第四章 处理器体系结构
——流水线的实现Part II

教 师： 郑贵滨
计算机科学与技术学院
哈尔滨工业大学

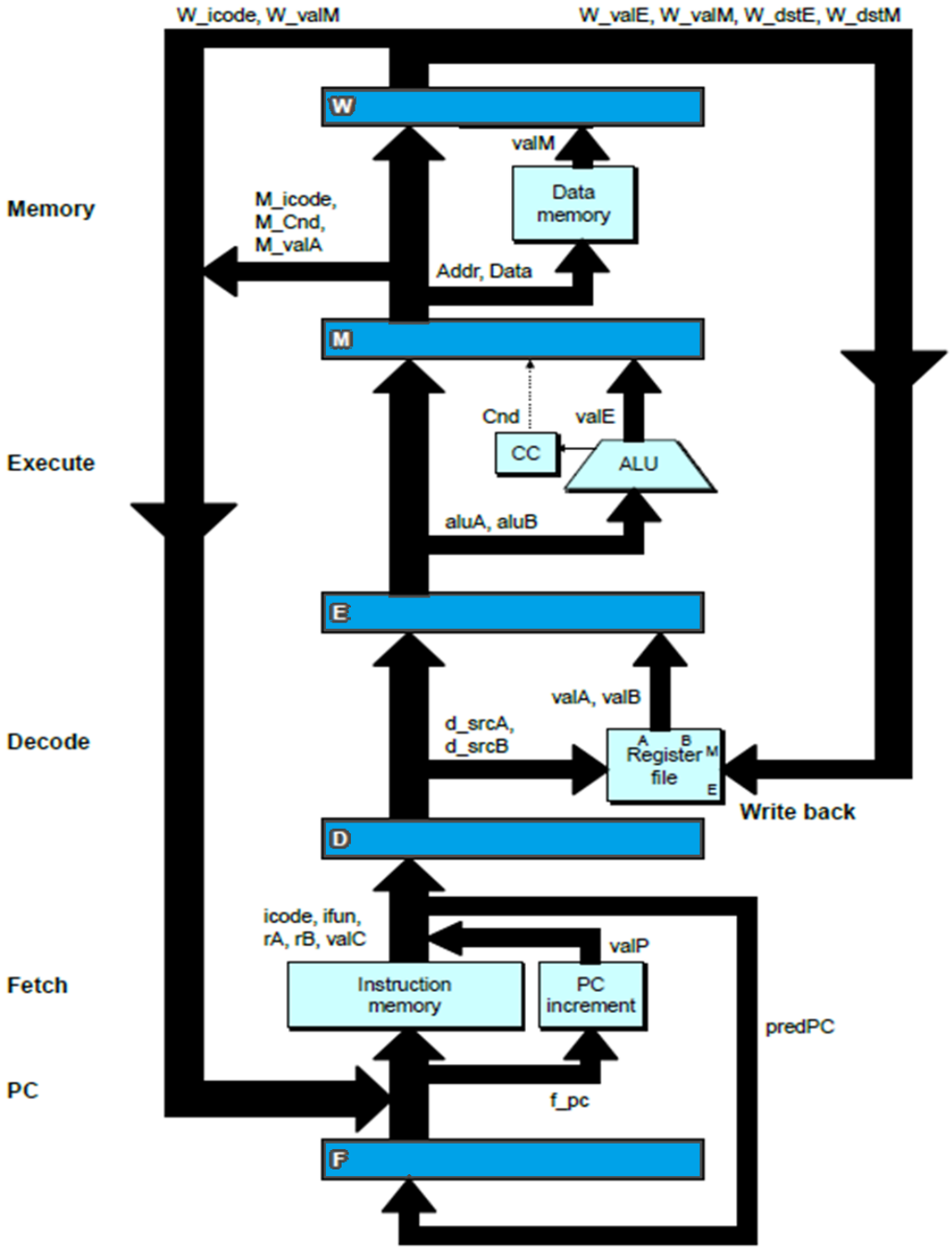
概述

使流水线处理器工作！

- **数据冒险**
  - 一条指令使用寄存器R作为目的操作数，随后很快，另一条指令将R用作源操作数
  - 一般情况，**不想降低流水线的速度**
- **控制冒险**
  - 条件分支预测错误
    - 我们的设计：预测所有分支均可能被选择
    - 朴素流水线执行两条额外的指令
  - 为ret指令获得返回地址
    - 朴素流水线执行三条额外的指令
- **确保它确实能有效地工作**
  - 如果多种特殊情况同时发生将会怎样?

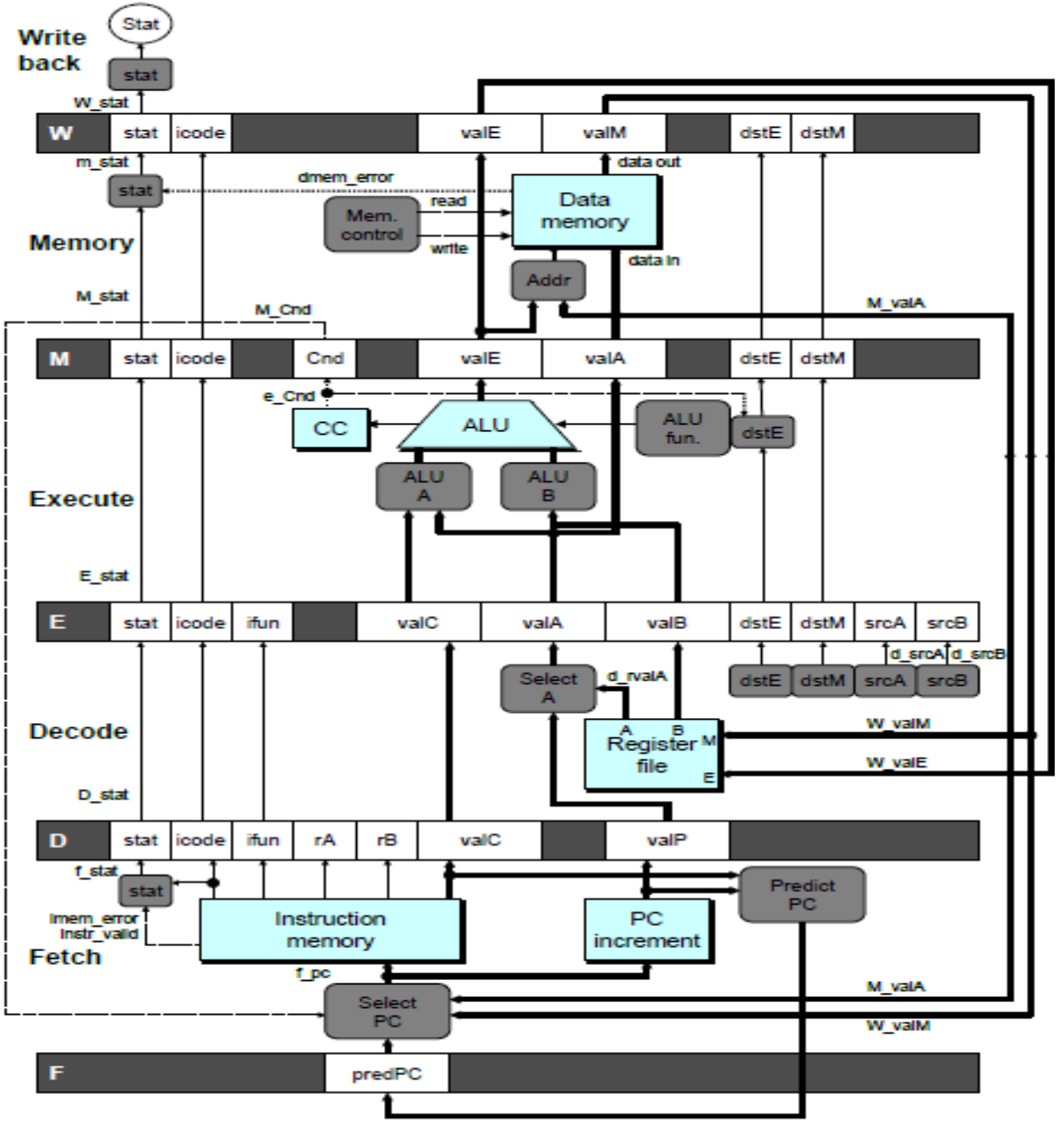
流水阶段

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



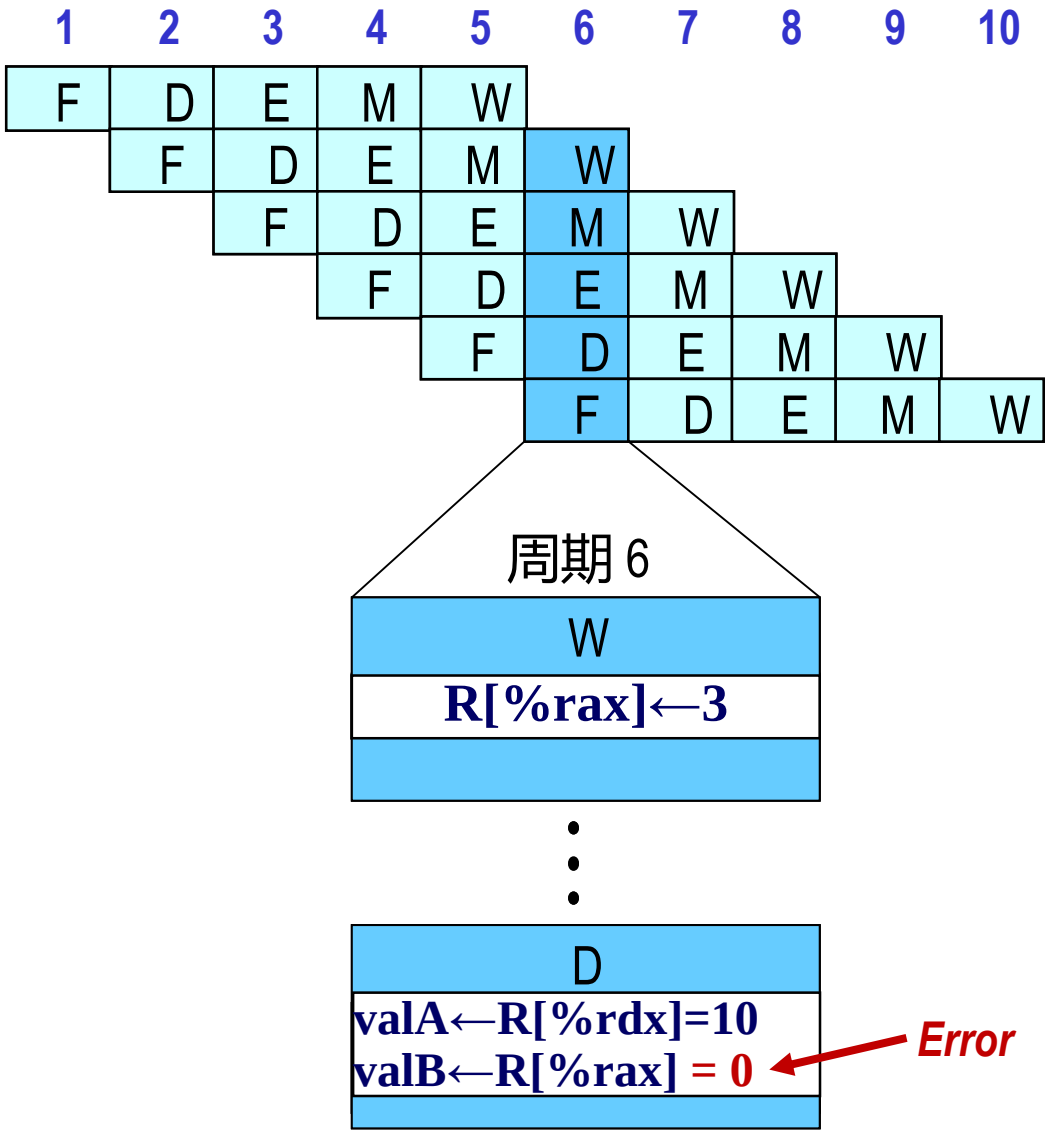
PIPE- 硬件

- 流水线寄存器保存指令执行过程的中间值
- 向上路径
  - 值从一个阶段向另一个阶段传递
  - 不能跳回到过去的阶段
    - e.g.，ValC已经通过译码阶段



