计算机组成原理

文杰

计算机科学与技术学院

wenjie@hit.edu.cn

个人主页：http://faculty.hitsz.edu.cn/wenjie

第五章 流水线处理器

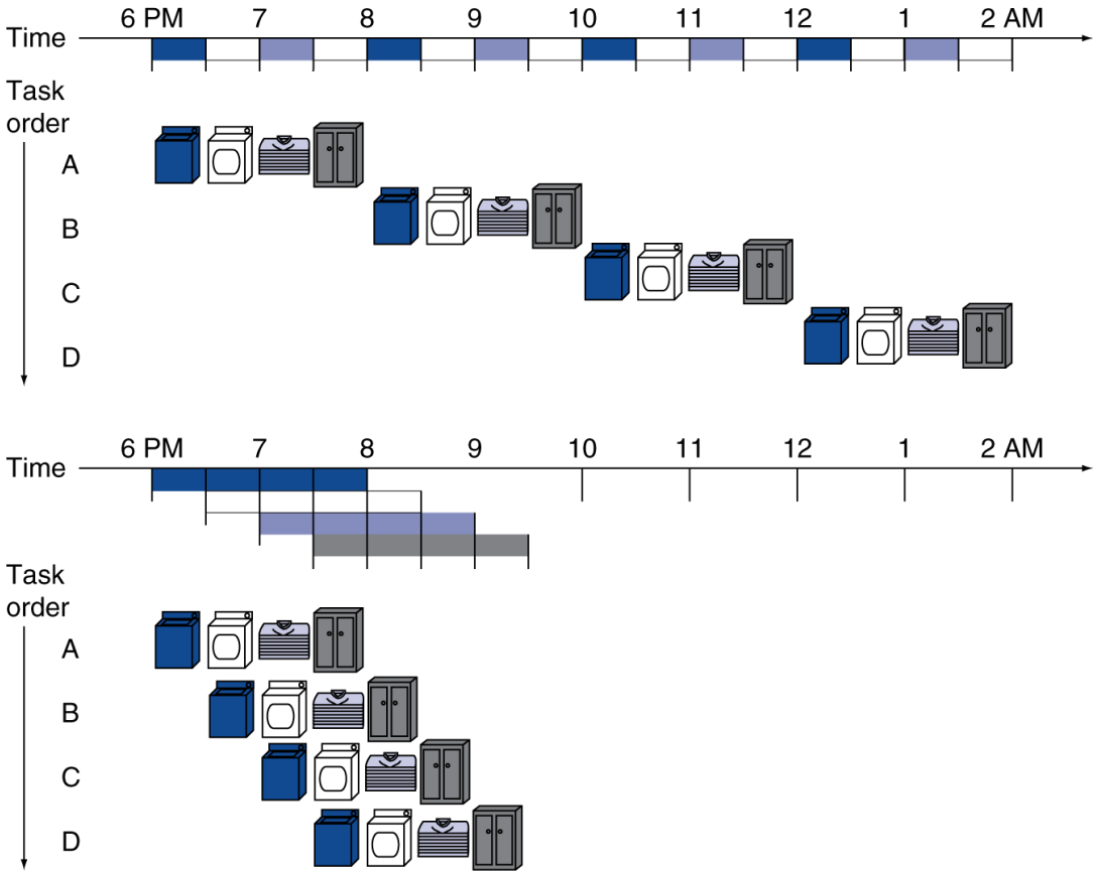
- **流水线概述**
- 流水线数据通路和控制
- 数据冒险：前递与停顿
- 控制冒险
- 例外

单周期处理器的性能问题

- **处理器中的关键路径决定了时钟周期**：单周期设计中，时钟周期对于每条指令必须相等；处理器中执行时间最长的指令决定了时钟周期
  - 例如load指令，经历了完整的五个步骤：指令存储器、寄存器堆、ALU、数据存储器、寄存器堆
- **缺陷：**
  - 虽然CPI为1，但由于时钟周期太长，单周期实现的整体性能可能很差。
  - 违反了设计思想：无法加速经常性事件，不能使用那些缩短指令执行时间而不改变最坏情况的实现技术
- **可通过流水线来提高性能**

流水线的概念介绍：一个比喻

- 以洗衣服为例类比流水线：重叠执行
  - 假设洗衣包括四个步骤：洗衣机中洗衣、烘干机中烘干、叠衣服、收纳到柜子中，每个步骤0.5小时



- 负载为4:
  - 加速比
    = 8/3.5 = 2.3
- 负载足够多:
  - 加速比
    = 2n/(0.5n+1.5) ≈ 4
    = 流水线中步骤的数目
    （流水线的级数）

RISC-V 指令执行的五个阶段

- 五个阶段
  - IF (instruction fetch): 从存储器中取出指令
  - ID (instruction decode): 读寄存器并译码指令
  - EX (execute): 执行操作或计算地址
  - MEM (memory access): 访问数据存储器中的操作数（如有必要）
  - WB (write back): 将结果写入寄存器（如有必要）
- 让五个阶段重叠执行

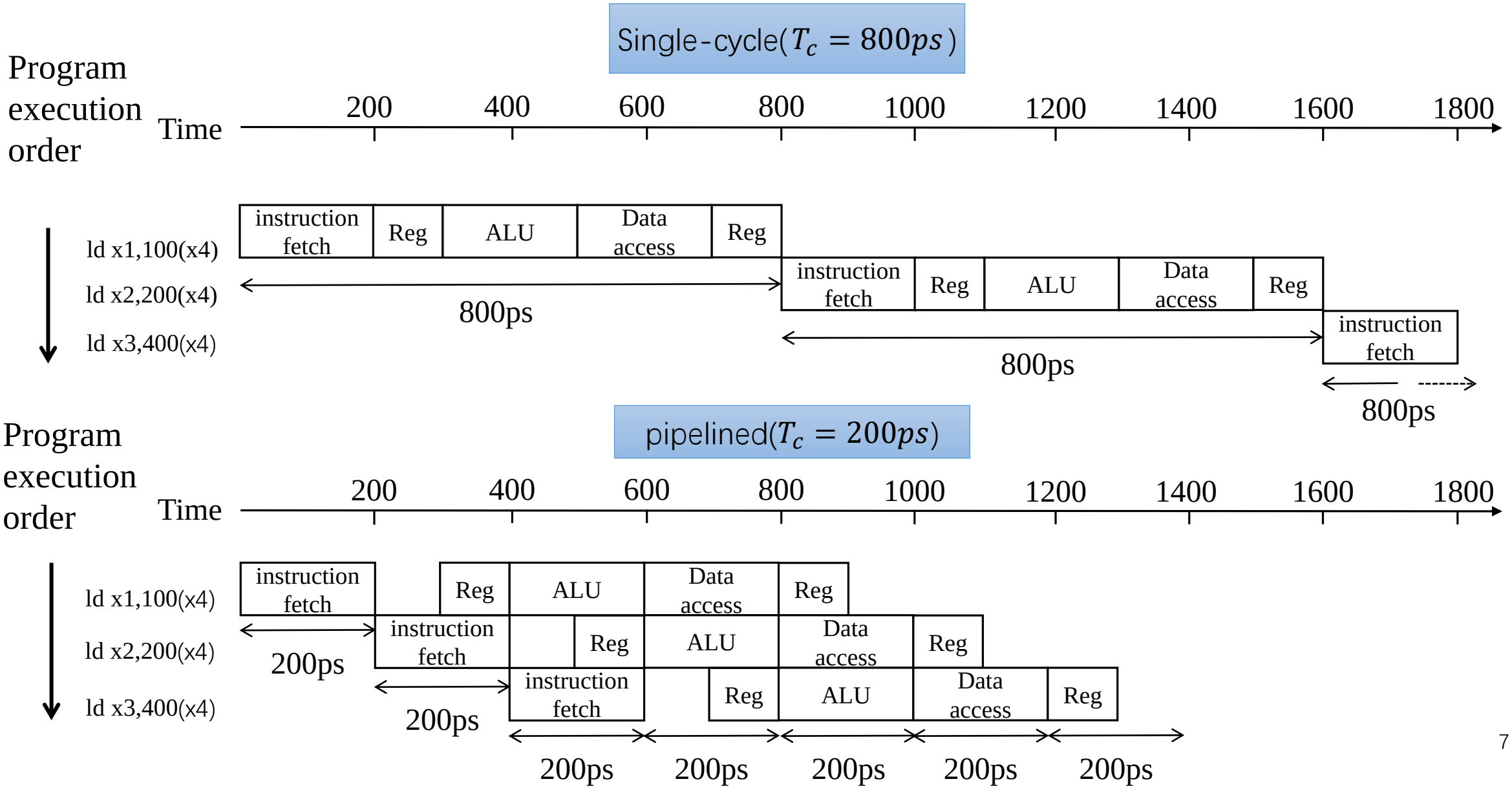
流水线性能

- 假设各个阶段的耗时：
  - 寄存器堆的读或写为100ps
  - 其他阶段为200ps

- 比较流水线指令执行与单周期指令执行的平均执行时间

指令类型	取指令	读寄存器	ALU 操作	数据存取	写寄存器	总时间
Load doubleword (ld)	200ps	100 ps	200ps	200ps	100ps	800ps
Store doubleword (sd)	200ps	100 ps	200ps	200ps		700ps
R-format (add, sub, and, or)	200ps	100 ps	200ps		100ps	600ps
Branch (beq)	200ps	100 ps	200ps			500ps

流水线性能


Single-cycle($T_c = 800ps$)
Program execution order
Time
200
400
600
800
1000
1200
1400
1600
1800
ld x1,100(x4)
ld x2,200(x4)
ld x3,400(x4)
instruction fetch
Reg
ALU
Data access
Reg
800ps
800ps
800ps
pipelined($T_c = 200ps$)
Program execution order
Time
200ps
200ps
200ps
200ps
200ps
200ps

流水线加速比

- 负载足够多:
  - 加速比
    = 2n/(0.5n+1.5) ≈ 4
    = 流水线中步骤的数目
    （流水线的级数）

- 如果流水线各阶段操作平衡
  - 例如：所有阶段需要相同的时间

    则：指令执行时间$_{流水线}$≈指令执行时间$_{非流水线}$/流水线级数
- 在理想条件和有大量指令的情况下，流水线带来的加速比约等于流水线的级数
- 若各阶段不完全平衡，加速比会变小
- 通过提高指令吞吐率来提高性能
  - 注意：单个指令的执行时间没有减少

面向流水线的指令系统设计

- RISC-V 指令系统是面向流水线设计的
  - 所有RISC-V指令长度相同，都是32 bits
    - 简化了流水线第一阶段取指令和第二阶段指令译码
    - x86: 1-17 bytes的变长指令不利于流水线实现
  - 只有6种指令格式，格式整齐，如源寄存器和目标寄存器位置相同
    - 能在一个阶段内完成译码和读寄存器
  - 存储器操作只出现在load/store指令中
    - 可以利用执行阶段来计算存储器地址，然后在下一阶段访问存储器

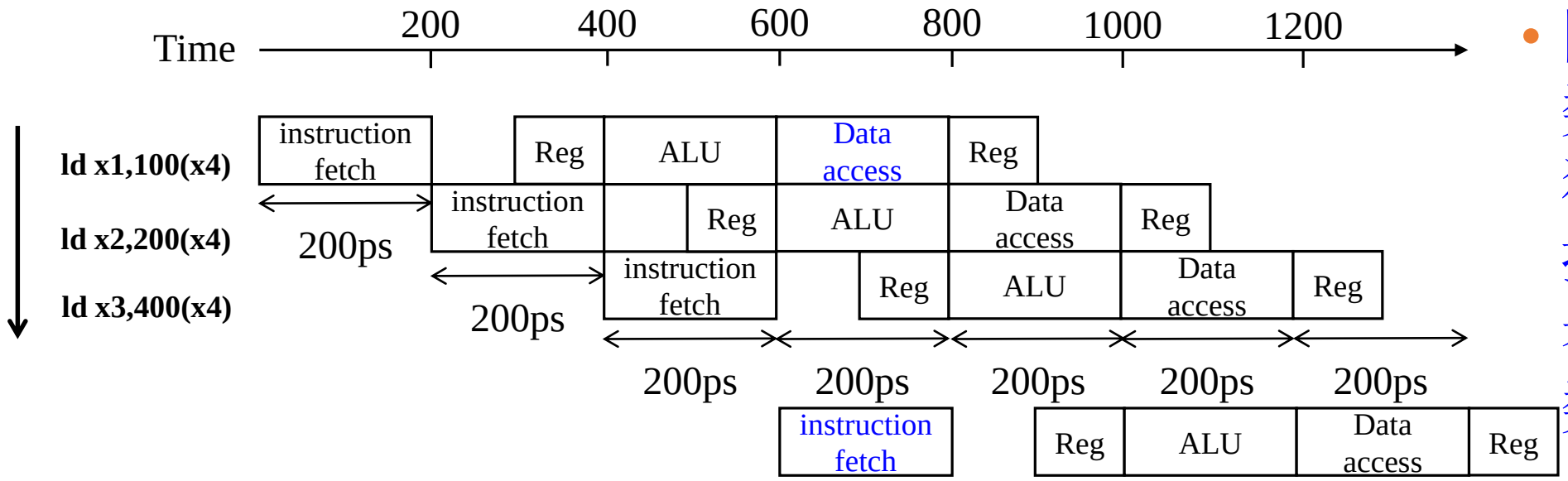
流水线冒险

流水线中有一种情况，在下一个时钟周期中下一条指令无法执行，这种情况被称为流水线冒险(hazard)

- 结构冒险
  - 因**缺乏硬件支持**而导致指令不能在预定的时钟周期内执行的情况
- 数据冒险
  - 因**无法提供指令执行所需数据**而导致指令不能在预期的时钟周期内执行的情况。如上下指令依赖。
- 控制冒险（分支冒险）
  - 由于**取到的指令并不是所需要的**，或者**指令地址的流向不是流水线所预期**的，导致正确的指令无法在正确的时钟周期内执行的情况

结构冒险

- **硬件资源**不支持多条指令在同一时钟周期执行
- 假设下图RISC-V流水线结构只有一个存储器
  - load/store需要访问存储器
  - 如果有第四条指令，则第一条指令从存储器取数据的同时，第四条指令从同一存储器取指令，流水线发生结构冒险

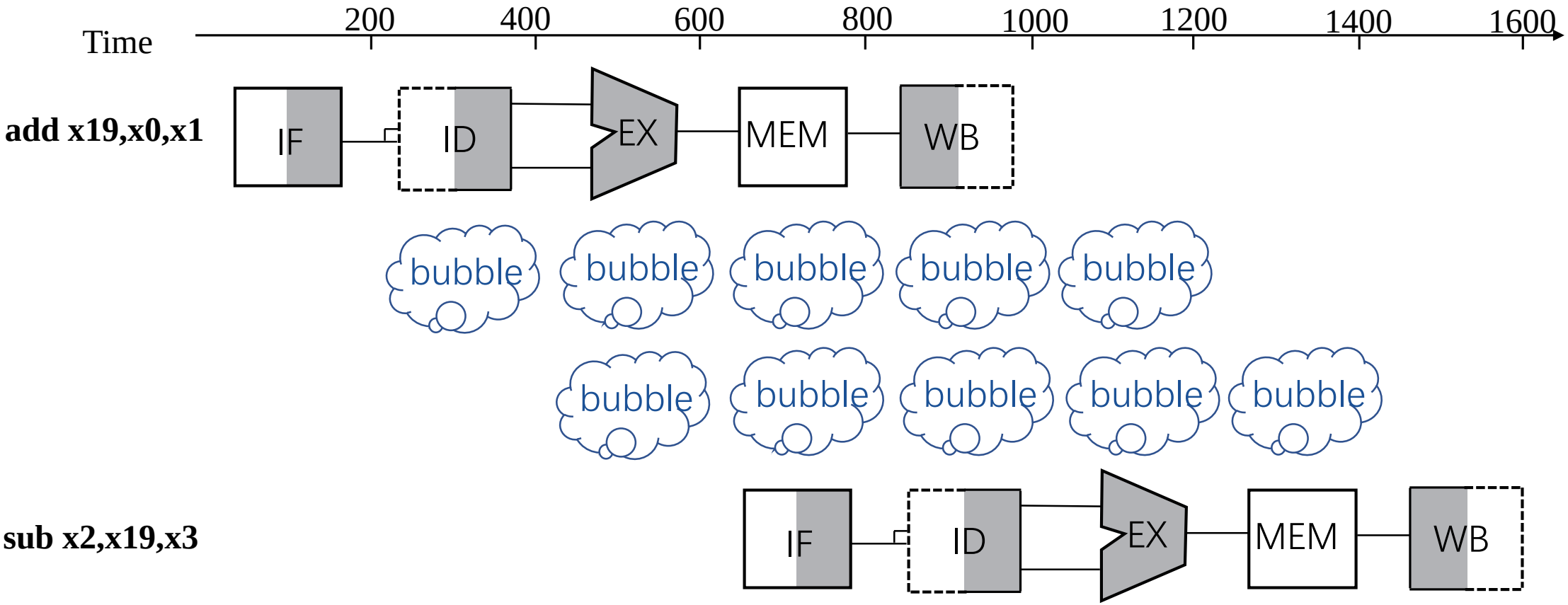


- 因此，流水线数据通路需要独立的指令/数据存储器，或者独立的指令/数据cache

数据冒险

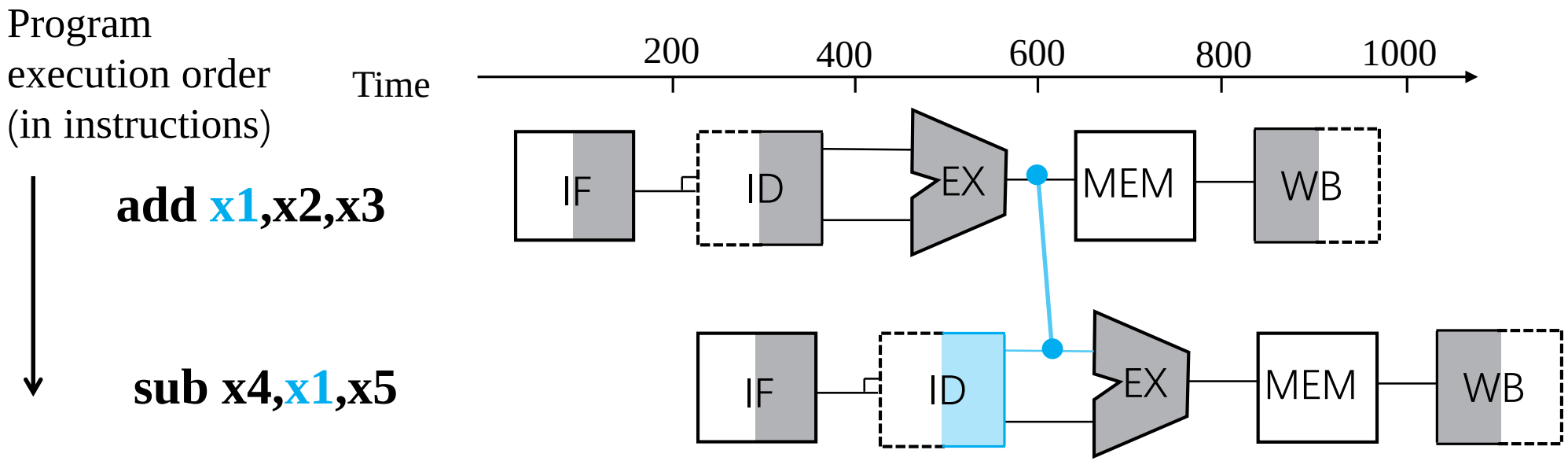
- 一条指令依赖于前面一条尚在流水线中的指令运行结果

```
add    x19, x0, x1
sub    x2, x19, x3
```



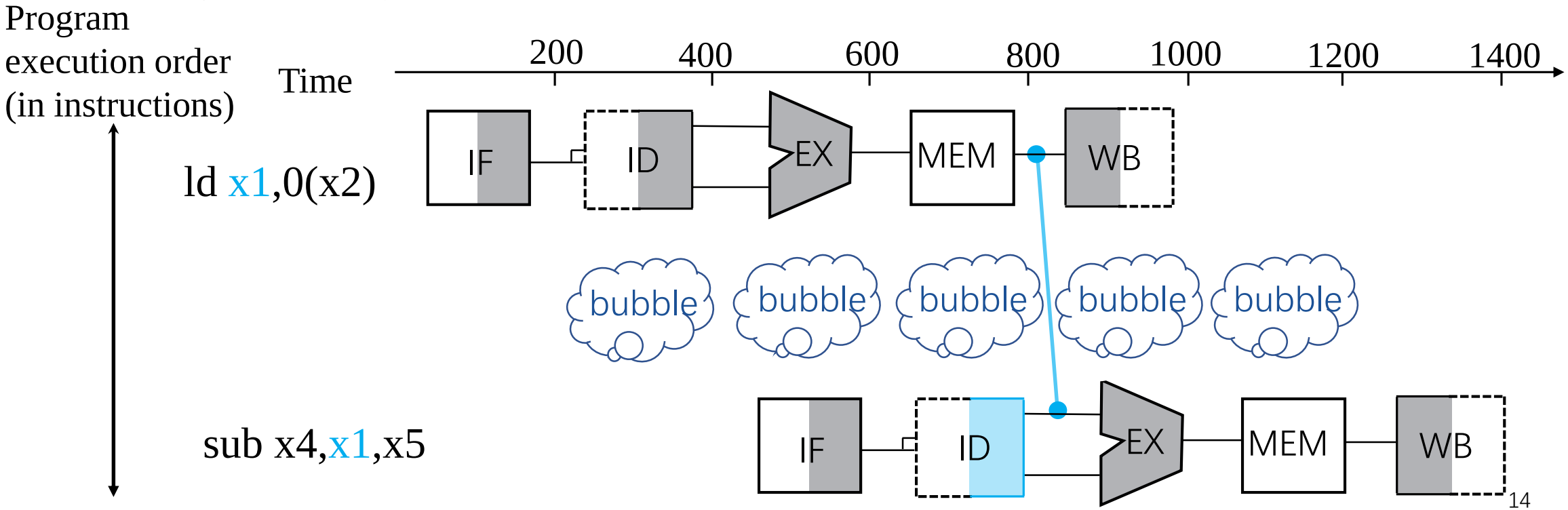
前递(forwarding)

- 一种解决数据冒险的方法
  - 一旦ALU计算出结果，就提前取到数据，而不是等到数据到达寄存器或存储器
  - 需要向内部资源添加额外的硬件
    - 寄存器堆或存储器右半部分为阴影：正在被读
    - 反之，则正在被写



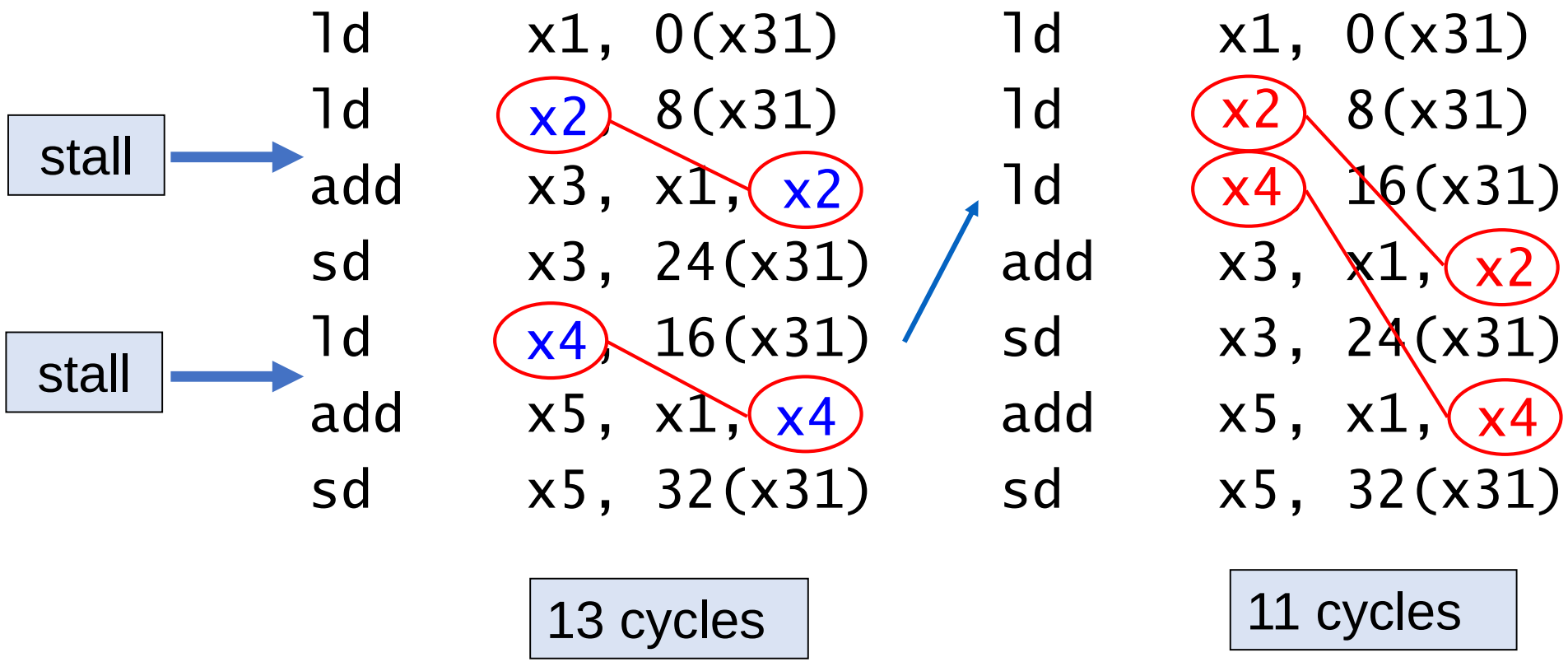
载入-使用型数据冒险(load-use data hazard)

