

第四章 处理器体系结构

——流水线的实现Part I

教 师： 郑贵滨
计算机科学与技术学院
哈尔滨工业大学

目录

- 流水线的通用原则
 - 目标
 - 难点
- 设计流水化的Y86-64处理器
 - 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



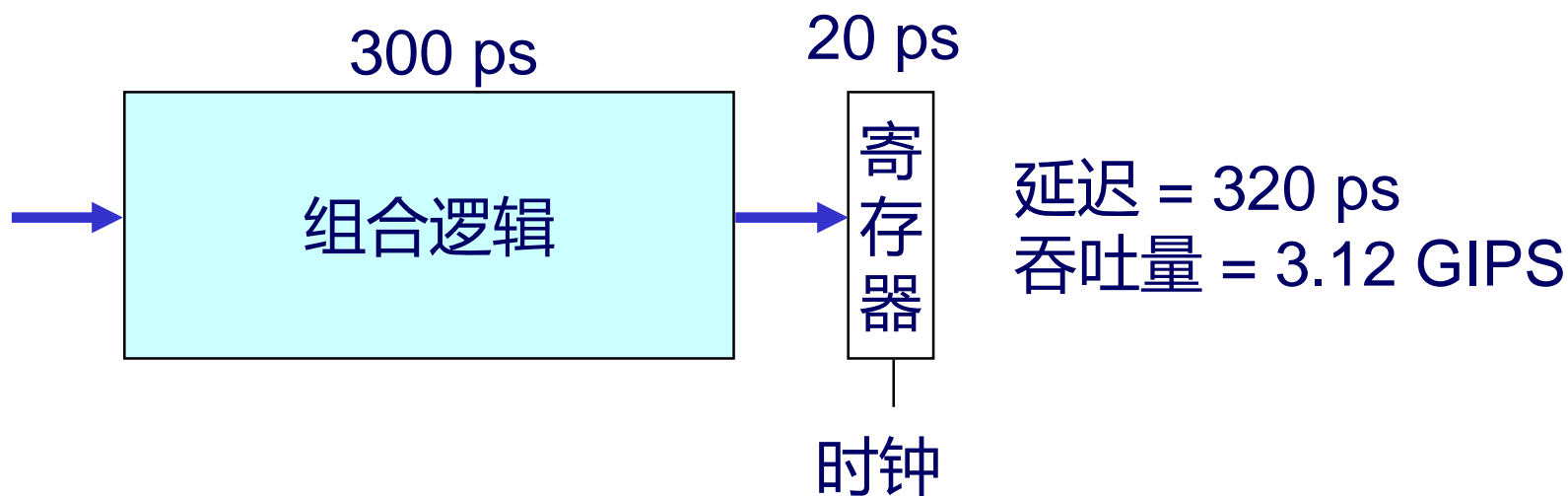
流水化



■ 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

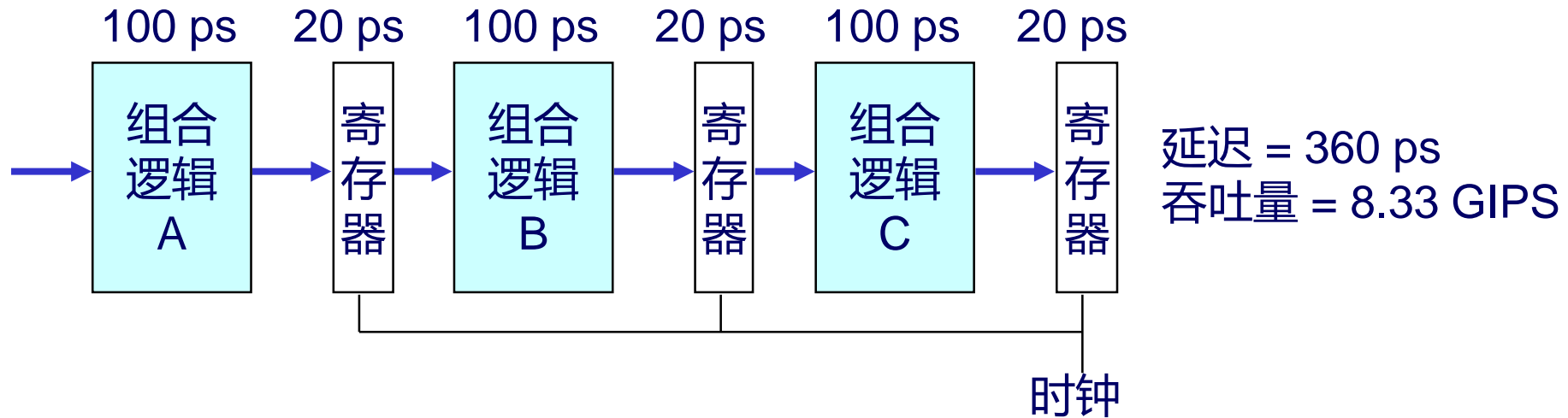
计算实例



■ 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阶段(3-Way)流水线

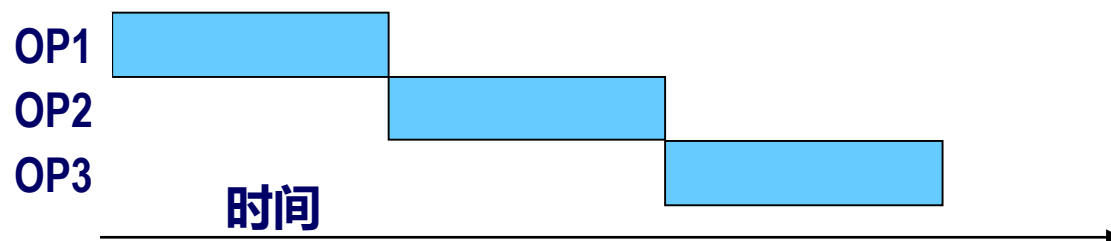


■ 分析

- 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

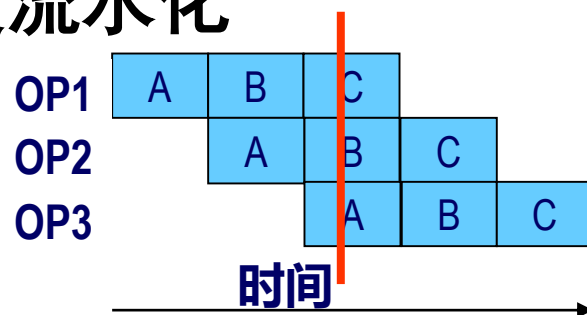
流水线图（一种时序图）

■ 未流水化



- 新操作只能在旧操作结束后开始

■ 3阶段流水化



- 可以同时处理最多3个操作

流水线操作

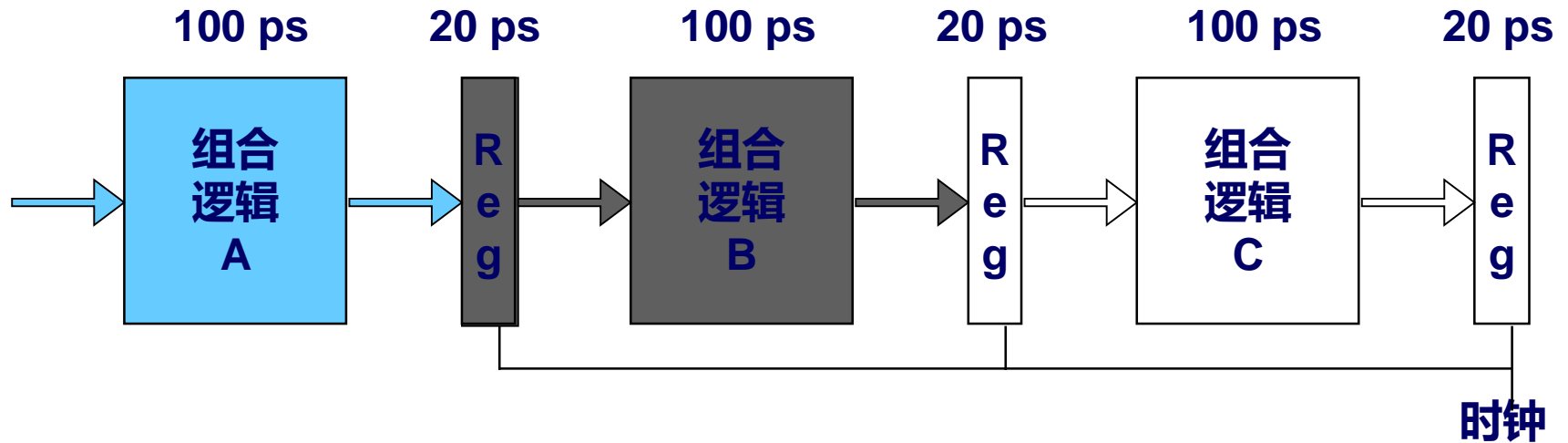
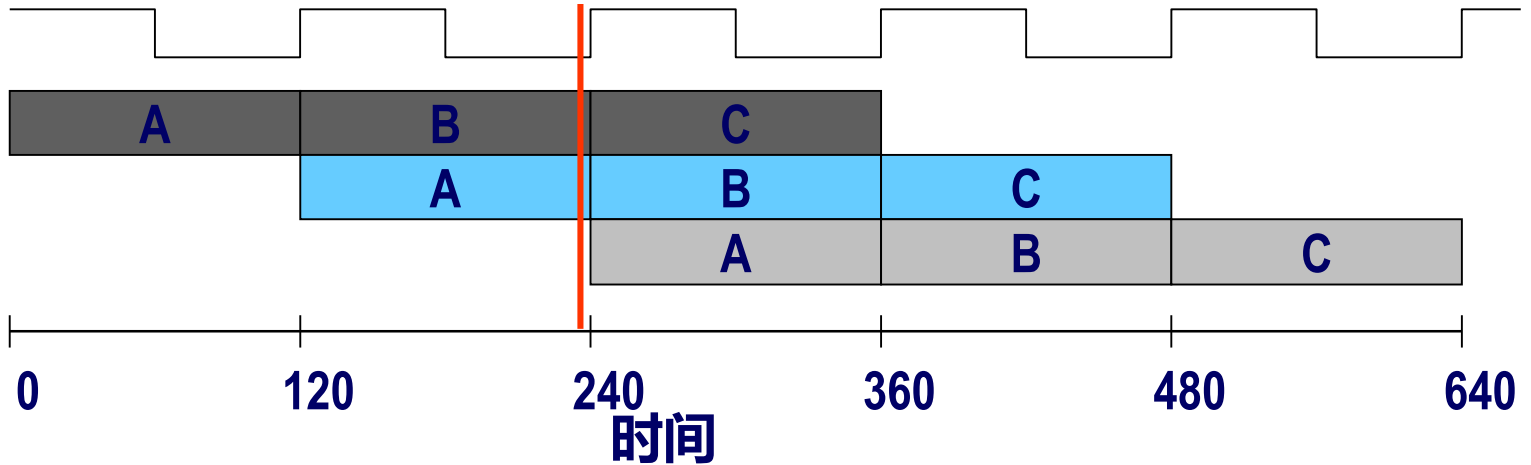
239

