Processor Architecture: Pipeline

COMP402127: Introduction to Computer Systems

https://xjtu-ics.github.io/

Danfeng Shan
Xi’an Jiaotong University

Today

- ☐ Overview
- ☐ Y86-64 Instruction Set Architecture
- ☐ Logic Design
- ☐ Sequential Implementation
- ☐ Pipelined Implementation: Principle
- ☐ **Pipelined Implementation: Implementation**
- ☐ Wrap-Up

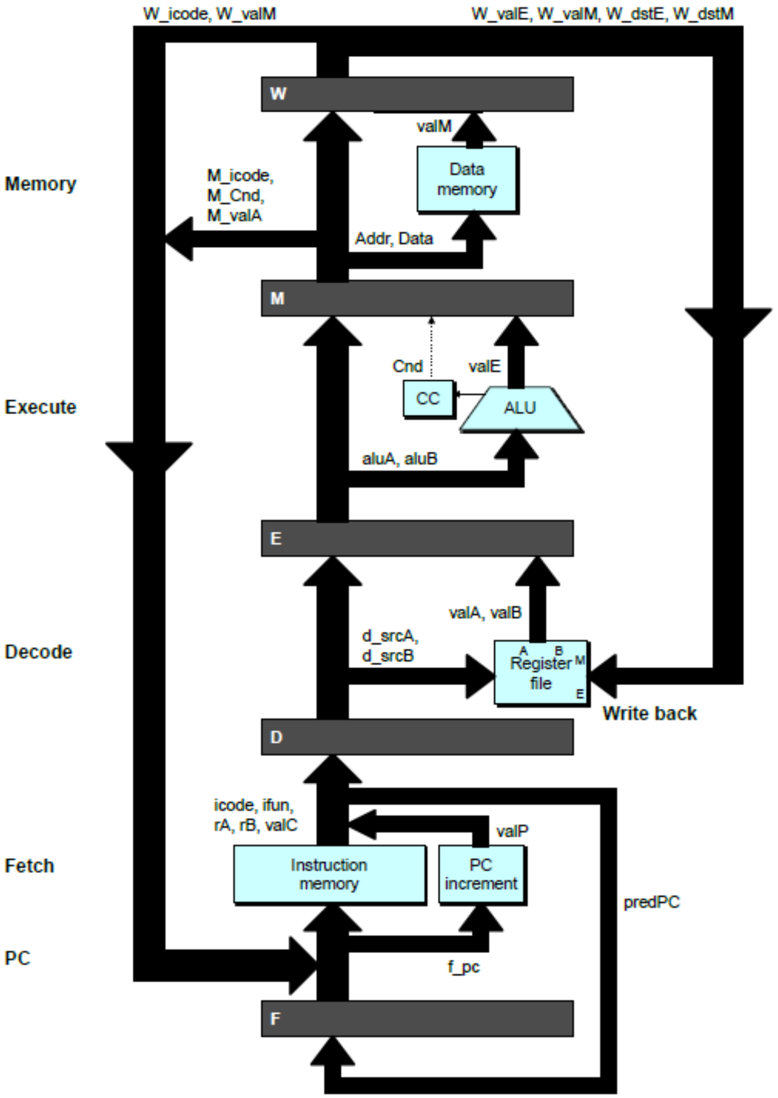
Overview

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**
  - Mispredict 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

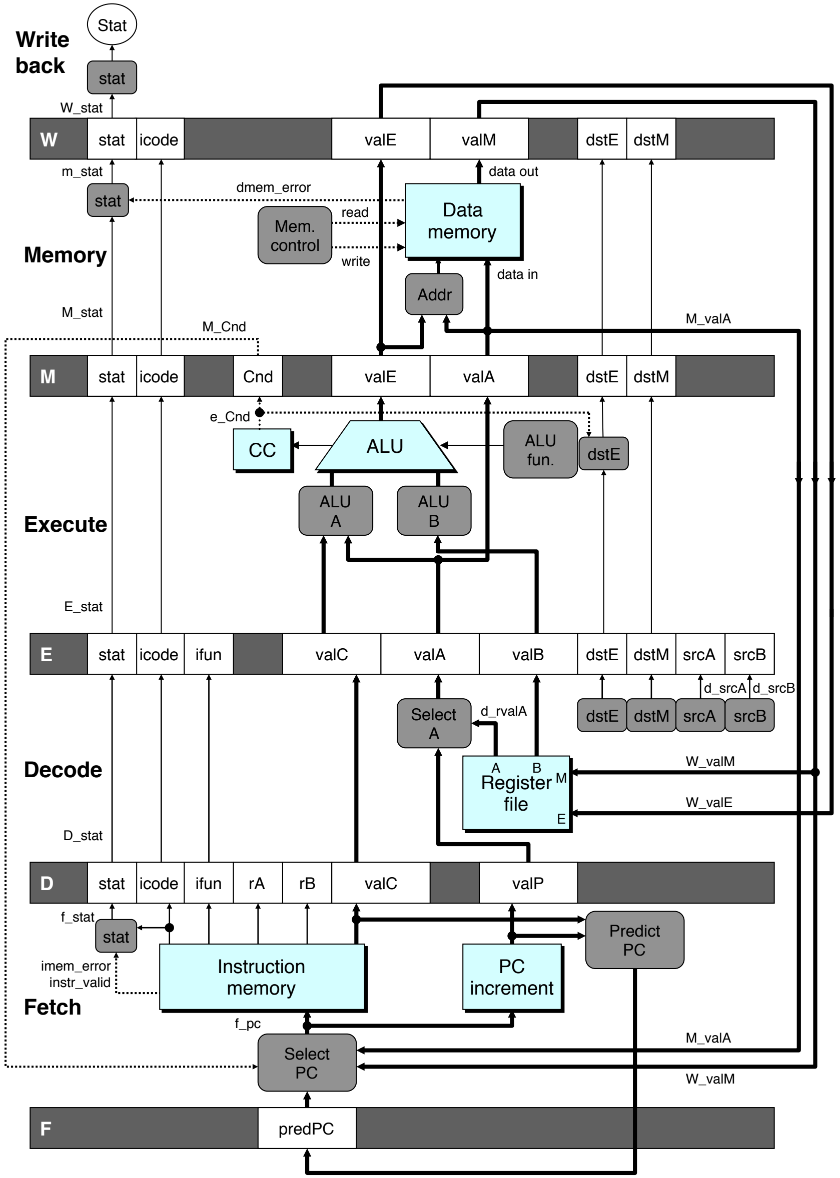


PIPE- Hardware

- Pipeline registers hold intermediate values from instruction execution

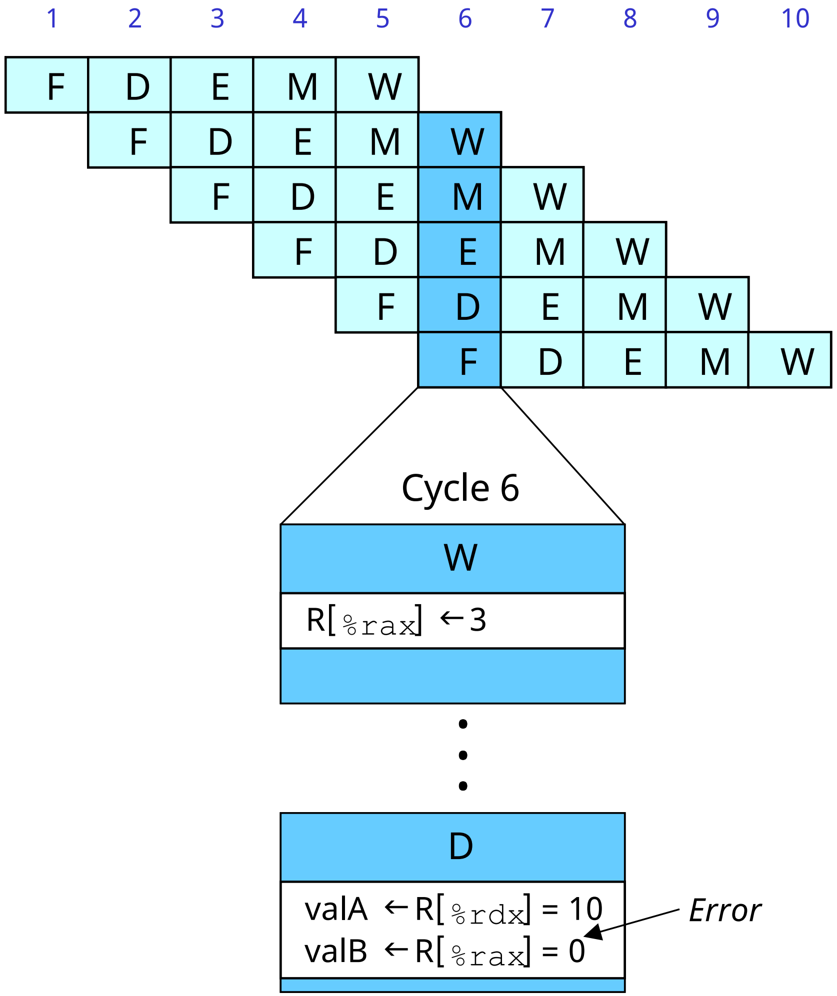
Forward (Upward) Paths

- Values passed from one stage to next
- Cannot jump past stages
  - e.g., valC passes through decode



