

第四章 处理器体系结构

4-4 ——流水线的实现基础

教 师：夏 文

计算机科学与技术学院

哈尔滨工业大学（深圳）

目 录

■ 流水线的通用原理

- 目标
- 难点

■ 设计流水化的Y86-64处理器-基础技术

- 调整SEQ
- 插入流水线寄存器
- 数据和控制冒险

真实世界的流行线：洗车

顺序



并行



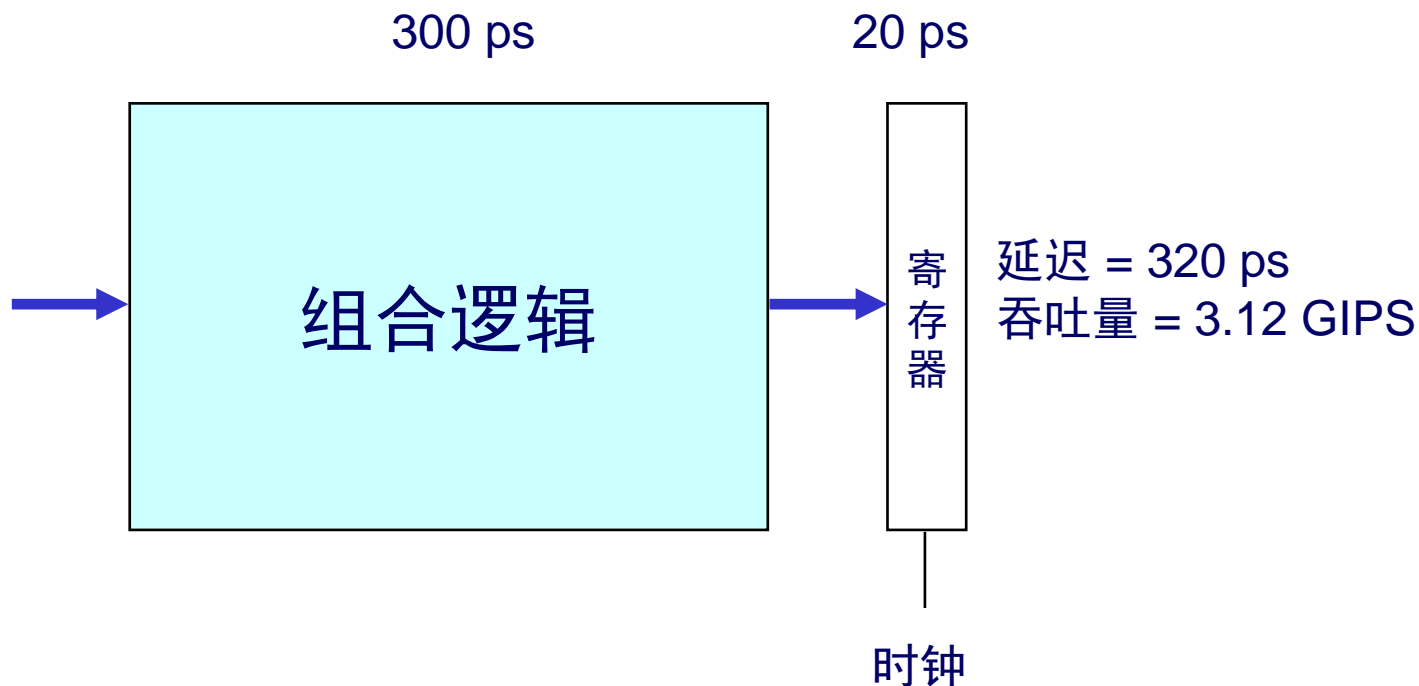
流水化



■ 思路：

- 把过程划分为几个独立的阶段
- 移动目标，顺序通过每一个阶段
- 在任何时刻，都会有多个对象被处理

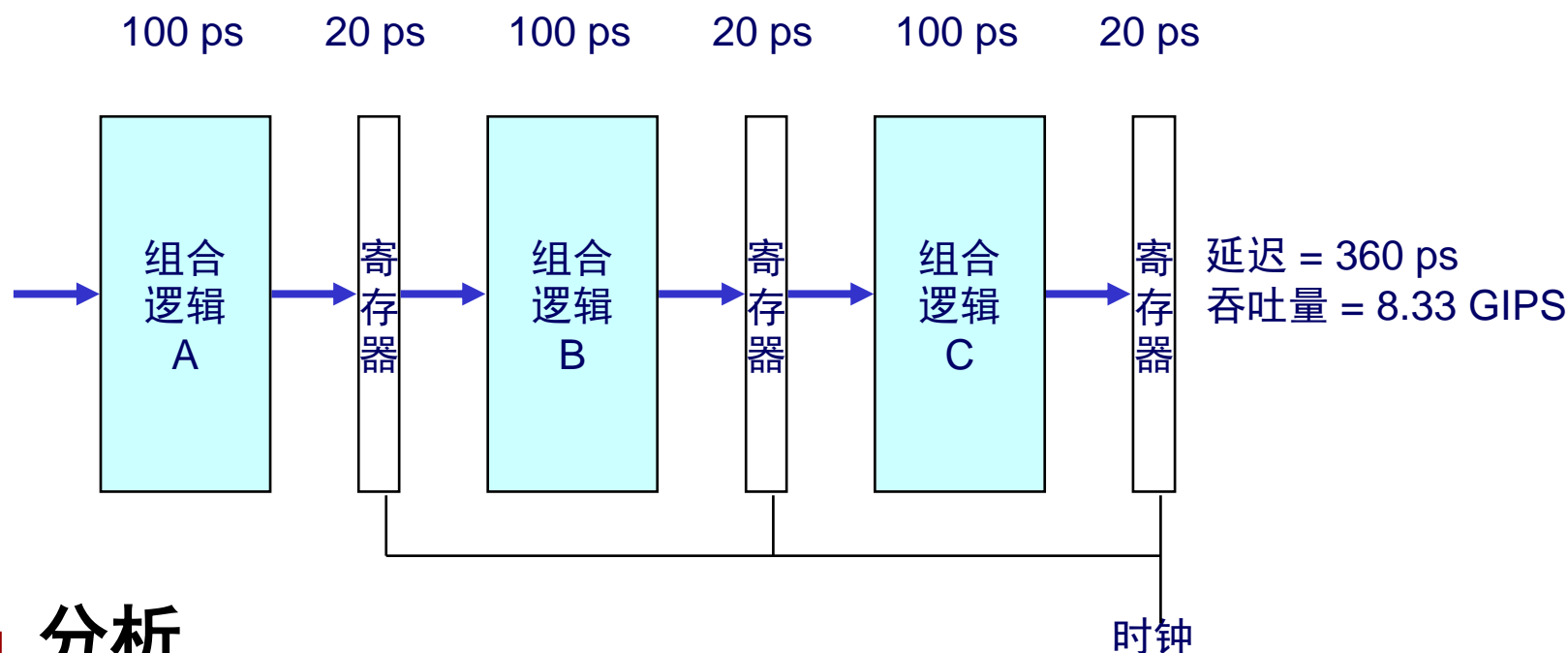
计算实例



■ 分析

- 计算需要300ps
- 将结果存到寄存器中需要20ps
- 时钟周期至少为320ps

3路 (3-Way) 流水线

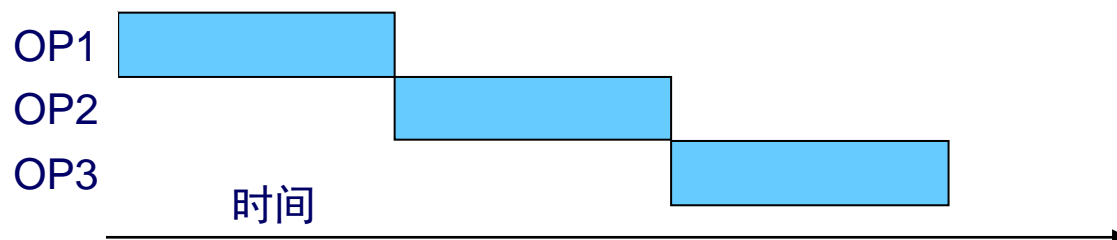


■ 分析

- 将计算逻辑划分为3个部分，每个部分100ps
- 当一个操作结束A阶段后，可以马上开始一个新的操作
 - 即每120 ps可以开始一个新的操作
- 整体延迟时间增加
 - 从开到结束一共360ps

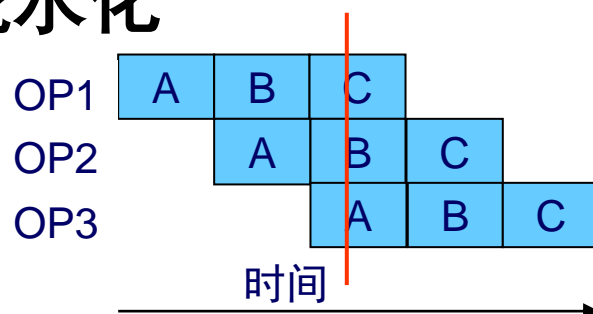
流水线图（一种时序图）

■ 未流水化



- 新操作只能在旧操作结束后开始

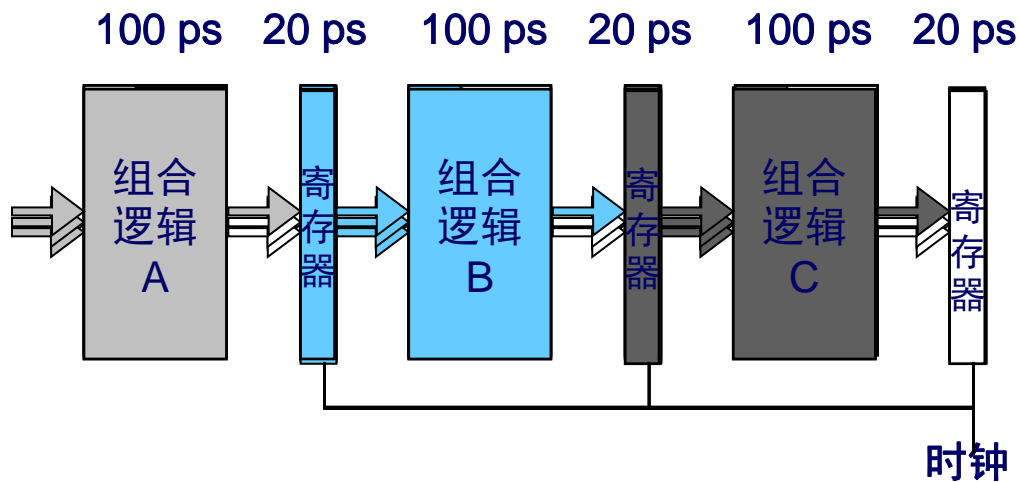
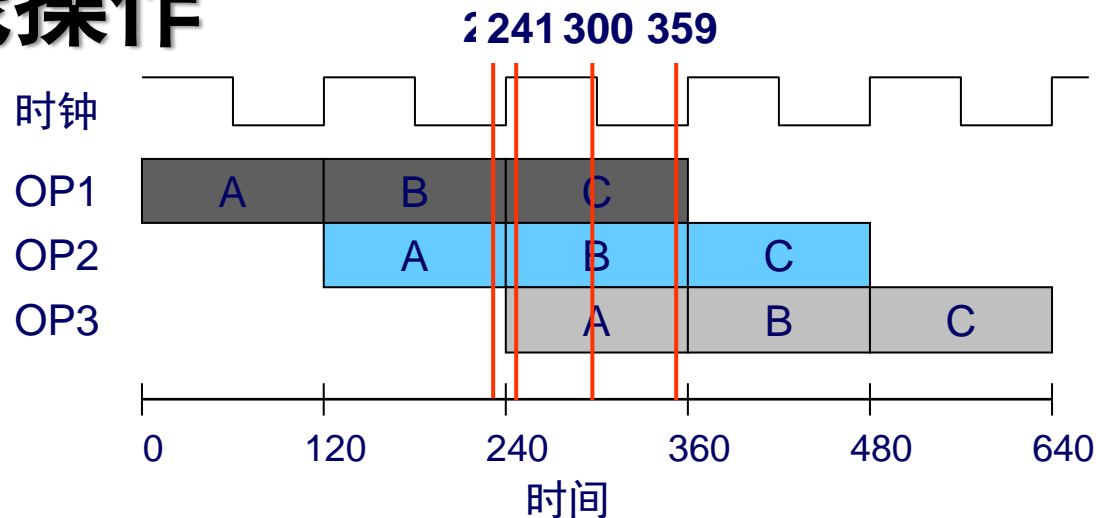
■ 3路流水化



