Pipelined Y86-64 Wrapup

CSCI 237: Computer Organization
20th Lecture, Apr 9, 2025

Jeannie Albrecht

Administrative Details

- Lab 4 today/tomorrow
  - C programming!
  - Due next Tue/Wed
- No Glow HW this week
  - Finish Lab 4 instead

Midterm

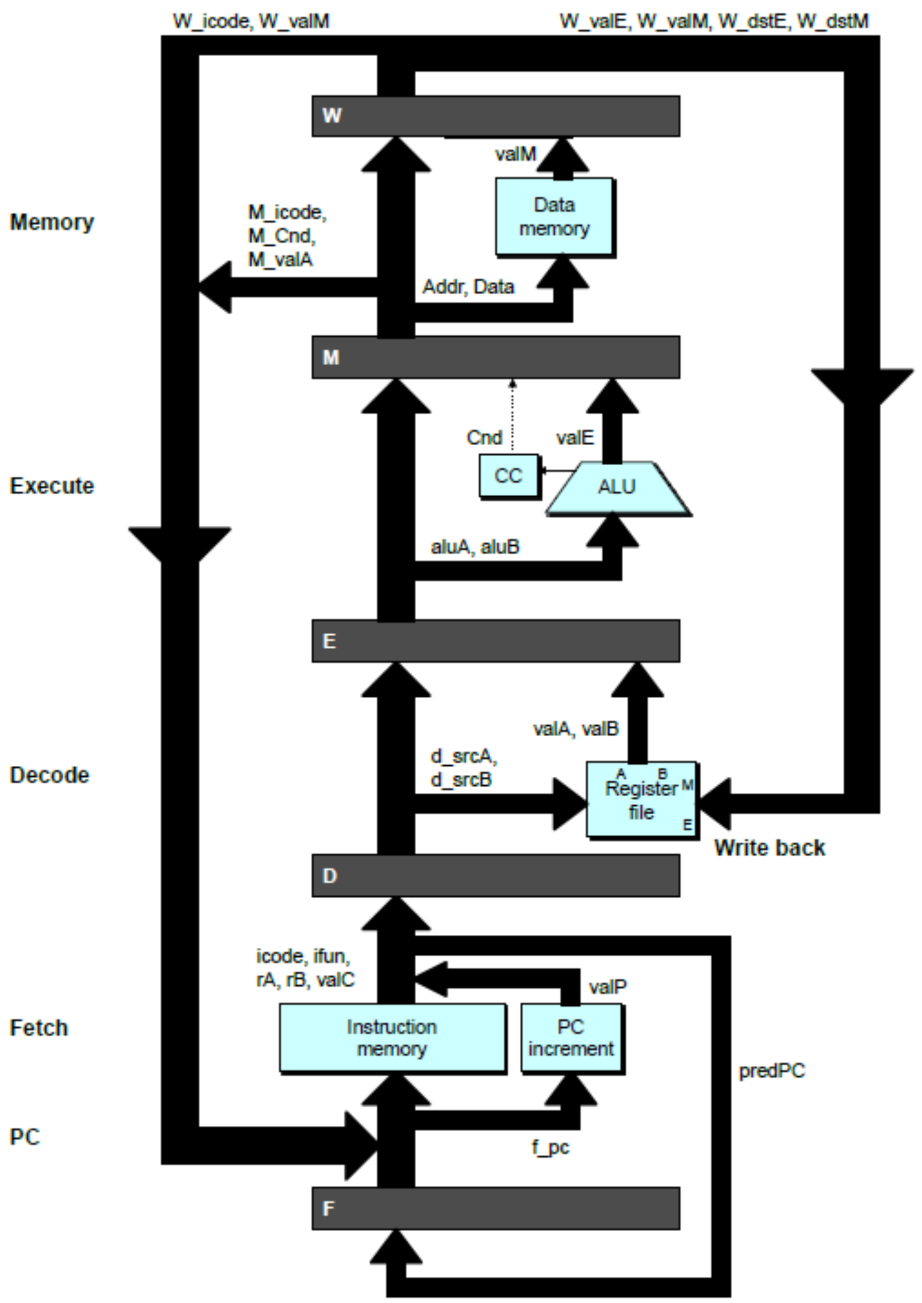
- Avg and median grade: 86%
  - Great job!
  - This is a little higher than usual
- Look it over (it will be waiting for you outside) and come see me if you’d like to discuss anything
- General observations:
  - Conditional moves are ”better” when extra computations are fast, easy, and safe
  - Using ( ) in x86 instructions:
    - Like a pointer in C
    - But not all instructions support using ( )
    - Often have to use move to put value in register first
  - Arrays are allocated contiguously

Last time

- General principles of pipelining (Ch 4.4)
  - Goals
  - Difficulties
- Creating a pipelined Y86-64 processor (Ch 4.5)
  - Rearranging SEQ
  - Inserting pipeline registers
  - Problems with data and control hazards

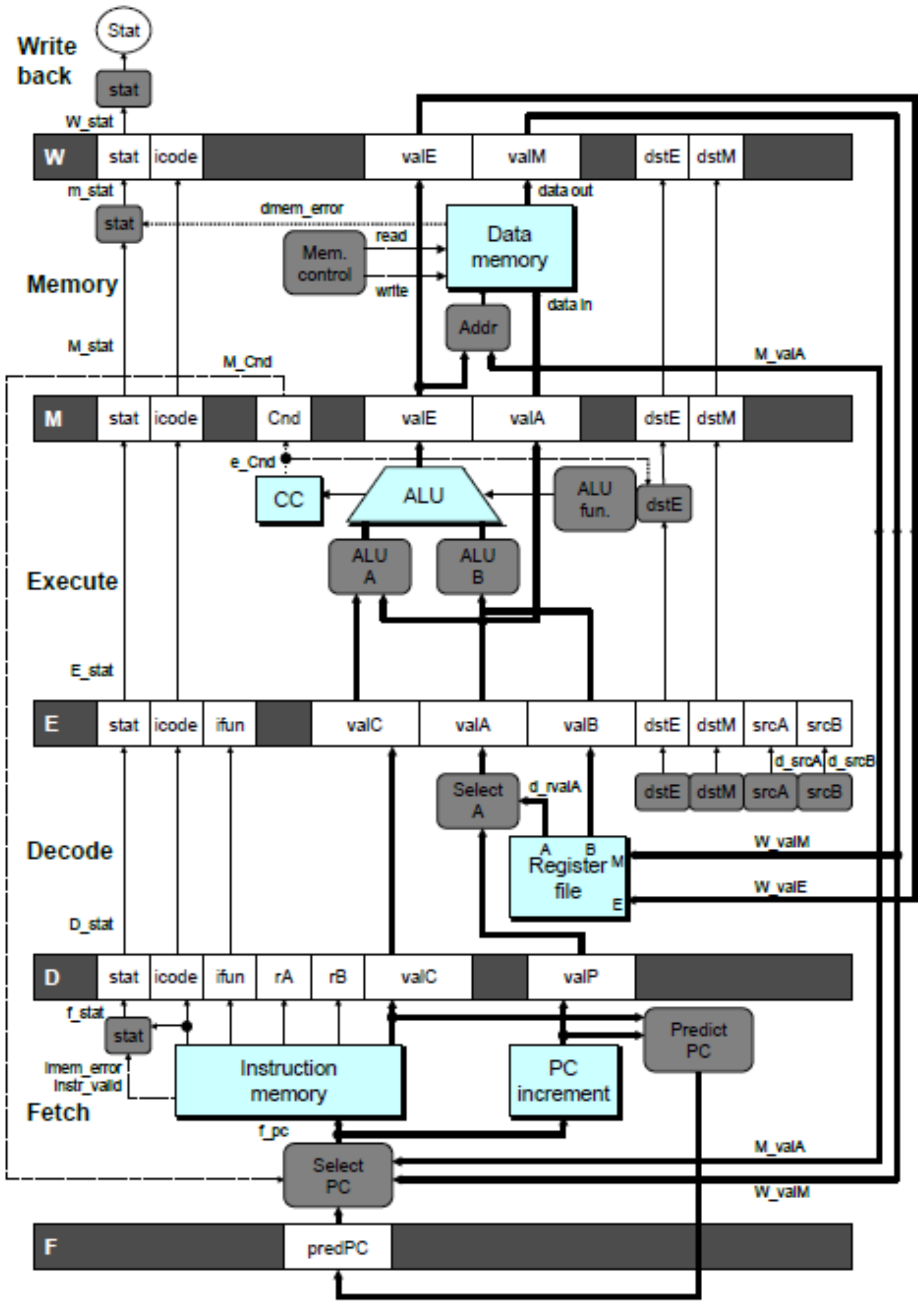
Recap: Pipeline Stages

- 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



Recap:
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



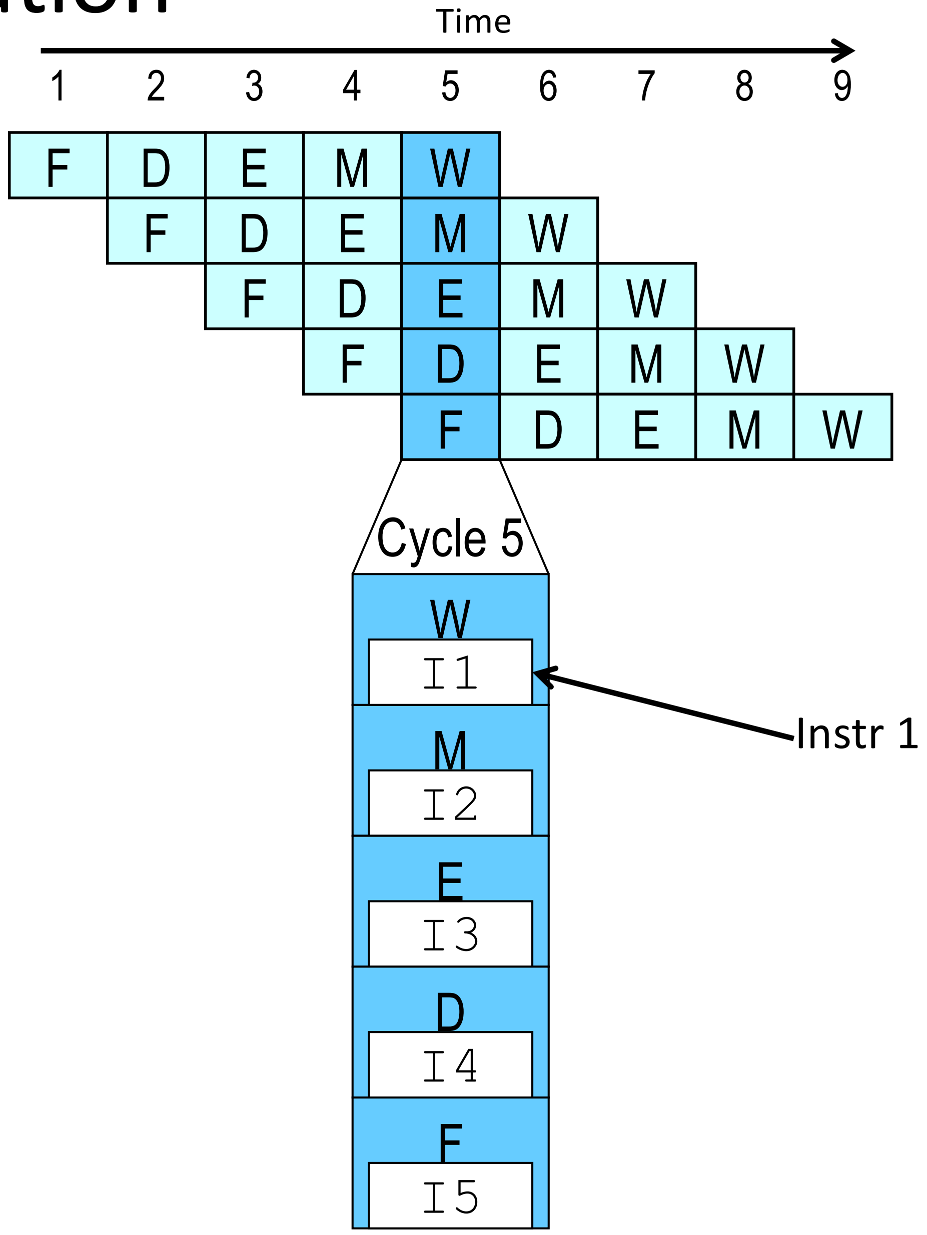
Today

Make the pipelined processor really work (mostly)!

- 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
  - Stalling, bubbling, data forwarding
- Advanced pipelining concepts: NOT COVERED IN CLASS!
- (slides are included at the end of this lecture for reference)
  - Load/Use Data Hazard
  - Control Hazards
    - Mispredict conditional branch
    - Getting return address for `ret` instruction
  - Special Control Combinations

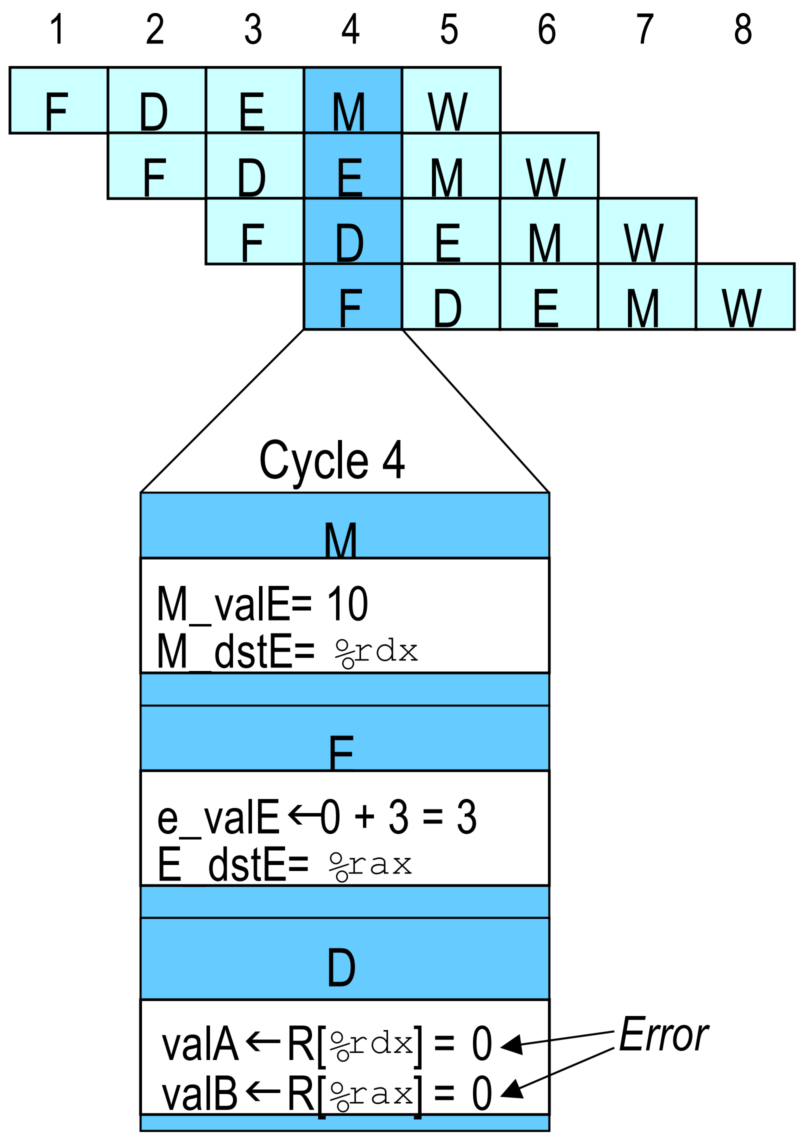
Pipeline Demonstration

```
irmovq   $1,%rax   #I1
irmovq   $2,%rcx   #I2
irmovq   $3,%rdx   #I3
irmovq   $4,%rbx   #I4
halt               #I5
```



