

# 计算机体系结构

胡伟武、汪文祥

# 指令流水线

- 一个简单的MIPS CPU
  - 数据通路
  - 控制逻辑
  - 时序控制
- 指令流水线
- 指令相关和流水线冲突
- 流水线的前递技术
- 流水线和例外
- 多功能部件与多拍操作

# 一个简单的MIPS CPU

# 一个简单的MIPS CPU

- 从MIPS指令集拣选部分代表性的指令，基本特征包括：
  - RISC结构
  - 32位指令及数据
  - 32个通用寄存器，0号GPR恒为0。
  - 指令格式

寄存器型	OP(6)	RS1(5)	RS2(5)	RD(5)	SA(5)	OPX(6)
------	-------	--------	--------	-------	-------	--------

立即数型	OP(6)	RS(5)	RD(5)	Immediate
------	-------	-------	-------	-----------

- 基本指令

	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0			
	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
<b>ADDU</b>	000000	rs	rt	rd	00000	100001																										
<b>SUBU</b>	000000	rs	rt	rd	00000	100010																										
<b>SLT</b>	000000	rs	rt	rd	00000	101010																										
<b>SLTU</b>	000000	rs	rt	rd	00000	101011																										
<b>AND</b>	000000	rs	rt	rd	00000	100100																										
<b>OR</b>	000000	rs	rt	rd	00000	100101																										
<b>XOR</b>	000000	rs	rt	rd	00000	100110																										
<b>NOR</b>	000000	rs	rt	rd	00000	100111																										
<b>SLLV</b>	000000	rs	rt	rd	00000	000100																										
<b>SRLV</b>	000000	rs	rt	rd	00000	000110																										
<b>SRAV</b>	000000	rs	rt	rd	00000	000111																										
<b>ADDIU</b>	001001	rs	rt	immediate																												
<b>LW</b>	100011	base	rt	offset																												
<b>SW</b>	101011	base	rt	offset																												
<b>BEQ</b>	000100	rs	rt	offset																												
<b>BNE</b>	000101	rs	rt	offset																												
<b>BLEZ</b>	000110	rs	00000	offset																												
<b>BGTZ</b>	000111	rs	00000	offset																												

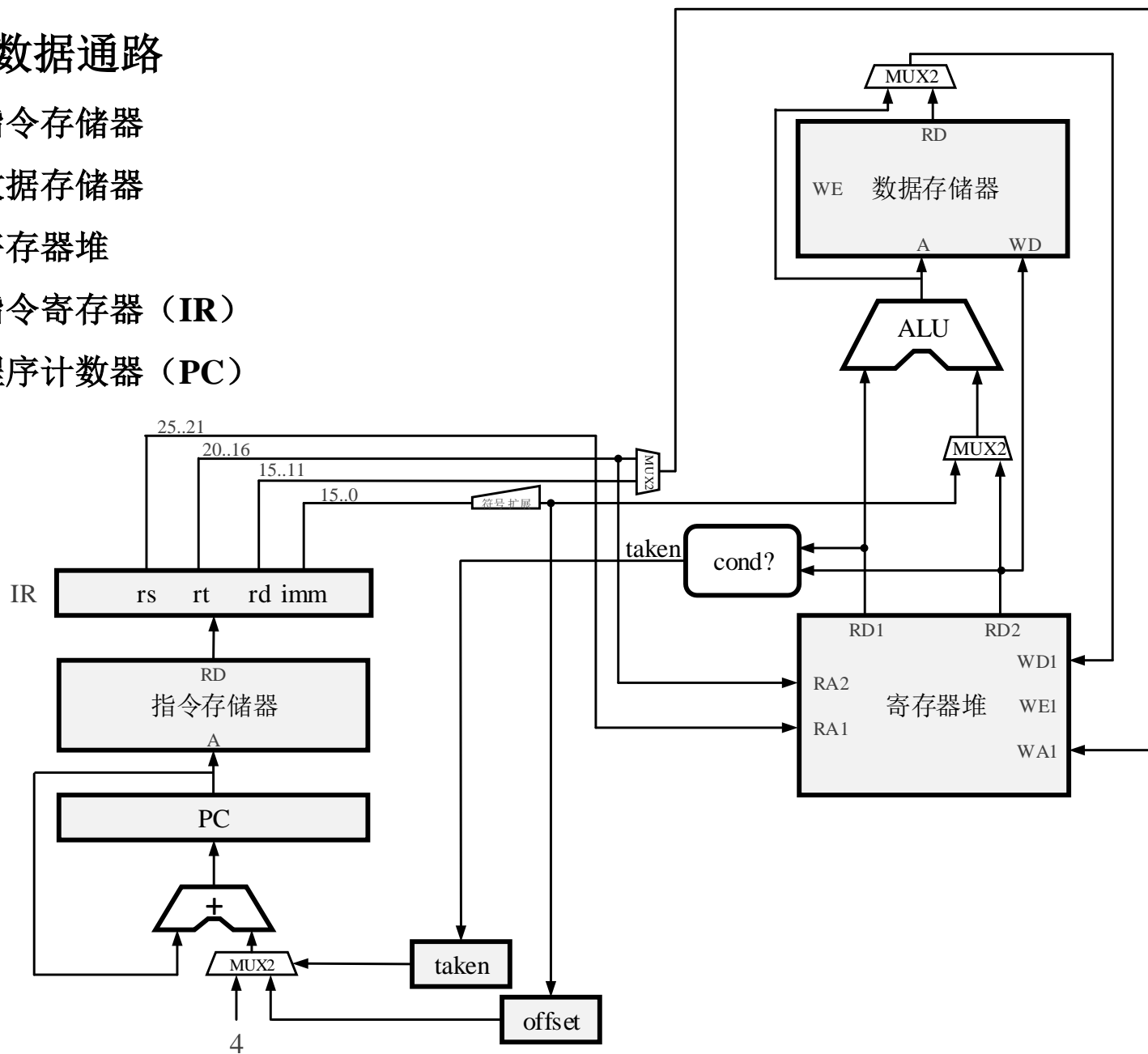
ALU

访存

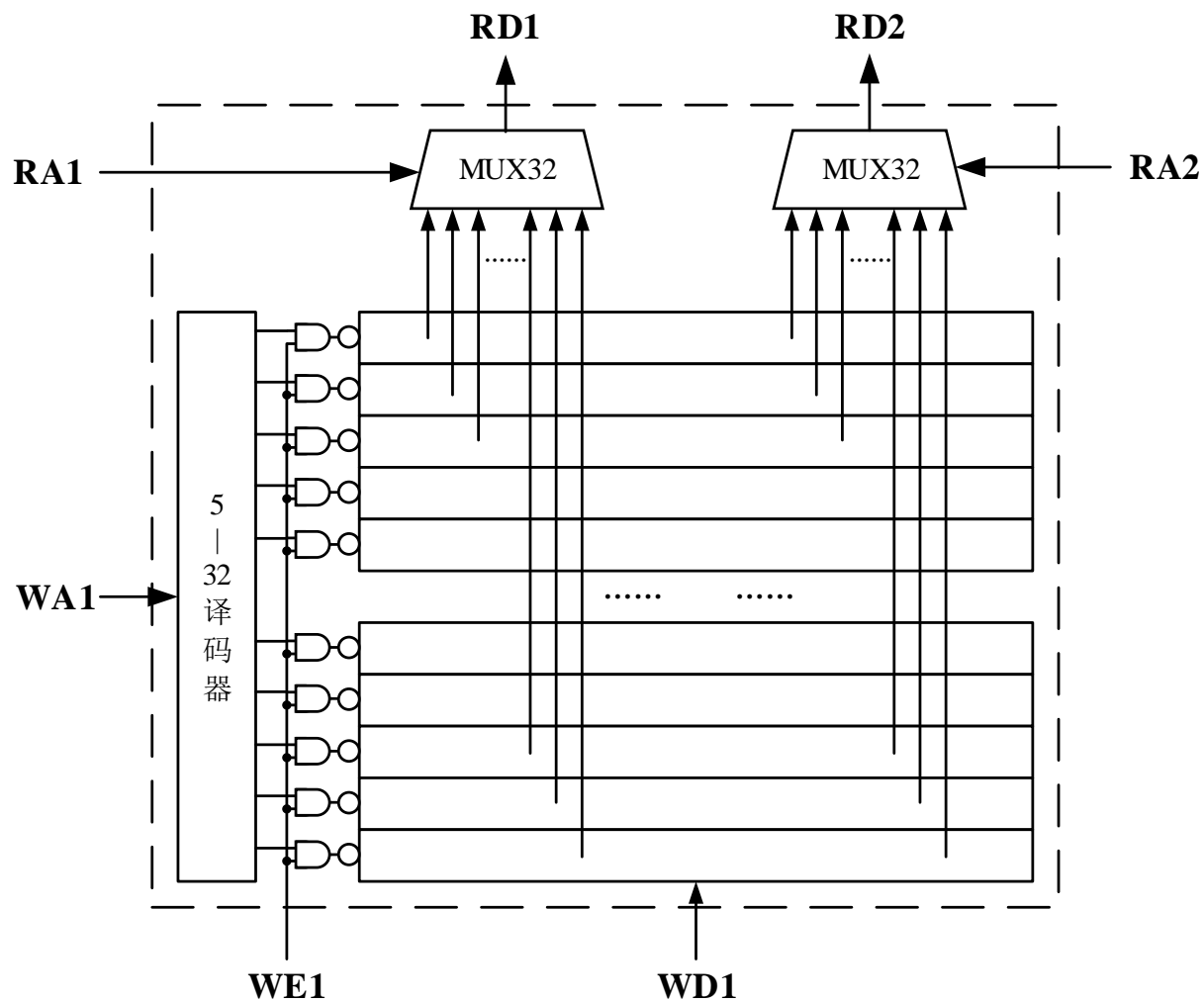
转移

## • 主要数据通路

- 指令存储器
- 数据存储器
- 寄存器堆
- 指令寄存器 (IR)
- 程序计数器 (PC)



- “两读一写”寄存器堆电路结构



- 主要控制逻辑——控制信号
  - C1 : PC计算源操作数2的二选一
  - C2 : ALU源操作数2的二选一
  - C3 : 寄存器写回结果的二选一
  - C4 : 目的寄存器号的二选一
  - C5 : 寄存器堆的写使能
  - C6 : 数据存储器的写使能
  - ALUOp: ALU控制



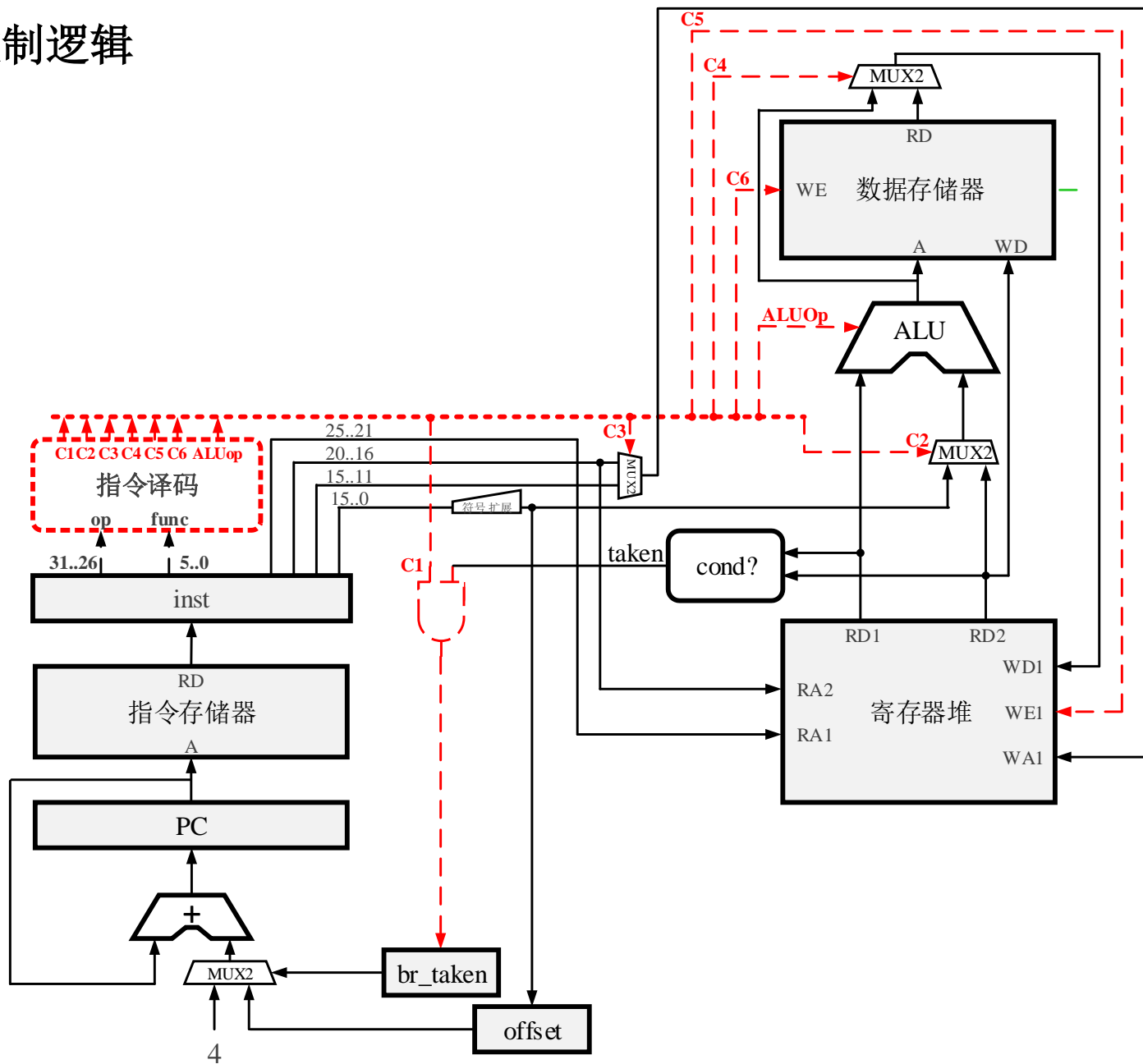
- 主要控制逻辑---真值表

指令	op域	func域	C1	C2	C3	C4	C5	C6	ALUOp
<b>ADDU</b>	000000	100001	0	0	0	0	1	0	0000
<b>SUBU</b>	000000	100010	0	0	0	0	1	0	0001
<b>SLT</b>	000000	101010	0	0	0	0	1	0	0010
<b>SLTU</b>	000000	101011	0	0	0	0	1	0	0011
<b>AND</b>	000000	100100	0	0	0	0	1	0	0100
<b>OR</b>	000000	100101	0	0	0	0	1	0	0101
<b>XOR</b>	000000	100110	0	0	0	0	1	0	0110
<b>NOR</b>	000000	100111	0	0	0	0	1	0	0111
<b>SLLV</b>	000000	000100	0	0	0	0	1	0	1000
<b>SRLV</b>	000000	000110	0	0	0	0	1	0	1010
<b>SRAV</b>	000000	000111	0	0	0	0	1	0	1011
<b>ADDIU</b>	001001	xxxxxxx	0	1	1	0	1	0	0000
<b>LW</b>	100011	xxxxxxx	0	1	1	1	1	0	0000
<b>SW</b>	101011	xxxxxxx	0	1	0	X	0	1	0000
<b>BEQ</b>	000100	xxxxxxx	1	X	0	X	0	0	XXXX
<b>BNE</b>	000101	xxxxxxx	1	X	0	X	0	0	XXXX
<b>BLEZ</b>	000110	xxxxxxx	1	X	0	X	0	0	XXXX
<b>BGTZ</b>	000111	xxxxxxx	1	X	0	X	0	0	XXXX

