# 第四章 处理器体系结构

## 4-5——流水线实现高级技术

# 本节主要内容

- **数据冒险的其处理方法**
  - 流水线暂停
  - 数据转发
- **加载使用冒险的处理方法**
- **控制冒险的处理方法**

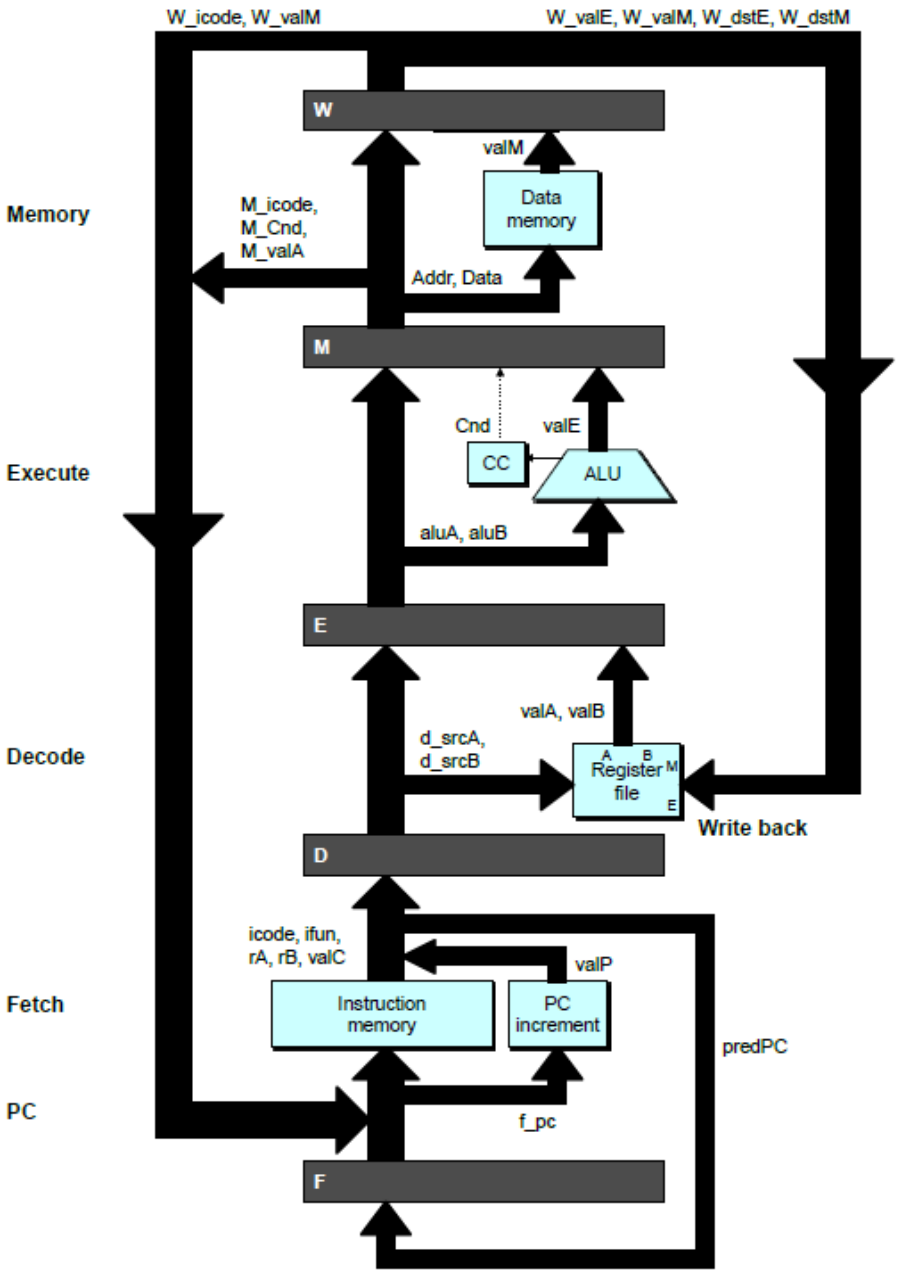
# 回顾

*使流水线处理器工作！*

- 数据冒险
  - 指令使用寄存器R为目的，瞬时之后使用寄存器R为源
  - 一般情况，不要降低流水线的速度
- 控制冒险
  - 条件分支错误
    - 我们的设计能够预测参与的所有分支
    - 执行2条额外的指令后正确的指令开始被执行
  - 从ret指令中获得返回地址
    - 执行3条额外的指令后正确的指令开始被执行

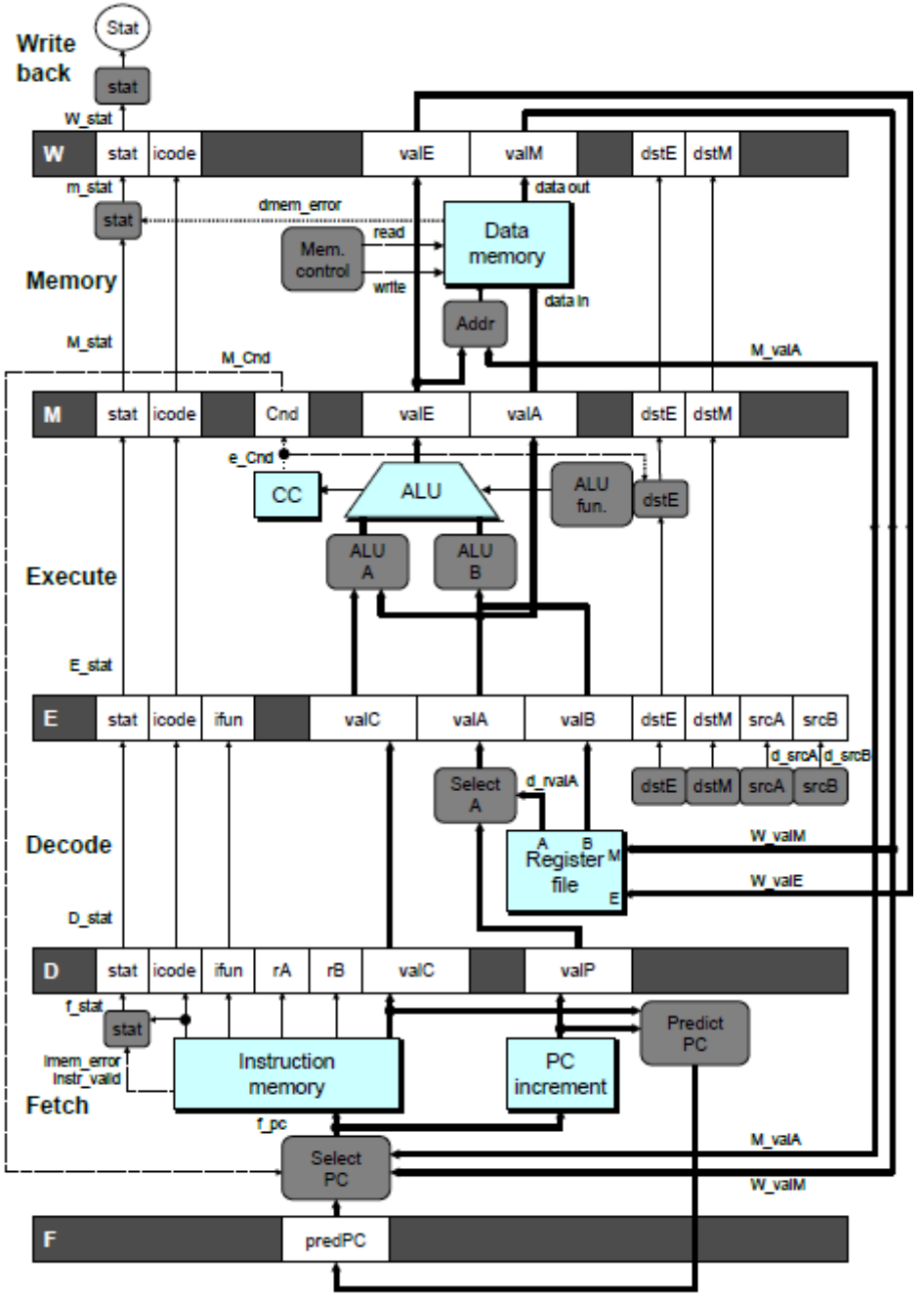
# 流水阶段

- 取指
  - 选择当前PC
  - 读取指令
  - 计算增加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



