第七章 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
- 指令间的并行性

单周期处理器的性能问题

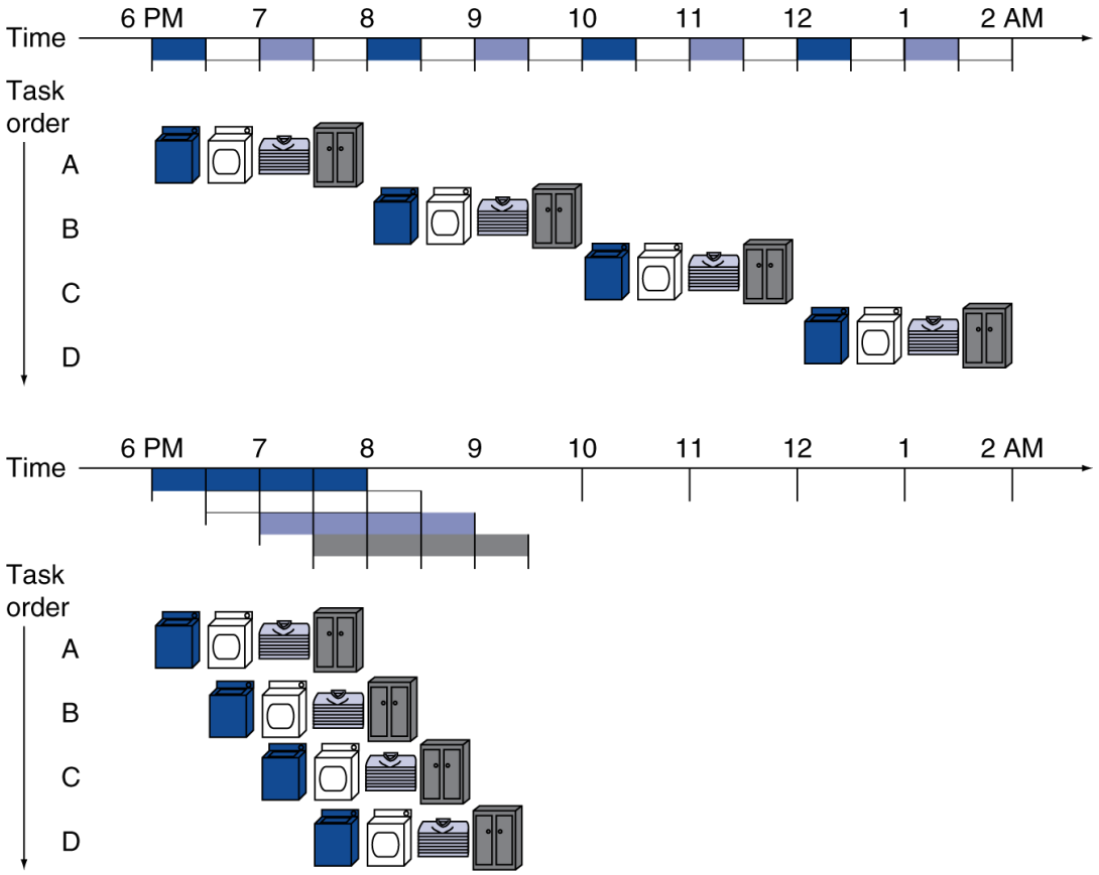
- 单周期设计中，每条指令时钟周期长度必须相等
- 因此，处理器中的最长路径决定了指令时钟周期长度
  - 这条路径很可能是一条load指令
  - 它连续使用了五个功能单元：指令存储器、寄存器堆、ALU、数据存储器、寄存器堆
- 违反了设计思想
  - 无法加速经常性事件：

因为不能缩短常用指令执行时间，从而不能改变最坏情况

解决办法：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



- 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**

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）
- 流水线让这五个阶段重叠执行

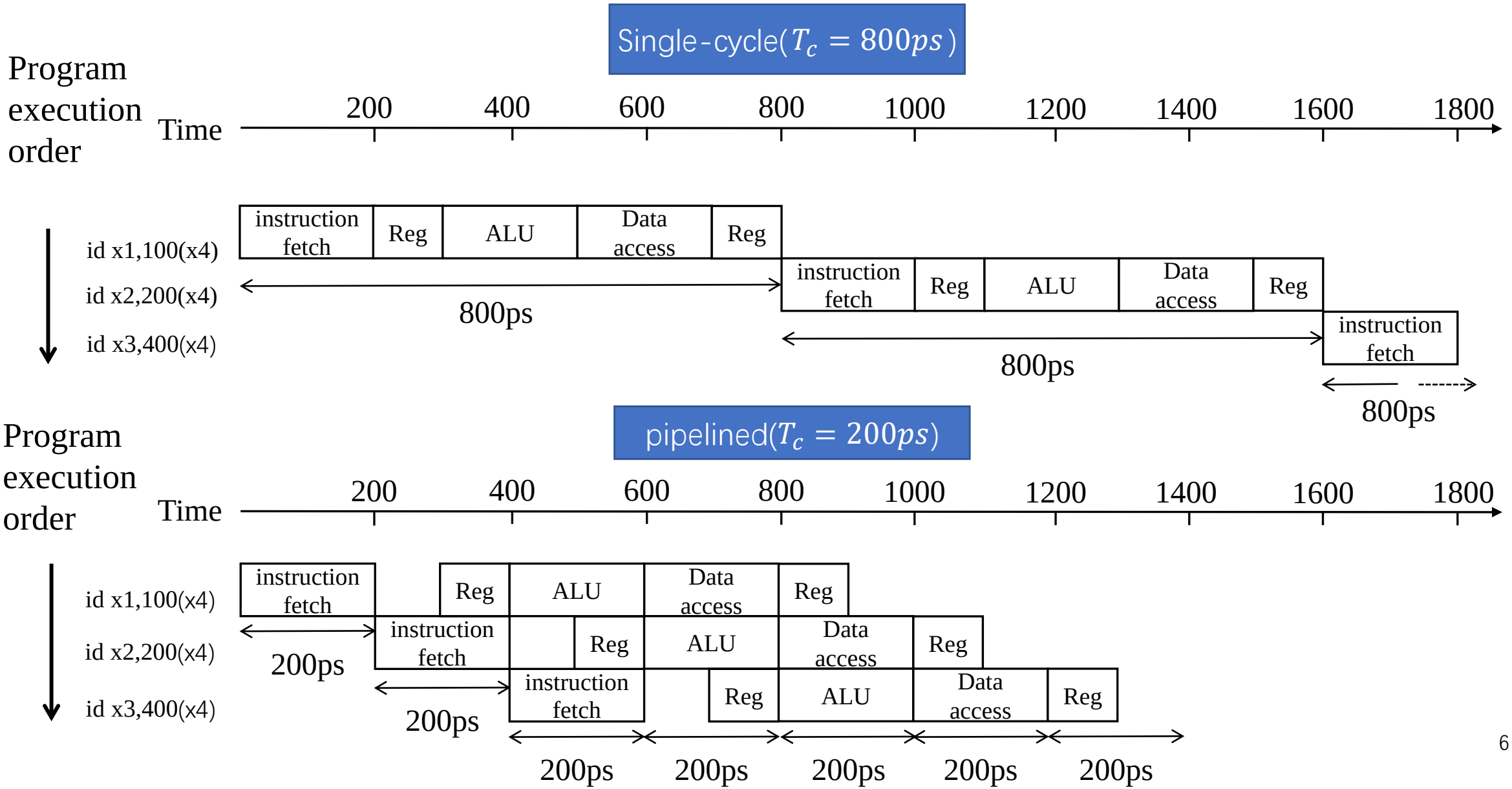
流水线性能

- 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	100 ps	800ps
Store doubleword (sd)	200ps	100 ps	200ps	200ps		700ps
R-format (add, sun, and, or)	200ps	100 ps	200ps		100 ps	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
id x1,100(x4)
id x2,200(x4)
id x3,400(x4)
instruction fetch
Reg
ALU
Data access
Reg
800ps
pipelined($T_c = 200ps$)
200ps

流水线加速比

- 如果流水线各阶段操作平衡
  - 比如，所有阶段需要相同的时间
  - 则，$\text{指令执行时间}_{\text{流水线}} = \dfrac{\text{指令执行时间非流水线}}{\text{流水线级数}}$

- 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至15个字节长度的指令
  - 只有几种指令格式，不同指令之间格式整齐
    - 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

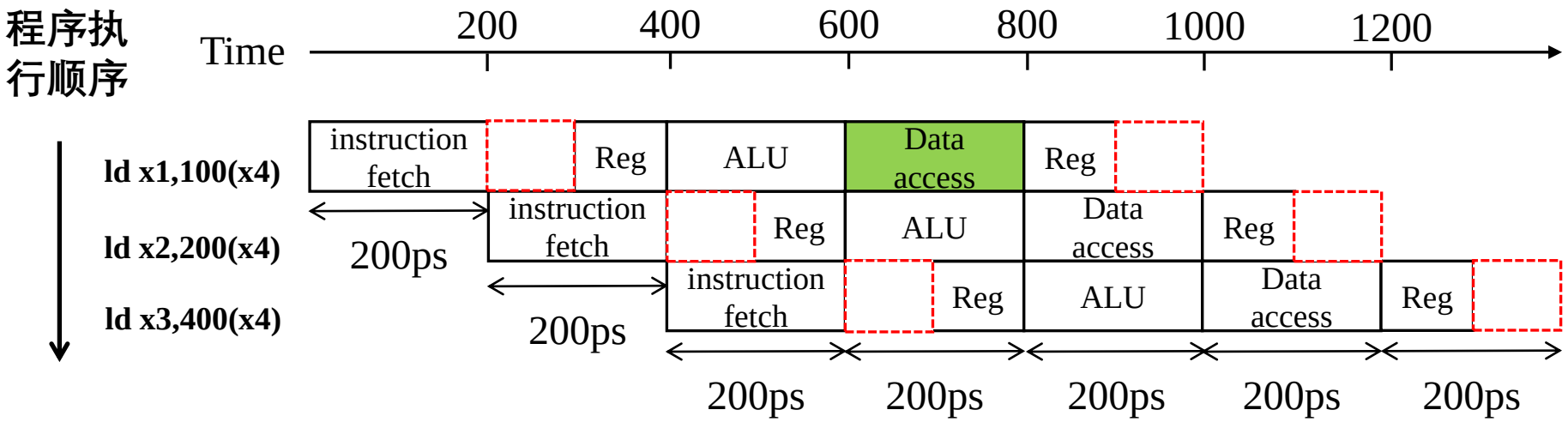
流水线冒险

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**冒险(hazard)，包括以下三种：**

- 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
- 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

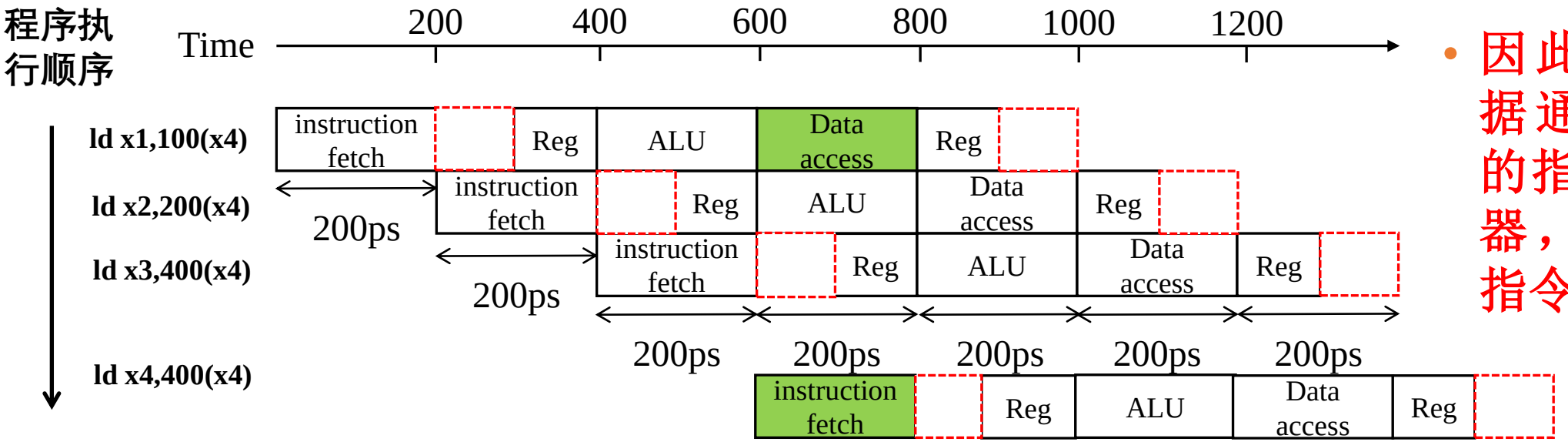
结构冒险

- **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



结构冒险

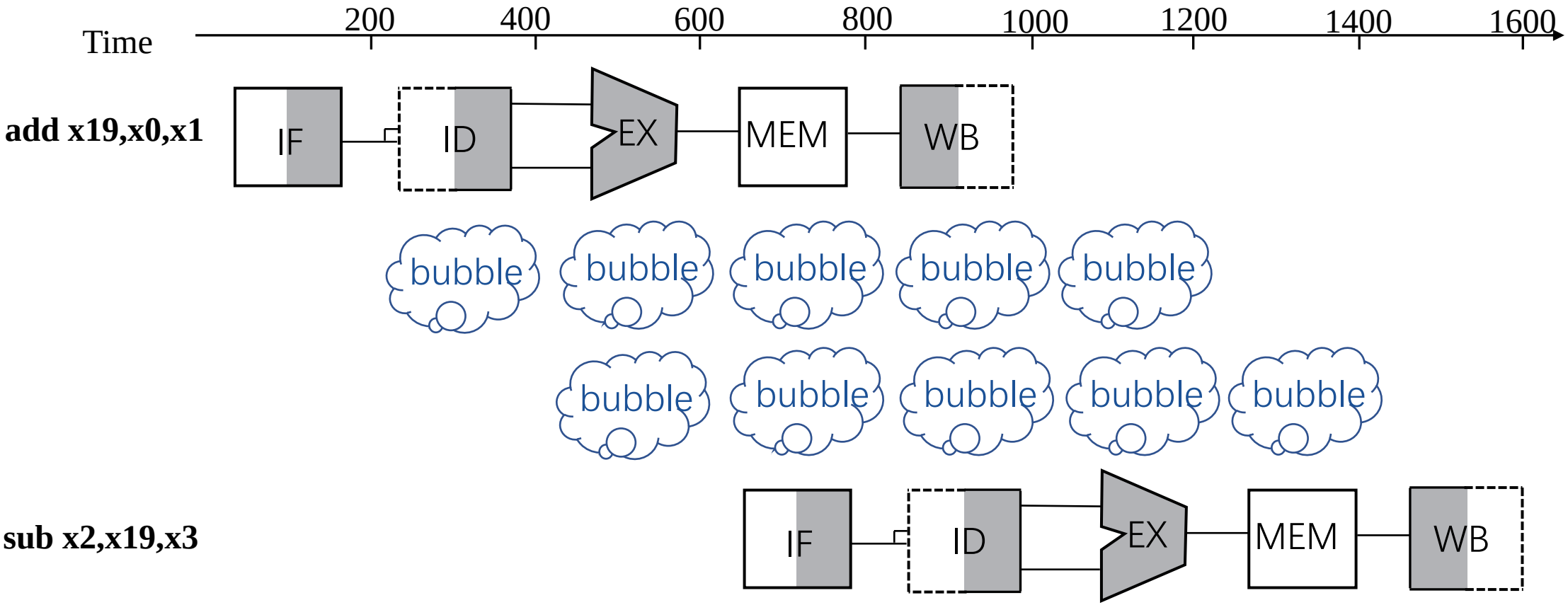
- **硬件资源**不支持多条指令在同一时钟周期执行
- 假设下图RISC-V流水线结构只有一个存储器
  - load/store需要访问存储器
  - 如果有第四条指令，则第一条指令从存储器取数据的同时，第四条指令从同一存储器取指令，流水线发生结构冒险



- 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caches

数据冒险

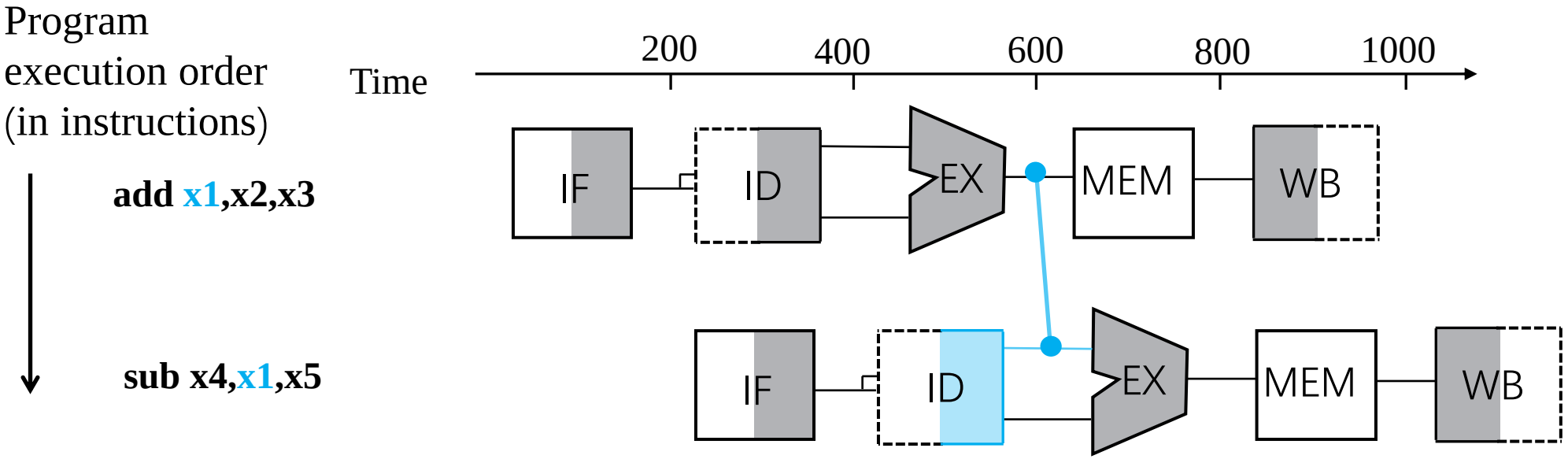
- 一条指令依赖于前面一条尚在流水线中的指令
  - add x19, x0, x1
    sub x2, x19, x3



前递（forward，也叫转发）

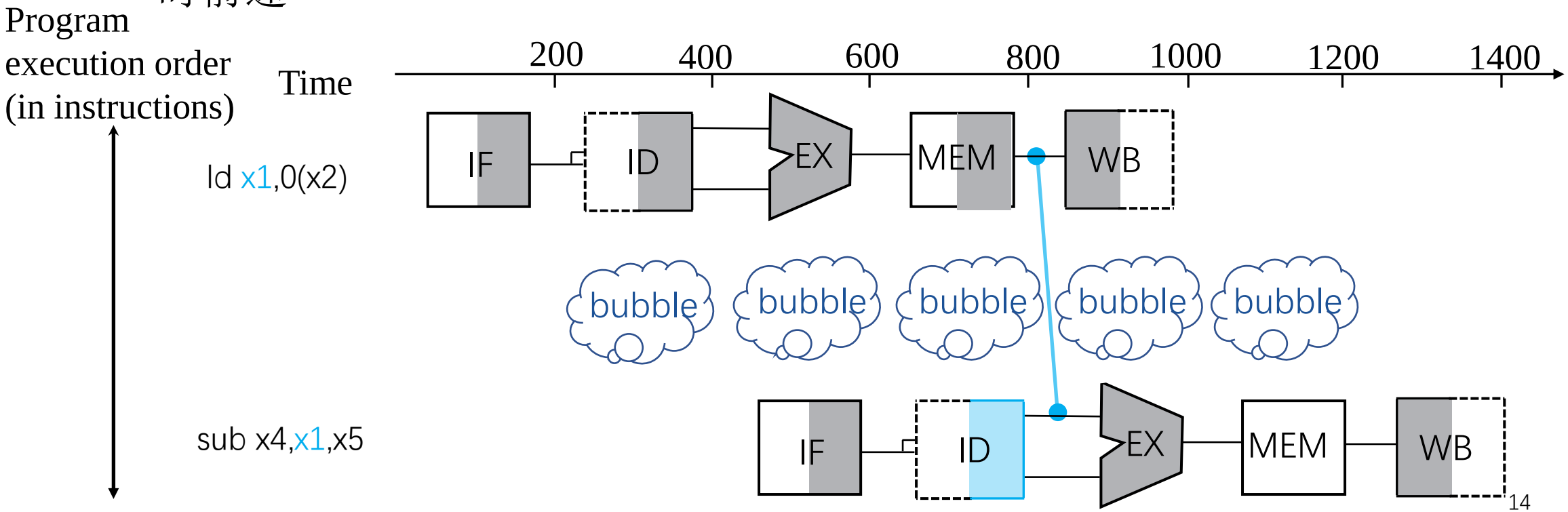
- 一种解决数据冒险的方法
  - 一旦ALU计算出结果，就提前取到数据，而不是等到数据到达寄存器或存储器
  - 需要向内部资源添加额外的硬件

注意：寄存器堆或存储器
右半阴影表示正在被读；
左半阴影表示正在被写。
（即：先写后读划分）



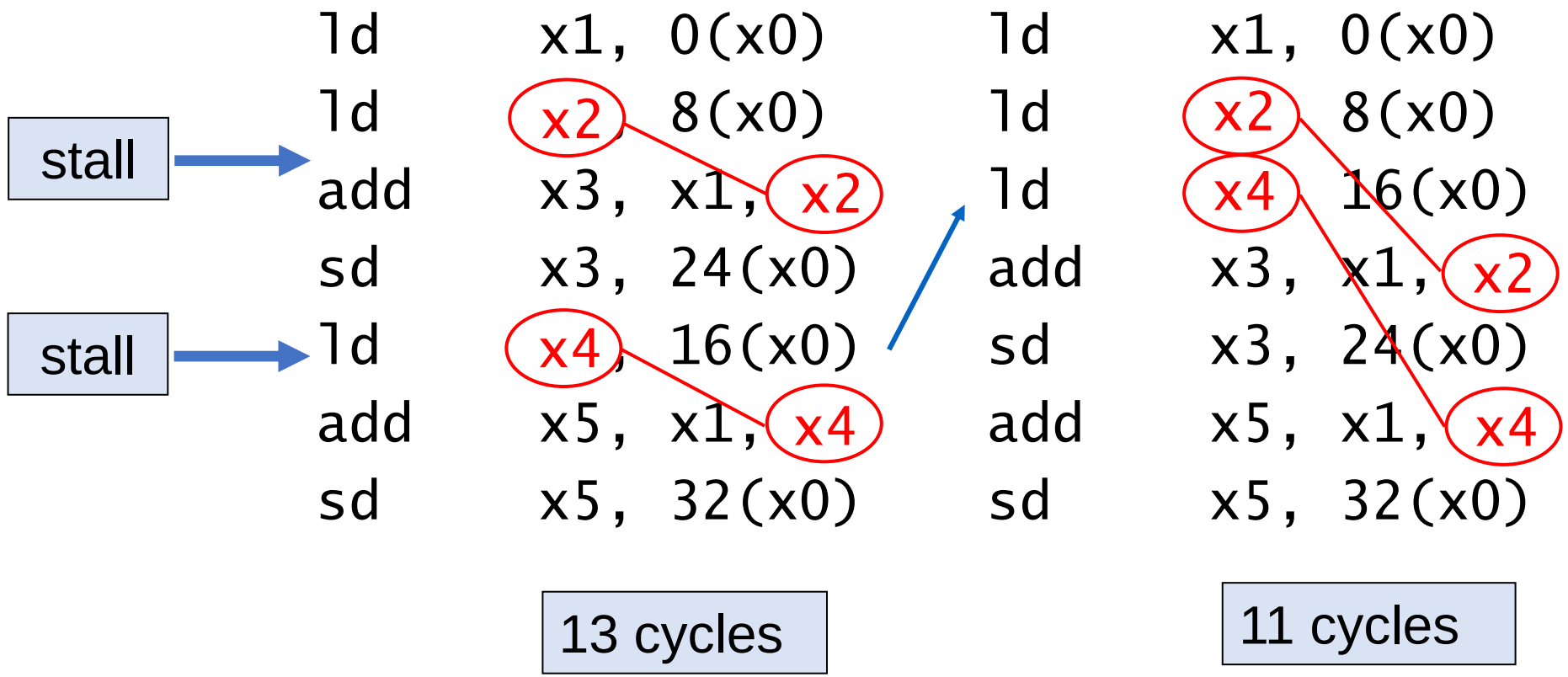
载入-使用型数据冒险

- 前递不能避免所有的流水线停顿
  - 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
  - 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



