南京大学
NANJING UNIVERSITY

指令流水线

殷亚凤
智能软件与工程学院
苏州校区南雍楼东区225
yafeng@nju.edu.cn，https://yafengnju.github.io/

指令流水线

- 流水线概述
- 流水线处理器的实现
- **流水线冒险及其处理**
- 高级流水线技术

流水线冒险及其处理

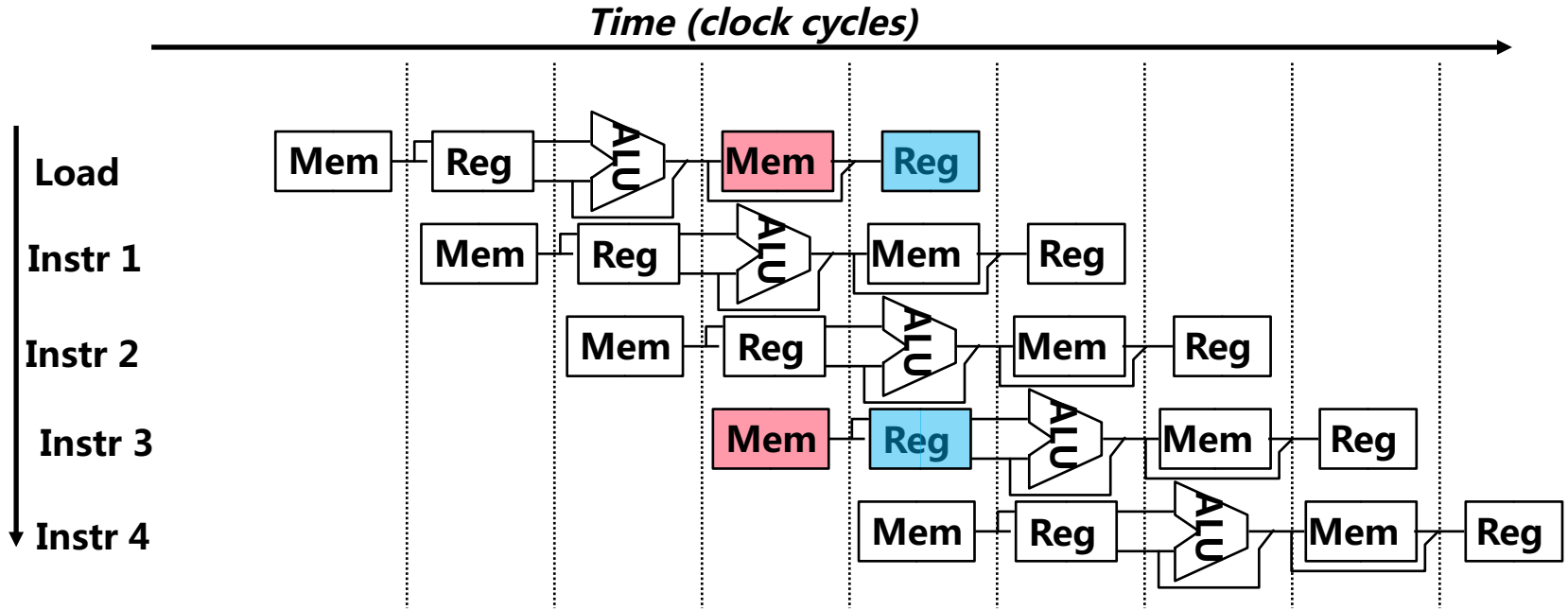
- **冒险(Hazards)：指流水线遇到无法正确执行后续指令或执行了不该执行的指令**
  - **结构冒险(Structural hazards)** (硬件资源冲突):

    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
  - **数据冒险(Data hazards)** (数据冲突):

    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
    - 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
  - **控制冒险(Control hazards)** (指令执行顺序改变):

    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
    - 采用静态或动态分支预测
    - 编译程序优化指令顺序(分支延迟)

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！

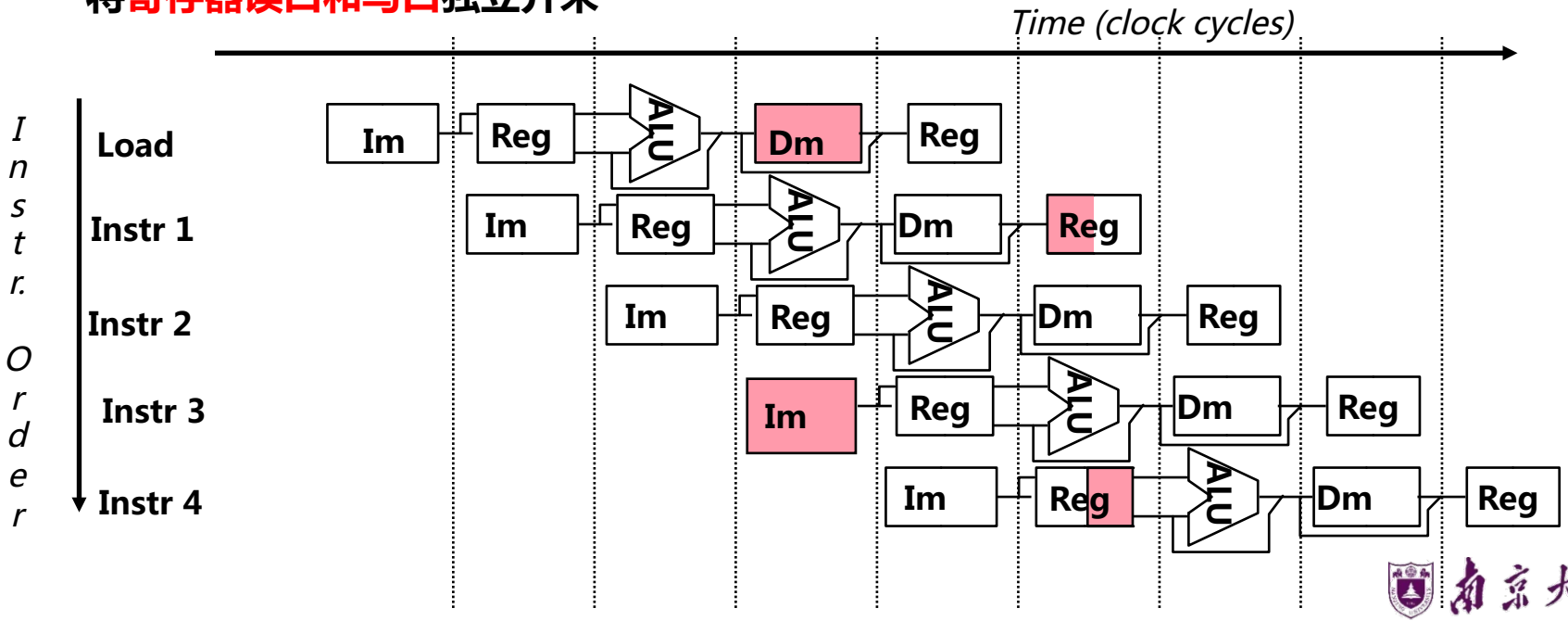
如果不对寄存器堆的写口和读口独立设置的话，则发生冲突！

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！）
- **将指令存储器(Im)和数据存储器(Dm)分开**
- **将寄存器读口和写口独立开来**



数据冒险现象

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！

三类数据冒险现象:

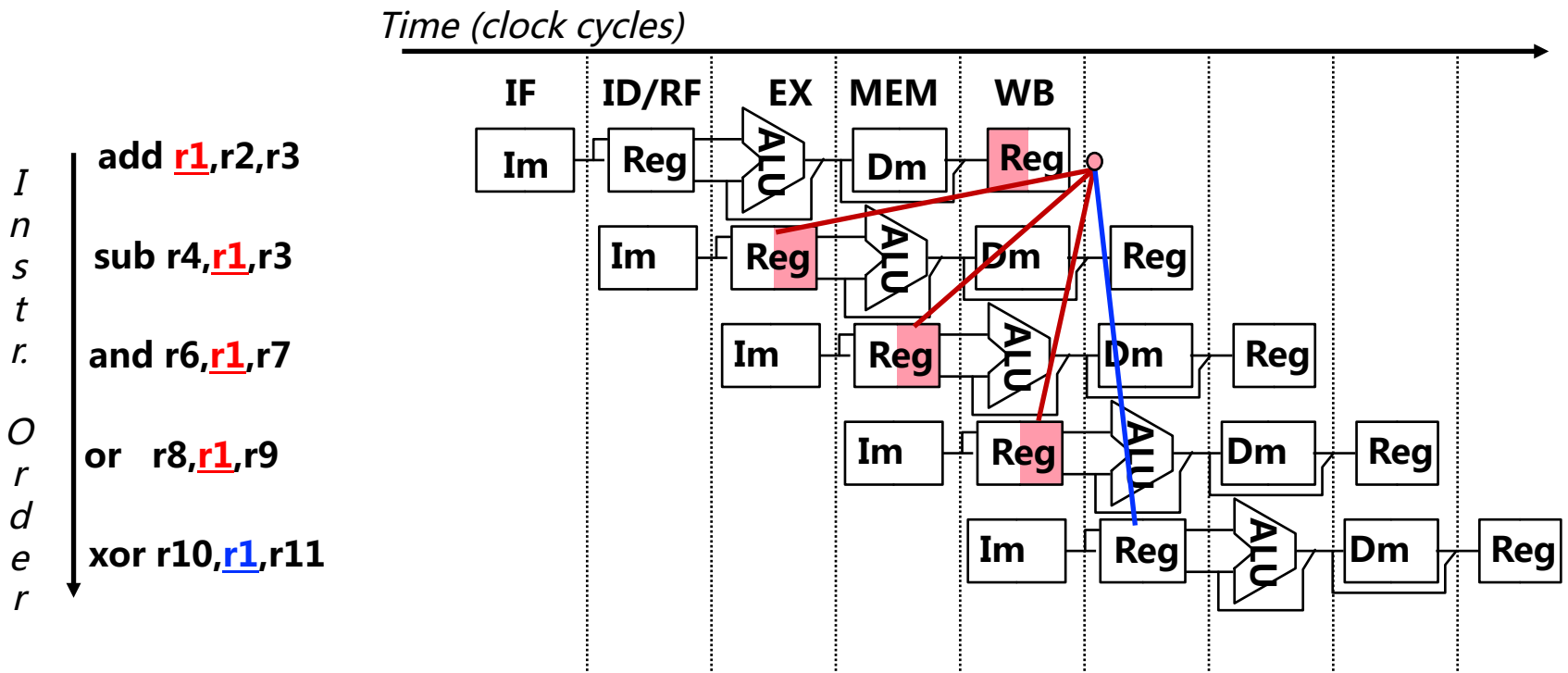
RAW: 写后读（基本流水线中经常发生，如上例）

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：r1



最后一条指令的r1才是新的值！

如何解决这个问题？

南京大学 NANJING UNIVERSITY

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）
  - 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4：转发（Forwarding或Bypassing 旁路）技术
  - 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！)**
- 方法5：编译优化：调整指令顺序（不能解决所有数据冒险）