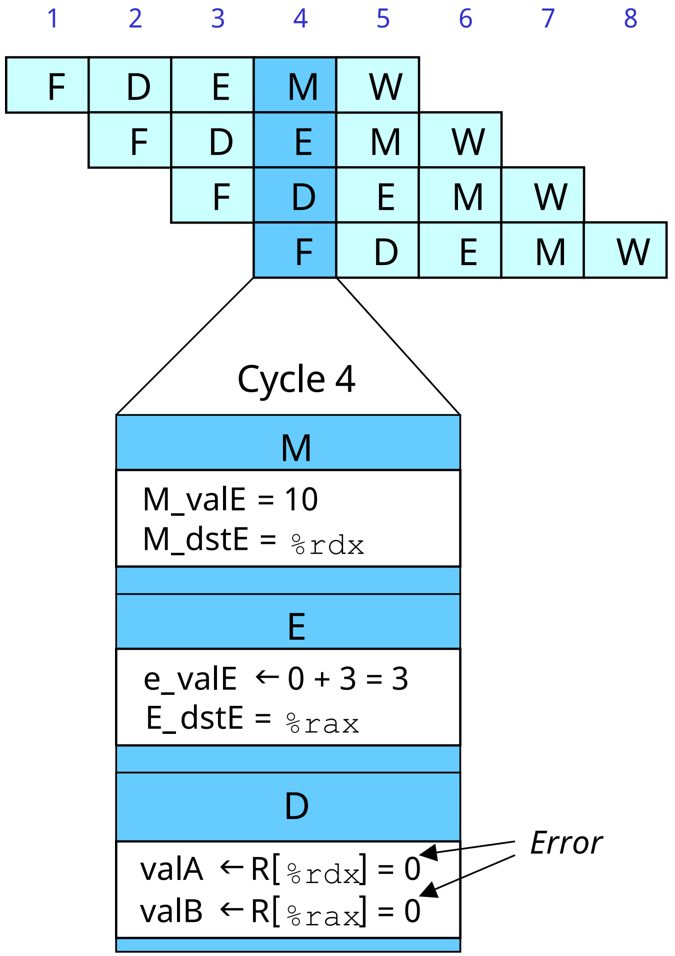
Data Dependencies: 2 Nop’s

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



Data Dependencies: No Nop

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



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

Cycle 4

M

M_valE = 10
M_dstE = `%rdx`

E

e_valE ← 0 + 3 = 3
E_dstE = `%rax`

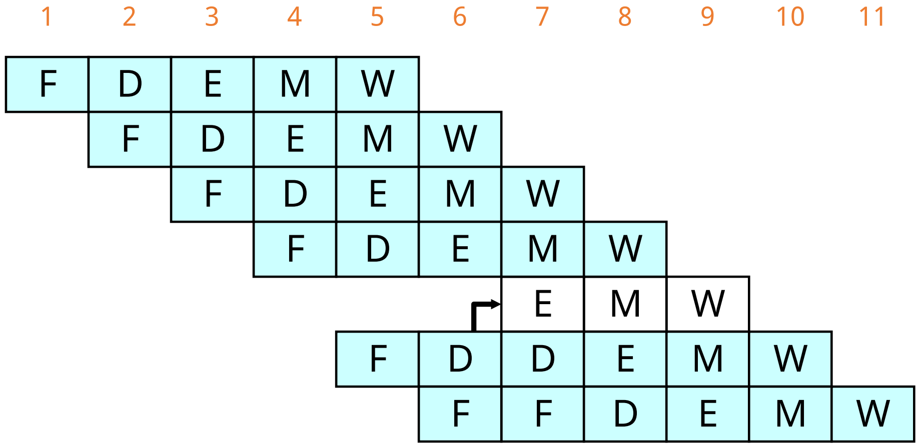
D

valA ← R[`%rdx`] = 0
valB ← R[`%rax`] = 0

Error

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
```

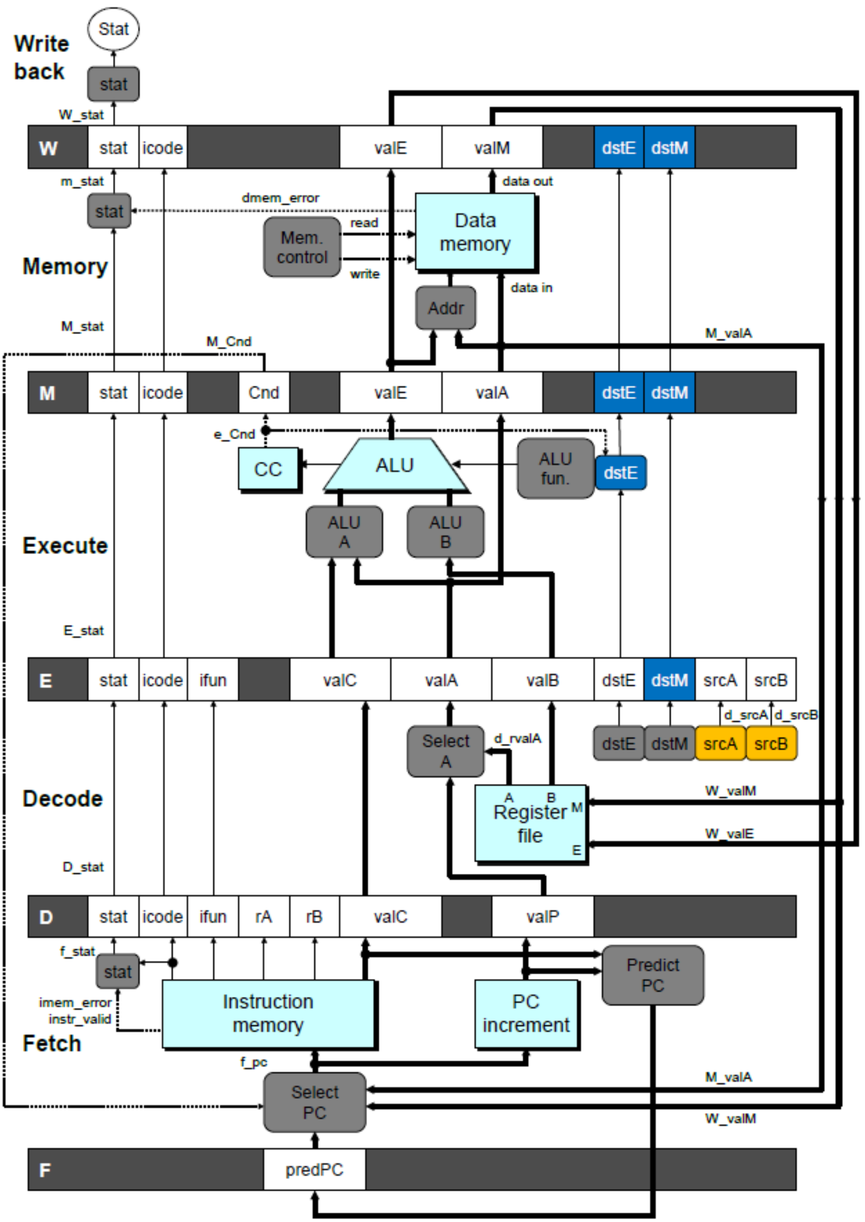


1	2	3	4	5	6	7	8	9	10	11
F	D	E	M	W						
	F	D	E	M	W					
		F	D	E	M	W				
			F	D	E	M	W			
						E	M	W		
				F	D	D	E	M	W	
					F	F	D	E	M	W

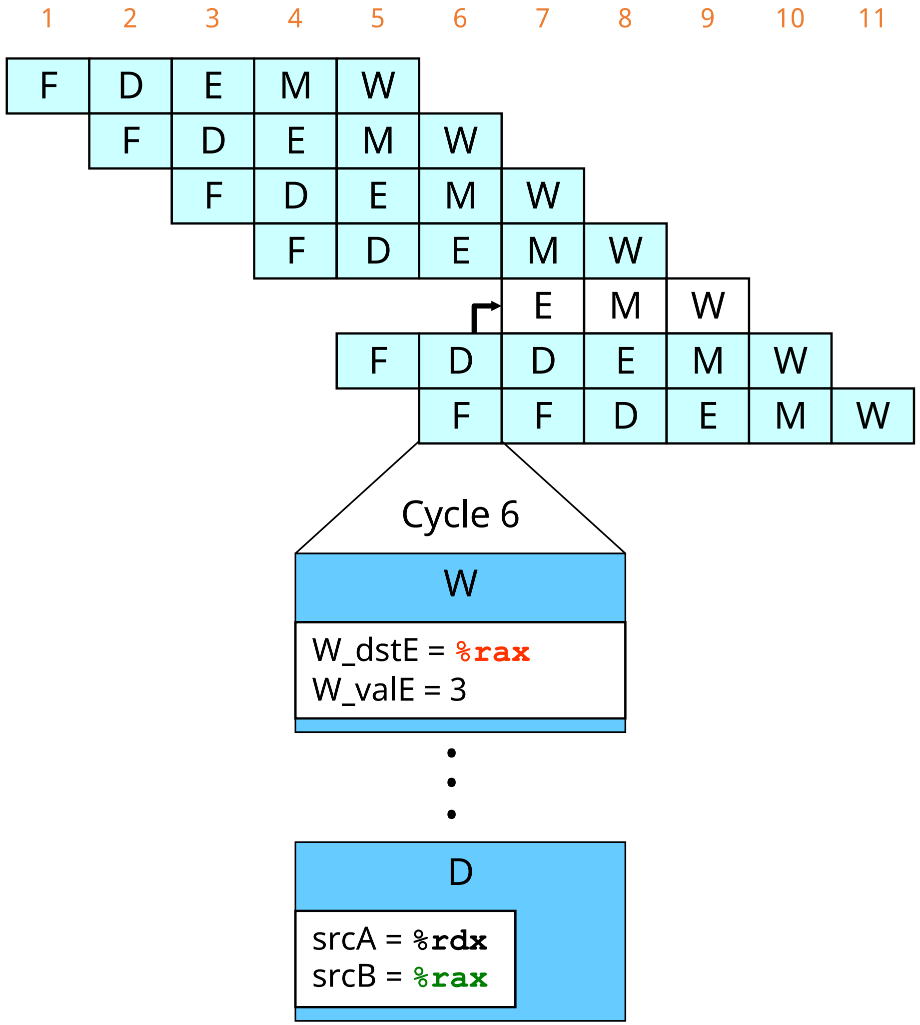
- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

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	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

Cycle 6

W

W_dstE = **%rax**
W_valE = 3

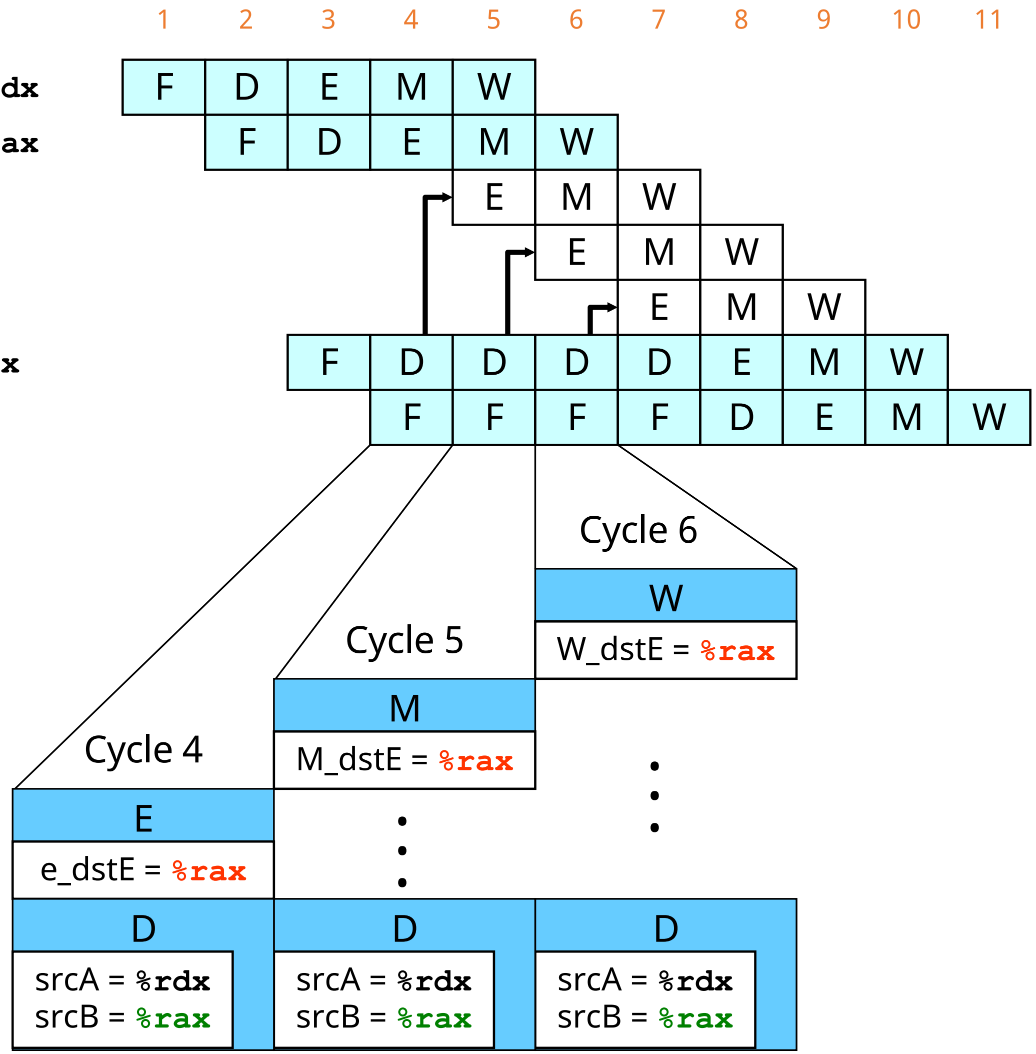
•
•
•

D

srcA = **%rdx**
srcB = **%rax**

Stalling X3

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



1	2	3	4	5	6	7	8	9	10	11
F	D	E	M	W						
	F	D	E	M	W					
				E	M	W				
					E	M	W			
						E	M	W		
		F	D	D	D	D	E	M	W	
			F	F	F	F	D	E	M	W

What Happens When Stalling?

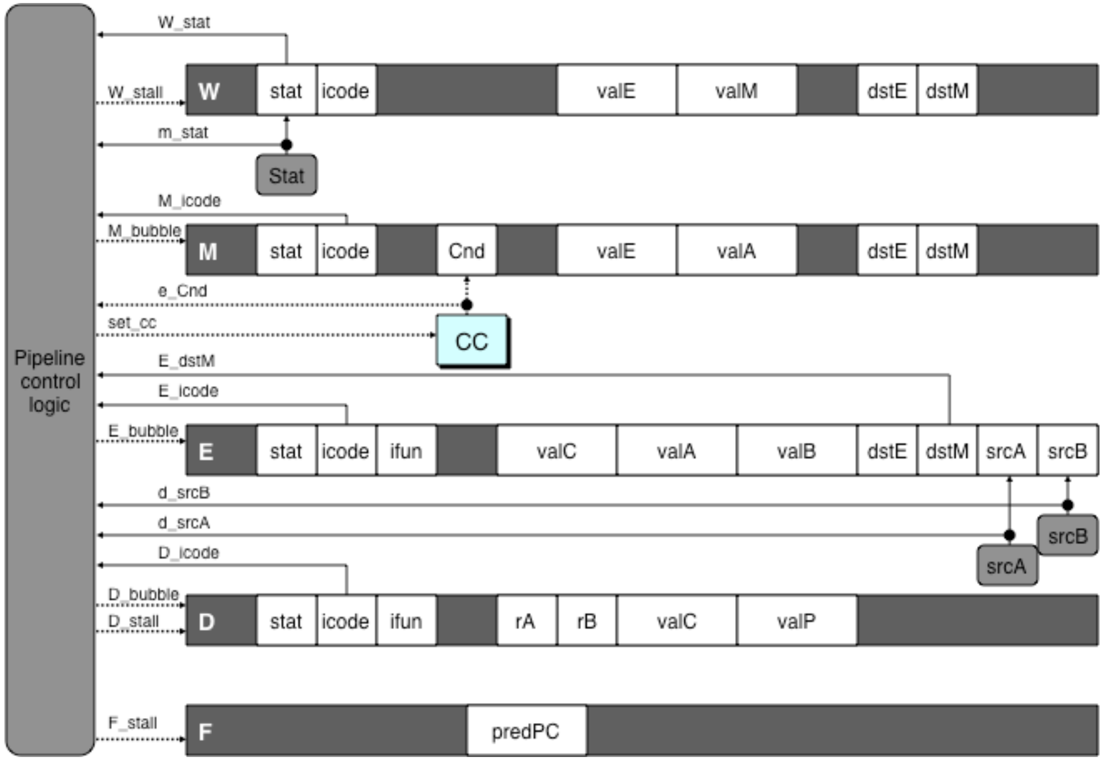
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

Cycle 8

Write Back	***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

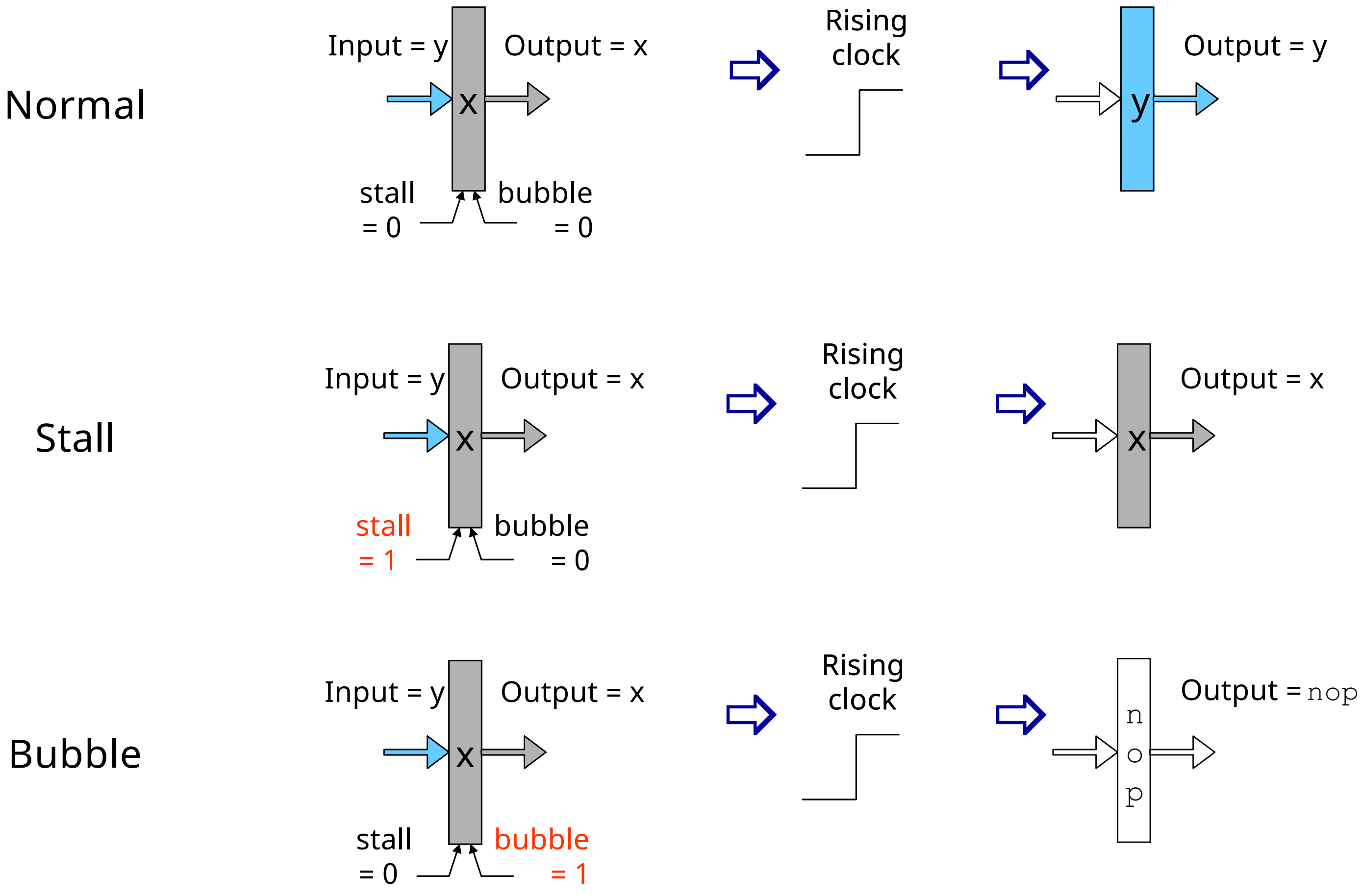
Implementing Stalling



Pipeline Control

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

Pipeline Register Modes



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 generated in execute or memory stage

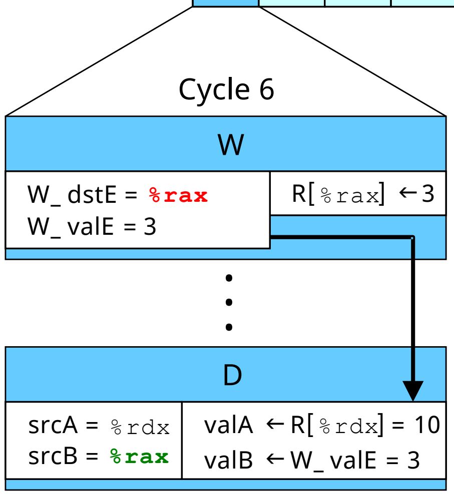
Trick

- 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
```

