第四章 处理器体系结构
——流水线实现高级技术

教 师： 史先俊
计算机科学与技术学院
哈尔滨工业大学

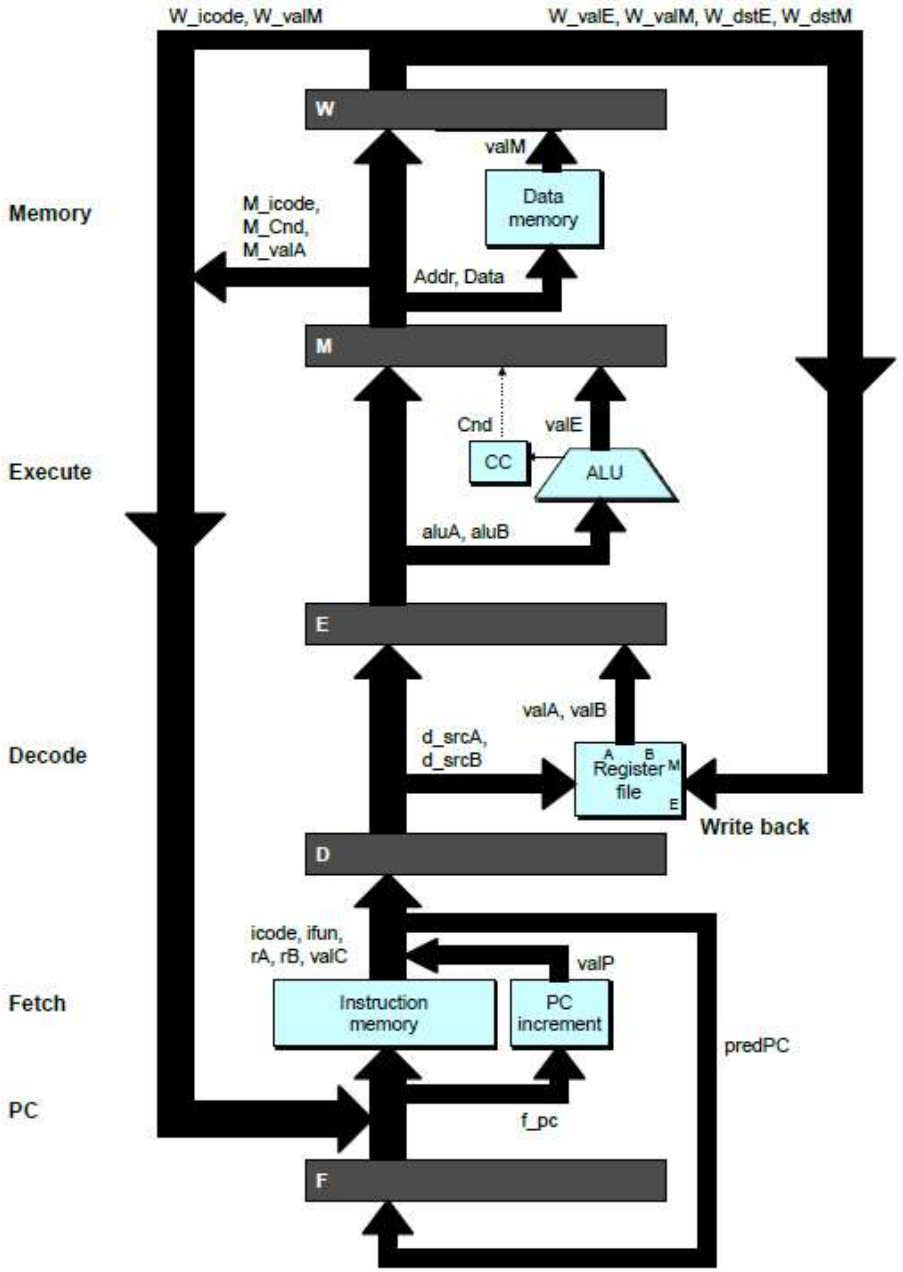
概述

使流水线处理器工作！

- 数据冒险
  - 指令使用寄存器R为目的，瞬时之后使用R寄存器为源
  - 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
    - 理想流水线执行两条额外的指令
  - 从ret指令中获得返回地址
    - 理想流水线执行三条额外的指令
- **确保它确实有效的工作**
  - 如果多种特殊情况同时发生将会怎样?

