Processor Architecture III: PIPE: Pipelined Implementation

Introduction to Computer Systems
11th Lecture, Oct. 23, 2023

Instructors:

Class 1: Chen Xiangqun, Liu Xianhua

Class 2: Guan Xuetao

Class 3: Lu Junlin

Overview

- **General Principles of Pipelining**
  - Goal
  - Difficulties

- **Creating a Pipelined Y86-64 Processor**
  - Rearranging SEQ
  - Inserting pipeline registers
  - Problems with data and control hazards
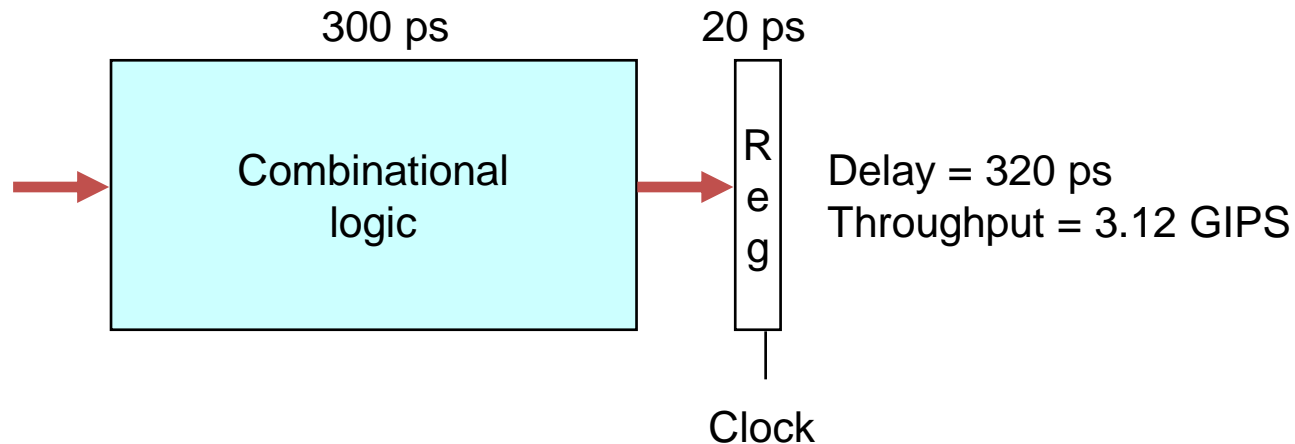
Real-World Pipelines: Car Washes

Sequential

Parallel

Pipelined

- **Idea**
  - **Divide process into independent stages**
  - **Move objects through stages in sequence**
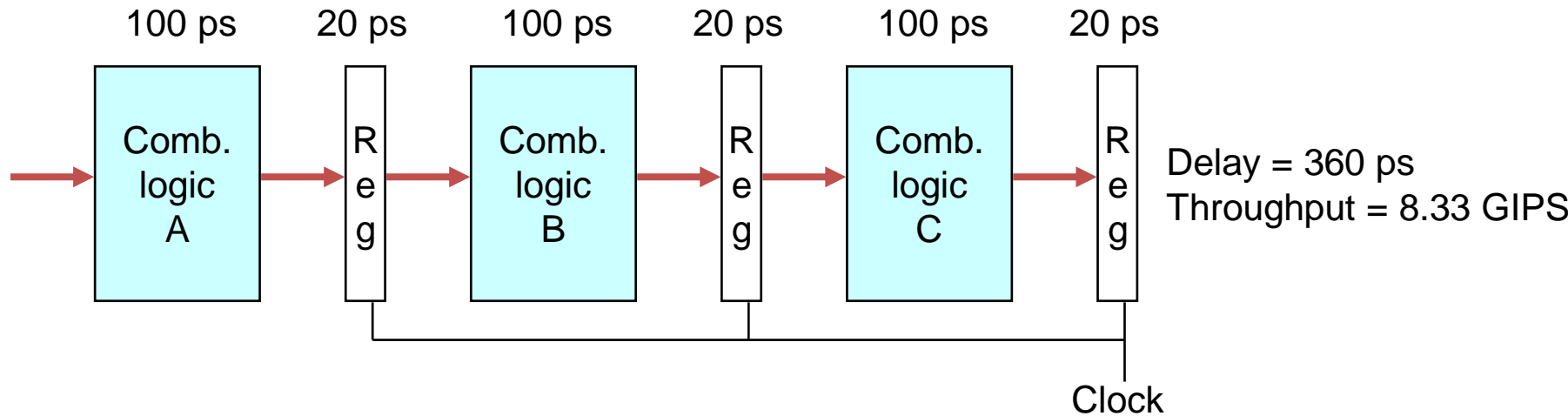 - **At any given times, multiple objects being processed**

Computational Example

300 ps   20 ps

Combinational logic

R e g

Clock

Delay = 320 ps
Throughput = 3.12 GIPS

- **System**
  - **Computation requires total of 300 picoseconds**
  - **Additional 20 picoseconds to save result in register**
  - **Must have clock cycle of at least 320 ps**

3-Way Pipelined Version

100 ps	20 ps	100 ps	20 ps	100 ps	20 ps

Comb. logic A → R e g → Comb. logic B → R e g → Comb. logic C → R e g →

Delay = 360 ps
Throughput = 8.33 GIPS

Clock

- ■ **System**
  - ■ **Divide combinational logic into 3 blocks of 100 ps each**
  - ■ **Can begin new operation as soon as previous one passes through stage A.**
    - ● **Begin new operation every 120 ps**
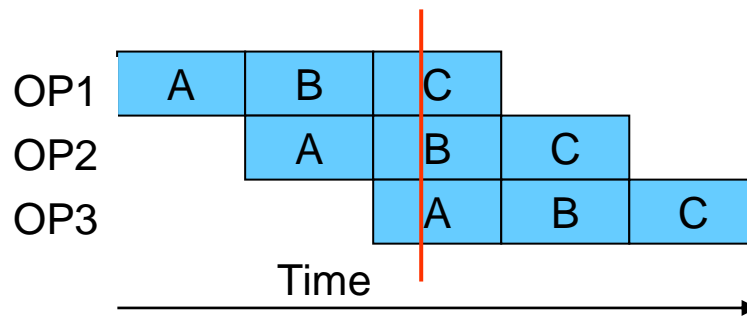 - ■ **Overall latency increases**
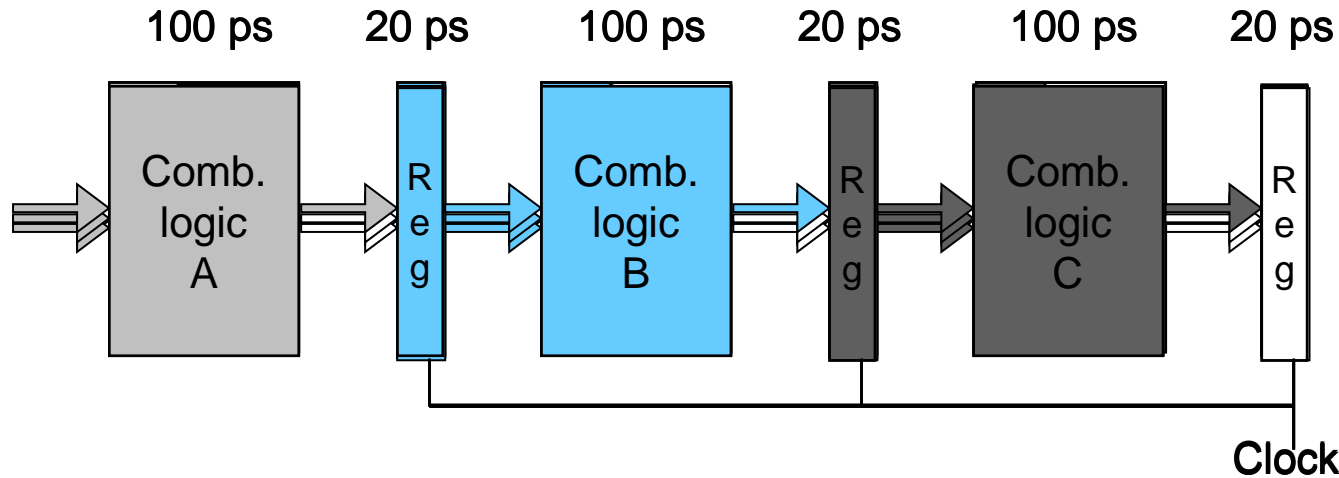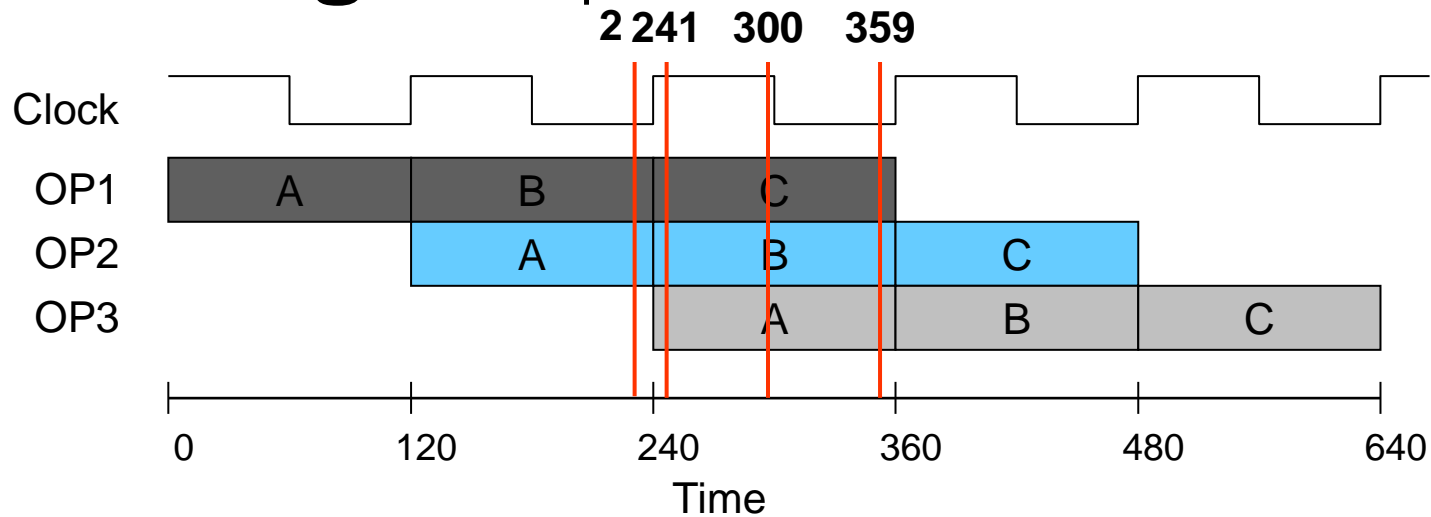   - ● **360 ps from start to finish**

Pipeline Diagrams

- ## Unpipelined

OP1
OP2
OP3
Time

  - **Cannot start new operation until previous one completes**

- ## 3-Way Pipelined

OP1   A   B   C
OP2   A   B   C
OP3   A   B   C
Time

  - **Up to 3 operations in process simultaneously**

Operating a Pipeline

Limitations: Nonuniform Delays

50 ps	20 ps	150 ps	20 ps	100 ps	20 ps
Comb. logic A	R e g	Comb. logic B	R e g	Comb. logic C	R e g

Delay = 510 ps
Throughput = 5.88 GIPS

Clock

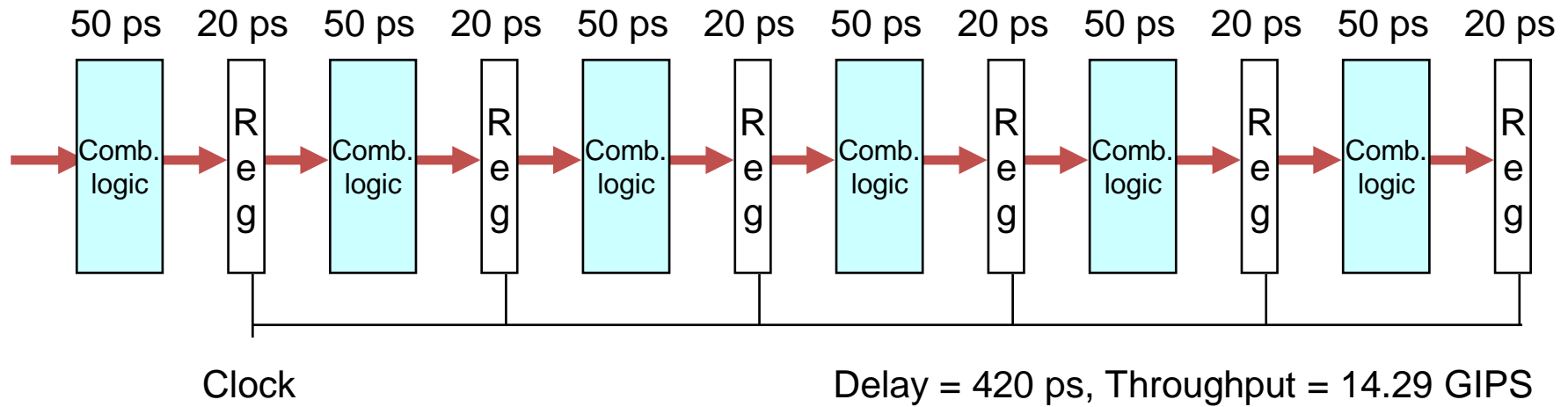|			
OP1	A	B	C
OP2		A	B	C
OP3			A	B	C
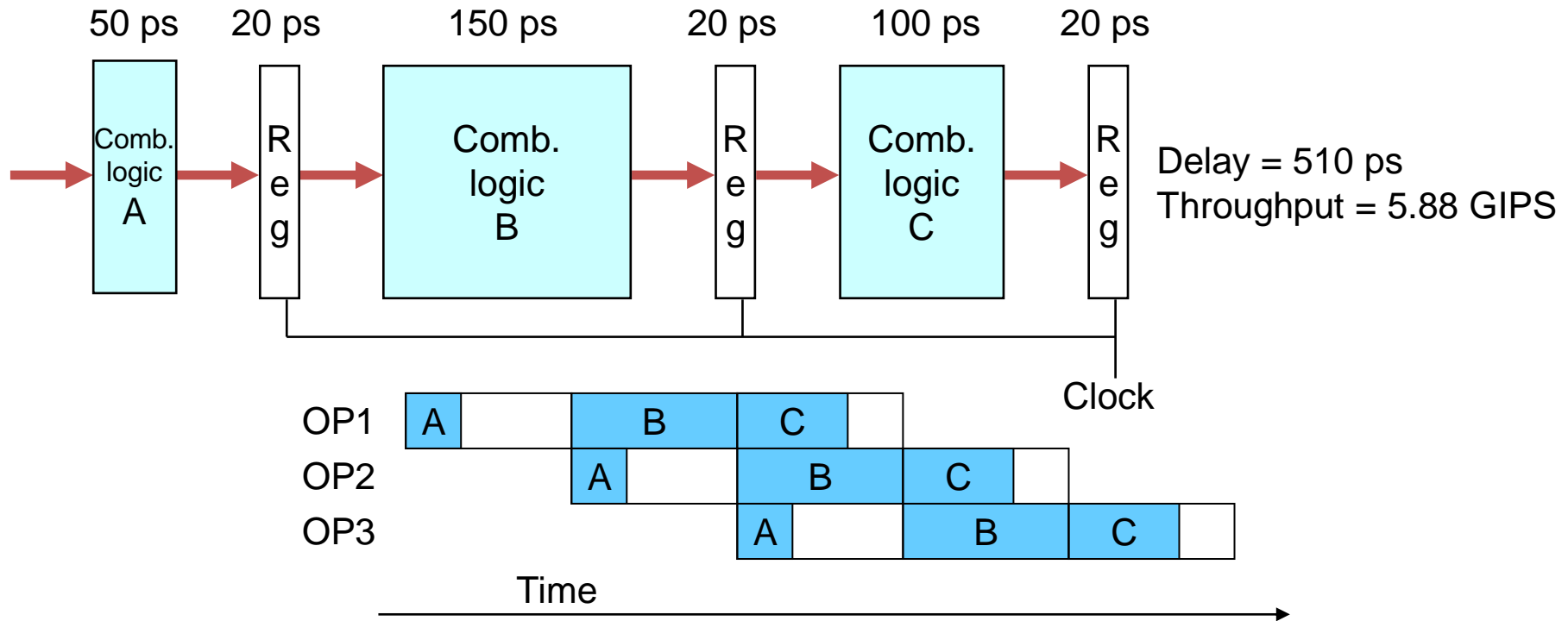
Time

- **Throughput limited by slowest stage**
- **Other stages sit idle for much of the time**
- **Challenging to partition system into balanced stages**

Limitations: Register Overhead

50 ps 20 ps 50 ps 20 ps 50 ps 20 ps 50 ps 20 ps 50 ps 20 ps 50 ps 20 ps

Clock   Delay = 420 ps, Throughput = 14.29 GIPS

- **As try to deepen pipeline, overhead of loading registers becomes more significant**

- **Percentage of clock cycle spent loading register:**
    - **1-stage pipeline: 6.25%**
    - **3-stage pipeline: 16.67%**
    - **6-stage pipeline: 28.57%**