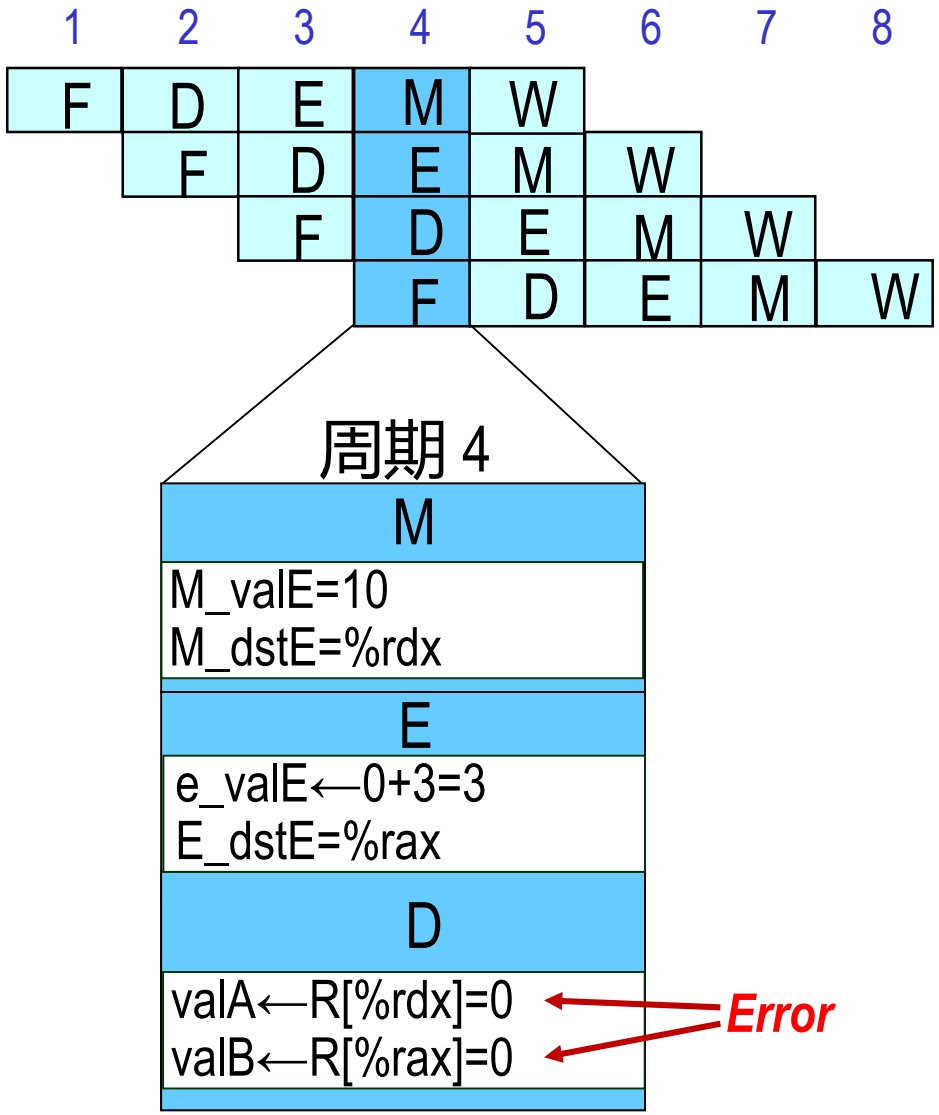
数据相关: 2条Nop指令

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



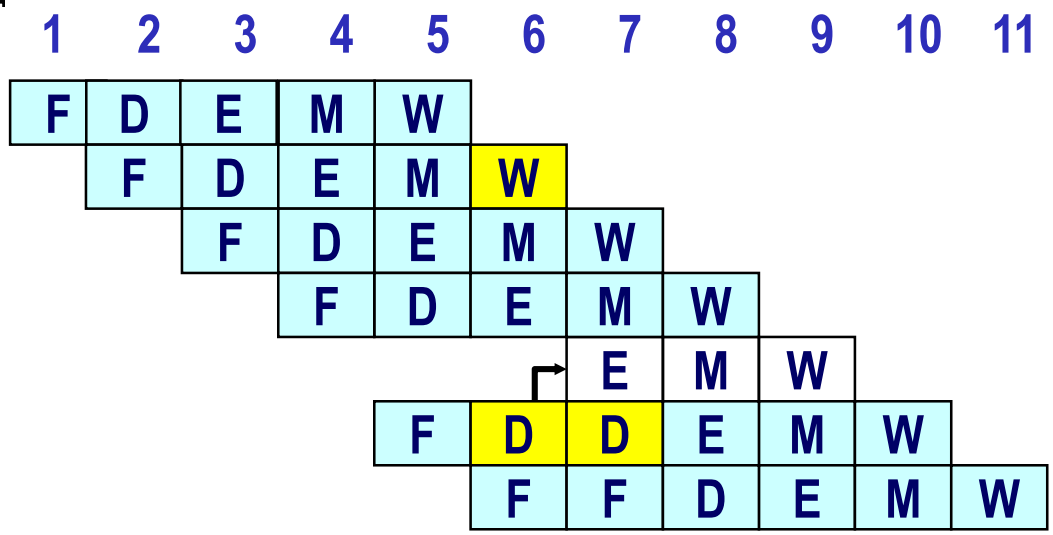
数据相关: 无Nop指令

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



用暂停避免数据冒险

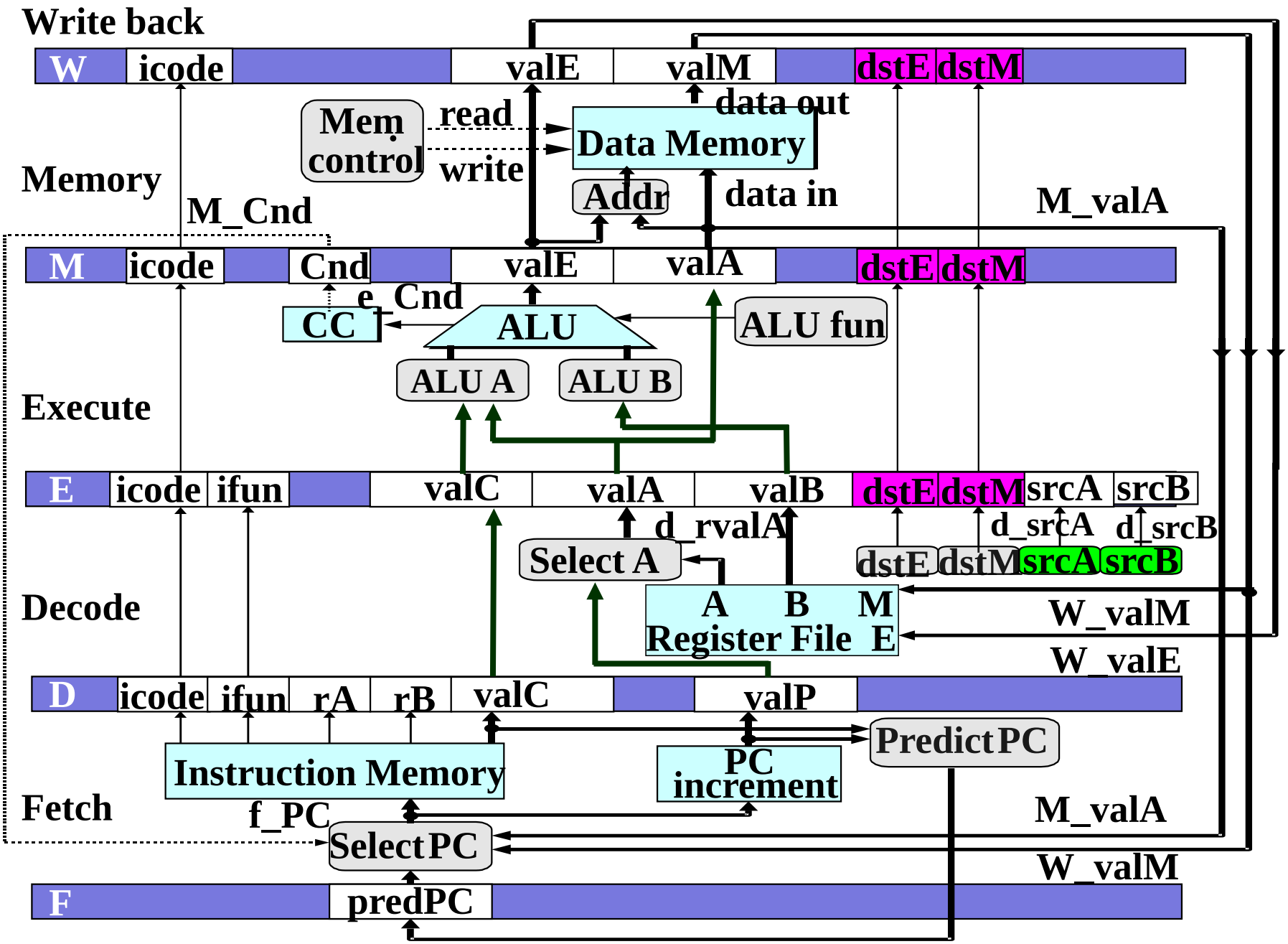
```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
          bubble
0x016: addq %rdx,%rax
0x018: halt
```



- 如果一条指令紧跟写寄存器的指令太近，则放慢该指令的执行
- 将指令阻塞在译码阶段
- 在指令执行阶段动态插入气泡（类似自动产生的nop）

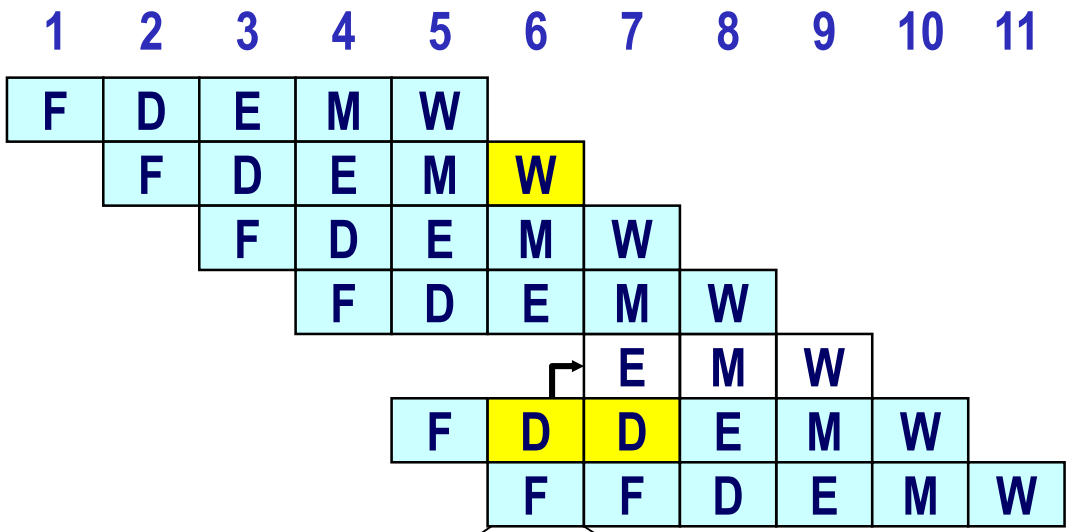
暂停条件

- **源寄存器**
  - 当前指令的srcA和srcB处于译码阶段
- **目的寄存器**
  - dstE 和dstM字段
  - 处于执行、访存和写回阶段的指令
- **特例**
  - 对于ID为15(0xF)的寄存器不需要暂停
    - 表示无寄存器操作数
    - 或表示失败的条件传送

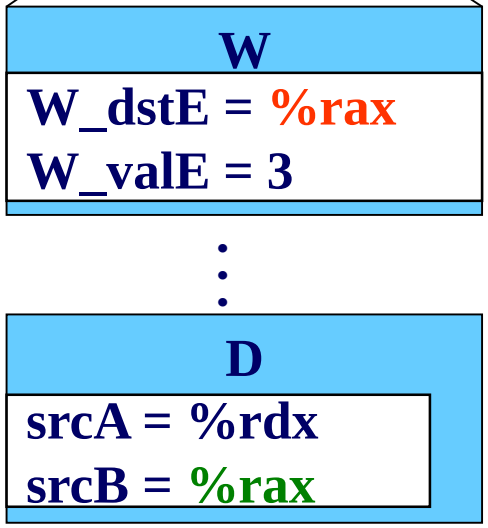

Write back
W
icode
valE
valM
dstE
dstM
Mem control
read
write
data out
Data Memory
Addr
data in
Memory
M_Cnd
M_valA
M
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun
ALU A
ALU B
Execute
E
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_rvalA
d_srcA
d_srcB
Select A
dstE
dstM
srcA
srcB
A B M
Register File E
Decode
W_valM
W_valE
D
icode
ifun
rA
rB
valC
valP
Predict PC
Instruction Memory
PC increment
Fetch
f_PC
M_valA
Select PC
W_valM
F
predPC

