CS:APP Chapter 4 Computer Architecture Pipelined Implementation Part II

流水线实现 第二部分

任课教师：

宿红毅　张艳　黎有琦　颜珂

原作者：

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

Carnegie Mellon University

概述 Overview

使流水线处理器工作 Make the pipelined processor work!

数据冒险 Data Hazards

- 以寄存器 R 作为源的指令紧跟在以寄存器 R 作为目的的指令之后 Instruction having register R as source follows shortly after instruction having register R as destination
- 常见情况，不想减慢流水线 Common condition, don't want to slow down pipeline

控制冒险 Control Hazards

- 错误预测条件分支 Mispredict conditional branch
  - 我们的设计预测所有分支为选择分支 Our design predicts all branches as being taken
  - 朴素流水线执行两条额外指令 Naïve pipeline executes two extra instructions
- 获得ret指令的返回地址 Getting return address for `ret` instruction
  - 朴素流水线执行三条额外指令 Naïve pipeline executes three extra instructions

概述 Overview

使流水线处理器工作 Make the pipelined processor work!

确保它真正工作 Making Sure It Really Works

- **如果多种特殊情况同时发生会怎么样? What if multiple special cases happen simultaneously?**

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 incremented PC

译码 Decode

- 读程序寄存器 Read program registers

执行 Execute

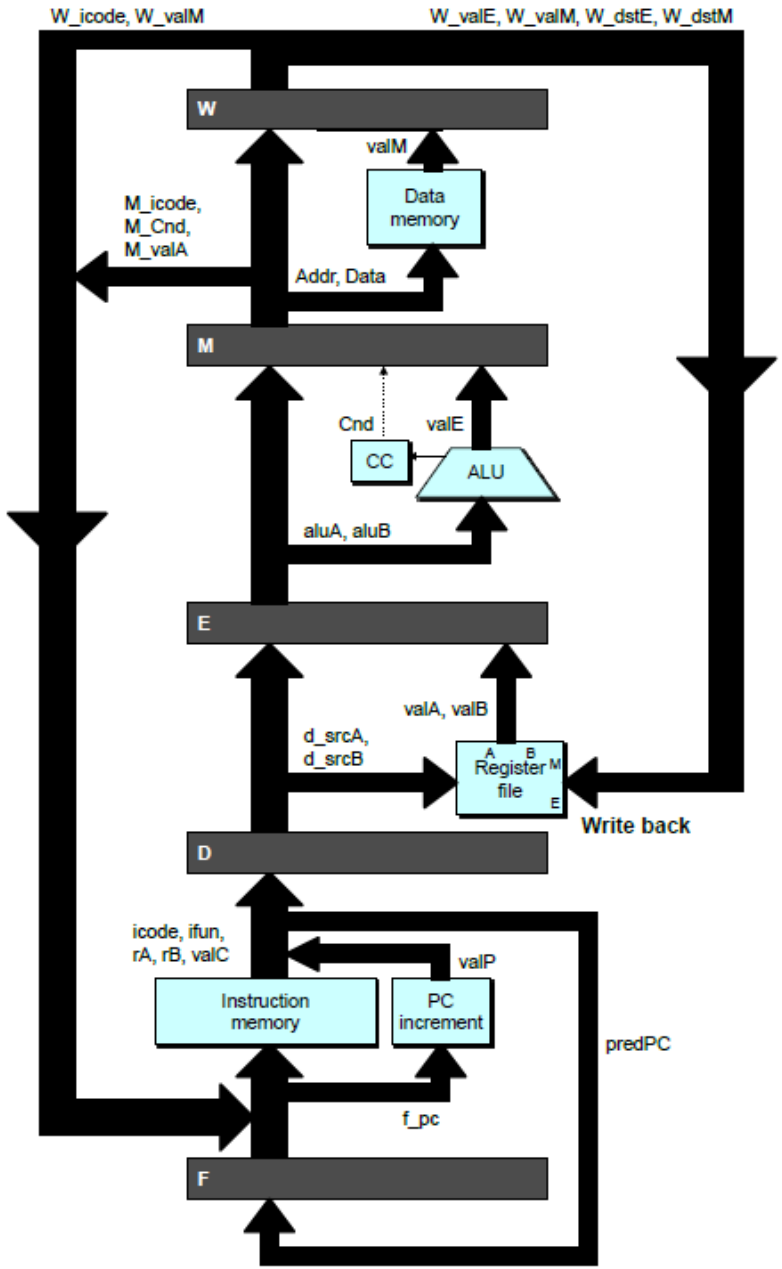
- 操作ALU Operate ALU

内存 Memory

- 读或写数据内存 Read or write data memory

写回 Write Back

- 更新寄存器文件（堆） Update register file

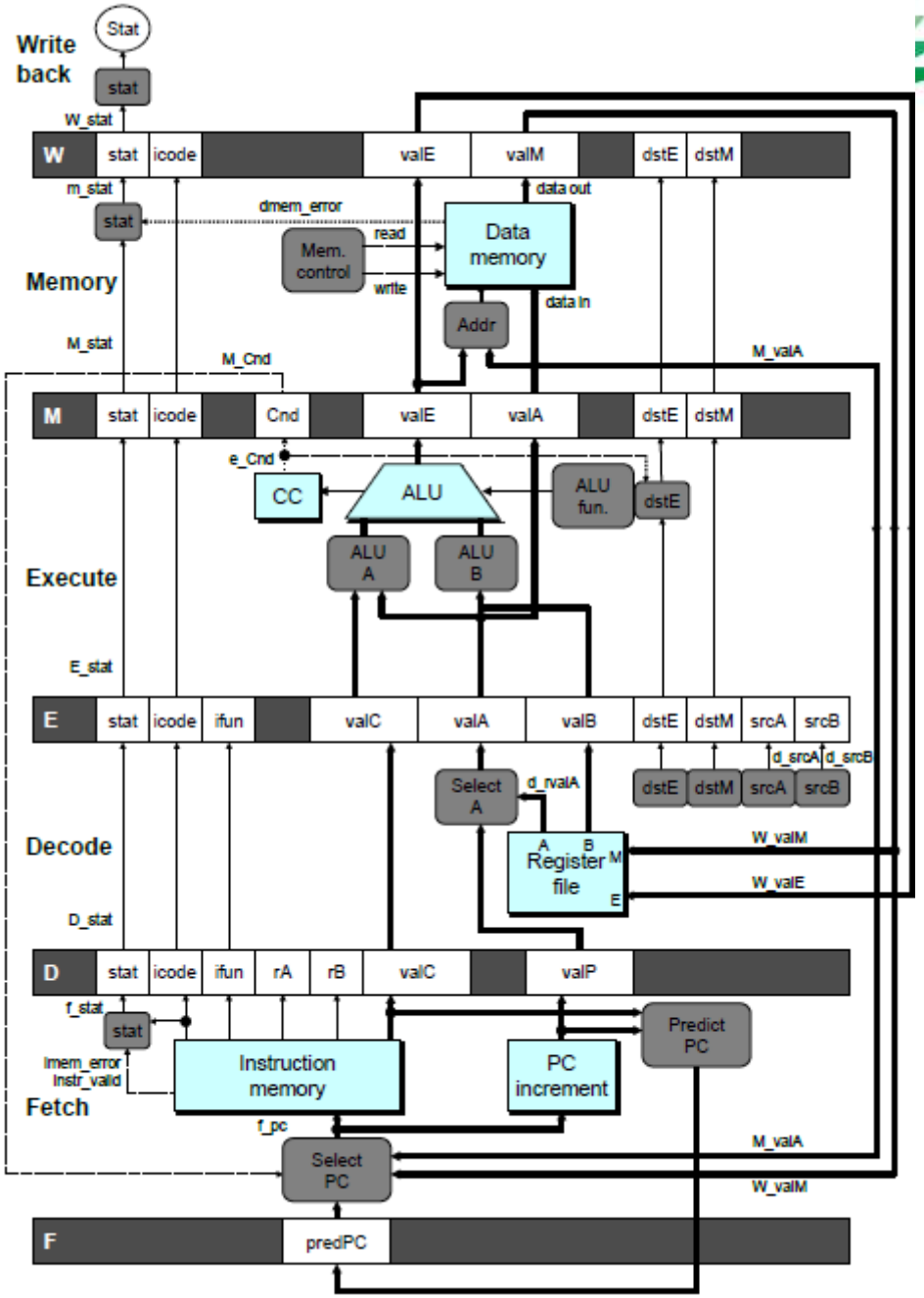


流水线硬件 PIPE- Hardware

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

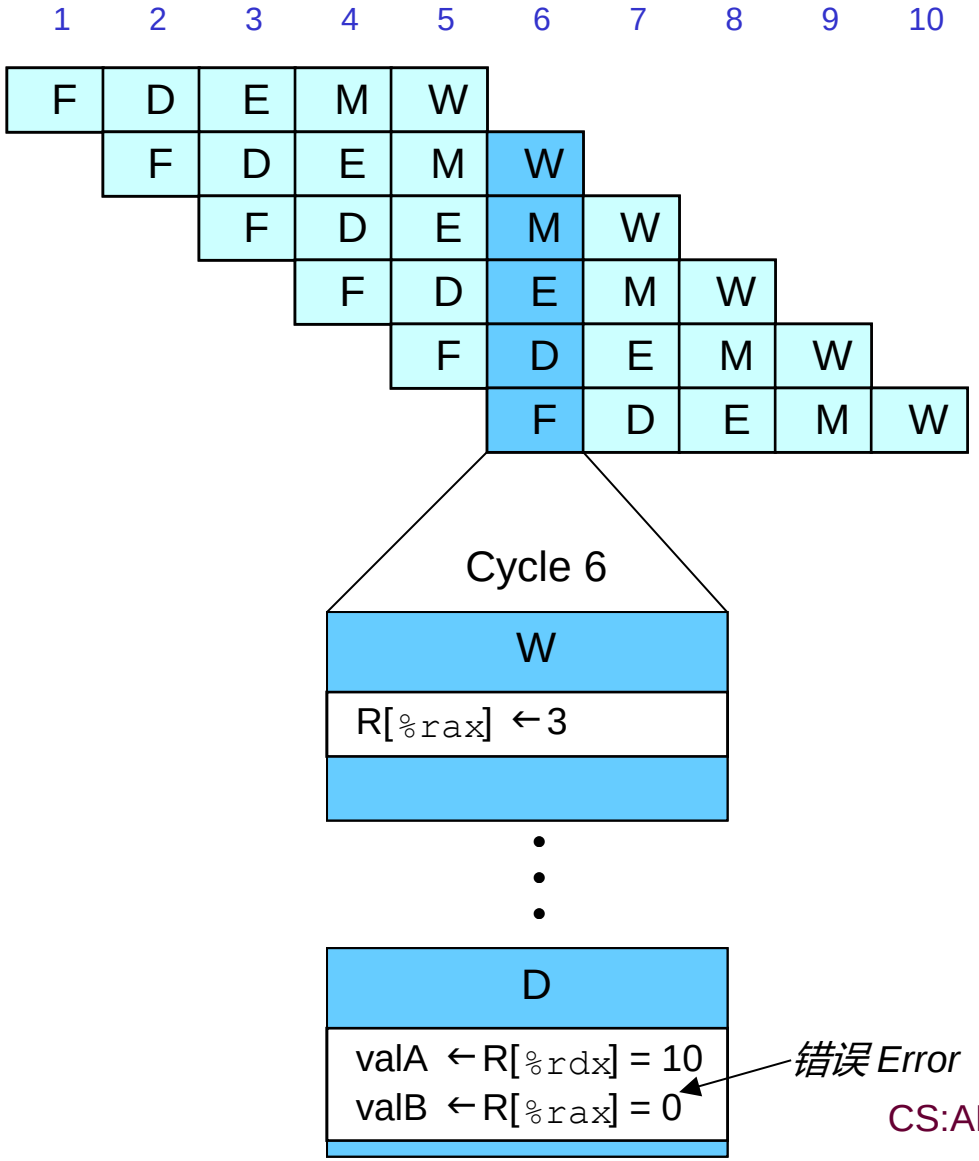
转发（向前）路径 Forward (Upward) Paths

- 从一个阶段向下一个阶段传递值 Values passed from one stage to next
- 不能跳转到过去阶段 Cannot jump past stages
  - 例如valC直传通过译码阶段 e.g., valC passes through decode