- **High speeds of modern processor designs obtained through very deep pipelining**
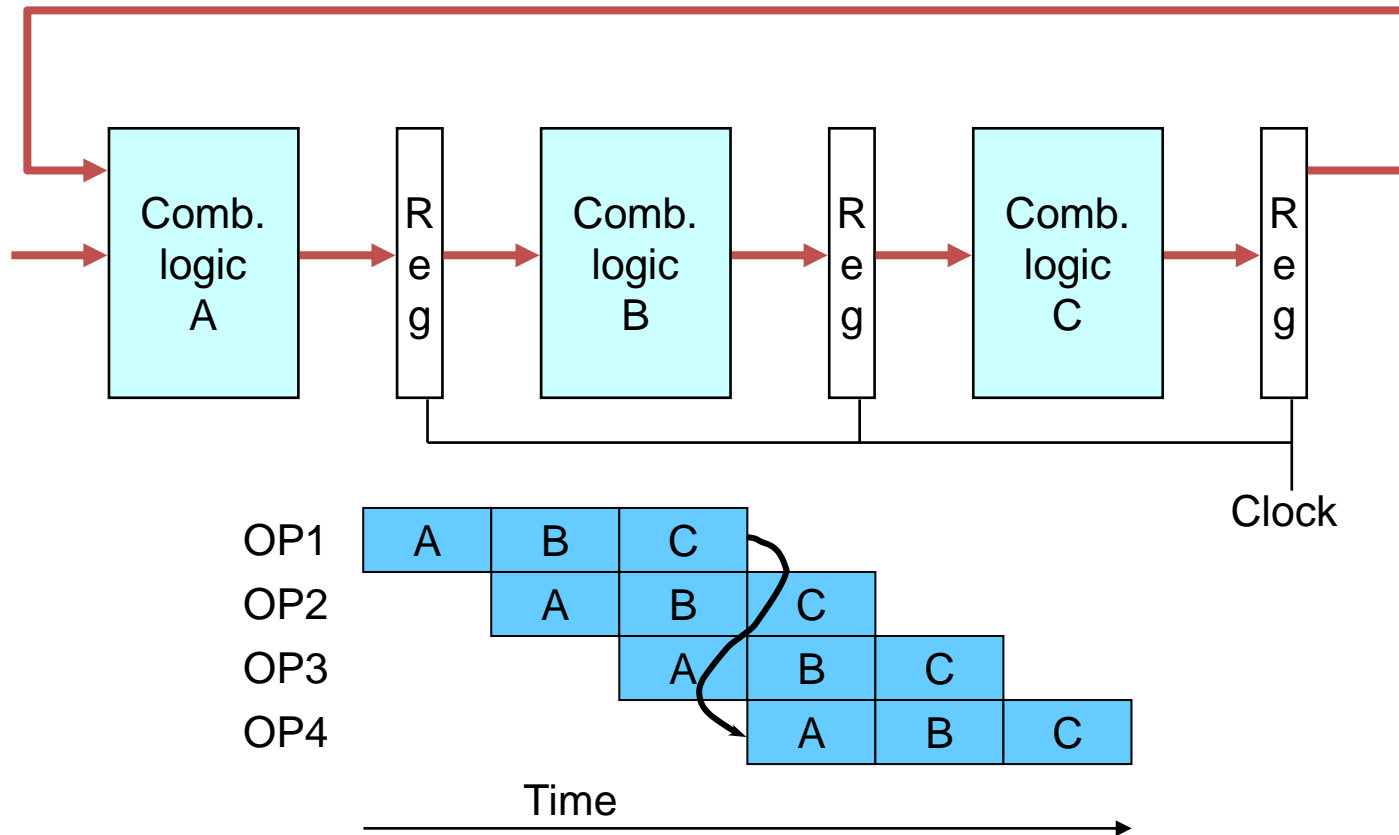
Data Dependencies
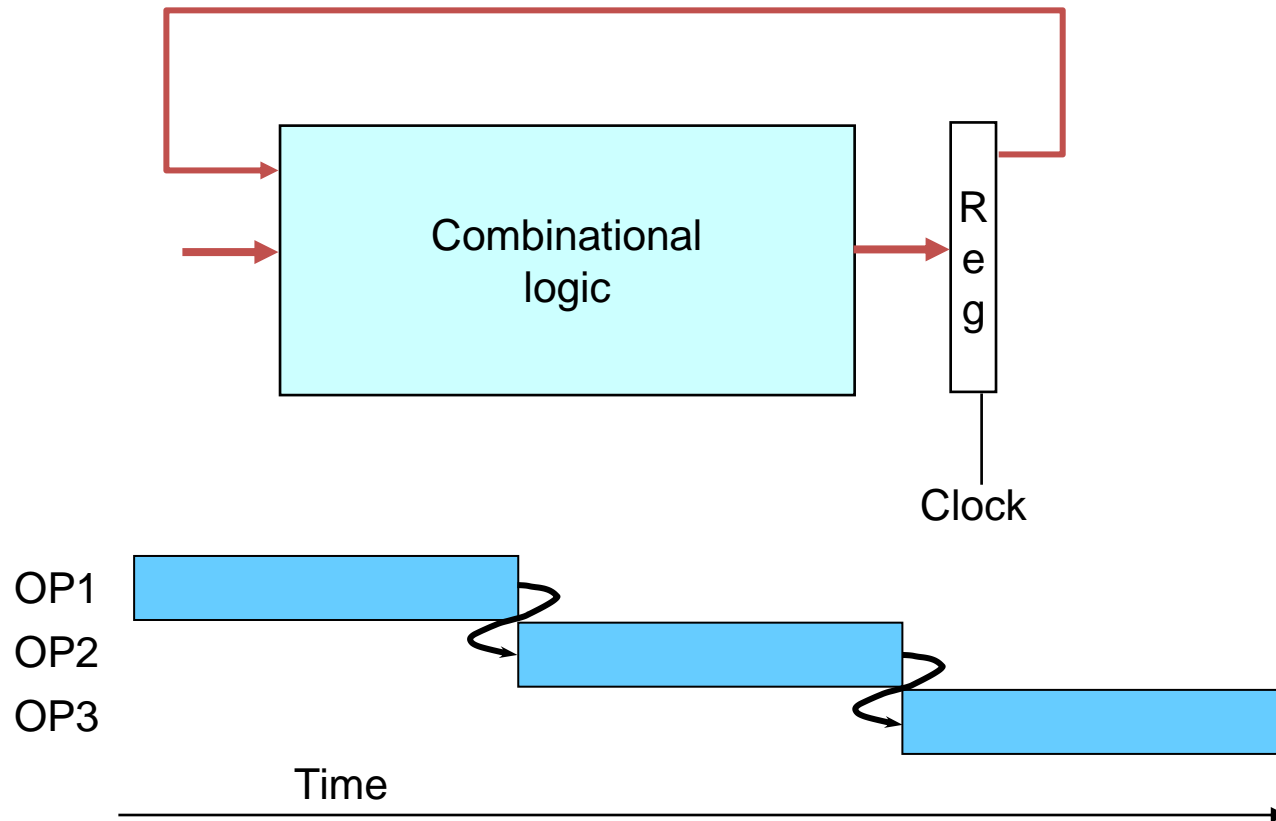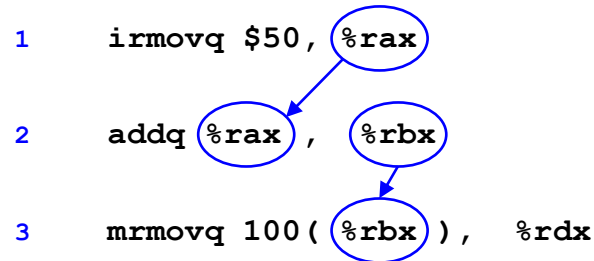

- ## System
  - **Each operation depends on result from preceding one**

Data Hazards

- **Result does not feed back around in time for next operation**
- **Pipelining has changed behavior of system**

Data Dependencies in Processors

```
1      irmovq $50, %rax

2      addq %rax, %rbx

3      mrmovq 100(%rbx), %rdx
```

- **Result from one instruction used as operand for another**
  - **Read-after-write (RAW) dependency**
- **Very common in actual programs**
- **Must make sure our pipeline handles these properly**
  - **Get correct results**
  - **Minimize performance impact**

SEQ Hardware

- **Stages occur in sequence**
- **One operation in process at a time**
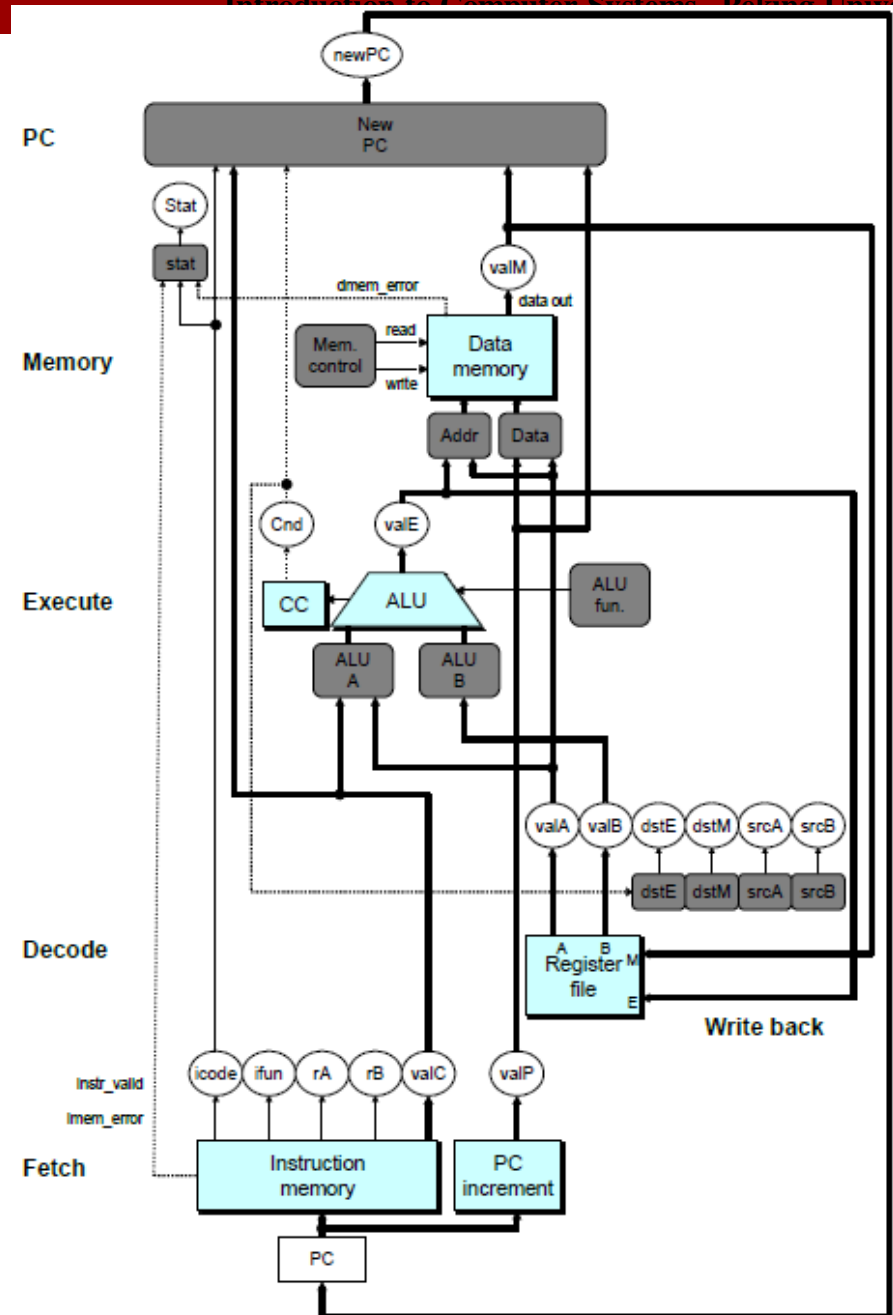
SEQ+ Hardware

- **Still sequential implementation**
- **Reorder PC stage to put at beginning**

- **PC Stage**
  - **Task is to select PC for current instruction**
  - **Based on results computed by previous instruction**

- **Processor State**
  - **PC is no longer stored in register**
  - **But, can determine PC based on other stored information**
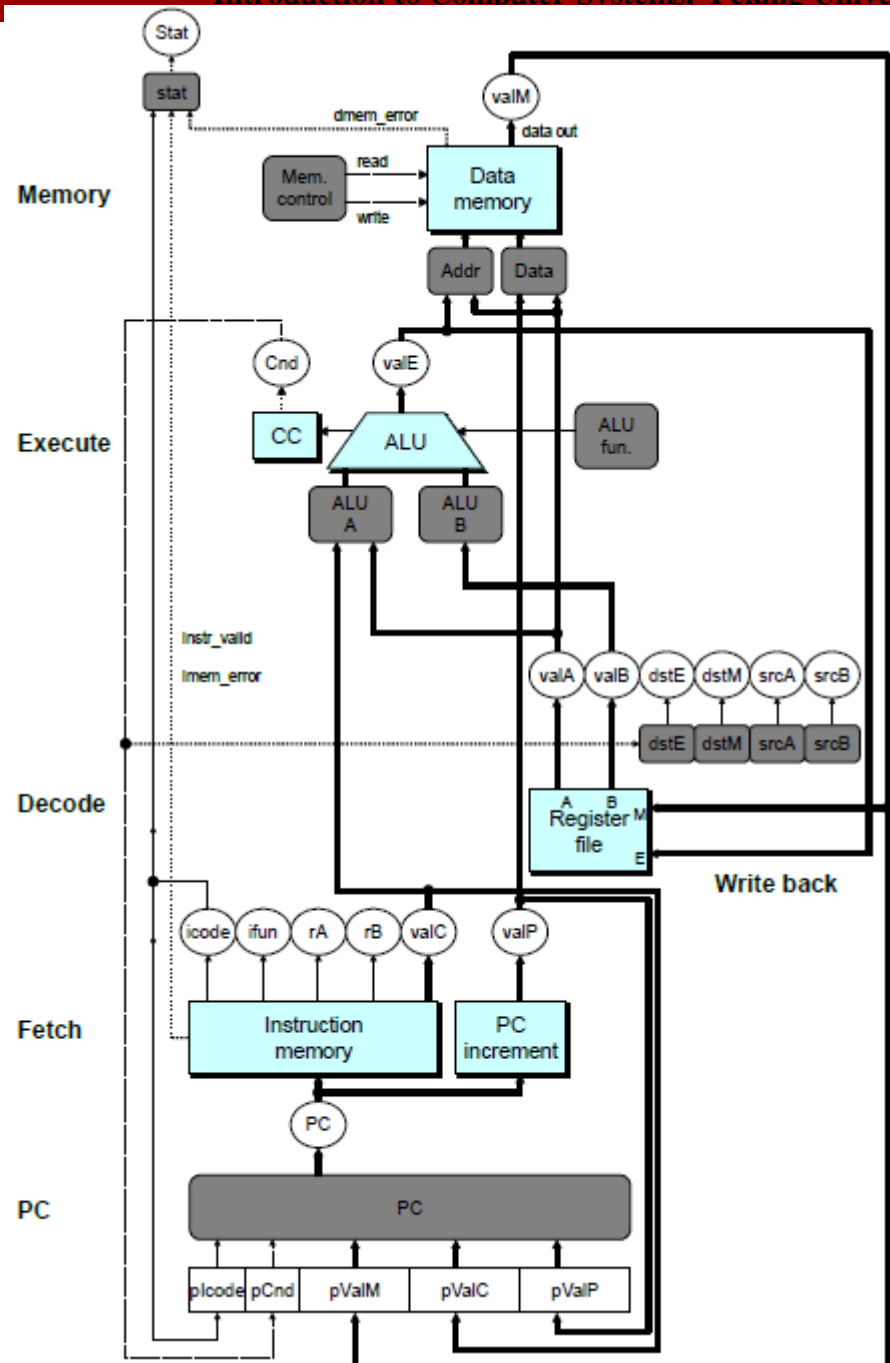
Adding Pipeline Registers
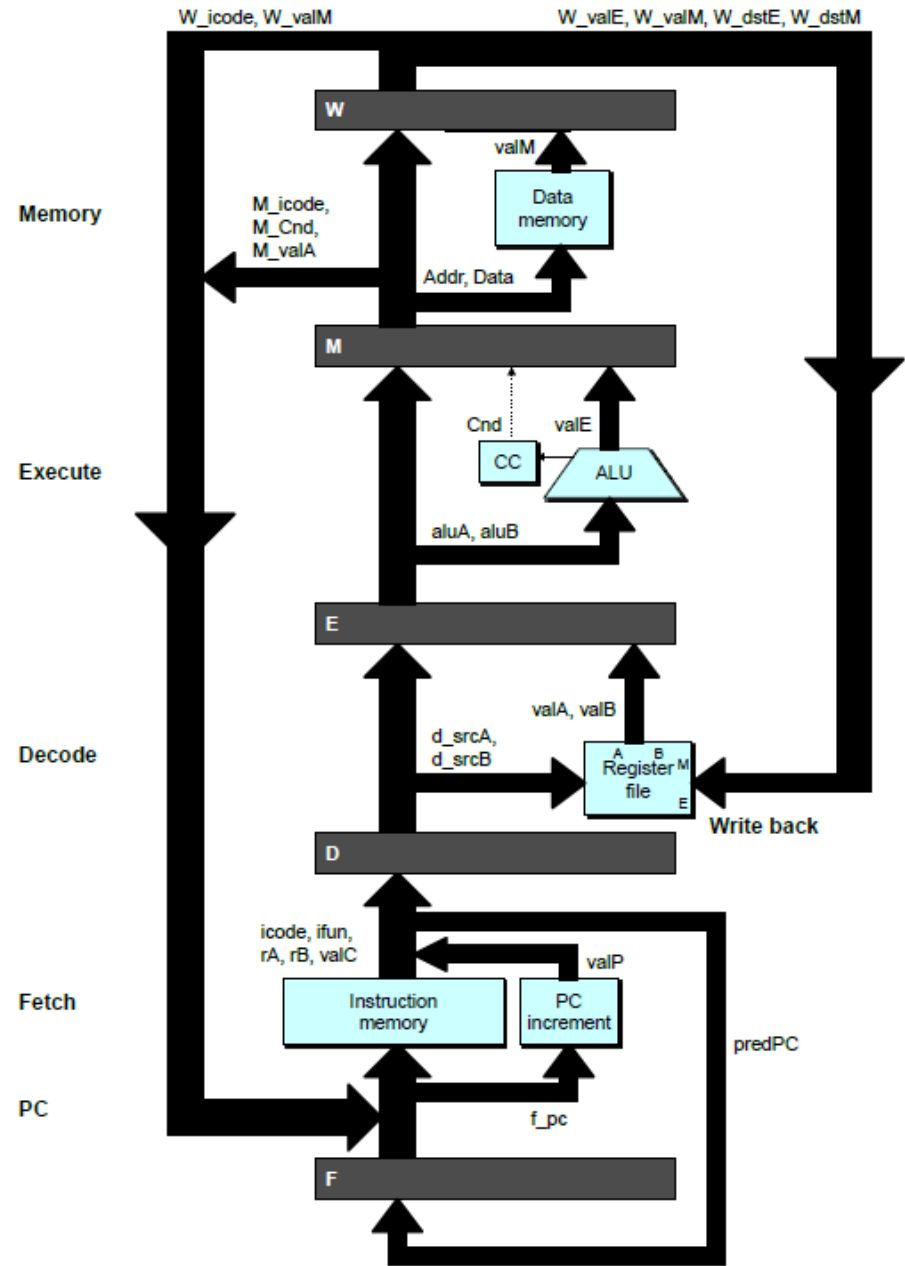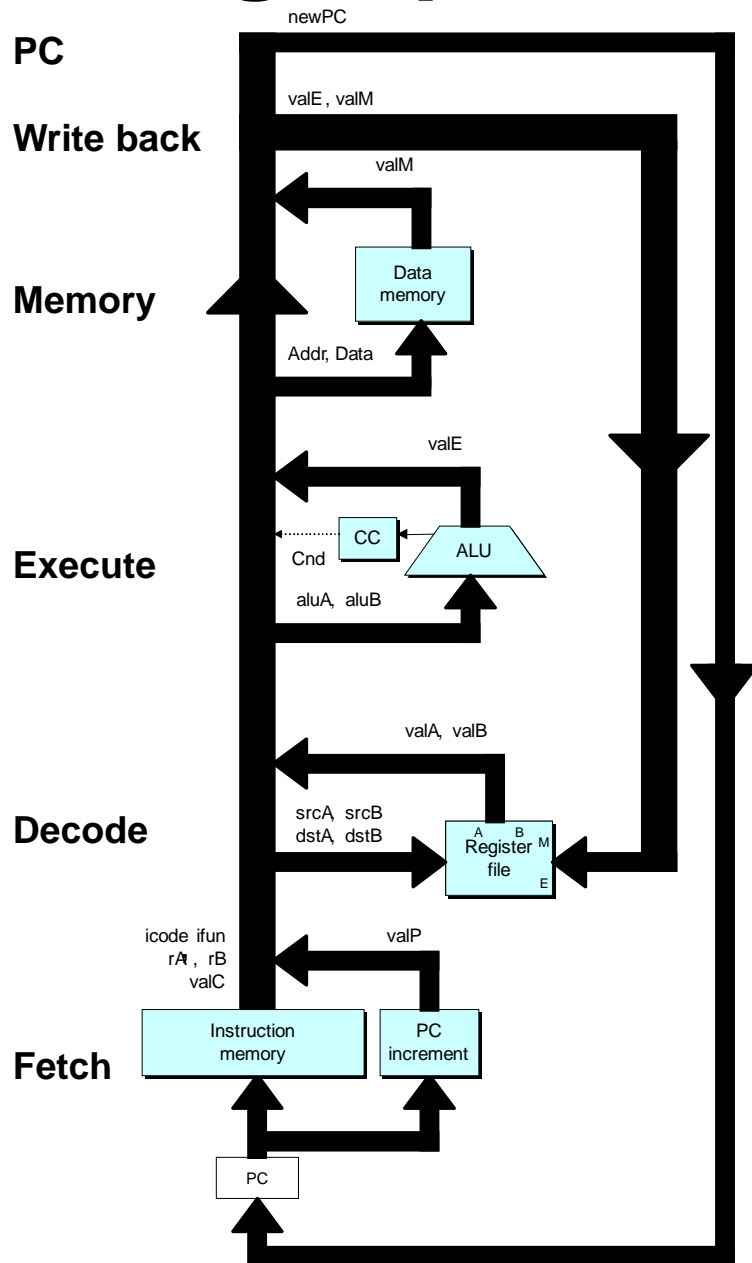
Pipeline Stages

