第十一章

指令流水线

合肥工业大学
系统结构研究所
陈 田

一、如何提高机器速度

1. 提高访存速度

   高速芯片　　Cache　　多体并行

2. 提高 I/O 和主机之间的传送速度

   中断　　DMA　　通道　　I/O 处理机　　多总线

3. 提高运算器速度

   高速芯片　　改进算法　　快速进位链

- 提高整机处理能力

  高速器件　　改进系统结构，开发系统的并行性

二、系统的并行性

1. 并行的概念

并行 { 并发：两个或两个以上事件在 同一时间段 发生

同时：两个或两个以上事件在 同一时刻 发生

时间上互相重叠

2. 并行性的等级

过程级（程序、进程）	粗粒度	软件实现
指令级（指令之间） （指令内部）	细粒度	硬件实现

三、指令流水原理

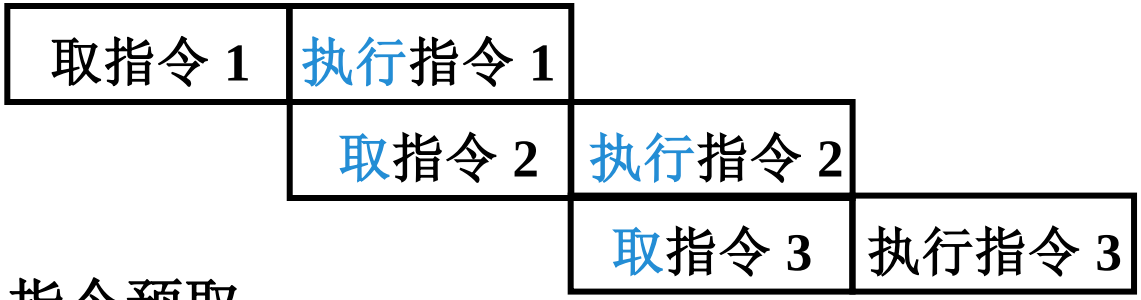
1. 指令的串行执行

取指令 1	执行指令 1	取指令 2	执行指令 2	取指令 3	执行指令 3	…

取指令　　取指令部件　　完成　　总有一个部件 空闲

执行指令　　执行指令部件　　完成

2. 指令的二级流水



指令预取

若 取指 和 执行 阶段时间上 完全重叠

指令周期 减半　速度提高 1 倍

3. 影响指令流水效率加倍的因素

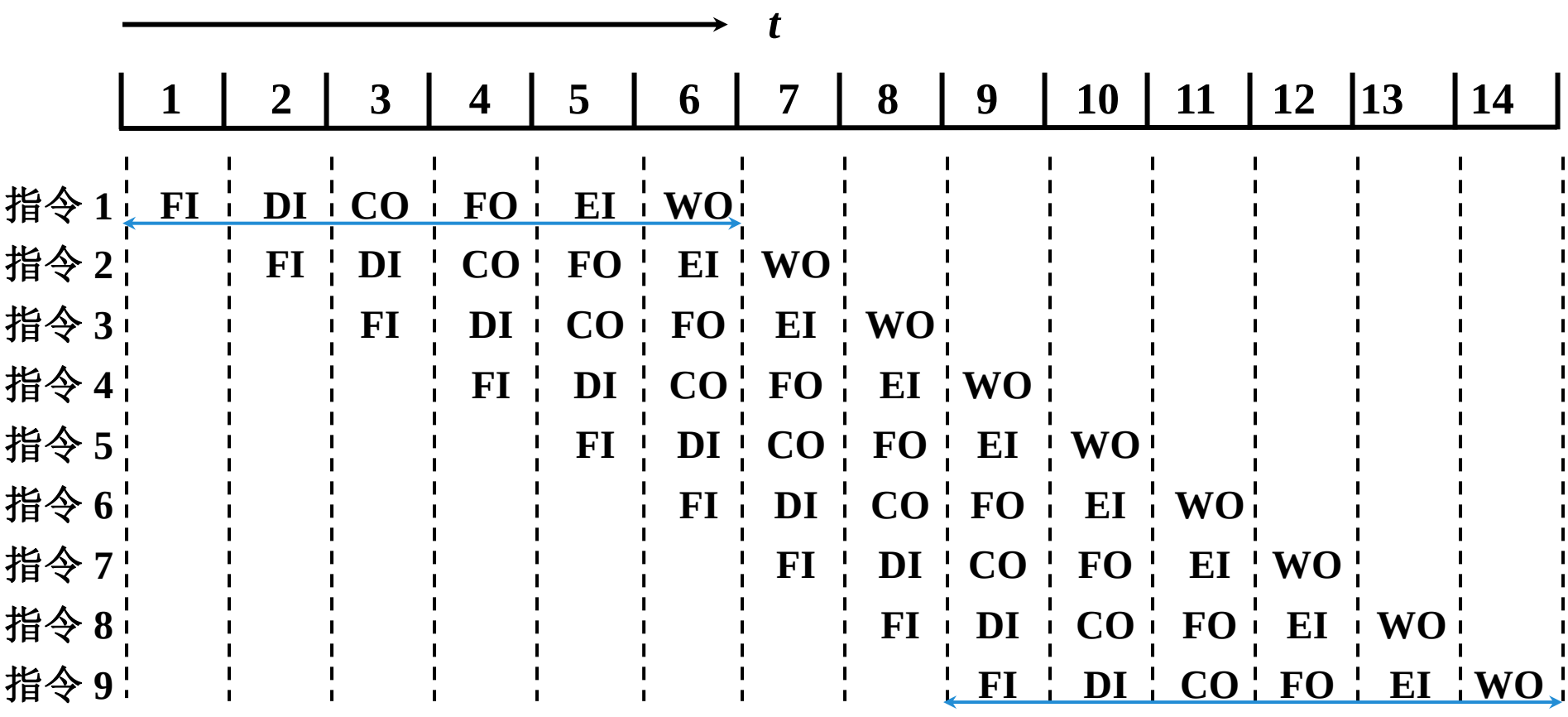
(1) 执行时间 > 取指时间

(2) 条件转移指令 对指令流水的影响

必须等 上条 指令执行结束，才能确定 下条 指令的地址，

造成时间损失　　猜测法

4. 指令的六级流水



完成一条指令 6 个时间单位

串行执行 6 × 9 = 54 个时间单位

六级流水 14 个时间单位

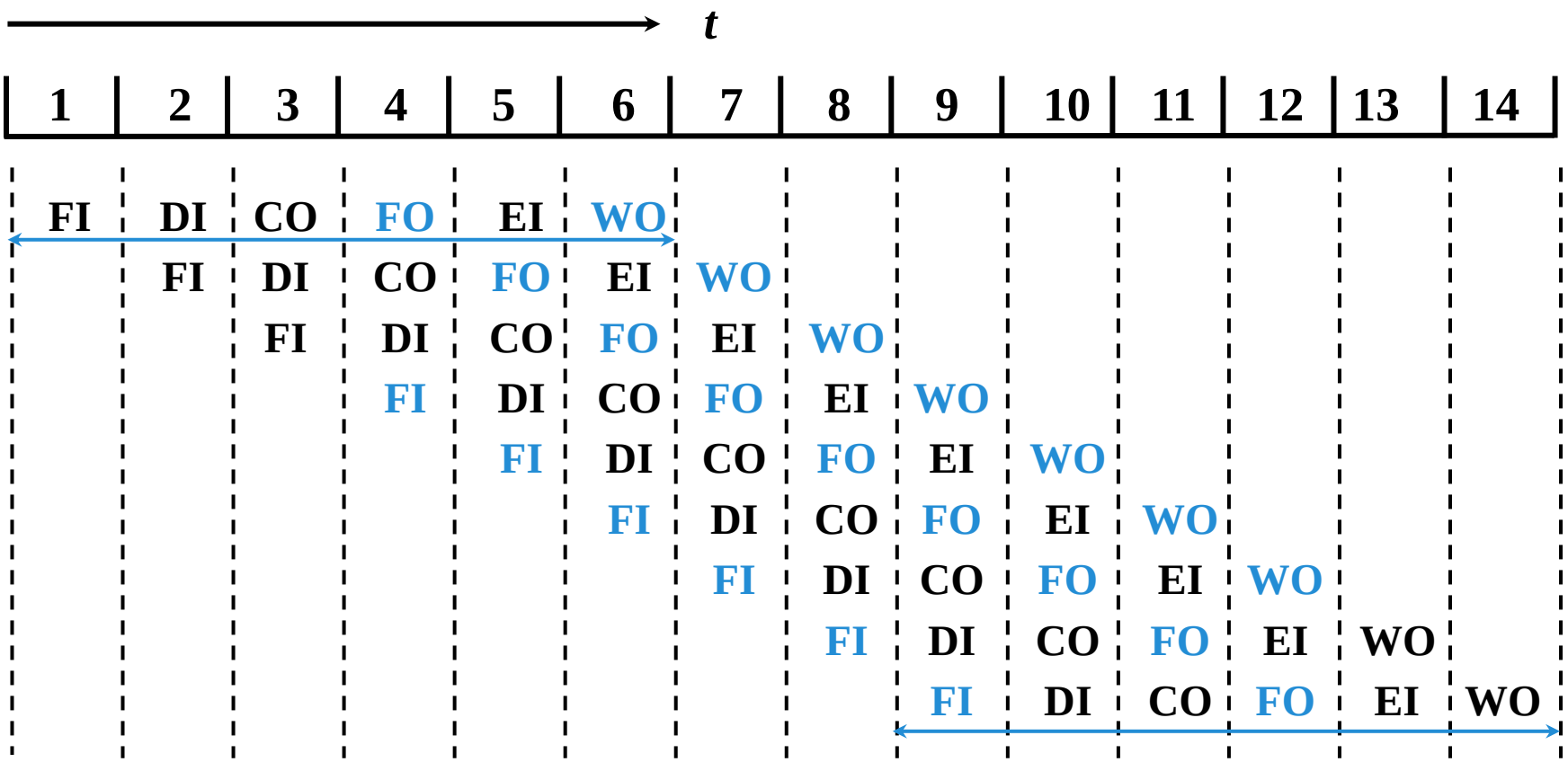
三、影响指令流水线性能的因素

程序的相近指令之间出现某种关联

使指令流水出现停顿，影响流水线效率

三、影响指令流水线性能的因素

1. 结构相关（冒险）不同指令争用同一功能部件产生资源冲突



解决办法

指令 1 与指令 4 冲突　　指令1、指令3、指令 6 冲突

指令 2 与指令 5 冲突　　…

- 停顿
- 指令存储器和数据存储器分开
- 指令预取技术 （适用于访存周期短的情况）

2. 数据相关（冒险）

不同指令因重叠操作，可能改变操作数的 读/写 访问顺序

- 写后读相关（RAW）：指令j在指令i写入寄存器前就读此寄存器

  SUB R_1，R_2，R_3 ；（R_2）－（R_3）⟶ R_1

  ADD R_4，R_5，R_1 ；（R_5）+（R_1）⟶ R_4

- 读后写相关（WAR）：指令j在指令i读寄存器前就写入此寄存器

  MUL R_1，R_2 ；(R_1)×（R_2）⟶(R1)

  MOV R_2，0 ；0⟶R_2

- 写后写相关（WAW）：指令j在指令i写入寄存器前就写入此寄存器

  MUL R_3，R_2，R_1 ；（R_2）×（R_1）⟶ R_3

  SUB R_3，R_4，R_5 ；（R_4）－（R_5）⟶ R_3

解决办法 • 后推法 • 采用 旁路技术

3. 控制相关（冒险）

由转移指令引起

```
      LDA    # 0
      LDX    # 0
M     ADD    X, D
      INX
      CPX    # N
      BNE    M
      DIV    # N
      STA    ANS
```