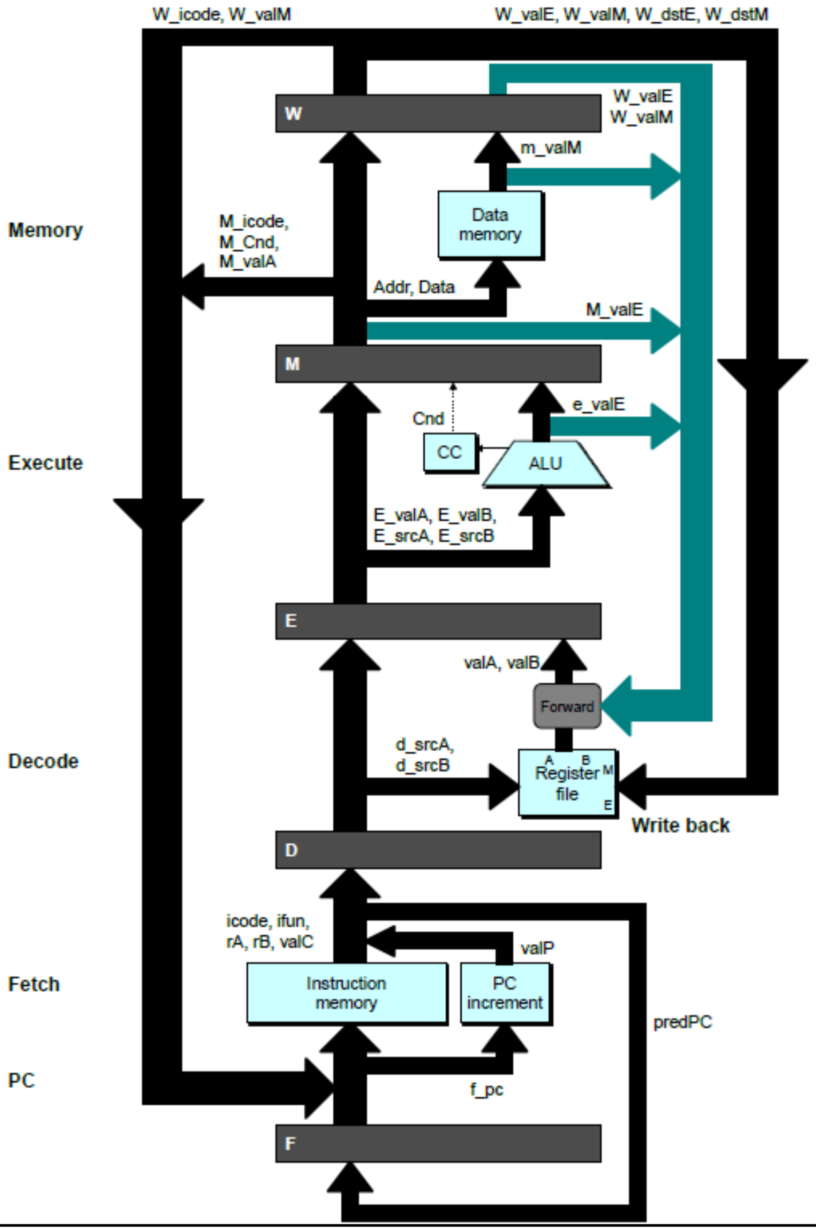
1	2	3	4	5	6	7	8	9	10
F	D	E	M	W					
	F	D	E	M	W				
		F	D	E	M	W			
			F	D	E	M	W		
				F	D	E	M	W	
					F	D	E	M	W