- 前递不能避免所有的流水线停顿
  - 当载入指令要取的数据还没被取回，其他指令就需要该数据时，无法及时前递
  - 一种解决方案：流水线停顿(stall)，也称为气泡(bubble)，为了解决冒险而实施的一种阻塞



重排代码以避免流水线停顿

- 重排代码，避免在load指令之后立即使用取到的数据
- C code: a = b + e; c = b + f;



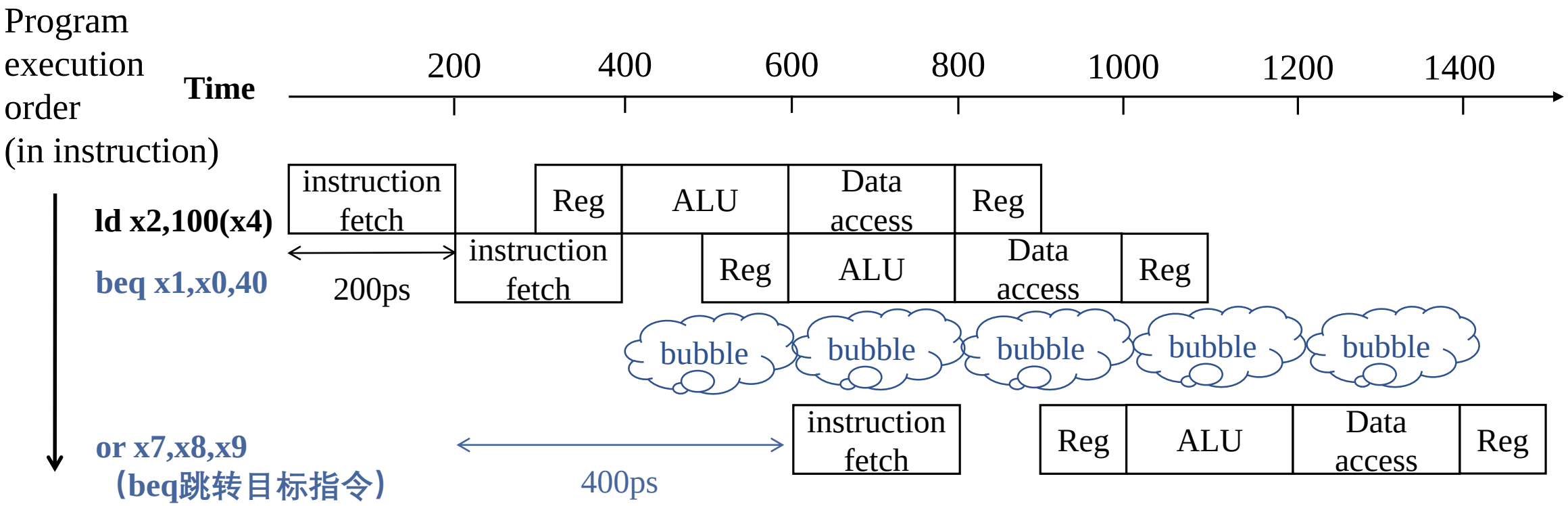
控制冒险

- 控制冒险也称为分支冒险（分支决定了控制流）：由于**取到的指令并不是所需要的，或者指令地址的流向不是流水线所预期**的，导致正确的指令无法在正确的时钟周期内执行的情况
  - 由于分支指令之后的IF阶段依赖于分支的结果
  - 此时，分支指令仍在ID阶段，无法决定下一条要运行的指令

- 解决方案之一
  - 在RISC-V流水线中尽可能早地完成寄存器比较、分支目标地址计算
  - 具体方案：添加硬件，使得以上能在ID阶段完成

方案一：每遇到条件分支指令就停顿的流水线

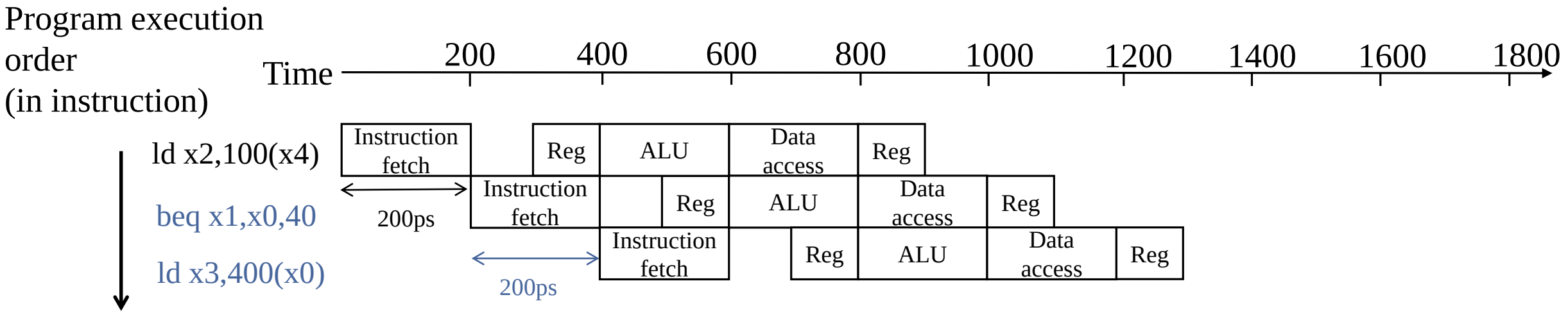
- 在取下一条指令之前，等待分支结果
  - beq指令的ID阶段“通过其他硬件电路”得到分支结果

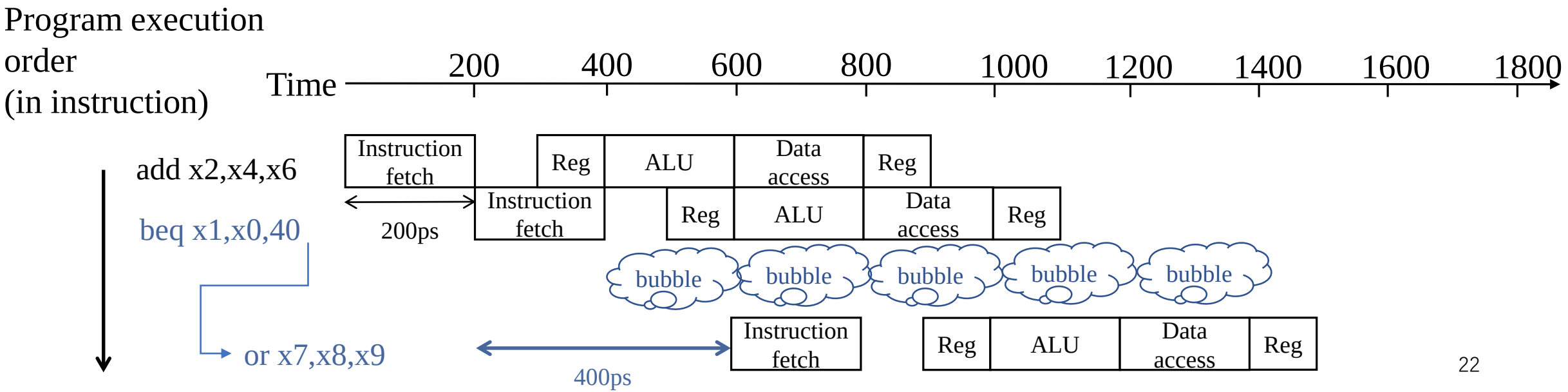


方案二：分支预测

- 对较长流水线而言，通常无法在第二阶段解决分支指令问题
  - 每个条件分支都停顿带来的代价太大，不可接受
- 采用**预测**来处理条件分支
  - 只有预测错误时，才引发停顿
- 例如：总是预测分支指令不跳转
  - 在分支指令之后，立即取指，流水线全速前进
  - 分支指令发生跳转时，流水线才会停顿

总是预测条件分支不发生跳转





两种更成熟的分支预测

- 静态分支预测
  - 根据典型的分支行为来决定是否跳转
  - 例如：循环+条件指令产生的分支
    - 在计算机程序中，循环底部是条件分支指令，并会跳转回循环顶部
    - 很可能发生分支并向回跳转，所以可以预测发生分支并跳到靠前的地址处
- 动态分支预测
  - 根据每个分支指令的行为进行预测
    - 一种常用实现方式：保存每个条件分支是否发生分支的历史纪录
  - 假设未来的行为会延续这个趋势
    - 当预测错误时，需要停顿、重新取指令，并更新历史记录

流水线总结

- 流水线通过提高吞吐率来提升性能
  - 并行执行多条指令
  - 每条指令拥有相同的延迟
    - 延迟：执行单条指令的时间
- 会受到冒险的影响
  - 结构、数据、控制
- 指令集的设计影响到流水线实现的复杂度：指令长度相同、指令格式整齐、存储操作只发生在load/store指令中（RISC-V）。

填空题 3分

设置

下面的程序在流水线处理器中执行时，程序 3 无需前递和停顿，程序 1 需要停顿、程序 2 可使用前递避免停顿。

作答

程序1

```
ld x10, 0(x5)
add x11, x10, x10
```

程序2

```
add x11, x10, x10
addi x12, x10, 5
addi x14, x11, 5
```

程序3

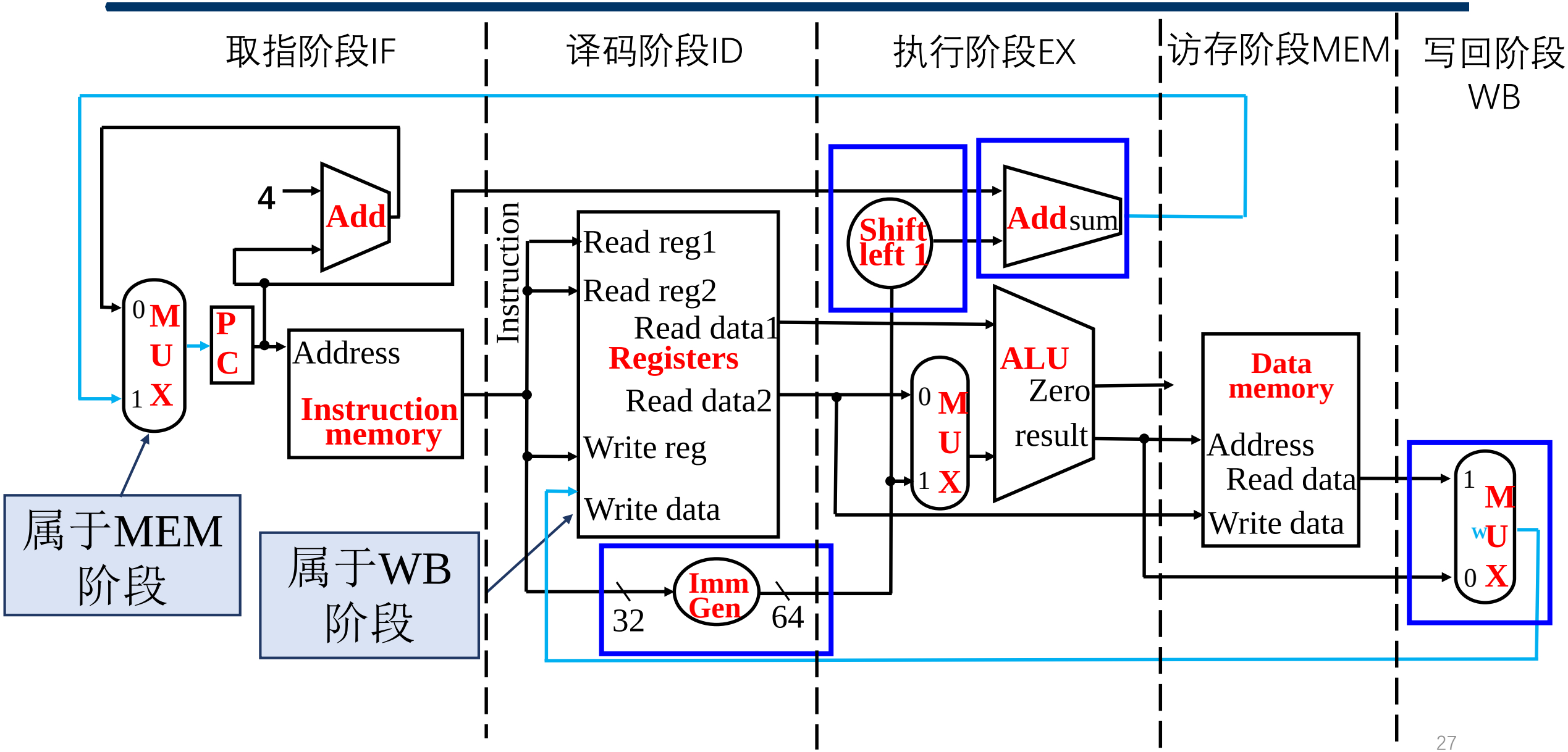
```
addi x12, x10, 5
addi x14, x10, 5
addi x16, x10, 7
addi x13, x10, 9
addi x17, x10, 8
addi x15, x10, 15
```

正常使用填空题需3.0以上版本雨课堂

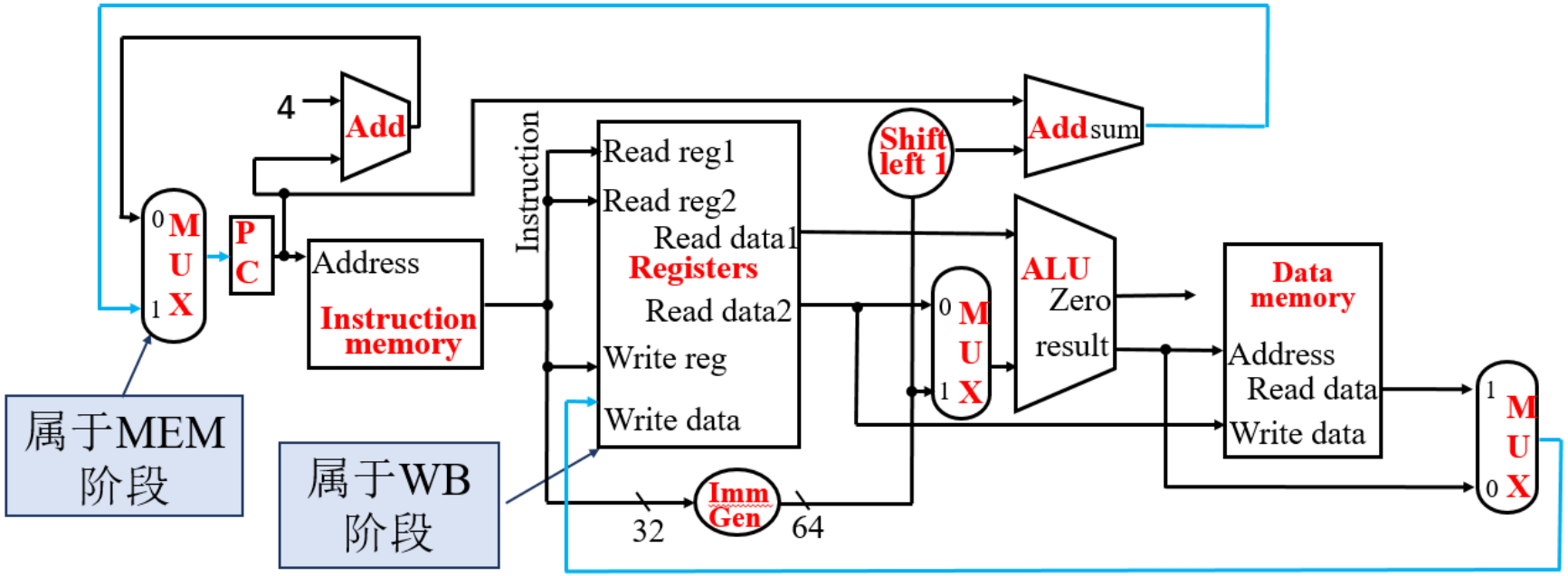
第五章

- 流水线概述
- **流水线数据通路和控制**
- 数据冒险：前递与停顿
- 控制冒险
- 例外

构建RISC-V流水线数据通路



构建RISC-V流水线数据通路



洗衣过程不会逆向移动，但是在从左到右的指令流动过程中，存在两个特殊情况：

- 写回阶段：将结果写回位于数据通路中的寄存器堆；
- 下一条指令PC的选择：在自增PC值与MEM(数据存储器访问)阶段的分支地址之间进行选择。

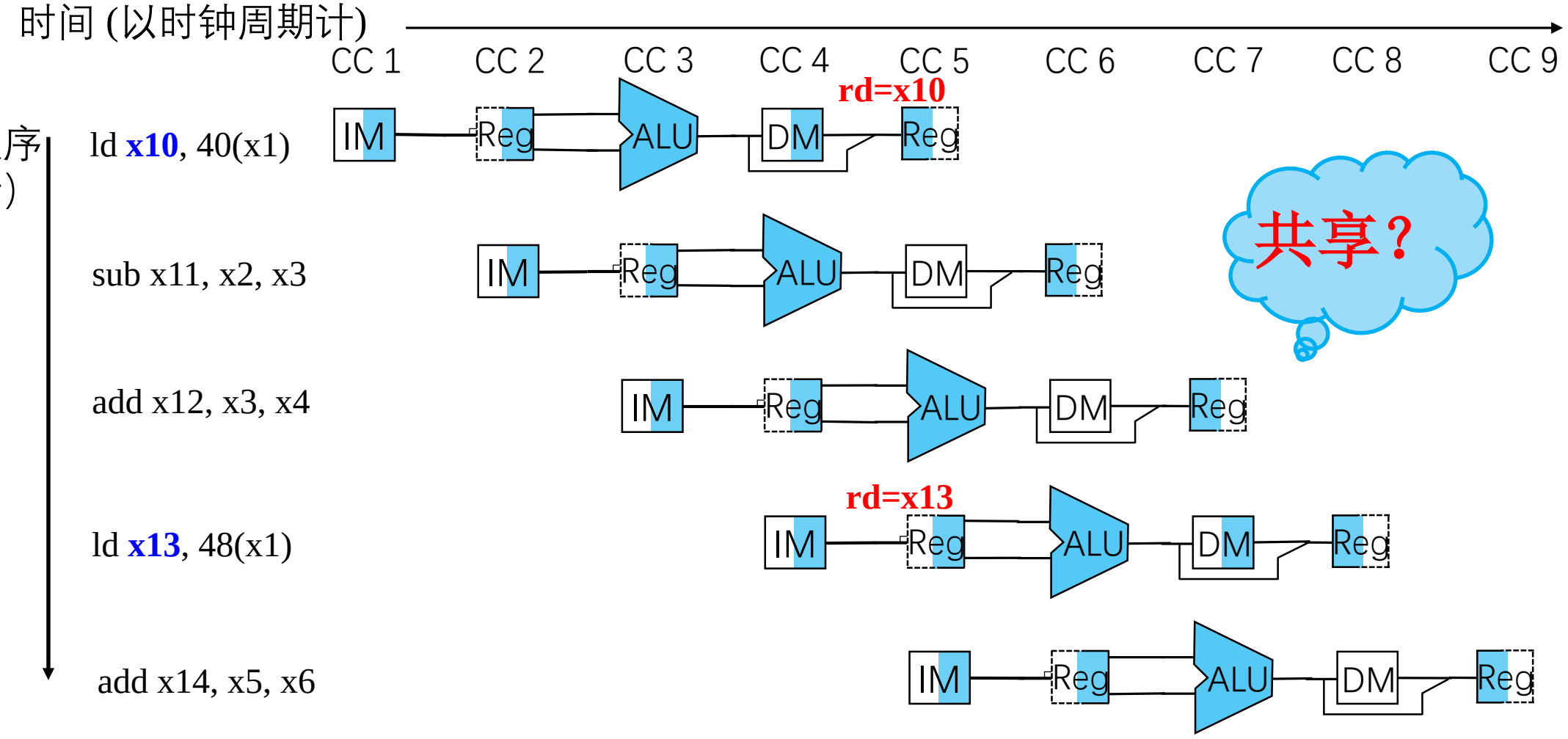
数据冒险

控制冒险

从右到左的数据流向可能引发冒险

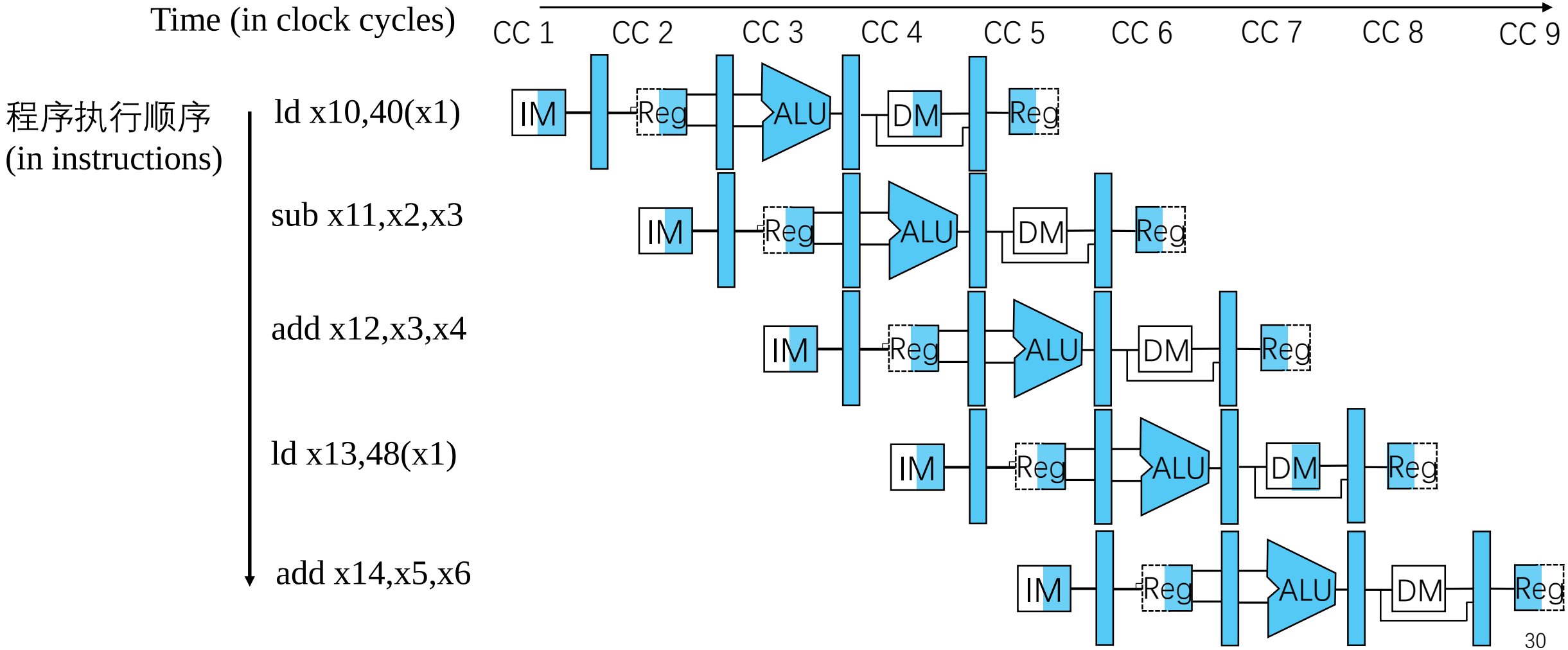
流水线通路如何执行？

- 每一条指令都有一条独立的数据通路？



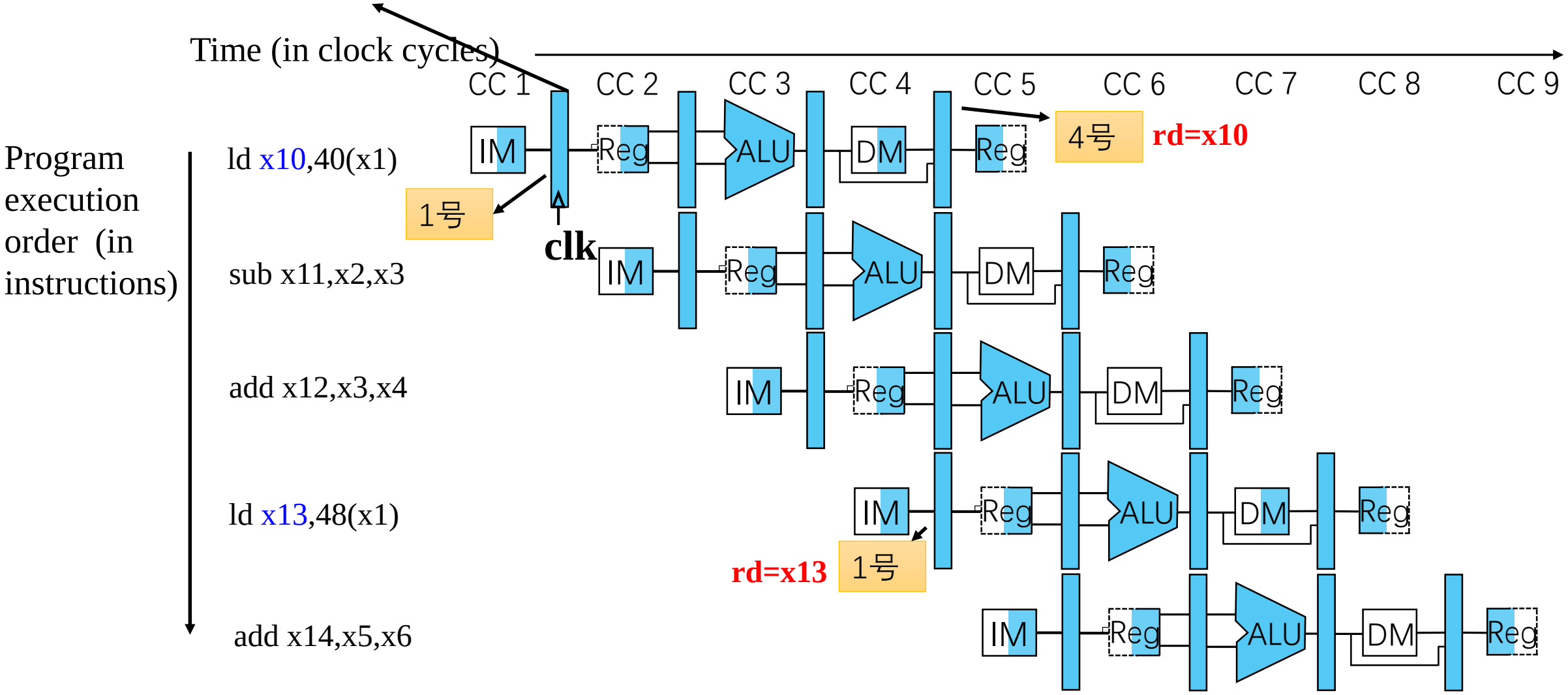
在流水线数据通路中观察指令的流动

Ripes：a graphical **processor simulator** and **assembly code editor** built for the RISC-V instruction set architecture （https://gitee.com/shzhou_admin/Ripes）



