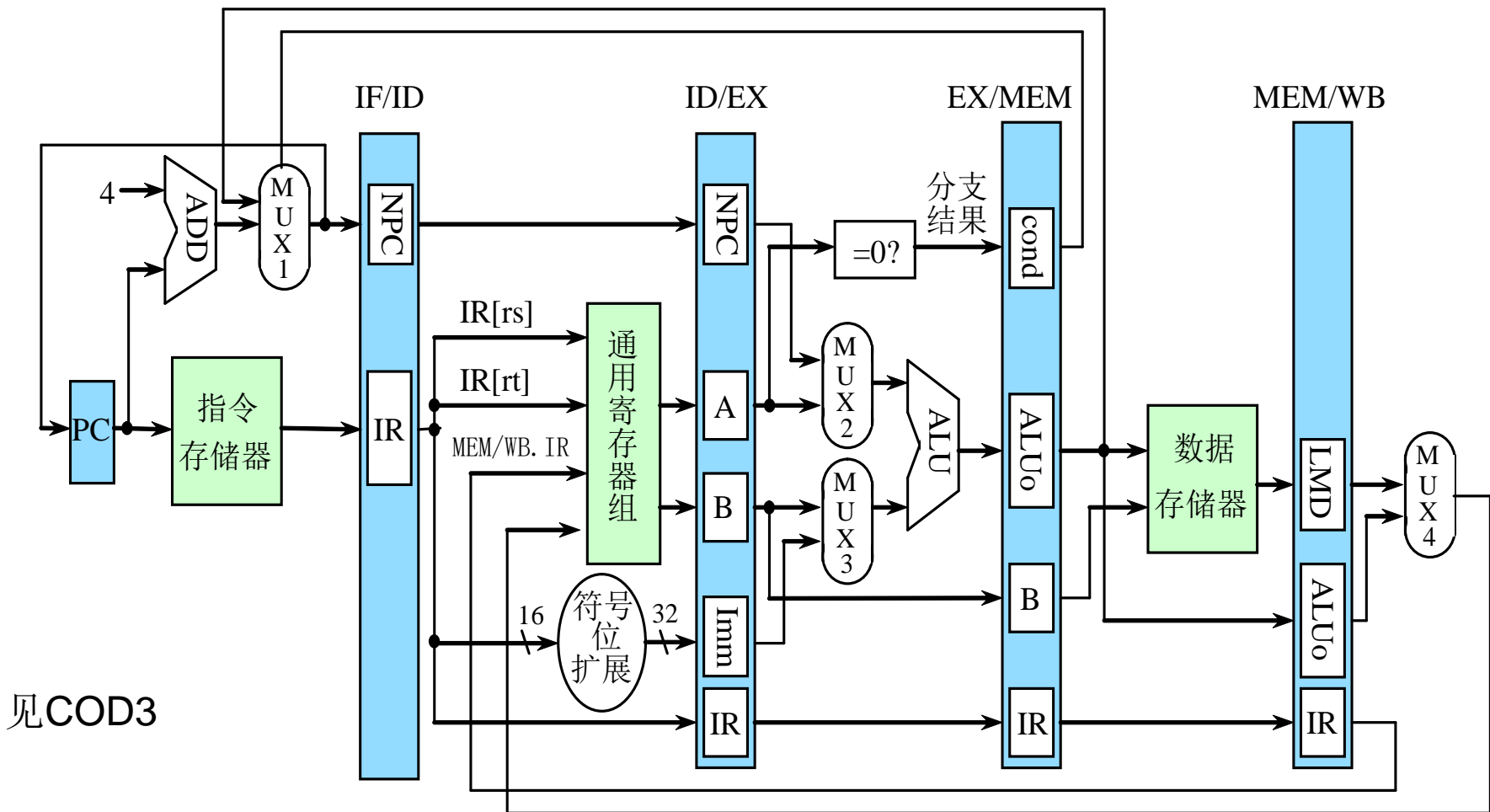


The RISC-V Processor Implementation: Pipeline——ILP

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

流水线技术 “5要素”

Instruction number	1	2	3	4	5	6
Instruction i	IF	ID	EX	MEM	WB	
Instruction $i+1$		IF	ID	EX	MEM	WB
Instruction $i+2$			IF	ID	EX	MEM
Instruction $i+3$				IF	ID	EX
Instruction $i+4$					IF	ID



- 并行【独占，暂存，背包】，同步

面向流水线的ISA: 4特点 (\$4.6.1)

1. 指令字等长

- 适于取指和译码

2. 指令格式简单规则

- 操作数位置固定: 可以在译码的同时取opr, 否则要增加一级流水段?

3. 只有load/store访存

- 可以在执行阶段计算访存地址, 然后在下一阶段访存。
- 如果R-type指令也可访存?

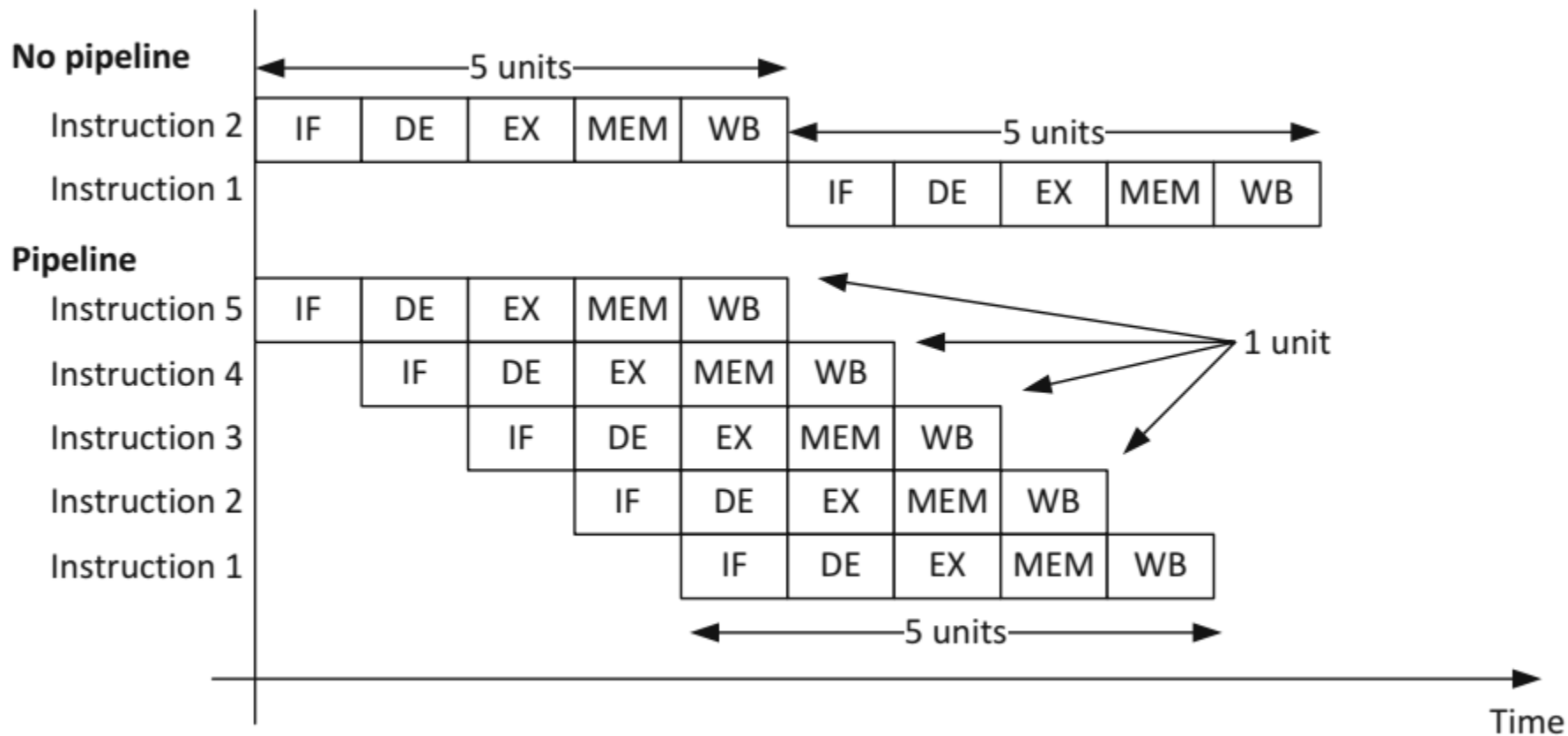
4. 每条指令最多只写一个结果, 且在最后一级进行 (WB)

- load-Rtype竞争 (race, RF端口结构冲突)
- 利于前推 (\$4.6.2)
- 利于异常处理 (\$5.7.9): 缺页异常须指令可重启
- 寻址方式简单: 间址寻址、堆栈寻址?
- 操作数在内存中“字对齐”——COD MIPS版, RV/ARM无?
 - 因此在取操作数时不需要访问存储器多次

Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

理想流水线：贪心、投机

$$\text{speedup} = \frac{\text{非流水线执行时间}}{\text{流水线执行时间}} = \lim_{n \rightarrow \infty} \frac{n * k * \Delta t}{k * \Delta t + (n - 1) * \Delta t} = k$$



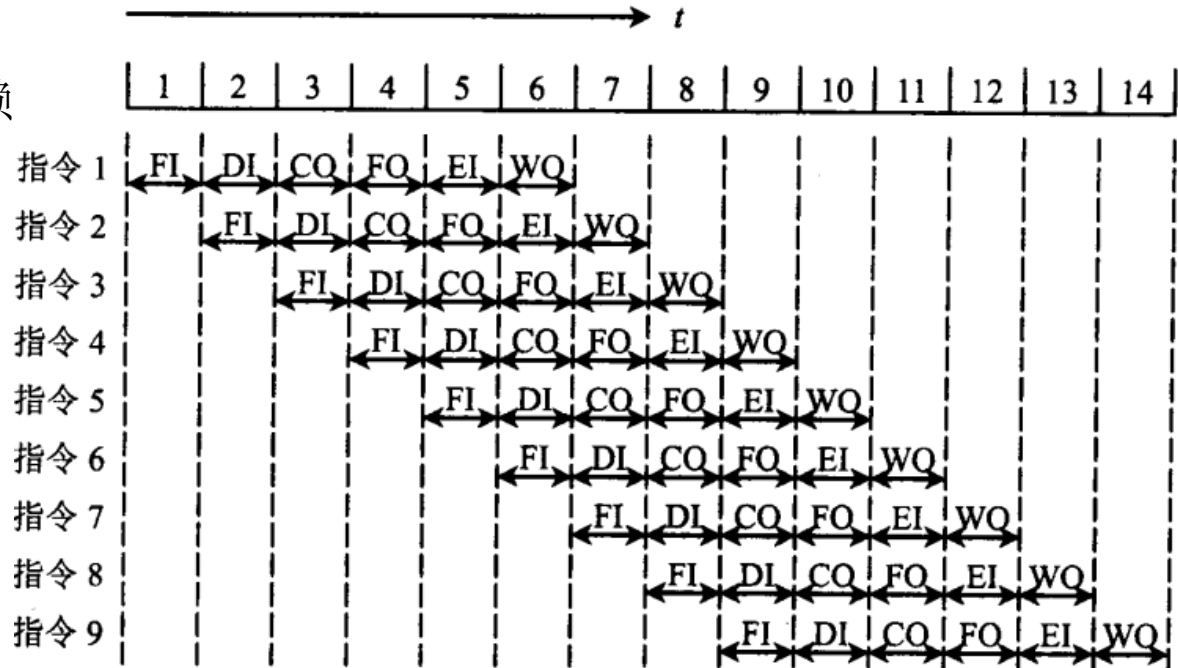
CPI ≥ 1 , IPC ≥ 1 ?

Pipelining Idealism:

平衡, 重复, 无冲突和依赖

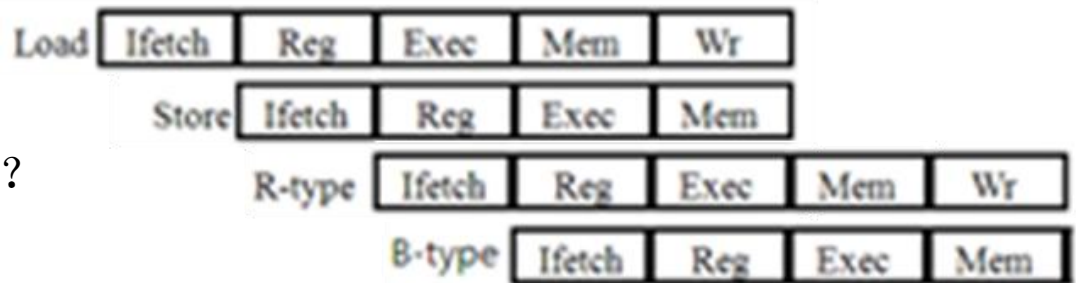
Speedup = k

k越大越好?



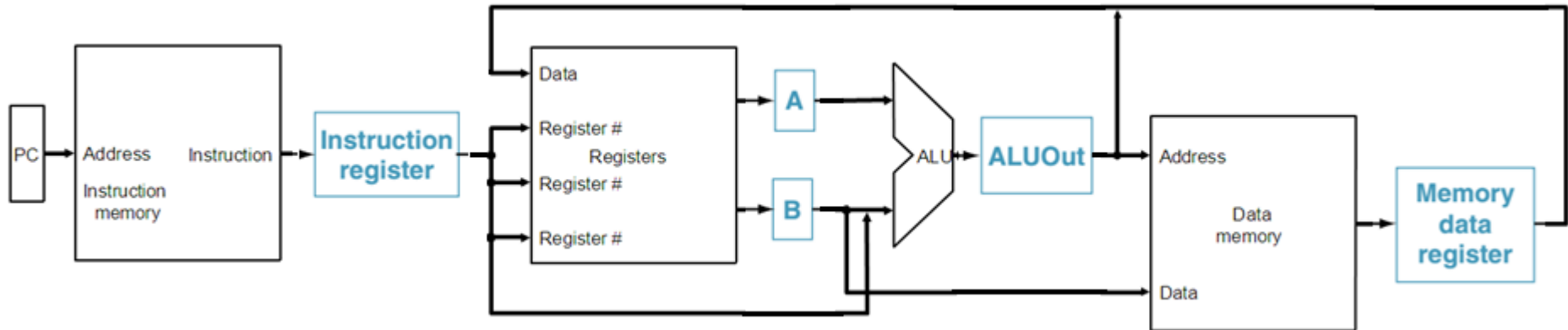
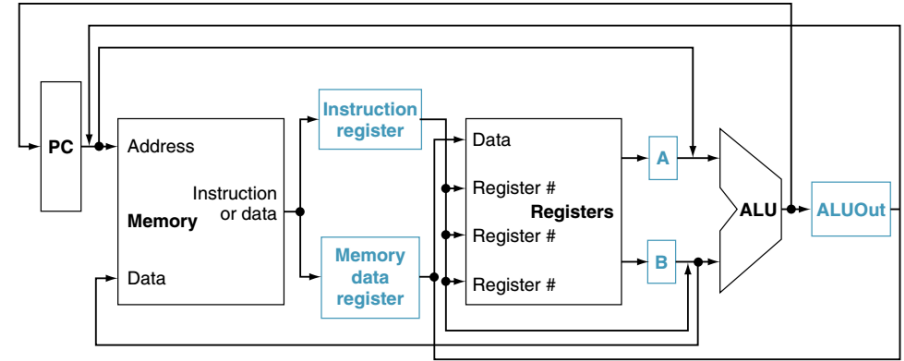
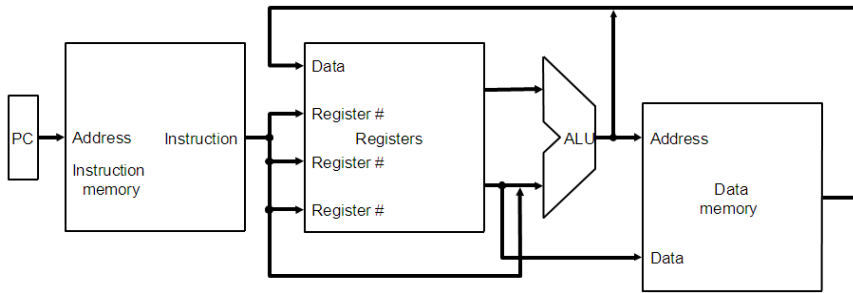
p211:

“任何指令都要经过每个流水段”?



