f2</code>	c6			
<code>stf f2, Z(r1)</code>				

Map Table	
Reg	T
f0	
f1	RS#2
f2	RS#5
r1	RS#1

CDB	
T	V
RS#1	[r1]

Tomasulo:

Cycle 7

针对WAR不需要W处stall: scoreboard将覆盖r1的寄存状态, 如果需要旧r1值 可以从RS副本获取
D stall on store RS: 结构冲突

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	no						
2	LD	yes	<code>ldf</code>	f1	-	RS#1	-	CDB.V
3	ST	yes	<code>stf</code>	-	RS#4	-	-	[r1]
4	FP1	yes	<code>multf</code>	f2	-	-	[f0]	[f1]
5	FP2	yes	<code>multf</code>	f2	-	RS#2	[f0]	-

addi 完成 (W)
将r1寄存状态通过CDB广播

RS#1 ready → 获取CDB的值

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

假设 `mul f` 需要3个cycle完成

Insn Status				
Insn	D	S	X	W
<code>ldf X(r1), f1</code>	c1	c2	c3	c4
<code>mul f f0, f1, f2</code>	c2	c4	c5+	c8
<code>stf f2, Z(r1)</code>	c3	c8		
<code>addi r1, 4, r1</code>	c4	c5	c6	c7
<code>ldf X(r1), f1</code>	c5	c7	c8	
<code>mul f f0, f1, f2</code>	c6			
<code>stf f2, Z(r1)</code>				

Map Table	
Reg	T
f0	
f1	RS#2
f2	RS#5
r1	

CDB	
T	V
RS#4	[f2]

Tomasulo:

Cycle 8

mul f finished (W)

不要刷新f2的寄存状态

已经被第二个mul f指令覆盖了(RS#5)

CDB 广播f2

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	no						
2	LD	yes	<code>ldf</code>	f1	-	-	-	[r1]
3	ST	yes	<code>stf</code>	-	RS#4	-	CDB.V	[r1]
4	FP1	no						
5	FP2	yes	<code>mul f</code>	f2	-	RS#2	[f0]	-

**RS#4 ready →
grab CDB value**

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

假设 `multf` 需要3个cycle完成

Insn Status				
Insn	D	S	X	W
<code>ldf X(r1), f1</code>	c1	c2	c3	c4
<code>multf f0, f1, f2</code>	c2	c4	c5+	c8
<code>stf f2, Z(r1)</code>	c3	c8	c9	
<code>addi r1, 4, r1</code>	c4	c5	c6	c7
<code>ldf X(r1), f1</code>	c5	c7	c8	c9
<code>multf f0, f1, f2</code>	c6	c9		
<code>stf f2, Z(r1)</code>				

Map Table	
Reg	T
f0	
f1	RS#2
f2	RS#5
r1	

CDB	
T	V
RS#2	[f1]

Tomasulo:

Cycle 9

2nd `ldf` finished (W)
刷新 f1 寄存器状态
CDB broadcast

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	yes	<code>stf</code>	-	-	-	[f2]	[r1]
4	FP1	no						
5	FP2	yes	<code>multf</code>	f2	-	RS#2	[f0]	CDB.V

RS#2 ready →
grab CDB value

Tomasulo动态指令发射算法

• Tomasulo动态指令发射实例

假设 `multf` 需要3个cycle完成

Insn Status				
Insn	D	S	X	W
<code>ldf X(r1), f1</code>	c1	c2	c3	c4
<code>multf f0, f1, f2</code>	c2	c4	c5+	c8
<code>stf f2, Z(r1)</code>	c3	c8	c9	c10
<code>addi r1, 4, r1</code>	c4	c5	c6	c7
<code>ldf X(r1), f1</code>	c5	c7	c8	c9
<code>multf f0, f1, f2</code>	c6	c9	c10	
<code>stf f2, Z(r1)</code>	c10			

Map Table	
Reg	T
f0	
f1	
f2	RS#5
r1	

CDB	
T	V

Tomasulo:

Cycle 10

`stf` finished (W)

map table中没有输出寄存器→不通过CDB广播

Reservation Stations								
T	FU	busy	op	R	T1	T2	V1	V2
1	ALU	no						
2	LD	no						
3	ST	yes	stf	-	RS#5	-	-	[r1]
4	FP1	no						
5	FP2	yes	multf	f2	-	-	[f0]	[f1]

free → allocate

超标量+动态指令发射

• Tomasulo动态指令发射实例

• 动态调度和多发射是正交的

- 例如, Pentium4: 动态调度的5路超标量
- 两个维度
 - **N**: 超标量宽度 (并行操作的数量)
 - **W**: 窗口大小 (保留站的数量)

• What do we need for an **N**-by-**W** Tomasulo?

