
第8章 中央处理器（2）

第一讲 中央处理器概述

第二讲 单周期数据通路的设计

第三讲 单周期控制器的设计

第四讲 多周期处理器的设计

第五讲 流水线处理器设计

第六讲 流水线冒险及其处理

第七讲 高级流水线技术

第六讲 流水线冒险的处理

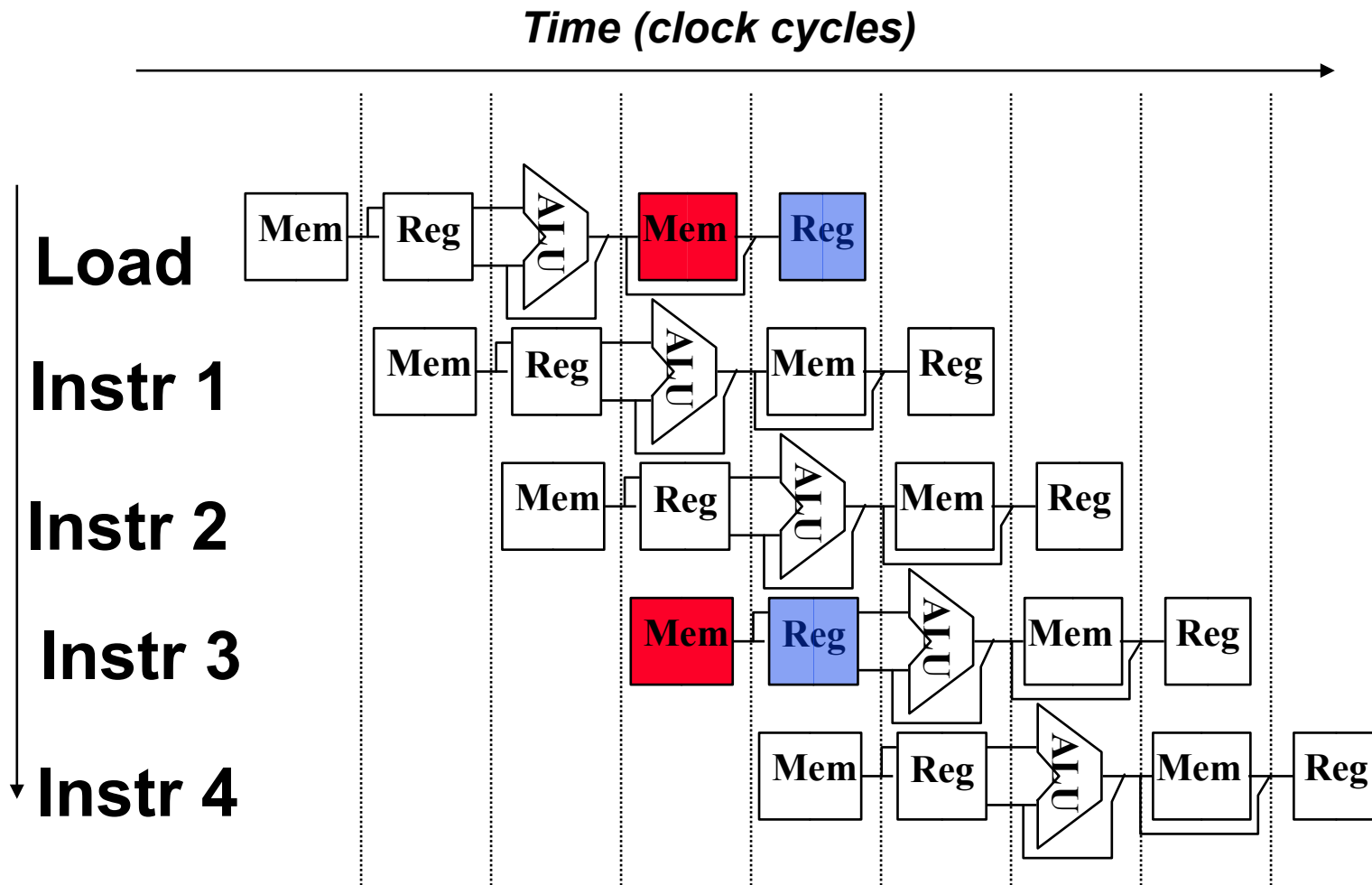
主要内容

- 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

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
 - 结构冒险Structural hazards (hardware resource conflicts):
现象: 同一个部件同时被不同指令所使用
 - 数据冒险Data hazards (data dependencies):
现象: 后面指令用到前面指令结果数据时, 前面指令的结果还没产生
 - 控制冒险Control (Branch) hazards (changes in program flow):
现象: 转移或异常改变执行流程, 后继指令在目标地址产生前已被取出