实现“转发”和“阻塞” 要修改数据通路：

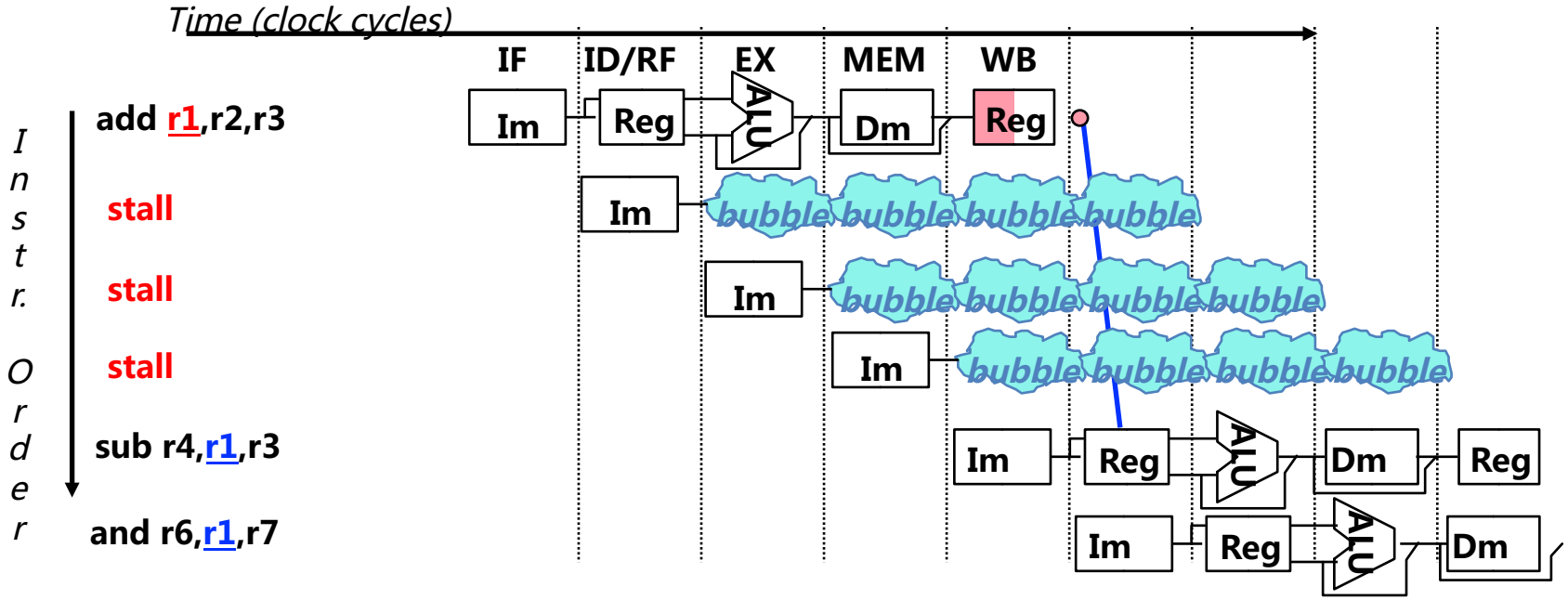
（1）检测何时需要“转发” ，并控制实现“转发”

（2）检测何时需要“阻塞” ，并控制实现“阻塞”

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！**

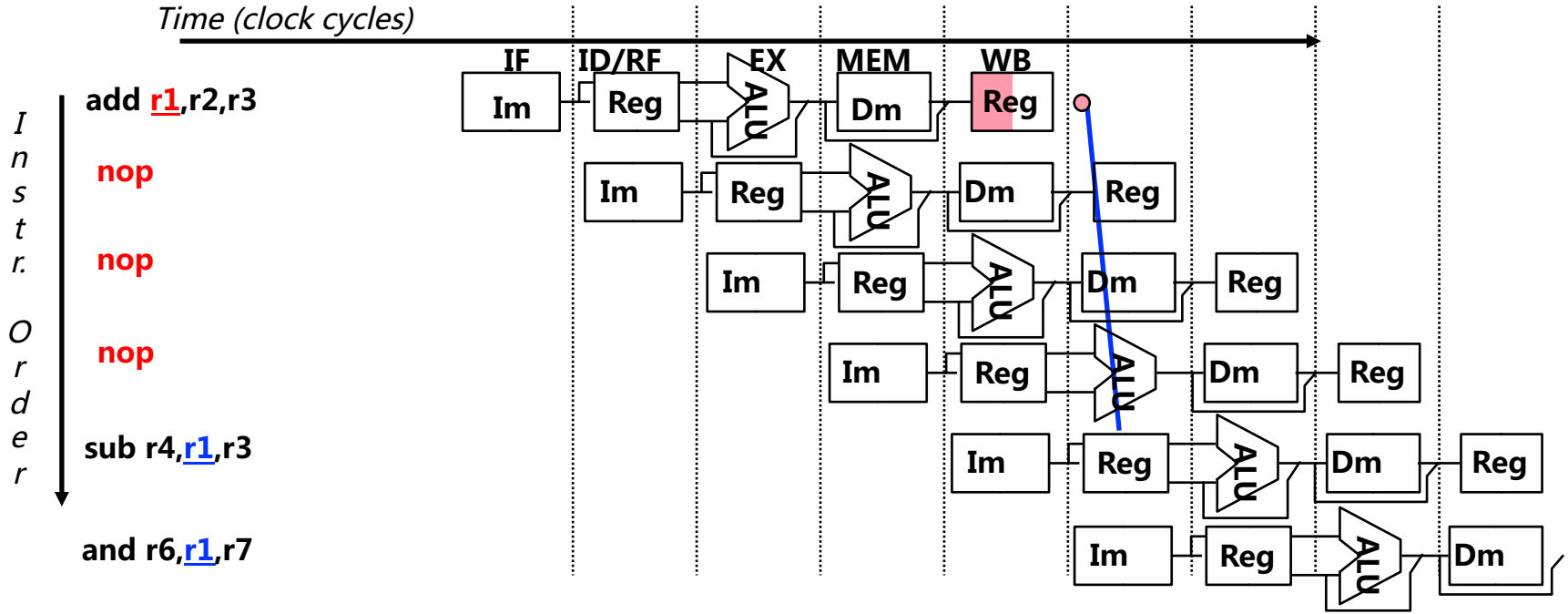
  这种做法称为流水线阻塞，也称为**插入“气泡Bubble”**



- 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。

数据冒险-方案2：软件上插入无关指令

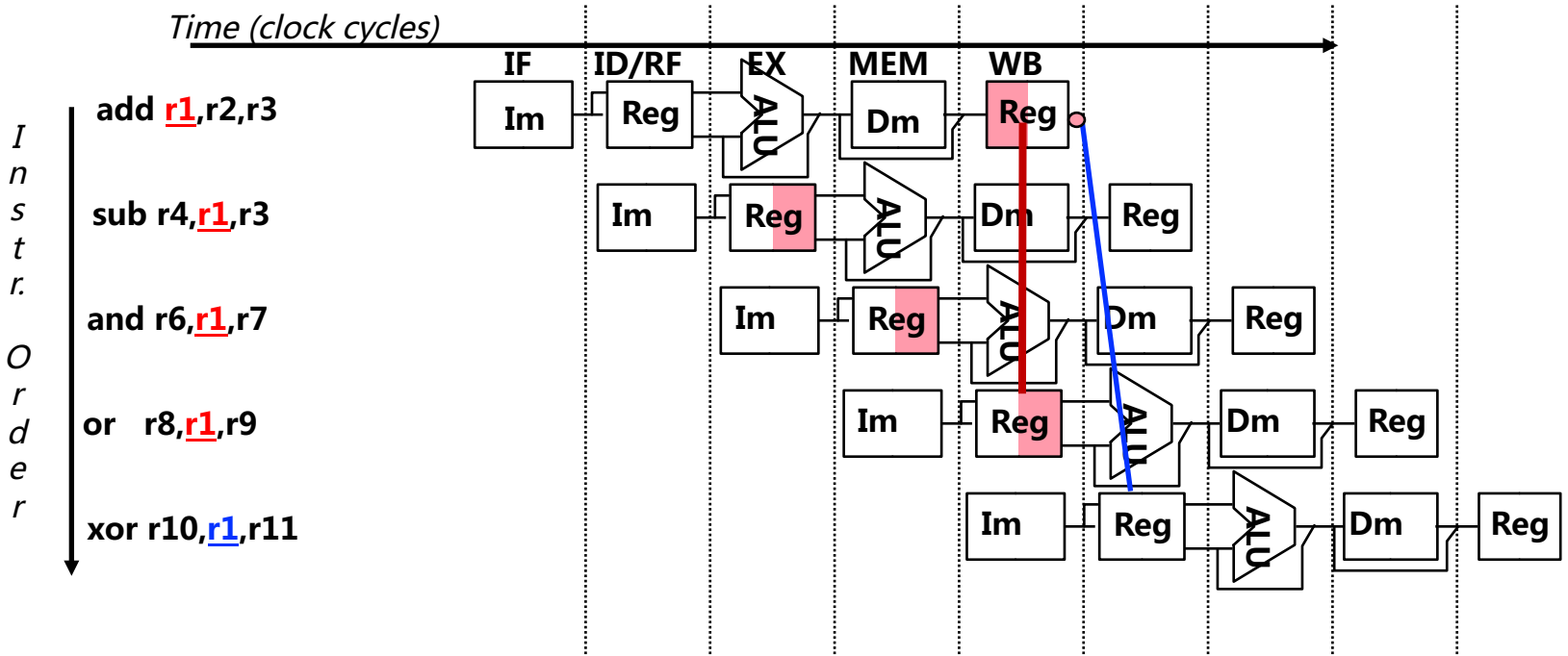
- **由编译器插入三条NOP指令，浪费三条指令的空间和时间，是最差的做法。**
  (好处：数据通路简单，即无需改数据通路)