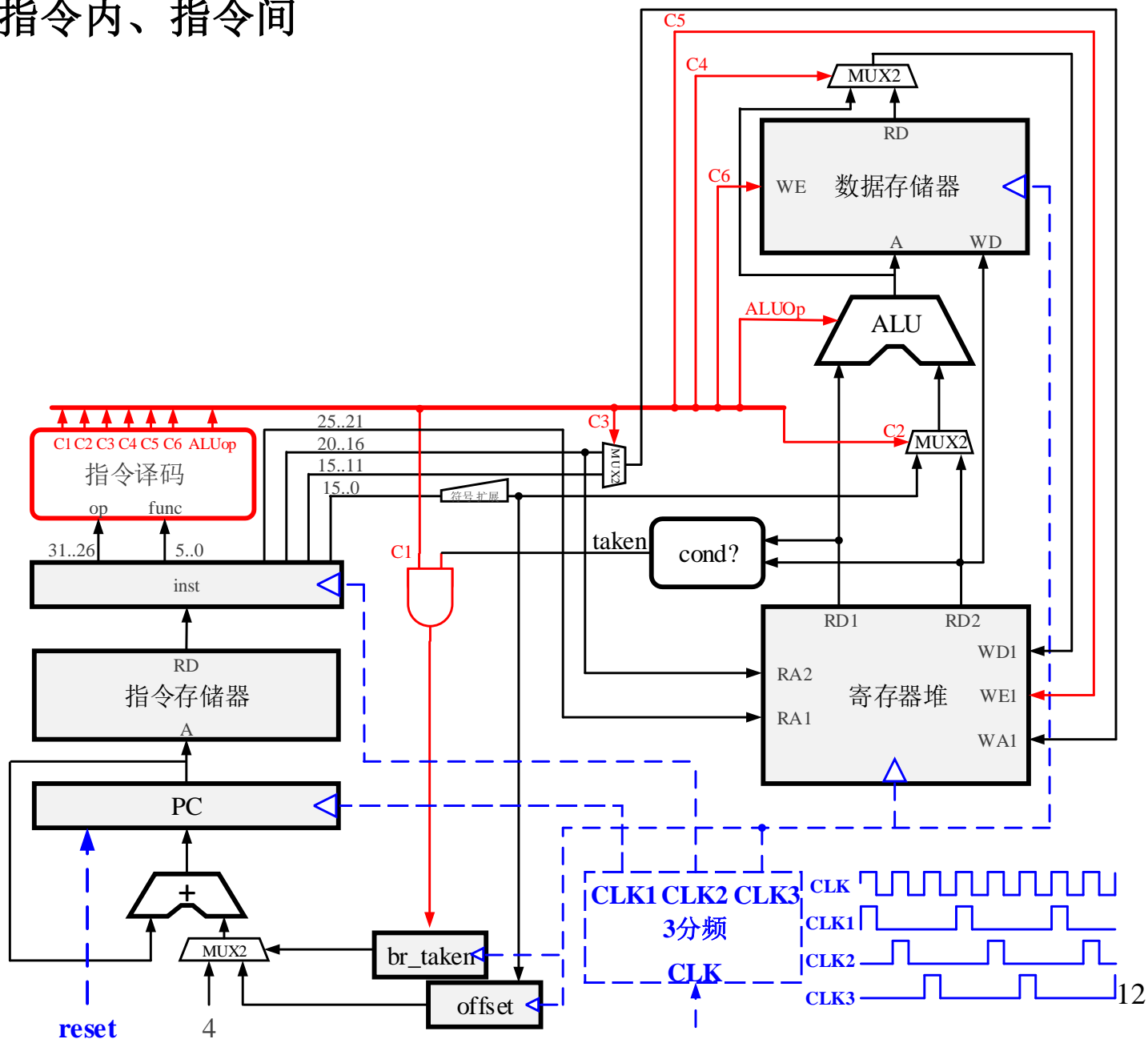
- 控制信号的  
Verilog描述

```
wire [5:0] op      = inst[31:26];
wire [5:0] func    = inst[5:0];
wire inst_addu     = (op==6'b0) && (func==6'b100001);
wire inst_subu     = (op==6'b0) && (func==6'b100010);
wire inst_slt      = (op==6'b0) && (func==6'b101010);
wire inst_sltu     = (op==6'b0) && (func==6'b101011);
wire inst_and      = (op==6'b0) && (func==6'b100100);
wire inst_or       = (op==6'b0) && (func==6'b100101);
wire inst_xor      = (op==6'b0) && (func==6'b100110);
wire inst_nor      = (op==6'b0) && (func==6'b100111);
wire inst_sllv     = (op==6'b0) && (func==6'b000100);
wire inst_srlv     = (op==6'b0) && (func==6'b000110);
wire inst_srav     = (op==6'b0) && (func==6'b000111);
wire inst_addiu    = op==6'b001001;
wire inst_lw       = op==6'b100011;
wire inst_sw       = op==6'b101011;
wire inst_beq      = op==6'b000100;
wire inst_bne      = op==6'b000101;
wire inst_blez     = op==6'b000110;
wire inst_bgtz     = op==6'b000111;
wire c1            = inst_beq  | inst_bne
                    | inst_blez | inst_bgtz;
wire c2            = inst_addiu | inst_lw | inst_sw;
wire c3            = inst_addiu | inst_lw;
wire c4            = inst_lw;
wire c5            = ~(inst_sw | c1);
wire c6            = inst_sw;
wire [3:0] aluop;
assign aluop[0]    = inst_subu | inst_sltu | inst_or
                    | inst_nor | inst_srav;
assign aluop[1]    = inst_slt | inst_sltu | inst_xor
                    | inst_nor | inst_srlv | inst_srav;
assign aluop[2]    = inst_and | inst_or | inst_xor | inst_nor;
assign aluop[3]    = inst_sllv | inst_srlv | inst_srav;
```

- 主要控制逻辑



- 时序：指令内、指令间



# 指令流水线

# 时序的改进

- 上述时序的三个步骤

- 指令地址送到PC、取指到IR、计算结果到GPR

第 1 条    

计算 PC	取指	执行
-------	----	----

第 2 条    

计算 PC	取指	执行
-------	----	----

第 3 条    

计算 PC	取指	执行
-------	----	----

- 可以合并为两个

- 计算下一条指令的PC和指令执行重叠
- 可以把计算下一拍PC值作为指令执行的一部分（转移指令的运算结果是PC的值）

第 1 条    

计算 PC	取指	执行
-------	----	----

第 2 条    

计算 PC	取指	执行
-------	----	----

第 3 条    

计算 PC	取指	执行
-------	----	----

第 1 条    

取指	执行
----	----

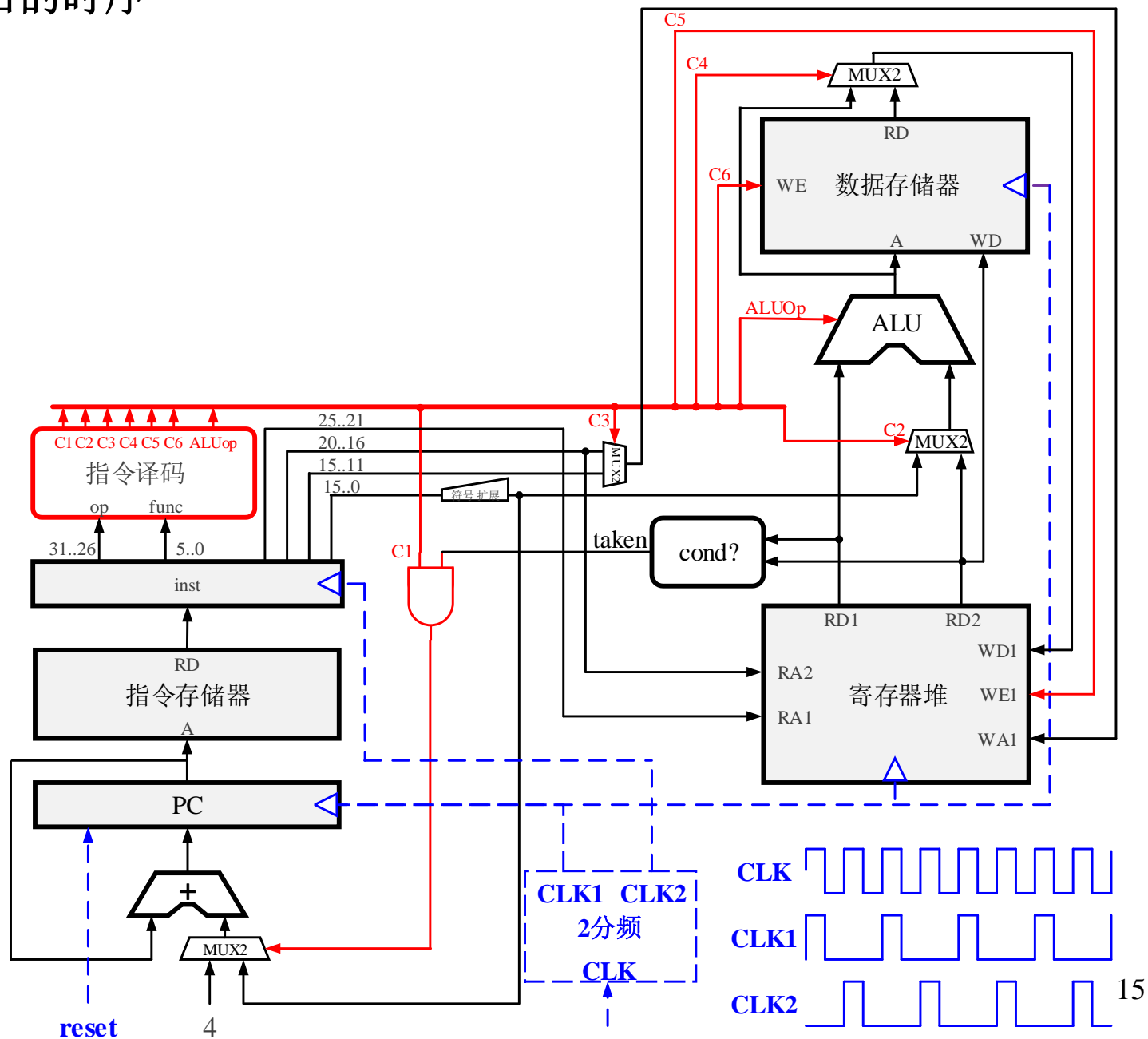
第 1 条    

取指	执行
----	----

第 1 条    

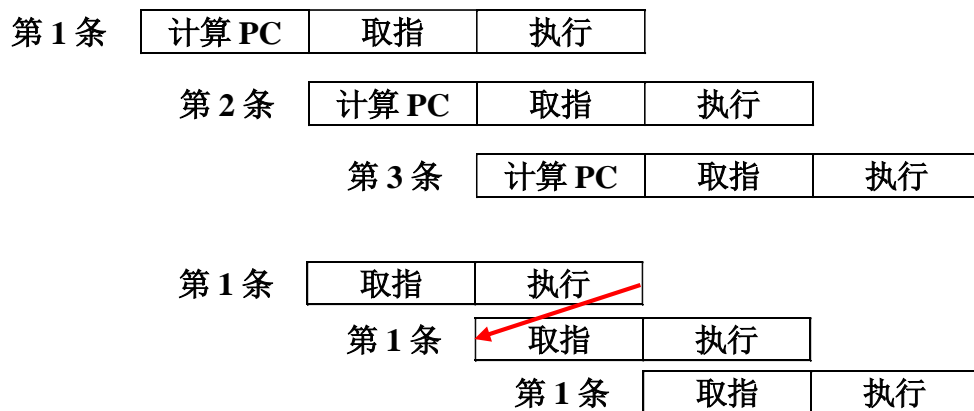
取指	执行
----	----

- 改进后的时序



# 能否把取指和运算也重叠？

- 大多数情况可以重叠
  - 第 $n+1$ 条指令执行时，第 $n$ 条指令已经执行完，因此第 $n+1$ 条指令可以用到第 $n$ 条指令的结果。
- 但有一个例外
  - 如果第 $n+1$ 条指令的取指也要用到第 $n$ 条指令的结果，则第 $n+1$ 条指令的取指必须等到第 $n$ 条指令结束后才能执行。
  - 这正是转移指令的情况，可以用延迟槽（delay slot）解决这个问题





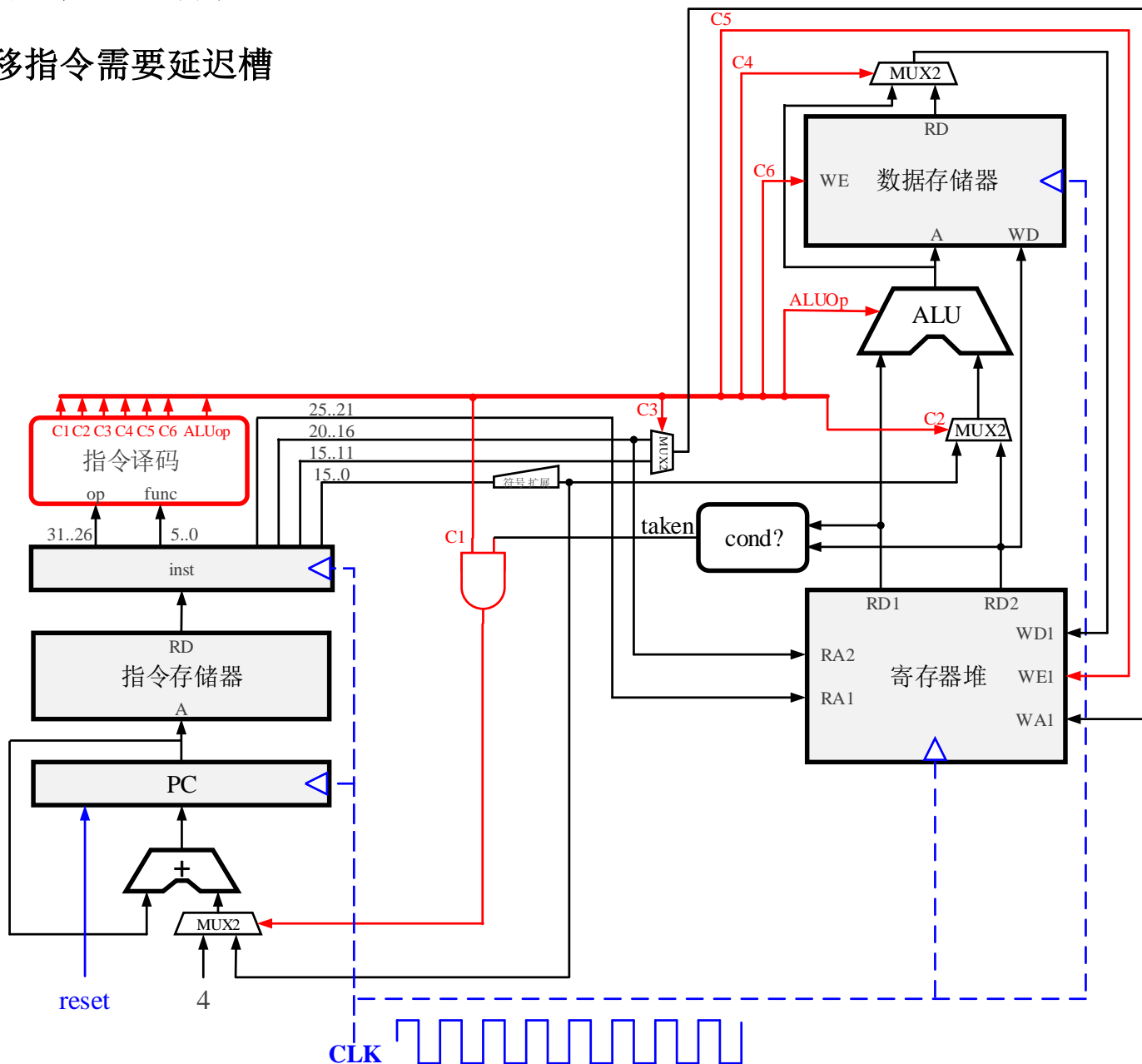
# 转移指令的延迟槽

- 紧挨着转移指令后面的一条指令不依赖转移条件执行
  - 从转移指令前面放一条指令到后面去
  - 也可以放一条nop
  - 例：把内存中100个整数的值分别加1（注意延迟槽SW指令访存地址偏移量的调整）

```
1 Loop: LW      R2, 0(R1)
2        ADDIU  R3, R2, #1
3        SW      0(R1), R3
4        ADDIU  R1, R1, -4
5        BNEZ   R1, Loop
6        NOP
```

```
1 Loop: LW      R2, 0(R1)
2        ADDIU  R3, R2, #1
3        ADDIU  R1, R1, -4
4        BNEZ   R1, Loop
3        SW      4(R1), R3
```