结构冒险现象



只有一个存储器时，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！

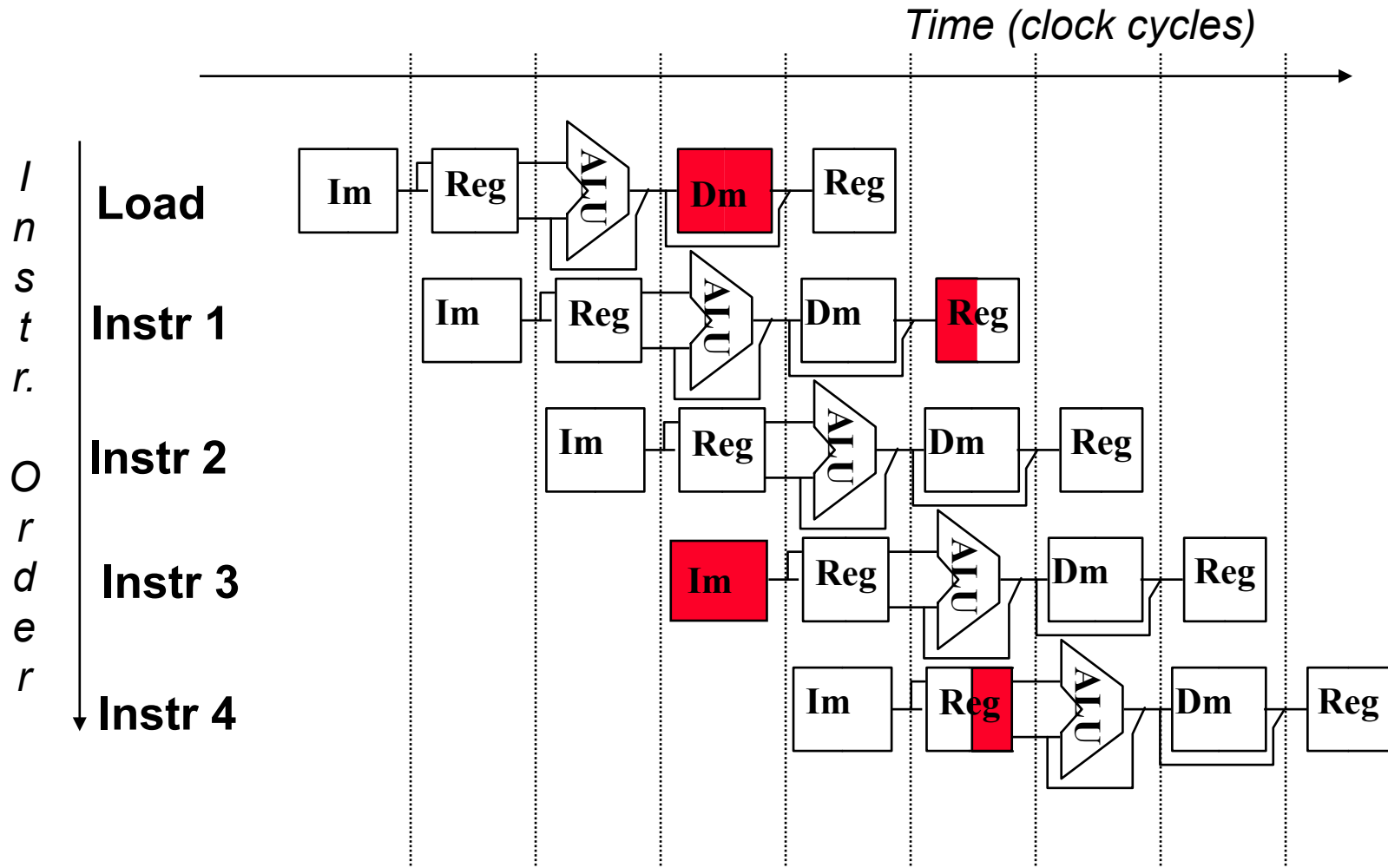
结构冒险也称硬件资源冲突：同一个执行部件被多条指令使用。

结构冒险的解决方法

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



数据冒险现象

以下指令序列中，寄存器r1会发生数据冒险

add r1, r2, r3

sub r4, r1, r3

and r6, r1, r7

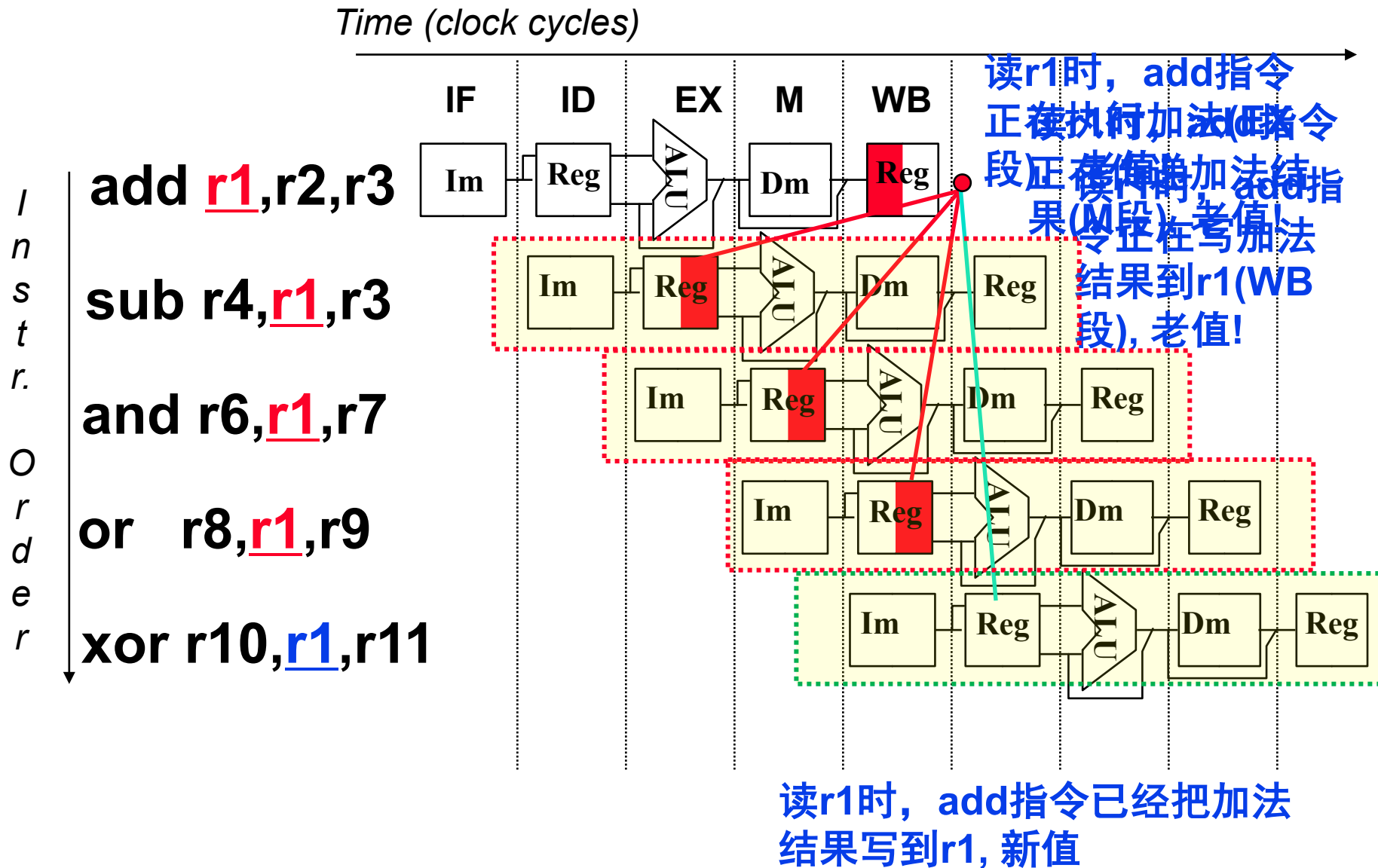
or r8, r1, r9

xor r10, r1, r11

想一下，哪条指令的r1是老的值？
哪条是新的值？

画出流水线图能很清楚理解！

关于r1 的数据冒险



数据冒险现象（小结）

所以——

最后一条指令的r1才是新的值！

如何解决这个问题？

add r1, r2, r3

sub r4, r1, r3

and r6, r1, r7

or r8, r1, r9

xor r10, r1, r11

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，乱序执行时会发生）

WAW: 写后写（基本流水线中不会发生，乱序执行时会发生）

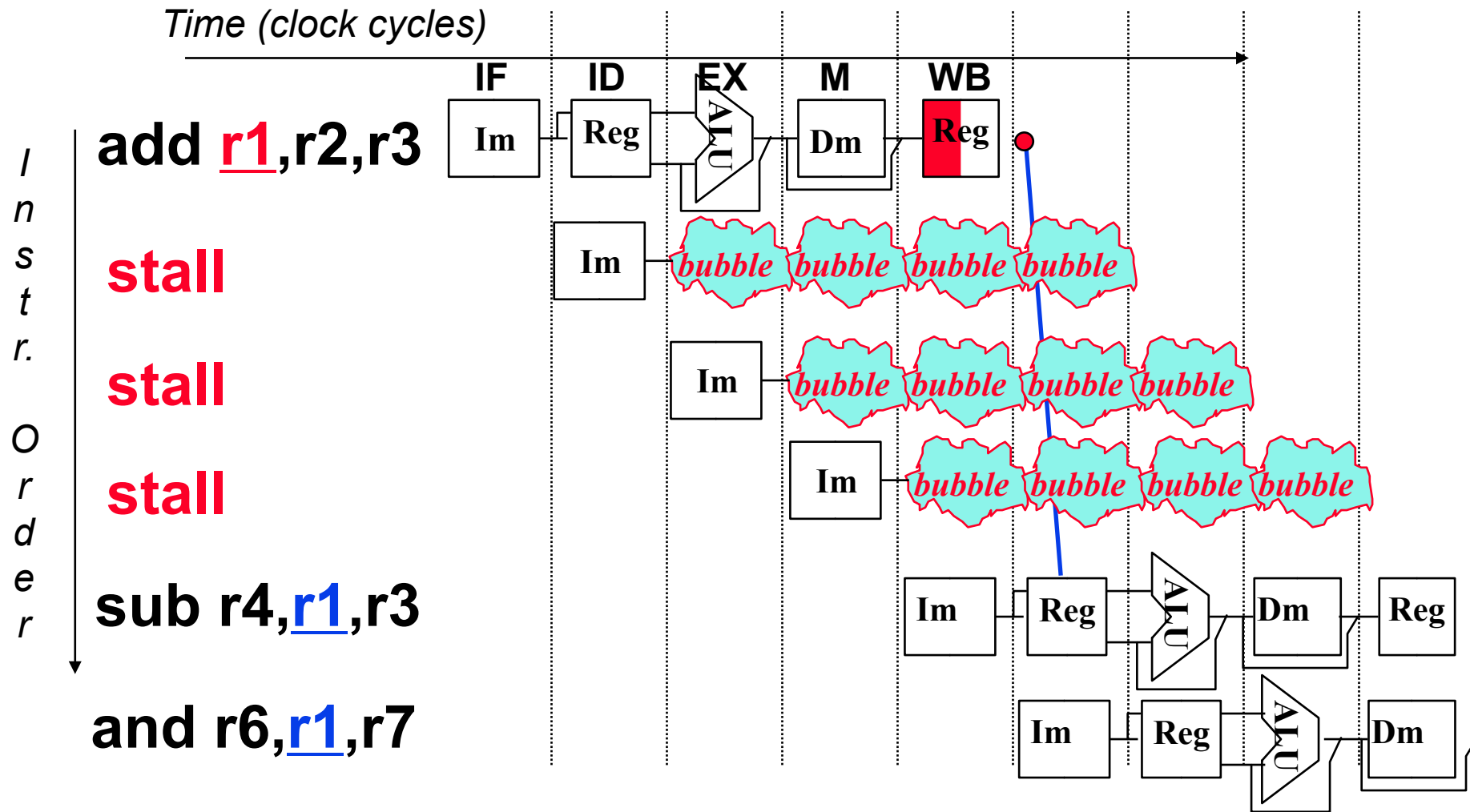
本讲介绍基本流水线，所以仅考虑RAW冒险

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：合理实现寄存器堆的读/写操作 (不能解决所有数据冒险)
- 方法4：转发 (Forwarding或Bypassing 旁路) 技术 (不能解决所有数据冒险)
- 方法5：编译优化：调整指令顺序 (不能解决所有数据冒险)

方案1: 在硬件上采取措施, 使相关指令延迟执行

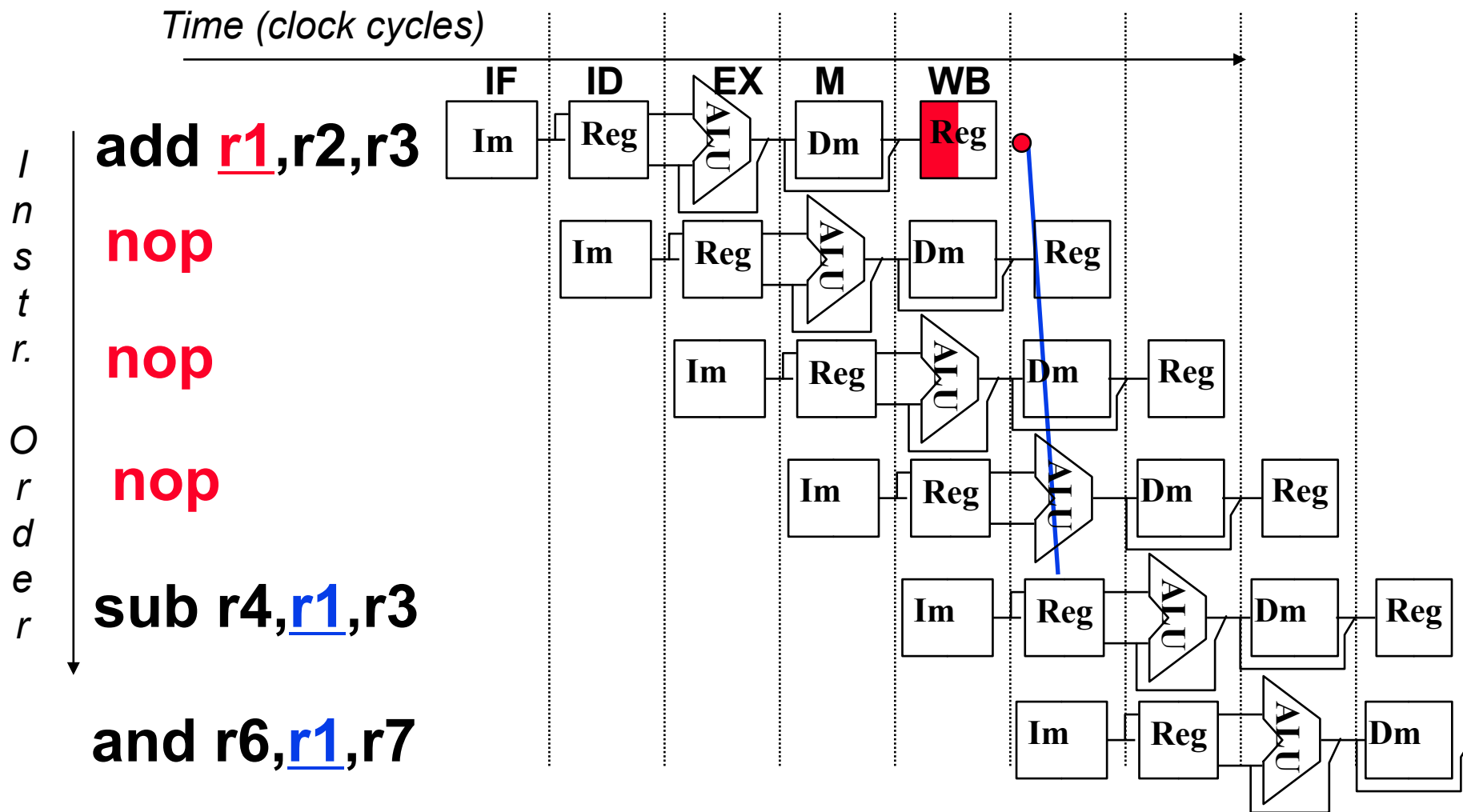
- 硬件上通过阻塞(stall)方式阻止后续指令执行, 延迟到有新值以后!
这种做法称为流水线阻塞, 也称为插入“气泡Bubble”



