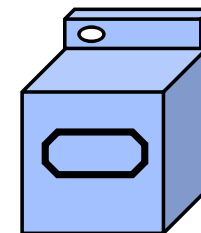
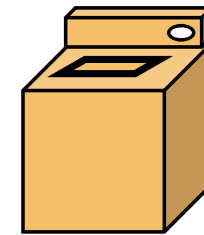
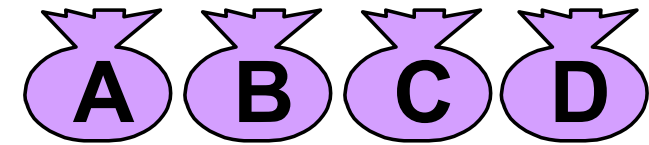


一个日常生活中的例子——洗衣服

◦ Laundry Example

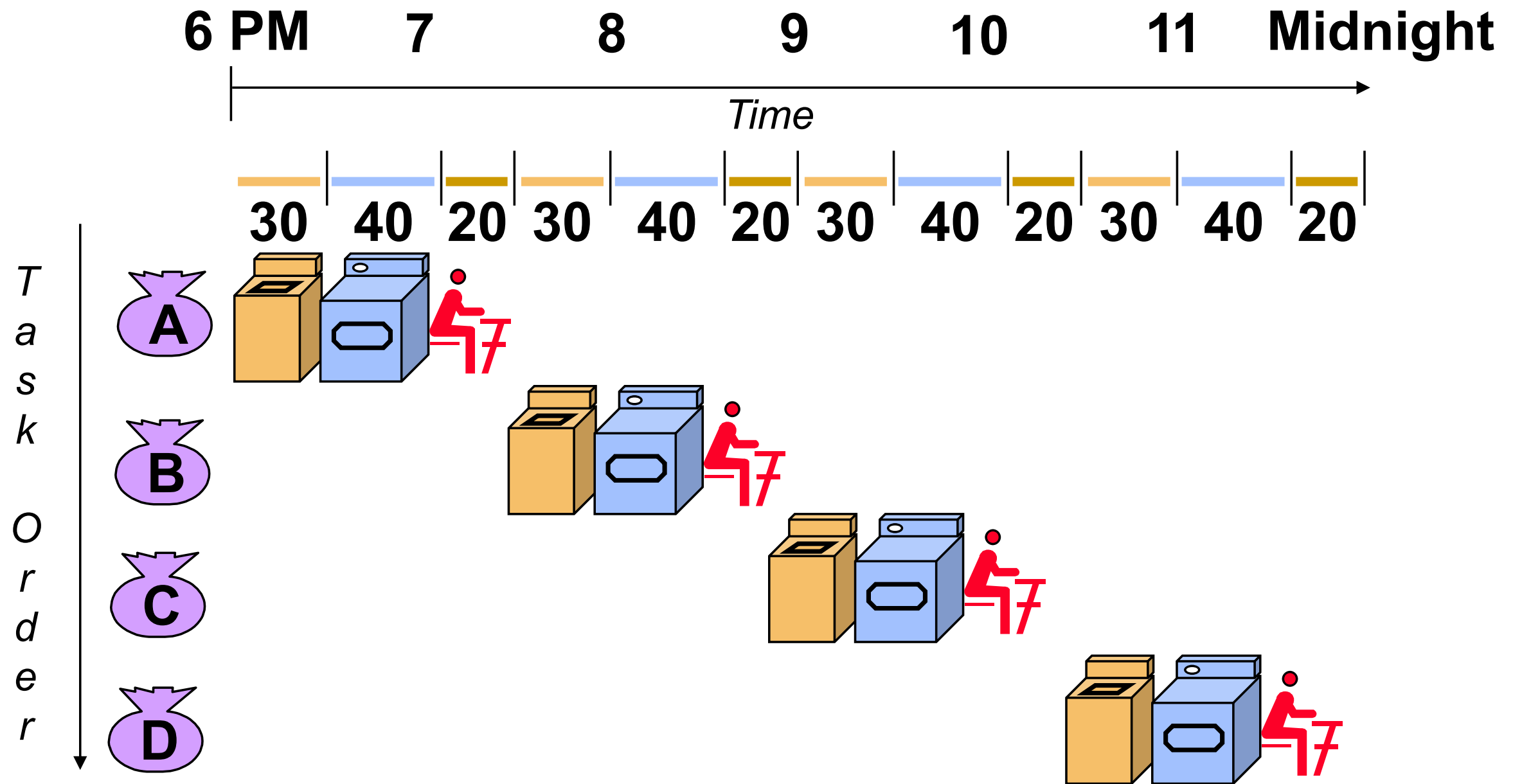
- A, B, C, D四个人，每人都有一批衣服需要 wash, dry, fold
- Wash阶段: 30 minutes
- Dry阶段: 40 minutes
- Fold阶段: 20 minutes



如果让你来管理洗衣店，你会如何安排？

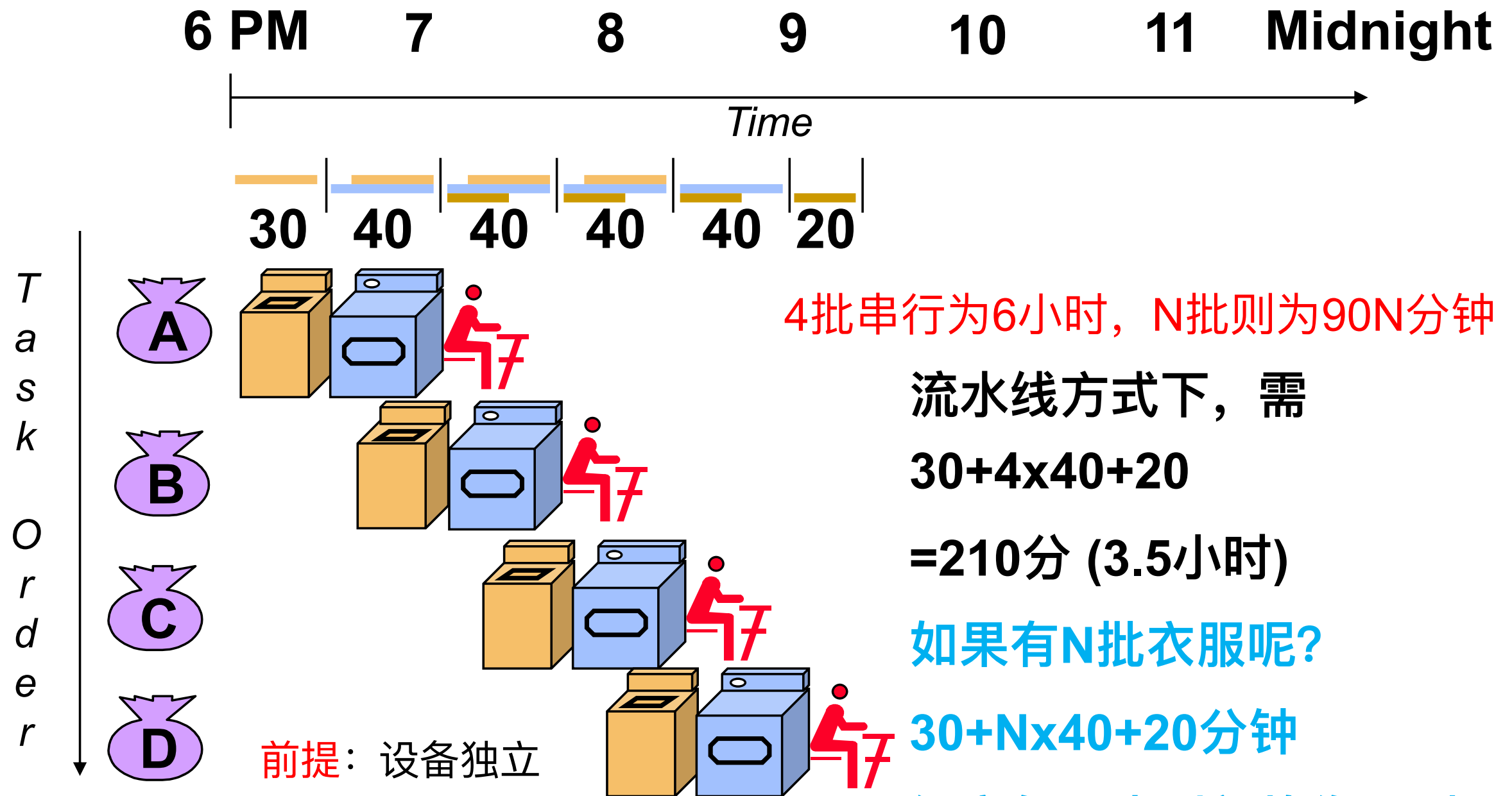
Pipelining: It's Natural !

Sequential Laundry (串行方式)



- 串行方式下，4 批衣服需要花费 6 小时 ($4 \times (30 + 40 + 20) = 360$ 分钟)
- N 批衣服，需花费的时间为 $N \times (30 + 40 + 20) = 90N$
- 如果用流水线方式洗衣服，则花多少时间呢？

Pipelined Laundry: (Start work ASAP)



流水线方式下，需

$$30+4\times 40+20$$

$$=210\text{分 (3.5小时)}$$

如果有N批衣服呢？

$$30+N\times 40+20\text{分钟}$$

假定每一步时间均衡，则
比串行方式提高约3倍！

流水方式下，所用时间主要
与最长阶段的时间有关！

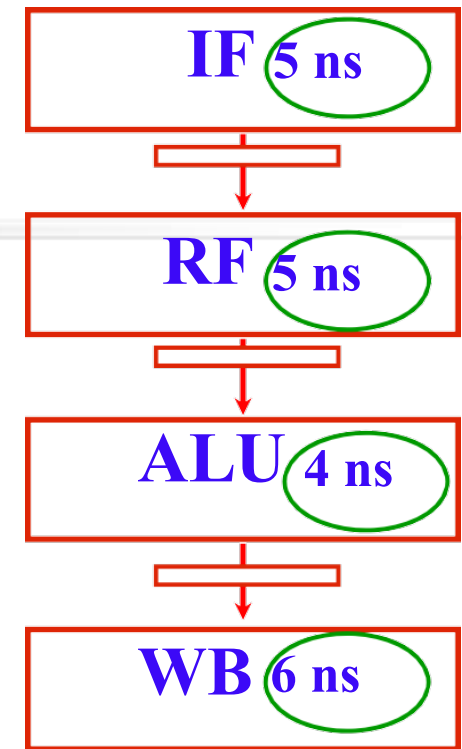
数据通路流水线化



目标: 保证(几乎) 1.0 的CPI, 同时提高时钟速率 **方法:**

将处理器转变为一个多级流水线

- Instruction Fetch IF: 维护 PC. 每周期取一个指令.
- Register File(RF/ID): 从寄存器组RF中读取操作数.
- ALU: 执行指定的运算.
- Data Access: 存储器访问(I type load store 指令需要)
- Reg Write-Back(WB): 将结果写回到寄存器中.



R型指令的 4 级流水线可以如图所示

Non-Pipelined Execution

Instruction	Fetch (I-MEM)	Reg. Read	ALU Op.	Data Mem	Reg. Write	Total Time
Load	10 ns	5 ns	10 ns	10 ns	5 ns	40 ns
Store	10 ns	5 ns	10 ns	10 ns		35 ns
R-Type	10 ns	5 ns	10 ns		5 ns	30 ns
Branch	10 ns	5 ns	10 ns			25 ns
Jump	10 ns	5 ns				10 ns

time 

40 ns



LW \$5,100(\$2)



40 ns



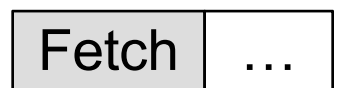
LW \$7,40(\$6)



40 ns



LW \$8,24(\$6)

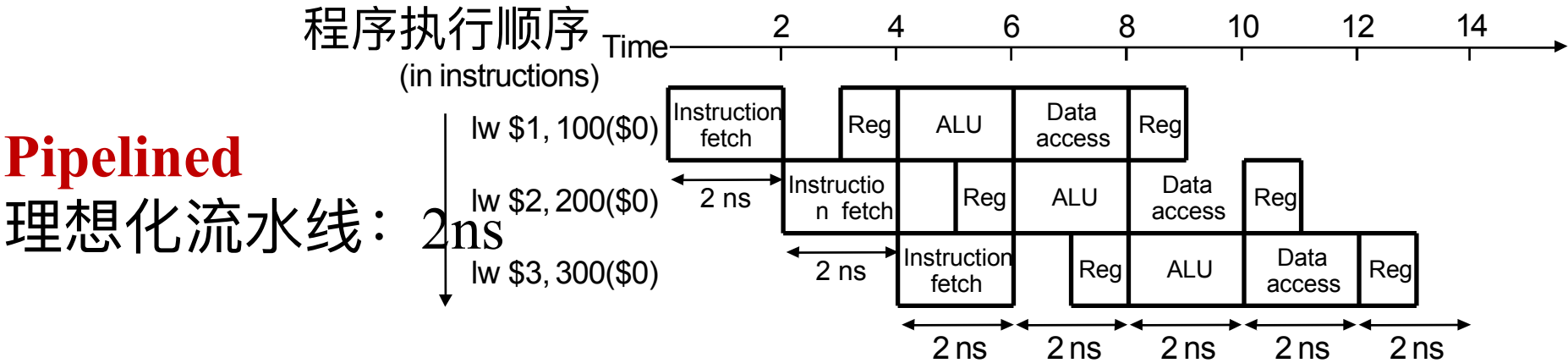
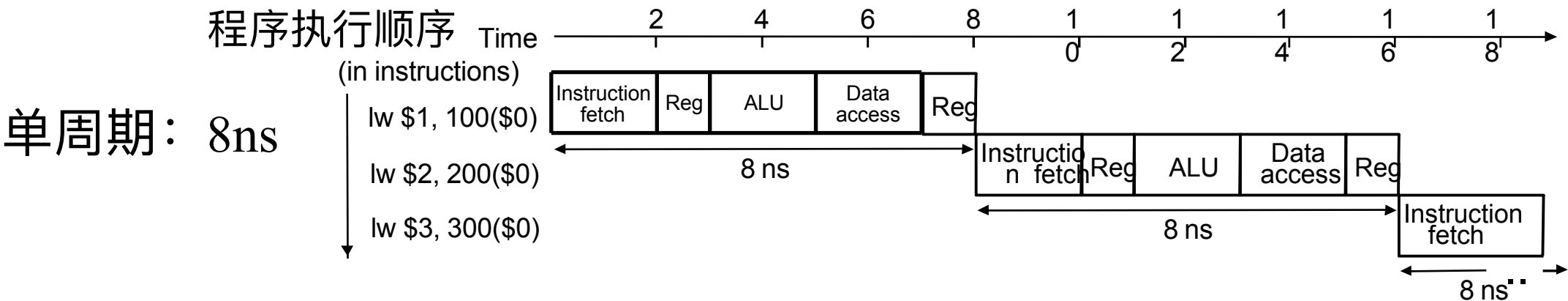


3 Instructions = 3*40 ns

Pipelining

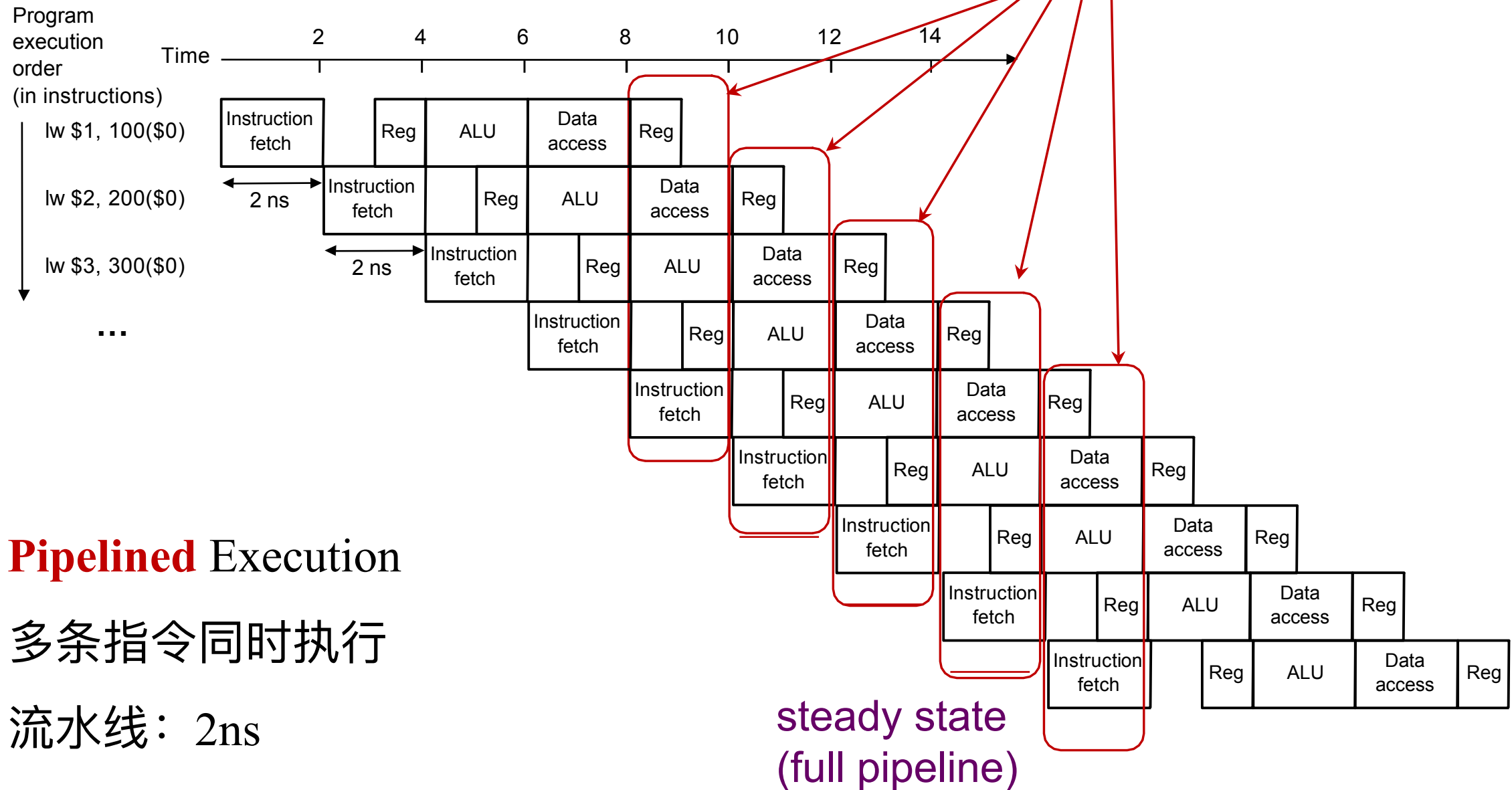
通过提高指令吞吐率(throughput)改善性能

Instruction Fetch	Register File Access (Read)	ALU Operation	Data Access	Register Access (Write)
2ns	1ns	2ns	2ns	1ns



Pipelining (Cont.)

多条指令同时执行



Pipelined Execution

多条指令同时执行

流水线: 2ns

目标: 在不增加成本的情况下提高吞吐量 (在指令处理的情况下, 硬件成本)

Pipelining

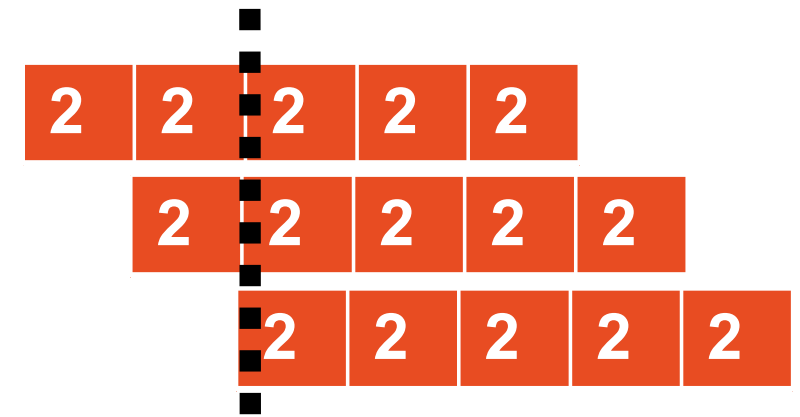
流水线性能

时钟周期等于最长阶段所花时间为：2ns

每条lw指令的执行时间为：2nsx5=10ns

N条指令的执行时间为：(5+(N-1))x2ns

在N很大时约为2Nns，比串行方式提高约4倍，若各阶段操作均衡(例如，各阶段都是2ns，则串行需10Nns，流水线仍为2Nns)，加速为5倍。



Pipeline Speedup

$$\text{Time to execute an instruction}_{\text{pipeline}} = \frac{\text{Time to execute an instruction}_{\text{sequential}}}{\text{Number of stages}}$$

- 如果不均衡，加速比会小一些
- 加速来自于提高了吞吐率 **throughput**
- 指令的延迟时间 **latency** 没有减少

流水线执行方式能大大提高指令吞吐率，现代计算机都采用流水线方式！

Pipelining Performance

Example:时钟周期200PS，多周期R指令4T（45%），LW指令5T（25%），SW指令4T（10%），BEQ指令3T(15%)，j指令3T(5%)，那么平均CPI为多少，如果采用5级流水线，假设流水线时钟周期需要加40ps,那么流水线实现同样的指令加速比为多少？

Solution:

多周期： $cpi = 4 * 45\% + 5 * 25\% + 4 * 10\% + 3 * 15\% + 3 * 5\% = 4.05$

$Avg. \text{ instr. exec time}_{unpipelined} = \text{Clock cycle time} \times Avg. CPI = 200 \times 4.05 = 810ps$

理想情况下流水线对所有指令CPI=1

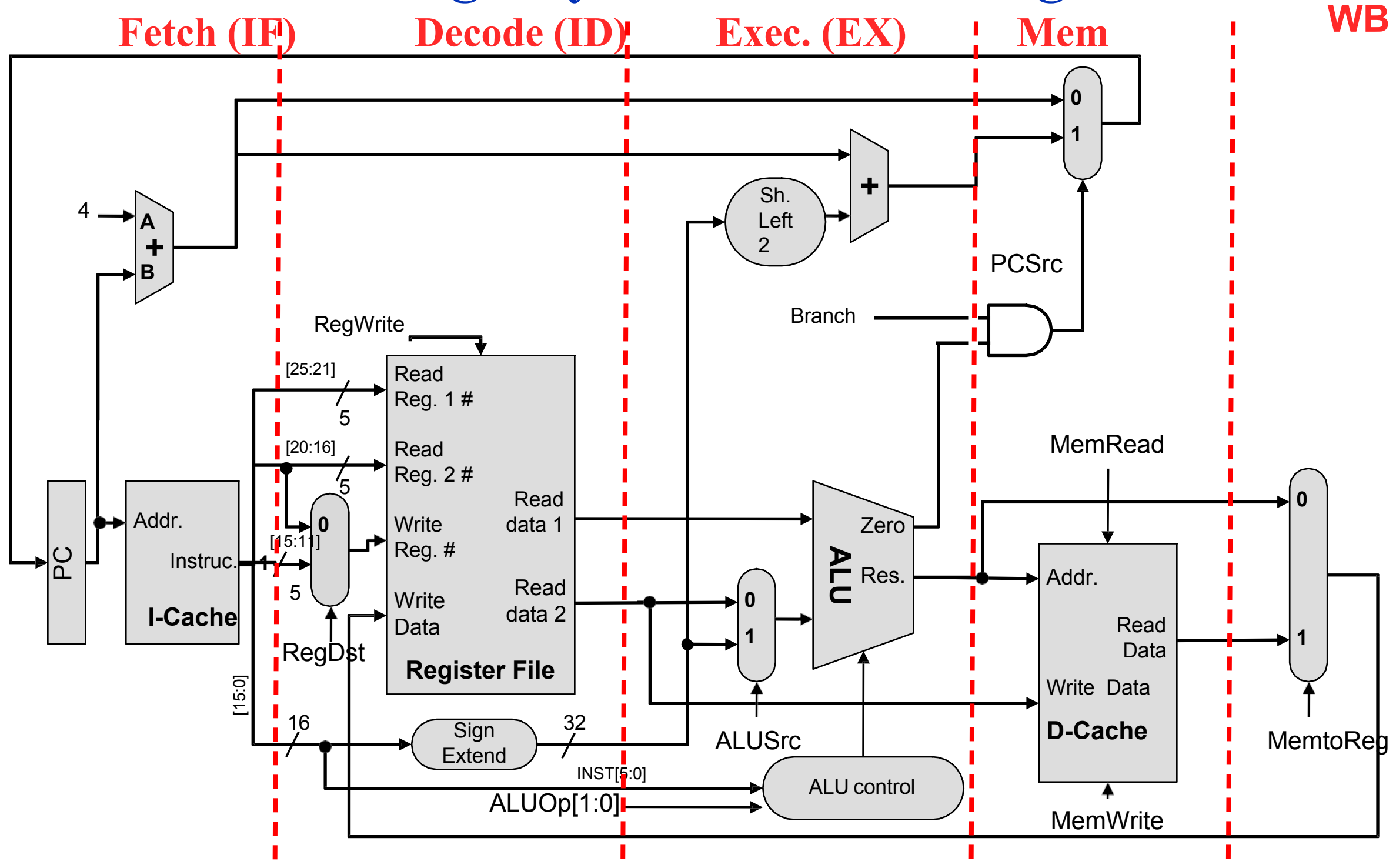
时钟周期=200ps + 40ps=240ps, 那么平均每条指令执行时间

$Avg. \text{ instr. exec time}_{pipelined} = 240ps \times 1 = 240ps$

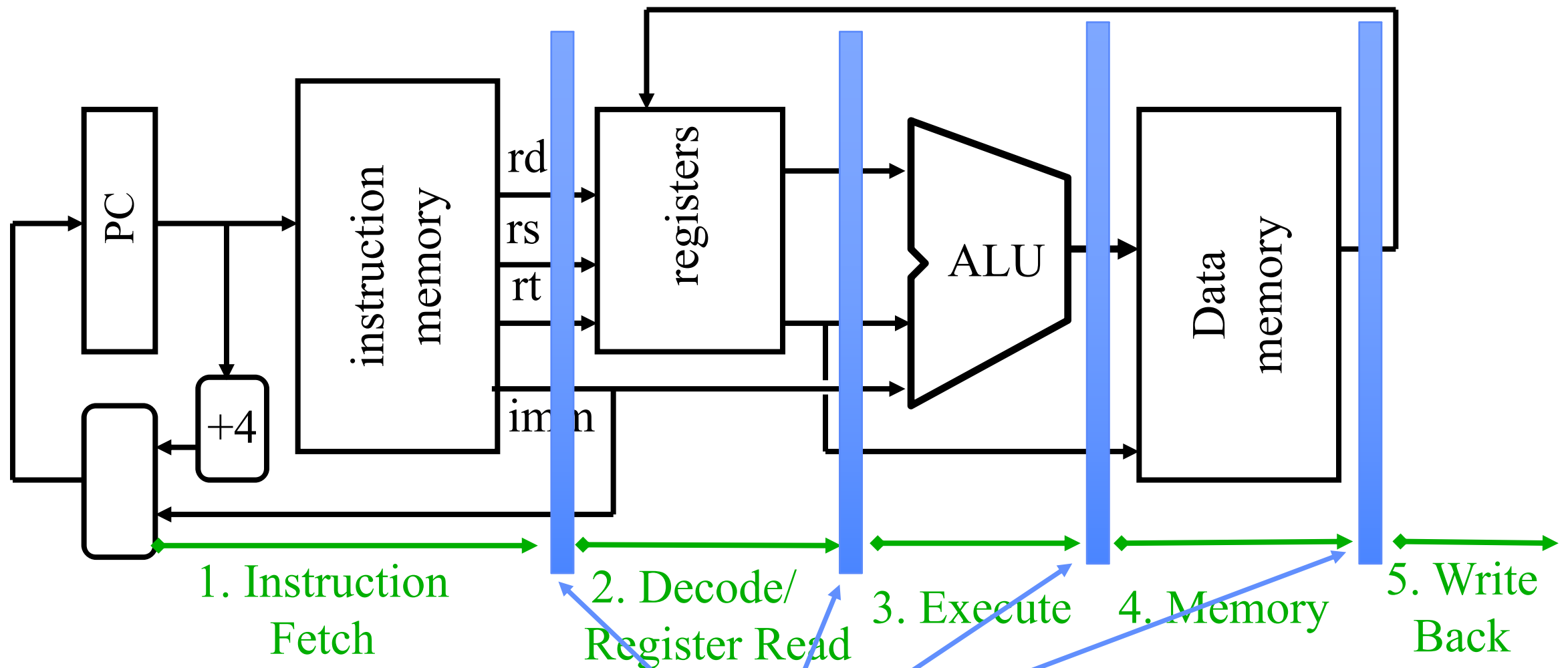
那么, 流水线的加速比是 $810ps / 240ps = 3.4 \text{ times}$.