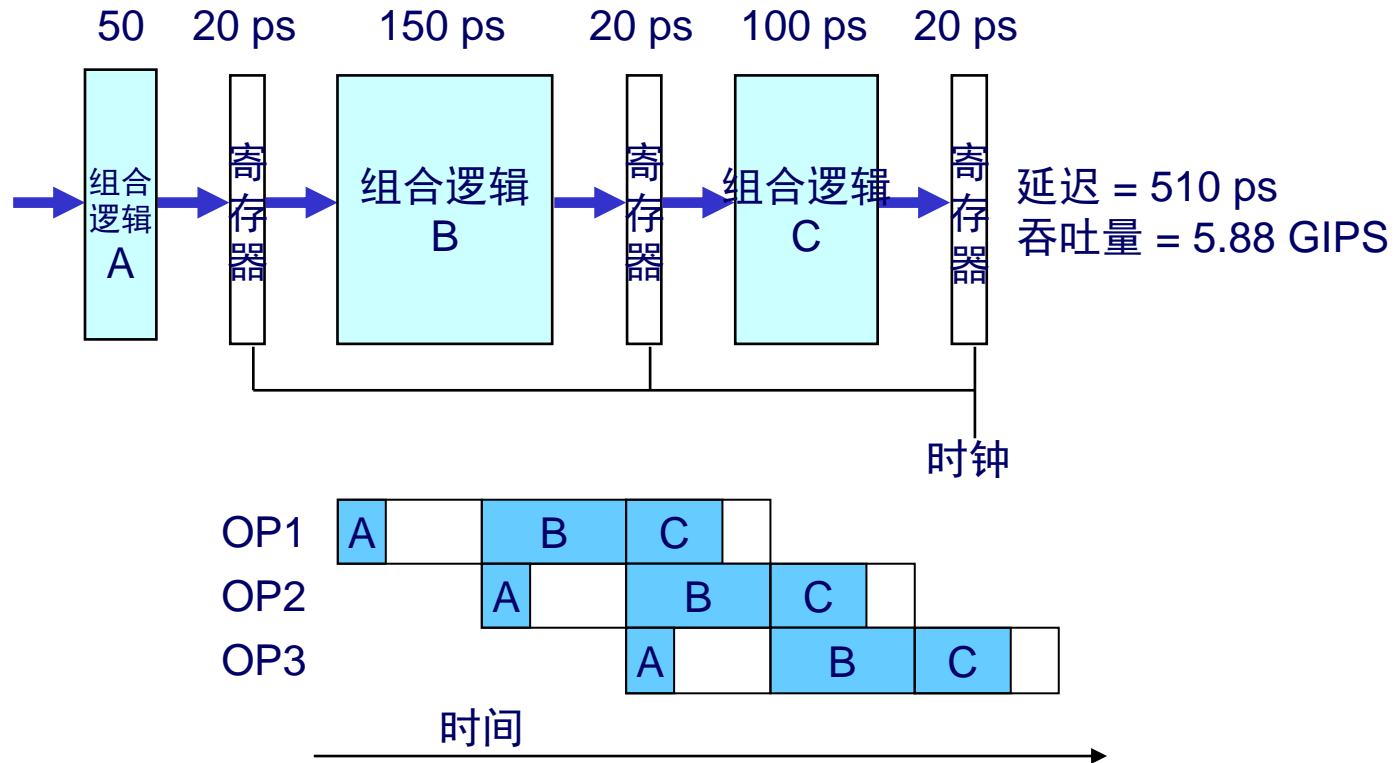
注意：横轴是时间，OP1才是最早进入流水线的，OP3最晚。（这与国内教材可能有很大不同）

- 可以同时处理至多3个操作

流水线操作

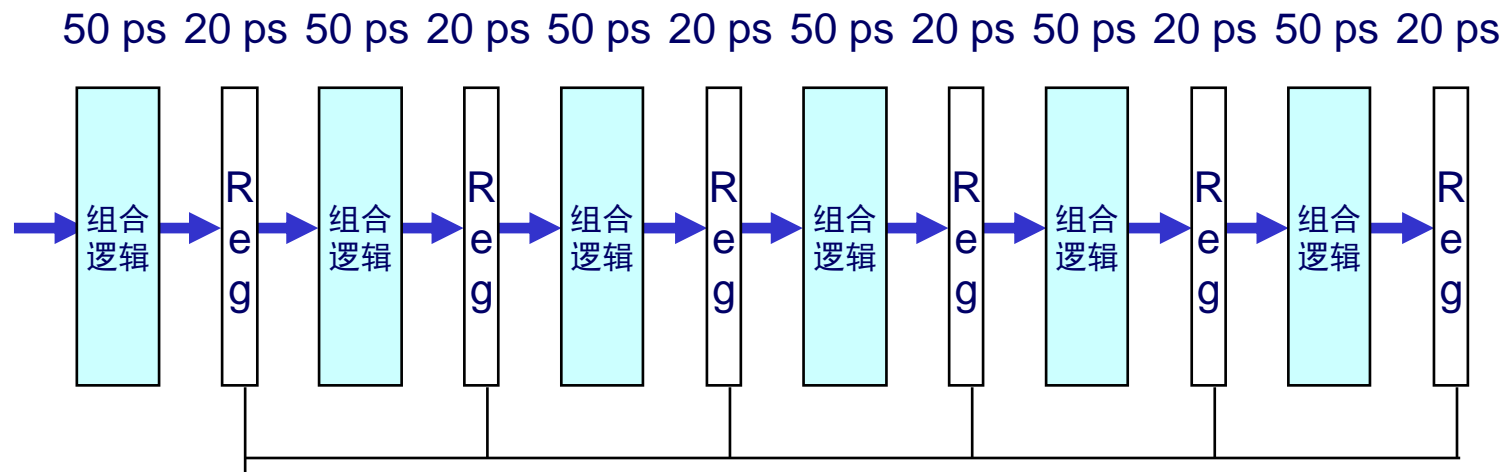


局限性：不一致的延迟



- 吞吐量由花费时间最长的阶段决定
- 其他阶段的许多时间都保持等待
- 将系统计算划分为一组具有相同延迟的阶段是一个严峻的挑战

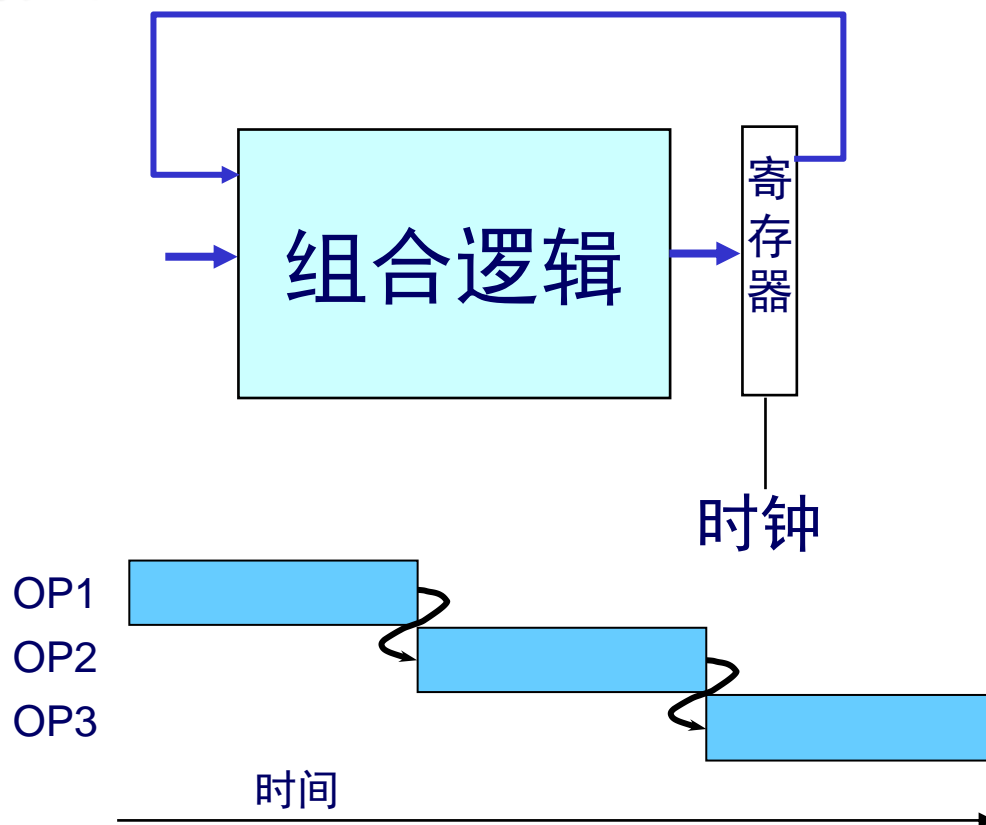
局限性：寄存器天花板



延迟 = 420 ps, 吞吐量 = 14.29 GIPS

- 当尝试加深流水线时，将结果载入寄存器的时间会对性能产生显著影响
- 载入寄存器的时间所占时钟周期的百分比：
 - 1阶段流水：6.25%
 - 3阶段流水：16.67%
 - 6阶段流水：28.57%
- 现代高速处理器具有很深的流水线，电路设计者必须很细心的设计流水线寄存器，使其延迟尽可能的小。

