

The RISC-V Processor Implementation: Pipeline——ILP

**“Computer Organization & Design”
David Patterson, John Hennessy**

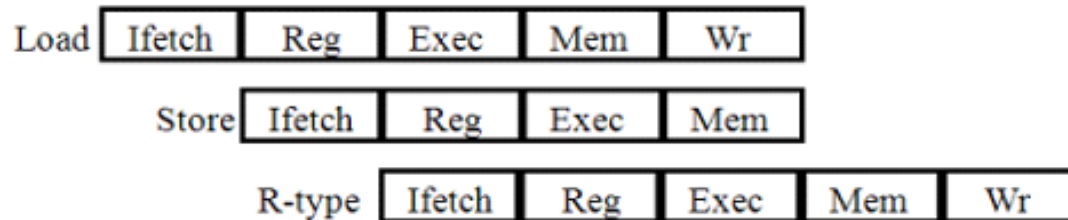
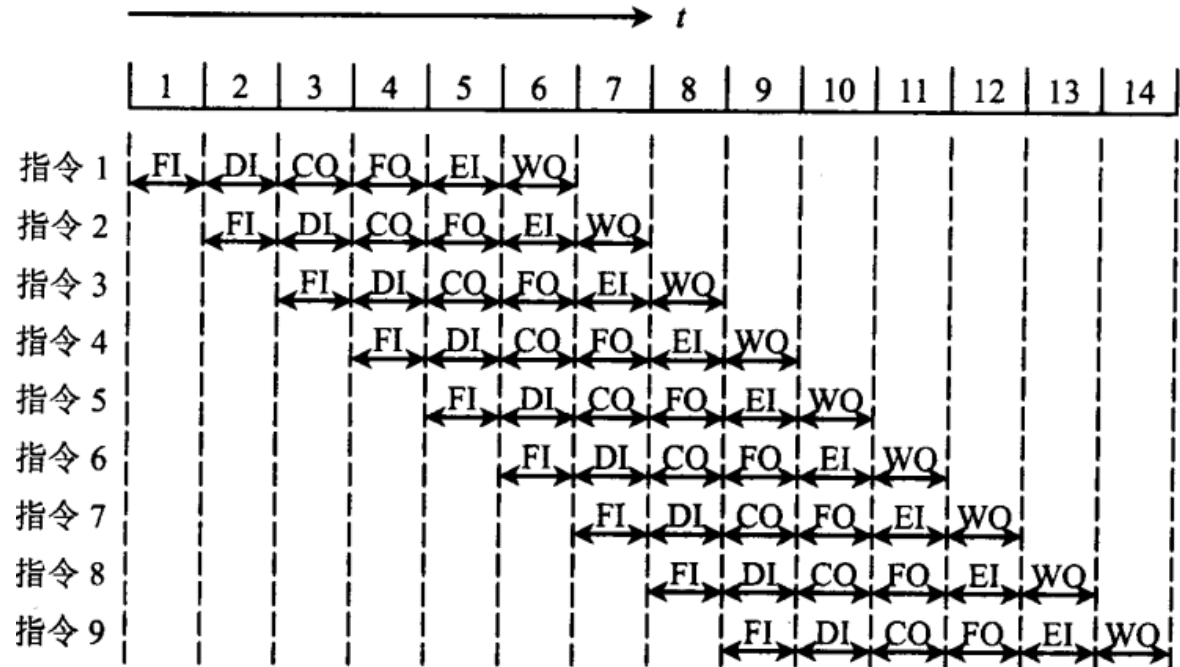
CPI = 1, CPI ≥ 1?

Pipelining Idealism:

平衡, 重复, 无冲突

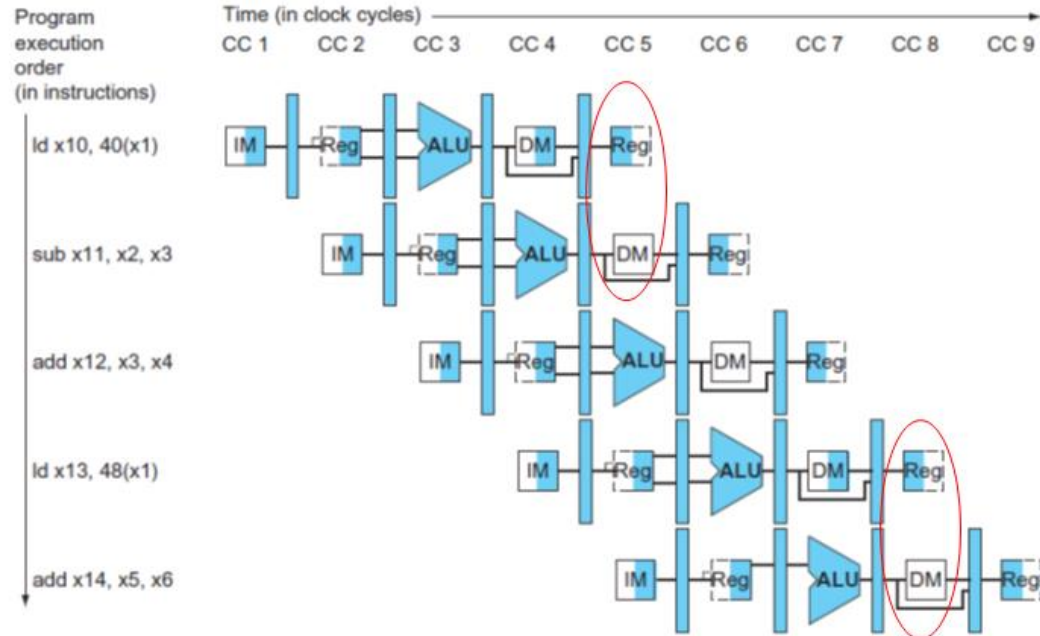
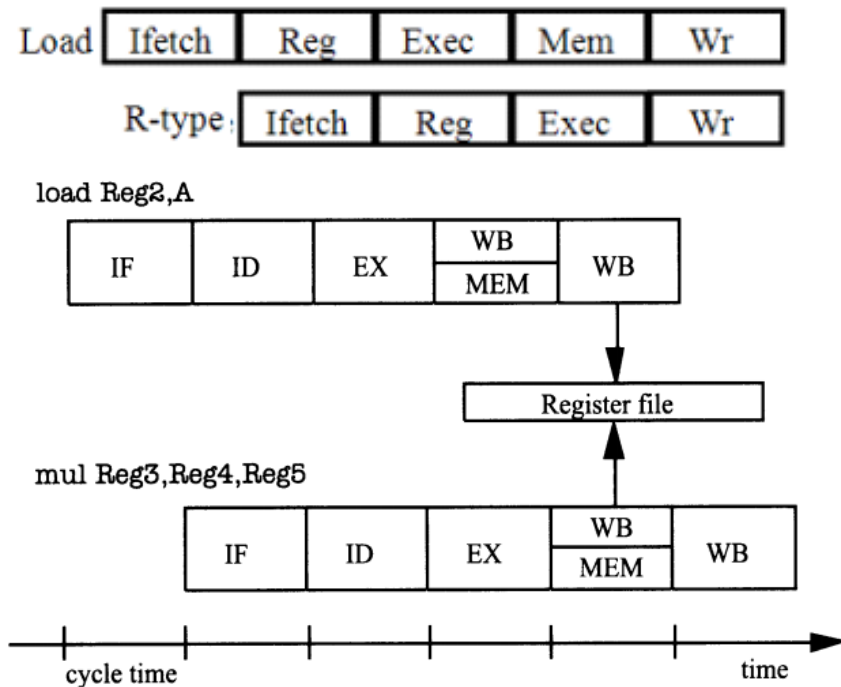
Speedup = k

CPI ≤ 1, IPC ≥ 1?

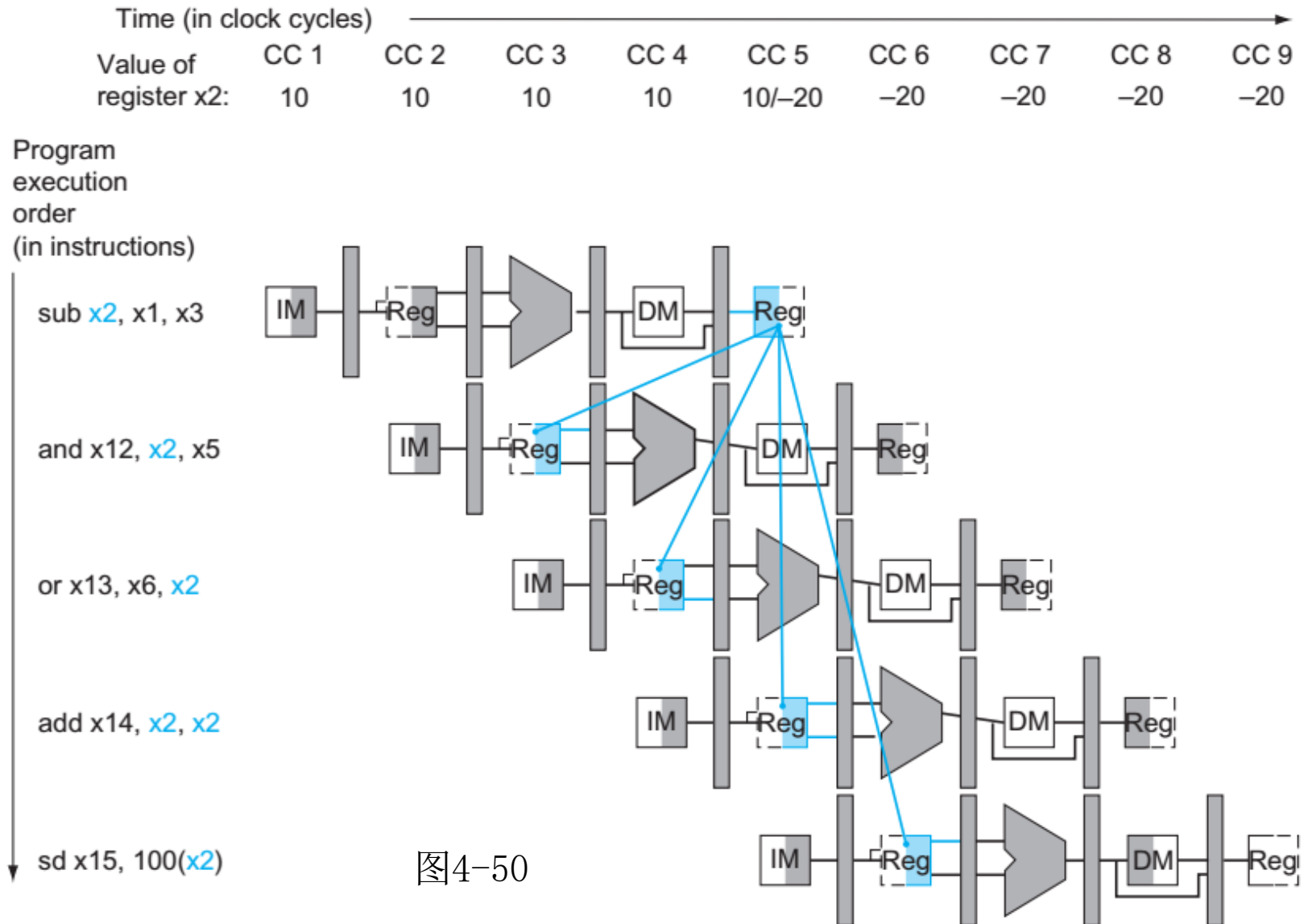


R-type指令完成时间: load-Rtype

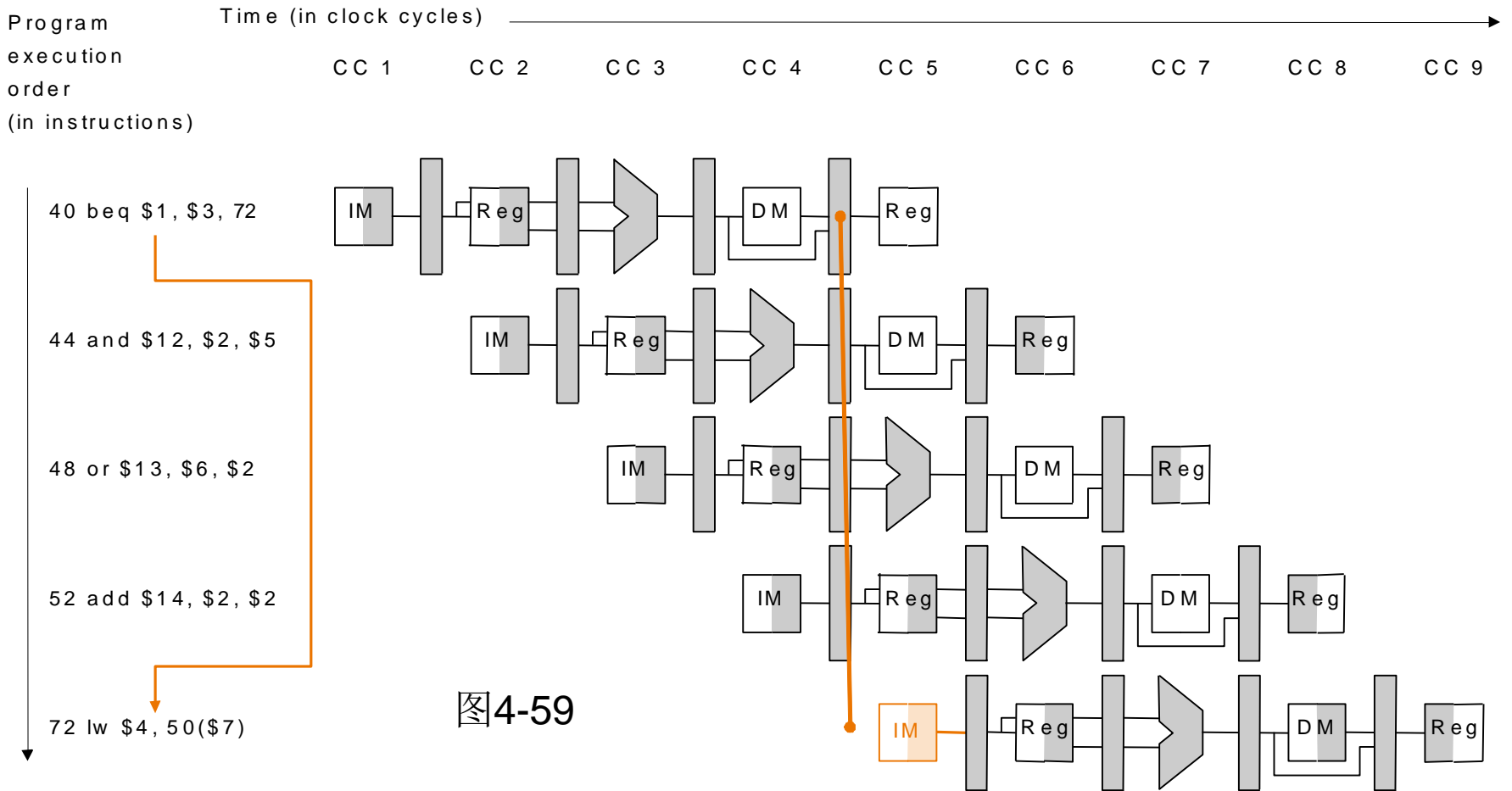
- R-type指令4个周期完成?
 - 例: 当前指令为 lw, 下一条指令为R-type。CC5?



