The RISC-V Processor Implementation: Pipeline——ILP

“Computer Organization & Design”

David Patterson, John Hennessy

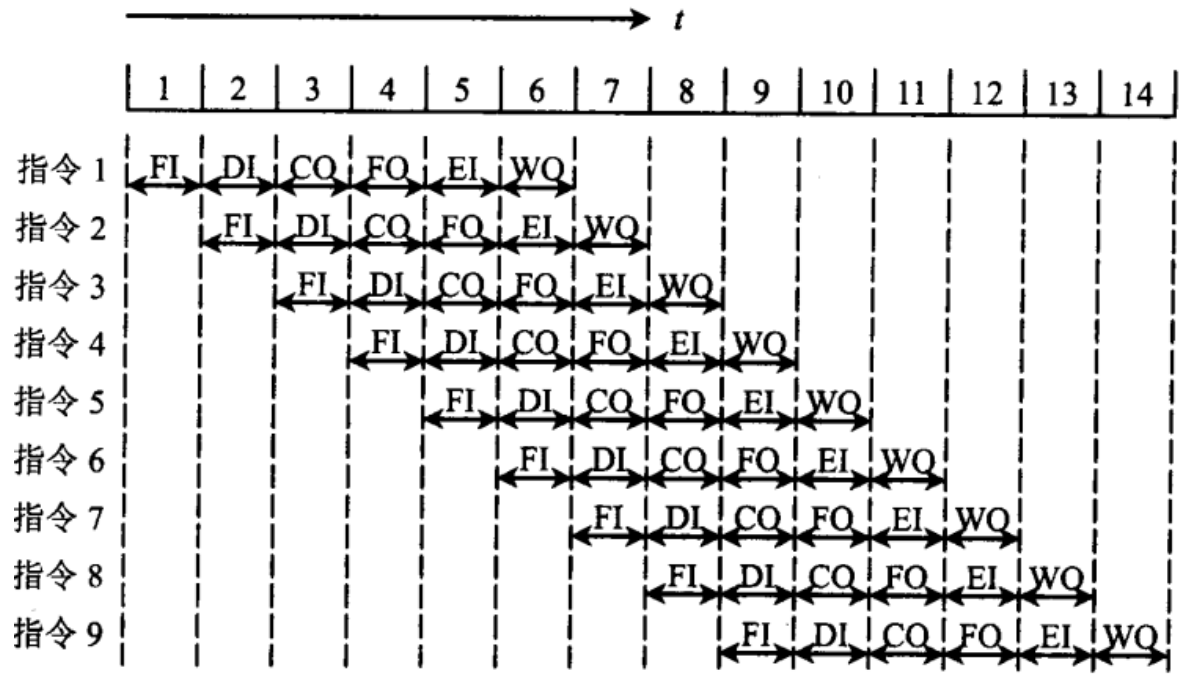
CPI = 1，CPI ≥ 1？

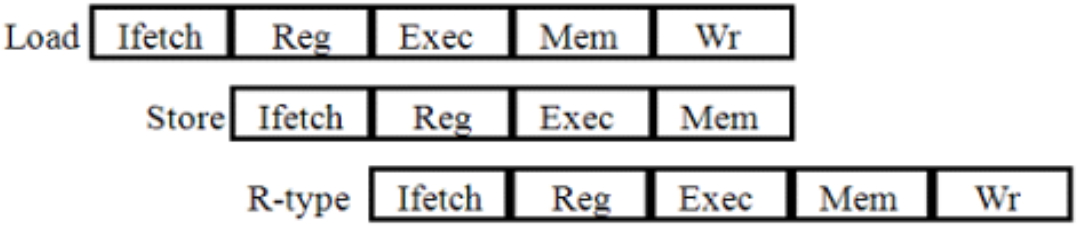
Pipelining Idealism：

平衡，重复，无冲突

Speedup = k

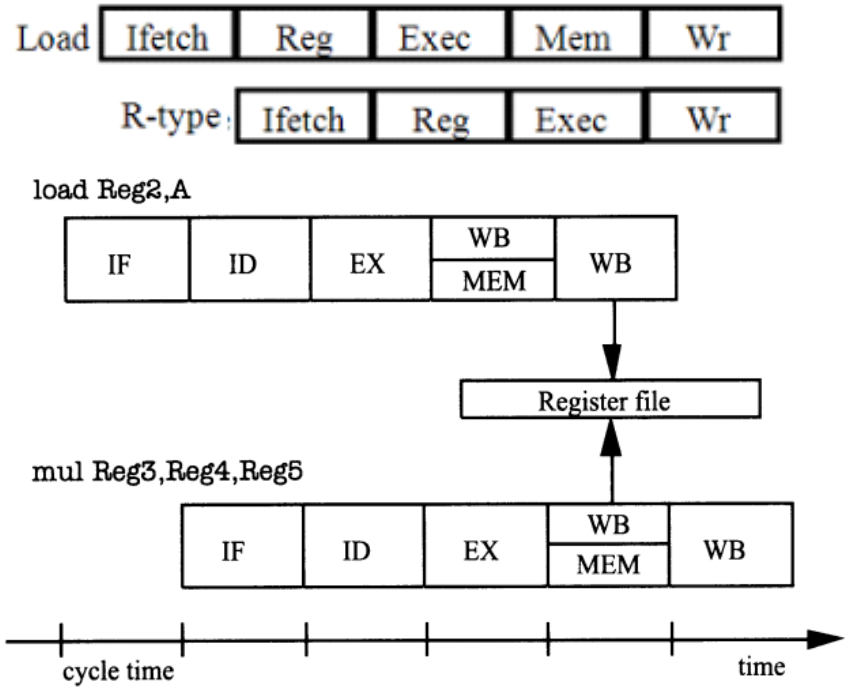
CPI ≤1，IPC ≥ 1?

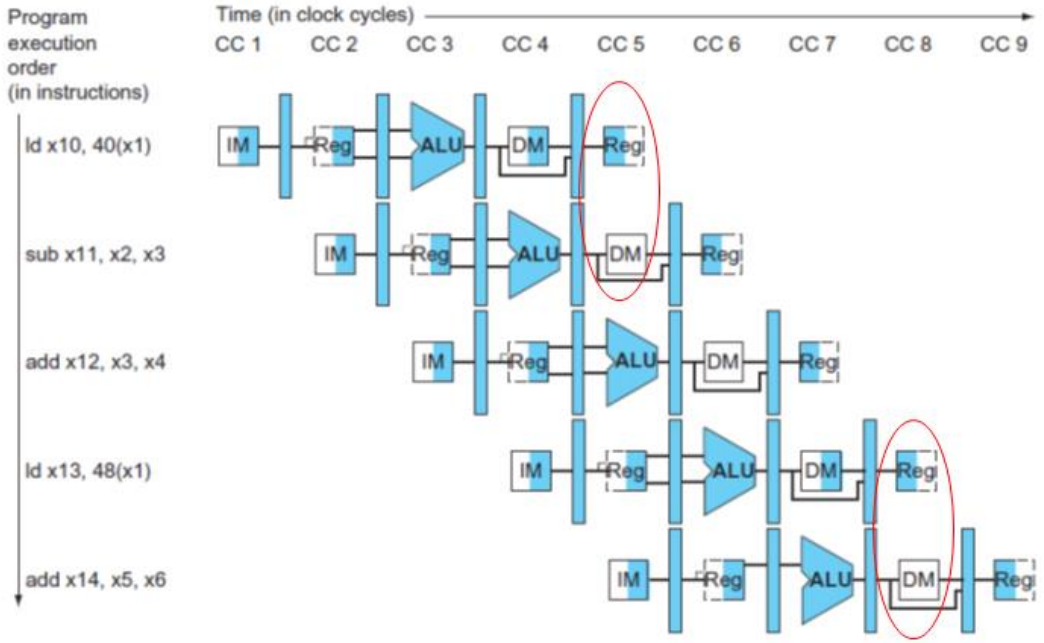




R-type指令完成时间：load-Rtype

- **R-type指令4个周期完成？**
  - **例：当前指令为 lw，下一条指令为R-type。CC5?**





哪个周期数据可用？

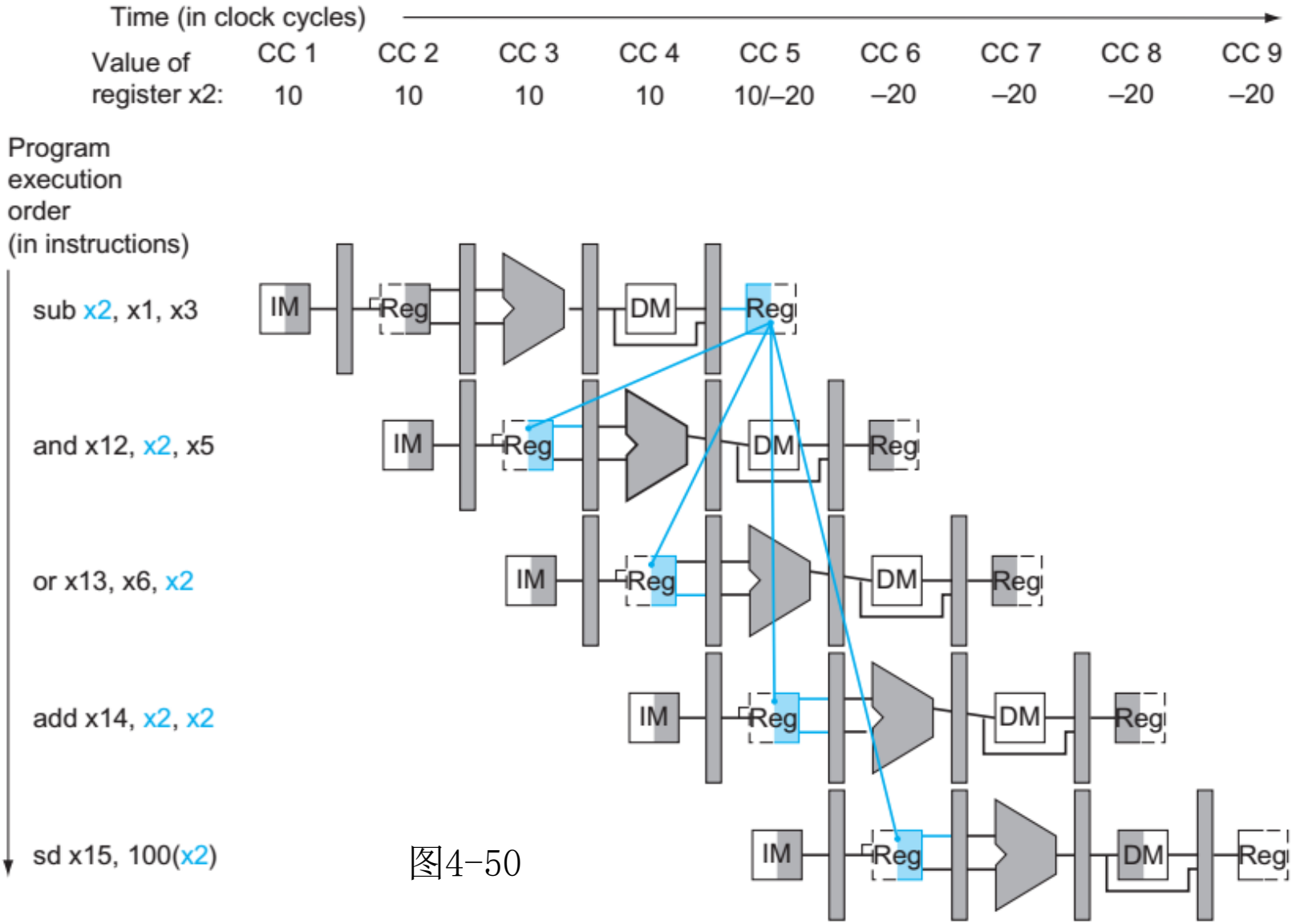


图4-50

beq taken?

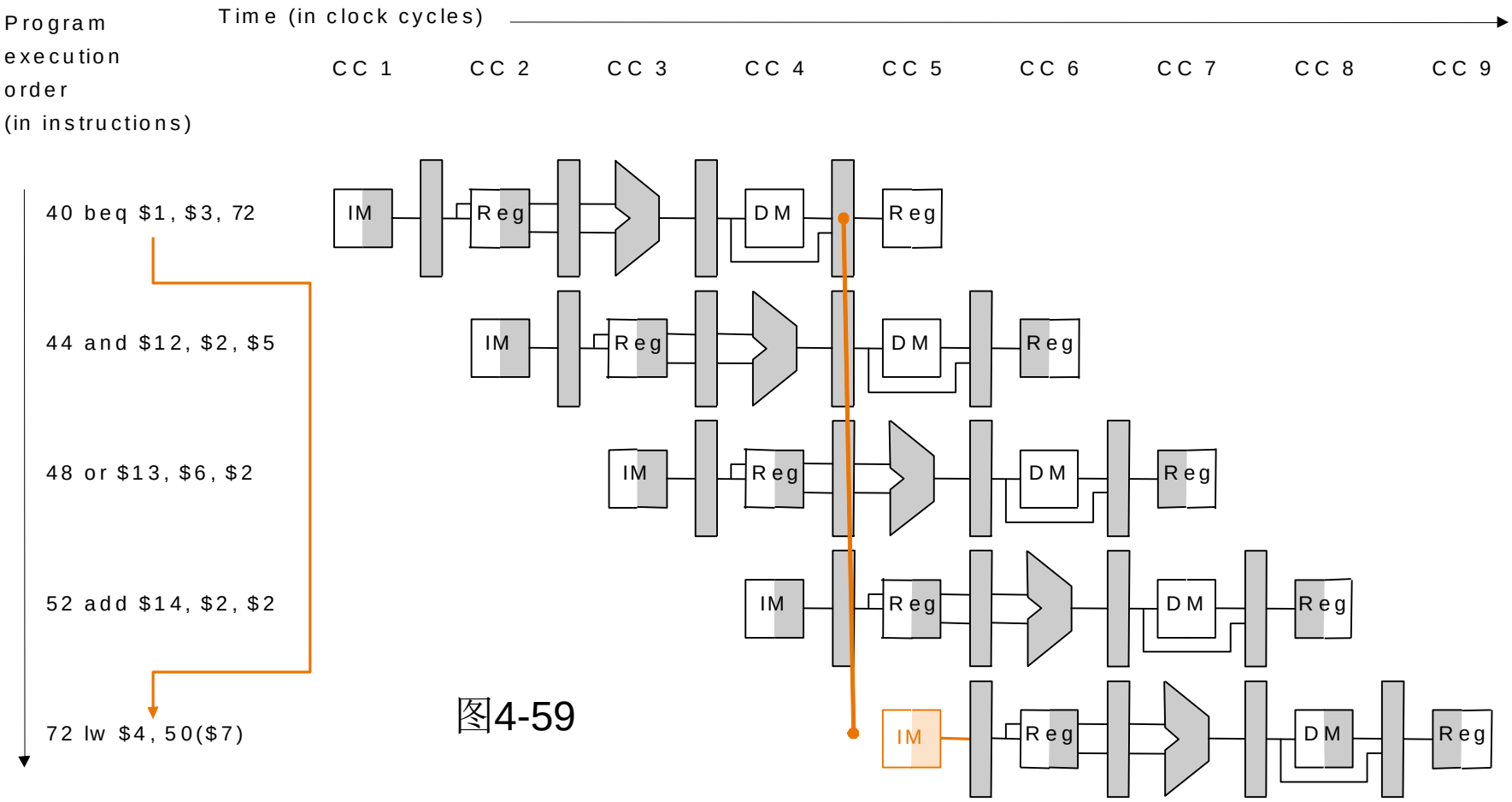
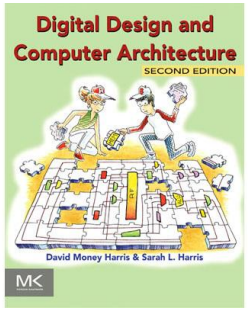
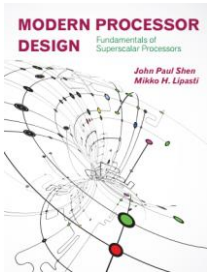


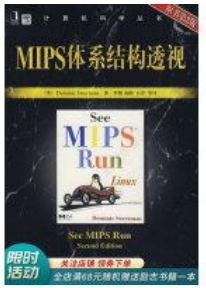
图4-59

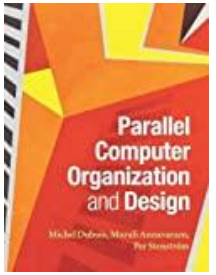
内容提要

- 流水线技术原理：4.6
- RV的五级流水线实现：4.6.1，4.7
  - Verilog行为级定义：4.14
- Hazard问题：4.6.2
  - 结构冲突：哈佛结构
  - 数据依赖：4.8
    - 编译技术：插入nop，指令重排，寄存器重命名
    - forwarding技术：RAW
    - Interlock技术：Stall
  - 控制相关：4.9
    - 编译技术：延迟分支
    - 硬件优化：提前完成，投机，预测
  - 异常：4.10
- 多发射技术：4.11
- 硬件多线程：6.4
- 多核：6.5