检测暂停条件

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



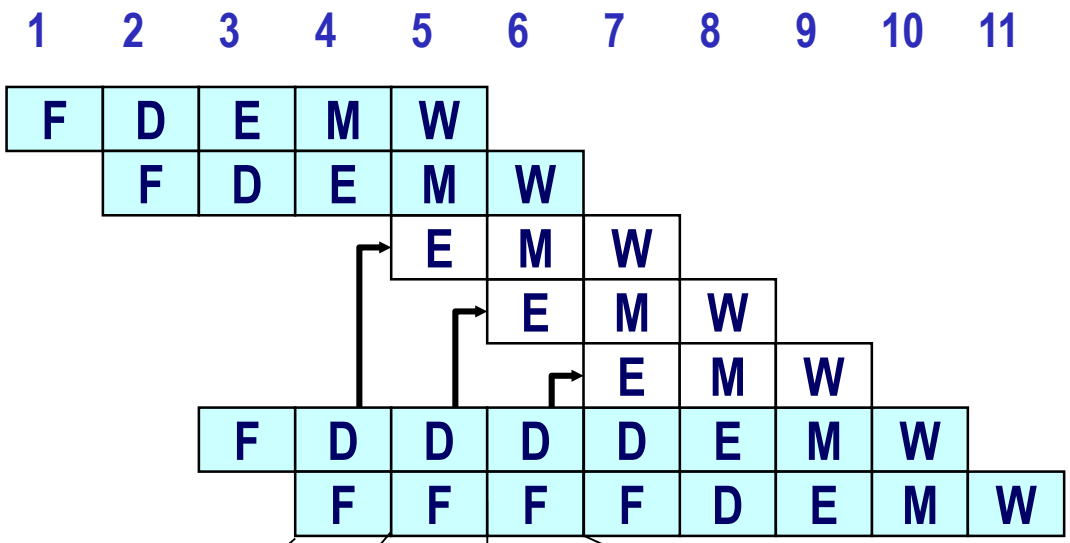
周期 6



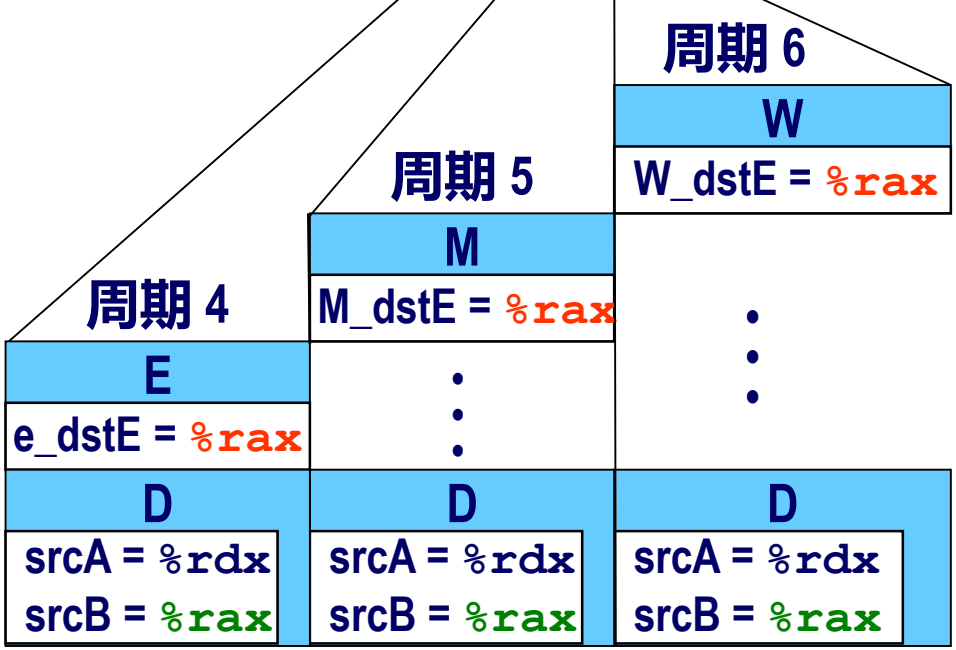

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

暂停 X3

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




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



暂停时发生了什么?

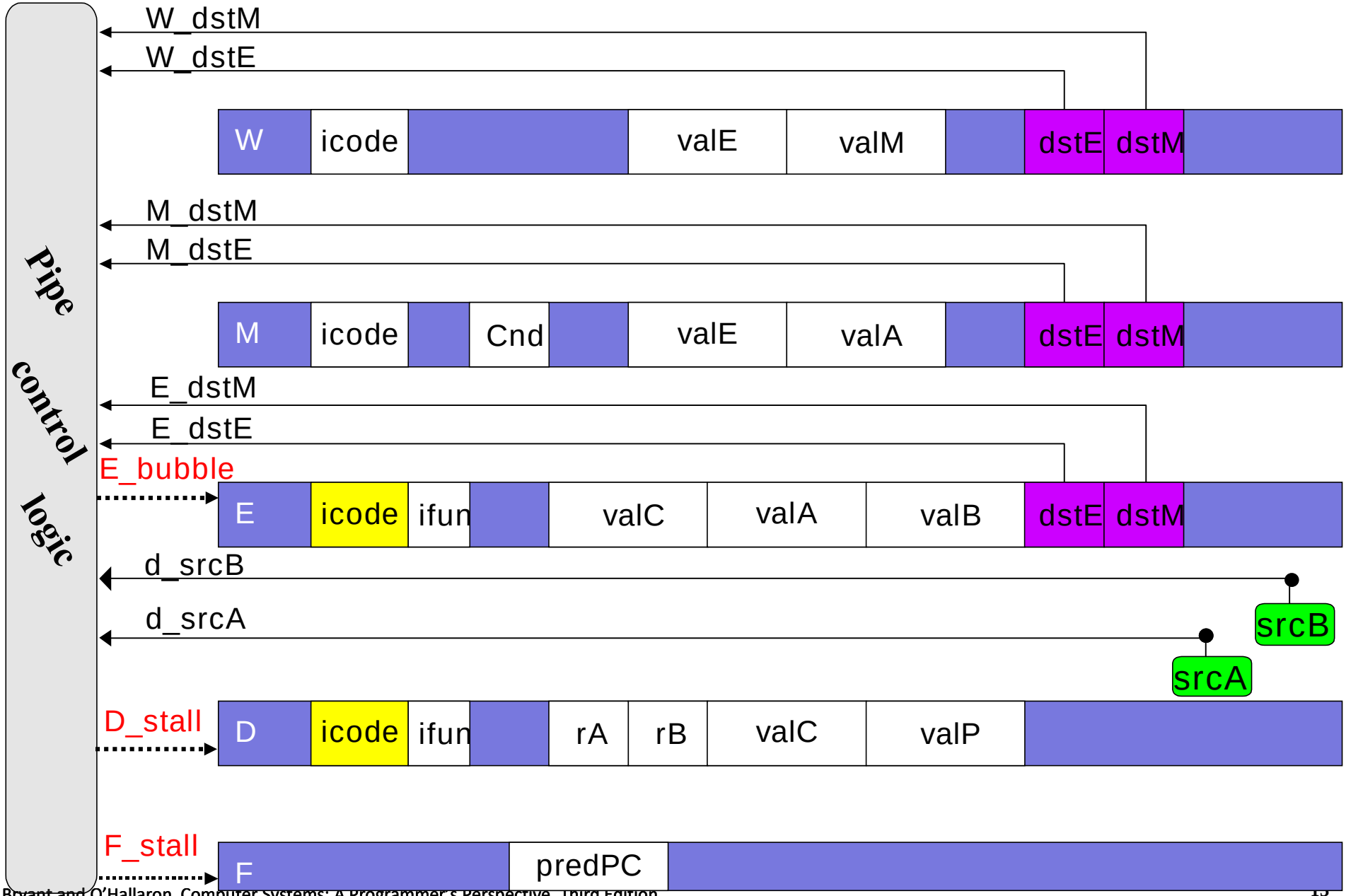
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

周期 8

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

- 指令停顿在译码阶段
- 紧随其后的指令阻塞在取指阶段
- 气泡插入到执行阶段
  - 像一条自动产生的nop指令
  - 穿过后续阶段

暂停实现


Pipe control logic
W_dstM
W_dstE
W
icode
valE
valM
dstE
dstM
M_dstM
M_dstE
M
icode
Cnd
valE
valA
dstE
dstM
E_dstM
E_dstE
E_bubble
E
icode
ifun
valC
valA
valB
dstE
dstM
d_srcB
d_srcA
srcB
srcA
D_stall
D
icode
ifun
rA
rB
valC
valP
F_stall
F
predPC

暂停实现

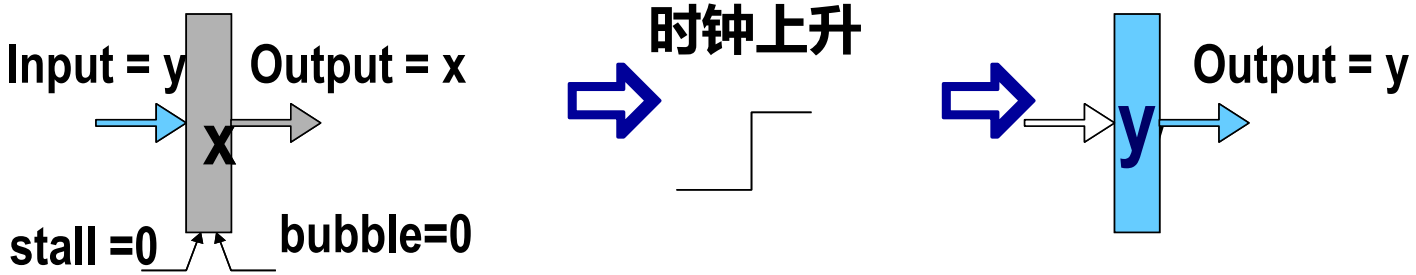
- 流水线控制
  - 组合逻辑检测暂停条件
  - 为流水线寄存器的更新方式设置模式信号

流水线控制的初始版本

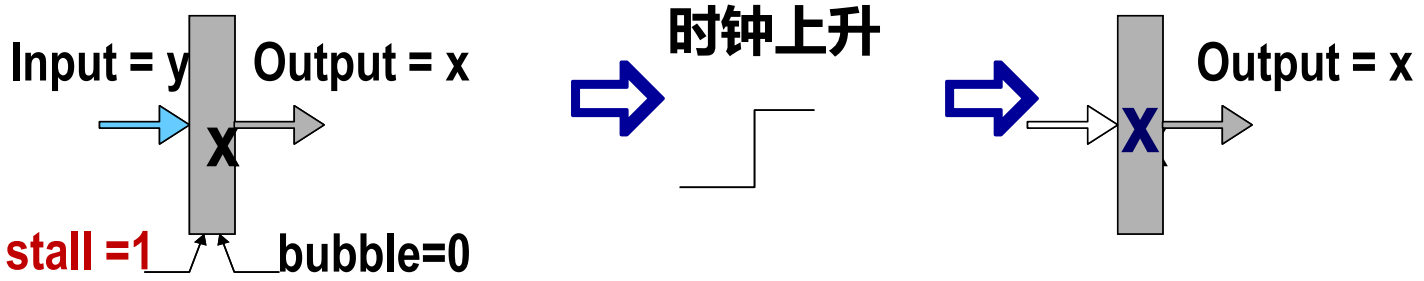
```
bool F_stall =
    d_srcA in {E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM }
    ||
    d_srcB in {E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM };

bool D_stall =
    d_srcA in {E_dstE, E_dstM, M_dstE, M_dstM,  W_dstE, W_dstM }
     ||
    d_srcB in {E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM };

bool E_bubble =
    d_srcA in {E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM }
   ||
    d_srcB in {E_dstE, E_dstM, M_dstE, M_dstM, W_dstE, W_dstM };
```

