

# Pipelined Implementation (3)

Lecture 13

November 6<sup>th</sup>, 2018

Jae W. Lee ([jaewlee@snu.ac.kr](mailto:jaewlee@snu.ac.kr))

Computer Science and Engineering  
Seoul National University

***Slide credits:*** [CS:APP3e] slides from CMU; [COD5e] slides from Elsevier Inc.

# Today

**Textbook: [CS:APP3e] 4.5.5, 4.5.7, and 4.5.8**

- **Pipeline Hazards (Cont)**
- **PIPE Stage Implementations**
- **PIPE Control Logic**

# Overview

***Make the pipelined processor work!***

## ■ Data Hazards

- Instruction having register R as source follows shortly after instruction having register R as destination
- Common condition, don't want to slow down pipeline

## ■ Control Hazards

- Mispredicted conditional branch
  - Our design predicts all branches as being taken
  - Naïve pipeline executes two extra instructions
- Getting return address for `ret` instruction
  - Naïve pipeline executes three extra instructions

## ■ Making Sure It Really Works

- What if multiple special cases happen simultaneously?

# Pipeline Stages

## Fetch

- Select current PC
- Read instruction
- Compute incremented PC

## Decode

- Read program registers

## Execute

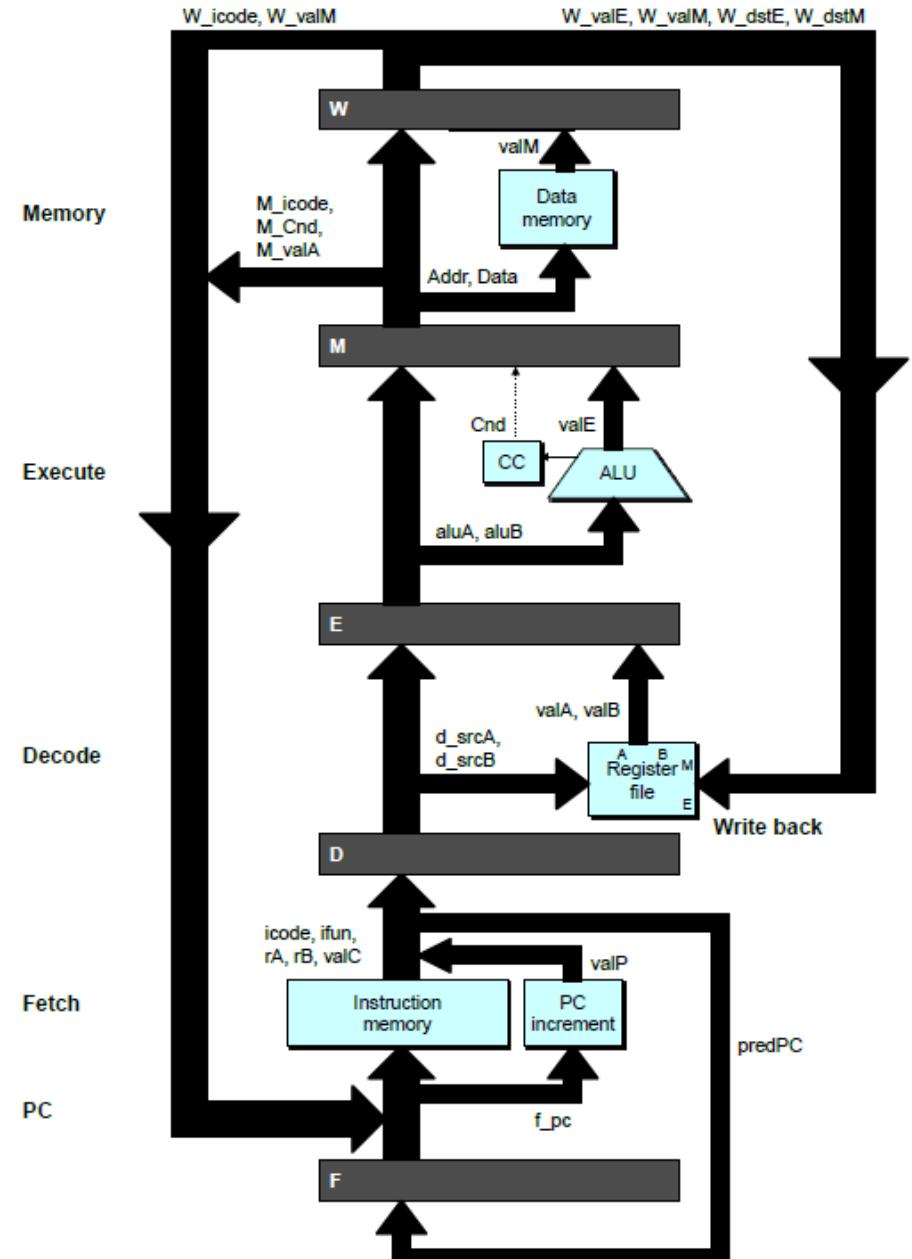
- Operate ALU

## Memory

- Read or write data memory

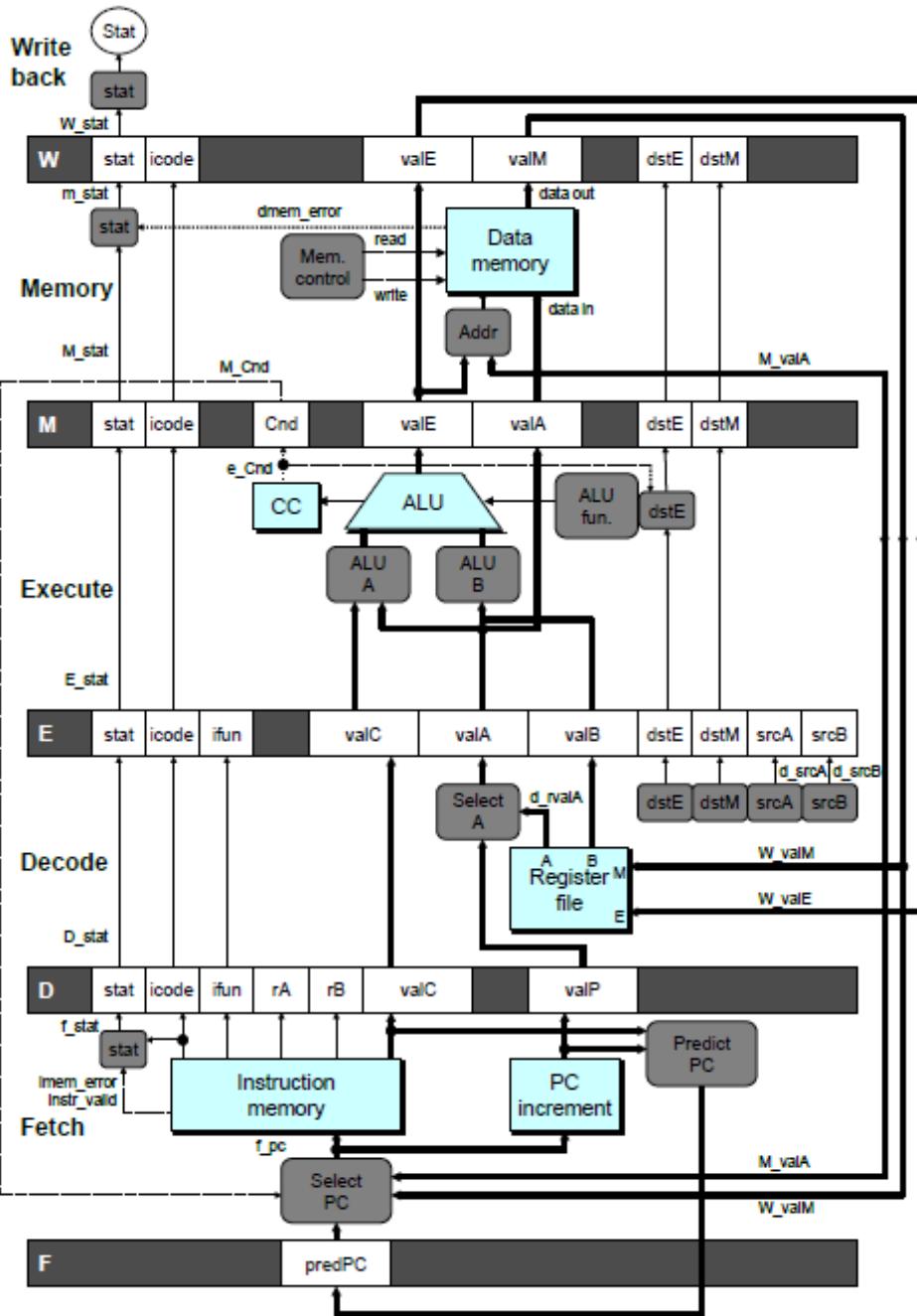
## Write Back

- Update register file



# **PIPE- Hardware**

- Pipeline registers hold intermediate values from instruction execution
  - **Forward (Upward) Paths**
    - Values passed from one stage to next
    - Cannot jump past stages
      - e.g., valC passes through decode

