Lecture 25: Pipeline Hazard

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
- 访问缺失对流水线的影响

总结：流水线的三种冲突/冒险（Hazard）情况

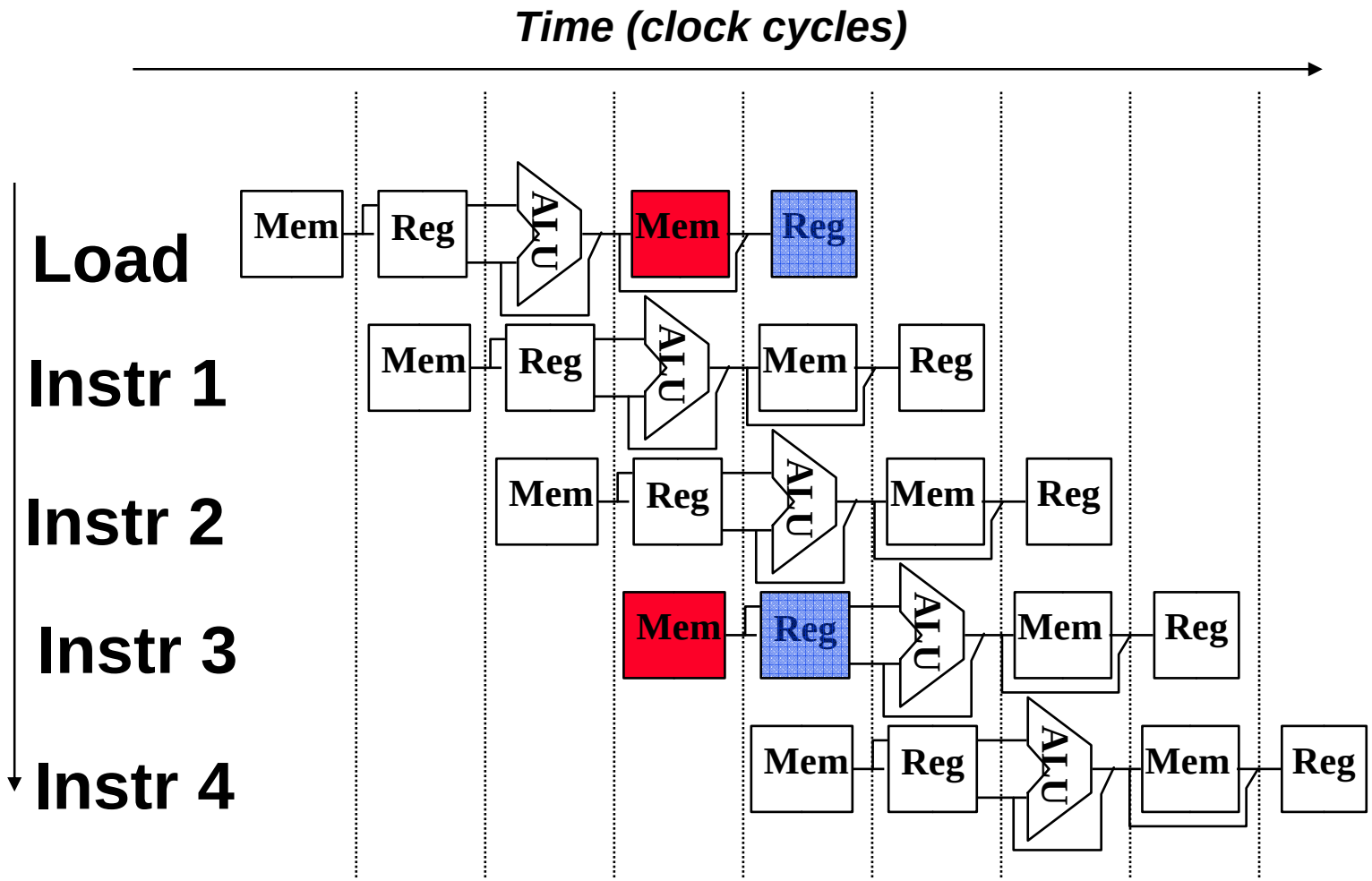
- **Hazards**：指流水线遇到无法正确执行后续指令或执行了不该执行的指令
  - **Structural hazards (hardware resource conflicts):**

    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



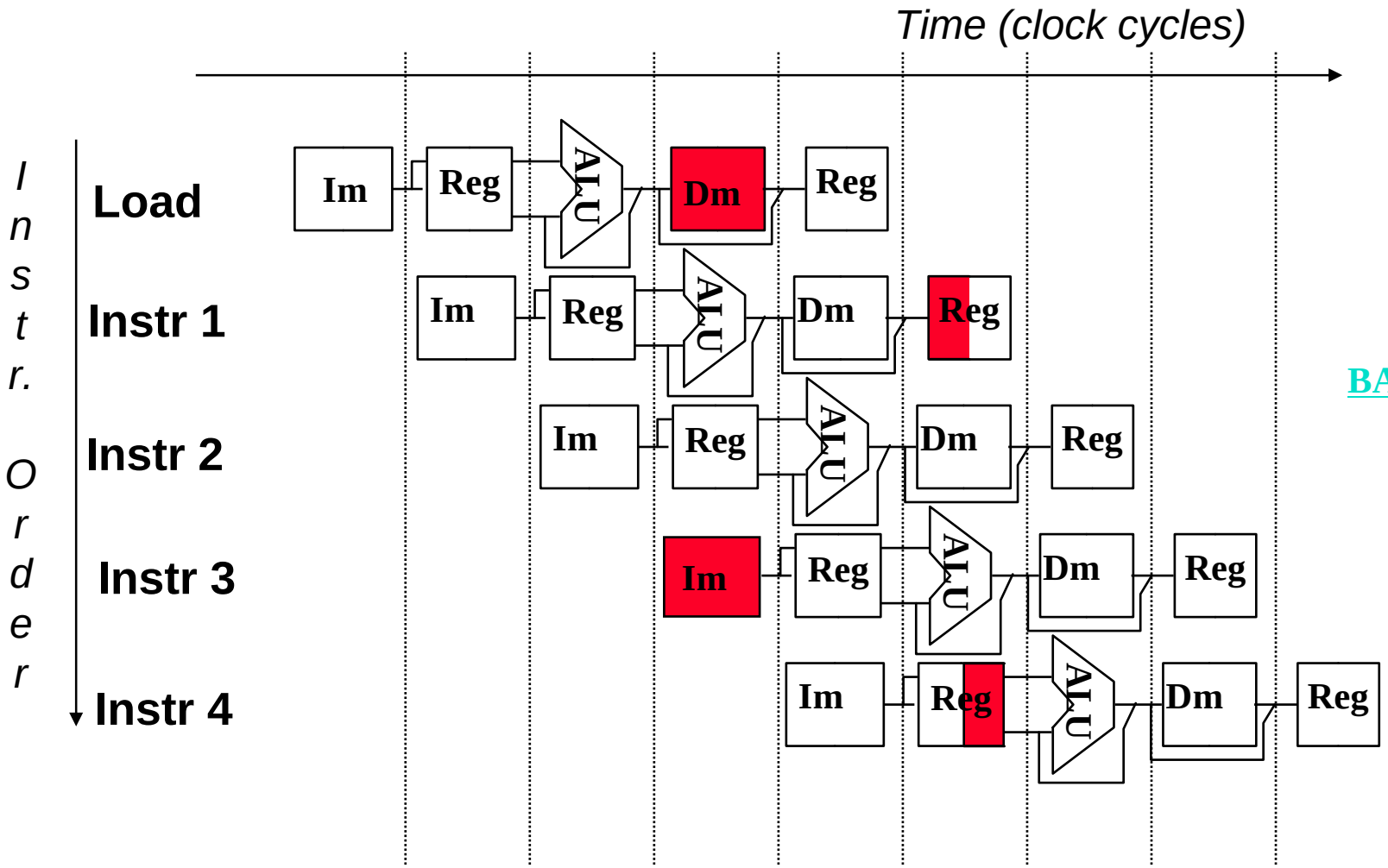
如果只有一个存储器，则在**Load**指令取数据同时又取指令的话，则发生冲突！

