

Chapter 4

处理器

——流水线**CPU**

Outline

- **流水线CPU概述**
- **流水线CPU datapath与控制**
- 流水线CPU 数据冒险的检测与处理
- 流水线CPU 控制冒险的检测与处理
- 中断与异常
- 多发射

RISC-V 流水线CPU的数据通路

IF: Instruction fetch

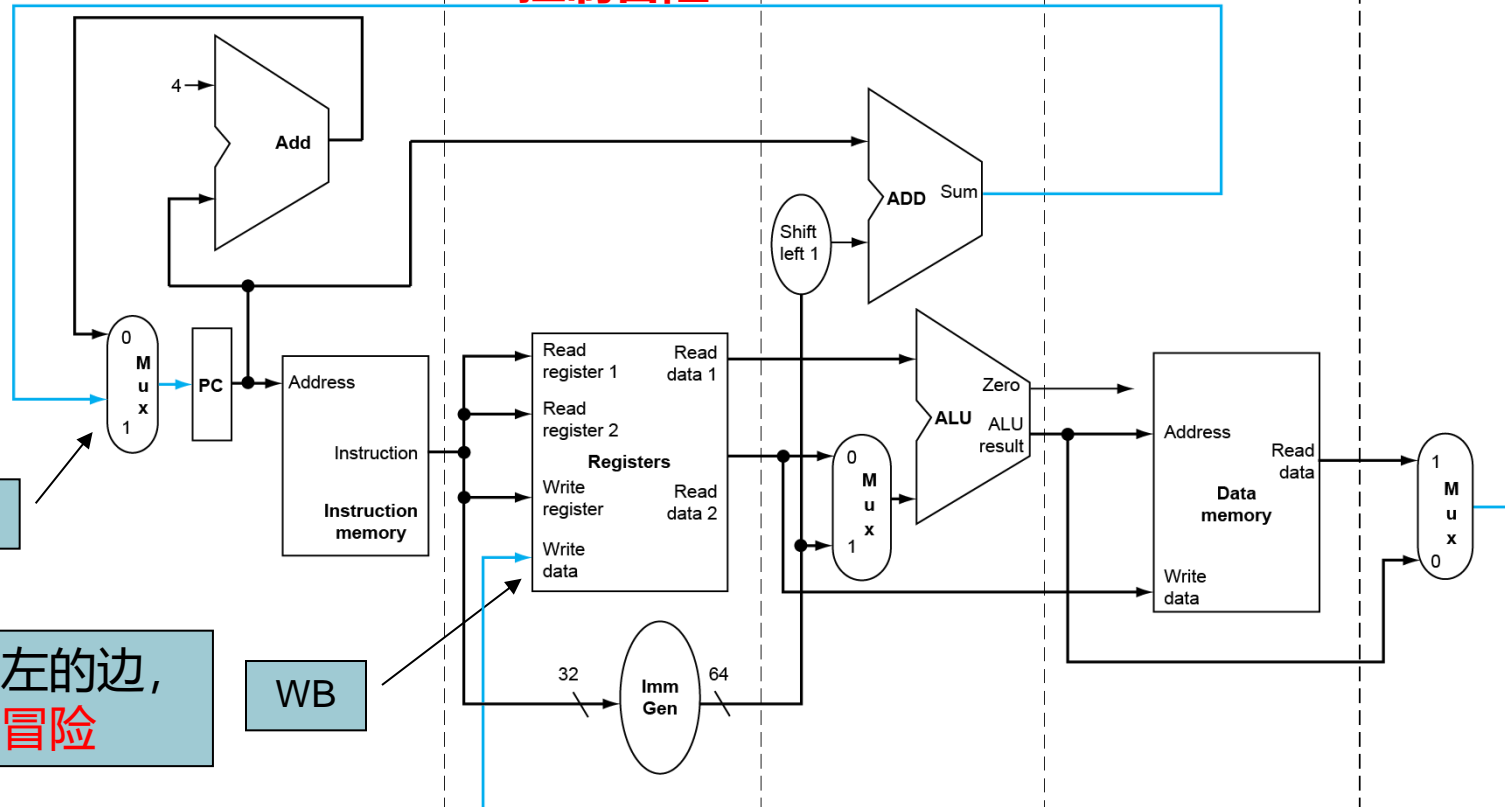
ID: Instruction decode/
register file read

EX: Execute/
address calculation

MEM: Memory access

WB: Write back

控制冒险

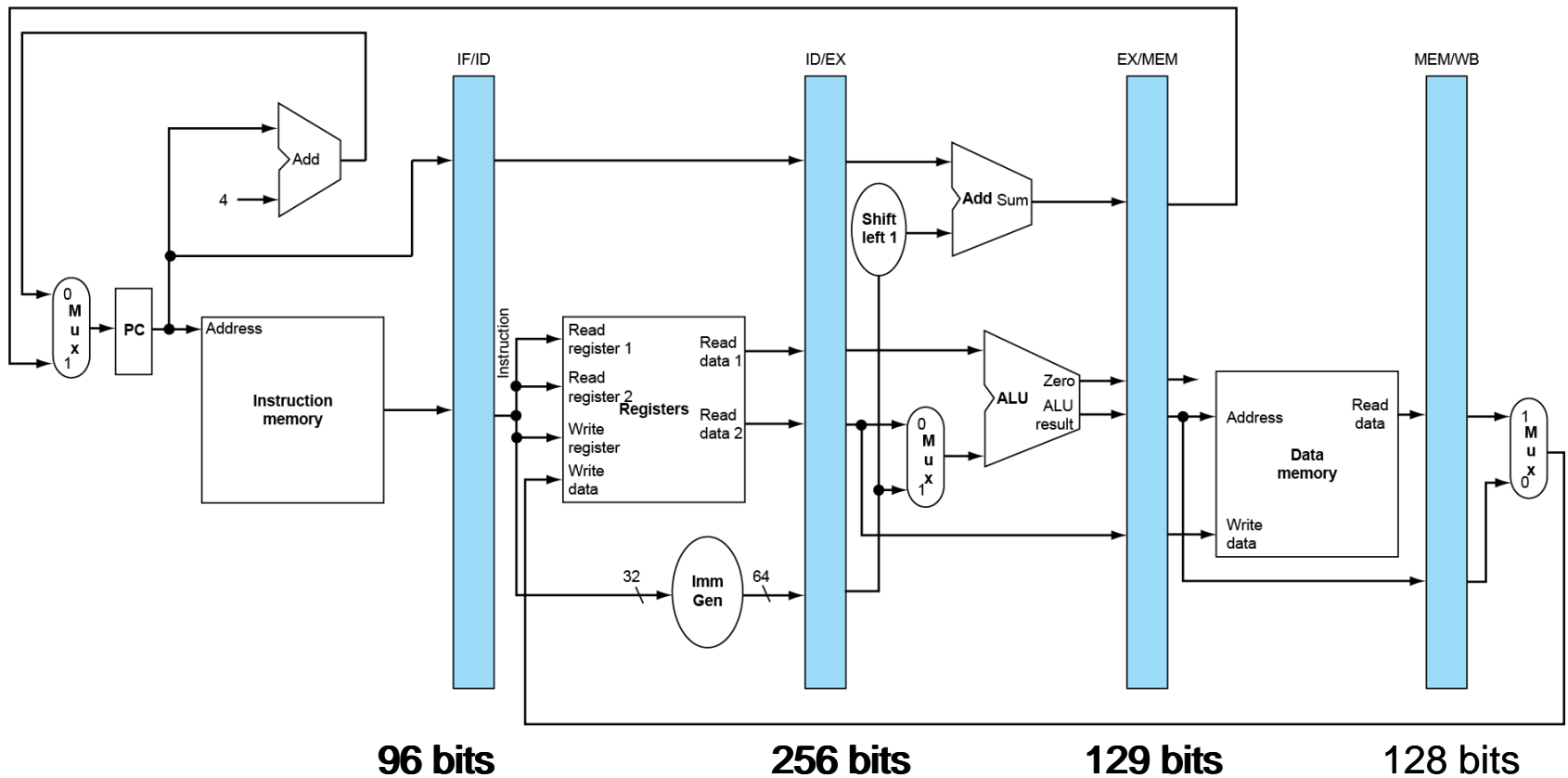


数据冒险

从右往左的边，
意味着冒险

流水线寄存器

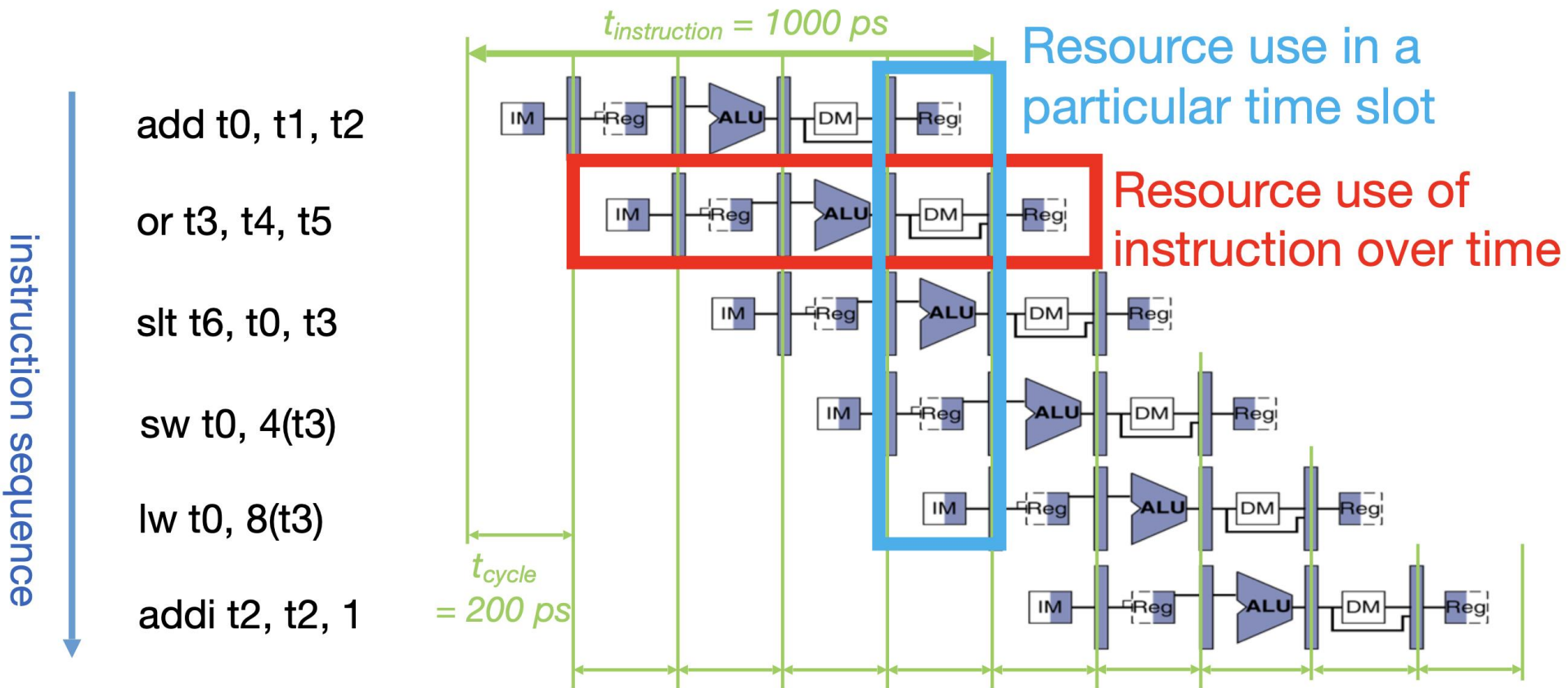
- 在阶段之间需要寄存器，
 - 存储每周期更新的状态信息



Pipeline Operation

- 指令逐级经过流水线数据通路
 - 每周期经过一级
- “单周期” 流水线图
 - 展示单个周期内流水线通路的使用
 - 突出展示单个周期内，资源的使用
 - c.f. “多周期” 流水线图
 - 展示一条指令在不同周期使用的资源
- 接下来了解load&store指令的“单周期” 流水线图