时钟

OP1

OP2

OP3



流水线操作

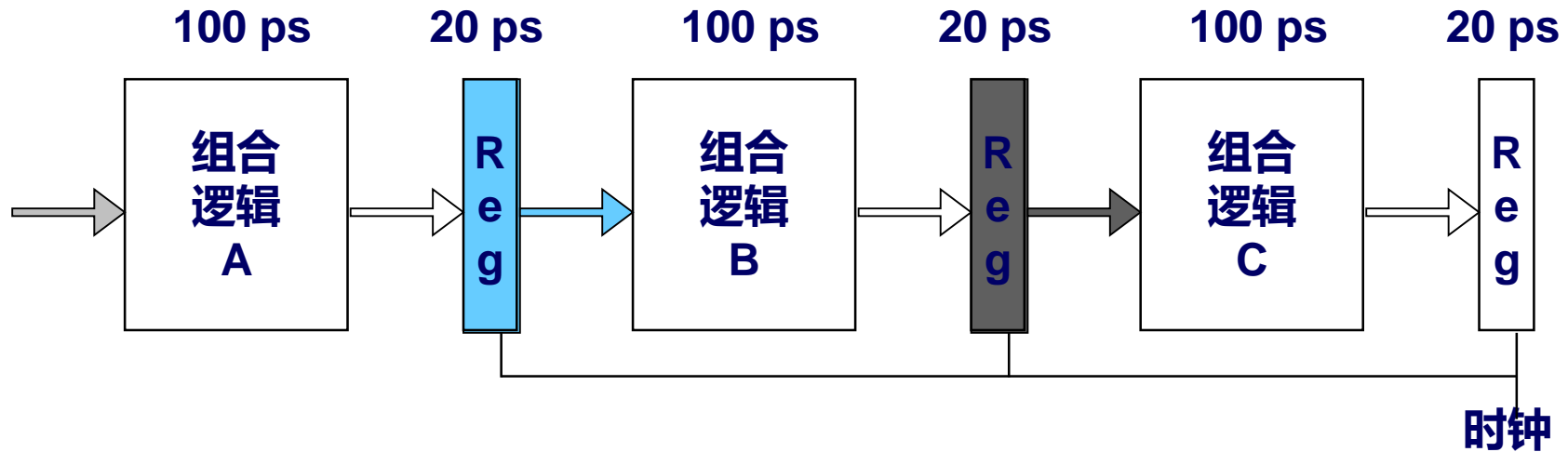
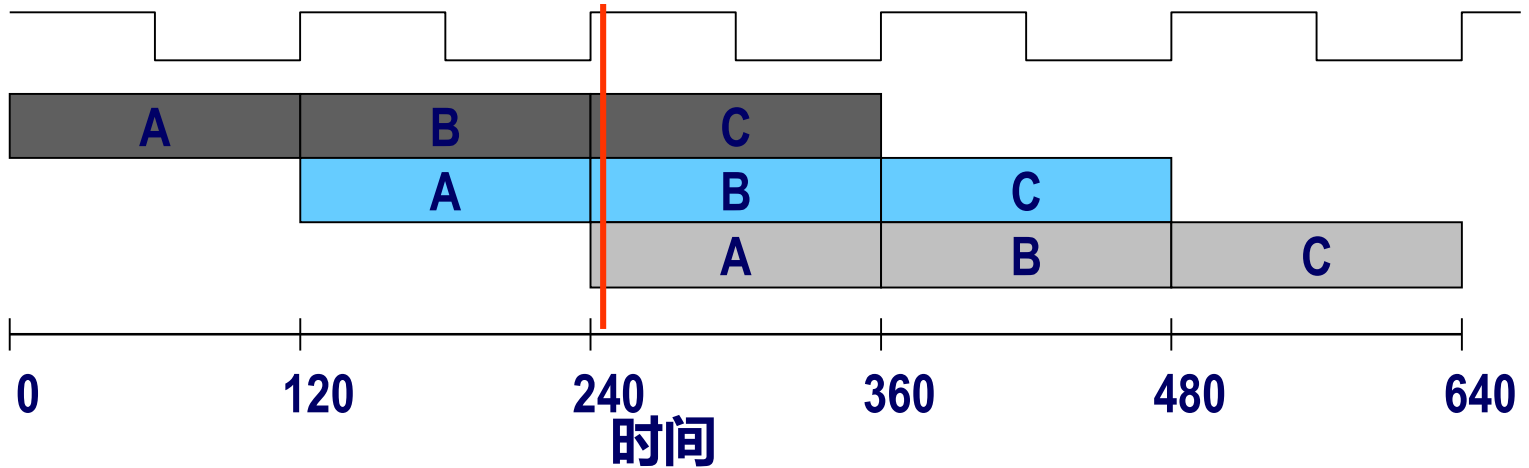
241

时钟

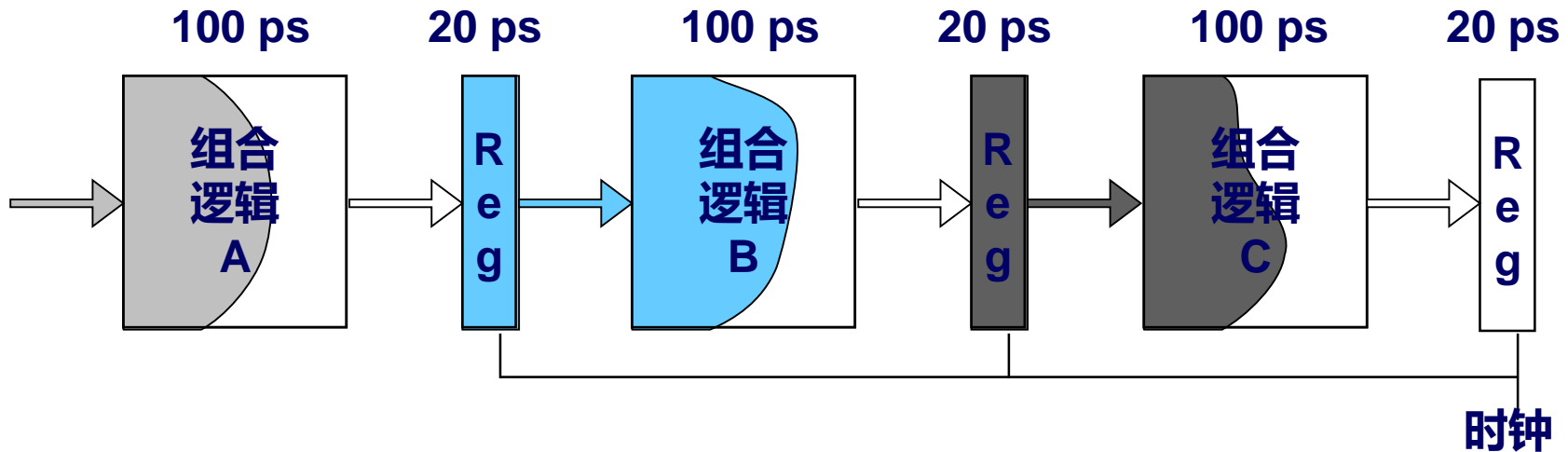
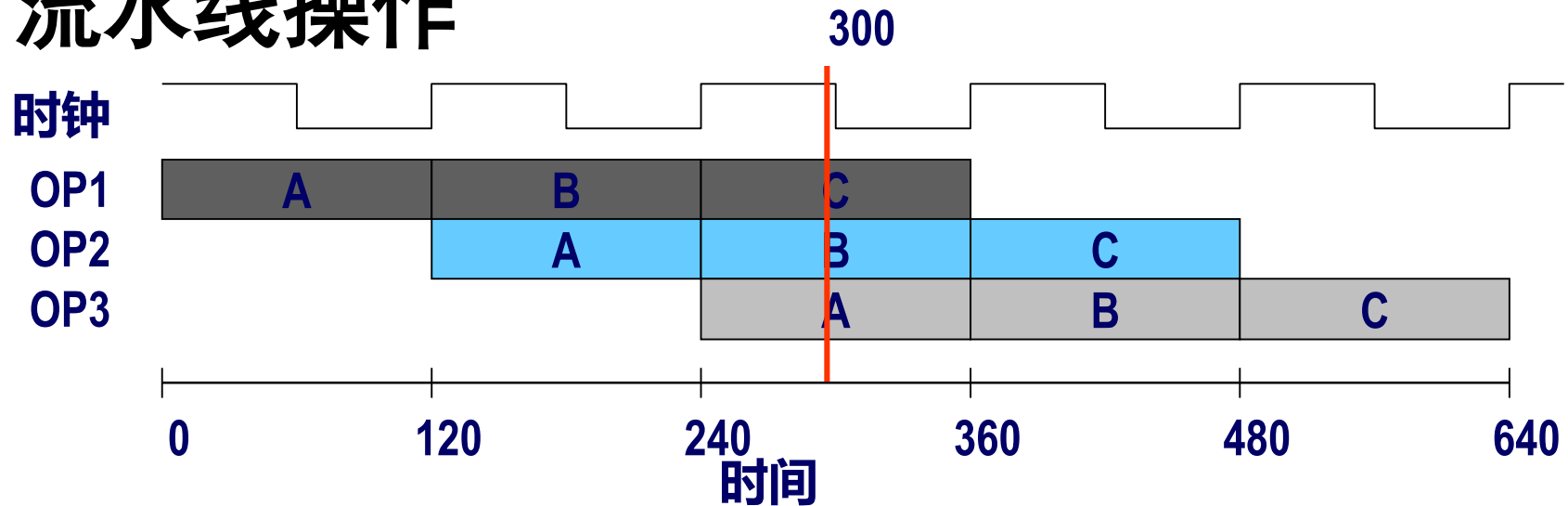
OP1

OP2

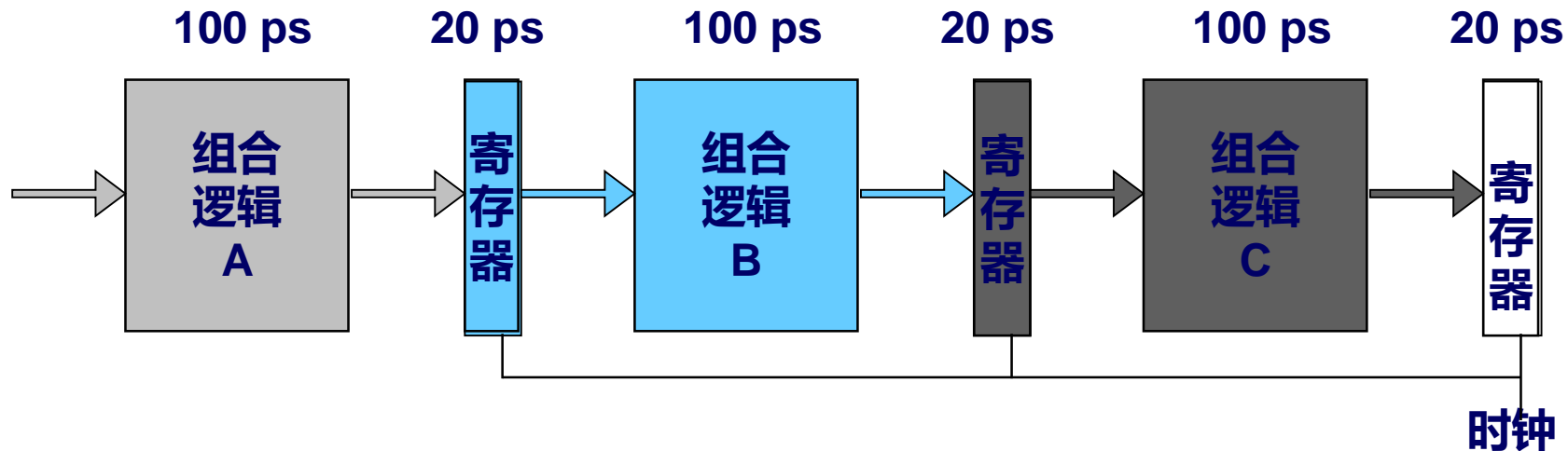
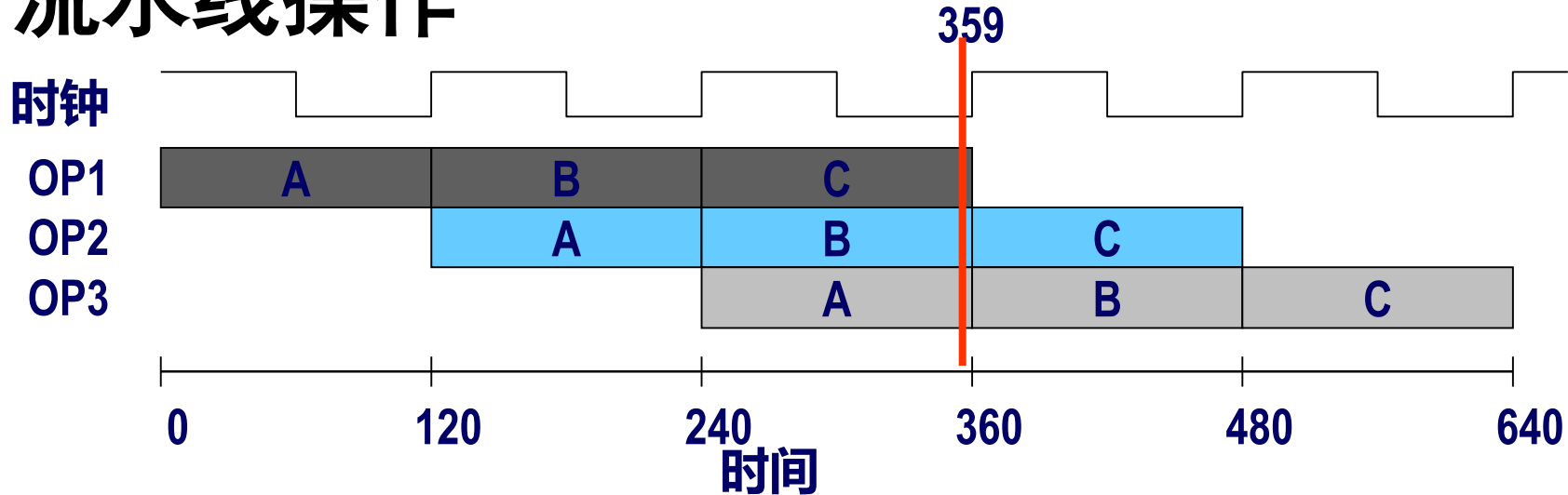
OP3