BNE 指令必须等

CPX 指令的结果

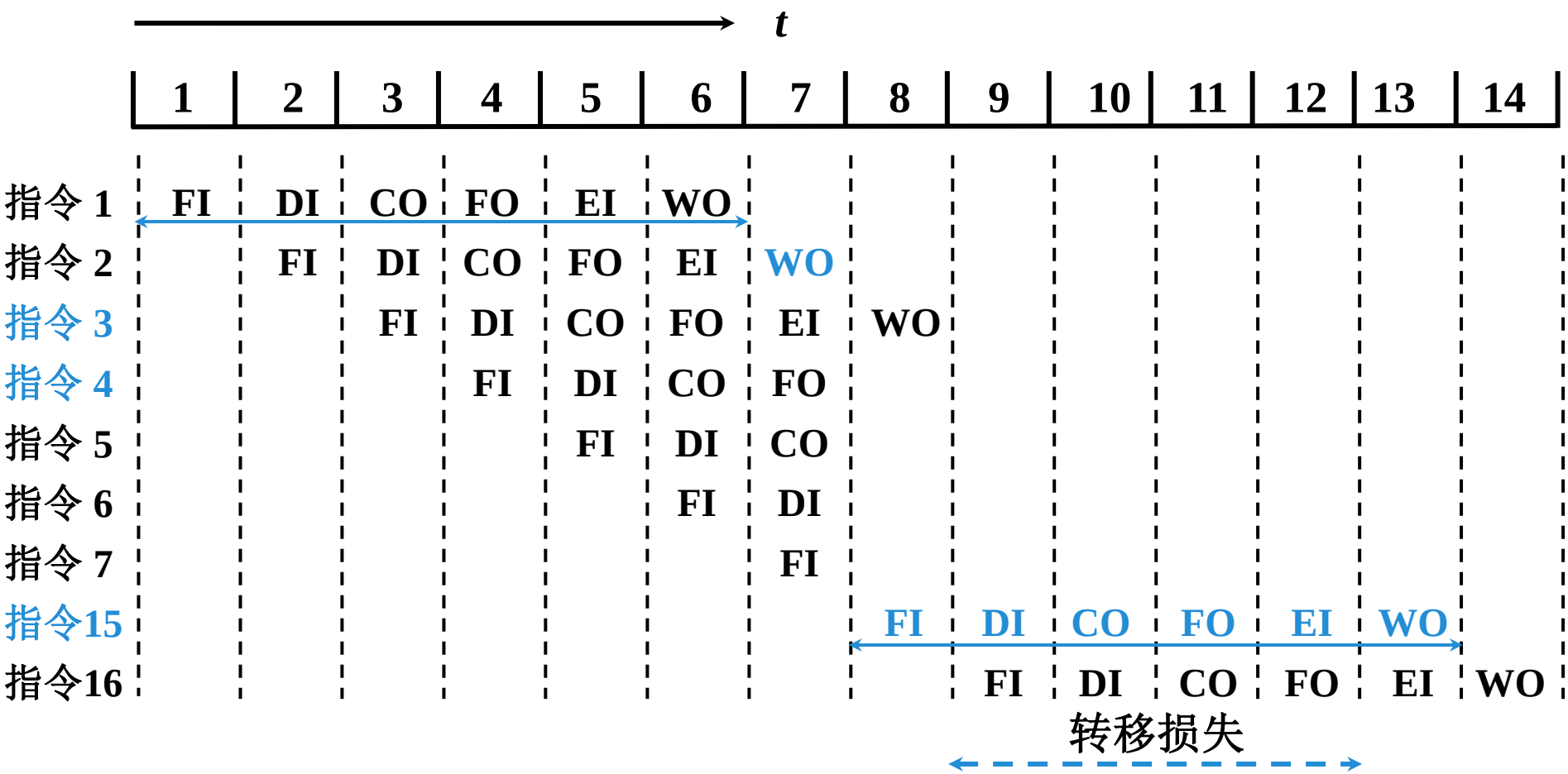
才能判断出

是转移

还是顺序执行

3. 控制相关

设 指令3 是转移指令



四、流水线性能

1. 吞吐率

单位时间内 流水线所完成指令 或 输出结果 的 数量

设 m 段的流水线各段时间为Δt

- 最大吞吐率:流水线在连续流动达到稳定状态后得到的吞吐率

$$T_{pmax} = \frac{1}{\Delta t}$$

- 实际吞吐率：流水线在运行有限个任务时实际能够达到的吞吐率

连续处理 n 条指令的吞吐率为

$$T_p = \frac{n}{m \cdot \Delta t + (n-1) \cdot \Delta t}$$

2. 加速比 S_p

m 段的 流水线的速度 与等功能的 非流水线的速度 之比

设流水线各段时间为 Δt

完成 n 条指令在 m 段流水线上共需

$$T = m \cdot \Delta t + (n-1) \cdot \Delta t$$

完成 n 条指令在等效的非流水线上共需

$$T' = nm \cdot \Delta t$$

则

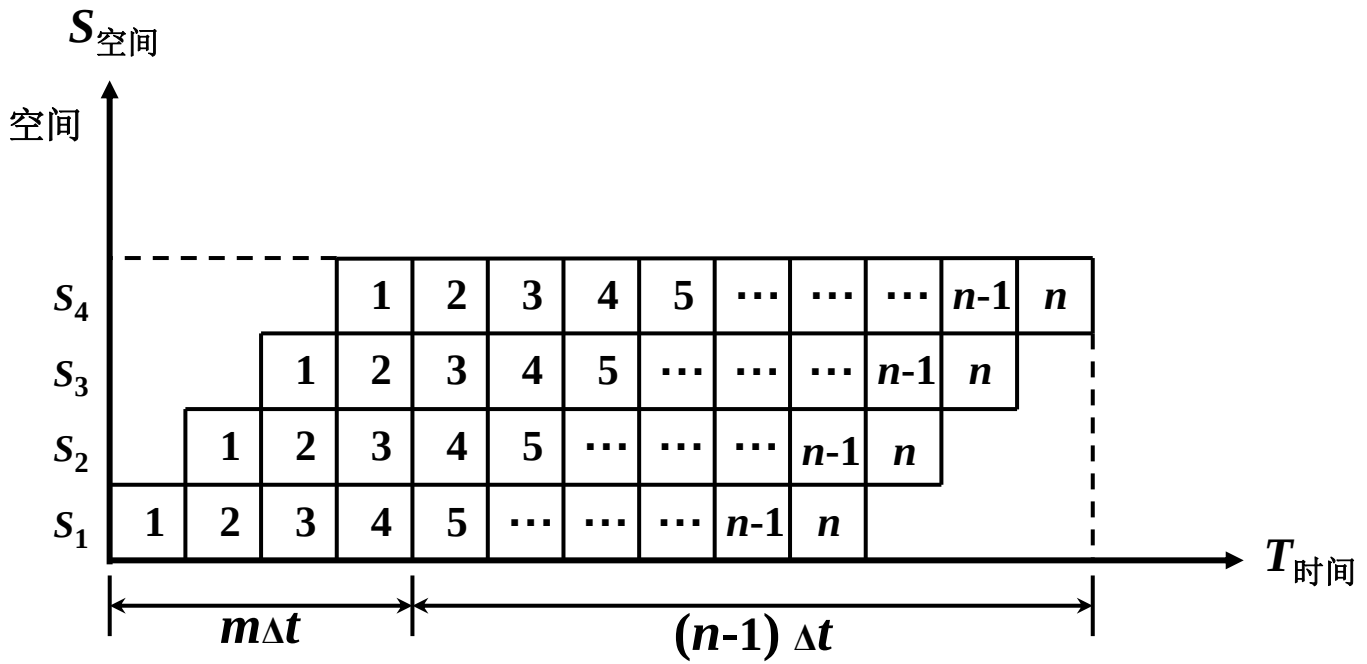
$$S_p = \frac{nm \cdot \Delta t}{m \cdot \Delta t + (n-1) \cdot \Delta t} = \frac{nm}{m + n - 1}$$

3. 效率

流水线中各功能段的 利用率

由于流水线有 建立时间 和 排空时间

因此各功能段的 设备不可能 一直 处于 工作 状态

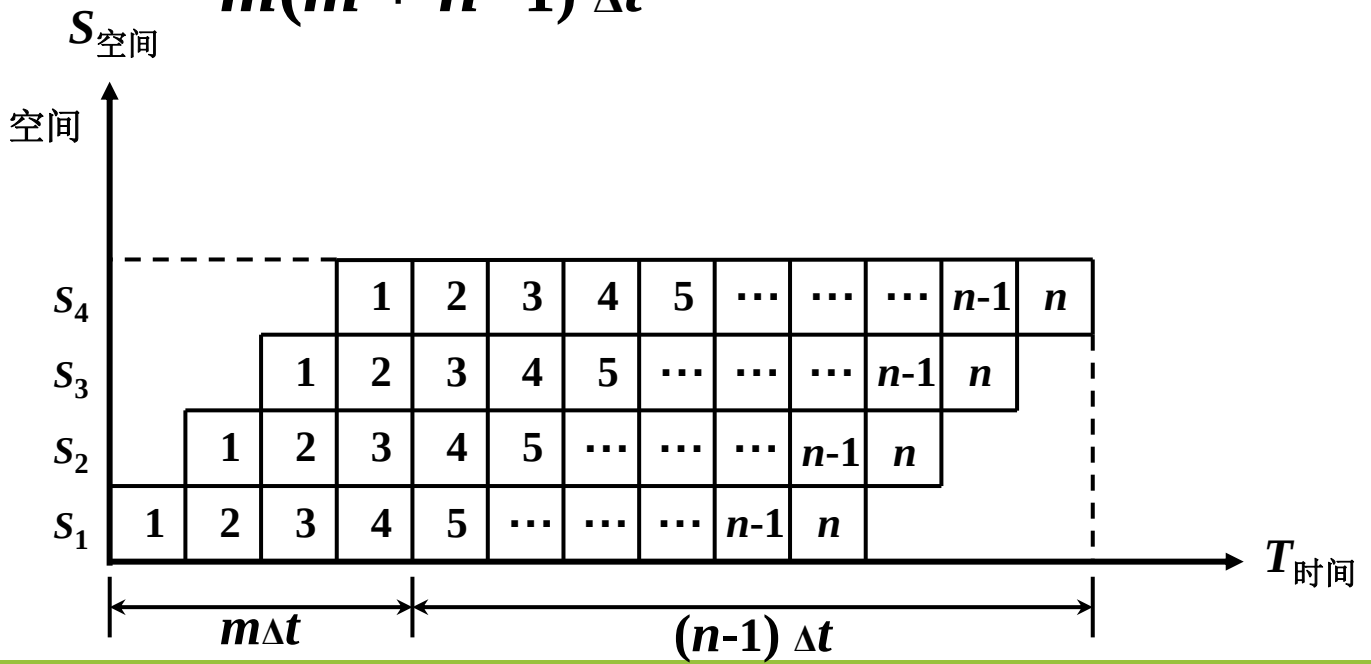


3. 效率

流水线中各功能段的 利用率

$$\text{效率} = \frac{\text{流水线各段处于工作时间的时空区}}{\text{流水线中各段总的时空区}}$$

$$= \frac{mn\Delta t}{m(m+n-1)\,\Delta t}$$



时空图

- 流水线中各功能部件的重叠工作情况可以用时空图表示。
- 时空图的两种形式：
  - 一种以流水段（流水部件）为纵坐标，以时间段为横坐标，用来表示每个部件在每个时间段的工作情况。
  - 另一种以指令序列为纵坐标，以时间段为横坐标，纵坐标的方向是从上到下的（指令序列通常从上到下表示）

MIPS指令概述

- MIPS (Microprocessor without Intellocked Pipleline Stages)是80年代初期由斯坦福大学Hennessy教授领导的研究小组研制成功；

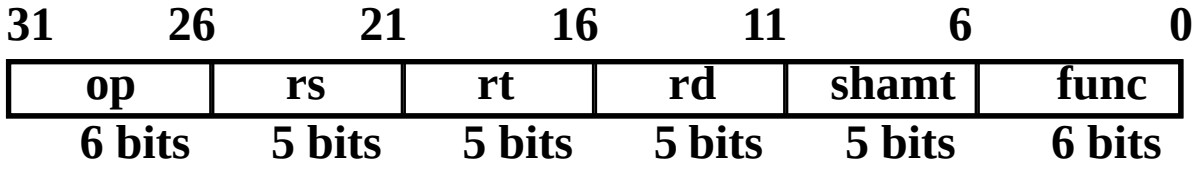
- 属于RISC(Reduced Instruction Set Computer)；

- MIPS指令集有MIPS I，MIPS II，MIPS III，MIPS IV，MIPS V，MIPS32，和MIPS64多个版本；

MIPS32指令格式

- 所有指令都是32位宽，须按字地址对齐

R-Type指令



❖ 有三种指令格式

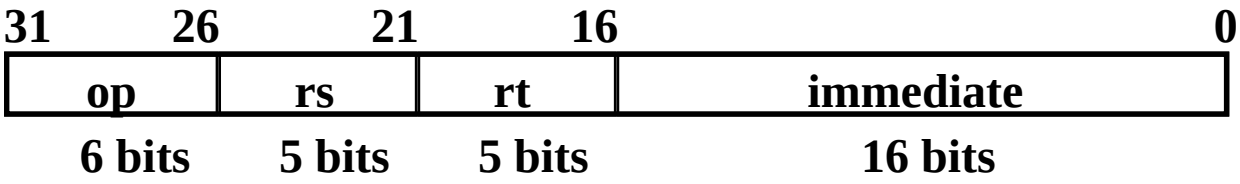
- R-Type

  两个操作数和结果都在**寄存器**的**运算**指令。如：sub rd, rs, rt

- ♦ Rs,Rt分别为第1、2源操作数；Rd为目标操作数

- I-Type
  - **运算指令**：一个寄存器、一个立即数。如：ori rt, rs, imm16
  - LOAD和STORE指令。如：lw rt, rs, imm16
  - 双目、Load/Store: Rs和立即数是源操作数，rt为目标操作数
  - 条件分支指令，rs,rt均为源操作数。如：beq rs, rt, imm16

I-Type指令

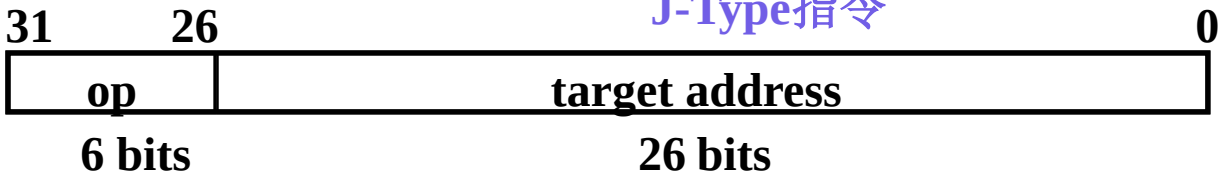


- J-Type

  无条件跳转指令，26位立即数作为跳转目标地址的部分地址

  如：j target

J-Type指令



MIPS的寄存器

寄存器名	寄存器编号	用途说明
$s0	0	保存固定的常数0
$at	1	汇编器的临时变量
$v0 ~ $v1	2 ~ 3	子函数调用返回结果
$a0 ~ $a3	4 ~ 7	函数调用参数1 ~ 3
$t0 ~ $t7	8 ~ 15	临时变量，函数调用时不需要保存和恢复
$s0 ~ $s7	16 ~ 23	函数调用时需要保存和恢复的寄存器变量
$t8 ~ $t9	24 ~ 25	临时变量，函数调用时不需要保存和恢复
$k0 ~ $k1	26 ~ 27	中断、异常处理程序使用
$gp	28	全局指针变量(Global Pointer)
$sp	29	堆栈指针变量(Stack Pointer)
$fp	30	帧指针变量(Frame Pointer)
$ra	31	返回地址(Return Address)

♦还有32个32位单精度浮点寄存器f_0-f_{31}

♦还有2个32位乘、商寄存器H_i和L_0；乘法法分别存放64位乘积的高、低 32位；除法时分别存放余数和商。

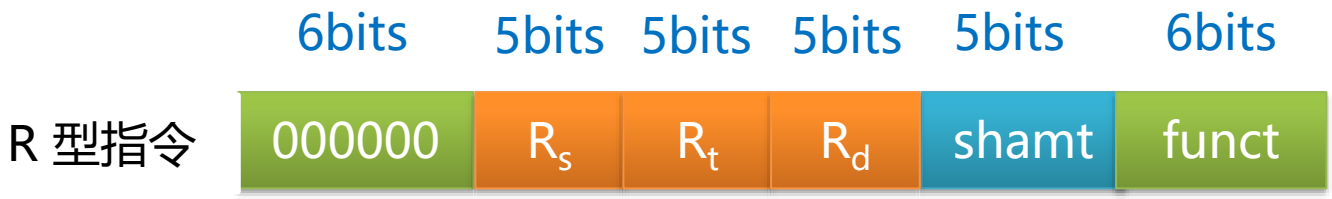
MIPS arithmetic and logic instructions

Instruction	汇编举例	含义	备注
add	add $1,$2,$3	$1 = $2 + $3	3个寄存器操作数
subtract	sub $1,$2,$3	$1 = $2 – $3	3个寄存器操作数
load word	lw $s1,100 ($S2)	$s1 =Memory[$s2+100]	从内存取一个字到寄存器
store word	sw $s1,100 ($ S2)	Memory[$s2+100] = $s1	从内存取一个字到寄存器
multiply	mult $2,$3	Hi, Lo = $2×$3	64-bit signed product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
branch on equal	beq $s1, $s2, L	if ($s1==$ s2) go to L	相等则转移
branch on not equal	bne $s1, $s2, L	if ($s1！=$ s2) go to L	不相等则转移

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	Logical AND
or	or $1,$2,$3	$1 = $2 | $3	Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	Logical NOR

MIPS寻址方式

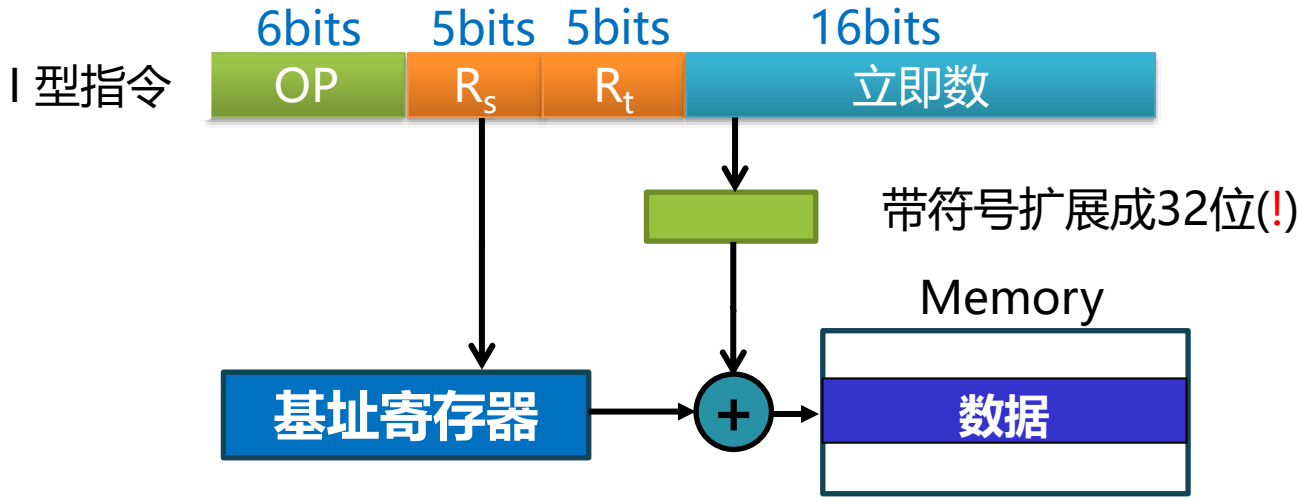
- 寄存器直接寻址(Register Addressing)



add $t0, $s1, $s2 ($t0=$s1+$s2)

MIPS寻址方式

- 基址寻址(Basic Addressing)