数据相关

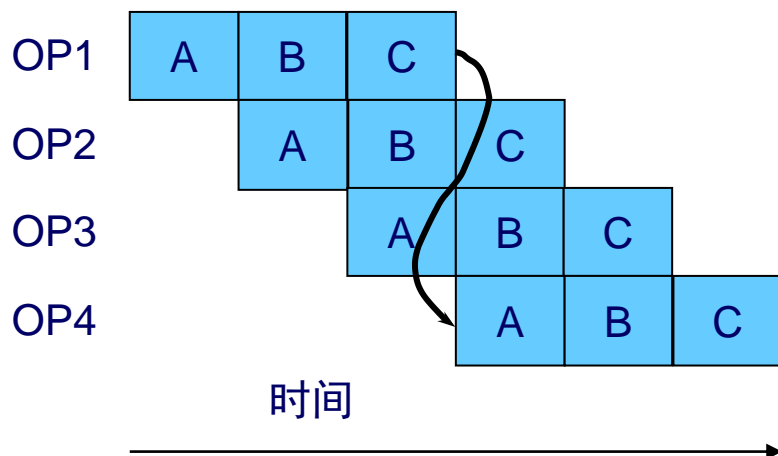
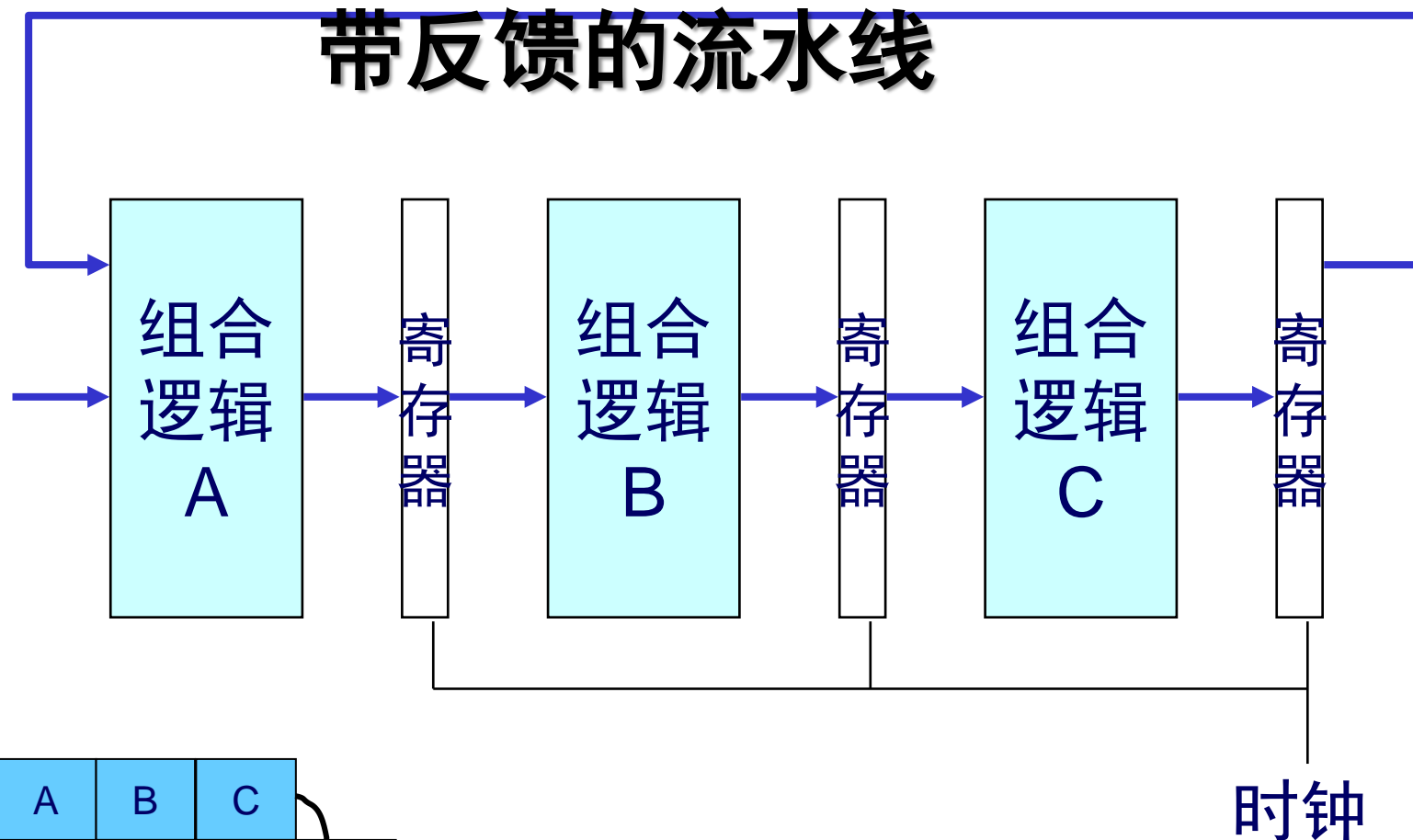


■ 分析

- 每个操作依赖于前一个操作的结果

带反馈的流水线

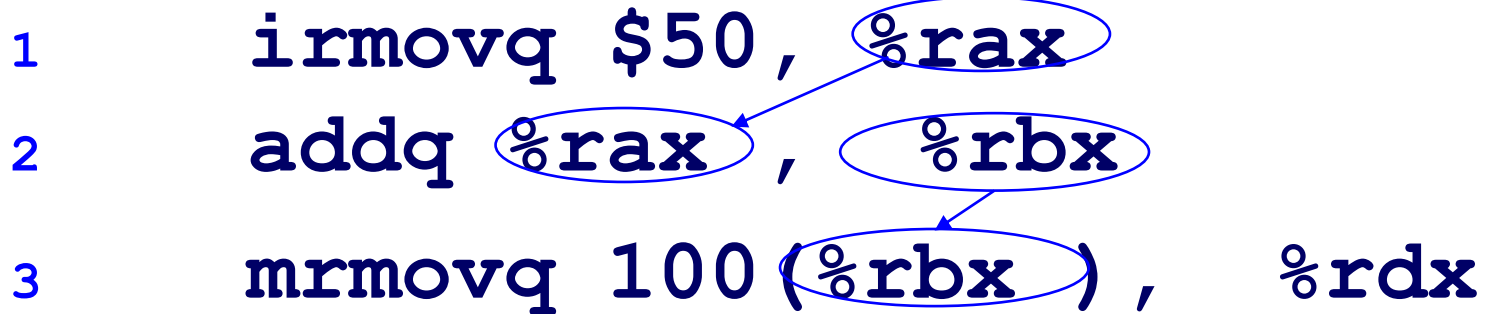
数据冒险



- 结果没有被及时地反馈给下一个操作
- 流水线改变了系统的行为

处理器中的数据相关

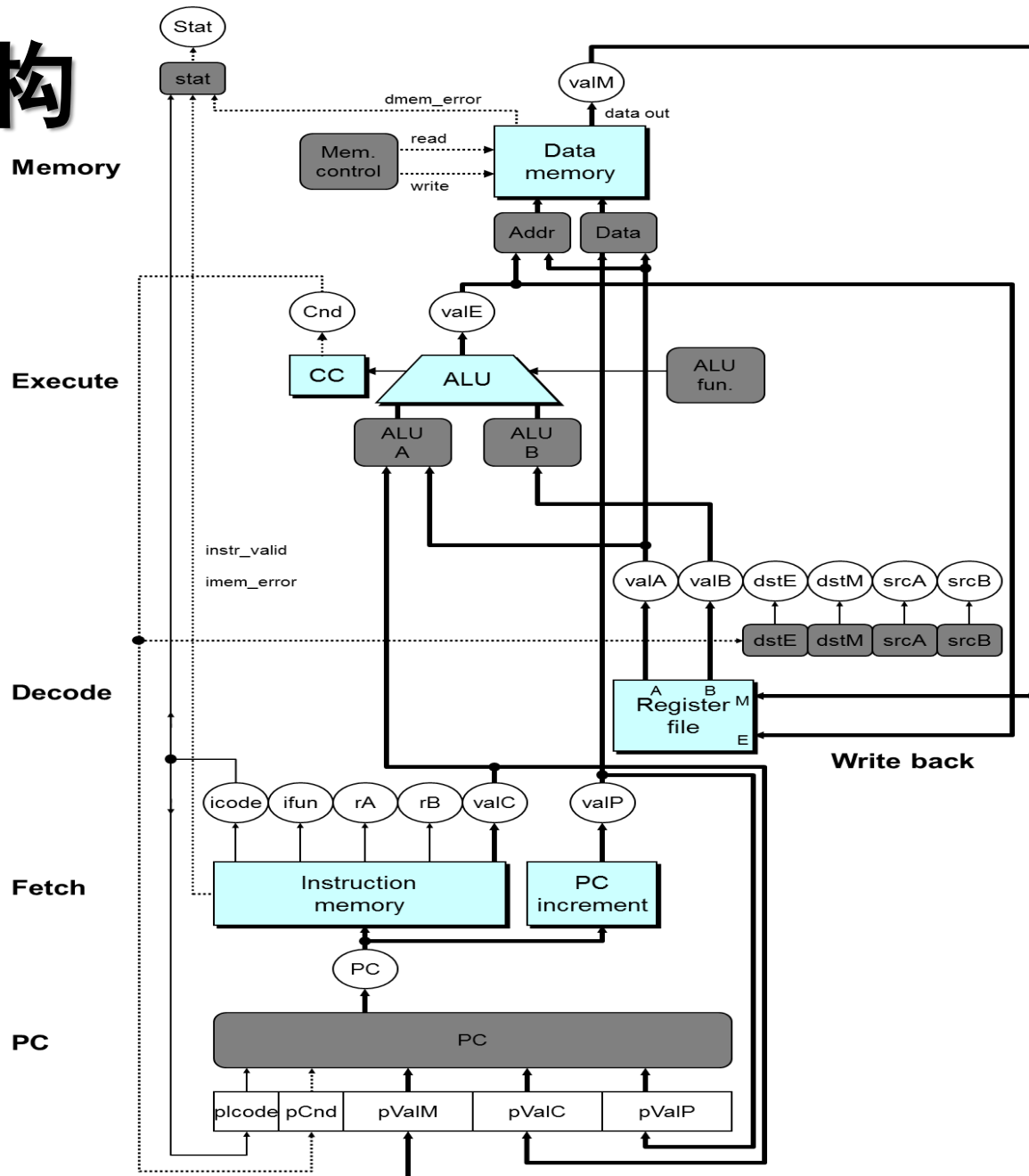
```
1      irmovq $50, %rax
2      addq %rax, %rbx
3      mrmovq 100(%rbx), %rdx
```