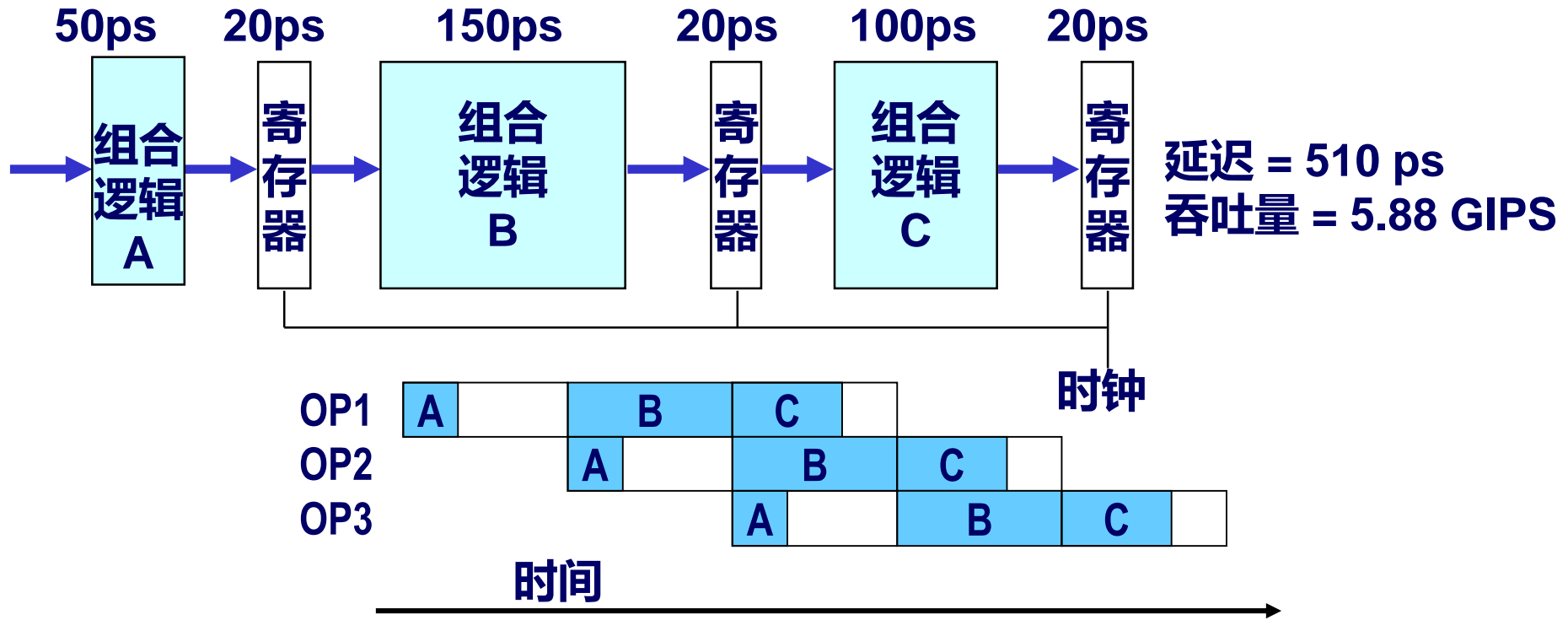
流水线操作



流水线操作

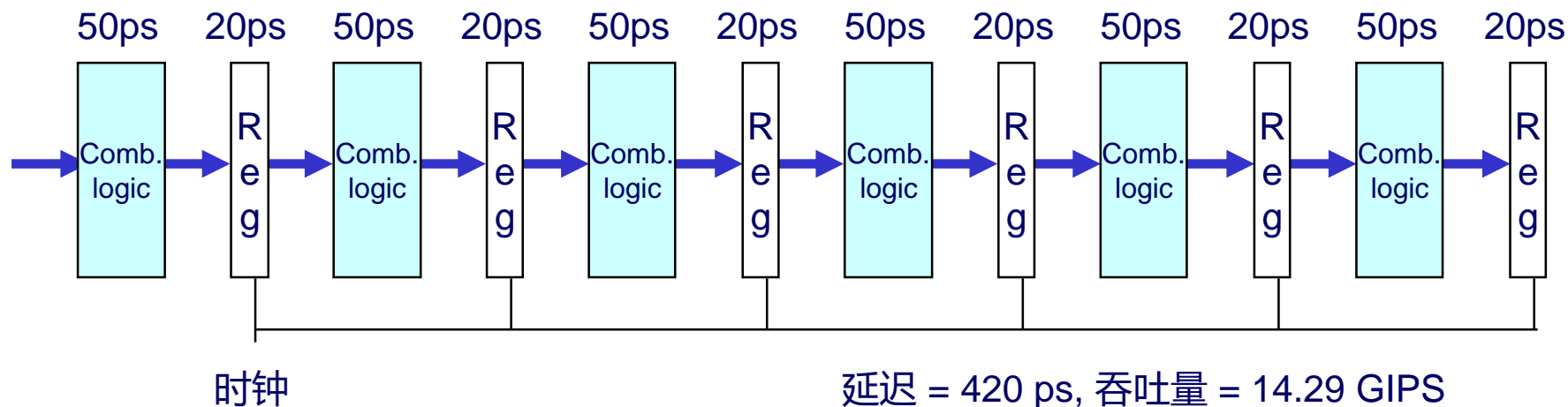


局限性: 不一致的划分/延迟



- 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

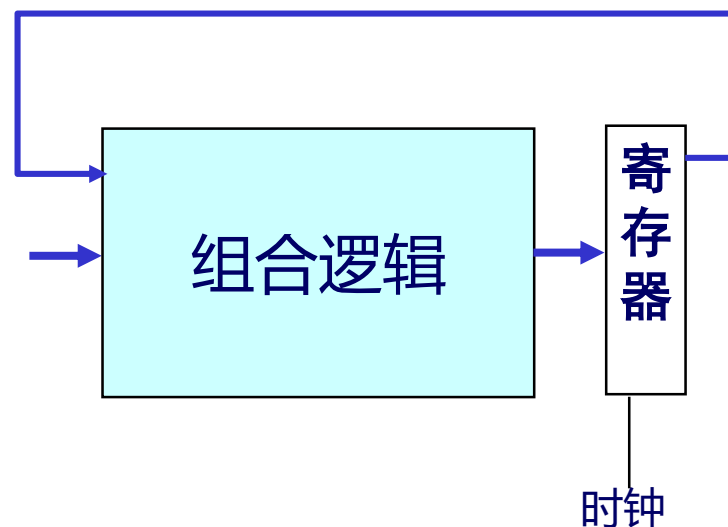
局限性: 寄存器天花板



- 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，电路设计者必须很小心地设计流水线寄存器，使其延迟尽可能的小。

数据相关

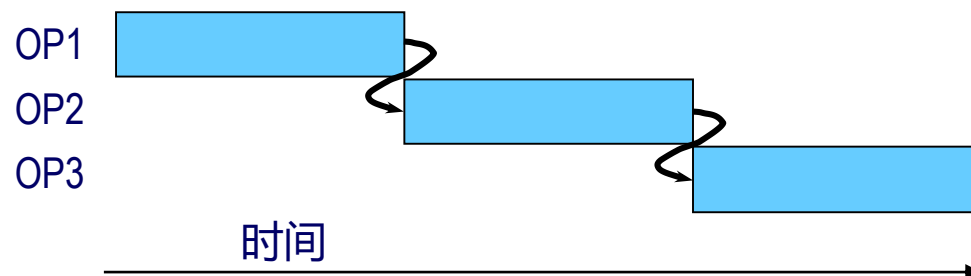
- 硬件：未流水线化，带反馈



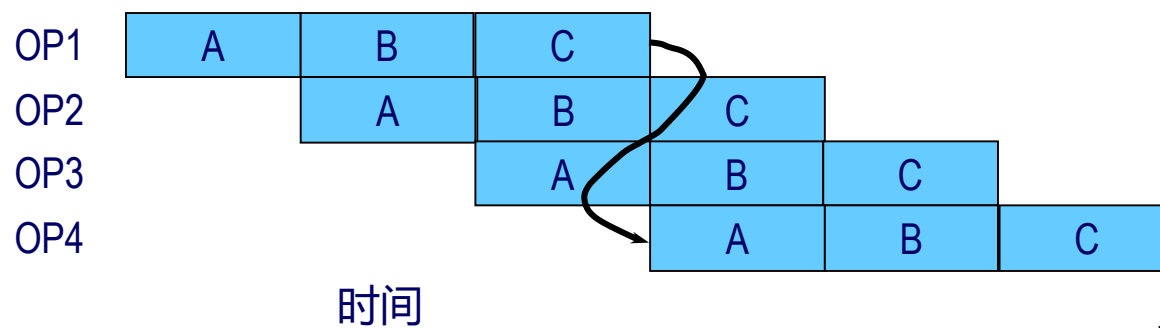
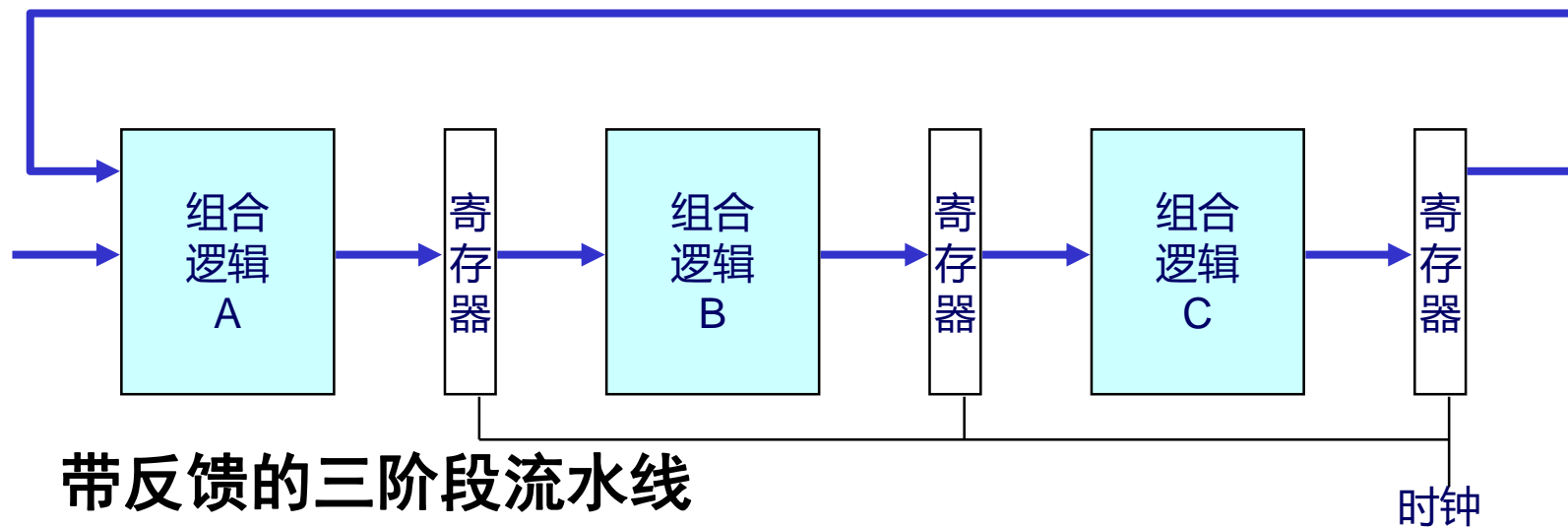
- 分析