哪个周期数据可用？



beq taken?



内容提要

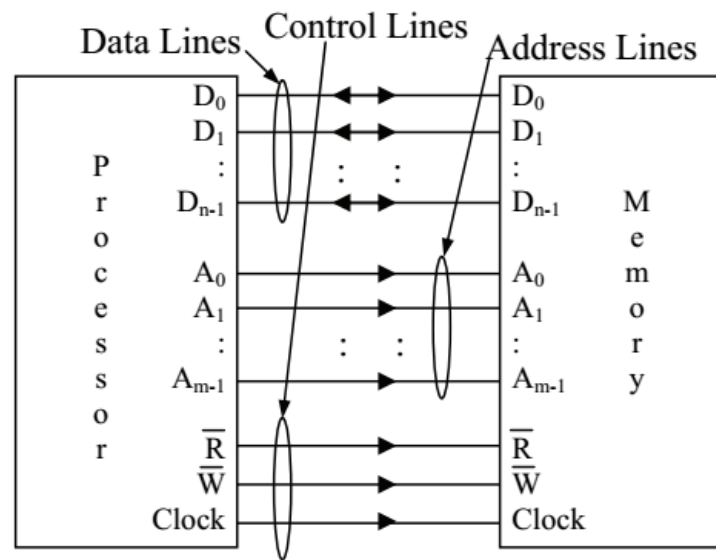
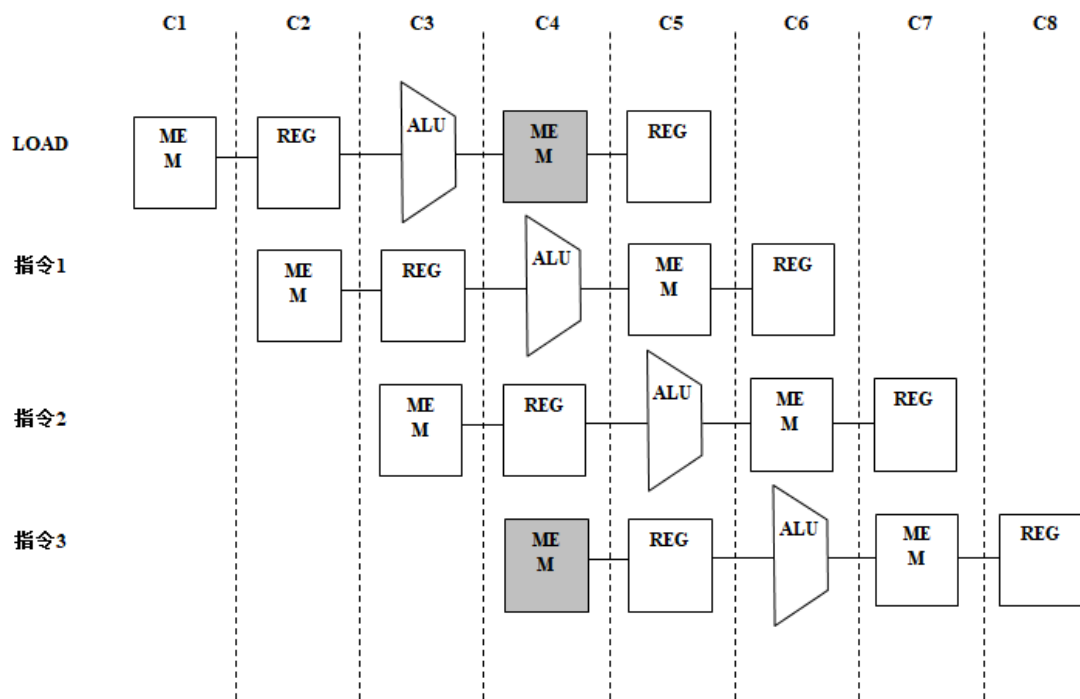
- 流水线技术原理：4.5
- RV的五级流水线实现：4.6
- 冒险（Hazard）问题：4.5.2
 - 结构冲突：stall语义，哈佛结构
 - 数据依赖：4.7
 - 编译技术：插入nop，指令重排，寄存器重命名
 - forwarding技术：RAW
 - Interlock技术：Stall
 - 控制延迟：4.8
 - 编译技术：flush语义，延迟分支
 - 硬件优化：提前完成，投机，预测
- 多发射技术：4.10
- 硬件多线程：7.5

流水线的“hazard冒险”问题

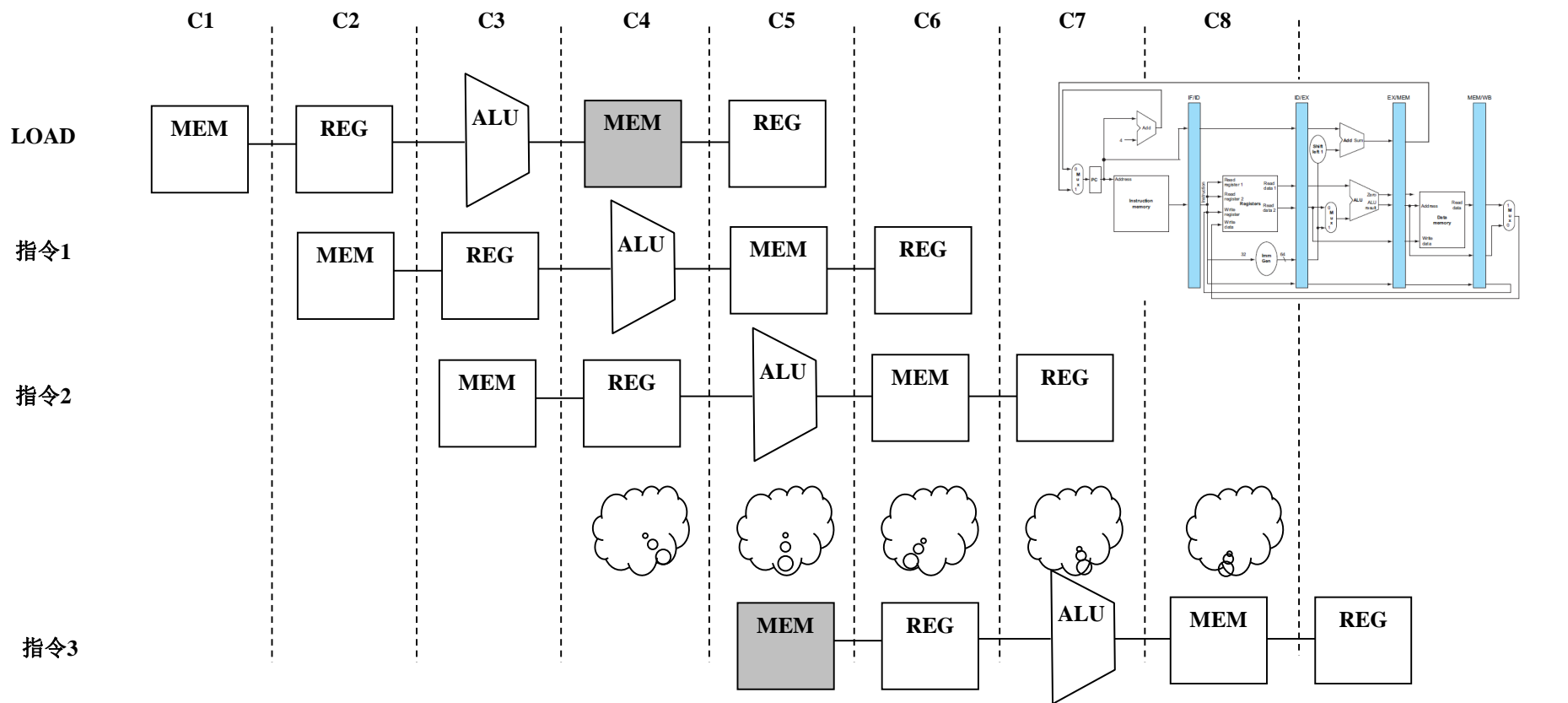
- 结构相关：当指令在重叠执行的过程中，硬件资源满足不了指令重叠执行的要求，发生**资源冲突**时将产生结构相关。
- 数据相关：当一条指令需要用到前面指令的执行结果，而这些指令均在流水线中重叠执行时，就可能引起数据相关。
- 控制相关：当流水线遇到分支指令和其他会改变PC值的指令时，会发生控制相关。
- ILP：资源conflict，数据dependency，控制penalty

访存结构相关

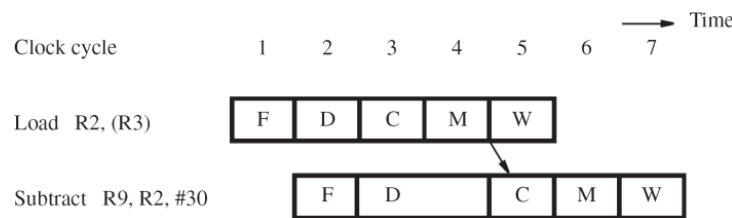
- 假设单存储器（单端口），则同一周期内并发取指与读写数据存在冲突



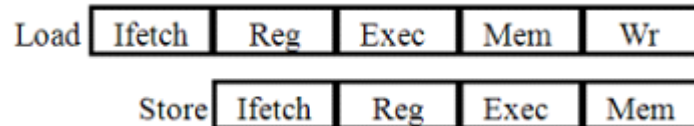
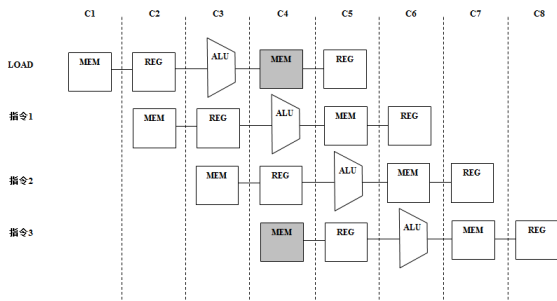
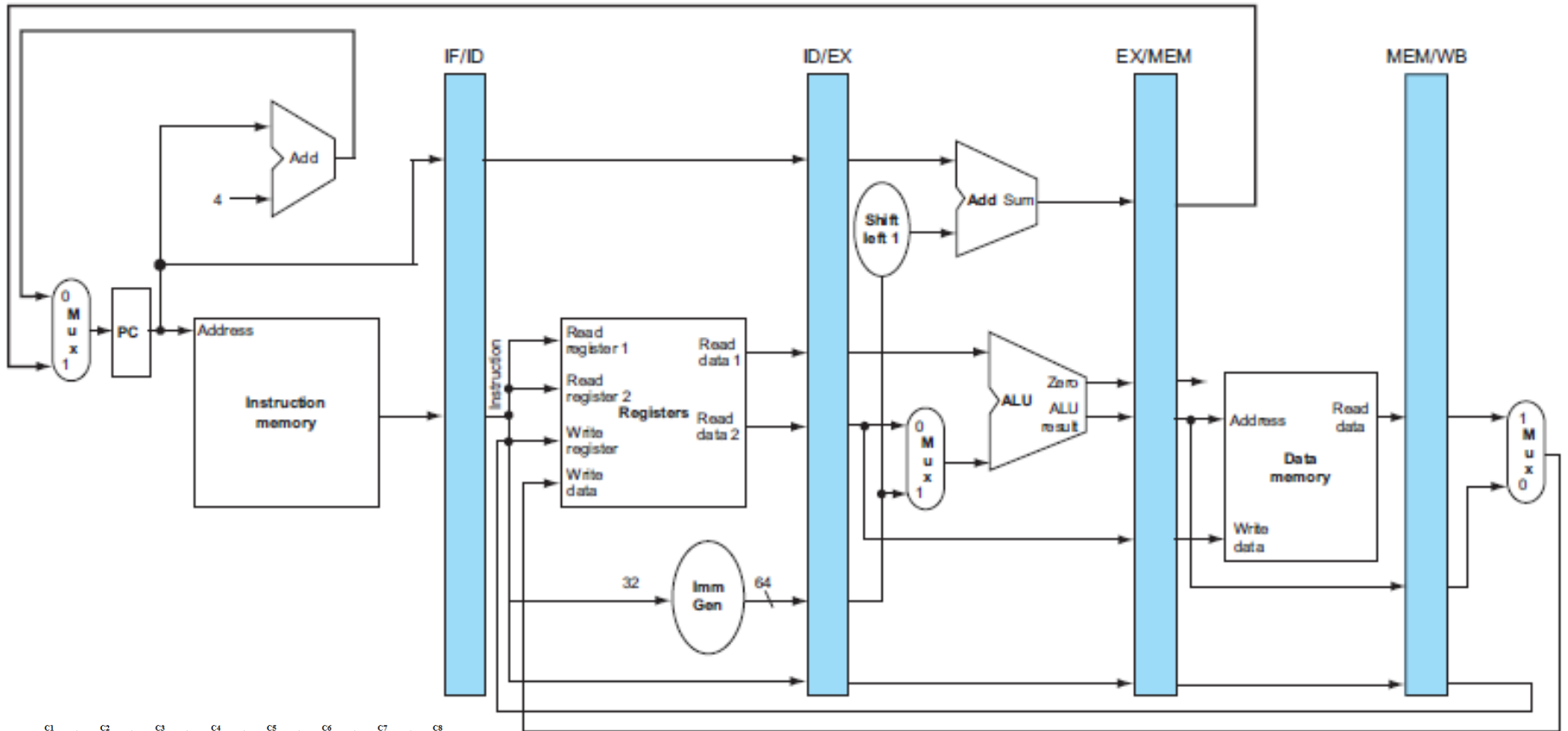
流水线stall（停顿，阻塞）语义



- 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
 - IF段停顿
 - PC不变，重复取指；IF/ID清零 = 插入气泡（bubble）



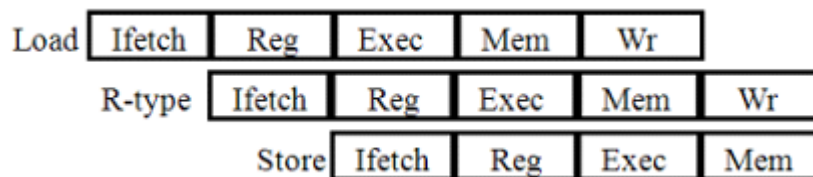
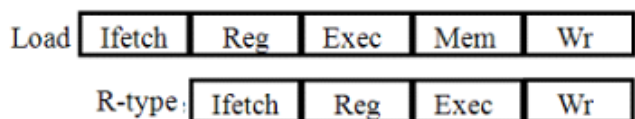
IF与MEM结构冲突消除：哈佛结构