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;



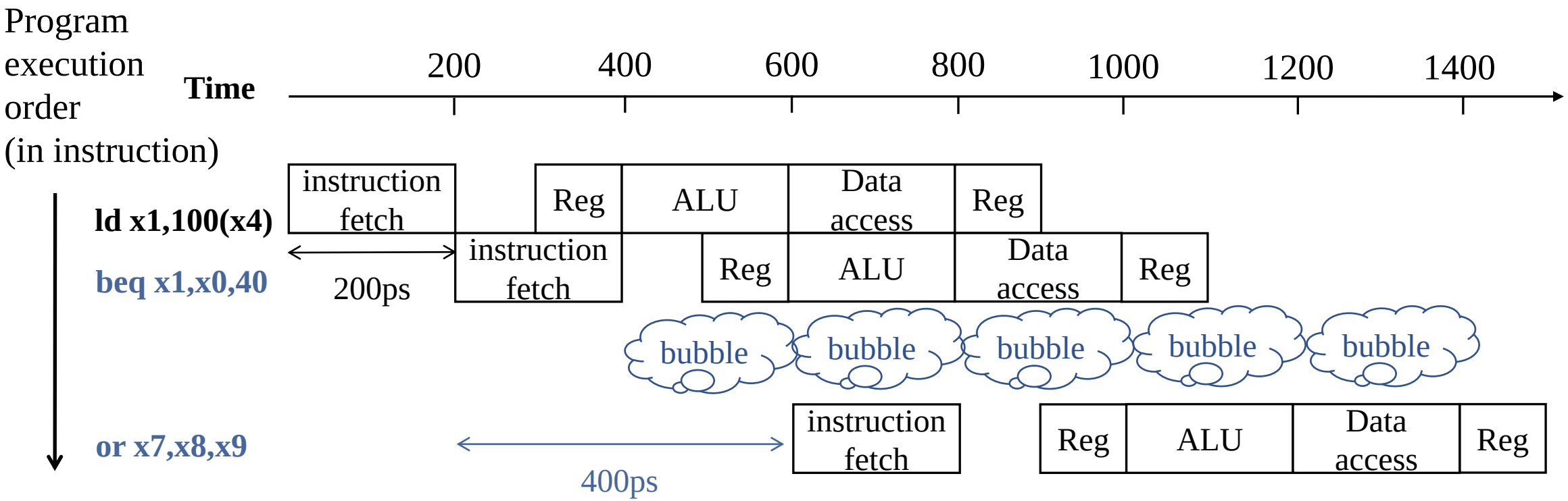
控制冒险

- 分支决定了控制流
  - 分支指令之后的IF阶段依赖于分支的结果
  - 流水线无法保证一直都可以取到正确的指令
    - 此时，分支指令仍在ID阶段

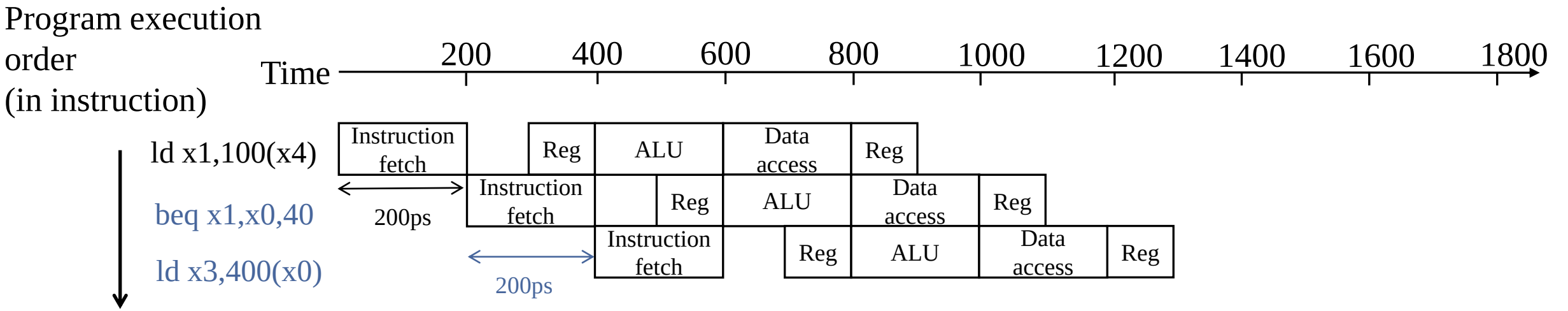
- 在RISC-V流水线中
  - 尽可能早地完成寄存器比较、分支目标地址计算
  - 添加硬件，使得以上能在ID阶段完成

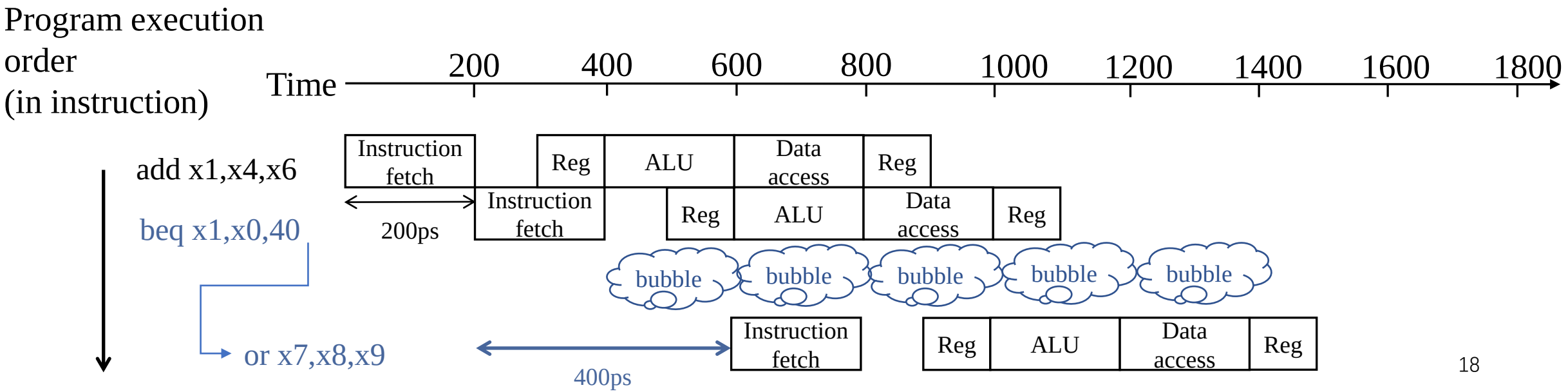
每遇到条件分支指令就停顿的流水线

- 在取下一条指令之前，等待分支结果
  - beq指令的ID阶段得到分支结果



总是预测条件分支不发生跳转





分支预测

- 对较长的流水线而言，通常无法在第二阶段解决分支指令的问题
  - 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
- 可以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

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

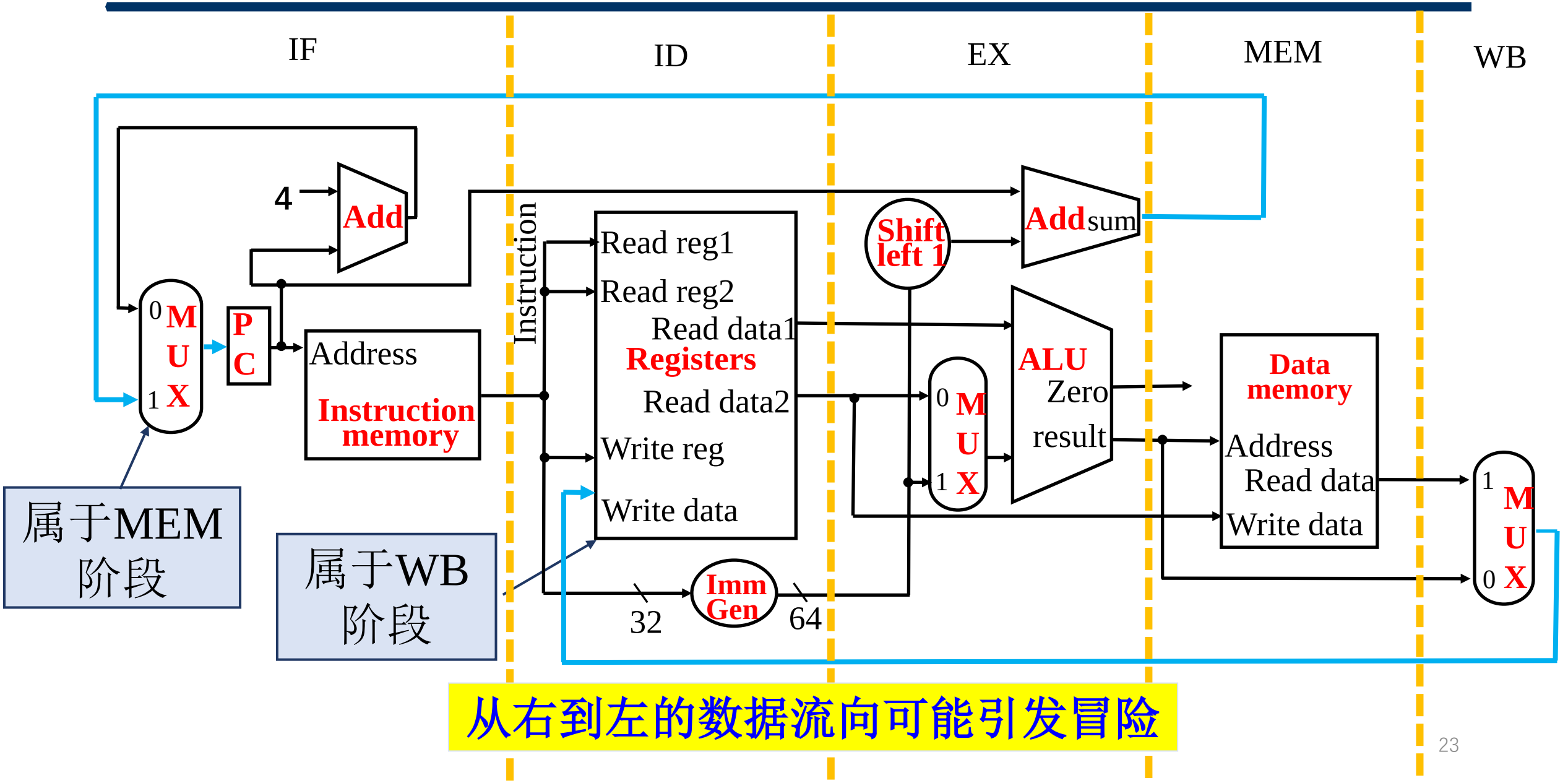
流水线总结

- 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

第八章

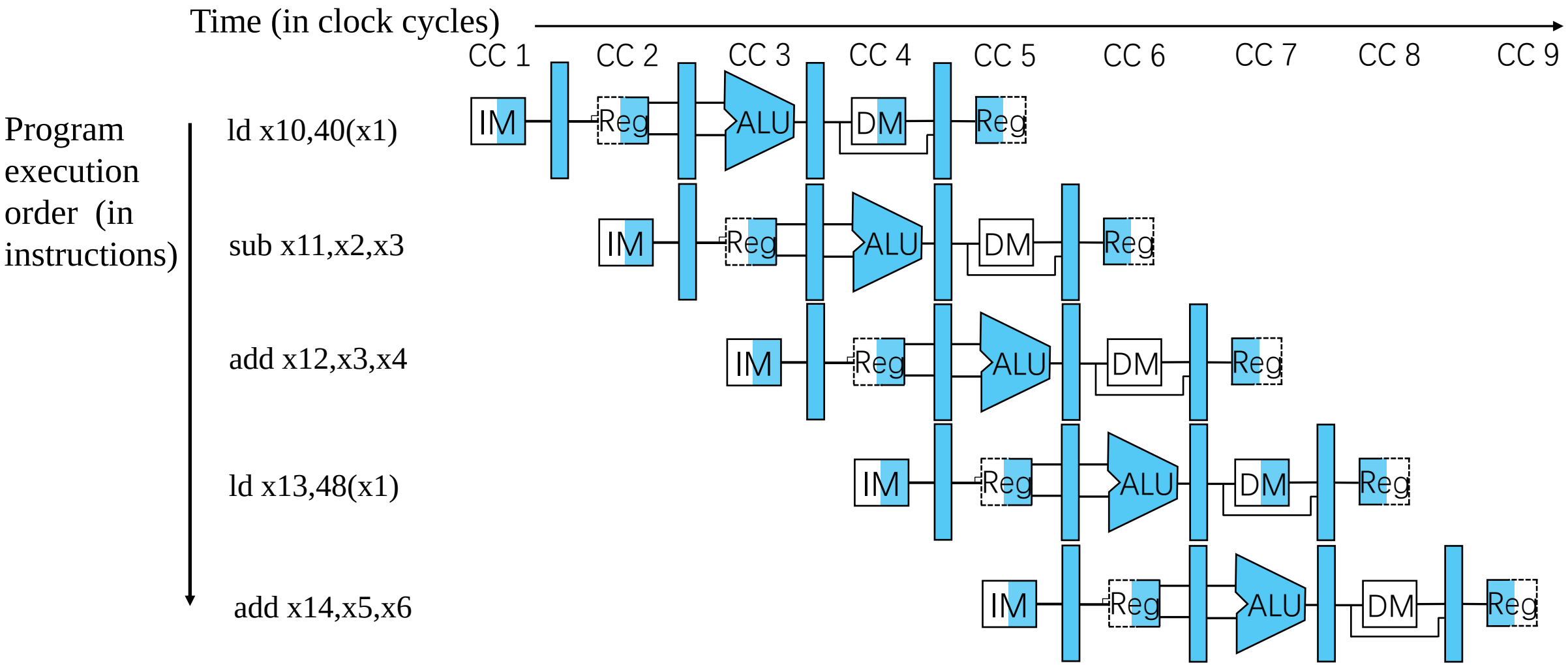
- 流水线概述
- **流水线数据通路和控制**
- 数据冒险：前递与停顿
- 控制冒险
- 例外
- 指令间的并行性

构建RISC-V流水线数据通路



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

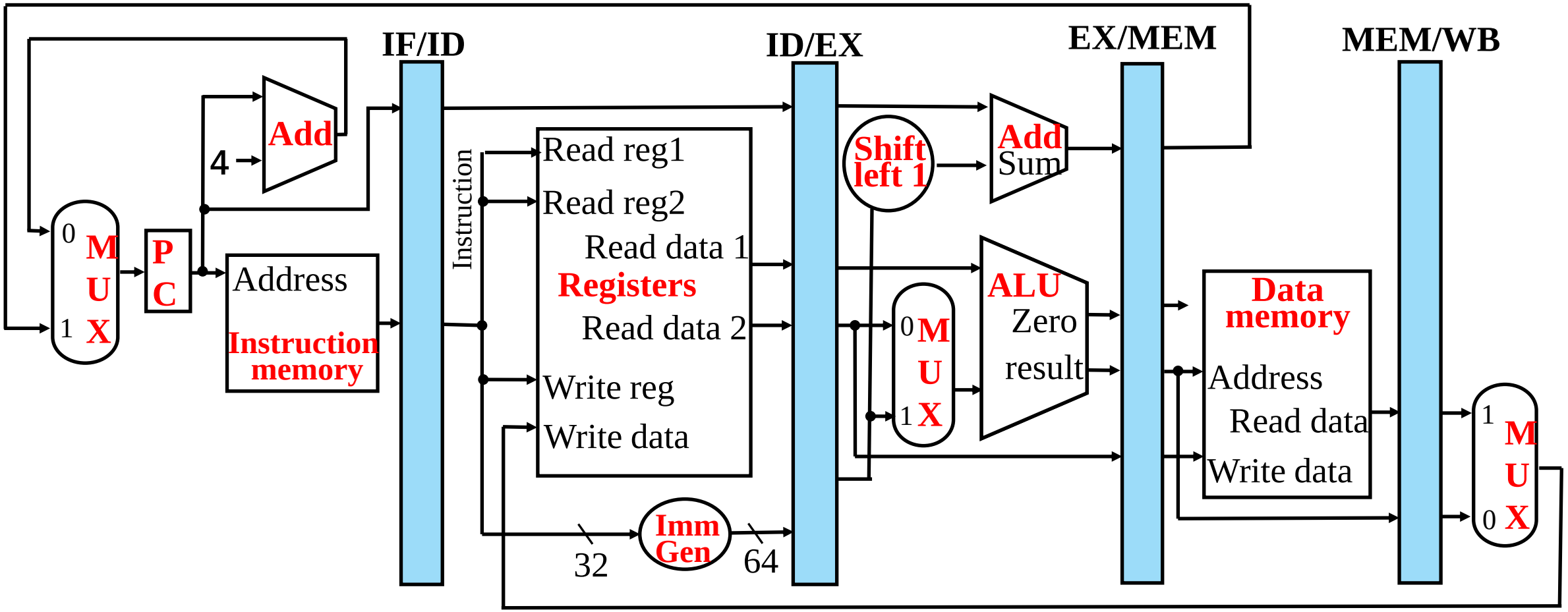


观察流水线操作

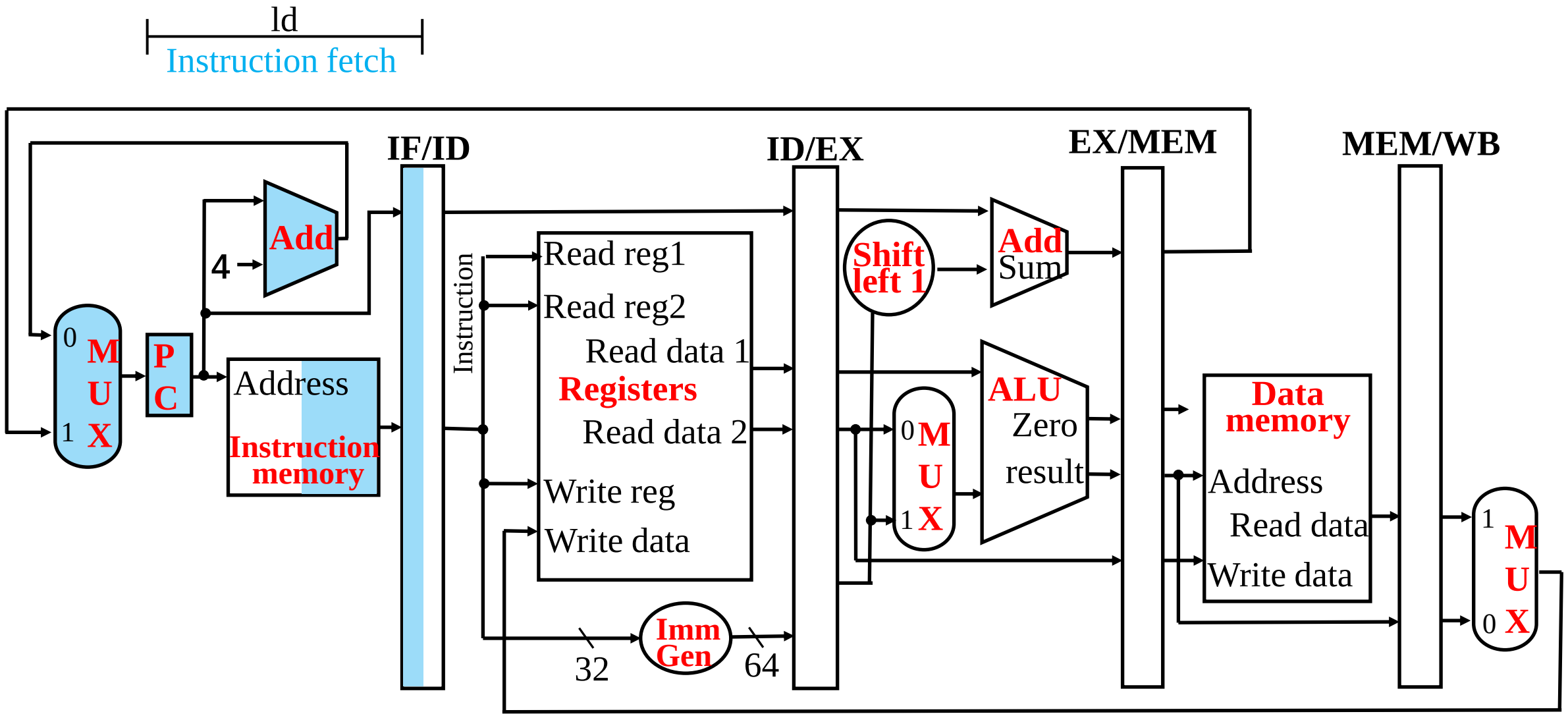
- 在流水线数据通路中，观察指令在不同周期的流动
  - 单时钟周期流水线图
    - 展现一个时钟周期中流水线的使用
    - 高亮表示使用到的硬件资源
  - 多时钟周期流水线图
    - 展现一段时间的情况
- 接下来，用单时钟周期流水线图，观察load和store指令的操作