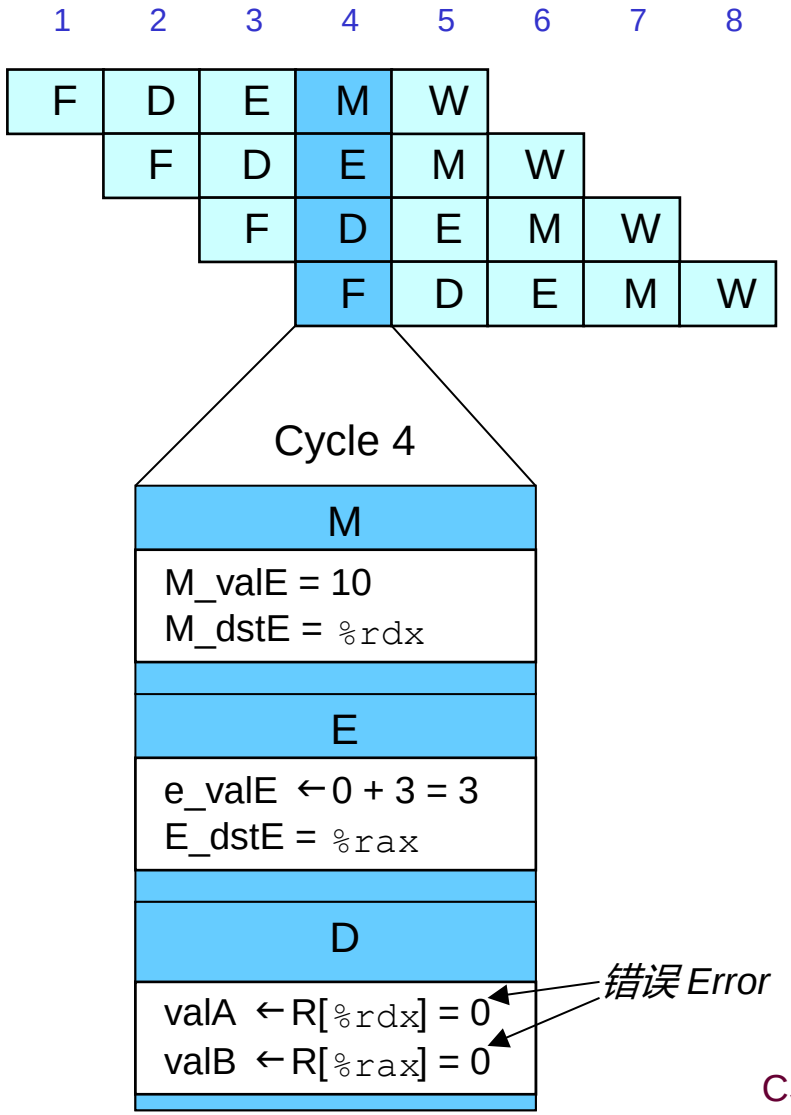
数据相关：2条空指令
Data Dependencies: 2 Nop's

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



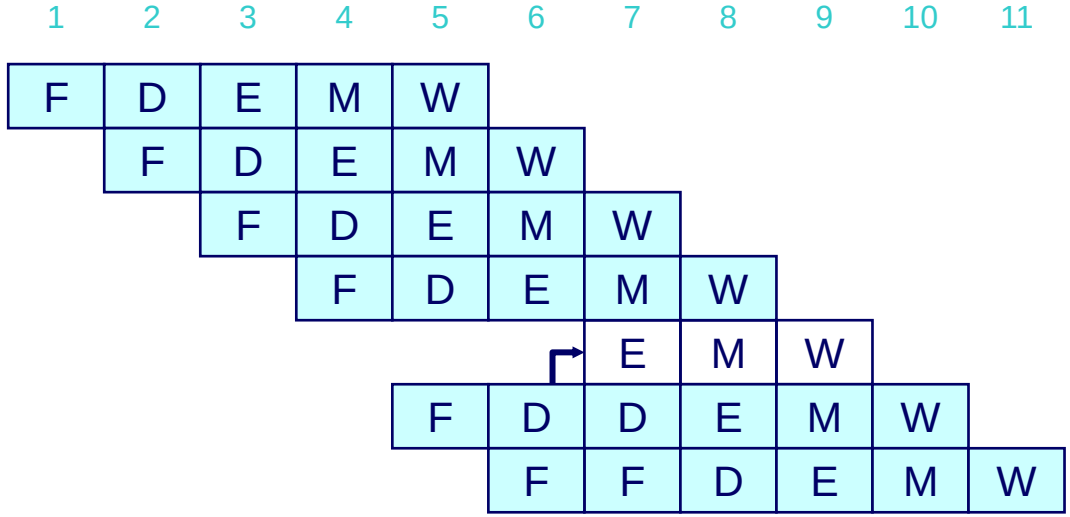
数据相关：没有空指令
Data Dependencies: No Nop

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



暂停解决数据相关
Stalling for Data Dependencies

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



- 如果一条写寄存器指令后面指令流太紧密，需要慢下来 If instruction follows too closely after one that writes register, slow it down
- 保持指令在译码阶段 Hold instruction in decode
- 动态注入空指令到执行阶段 Dynamically inject nop into execute stage

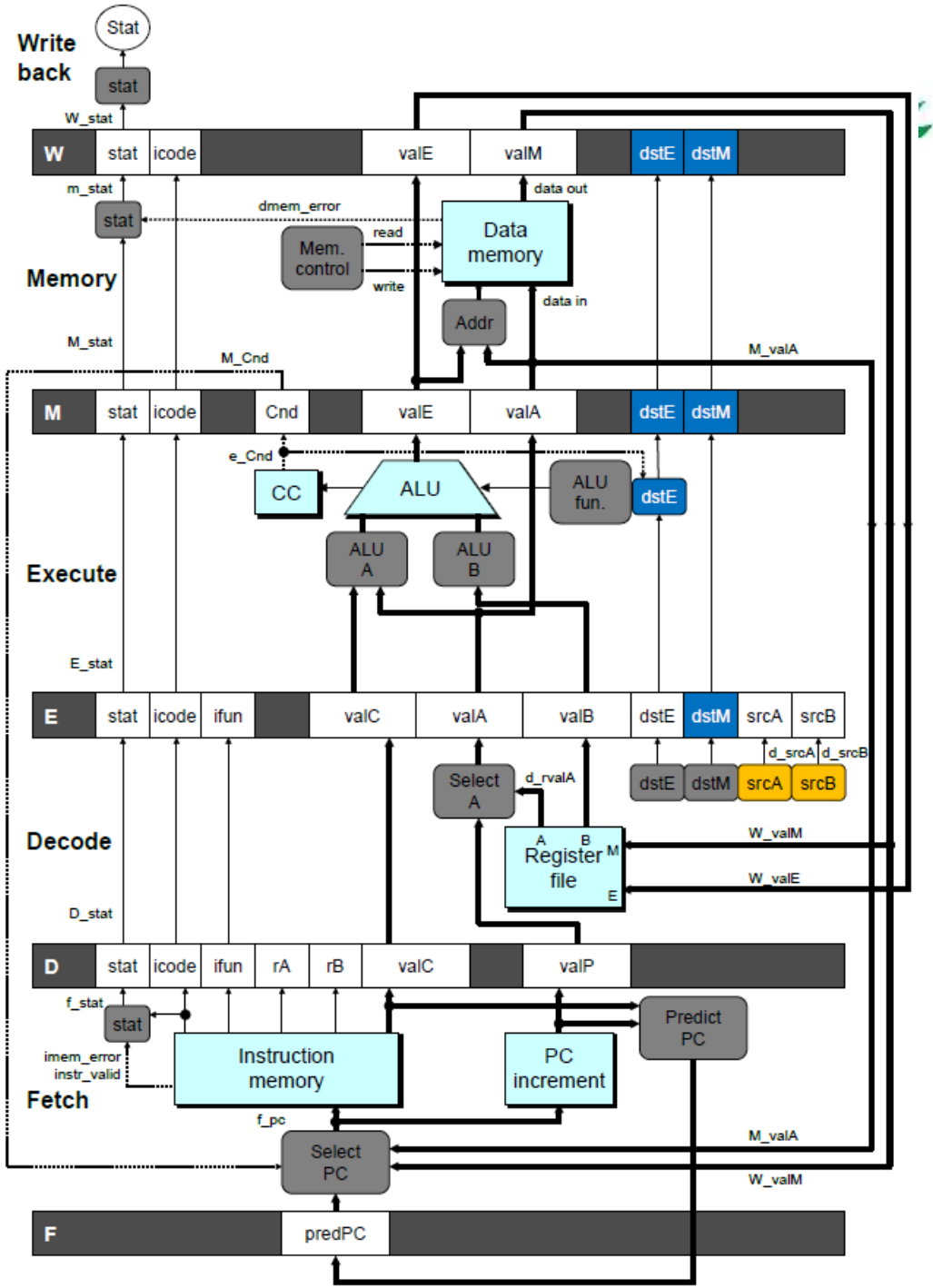
暂停条件 Stall Condition

源寄存器 Source Registers

- 译码阶段的当前指令的srcA和srcB srcA and srcB of current instruction in decode stage

目的寄存器 Destination Registers

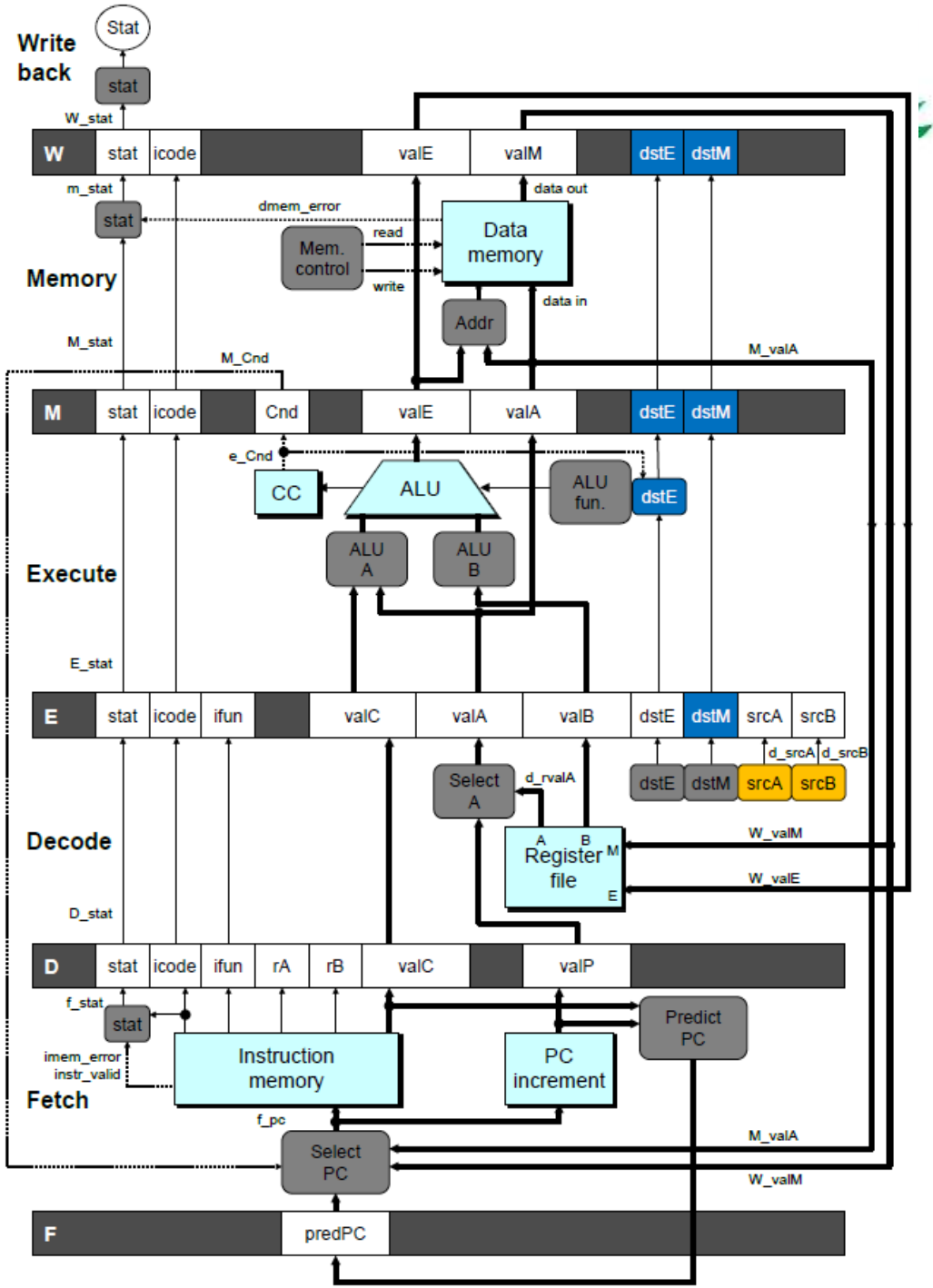
- dstE和dstM字段 dstE and dstM fields
- 指令在执行、内存和写回阶段 Instructions in execute, memory, and write-back stages