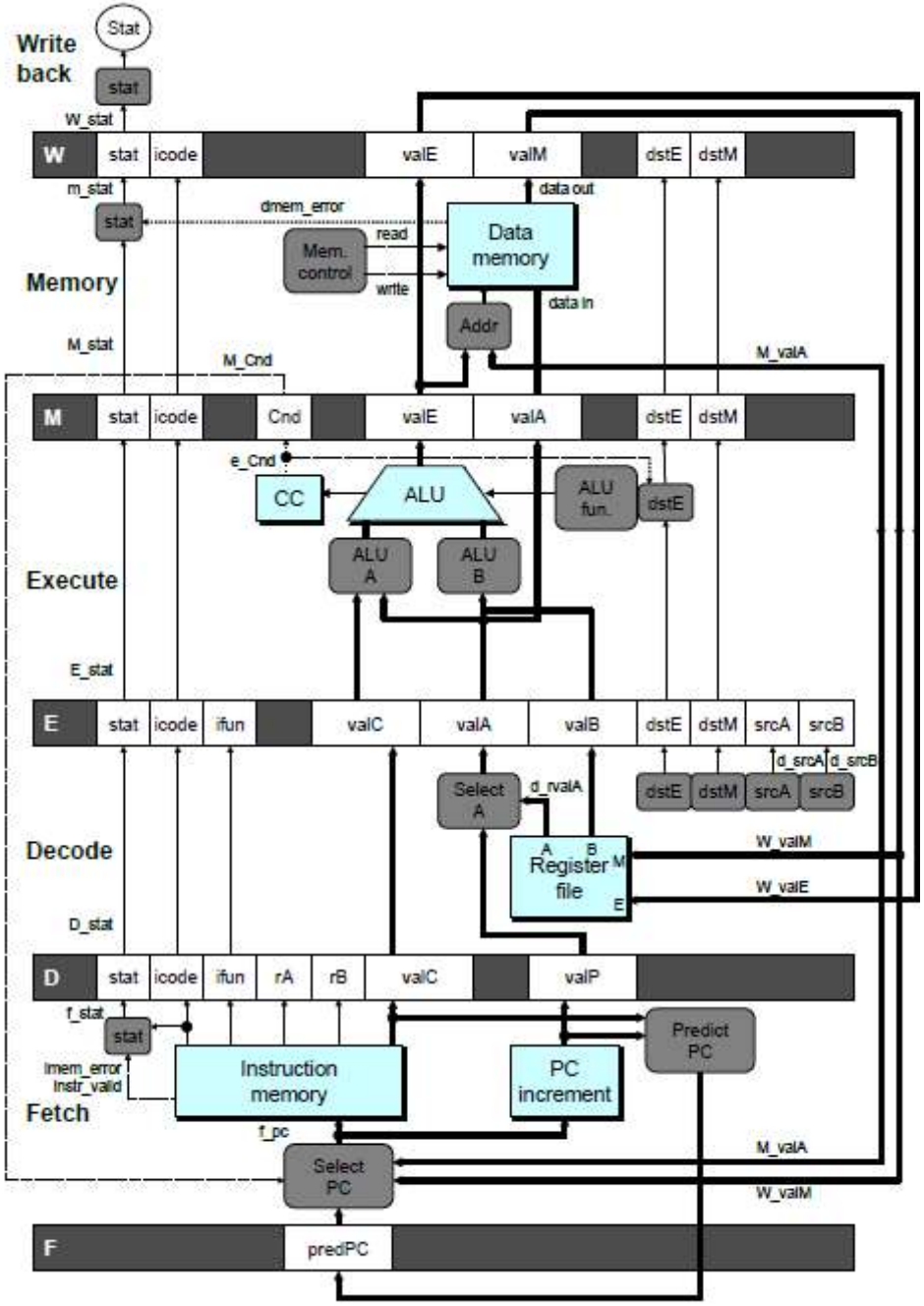
流水阶段

- 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



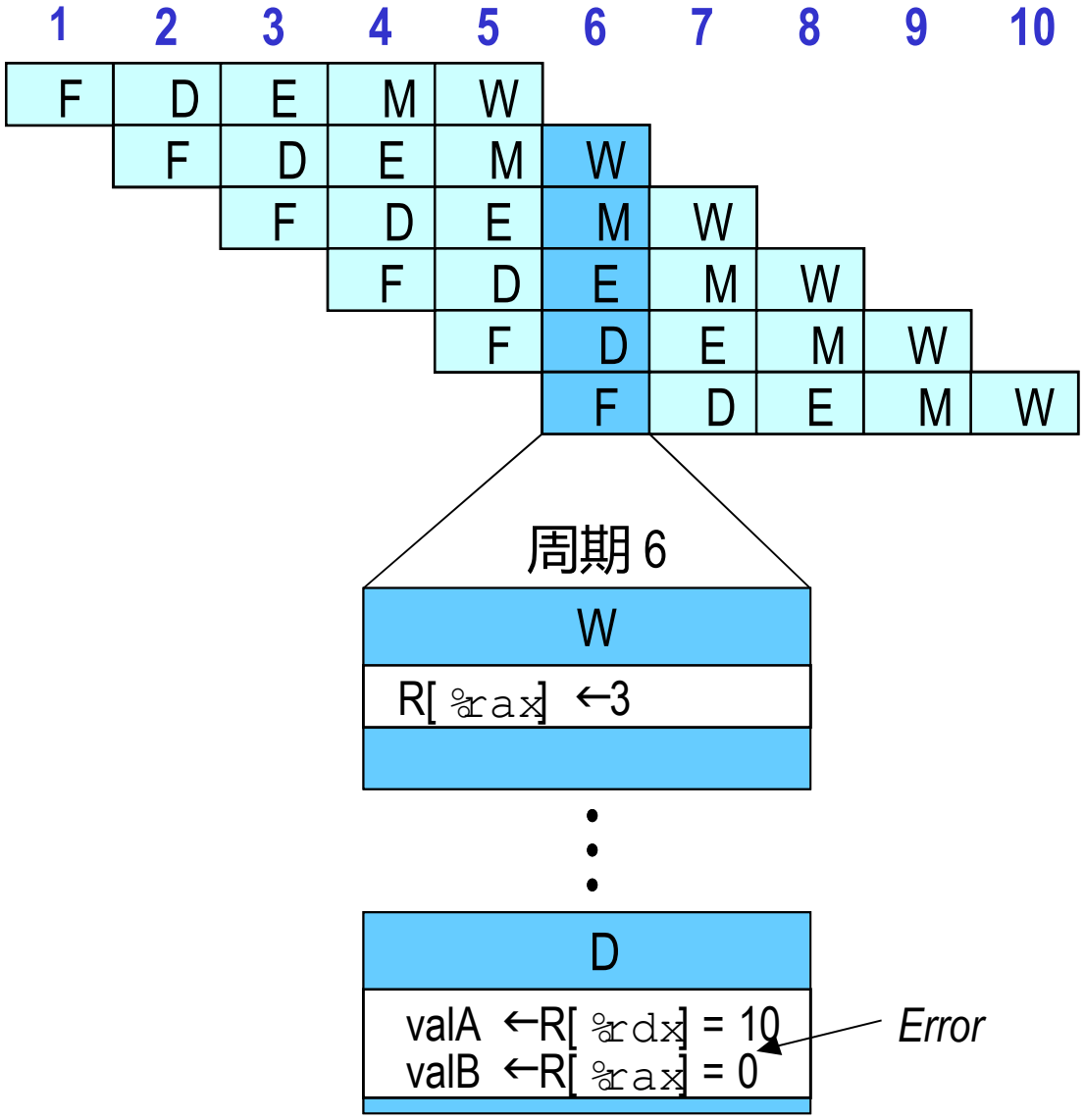
PIPE- 硬件

- 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



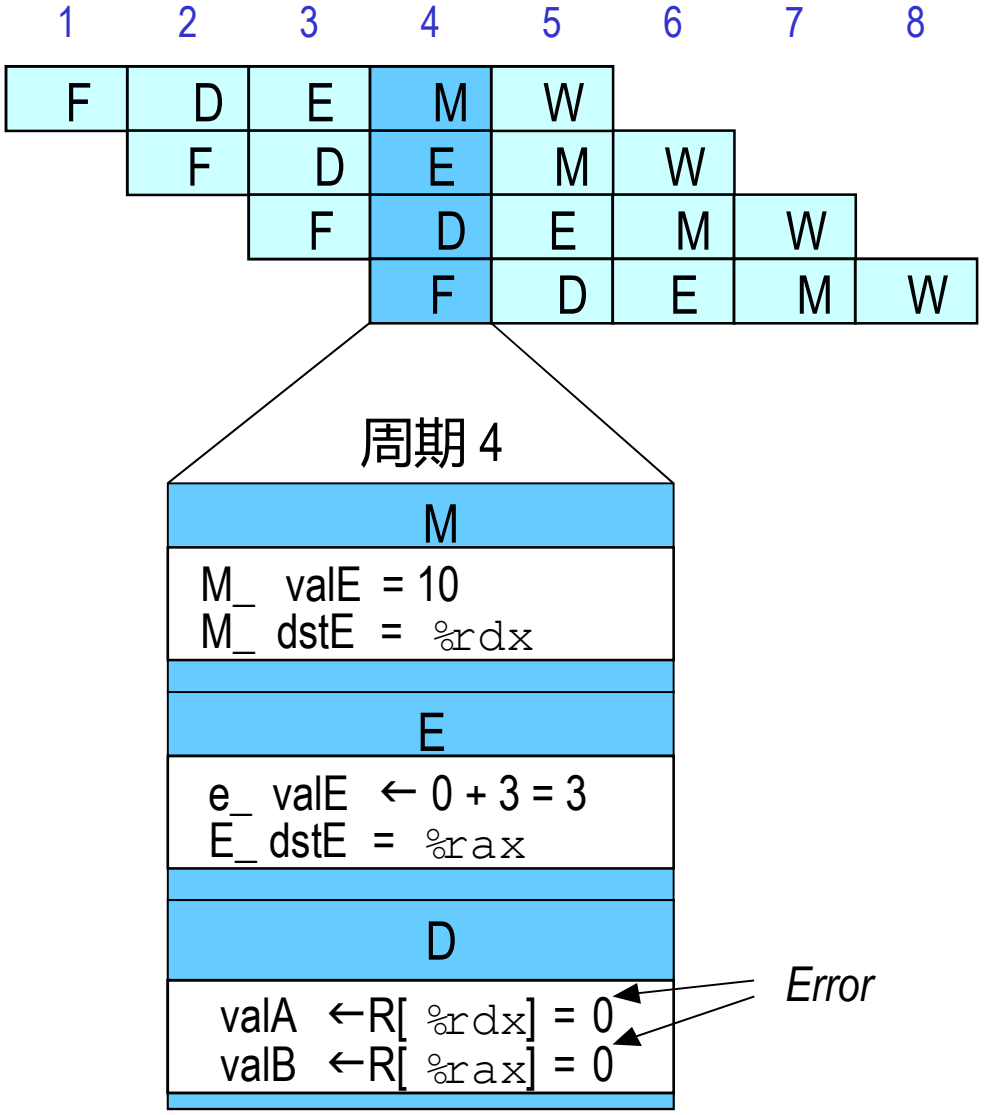
数据相关: 两条Nop指令

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



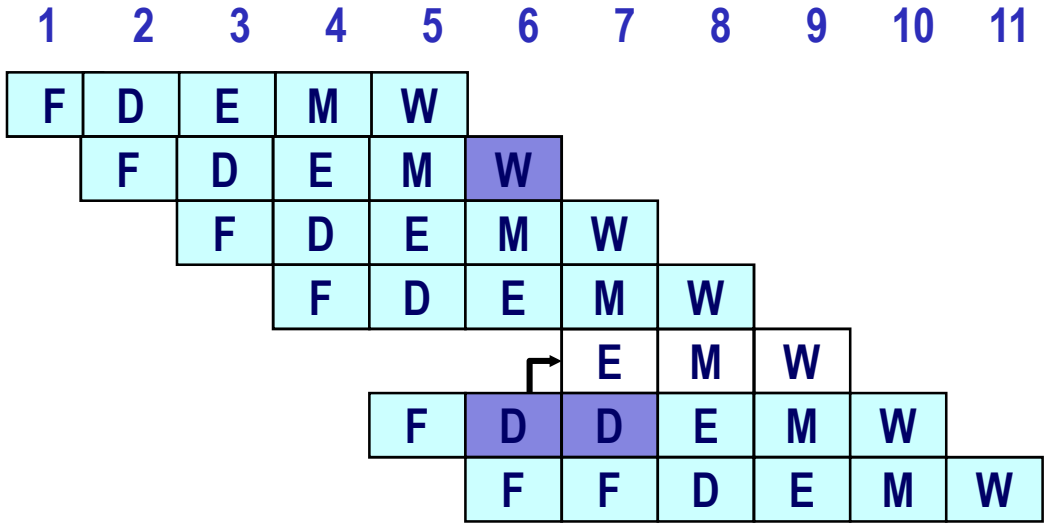
数据相关: 无Nop指令

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



数据相关的暂停

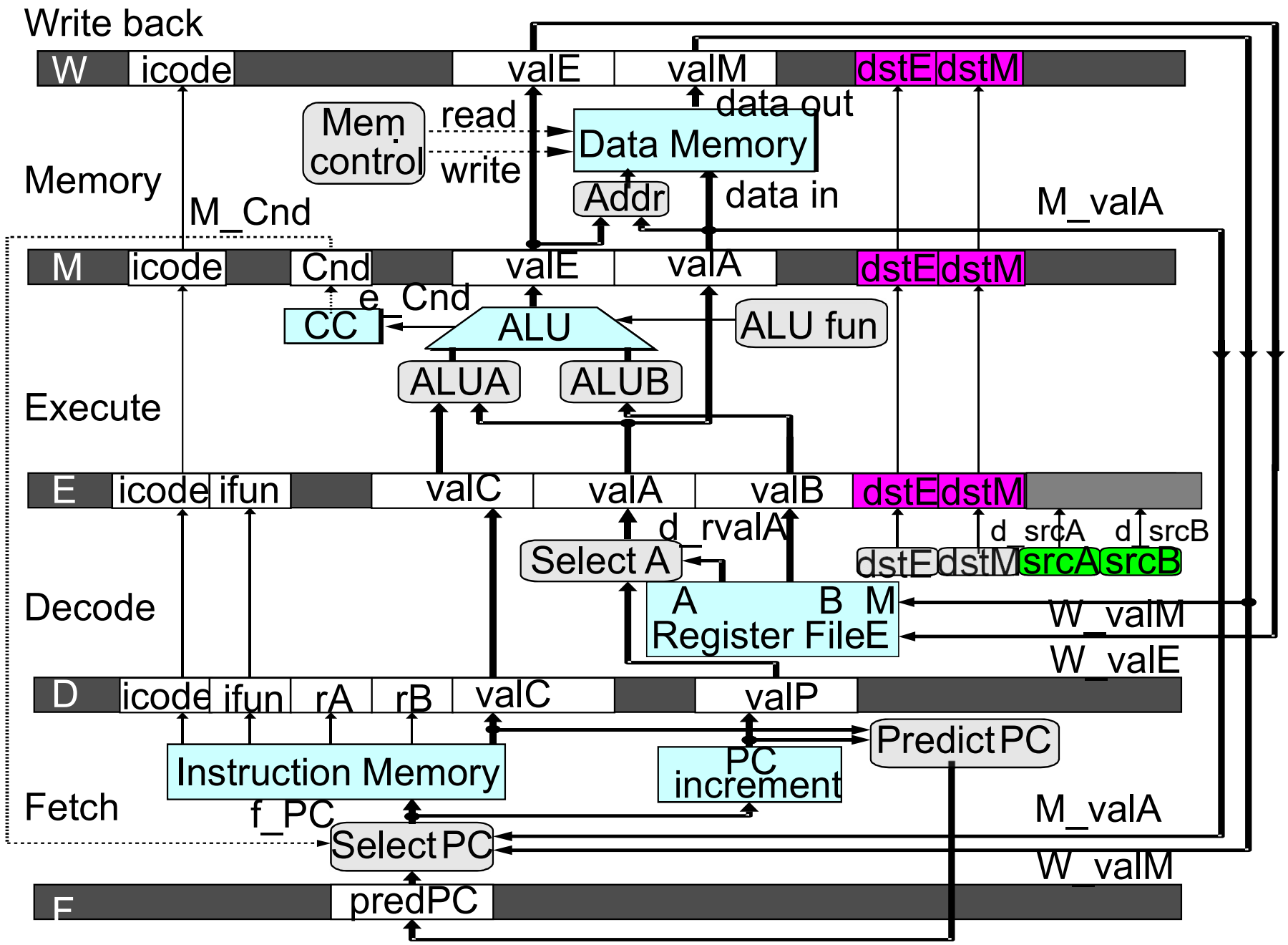
```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```



- 如果一条指令紧跟写寄存器指令，则将该指令执行速度放慢
- 将指令阻塞在译码阶段
- 在指令执行阶段动态插入nop

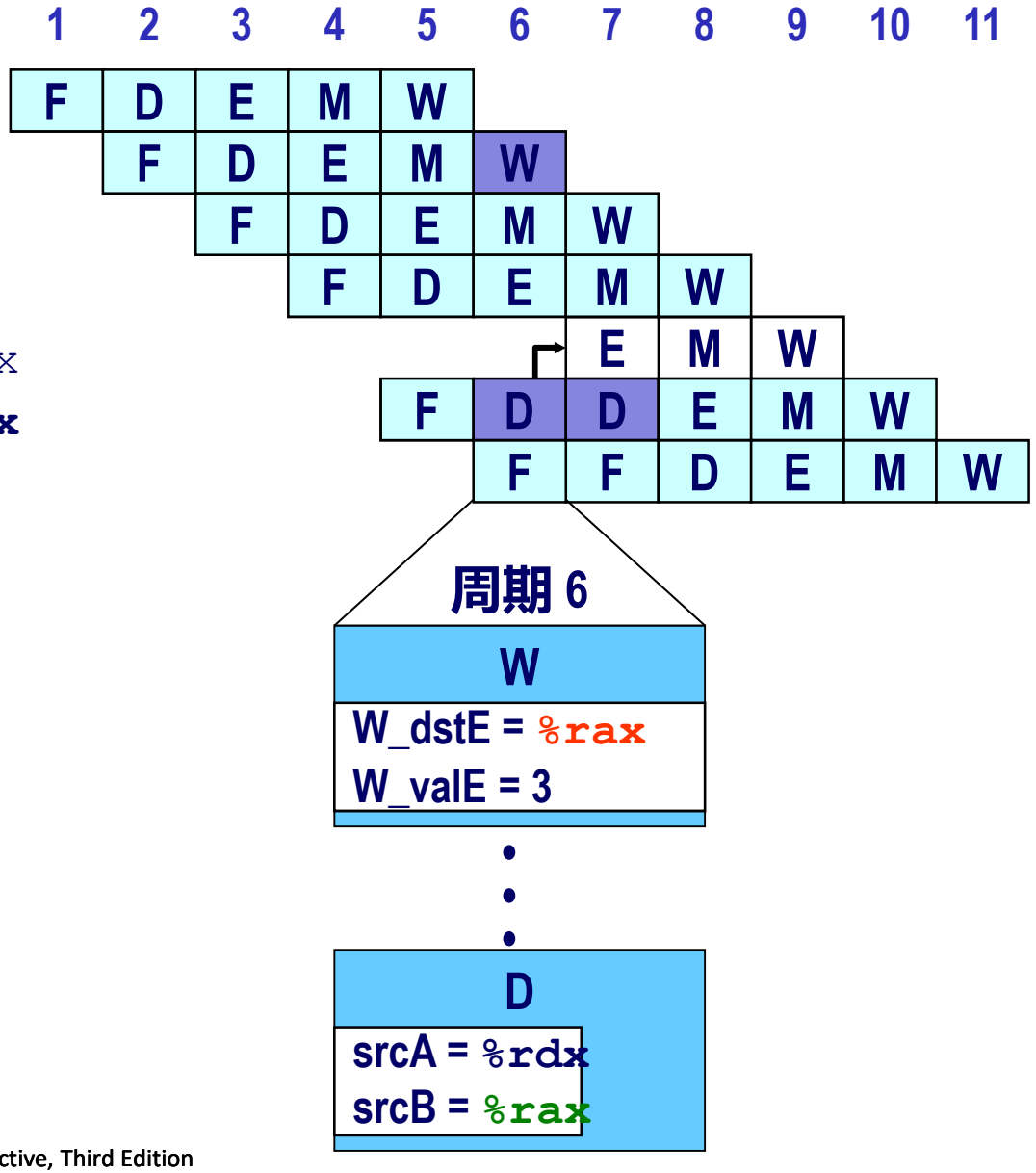
暂停条件

- 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
W_valM
W_valE
Decode
D
icode
ifun
rA
rB
valC
valP
Predict PC
Instruction Memory
PC increment
Fetch
f_PC
M_valA
Select PC
W_valM
F
predPC

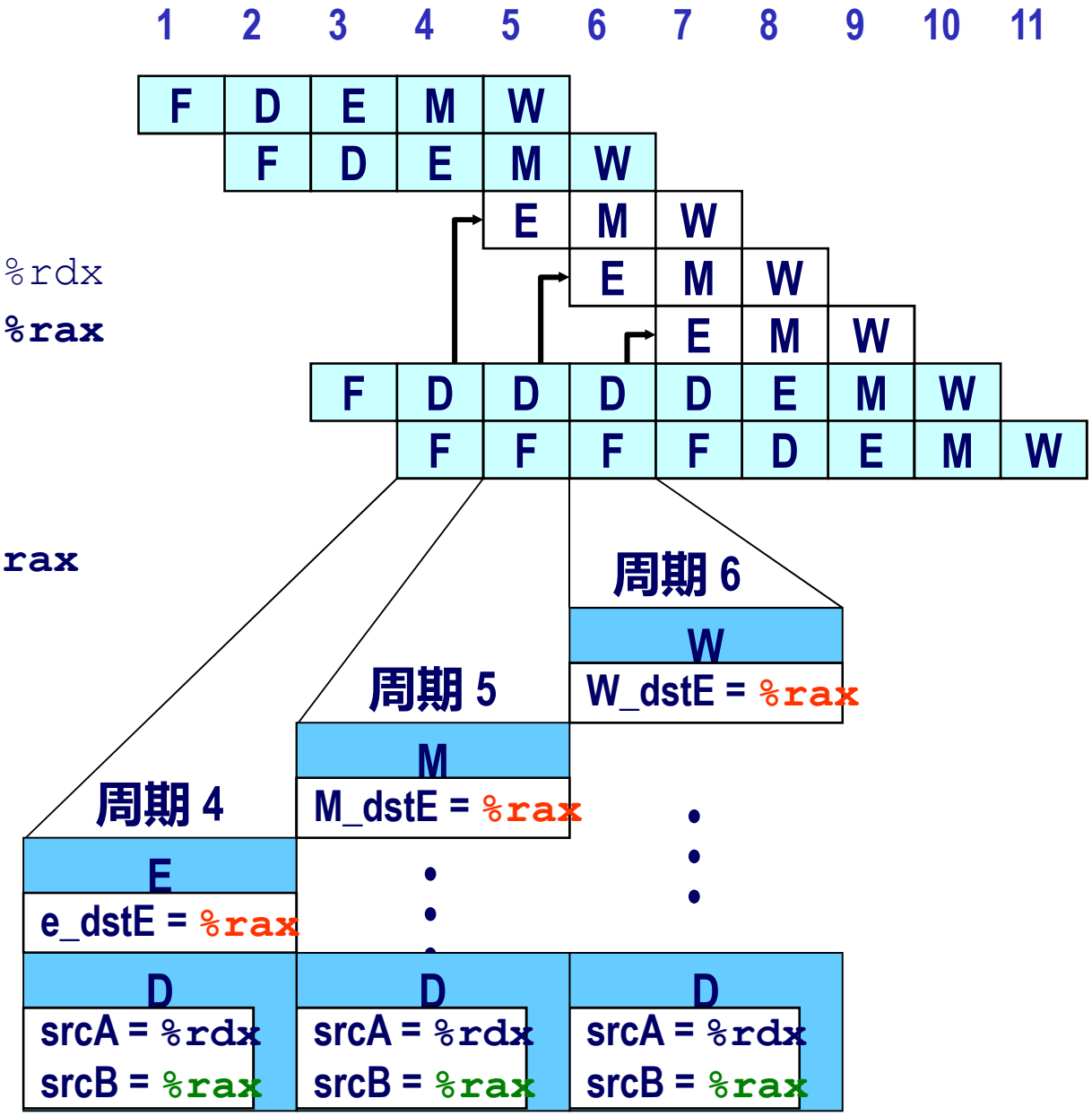
检测暂停条件