加入流水线寄存器

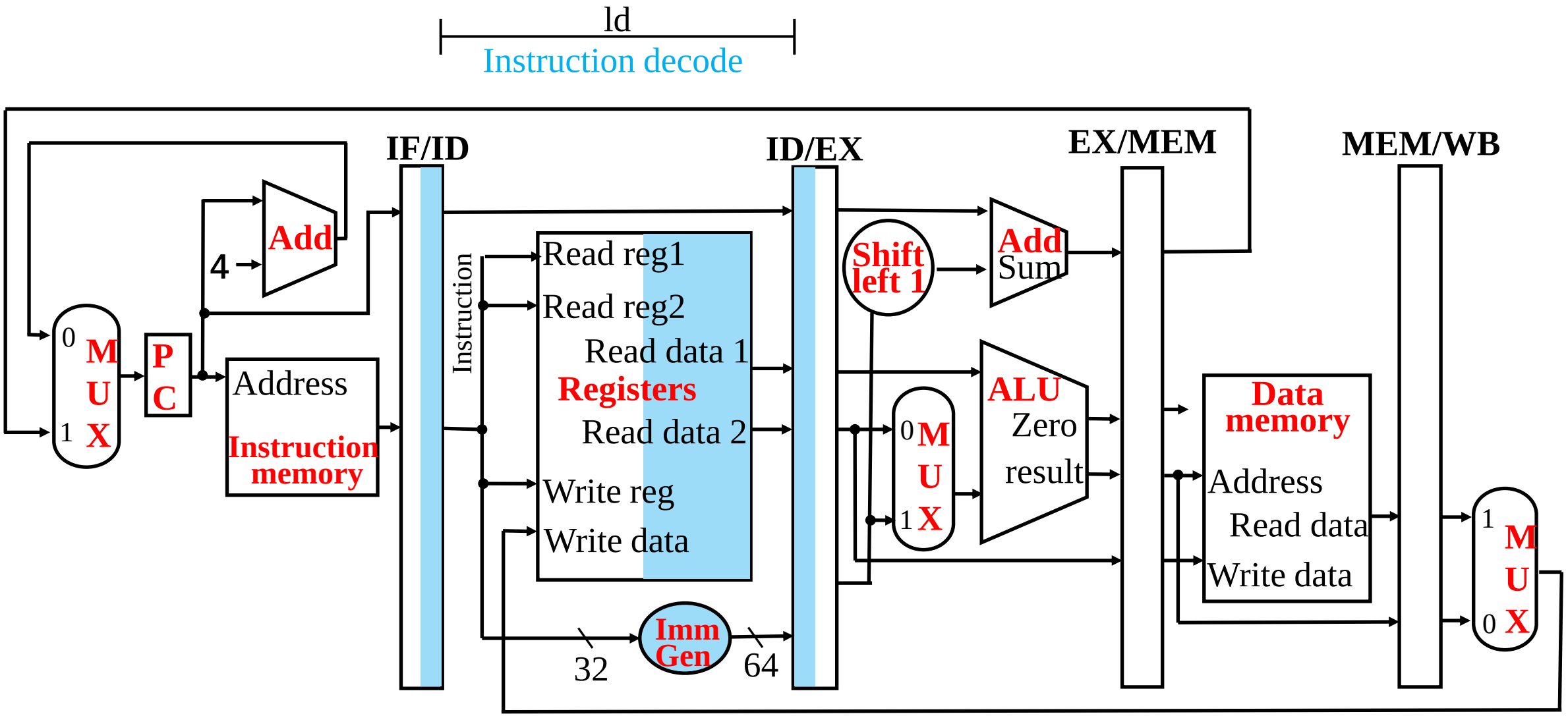
- 在不同阶段之间需要寄存器
  - 保存前一个阶段产生的信息



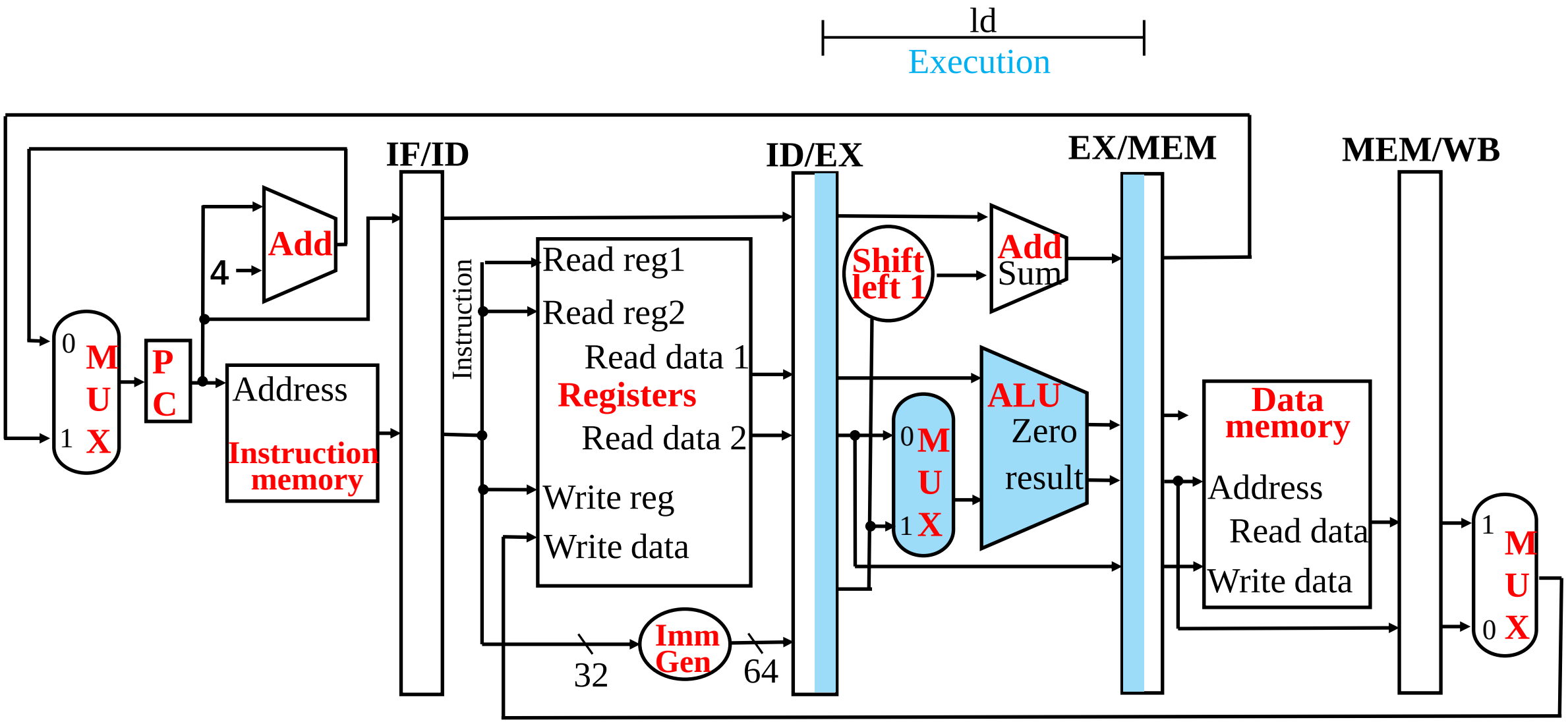
IF for Load, Store, …



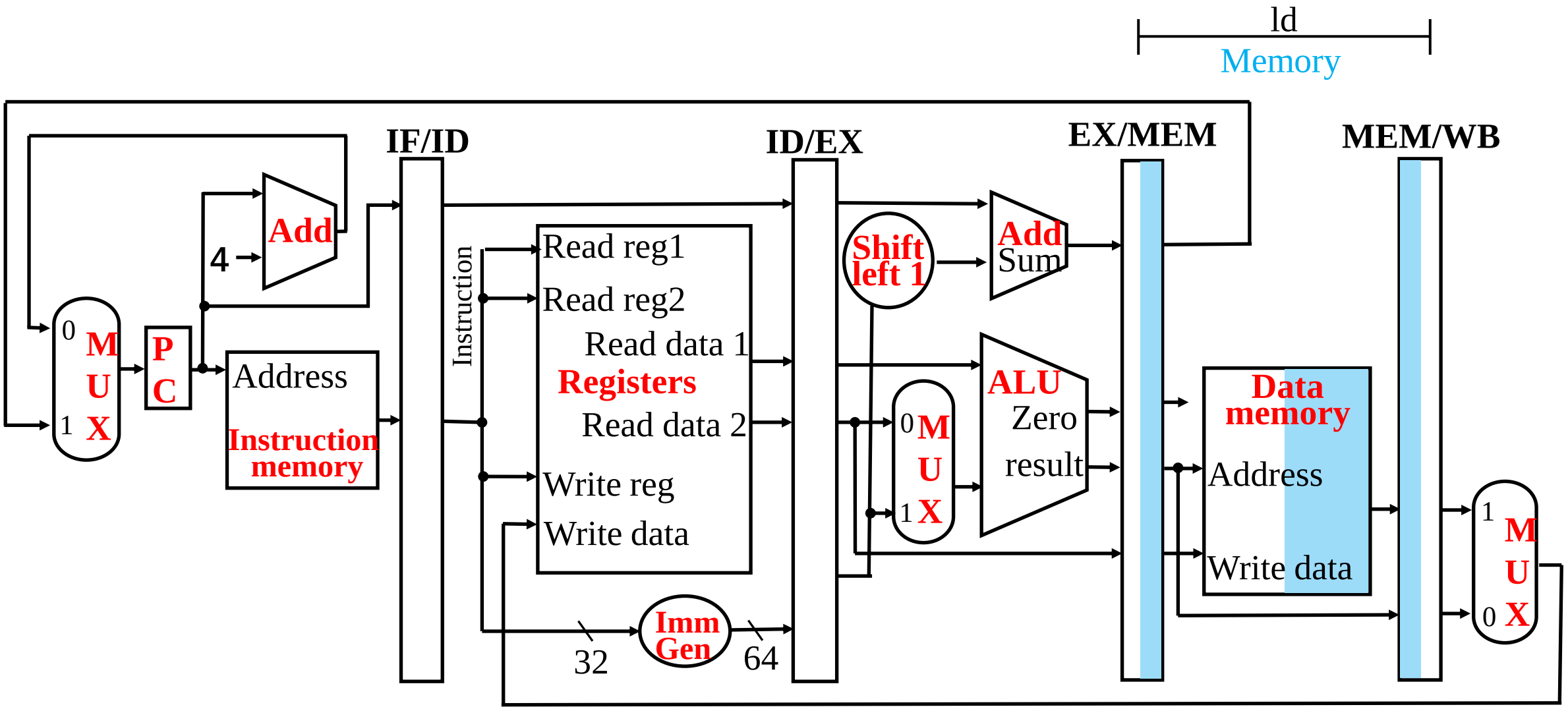
ID for Load, Store, …



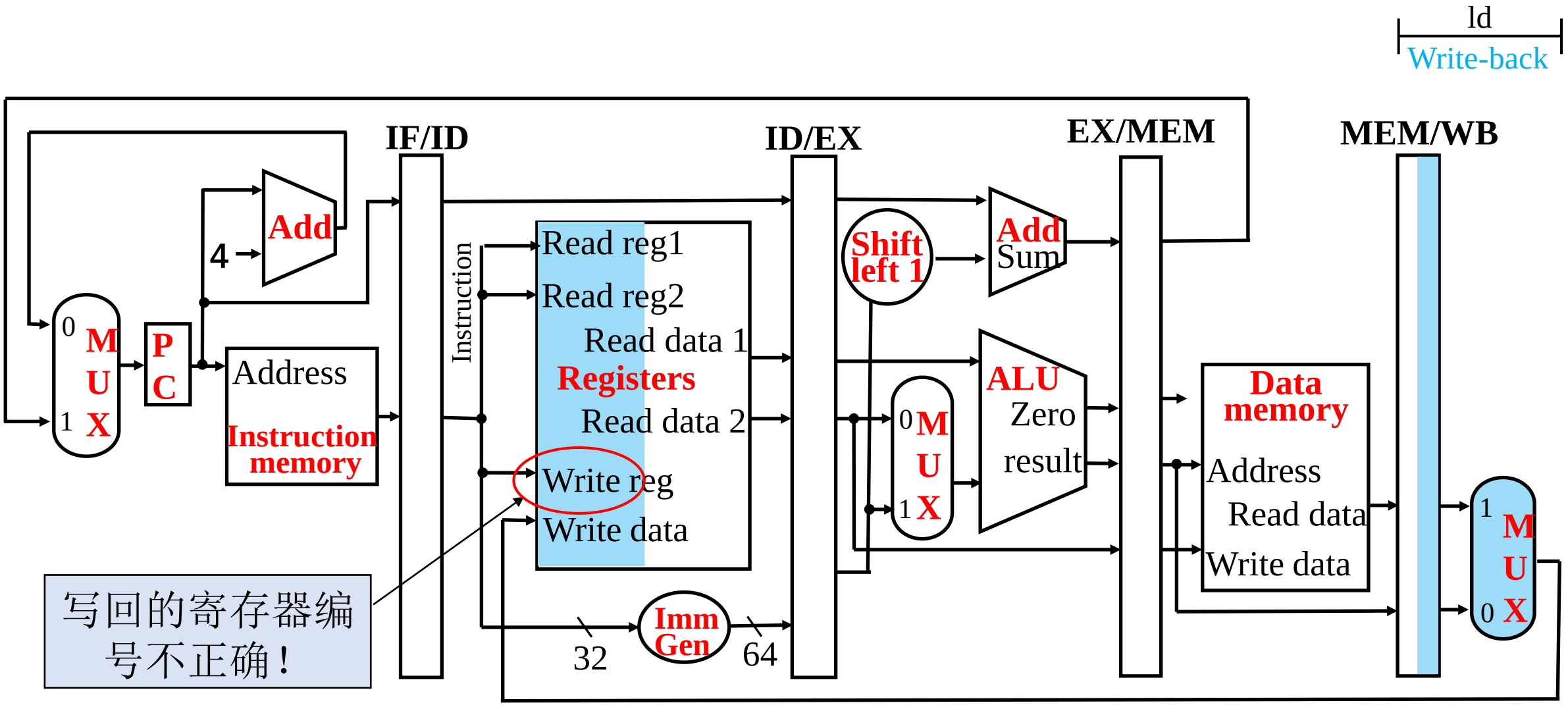
EX for Load


ld
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
U
X
0
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
Data memory
Address
Read data
Write data

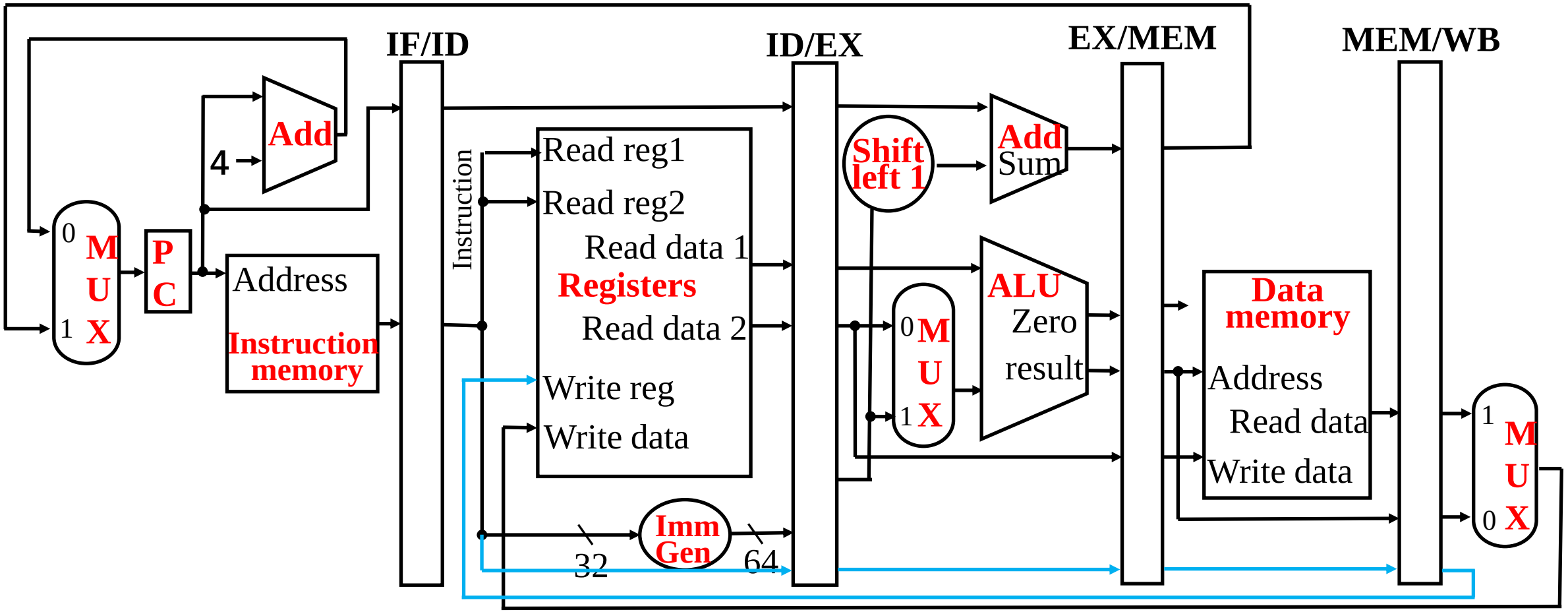
MEM for Load


ld
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
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
0
M
U
X
1
ALU
Zero
result
Data memory
Address
Write data
1
M
U
X
0

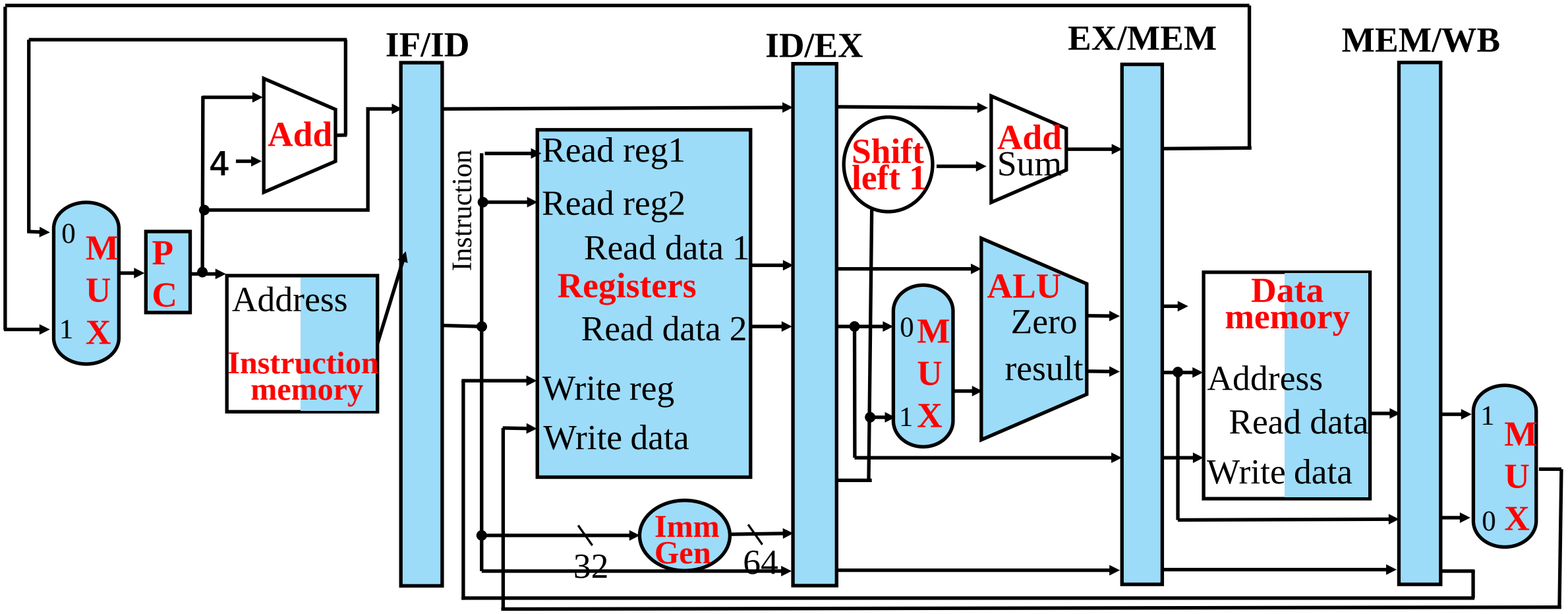
WB for Load


ld
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
U
X
0
1
PC
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
Data memory
Address
Read data
Write data
写回的寄存器编号不正确！

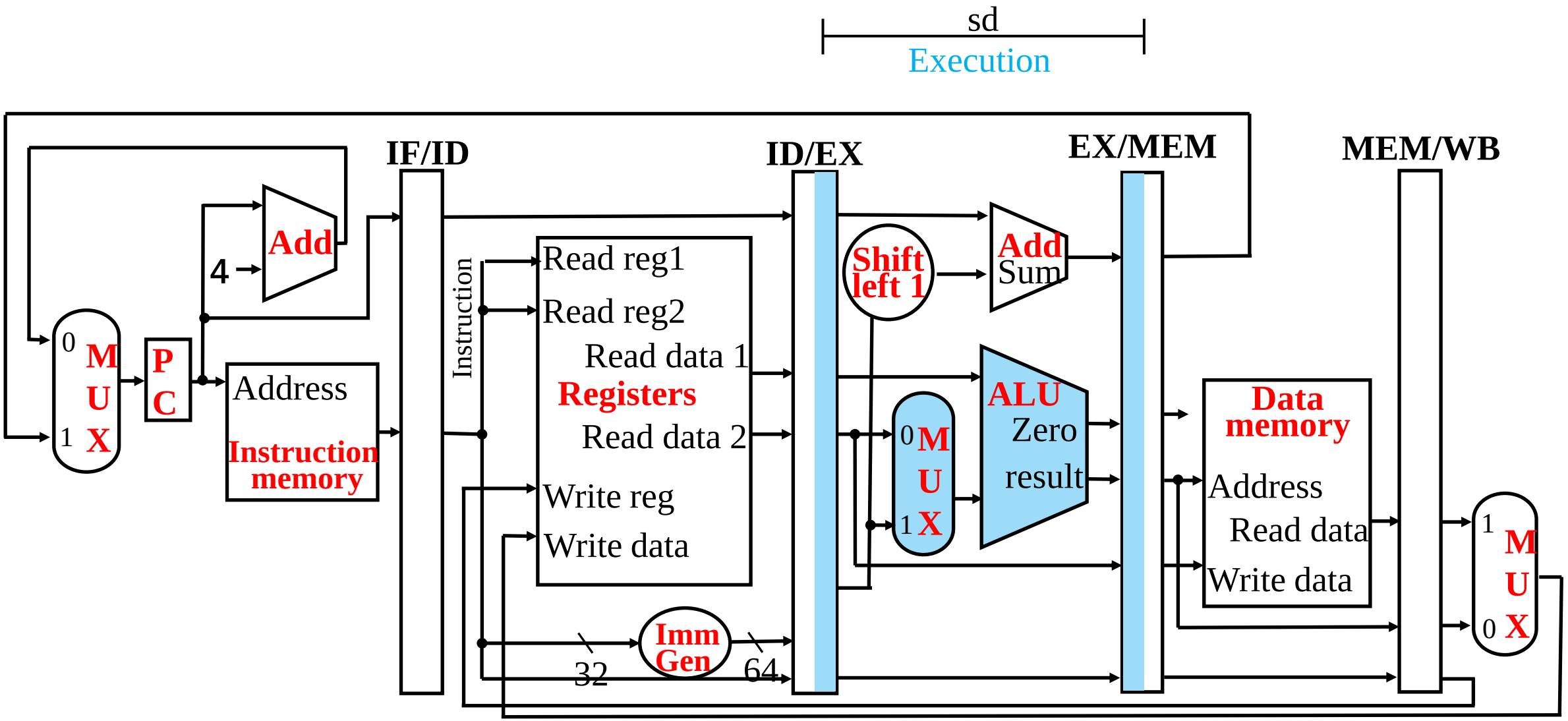
更正之后的数据通路


IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
U
X
0
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
Data memory
Address
Read data
Write data