与方案1比，哪个更快？　　一样，都是多三个时钟周期！

数据冒险-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！

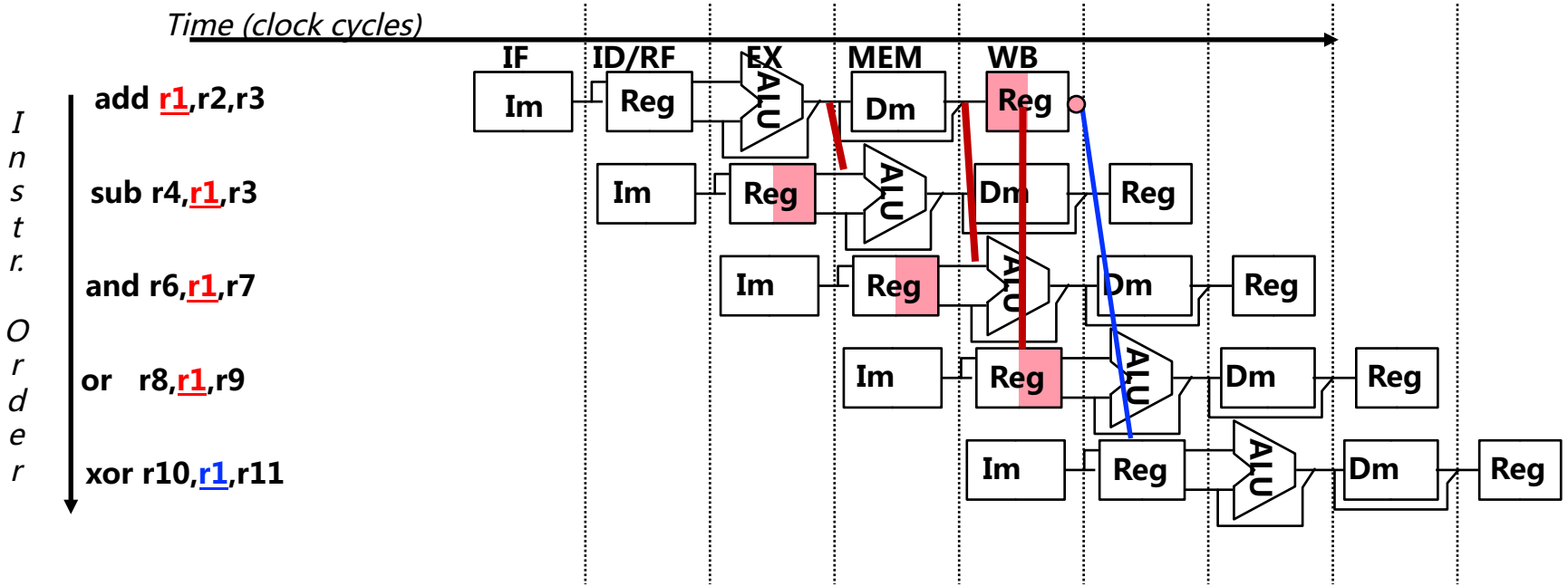


寄存器**写口/读口**分别在**前/后半周期**进行操作，使写入数据被直接读出。

但是，只能解决部分数据冒险！

数据冒险-方案4：利用数据通路中的中间数据：转发+阻塞

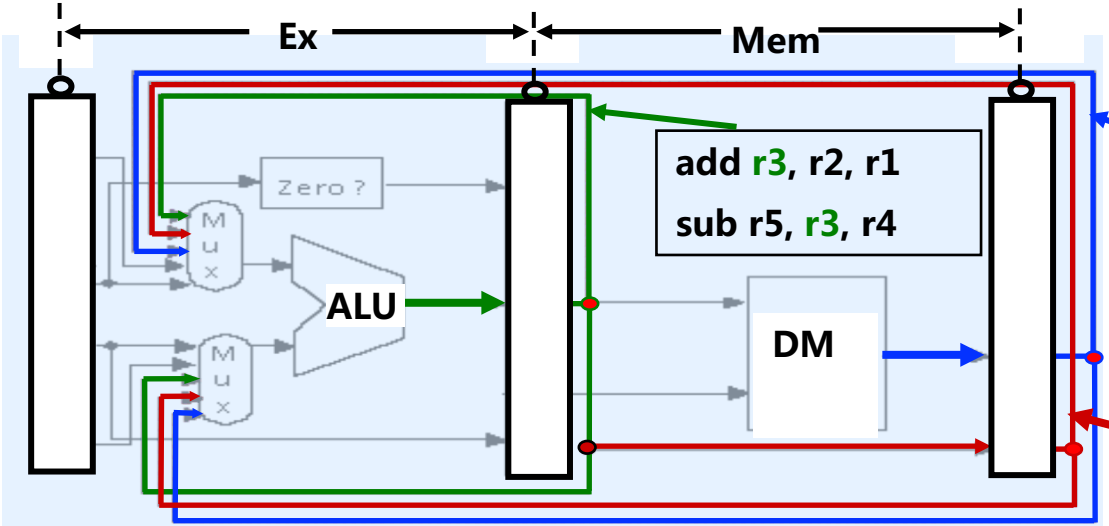
- 仔细观察后发现：流水段寄存器中已有需要的值r1！ 在哪个流水段R中？



把数据从流水段寄存器中直接取到ALU的输入端 ← 称为转发或旁路

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

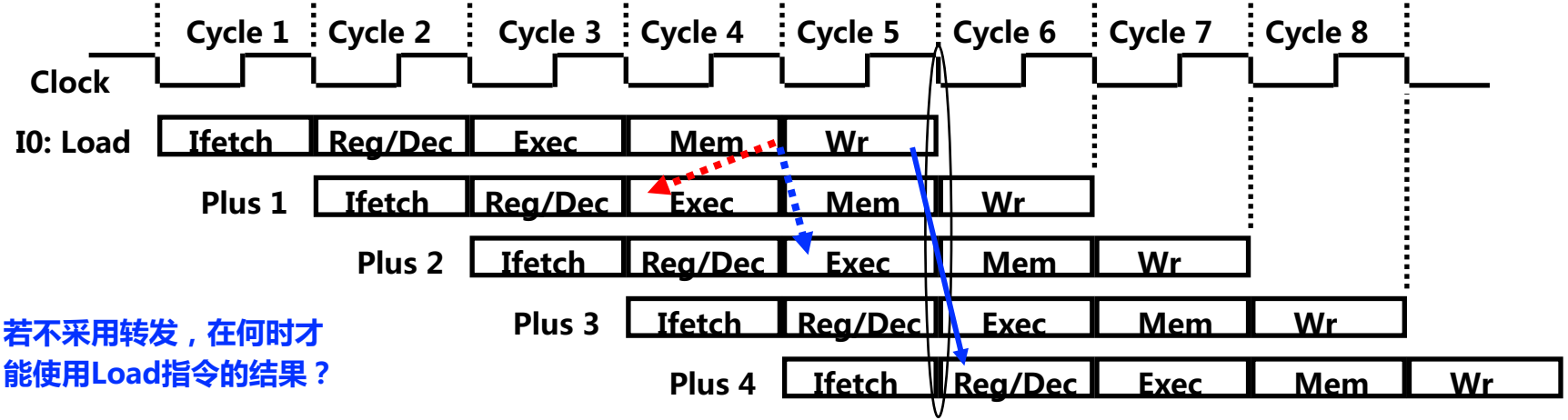


如果指令序列为

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？
（不能！）

回顾：Load指令引起的延迟现象

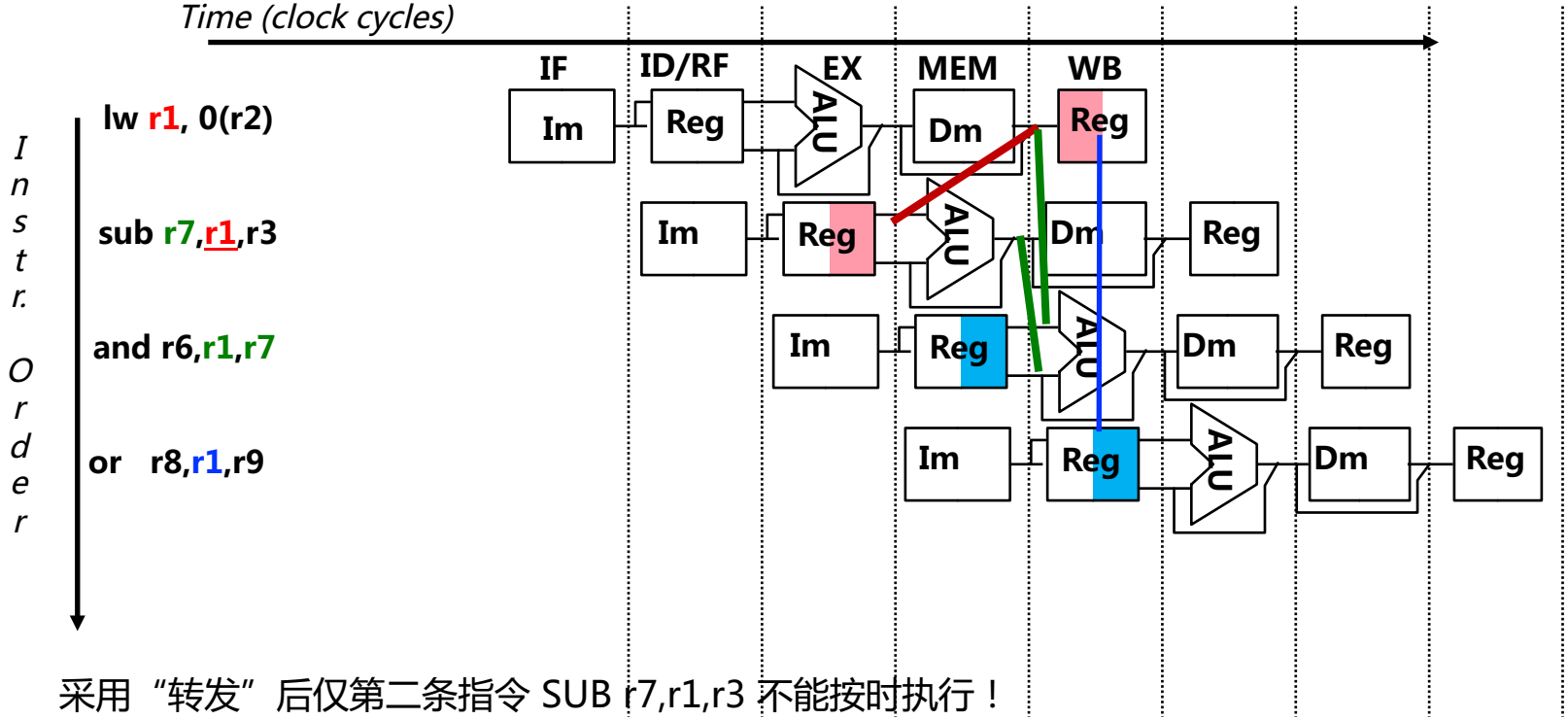


- **Load指令最早在哪个流水线寄存器中开始有后续指令需要的值？**

  - 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)”