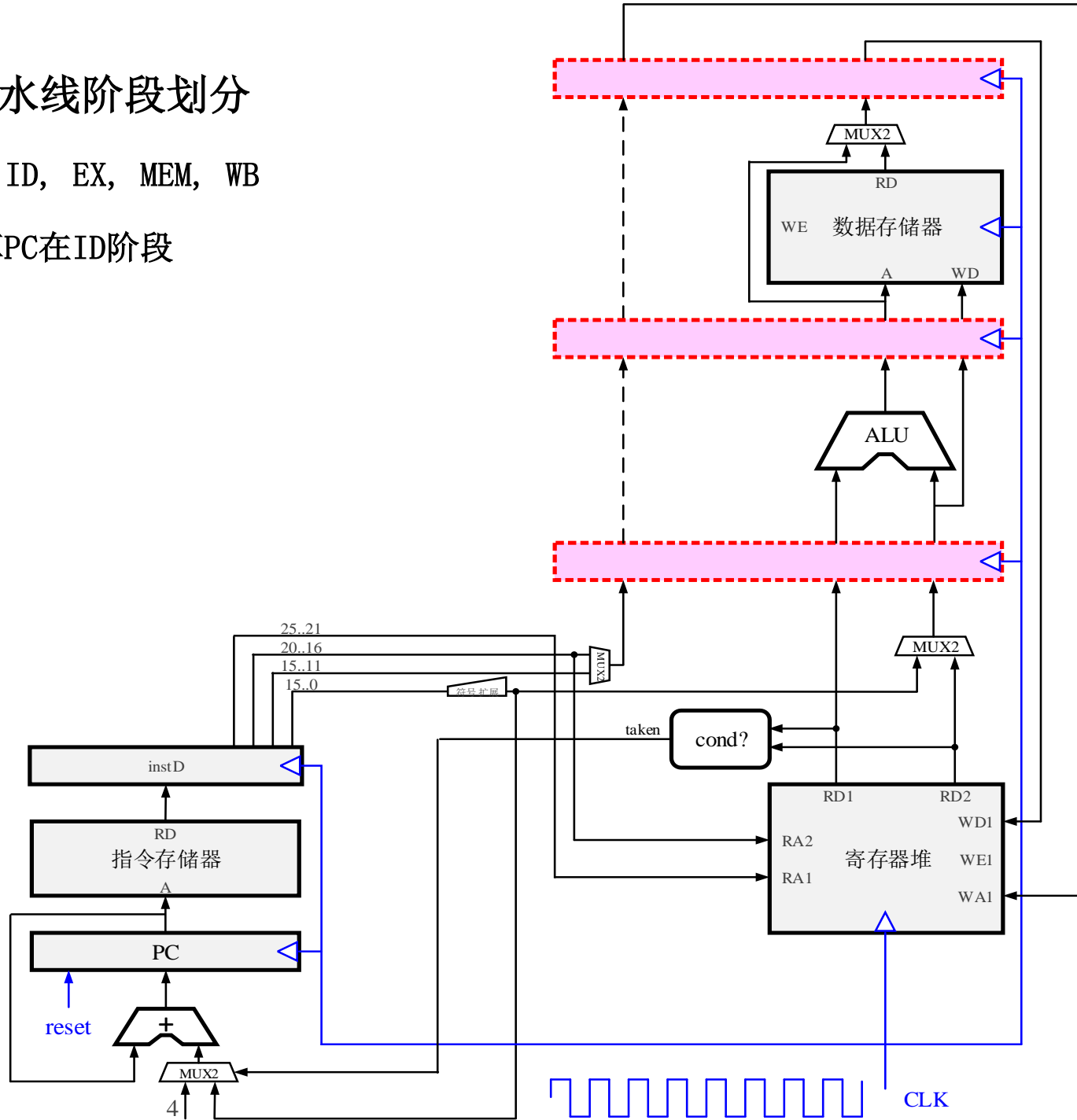
- 再次改进后的时序
  - 转移指令需要延迟槽



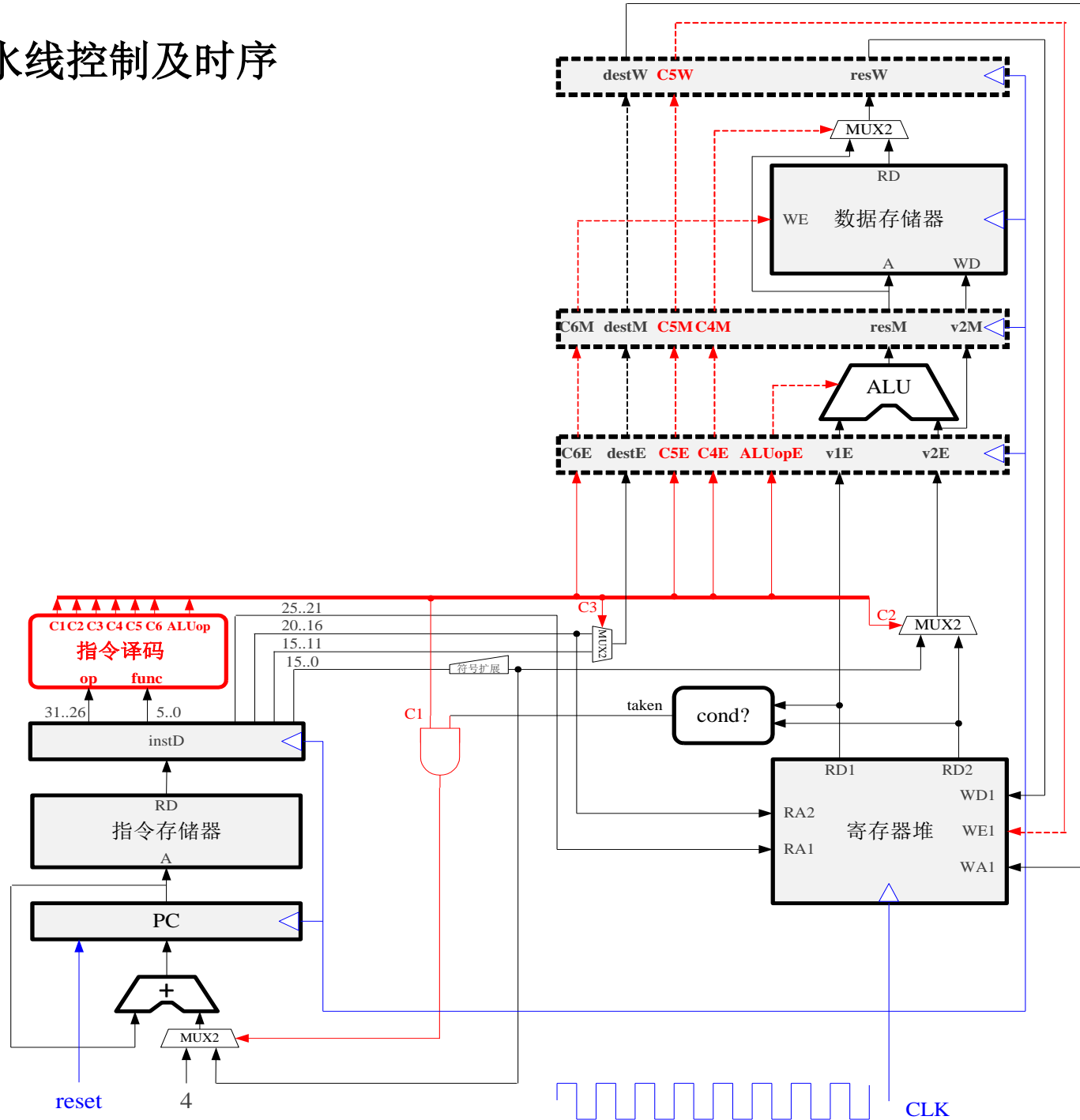
# 上述流水线的进一步改进

- 上述流水线是两级流水线
  - 取指和执行
  - 执行阶段做的事情较多
    - 译码（包括读取寄存器的值）
    - 运算（ALU操作）
    - 访存（取数或存数）
    - 写回到寄存器
  - 时钟周期较长，一拍内必须做完上述四件事情
- 可以把执行阶段再细分
  - 执行阶段分成译码、运算、访存、写回

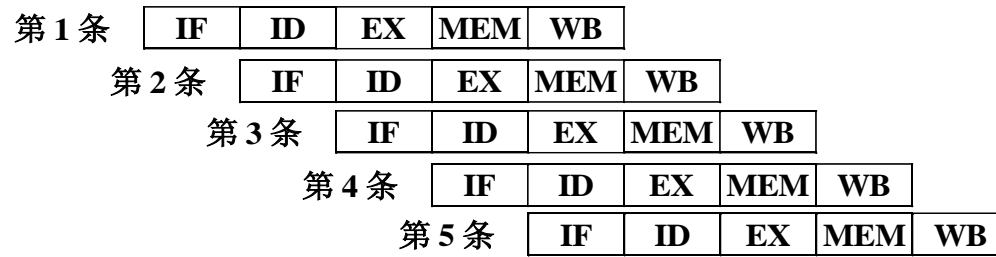
- 指令流水线阶段划分
  - IF, ID, EX, MEM, WB
  - 计算PC在ID阶段



- 指令流水线控制及时序



# 标准指令流水线



## 指令相关和流水线冲突

# 流水线的相关问题

- 指令相关的概念
  - 流水线变深了，相关问题更为突出。
  - 在流水线中，如果某指令的某个阶段必须等到它前面另一条指令的某个阶段后才能开始，则这两条指令存在相关
  - 相关的指令要隔开足够远，否则后面的指令就必须等待
- 在我们的流水线中，可能发生如下相关
  - 红线表示数据相关
  - 蓝线表示转移相关（从中可看出转移指令在译码阶段执行的必要性）
  - 还有结构相关（在我们的例子中不存在）



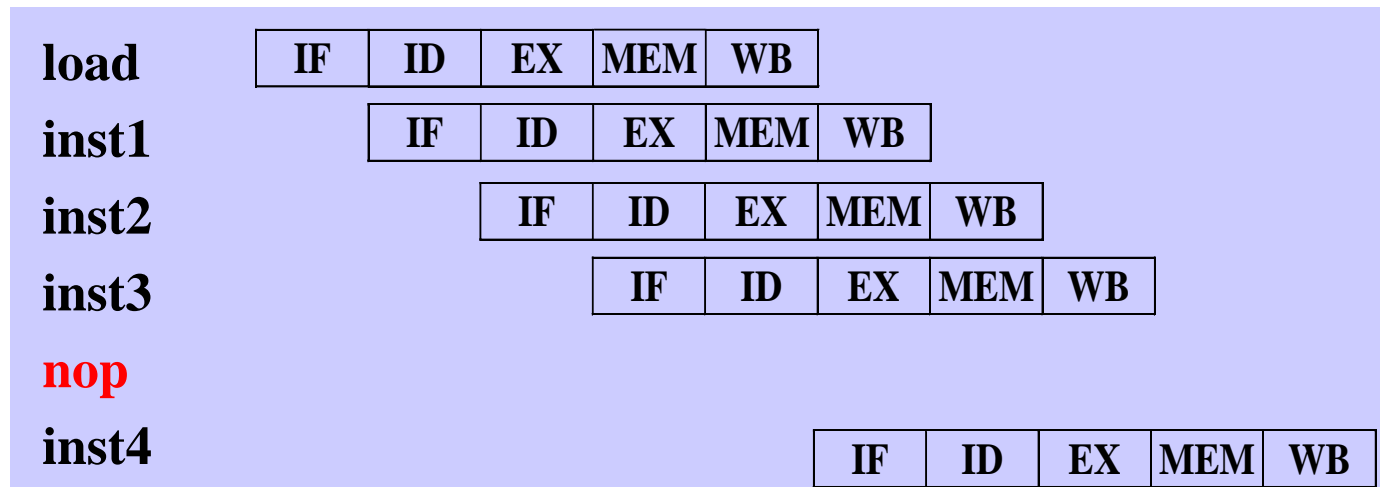
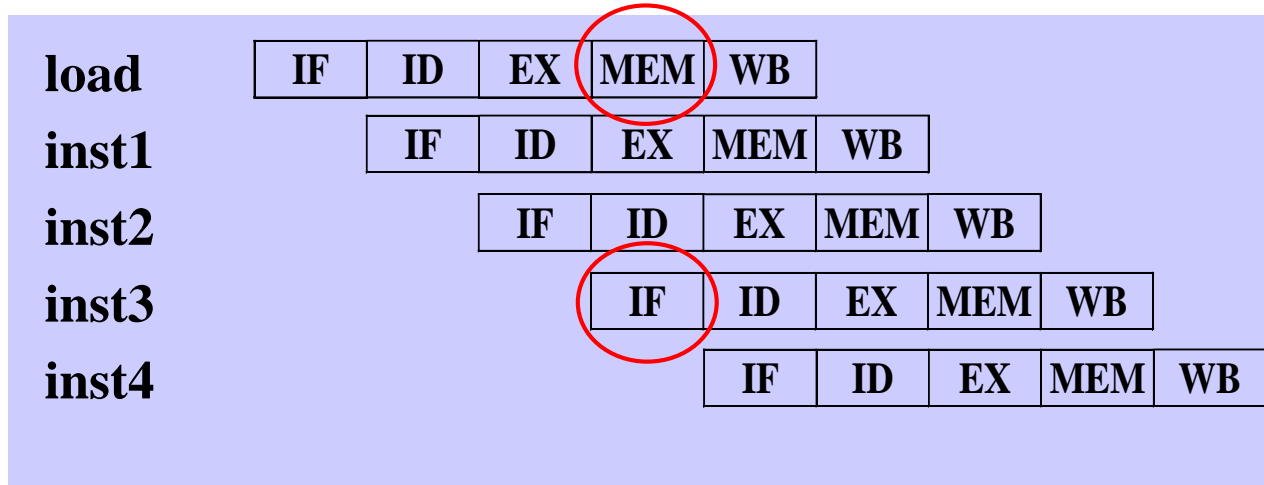


# 指令流水线的相关

- 数据相关：使用同一个寄存器引起的相关
  - 如后面的指令用到前面指令的结果
- 控制相关：与PC有关的相关
  - 每条指令取指用到PC，转移指令修改PC
- 结构相关：资源冲突
  - 多条指令同时使用一个功能部件
- 相关引起流水线阻塞

# 由访存引起的结构相关

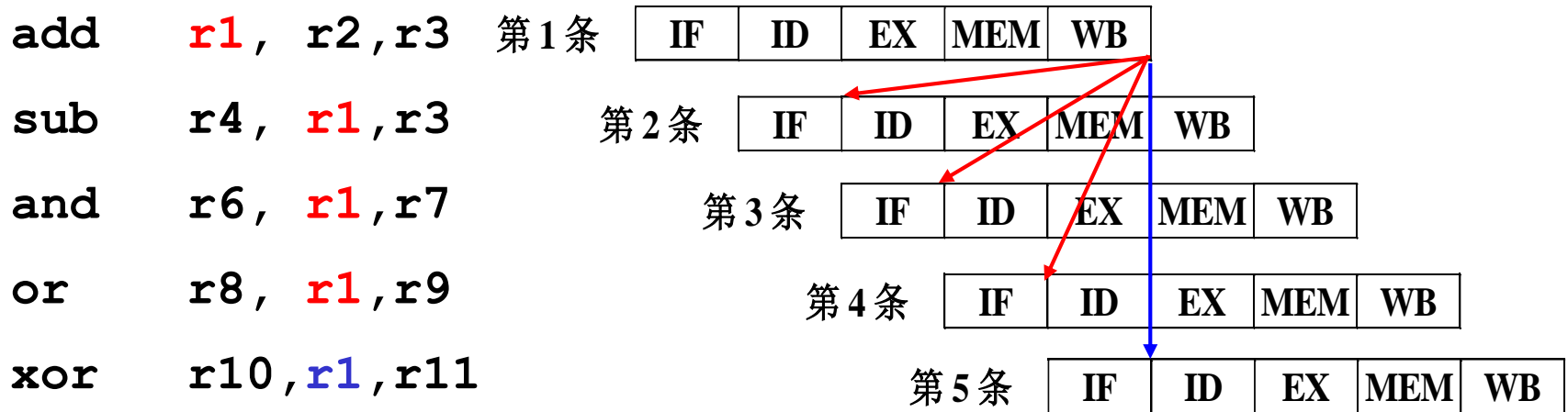
- 访存和取指都需要存储器端口
  - Havard结构



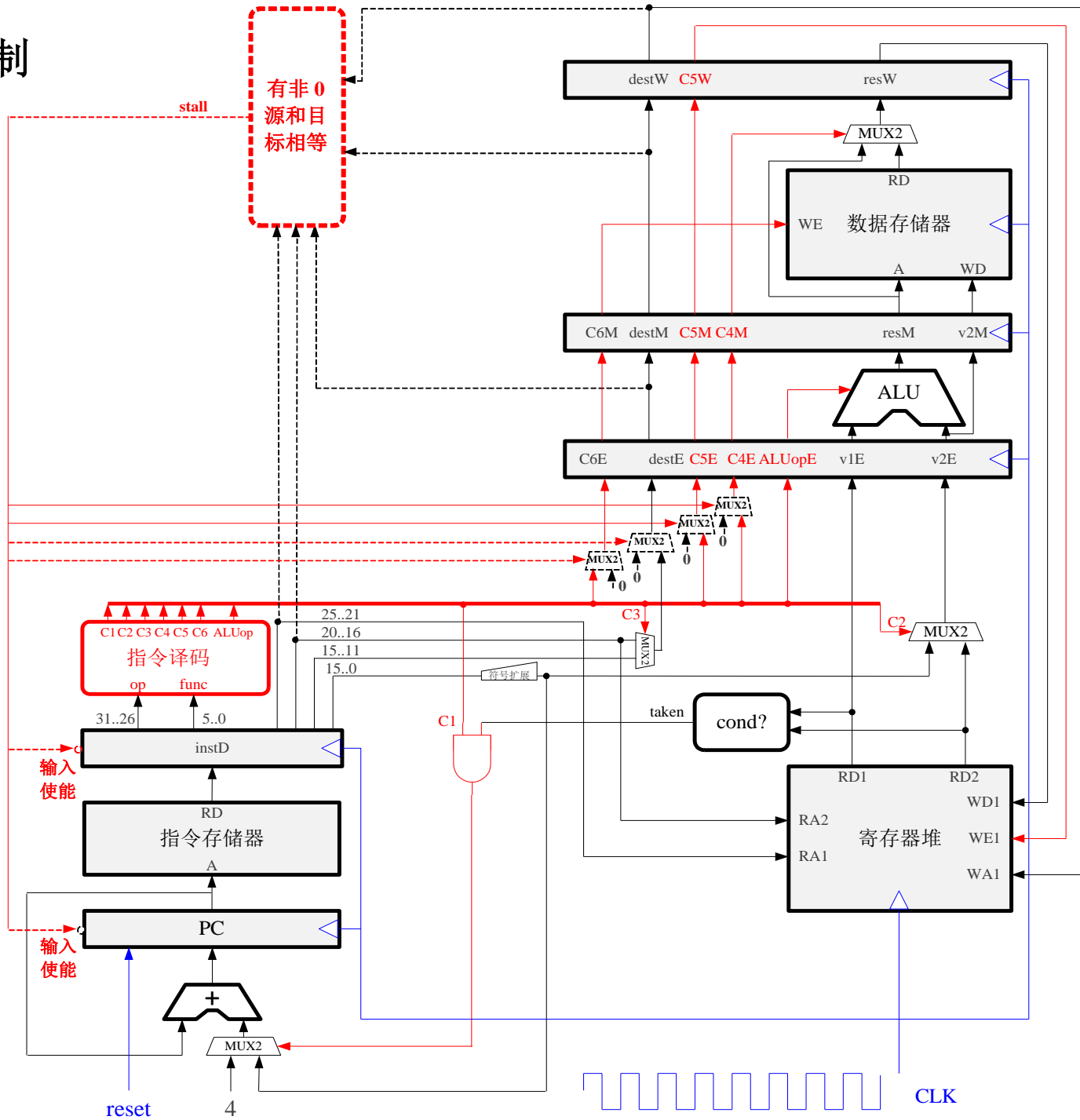
# 数据相关

- RAW (Read After Write)
  - 后面指令用到前面指令所写的的数据
- WAW (Write After Write)
  - 两条指令写同一个单元
  - 在简单流水线中没有此类相关，因为不会乱序执行
- WAR (Write After Read)
  - 后面指令覆盖前面指令所读的单元
  - 在简单流水线中没有此类相关
- 在动态流水线中会有WAR和WAW相关