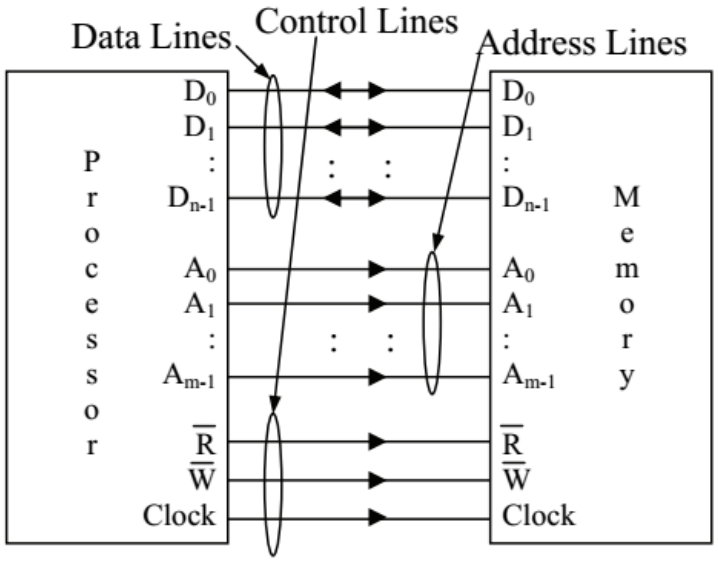


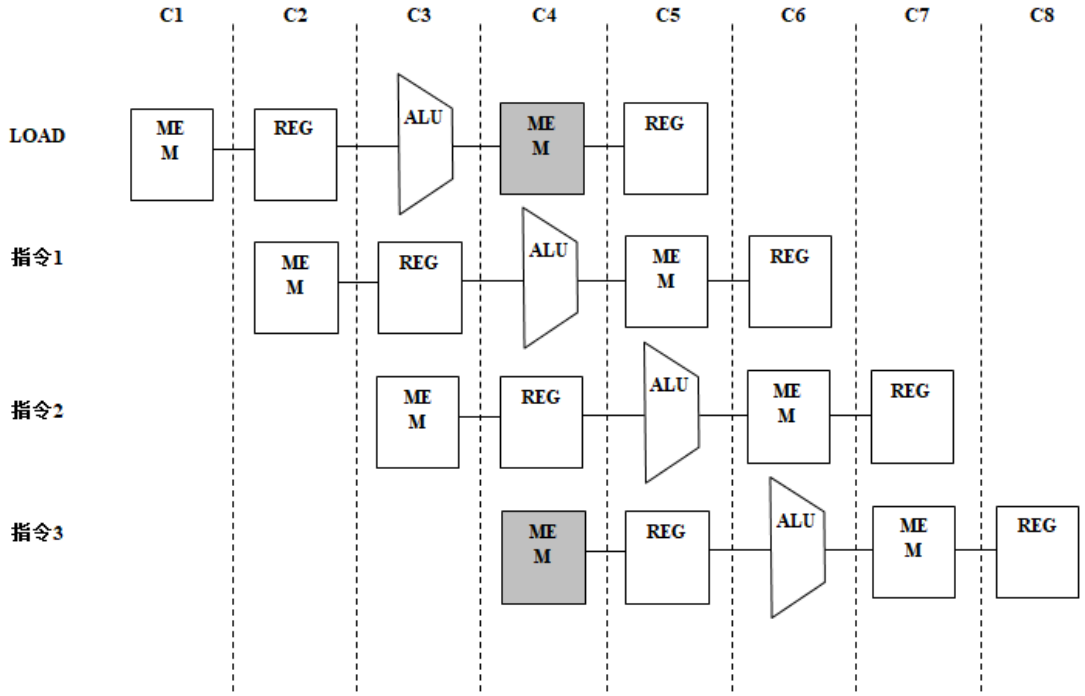
流水线的“hazard冒险”问题

- 结构相关：当指令在重叠执行的过程中，硬件资源满足不了指令重叠执行的要求，发生资源冲突时将产生结构相关。
- 数据相关：当一条指令需要用到前面指令的执行结果，而这些指令均在流水线中重叠执行时，就可能引起数据相关。
- 控制相关：当流水线遇到分支指令和其他会改变PC值的指令时，会发生控制相关。
- ILP：资源confict，数据dependency，控制penalty

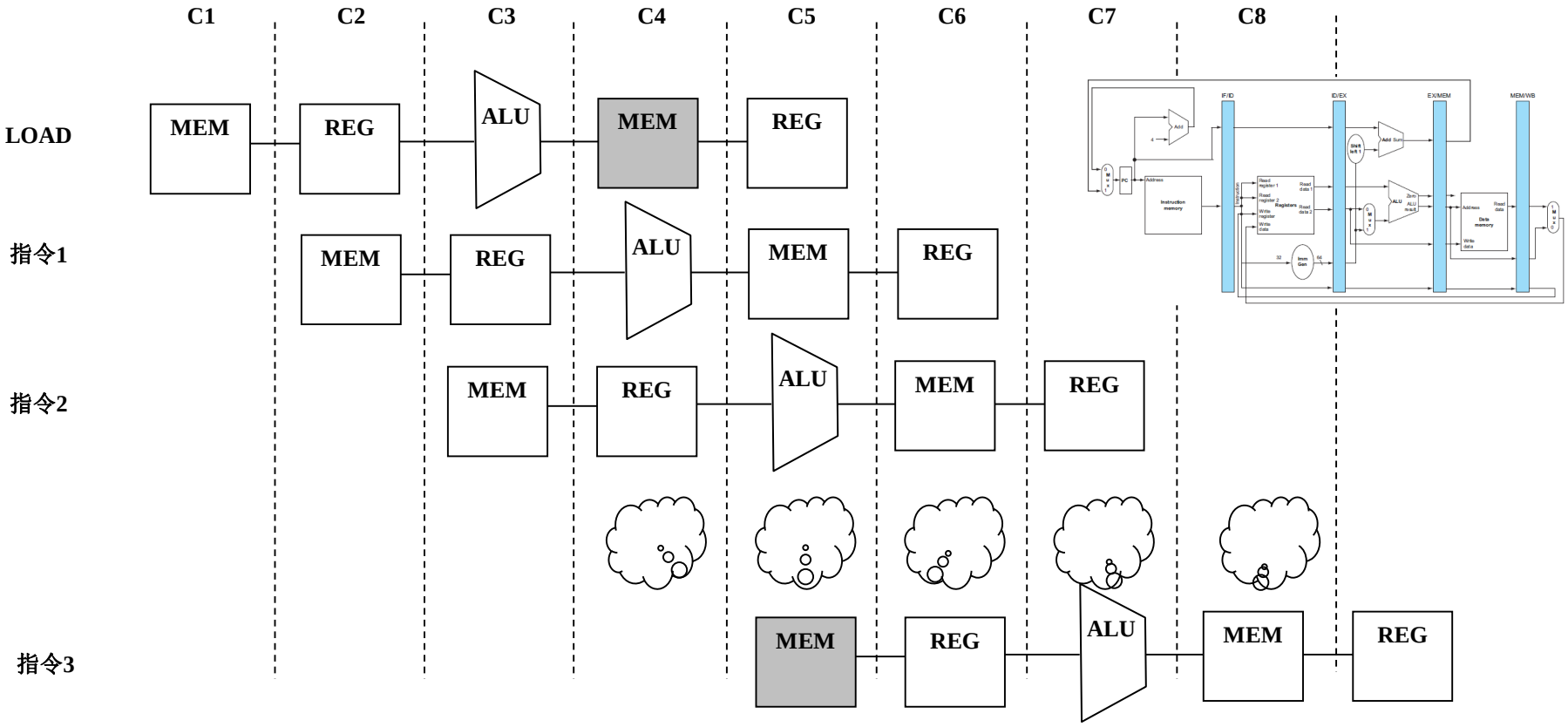
访存结构相关

- 假设：指令和数据都存放在同一个存储器中（合体）
  - 单端口存储器：每次只能执行一个读或一个写
  - 则同一周期（C4）内并发取指与读写数据存在冲突

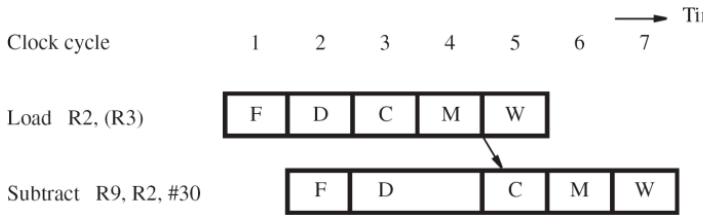




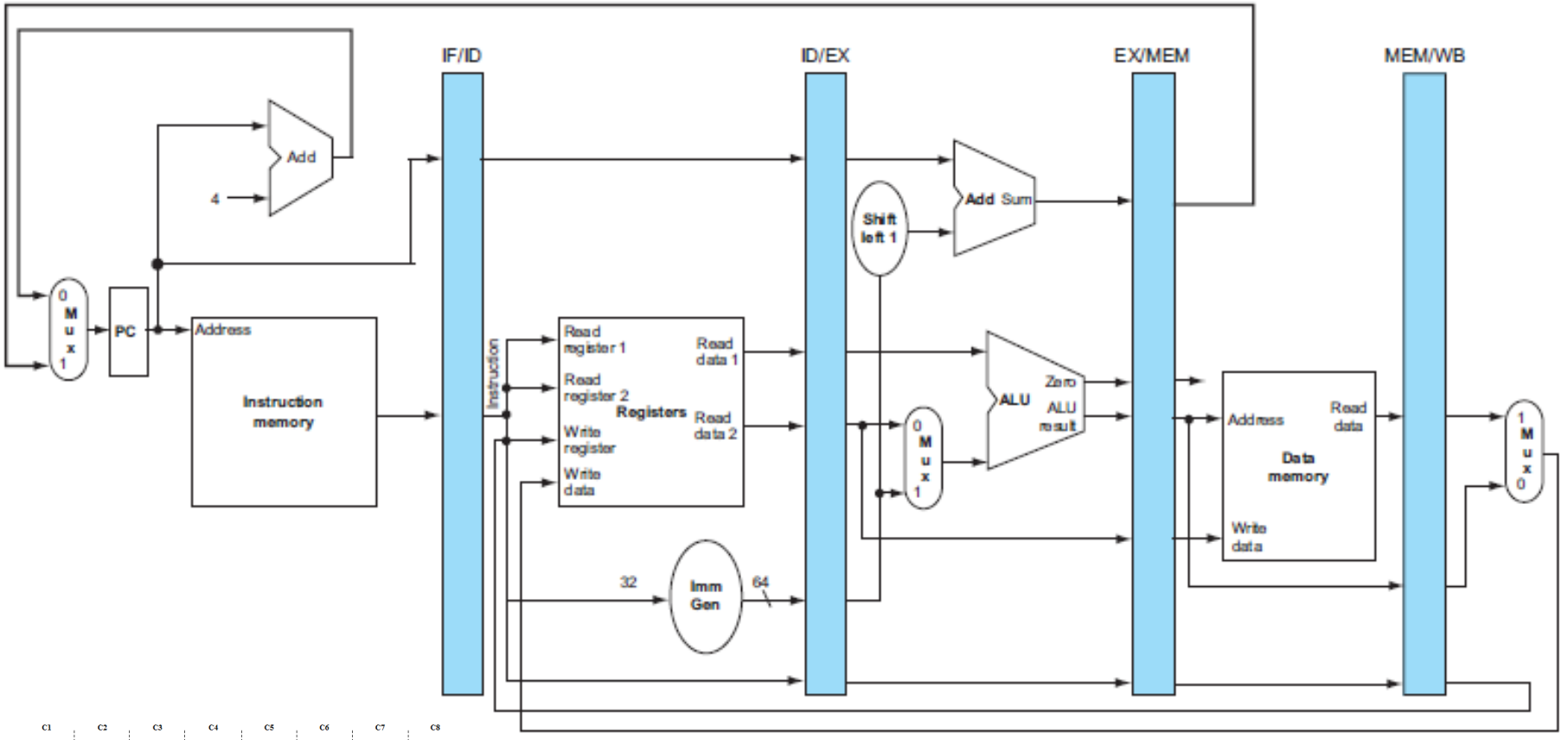
stall流水线的语义（停顿，阻塞）

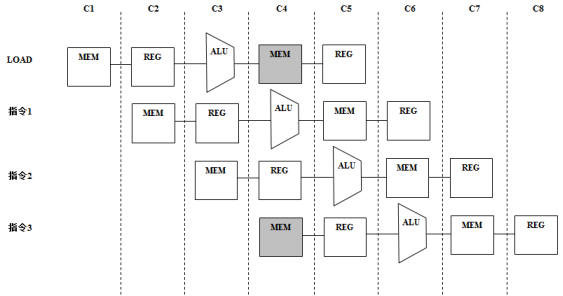


- 程序行为：等价于插入一条nop
- ppl行为：stretch & delay，使流水线停顿一拍
  - “freeze up & bubble down”：interlock
- 访存结构相关：IF段停顿
  - PC不变，重复取指；IF/ID清零 = 插入气泡（bubble）