五级流水线的实现

Divide the single-cycle CPU into 5 stages



Pipeline registers



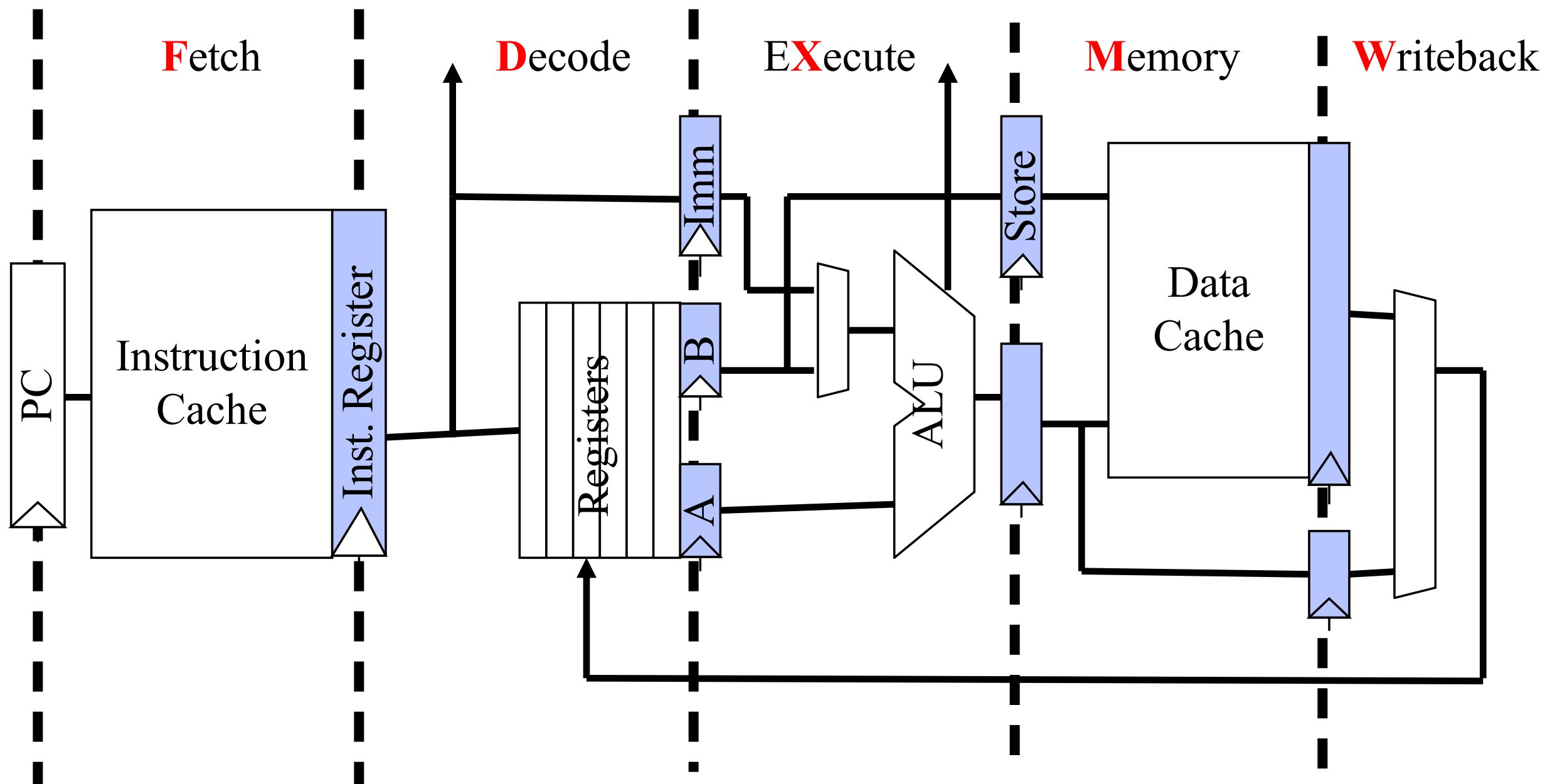
❖ 在不同阶段之间增加寄存器

➤ 保存前一个周期产生的信息

❖ **5 阶段流水线**

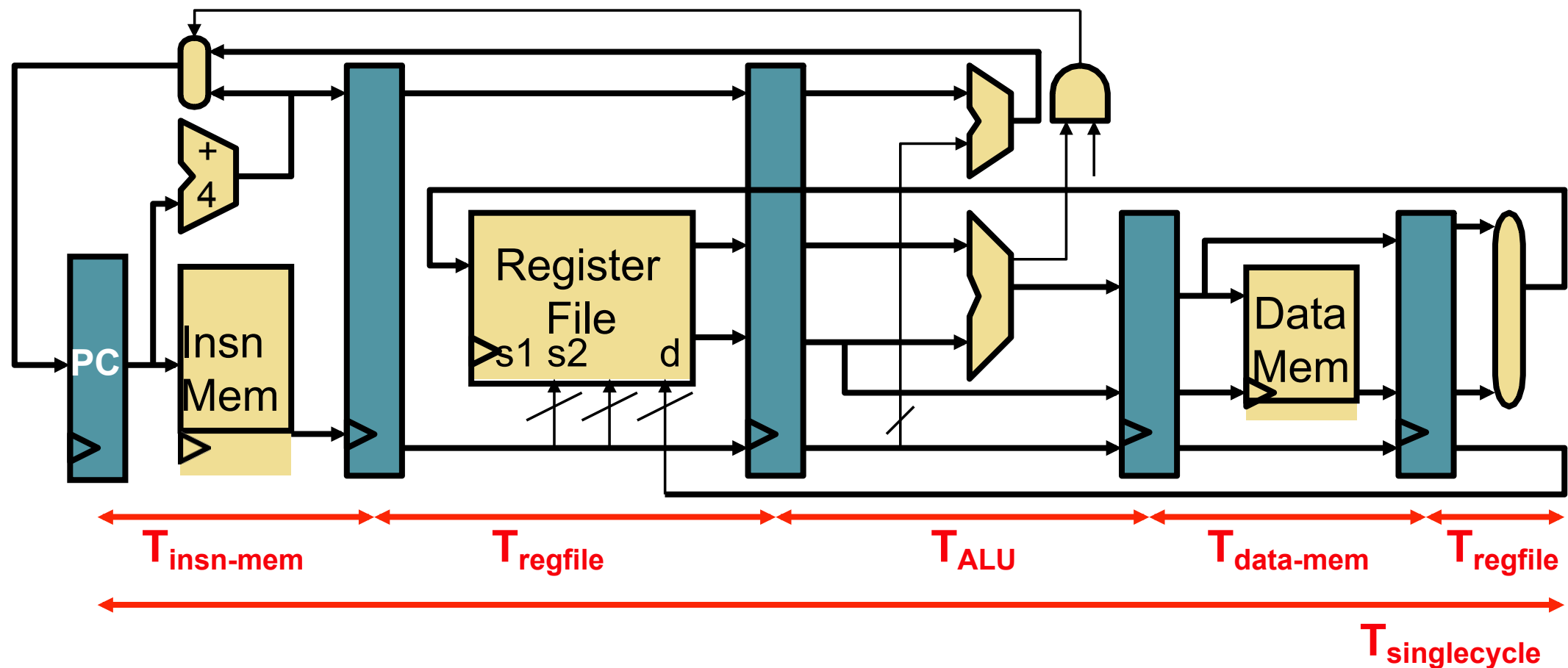
➤ 意味着时钟频率实际上变为原来的**5倍**

Classic 5-Stage RISC Pipeline



请注意，为了保持流水线充满，我们必须在每个时钟周期获取一条新指令，因此，每个时钟周期PC要更新 $PC = PC + 4$ 。

5 Stage Pipeline



- **Pipelining:** 把数据通路划分为N阶段 (here 5)

- 每个时钟周期每条指令在一个阶段

(Pentium 4 had 22 stages!)

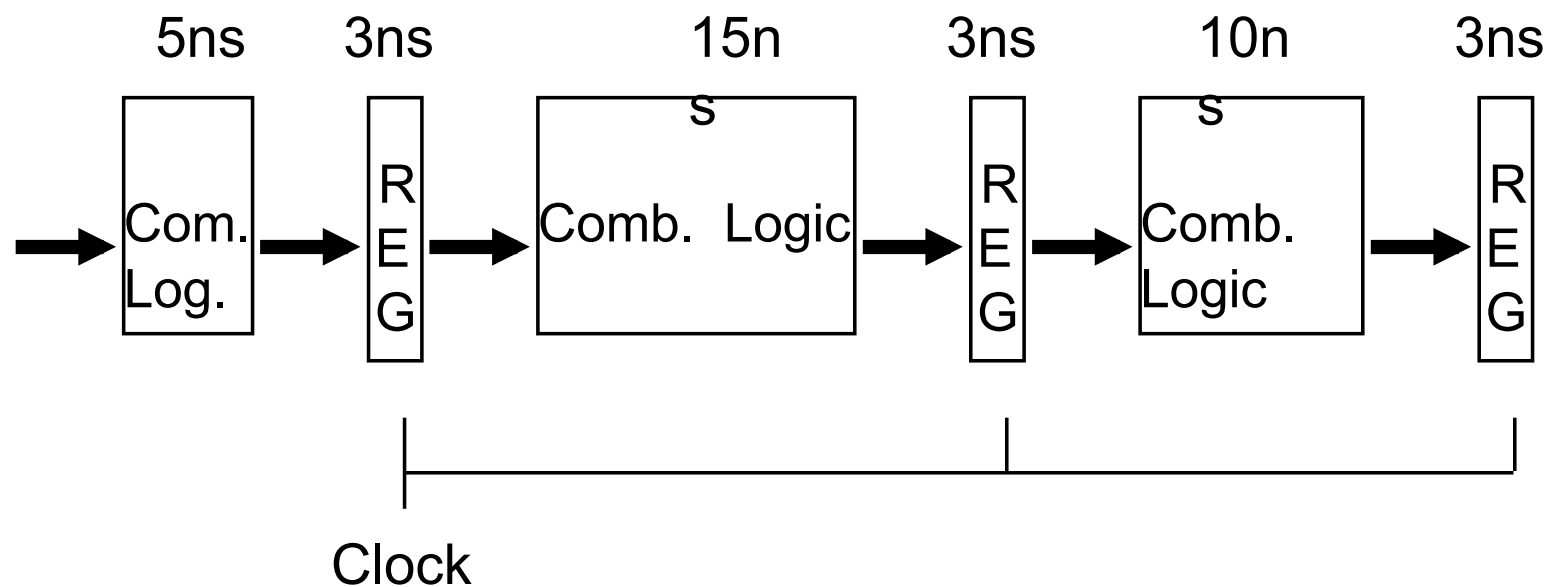
时钟周期 = $\text{MAX}(T_{\text{insn-mem}}, T_{\text{regfile}}, T_{\text{ALU}}, T_{\text{data-mem}})$

基本 CPI = 1: 每个周期一条指令进入, 同时一条指令离开

实际 CPI > 1: 流水线经常阻塞“stall”

- 单条指令的执行时间有所增加 (流水线开销)

Nonuniform Pipelining非均匀流水线



$$T_{\text{clock}} = \text{MAX}(\text{Comb. Logic}) + T_{\text{REG}}$$

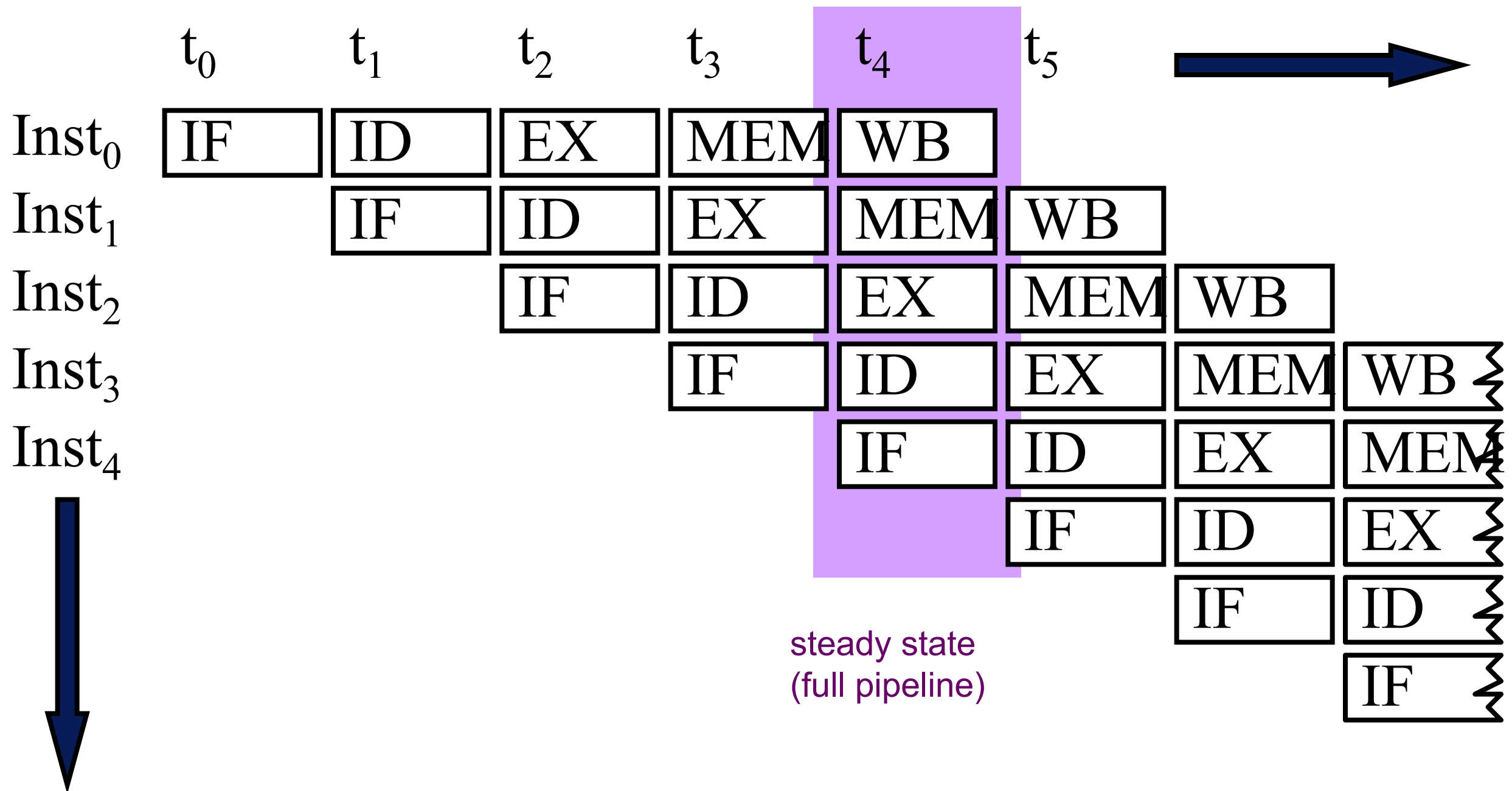
$$= 15 + 3 = 18 \text{ ns}$$

$$\text{Delay} = 18 * 3 = 54 \text{ ns}$$

$$\text{Throughput} = 55 \text{ MHz} = 1/18 \text{ ns}$$

- 吞吐量受最慢阶段限制，延迟由时钟周期 * 阶段数决定
- 必须尝试平衡阶段

流水线操作说明：操作视角



流水线操作说明：资源视角

	t ₀	t ₁	t ₂	t ₃	t ₄	t ₅	t ₆	t ₇	t ₈	t ₉	t ₁₀
IF	I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	I ₈	I ₉	I ₁₀
ID		I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	I ₈	I ₉
EX			I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	I ₈
MEM				I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇
WB					I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆

每个周期最多有五个功能部件同时在工作，理想情况下，有：

每个周期有一条指令进入流水线，每个周期都有一条指令完成，每条指令的有效周期(CPI)为1

指令流水线: 不是理想化的流水线

■ 操作不理想!

⇒ 指令种类不同 → 不是所有的指令都需要相同的阶段, 比如R
指令不需要访存

强制不同的指令经过相同的流水线阶段

→ 对有些指令在流水线某个阶段不做任何事。

■ 每个阶段是不一致的!

⇒ 不同的流水线阶段 → 延时时间不同

需要每个阶段由相同的时钟控制

→ 虽然有些阶段用时较少, 但仍然需要相同的时钟周期 (以最慢的为准)

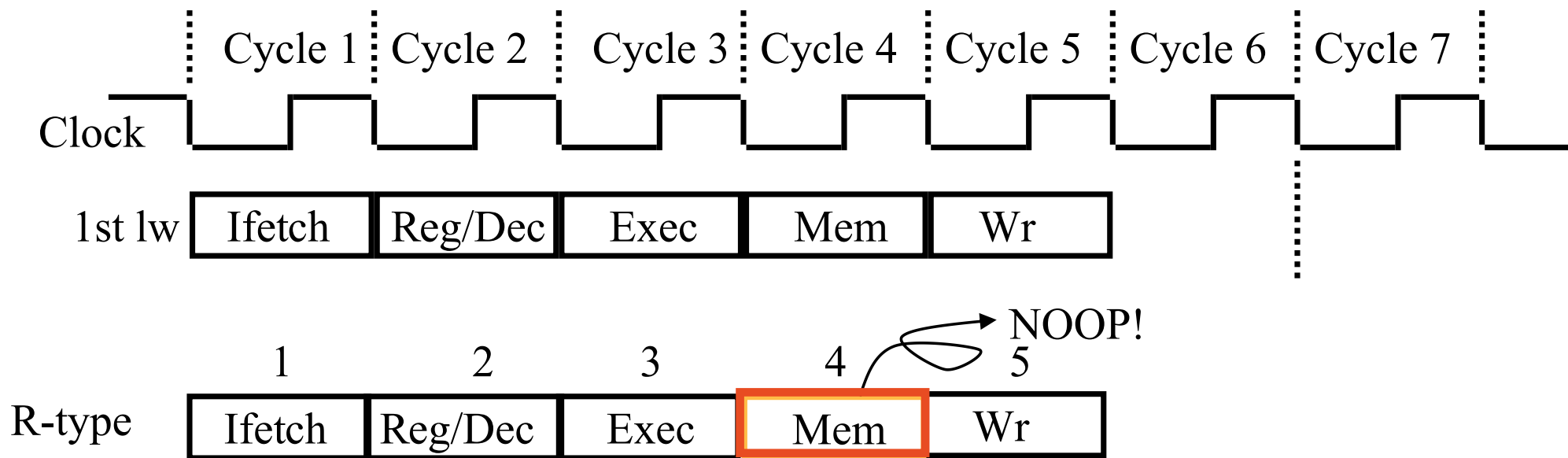
■ 指令之间不是独立的!

⇒ 指令之间有依赖性

需要检测 and 解决指令的依赖性, 以保证流水线的正确结果

→ 流水线阻塞 (流水线会停顿)

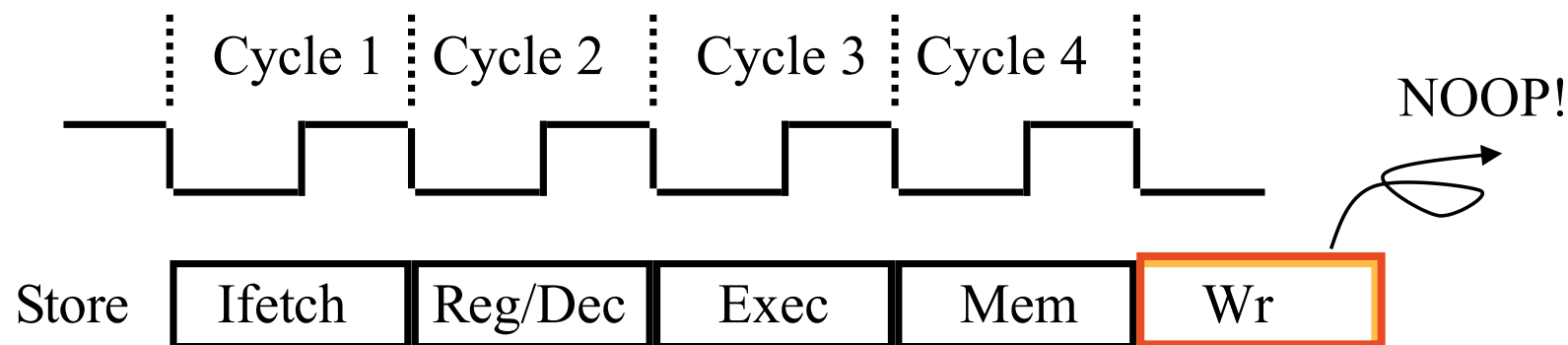
Load指令的流水线



R-type指令的第4阶段加一个NOP阶段以延迟“写”操作:

把“写”操作安排在第5阶段, 这样使R-Type的Mem阶段为空NOP

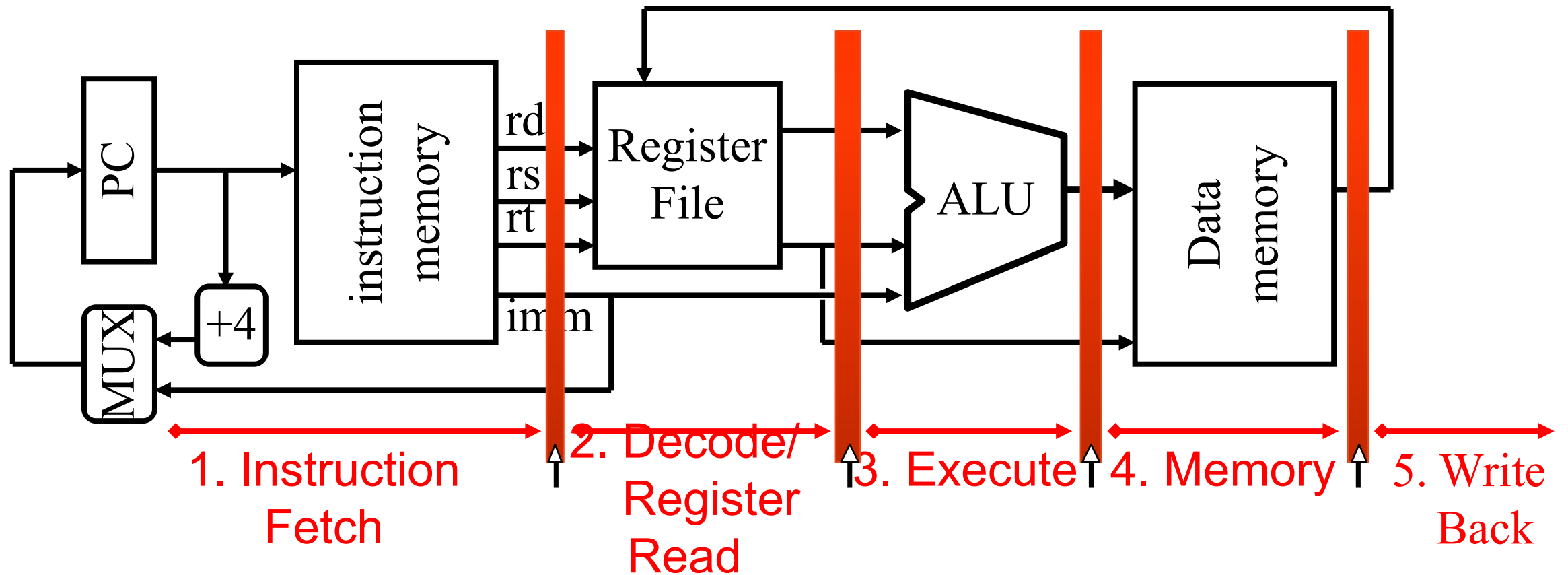
Store指令的没有最后的写寄存器操作



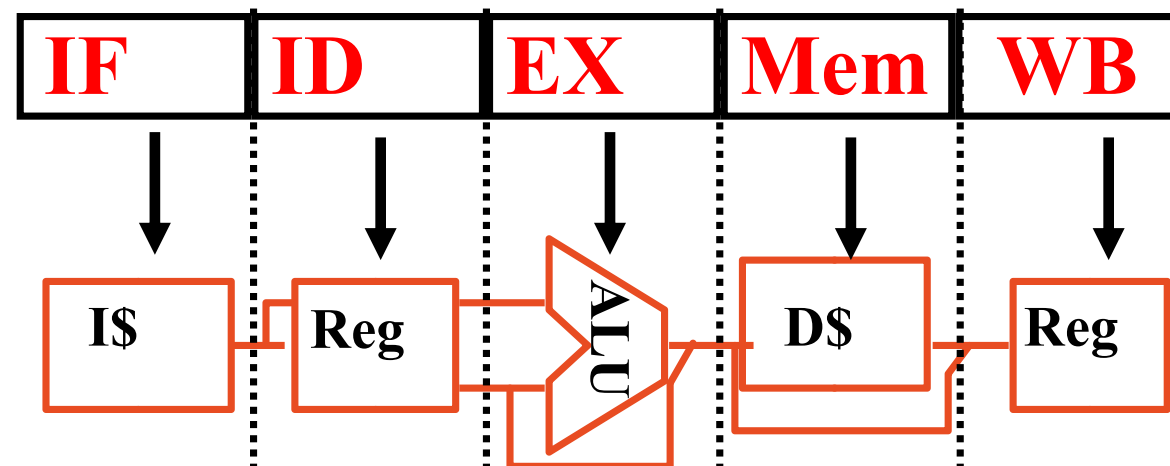
Store指令加一个空的写阶段, 使流水线更规整!

这样使流水线中的每条指令都有相同多个阶段, 且相同的阶段做相同的事!

Graphical Pipeline Diagrams



◦ Use datapath figure below to represent pipeline:



Pipelining the Fetch Phase

- 请注意，为了保持流水线充满，我们必须在每个时钟周期获取一条新指令
- 因此，每个时钟周期PC要更新

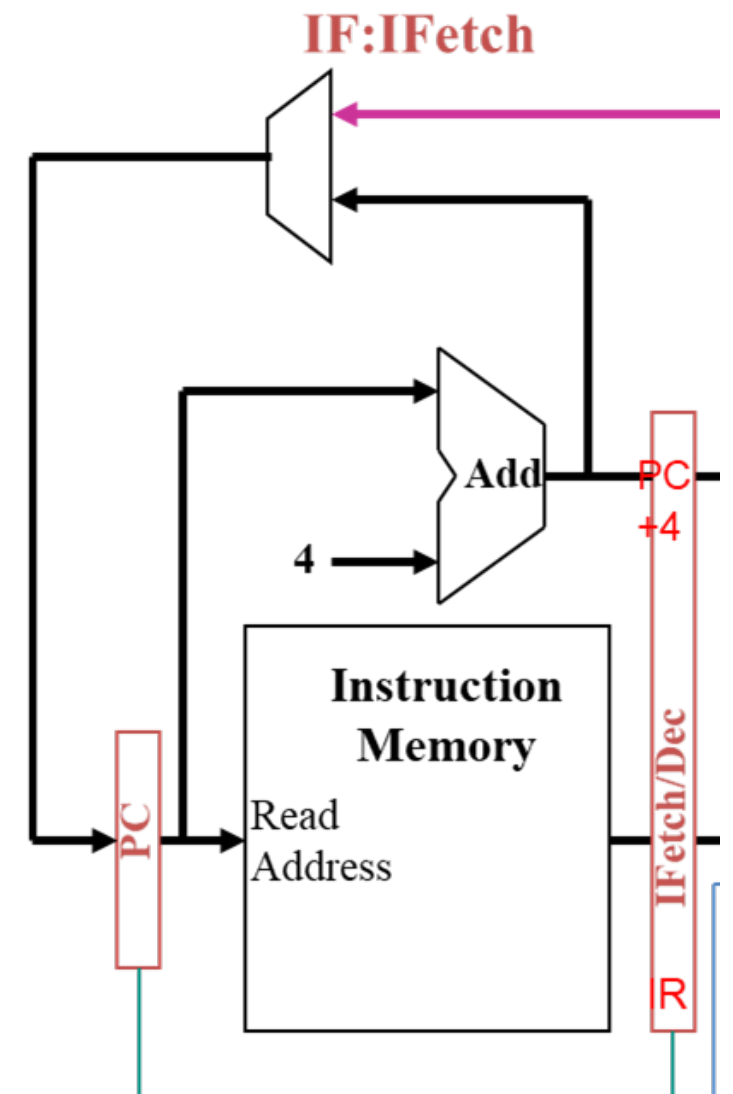
$$PC = PC + 4$$

写入流水线寄存器：

$$IR \leftarrow \text{Mem}[PC]$$

$$PC+4 \leftarrow PC + 4$$

Fetch

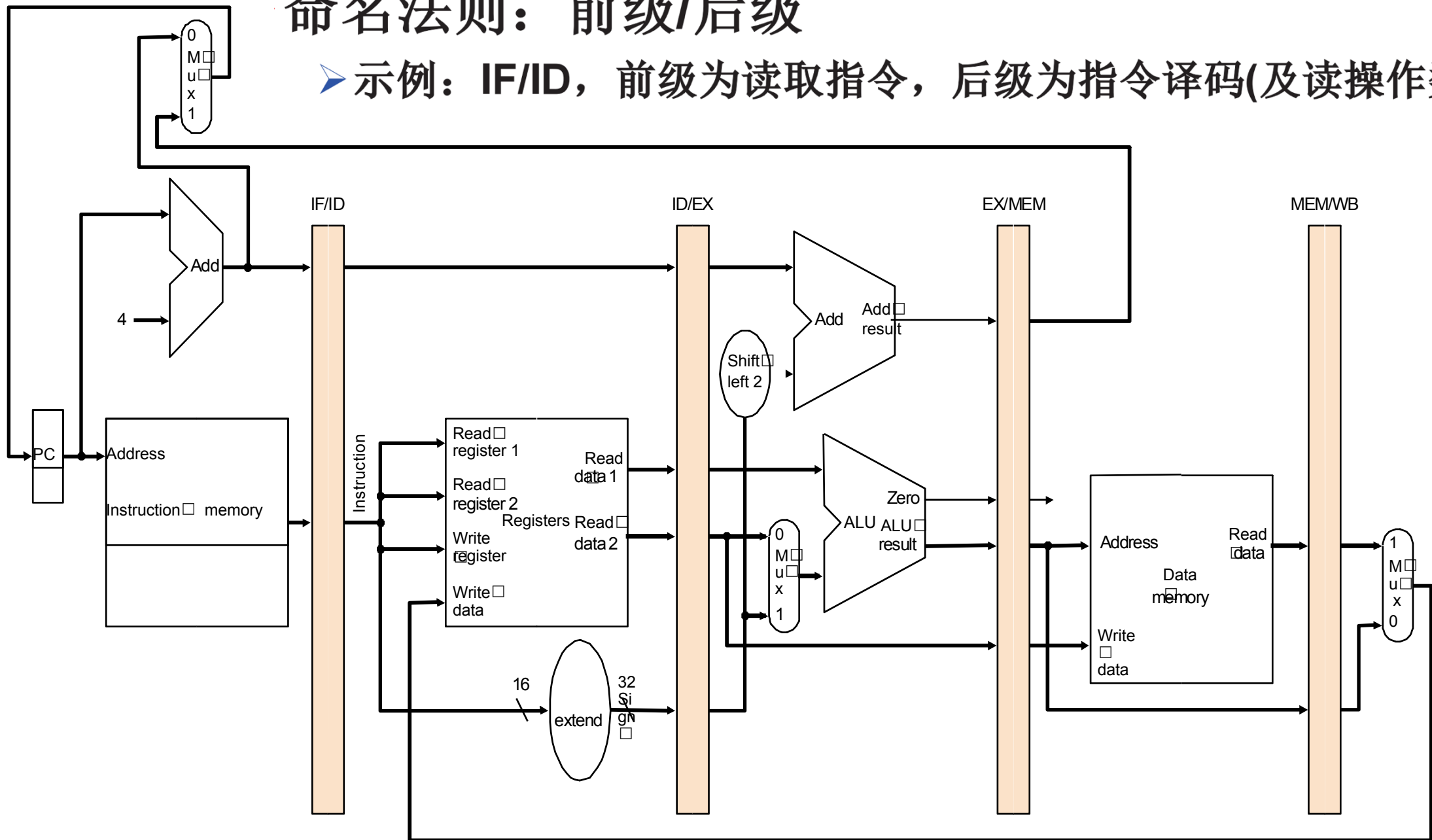


- 我们将只考虑基本（简单）流水线控制，稍后处理与分支和数据冒险相关的问题
- PC 和流水线寄存器在每个时钟周期更新，因此这些寄存器不需要单独的写使能信号

Pipelined Datapath

命名法则：前级/后级

➤ 示例：IF/ID，前级为读取指令，后级为指令译码(及读操作数)