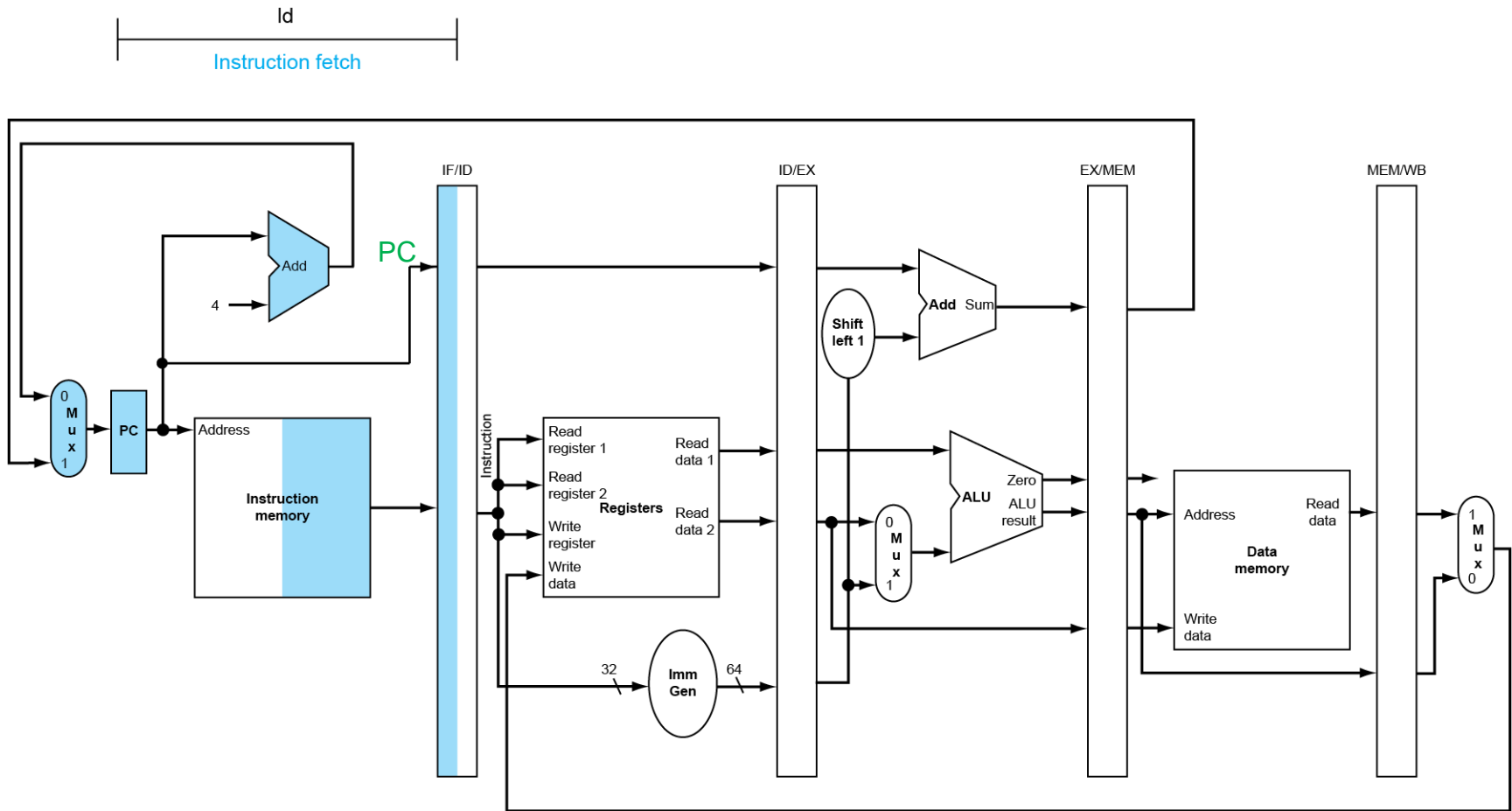
RISC-V 流水线



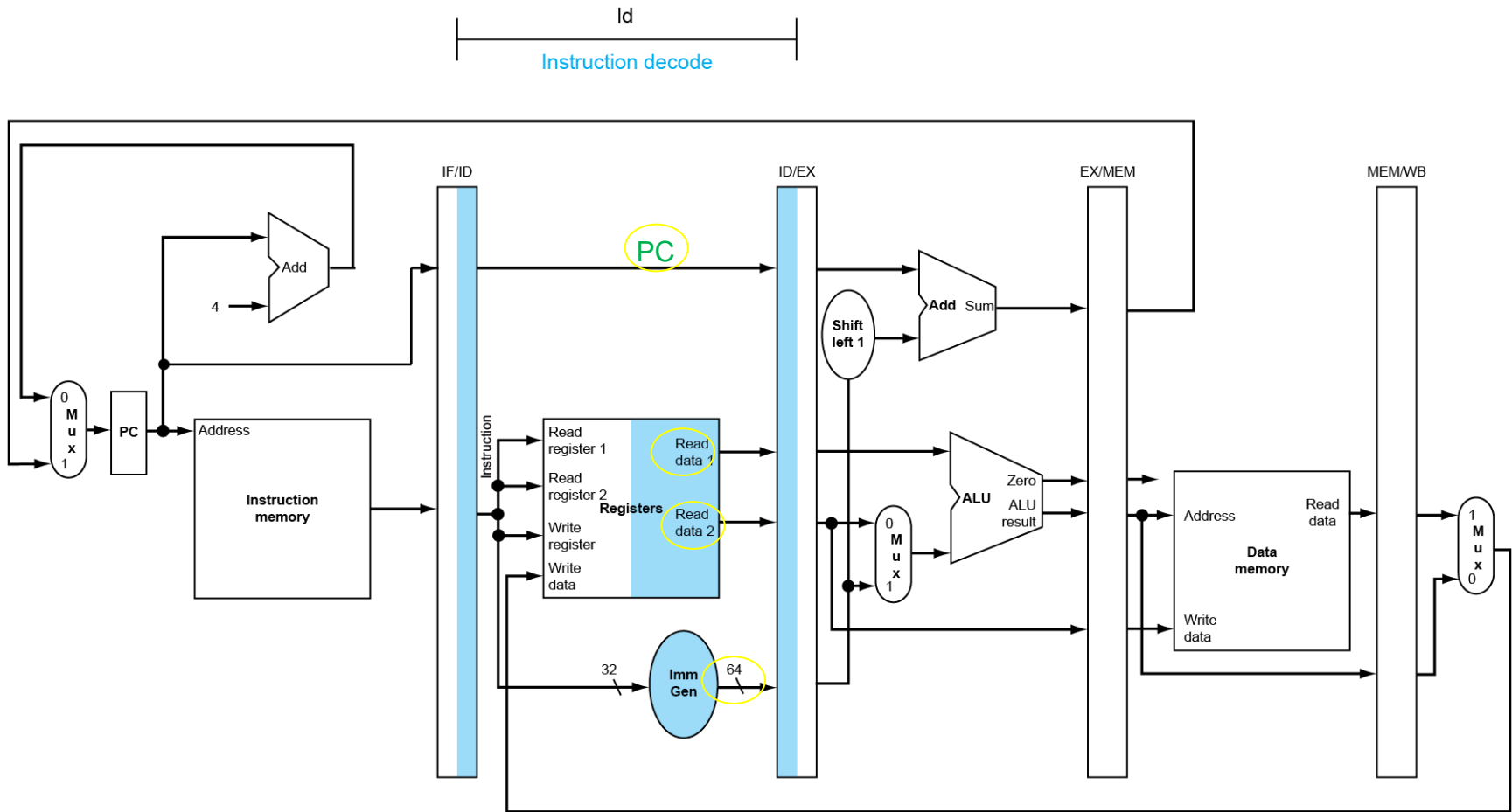
蓝色：“单周期”流水线图

红色：“多周期”流水线图

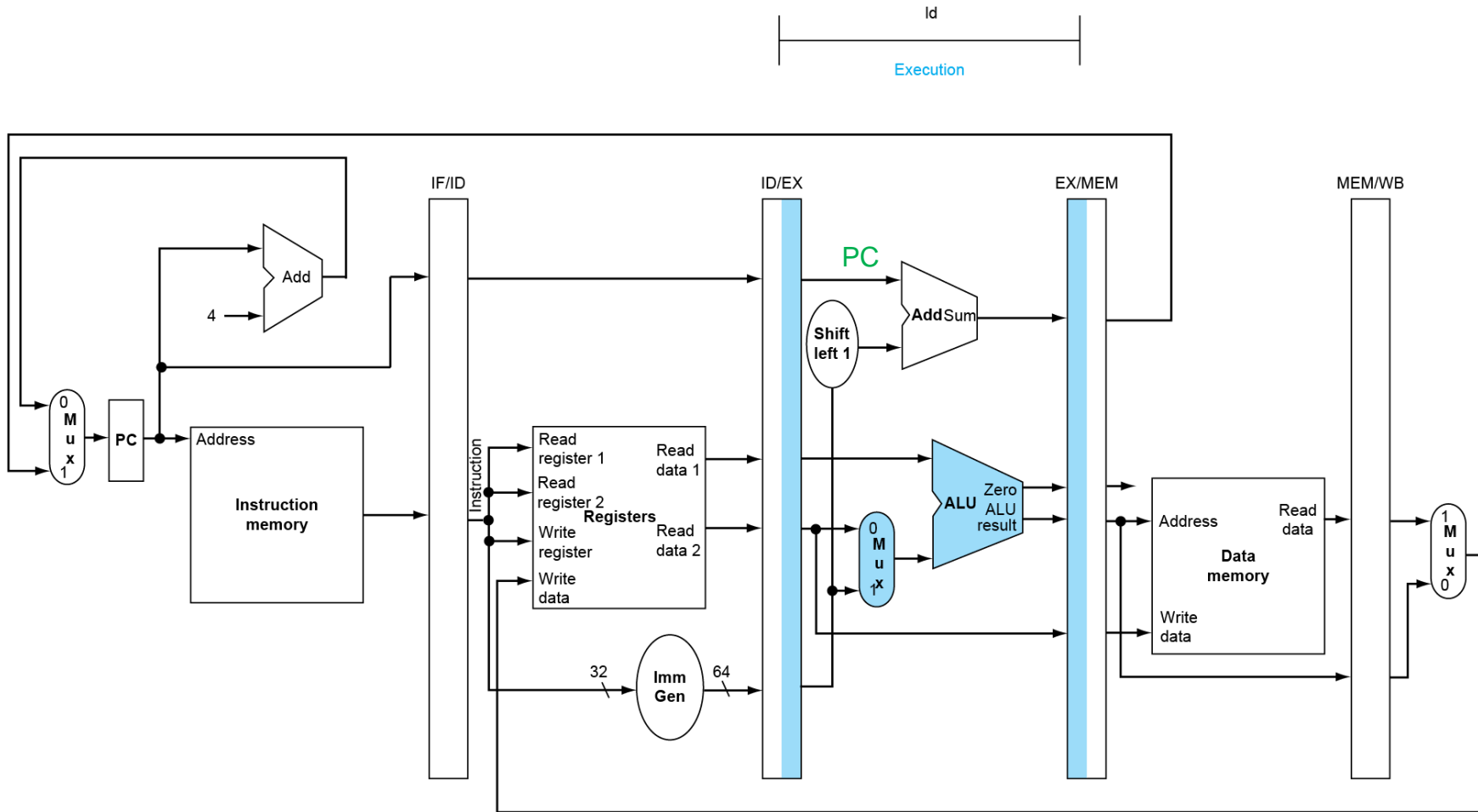
IF for Load, Store, ...



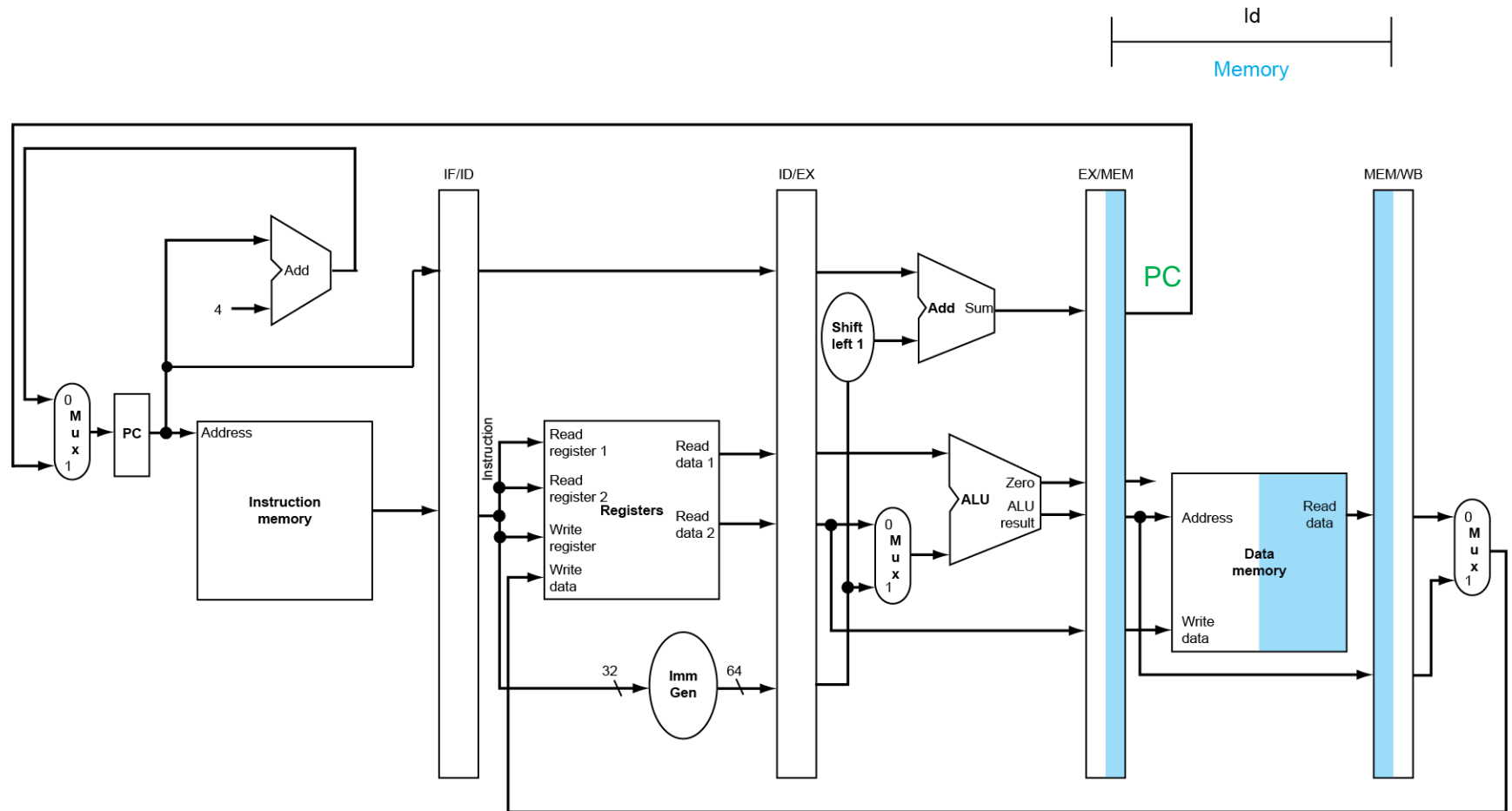
ID for Load, Store, ...