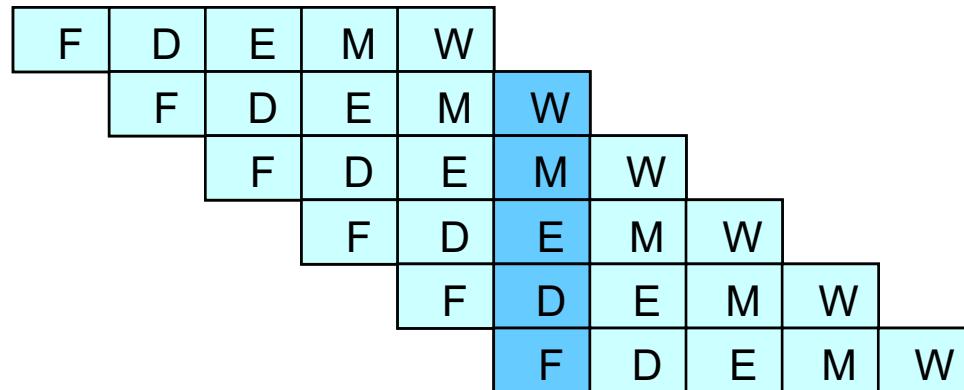


# Data Dependencies: 2 Nop's

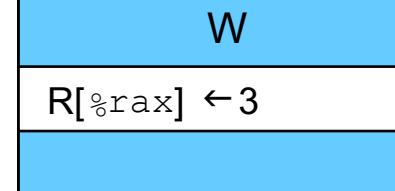
# demo-h2.ys

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

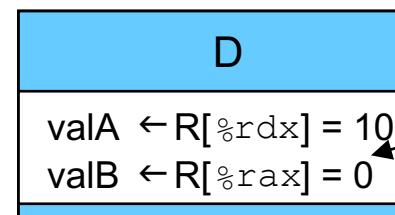
1 2 3 4 5 6 7 8 9 10



Cycle 6



•  
•  
•

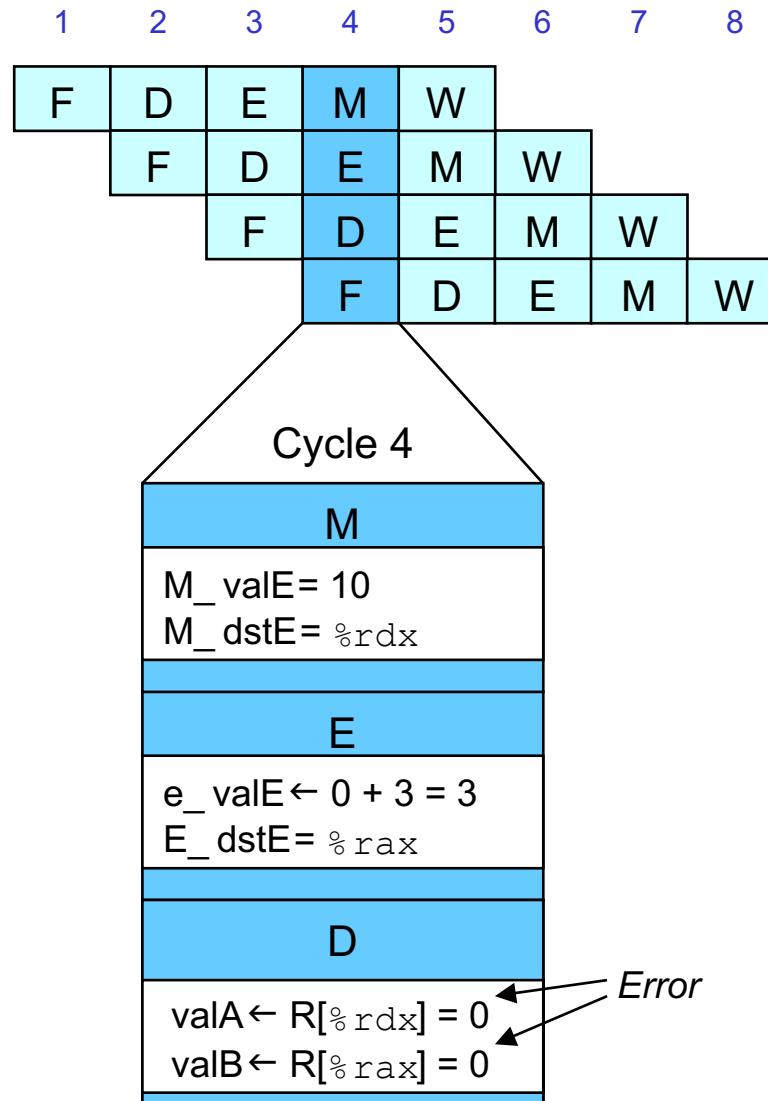


Error

# Data Dependencies: No Nop

# demo-h0.ys

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



# Stalling for Data Dependencies

# demo-h2.ys

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

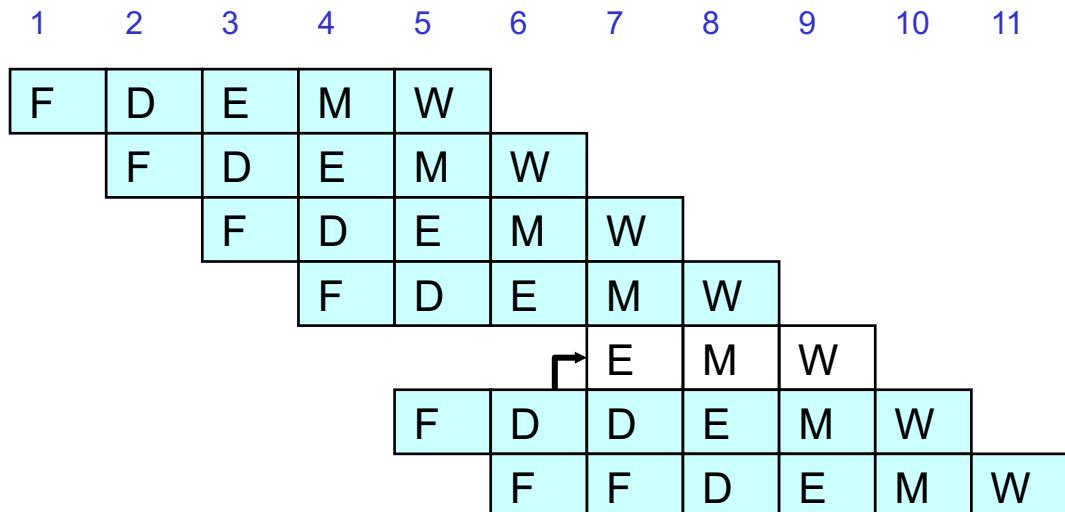
0x014: nop

0x015: nop

**bubble**

0x016: addq %rdx,%rax

0x018: halt



- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

# Stall Condition

# ■ Source Registers

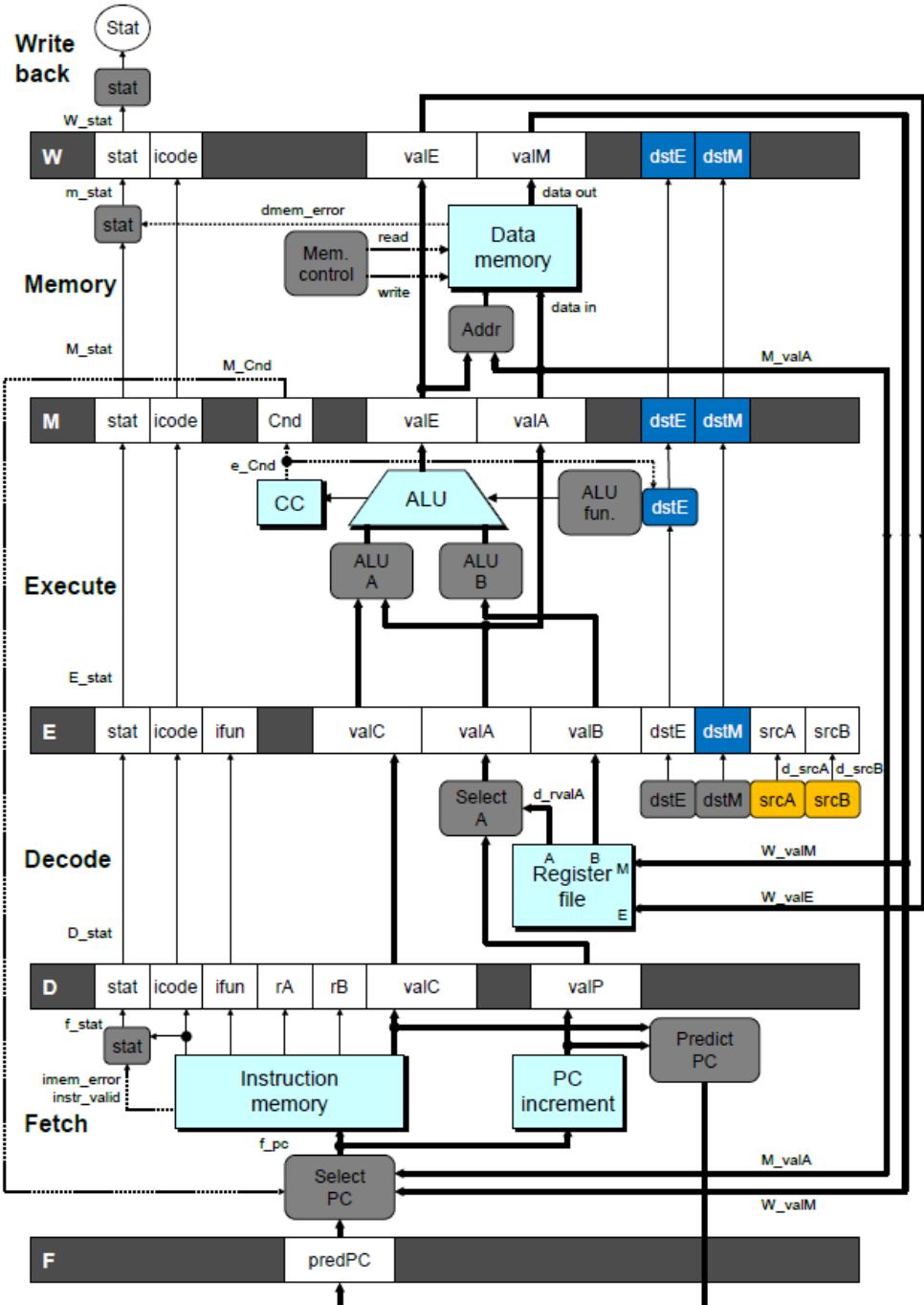
- srcA and srcB of current instruction in decode stage

## ■ Destination Registers

- dstE and dstM fields
  - Instructions in execute, memory, and write-back stages

## ■ Special Case

- Don't stall for register ID 15 (0xF)
    - Indicates absence of register operand
    - Or failed cond. move