功能：时钟上升沿到来时，保存前级结果；之后输出至下级组合逻辑

Stage 2: 译码

► 对操作码译码

为后面阶段建立控制信号

► 从寄存器堆读取操作数

regA (rs)和 regB(rb) 的寄存器地址
由指令给出

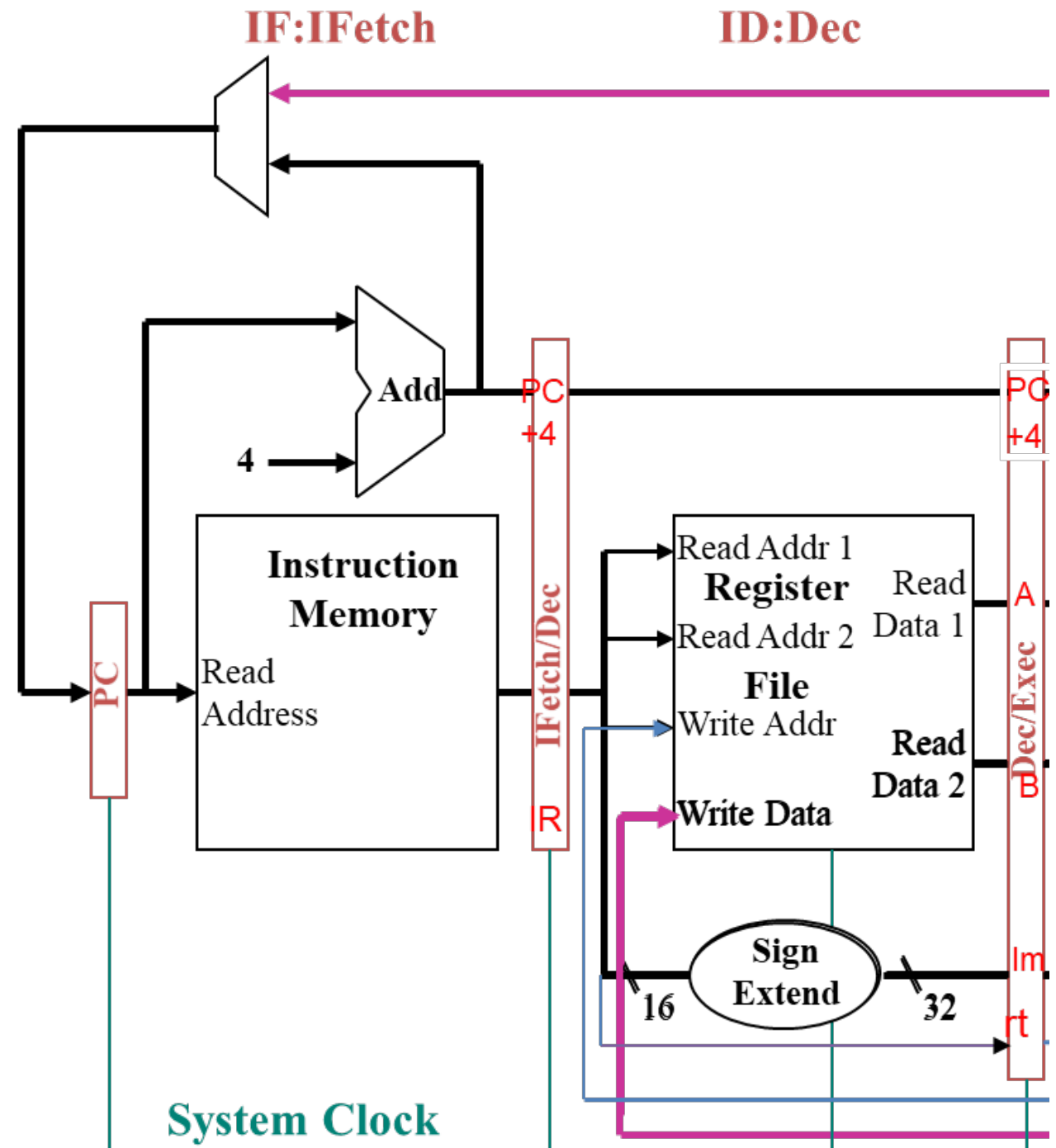
► 写入流水线寄存器(ID/EX)相关状态值

❑ 读出的寄存器值($A \leftarrow \text{Regs}[\text{rs}]$,
 $B \leftarrow \text{Regs}[\text{rt}]$)

❑ $\text{Imm} \leftarrow \text{sign-extend of}$

❑ 目的寄存器地址

❑ $\text{PC}+4$



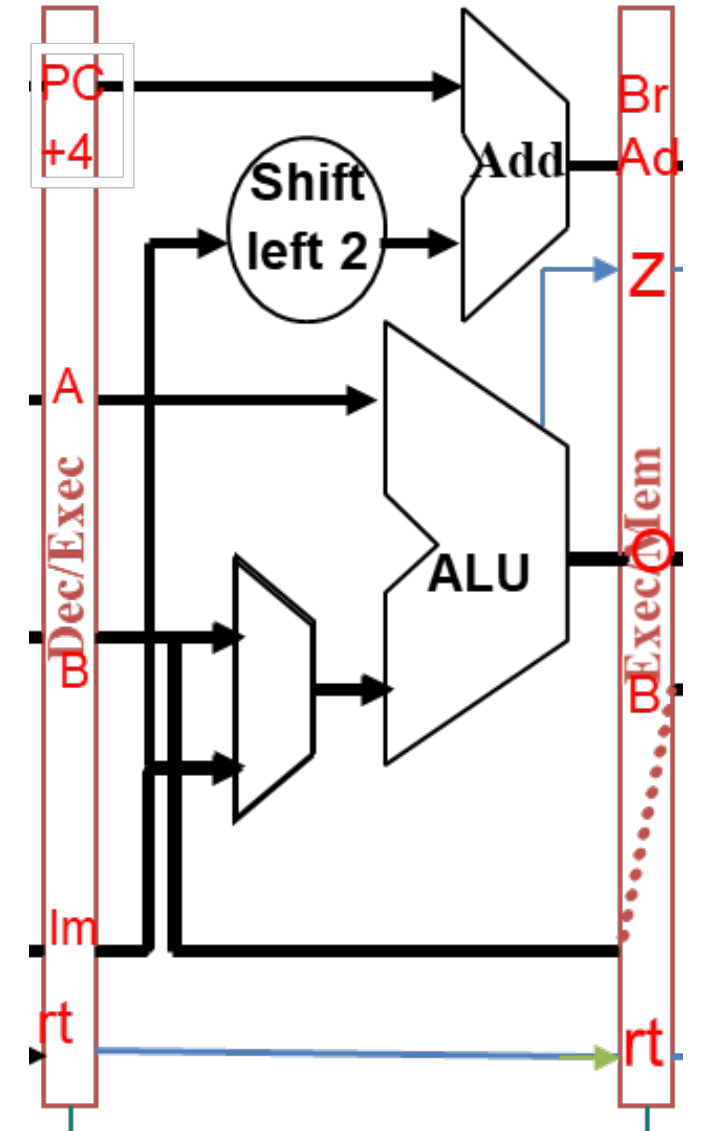
Stage 3: Execute

完成 ALU 的操作.

- ▶ 输入操作数Input 操作数可以是:
 - regA(rs) 或 RegB(rb)寄存器的内容
 - 指令中的立即数

Branches条件分支指令:计算 $PC+4+offset$

- ▶ 写入状态值到流水线寄存器 (EX/Mem)
- ▣ ALU 运算结果, 寄存器rt的内容 (sw指令需要) 和 $PC+4+offset$
- ▣ 目的寄存器地址及控制信号



Stage 4: Memory Operation

内存访问 (lw/sw指令)

- ❑ ALU 结果包含 **lw** 和 **sw** 的访问地址
- ❑ 控制信号包含内存访问的读写信号

写状态值到下一级流水线寄存器 (**Mem/WB**)

- ❑ ALU 结果或内存读取的数据
- ❑ 目的及寄存器地址及控制信号等

Stage 5: Write back

结果写入寄存器堆 (if required)

- ❑ 写内存读取数据到目的寄存器 (rt) (对于lw 指令)
- ❑ 写 ALU 结果到目的寄存器(rd/rs) (对于运算指令)
- ❑ 控制信号 (寄存器写使能等)

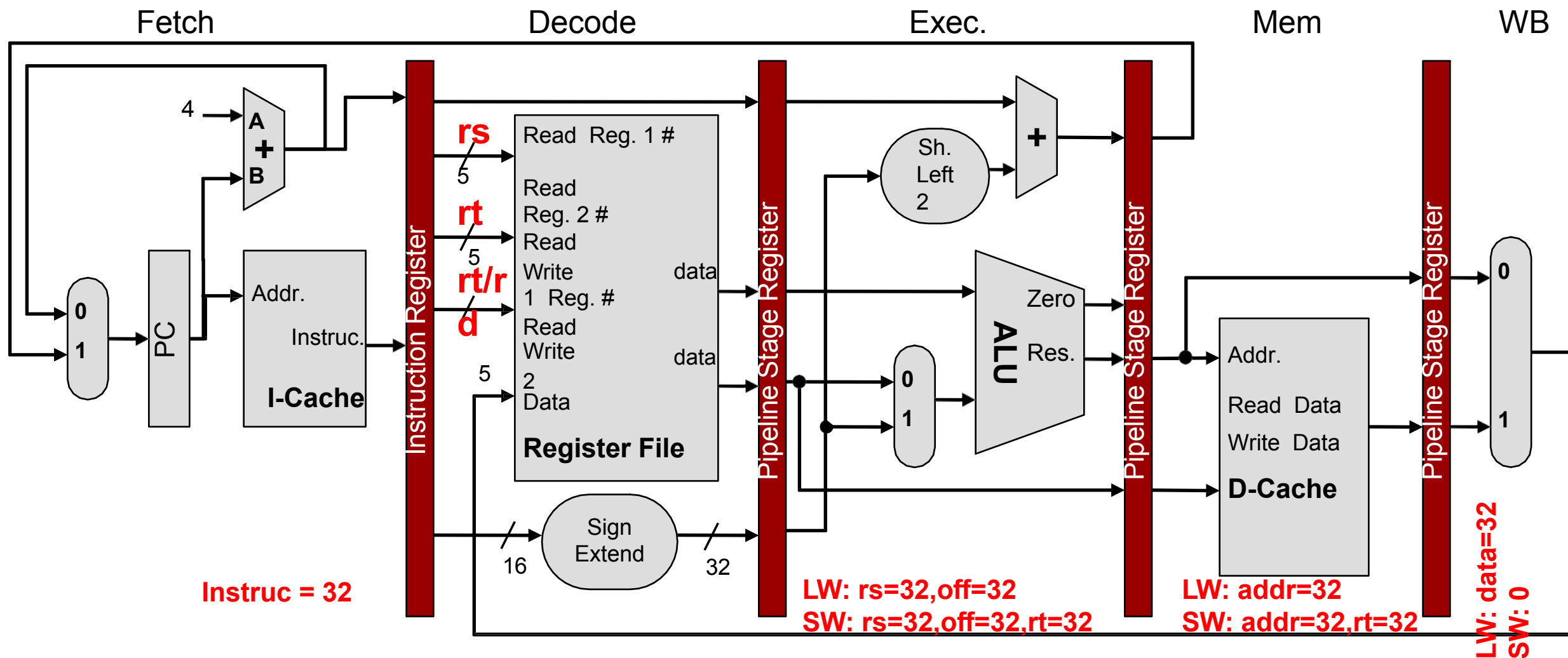
流水线各阶段操作细节

Stage	Any Instruction		
IF	IF/ID.IR \leftarrow MEM[PC]; IF/ID.NPC \leftarrow PC+4 PC \leftarrow if ((EX/MEM.opcode=branch) & EX/MEM.cond) {EX/MEM.ALUoutput} else {PC + 4}		
ID	ID/EX.A \leftarrow Regs[IF/ID.IR[Rs1]]; ID/EX.B \leftarrow Regs[IF/ID.IR[Rs2]] ID/EX.NPC \leftarrow IF/ID.NPC; ID/EX.Imm \leftarrow extend(IF/ID.IR[Imm]); ID/EX.Rw \leftarrow IF/ID.IR[Rs2 or Rd]		
	ALU Instruction	Load / Store	Branch
EX	EX/MEM.ALUoutput \leftarrow ID/EX.A func ID/EX.B, or EX/MEM.ALUoutput \leftarrow ID/EX.A op ID/EX.Imm	EX/MEM.ALUoutput \leftarrow ID/EX.A + ID/EX.Imm EX/MEM.B \leftarrow ID/EX.B	EX/MEM.ALUoutput \leftarrow ID/EX.NPC + (ID/EX.Imm \ll 2) EX/MEM.cond \leftarrow br condition
MEM	MEM/WB.ALUoutput \leftarrow EX/MEM.ALUoutput	MEM/WB.LMD \leftarrow MEM[EX/MEM.ALUoutput] or MEM[EX/MEM.ALUoutp ut] \leftarrow EX/MEM.B	Branch requires 3 cycles; R、Store requires 4 cycles, and all other instructions require 5 cycles.
WB	Regs[MEM/WB.Rw] \leftarrow MEM/WB.ALUOutput	For load only: Regs[MEM/WB.Rw] \leftarrow MEM/WB.LMD	

Basic 5 Stage Pipeline

- 每个流水线寄存器 **pipeline register** 容量怎样计算 (找出每条指令每个阶段的最大信息)
- 简单地, 只考虑 LW/SW 两条指令 (忽略控制信号)

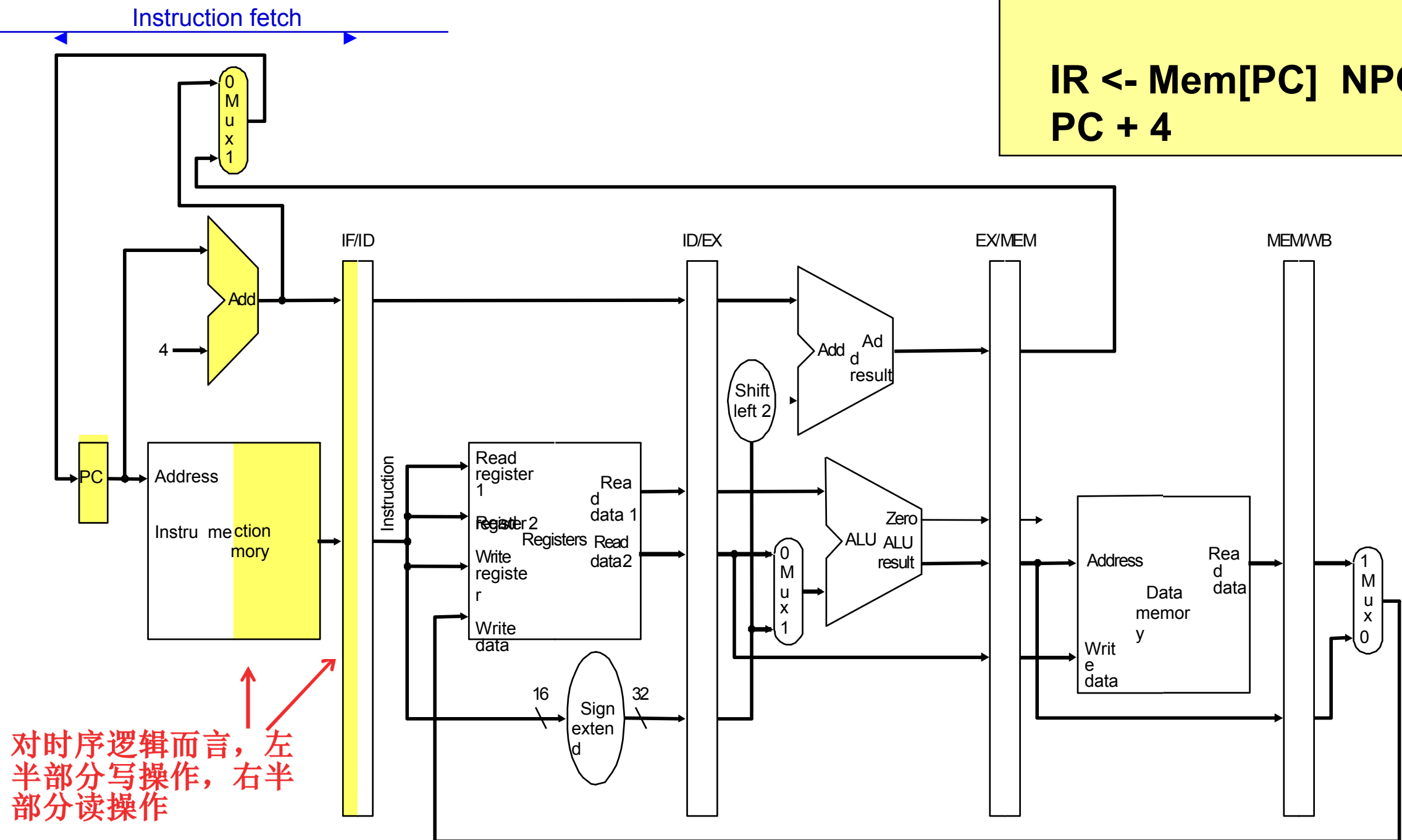
LW \$10,40(\$1)	Op = 35	rs=1	rt=10	immed.=40
SW \$15,100(\$2)	Op = 43	rs=2	rt=15	immed.=100



1w: Instruction Fetch (IF)

Passed To Next Stage

$IR \leftarrow \text{Mem}[\text{PC}] \quad \text{NPC} \leftarrow \text{PC} + 4$



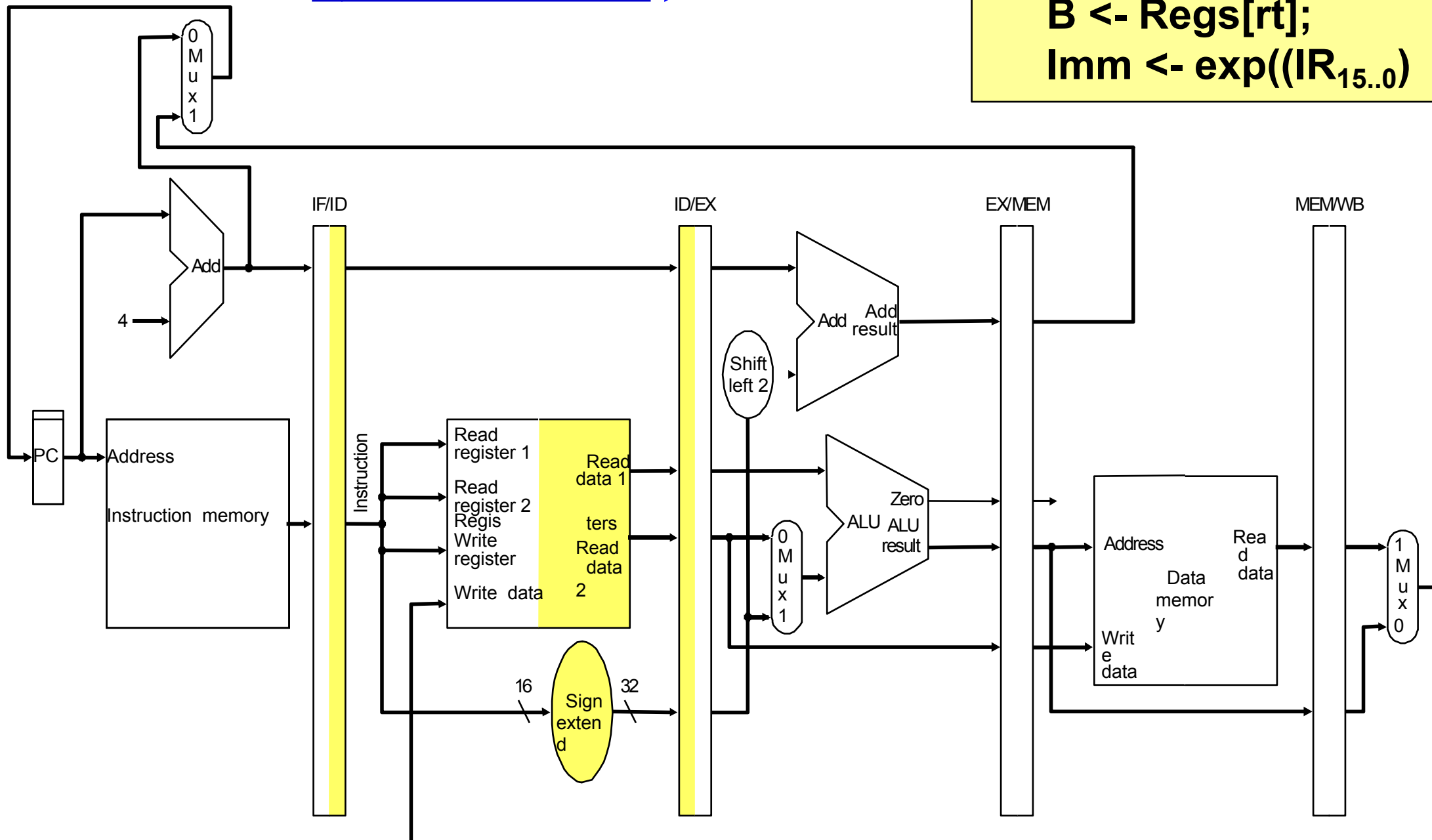
- 阴影表示部件正在被使用部分
- 寄存器堆或内存右边的阴影 right half (ID or WB) 表示这个部件这阶段正在读操作 read
- 阴影左边 left half 表示部件在这阶段正在写操作 written

lw: Instruction Decode (ID)

← Instruction decode →

Passed To Next Stage

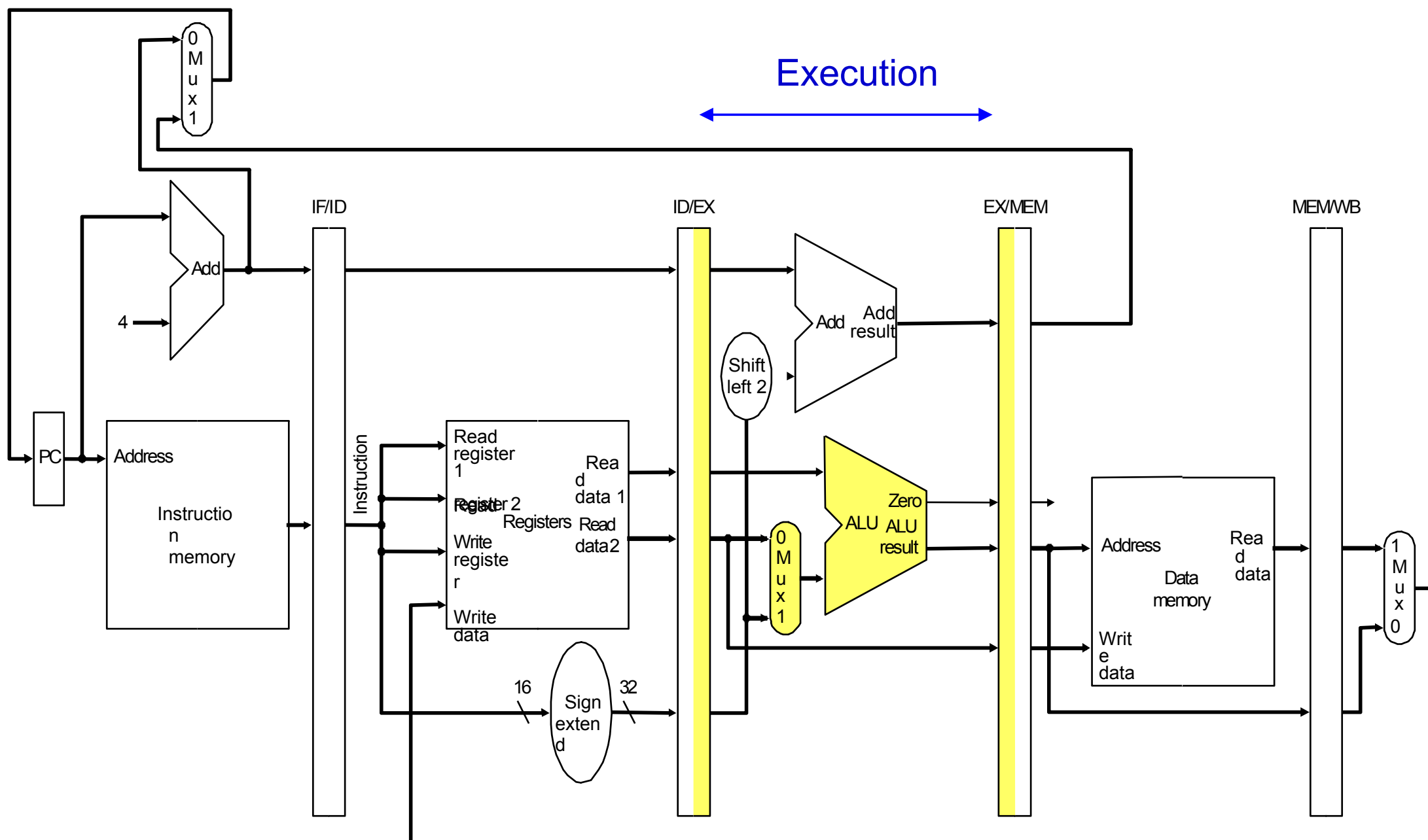
$A \leftarrow \text{Regs}[rs];$
 $B \leftarrow \text{Regs}[rt];$
 $\text{Imm} \leftarrow \text{exp}((\text{IR}_{15..0}))$