- ## 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
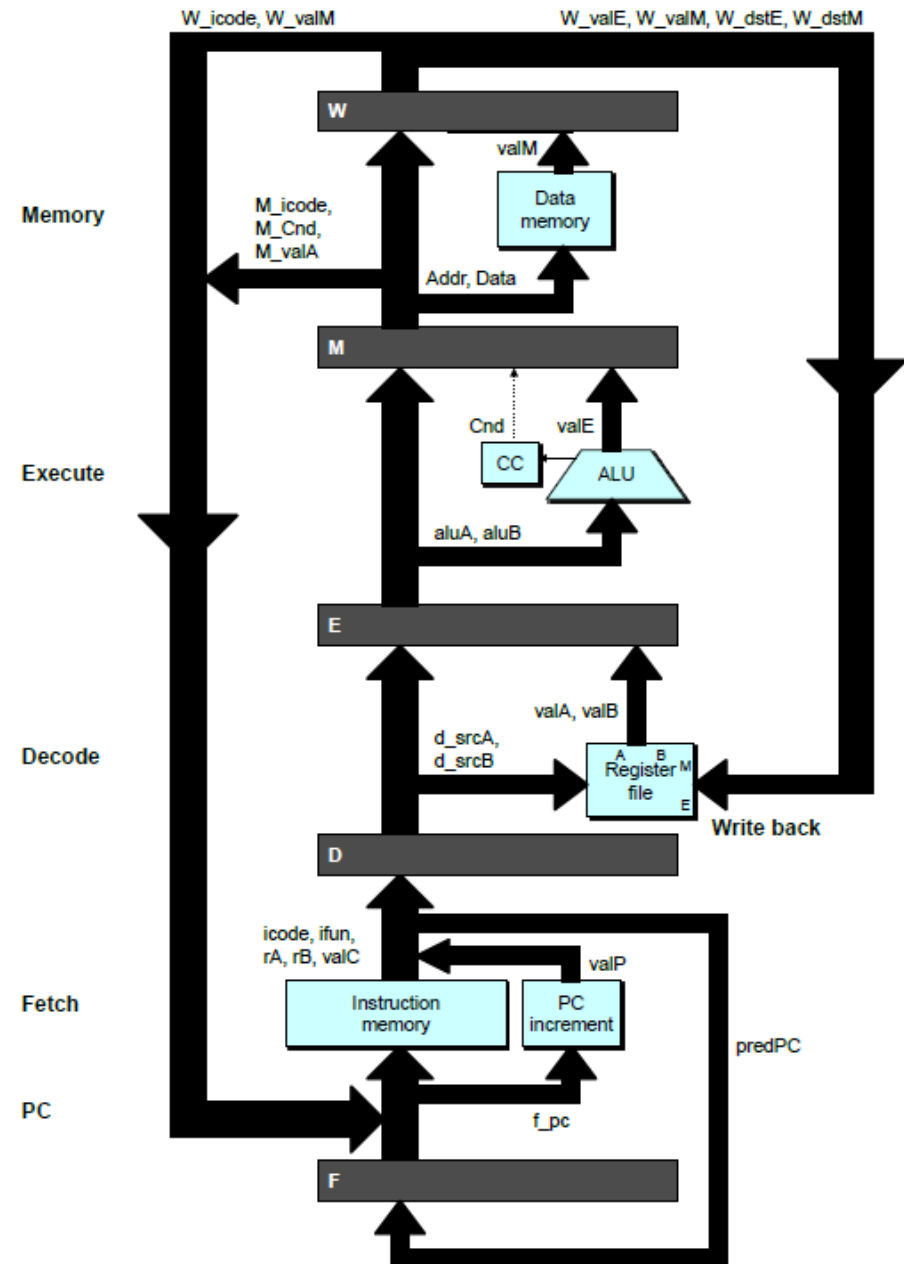
PIPE- Hardware

- **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
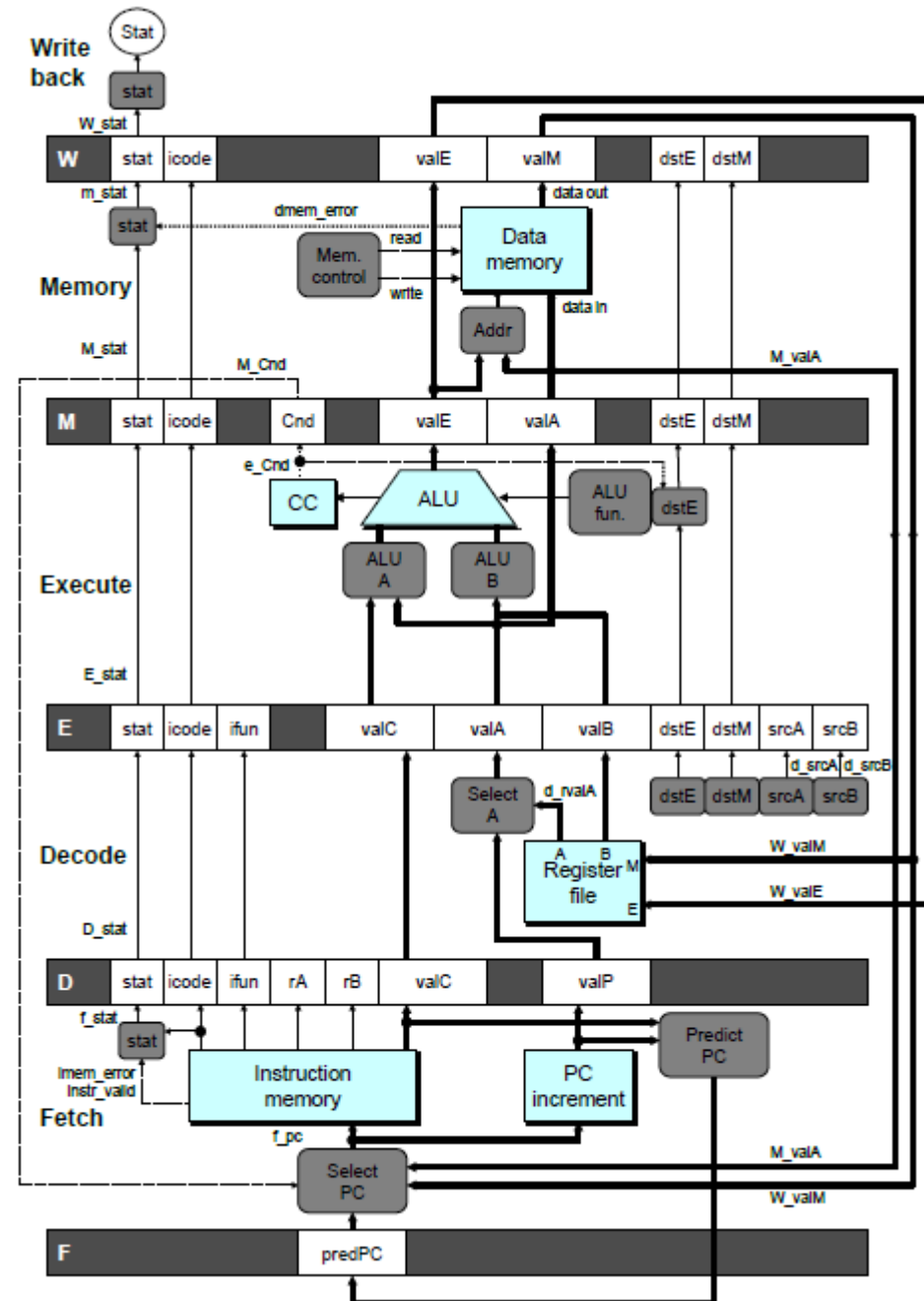
Signal Naming Conventions

- ## S_Field
  - Value of Field held in stage S pipeline register

- ## s_Field
  - Value of Field computed in stage S
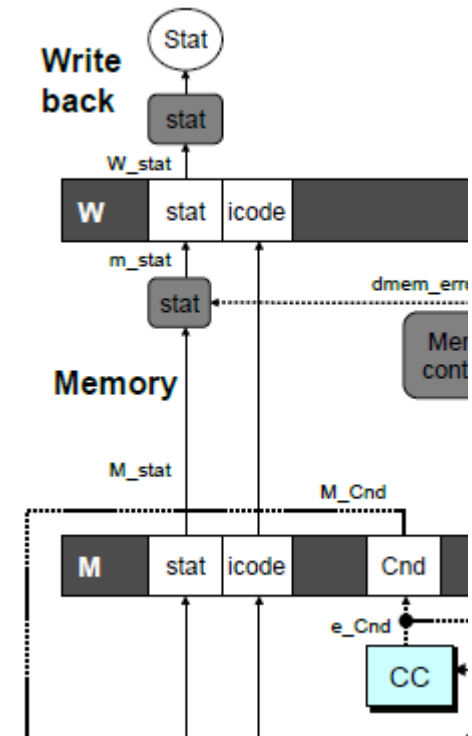
Feedback Paths

- **Predicted PC**
  - **Guess value of next PC**

- **Branch information**
  - **Jump taken/not-taken**
  - **Fall-through or target address**

- **Return point**
  - **Read from memory**

- **Register updates**
  - **To register file write ports**
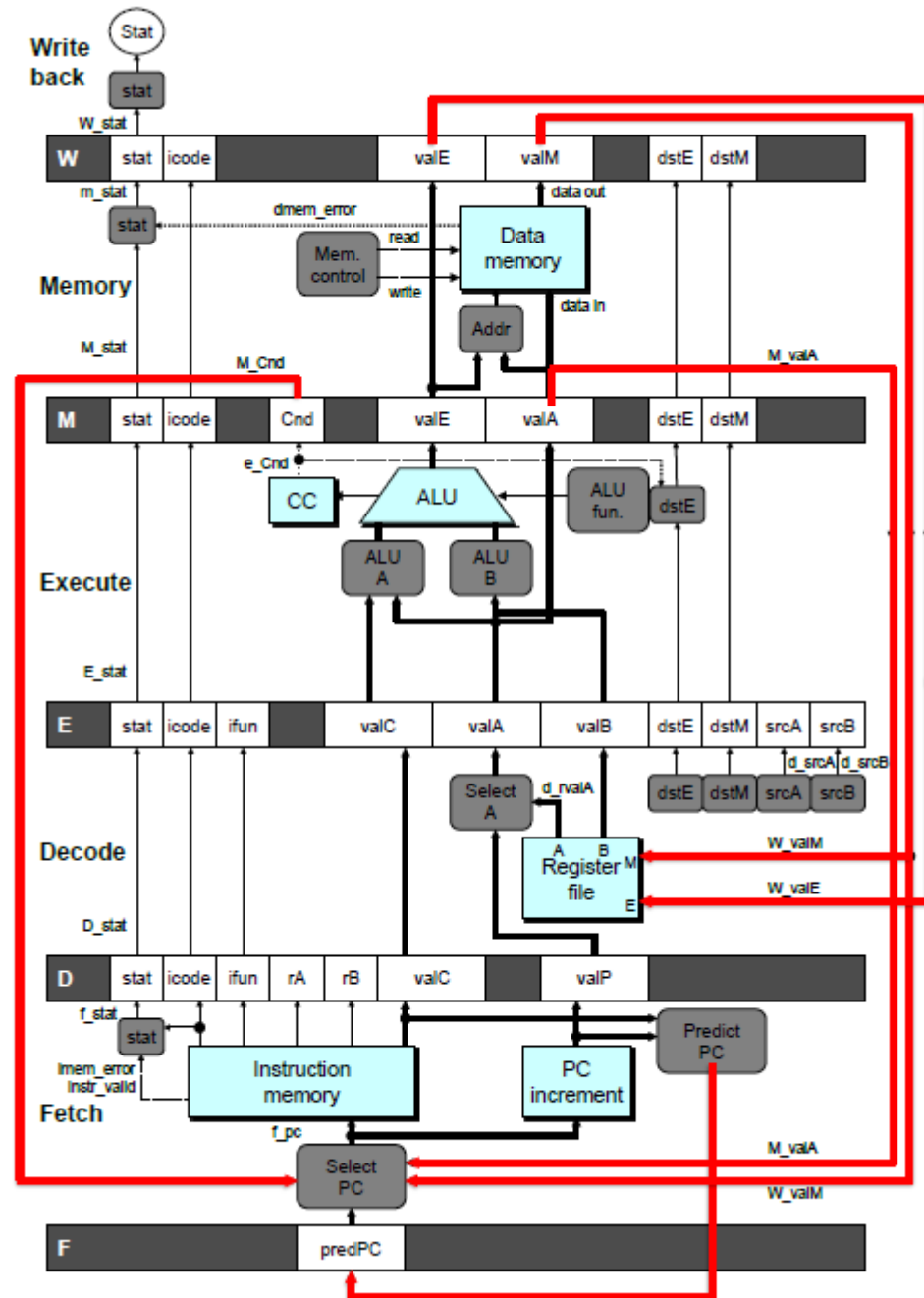
Predicting the PC
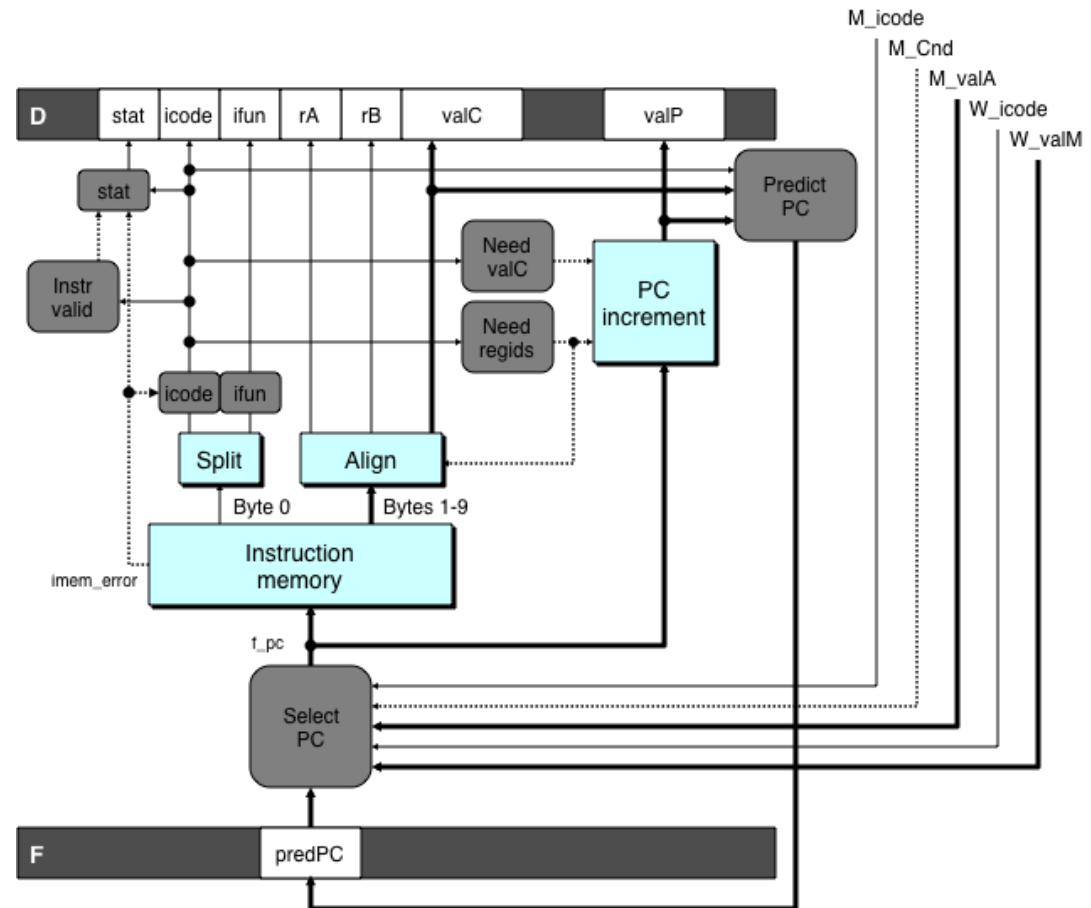
- **Start fetch of new instruction after current one has completed fetch stage**
  - **Not enough time to reliably determine next instruction**
- **Guess which instruction will follow**
  - **Recover if prediction was incorrect**

