

流水线技术(1)

执行指令过程

❖ 每条指令均可在 5 个时钟周期内完成



第一段：取指令(IF)

❖ 取指令(Instruction Fetch, IF)

- ❑ 向存储器发出程序计数器(Program Counter, PC)内容
- ❑ 从存储器中取回当前指令
- ❑ 通过加 4 更新 PC 内容指向顺序的下一条指令

❖ 优化

- ❑ 指令缓冲器
- ❑ 指令预取

❖ 性能影响

- ❑ 代码密度
- ❑ 可变的指令长度

第二段：指令译码与取寄存器(ID)

- ❖ 指令译码与取寄存器(Instruction Decode/Register Fetch)
 - ✧ 指令译码
 - ✧ 读源寄存器；为分支转移指令完成“相等”比较
 - ✧ 偏移量符号扩展；为分支转移指令完成目标地址计算
- ❖ 指令译码和读寄存器同时进行
 - ✧ **固定域译码**：在 RISC 架构中寄存器标识符在指令的固定位置
- ❖ 性能影响
 - ✧ 指令格式的规律性
 - ✧ 指令长度
- ❖ 分支指令目标地址计算的两种选择
 - ✧ 指令译码阶段
 - ✧ 指令执行阶段

第三段：执行与有效地址(EX)

- ❖ 执行与有效地址(Execution / Effective address)
 - ✧ 存储器访问：ALU 将基址寄存器和偏移量相加形成有效地址
 - ✧ 寄存器-寄存器：ALU 对从寄存器文件中读出的数值完成由 ALU 操作码指定的操作
 - ✧ 寄存器-立即数：ALU 对从寄存器文件中读出的第一个操作数和经过符号扩展的立即数完成 ALU 操作码指定的操作
 - ✧ 条件分支：确定是否条件为真
- ❖ 二者能够在单个时钟周期完成，是因为没有指令在计算某个数据地址的时候同时对其进行操作。
- ❖ 上面操作所需的所有内容均在 ID 段获得

第四段：存储器访问(MEM)

- ❖ 存储器访问(memory access)
 - ✧ 存储器访问：获得有效地址，完成 LOAD 和 STORE 指令
 - ✧ 分支/跳转：使用计算得到的有效地址置位 PC
- ❖ 对于所有操作，有效地址均在前一个流水段计算完成。

第五段：写回(WB)

❖ 写回(Write Back)

- ✧ 只对寄存器-寄存器 ALU 指令或 LOAD 指令有效
- ✧ 将操作结果(ALU指令)或读入的数据(LOAD指令)写回到目标寄存器中

RISC架构的非流水实现

- ❖ 利用合理配置的硬件资源，每个阶段都可以在一个时钟周期内完成。
- ❖ 每条指令执行最多需要 5 个时钟周期
 - ✧ 分支转移指令：3 个时钟周期
 - ✧ STORE 指令：4 个时钟周期
 - ✧ 其它指令：5 个时钟周期
- ❖ 例：假设分支转移指令和 STORE 指令的频率分别为 12% 和 10%，则一个典型指令分布的 $CPI = 4.66$ 。
- ❖ 该实现无论从最佳性能上还是从最小硬件数量上都不是最优的。
 - ✧ 容易理解

指令格式

- ❖ 所有指令采用 32 位编码，其中操作码为 6 位。

opcode

arguments

- ❖ 只有两种寻址方式，三种指令类型：I 类型、R 类型和 J 类型

- ❖ 缩写

缩写符	含义
opcode	6 位操作码
rs	5 位源寄存器说明符
rt	5 位目标(源/目的)寄存器或分支条件寄存器说明符
immediate	16 位立即数，分支转移位移或地址位移
target	26 位跳转目标地址
rd	5 位目的寄存器说明符
shamt	5 位移位量
funct	6 位功能域