- RS: **N** tag/value w-ports (D), **N** value r-ports (S), **2N** tag CAMs (W)
- 选择逻辑: **W**→**N** 优先级编码器(S)
- MT: **2N** r-ports (D), **N** w-ports (D)
- RF: **2N** r-ports (D), **N** w-ports (W)
- CDB: **N** (W)
- Which are the expensive pieces?

超标量+动态指令发射

• Tomasulo动态指令发射实例

- 超标量选择逻辑: $W \rightarrow N$ 优先编码器
 - 有点复杂 ($N^2 \log W$)
 - 可以简化使用不同的 RS 设计
- **split设计**
 - 除以 RS存入 N 个bank: 每个 FU 存入 1 个?
 - 实施 N 个单独的 $W/N \rightarrow 1$ 编码器
 - + 更简单: $N * \log W / N$
 - 调度灵活性较低
- **FIFO design**
 - 只能发射每个 RS bank的head
 - + 更简单: 根本没有选择逻辑
 - 时间安排灵活性较低 (但出乎意料的不算太糟糕)

目录

CONTENTS



01. 超标量架构数据控制冲突
02. 动态发射与乱序执行设计
03. 分支处理机制与地址预测
04. 经典的MIPS架构实例分析

- 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）完成后会发生怎样

- 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**

主讲：陶耀宇、李萌

- 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:
 - 总是被采取; 向后被采取

- **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**

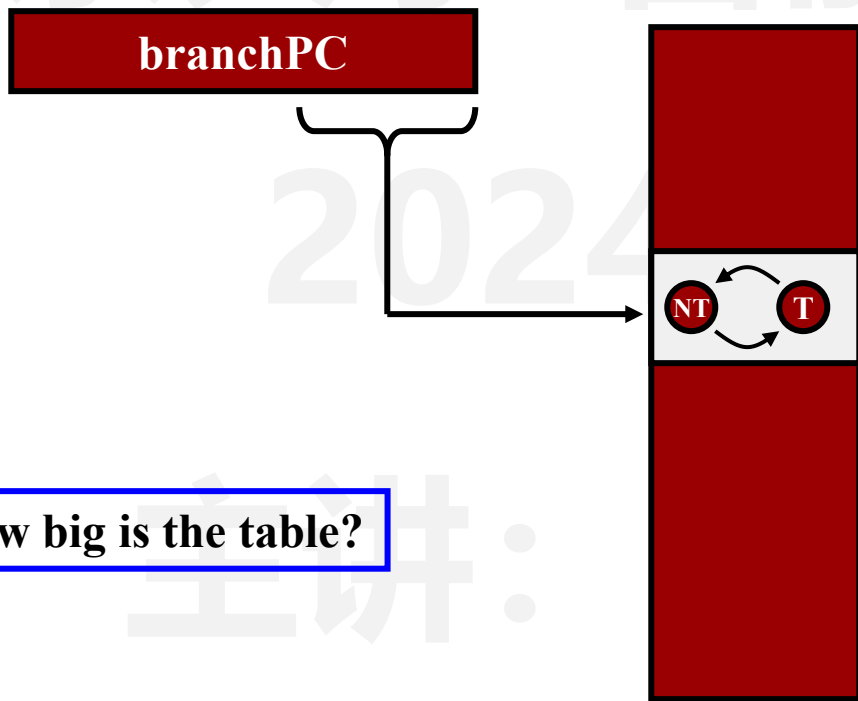
- **恢复逻辑**

分支预测

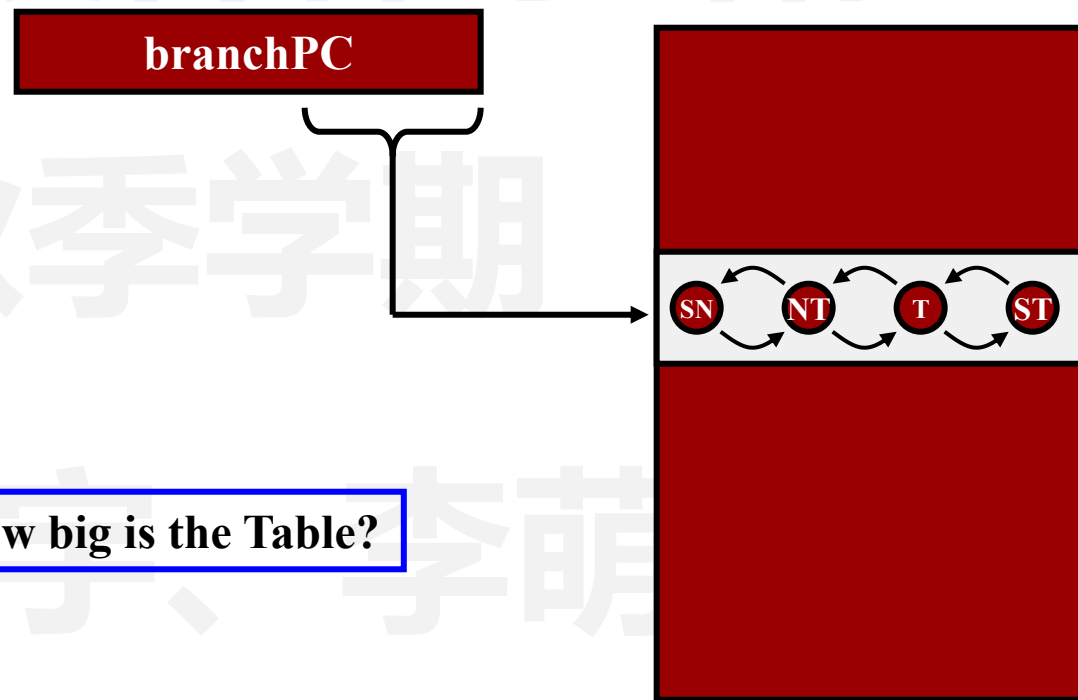
- 方向预测 – 基于历史的简单状态机FSM

- 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)



- 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%，我们需要查看参考模式，看看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)