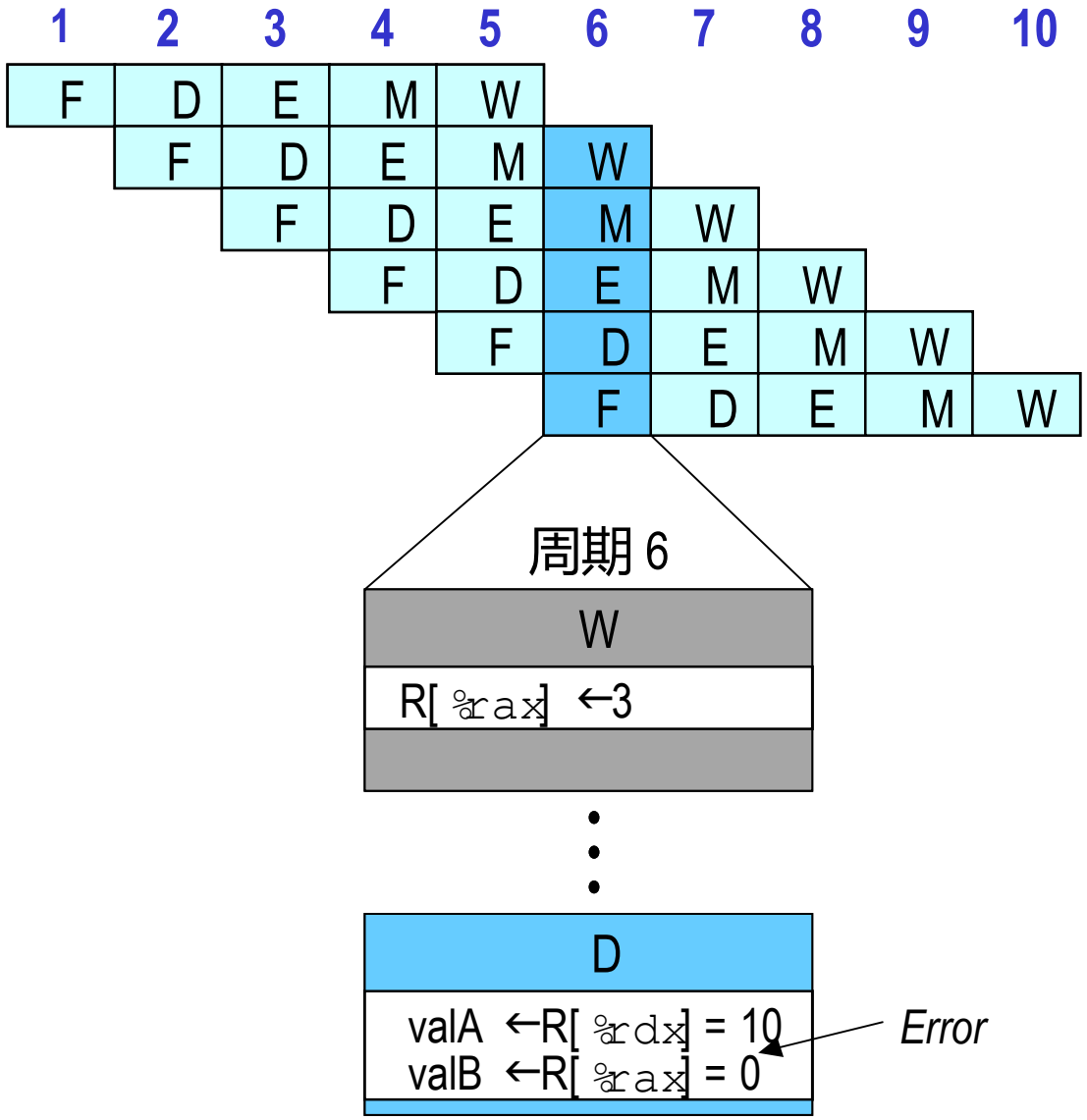
# PIPE- 硬件

- 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已经经过译码



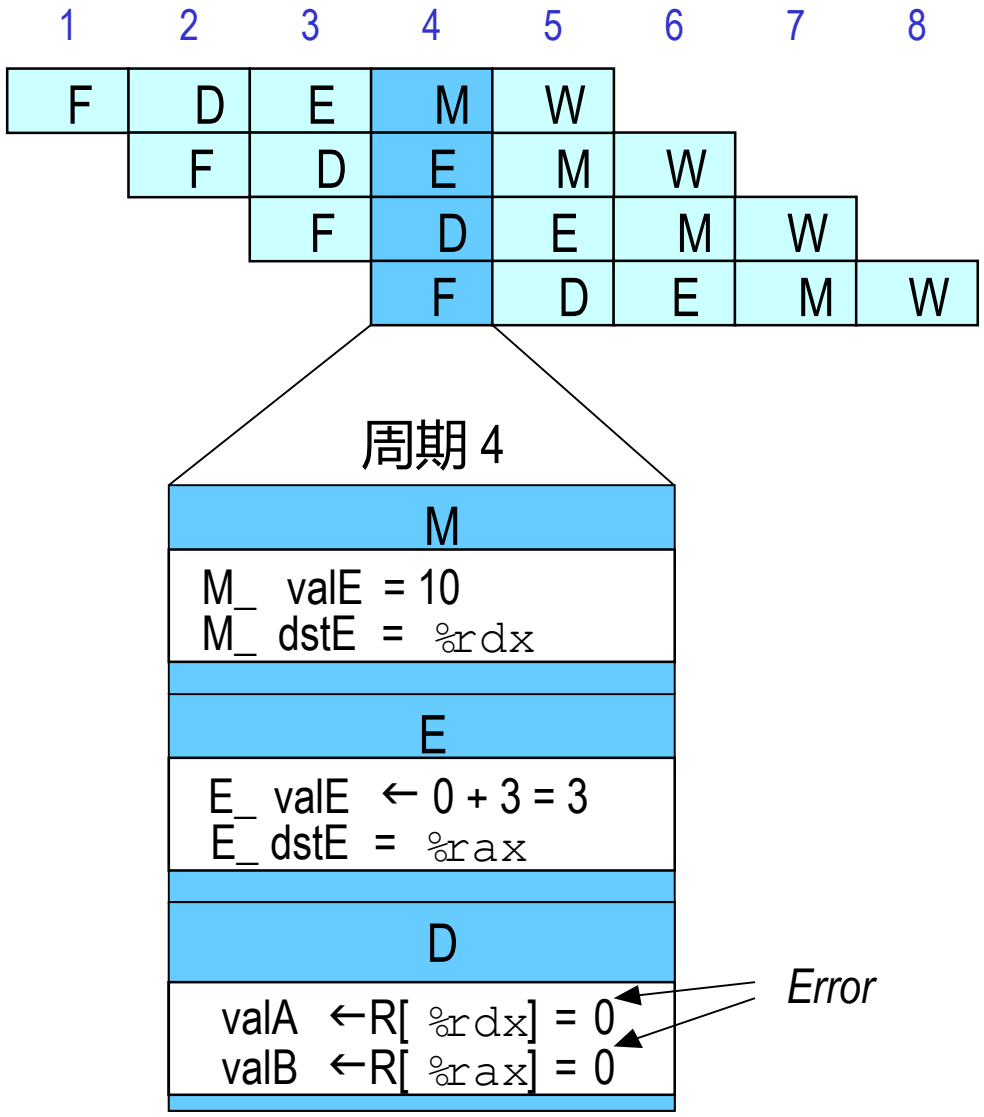
# 数据相关: 两条Nop指令

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



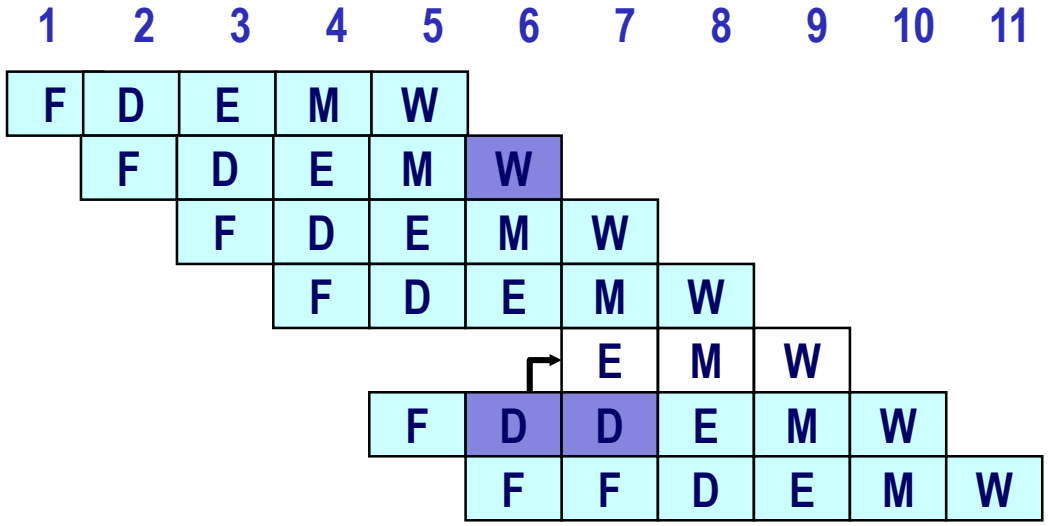
# 数据相关: 无Nop指令

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



# 数据相关的暂停

```
# 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指令后，并且两条指令存在数据冒险，则该指令执行速度可能放慢

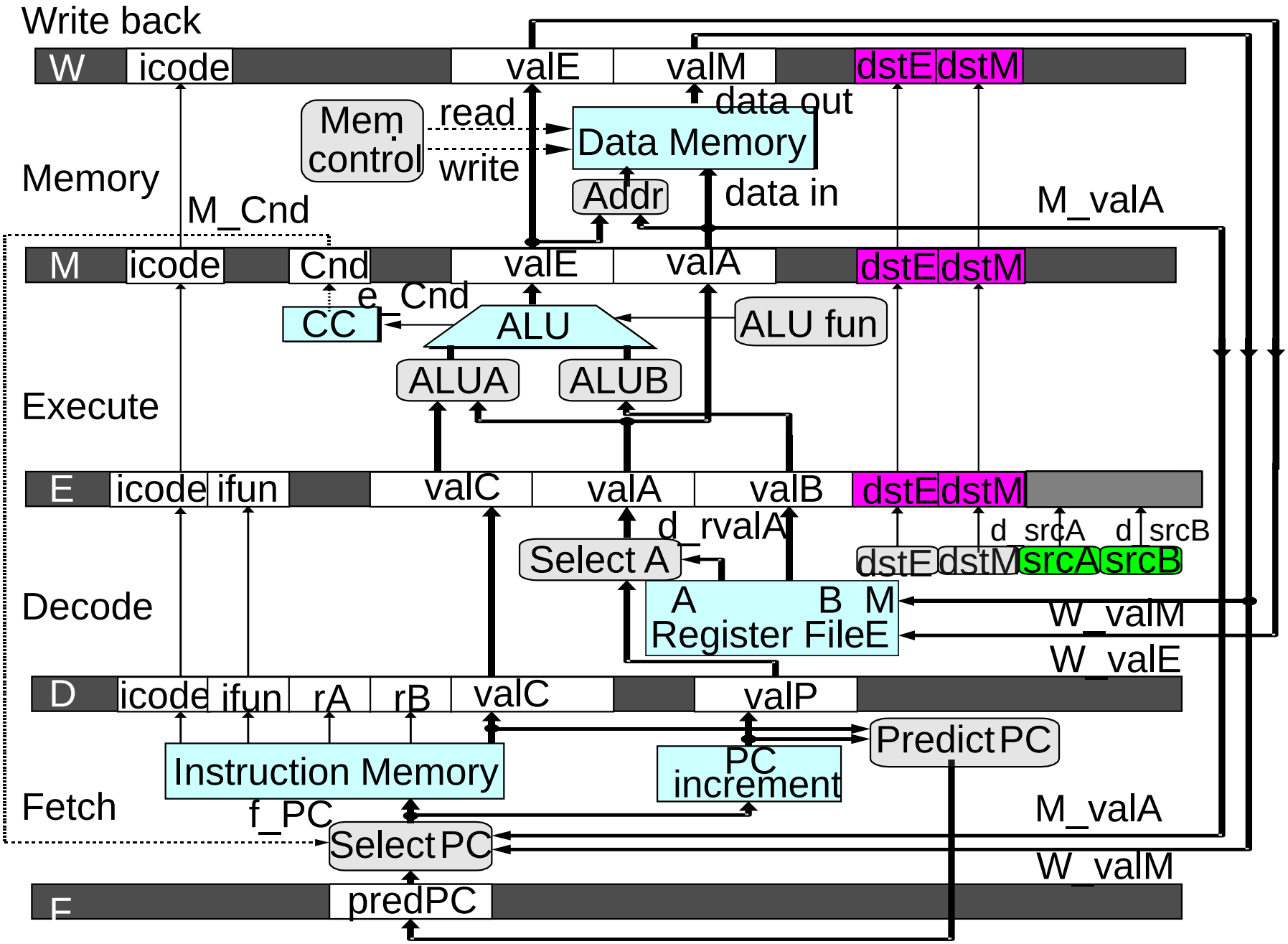
**以下这两种方法都可以，但实现不一样：**

1）将指令阻塞在译码阶段

2）在指令执行阶段动态插入nop

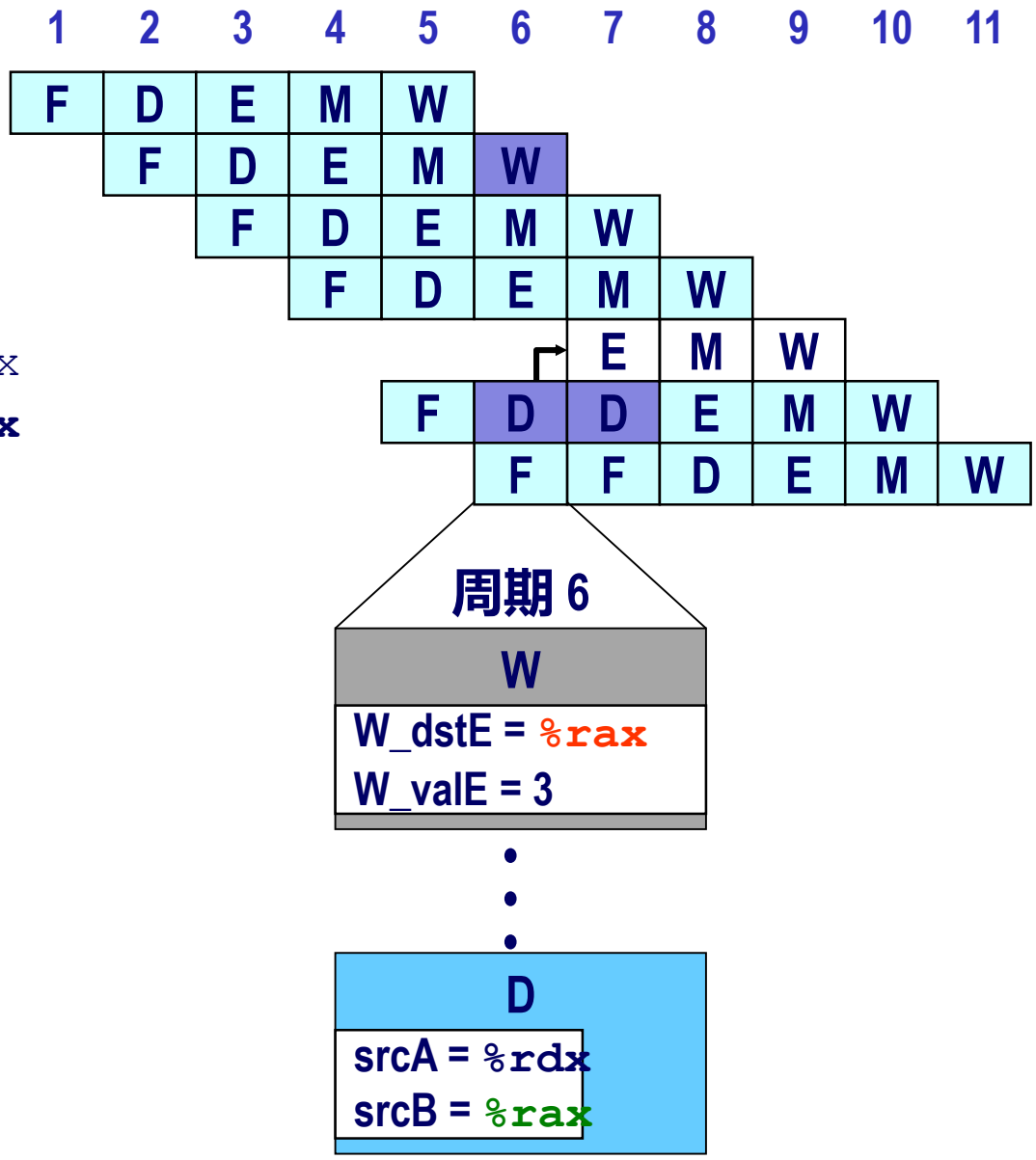
# 暂停条件

- 源寄存器
  - 当前指令的srcA和srcB都处于译码阶段
- 目的寄存器
  - dstE 和dstM 域
  - 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
    - 或表示失败的条件和移动


Write back
W
icode
valE
valM
dstE
dstM
Mem control
read
write
Data Memory
data out
data in
Addr
Memory
M_Cnd
M_valA
M
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun
ALUA
ALUB
Execute
E
icode
ifun
valC
valA
valB
dstE
dstM
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

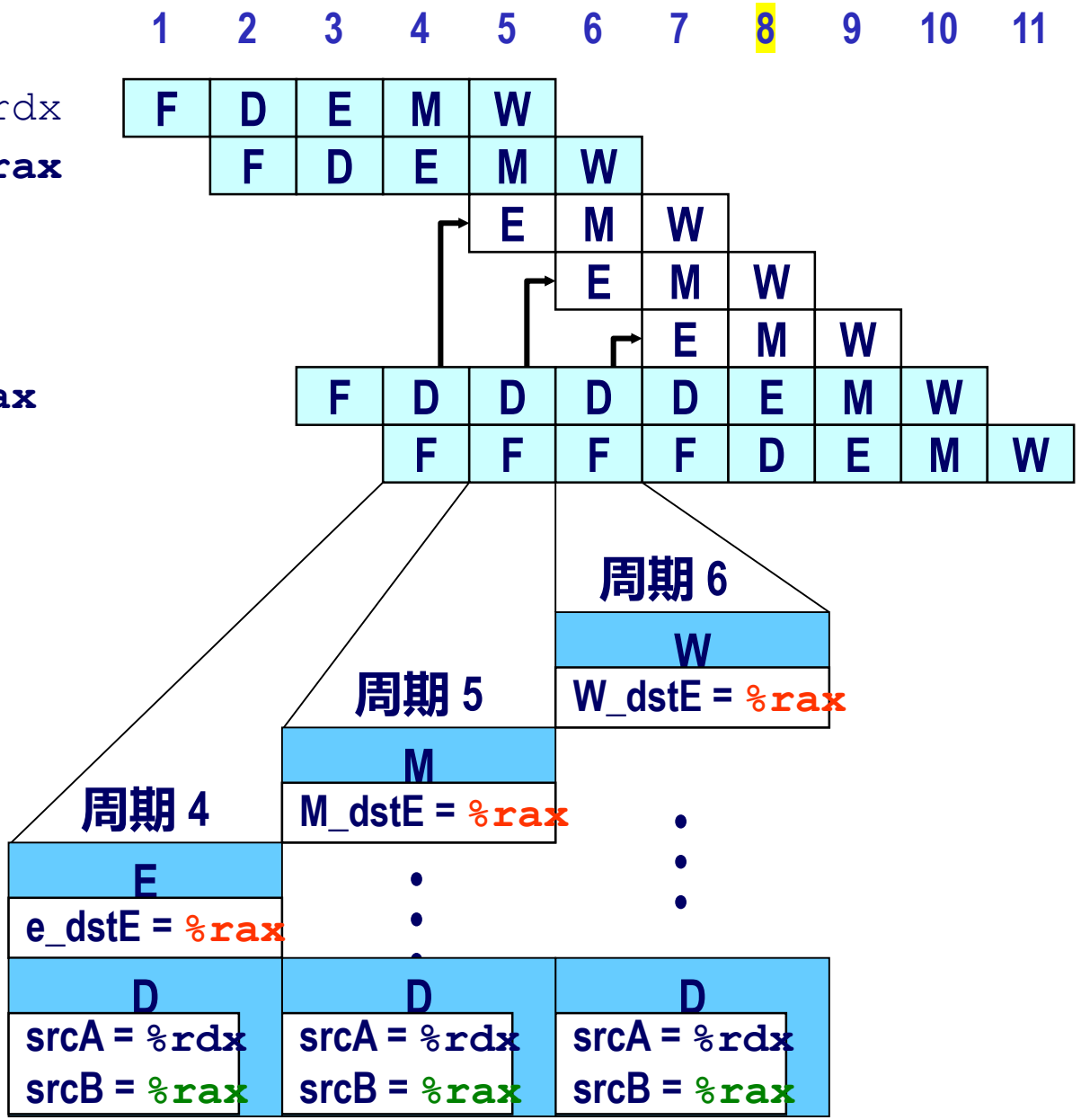
# 检测暂停条件

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

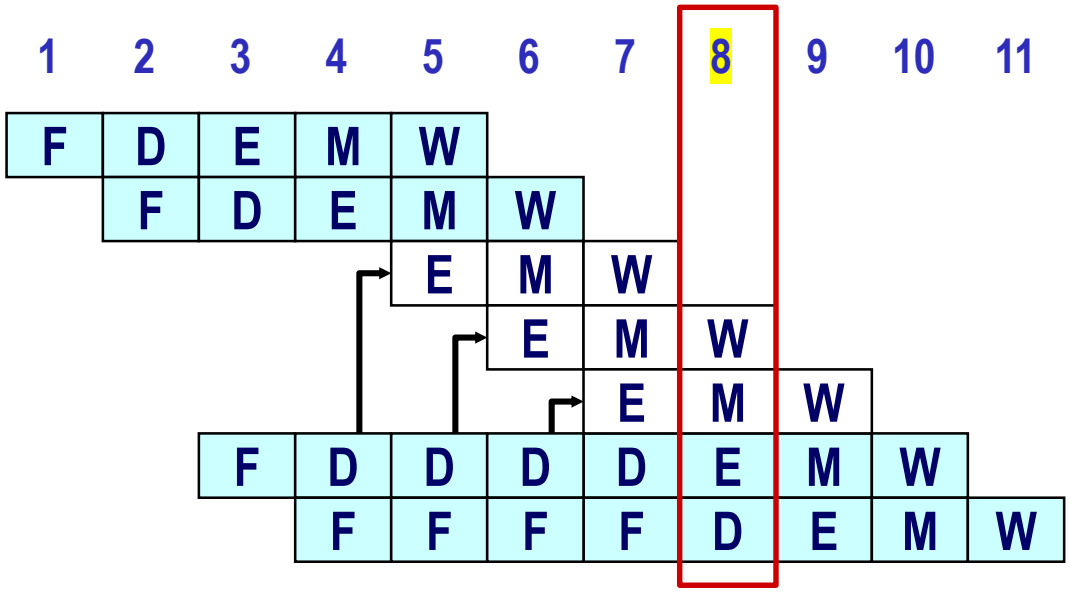


# 暂停 X3

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



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




## 暂停时发生了什么？（以周期8为例）