- 缺点: 控制比较复杂, 需要改数据通路; 指令被延迟三个时钟执行。

方案 2: 软件上插入无关指令

- 由编译器插入三条NOP指令，浪费三条指令的空间和时间。
好处：数据通路简单，即无需改数据通路。

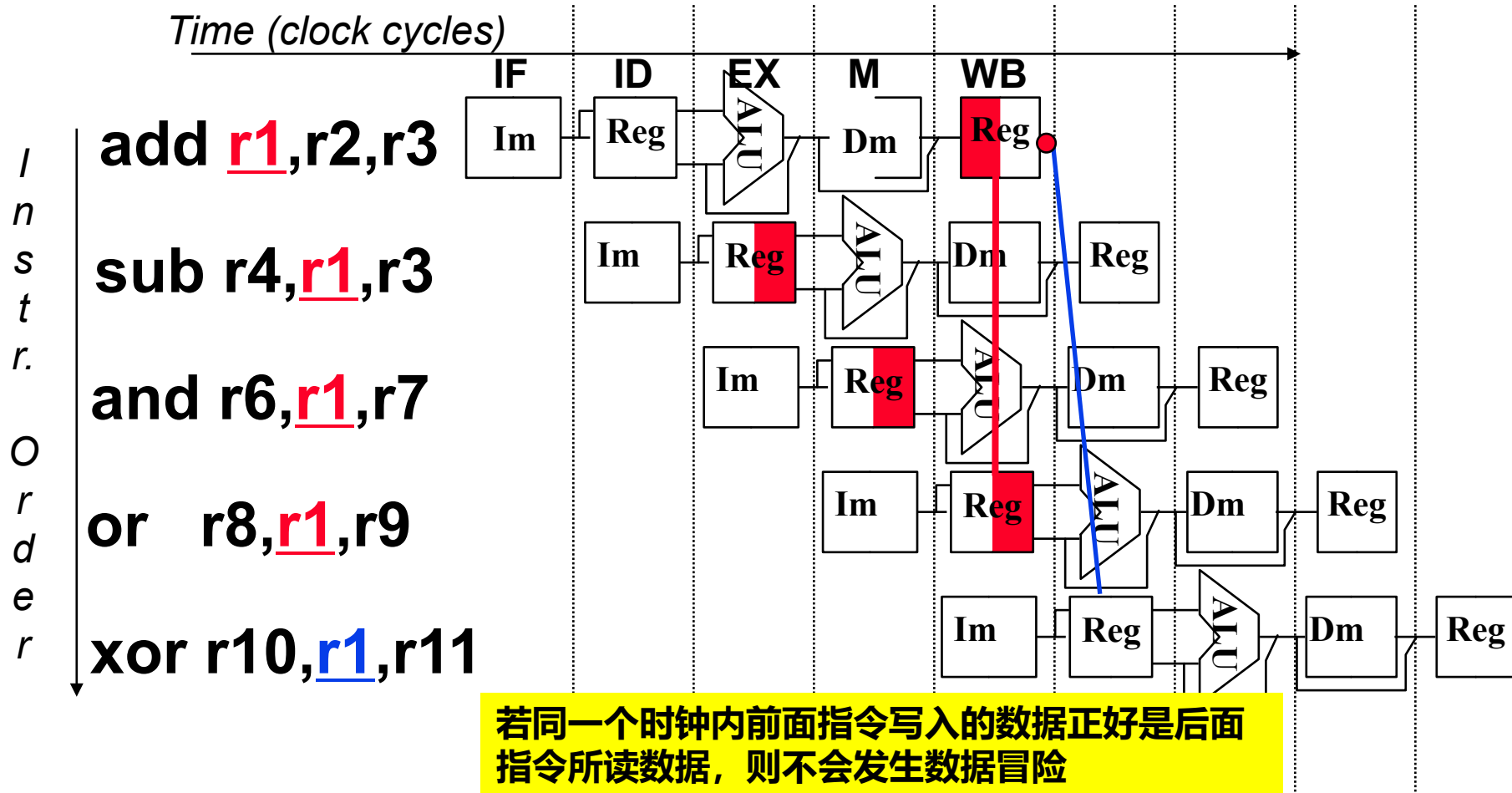


与方案1比，哪个更快？

一样，都是多三个时钟周期！

方案3: 同一周期内寄存器堆先写后读

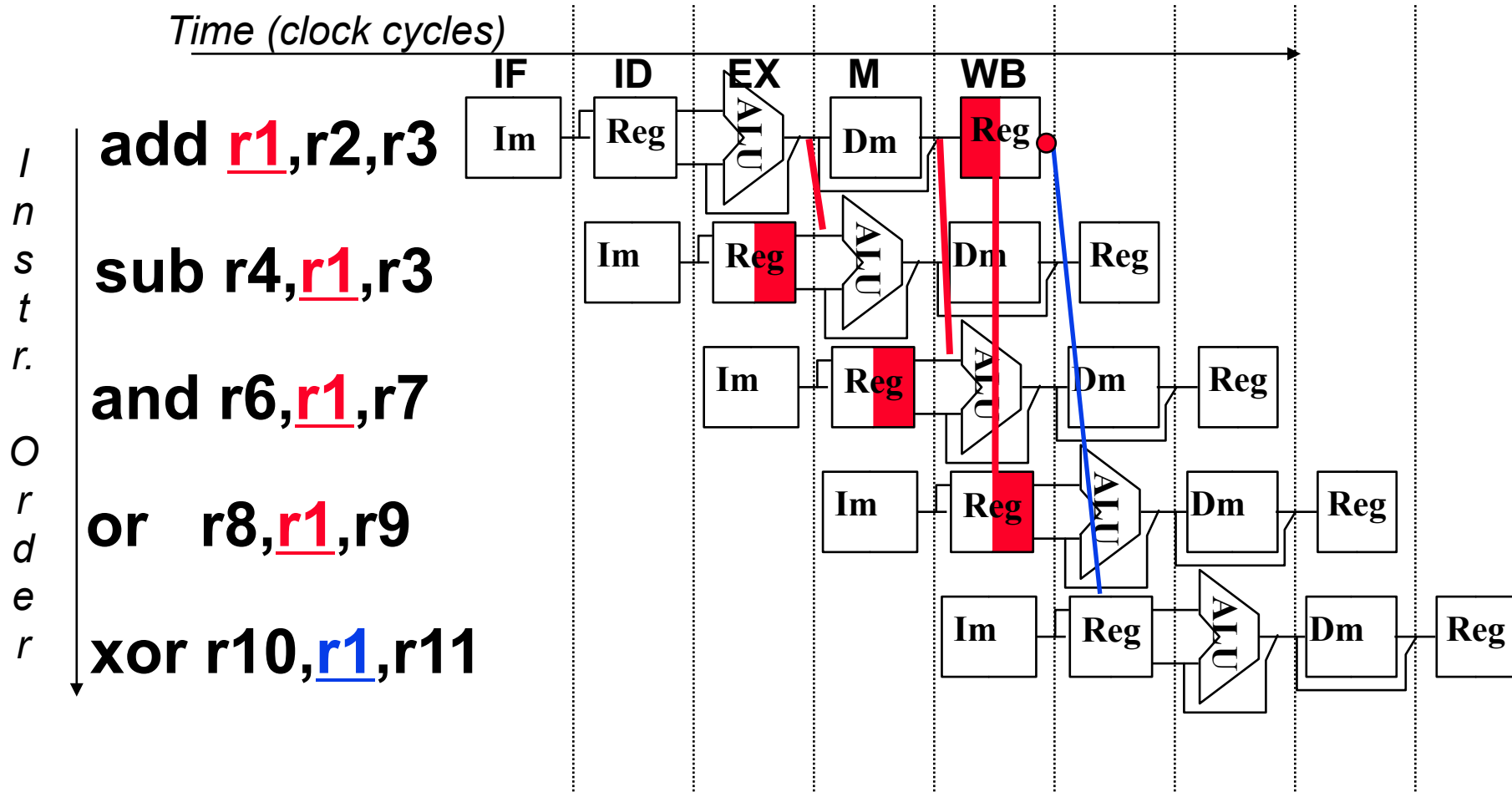
- 寄存器堆的读口和写口是相互独立的部件!



寄存器写口/读口分别在前/后半周期进行操作，使写入数据被直接读出
但是，只能解决部分数据冒险！

方案4: 利用DataPath中的中间数据: 转发

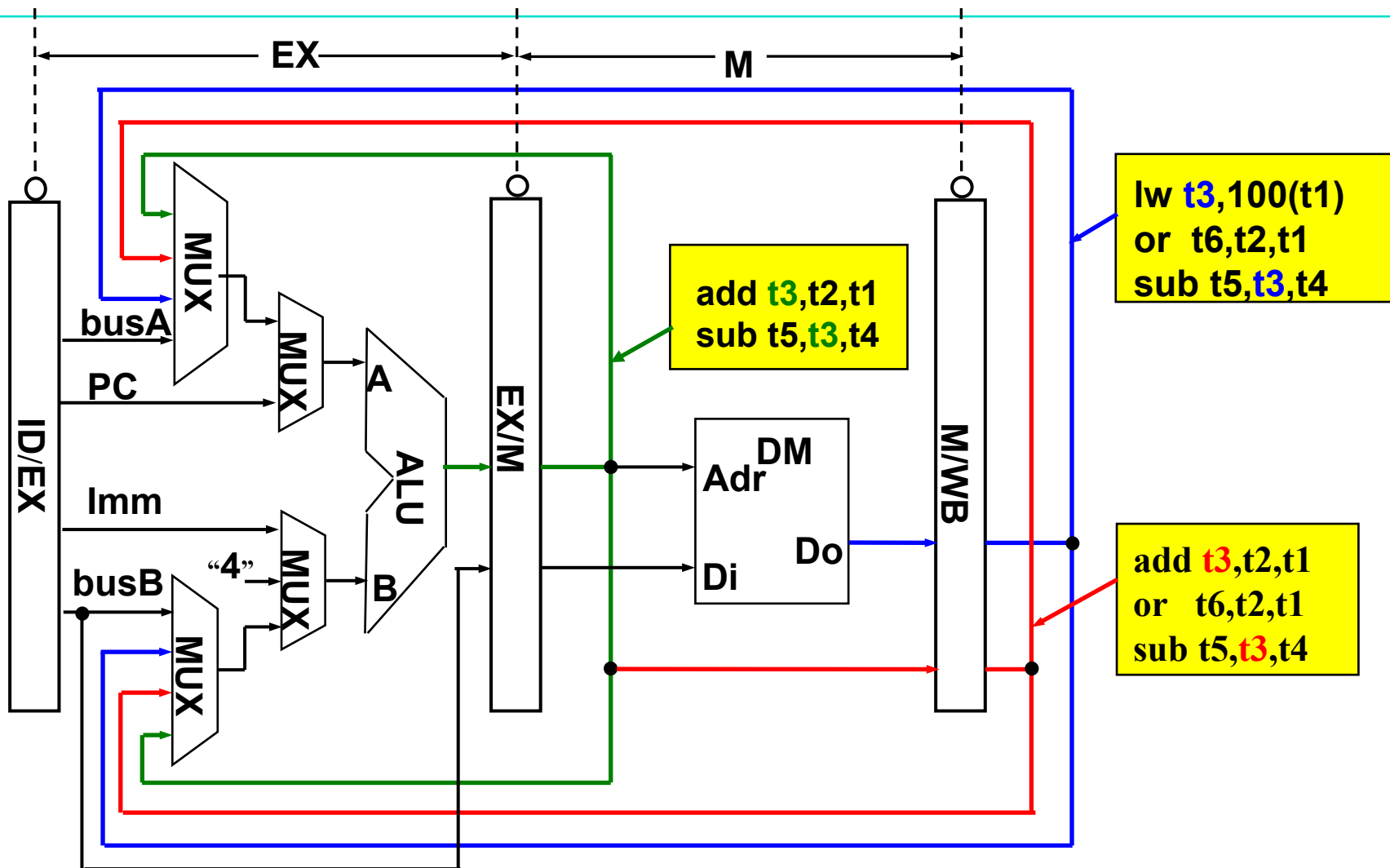
- 仔细观察后发现: 流水段寄存器中已有需要的值r1!



把数据从流水段寄存器中直接取到ALU的输入端

称为转发 (Forwarding)
或旁路 (Bypassing)