流水线数据通路

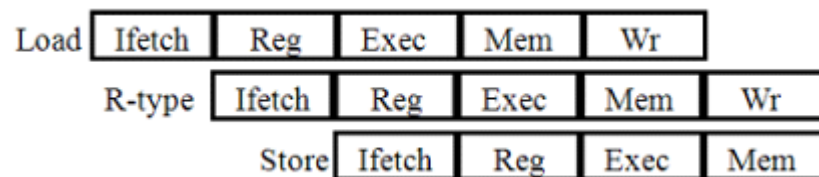
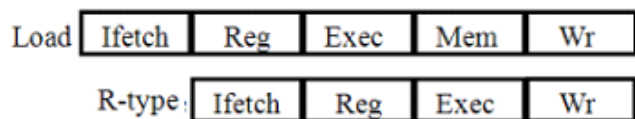
Instruction number	1	2	3	4	5	6
Instruction i	IF	ID	EX	MEM	WB	
Instruction $i+1$		IF	ID	EX	MEM	WB
Instruction $i+2$			IF	ID	EX	MEM
Instruction $i+3$				IF	ID	EX
Instruction $i+4$					IF	ID



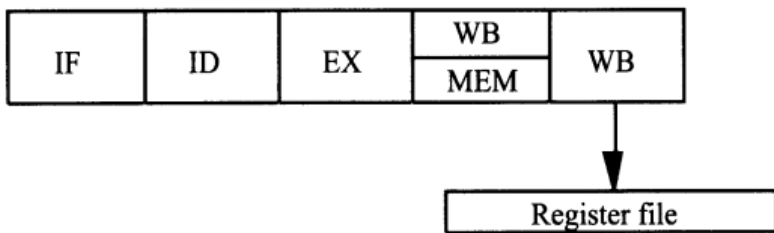
- IF、MEM并行

lw-Rtype指令结果写RF端口冲突消除

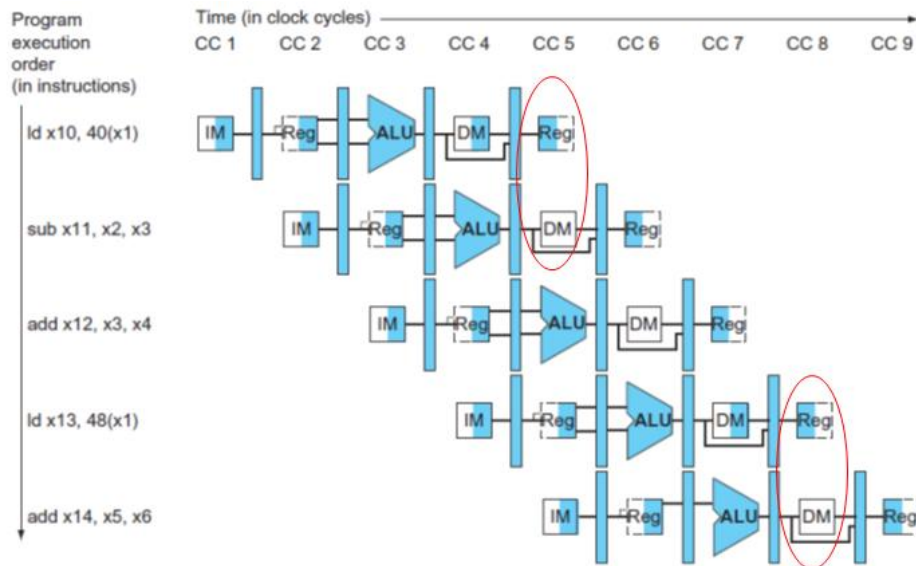
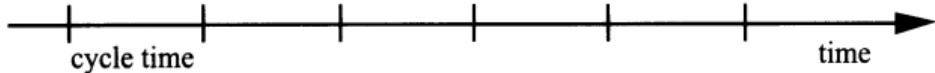
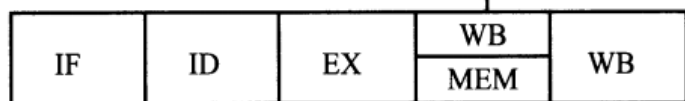
- 流水线RF写约定: RF “两读一写”
 - “每条指令最多只写一个结果, 且在最后一级进行 (WB)”



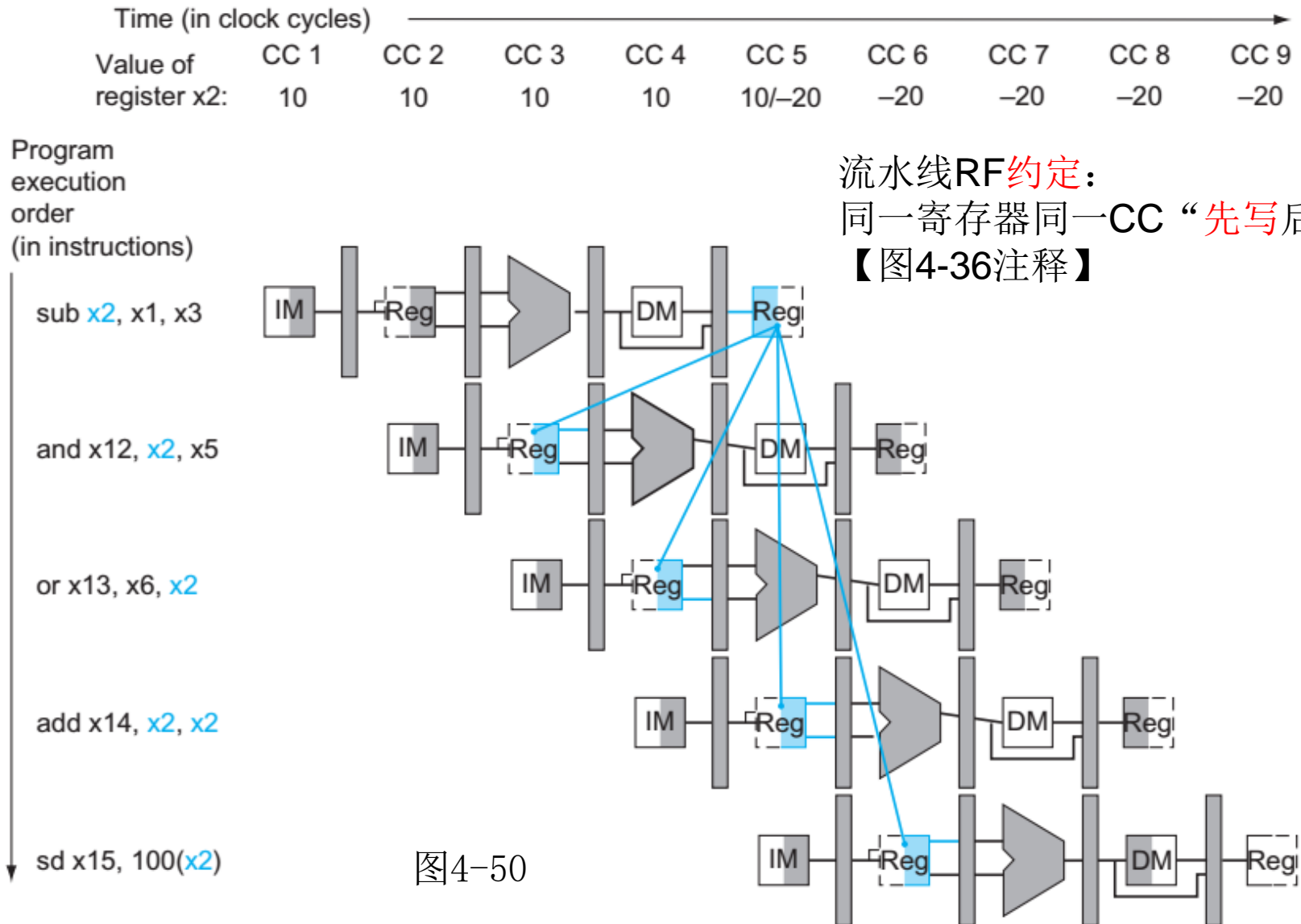
load Reg2,A



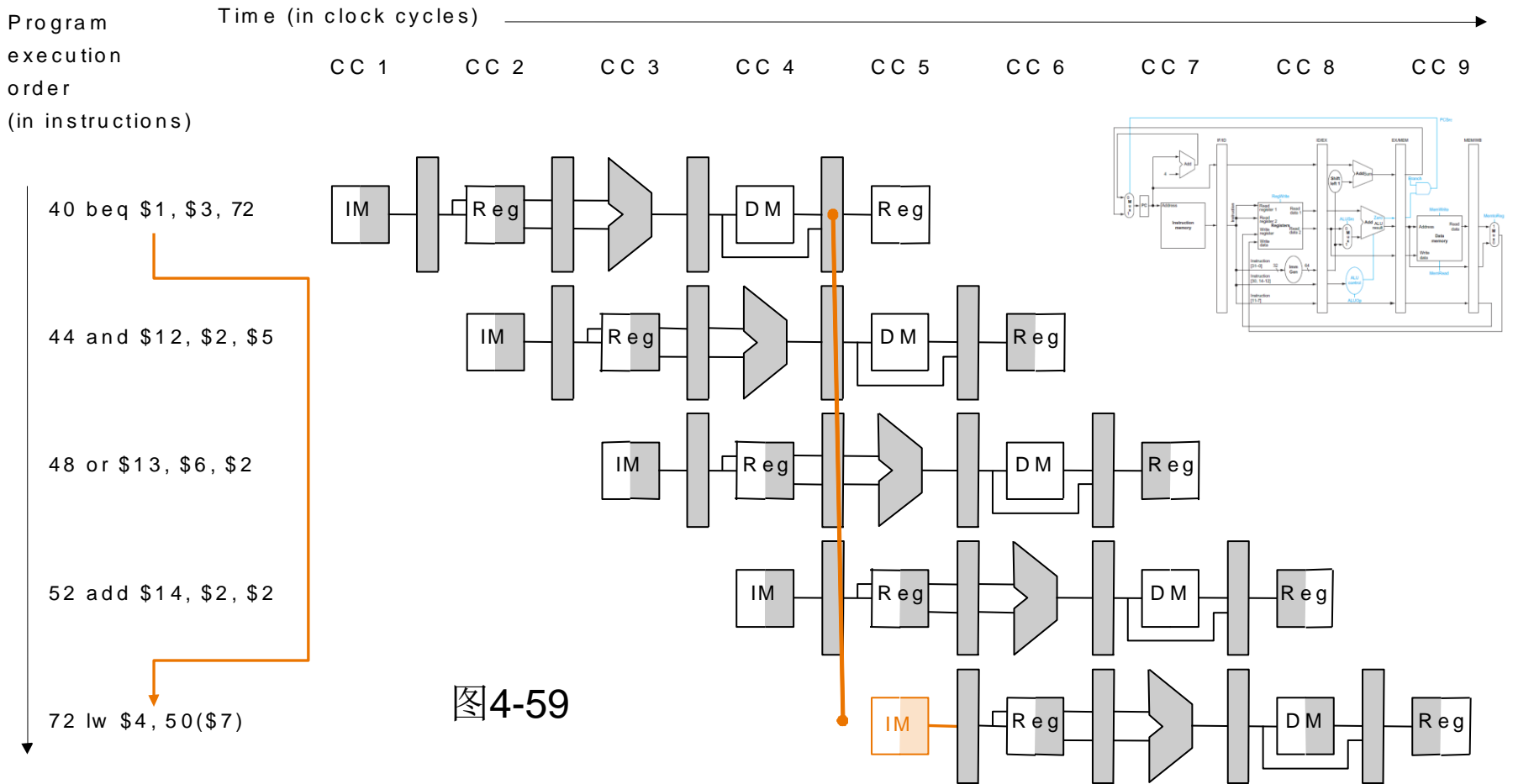
mul Reg3,Reg4,Reg5



哪个周期数据可用？

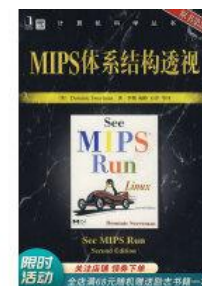
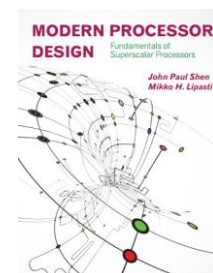
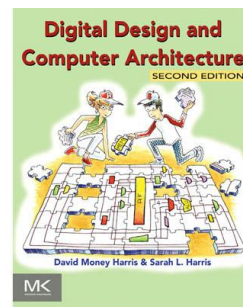


beq taken?

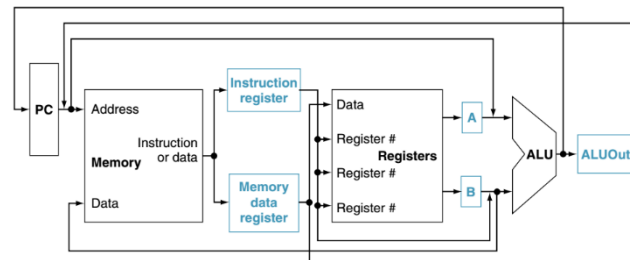


内容提要

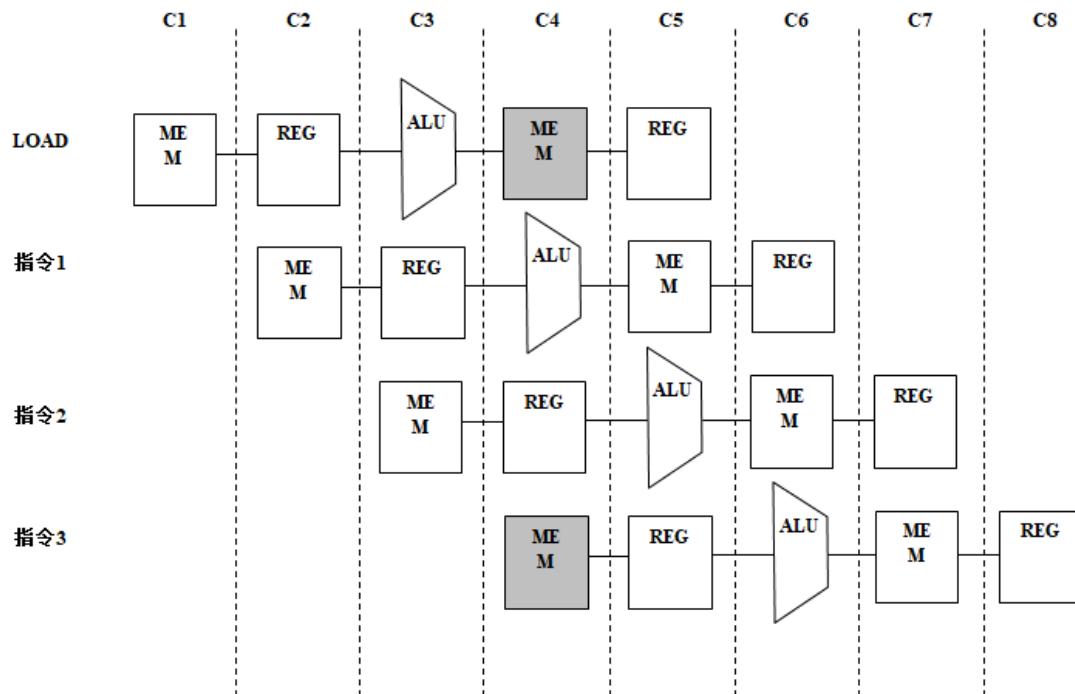
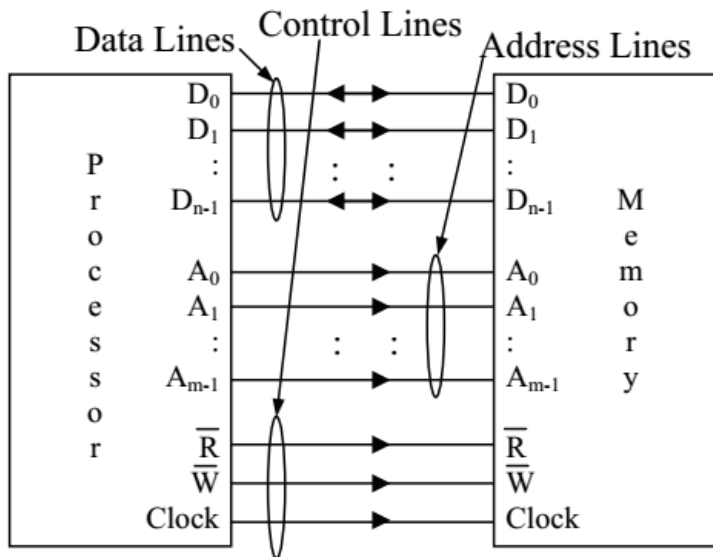
- 流水线技术原理：4.6
- RV的五级流水线实现：4.6.1, 4.7
 - Verilog行为级定义：4.14
- Hazard问题：4.6.2
 - 结构冲突：哈佛结构
 - 数据依赖：4.8
 - 编译技术：插入nop, 指令重排, 寄存器重命名
 - forwarding技术：RAW
 - Interlock技术：Stall
 - 控制相关：4.9
 - 编译技术：延迟分支
 - 硬件优化：提前完成, 投机, 预测
 - 异常：4.10
- 多发射技术：4.11
- 硬件多线程：6.4
- 多核：6.5