“转发”技术使Load-use冒险只需延迟一个周期

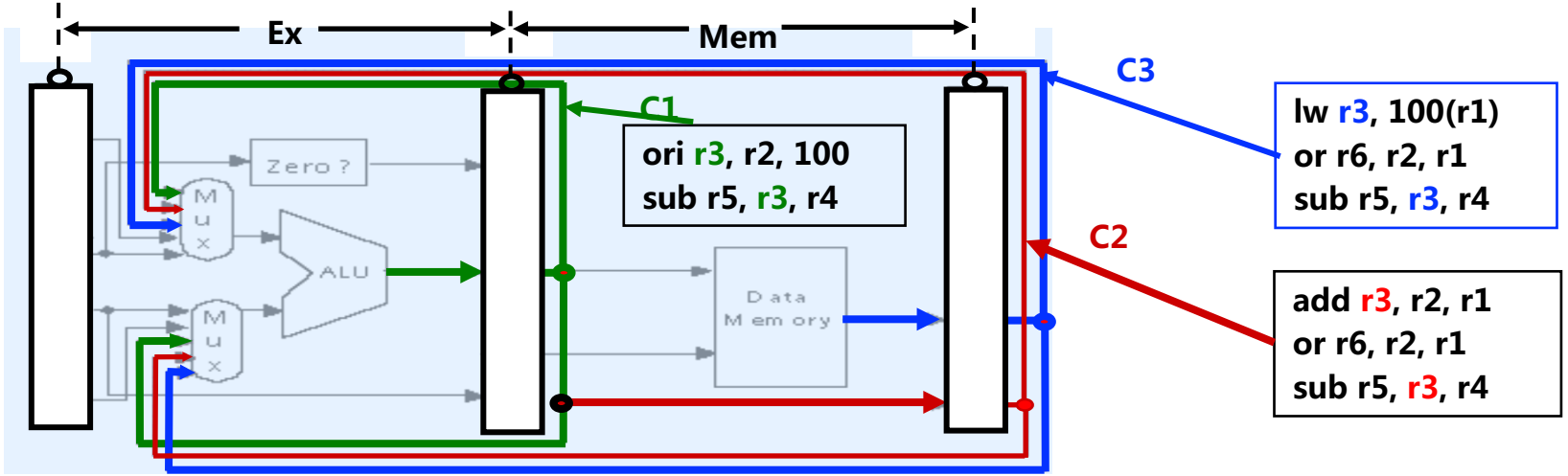


采用“转发”后仅第二条指令 SUB r7,r1,r3 不能按时执行！

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”

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

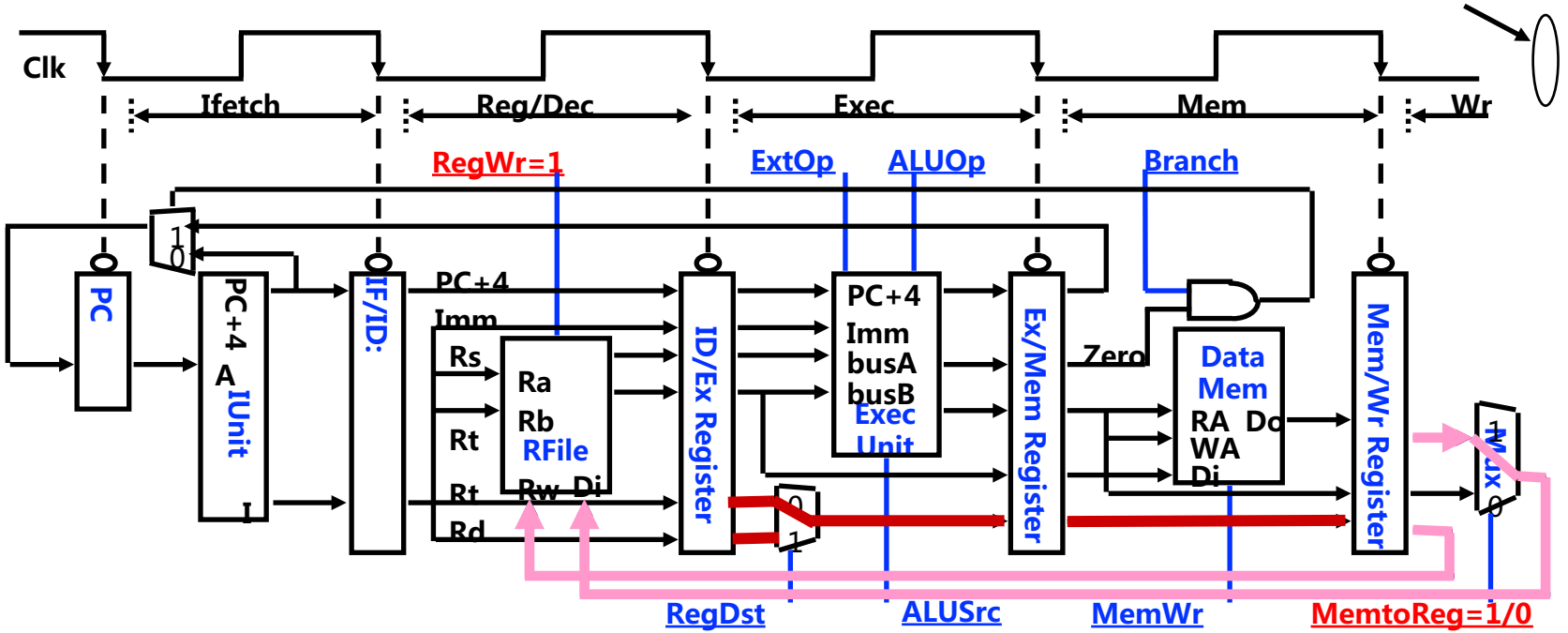
C2(a): MEM/WB. RegisterRd=ID/EX. RegisterRs

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
add r4, r3, r2
sub r5, r3, r2
```

因为beq指令无需写结果，故不能进行转发！

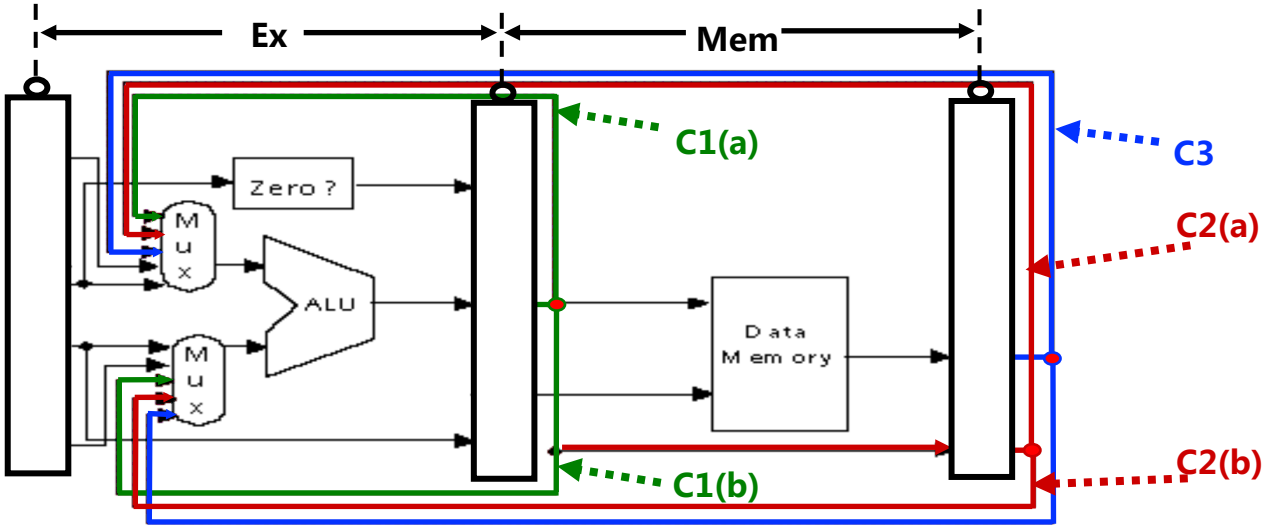
转发条件的进一步完善

- 以下两种情况下，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Rt
    - 即：EX / MEM 或 MEM / WB 流水段寄存器的RegWr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.RegWr
    and EX/MEM. RegisterRd ≠ 0
    and EX/MEM. RegisterRd=ID/EX. RegisterRs
  - **C1(b)**: EX/MEM.RegWr
    and EX/MEM. RegisterRd ≠ 0
    and EX/MEM. RegisterRd=ID/EX. RegisterRt
  - **C2(a)**: MEM/WB.RegWr
    and MEM/WB. RegisterRd ≠ 0
    and MEM/WB. RegisterRd=ID/EX. RegisterRs
  - **C2(b)**: MEM/WB.RegWr
    and MEM/WB. RegisterRd ≠ 0
    and MEM/WB. RegisterRd=ID/EX. RegisterRt