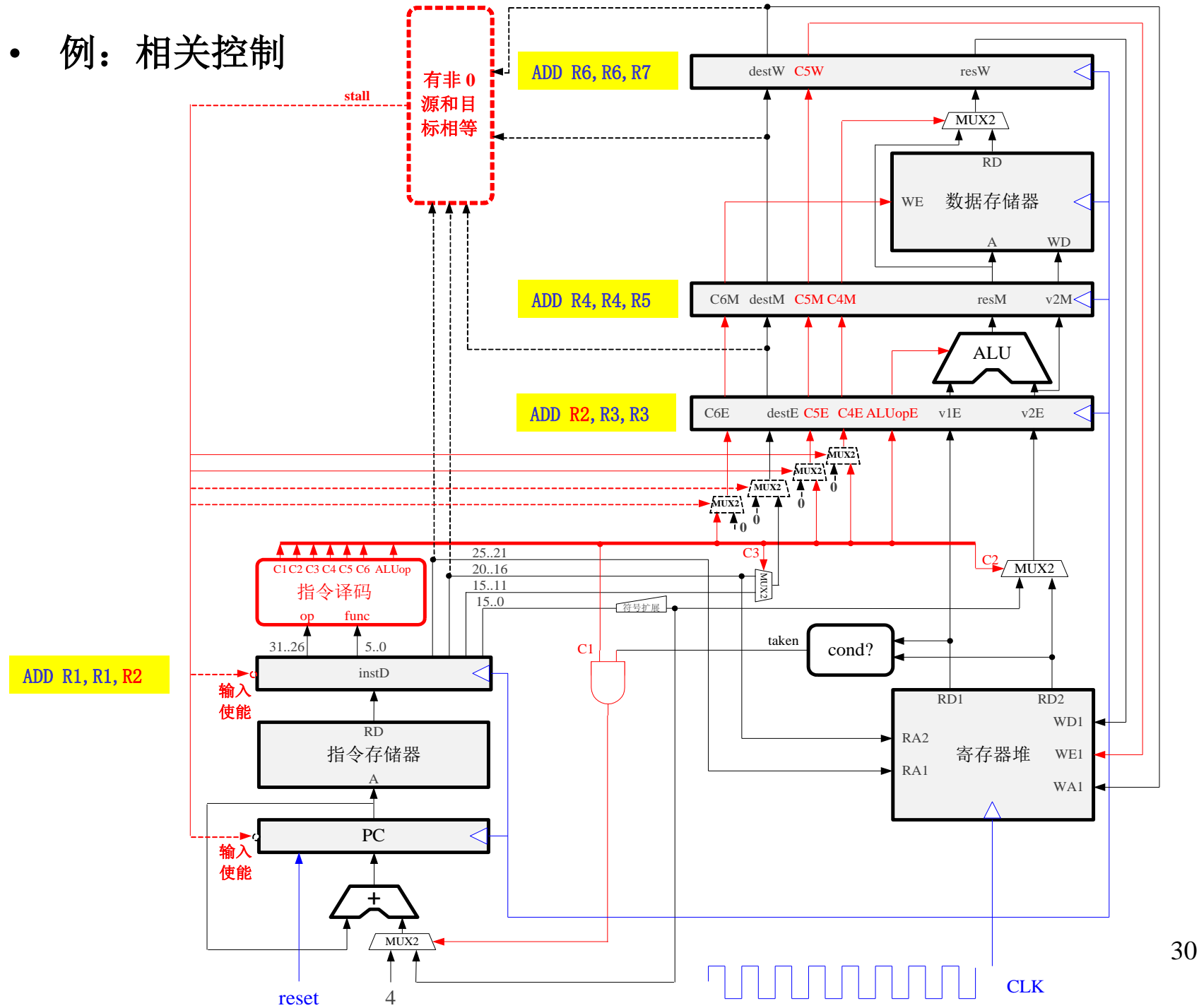
# 数据相关的例子



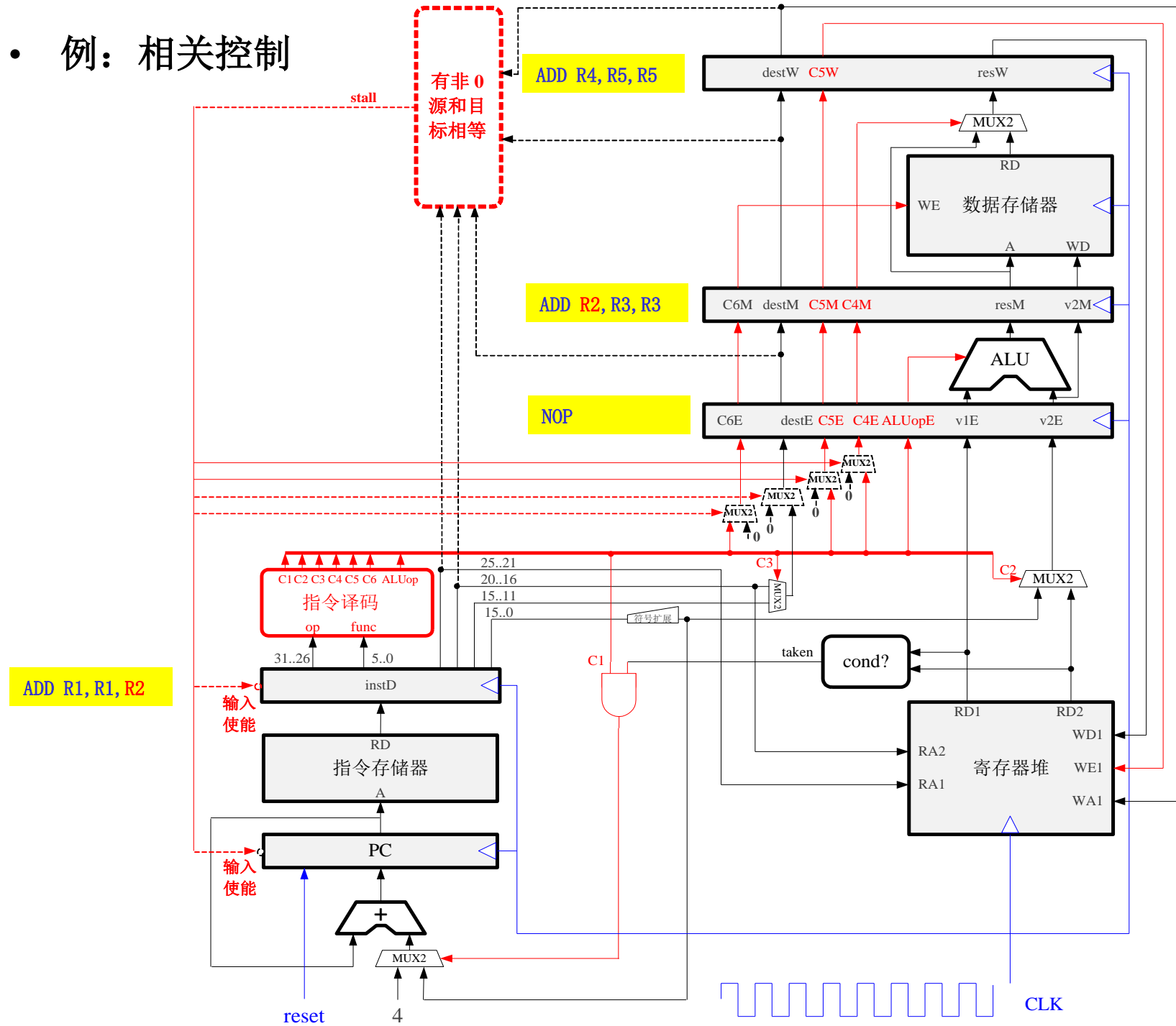
# 相关控制



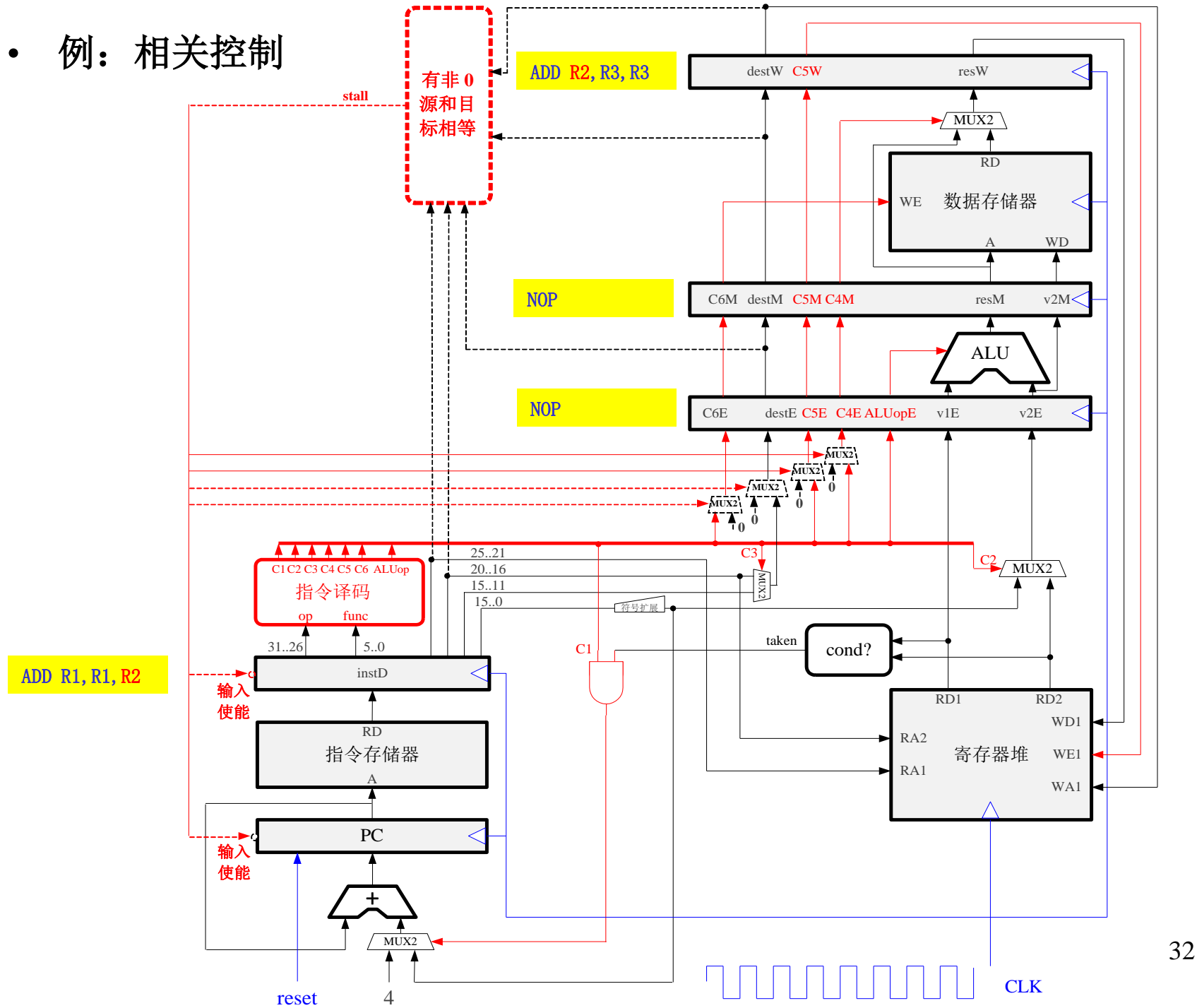
- 例：相关控制



# 例：相关控制

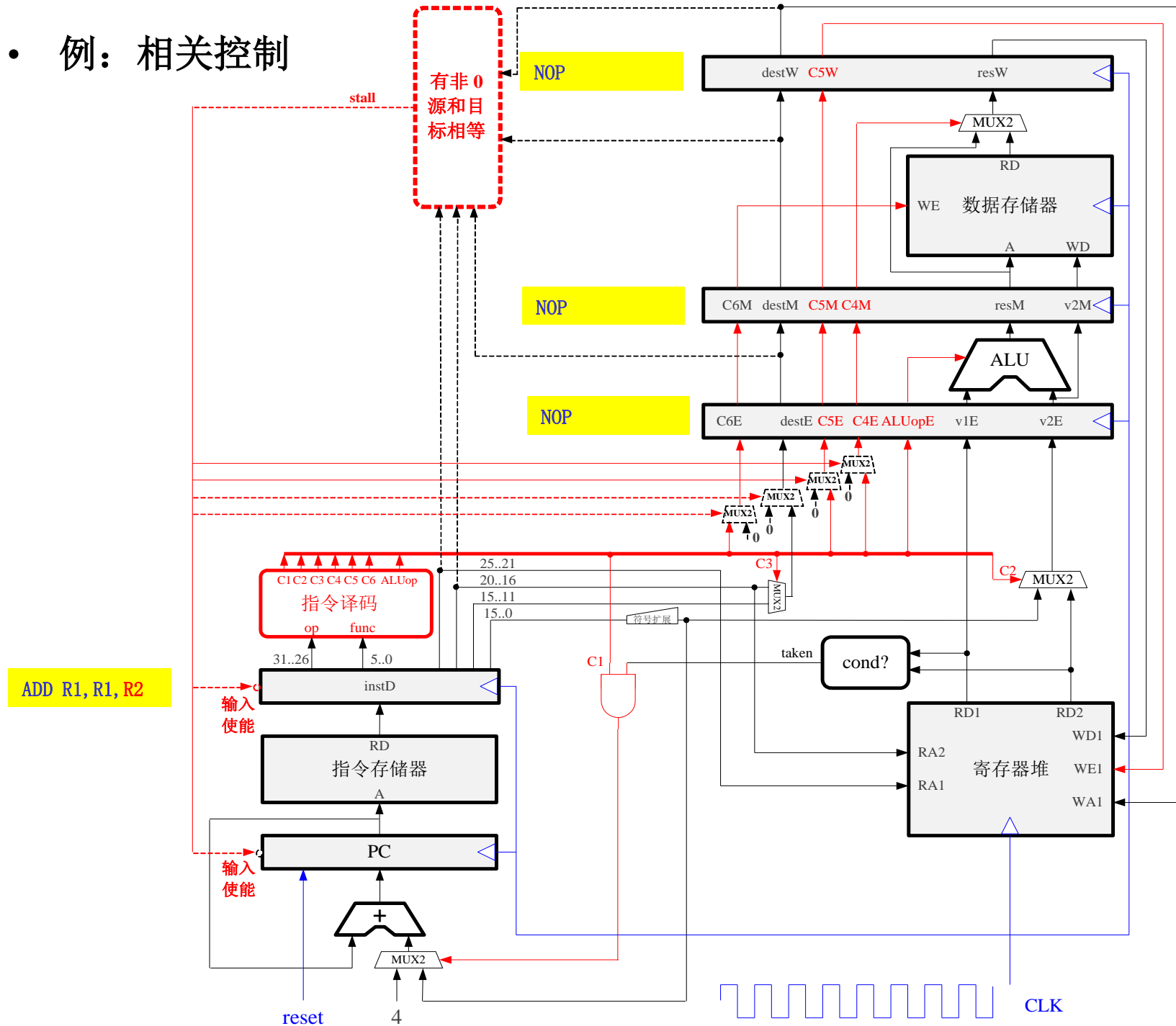


- 例：相关控制

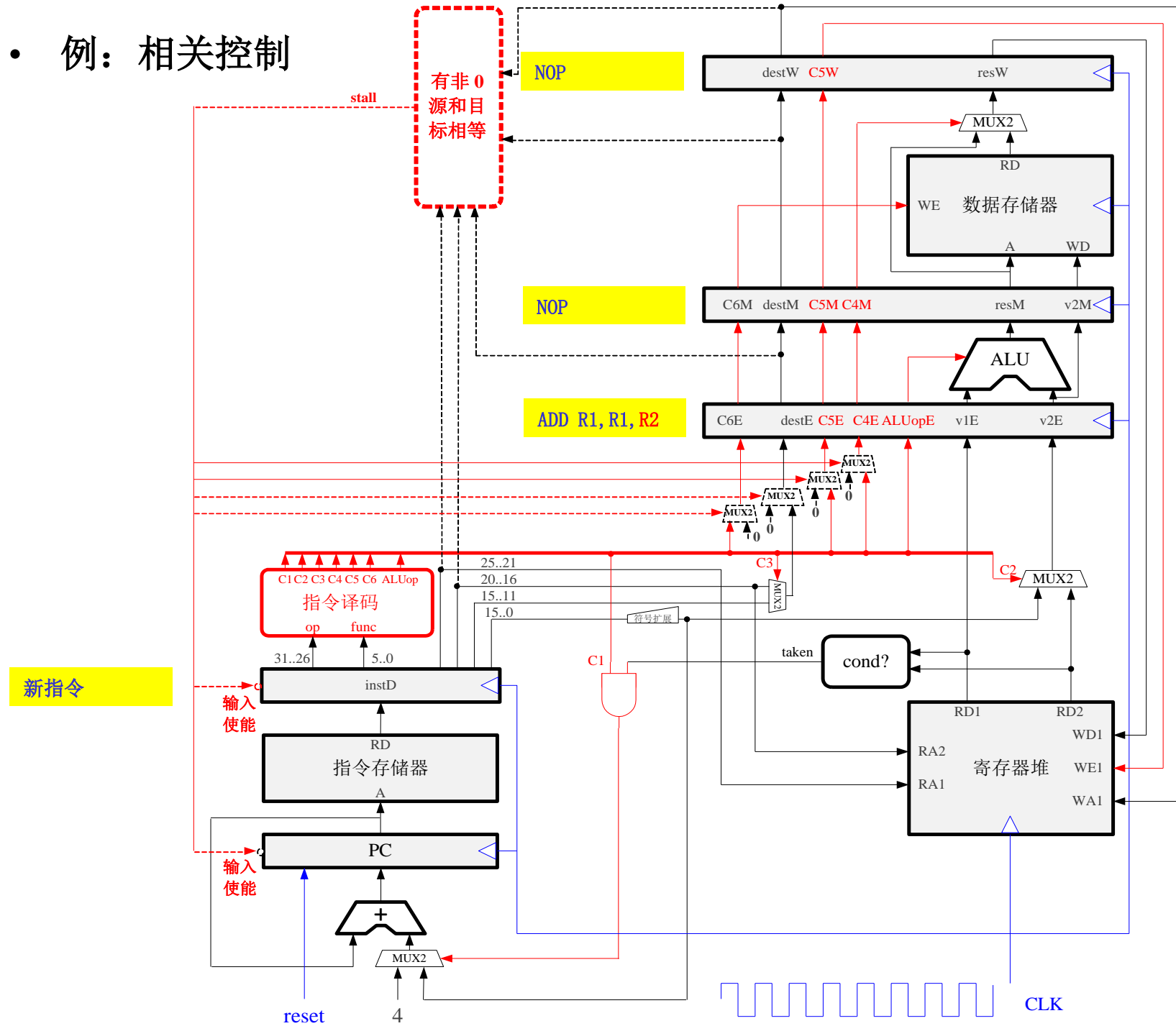




# 例：相关控制

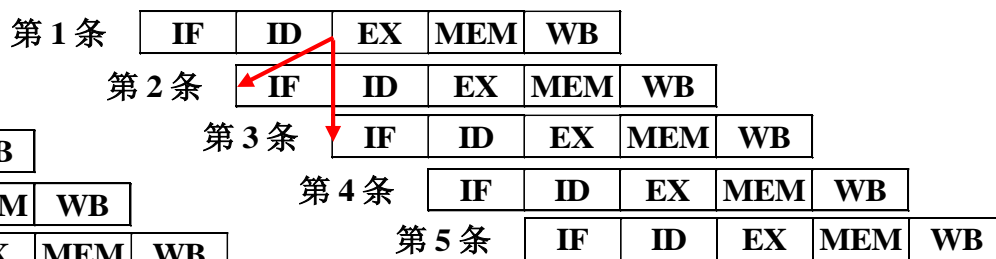
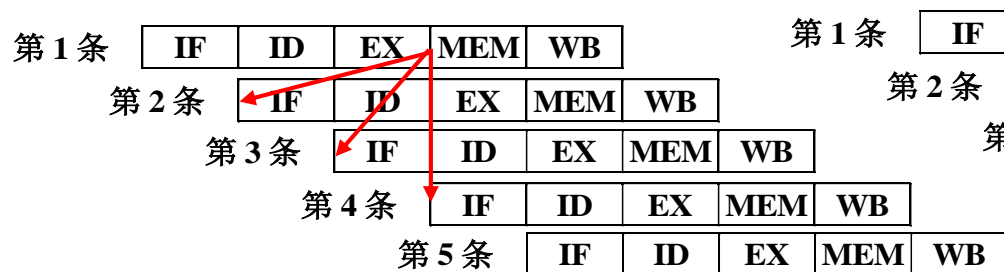


# 例：相关控制

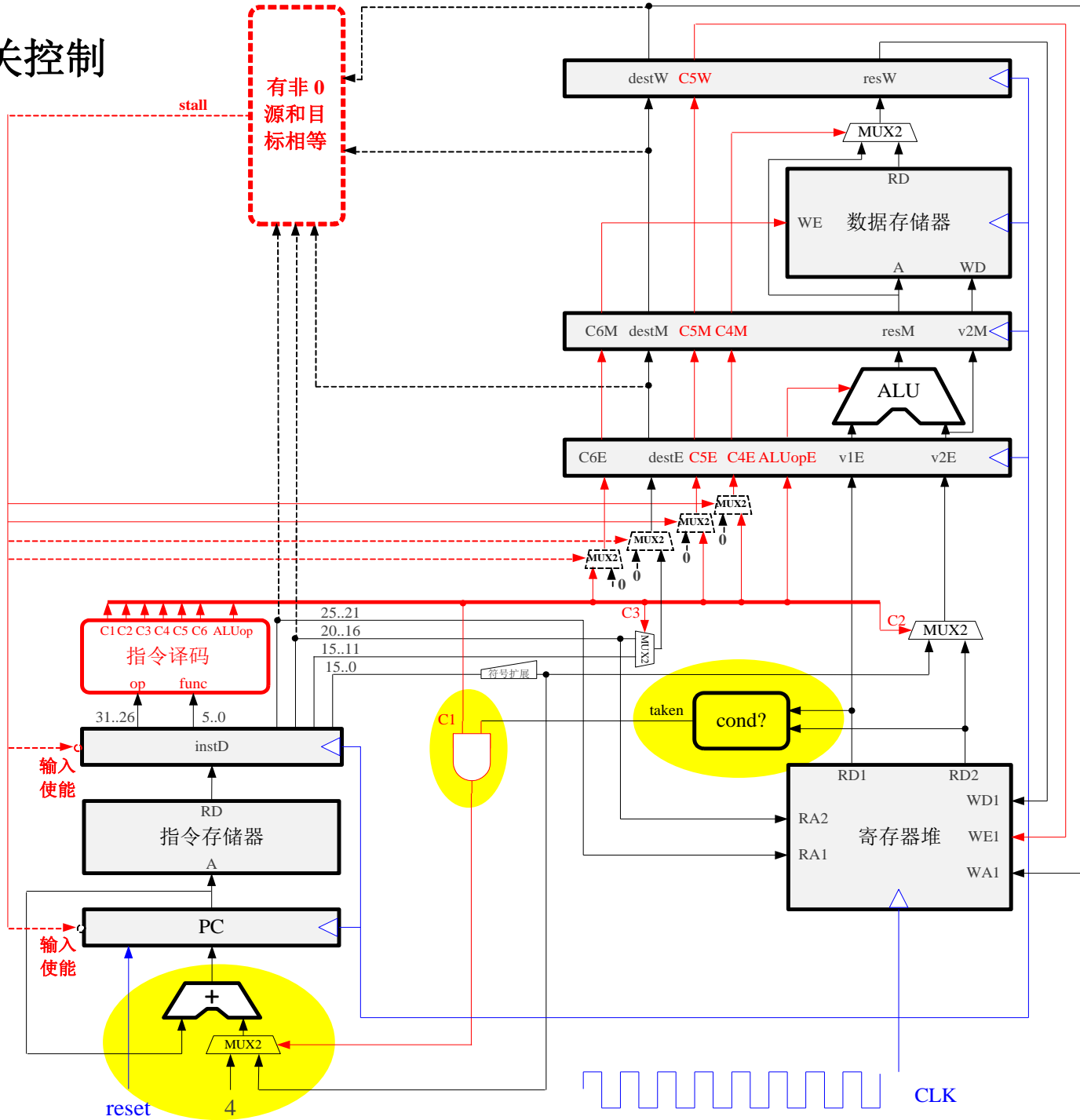


# 控制相关

- 控制相关直接影响后续取指
  - 转移指令计算的下一条指令地址在EX阶段计算，下一条指令等2拍
- 五级流水避免控制相关引起流水线阻塞的典型方法
  - 使用专门的地址运算部件把地址计算提前到译码阶段可以少等一拍
  - 使用一个转移指令延迟槽（delay slot）可以不用等待



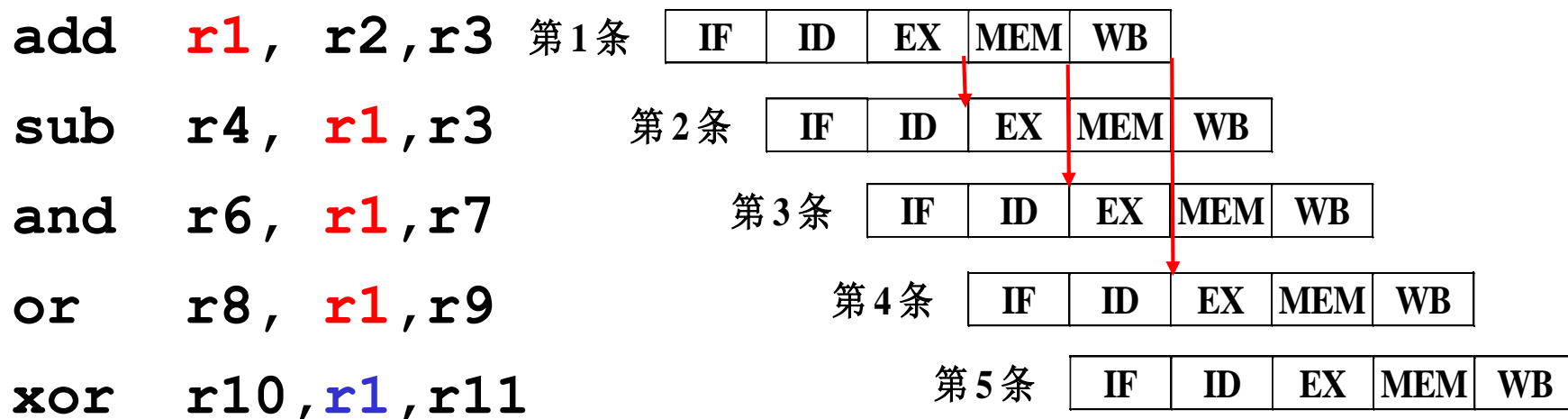
# 转移相关控制



# 流水线的前递技术

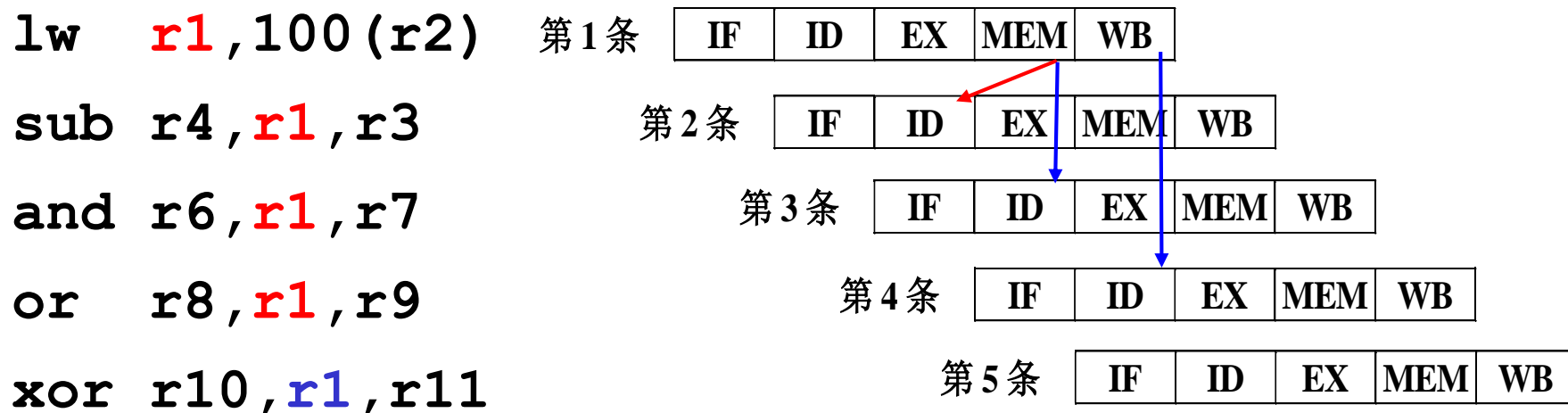
# 解决RAW相关的前递（Forwarding）技术

- 在执行（EX）阶段的运算结果出来后直接送到后续指令的EX阶段



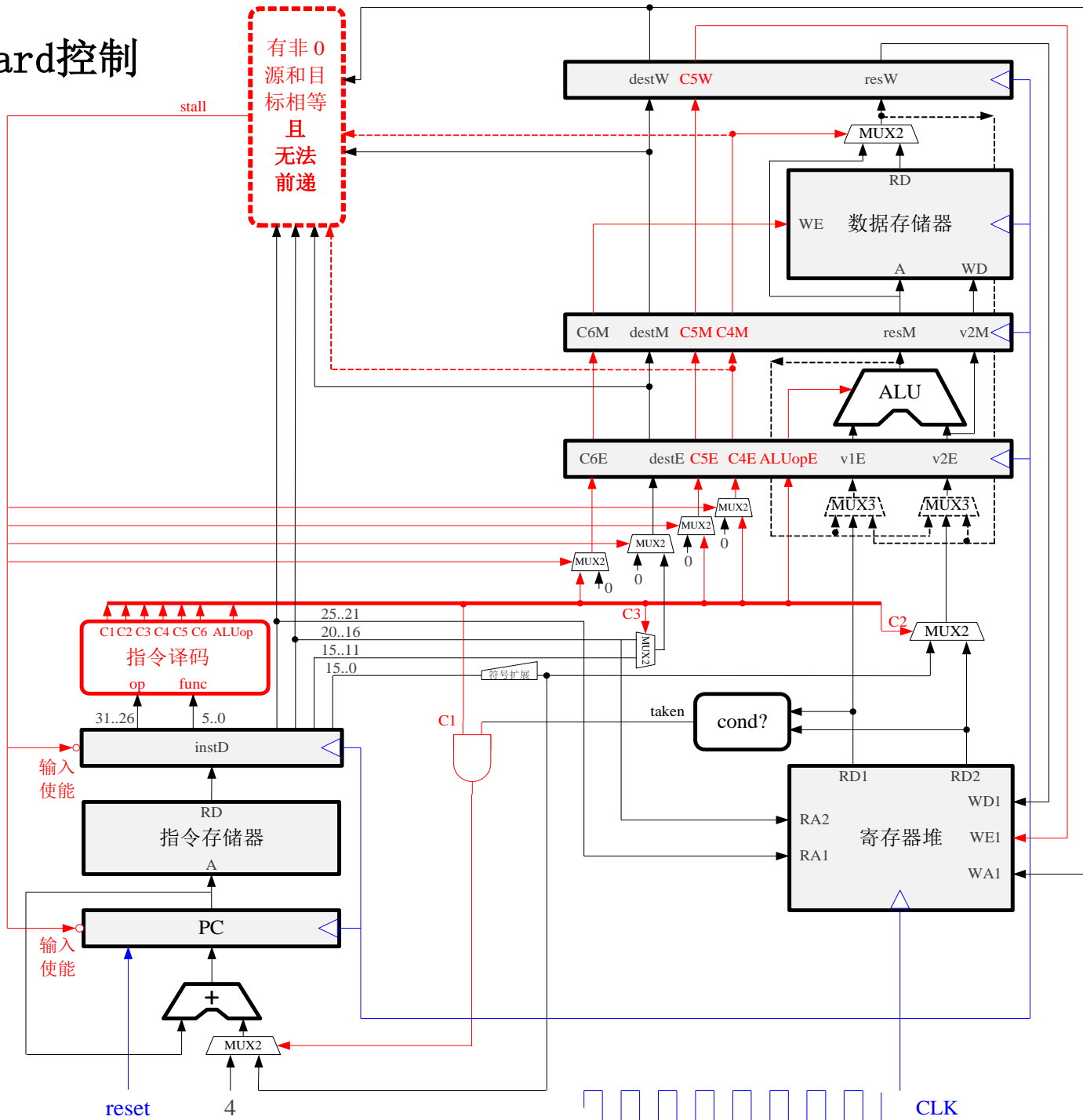
# 前递情况下的数据相关

- 取数指令的结果在MEM阶段才有效，后续相关指令需要阻塞一拍



- ALU Forward控制

只考虑ALU  
前递，不包括访存和转移前递





# Forwarding的相关处理逻辑

- 以ALU左端的输入为例（优先级由高到低）