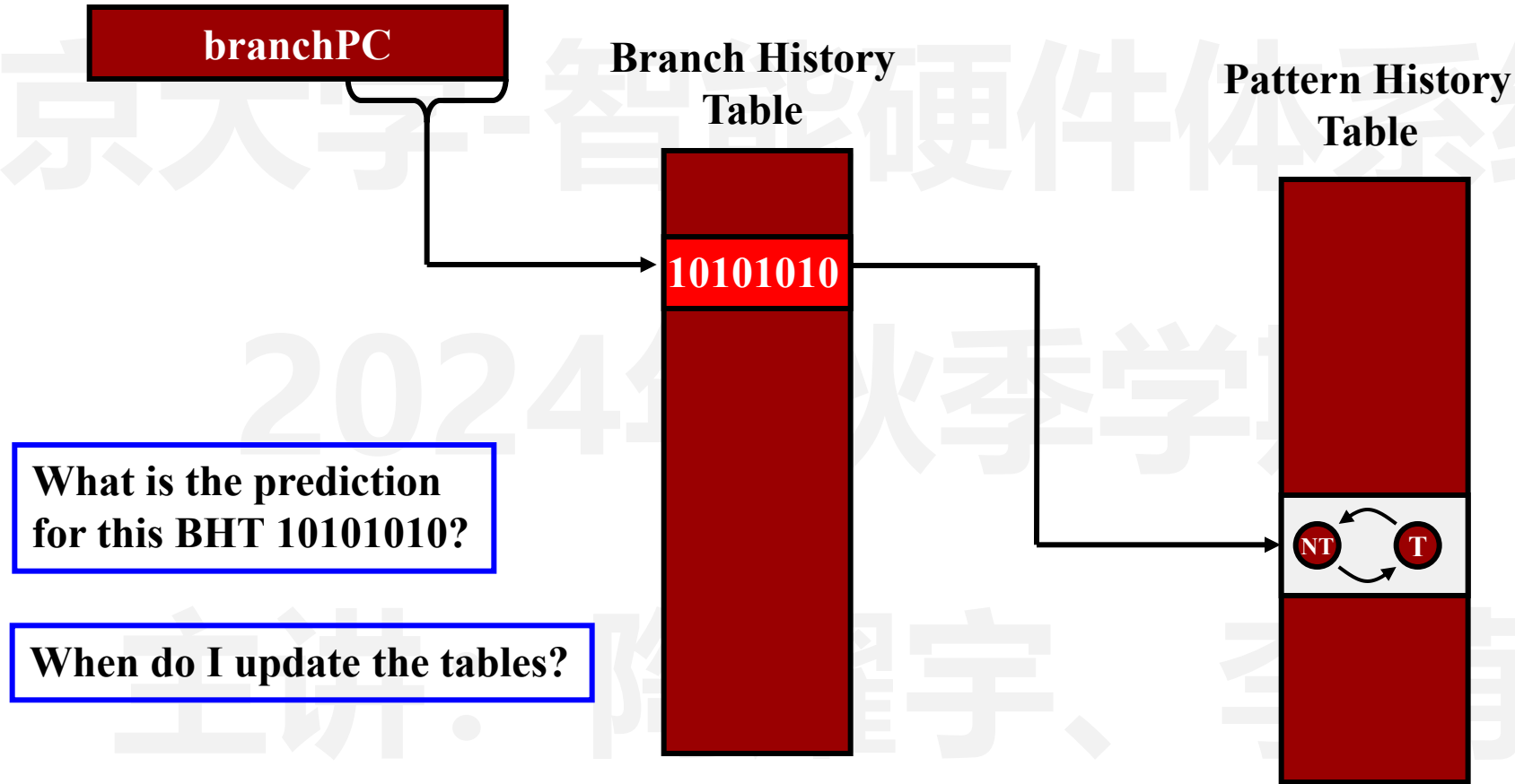
10

Using History Patterns

分支预测

- 方向预测 – 基于历史的简单状态机FSM

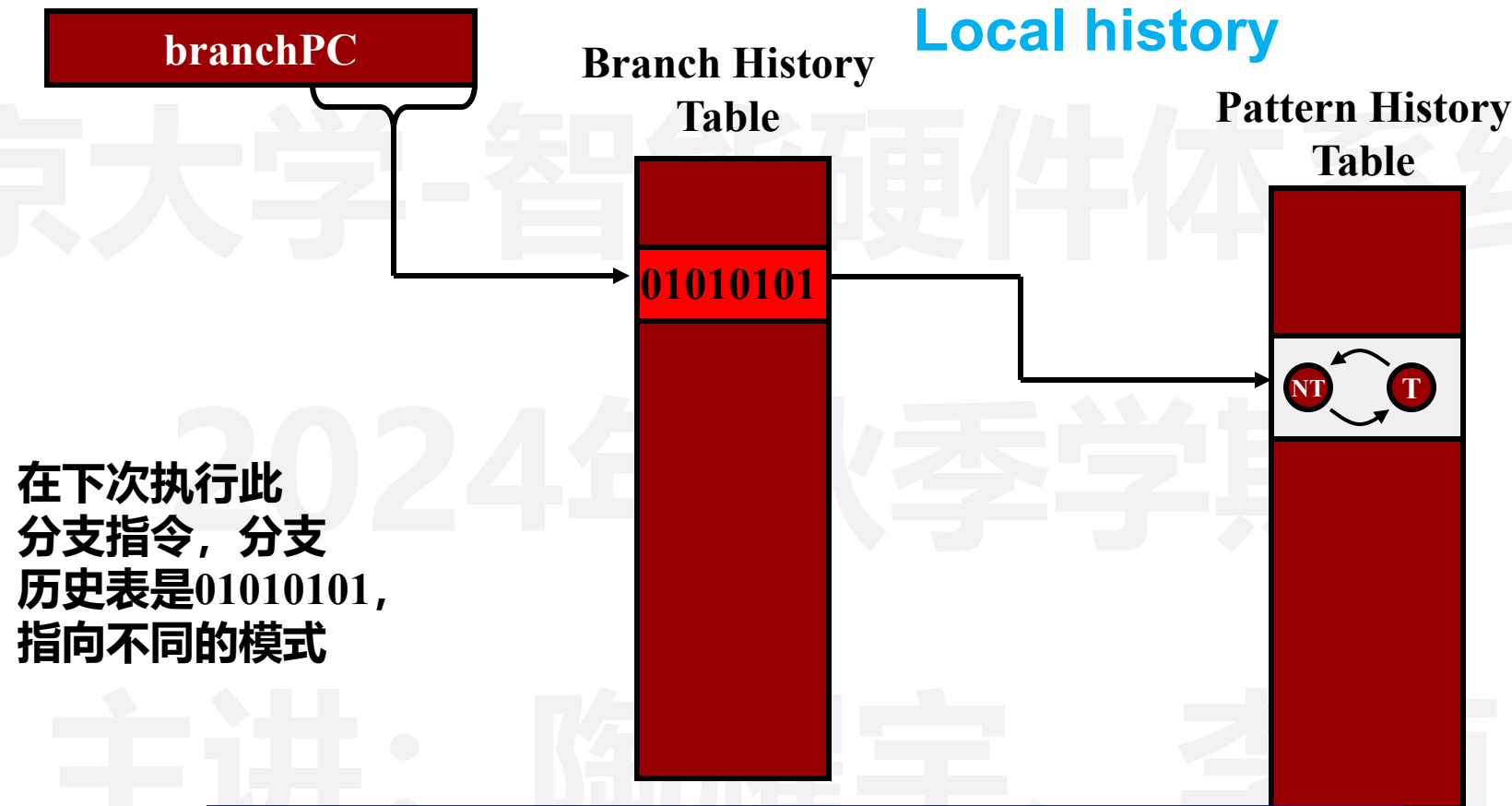