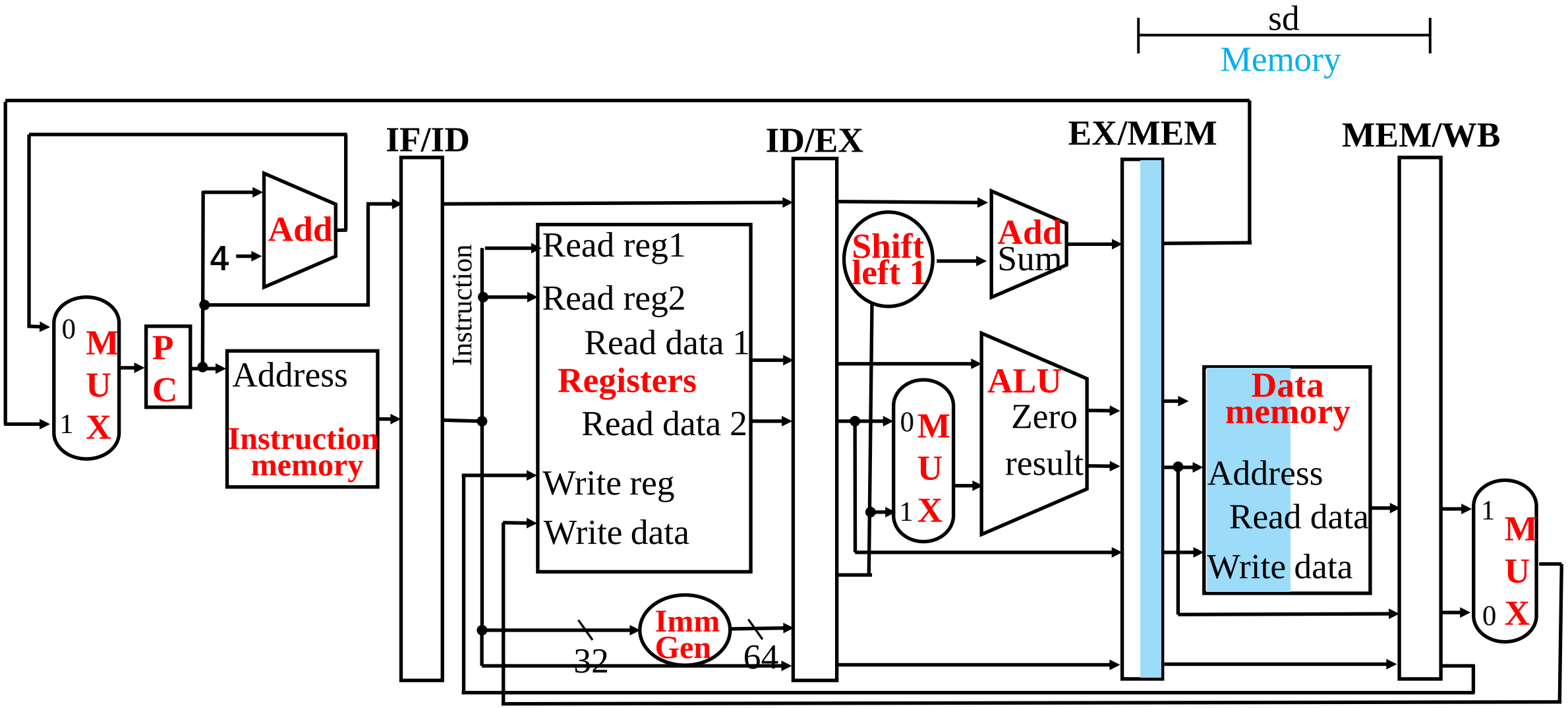
用于ld指令的全部五个流水线阶段部分



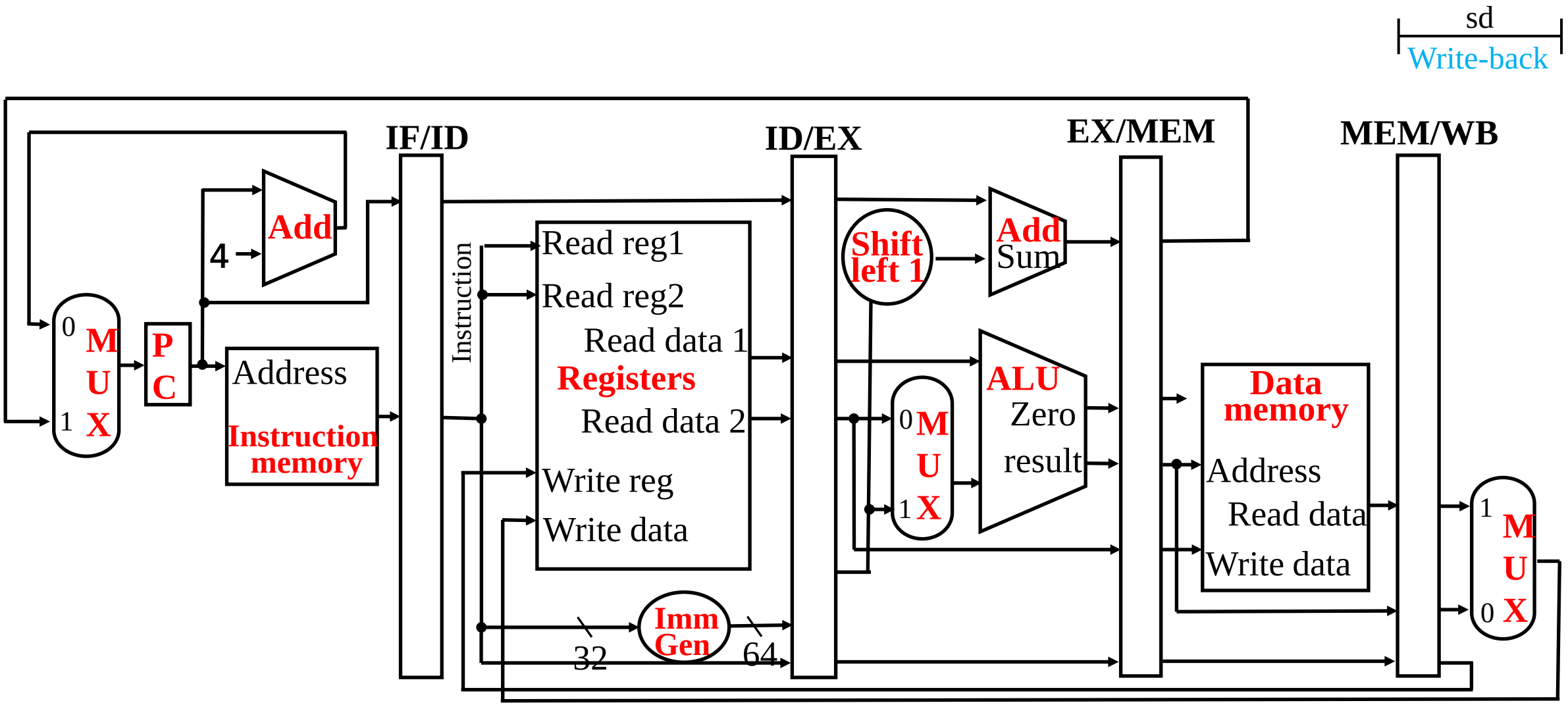
EX for Store


sd
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
U
X
0
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
Data memory
Address
Read data
Write data

MEM for Store

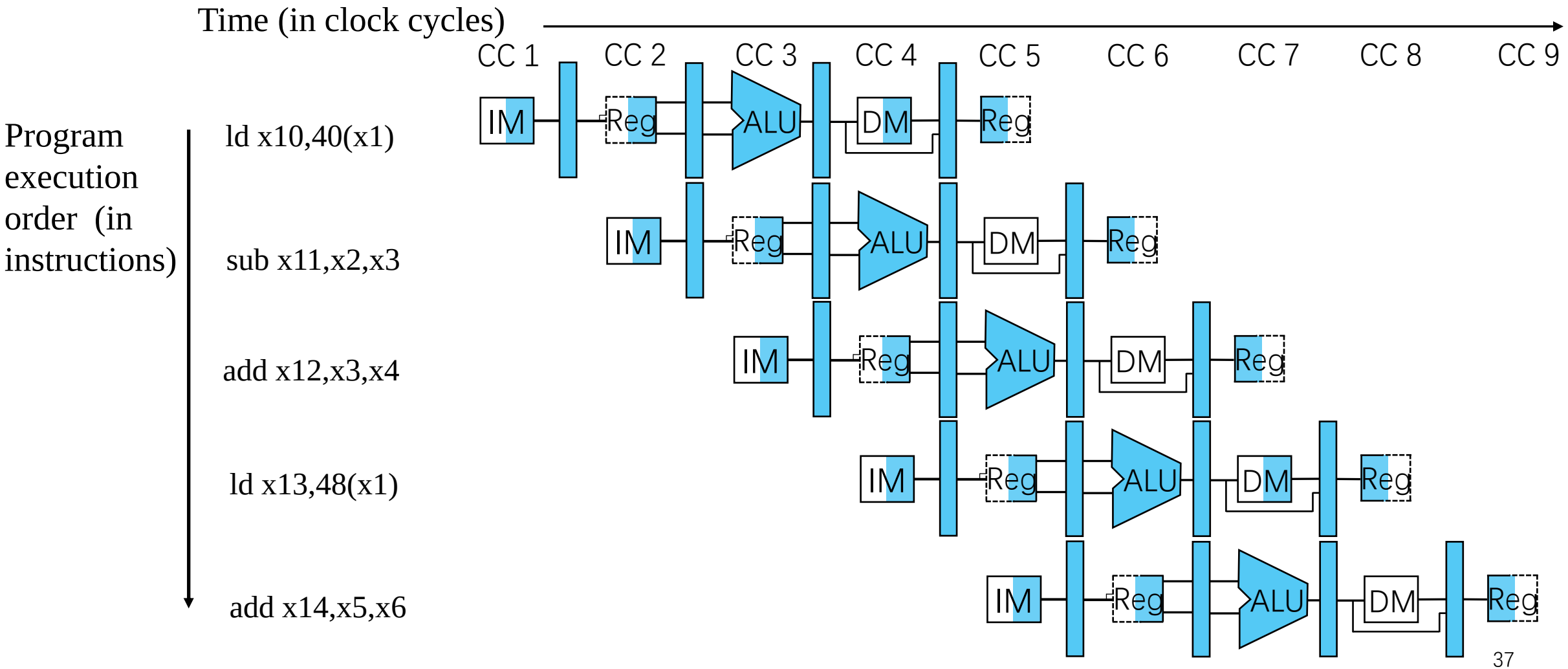


WB for Store

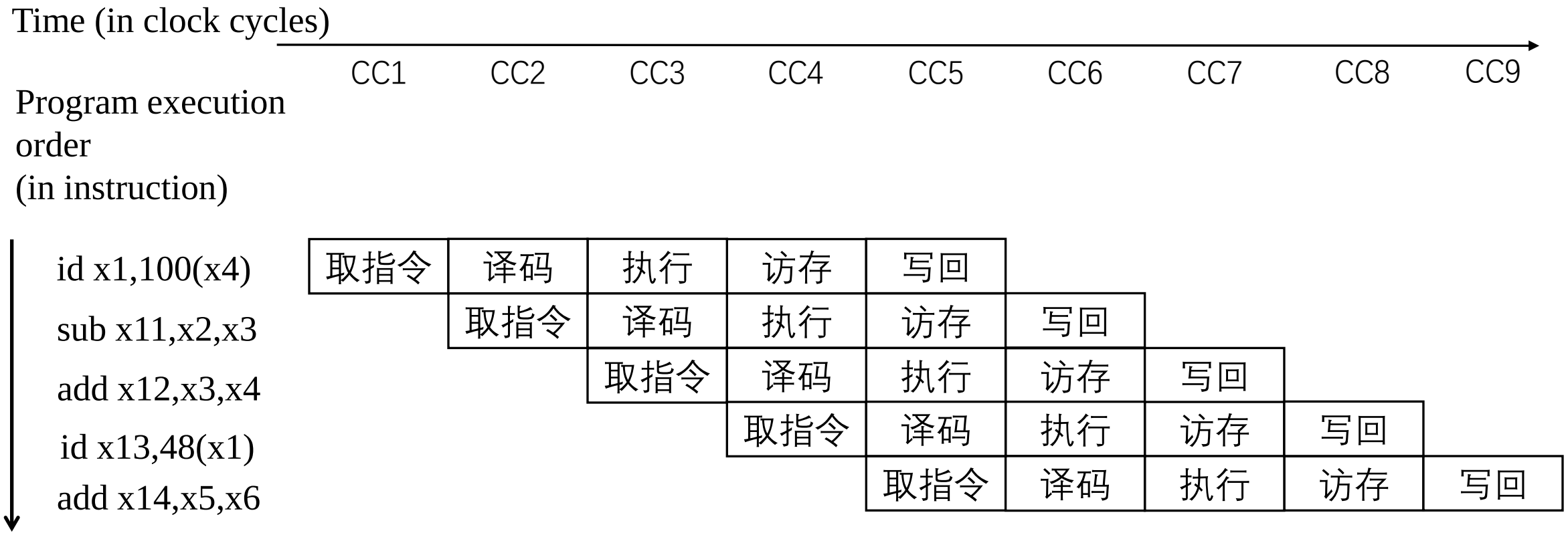

sd
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
U
X
0
1
P
C
Address
Instruction memory
Instruction
Read reg1
Read reg2
Read data 1
Registers
Read data 2
Write reg
Write data
Imm Gen
32
64
Shift left 1
Add
Sum
ALU
Zero
result
Data memory
Address
Read data
Write data

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

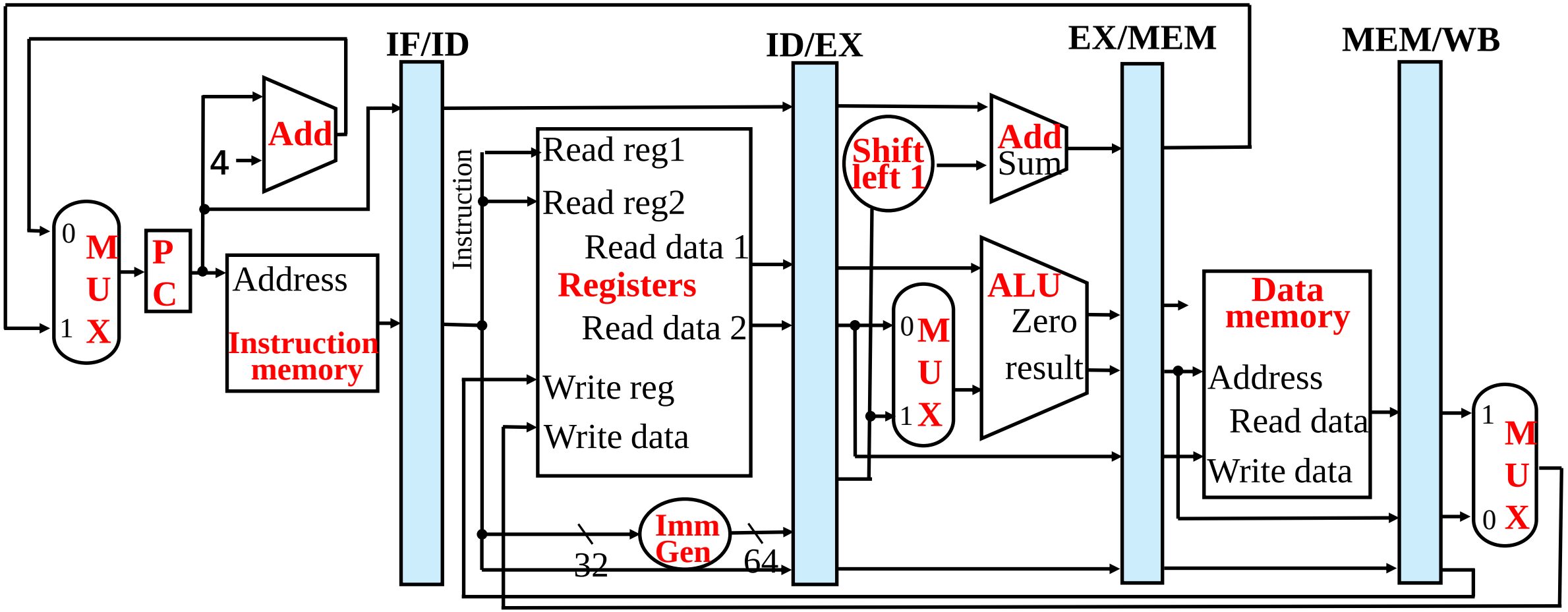


多时钟周期流水线图


Time (in clock cycles)
CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
CC9
Program execution order (in instruction)
id x1,100(x4)
sub x11,x2,x3
add x12,x3,x4
id x13,48(x1)
add x14,x5,x6
取指令
译码
执行
访存
写回

单时钟周期流水线图

- 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

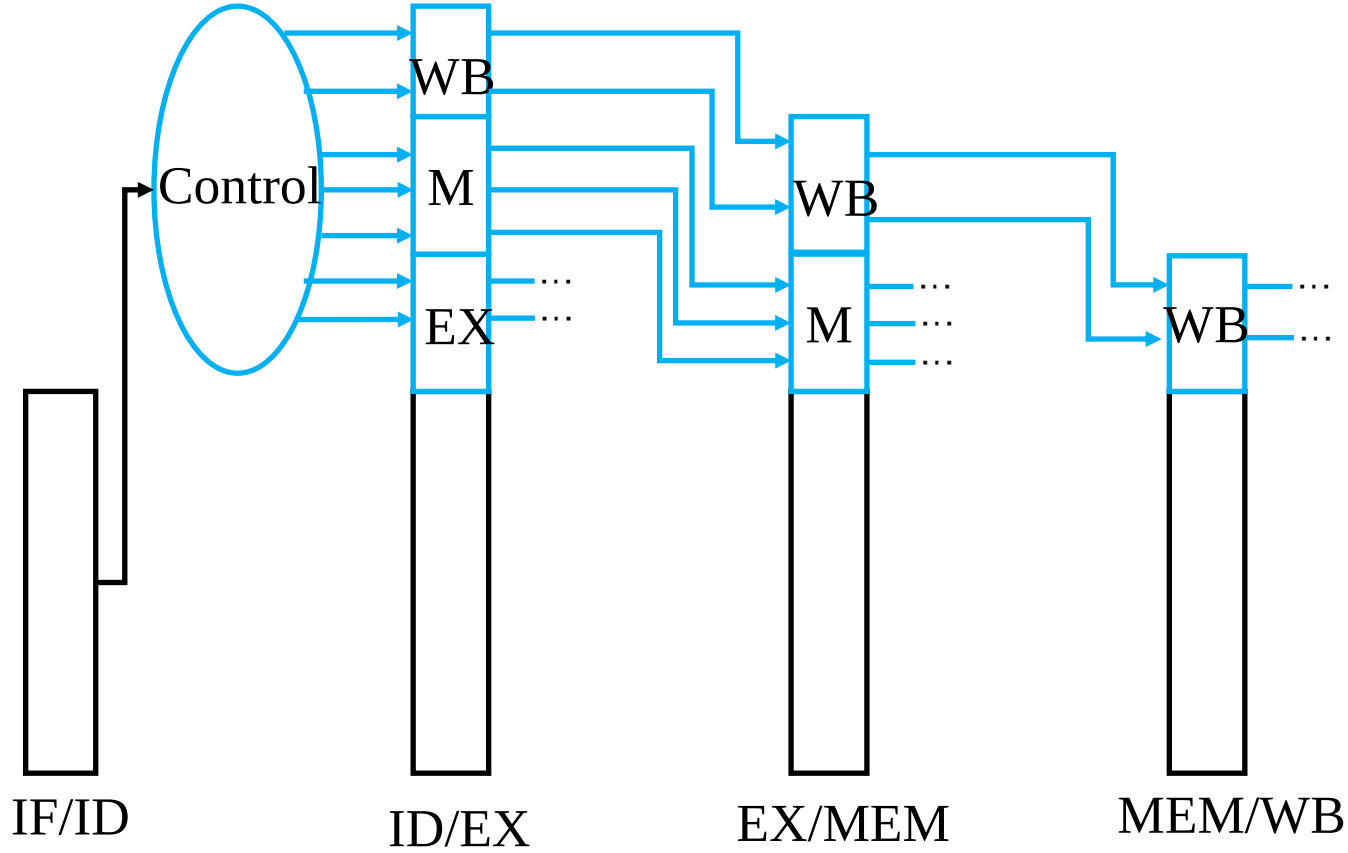


Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

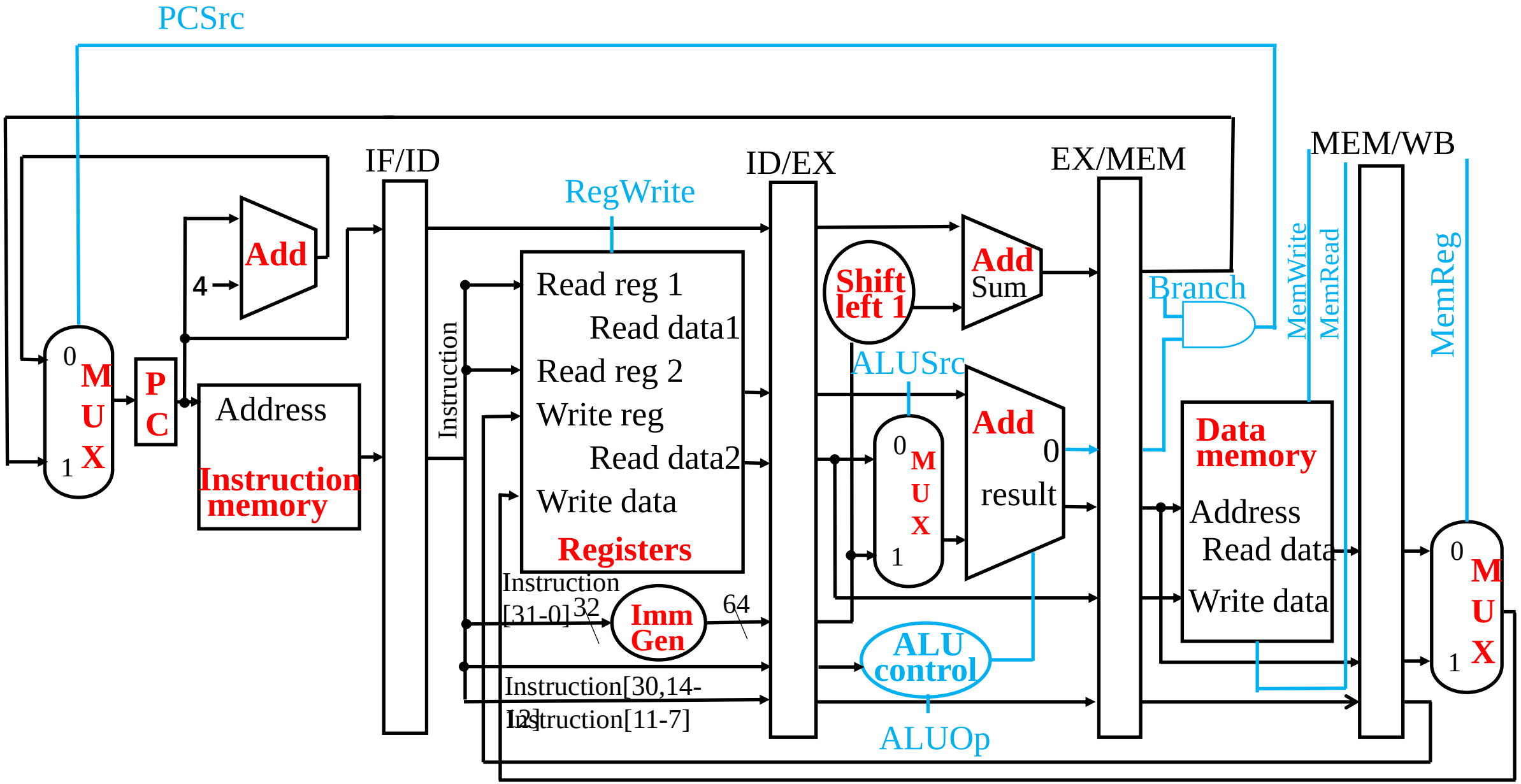
FIGURE 4.47 **The values of the control lines are the same as in Figure 4.18, but they have been shuffled into three groups corresponding to the last three pipeline stages.**

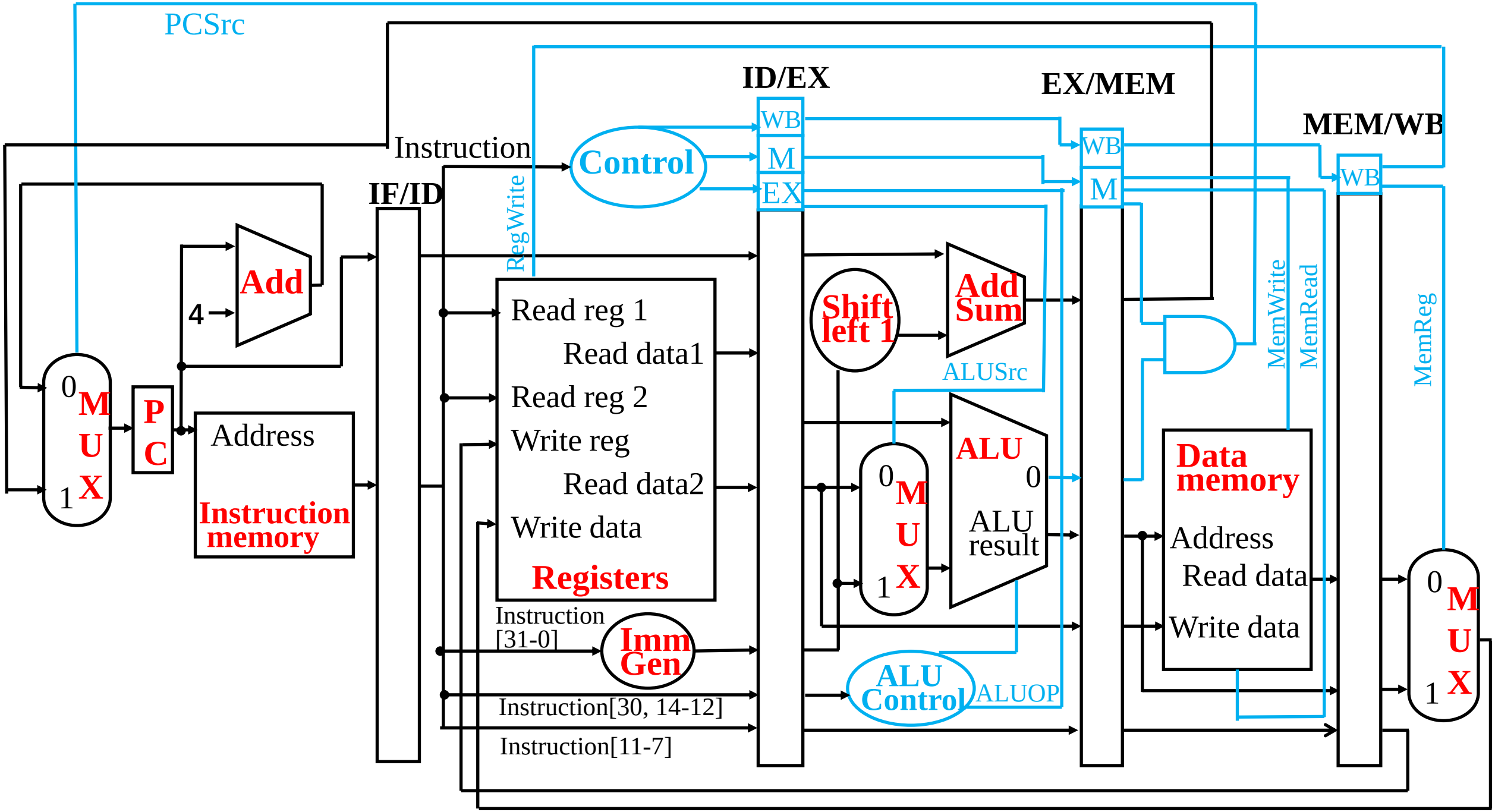
流水线控制

- 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五组
  - IF和ID阶段没有需要特别控制的内容



简单的流水线控制


PCSrc
IF/ID
RegWrite
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
U
X
1
P
C
Address
Instruction
memory
Instruction
Read reg 1
Read data1
Read reg 2
Write reg
Read data2
Write data
Registers
Instruction
[31-0]
32
Imm
Gen
64
Instruction[30,14-
12]
Instruction[11-7]
Shift
left 1
Add
Sum
ALUSrc
0
M
U
X
1
Add
0
result
ALU
control
ALUOp
Branch
MemWrite
MemRead
MemReg
Data
memory
Address
Read data
Write data
0
M
U
X
1