Data Dependencies: No Nop

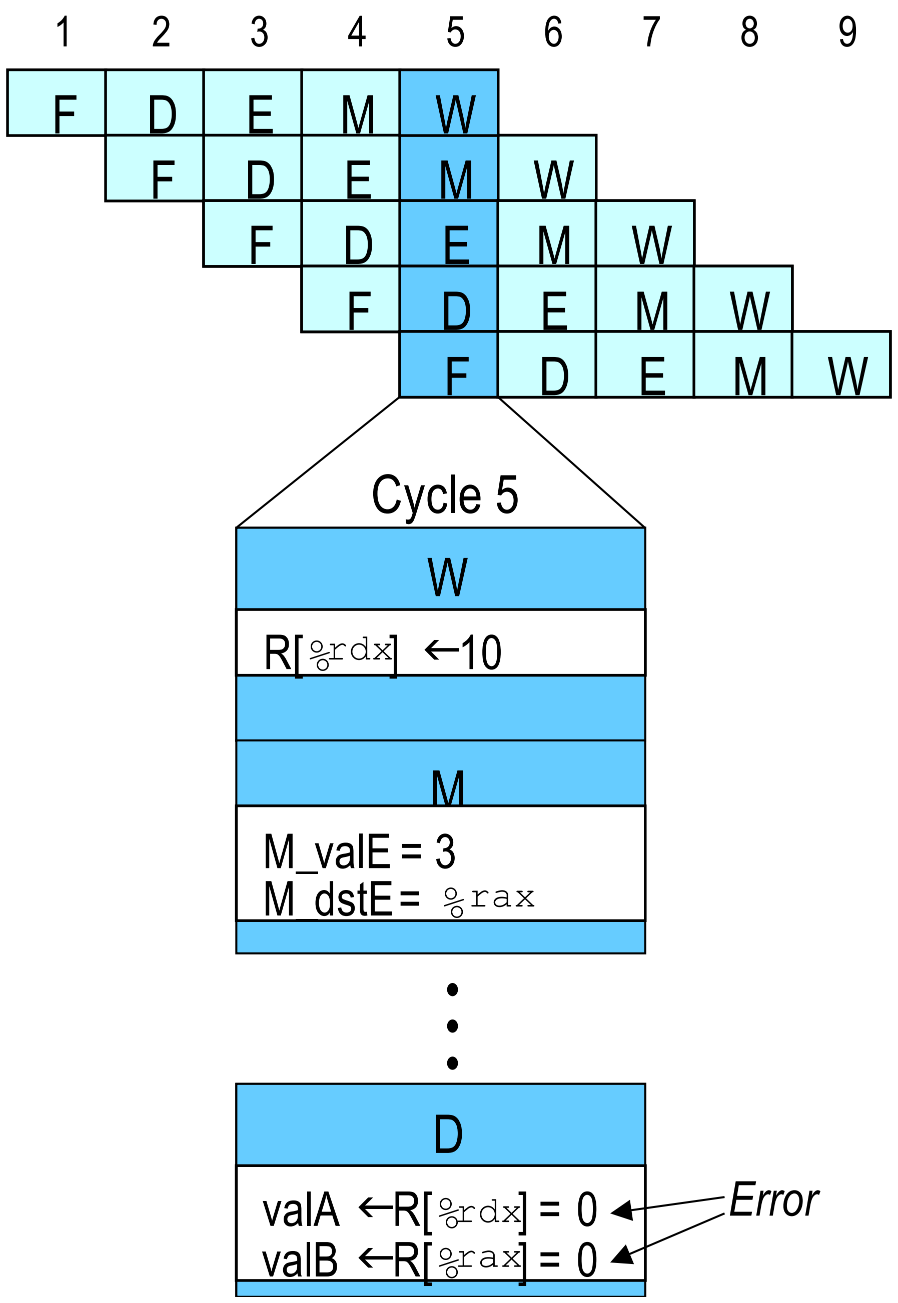
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



How can we avoid these errors?

Data Dependencies: 1 Nop

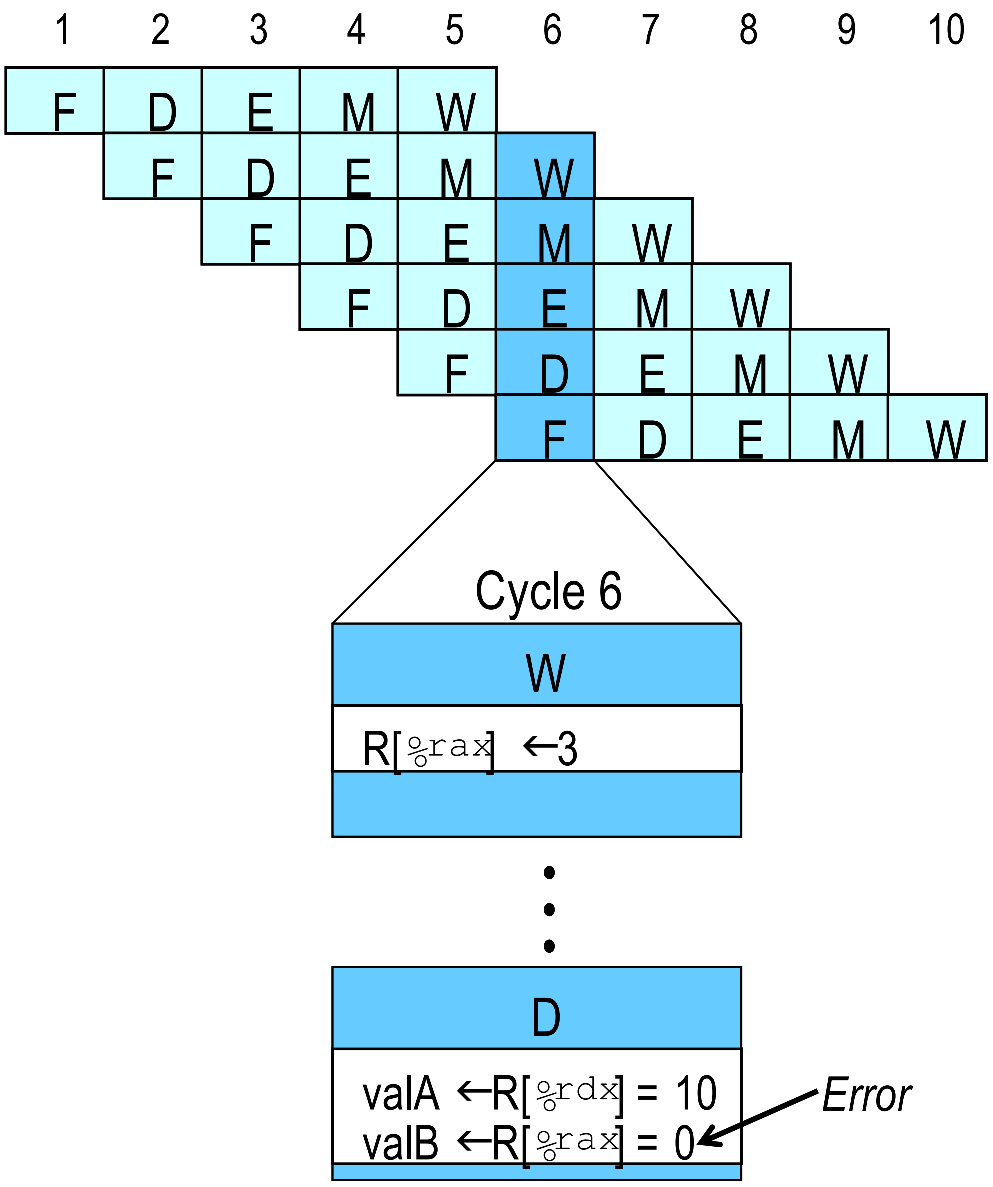
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```



How can we avoid these errors?

Data Dependencies: 2 Nop's

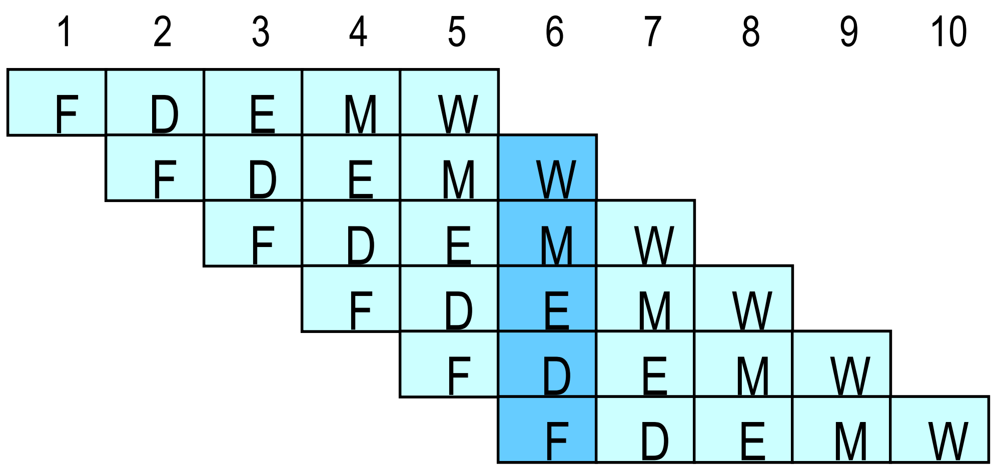
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



How can we avoid these errors?

Dealing with Data Dependencies

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

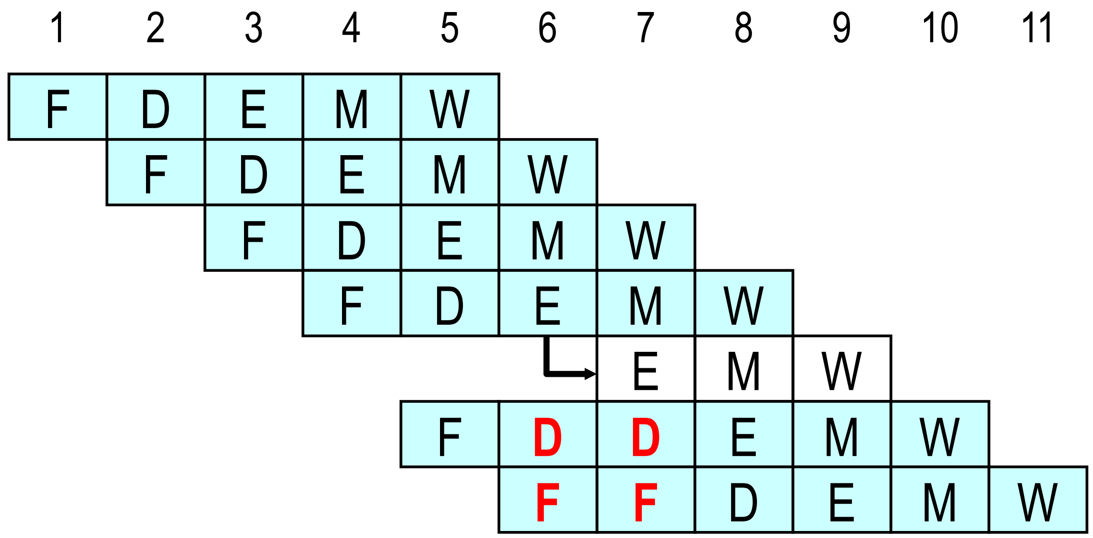


1	2	3	4	5	6	7	8	9	10
F	D	E	M	W					
	F	D	E	M	W				
		F	D	E	M	W			
			F	D	E	M	W		
				F	D	E	M	W	
					F	D	E	M	W

- If instruction follows too closely after one that writes register, we need to slow it down
- How?