```
beq r3, r2, 100
add r4, r3, r2
sub r5, r3, r2
```

转发路径和转发条件

- 加MUX，使流水段寄存器值返送ALU输入端



C1反映本条指令和随后指令间的相关关系

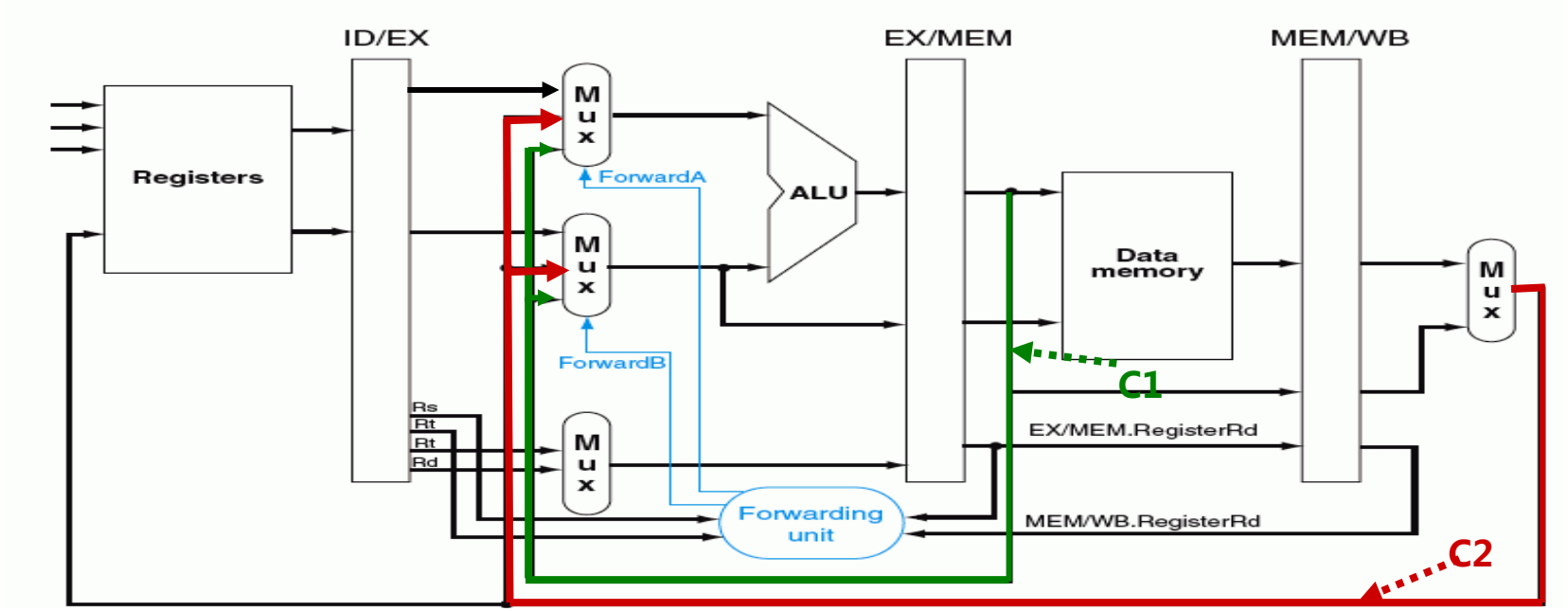
C2反映本条指令和随后第二条指令间的相关关系

红线和蓝线可以合并，在原数据通路中确实是合并在一起的。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, 无需另有一个检测条件C3! C3就是C2！
C1和C2分别反映哪两条指令的关系呢？

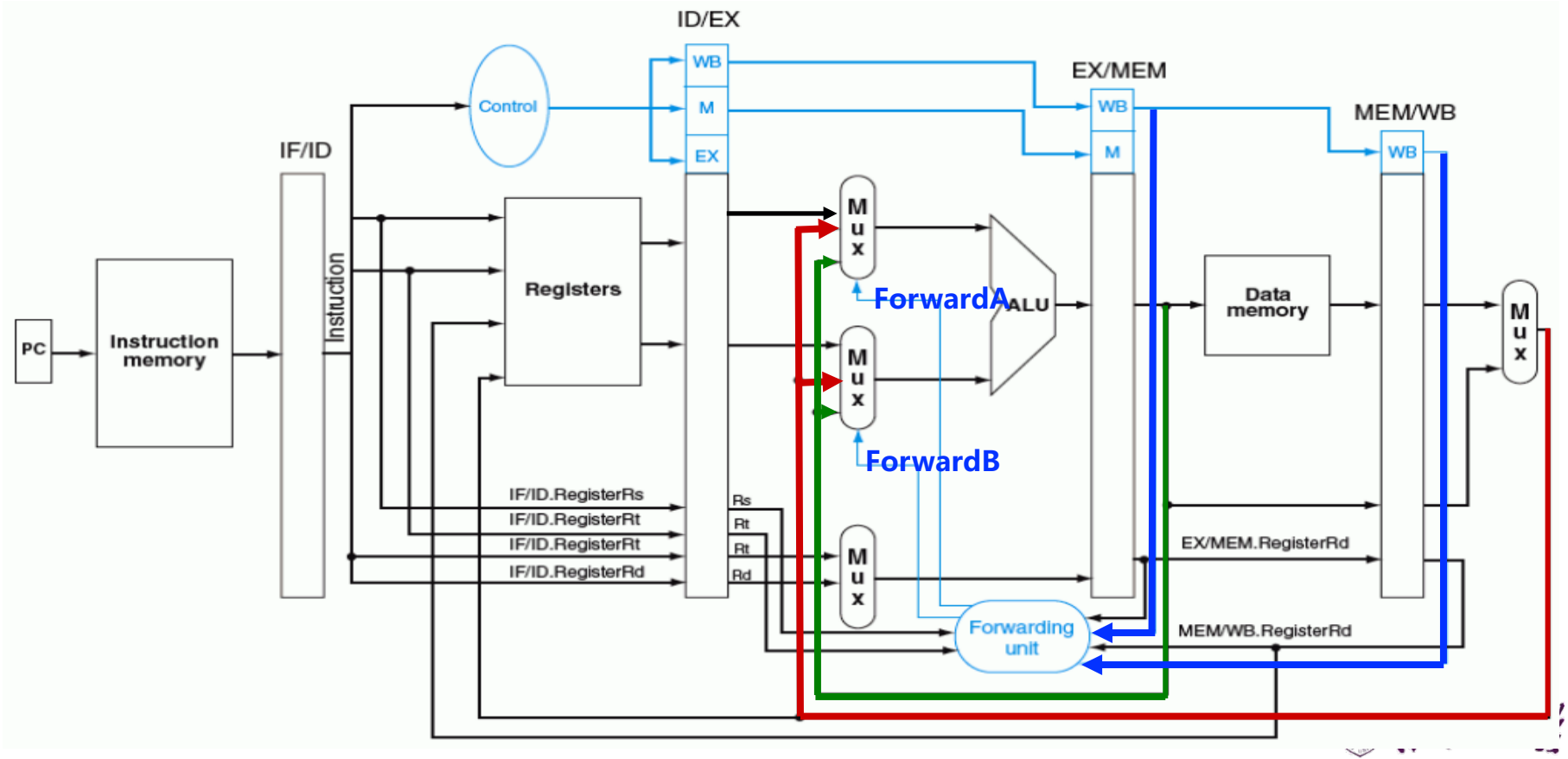
转发路径和转发条件

ForwardA (ForwardB) = { 01 当c2(a)=1 或C2(b)=1 时
10 当c1(a)=1或C1(b)=1 时



“转发检测”部件中缺何条件？ MEM/WB.RegWr=1、EX/MEM.RegWr=1

带转发的流水线数据通路



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(a)=1 or c2(b)=1 时; 10 当c1(a)=1 or c1(b)=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(a)和C2(b)为：

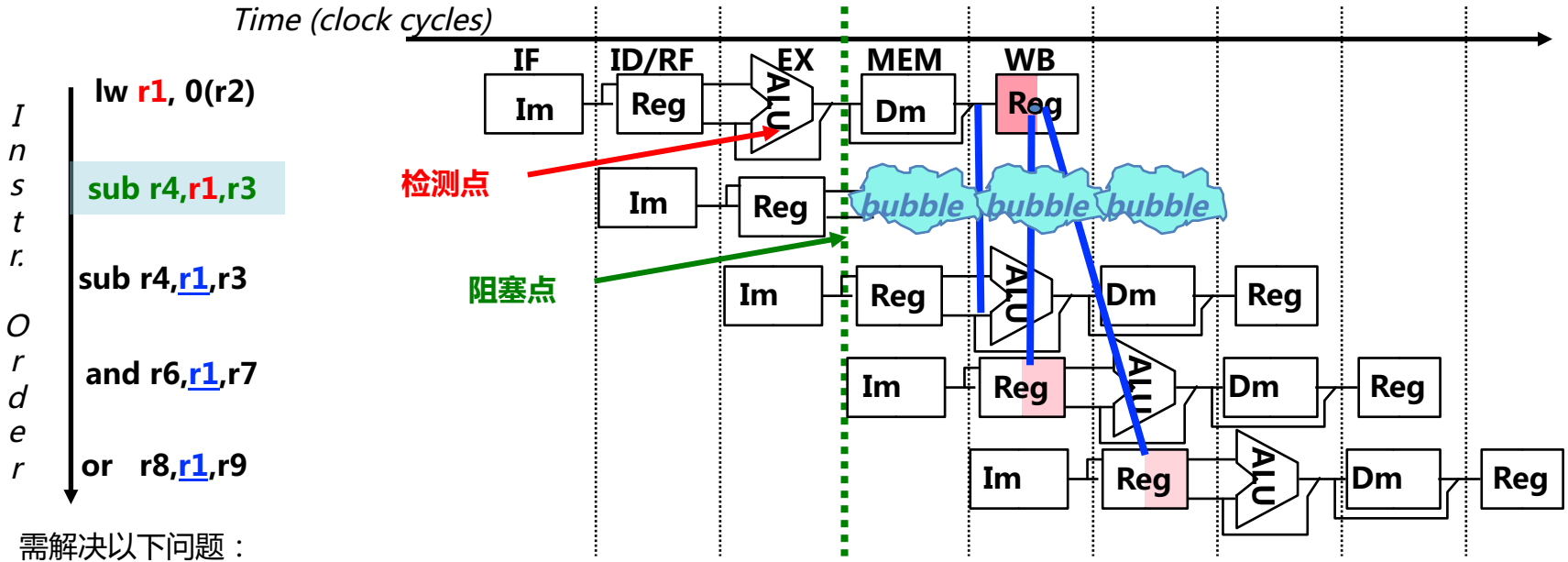
C2(a)=MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) **and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)** and (MEM/WB.RegisterRd=ID/EX.RegisterRs)

C2(b)=MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) **and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)** and (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

硬件阻塞方式(Load-use Data Hazard)



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)

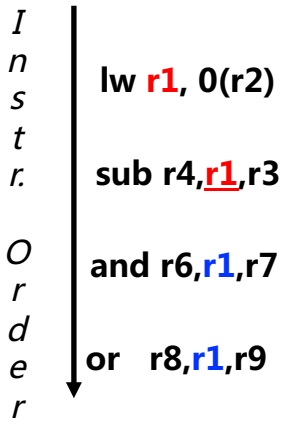
→ 前面指令为Load 并且

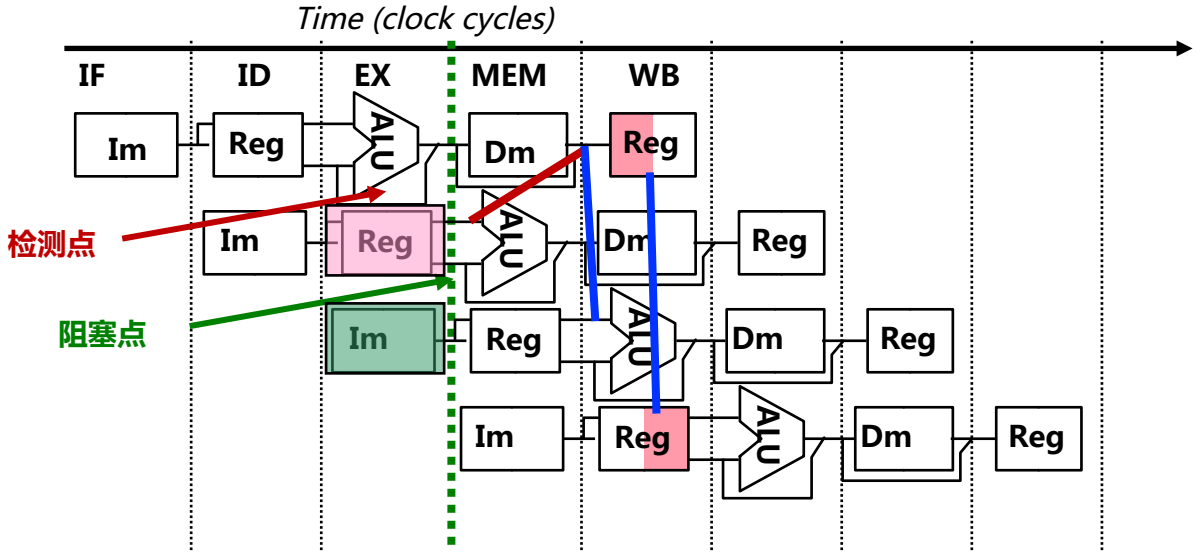
→ 前面指令的目的寄存器等于当前刚取出指令的源寄存器

(2) 如何修改数据通路来实现阻塞

硬件阻塞方式(Load-use Data Hazard)

阻塞前的情况：





检测“阻塞”过程中：

1) sub指令在IF/ID寄存器中，**并正被译码**，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

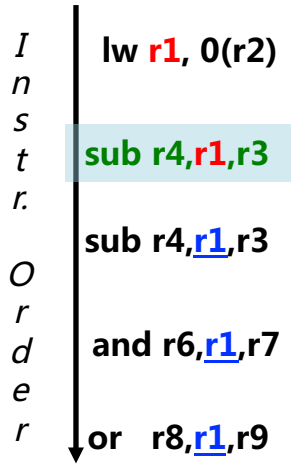
在阻塞点，必须将上述两条指令的执行结果清除，并延迟一个周期执行这两条指令

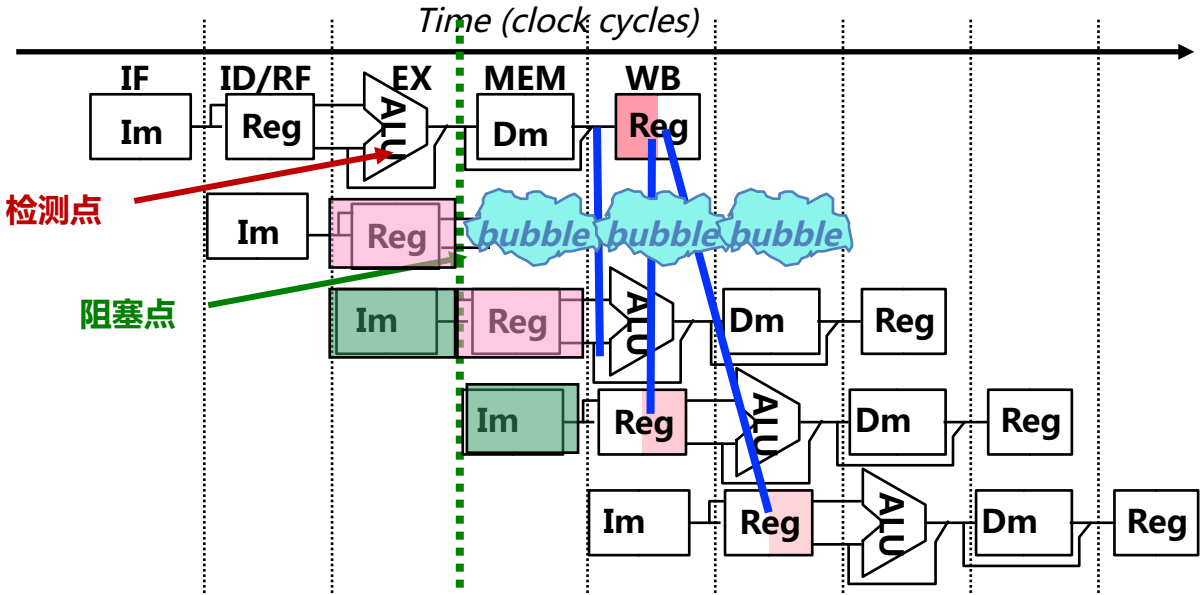
延迟一个周期执行后面指令，相当于把阻塞点前面一个周期的状态再保持一个周期

lw指令还是继续正常执行下去

硬件阻塞方式(Load-use Data Hazard)

阻塞后的情况：





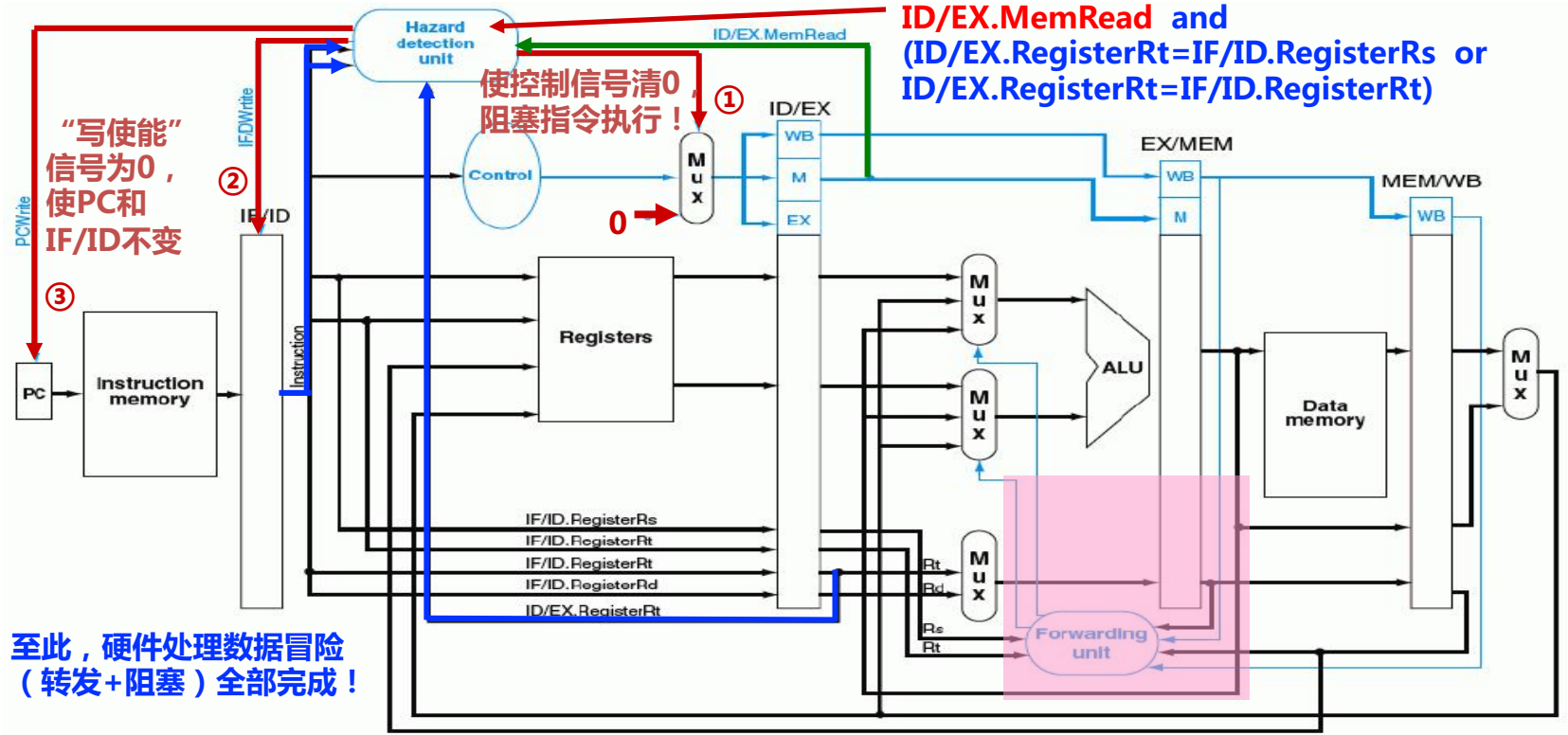
在阻塞点，将上述两条指令的执行结果清除，并延迟一个周期执行这两条指令