```
# 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

# 暂停时发生了什么？（以周期8为例）

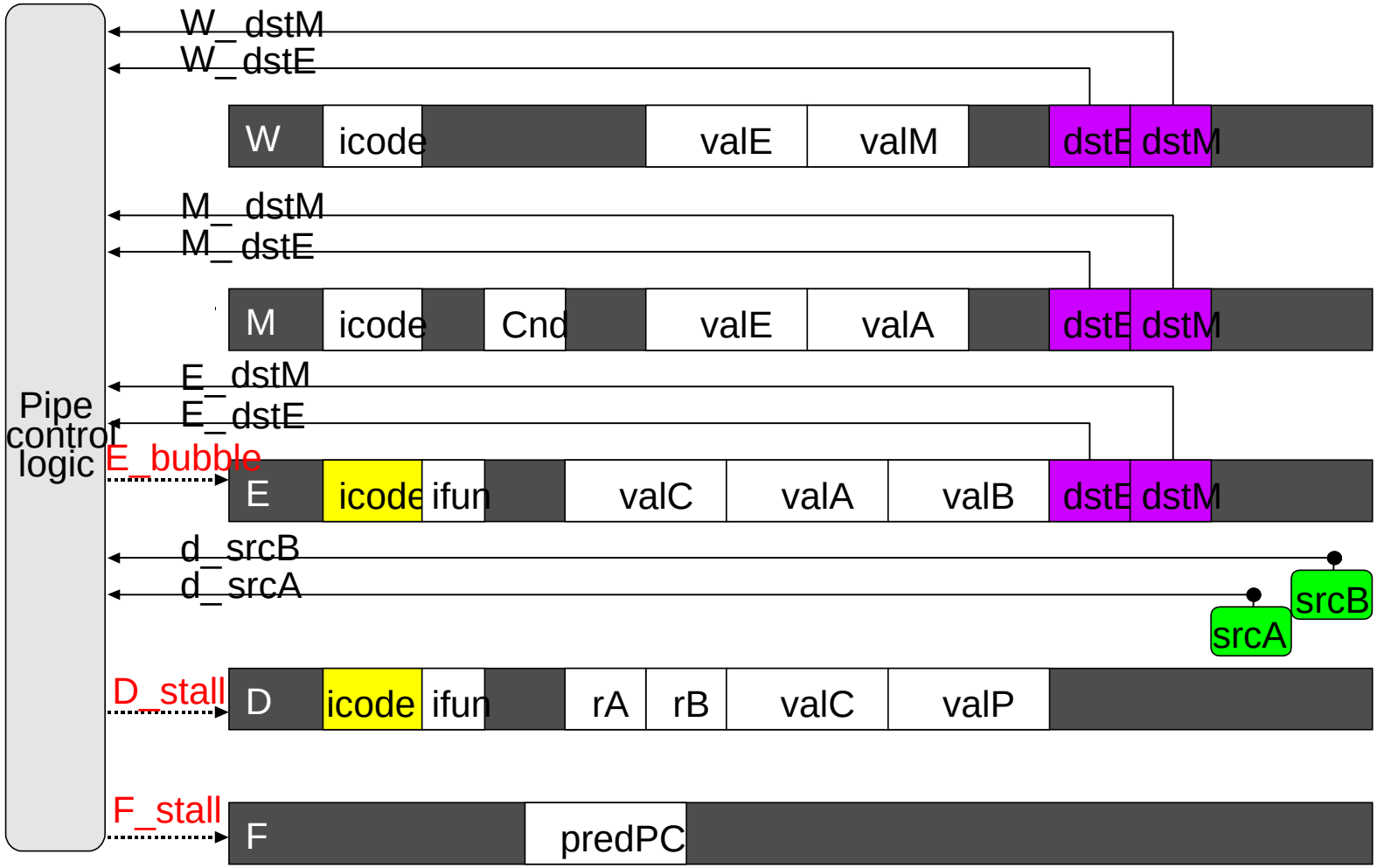
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

**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（F/D流水线寄存器不能写入更新）
- 紧随其后的指令阻塞在取指阶段（PC不能写入更新）
- 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