Stalling for Data Dependencies

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

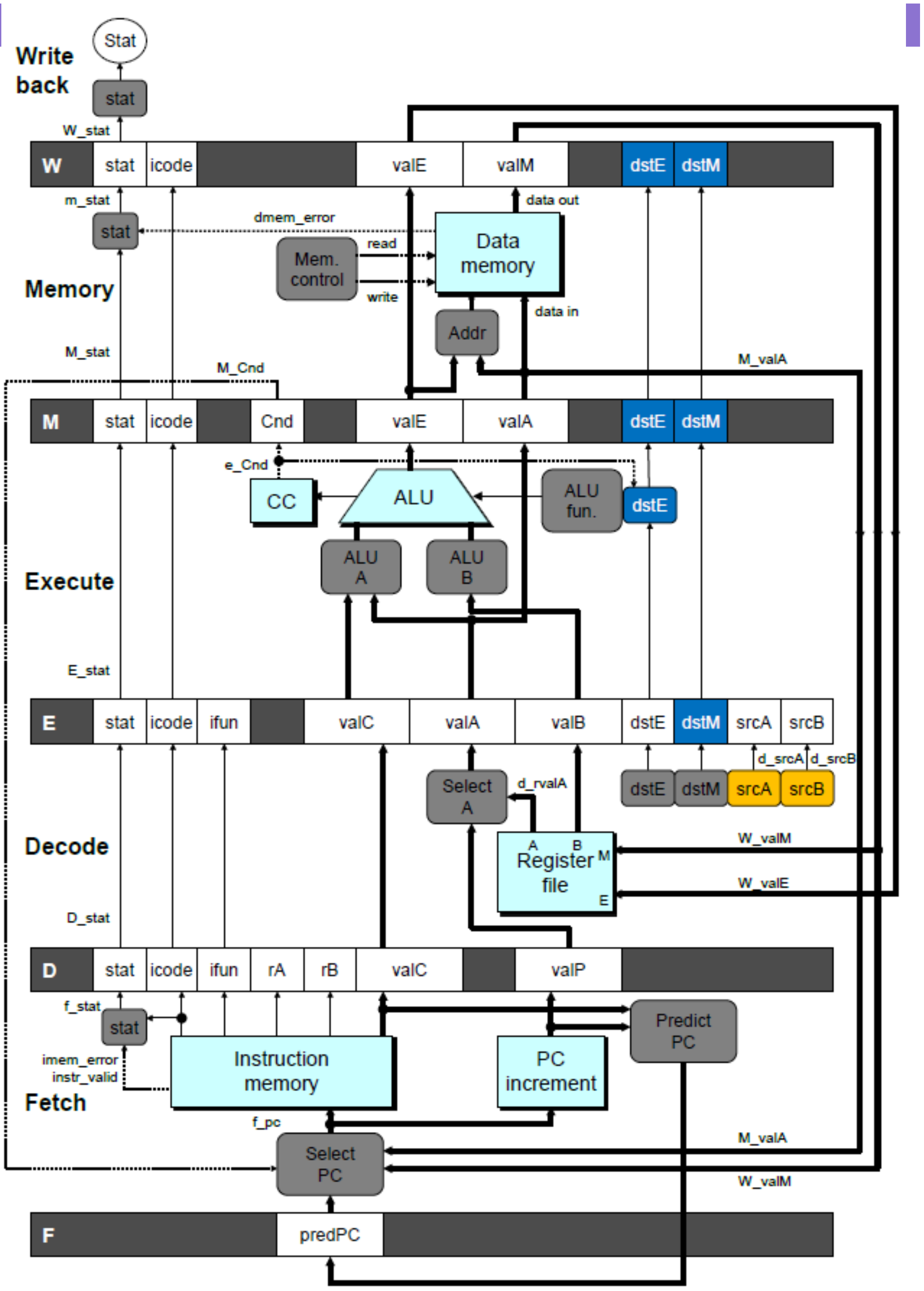


	1	2	3	4	5	6	7	8	9	10	11
0x000: irmovq $10,%rdx	F	D	E	M	W						
0x00a: irmovq $3,%rax		F	D	E	M	W					
0x014: nop			F	D	E	M	W				
0x015: nop				F	D	E	M	W			
bubble							E	M	W		
0x016: addq %rdx,%rax					F	**D**	**D**	E	M	W	
0x018: halt						**F**	**F**	D	E	M	W

- If instruction follows too closely after one that writes register, we need to slow it down
- Solution: Hold instruction in decode (stall the pipeline)
- Dynamically inject nop into execute stage (bubble)

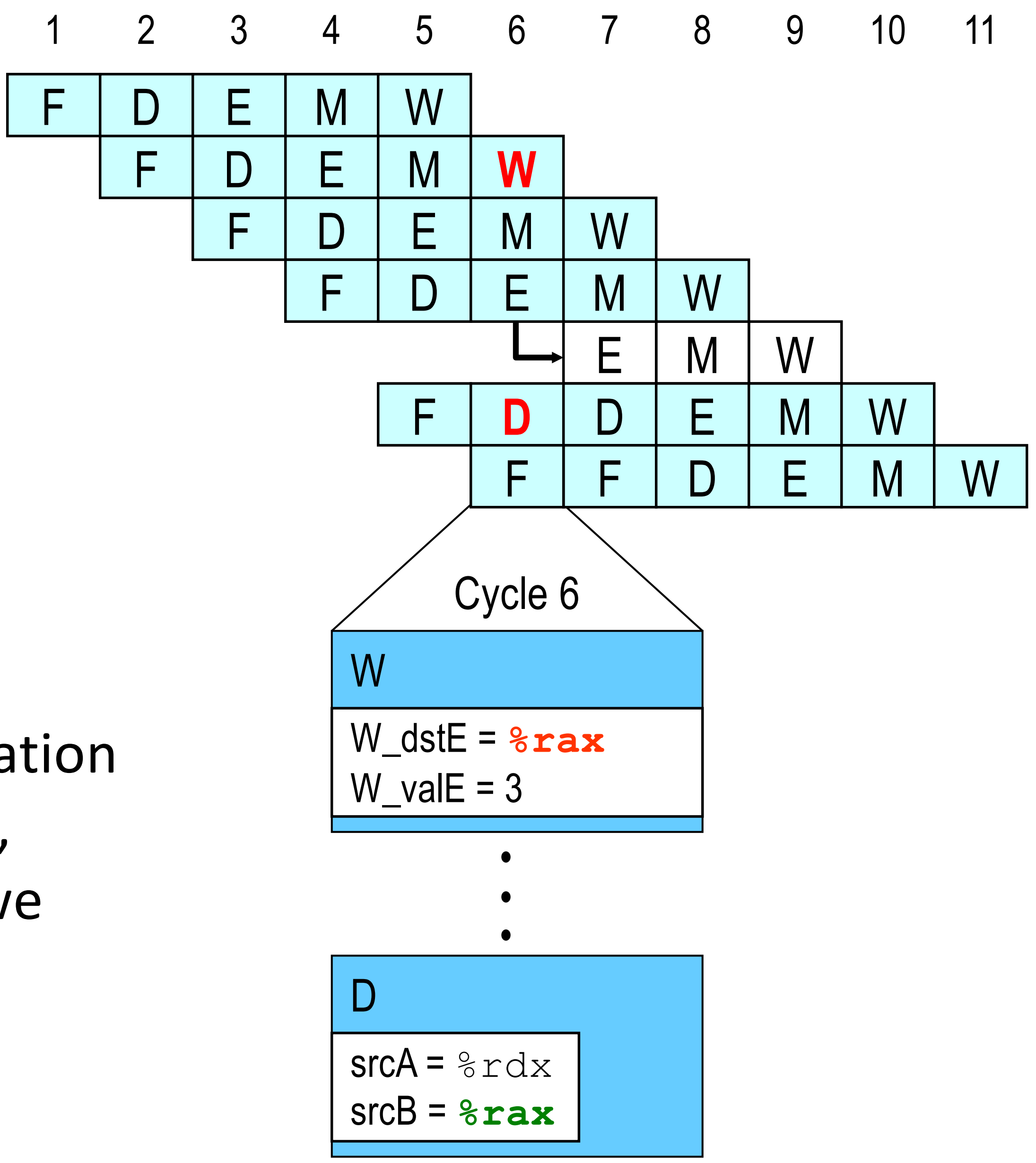
Stall Condition

- Source Registers
  - srcA and srcB of current instr 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

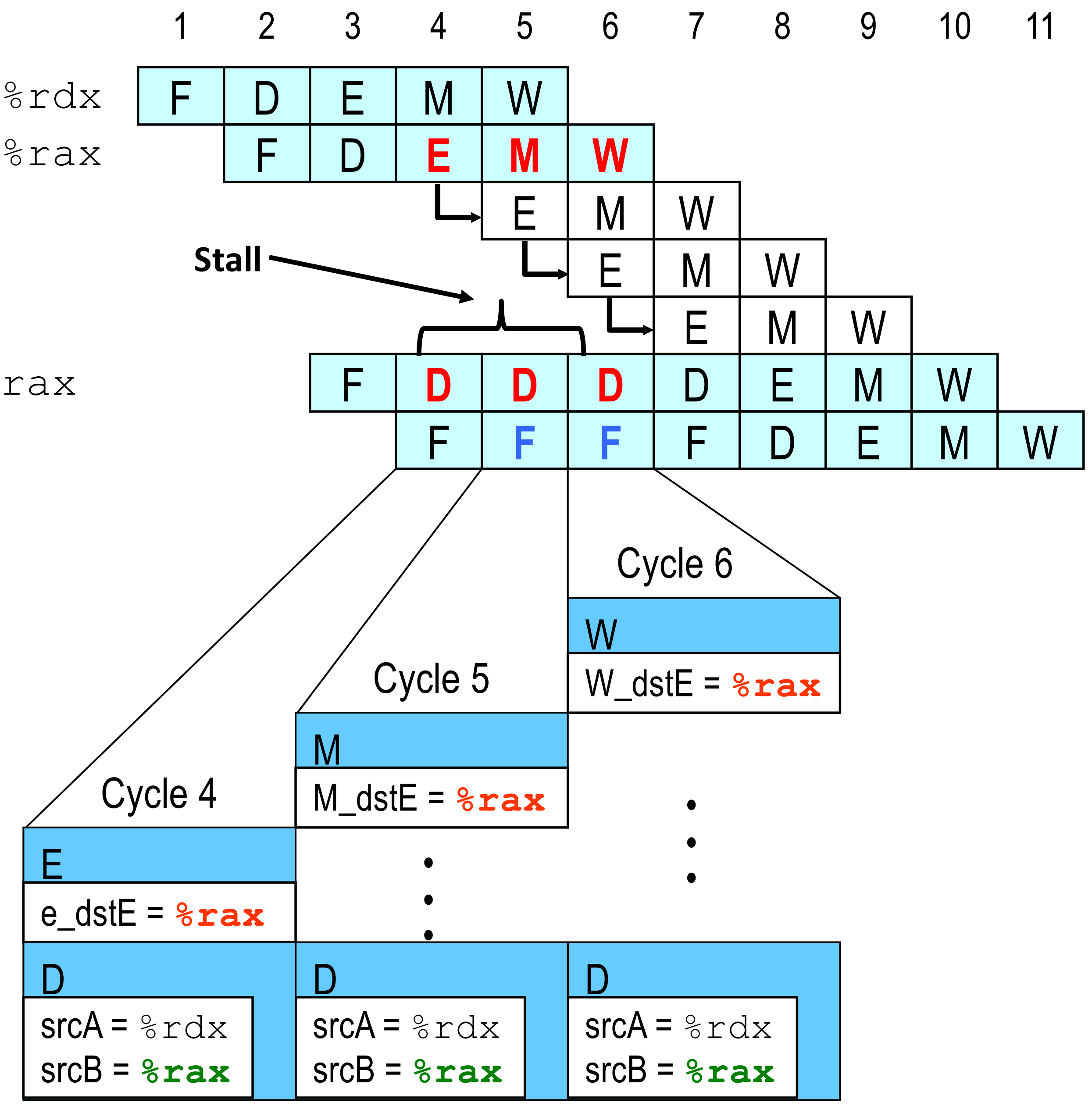
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```



If source of instruction in decode is same as destination for instruction in execute, memory, or write-back, we must stall and bubble.

Stalling x3

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



What Happens When Stalling?

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

Cycle 4

Write Back	
Memory	`0x000: irmovq $10,%rdx`
Execute	`0x00a: irmovq  $3,%rax`
Decode	`0x014: addq %rdx,%rax`
Fetch	`0x016: halt`

- 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

What Happens When Stalling?

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

	Cycle 5
Write Back	`0x000: irmovq $10,%rdx`
Memory	`0x00a: irmovq  $3,%rax`
Execute	***bubble***
Decode	`0x014: addq %rdx,%rax`
Fetch	`0x016: halt`

- 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

What Happens When Stalling?

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

Cycle 6	
Write Back	`0x00a: irmovq  $3,%rax`
Memory	***bubble***
Execute	***bubble***
Decode	`0x014: addq %rdx,%rax`
Fetch	`0x016: halt`

- 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

What Happens When Stalling?

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

	Cycle 7
Write Back	***bubble***
Memory	***bubble***
Execute	***bubble***
Decode	`0x014: addq %rdx,%rax`
Fetch	`0x016: halt`

- 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

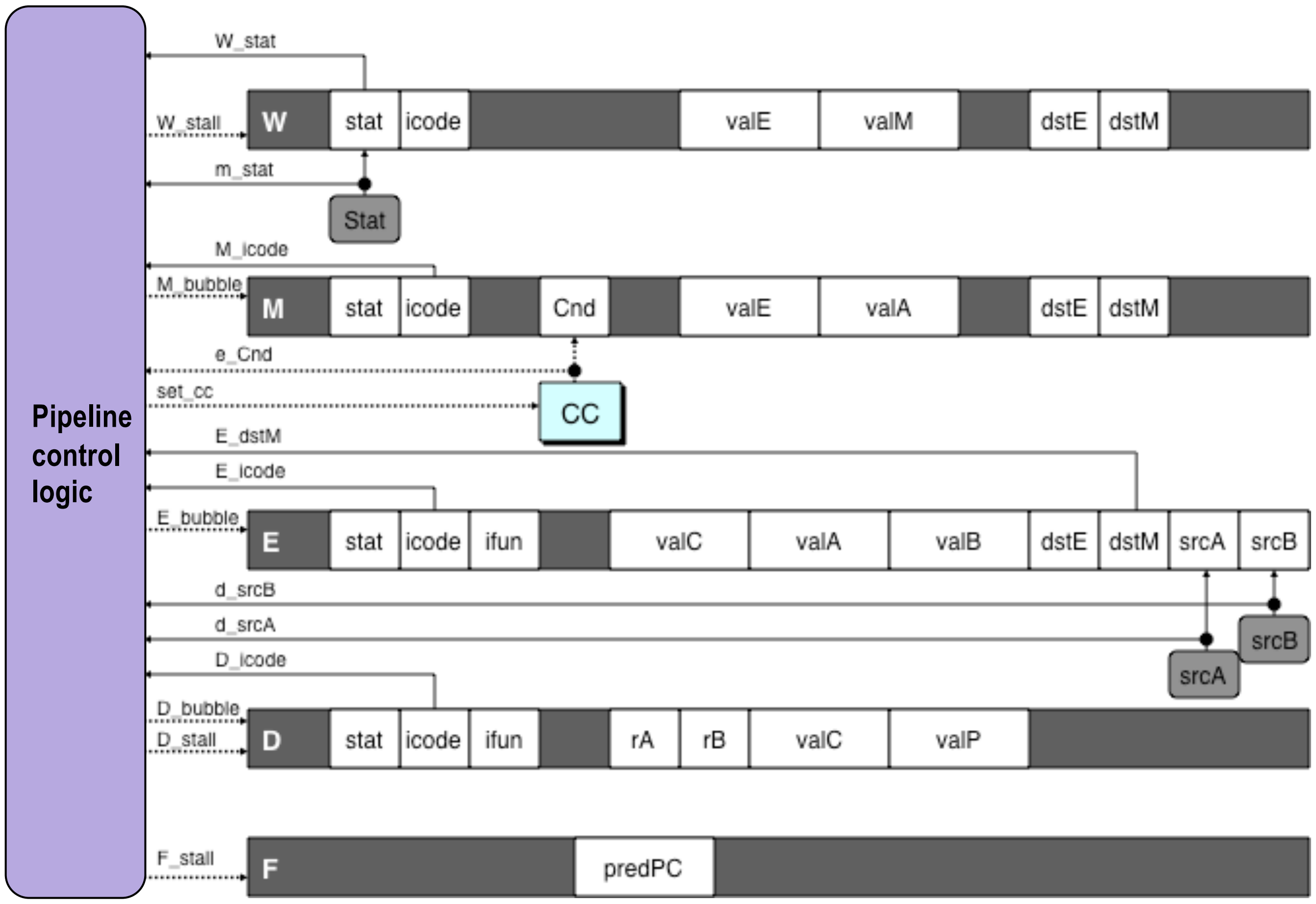
What Happens When Stalling?