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]
Shift left 1
Add Sum
ALUSrc
ALU
0
ALU result
ALU Control
ALUOP
MemWrite
MemRead
MemReg
Data memory
Address
Read data
Write data

第七章

- 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
- 指令间的并行性

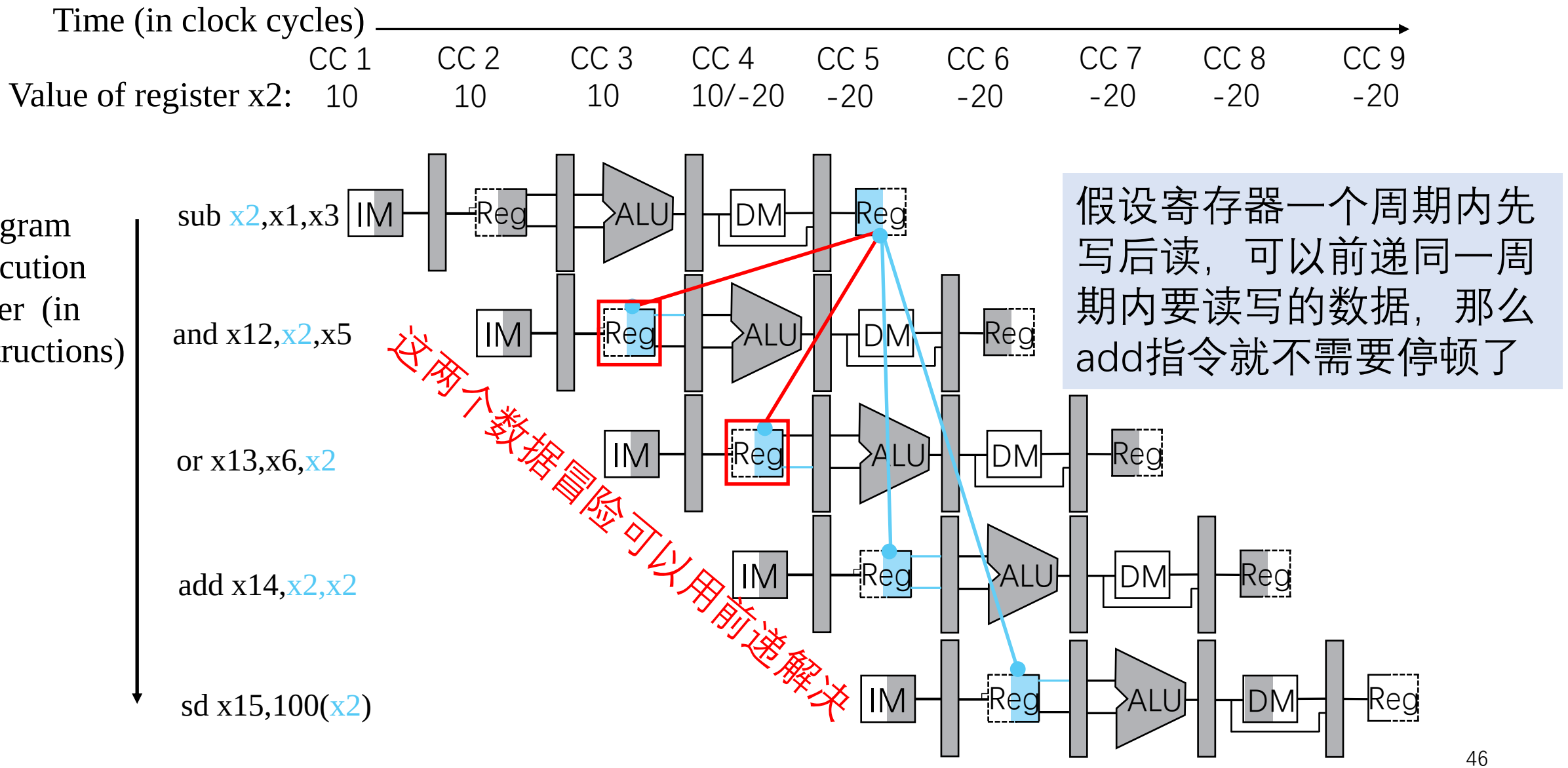
ALU相关指令中的数据冒险

- 考虑以下指令序列:

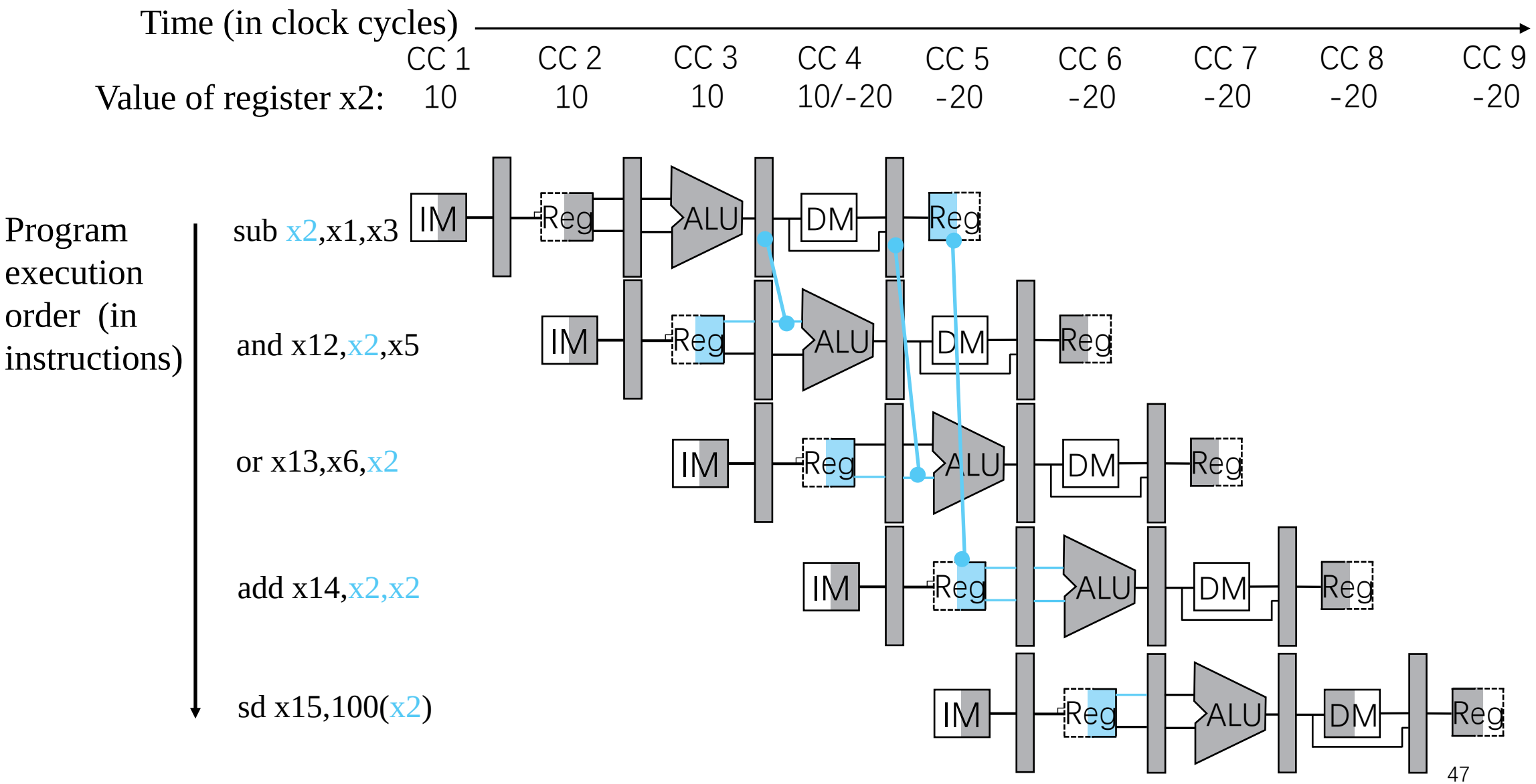
```
sub   x2, x1,x3
and   x12,x2,x5
or    x13,x6,x2
add   x14,x2,x2
sd    x15,100(x2)
```

- 我们能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/-20
-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
这两个数据冒险可以用前递解决
假设寄存器一个周期内先写后读，可以前递同一周期内要读写的数据，那么add指令就不需要停顿了

前递的结果


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/-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

检测前递发生

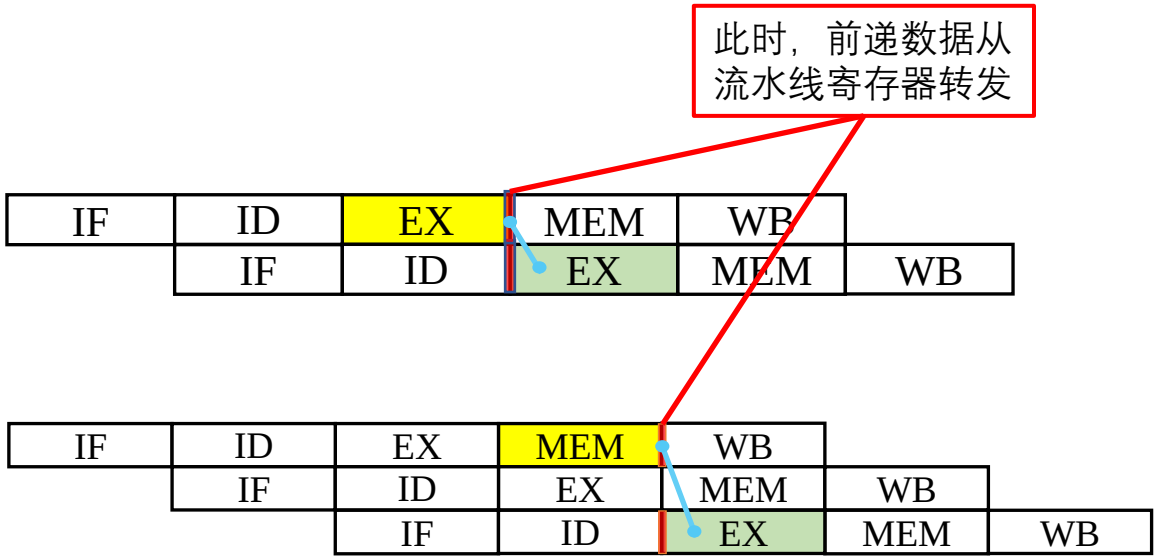
- 命名流水线寄存器字段

  例如：ID/EX.RegisterRs1表示的是：一个寄存器号，它的值存在ID/EX流水线寄存器中。
  即这个寄存器堆中第一个读端口的值。

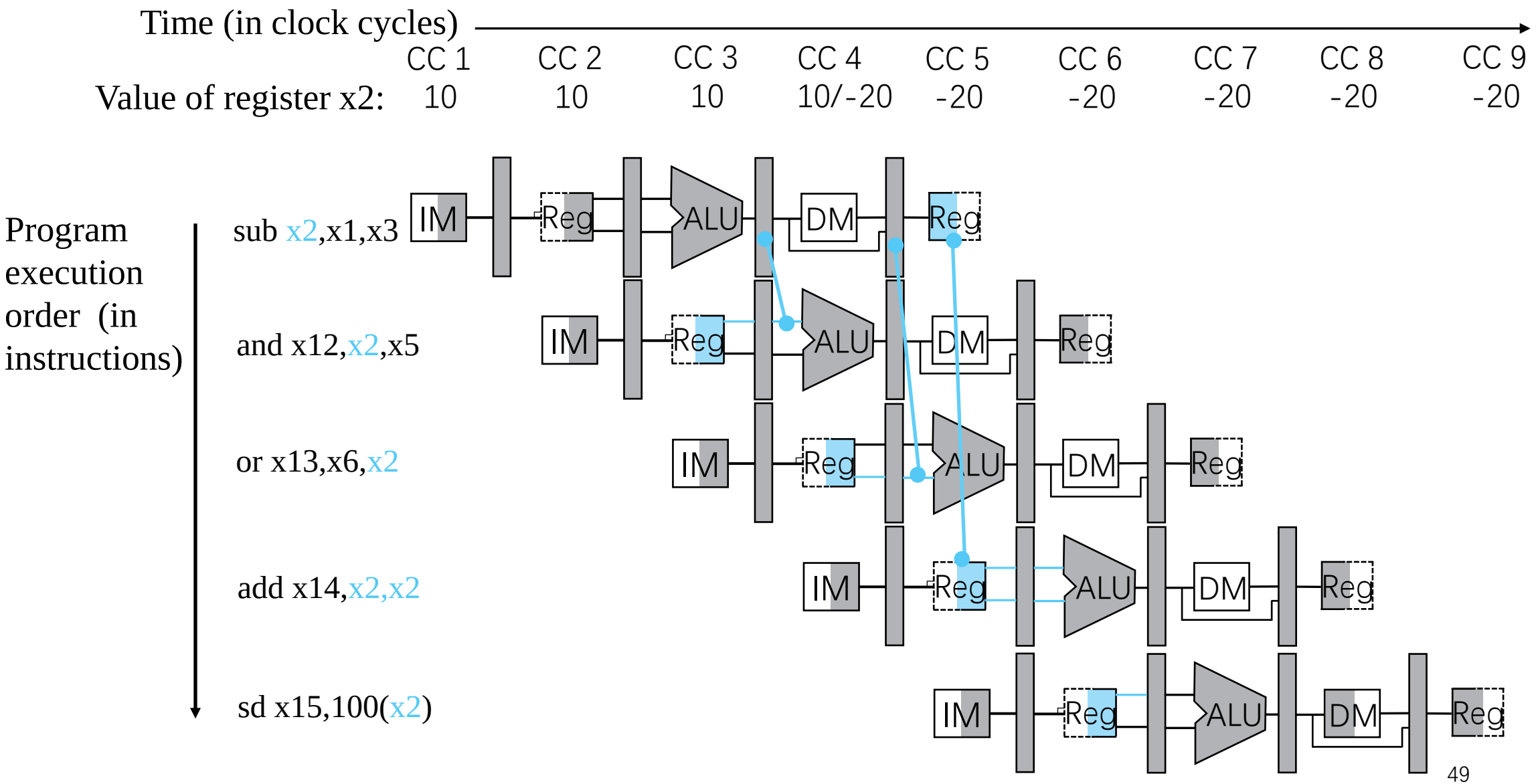
- EX阶段中，ALU操作数寄存器字段名称分别是：
  - ID/EX.RegisterRs1
  - ID/EX.RegisterRs2
- 两组数据冒险：EX冒险和MEM冒险

  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



前递的结果


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/-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

检测前递发生

- 并不是所有指令都会写回寄存器，包括写回寄存器如果是x0也是不需要将数据前递出去，因此需要额外判断：
  - 在**EX**阶段判断（3个条件同时满足）：

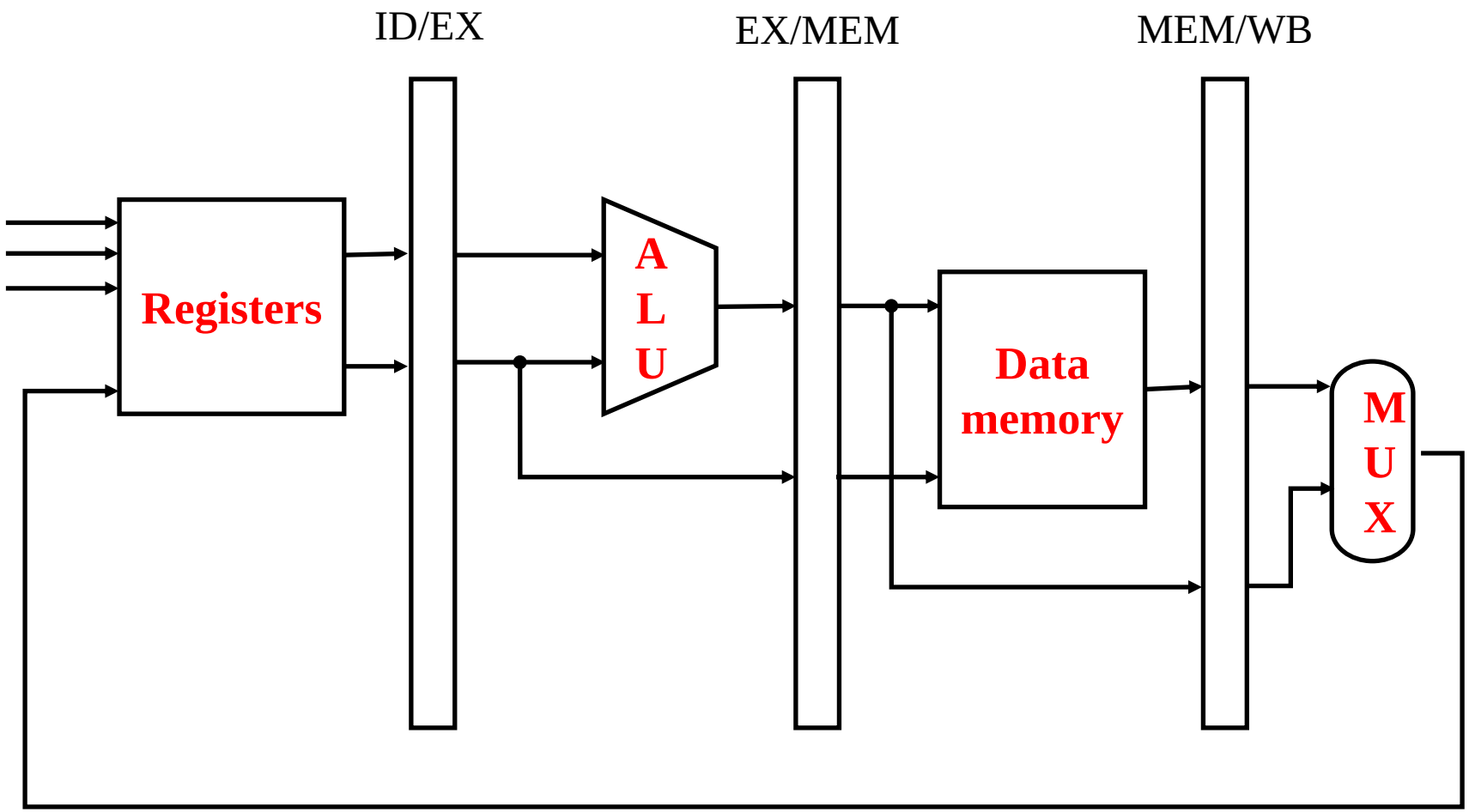
    EX/MEM.RegWrite and（EX/MEM.RegisterRd≠0）and

    EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 在**MEM**阶段判断（3个条件同时满足）：

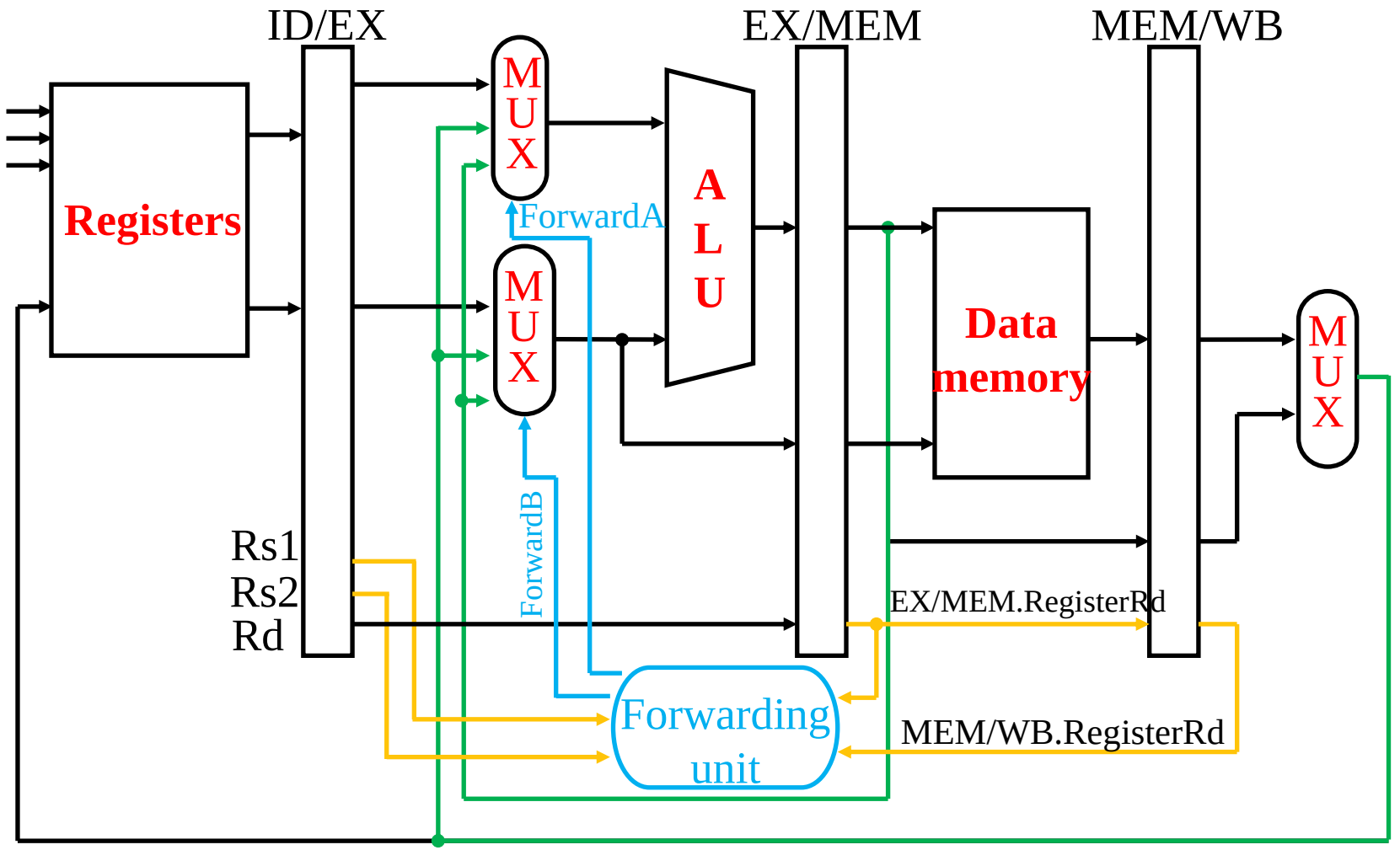
    MEM/WB.RegWrite and（MEM/WB.RegisterRd≠0）and

    MEM/WB.RegisterRd = ID/EX.RegisterRs

添加前递前



添加前递后



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

Mux control	Source	Explanation
ForwardA = 00	ID/EX	ALU的第一个操作数来自寄存器堆
ForwardA = 10	EX/MEM	ALU的第一个操作数来自上一个ALU计算结果的前递
ForwardA = 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
ForwardB = 00	ID/EX	ALU的第二个操作数来自寄存器堆
ForwardB = 10	EX/MEM	ALU的第二个操作数来自上一个ALU计算结果的前递
ForwardB = 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