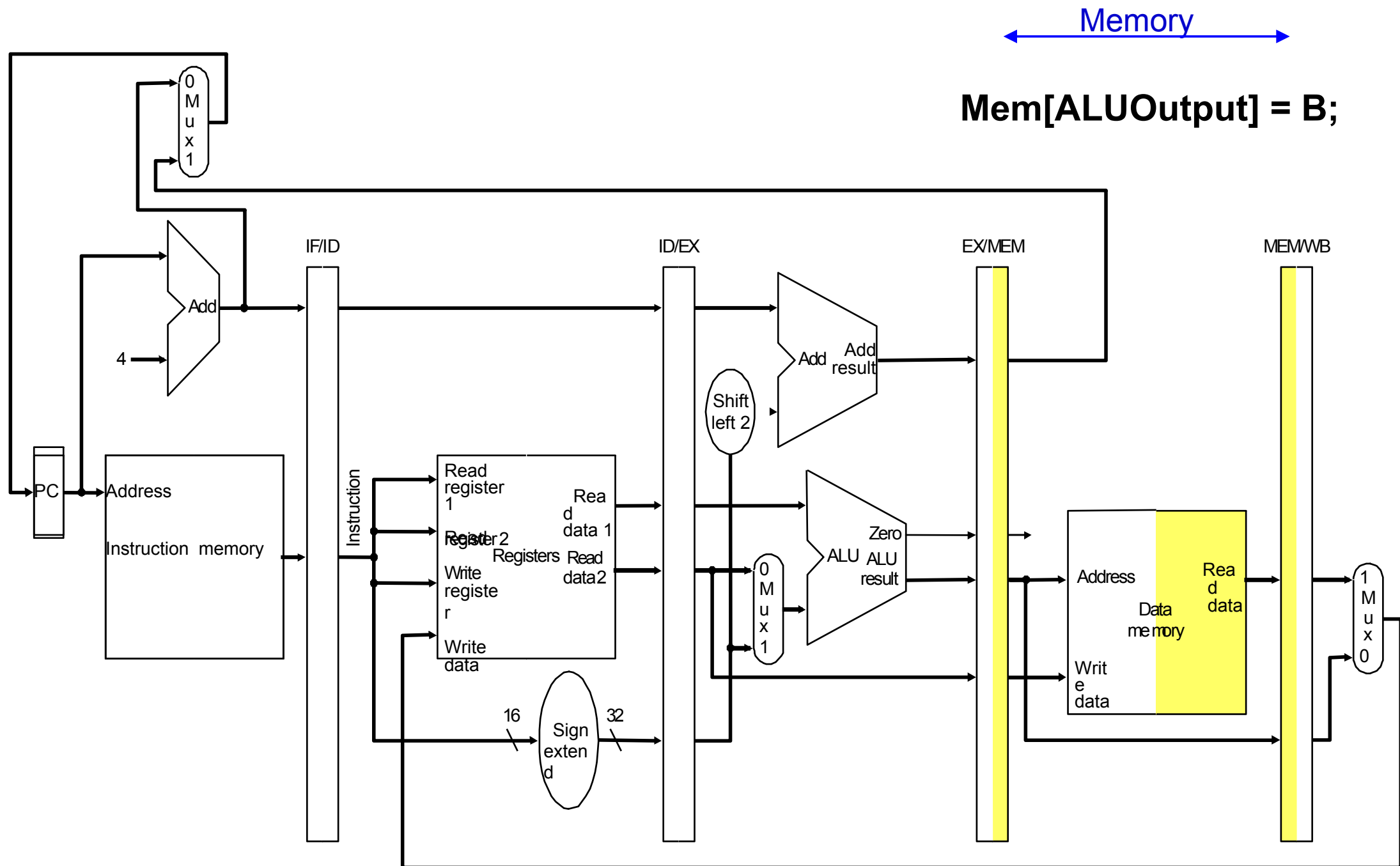
1w: Execution (EX)

Passed To Next Stage

$\text{ALUOutput} \leftarrow A + \text{Imm};$

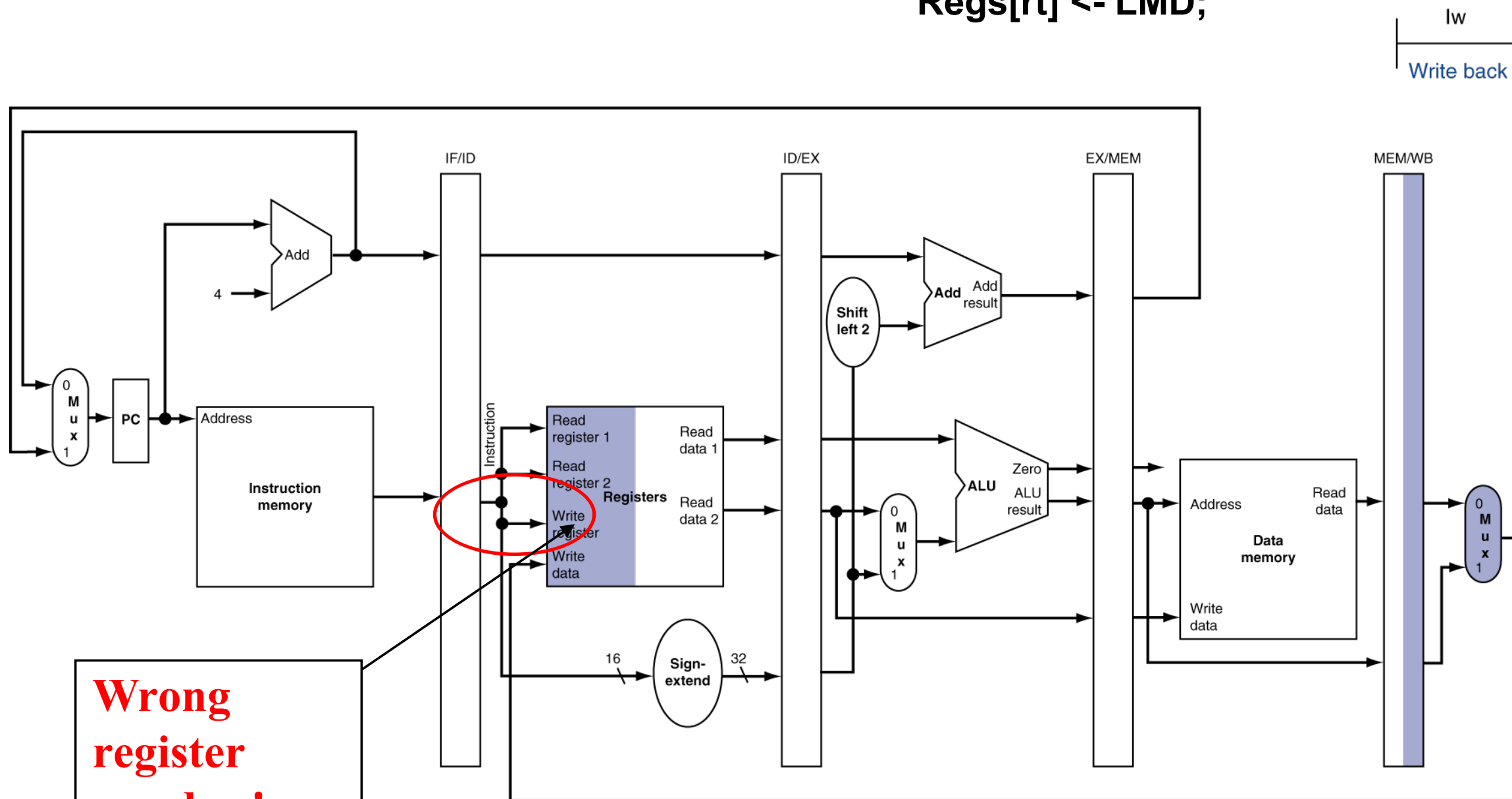


1w: Memory (MEM)



WB for Load – 这里有些不对头!

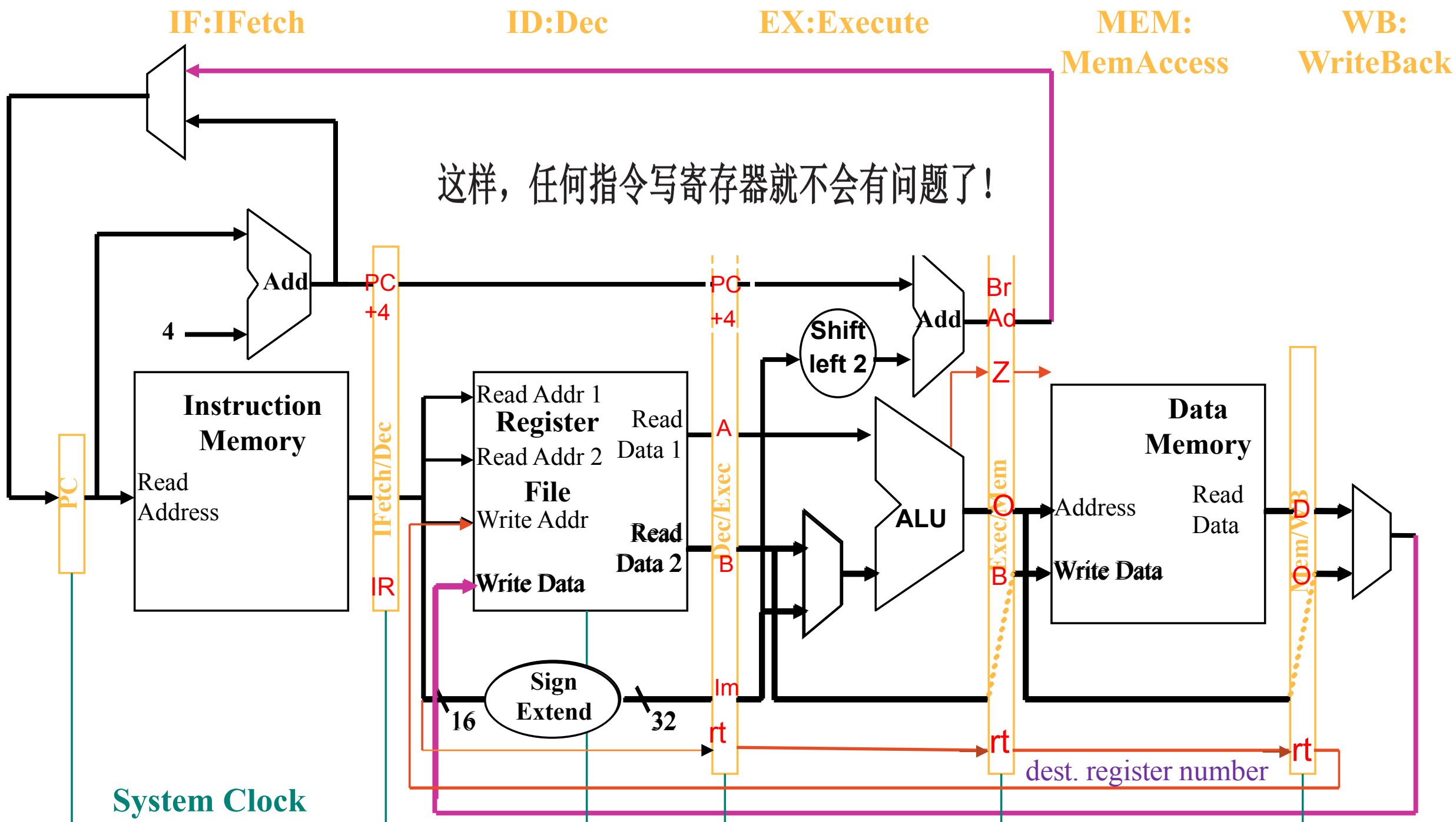
Regs[rt] <- LMD;



**Wrong
register
number!
这里的寄存
器号不对**

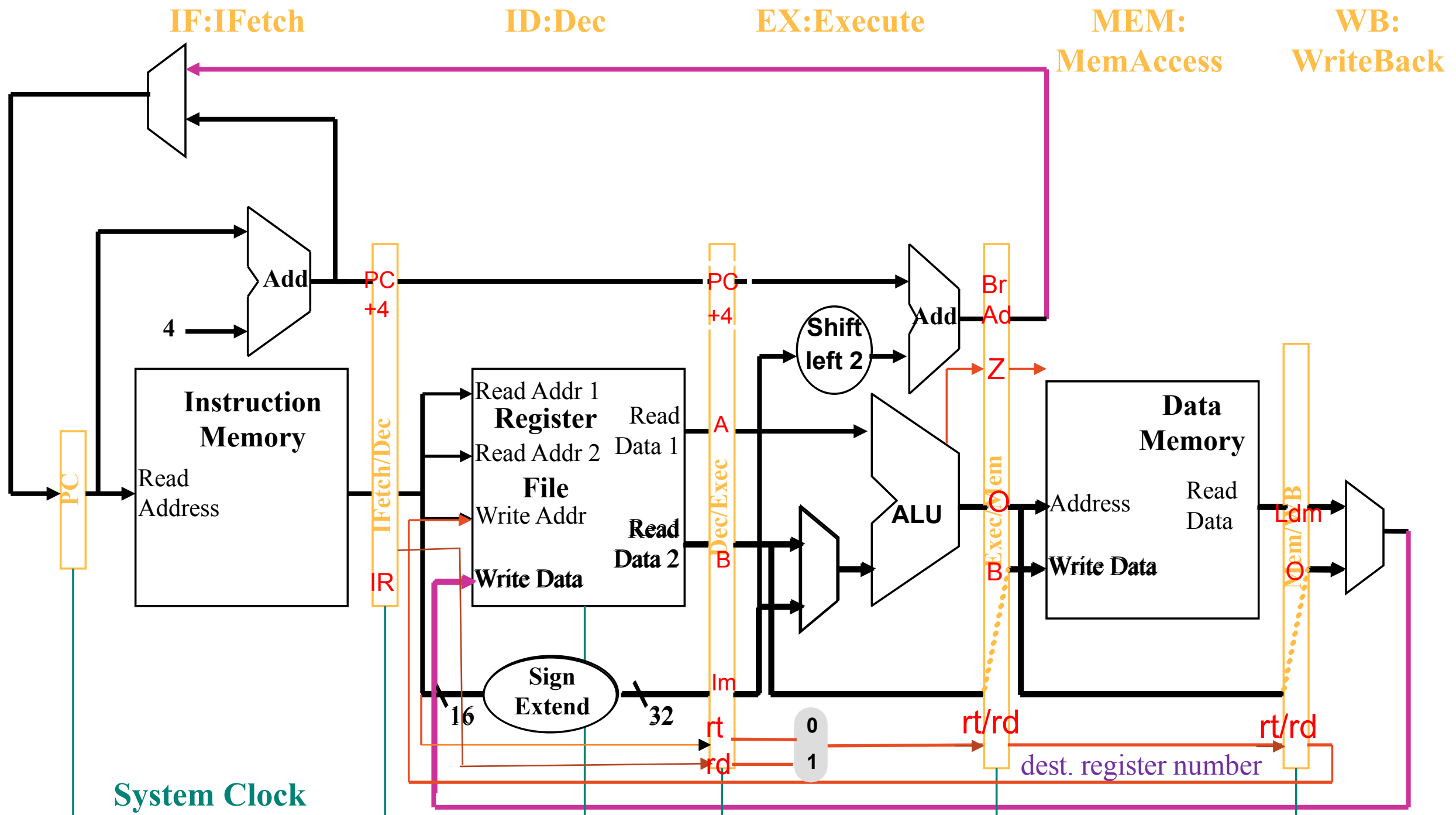
WB for Load Datapath Modifications

对 MIPS 数据通路修正，rt 随流水线寄存器传送



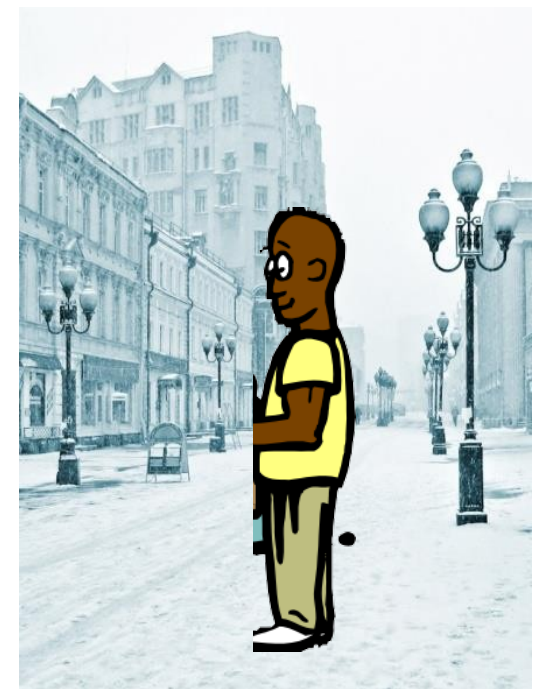
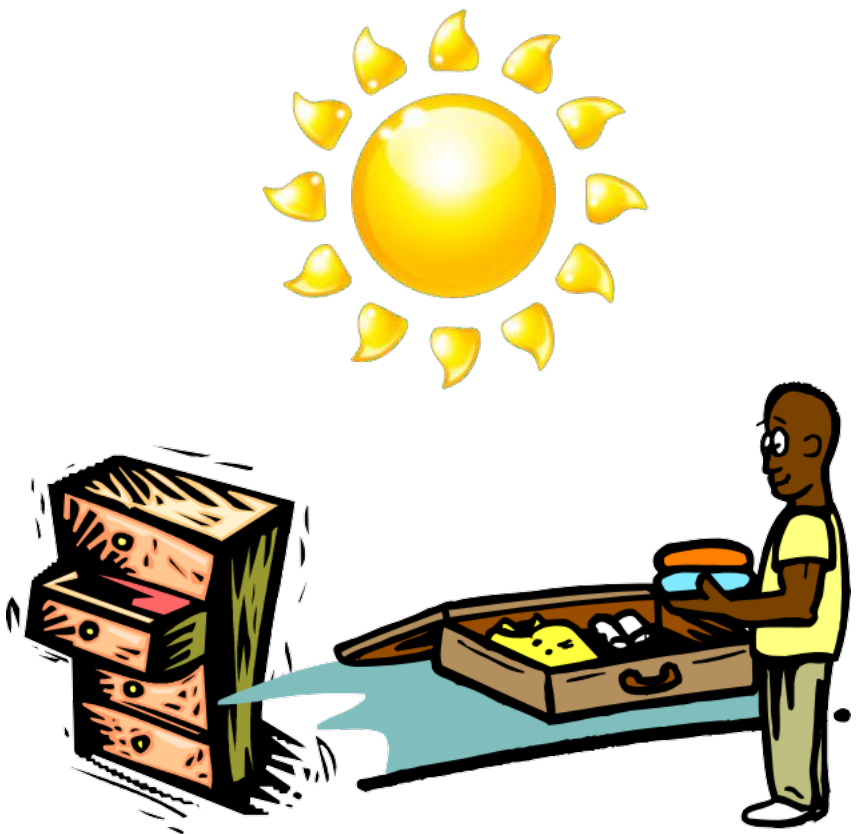
MIPS Pipeline Datapath Modifications

对 MIPS 数据通路修正，目标寄存器有可能是 rt 或 rd,在译码后通过2选一选择后随流水线寄存器传送



Pipeline Packing List

- 就像你去旅行时，你必须提前打包你需要的所有东西，因为你不能回到你的壁橱里，所以在流水线中，你必须随身携带所有需要的控制和数据，直到你使用它



How many bits wide is each pipeline register?

- PC – 32 bits
- IF/ID – 64 bits(PC+IR)
- ID/EX – $9(\text{con}) + 32 \times 4 + 10 = 147$

(rs=32,pc=32,off=32,rt=32,rtaddr=5,rdaddr=5)

- EX/MEM – $5(\text{con}) + 1 + 32 \times 3 + 5 = 107$

(zero=1,rtaddr=5,Aluout=32,Braddr=32,rt=32,rt/rdaddr=5)

- MEM/WB – $2(\text{con}) + 32 \times 2 + 5 = 71$

(Ldata=32,Aluout=32, rt/rdaddr=5)

时钟驱动的流水线时空图

- 行：某个时钟，指令流分别处于哪些阶段
- 列：某个部件，在时间方向上执行了哪些指令

在clk5后，指令流全部充满

所有部件都在执行指令，只是不同的指令

lw \$9, 0(\$7)

add \$5, \$9, \$10

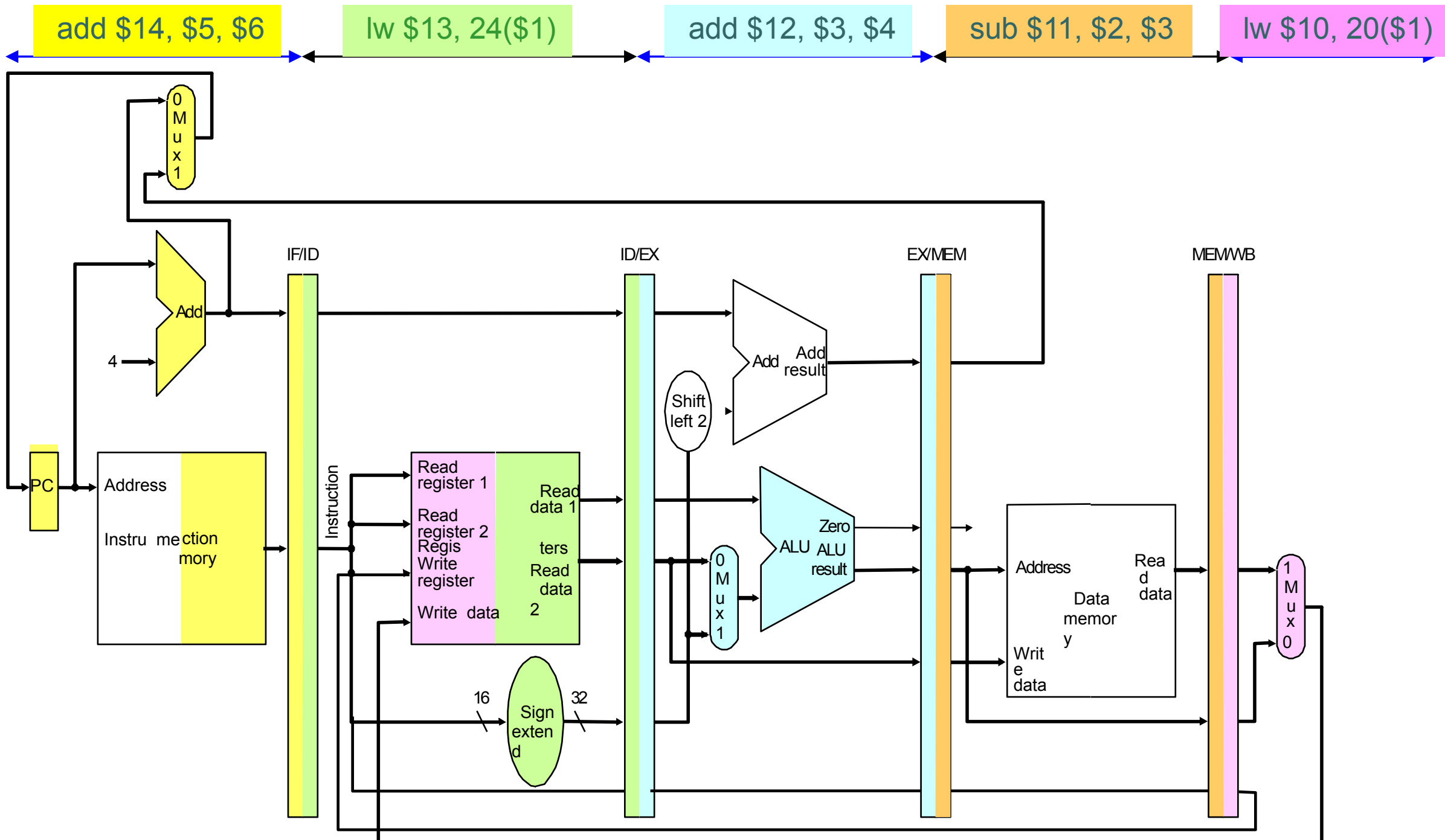
sub \$11, \$6, \$8

and \$12, \$6, \$7

or \$11, \$6, \$9

指令	Cycle #	1	2	3	4	5	6	7	8	9	10
lw	1	F	D	X	M	W					
add	2		F	D	X	M	W				
sub	3			F	D	X	M	W			
and	4				F	D	X	M	W		
or	5					F	D	X	M	W	

Pipelining Example



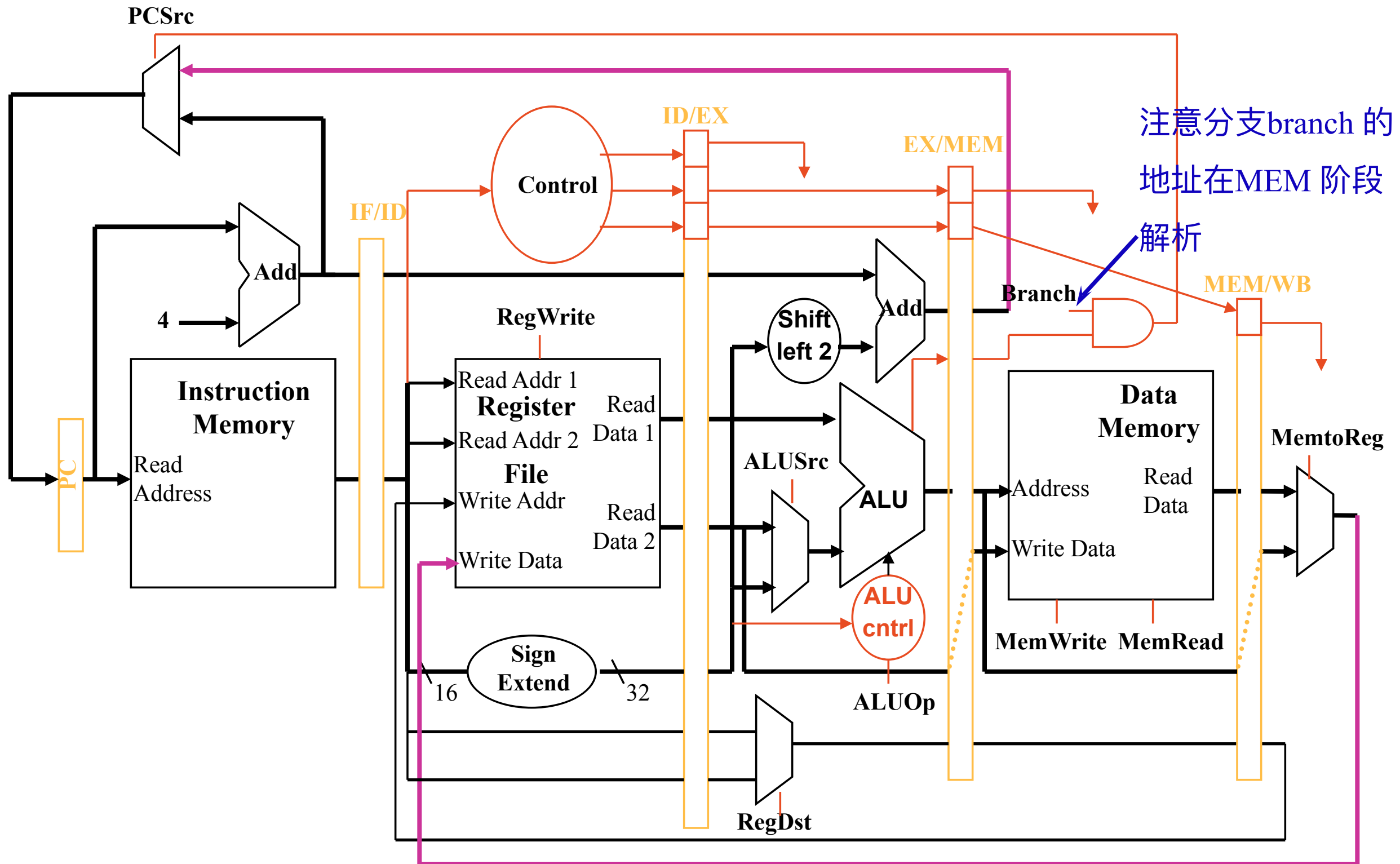
在clk5后, 指令流全部充满

Pipelined datapath:控制信号怎么办?