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

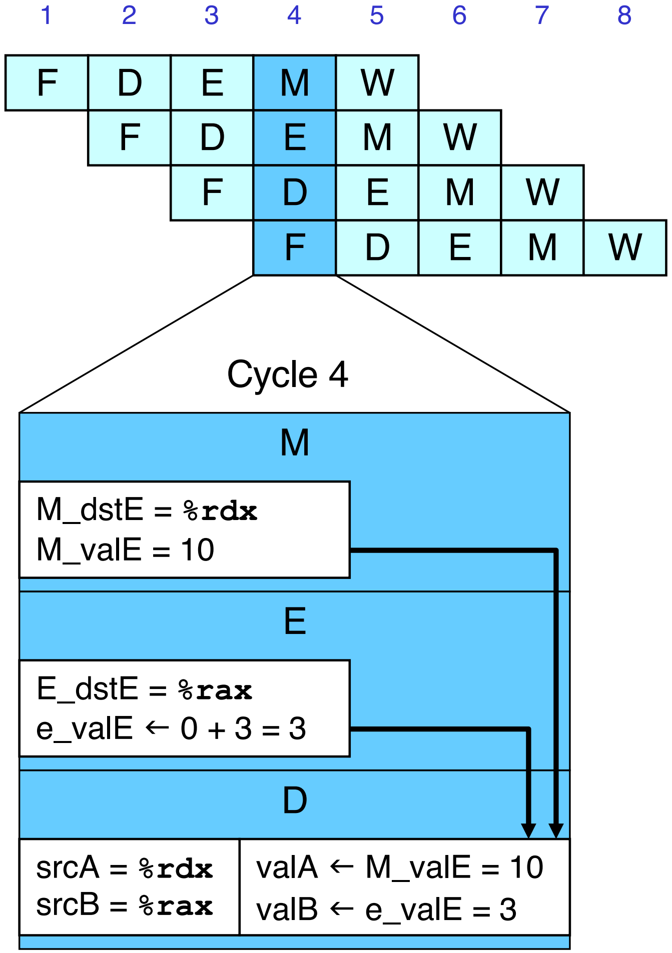
Bypass Paths

- ☐ **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
```



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

Cycle 4

M

M_dstE = `%rdx`
M_valE = 10

E

E_dstE = `%rax`
e_valE ← 0 + 3 = 3

D

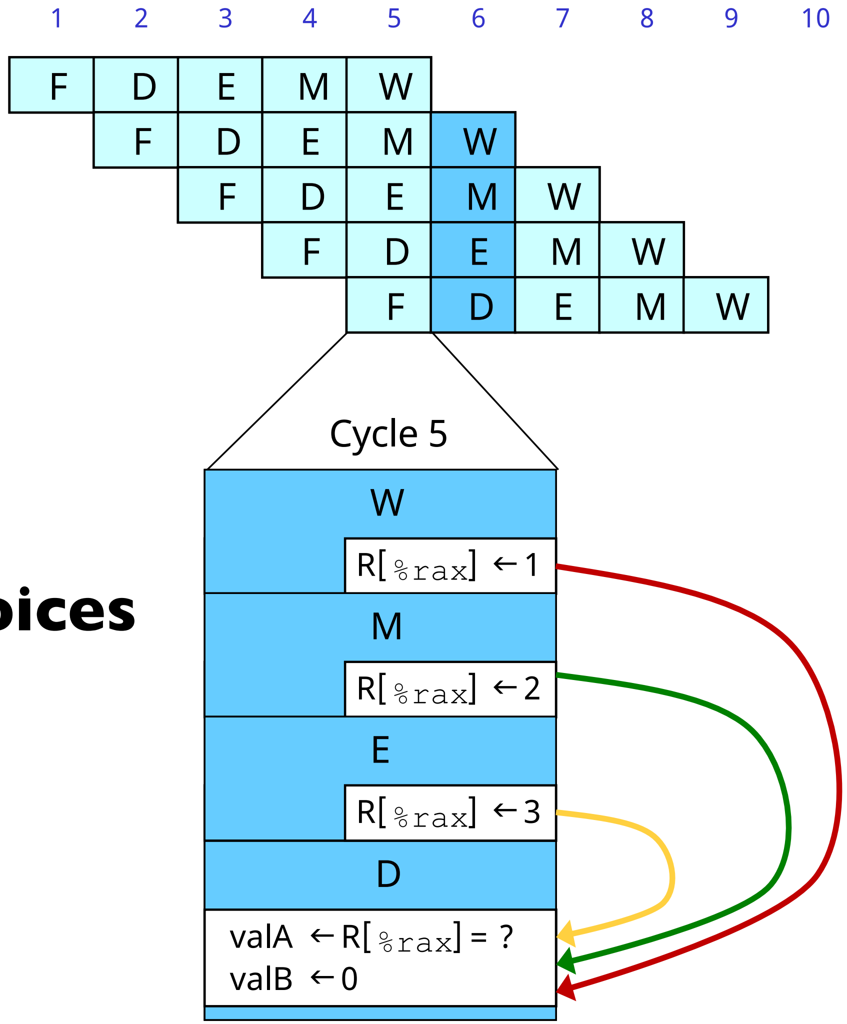
srcA = `%rdx`
srcB = `%rax`

valA ← M_valE = 10
valB ← e_valE = 3

- ☐ **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

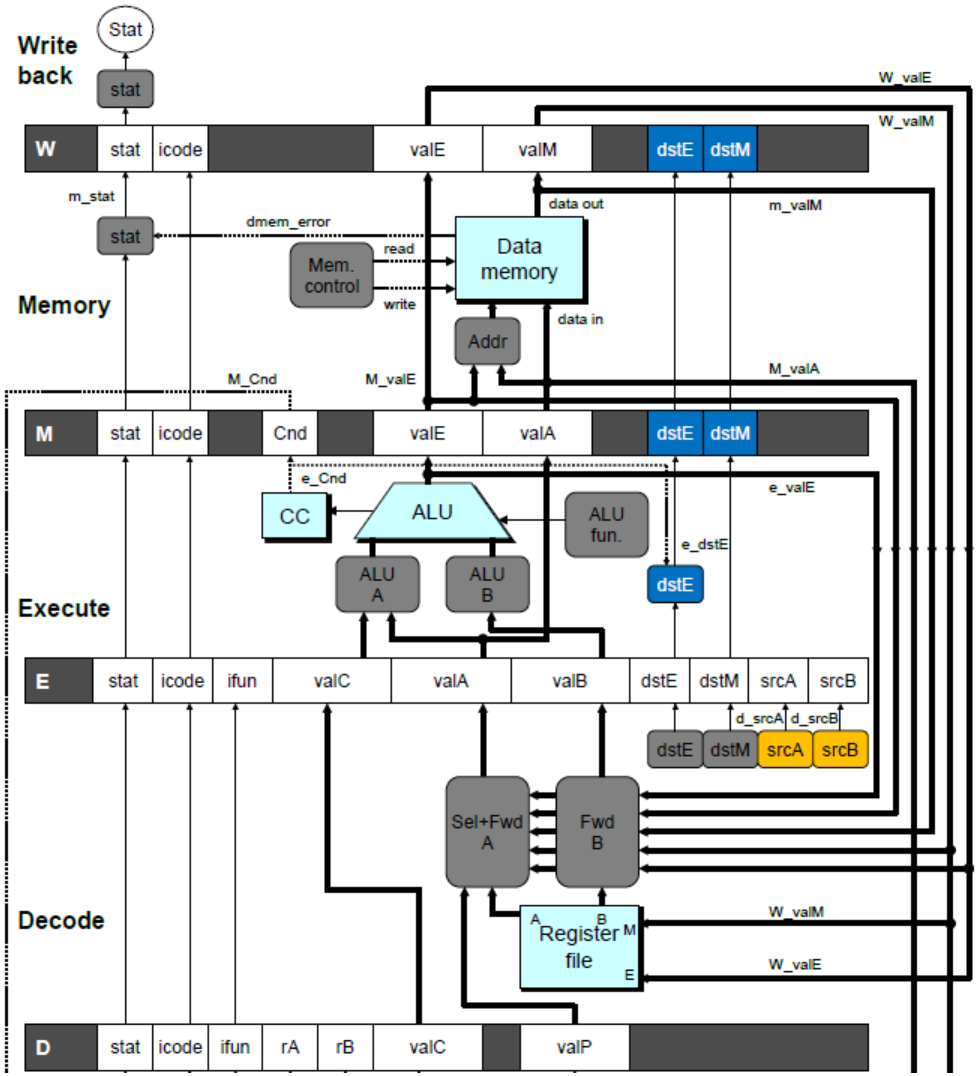
```
# demo-priority.ys
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
 0x01e: rrmovq %rax, %rdx
0x020: halt
```



Multiple Forwarding Choices

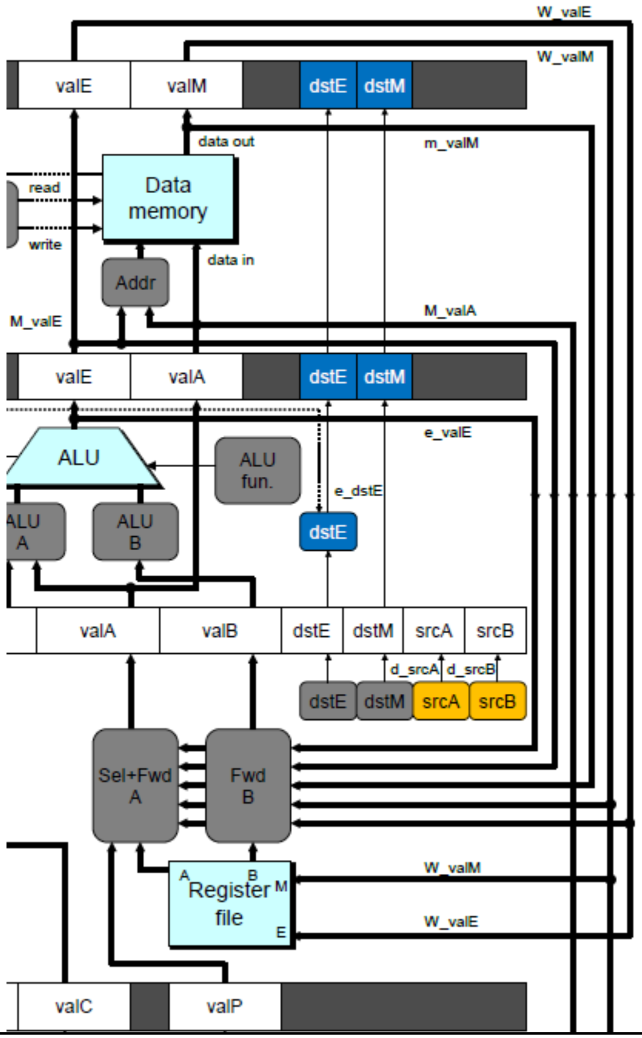
- Which one should have priority
- Match serial semantics
- Use matching value from earliest pipeline stage

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

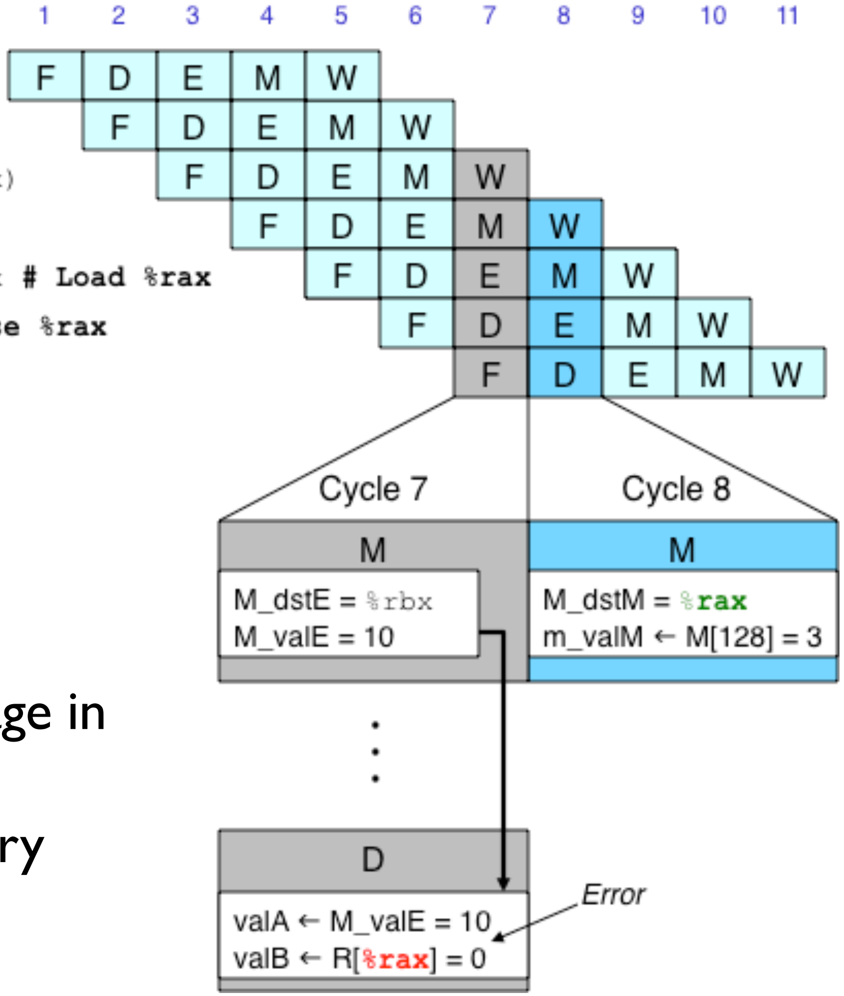
Implementing Forwarding