① 将ID/EX段寄存器中所有控制信号清0 ，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



数据冒险-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！

Slow code:

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

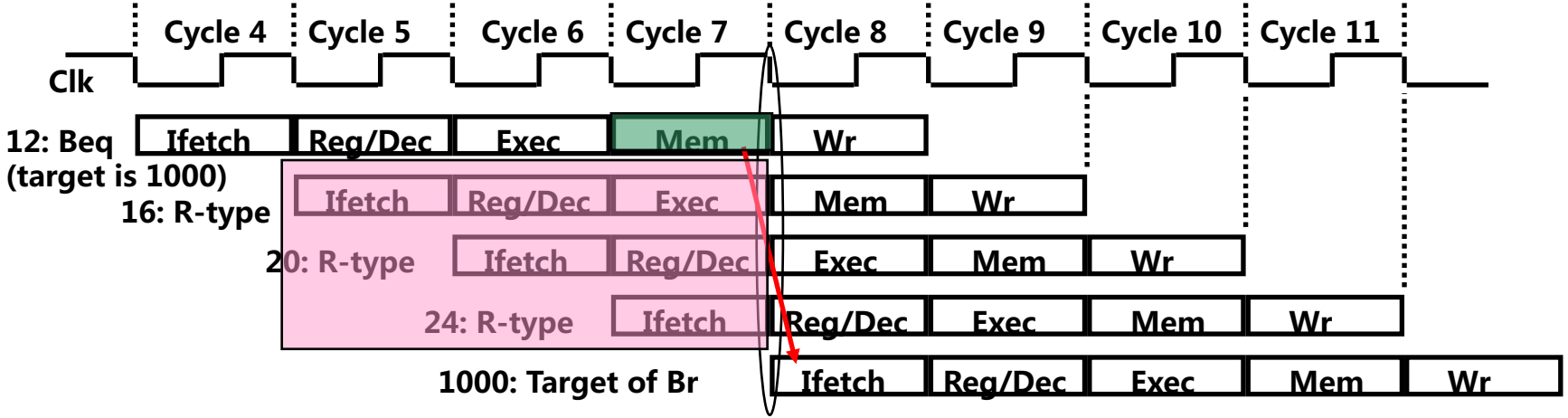
调整后

Fast code:

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



- 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**

控制冒险的解决方法

- **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)
  - **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：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

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**
  - 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
  - 是否可把判断转移的工作提前，而不等到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）

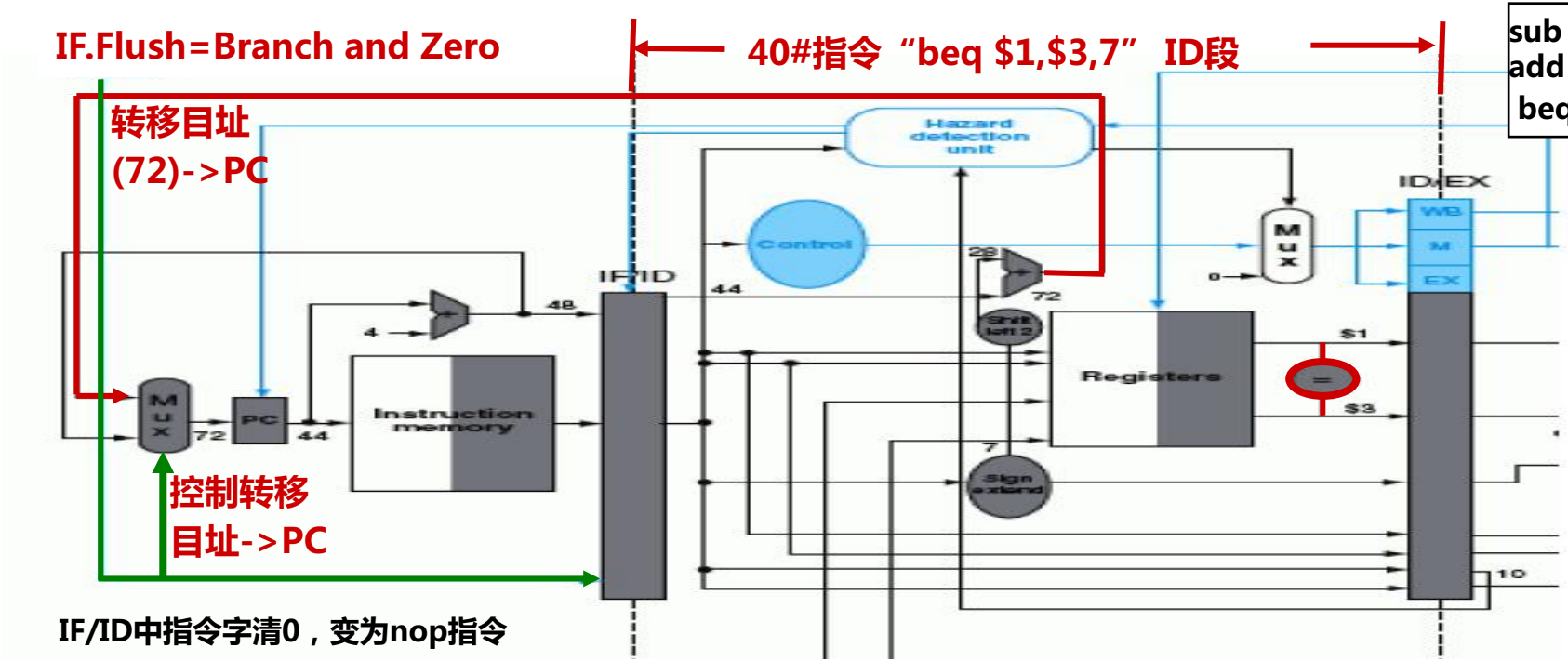
    （许多条件判断都很简单）
- **预测错误的检测和处理（称为“冲刷、冲洗” -- Flush）**
  - 当Branch=1并且Zero=1时，发生转移（taken）
  - **增加控制信号：IF.Flush=Branch and Zero**，取值为1时，说明预测失败
  - 预测失败 (条件满足) 时，完成以下两件事（延迟损失时间片C=1时）：

    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！ 即：这里 C=1

带静态分支预测处理的数据通路



IF/ID中指令字清0，变为nop指令

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!

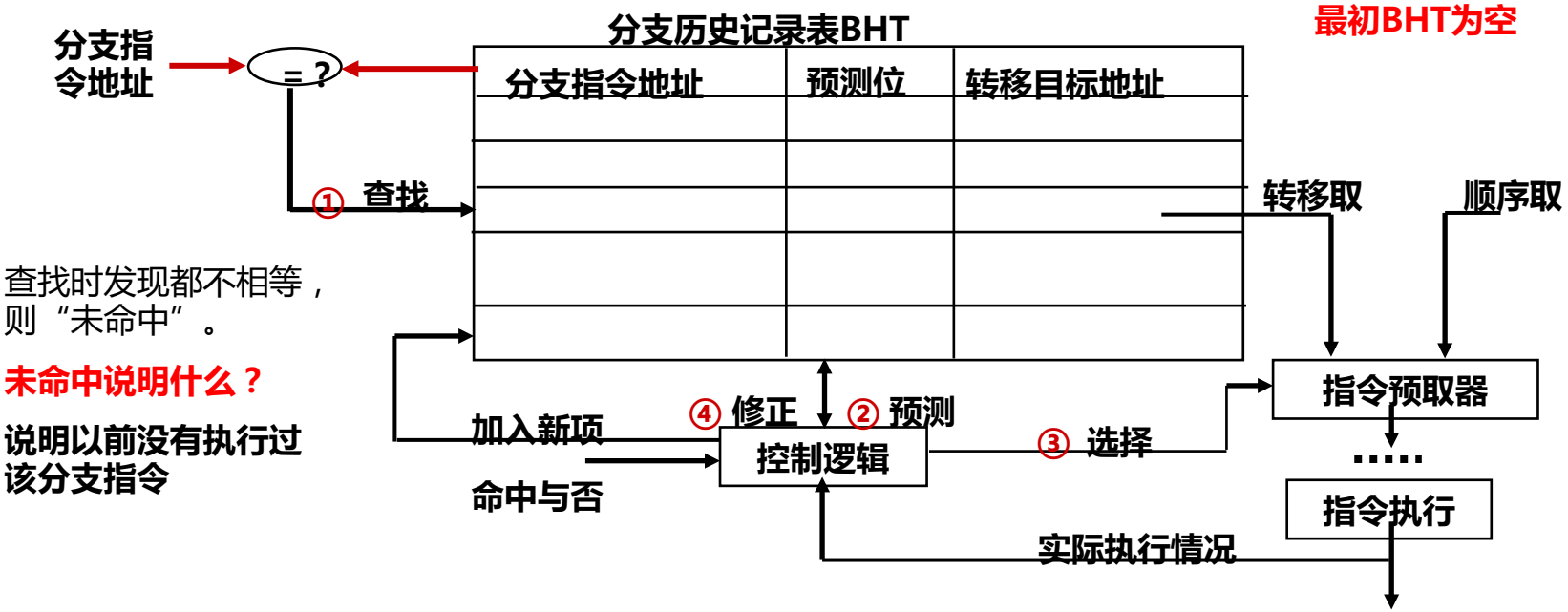
需重新改“转发”条件和转发线路！

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）
  - 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



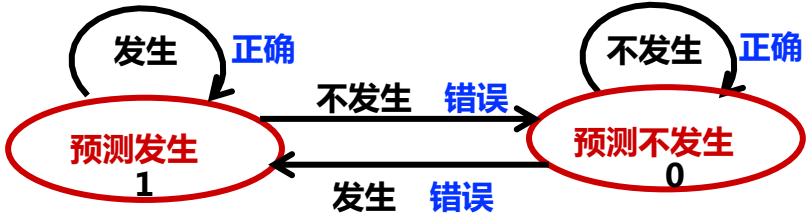
- **命中时**：根据预测位，选择“转移取”还是“顺序取”
- **未命中时**：加入新项，并填入指令地址和转移目标地址、初始化预测位

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



- 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。

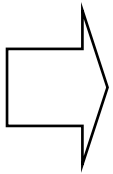
```
Loop:    g = g +A[i];
         i = i+ j;
         if (i != h) go to Loop:
```

Assuming variables g, h, i, j ~ $1, $2, $3, $4 and base address of array is in $5