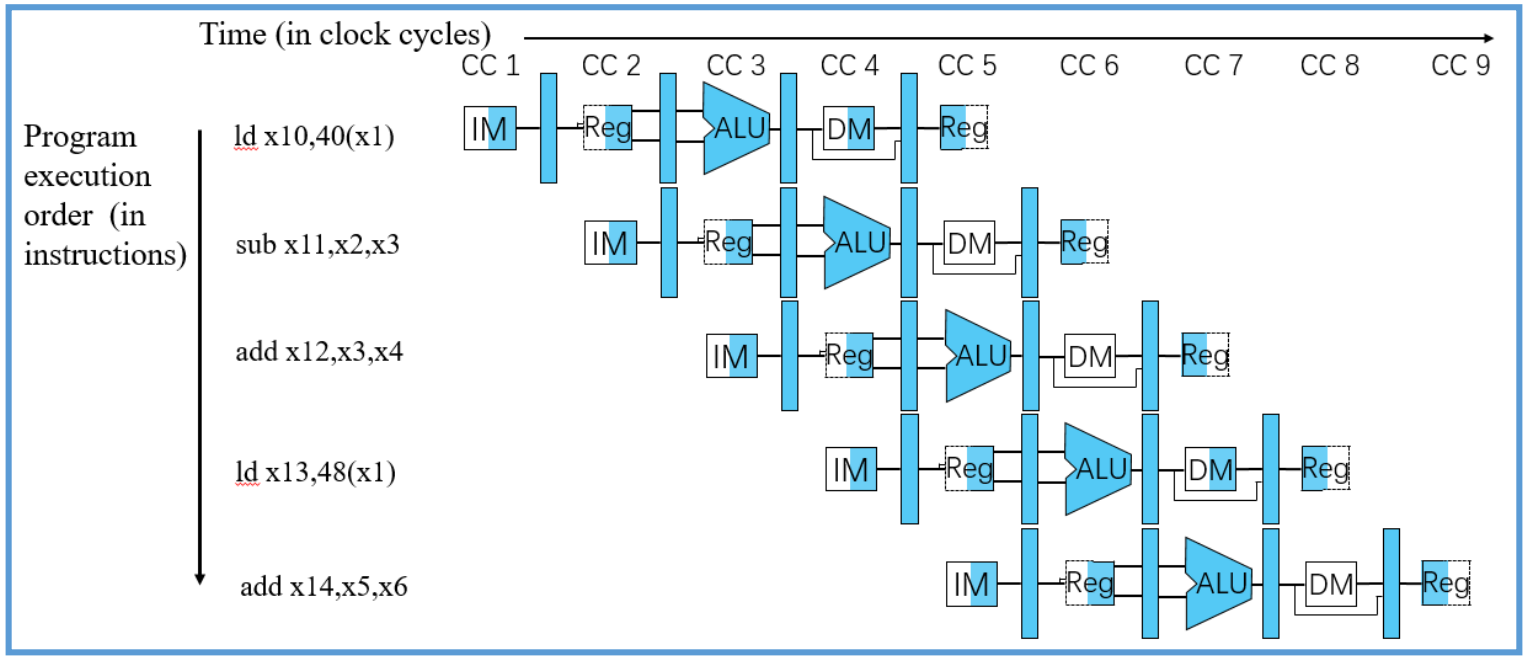
观察采用流水线寄存器的流水线操作

- 加入流水线寄存器——在流水线数据通路中，观察指令在不同周期的流动



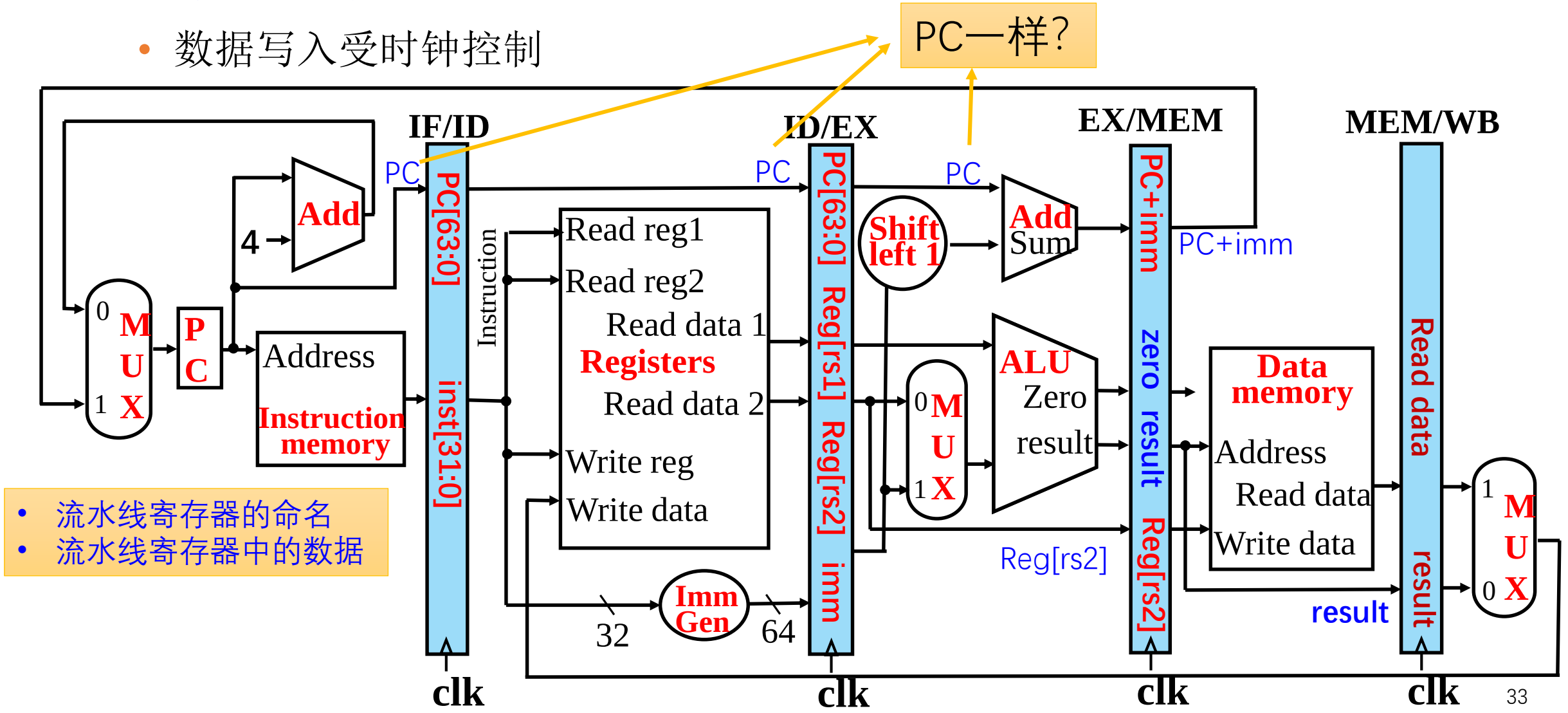
观察流水线操作

- 在流水线数据通路中，观察指令在不同周期的流动
  - 单时钟周期流水线图（一个时钟周期的垂直切片）
    - 展现流水线在指定时钟周期上每条指令对数据通路的使用情况
    - 高亮表示使用到的硬件资源
  - 多时钟周期流水线图
    - 展现一段时间的情况



加入流水线寄存器

- 流水线寄存器：保存前一个阶段产生的信息
  - 数据写入受时钟控制



- 流水线寄存器的命名
- 流水线寄存器中的数据

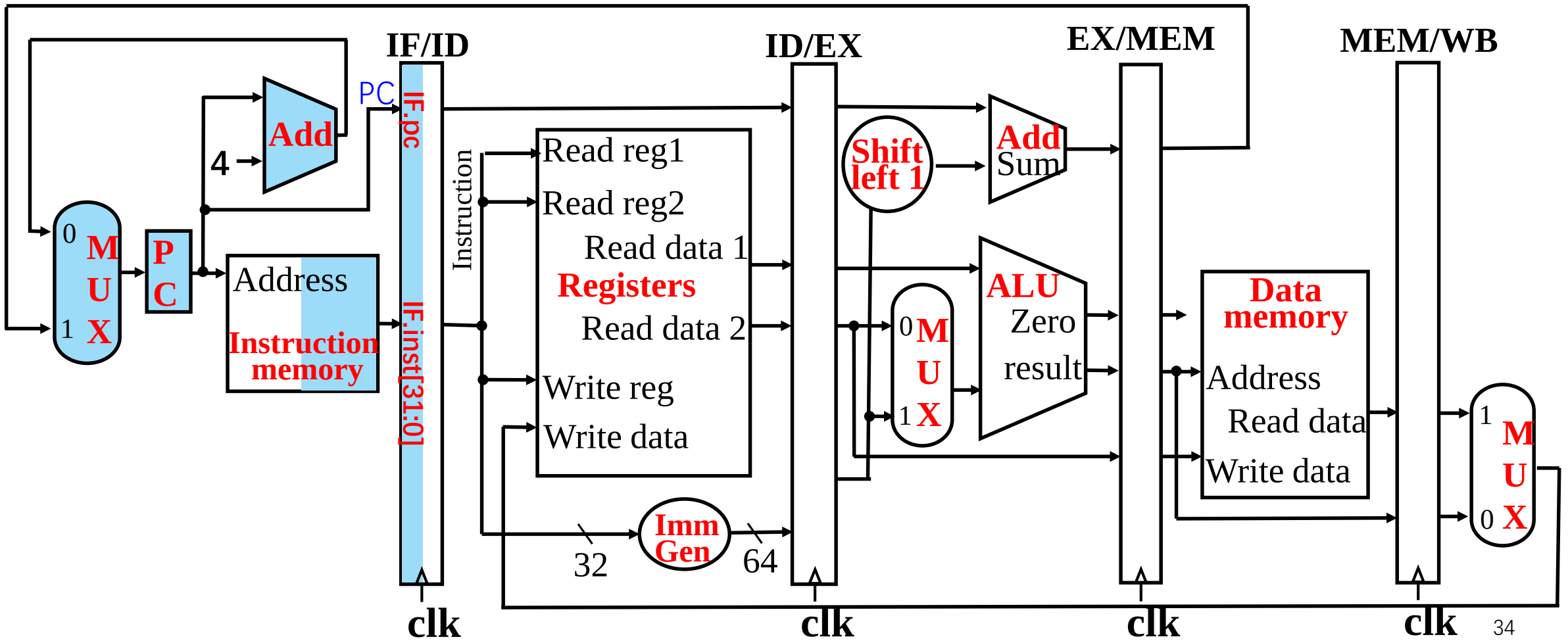
IF for Load, Store, …

观察load和store指令的操作

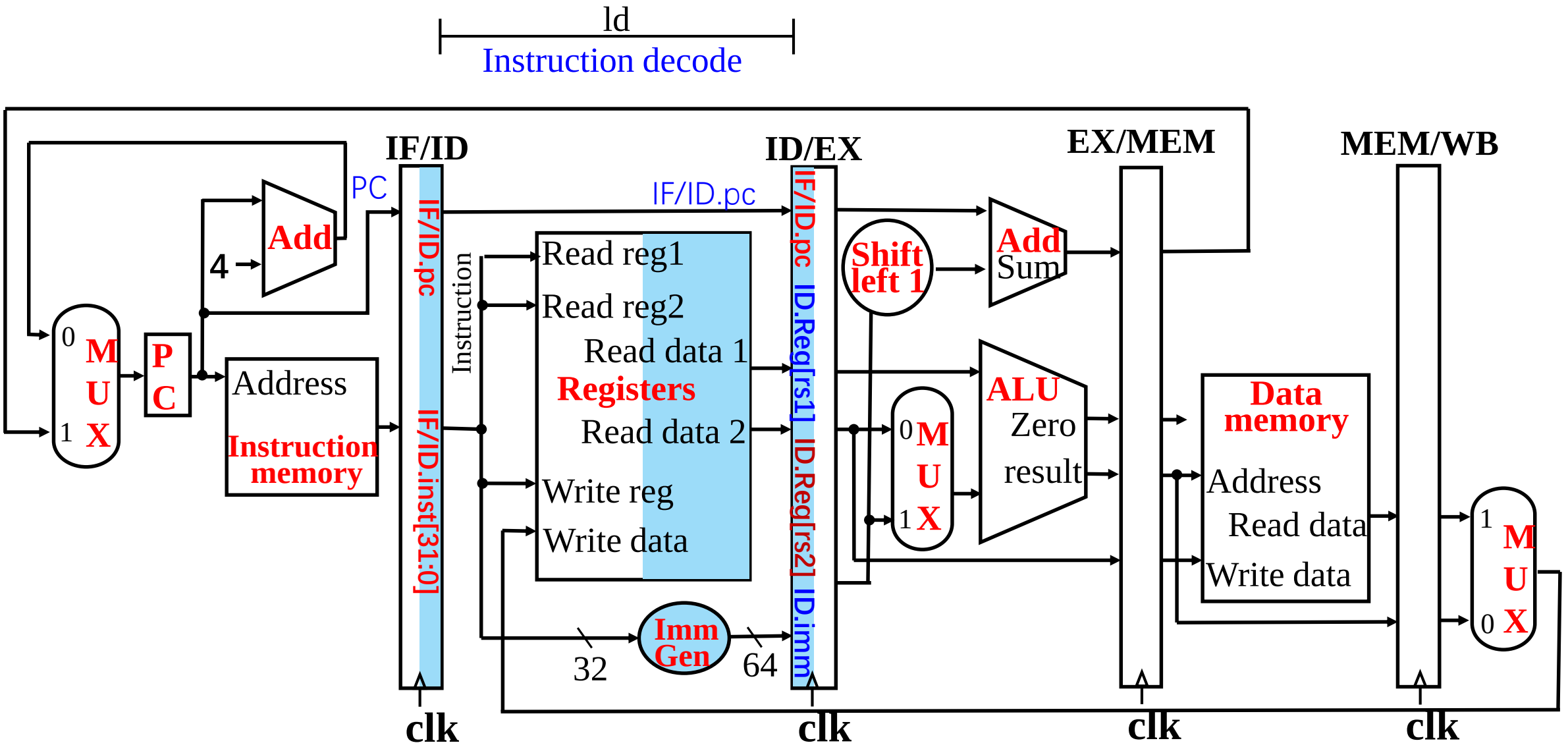
ld

Instruction fetch

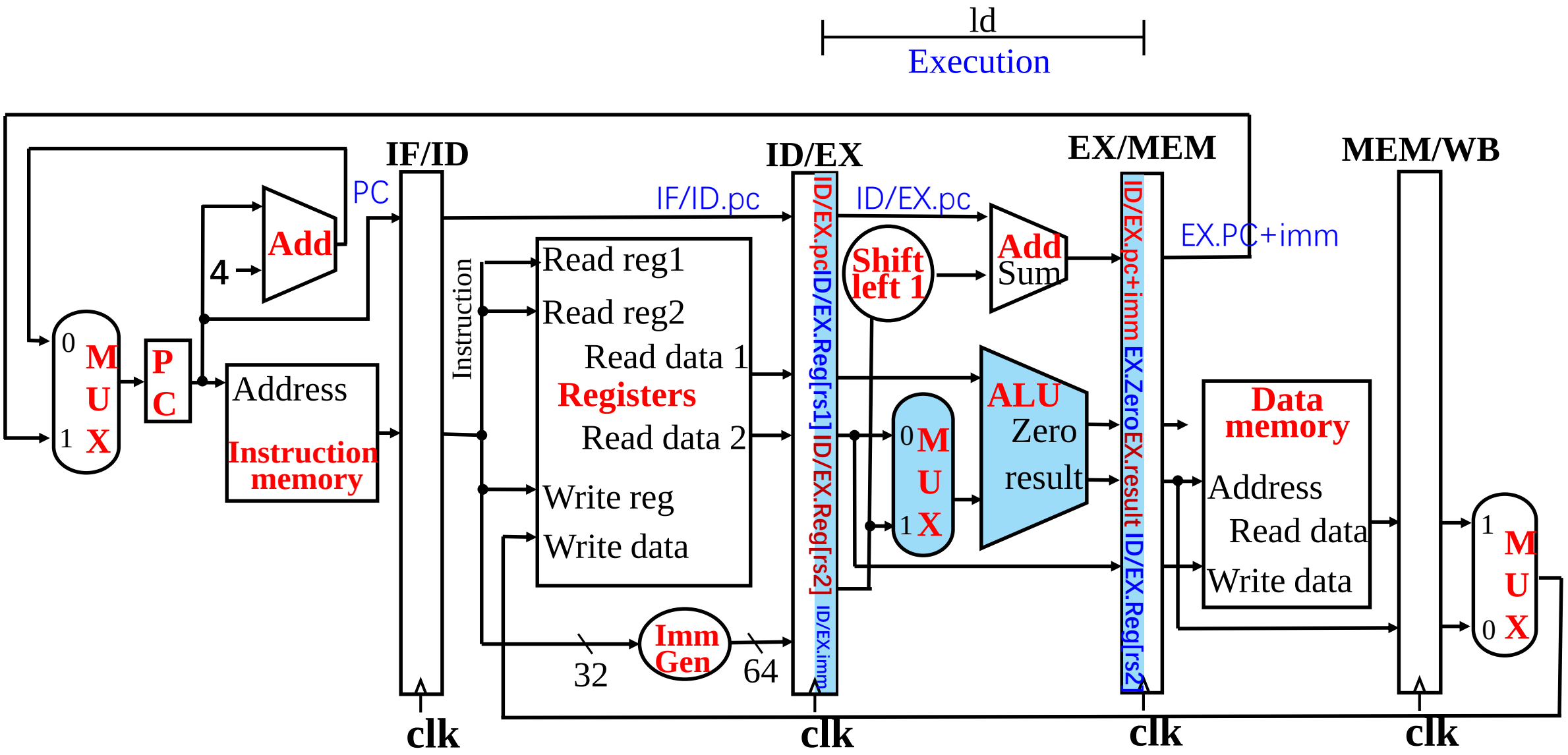
右半部分高亮：寄存器或存储器被读取
左半部分高亮：寄存器或存储器被写入



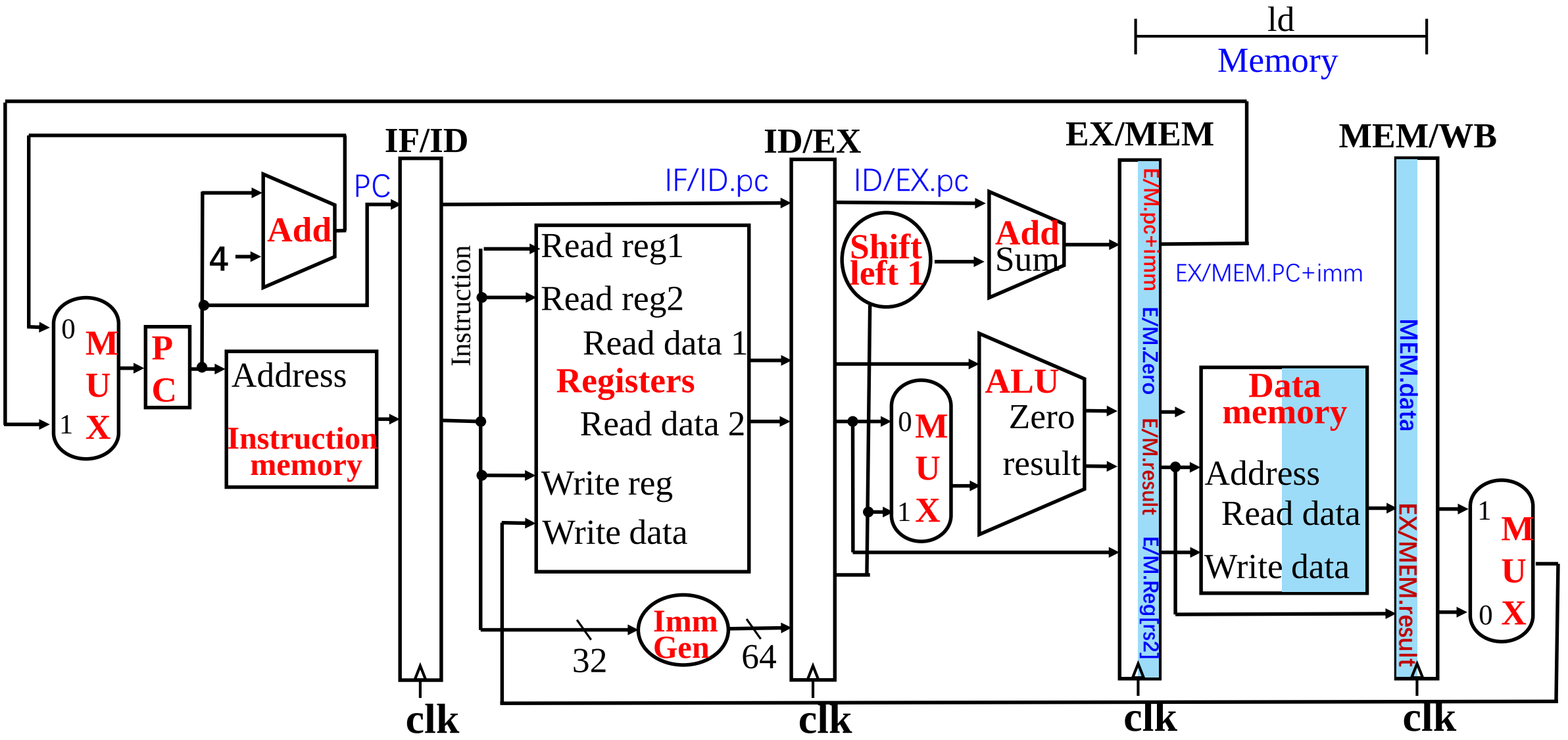
ID for Load, Store, …



EX for Load


ld
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
PC
IF/ID.pc
ID/EX.pc
EX.PC+imm
Add
4
M
U
X
0
1
P
C
Address
Instruction memory
Instruction
Read reg1
Read reg2
Registers
Read data 1
Read data 2
Write reg
Write data
Imm Gen
32
64
ID/EX.pc
ID/EX.Reg[rs1]
ID/EX.Reg[rs2]
ID/EX.imm
Shift left 1
Add
Sum
ALU
Zero
result
ID/EX.pc+imm
EX.Zero
EX.result
ID/EX.Reg[rs2]
Data memory
Address
Read data
Write data
clk