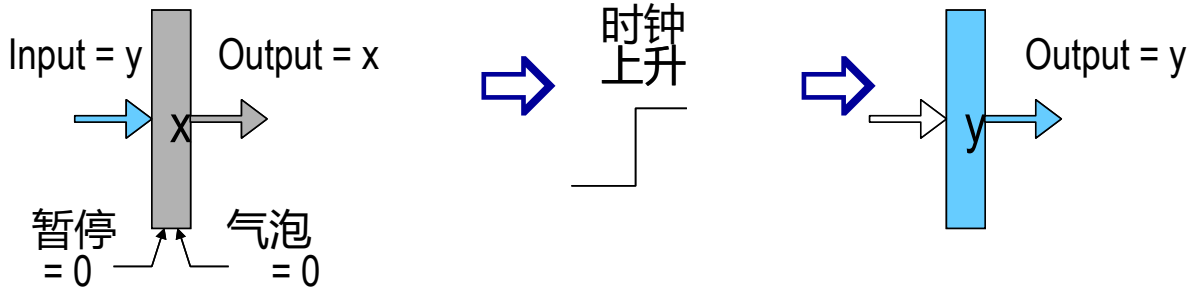
# 暂停实现




- 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

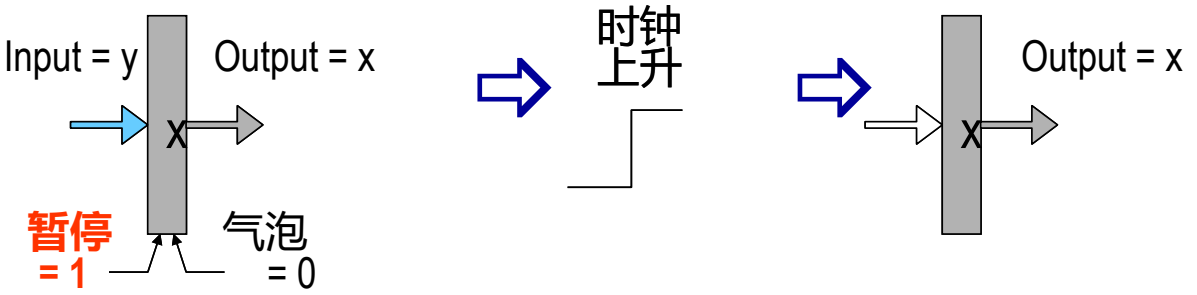
# 流水线寄存器模式

正常

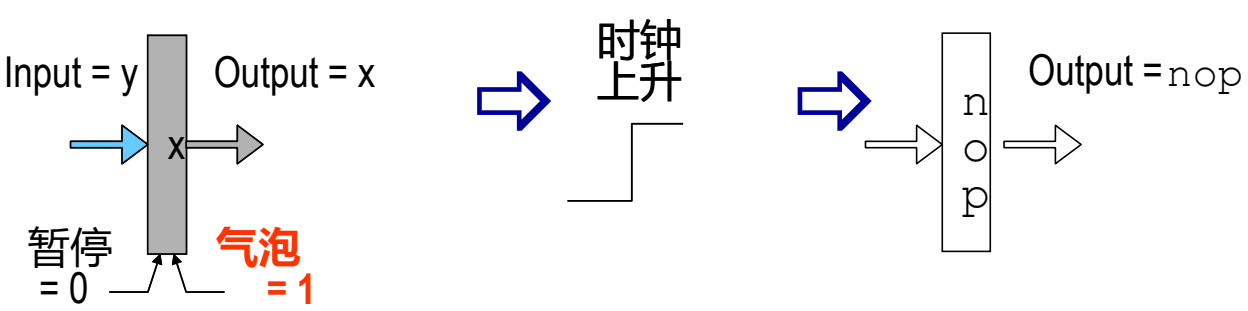



暂停

寄存器写使能暂时失效




气泡



# 数据转发—增加旁路路径解决数据冒险

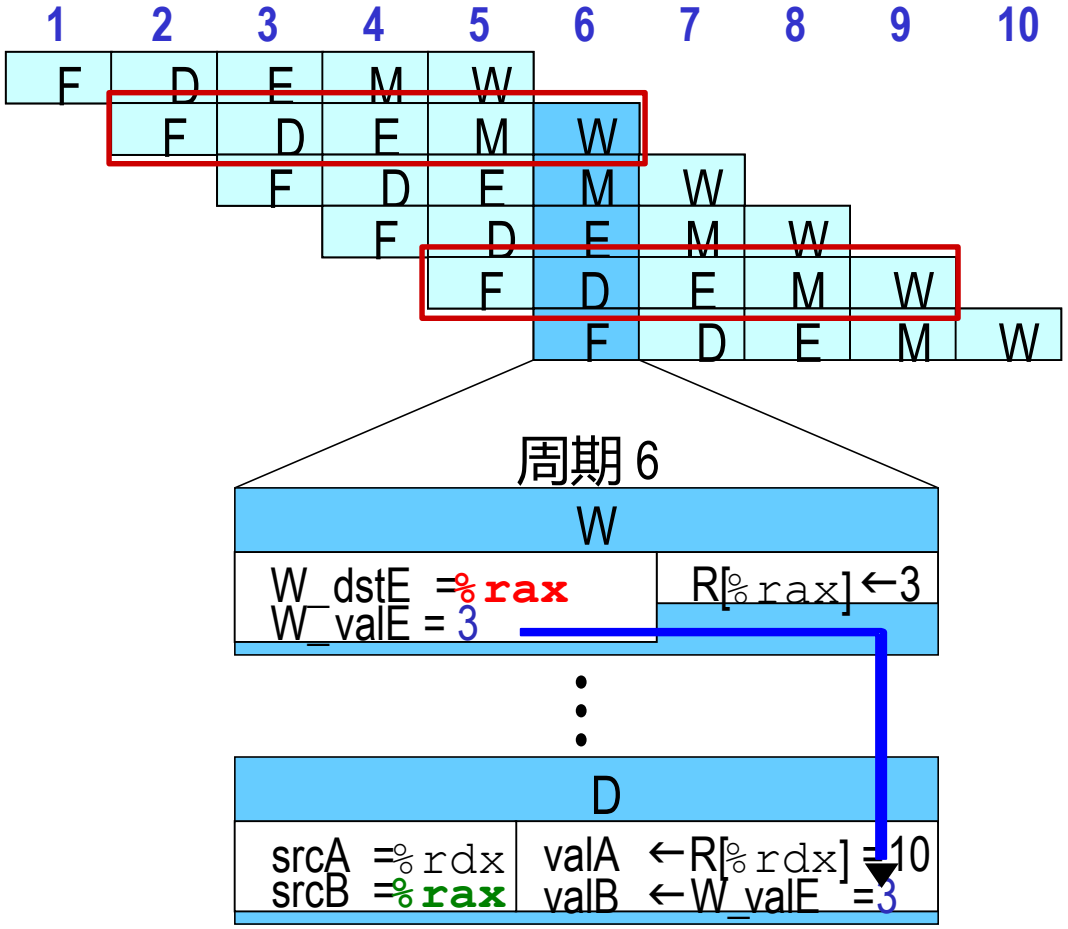
前递也叫转发，也叫旁路

- 理想的流水线
  - 源寄存器的写要在写回阶段才能进行
  - 操作数在译码阶段从寄存器文件中读入
    - 需要在开始阶段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
  - 需要在译码阶段结束时有效（和计组课程这里有区别）
- 转发源： e_valE m_valM M_valE W_valM W_valE
- 转发目的: val_A val_B

# 数据转发示例

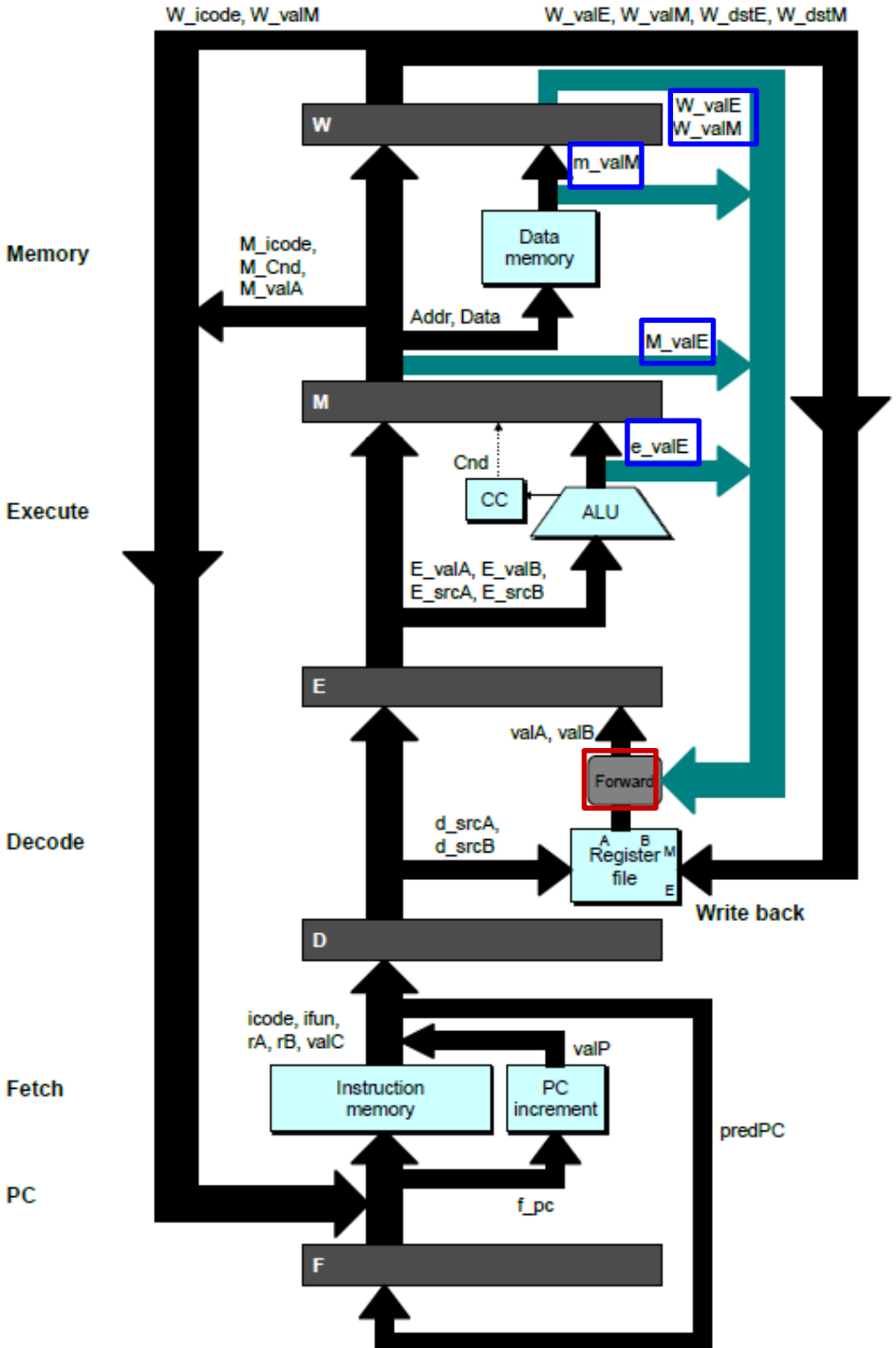
```
# demo-h2.ys
0x000:irmovq $10,%rdx
0x00a:irmovq $3,%rax
0x014:nop
0x015:nop
0x016:addq %rdx,%rax
0x018: halt
```

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



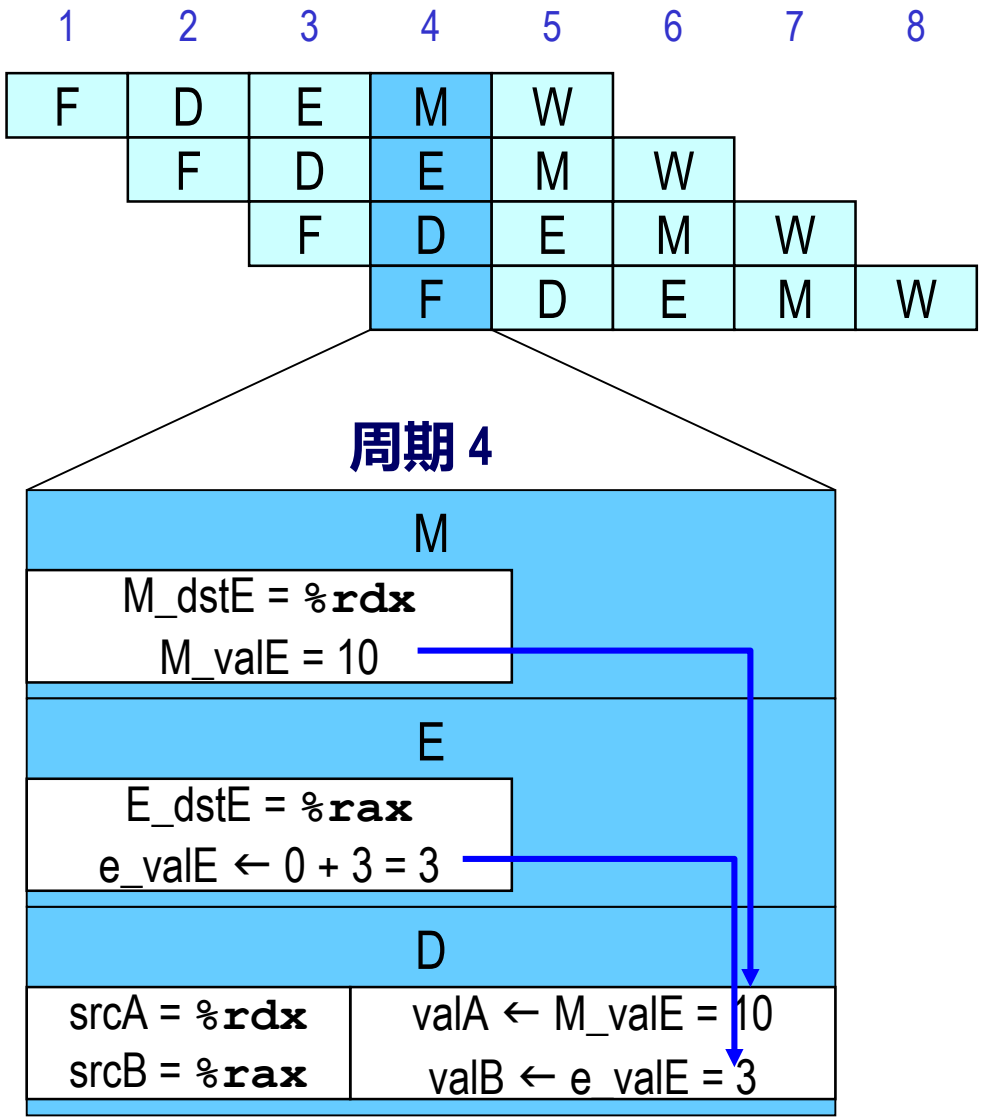
# 旁路路径

- 译码阶段
  - 转发逻辑选中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: valE
  - 访存: valE, valM
  - 写回: valE, valM



# 数据转发示例 #2

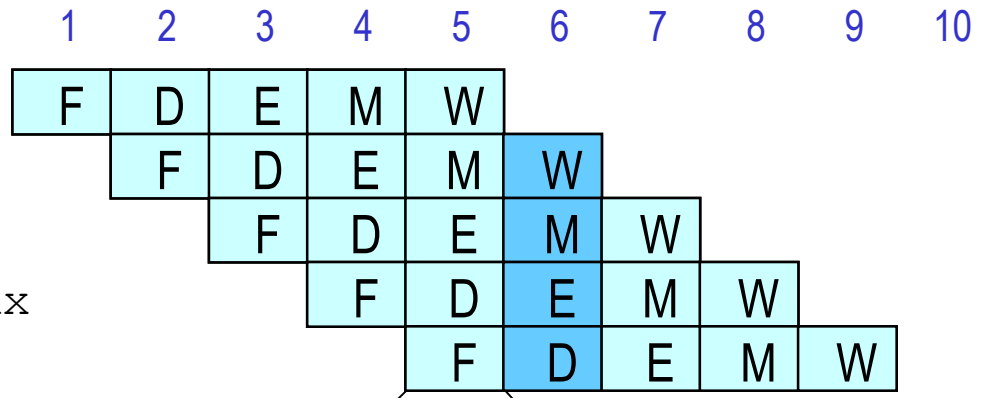
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