Our Prediction Strategy

- **Instructions that Don't Transfer Control**
  - **Predict next PC to be valP**
  - **Always reliable**

- **Call and Unconditional Jumps**
  - **Predict next PC to be valC (destination)**
  - **Always reliable**

- **Conditional Jumps**
  - **Predict next PC to be valC (destination)**
  - **Only correct if branch is taken**
    - **Typically right 60% of time**

- **Return Instruction**
  - **Don't try to predict**

Recovering from PC Misprediction
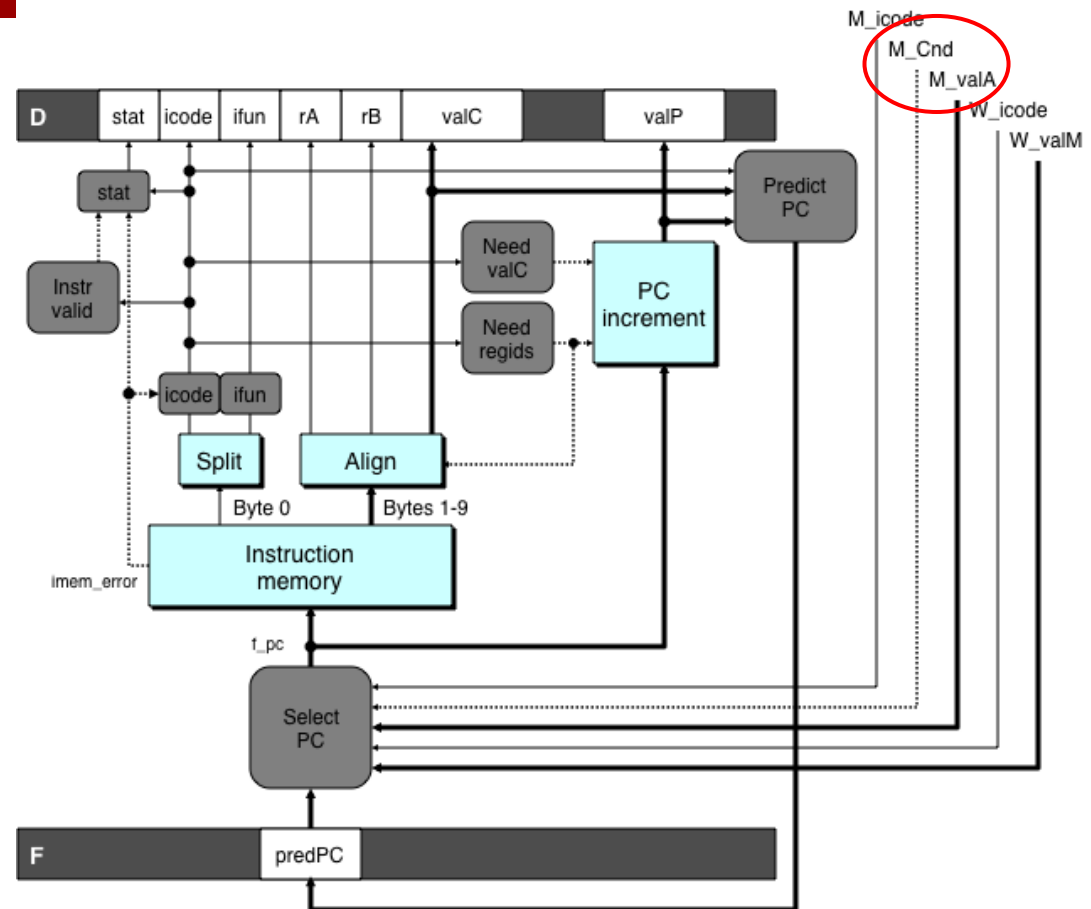
- **Mispredicted Jump**
  - **Will see branch condition flag once instruction reaches memory stage**
  - **Can get fall-through PC from valA (value M_valA)**
- **Return Instruction**
  - **Will get return PC when `ret` reaches write-back stage (W_valM)**

Pipeline Demonstration

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

■ **File: demo-basic.ys**

1	2	3	4	5	6	7	8	9
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

Cycle 5

W	I1
M	I2
E	I3
D	I4
F	I5
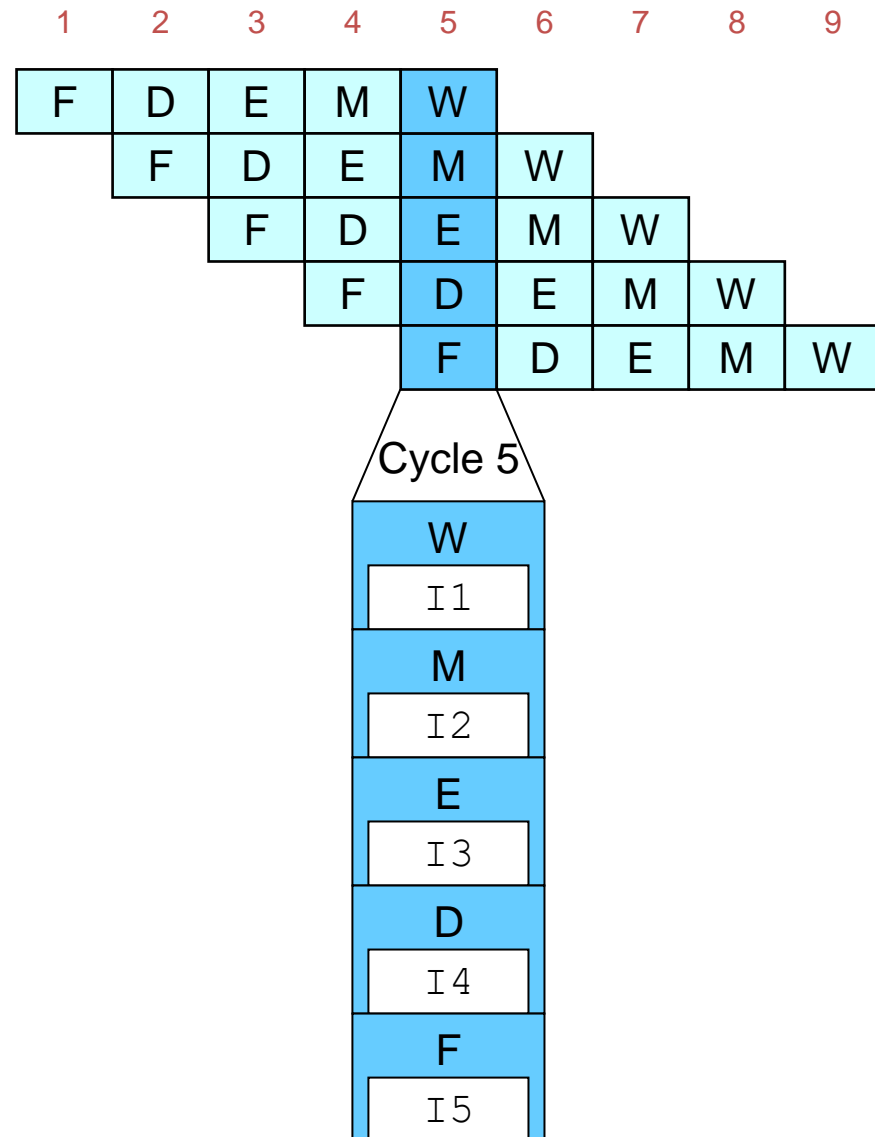
Data Dependencies: No Nop

 # 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
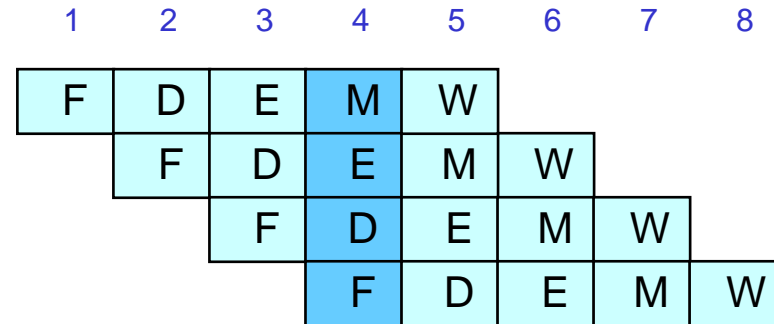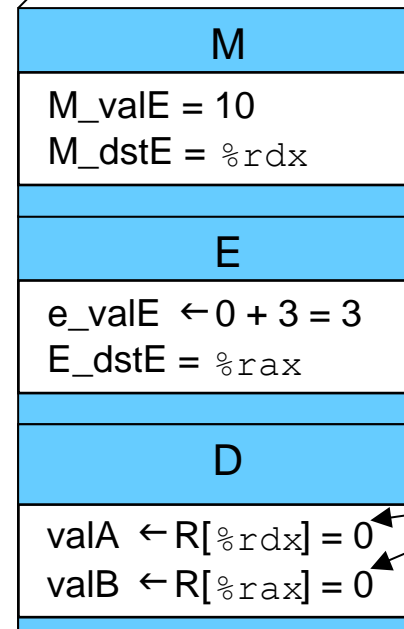
Error

Data Dependencies: 1 Nop

```
# demo-h1.ys
```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

```
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: addq %rdx,%rax
```
F D E M W

```
0x017: halt
```
F D E M W

Cycle 5

W

R[%rdx] ← 10

M

M_valE = 3
M_dstE = %rax

•
•
•

D

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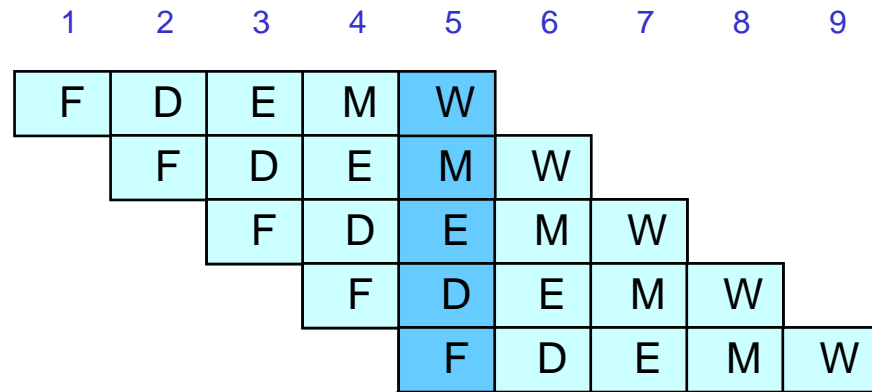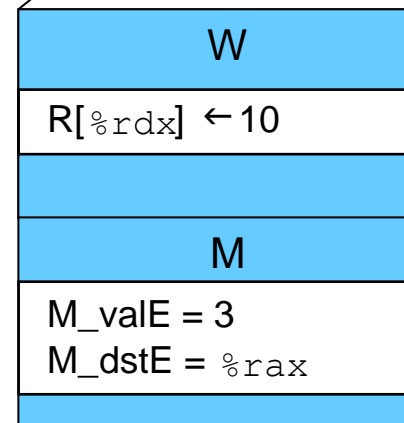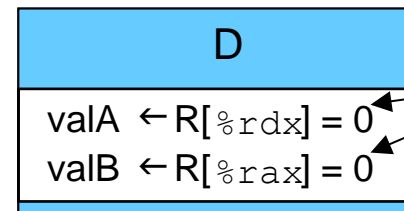
Error

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
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

Cycle 6

W
R[%rax] ← 3

• • •

D
valA ← R[%rdx] = 10
valB ← R[%rax] = 0

Error
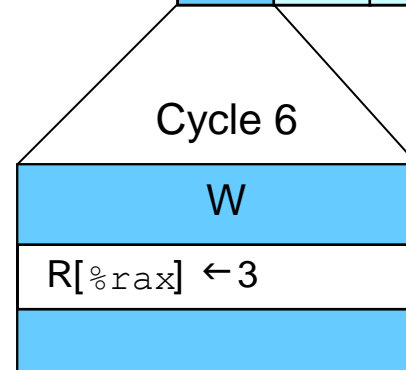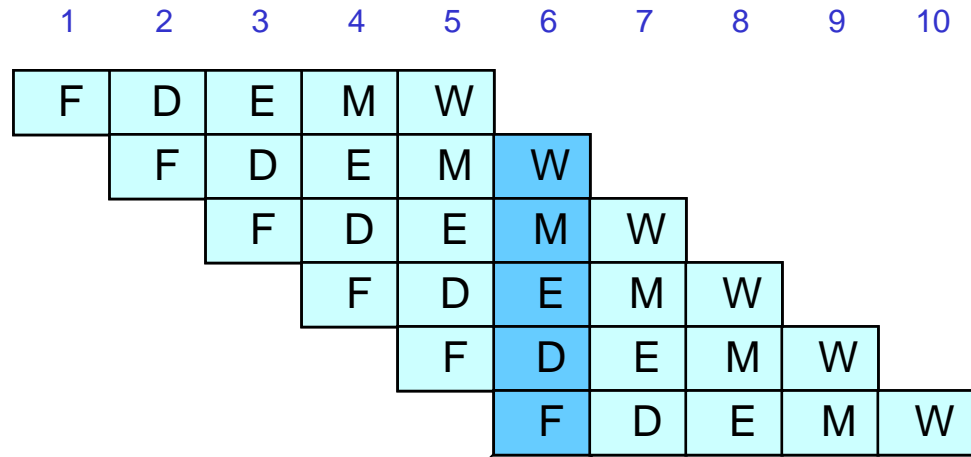
Data Dependencies: 3 Nop's

 # demo-h3.ys

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: 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			
					F	D	E	M	W		
						F	D	E	M	W	
							F	D	E	M	W

Cycle 6

W
R[%rax] ← 3

Cycle 7

D
valA ← R[%rdx] = 10
valB ← R[%rax] = 3
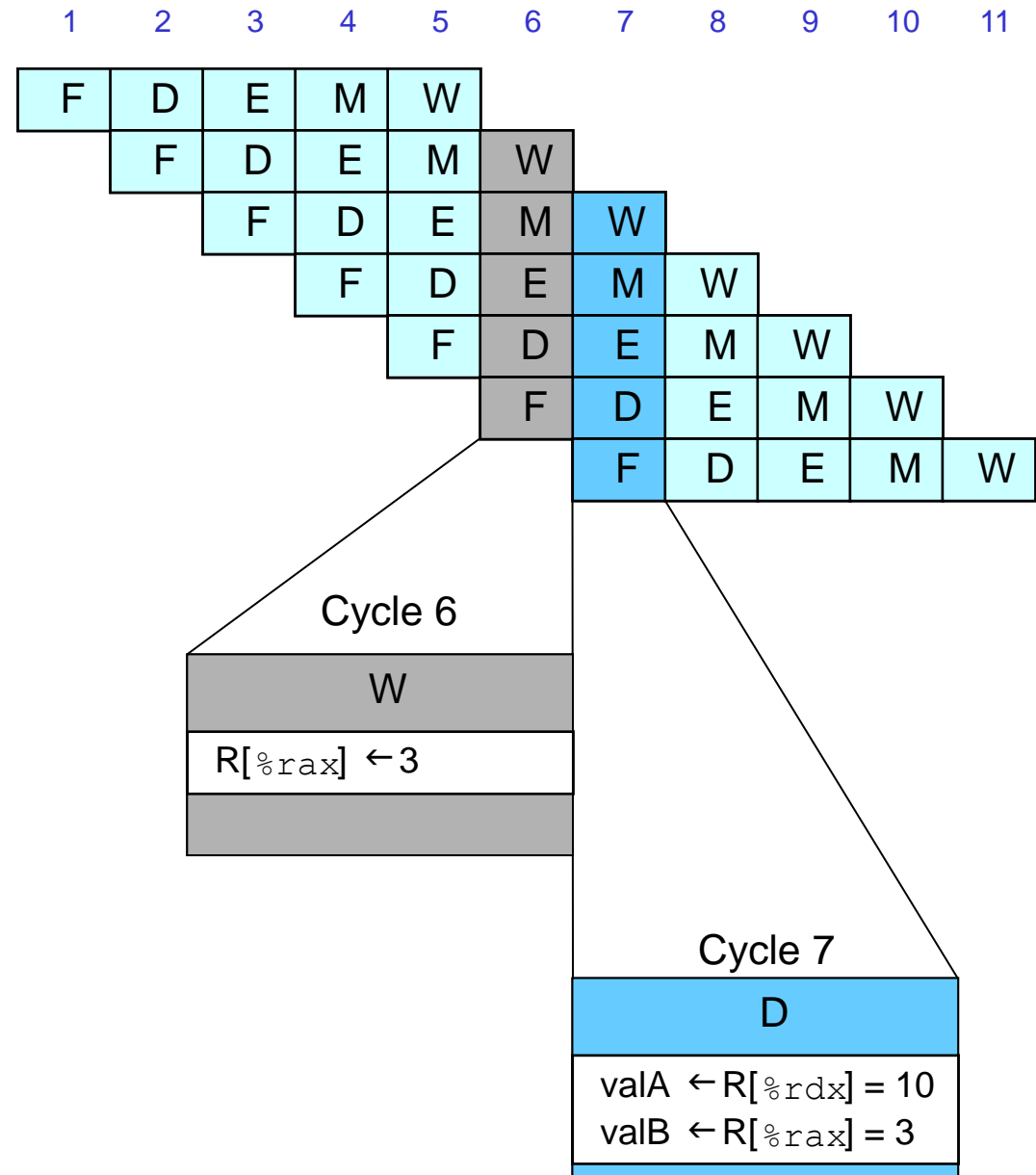
Stalling for Data Dependencies

```
# 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

- **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**
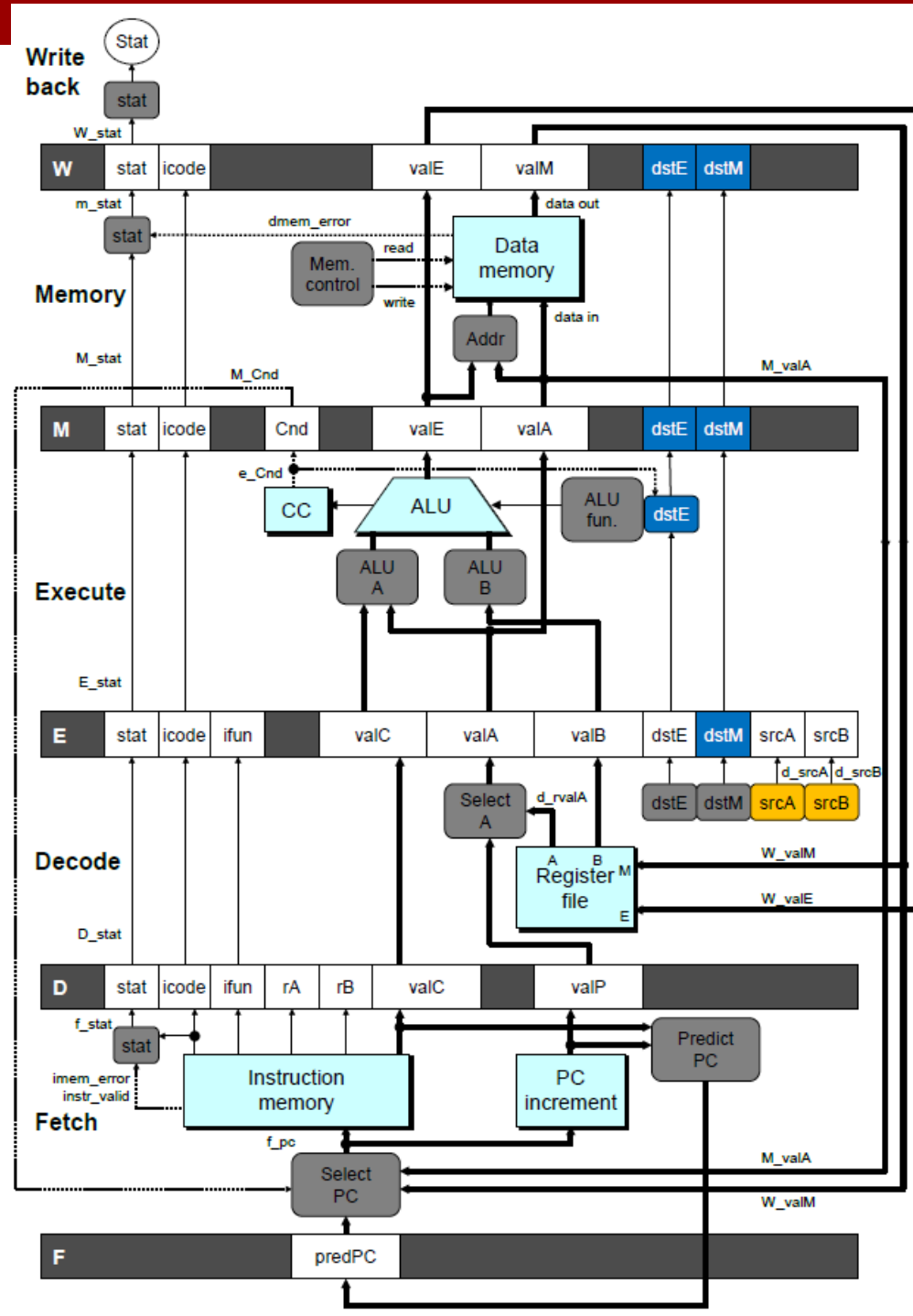
Detecting Stall Condition