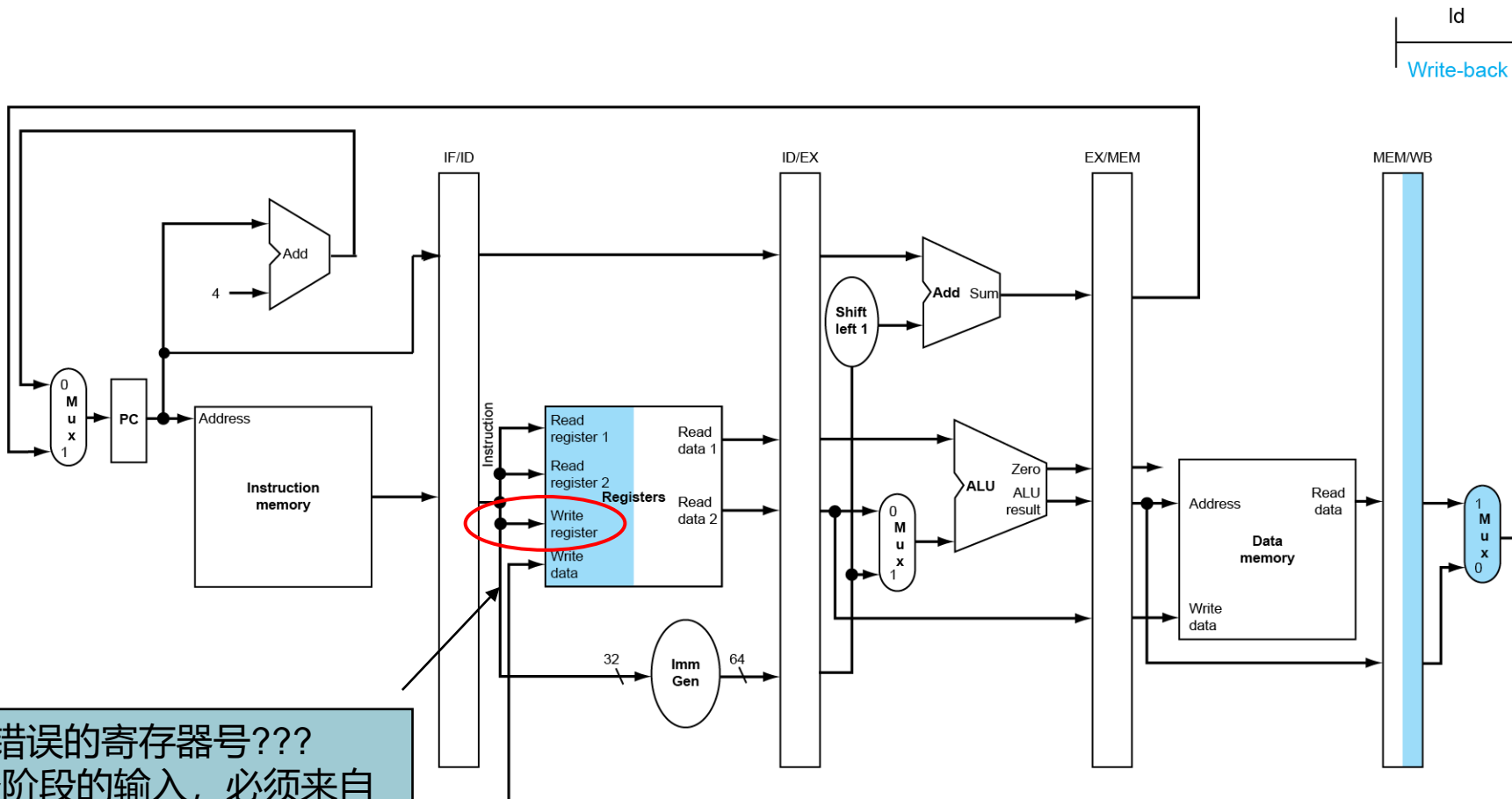
EX for Load



MEM for Load

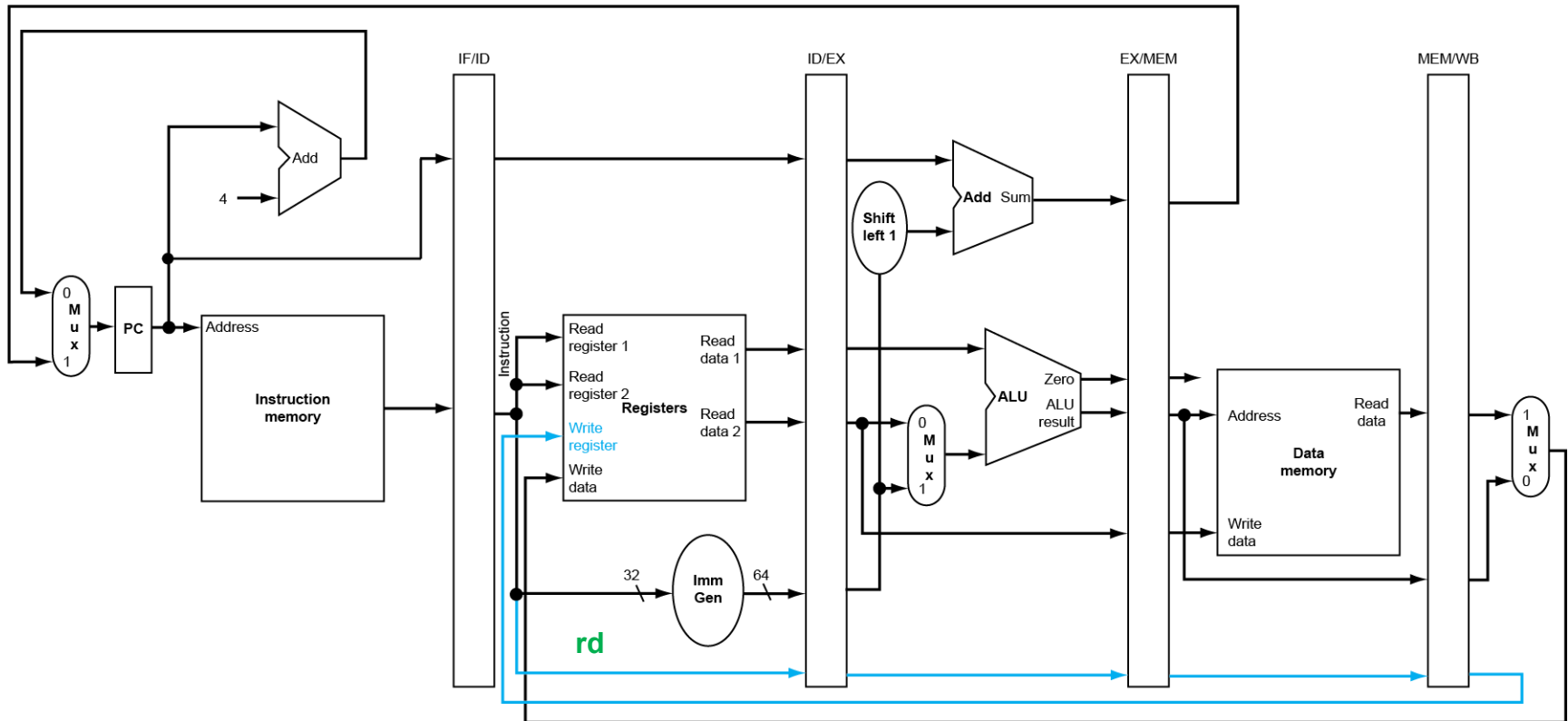


WB for Load

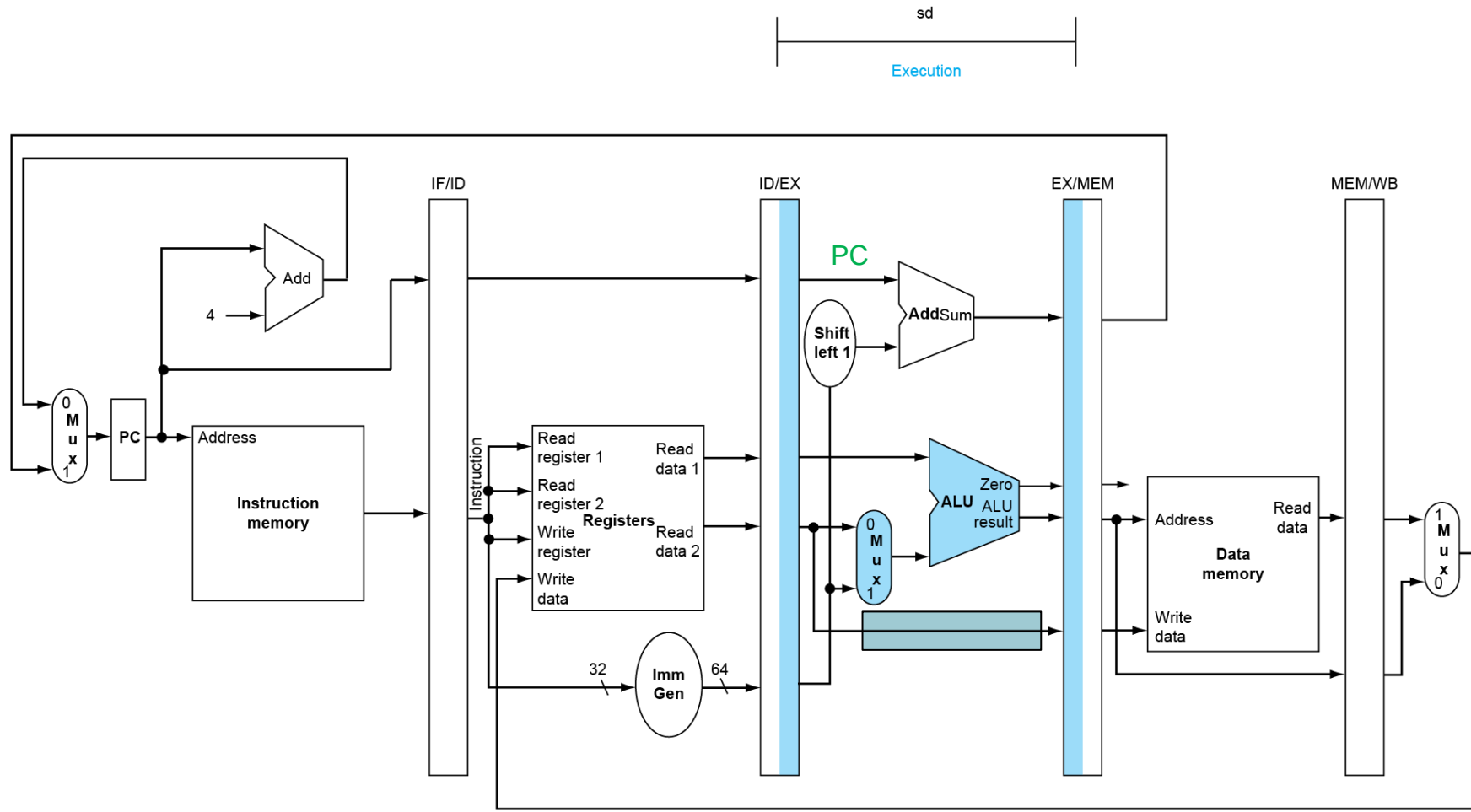


错误的寄存器号???
每个阶段的输入，必须来自
前面的流水线寄存器

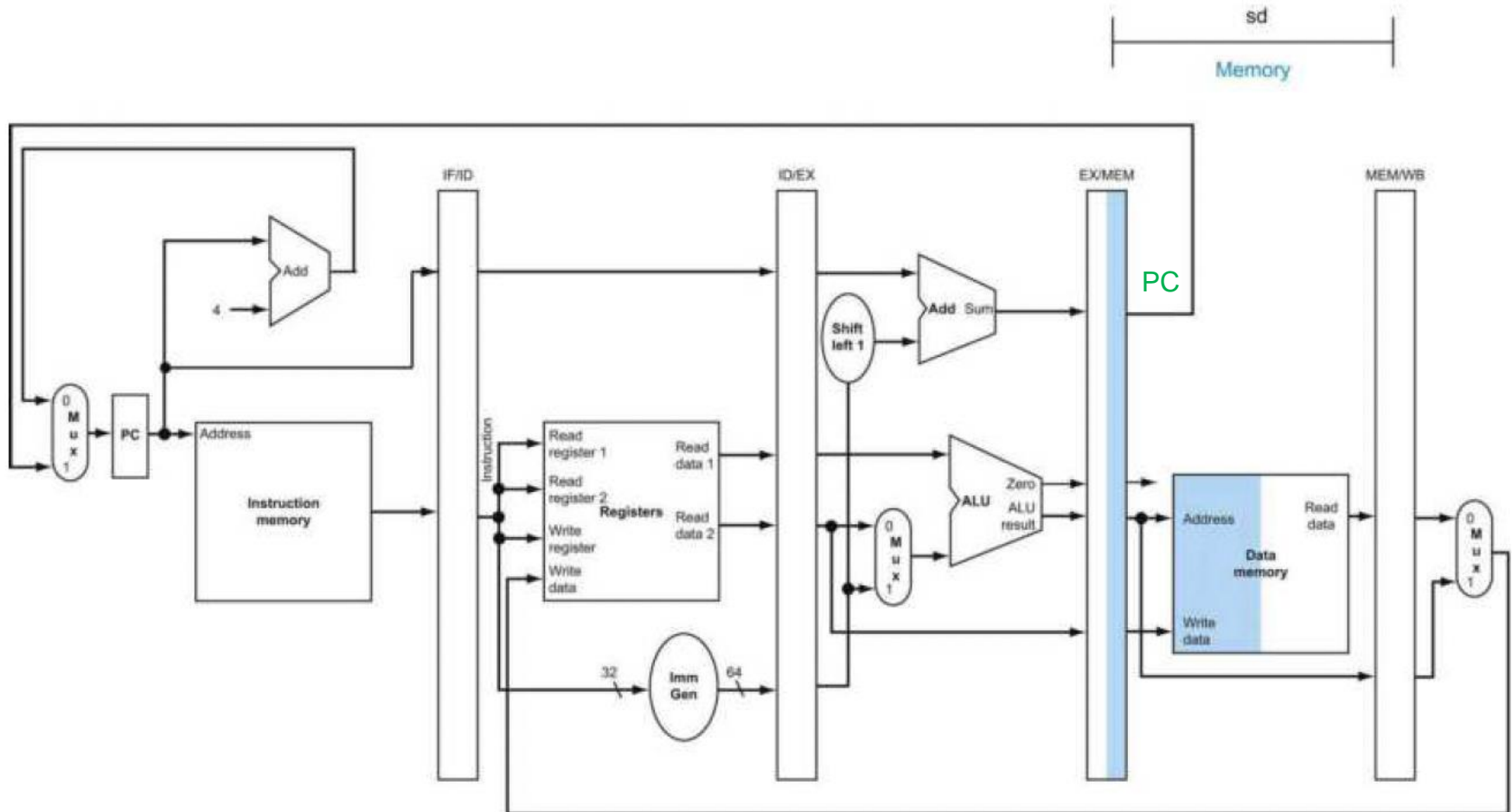
为Load修复流水线数据通路



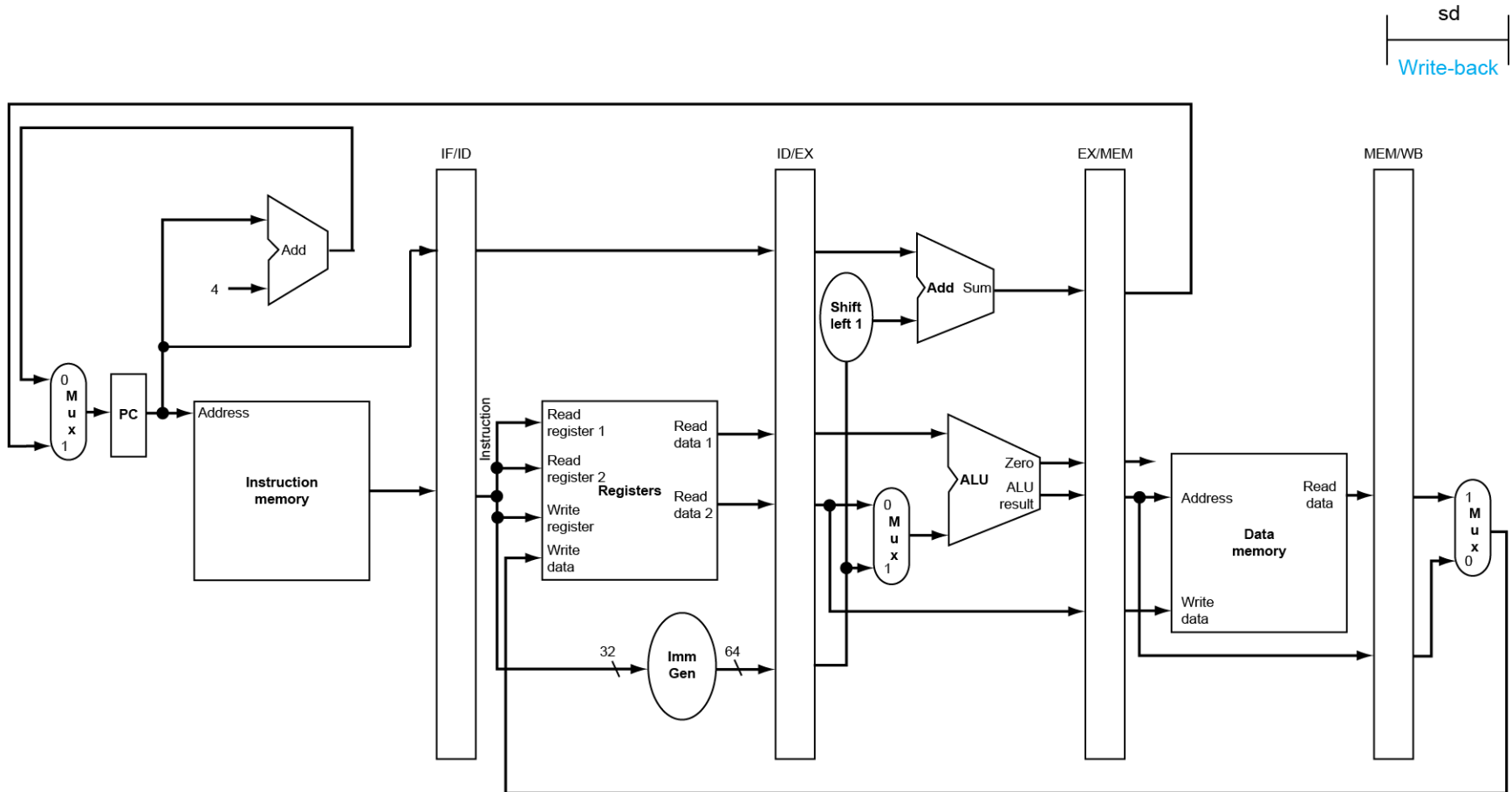
EX for Store



MEM for Store

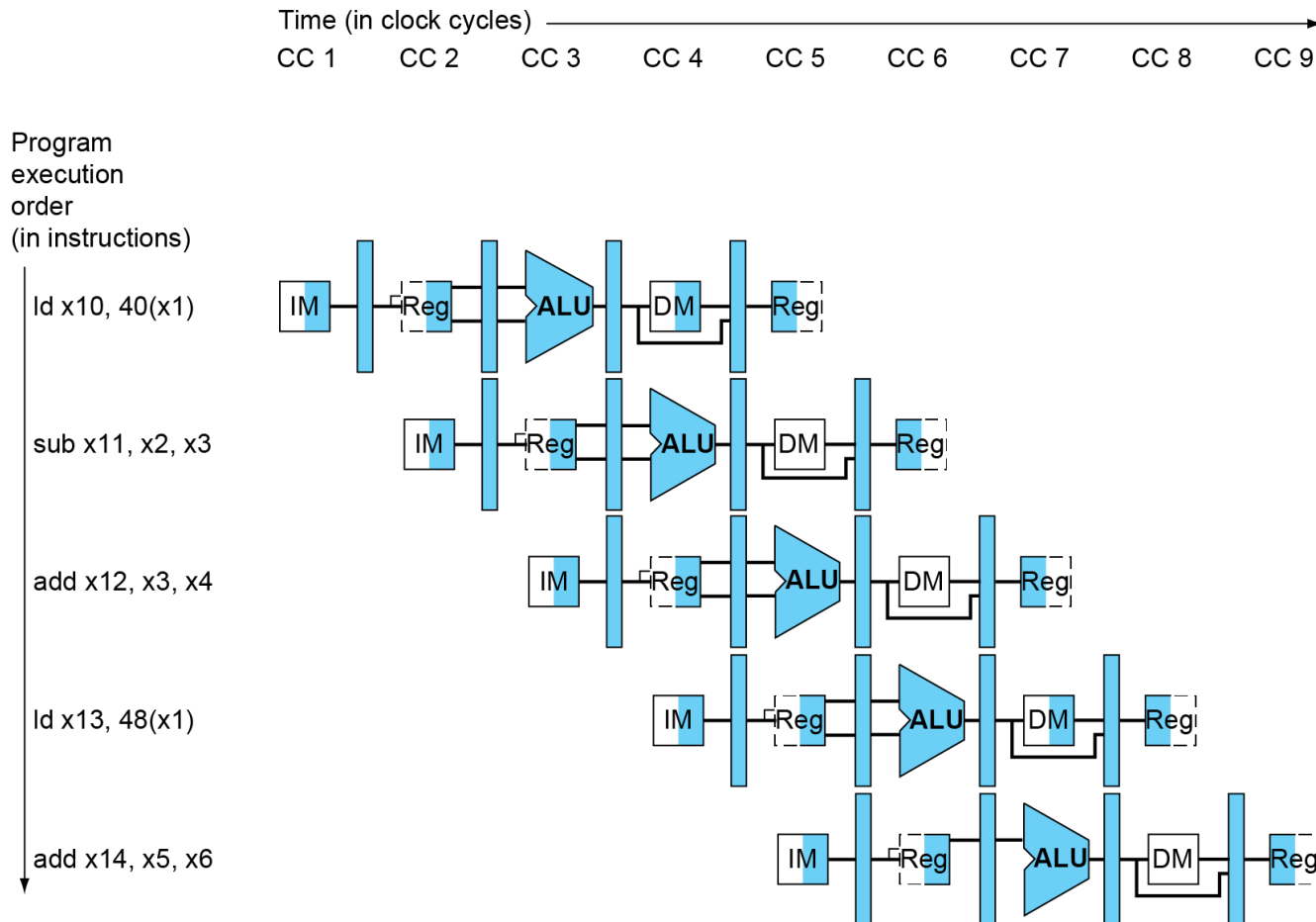


WB for Store



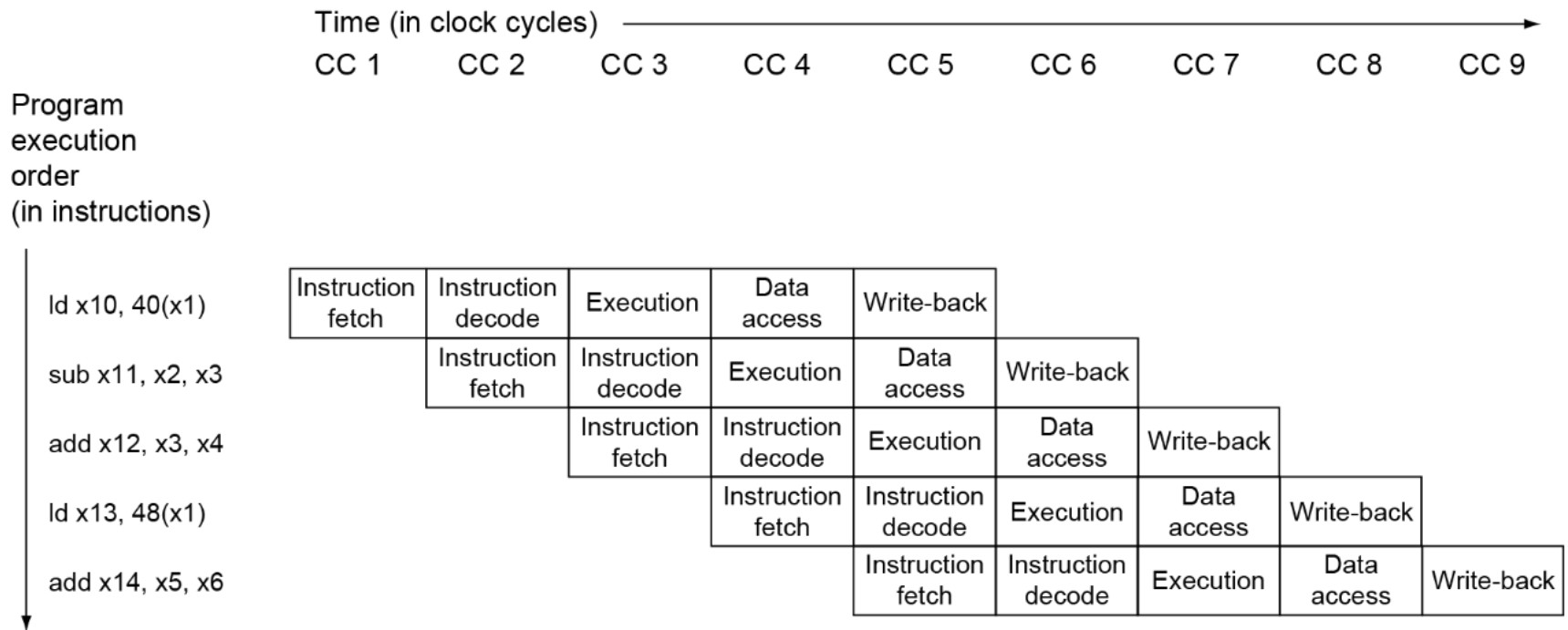
多周期流水线图

■ 展示资源使用的一种方法



多周期流水线图

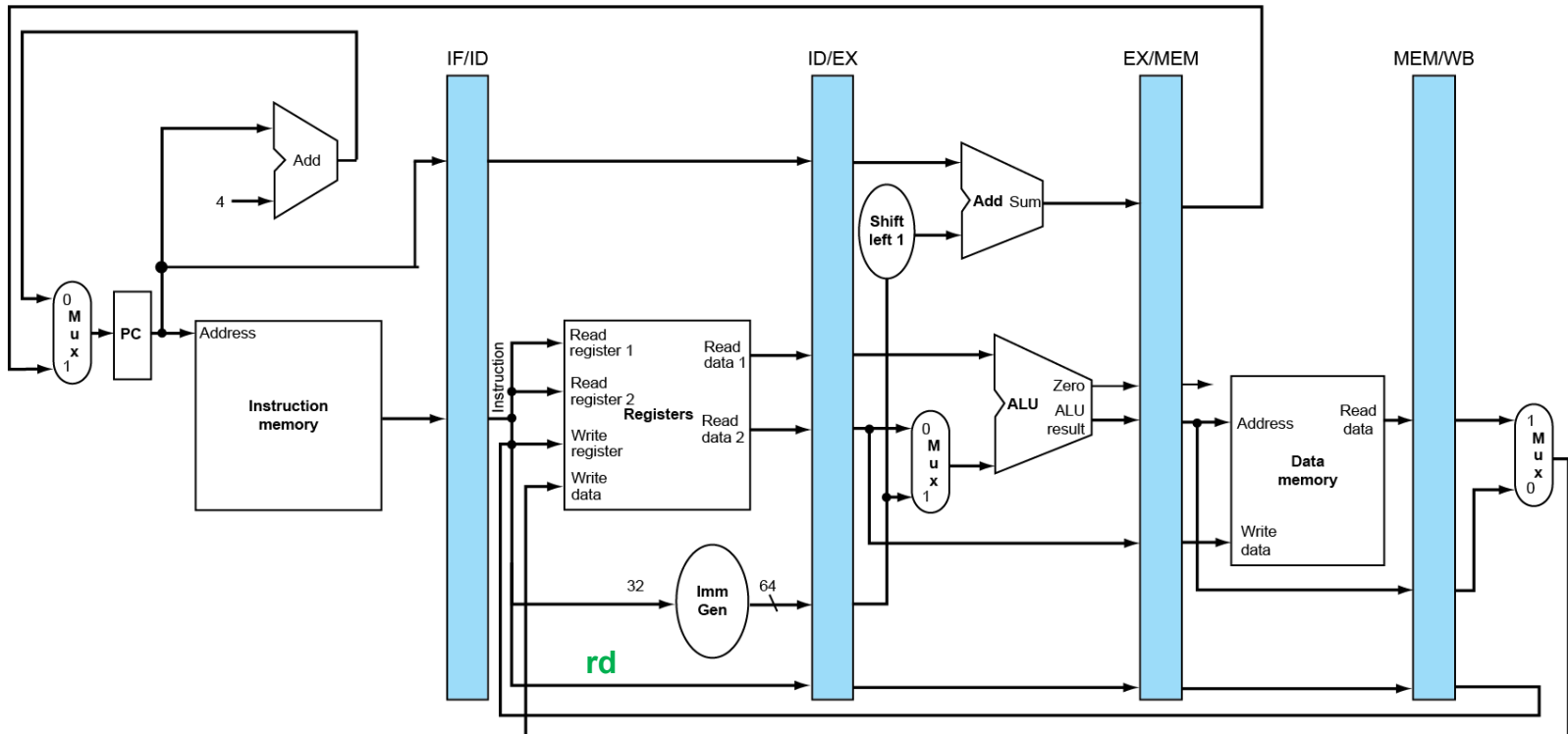
■ 展示资源使用的另一种方法



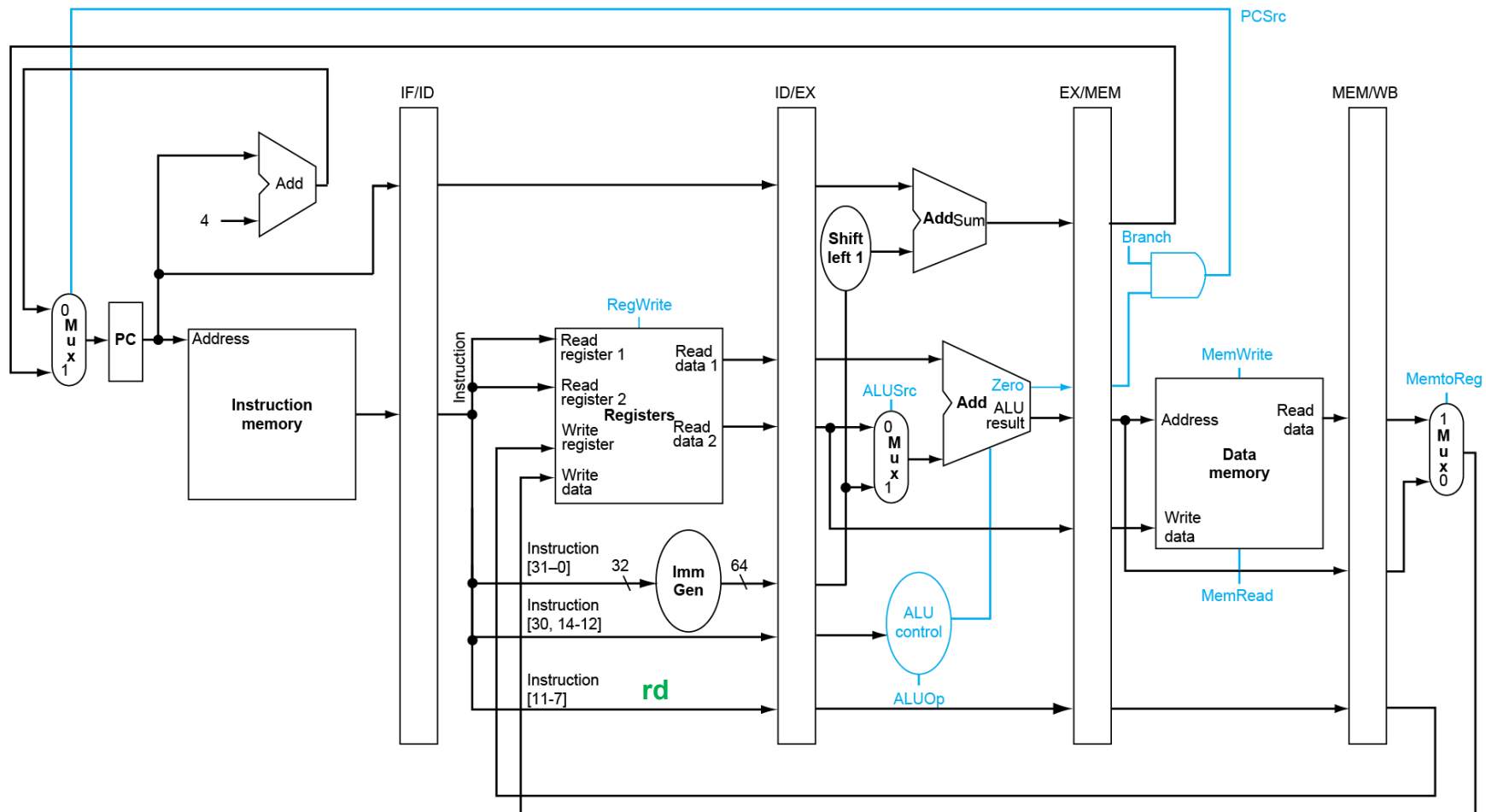