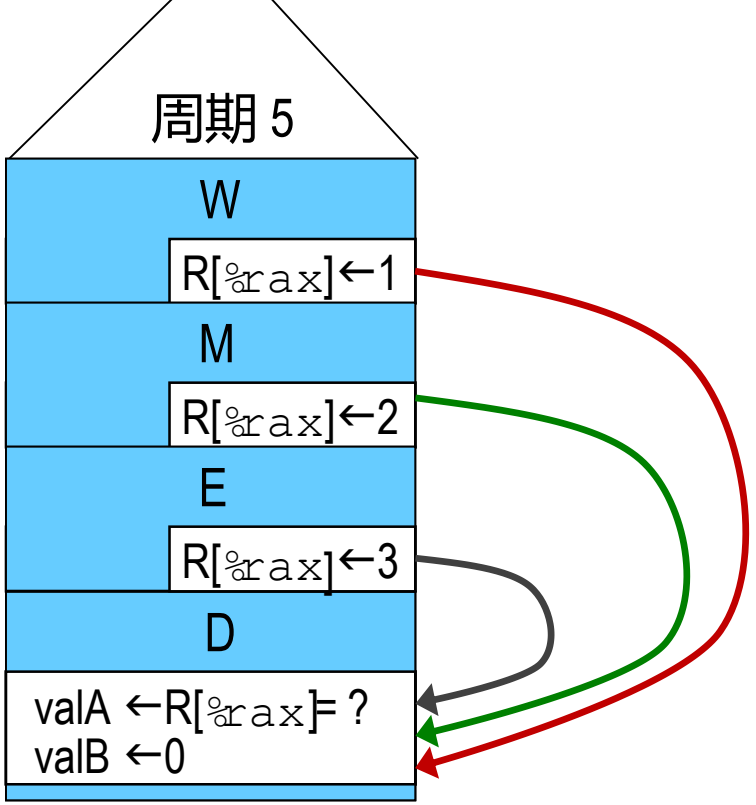



- 寄存器%rdx
  - 由ALU在前一个周期产生
  - 转发自访存阶段作为valA
- 寄存器%rax
  - 值只能由ALU产生
  - 转发自执行阶段作为valB

# 转发优先级

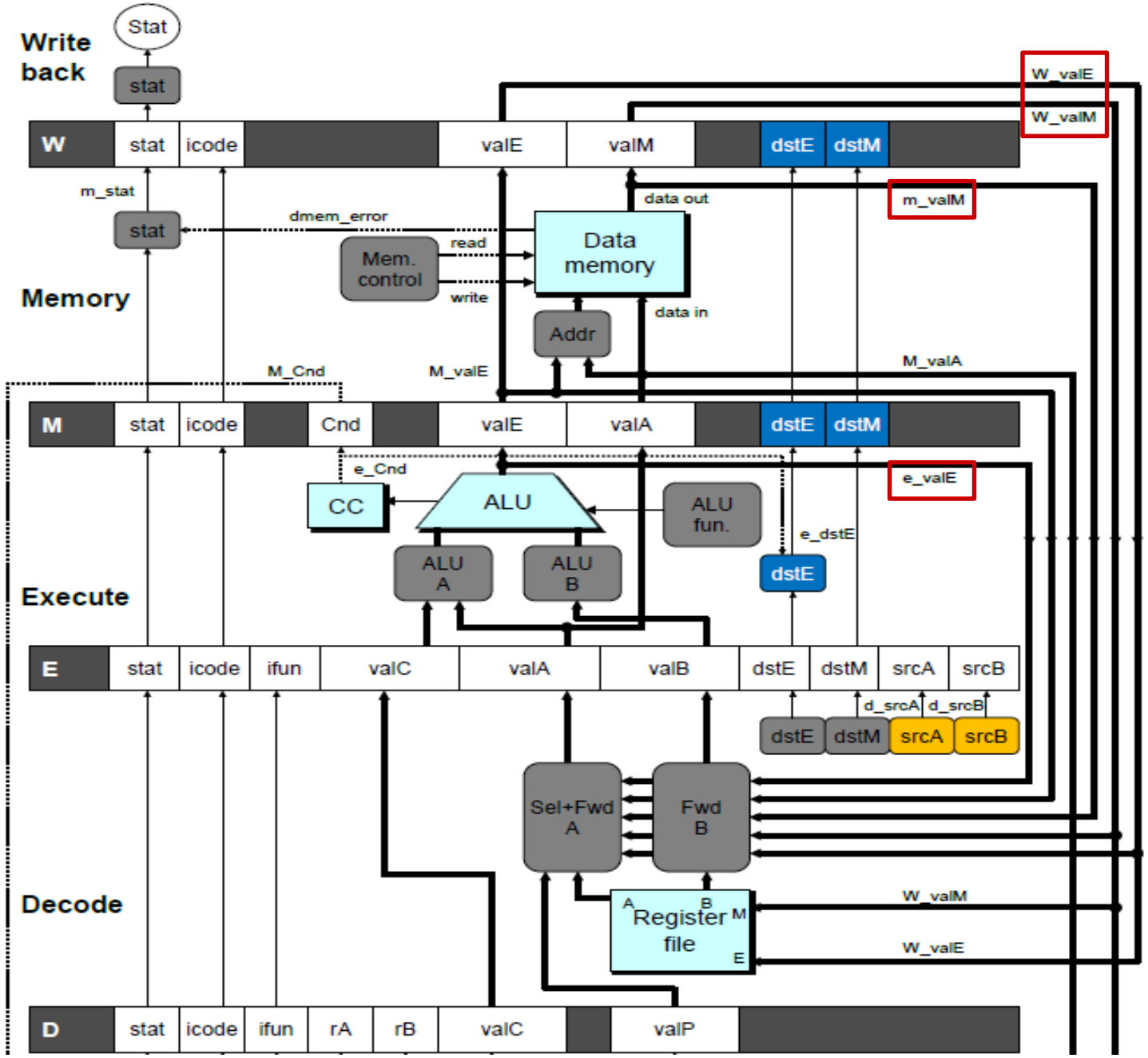
```
# demo-priority.ys
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
  0x020: halt
```




- 多重转发选择
  - 哪一个应该具有最高优先级，选最近的指令转发
  - 匹配串行语义
  - 使用从最早的流水线阶段获取的匹配值

理解“最早的流水线阶段”：流水线阶段：F D E M W。多重转发只会转发阶段E M W的寄存器，所以优先级：E > M >W。（这里的早就是离当前指令最近的意思）

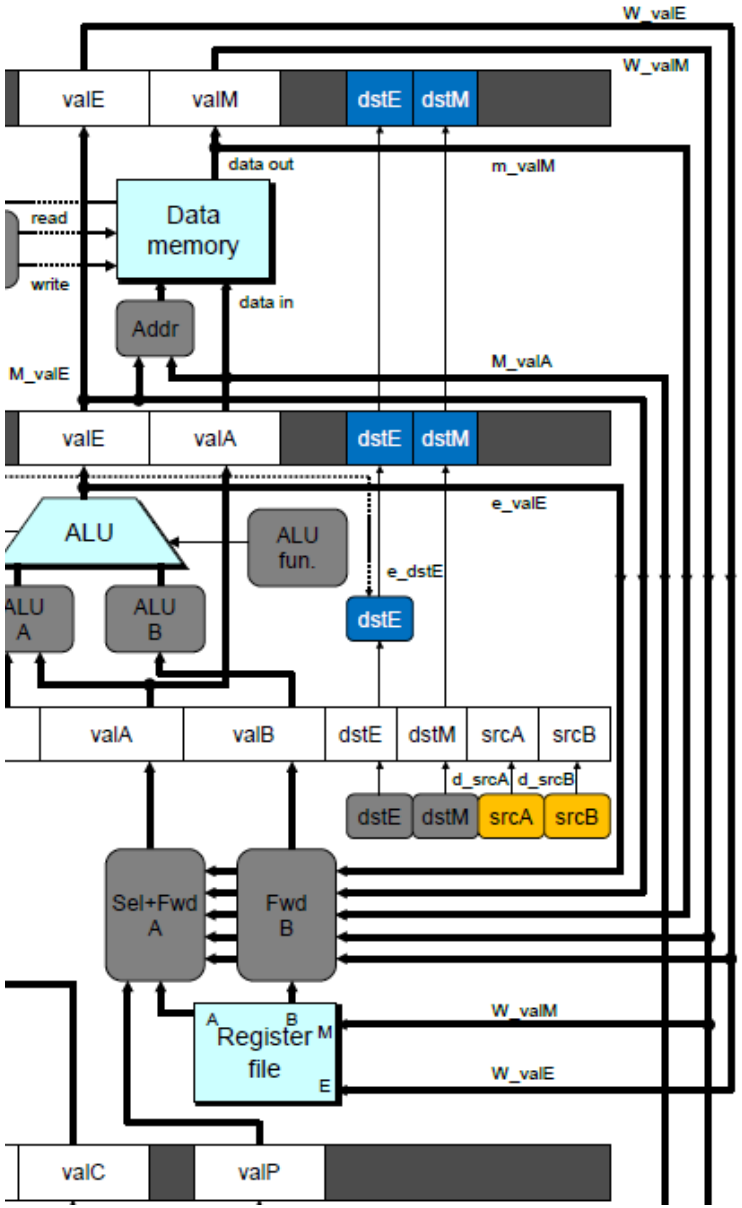



# 实现转发

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

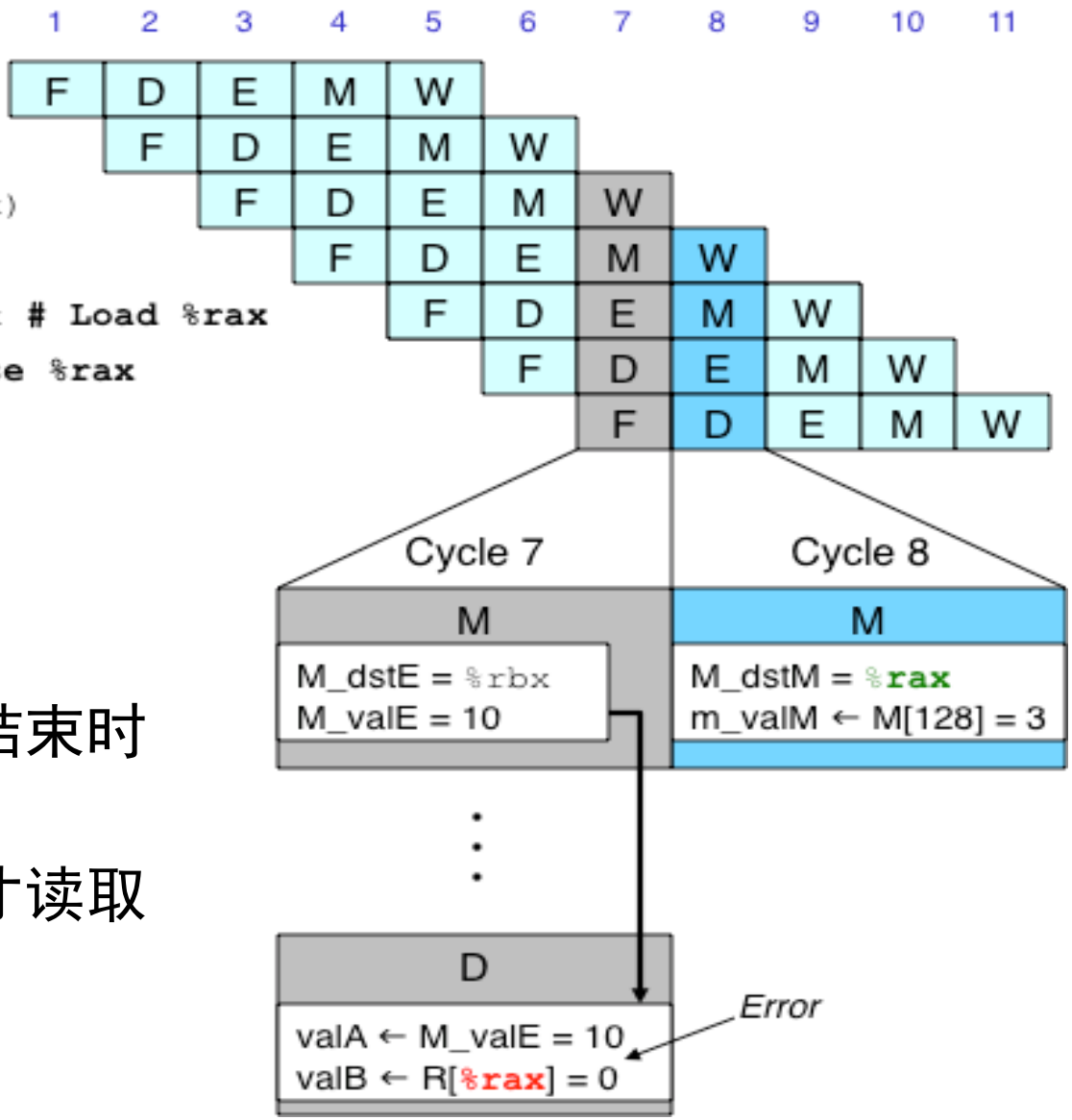
# 实现转发




```
## 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;
];
```