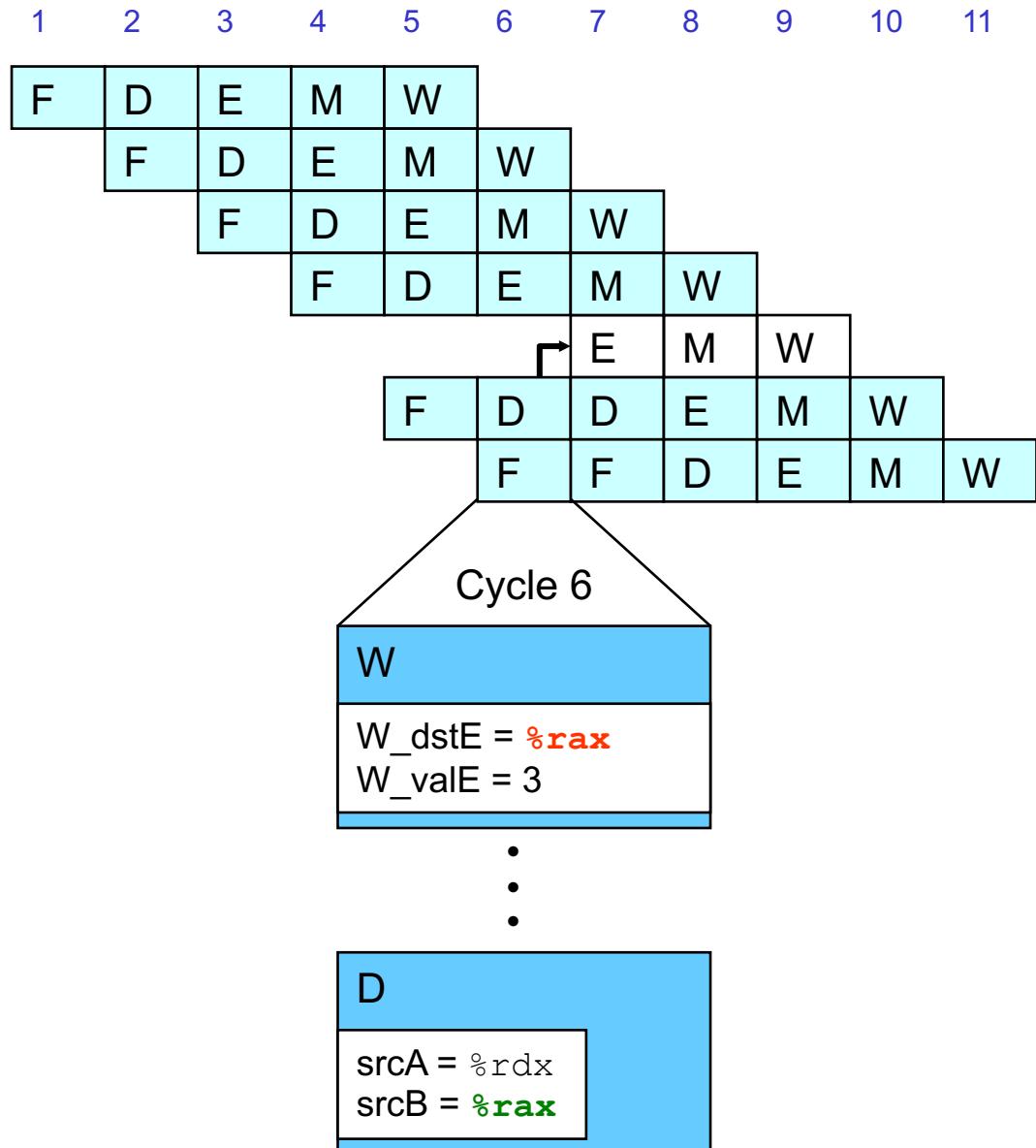
# Detecting Stall Condition

# demo-h2.ys

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

**bubble**

```
0x016: addq %rdx,%rax
0x018: halt
```



# Stalling X3

# demo-h0.ys

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

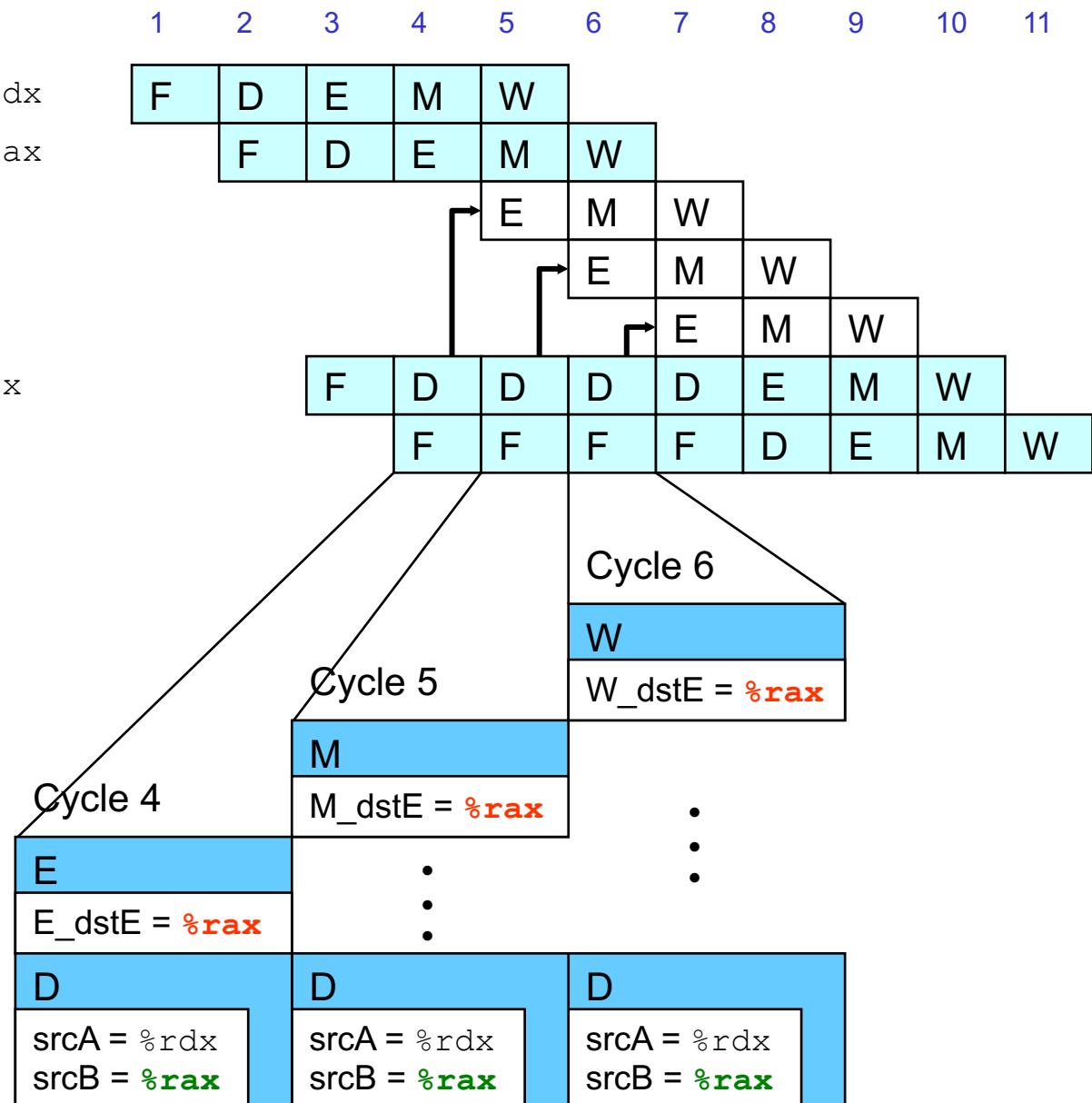
*bubble*

*bubble*

*bubble*

0x014: addq %rdx,%rax

0x016: halt



# What Happens When Stalling?

```
# demo-h0.ys
```

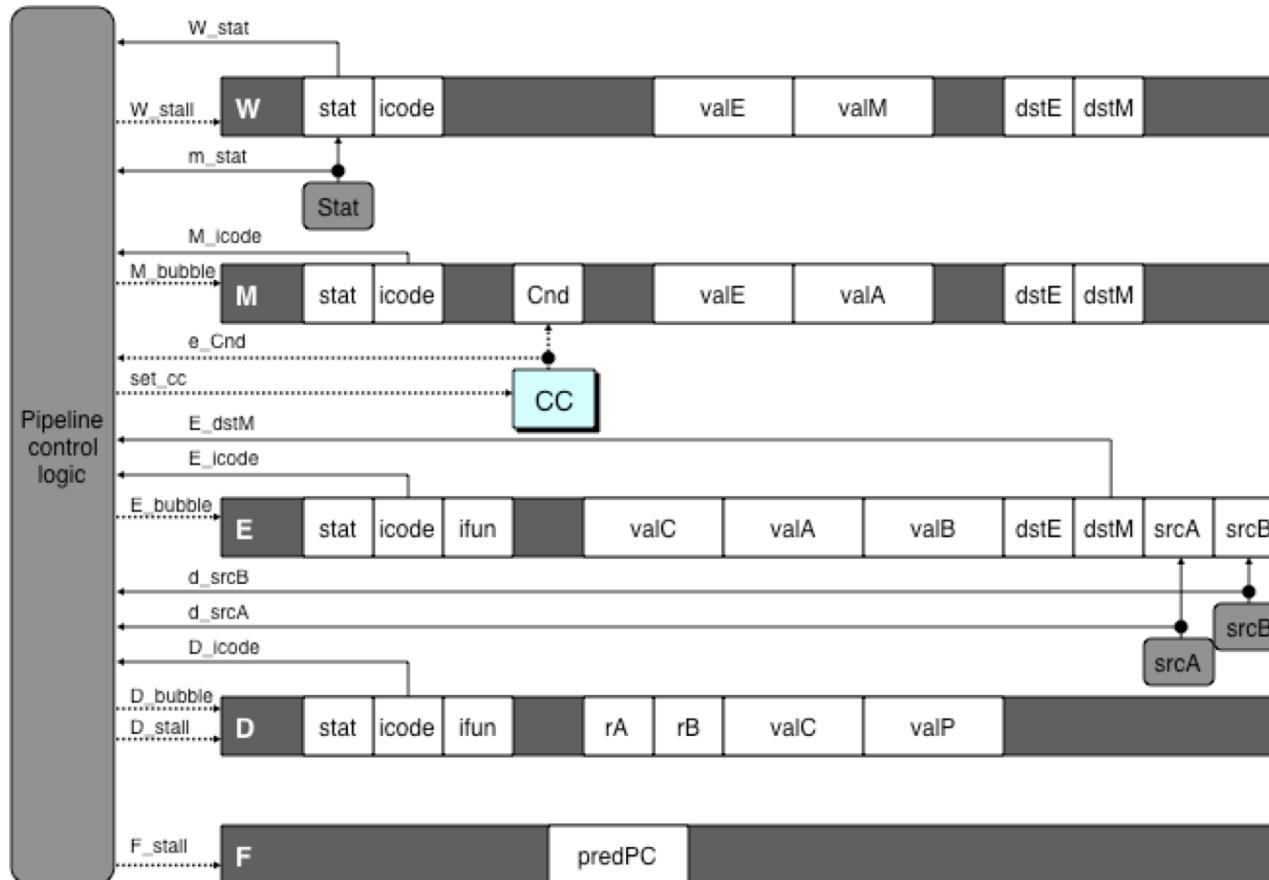
```
0x000: irmovq $10,%rdx  
0x00a: irmovq $3,%rax  
0x014: addq %rdx,%rax  
0x016: halt
```

Cycle 8

Write Back	<i>bubble</i>
Memory	<i>bubble</i>
Execute	0x014: addq %rdx,%rax
Decode	0x016: halt
Fetch	

- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
  - Like dynamically generated nop's
  - Move through later stages

# Implementing Stalling

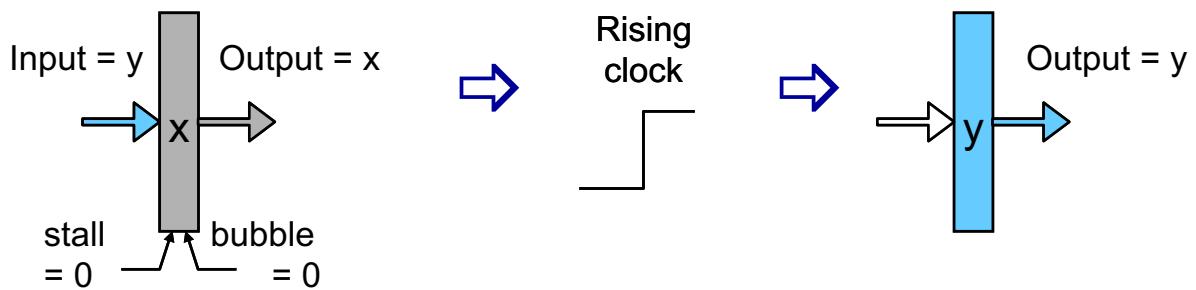


## ■ Pipeline Control

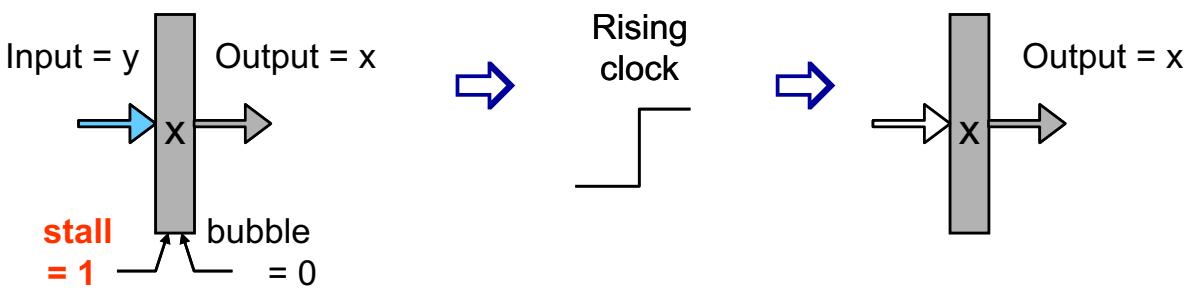
- Combinational logic detects stall condition
- Sets mode signals for how pipeline registers should be updated