单周期流水线图

■ 给定周期的流水线状态 (clock 5)

add x14, x5, x6	ld x13, 48(x1)	add x12, x3, x4	sub x11, x2, x3	ld x10, 40(x1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back

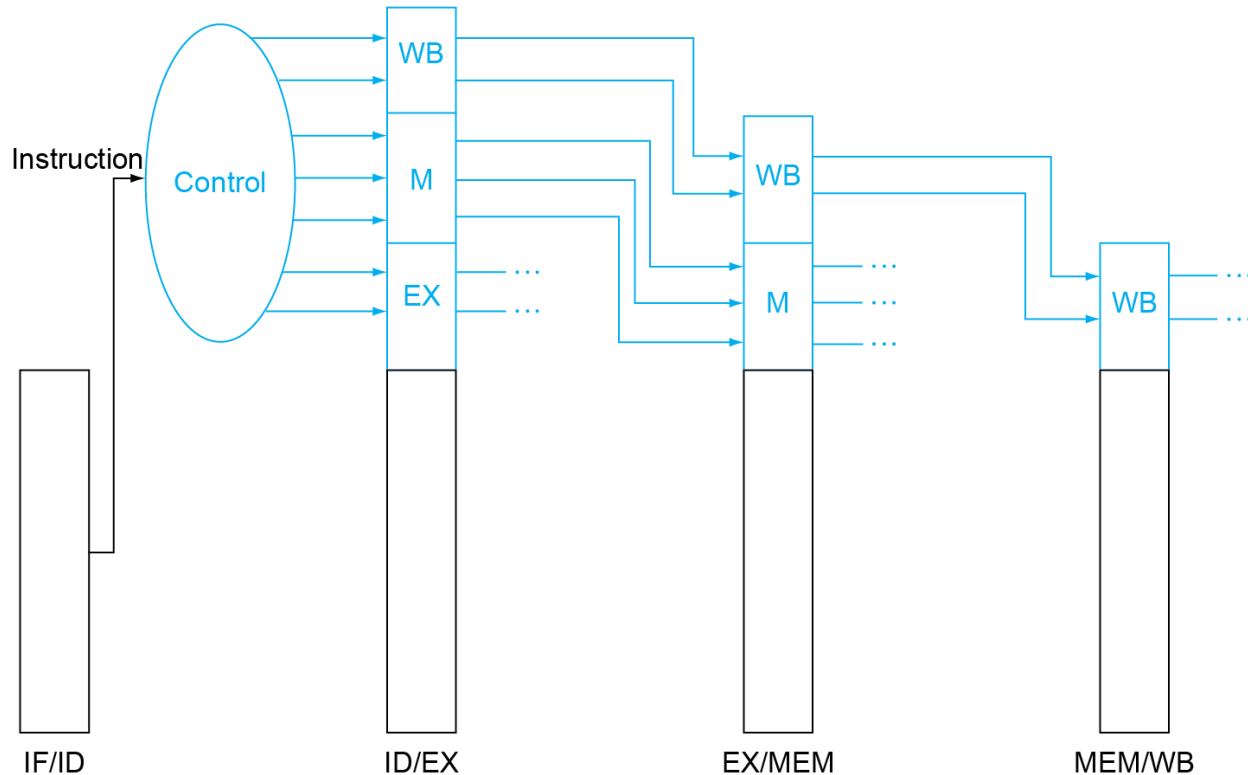


流水线的控制 (简化版)



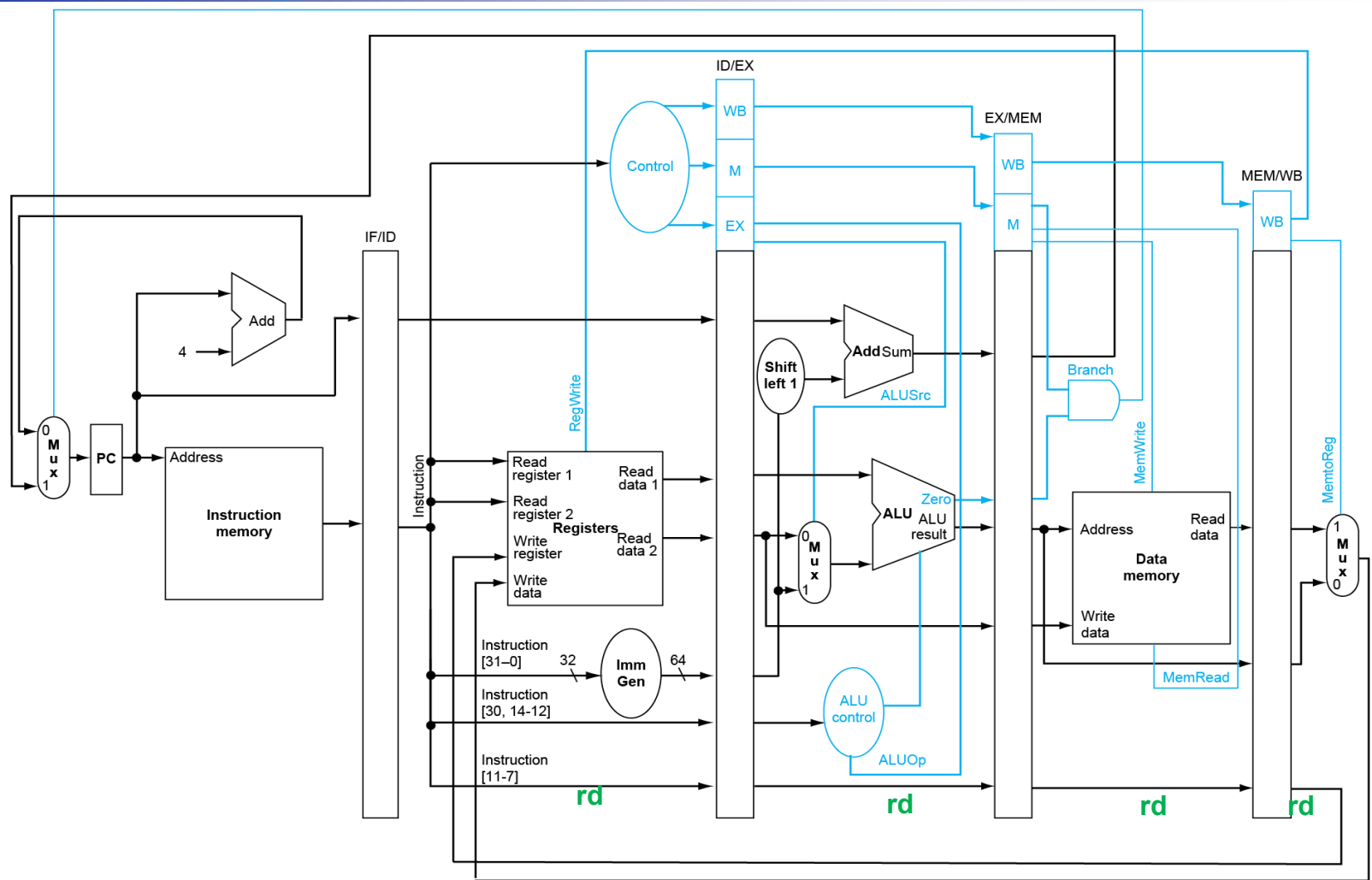
流水线的控制

- 控制信号来自指令
 - 跟单周期CPU一样
 - 不同的是，需要逐级往后传递



流水线的控制

注：此图是在MEM阶段更新PC



$$=64+32$$

$$=8+64+64+64+64+4+5$$

$$=2+3+64+1+64+64+5$$

$$=2+64+64+5$$


Outline

- **流水线CPU概述**
- **流水线CPU datapath与control**
- **流水线CPU 数据冒险的检测与处理**
- **流水线CPU 控制冒险的检测与处理**
- **中断与异常**
- **多发射**