- 每个操作依赖于前一个操作的结果

- 流水线图



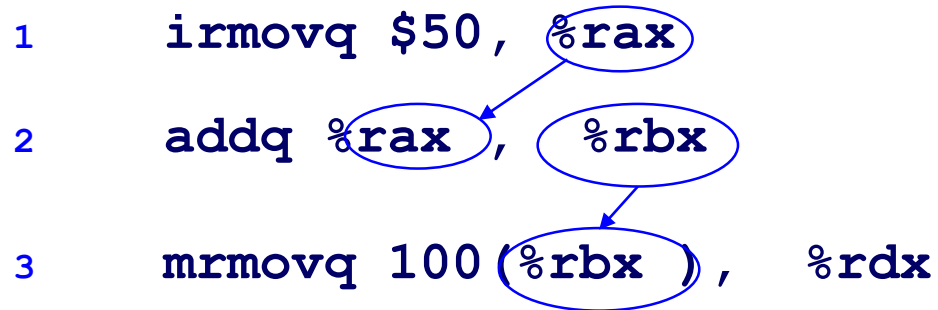
数据冒险



- 结果没有被及时的反馈给下一个操作
- 流水线改变了系统的行为

处理器中的数据相关

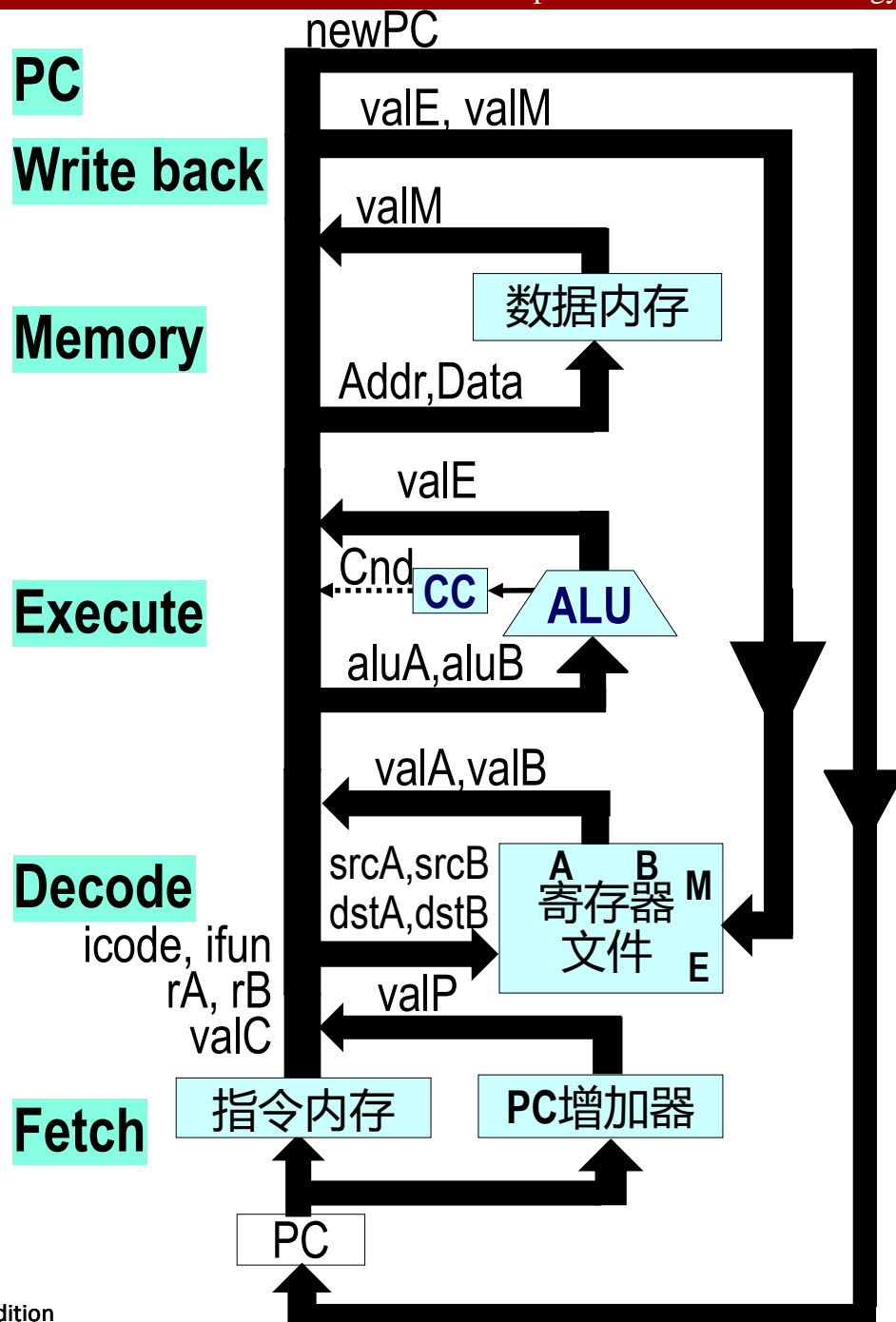
```
1    irmovq $50, %rax
2    addq %rax, %rbx
3    mrmovq 100(%rbx), %rdx
```



- 一条指令的结果作为另一条指令的操作数
 - 写后读(Read-after-write, RAW)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

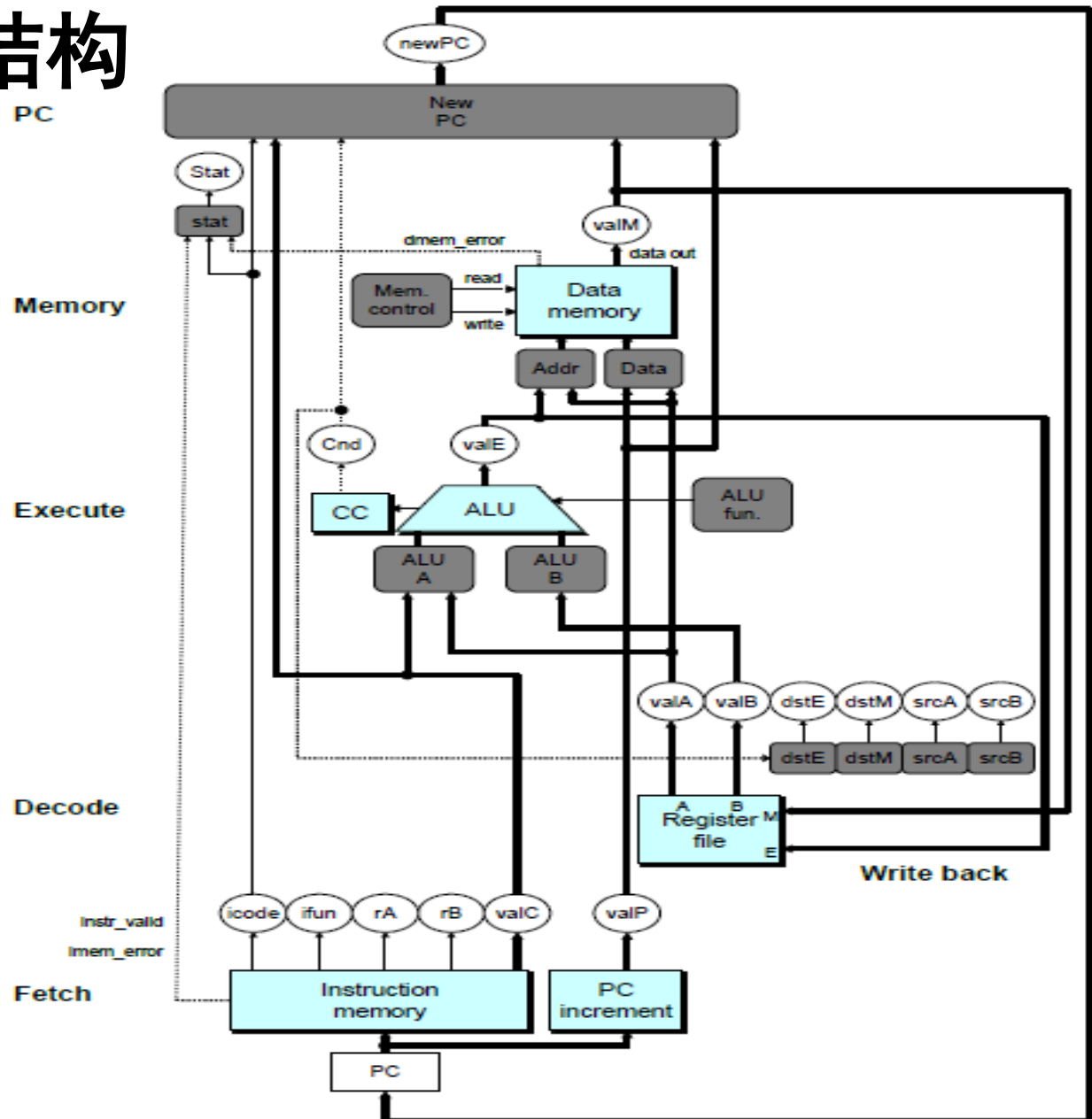
SEQ各阶段

- **取指 - Fetch**
 - 从指令存储器读取指令
- **译码 - Decode**
 - 读程序寄存器
- **执行 - Execute**
 - 计算数值或地址
- **访存 - Memory**
 - 读或写数据
- **写回 - Write back**
 - 写程序寄存器
- **PC更新- PC update**
 - 更新程序计数器