# 只用转发不能解决load-use情况，还需加暂停

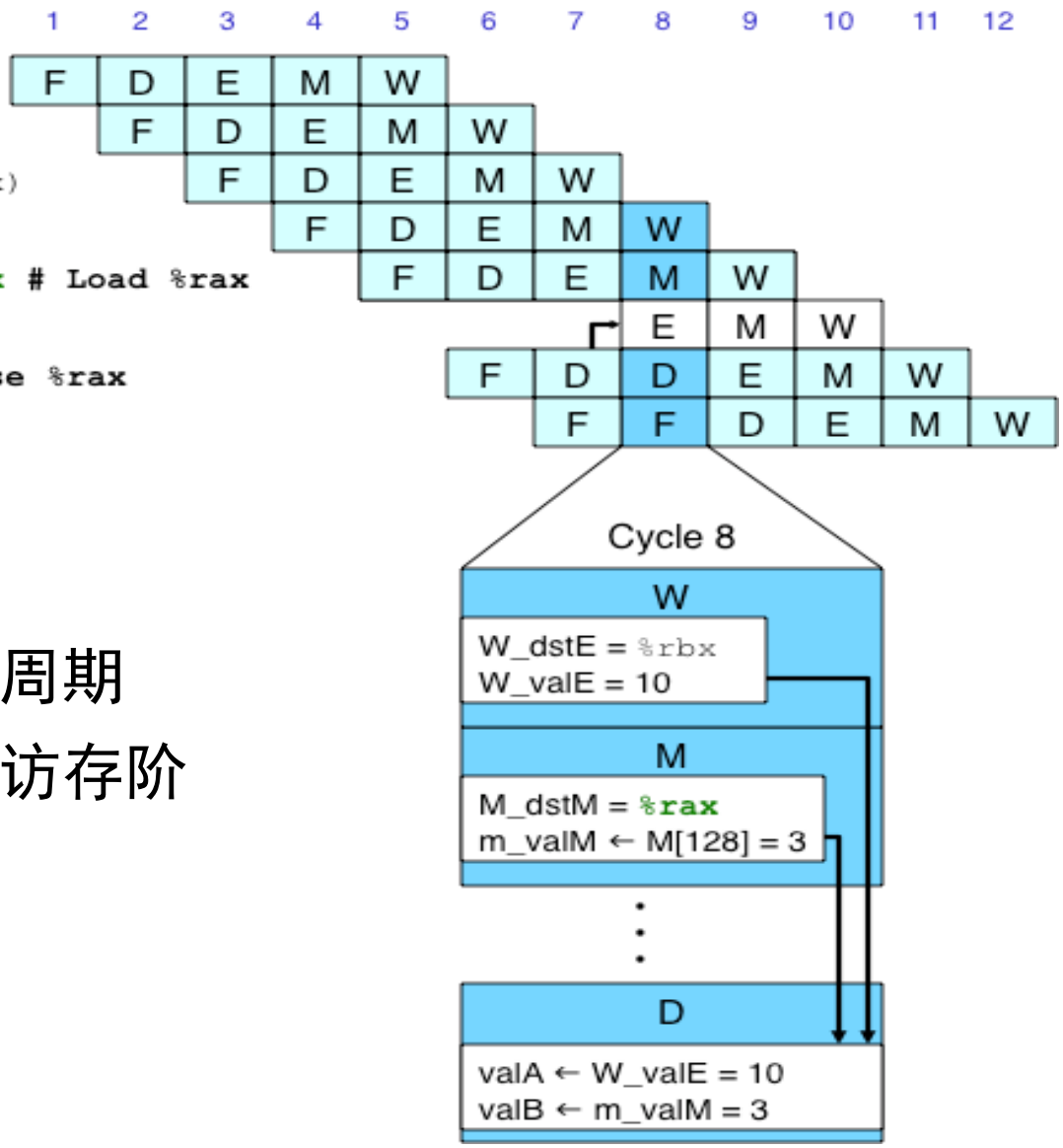
```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```




- 加载-使用 依赖
  - 在周期7译码阶段结束时需要的值
  - 在周期8访存阶段才读取该值

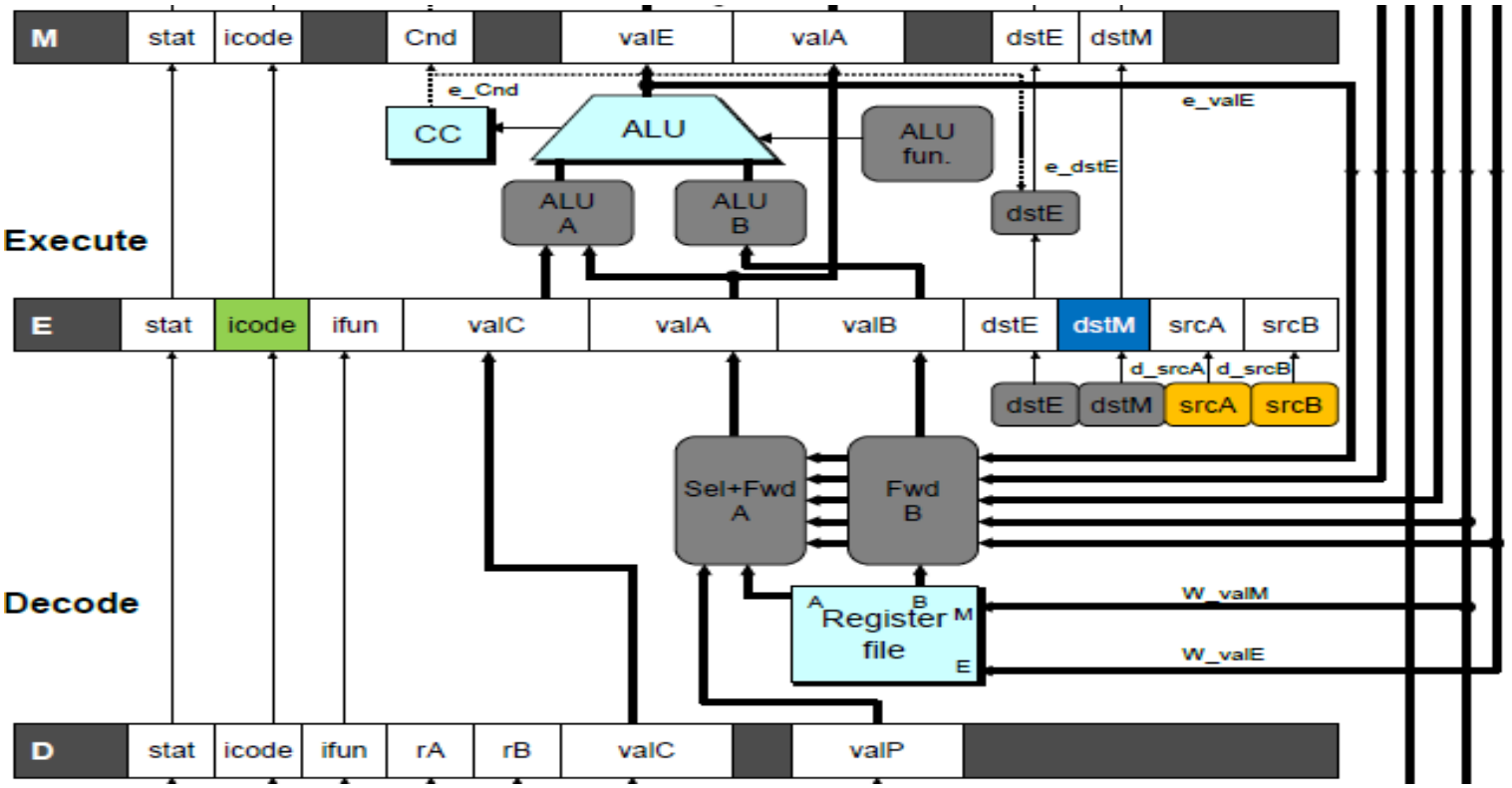
# 解决 加载/使用 冒险的办法：暂停+前递

```
# 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
```




- 使用指令暂停一个周期
- 然后就可以获取从访存阶段转发的加载值

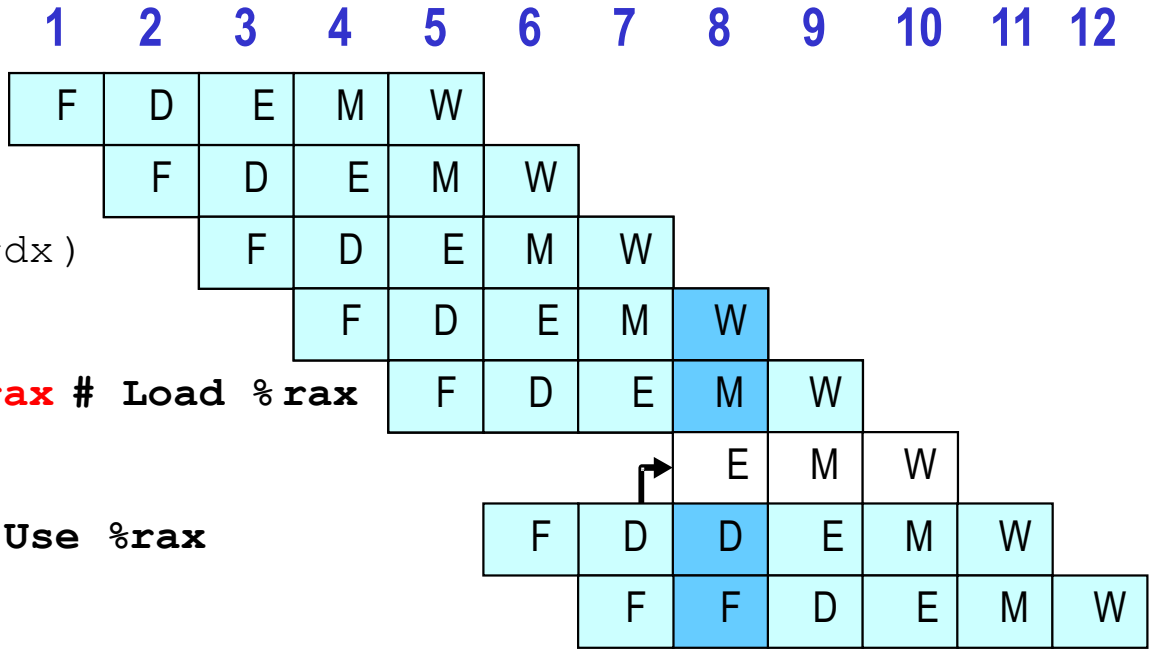
# 检测 加载/使用 冒险




| 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
```



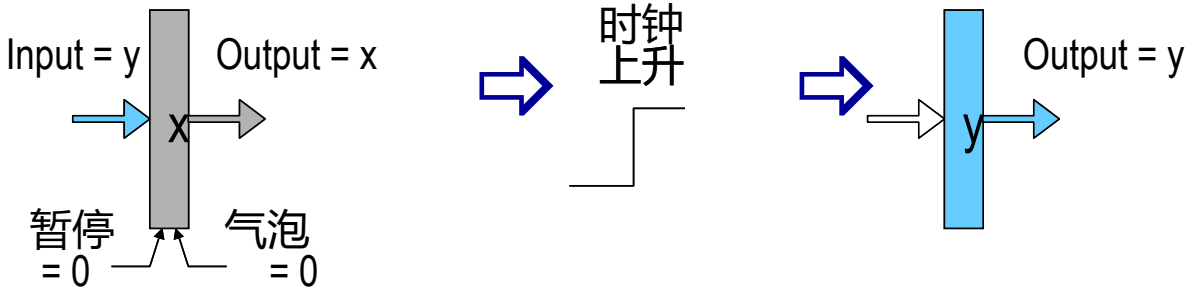

- 将指令暂停在取指和译码阶段
- 在执行阶段注入气泡

| 条件 | F | D | E | M | W |
|---|---|---|---|---|---|
| 加载/使用 冒险 | 暂停 | 暂停 | 气泡 | 正常 | 正常 |