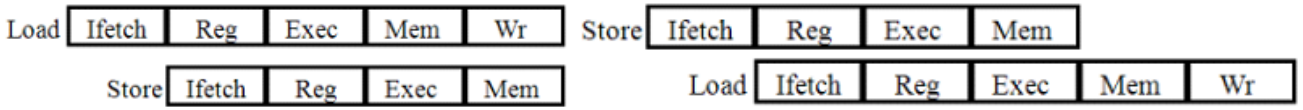


IF与MEM结构冲突消除：哈佛结构



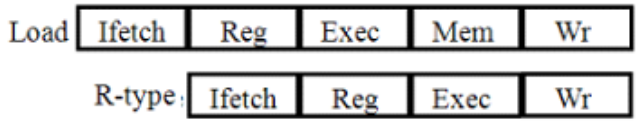


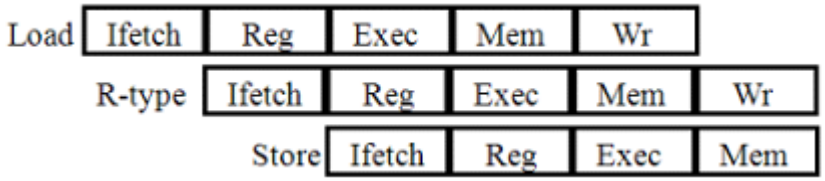
lw/sw是否存在D$冲突？

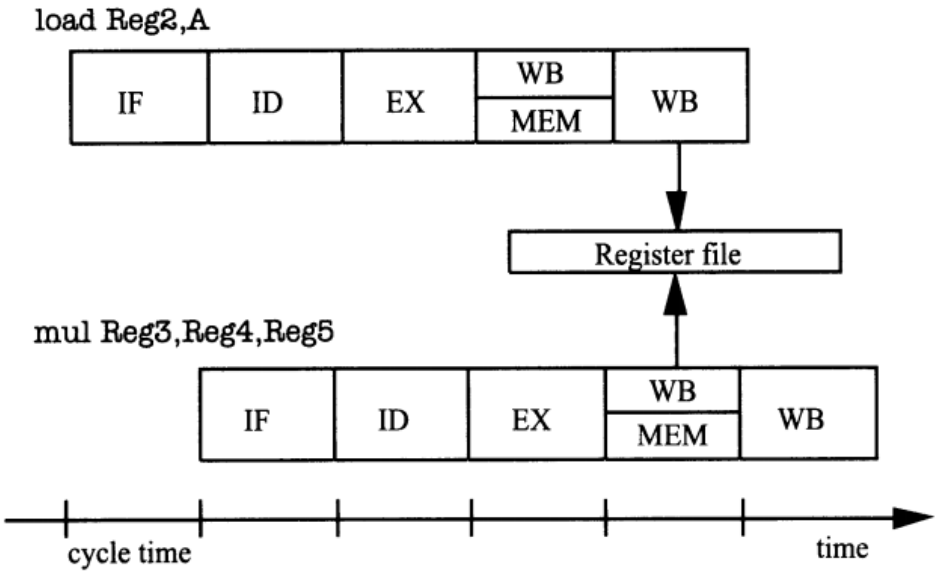


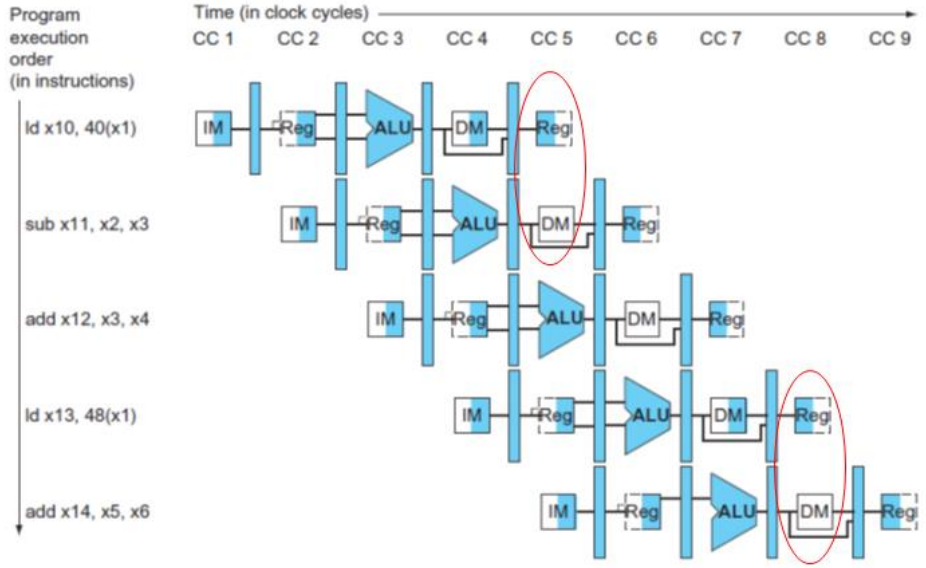
lw-Rtype结构冲突（RF写端口冲突）消除

- **规定**：R-type指令在WB段完成







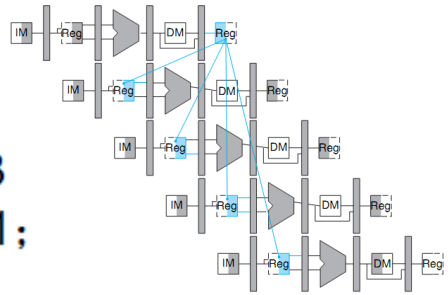


RF“两读一写”，写端口冲突！

数据相关分类

- 写后读（RAW）
  - *j*的执行要用到*i*的计算结果，但*j*可能在*i*写入其计算结果之前就先行对保存该结果的寄存器进行读操作。
  - **类型**：ID段依赖于EXE或MEM段
- 读后写（WAR）：反依赖anti-dependence
  - *j*可能在*i*读取某个源寄存器的内容之前就先对该寄存器进行写操作，导致*i*读出的是错误的数据。
- 写后写（WAW）：输出依赖
  - *j*和*i*的目的操作数一样，写入顺序错误，在目标寄存器中留下的是*i*的值而不是*j*的值。
- 哪些真会发生？
  - ppl：in-order, out-of-order?

```
1   add r0, r2, r1
2   and r2, r0, r3
3   sub r2, r3, #1;
```

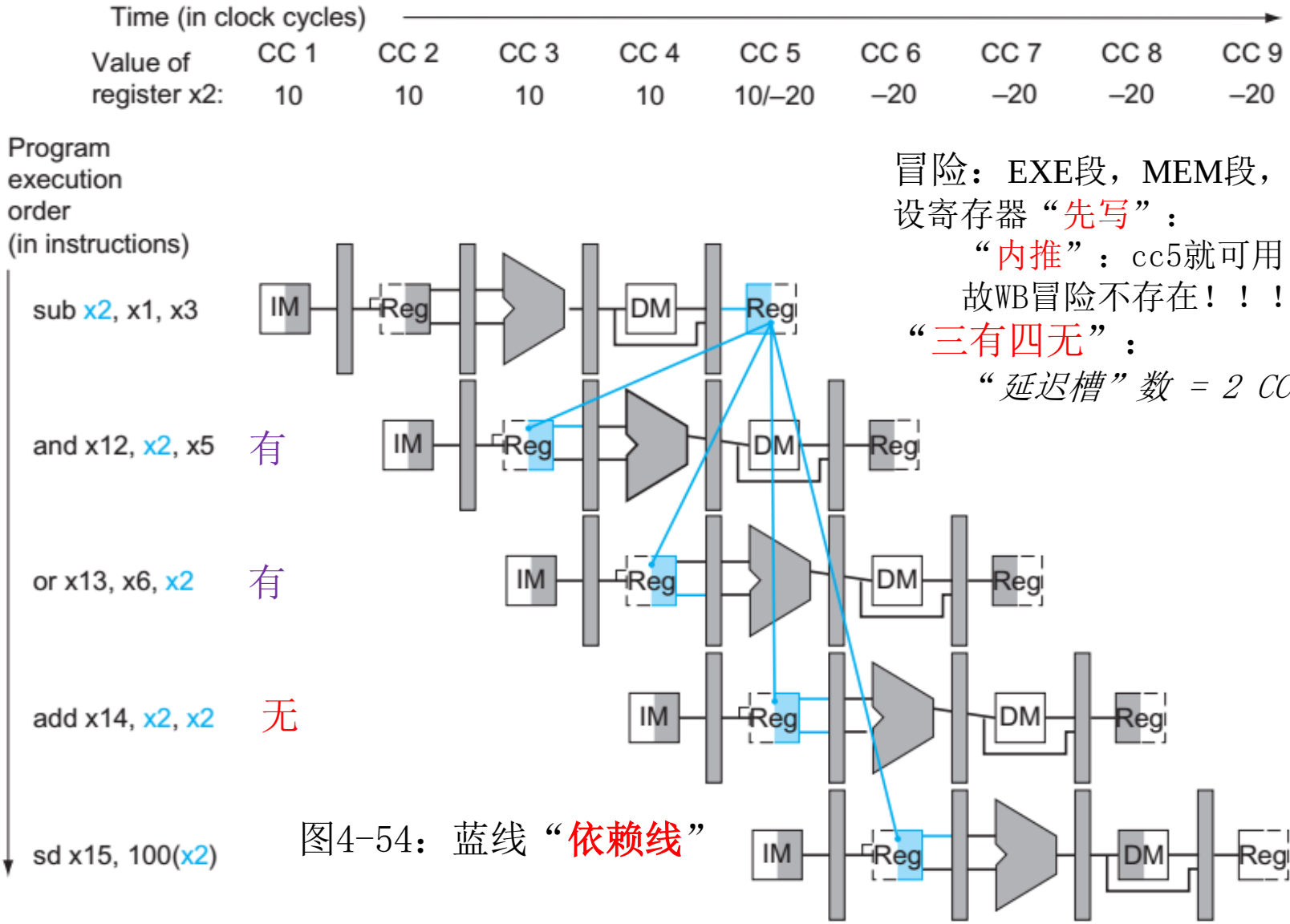


哪些数据依赖会发生？

1：x=2；
2：y=x；
3：y=x+z；
4：z=6；

- 共享资源：Reg依赖，mem依赖
- 程序中存在
  - 真依赖（数据依赖）：RAW，生产者-消费者
  - 假依赖（名字依赖）：WAR/WAW， **RV第二版中文版**p244
    - 后继指令并不使用先前指令的结果值，只是存储位置相同
      - 寄存器不够用：寄存器重命名技术
      - 逻辑寄存器/体系结构寄存器（32个），物理寄存器（128个）
- 物理上：“简单流水线中WAR/WAW不产生冒险”！
  - **RV第二版中文版p247**；llxx“这三点是充要条件”？
    - 按序执行（程序序），且
    - **Reg-Reg指令**仅在最后流水段写回，且
      - “每条指令只写一个结果，且只在WB段写RF”
    - load/store指令在相同的流水段访存

RAW（哪个周期数据可用？）



冒险：EXE段，MEM段，WB段

设寄存器“先写”：

“内推”：cc5就可用！

故WB冒险不存在！！！ p224

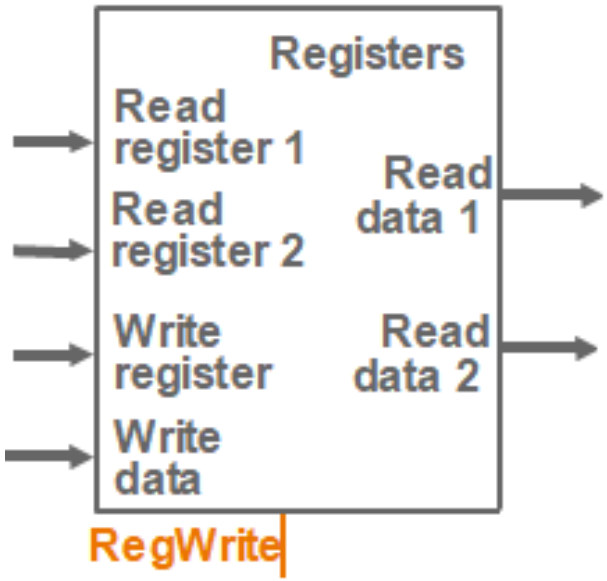
“三有四无”：

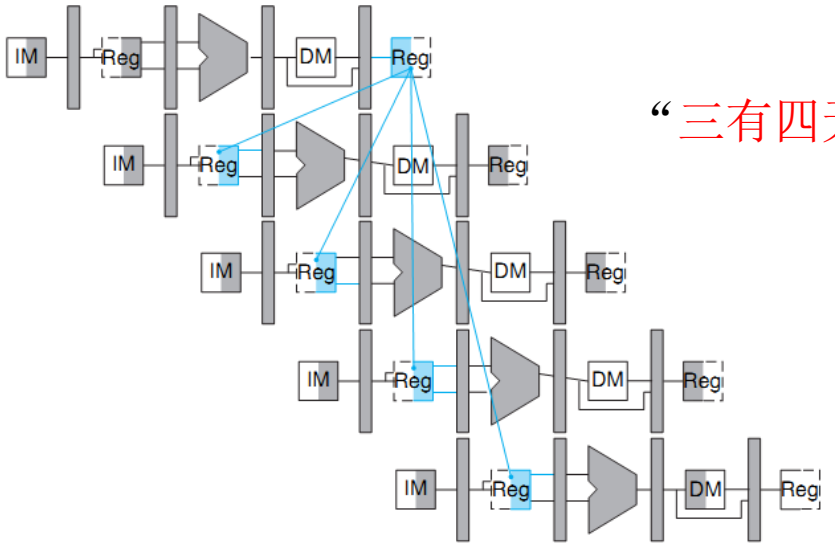
“延迟槽”数 = 2 CC

图4-54：蓝线“依赖线”

RegFile同时读写操作约定

- 不同寄存器读写：在一个周期中，RF支持两读一写
- 同一寄存器读写： `add $t0, $s2, $t0`
  - 不能同时进行读和写：写控制RegWrite与clk同步，两种写方式
    - 后写（late write）：前半周期读，后半周期写。——单周期版？
      - 在一个周期内完成读写操作，但读出的是上一个周期写入的值。图4-7
    - 先写（early write）：前半周期写，后半周期读。——流水线版？
      - 在一个周期内完成读写操作，读出的是当前周期写入的值，如CC5。图4-36，4-54

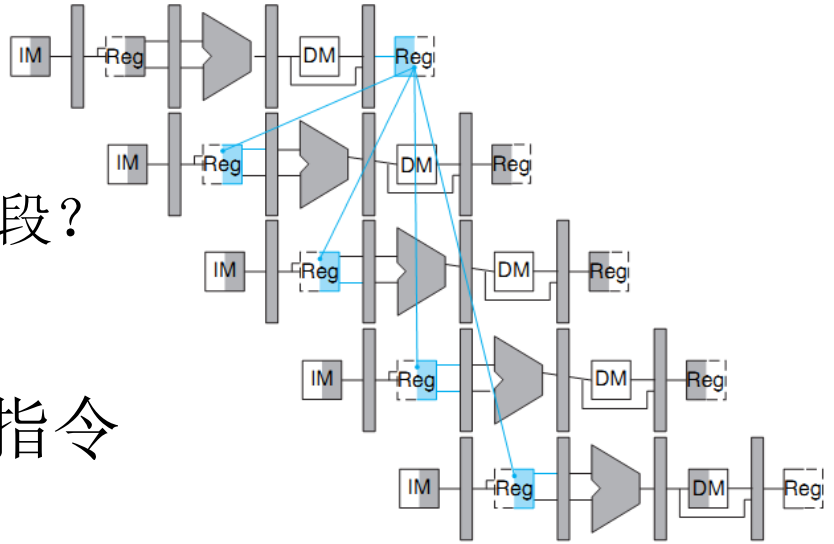




“三有四无”

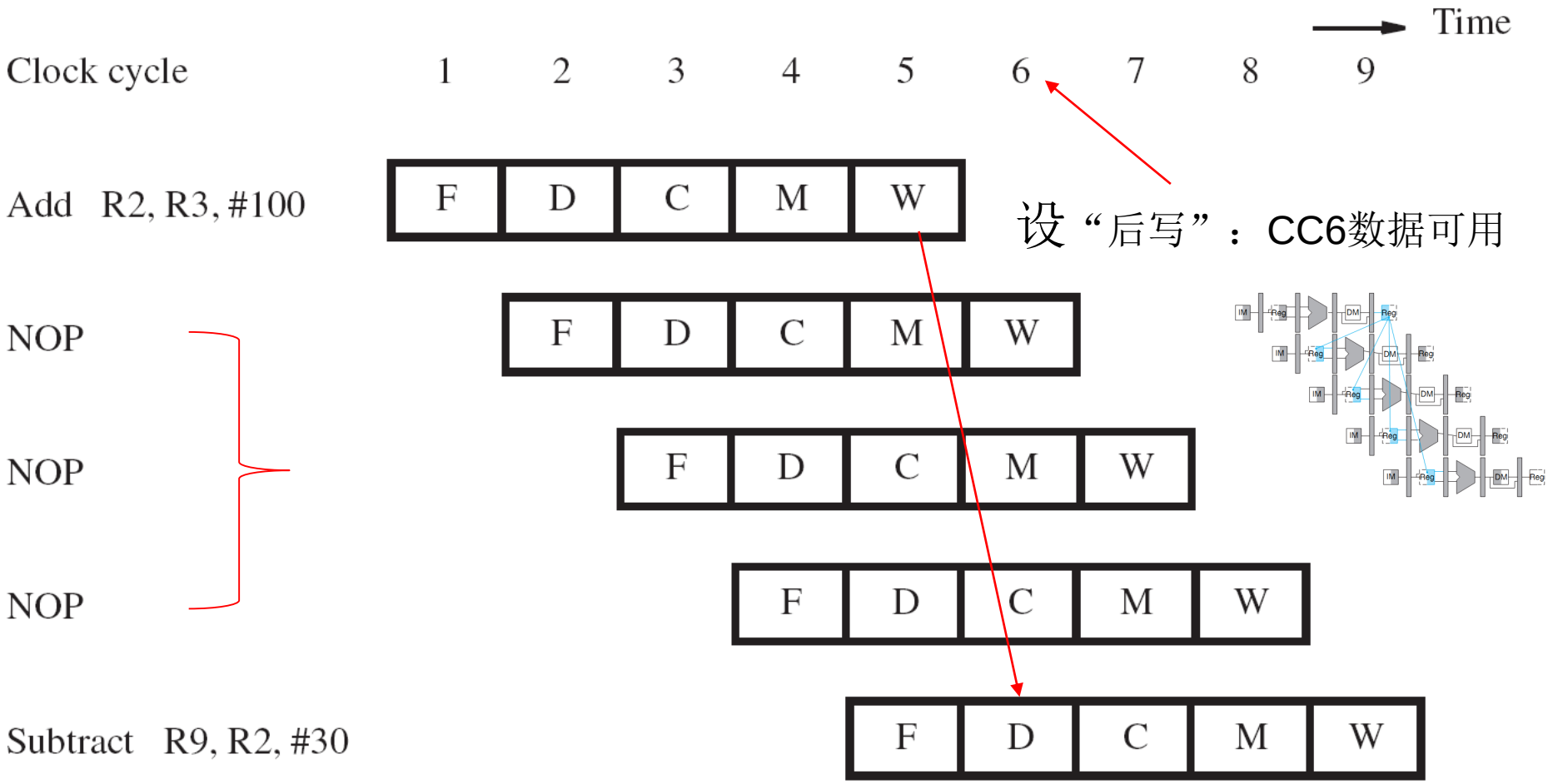
RAW问题：Stall，类型，检测

- Stall实现：可能要插入多个？
  - 编译：延迟槽数“三有四无”
    - 指令调度：插入nop，静态技术
    - 兼容性问题
  - uArch：Interlock
    - freeze up & bubble down：哪一段？
- 减少stall损失
  - 指令调度：编译器填充无关指令
  - 数据定向：动态技术
    - 数据前推（Forwarding）/旁路（Bypassing）
    - 使关键路径加长，不利于提升主频！