```
# demo-h2.ys

0x000: irmovq $10,%rdx

0x00a: irmovq  $3,%rax

0x014: nop

0x015: nop

       bubble

0x016: addq %rdx,%rax

0x018: halt
```
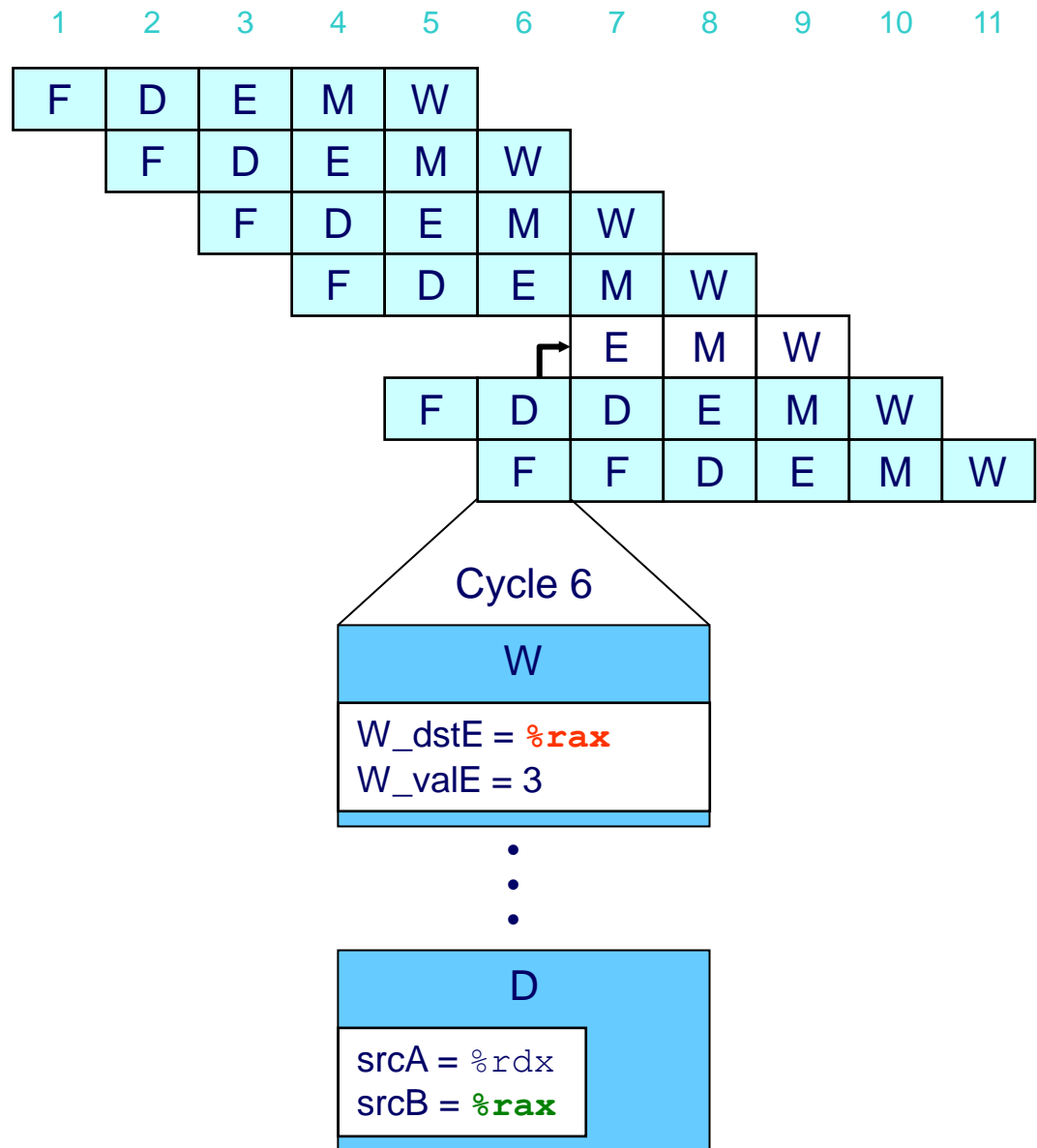
	1	2	3	4	5	6	7	8	9	10	11
	F	D	E	M	W						
		F	D	E	M	W					
			F	D	E	M	W				
				F	D	E	M	W			
						E	M	W			
					F	D	D	E	M	W	
						F	F	D	E	M	W

Cycle 6

W
W_dstE = **%rax**
W_valE = 3

•
•
•

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

Stalling X3

 # demo-h0.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					
bubble				E	M	W					
bubble					E	M	W				
bubble						E	M	W			
`0x014: addq %rdx,%rax`		F	D	D	D	D	E	M	W		
`0x016: halt`			F	F	F	F	D	E	M	W	

Cycle 6

W
W_dstE = **%rax**

Cycle 5

M
M_dstE = **%rax**

· · ·

Cycle 4

E
e_dstE = **%rax**

D	D	D
srcA = %rdx srcB = **%rax**	srcA = %rdx srcB = **%rax**	srcA = %rdx srcB = **%rax**
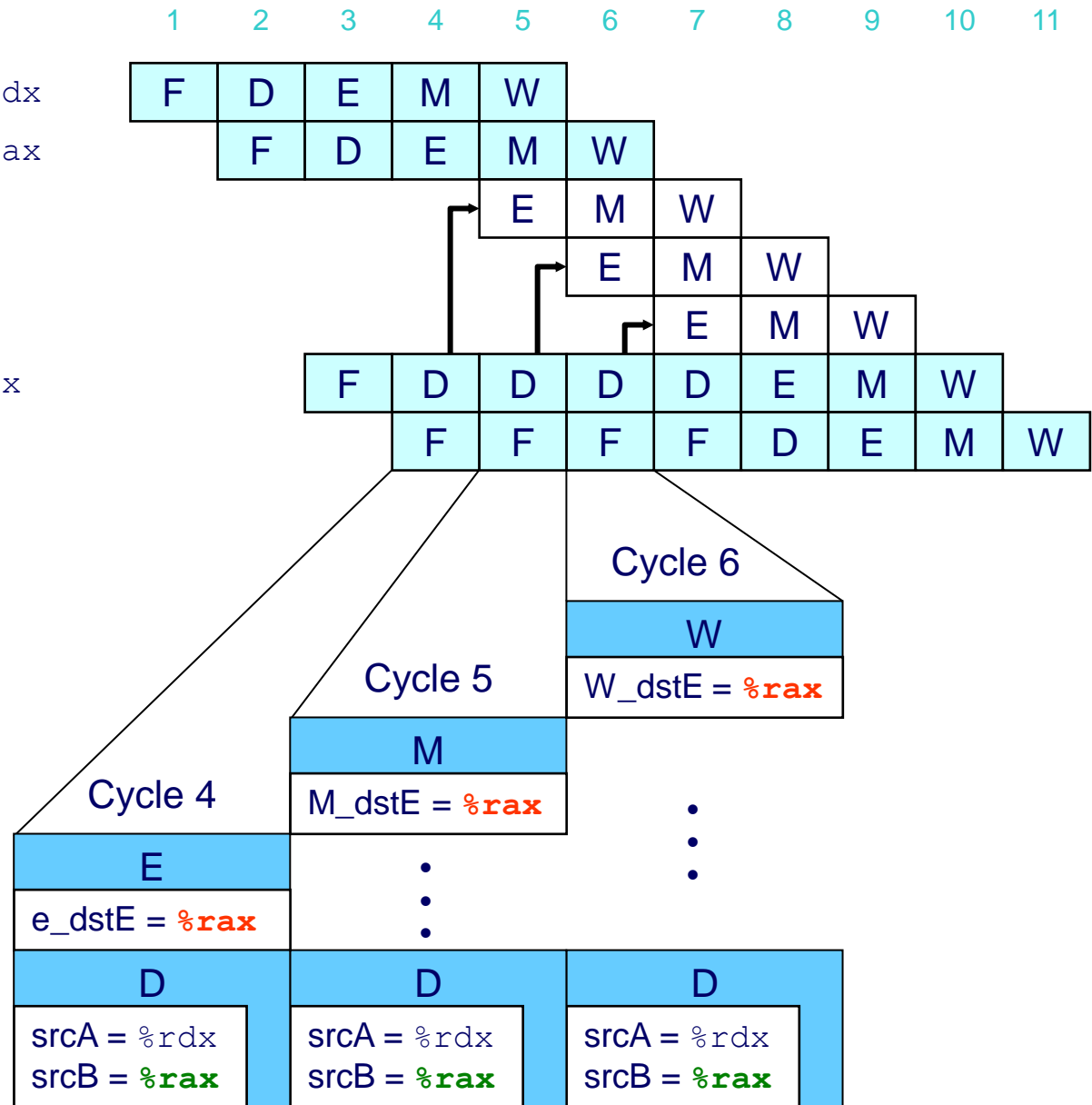
What Happens When Stalling?

```
# 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
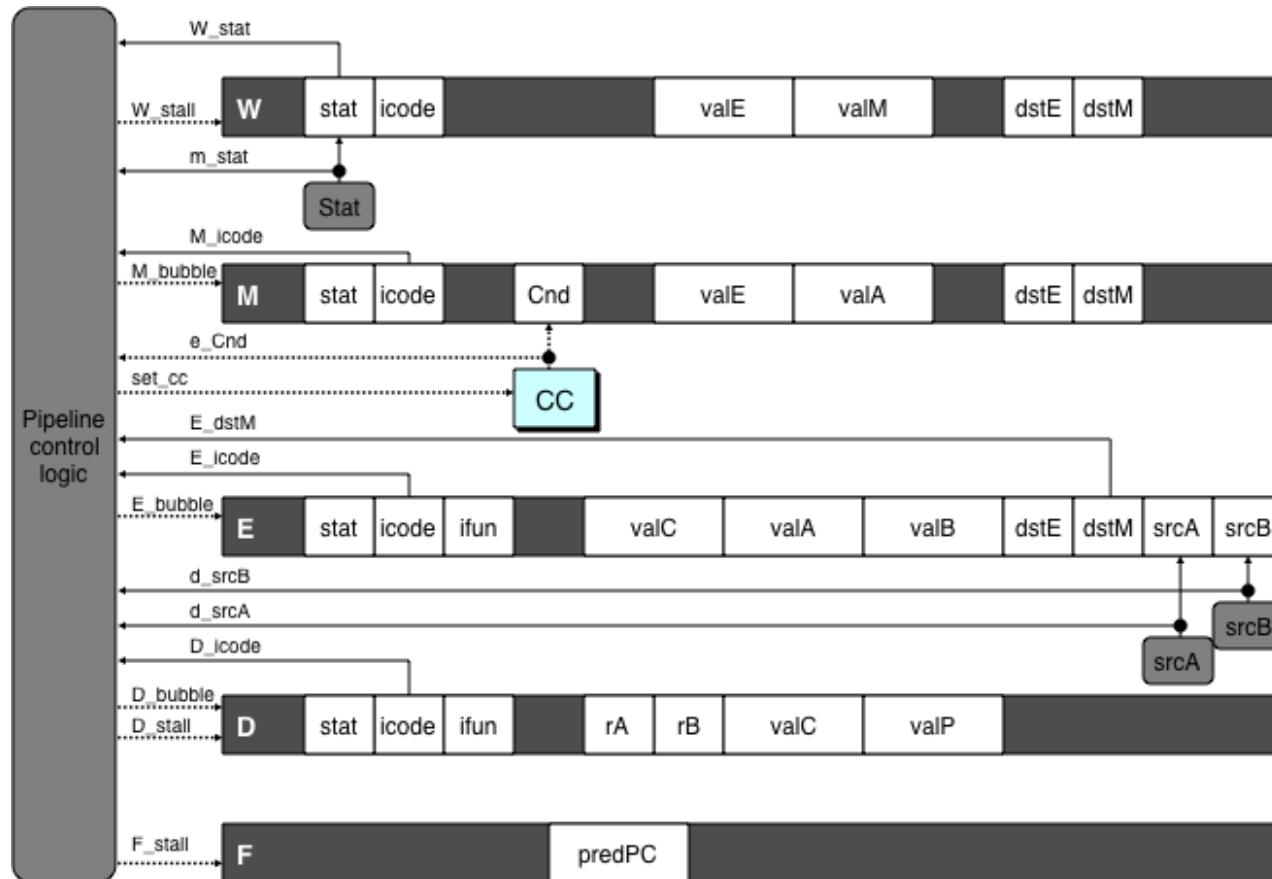
■ Pipeline Control

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

Pipeline Register Modes

Normal

Input = y   Output = x

x

stall   bubble
= 0   = 0

⇨ Rising clock

⇨ Output = y

y

Stall

Input = y   Output = x

x

stall = 1   bubble = 0

⇨ Rising clock

⇨ Output = x

x

Bubble

Input = y   Output = x

x

stall = 0   **bubble = 1**

⇨ Rising clock

⇨ Output = `nop`

n
o
p
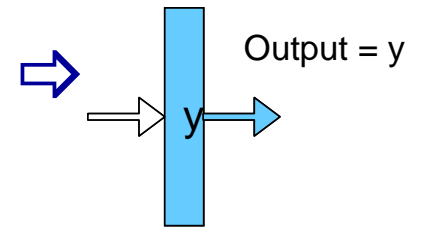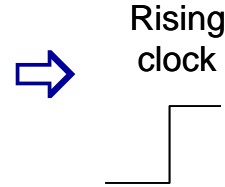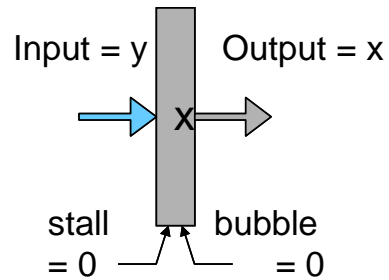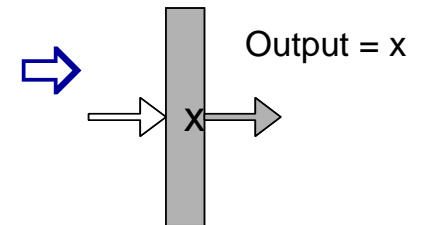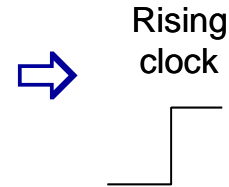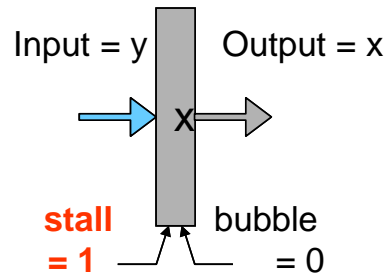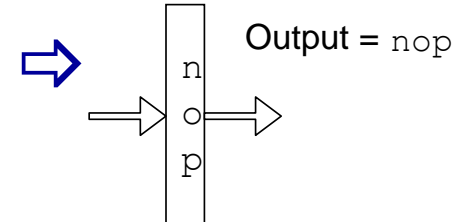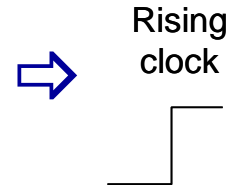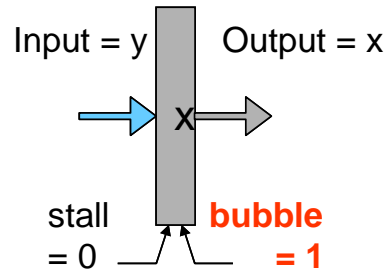
Data Forwarding