```
# 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

```
# 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?

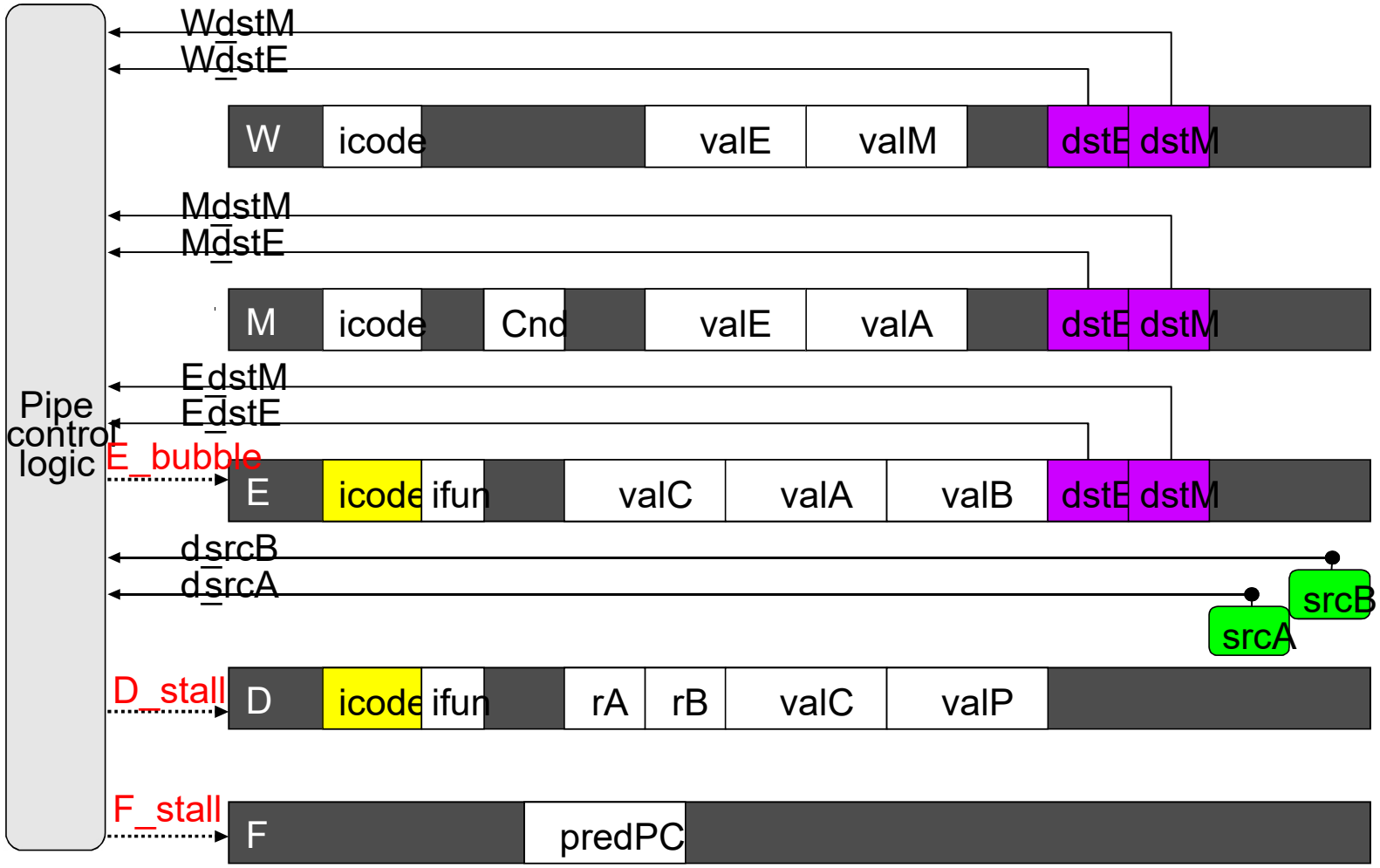
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

周期 8

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

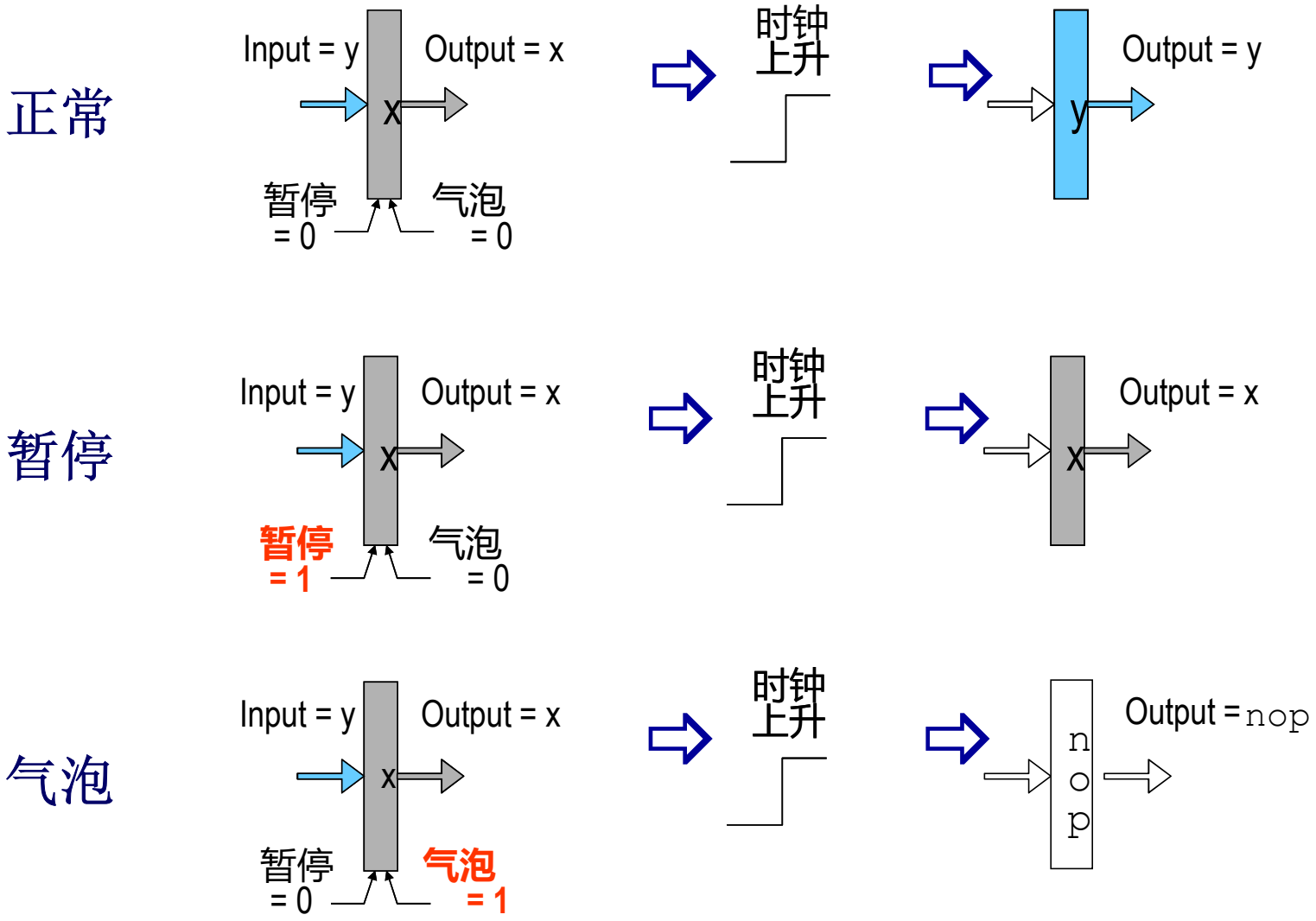
- 指令停顿在译码阶段
- 紧随其后的指令阻塞在取指阶段
- 气泡插入到执行阶段
  - 像一条自动产生的nop指令
  - 穿过后续阶段

暂停实现



- 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

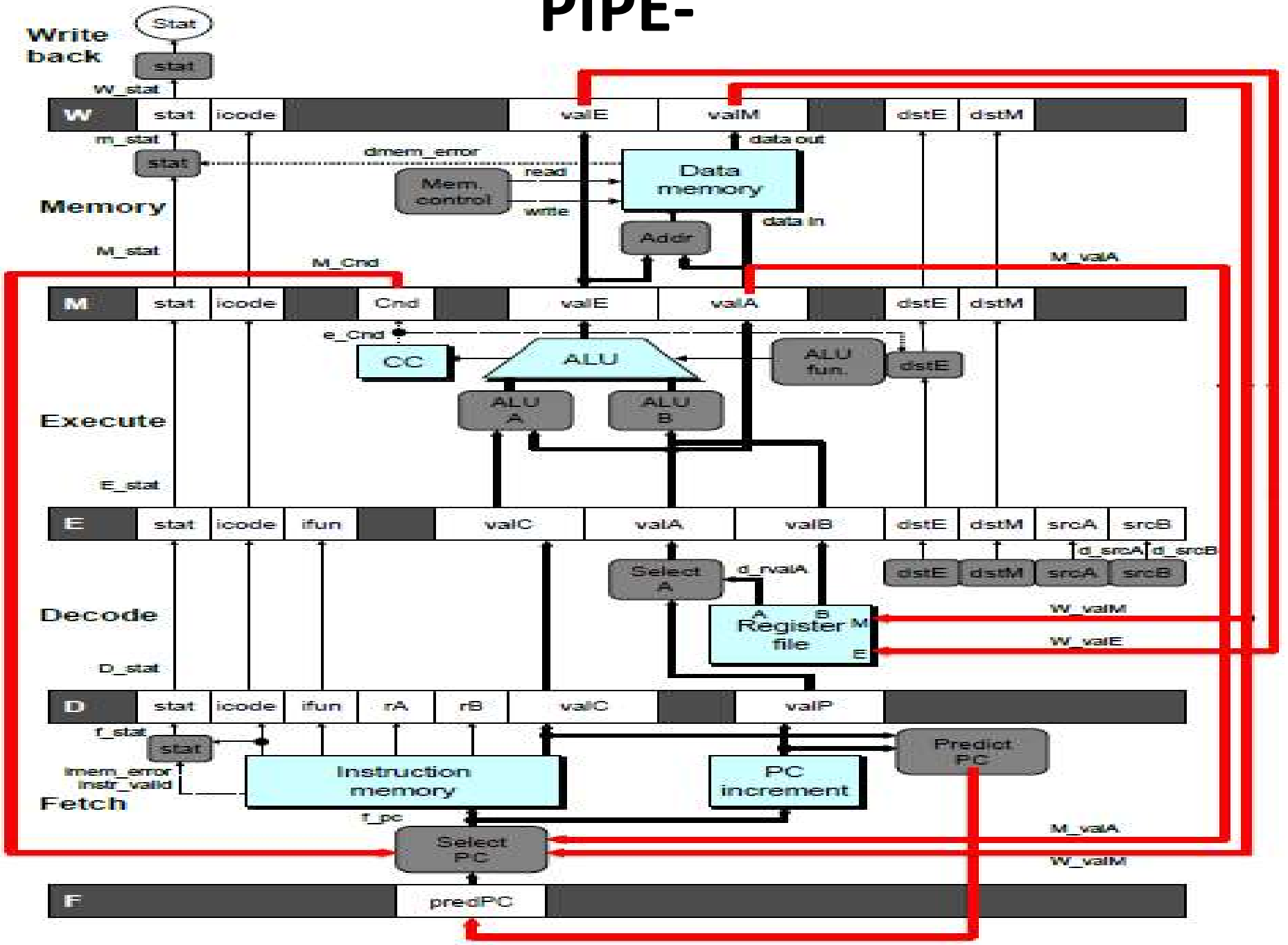
流水线寄存器模式


正常
Input = y
Output = x
x
暂停
= 0
气泡
= 0
时钟
上升
Output = y
y
暂停
Input = y
Output = x
x
暂停
= 1
气泡
= 0
时钟
上升
Output = x
x
气泡
Input = y
Output = x
x
暂停
= 0
气泡
= 1
时钟
上升
Output = nop
nop

数据转发—增加旁路路径解决数据冒险

- 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
- **转发源: e_valE m_valM M_valE W_valM W_valE**
- **转发目的: val_A val_B**

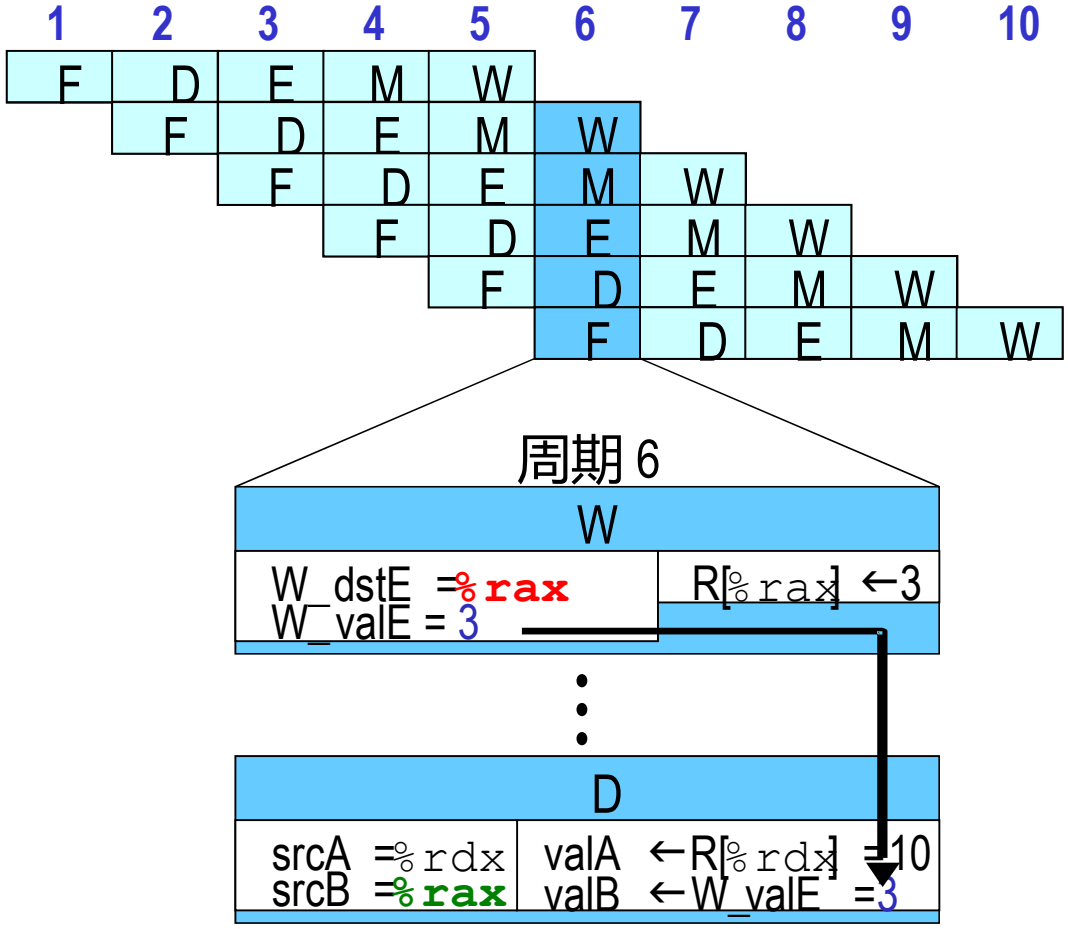
PIPE-


Write back
Stat
stat
W_stat
W
stat
icode
valE
valM
dstE
dstM
m_stat
stat
dmem_error
data out
Mem. control
read
write
Data memory
data in
Addr
Memory
M_stat
M_Cnd
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
dstE
ALU A
ALU B
Execute
E_stat
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
Select A
d_rvalA
dstE
dstM
srcA
srcB
W_valM
A
B
Register file
M
E
W_valE
Decode
D_stat
D
stat
icode
ifun
rA
rB
valC
valP
f_stat
stat
Predict PC
imem_error
instr_valid
Instruction memory
PC increment
Fetch
f_pc
M_valA
Select PC
W_valM
F
predPC

数据转发示例

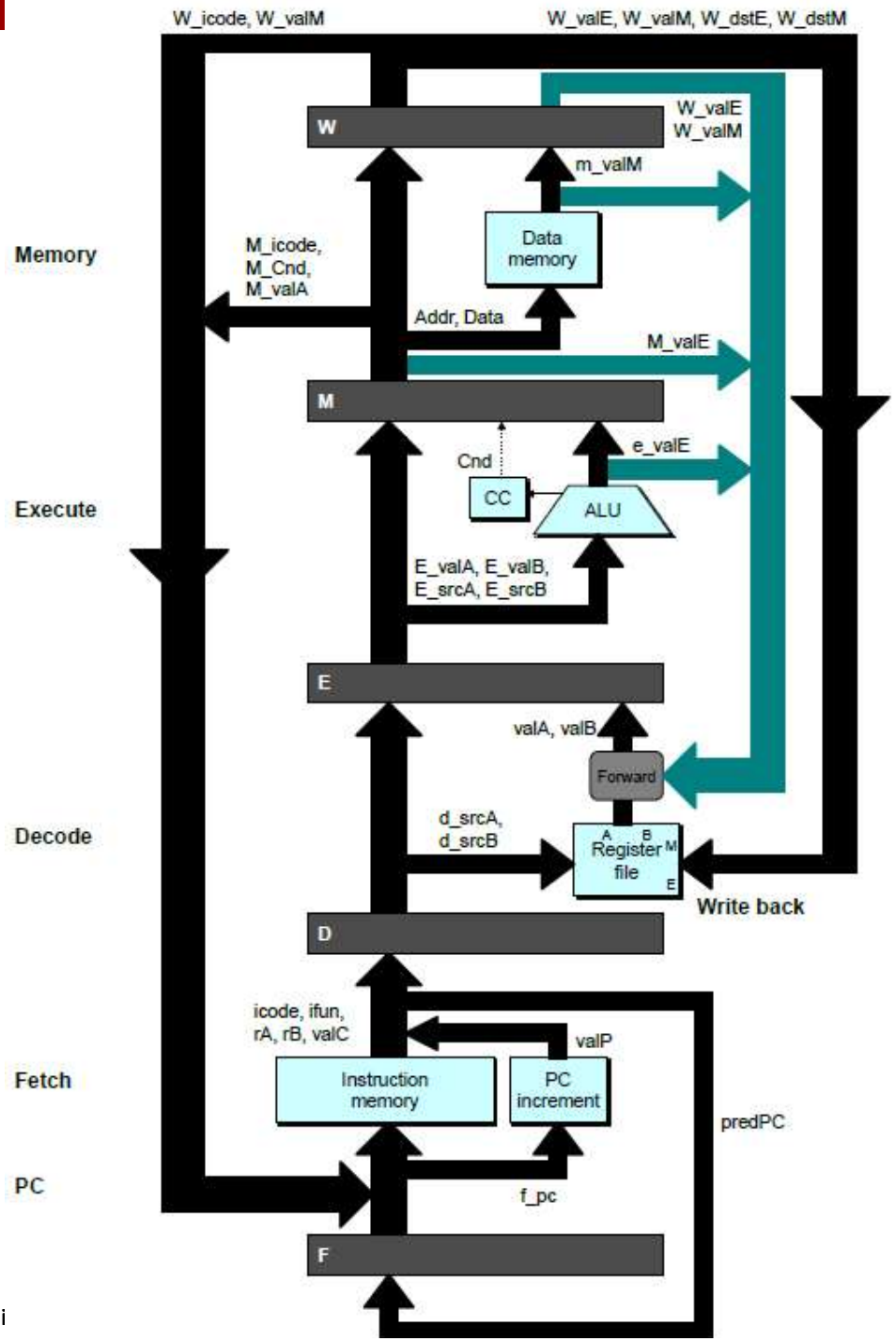
```
# demo-h2.ys
0x000:irmovq$10,%rdx
0x00a:irmovq $3,%rax
0x014:nop
0x015:nop
0x016:addq%rdx,%rax
0x018: halt
```

- 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



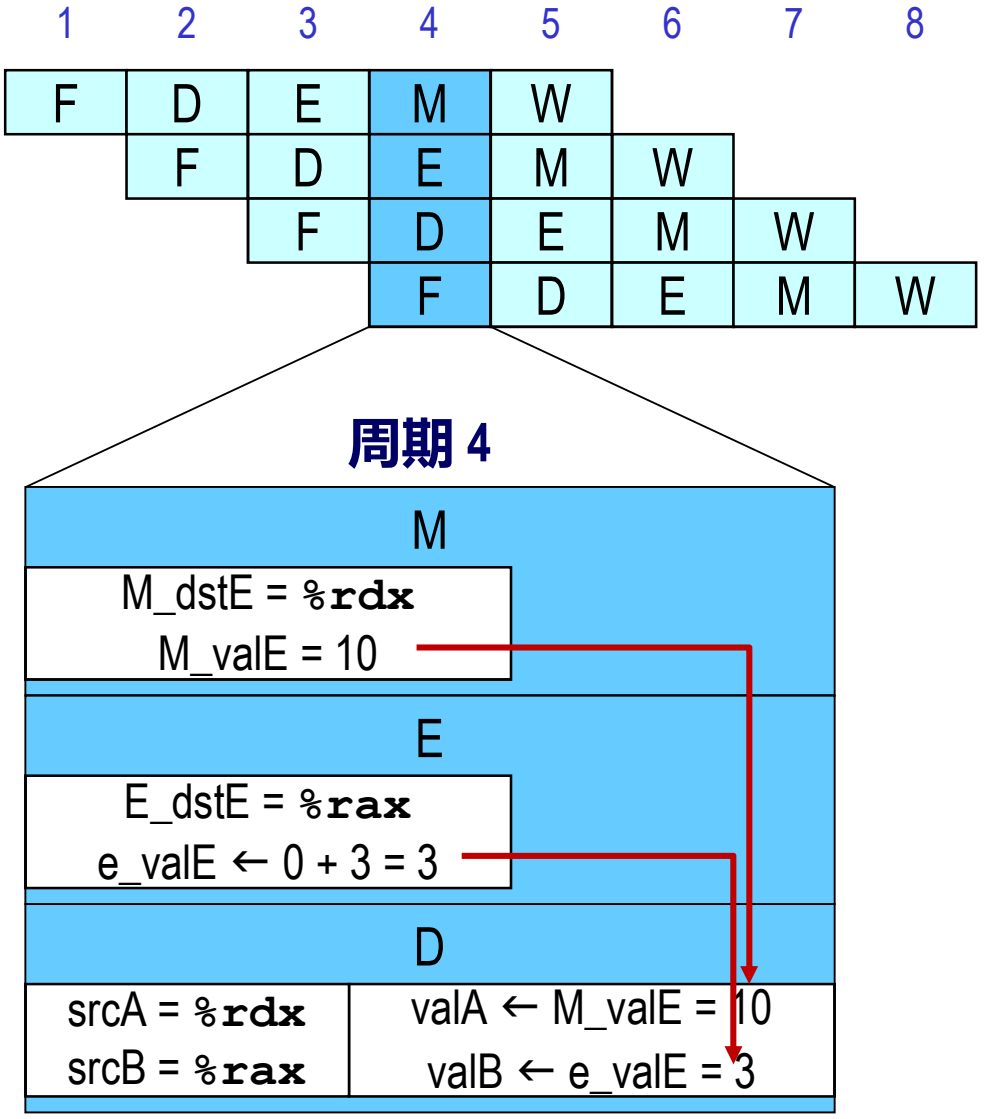
旁路路径

- 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

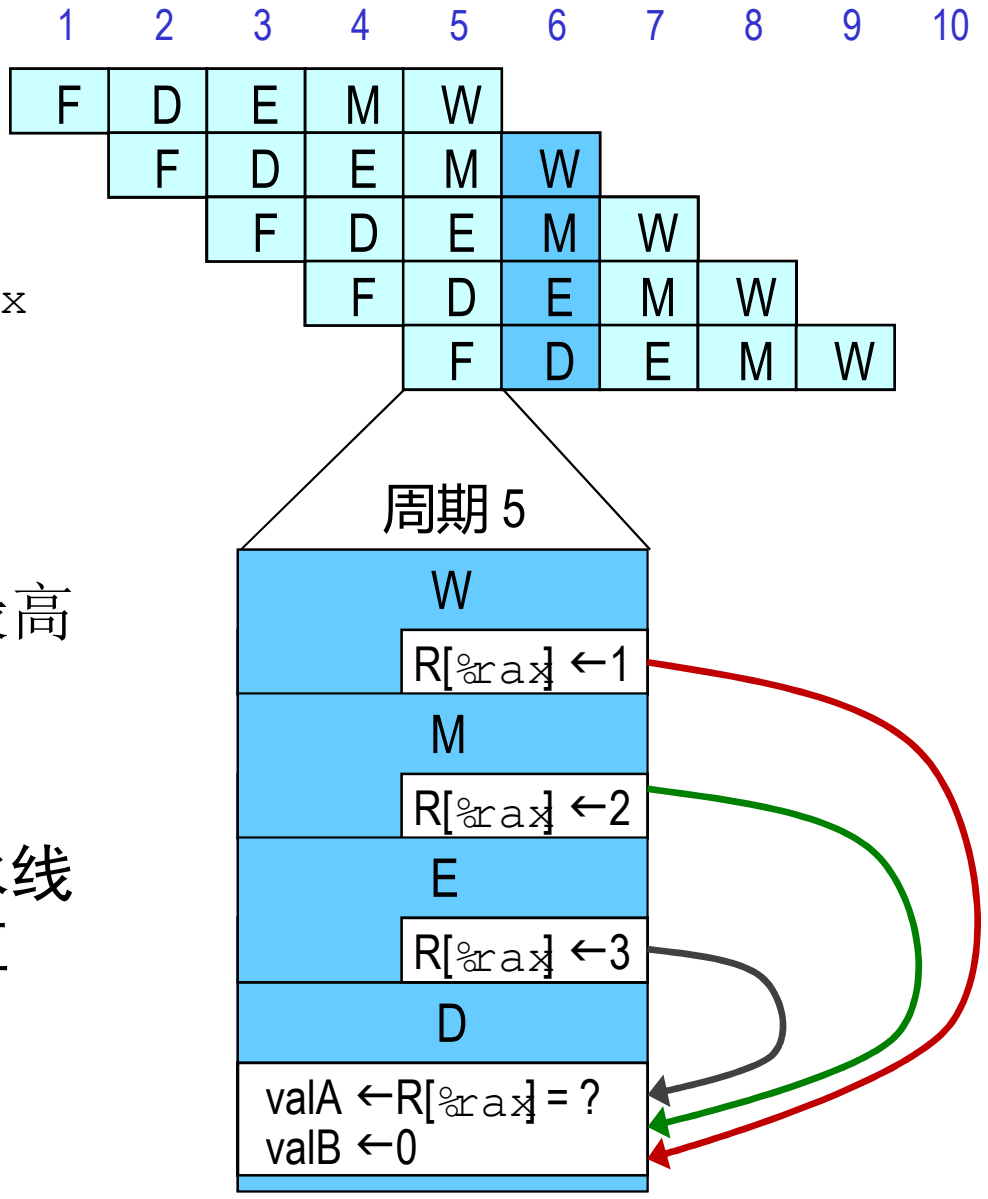
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



- 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

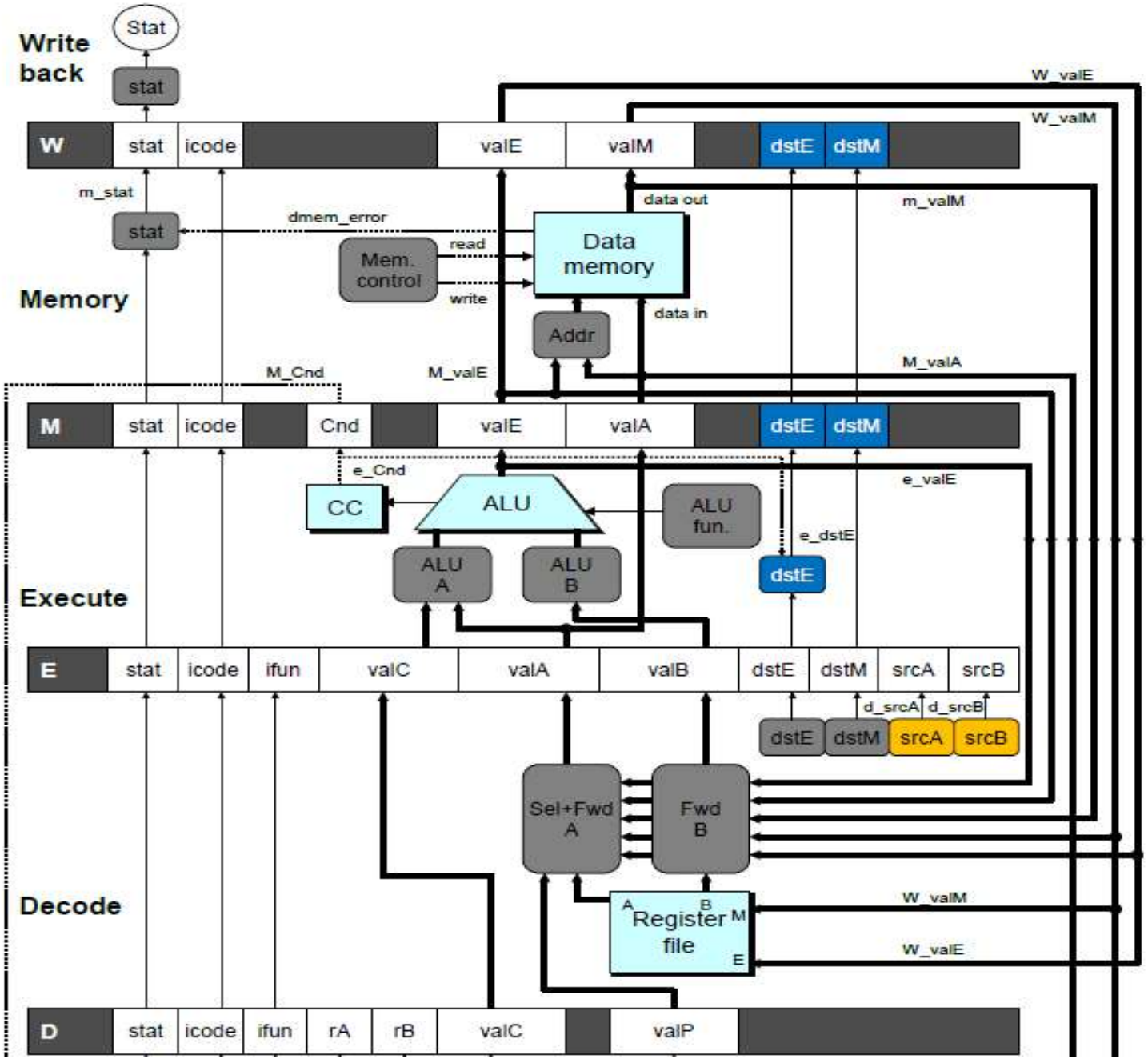
转发优先级