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

	Cycle 8
Write Back	***bubble***
Memory	***bubble***
Execute	`0x014: addq %rdx,%rax`
Decode	`0x016: halt`
Fetch	

- 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

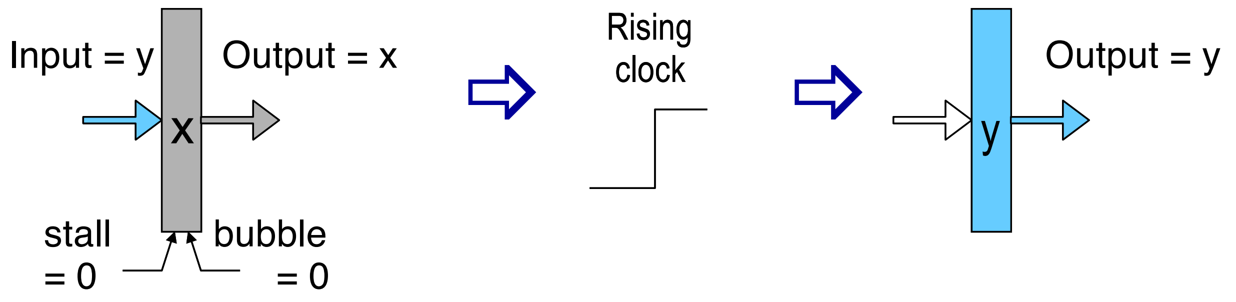
Implementing Stalling



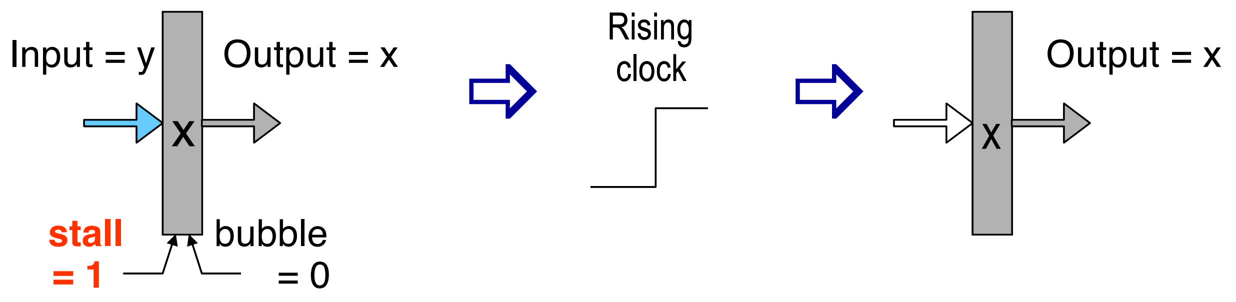
- Pipeline Control
  - Combinational logic detects stall condition
  - Sets mode signals for how pipeline registers should update

Pipeline Register Modes

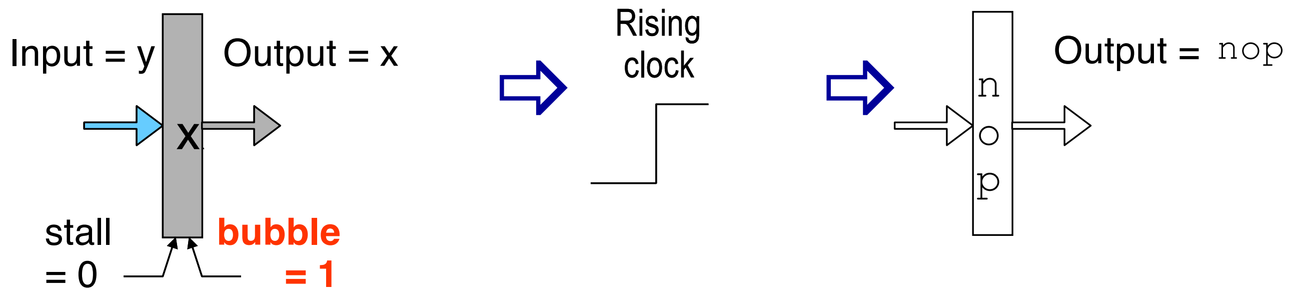
Normal



Stall



Bubble

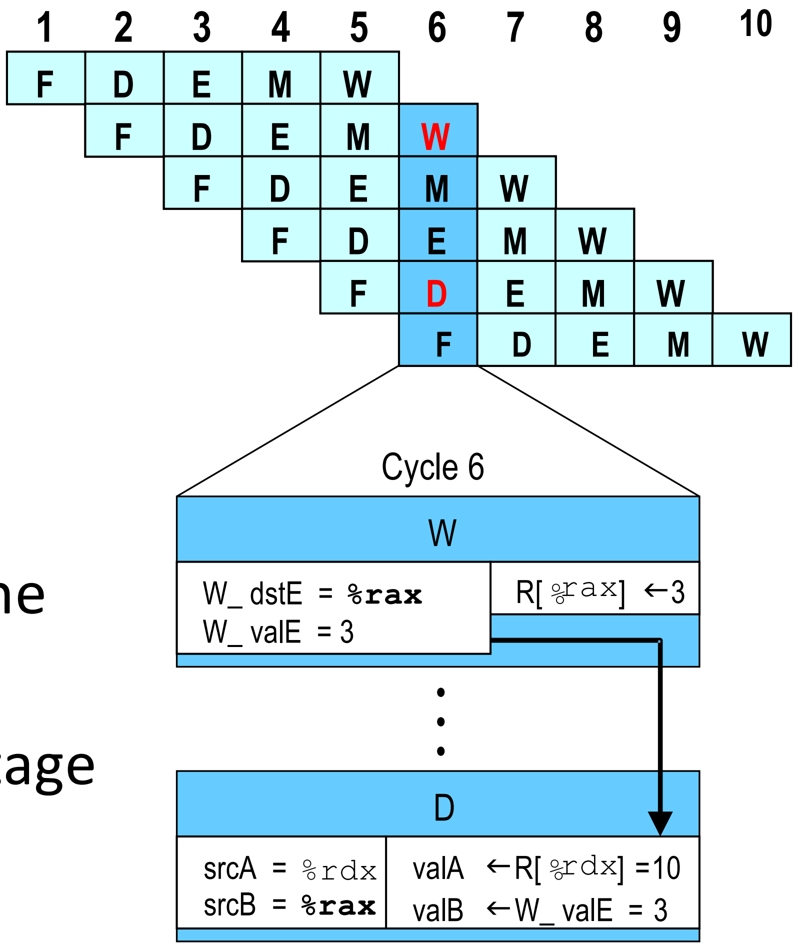


Data Forwarding

- 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
  - Desired value generated in execute or memory stage
  - Why wait for completion of write-back?
- Trick
  - Pass value directly from generating instruction to decode stage
  - Needs to be available at end of decode stage

Data Forwarding Example

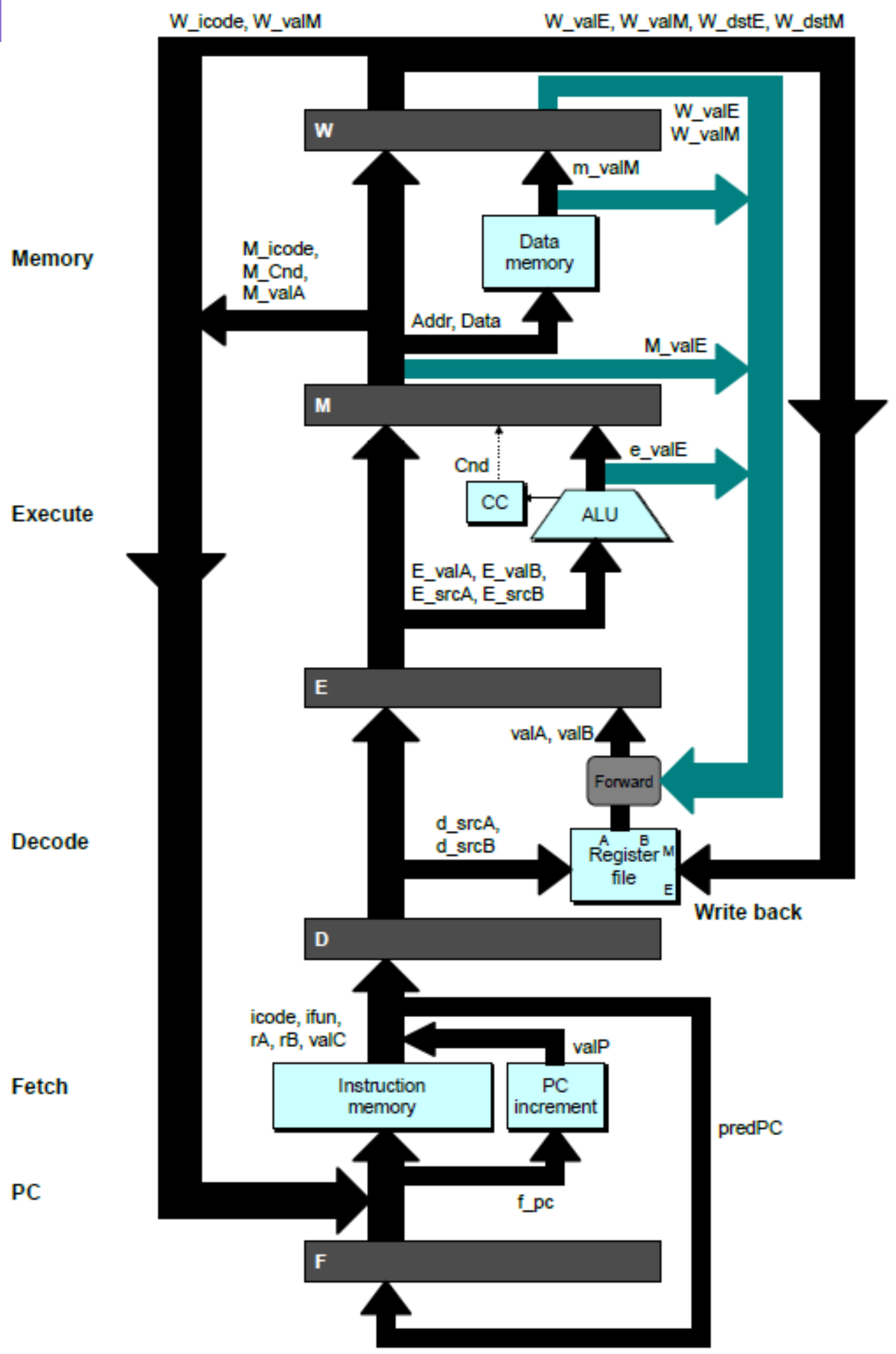
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



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

Bypass Paths

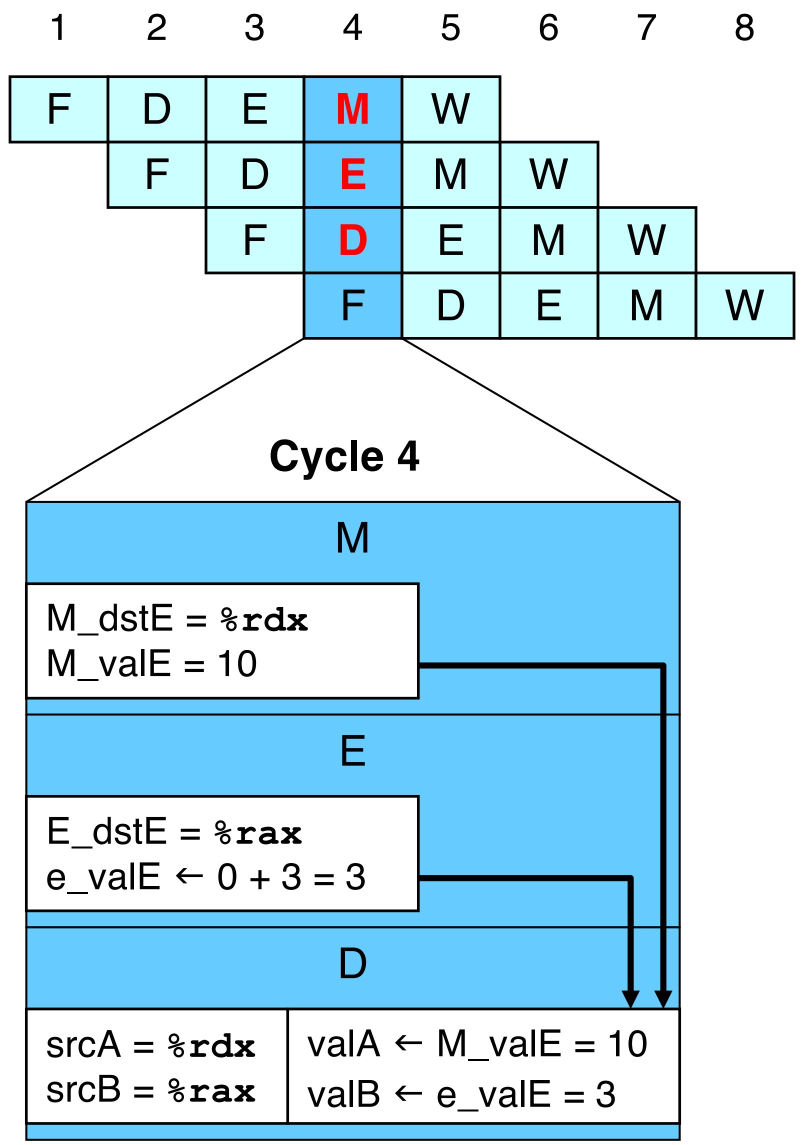
- Decode Stage
  - Forwarding logic selects valA and valB
  - Normally from register file
  - Forwarding: get valA or valB from later pipeline stage
- Forwarding Sources
  - Execute: valE
  - Memory: valE, valM
  - Write back: valE, valM