化解RAW：编译器, $4.6.2

- 指令调度（注意：此例RF为“后写”——“4有5无”）



减少RAW损失：Forwarding to EX

- “bypassing the use of RF”

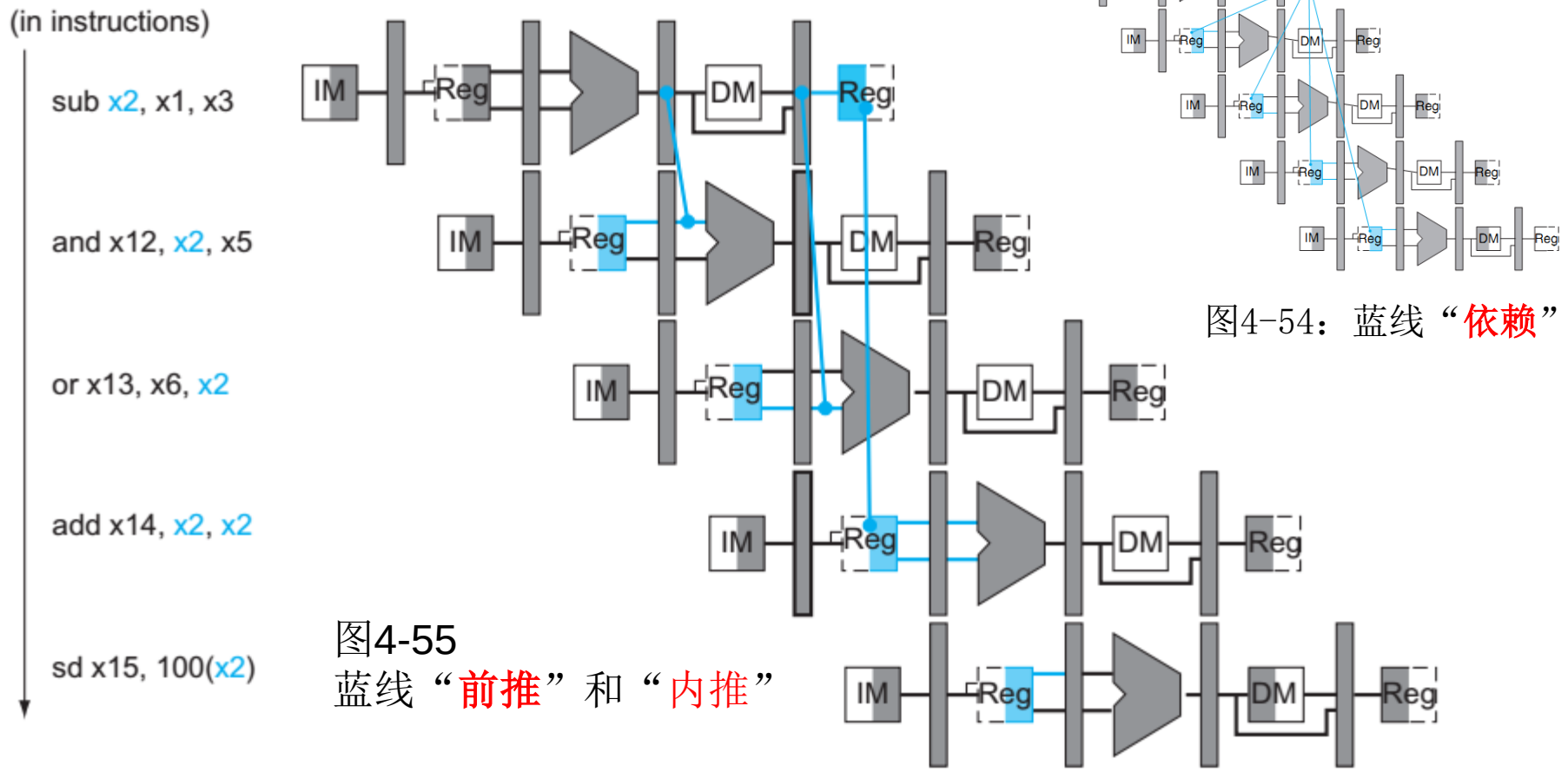


图4-54：蓝线“依赖”

图4-55
蓝线“前推”和“内推”

EX/MEM、MEM/WB、WB(RF先写，内推，见p220、p224)

EX段旁路：向EX段前推，检测规则？$4.8

- Rtype-Rtype：须将EX/MEM、MEM/WB向EX段前推
  - FWU功能：检测，前推控制（MUX）

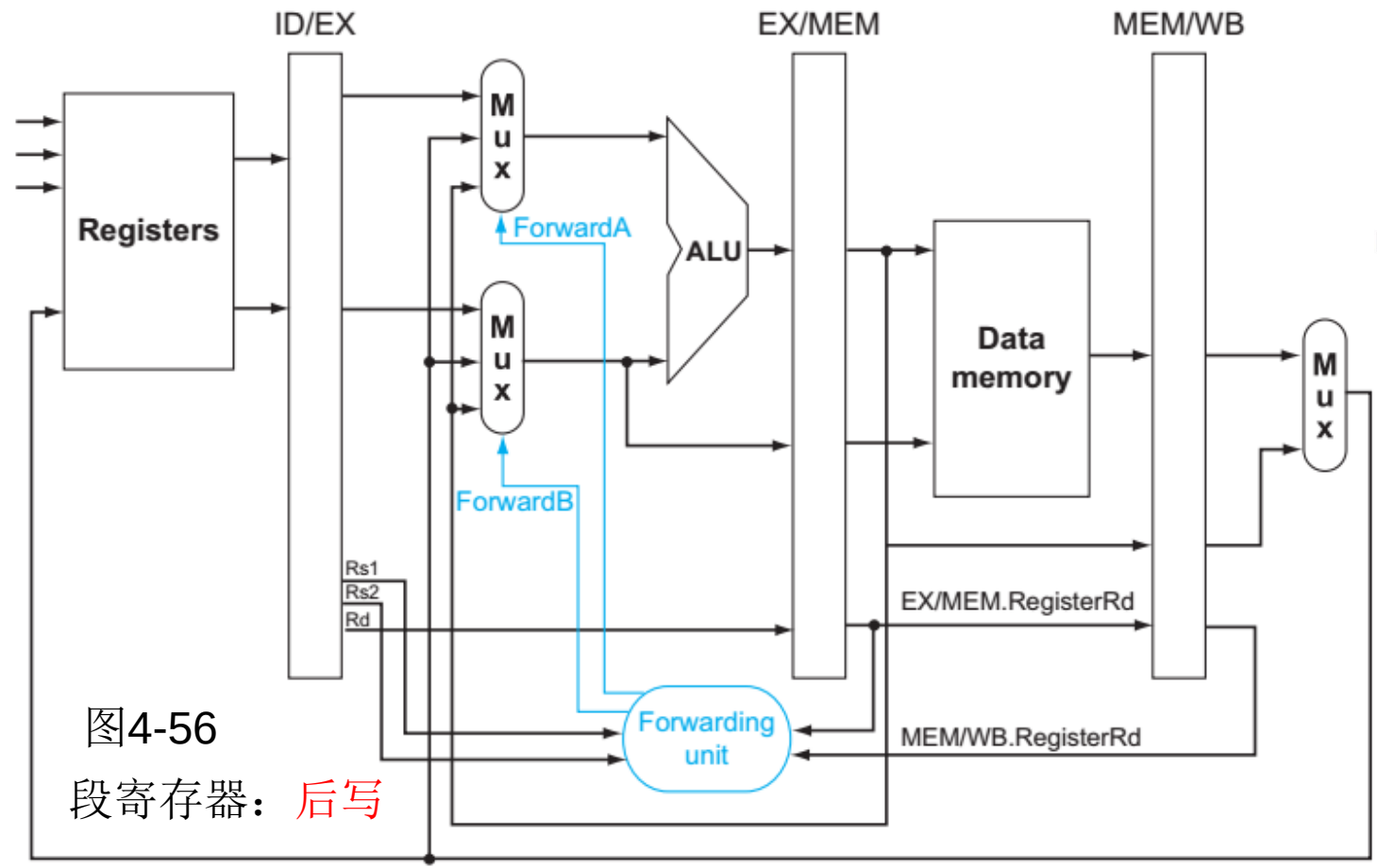
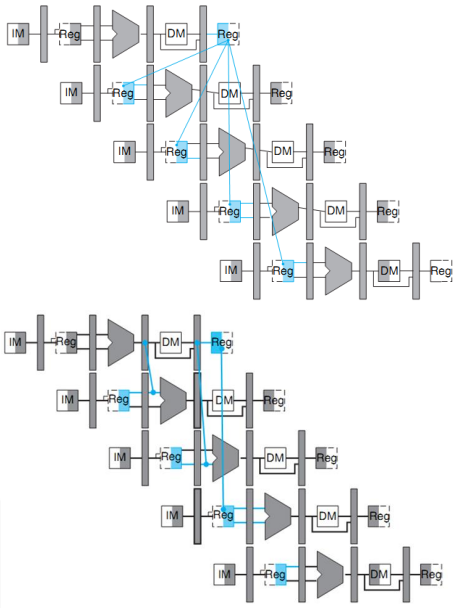


图4-56

段寄存器：后写



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

EX段旁路检测规则（1/3）：“三有四无”，$4.8

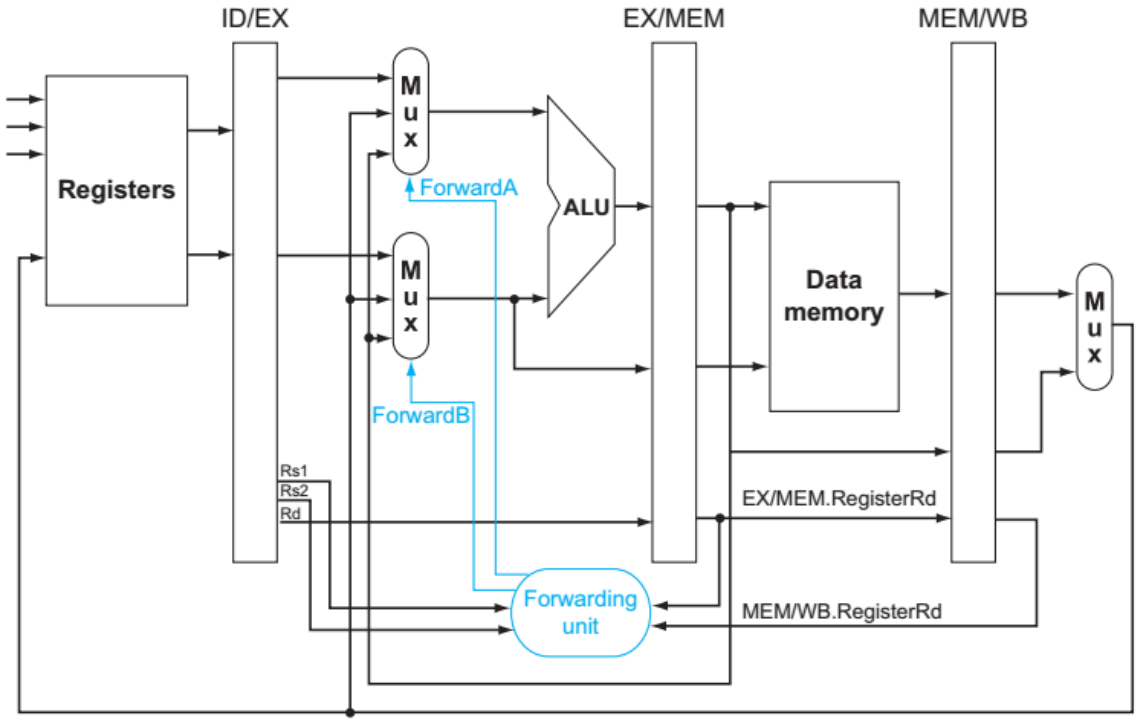
- EX/MEM前推
  - 指令二依赖指令一
- MEM/WB前推
  - 指令三依赖指令一

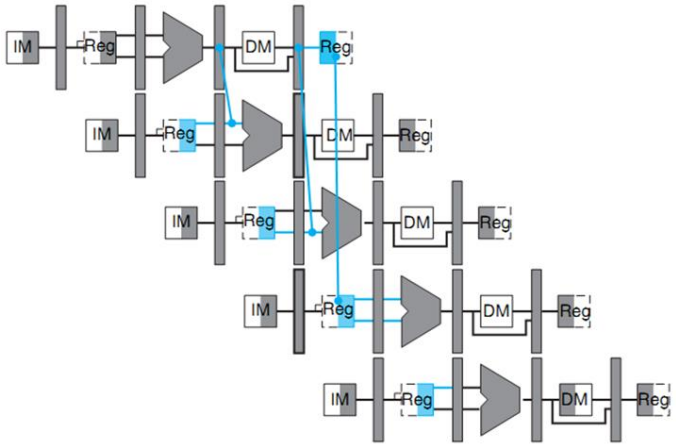
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1

1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

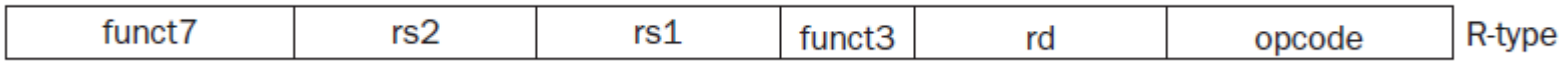
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1

2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2





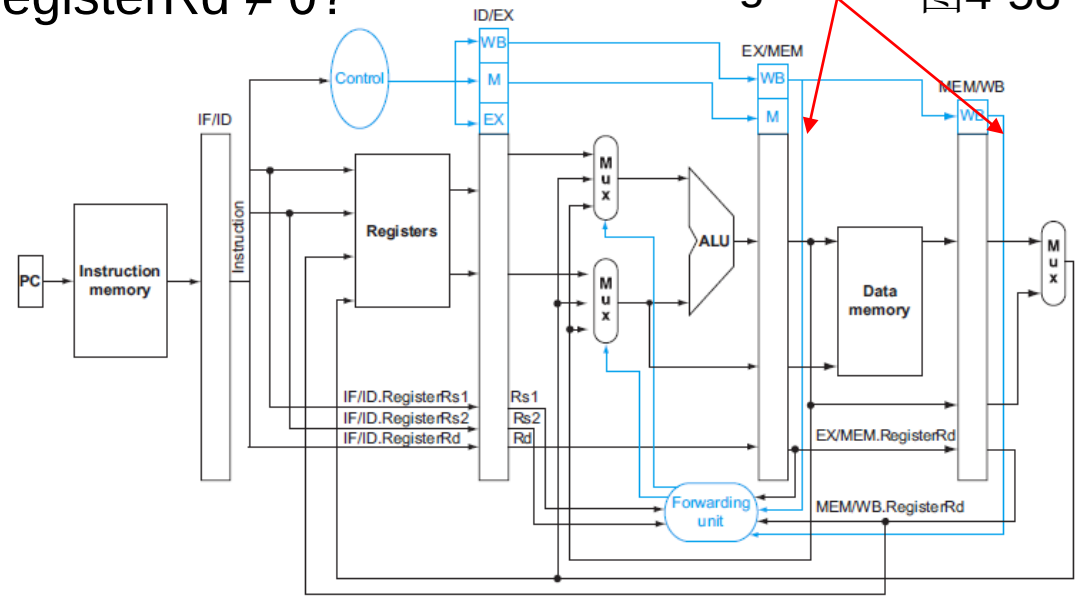

```
sub  x2, x1, x3
and  x12,  x2, x5
or   x13,  x6, x2
add  x14,  x2, x2
sd   x15,  100(x2)
```



EX段旁路检测规则（2/3）： p221/p224，bug？

- 无须前推
  - 非写回指令：RegWrite?——示例？
  - 目标寄存器为$zero：RegisterRd ≠ 0?