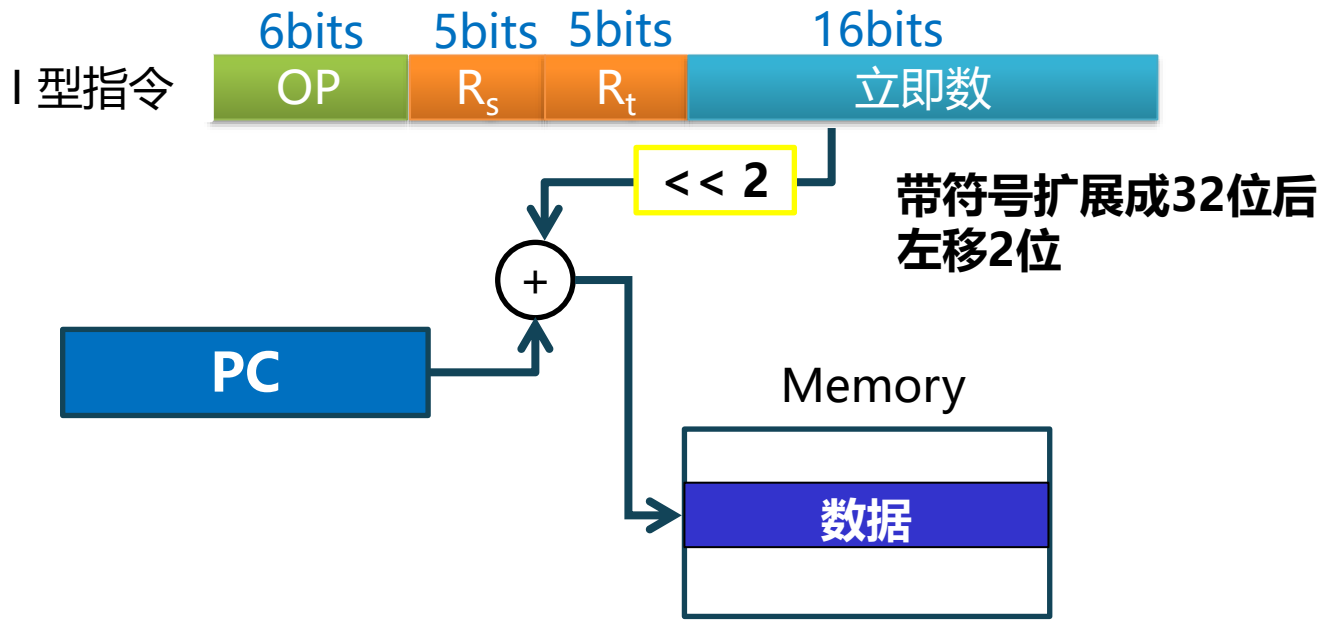
- 使用基址寻址的指令：lw ,sw, lh, sh, lb, lbu等

LB rt , offset (base)

MIPS寻址方式

- 相对寻址

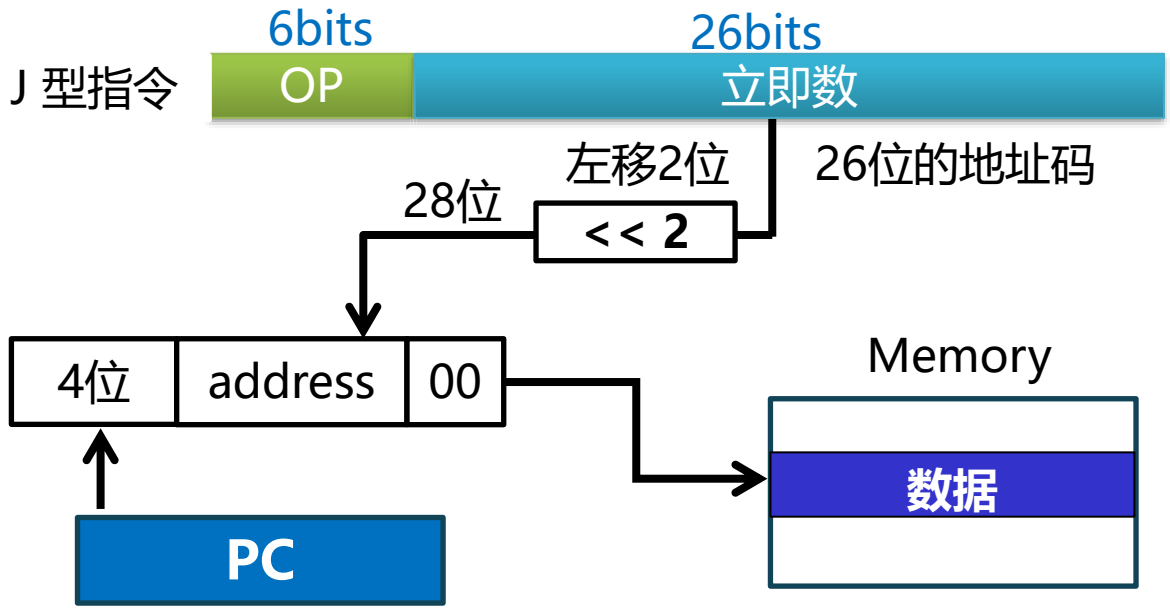
```
if (GRP[rs] == GPR[rt])
PC = PC + 4 + BranchAddr
```



- 使用相对寻址的指令：beq, bne

MIPS寻址方式

- 伪直接寻址(页面寻址)



- 使用伪直接寻址的指令：j, jal

MIPS指令详解

1 R型指令

操作数和保存结果均通过寄存器进行；

- op：操作码，所有R型指令中都全为0；
- rs：寄存器编号，对应第1个源操作数；
- rt：寄存器编号，对应第2个源操作数；
- rd：寄存器编号，据此保存结果；
- shamt：常数，在移位指令中使用；
- funct：功能码，指定指令的具体功能；

MIPS指令详解

1 R型指令

指令	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	指令功能
add	000000	rs	rt	rd	00000	100000	寄存器加
sub	000000	rs	rt	rd	00000	100010	寄存器减
and	000000	rs	rt	rd	00000	100100	寄存器与
or	000000	rs	rt	rd	00000	100101	寄存器或
xor	000000	rs	rt	rd	00000	100110	寄存器异或
sll	000000	00000	rt	rd	sa	000000	逻辑左移
srl	000000	00000	rt	rd	sa	000010	逻辑右移
sra	000000	00000	rt	rd	sa	000011	算术右移
jr	000000	rs	00000	00000	00000	001000	寄存器跳转

MIPS指令详解

1 R型指令

- R型指令存在3种不同类型
  - ♦3寄存器R型指令

指令	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	指令功能
add	000000	rs	rt	rd	00000	100000	寄存器加
sub	000000	rs	rt	rd	00000	100010	寄存器减
and	000000	rs	rt	rd	00000	100100	寄存器与
or	000000	rs	rt	rd	00000	100101	寄存器或
xor	000000	rs	rt	rd	00000	100110	寄存器异或

指令功能：$rd ← $rs op $rt

MIPS指令详解

1 R型指令

- 2寄存器R型指令

指令	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	指令功能
sll	000000	00000	rt	rd	sa	000000	逻辑左移
srl	000000	00000	rt	rd	sa	000010	逻辑右移
sra	000000	00000	rt	rd	sa	000011	算术右移

指令功能：$rd ← $rt shift sa

MIPS指令详解

1 R型指令

- 1寄存器R型指令

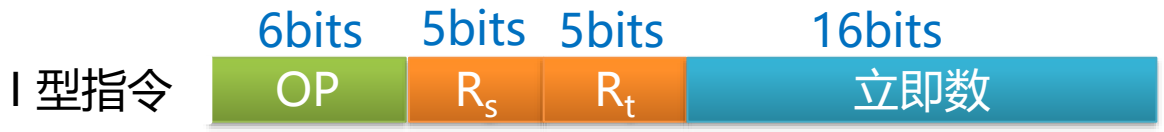
指令	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	指令功能
jr	000000	rs	00000	00000	00000	001000	寄存器跳转

jr rs;

PC ← rs

MIPS指令详解

2 I 型指令



操作数中涉及立即数，结果保存到寄存器；

- op：标识指令的操作功能；
- rs：第1个源操作数，是寄存器操作数；
- rt：目的寄存器编号，用来保存运算结果；
- imm：第2个源操作数，立即数；

MIPS指令详解

2 I 型指令

指令	[31:26]	[25:21]	[20:16]	[15:0]	指令功能
addi	001000	rs	rt	imm	寄存器和立即数“加”
andi	001100	rs	rt	imm	寄存器和立即数“与”
ori	001101	rs	rt	imm	寄存器和立即数“或”
xori	001110	rs	rt	imm	寄存器和立即数“异或”
lw	100011	rs	rt	imm	从存储器中读取数据
sw	101011	rs	rt	imm	把数据保存到存储器
beq	000100	rs	rt	imm	寄存器相等则转移
bne	000101	rs	rt	imm	寄存器不等则转移
lui	001111	00000	rt	imm	设置寄存器的高16位

MIPS指令详解

2 I 型指令

- I型指令存在4种不同类型

♦ 面向运算的I型指令

指令	[31:26]	[25:21]	[20:16]	[15:0]	指令功能
addi	001000	rs	rt	imm	寄存器和立即数“加”
andi	001100	rs	rt	imm	寄存器和立即数“与”
ori	001101	rs	rt	imm	寄存器和立即数“或”
xori	001110	rs	rt	imm	寄存器和立即数“异或”

addi/andi/ori/xori rt, rs, imm; # $rt ← $rs op E(imm)

第一条指令是进行符号扩展，其余是0扩展

MIPS指令详解

2 I 型指令

- 面向访存的I型指令

指令	[31:26]	[25:21]	[20:16]	[15:0]	指令功能
lw	100011	rs	rt	imm	从存储器中读取数据
sw	101011	rs	rt	imm	把数据保存到存储器

MIPS32中唯一两条访问存储器的指令(RISC)

```
lw  rt, imm(rs)   #  $rt ← mem[$rs+E(imm)]
sw  rt, imm(rs)   #  mem[$rs+E(imm)] ←$rt
```

MIPS指令详解

2 I 型指令

- 面向数位设置的I型指令

指令	[31:26]	[25:21]	[20:16]	[15:0]	指令功能
lui	001111	00000	rt	imm	设置寄存器的高16位

lui rt, imm # $rt ← imm<<16（空位补0）

MIPS指令详解

2 I 型指令

- 面向条件转移(分支)的I型指令

指令	[31:26]	[25:21]	[20:16]	[15:0]	指令功能
beq	000100	rs	rt	imm	寄存器相等则转移
bne	000101	rs	rt	imm	寄存器不等则转移

beq rs, rt, imm　　#if($rs==$rt) PC←PC+E(imm)<<2

bne rs, rt, imm　　#if($rs!=$rt) PC←PC+E(imm)<<2

是标准的PC相对寻址方式

其中imm要先“带符号扩展”成32位，再左移2位。

MIPS指令详解

3 J 型指令

指令	[31:26]	[25:0]	指令功能
j	000010	address	无条件跳转
jal	001100	address	调用与联接

j address; $PC← ($PC+4)_{H4} ∪ (address<<2)

jal address; $ra←$PC+4 (保存返回地址)

$PC ← ($PC+4)_{H4} ∪ (address<<2)

执行指令的主要步骤（MIPS）

1、取指（Fetch）	从存储器取指令，更新PC
2、译码（Decode）	指令译码，从寄存器堆读出寄存器的值
3、执行（Execute）	运算指令：进行算术逻辑运算 访存指令：计算存储器的地址
4、访存（Memory）	Load指令：从存储器读数据 Store指令：将数据写入存储器
5、回写（Write-back）	将数据写入寄存器堆

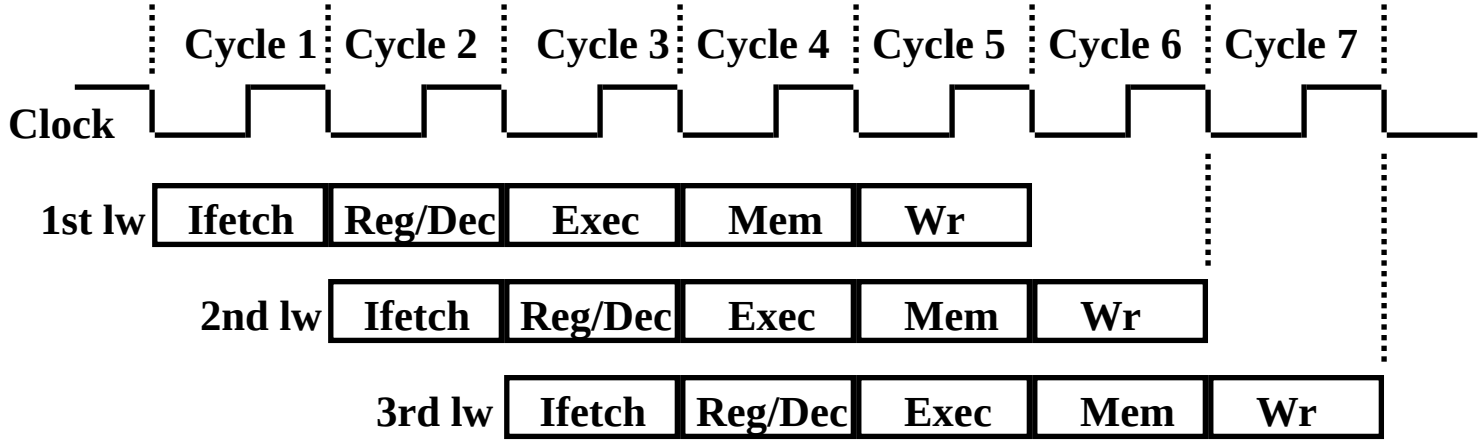
流水线指令集的设计

- **具有什么特征的指令集有利于流水线执行呢？**
  - **长度尽量一致，有利于简化取指令和指令译码操作**
    - MIPS指令32位，下址计算方便: PC+4
    - X86指令从1字节到17字节不等，使取指部件及其复杂
  - **格式少，且源寄存器位置相同，有利于在指令未知时就可取操作数**
    - MIPS指令的Rs和Rt位置一定，在指令译码时就可读Rs和Rt的值
      （若位置随指令不同而不同，则需先确定指令后才能取寄存器编号）
  - **load / Store指令才能访问存储器，有利于减少操作步骤，规整流水线**
    - lw/sw指令的地址计算和运算指令的执行步骤规整在同一个周期
    - X86运算类指令操作数可为内存数据，需计算地址、访存、执行
  - **内存中”对齐”存放，有利于减少访存次数和流水线的规整**