MEM for Load


ld
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
PC
IF/ID.pc
ID/EX.pc
Add
4
0
M
U
X
1
P
C
Address
Instruction memory
Instruction
Read reg1
Read reg2
Read data 1
Registers
Read data 2
Write reg
Write data
Imm Gen
32
64
Shift left 1
Add
Sum
ALU
Zero
result
E/M.pc+imm
E/M.Zero
E/M.result
E/M.Reg[rs2]
EX/MEM.PC+imm
Data memory
Address
Read data
Write data
MEM.data
EX/MEM.result
clk
clk
clk
clk

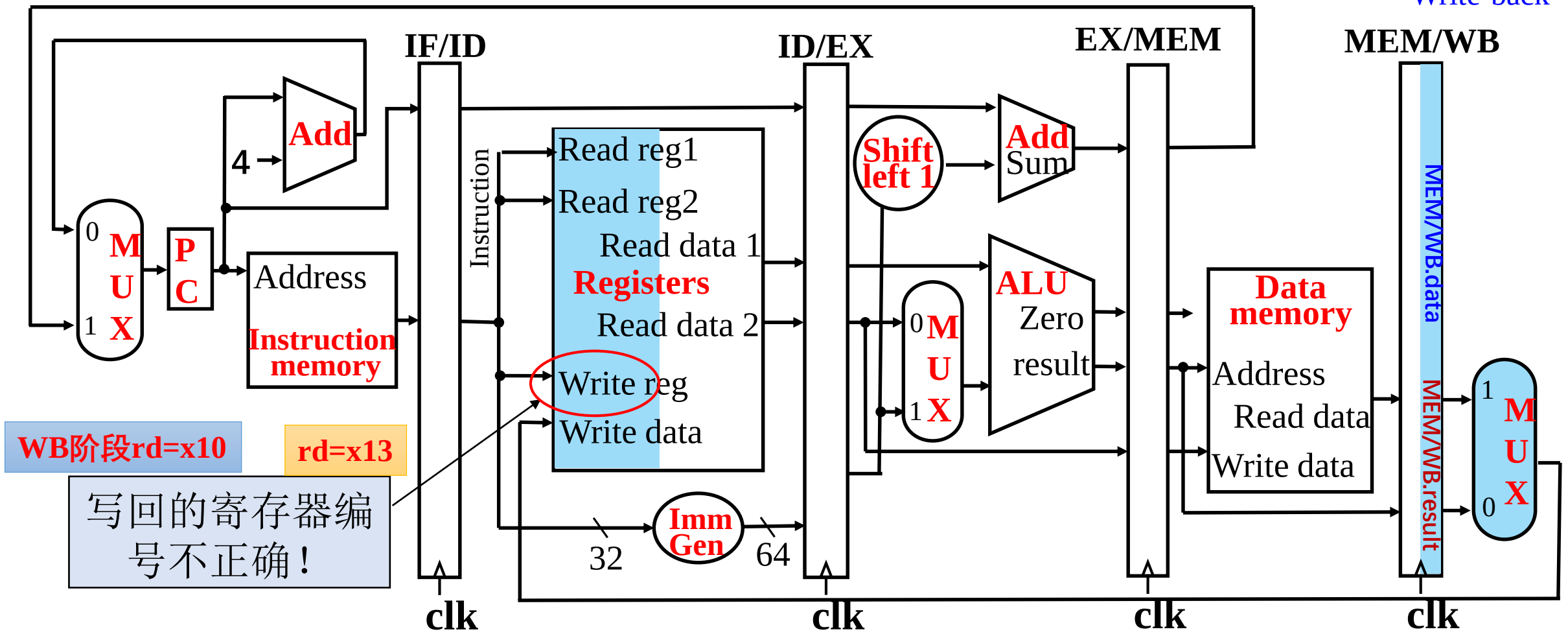
WB for Load

ld x13,48(x1)

ld x10,40(x1)

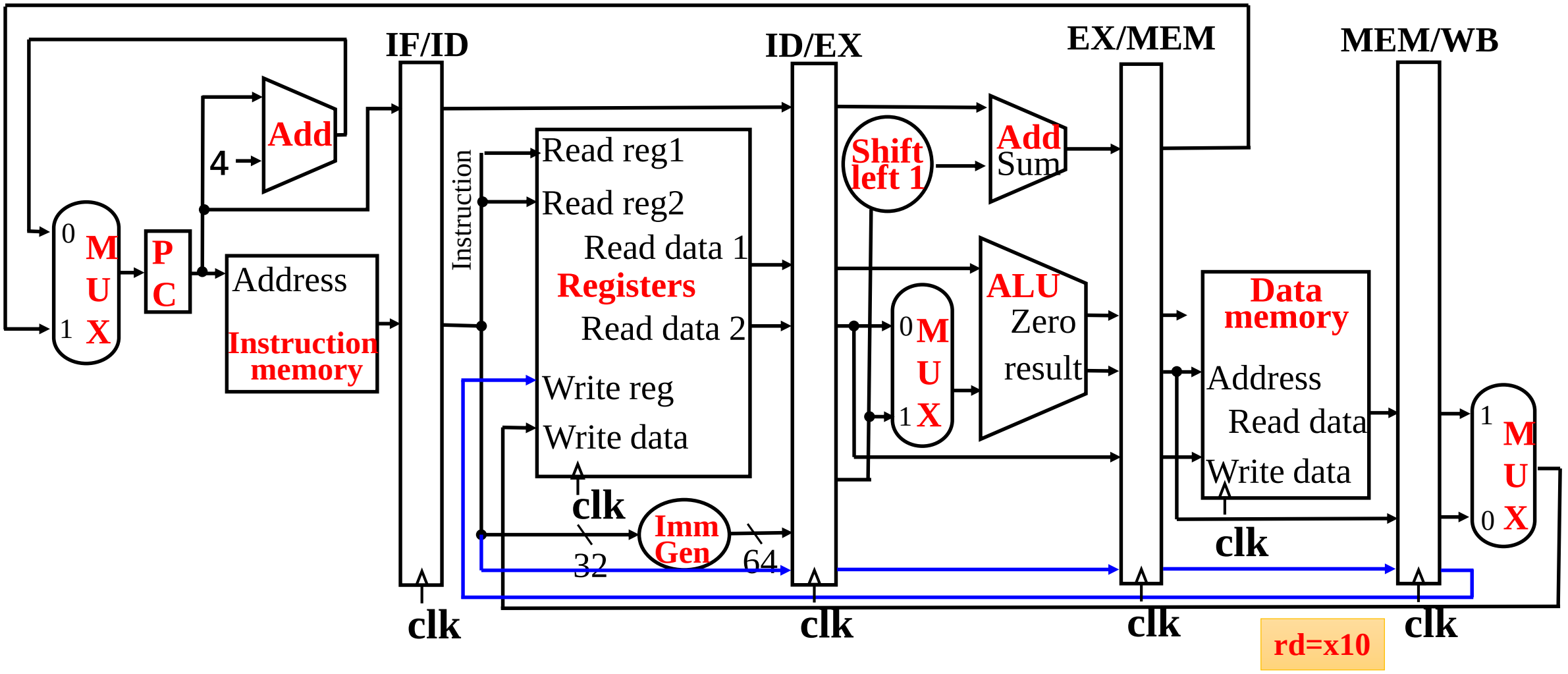
ld

Write-back

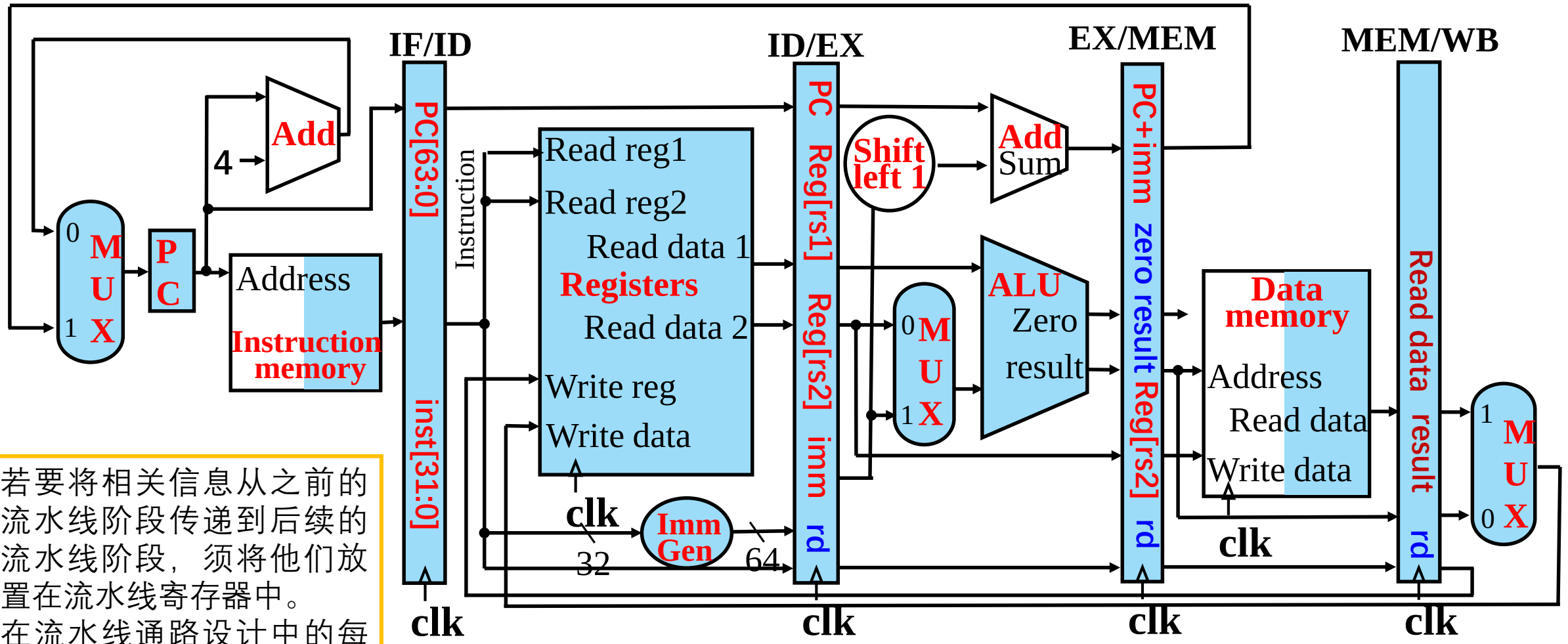


更正之后的数据通路

ld x10,40(x1)



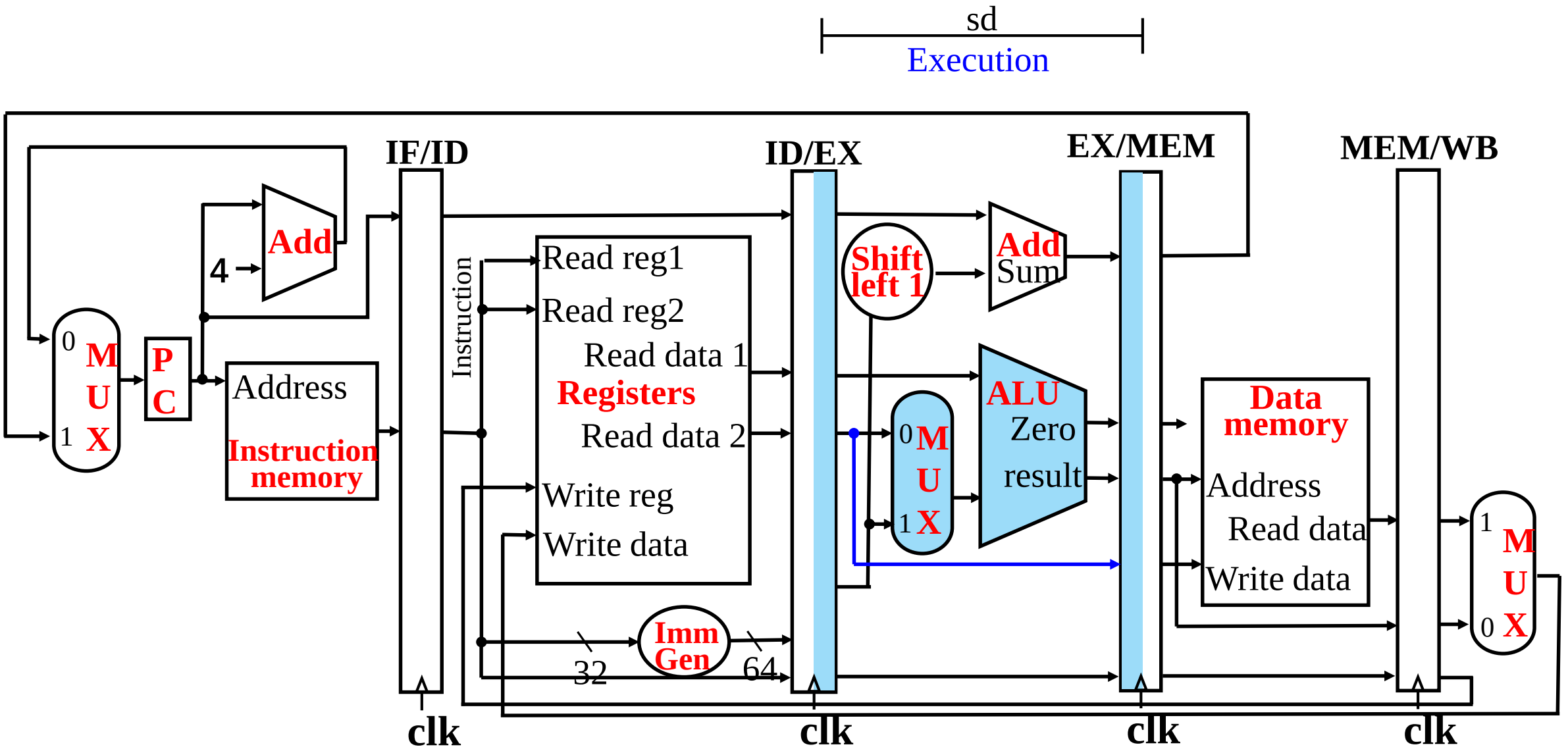
用于加载指令的全部五个流水线阶段部分



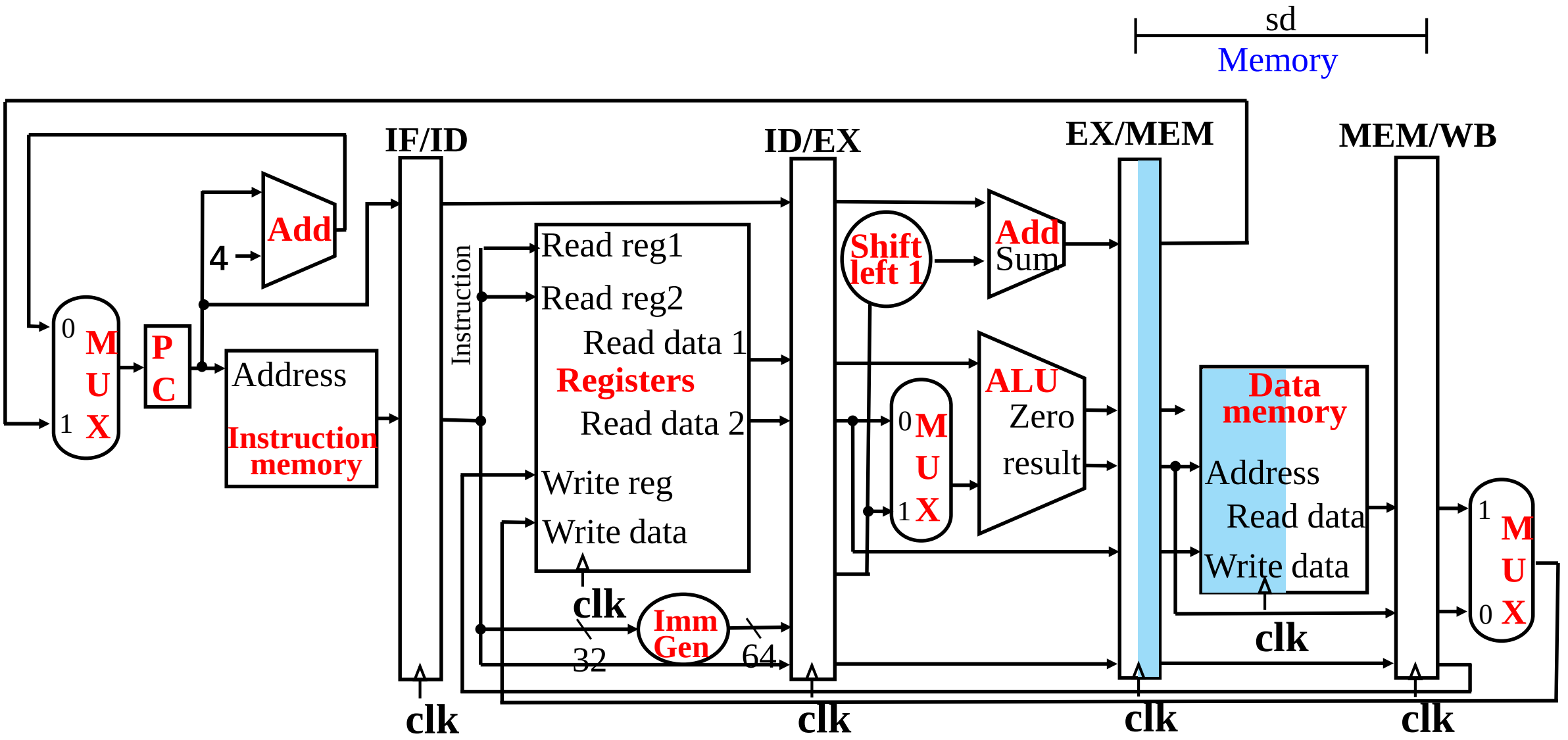
- 若要将相关信息从之前的流水线阶段传递到后续的流水线阶段，须将他们放置在流水线寄存器中。
- 在流水线通路设计中的每一个逻辑部件，只能在单个流水线阶段中被使用，否则会发生结构冒险。

EX for Store

memop rs2, offset(rs1)