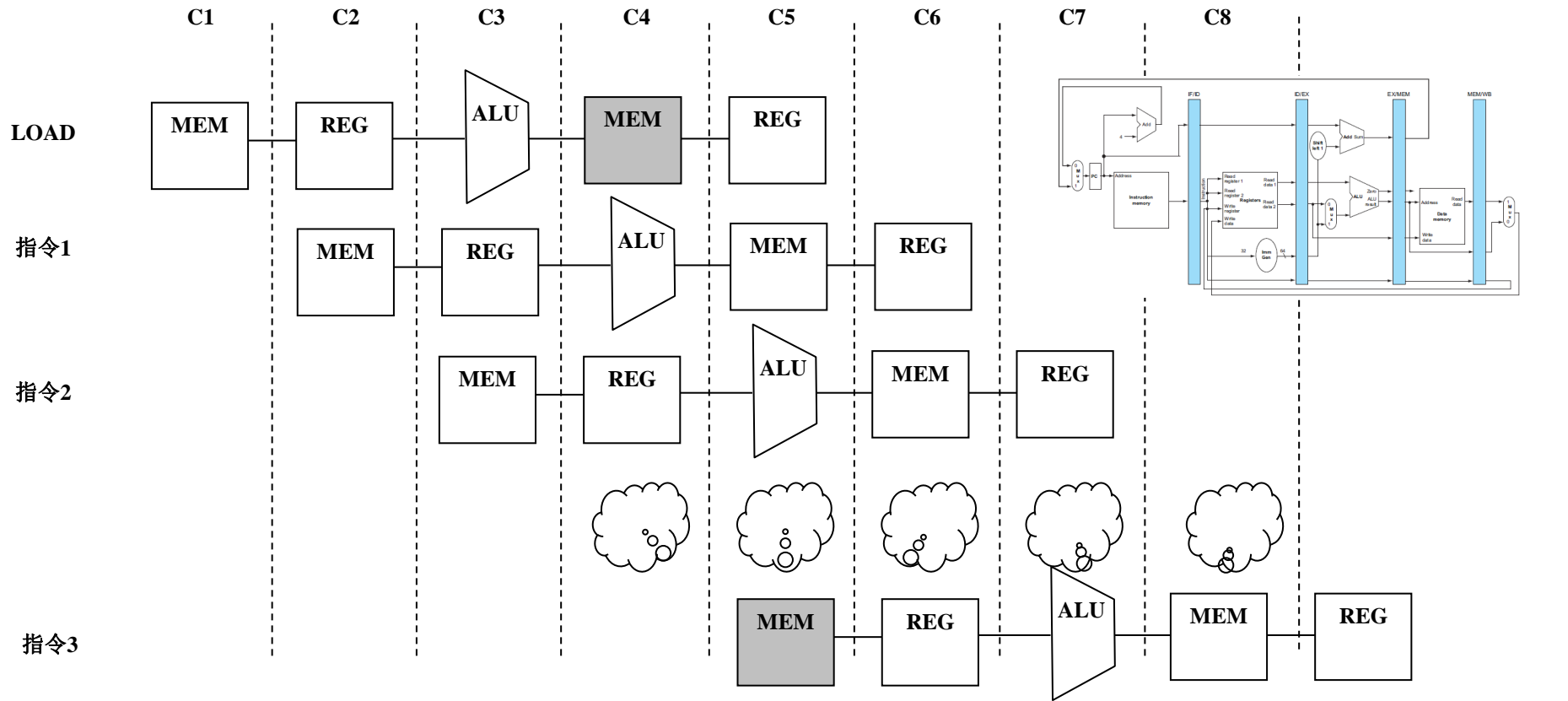
存储器结构相关



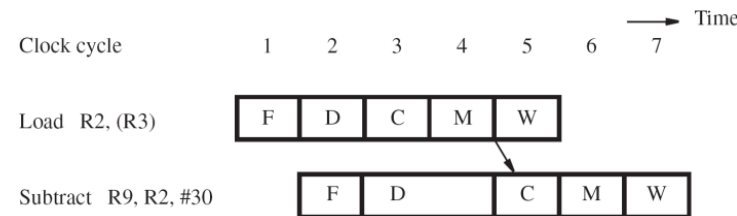
- 假设：指令和数据都存放在同一个存储器中（合体），且为单端口存储器【每次只能执行一个读或一个写】
 - 则同一周期（CC4）内取指与访存并发，存在冲突
 - 无法并行：排队stall【谁stall？】
 - 并行：分体，独占



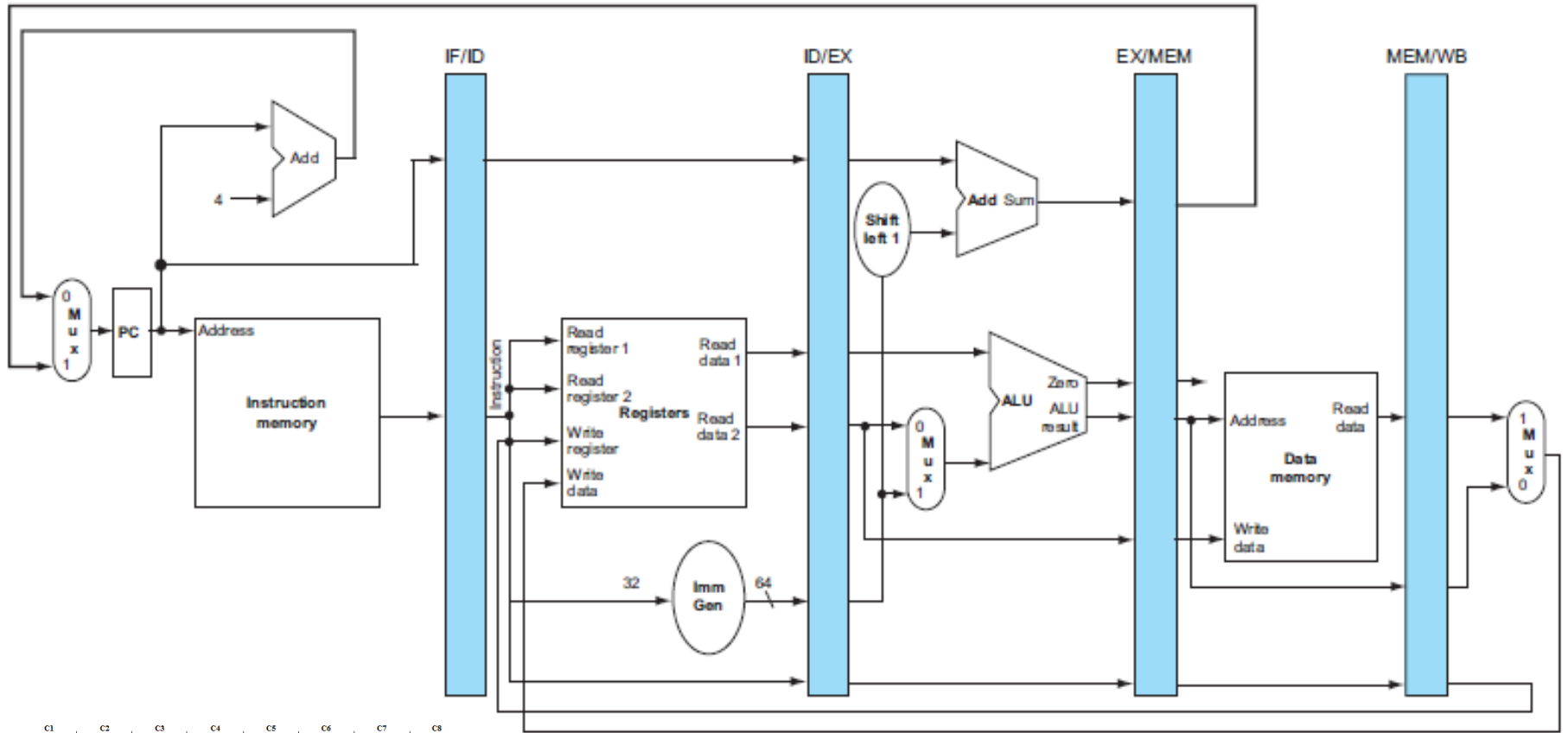
流水线的stall语义（停顿，阻塞）



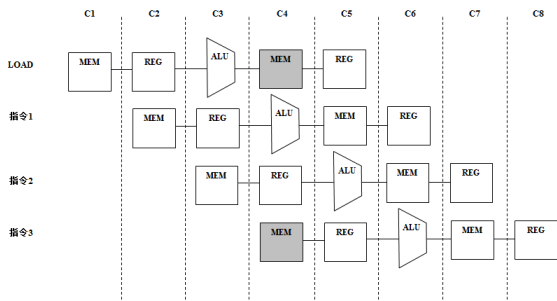
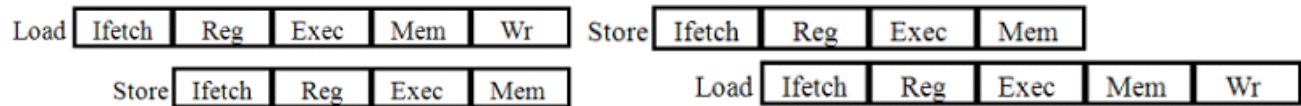
- 程序行为：等价于插入一条nop
- ppl行为：stretch & delay，使流水线停顿一拍
 - “freeze up & bubble down”：interlock
- 访存结构相关：IF段停顿【谁stall?】
 - PC不变，重复取指；IF/ID清零 = 插入气泡（bubble）



IF与MEM结构冲突消除：哈佛结构

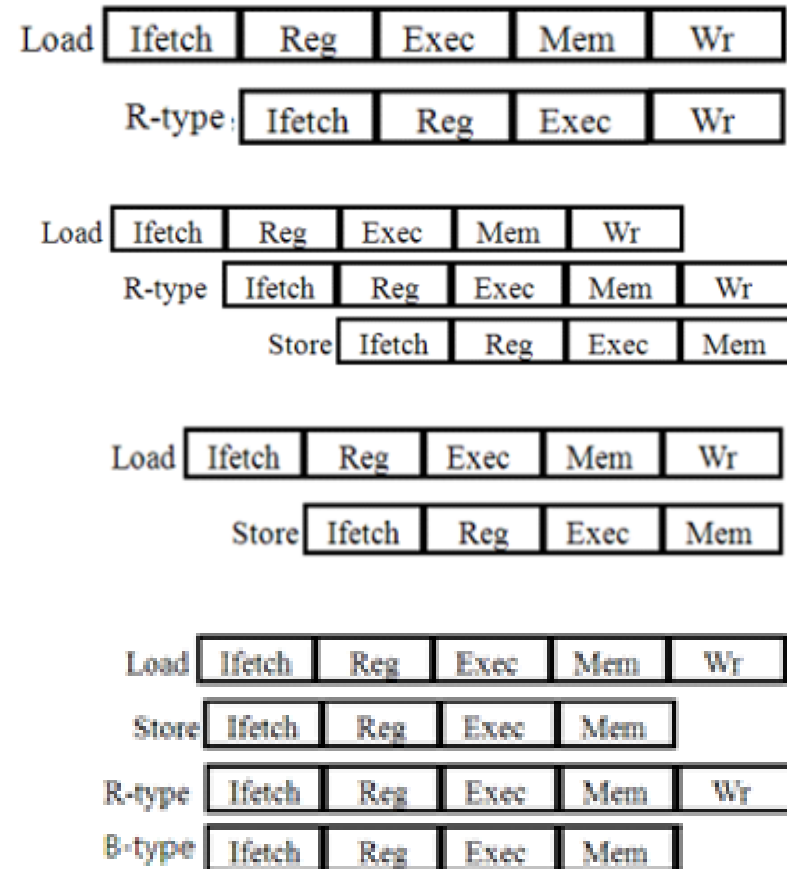


lw/sw存在D\$冲突?



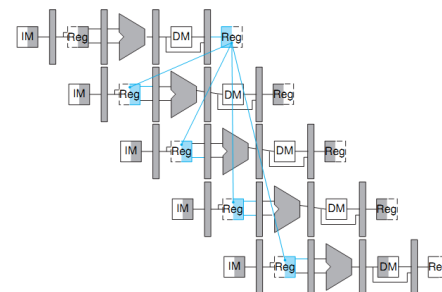
每个CC完成的动作：结构冲突？

- PC：一读一写
 - 先读后写
 - beq, jal/jalr：一次读两次写（实现？）
 - 分支预测器：使PC“一读一写”？
- RF：两读一写
 - 先读后写？
 - 注意：单周期版为先读后写
 - 先写后读：“三有四无”
- Imem：一读
- Dmem：一读/一写
- 段寄存器：一读一写
 - 数据通路寄存器
 - 控制信号寄存器
 - 先读后写，down读up写



数据相关分类

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

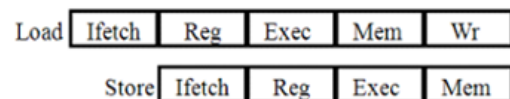
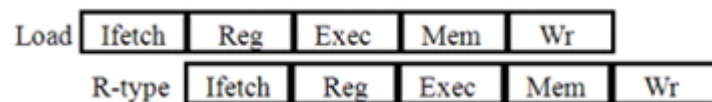


- 写后读 (**RAW**)
 - j 的执行要用到 i 的计算结果，但 j 可能在 i 写入其计算结果之前就先行对保存该结果的寄存器进行读操作。
 - 类型：ID段依赖于EXE或MEM段
- 读后写 (**WAR**)：反依赖anti-dependence
 - j 可能在 i 读取某个源寄存器的内容之前就先对该寄存器进行写操作，导致 i 读出的是错误的数据。
 - 原因：名字复用（寄存器数有限，被迫复用）
- 写后写 (**WAW**)：输出依赖
 - j 和 i 的目的操作数一样，写入顺序错误，在目标寄存器中留下的是 i 的值而不是 j 的值。
- 哪些真会发生？证明？
 - ppl: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

哪些数据依赖会发生？

- 共享资源：Reg依赖，mem依赖
- 程序中存在：RV2中文版p244
 - 真依赖（数据依赖，值依赖）：RAW，生产者-消费者
 - 假依赖（名字依赖）：前后指令不存在逻辑关系，仅地址相同
 - WAR：反相关。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）
 - WAW：输出相关。前后执行写相同地址。须保证PO序
- “简单流水线中不产生WAR/WAW冒险！”，证明？
 - RV2中文版p247；【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

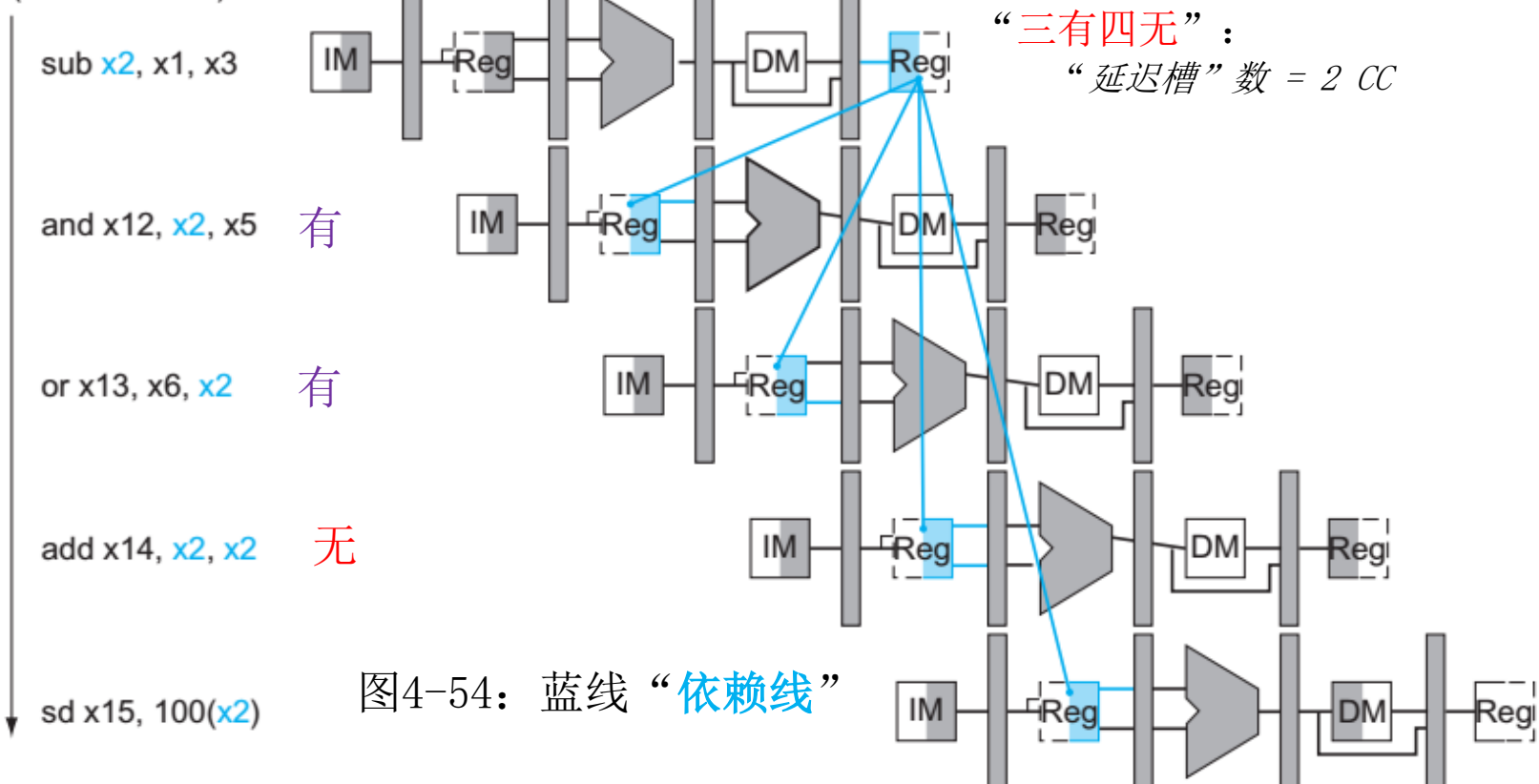
1: x=2;
2: y=x;
3: y=x+z;
4: z=6;



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)