Data Forwarding Example #2

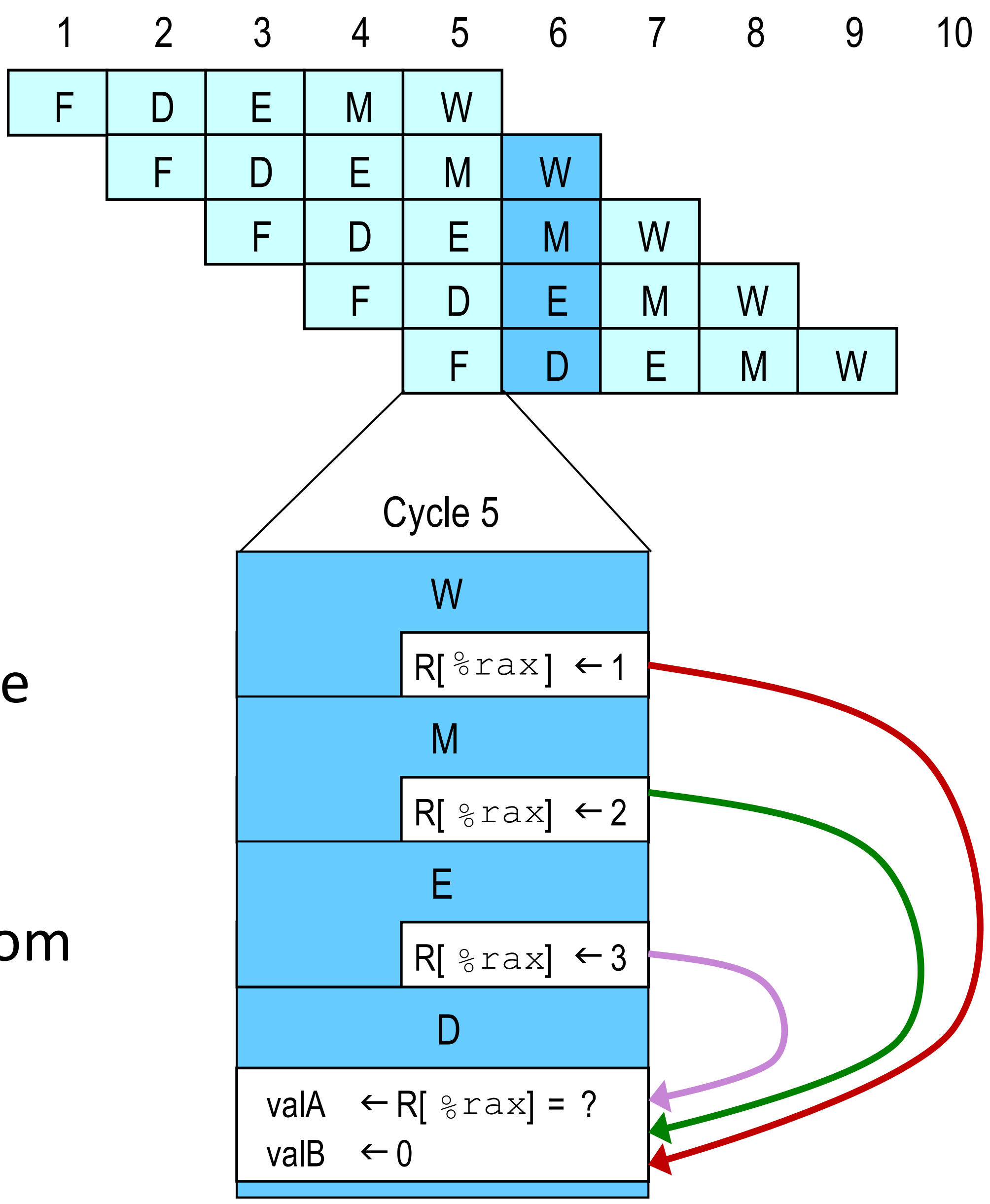
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

- 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



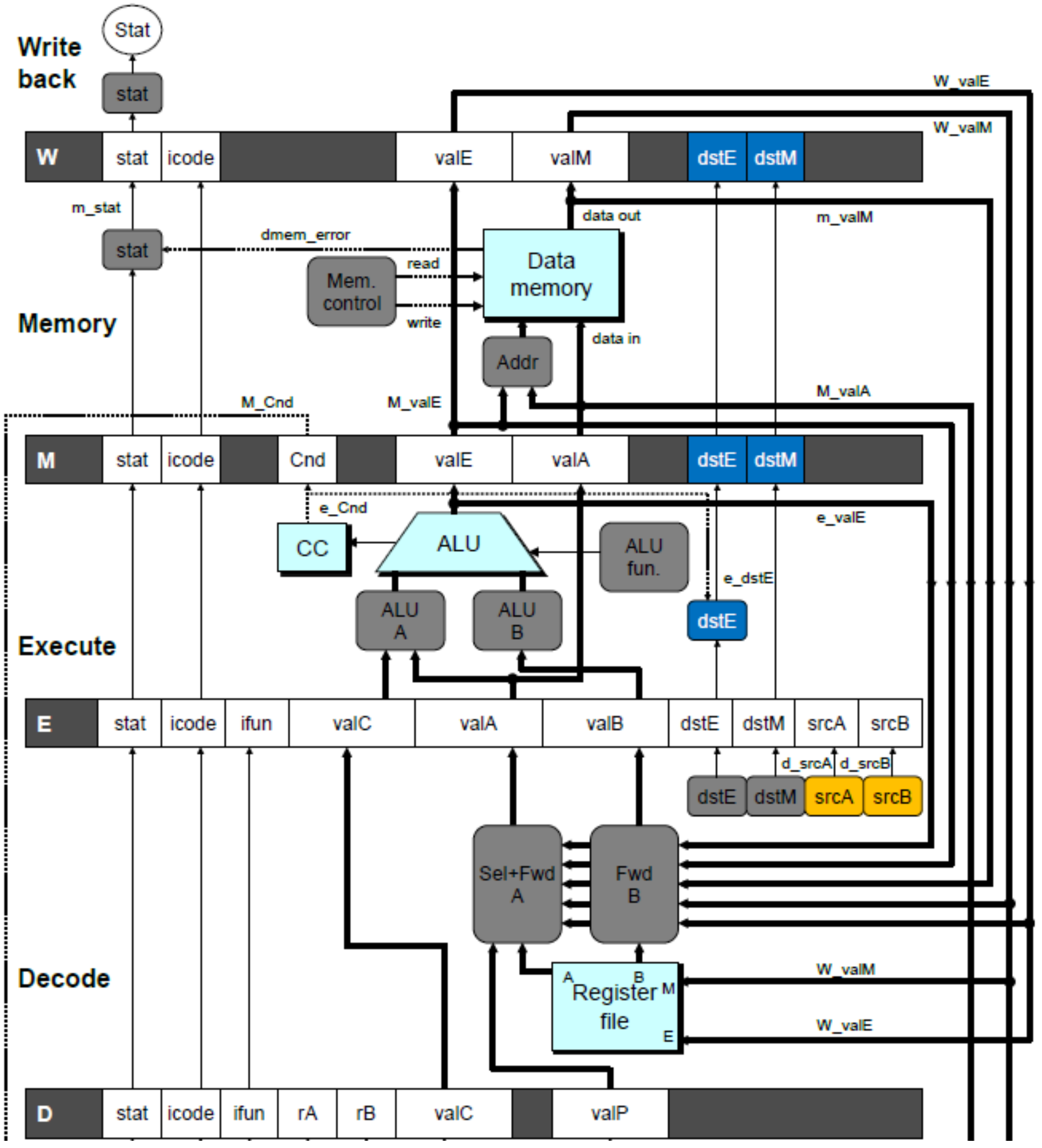
Forwarding Priority

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



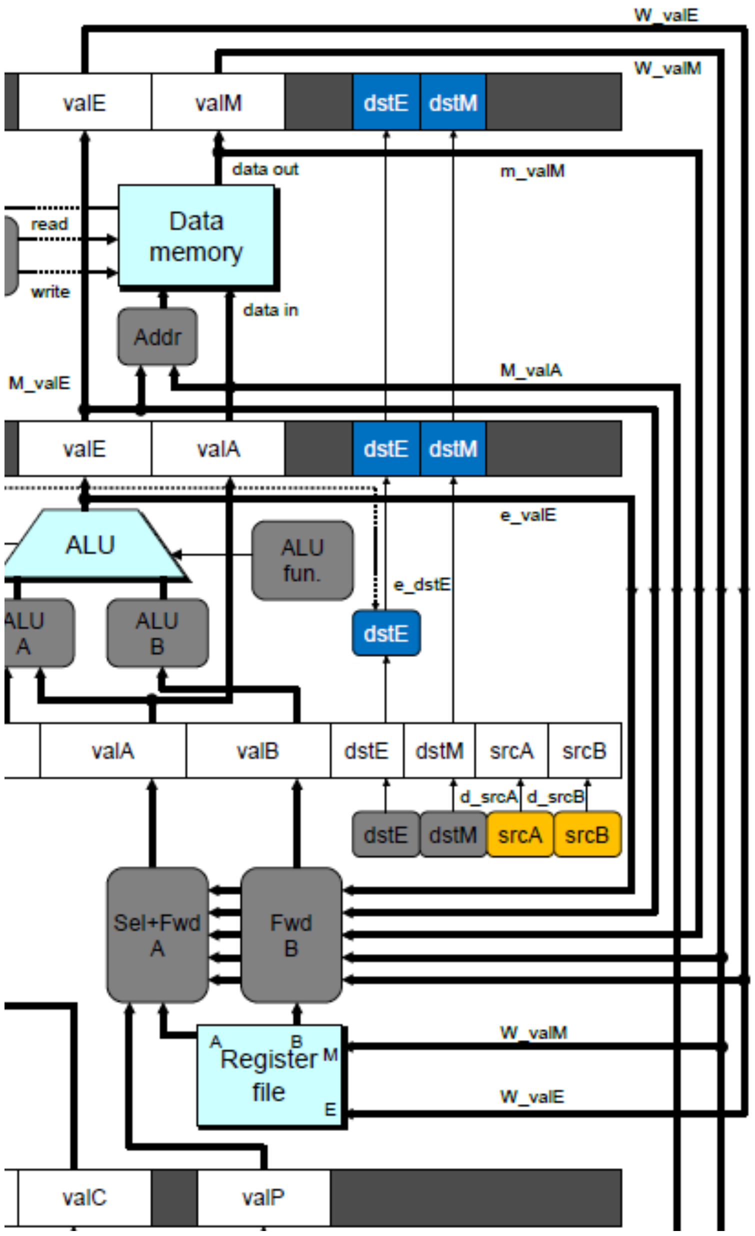
- Multiple Forwarding Choices
  - Which one should have priority?
  - Match SEQ semantics
  - Use matching value from *earliest* pipeline stage

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



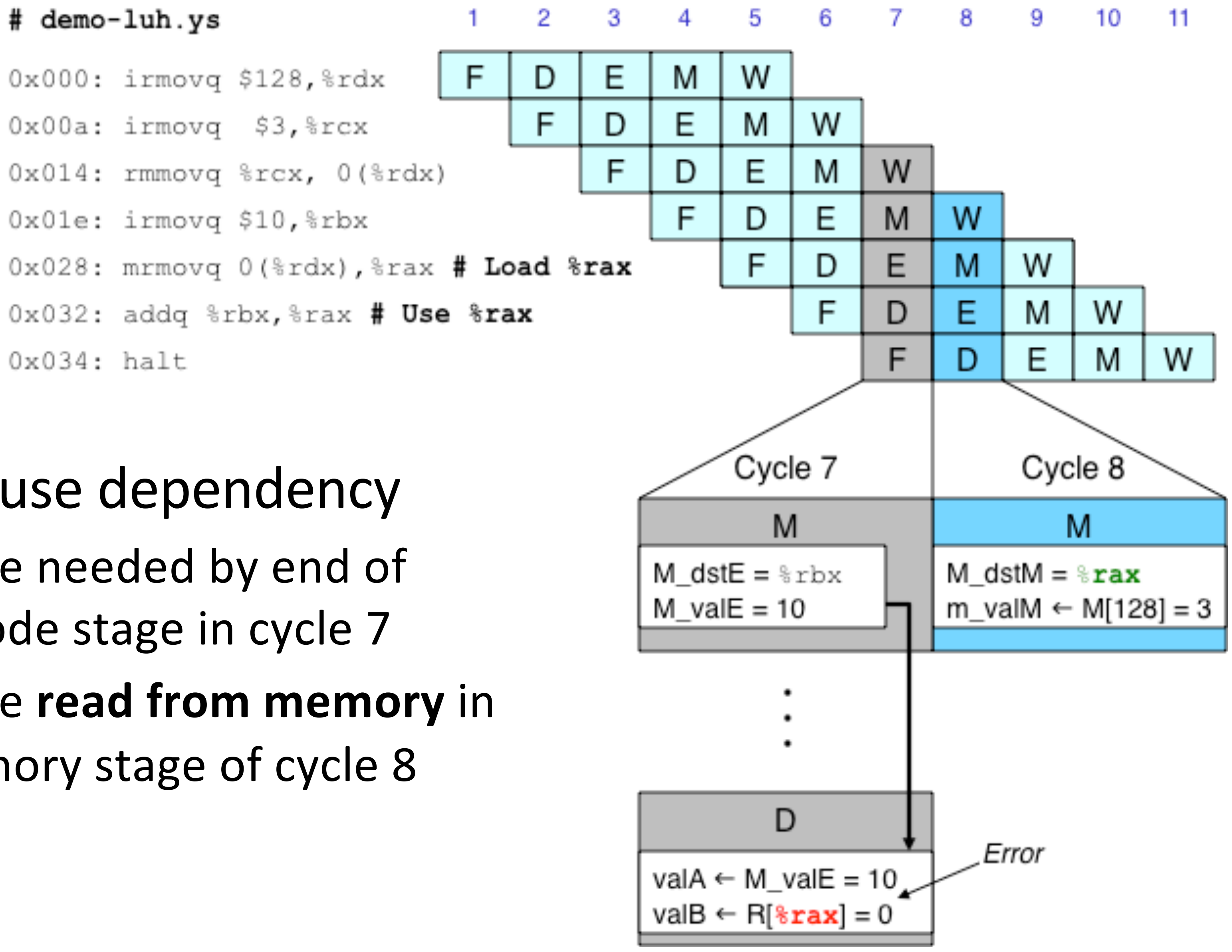
```
## What should be the A value?
int d_valA = [
  # Use incremented PC
    D_icode in { ICALL, IJXX } : D_valP;
  # Forward valE from execute
    d_srcA == e_dstE : e_valE;
  # Forward valM from memory
    d_srcA == M_dstM : m_valM;
  # Forward valE from memory
    d_srcA == M_dstE : M_valE;
  # Forward 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;
];
```

PIPELINE COVERAGE STOPS HERE!

