CS:APP Chapter 4
Computer Architecture
Pipelined Implementation
Part I
流水线实现
第一部分

任课教师：
宿红毅　　张艳　　　黎有琦　　　　李秀星

原作者：

Randal E. **Bryant and** David R. O’Hallaron

概述 Overview

流水线的一般原理 General Principles of Pipelining

- 目标 Goal
- 难点 Difficulties

创建一个具有流水线的Y86-64处理器 Creating a Pipelined Y86-64 Processor

- 重新安排顺序处理器SEQ Rearranging SEQ
- 插入流水线寄存器 Inserting pipeline registers
- 数据和控制冒险问题 Problems with data and control hazards

真实世界的流水线：洗车
Real-World Pipelines: Car Washes

顺序 Sequential

(AP PHOTO)

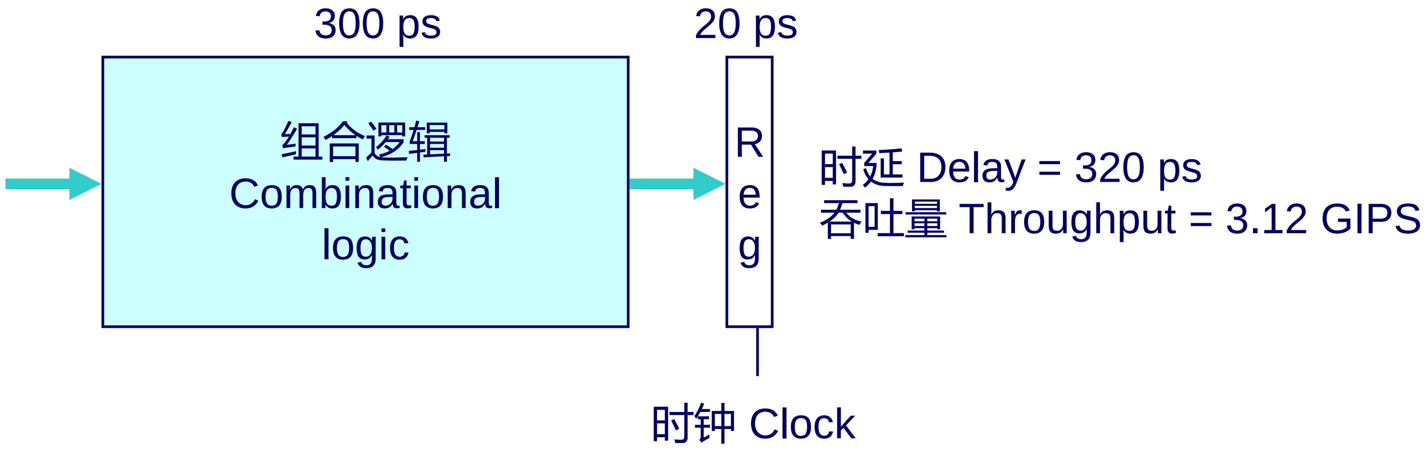
并行 Parallel

流水线 Pipelined

思想 Idea

- 将洗车过程分成若干独立的阶段 Divide process into independent stages
- 顺序移动目标通过各个阶段 Move objects through stages in sequence
- 在任何给定时间，在对多个目标进行处理 At any given times, multiple objects being processed

计算示例 Computational Example

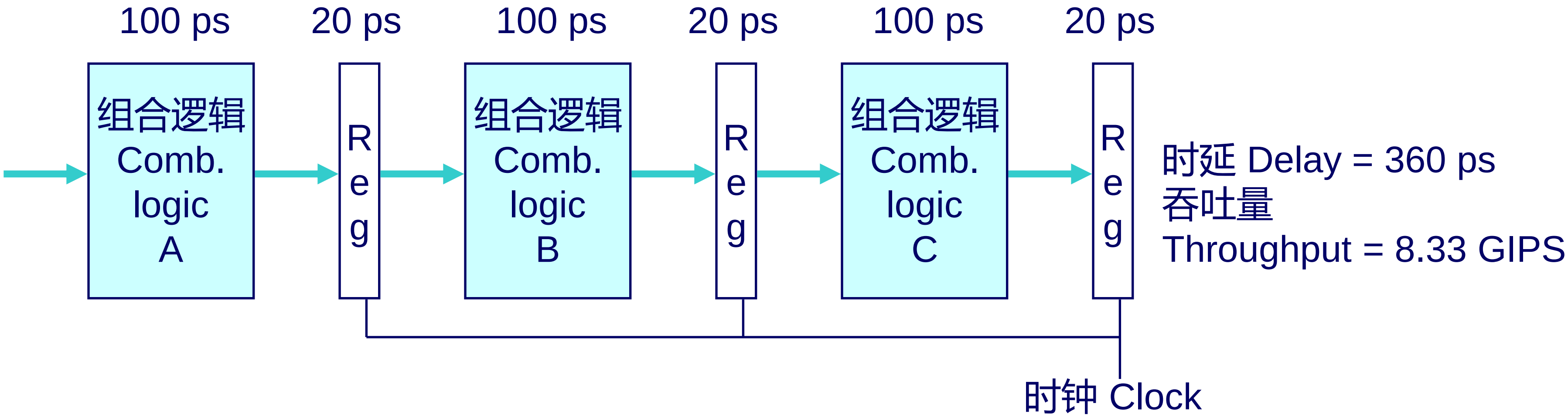


时延 Delay = 320 ps
吞吐量 Throughput = 3.12 GIPS

系统 System

- **计算需要总计300ps Computation requires total of 300 picoseconds**
- **另外20ps保存结果在寄存器中 Additional 20 picoseconds to save result in register**
- **时钟周期必须至少320ps Must have clock cycle of at least 320 ps**

3级流水线版本
3-Way Pipelined Version

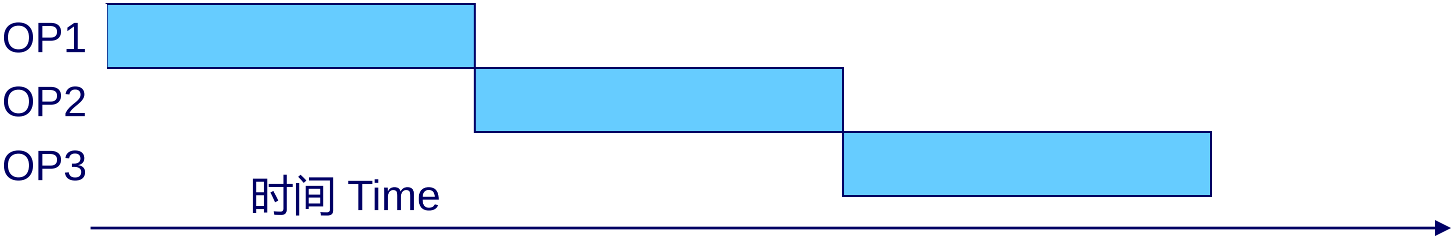


系统 System

- **将组合逻辑分成3块，每块需要100ps Divide combinational logic into 3 blocks of 100 ps each**
- **只要上一个操作通过阶段A，就可以立即开始新的操作 Can begin new operation as soon as previous one passes through stage A.**
  - **每隔120ps开始一个新操作 Begin new operation every 120 ps**
- **总体时延增加 Overall latency increases**
  - **从开始到结束需360ps 360 ps from start to finish**

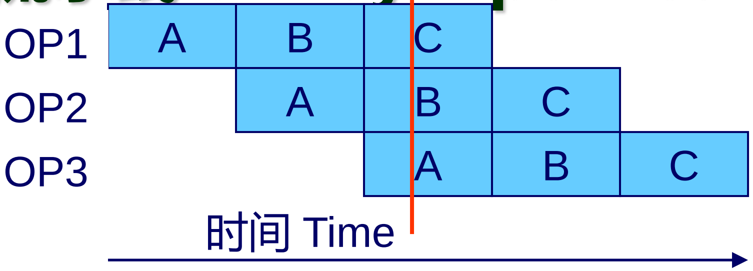
流水线图 Pipeline Diagrams

非流水线 Unpipelined



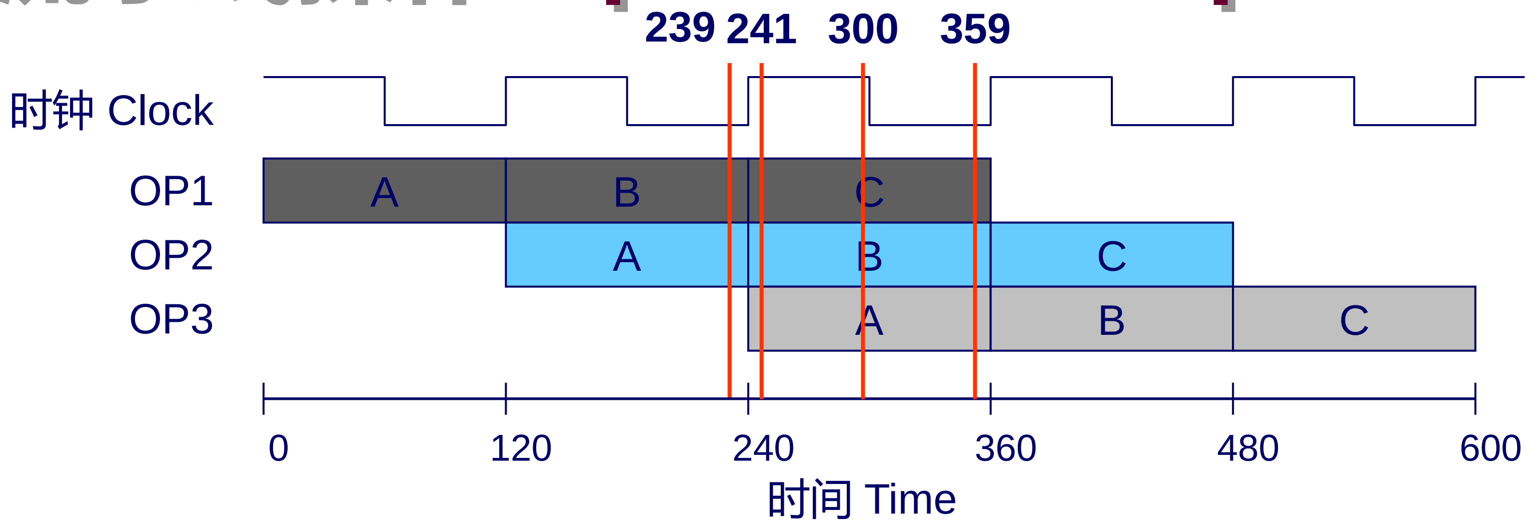
- 在上一个操作完成前不能开始新的操作 Cannot start new operation until previous one completes