备注：从load类指令角度看，需要对F、D暂停，对E进行气泡化处理，即对流水线寄存器做控制；本例中，对第7周期的流水线寄存器做控制，产生了8周期的结果。

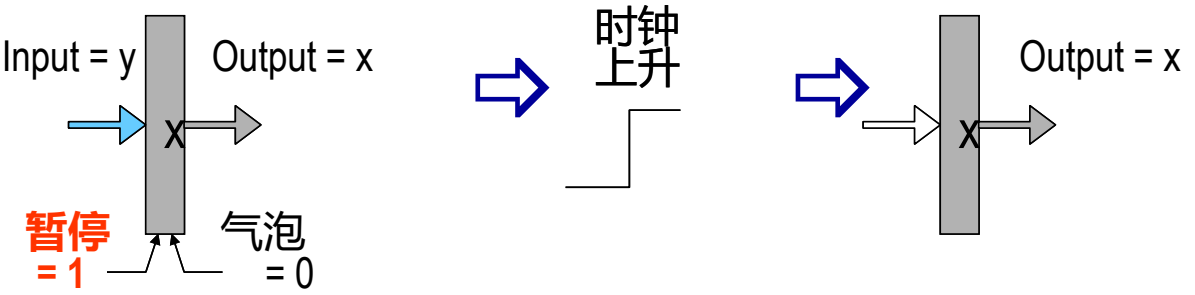
# 流水线寄存器模式

正常

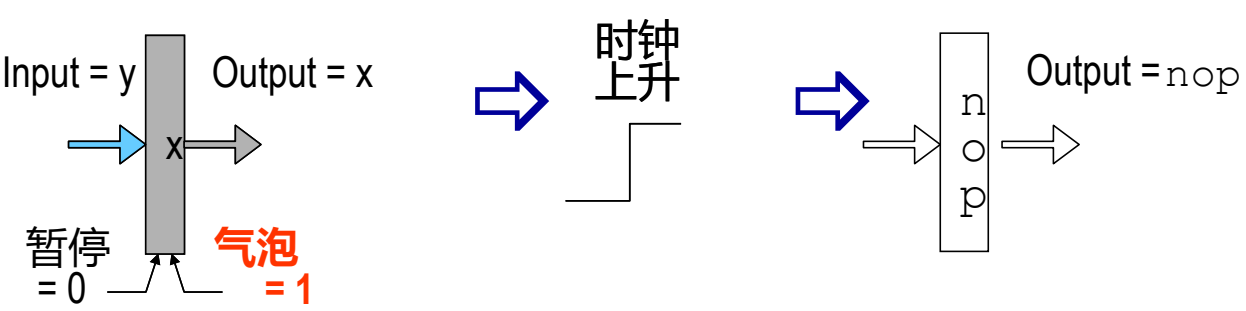



暂停

寄存器写使能暂时失效




气泡



# 分支预测错误示例

**demo-j.ys**

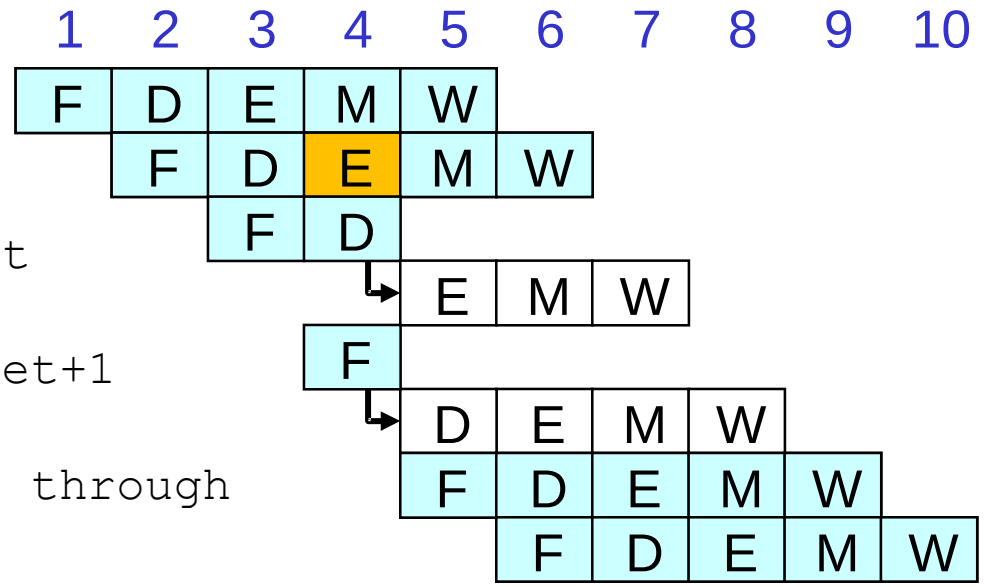
```
0x000:      xorq %rax,%rax
0x002:      jne  t                  # Not taken
0x00b:      irmovq $1, %rax         # Fall through
0x015:      nop
0x016:      nop
0x017:      nop
0x018:      halt
0x019:  t:  irmovq $3, %rdx         # Target
0x023:      irmovq $4, %rcx         # Should not execute
0x02d: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

- 只能执行最早的7条指令

# 处理预测错误

**#demo-j.ys**

```
0x000: xorq %rax, %rax
0x002: jne target #Not Taken
0x016: irmovq $3, %rdx #target
       bubble
0x020: irmovq $4, %rcx # target+1
       bubble
0x00b: irmovq $1, %rax # Fall through
0x015: halt
```



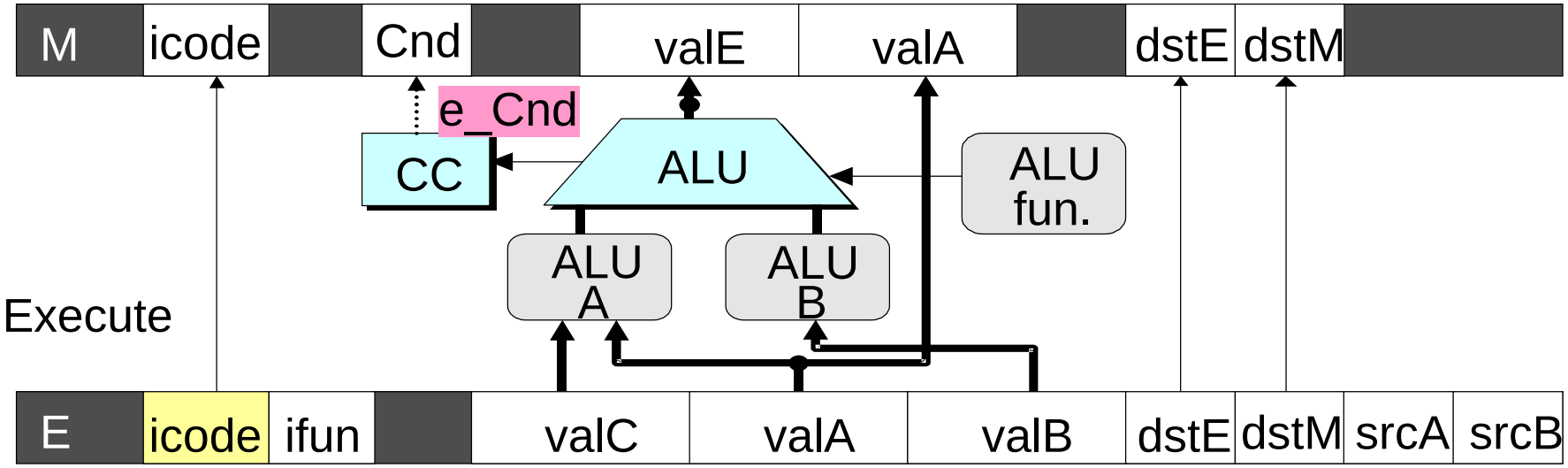

## 作为预测分支

- 取出 2 条目标指令

## 当预测错误时取消

- **在执行阶段检测到未选择该分支**
- 在紧跟的指令周期中，将处于执行和译码阶段的指令用气泡替换掉
- **此时没有出现副作用**

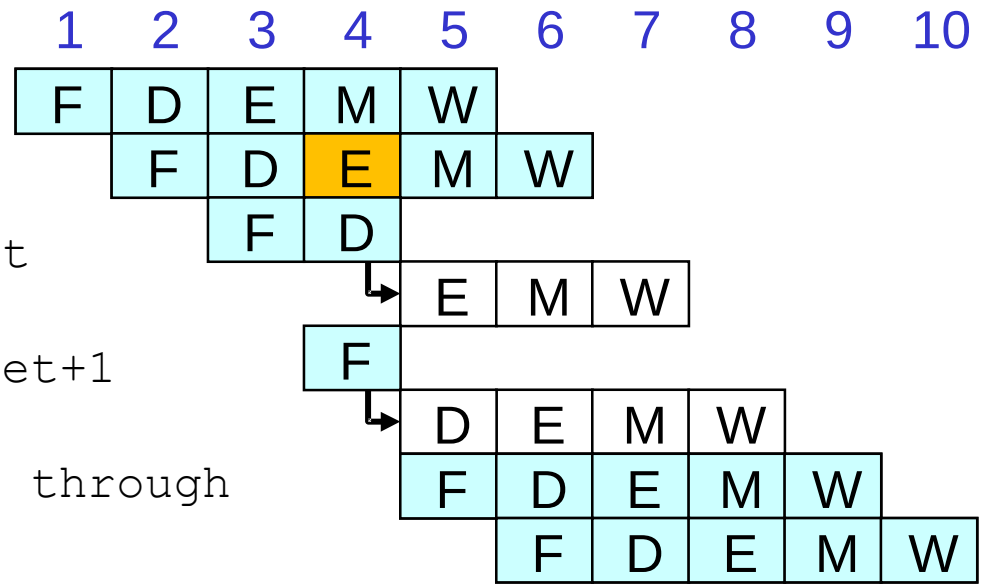
# 检测分支预测错误




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

# 预测错误的控制

```
#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
```




| 条件 | F | D | E | M | W |
|---|---|---|---|---|---|
| 分支预测错误 | 正常 | 气泡 | 气泡 | 正常 | 正常 |

备注：从Jxx条件跳转指令（失败情况下）角度看，需要对D、E进行气泡化处理，即对流水线寄存器做控制；本例中，对第4周期的流水线寄存器做控制，产生了5周期的结果。

# Return示例

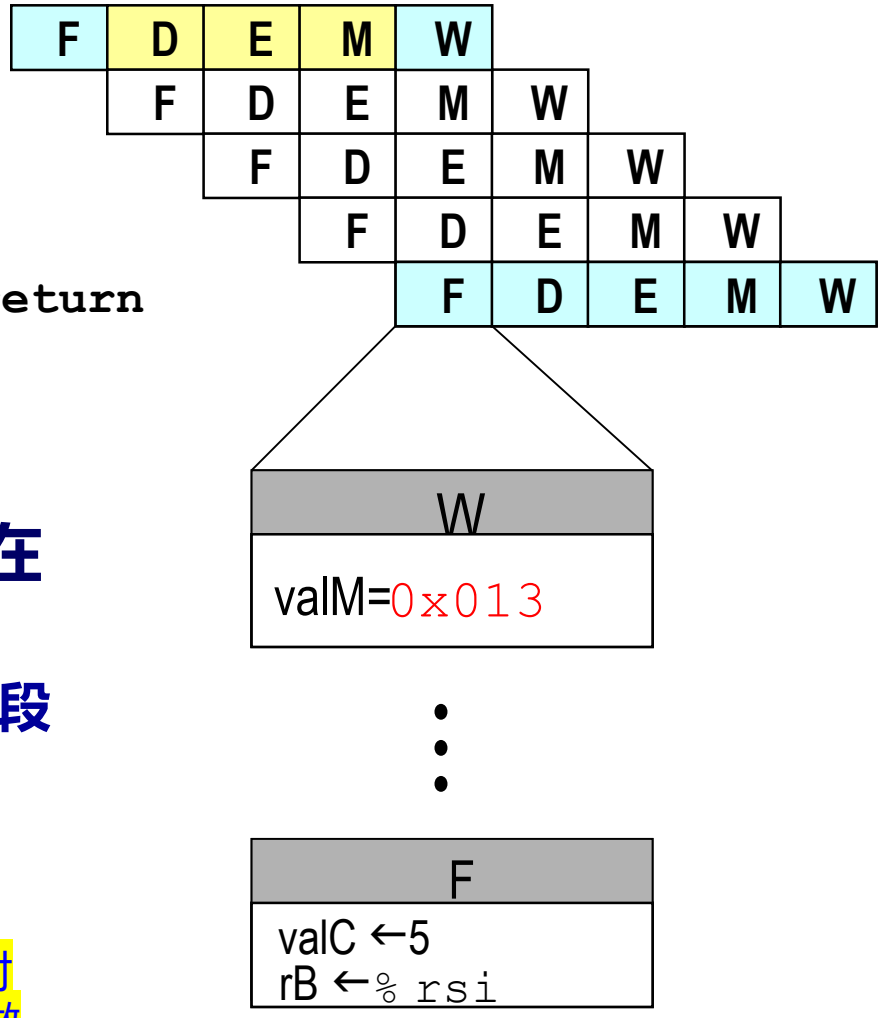
demo-retb.ys