- 控制信号在RF/ID阶段产生—由控制逻辑电路译码生成
- 控制信号也必须流水线化
 - 针对EX阶段的控制信号
 - ExtOP, ALUSrc等；
 - 一个时钟周期后用到
 - 针对Memory的控制信号
 - MemWr, Branch；
 - 两个周期后用到
 - 针对寄存器回写的控制信号
 - MemToReg, RegWr等；
 - 三个周期后用到

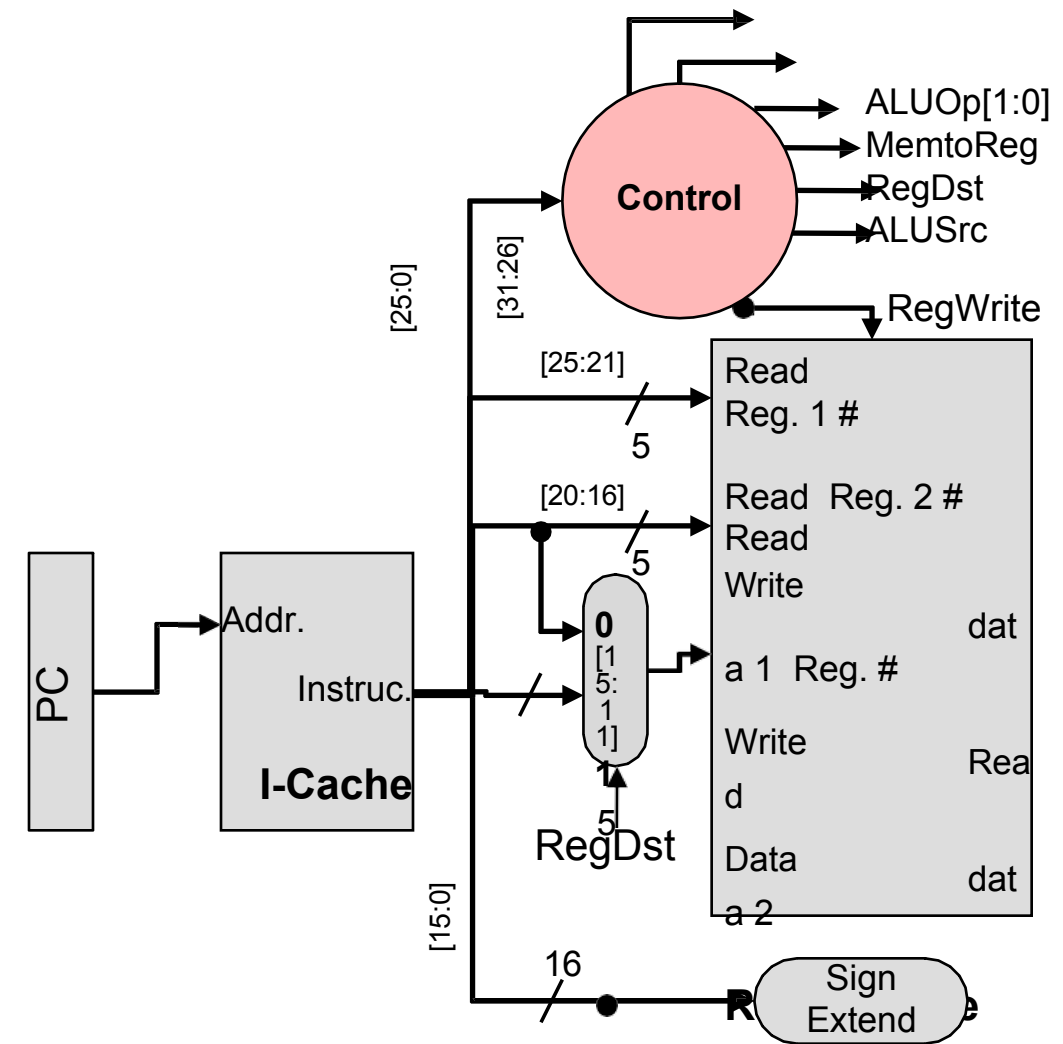
(只有这三个阶段有控制信号)

MIPS Pipeline Data and Control Paths



Control Signal Generation

- 回想一下单周期 CPU 讨论中没有状态机控制，而是一个简单的转换器（组合逻辑），将 6 位操作码转换为这 9 个控制信号
- 由于单周期和流水线 CPU 的数据通路本质上是相同的，因此控制也相同
- 主要区别在于控制信号在一个时钟周期内生成并用于后续周期（稍后的流水线阶段）
- 我们可以在 ID 阶段产生所有的信号，并使用流水线寄存器来存储它们，并将它们传递给使用它们的阶段



Control Settings

	EX Stage				MEM Stage			WB Stage	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Brch	Mem Read	Mem Write	Reg Write	Mem toReg
R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Control Signals per Stage

- How many control signals are needed in each stage

	Execution Stage = 4 signals (10 if you count function codes)				Mem stage = 3 signals			WB Stage = 2 signals	
Instruction	Reg Dst	ALU Src	ALU Op[1:0]	Func[5:0]	Branch	Mem Read	Mem Write	Reg Write	Memto-Reg
R-format	1	0	10	...	0	0	0	1	0
LW	0	1	00	X	0	1	0	1	1
SW	X	1	00	X	0	0	1	0	X
Beq	X	0	01	X	0	0	0	0	X

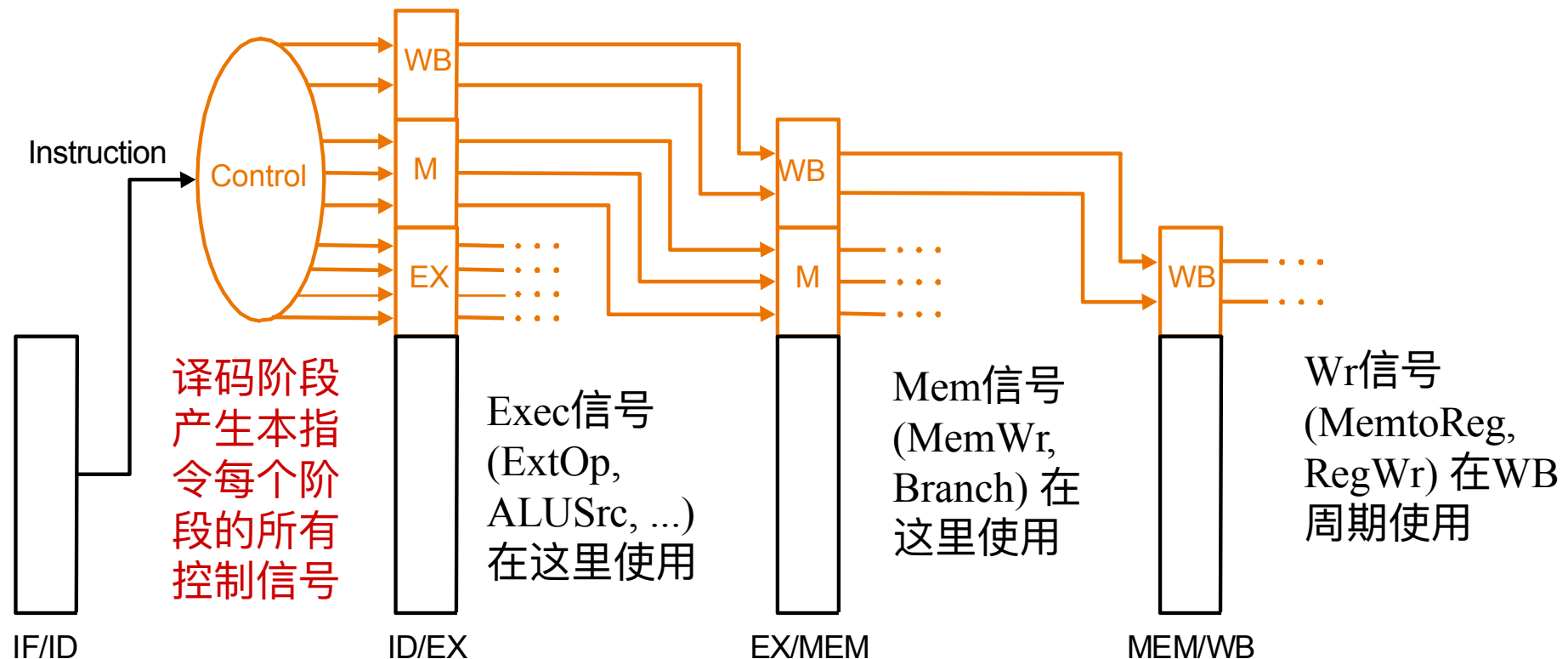
Pipeline Control

- 每个阶段 (IF, ID, EX, MEM, WB)需要控制什么?
 - ☐ IF: 取指 、 PC 递增
 - ☐ ID: 指令译码和从寄存器堆取操作数/或立即数
 - ☐ EX: Execution stage
 - RegDst
 - ALUOp[1:0]
 - ALUSrc
 - ☐ MA: Memory stage
 - Branch
 - MemRead
 - MemWrite
 - ☐ WB: Writeback
 - MemtoReg
 - RegWrite (注意这个信号在ID阶段的寄存器堆)

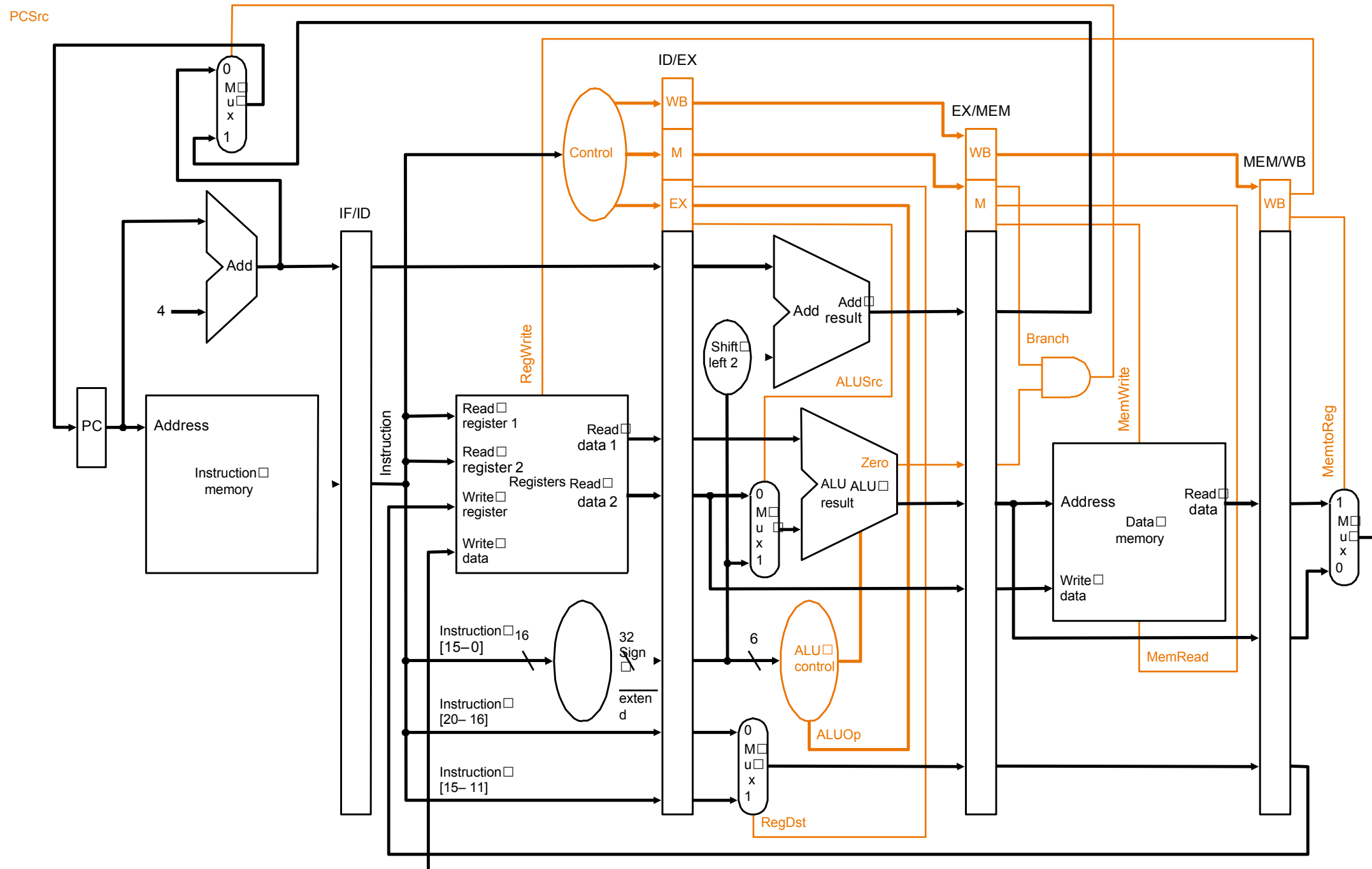
Pipeline Control

- ❑ 需要对流水线寄存器扩展，增加ID阶段产生的控制信息
- ❑ 像传递数据一样在流水线寄存器传递控制信号

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

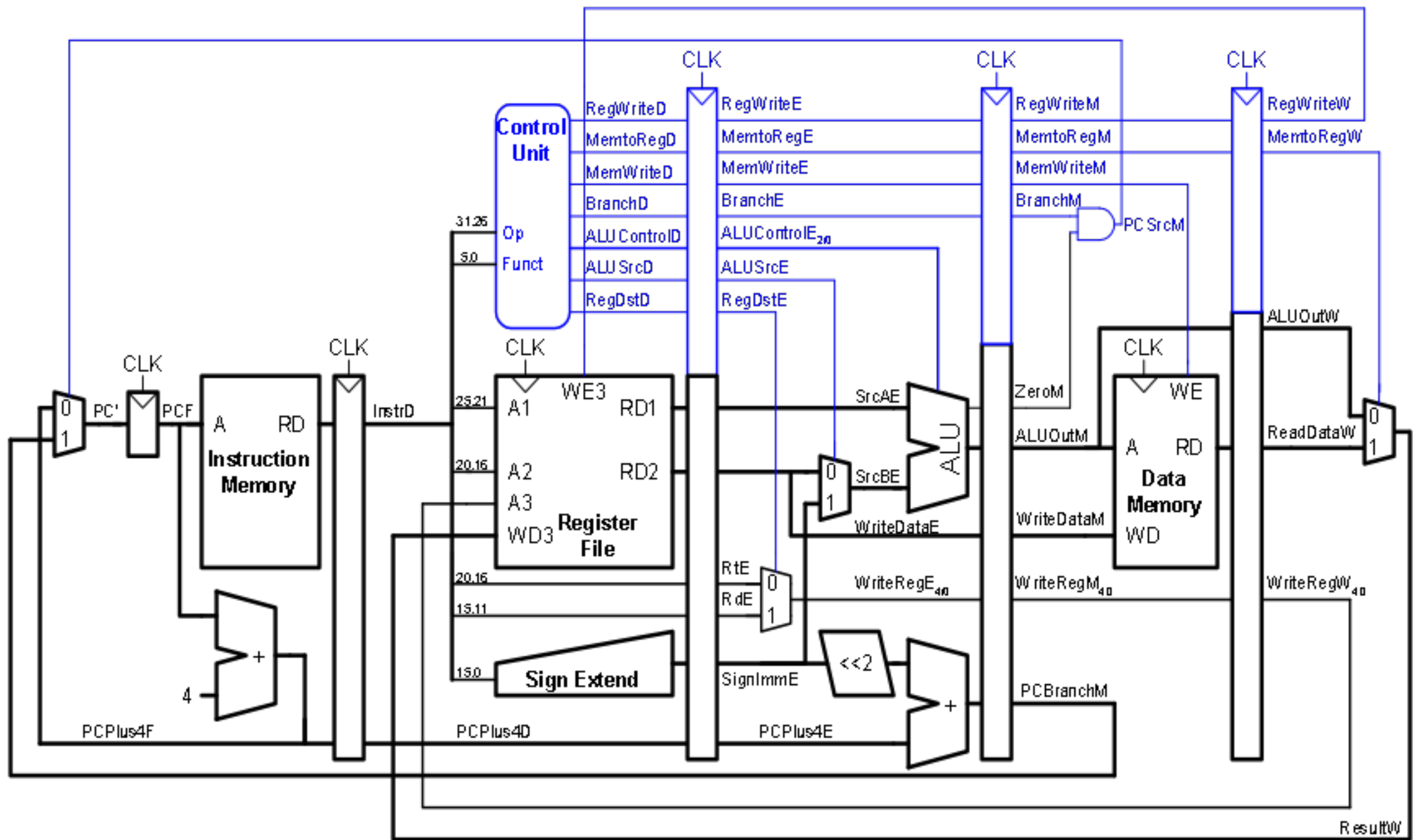


Datapath with Control



译码产生控制信号，与单周期完全相同，控制信号在寄存器间传递，直至不再需要

控制信号在寄存器间传递，直至不再需要



Stage Control

- **Instruction Fetch:** 总是要读取指令存储器和写入 更新的PC，所以在这个流水线阶段没有什么特别需要控制的
- **ID/RF:**与前一阶段一样，每个时钟周期都会发生相同的事情，因此无需设置选择控制线
- **Execution:**设置的信号是 RegDst、ALUOp/Func 和 ALUSrc。这些信号确定指令的结果是否写入位 rt (20-16)（对于load）或 rd(15-11)（对于 R 格式）寄存器，并指定 ALU 操作，确定 ALU 的第二个操作数是寄存器还是符号扩展立即数
- **Memory Stage:** 这个阶段设置的控制线是Branch、MemRead和MemWrite。这些信号分别为 BEQ、LW 和 SW 指令设置
- **WriteBack:** 两条控制线分别是 RegWrite 和 MemToReg，决定写入寄存器数据是ALU 结果还是内存值

Exercise:

- show this sequence executing on the pipeline:

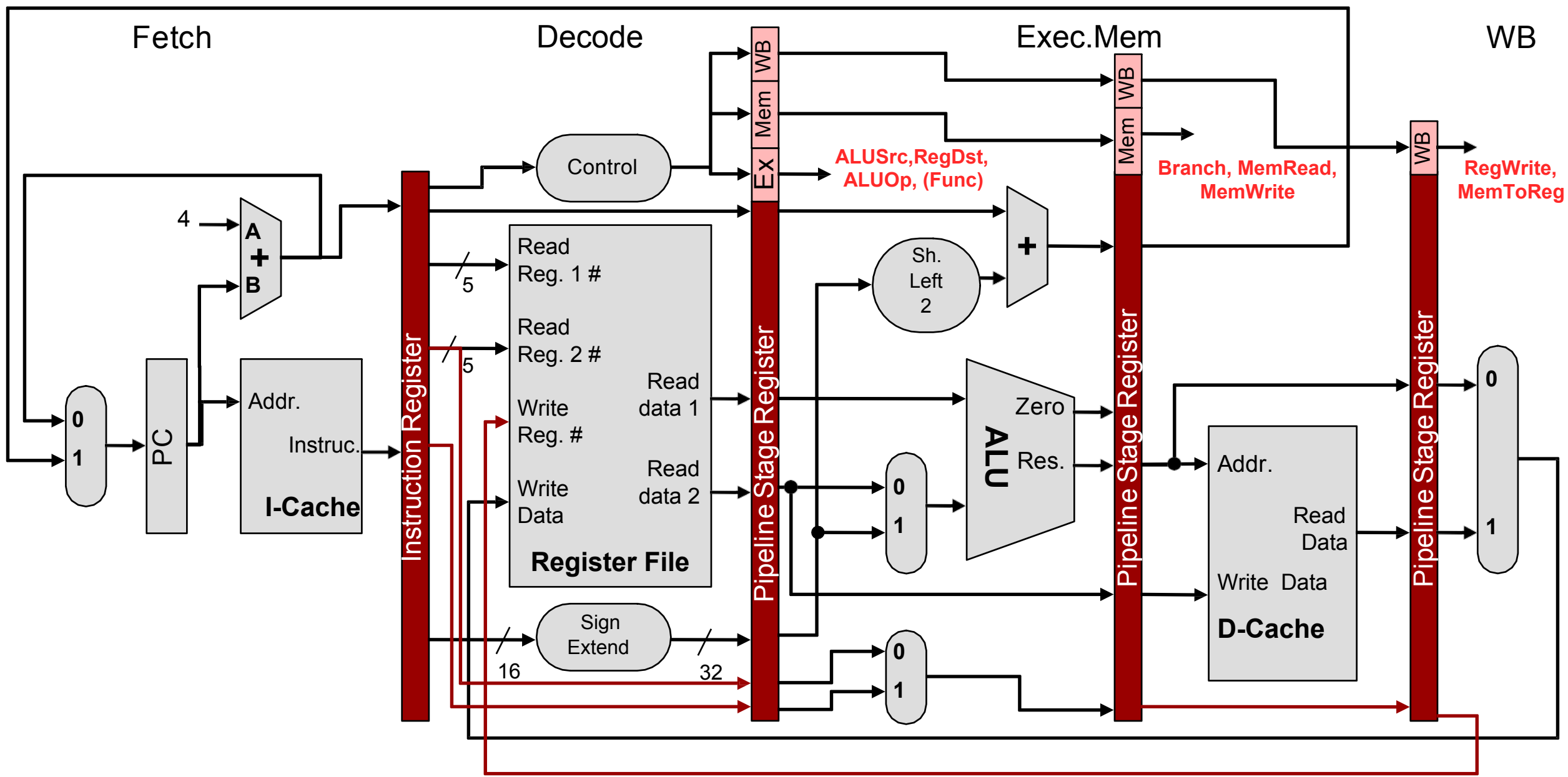
1. LW \$10,9(\$1)

4. OR \$13,\$6,\$7

2. SUB \$11,\$2,\$3

5. ADD \$14,\$8,\$9

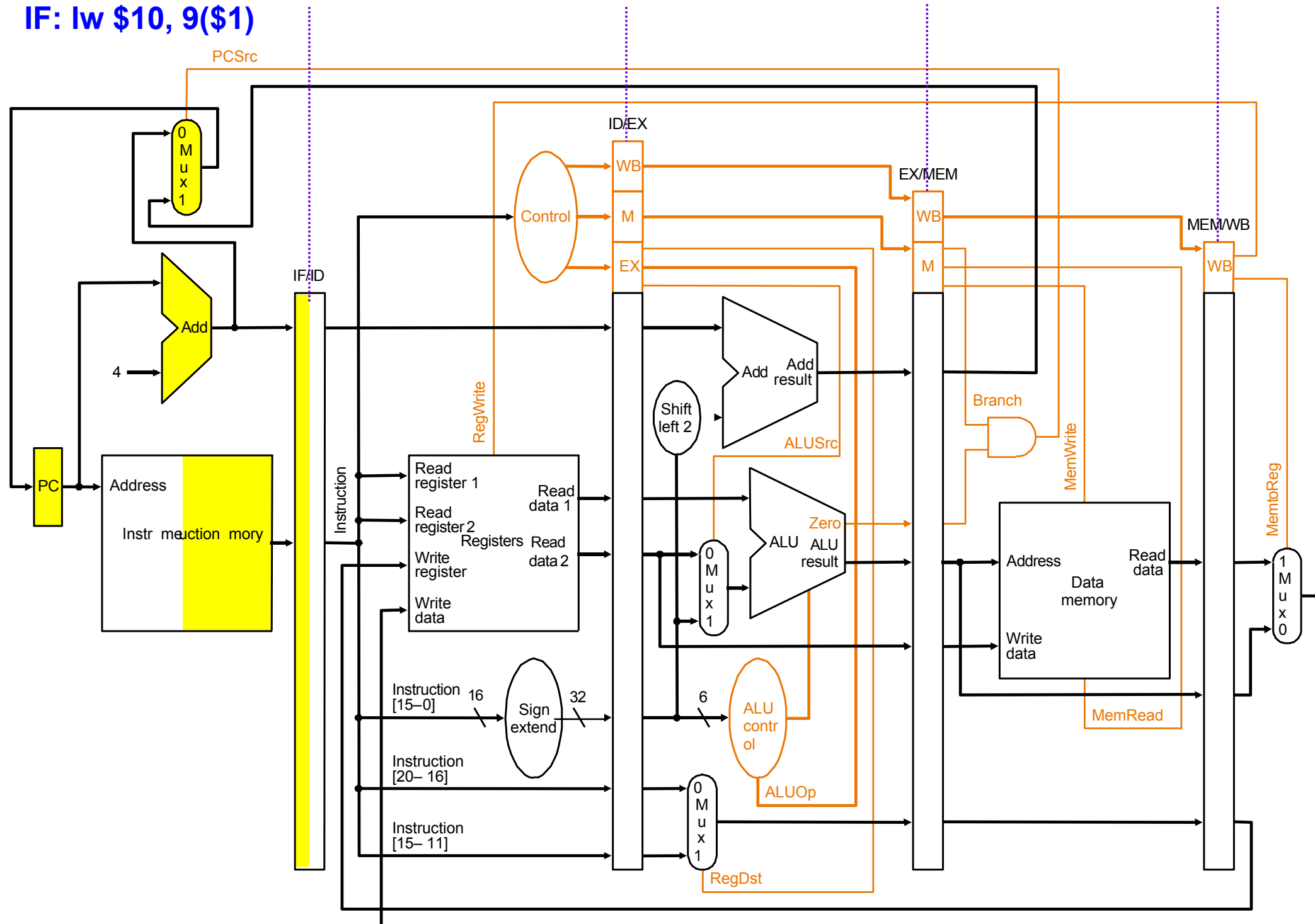
3. AND \$12,\$4,\$5



Datapath with Control

Cycle 1

IF: lw \$t0, 9(\$t1)



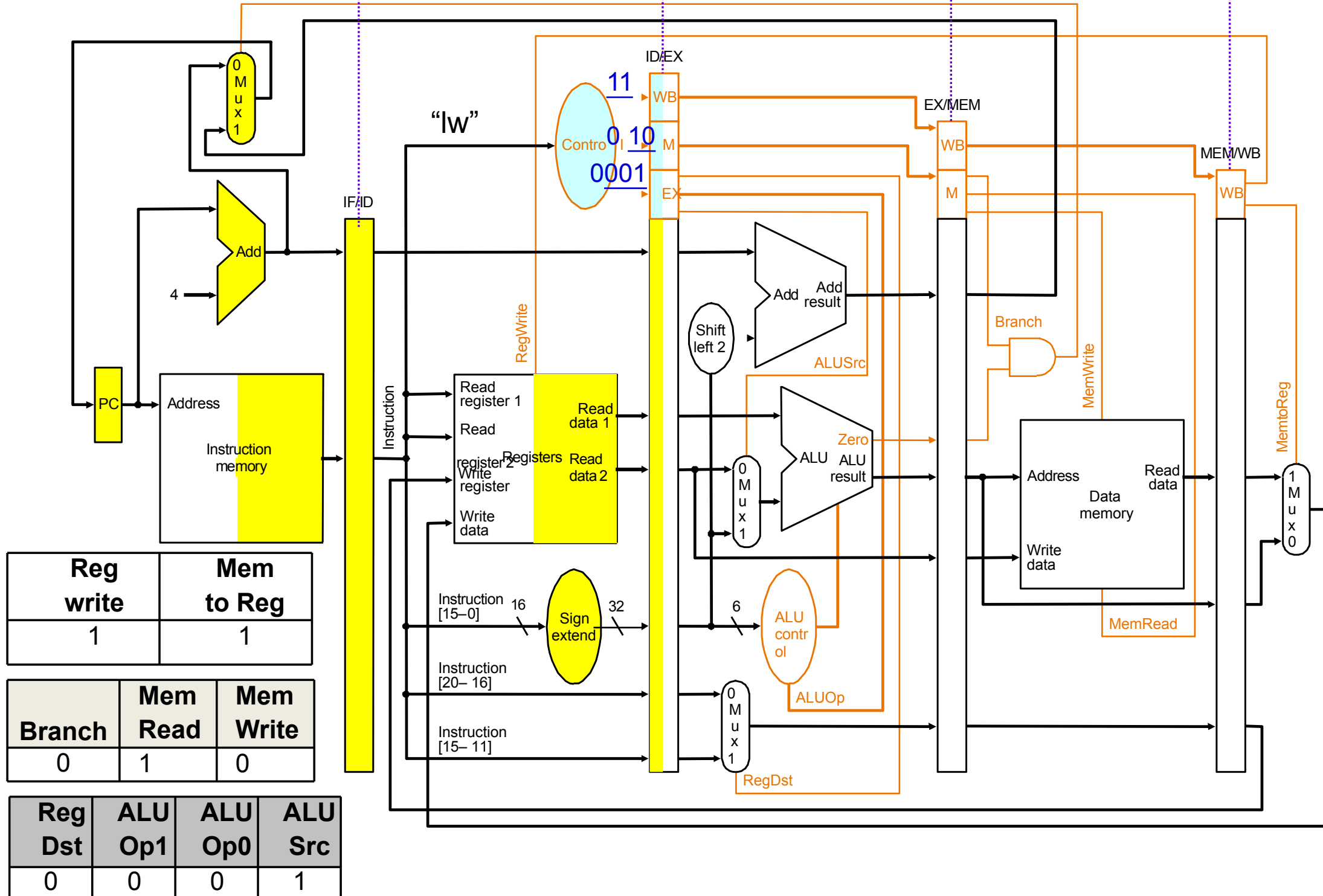
Datapath with Control

Cycle 2

IF: sub \$11, \$2, \$3

ID: lw \$10, 9(\$1)

PCSrc



Datapath with Control

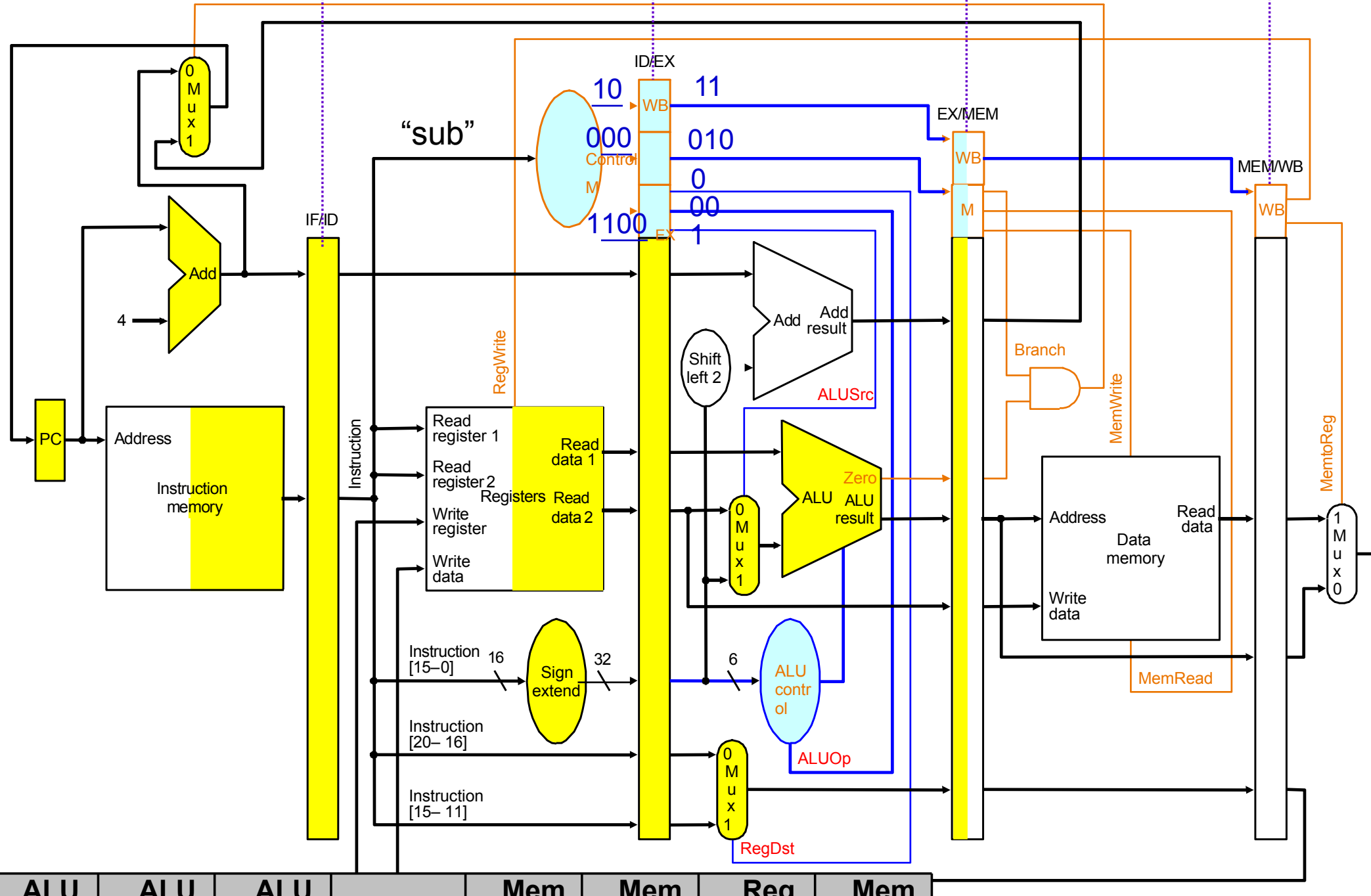
Cycle 3

IF: and \$12, \$4, \$5

ID: sub \$11, \$2, \$3

EX: lw \$10, 9(\$1)