  - 当src1D==destE且执行级操作是LW时，后面流水线暂停，往前面流水线送空操作
  - 当src1D==destE且执行级操作不是LW时，选择左边通路
  - 当src1D==destM时，选择右边通路
  - 当src1D和前面两级的目标寄存器域destE和destM都不相等时，选择中间通路

# 通过静态调度解决相关

如下程序段的优化和非优化代码

**a = b + c;**

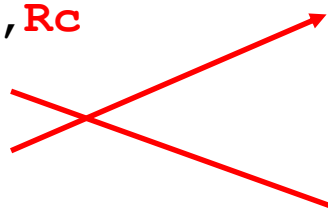
**d = e - f;**

Slow code:

```
LW    Rb, b
LW    Rc, c
ADD   Ra, Rb, Rc
SW    a, Ra
LW    Re, e
LW    Rf, f
SUB   Rd, Re, Rf
SW    d, Rd
```

Fast code:

```
LW    Rb, b
LW    Rc, c
LW    Re, e
ADD   Ra, Rb, Rc
LW    Rf, f
SW    a, Ra
SUB   Rd, Re, Rf
```



# 流水线和例外

# 例外(Exception)与流水线

- 例外原因
  - I/O请求：外部中断
  - 指令例外：用户请求例外
    - 系统调用、断点、跟踪调试指令
  - 运算部件
    - 整数运算溢出、浮点异常
  - 存储管理部件
    - 访存地址不对齐、用户访问系统空间、TLB失效、缺页、存储保护错（写只读页）
  - 保留指令错：未实现指令
  - 硬件错
  - 等等

- 例外发生的流水阶段
  - 取指：访存例外
  - 译码：保留指令、中断指令如Trap、Syscall
  - 执行：整数溢出、浮点异常（如除零）等
  - 访存：访存例外
  - 其它：外部中断，可能在任何时候发生
- 例外特征
  - 同步与异步
  - 用户请求与系统强制
  - 可屏蔽与不可屏蔽
  - 指令内与指令间
  - 可恢复与结束