- Ch 4.5 includes additional details about pipelined processors
  - Load/use data hazards, control hazards, control combinations
  - All very interesting topics! But we are moving on.
- The following slides were not covered in class, but are left here for your reference and to quench your curiosity ☺
- You will not be expected to know any of this material on any future exams!

- We are leaving the processor and moving on to memory (Ch 6)! 🎉

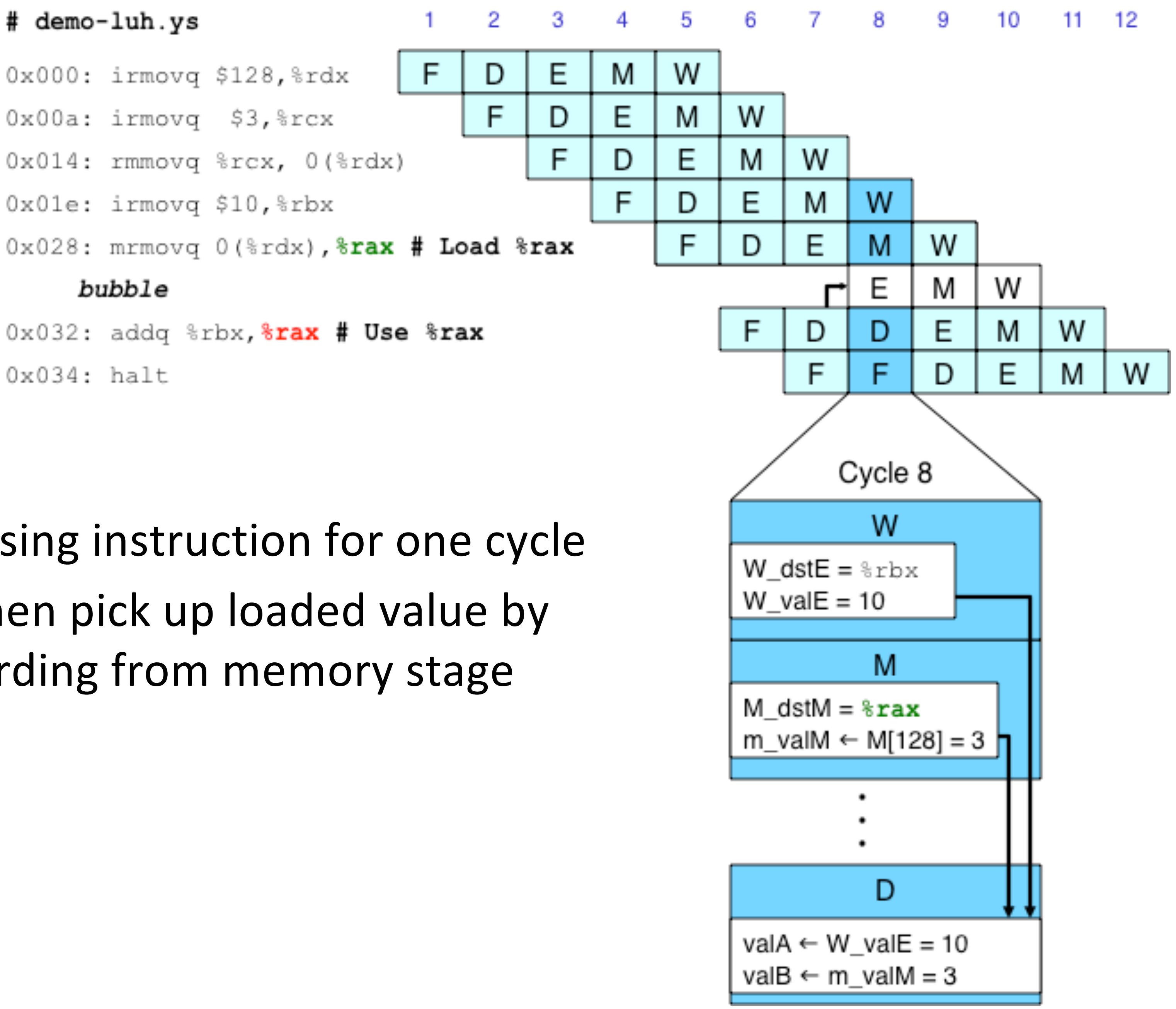
Limitation of Forwarding: Load/Use Data Hazard



```
# 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
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

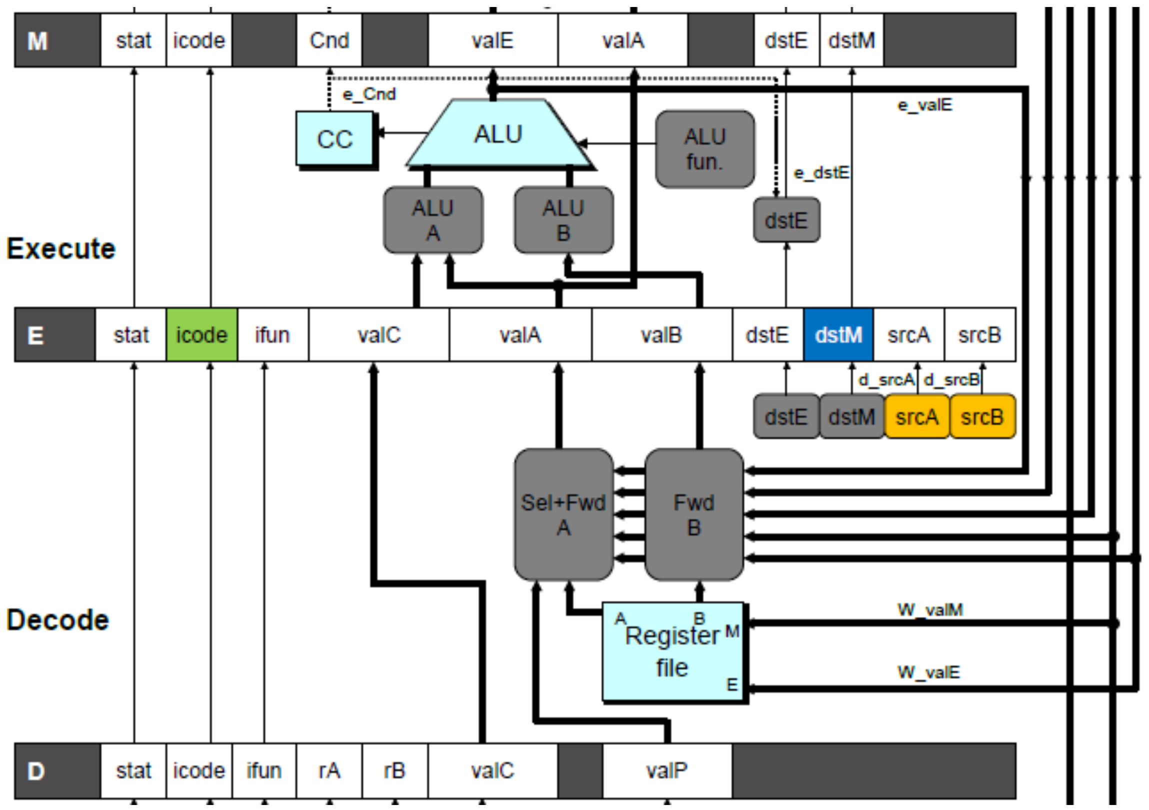
Avoiding Load/Use Hazard



```
# 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
```

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

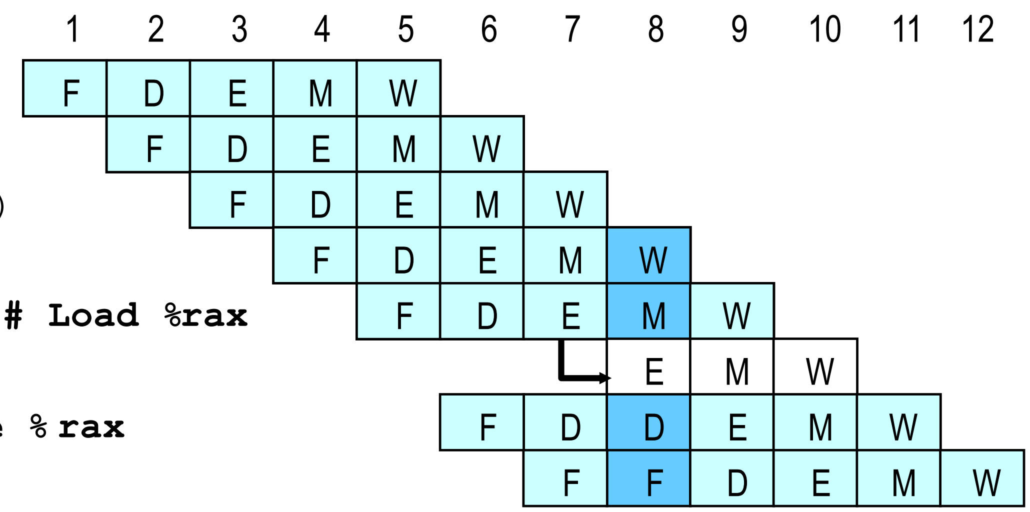
Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	`E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }`

Control for Load/Use Hazard

```
# 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
```



- 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

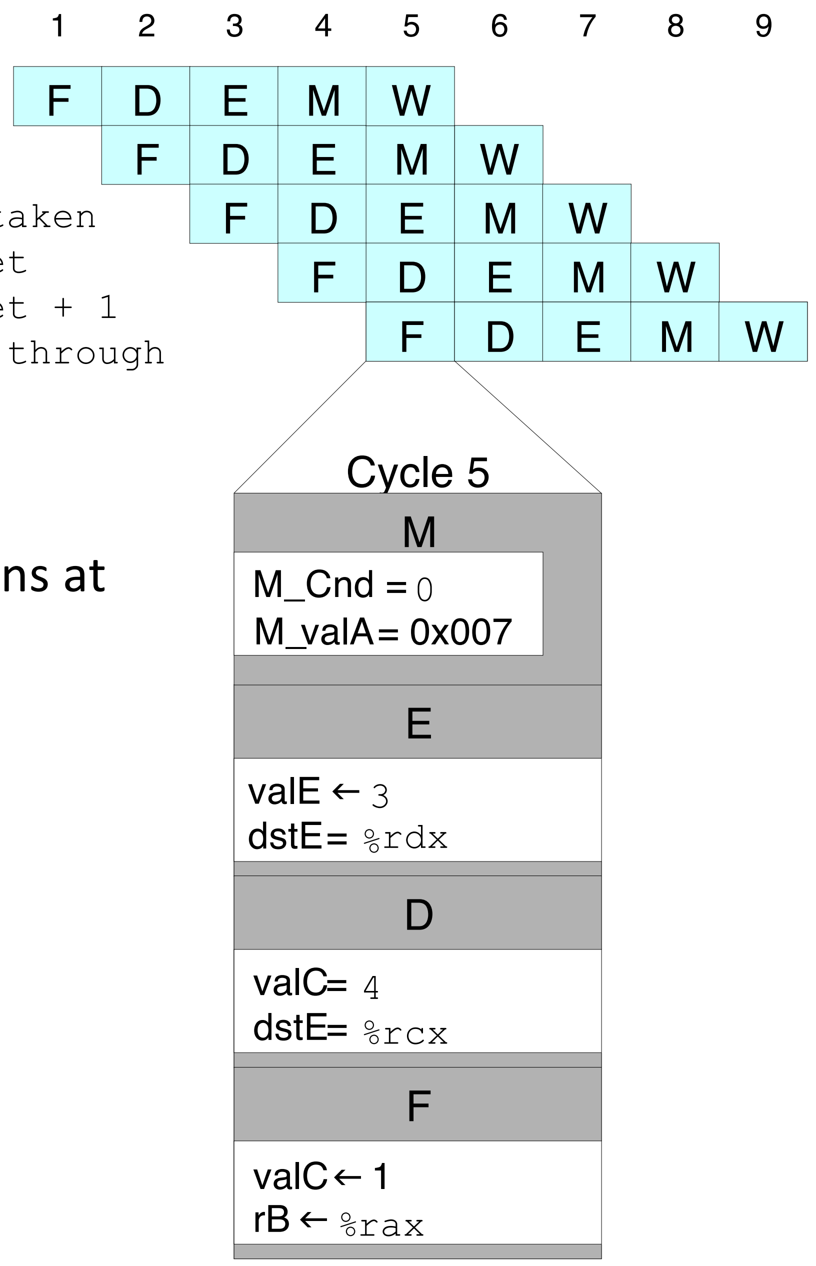
Control Hazard: Branch Mispredictions

```
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 $2, %rdx       # Target
0x023:    irmovq $3, %rcx       # Should not execute
0x02d:    irmovq $4, %rdx       # Should not execute
```

- Should only execute first 7 instructions

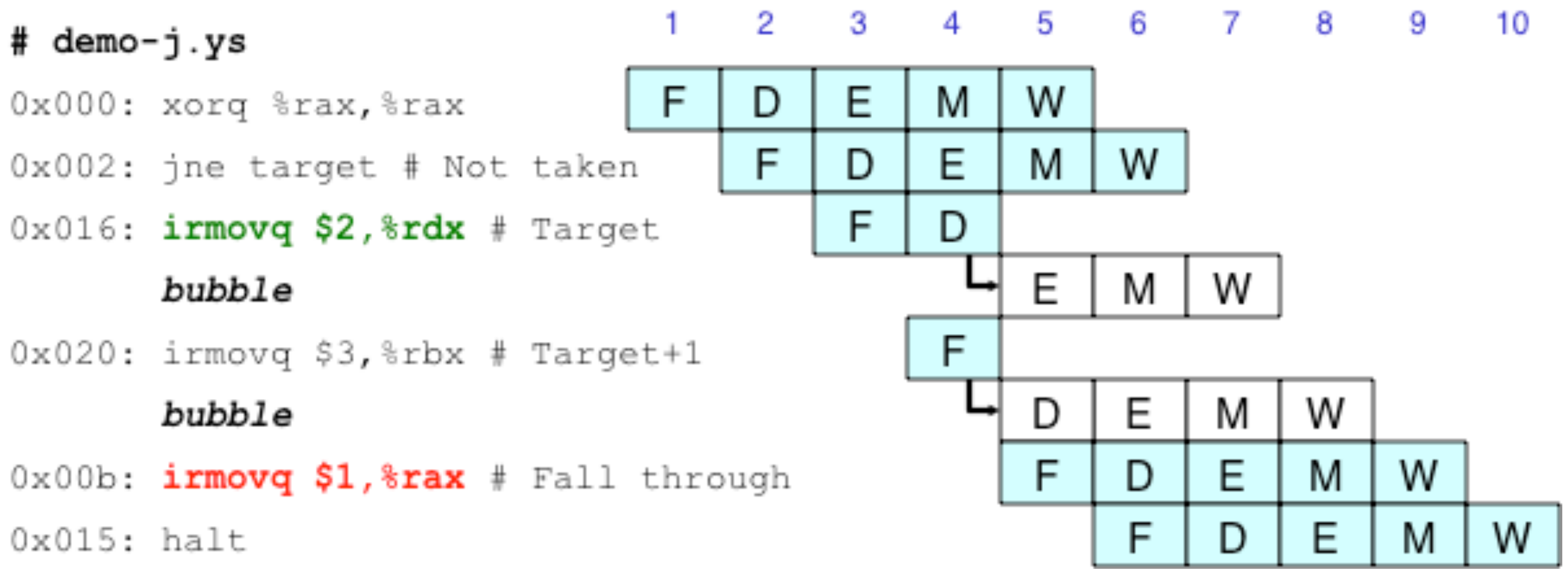
Branch Misprediction Trace