PCSrc



Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
1	1	0	0	0	0	0	1	0

Datapath with Control

Cycle 4

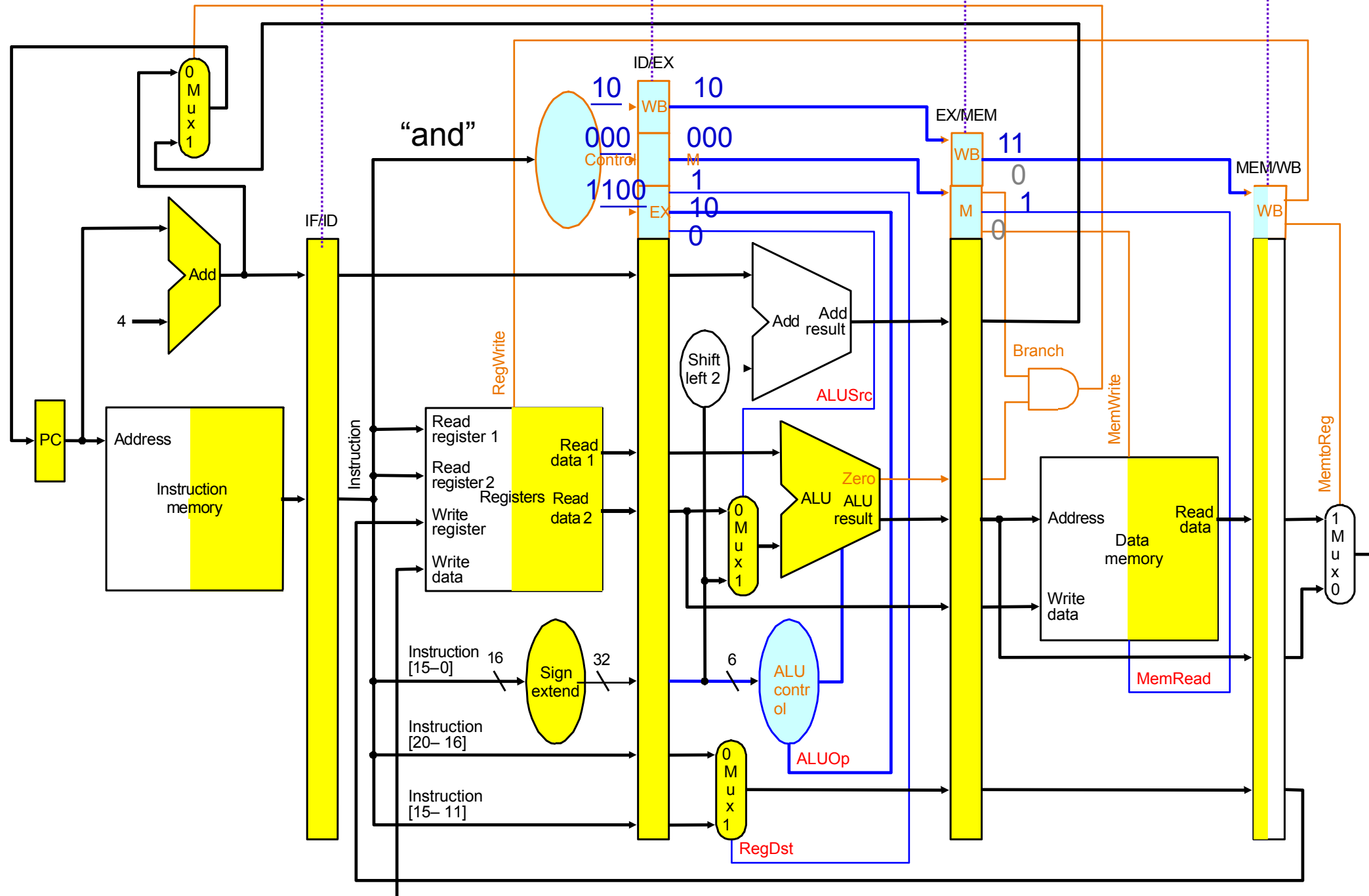
IF: or \$13, \$6, \$7

ID: and \$12, \$4, \$5

EX: sub \$11, \$2, \$3

MEM: lw \$10, 9(\$1)

PCSrc



Datapath with Control

Cycle 5

IF: add \$14, \$8, \$9

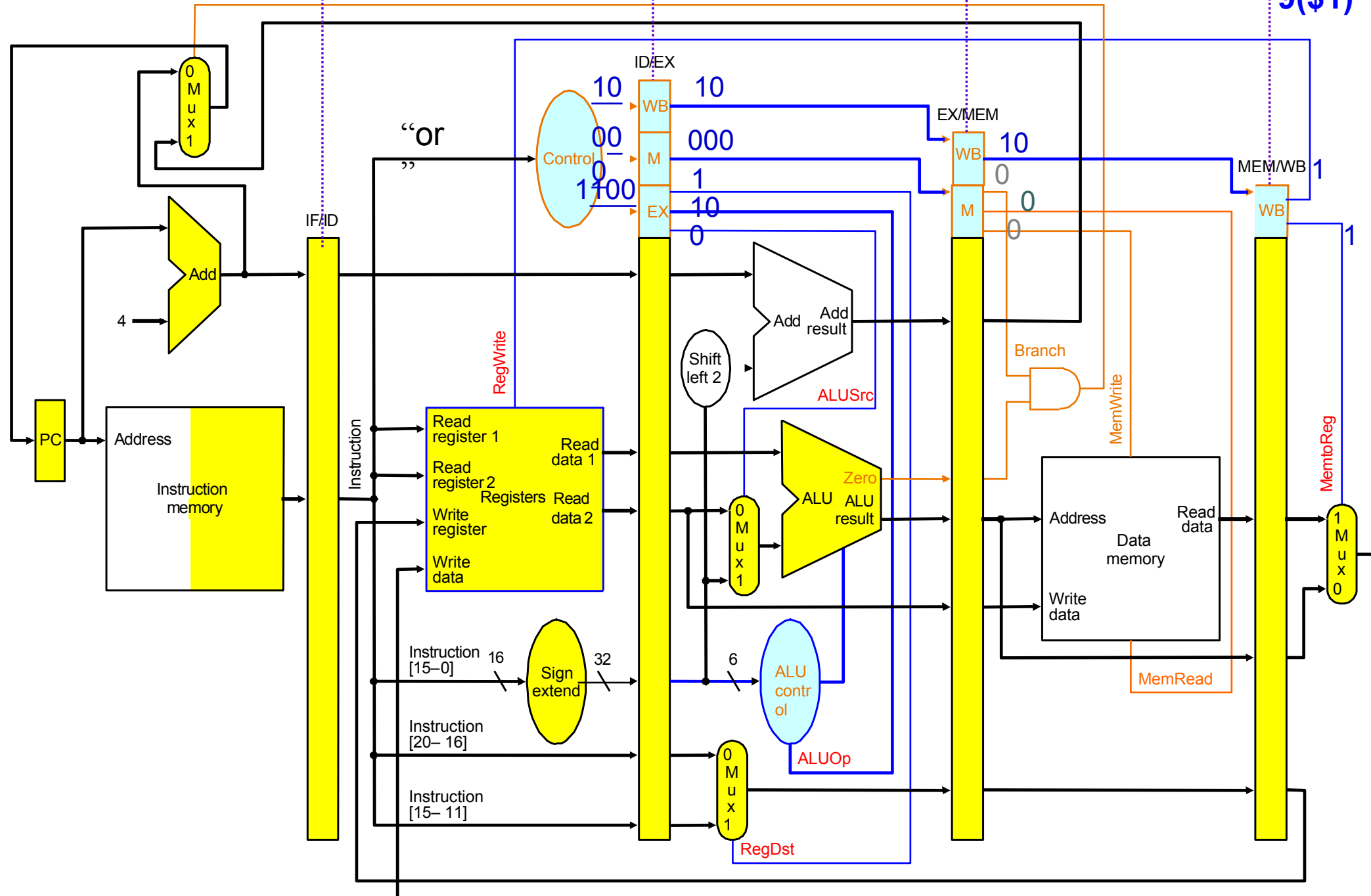
ID: or \$13, \$6, \$7

EX: and \$12, \$4, \$5

MEM: sub \$11, ..

WB: lw \$10, 9(\$1)

PCSrc

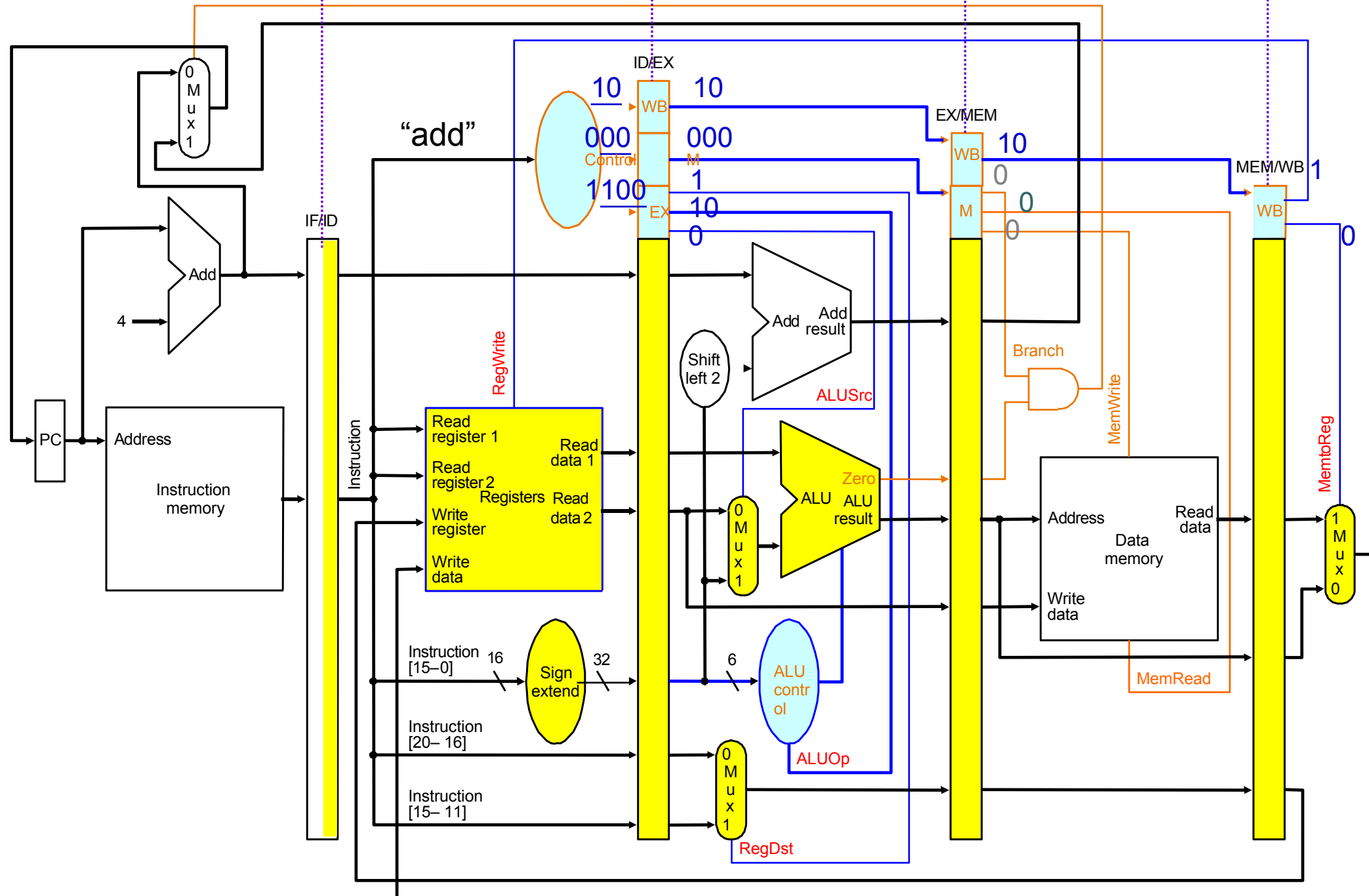


Datapath with Control

Cycle 6

IF: xxxx ID: add \$14, \$8, \$9 EX: or \$13, \$6, \$7 MEM: and \$12... WB: sub \$11, ..

PCSrc

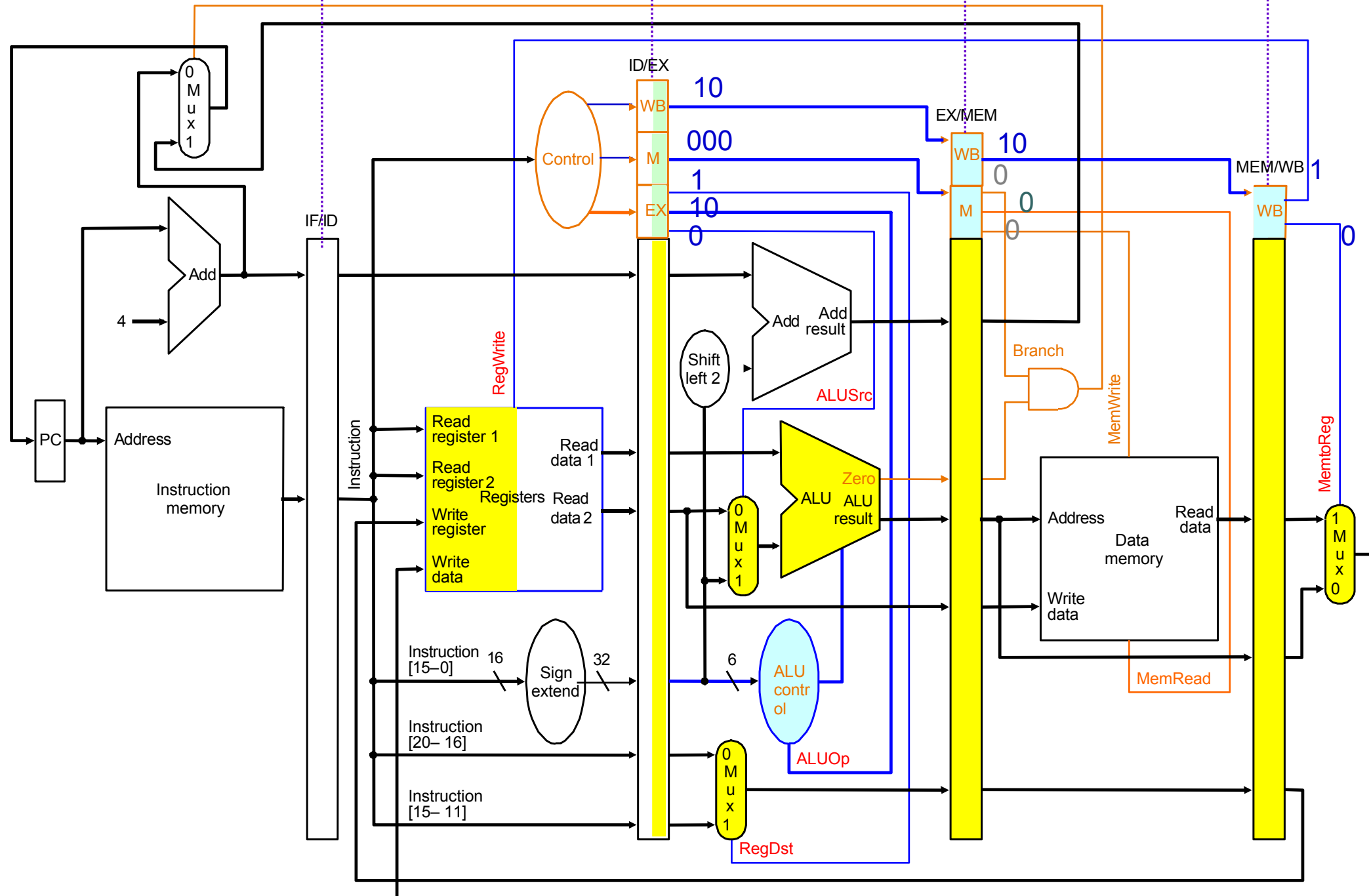


Datapath with Control

Cycle 7

IF: xxxx ID: xxxx EX: add \$14, \$8, \$9 MEM: or \$13, .. WB: and \$12...

PCSrc



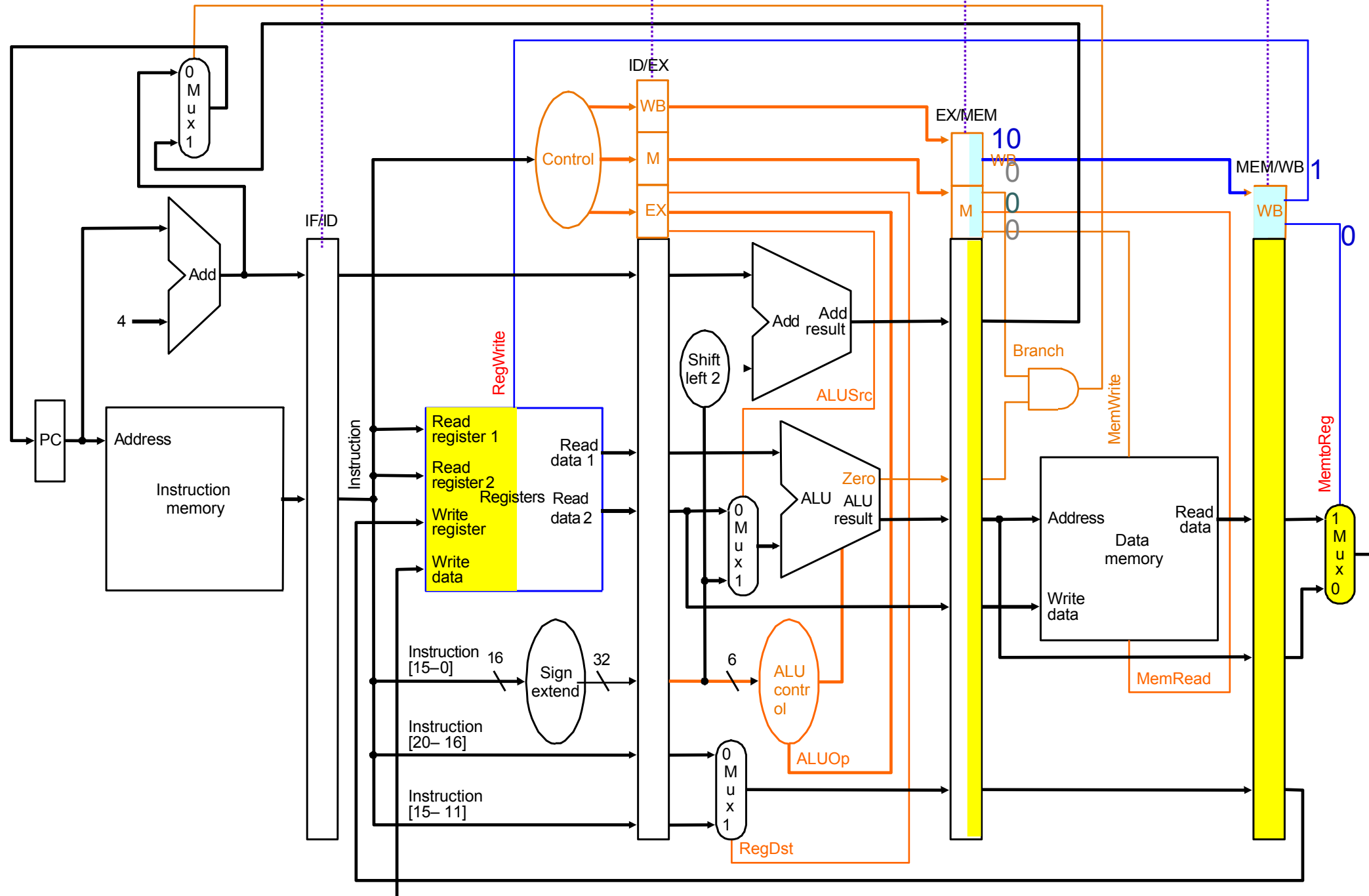
Datapath with Control

Cycle 8

IF: xxxx ID: xxxx EX: xxxx MEM: add \$14, ..

WB: or \$13...

PCSrc



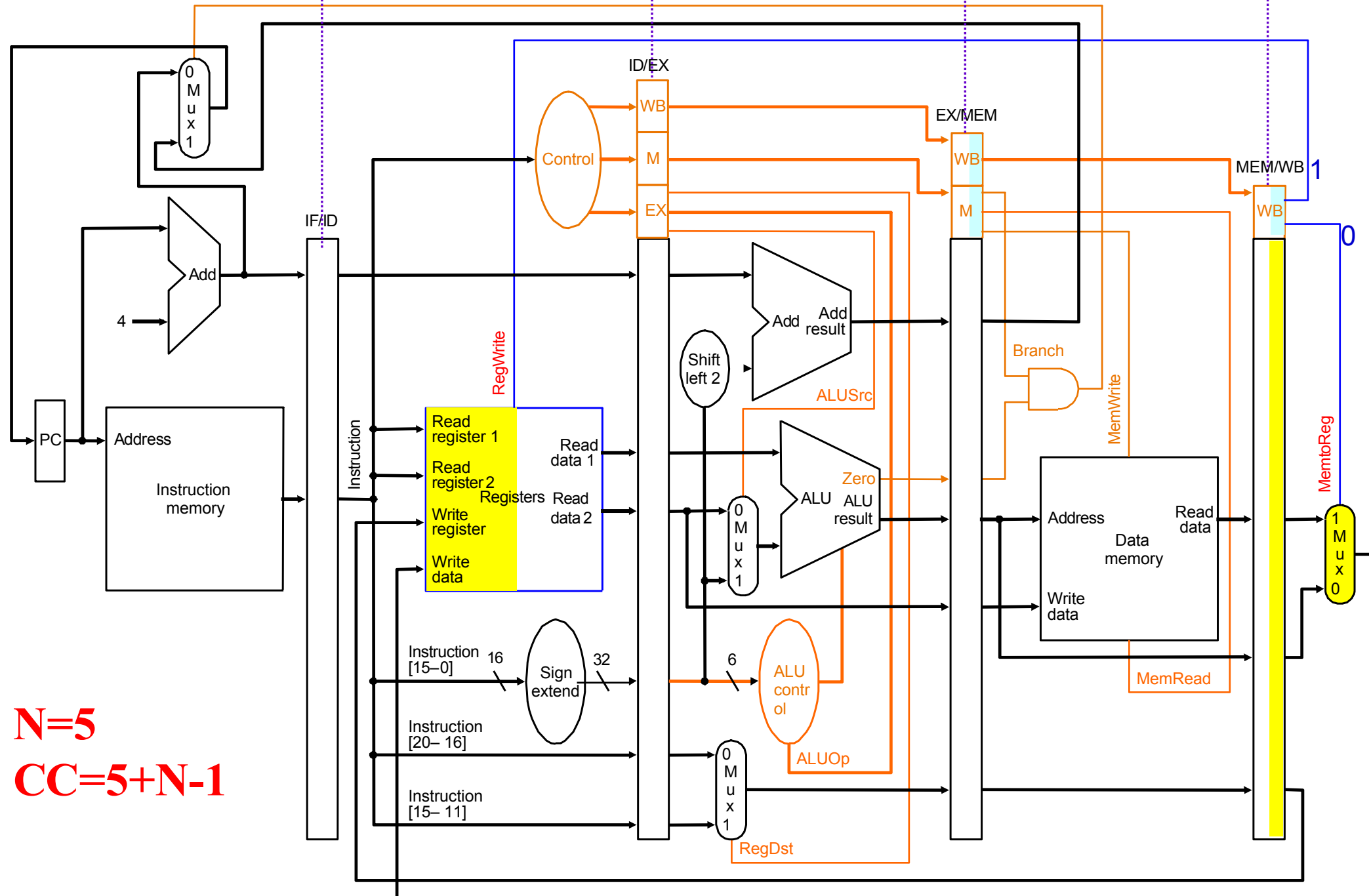
Datapath with Control

Cycle 9

IF: xxxx ID: xxxx EX: xxxx MEM: xxxx

WB: add \$14..

PCSrc



$N=5$

$CC=5+N-1$

- $N=5$,
- $CC=(5+(N-1))=5+4=9$

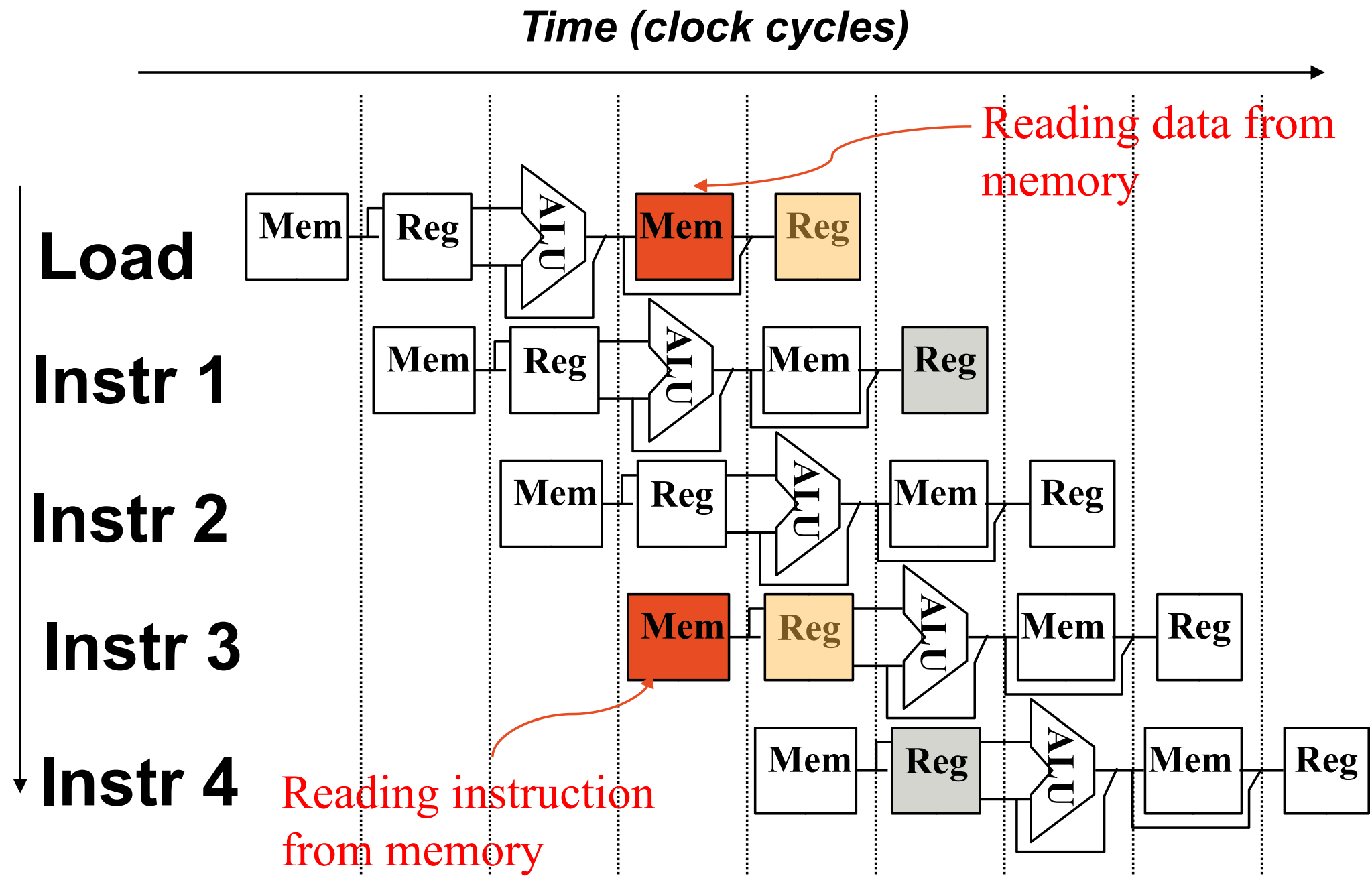
[illegible]

Can Pipelining Get Us Into Trouble?