# Pipeline Register Modes

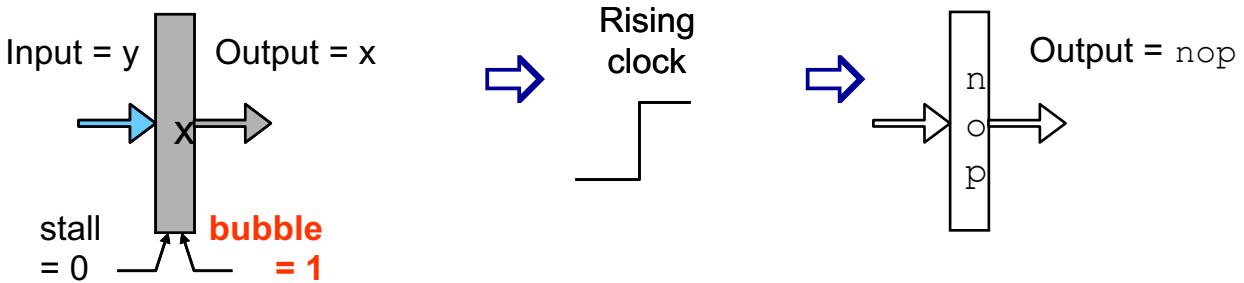
## Normal



## Stall



## Bubble



# Data Forwarding

## ■ Naïve Pipeline

- Register isn't written until completion of write-back stage
- Source operands read from register file in decode stage
  - Needs to be in register file at start of stage

## ■ Observation

- Value to be written to register generated much earlier (in execute or memory stage)

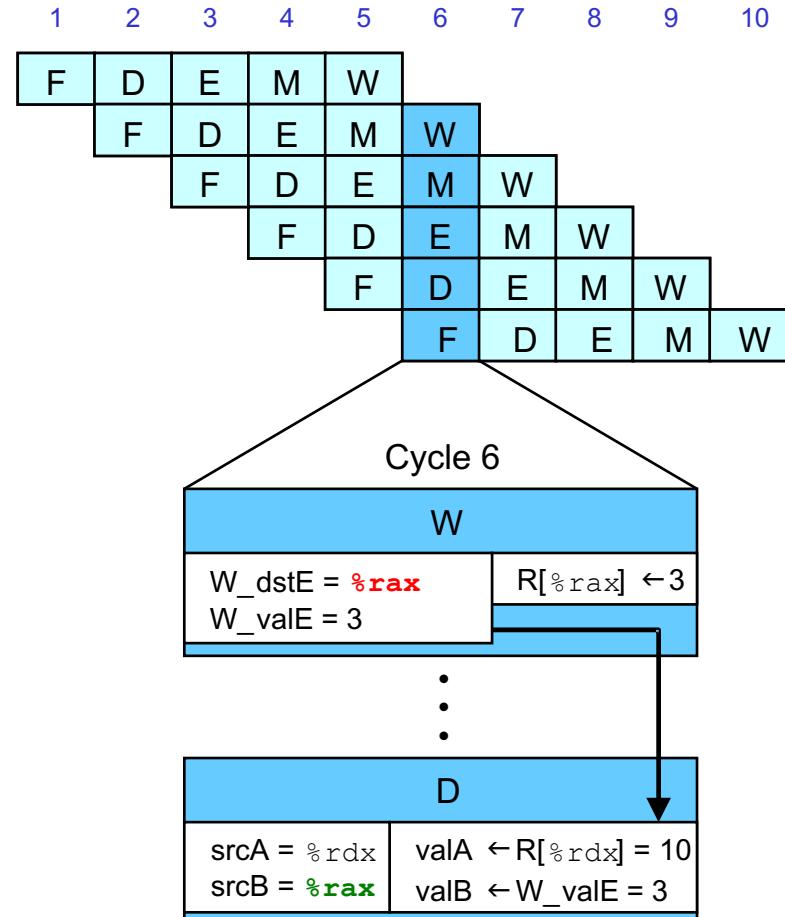
## ■ Trick

- Pass value directly from generating instruction to decode stage
- Needs to be available at end of decode stage

# 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 in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage



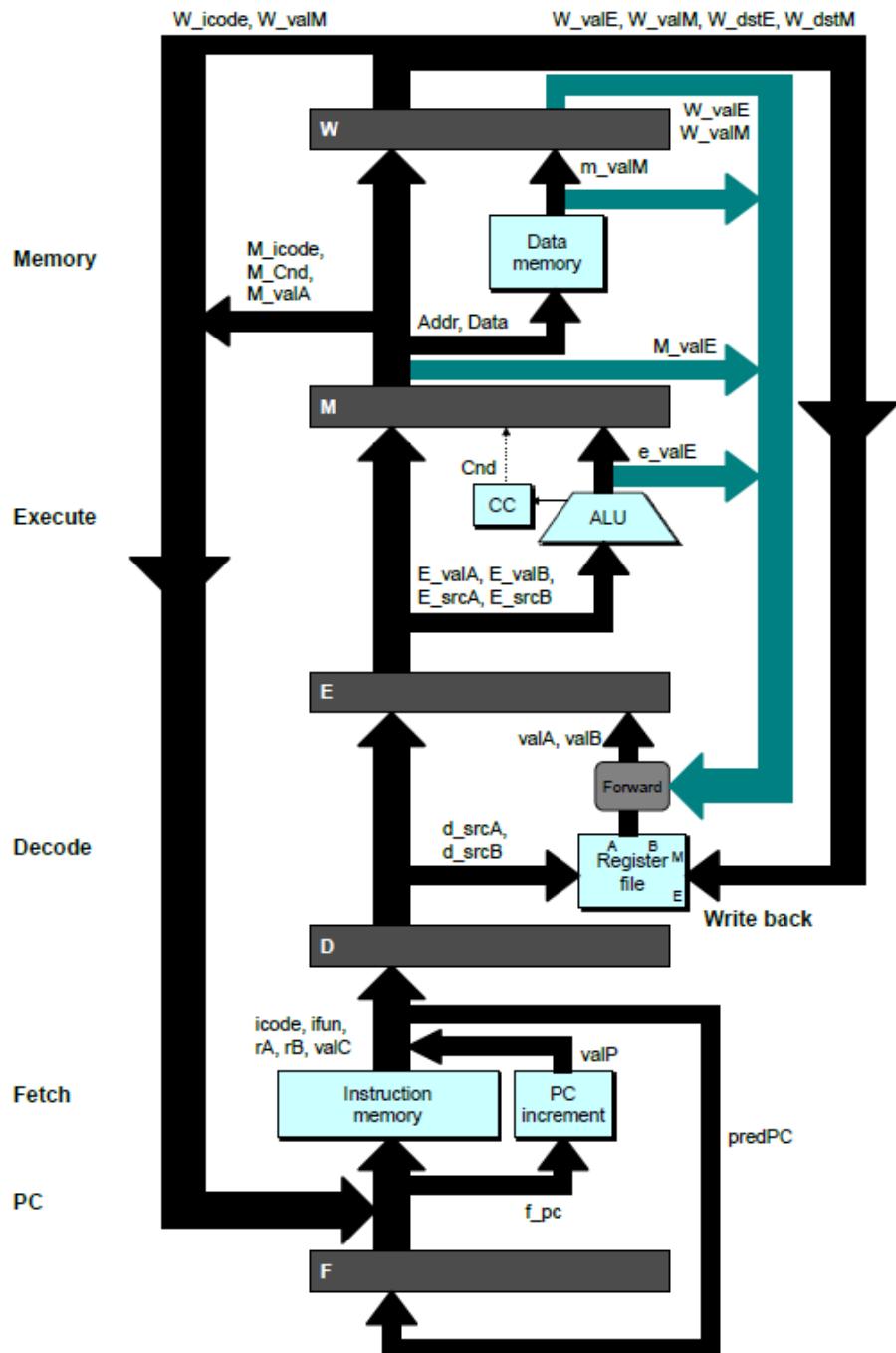
# Bypass Paths

## ■ Decode Stage

- Forwarding logic selects valA and valB
- Normally from register file
- Forwarding: get valA or valB from later pipeline stage

## ■ Forwarding Sources

- Execute: valE
- Memory: valE, valM
- Write back: valE, valM



# Data Forwarding Example #2

# demo-h0.ys

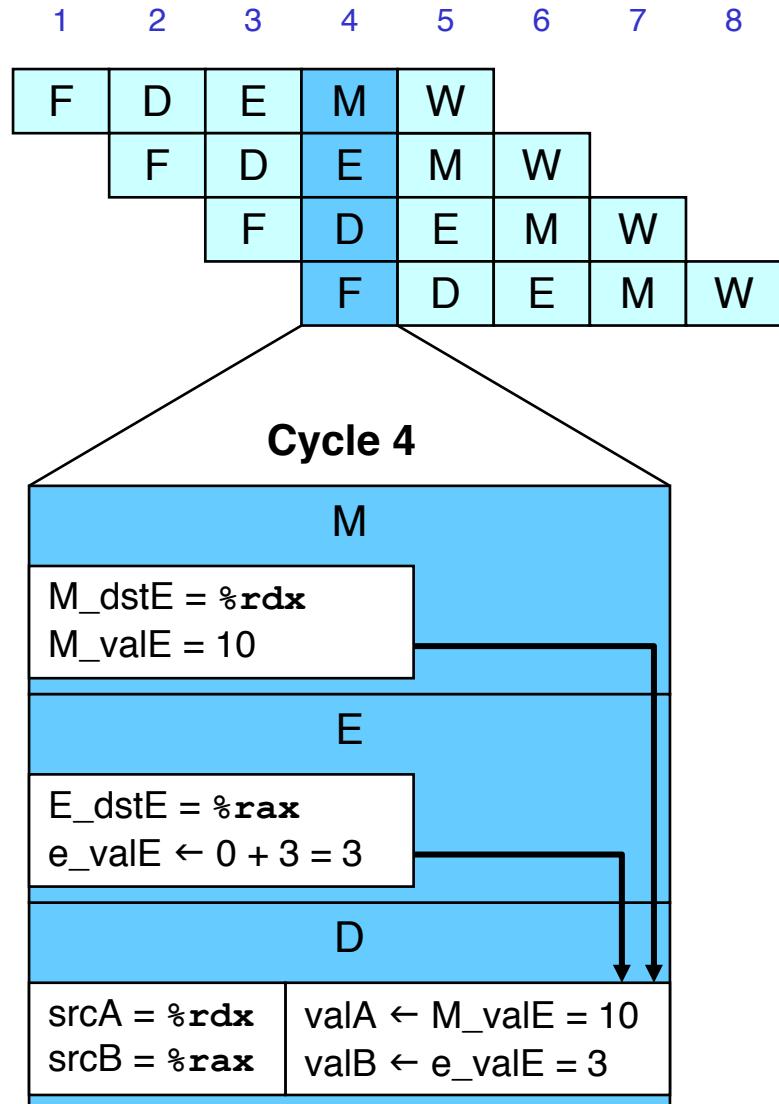
```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

## ■ Register %rdx

- Generated by ALU during previous cycle
- Forward from memory as valA

## ■ Register %rax

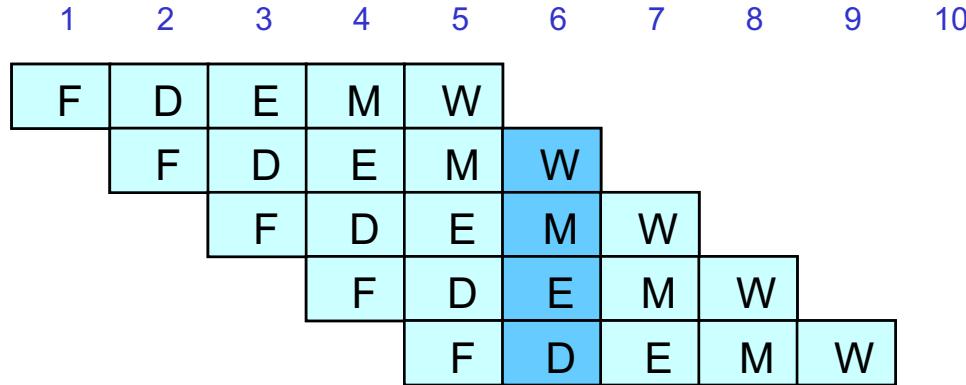
- Value just generated by ALU
- Forward from execute as valB



# Forwarding Priority

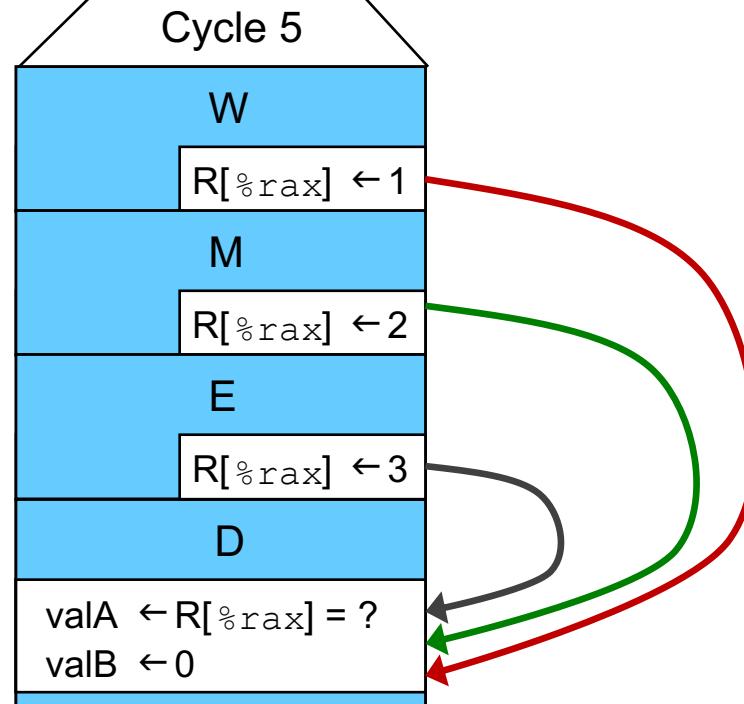
# demo-priority.ys