流水线寄存器模式

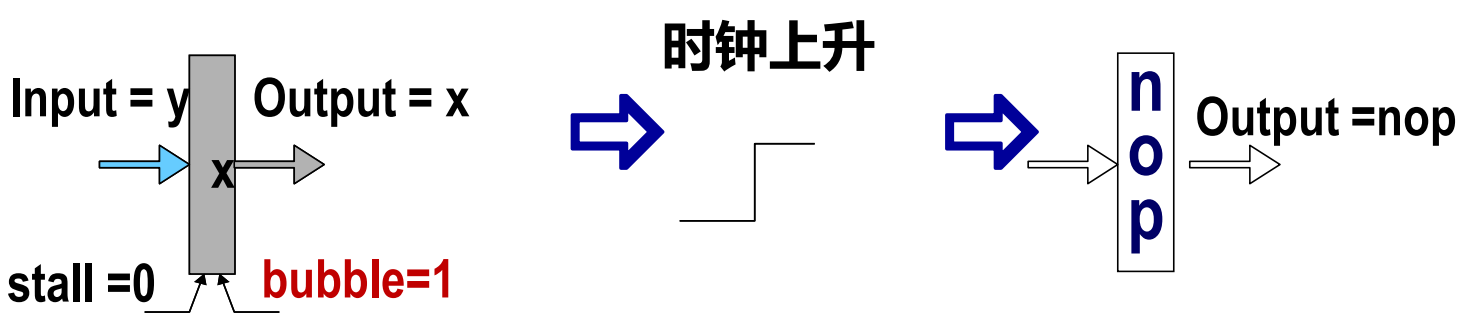
正常



暂停



气泡

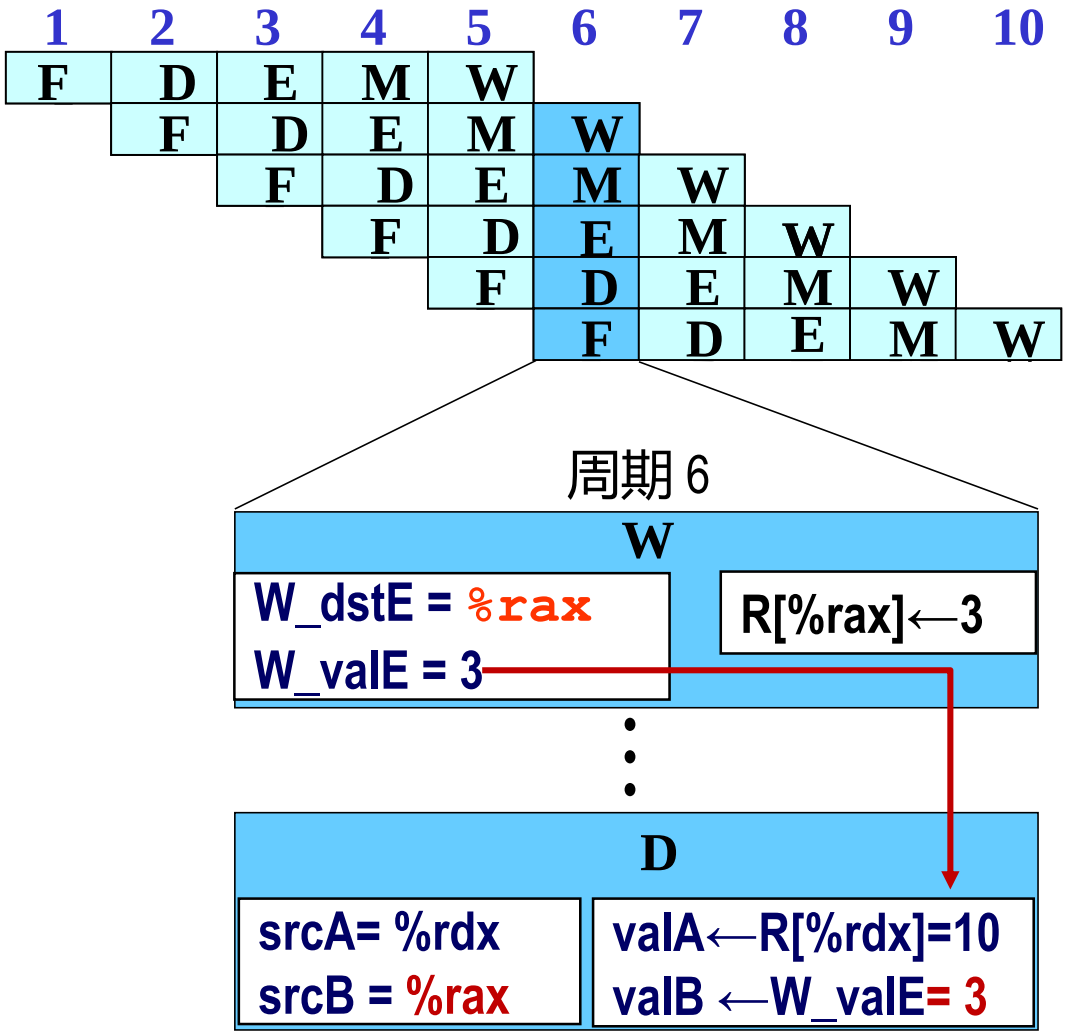


数据转发

- （Naïve ）朴素的流水线
  - 寄存器的写要在写回阶段完成才会发生
  - 源操作数：在译码阶段从寄存器文件中读入
    - 需要在阶段的开始时，就已经保存在寄存器文件中
- 观察
  - 在执行阶段和访存阶段产生的值
- 窍门
  - 将指令生成的值直接传递到译码阶段
  - 需要在译码阶段结尾时可用/有效

数据转发示例

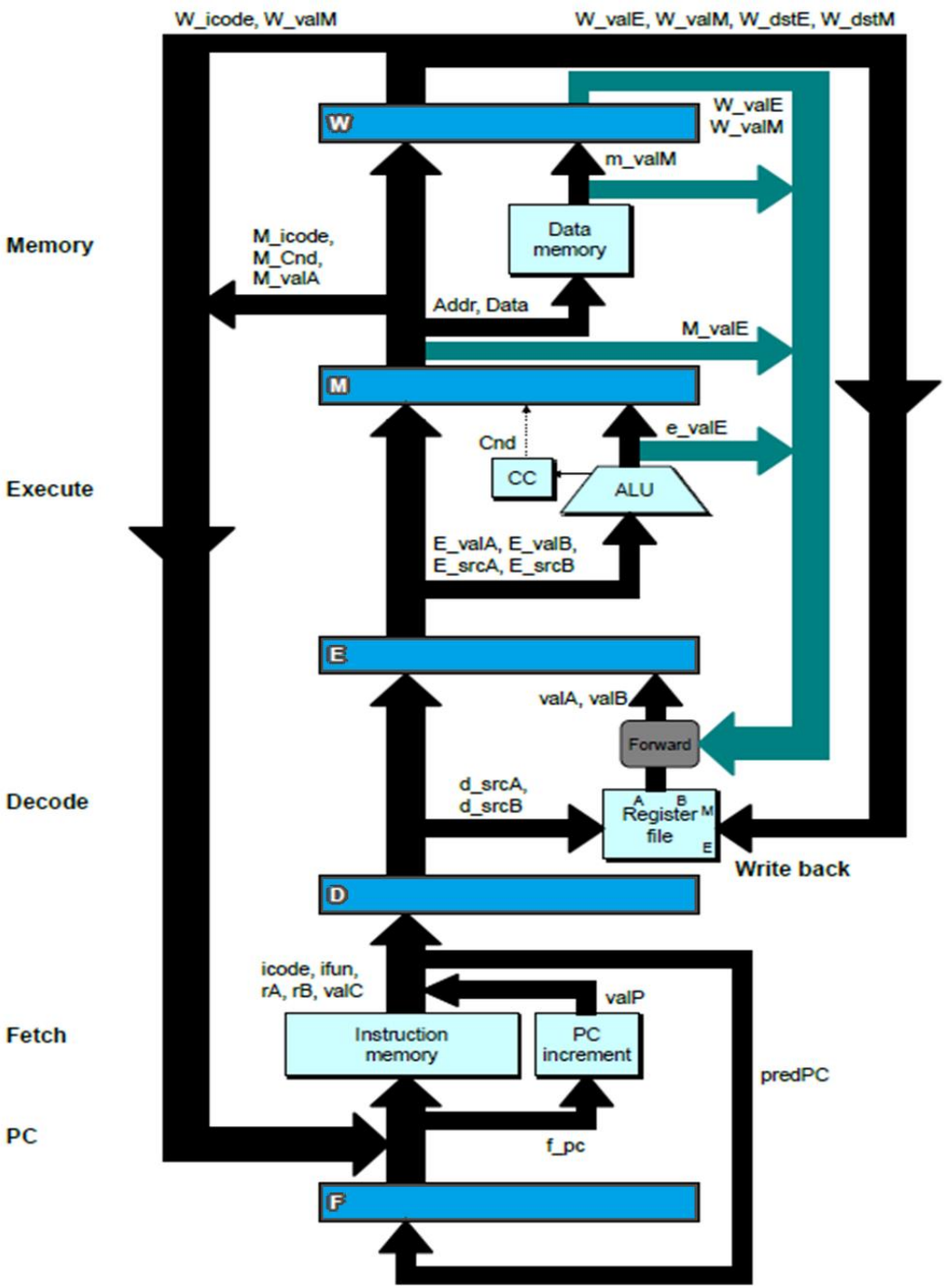
```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



- irmovq 处于写回阶段
- 结果值保存到W流水线寄存器
- 转发给译码阶段，作为valB

旁路路径

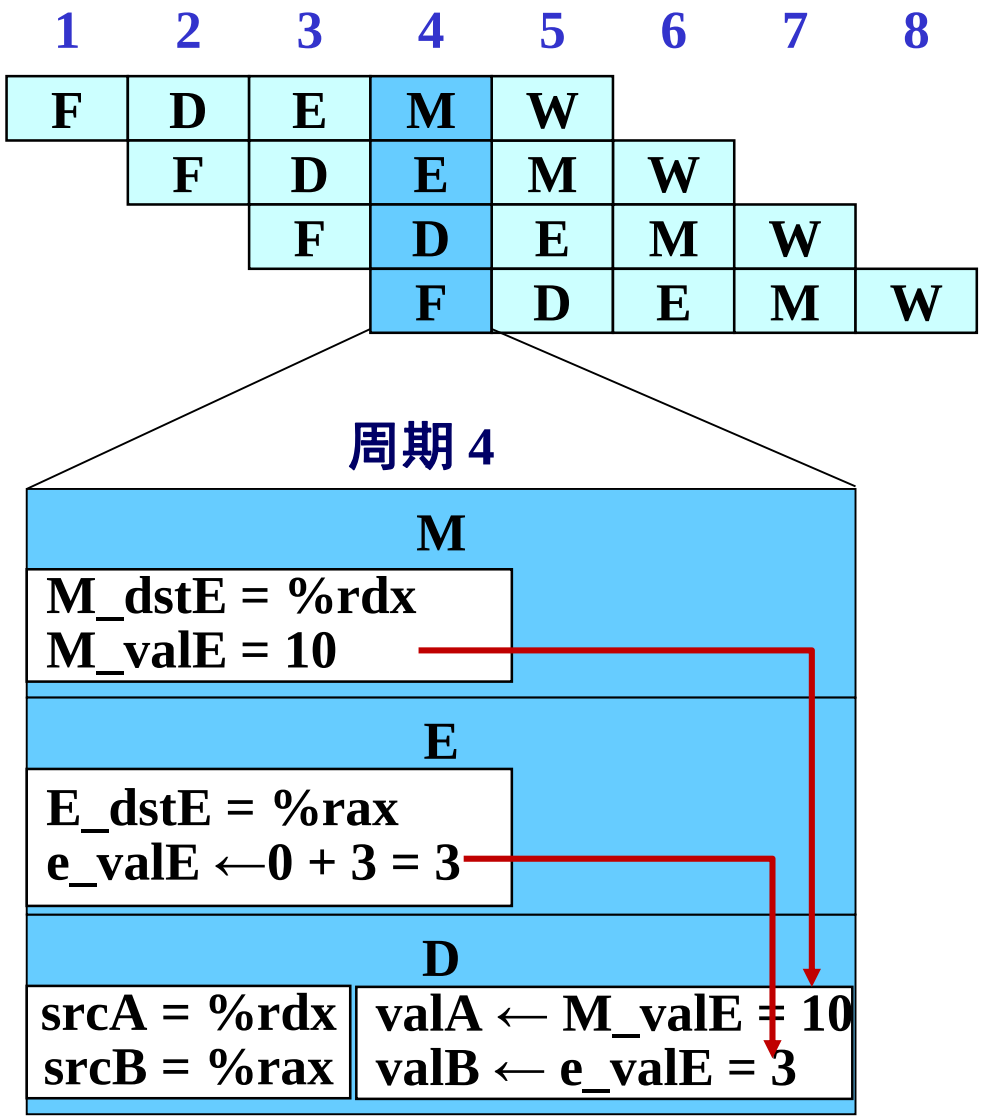
- 译码阶段
  - “转发逻辑” 选择valA和valB
  - 通常来自寄存器文件
  - 转发： 从后面的流水线阶段获得valA和valB
- 转发源
  - 执行阶段: valE
  - 访存阶段: valE, valM
  - 写回阶段: valE, valM



数据转发示例 #2

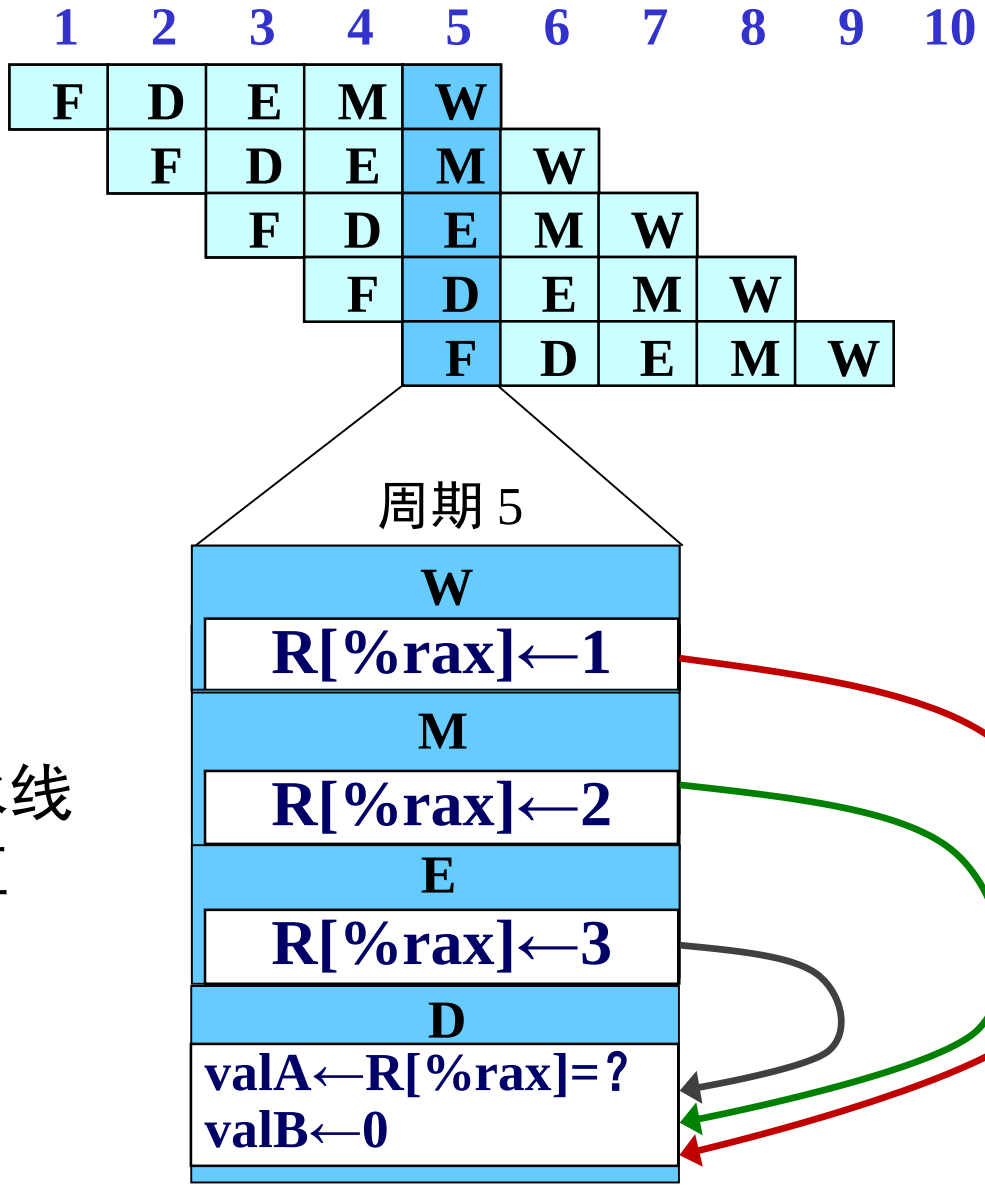
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

- 寄存器%rdx
  - 由ALU在前一个周期产生
  - 从访存阶段转发，作为valA
- 寄存器%rax
  - 值刚刚由ALU产生
  - 从执行阶段转发，作为valB



转发优先级

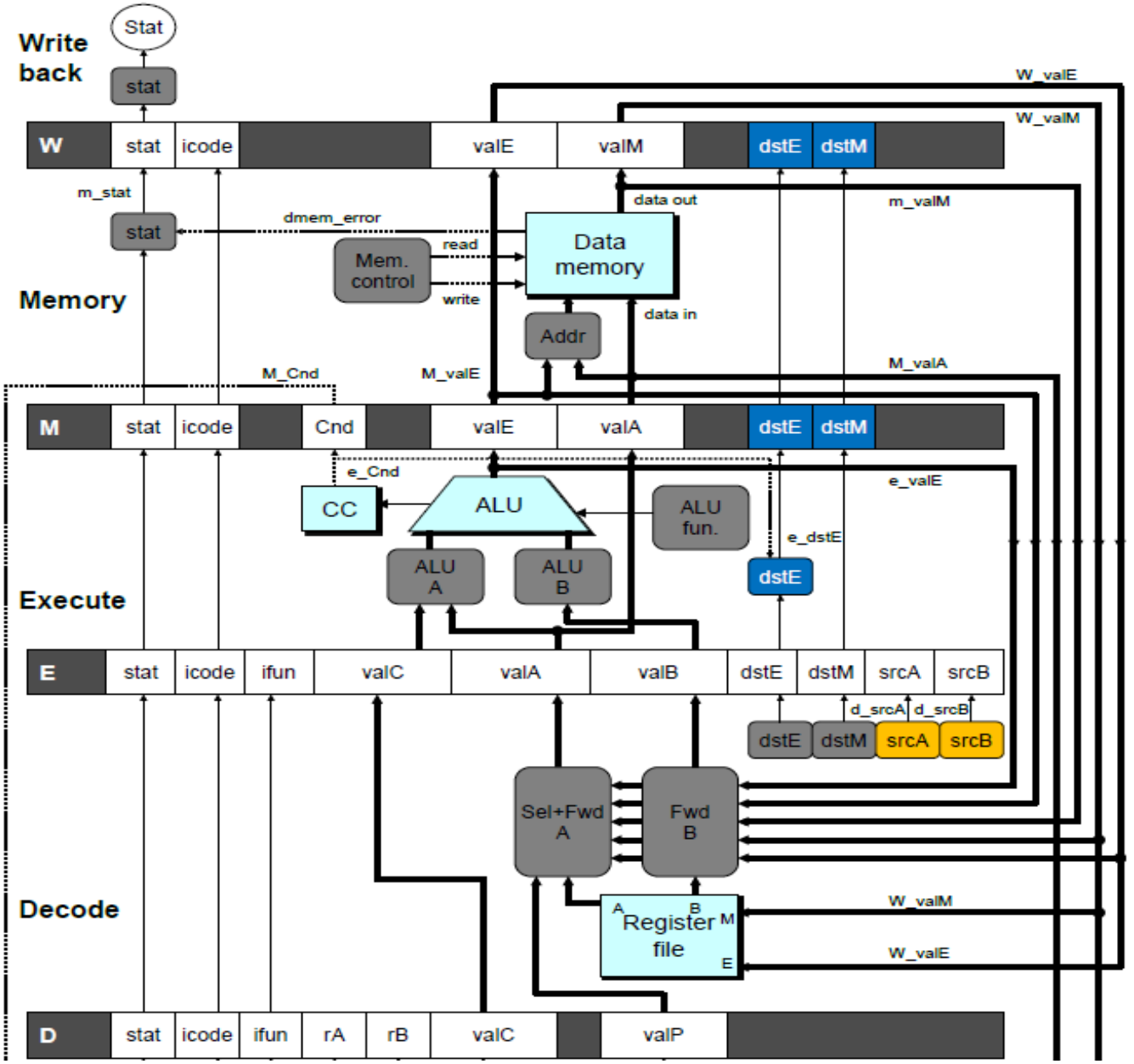
```
# demo-priority.ys
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```