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



BACK

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

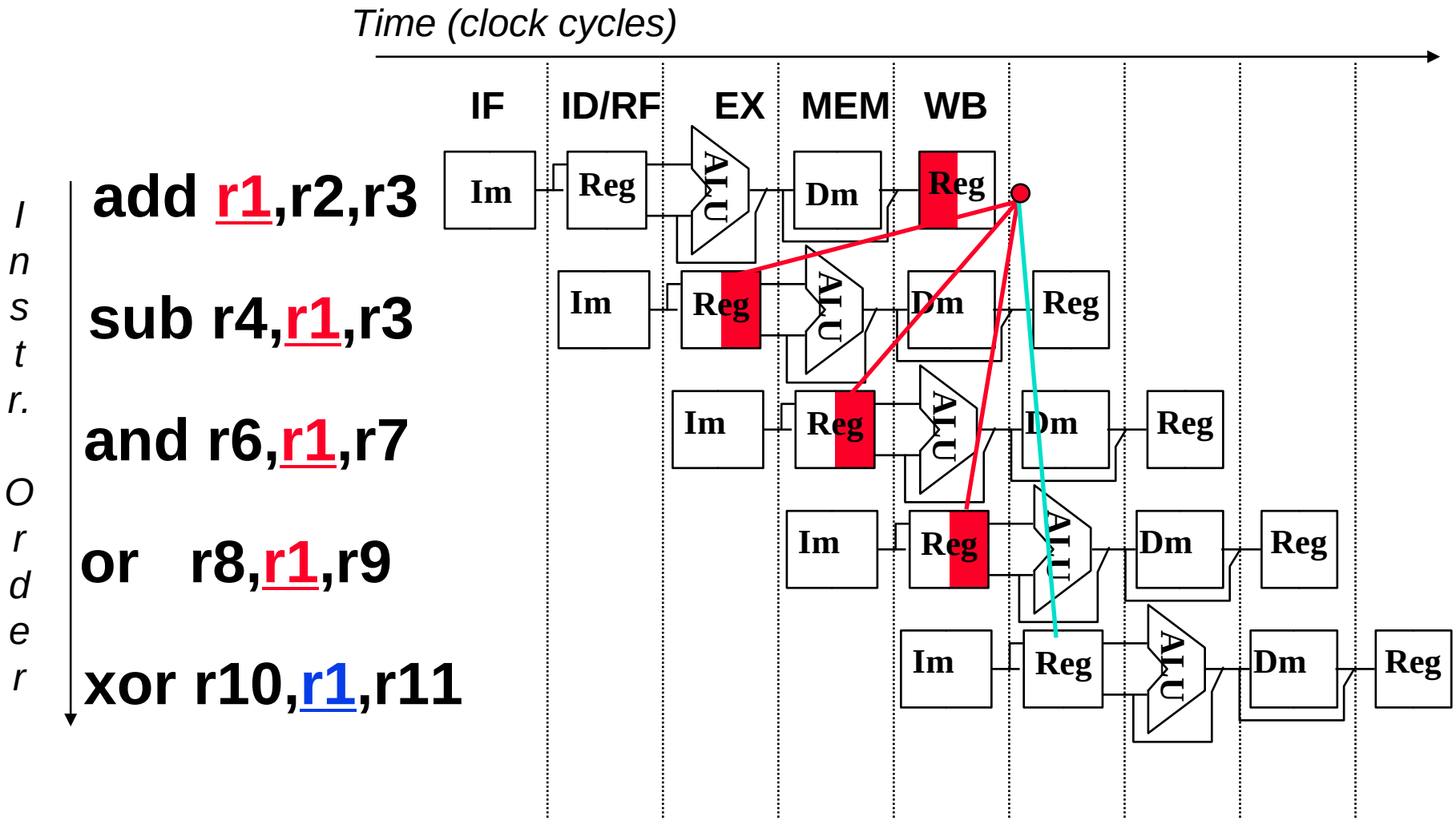
RAW：写后读（基本流水线中经常发生，如上例）

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



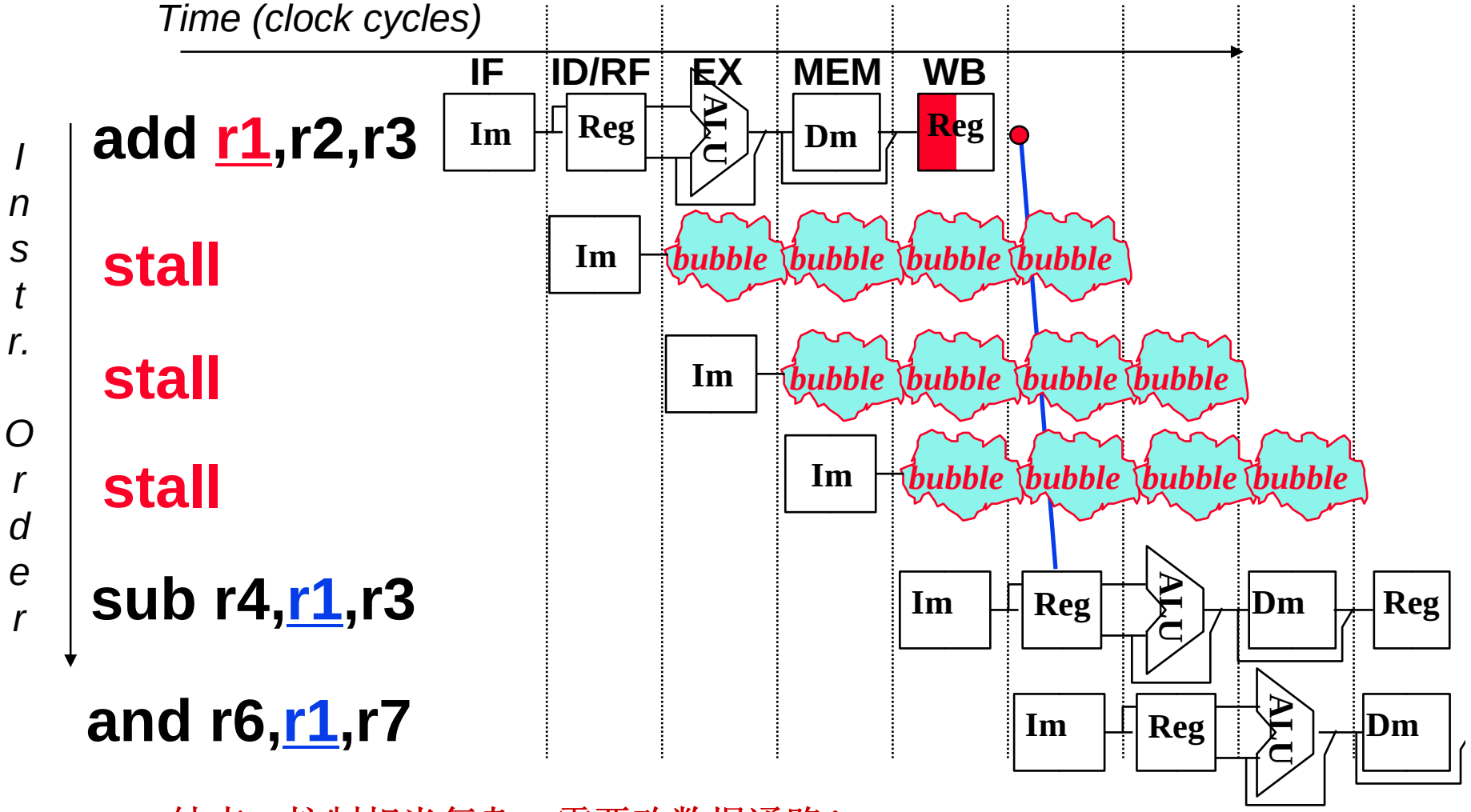
最后一条指令的**r1**才是新的值！

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！

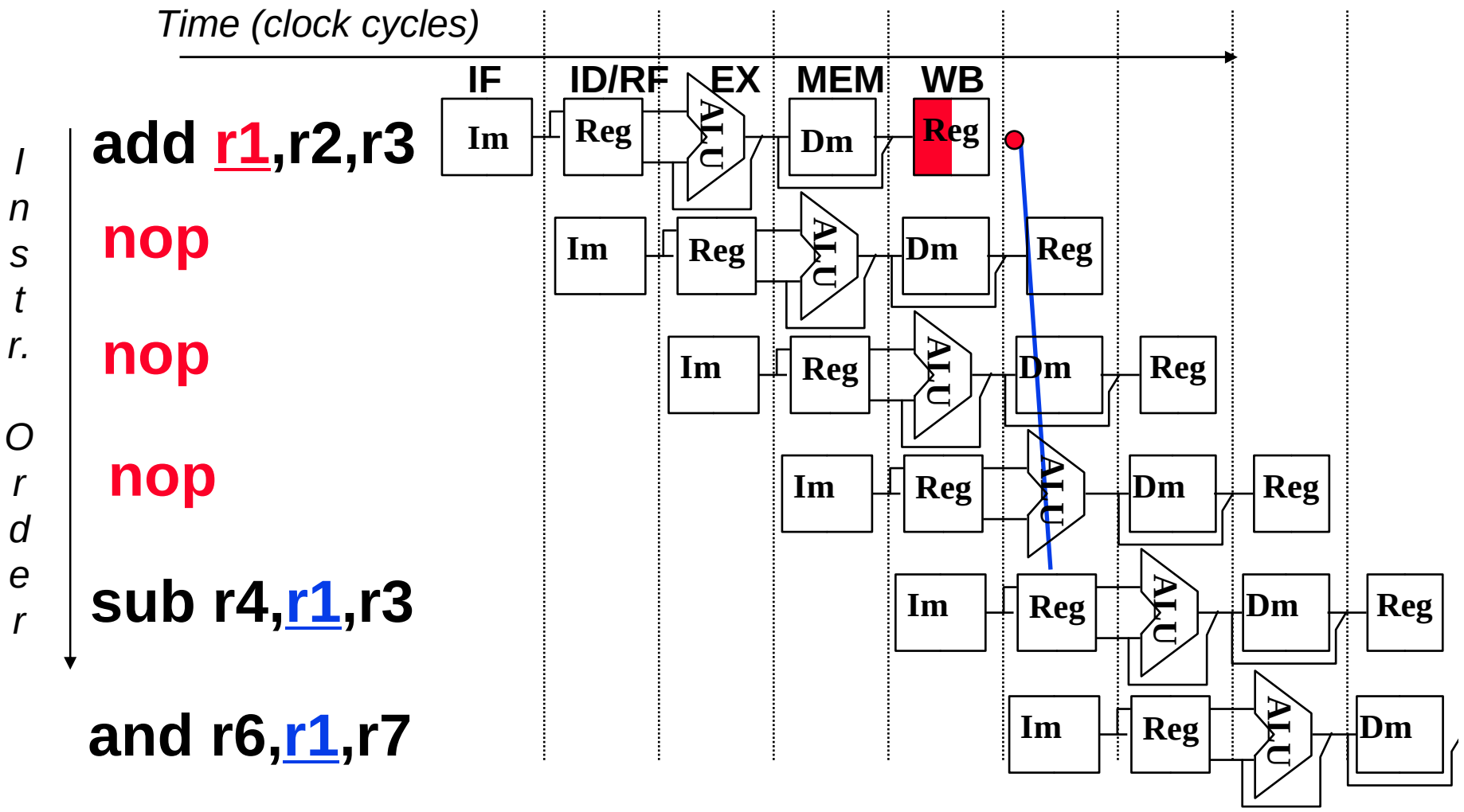
  这种做法称为流水线阻塞，也称为“气泡Bubble”



- 缺点：控制相当复杂，需要改数据通路！

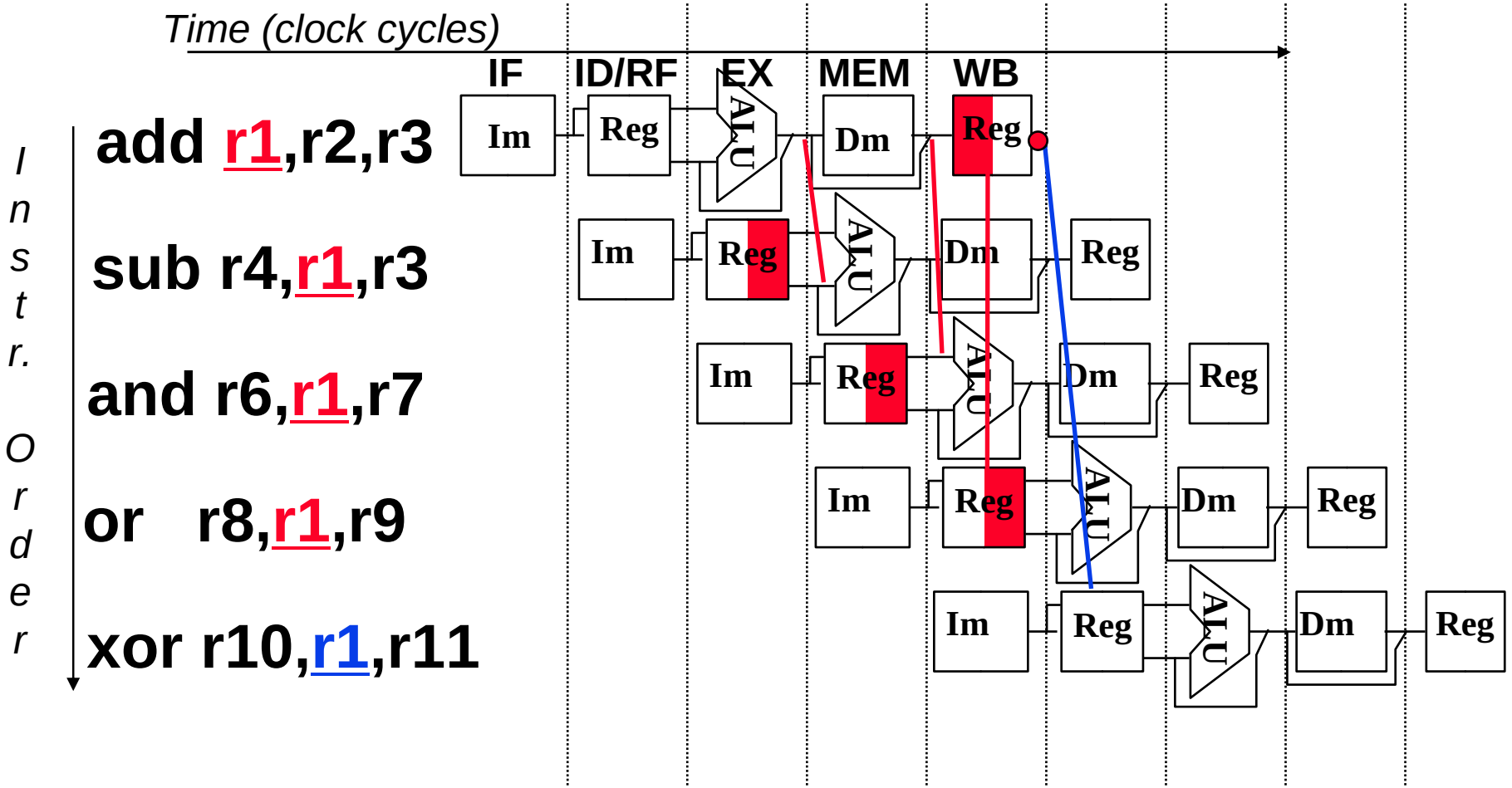
方案 2: 软件上插入无关指令

- 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？



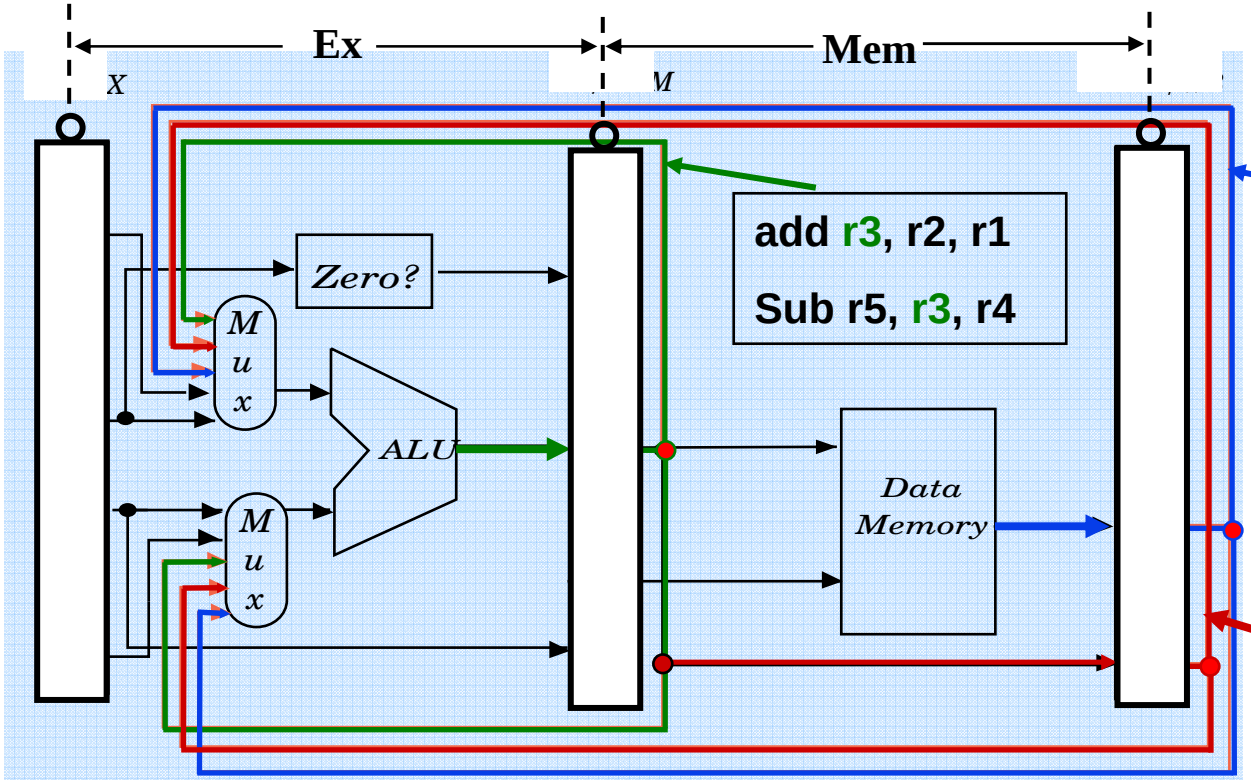
1. 把数据从流水段寄存器中直接取到ALU的输入端
2. 寄存器写/读口分别在前/后半周期，使写入被直接读出

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)

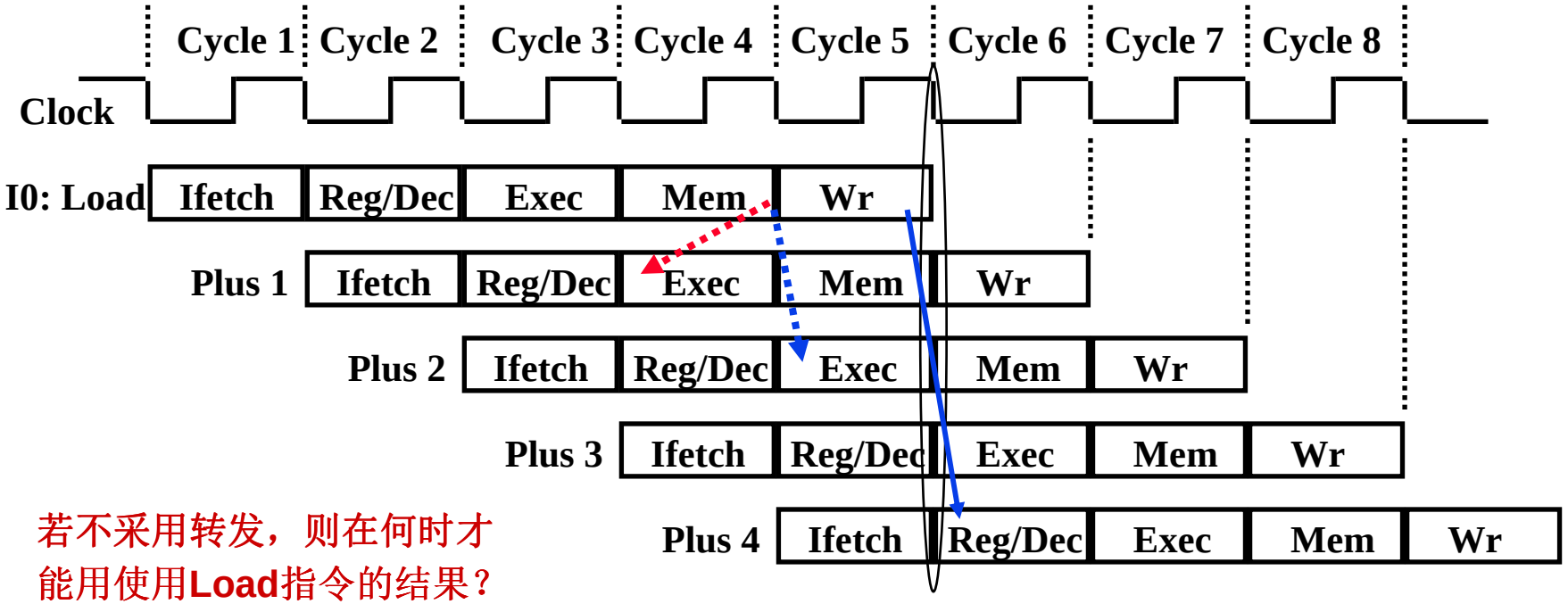
Or r6, r3, r1

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指令引起的延迟现象



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？

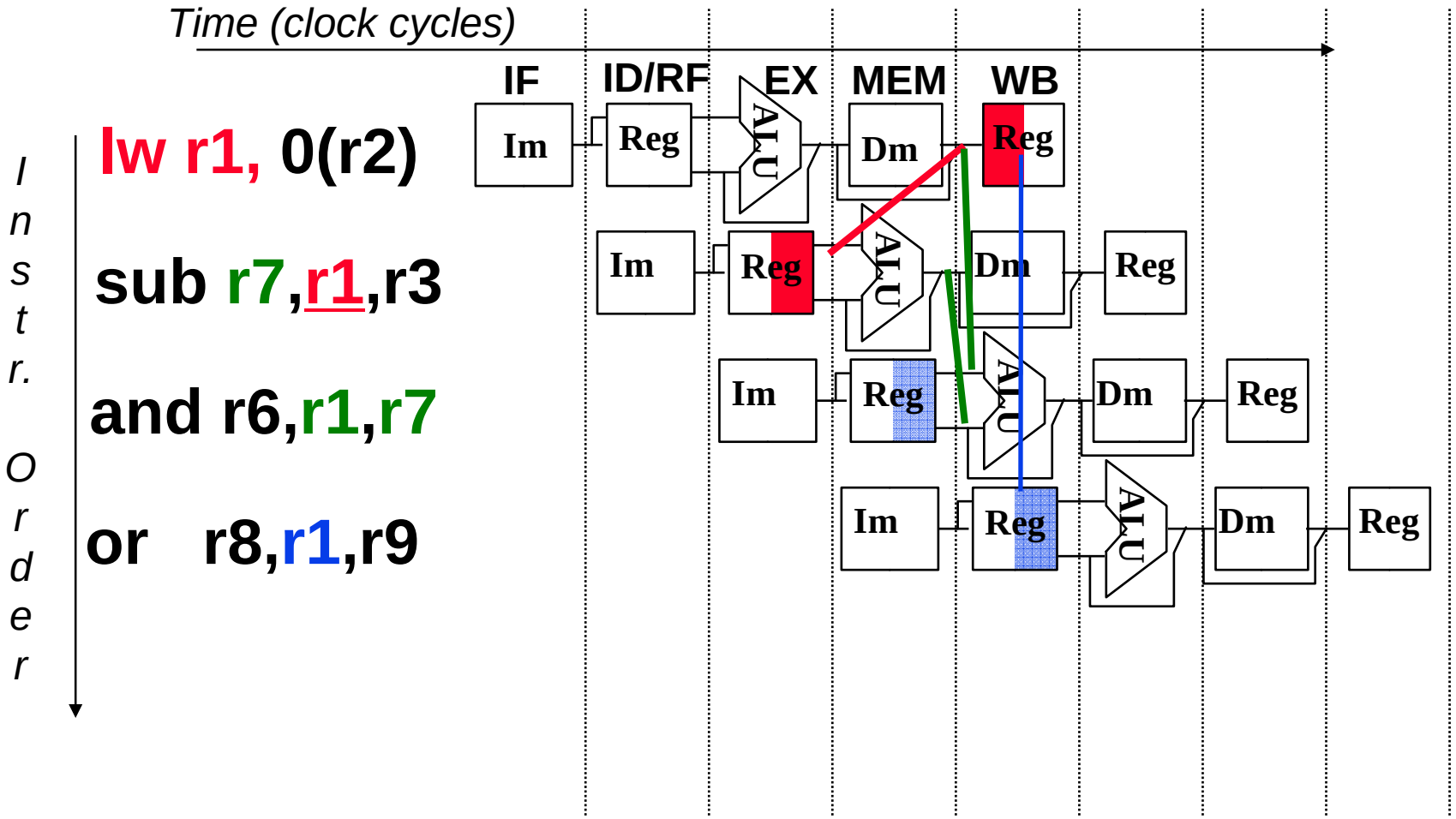
  实际上，在第四周期结束时，数据在流水段寄存器中已经有值。

  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)”