Local history



Using History Patterns

分支预测

- 方向预测 – 基于历史的简单状态机FSM



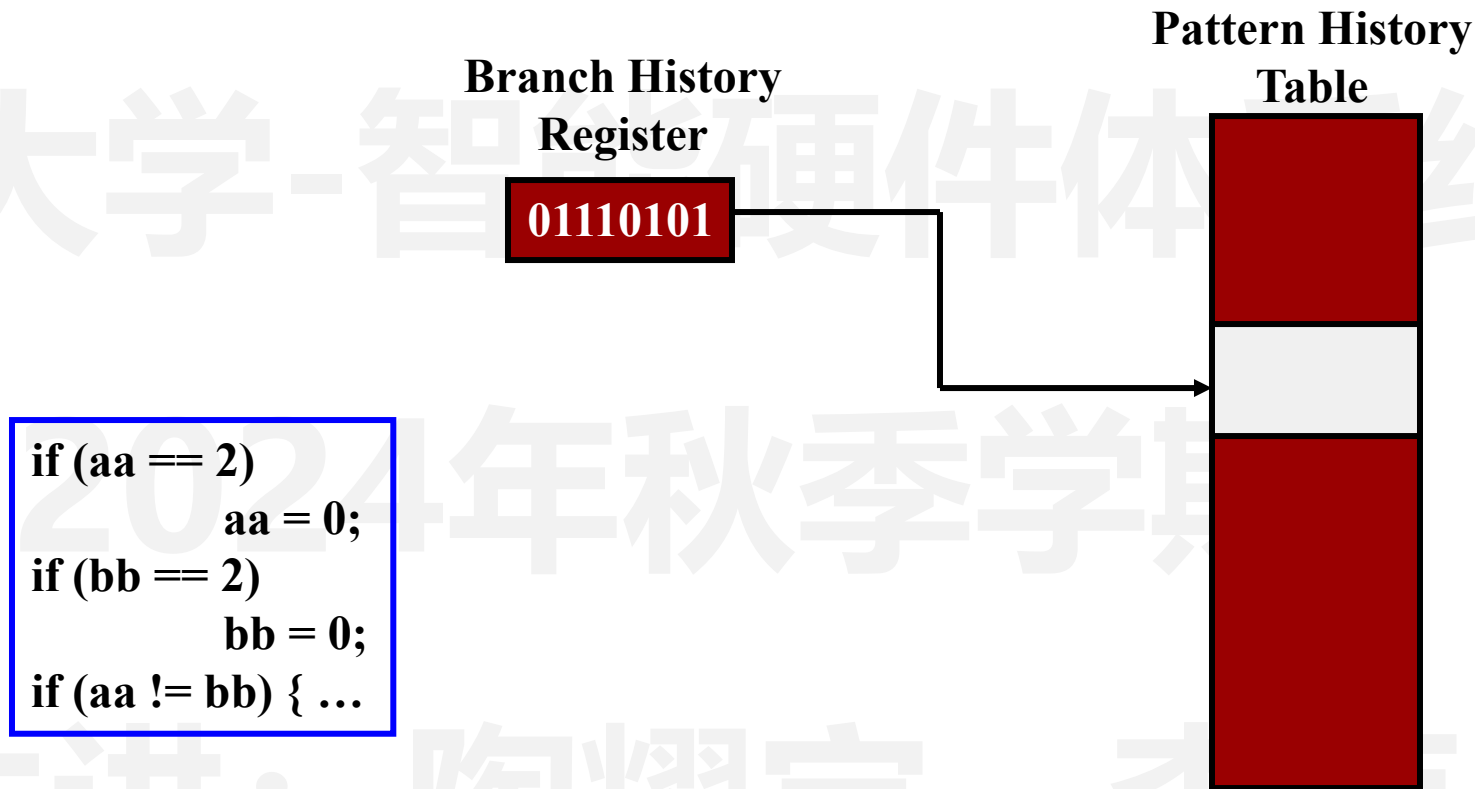