总之，规整、简单和一致等特性有利于指令的流水线执行

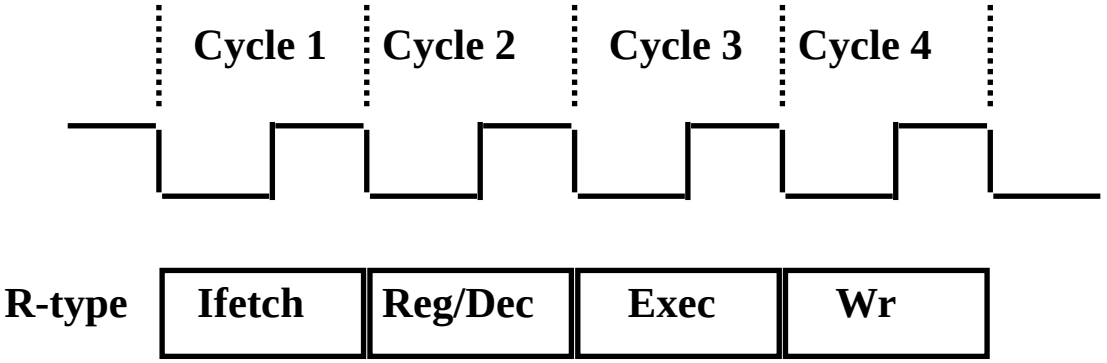
流水线执行方式能大大提高指令吞吐率，现代计算机都采用流水线方式！

Load指令的流水线



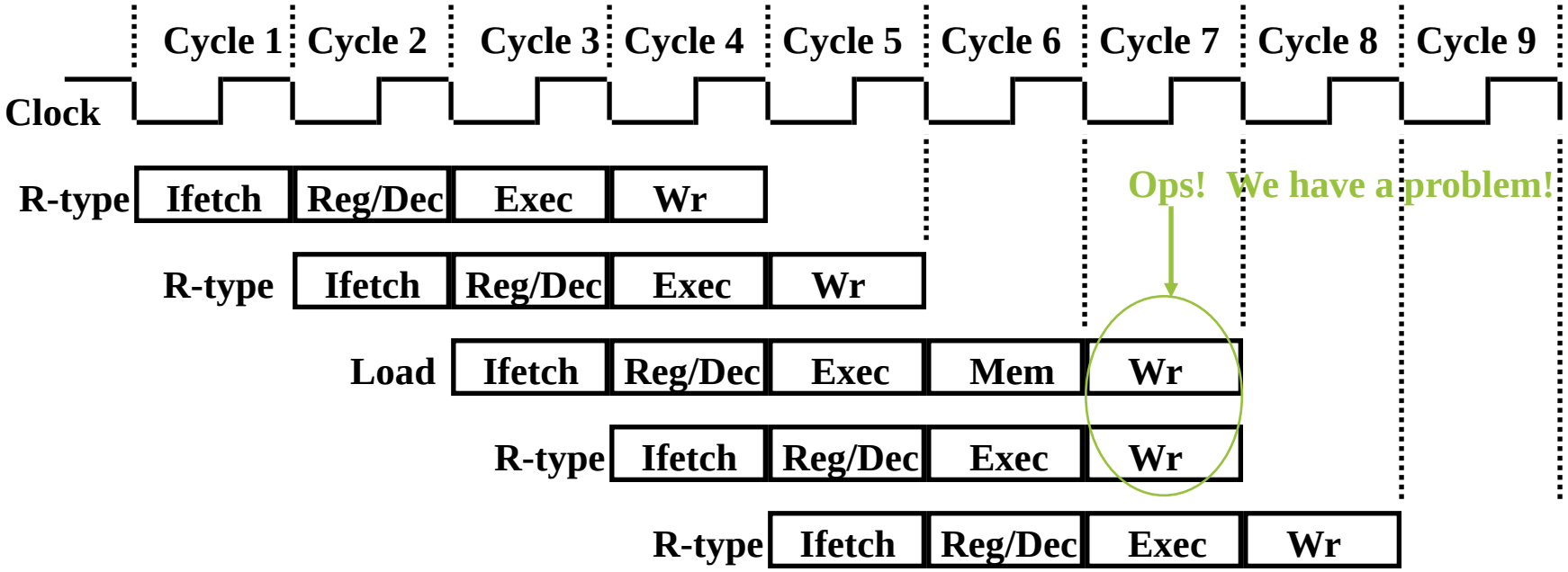
- 每个周期有五个功能部件同时在工作
- 后面指令在前面完成取指后马上开始
- 每个**load**指令仍然需要五个周期完成
- 但是吞吐率**(throughput)**提高许多，理想情况下，有：
  - 每个周期有一条指令进入流水线
  - 每个周期都有一条指令完成
  - 每条指令的有效周期(CPI)为1

R-type指令的4个阶段



- Ifetch: 取指令并计算PC+4
- Reg/Dec: 从寄存器取数，同时指令在译码器进行译码
- Exec: 在ALU中对操作数进行计算
- Wr: ALU计算的结果写到寄存器

含R-type和 Load 指令的流水线

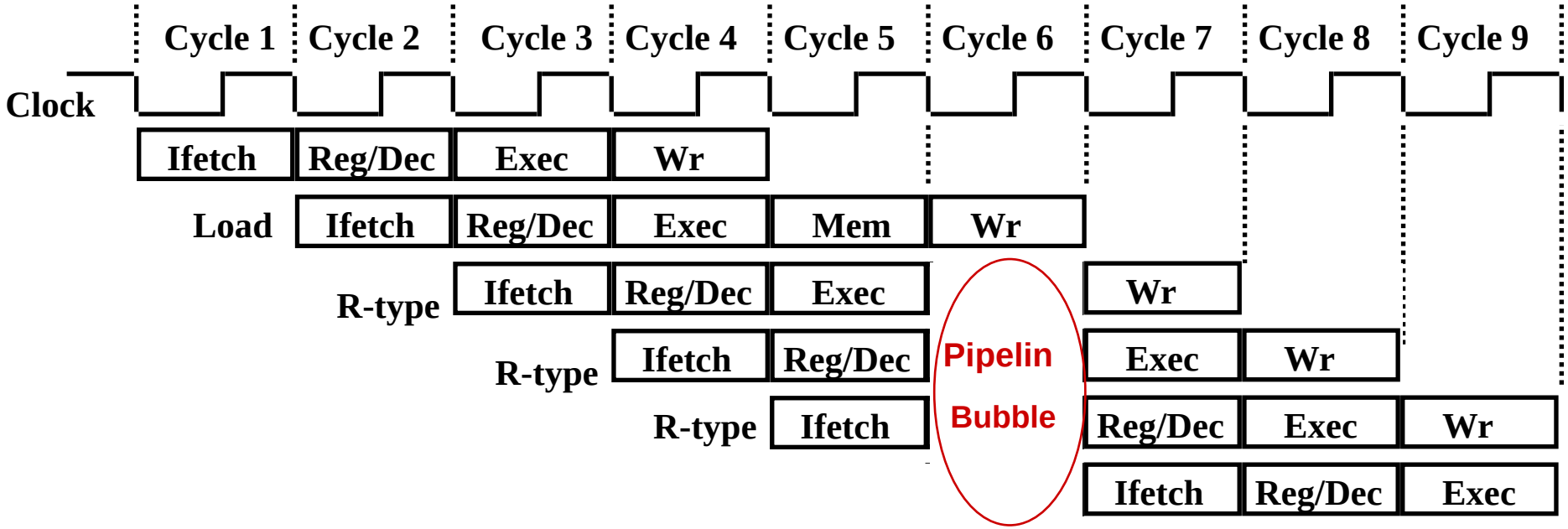


- 上述流水线有个问题：两条指令试图同时写寄存器，因为
  - Load在第5阶段用寄存器写口
  - R-type在第4 阶段用寄存器写口

  或称为资源冲突！
- 把一个功能部件同时被多条指令使用的现象称为结构冒险(Struture Hazard)
- 为了流水线能顺利工作，规定：
  - 每个功能部件每条指令只能用一次（如：写口不能用两次或以上）
  - 每个功能部件必须在相同的阶段被使用（如：写口总是在第五阶段被使用）

可以用以下两种方法解决上述结构冒险问题！

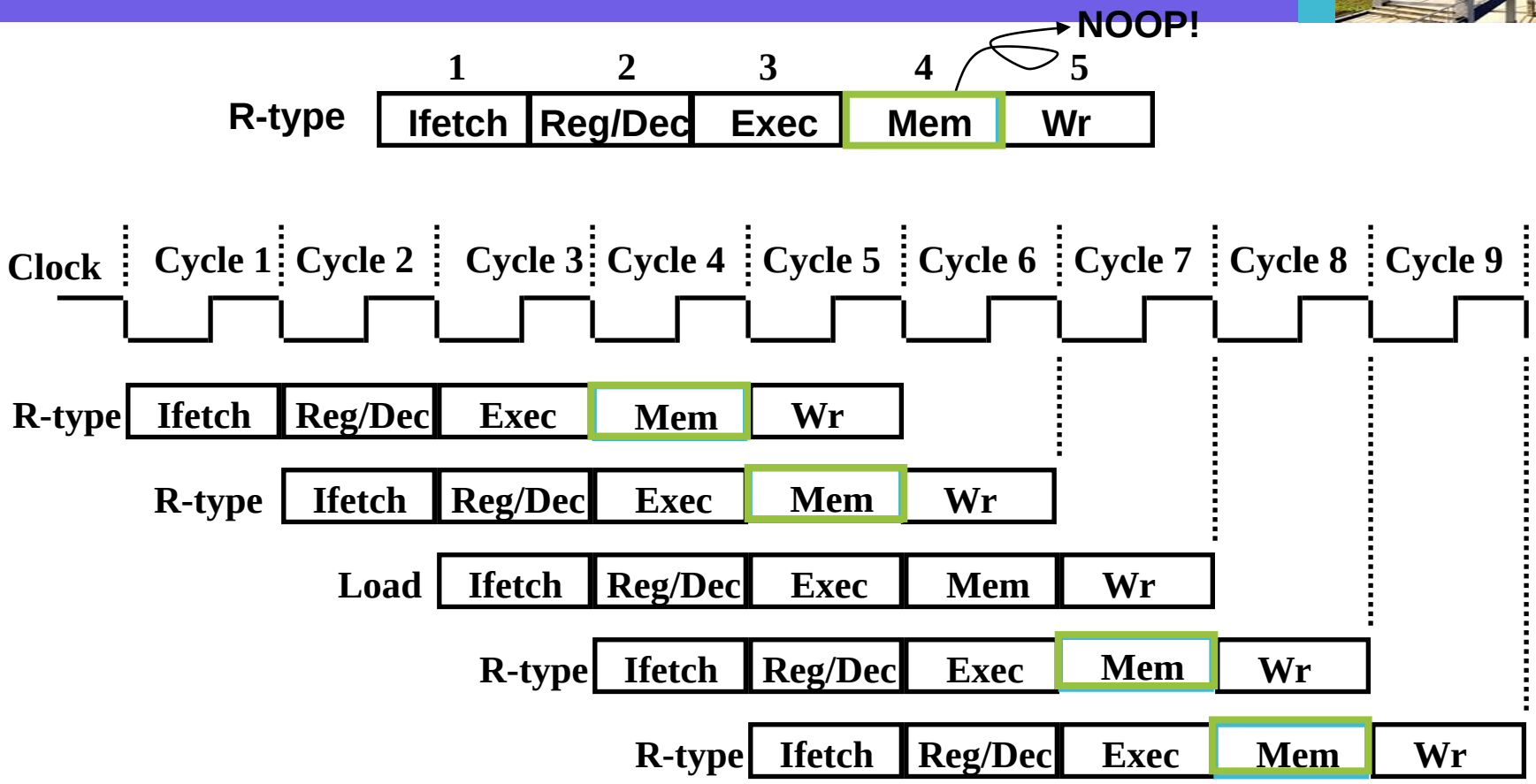
解决方案1: 在流水线中插入“Bubble”（气泡）



- 插入“**Bubble**”到流水线中，以禁止同一周期有两次写寄存器。缺点：
  - 控制逻辑复杂
  - 第5周期没有指令被完成（CPI不是1，而实际上是2）

方案不可行！

解决方案2: R-type的Wr操作延后一个周期执行



- 加一个NOP阶段以延迟“写”操作:
  - 把“写”操作安排在第5阶段, 这样使R-Type的Mem阶段为空NOP

这样使流水线中的每条指令都有相同多个阶段!

流水线冒险的处理

- 流水线冒险的几种类型
- 数据冒险的现象和对策
  - 数据冒险的种类
    - 相关的数据是ALU结果：可以通过转发解决
    - 相关的数据是DM读出的内容：随后的指令需被阻塞一个时钟
  - 数据冒险和转发
    - 转发检测 / 转发控制
  - 数据冒险和阻塞
    - 阻塞检测 / 阻塞控制
- 控制冒险的现象和对策
  - 静态分支预测技术
  - 动态分支预测技术
  - 缩短分支延迟技术
- 流水线中对异常和中断的处理
- 访问缺失对流水线的影响

流水线的三种冲突/冒险（Hazard）情况

- **Hazards：指流水线遇到无法正确执行后续指令或执行了不该执行的指令**
  - Structural hazards (hardware resource conflicts):

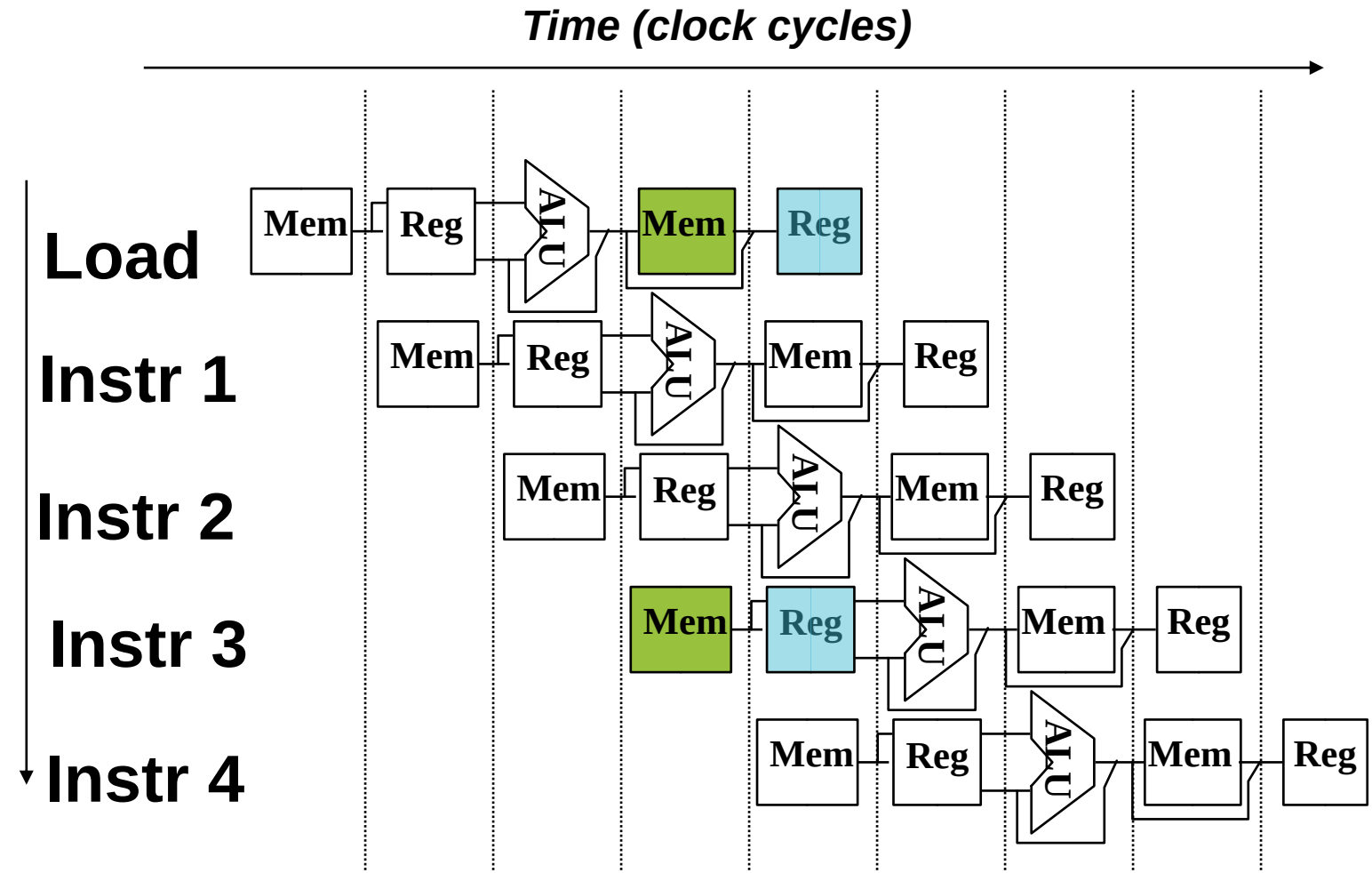
    现象：同一个部件同时被不同指令所使用
    - 一个部件每条指令只能使用1次，且只能在特定周期使用
    - 设置多个部件，以避免冲突。如指令存储器IM 和数据存储器DM分开
  - Data hazards (data dependencies):

    现象：后面指令用到前面指令结果时，前面指令结果还没产生
    - 采用转发(Forwarding/Bypassing)技术
    - Load-use冒险需要一次阻塞(stall)
    - 编译程序优化指令顺序
  - Control (Branch) hazards (changes in program flow):