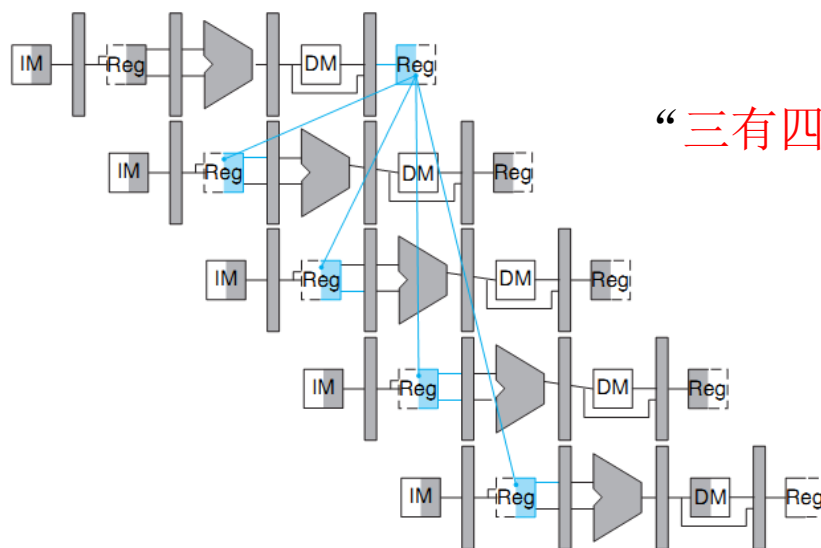
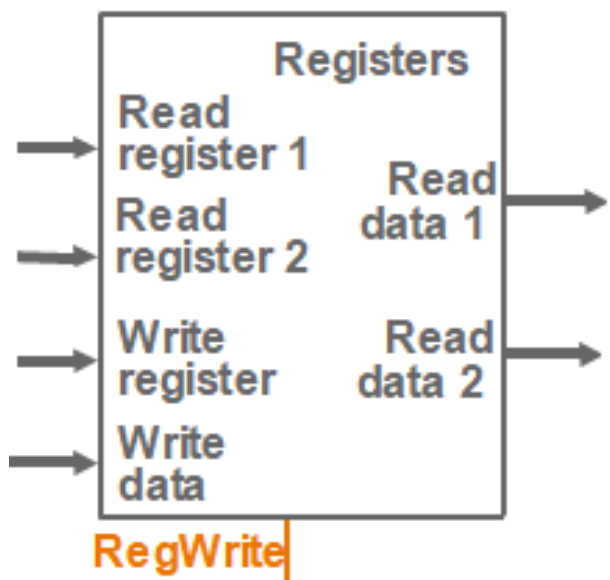


冒险: EXE段, MEM段, WB段
 设寄存器“先写”: cc5新值就可用“内推”, 消除WB段冒险! p224
 “三有四无”:
 “延迟槽”数 = 2 CC

图4-54: 蓝线“依赖线”

RegFile同时读写操作约定

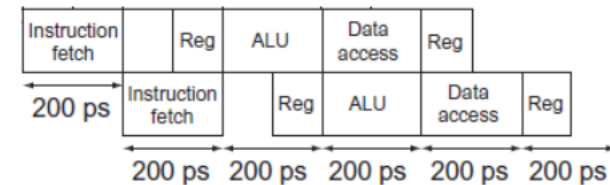
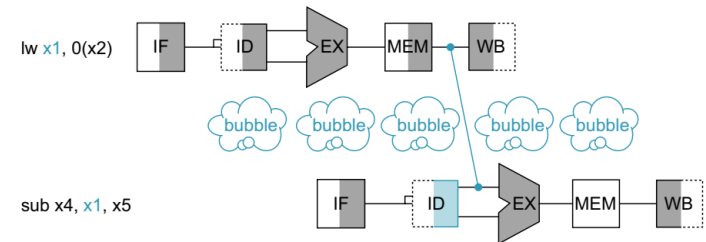
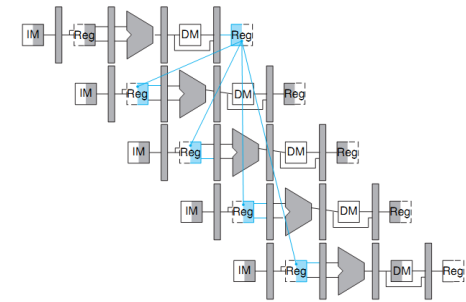
- **不同**寄存器读写：在一个**周期**中，RF支持两读一写
- **同一**寄存器读写：`add $t0, $s2, $t0`
 - 不能**同时**进行读和写：**写控制**RegWrite与clk同步，两种写方式
 - 后写 (late write)：前半周期读，后半周期写。——单周期版？
 - 在一个**周期内**完成读写操作，但读出的是**上一个**周期写入的值。图4-7
 - 先写 (early write)：前半周期写，后半周期读。——流水线版？
 - 在一个**周期内**完成读写操作，读出的是**当前**周期写入的值，如CC5。图4-36, 4-54



“三有四无”

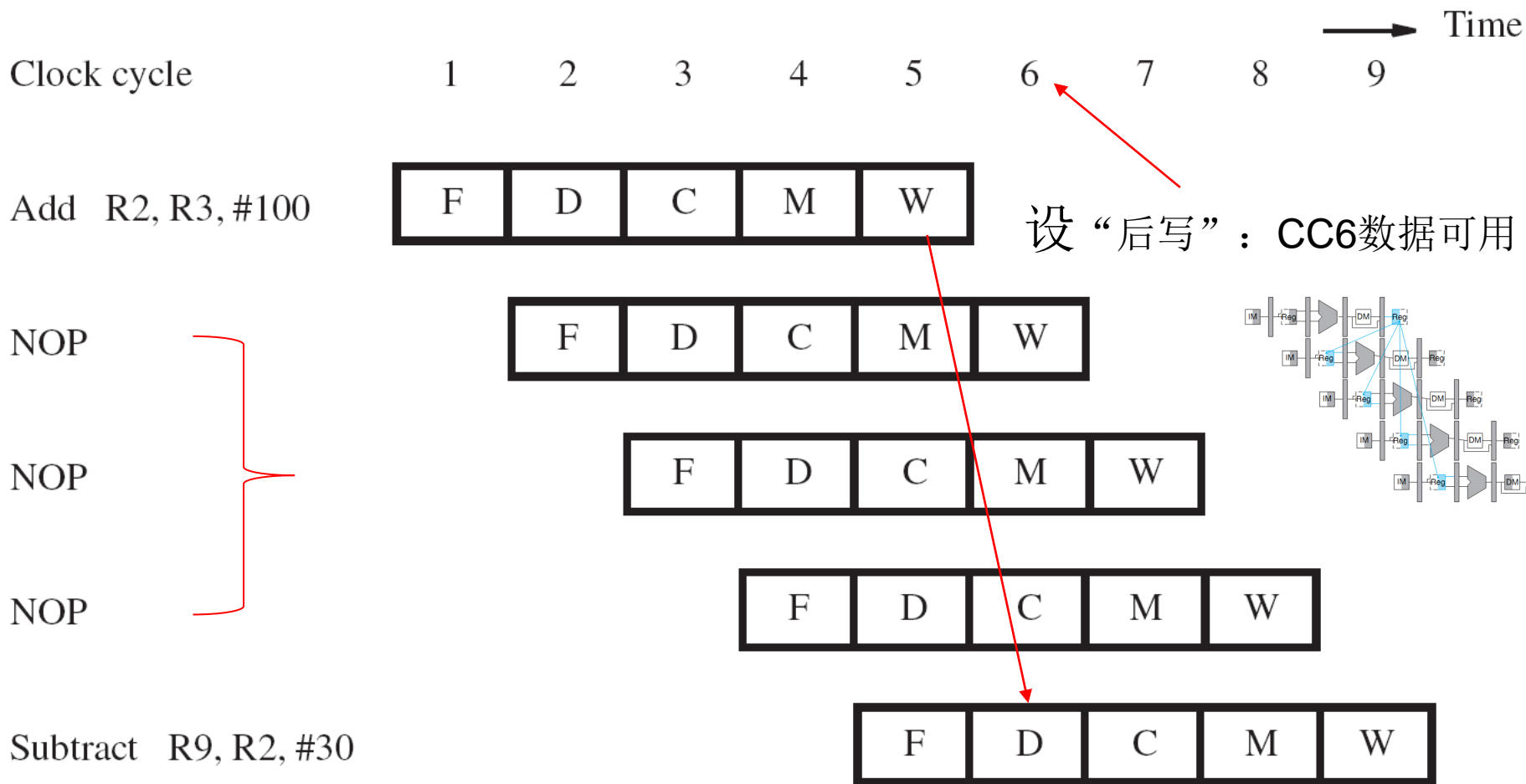
RAW问题：Stall, 类型, 检测

- 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
 - 动态调度：多发射，超标量
 - 数据定向：动态技术【实时检测】
 - 数据前推/旁路
 - Forwarding, 前一条指令直接传递给后面的指令
 - Bypassing, 绕过RF
 - 使关键路径延长，不利于提升主频！



化解RAW: 编译器, \$4.6.2

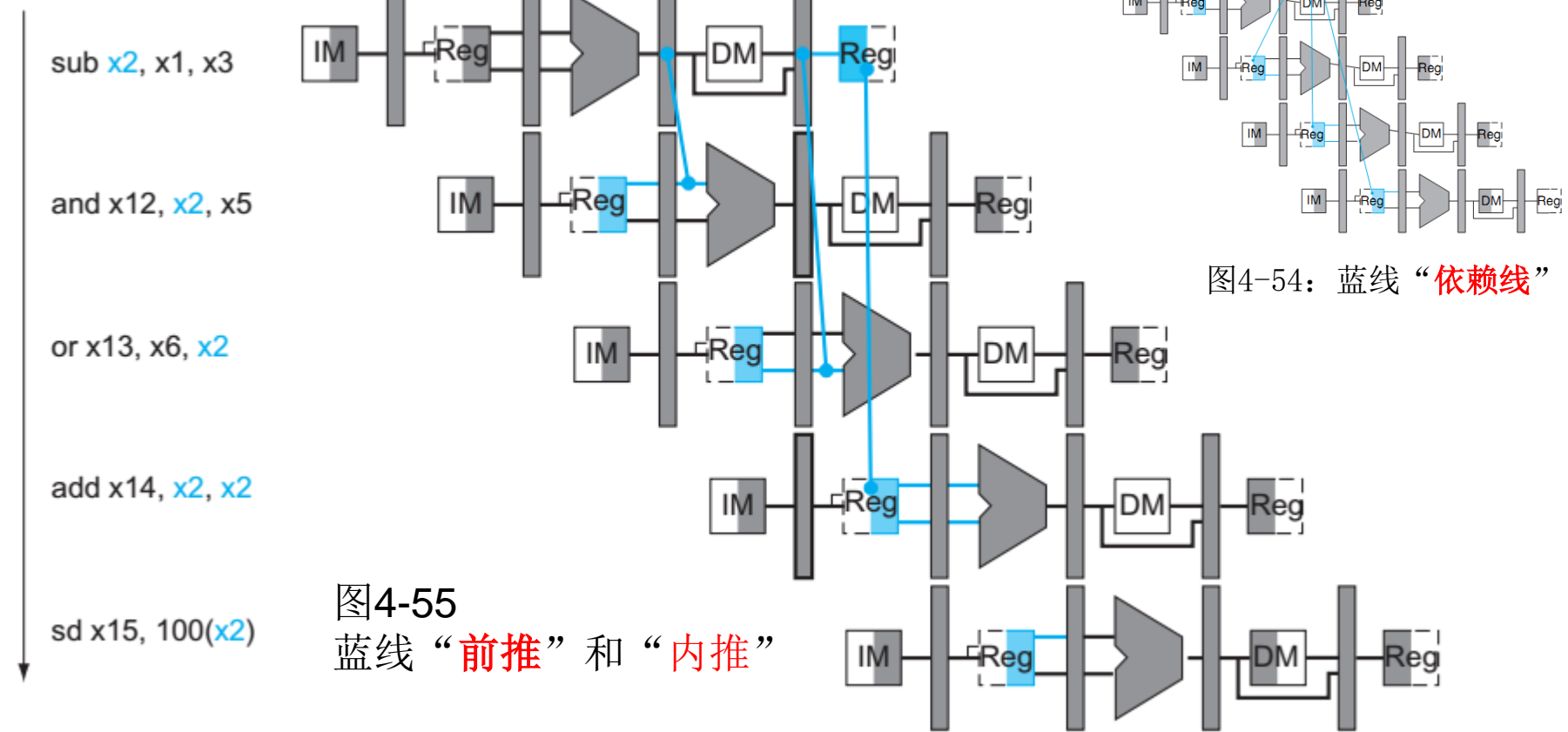
- 指令调度 (注意: 此例RF为“后写”——“4有5无”)



减少RAW损失: Forwarding to EX

- “bypassing the use of RF”

(in instructions)



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)

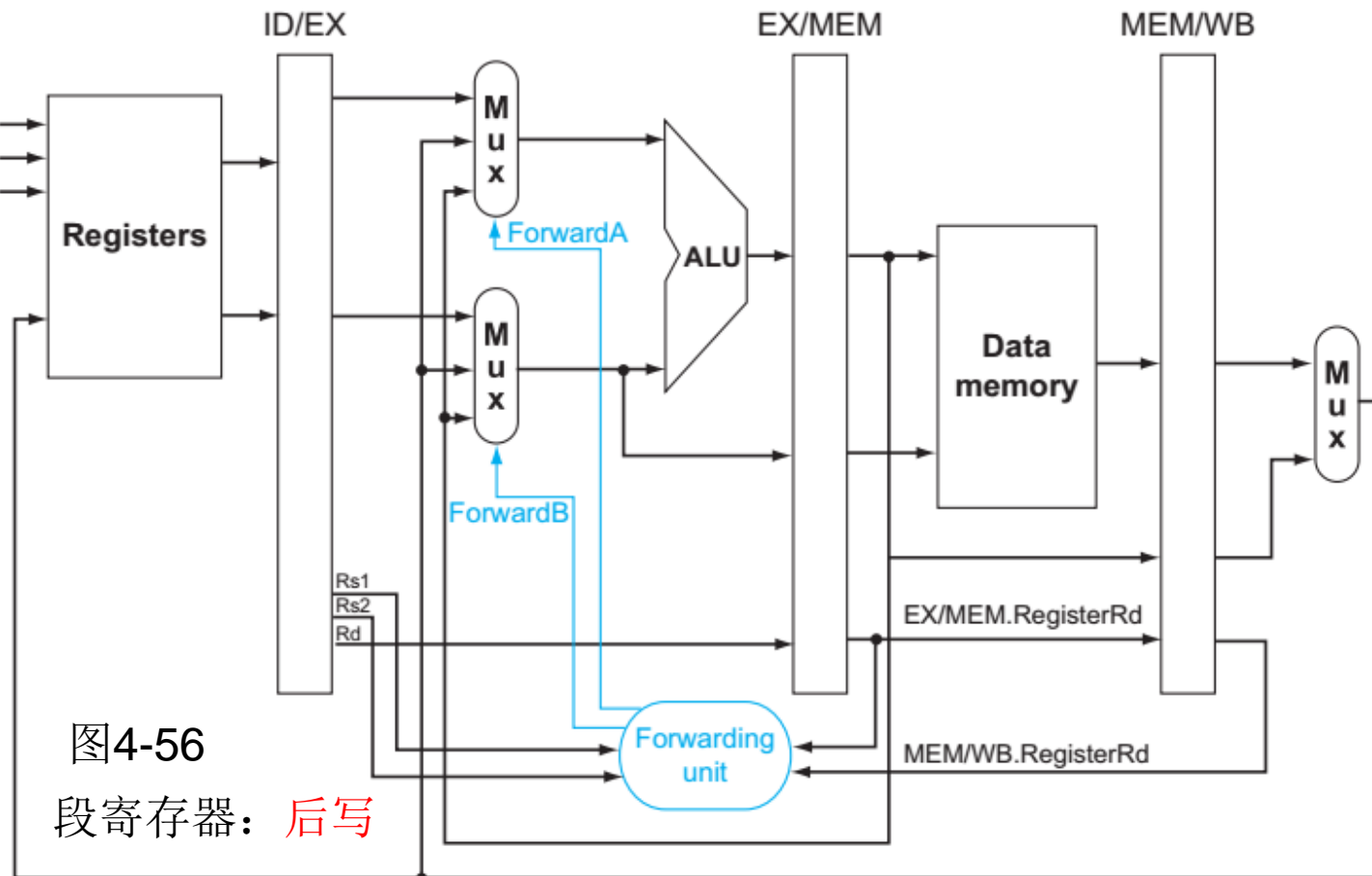
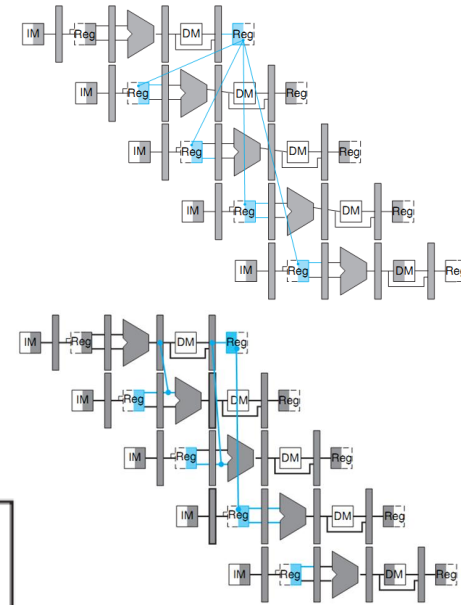