	同步/异步	用户/强制	可/不可屏蔽	指令内/间	可恢复/结束
I/O 请求	异步	强制	不可屏蔽	指令间	可恢复
系统调用	同步	用户	不可屏蔽	指令间	可恢复
跟踪调试	同步	用户	可屏蔽	指令间	可恢复
断点	同步	用户	可屏蔽	指令间	可恢复
整数溢出	同步	强制	可屏蔽	指令内	可恢复
浮点异常	同步	强制	可屏蔽	指令内	可恢复
缺页	同步	强制	不可屏蔽	指令内	可恢复
访存地址不齐	同步	强制	可屏蔽	指令内	可恢复
存储保护错	同步	强制	不可屏蔽	指令内	可恢复
非法指令	同步	强制	不可屏蔽	指令内	结束
硬件错	异步	强制	不可屏蔽	指令内	结束
电源错	异步	强制	不可屏蔽	指令内	结束

- 在前述例外中，**指令内可恢复**例外的处理比较困难，条件转移指令的 delay slot 又增加了例外处理的难度
- 精确例外：在处理例外时，发生例外指令前面的所有指令都执行完，例外指令后面的所有指令还未执行。精确例外是存储管理和IEEE运算规范的要求
  - 发生例外指令前面的指令继续执行完
  - 后面的指令不能修改机器状态，对运算状态字的修改可能在EX阶段进行
  - 多条指令发生例外

- 可以把每条指令的例外延迟到WB时再处理
  - 对机器状态的修改也在WB阶段进行
  - 对状态寄存器的修改从EX阶段延迟到WB阶段

LD r1, 0(r2)	IF	ID	EX	MEM	WB	
ADD r5, r3, r4		IF	ID	EX	MEM	WB

LD r1, 0(r2)	IF	ID	EX	MEM	WB	
ADD r5, r3, r4		IF	ID	EX	MEM	WB

LD r1, 0(r2)	IF	ID	EX	MEM	WB	
ADD r5, r3, r4		IF	ID	EX	MEM	WB

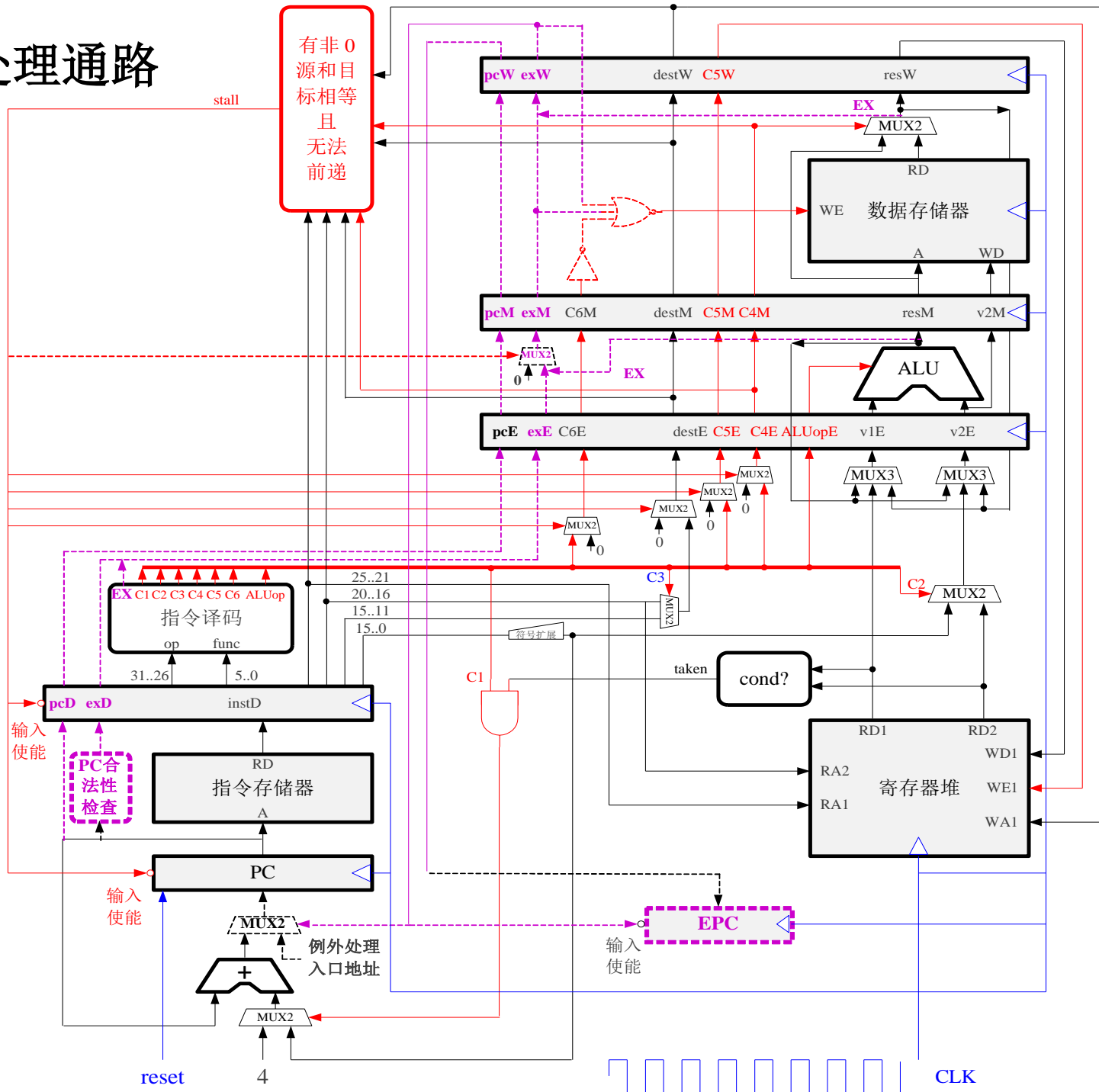
LD r1, 0(r2)	IF	ID	EX	MEM	WB	
ADD r5, r3, r4		IF	ID	EX	MEM	WB



- 简单流水线的例外处理

- 任何一级流水发生例外时，在流水线中记录下发生例外的事件，直到WB阶段再处理
- 如果在EX阶段要修改机器状态（如状态寄存器），保存下来直到WB阶段再修改。
- 指令的PC值随指令流水前进到WB阶段例外处理专用
- 外部中断作为IF的例外处理
- 指定一个通用寄存器中（如最后一个）为例外处理时保存PC值专用。
- 当发生例外的指令在WB阶段时：
  - 保存该指令的PC（也在WB阶段），有些机器还保存其它状态
  - 置PC值为例外处理程序入口地址

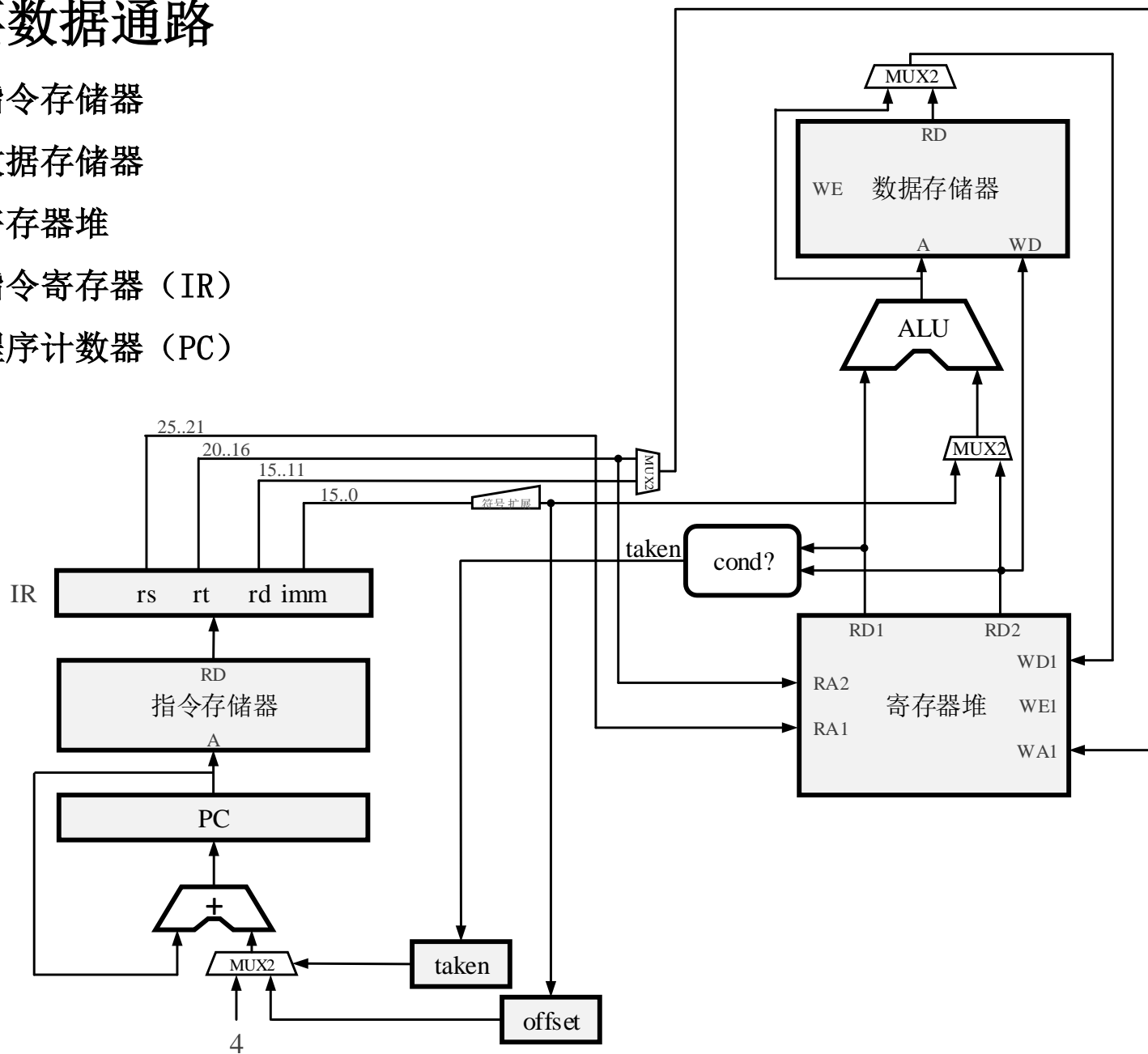
• 例外处理通路



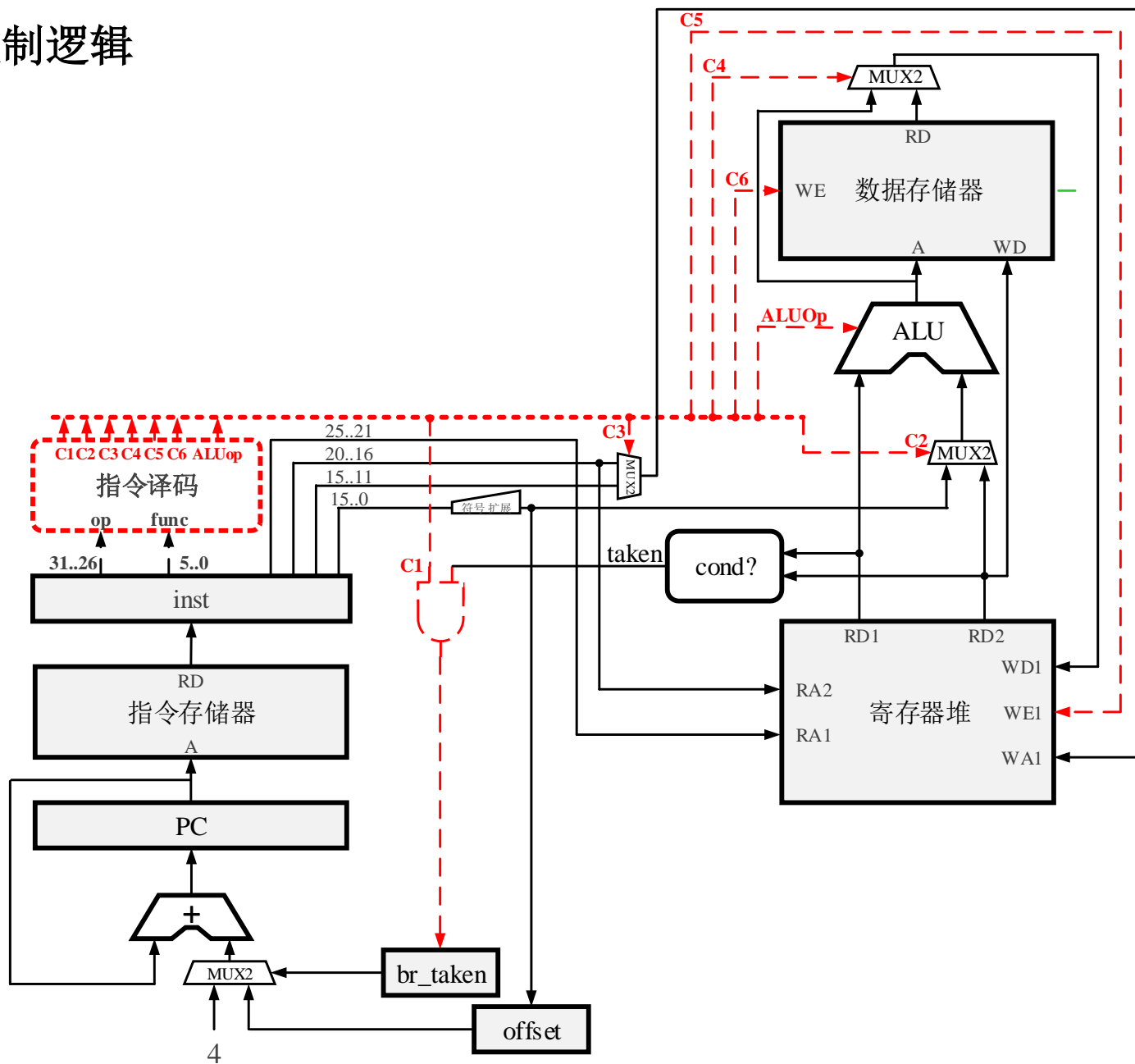
# 回顾

## • 主要数据通路

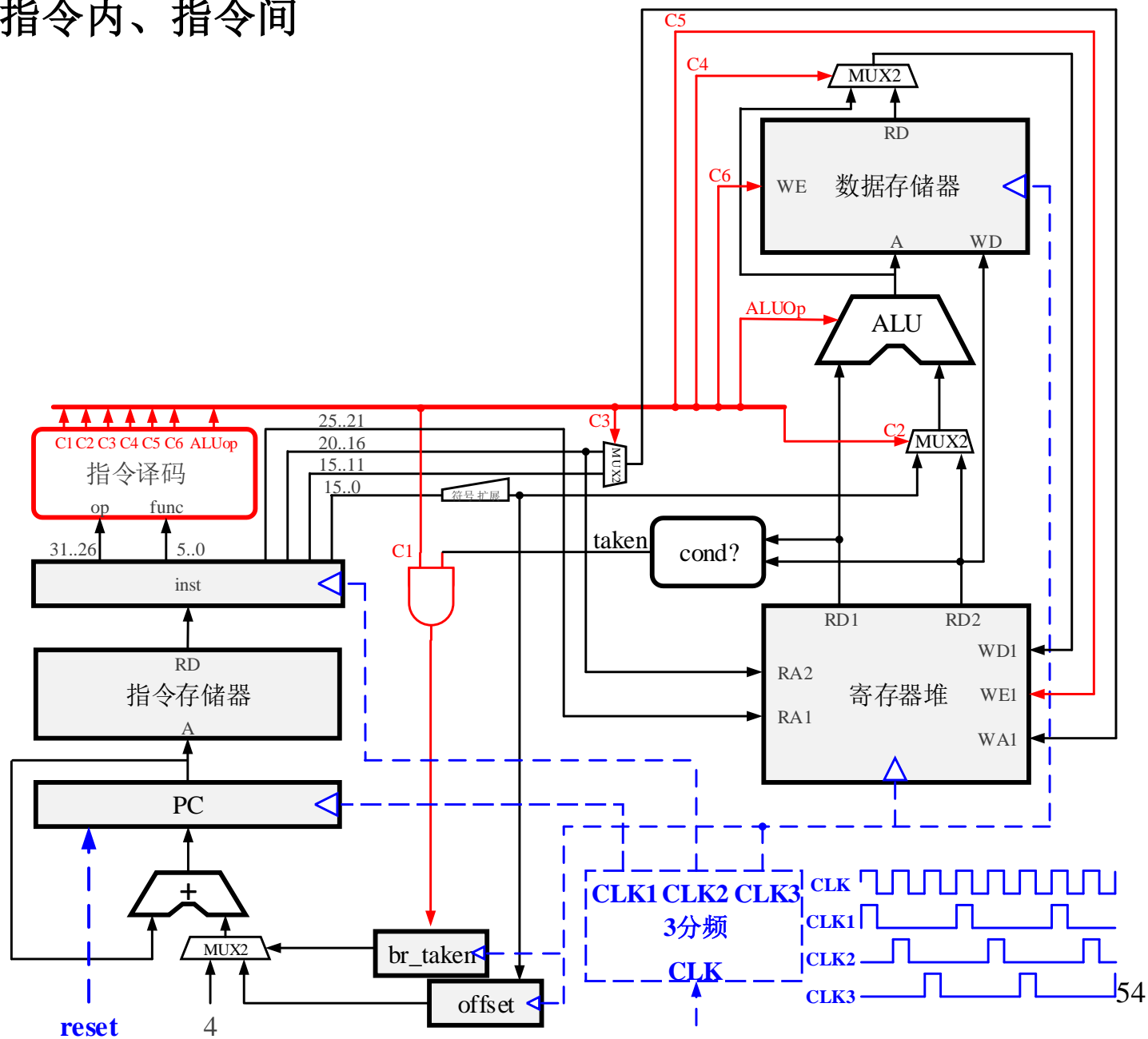
- 指令存储器
- 数据存储器
- 寄存器堆
- 指令寄存器 (IR)
- 程序计数器 (PC)