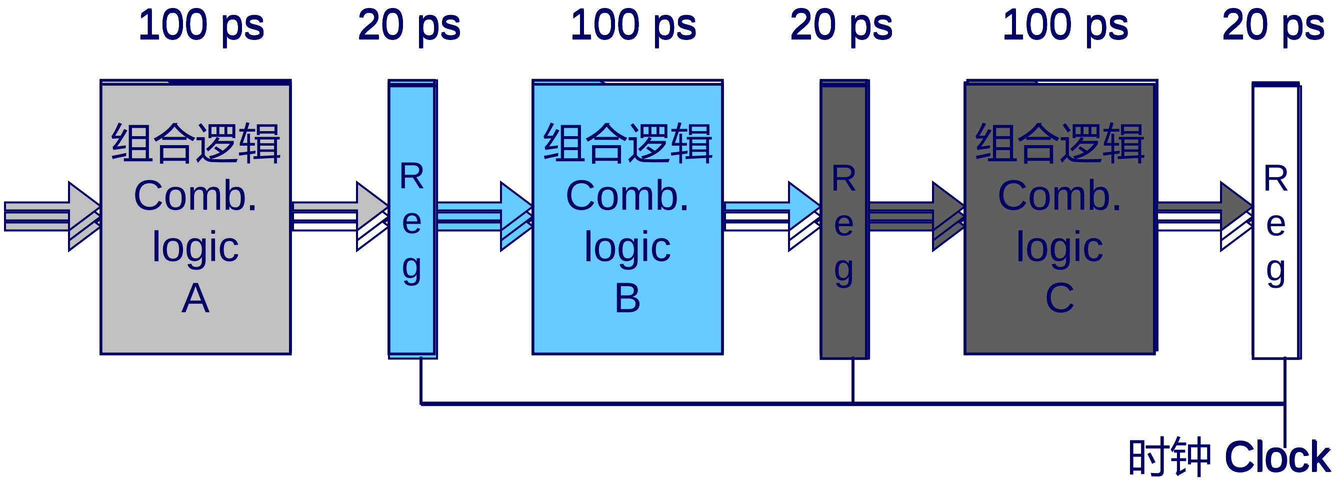
3级流水线 3-Way Pipelined



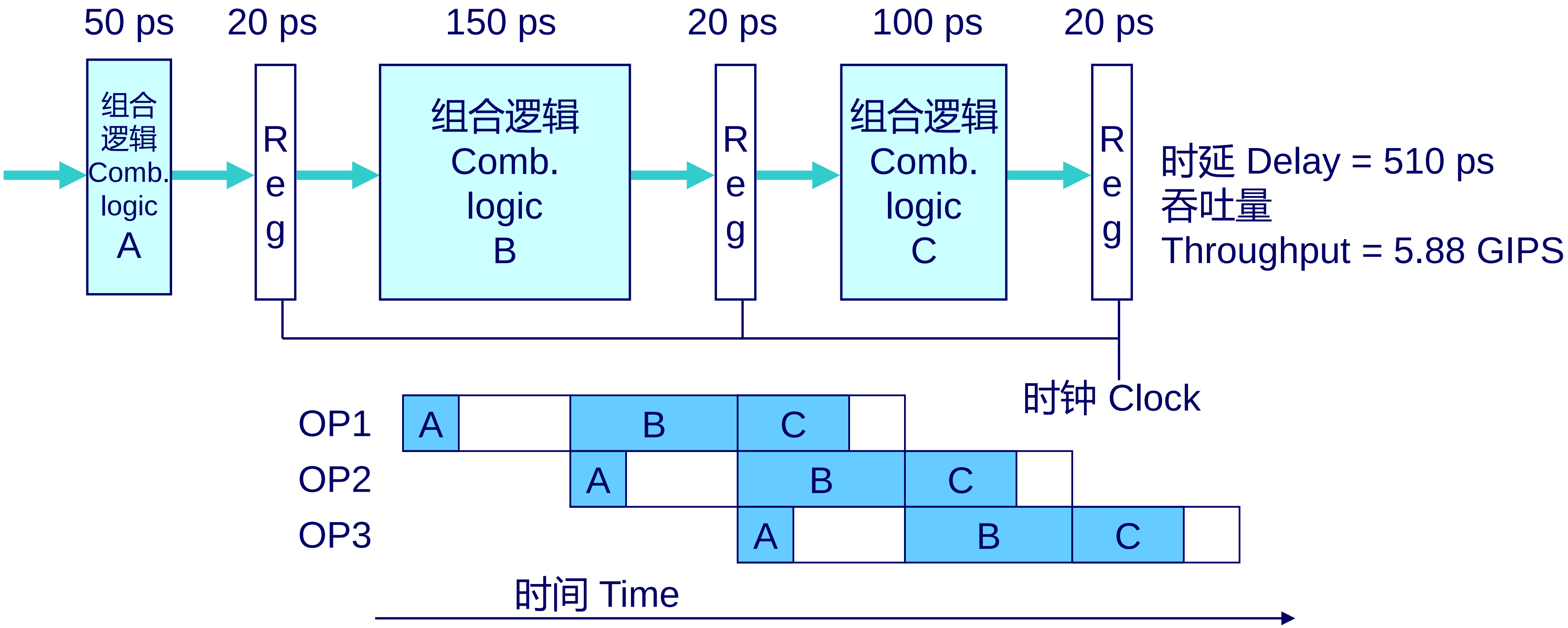
- 最多3个操作在同时处理 Up to 3 operations in process simultaneously

流水线操作 Operating a Pipeline



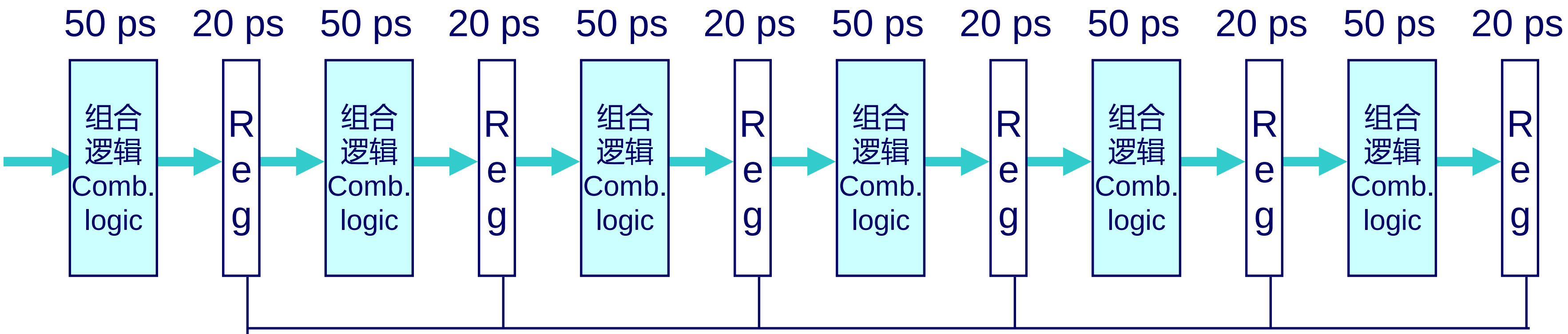


限制：非统一时延
Limitations: Nonuniform Delays



- 吞吐量受限于最慢的阶段 Throughput limited by slowest stage
- 其它阶段大部分时间都处于空闲状态 Other stages sit idle for much of the time
- 挑战在于把系统分成平衡的阶段 Challenging to partition system into balanced stages

限制：寄存器开销 Limitations: Register Overhead

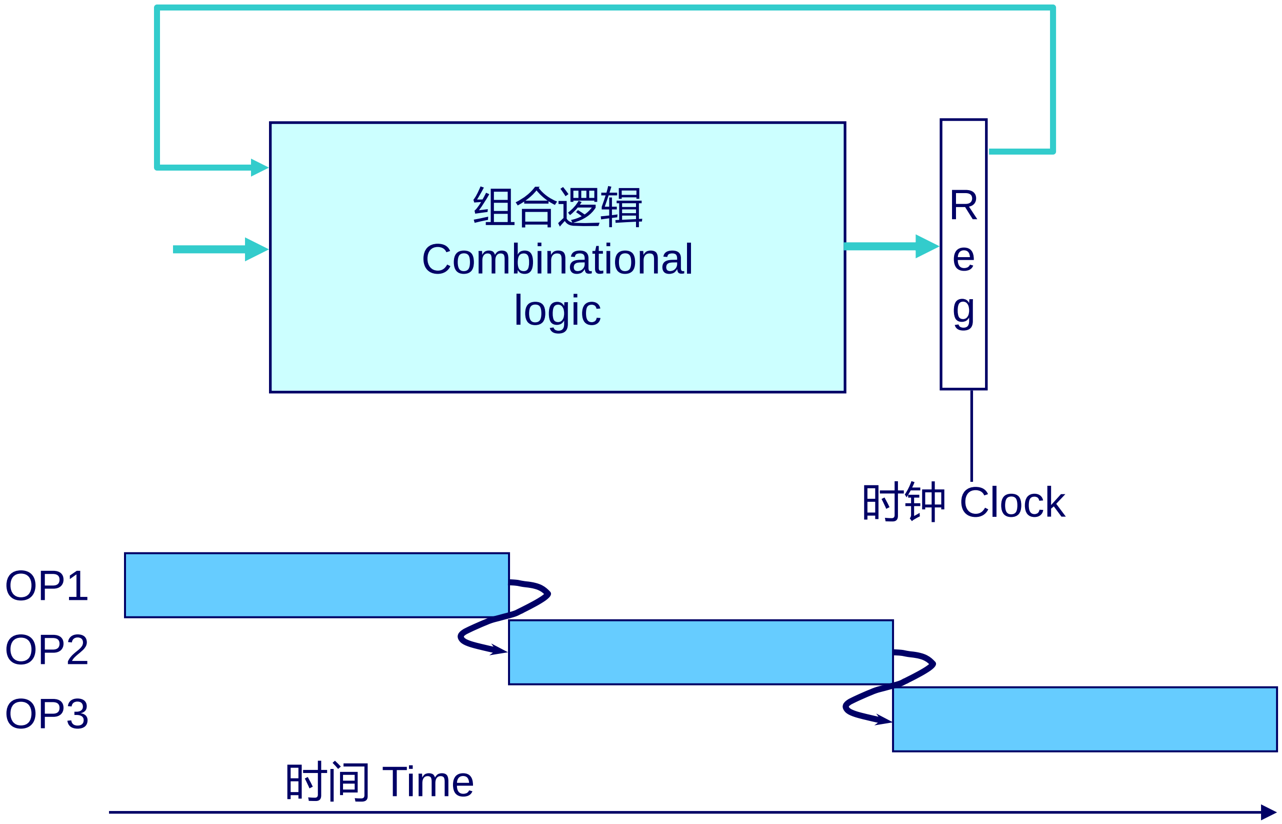


时钟 Clock

时延 Delay = 420 ps, 吞吐量 Throughput = 14.29 GIPS

- 随着流水线深度加深，装载寄存器的开销变得越来越大 As try to deepen pipeline, overhead of loading registers becomes more significant
- 时钟周期花费在装载寄存器的百分比：Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25% 1阶段流水线：20/320
  - 3-stage pipeline: 16.67% 3阶段流水线：60/360
  - 6-stage pipeline: 28.57% 6阶段流水线：120/420
- 现代处理器设计的高速度通过非常深度的流水线获得的 High speeds of modern processor designs obtained through very deep pipelining