```
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
```

- 之前执行了3条额外的指令

# 正确的Return示例

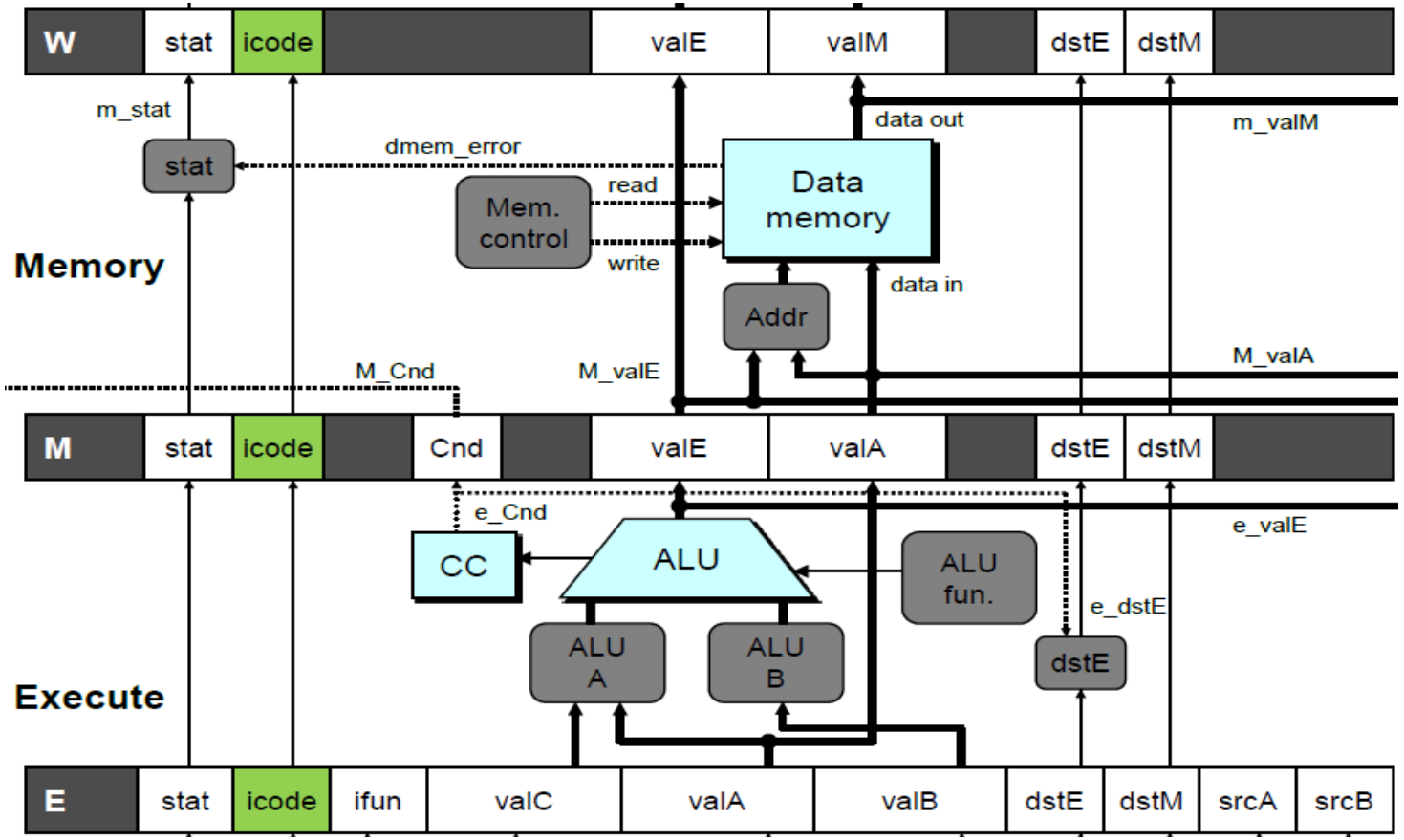
```
1: 0x026: ret
2:        bubble
3:        bubble
4:        bubble
5: 0x013: irmovq $5, %rsi #return
```




- 当ret经过流水线时，暂停在取指阶段
  - 当处于译码、执行和访存阶段
- 在译码阶段注入气泡
- 当到达写回阶段释放暂停

注释：释放暂停指的是不继续bubble，对ret指令只需暂停三个周期即可，不过多做暂停处理了

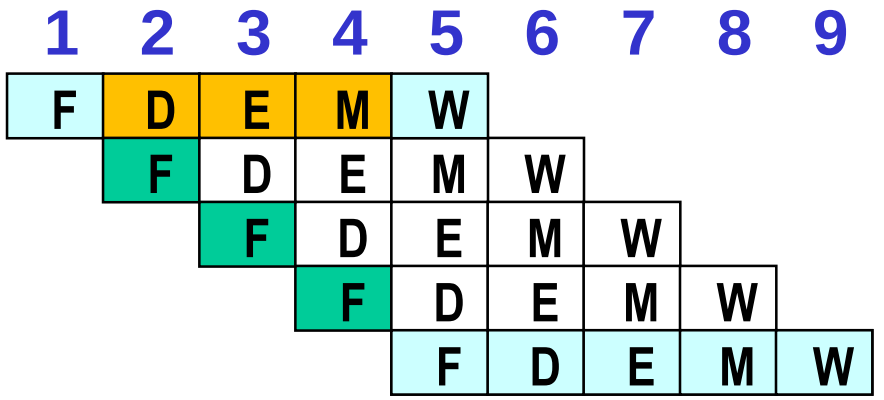
# 检测Return




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

# Return的控制

```
 # demo_retb

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




| 条件 | F | D | E | M | W |
|---|---|---|---|---|---|
| 处理 ret | 暂停 | 气泡 | 正常 | 正常 | 正常 |

备注：如果在D、E、M阶段分别检测出Icode是ret指令，都需要立刻对F流水线寄存器做暂停处理、对D流水线寄存器做气泡化处理，每一个当前做都会直接影响到下个阶段的结果。上面ret这个例子2、3、4三个阶段都做处理，3、4、5则表现出了处理结果。

# 解决三种冒险：检测+处理动作

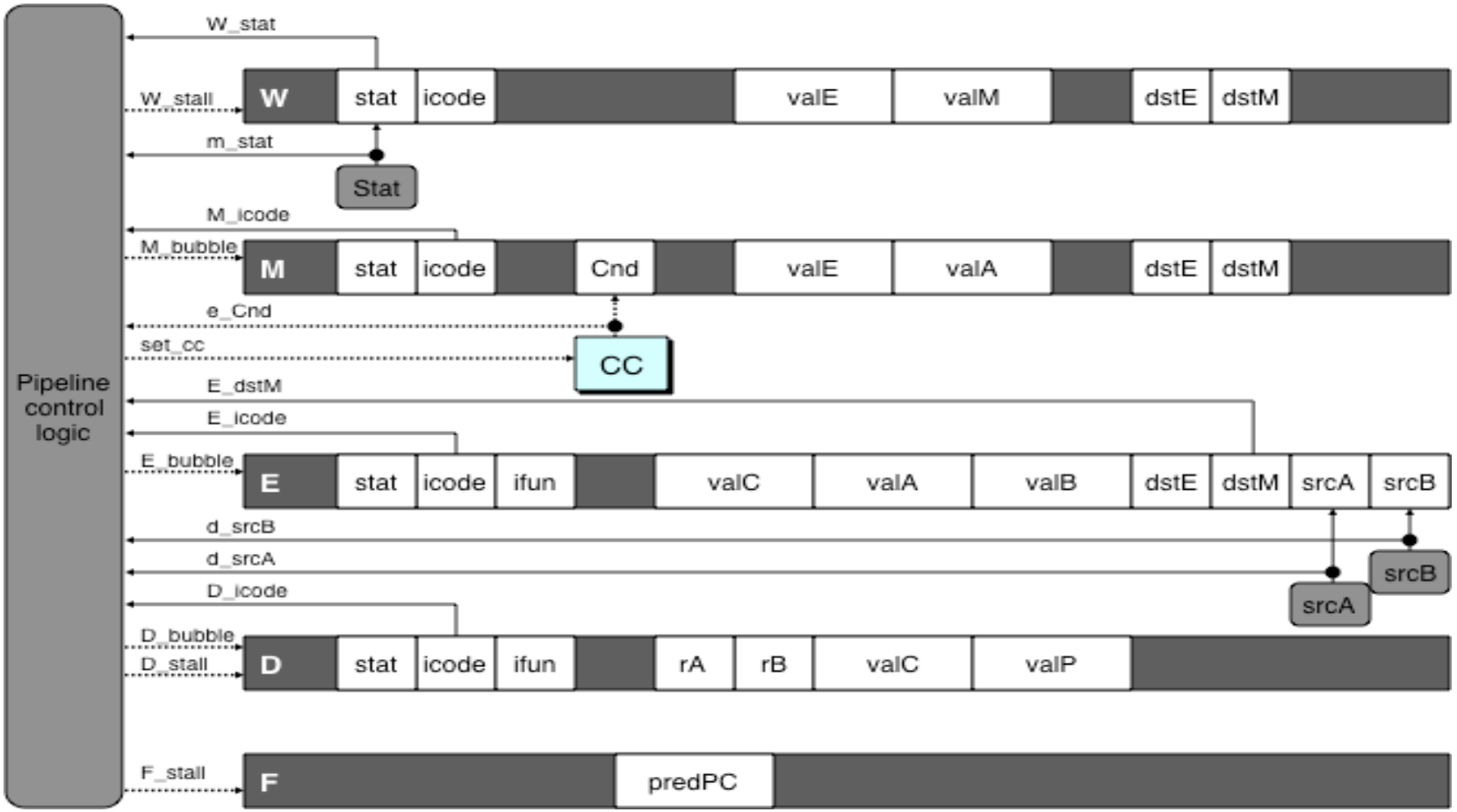
- 检测

| 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(在下一个周期)

| 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开始的时候

# 流水线控制的初始版本

- 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(在下一个周期)

| 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 |

```
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 };
```

# 流水线控制的初始版本

- 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 |

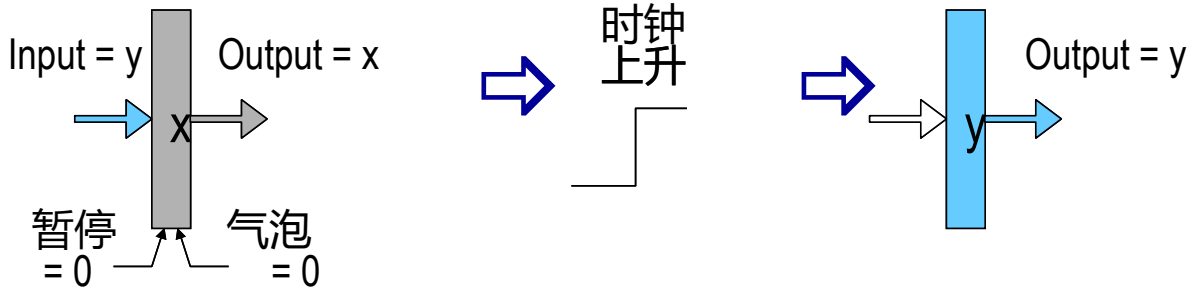
- 动作(在下一个周期)

| 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 |

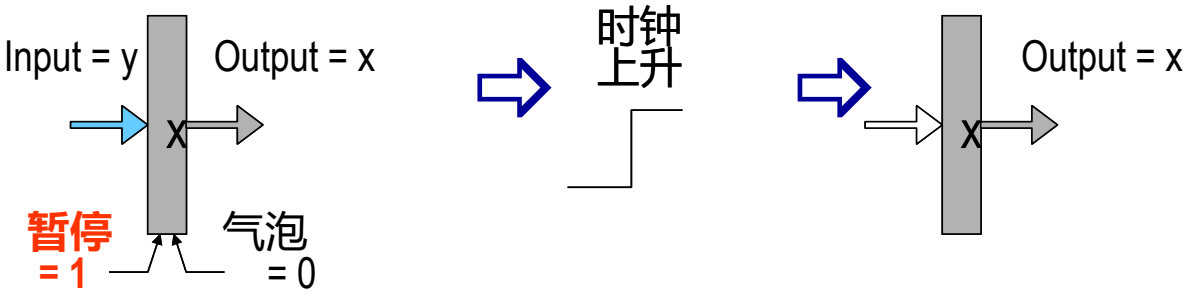
```
bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # stalling at fetch while ret passes
      through pipeline
     IRET in { D_icode, E_icode, M_icode };

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 };
```

# 流水线寄存器模式（优先级）

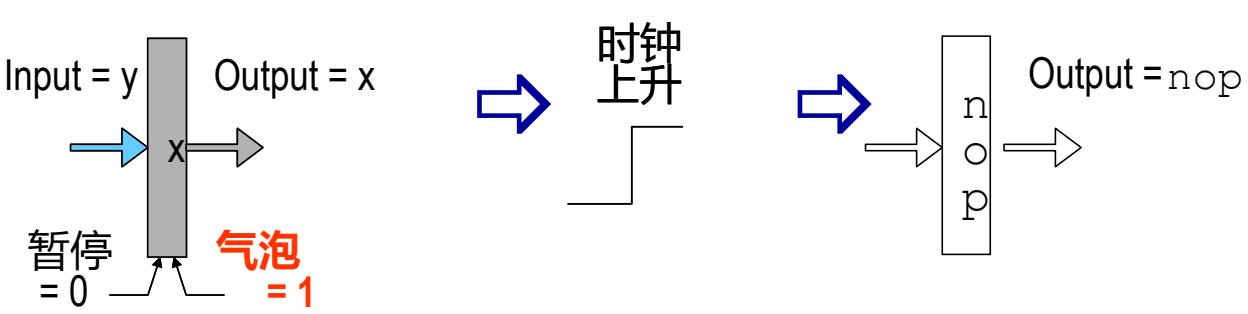
正常




暂停




气泡



# 4-5 习题

**1. Y86-64流水线CPU中的冒险的种类与处理方法。**

**(1)数据冒险：指令使用寄存器R为目的，瞬时之后使用R寄存器为源。**
**处理方法有：**
**①暂停：通过在执行阶段插入气泡（bubble/nop），使得当前指令执行暂停在译码阶段；**
**②数据转发：增加valM/valE的旁路路径，直接送到译码阶段；**

**(2)加载使用冒险：指令暂停在取指和译码阶段,在执行阶段插入气泡（bubble/nop）**

**(3)控制冒险：分支预测错误：在条件为真的地址target处的两条指令分别插入1个bubble。ret：在ret后插入3个bubble。**

# 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
  - **检测到分支预测错误时取消指令**
    - 两个时钟周期被浪费
  - **暂停在取指阶段，直到ret通过流水线**
    - 三个时钟周期被浪费

*Enjoy!*