SEQ 的硬件结构

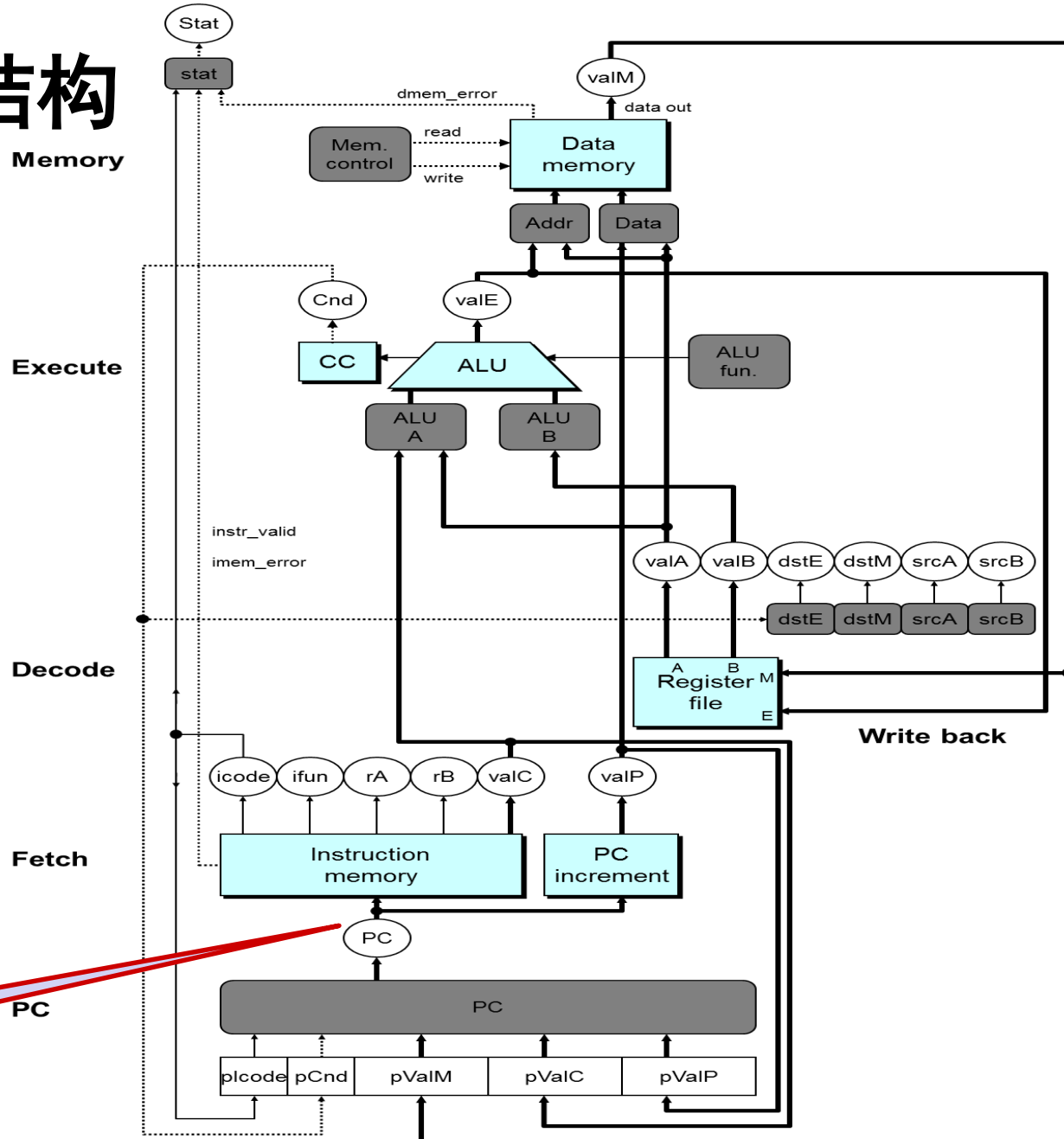
- 阶段顺序发生
- 一次只能处理一个操作



SEQ+ 的硬件结构

- 顺序实现
- 将PC更新阶段放在开始
- PC更新 阶段
 - 让PC指向当前指令
 - 根据前一条指令的计算结果更新
- 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

本条指令的地址



添加流水线寄存器

PC

Write back

Memory

Execute

Decode

Fetch

newPC

valE, valM

valM

数据内存

Addr, Data

valE

Cnd

CC

ALU

aluA, aluB

valA, valB

srcA, srcB

dstA, dstB

A B M
寄存器
文件 E

valP

icode, ifun

rA, rB

valC

指令内存

PC增加器

PC

Memory

Execute

Decode

Fetch

PC

W_icode, W_valM

W_valE, W_valM, W_dstE, W_dstM

W

M_icode,
M_Cnd,
M_valA

valM

数据内存

Addr, Data

M

Cnd

valE

aluA, aluB

aluA, aluB

aluA, aluB

E

valA, valB

d_srcA,
d_srcBA B M
寄存器
文件 E

D

icode, ifun
rA, rB, valC

valP

指令内存

PC增加器

F

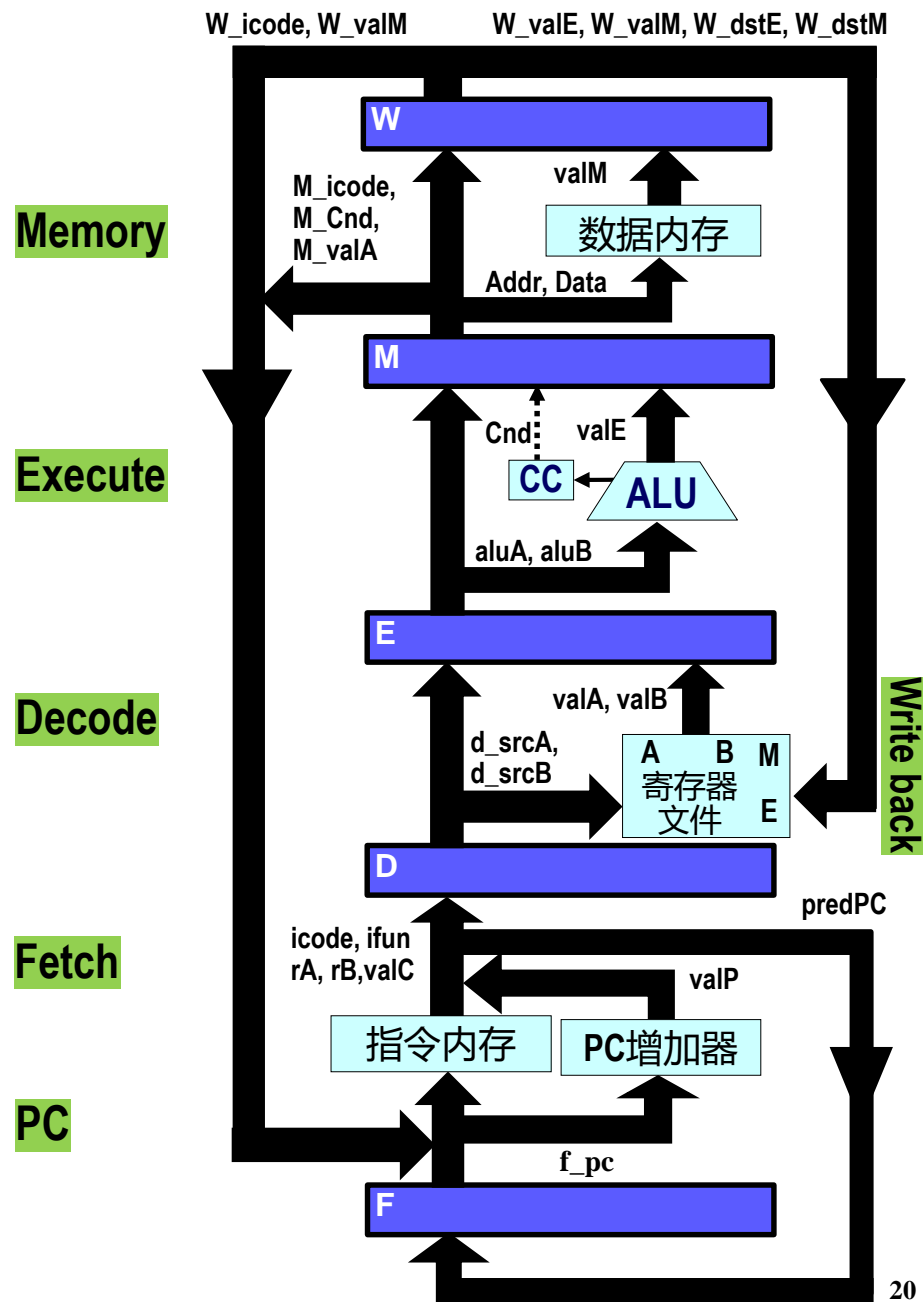
f_pc

Write back

predPC

流水线阶段

- **取指**
 - 选择当前PC
 - 读取指令
 - 计算PC增加后的值
- **译码**
 - 读取程序寄存器
- **执行**
 - 操作ALU
- **访存**
 - 读或写数据存储器
- **写回**
 - 更新寄存器文件

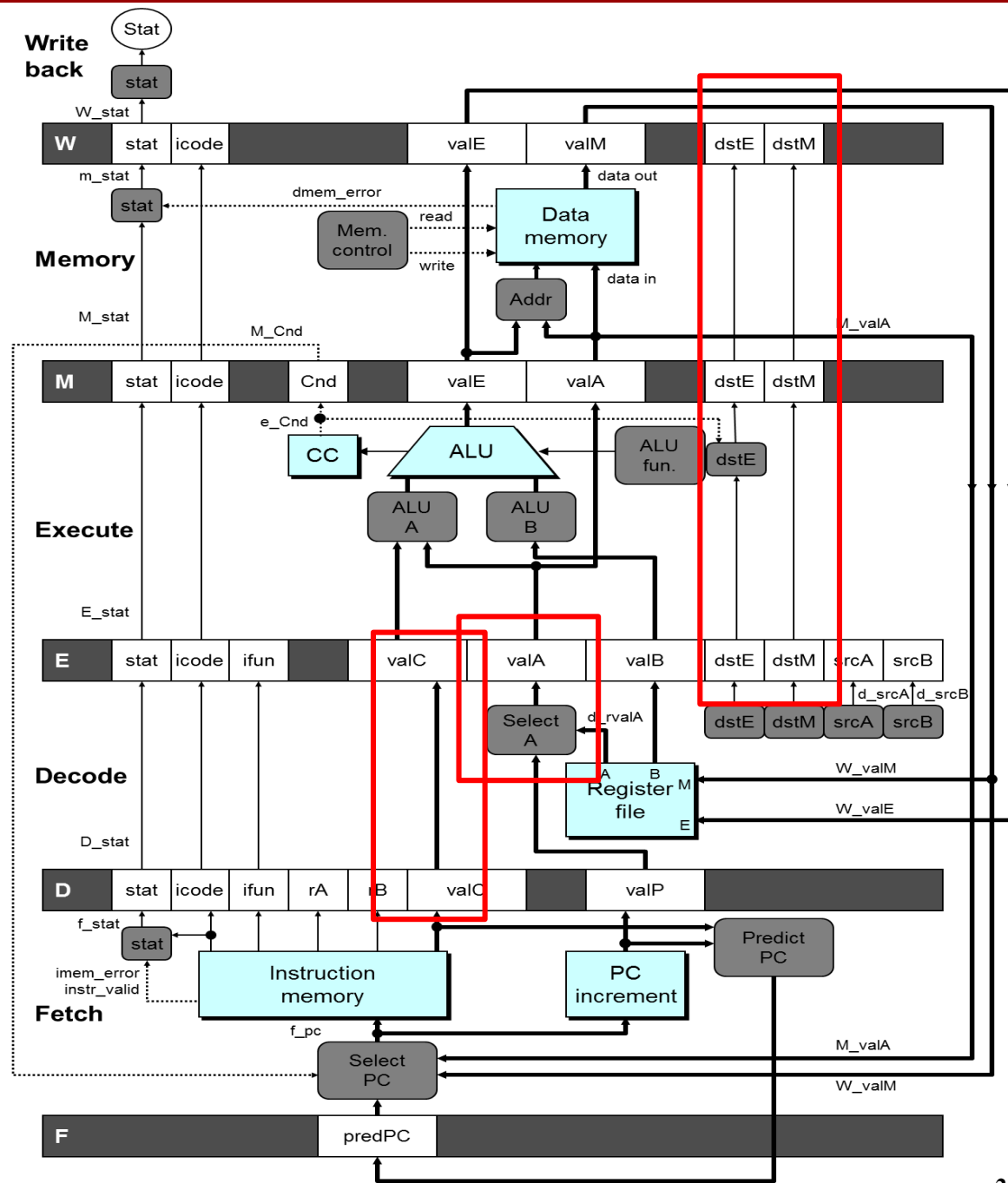


PIPE- 硬件结构

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
 - e. g., valC 在解码阶段产生，在执行阶段使用