ALU 指令中的数据冒险

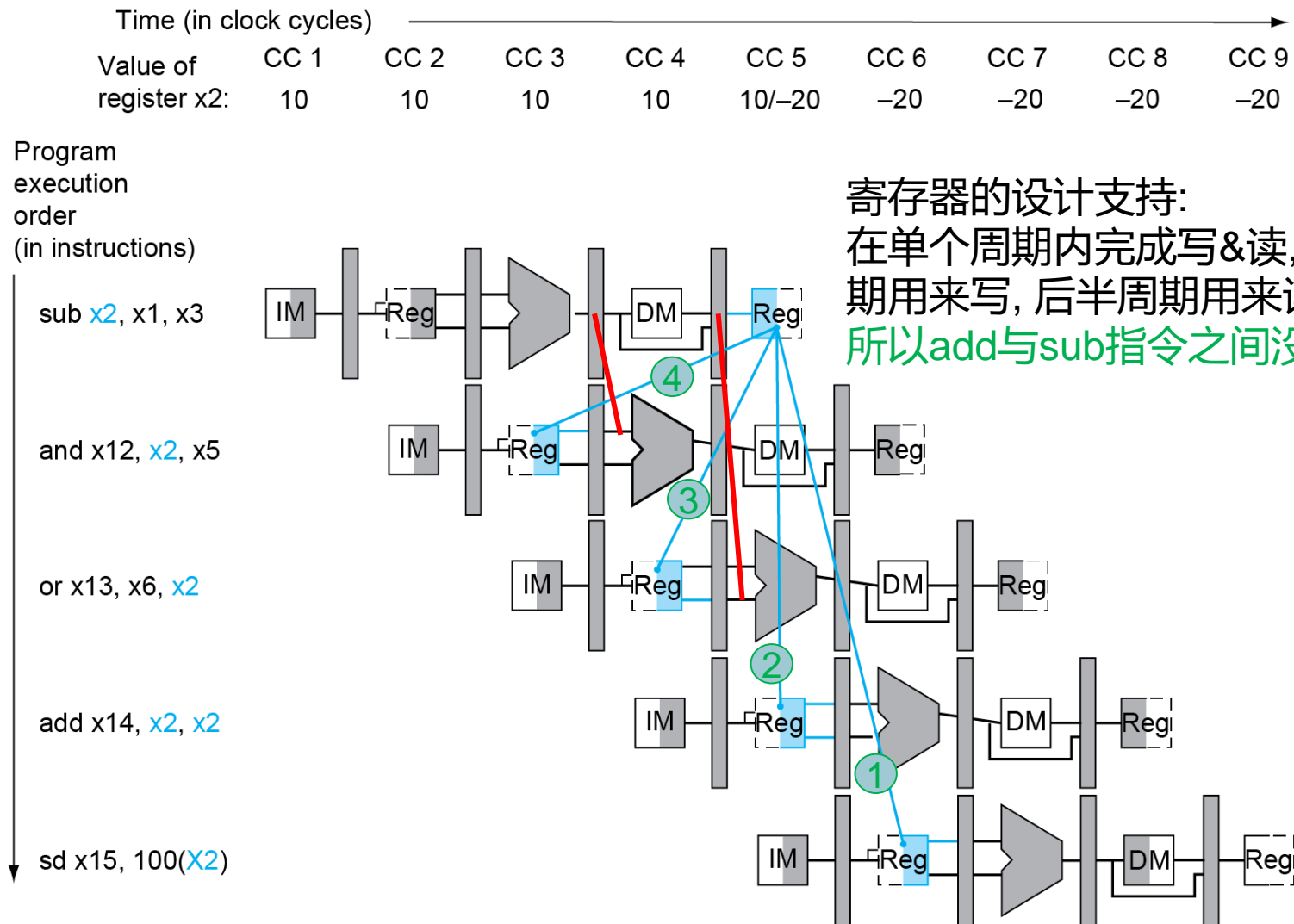
- 例子:

```
sub  x2, x1, x3
and  x12, x2, x5
or   x13, x6, x2
add  x14, x2, x2
sd   x15, 100(x2)
```



- 利用转发，可以解决上面的冒险
 - 但是硬件怎么检测，何时转发？

数据依赖 & 转发



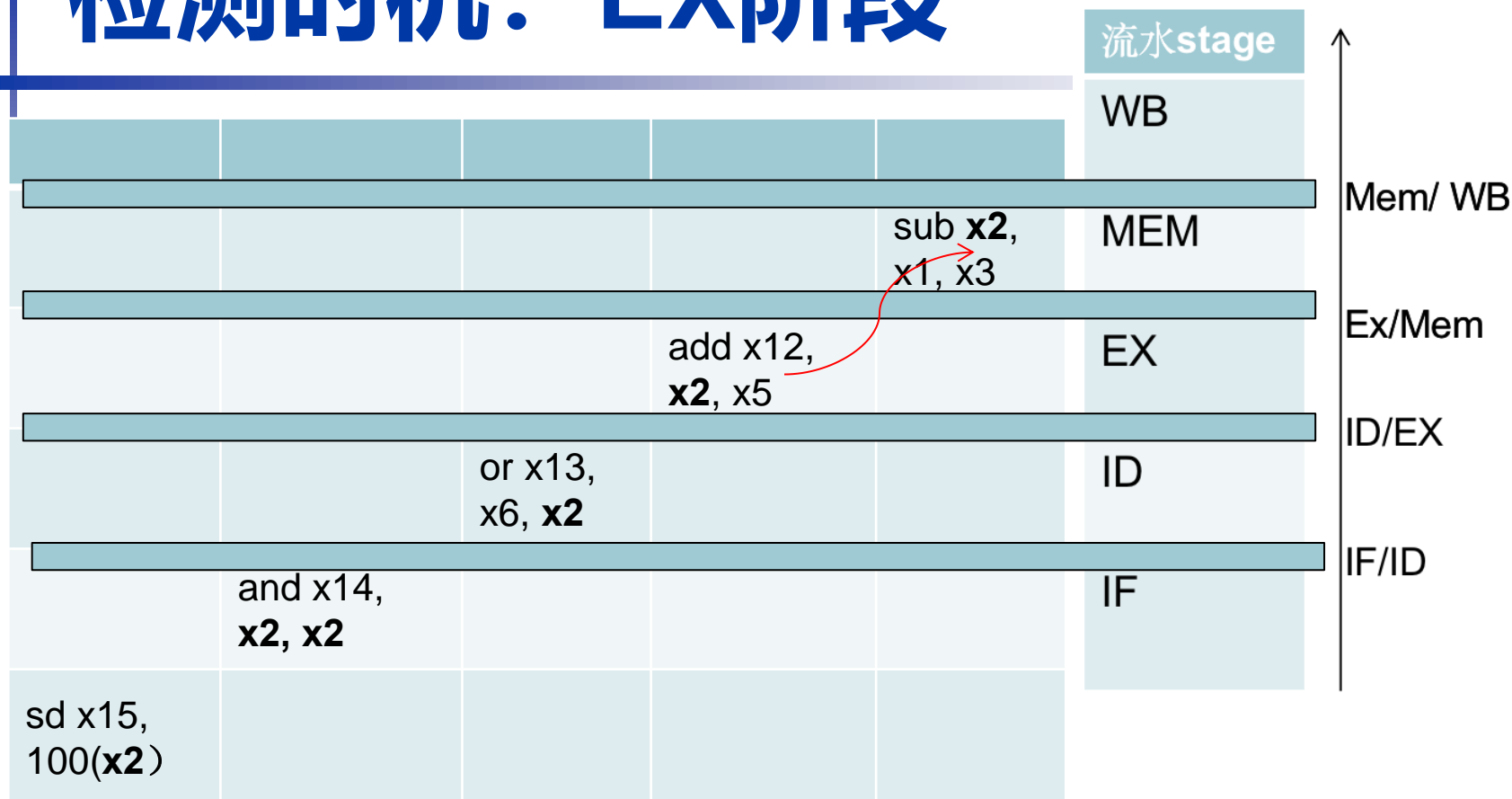
检测是否需要转发

- 指令中的寄存器编号信息rs1,rs2, rd逐级传递
 - ID/EX.RegisterRs1 表示ID/EX流水线寄存器上保存的Rs1信息
- EX级所需的ALU操作数寄存器为
 - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- 如果下列条件成立, 则发生数据冒险
 - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
 - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2
 - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
 - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

Fwd from
EX/MEM
pipeline reg

Fwd from
MEM/WB
pipeline reg

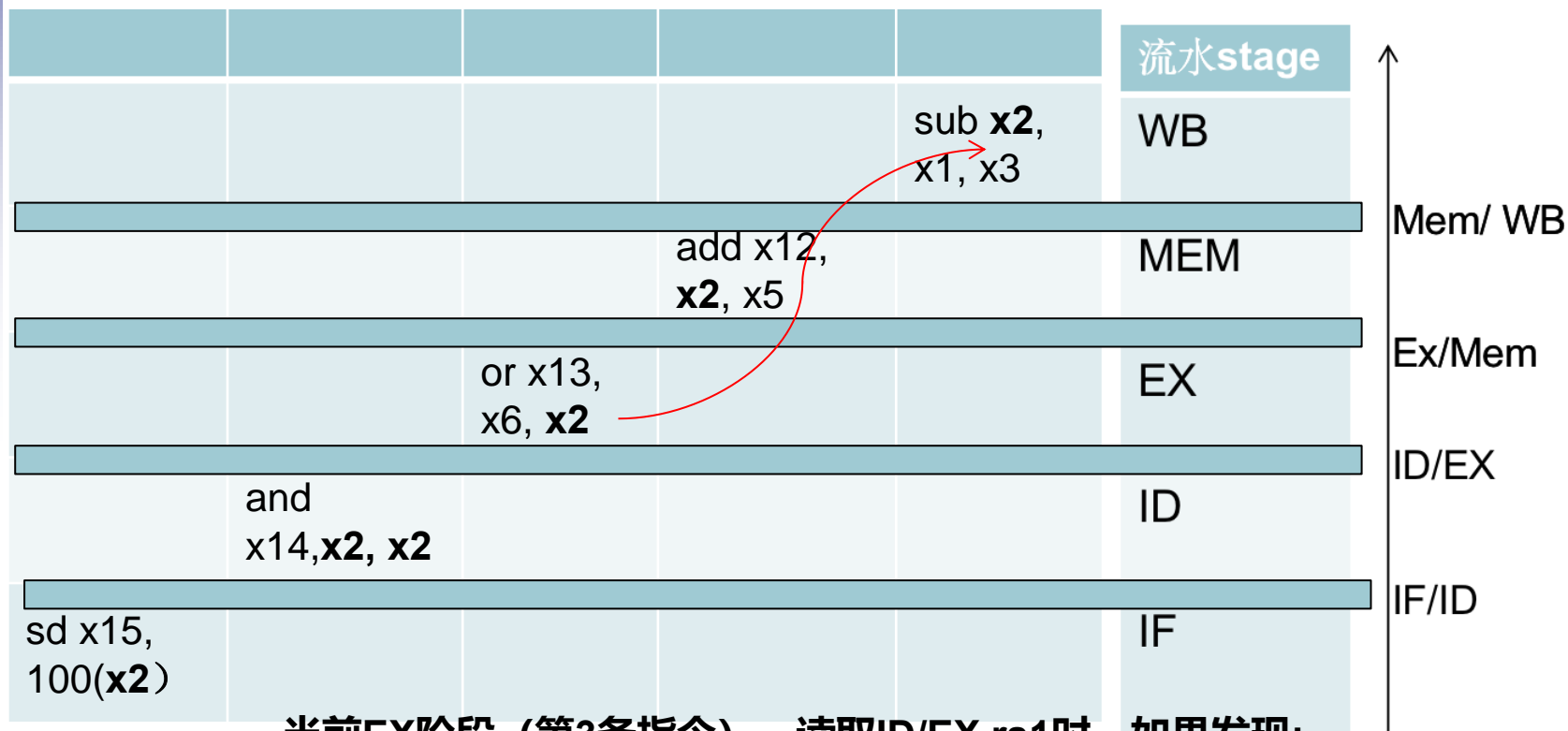
检测时机：EX阶段



当前EX阶段（第2条指令），读取ID/EX.rs1时，如果发现：

- MEM阶段（第1条指令）的Ex/Mem.rd == ID/EX.rs1
- 且MEM阶段需要写寄存器
- 且Ex/Mem.rd != 0
- 则，必有数据冒险

检测时机：EX阶段



当前EX阶段（第3条指令），读取ID/EX.rs1时，如果发现：

- WB阶段（第1条指令）的Mem/WB.rd == ID/EX.rs1，
- 且WB阶段需要写寄存器，
- 且Mem/WB.rd != 0
- 则，必有数据冒险

检测是否需要转发

- 指令中的寄存器编号信息rs1,rs2, rd逐级传递
 - ID/EX.RegisterRs1 表示ID/EX流水线寄存器上保存的Rs1信息
- EX级所需的ALU操作数寄存器为
 - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- 如果下列条件成立, 则发生数据冒险
 - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
 - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2
 - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
 - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

从
EX/MEM
转发

从
MEM/WB
转发

检测是否需要转发，还需要满足

- 转发来源对应的指令确实需要写回寄存器!
 - EX/MEM.RegWrite, MEM/WB.RegWrite
- 并且转发来源对应的指令的Rd不是x0
 - EX/MEM.RegisterRd \neq 0,
MEM/WB.RegisterRd \neq 0