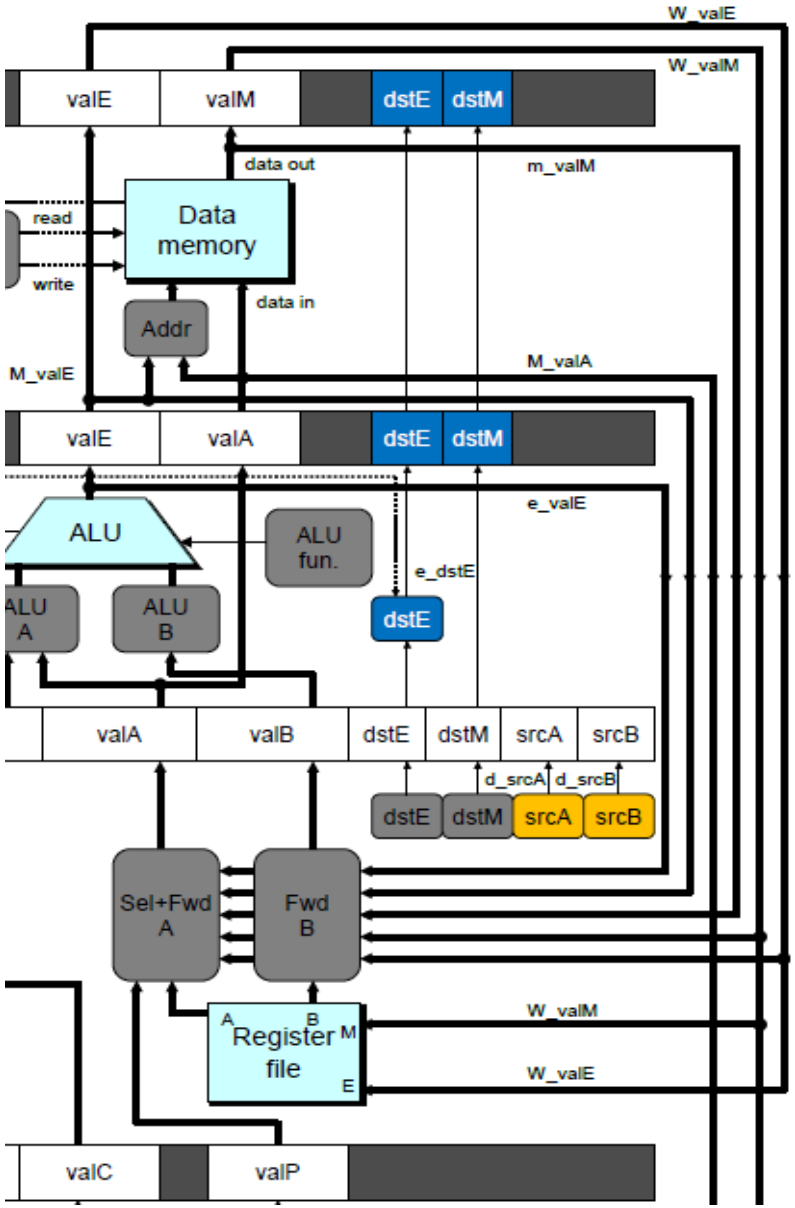
- 多重转发选择
  - 哪一个应该优先
  - 匹配串行语义
  - 使用从最早的流水线阶段获取的匹配值