RegWrite 图4-58



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

EX段RAW检测规则：指令2

```
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
```

Forwarding控制器

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

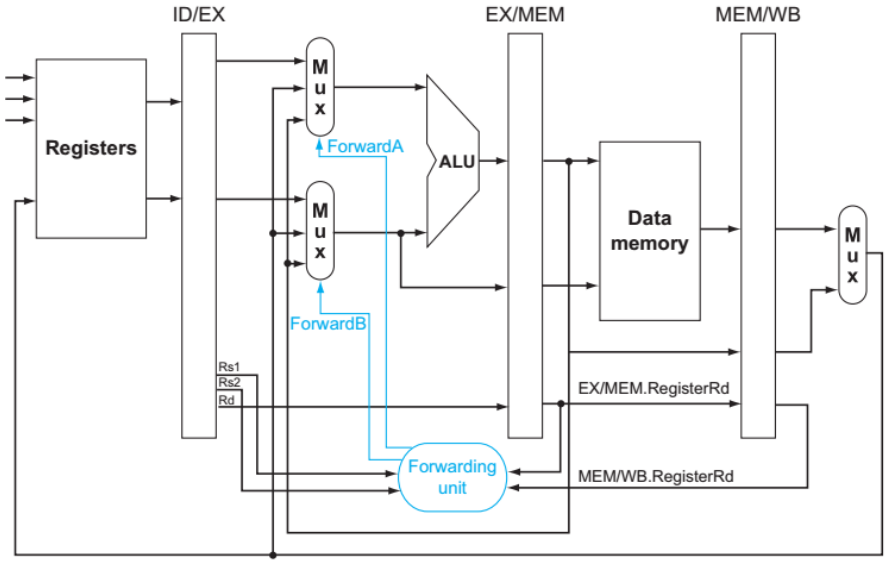


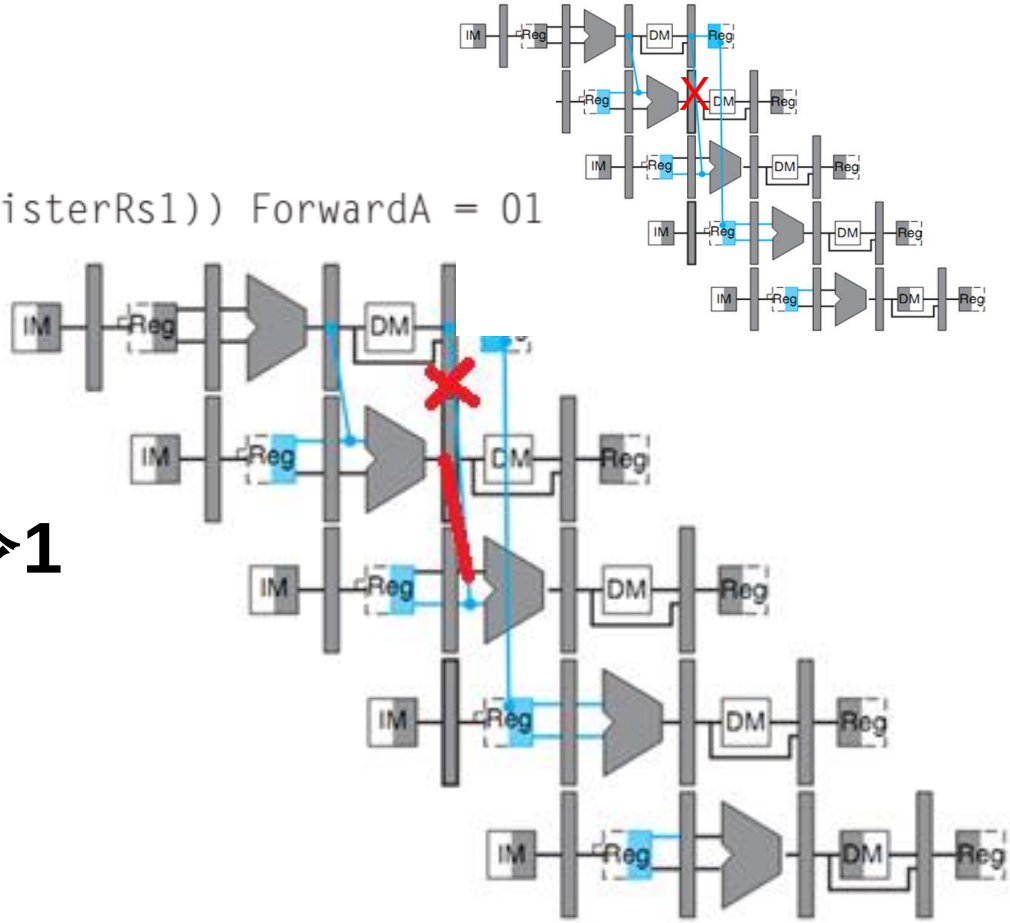
图4-57

```
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


if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

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

EX段旁路检测规则（3/3）：求和，p224

指令3 依赖 指令1，有

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
 sub x2, x1, x3
 and x12, x2, x5
 or  x13, x6, x2
 add x14, x2, x2
 sd  x15, 100(x2)
```



求和：指令3 不依赖 指令1

```
  add x1, x1, x2
  add x1, x1, x3
  add x1, x1, x4
  . . .
```

故应：

```
if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 0)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
         and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
```

```
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
```

EX hazards：指令2依赖指令1

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 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 ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

EX hazards：指令3依赖指令1，
不依赖指令2

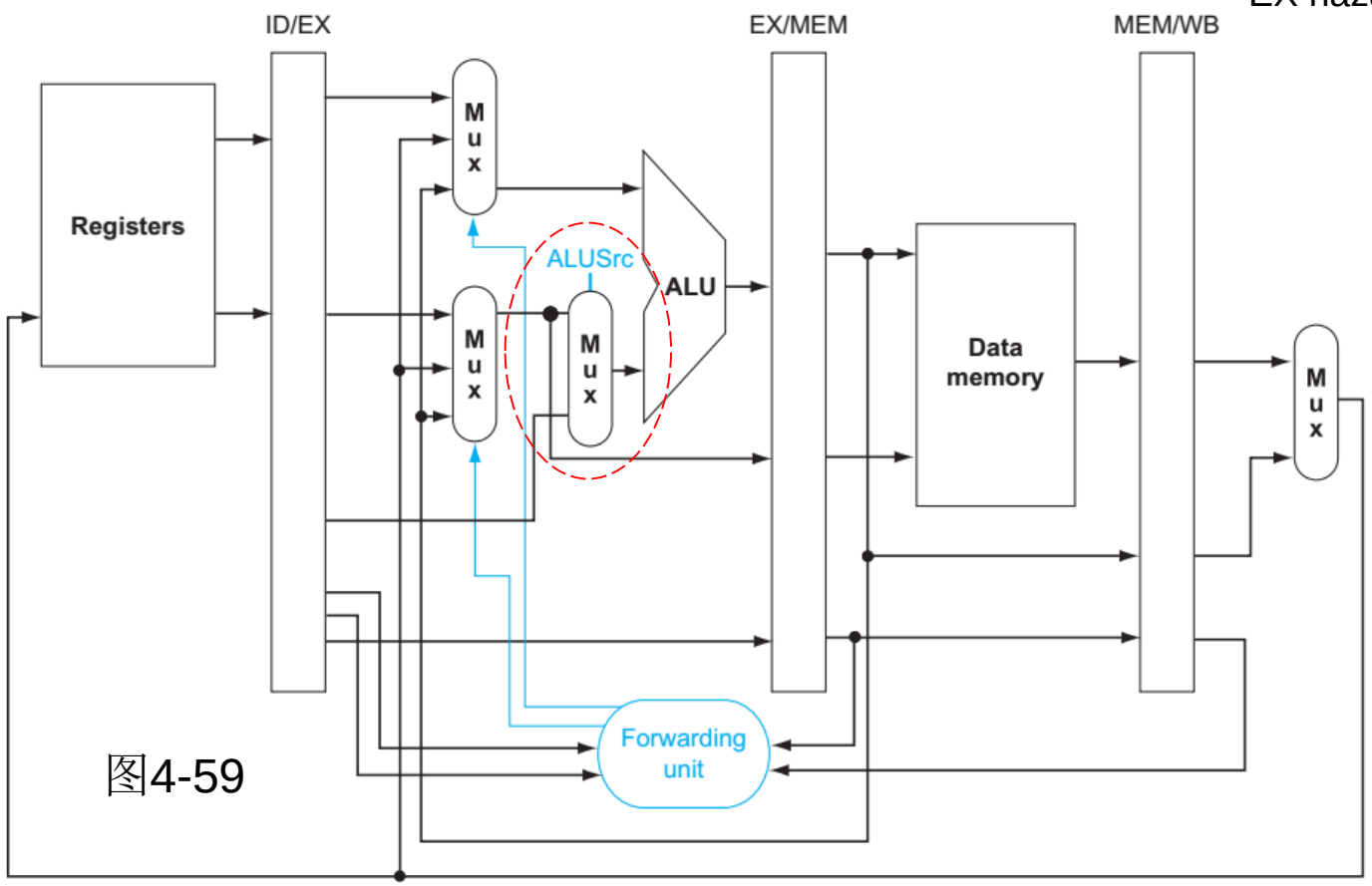
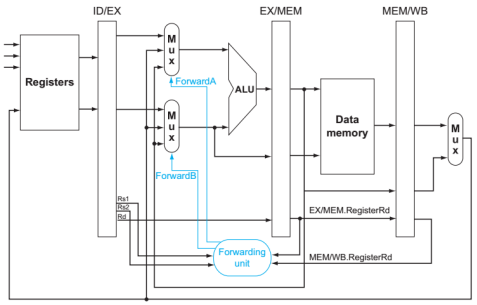
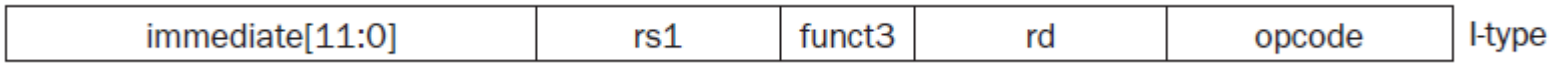


图4-59



注意：I-type ALU
MUX，立即数
（图4-56无）



EX段冒险：完整的数据通路与控制

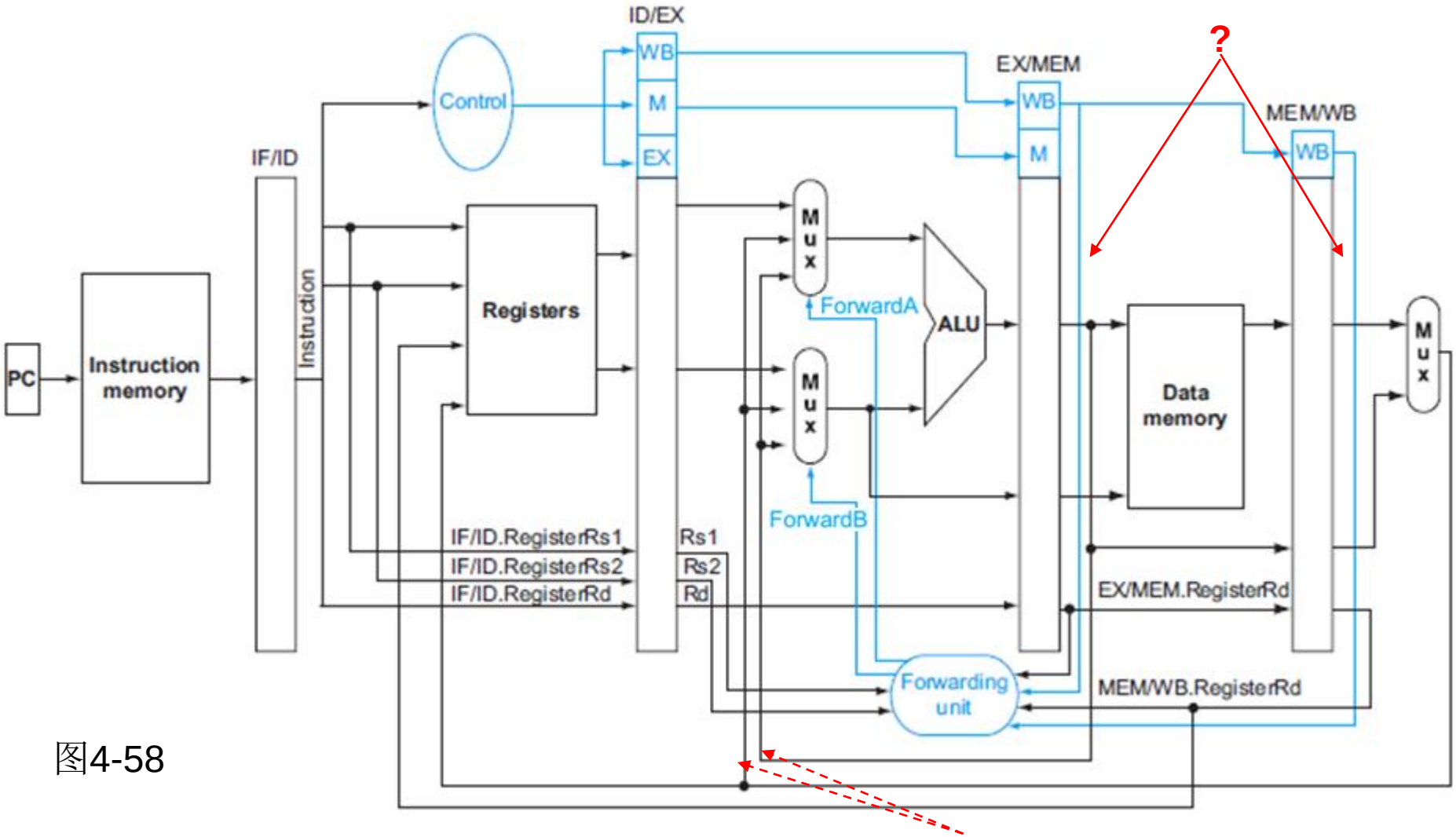


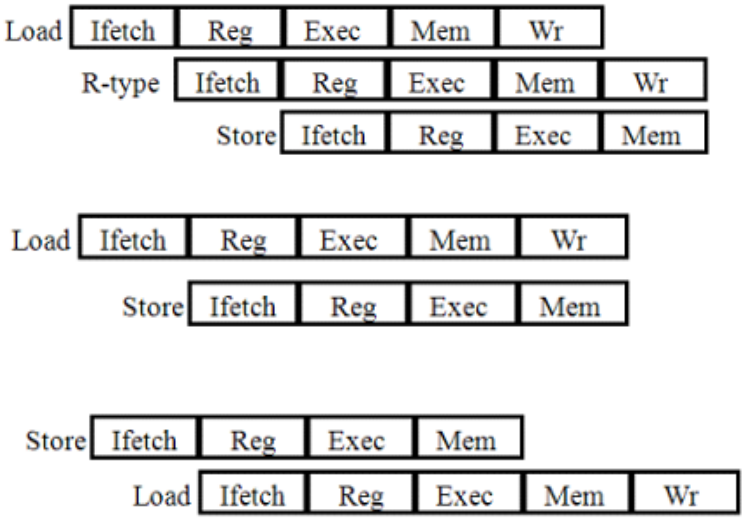
图4-58

将EX/MEM（ALU结果）和MEM/WB（ALU/lw结果）前推至EX段（ALU输入）

访存指令的RAW问题：MEM段冒险

- 设计假设：in-order ppl，无store-buf

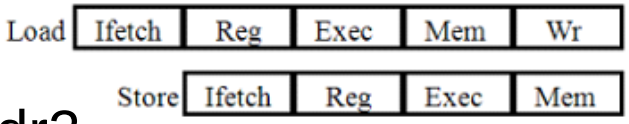
- 寄存器依赖：EX段冒险
  - load-used：load-Rtype，——注意：不是“写端口的结构冲突”！
  - *间接寻址*：100[x1]中为指针，x2写入该指针指向之内存单元
    - lw x3, 100[x1]
    - sw x2, 200[x3]
- 存储器依赖：MM段冒险
  - Copy：不同地址
    - lw x2, 100(x1)
    - sw x2, 110(x1)
  - 校验：同一地址
    - sw x2, 100[x1]
    - lw x2, 100[x1]
    - xor x1, x2, x2