“Forwarding”技术使Load-use冒险只需延迟一个周期

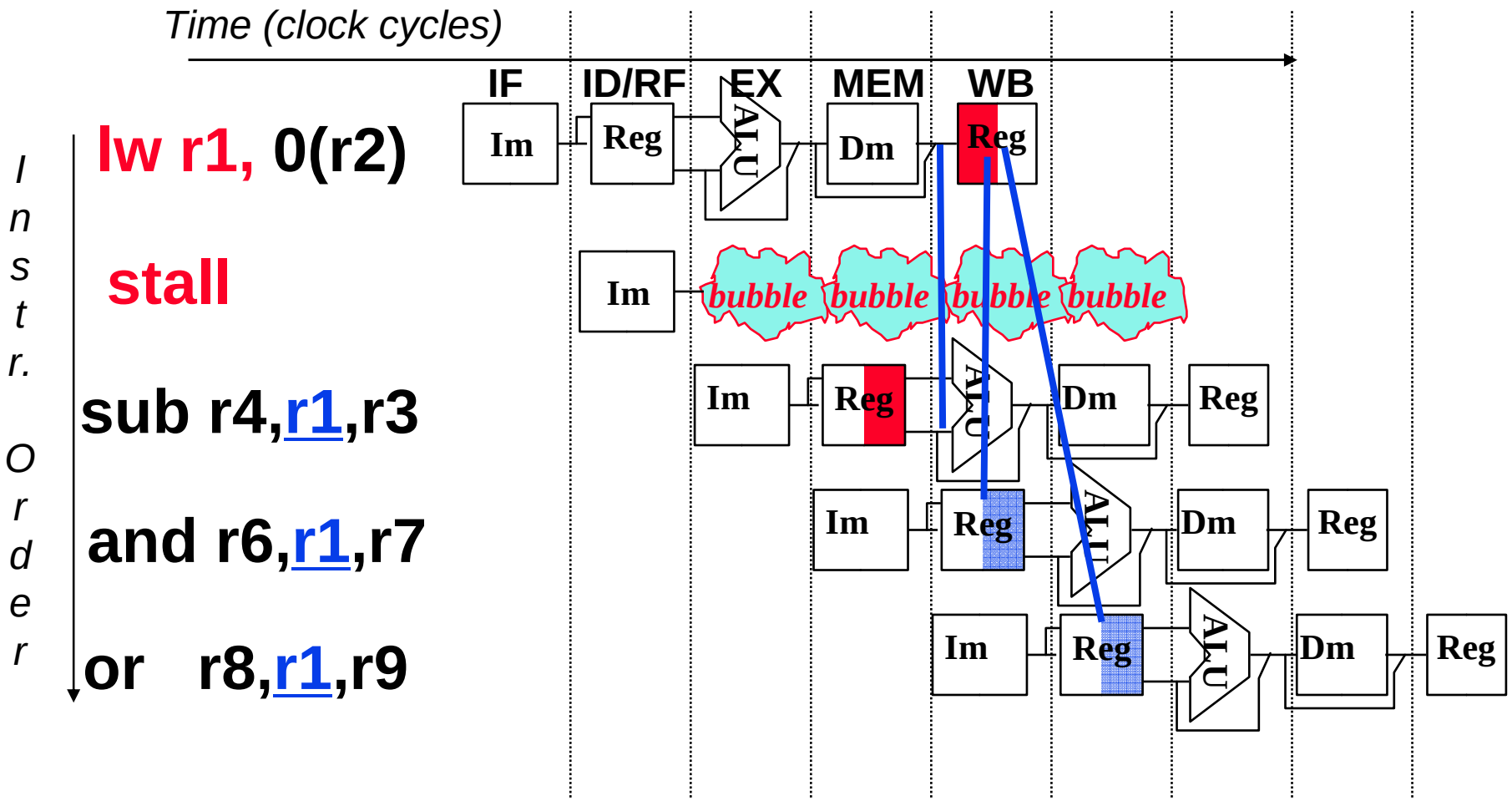


采用“转发”后仅第二条指令 **SUB r7,r1,r3** 不能按时执行！ 需要阻塞一个周期。

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

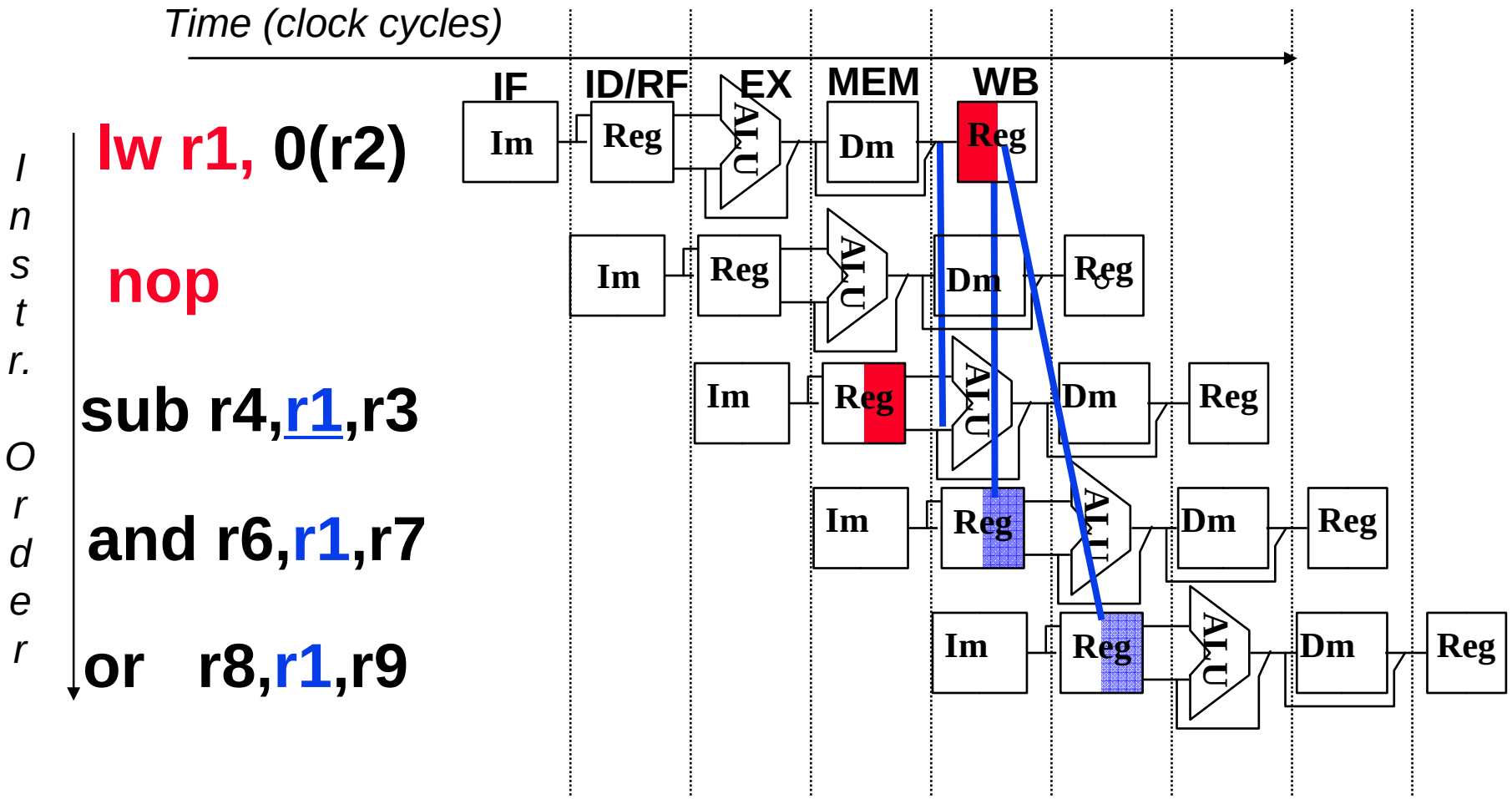
用硬件阻塞一个周期（指令被重复执行一次）



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
```

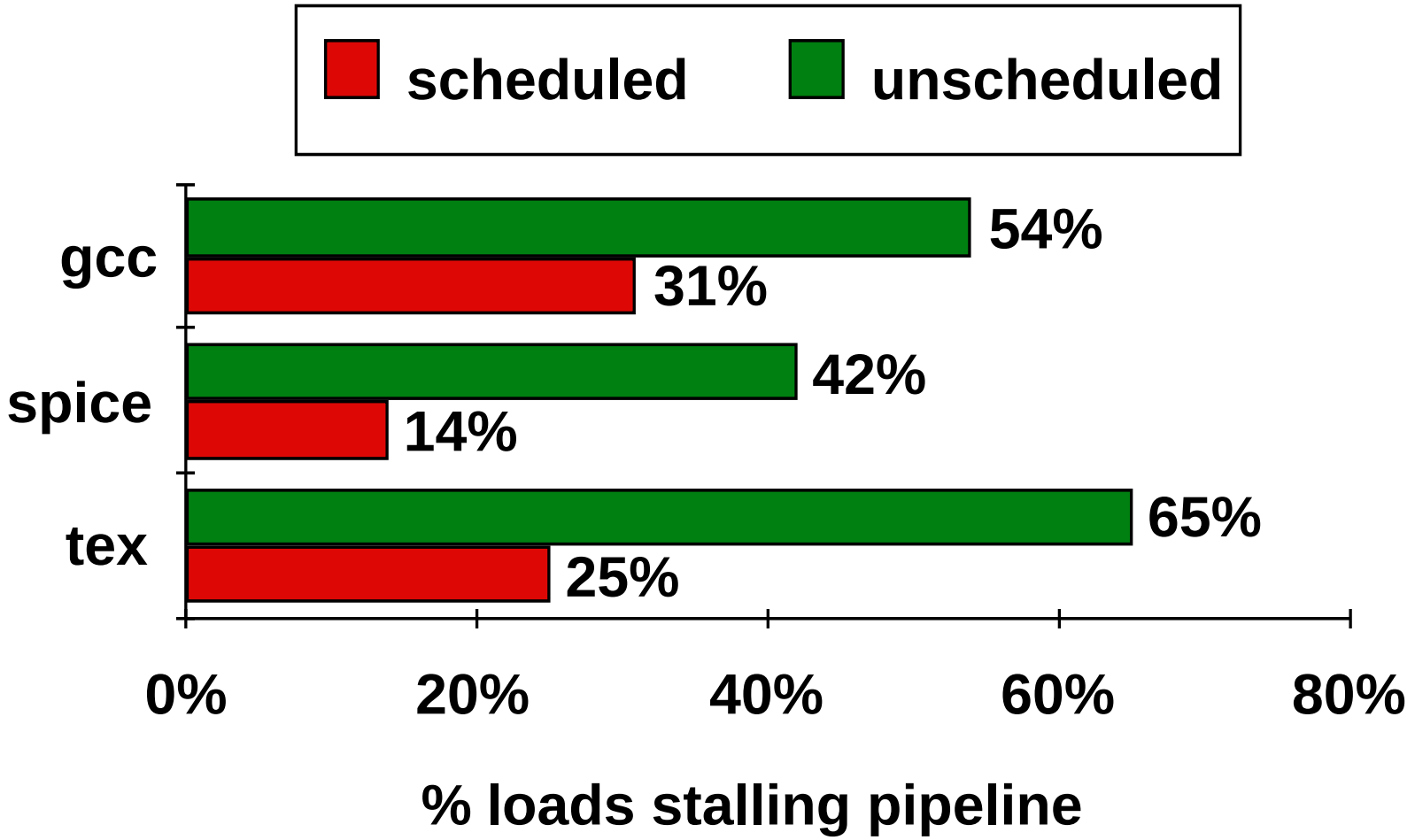
调整后

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！

编译器优化以避免阻塞的情况调查:



由此可见，优化调度后load阻塞现象大约降低了1/2~1/3

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

    称为Load-use数据冒险！

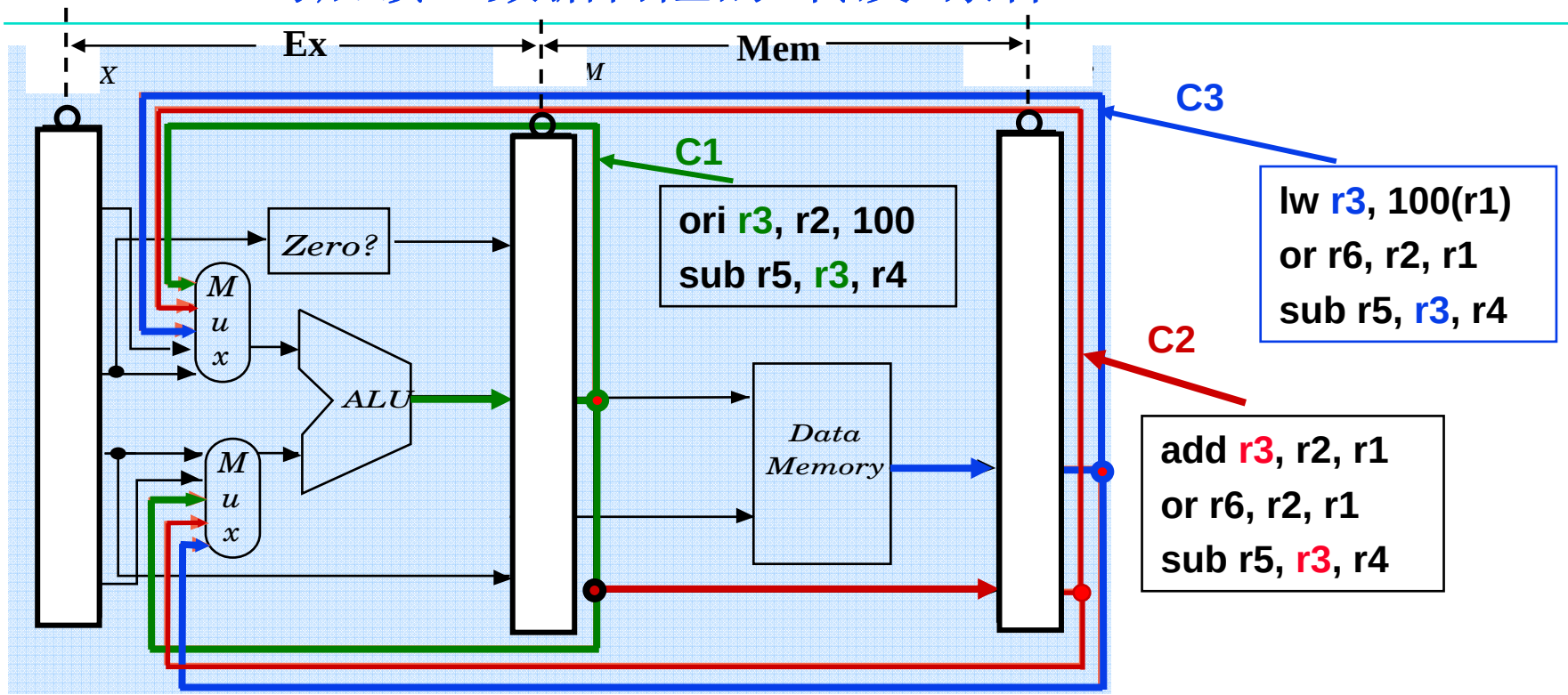
实现“转发”和“阻塞” 要修改数据通路：

（1）检测何时需要“转发”，并控制实现“转发”

（2）检测何时需要“阻塞”，并控制实现“阻塞”

BACK

RAW（写后读）数据冒险的“转发”条件



后面指令需用ALU输出结果

C1: 目寄是后一条指令的源寄

C2: 目寄是后第二条指令的源寄

(例如：R-Type后跟R- / lw / sw / beq等)

后面指令需用从DM读出的结果

C3: 目寄是后第二条指令的源寄

(例如：load指令后跟R-Type / beq等)

用流水段寄存器来表示转发条件（C3以后考虑）

C1(a): EX/MEM. RegisterRd=ID/EX. RegisterRs

C1(b): EX/MEM. RegisterRd=ID/EX. RegisterRt

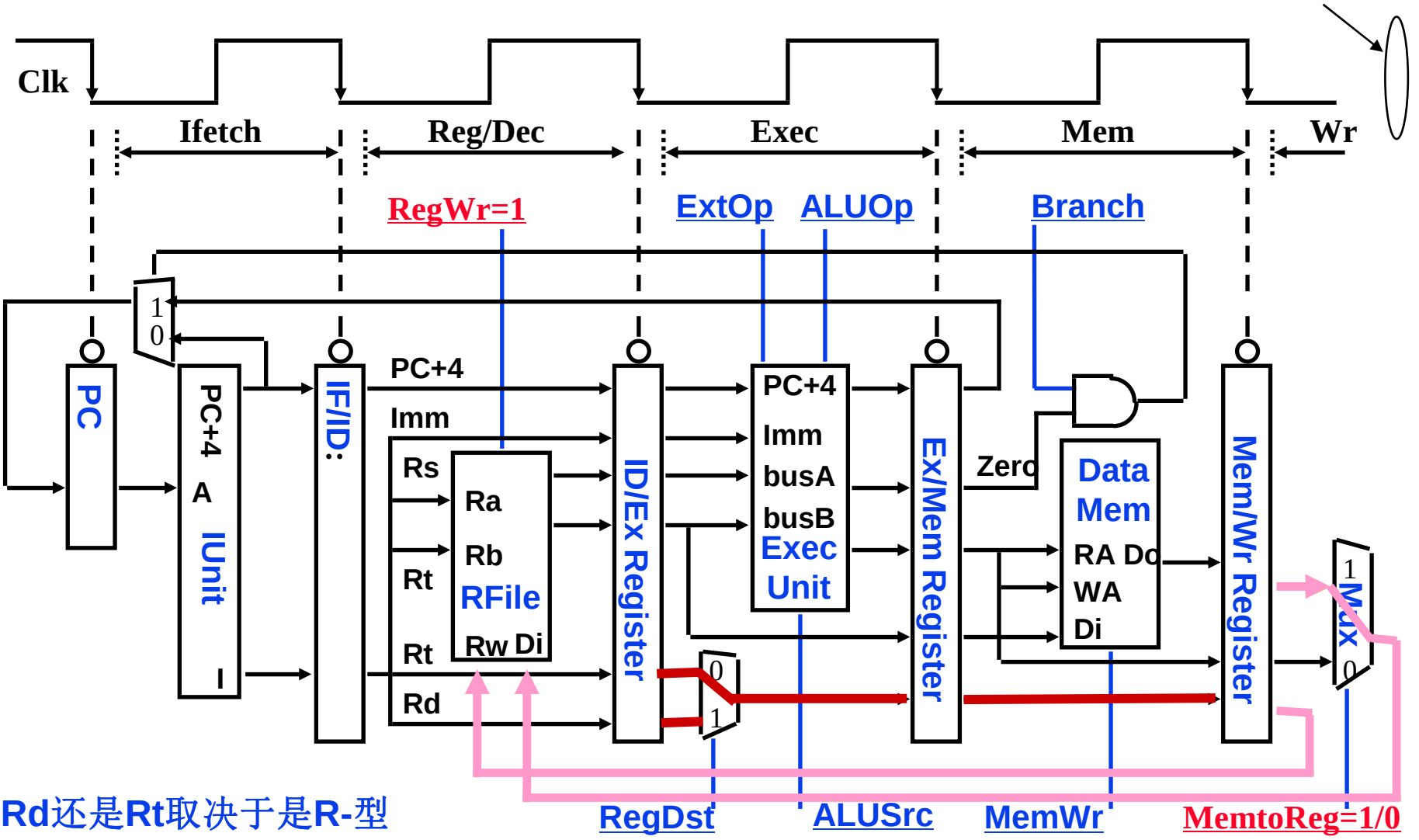
C2(a): MEM/WB. RegisterRd=ID/EX. RegisterRs

C2(b): MEM/WB. RegisterRd=ID/EX. RegisterRt

这里的**RegisterRd**是指目的寄存器

实际上是**R-type**的**Rd** 或 **I-Type**的**rt**

指令的回写（Write Back）阶段



Rd还是Rt取决于是R-型指令，还是I-型指令！

若是beq指令会怎样？

```
beq r3, r2, 100
sub r5, r3, r2
```

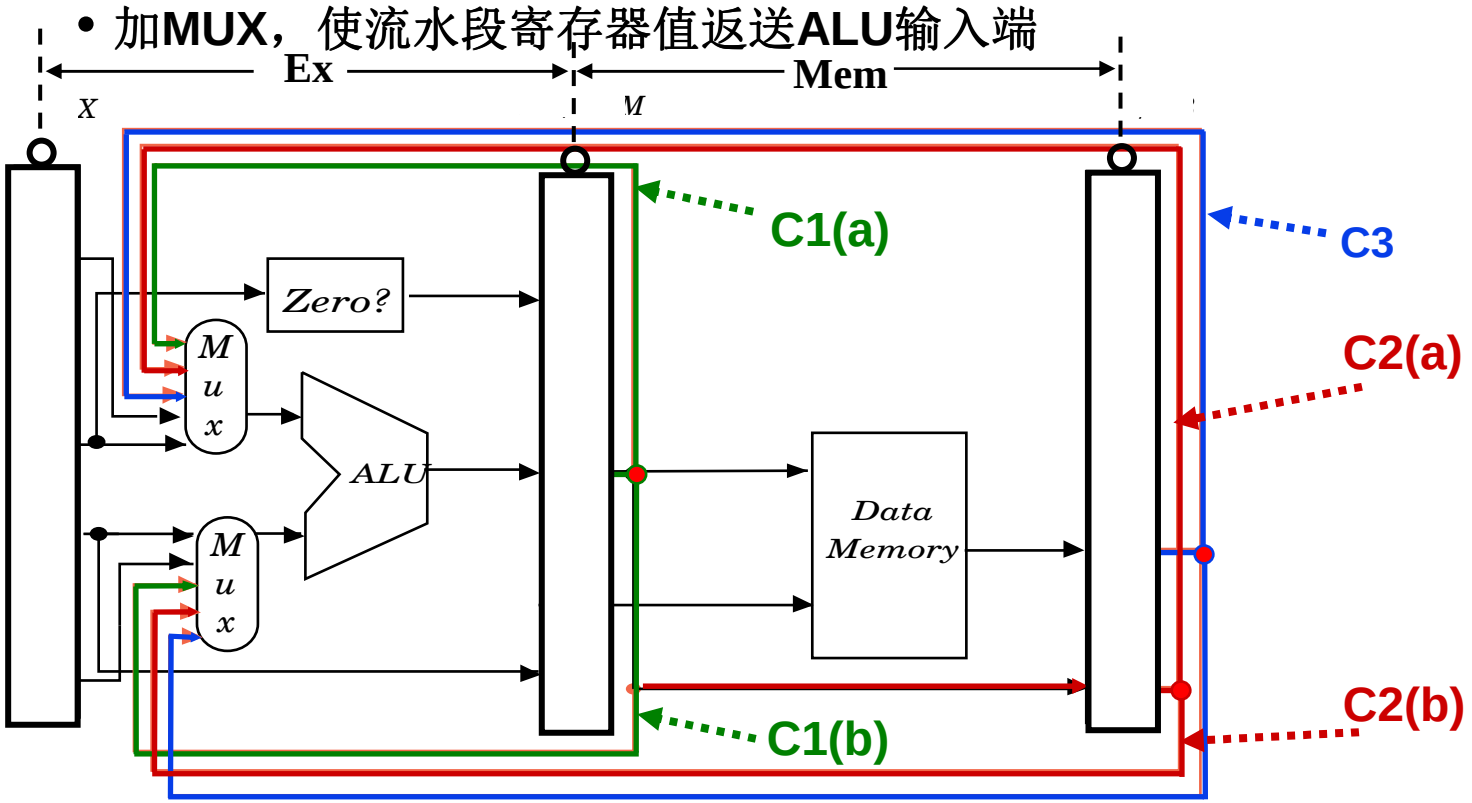
因为beq指令没有写结果，不用进行转发！

转发条件的进一步完善