```
## 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

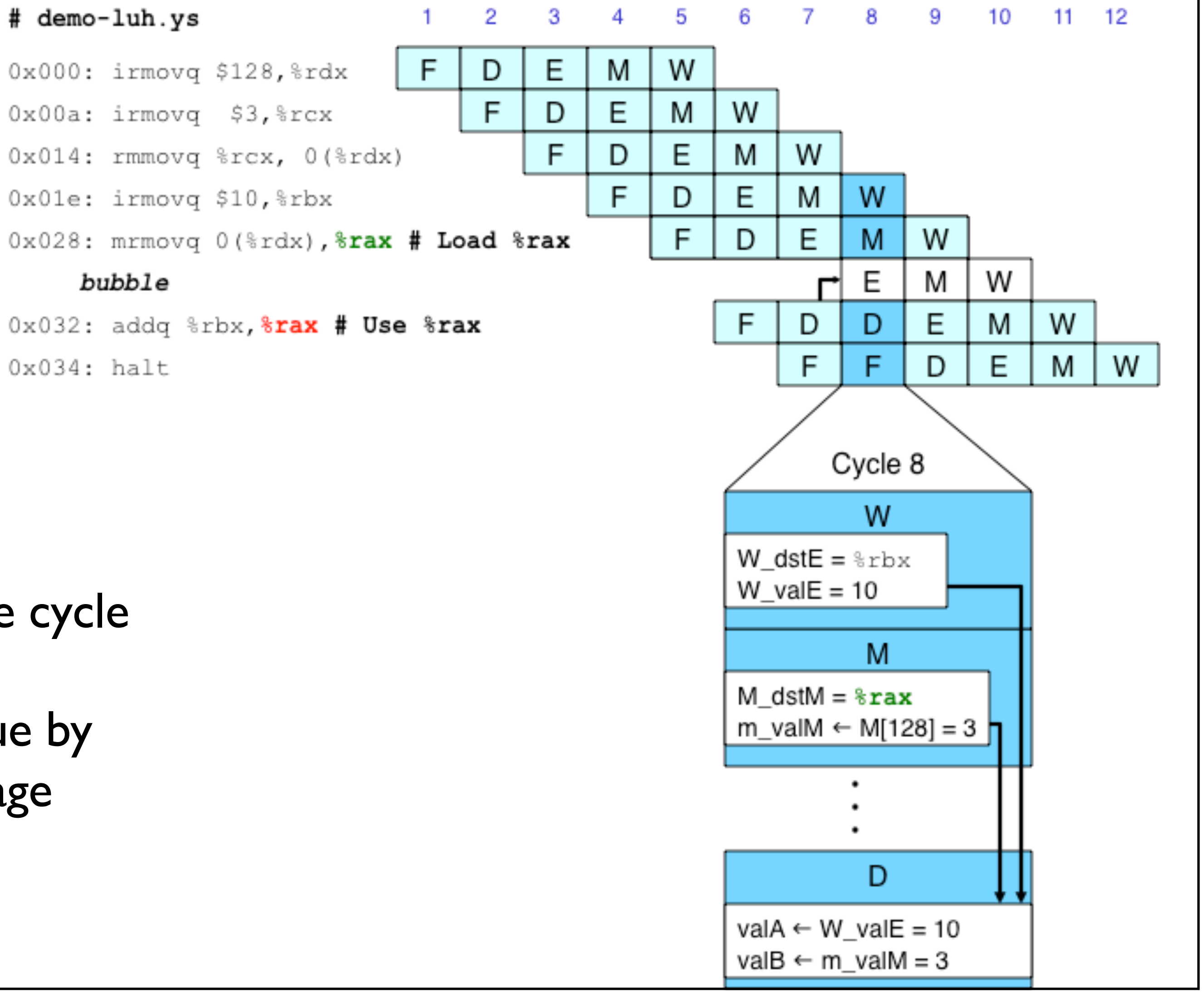
```
# demo-luh.ys

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
0x000	F	D	E	M	W							
0x00a		F	D	E	M	W						
0x014			F	D	E	M	W					
0x01e				F	D	E	M	W				
0x028					F	D	E	M	W			
bubble								E	M	W		
0x032						F	D	D	E	M	W	
0x034							F	F	D	E	M	W

Cycle 8

W

W_dstE = %rbx
W_valE = 10

M

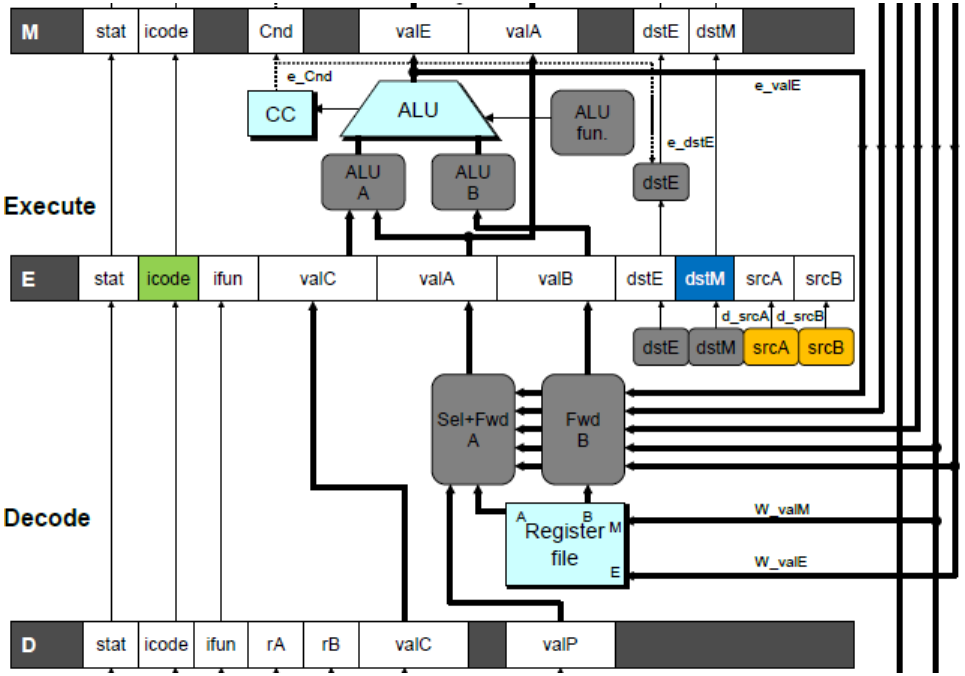
M_dstM = %rax
m_valM ← M[128] = 3

D

valA ← W_valE = 10
valB ← m_valM = 3

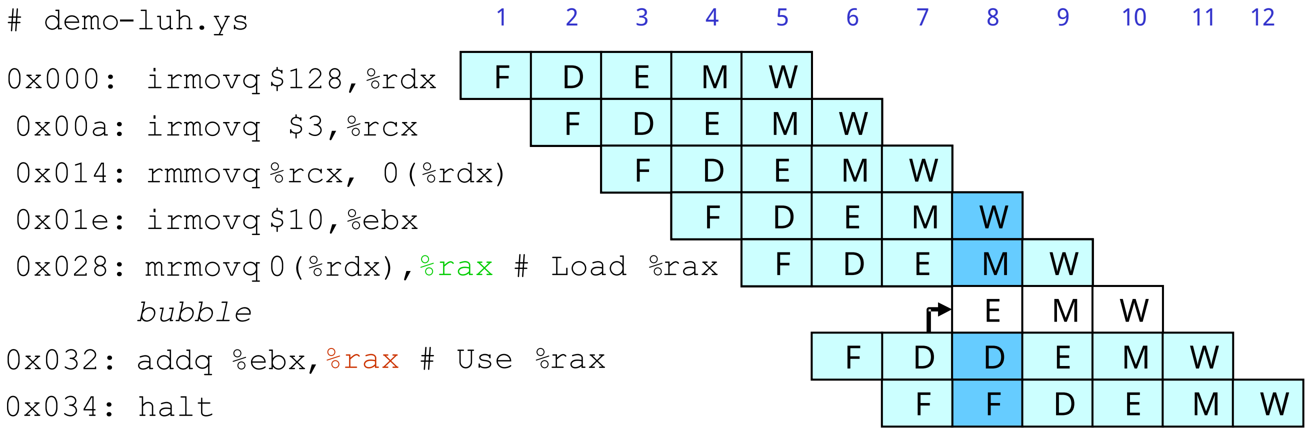
- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

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	1	2	3	4	5	6	7	8	9	10	11	12
`0x000: irmovq $128,%rdx`	F	D	E	M	W							
`0x00a: irmovq  $3,%rcx`		F	D	E	M	W						
`0x014: rmmovq %rcx, 0(%rdx)`			F	D	E	M	W					
`0x01e: irmovq $10,%ebx`				F	D	E	M	W				
`0x028: mrmovq 0(%rdx),%rax # Load %rax`					F	D	E	M	W			
bubble								E	M	W		
`0x032: addq %ebx,%rax # Use %rax`						F	D	D	E	M	W	
`0x034: halt`							F	F	D	E	M	W

- 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

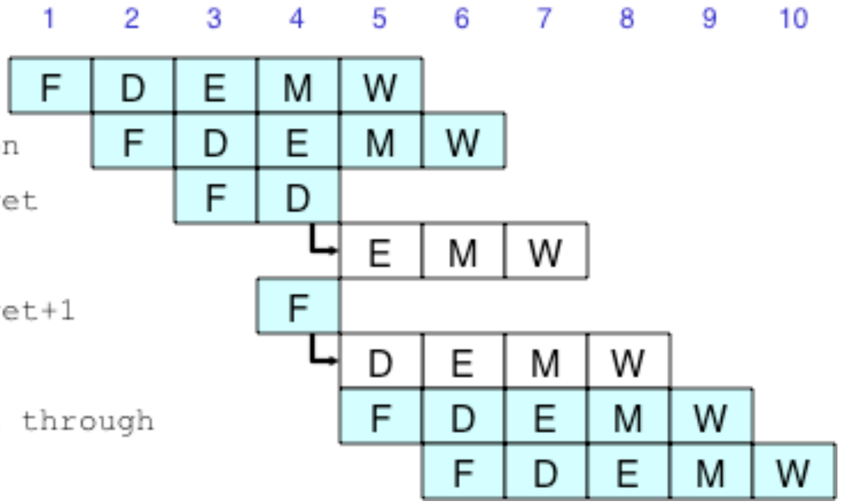
Branch Misprediction Example

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 7 instructions

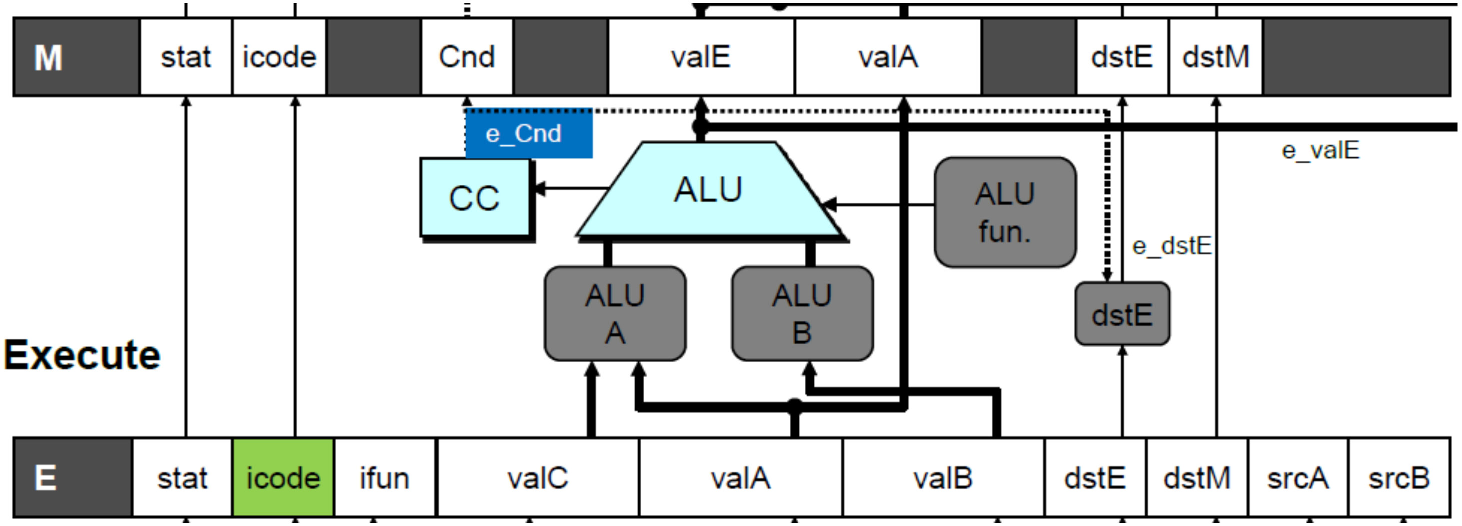
Handling Misprediction



# demo-j.ys	1	2	3	4	5	6	7	8	9	10
`0x000: xorq %rax,%rax`	F	D	E	M	W					
`0x002: jne target # Not taken`		F	D	E	M	W				
`0x016: irmovq $2,%rdx # Target`			F	D						
bubble					E	M	W			
`0x020: irmovq $3,%rbx # Target+1`				F						
bubble					D	E	M	W		
`0x00b: irmovq $1,%rax # Fall through`					F	D	E	M	W	
`0x015: halt`						F	D	E	M	W

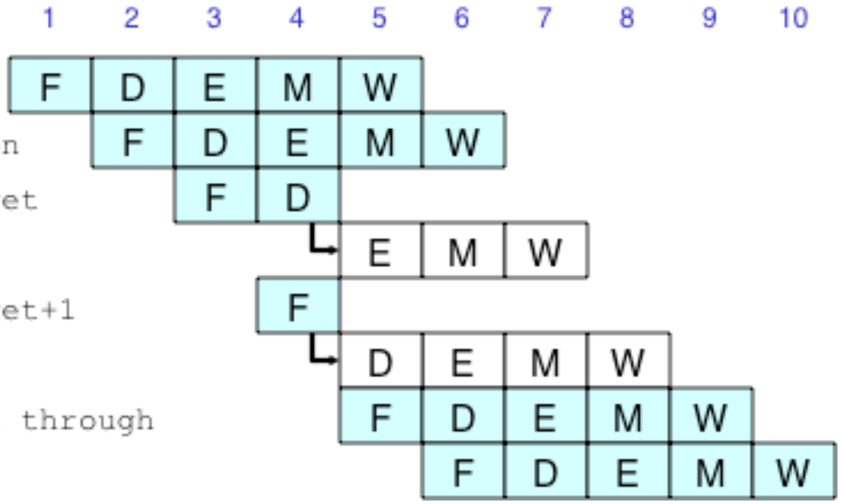
- ☐ **Predict branch as taken**
  - Fetch 2 instructions at target
- ☐ **Cancel when mispredicted**
  - Detect branch not-taken in execute stage
  - On following cycle, replace instructions in execute and decode by bubbles
  - No side effects have occurred yet

Detecting Mispredicted Branch



Condition	**Trigger**
Mispredicted Branch	`E_icode = IJXX & !e_Cnd`

Control for Misprediction



`# demo-j.ys`

	1	2	3	4	5	6	7	8	9	10
`0x000: xorq %rax,%rax`	F	D	E	M	W					
`0x002: jne target # Not taken`		F	D	E	M	W				
`0x016: irmovq $2,%rdx # Target`			F	D						
bubble					E	M	W			
`0x020: irmovq $3,%rbx # Target+1`				F						
bubble					D	E	M	W		
`0x00b: irmovq $1,%rax # Fall through`					F	D	E	M	W	
`0x015: halt`						F	D	E	M	W

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

Return Example

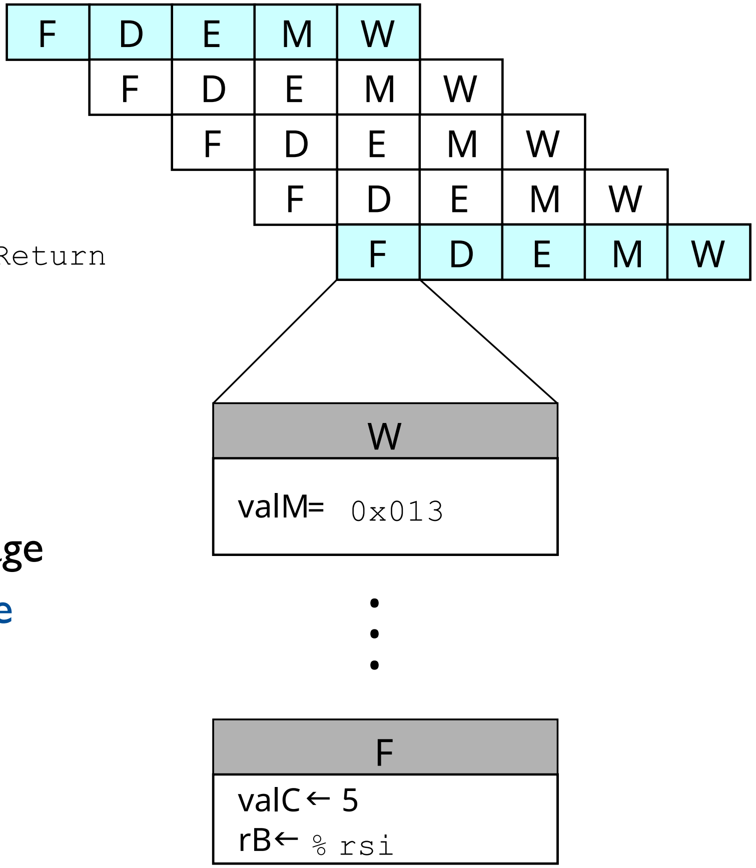
demo-retb.ys

```
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
```

- Previously executed three additional instructions

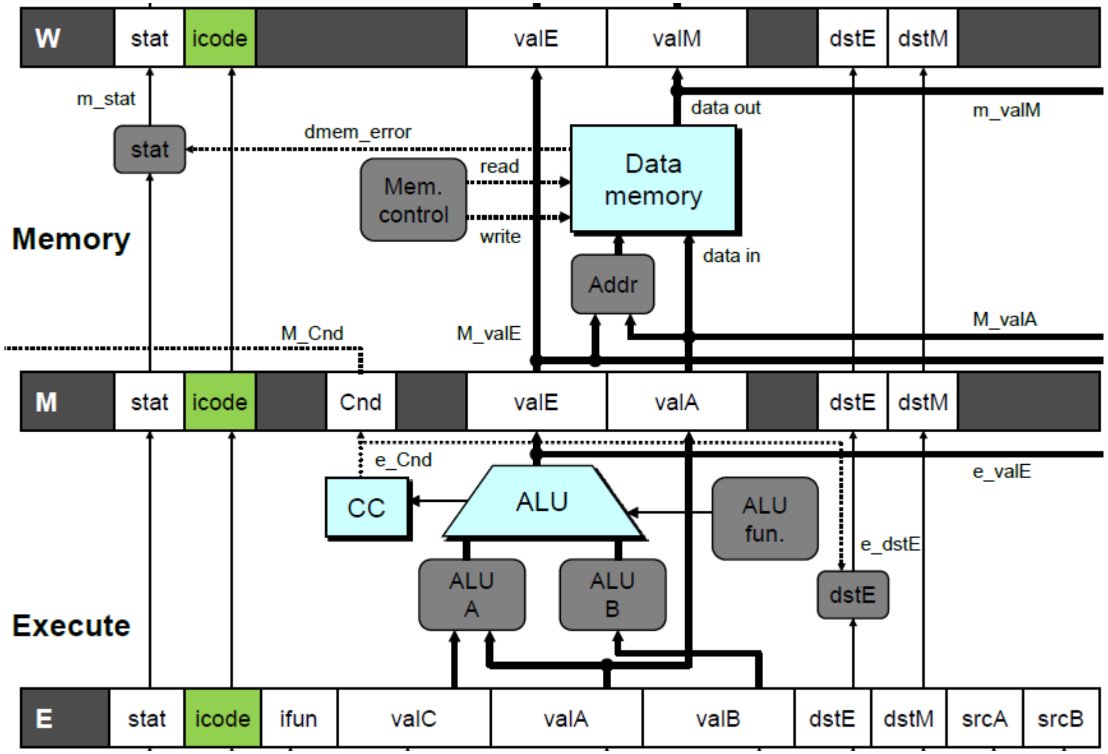
Correct Return Example

```
# 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 `ret`	`IRET in { D_icode, E_icode, M_icode }`

Control for Return

`# demo-retb`

`0x026:    ret`	F	D	E	M	W				
`bubble`		F	D	E	M	W			
`bubble`			F	D	E	M	W		
`bubble`				F	D	E	M	W	
`0x014:   irmovq $5,%rsi # Return`					F	D	E	M	W