```
   # 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
  - 匹配串行语义
  - **使用从最早的流水线阶段获取的匹配值**

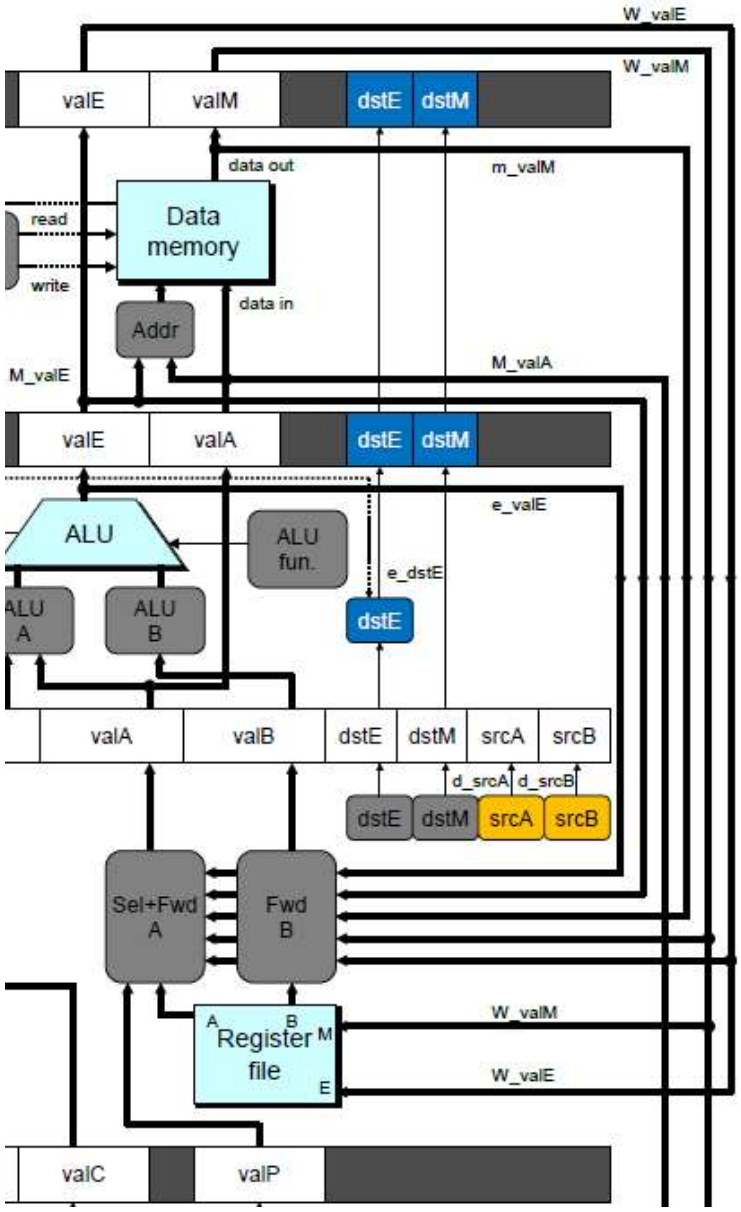
实现转发



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

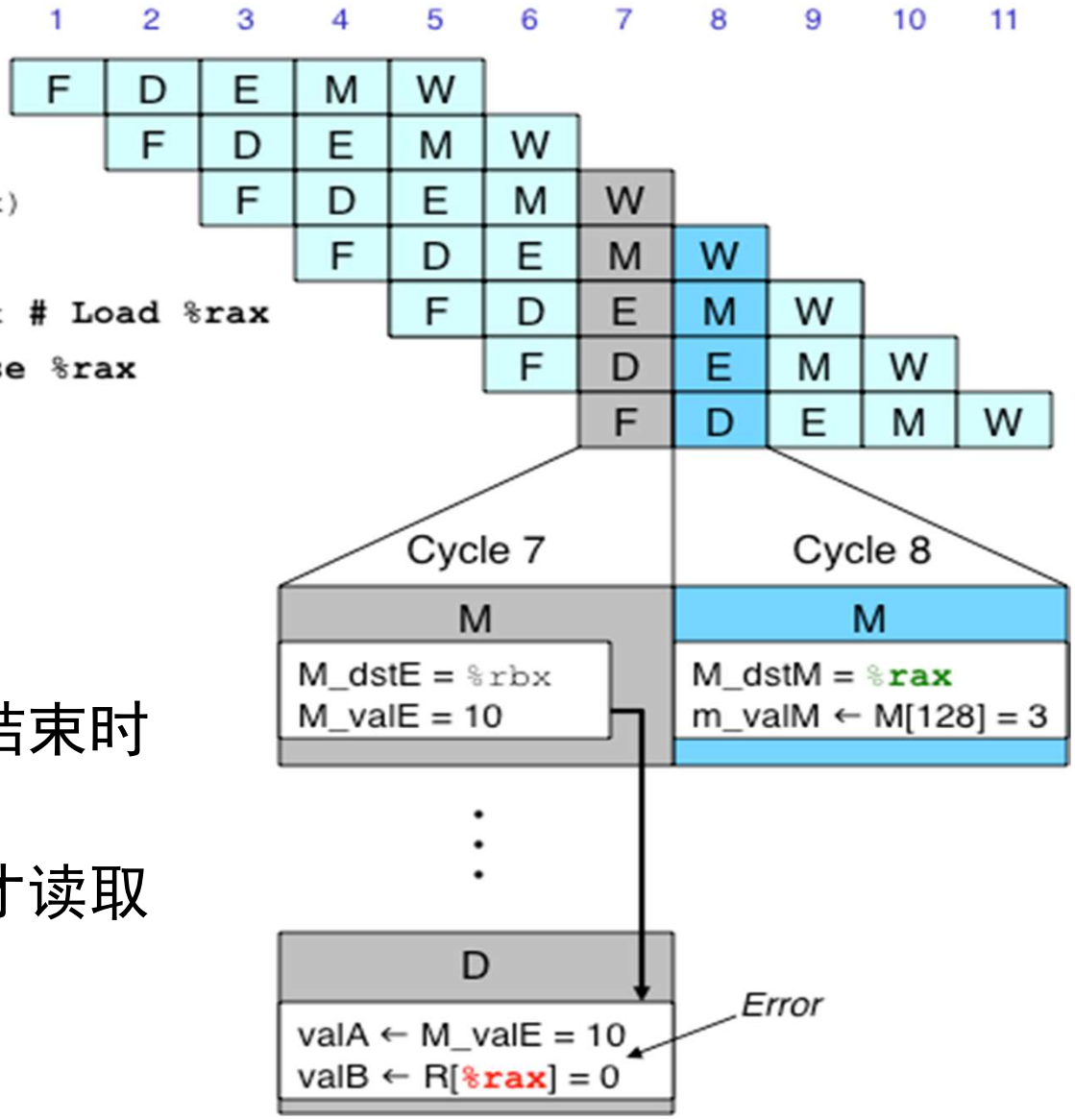
实现转发




```
## 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

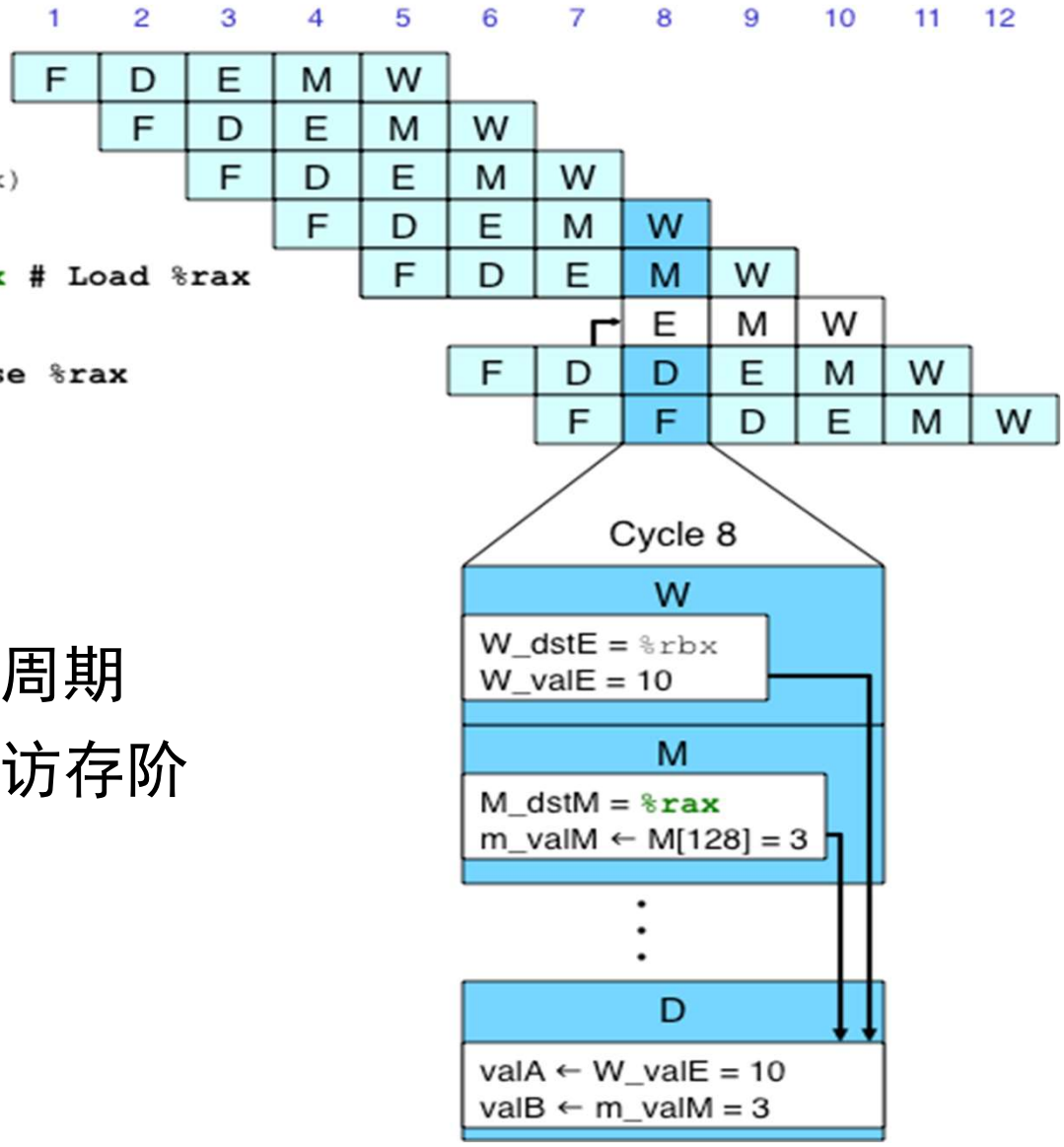
```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```