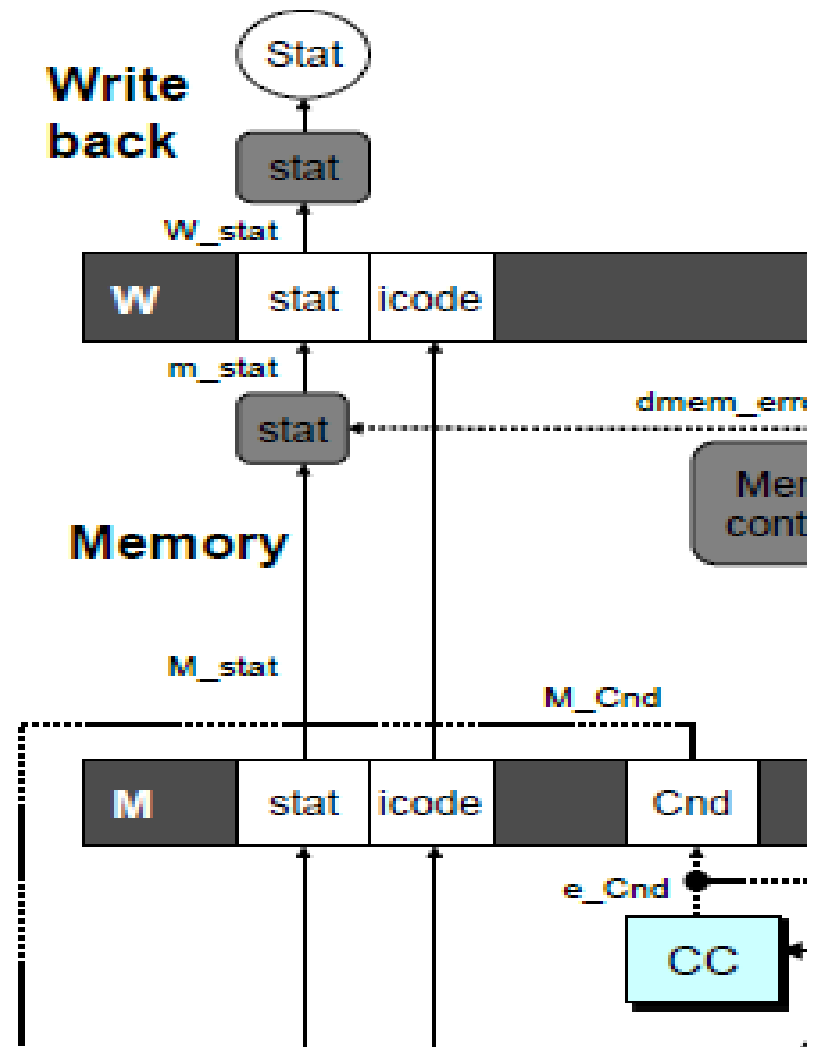
信号命名规则

■ S_Field

- 流水线S阶段的寄存器中的Field字段值

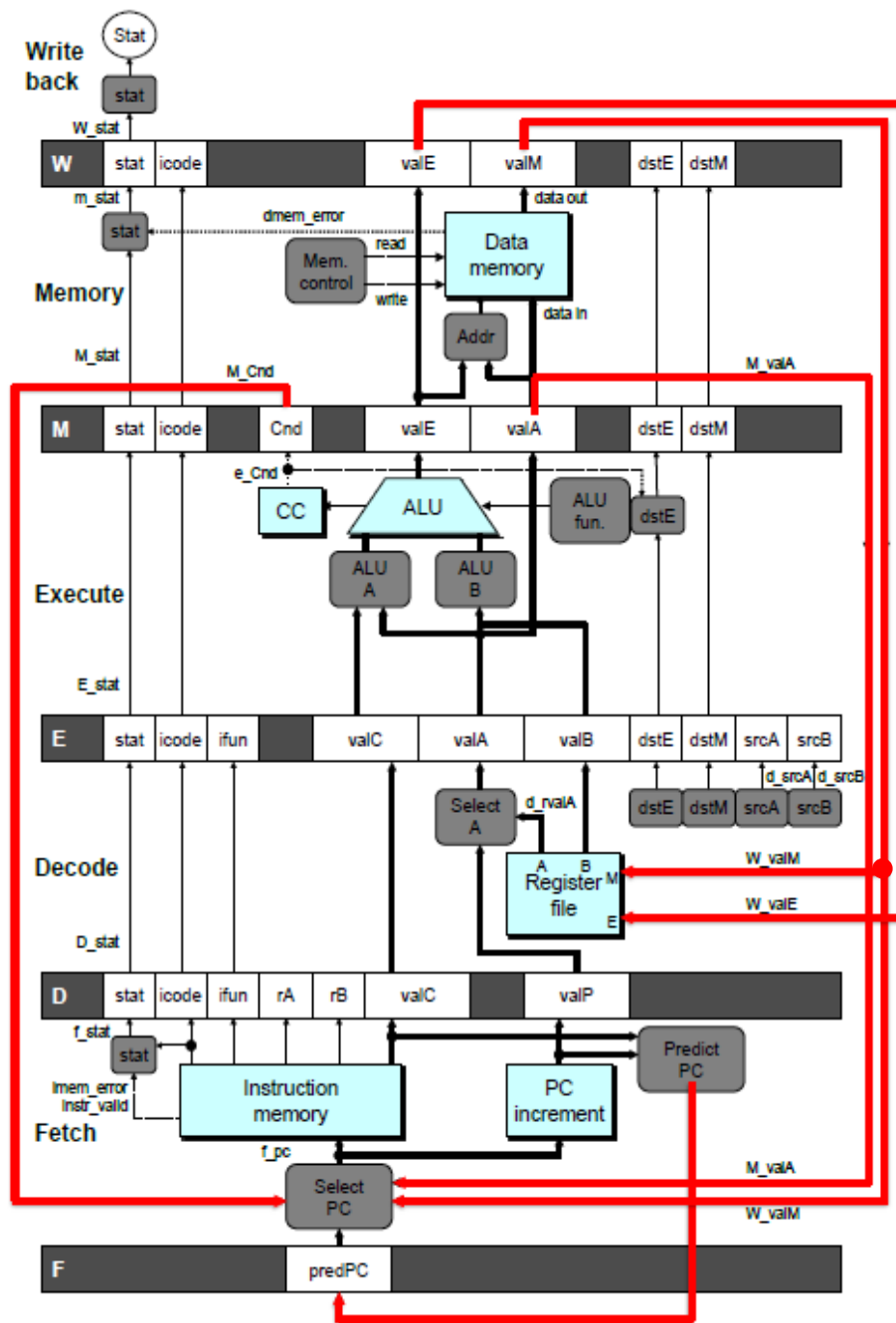
■ s_Field

- 流水线S阶段计算出的Field字段值



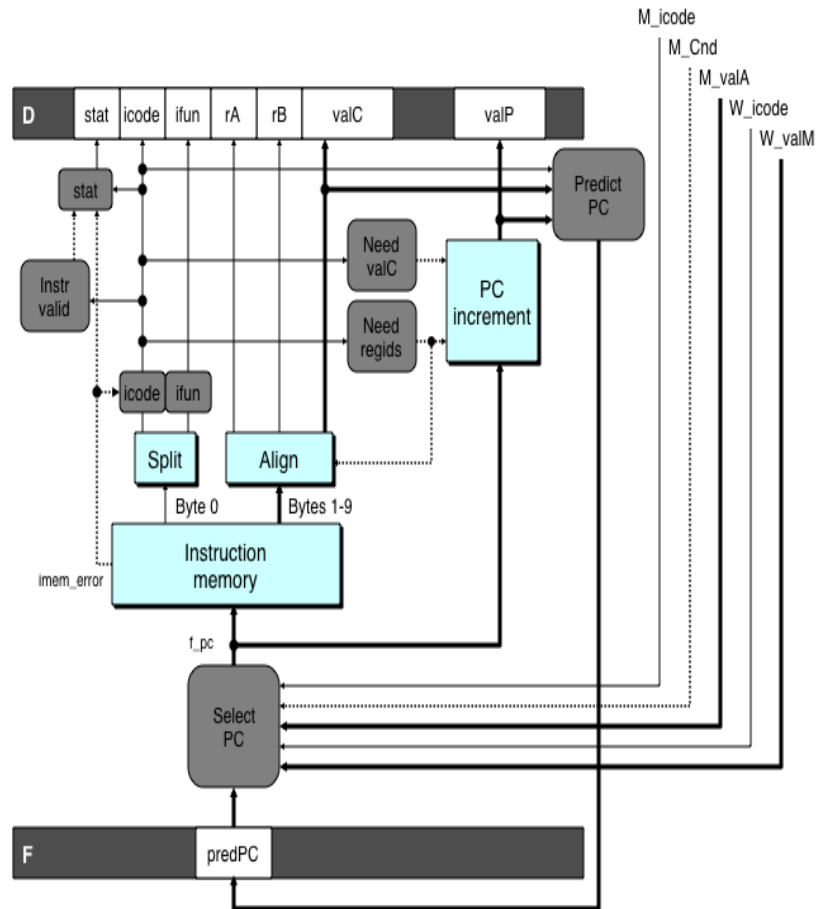
反馈的路径

- 预测下一个PC
 - 猜测下一个PC的值
- 分支信息
 - 是/否跳转
 - 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，总是选择分支)
- 如果分支被选中则预测正确
 - 研究表明成功率大约为60%

■ 返回指令：不进行预取(暂停处理新指令，直至ret通过写回阶段)