硬件上的改动以支持“转发”技术——四种情况



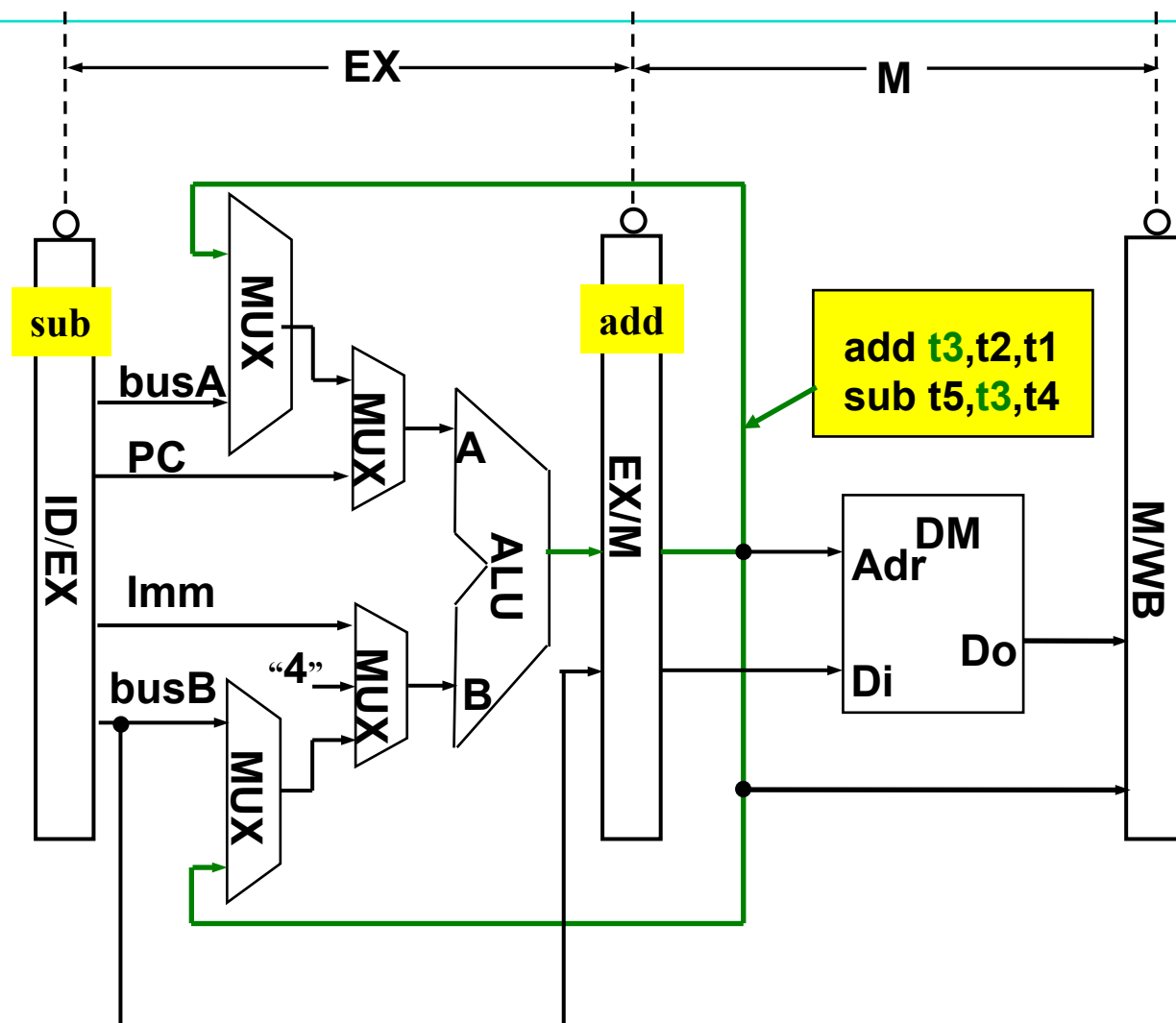
如果指令序列为

lw t3, 100(t1)
or t6, t3, t1
sub t5, t3, t4

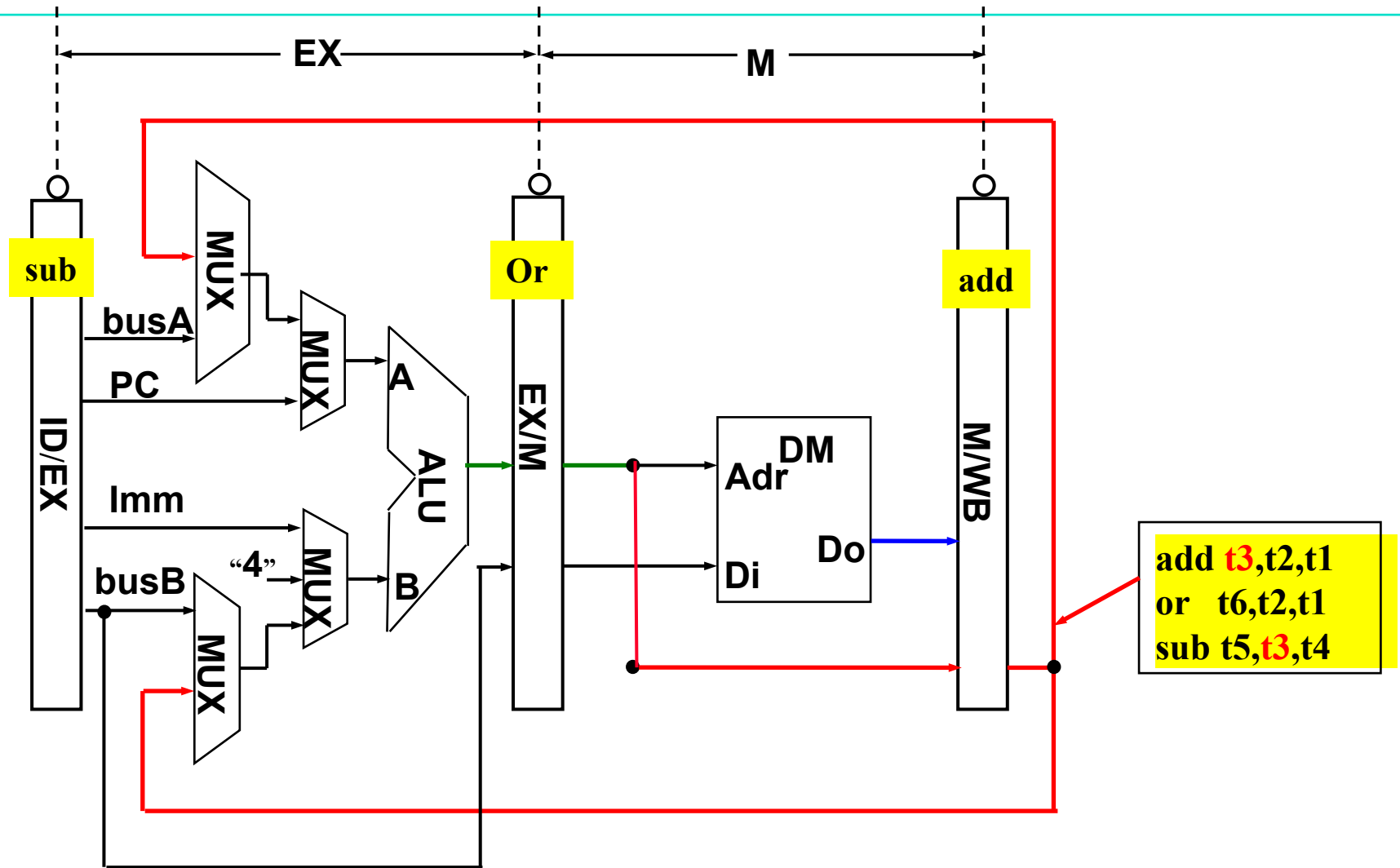
能用“转发”技术解决第1、2两条指令间的数据冒险吗？

接下来展开分析。。

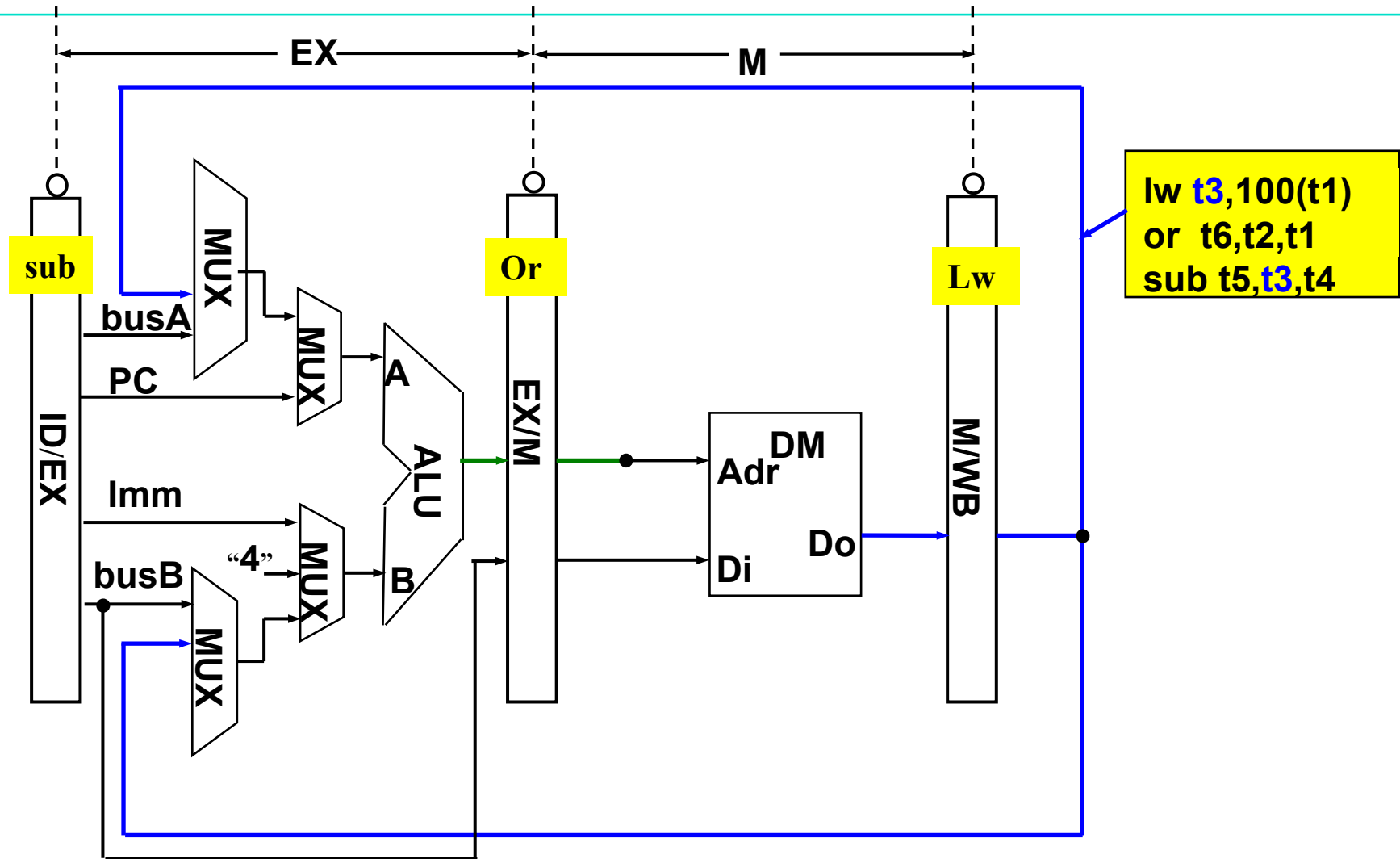
硬件上的改动以支持“转发”技术（续1）



硬件上的改动以支持“转发”技术（续2）

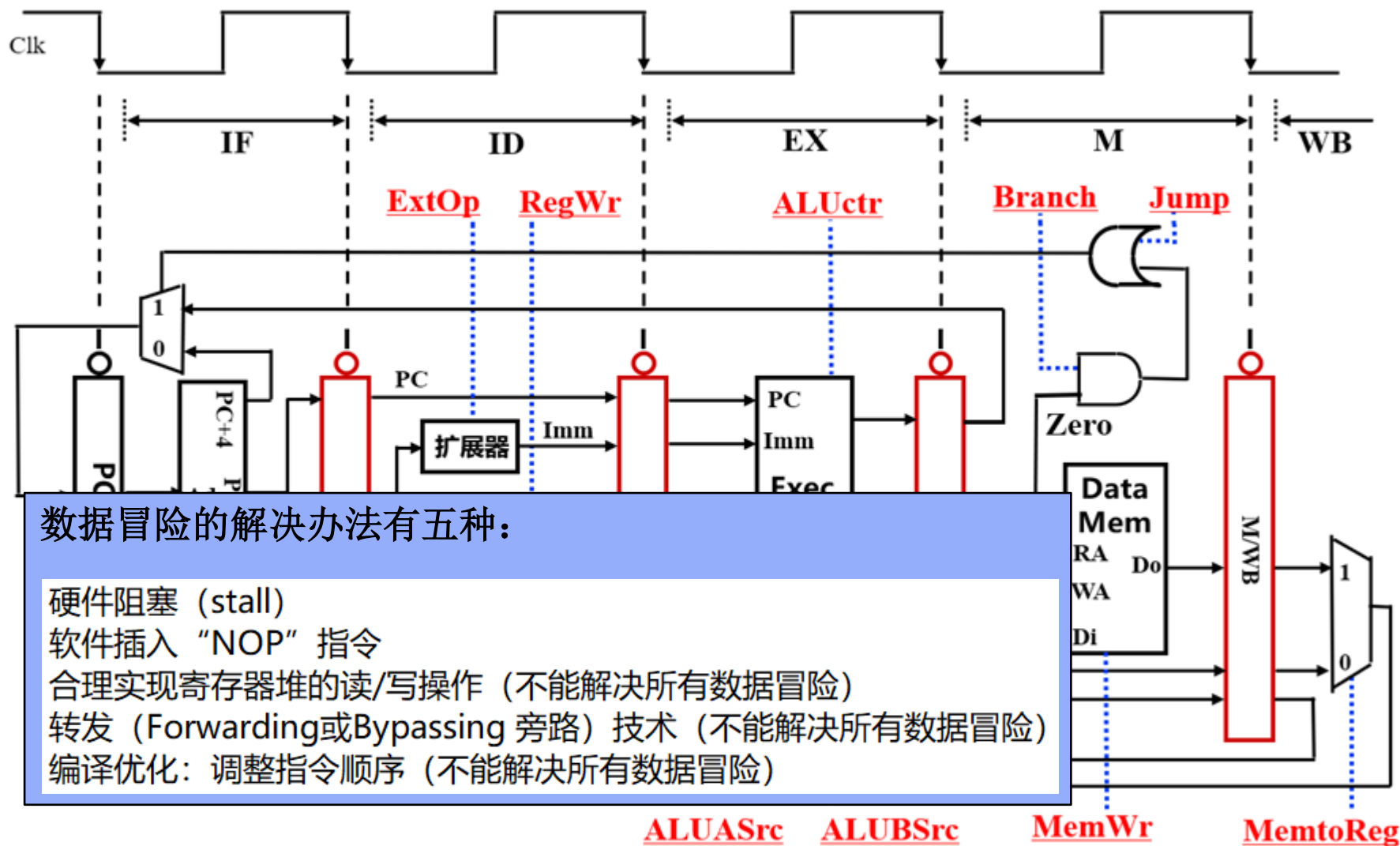


硬件上的改动以支持“转发”技术（续3）

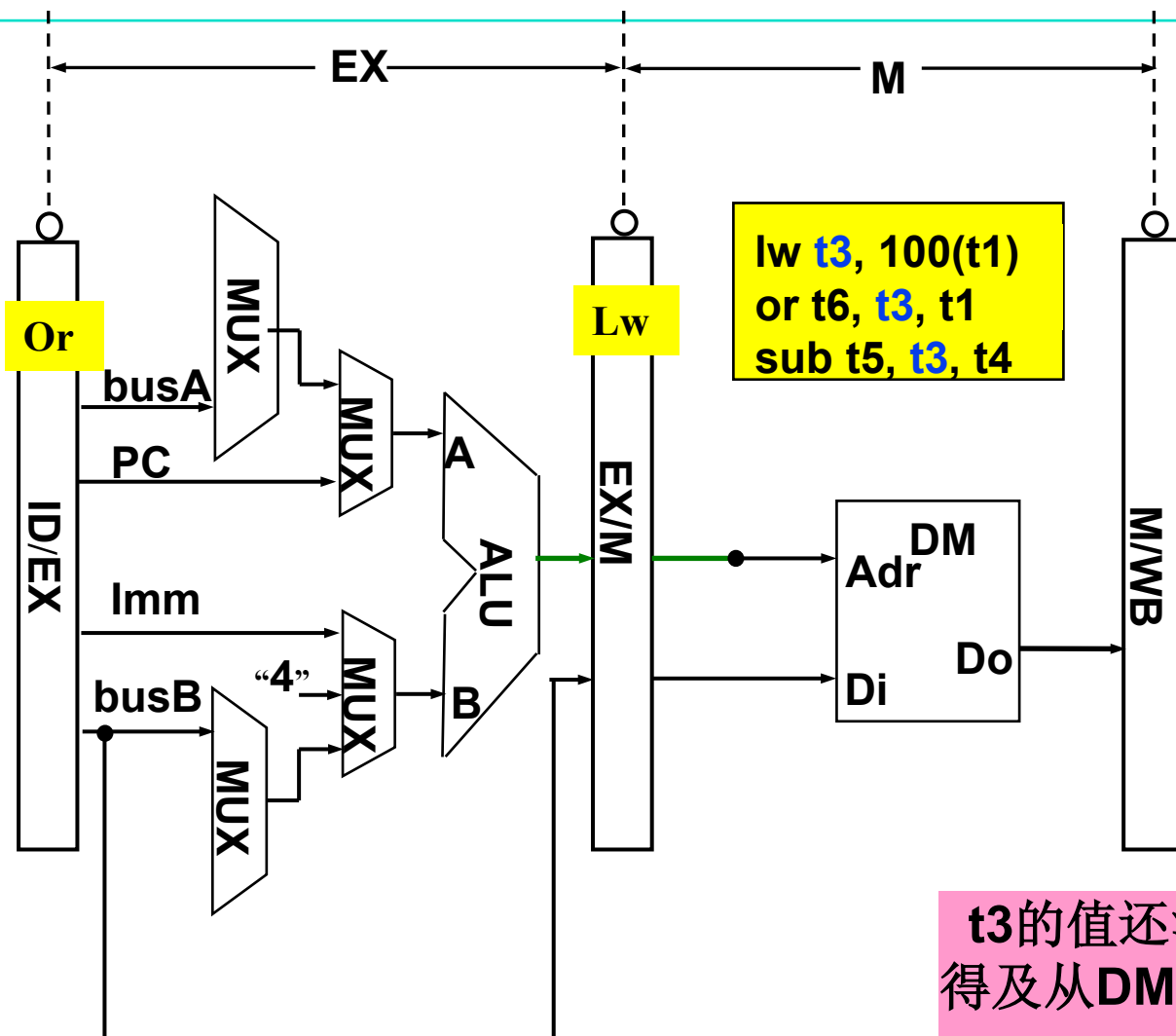


回顾第24次课

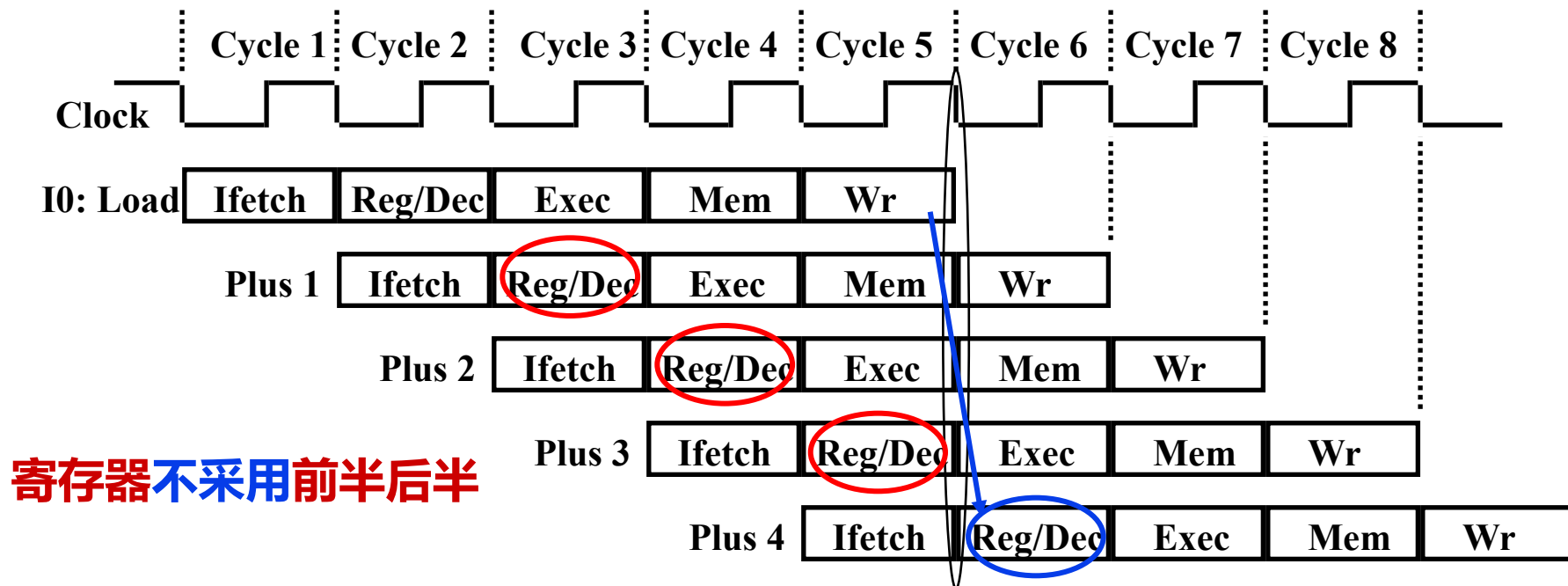
控制信号如何产生和传递？