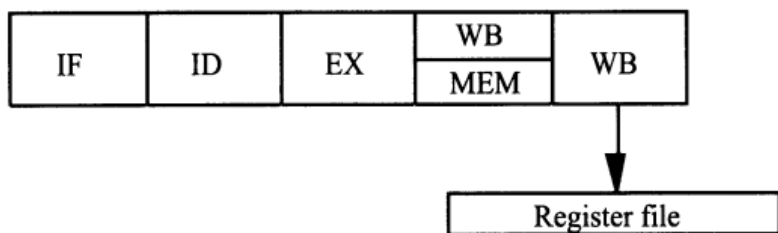
lw/sw是否存在D\$冲突?

lw-Rtype结构冲突（RF写端口冲突）消除

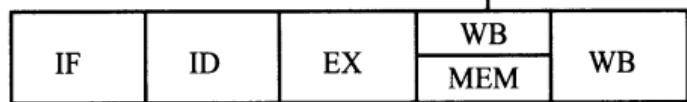
- 规定：** R-type指令在WB段完成



load Reg2,A



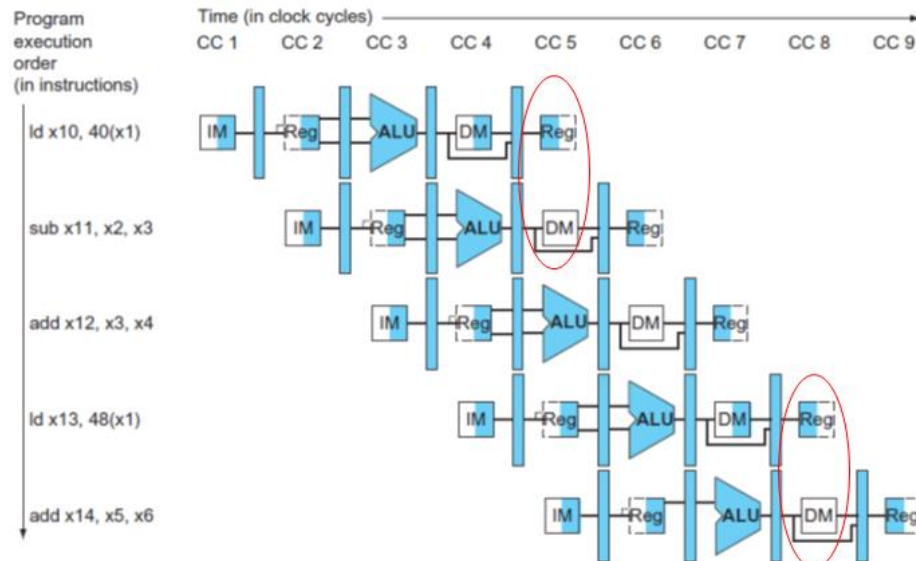
mul Reg3,Reg4,Reg5



cycle time

time

RF “两读一写”，写端口冲突！

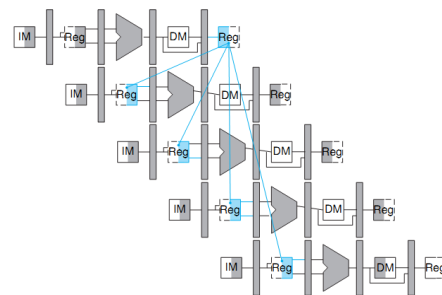


数据相关分类

- 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
 - j 可能在 i 读取某个源寄存器的内容之前就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。

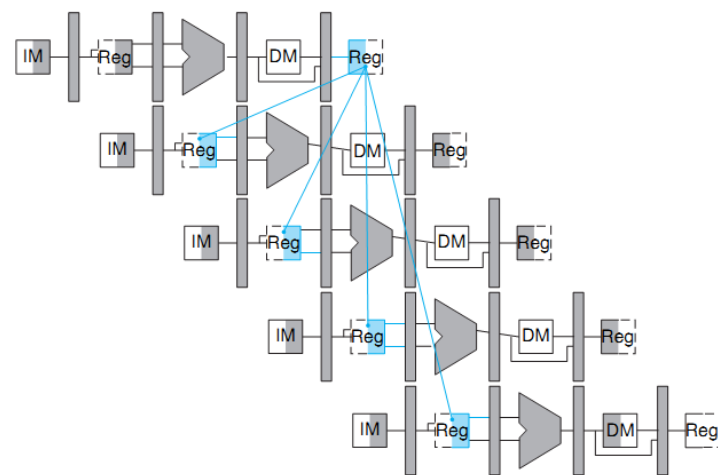
- 哪些真会发生?

- 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？

- 共享资源：Reg, mem
- 程序中存在
 - 真依赖：RAW, 生产者-消费者
 - 假依赖（名字依赖）：p233
 - 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）：

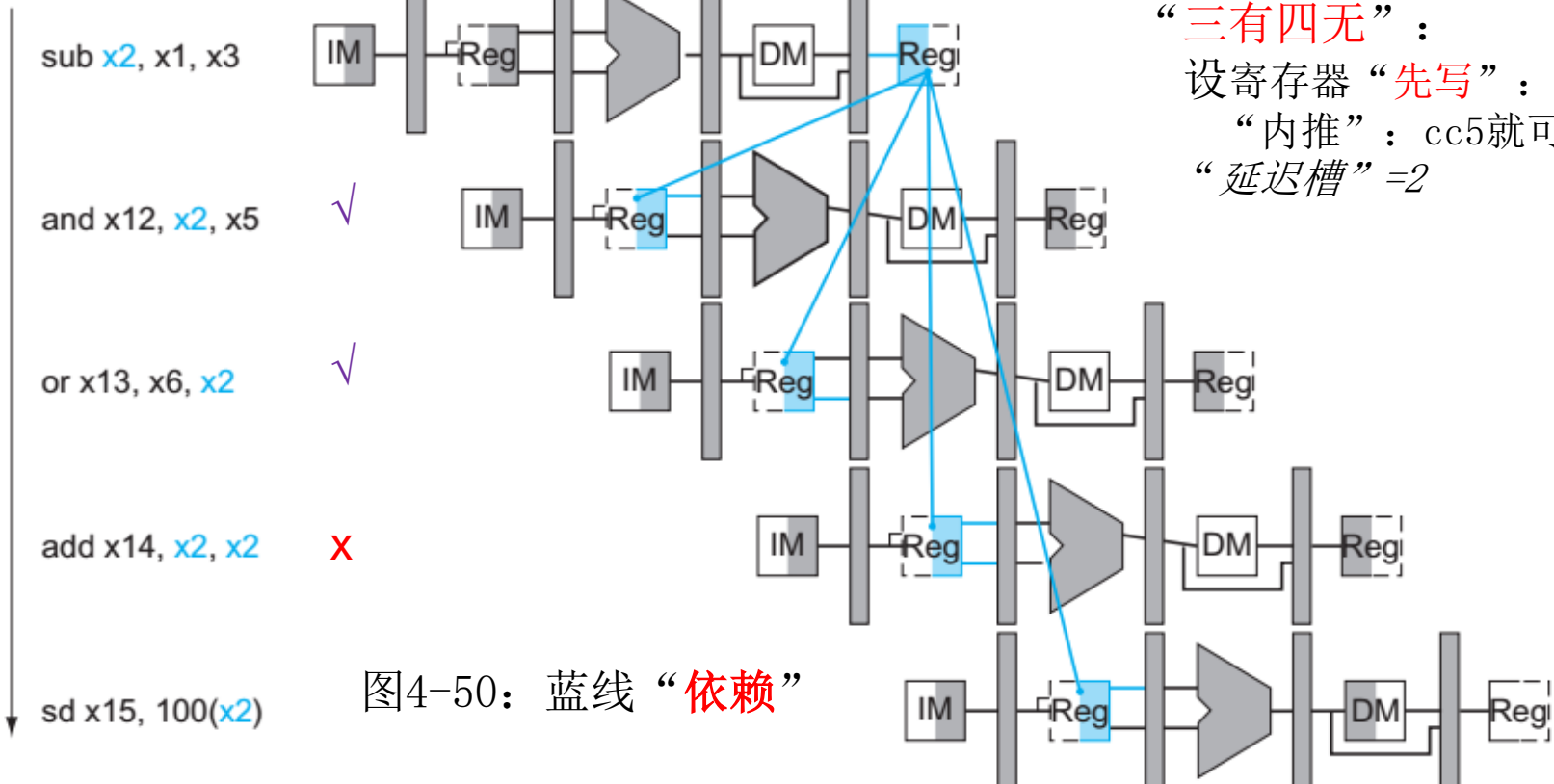


- 物理上：“简单流水线中不存在WAR、WAW”！！！！
 - “每条指令只写一个结果，且只在WB段写RF”
 - **RV第二版p350**；llxx “这三点是充要条件”？
 - 按序执行，且
 - **Reg-Reg**指令仅在最后流水段写回，且
 - load/store指令在相同的流水段访存

RAW (哪个周期数据可用?)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register x2:	10	10	10	10	10/-20	-20	-20	-20	-20

Program execution order (in instructions)

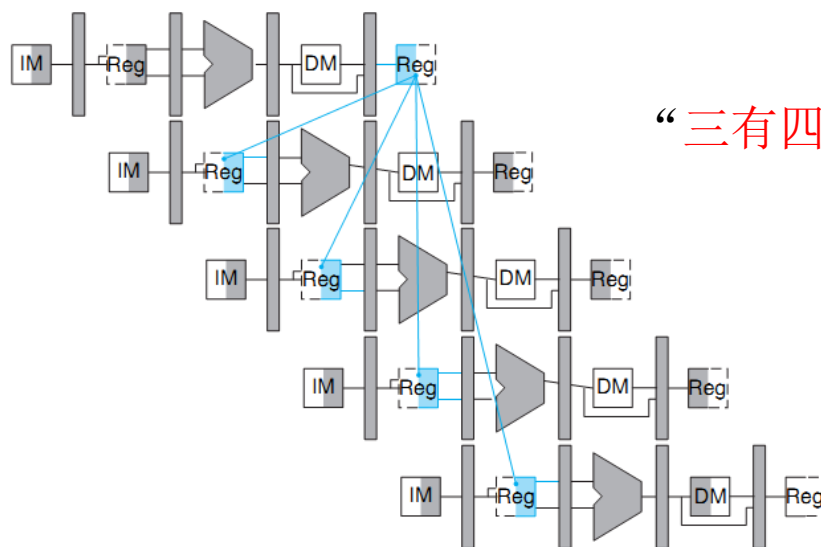
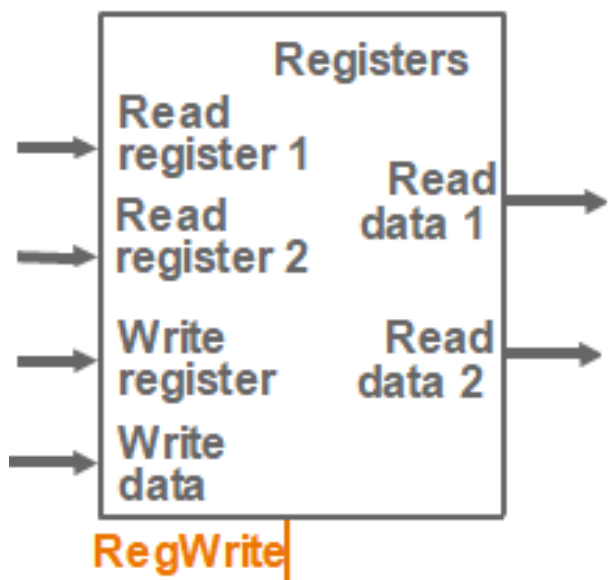


依赖类型：
 EXE段，MEM段，WB段
 “三有”：“有”：
 设寄存器“先写”：
 “内推”：cc5就可用！
 “延迟槽”=2

图4-50：蓝线“依赖”

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）：与后写相反。——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。图4-32，4-50