- COD p225精解：“loads immediately followed by stores”，译错？

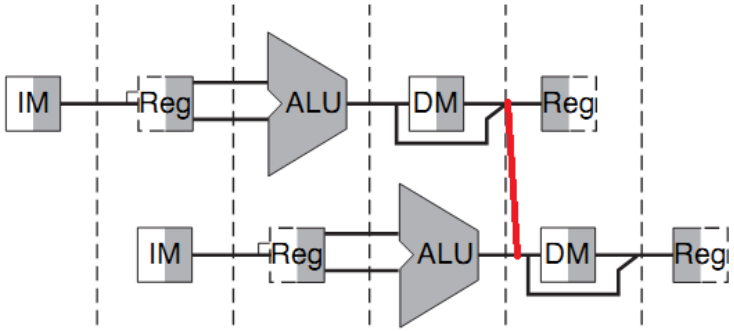
MEM段冒险：MEM copy问题

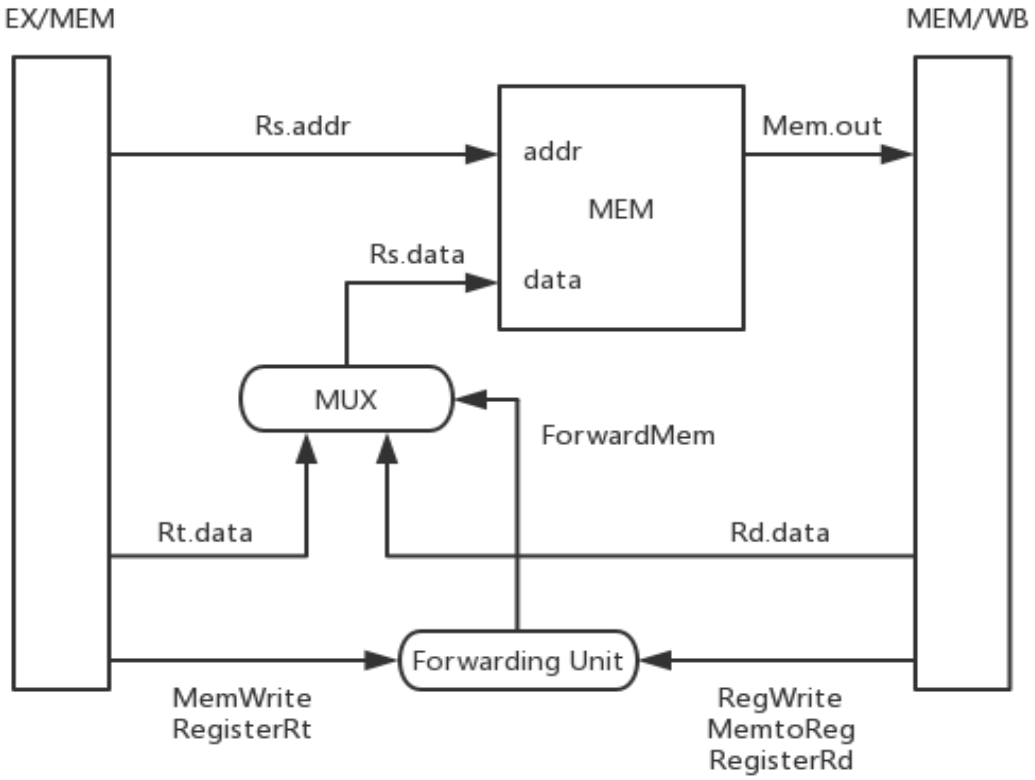
- “RISC中，mem-to-mem拷贝频繁”
  - 由lw-sw实现：lw $1, addr1；sw $1, addr2



- MEM段旁路：MEM/WB => MEM，无需插入bubble！
  - 判据，数据通路？

```
lw x2, 100(x1)
sw x2, 110(x1)
or  x13, x6 x2
add x14, x2, x2
sw x15, 100(x2)
```





lw-sw前推？

```
if  (EX/MEM.MemWrite and MEM/WB.MemtoReg
  and  (EX/MEM.RegisterRt \= 0)
  and  (MEM/WB.RegisterRd = EX/MEM.RegisterRt) ) ForwardMEM = 1
```

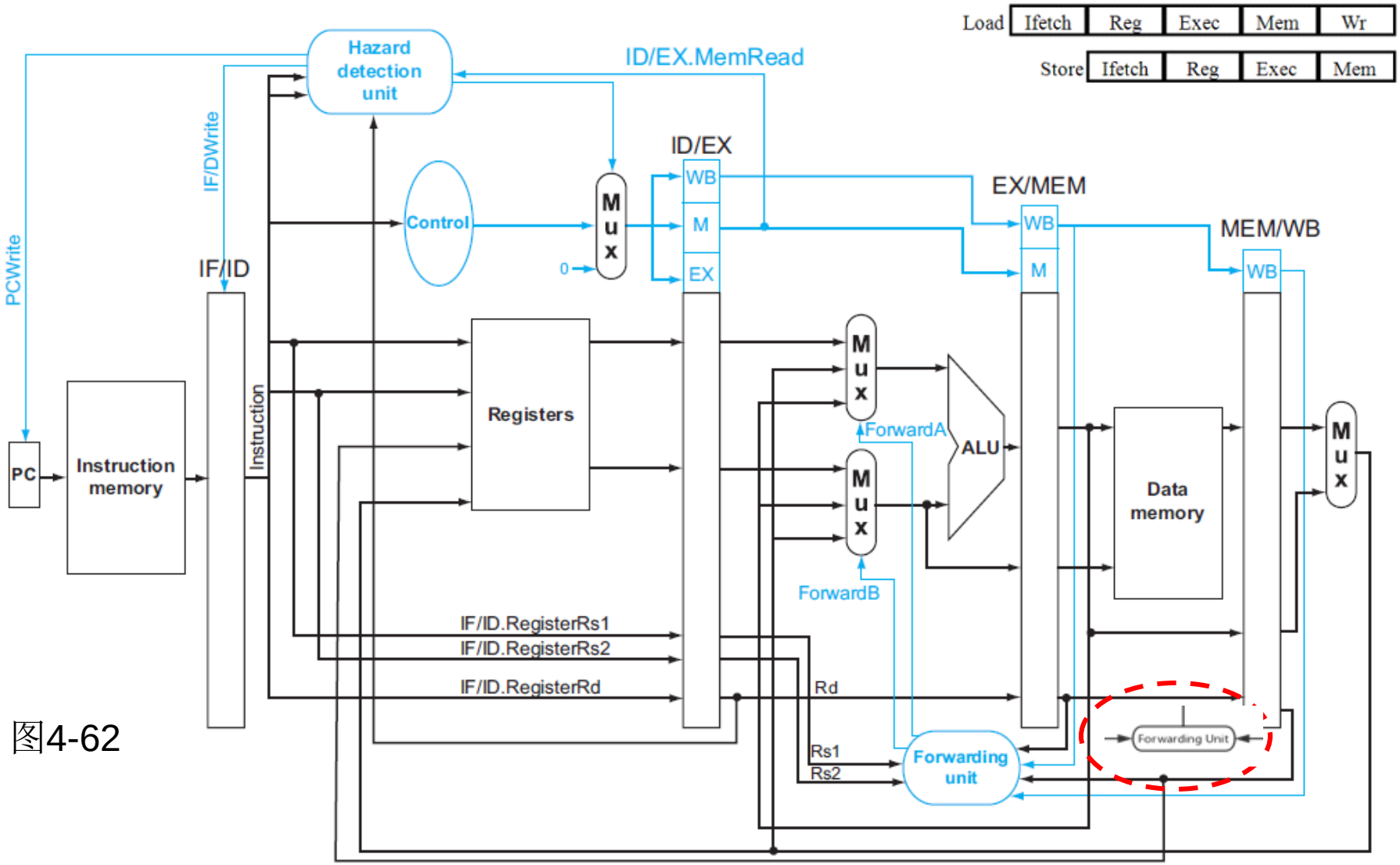
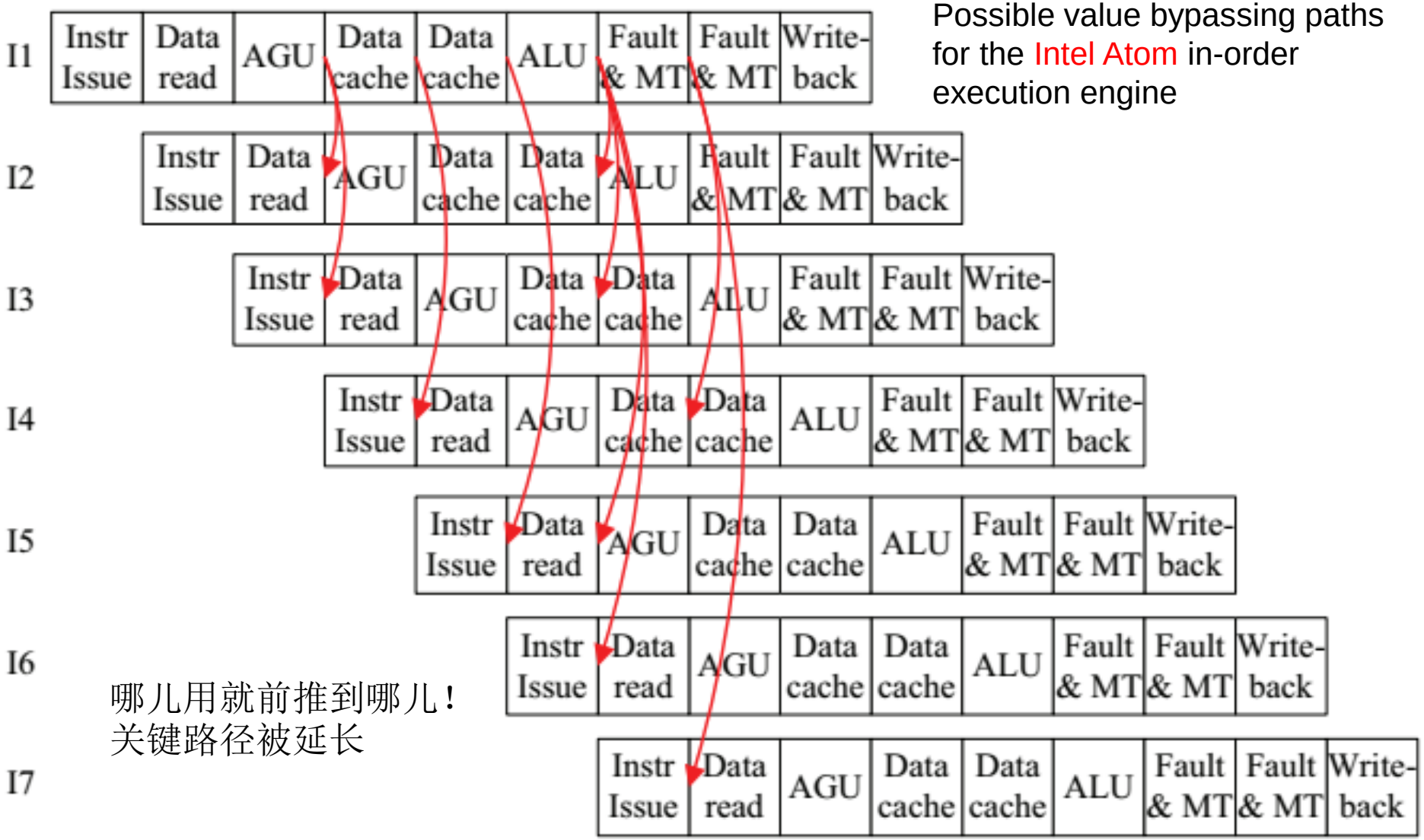


图4-62

Multilevel Bypass：需要多个段前推！



Possible value bypassing paths for the Intel Atom in-order execution engine

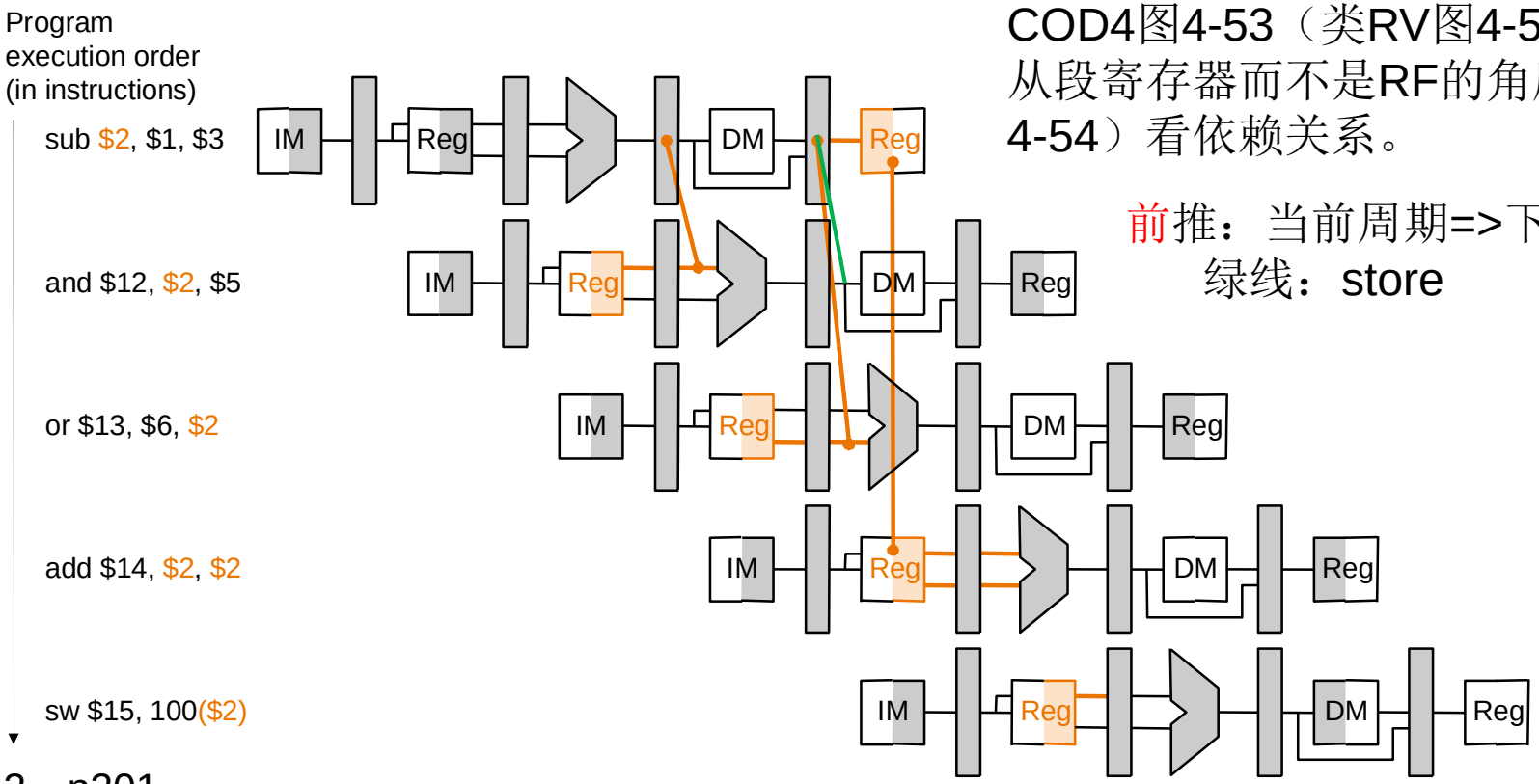
哪儿用就前推到哪儿！
关键路径被延长

Antonio Gonzalez，Processor Microarchitecture_ An Implementation Perspective -Morgan & Claypool Publishers, 2010

forwarding可消解的RAW

Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X



COD4图4-53（类RV图4-55），从段寄存器而不是RF的角度（图4-54）看依赖关系。

前推：当前周期=>下一周期

绿线：store

$4.6.2，p201

RISC ISA的第四个特征：“每条指令仅在最后一个流水段写一个结果”，利于forwarding！

forwarding无法解决的RAW：backwards

例：load-use-data hazard（亦称load-used hazard）

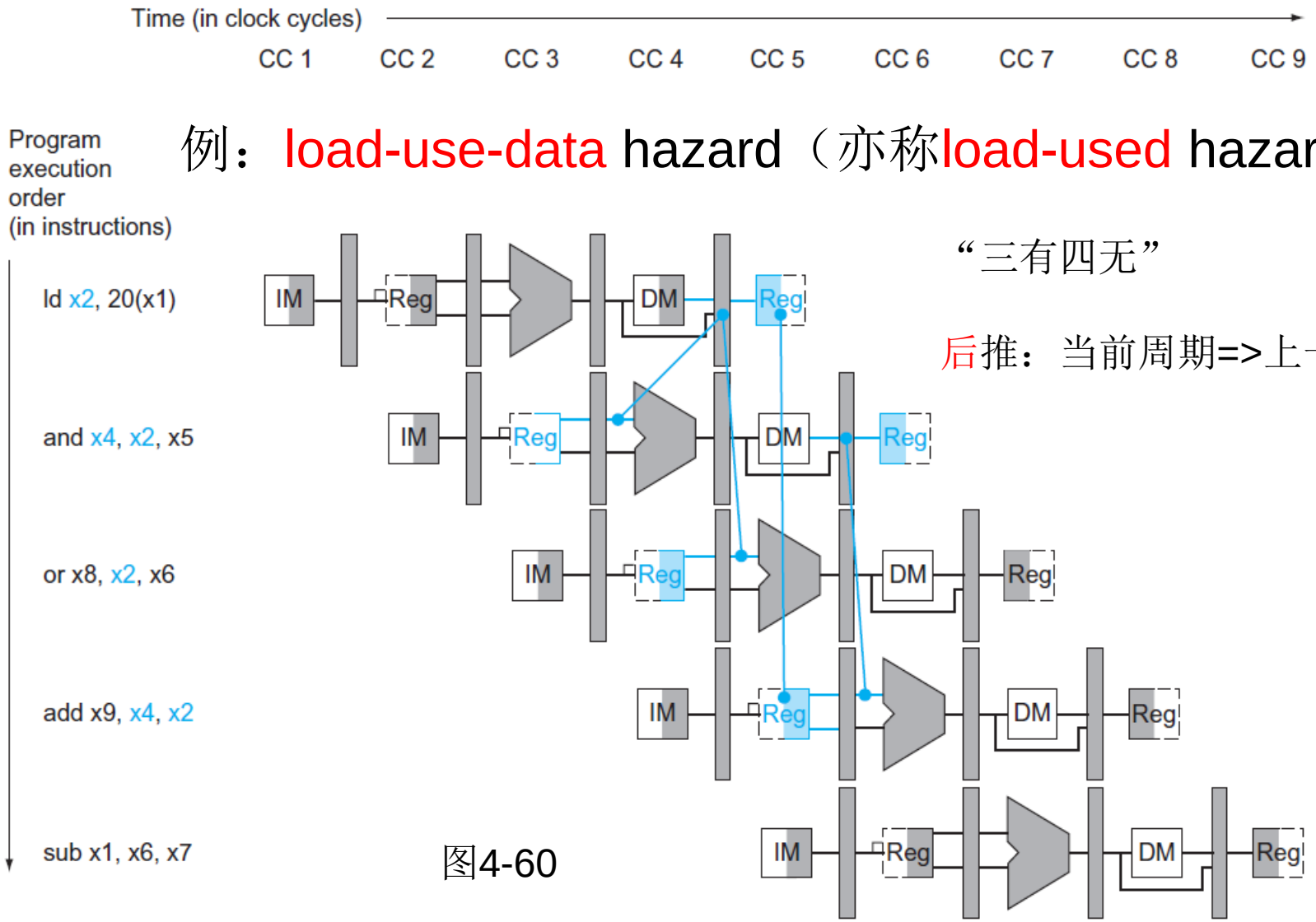


图4-60

“三有四无”