为什么会产生冒险？



无法“转发”的情况（续4）——只和load有关



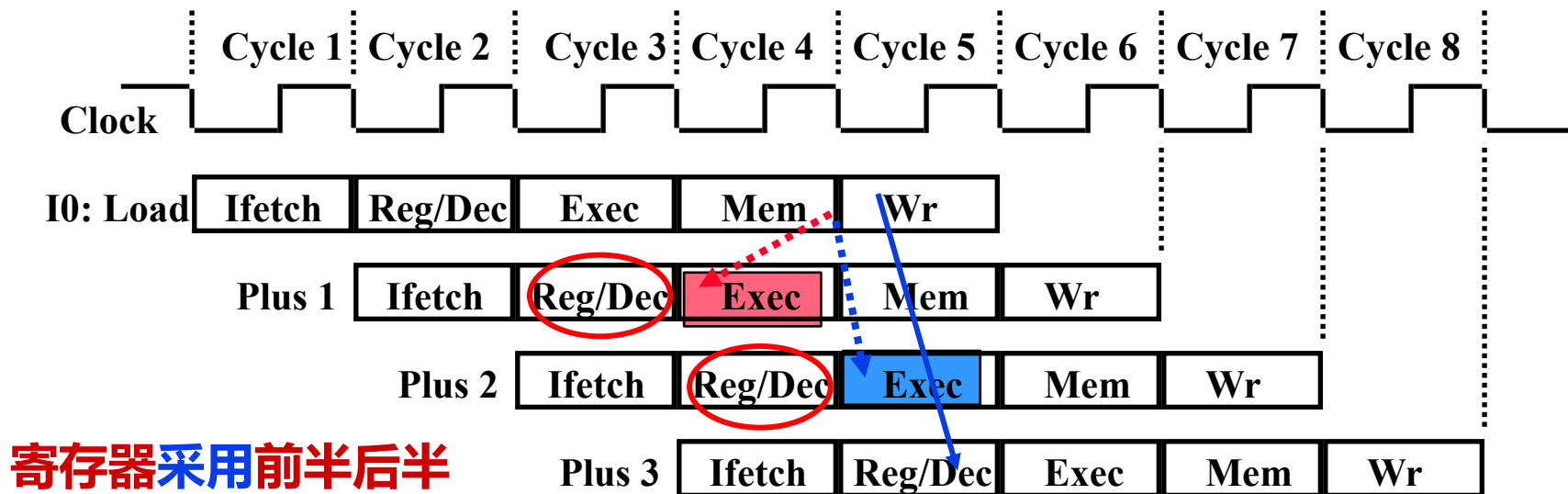
回顾: Load指令引起的延迟现象



寄存器采用前半后半

若不采用转发, 在何时才能使用Load指令的结果?

回顾: Load指令引起的延迟现象 (续)



- 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?

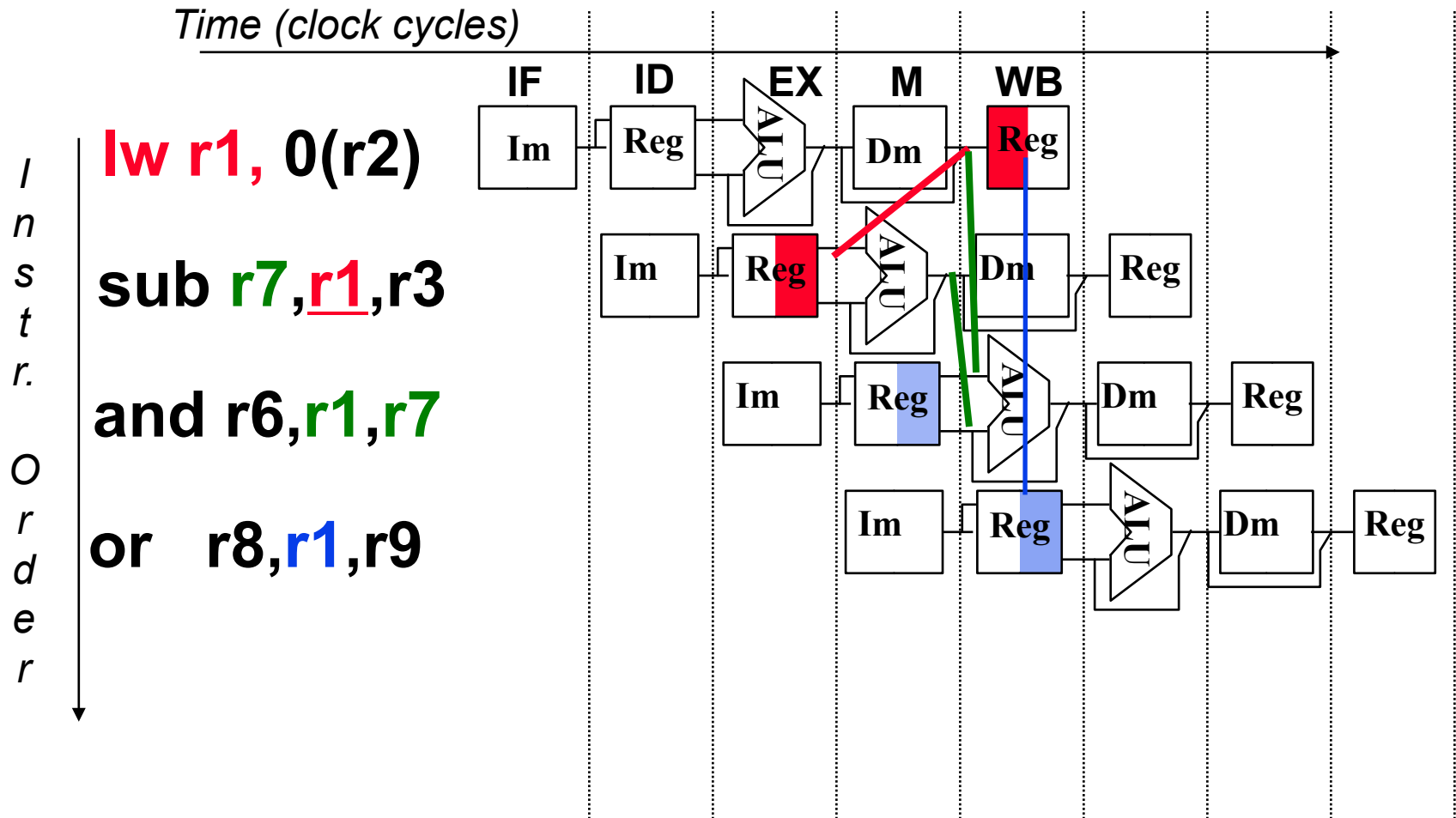
实际上, 在第四周期结束时, 数据在流水段寄存器中已经有值。

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)”

“Forwarding”技术使Load-use冒险只需延迟一个周期

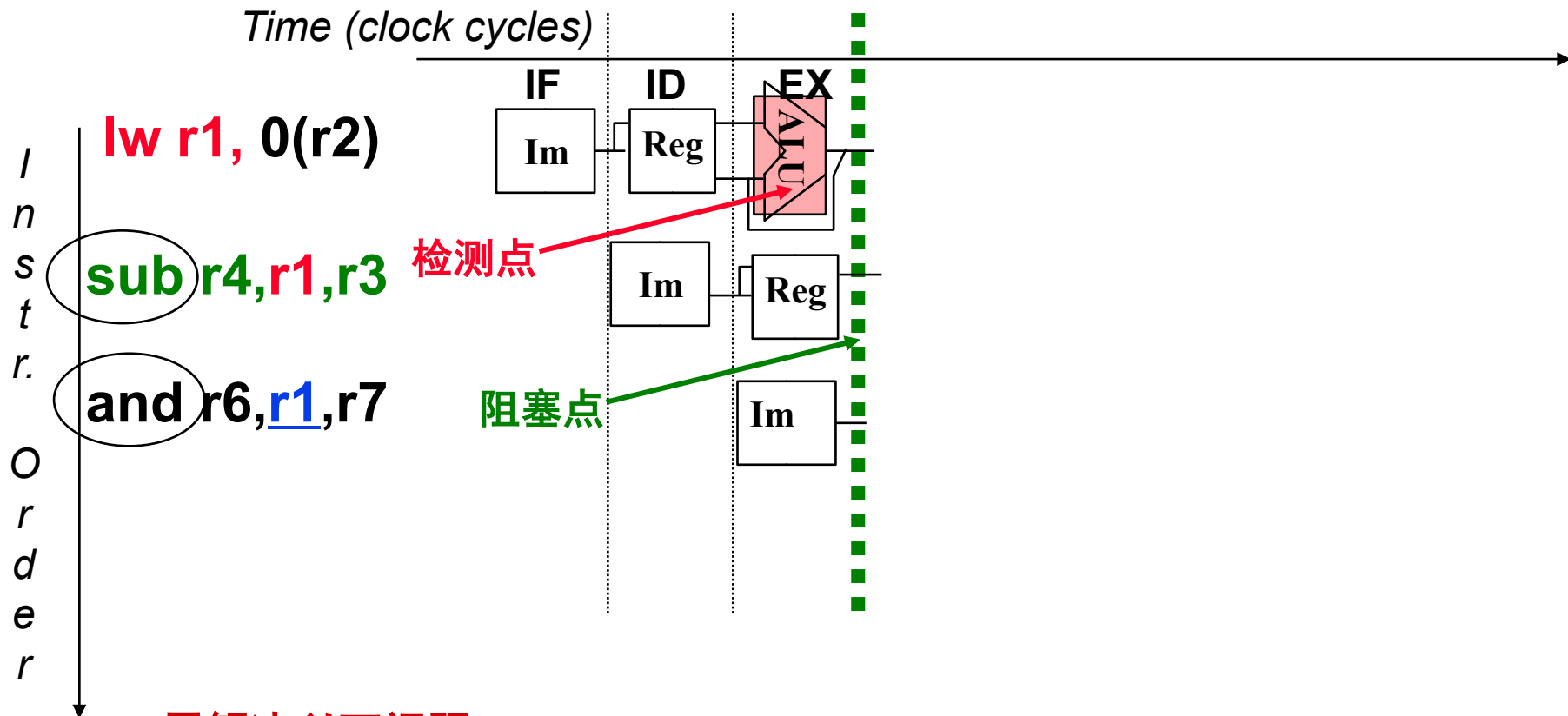


采用“转发”后仅第二条指令 SUB r7,r1,r3 不能按时执行!