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

Special Control Cases

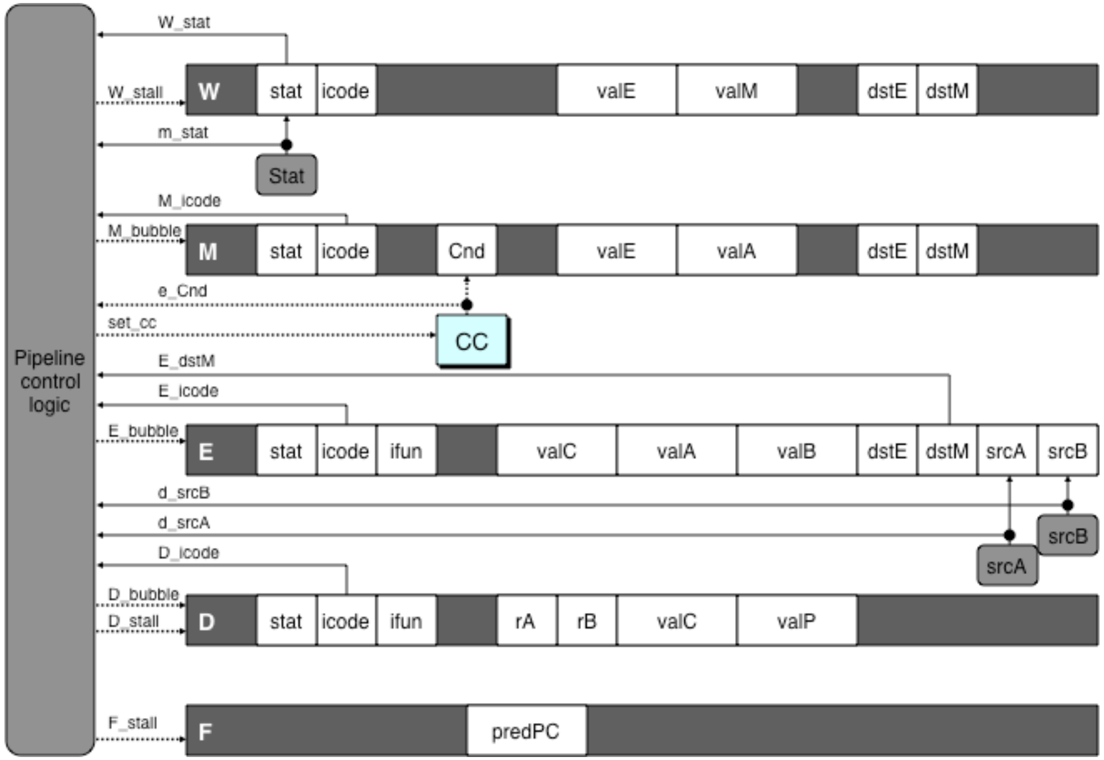
☐ Detection

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

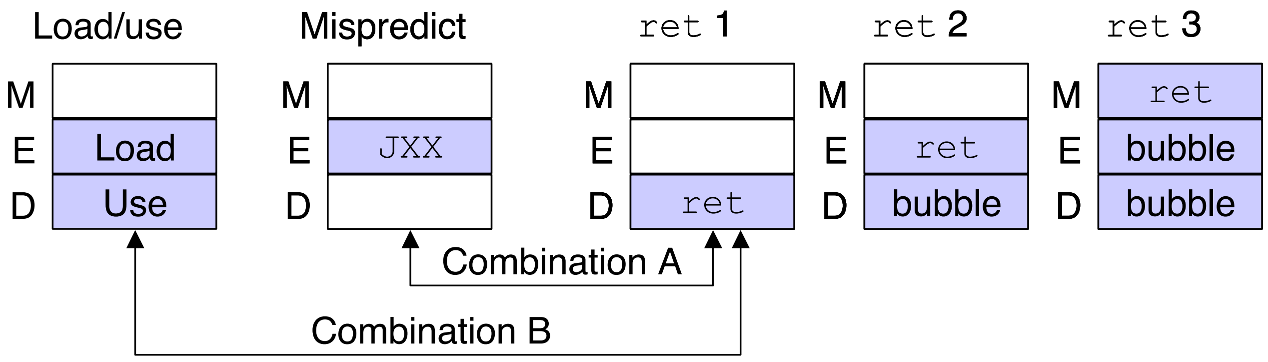


- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle

Initial 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

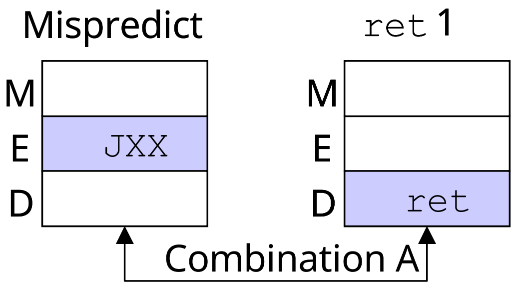
Combination A

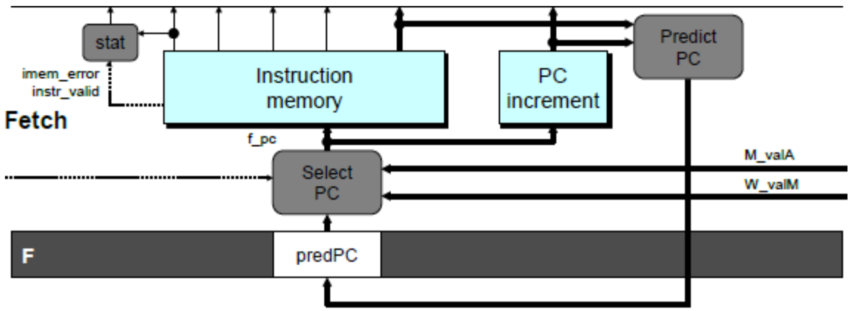
- Not-taken branch
- `ret` instruction at branch target

Combination B

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

Control Combination A

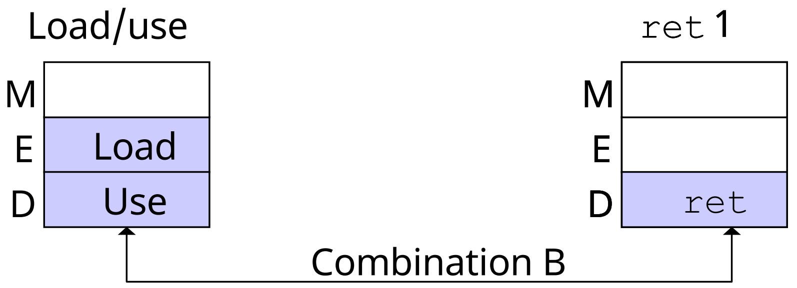




Condition	**F**	**D**	**E**	**M**	**W**
Processing ret	**stall**	**bubble**	**normal**	**normal**	**normal**
Mispredicted Branch	**normal**	**bubble**	**bubble**	**normal**	**normal**
Combination	***stall***	***bubble***	***bubble***	***normal***	***normal***

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

Handling Control Combination B

```
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 combination

Today

- ☐ Overview
- ☐ Y86-64 Instruction Set Architecture
- ☐ Logic Design
- ☐ Sequential Implementation
- ☐ Pipelined Implementation: Principle
- ☐ Pipelined Implementation: Implementation
- ☐ **Wrap-Up**

Overview

- ☐ **Wrap-Up of PIPE Design**
  - ◆ Exceptional conditions
  - ◆ Performance analysis
  - ◆ Fetch stage design
- ☐ **Modern High-Performance Processors**
  - ◆ Out-of-order execution

Exceptions

- Conditions under which processor cannot continue normal operation

Causes

- Halt instruction (Current)
- Bad address for instruction or data (Previous)
- Invalid instruction (Previous)

Typical Desired Action

- Complete some instructions
  - Either current or previous (depends on exception type)
- Discard others
- Call exception handler
  - Like an unexpected procedure call

Our Implementation

- Halt when instruction causes exception

Exception Examples

☐ Detect in Fetch Stage

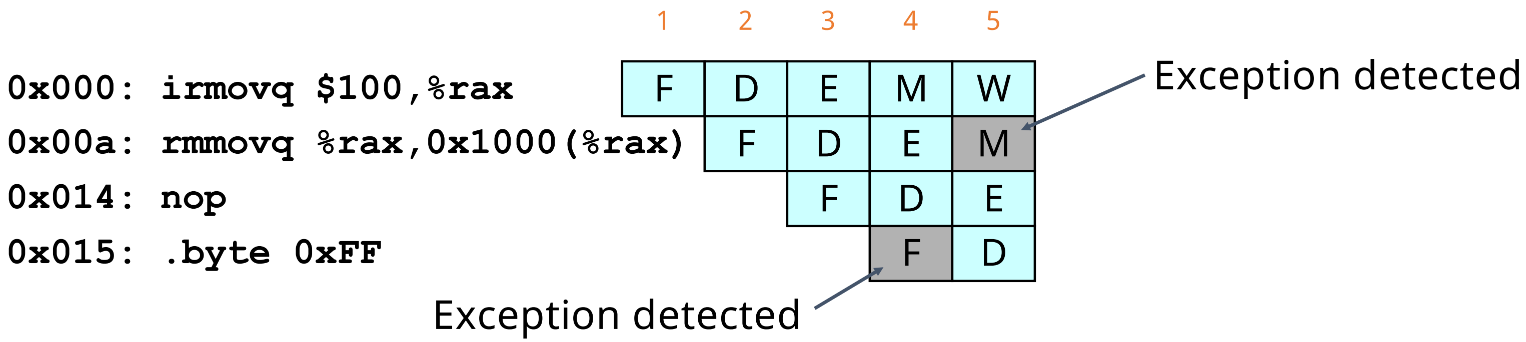
```
jmp $-1                         # Invalid jump target

.byte 0xFF                      # Invalid instruction code

halt                            # Halt instruction
```

☐ Detect in Memory Stage

```
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # invalid address
```

Exceptions in Pipeline Processor #1

```
# demo-exc1.ys
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # Invalid address
nop
.byte 0xFF                # Invalid instruction code
```



	1	2	3	4	5
`0x000: irmovq $100,%rax`	F	D	E	M	W
`0x00a: rmmovq %rax,0x1000(%rax)`		F	D	E	M
`0x014: nop`			F	D	E
`0x015: .byte 0xFF`				F	D

Exception detected

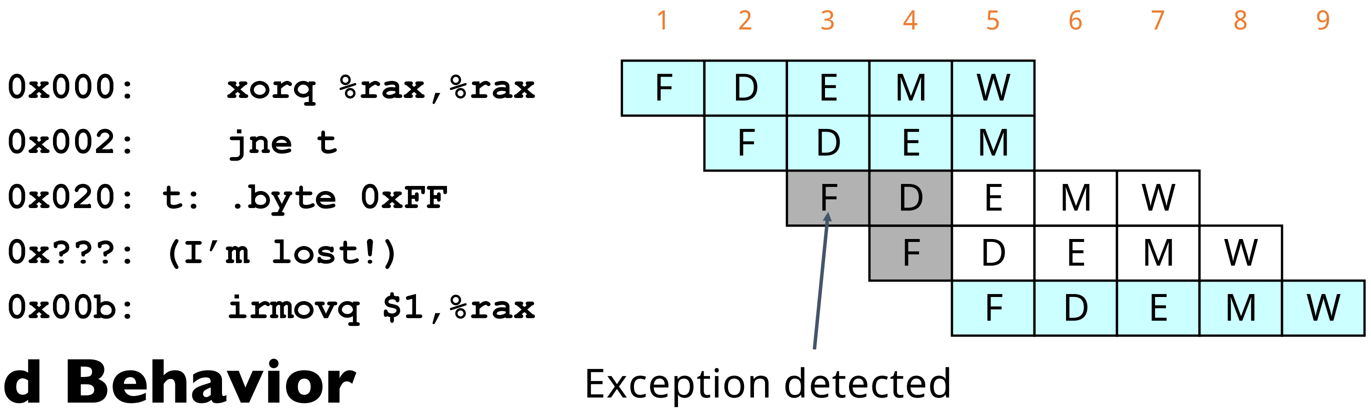
Exception detected

Desired Behavior

- `rmmovq` should cause exception
- Following instructions should have no effect on processor state

Exceptions in Pipeline Processor #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
```



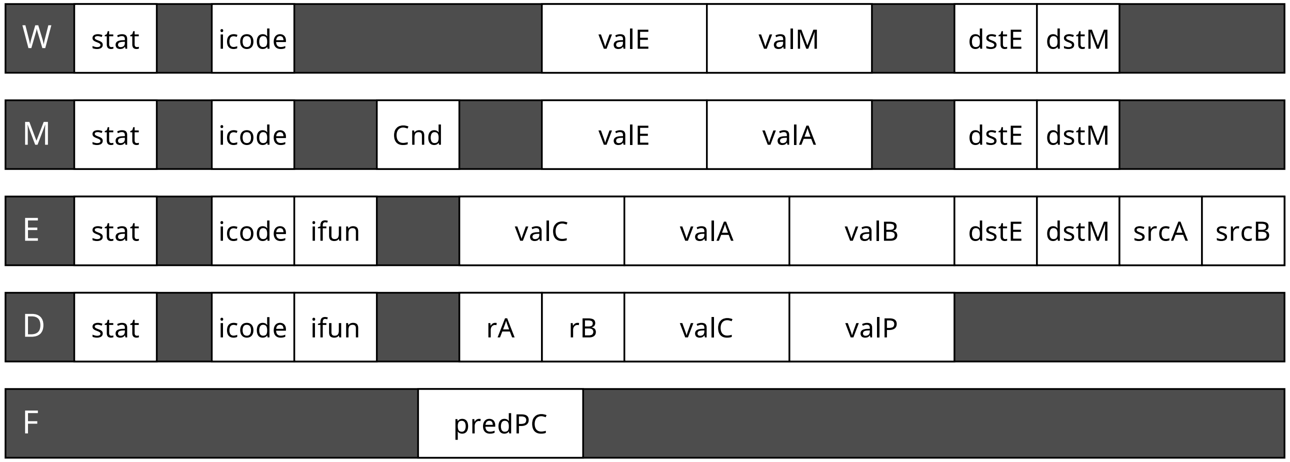
	1	2	3	4	5	6	7	8	9
`0x000:    xorq %rax,%rax`	F	D	E	M	W				
`0x002:    jne t`		F	D	E	M				
`0x020: t: .byte 0xFF`			F	D	E	M	W		
`0x???: (I’m lost!)`				F	D	E	M	W	
`0x00b:    irmovq $1,%rax`					F	D	E	M	W

Exception detected

☐ Desired Behavior

- No exception should occur

Maintaining Exception Ordering



- Add status field to pipeline registers
- Fetch stage sets to either “AOK,” “ADR” (when bad fetch address), “HLT” (halt instruction) or “INS” (illegal instruction)
- Decode & execute pass values through
- Memory either passes through or sets to “ADR”
- Exception triggered only when instruction hits write back