图4-56

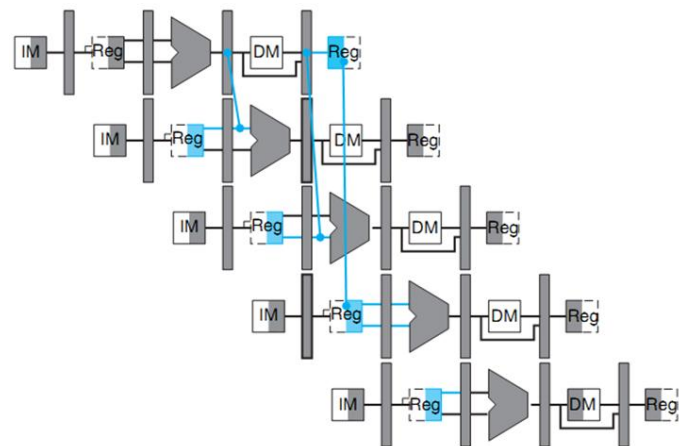
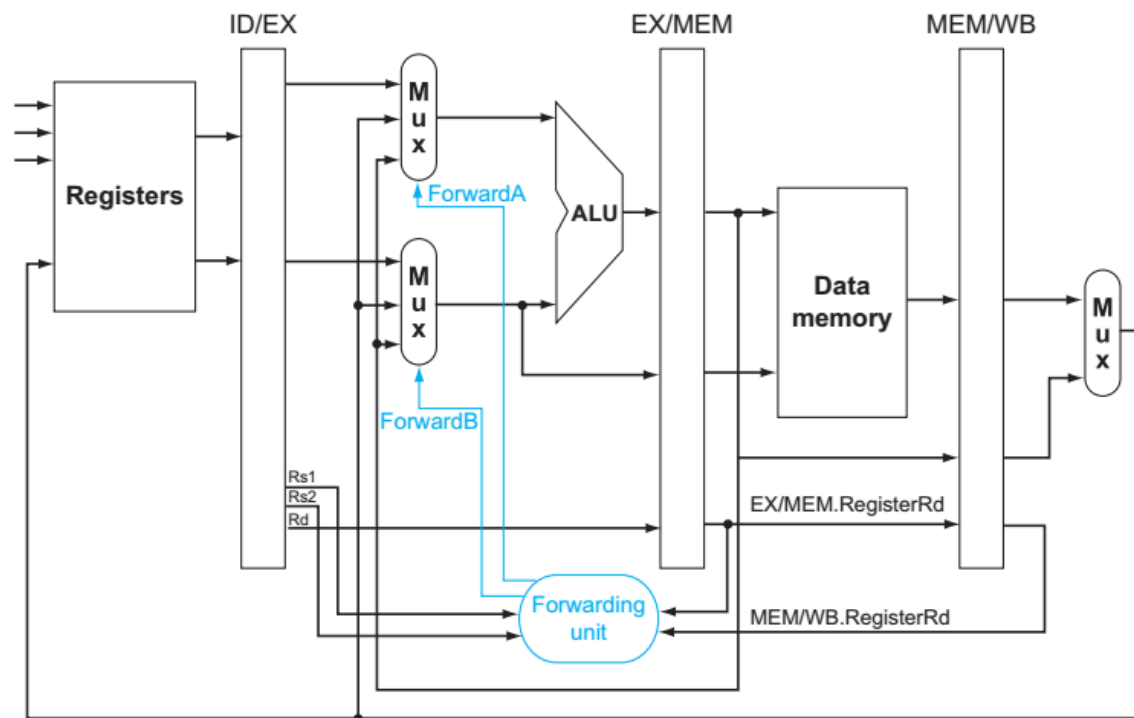
段寄存器: 后写

```
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, bug?

- 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 (2/3) : p221/p224

- 无须前推：两种case

- Case1: 非写回指令: RegWrite?——示例?

- Case2: 目标寄存器为\$zero: RegisterRd ≠ 0?

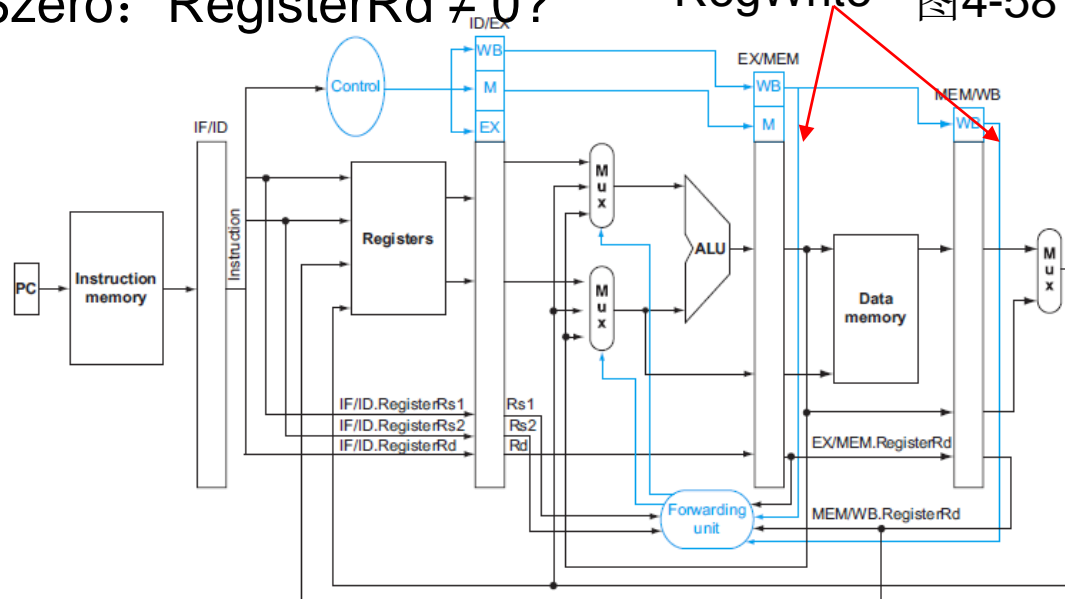
RegWrite 图4-58

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: CC#?

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.

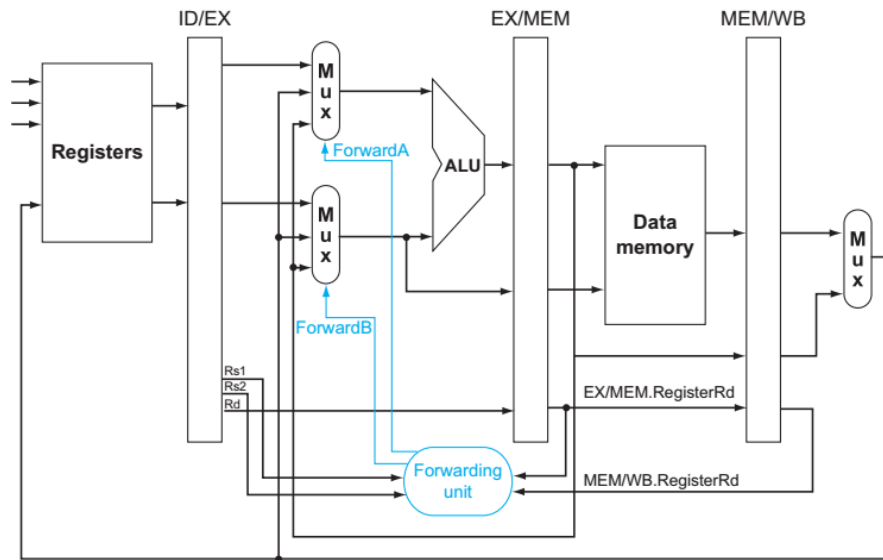


图4-57

```

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

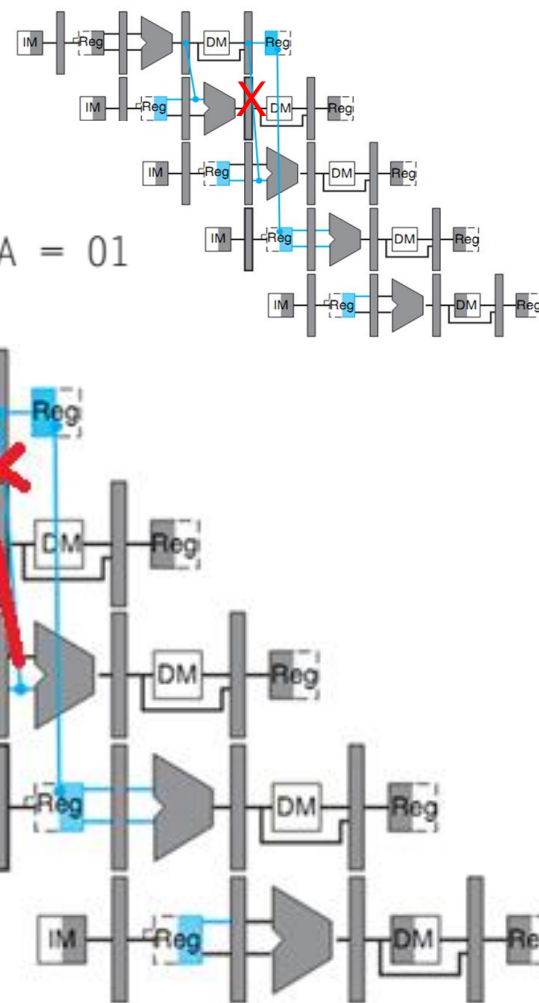
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
sub x2, x1, x3
and x12, x2, x5
or x13, x6, x2
add x14, x2, x2
sd x15, 100(x2)
```

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

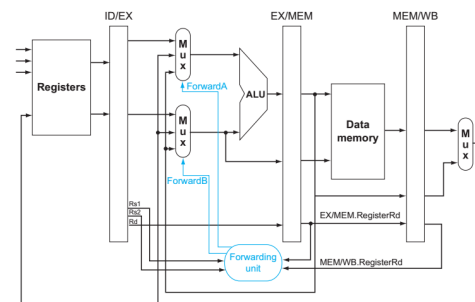
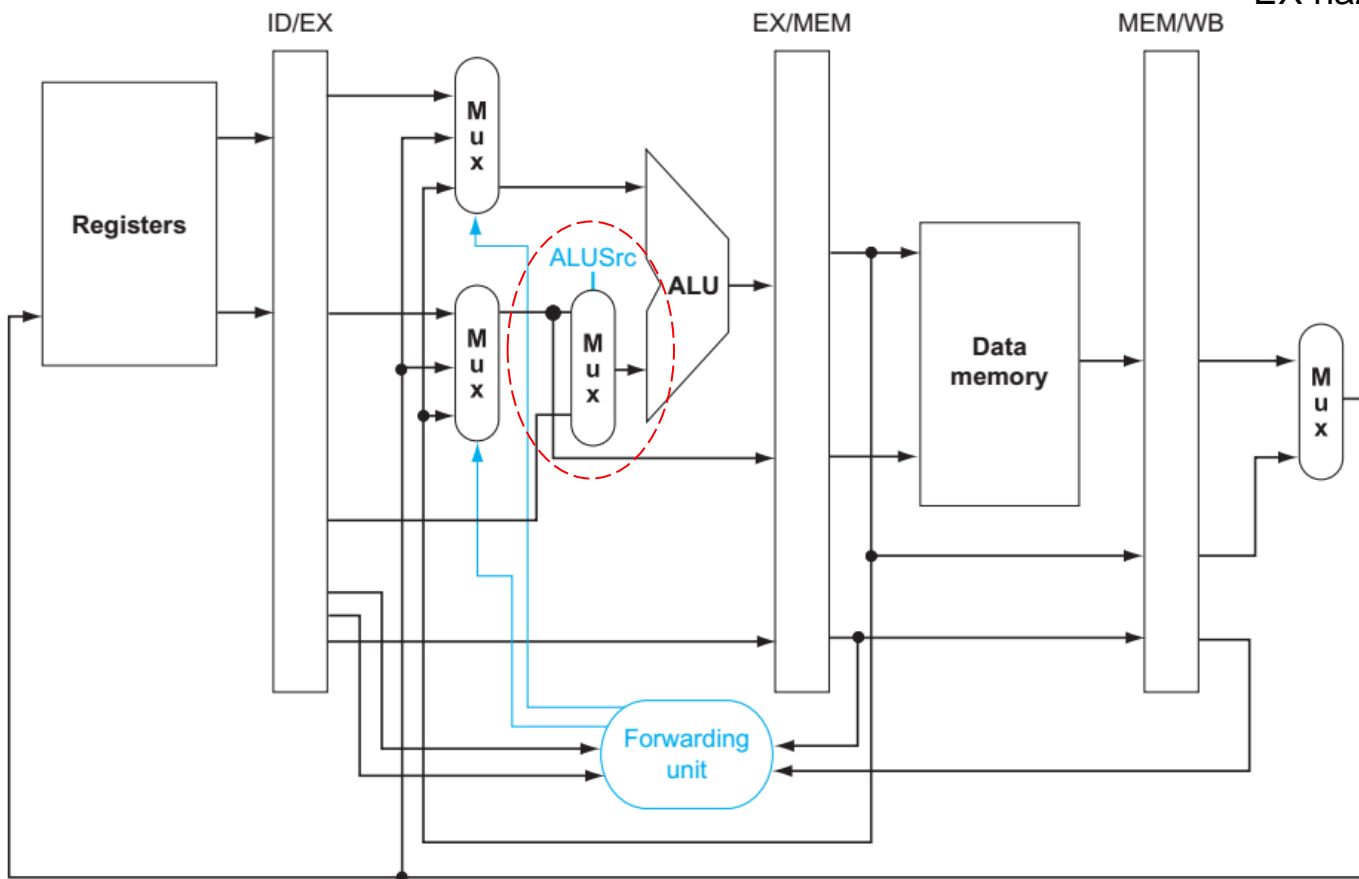


图4-56

图4-59

注意: I-type ALU 立即数MUX, 图4-56无

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

I-type

EX段冒险：完整的数据通路与控制

- 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

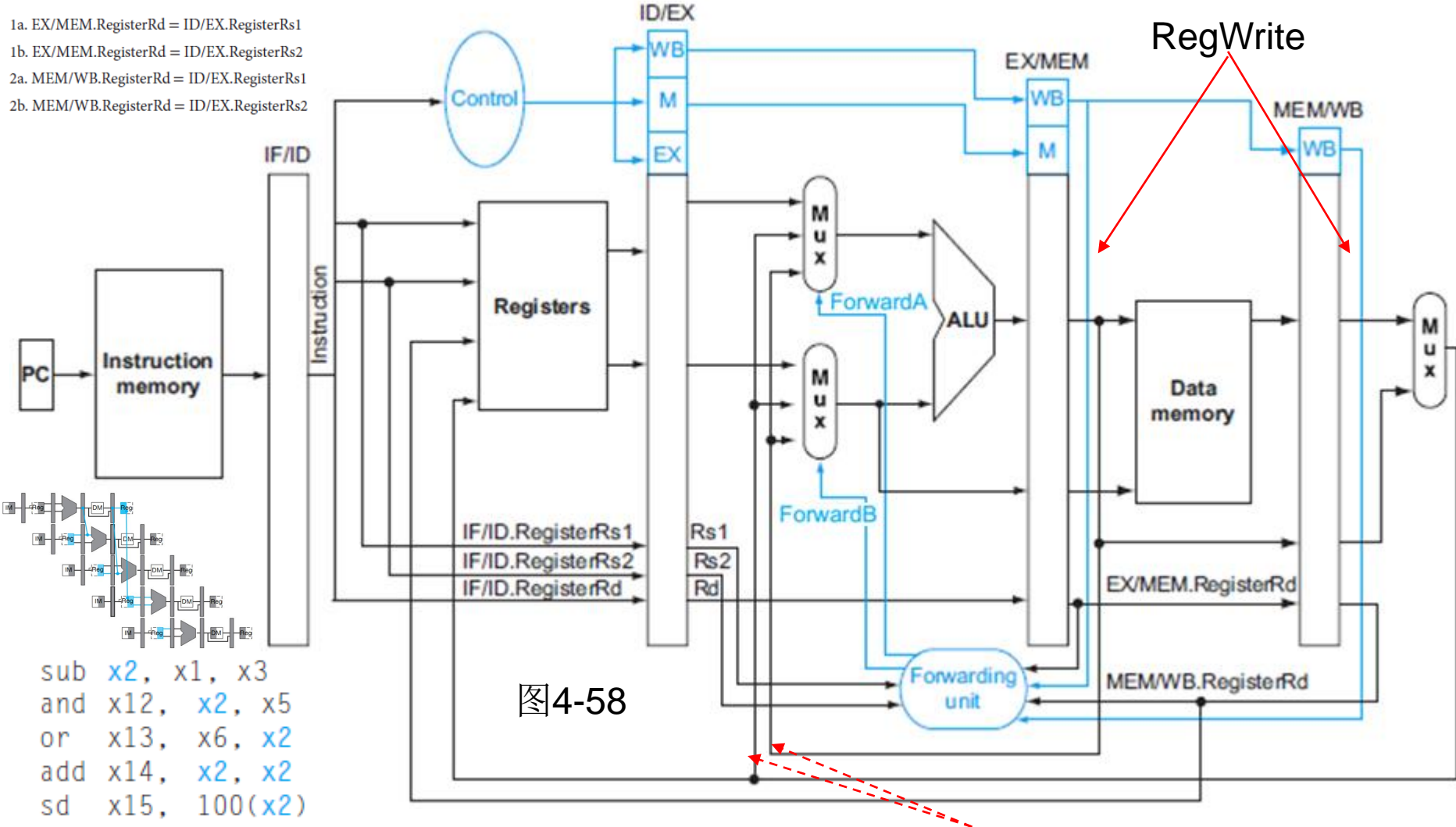


图4-58

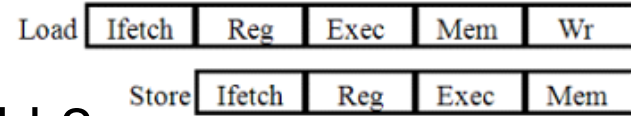
```