后推：当前周期=>上一周期

load-use-data hazard，$4.6.2

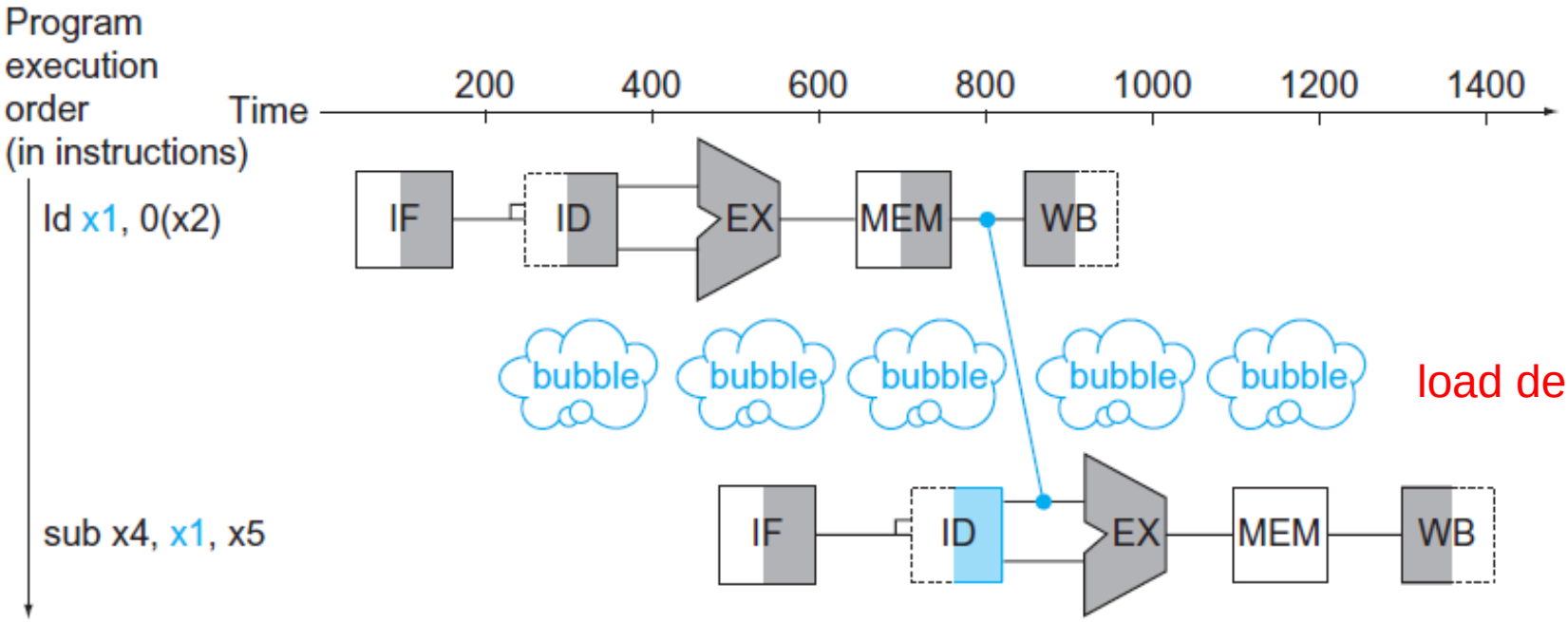
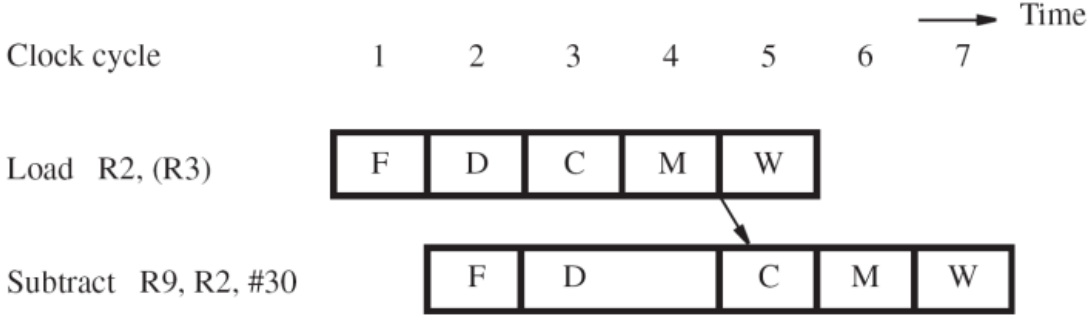


图4-32

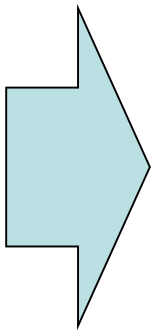
- 编译法：插入nop（ = sll $0, $0, 0 ），或指令重排序
- 硬件法：**Interlocking**
  - stall：**stretch & delay**
  - FW：MEM/WB->EX
  - 判据？



Code Scheduling：编译，$4.6.2

- 指令重排序：程序序≠执行序
  - RISC-V code for a=b+e; c=b+f;
  - “快2个cycle”？——假设有前推，时空图？
- 不是所有指令顺序都能调整（结合律）！

```
ld      x1, 0(x31)   // Load b
ld      x2, 8(x31)   // Load e
add     x3, x1, x2   // b + e
sd      x3, 24(x31)  // Store a
ld      x4, 16(x31)  // Load f
add     x5, x1, x4   // b + f
sd      x5, 32(x31)  // Store c
```



```
ld    x1, 0(x31)
ld    x2, 8(x31)
ld    x4, 16(x31)
add   x3, x1, x2
sd    x3, 24(x31)
add   x5, x1, x4
sd    x5, 32(x31)
```

RAW Stall：Interlocking，$4.8

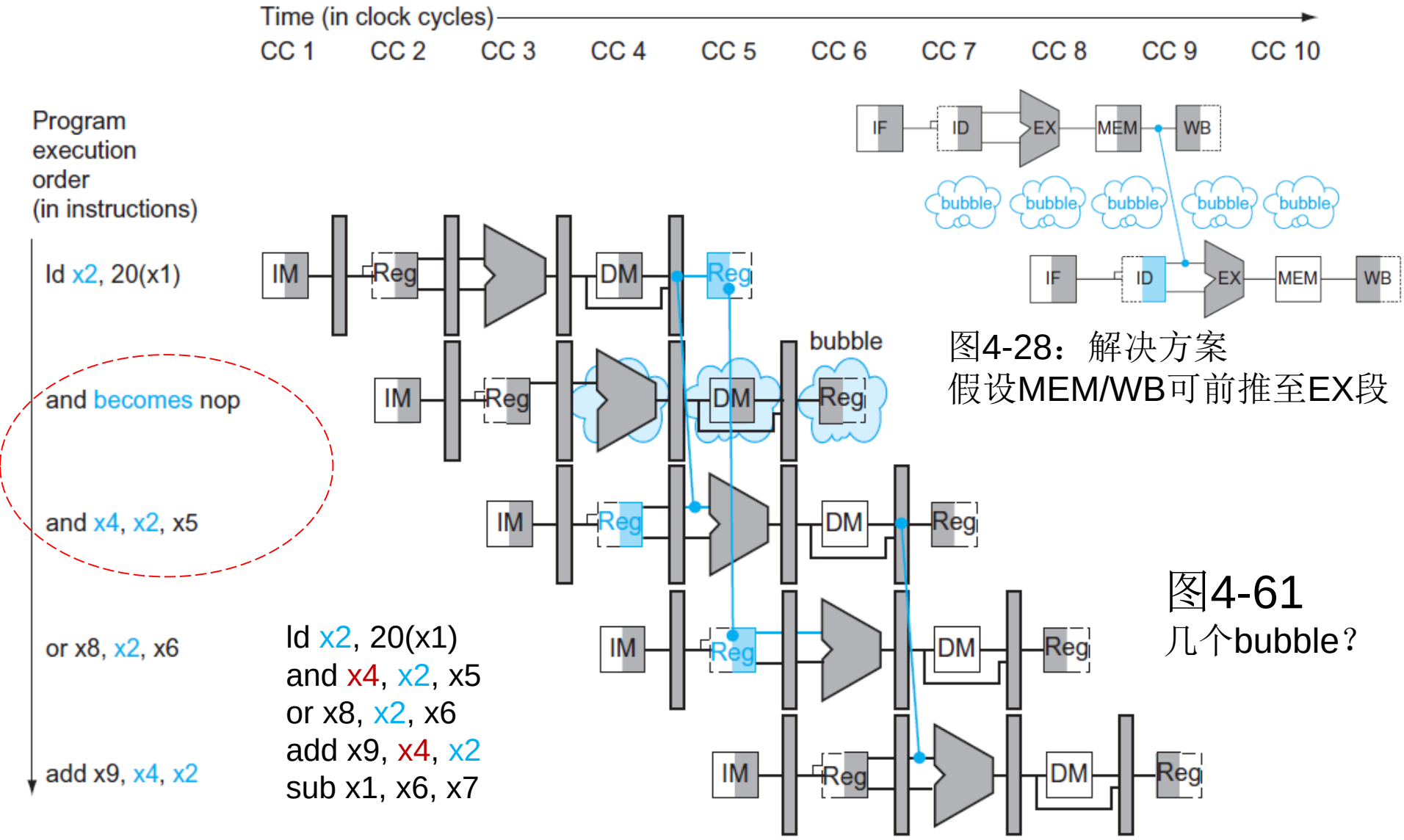


图4-28：解决方案
假设MEM/WB可前推至EX段

图4-61
几个bubble？

Interlock：load-use-data stalling

- Stall语义：freeze up and bubble down
  - We can stall the pipeline by keeping an instruction in the same stage
- load-use-data stalling
  - 冻结EX前面的所有流水段IF和ID，使之重复操作
    - 增加两个控制信号：PCWrite和IF/IDWrite
    - 使PC和IF/ID的时钟无效：阻止更新PC和IF/ID
  - 向其后的流水段插入一个气泡，并向下传递
    - ID/EX清零，且“let them percolate through the pipeline”
- HDU(hazard detection unit)：检测
  - load-use-data stalling规则？

ID段interlock

```
if  (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
      (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
      stall the pipeline
```

ld x2, 20(x1)
and x4, x2, x5
or x8, x2, x6
add x9, x4, x2
sub x1, x6, x7

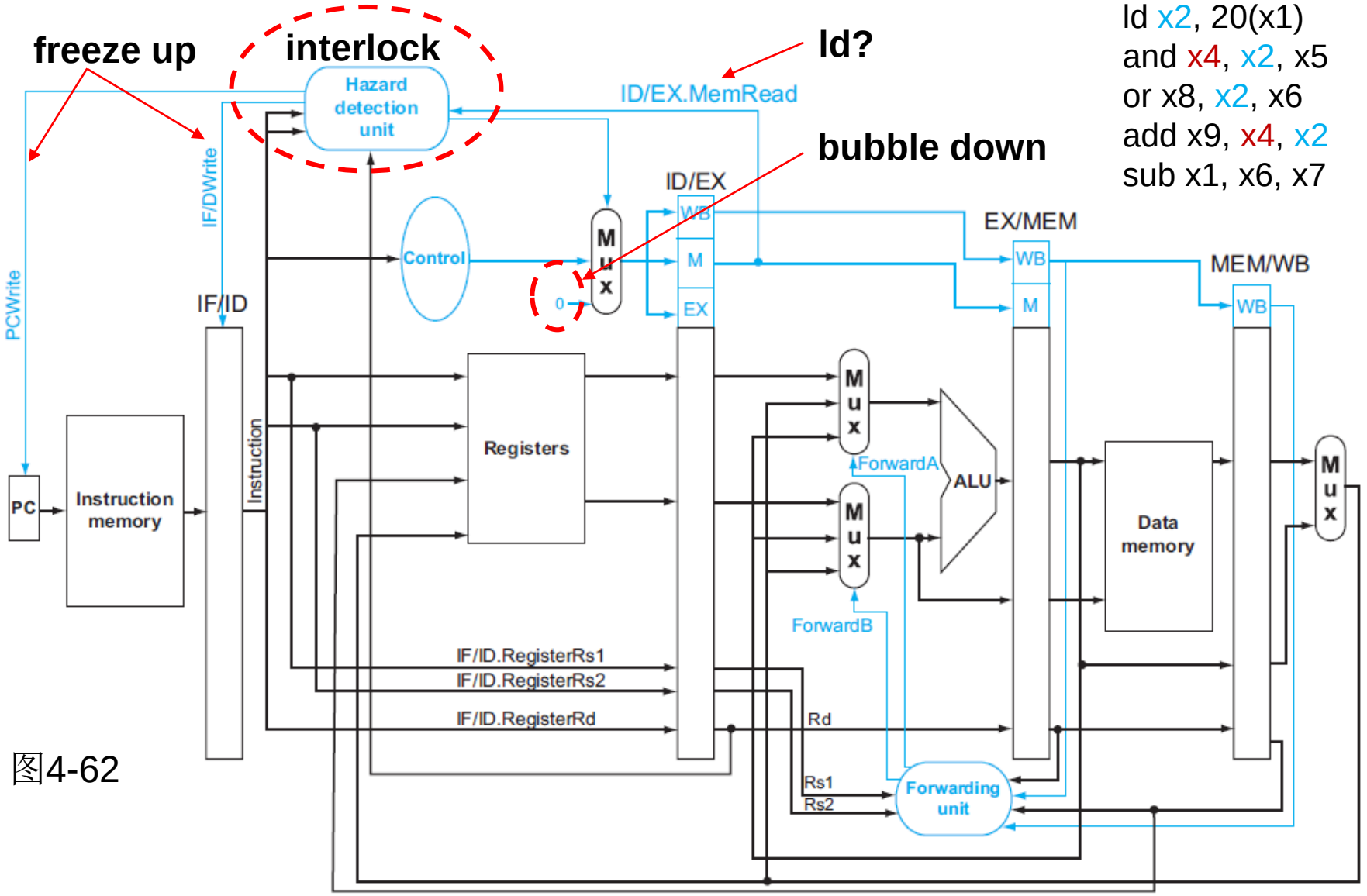
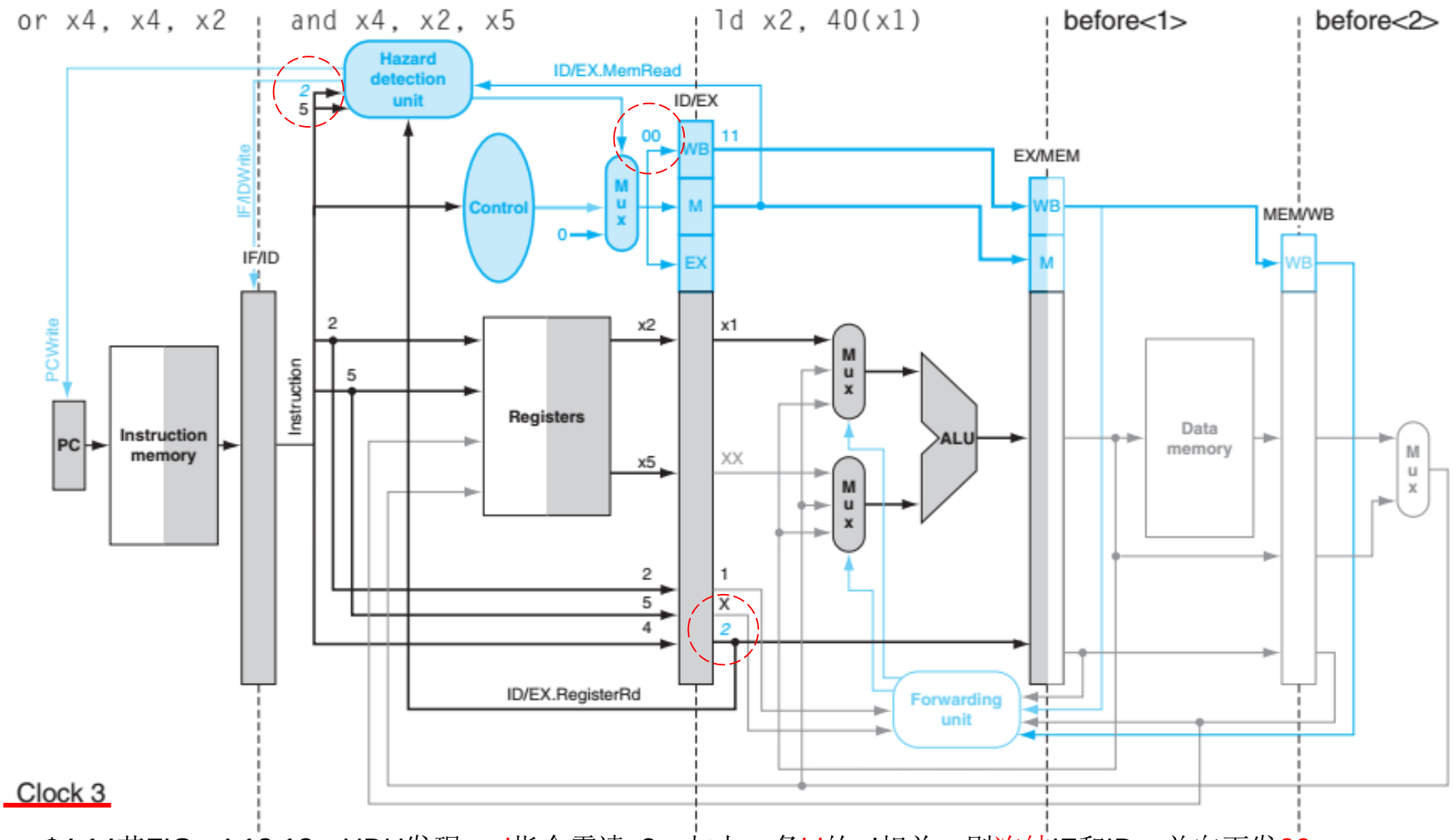