检测是否需要转发的条件

1. *EX hazard*:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

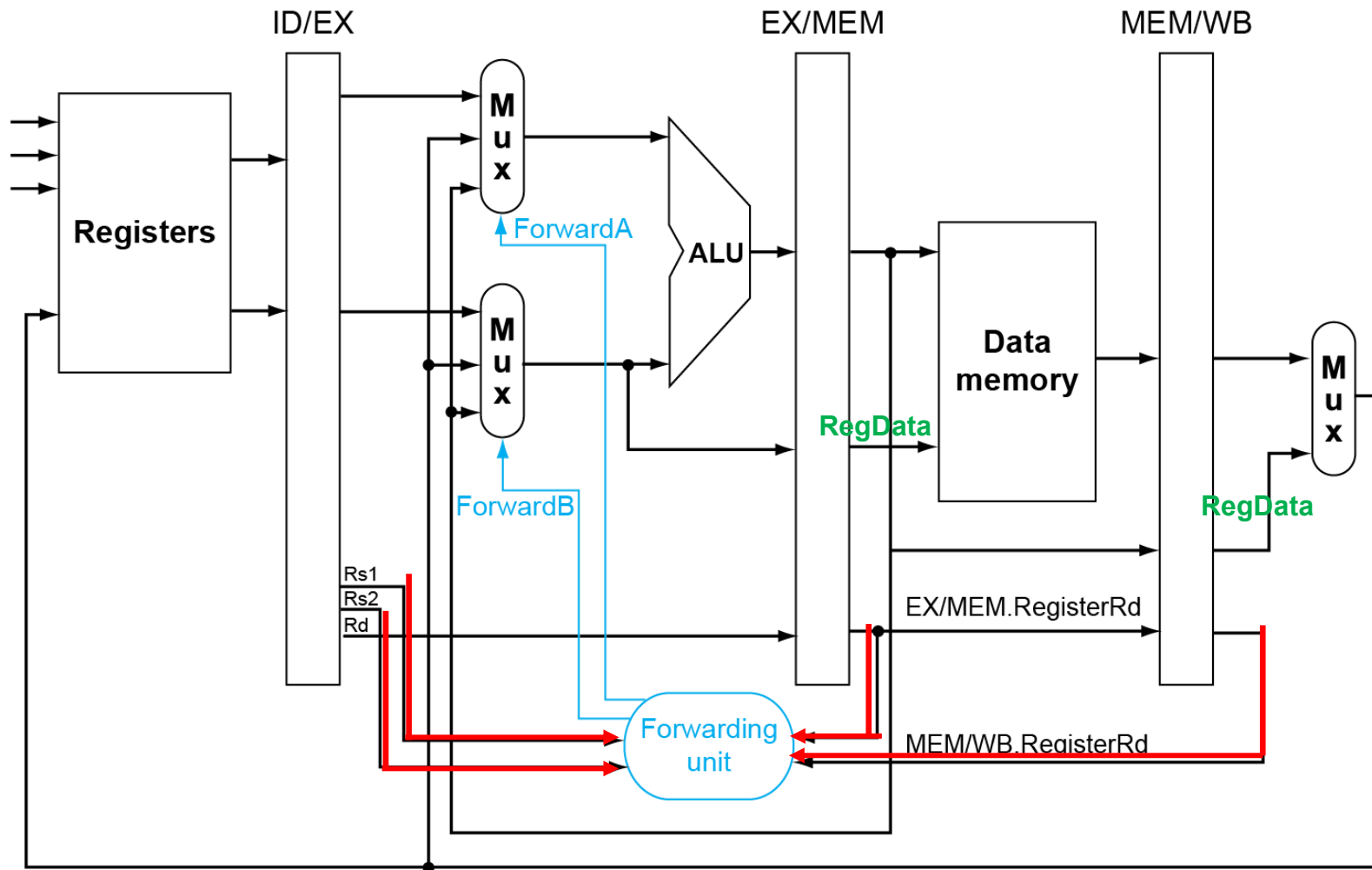
2. *MEM hazard*:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
and (MEM/WB.RegisterRd ≠ 0)
```

转发的条件

选择器控制信号	来源	解释
ForwardA = 00	ID/EX	第1个ALU操作数来自寄存器文件
ForwardA = 10	EX/MEM	第1个ALU操作数转发自 上条 指令的ALU结果
ForwardA = 01	MEM/WB	第1个ALU操作数转发自 上上条 指令的ALU结果
ForwardB = 00	ID/EX	第2个ALU操作数来自寄存器文件
ForwardB = 10	EX/MEM	第2个ALU操作数来自 上条 指令的ALU结果
ForwardB = 01	MEM/WB	第2个ALU操作数转发自 上上条 指令的ALU结果

转发所需的数据通路



转发: 更早地从流水线寄存器中获取所需的值, 而不用等待寄存器文件被正确地更新? 在哪个阶段转发?

EX阶段检测到2个数据冒险

- 例子:

add x1, x1, x2

add x1, x1, x3

add x1, x1, x4

- 检测到2个数据冒险

- 使用最新的x1的值

- 修改MEM冒险条件

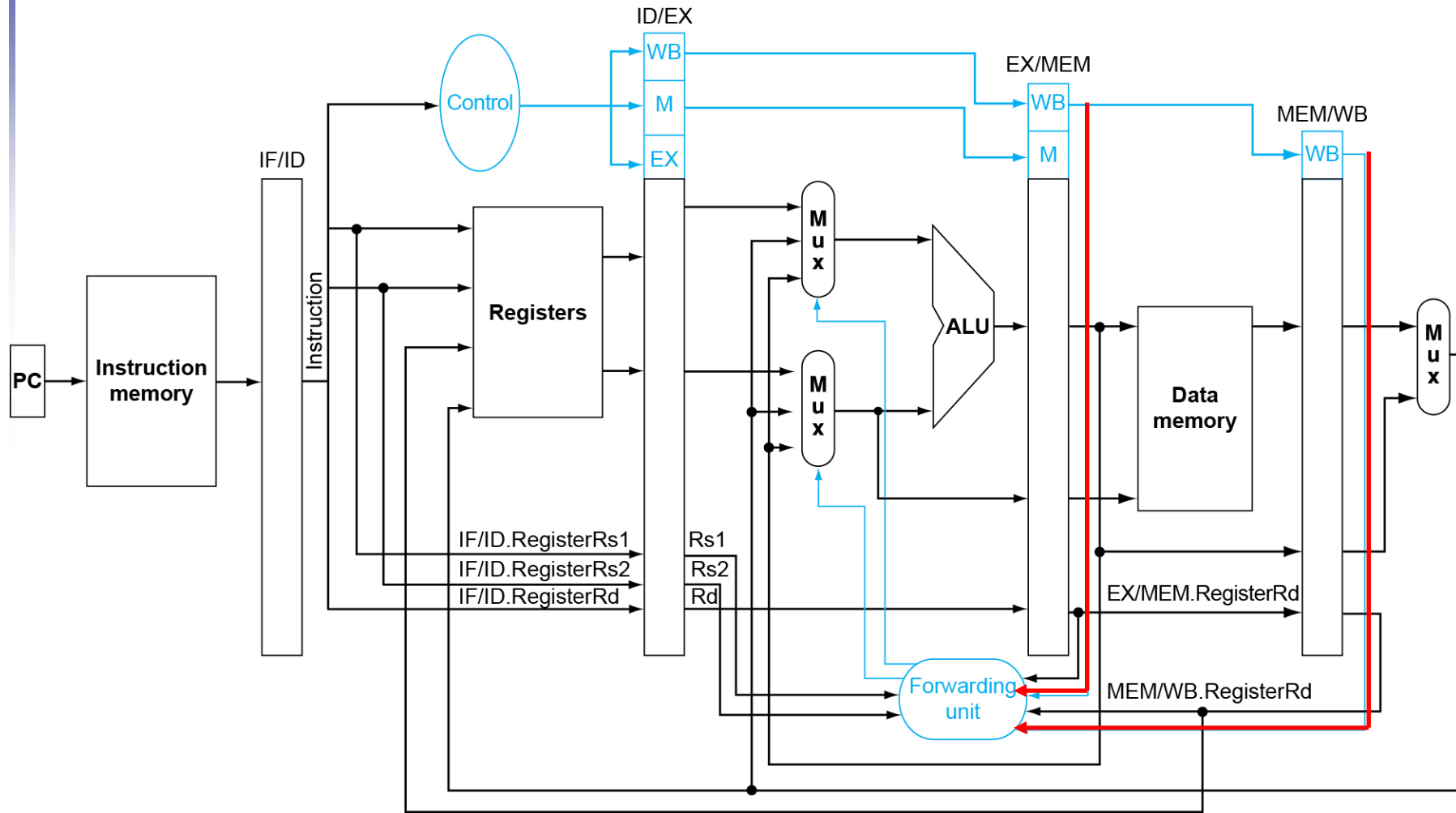
- 只有EX冒险的条件不成立时，才考虑MEM冒险条件

修改后的转发条件

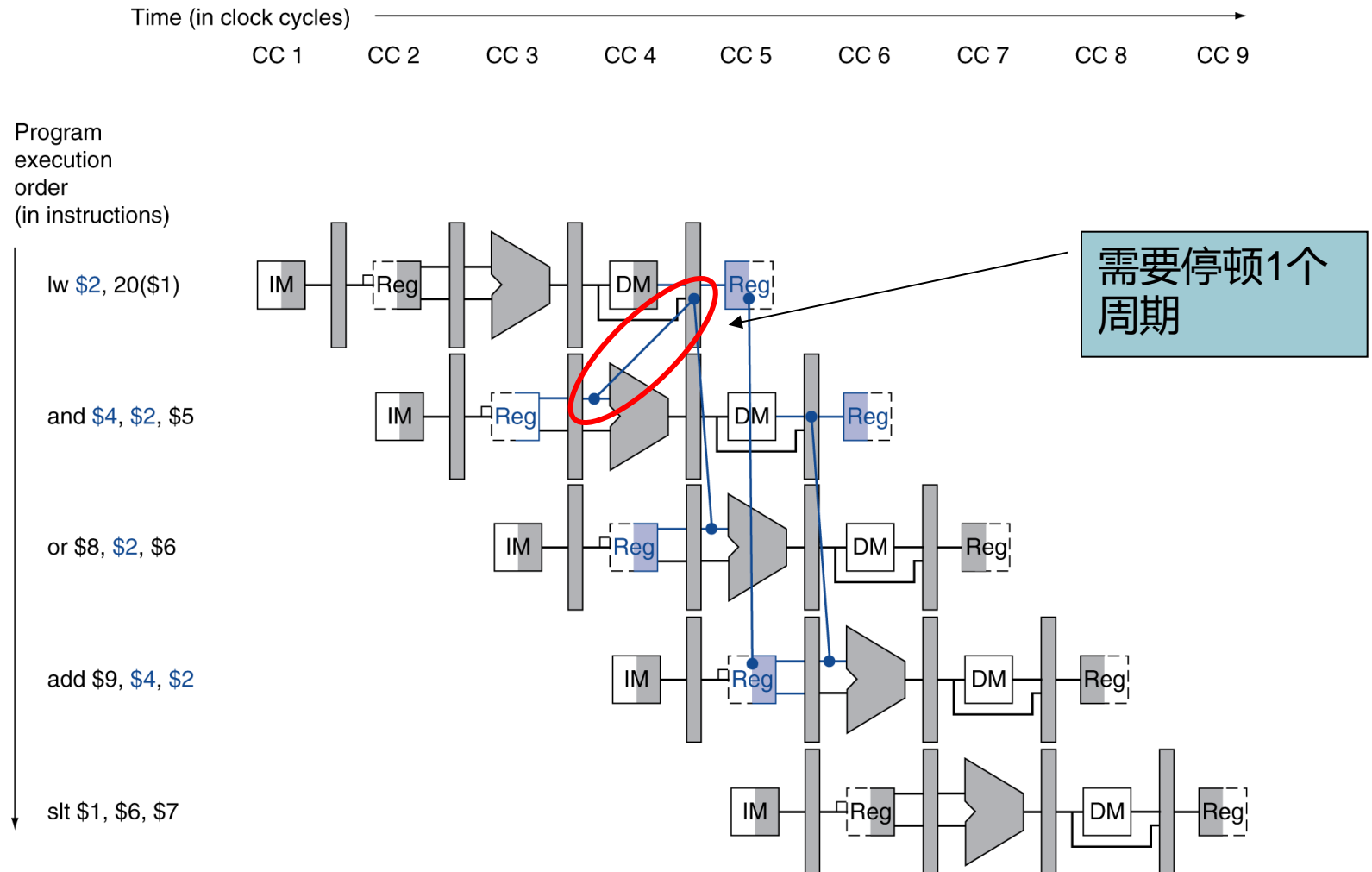
■ MEM 冒险

- if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd \neq 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
- if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd \neq 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

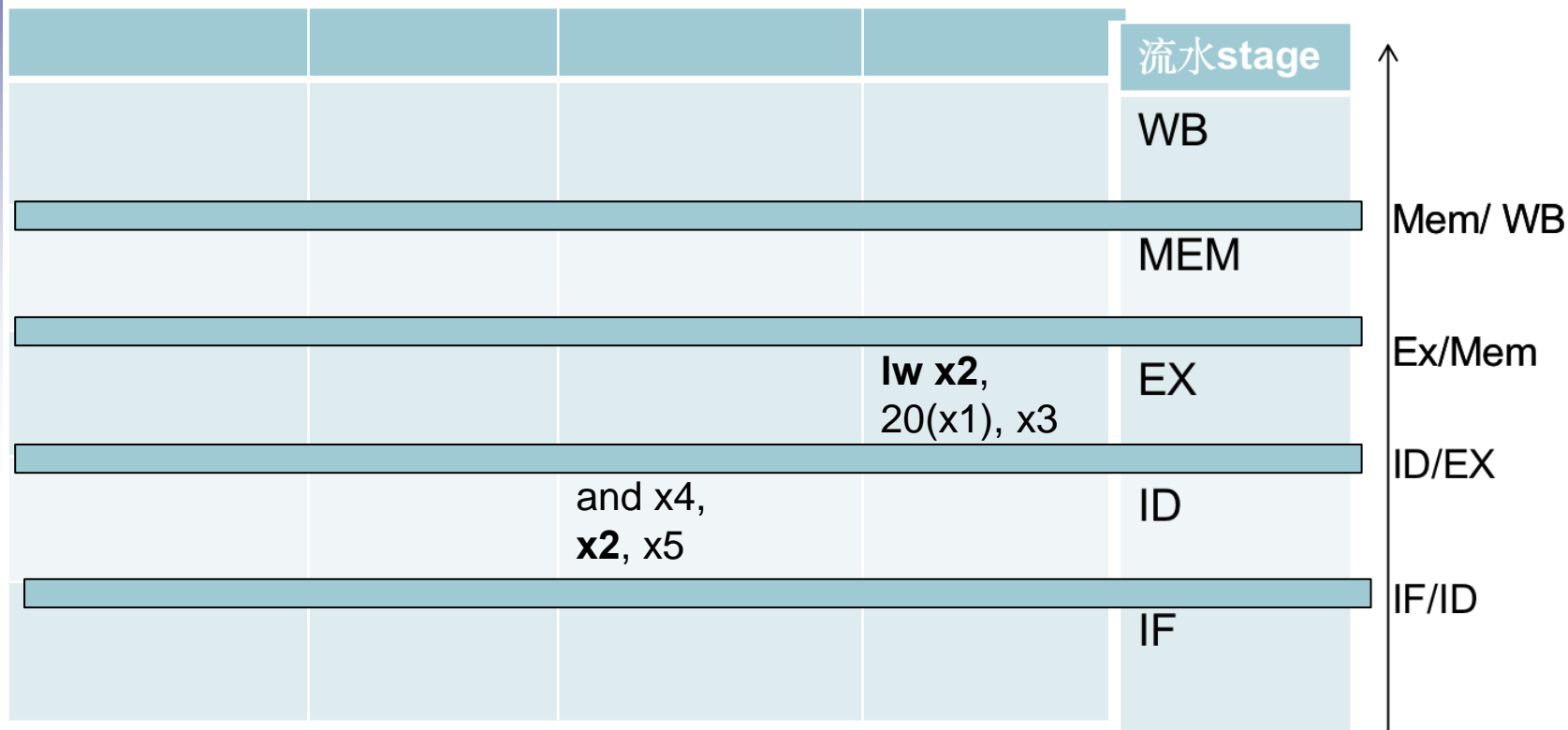
修改后的转发所需数据通路



Load-Use 数据冒险



Load-Use冒险的检测时机：ID阶段



当前ID阶段（第2条指令——use指令），读取IF/ID.rs1时，如果发现：

- EX阶段（第1条指令——load指令）的ID/EX.rd == IF/ID.rs1
- 且ID/EX.memRead
- 则，必有load-use冒险

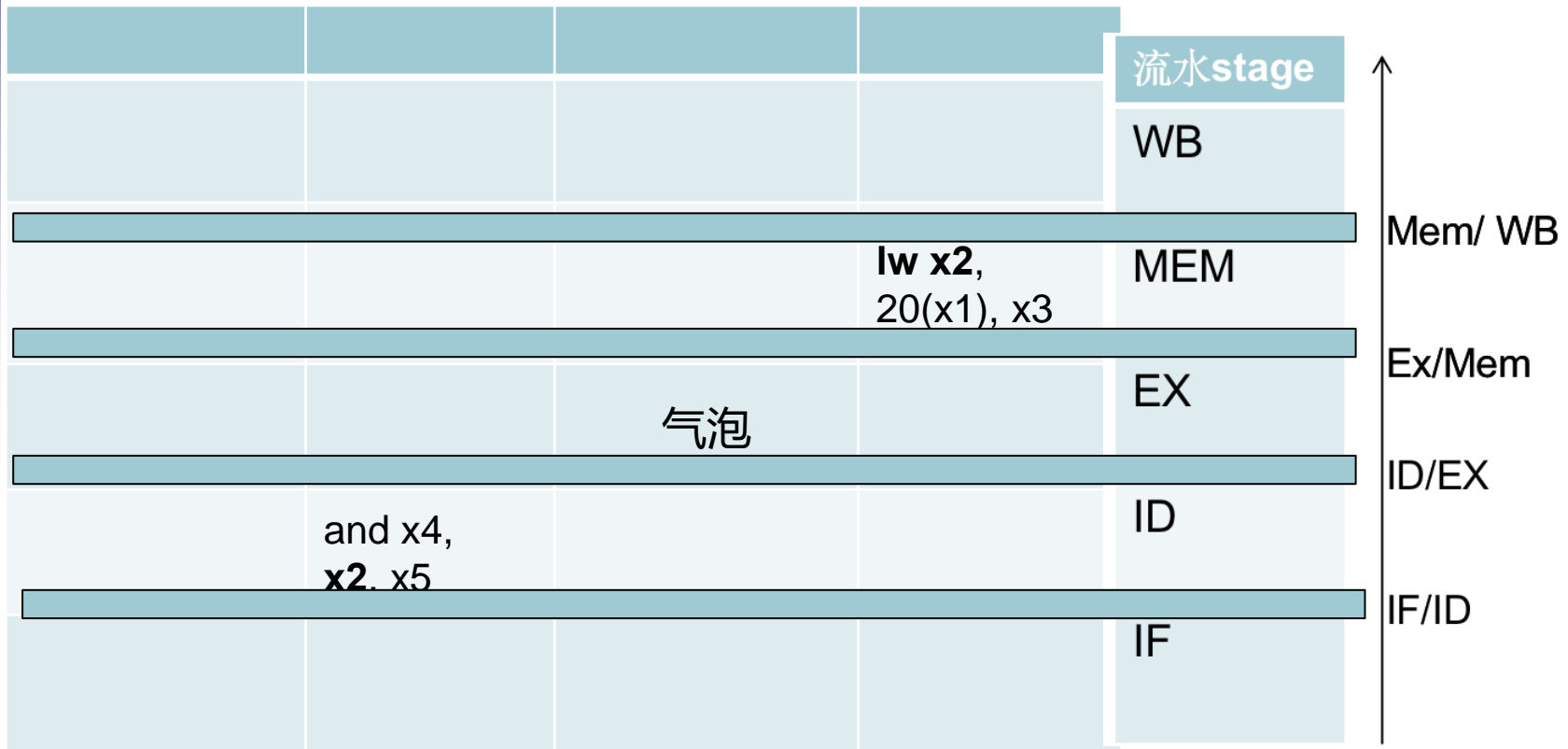
Load-Use冒险的检测

- 当use指令解码时 (ID阶段) 检测
- ALU的操作数寄存器在ID阶段为
 - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- 下列条件成立时, 发生load-use冒险
 - ID/EX.MemRead and
((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
(ID/EX.RegisterRd = IF/ID.RegisterRs2))
- 如果检测到冒险, 停顿 & 插入冒泡
 - 当前在use指令的ID阶段就检测, 能否后移到EX阶段检测?
 - 不能, 因为放在ID阶段可以提前插入冒泡; 放在EX阶段可能跟R-R数据冒险的检测条件混淆, 使得问题复杂化

如何停顿流水线

- **Flush（产生）一个冒泡**
 - ID/EX寄存器中的控制信号为0，清空当前指令
 - EX, MEM 和 WB 传递空操作
- **Repeat（重复）当前指令及后续指令**
 - 阻止PC和IF/ID寄存器的更新
 - 当前指令再次经过ID阶段
 - 随后的指令再次经过IF阶段
 - 1个周期的停顿后，足够load指令在MEM读取数据
 - 下一个周期到达WB阶段，可以同时转发到EX阶段
 - 这时，使用R-R的MEM冒险中的转发逻辑进行检测

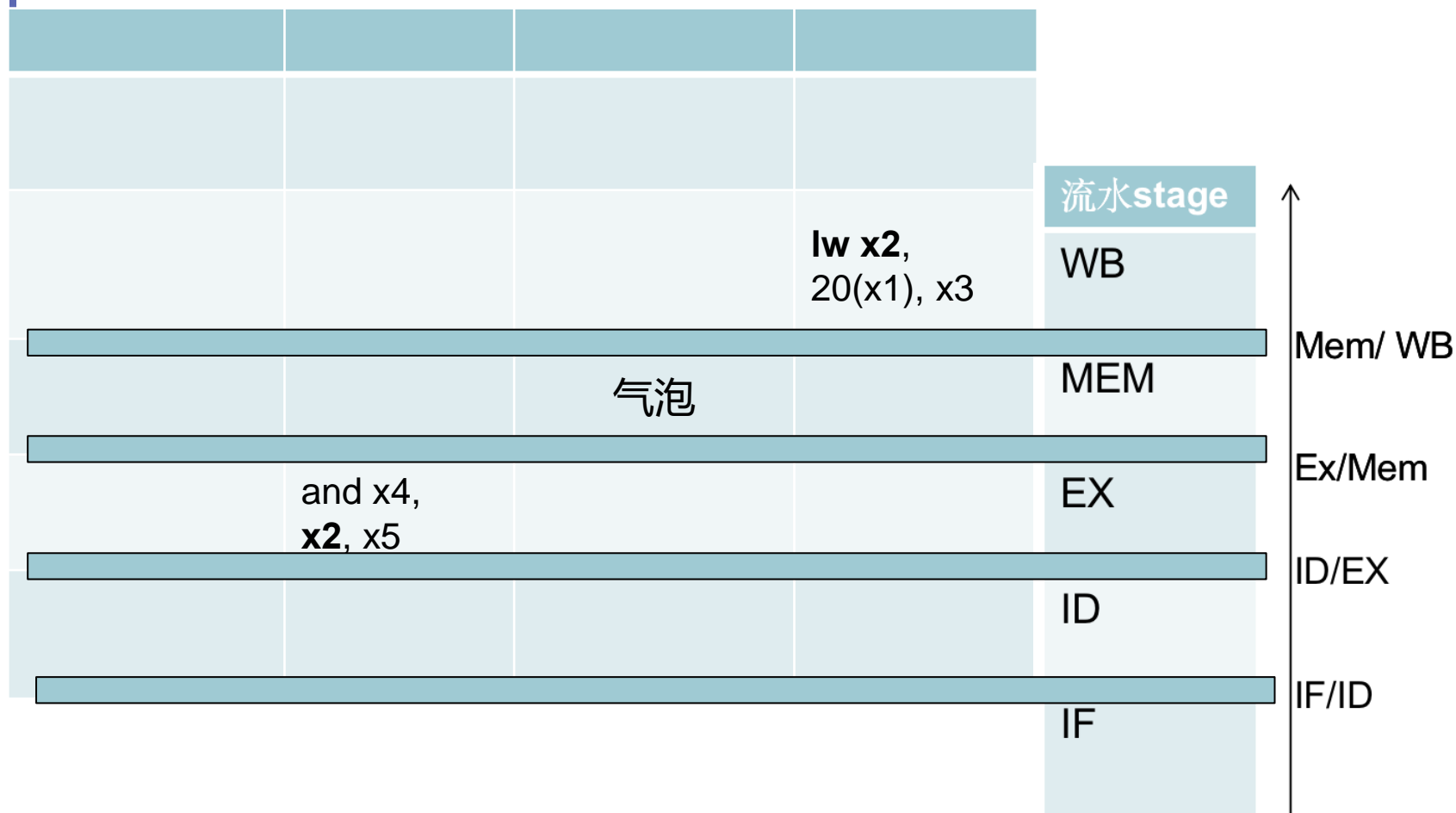
Load-Use冒险的检测&停顿时机：ID阶段



停顿：阻止PC和IF/ID寄存器的更新：

- 当前指令再次经过ID阶段
- 随后的指令再次经过IF阶段
- 1个周期的停顿后，足够load指令在MEM读取数据

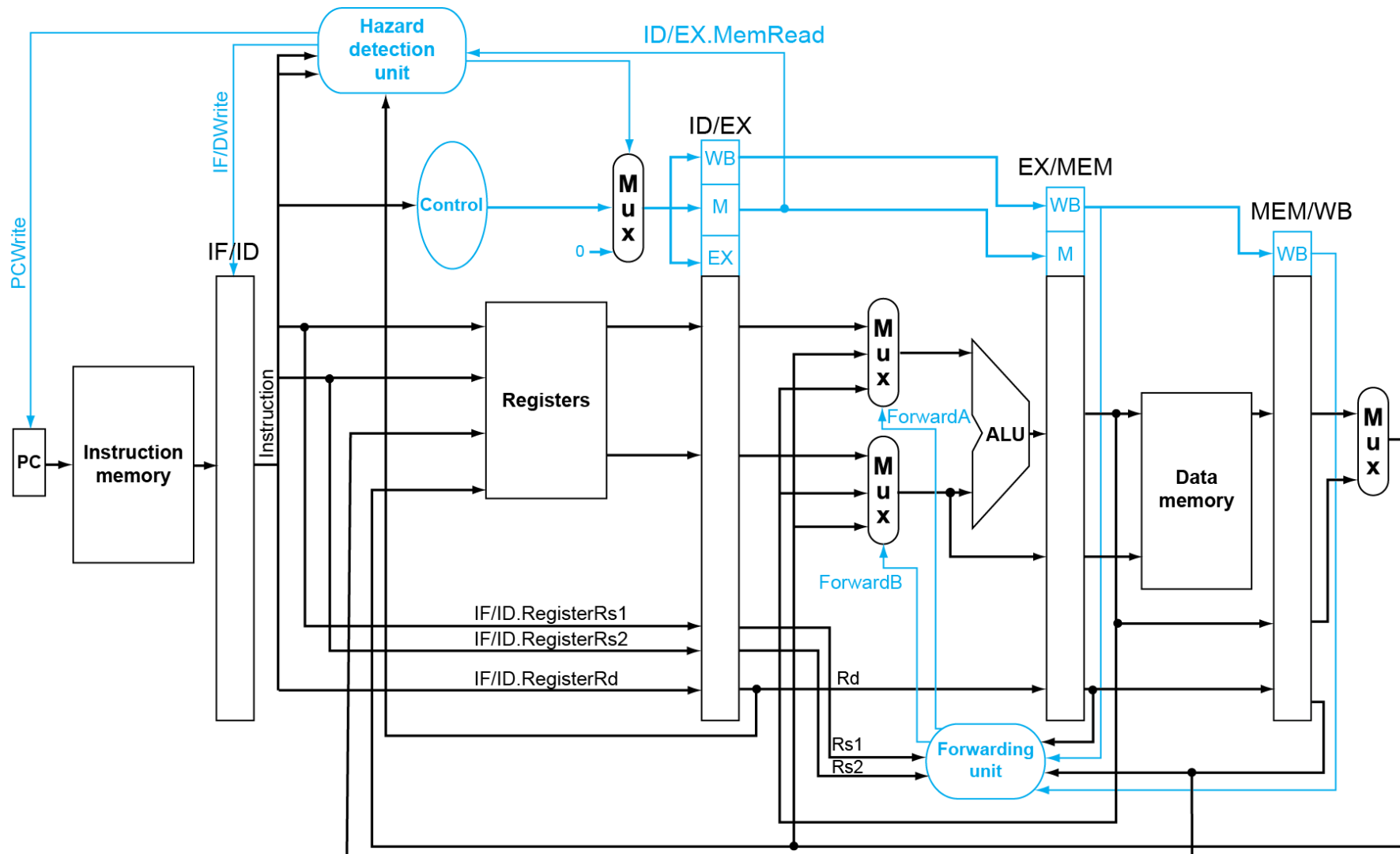
Load-Use冒险停顿之后，利用MEM转发：



1个周期的停顿后，足够load指令在MEM读取数据

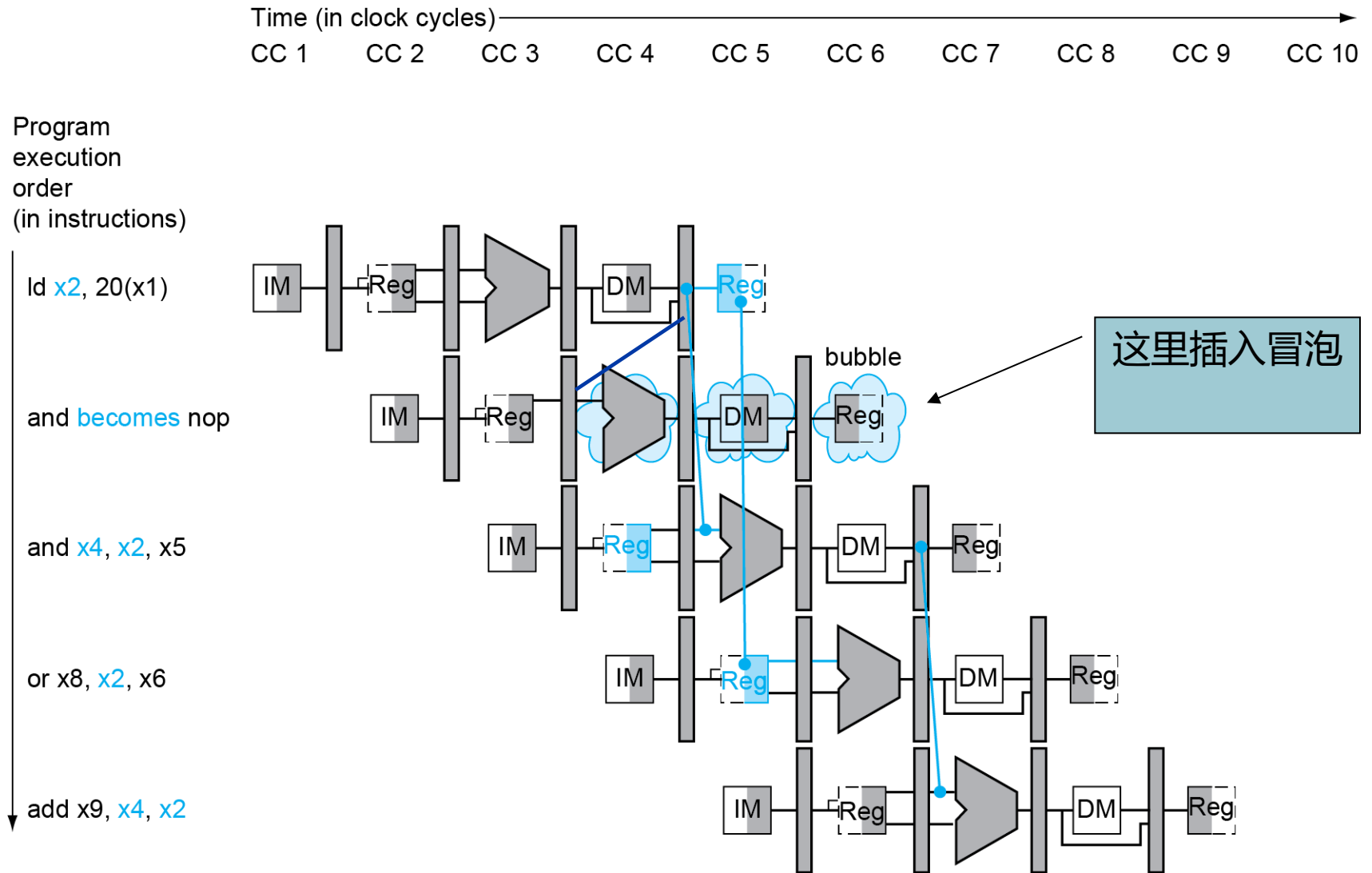
- 下一个周期到达WB阶段，可以同时转发到EX阶段
- 这时，使用R-R的MEM冒险中的转发逻辑进行检测

load-use冒险检测的数据通路



load-use冒险检测单元需要停顿当前及后续指令，
在哪个阶段完成？

Load-Use 数据冒险



停顿和性能的关系

The BIG Picture

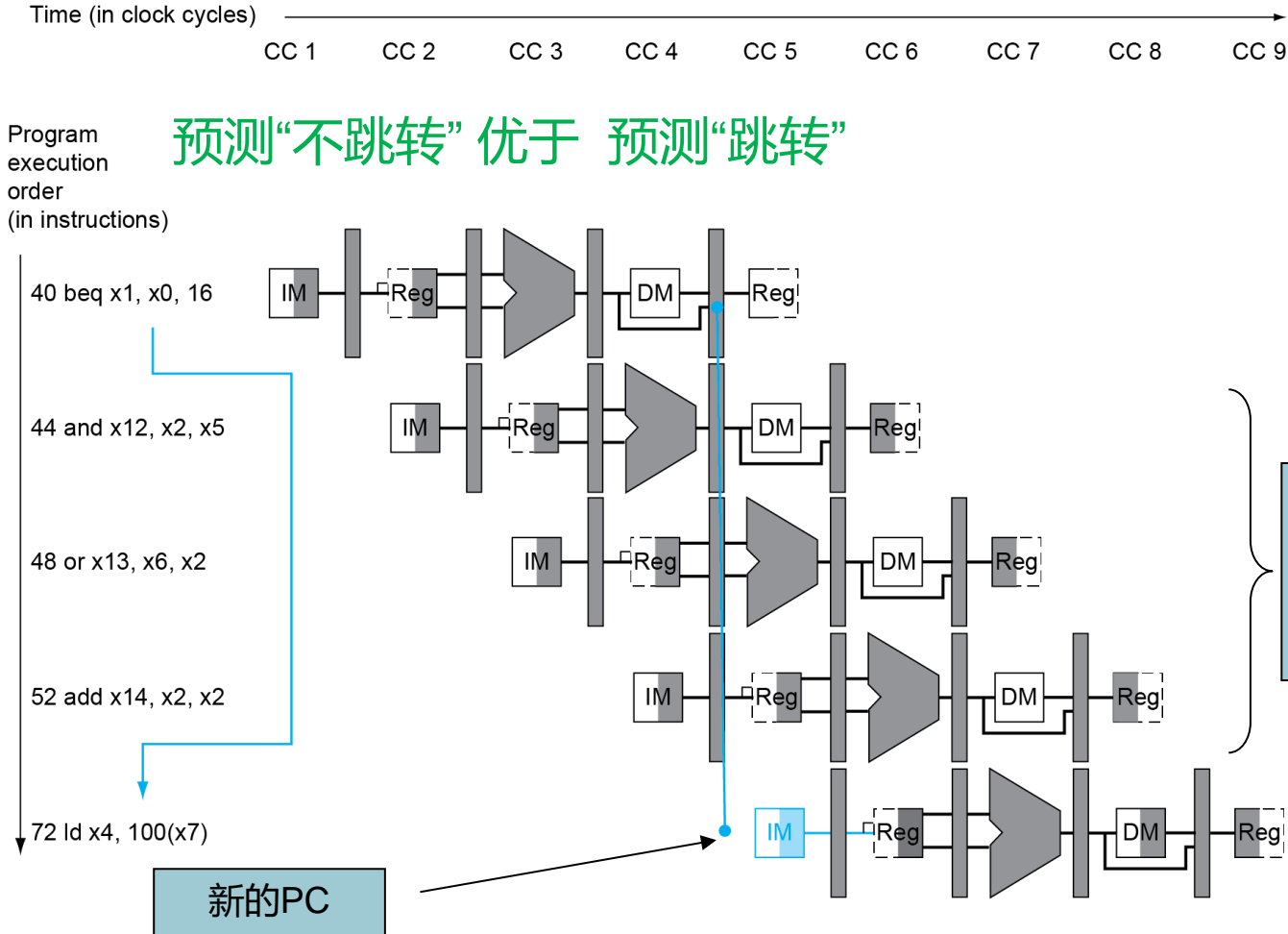
- 停顿会降低性能
 - 但是确保硬件正确所必须的
- 编译器可以重排代码来避免冒险和停顿
 - 需要了解流水线CPU的结构

Outline

- **流水线CPU概述**
- **流水线CPU datapath与控制**
- **流水线CPU 数据冒险的检测与处理**
- **流水线CPU 控制冒险的检测与处理**
- **中断与异常**
- **多发射**

分支冒险

■ 假设在MEM阶段才确定分支的结果



减少分支导致的延迟

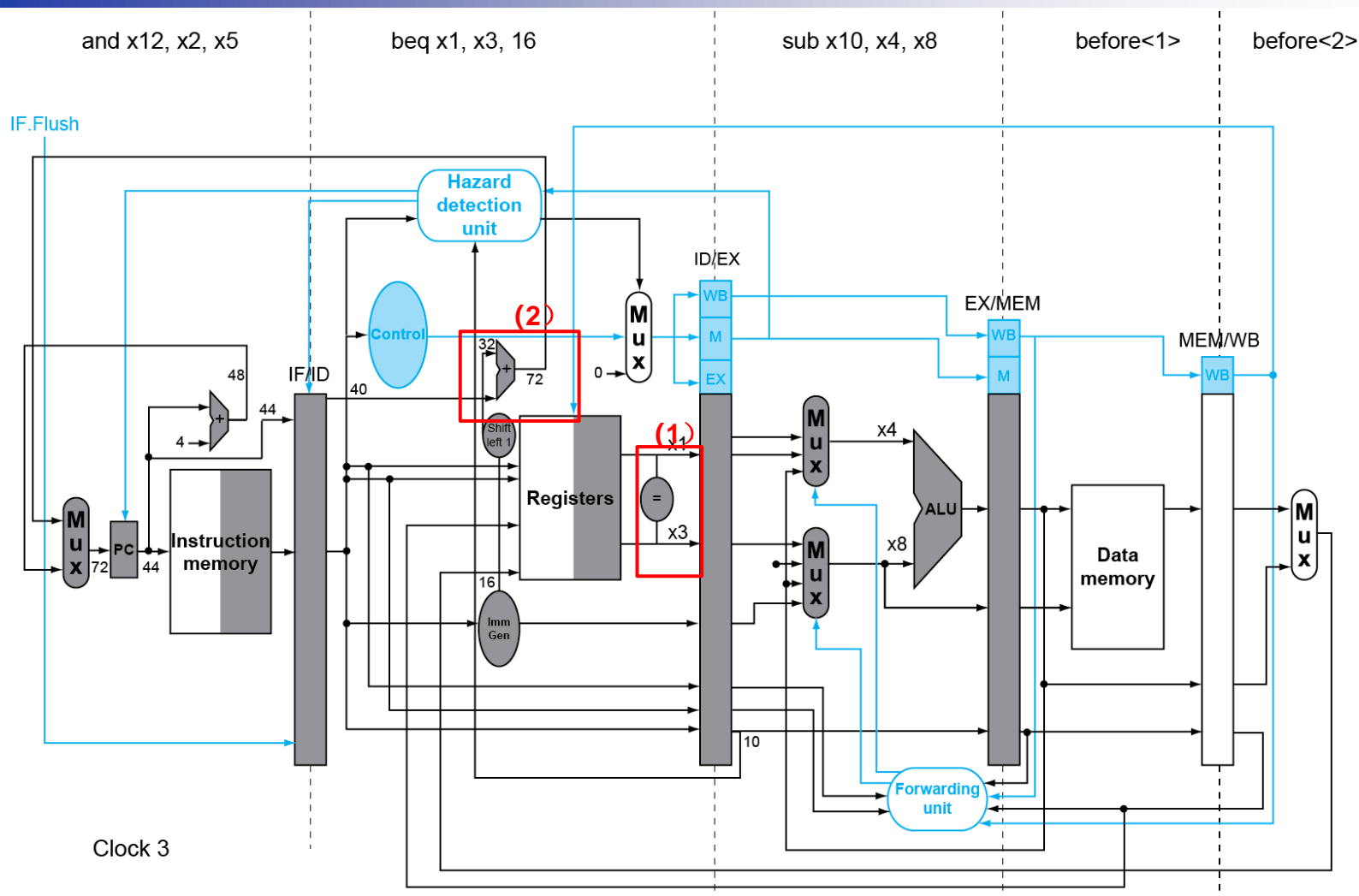
- 修改硬件，提前在ID阶段确定分支的结果(对于深流水线CPU，设计上很困难)
 - 加法器：计算目标地址
 - 寄存器比较器
- 分支跳转的例子：