```
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
          … …
```

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;
}
```



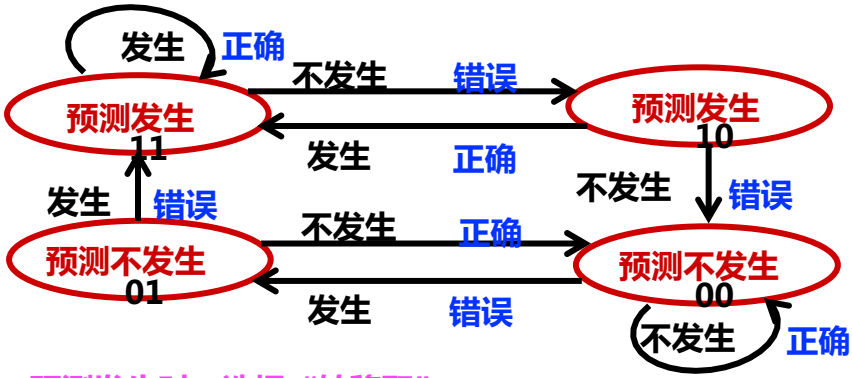
```
… …
Loop-i:  beq $t1,$a0, exit-i    # 若( i=N)则跳出外循环
         add $t2, $zero, $zero  #j=0
Loop-j:  beq $t2, $a0, exit-j   # 若(j=N)则跳出内循环
         addi $t2, $t2, 1                  # j=j+1
         addi $t0, $t0, 1                  #sum=sum+1
         j  Loop-j
exit-j:  addi $t1, $t1, 1       # i=i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

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×(N–1)次预测错误（第一次循环有1次预测错，后面(N-1)次循环每次有2次预测错）。 **N越大预测准确率越高！**

两位预测状态图



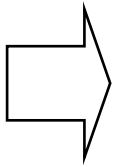
预测发生时，选择“转移取”
预测不发生时，选择“顺序取”