    现象：转移或异常改变执行流程，顺序执行指令在目标地址产生前已被取出
    - 采用静态或动态分支预测
    - 编译程序优化指令顺序(实行分支延迟)

SKIP

Structural Hazard（结构冒险）现象



如果只有一个存储器，则在Load指令取数据同时又取指令的话，则发生冲突！

如果不对寄存器堆的写口和读口独立设置的话，则发生冲突！

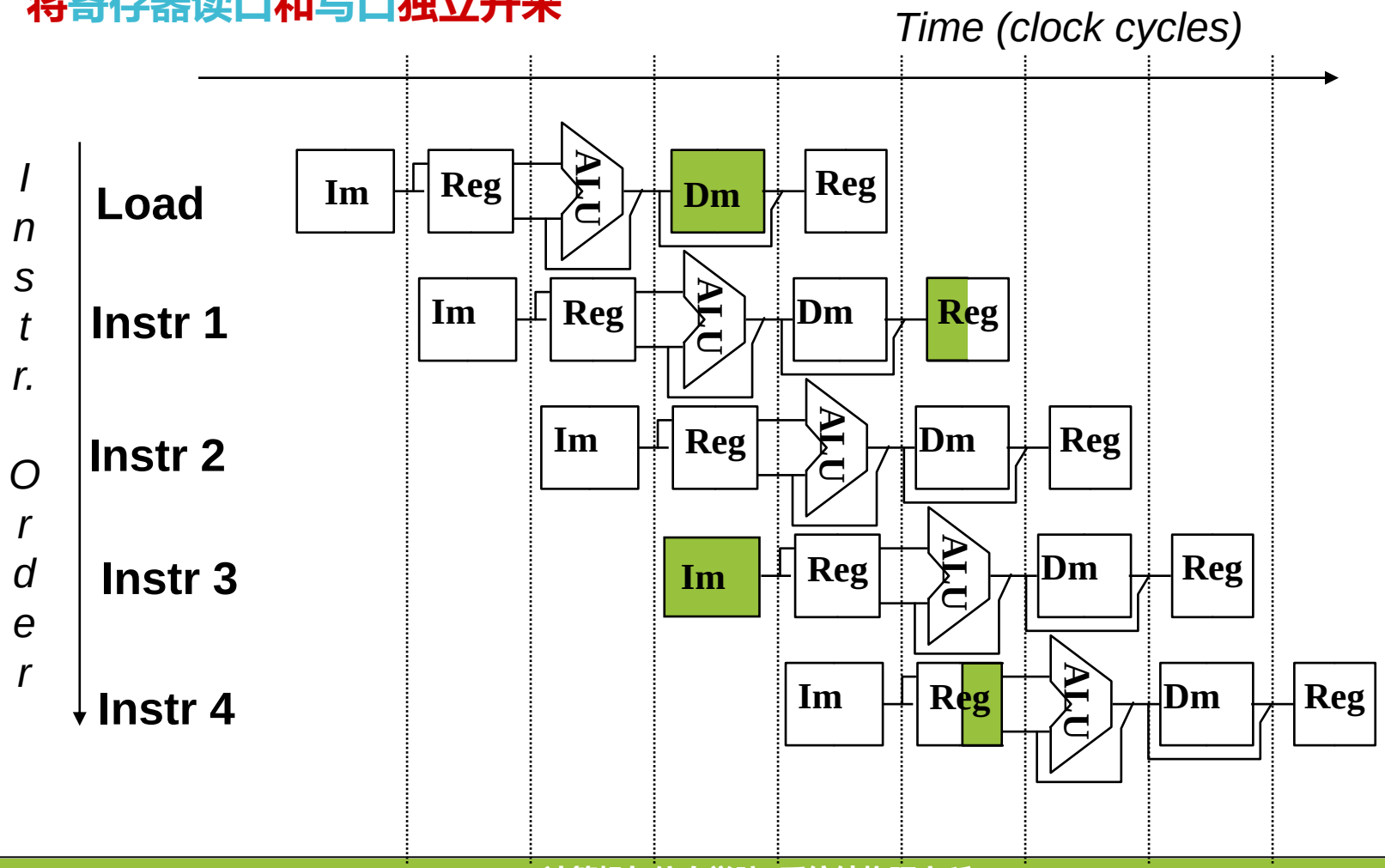
结构冒险也称为硬件资源冲突：同一个执行部件被多条指令使用。

Structural Hazard的解决方法

为了避免结构冒险，规定流水线数据通路中功能部件的设置原则为：
每个部件在特定的阶段被用！（如：ALU总在第三阶段被用！）

将Instruction Memory (Im) 和 Data Memory (Dm)分开

将寄存器读口和写口独立开来



Data Hazard现象

举例说明：以下指令序列中，寄存器r1会发生数据冒险

想一下，哪条指令的r1是老的值？
哪条是新的值？

```
add r1 ,r2,r3
sub r4, r1 ,r3
and r6, r1 ,r7
or  r8, r1 ,r9
xor r10, r1 ,r11
```

读r1时，add指令正在执行加法(EXE)，老值!

读r1时，add指令正在传递加法结果(MEM), 老值!

读r1时，add指令正在写加法结果到r1(WB), 老值!

读r1时，add指令已经把加法结果写到r1, 新值

画出流水线图能很清楚理解！

补充：三类数据冒险现象

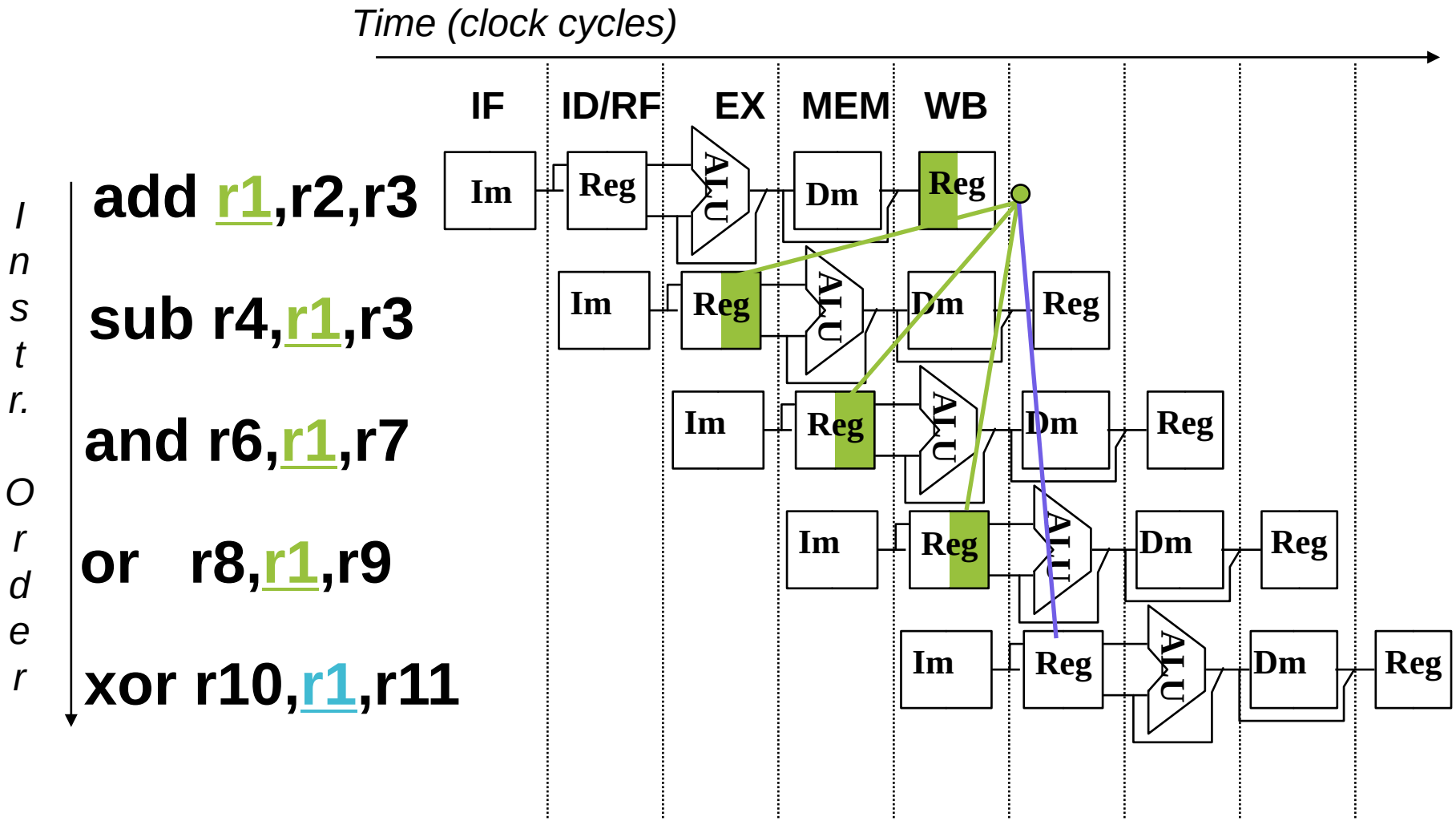
RAW：写后读（基本流水线中经常发生，如上例）

WAR：读后写（基本流水线中不会发生，多个功能部件时会发生）

WAW：写后写（基本流水线中不会发生，多个功能部件时会发生）

本讲介绍基本流水线，所以仅考虑RAW冒险

Data Hazard on r1



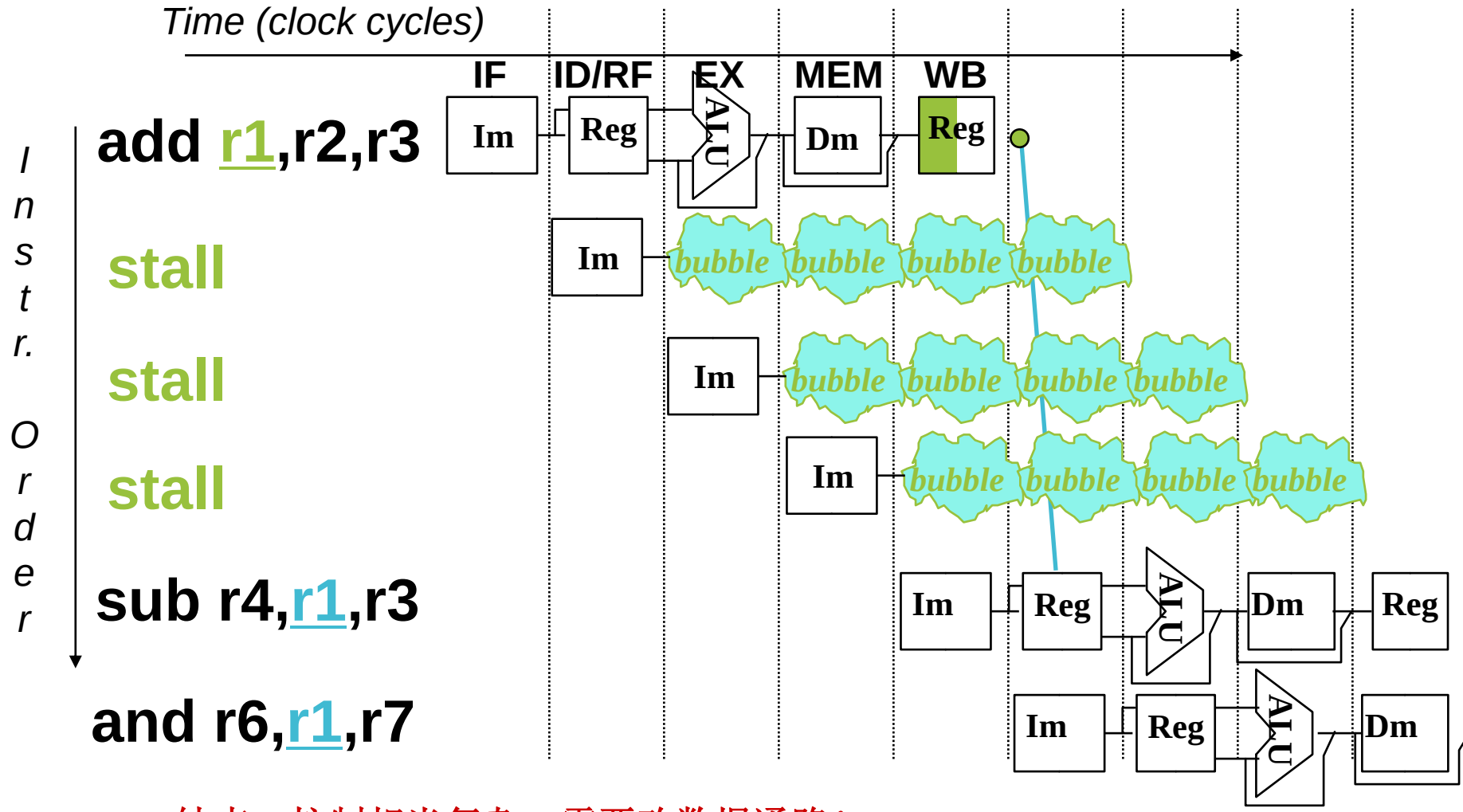
最后一条指令的r1才是新的值！

如何解决这个问题？

方案1: 在硬件上采取措施，使相关指令延迟执行

- 硬件上通过阻塞(stall)方式阻止后续指令执行，延迟到有新值以后！

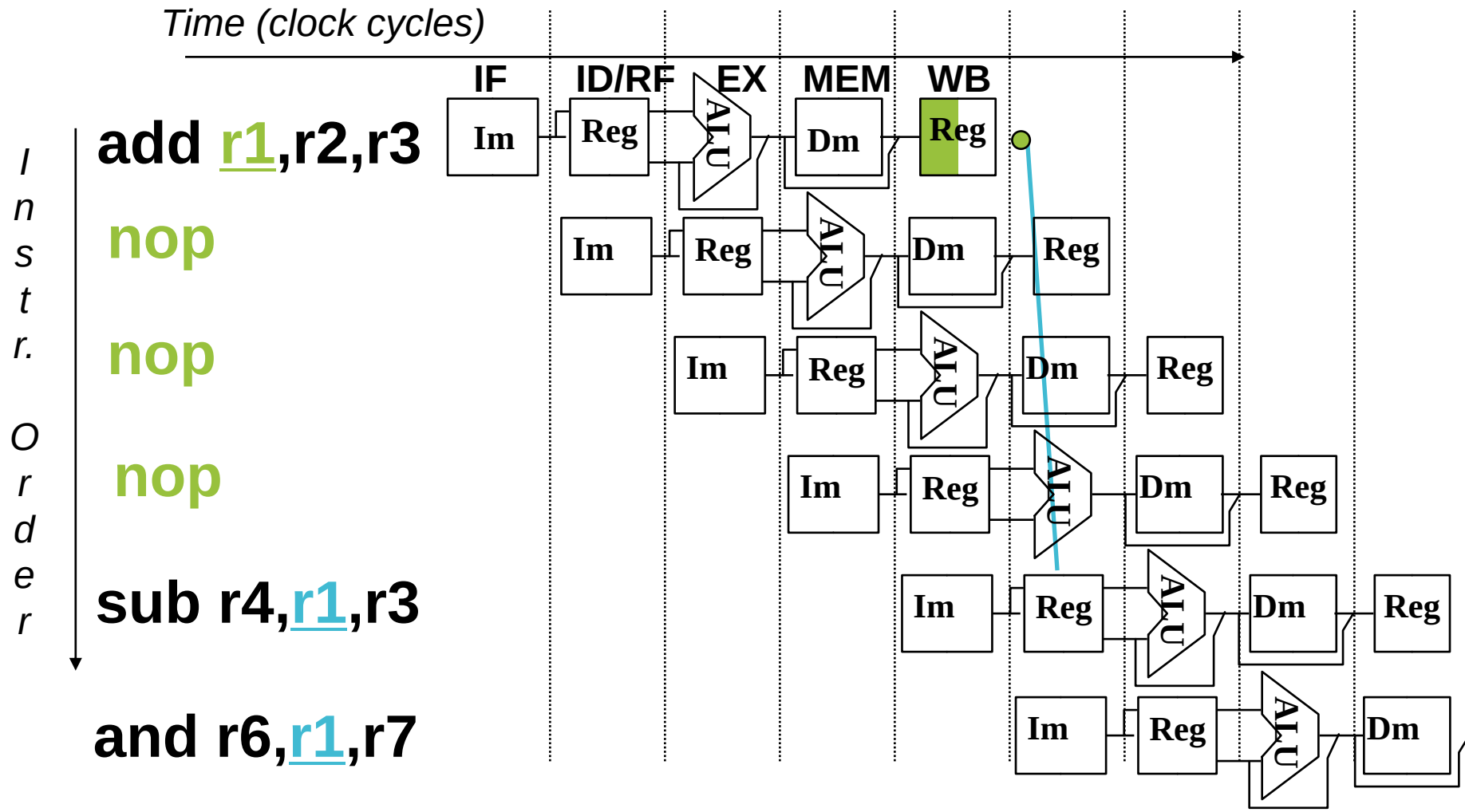
  这种做法称为流水线阻塞，也称为“气泡Bubble”