I 类指令



❖ 指令

- ❑ load 和 store 指令
- ❑ 所有带有立即数的指令
- ❑ 寄存器跳转指令，寄存器跳转和链接指令
- ❑ 条件分支指令

❖ 注意事项

- ❑ 5 比特编码源和目标寄存器：最多32个寄存器
- ❑ 16比特编码立即数

R 类指令



❖ 指令

✧ 寄存器-寄存器 ALU 操作

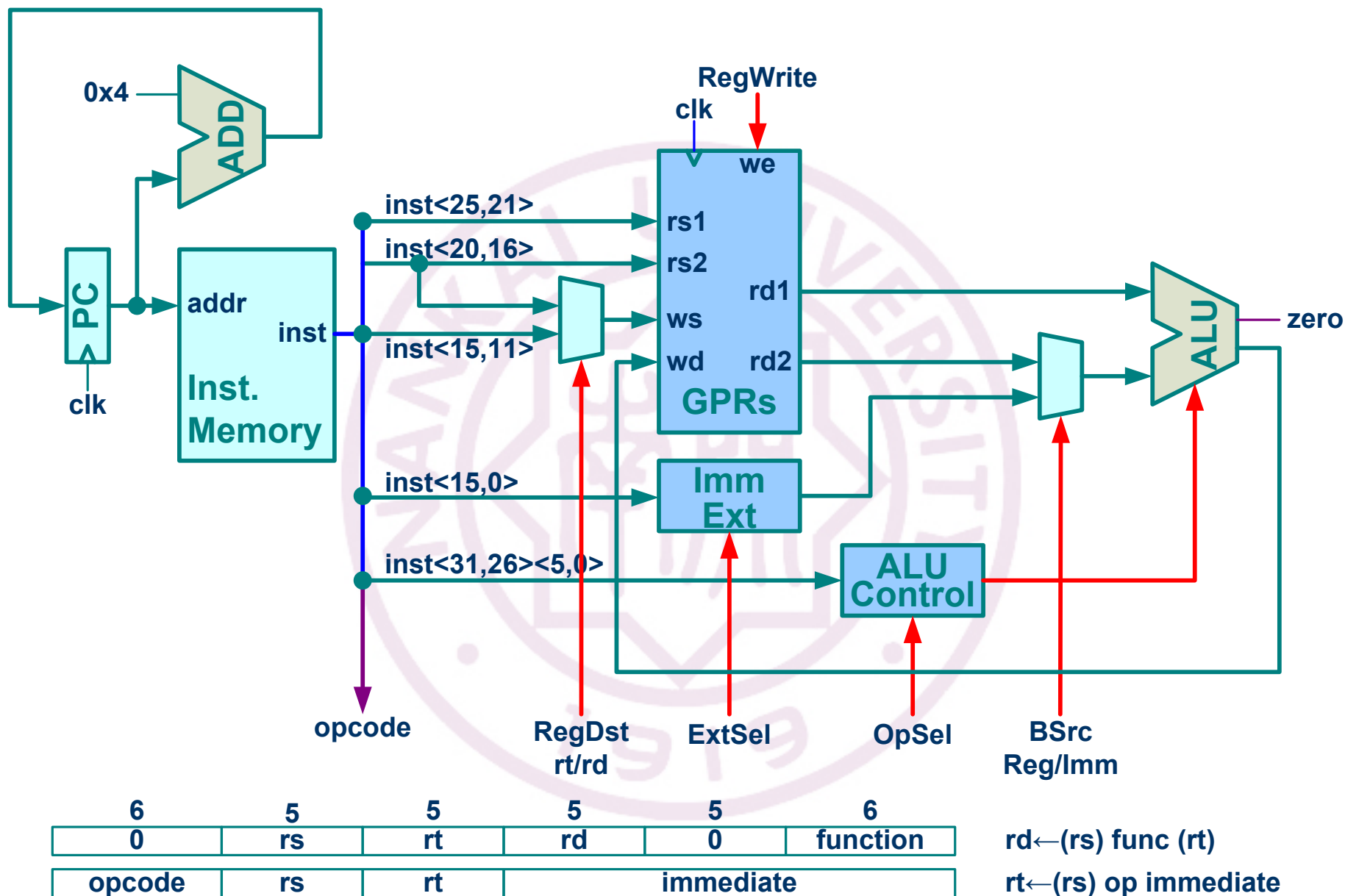
J 类指令



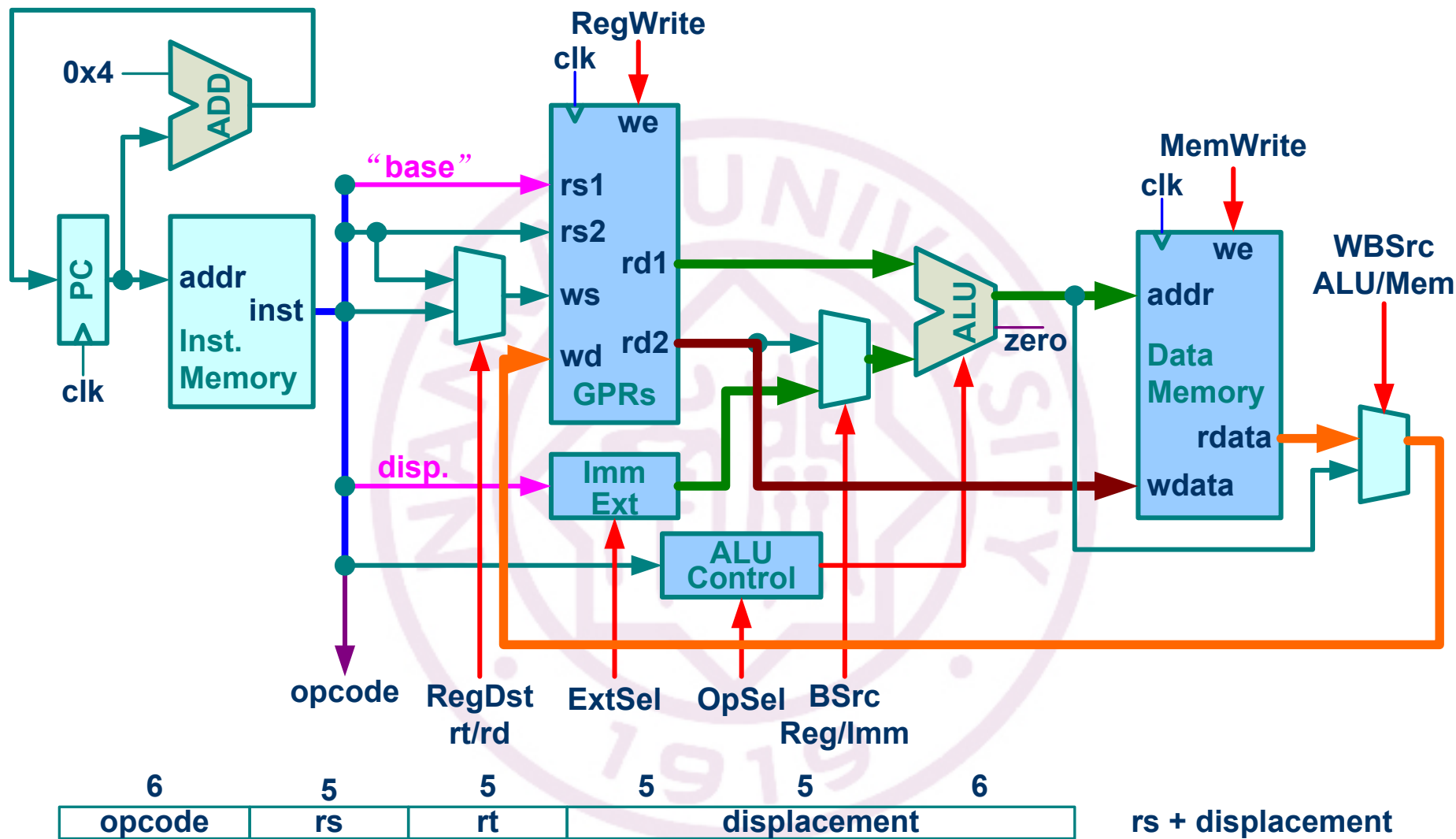
❖ 指令

- ❑ 跳转，跳转和链接
- ❑ 陷阱，从意外返回

ALU指令数据通路



LOAD和STORE指令数据通道 (哈佛结构)

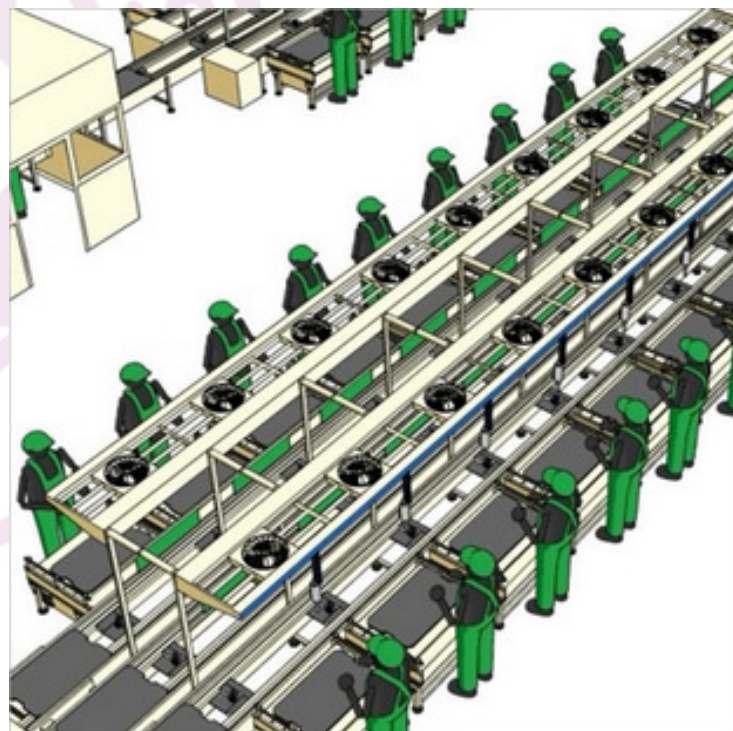


rs is the base register.

rt is the destination of a Load or the source for a Store.

什么是流水线

- ❖ 流水线(pipelining)是用来制造快速处理器的关键实现技术
 - ✧ 利用连续指令流中指令间存在的并行性，使得多条指令重叠执行。
- ❖ 类似于装配线(assembly line)
 - ✧ 包含许多同时进行的步骤（作用对象不同）
 - ✧ 每个步骤都对最终产品做出一些贡献



- ❖ **流水段(pipe stage or pipe segment)**: 流水线的组成部件
 - ✧ 流水段首尾相连, 形成管道(pipe)
 - ✧ 指令从一端进入, 经过所有流水段, 从另一端离开
- ❖ **吞吐量(throughput)**: 单位时间内离开流水线的指令数目
- ❖ **处理器周期(processor cycle)**: 指令通过一个流水段的时间
 - ✧ 由流水线中最慢流水段的处理时间决定
 - ✧ 定义为 1 个时钟周期(clock cycle)

设计目标

❖ 设计目标：平衡每个流水段的“长度”（所耗费的时间）

❖ 理想情形

✧ 每条指令执行时间

$$Time_{instruction} = \frac{Time_{instruction\ on\ unpipelined\ machine}}{Number_{pipe\ stages}}$$

✧ 加速比

$$\begin{aligned} speedup &= \frac{Time_{instruction\ on\ unpipelined\ machine}}{Time_{instruction}} \\ &= Number_{pipe\ stage} \end{aligned}$$

❖ 流水线减少了每条指令的平均执行时间

✧ 如果执行每条指令需要多个时钟周期，则流水线减少了每条指令执行周期数 CPI。

❖ 指令流水线对程序员是“透明的”。

RISC架构的流水实现

❖ RISC架构的主要原则

- ❑ 所有数据操作均在寄存器中进行
- ❑ 影响存储器的操作只有 LOAD 和 STORE 指令
- ❑ 较少的指令格式（易于译码）
- ❑ 固定长度的指令编码

❖ 指令流水线在 RISC 架构上实现得更加简单（思考题）

❖ 课程将以 RISC V 架构做为典型案例

- ❑ 倪光南院士：RISC-V 架构未来将会和 X86 以及 ARM 在 CPU 领域“三分天下”。
- ❖ X86 主要用于通用电脑（美国籍）
- ❖ ARM 主要用于小型通用电子设备（英国籍，正在“移民”到美国籍）
- ❖ RISC-V 架构开源共享，且具备较强的灵活性（无国籍）