```
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```



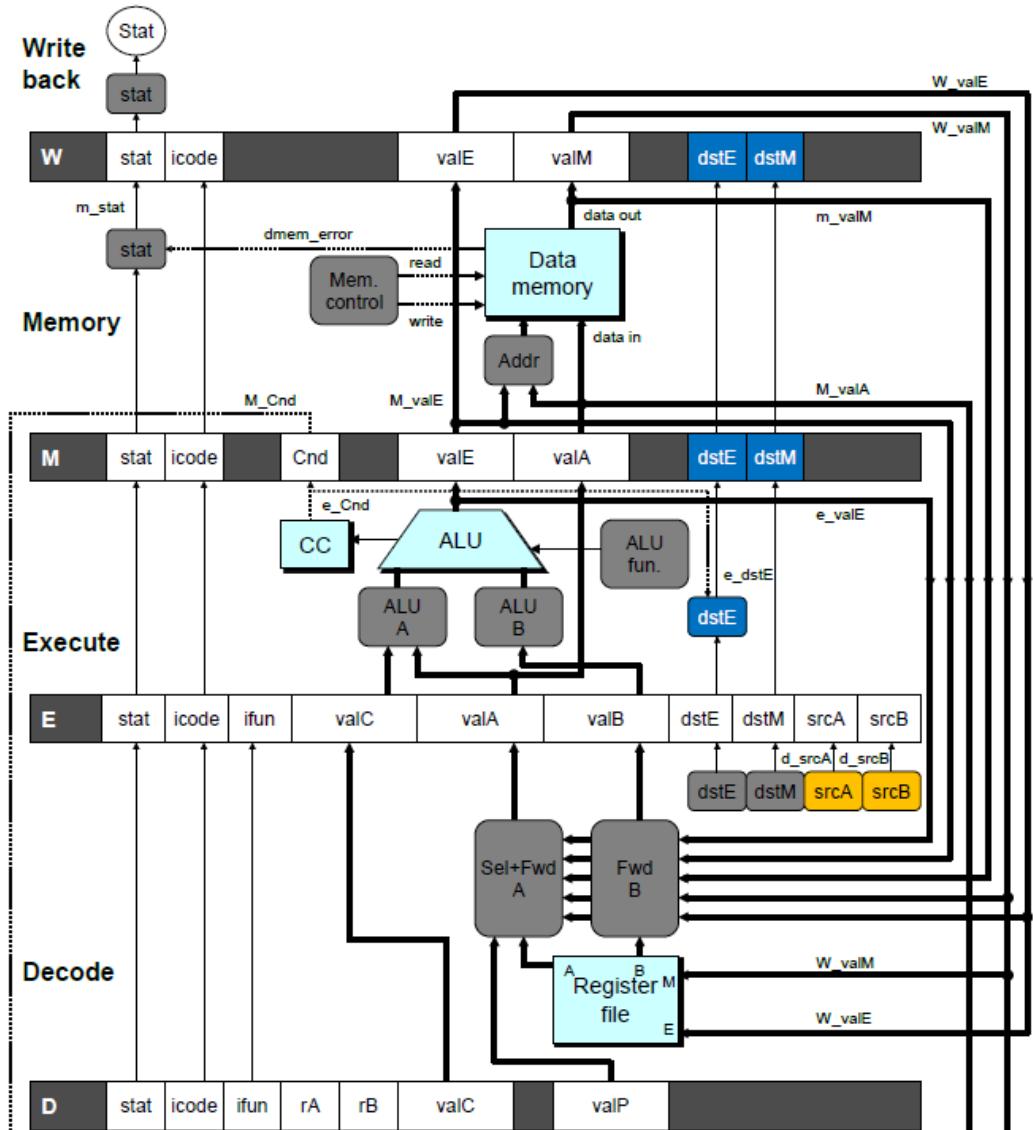
## ■ Multiple Forwarding Choices

- Which one should have priority
- Match serial semantics
- Use matching value from earliest pipeline stage

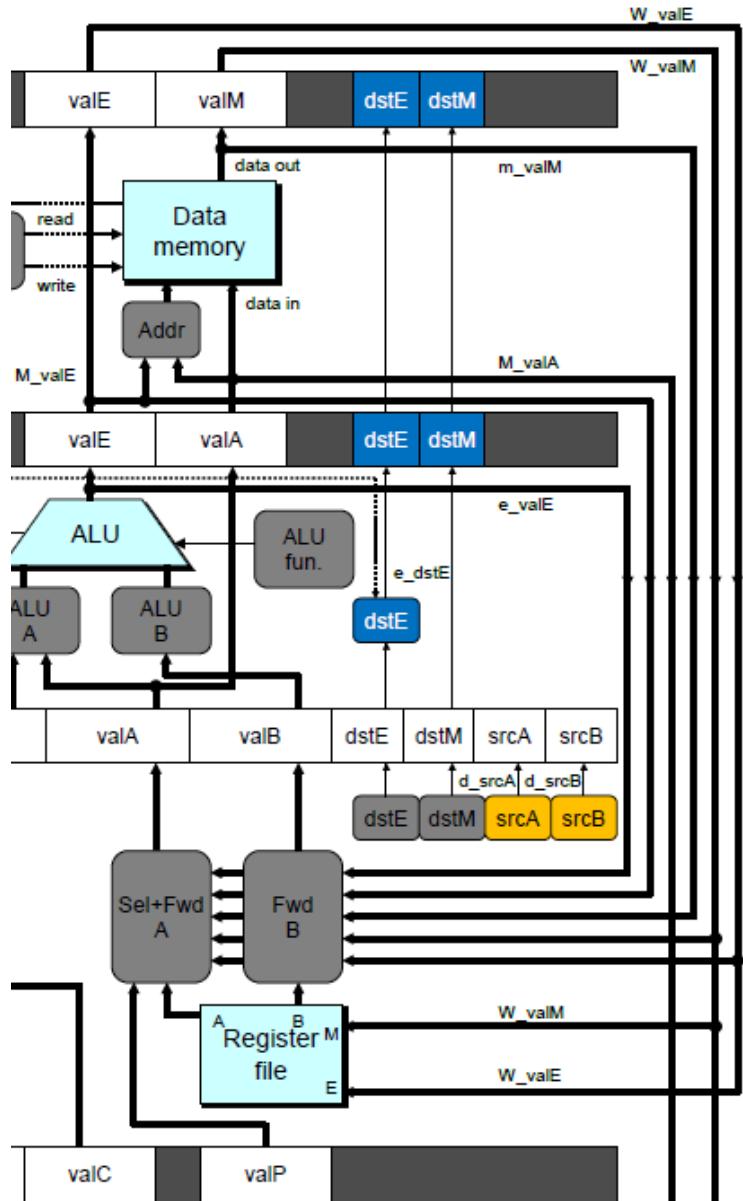


# Implementing Forwarding

- Add additional feedback paths from E, M, and W pipeline registers into decode stage
- Create logic blocks to select from multiple sources for valA and valB in decode stage



# 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;
];

```

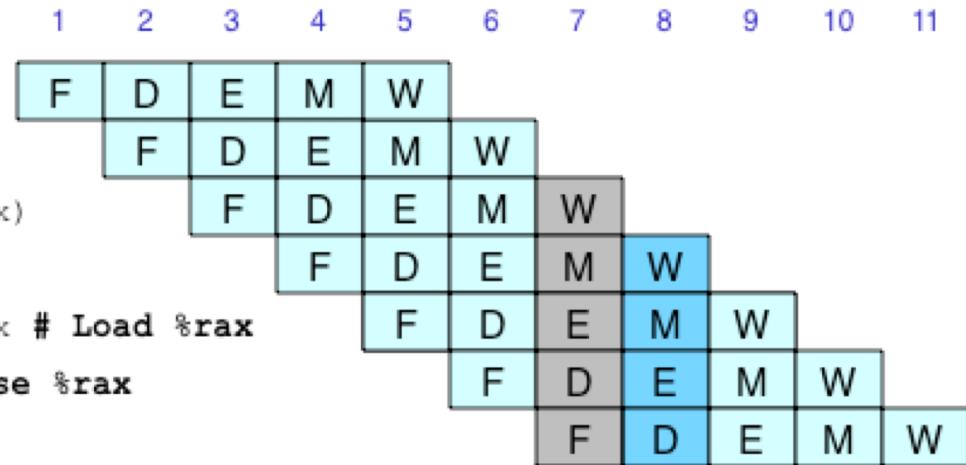
# Limitation of Forwarding

# demo-luh.ys