暂停条件 Stall Condition

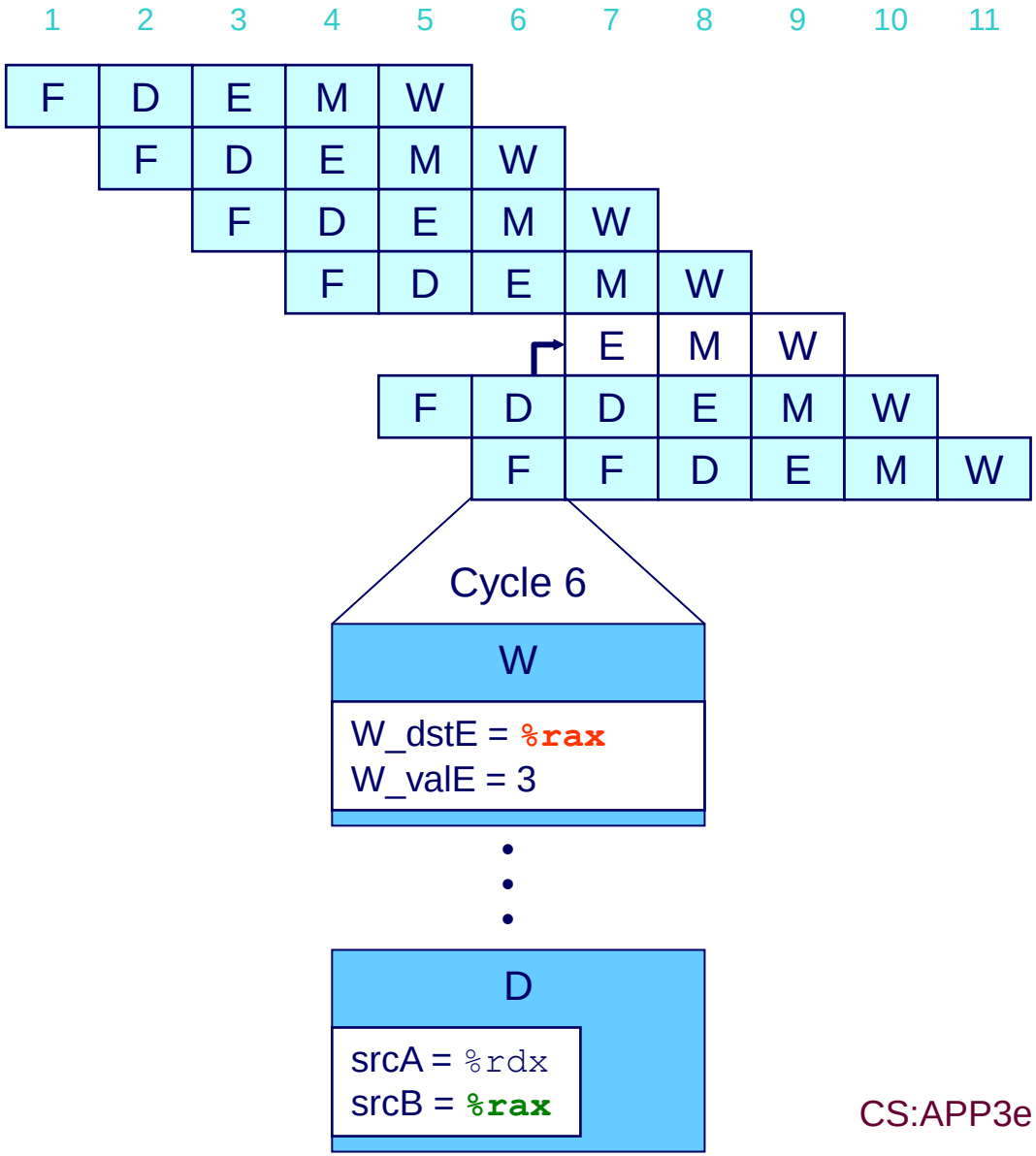
特殊情况 Special Case

- 不要暂停ID为15（0xF）的寄存器 Don't stall for register ID 15 (0xF)
  - 指明没有寄存器操作数 Indicates absence of register operand
  - 或失败的条件传送 Or failed cond. move



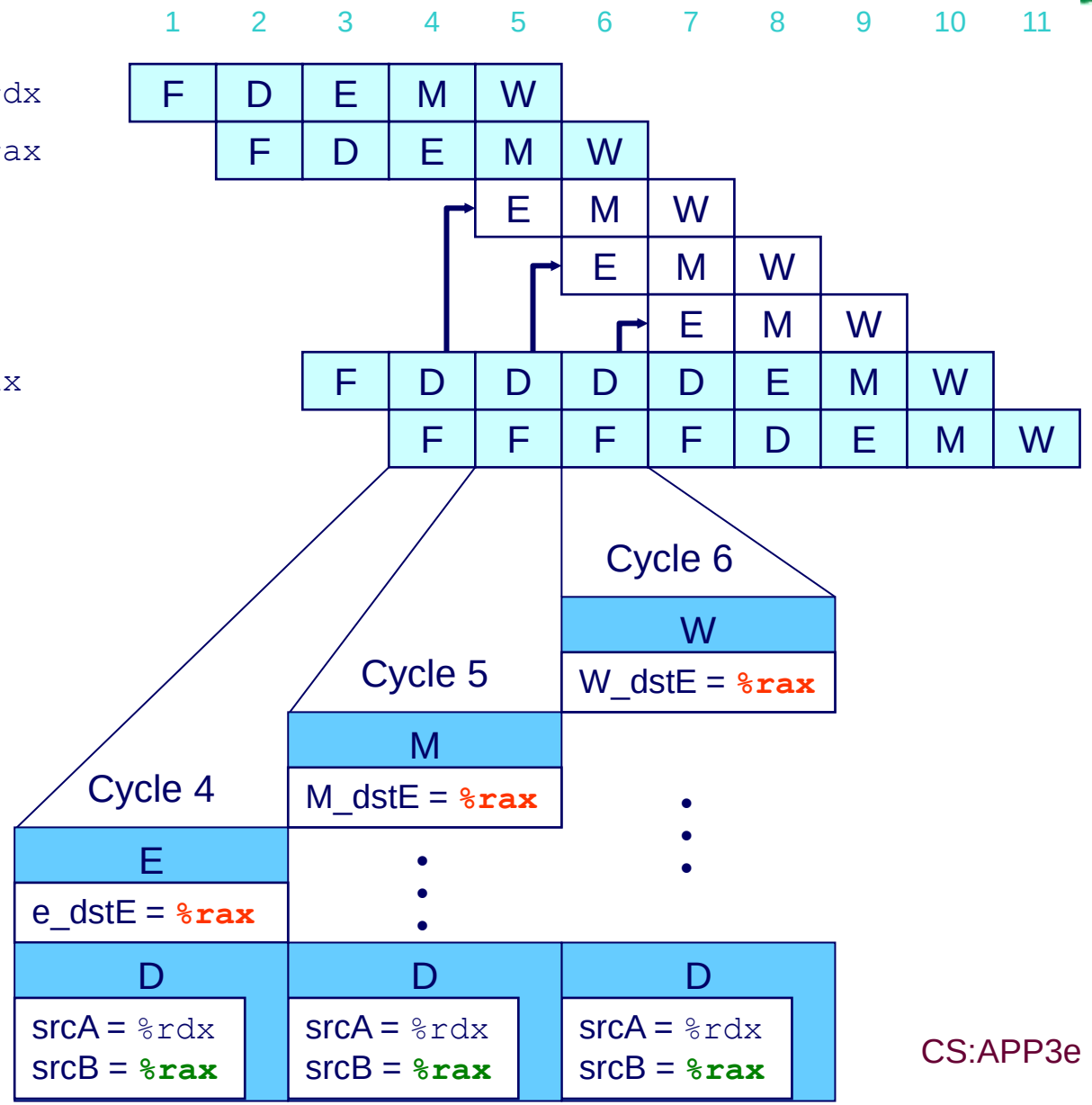
检测暂停条件 Detecting Stall Condition

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



暂停三次 Stalling X3

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



当暂停时发生了什么?
What Happens When Stalling?

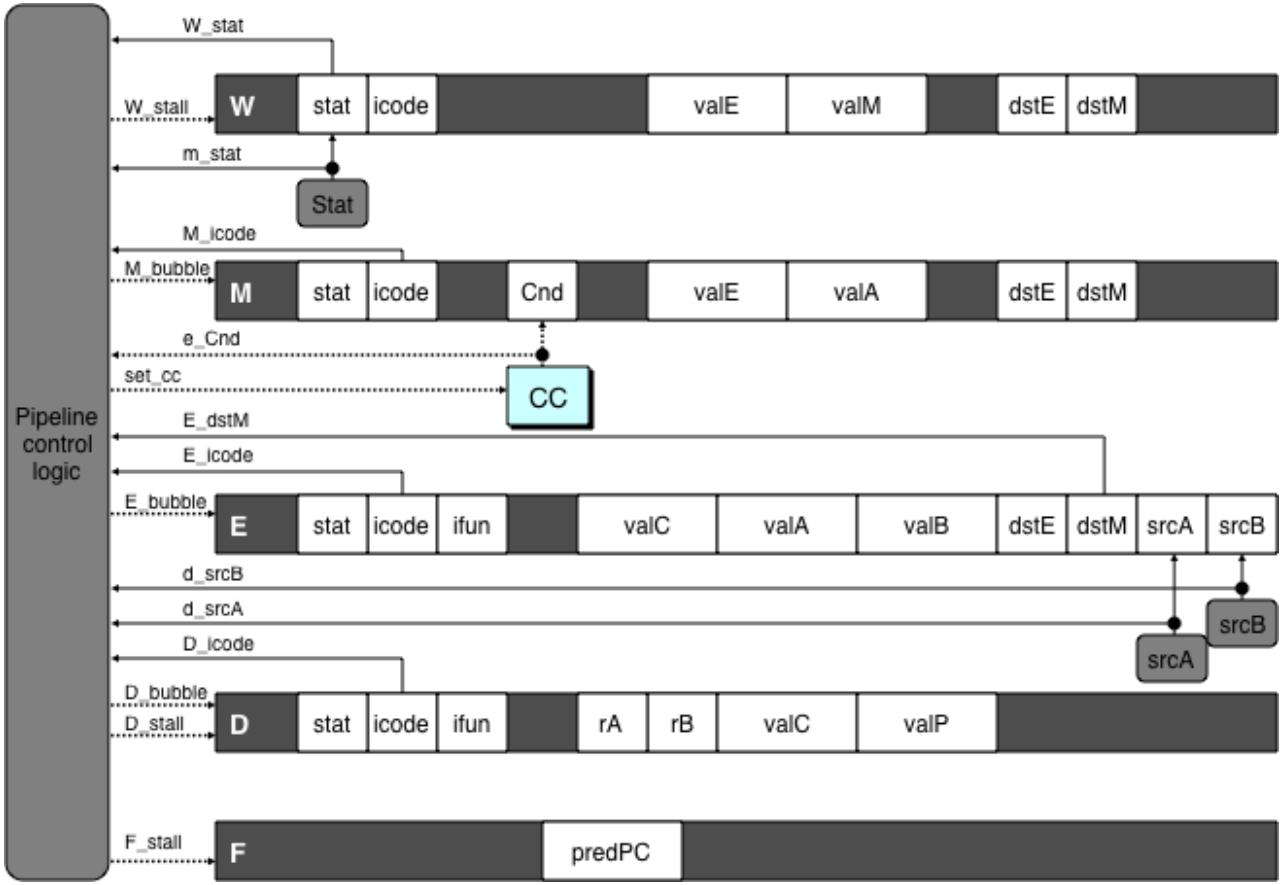
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

Cycle 8

Write Back	***bubble***
Memory	***bubble***
Execute	`0x014: addq %rdx,%rax`
Decode	`0x016: halt`
Fetch	

- 暂停指令保持在译码阶段 Stalling instruction held back in decode stage
- 后续指令停留在取指阶段 Following instruction stays in fetch stage
- 气泡注入到执行阶段 Bubbles injected into execute stage
  - 类似于动态产生空指令 Like dynamically generated nop's
  - 移动通过后面的阶段 Move through later stages

实现暂停 Implementing Stalling

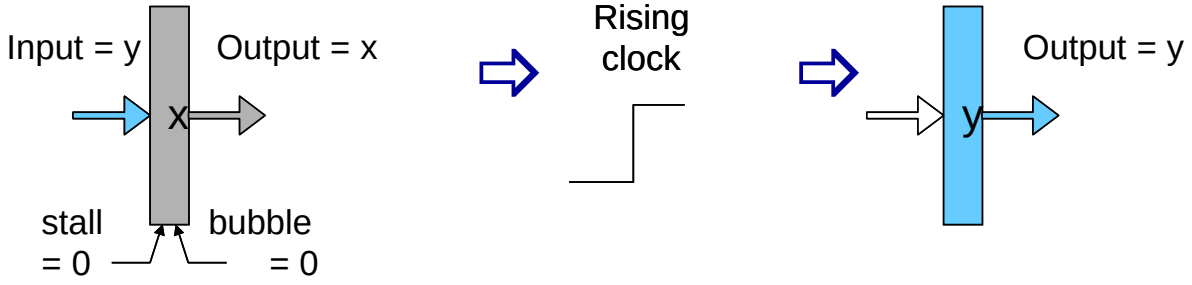


流水线控制 Pipeline Control

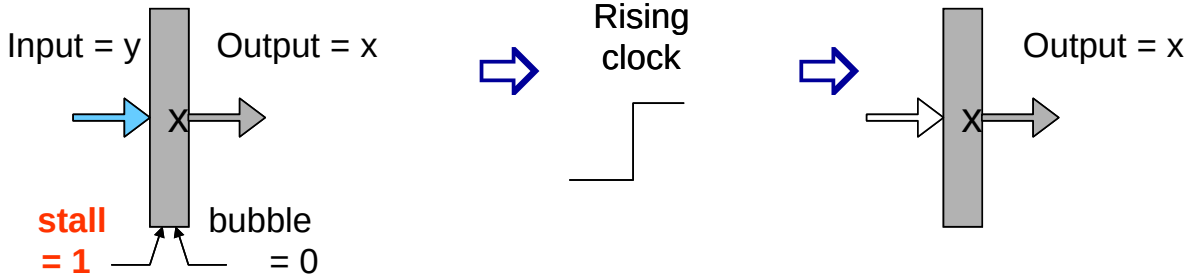
- 组合逻辑检测暂停条件 Combinational logic detects stall condition
- 设置模式信号指示流水线寄存器该如何更新 Sets mode signals for how pipeline registers should update

流水线寄存器模式
Pipeline Register Modes

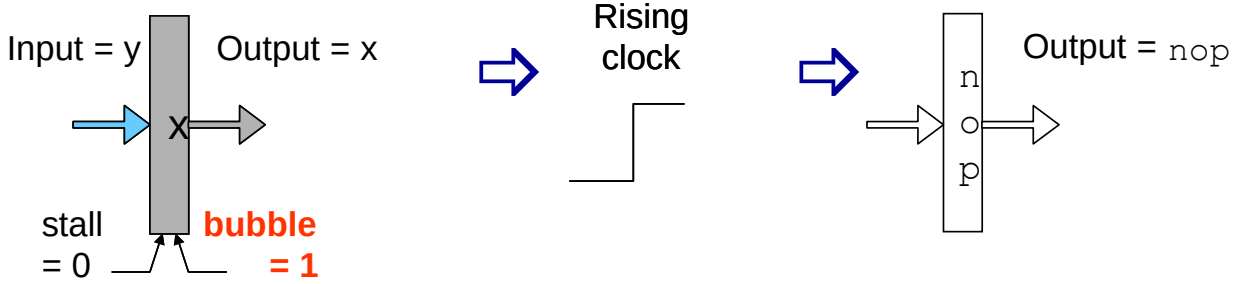
正常 Normal



暂停 Stall



气泡 Bubble



数据转发 Data Forwarding

朴素流水线 Naïve Pipeline

- 直到写回阶段完成才进行寄存器写操作 Register isn't written until completion of write-back stage
- 译码阶段就需要从寄存器文件(堆)读源操作数 Source operands read from register file in decode stage
  - 需要在阶段开始就在寄存器文件(堆)中才行 Needs to be in register file at start of stage