经典 RISC 五段流水线

❖ 构造指令五段流水线

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

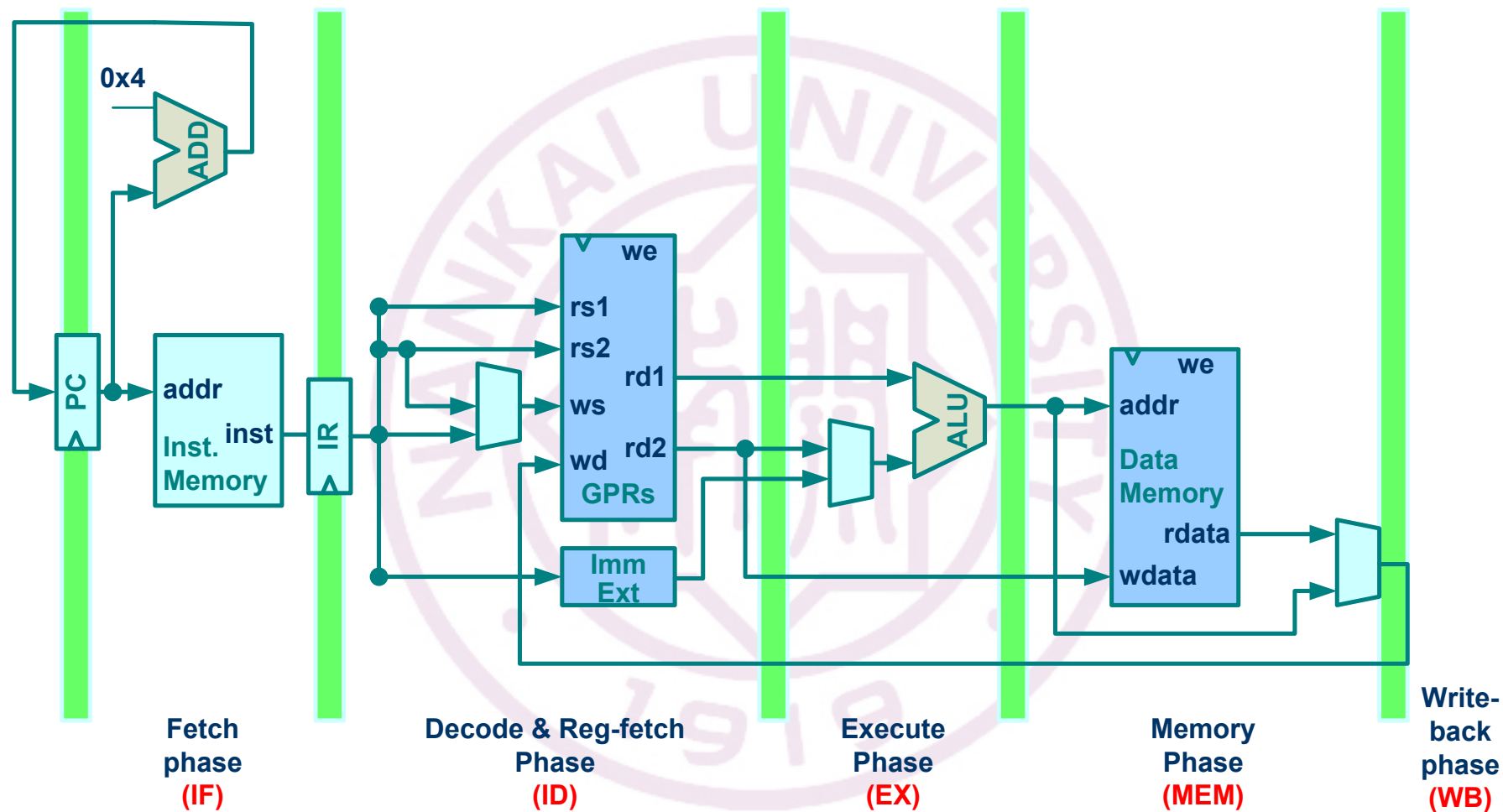
❖ 流水操作

$$t_{clock\ pipeline} > \max\{t_{IF}, t_{RF}, t_{ALU}, t_{DM}, t_{WB}\} = t_{DM}$$