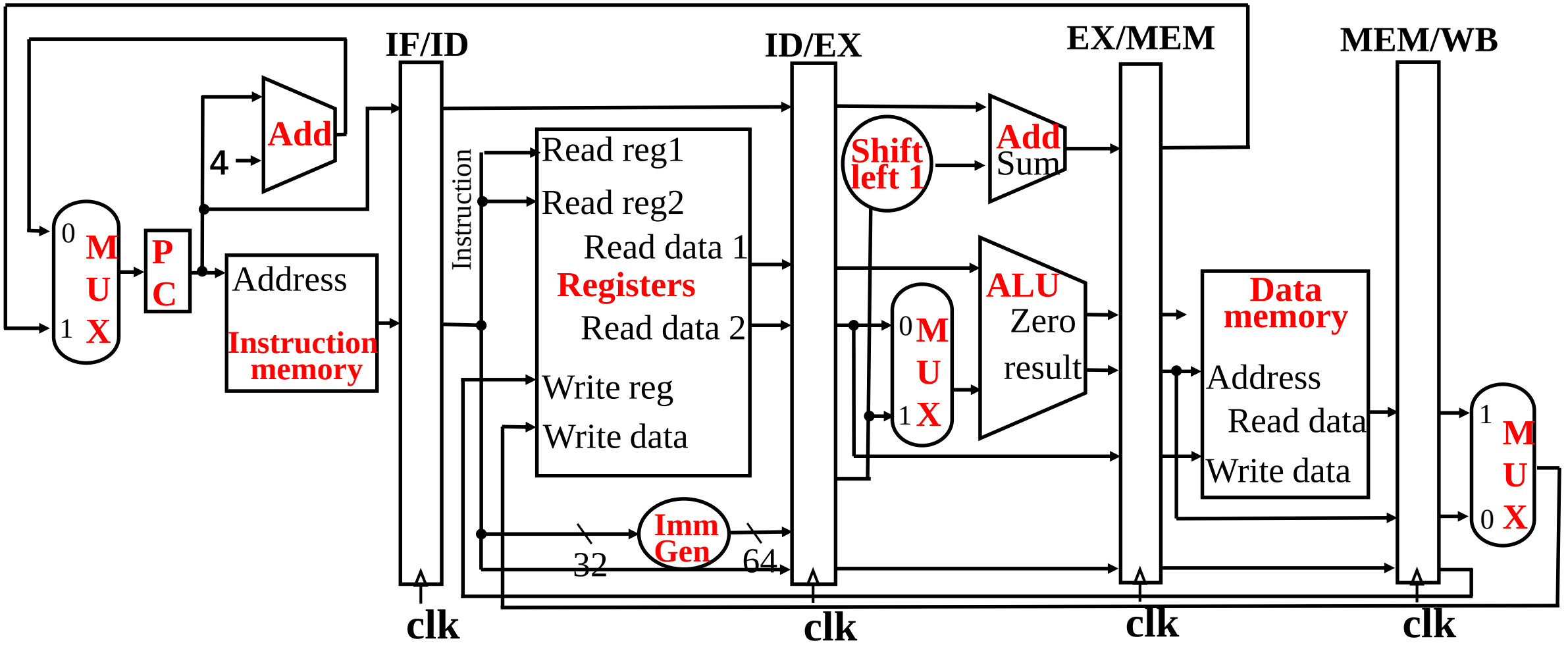
MEM for Store



WB for Store

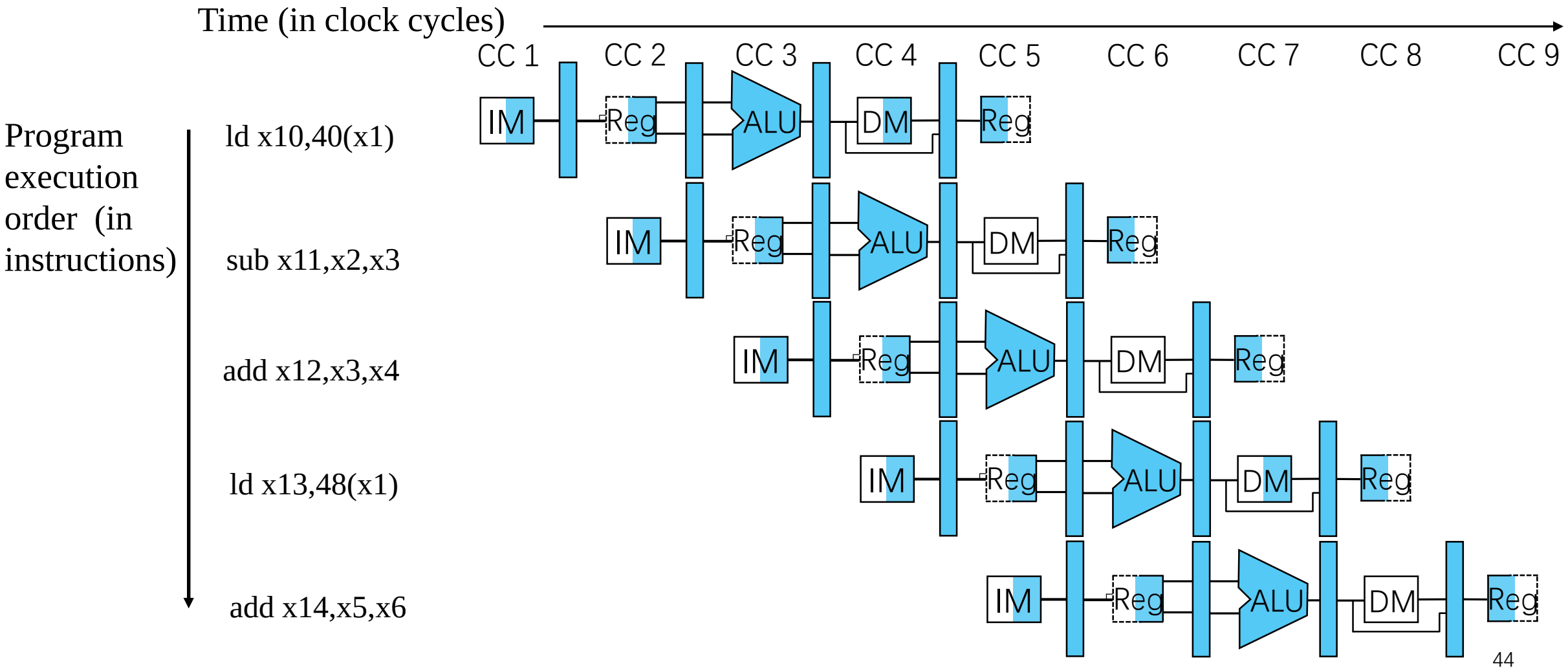
存储指令在“写回阶段”不执行任何操作。

sd
Write-back



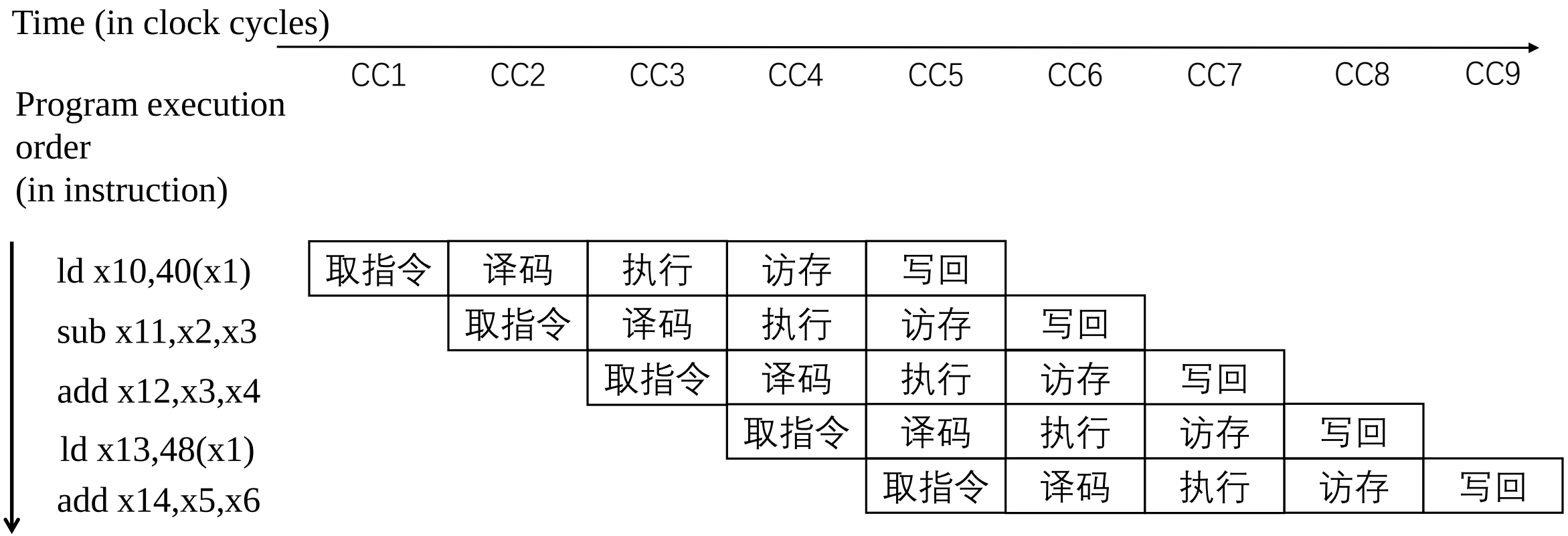
多时钟周期流水线图

- 显示了每个流水线阶段中使用的物理资源



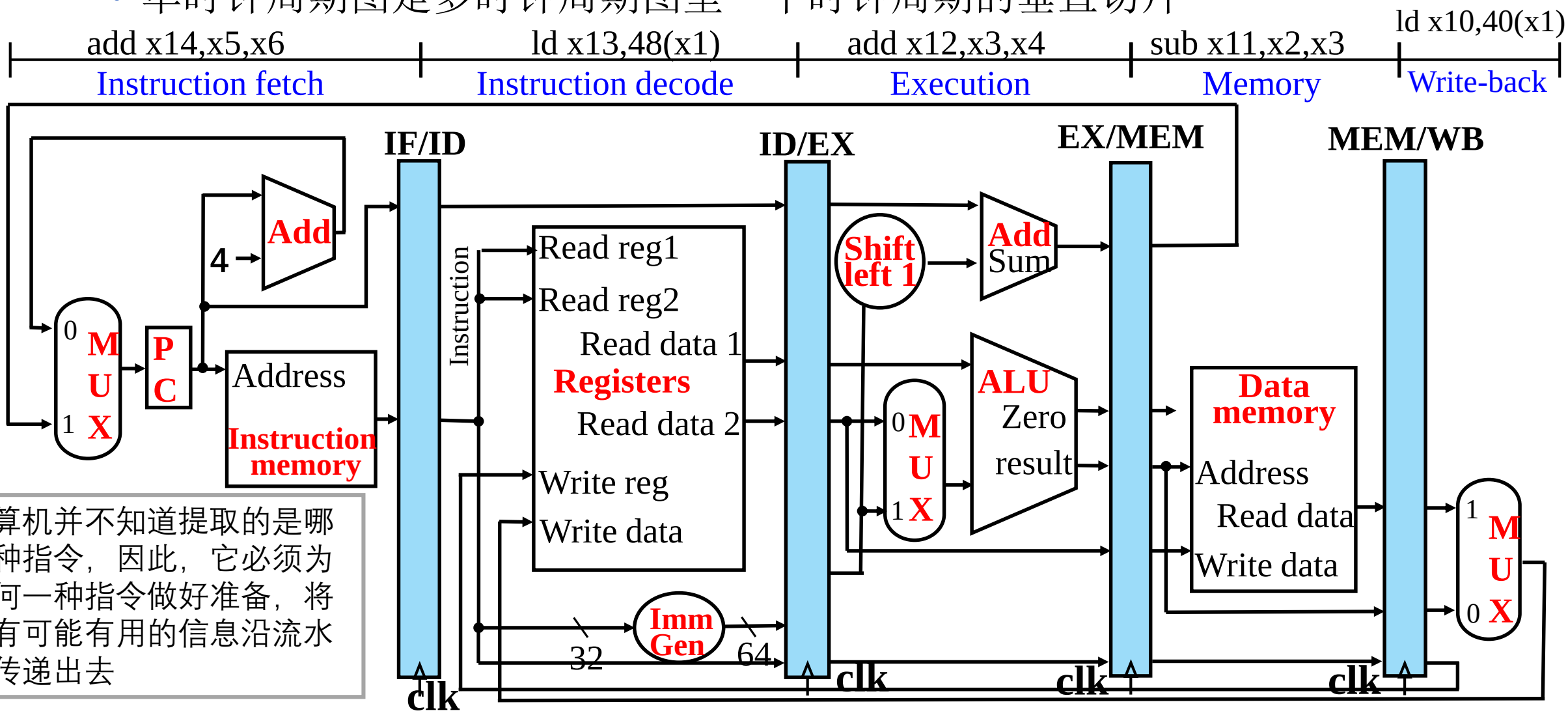
多时钟周期流水线图

用矩形块来命名每个流水线阶段



一个时钟周期下的流水线图

- 显示了给定周期中流水线的状态
  - 单时钟周期图是多时钟周期图里一个时钟周期的垂直切片



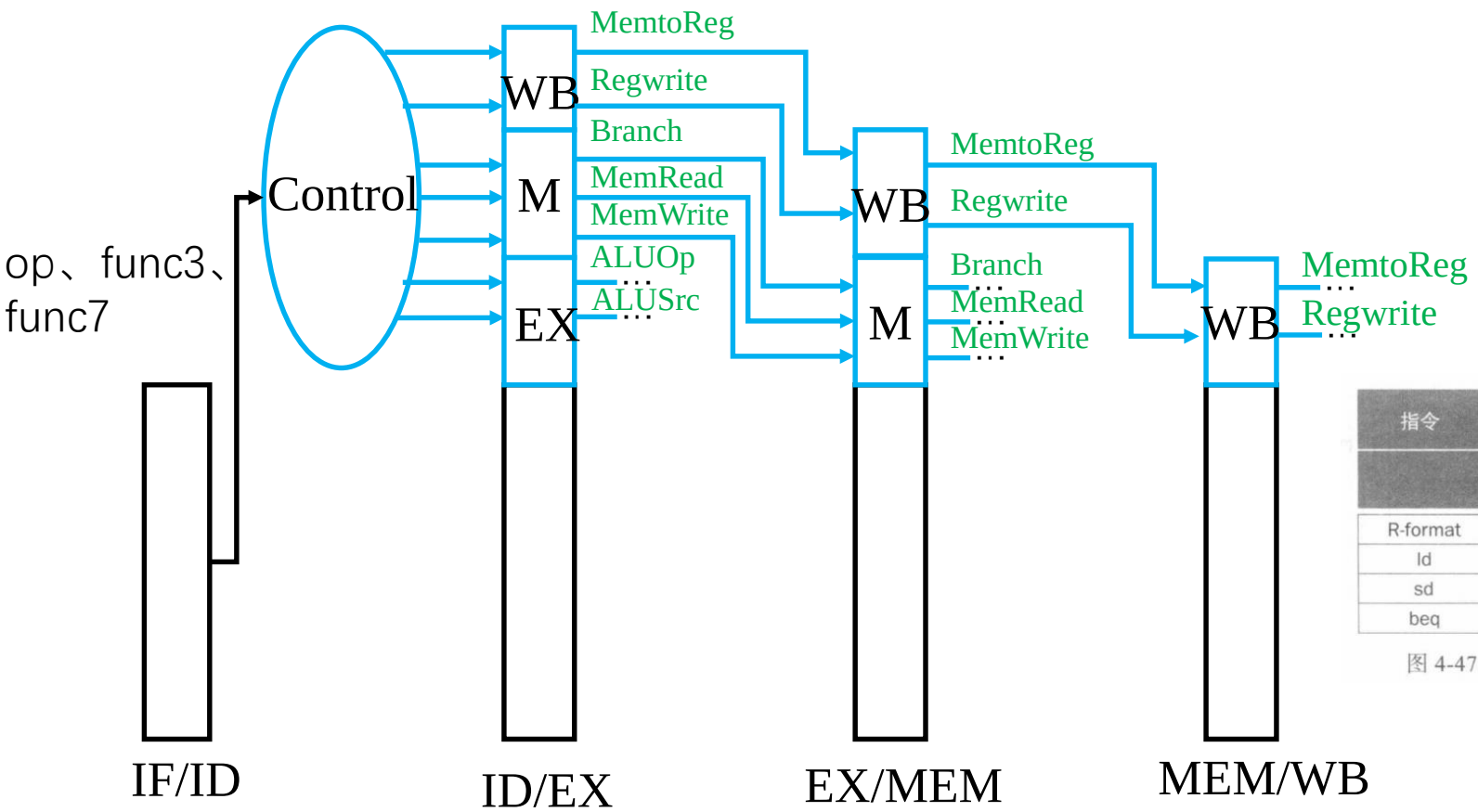
计算机并不知道提取的是哪一种指令，因此，它必须为任何一种指令做好准备，将所有可能有用的信息沿流水线传递出去

第五章 流水线处理器

- **流水线概述**
- 流水线数据通路和控制
- 数据冒险：前递与停顿
- 控制冒险
- 例外

流水线控制

- 控制信号从指令中产生，与单周期实现相似
- 根据流水线阶段将控制线也划分成5组
  - IF和ID阶段没有需要特别控制的操作



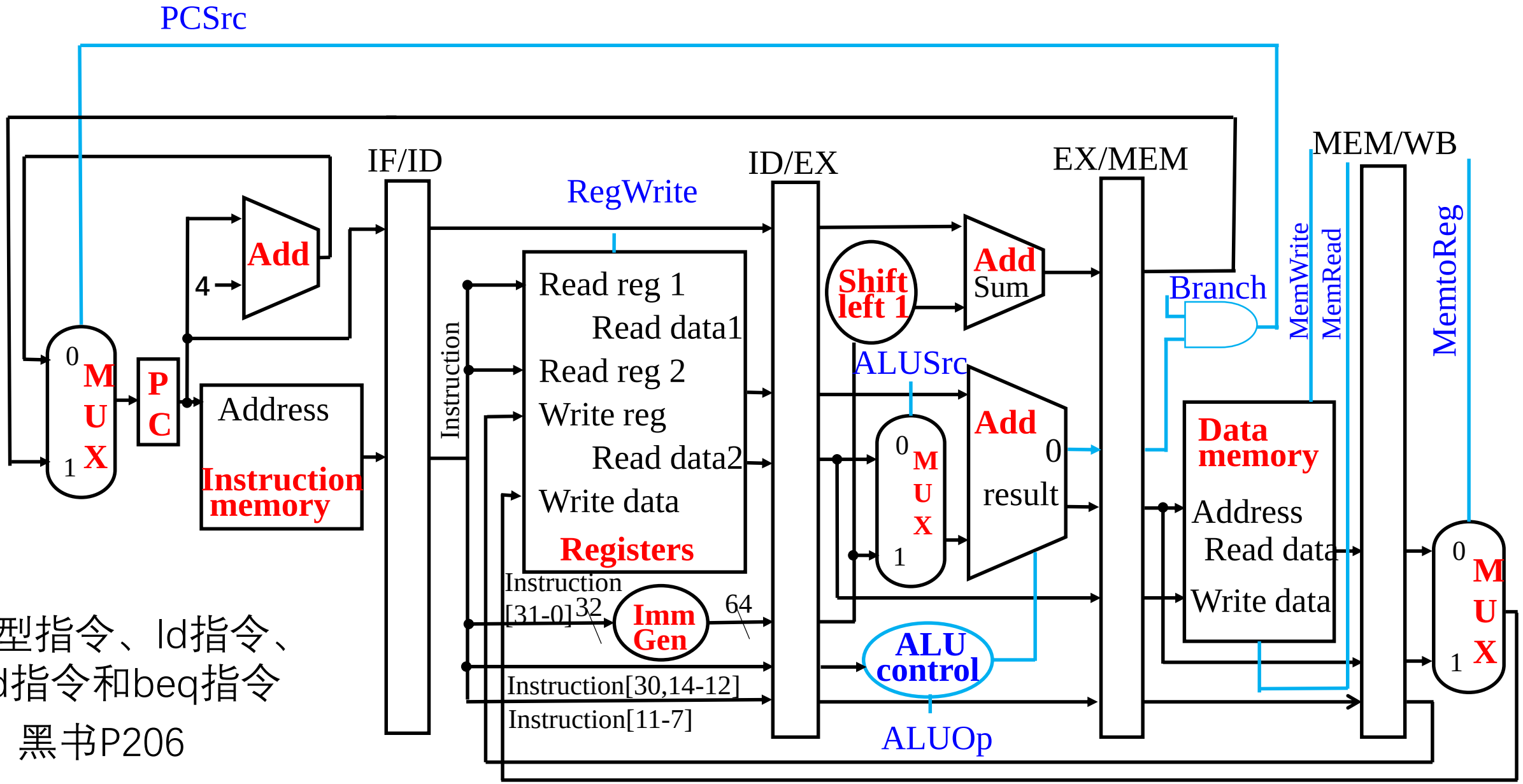
- MemtoReg选择ALU或数据存储器值写入到寄存器堆
- ALUSrc：ALU前的选择器控制信号，用于选择Reg[rs2]或立即数值
- Branch：分支指令的控制信号

指令	执行/地址计算阶段控制线		存储器访问阶段控制线			写回阶段控制线	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

图 4-47　根据流水线的最后三个阶段划分成三组的控制线，其值与图 4-18 中相同

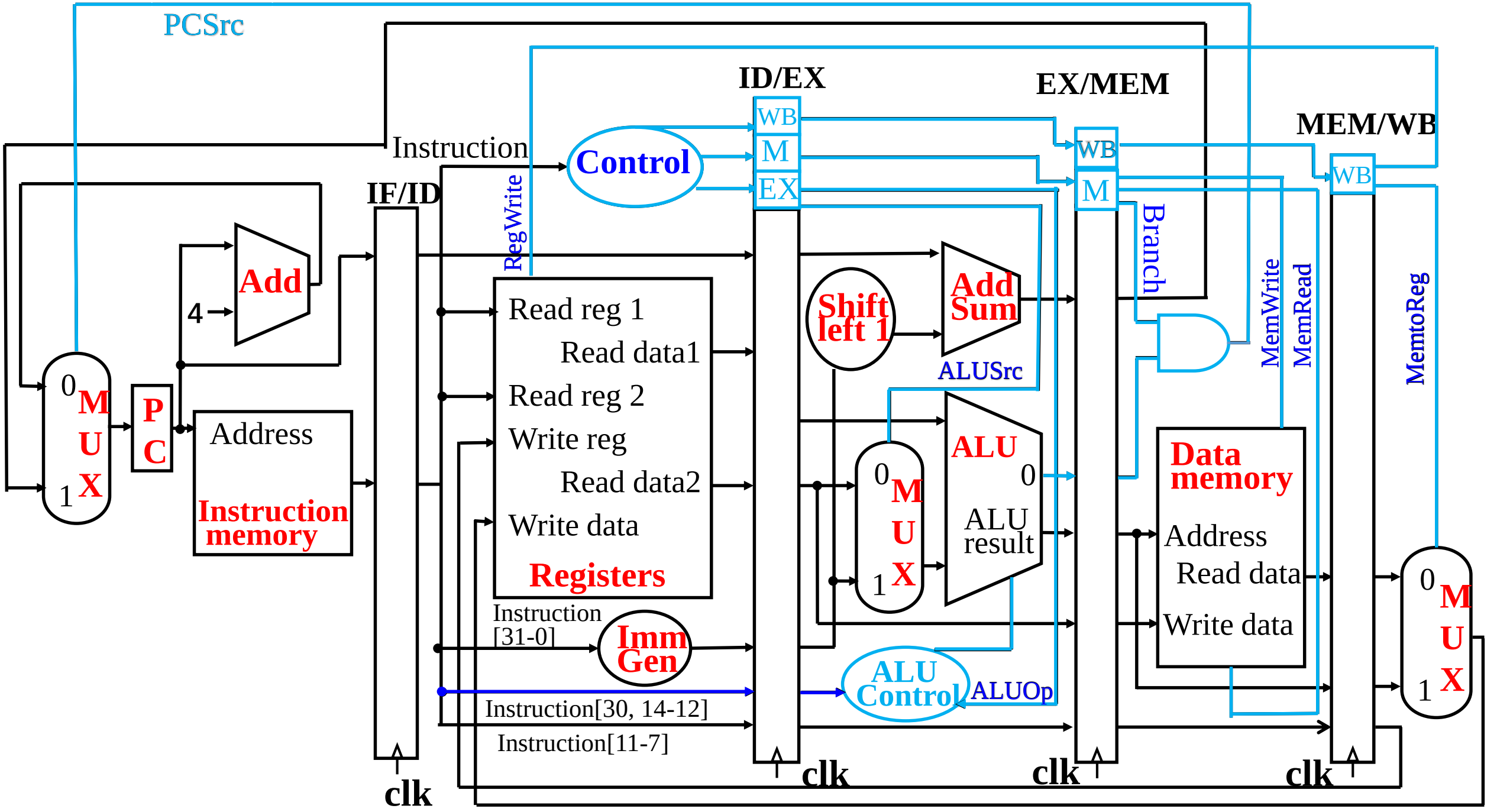
见黑书P207页

简单的流水线控制



R型指令、ld指令、
sd指令和beq指令

黑书P206


PCSrc
ID/EX
EX/MEM
MEM/WB
Instruction
IF/ID
Control
WB
M
EX
RegWrite
Add
4
0
M
U
X
1
P
C
Address
Instruction memory
Read reg 1
Read data1
Read reg 2
Write reg
Read data2
Write data
Registers
Instruction [31-0]
Imm Gen
Instruction[30, 14-12]
Instruction[11-7]
clk
Shift left 1
Add Sum
ALUSrc
ALU
0
ALU result
ALU Control
ALUOp
Branch
MemWrite
MemRead
MemtoReg
Data memory
Address
Read data
Write data

第五章

- 流水线概述
- 流水线数据通路和控制
- **数据冒险：前递与停顿**
- 控制冒险
- 例外

ALU相关指令中的数据冒险

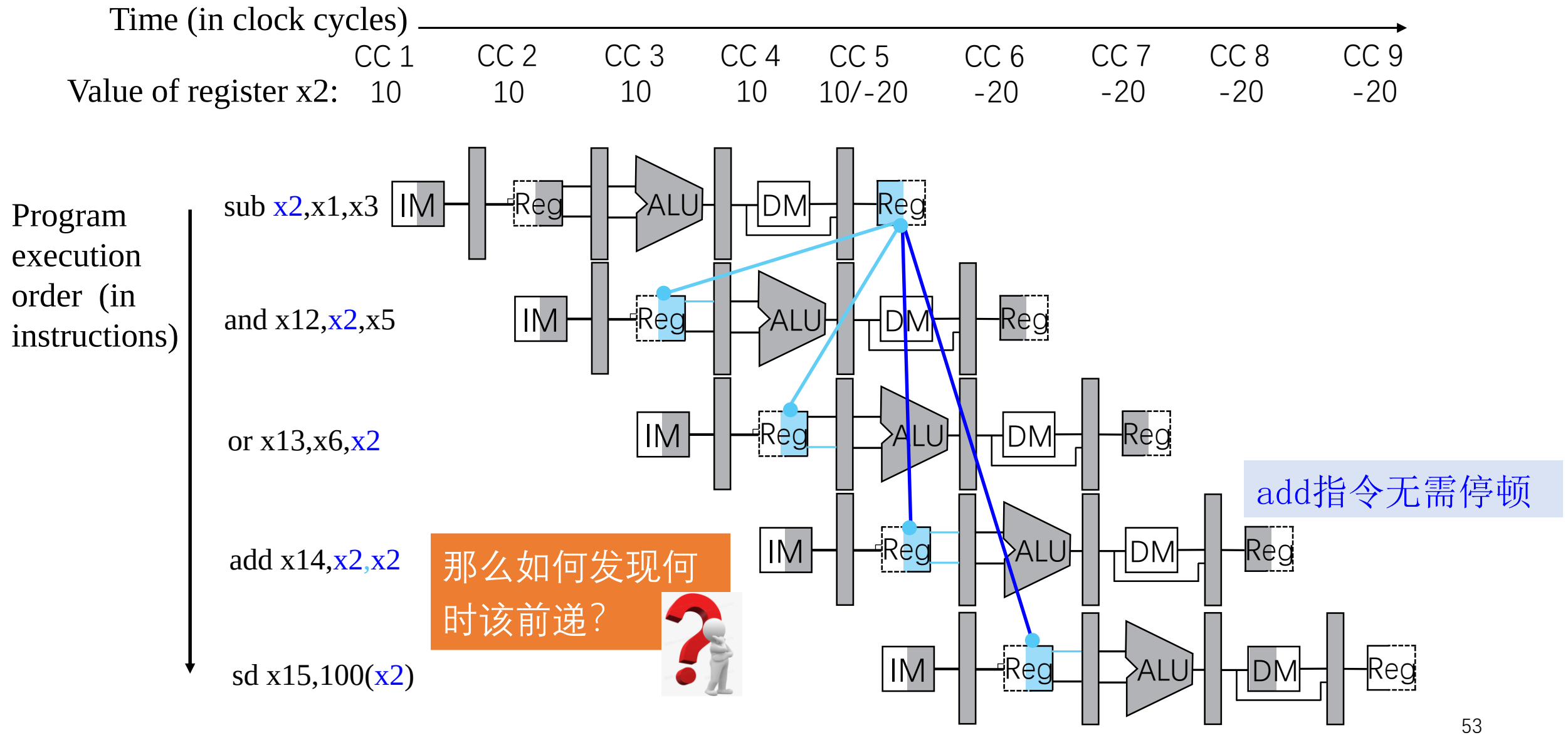
- 考虑以下指令序列:

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

x2: 10 (before)
x2: -20 (after)

- 通过前递来解决这些冒险
  - 那么如何发现何时该前递？

依赖关系与前递

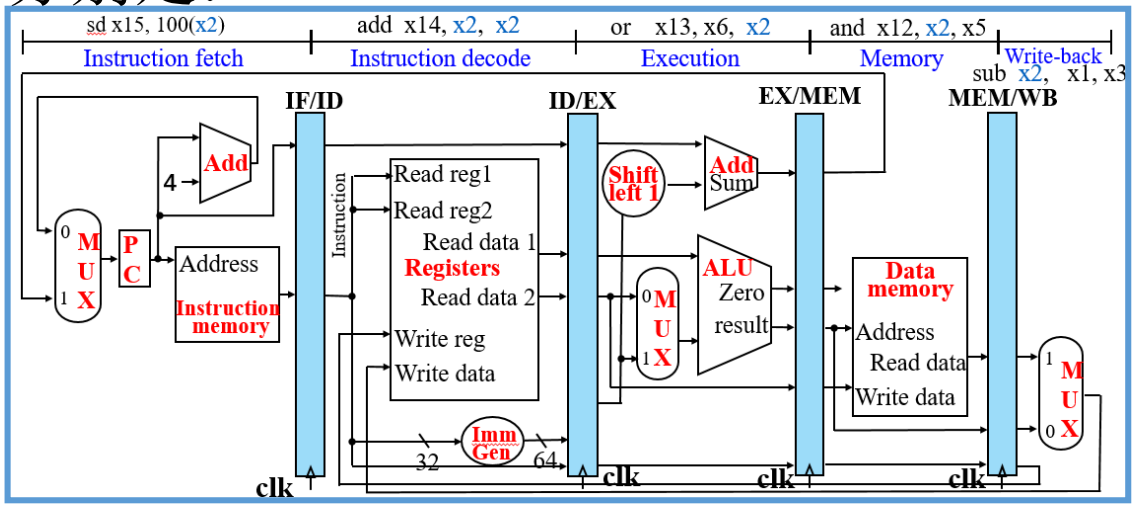

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Value of register x2:
10
10
10
10
10/-20
-20
-20
-20
-20
Program execution order (in instructions)
sub x2,x1,x3
and x12,x2,x5
or x13,x6,x2
add x14,x2,x2
sd x15,100(x2)
IM
Reg
ALU
DM
Reg
那么如何发现何时该前递?
add指令无需停顿