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

FWU: CC4将ALU结果前推至EX段 (ALU输入), CC5将ALU/lw结果前推至EX段₂₉

MEM段冒险：MEM copy问题， p225

- “RISC中，mem-to-mem拷贝频繁”

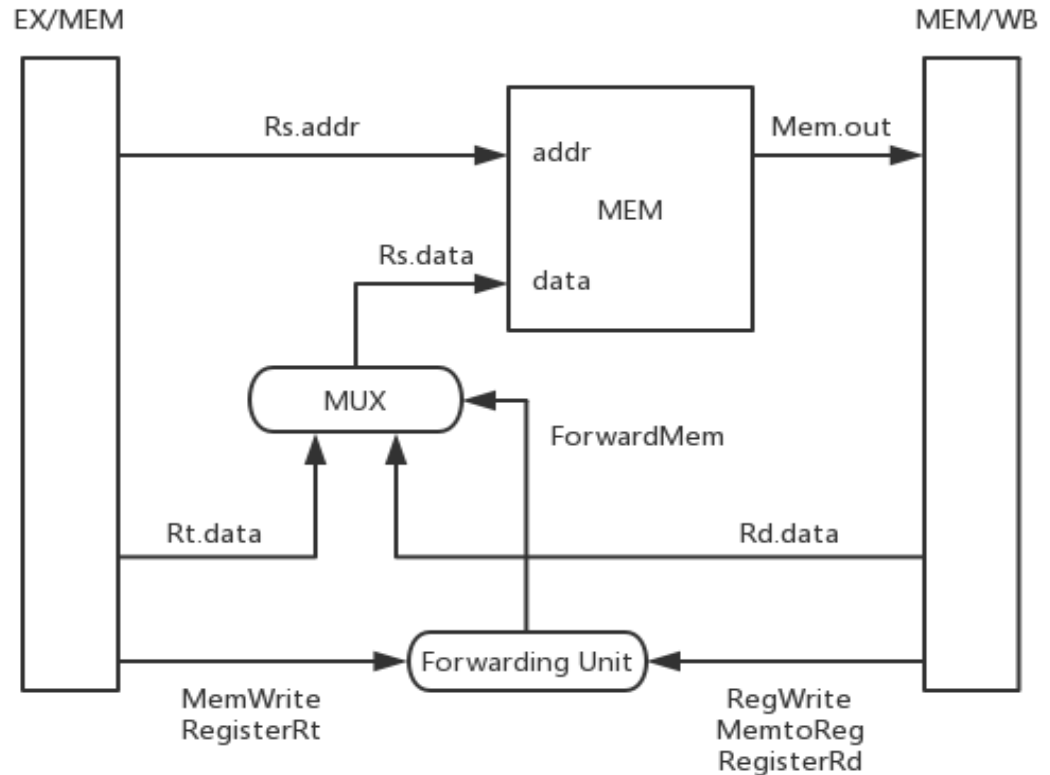
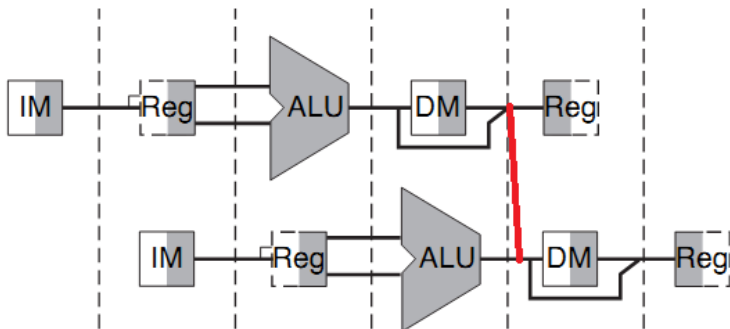


– 由lw-sw实现：lw \$1, addr1; sw \$1, addr2

- MEM段旁路：MEM/WB => MEM，无需插入bubble！

– 判据，DP，CC#？

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



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

MEM前推

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

Llxx编，正确？

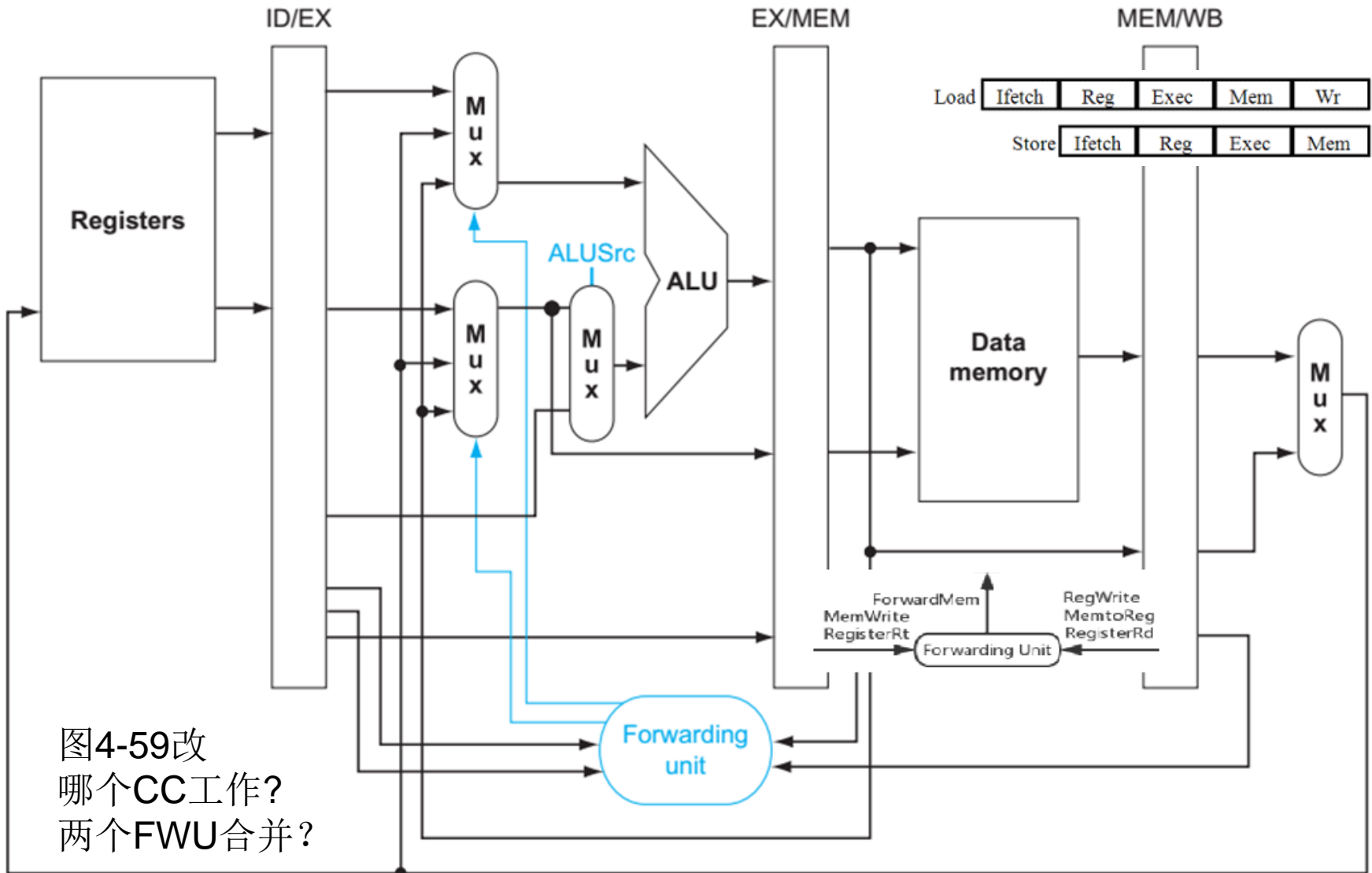
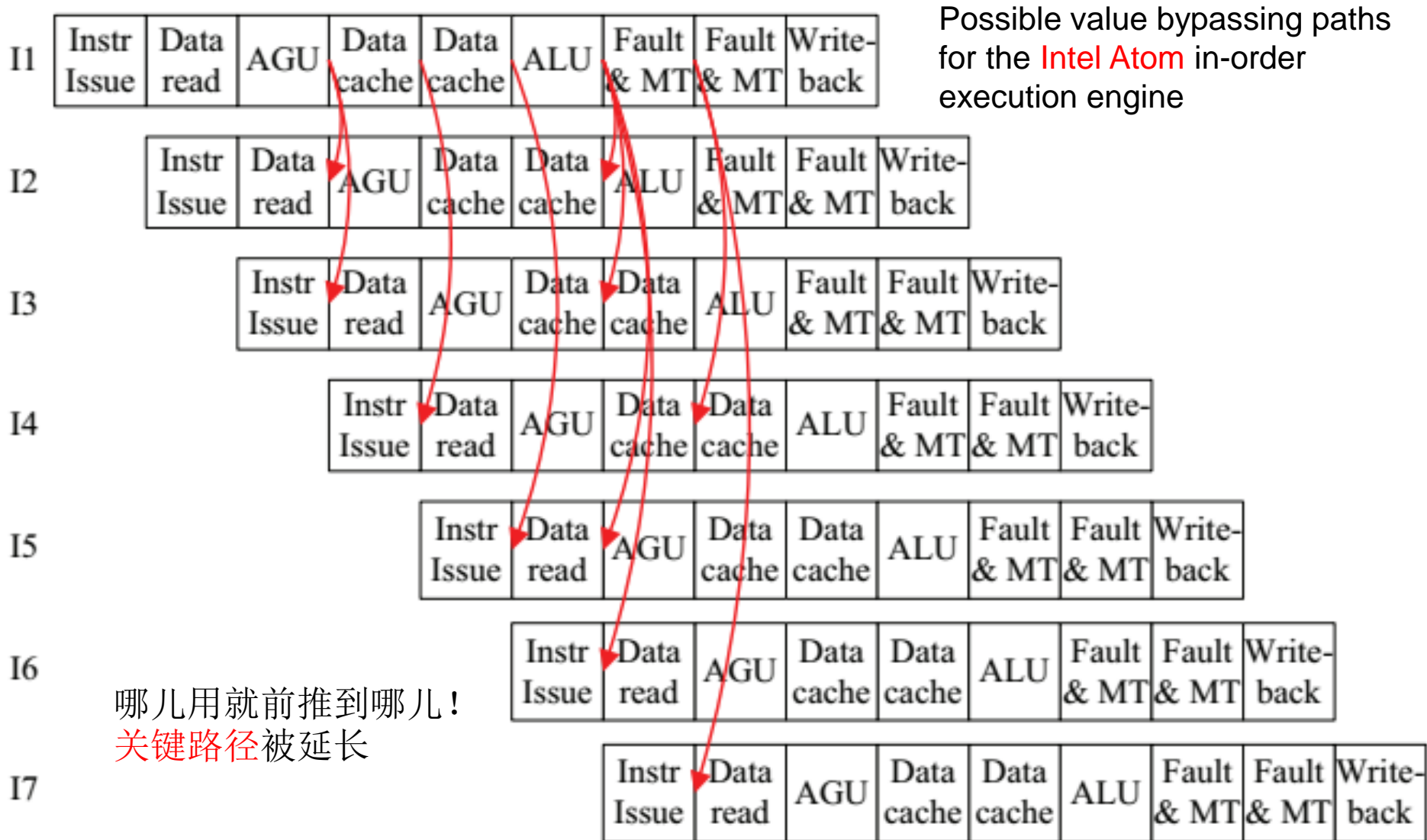


图4-59改
哪个CC工作？
两个FWU合并？

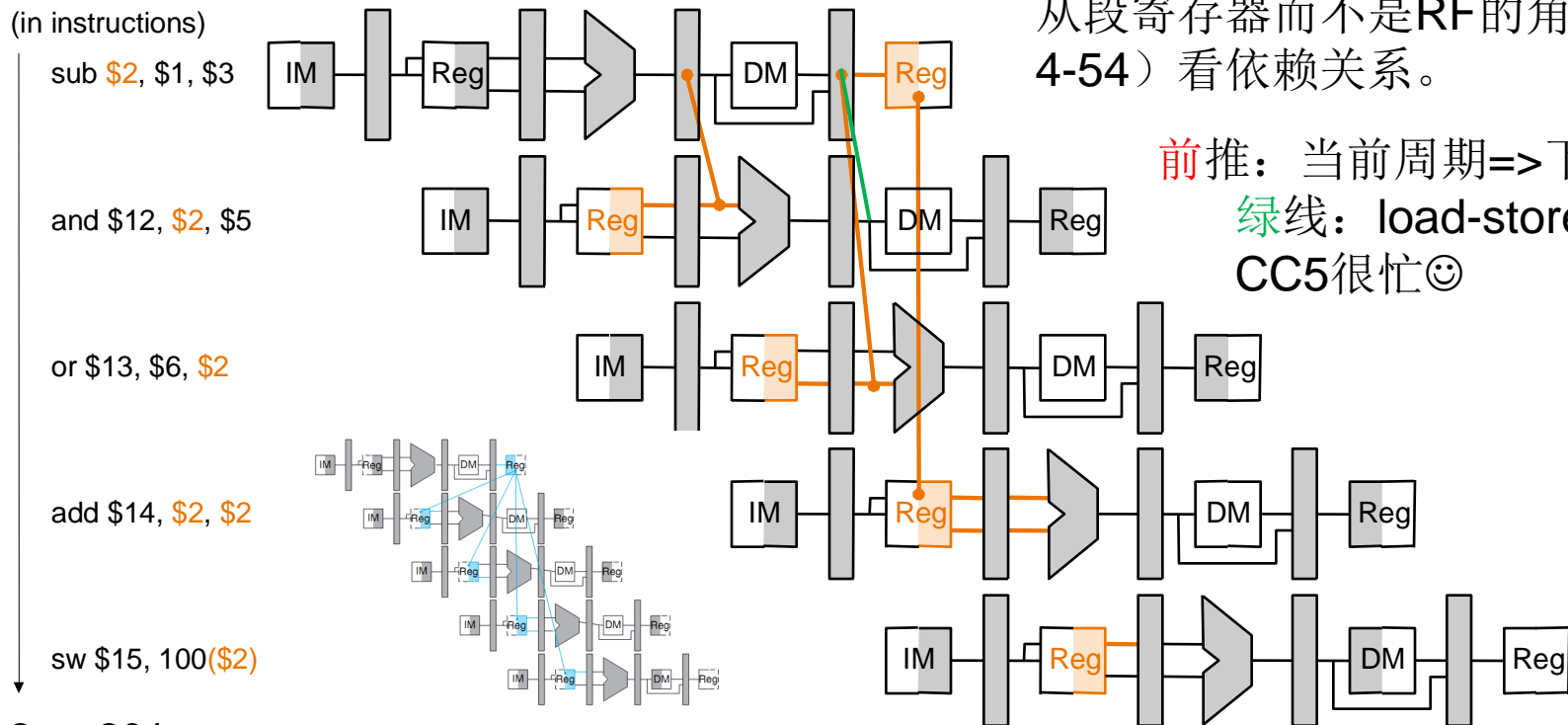
Multilevel Bypass: 需要多个段前推!