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

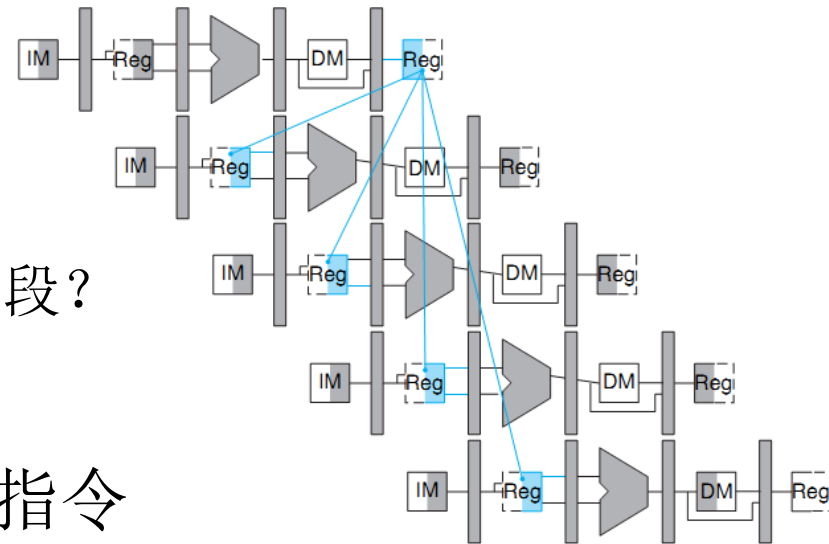
- freeze up & bubble down: 哪一段？

- 减少stall损失

- 指令调度：编译器填充无关指令

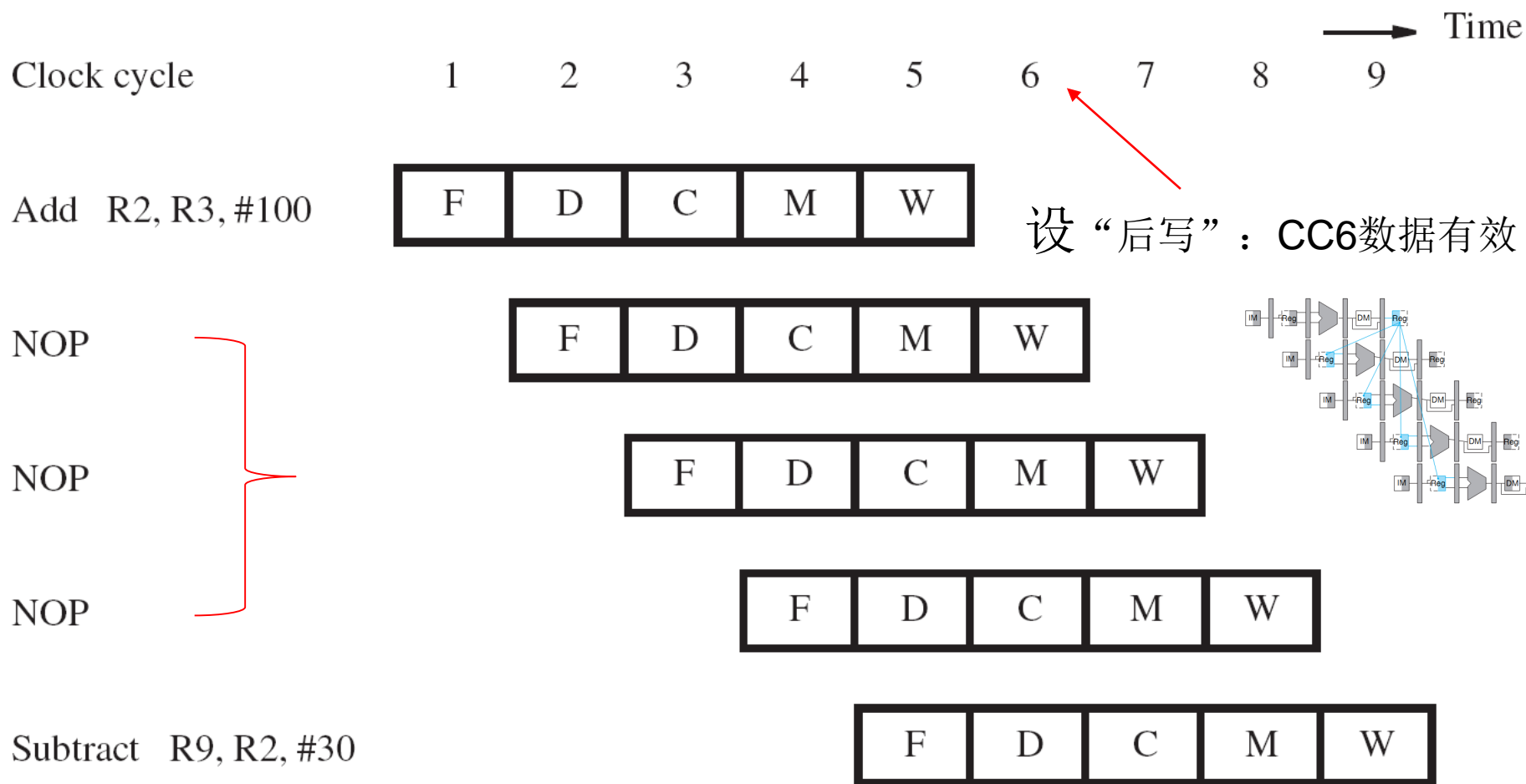
- 数据定向技术：动态技术

- 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, \$4.5.2

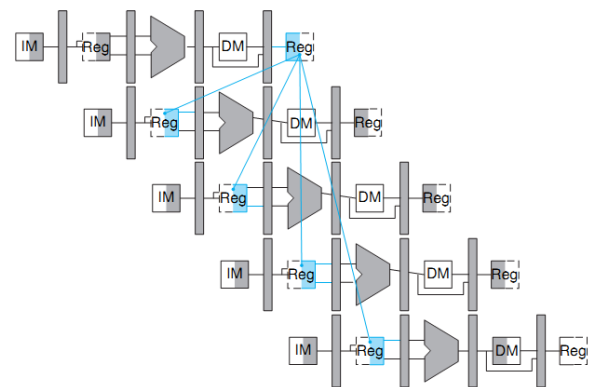
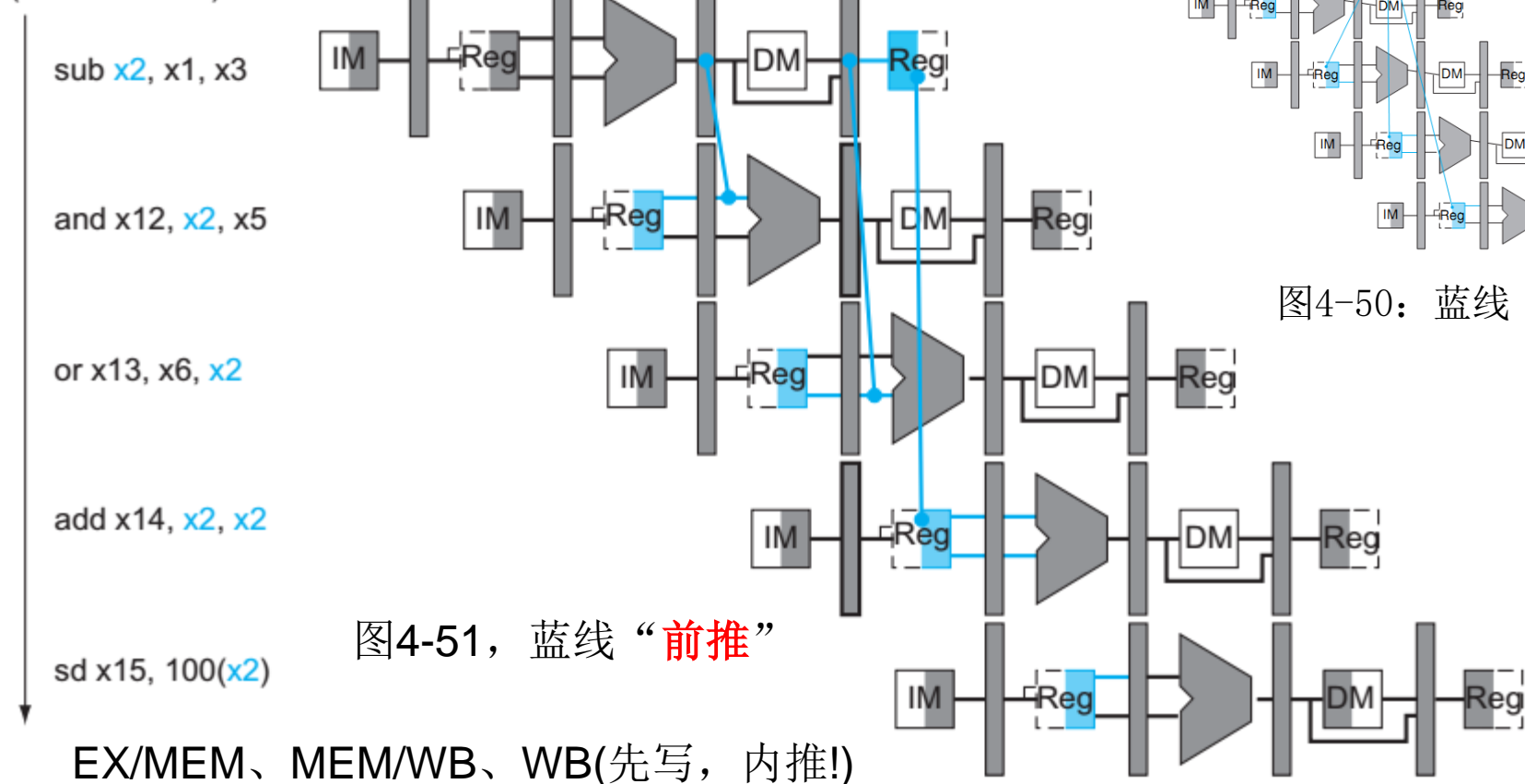
- 指令调度 (注意: 此例RF为“后写”——“4有5无”)



减少RAW损失: Forwarding to EX

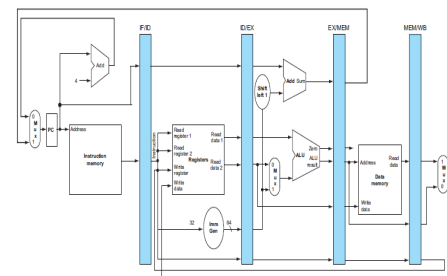
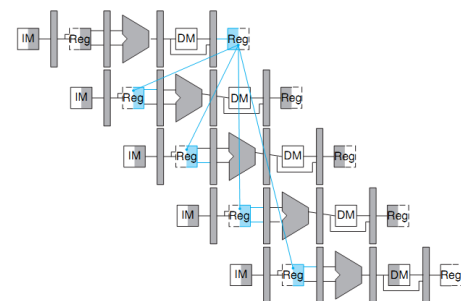
- “三有四无”

(in instructions)



EX段旁路：向EX段前推，检测规则？

- 将EX/MEM、MEM/WB前推



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

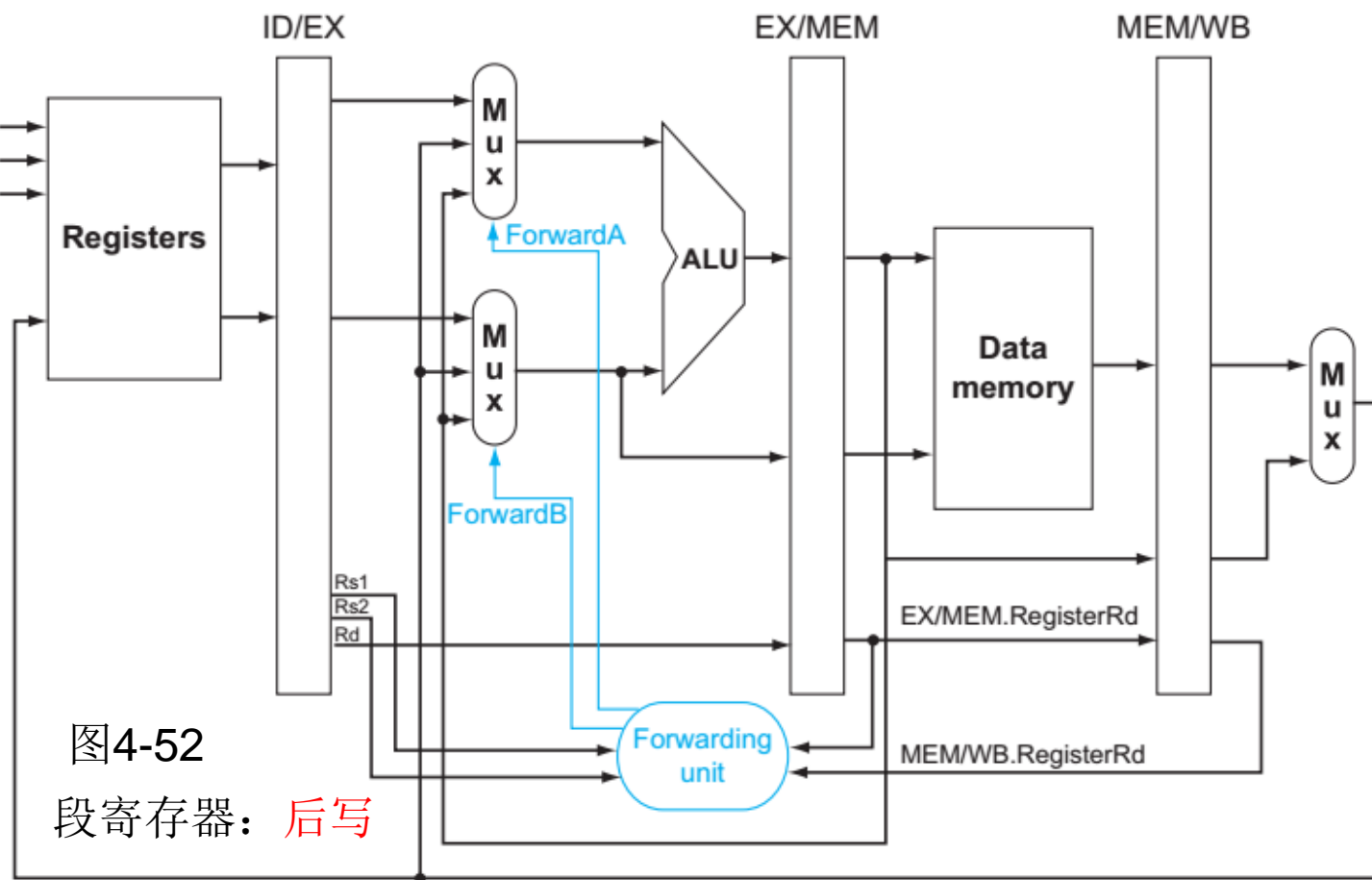


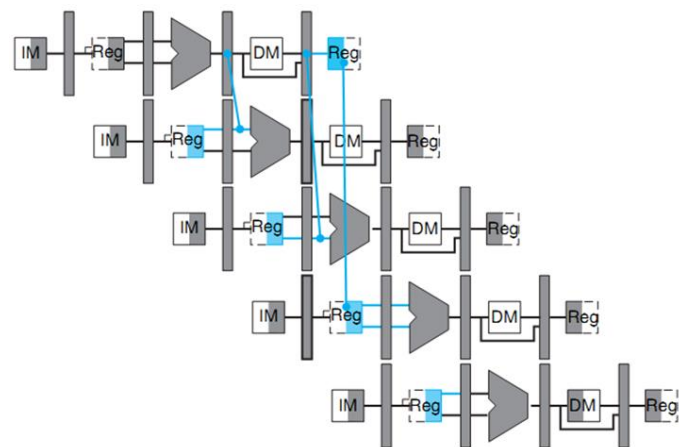
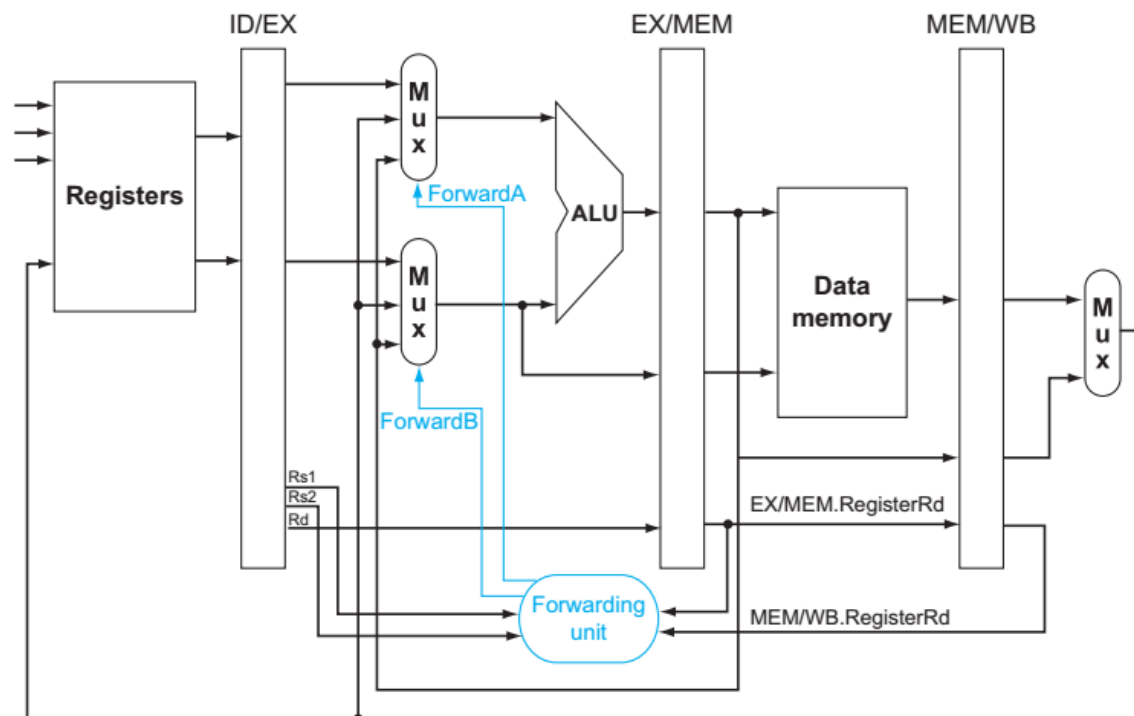
图4-52