◦ Yes: Pipeline Hazards

- **structural hazards**: 试图被两条不同的指令同时使用相同的资源
 - **data hazards**: 试图在数据准备好之前使用它
 - 一条指令的源操作数是它前面指令的目的操作数，使用时仍然在流水线中，还没写入寄存器
 - **control hazards**: 试图尝试在转移条件没确定时以及新的 PC 目标地址计算出来之前对程序控制流提前做出决定
 - 比如 branch 条件分支、jump 指令, exceptions 异常终端等
- ┆ 流水线控制必须能够**检测 detect**到冒险的存在
- ┆ 采取行动**解决**冒险

Structural Hazard (结构冒险) 现象



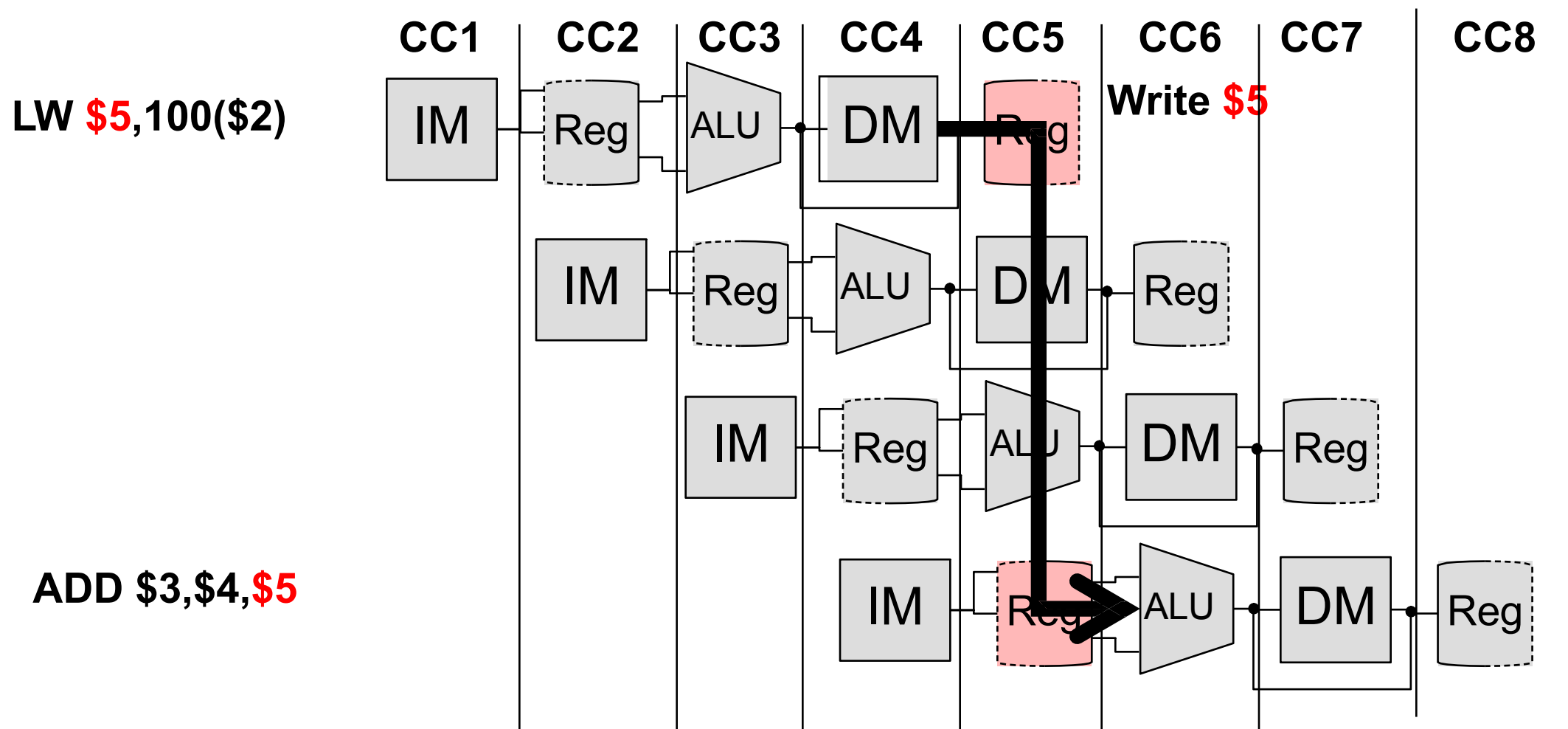
如果只有一个存储器，则在Load指令取数据同时又取指令的话，则发生冲突！
如果不对寄存器堆的写口和读口独立设置的话，则发生冲突！

结构冒险也称为硬件资源冲突：同一个执行部件被多条指令使用。

Register File

- 寄存器读和写两个阶段共用同一寄存器堆。

通过使写入操作总在一个周期的前半段完成而读出总在后半段完成来避免此问题。



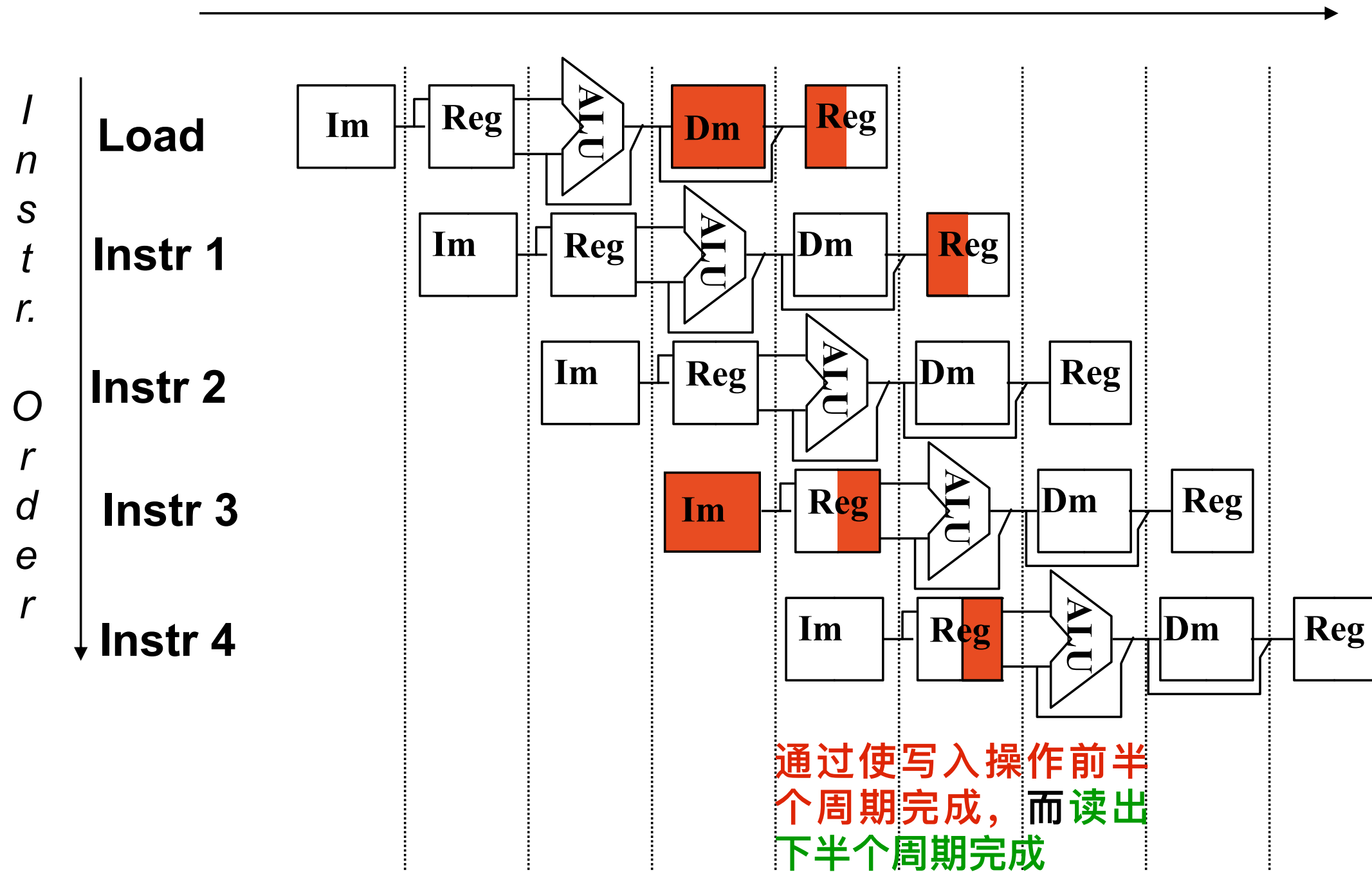
寄存器的写口和读口可看成是独立的两个部件！

Structural Hazard的解决方法

将Instruction Memory (Im) 和 Data Memory (Dm)分开

将寄存器读口和写口独立开来

Time (clock cycles)



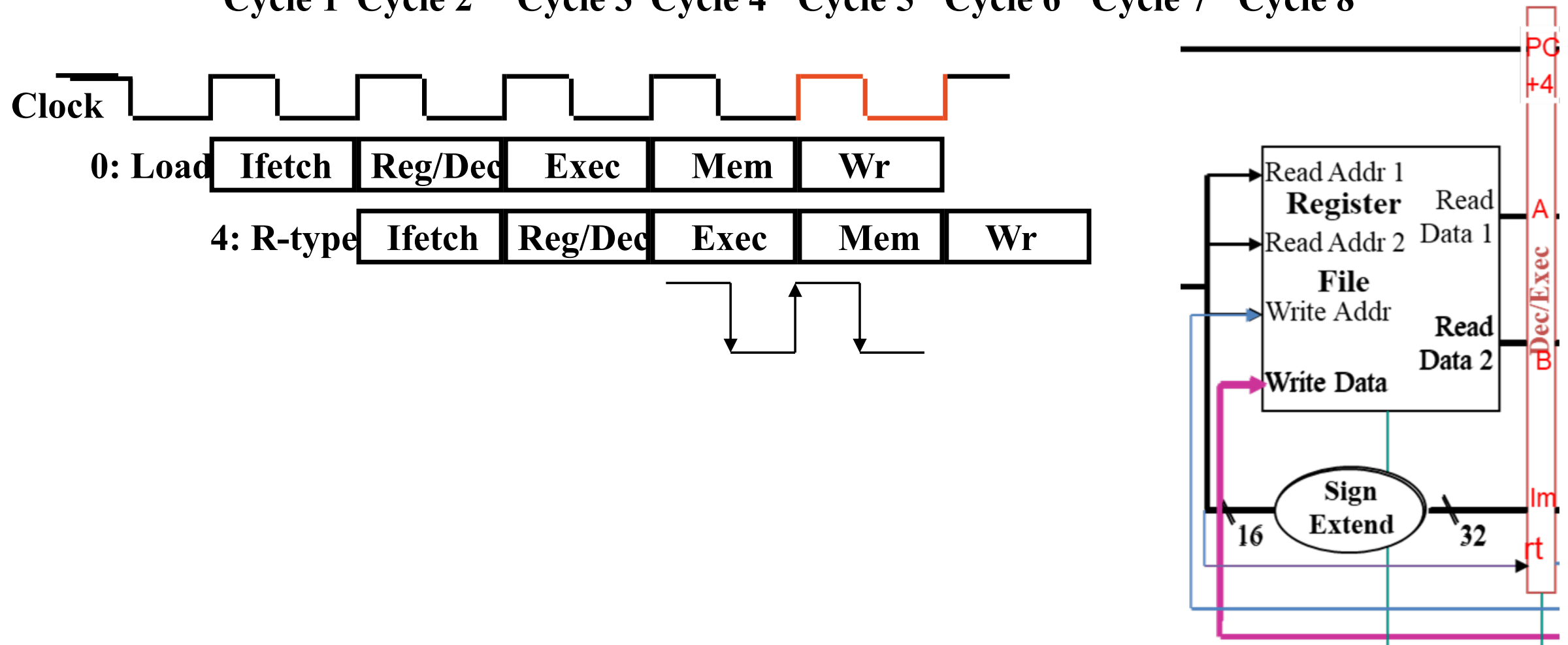
Load 指令采用第五周期上升沿写入的状态:

Clk4结束的上升沿后

指令流: 结果存入MEM/WB

Clk5 开始的上升沿(Cl k4结束)可从MEM/WB读取到addr和data的有关数据, 送到寄存器, **Clk5 下降沿** 结果存入寄存器, 还有接近一半的周期时间, 这个时间足够从寄存器读取到正确的数据。

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8



Data Hazards 数据冒险

初始化, (\$2)=10, 执行完第一条语句(\$2)=-20

	Time (in clock cycles) →								
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
寄存器\$2的值:	10	10	10	10	10/ -20	-20	-20	-20	

程序的执行顺序

(in instructions)

sub \$2, \$1, \$3

$r_2 \leftarrow r_1 \text{ op } r_3$

and \$12, \$2, \$5

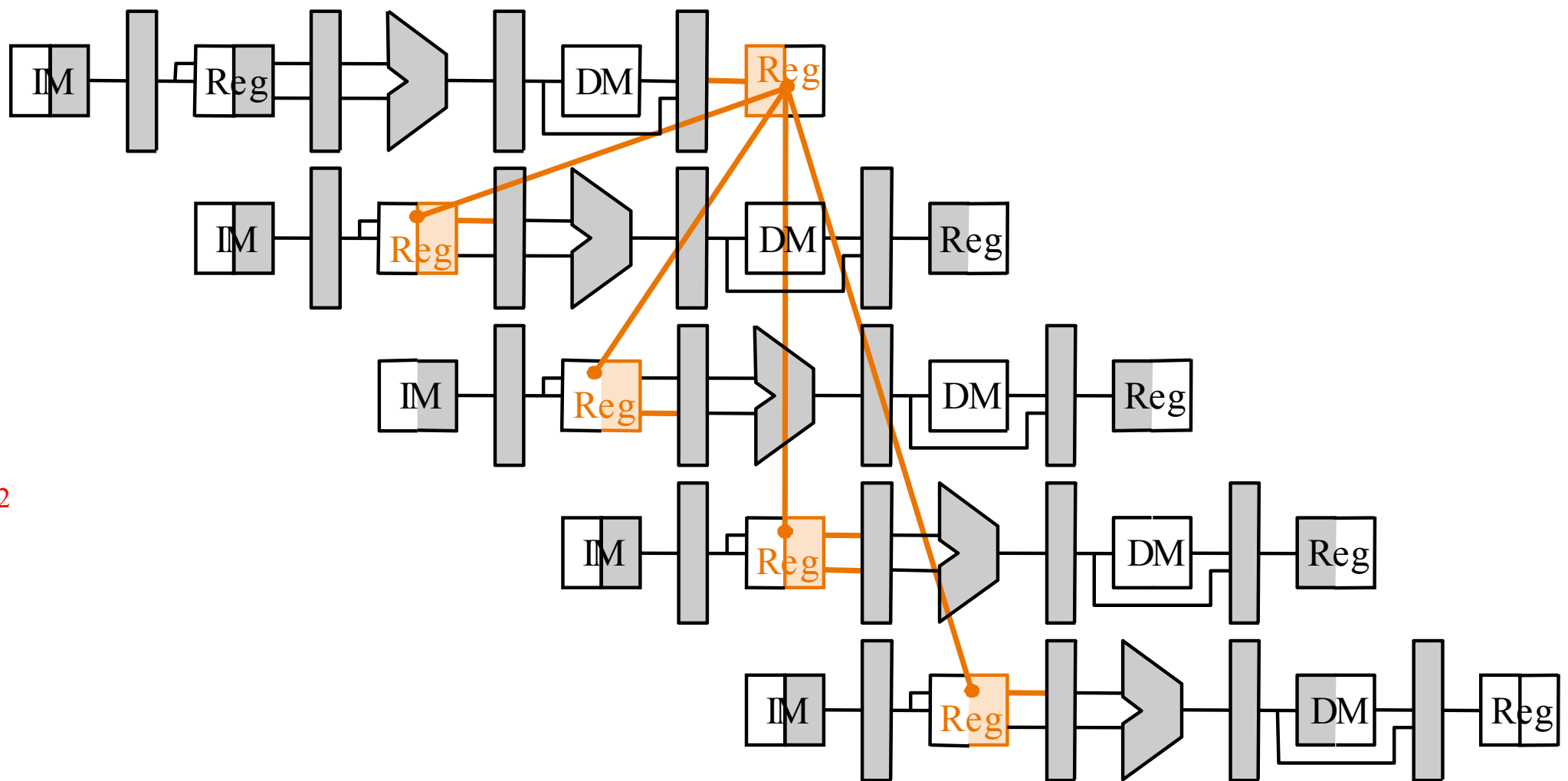
$r_{12} \leftarrow r_2 \text{ op } r_5$

or \$13, \$6, \$2

$r_{13} \leftarrow r_6 \text{ op } r_2$

add \$14, \$2, \$2

sw \$15, 100(\$2)

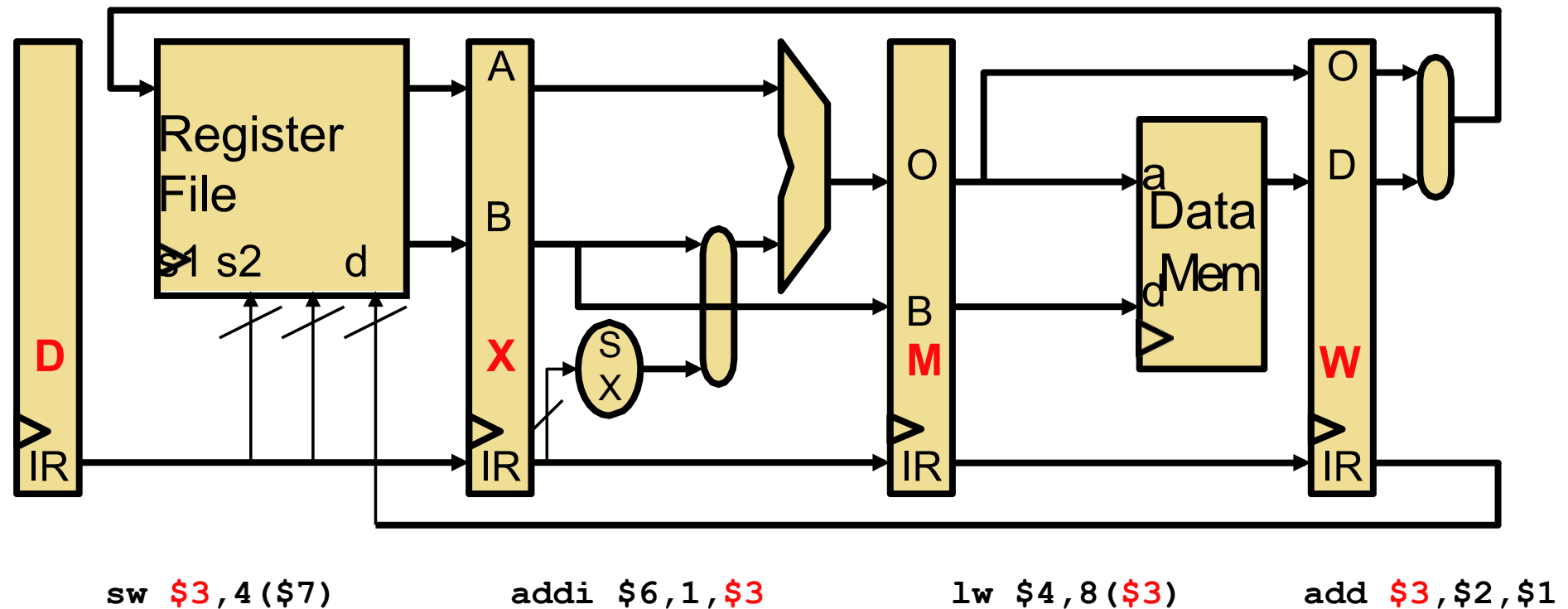


Read-after-Write Hazards

(RAW)写入后才能读到正确的值

Data Hazards 数据冒险

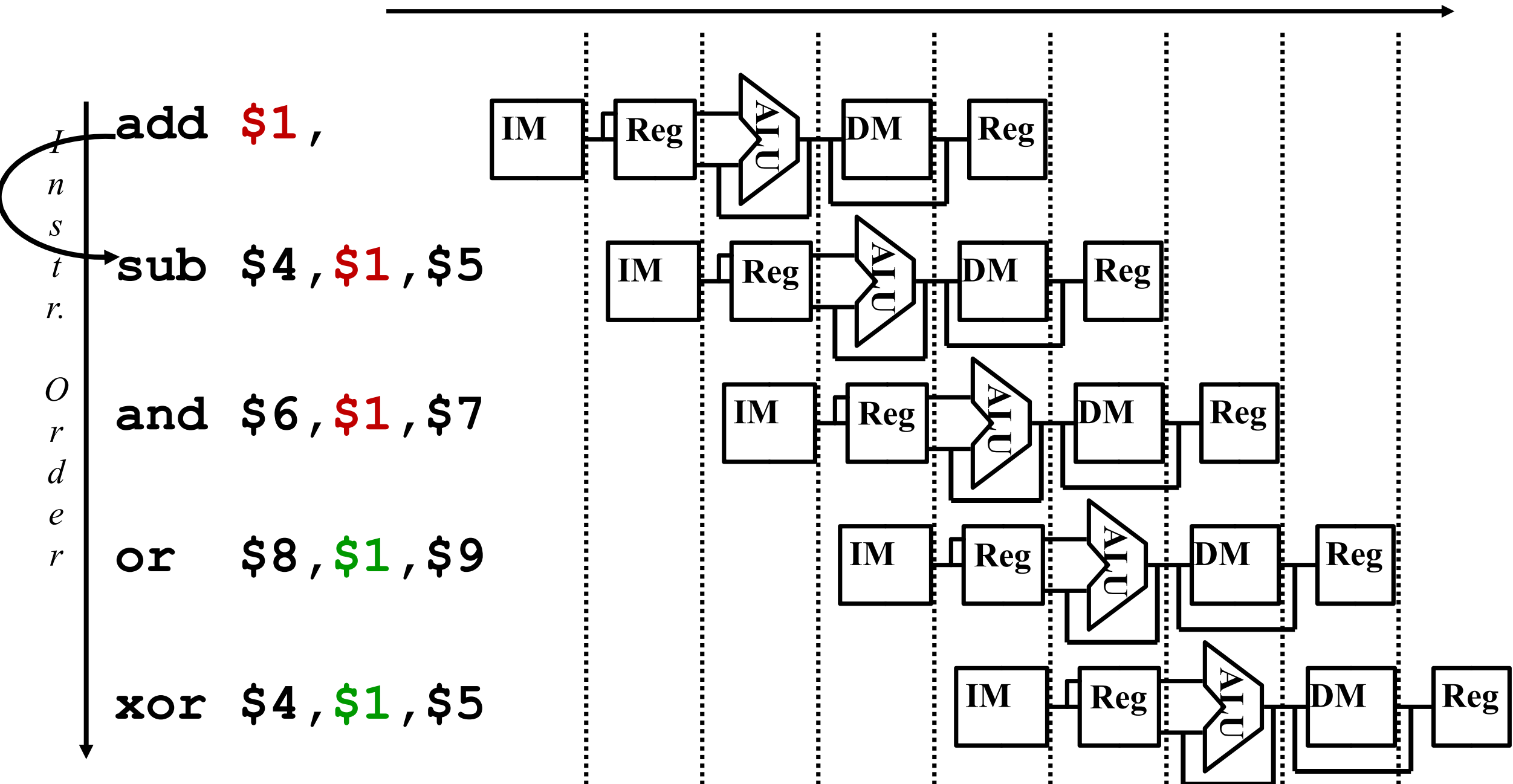
- 指令之间存在数据依赖 **data dependences**



- 这个程序“program”能在流水线中正确执行吗?
- 哪条指令能得到正确的输入?
- 当前周期 add 正在写结果到 **\$3**
 - lw 读 **\$3** 在两个周期前 → 得到错误的值
 - addi 读 **\$3** 在一个周期前 → 也是错误值
- sw 这个周期正在读 **\$3** → 可能对 (取决于寄存器的设计)

Register Usage Can Cause Data Hazards

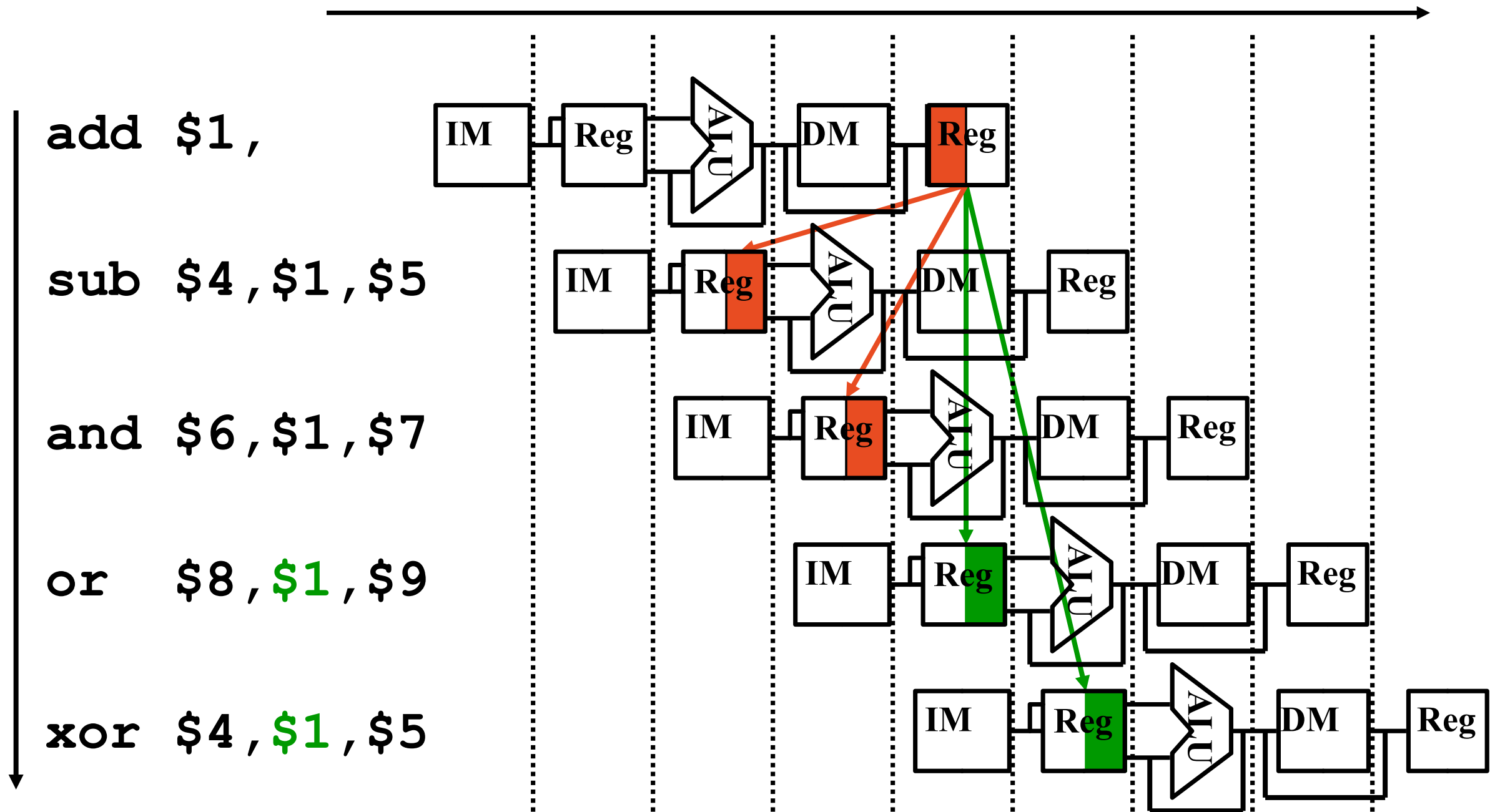
- Dependencies backward in time cause hazards



- ❑ Read before write data hazard
- ❑ 读取的操作数\$1在\$1被写入之前

Register Usage Can Cause Data Hazards

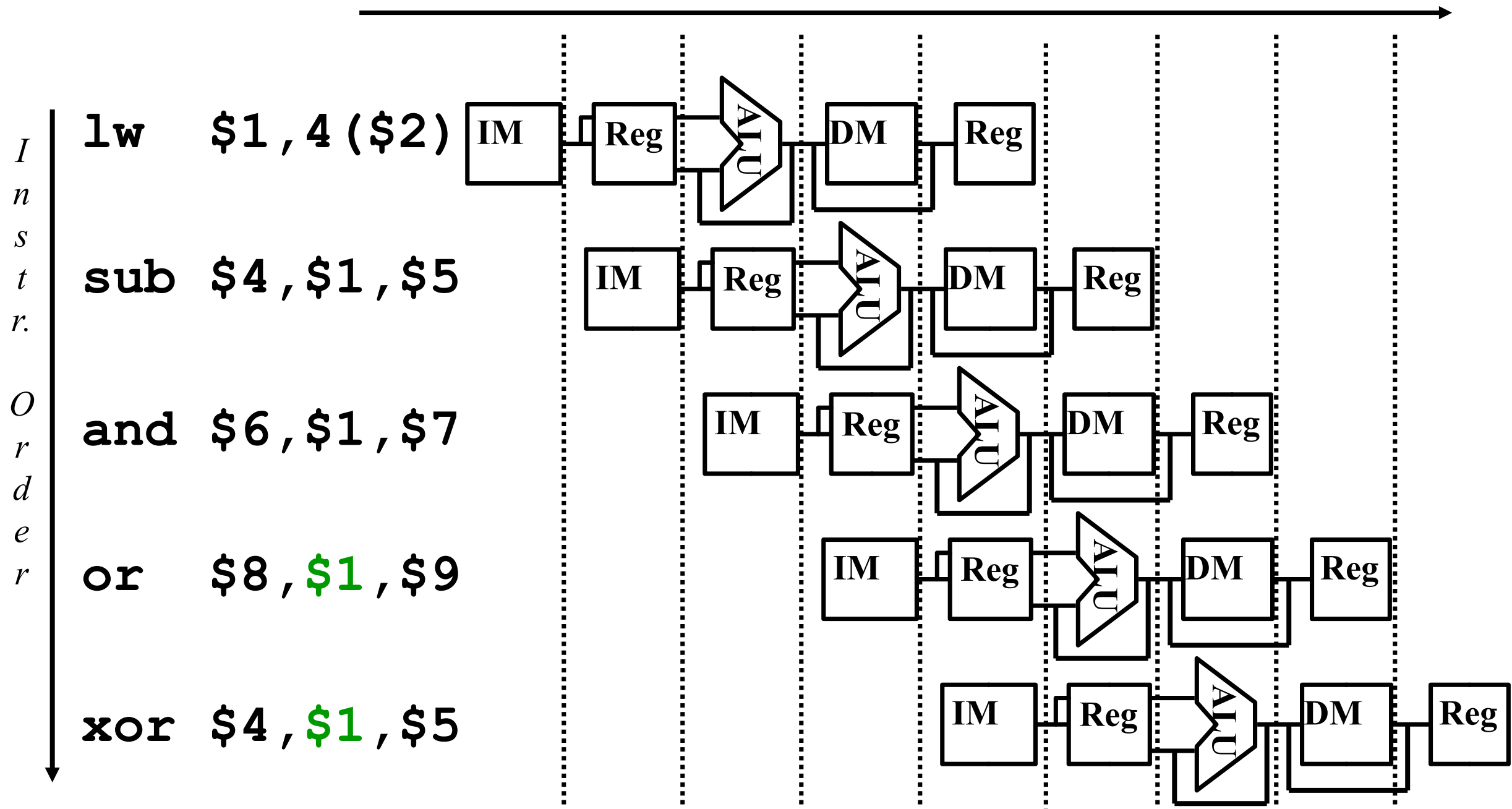
- Dependencies backward in time cause hazards