实现转发

- 在译码阶段从E、M和W流水线寄存器中添加额外的反馈路径
- 在译码阶段创建逻辑块来从valA和valB的多来源中进行选择



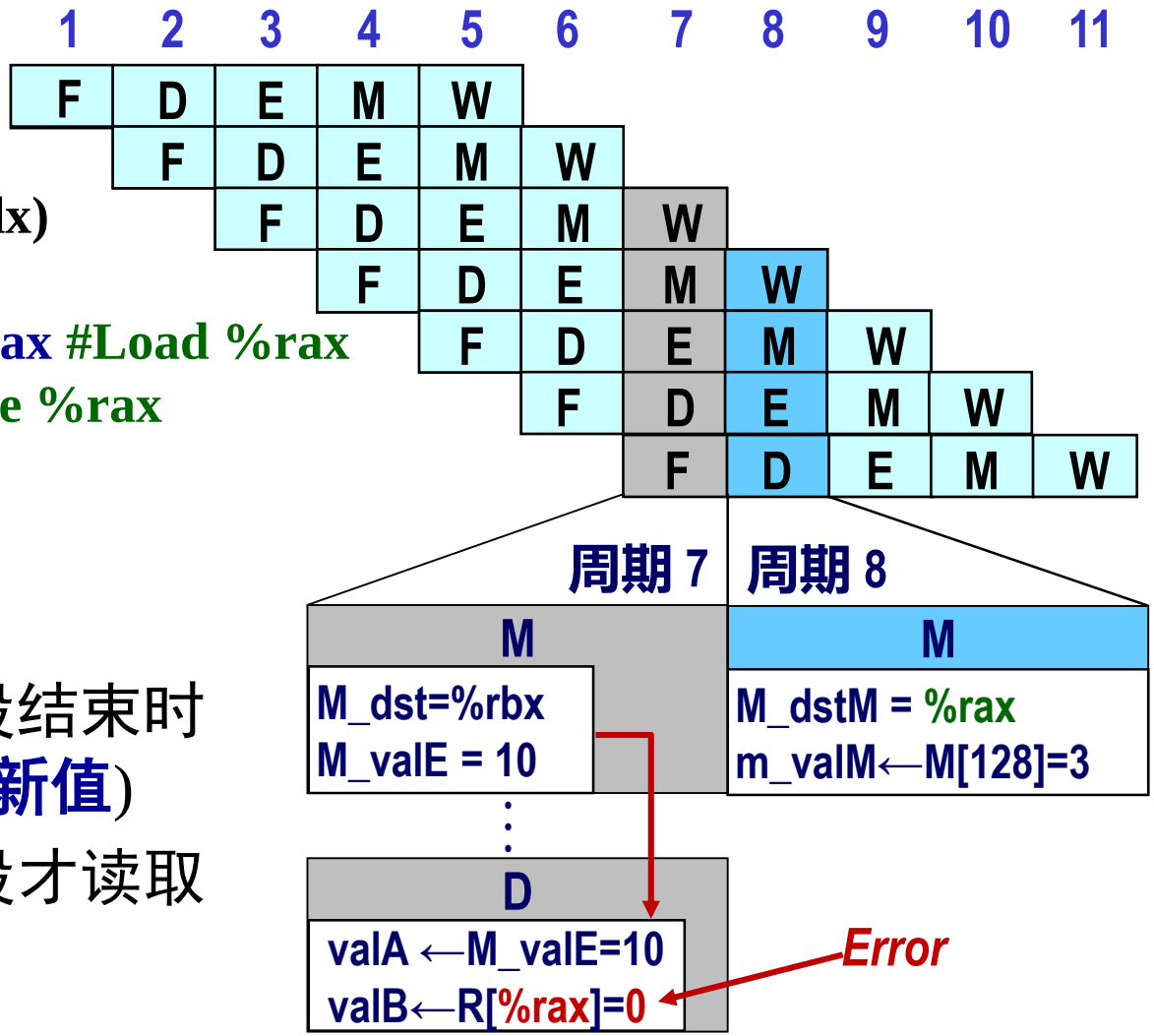
实现转发




```
## 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

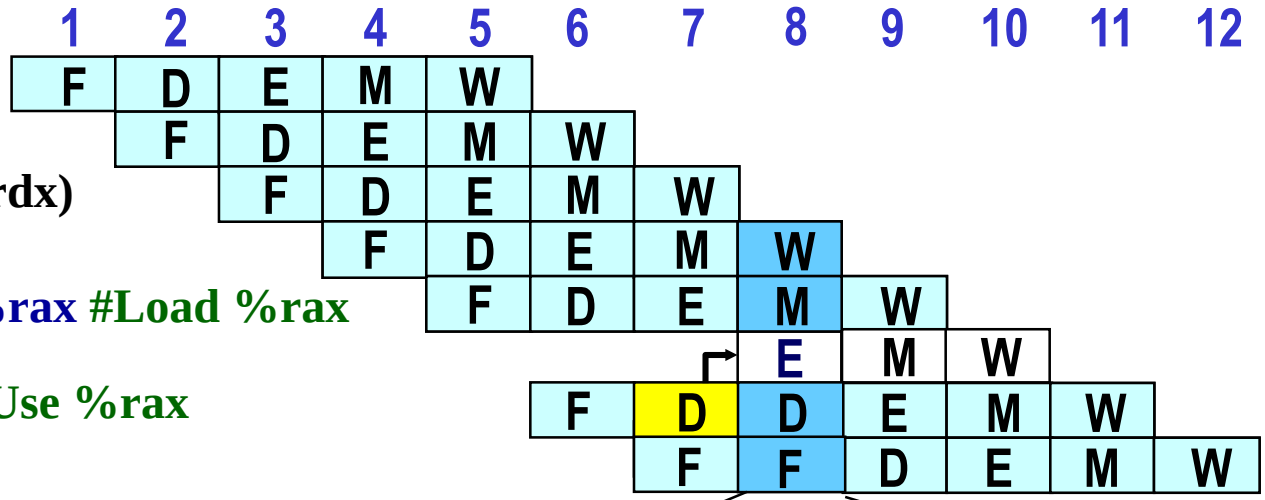
0x000: I rmovq $128, %rdx
0x00a: irmovq $3, %rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10, %rbx
0x028: mrmovq 0(%rdx), %rax #Load %rax
0x032: addq %rbx,%rax #Use %rax
0x034: halt

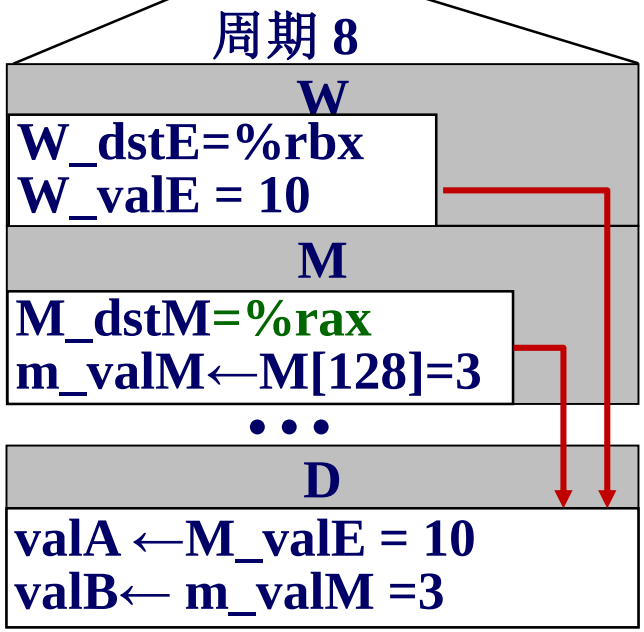


- 加载/使用 依赖
  - 在周期7译码阶段结束时需要的值(**%rax 新值**)
  - 在周期8访存阶段才读取到该值(**m_valM**)

避免 加载/使用 冒险

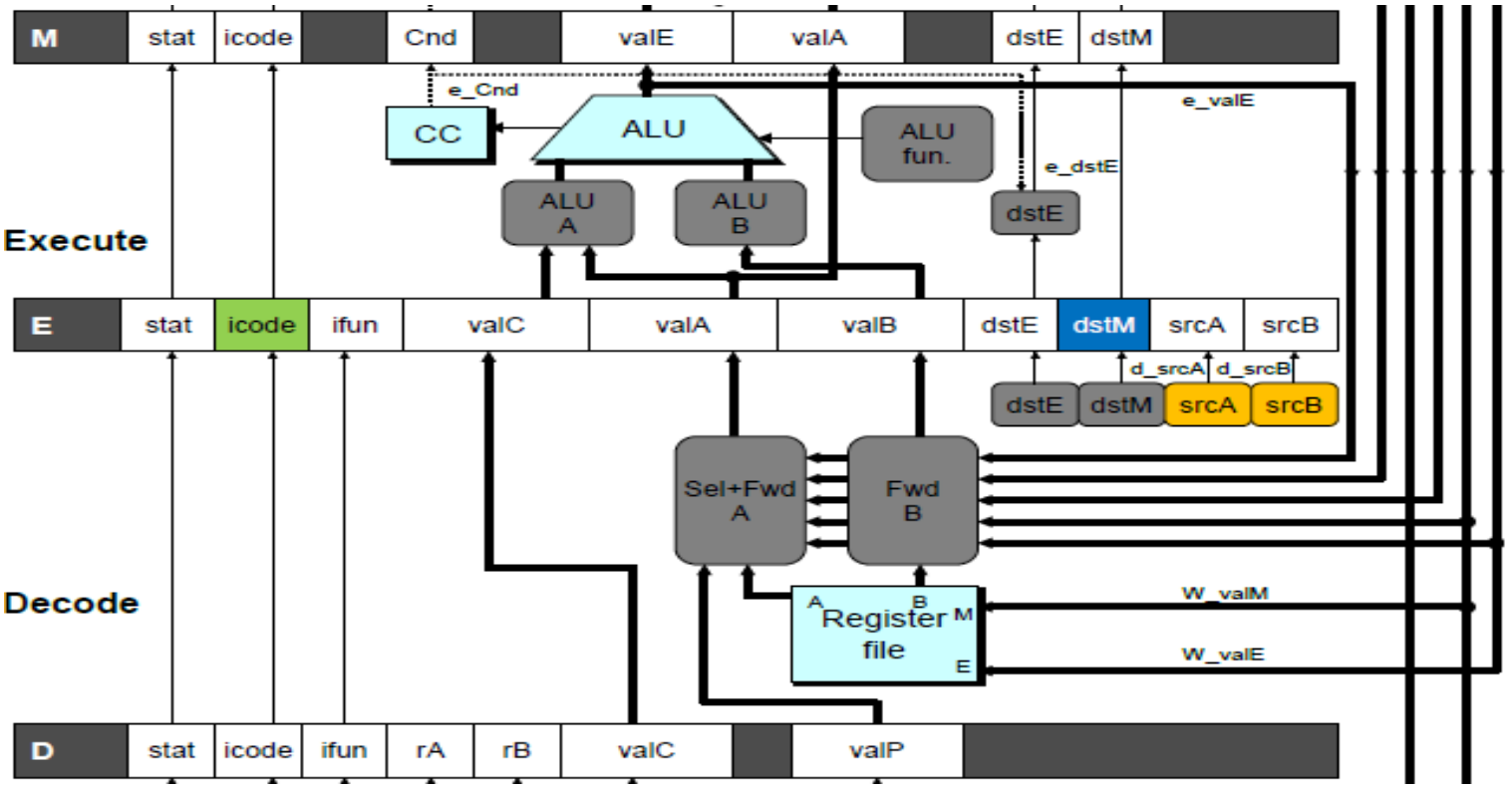
```
0x000: I rmovq $128, %rdx
0x00a: irmovq $3, %rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10, %rbx
0x028: mrmovq 0(%rdx), %rax #Load %rax
                bubble
0x032: addq %rbx, %rax #Use %rax
0x034: halt
```





- 使用数据的指令暂停一个周期
- 然后就可以获取从访存阶段转发的加载值

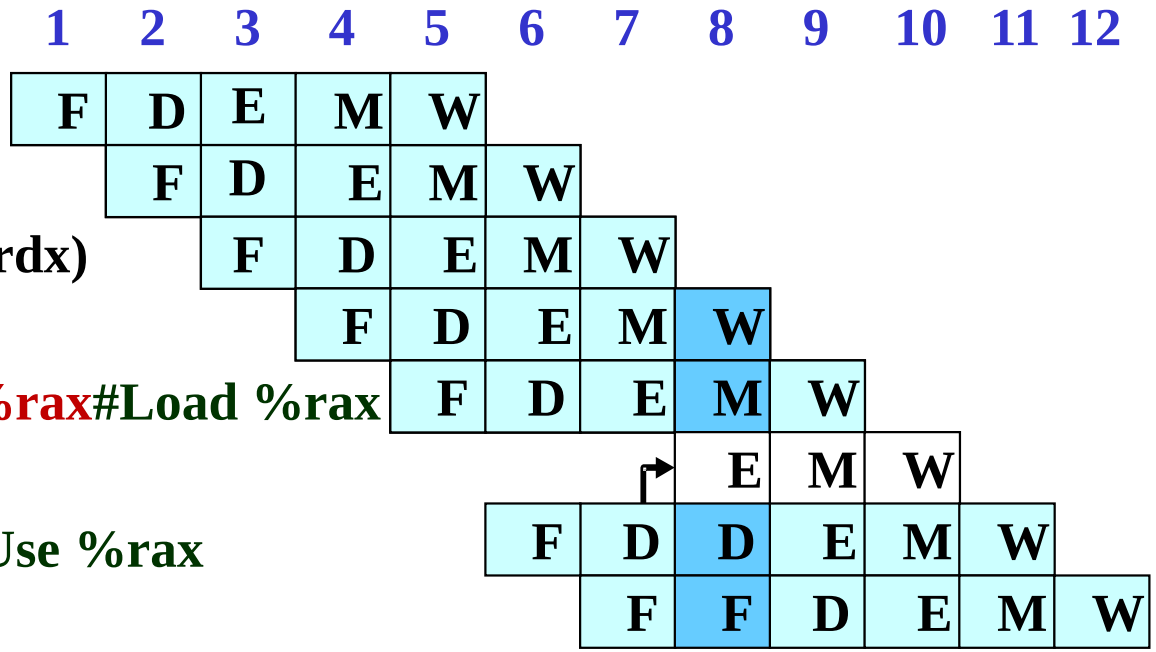
检测 加载/使用 冒险



条件	触发
加载/使用 冒险	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }

加载/使用 冒险的控制

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



- 将指令暂停在取指和译码阶段
- 在执行阶段注入气泡

条件	F	D	E	M	W
加载/使用 冒险	暂停	暂停	气泡	正常	正常

分支预测错误示例

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
```

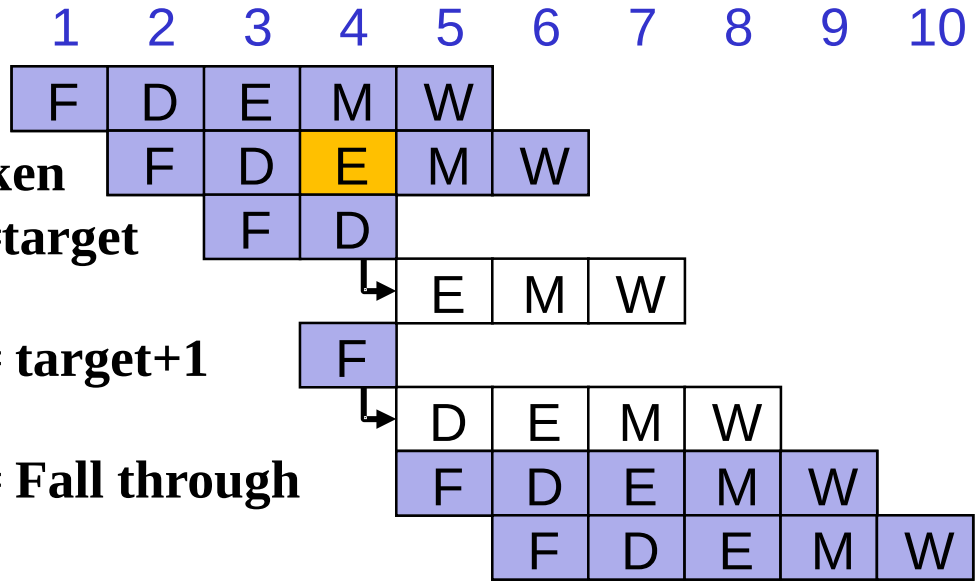
- 应该只执行前8条指令

处理预测错误

```
#prog7
0x000: xorq %rax, %rax
0x002: jne target #Not Taken
0x00b: irmovq $1, %rax # Fall through
0x015: halt
0x016: target:
0x016: irmovq $2, %rdx #target
0x020: irmovq $3, %rbx # target+1
0x02a: halt
```