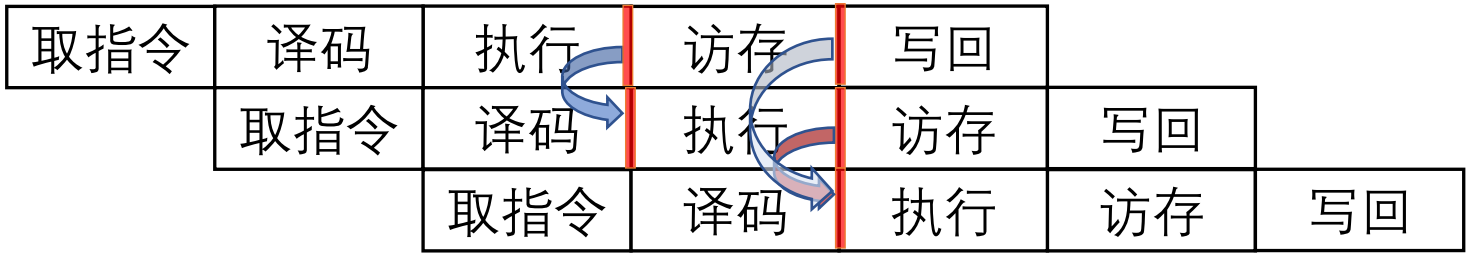
检测EX冒险的条件、相应的前递控制

- EX冒险（同时满足3个条件）
  - if (EX/MEM.RegWrite
    and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
    则执行 ForwardA = 10
  - if (EX/MEM.RegWrite
    and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
    则执行 ForwardB = 10

MEM冒险检测仍有问题：两次数据冒险

例子： 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

- MEM 冒险（同时满足6个条件，其中包括了EX的3个）
  - **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))

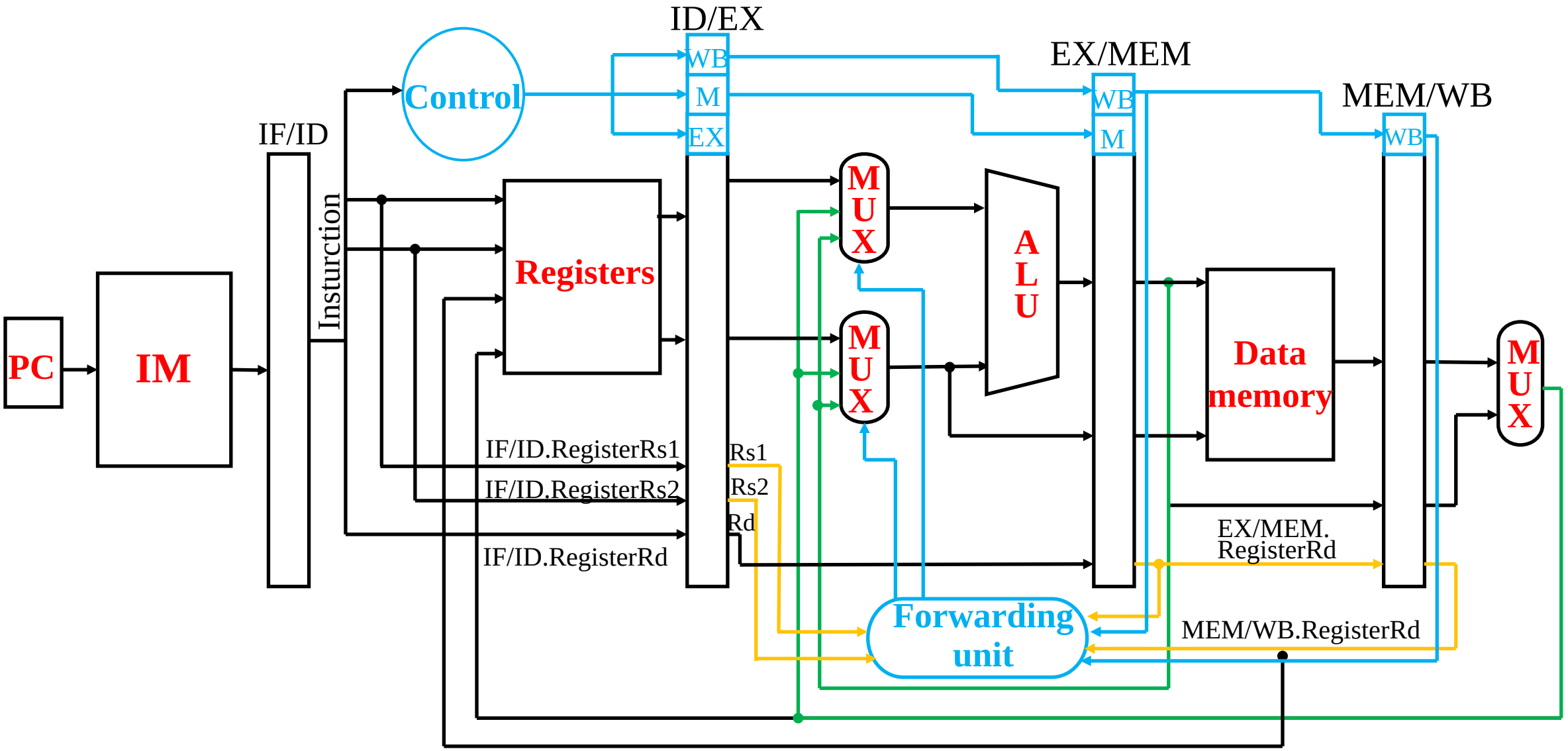
    则执行

    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))

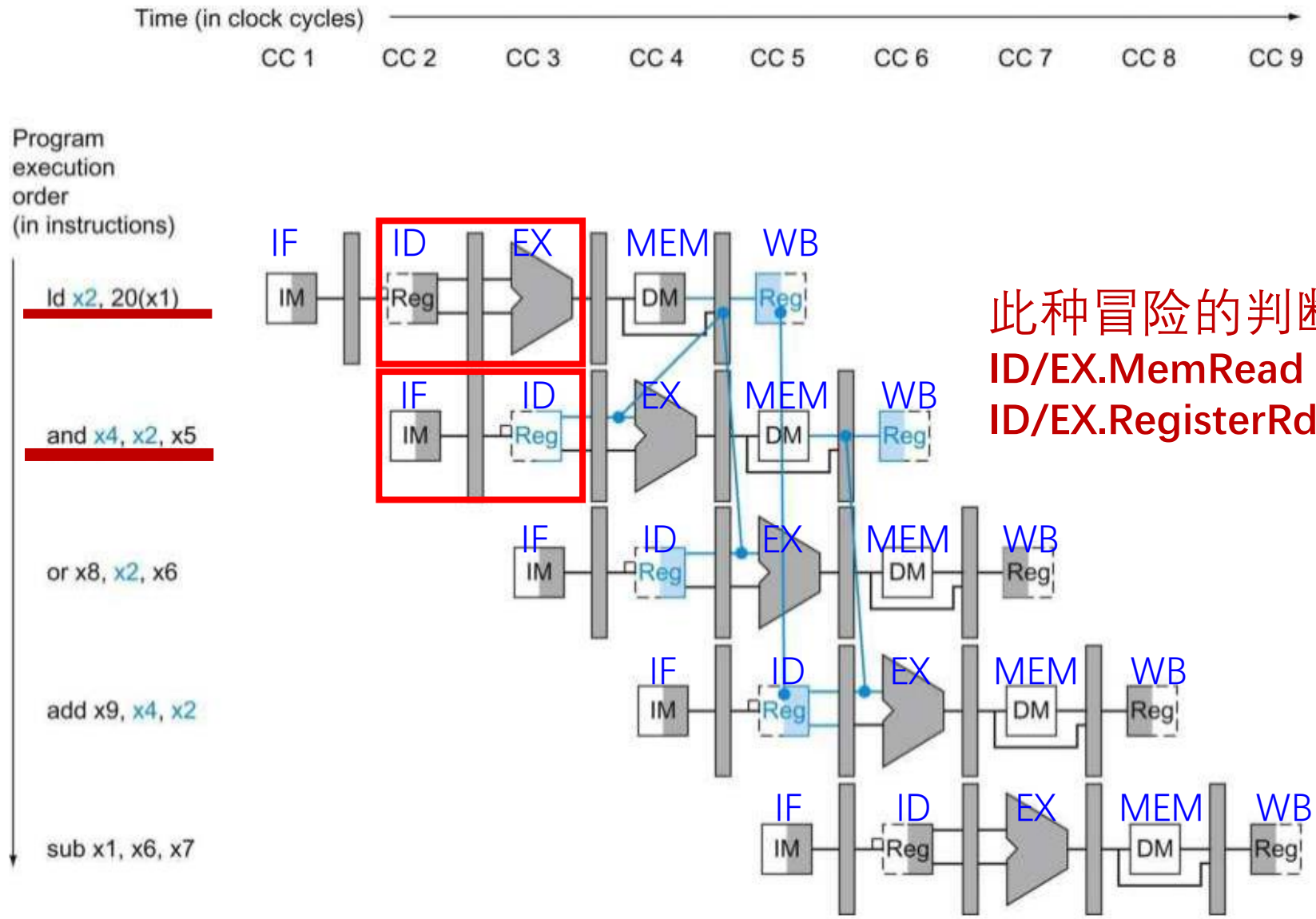
    则执行

    ForwardB = 01

通过前递解决冒险的数据通路



载入-使用型数据冒险(ld+R型)



此种冒险的判断条件:

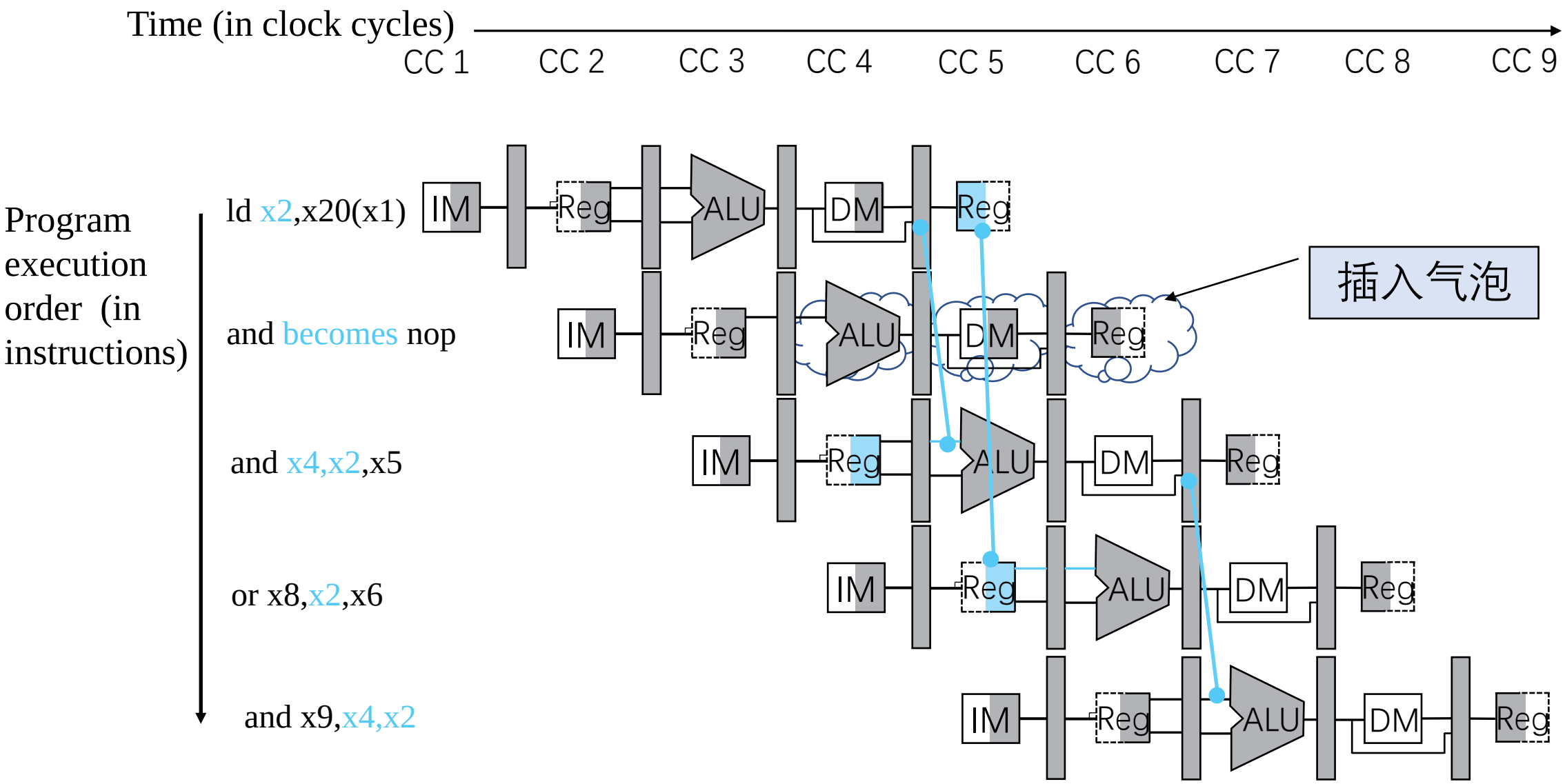
ID/EX.MemRead and
ID/EX.RegisterRd = IF/ID.RegisterRs

检测 载入-使用型数据冒险

- 在load指令的第二阶段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
    ((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）

还有一种数据冒险，同学们可以自行考虑一下：
加载指令紧跟在存储指令之后的情况（sd…, ld…）

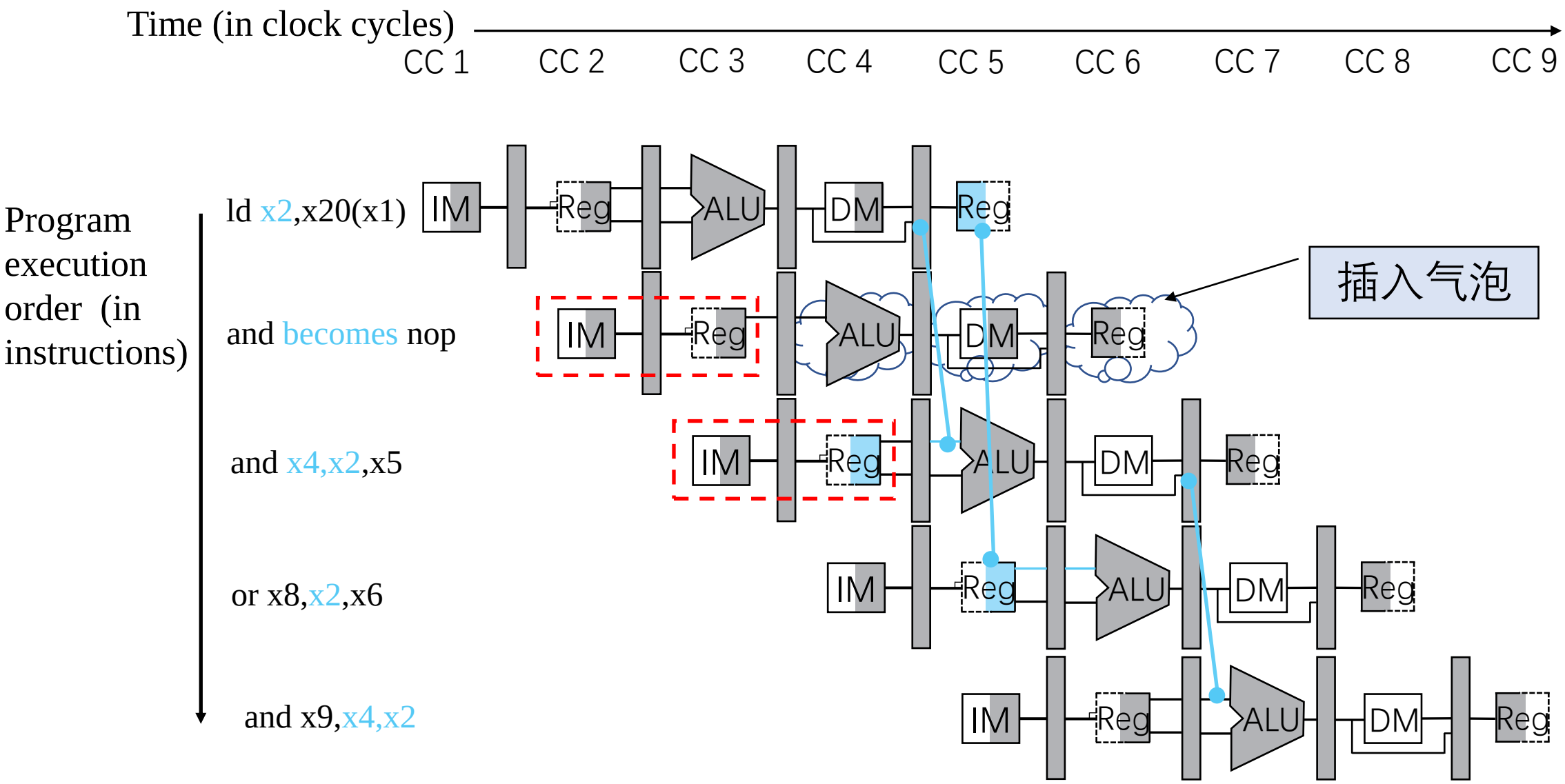
载入–使用型数据冒险

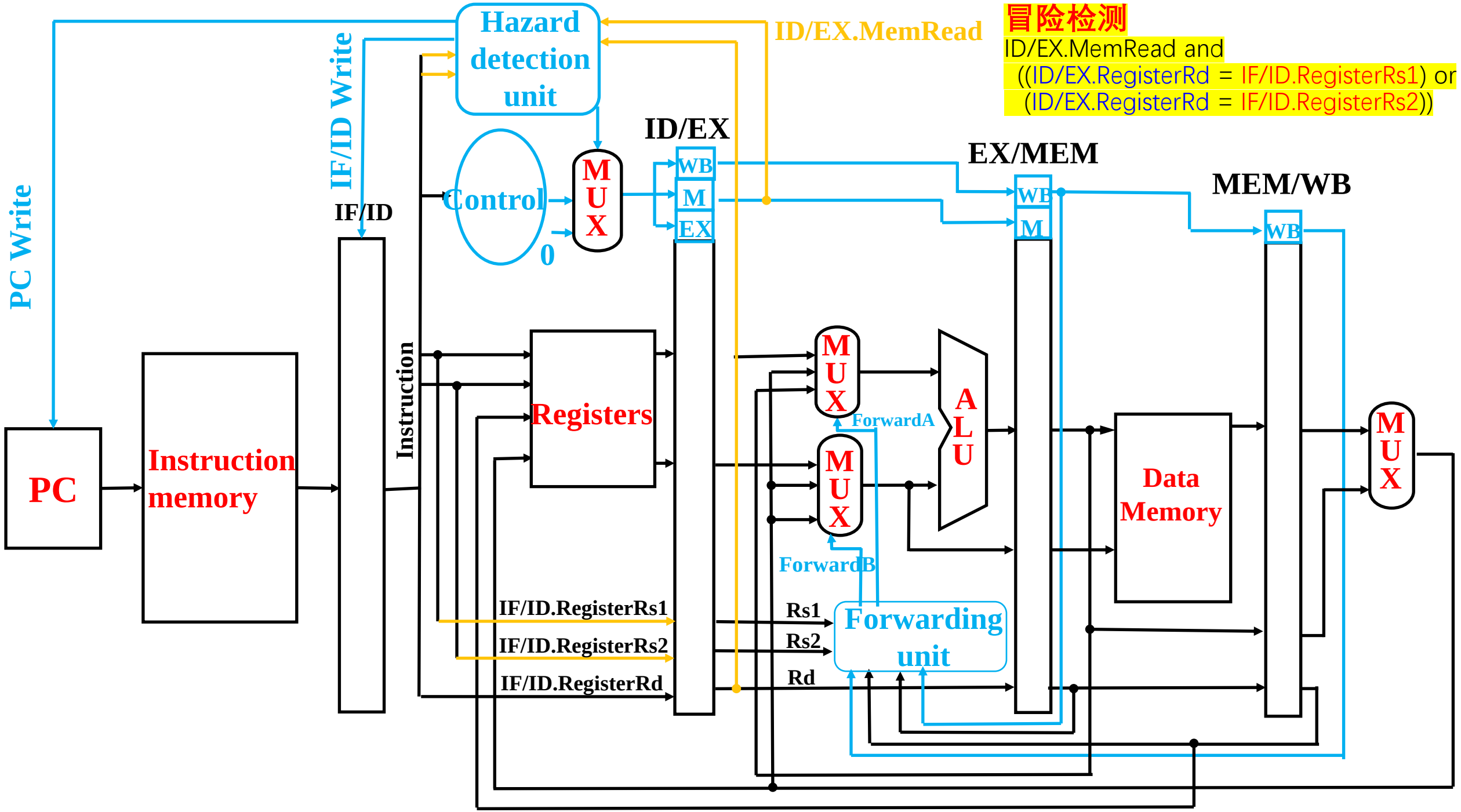

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
ld x2,x20(x1)
and becomes nop
and x4,x2,x5
or x8,x2,x6
and x9,x4,x2
IM
Reg
ALU
DM
Reg
插入气泡

如何实现流水线的停顿

- 此时，被停顿的指令正处于ID阶段
- 将ID/EX寄存器中的控制信号全部置为0
  - EX，MEM，WB 阶段执行空指令 nop
  - 在控制值均为0的情况下，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寄存器中的相同字段读寄存器
  - 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
    - 之后，就可以把取到的数据前递到EX阶段

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
插入气泡


冒险检测
ID/EX.MemRead and
((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
(ID/EX.RegisterRd = IF/ID.RegisterRs2))
Hazard detection unit
ID/EX.MemRead
PC Write
IF/ID Write
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
0
MUX
WB
M
EX
PC
Instruction memory
Instruction
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

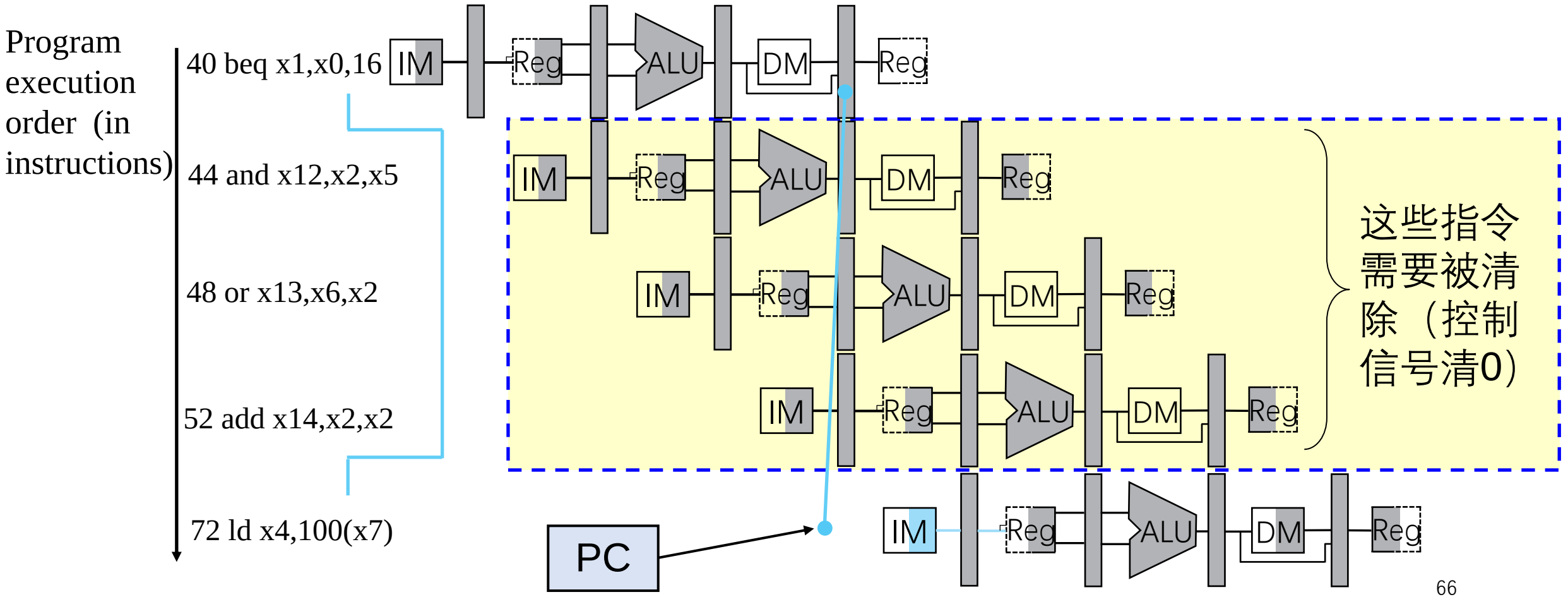
- 停顿会导致性能下降
  - 但为了得到正确的结果，需要停顿
- 编译器能编排代码，尽量避免冒险和停顿
  - 这需要流水线结构的知识