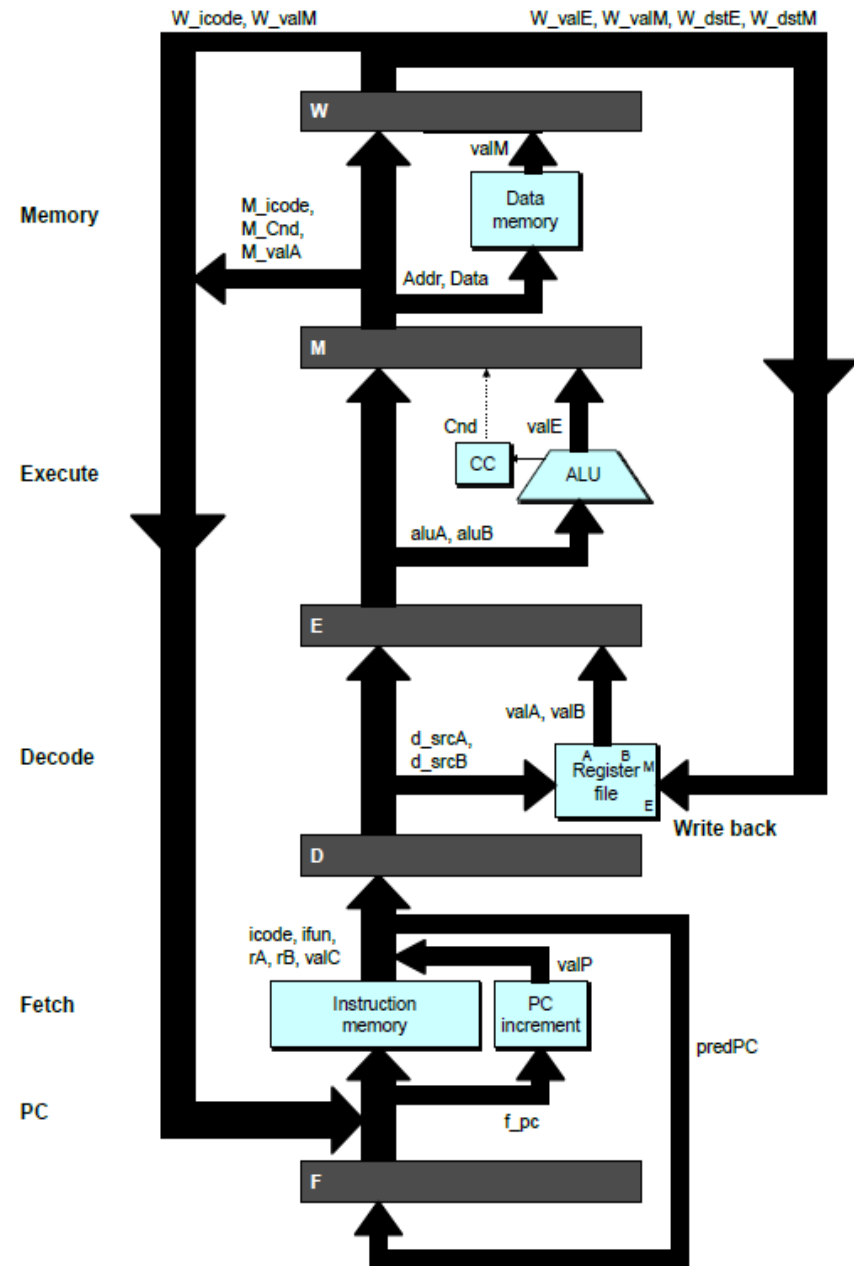
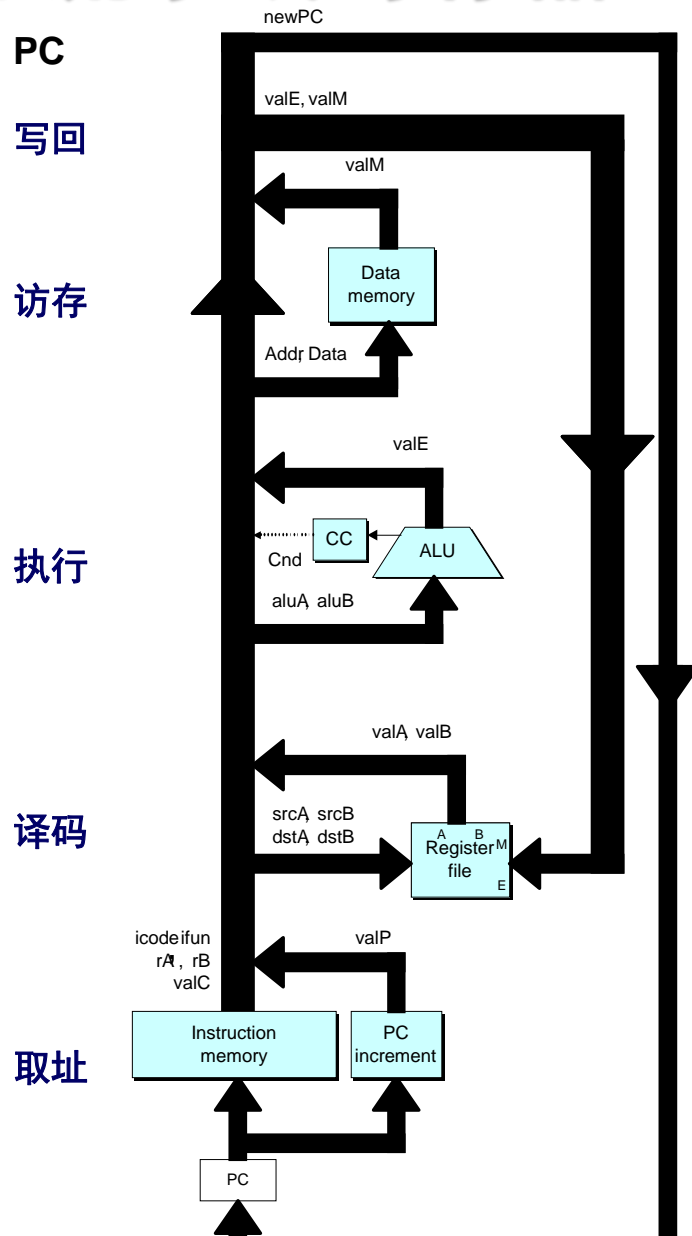
- 一条指令的结果作为另一条指令的操作数
 - 读后写数据相关
- 这些现象在实际程序中很常见
- 必须保证我们的流水线可以正确处理：
 - 得到正确的结果
 - 最小化对性能的影响

SEQ+ 的硬件结构

- 顺序实现
- 重启PC阶段放在开始
- PC 阶段
 - 选择PC执行当前指令
 - 根据前一条指令的计算结果
- 处理器状态
 - PC不再保存在寄存器中
 - 但是，可以根据其他信息决定PC