图4-62

load-used， single-cycle diagram

```
ld    x2, 40(x1)
and   x4, x2, x5
or    x4, x4, x2
add   x9, x4, x2
```



$4.14节FIG e4.13.18：HDU发现and指令需读x2，与上一条ld的rd相关，则冻结IF和ID，并向下发00

load-used， single-cycle diagram

```
ld    x2, 40(x1)
and   x4, x2, x5
or    x4, x4, x2
add   x9, x4, x2
```

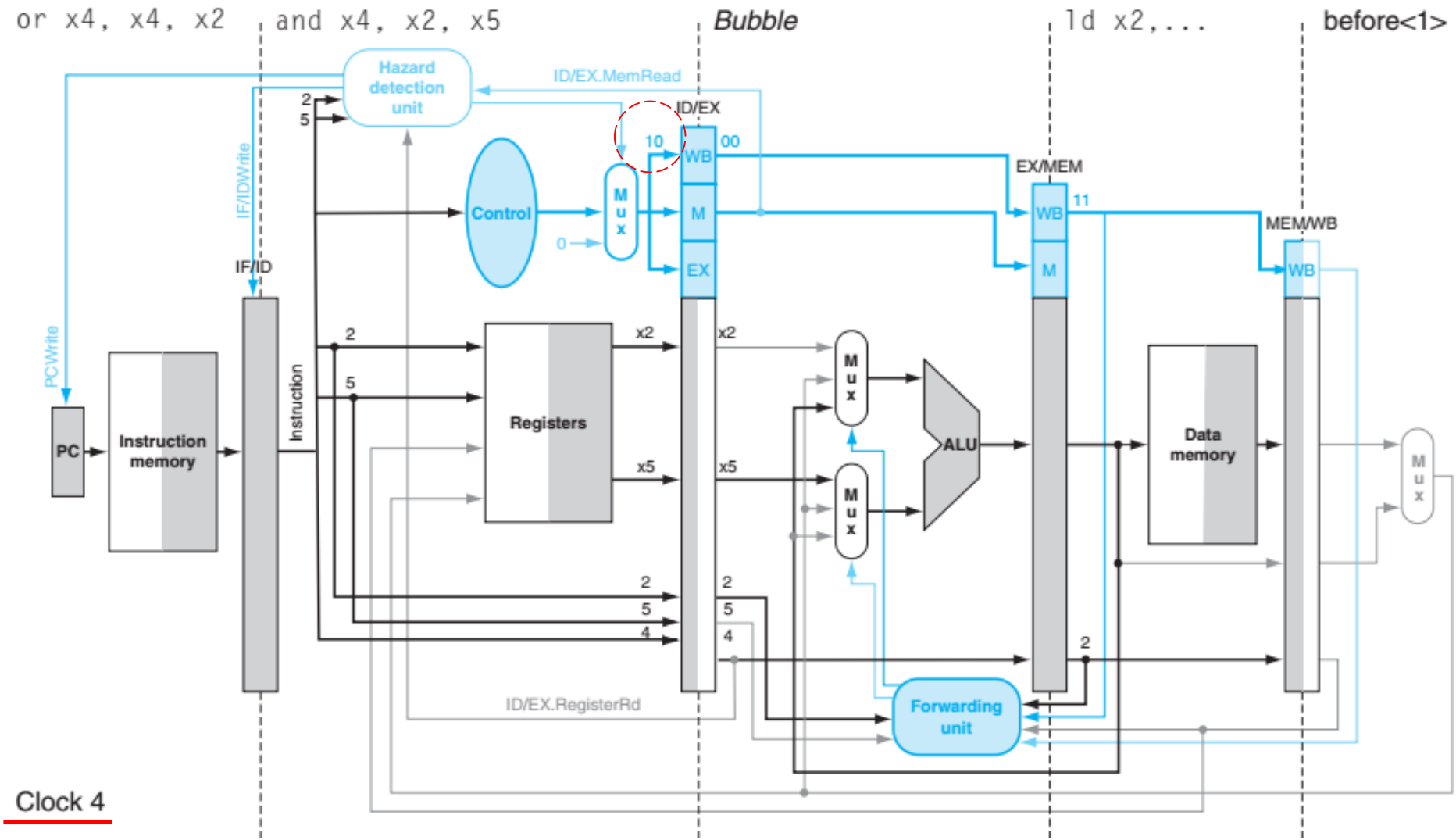


FIGURE e4.13.19：and和or重复，EX段bubble。注意，此时FW也工作，但无害。

load-used， single-cycle diagram

```
ld    x2, 40(x1)
and   x4, x2, x5
or    x4, x4, x2
add   x9, x4, x2
```

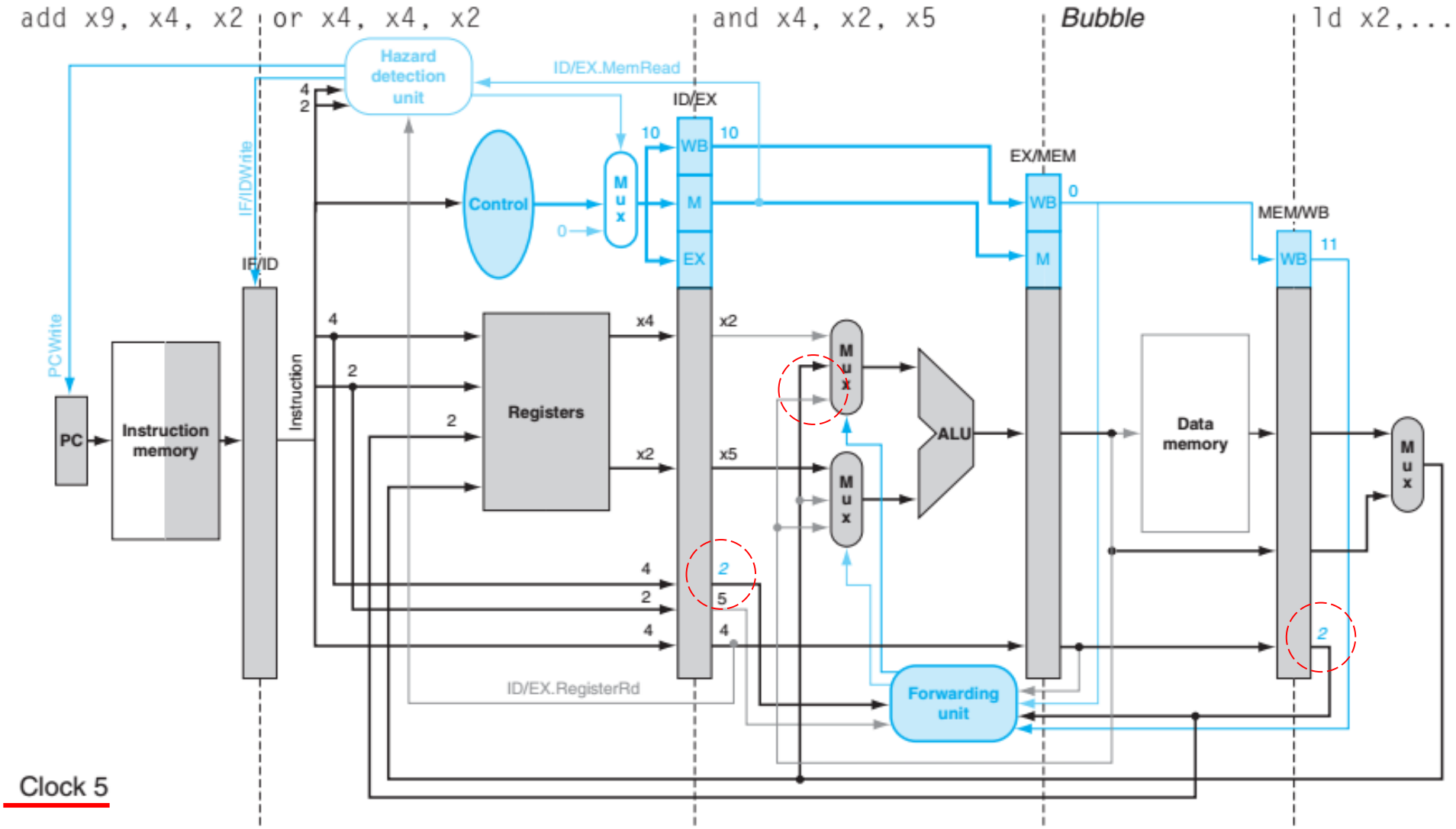
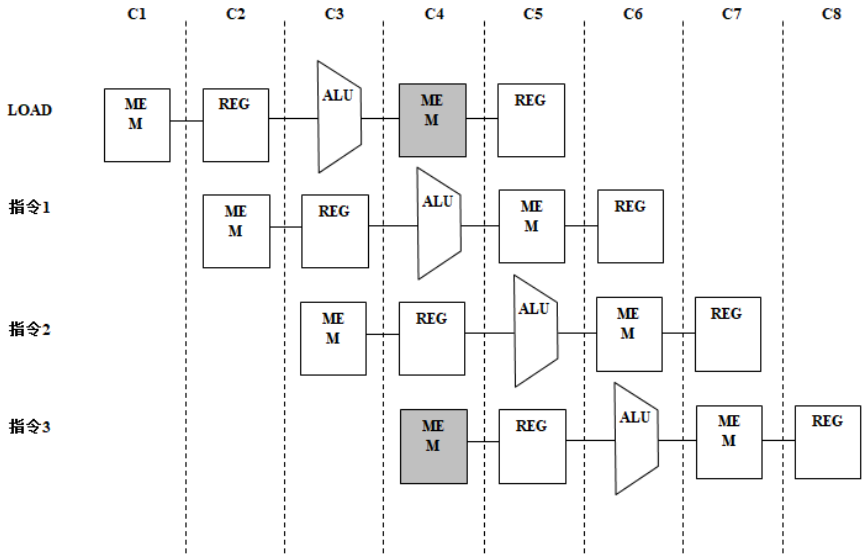


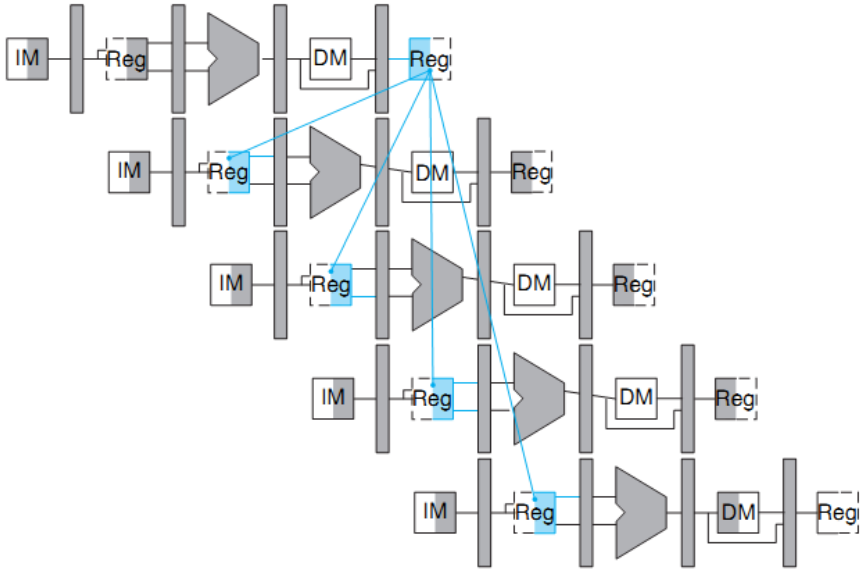
FIGURE e4.13.19： and进入EX，需FWU从MEM/WB前推x2。此周期ld写回（先写），故or无需FW x2

小结：基本5段流水线中的冒险

- 假设：单周期整数部件，单周期访存
  - 结构冲突：同一cc，哈佛结构已经消除，无须stall
    - 单端口MEM：IF，MEM
    - RF单一写端口：lw-Rtype
    - 胡伟武书：“RAW阻塞可能造成IF或ID结构冲突”？
  - RAW依赖：不同cc，生产者-消费者，消费早于生产，“三有四无”
    - ALU源操作数：前推，消除
      - “每条指令只写一个结果，且只在WB段写RF”
      - load-used-hazard：stall+前推，缓解
    - Store-hazard：lw-sw，前推，消除
  - 分支
- 冒险处理：FWU（关键路径），interlock（检测，stall），flush
  - “基本五段流水线的所有冒险在ID段处理完，其后不会被阻塞”
- RAW优化（CPI / IPC?）：前推，指令调度（编译），预测/投机
  - ALU源操作数：从EX/ME、ME/WB向EX段前推
    - load-used-hazard：向EX段前推，ME/WB
  - Store-hazard：从ME/WB向ME段前推

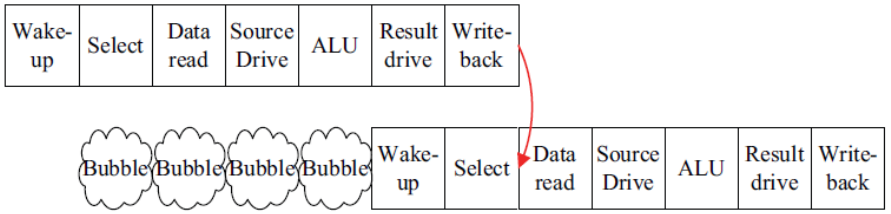
结构冲突，RAW，时序





- 结构冲突—同一cc，RAW依赖—不同cc

作业



- 思考
  - 流水线深好浅好？
    - $4.18 三级流水线
  - 影响流水线性能发挥的因素有哪些？
  - 哪些情形可能造成流水线stall，有哪些解决方案？
    - Stall的语义是“freeze up，bubble down”，如何实现？
    - 指令流水线中在哪个阶段会产生什么相关？
    - 有哪些RAW类型？
    - MEM段前推的判据？
    - 何时检查FW和interlock？
    - load-use-data Forwarding规则？
    - ld/sd存在D$冲突？
    - 编译器如何优化stall？
  - 结构相关与数据相关的本质区别？
  - 为何RISC只有load-store指令访存？
  - 为何RISC一条指令仅在最后一个流水段写一个结果？
- 作业： 4.16.4~5，4.22


Thank you