处理预测错误

```
#prog7
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
```



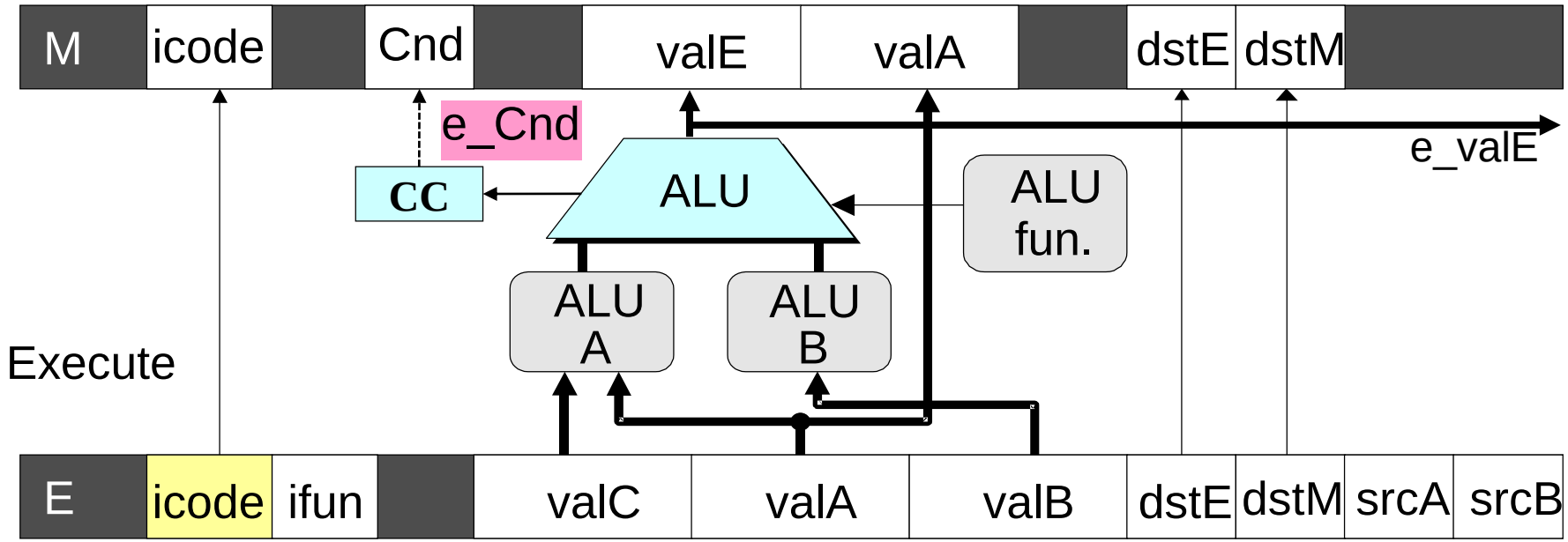
作为预测分支

- 取出 2 条目标指令

当预测错误时的取消动作

- 在执行阶段检测到不应选择该分支
- 在随后的指令周期中，将处于执行和译码阶段的指令用气泡替换掉
- 不会再有副作用

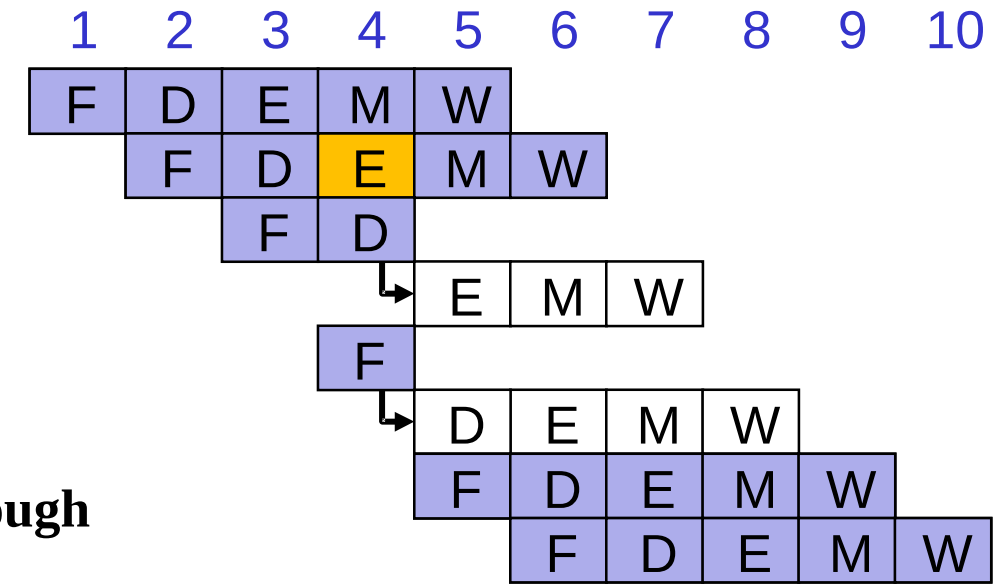
检测分支预测错误



条件	触发
分支预测错误	`E_icode = IJXX & !e_Cnd`

预测错误的控制

```
#prog7
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
```



条件	F	D	E	M	W
分支预测错误	正常	气泡	气泡	正常	正常

Return示例

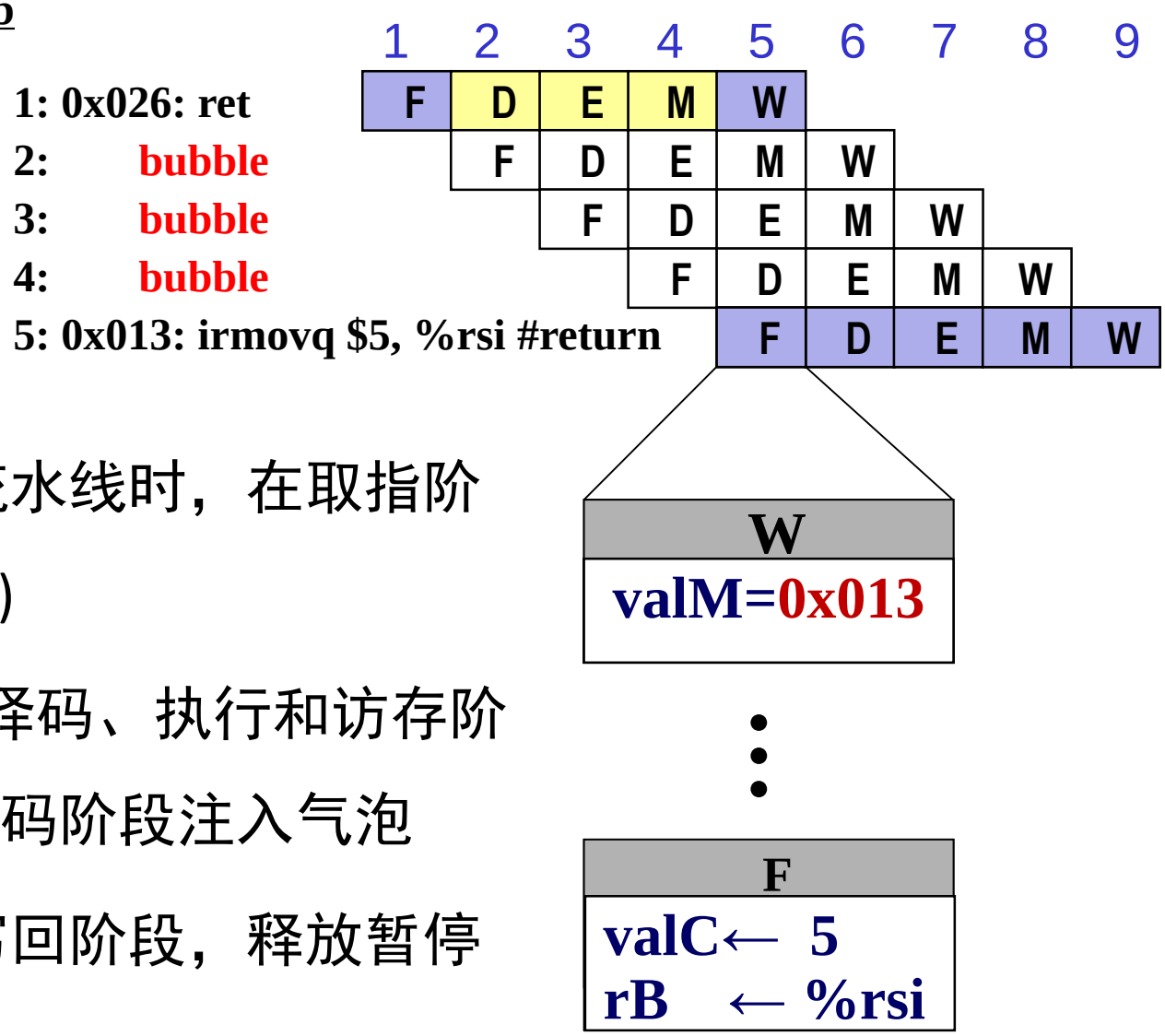
demo-retb.ys

- 之前执行了3条额外的指令

```
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
```

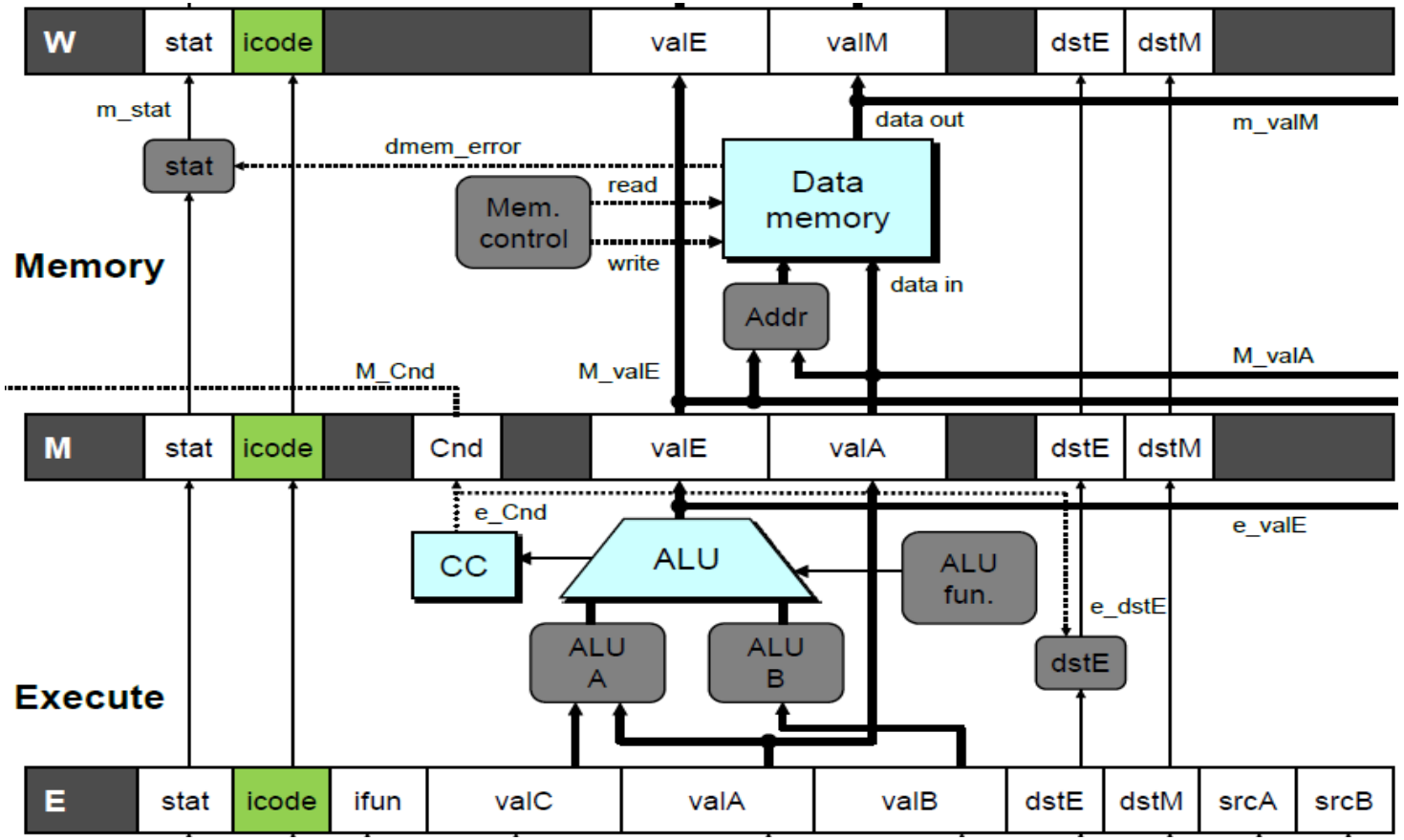
正确的Return示例

#demo_retb



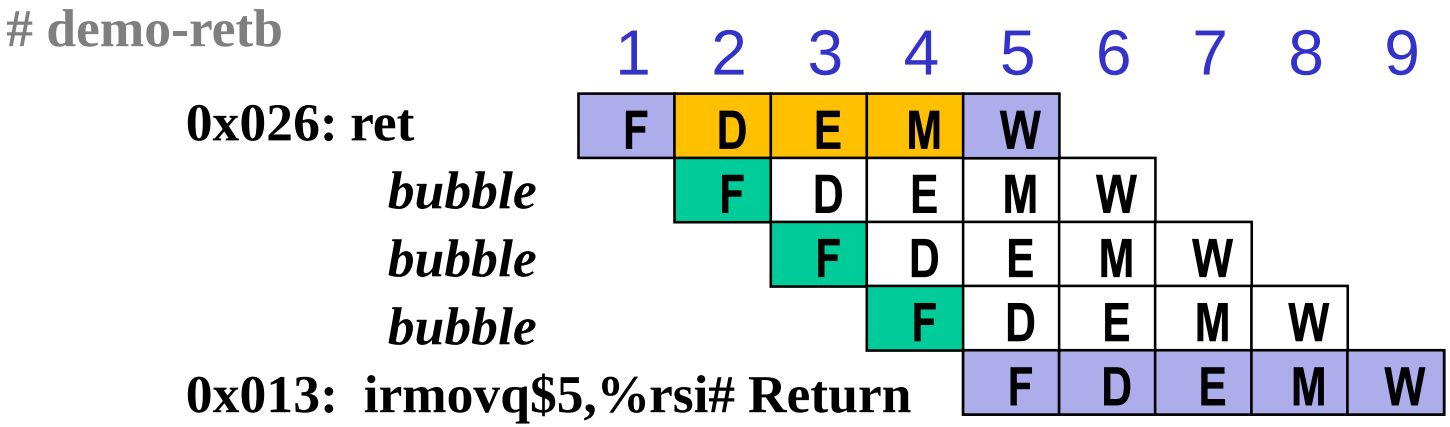
- 当ret经过流水线时，在取指阶段暂停(stall)
- 当 ret 处于译码、执行和访存阶段时，在译码阶段注入气泡
- 当ret到达写回阶段，释放暂停

检测Return



条件	触发
处理 `ret`	`IRET in { D_icode, E_icode, M_icode }`

Return的控制



条件	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常

特殊控制情况

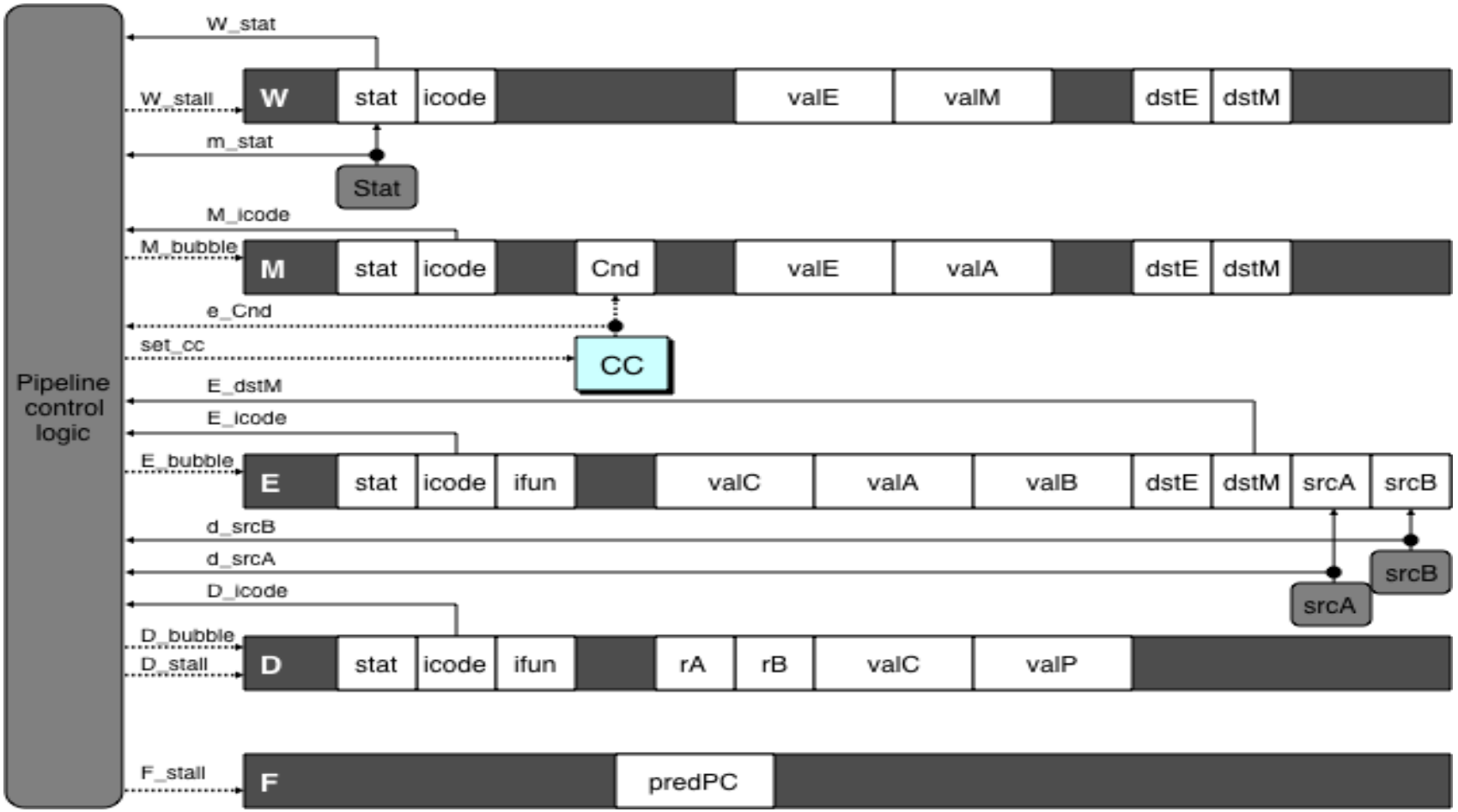
- 检测

条件	触发
处理 ret	IRET in { D_icode, E_icode, M_icode }
加载/使用 冒险	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
分支预测错误	E_icode = IJXX & !e_Cnd

- 动作

条件	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
加载/使用 冒险	暂停	暂停	气泡	正常	正常
分支预测错误	正常	气泡	气泡	正常	正常

实现流水线控制



- 组合逻辑产生流水线控制信号
- 动作发生在每个追随周期(following cycle)开始的时候