添加流水线寄存器



流水线阶段

■ 取指

- 选择当前PC
- 读取指令
- 计算PC的值

■ 译码

- 读取程序寄存器

■ 执行

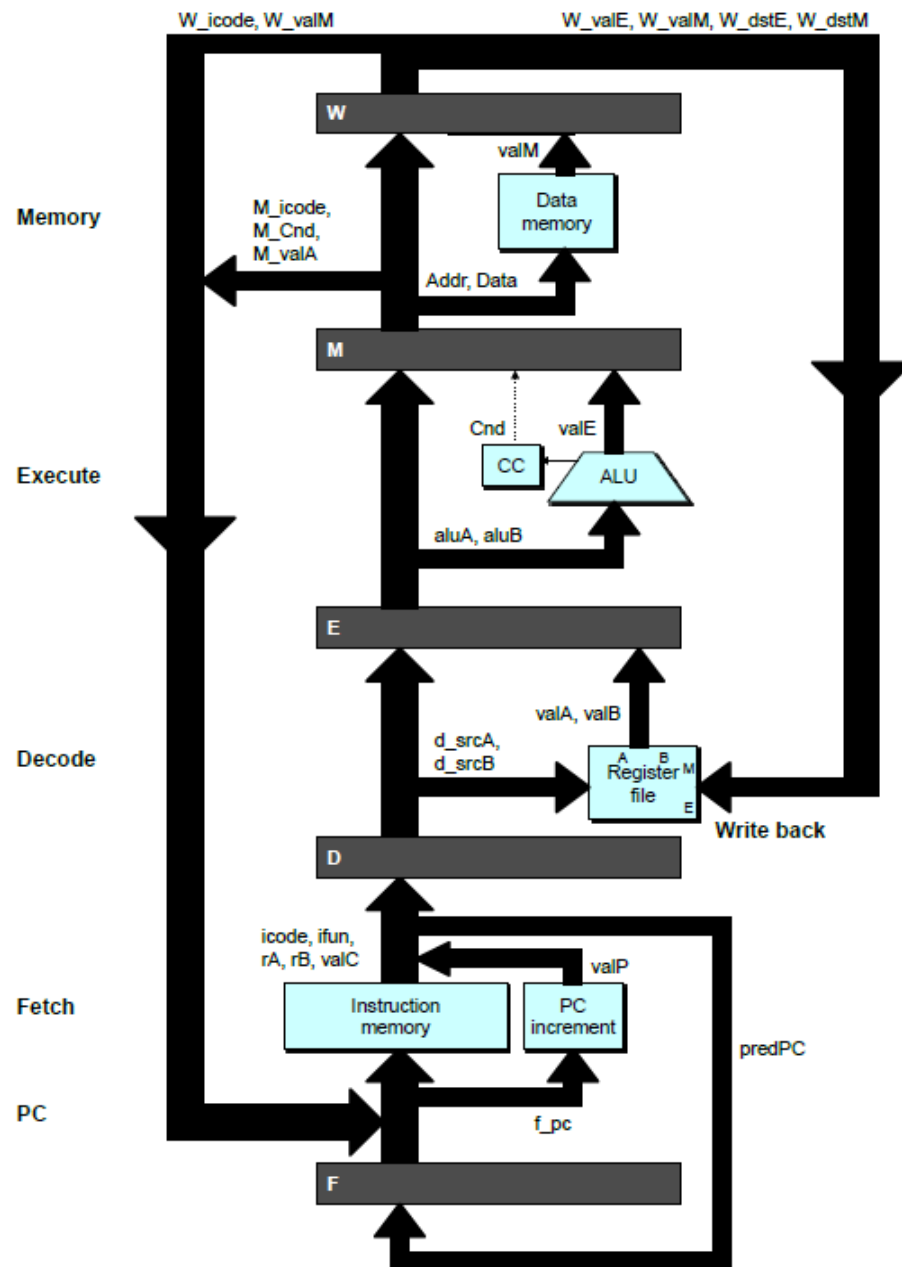
- 操作ALU

■ 访存

- 读或写存储器

■ 写回

- 更新寄存器文件



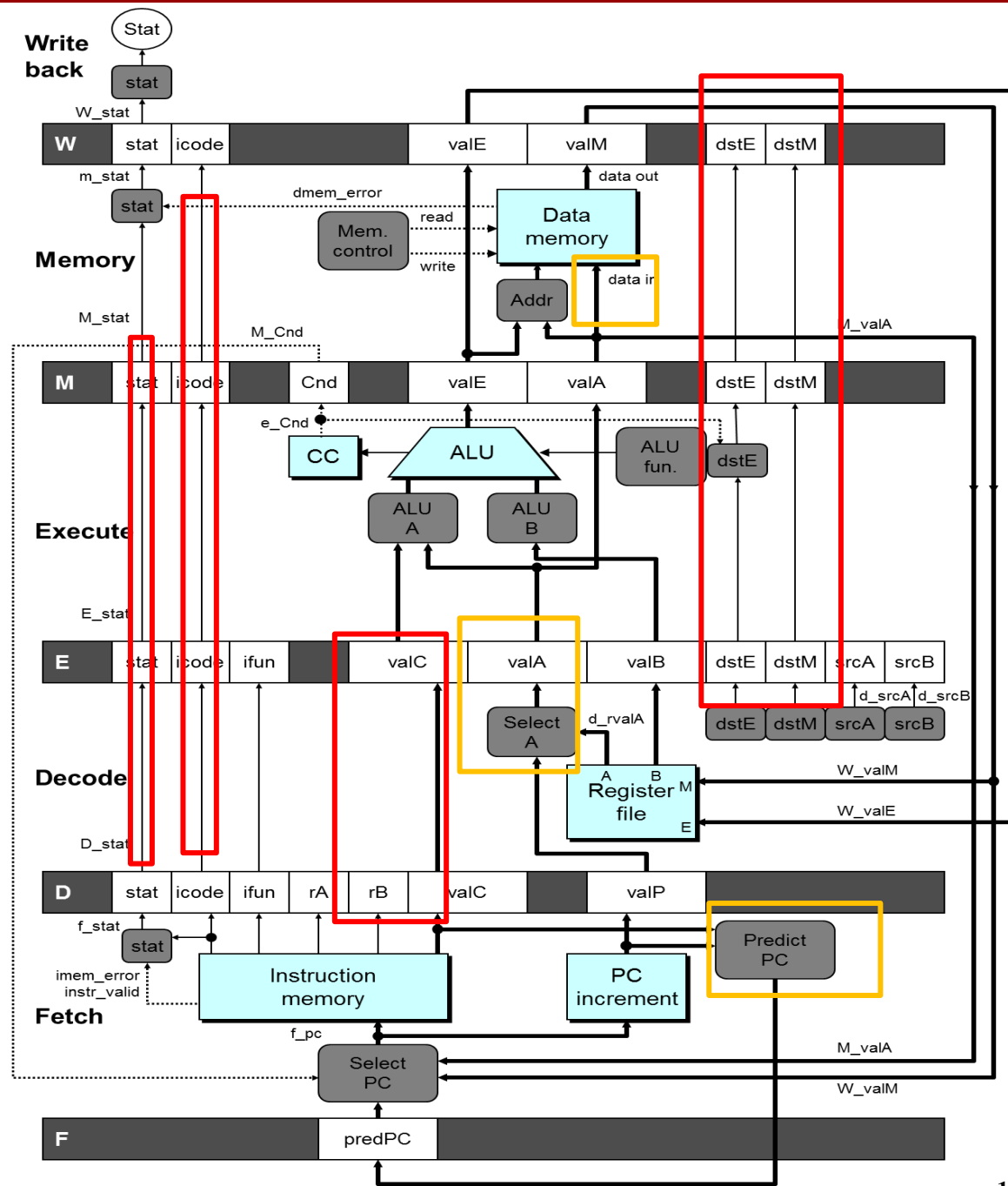
PIPE- 硬件结构

- 流水线寄存器保存指令执行的中间值

■ 前向路径

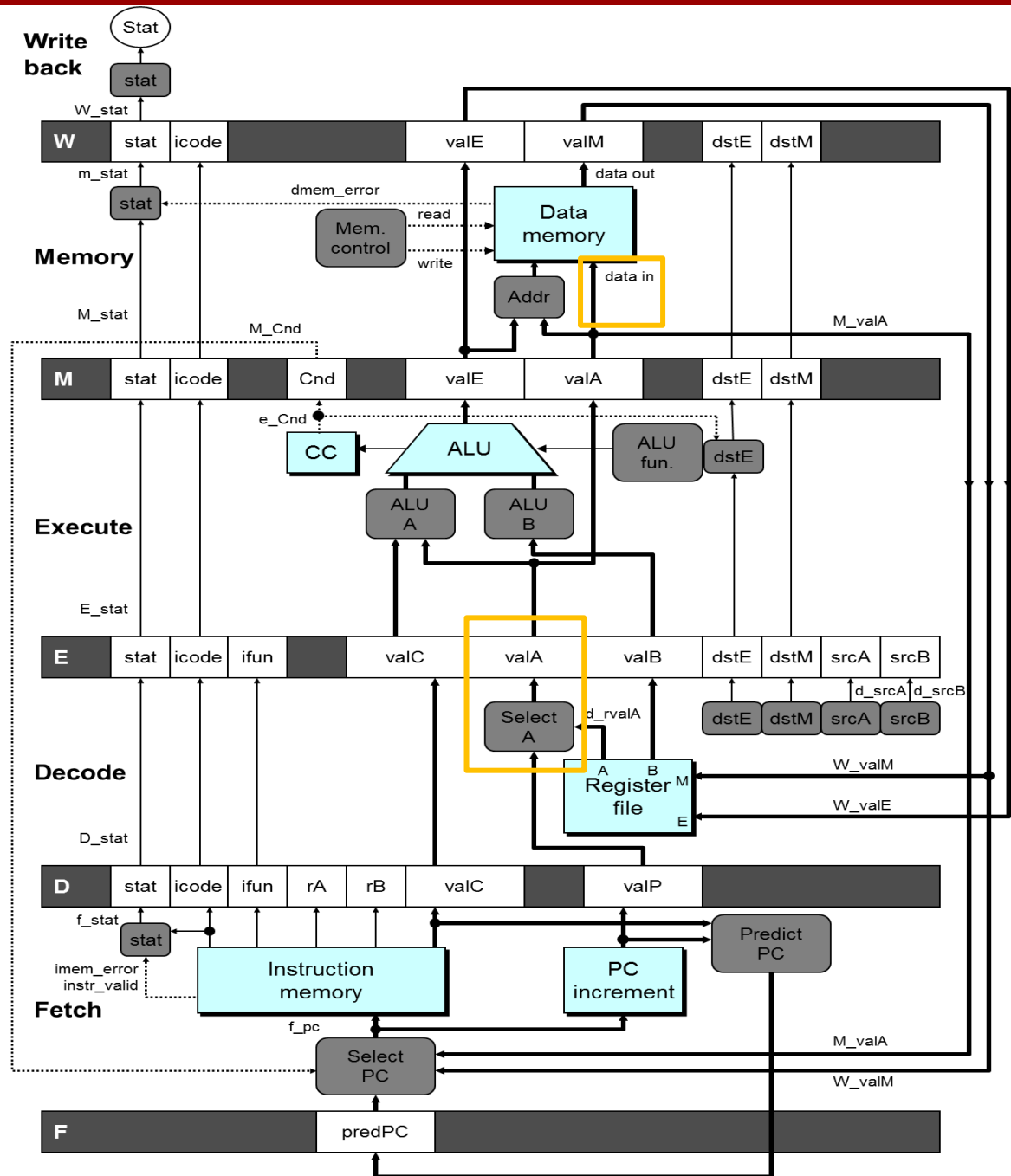
- 值从一个阶段送到下一个阶段
- 不能跳到过去的阶段
 - 如valC 通过译码阶段

■ 简化结构



除了call指令在访存阶段用到了valP，其他指令在执行、访存和写回阶段都没有用到valP。Call指令没有用到valA，因此不妨将valA和valP复用（通过select A选择到底使用valA还是valP）。

指令	call Dest
取指	icode:ifun \leftarrow M ₁ [PC]
	valC \leftarrow M ₈ [PC+1]
	valP \leftarrow PC+9
译码	
	valB \leftarrow R[%rsp]
执行	valE \leftarrow valB+(-8)
访存	M ₈ [valE] \leftarrow valP
写回	R[%rsp] \leftarrow valE
更新PC	PC \leftarrow valC