- ## 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
```


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

Cycle 6

W	
W_dstE = **%rax** W_valE = 3	R[%rax] ← 3

D	
srcA = %rdx srcB = **%rax**	valA ← R[%rdx] = 10 valB ← W_valE = 3
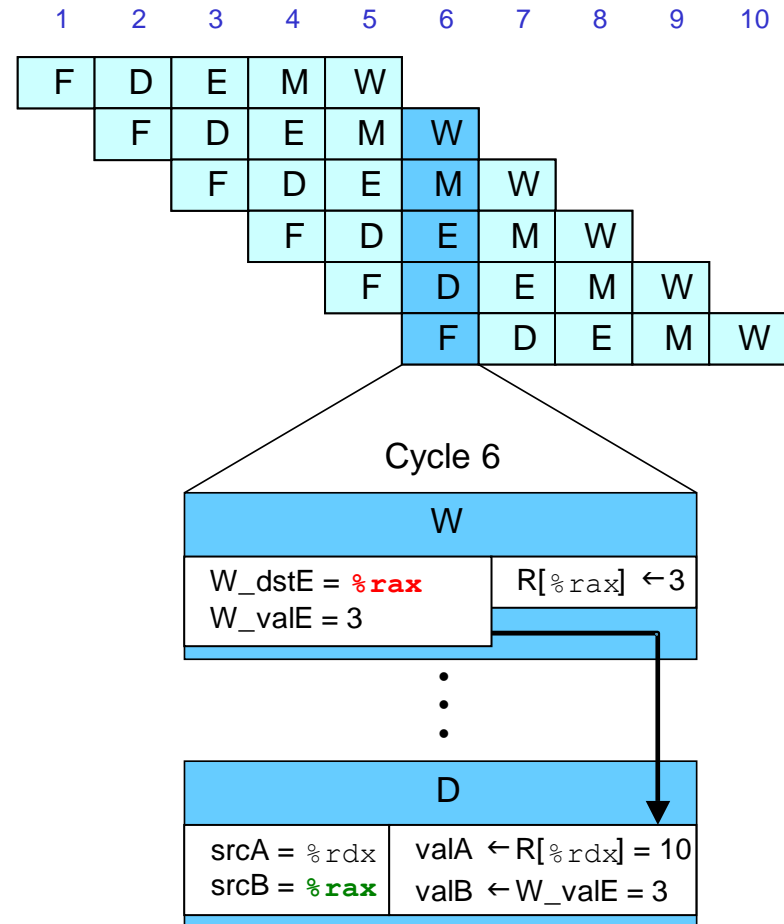
Bypass Paths

■ **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**
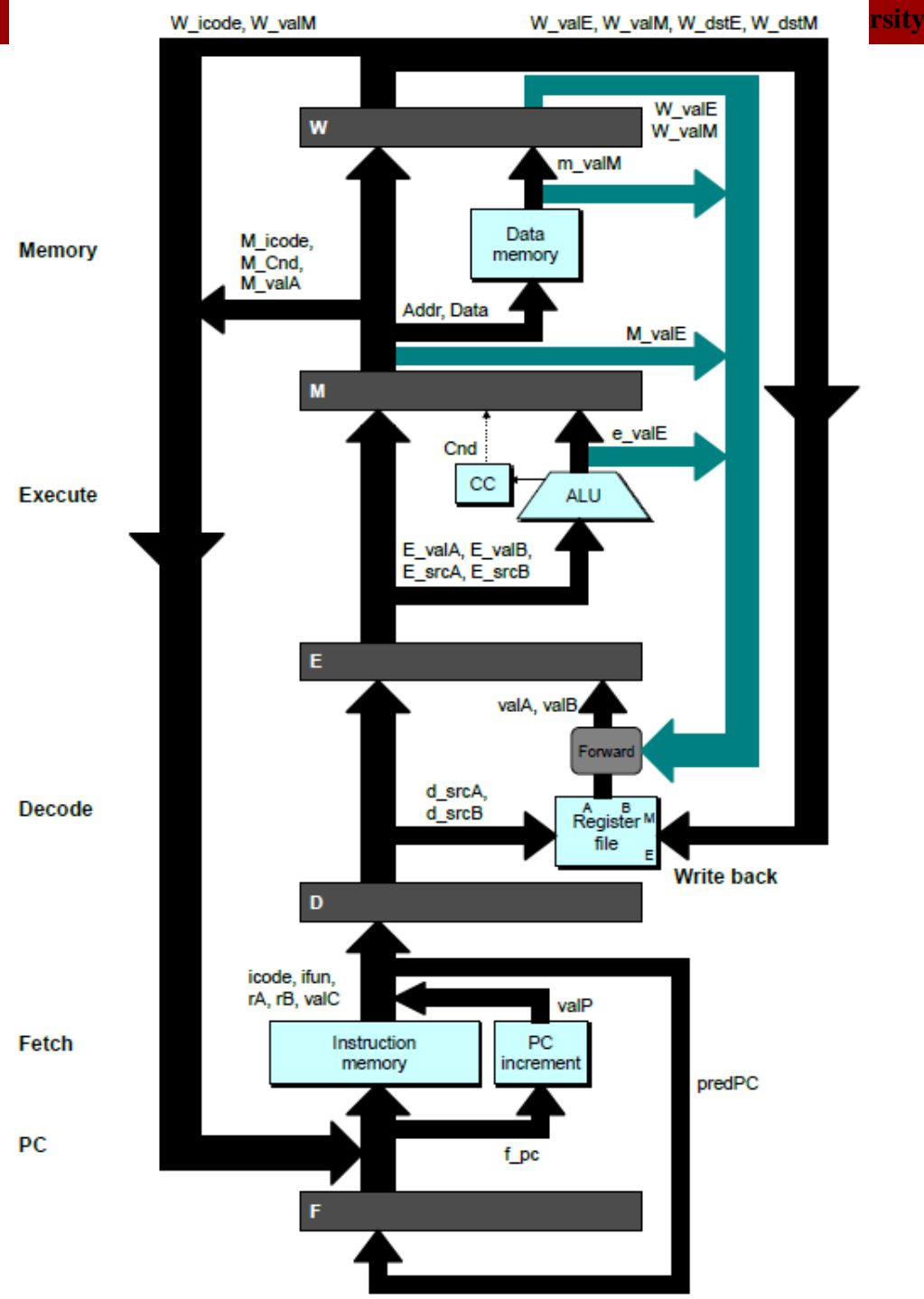
Data Forwarding Example #2

 # 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
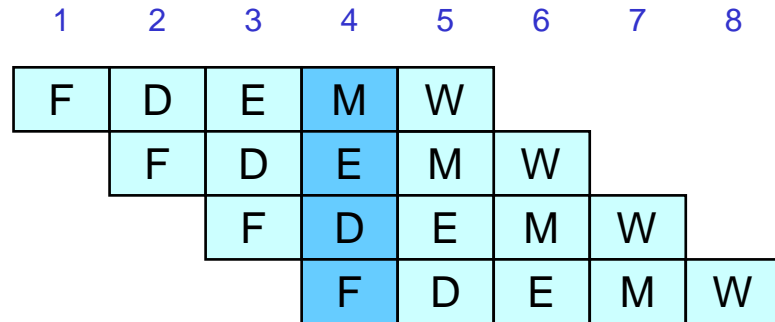
- **Register %rdx**
  - **Generated by ALU during previous cycle**
  - **Forward from memory as valA**

- **Register %rax**
  - **Value just generated by ALU**
  - **Forward from execute as valB**

Cycle 4

M

M_dstE = %rdx
M_valE = 10

E

E_dstE = %rax
e_valE ← 0 + 3 = 3

D

srcA = %rdx   valA ← M_valE = 10
srcB = %rax   valB ← e_valE = 3
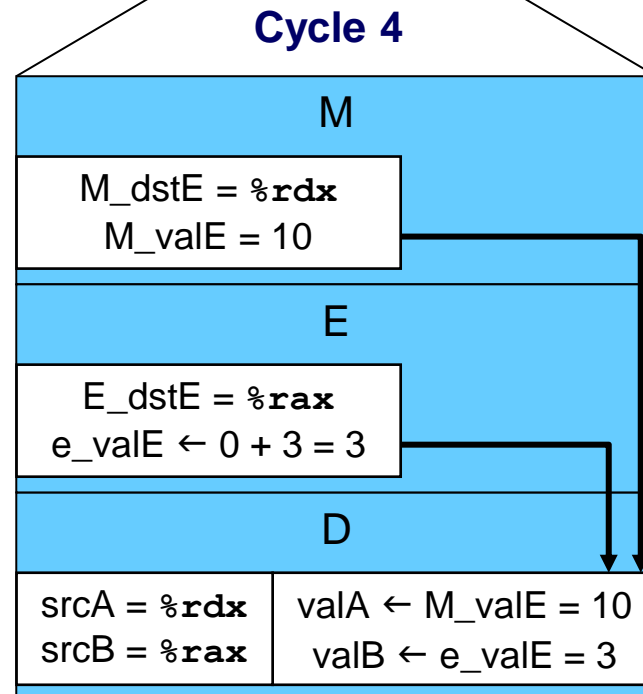
Forwarding Priority

```
# demo-priority.ys

0x000: irmovq $1, %rax

0x00a: irmovq $2, %rax

0x014: irmovq $3, %rax

0x01e: rrmovq %rax, %rdx

0x020: halt
```


Cycle 5
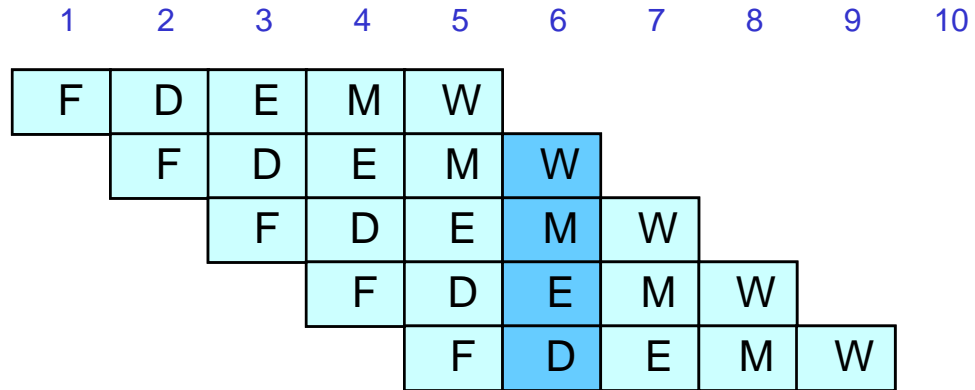
■ **Multiple Forwarding Choices**

- **Which one should have priority**

- **Match serial semantics**

- **Use matching value from earliest pipeline stage**
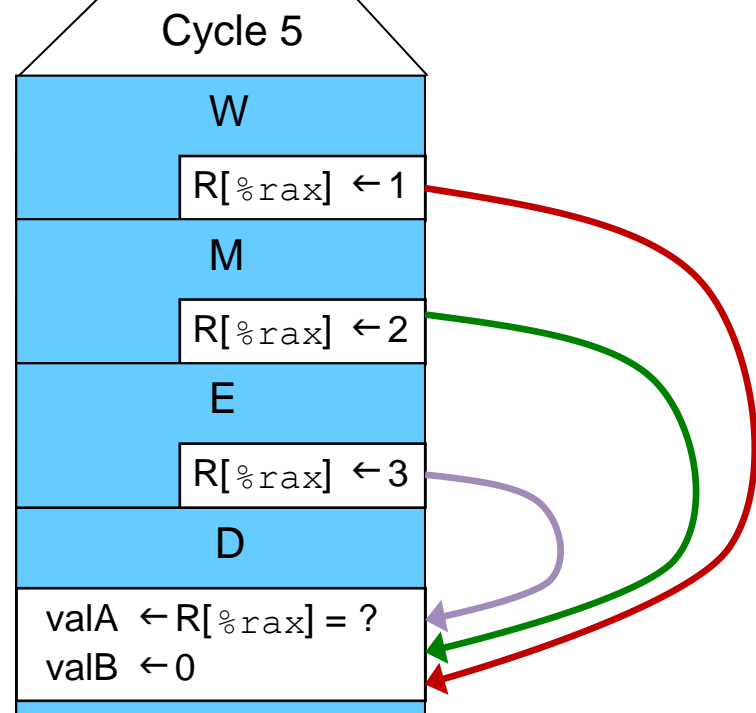
Implementing Forwarding
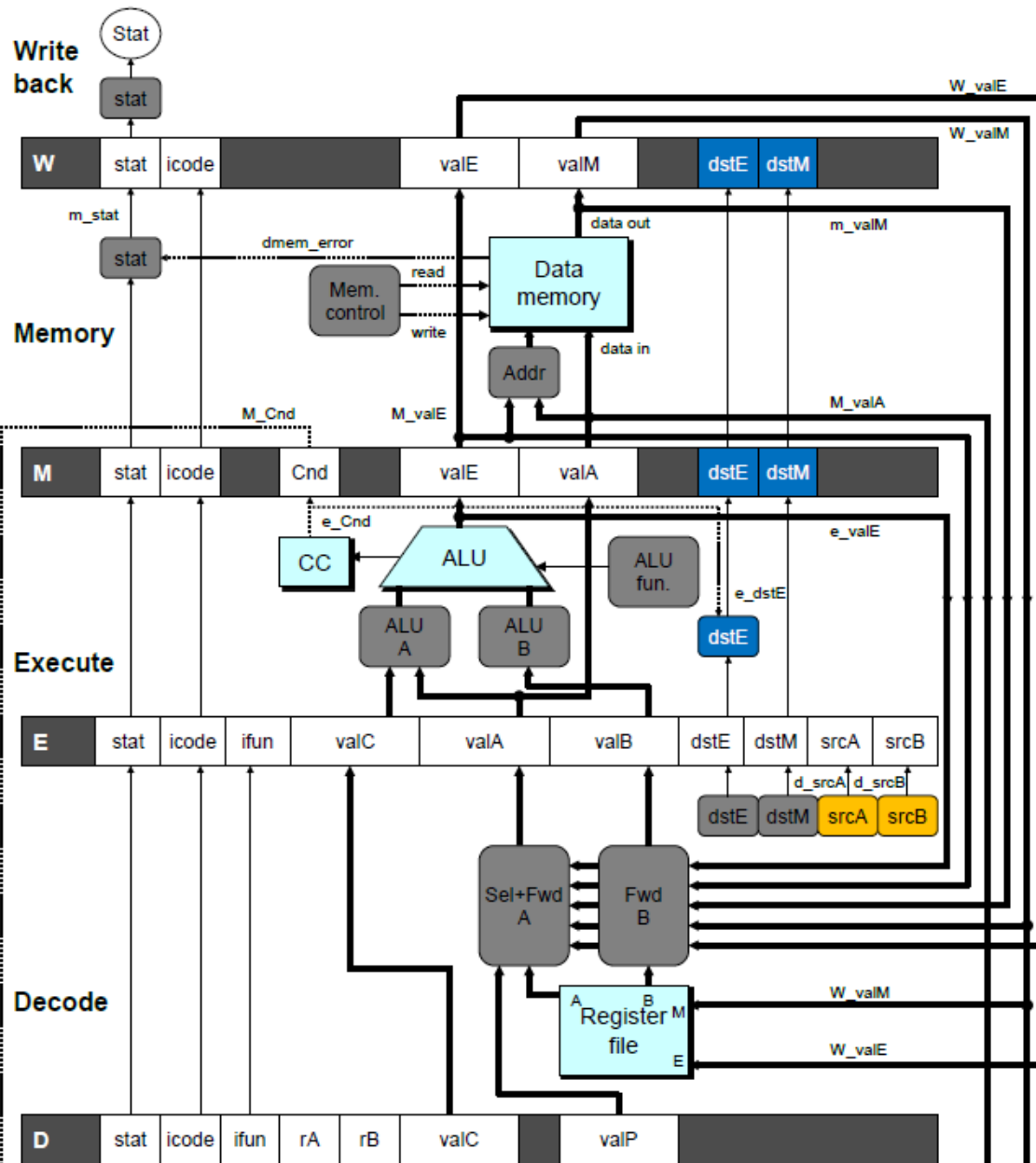
- **Add additional feedback paths from E, M, and W pipeline registers into decode stage**

- **Create logic blocks to select from multiple sources for valA and valB in decode stage**

Implementing Forwarding
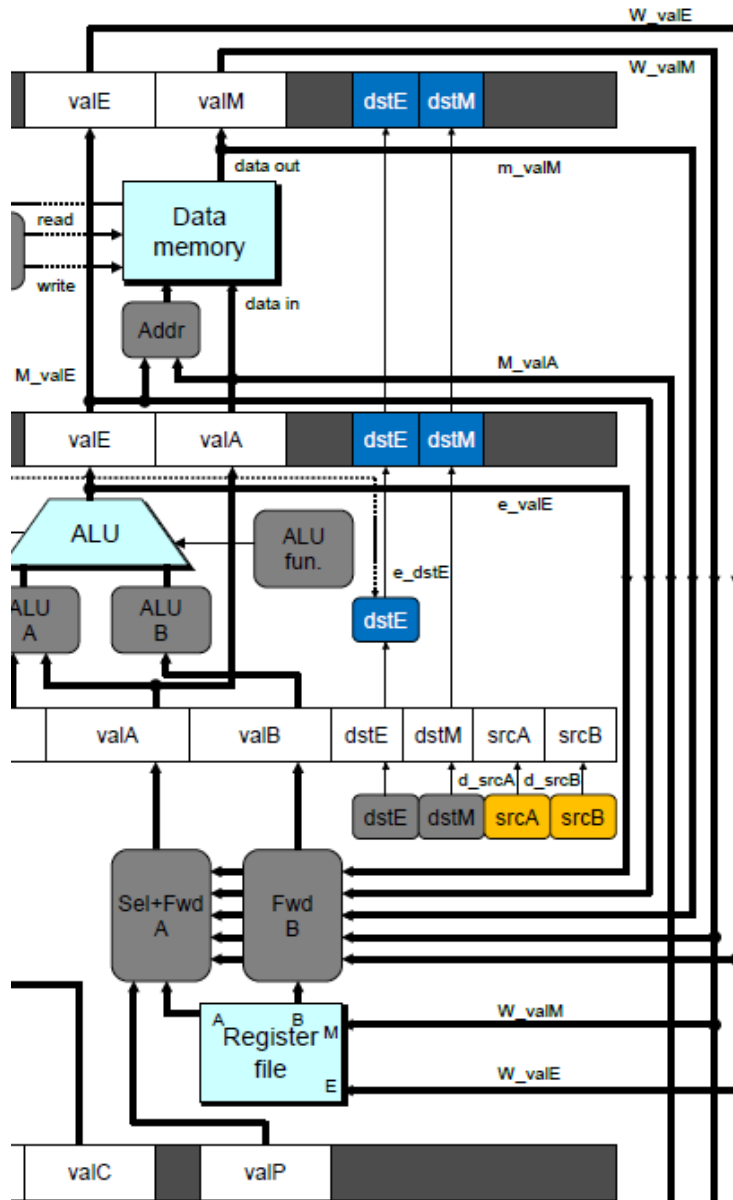