在下次执行此
分支指令，分支
历史表是01010101，
指向不同的模式

What is the accuracy of a flip/flop branch 0101010101010...?

Using History Patterns

- 方向预测 – 基于历史的简单状态机FSM

Global history



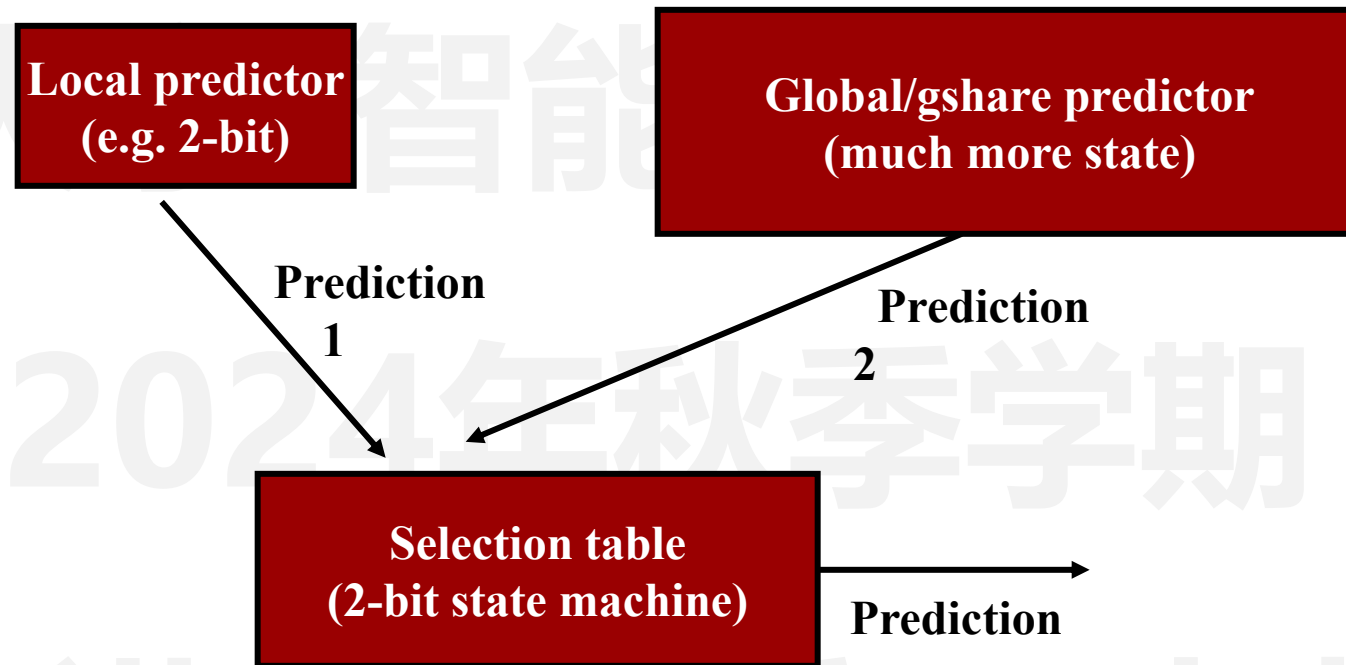
How can branches interfere with each other?