观察 Observation

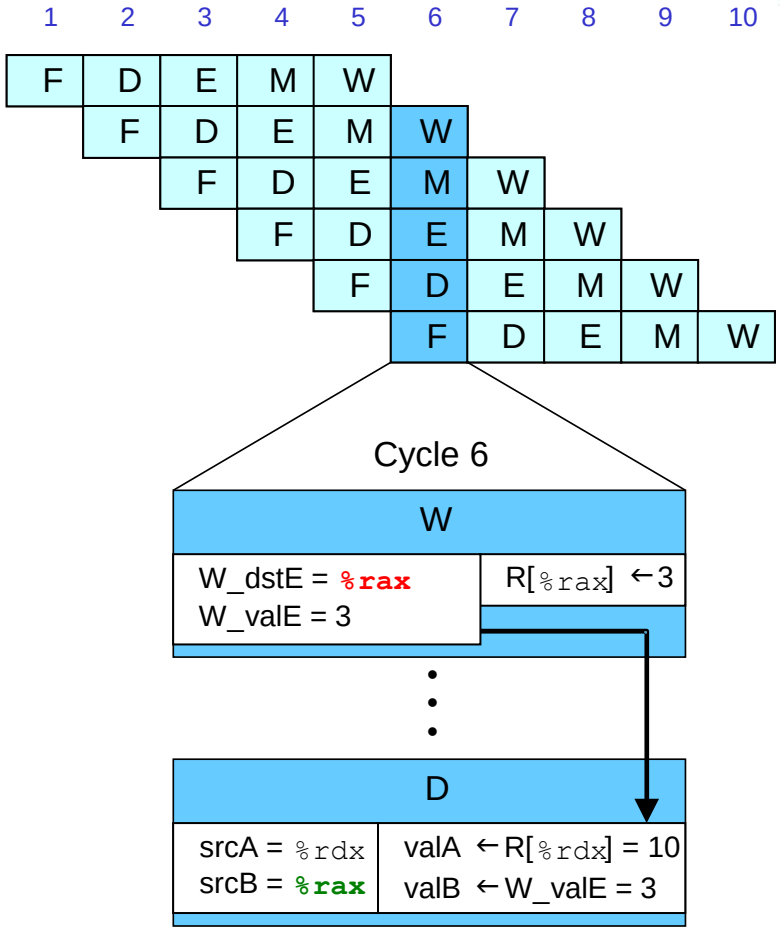
- 在执行或内存阶段产生值 Value generated in execute or memory stage

技巧 Trick

- 从生成指令处直接传递值到译码阶段 Pass value directly from generating instruction to decode stage
- 需要在译码阶段结束时可用 Needs to be available at end of decode stage

数据转发示例 Data Forwarding Example

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

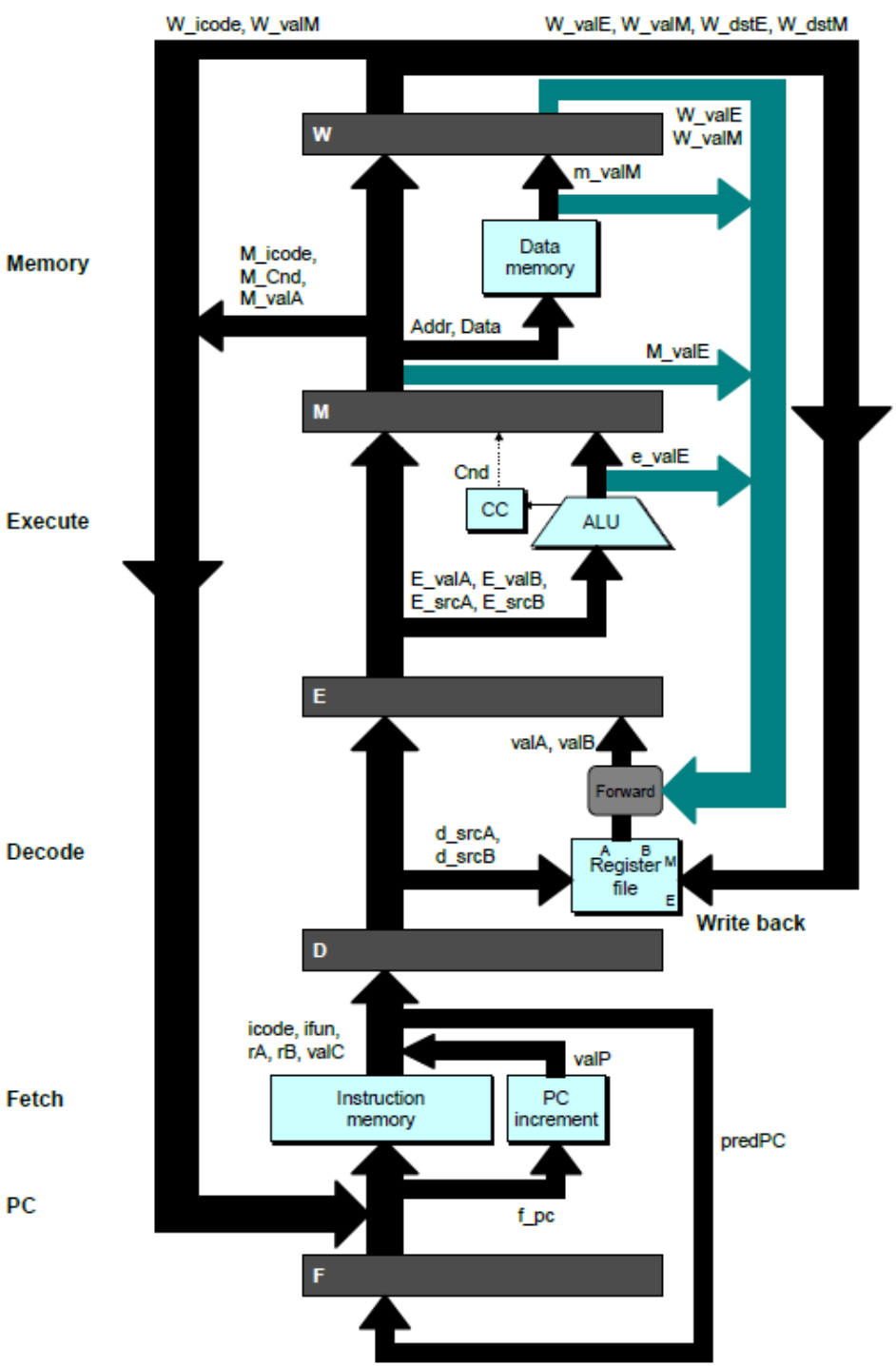


- irmovq在写回阶段 `irmovq` in write-back stage
- 目的值在W流水线寄存器中 Destination value in W pipeline register
- 转发为译码阶段的valB Forward as valB for decode stage

旁路路径 Bypass Paths

译码阶段 Decode Stage

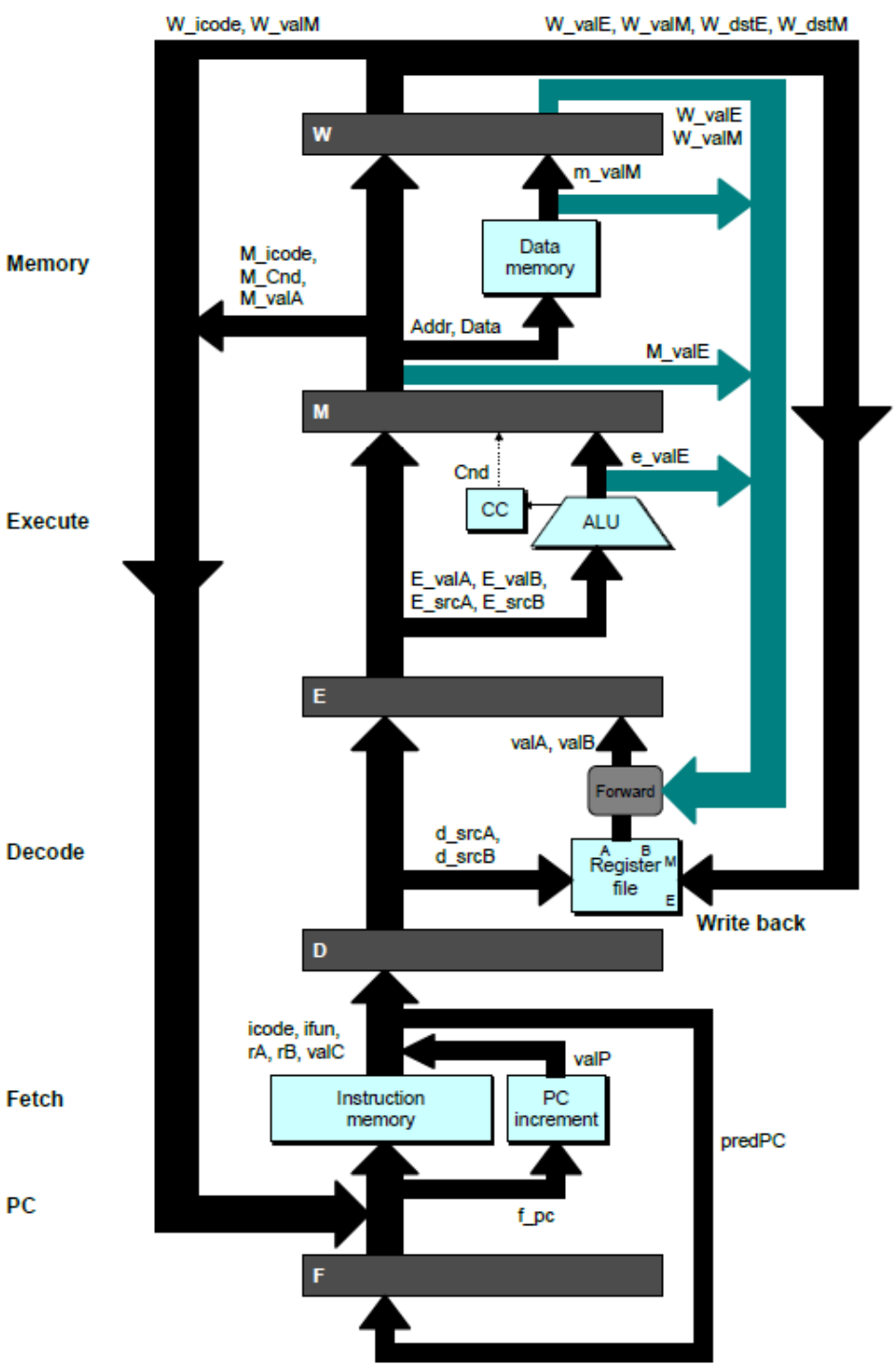
- 转发逻辑选择valA和valB
  Forwarding logic selects valA and valB
- 正常来自寄存器文件(堆)
  Normally from register file
- 转发：从后面流水线阶段得到valA或valB
  Forwarding: get valA or valB from later pipeline stage



旁路路径 Bypass Paths

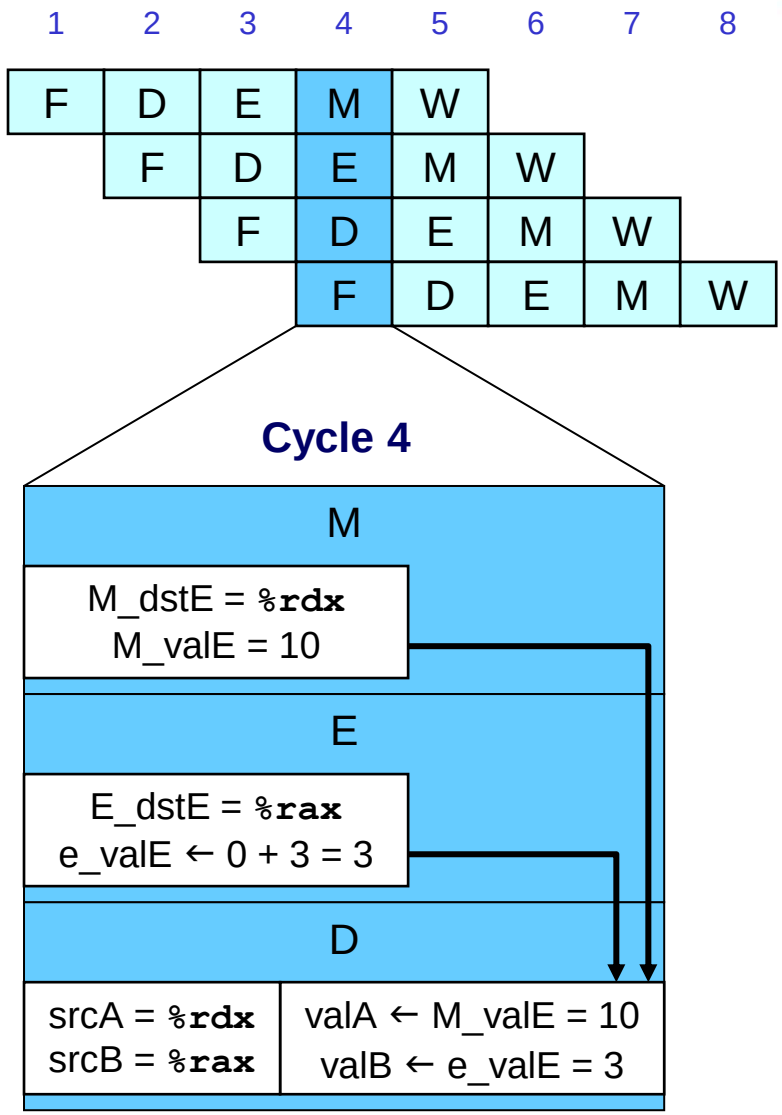
转发源 Forwarding Sources

- 执行阶段：valE Execute: valE
- 内存阶段：valE和valM Memory: valE, valM
- 写回阶段：valE和valM Write back: valE, valM