段寄存器：后写

EX段旁路检测规则 (1/2) : “三有四无”, \$4.7

- 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



```

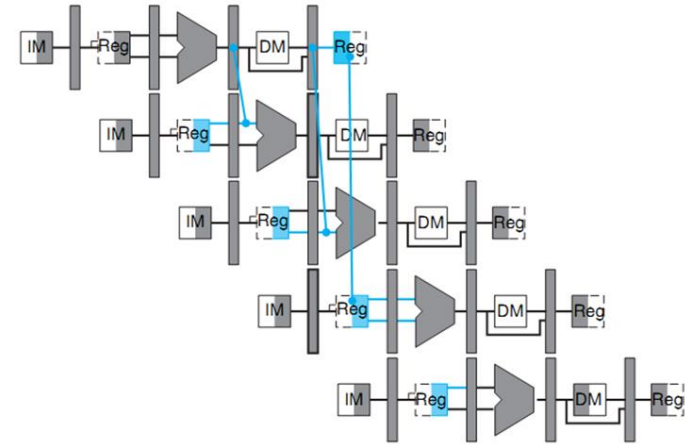
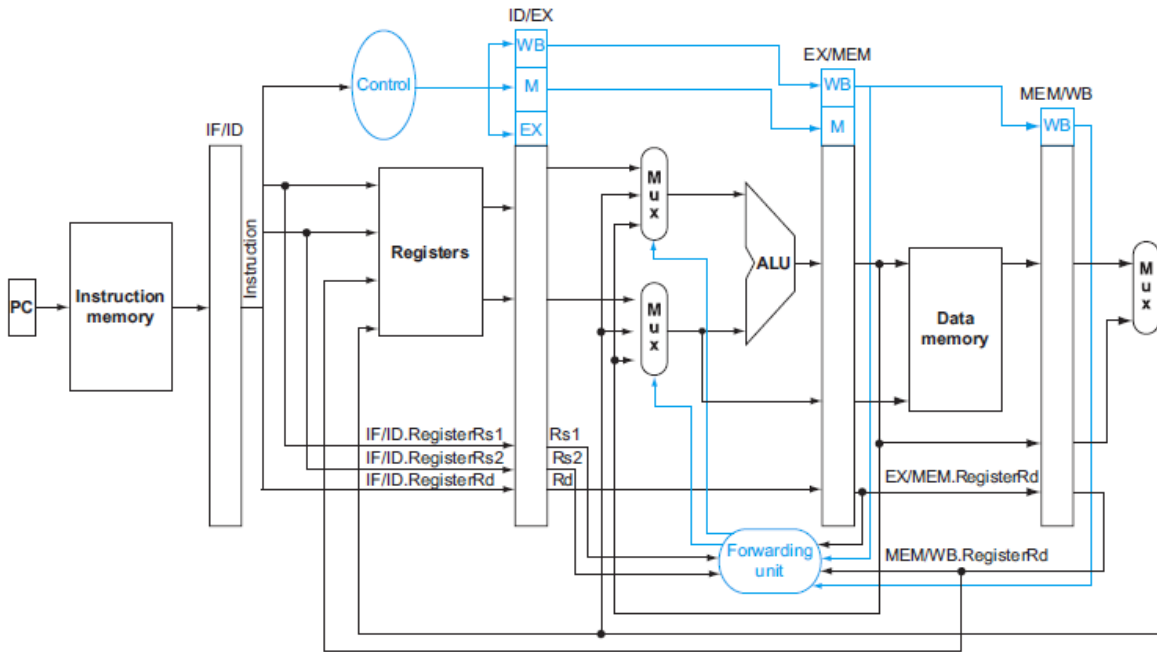
sub x2, x1, x3
and x12, x2, x5
or x13, x6, x2
add x14, x2, x2
sd x15, 100(x2)
    
```

funct7	rs2	rs1	funct3	rd	opcode	R-type
--------	-----	-----	--------	----	--------	--------

EX段旁路检测规则 (2/2) : 无须前推!

- 无须前推的情况
 - 非写回指令
 - RegWrite?
 - 目标寄存器为\$zero
 - RegisterRd \neq 0?

- 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



```

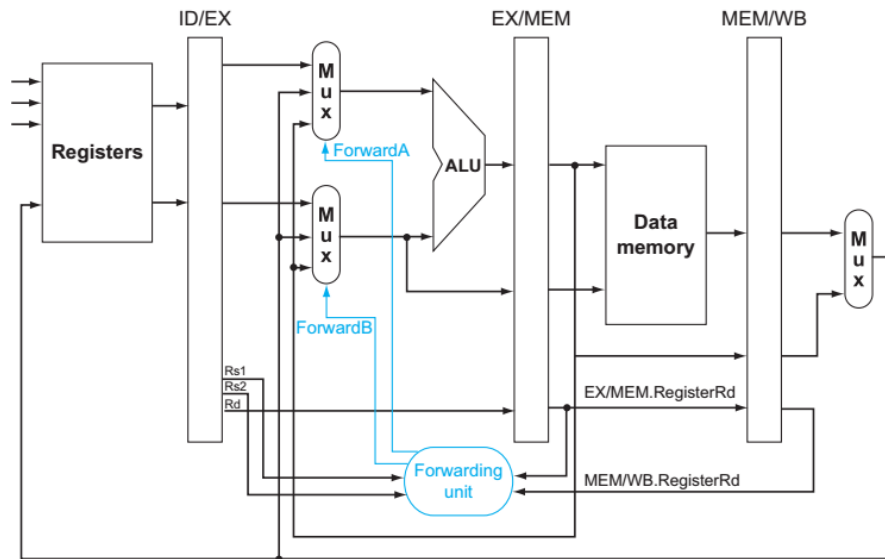
sub x2, x1, x3
and x12, x2, x5
or x13, x6, x2
add x14, x2, x2
sd x15, 100(x2)
    
```

funct7	rs2	rs1	funct3	rd	opcode	R-type
--------	-----	-----	--------	----	--------	--------

Forwarding控制器: bug?

Mux control	Source	Explanation
<u>ForwardA = 00</u>	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.
<u>ForwardB = 00</u>	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4-53



```

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
    
```

假依赖：求和，中文版p213

第三条依赖第一条，有

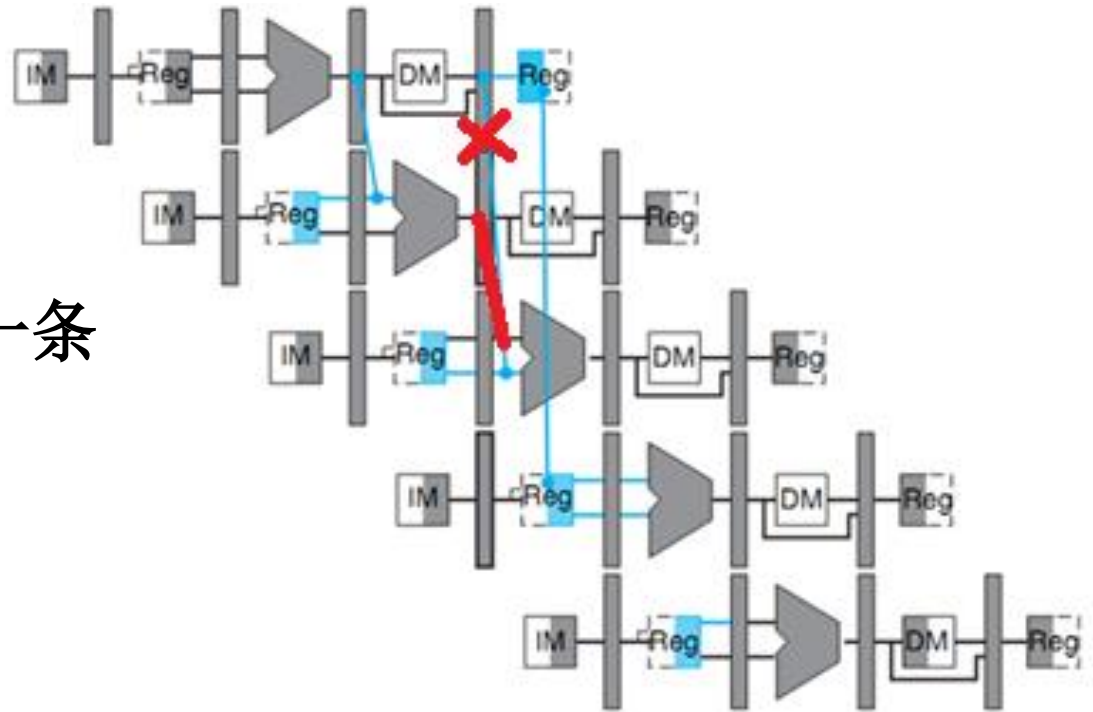
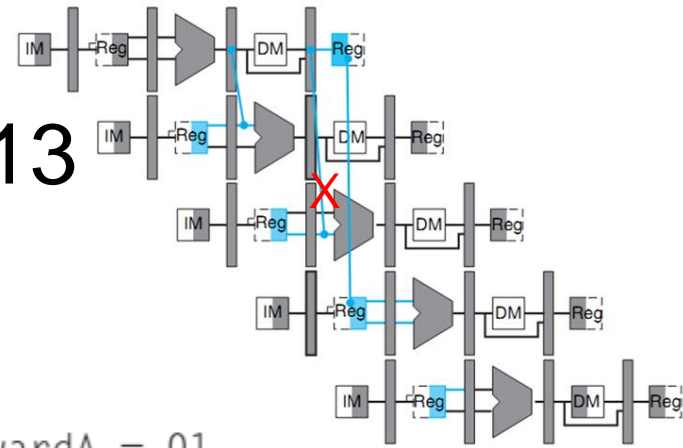
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
```

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

求和：第三条不依赖第一条

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