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!

数据冒险处理最佳方案：“转发” + “Load-use阻塞”

包括load-use数据冒险的解决方式——转发+ 阻塞



需解决以下问题：

(1) 判断什么条件下需要阻塞

(2) 修改数据通路来实现阻塞

ID/EX.MemRead

&& (ID/EX.Rd==IF/ID.Rs1

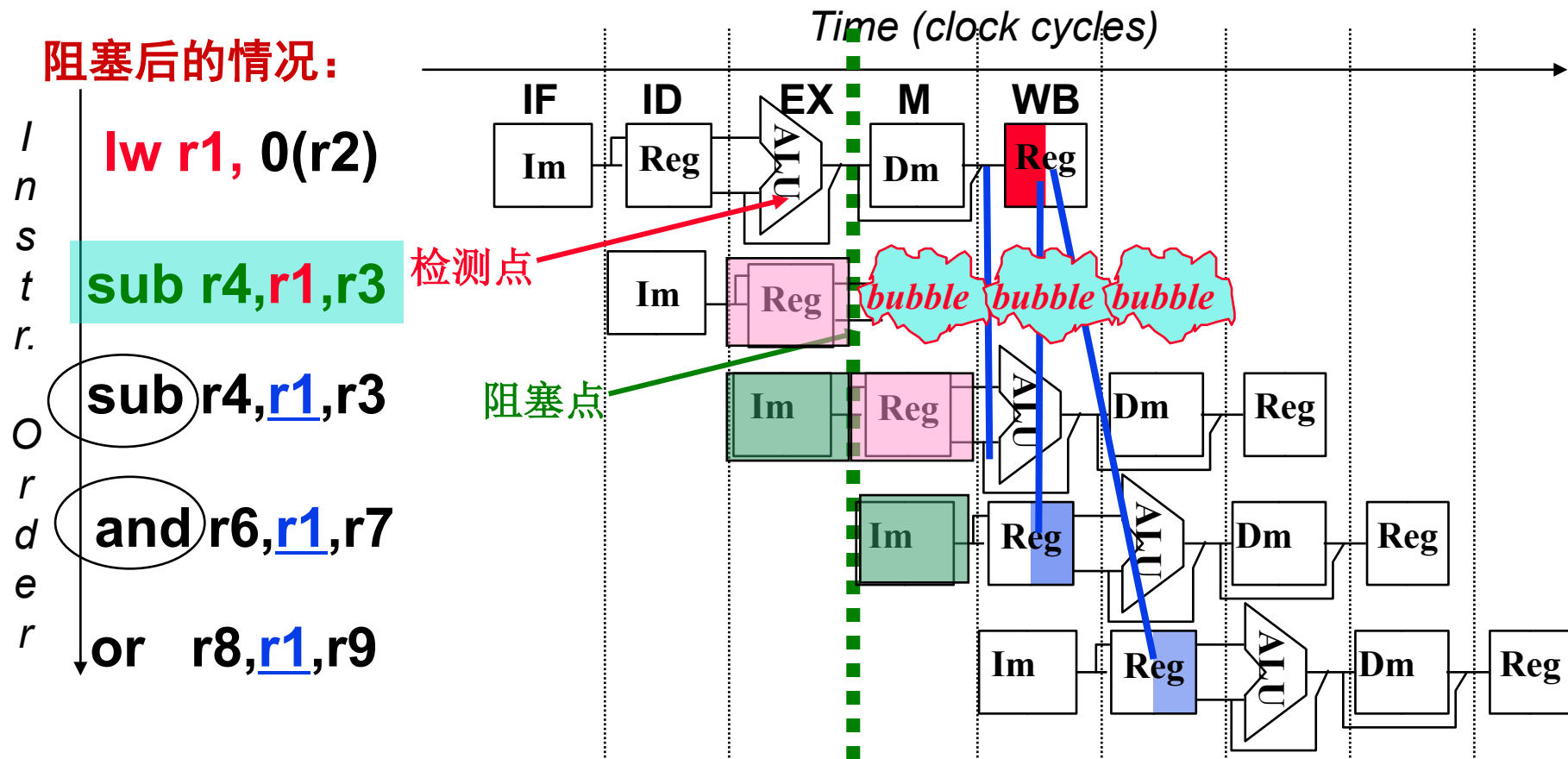
|| ID/EX.Rd==IF/ID.Rs2)

前面指令为Load 并且

前面指令的目的寄存器等于当前刚取出指令的源寄存器

包括load-use数据冒险的解决方式——转发+ 阻塞

阻塞后的情况：

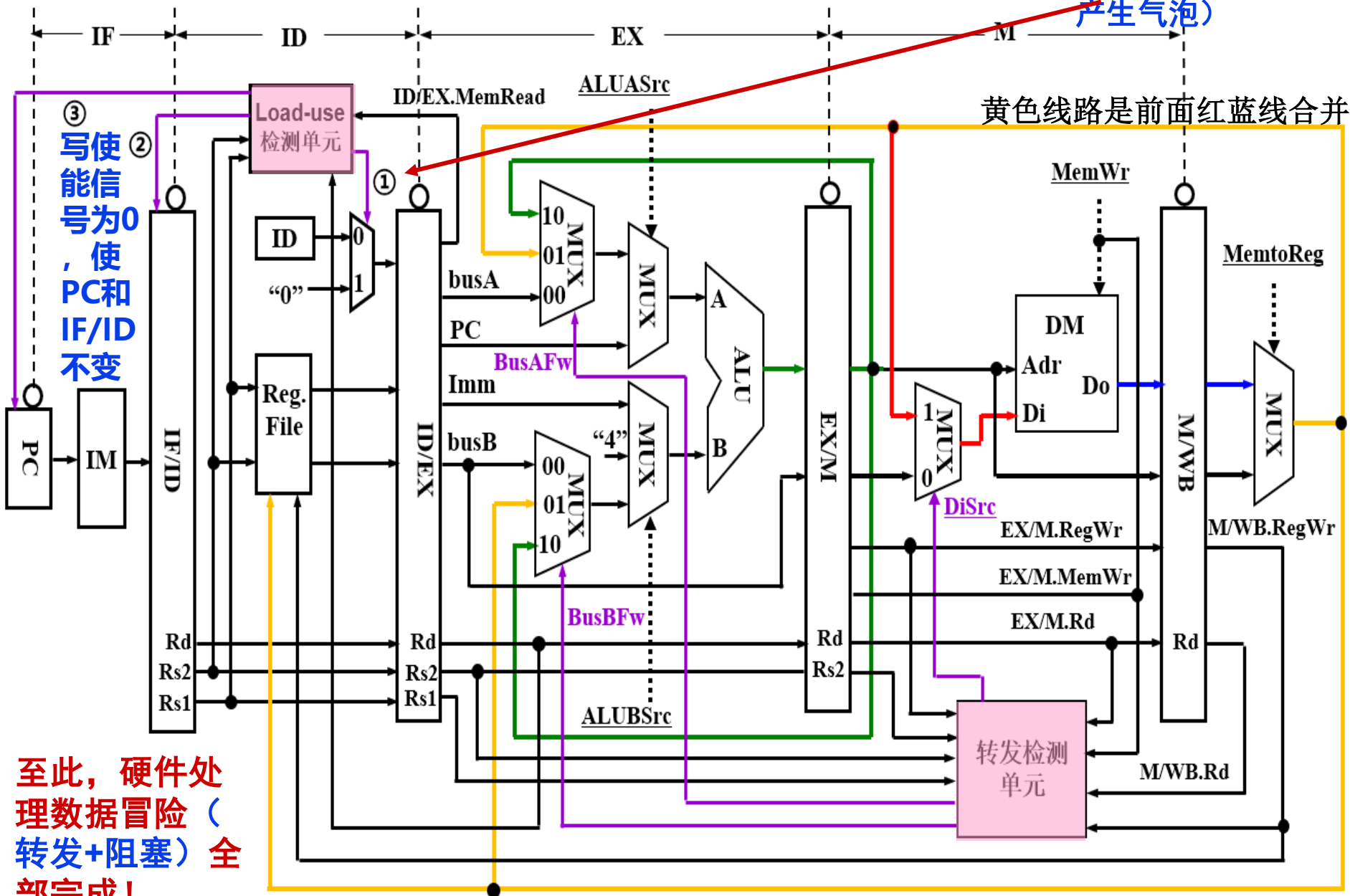


在阻塞点，将load后面两条指令的执行结果清除，并延迟一个周期执行

- ① 将ID/EX段寄存器中所有控制信号清0，以插入一个“气泡”
- ② IF/ID寄存器中的信息不变（还是sub指令），sub指令将重新译码执行
- ③ PC中的值不变（还是and指令地址），and指令重新被取出执行

带“转发”和“阻塞”检测的流水线数据通路

使控制信号清0，
阻塞指令执行！（
产生气泡）



方案5：编译器进行指令顺序调整来解决数据冒险

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

编译器的优化很重要！

优化调度后load阻塞现象大约可以降低1/2~1/3

Slow code: (需阻塞2个时钟)

```
lw    t2, b
lw    t3, c
add   t1, t2, t3
sw    t1, a
lw    t5, e
lw    t6, f
sub   t4, t5, t6
sw    t4, d
```

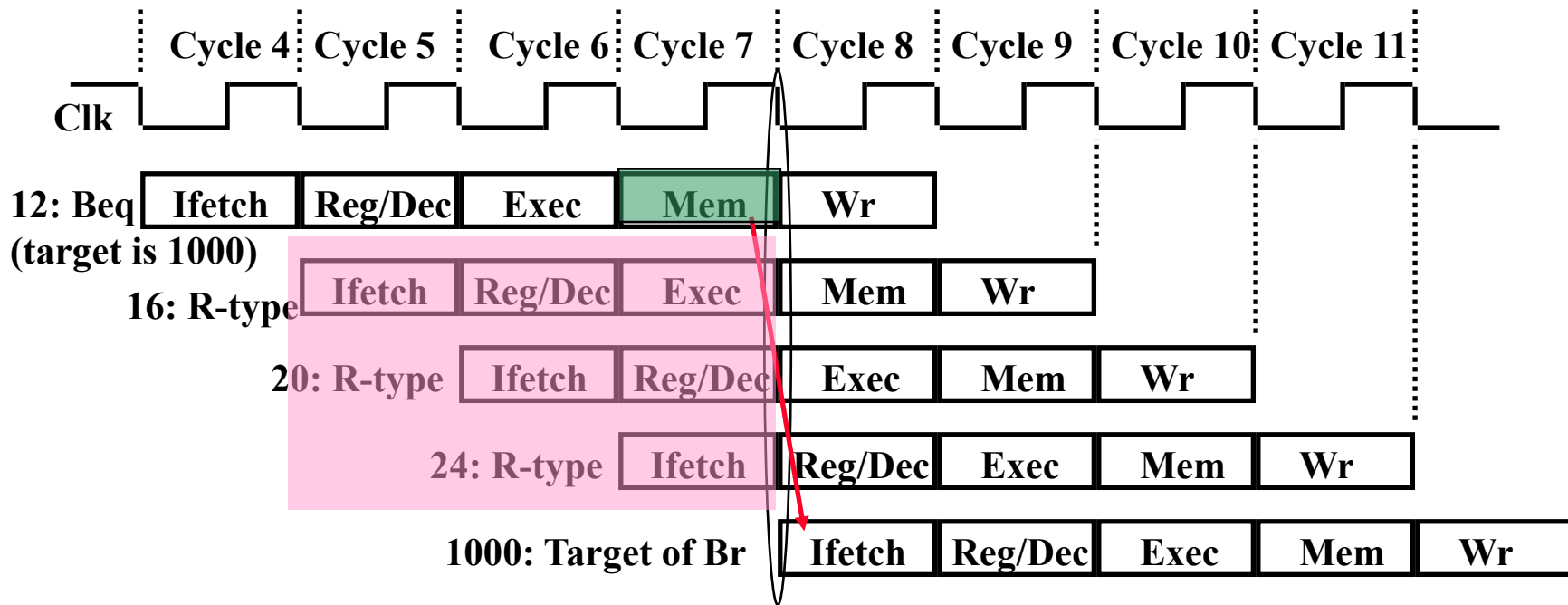
调整后

Fast code: (无需阻塞)

```
lw    t2, b
lw    t3, c
lw    t5, e
add   t1, t2, t3
lw    t6, f
sw    t1, a
sub   t4, t5, t6
sw    t4, d
```