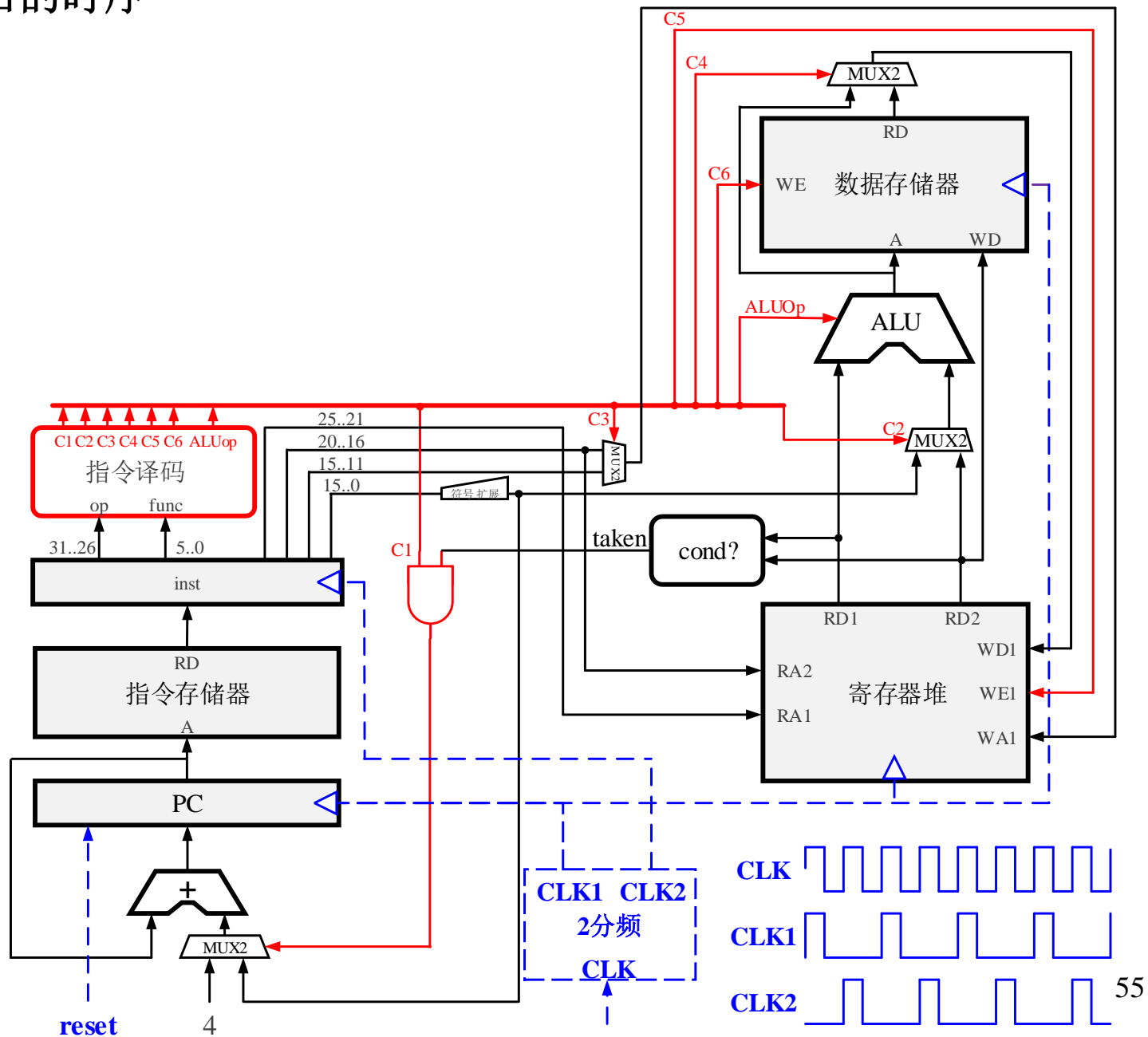
- 主要控制逻辑



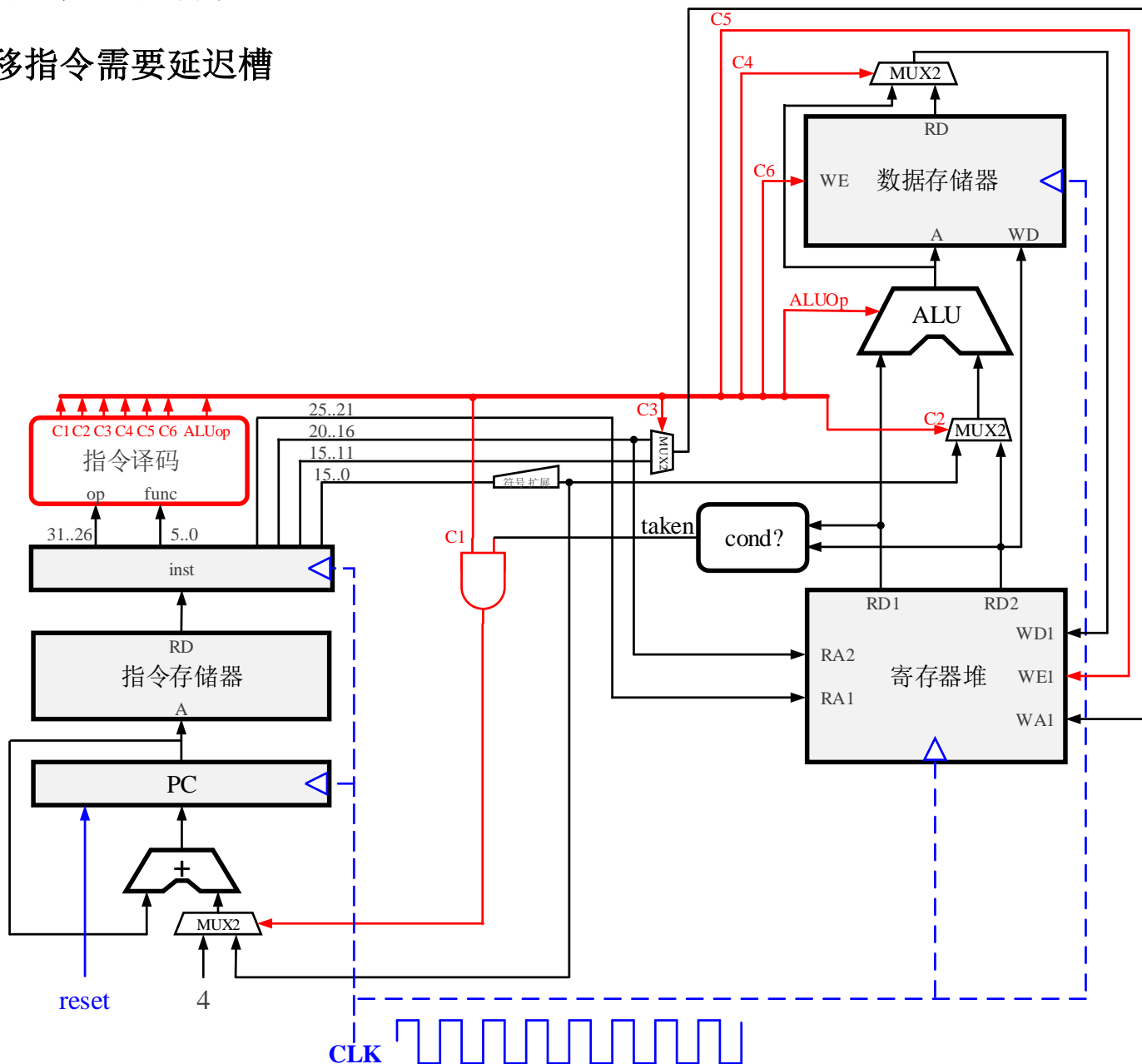
- 时序：指令内、指令间



- 改进后的时序

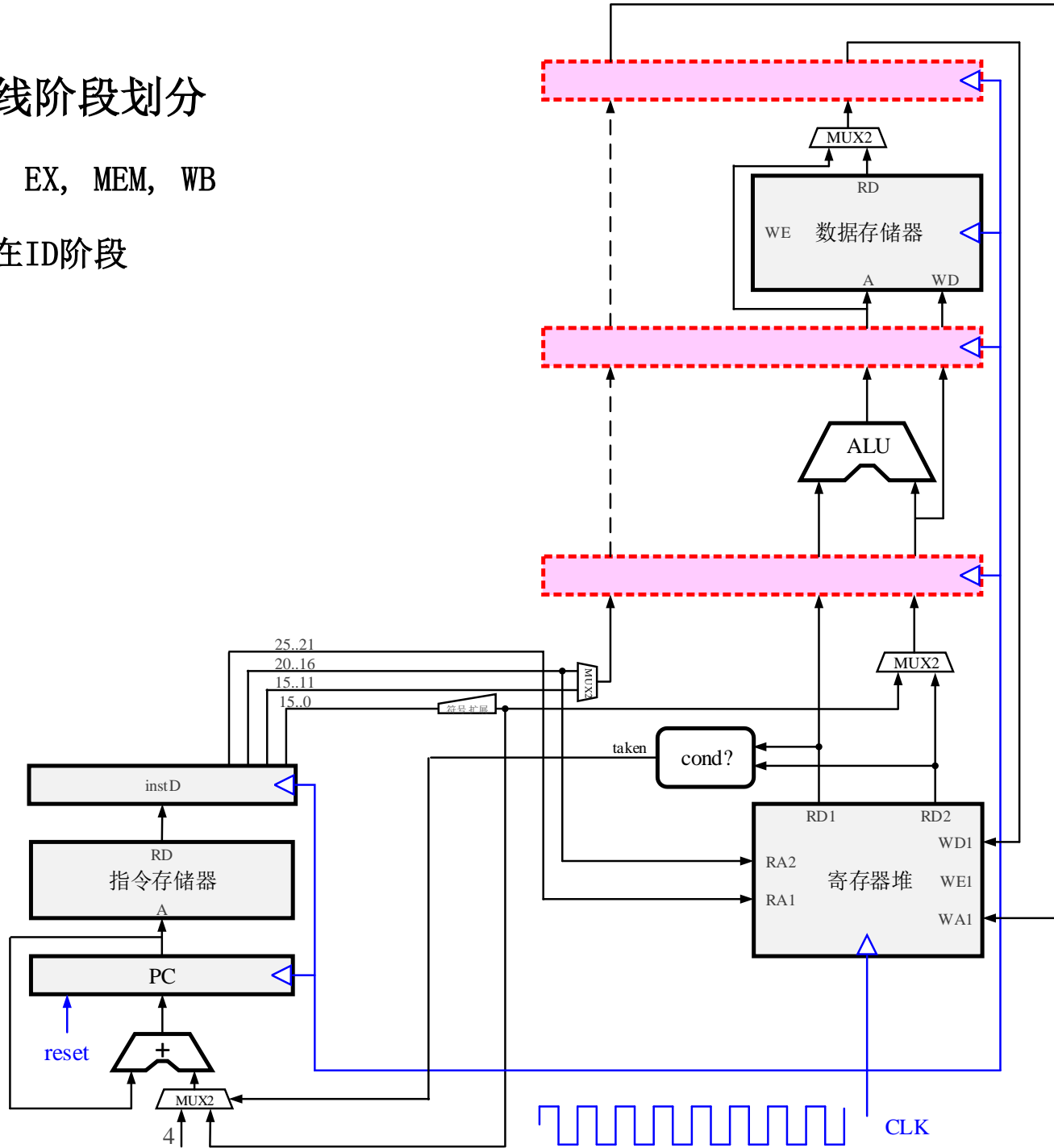


- 再次改进后的时序
  - 转移指令需要延迟槽

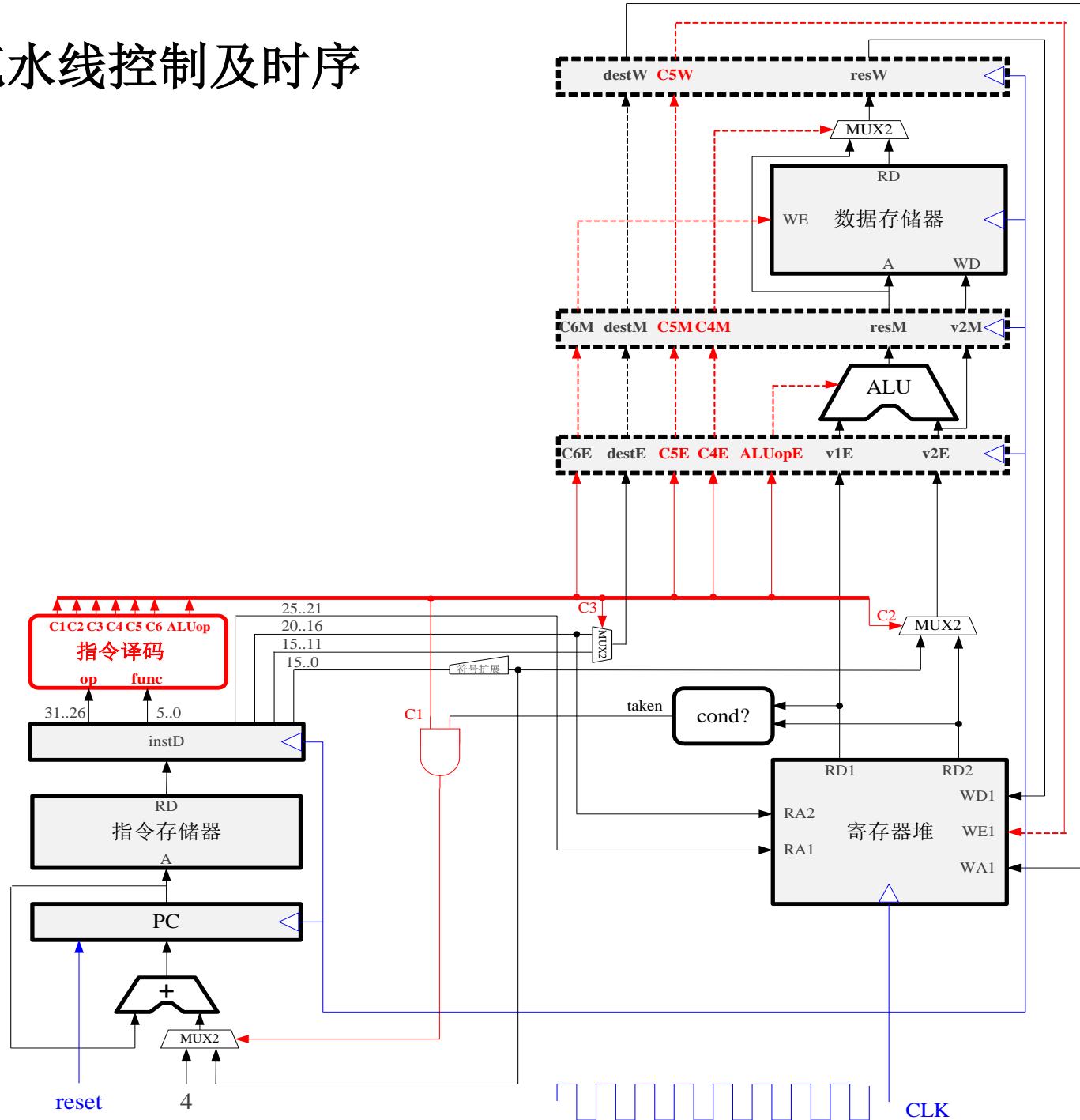




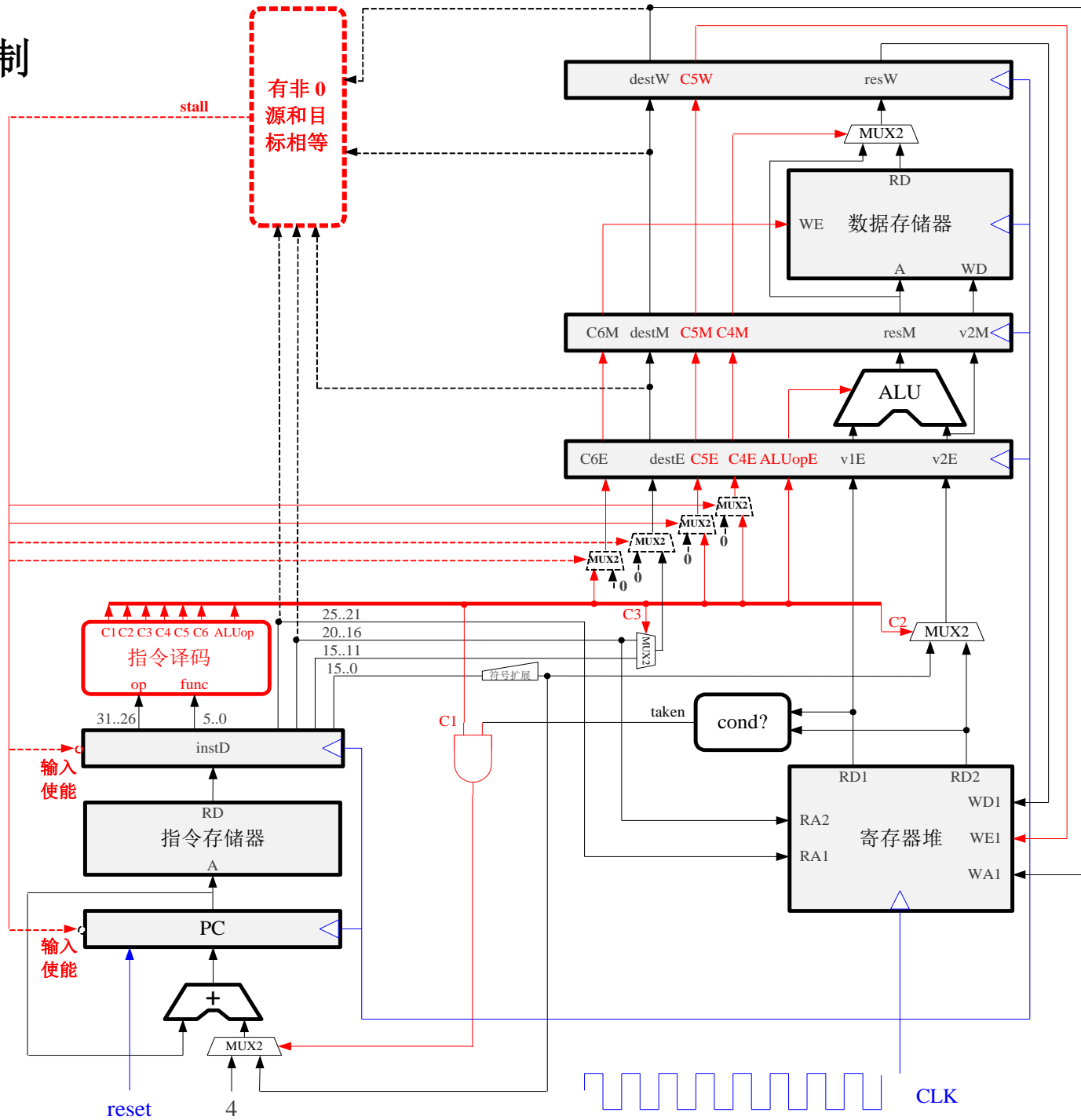
- 指令流水线阶段划分
  - IF, ID, EX, MEM, WB
  - 计算PC在ID阶段