```
beq r3, r2, 100
sub r5, r3, r2
```

- 以下两种情况下，根据前面的转发条件转发会发生错误
  - 指令的结果不写入目的寄存器Rd时
    - 例如，Beq指令只对rs和rt相减，不写结果到目的寄存器
    - 即：EX / MEM 或 MEM / WB 流水段寄存器的RegWrite信号为0
  - Rd等于$0时
    - 例如，指令 sll $0, $1, 2 的转发结果为(R[$1]<<2)，但实际上应该是0
- 因此，修改转发条件为：
  - C1(a): EX/MEM.RegWrite
    and EX/MEM. RegisterRd ≠ 0
    and EX/MEM. RegisterRd=ID/EX. RegisterRs
  - C1(b): EX/MEM.RegWrite
    and EX/MEM. RegisterRd ≠ 0
    and EX/MEM. RegisterRd=ID/EX. RegisterRt
  - C2(a): MEM/WB.RegWrite
    and MEM/WB. RegisterRd ≠ 0
    and MEM/WB. RegisterRd=ID/EX. RegisterRs
  - C2(b): MEM/WB.RegWrite
    and MEM/WB. RegisterRd ≠ 0
    and MEM/WB. RegisterRd=ID/EX. RegisterRt

转发路径和转发条件



C1反映本条指令和随后指令间的相关关系

C2反映本条指令和随后第二条指令间的相关关系

C1(a)和C1(b)可以合并为一个条件C1，并把转发线合一起后同时送A口和B口

即：C1=C1(a) or C1(b)，同样：C2=C2(a) or C2(b)，转发线合起来

实际上红线和兰线可以合并，而且在原数据通路中是合并在一起的。记得吗？

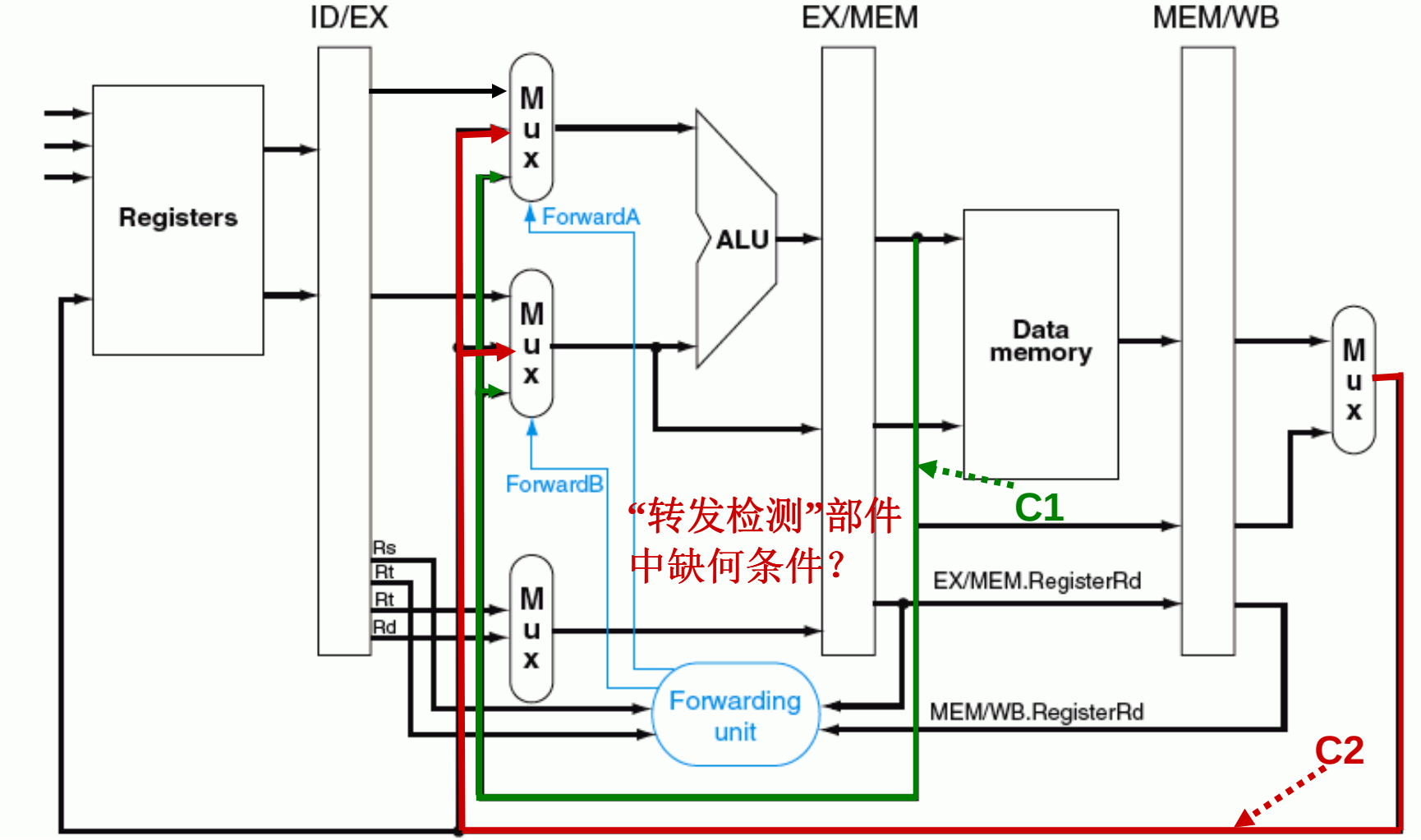
由一个二路选择器（控制端为MemtoReg）合并输出到寄存器堆！

所以不需另外有一个检测条件C3!

C1和C2分别反映哪两条指令的关系呢？

转发路径和转发条件

ForwardA (ForwardB) = { 01 当c2=1时
10 当c1=1时

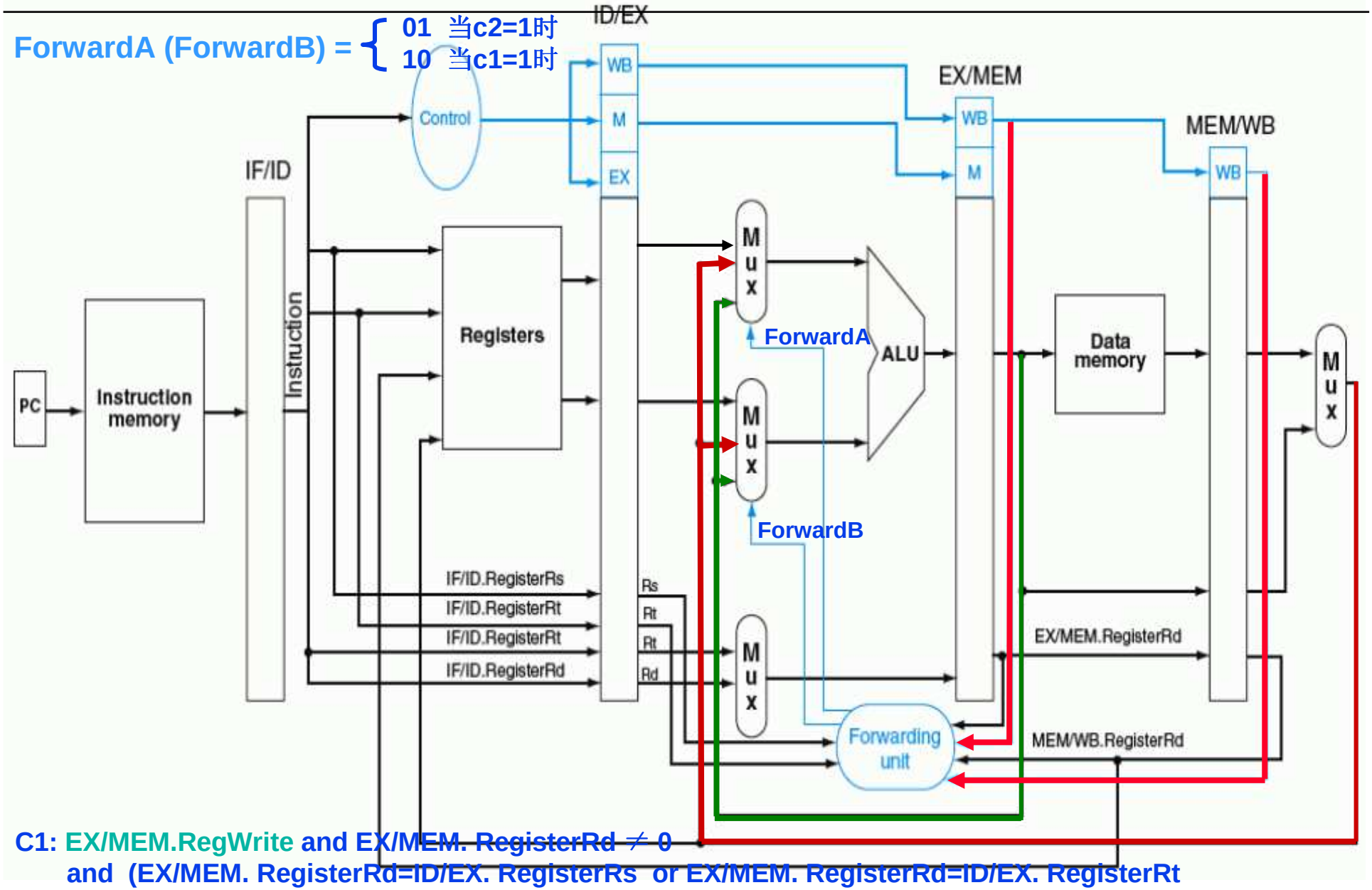


C1: EX/MEM.RegWrite and EX/MEM. RegisterRd ≠ 0
and (EX/MEM. RegisterRd=ID/EX. RegisterRs or EX/MEM. RegisterRd=ID/EX. RegisterRt

C2: MEM/WB.RegWrite and MEM/WB. RegisterRd ≠ 0
and (MEM/WB. RegisterRd=ID/EX. RegisterRs or MEM/WB. RegisterRd=ID/EX. RegisterRt

带转发的流水线数据通路

ForwardA (ForwardB) = { 01 当c2=1时
10 当c1=1时



C1: EX/MEM.RegWrite and EX/MEM. RegisterRd ≠ 0
and (EX/MEM. RegisterRd=ID/EX. RegisterRs or EX/MEM. RegisterRd=ID/EX. RegisterRt

C2: MEM/WB.RegWrite and MEM/WB. RegisterRd ≠ 0
and (MEM/WB. RegisterRd=ID/EX. RegisterRs or MEM/WB. RegisterRd=ID/EX. RegisterRt

更加复杂的数据冒险问题

- 考察以下指令序列，采用前述转发条件会发生什么情况？