如果硬件不支持阻塞处理的话，则编译器可以将顺序调整和插入NOP指令结合起来，在找不到可插入的指令时，插入NOP指令！

控制冒险 现象——延迟损失时间片C



虽然Beq指令在第四周期取出，但：

- “是否转移”在M阶段确定，目标地址在第七周期才被送到PC输入端
- 第八周期才取出目标地址处的指令执行

结果：在取目标指令之前，已有三条指令被取出，取错了三条指令！

- 发生转移时，要在流水线中清除Beq后面的三条指令，分别在EX、ID、IF段中
- 延迟损失时间片C**：发生转移时，给流水线带来的延迟损失

这里 C=3

Control Hazard的解决方法

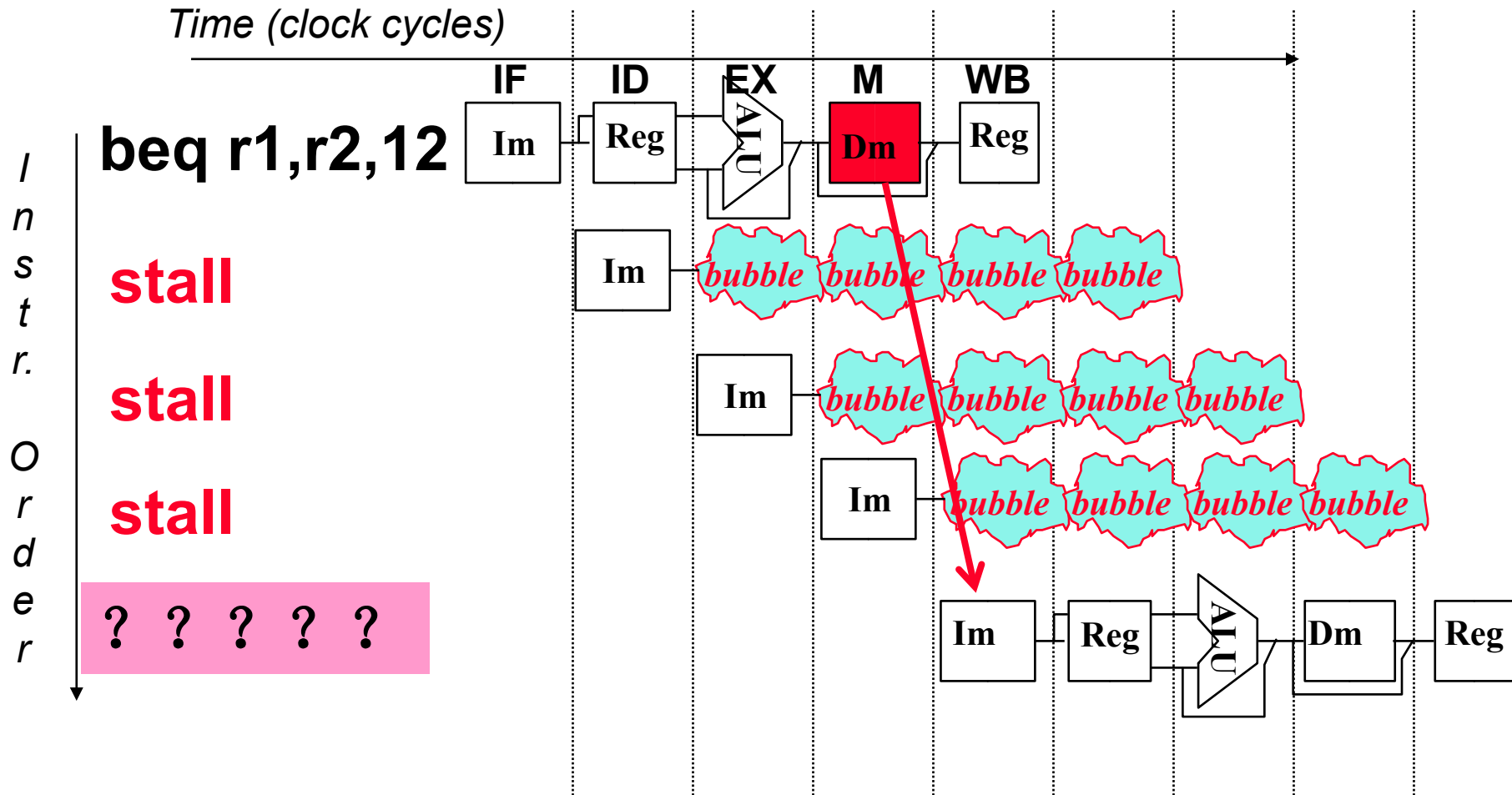
- 方法1：硬件上阻塞 (stall) 分支指令后三条指令的执行
- 方法2：软件上插入三条 “NOP” 指令
(以上两种方法的效率太低，需结合分支预测进行)
- 方法3：分支预测 (Predict)
 - 简单 (静态) 预测:
 - 总是预测条件不满足(not taken)，即：不跳转
 - 动态预测:
 - 根据程序执行的历史情况进行动态预测调整

注：流水线控制必须确保被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

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, 延迟到分支指令能够确定是否转移以后! ——使接下来三条指令清0或 其操作信号清0, 以插入气泡。

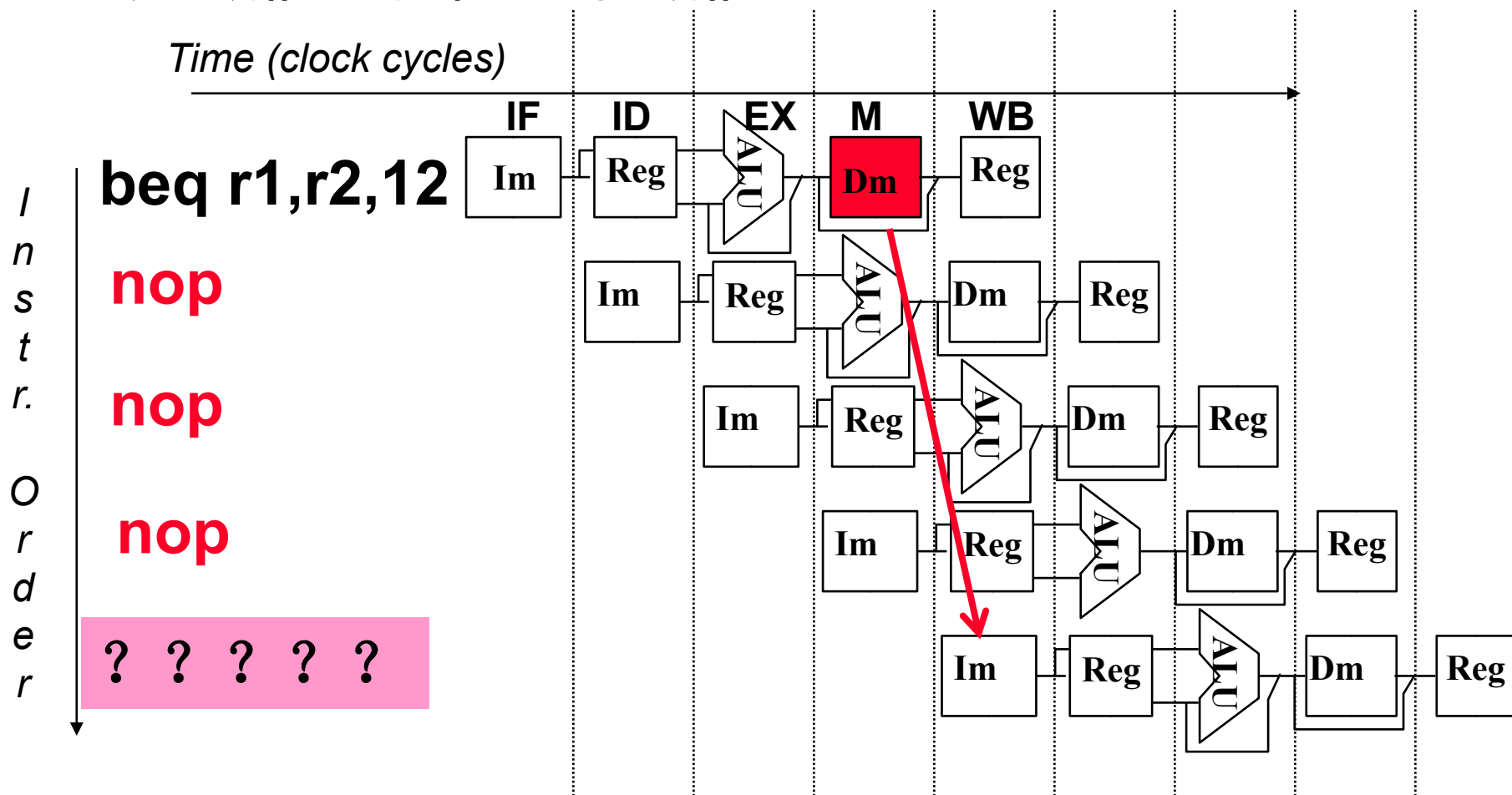


- 缺点: 控制比较复杂, 需要改数据通路; 指令被延迟三个时钟执行。

方案 2: 软件上插入无关指令

• 由编译器插入三条NOP指令，延迟到分支指令能够确定是否转移有新值以后再执行后续有效的指令！浪费三条指令的空间和时间。

好处：数据通路简单，即无需改数据通路。



与方案1比，哪个更快？

一样，都是多三个时钟周期！

简单（静态）分支预测方法

基本做法

- 总预测条件不满足(not taken), 即: 不跳转
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: 循环顶部(底部)分支总是预测为不满足(满足)。能达65%-85%的预测准确率
- 预测失败时, 需把流水线中三条错误预测指令 ($C=3$) 丢弃掉
 - 将被丢弃指令的控制信号值或指令设置为0
(注: 涉及到当时在IF、ID和EX三个阶段的指令)

性能

- 如果转移概率是50%, 则预测正确率仅有50%

预测错误的代价 ($C=3, 2, 1?$)

- 预测错误的代价与何时能确定是否转移有关。越早确定代价越少
- 是否可以把“是否转移”的确定工作提前, 而不等MEM阶段才确定?
可以!

那最早可以提前到哪个阶段呢?