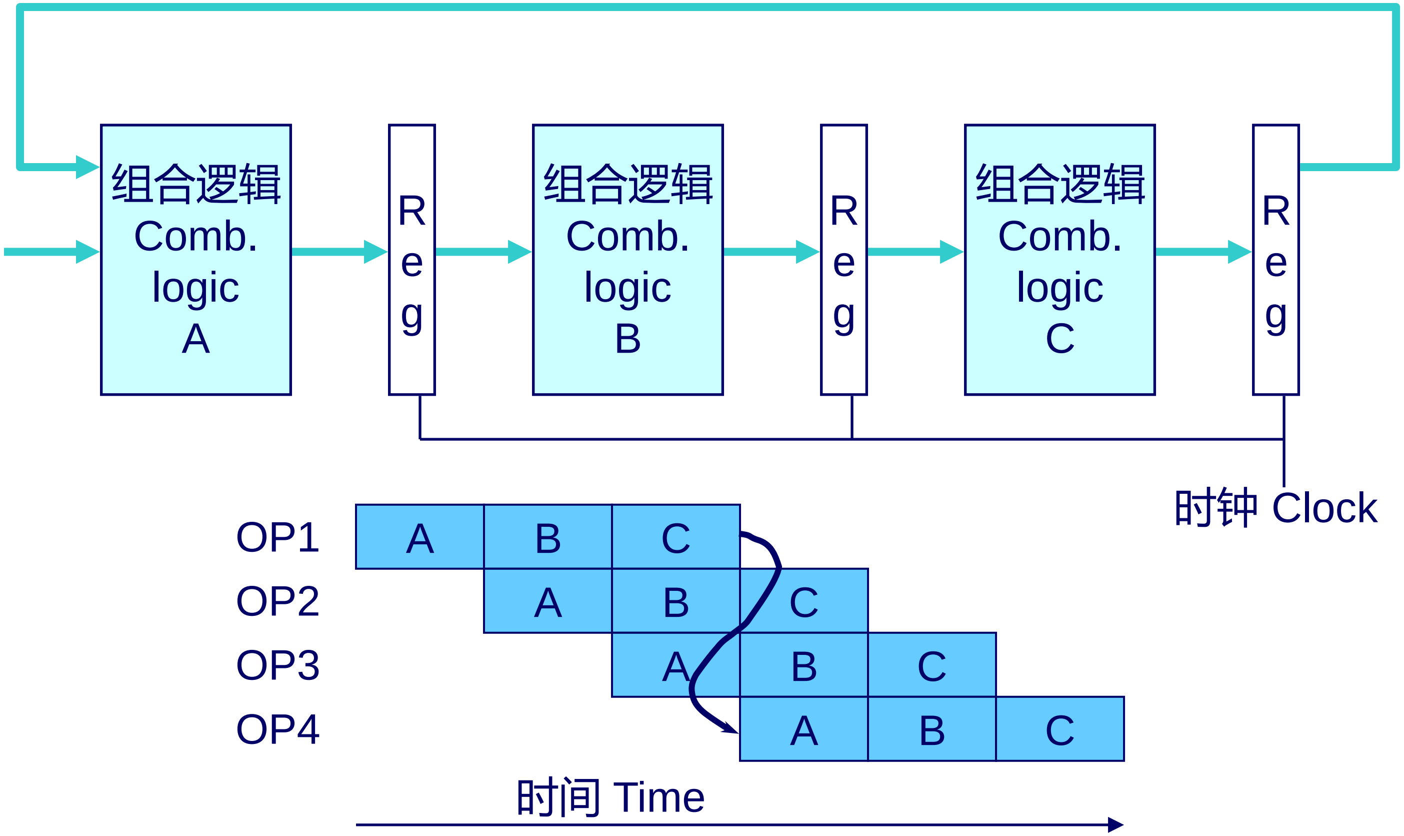
数据相关 Data Dependencies



系统 System

- **每个操作依赖于上一次操作的结果 Each operation depends on result from preceding one**

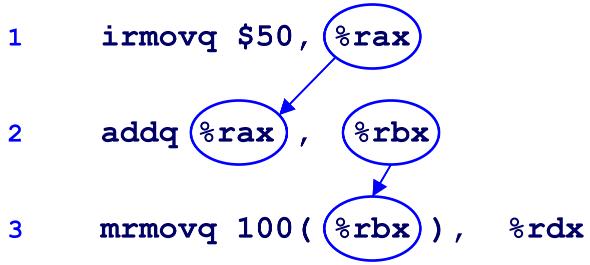
数据冒险 Data Hazards



OP1	A	B	C			
OP2		A	B	C		
OP3			A	B	C	
OP4				A	B	C

- **结果没有及时反馈给下一次操作 Result does not feed back around in time for next operation**
- **流水线改变了系统的行为 Pipelining has changed behavior of system**

处理器中的数据相关 Data Dependencies in Processors

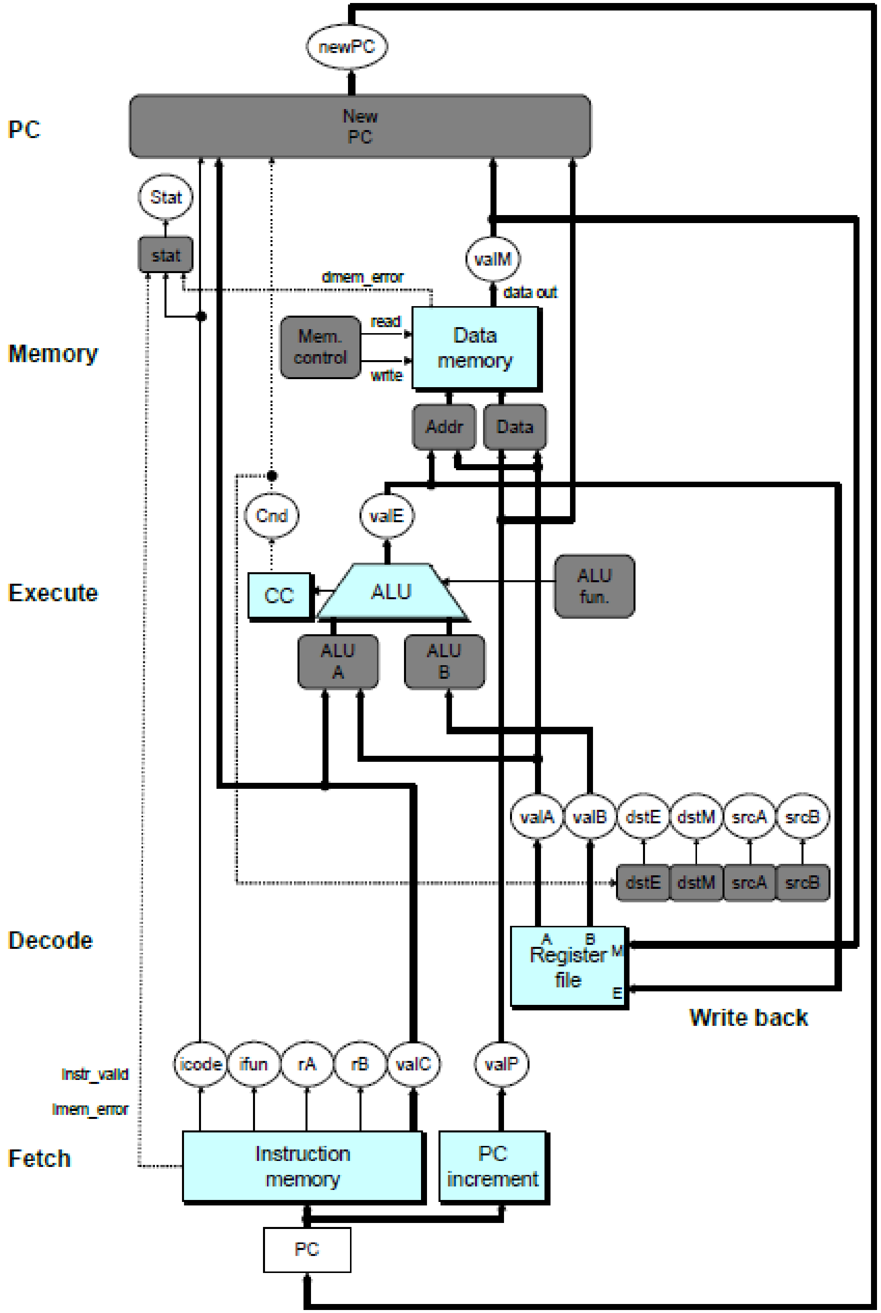


```
irmovq $50, %rax

addq %rax , %rbx

mrmovq 100( %rbx ), %rdx
```

- 一条指令的结果用作另一条指令的操作数 Result from one instruction used as operand for another
  - 写后读（RAW）相关 Read-after-write (RAW) dependency
- 在实际程序中非常常见 Very common in actual programs
- 必须确保流水线能够正确处理这些情况 Must make sure our pipeline handles these properly
  - 得到正确的结果 Get correct results
  - 最小化对性能的影响 Minimize performance impact

SEQ硬件
SEQ Hardware

- 顺序产生各个阶段 Stages occur in sequence
- 一次只有一个操作在进行处理 One operation in process at a time



SEQ+硬件
SEQ+ Hardware

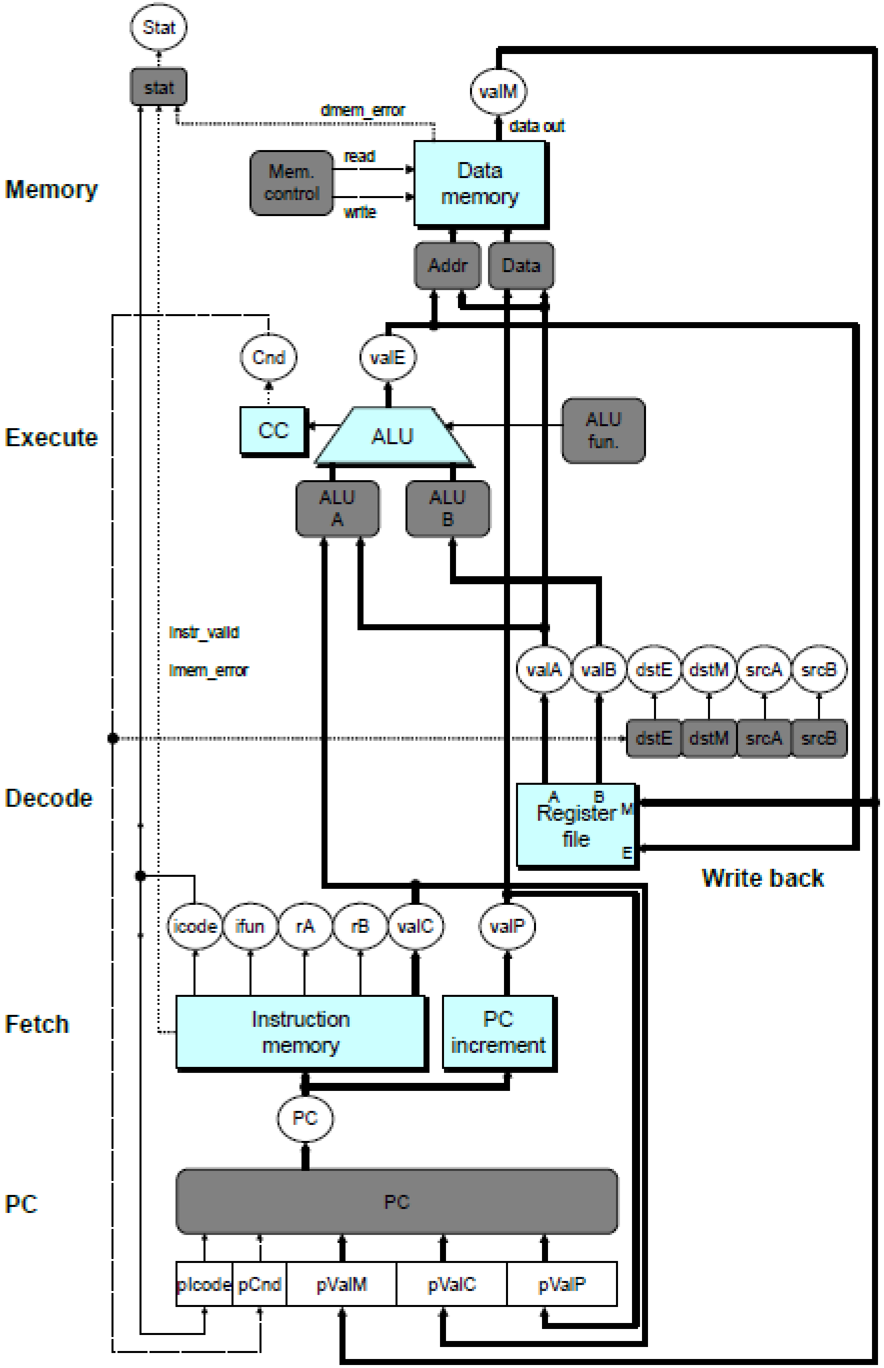
- 仍然是顺序实现 Still sequential implementation
- 记录PC阶段放在开始 Reorder PC stage to put at beginning