```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4   <-- 本条指令
……
```

ForwardA (ForwardB) = { 01 当c2=1时 ; 10 当c1=1时

对于左边的指令序列，C1和C2的值各是什么？

C1=C2=1，使得Forward信号取值不确定！

可能会使转发到第3条指令的操作数是第1条指令结果，而不是第2条指令的结果！

怎样改写“转发”检测条件：改C1还是改C2?　应该让C1=1,C2=0!

- 需要改写“转发”条件C2为：

MEM/WB.RegWrite
and MEM/WB.RegisterRd ≠ 0
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs or EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
and (MEM/WB.RegisterRd=ID/EX.RegisterRs or MEM/WB.RegisterRd=ID/EX.RegisterRt)

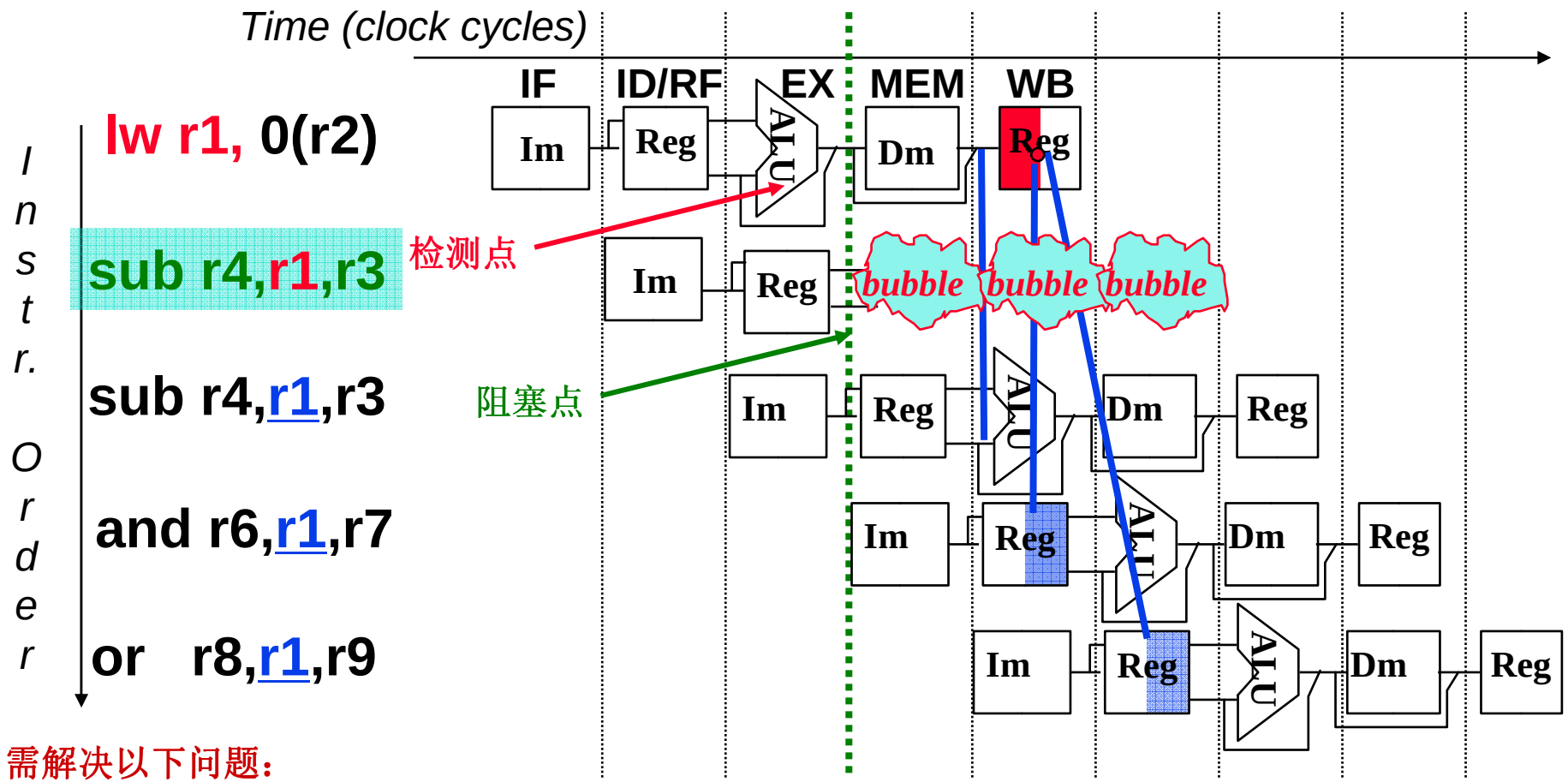
上述公式相当于加了一个条件限制：

如果本条指令源操作数和上条指令的目的寄存器一样，则不转发上上条指令的结果，
而转发上条指令的结果（即：此时的C1=1而C2=0）

至此，解决了RAW数据冒险的“转发”处理

BACK

Load-use Data Hazard（硬件阻塞方式）



需解决以下问题：

(1) 判断什么条件下需要阻塞

ID/EX.MemRead

and (ID/EX.RegisterRt=IF/ID.RegisterRs

or ID/EX.RegisterRt=IF/ID.RegisterRt)

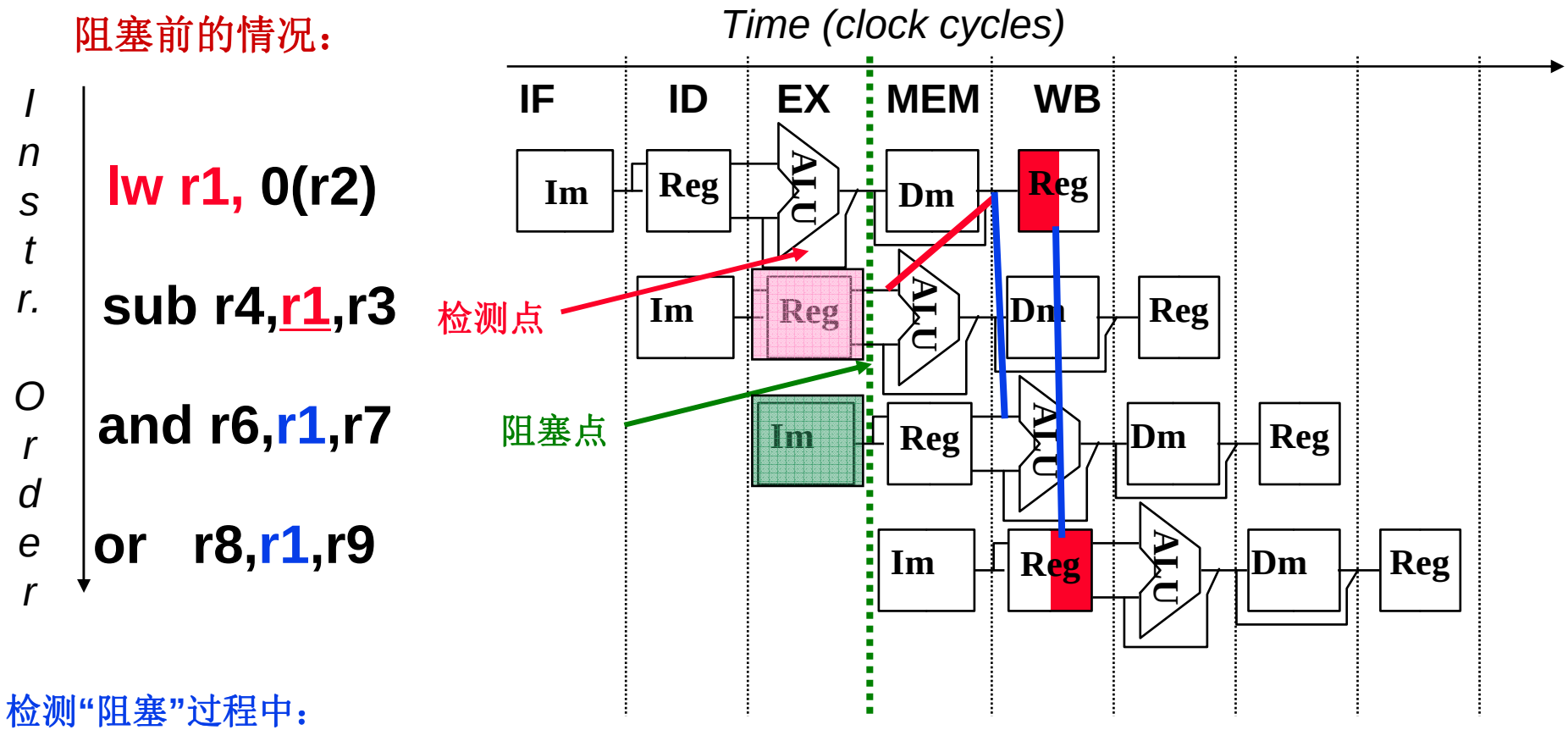
前面指令为Load 并且

前面指令的目的寄存器等于当前刚取出指令的源寄存器

(2) 如何修改数据通路来实现阻塞

Load-use Data Hazard（硬件阻塞方式）

阻塞前的情况：



检测“阻塞”过程中：

1) sub指令在IF/ID段寄存器中，并正被译码/取数，控制信号和Rs/Rt的值将被写到ID/EX段寄存器

2) and指令地址在PC中，正被取出，取出的指令将被写到IF/ID段寄存器中

在阻塞点，必须将上述两条指令的执行结果清除，并延迟一个周期执行这两条指令

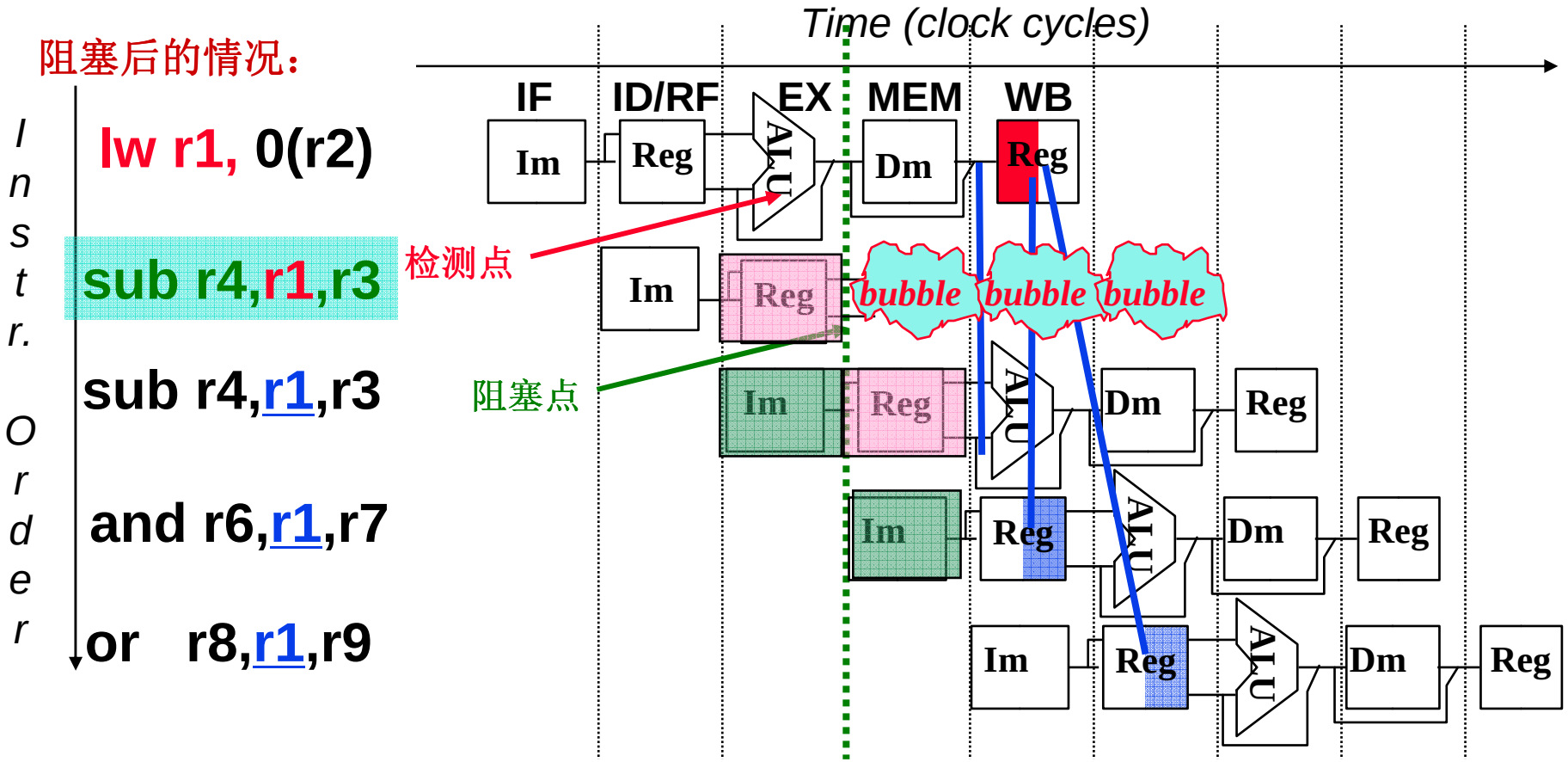
延迟一个周期执行后面的指令，相当于把阻塞点前面一个周期的状态再保持一个周期

lw指令还是继续正常执行下去

想想看，如何做到继续保持状态？

Load-use Data Hazard（硬件阻塞方式）

阻塞后的情况：



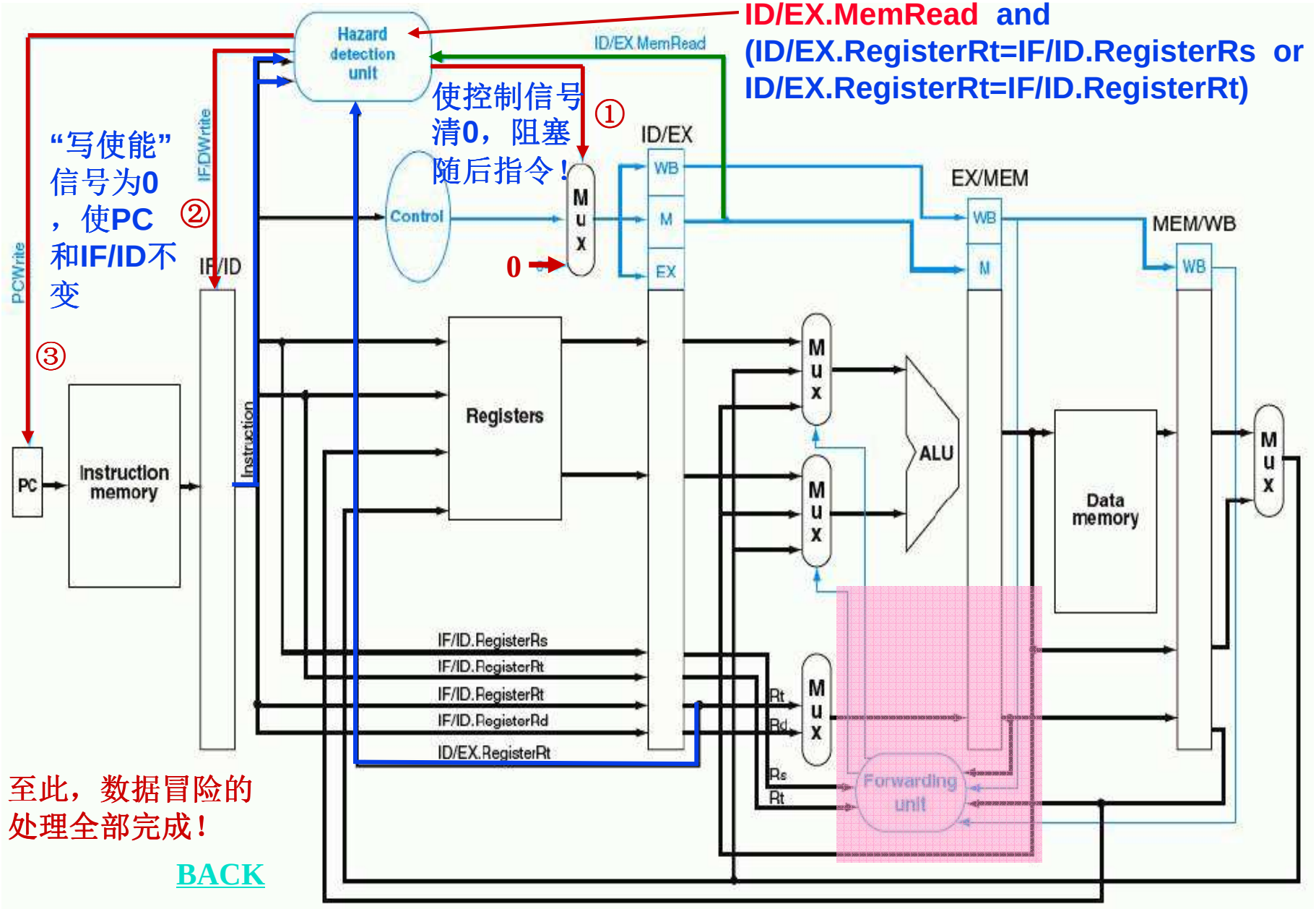
在阻塞点，必须将上述两条指令的执行结果清除，并延迟一个周期执行这两条指令

① 将ID/EX段寄存器中所有控制信号清0

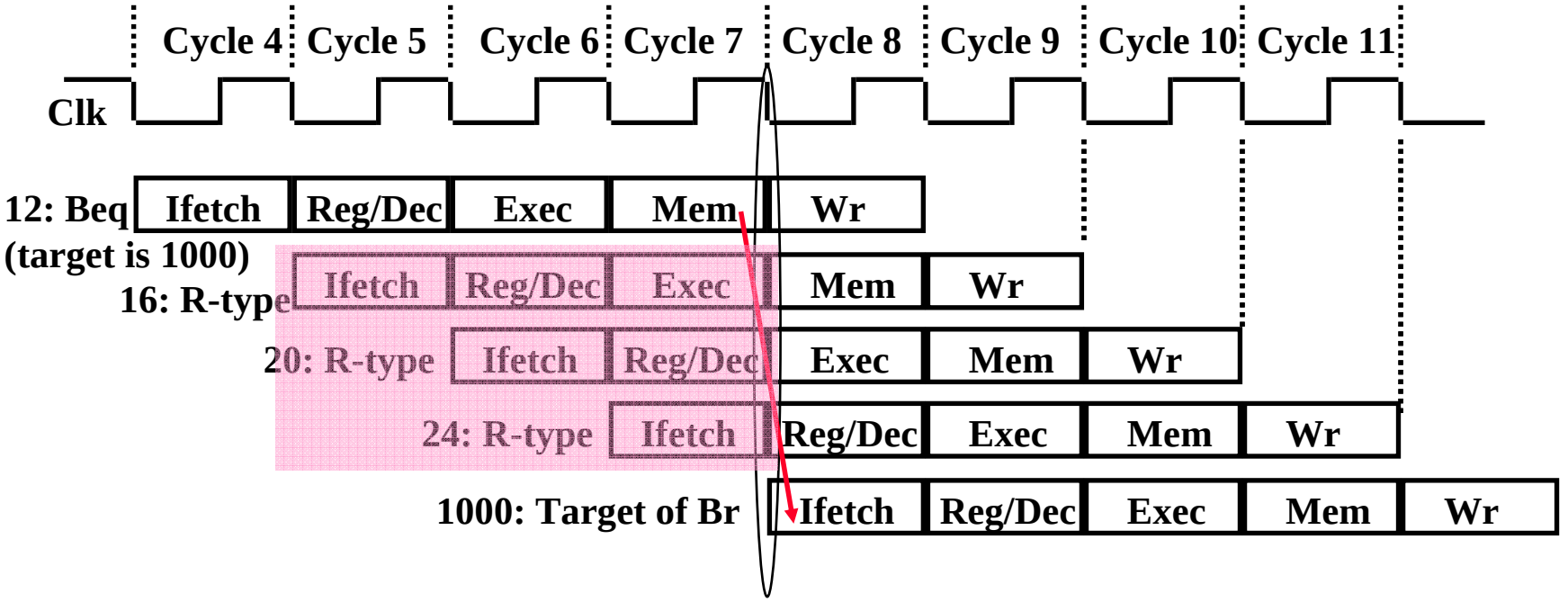
② IF/ID寄存器中的信息不变，sub指令重新译码执行

③ PC中的值不变，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



转移分支指令(Branch)引起的“延迟”现象 （复习）



- 虽然Beq指令在第四周期取出，但:
  - 目标地址在第七周期才被送到PC的输入端
  - 第八周期才能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！
- 这种现象称为控制冒险（Control Hazard ）

  （注：也称为分支冒险或转移冒险（Branch Hazard） ）

BACK

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

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：继续执行分支指令的后续指令

    可加启发式规则：

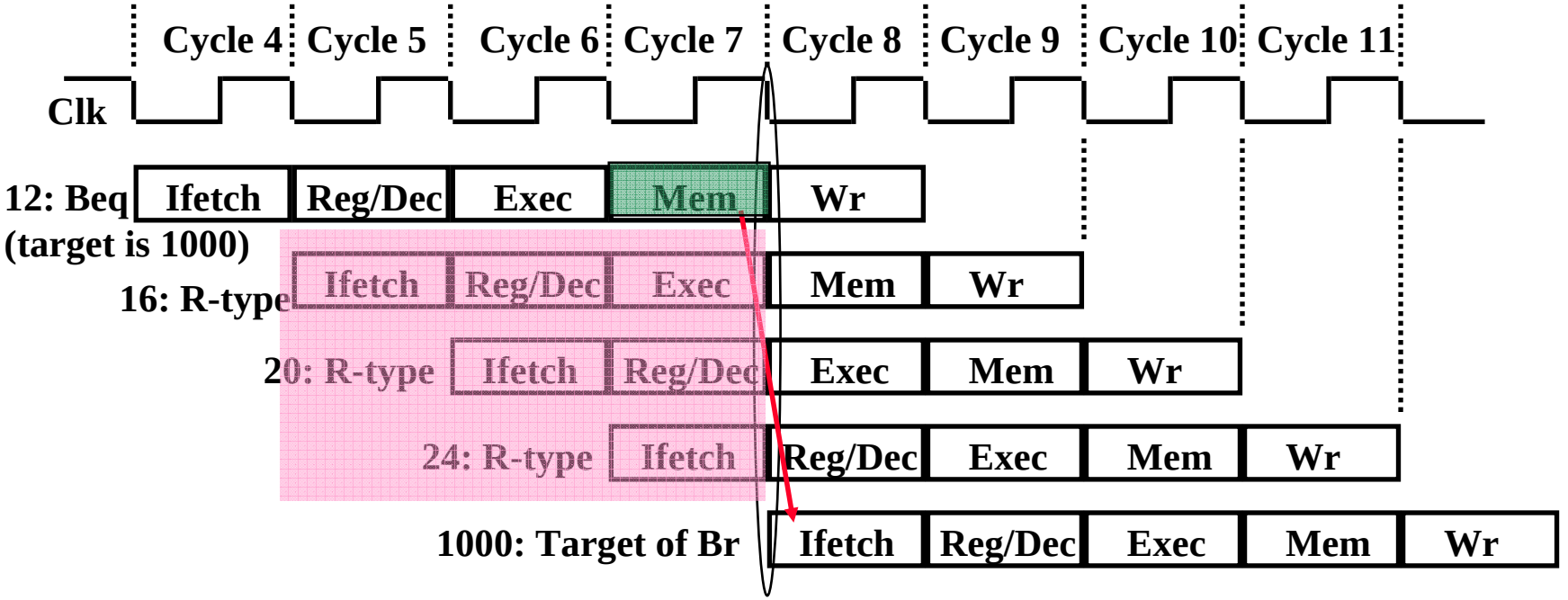
    在特定情况下总是预测满足(taken)，其他情况总是预测不满足