简单（静态）分支预测方法

- 缩短分支延迟，减少错误预测代价
 - 可以将“转移地址计算”和“分支条件判断”操作调整到ID阶段来缩短延迟
 - 将转移地址生成从M阶段移到ID阶段，可以吗？为什么？
(可：IF/ID流水段寄存器中已有PC的值，ID段已有扩展后立即数Imm)
 - 将“判0”操作从EX阶段移到ID阶段，可以吗？为什么？
(用逻辑运算(如，先按位异或，再结果各位相或)直接比较Rs1和Rs2的值)
(简单判断用逻辑运算，复杂判断可以用专门指令生成条件码)
(许多条件判断都很简单)
- 预测错误的检测和处理（称为“冲刷、冲洗” -- Flush）
 - 如果原来预测不转移
 - 但是发现Branch=1并且Zero=1时，则beq预测失败
 - 此时需要完成以下两件事（延迟损失时间片C=1时）：
 - ① 将转移目标地址->PC
 - ② 清除IF段中取出的指令，即：将IF/ID中的指令字清0，转变为nop指令

原来要清除三条指令（C=3），调整后只需要清除一条指令（C=1），因而只延迟一个时钟周期，每次预测错误减少了两个周期的代价！

动态分支预测方法

- 简单的静态分支预测方法的预测成功率不高，应考虑动态预测
- 动态预测基本思想：
 - 利用最近转移发生的情况，来预测下一次可能转移还是不转移
 - 根据实际情况来调整预测，
 - 转移发生的历史情况记录在BHT中（有多个不同的名称）
 - 分支历史记录表BHT (Branch History Table)
 - 分支预测缓冲BPB (Branch Prediction Buffer)
 - 分支目标缓冲BTB (Branch Target Buffer)

分支历史记录表BHT（或BTB、BPB）



动态分支预测方法（续）

- 简单的静态分支预测方法的预测成功率不高，应考虑动态预测
- 动态预测基本思想：
 - 利用最近转移发生的情况，来预测下一次可能转移还是不转移
 - 根据实际情况来调整预测，
 - 转移发生的历史情况记录在BHT中（有多个不同的名称）
 - **每个表项由分支指令地址低位作索引，故在IF阶段就可以取到预测位**
 - **低位地址相同的分支指令共享一个表项，所以，可能取的是其他分支指令的预测位。会不会有问题？**
 - **由于仅用于预测，所以不影响执行结果**

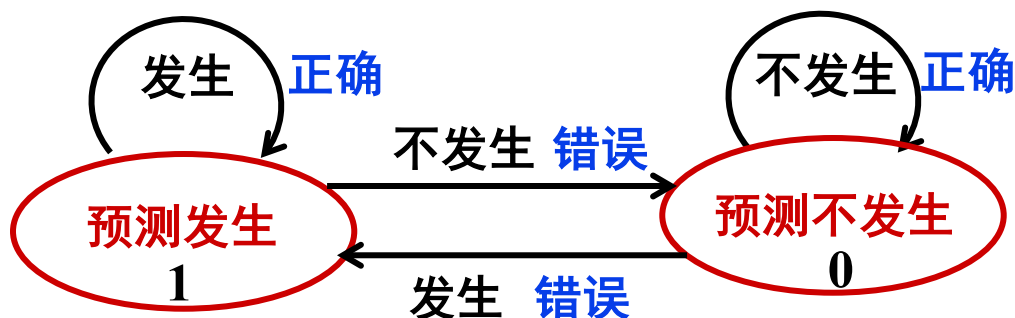
现在几乎所有的处理器都采用动态预测（dynamic predictor）

动态预测基本方法

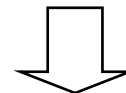
- 按照预测状态图进行预测和调整
- 采用一位预测位：总是按上次实际发生的情况来预测下次
 - 1表示最近一次发生过转移（taken），0表示未发生（not taken）
 - 缺点：当连续两次的分支情况发生改变时，预测错误
- 采用二位预测位
 - 用2位组合四种情况来表示预测和实际转移情况
 - 在连续两次分支发生不同时，只会有一次预测错误

采用较多的是二位或二位以上预测位。如：Pentium 4的BTB2采用4位预测位

一位预测状态图



```
Loop:  g = g + A[i];
        i = i + j;
        if (i != h) go to Loop:
Assuming variables g, h, i, j ~
t1, t2, t3, t4 and base address
of array is in t5
```



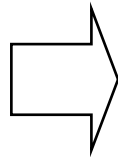
```
Loop: add t7, t3, t3      ; i*2
      add t7, t7, t7      ; i*4
      add t7, t7, t5
      lw  t6, 0(t7)       ; t6=A[i]
      add t1, t1, t6       ; g= g+A[i]
      add t3, t3, t4
      bne t3, t2, Loop
      ... ..
```

- 指令预取时，按照预测位读取相应分支的指令
 - 预测发生时，选择“转移取”
 - 预测不发生时，选择“顺序取”
- 指令执行时，按实际执行结果修改预测位
 - 对照状态转换图来进行修改
 - 例如：对于一个循环分支
 - 若初始状态为0(再次循环时为0)，则第一次和最后一次都错
 - 若初始状态为1，则只有最后一次会错。(再次循环时又改为0，还是有两次错)

即：只要本次和上次的发生情况不同，就会出现一次预测错误。

举例：双重循环的一位动态预测

```
into sum (int N)
{
  int i, j, sum=0;
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      sum=sum+1;
  return sum;
}
```



```
... ..
Loop-i: beq t1,a0, exit-i  # 若( i=N)则跳出外循环
        add t2, zero, zero #j=0
Loop-j: beq t2, a0, exit-j  # 若(j=N)则跳出内循环
        addi t2, t2, 1      # j=j+1
        addi t0, t0, 1      #sum=sum+1
        j Loop-j
exit-j:  addi t1, t1, 1      # i=i +1
        j Loop-i
exit-i:  ... ..
```

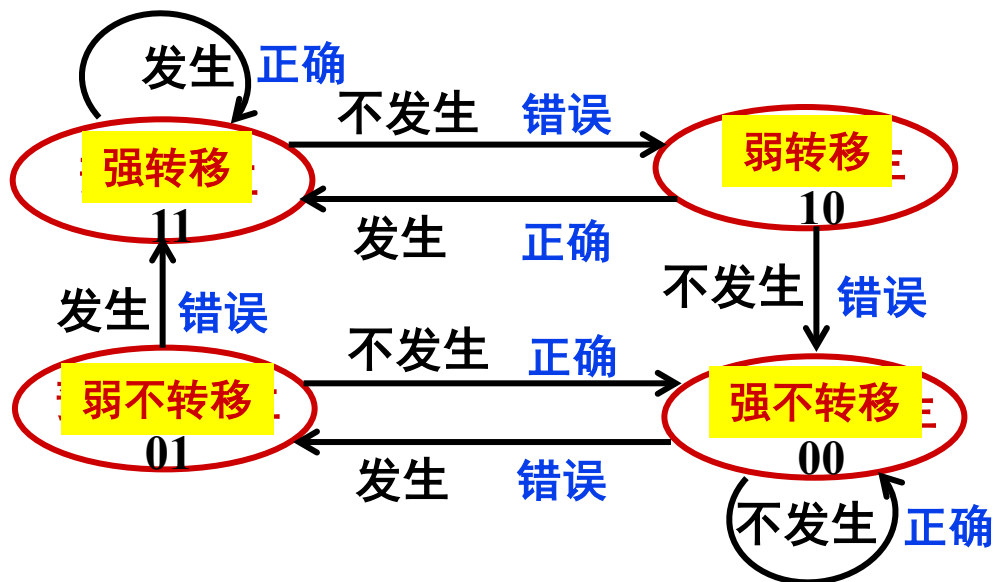
外循环中的分支指令共执行 $1 \times (N+1)$ 次，
内循环中的分支指令共执行 $N \times (N+1)$ 次。

$N=10$, 准确率分别90.9%和82.7%

$N=100$, 准确率分别99%和98%

若预测位初始为0，则外循环只有最后一次预测错误；跳出内循环时预测位变为1，再进入内循环时，第一次总是预测错误，并且任何一次循环的最后一次总是预测错误，因此，总共有 $1+2 \times (N-1)$ 次预测错误（第一次内循环有1次预测错，后面 $(N-1)$ 次内循环每次有2次预测错）。 **N 越大预测准确率越高！**

两位预测状态图

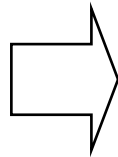


预测发生时，选择“转移取”
预测不发生时，选择“顺序取”

- 基本思想：只有两次预测错误才改变预测方向
 - 11状态时预测发生（强转移），实际不发生时，转到状态10（弱转移），下次仍预测为发生，如果再次预测错误（实际不发生），才使下次预测调整为不发生00
- 好处：连续两次发生不同的分支情况时，会预测正确

举例：双重循环的两位动态预测

```
into sum (int N)
{
  int i, j, sum=0;
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      sum=sum+1;
  return sum;
}
```



```
... ..
Loop-i: beq t1,a0, exit-i  # 若( i=N)则跳出外循环
        add t2, zero, zero #j=0
Loop-j: beq t2, a0, exit-j  # 若(j=N)则跳出内循环
        addi t2, t2, 1      # j=j+1
        addi t0, t0, 1      #sum=sum+1
        j Loop-j
exit-j: addi t1, t1, 1      # i=i +1
        j Loop-i
exit-i: ... ..
```

外循环中的分支指令共执行 $1 \times (N+1)$ 次， **$N=10$, 准确率分别90.9%和90.9%**
内循环中的分支指令共执行 $N \times (N+1)$ 次。 **$N=100$, 准确率分别99%和99%**

若预测位初始为00（强不转移），外循环只有最后一次预测错误；
跳出内循环时预测位变为01（弱不转移），再进入内循环时，第一次预测正确，且预测位再次变为00，如此循环，最终只有最后一次预测错误，
因此，总共有 N 次预测错误。

N 越大准确率越高！

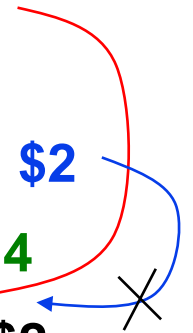
方案4： 延迟分支（分支延迟时间片的调度）

- 属于静态调度技术，由编译程序重排指令顺序来实现
- 基本思想：
 - 把分支指令前面的与分支指令无关的指令调到分支指令后面执行，以填充延迟时间片（也称分支延迟槽 **Branch Delay slot**），不够时用 **nop** 操作填充

举例：如何对以下程序段进行分支延迟调度？

（假定时间片 $C=2$ ）


```
lw $1, 0($2)
lw $3, 0($2)
add $6, $4, $2
beq $3, $5, 4
add $3, $3, $2
sw $1, 0($2)
.....
```



调度后

```
lw $3, 0($2)
add $6, $4, $2
beq $3, $5, 8
lw $1, 0($2)
nop
add $3, $3, $2
sw $1, 0($2)
```

若 $C=1$ ，则可以：



```
lw $3, 0($2)
add $6, $4, $2
beq $3, $5, 6
lw $1, 0($2)
add $3, $3, $2
sw $1, 0($2)
.....
```

调度后可能带来其他问题：产生新的
load-use数据冒险

调度后，降低了分支延迟损失

另一种控制冒险：异常和中断

- 异常和中断会改变程序的执行流程
- 某条指令发现异常时，后面多条指令已被取到流水线中正在执行
 - 例如ALU指令发现“溢出”时，已经到EX阶段结束了，此时，它后面已有两条指令进入流水线了
- 流水线数据通路如何处理异常？(举例说明)
 - 假设指令add r1,r2,r3产生了溢出
 - 处理思路：
 - ✓ 清除add指令以及后面的所有已在流水线中的指令
 - ✓ 关中断（将中断允许触发器清0）
 - ✓ 保存PC或PC-4（断点）到EPC
 - ✓ 将异常/中断处理程序首地址送PC

三种处理器实现方式的比较

◦ 单周期、多周期、流水线三种方式比较

假设各主要功能单元的操作时间为：

- 存储单元：200ps
- ALU和加法器：100ps
- 寄存器堆（读 / 写）：50ps

假设MUX、控制单元、PC、扩展器和传输线路都没有延迟，指令组成为：
25%取数、10%存数、52%ALU、11%分支、2%跳转，则下面实现方式中，
哪个更快？快多少？

- (1) 单周期：每条指令在一个固定长度的时钟周期内完成
- (2) 多周期：每类指令时钟数为取数-5，存数-4，ALU-4，分支-3，跳转-3
- (3) 流水线：每条指令分取指令、取数/译码、执行、存储器存取、写回五阶段
(假定没有结构冒险，数据冒险采用转发处理，分支延迟槽为1，预测准确率为75%；无条件跳转指令的更新地址工作也在ID段完成。不考虑流水段寄存器延时，不考虑异常、中断和访存缺失引起的流水线冒险)

三种处理器实现方式的比较

解：CPU执行时间 = 指令条数 x CPI x 时钟周期长度

三种方式指令条数都一样，所以只要比较CPI和时钟周期长度即可。

各指令类型要求的时间长度为：

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

对于单周期方式：

时钟周期将由最长指令来决定，应该是load指令，为600ps

所以，N条指令的执行时间为600N(ps)

对于多周期方式：

时钟周期将取功能部件最长所需时间，应该是存取操作，为200ps

根据各类指令的频度，计算平均时钟周期数为：

CPU时钟周期 = $5 \times 25\% + 4 \times 10\% + 4 \times 52\% + 3 \times 11\% + 3 \times 2\% = 4.12$

所以，N条指令的执行时间为 $4.12 \times 200 \times N = 824N(ps)$

三种处理器实现方式的比较

注意：这里的CPI已经并非是一条指令单实际执行总时长了

对于流水线方式：

Load指令：当发生Load-use依赖时，执行时间为2个时钟，否则1个时钟，
故平均执行时间为1.5个时钟；

Store、ALU指令：1个时钟；

Branch指令：预测成功时，1个时钟，预测错误时，2个时钟，
所以：平均约为： $.75 \times 1 + .25 \times 2 = 1.25$ 个；

Jump指令：2个时钟（总要等到译码阶段结束才能得到转移地址）

平均CPI为： $1.5 \times 25\% + 1 \times 10\% + 1 \times 52\% + 1.25 \times 11\% + 2 \times 2\% = 1.17$

所以，N条指令的执行时间为 $1.17 \times 200 \times N = 234N(\text{ps})$

第六讲小结

- 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
- 异常和中断是一种特殊的控制冒险

全课程终

谢谢大家