```
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
          … …
```

- **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（再次进入循环时为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%

若预测位初始为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！**

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，无需在硬件线路中阻塞branch指令后面指令的执行

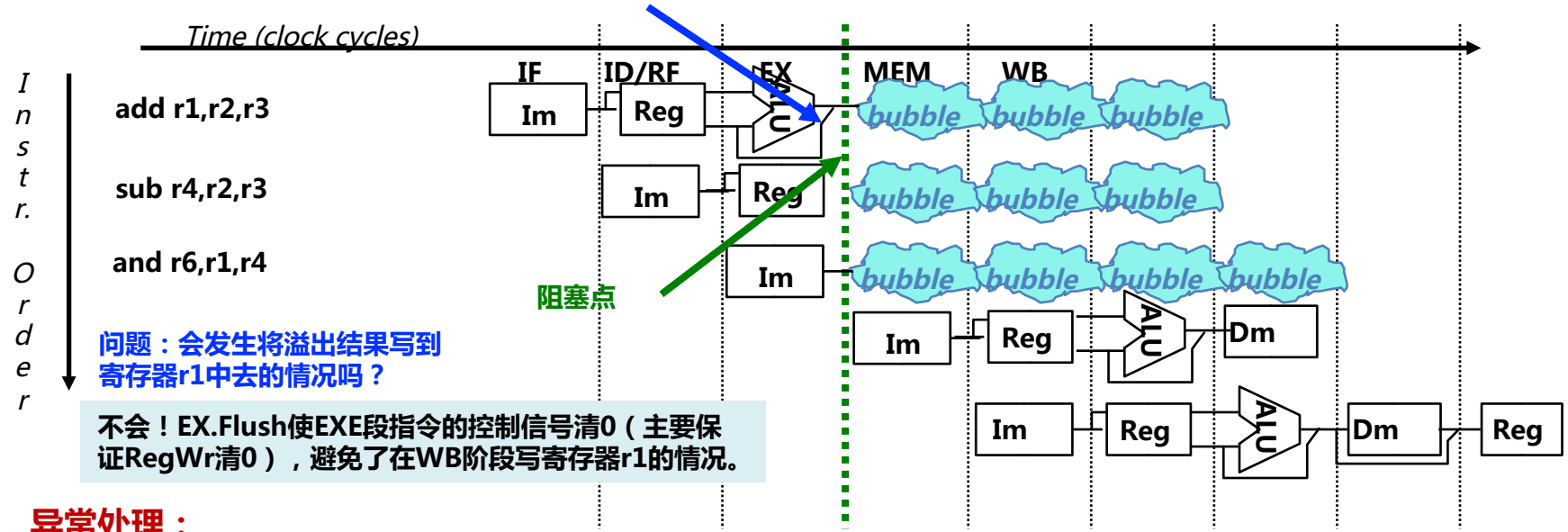
另一种控制冒险：异常和中断

- **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

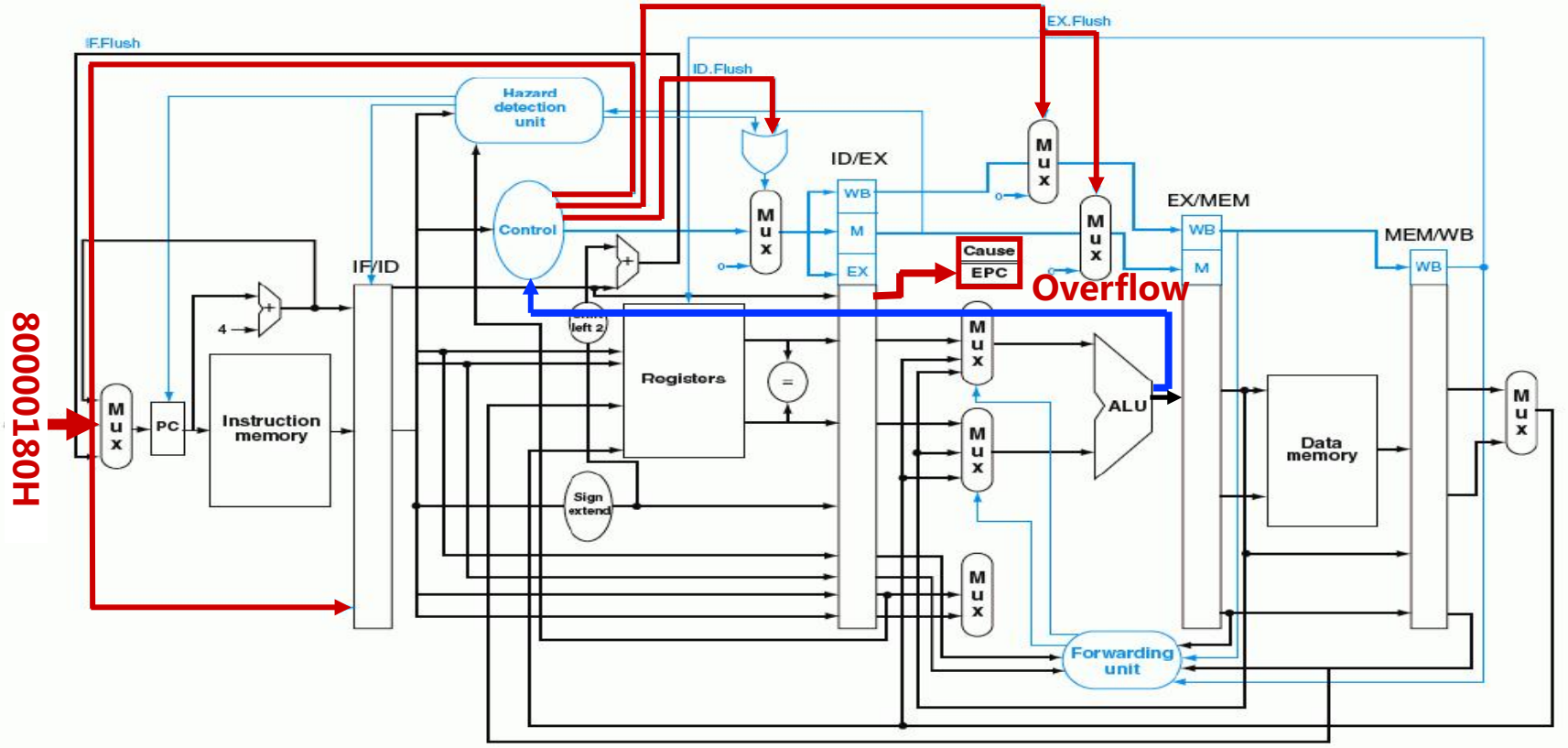
- 异常（溢出）在第一条指令的EXE阶段被检出



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，其原因存到专门寄存器中并流到最后阶段处理，使前面指令的异常优先级高于后面指令
  - 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

课程习题（作业）——截止日期：12月8日晚23:59

- **课本203-205页**：第4、5、6、7、9、10、11、12题
- 提交方式：https://selearning.nju.edu.cn/（教学支持系统）

第6章-指令流水线-课后习题

课本203-205页：第4、5、6、7、9、10、11、12题

- 命名：学号+姓名+第*章。
- 若提交遇到问题请及时发邮件或在下一次上课时反馈。

课程习题（作业）——截止日期：12月8日晚23:59

4. 假定某计算机工程师想设计一个新的CPU，一个典型程序的核心模块有一百万条指令，每条指令执行时间为100ps。请问：

（1）在非流水线处理器上执行该程序需要花多长时间？

（2）若新CPU采用20级流水线，执行上述同样的程序，理想情况下，它比非流水线处理器快多少？

（3）实际流水线并不是理想的，流水段之间的数据传送会有额外开销。这些开销是否会影响指令执行时间和指令吞吐率？

5. 假定最复杂的一条指令所用的组合逻辑分成6部分，依次为A～F，其延迟分别为80ps、30ps、60ps、50ps、70ps、10ps。在这些组合逻辑块之间插入必要的流水段寄存器就可实现相应的指令流水线，寄存器延迟为20ps。理想情况下，以下各种方式所得到的时钟周期、指令吞吐率和指令执行时间各是多少？应该在哪里插入流水段寄存器？

（1）插入1个流水段寄存器，得到一个两级流水线。

（2）插入2个流水段寄存器，得到一个三级流水线。

（3）插入3个流水段寄存器，得到一个四级流水线。

（4）吞吐量最大的流水线。

课程习题（作业）——截止日期：12月8日晚23:59

6. 以下指令序列中，哪些指令对之间发生数据相关？假定采用“取指、译码/取数、执行、访存、写回”5段流水线方式，如果不用“转发”技术，需要在发生数据相关的指令前加入几条 nop 指令才能使这段程序避免数据冒险？如果采用“转发”是否可以完全解决数据冒险？不行的话，需要在发生数据相关的指令前加入几条 nop 指令才能使这段 MIPS 程序不发生数据冒险？

```
addu    $s3, $s1, $s0
addu    $t2, $s3, $s3
lw      $t1, 0($t2)
add     $t3, $t1, $t2
```

7. 假定以下 MIPS 指令序列在图 6.18 所示的流水线数据通路中执行：

```
addu    $s3, $s1, $s0
subu    $t2, $s3, $s3
lw      $t1, 0($t2)
add     $t3, $t1, $t2
add     $t1, $s4, $s5
```

请问：

(1) 上述指令序列中，哪些指令的哪个寄存器需要转发？转发到何处？

(2) 上述指令序列中，是否存在 load-use 数据冒险？

(3) 第 5 周期结束时，各指令执行状态是什么？哪些寄存器的数据正被读出？哪些寄存器将被写入？

课程习题（作业）——截止日期：12月8日晚23:59

9. 在一个带转发的5段流水线中执行以下MIPS程序段，怎样调整指令序列使其性能达到最好？

```
lw    $2, 100($6)
add   $2, $2, $3
lw    $3, 200($7)
add   $6, $4, $7
sub   $3, $4, $6
lw    $2, 300($8)
beq   $2, $8, Loop
```

10. 在一个采用“取指、译码/取数、执行、访存、写回”的5段流水线中，若检测结果是否为“0”和将转移目标地址(Btarg和Jtarg)送PC的操作在执行阶段进行，则分支延迟损失时间片(即分支延迟槽)为多少？在带转发的5段流水线中，对于以下MIPS指令序列，哪些指令执行时会发生流水线阻塞？各需要阻塞几个时钟周期？

```
Loop:  add    $t1, $s3, $s3
       add    $t1, $t1, $t1
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       add    $s3, $s3, $s4
       j      Loop
Exit:
```

课程习题（作业）——截止日期：12月8日晚23:59

11. 假设数据通路中各主要功能部件的操作时间是：存储单元为 200ps；ALU 和加法器为 100ps；寄存器堆读口或写口为 50ps。程序中指令的组成比例为：取数为 25%、存数为 10%、ALU 为 52%、分支为 11%、跳转为 2%。假设控制单元和传输线路等延迟都忽略不计，则以下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。时钟周期取存储单元操作时间的一半，每类指令时钟数是：取数为 7、存数为 6、ALU 为 5、分支为 4、跳转为 4。

（3）流水线方式。时钟周期取存储操作时间的一半，采用“取指 1、取指 2、取数/译码、执行、存取 1、存取 2、写回”7 段流水线；没有结构冒险；数据冒险采用“转发”技术处理；load 指令与后续各指令之间存在依赖关系的概率分别 1/2，1/4，1/8，…；分支延迟损失时间片为 2，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。

课程习题（作业）——截止日期：12月8日晚23:59

12. 有一段程序的核心模块中有 5 条分支指令，该模块将会被执行成千上万次，在其中一次执行过程中，5 条分支指令的实际执行情况如下（T：taken；N：not taken）。

分支指令 1：T-T-T。

分支指令 2：N-N-N-N。

分支指令 3：T-N-T-N-T-N。

分支指令 4：T-T-T-N-T。

分支指令 5：T-T-N-T-T-N-T。

假定各个分支指令在每次模块执行过程中实际执行情况都一样，并且动态预测时每个分支指令都有自己的预测表项，每次执行该模块时的初始预测位都相同。请分析并给出以下几种预测方案的预测准确率。

（1）静态预测，总是预测转移（taken）。

（2）静态预测，总是预测不转移（not taken）。

（3）一位动态预测，初始预测转移（taken）。

（4）二位动态预测，初始预测弱转移（taken）。

提问

Q & A

殷亚凤
智能软件与工程学院
苏州校区南雍楼东区225
yafeng@nju.edu.cn，https://yafengnju.github.io/

南京大學 NANJING UNIVERSITY