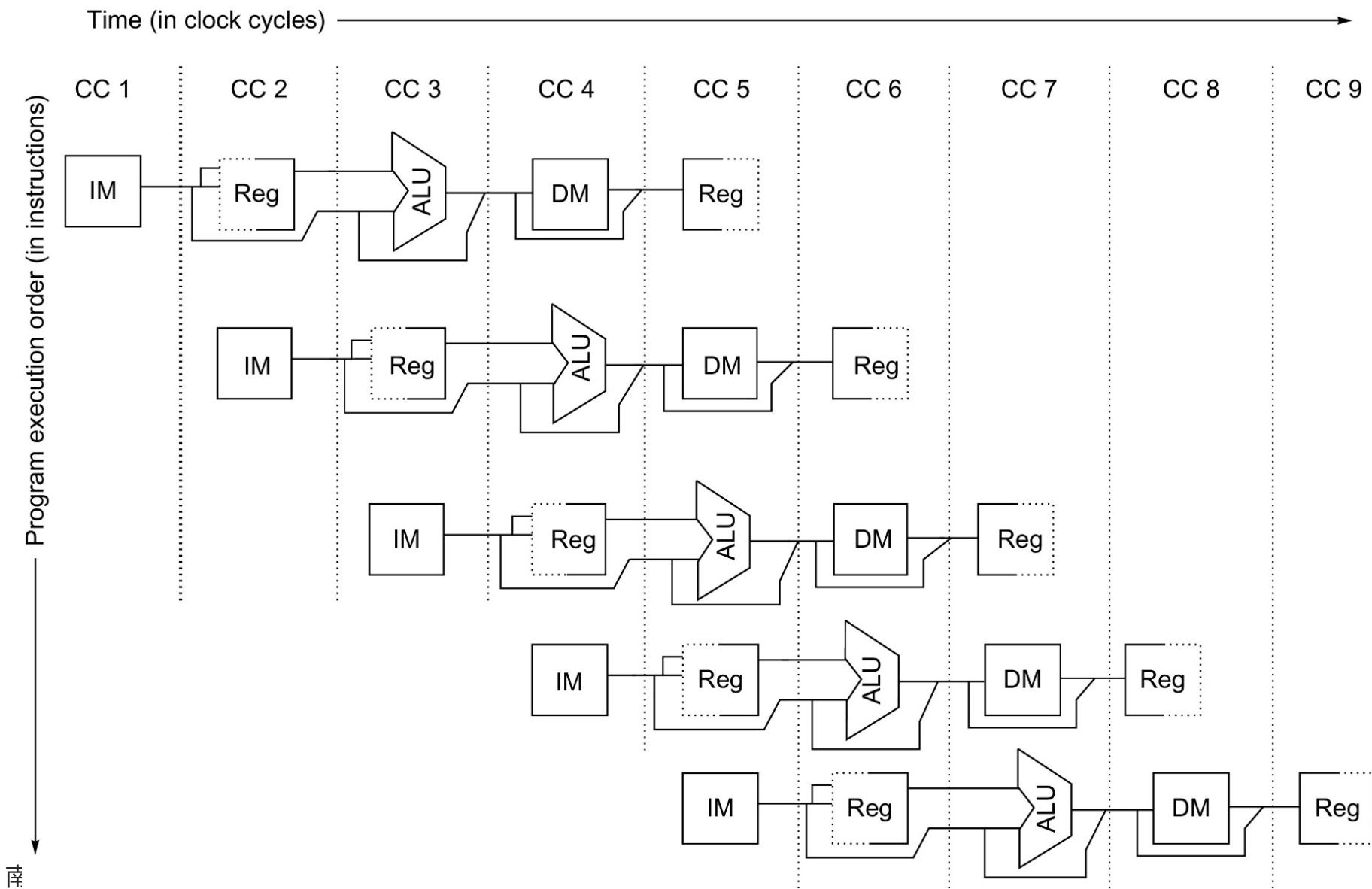
❖ 加速比（哈佛结构）

$$1 < speedup = \frac{t_{clock\ unpipelined}}{t_{clock\ pipelined}} < 5$$

经典 RISC 五段指令流水线



随时钟移动的数据通道序列

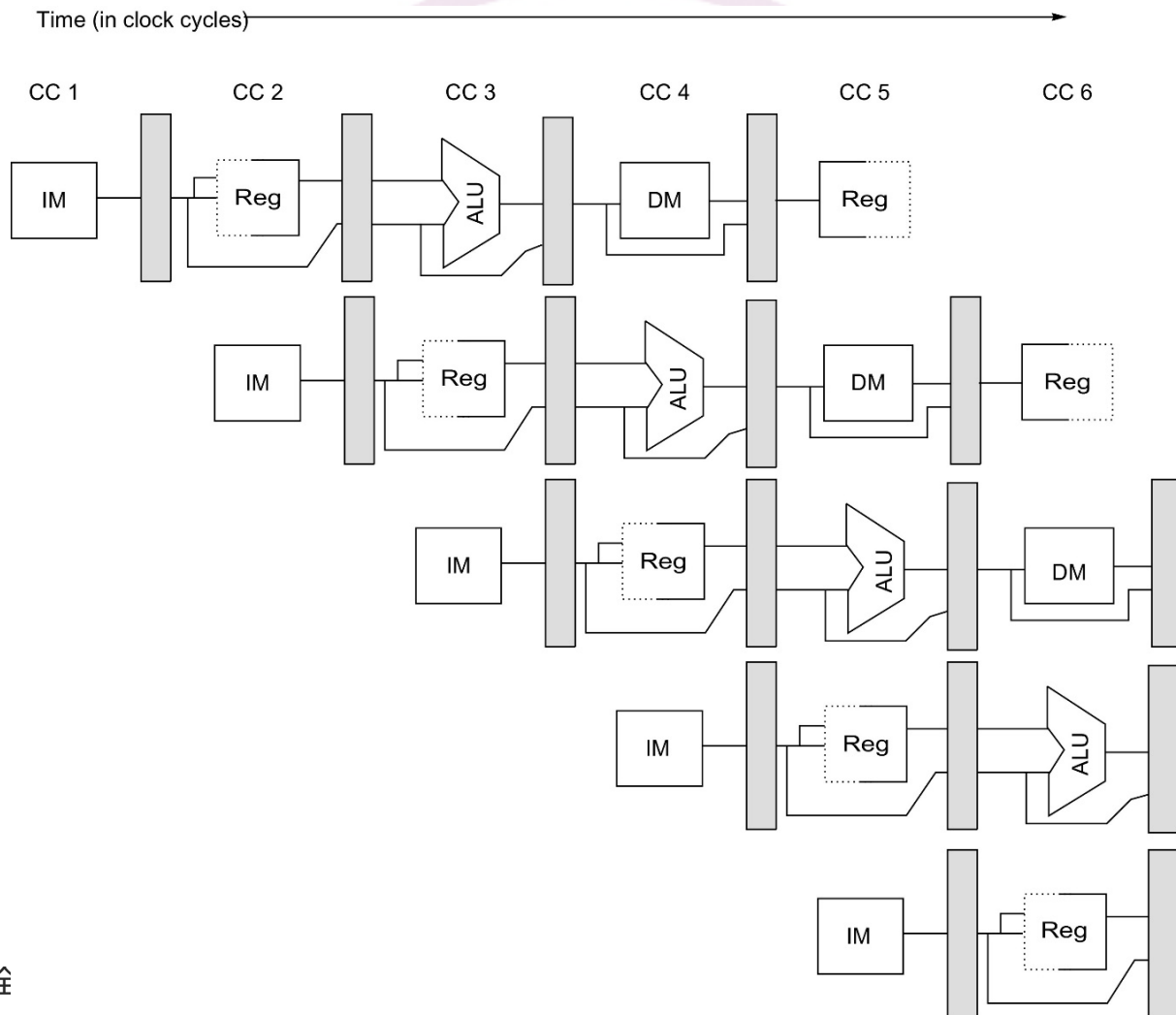


三点观察

- ❖ 将存储器分成指令存储器和数据存储器两个部分
 - ✧ 分别由指令缓存器和数据缓存器具体实现
 - ✧ 存储带宽必须是原来带宽的五倍——高性能的代价
- ❖ 两个流水段访问寄存器文件：ID 段读，WB 段写
 - ✧ 每个时钟周期完成两次读和一次写操作，故寄存器文件必须同时支持两路读操作和一路写操作
 - ✧ 为解决同一时钟周期读写同一寄存器的资源冲突问题，实现前半个周期写操作和后半个周期读操作（思考：交换顺序如何？）。
- ❖ 处理 PC
 - ✧ 在 IF 段，每个时钟周期完成 $PC \leftarrow PC + 4$
 - ✧ 在 ID 段，添加一个加法器计算可能的分支转移目标地址
 - ✧ 在 ALU 段，配置具有两个寄存器内容比较功能的 ALU

流水线寄存器

- ❖ 在时钟周期尾部存储流水段的所有结果，用于下个时钟周期下个流水段的输入。



流水线性能

- ❖ 流水线增加了处理器的指令吞吐量——单位时间内完成指令的数目
 - ❖ 流水线没有减少每条指令的执行时间，甚至稍微增加了其执行时间
 - ✧ 原因：流水线的控制开销
 - ✧ 流水线寄存器延时：建立时间（触发写操作之前输入信号必须稳定）
 - ✧ 时钟偏移(clock skew)：时钟到达任意两个寄存器之间的最大延时，也带来了时钟周期的下限。
 - ❖ 指令吞吐量的增加意味着
 - ✧ 程序运行得更快
 - ✧ 具有较低的总体运行时间
- 即使没有一条指令运行得更快！

流水线危害

- ❖ **流水线危害(hazards)**: 阻止指令流中下一条指令在其指定时钟周期内执行的情形。
 - ✧ 降低流水线的性能
- ❖ **流水线危害**将导致流水线操作 “**停顿(stall)**”
- ❖ 在某条指令操作被停止期间,
 - ✧ 该指令的后续指令也将停顿
 - ✧ 先于该指令发出的指令将继续执行 (否则, **危害**将永远不会消除)

流水线危害的类型

- ❖ **结构危害(structural hazards)**: 硬件资源在重叠执行中不能同时支持所有可能的指令组合, 即: **资源冲突**.
 - ✧ 使用频度较低的特殊目的功能单元, 如浮点数除法器 etc
 - ✧ 当程序员和编译程序意识到这些指令的低吞吐率时 (它们不是影响指令流水线性能的主要因素), 将不予关注
- ❖ **数据危害(data hazards)**: 一条指令依赖于在流水线中重叠执行的早期指令的运算结果
- ❖ **控制危害(control hazards)**: 分支转移指令和其它改变 PC 取值指令的流水操作

危害下的流水线性能

❖ 流水线危害导致流水线性能退化

$$speedup_{\text{pipelining}} = \frac{\bar{t}_{\text{unpipelined}}}{\bar{t}_{\text{pipelined}}} = \frac{CPI_{\text{unpipelined}} \times Cycle_{\text{unpipelined}}}{CPI_{\text{pipelined}} \times Cycle_{\text{pipelined}}}$$

❖ 令流水线理想 CPI 为 1，则流水线 CPI 为

$$\begin{aligned} CPI_{\text{pipelined}} &= CPI_{\text{ideal}} + \text{Pipeline stall cycles per instruction} \\ &= 1 + \text{Pipeline stall cycles per instruction} \end{aligned}$$

❖ 如果忽略周期时间开销，且假设流水段是完全平衡的，两颗处理器的周期时间相等，则

$$speedup = \frac{CPI_{\text{unpipelined}}}{1 + \text{Pipeline stall cycles per instruction}}$$

危害下的流水线性能

❖ 简化情形：所有指令执行需要相同的周期数

$$speedup = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

✧ 如果不存在流水线停止，则获得直观结果——可以通过流水线深度来改进流水线性能。

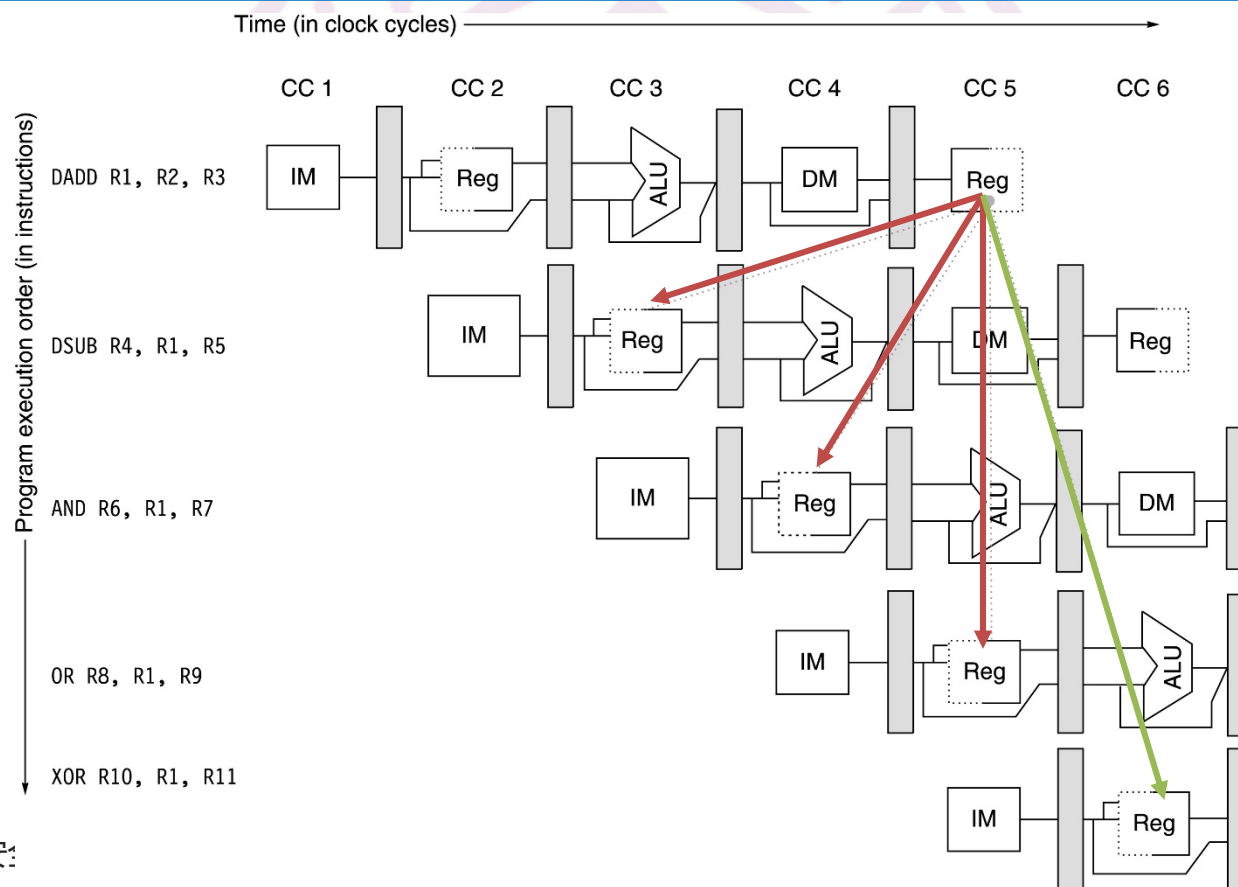
数据危害

- ❖ 指令重叠执行改变了指令间相对定时关系，是影响指令流水操作的主要原因
 - ✧ 指令重叠(overlap)执行导致了数据危害和控制危害
- ❖ 数据危害：操作数的实际读/写顺序与指令串行执行时的顺序不同
 - ✧ 写后读(Read After Write, RAW)危害：
 - ✧ 常见情形
 - ✧ 读后写(Write After Read, WAR)危害：
 - ✧ 简单五段指令流水线不可能发生
 - ✧ 可能在重排序(reordered)指令时发生
 - ✧ 写后写(Write After Write, WAW)危害：
 - ✧ 简单五段指令流水线不可能发生
 - ✧ 可能在重排序指令或执行时间变化时发生