```
# demo-j.ys
0x000:    xorq %rax,%rax
0x002:    jne  t              # Not taken
0x019: t: irmovq $2, %rdx     # Target
0x023:    irmovq $3, %rcx     # Target + 1
0x00b:    irmovq $1, %rax     # Fall through
```



- Incorrectly execute two instructions at branch target

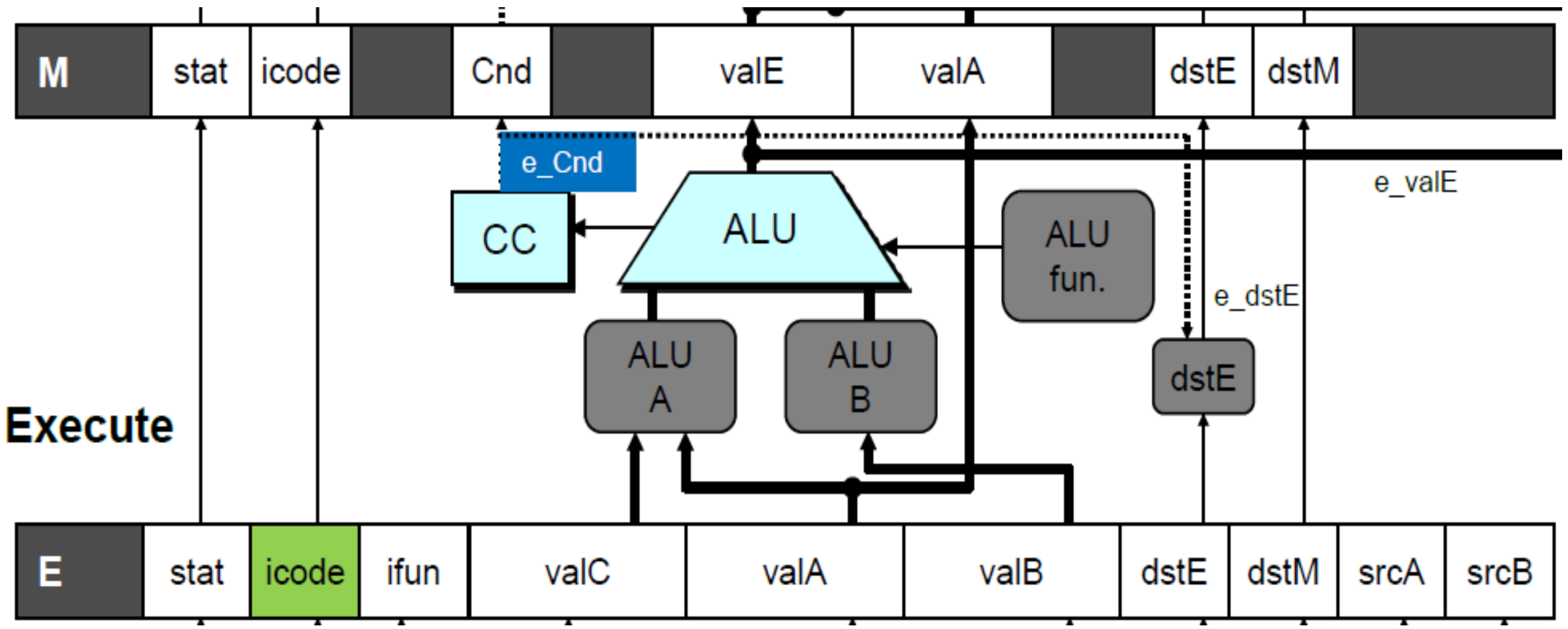
Handling Misprediction



```
# 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

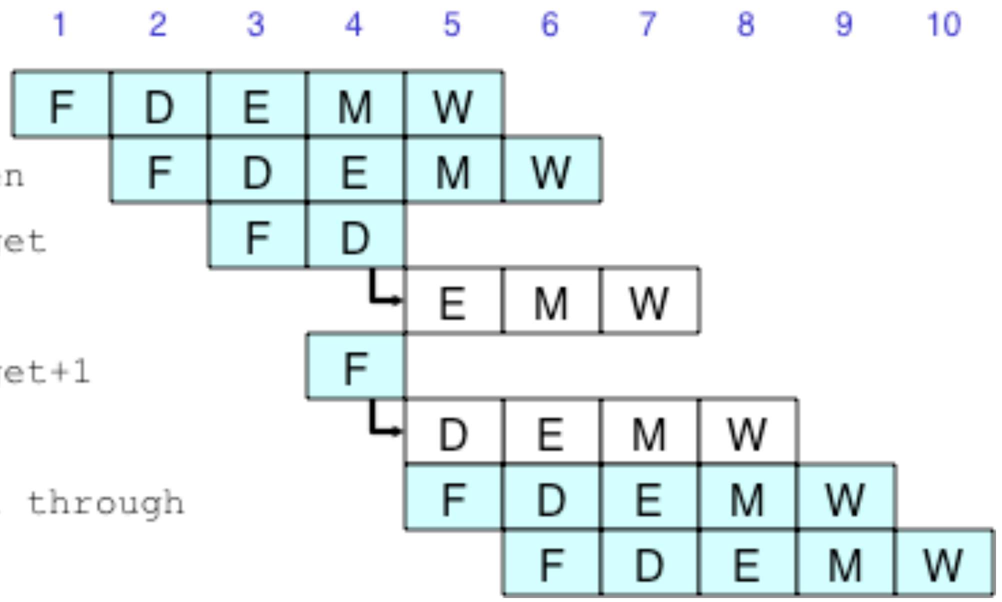
Detecting Mispredicted Branch



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

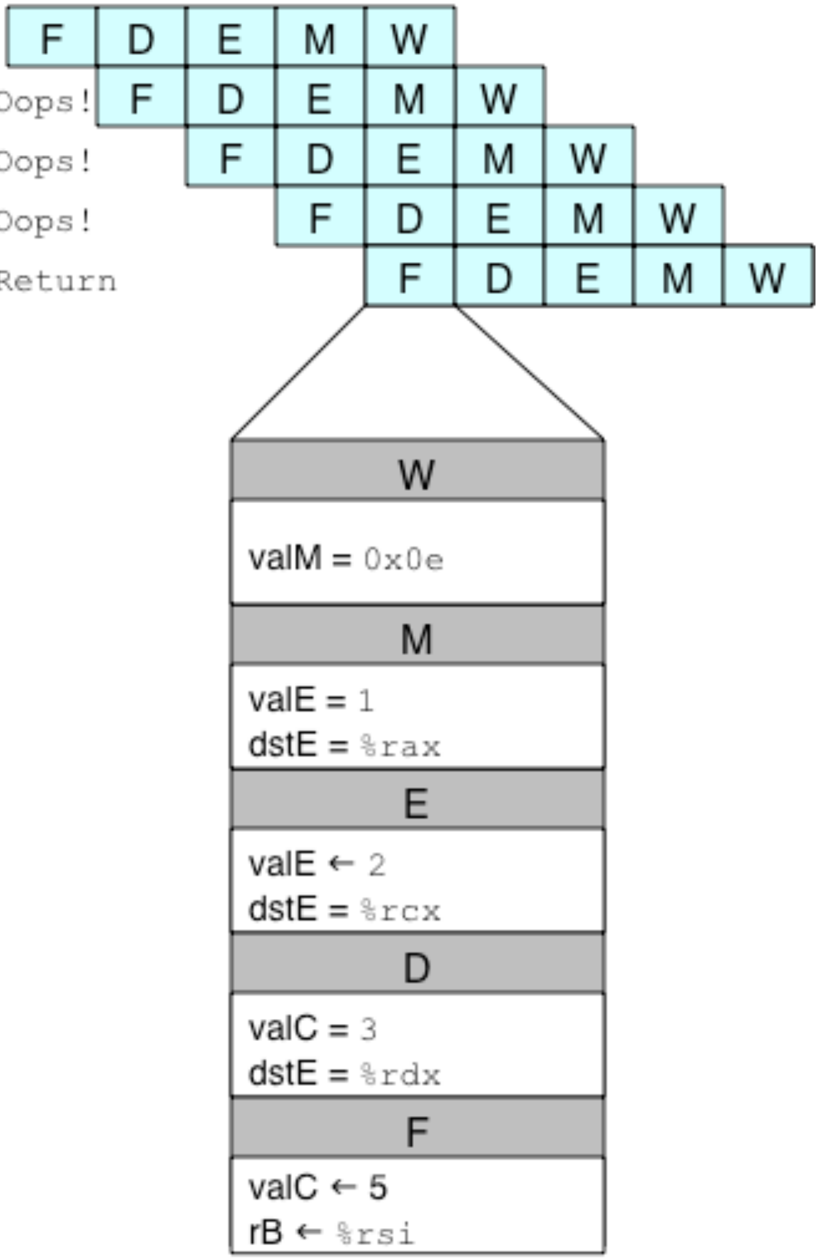
Control Hazard: Dealing with Returns

```
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
```

- Pipeline will execute three additional instructions past ret

Incorrect Return Example

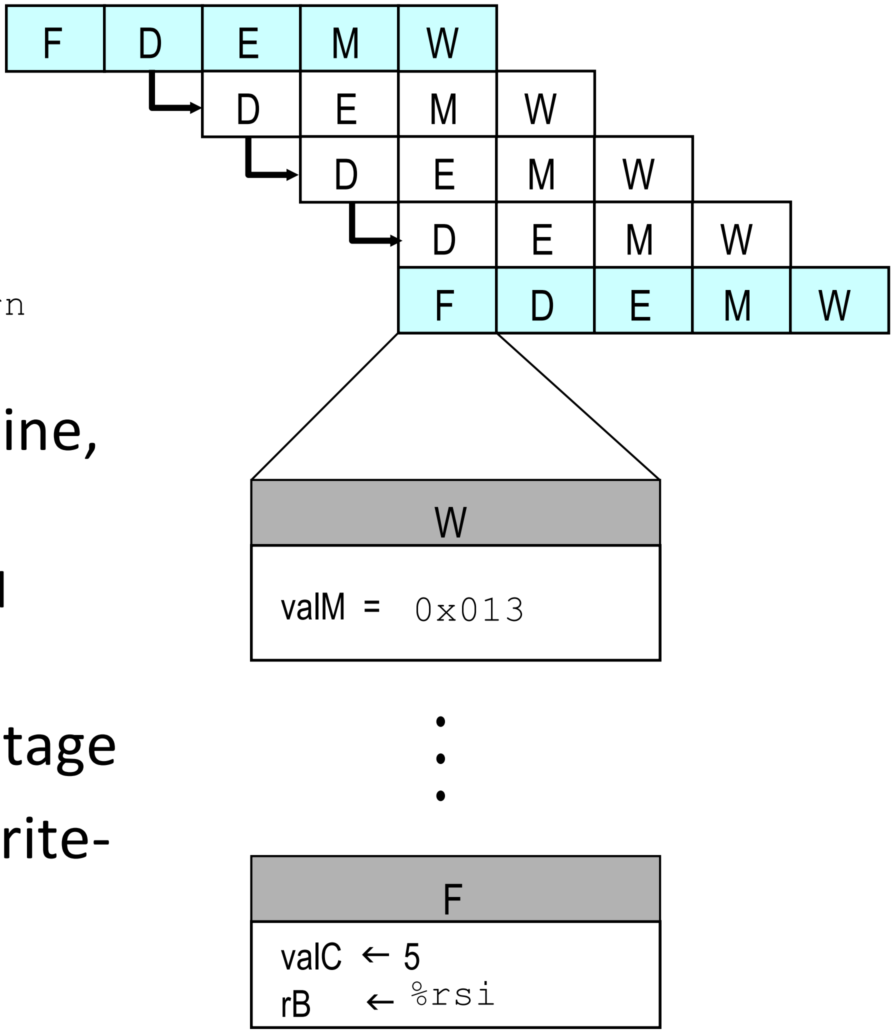
```
# demo-ret

0x033:    ret
0x034:    irmovq $1,%rax # Oops!
0x03e:    irmovq $2,%rcx # Oops!
0x048:    irmovq $3,%rdx # Oops!
0x052:    irmovq $5,%rsi # Return
```