检测前递发生

- 命名流水线寄存器字段
  - 例：ID/EX.RegisterRs1 代表 ID/EX流水线寄存器中，存放寄存器字段的编号Rs1
- EX阶段中，ALU操作数寄存器字段名称分别是：
  - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- 两组数据冒险：EX冒险和MEM冒险

EX冒险
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

MEM冒险
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2



sub x2,x1,x3	取指令	译码	执行	访存	写回	
and x12,x2,x5		取指令	译码	执行	访存	写回

sub x2,x1,x3	取指令	译码	执行	访存	写回	
and x12,x2,x5		取指令	译码	执行	访存	写回
or x13,x6,x2			取指令	译码	执行	访存	写回

检测前递发生

EX/MEM. RegisterRd=8
ID/EX.RegisterRs1 = 8

- 并不是所有指令都会写回寄存器，在前递的时候还应：
  - 检查RegWrite信号是否有效
    - EX冒险：EX/MEM.RegWrite ≠ 0
    - MEM冒险：MEM/WB.RegWrite ≠ 0

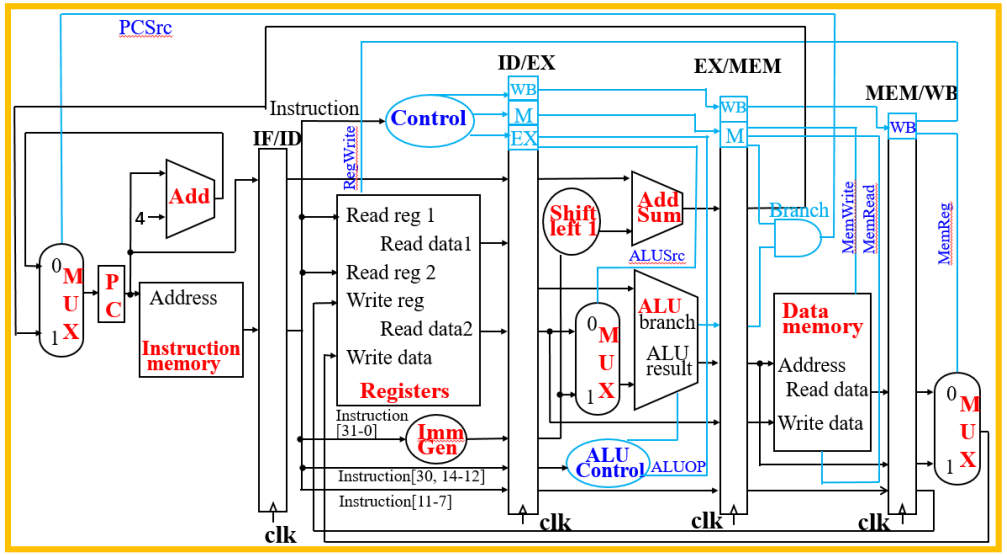
R型	func7	rs2	rs1	func3	rd	opcode
S型	imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode

```
sd   x1, 8(x2)
add  x1, x8, x2
```

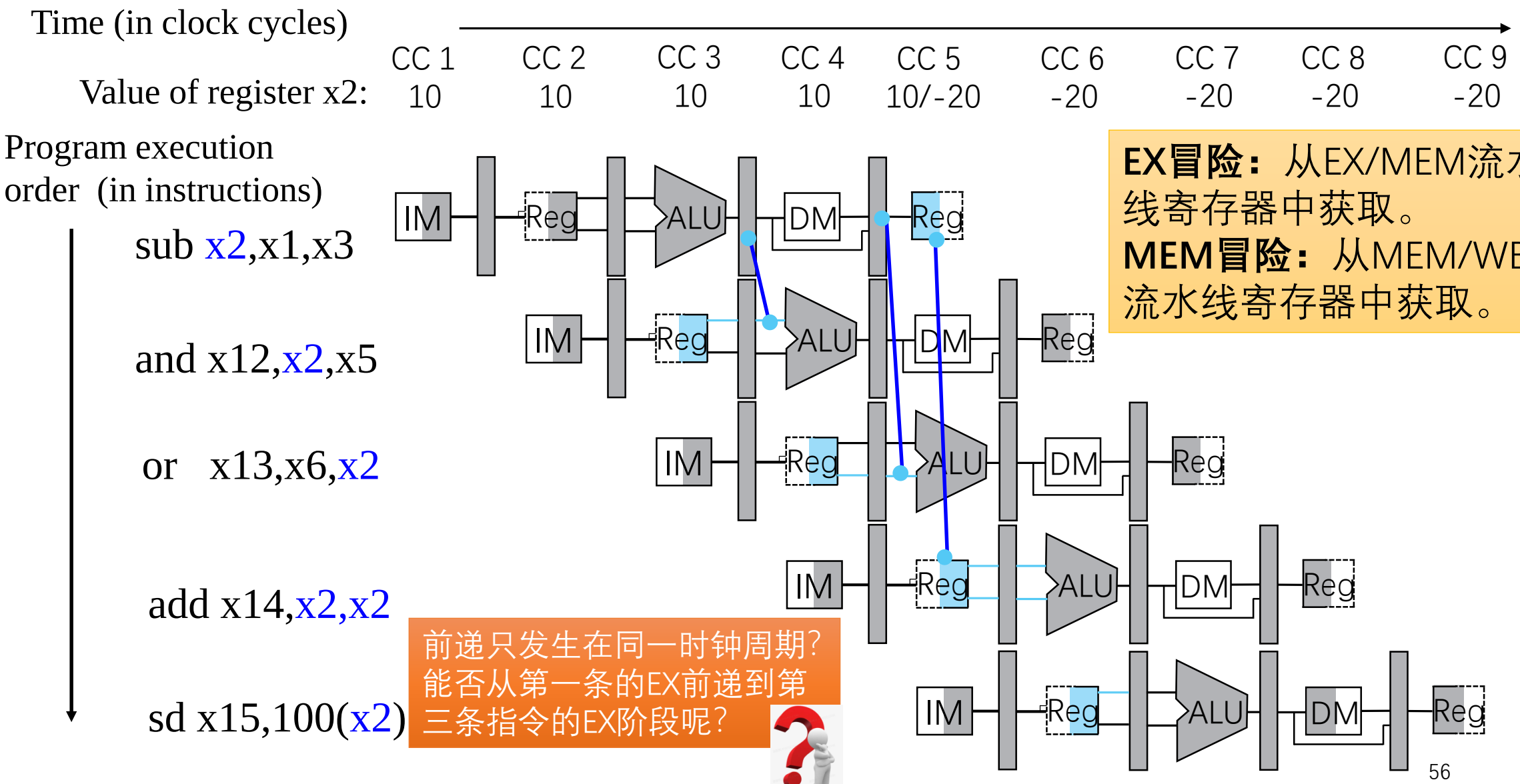
- 冒险的检测条件仍然不完善
  - Rd=x0？
    - EX冒险：EX/MEM. RegisterRd ≠ 0
    - MEM冒险：MEM/WB. RegisterRd ≠ 0
  - 二次冒险

```
sub  x0,  x1, x3
and  x12, x0, x5
```