RAW 危害

❖ 例:

```
DADD    R1, R2, R3
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```

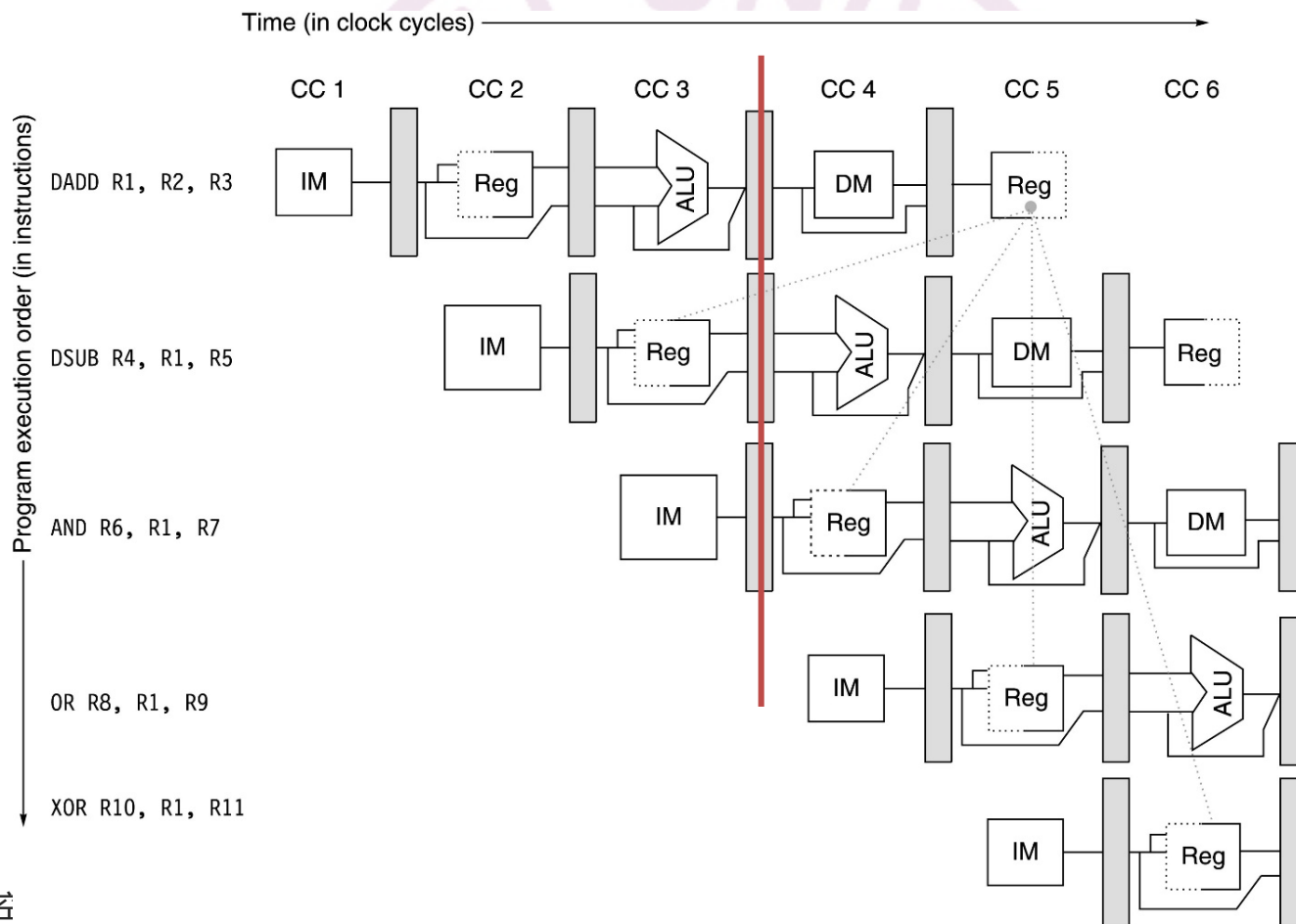


向前通道：最小化数据危害停顿

❖ 向前(forwarding)：旁路(bypassing)和短路(short-circuiting)

✧ 观察：在 DADD 实际产生结果之前，DSUB 并不真正需要该结果

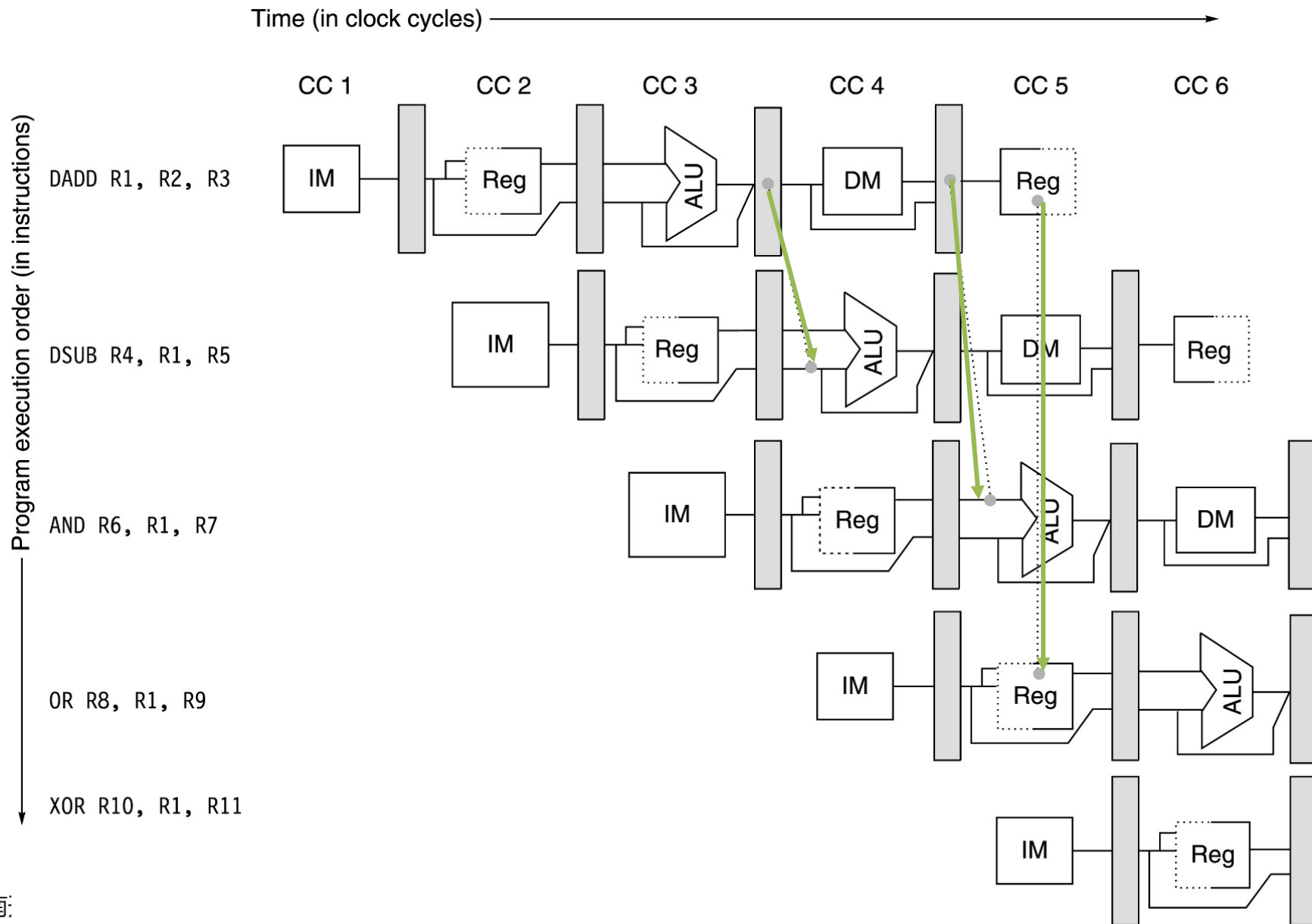
✧ 思路：将 DADD 结果从流水线寄存器移动到 DSUB 需要的地方



向前通道的工作过程

- ❖ 来自 EX/MEM 和 MEM/WB 流水线寄存器的 ALU 结果总是反馈到 ALU 输入端
- ❖ 如果向前通道硬件检测到早期 ALU 操作写入结果的寄存器对应当前 ALU 操作的源寄存器，则控制逻辑选择将结果传输给 ALU 输入端，而不是从寄存器文件中去读取数据。

使用向前通道避免数据危害

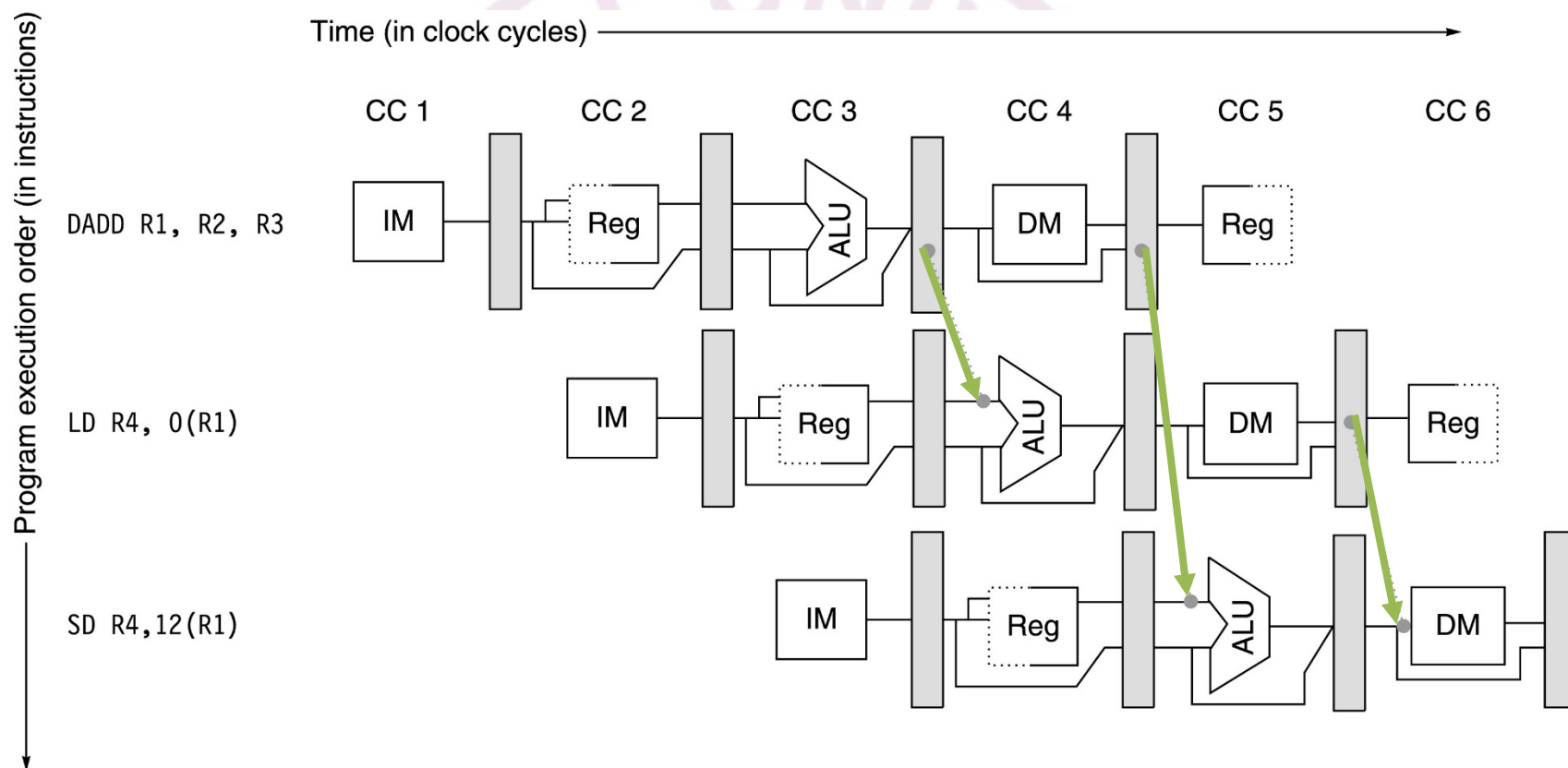


RAW 危害



例:

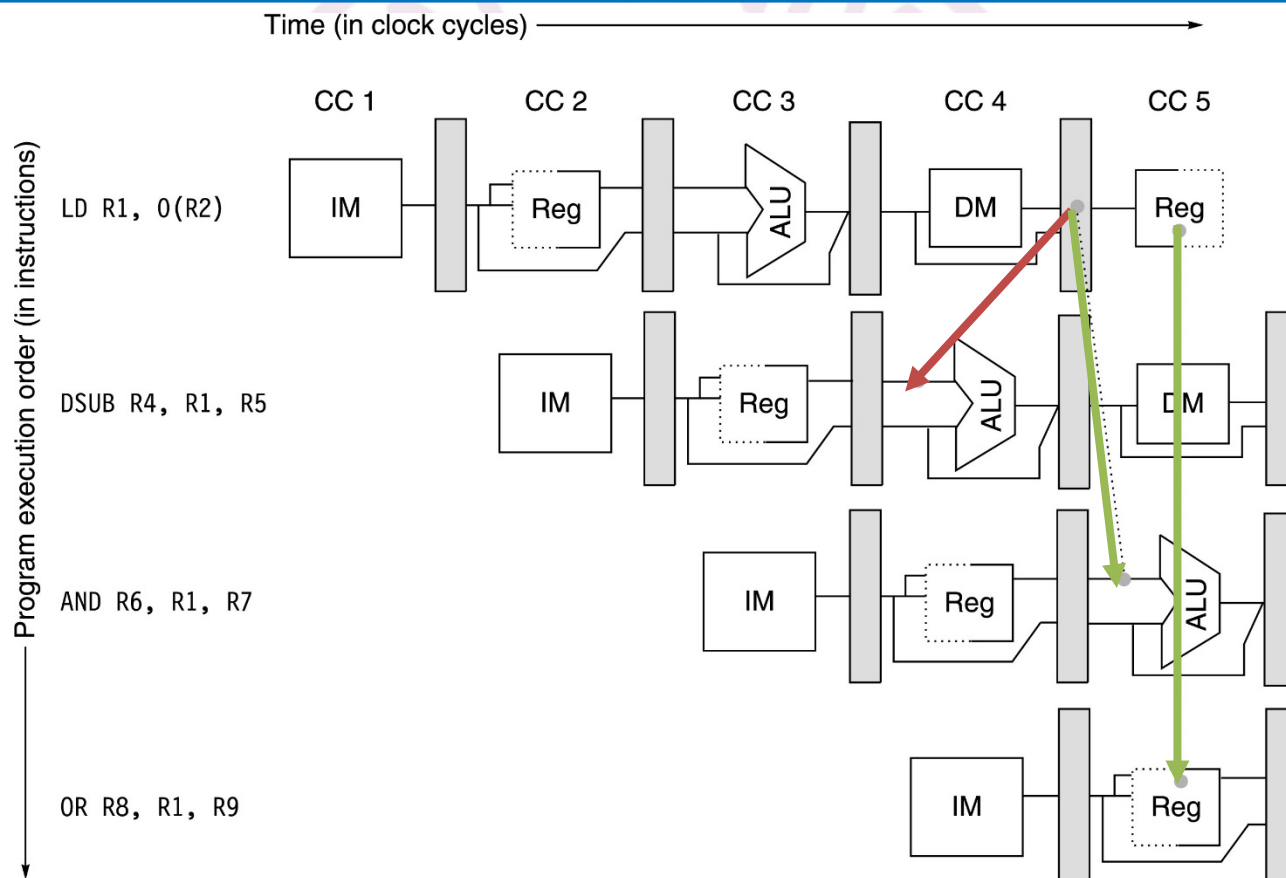
```
DADD R1, R2, R3  
LD R4, 0(R1)  
SD R4, 12(R1)
```



必须停顿的数据危害

❖ 例:

```
LD      R1, 0(R2)
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
```



流水线互锁(pipeline interlock)

- ❖ 用途：维持正确执行模式的硬件
- ❖ 功能：检测危害且停顿流水线操作直到危害被清除
 - ✧ 引入流水线停顿或气泡

❖ 流水线停顿或气泡

ld x1,0(x2)	IF	ID	EX	MEM	WB				
sub x4,x1,x5		IF	ID	EX	MEM	WB			
and x6,x1,x7			IF	ID	EX	MEM	WB		
or x8,x1,x9				IF	ID	EX	MEM	WB	
ld x1,0(x2)	IF	ID	EX	MEM	WB				
sub x4,x1,x5		IF	ID	Stall	EX	MEM	WB		
and x6,x1,x7			IF	Stall	ID	EX	MEM	WB	
or x8,x1,x9				Stall	IF	ID	EX	MEM	WB

- ✧ 停顿指令的 CPI 随着停顿长度增加而增加。

控制危害：分支转移危害


❖ 控制危害将比数据危害带来更大的性能损失

❖ 分支转移危害

- ❑ 分支转移指令，可能发生(taken)转移，也可能不发生(untaken)转移；
- ❑ 如果发生转移，那么 PC 将在 ID 段尾部（完成地址计算和比较）改变。

❖ 最简单的解决方法

- ❑ 重新取入分支转移指令后的指令或转移目标地址的指令

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

- ❑ 每条分支转移指令停顿 1 个周期，将导致大约10%~30%的性能损失。

减小流水线分支转移惩罚

❖ 冻结(freeze)流水线

- ❑ 保持或删除分支转移指令后的指令，直到获得目标地址为止。
- ❑ 对于硬件和软件均最简单的实现方法，分支惩罚是固定的。

❖ 预测不发生转移

- ❑ 如果发生转移，则将已取入指令变为空操作(no-op)指令，重新从目标地址取入指令。

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB
Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

减小流水线分支转移惩罚

❖ 预测发生转移

- ✧ 只要译码为分支指令且获得目标地址，则假设发生转移而取指和执行
- ✧ 如果预测正确，则获得一个周期奖励。因为在 ID 段获得目标地址，比在 ALU 段获得条件结果“提前”一个时钟周期。

❖ 无论预测转移或不转移，编译程序均可以通过组织代码使最频繁路径与硬件选择相匹配来提高性能。

❖ 延迟分支转移(delayed branch)