- 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

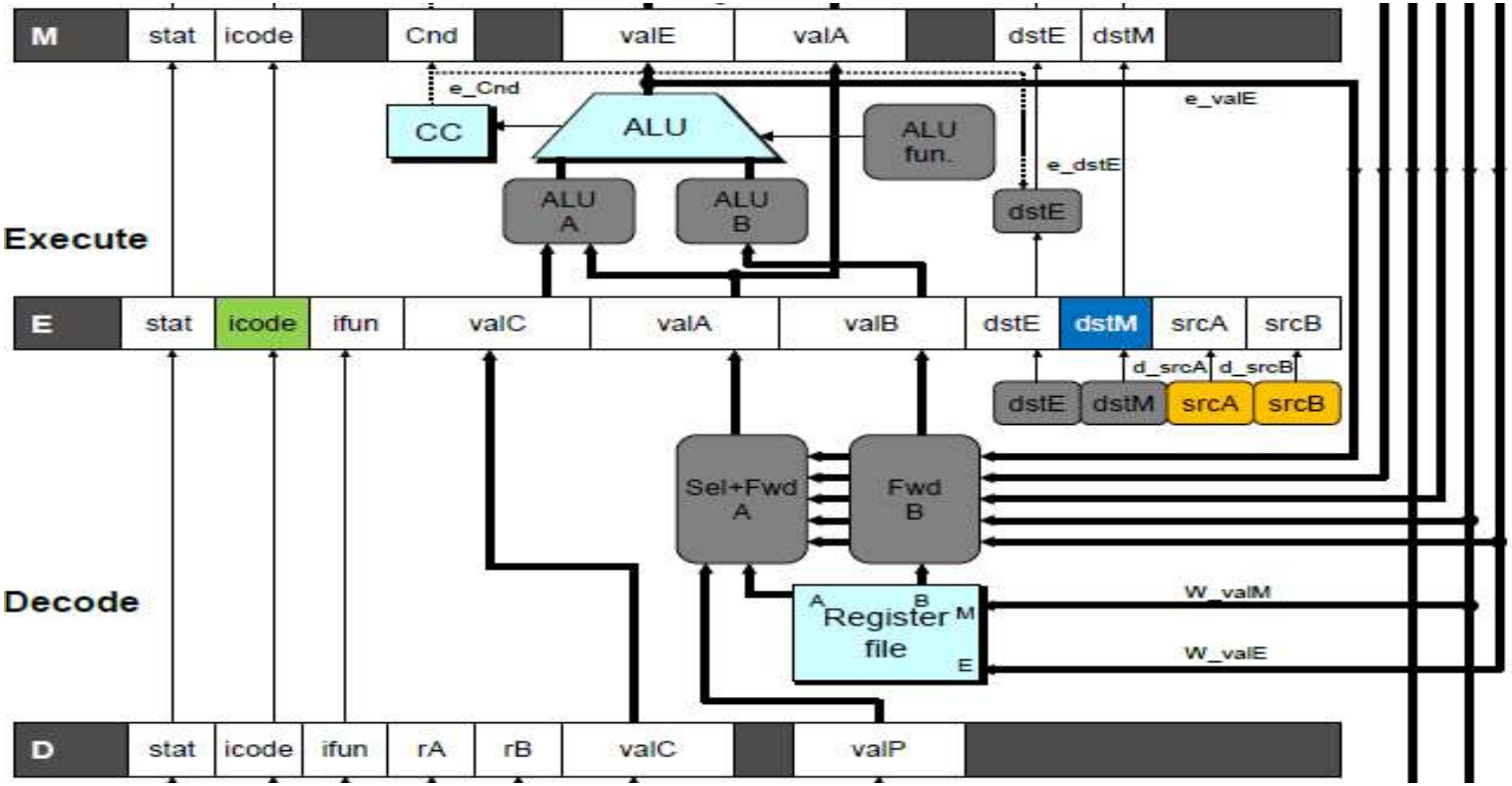
避免 加载/使用 冒险

```
# 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

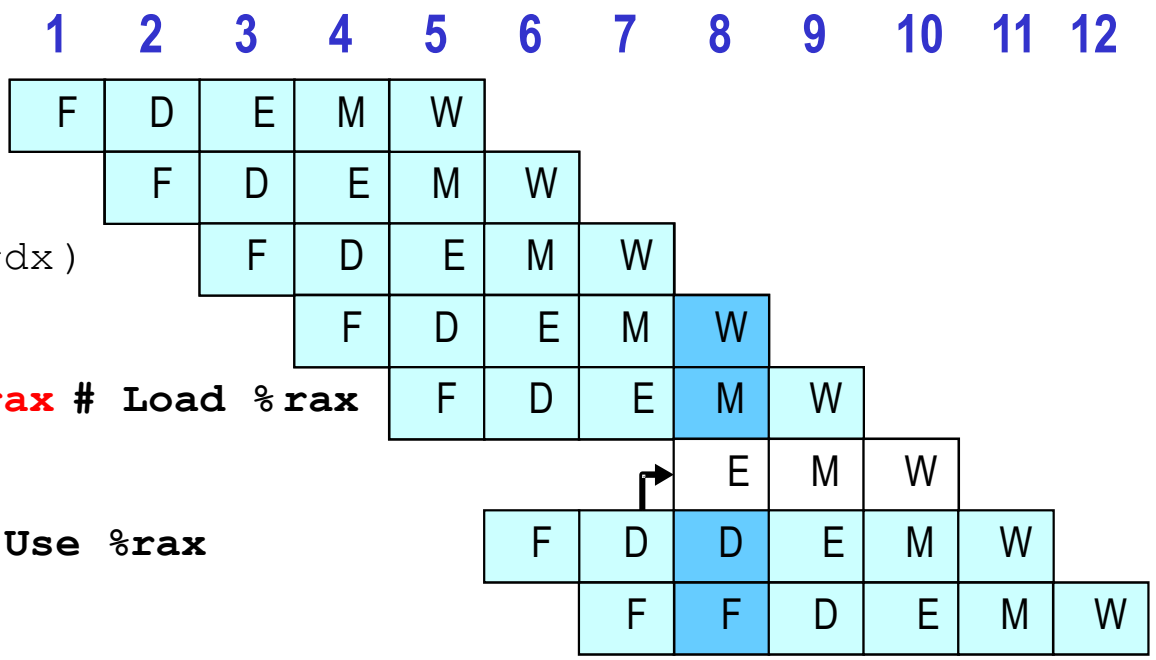
检测 加载/使用 冒险



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

分支预测错误示例

demo-j.ys

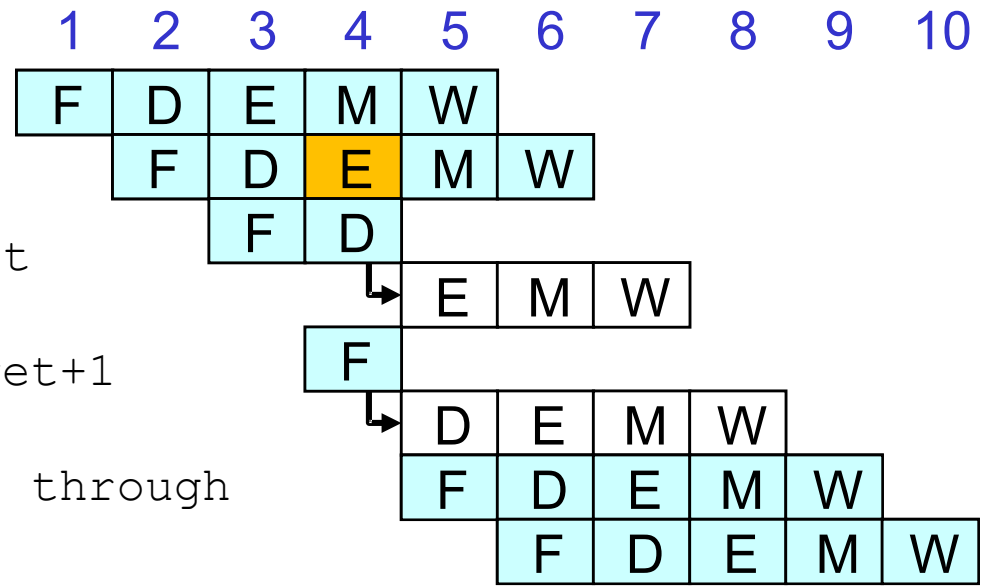
```
0x000:     xorq %rax,%rax
0x002:     jne  t                # Not taken
0x00b:     irmovq $1, %rax       # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx       # Target
0x023:     irmovq $4, %rcx       # Should not execute
0x02d:     irmovq $5,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

- 只能执行最早的8条指令

处理预测错误

#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
```



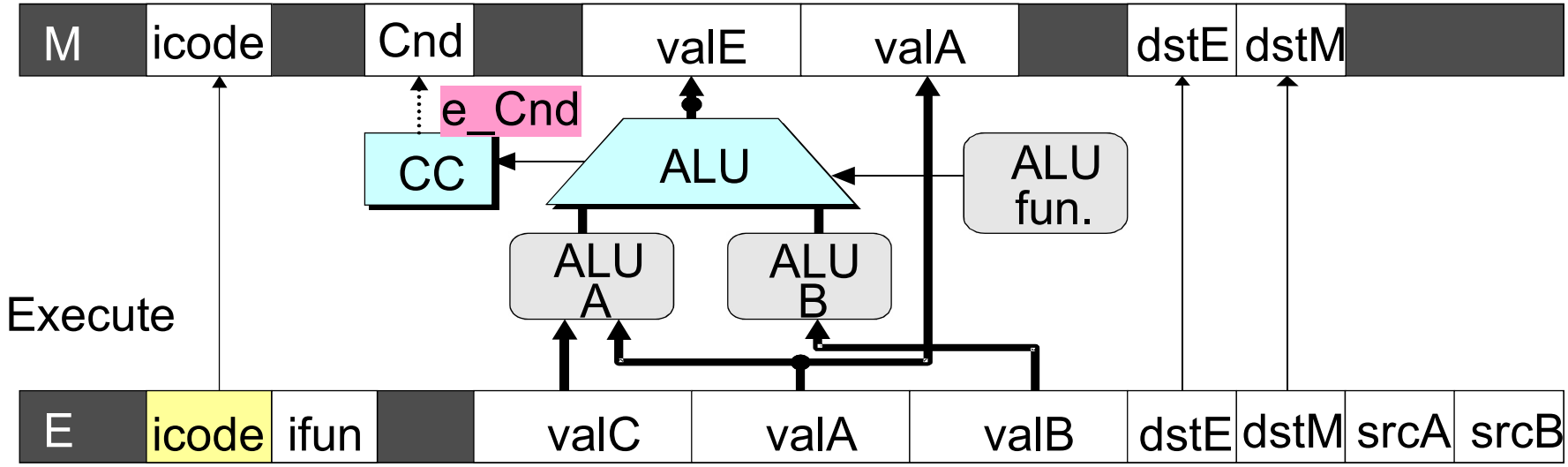
作为预测分支

- 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**

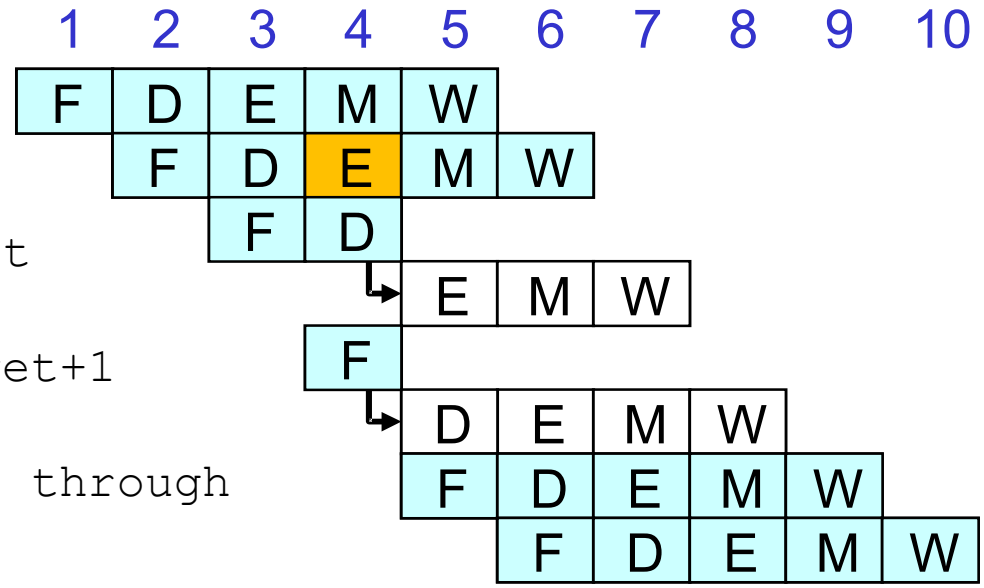
检测分支预测错误



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

Return示例

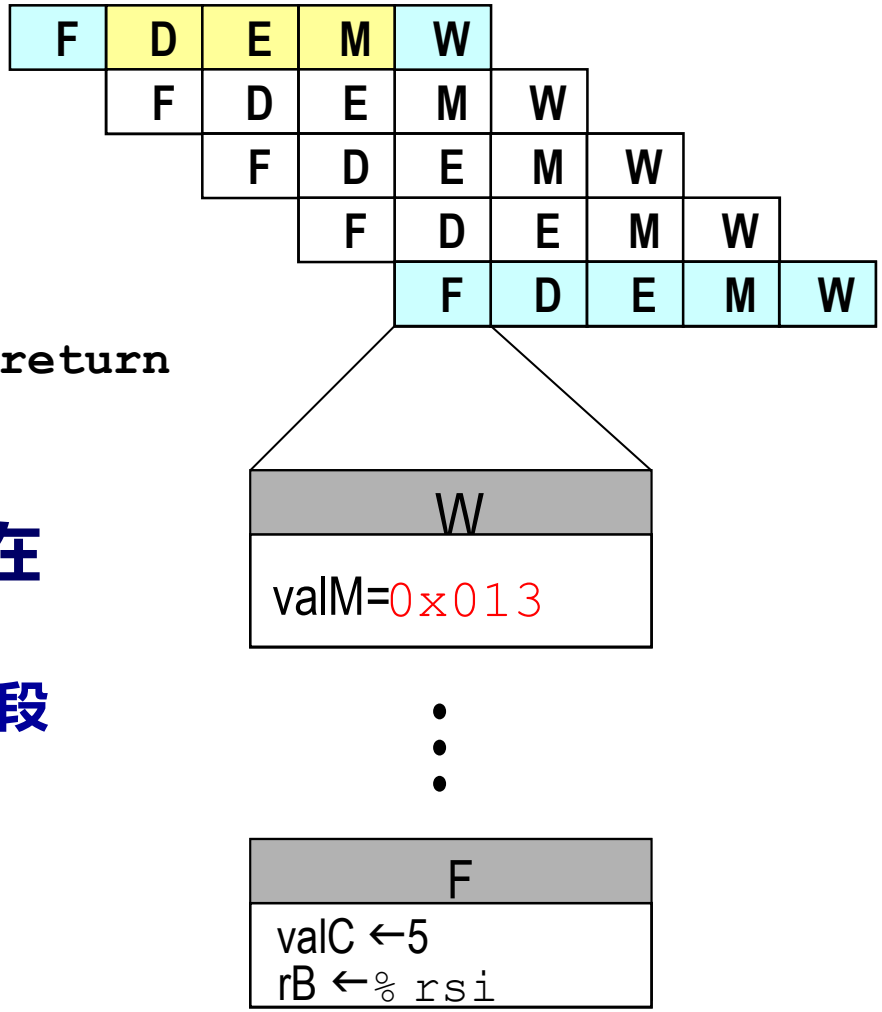
demo-retb.ys

```
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

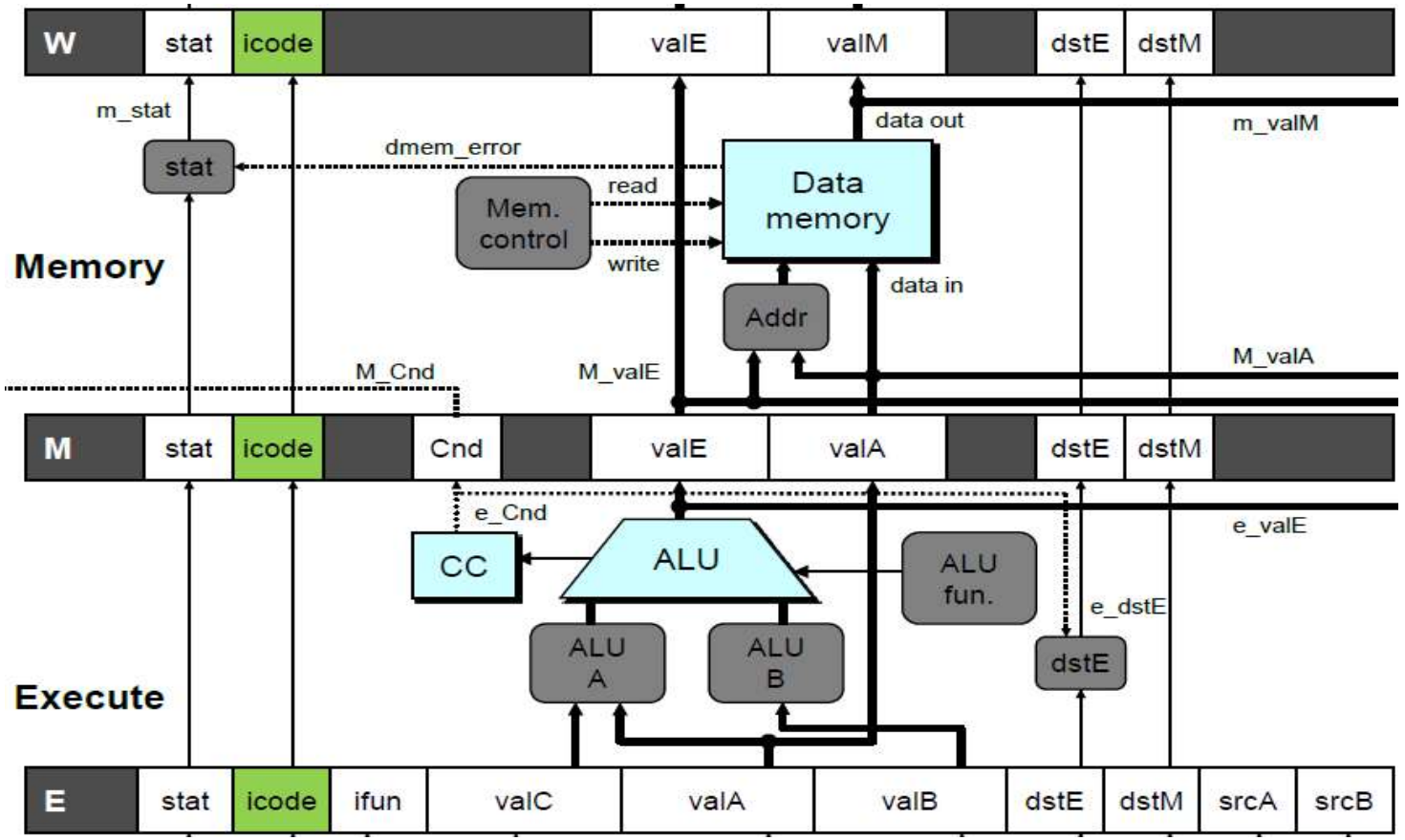
正确的Return示例

```
#demo_retb
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

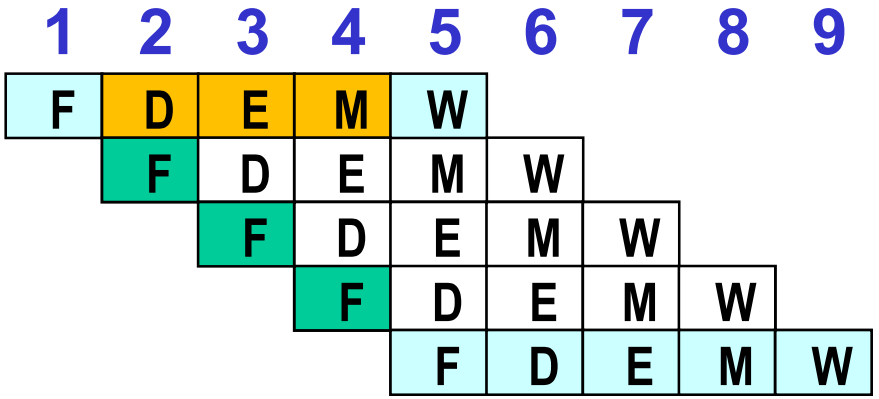
检测Return



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

Return的控制

```
 # demeretb

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



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

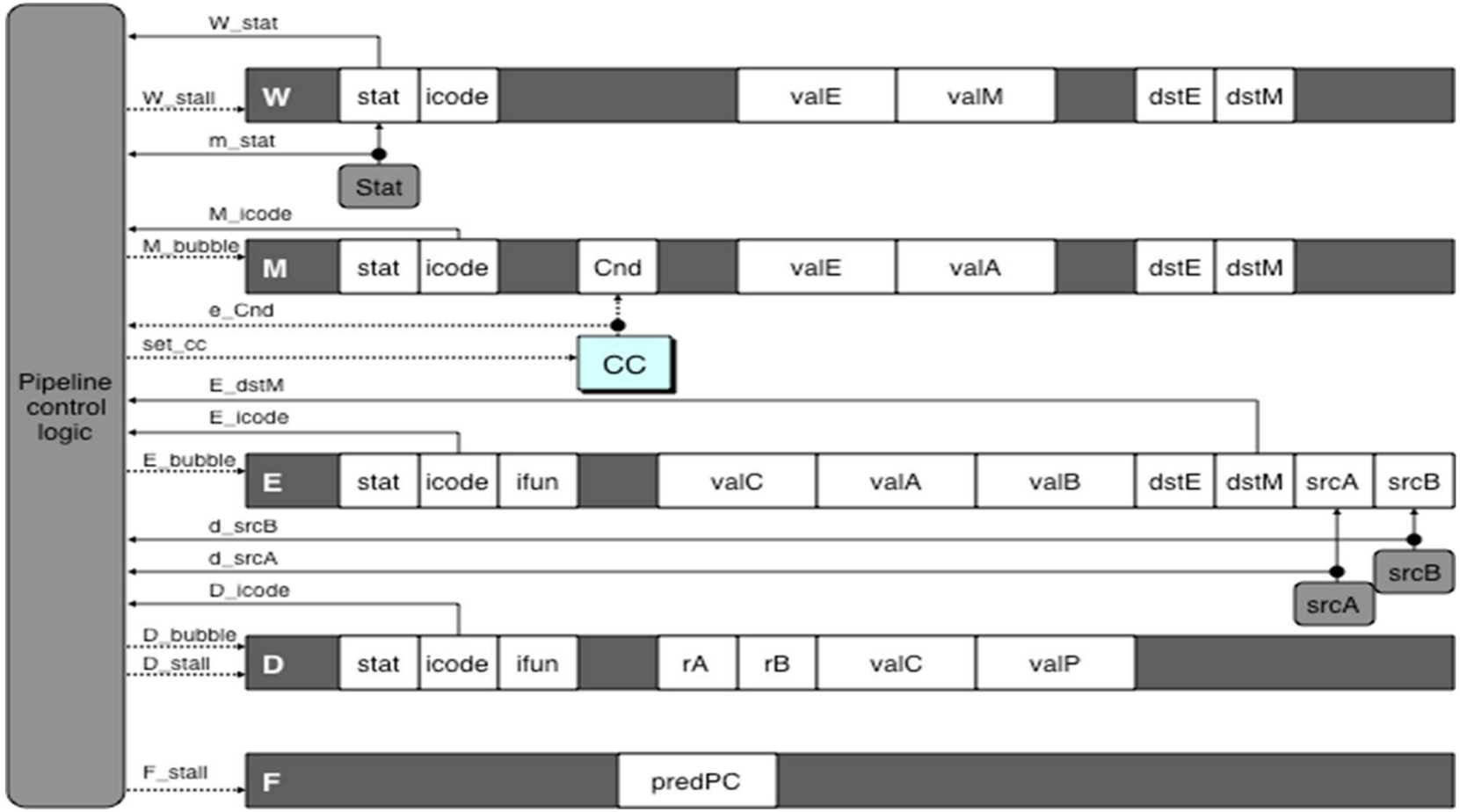
- 检测

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(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