数据转发示例 Data Forwarding Example#2

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



寄存器%rdx Register `%rdx`

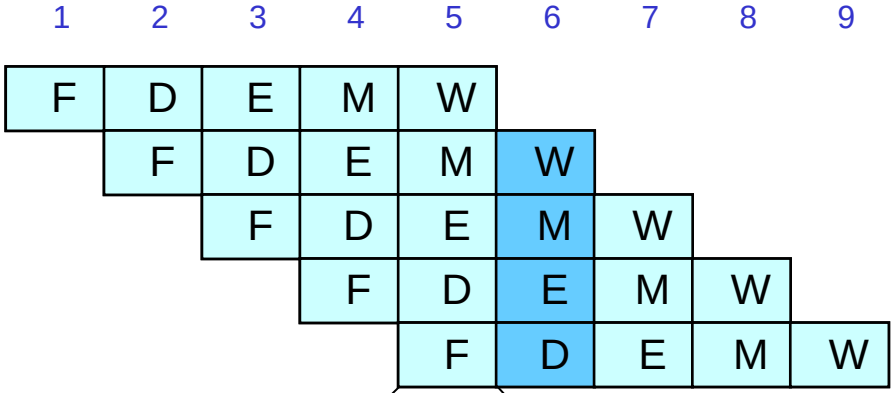
- 上个周期ALU产生
  Generated by ALU during previous cycle
- 从内存阶段转发作valA
  Forward from memory as valA

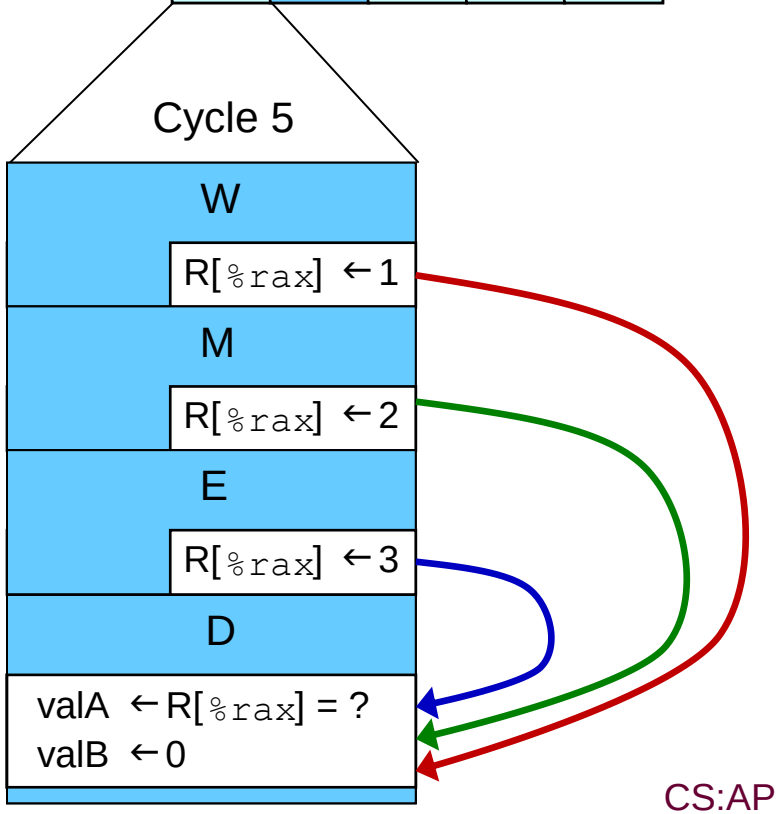
寄存器%rax Register `%rax`

- 刚由ALU产生的值 Value just generated by ALU
- 从执行阶段转发作valB
  Forward from execute as valB

转发优先级 Forwarding Priority

```
# demo-priority.ys
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```

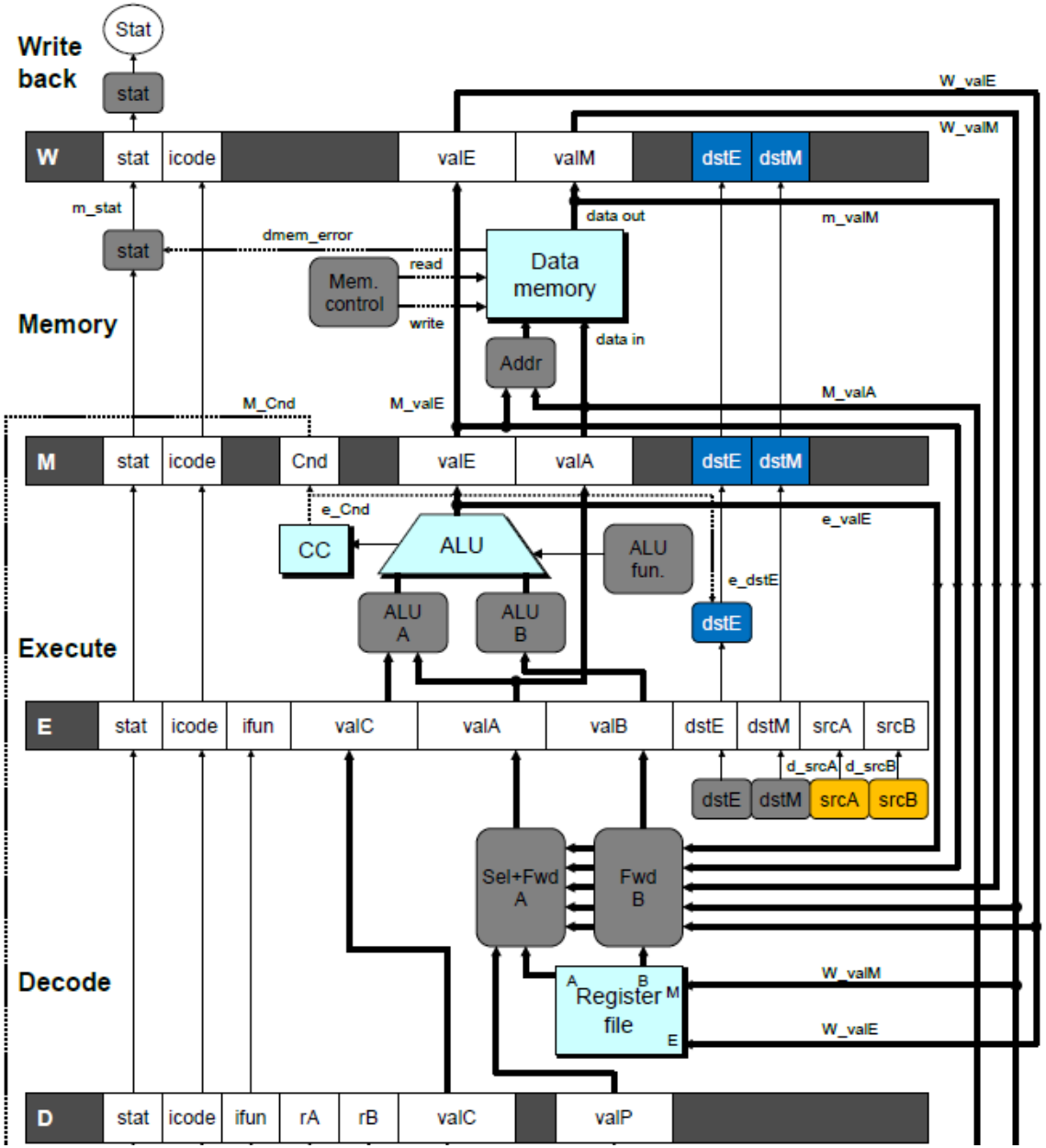




多种转发选择 Multiple Forwarding Choices

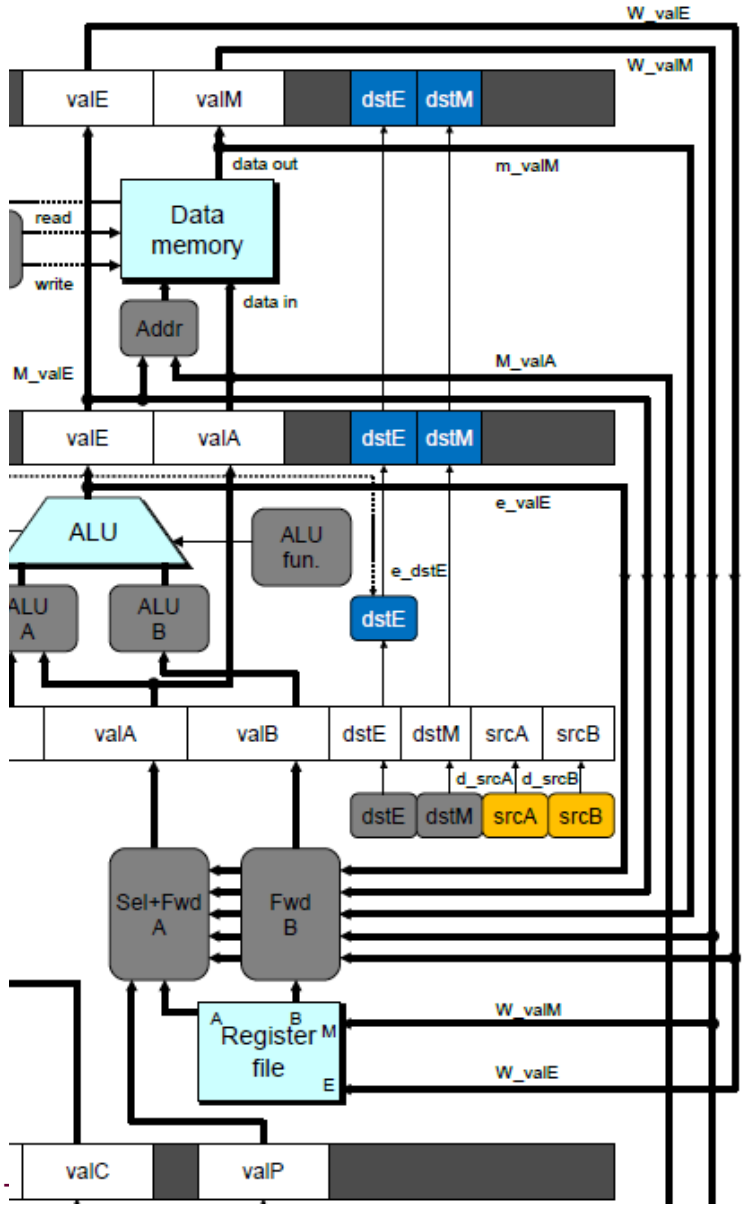
- 哪个应该有优先权 Which one should have priority
- 串行匹配语义 Match serial semantics
- 使用来自最早流水线阶段的匹配值 Use matching value from earliest pipeline stage

实现转发 Implementing Forwarding



- 增加附加的反馈路径，从E、M和W流水线寄存器转发到译码阶段 Add additional feedback paths from E, M, and W pipeline registers into decode stage
- 创建逻辑块从多个源选择作译码阶段的valA和valB Create logic blocks to select from multiple sources for valA and valB in decode stage

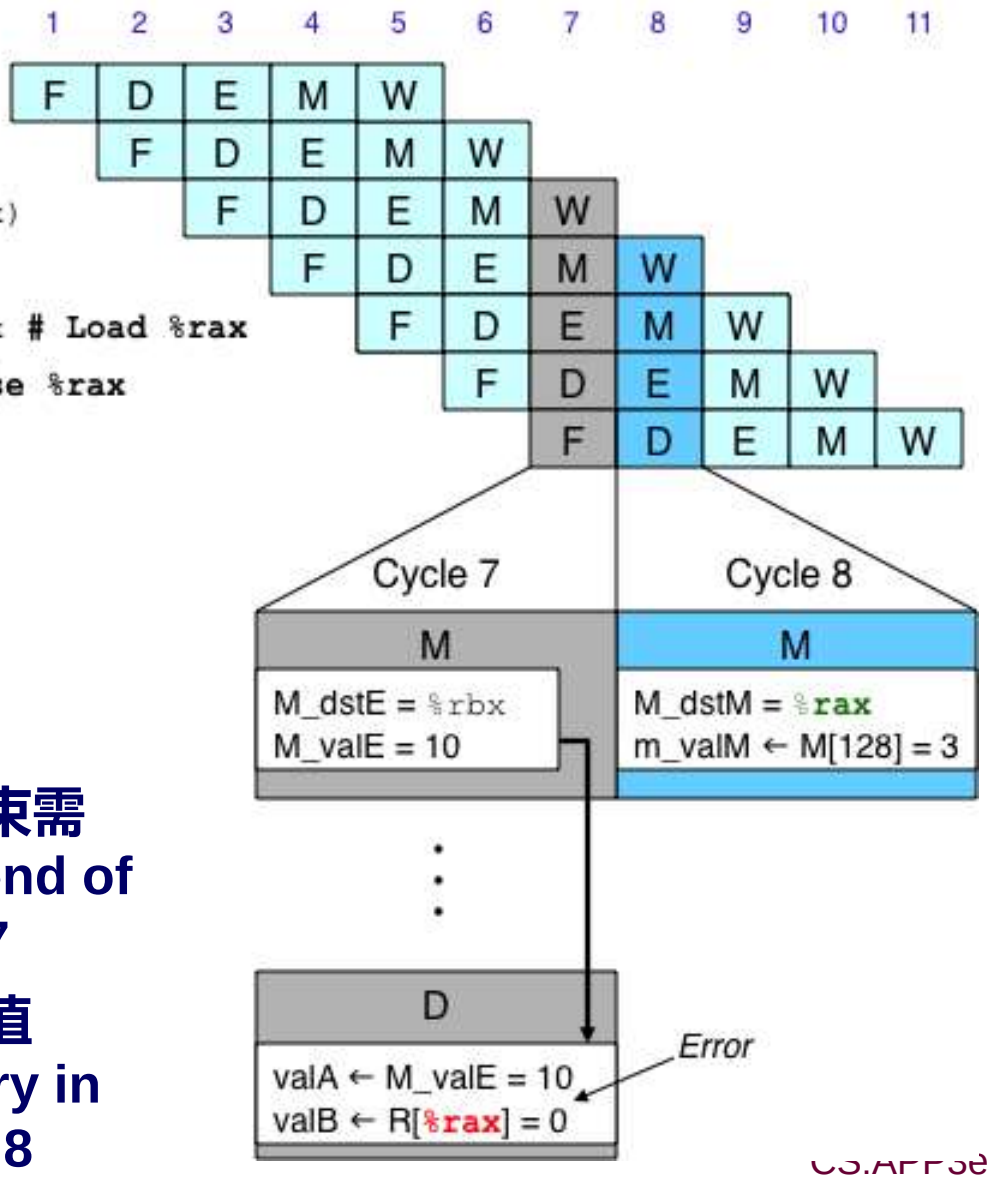
实现转发 Implementing Forwarding