```
branch instruction  
sequential successor1  
branch target if taken
```

- ✧ sequential successor₁ 位置称为分支延迟槽(branch delay slot)
 - ✧ 无论分支指令是否发生转移，处于分支延迟槽中的指令一定执行
 - ✧ 编译程序的工作是使得后继指令有效和有用

减小流水线分支转移惩罚

- ✧ 当动态分支预测技术出现之后，由于该方法实现过于复杂，在 RISC V 中被淘汰。
- ✧ 时序示意图

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB
Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target+1				IF	ID	EX	MEM	WB	
Branch target+2					IF	ID	EX	MEM	WB

✧ 分支预测机制的性能

$$speedup_{\text{pipeline}} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

例：不同分支预测机制对 CPI 的影响

- ❖ 对于更深的指令流水线，假设在条件比较中没有停顿，在已知分支目标地址之前至少需要三个流水段，且在计算分支条件之前需要一个额外的时钟周期。三个流水段的延迟使得三种最简单预测机制的分支惩罚为

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

假设：各种分支指令的频度如下，

Unconditional branch	4%
Conditional branch, untaken	6%
Conditional branch, taken	10%

求分支指令对 CPI 的影响。

例：不同分支预测机制对 CPI 的影响

Branch scheme	Additions to the CPI from branch costs			
	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

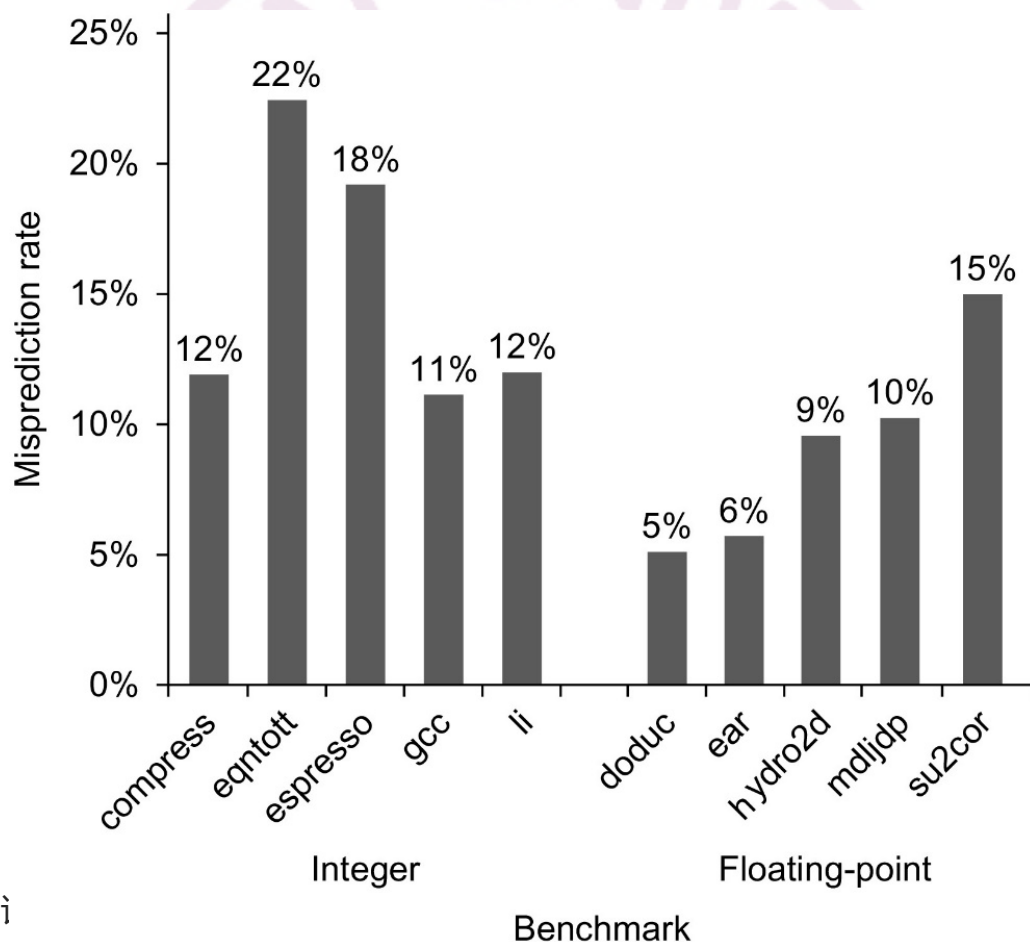