```

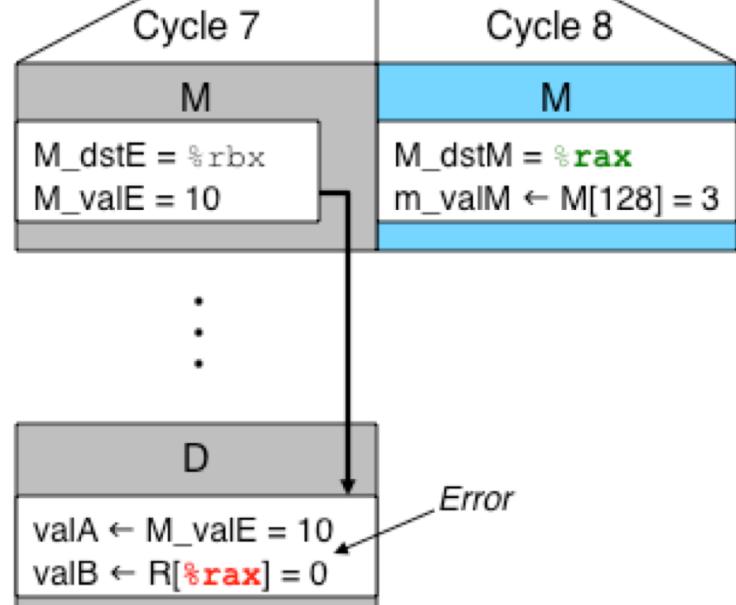
0x000: irmovq $128,%rdx
0x00a: irmovq $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %rbx,%rax # Use %rax
0x034: halt

```



## Load-use dependency

- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8



# Avoiding Load/Use Hazard

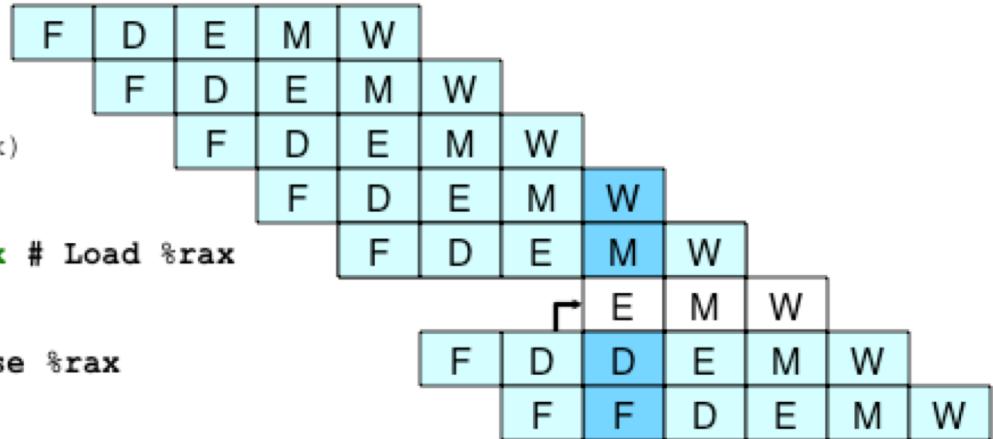
# demo-luh.ys

```
0x000: irmovq $128,%rdx
0x00a: irmovq $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
```

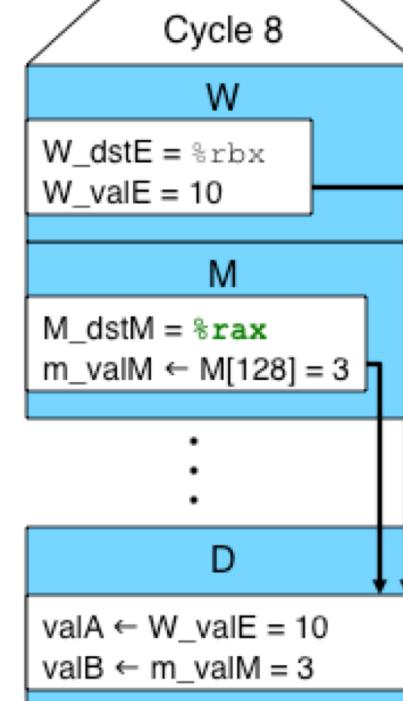
**bubble**

```
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```

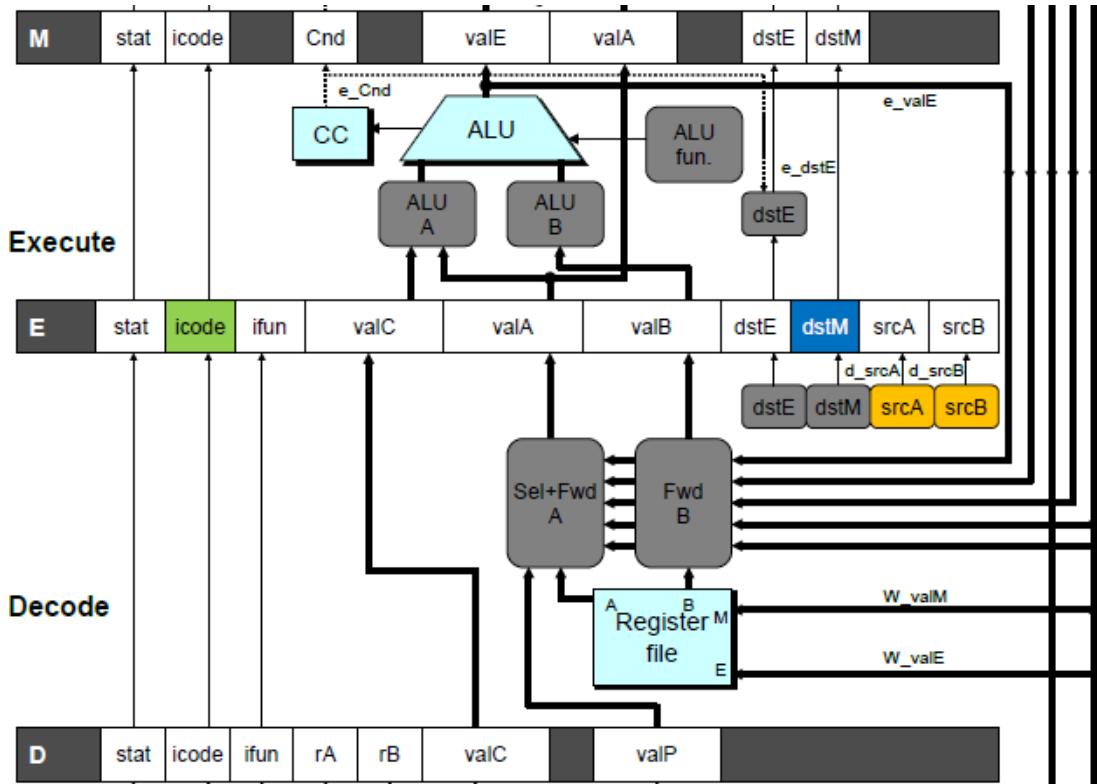
1 2 3 4 5 6 7 8 9 10 11 12



- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage



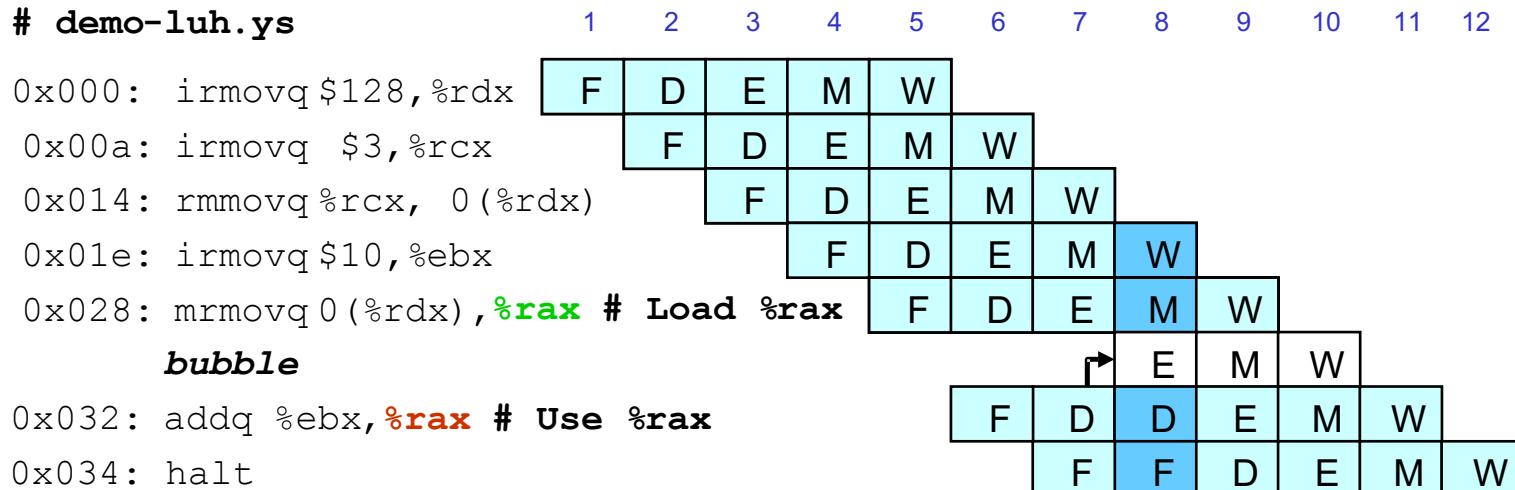
# Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	<pre>E_icode in { IMRMOVQ, IPOPQ } &amp;&amp; E_dstM in { d_srcA, d_srcB }</pre>

# Control for Load/Use Hazard

# demo-luh.ys



- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

# Branch Misprediction Example

demo-j.ys

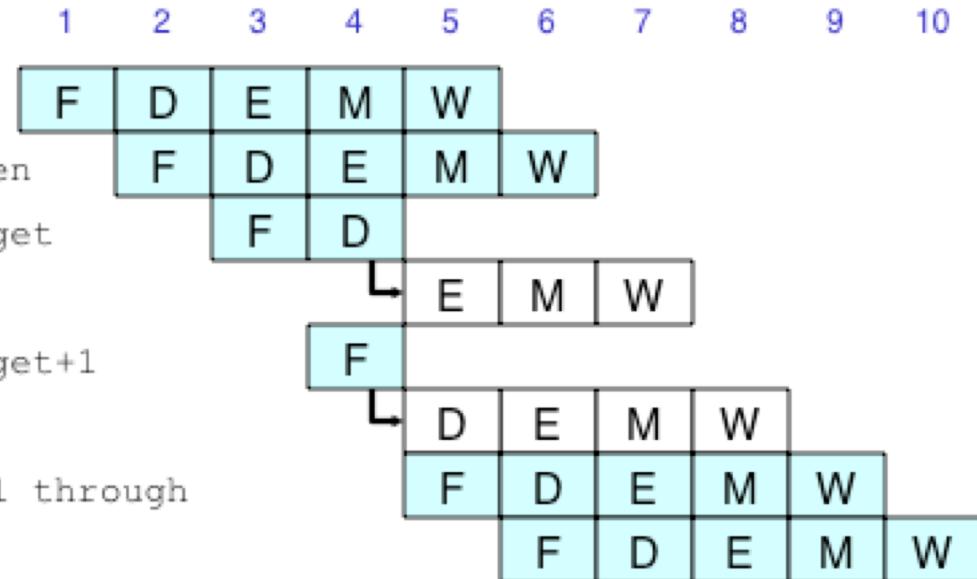
```
0x000: xorq %rax,%rax
0x002: jne t          # Not taken
0x00b: irmovq $1, %rax # Fall through
0x015: nop
0x016: nop
0x017: nop
0x018: halt
0x019: t: irmovq $3, %rdx # Target
0x023: irmovq $4, %rcx  # Should not execute
0x02d: irmovq $5, %rdx  # Should not execute
```