# 指令流水线控制及时序

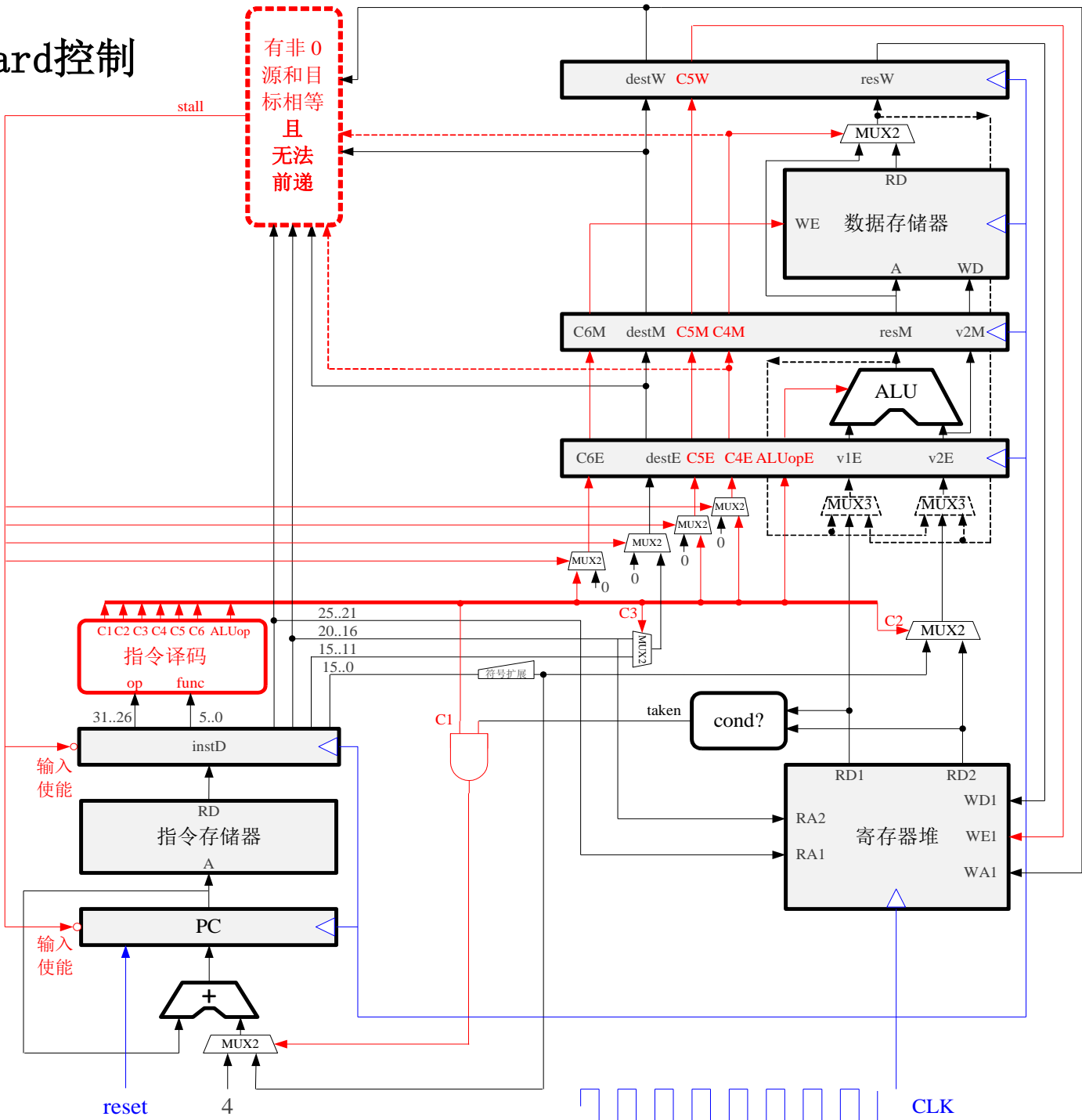


- 相关控制

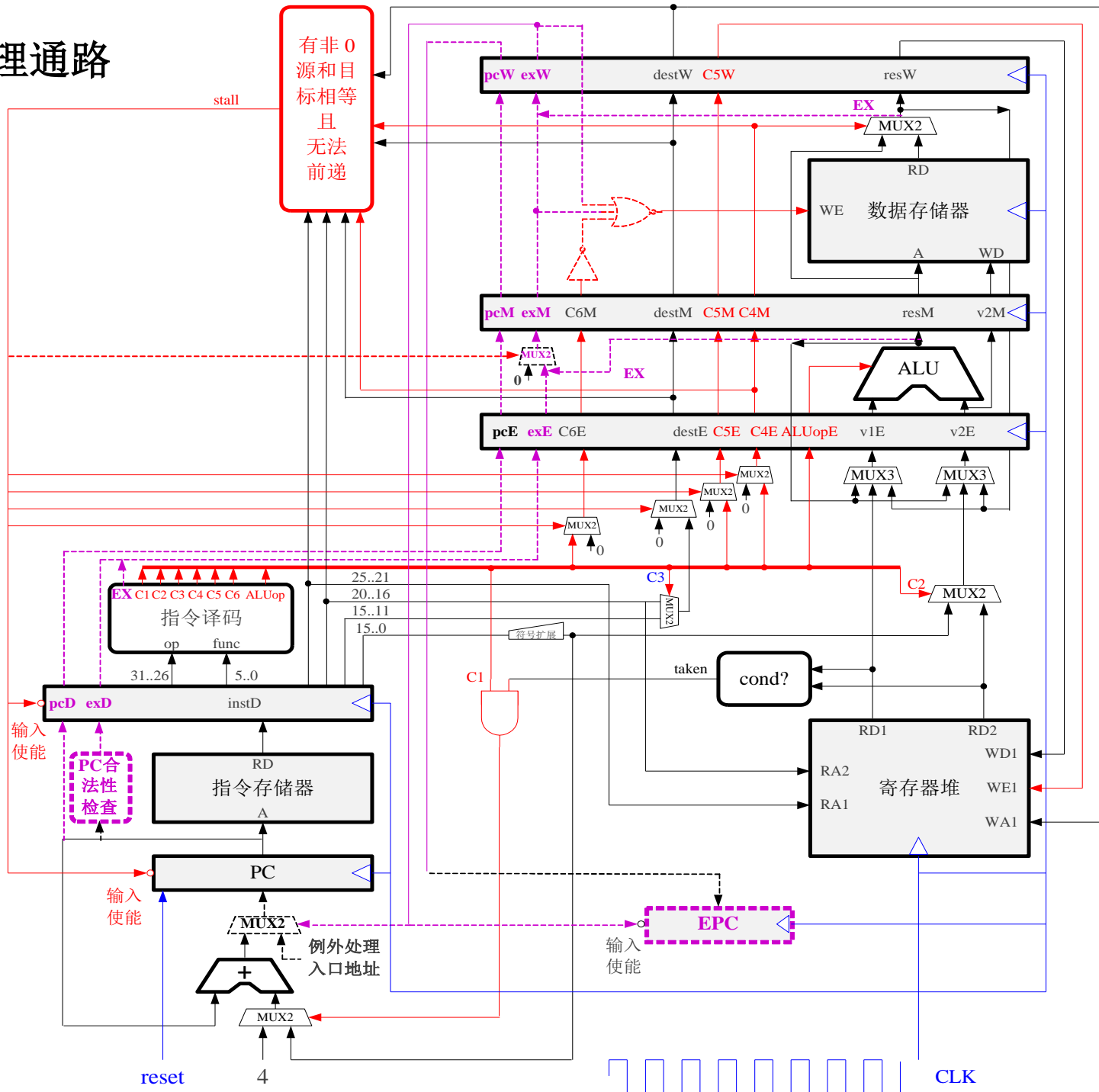


• ALU Forward控制

只考虑ALU  
前递，不包括访存和转  
移前递



• 例外处理通路

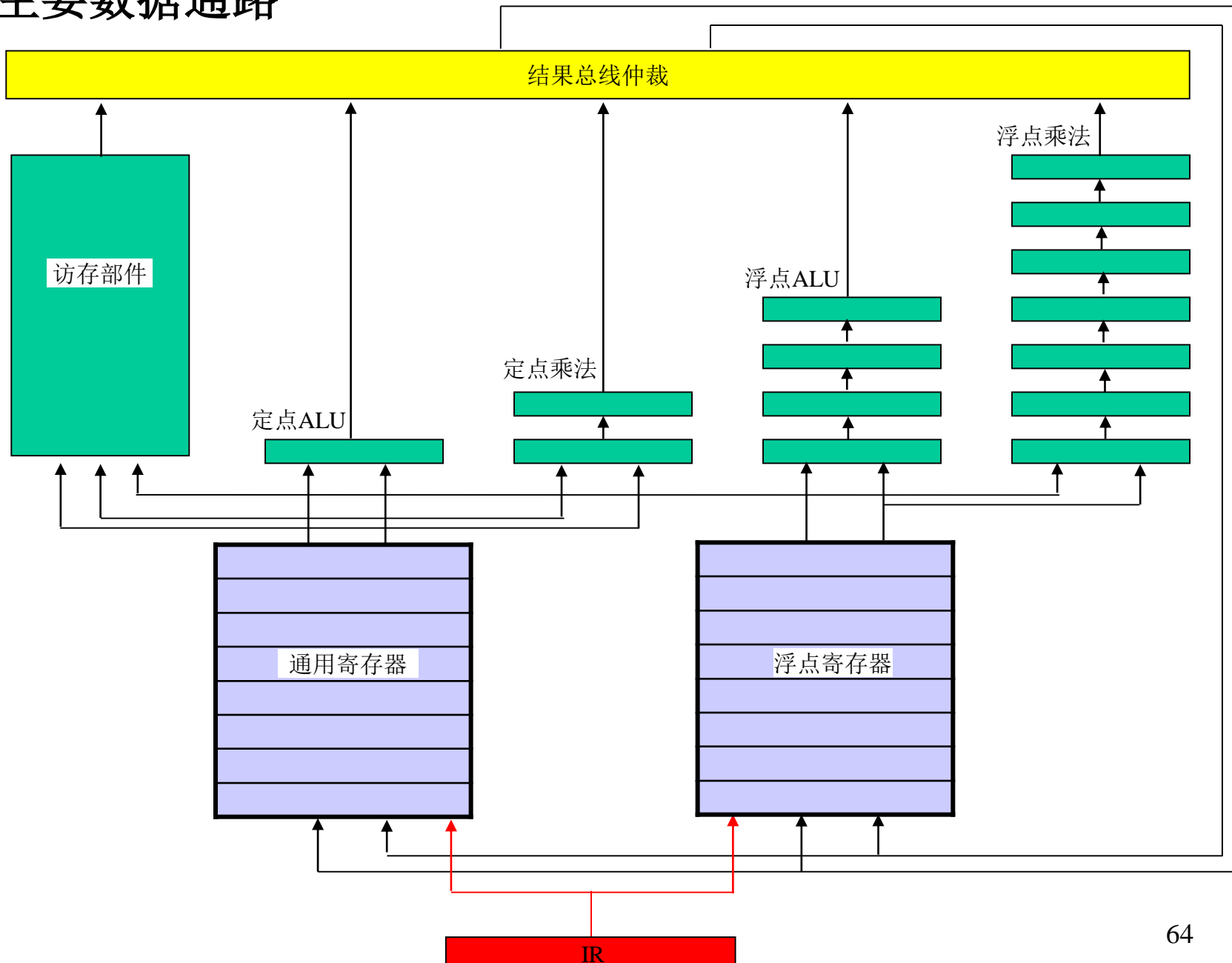


## 多功能部件与多拍操作

# 多功能部件与多拍操作

- 多功能部件
  - 定点ALU、定点乘法、浮点ALU、浮点乘法、访存
- 不同功能部件有不同延迟
  - 定点ALU 1拍，定点乘法2拍，浮点ALU 4拍，浮点乘法7拍
  - 访存部件延迟不确定（CACHE不命中、访存总线竞争、动态存储器刷新等原因），每次只能一个操作（内部不流水）

- 主要数据通路





# 多功能部件及多拍操作引起的问题（1）

- 结构相关

- 访存部件不流水引起多个访存操作的等待
- 结果总线相关：不同功能部件延迟不一致，定点ALU操作的MEM周期就是为了避免结果总线冲突的stall操作

LD r1, 0(r2)	IF	ID	EX	MEM			WB			
LD r3, 0(r4)		IF	ID	EX	stall	stall	stall	MEM	WB	
ADD.S fr0, fr1, fr2			IF	ID	EX1	EX2	EX3	EX4	stall	WB

LD r1, 0(r2)	IF	ID	EX	MEM	WB
ADD r5, r3, r4		IF	ID	EX	WB

LD r1, 0(r2)	IF	ID	EX	MEM	WB	
ADD r5, r3, r4		IF	ID	EX	MEM	WB


# 多功能部件及多拍操作引起的问题（2）

- WAW相关
  - 即使有多个写端口也会阻塞
- 还是不会有WAR相关
  - 读数总是在较早的ID阶段

LD r1, 0(r2)	IF	ID	EX	MEM	WB
ADD r1, r3, r4		IF	ID	EX	WB

LD r1, 0(r2)	IF	ID	EX	MEM	WB	
ADD r1, r3, r4		IF	ID	EX	stall	WB

LD r1, 0(r2)	IF	ID	EX	MEM		WB
ADD r1, r3, r4		IF	ID	EX	WB	



LD r1, 0(r2)	IF	ID	EX	MEM			WB	
ADD r1, r3, r4		IF	ID	EX	stall	stall	stall	WB

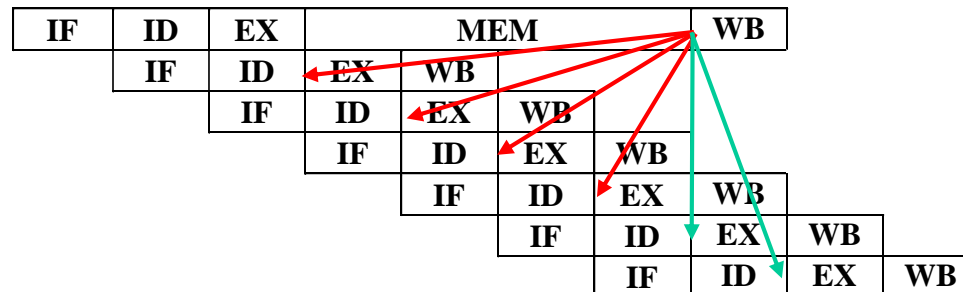
# 多功能部件及多拍操作引起的问题（3）

- RAW相关更多
  - 有些操作需要多拍产生结果
  - Forwarding技术的作用也很有限

LD r1, 0(r2)  
ADDI r3, r1, 1

IF	ID	EX	MEM	WB	
	IF	ID	stall	EX	WB

LD r1, 32(r0)  
ADDI r2, r1, 1  
ADDI r3, r1, 2  
ADDI r4, r1, 3  
ADDI r5, r1, 4  
ADDI r6, r1, 5  
ADDI r7, r1, 6



# 多功能部件及多拍操作引起的问题（4）

- 指令乱序结束
  - 例外处理更加困难，例外的发生难以预料
  - 在下面的例子中，没有任何相关，因此，ADD.D和SUB.D指令可以比DIV.D先结束。如果在ADD.D结束后DIV.D发生例外，此时无法恢复例外现场。

**DIV.D f0, f2, f4**

**ADD.D f10, f10, f8**

**SUB.D f12, f12, f14**

# 多功能部件及多拍操作流水线

- 通过流水线堵塞来保证避免数据冲突
  - 把所有流水线执行阶段指令的目标寄存器号和译码阶段的源寄存器号以及目标寄存器号进行比较，如果发现有寄存器号相等的情况就阻塞译码阶段的指令，这样就可以避免由于数据相关引起冲突
  - 要把译码阶段指令的目标寄存器号也跟前面指令的目标寄存器号进行比较以避免WAW相关导致冲突
- 结构相关
  - 通过请求和仲裁允许流水线前进。
- 例外处理
  - 不管指令要执行多少拍，都可以要求所有指令顺序写回，而且所有例外在写回阶段进行统一处理。
  - 有的早期机器不保证精确例外。

# 作 业