  - 预测失败时，需把流水线中三条错误预测指令丢弃掉
    - 将三条丢弃指令的控制信号值设置为0，使其后续过程中执行nop操作

    （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
  - 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
  - 可以把“是否转移”的确定工作提前，而不要等到MEM阶段才确定

那最早可以提前到哪个阶段呢？

SKIP

复习: Control Hazard现象



- 虽然Beq指令在第四周期取出，但:
  - “是否转移”在Mem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EXE 、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

BACK

简单（静态）分支预测方法

- 缩短分支延迟，减少错误预测代价
  - 可以通过调整“转移地址计算”和“分支条件判断”操作到ID阶段来缩短延迟
    - 将转移地址生成从MEM阶段移到ID阶段，可以吗？为什么？

    （是可能的：IF/ID流水段寄存器中已经有PC的值和立即数）
    - 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)来直接比较Rs和Rt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）
  - 当Branch=1并且Zero=1时，发生转移（taken）
  - 增加控制信号：IF.Flush=Branch and Zero，取值为1时，说明预测失败
  - 预测失败(条件满足)时，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

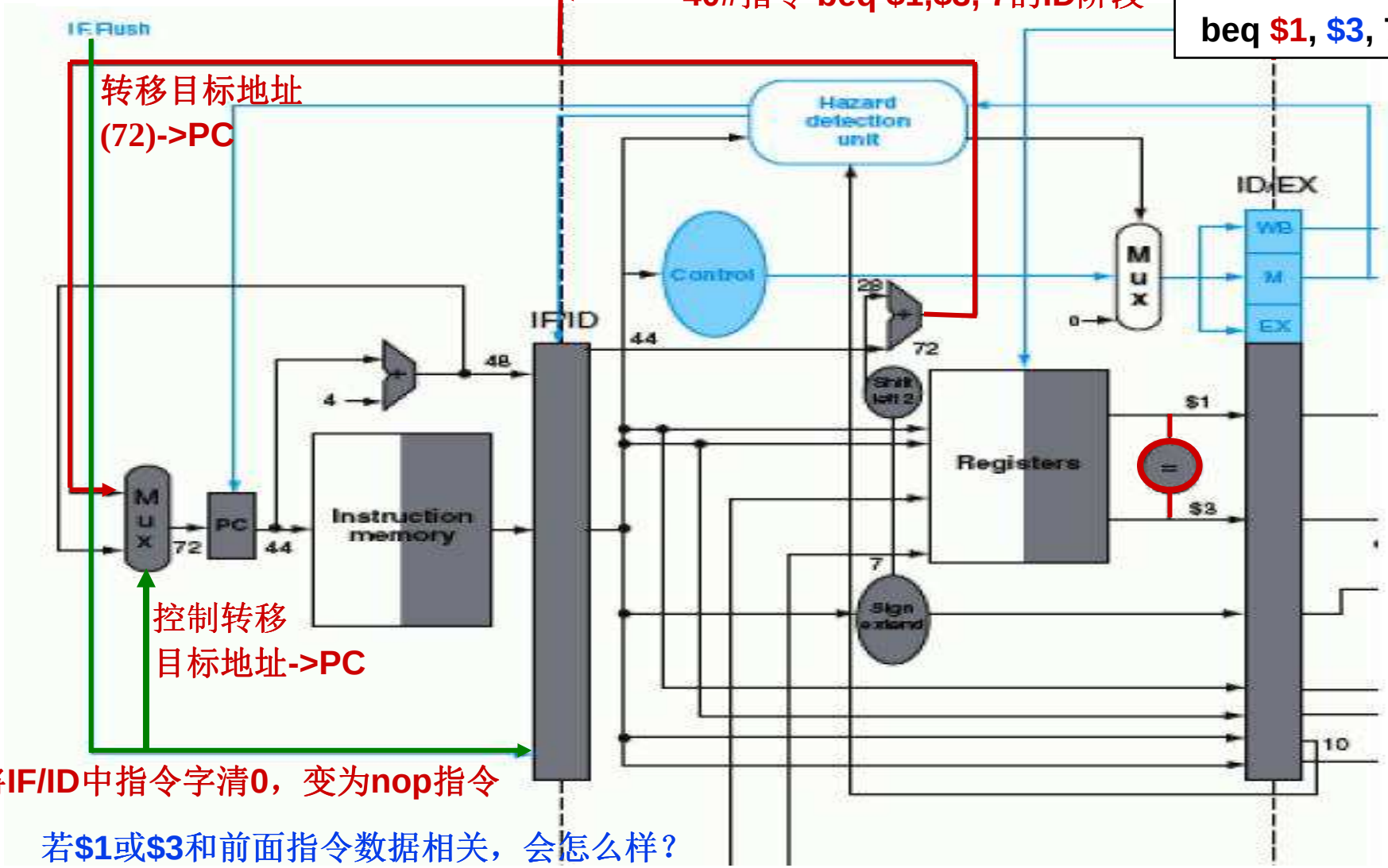
原来要清除三条指令，调整后只需要清除一条指令，因而只延迟一个时钟周期，每次预测错误减少了两个周期的代价！

带静态分支预测处理的数据通路

sub $3, $5, $1
add $1, $5, $2
beq $1, $3, 7

IF.Flush=Branch and Zero

40#指令 beq $1,$3, 7的ID阶段



若$1或$3和前面指令数据相关，会怎么样？

- 上上条指令的EXE段结果可转发回来进行判断
- 上条指令的EXE段结果来不及转发回来，引起1次阻塞!

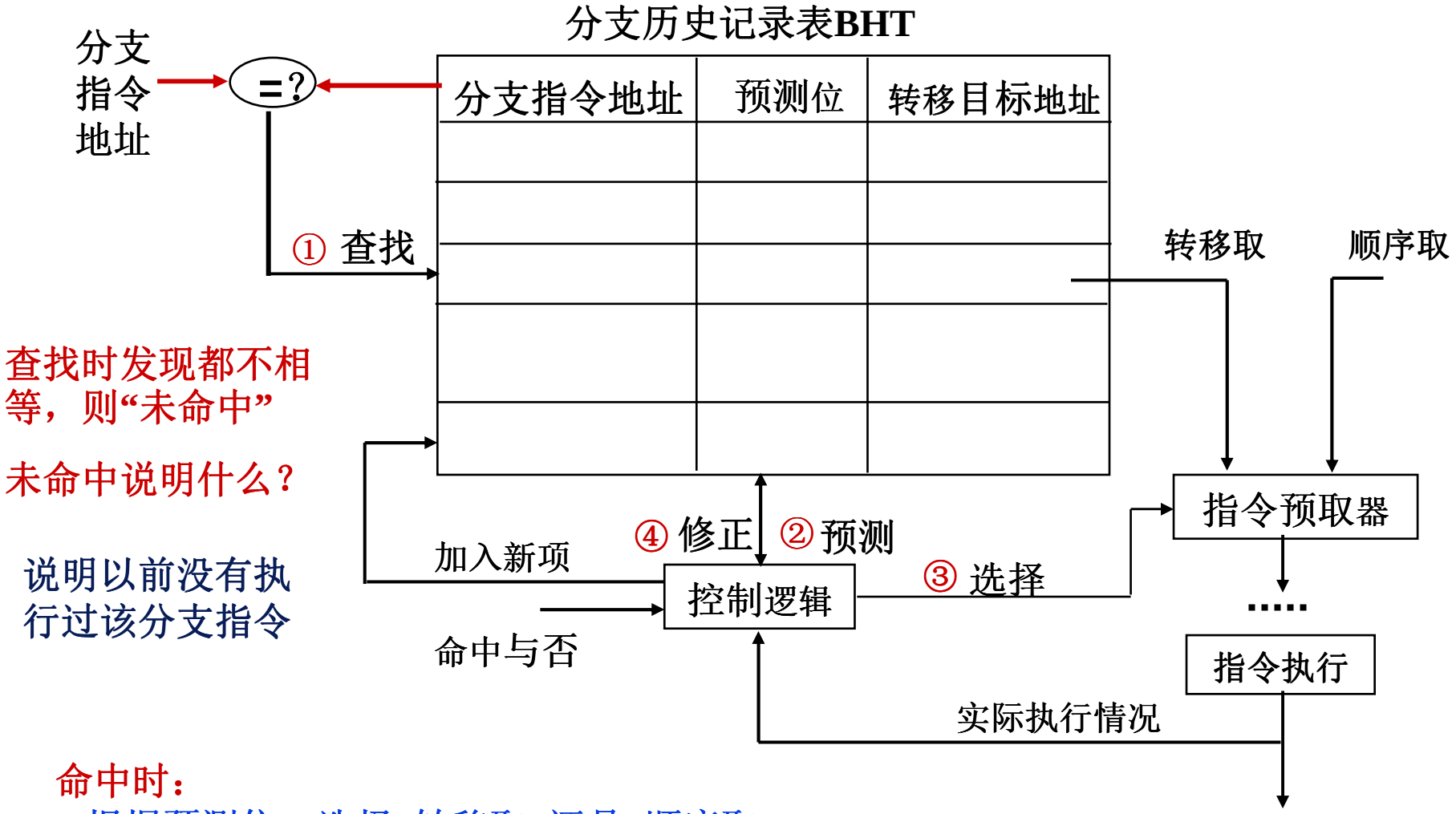
BACK

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，来预测下一次可能发生的转移
  - 预测后，在实际发生时验证并调整预测
  - 转移发生的历史情况记录在**BHT**中
    - 分支历史记录表**BHT（Branch History Table）**
    - 分支目标缓冲**BTB（Branch Target Buffer）**
  - 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**）

分支历史记录表BHT



命中时：

根据预测位，选择“转移取”还是“顺序取”

未命中时：

加入新项，并填入指令地址和转移目标地址、初始化预测位

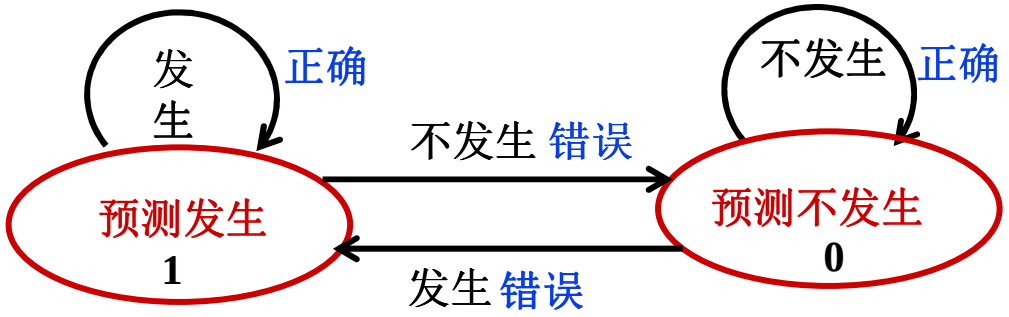
动态预测基本方法

- 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

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 ~ $1, $2, $3, $4 and base address of array is in $5

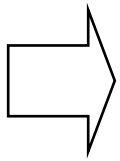
- 指令预取时，按照预测读取相应分支的指令
  - 预测发生时，选择“转移取”
  - 预测不发生时，选择“顺序取”
- 指令执行时，按实际执行结果修改预测位
  - 对照状态转换图来进行修改
  - 例如：对于一个循环分支
    - 若初始状态为0(再次循环时为0)，则第一次和最后一次都错
    - 若初始状态为1，则只有最后一次会错