PC阶段 PC Stage

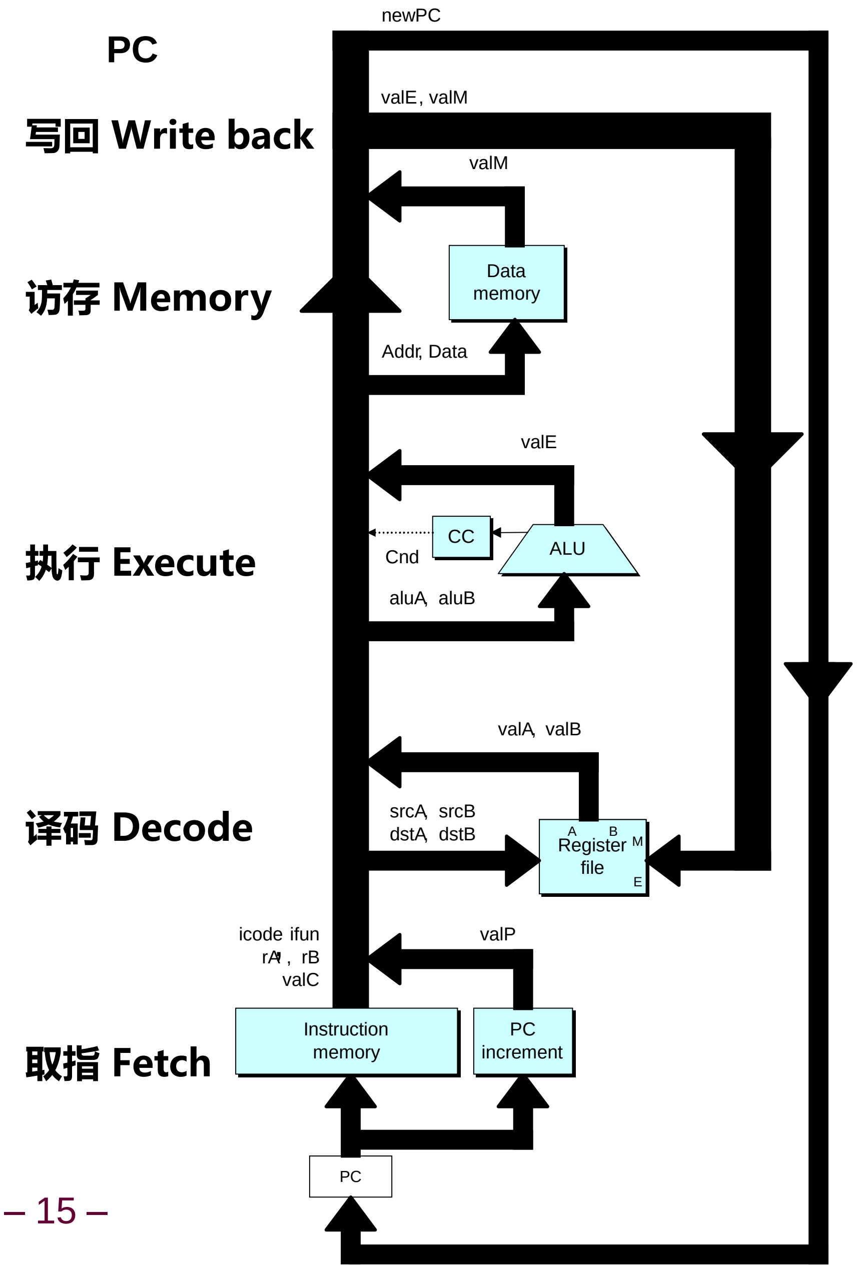
- 任务是为当前指令选择PC Task is to select PC for current instruction
- 根据上条指令计算的结果 Based on results computed by previous instruction

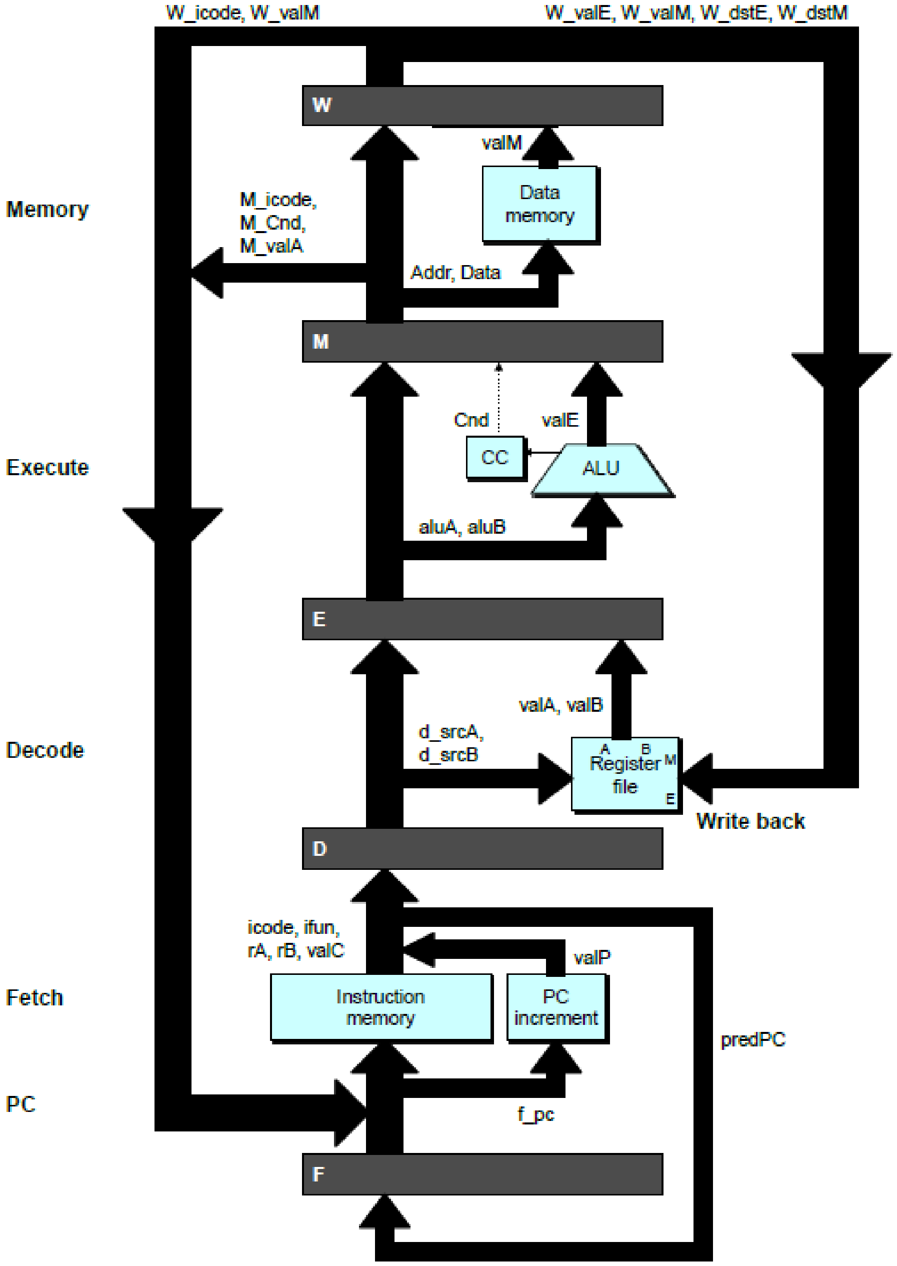
处理器状态 Processor State

- PC不再存储在寄存器中 PC is no longer stored in register
- 但是，可以根据其它存储信息确定PC But, can determine PC based on other stored information