```
## What should be the A value?
int d_valA = [
  # Use incremented PC
    D_icode in { ICALL, IJXX } : D_valP;
  # Forward valE from execute
    d_srcA == e_dstE : e_valE;
  # Forward valM from memory
    d_srcA == M_dstM : m_valM;
  # Forward valE from memory
    d_srcA == M_dstE : M_valE;
  # Forward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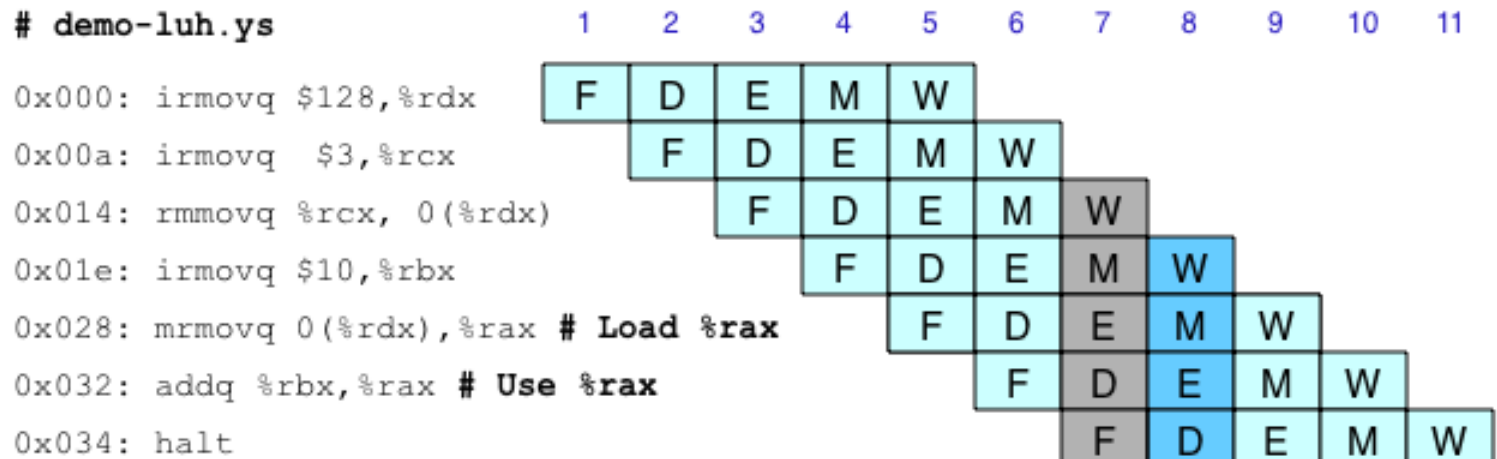
■ **Load-use dependency**

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

Cycle 7

M

$M\_dstE = \%rbx$
$M\_valE = 10$

Cycle 8

M

$M\_dstM = \%rax$
$m\_valM \leftarrow M[128] = 3$

D

$valA \leftarrow M\_valE = 10$
$valB \leftarrow R[\%rax] = 0$

Error

Avoiding 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,%rbx                          F    D    E    M    W

0x028: mrmovq 0(%rdx),%rax # Load %rax                F    D    E    M    W

       bubble                                                  E    M    W

0x032: addq %rbx,%rax # Use %rax                     F    D    D    E    M    W

0x034: halt                                               F    F    D    E    M    W
```
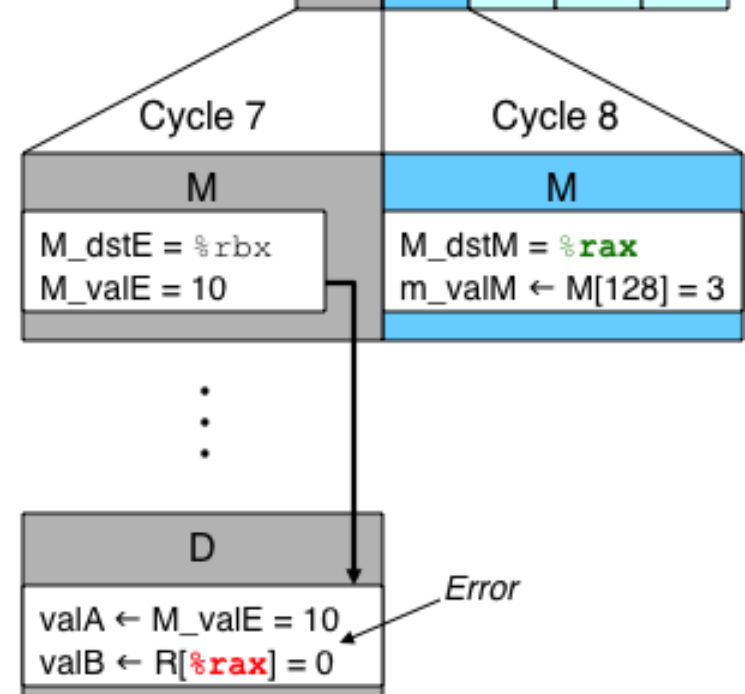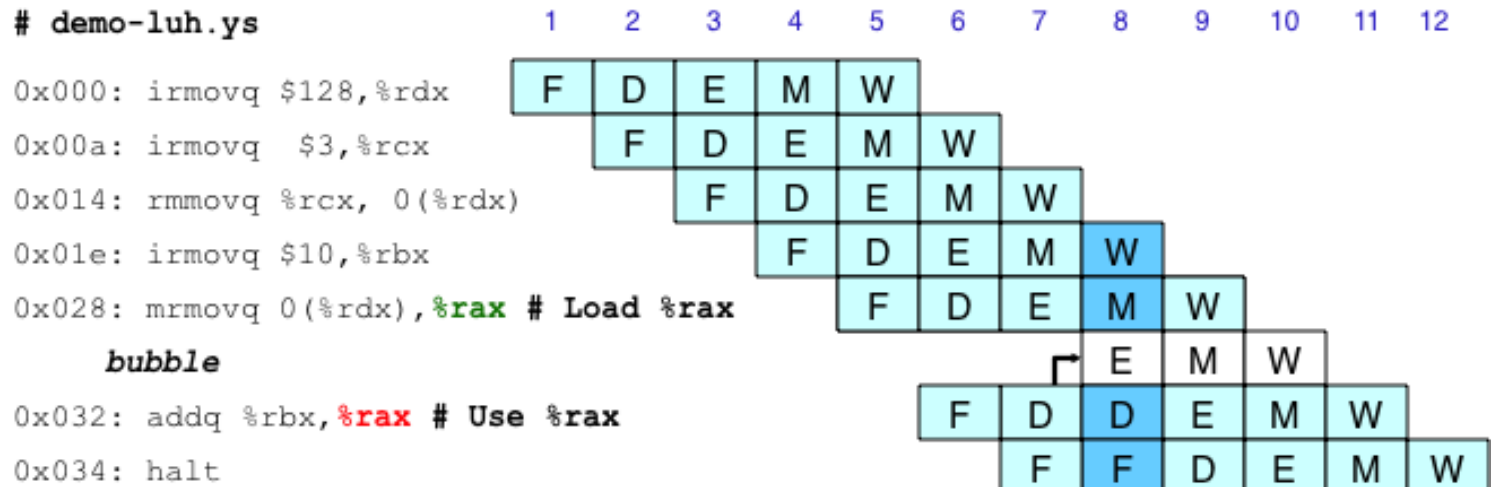
- **Stall using instruction for one cycle**

- **Can then pick up loaded value by forwarding from memory stage**

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

Detecting Load/Use Hazard

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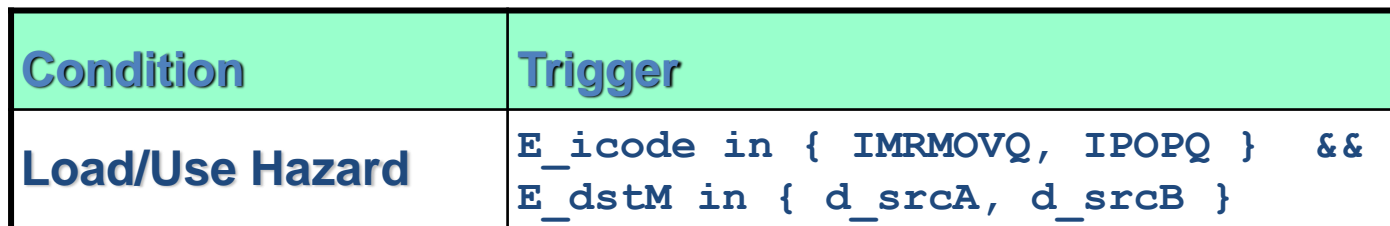
Control for Load/Use Hazard

```
# 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

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**

Branch Misprediction Trace

 # demo-j

		1	2	3	4	5	6	7	8	9
0x000:	xorq %rax,%rax	F	D	E	M	W				
0x002:	jne t # Not taken		F	D	E	M	W			
0x019: t:	irmovq $3, %rdx # Target			F	D	E	M	W		
0x023:	irmovq $4, %rcx # Target+1				F	D	E	M	W	
0x00b:	irmovq $1, %rax # Fall Through					F	D	E	M	W

■ **Incorrectly execute two instructions at branch target**

Cycle 5

M

M_Cnd = 0
M_valA = 0x007

E

valE ← 3
dstE = %rdx

D

valC = 4
dstE = %rcx

F

valC ← 1
rB ← %rax
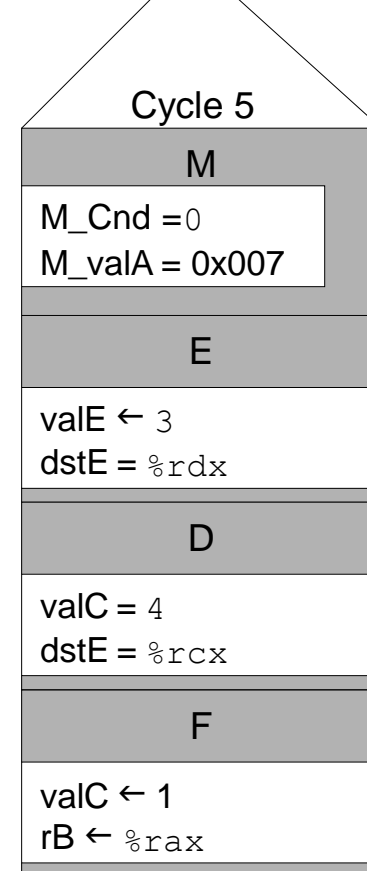
Handling Misprediction

```
# 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
```

	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
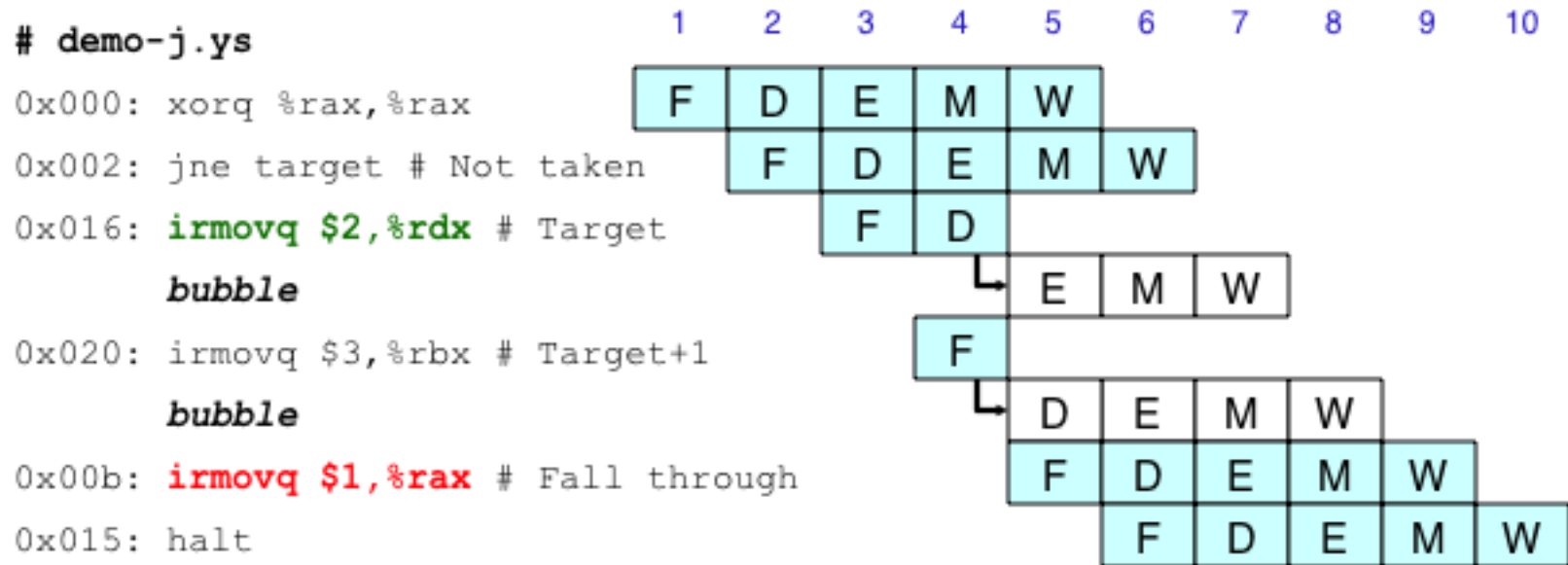
Predict branch as taken

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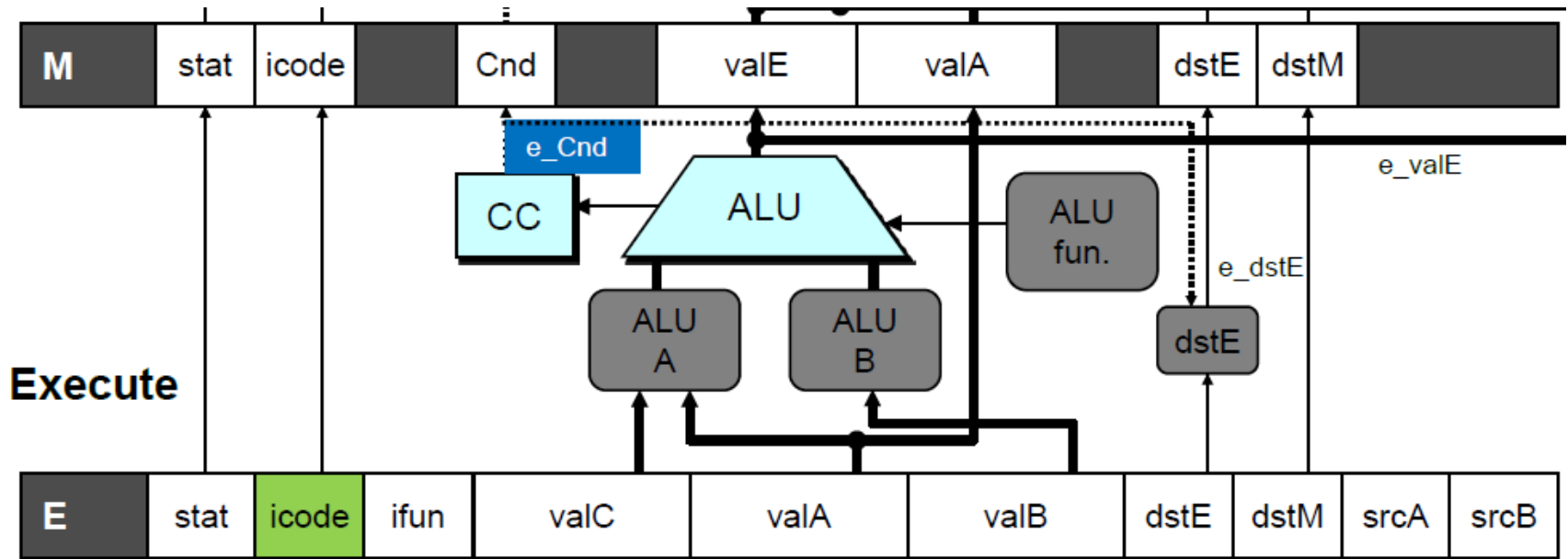
Condition	Trigger
Mispredicted Branch	`E_icode == IJXX & !e_Cnd`

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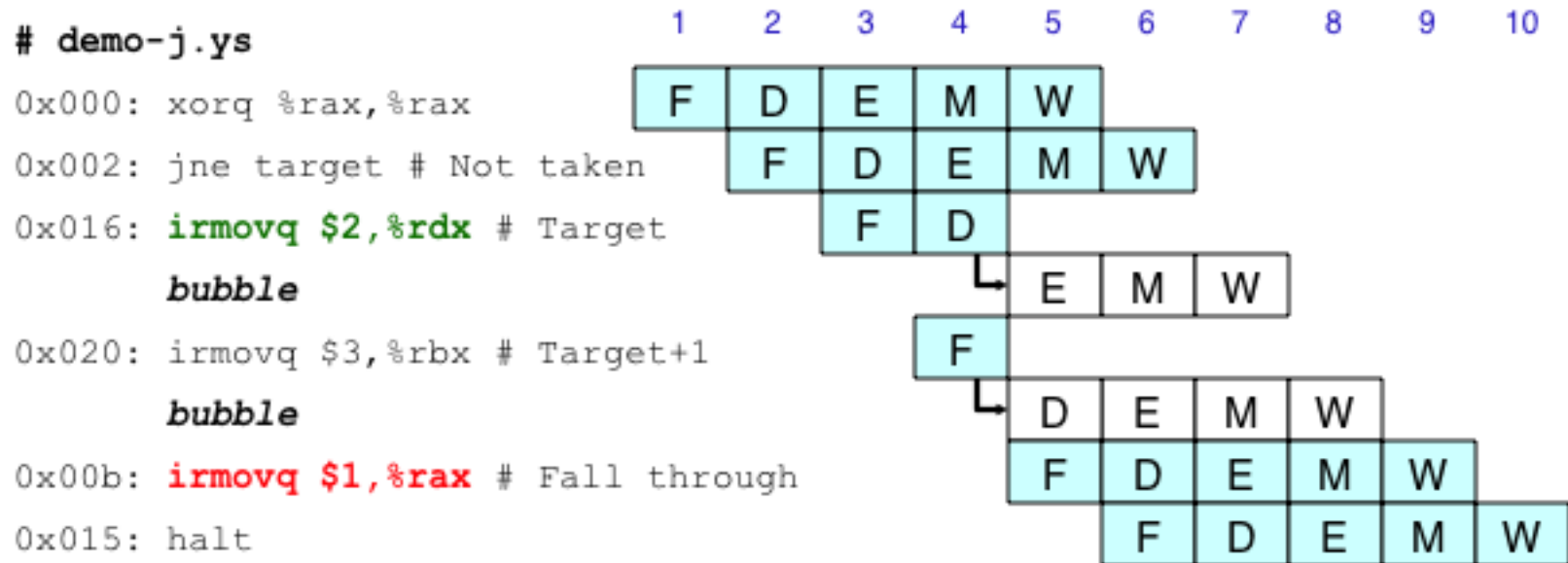
Condition	F	D	E	M	W
Mispredicted Branch	normal	bubble	bubble	normal	normal