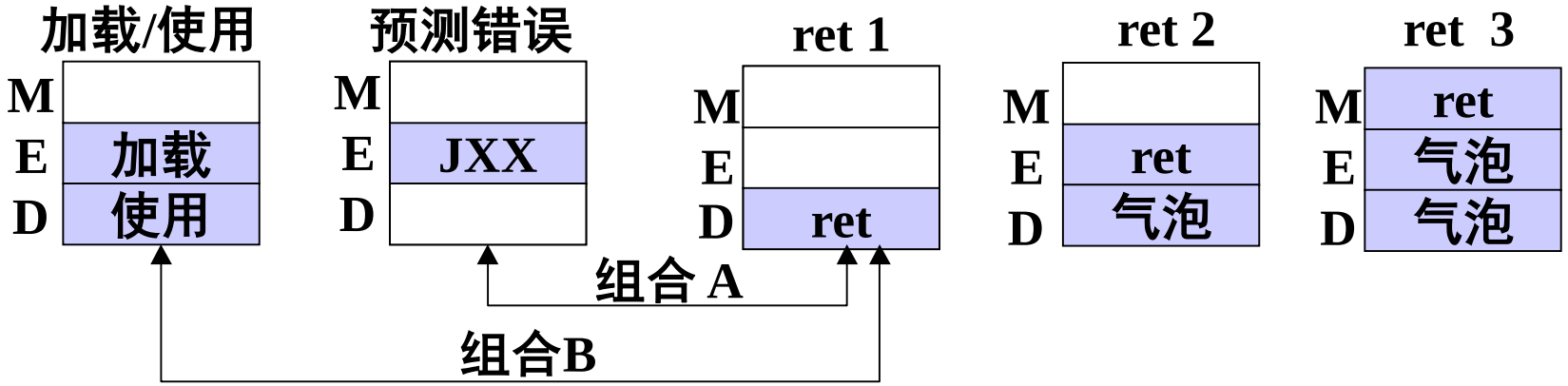
流水线控制的初始版本

```
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  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 };
```

流水线控制的初始版本

```
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 };
```

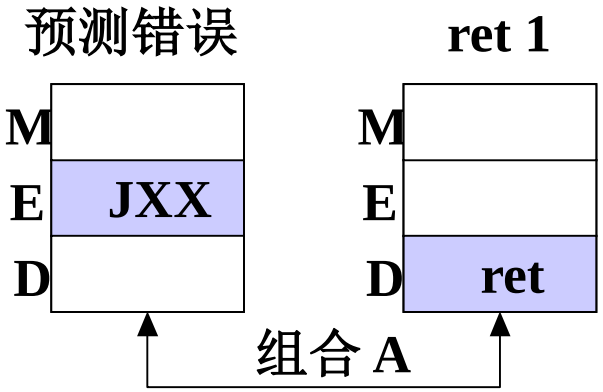
控制组合

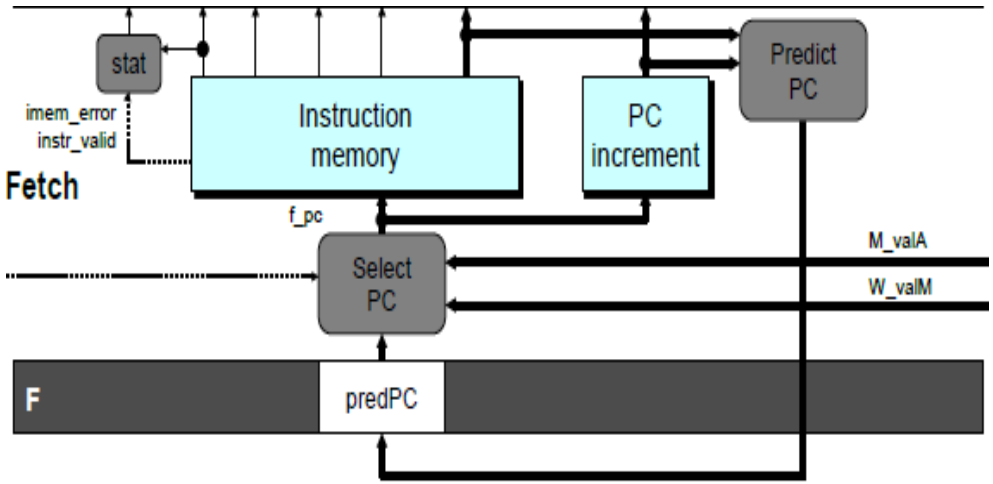


- 在一个时钟周期内可能出现多个特殊情况

- **组合A**
  - 不应选择的分支(not-taken branch)
  - ret在不应选择的分支中
- **组合B**
  - 指令从内存读取到%rsp
  - 紧跟着ret指令

控制组合A





条件	F	D	E	M	W
处理ret	暂停	气泡	正常	正常	正常
分支预测错误	正常	气泡	气泡	正常	正常
组合	***暂停***	*气泡*	*气泡*	*正常*	*正常*

- 当成分支预测错误来处理
- 暂停F流水线寄存器
- 下一个周期，PC选择逻辑将会选择JXX后面那条指令的地址(**M_valM**)

控制组合 B

条件	F	D	E	M	W
处理ret	暂停	气泡	正常	正常	正常
加载/使用 冒险	暂停	暂停	气泡	正常	正常
组　合	*暂停*	*气泡 + 暂停*	*气泡*	*正常*	*正常*
期望的情况	暂停	暂停	气泡	正常	正常

- 将会尝试插入气泡和暂停流水线寄存器D
- 处理器发出流水线错误信号
- 组合B需要特殊处理：暂停D
  - 加载/使用 冒险应该有优先权
  - ret指令被保持在译码阶段以推迟一个周期

正确的流水线控制逻辑

```
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 });
```

条件	F	D	E	M	W
处理ret	暂停	气泡	正常	正常	正常
加载/使用 冒险	暂停	暂停	气泡	正常	正常
组合	*暂停*	*暂停*	*气泡*	*正常*	*正常*

- 加载/使用 冒险应该有优先权
- ret指令应该被保持在译码阶段以推迟一个周期

流水线总结

- 数据冒险
  - 大部分使用转发处理
    - 没有性能损失
  - 加载/使用 冒险需要一个周期的暂停
- 控制冒险
  - 将检测到分支预测错误时取消指令
    - 两个时钟周期被浪费
  - 暂停取指阶段，直到ret通过流水线
    - 三个时钟周期被浪费
- 控制组合
  - 必须仔细分析
  - 首个版本有细微的缺陷
    - 只有不寻常的指令组合才会出现