Exception Handling Logic

☐ Fetch Stage

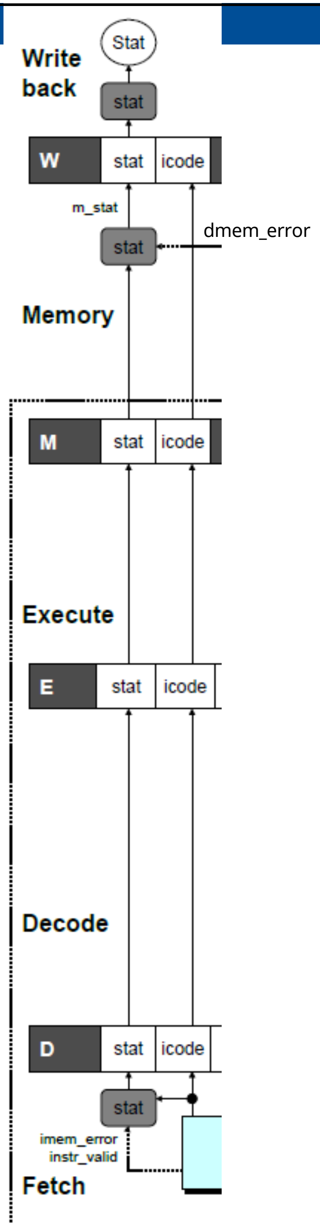
```
# Determine status code for fetched instruction
int f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];
```

☐ Memory Stage

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

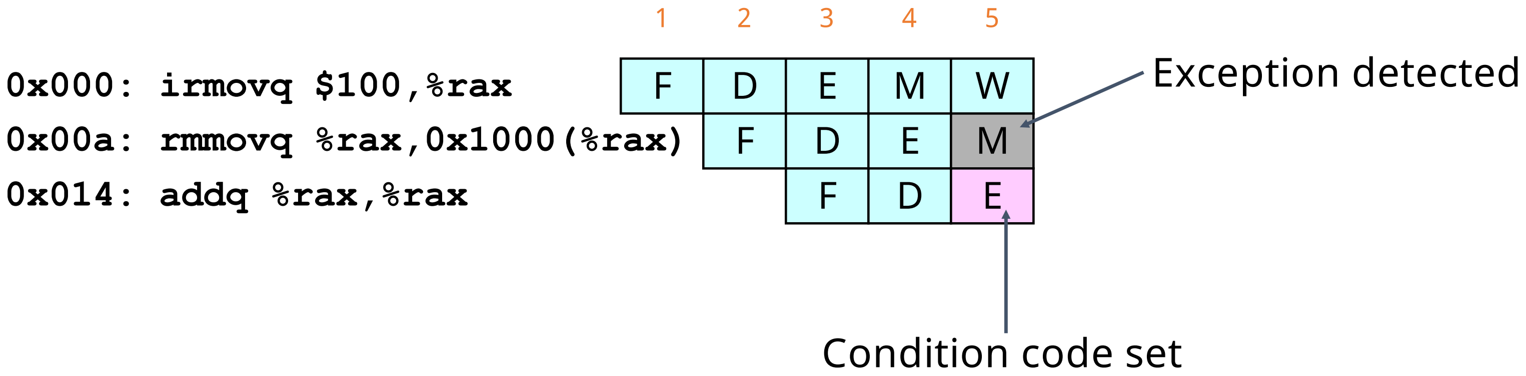
☐ Writeback Stage

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



Side Effects in Pipeline Processor

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



	1	2	3	4	5
`0x000: irmovq $100,%rax`	F	D	E	M	W
`0x00a: rmmovq %rax,0x1000(%rax)`		F	D	E	M
`0x014: addq %rax,%rax`			F	D	E

Exception detected

Condition code set

☐ Desired Behavior

- `rmmovq` should cause exception
- No following instruction should have any effect

Avoiding Side Effects

☐ Presence of Exception Should Disable State Update

- Invalid instructions are converted to pipeline bubbles
    - Except have stat indicating exception status
- Data memory will not write to invalid address
- Prevent invalid update of condition codes
    - Detect exception in memory stage
    - Disable condition code setting in execute
    - Must happen in same clock cycle
- Handling exception in final stages
    - When detect exception in memory stage
        - Start injecting bubbles into memory stage on next cycle
    - When detect exception in write-back stage
        - Stall excepting instruction
- Included in HCL code

Control Logic for State Changes

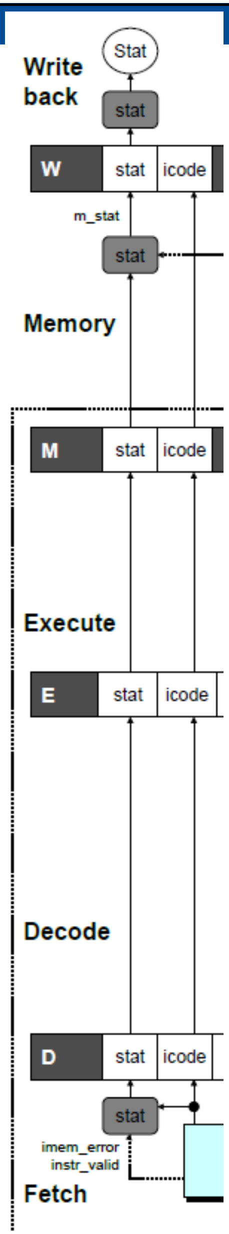
☐ Setting Condition Codes

```
# 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 };
```

☐ Stage Control

- Also controls updating of memory

```
# 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 };
```



Rest of Real-Life Exception Handling

- **Call Exception Handler**
  - Push PC onto stack
    - Either PC of faulting instruction or of next instruction
    - Usually pass through pipeline along with exception status
  - Jump to handler address
    - Usually fixed address
    - Defined as part of ISA
- **Implementation**
  - Haven't tried it yet!

Performance Metrics

- **Clock rate**
  - Measured in Gigahertz
  - Function of stage partitioning and circuit design
    - Keep amount of work per stage small
- **Rate at which instructions executed**
  - CPI: cycles per instruction
  - On average, how many clock cycles does each instruction require?
  - Function of pipeline design and benchmark programs
    - E.g., how frequently are branches mispredicted?

CPI for PIPE

☐ CPI ≈ 1.0

- Fetch instruction each clock cycle
- Effectively process new instruction almost every cycle
    - Although each individual instruction has latency of 5 cycles

☐ CPI > 1.0

- Sometimes must stall or cancel branches

☐ Computing CPI

- C clock cycles
- I instructions executed to completion
- B bubbles injected (C = I + B)

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

- Factor B/I represents average penalty due to bubbles

CPI for PIPE (Cont.)

B/I = LP + MP + RP

Typical Values

- LP: Penalty due to load/use hazard stalling
  - Fraction of instructions that are loads 0.25
  - Fraction of load instructions requiring stall 0.20
  - Number of bubbles injected each time 1

  ⇒ LP = 0.25 * 0.20 * 1 = 0.05
- MP: Penalty due to mispredicted branches
  - Fraction of instructions that are cond. jumps 0.20
  - Fraction of cond. jumps mispredicted 0.40
  - Number of bubbles injected each time 2

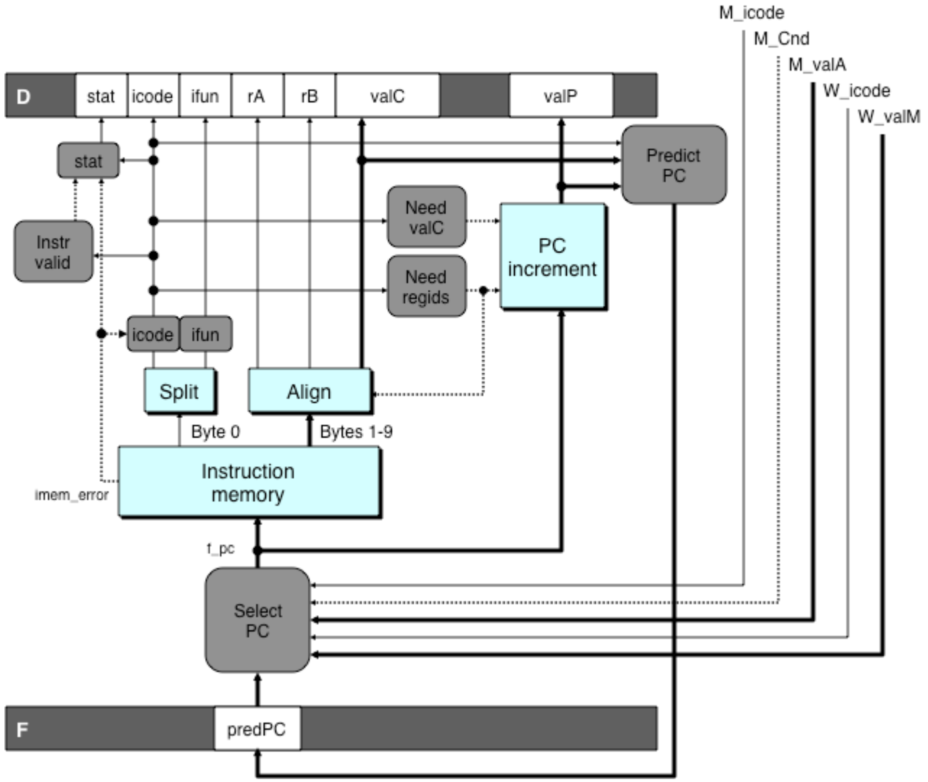
  ⇒ MP = 0.20 * 0.40 * 2 = 0.16
- RP: Penalty due to `ret` instructions
  - Fraction of instructions that are returns 0.02
  - Number of bubbles injected each time 3

  ⇒ RP = 0.02 * 3 = 0.06
- Net effect of penalties 0.05 + 0.16 + 0.06 = 0.27

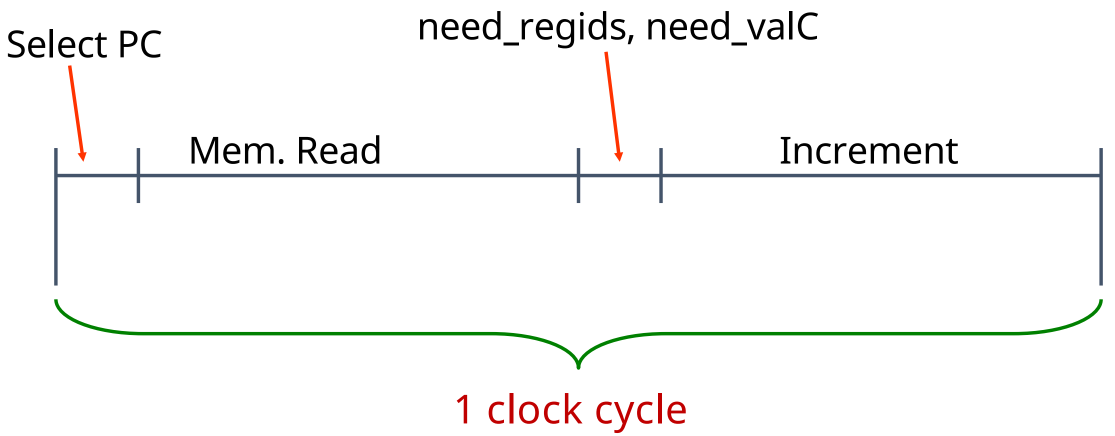
  ⇒ CPI = 1.27 (Not bad!)

Fetch Logic Revisited

- ☐ **During Fetch Cycle**
  1. Select PC
  2. Read bytes from instruction memory
  3. Examine icode to determine instruction length
  4. Increment PC
- ☐ **Timing**
  - ◆ Steps 2 & 4 require significant amount of time

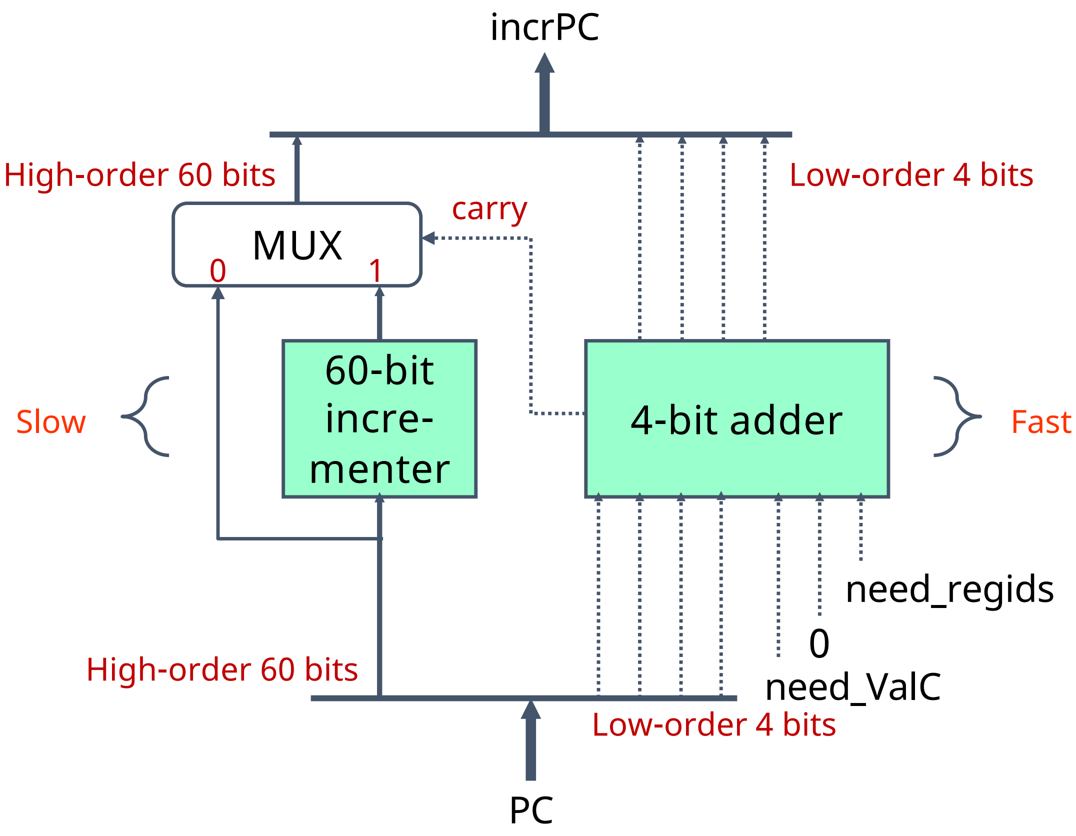


Standard Fetch Timing

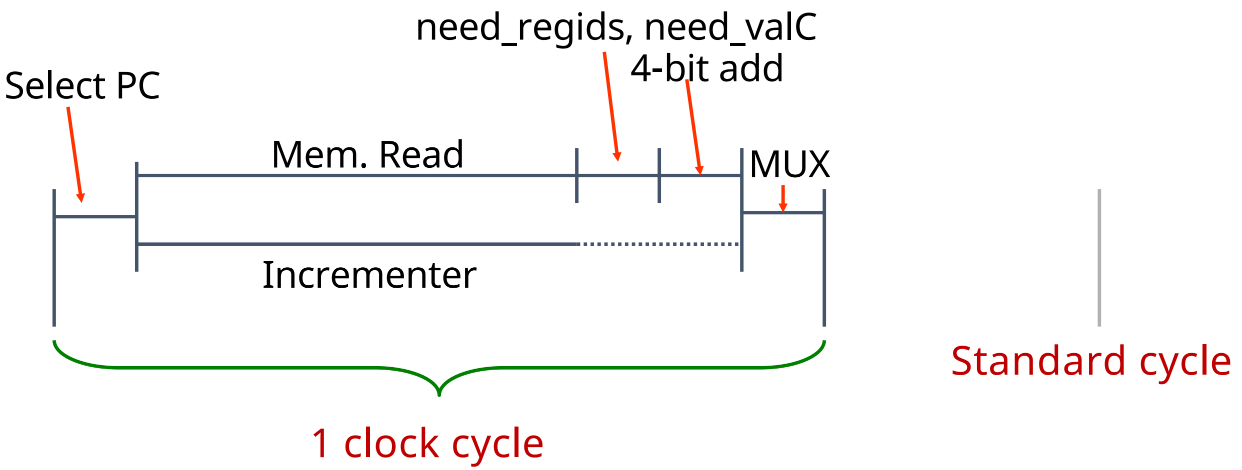


- Must Perform Everything in Sequence
- Can't compute incremented PC until know how much to increment it by