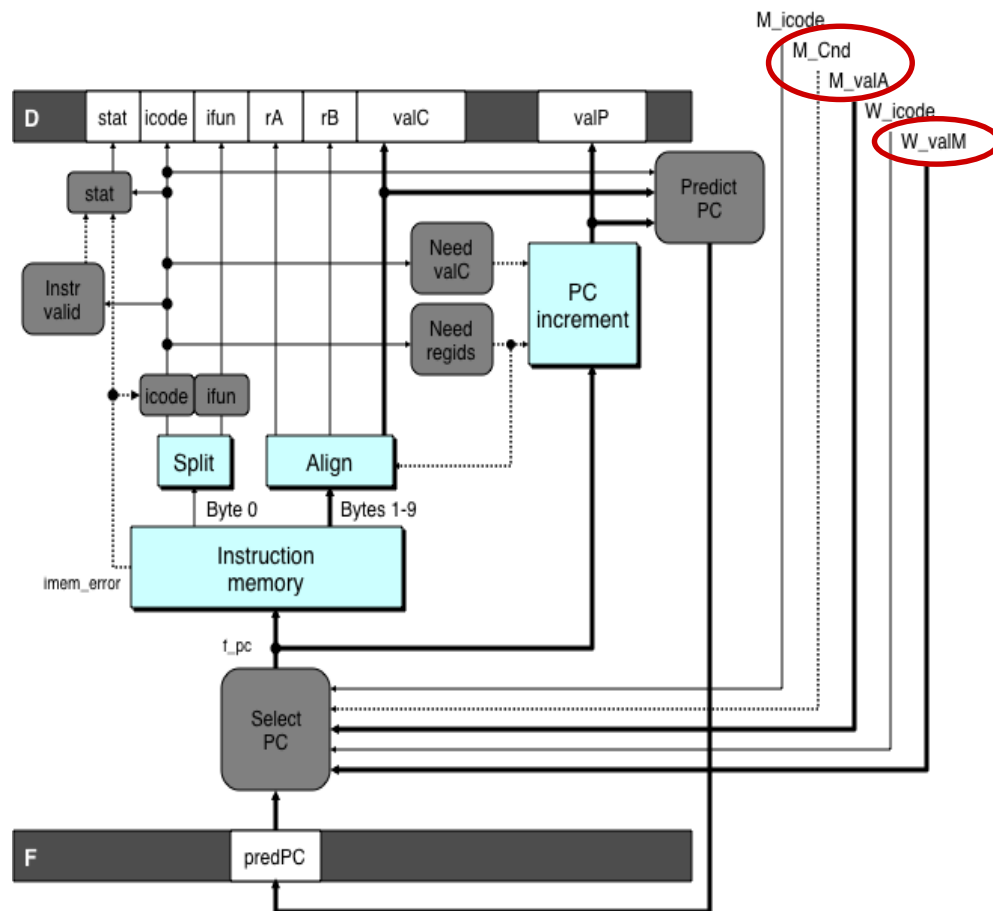
从预测错误中恢复

■ 跳转错误

- 指令进入访存阶段，就能看到分支条件标志 M_Cnd
- 可从valA(M_valA)中得到向后传（图中是上行）的正确PC值

■ 返回指令

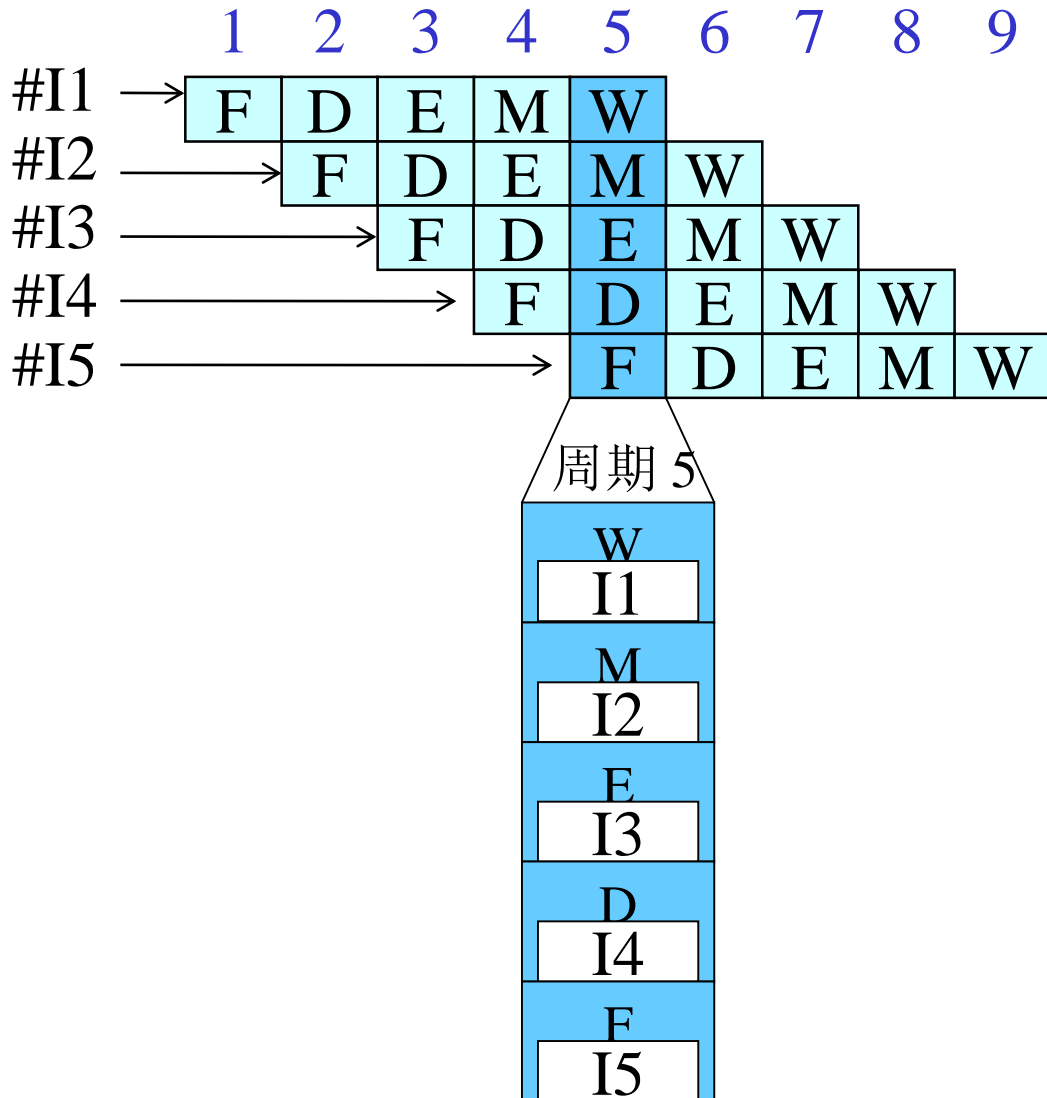
- 获取返回地址，当ret到达写回阶段，获取返回的PC值(W_valM)



流水线示例

File: demo-basic.js

```
irmovl $1,%eax
irmovl $2,%ecx
irmovl $3,%edx
irmovl $4,%ebx
halt
```