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

- 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 (作为 FA 太慢)
- 通常由分支预测器限定

Branch PC	Target address
0x05360AF0	0x05360000
...	...
...	...
...	...
...	...
...	...

- 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

Tamosulo

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 级管道中，状态仅在 MEM（用于存储）和 WB（用于寄存器）期间提交完成

- 恢复似乎真的很难

- 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

$R1 = \text{MEM}[R2 + 0]$

$\text{BEQ } R1, R3 \text{ DONE} \leftarrow \text{Predicted not taken}$

$R4 = R5 + R6$

- 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。

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.

- **指示按顺序完成**

- **指示仍然无序执行**

- 还是用RS

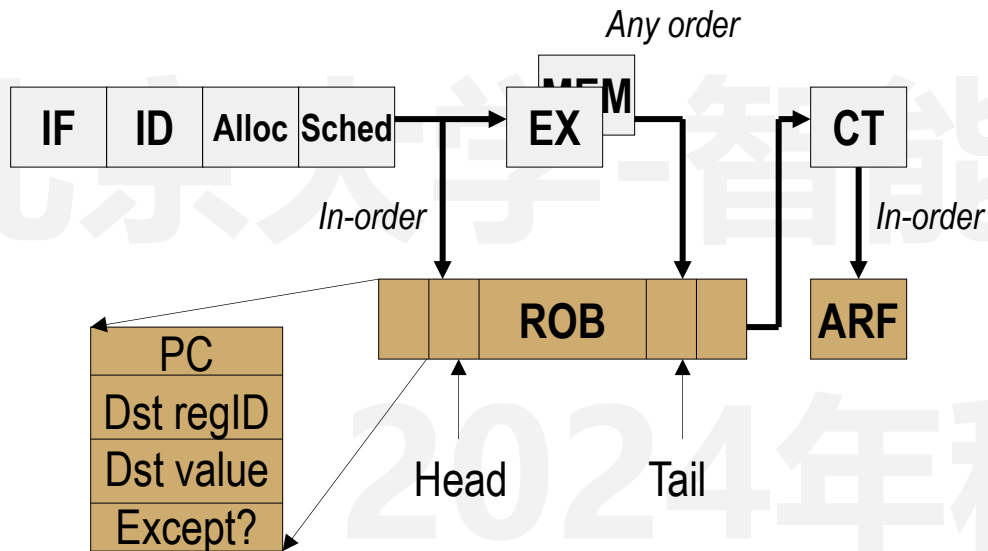
- **同时向RS和ROB发出指令**

- 重命名为 ROB 条目, 而不是 RS。

- 什么时候执行完成指令离开 RS

- 仅当程序顺序中该指令之前的所有指令都完成后, 该指令才会退出

- 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**

- 在尾部分配结果存储

- @ **Sched**

- 获取输入(ROB T-to-H then ARF)
- W等到所有输入准备就绪

- @ **WB**

- 将结果/错误写入 ROB
- 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