A Fast PC Increment Circuit



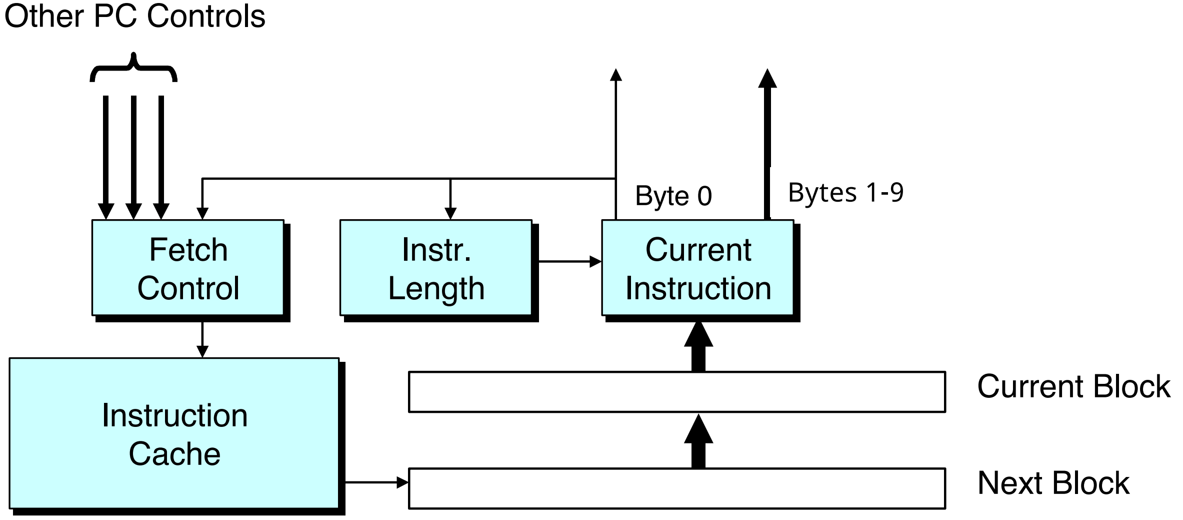
Modified Fetch Timing



◻ 60-Bit Incrementer

- Acts as soon as PC selected
- Output not needed until final MUX
- Works in parallel with memory read

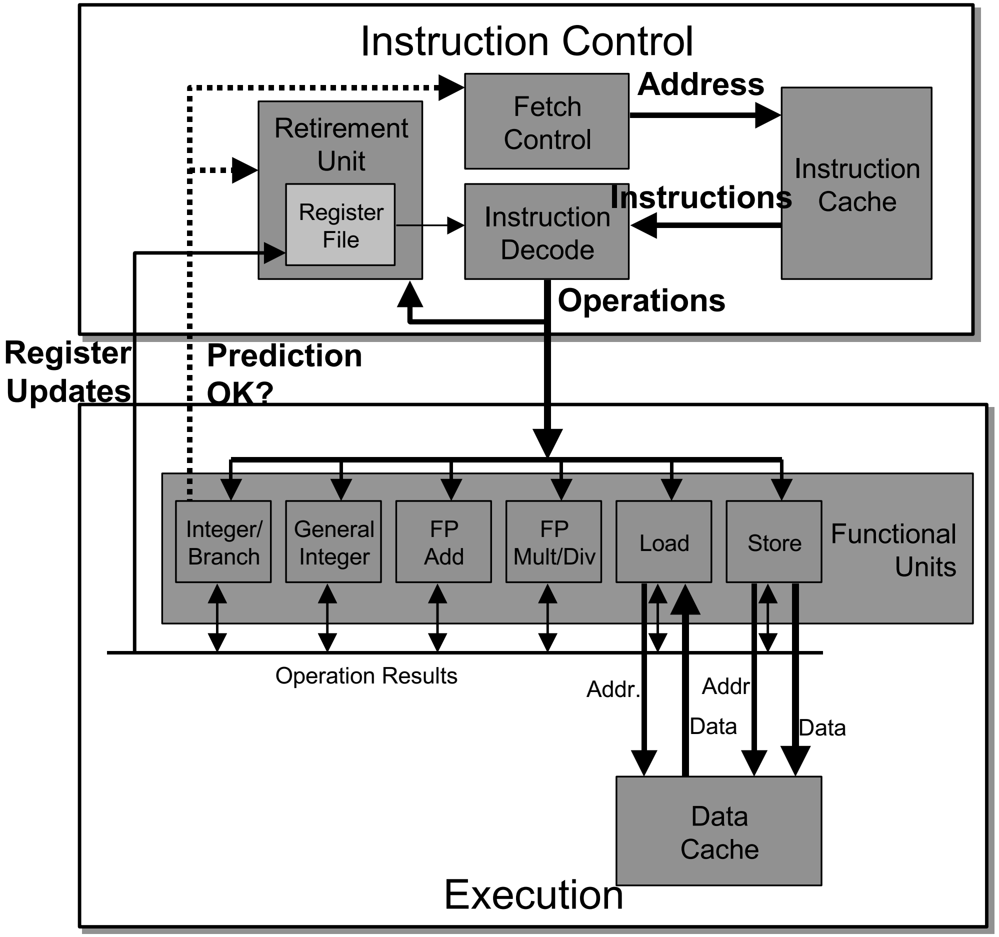
More Realistic Fetch Logic



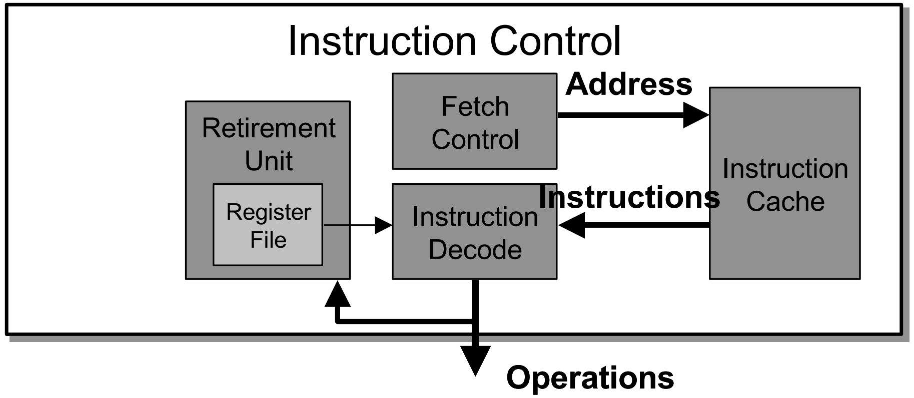
Fetch Box

- Integrated into instruction cache
- Fetches entire cache block (16 or 32 bytes)
- Selects current instruction from current block
- Works ahead to fetch next block
    - As reaches end of current block
    - At branch target

Modern CPU Design

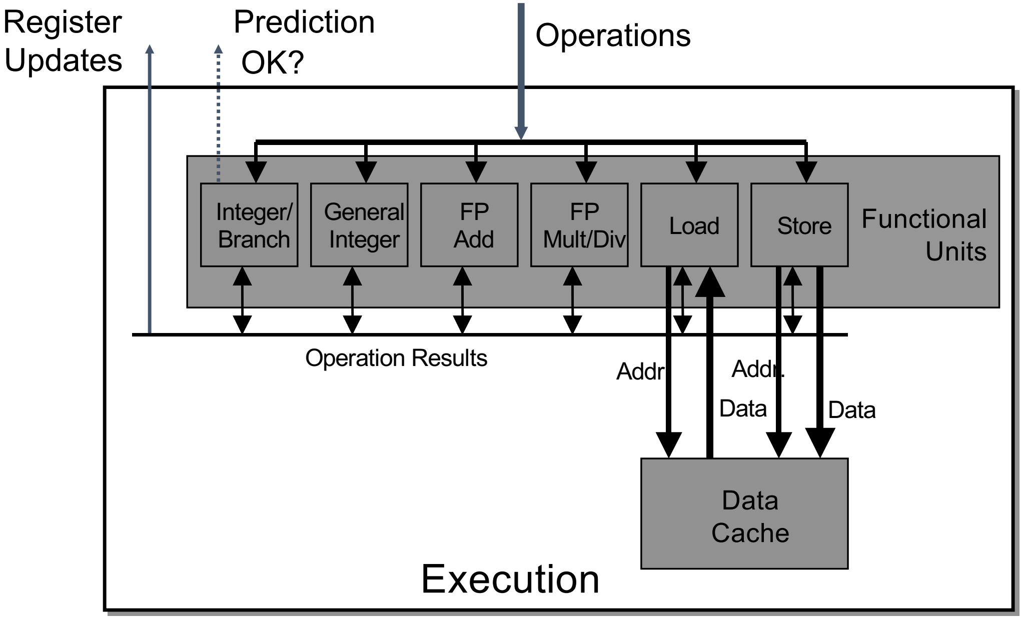


Instruction Control



- **Grabs Instruction Bytes From Memory**
  - Based on Current PC + Predicted Targets for Predicted Branches
  - Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target
- **Translates Instructions Into *Operations***
  - Primitive steps required to perform instruction
  - Typical instruction requires 1–3 operations
- **Converts Register References Into *Tags***
  - Abstract identifier linking destination of one operation with sources of later operations

Execution Unit



- Multiple functional units
  - Each can operate in independently
- Operations performed as soon as operands available
  - Not necessarily in program order
  - Within limits of functional units
- Control logic
  - Ensures behavior equivalent to sequential program execution

CPU Capabilities of Intel Haswell

Multiple Instructions Can Execute in Parallel

- 2 load
- 1 store
- 4 integer
- 2 FP multiply
- 1 FP add / divide

Some Instructions Take > 1 Cycle, but Can be Pipelined

Instruction	Latency	Cycles/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 Operation

- **Translates instructions dynamically into “Uops”**
  - ~118 bits wide
  - Holds operation, two sources, and destination
- **Executes Uops with “Out of Order” engine**
  - Uop executed when
    - Operands available
    - Functional unit available
  - Execution controlled by “Reservation Stations”
    - Keeps track of data dependencies between uops
    - Allocates resources

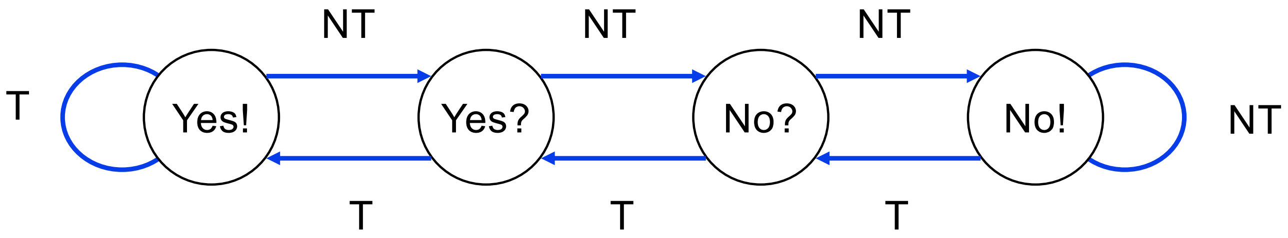
High-Performance Branch Prediction

- **Critical to Performance**
  - Typically 11–15 cycle penalty for misprediction
- **Branch Target Buffer**
  - 512 entries
  - 4 bits of history
  - Adaptive algorithm
    - Can recognize repeated patterns, e.g., alternating taken–not taken
- **Handling BTB misses**
  - Detect in ~cycle 6
  - Predict taken for negative offset, not taken for positive
    - Loops vs. conditionals

Example Branch Prediction

Branch History

- Encode information about prior history of branch instructions
- Predict whether or not branch will be taken



State Machine

- Each time branch taken, transition to right
- When not taken, transition to left
- Predict branch taken when in state Yes! or Yes?

Processor Summary

☐ Design Technique

- Create uniform framework for all instructions
    - Want to share hardware among instructions
- Connect standard logic blocks with bits of control logic

☐ Operation

- State held in memories and clocked registers
- Computation done by combinational logic
- Clocking of registers/memories sufficient to control overall behavior

☐ Enhancing Performance

- Pipelining increases throughput and improves resource utilization
- Must make sure to maintain ISA behavior