数据相关: 3 Nop's

demo-h3.js

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

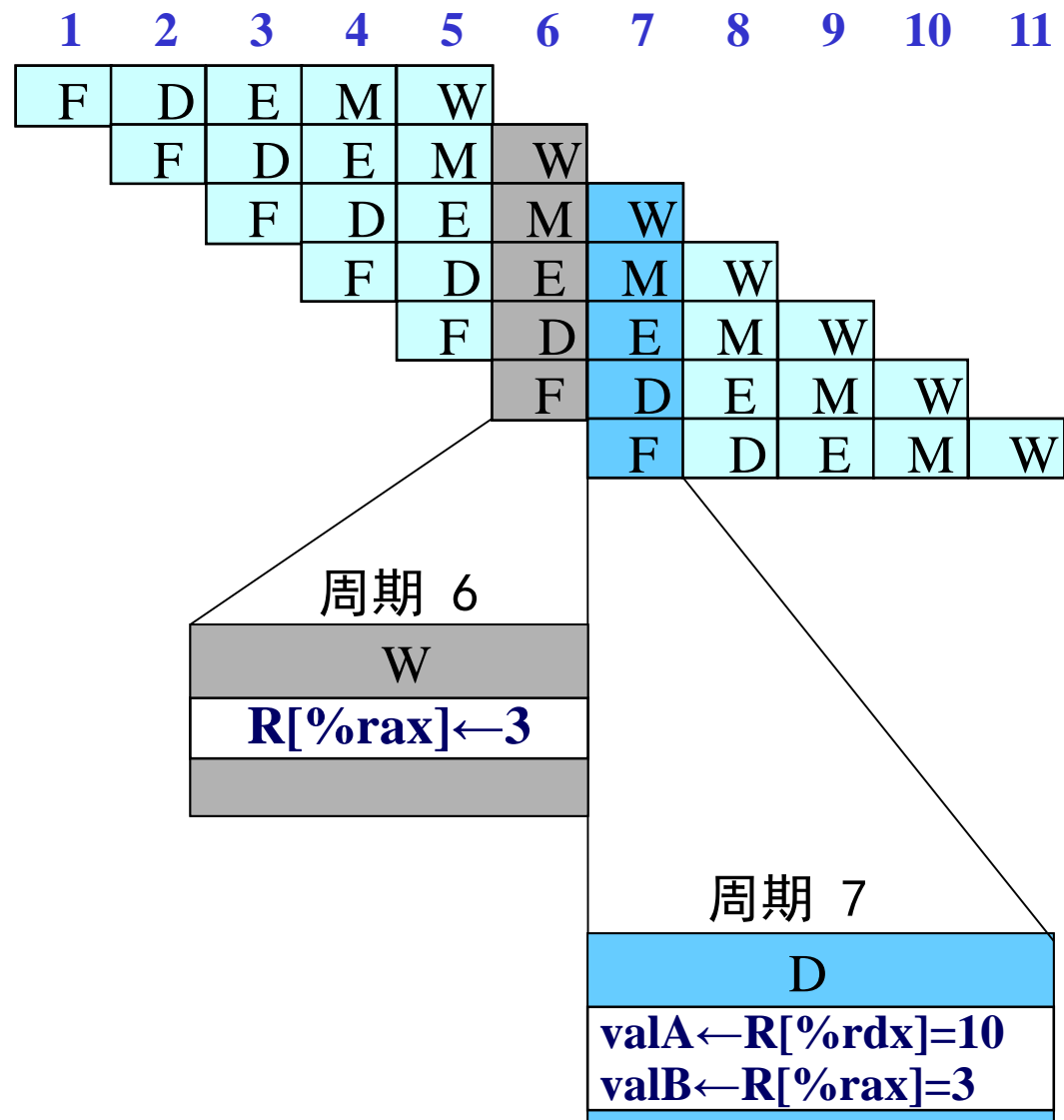
0x014: **nop**

0x015: **nop**

0x016: **nop**

0x017: addq %rdx, %rax

0x019: halt



数据相关: 2条Nop指令

demo-h2.js

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

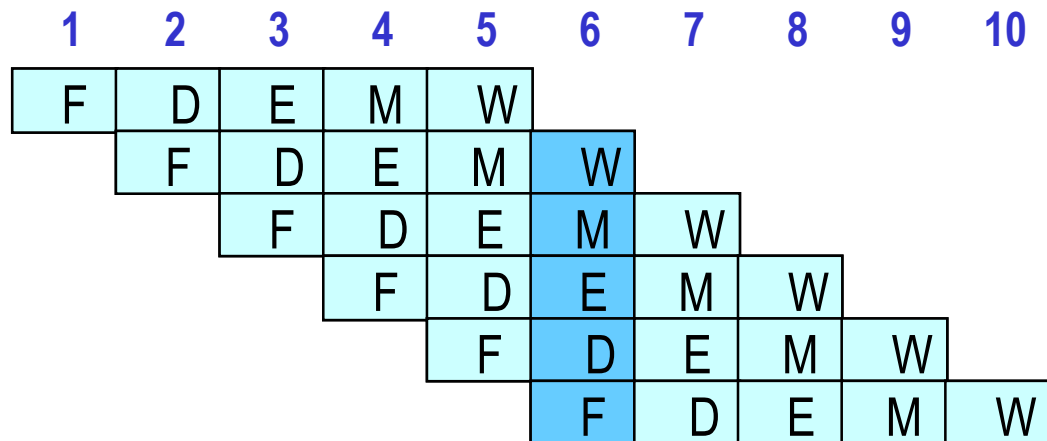
0x014: **nop**

0x015: **nop**

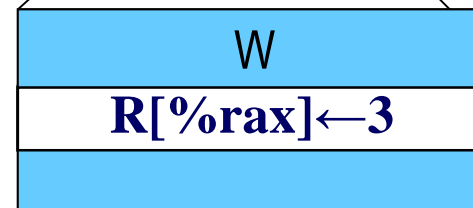
0x016: addq %rdx, %rax

0x018: halt

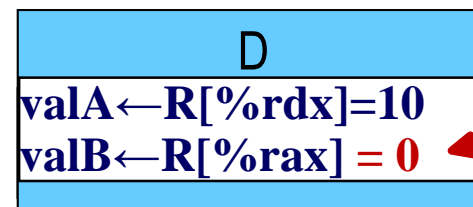
假设：程序开始时，各通用寄存器的值均为0



周期 6



⋮



Error

数据相关: 1条Nop指令

demo-h1.js

0x000: irmovq \$10,%rdx

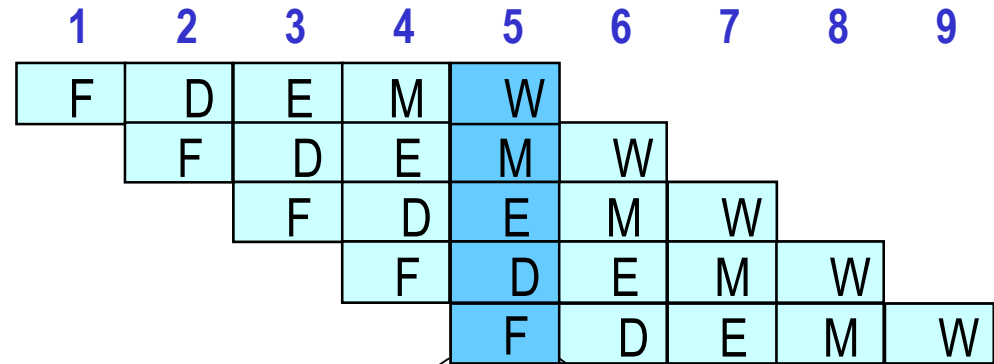
0x00a: irmovq \$3,%rax

0x014: **nop**

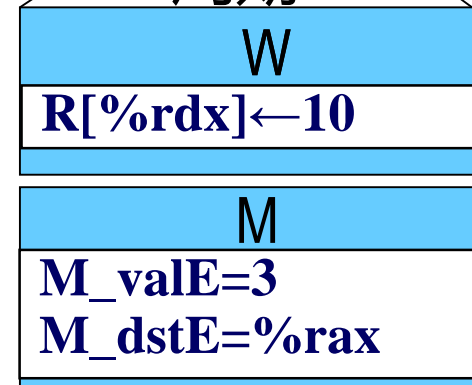
0x015: addq %rdx, %rax

0x017: halt

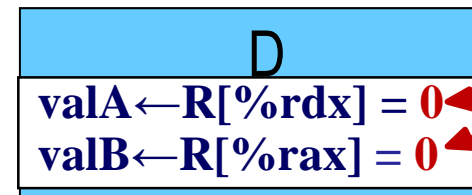
假设：程序开始时，各通用寄存器的值均为0



周期 5



⋮



Error

数据相关:无Nop指令

demo-h0.js

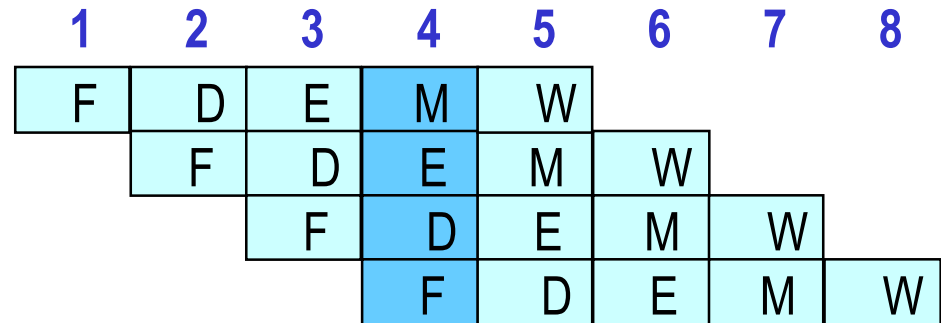
0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

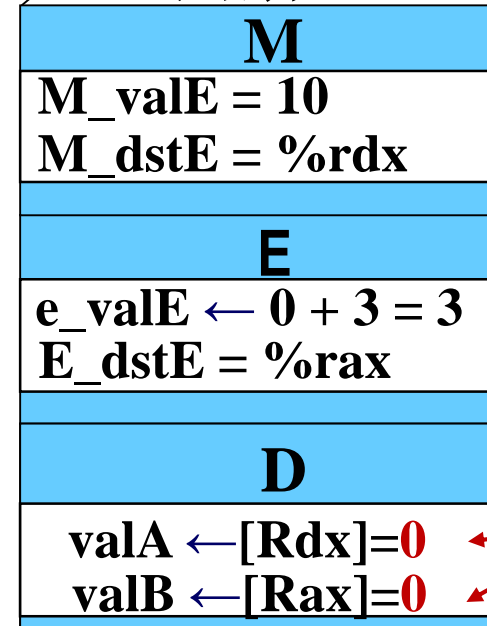
0x014: addq %rdx, %rax

0x016: halt

假设：程序开始时，各通用寄存器的值均为0



周期 4



Error

分支预测错误示例

demo-j.js

```
0x000:  xorq %rax,%rax
0x002:  jne t          # Not taken
0x00b:  irmovq $1, %rax # Fall through
0x015:  nop
0x016:  nop
0x017:  nop
0x018:  halt
0x019:  t: irmovq $3, %rdx # Target (Should not execute)
0x023:  irmovq $4, %rcx # Should not execute
0x02d:  irmovq $5, %rdx # Should not execute
```

- 应该只执行前8条指令

分支预测错误追踪

demo-j.js

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

- 在分支目标处，错误地执行了两条指令

0x000: xorq %rax,%rax

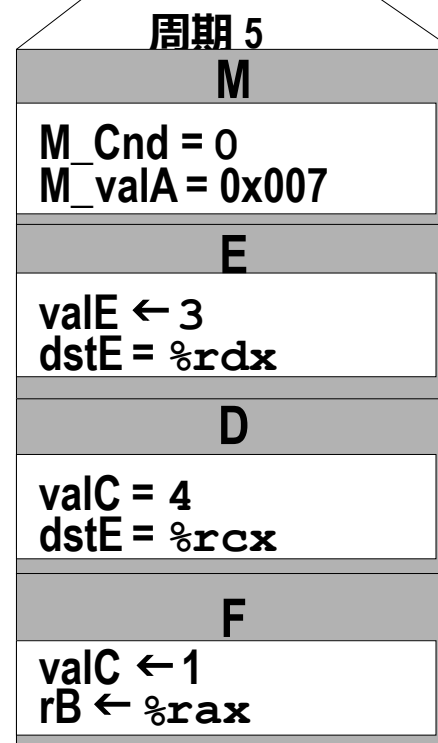
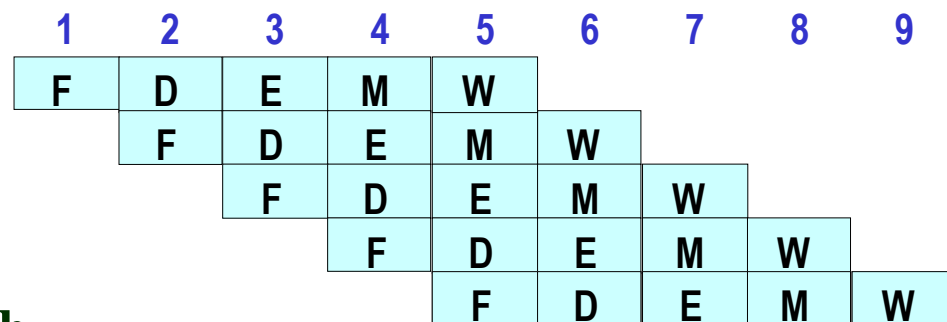
0x002: jne t # Not taken

0x00b: irmovq \$1, %rax

0x015: nop

0x016: nop

...



返回示例

demo-ret.ys

- 需要大量的nop指令来避免数据冒险

```

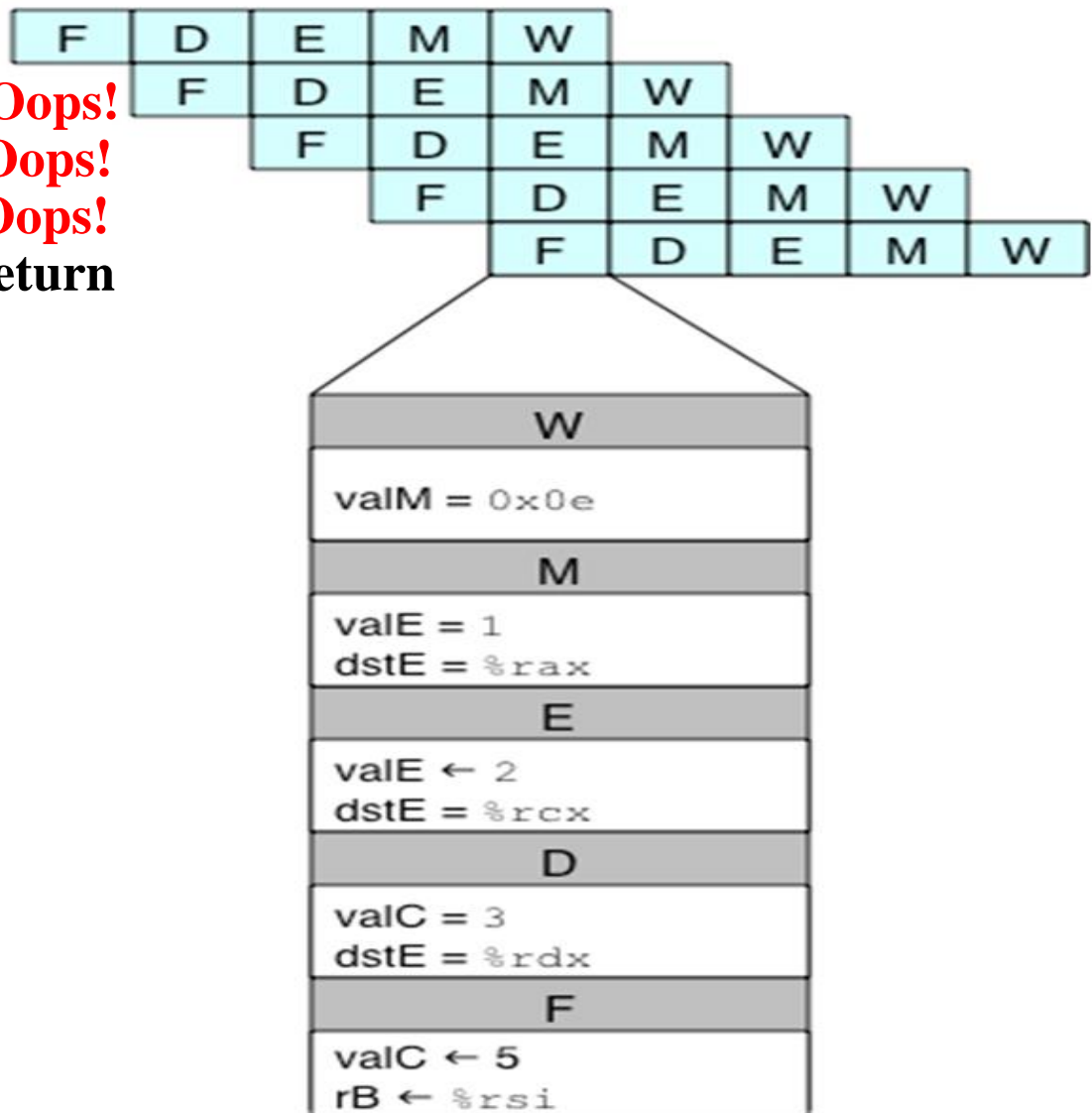
0x000:  irmovq Stack,%rsp # Intialize stack pointer
0x00a:  nop                # Avoid hazard on %rsp
0x00b:  nop
0x00c:  nop
0x00d:  call p             # Procedure call
0x016:  irmovq $5,%rsi     # Return point
0x020:  halt
0x020: .pos 0x20
0x020: p: nop          # procedure
0x021:  nop
0x022:  nop
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

```

错误的返回示例

0x023: ret
0x024: **irmovq \$1,%rax # Oops!**
0x02e: **irmovq \$2,%rcx # Oops!**
0x038: **irmovq \$3,%rdx # Oops!**
0x016: **irmovq \$5,%rsi # Return**

- 在ret之后，错误地执行了3条指令



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，稍后有一条指令读该寄存器
- 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：下次再讲