- 缺点：控制相当复杂，需要改数据通路！

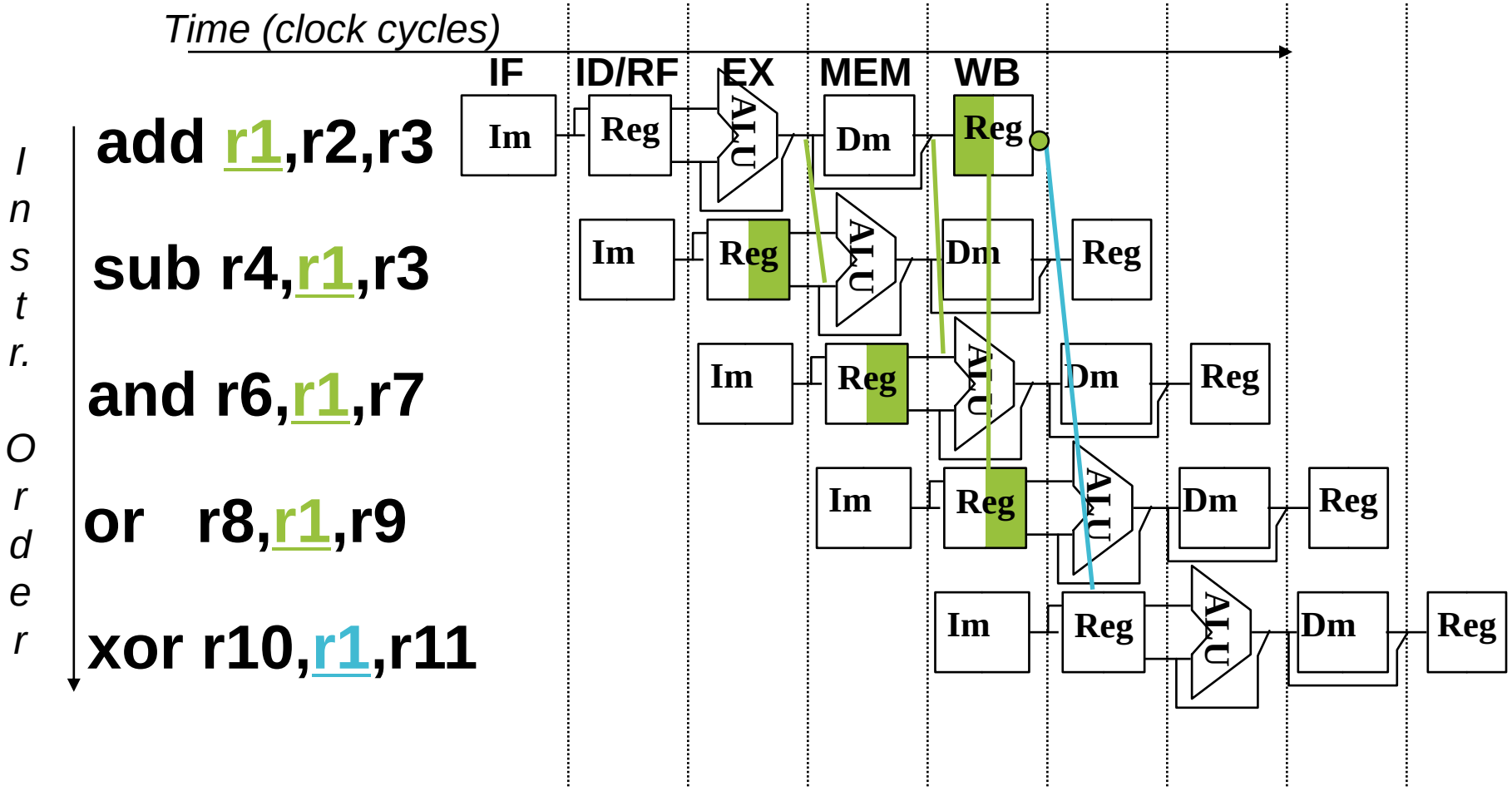
方案 2: 软件上插入无关指令

- 最差的做法：由编译器插入三条NOP指令，浪费三条指令的空间和时间



方案3：利用DataPath中的中间数据

- 仔细观察后发现：流水段寄存器中已有需要的值r1！ 在哪个流水段R中？



1. 把数据从流水段寄存器中直接取到ALU的输入端

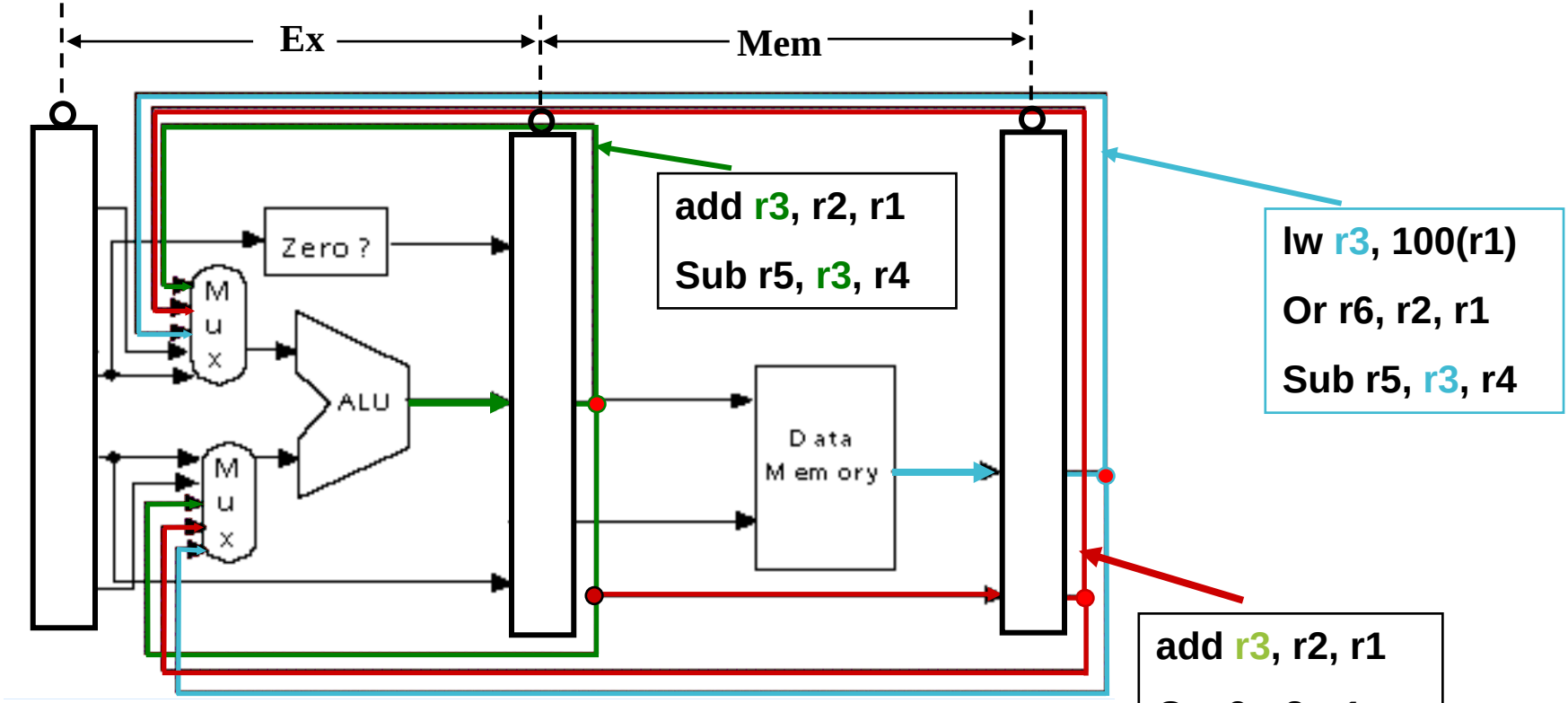
2. 寄存器写/读口分别在前/后半周期，使写入被直接读出

称为转发（Forwading）或旁路（Bypassing）

BACK

硬件上的改动以支持“转发”技术

- 加MUX，使流水段寄存器值返送ALU输入端
- 假定流水段寄存器能读出新写入的值 (否则，需要更多的转发数据)



如果指令序列为：

lw r3, 100(r1)

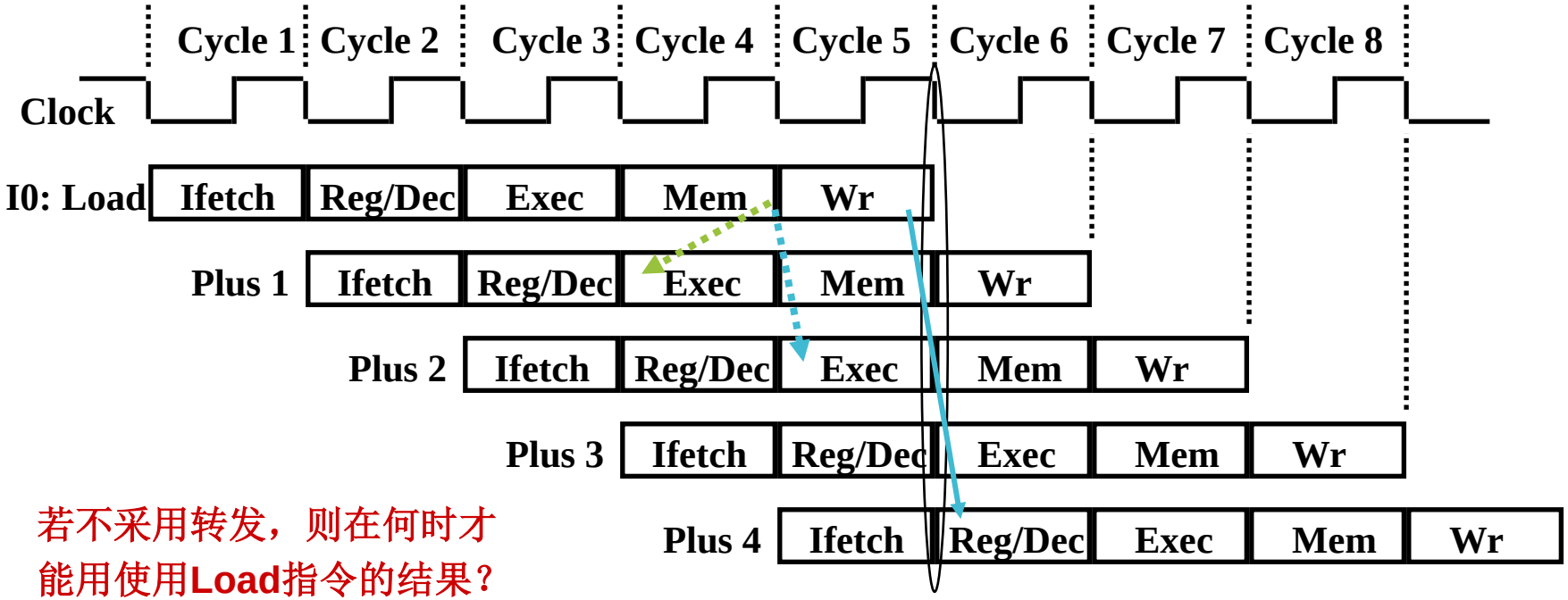
Or r6, r3, r1

Sub r5, r3, r4

能用“转发”技术解决第1、2两条指令间的数据冒险吗？

请看后面的幻灯片！

复习: Load指令引起的延迟



若不采用转发，则在何时才能用使用Load指令的结果？

- **Load**指令最早在哪个流水线寄存器中开始有后续指令需要的值？

  实际上，在第四周期结束时，数据在流水段寄存器中已经有值。

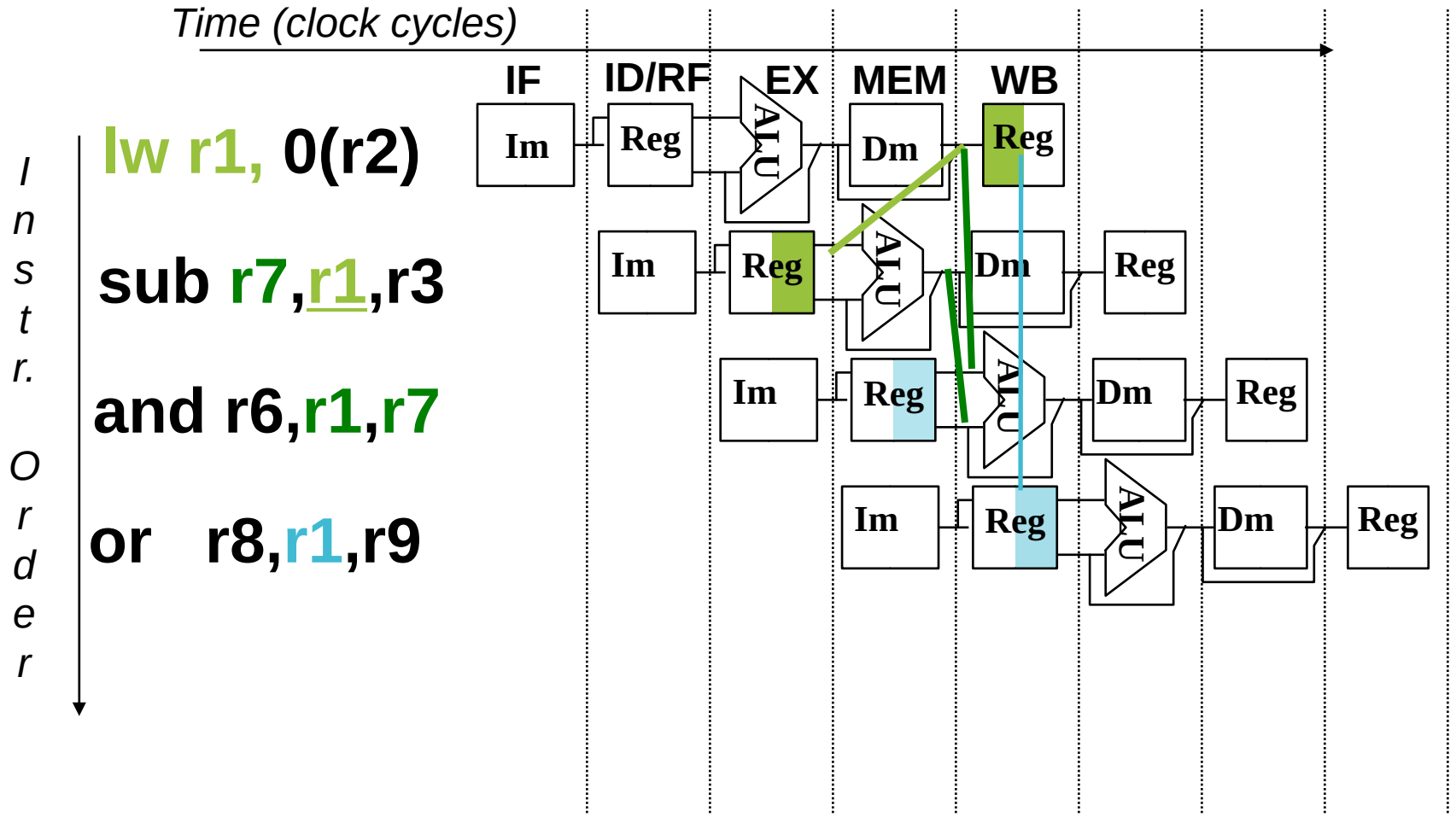
  采用数据转发技术可以使load指令后面第二条指令得到所需的值

  但不能解决load指令和随后的第一条指令间的数据冒险，要延迟执行一条指令！

这种load指令和随后指令间的数据冒险，称为“装入- 使用数据冒险(load- use Data Hazard)”