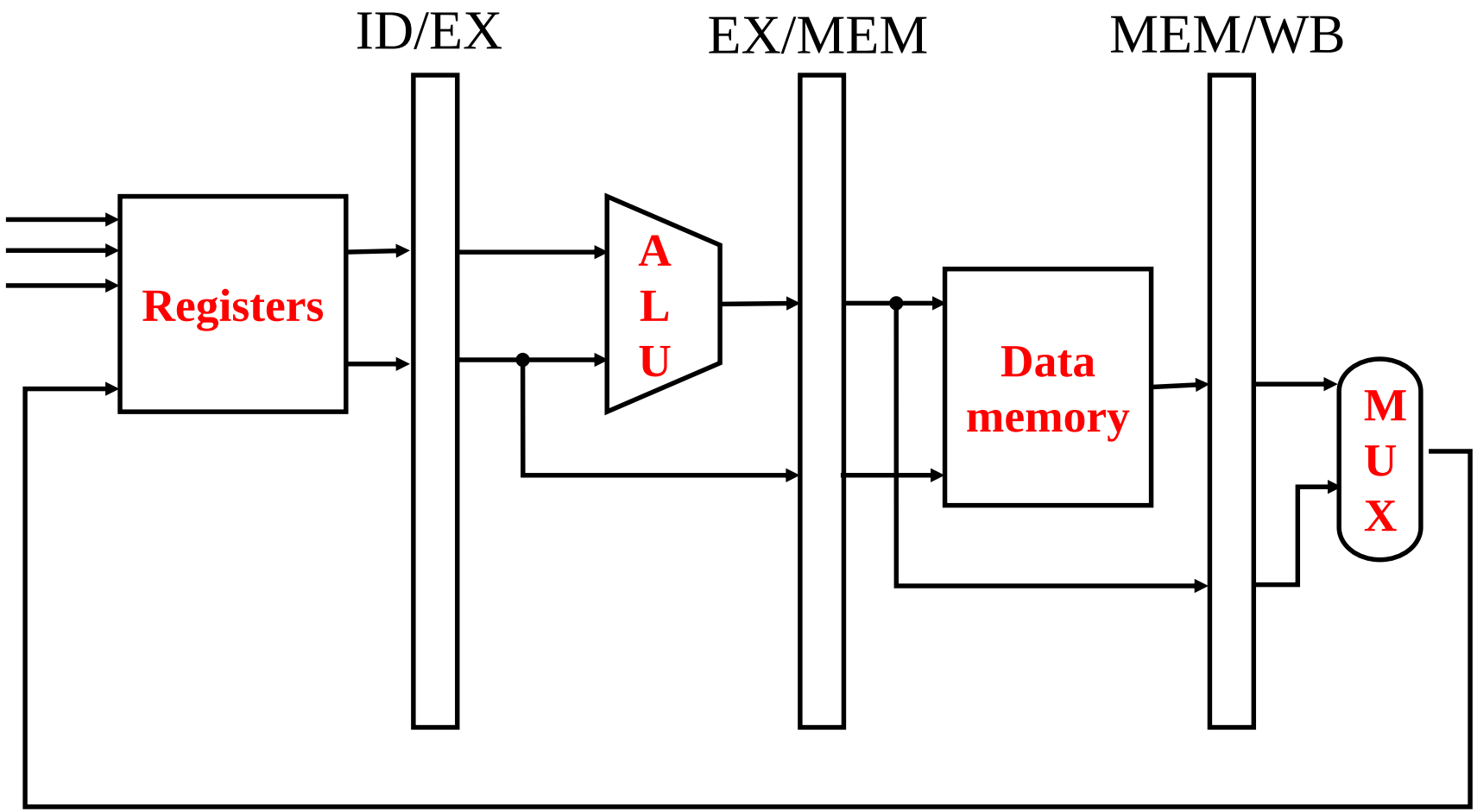
```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```



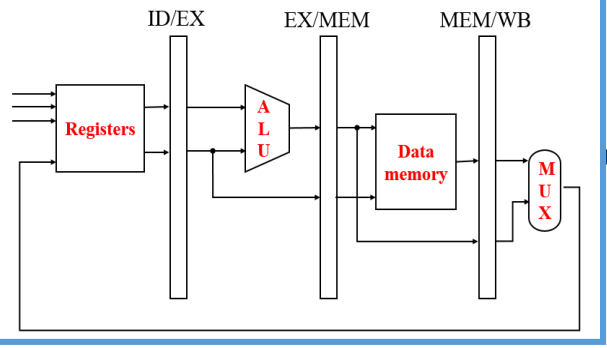
依赖关系的变化


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 x2:
10
10
10
10
10/-20
-20
-20
-20
-20
Program execution order (in instructions)
sub x2,x1,x3
and x12,x2,x5
or x13,x6,x2
add x14,x2,x2
sd x15,100(x2)
IM
Reg
ALU
DM
Reg
EX冒险：从EX/MEM流水线寄存器中获取。
MEM冒险：从MEM/WB流水线寄存器中获取。
前递只发生在同一时钟周期？能否从第一条的EX前递到第三条指令的EX阶段呢？

添加前递前



添加前递所需硬件




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

and x12, x2, x5

sub x2, x1, x3

ID/EX

EX/MEM

MEM/WB

Registers

MUX

ALU

Data memory

MUX

ForwardA

ForwardB

Rs1

Rs2

Rd

EX/MEM.RegisterRd

MEM/WB.RegisterRd

Forwarding unit

EX冒险

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

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

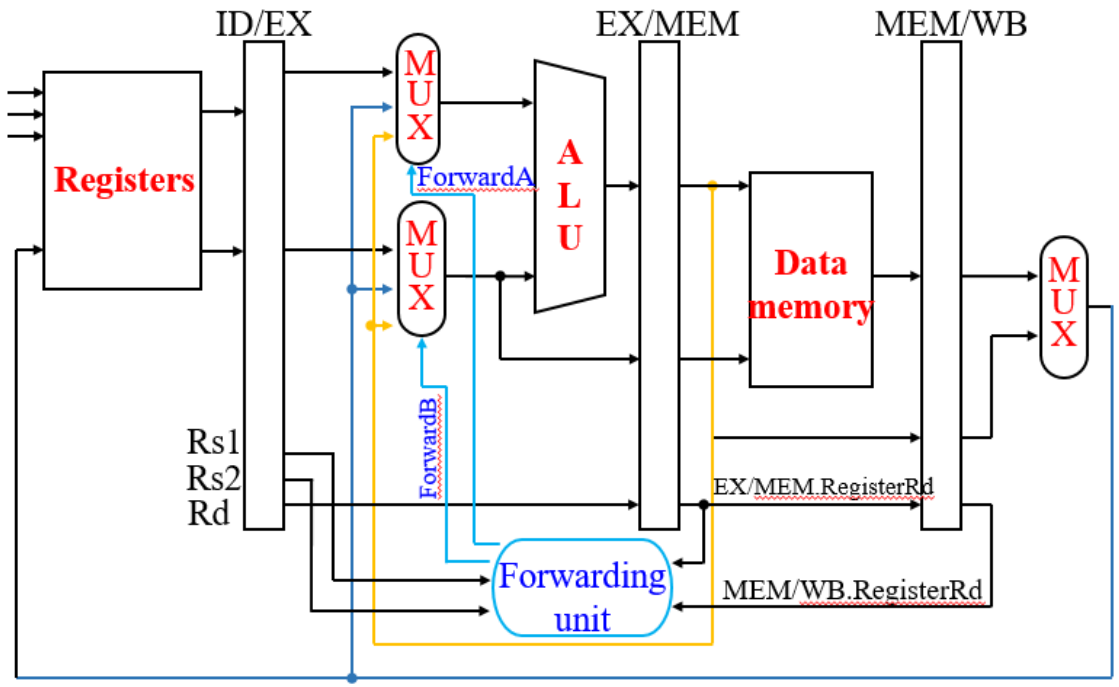
MEM冒险

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

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

加入前递之后，Rs1和Rs2字段也要添加到ID/EX流水线寄存器中

前递硬件中，多选器的控制值



ForwardB	Source	Explanation
00	ID/EX	ALU的第二个操作数来自寄存器堆
10	EX/MEM	ALU第二个操作数来自上一个ALU计算结果的前递
01	MEM/WB	ALU的第二个操作数来自数据存储器或者更早的ALU计算结果的前递

ForwardA	Source	Explanation
00	ID/EX	ALU的第一个操作数来自寄存器堆
10	EX/MEM	ALU的第一个操作数来自上一个ALU计算结果的前递
01	MEM/WB	ALU的第一个操作数来自数据存储器或者更早的ALU计算结果的前递

检测EX冒险的条件、相应的前递控制

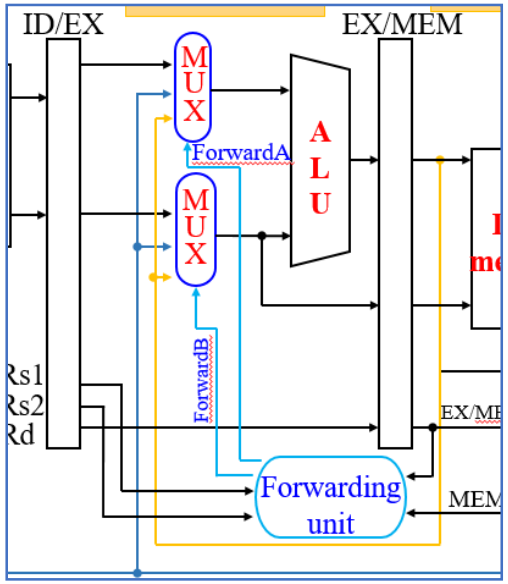
- EX 冒险
  - if (EX/MEM.RegWrite **and** (EX/MEM.RegisterRd ≠ 0)

    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))

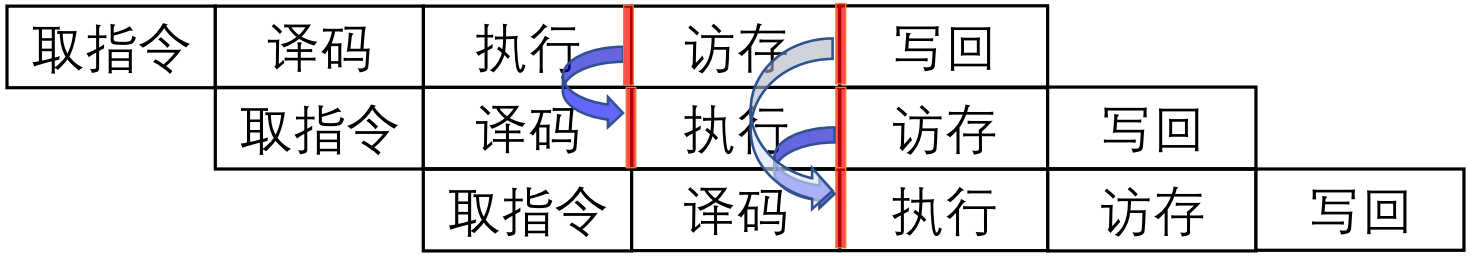
    ForwardB = 10



MEM冒险检测仍有问题：两次数据冒险

- 考虑以下指令序列:

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



- 两组数据冒险都发生
  - 对于第三条指令而言，应该使用最近的结果

- MEM冒险前递的条件：只有EX冒险不发生的时候才前递

检测MEM冒险的条件、相应的前递控制

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)) //无EX冒险

and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))

ForwardA = 01

if (MEM/WB.RegWrite **and** (MEM/WB.RegisterRd ≠ 0) **and**

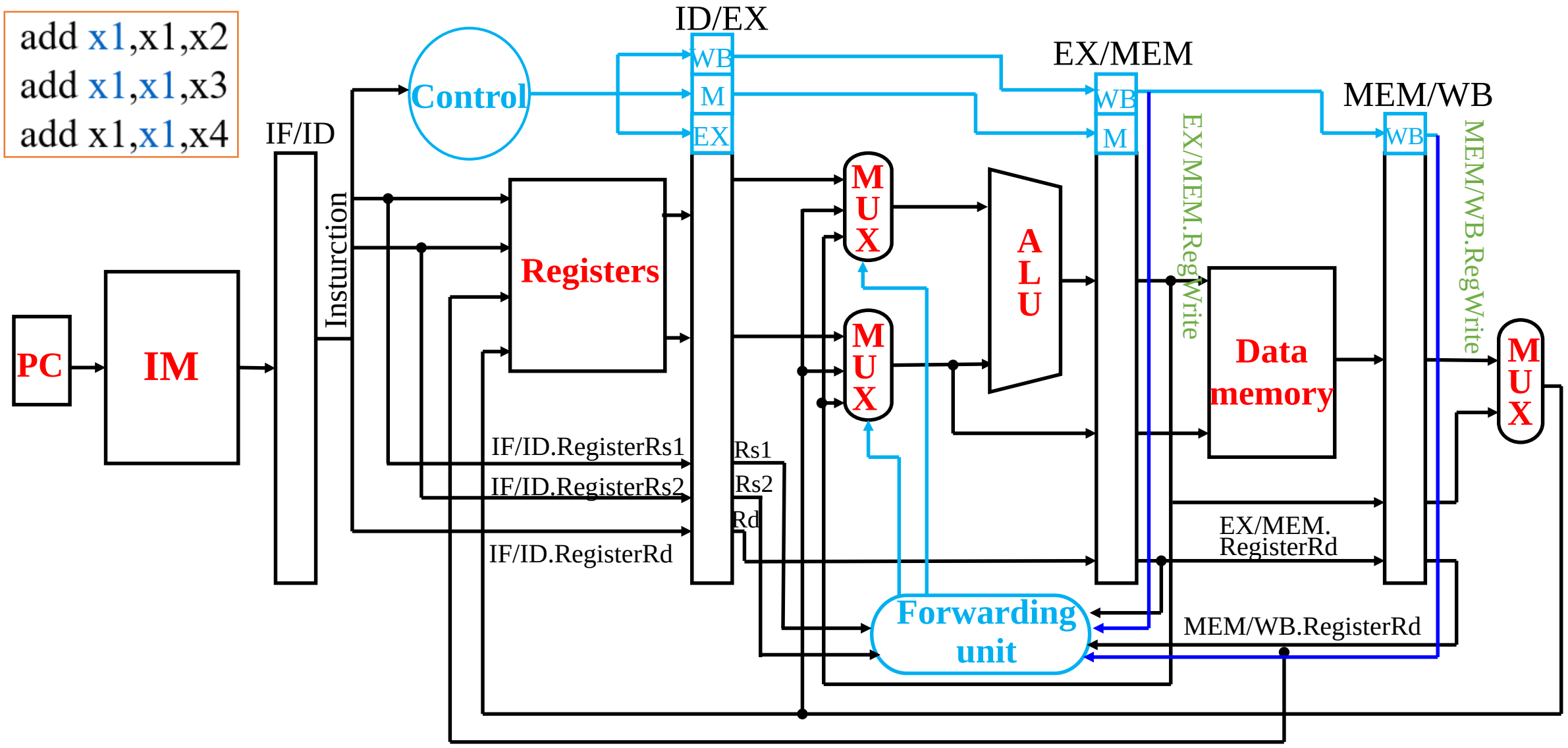
not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) **and**

(EX/MEM.RegisterRd = ID/EX.RegisterRs2)) //无EX冒险

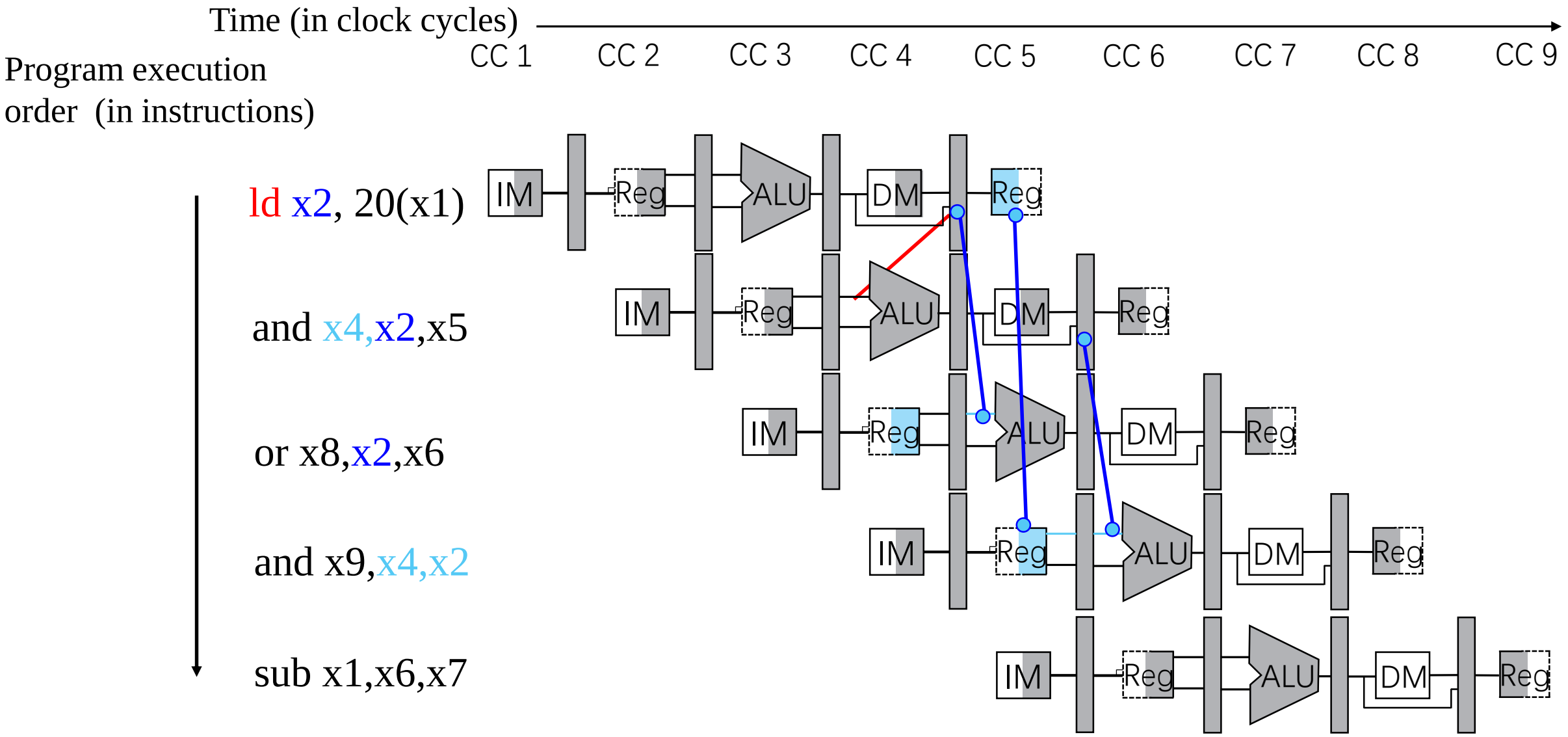
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))

ForwardB = 01

通过前递解决冒险的数据通路



载入-使用(load-use)型数据冒险



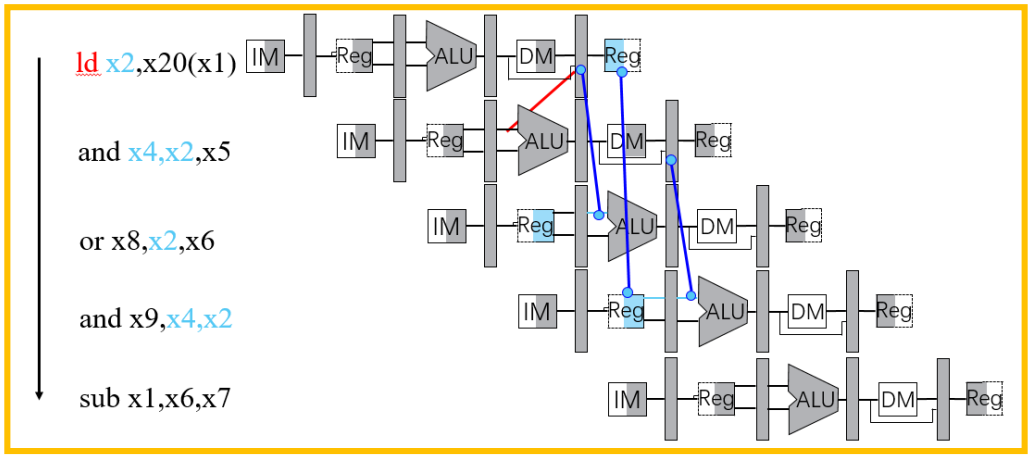
发生于一条数据载入指令(ld)后，紧跟一条需要使用ld所读取的数据的情况

检测 载入-使用型数据冒险

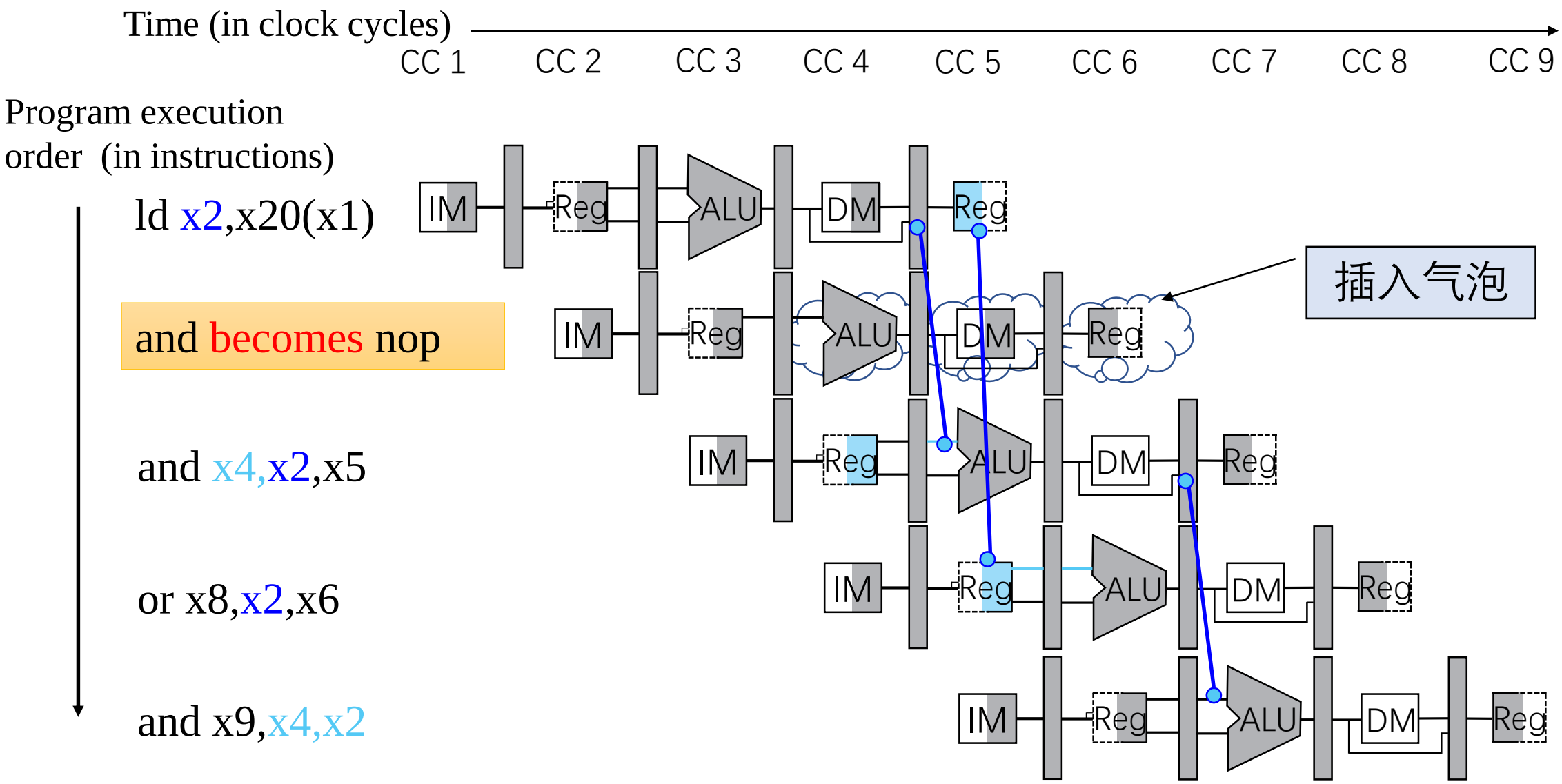
- 在load指令的EX阶段和下一条指令的ID阶段进行检测
- ALU操作数寄存器字段名称分别是：
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- 检测条件
  - ID/EX.MemRead and //是否为load指令？

    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or

    (ID/EX.RegisterRd = IF/ID.RegisterRs2))
- 如果检测到这种冒险，停顿流水线（插入气泡）



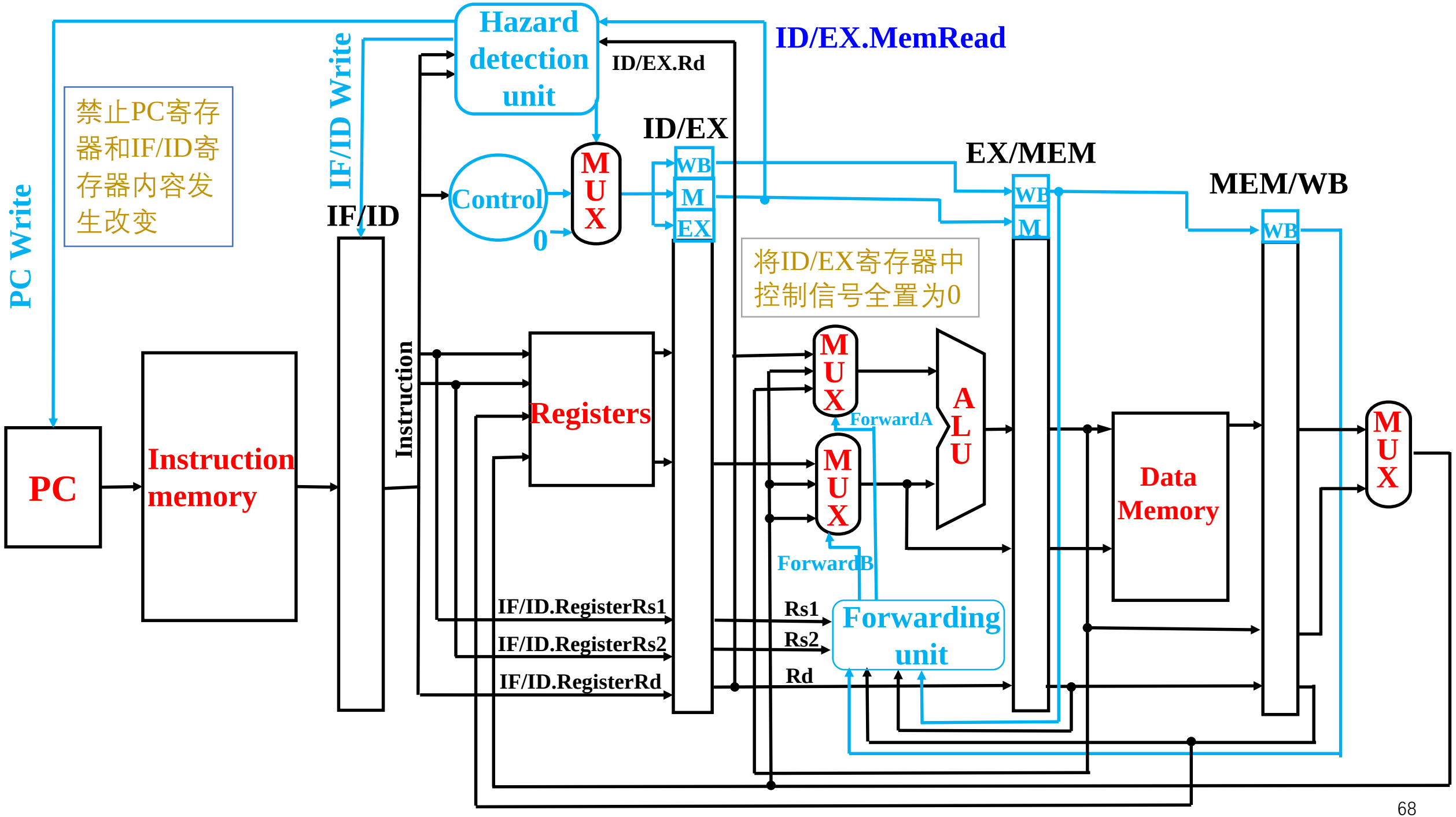
载入–使用型数据冒险


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)
ld x2,x20(x1)
and becomes nop
and x4,x2,x5
or x8,x2,x6
and x9,x4,x2
IM
Reg
ALU
DM
Reg
插入气泡

如何停顿流水线?

被停顿的指令正处于ID阶段:

- 将ID/EX寄存器中控制信号全置为0(RegWrite, MemWrite...)
  - 被停顿的指令在EX、MEM、WB 阶段执行空指令 nop
  - 在控制值均为0时，不会有寄存器或者存储器被写入数据
- 禁止PC寄存器和IF/ID寄存器内容发生改变
  - ID阶段的寄存器会继续使用IF/ID寄存器中相同字段读寄存器
  - 下一条指令会重新取指
  - 1个时钟周期的停顿，能够让ld指令的MEM阶段完成
    - 就可以把取到的数据前递到EX阶段


Hazard detection unit
ID/EX.MemRead
ID/EX.Rd
IF/ID Write
PC Write
禁止PC寄存器和IF/ID寄存器内容发生改变
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
0
MUX
WB
M
EX
将ID/EX寄存器中控制信号全置为0
Instruction
PC
Instruction memory
Registers
ALU
ForwardA
ForwardB
Data Memory
IF/ID.RegisterRs1
IF/ID.RegisterRs2
IF/ID.RegisterRd
Rs1
Rs2
Rd
Forwarding unit

停顿与性能

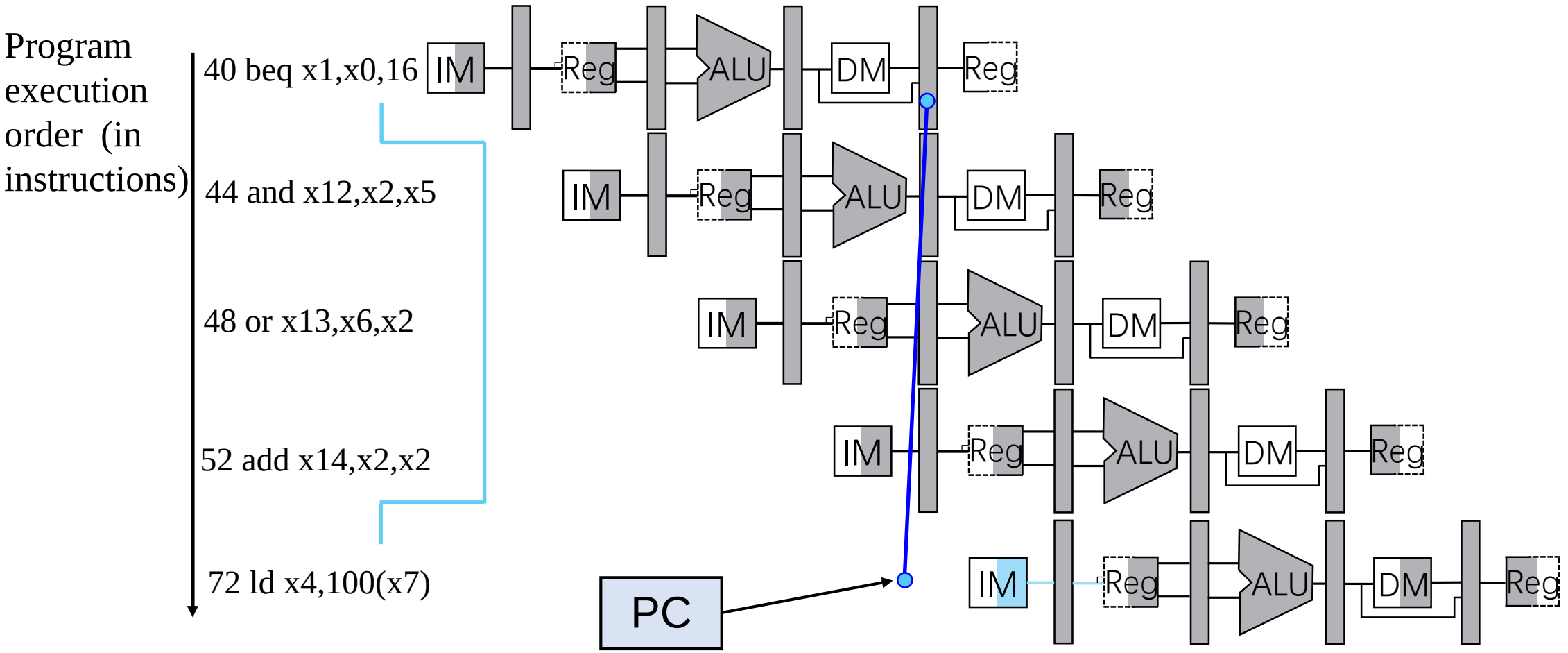
- 停顿会导致性能下降
  - 但为了得到正确的结果，需要停顿
- 编译器能编排代码，尽量避免冒险和停顿
  - 这需要流水线结构的知识

第五章

- 流水线概述
- 流水线数据通路和控制
- 数据冒险：前递与停顿
- **控制冒险**
- 例外

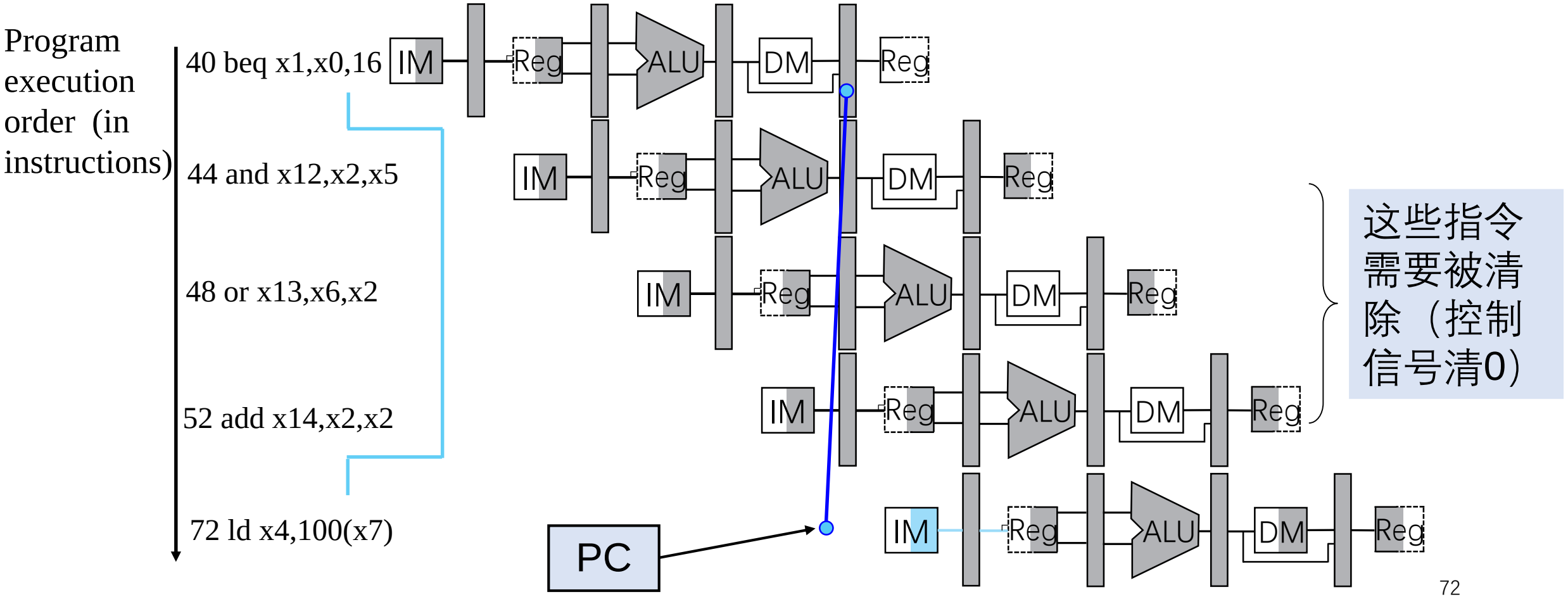
控制冒险

- 假设分支结果在MEM阶段完成确认
  - 分支指令修改PC值发生在MEM阶段



假设分支不发生

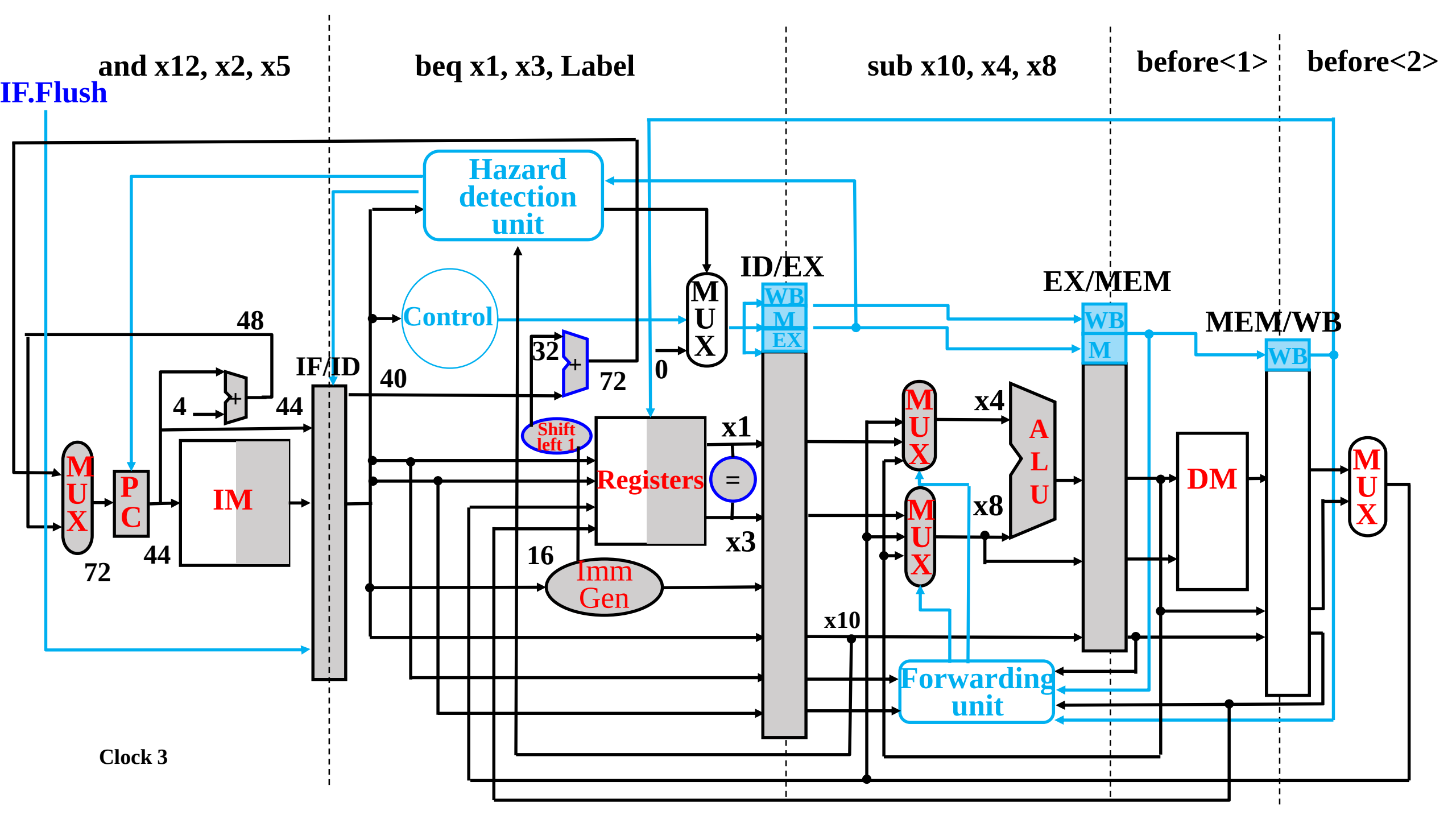
- 默认不发生分支跳转
- 如果发生跳转，则清除掉后续的三条指令

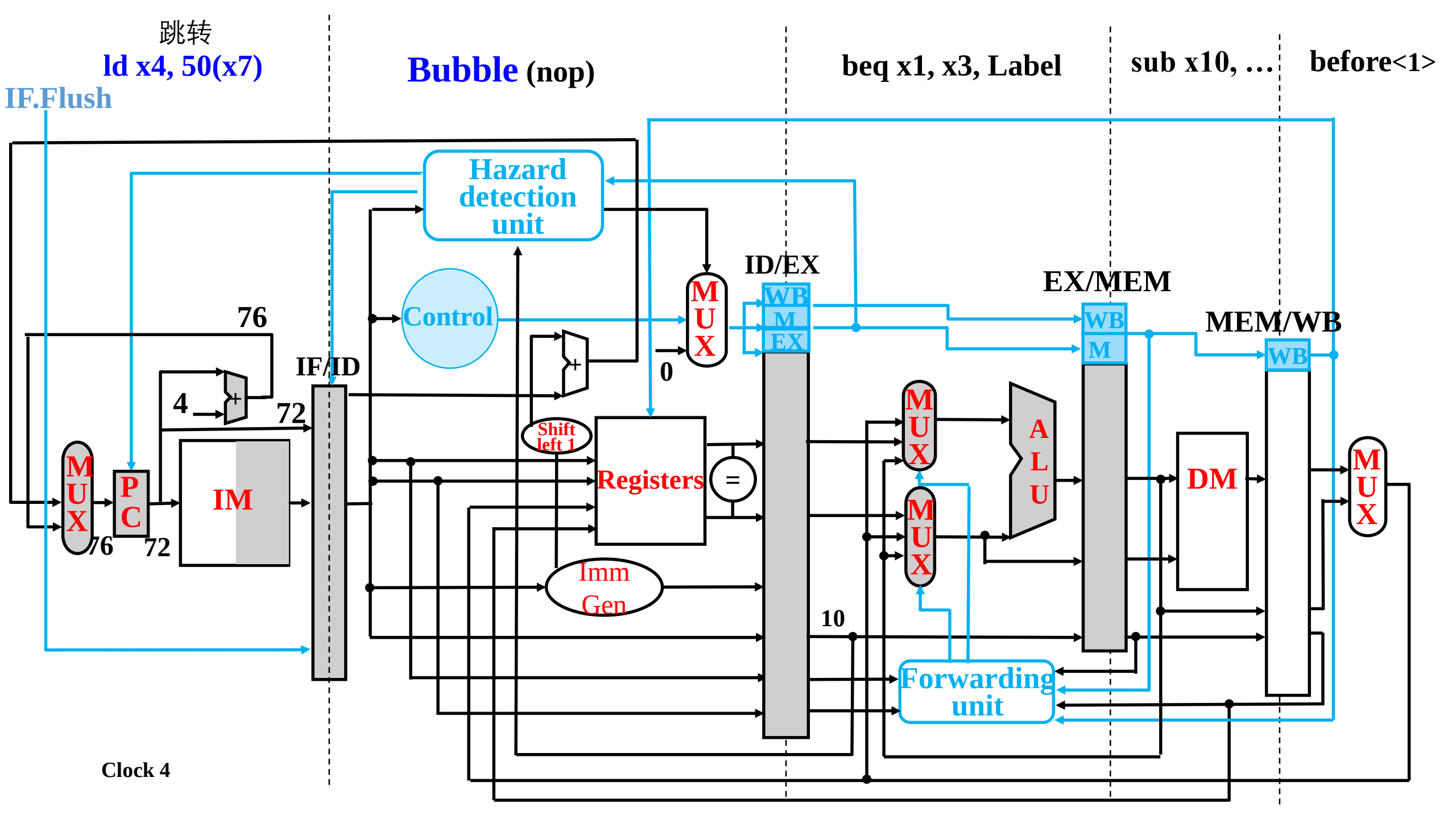


缩短分支延迟

- 移动硬件，使得分支决定提前到ID阶段
  - 需要提早计算分支目标地址（IF/ID寄存器中有PC和Immediate）
  - 需要提早判断分支条件
- 例：分支跳转发生