```
Loop: add $7, $3, $3      ; i*2
      add $7, $7, $7      ; i*4
      add $7, $7, $5
      lw $6, 0($7)        ; $6=A[i]
      add $1, $1, $6      ; g= g+A[i]
      add $3, $3, $4
      bne $3, $2, Loop
        … …
```

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**N+1**次，
内循环中的分支指令共执行**N×(N+1)**次。

N=10, 分别90.9%和82.7%

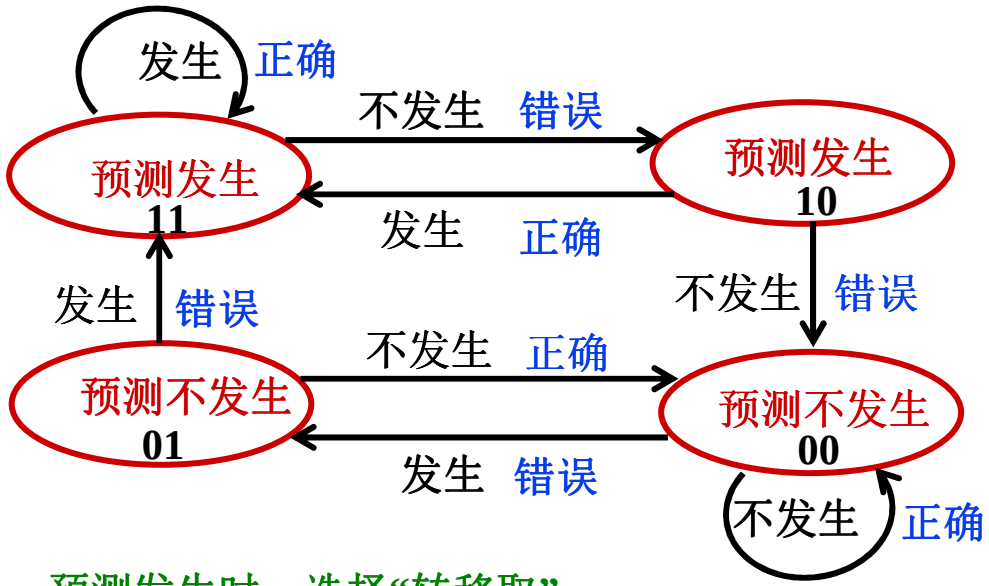
N=100, 分别99%和98%

预测位初始为**0**，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×(N–1)**次预测错误。

N越大准确率越高！

BACK

两位预测状态图