BACK

“Forwarding”技术使Load-use冒险只需延迟一个周期

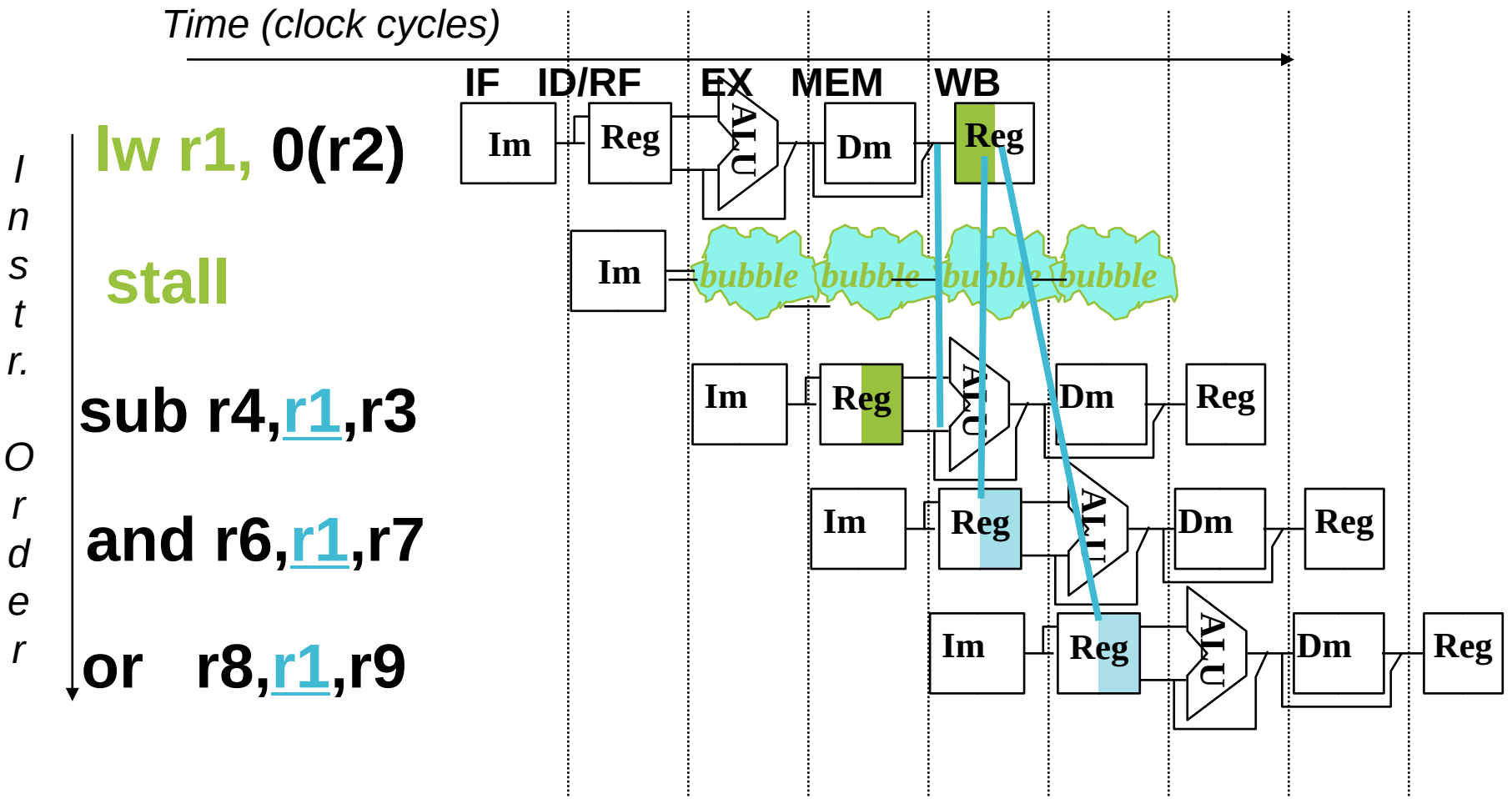


采用“转发”后仅第二条指令 **SUB r7,r1,r3** 不能按时执行！需要阻塞一个周期。

发生“装入- 使用数据冒险”时，需要对**load**后的指令阻塞一个时钟周期！

方案1: 硬件阻止指令执行来解决load-use

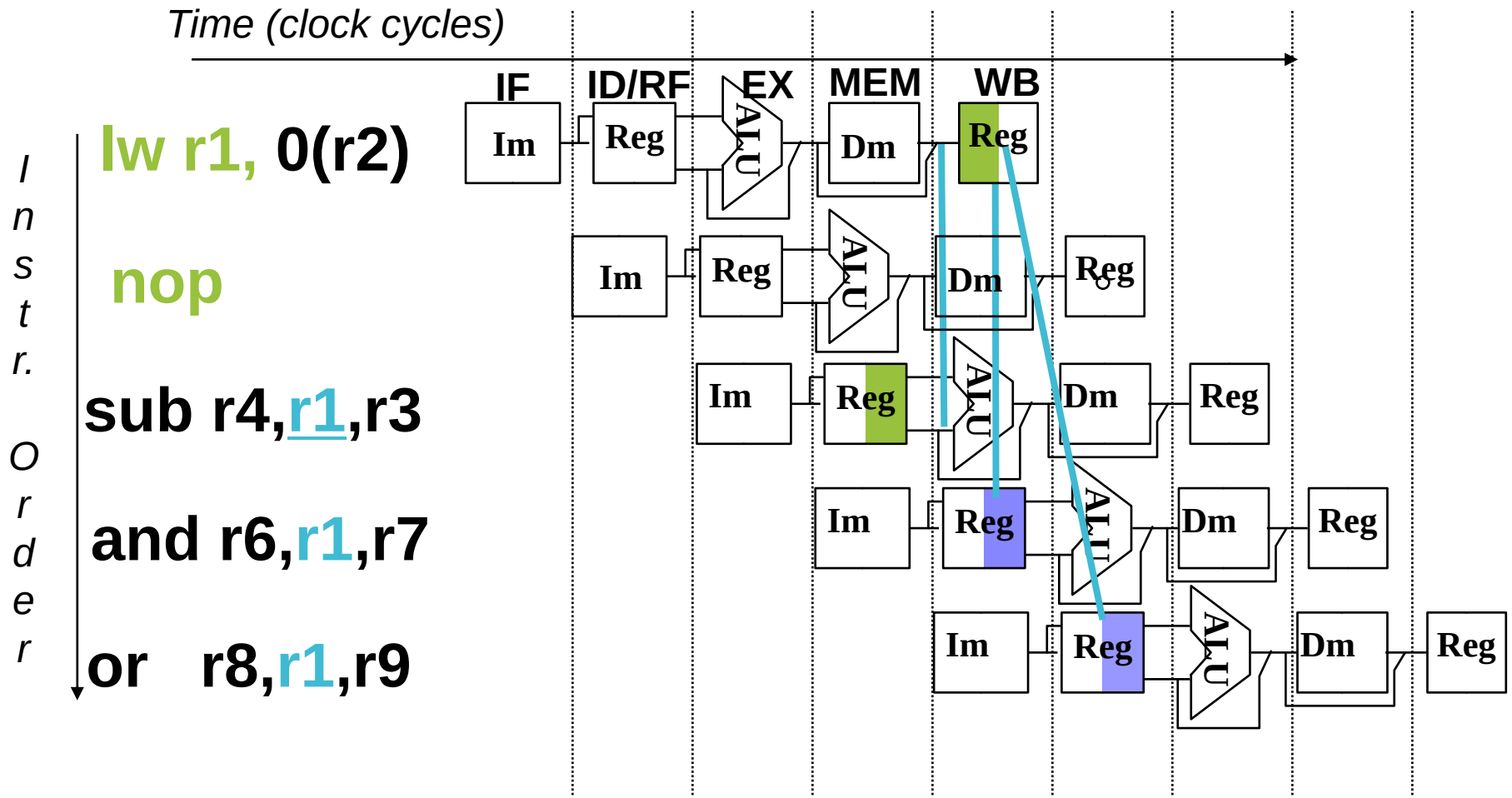
用硬件阻塞一个周期（指令被重复执行一次）



方案2: 软件上插入NOP指令来解决load-use

- 用软件插入一条NOP指令！（有些处理器不支持硬件阻塞处理）

例如：MIPS 1 处理器没有硬件阻塞处理，而由编译器（或汇编程序员） 来处理。



方案3：编译器进行指令顺序调整来解决load-use

以下源程序可生成两种不同的代码，优化的代码可避免Load阻塞

a = b + c;

d = e – f;

假定 **a, b, c, d ,e, f** 在内存

Slow code:

```
lw      $2, [b]
lw      $3, [ c ]
add     $1, $2, $3
sw      [a, $1
lw      $5, [e]
lw      $6, [f]
sub     $4, $5, $6
sw      [d], $4
```

调整后

Fast code:

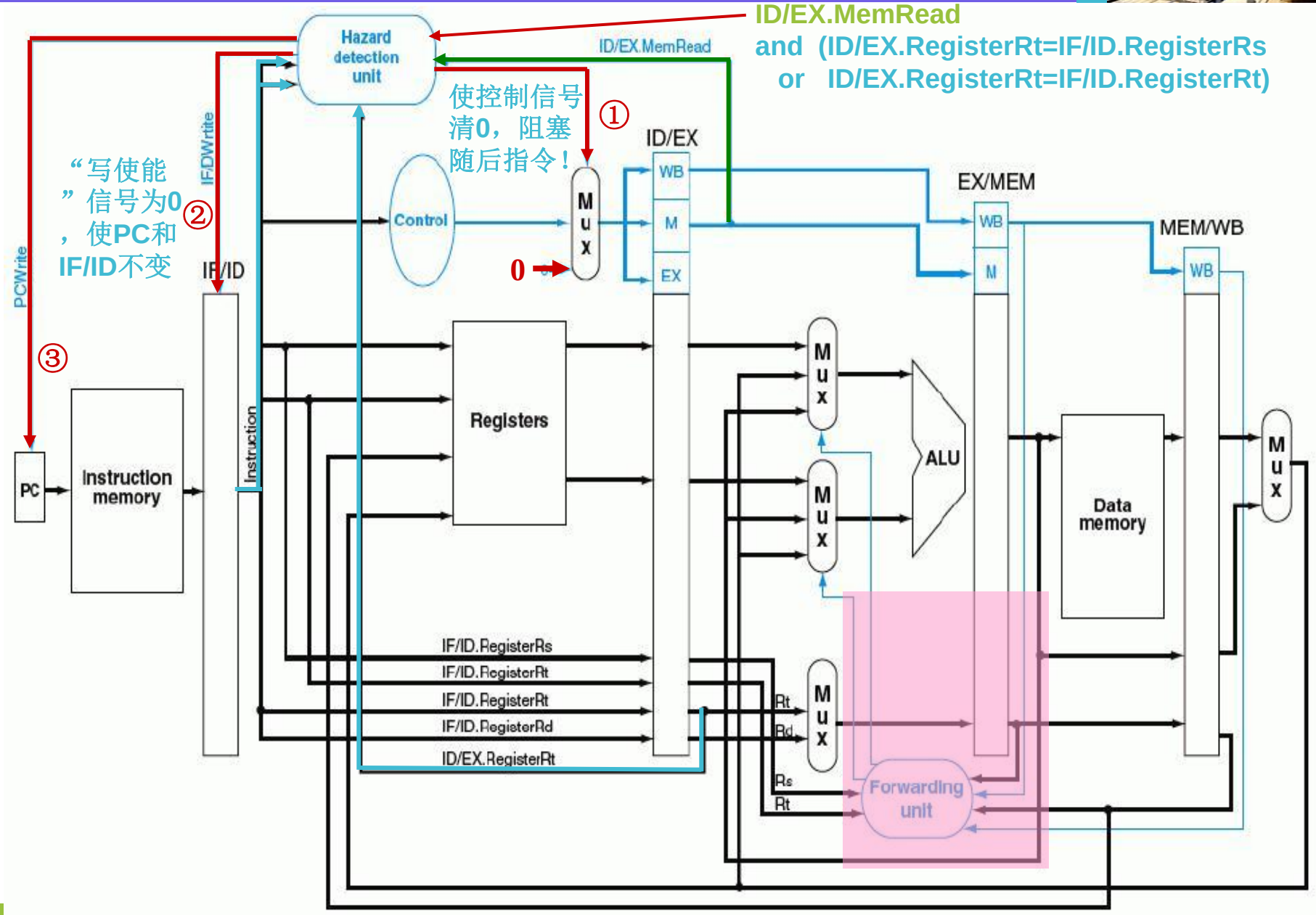
```
lw      $2, [b]
lw      $3, [c]
lw      $5, [e]
add     $1, $2, $3
lw      $6, [f]
sw      [a], $1
sub     $4, $5, $6
sw      [d], $4
```

编译器的优化很重要！

数据冒险的解决方法

- 方法1：硬件阻塞（stall）
- 方法2：软件插入“NOP”指令
- 方法3：编译优化：调整指令顺序，能解决所有数据冒险吗?
- 方法4：合理实现寄存器堆的读/写操作，能解决所有数据冒险吗?
  - 前半时钟周期写，后半时钟周期读
  - 若同一个时钟内前面指令写入的数据正好是后面指令所读数据，则不会发生数据冒险
- 方法5：转发（Forwarding或Bypassing 旁路）技术，能解决所有数据冒险吗?
  - 若相关数据是ALU结果，则如何?
    可通过转发解决
  - 若相关数据是上条指令DM读出内容，则如何?
    不能通过转发解决，随后指令需被阻塞一个时钟 或 加NOP指令

带“转发”和“阻塞”检测的流水线数据通路



Control Hazard的解决方法

- **方法1：硬件上阻塞（stall）分支指令后三条指令的执行**
  - 使后面三条指令清0或 其操作信号清0，以插入三条NOP指令
- **方法2：软件上插入三条“NOP”指令**

  （以上两种方法的效率太低，需结合分支预测进行）
- **方法3：分支预测（Predict）**
  - 简单（静态）预测：
    - 总是预测条件不满足(not taken)，即：继续执行分支指令的后续指令

      可加启发式规则：在特定情况下总是预测满足(taken)，其他情况总是预测不满足。如：循环顶（底）部分支总是预测为不满足（满足）。能达65%-85%的预测准确率
  - 动态预测：
    - 根据程序执行的历史情况，进行动态预测调整，能达90%的预测准确率

  注：采用分支预测方式时，流水线控制必须确保错误预测指令的执行结果不能生效，而且要能从正确的分支地址处重新启动流水线工作
- **方法4：延迟分支（Delayed branch）（通过编译程序优化指令顺序！）**
  - 把分支指令前面与分支指令无关的指令调到分支指令后面执行，也称延迟转移