```
36:  sub  x10, x4, x8
40:  beq  x1,  x3, Label  //假设Label对应的偏移立即数字段为16
44:  and  x12, x2, x5                          PC-relative branch to 40+16*2=72
48:  or   x13, x2, x6
52:  add  x14, x4, x2
56:  sub  x15, x6, x7
   ...
72:  ld   x4, 50(x7)
```


and x12, x2, x5
beq x1, x3, Label
sub x10, x4, x8
before<1>
before<2>
IF.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
48
4
44
IF/ID
40
32
72
0
MUX
PC
IM
72
44
Shift left 1
Registers
x1
x3
=
16
Imm Gen
x4
x8
ALU
DM
x10
Forwarding unit
Clock 3

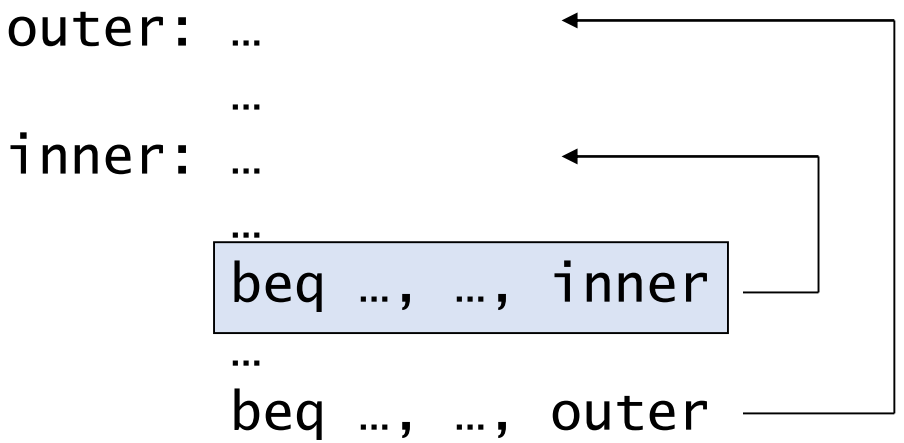

跳转
ld x4, 50(x7)
Bubble (nop)
beq x1, x3, Label
sub x10, …
before<1>
IF.Flush
Hazard detection unit
Control
MUX
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
76
4
72
IF/ID
MUX
PC
IM
76
72
Shift left 1
Registers
=
Imm Gen
MUX
MUX
ALU
DM
MUX
10
Forwarding unit
Clock 4

动态分支预测

- 对于更深的流水线，从时钟周期数的角度来说，分支预测错误的代价会增大。
- 使用动态分支预测
  - 一种实现方式：采用**分支预测缓存**或**分支历史表**
  - 是一块按照分支指令的低位地址索引定位的小容量存储器
  - 包含一个或多个位（bit）以表明一个分支最近是否发生了跳转
- 1-Bit预测机制：用1位表示最近是否发生了跳转
  - 查表：使用表中记录的结果作为预测结果
  - 开始取指令
  - 如果预测错误，则清掉错误指令，并更改分支预测缓存中的记录

1-Bit预测机制的缺点

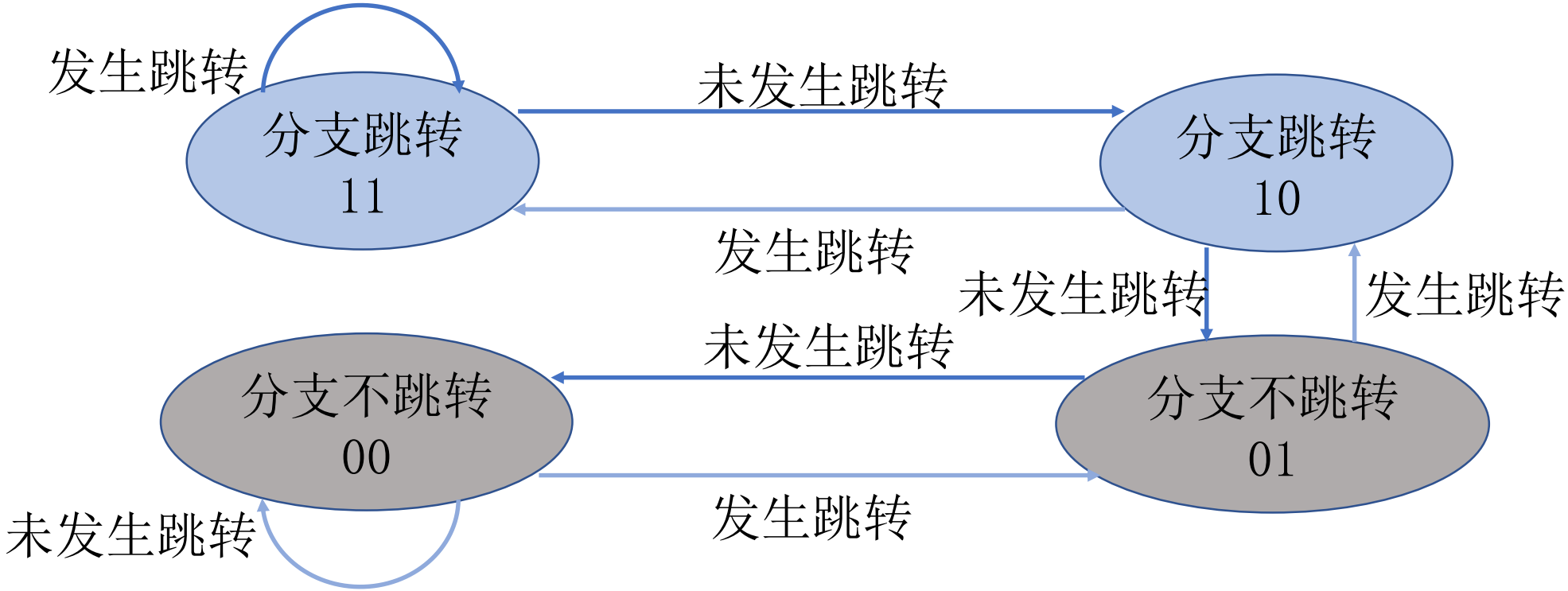
- 如下例：一个条件分支总是发生跳转，但一旦其不发生跳转时，会导致两次预测错误，而不是只造成一次错误



  - 内层循环的最后一次迭代不发生跳转，预测发生跳转，预测错误，更改预测值为不跳转。
  - 下一次外层循环开始后，内层循环第一次迭代发生跳转，预测不发生跳转，预测错误，更改预测值为跳转

2-Bit预测机制

- 对于左侧两个状态，只有在发生了连续两次错误时预测结果才会被改变（跳转加1，不跳转减1）



在一个分支经常跳转或经常不跳转的情况下（大多数分支都是这样的），只会发生一次预测失效

分支目标计算

- 除了判断分支是否发生，还要计算分支目标地址
  - 发生跳转时需要一个时钟周期的代价来计算分支目标地址
- 使用分支目标缓存
  - 缓存分支目标PC值或目标指令
  - 根据当前分支指令的PC值来索引定位
    - 如果缓存中有目标PC值或指令，且分支预测跳转，就能读取缓存结果，立即使用

例外(exception)和中断(interrupt)

- 例外和中断是控制逻辑需要实现的任务之一
  - 除分支指令外，另一种改变指令执行控制流的方式
- 不同的ISA对于例外和中断的定义不同
  - Intel x86使用中断同时指代两者
  - 对于RISC-V
    - **例外** 指 **意外的控制流变化**，而这些变化无须区分产生原因是来自于处理器内部还是外部
    - **中断** 仅指 **由处理器外部事件引发** 的控制流变化
- 对例外进行时序优化是困难的

RISC-V中的例外和中断示例

事件类型	来源	RISC-V中的表示
系统重启	外部	例外
I/O设备请求	外部	中断
用户程序进行操作系统调用	内部	例外
未定义指令	内部	例外
硬件故障	皆可	皆可

RISC-V体系结构中如何处理例外

- 若执行某指令时出现硬件故障：保存发生例外的指令地址，将控制权转交给操作系统
  - 使用**系统例外程序计数器**(Supervisor Exception Program Counter, SEPC)保存发生例外的指令地址，64位寄存器
  - 跳转到统一的入口地址，进行例外处理
    - $0000\ 0000\ 1C09\ 0000_{hex}$
  - 保存例外发生的原因
    - **系统例外原因寄存器**(Supervisor Exception Cause Register, SCAUSE)
    - SCAUSE：64位寄存器，大多数位未被使用。

另一种例外处理方式(x86等)

- 向量式中断（x86中统称为中断）：采用基址寄存器加上例外原因（作为偏移）作为目标地址来完成控制流转换。
  - 基址寄存器中保存了向量式中断内存区域的起始地址
  - 例外原因决定后续控制流的起始地址
- 典型例外的偏移量：
  - 未定义指令的偏移量：　　$00\ 0100\ 0000_2$
  - 硬件故障的偏移量：　　$01\ 1000\ 0000_2$
- 操作系统可根据例外向量起始地址来确定例外原因

例外处理程序的工作

- 如果可以重启程序的执行（I/O设备请求等）
  - 完成例外处理的所有操作
  - 使用**系统例外程序计数器**（ SEPC寄存器）中的内容重启程序的正常执行
- 否则（未定义指令或硬件故障等）
  - 停止当前程序的执行
  - 使用SEPC、SCAUSE（**系统例外原因寄存器**）等来报告错误

流水线实现中的例外

- 流水线实现中，将例外处理看作另一种控制冒险。
- 假设add指令执行阶段发生了硬件故障：

```
40          sub  x11, x2, x4
44          and  x12, x2, x5
48          or   x13, x2, x6
4C          add  x1,  x2, x1
50          sub  x15, x6, x7
54          ld   x16, 100(x7)
…
```

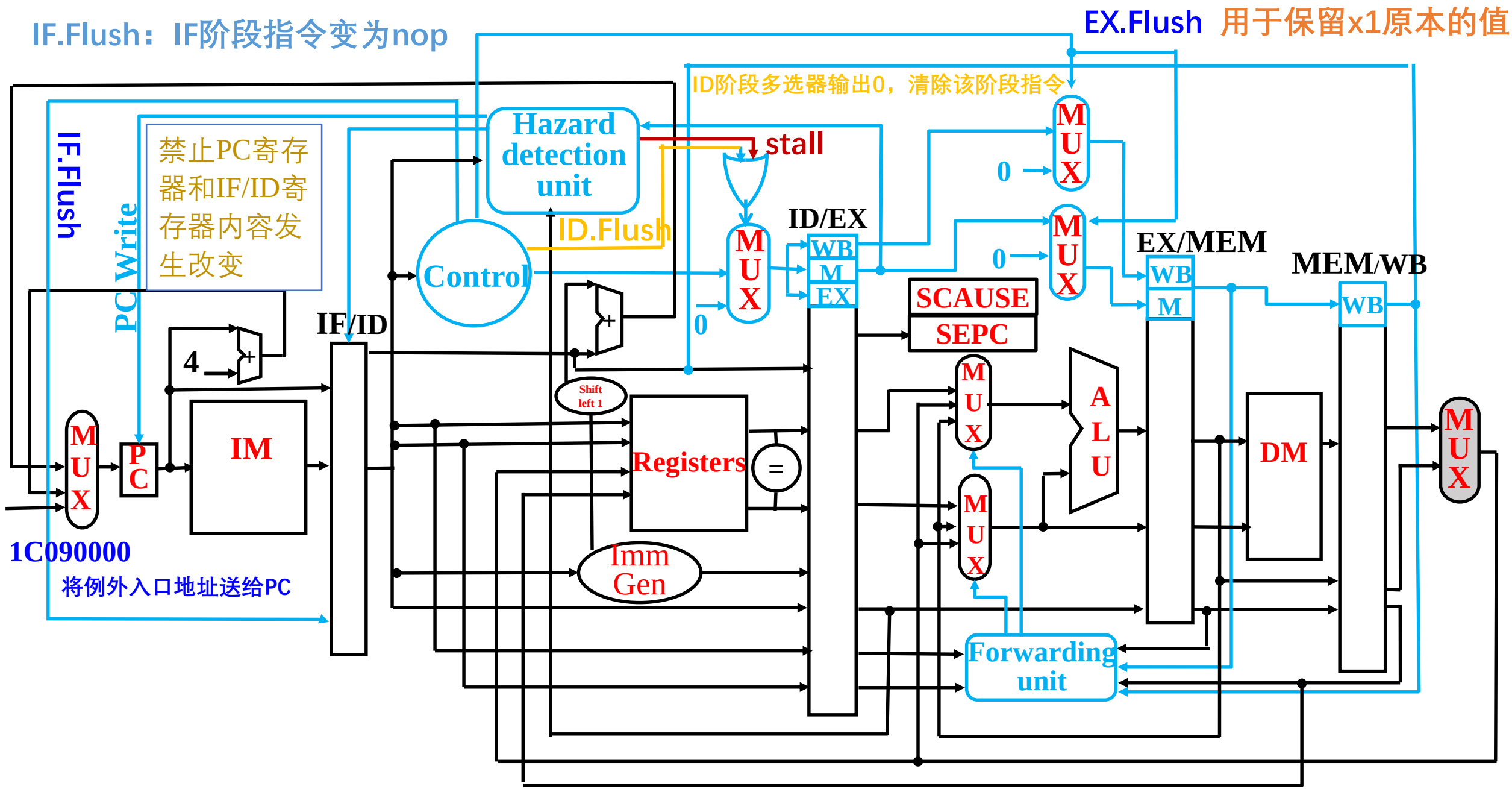
- Handler

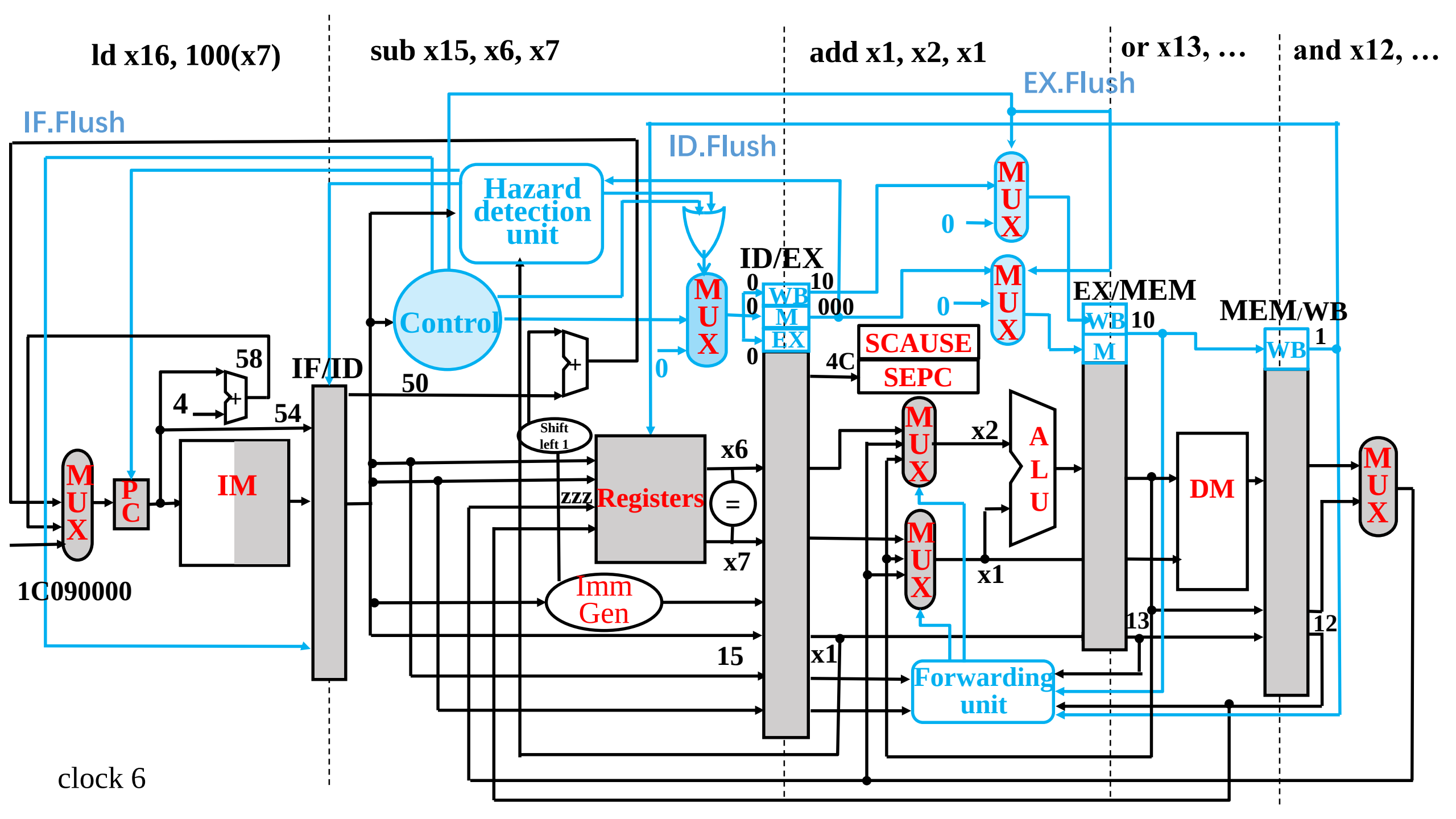
```
1C090000          sd  x26, 1000(x10)
1C090004          sd  x27, 1008(x10)
…
```

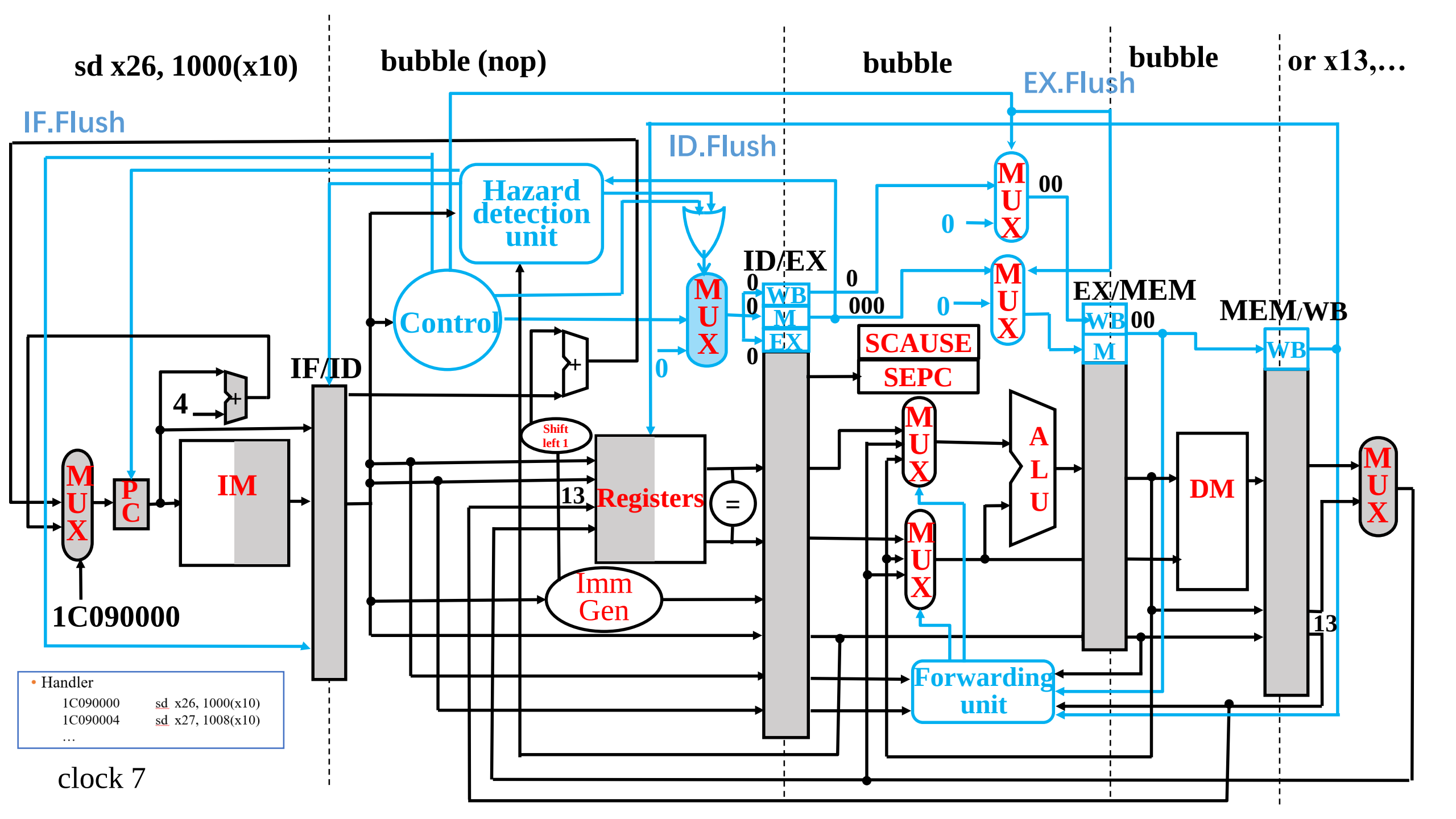
流水线实现中的例外

- 考虑add指令 add x1, x2, x1 的EX阶段发生了硬件故障
  - IF阶段的指令：变为nop操作
  - ID阶段的指令：增加新的逻辑部件，使得译码阶段的输出为0
  - EX阶段的指令：保留目的寄存器x1原本的值，多路选择器控制信号输出为0
  - add之前的指令，正常完成
  - 设置SEPC（**系统例外程序计数器**）和SCAUSE （**系统例外原因寄存器**）的寄存器值
  - 转到例外处理程序

考虑add指令 add x1, x2, x1 的EX阶段发生了硬件故障




ld x16, 100(x7)
sub x15, x6, x7
add x1, x2, x1
or x13, …
and x12, …
IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
SCAUSE
SEPC
4C
IF/ID
58
4
54
50
IM
PC
MUX
1C090000
Shift left 1
Registers
zzz
x6
x7
=
Imm Gen
15
x2
x1
ALU
DM
13
12
Forwarding unit
clock 6


sd x26, 1000(x10)
bubble (nop)
bubble
bubble
or x13,…
IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
MUX
ID/EX
WB
M
EX
SCAUSE
SEPC
EX/MEM
MEM/WB
00
000
IF/ID
4
PC
IM
1C090000
Shift left 1
13
Registers
Imm Gen
=
ALU
DM
Forwarding unit
• Handler
1C090000 sd x26, 1000(x10)
1C090004 sd x27, 1008(x10)
…
clock 7

流水线实现中的例外处理小结

- IF阶段的指令：变为nop操作
- ID阶段的指令：增加新的逻辑部件，使译码阶段输出为0
- EX阶段的指令：需要保留x1原本的值
- 例外之前的指令，正常完成
- 设置SEPC和SCAUSE的寄存器值
- 转到例外处理程序

投票 最多可选6项

设置

流水线CPU部分，对哪里有疑问?

A 流水线数据通路与控制

B 数据冒险

C 控制冒险

D 例外

E 其他请发弹幕

F 无

提交

流水线技术是一种在顺序指令流中，开发指令间并行性的技术。