Return Example

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**

Incorrect Return Example

```
# demo-ret
```

0x033:	ret	F	D	E	M	W				
0x034:	irmovq $1,%rax # Oops!		F	D	E	M	W			
0x03e:	irmovq $2,%rcx # Oops!			F	D	E	M	W		
0x048:	irmovq $3,%rdx # Oops!				F	D	E	M	W	
0x052:	irmovq $5,%rsi # Return					F	D	E	M	W
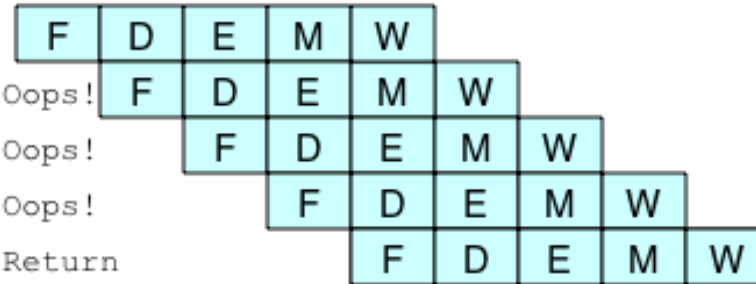
■ **Incorrectly execute 3 instructions following `ret`**

W
valM = 0x0e

M
valE = 1
dstE = %rax

E
valE ← 2
dstE = %rcx

D
valC = 3
dstE = %rdx

F
valC ← 5
rB ← %rsi
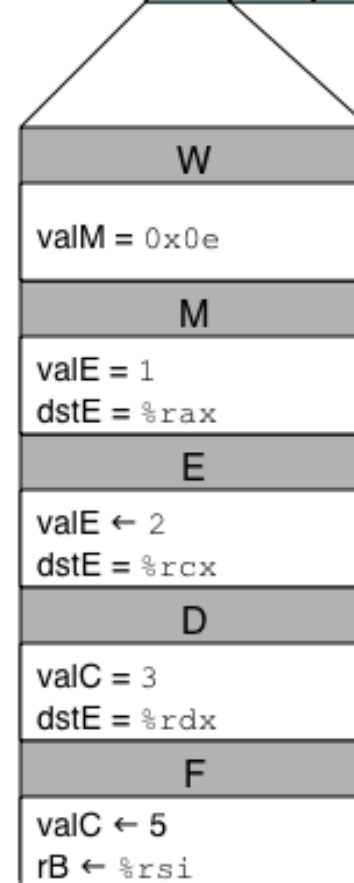
Correct Return Example

```
# 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	
0x013:	irmovq $5,%rsi # Return					F	D	E	M	W

W
valM = 0x013

•
•
•

F
valC ← 5
rB ← %rsi
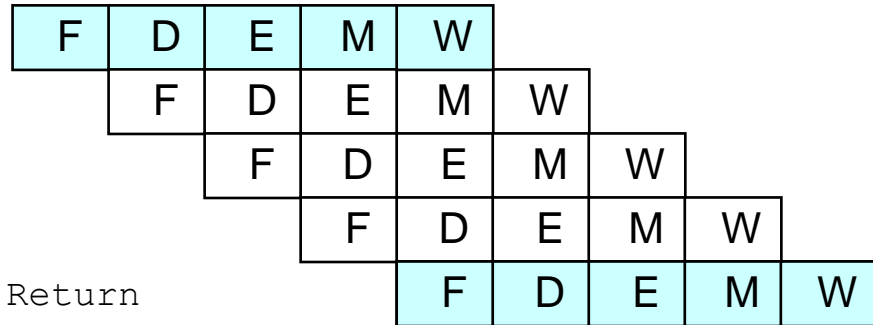
- **As `ret` passes through pipeline, stall at fetch stage**
  - **While in decode, execute, and memory stage**
- **Inject bubble into decode stage**
- **Release stall when reach write-back stage**
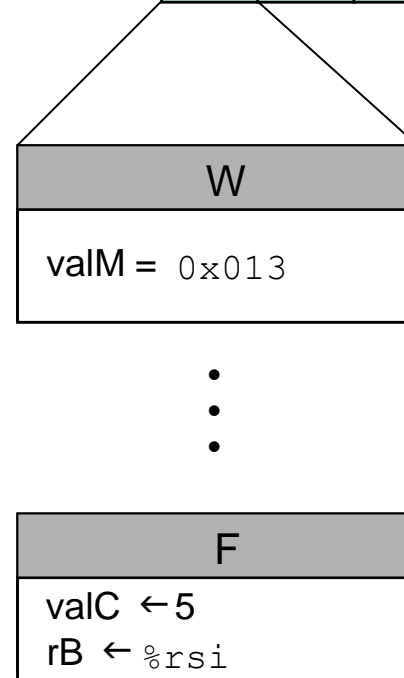
Detecting Return
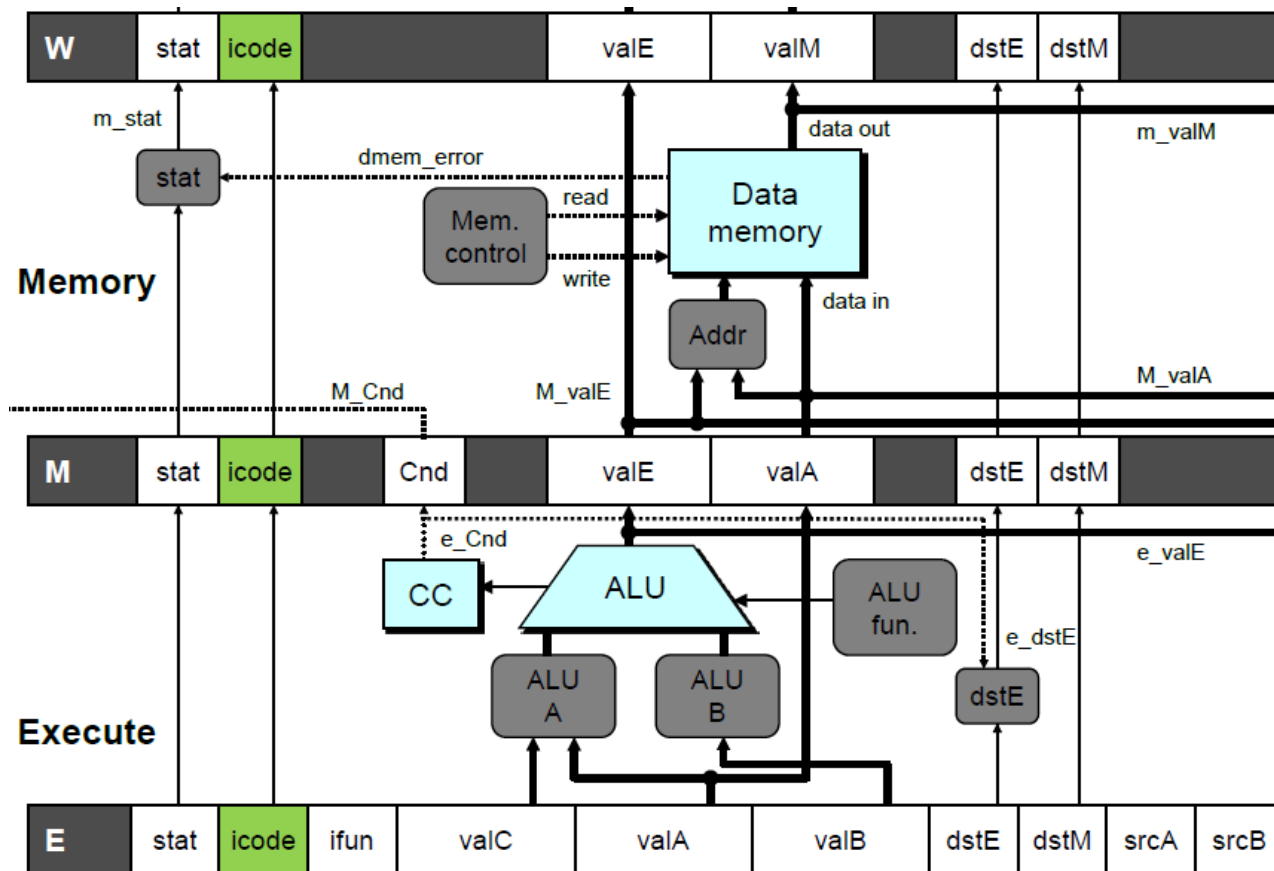
Condition	Trigger
Processing `ret`	`IRET in { D_icode, E_icode, M_icode }`

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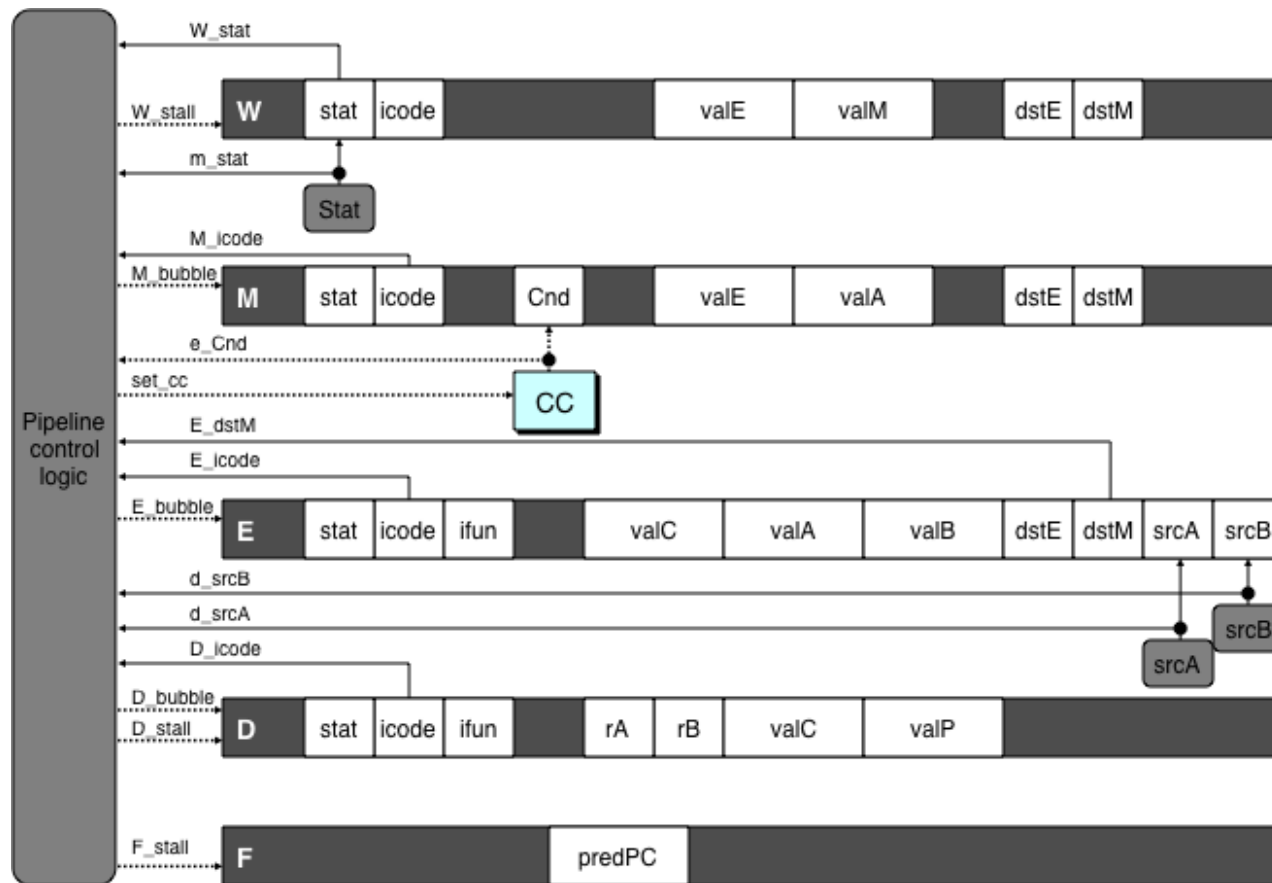
- **Combinational logic generates pipeline control signals**
- **Action occurs at start of following cycle**

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 };
```
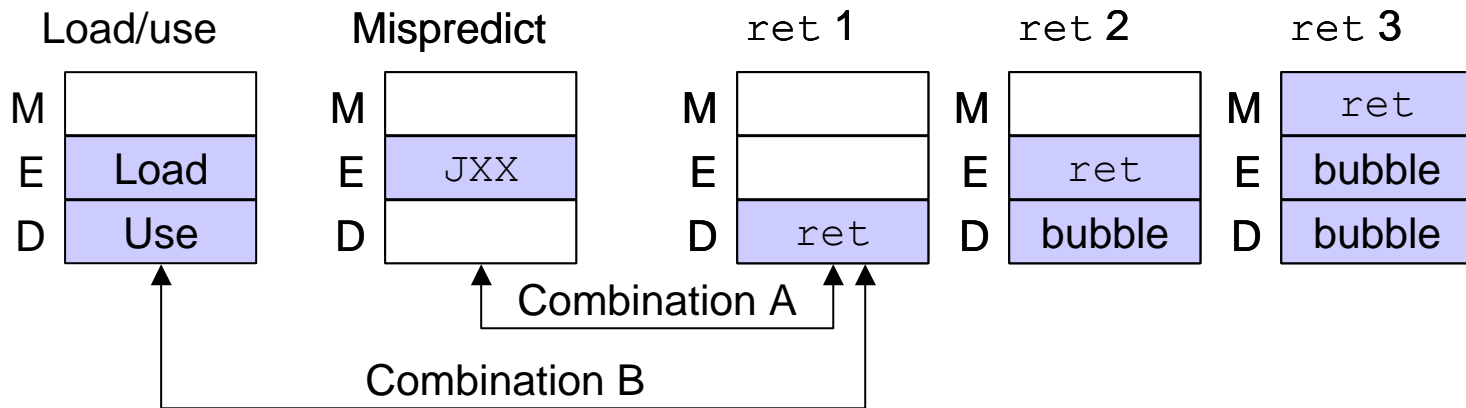
Control Combinations

Control Combinations

	Load/use		Mispredict		ret 1		ret 2		ret 3
M		M		M		M		M	ret
E	Load	E	JXX	E		E	ret	E	bubble
D	Use	D		D	ret	D	bubble	D	bubble

Combination A

Combination B

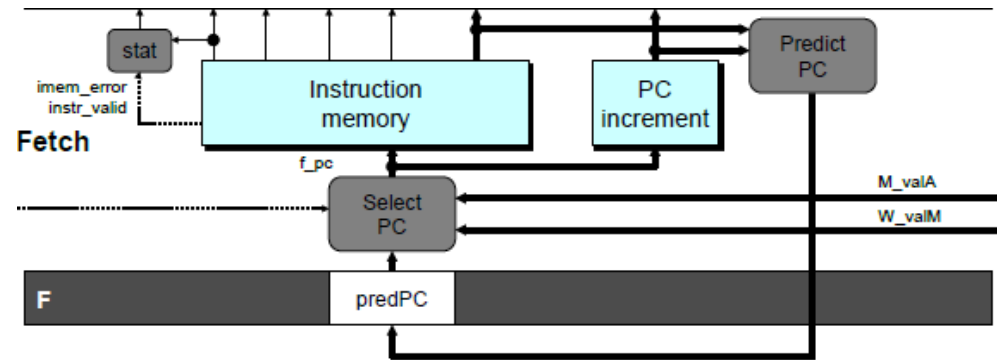- **Special cases that can arise on same clock cycle**

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

Mispredict   ret **1**

M	
E	JXX
D	

M	
E	
D	ret

Combination A

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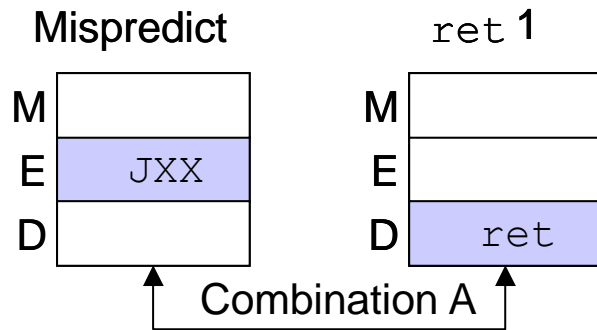
Condition	F	D	E	M	W
Processing ret	**stall**	**bubble**	**normal**	**normal**	**normal**
Mispredicted Branch	**normal**	**bubble**	**bubble**	**normal**	**normal**
Combination	*stall*	*bubble*	*bubble*	*normal*	*normal*

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

Control Combination B

Load/use

M	
E	Load
D	Use

ret **1**

M	
E	
D	ret

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

Load/use

M	
E	Load
D	Use

ret 1

M	
E	
D	ret

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 Part 1: Summary

- **Concept**
  - **Break instruction execution into 5 stages**
  - **Run instructions through in pipelined mode**

- **Limitations**
  - **Can't handle dependencies between instructions when instructions follow too closely**
  - **Data dependencies**
    - **One instruction writes register, later one reads it**
  - **Control dependency**
    - **Instruction sets PC in way that pipeline did not predict correctly**
    - **Mispredicted branch and return**

- **Fixing the Pipeline**
  - **We'll do that next time**

Pipeline Part 2: 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