```
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: 第二条指令依赖第一条

```

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: 第三条指令依赖第一条, 不依赖第二条

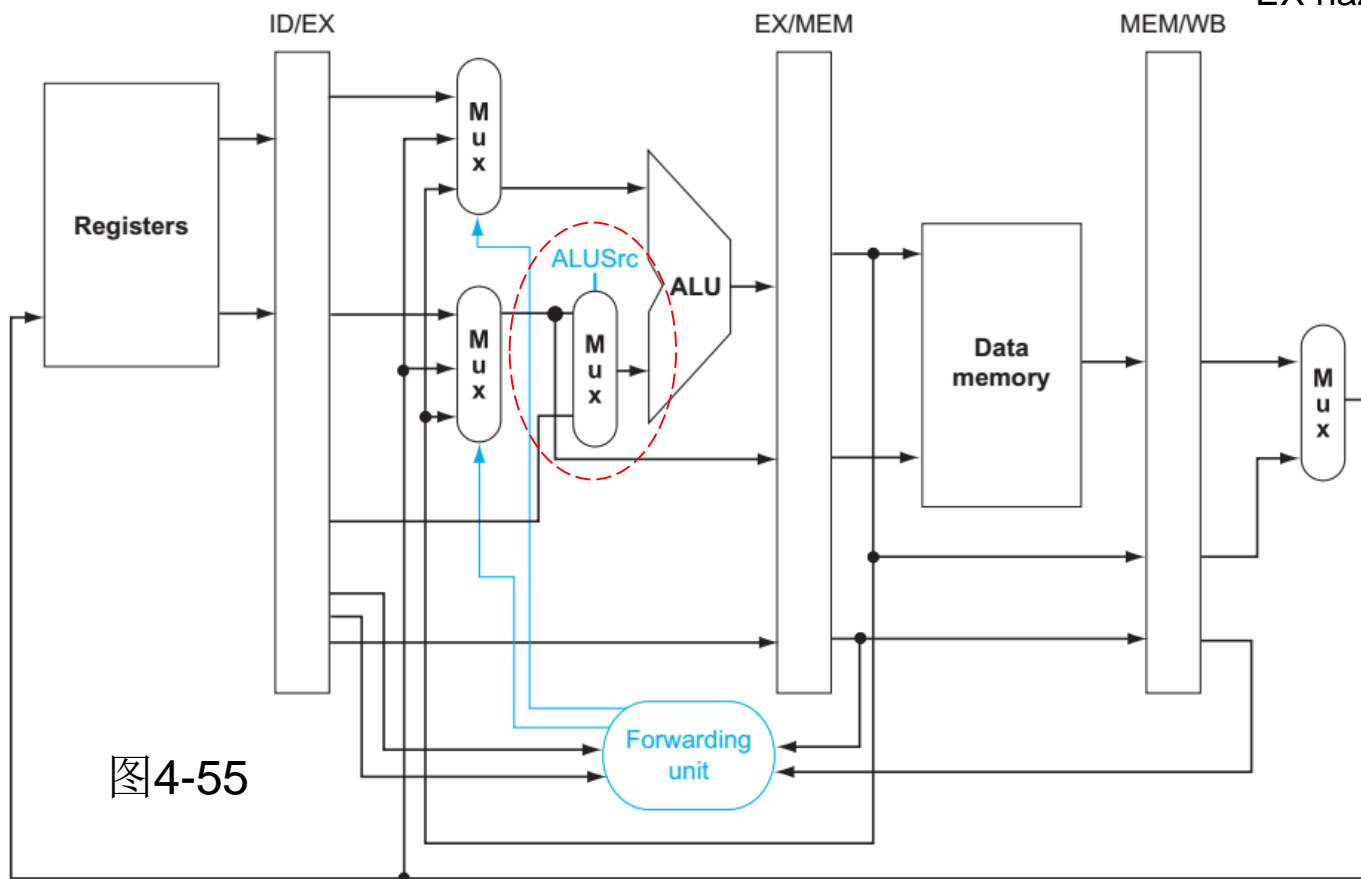
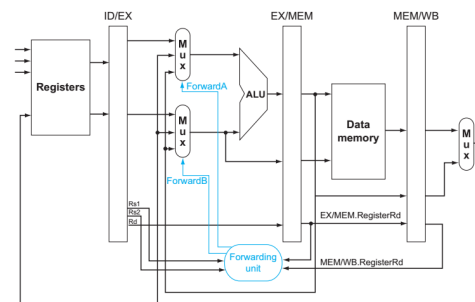


图4-55



I-type ALU: 立即数
图4-52无

immediate[11:0]	rs1	funct3	rd	opcode
-----------------	-----	--------	----	--------

I-type

EX段前推：数据通路与控制

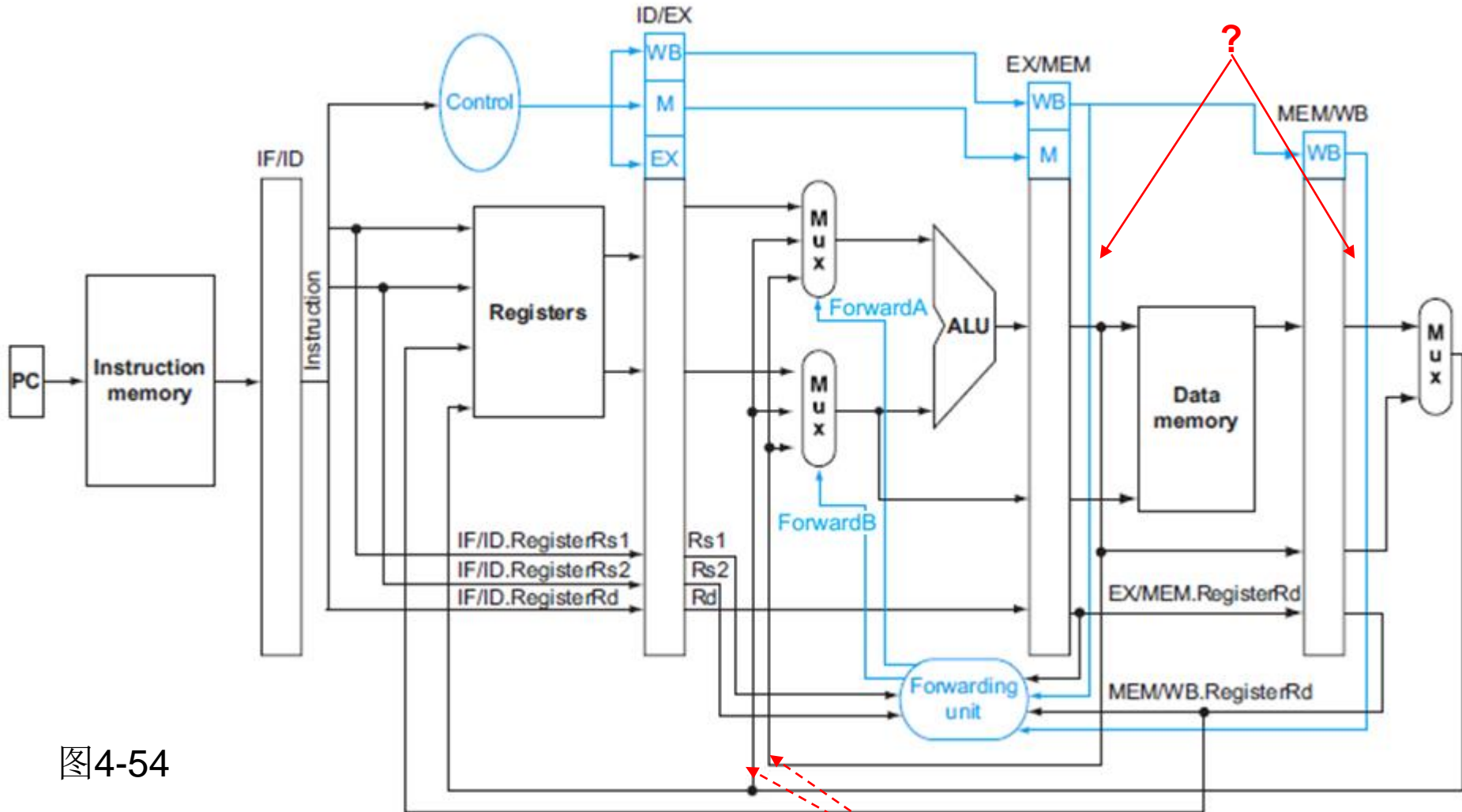


图4-54

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

访存指令的数据依赖问题

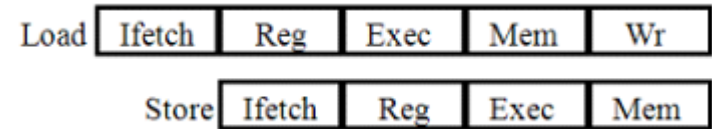
- `st x1, 100[x1]`
- `ld x3, 200[x2]`

- `ld x2, 100(x1)`
- `sd x2, 110(x1)`

- 设计假设: **Store**指令按序执行

store-hazard: p214精解译错?

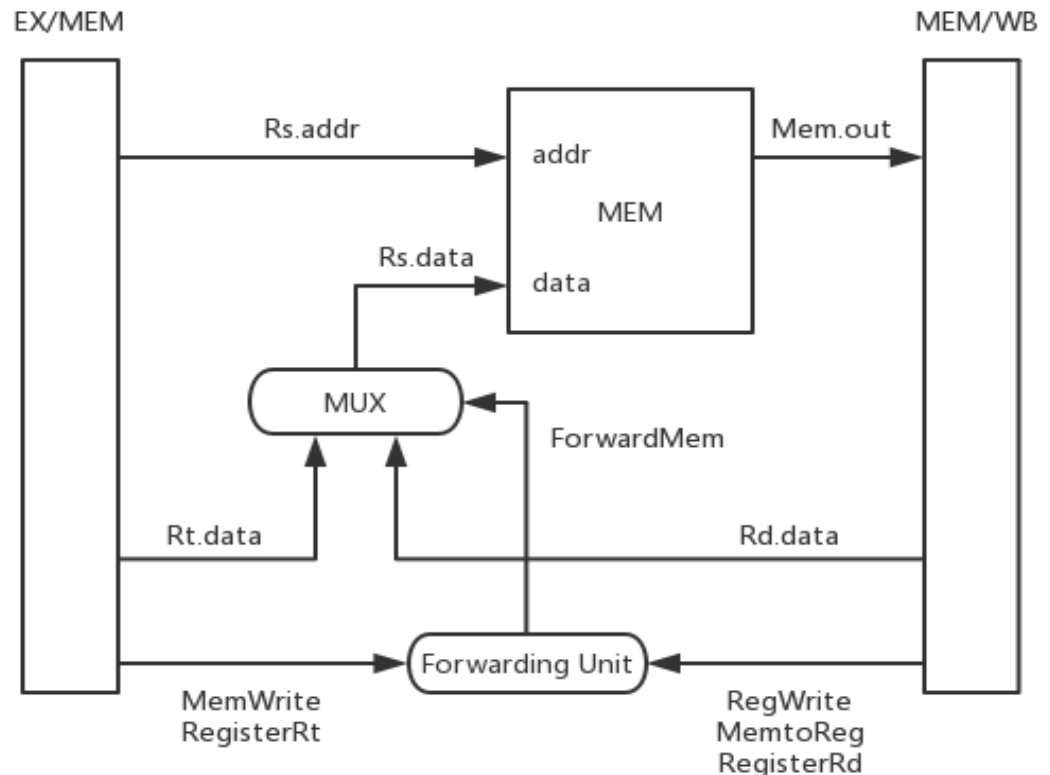
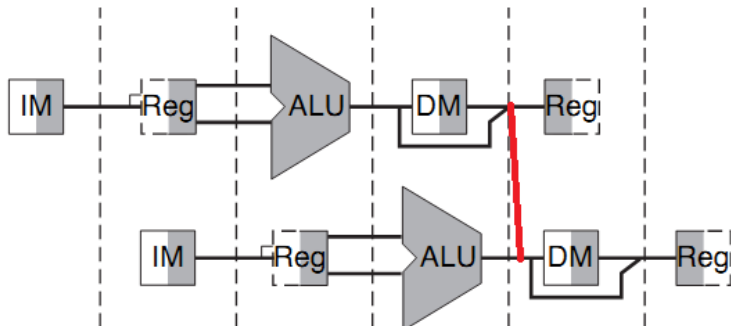
- MEM copy问题: “RISC中, mem-to-mem拷贝频繁”
 - 由lw-sw实现: “loads immediately followed by stores”
 - 如: lw \$1, addr1; sw \$1, addr2



- MEM段旁路: MEM/WB => MEM

- 判据, 数据通路?
- 无需插入bubble!