```
Loop: add $7, $3, $3      ; i*2
      add $7, $7, $7      ; i*4
      add $7, $7, $5
      lw $6, 0($7)        ; $6=A[i]
      add $1, $1, $6      ; g= g+A[i]
      add $3, $3, $4
      bne $3, $2, Loop
        … …
```

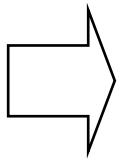
预测发生时，选择“转移取”

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
  - 例如，对于循环分支的预测（假定预测初始位为11）
    - 第一次：初始态为11（再次循环时状态为10），预测发生，实际也发生，正确
    - 中间：状态为“11”，预测发生，实际也发生，正确
    - 最后一次：状态为“11”，预测发生，但实际不发生，错

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**N+1**次，
内循环中的分支指令共执行**N×(N+1)**次。

N=10, 分别90.9%和90.9%

N=100, 分别99%和99%

预测位初始为**00**，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**，再进入内循环时，第一次预测正确，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！

BACK

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？

（假定时间片为2）

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

调度后

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

若分支条件判断和目标地址计算提前到ID阶段，则分支延迟时间片减少为1

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

BACK

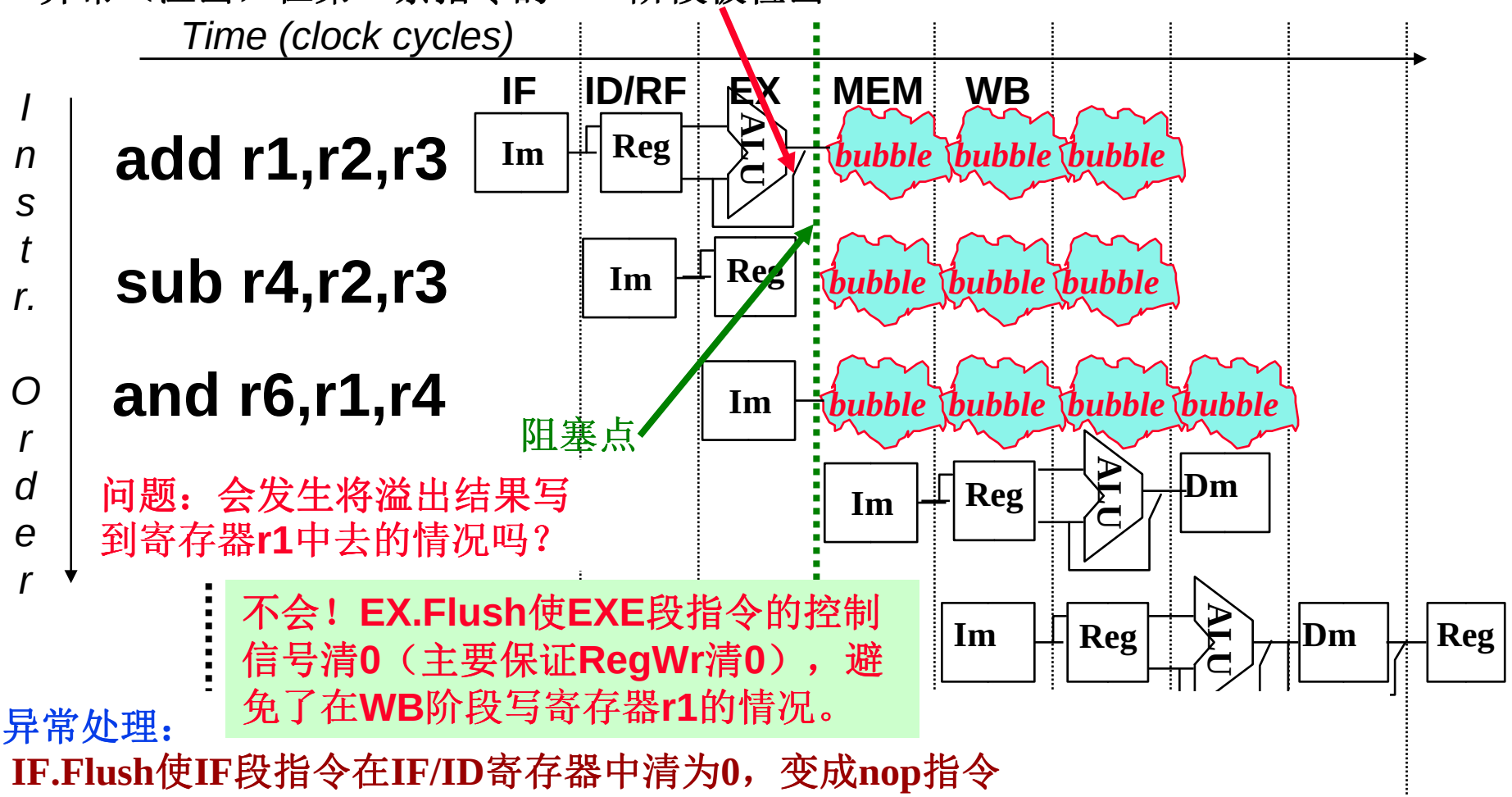
另一种控制冒险：异常和中断

- 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

    （记住：**MIPS**异常处理程序的首地址为**0x8000 0180**）
  - 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**
    - **0x8000 0180**送**PC**（从**0x8000 0180**处开始取指令）

异常的处理

- 异常（溢出）在第一条指令的EXE阶段被检出



异常处理：

IF.Flush使IF段指令在IF/ID寄存器中清为0，变成nop指令

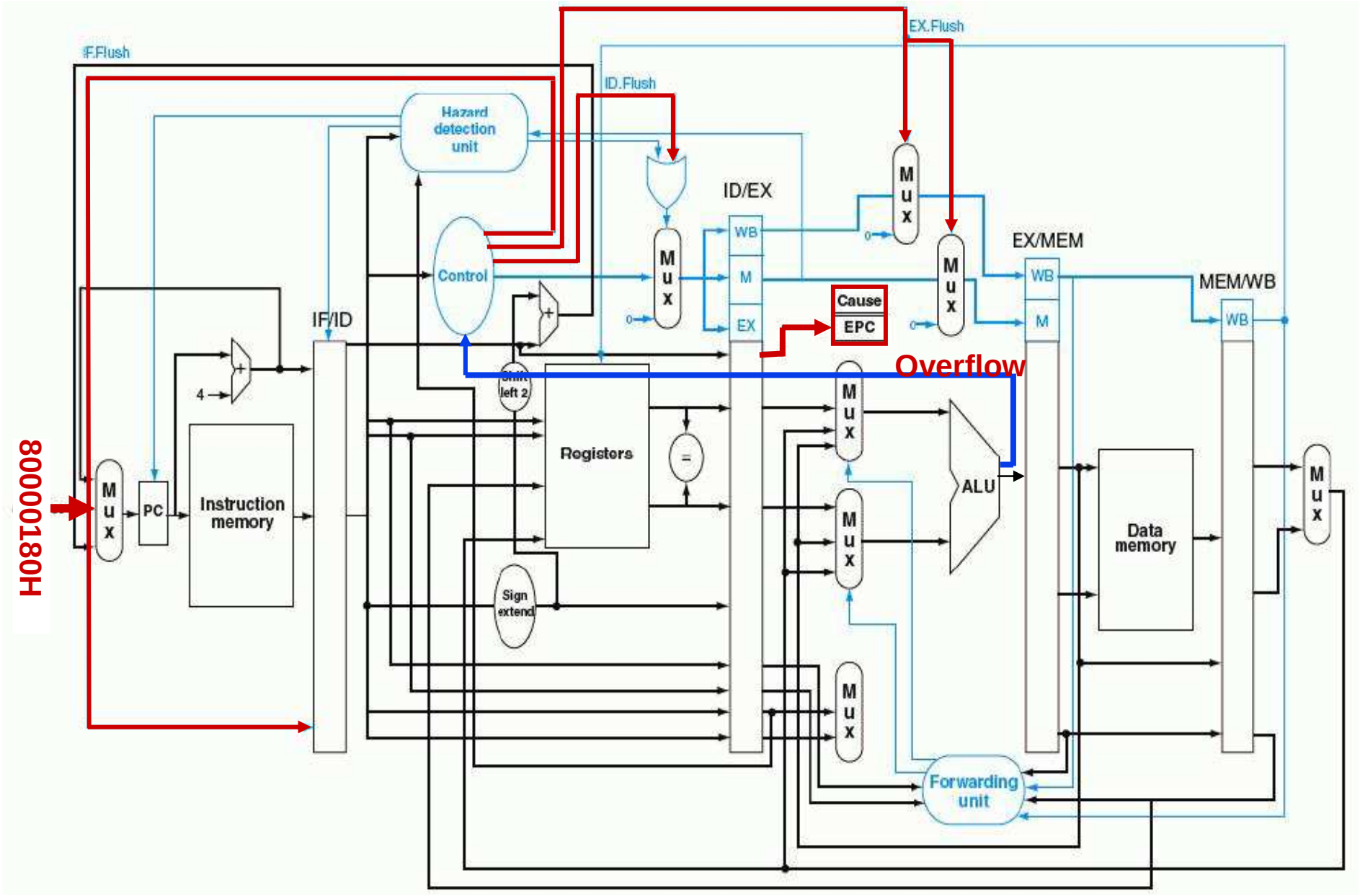
ID.Flush与数据冒险阻塞检测信号相或(or)后，使ID段指令的控制信号清0

EX.Flush使EX段指令的控制信号清0

关中断，并将断点（可能是PC、可能是PC+4）保存到EPC中

将0x8000 0180作为PC的一个输入，并控制PC输入端的多路选择器

带异常处理的流水线数据通路

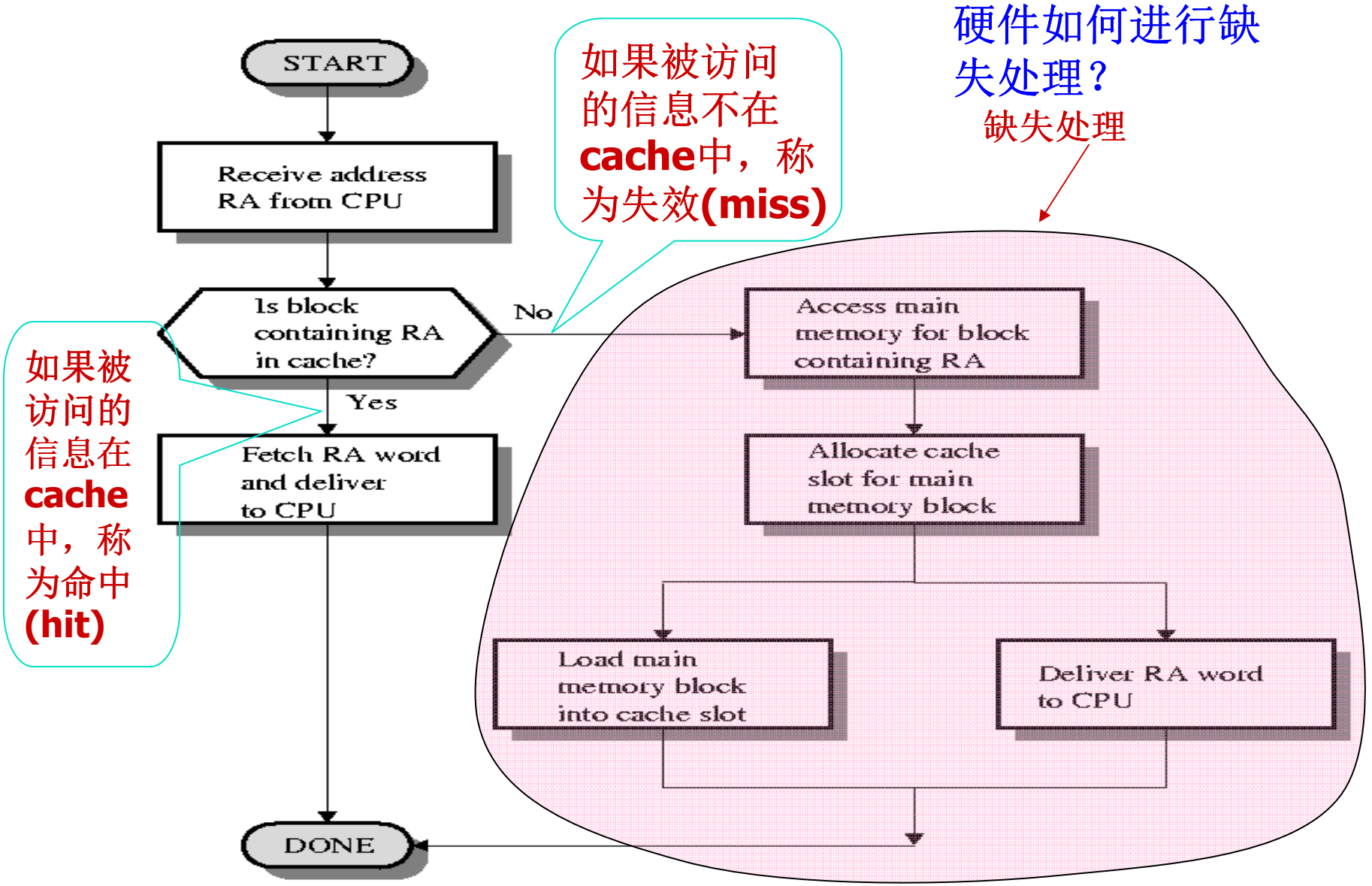


流水线方式下的异常处理的难点问题

- 流水线中同时有5条指令，到底是哪一条发生异常？
  - 根据异常发生的流水段可确定是哪条指令，因为各类异常发生的流水段不同
    - “溢出”在EXE段检出
    - “无效指令”在ID段检出
    - “除数为0”在ID段段检出
    - “无效指令地址”在IF段检出
    - “无效数据地址”在Load/Store指令的EXE段检出
- 外部中断与特定指令无关，如何确定处理点？
  - 可在IF段或WB段中进行中断查询，需要保证当前WB段的指令能正确完成，并在有中断发生时，确保下个时钟开始执行中断服务程序
- 检测到异常时，指令已经取出多条，当前PC的值已不是断点，怎么办？
  - 指令地址存放在流水段R，可把这个地址送到EPC保存，以实现精确中断

  （非精确中断不能提供准确的断点，而由操作系统来确定哪条指令发生了异常）
- 一个时钟周期内可能有多个异常，该先处理哪个？
  - 异常：检出异常后，存到专门寄存器中（前面指令优先级高？）
  - 中断：在中断查询程序或中断优先级排队电路中按顺序查询
- 系统中只有一个EPC，多个中断发生时，一个EPC不够放多个断点，怎么办？
  - 总是把优先级最高的送到EPC中
- 在异常处理过程中，又发生了新的异常或中断，怎么办？
  - 利用中断屏蔽和中断嵌套机制来处理　　后面三个问题在第九章中详细介绍！

复习：Cache 的操作过程



Cache缺失处理会引起流水线阻塞（停顿）

- 在使用Cache的系统中，数据通路中的IM和DM分别是Code Cache和Data Cache
- CPU执行指令过程中，取指令或取数据时，如果发生缺失，则指令执行被阻塞
- Cache缺失的检测（如何进行的？记得吗？）
  - Cache中有相应的检测线路（地址高位与Cache标志比较）
- 阻塞处理过程
  - 冻结所有临时寄存器和程序员可见寄存器的内容（即：使整个计算机阻塞）
  - 由一个单独的控制器处理Cache缺失，其过程 (假定是指令缺失) 还记得吗？：
    - 把发生缺失的指令地址（PC- 4）所在的主存块首址送到主存
    - 启动一次“主存块读”操作，并等待主存完成一个主存块(Cache行)的读操作
    - 把读出的一个主存块写到Cache对应表项的数据区

      （若对应表项全满的话，还要考虑淘汰掉一个已在Cache中的主存块）
    - 把地址高位部分（标记）写到Cache的“tag”字段，并置“有效位”
    - 重新执行指令的第一步：“取指令”
  - 若是读数据缺失，其处理过程和指令缺失类似
    - 从主存读出数据后，从“取数”那一步开始重新执行就可以了
  - 若是写数据缺失，则要考虑用哪种“写策略”解决“一致性”问题
- 比数据相关或分支指令引起的流水线阻塞简单：只要保持所有寄存器不变
- 与中断引起的阻塞处理不同：不需要程序切换

TLB缺失和缺页也会引起流水线阻塞

- TLB缺失处理
  - 当TLB中没有一项的虚页号与要找的虚页号相等时，发生TLB miss
  - TLB miss说明可能发生以下两种情况之一：
    - 页在内存中：只要把主存中的页表项装载到TLB中
    - 页不在内存中(缺页)：OS从磁盘调入一页，并更新主存页表和TLB
- 缺页（page fault）处理
  - 当主存页表的页表项中“valid”位为“0”时，发生page fault
  - Page fault是一种“故障”异常，按以下过程处理（MIPS异常处理）
    - 在Cause寄存器置相应位为“1”
    - 发生缺页的指令地址（PC- 4）送EPC
    - 0x8000 0180(异常查询程序入口)送PC
      - 执行OS的异常查询程序，取出Cause寄存器中相应的位分析，得知发生了“缺页”，转到“缺页处理程序”执行
  - page fault一定要在发生缺失的存储器操作时钟周期内捕获到，并在下个时钟转到异常处理，否则，会发生错误。
    - 例：lw $1, 0($1) ，若没有及时捕获“异常”而使$1改变，则再重新执行该指令时，所读的内存单元地址被改变，发生严重错误！

处理Cache缺失和缺页的不同之处在哪里？

哪种要进行程序切换？

（缺页）异常处理时要考虑的一些细节

- 缺页异常结束后，回到哪里继续执行？
  - 指令缺页：重新执行发生缺页的指令
  - 数据缺页：
    - 简单指令（仅一次访存）：强迫指令结束，重新执行缺页指令
    - 复杂指令（多次访存）：可能会发生多次缺页，指令被中止在中间某个阶段，缺页处理后，从中间阶段继续执行；因而，需要能够保存和恢复中间机器状态
- 异常发生后，又发生新的异常，怎么办？
  - 在发现异常、转到异常处理程序中，若在保存正在运行进程的状态时又发生新的异常，则因为要处理新的异常，会把原来进程的状态和保存的返回断点破坏掉，所以，应该有一种机制来“禁止”响应新的异常处理
  - 通过“中断/异常允许”状态位（或“中断/异常允许”触发器）来实现
  - “中断/异常允许”状态位置1，则“开中断”（允许异常），清0则“关中断 ”（ 禁止异常）
  - 也可由OS通过管态指令来设置该位的状态

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）单周期方式：每条指令在一个固定长度的时钟周期内完成

（2）多周期方式：每类指令时钟数:取数**-5**，存数**-4**，**ALU-4**，分支**-3**，跳转**-3**

（3）流水线方式：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%**；不考虑异常、中断和访问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

三种方式的指令条数都一样，所以只要比较CPI和时钟周期长度即可。

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

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

三种处理器实现方式的比较

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=5x25%+4x10%+4x52%+3x11%+3x2%=4.12

所以， N条指令的执行时间为4.12x200xN=824N(ps)

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：.75x1+.25x2=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：1.5x25%+1x10%+1x52%+1.25x11%+2x2%=1.17

所以， N条指令的执行时间为1.17x200xN=234N(ps)

流水线冒险对程序性能的影响

- 结构冒险对浮点运算的性能影响较大，因为浮点运算单元不能有效被流水化，可能造成运算单元的资源冲突
- 控制冒险更多出现在整数运算程序中，因为分支指令对应于循环或选择结构，大多由整数运算结果决定分支
- 数据冒险在整数运算程序和浮点运算程序中都一样
  - 浮点程序中的数据冒险容易通过编译器优化调度来解决
    - 分支指令少
    - 数据访问模式较规则
  - 整数程序的数据冒险不容易通过编译优化调度解决
    - 分支指令多
    - 数据访问模式不规则
    - 过多使用指针

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