```
## What should be the A value?
int d_valA = [
  # Use incremented PC
     D_icode in { ICALL, IJXX } : D_valP;
  # Forward valE from execute
     d_srcA == e_dstE : e_valE;
  # Forward valM from memory
     d_srcA == M_dstM : m_valM;
  # Forward valE from memory
     d_srcA == M_dstE : M_valE;
  # Forward valM from write back
     d_srcA == W_dstM : W_valM;
  # Forward valE from write back
     d_srcA == W_dstE : W_valE;
  # Use value read from register file
     1 : d_rvalA;
];
```

转发的限制 Limitation of Forwarding

```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```

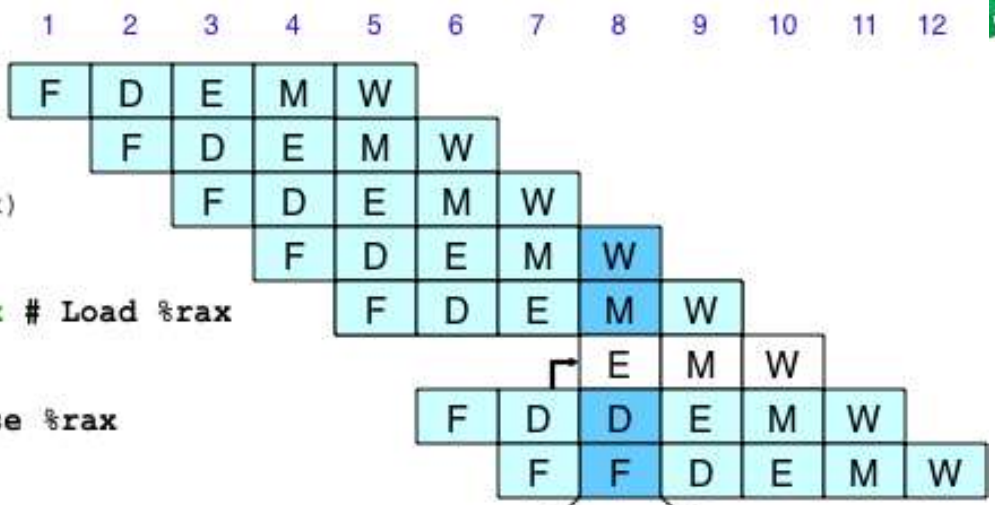


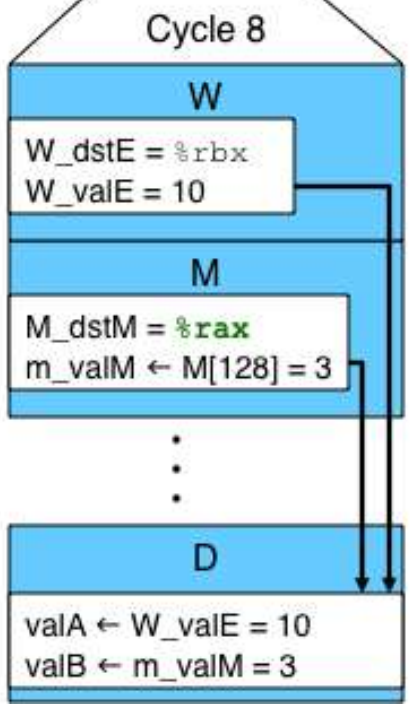
装载-使用相关 Load-use dependency

- 第7个周期译码阶段的结束需要值 Value needed by end of decode stage in cycle 7
- 第8个周期内存阶段读取值 Value read from memory in memory stage of cycle 8

避免装载/使用冒险 Avoiding Load/Use Hazard

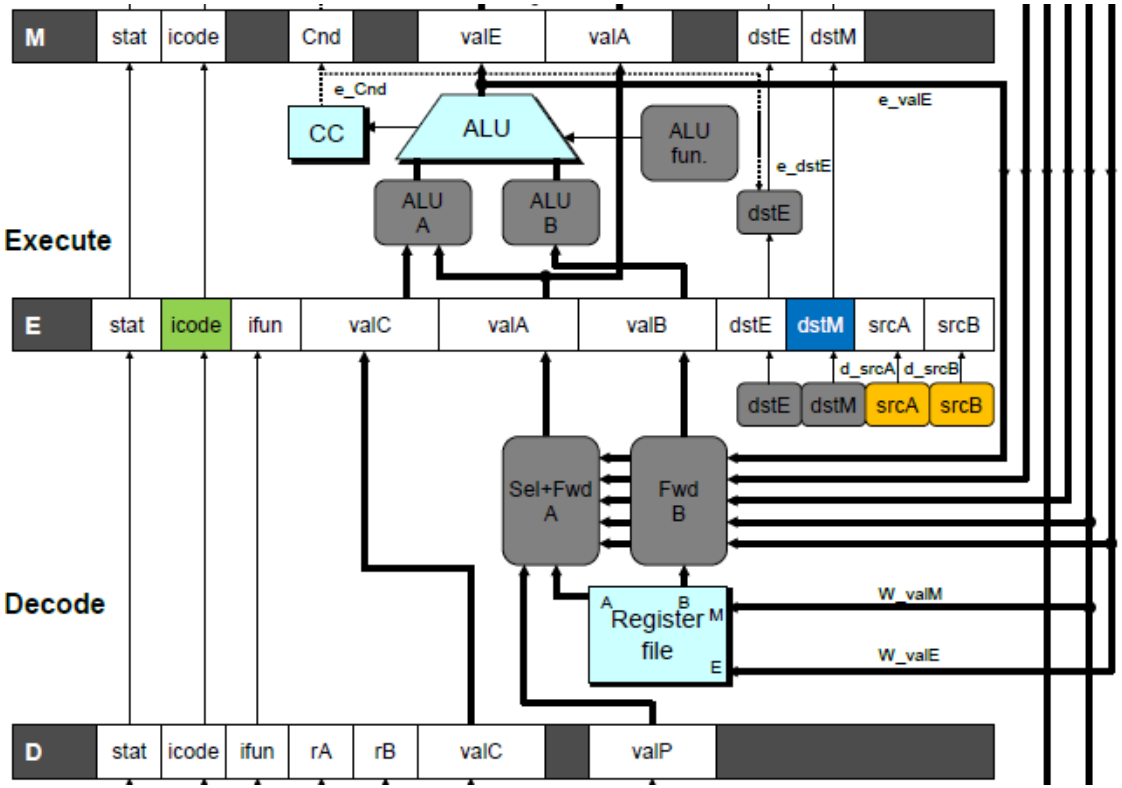
```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
     bubble
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```





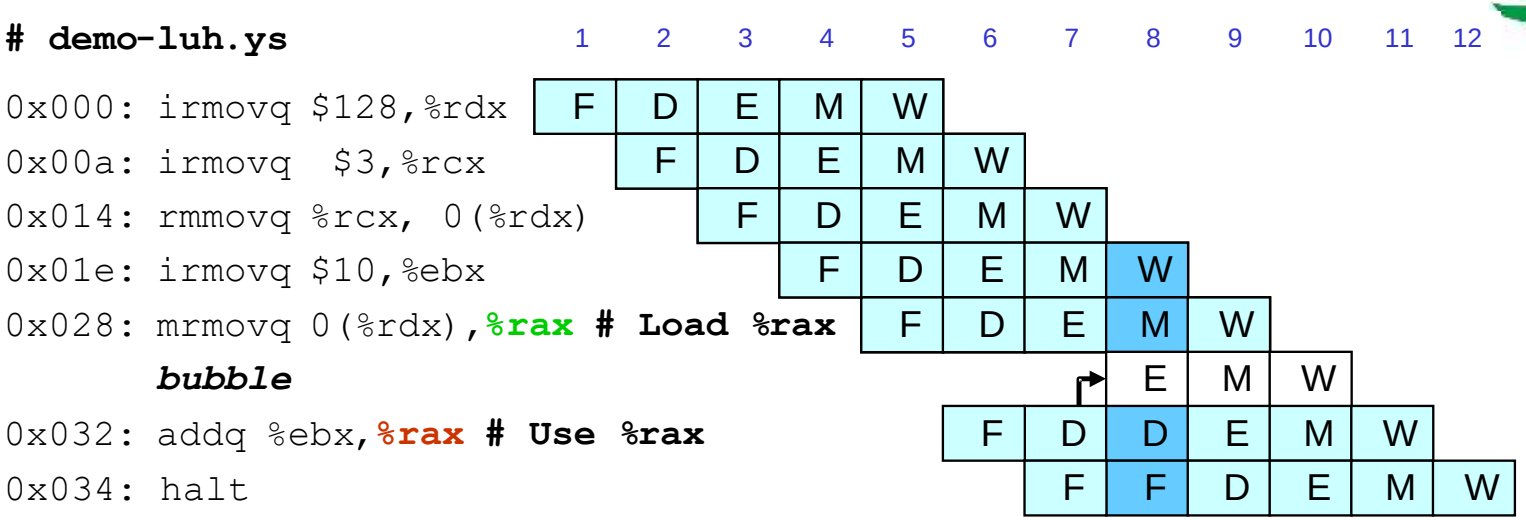
- 暂停使用指令一个周期 Stall using instruction for one cycle
- 然后能够通过从内存阶段转发获取装载的值 Can then pick up loaded value by forwarding from memory stage

检测装载/使用冒险 Detecting Load/Use Hazard



状况 Condition	触发 Trigger
装载/使用冒险 Load/Use Hazard	`E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }`

装载/使用冒险控制 Control for Load/Use Hazard



- 暂停指令在取指和译码阶段 Stall instructions in fetch and decode stages
- 注入气泡到执行阶段 Inject bubble into execute stage

状况 Condition	F	D	E	M	W
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal

分支预测错误示例
Branch Misprediction Example

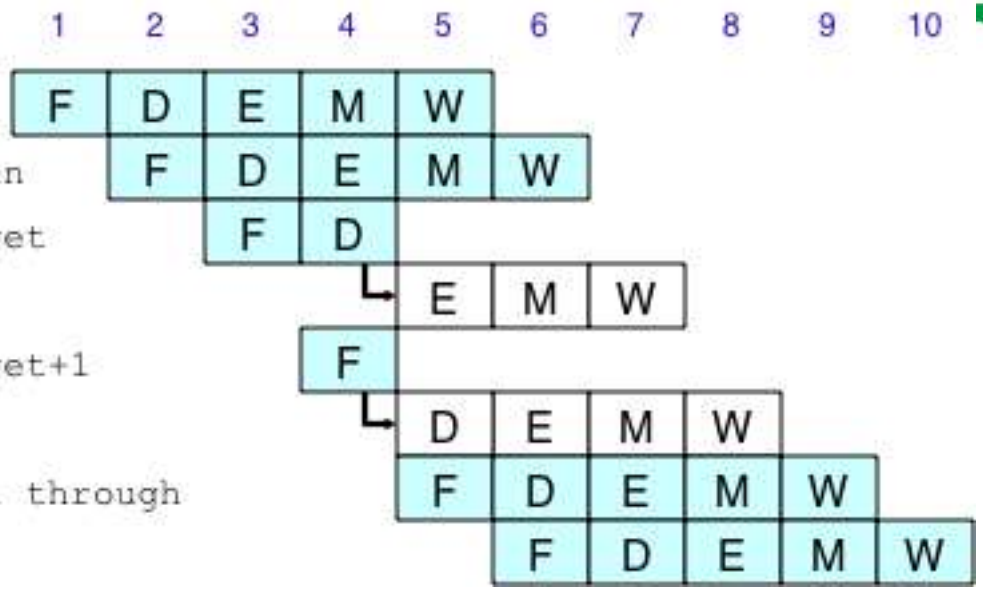
demo-j.ys