```
ld x2, 100(x1)
sd x2, 110(x1)
or x13, x6 x2
add x14, x2, x2
sd x15, 100(x2)
```



lw-sw前推?

$$\text{if (EX/MEM.MemWrite and MEM/WB.MemtoReg}$$

$$\text{and (EX/MEM.RegisterRt \neq 0)}$$

$$\text{and (MEM/WB.RegisterRd = EX/MEM.RegisterRt)) ForwardMEM = 1}$$

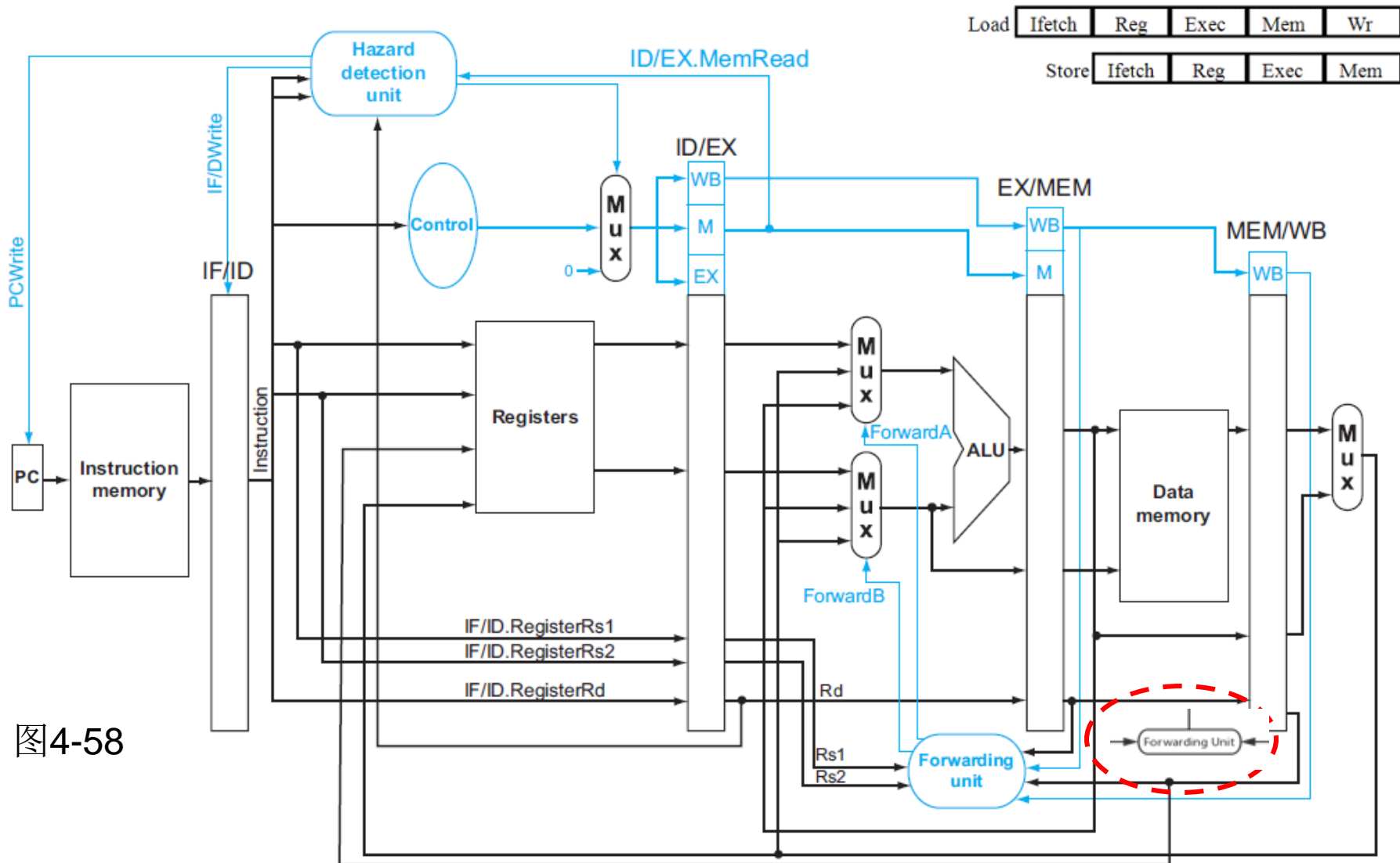


图4-58

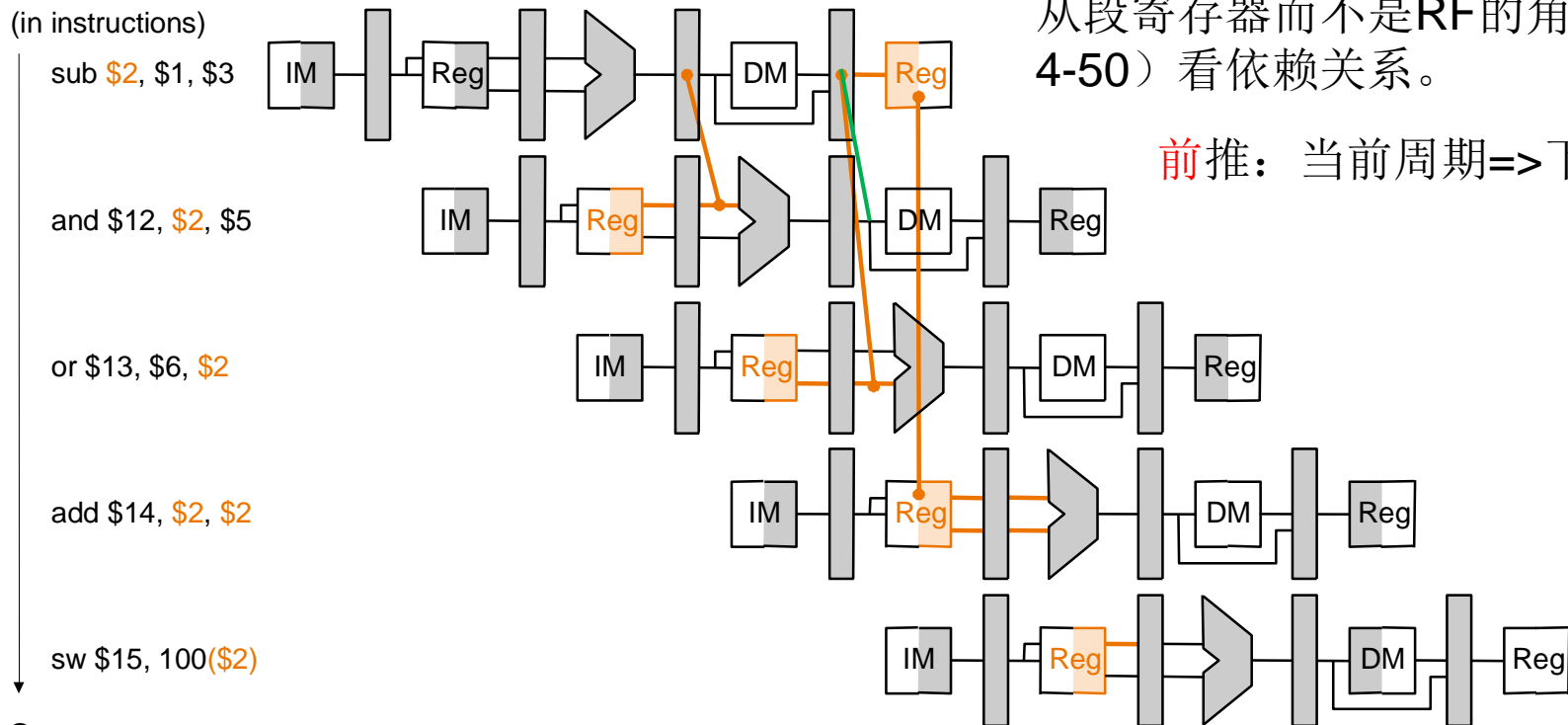
Multilevel Bypass: 需要多个段前推!



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

Program execution order (in instructions)



COD4图4-53（类COD5图4-51），从段寄存器而不是RF的角度（图4-50）看依赖关系。

前推：当前周期=>下一周期

\$4.5.2

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！ 30

forwarding无法解决的RAW: backwards

Time (in clock cycles) →
CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9

Program execution order (in instructions)

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

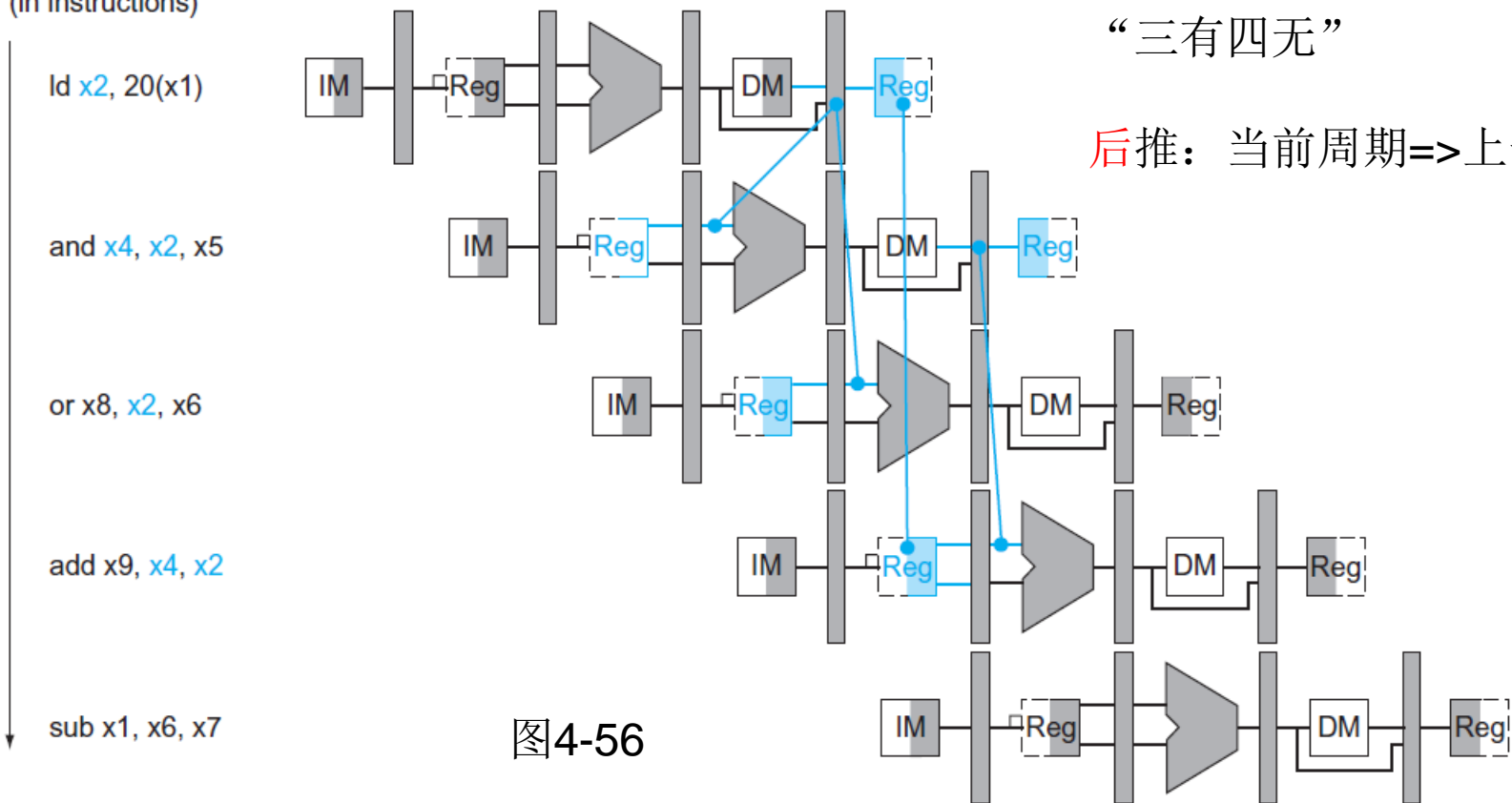


图4-56

load-use-data hazard, §4.5.2

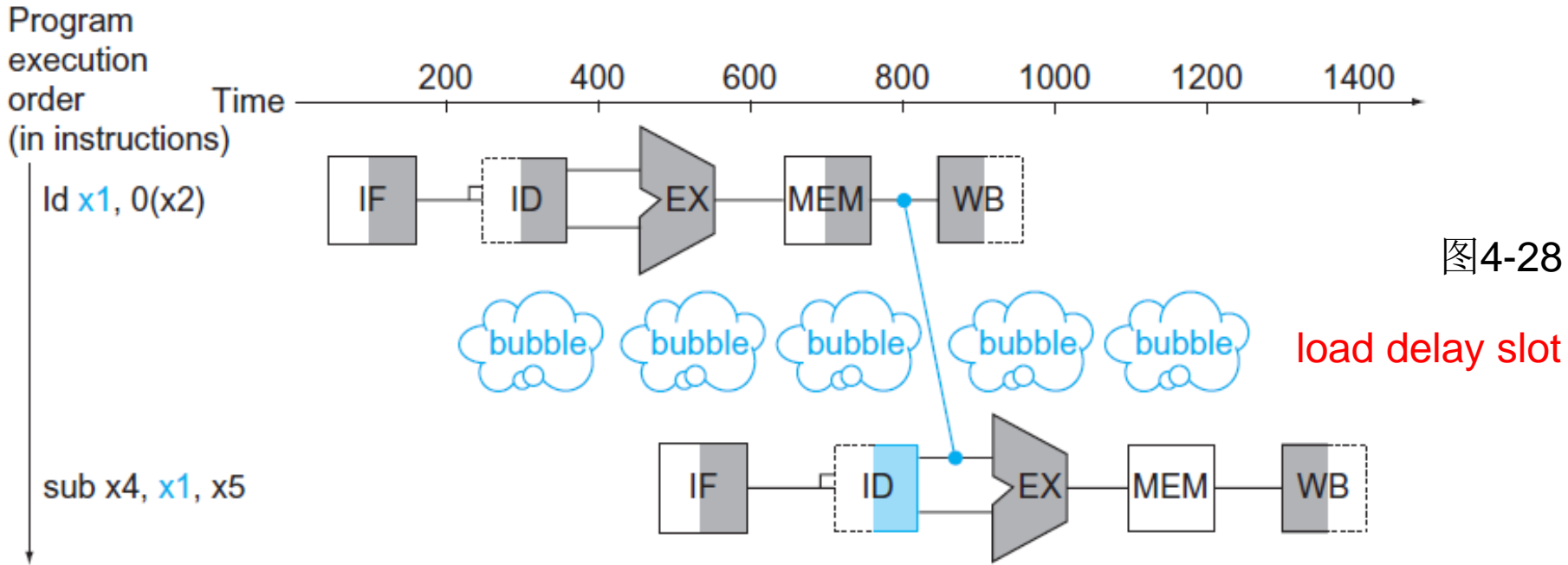
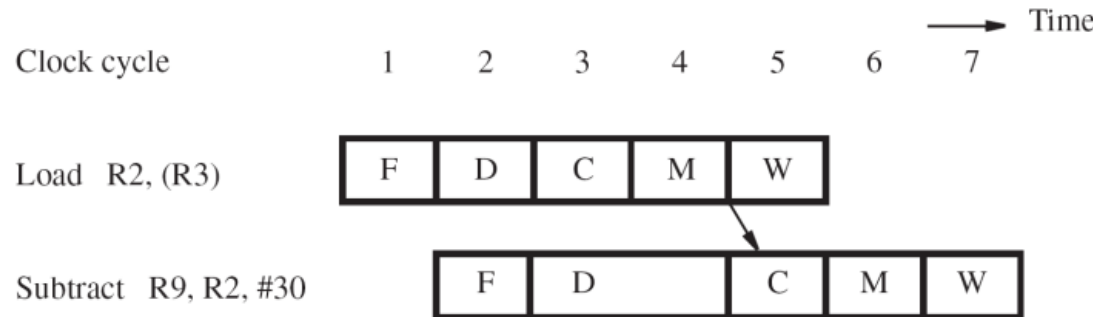


图4-28

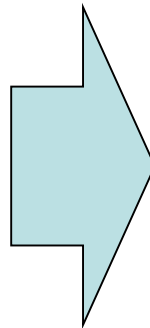
- 编译法：插入nop (= sll \$0, \$0, 0) ，或指令重排序
- 硬件法： **Interlocking**
 - stall: **stretch & delay**
 - FW: MEM/WB->EX
 - 判据？



Code Scheduling: 编译, §4.5.2

- 指令重排序: 程序序 \neq 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, \$4.7

Time (in clock cycles) →
 CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9 CC 10

Program execution order (in instructions)

ld x2, 20(x1)

and becomes nop

and x4, x2, x5

or x8, x2, x6

add x9, x4, x2

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

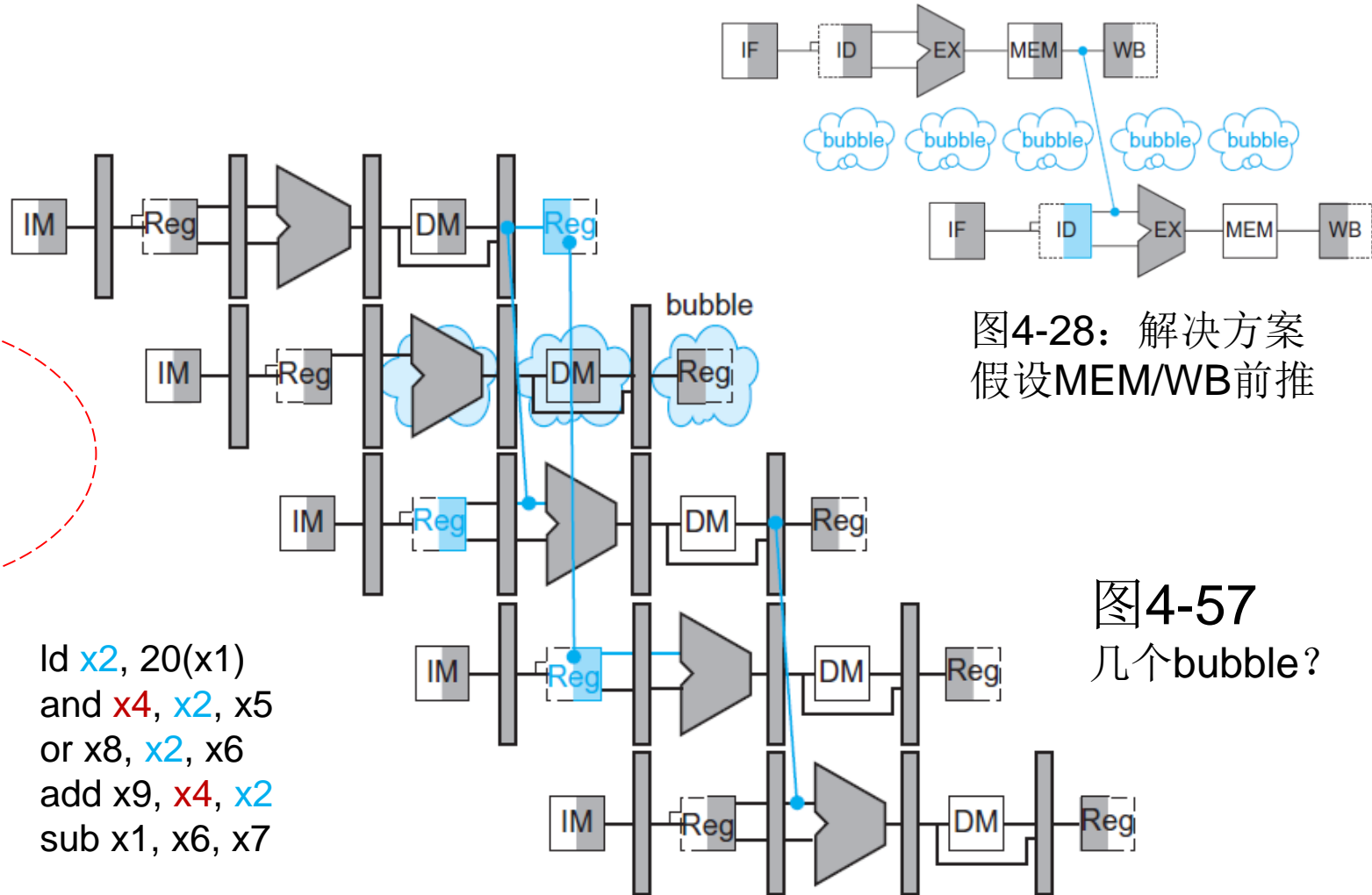


图4-28: 解决方案
 假设MEM/WB前推

图4-57
 几个bubble?

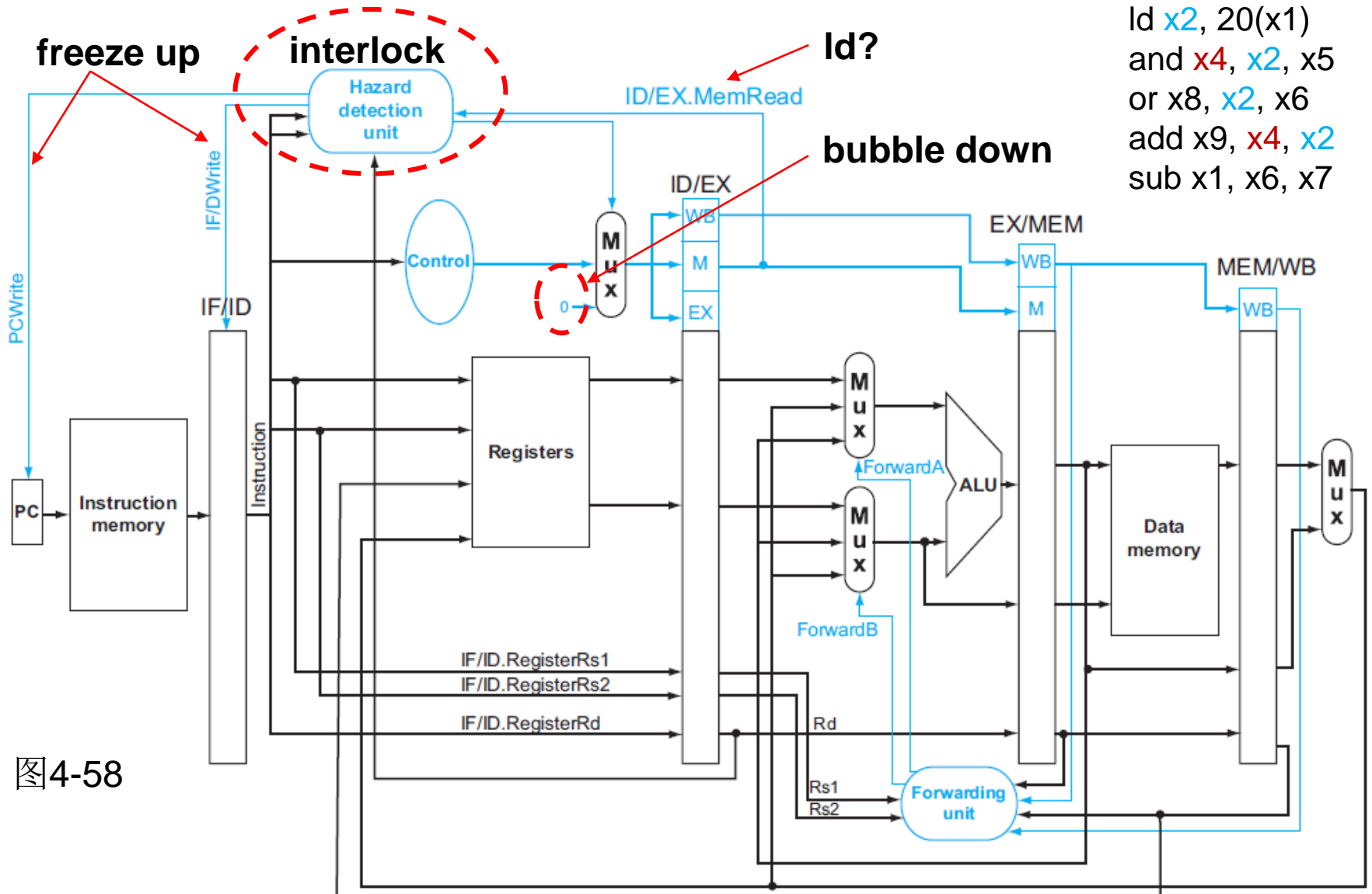
Interlock: load-use-data stalling

- We can stall the pipeline by keeping an instruction in the same stage
- Stall语义: freeze up and bubble down
 - 冻结其前面的所有流水段: 使之重复操作, IF, ID
 - 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: “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
    
```