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)



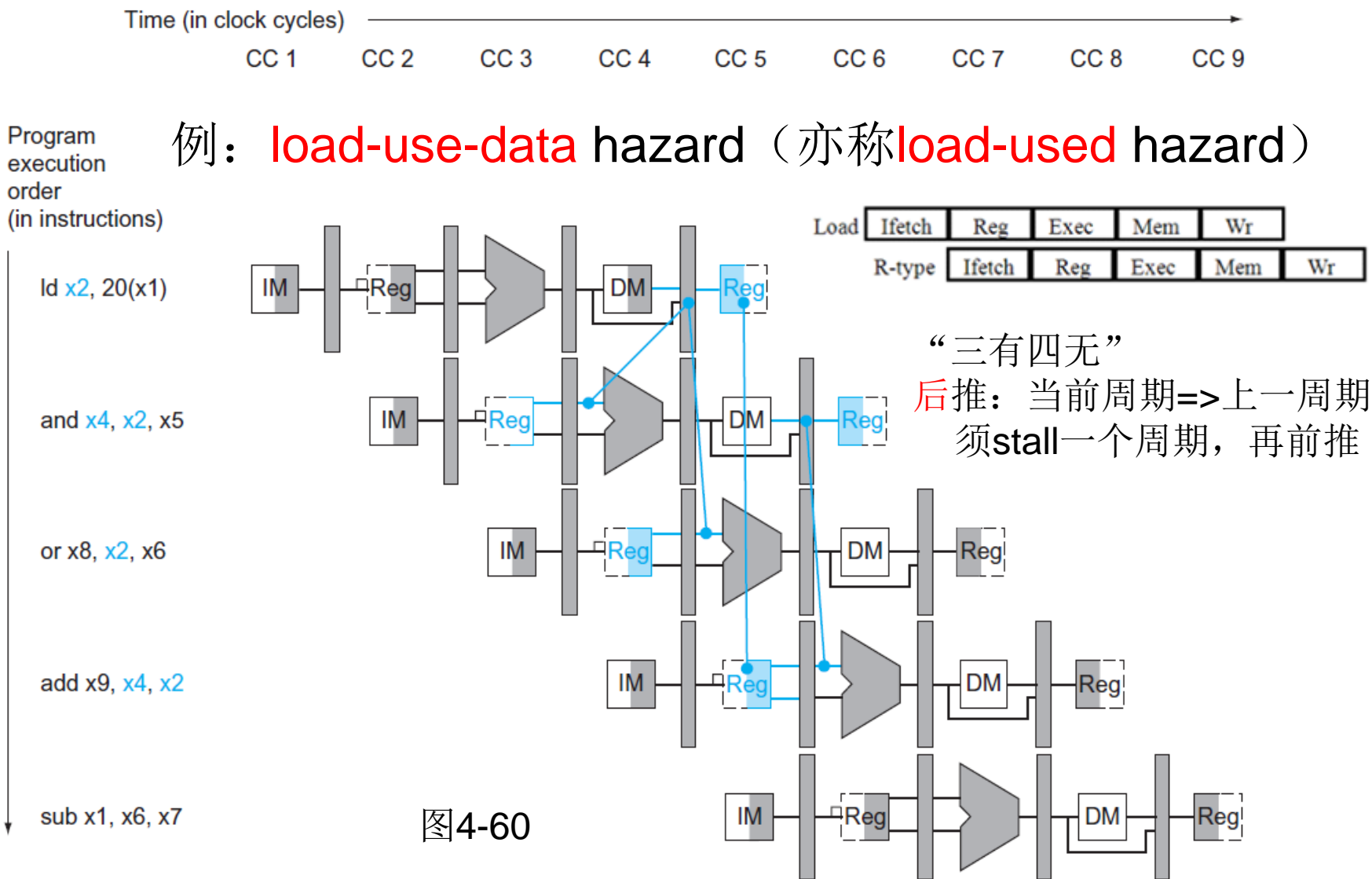
COD4图4-53（类RV2图4-55），从段寄存器而不是RF的角度（图4-54）看依赖关系。

前推：当前周期=>下一周期
绿线：load-store
CC5很忙☺

\$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！ 33

forwarding无法解决的RAW: backwards



load-use-data hazard, §4.6.2

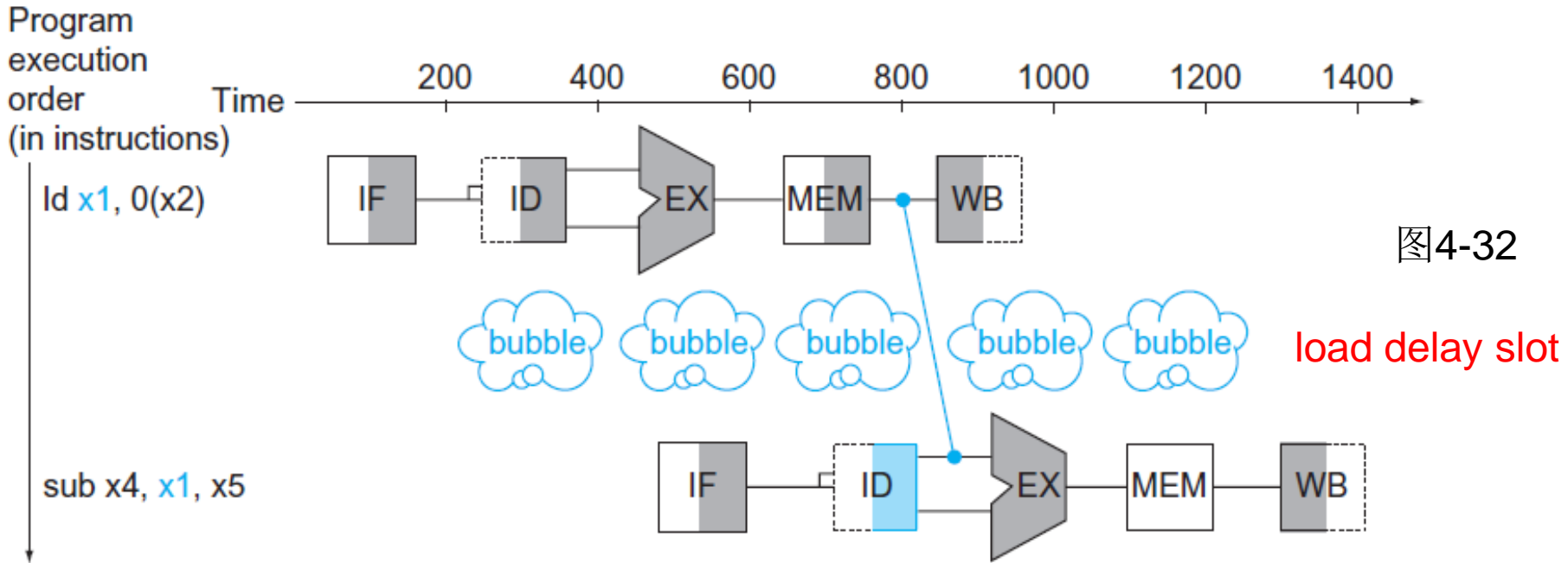
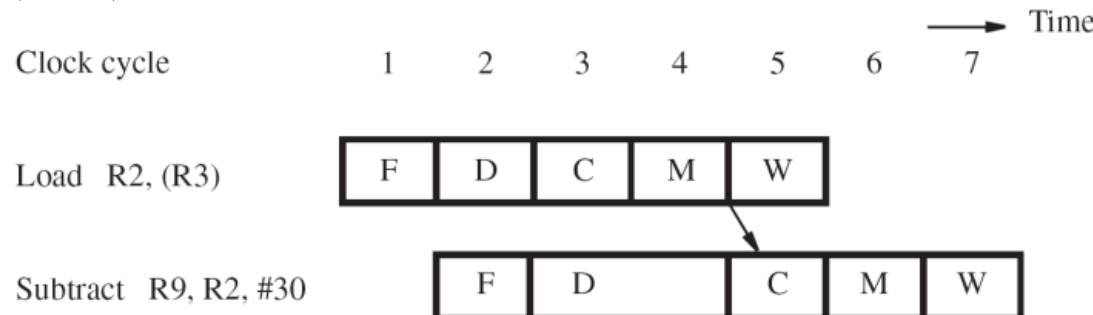


图4-32

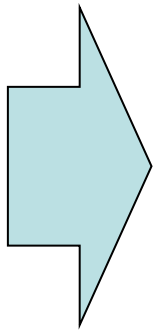
- 编译法: 插入nop (= sll \$0, \$0, 0) , 或, 指令重排序/指令调度
- 硬件法: **Interlocking**
 - stall: **stretch & delay**
 - FW: MEM/WB->EX
 - 判据?



Code Scheduling: 编译, §4.6.2

- 指令重排序: 程序序≠执行序, 【静态指令调度】
 - 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! 【满足结合律!】
 - 定点, 浮点

```
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
```

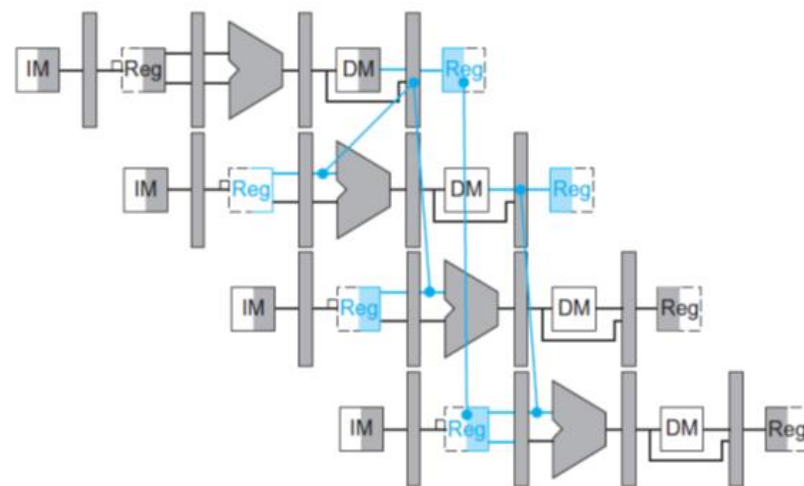
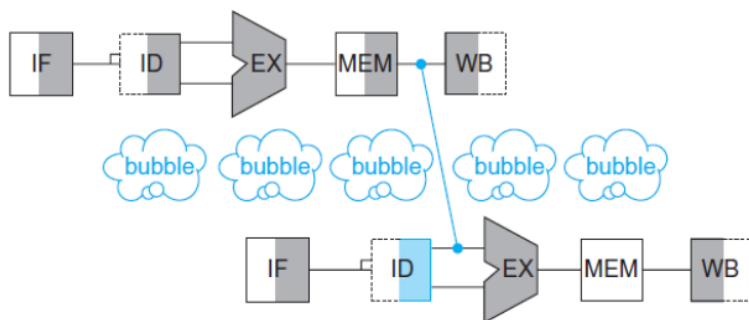


```
ld    x1, 0(x31)
ld    x2, 8(x31)
ld    x4, 16(x31)
add   x3, x1, x2
sd    x3, 24(x31)
add   x5, x1, x4
sd    x5, 32(x31)
```

此段代码的执行时间?

硬件stall: A-B互锁 “interlock”

- **Interlock:** 安全机制, 针对hazard
 - 电梯运行时禁止开门
- 流水线技术: 乐观, 投机, 存在冒险
- Interlocked Pipelined Stages
 - **RAW: EX、MEM、WB**
 - HDU(hazard detection unit)



RAW Stall: Interlocking, \$4.8

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

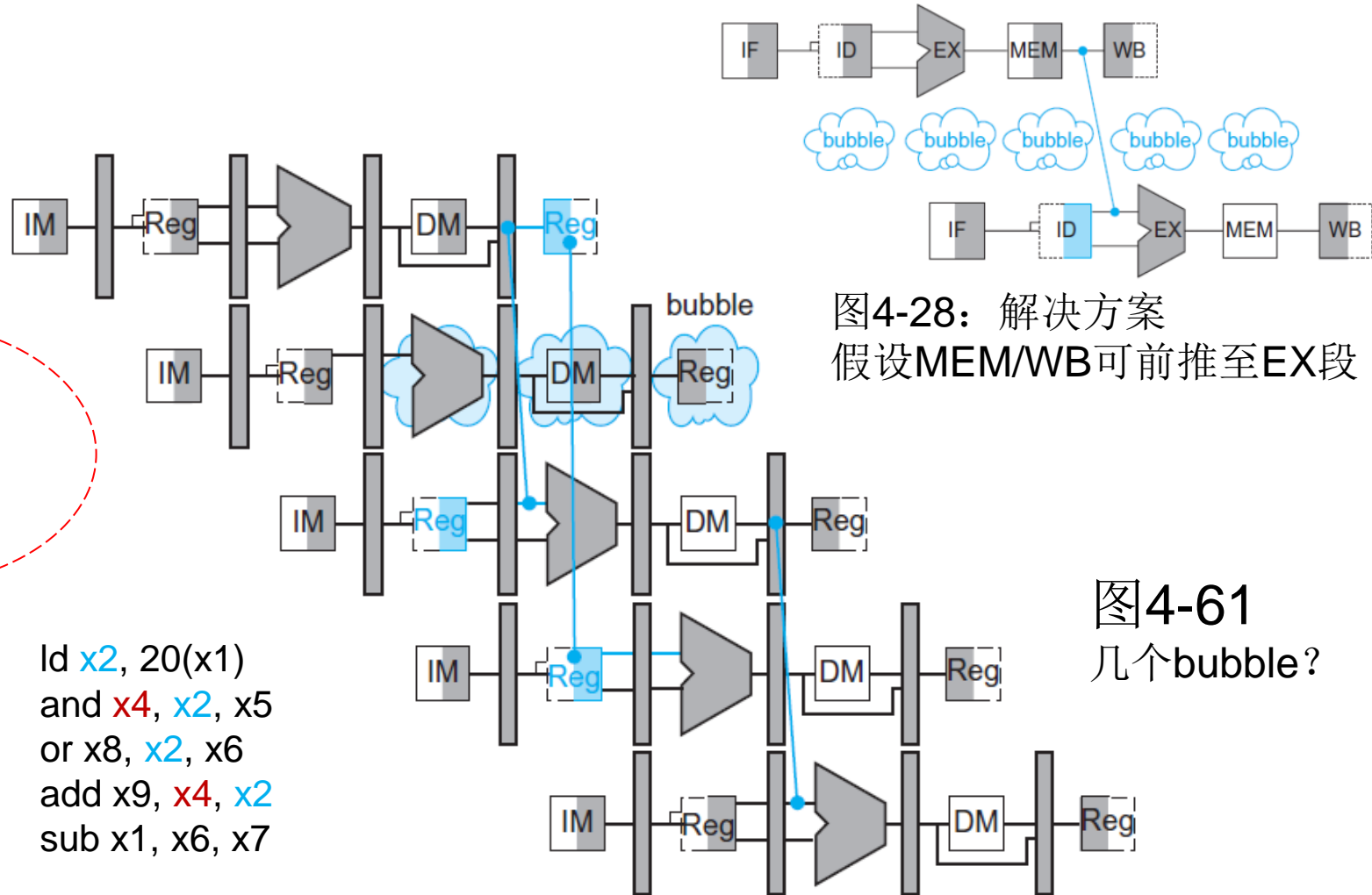


图4-28: 解决方案
 假设MEM/WB可前推至EX段

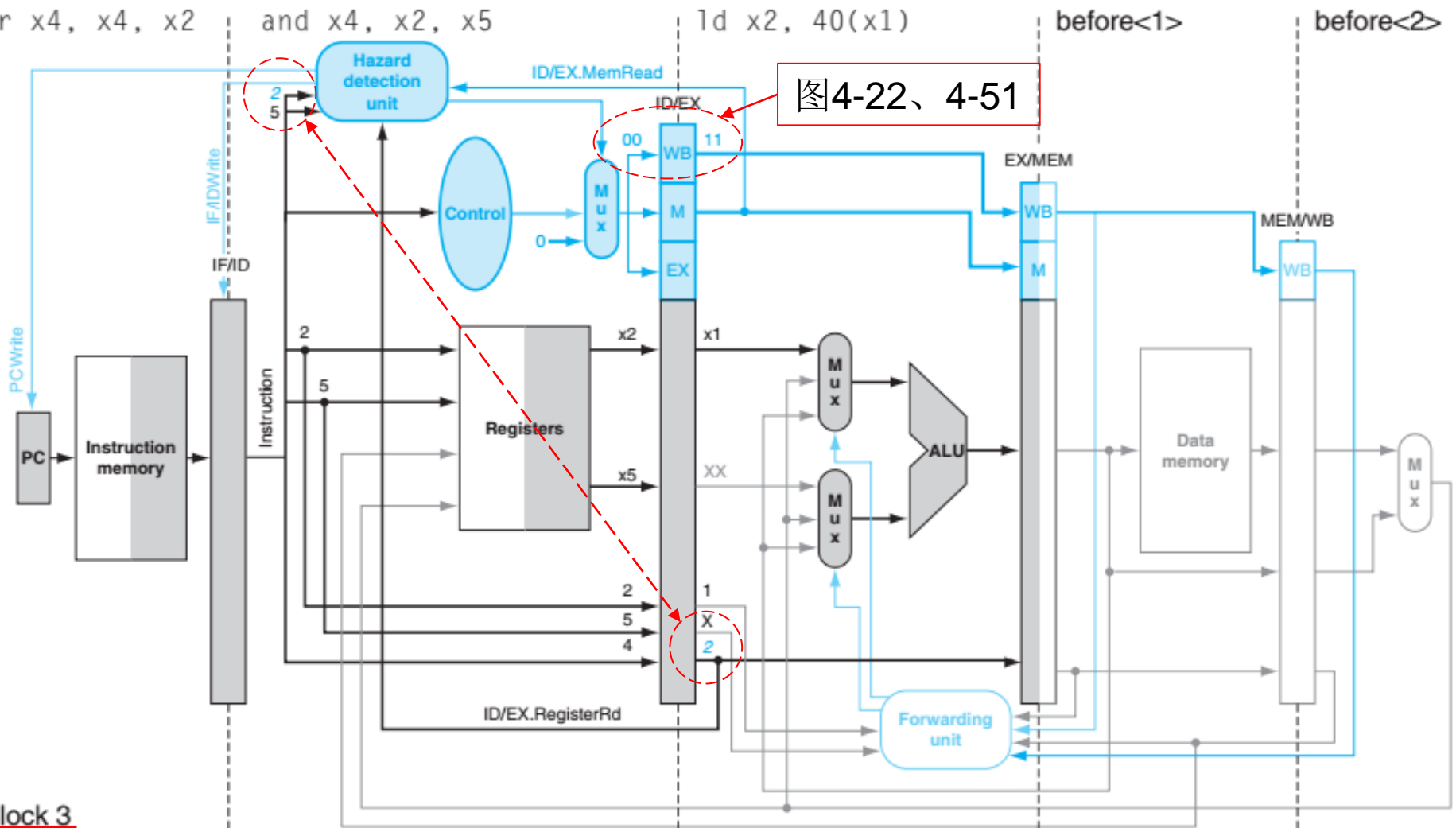
图4-61
 几个bubble?