- Should only execute first 8 instructions

# 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
```



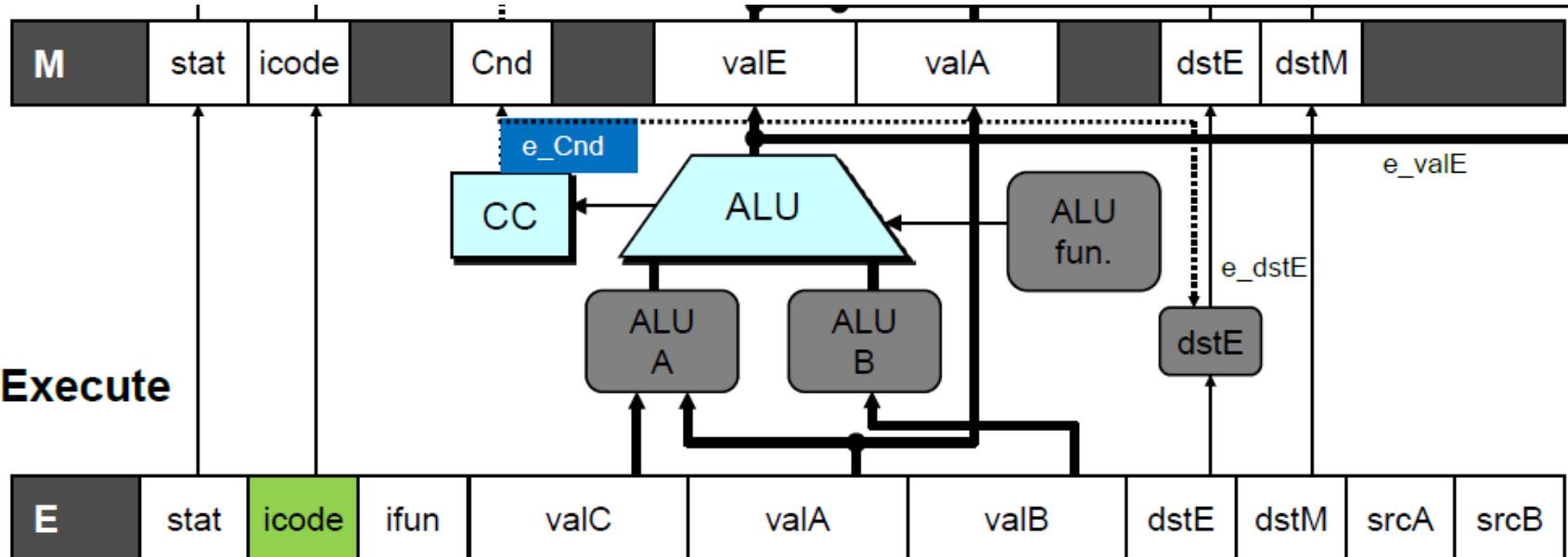
## Predict branch as taken

- Fetch 2 instructions at target

## Cancel when mispredicted

- Detect branch not-taken in execute stage
- On following cycle, replace instructions in execute and decode by bubbles
- No side effects have occurred yet

# Detecting Mispredicted Branch

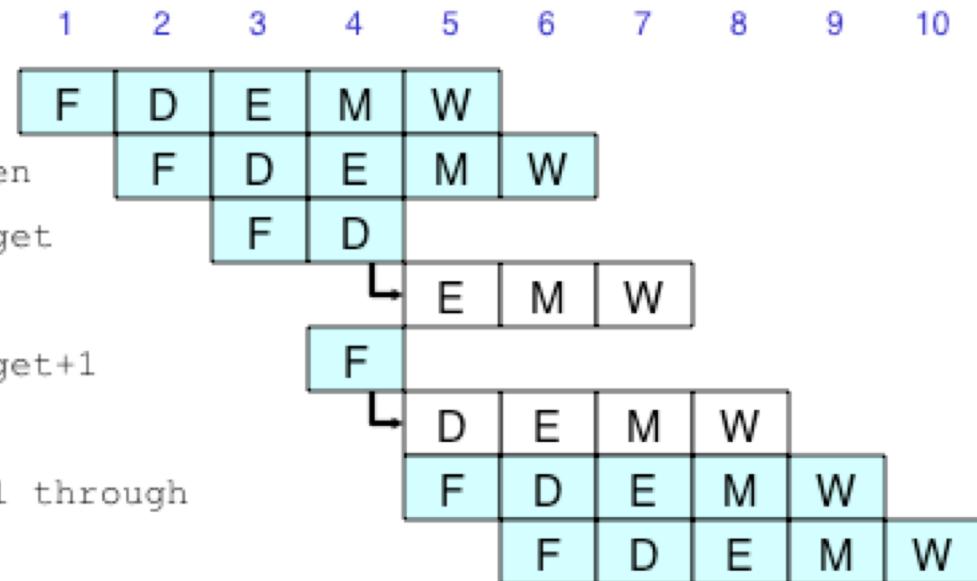


Condition	Trigger
Mispredicted Branch	$E\_icode == IJXX \& !e\_Cnd$

# 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
```



Condition	F	D	E	M	W
Mispredicted Branch	normal	bubble	bubble	normal	normal

**demo-retb.ys**

# Return Example

```
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
```

- Previously executed three additional instructions

# Correct Return Example

```
# demo-retb
```

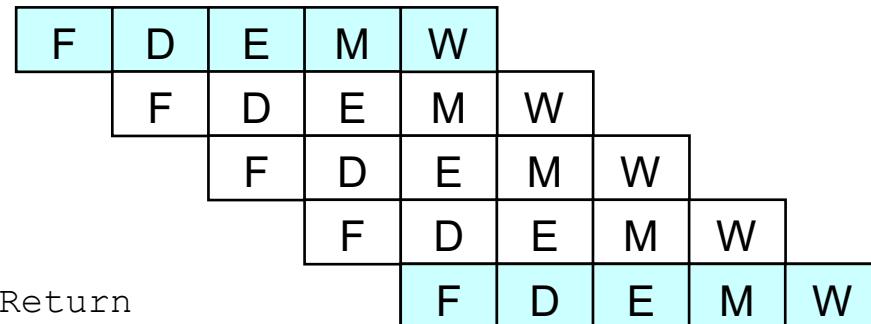
0x026: ret

*bubble*

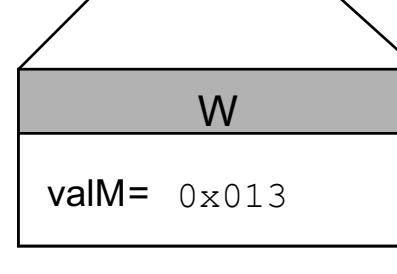
*bubble*

*bubble*

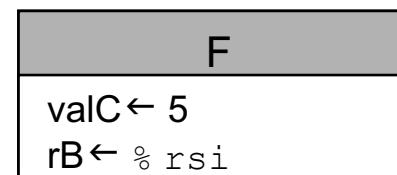
0x013: irmovq \$5,%rsi # Return



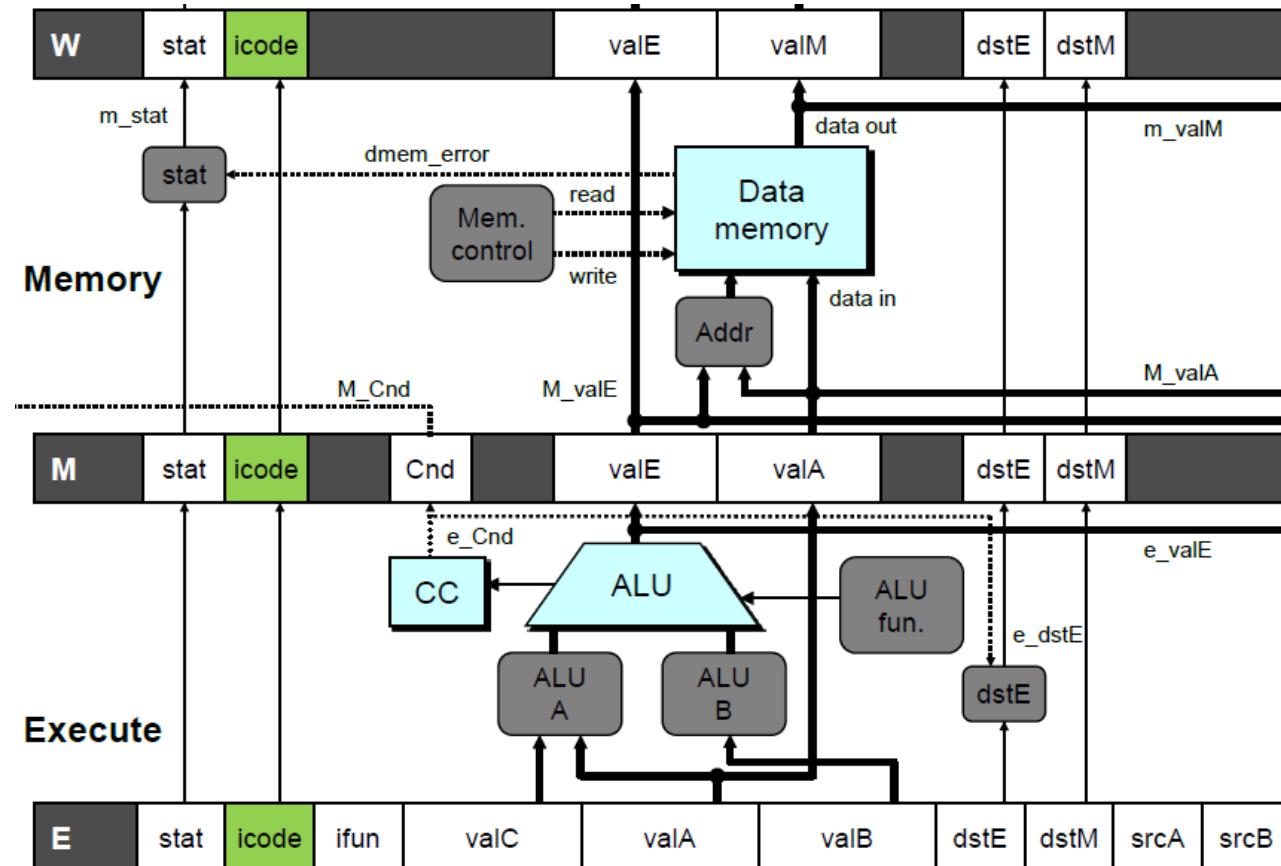
- As `ret` passes through pipeline, stall at fetch stage
  - While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage



⋮



# Detecting Return



Condition	Trigger
Processing <code>ret</code>	<code>IRET in { D_icode, E_icode, M_icode }</code>