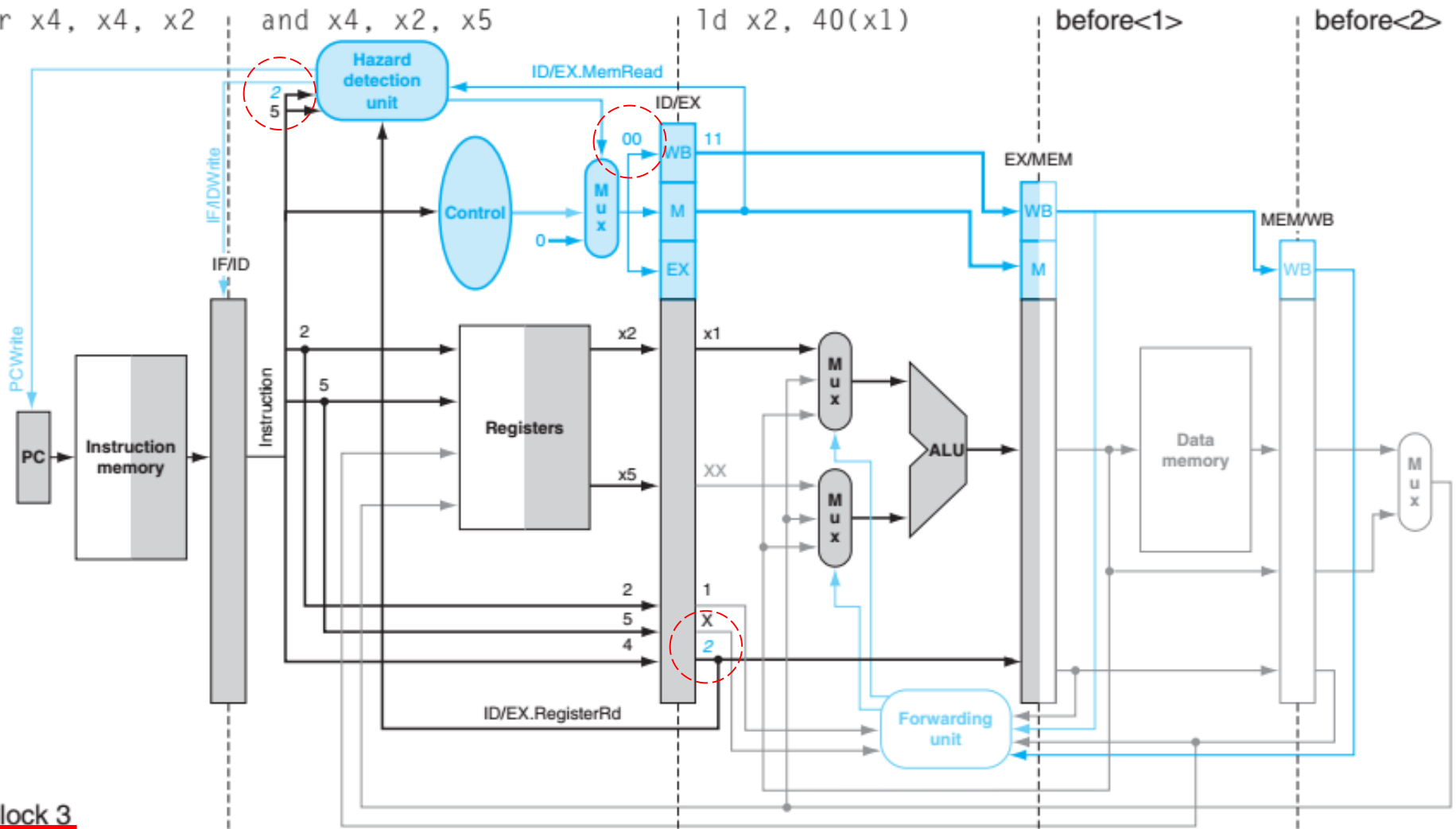


Id x2, 20(x1)
 and x4, x2, x5
 or x8, x2, x6
 add x9, x4, x2
 sub x1, x6, x7

图4-58

load-used, single-cycle diagram

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



Clock 3

\$4.13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
```

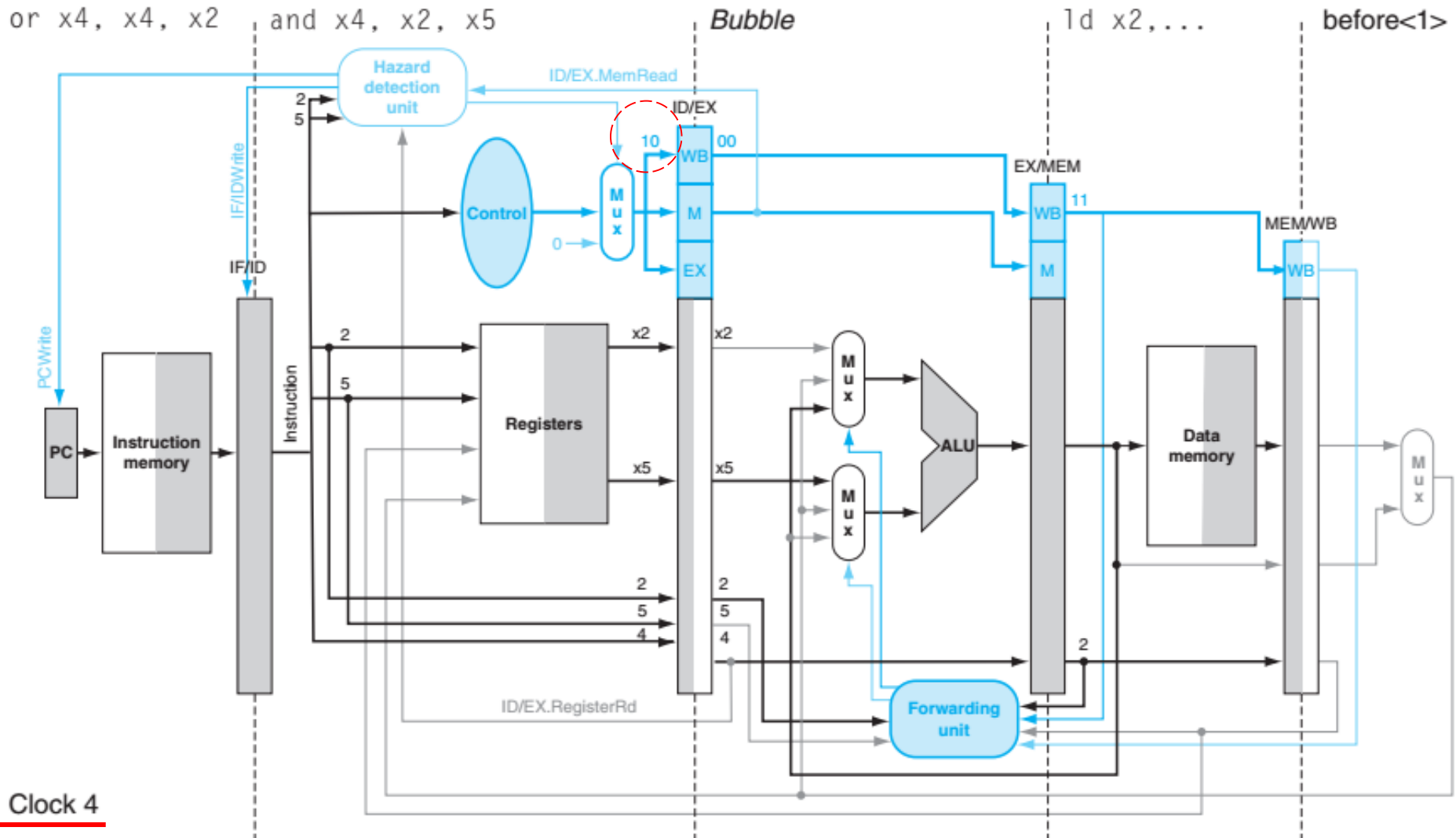


FIGURE e4.13.19: **and**和**or**重复, EX段bubble。注意, 此时FW也工作, 但无害。

load-used, single-cycle diagram

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

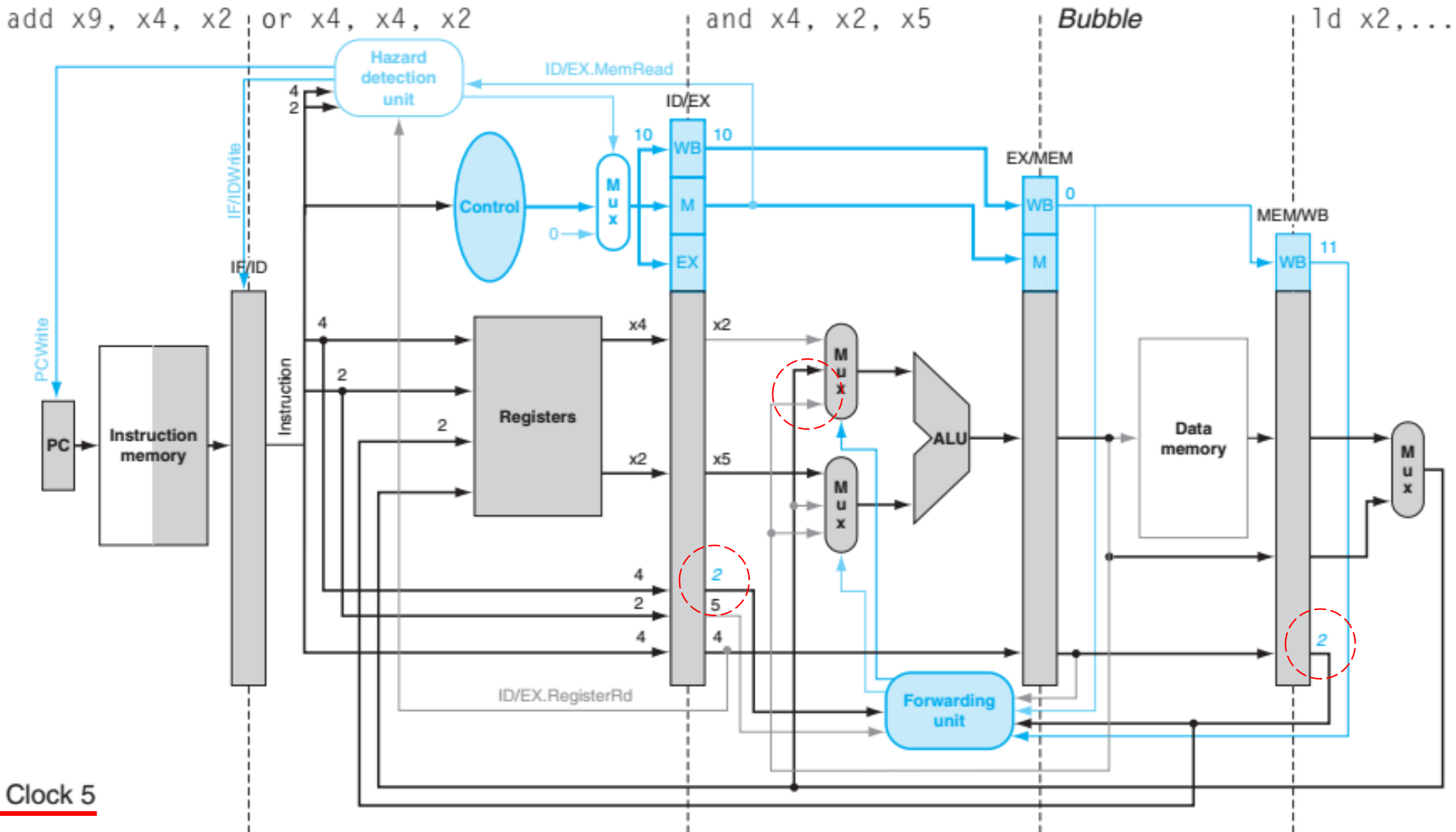
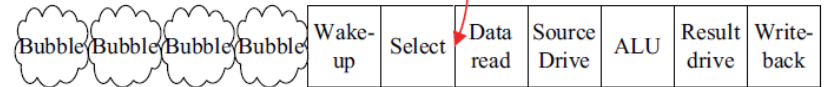
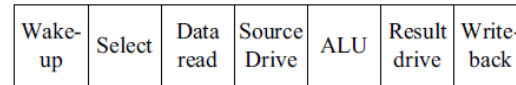


FIGURE e4.13.19: `and`进入EX, 需FWU从MEM/WB前推x2。此周期ld写回(先写), 故or无需FW x2

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：基本流水线模型中已经消除，无须stall
 - MEM单一读写端口：IF，MEM
 - RF单一写端口：lw-Rtype
 - 胡伟武：“RAW阻塞可能造成IF或ID结构冲突”！
 - RAW依赖：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：向EX段前推，EX/ME，ME/WB
 - load-used-hazard: 向EX段前推，ME/WB
 - Store-hazard: 向ME段前推，ME/WB

作业



- 思考
 - 流水线深好浅好？
 - 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