信号重新排列与命名规则

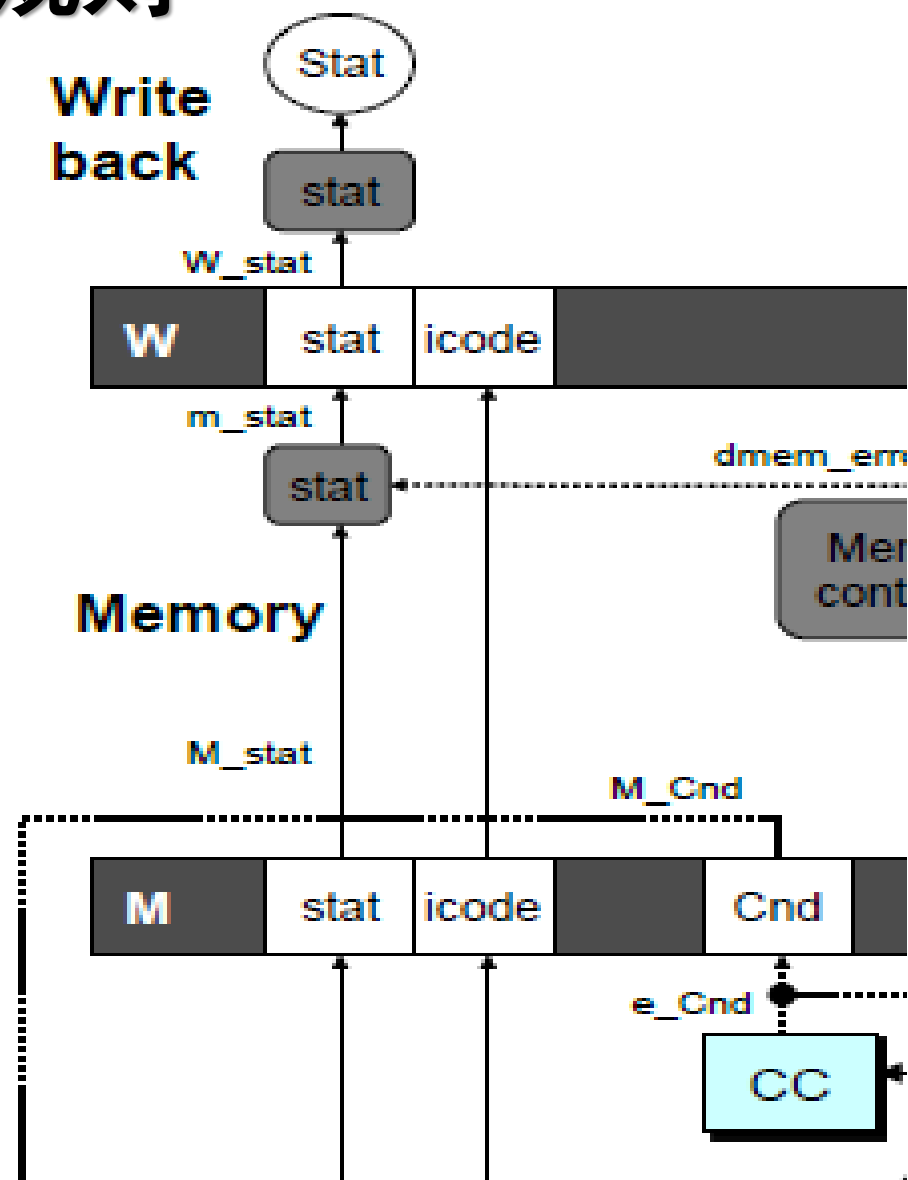
图中所有从寄存器中引出来的都是大写字母开头，其他为小写字母开头。

■ S_Field

- 流水线S阶段的寄存器的相关字段的名称用大写字母表示 F D E M W

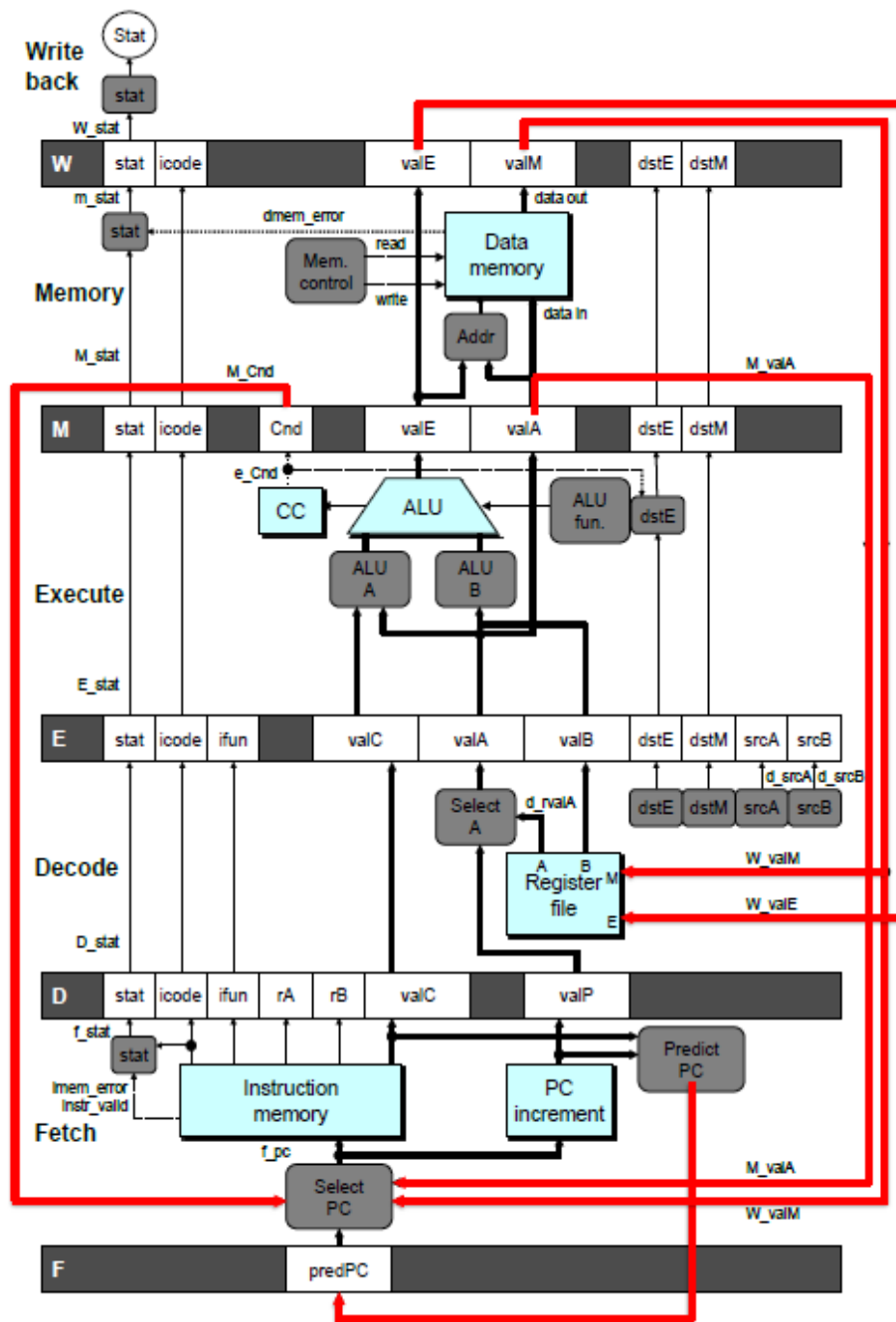
■ s_Field

- 流水线S阶段的相关字段的相关值用小写字母表示 f d e m w



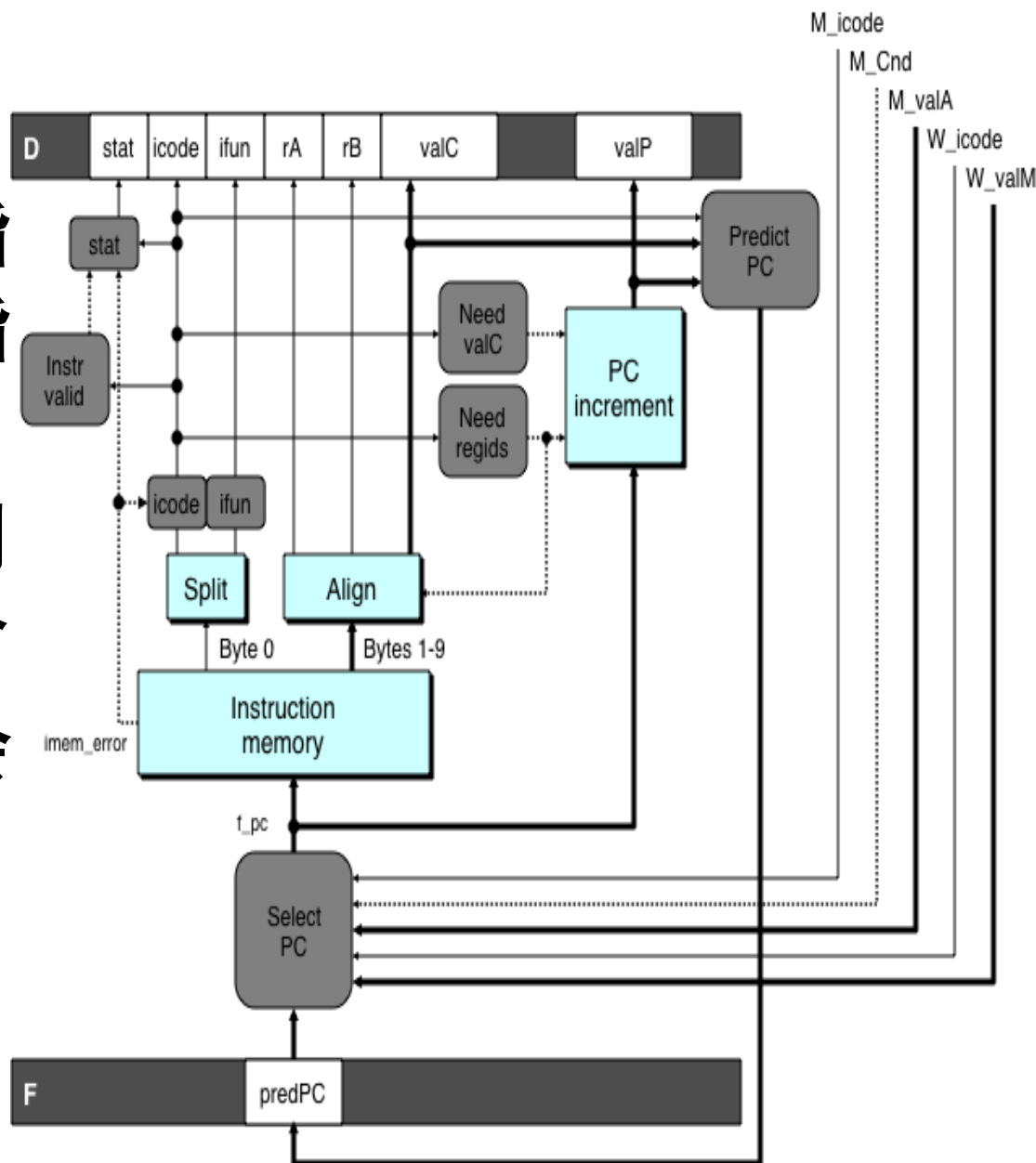
反馈路径

- 预测下一个PC
 - 猜测下一个PC的值
- 分支信息
 - 跳转或不跳转
 - 预测失败或成功
- 返回点
 - 从内存中读取
- 寄存器更新
 - 通过寄存器文件写端口