增加流水线寄存器 Adding Pipeline Registers





流水线阶段 Pipeline Stages

取指 Fetch

- 选择当前PC Select current PC
- 读指令 Read instruction
- 计算PC增加值 Compute incremen PC

译码 Decode

- 读程序寄存器 Read program regi

执行 Execute

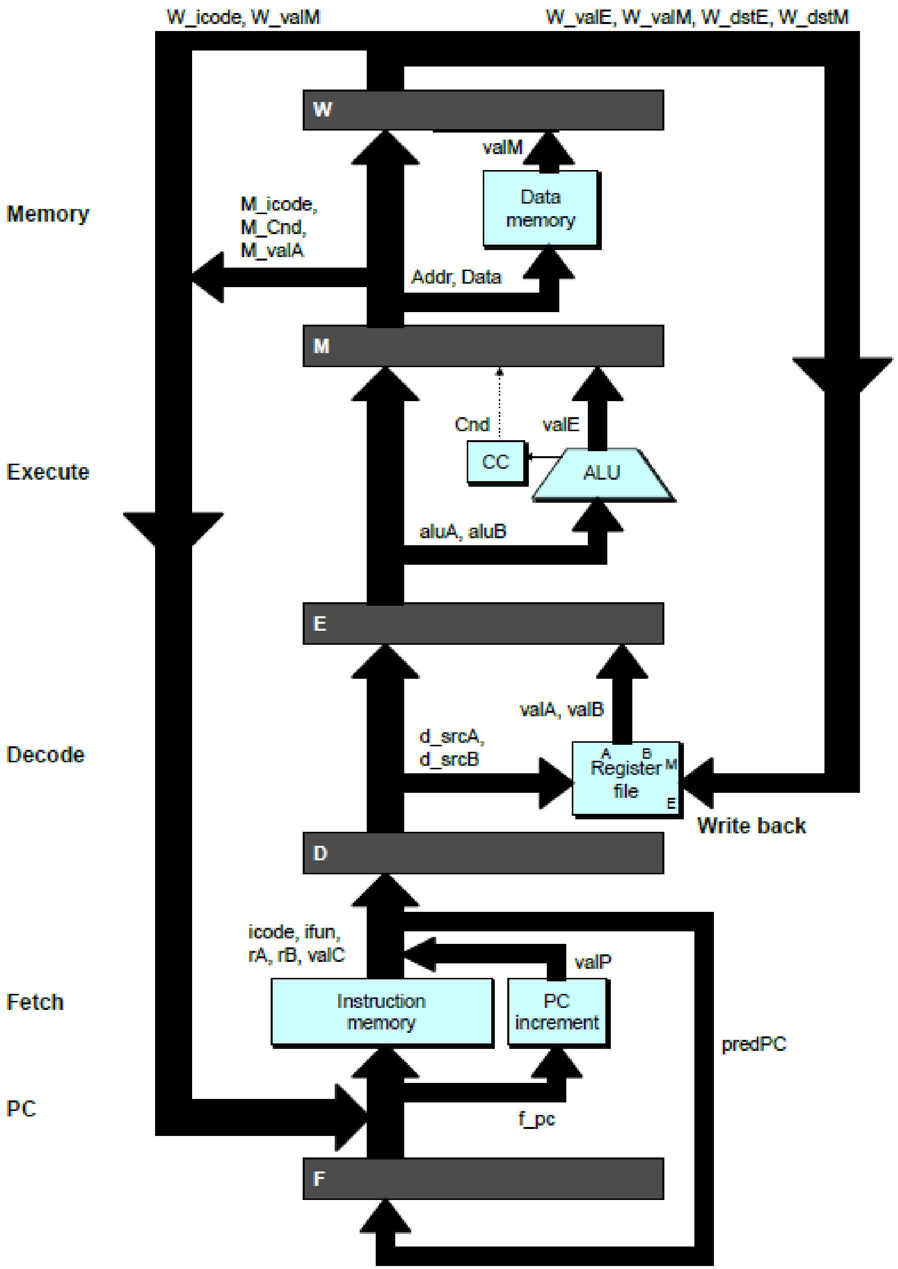
- 操作ALU Operate ALU

访存 Memory

- 读或写数据内存 Read or write dat memory

写回 Write Back

- 更新寄存器文件(堆) Update register file

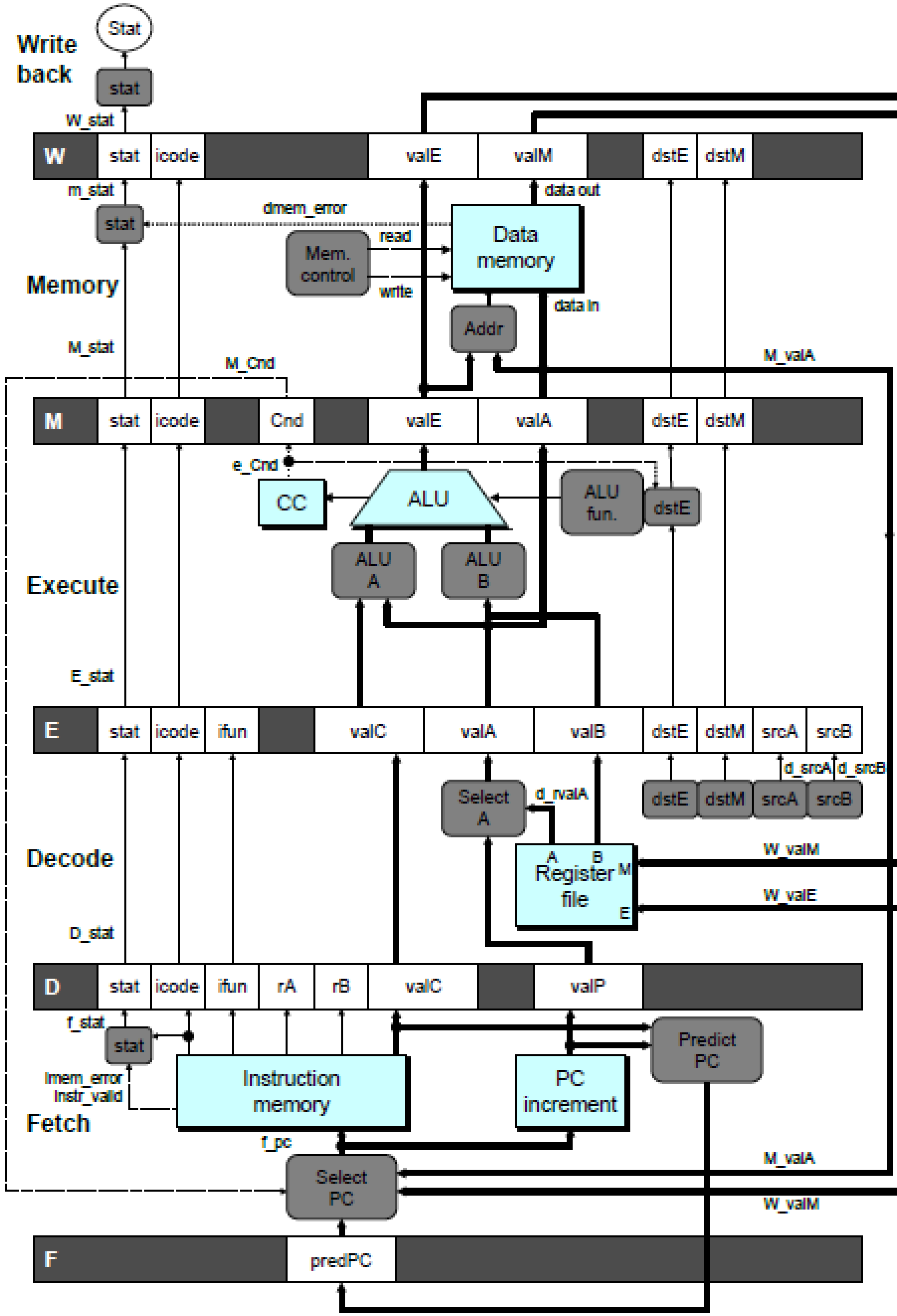


PIPE-硬件 PIPE- Hardware

- 流水线寄存器存储指令执行中的中间值 Pipeline registers hold intermediate values from instruction execution

转发（向前）路径 Forward (Upward) Paths

- 从一个阶段到下一个阶段传递值 Values passed from one stage to next
- 不能回跳到过去的阶段 Cannot jump past stages
  - 例如valC直传通过译码阶段 e.g., valC passes through decode



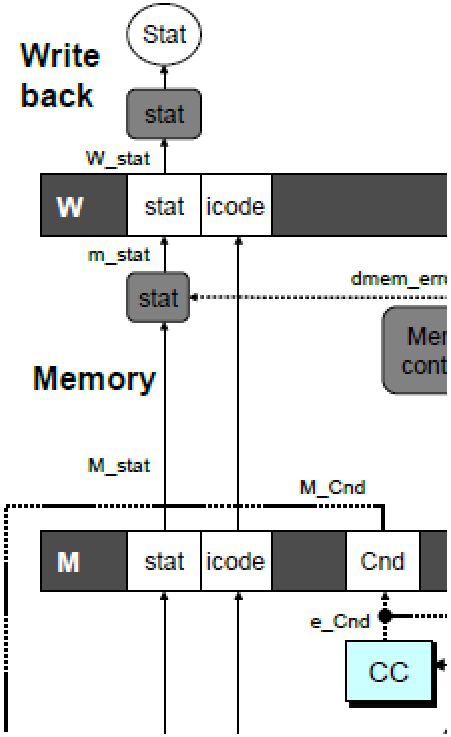
信号命名规则
Signal Naming Conventions

S_Field

- S阶段流水线寄存器中Field字段的值 Value of Field held in stage S pipeline register

s_Field

- S阶段中计算的Field字段的值 Value of Field computed in stage S



反馈路径 Feedback Paths

预测PC Predicted PC

- 猜测下一次PC的值 Guess value of next PC

分支信息 Branch information

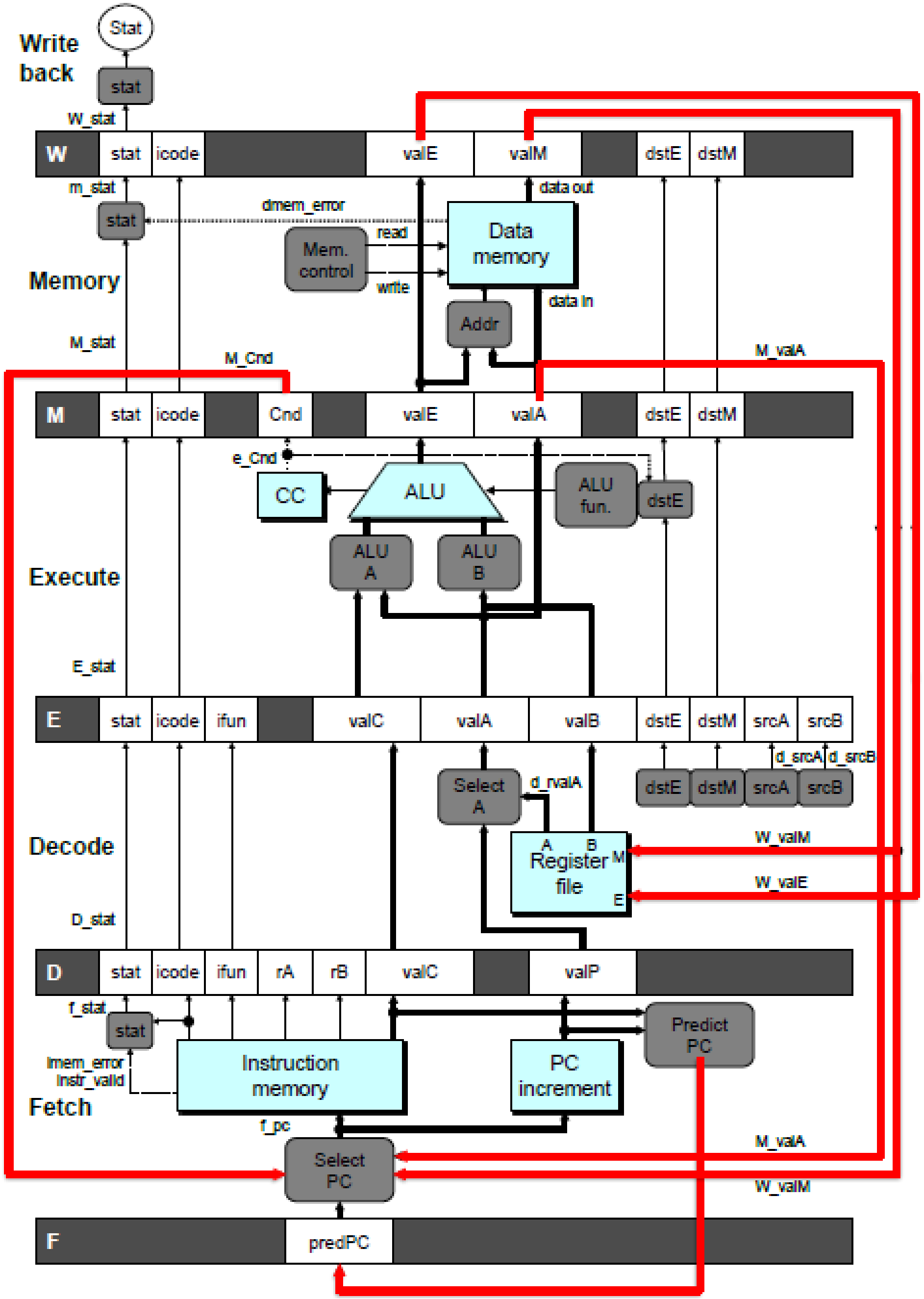
- 跳转/不跳转 Jump taken/not-taken
- 直落或目标地址 Fall-through or target address