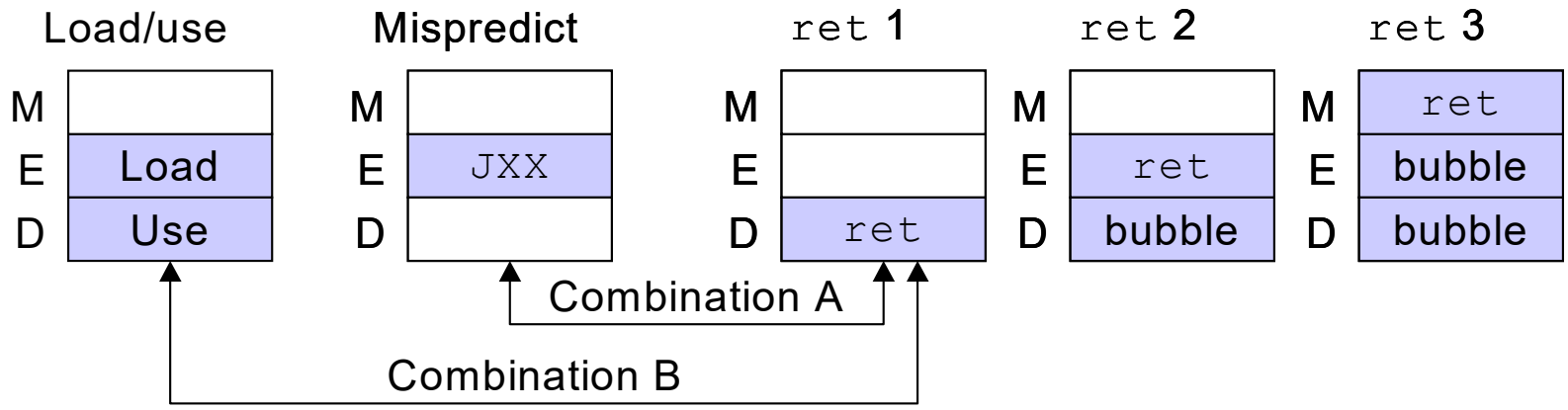
流水线控制的初始版本

```
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

```
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 };
```

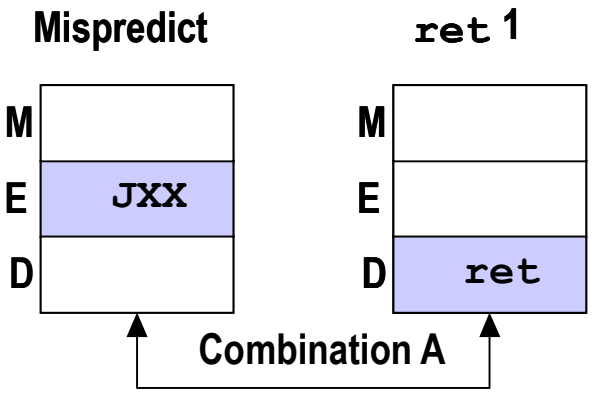
控制组合

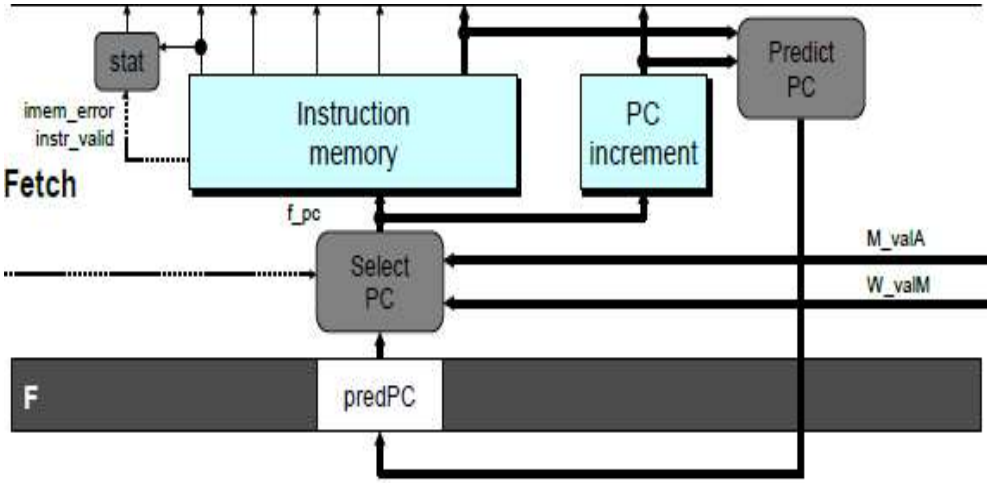


- 在一个时钟周期内可能出现多个特殊情况

- 组合**A**
  - 不选择分支
  - 位于分支目标的`ret` 指令
- 组合**B**
  - **指令从内存读取到`%rsp`**
  - 紧跟着ret指令

控制组合 A

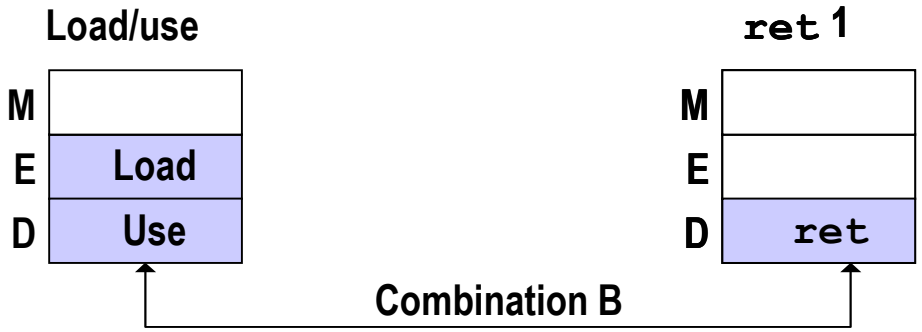




条件	F	D	E	M	W
处理ret	暂停	气泡	正常	正常	正常
分支预测错误	正常	气泡	气泡	正常	正常
组合	***暂停***	***气泡***	***气泡***	***正常***	***正常***

- 当分支预测错误时应该处理
- 暂停F流水线寄存器
- 但是PC的选择逻辑将会使用M_valM

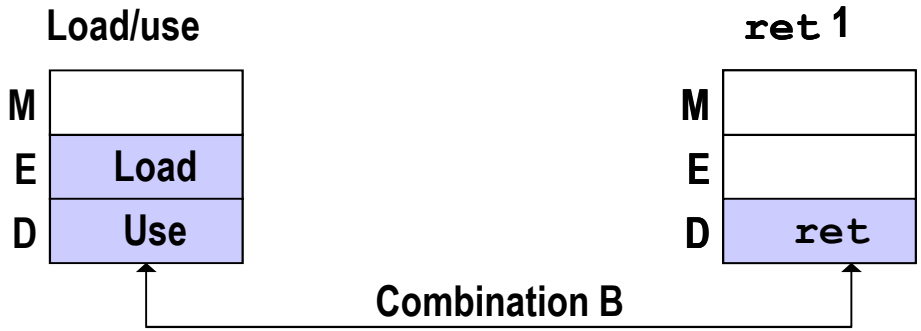
控制组合 B



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***

- 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

处理控制组合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***	***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

正确的流水线控制逻辑

```
bool D_气泡 =
    # 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
  - **暂停在取指阶段直到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**

第四章 处理器体系结构
——处理器的性能

教 师： 史先俊
计算机科学与技术学院
哈尔滨工业大学

目录

- PIPE设计总结
  - 异常条件
  - 性能分析
  - 取指阶段的设计
- 现代高性能处理器
  - 乱序执行

异常

- 处理器不能继续正常操作的条件
- 原因
  - 停机指令 (当前)
  - 取指或读数试图访问一个非法地址 (之前)
  - 非法指令 (之前)
- 期望行为
  - 完成一些指令
    - 或者当前或者之前，取决于异常类型
  - 抛弃其他指令
  - 调用异常处理程序
    - 类似于异常过程调用
- 我们的实现方法
  - 当指令引起异常时就停机

异常例子

- 取指阶段的异常

```
jmp $-1                          # 无效跳转目标

.byte 0xFF                       # 无效指令代码

halt                             # 停止指令
```

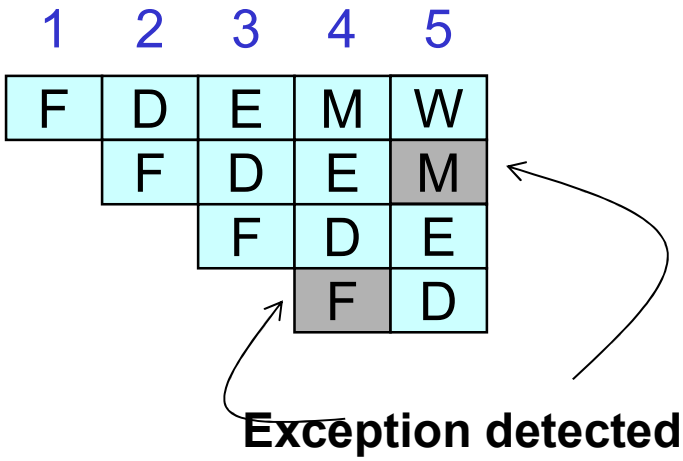
访存阶段的异常

```
irmovq $100,%rax
rmmovq %rax,0x10000(%rax)  # 无效地址
```

流水线处理器中的异常#1

```
# demo-exc1.ys
irmovq $100,%rax
rmmovq %rax,0x10000(%rax)  # 无效地址
nop
.byte 0xFF                 # 无效指令代码
```

```
0x000: irmovq $100,%rax
0x00a: rmmovq %rax,0x1000(%rax)
0x014: nop
0x015: .byte 0xFF
```

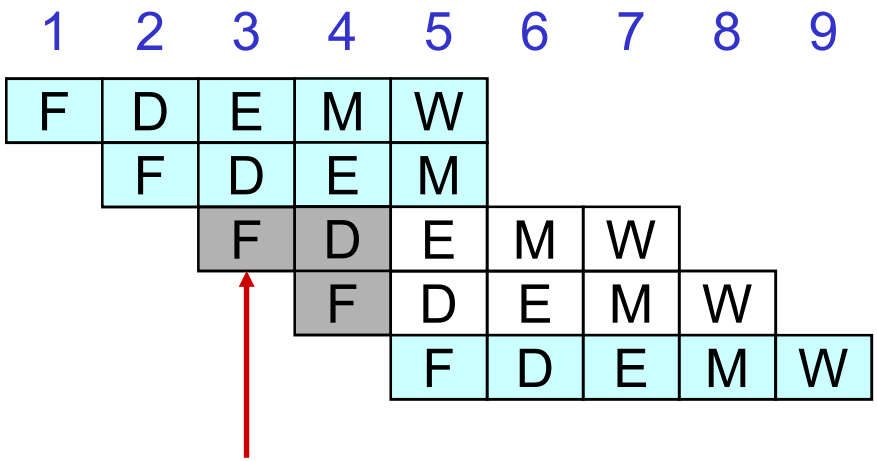


- 期望的行为
  - rmmovq 引起异常
  - 其他指令不受它的影响

流水线处理器中的异常#2

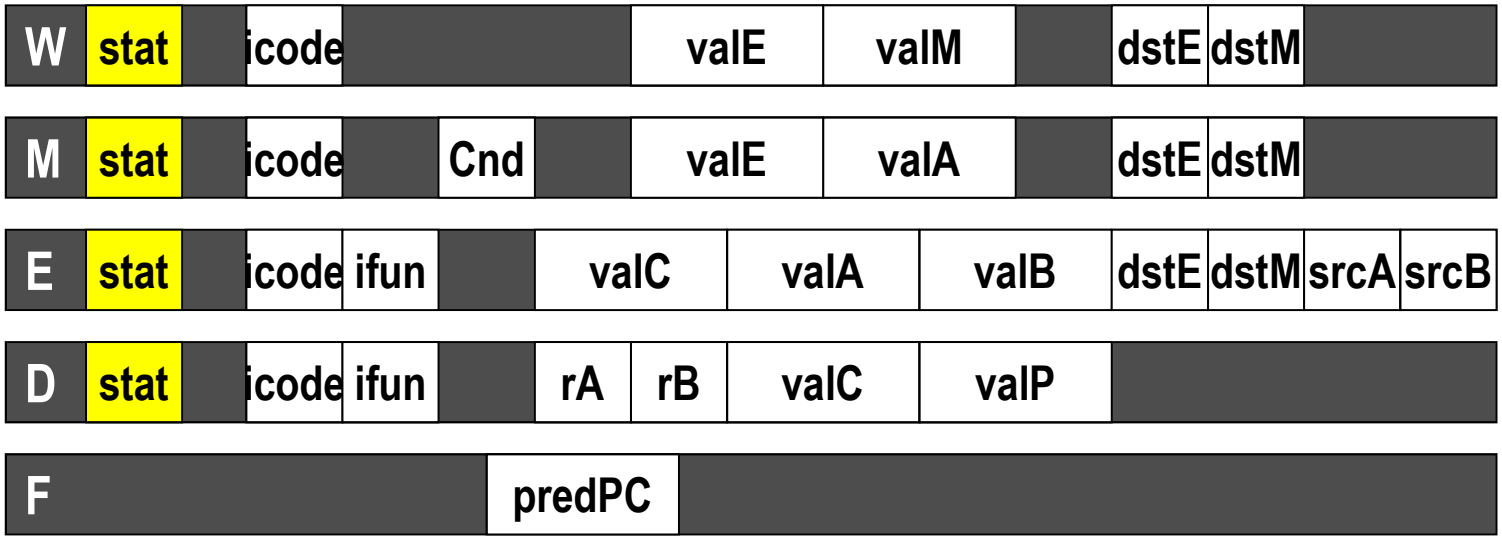
```
# demo-exc2.ys
0x000:     xorq %rax,%rax    # Set condition codes
0x002:     jne t             # Not taken
0x00b:     irmovq $1,%rax
0x015:     irmovq $2,%rdx
0x01f:     halt
0x020: t:  .byte 0xFF        # Target
```

```
0x000:     xorq %rax,%rax
0x002:     jne t
0x020: t:  .byte 0xFF
0x???: (I'm lost!)
0x00b:     irmovq $1,%rax
```



- 期望的行为
  - 没有异常发生

维护异常的顺序



- 为流水线寄存器增加状态字段
- 取指阶段设为 “AOK,” “ADR” (当取指地址错误), “HLT” (停机指令) 或者 “INS” (非法指令)
- 解码和执行阶段传递值
- 访存阶段传递或设置为“ADR”
- 当指令进入写回阶段时，异常被触发

异常处理逻辑

- 取指阶段

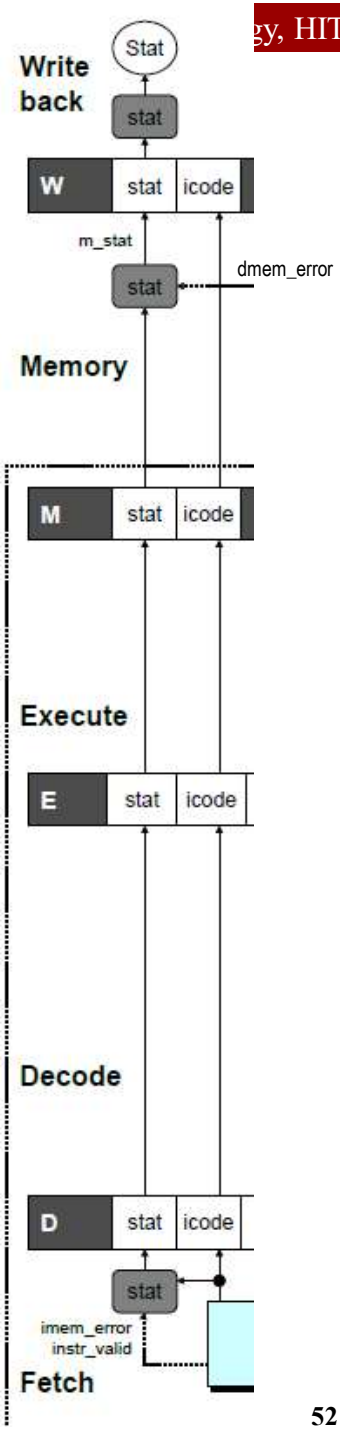
```
# Determine status code for fetched instruction
int f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];
```

- 访存阶段

```
# Update the status
int m_stat = [
        dmem_error : SADR;
        1 : M_stat;
];
```

- 写回阶段

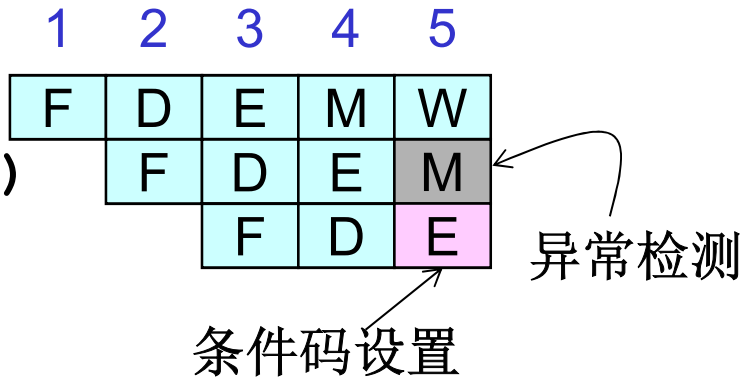
```
int Stat = [
        # SBUB in earlier stages indicates bubble
        W_stat == SBUB : SAOK;
        1 : W_stat;
];
```