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: 在ID段!
 - 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（门控）: 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，控制
 - load-use-data stalling规则，CC#?

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 【电子版】： 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
```

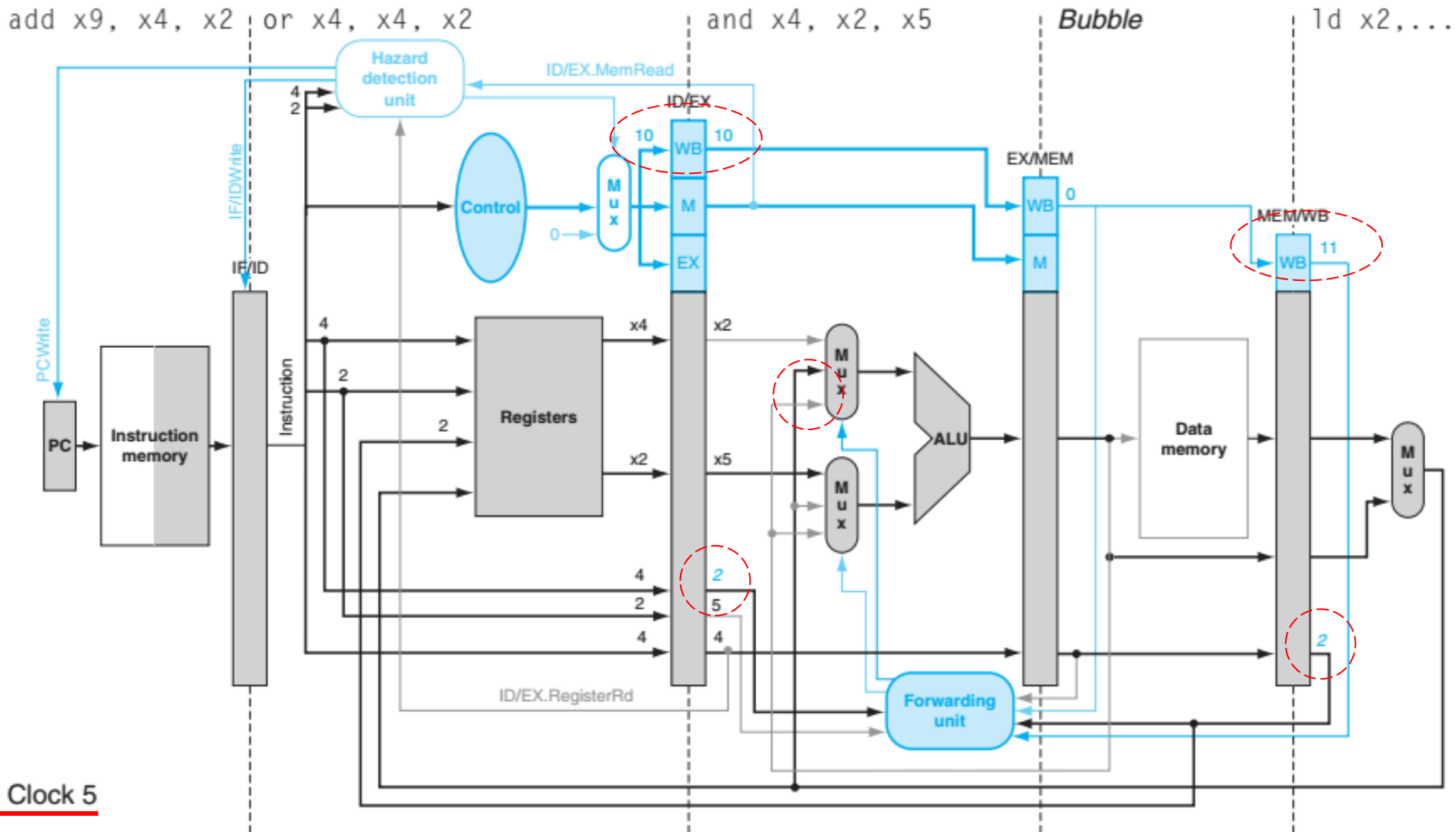
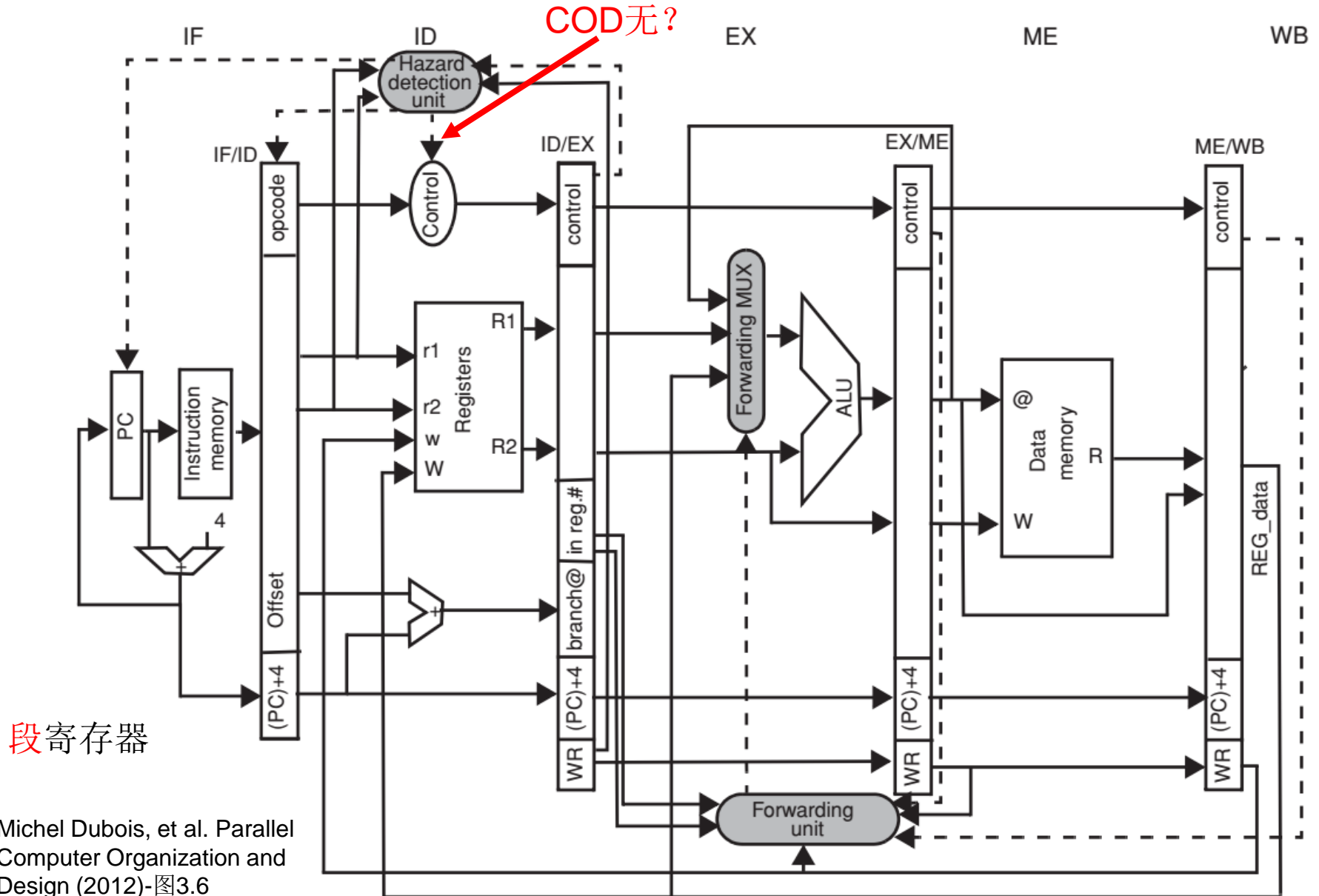


FIGURE e4.13.19: `and`进入EX, 需FWU从MEM/WB前推x2。同时, 此周期ld写回(先写), 故or的x2无需FW

参考版本: HDU->Controler, 能否在EX段?

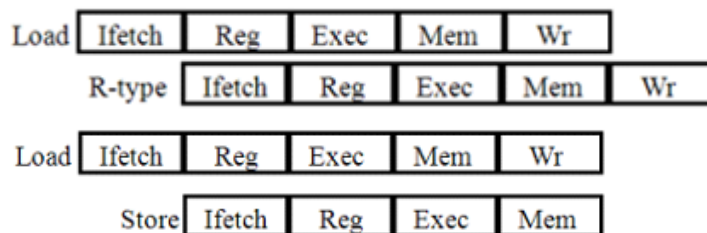


访存指令的RAW: in-order ppl, 无store-buf

- 寄存器依赖【load-used】：EX段冒险，one stall

– load-Rtype: 不是“写端口的结构冲突”！

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



- 存储器依赖：MM段冒险，无stall

– 内存Copy: 不同地址，前推

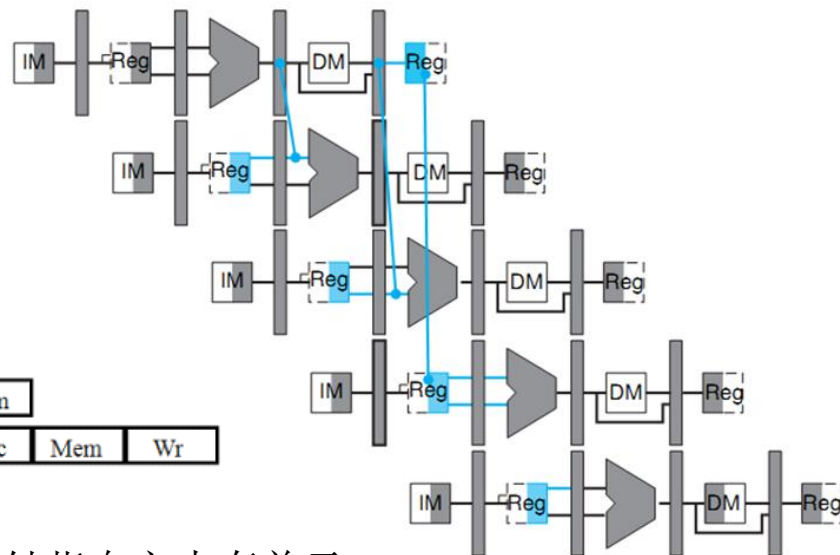
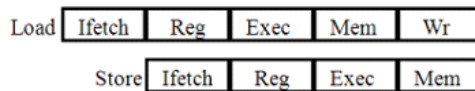
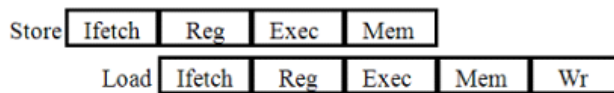
- lw x2, 100(x1)
- sw x2, 110(x1)

– 校验: 同一地址，无冒险

- sw x2, 100[x1]
- lw x2, 100[x1]
- xor x1, x2, x2

– 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]



小结：【PH基本5段流水线】的冒险

- 假设：单周期整数部件，单周期访存，单发射

- 结构冲突：同一CC，消除，无须stall

- 单端口MEM：IF，MEM，哈佛结构

- RF单一写端口：lw-Rtype，Rtype指令延期WB

- 流水线ISA特点4：“每条指令只写一个结果，且只在WB段写RF”

- 胡伟武《计算机体系结构基础》：“RAW阻塞可能造成IF或ID结构冲突”？

- RAW冒险：不同CC，生产者-消费者，并行

- “三有四无”：ID、EX、MEM、WB？

- 流水线ISA特点4：“每条指令只写一个结果，且只在WB段写RF”

- ID段：RF内推，“RF先写后读”，消除

- EX段：ALU源操作数

- MEM和WB段前推至EX，消除

- load-used-hazard: stall, 且WB前推至EX, 缓解

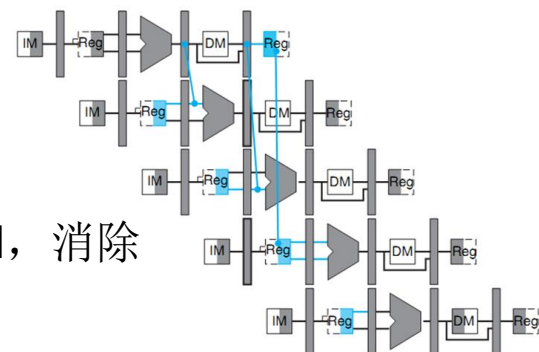
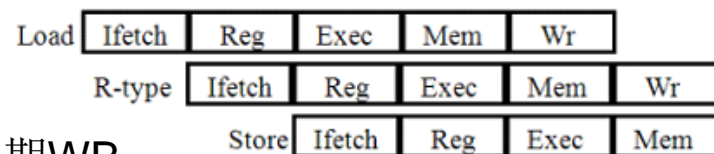
- MEM段：Store-hazard (lw-sw)，WB前推至MEM，消除

- 分支

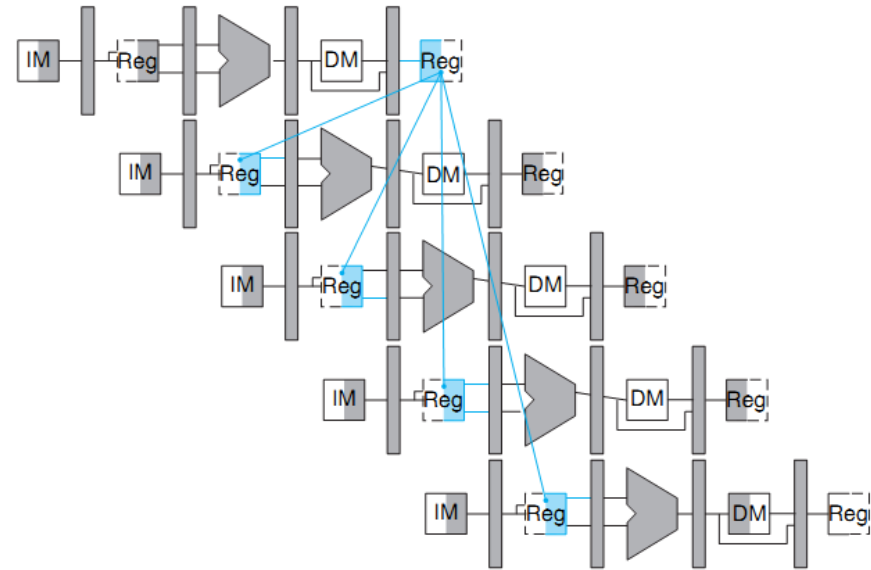
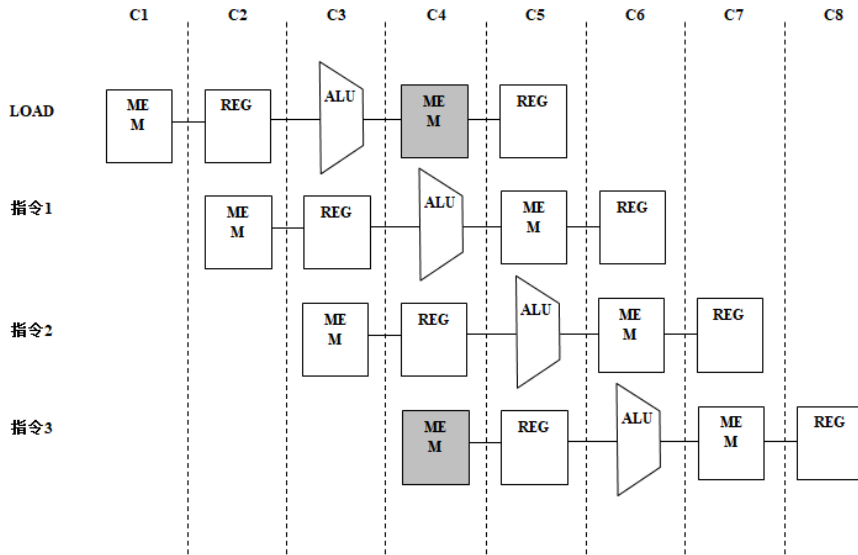
- 理想流水线：贪心、投机（冒险）

- 冒险处理（关键路径）：FWU，HDU（interlock, stall），flush

- Dubois：“基本五段ppl的所有冒险在ID段处理完，其后不会被阻塞”？

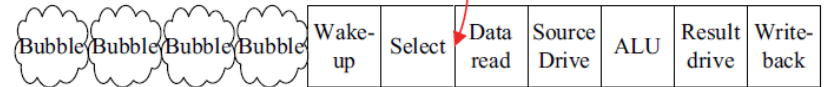
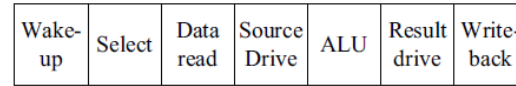


结构冲突, RAW, 时序 (发生, 处理)



- 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