预测PC

- 当前指令完成取指后，开始一条新指令的取指
 - 没有足够的时间决定下一条指令
- 猜测哪条指令将会被取出
 - 如果预测错误，就还原



预测策略

■ 非转移指令

- 预测PC为valP
- 永远可靠

■ 调用指令或无条件转移指令

- 预测PC为valC（调用的入口地址或转移目的地址）
- 永远可靠

■ 条件转移指令

- 预测PC为valC（转移目的地址）
- 如果分支被选中则预测正确
 - 研究表明成功率大约为60% ==回跳为valC更好

■ 返回指令

- 不进行预取 ==CPU硬件栈

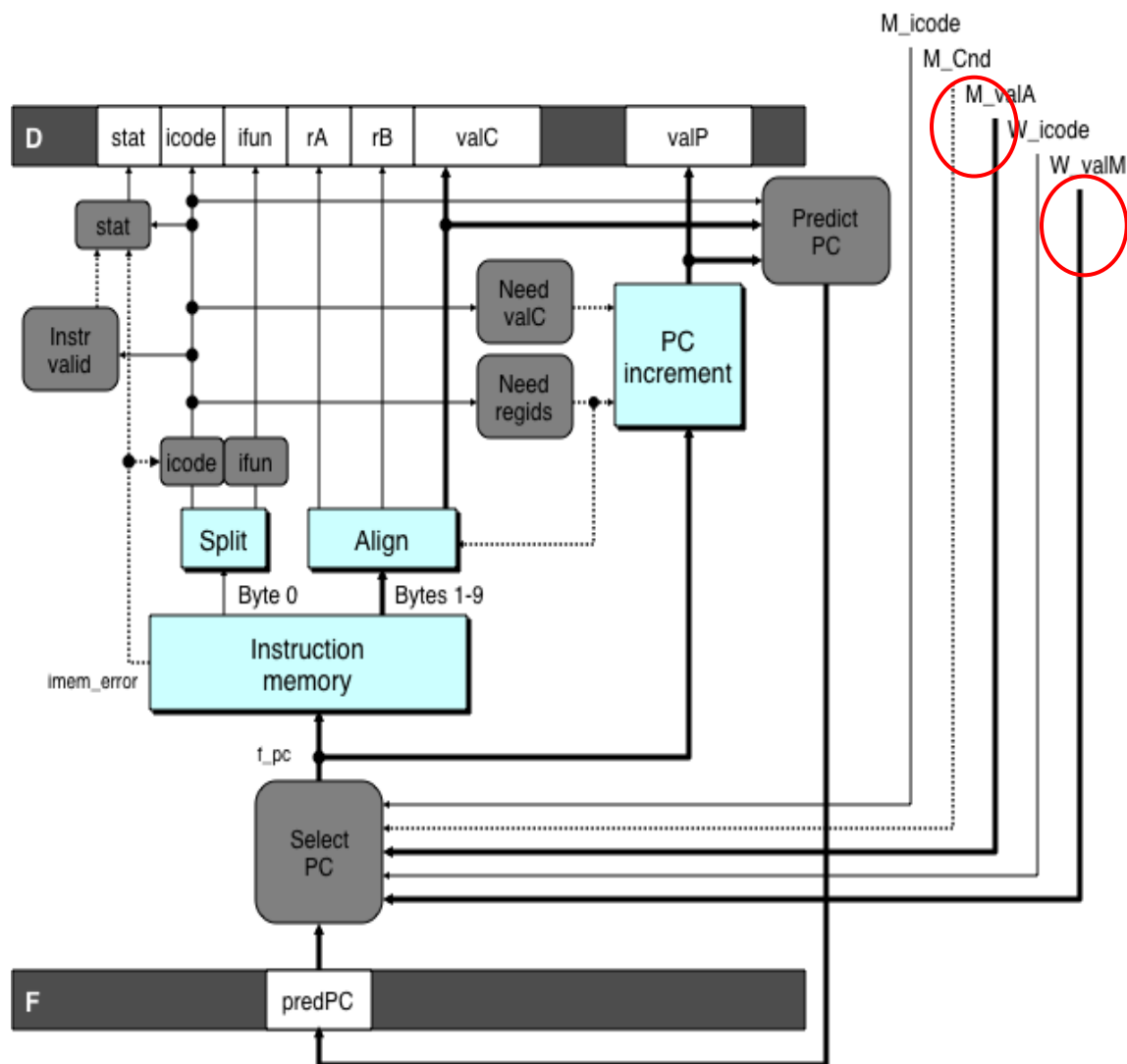
从预测错误中恢复

■ 跳转错误

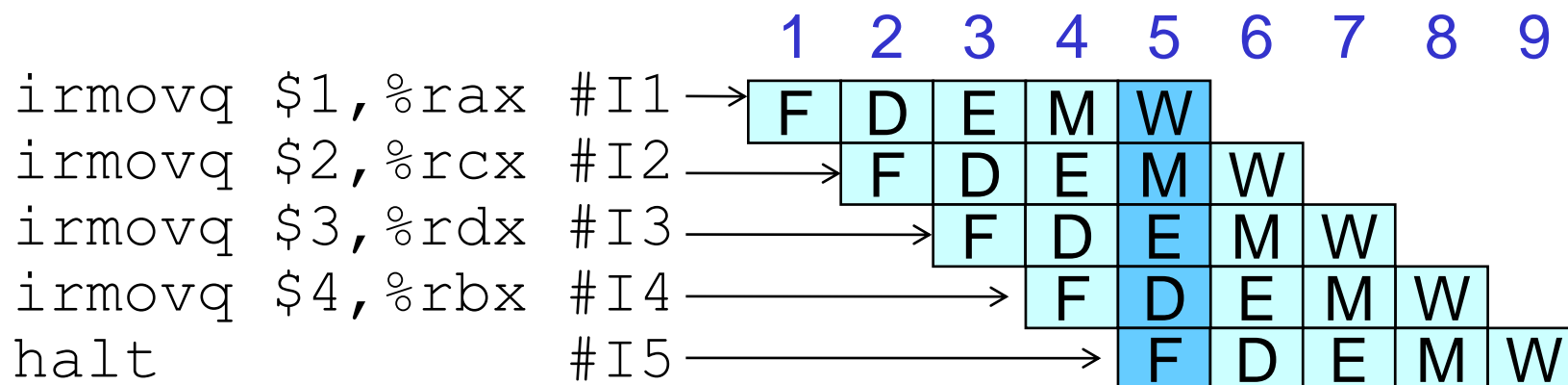
- 查看分支条件，如果指令进入访存阶段
- 从valA中得到失败的PC

■ 返回指令

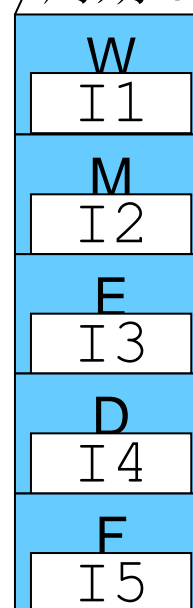
- 获取返回地址，当ret到达写回阶段



流水线示例



周期 5



■ File: demo-basic.js

F-Fetch-----取指

D-Decode-----译码

E-Execute----执行

M-Memory---访存

W-WriteBack-写回

(更新PC在取值阶段完成)

数据相关: No Nop

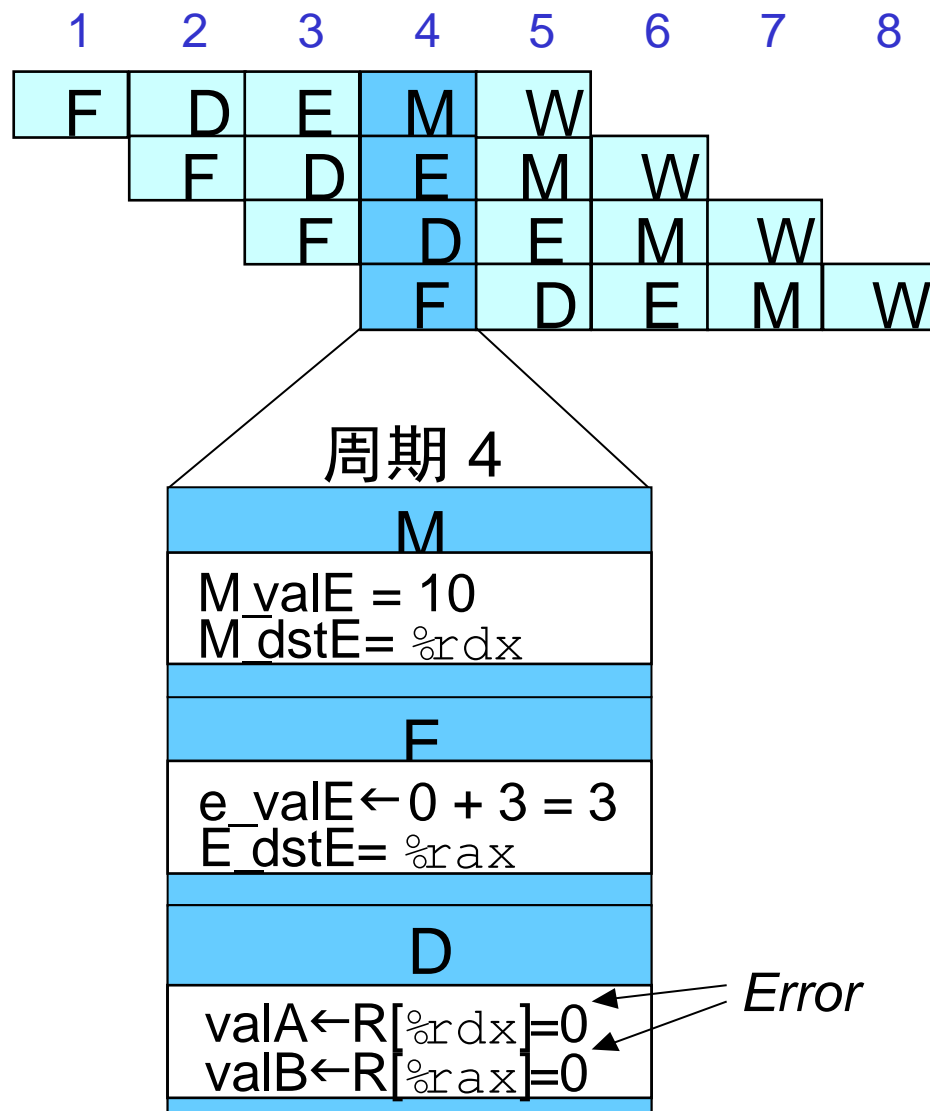
demo-h0.y

0x000:irmovq\$10,%rdx

0x00a:irmovq \$3,%rax

0x014:addq %rdx,%rax

0x016: halt



数据相关: 1 Nop

demo-h1.ys

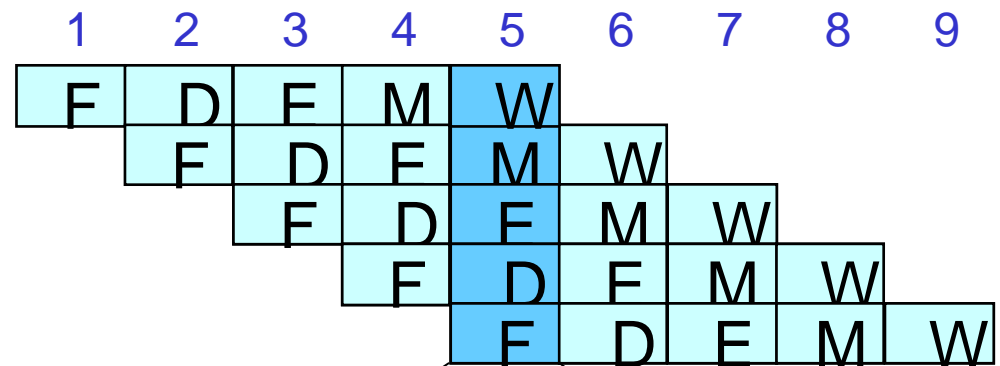
0x000:irmovq \$10,%rdx

0x00a:irmovq \$3,%rax

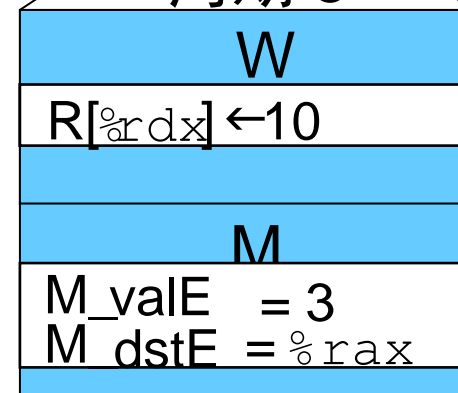
0x014:**nop**

0x015:addq %rdx,%rax

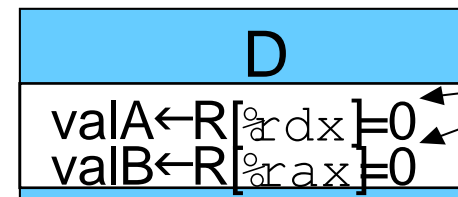
0x017: halt



周期 5



⋮



Error

数据相关: 2 Nop's

demo-h2.y_s

0x000:irmovq\$10,%rdx

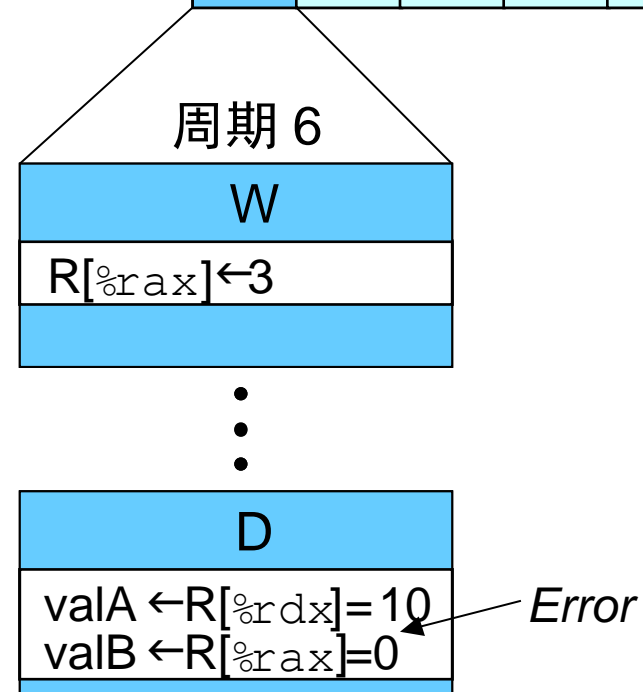
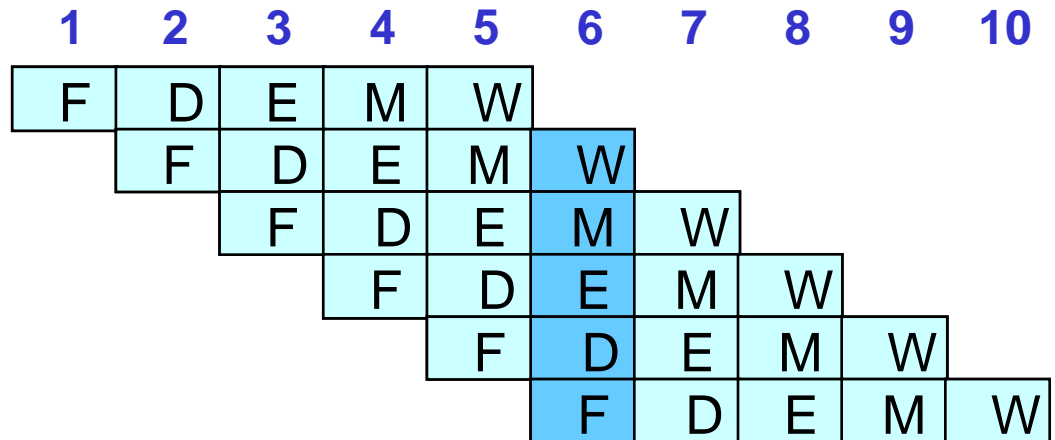
0x00a:irmovq \$3,%rax