流水线处理器中的副作用

```
# demo-exc3.ys
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # invalid address
addq %rax,%rax            # Sets condition codes
```

```
0x000: irmovq $100,%rax
0x00a: rmmovq %rax,0x1000(%rax)
0x014: addq %rax,%rax
```



- 期望的行为
  - rmmovq 指令引起异常
  - 其他指令不受影响

避免副作用

- 异常出现应该禁止状态更新
  - 非法指令转换为流水线气泡
    - 除非状态指示异常状态
  - 数据不会被写入无效的地址
  - 防止条件码进行无效更新
    - 在访存阶段检测异常
    - 在执行阶段禁止条件码更新
    - 必须在相同的时钟周期内发生
  - 在最后阶段处理异常
    - 当在访存阶段探测到异常时
      - 在下一个时钟周期将气泡插入访存阶段
    - 当在写回阶段探测到异常时
      - 停止异常指令
  - 包含在HCL代码中

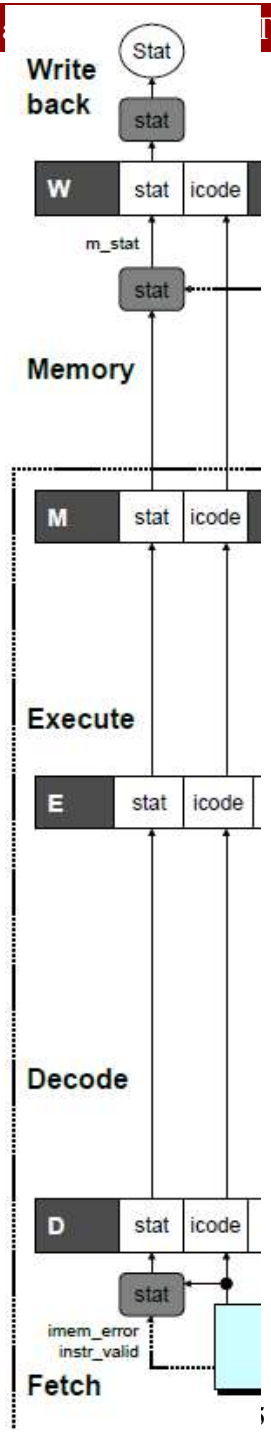
状态变化的控制逻辑

- 设置条件码

```
# Should the condition codes be updated?
bool set_cc = E_icode == IOPQ &&
        # State changes only during normal operation
        !m_stat in { SADR, SINS, SHLT }
        && !W_stat in { SADR, SINS, SHLT };
```

- 阶段控制
  - 也控制访存阶段的更新

```
# Start injecting bubbles as soon as exception passes
through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT }
        || W_stat in { SADR, SINS, SHLT };

# stall pipeline register W when exception encountered
bool W_stall = W_stat in { SADR, SINS, SHLT };
```



其他实际的异常处理

- 调用异常处理程序
  - 将PC入栈
    - PC指向故障指令或下一条指令
    - 通常和异常状态一起通过流水线传输
  - 跳转到处理程序的入口地址
    - 通常是固定地址
    - 被定义为ISA的一部分
- 实现
  - 尚未实现！ ---OS部分

性能评估

- 时钟频率
  - 以Ghz计算
  - 阶段功能的划分和电路的设计
    - 保持每个阶段的工作量尽可能的小
- 指令的执行速率
  - CPI: 每指令周期数
  - 平均来说，每条指令需要的时钟周期数
  - 流水线功能设计和基准程序
    - 例如：分支预测错误的频率

PIPE的CPI

- **CPI ≈ 1.0**
  - 每个周期取一条指令
  - 几乎每个周期都有效的执行一条新指令
    - 虽然每个单独的指令具有5个周期的延迟
- **CPI > 1.0**
  - 有时必须停顿或取消分支
- **计算 CPI**
  - C：时钟周期
  - I：执行完成的指令数
  - B：插入的气泡个数 (C = I + B)

$$CPI = C/I = (I+B)/I = 1.0 + B/I$$

  - 因子B/I代表因气泡而产生的平均处罚

PIPE的CPI（续)

$$B/I = LP + MP + RP$$

Typical Values

- LP:由加载/使用冒险停顿产生的处罚
  - 加载指令的比例 0.25
  - 加载指令需要停顿的比例 0.20
  - 每次插入气泡的数量 1

  ⇒ LP = 0.25 * 0.20 * 1 = 0.05
- MP:由错误的分支预测产生的处罚
  - 条件转移指令的比例 0.20
  - 条件转移预测错误的比例 0.40
  - 每次插入气泡的数量 2

  ⇒ MP = 0.20 * 0.40 * 2 = 0.16
- RP: 由ret指令产生的处罚
  - 返回指令站的比例 0.02
  - 每次插入的气泡数量 3

  ⇒ RP = 0.02 * 3 = 0.06

PIPE的CPI（续)

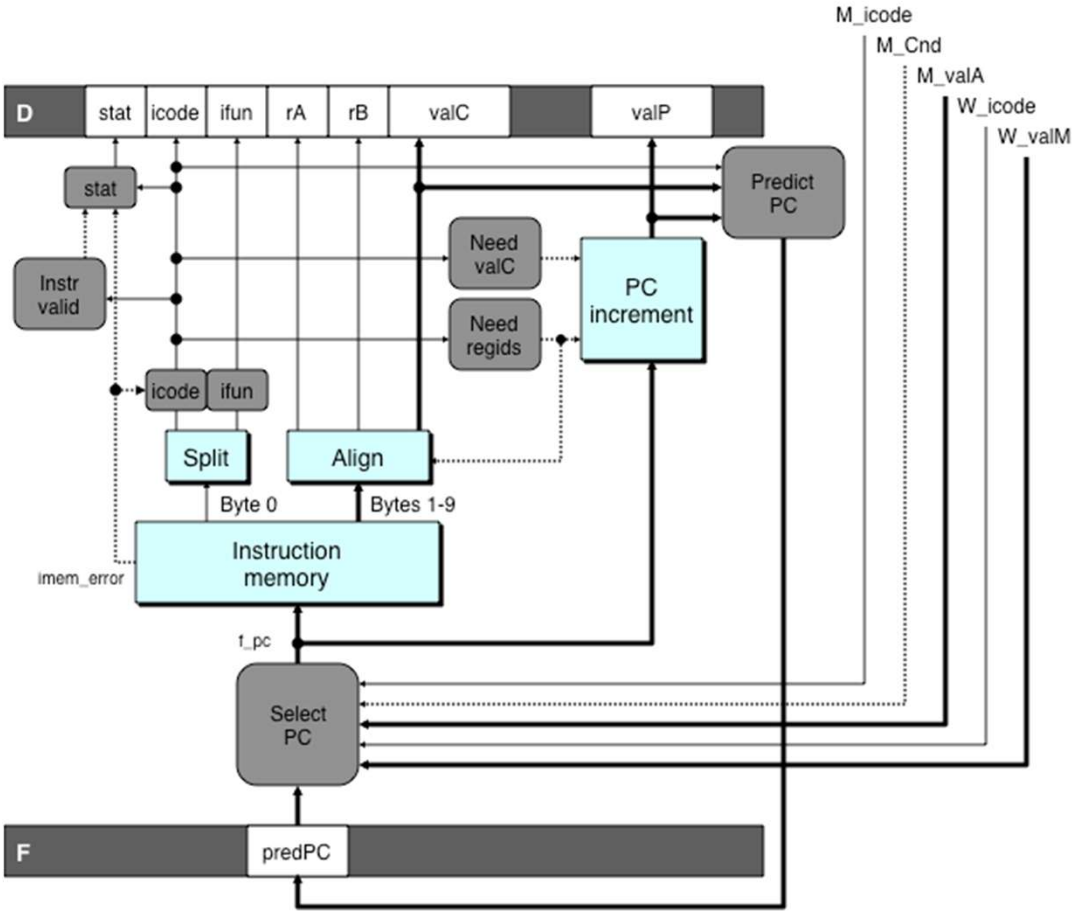
$$B/I = LP + MP + RP$$

- 处罚造成的影响（三种处罚的总和) 0.05 + 0.16 + 0.06 = 0.27

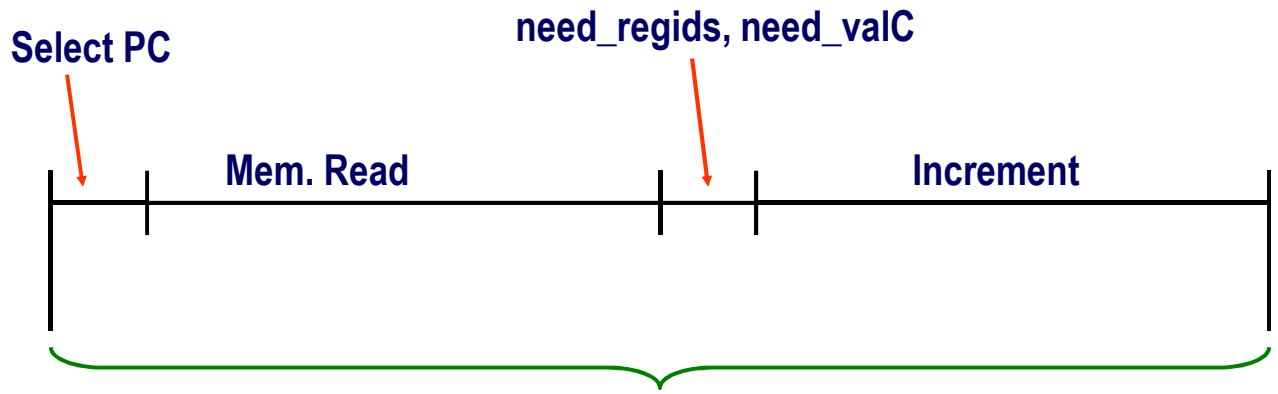
  $\Rightarrow$ CPI = 1.27　(Not bad!)

取指逻辑回顾

- 在取指周期期间
  1. 选择PC
  2. 从指令存储其中读取指令
  3. 检查icode确定指令长度
  4. 递增PC
- 时间
  - 第二、四步需要大量时间



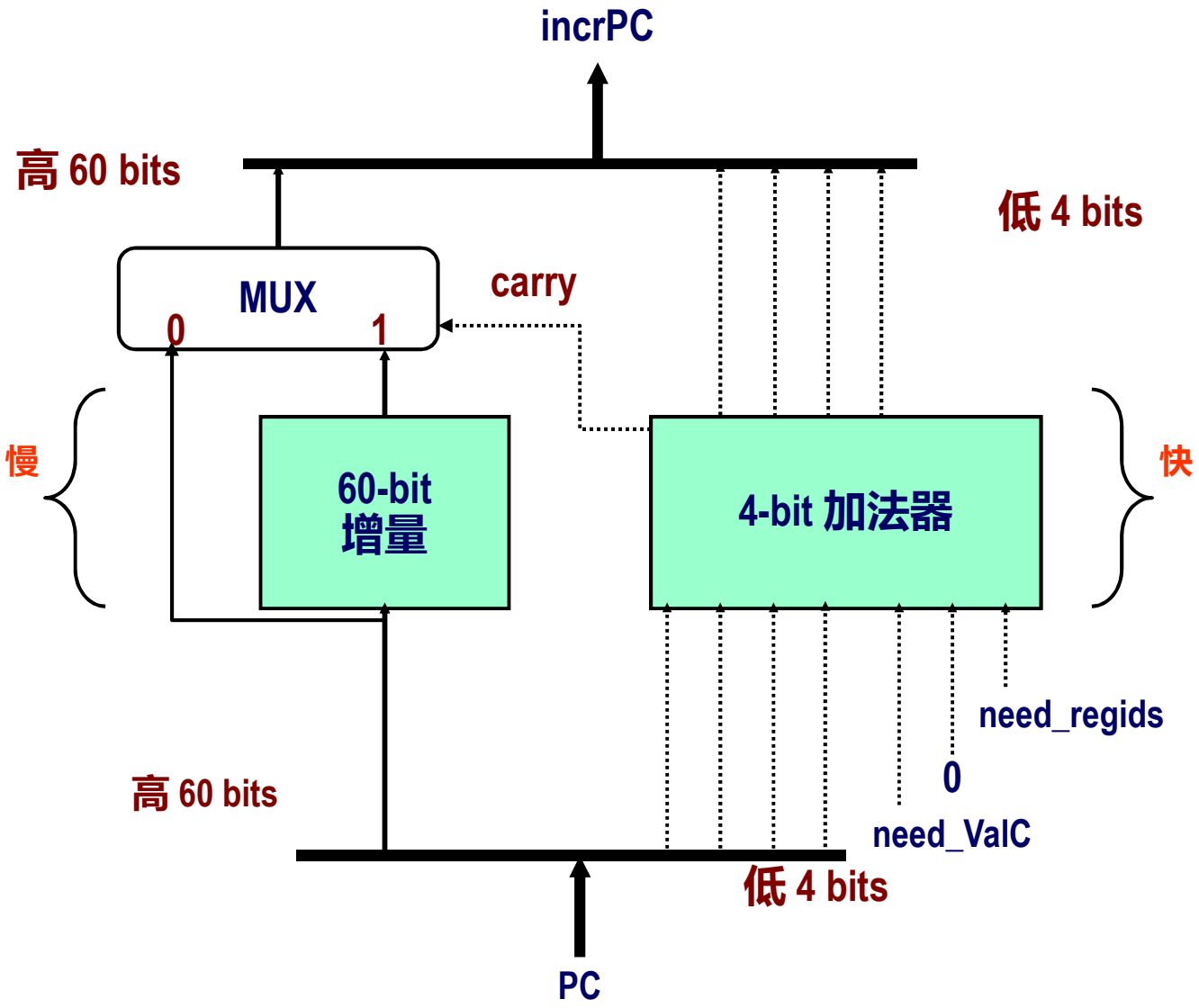
标准取指时序



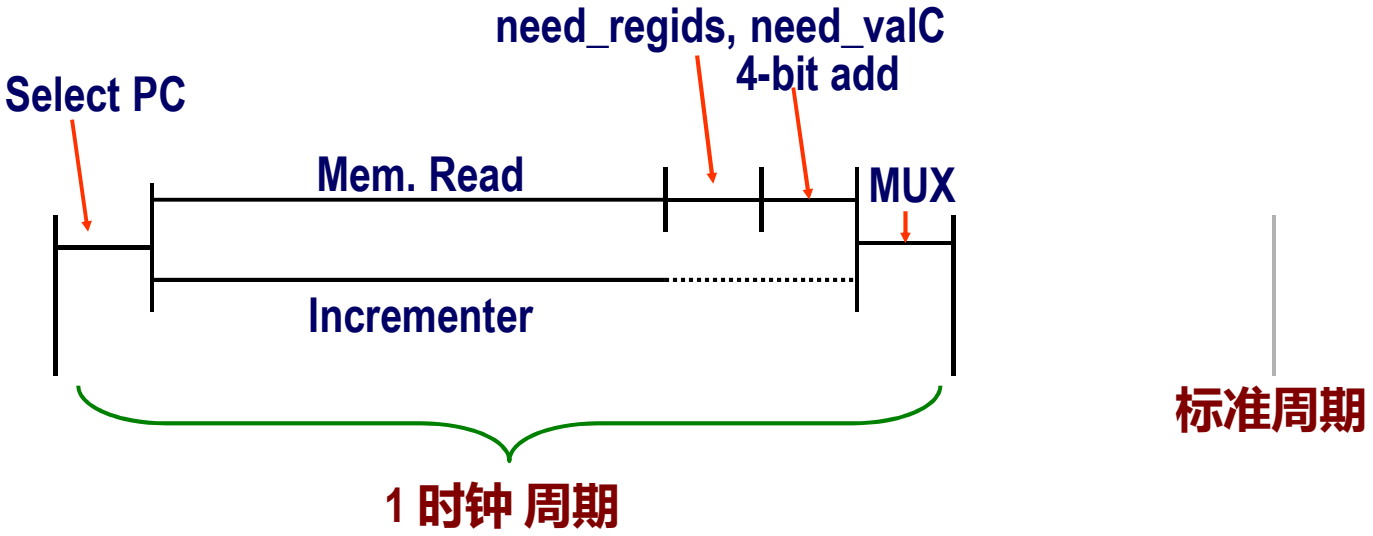
1 时钟 周期

- 确保每一部分都顺序执行
- 在确定PC增加多少之后才能计算PC的值

快速增加PC的电路

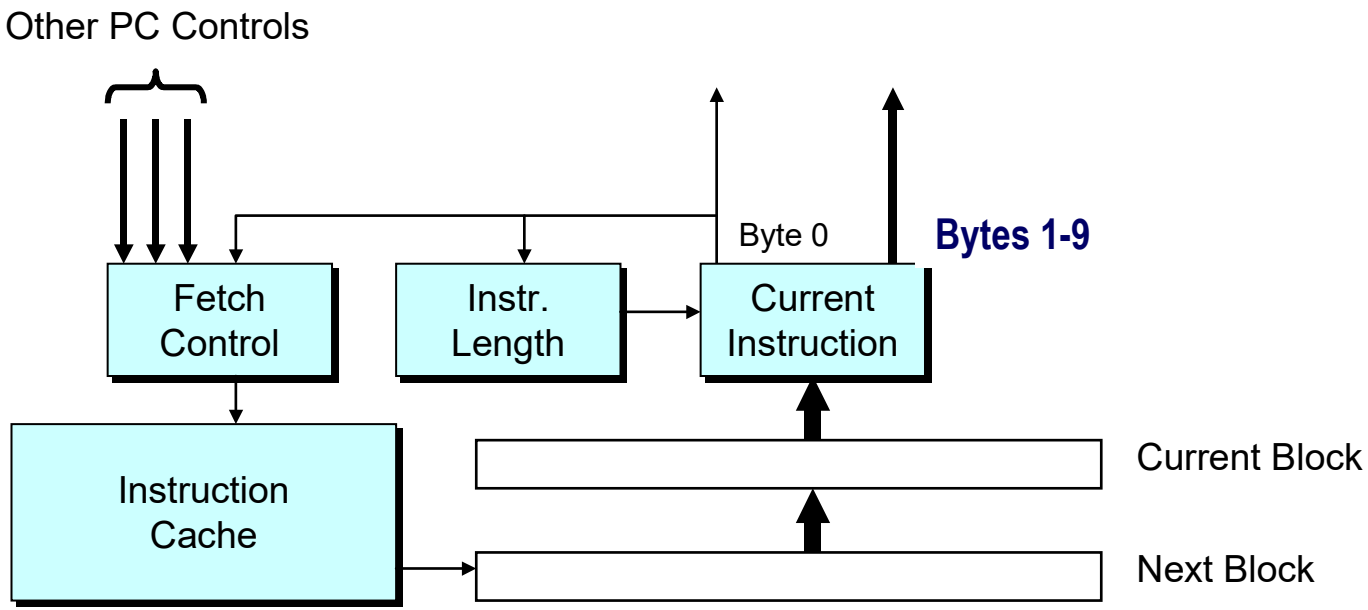

incrPC
高 60 bits
低 4 bits
MUX
0
1
carry
慢
60-bit
增量
4-bit 加法器
快
need_regids
0
need_ValC
高 60 bits
低 4 bits
PC