```
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
0x023:     irmovq $4, %rcx       # Should not execute
0x02d:     irmovq $5, %rdx       # Should not execute
```

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

处理预测错误 Handling Misprediction

```
# demo-j.ys
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
       bubble
0x020: irmovq $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



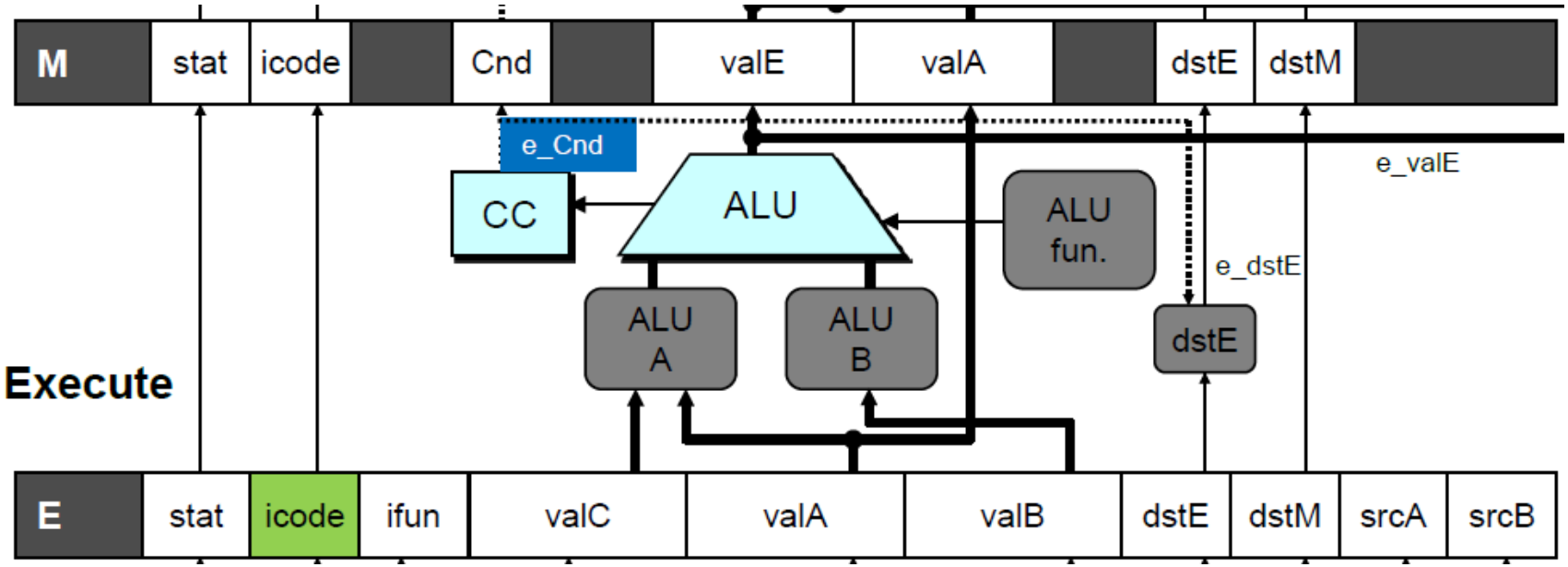
预测分支为选择分支 Predict branch as taken

- 取目标处2条指令 Fetch 2 instructions at target

当预测错误时取消 Cancel when mispredicted

- 在执行阶段检测出分支不选择 Detect branch not-taken in execute stage
- 在下一个周期，用气泡替代执行和译码阶段的指令 On following cycle, replace instructions in execute and decode by bubbles
- 还没有副作用发生 No side effects have occurred yet

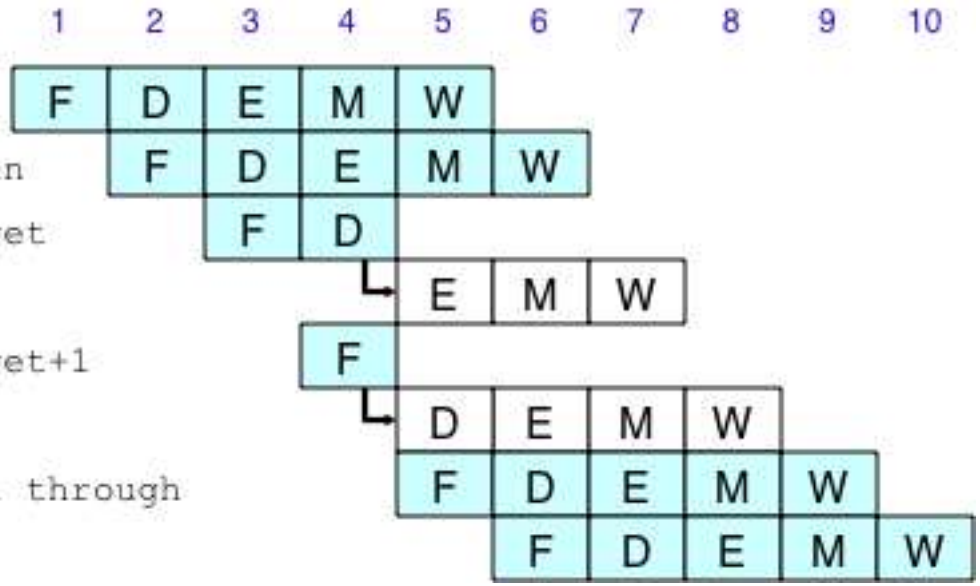
检测分支预测错误
Detecting Mispredicted Branch



状况 Condition	触发 Trigger
分支预测错误 Mispredicted Branch	`E_icode = IJXX & !e_Cnd`

预测错误控制 Control for Misprediction

```
# demo-j.ys
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
       bubble
0x020: irmovq $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



状况 Condition	F	D	E	M	W
分支预测错误 Mispredicted Branch	正常 normal	气泡 bubble	气泡 bubble	正常 normal	正常 normal

返回示例 Return Example

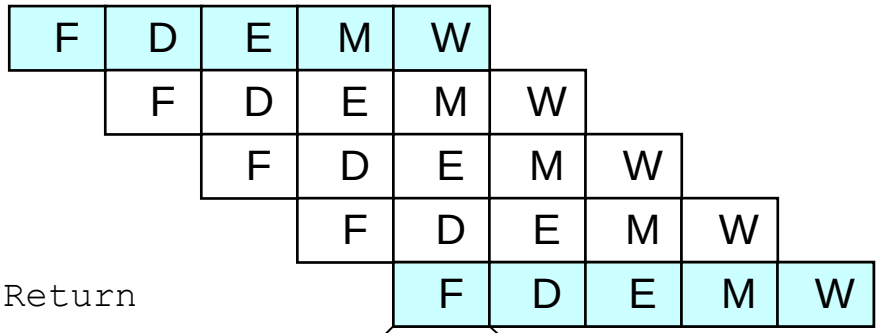
demo-retb.ys

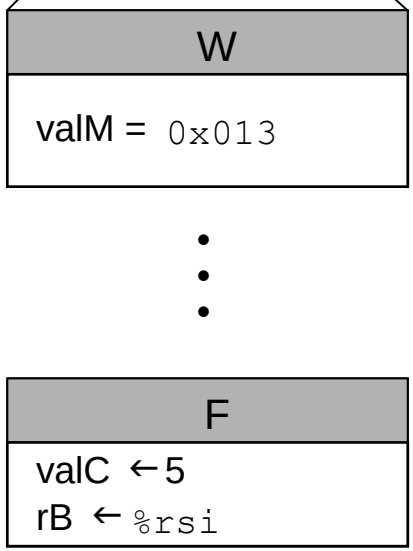
```
0x000:     irmovq Stack,%rsp  # Intialize stack pointer
0x00a:     call p             # Procedure call
0x013:     irmovq $5,%rsi     # Return point
0x01d:     halt
0x020:  .pos 0x20
0x020:  p: irmovq $-1,%rdi    # procedure
0x02a:     ret
0x02b:     irmovq $1,%rax     # Should not be executed
0x035:     irmovq $2,%rcx     # Should not be executed
0x03f:     irmovq $3,%rdx     # Should not be executed
0x049:     irmovq $4,%rbx     # Should not be executed
0x100:  .pos 0x100
0x100:  Stack:                # Stack: Stack pointer
```

- 以前执行三条附加的指令 Previously executed three additional instructions

正确返回示例 Correct Return Example

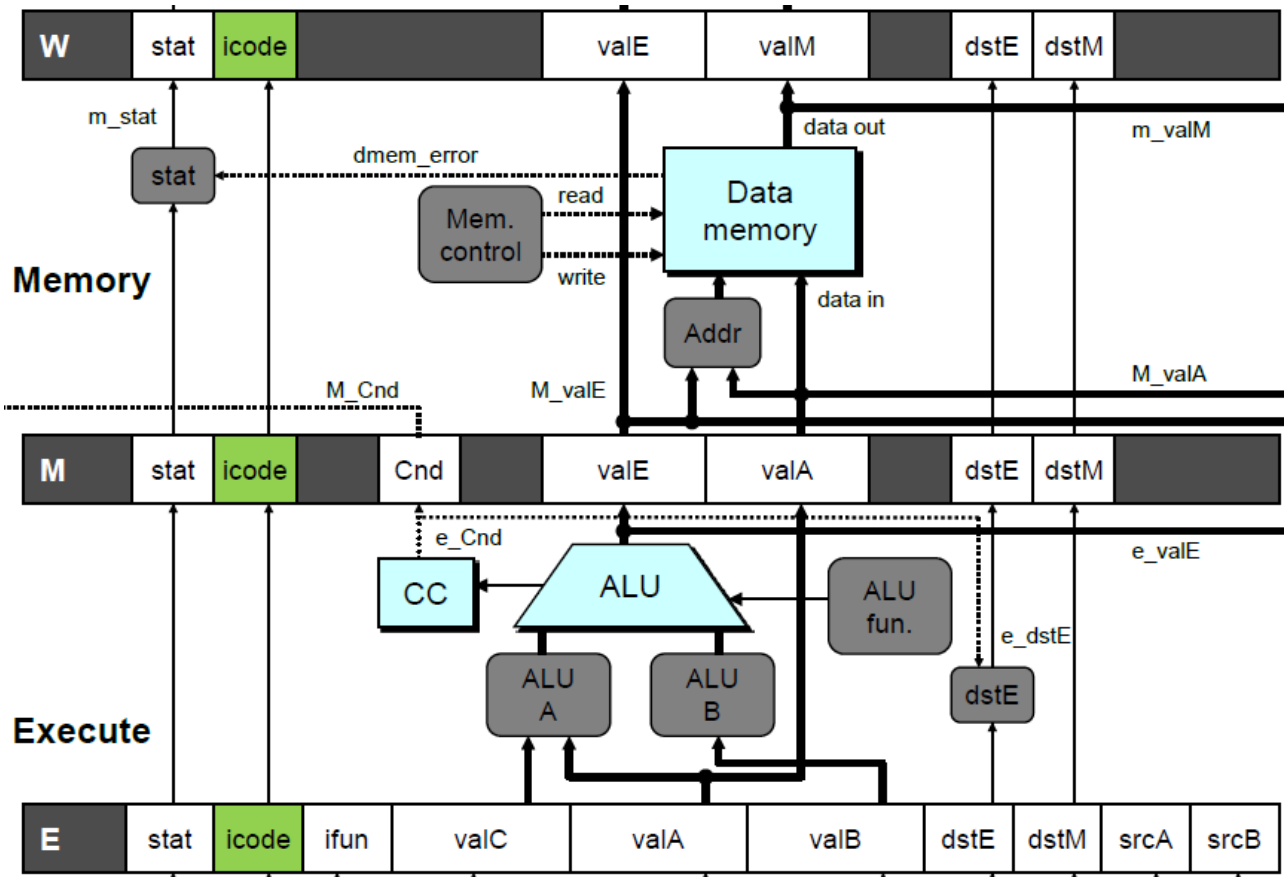
```
# demo-retb
0x026:      ret
            bubble
            bubble
            bubble
0x013:      irmovq $5,%rsi # Return
```





- 当ret通过流水线时，暂停取指阶段 As `ret` passes through pipeline, stall at fetch stage
  - 同时ret流过译码、执行和内存阶段 While in decode, execute, and memory stage
- 注入气泡到译码阶段 Inject bubble into decode stage
- 当到达写回阶段时释放暂停 Release stall when reach write-back stage

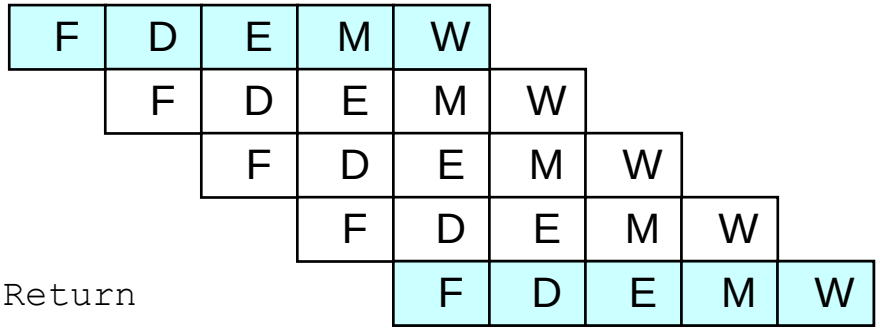
检测返回 Detecting Return



状况 Condition	触发 Trigger
处理ret Processing `ret`	`IRET in { D_icode, E_icode, M_icode }`

返回控制 Control for Return