0x014:nop

0x015:nop

0x016:addq %rdx,%rax

0x018: halt



数据相关: 3 Nop's

demo-h3.y

0x000: irmovq \$10, %rdx

0x00a: irmovq \$3, %rax

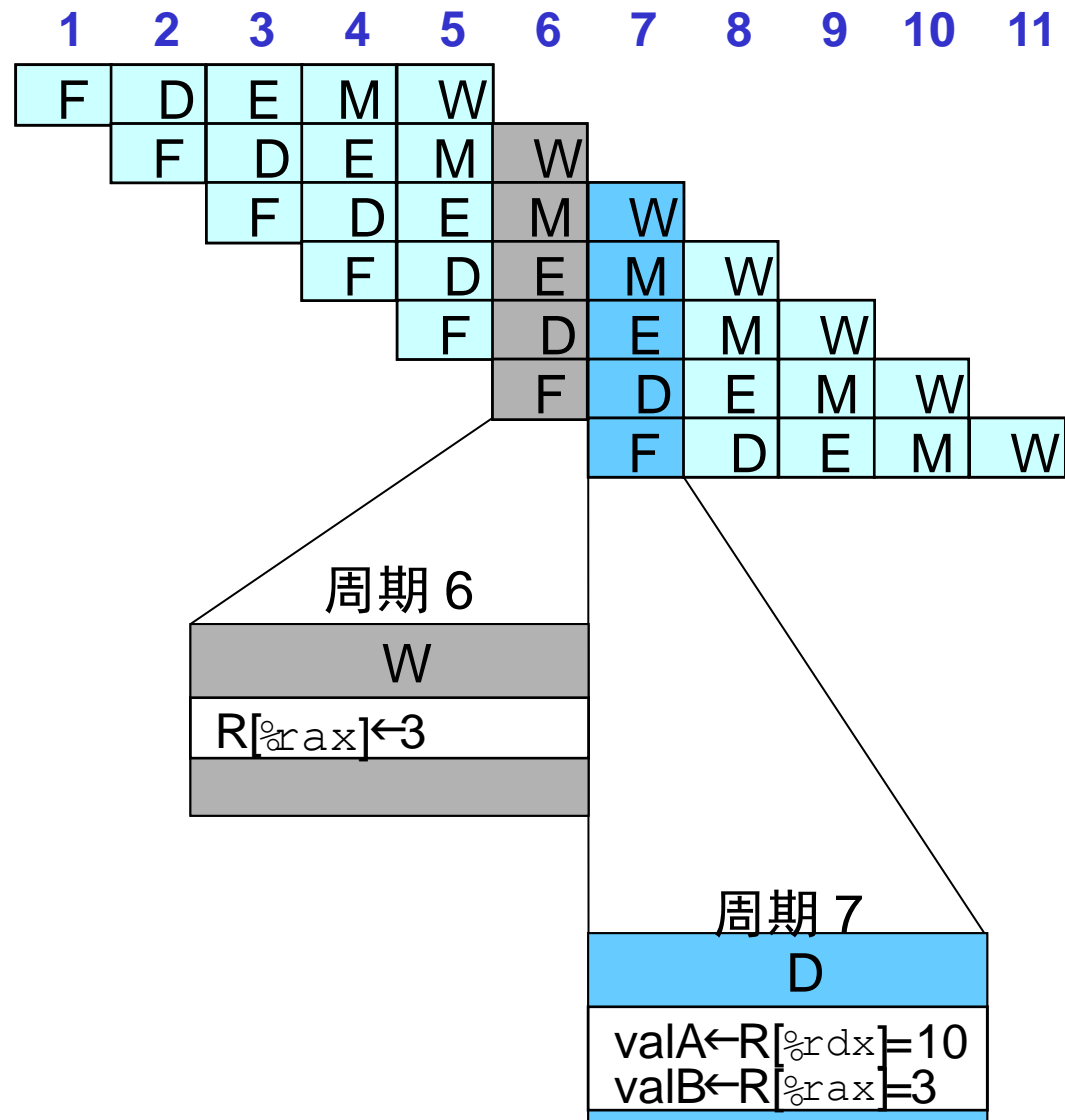
0x014: **nop**

0x015: **nop**

0x016: **nop**

0x017: addq %rdx, %rax

0x019: halt



分支预测错误示例

demo-j.ys

本应顺序执行，不跳转，但是分支预测错误，执行了跳转

```

0x000:    xorq %rax,%rax  #这里异或之后结果为0，ZF置1
0x002:    jne  t          #jne就是ZF为0就跳转，所以这里不跳转
0x00b:    irmovq $1, %rax  #分支预测错误，本应被执行，但是
                        未被执行

0x015:    nop
0x016:    nop
0x017:    nop
0x018:    halt            #退出程序
0x019:  t:  irmovq $3, %rdx  #跳转到的地址(本不应执行)
0x023:    irmovq $4, %rcx  #本不应执行
0x02d:    irmovq $5, %rdx  #本不应执行

```

- 应该只执行前7条指令

错误预测追踪

demo -j

0x000: `xorq %rax,%rax`

0x002: `jne t # Not taken`

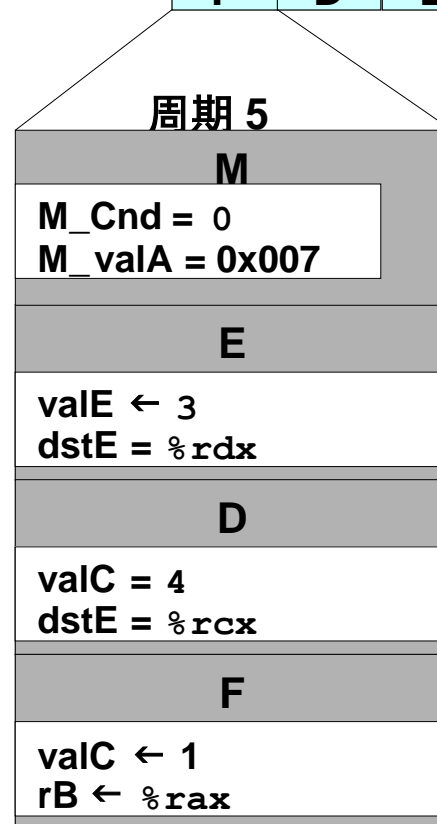
0x019: t: `irmovq $3, %rdx # Target`

0x023: `irmovq $4, %rcx # Target+1`

0x00b: `irmovq $1, %rax # Fall Through`

1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	E	M	W			
		F	D	E	M	W		
			F	D	E	M	W	
				F	D	E	M	W

- 在分支目标处，错误地执行了两条指令



返回示例

demo-ret.ys

```

0x000:    irmovq Stack,%rsp    # Intialize stack pointer
0x00a:    nop                    # Avoid hazard on %rsp
0x00b:    nop
0x00c:    nop
0x00d:    call p                  # Procedure call
0x016:    irmovq $5,%rsi          # Return point
0x020:    halt
0x020:    .pos 0x20
0x020: p: nop                  # procedure
0x021:    nop
0x022:    nop
0x023:    ret
0x024:    irmovq $1,%rax          # Should not be executed
0x02e:    irmovq $2,%rcx          # Should not be executed
0x038:    irmovq $3,%rdx          # Should not be executed
0x042:    irmovq $4,%rbx          # Should not be executed
0x100:    .pos 0x100
0x100:    Stack:                  # Initial stack pointer

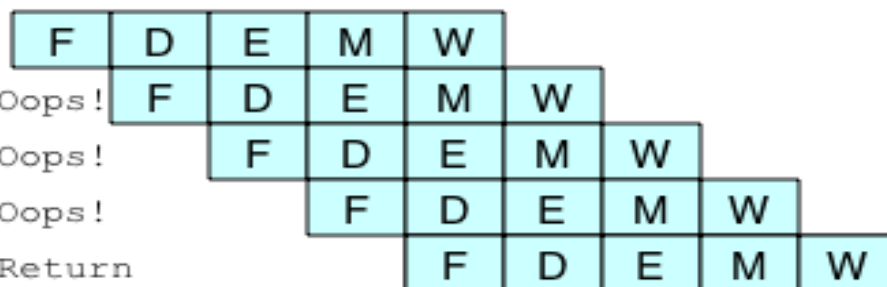
```

- 需要大量的nop指令来避免数据冒险

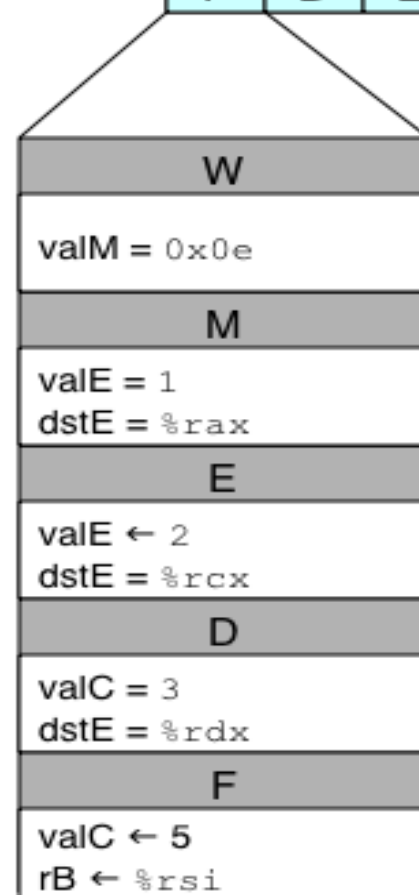
错误的返回示例

demo-ret

```
0x033:    ret
0x034:    irmovq $1,%rax # Oops!
0x03e:    irmovq $2,%rcx # Oops!
0x048:    irmovq $3,%rdx # Oops!
0x052:    irmovq $5,%rsi # Return
```



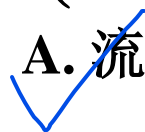
- 在ret之后，错误地执行了3条指令



习题

1.为了使计算机运行得更快，现代 CPU 采用了许多并行技术，将处理器的硬件组织成若干个阶段并让这些阶段并行操作的技术是（ ），该技术的 CPI 一般不小于1。

A. 流水线 B.超线程 C.超标量 D.向量机



流水线总结

■ 概念

- 将指令的执行划分为5个阶段
- 在流水化模型中运行指令

■ 局限性

- 当两条指令距离很近时，不能处理指令之间的（数据/控制)相关
- 数据相关
 - 一条指令写寄存器，稍后会有一条指令读寄存器
- 控制相关
 - 指令设置PC的值，流水线没有预测正确
 - 错误分支预测和返回

■ 改进流水线

- 下一节讲

Enjoy!