```
36:  sub  x10, x4, x8
40:  beq  x1,  x3, 16    // 分支指令 (PC-相对寻址)
                        // 分支目标: 40+16*2=72

44:  and  x12, x2, x5
48:  orr  x13, x2, x6
52:  add  x14, x4, x2
56:  sub  x15, x6, x7

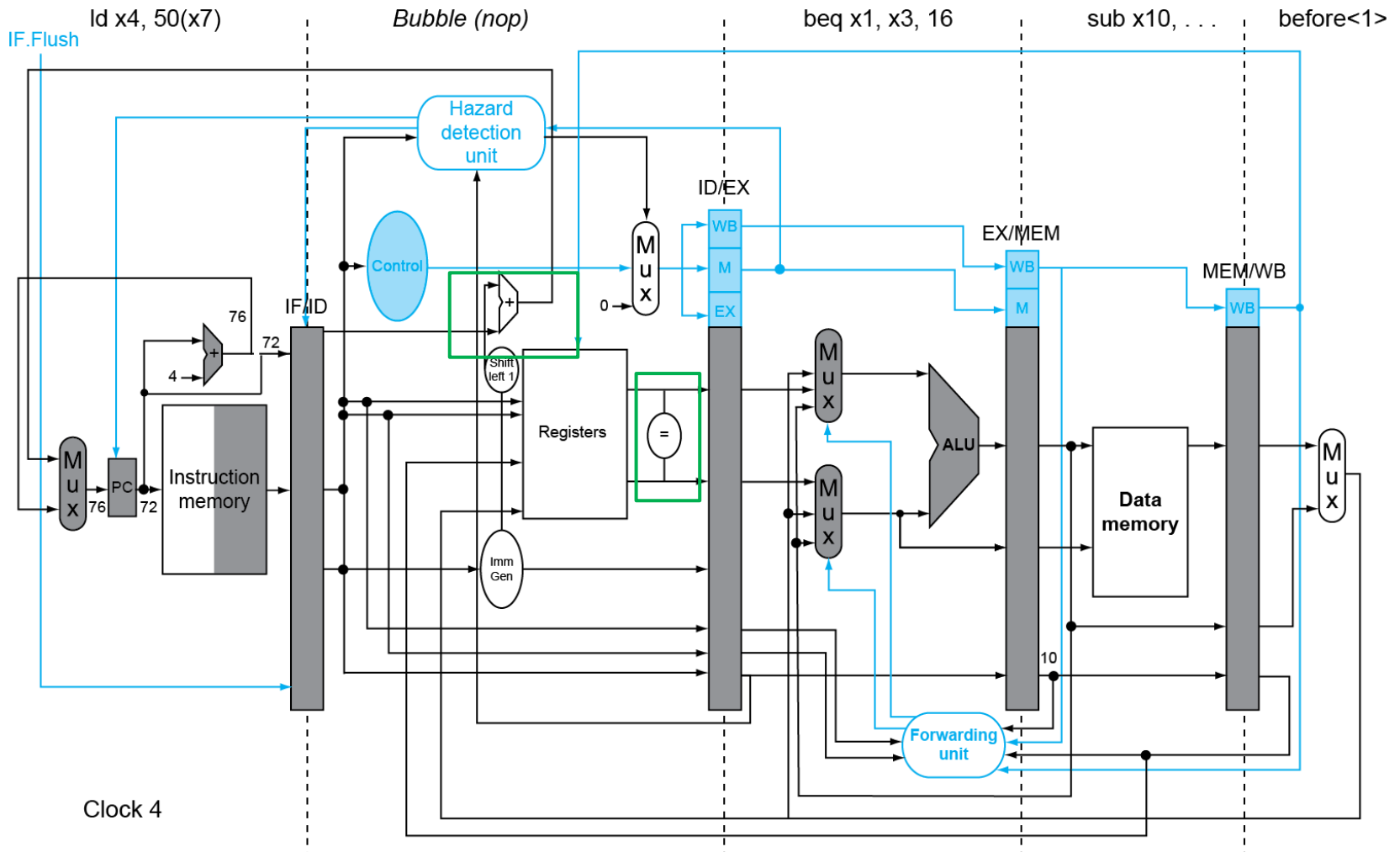
...
72:  ld   x4, 50(x7)
```

分支跳转的例子——第3个时钟



如果ID阶段能够：（1）确定分支结果，（2）计算出跳转地址并更新PC，
则只需清空随后的一条指令：clear IF/ID

分支跳转的例子——第4个时钟

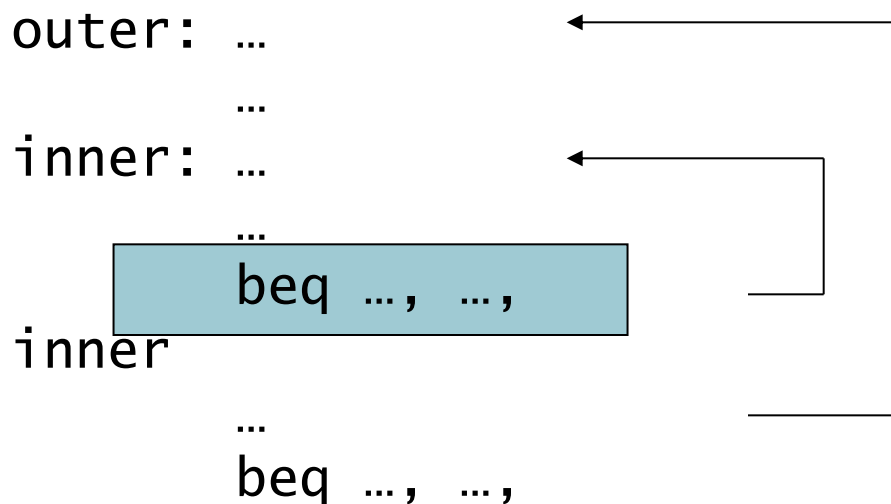


动态分支预测

- 在更深的、超标量流水线，分支惩罚会更明显
- 动态预测来减少惩罚
 - 分支预测缓冲区（作为分支历史记录表）
 - key: 最近的分支指令的地址
 - value: 分支结果（跳转/不跳转）
 - 执行一个分支指令时
 - 检查历史记录表，根据历史来**预测**分支的结果
 - 根据预测结果，开始取指
 - 如果预测失败，则冲刷后续流水线，并更新历史记录

1-Bit 预测器的缺点

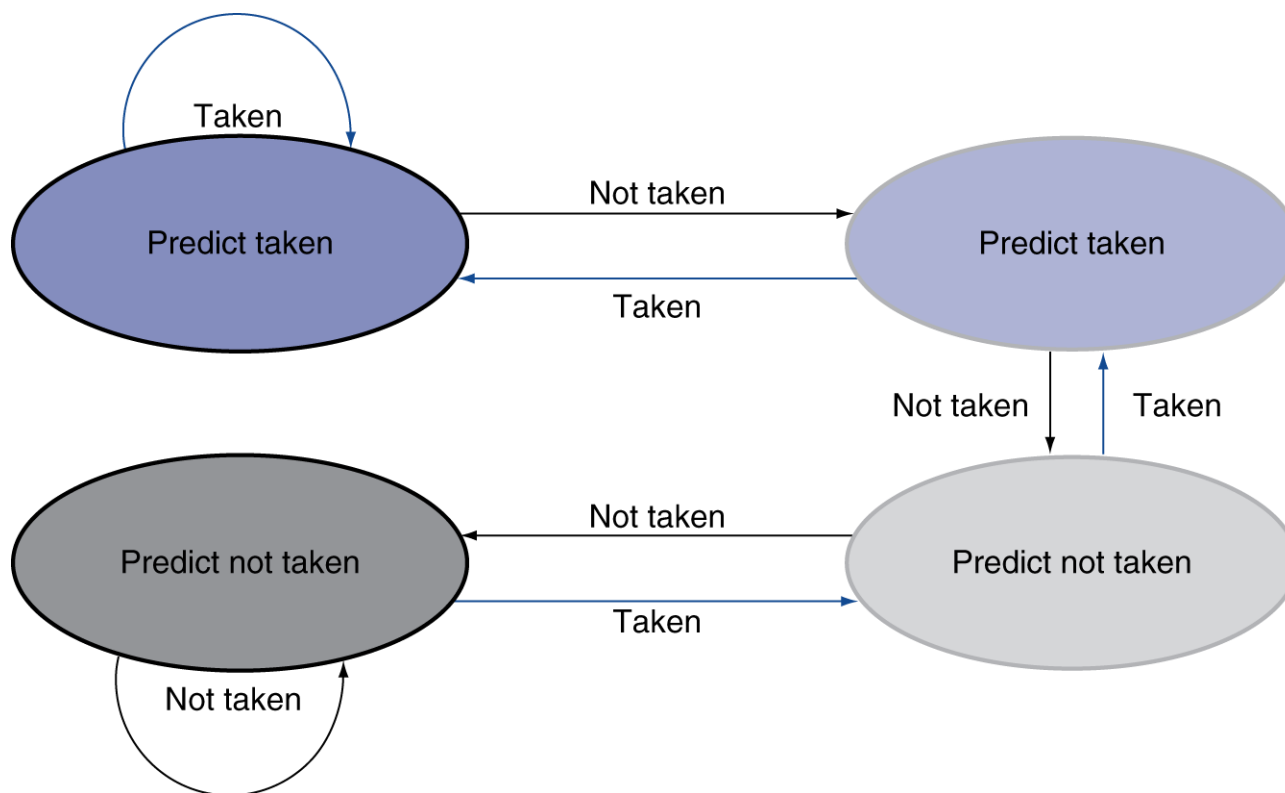
- 内层循环的分支会连续预测失败2次!



- 内层循环的最后一次迭代会预测失败
 - 预测跳转（继续循环），实际不跳转（退出循环）
- 然后下一轮内层循环的第一次迭代会预测失败
 - 预测不跳转，实际跳转

2-Bit 预测器

- 只有连续两次预测失败时，才修改预测方向



计算分支的目标地址

- 即使有预测器，仍需要计算分支的目标地址
 - 1-cycle penalty for a taken branch
 - 对于跳转的分支，会有一个周期的惩罚
- 分支目标缓冲区
 - 目标地址的缓存
 - key: 分支指令的地址
 - value: 分支的目标地址
 - 如果分支跳转，可以立即获取分支的目标地址

总结

- ISA 影响CPU通路和控制的设计
- 通路和控制，也影响ISA的设计
- 流水线通过并行提升了指令的吞吐量
 - 单位时间完成更多的指令
 - 但是单条指令的延时并没有减少
- 三种冒险：结构、数据、控制冒险
- 多发射和动态调度（指令级并行）
 - 指令间的依赖限制了并行性
 - 硬件的复杂性导致了功耗墙

hazard检测小结 (相邻2条指令)