另一种控制冒险：异常或中断控制冒险的处理

动态预测基本方法

- **采用一位预测位：总是按上次实际发生的情况来预测下次**
  - 1表示最近一次发生过转移（taken），0表示未发生（not taken）
  - 预测时，若为1，则预测下次taken，若为0，则预测下次not taken
  - 实际执行时，若预测错，则该位取反，否则，该位不变
  - 可用一个简单的预测状态图表示
  - 缺点：当连续两次的分支情况发生改变时，预测错误
    - 例如，循环迭代分支时，第一次和最后一次会发生预测错误，因为循环的第一次和最后一次都会改变分支情况，而在循环中间的各次总是会发生分支，按上次的实际情况预测时，都不会错。
- **采用二位预测位**
  - 用2位组合四种情况来表示预测和实际转移情况
  - 按照预测状态图进行预测和调整
  - 在连续两次分支发生不同时，只会有一次预测错误

采用比较多的是二位预测位，也有采用二位以上预测位。
如：Pentium 4 的BTB2采用4位预测位

BACK

小结

- **流水线冒险的几种类型：**
  - 资源冲突、数据相关、控制相关（改变指令流的执行方向）
- **数据冒险的现象和对策**
  - 数据冒险的种类
    - 相关的数据是ALU结果，可以通过转发解决
    - 相关的数据是DM读出的内容，随后的指令需被阻塞一个时钟
  - 数据冒险和转发
    - 转发检测 / 转发控制
  - 数据冒险和阻塞
    - 阻塞检测 / 阻塞控制
- **控制冒险的现象和对策**
  - 静态分支预测技术
  - 缩短分支延迟技术
  - 动态分支预测技术
- **异常和中断是一种特殊的控制冒险**
- **访存缺失（Cache缺失、TLB缺失、缺页）会引起流水线阻塞**

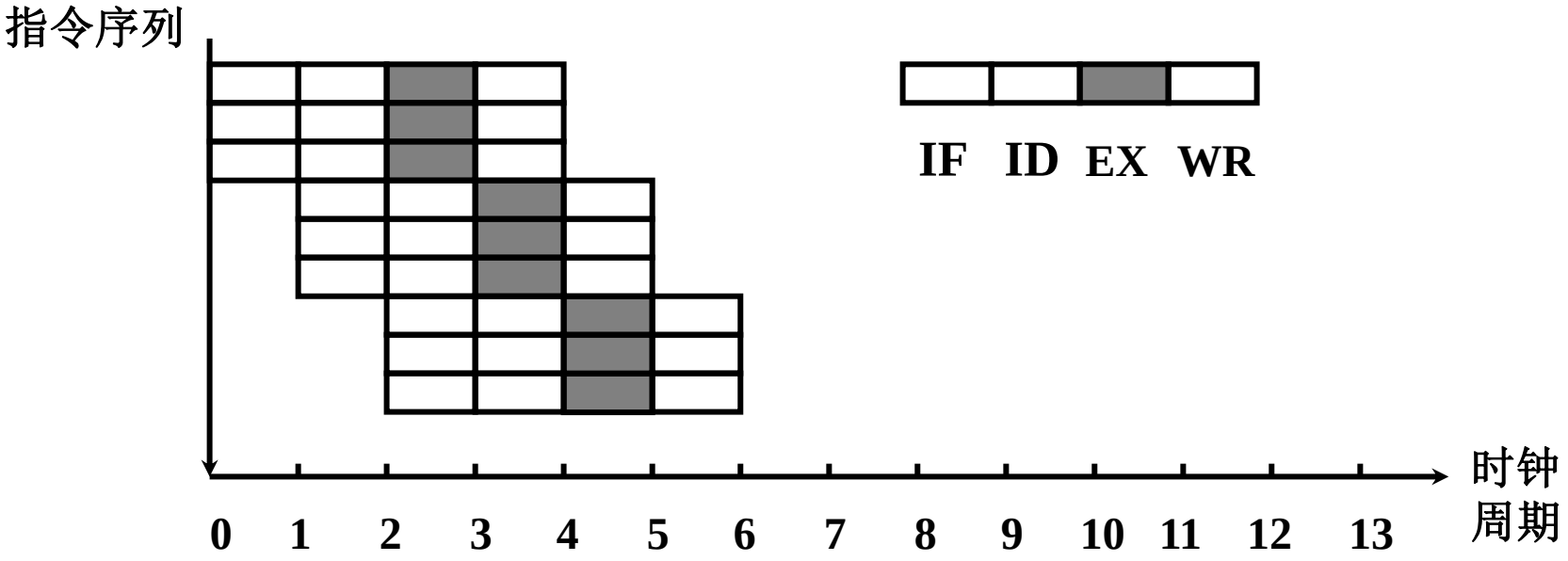
五、流水线的多发技术

- 流水线的多发技术：为了提高流水线性能，设法在一个时钟周期内产生更多条指令。
  - **超标量技术**
  - **超流水线技术**
  - **超长指令字**

1. 超标量技术(Super Scalar)

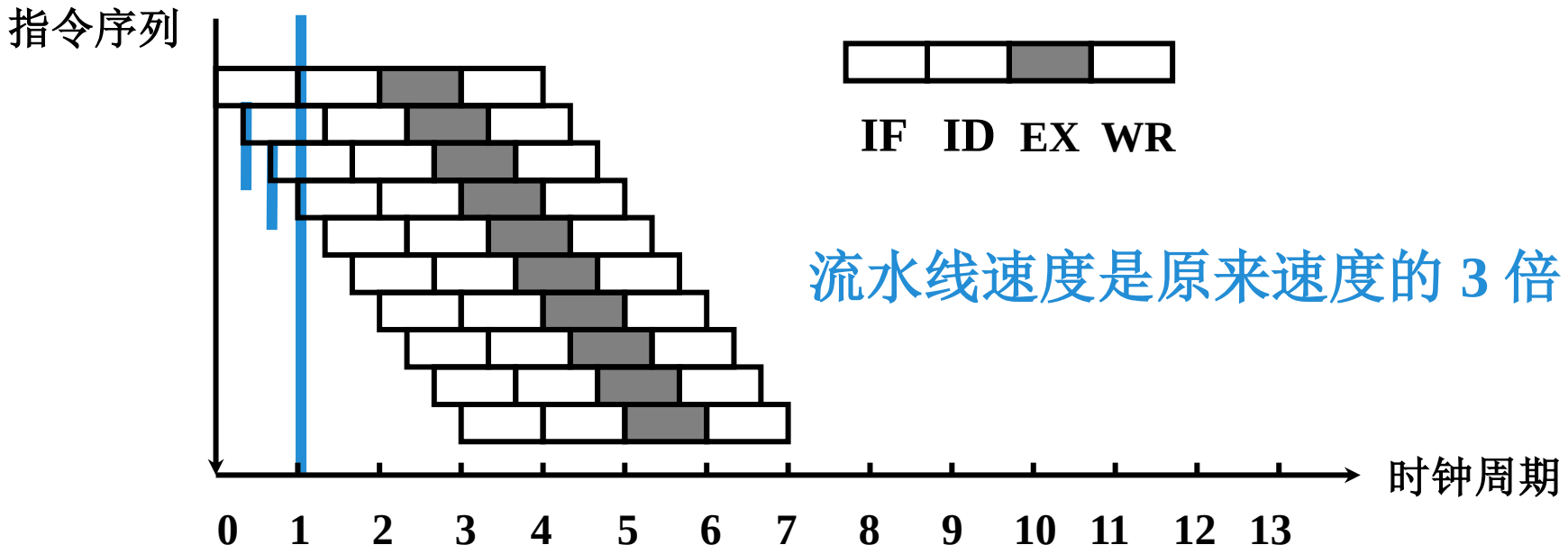
- 每个时钟周期内可 并发多条独立指令
- 配置多个功能部件，一个时钟周期内，有多个功能部件同时执行多条指令 。
- 不能调整 指令的 执行顺序

可以通过编译优化技术，把可并行执行的指令搭配起来



2. 超流水线技术（Super Pipe Lining）

- 将流水寄存器插入流水线段，好比将流水线再次分段
- 在 一个时钟周期 内 再分段 （3 段）
  在一个时钟周期内 一个功能部件使用多次（3 次）
- 不能调整 指令的 执行顺序　靠编译程序解决优化问题



2. 超流水线技术（Super Pipe Lining）

- 将五级流水线细分为更多的阶段，增加流水线的深度
- 提升时钟频率，从而提高指令吞吐率
- 五级流水线

- 时钟周期：200ps+50ps=250ps

- 十级流水线

- 时钟周期：100ps+50ps=150ps

流水线的深度

- 流水线的级数是越多越好吗?

答：否!

- 五级流水线
  - 时钟周期：200ps+50ps=250ps
  - 单条指令的延迟：1250ps
  - 流水线寄存器延迟所占比例：50ps / 250ps = 20%
- 十级流水线

  - 时钟周期：100ps+50ps=150ps
  - 单条指令的延迟：1500ps
  - 流水线寄存器延迟所占比例：50ps / 150ps = 33%

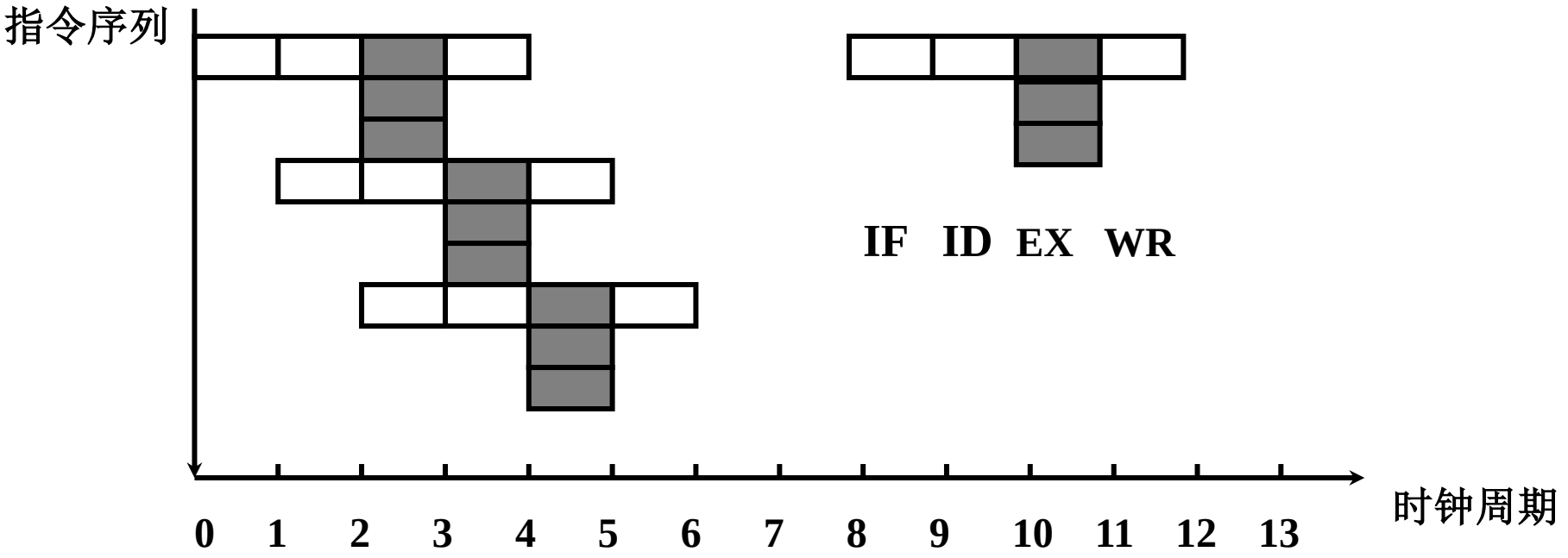
3. 超长指令字技术（VLIW）

- 由编译程序 挖掘 出指令间 潜在 的 并行性

  将 多条 能 并行操作 的指令组合成 一条

  具有 多个操作码字段 的 超长指令字（可达几百位）

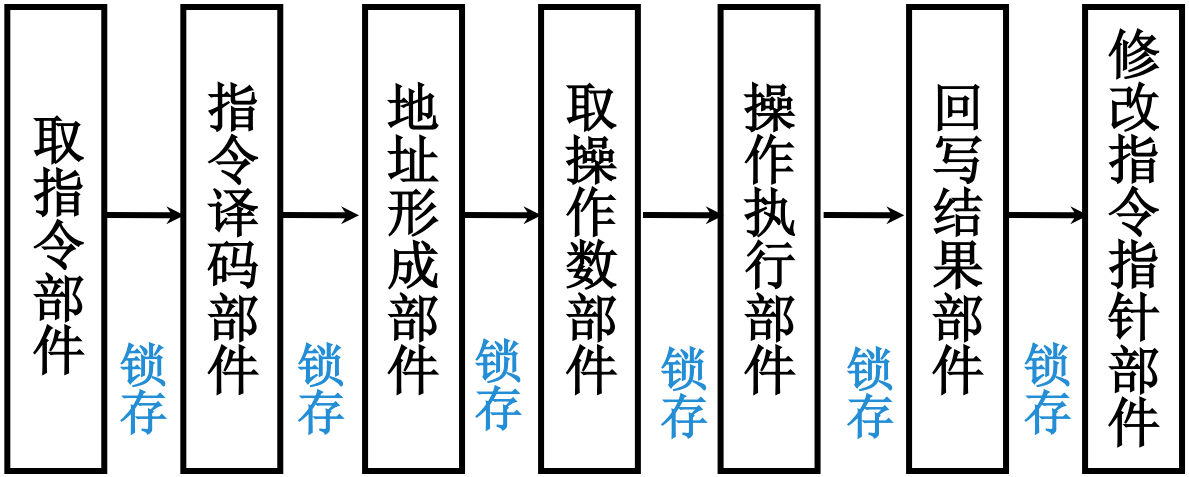
- 形成的超长指令字控制VLIW机中独立工作的多个处理部件



六、流水线结构

1. 指令流水线结构

完成一条指令分 7 段， 每段需一个时钟周期



若 流水线不出现断流　　1 个时钟周期出 1 结果

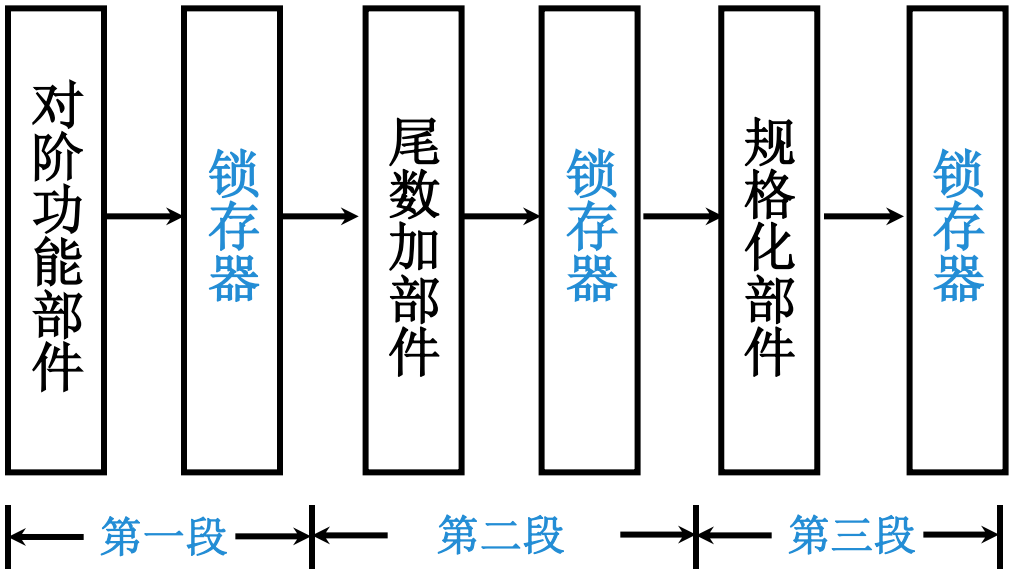
不采用流水技术　　7 个时钟周期出 1 结果

理想情况下，7 级流水 的速度是不采用流水技术的 7 倍

2. 运算流水线

完成 浮点加减 运算 可分

对阶、尾数求和、规格化 三段



分段原则　每段 操作时间 尽量 一致

Computer Organization

Thank You !

Institute of Computer Architecture

合肥工业大学
系统结构研究所
陈 田