调整取指时间



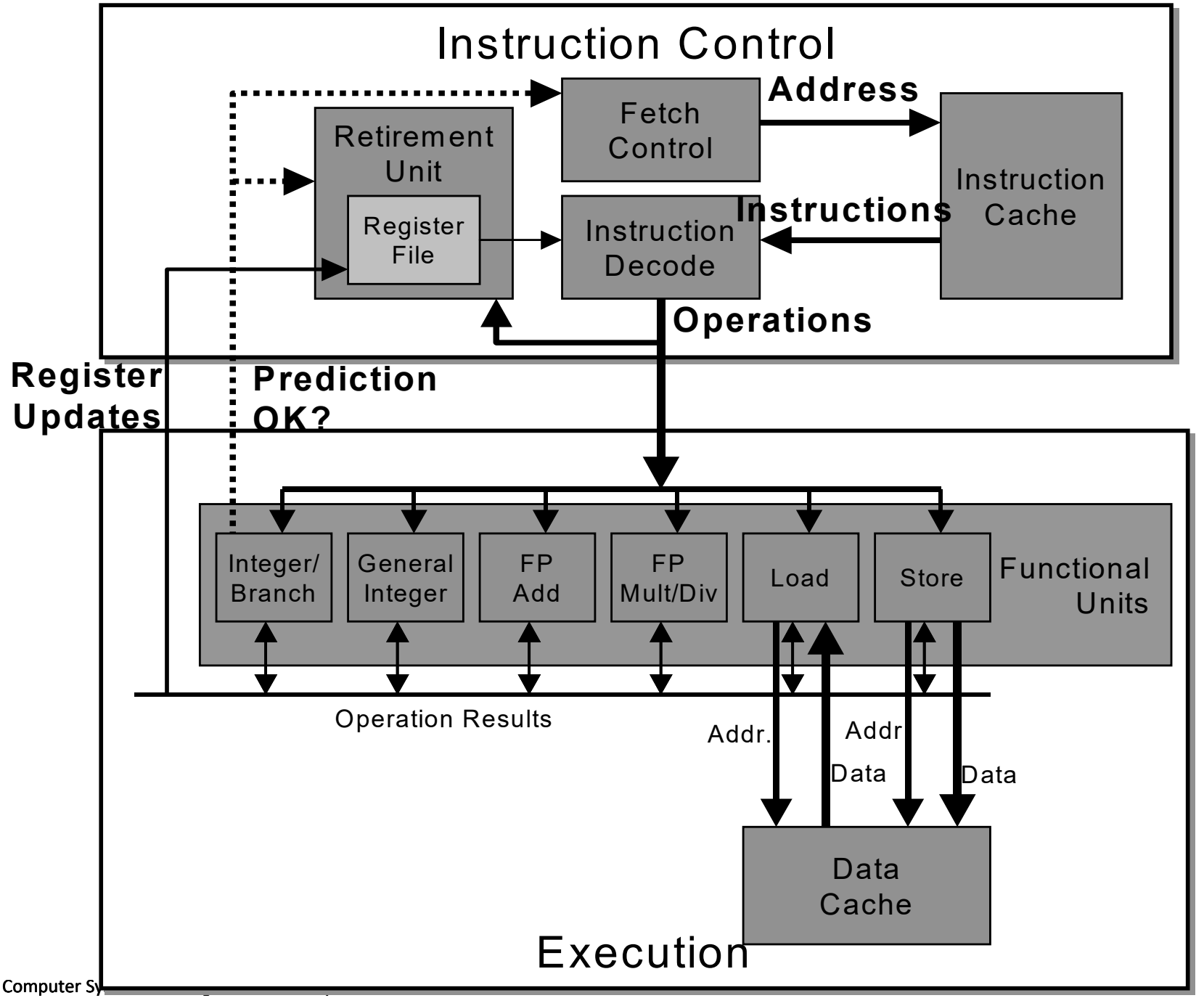
- 60-Bit 增量
  - 当PC被选择时立即执行
  - 不必等到MUX再输出
  - 和存储器读并行执行

更实际的取指逻辑

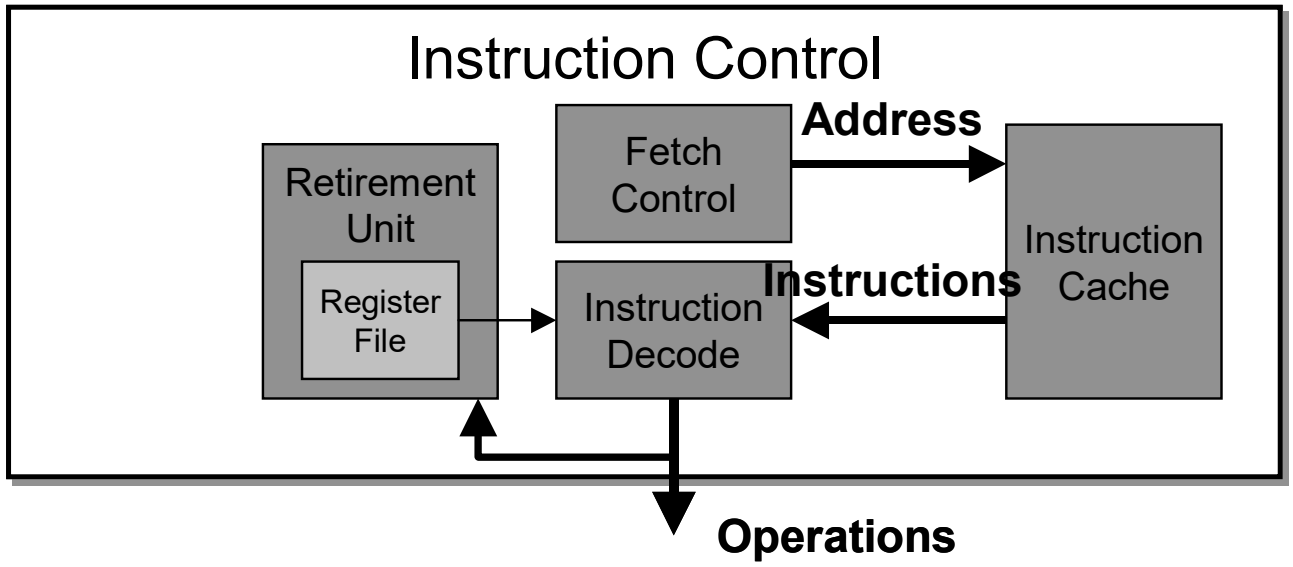


- 取指盒子
  - 集成到指令缓存
  - 取整个cache块 (16 or 32 bytes)
  - 从当前块选择当前指令
  - 提前获取下一个块
    - 当到达当前块的末尾
    - 或在分支目标处

现代CPU设计

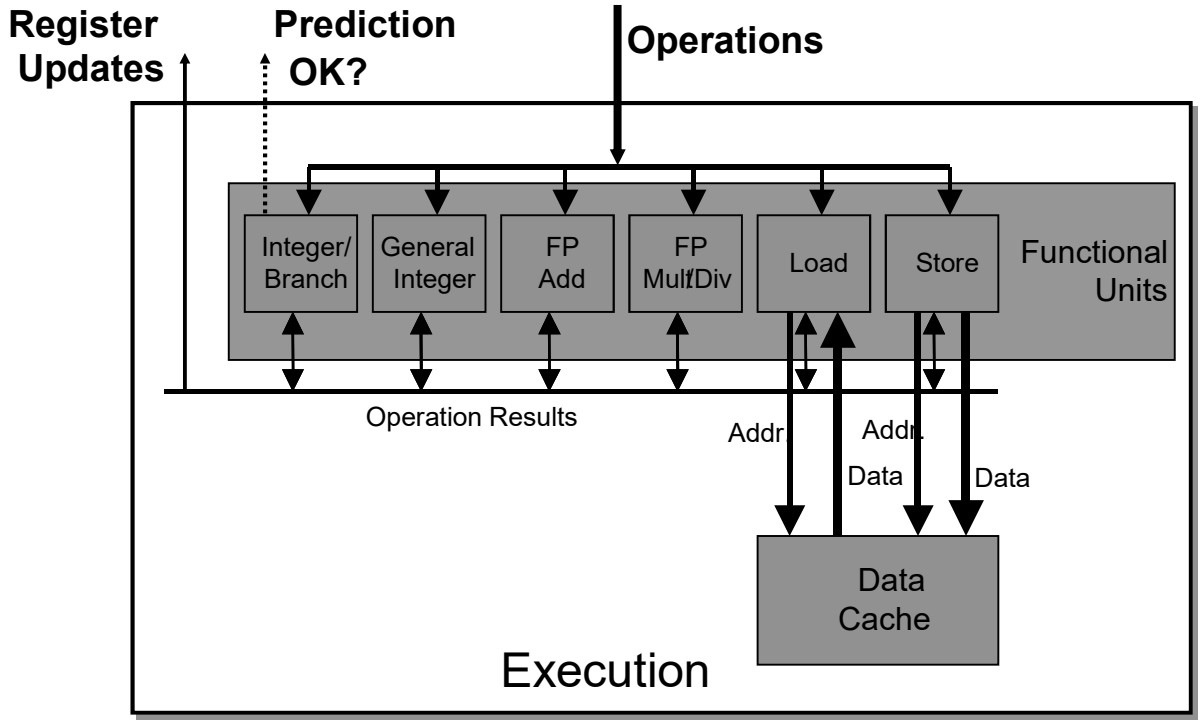

Instruction Control
Retirement Unit
Register File
Fetch Control
Address
Instruction Cache
Instructions
Instruction Decode
Operations
Register Updates
Prediction OK?
Integer/ Branch
General Integer
FP Add
FP Mult/Div
Load
Store
Functional Units
Operation Results
Addr.
Addr
Data
Data
Data Cache
Execution

指令控制



- 从内存中获取指令字节
  - 当前PC值加预测目标得到预测分支
  - 使用硬件动态猜测是否采取/不采取分支
- 将指令转换为操作
  - 指令执行所需的基本步骤
  - 典型指令需要1-3个操作
- 将寄存器转换为标签
  - 抽象的标识符将一个操作的目的和后一个操作的源相连接

执行单元



- 多功能单元
  - 每一个可以独立操作
- 一旦操作数就绪操作就可以执行
  - 不一定依据程序顺序执行
  - 受限于功能单元
- 控制逻辑
  - 确保执行结果和顺序执行一样

Intel Haswell CPU的能力

- 多条指令可以并行执行
  - 2 load（加载)
  - 1 store（存储)
  - 4 integer（整型)
  - 2 FP multiply（浮点乘)
  - 1 FP add / divide（浮点加/除)
- 有些指令花费多于一个周期，但是仍可以流水化

指令	延迟	周期数（周期s/Issue)
Load / Store	4	1
Integer Multiply	3	1
Integer Divide	3—30	3—30
Double/Single FP Multiply	5	1
Double/Single FP Add	3	1
Double/Single FP Divide	10—15	6—11

Haswell 操作

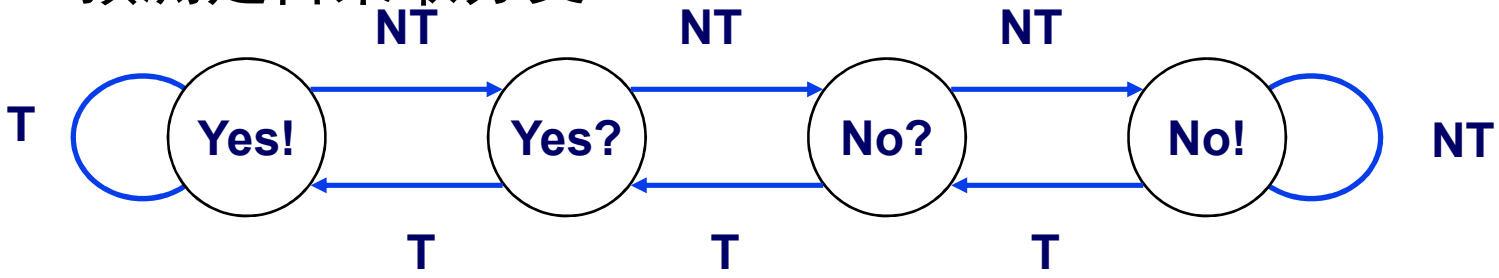
- 将指令动态转换为 “Uops”
  - ~118 bits 宽
  - 保持操作，两个源，一个目的
- 使用乱序引擎执行Uops
  - 执行Uop当
    - 操作数可用
    - 功能单元可用
  - 执行由“预约站”控制
    - 跟踪uops之间的数据相关
    - 分配资源

高性能分支预测

- 影响性能的关键
  - 处理预测错误通常需要11-15个周期
- 分支目标缓存
  - 512 个目的地址
  - 4 bits 用于历史信息
  - 自适应算法
    - 可以识别重复的模式，例如交替跳转或不跳转
- 处理BTB未命中
  - 在第六个周期检测
  - 负偏移地址时采用预测，正偏移时不采用预测
    - 循环vs条件

分支预测示例

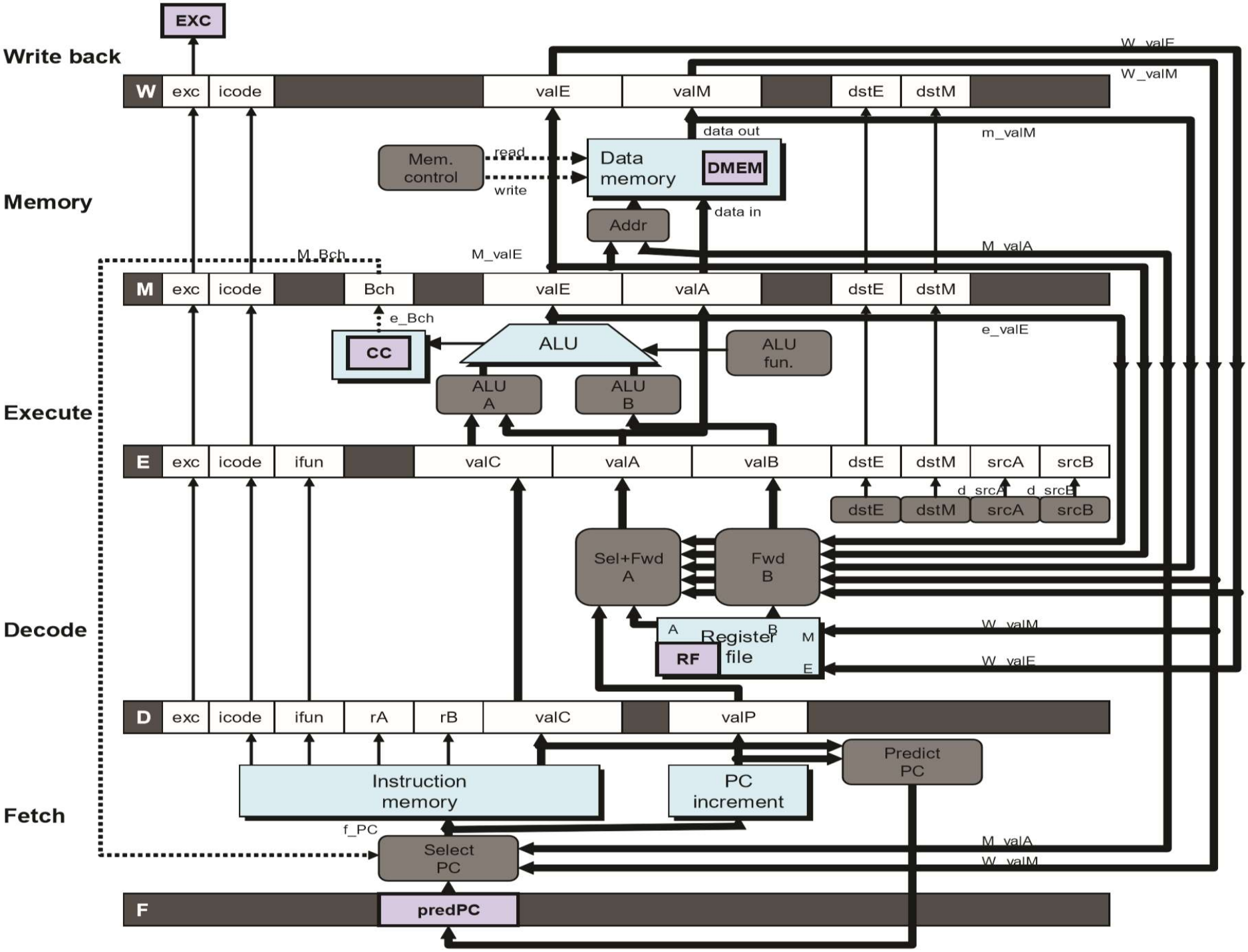
- 分支历史
  - 编码分支指令先前的历史信息
  - 预测是否采取分支



- 状态机
  - 每次采取分支后，向右过渡
  - 不采取则向左过渡
  - 在状态“Yes！”或“Yes?”下，预测采取分支

处理器总结

- 设计技术
  - 对所有的指令建立统一的框架
    - 便于在指令之间共享硬件
  - 将标准逻辑块与控制逻辑位连接起来
- 操作
  - 状态被保存在存储器或时钟寄存器
  - 组合逻辑进行计算
  - 寄存器/存储器时钟用于控制整体的行为
- 提高性能
  - 流水化提高了吞吐量和资源利用率
  - 必须保证服从ISA行为


EXC
Write back
W
exc
icode
valE
valM
dstE
dstM
W_valE
W_valM
data out
m_valM
Mem. control
read
write
Data memory
DMEM
data in
Addr
Memory
M_Bch
M_valE
M_valA
M
exc
icode
Bch
valE
valA
dstE
dstM
e_Bch
e_valE
CC
ALU
ALU fun.
ALU A
ALU B
Execute
E
exc
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
Sel+Fwd A
Fwd B
Register file
RF
A
B
M
E
W_valM
W_valE
Decode
D
exc
icode
ifun
rA
rB
valC
valP
Predict PC
Instruction memory
PC increment
f_PC
Select PC
M_valA
W_valM
Fetch
F
predPC