❑ Read before write data hazard

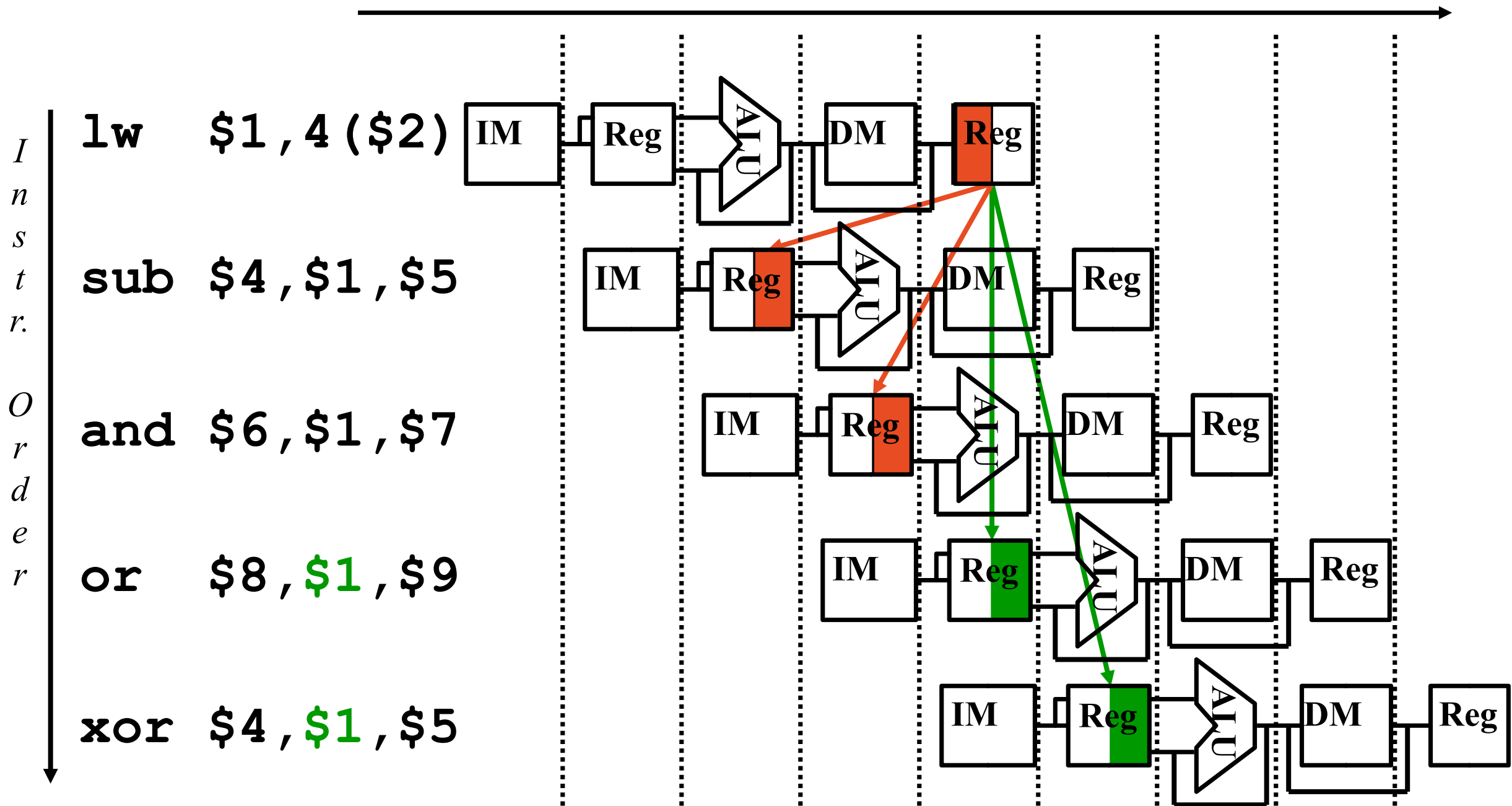
Loads Can Cause Data Hazards

- Dependencies backward in time cause **hazards**



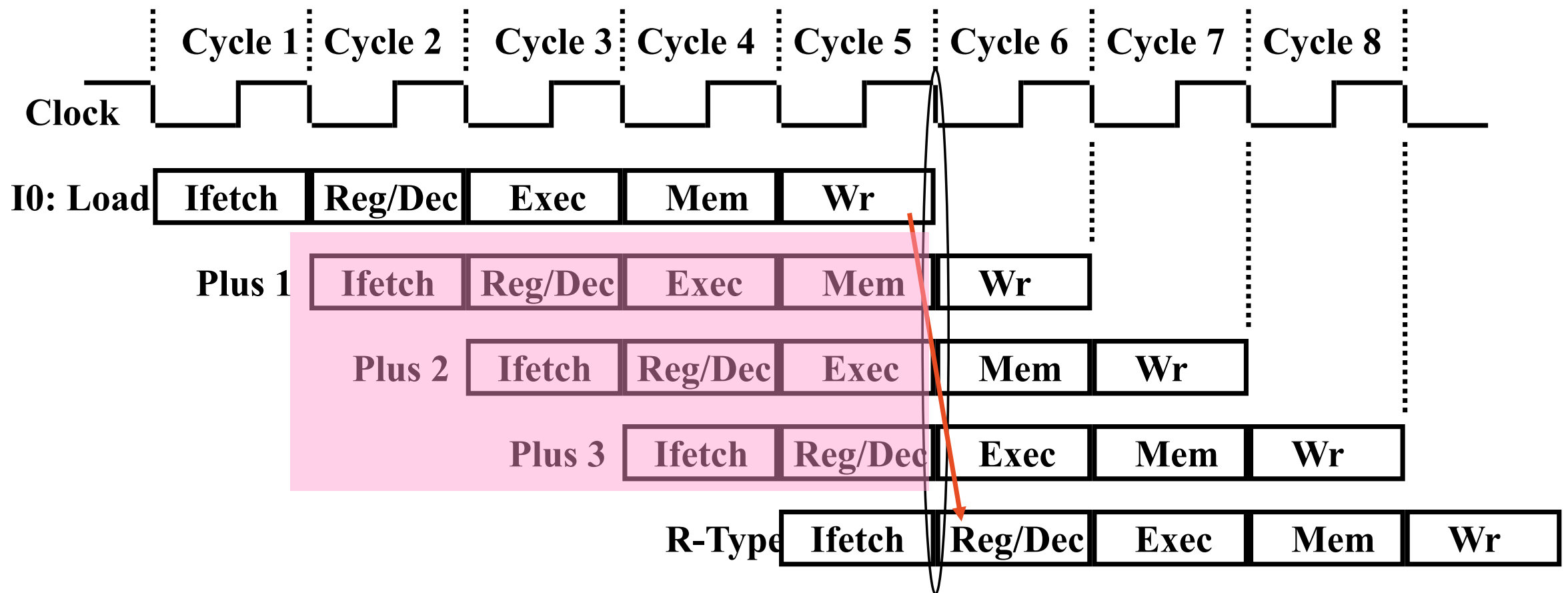
Loads Can Cause Data Hazards

- Dependencies backward in time cause hazards



❑ Load-use data hazard

Load delay and data hazard




◦ Load 1st 时钟周期开始, 但是:

- 数据写回 REG 在 5th 时钟周期 (如果寄存器读口和写口没有独立开来)
- 在 6th 时钟周期才可以获取更新后的数据

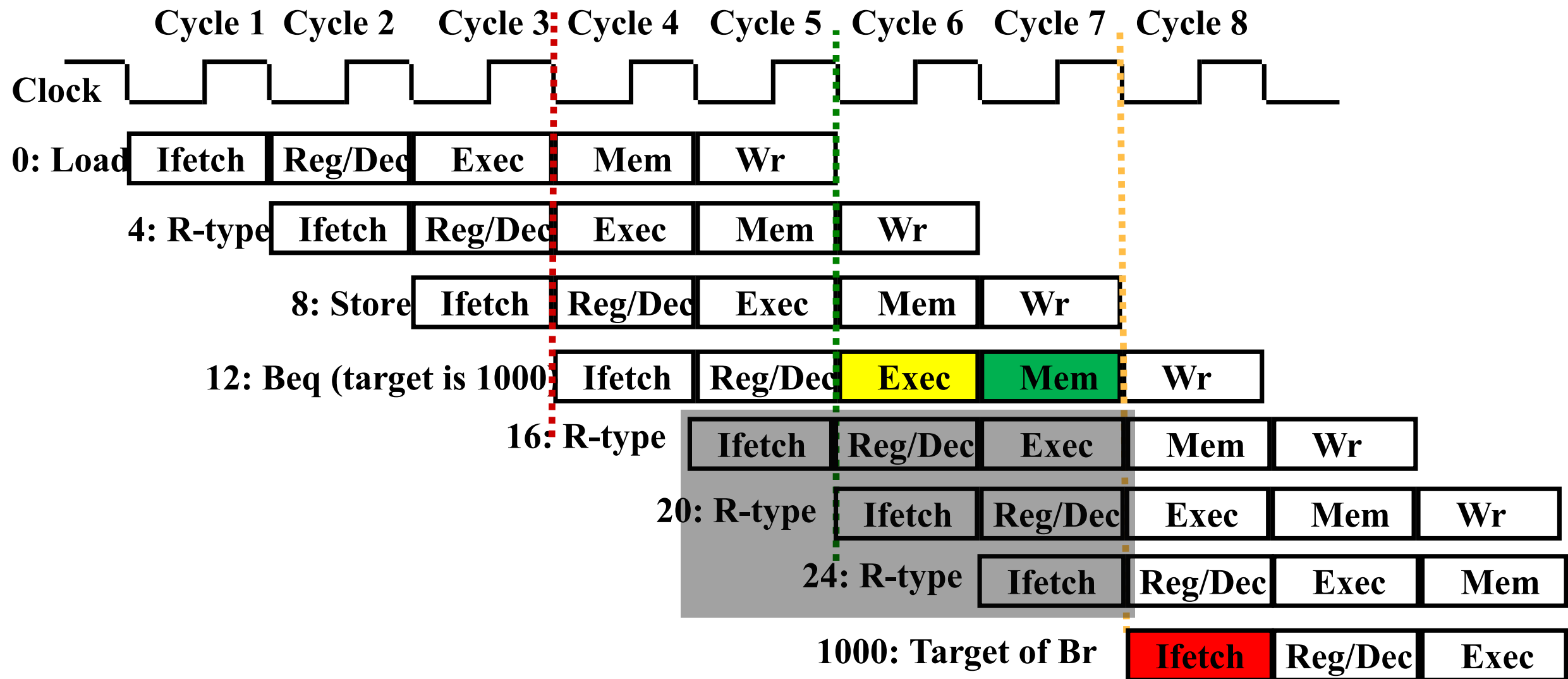
结果: 3条指令已在流水线中启动 (如果后续指令使用load的数据, 则需要等待3个时钟周期才是正确数据!)

Data Hazard or Data Dependency 数据冒险或数据依赖性

Control Hazards控制竞争

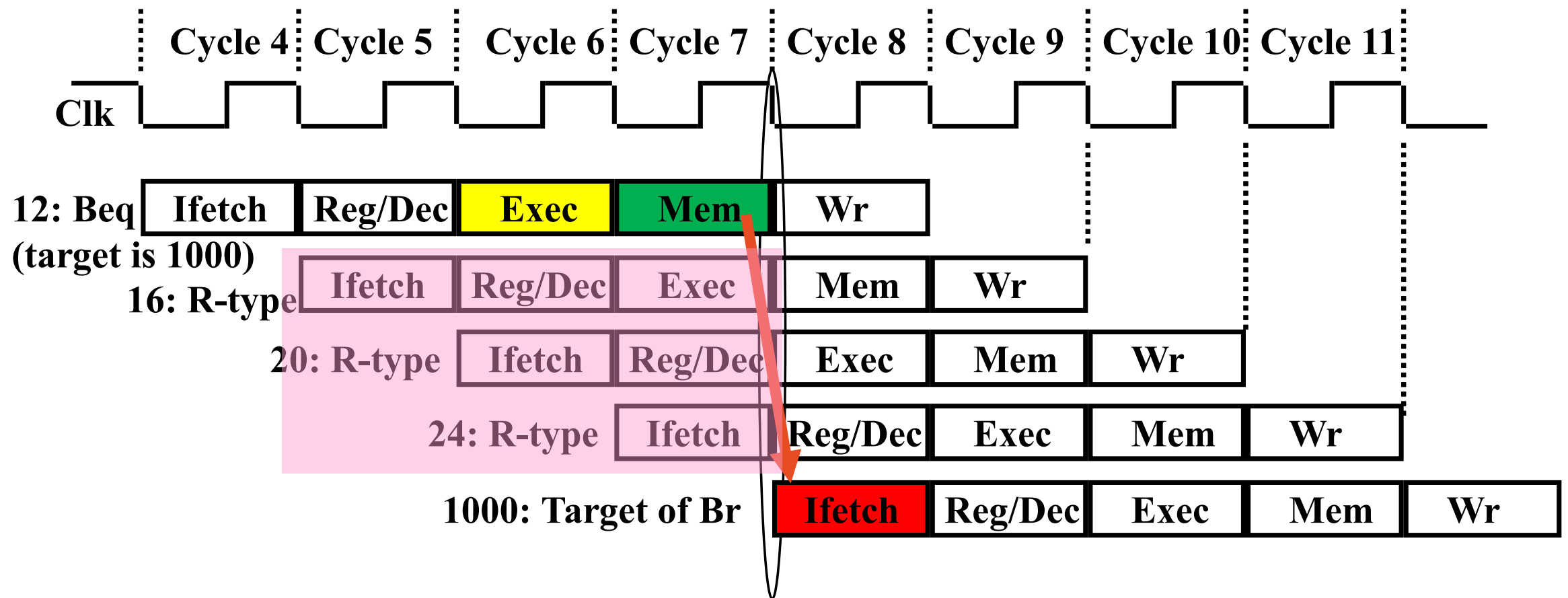
- 
- 当指令流向不是预期的方式
 - 有条件的分支 (beq, bne)需阻塞3个周期
 - 无条件的跳转 (j)
 - 可能的解决方案
 - Stall
 - 将判断提前
 - 预测
 - 延迟决定(即延迟分支)(requires compiler support)
 - 控制竞争不像数据竞争那样频繁的发生，但是也没有像 forwarding 这样解决它的那么有效的方式

beq流水线执行过程



- Branch指令何时确定是否转移？转移目标地址在第几周期计算出来？
 - 第六周期得到Zero和转移地址、第七周期控制转移地址送到PC输入端、第八周期开始才能根据转移地址取指令
 - 如果Branch指令执行结果是需要转移（称为taken），则流水线会怎样？

转移分支指令(Branch)引起的“延迟”现象



◦ 虽然Beq指令在第四周期取出，但：

- 目标地址在第七周期才被送到PC的输入端
- 第八周期才能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！

◦ 这种现象称为控制冒险 (Control Hazard)

(注：也称为分支冒险或转移冒险 (Branch Hazard))

总结：流水线的三种冲突/冒险 (Hazard) 情况

◦ 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

- Structural hazards (hardware resource conflicts):

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

- Data hazards (data dependencies):

现象：后面指令用到前面指令结果时，前面指令结果还没产生

- 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

- Control (Branch) hazards (changes in program flow):

现象：转移或异常改变执行流程，顺序执行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(实行分支延迟)

内容小结

- 指令的执行可以像洗衣服一样，分为N个步骤，并用流水线方式进行
 - 均衡时指令吞吐率提高N倍，但不能缩短一条指令的执行时间
 - 流水段数以最复杂指令所需步骤数为准（有些指令的某些阶段为空操作），每个阶段的宽度以最复杂阶段所需时间为准（尽量调整使各阶段均衡）
- 以Load指令为准，分为五个阶段
 - 取指令段(IF)
 - 取指令、计算PC+4 (IUnit: Instruction Memory、Adder)
 - 译码/读寄存器(ID/Reg)段
 - 指令译码、读Rs和Rt (寄存器读口)
 - 执行(EXE)段
 - 计算转移目标地址、ALU运算 (Extender、ALU、Adder)
 - 存储器(MEM)段
 - 读或写存储单元 (Data Memory)
 - 写寄存器(Wr)段
 - ALU结果或从DM读出数据写到寄存器 (寄存器写口)
- 流水线控制器的实现
 - IF和ID/Reg段不需控制信号控制，只有EXE、MEM和Wr需要
 - ID段生成所有控制信号，并随指令的数据同步向后续阶段流动
- 寄存器和存储器的竞争问题可利用时钟信号来解决
- 流水线冒险：结构冒险、控制冒险、数据冒险
(下一讲主要介绍解决流水线冒险的数据通路如何设计)

举例

- 流水线优化了单周期CPU，是一种强大的逻辑设计方法，可以减少时钟时间并提高吞吐量，即使它增加了单个任务的延迟并增加了额外的逻辑，但在流水线CPU中，多条指令在执行中重叠，这是并行的一个很好的例子，这是计算机体系结构中的一个伟大思想。
- 寄存器堆的写入地址应通过流水线寄存器传递到回写阶段，以便指令将结果写入正确的寄存器。
- 流水线的一个很大的优点是可以在更短时钟时间内提高性能。

Q1. 给定如下参数，确定单周期时钟周期

Element	Register clk-to-q	Register Setup	MUX	ALU	Mem Read	Mem Write	RegFile Read	RegFile Setup
Parameter	$t_{\text{clk-to-q}}$	t_{setup}	t_{mux}	t_{ALU}	t_{MEMread}	t_{MEMwrite}	t_{RFread}	T_{RFsetup}
Delay(ps)	30	20	25	200	250	200	150	20

$$t_{\text{clk,single}} \geq t_{\text{PC, clk-to-q}} + t_{\text{IMEMread}} + t_{\text{RFread}} + t_{\text{ALU}} + t_{\text{DMEMread}} + t_{\text{mux}} + t_{\text{RFsetup}}$$

$$= 30 + 250 + 150 + 200 + 250 + 25 + 20 = 925 \text{ ps}$$

$$f_{\text{clk,single}} = 1/t_{\text{clk,pipe}} \leq 1/(925 \text{ ps}) = 1.08 \text{ GHz}$$

◦ Q2. 如果是流水线CPU，时钟周期及频率是多少？

$$t_{\text{clkpipe}} \geq \max$$

$$t_{\text{clktoq}} + t_{\text{MEMread}} + t_{\text{setup}}$$

$$t_{\text{clktoq}} + t_{\text{RFread}} + t_{\text{setup}} (\text{Decode})$$

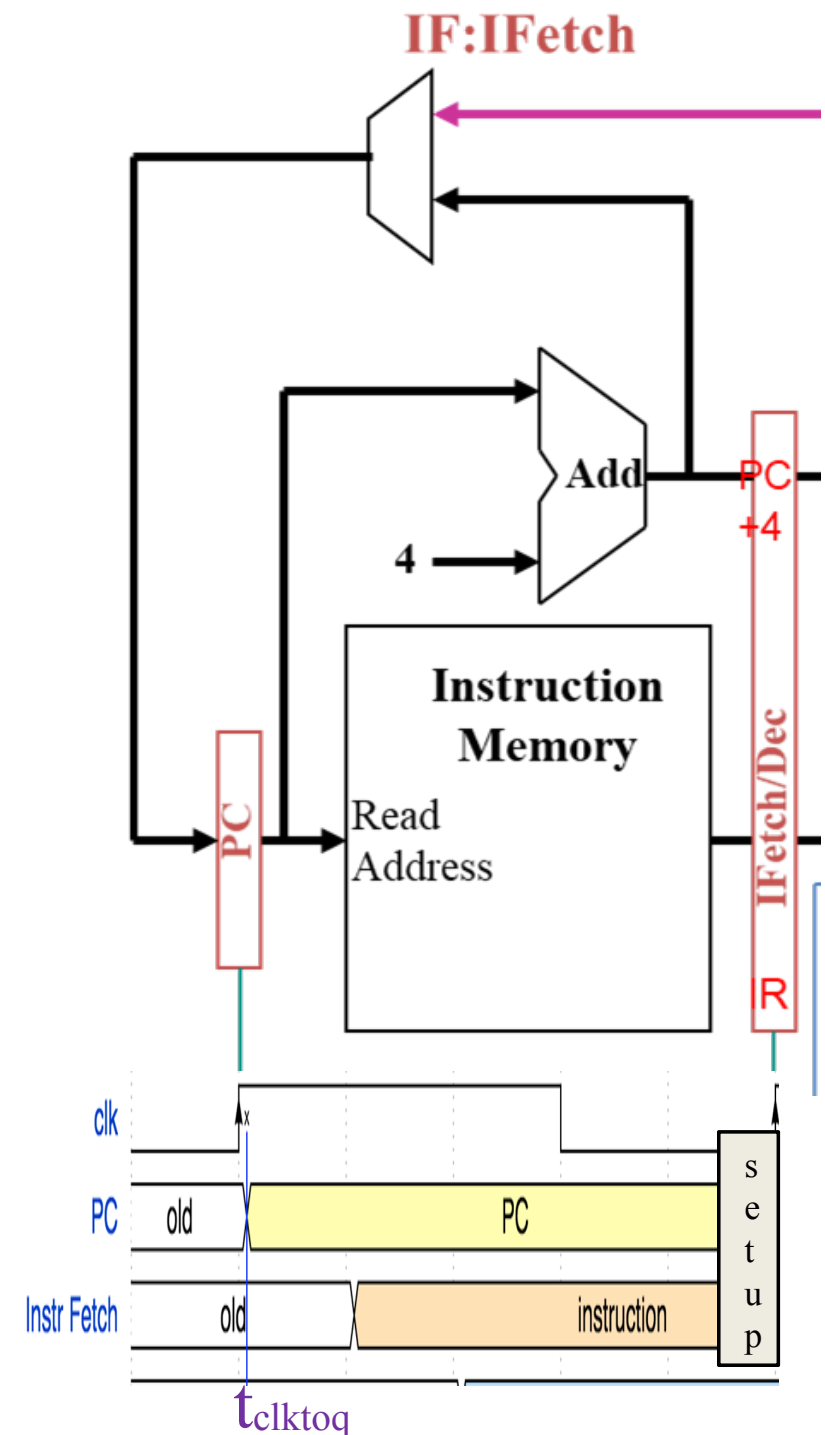
$$t_{\text{clktoq}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{setup}} (\text{Execute})$$

$$t_{\text{clktoq}} + t_{\text{MEMread}} + t_{\text{setup}} (\text{Memory})$$

$$t_{\text{clktoq}} + t_{\text{mux}} + t_{\text{RFsetup}} (\text{Writeback})$$

$$= 30 + 250 + 20 = 300 \text{ ps}$$

$$f_{\text{clk,pipe}} = 1/t_{\text{clk,pipe}} \leq 1/(300 \text{ ps}) = 3.33 \text{ GHz}$$



◦ Q3. 加速比多少？为什么小于五？

◦ 两者CPI都等于1

$$\text{Speed-up} = t_{\text{clk,single}} / t_{\text{clk,pipe}} = f_{\text{clk,pipe}} / f_{\text{clk,single}}$$

$$= 3.33 / 1.08 = 3.08.$$

因为流水线各个阶段不均衡，而且有额外流水线寄存器的开销 ($t_{\text{clk-to-q}}$, t_{setup}). 除此之外还有后面要讲的冒险现象引起的延迟.

在流水线CPU，执行以下指令：

lw

add

sub

\$6, 0(\$7)

\$8, \$9, \$10

\$11, \$6, \$8

问减法（子）指令的EX阶段处理的数据操作数有什么问题？在现有你对流水线的了解，有什么办法能得到正确的结果

解：画一个简单的图表

指令	C1	C2	C3	C4	C5	C6					
Lw \$6,0(\$7)	IF	ID	EX	MEM	WB						
Add \$8,\$9,\$10		IF	ID	EX	MEM	WB					
Sub \$11,\$6,\$8			IF	ID	EX	MEM					

指令

Lw \$6,0(\$7)

Add \$8,\$9,\$10

Sub \$11,\$6,\$8

C1

IF

C2


ID

IF

C3

EX


ID



C4

MEM

EX



C5

WB

MEM

IF

C6

WB

ID

C7

EX

C8

MEM

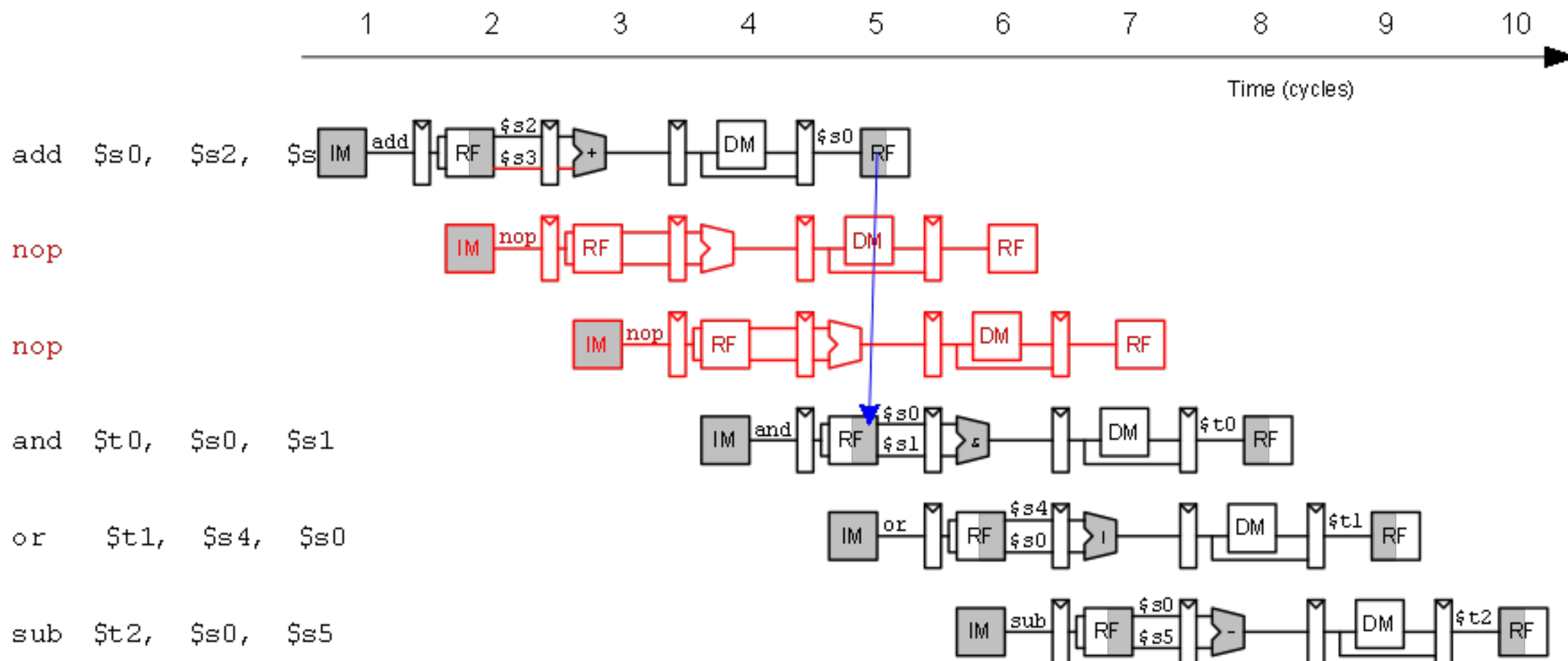
C9

WB

阻塞=使随后的指令等待，直到其源数据值可用

编译时间检测和消除

- 一条指令写入一个寄存器（\$s0），下一条指令读取该寄存器=>写入后读取（RAW）相关性。如果流水线不能正确地处理数据相关性，将造成错误的结果。



流相关性，由于架构寄存器数量有限，造成数据冒险。