- ❖ 假设：基本 $CPI = 1$ ，且分支指令是停顿的唯一原因，则
- ✧ 理想流水线比使用停顿流水机制的流水线快 $1 + 0.56 = 1.56$ 倍
 - ✧ 预测不转移机制比停顿流水机制好 $\frac{1+0.56}{1+0.38} \approx 1.13$ 倍

通过预测机制降低分支代价

- ❖ 随着流水线越来越深，分支潜在的惩罚也在增加。
 - ✧ 延迟分支和类似机制变得不能胜任
- ❖ 需要更积极的方法来预测分支指令的行为
 - ✧ **静态预测**：依赖于编译时可用信息的低成本静态方法
 - ✧ **动态预测**：基于程序行为动态预测分支的策略
- ❖ 分支预测方案的有效性取决于
 - ✧ 方案的准确性
 - ✧ 条件分支出现的频度

静态分支转移预测

- ❖ **关键**：利用早期运行时收集的信息
- ❖ **观察**：分支指令的转移行为通常呈双峰分布，即：某条分支指令高度偏向于“转移”或“不转移”。



分支预测缓存或分支历史表

❖ 原理

- ❑ 以分支指令低位地址做为索引的小型存储器
- ❑ 使用一个比特表示分支指令最近一次行为（转移或不转移）

❖ 特点

- ❑ 无标签，可能利用的是其它相同低位地址分支指令的历史信息
- ❑ 只有当分支延时时间超过目标地址计算时间时，该方法才有效

❖ 预测性能：取决于感兴趣分支指令的预测频度和匹配时预测的准确性

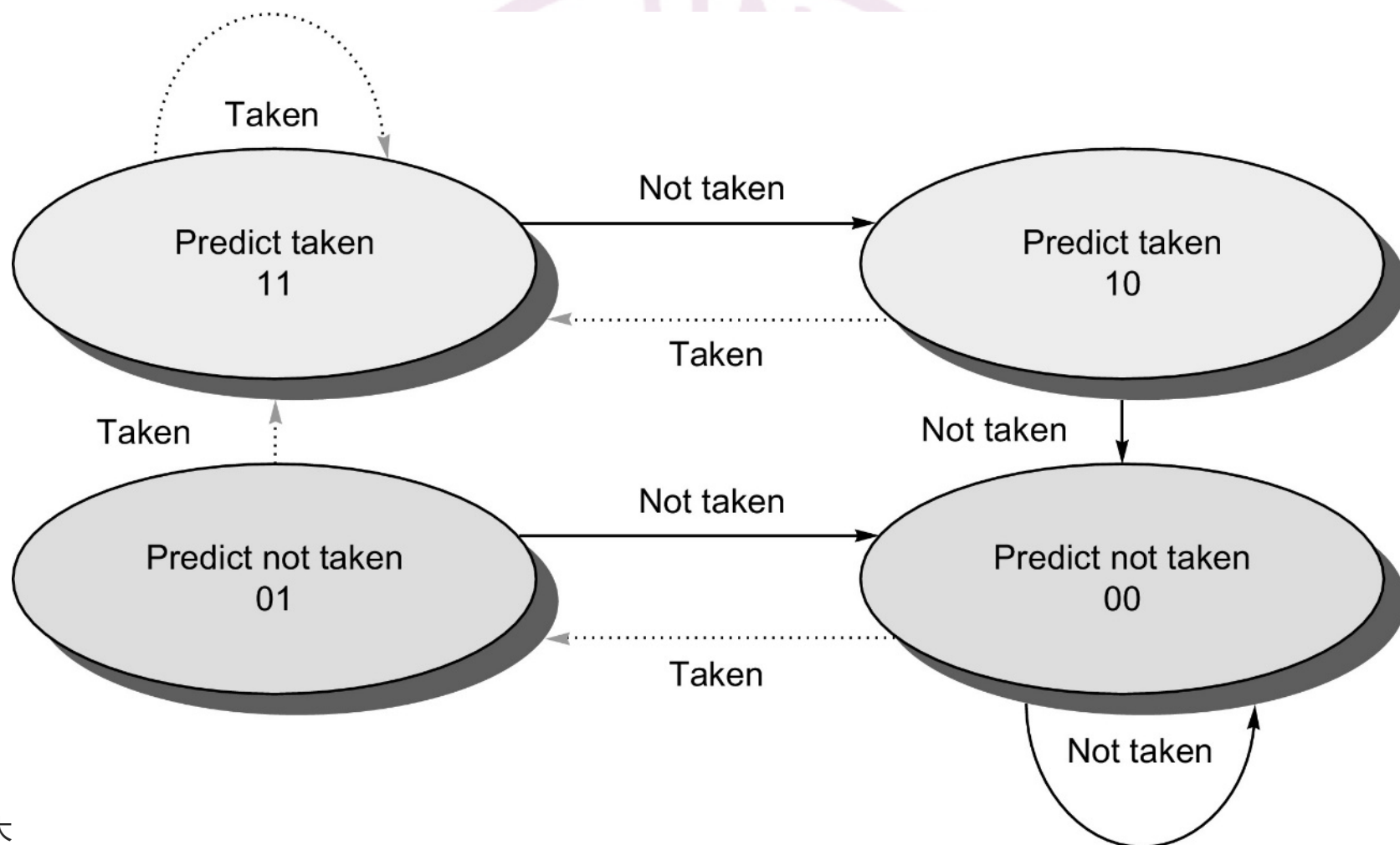
❖ 单比特预测方法的性能缺陷

- ❑ 即使一条分支指令几乎总是跳转，方法仍然可能出现两次预测错误，而不是一次。

分支预测缓存或分支历史表

❖ 2-比特预测方法

✧ 预测在改变之前必须经历两次错误



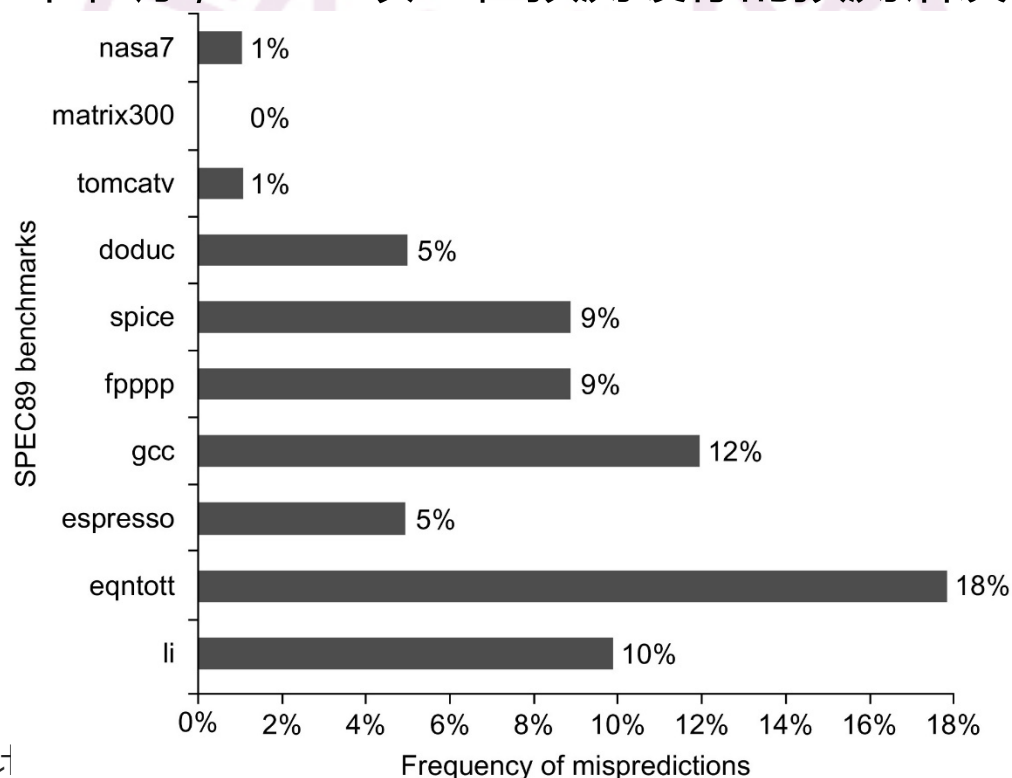
分支预测缓存或分支历史表

❖ 实现

- ❑ 在 IF 段期间，使用指令地址访问的小型、特定 Cache；或
- ❑ 附加在指令缓存器每个数据块上的两个比特，与指令一起获取。

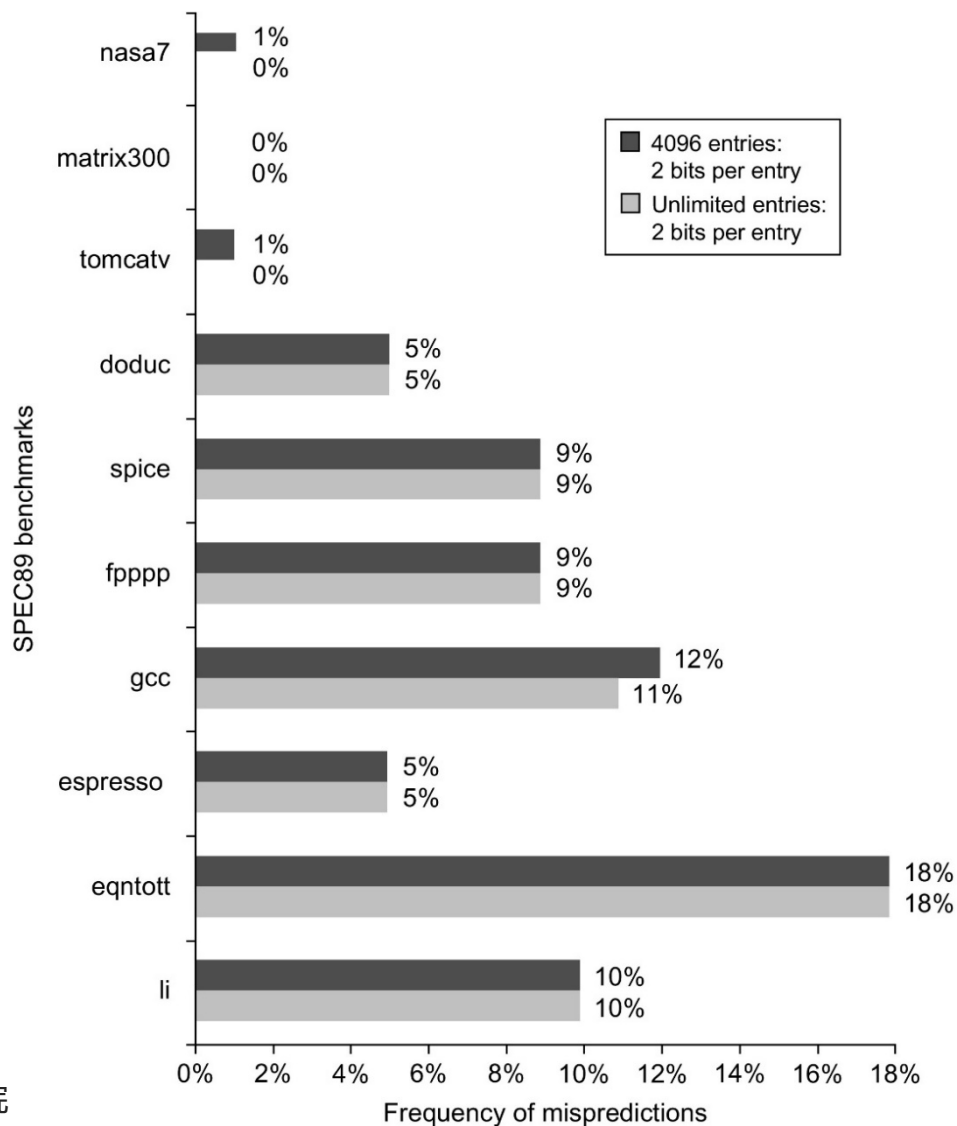
❖ 性能

- ❑ SPEC89基准程序，4096项 2 位预测缓存的预测错误率



分支预测缓存或分支历史表

❖ 简单增加缓存项容量或比特数不会对性能带来明显改进。



诚信 创新 实践