第七章

- 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
- 指令间的并行性

控制冒险

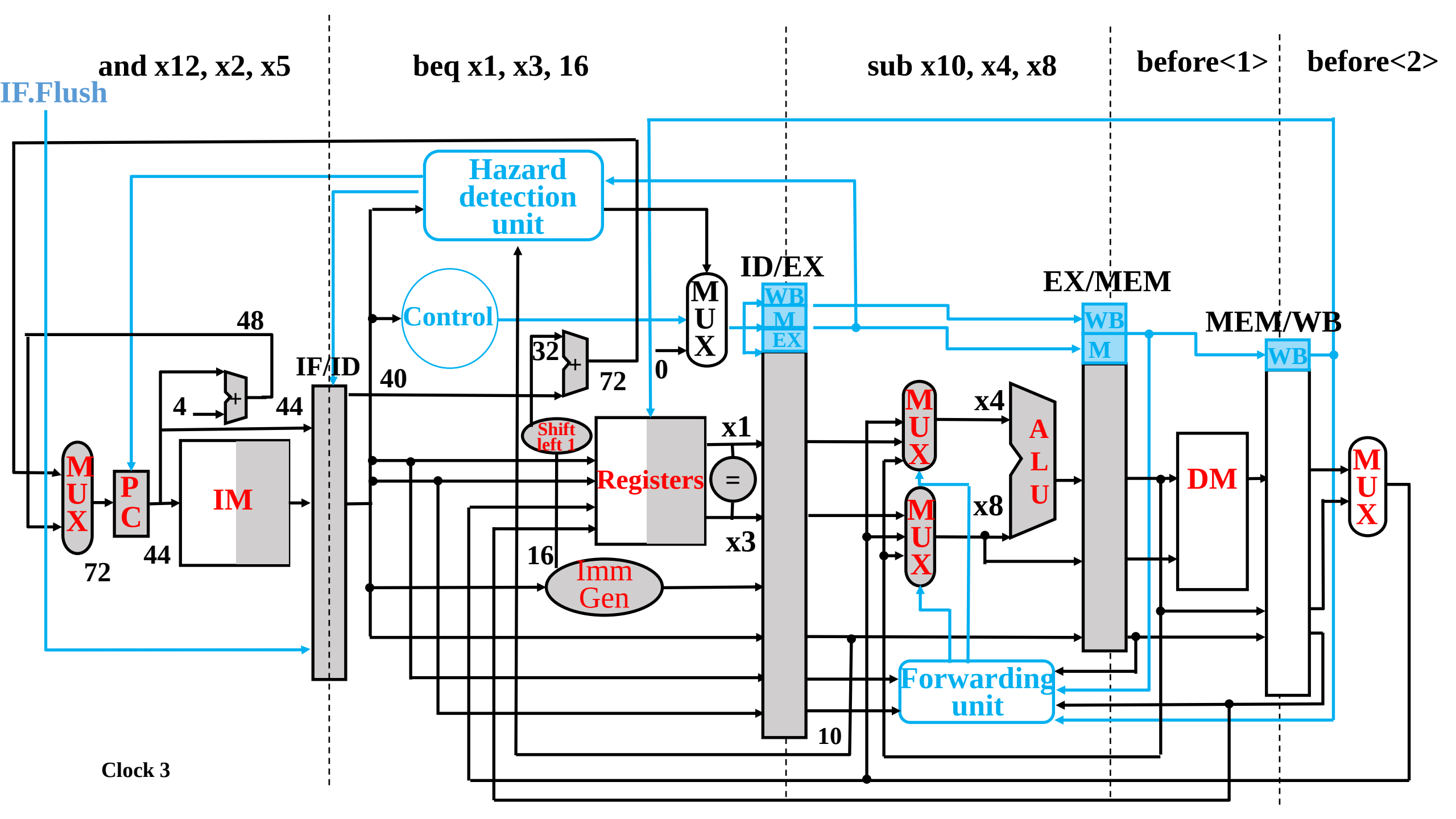
- 假设分支结果在MEM阶段完成确认
  - 分支指令修改PC值发生在MEM阶段

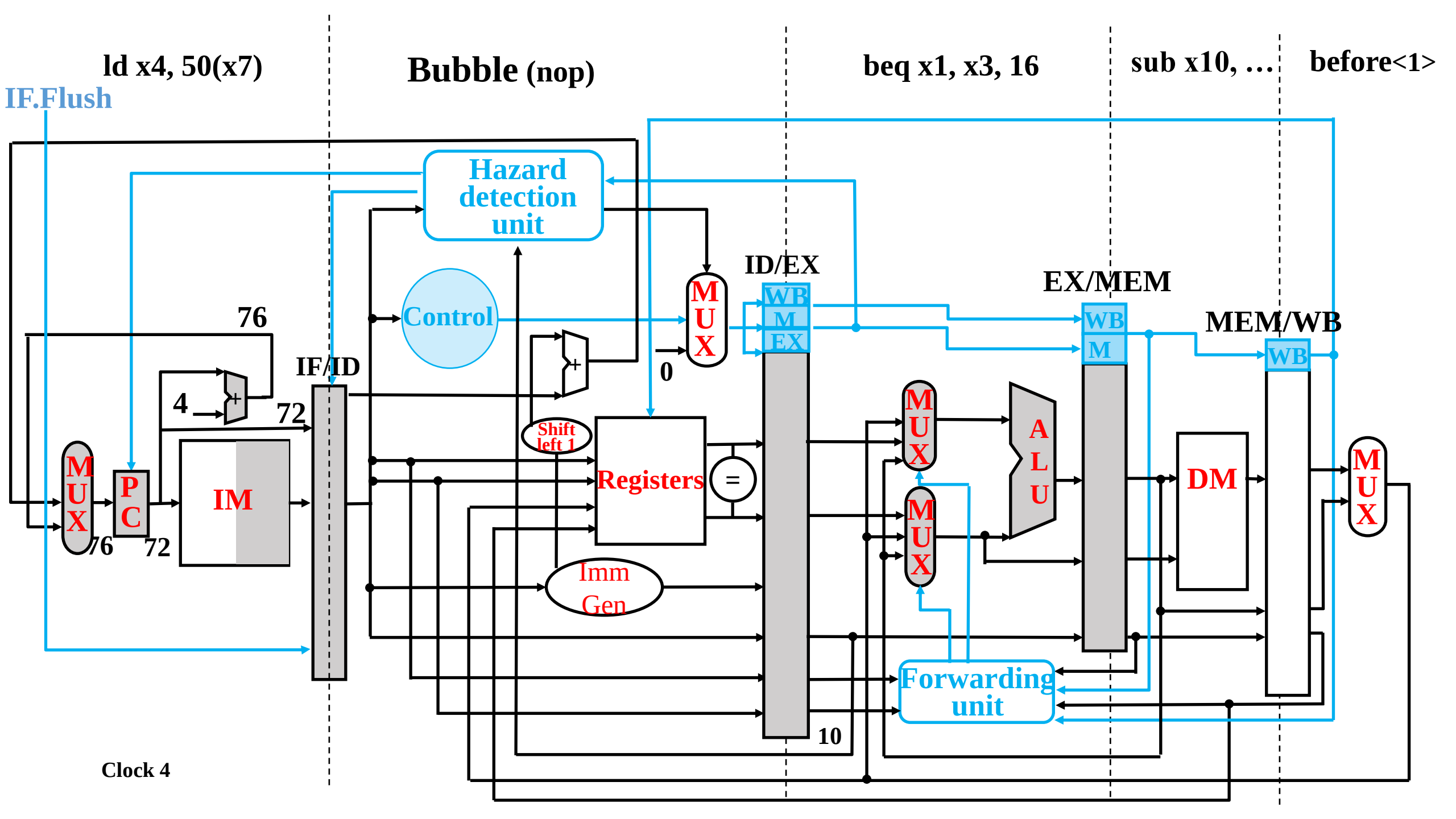


缩短分支延迟

- 增加硬件，将分支决定提前到ID阶段
- 需要两个操作提早发生
  - 计算分支目标地址
  - 判断分支条件
- 举例：分支跳转发生

```
36:   sub   x10, x4, x8
40:   beq   x1,  x3, 16   // PC-relative branch
                          // to 40+16*2=72
44:   and   x12, x2, x5
48:   orr   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, 16
sub x10, x4, x8
before<1>
before<2>
IF.Flush
Hazard detection unit
Control
MUX
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
48
4
+
44
IF/ID
40
32
+
72
0
MUX
PC
IM
72
44
Shift left 1
Registers
x1
=
x3
16
Imm Gen
MUX
x4
MUX
x8
ALU
DM
MUX
10
Forwarding unit
Clock 3


ld x4, 50(x7)
Bubble (nop)
beq x1, x3, 16
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

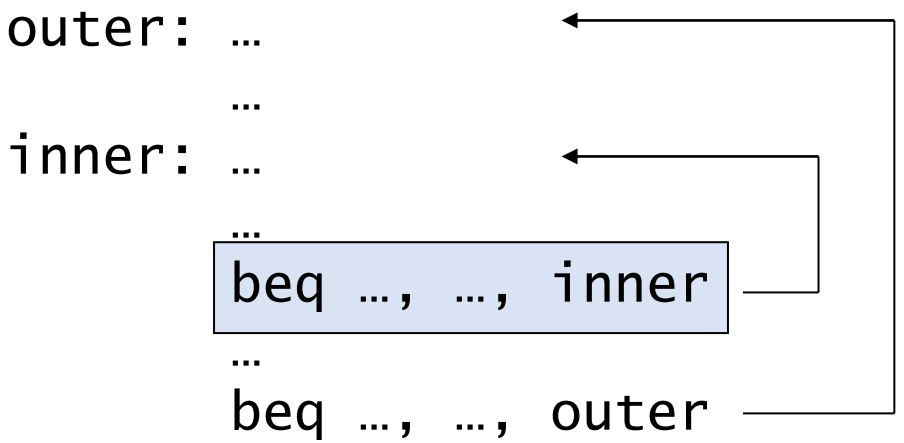
- 对于更深的流水线，从时钟周期数的角度来说，分支预测错误的代价会增大。
- 使用动态分支预测
  - 一种实现方式：采用**分支预测缓存**或**分支历史表**

    指的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

    包含一个或多个比特以表明一个分支最近是否发生了跳转
- 1-Bit预测机制：用1-Bit表示最近是否发生了跳转
  - 预测分支是否发生时
    - 查表，使用表中记录结果作为预测结果
    - 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

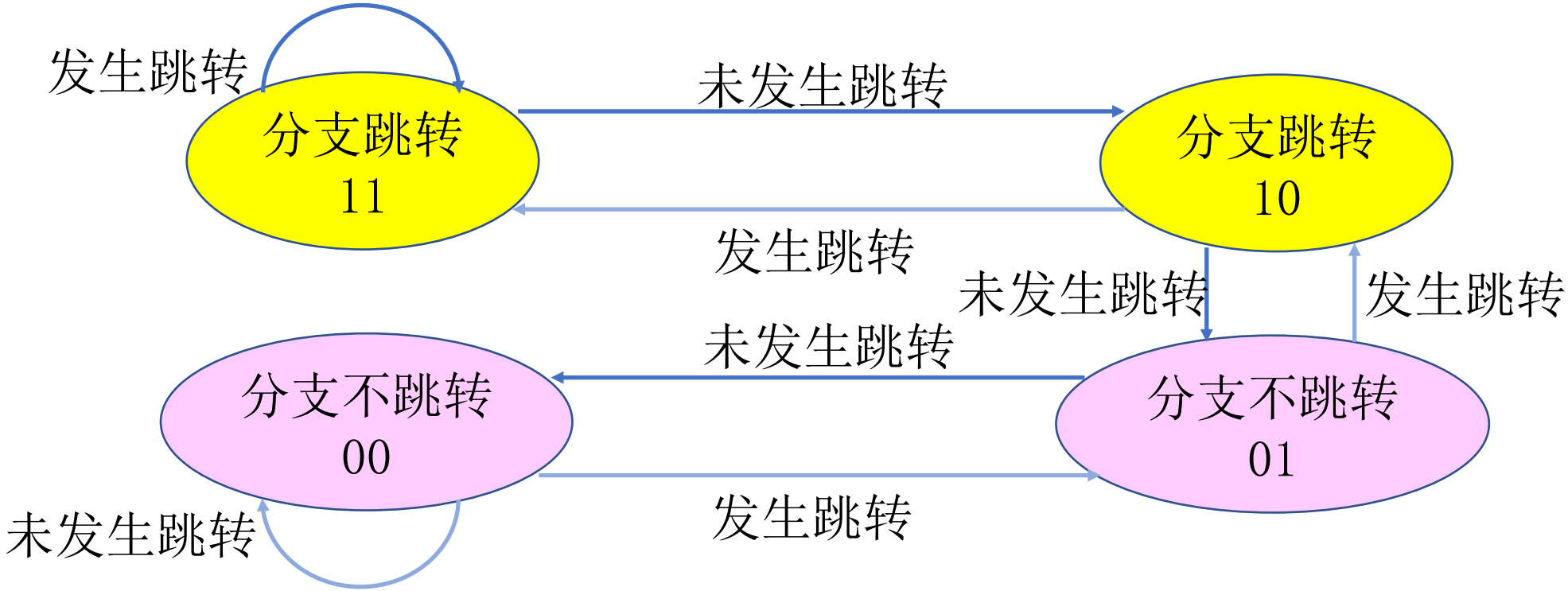
- 一个条件分支总是发生跳转，但一旦其不发生跳转时，会导致两次预测错误，而不是只造成一次错误



  - 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
  - 下一次，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

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



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

解决办法：使用分支目标缓存

- 缓存了成功跳转过的分支目标PC值或目标指令
- 根据当前分支指令的PC值来索引定位目标PC或目标指令
  - 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，立即跳转或使用。

第七章

- 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**
- 指令间的并行性

例外和中断

- 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，它是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，比如： Intel x86使用中断同时指代两者
- 例外(exception)
  - 对于RISC-V来说，指代**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(interrupt)
  - 仅指代由处理器外部事件引发的控制流变化
- 就目前而言，我们只涉及例外的控制逻辑
- 对例外进行时序优化是困难的

举例

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

我们只涉及例外类型的控制逻辑实现

RISC-V体系结构中如何处理例外

- 保存发生例外的指令地址
  - 在RISC-V中，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
- 保存例外发生的原因
  - 在RISC-V中使用的方法是：设置**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。
  - 例如，2被编码为未定义指令，12被编码为硬件故障
- 跳转到统一的入口地址，进行例外处理
  - $0000\ 0000\ 1C09\ 0000_{hex}$

另一种例外处理方式(x86等)

- 采用向量式中断（x86中统称为中断）
  - 例外原因决定后续控制流的起始地址

目标地址= 基址寄存器 + 例外原因（作为偏移量）

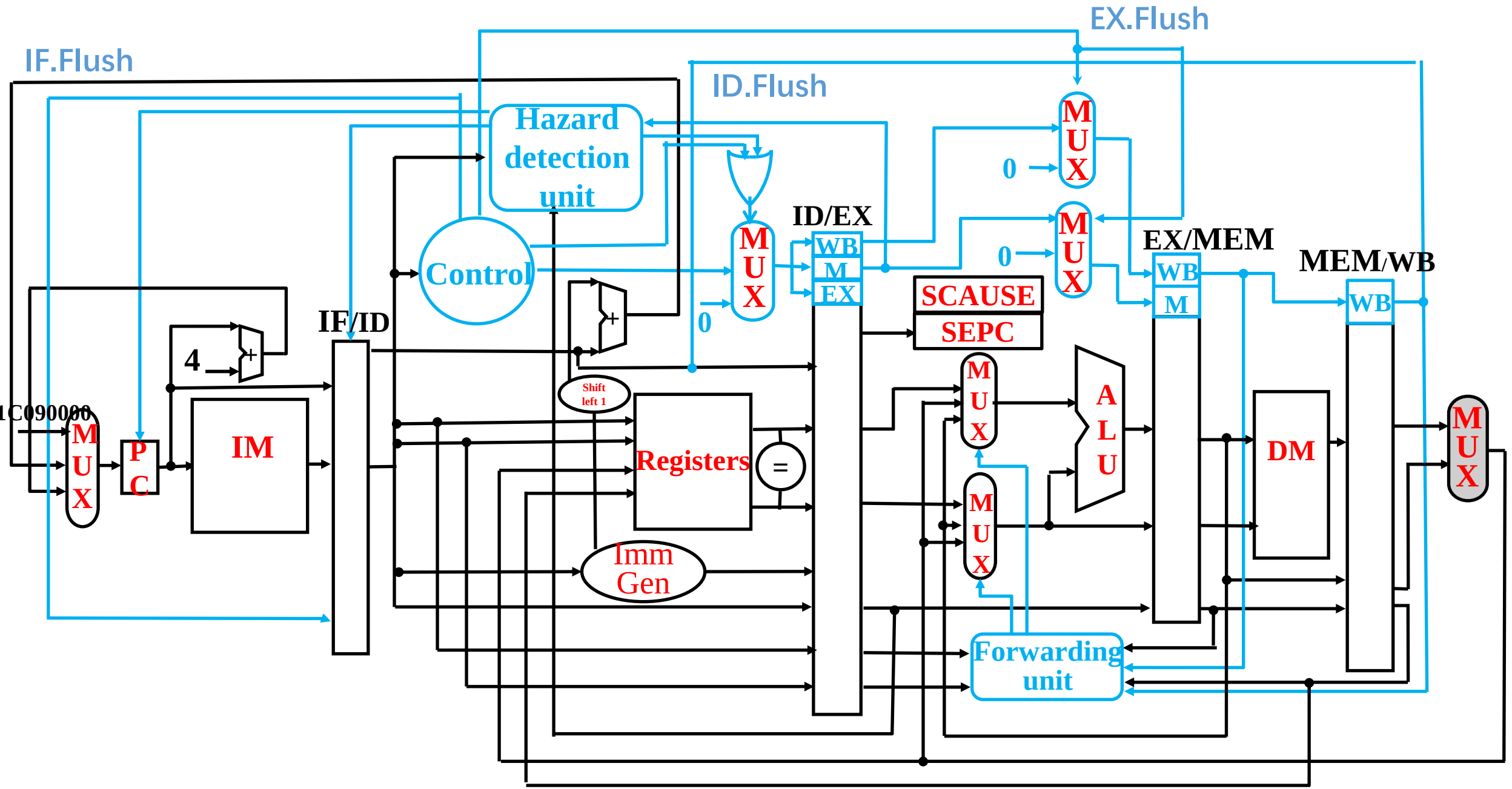
从而，转到例外处理程序

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SEPC寄存器中的内容重启程序从发生位置开始
- 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
- 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：需要保留x1原本的值
  - 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


EX.Flush
IF.Flush
ID.Flush
Hazard detection unit
Control
MUX
ID/EX
WB
M
EX
EX/MEM
MEM/WB
SCAUSE
SEPC
0
4
IF/ID
1C090000
PC
IM
Shift left 1
Registers
=
Imm Gen
ALU
DM
Forwarding unit

例外处理例子

- Exception on add in

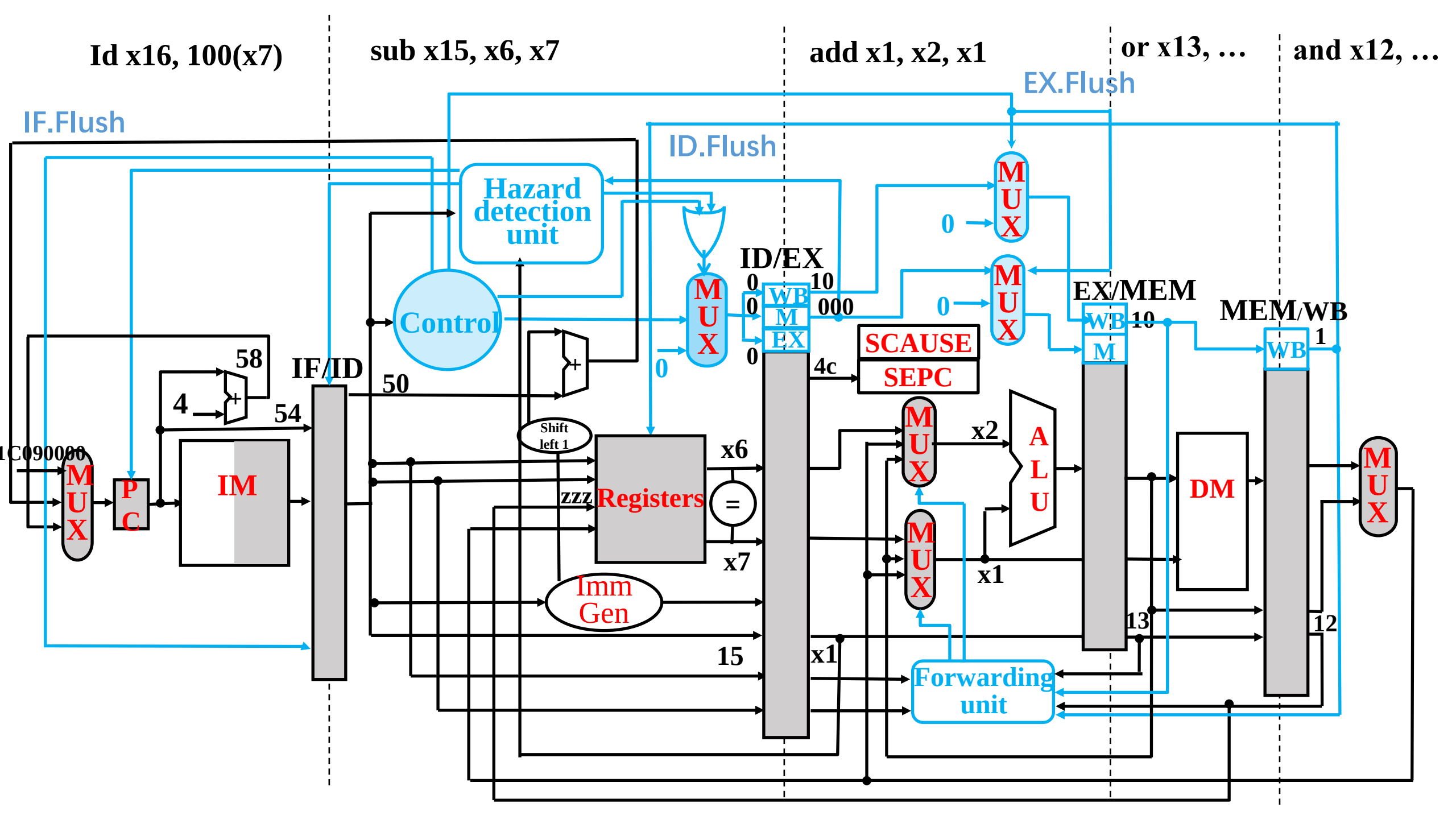
```
40      sub   x11, x2, x4
44      and   x12, x2, x5
48      orr   x13, x2, x6
4c      add   x1,  x2, x1
50      sub   x15, x6, x7
54      ld    x16, 100(x7)
…
```

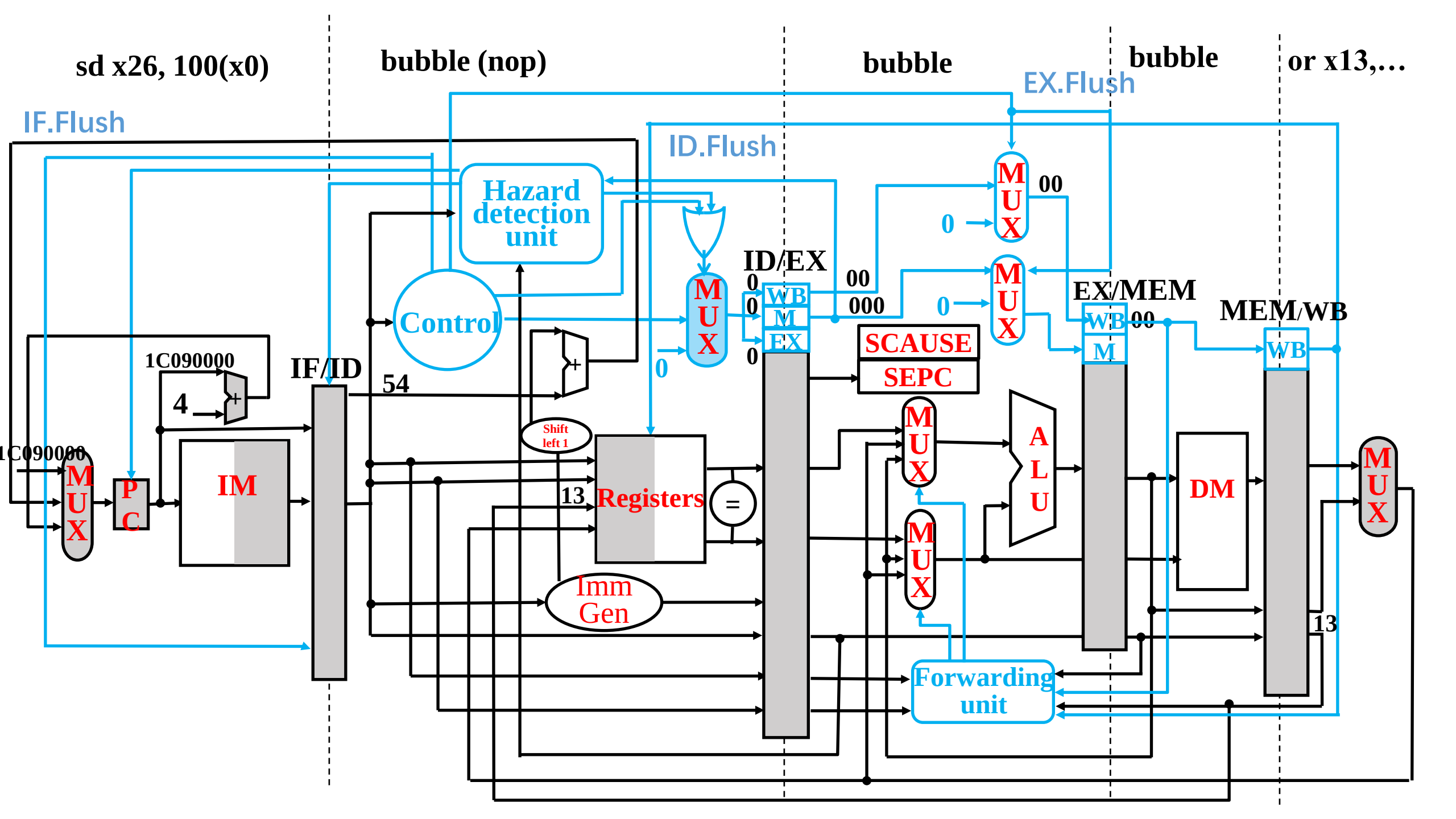
- Handler