返回点 Return point

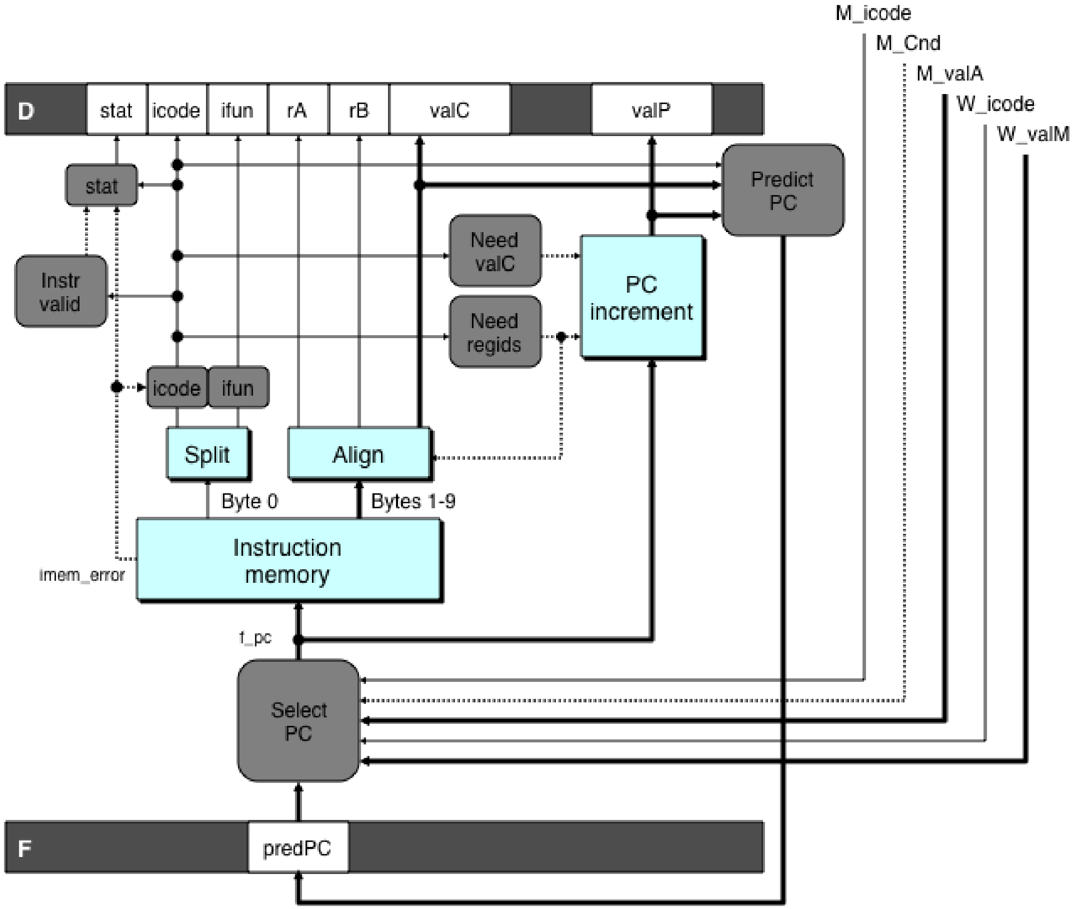
- 从内存读 Read from memory

寄存器更新 Register update

- 寄存器文件的写端口 To register file write ports



预测PC Predicting the PC



- 当前指令已经完成取指阶段后，开始新指令取指阶段 Start fetch of new instruction after current one has completed fetch stage
  - 没有充足的时间来可靠地确定下一条指令 Not enough time to reliably determine next instruction
- 猜测哪条是下一条指令 Guess which instruction will follow
  - 如果预测不正确则恢复 Recover if prediction was incorrect

我们的预测策略
Our Prediction Strategy

不转换控制的指令 Instructions that Don’t Transfer Contro

- 预测下一个PC为valP Predict next PC to be valP
- 总是可靠的 Always reliable

过程调用和无条件跳转指令 Call and Unconditional Jumps

- 预测下一个PC为valC（目标地址） Predict next PC to be valC (destination)
- 总是可靠的 Always reliable

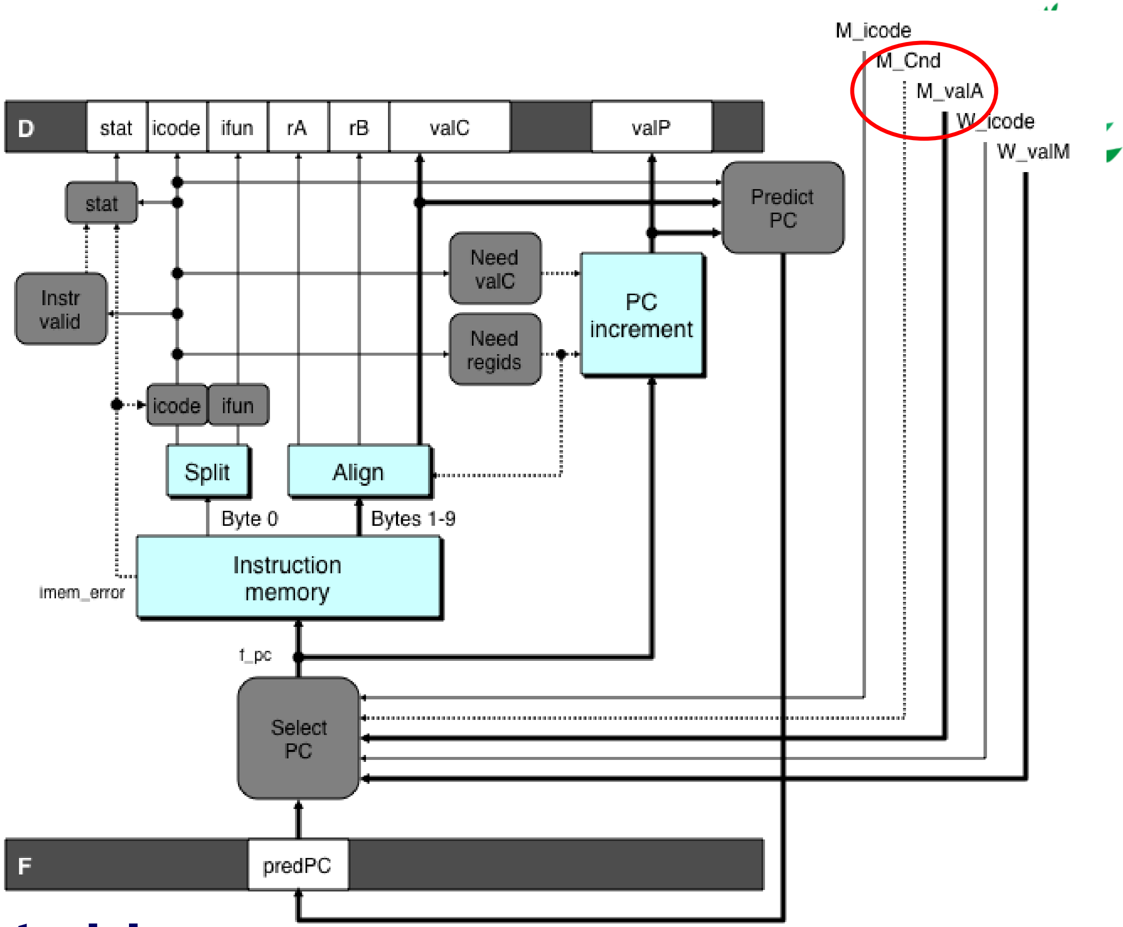
条件跳转指令 Conditional Jumps

- 预测下一个PC为valC（目标地址） Predict next PC to be valC (destination)
- 仅在选择分支时正确 Only correct if branch is taken
  - 典型的正确率为60% Typically right 60% of time

返回指令 Return Instruction

- 不进行预测 Don’t try to predict

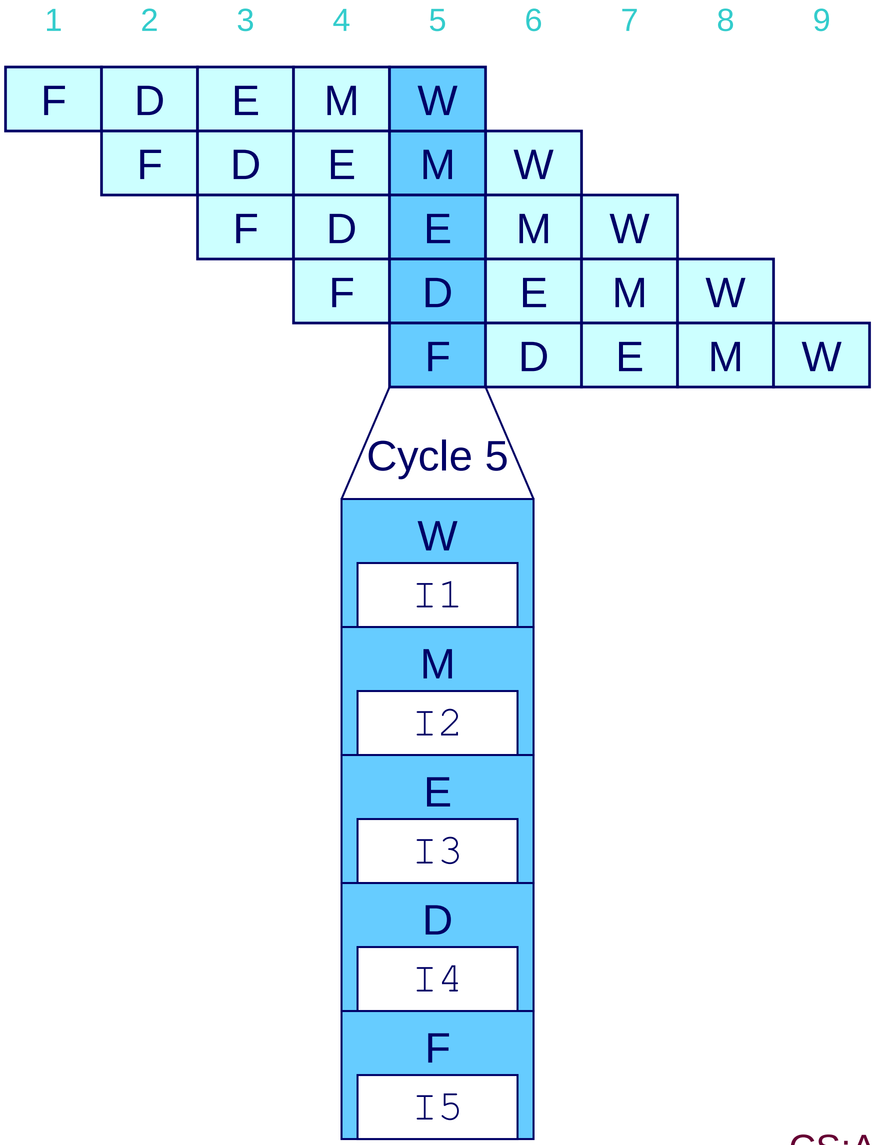
从PC预测错误中恢复 Recovering from PC Misprediction