```
# demo-retb
0x026:      ret
            bubble
            bubble
            bubble
0x014:      irmovq $5,%rsi # Return
```



状况 Condition	F	D	E	M	W
处理ret Processing `ret`	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal

特殊控制情况 Special Control Cases

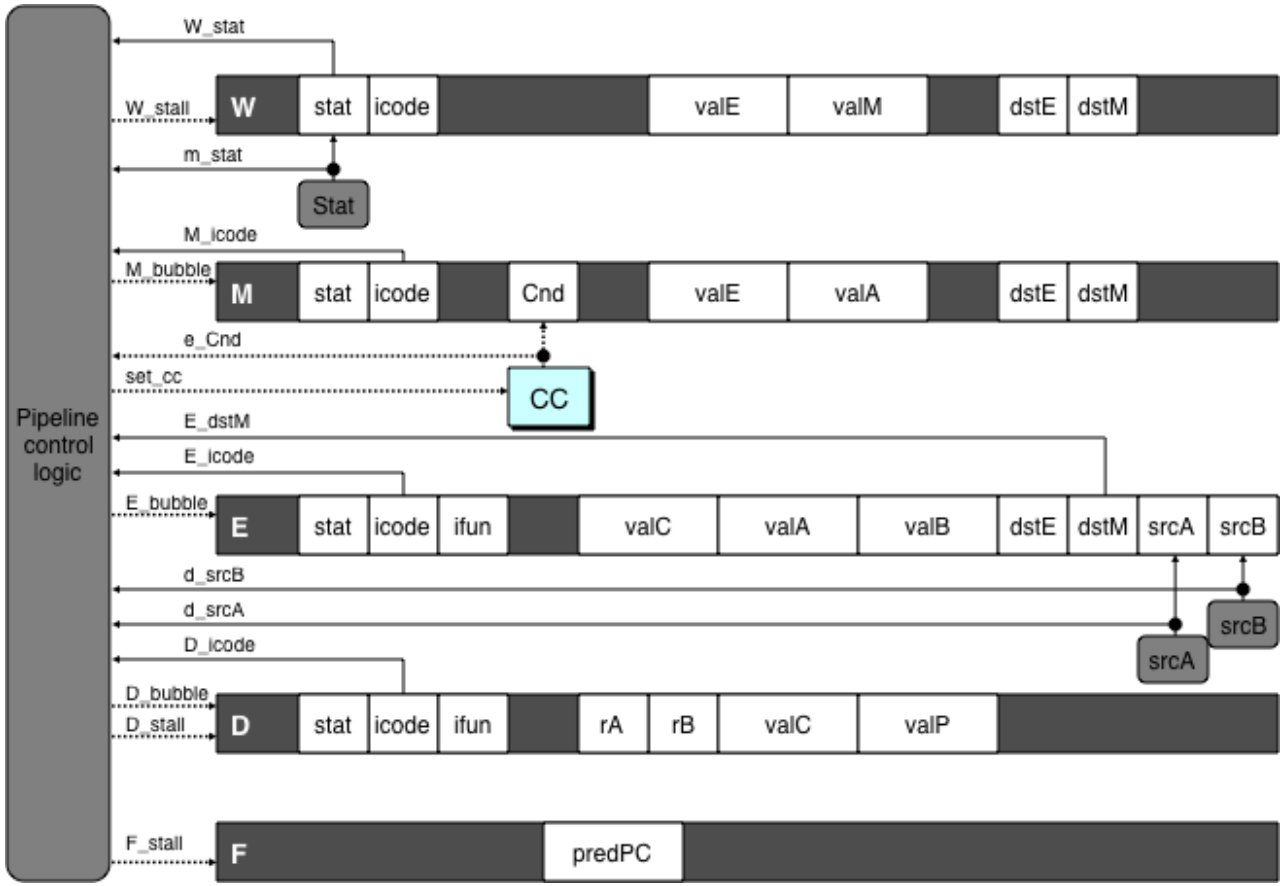
检测 Detection

状况 Condition	触发 Trigger
处理ret Processing `ret`	IRET in { D_icode, E_icode, M_icode }
装载/使用冒险 Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
分支预测错误 Mispredicted Branch	E_icode = IJXX & !e_Cnd

动作（下一个周期） Action (on next cycle)

Condition	F	D	E	M	W
处理ret Processing `ret`	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal
分支预测错误 Mispredicted Branch	正常 normal	气泡 bubble	气泡 bubble	正常 normal	正常 normal

实现流水线控制
Implementing Pipeline Control



- 组合逻辑产生流水线控制信号 Combinational logic generates pipeline control signals
- 动作发生在下一个周期的开始 Action occurs at start of following cycle

流水线控制的初始版本
Initial Version of Pipeline Control

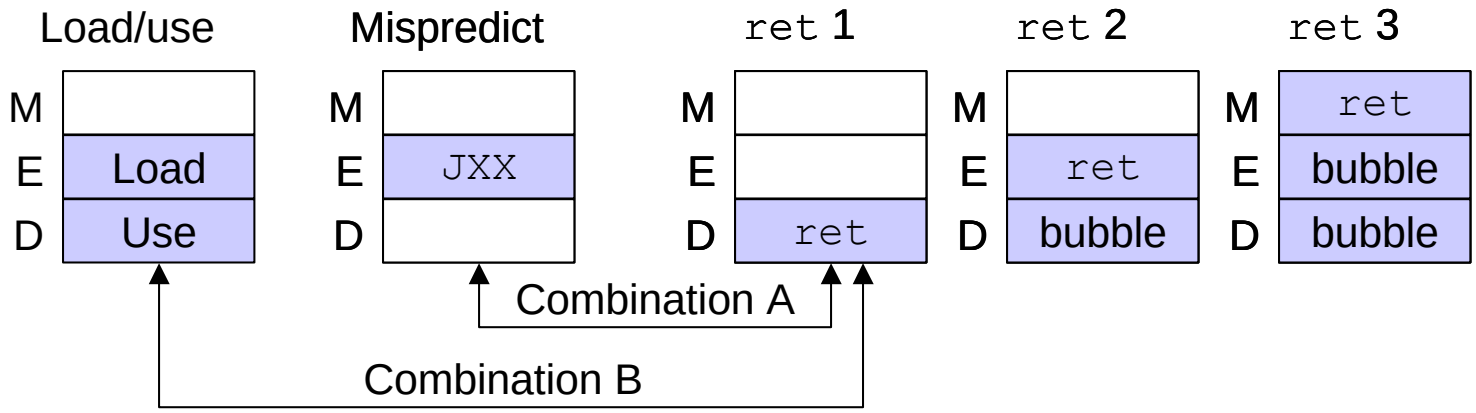
```
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Stalling at fetch while ret passes through pipeline
     IRET in { D_icode, E_icode, M_icode };

bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };
```

控制组合 Control Combinations



- 在同一个时钟周期可能引起的特殊情况 Special cases that can arise on same clock cycle

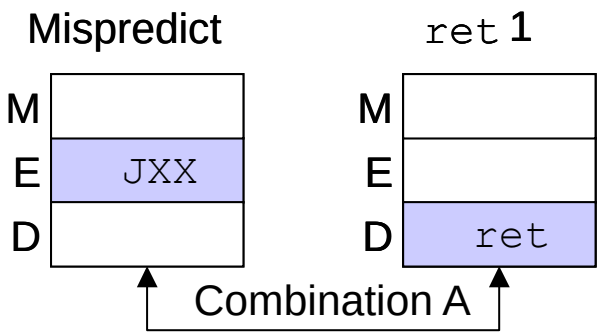
组合A Combination A

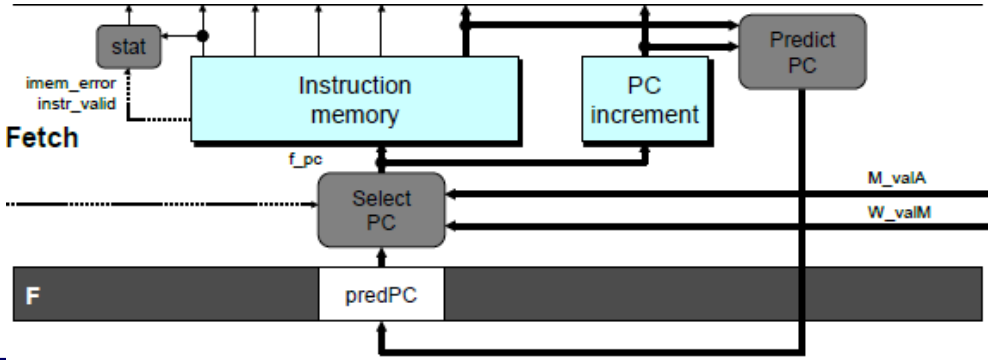
- 分支不选择 Not-taken branch
- Ret指令在分支目标处 `ret` instruction at branch target

组合B Combination B

- 指令从内存读到%rsp Instruction that reads from memory to `%rsp`
- 后跟ret指令 Followed by `ret` instruction

控制组合A Control Combination A

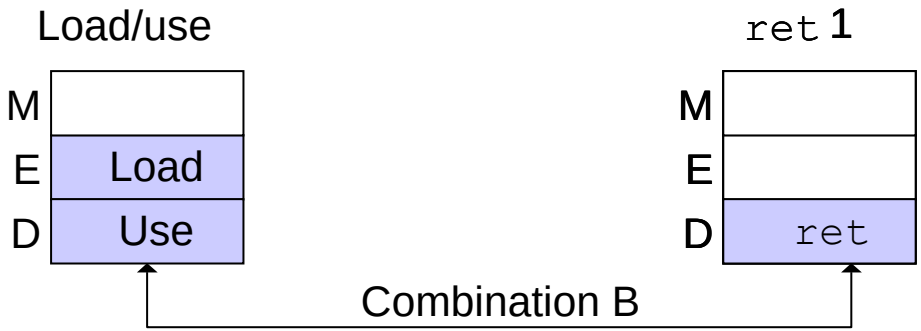




状况 Condition	F	D	E	M	W
处理ret Processing ret	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
分支预测错误 Mispredicted Branch	正常 normal	气泡 bubble	气泡 bubble	正常 normal	正常 normal
组合 Combination	*暂停* *stall*	*气泡* *bubble*	*气泡* *bubble*	*正常* *normal*	*正常* *normal*

- 当分支预测错误时应该处理 Should handle as mispredicted branch
- 暂停F流水线寄存器 Stalls F pipeline register
- 但PC选择逻辑无论如何使用M_valM But PC selection logic will be using M_valM anyhow

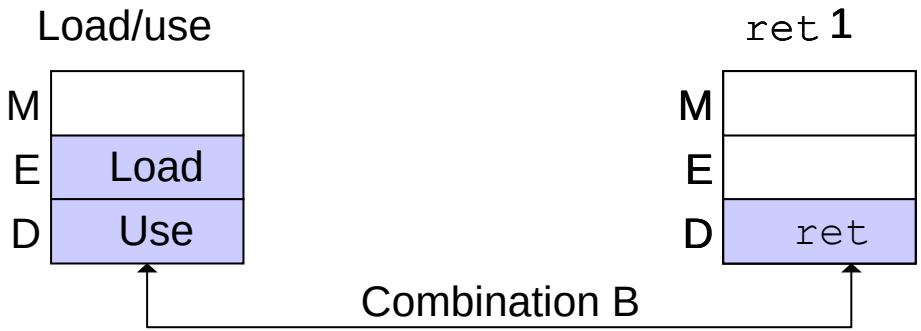
控制组合B Control Combination B



状况 Condition	F	D	E	M	W
处理ret Processing ret	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal
组合Combination	*暂停* *stall*	*气泡+暂停* *bubble + stall*	*气泡* *bubble*	*正常* *normal*	*正常* *normal*

- 尝试注入气泡和暂停流水线寄存器D Would attempt to bubble *and* stall pipeline register D
- 处理器信号指示流水线错误 Signaled by processor as pipeline error

处理控制组合B Handling Control Combination B



状况 Condition	F	D	E	M	W
处理ret Processing ret	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal
组合 Combination	*暂停* *stall*	*暂停* *stall*	*气泡* *bubble*	*正常* *normal*	*正常* *normal*

- 装载/使用冒险应该得到优先权 Load/use hazard should get priority
- ret指令保持在译码阶段一段附加周期 `ret` instruction should be held in decode stage for additional cycle

修正流水线控制逻辑
Corrected Pipeline Control Logic

```
bool D_bubble =
     # Mispredicted branch
     (E_icode == IJXX && !e_Cnd) ||
     # Stalling at fetch while ret passes through pipeline
      IRET in { D_icode, E_icode, M_icode }
       # but not condition for a load/use hazard
       && !(E_icode in { IMRMOVQ, IPOPQ }
            && E_dstM in { d_srcA, d_srcB });
```

状况 Condition	F	D	E	M	W
处理ret Processing ret	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal
组合 Combination	*暂停* *stall*	*暂停* *stall*	*气泡* *bubble*	*正常* *normal*	*正常* *normal*

- 装载/使用冒险应该得到优先权 Load/use hazard should get priority
- ret指令保持在译码阶段一段附加周期 `ret` instruction should be held in decode stage for additional cycle

流水线小结 Pipeline Summary

数据冒险 Data Hazards

- 大多通过转发进行处理 Most handled by forwarding
  - 没有性能损失 No performance penalty
- 装载/使用冒险需要暂停一个周期 Load/use hazard requires one cycle stall

控制冒险 Control Hazards

- 当检测到分支预测错误时取消指令 Cancel instructions when detect mispredicted branch
  - 浪费两个时钟周期 Two clock cycles wasted
- 当ret通过流水线时暂停取指阶段 Stall fetch stage while `ret` passes through pipeline
  - 浪费三个时钟周期 Three clock cycles wasted

控制组合 Control Combinations

- 必须仔细分析 Must analyze carefully
- 第一个版本有细微的错误 First version had subtle bug
  - 仅在不寻常的指令组合中出现 Only arises with unusual instruction combination