# Control for Return

```
# demo-retb
```

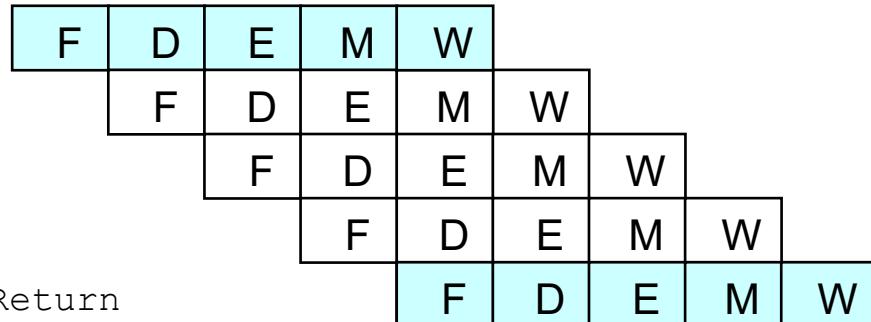
0x026: ret

*bubble*

*bubble*

*bubble*

0x014: irmovq \$5,%rsi # Return



Condition	F	D	E	M	W
Processing ret	stall	<b>bubble</b>	normal	normal	normal

# Special Control Cases

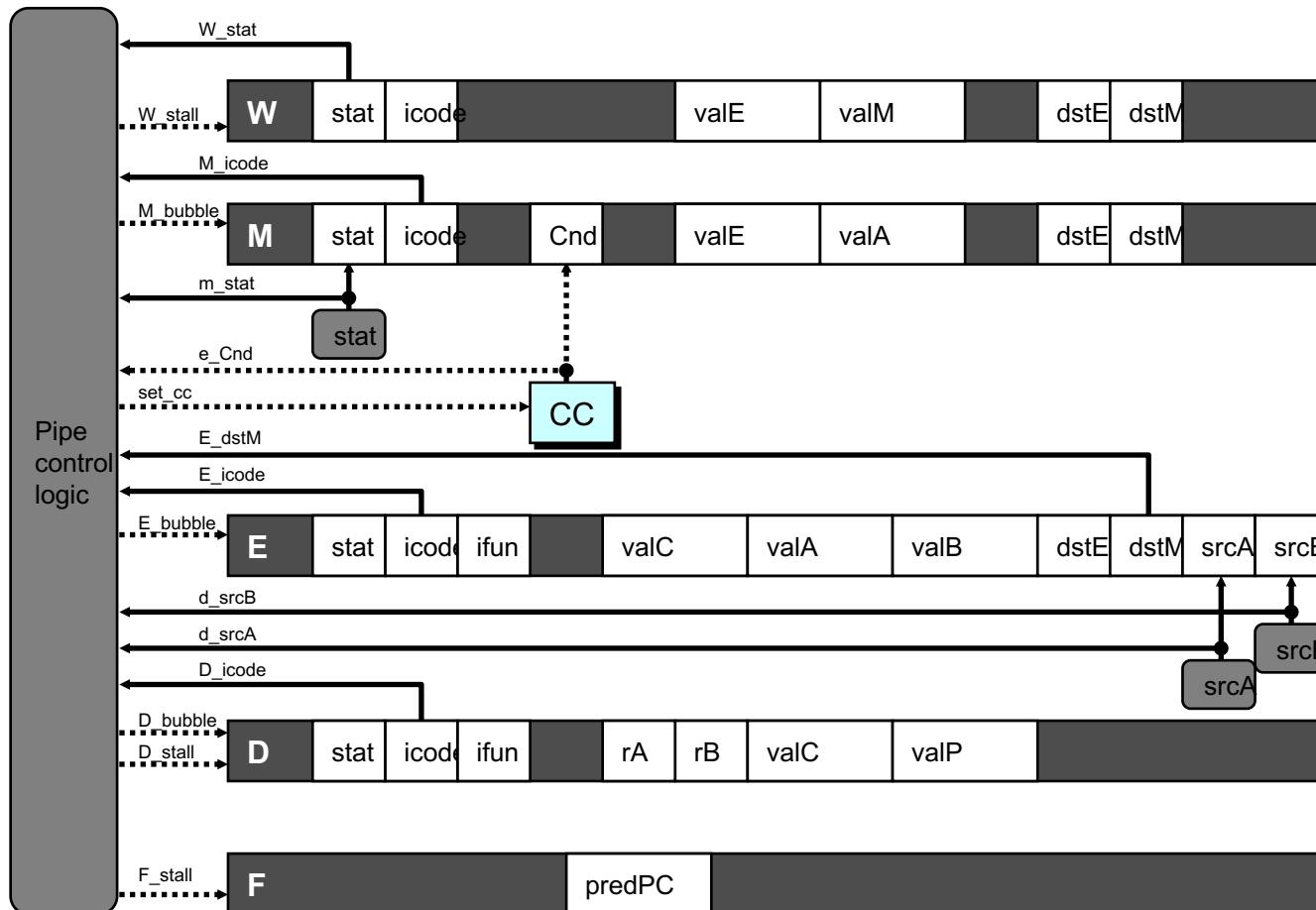
## ■ Detection

Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode == IJXX & !e_Cnd

## ■ Action (on next cycle)

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

# Implementing Pipeline Control



- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle

# Initial Version of Pipeline Control

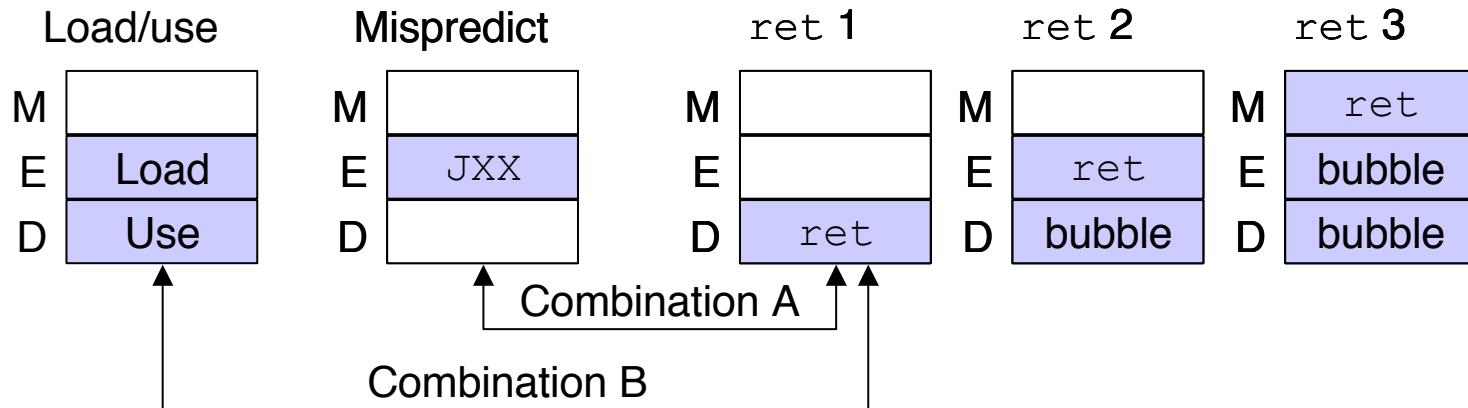
```
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode } ;

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 } ;
```

# Control Combinations



- Special cases that can arise on same clock cycle

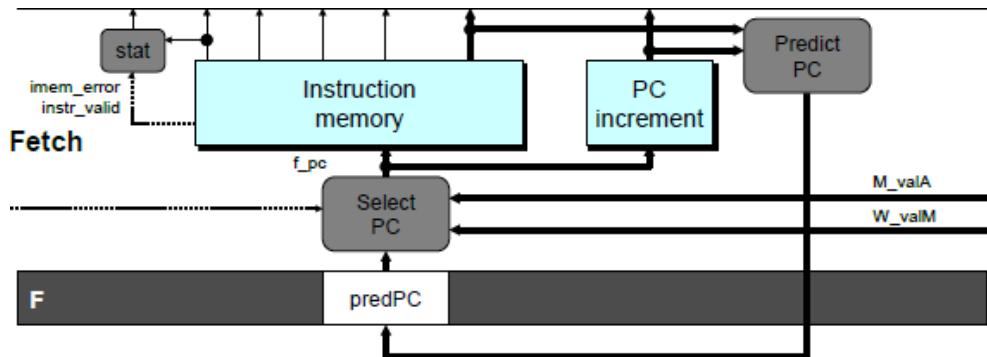
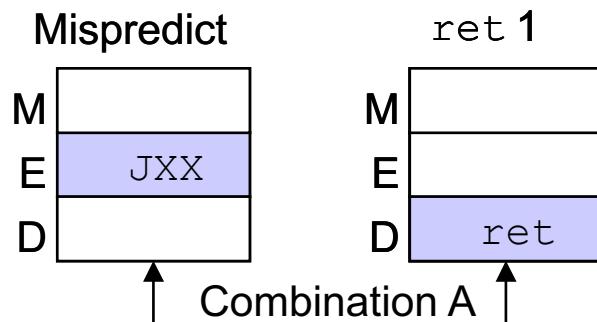
## ■ Combination A

- Not-taken branch
- `ret` instruction at branch target

## ■ Combination B

- Instruction that reads from memory to `%rsp`
- Followed by `ret` instruction

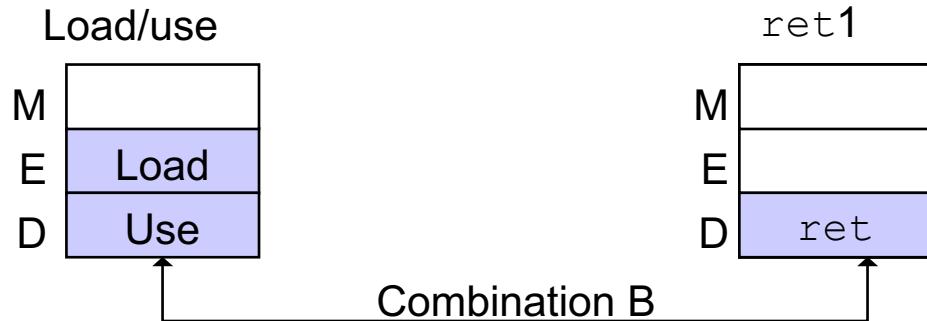
# Control Combination A



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal
<i>Combination</i>	<i>stall</i>	<i>bubble</i>	<i>bubble</i>	<i>normal</i>	<i>normal</i>

- Should handle as mispredicted branch
- Stalls F pipeline register
- But PC selection logic will be using M\_valM anyhow

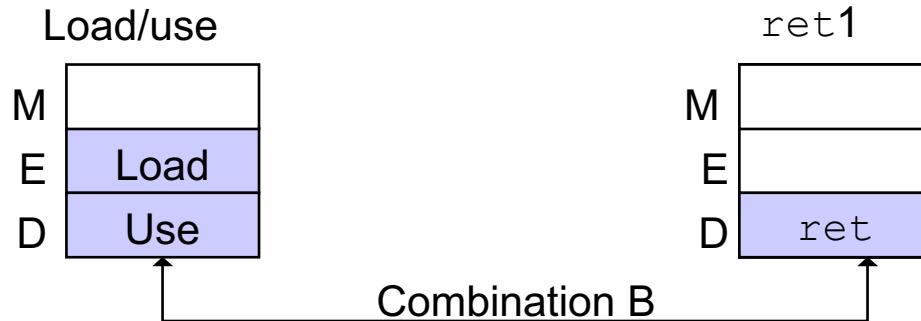
# Control Combination B



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	<i>stall</i>	<i>bubble + stall</i>	<i>bubble</i>	<i>normal</i>	<i>normal</i>

- Would attempt to bubble *and* stall pipeline register D
- Signaled by processor as pipeline error

# Handling Control Combination B



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
<b>Combination</b>	<b>stall</b>	<b>stall</b>	<b>bubble</b>	<b>normal</b>	<b>normal</b>

- Load/use hazard should get priority
- ret instruction should be held in decode stage for additional cycle

# 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 });

```

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
<i>Combination</i>	<i>stall</i>	<i>stall</i>	<i>bubble</i>	<i>normal</i>	<i>normal</i>

- Load/use hazard should get priority
- ret instruction should be held in decode stage for additional cycle

# Pipeline Summary

## ■ Data Hazards

- Most handled by forwarding
  - No performance penalty
- Load/use hazard requires one cycle stall

## ■ Control Hazards

- Cancel instructions when detect mispredicted branch
  - Two clock cycles wasted
- Stall fetch stage while `ret` passes through pipeline
  - Three clock cycles wasted

## ■ Control Combinations

- Must analyze carefully
- First version had subtle bug
  - Only arises with unusual instruction combination