- Incorrectly execute 3 instructions following ret

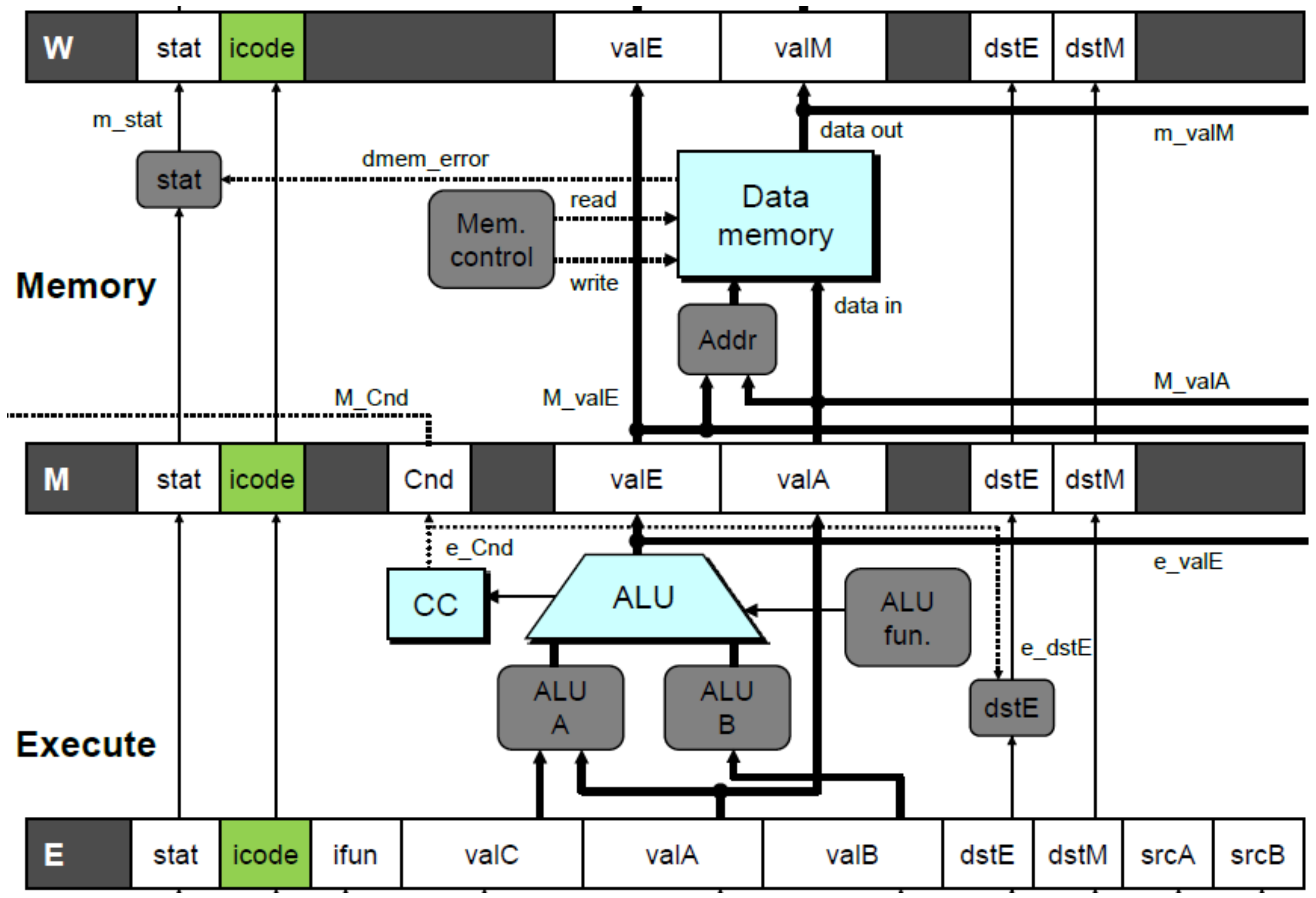
Correct Return Example

```
# 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

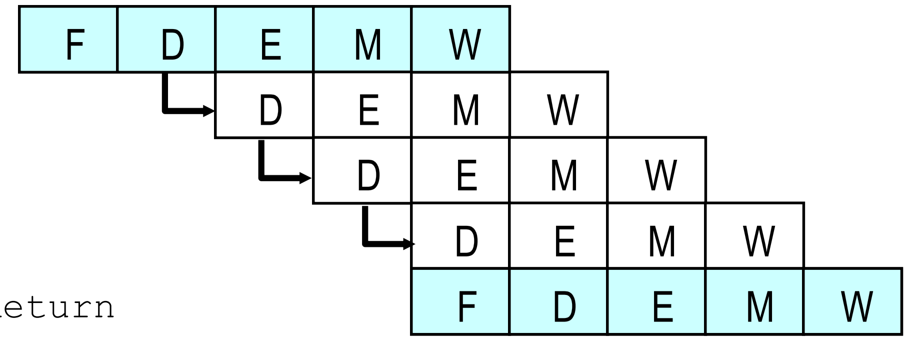
Detecting Return



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

Control for Return

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



Condition	F	D	E	M	W
Processing `ret`	**stall**	**bubble**	**normal**	**normal**	**normal**

Special Control Cases

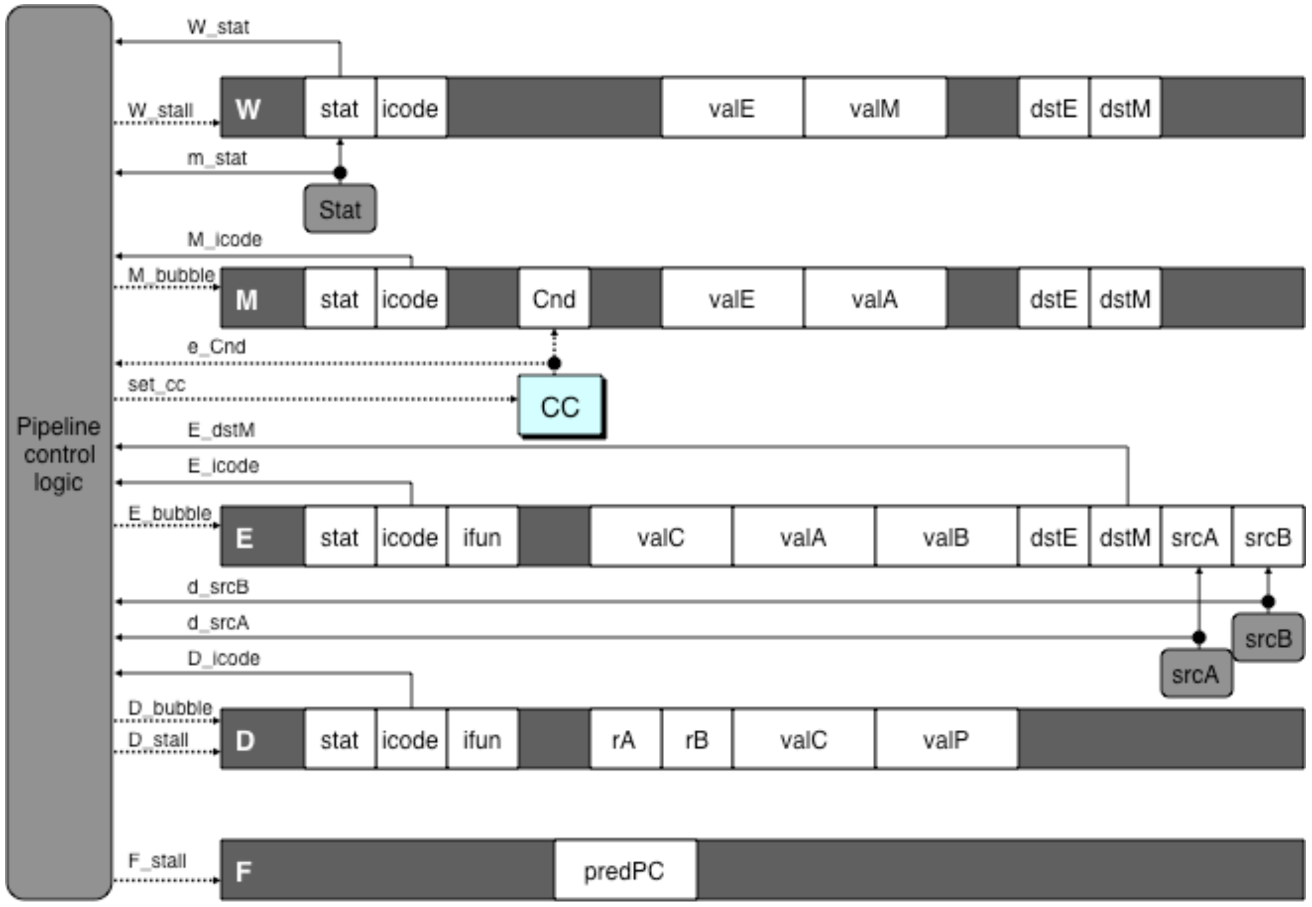
- Detection

Condition	Trigger
Processing `ret`	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && 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

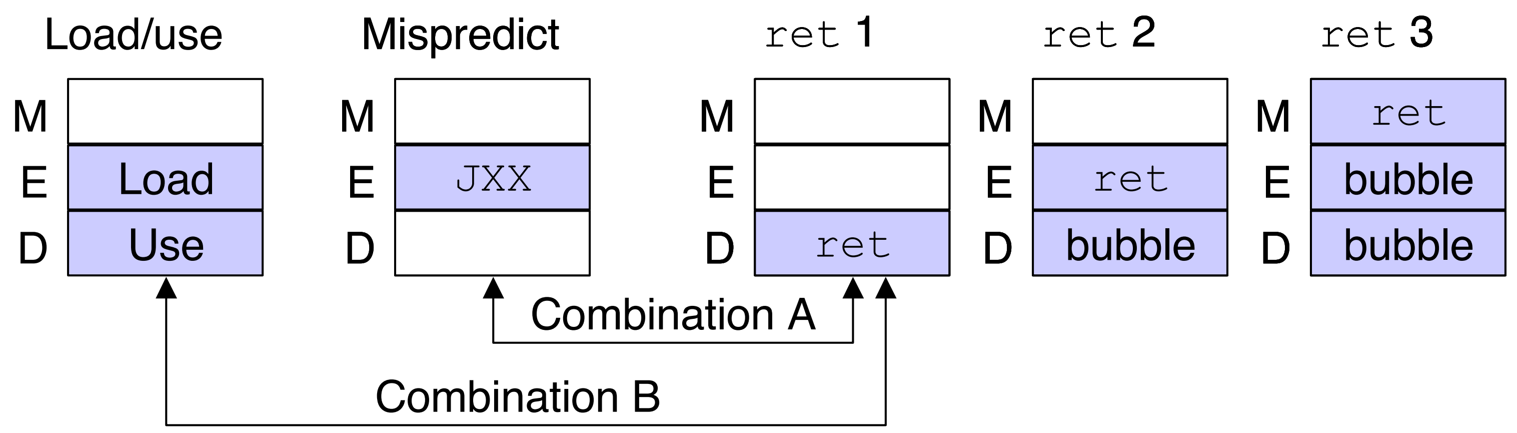


- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle

Initial (Buggy) Version of Pipeline Control

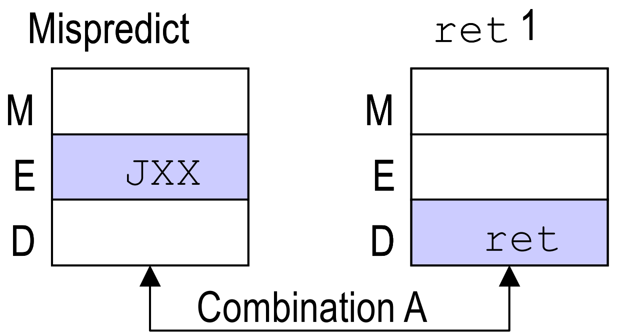
```
bool F_stall =
     # Conditions for a load/use hazard
     E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB } ||
     # Stalling at fetch while ret passes through pipeline
     IRET in { D_icode, E_icode, M_icode };

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 };
```

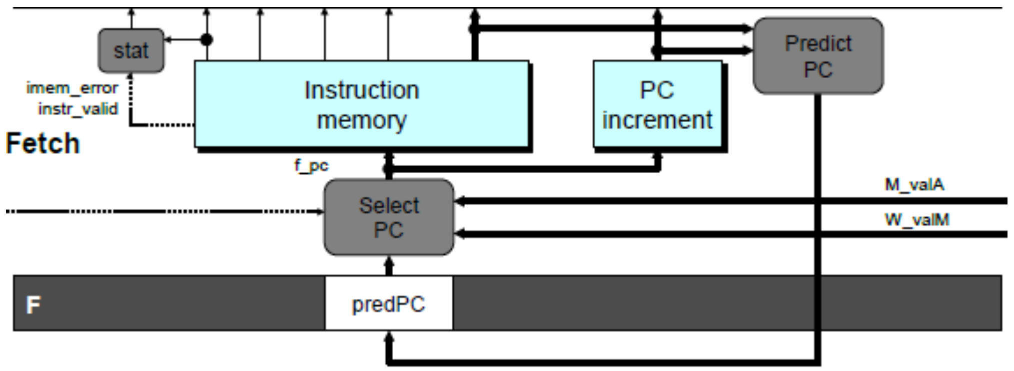
Control Combinations



- 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

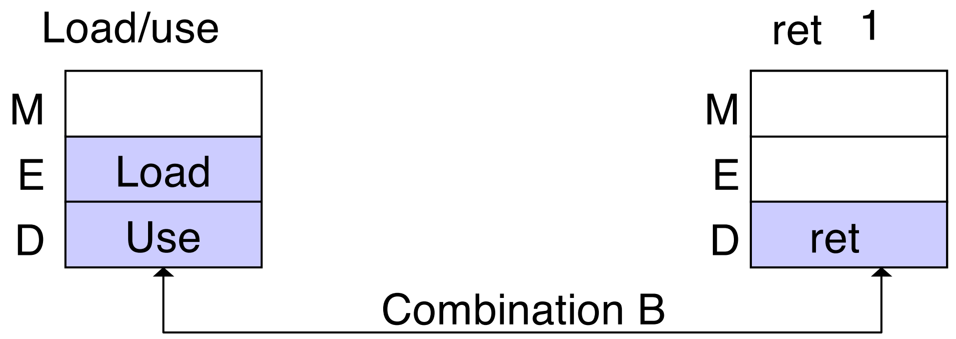




Condition	F	D	E	M	W
Processing ret	**stall**	**bubble**	**normal**	**normal**	**normal**
Mispredicted Branch	**normal**	**bubble**	**bubble**	**normal**	**normal**
Combination	***stall***	***bubble***	***bubble***	***normal***	***normal***

- Should handle as mispredicted branch
- Stalls F pipeline register
- But PC selection logic will be using M_valM anyway

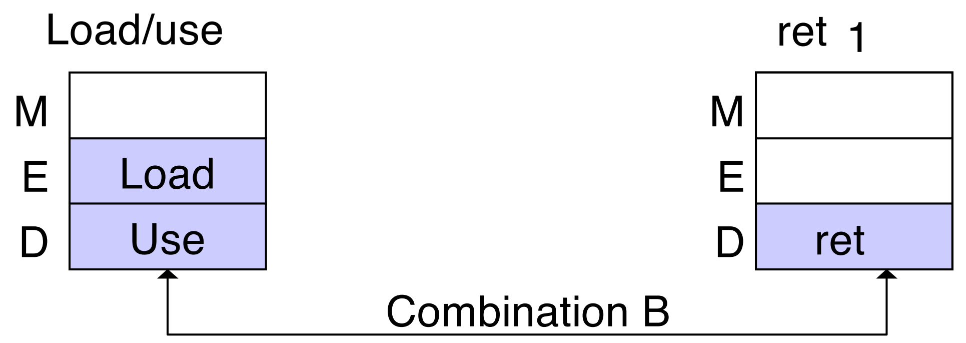
Control Combination B



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

- 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**
Combination	***stall***	***stall***	***bubble***	***normal***	***normal***

- 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**
Combination	***stall***	***stall***	***bubble***	***normal***	***normal***

- 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 combinations