```
1C090000    sd   x26, 1000(x10)
1c090004    sd   x27, 1008(x10)
…
```

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	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

FIGURE 4.47 **The values of the control lines are the same as in Figure 4.18, but they have been shuffled into three groups corresponding to the last three pipeline stages.**


Id 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
IF/ID
ID/EX
EX/MEM
MEM/WB
WB
M
EX
SCAUSE
SEPC
4c
58
4
54
50
1C090000
PC
IM
Shift left 1
Registers
zzz
x6
x7
=
Imm Gen
15
x1
x2
ALU
DM
13
12
1
10
000
Forwarding unit
MUX


sd x26, 100(x0)
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
1C090000
4
IF/ID
54
Shift left 1
13
Registers
Imm Gen
IM
PC
ALU
DM
Forwarding unit
00
000
0

第七章

- 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
- **指令间的并行性**

指令级并行(ILP)

- 流水线技术挖掘了指令间潜在的并行性，这种并行性被称为指令级并行
- 提高指令集并行度主要有两种方法
  - **增加流水线级数**
    - 每个阶段更少的工作，时钟周期可以变短
  - **多发射**
    - 增加流水线内部的功能部件数量，一个时钟周期内发射多条指令
    - CPI可以小于 1 ，使用IPC（CPI的倒数）来衡量
    - 例如，4GHz 发射宽度为4的多发射处理器：16 BIPS (Billion Instructions Per Second),最高CPI = 0.25，最高IPC = 4
    - 多发射技术会有一些限制，比如哪些指令可以同时执行、如果发生冒险如何处理等

多发射

- 实现多发射处理器主要有两种方法：编译器实现和硬件实现
- 具体来说，多发射分两类：

1）静态多发射

- 编译器完成指令发射相关判断
- 编译器将指令打包并放入发射槽。
- 编译器处理数据和控制冒险

2）动态多发射（超标量）

- 由硬件（CPU）来判断当前周期可以发射的指令数
- 仍然需要编译器进行指令调度，消除掉指令间的相关，提高指令的发射率
- 处理器在执行过程中使用硬件技术来解决部分或所有类型的冒险

推测

- 推测
  - **编译器或者处理器“猜测”指令的行为，以尽早消除掉指令与其他指令之间的依赖关系**
  - 允许其他与被推测指令相关的指令提早开始执行
  - 必须保证正确性
    - 预测结果正确性的检查机制
    - 预测出错后的恢复机制

例如：1）分支指令结果推测

2）store-load指令序列，可以推测两条指令访存地址不同，允许load先于store执行

在两种（静态和动态）多发射中都可以用到推测

编译器/硬件推测

- 编译器推测
  - 可以调度、编排指令
  - 需要插入额外的指令来检查推测的正确性，并在检测到错误时进行恢复
- 硬件推测
  - 通常会保存推测的结果直到推测被确定是正确的
  - 如果正确，使用保存的推测结果更新寄存器或存储器
  - 如果错误，硬件清除推测结果，并从正确的指令处重新开始执行

推测和例外

- **推测会引入另一个问题**：对某条指令进行推测还可能引入不必要的例外
  - 比如：假设某条load指令处于推测式执行，同时该指令的访存地址发生了越界，则会引发例外。如果推测是错误的，就意味着发生了本不该发生的例外。
- 编译器推测：通过添加特定的ISA支持来避免这样的问题
- 硬件推测：例外将被记录直到确认推测正确，这时被推测的指令完成，检测到例外，转入正常的例外处理程序。

静态多发射

- 静态多发射处理器是由编译器来支持指令打包和处理指令间的冒险
- 编译器将同一周期发射出去的指令集合打包，一般称为**发射指令包**
  - 通常会对同一周期发射的指令类型进行限制
- 将发射指令包看成一条预先定义好、需要进行多种操作的“大指令”
  - 符合**超长指令字**(Very Long Instruction Word)的设计思路

静态多发射代码调度

- 编译器必须消除部分/全部冒险
  - 将代码打包成发射指令包
  - 单个指令包内没有任何相关
  - 指令包之间可能出现相关
  - 插入必要的nop指令

静态多发射的RISC-V处理器

- 假设是双发射处理器（单个周期内发射两条指令）
  - ALU指令或分支指令放在前面
  - load指令或store指令放在后面
  - 如果一条指令无法发射，替换成nop指令
  - 指令地址需要64位边界对齐

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

调度举例

- 针对双发射RISC-V处理器，对以下代码进行调度

```
Loop: ld   x31,0(x20)     // x31=array element
      add  x31,x31,x21    // add scalar in x21
      sd   x31,0(x20)     // store result
      addi x20,x20,-8     // decrement pointer
      blt  x22,x20,Loop   // branch if x22 < x20
```

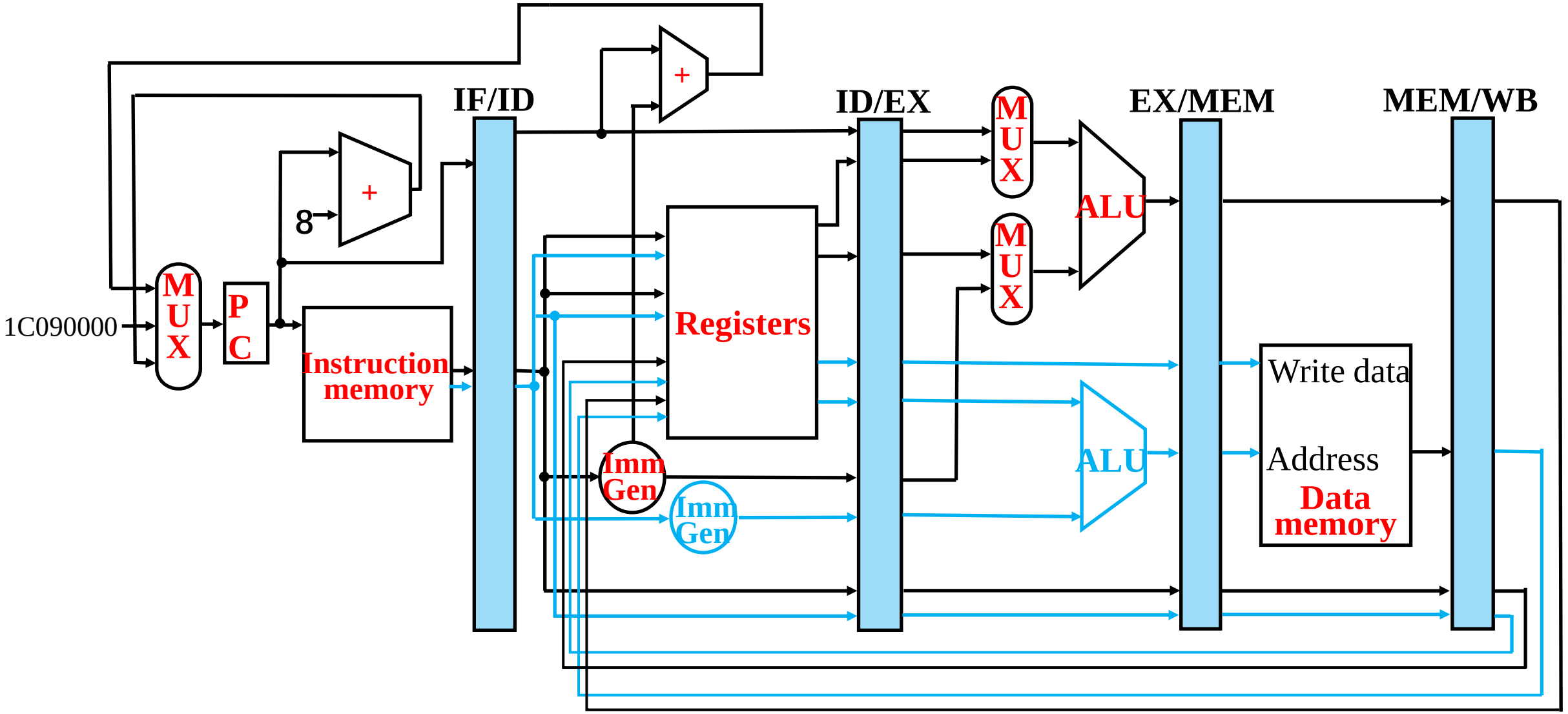
	ALU/branch	Load/store	cycle
Loop:	nop	ld x31,0(x20)	1
	addi x20,x20,-8	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sd x31,8(x20)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

需要增加的硬件资源

- 冒险检测和流水线停顿逻辑
  - 假设在这个例子中，编译器只负责单个指令包中检测所有类型的冒险；硬件来检测两个指令包之间的数据冒险
- 寄存器堆读写口
  - 两条指令需要同时取指和译码
- 加法器
  - ALU部件只负责ALU指令的执行，还需要加法器来进行访存地址的计算

静态双发射数据通路


IF/ID
ID/EX
EX/MEM
MEM/WB
1C090000
MUX
PC
8
+
+
Instruction memory
Registers
Imm Gen
Imm Gen
MUX
MUX
ALU
ALU
Write data
Address
Data memory

案例：双发射RISC-V处理器中的冒险

- 更多的指令在并行执行
- EX数据冒险
  - 在单发射中，通过前递避免停顿
  - 现在，在同一个发射指令包中，load/store指令不能使用ALU指令的结果
    - ```
      add  x10, x0, x1
      ld   x2, 0(x10)
      ```
  - 必须要分成两个包
- 载入-使用型冒险
  - 仍然是一个时钟周期的延迟
- 需要更激进的调度
      ```

调度举例

- 针对双发射RISC-V处理器，对以下代码进行调度

```
Loop: ld   x31,0(x20)     // x31=array element
      add  x31,x31,x21    // add scalar in x21
      sd   x31,0(x20)     // store result
      addi x20,x20,-8     // decrement pointer
      blt  x22,x20,Loop   // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
Loop:	nop	ld x31,0(x20)	1
	addi x20,x20,-8	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sd x31,8(x20)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

动态多发射

- 动态多发射处理器也称为超标量处理器
- 由CPU（硬件）来判断当前周期可以发射的指令数：0，1，2，…

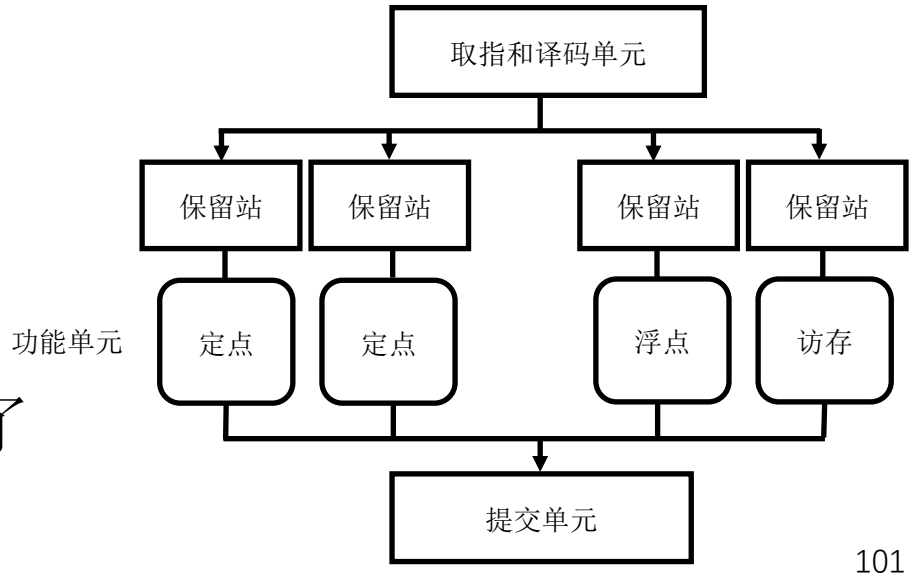
  目的是为了避免结构和数据冒险
- 仍然需要编译器进行指令调度，消除指令间相关，提高发射
  - 编译器指令调度仍有帮助。
  - 编译生成代码的正确性应该与发射率或处理器的流水线结构无关。但是有时代码需要重新编译才能正确运行在不同处理器实现上。
  - 无论调度与否，硬件必须保证代码运行的正确性。

动态调度流水线

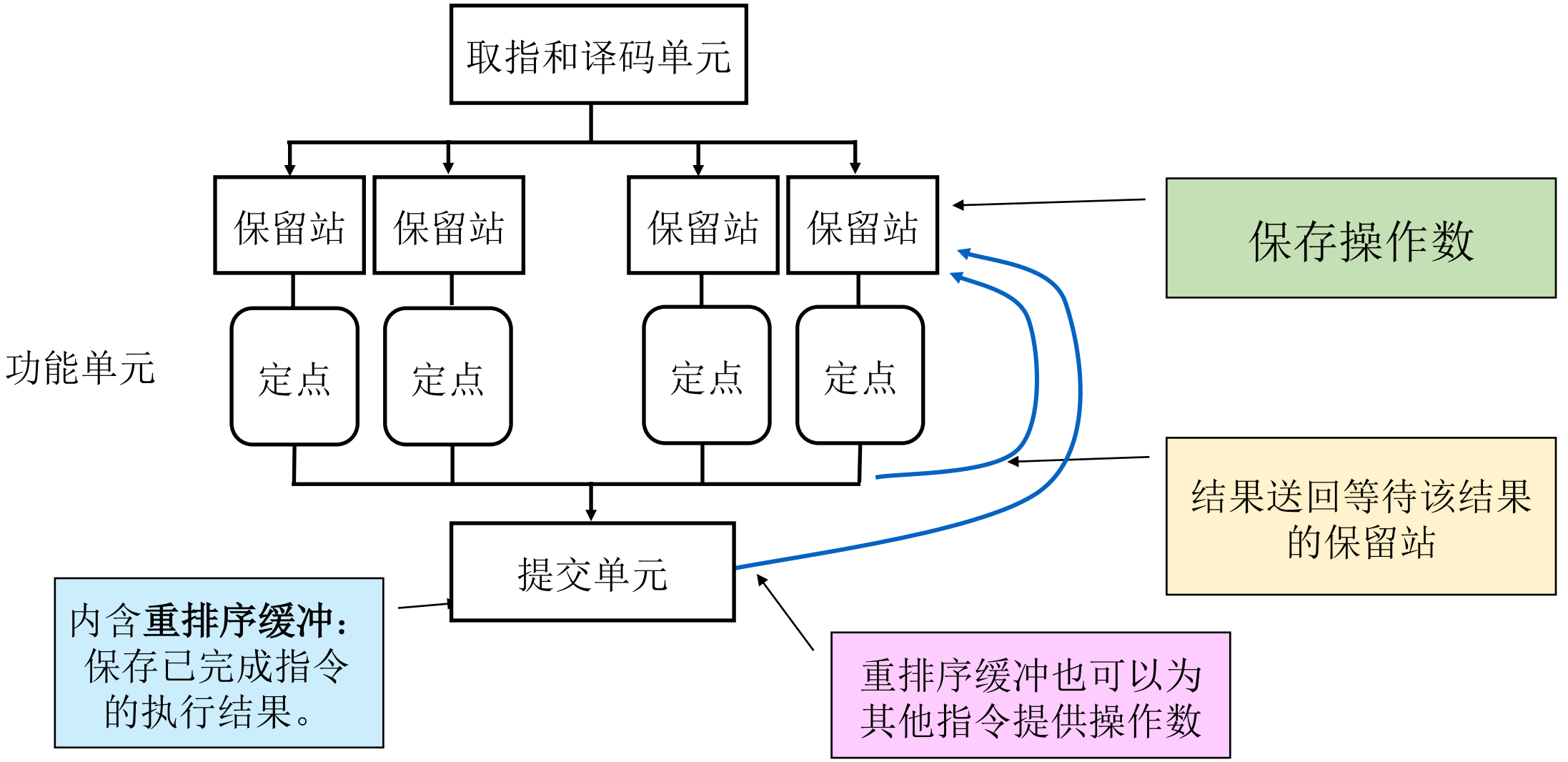
- 流水线被分成三个主要部分：取指和发射单元、多功能部件、提交单元
  - 取指和发射单元：取指令、译码、将指令发送到相应的功能单元上执行
  - 保留站：存放指令的操作和所需的操作数的缓冲区
  - 功能单元：只要保留区的操作数准备好，就可以执行指令。完成后，结果被传送给保留站中正等待该结果的指令，同时也传送到提交单元中保存
  - 提交单元：判定指令何时提交（更新程序员可见的寄存器和存储器）
- 举例

```
ld   x31,20(x21)
add  x1,x31,x2
sub  x23,x23,x3
andi x5,x23,20
```

  - 在add等待ld的时候，sub指令就能执行了

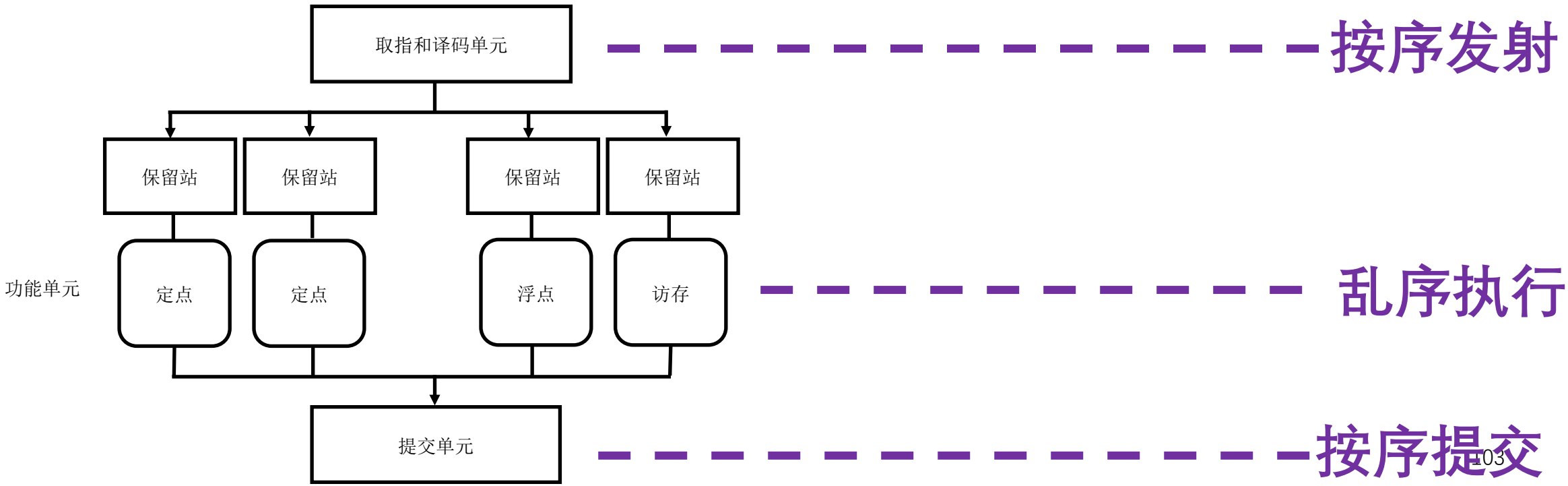


动态调度流水线的结构


取指和译码单元
保留站
保留站
保留站
保留站
功能单元
定点
定点
定点
定点
提交单元
保存操作数
结果送回等待该结果的保留站
内含**重排序缓冲**：保存已完成指令的执行结果。
重排序缓冲也可以为其他指令提供操作数

动态调度流水线的特点

- **乱序执行**：不违背程序原有数据流顺序的前提下以某种顺序执行
  - 功能部件允许乱序执行
- **按序提交**：按照取指的顺序更新程序员可见的处理器状态
  - 取指和译码、提交都是按序执行



为什么超标量处理器需要动态调度技术？

问题：既然编译器也能进行指令调度来解决数据相关，为什么还需要动态调度？

- **首先**，不是所有流水线停顿都是可预测的
  - 比如，存储层次中的缓存失效
- **其次**，如果使用动态分支预测技术，那么在编译程序时是无法知道指令的真是执行顺序
- **最后**，不同的流水线实现具有不同的延迟和发射宽度，这会改变编译代码的最佳配置

问题：多发射有用吗？

答： 有用，但是一直保持高发射率是困难的

- 很少有应用能一直保持两条以上的发射率

原因在于：

1）主要的性能瓶颈在于指令之间的依赖关系

- 这种依赖关系无法消除
- 例如：指针特别容易产生存储器别名，导致潜在的数据相关

2）存储层次中的各级失效使得流水线不能满负荷运转

能效

- 深度挖掘指令级并行能力带来的负面影响：处理器能效降低
- 如今，已经撞上了功耗墙，因此转向设计单芯片多处理器架构
- 从单核设计转向多核设计，流水线技术和功耗都有明显

微处理器	年份	时钟频率	流水线级数	发射宽度	乱序/推测	单芯片核数	功耗
Intel 486	1989	25 MHz	5	1	否	1	5W
Intel Pentium	1993	66 MHz	5	2	否	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	是	1	29W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	是	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	是	1	103W
Intel Core	2006	2930 MHz	14	4	是	2	75W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	是	2～4	87W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	是	8	77W

图 4-70　Intel 系列处理器参数比较，包括流水线复杂度、核数和功耗。其中，奔腾 4 的流水线级数不包括提交阶段。如果把它考虑进来，奔腾 4 的流水级数将会更深