- 错误预测跳转 Mispredicted Jump
  - 一旦指令到达访存阶段，看到分支条件标志 Will see branch condition flag once instruction reaches memory stage
  - 可以从valA（M_valA值）中得到直落PC Can get fall-through PC from valA (value M_valA)
- 返回指令 Return Instruction
  - 当返回指令到达写回阶段（W_valM）时得到返回PC Will get return PC when `ret` reaches write-back stage (W_valM)

流水线演示 Pipeline Demonstration

```
irmovq   $1,%rax   #I1
irmovq   $2,%rcx   #I2
irmovq   $3,%rdx   #I3
irmovq   $4,%rbx   #I4
halt               #I5
```

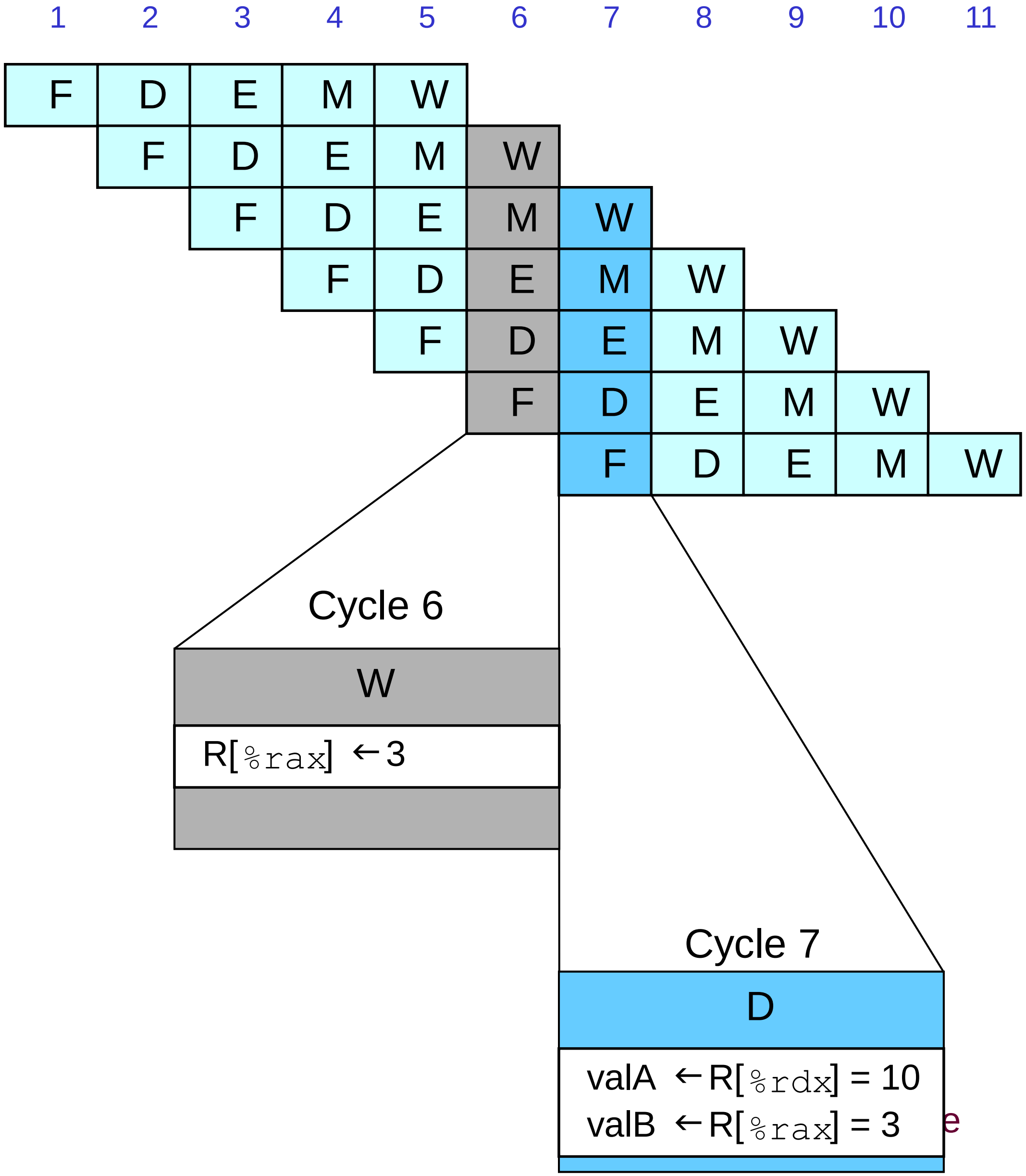


1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	E	M	W			
		F	D	E	M	W		
			F	D	E	M	W	
				F	D	E	M	W

文件 File: `demo-basic.ys`

数据相关：3条空指令
Data Dependencies: 3 Nop's

```
# demo-h3.ys

0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```



1	2	3	4	5	6	7	8	9	10	11
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		
					F	D	E	M	W	
						F	D	E	M	W

Cycle 6

W

R[%rax] ← 3

Cycle 7

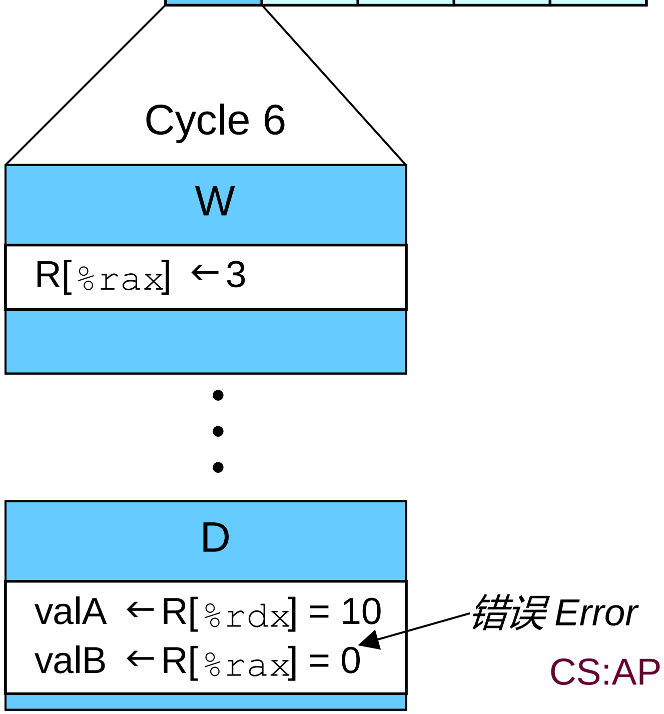
D

valA ← R[%rdx] = 10
valB ← R[%rax] = 3

数据相关：2条空指令
Data Dependencies: 2 Nop’s

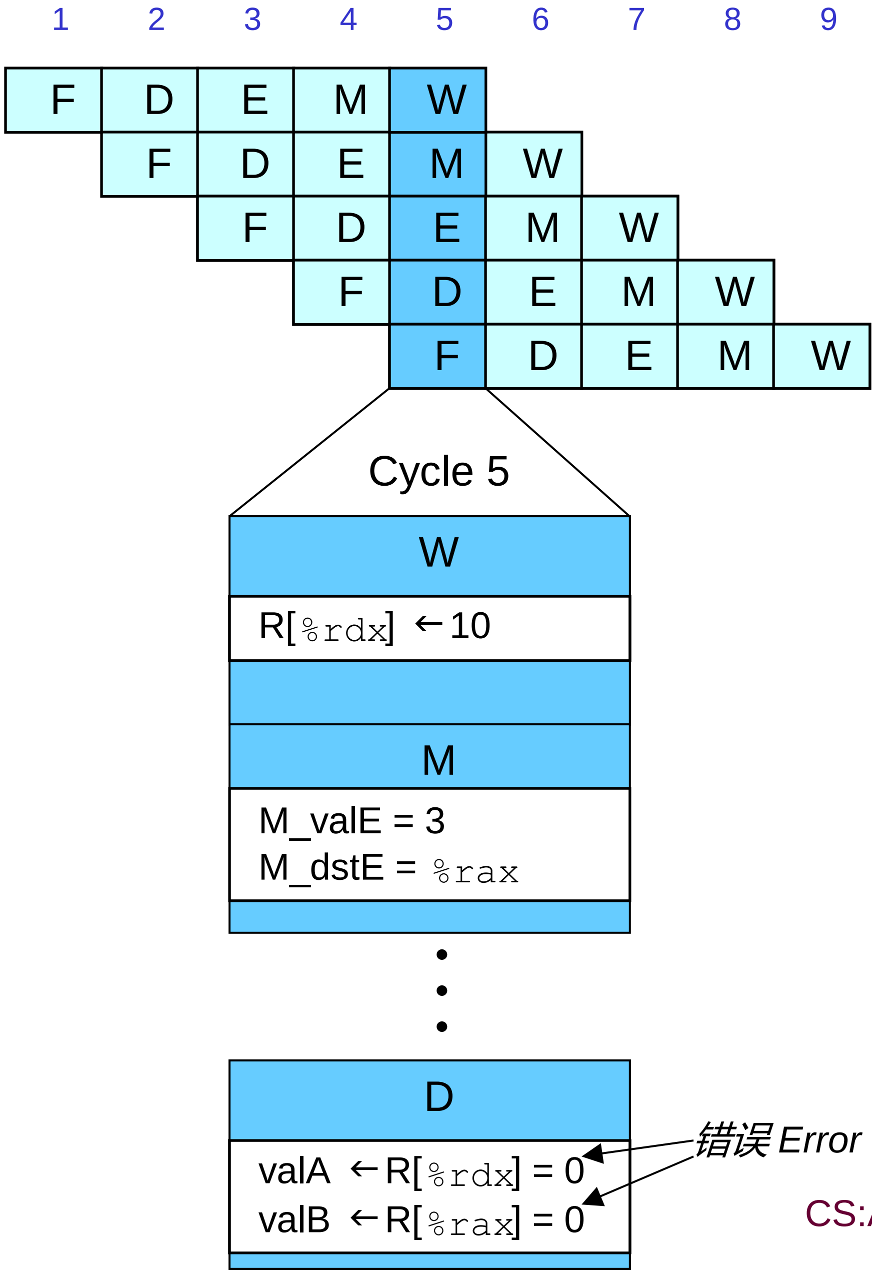
```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

1	2	3	4	5	6	7	8	9	10
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	
					F	D	E	M	W



数据相关：1条空指令 Data Dependencies: 1 Nop

```
# demo-h1.ys

0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```



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

Cycle 5

W

R[%rdx] ← 10

M

M_valE = 3
M_dstE = %rax

•
•
•

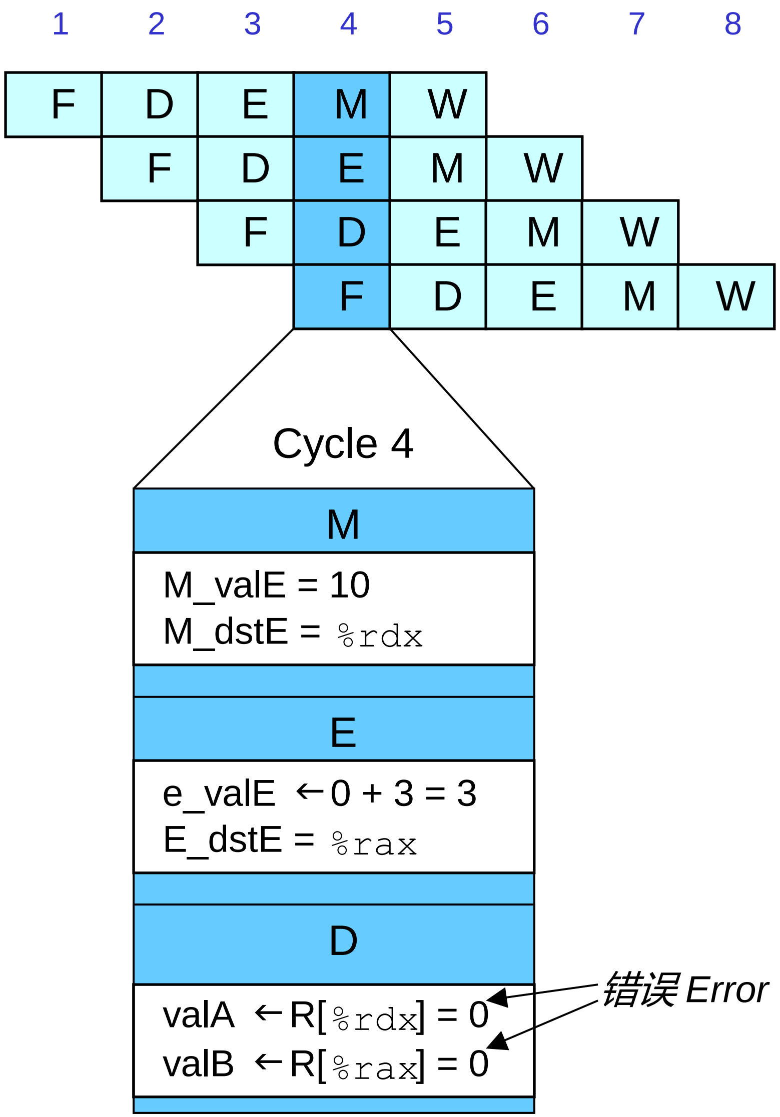
D

valA ← R[%rdx] = 0
valB ← R[%rax] = 0

错误 Error

数据相关：无空指令 Data Dependencies: No Nop

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



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

Cycle 4

M

M_valE = 10
M_dstE = %rdx

E

e_valE ← 0 + 3 = 3
E_dstE = %rax

D

valA ← R[%rdx] = 0
valB ← R[%rax] = 0

错误 Error

分支预测错误示例
Branch Misprediction Example

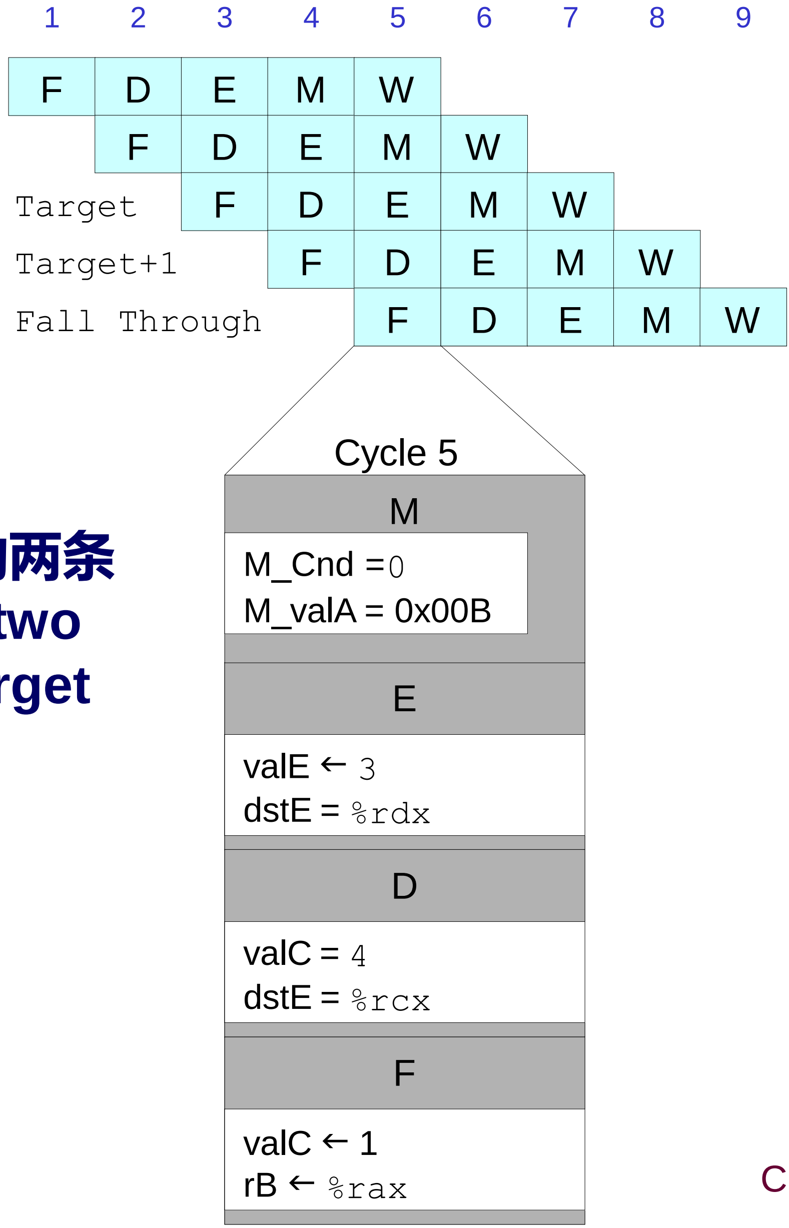
demo-j.ys

```
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
```

- **应该仅执行前8条指令 Should only execute first 8 instructions**

分支预测错误跟踪 Branch Misprediction Trace

```
# 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
0x000	F	D	E	M	W				
0x002		F	D	E	M	W			
0x019			F	D	E	M	W		
0x023				F	D	E	M	W	
0x00b					F	D	E	M	W

Cycle 5

M

M_Cnd = 0
M_valA = 0x00B

E

valE ← 3
dstE = %rdx

D

valC = 4
dstE = %rcx

F

valC ← 1
rB ← %rax

- 不正确地执行分支目标处的两条指令 Incorrectly execute two instructions at branch target

返回示例
Return Example

demo-ret.ys

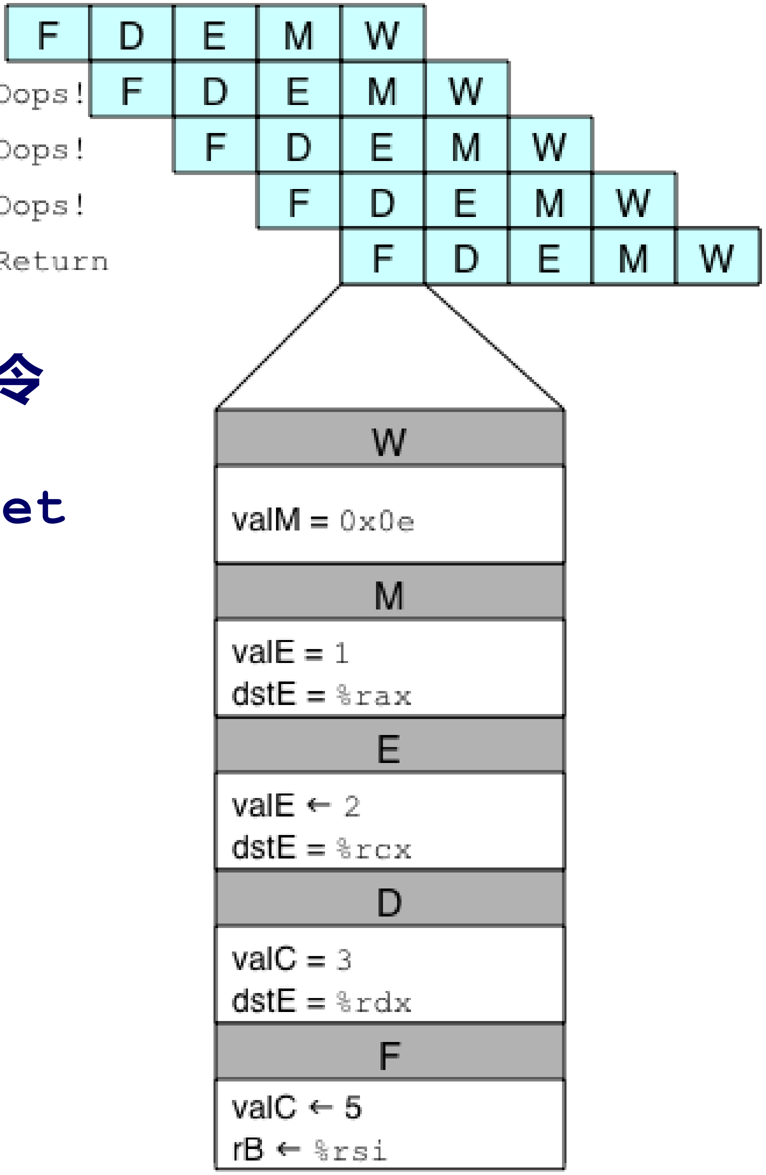
```
0x000:    irmovq Stack,%rsp     # Intialize stack pointer
0x00a:    nop                   # Avoid hazard on %rsp
0x00b:    nop
0x00c:    nop
0x00d:    call p                # Procedure call
0x016:    irmovq $5,%rsi        # Return point
0x020:    halt
0x020: .pos 0x20
0x020: p: nop              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

- **需要很多空指令来避免数据冒险 Require lots of nops to avoid data hazards**

不正确返回示例 Incorrect Return Example

```
# demo-ret

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

- **错误执行ret后面的3条指令**
  Incorrectly execute 3 instructions following `ret`



W

valM = 0x0e

M

valE = 1
dstE = %rax

E

valE ← 2
dstE = %rcx

D

valC = 3
dstE = %rdx

F

valC ← 5
rB ← %rsi

流水线小结 Pipeline Summary

概念 Concept

- 把指令执行分成5个阶段 Break instruction execution into 5 stages
- 以流水线模式运行指令 Run instructions through in pipelined mode

限制 Limitations

- 当指令流太紧密时不能处理指令之间的相关性 Can’t handle dependencies between instructions when instructions follow too closely
- 数据相关 Data dependencies
  - 一条指令写寄存器，然后一条指令读它 One instruction writes register, later one reads it
- 控制相关 Control dependency
  - 指令设置PC的方式，不是流水线正确预测的结果 Instruction sets PC in way that pipeline did not predict correctly
  - 预测失误的分支和返回 Mispredicted branch and return

修正流水线 Fixing the Pipeline

- 下一次课完成这个工作 We’ll do that next time