

# **CSC 252: Computer Organization**

## **Spring 2024: Lecture 15**

Instructor: Yuhao Zhu

Department of Computer Science  
University of Rochester

# Another Way to Look At the Microarchitecture

## Principles:

- Execute each instruction one at a time, one after another
- Express every instruction as series of **simple steps**
- Dedicated hardware structure for completing each step
- Follow same general flow for each instruction type

**Fetch:** Read instruction from instruction memory

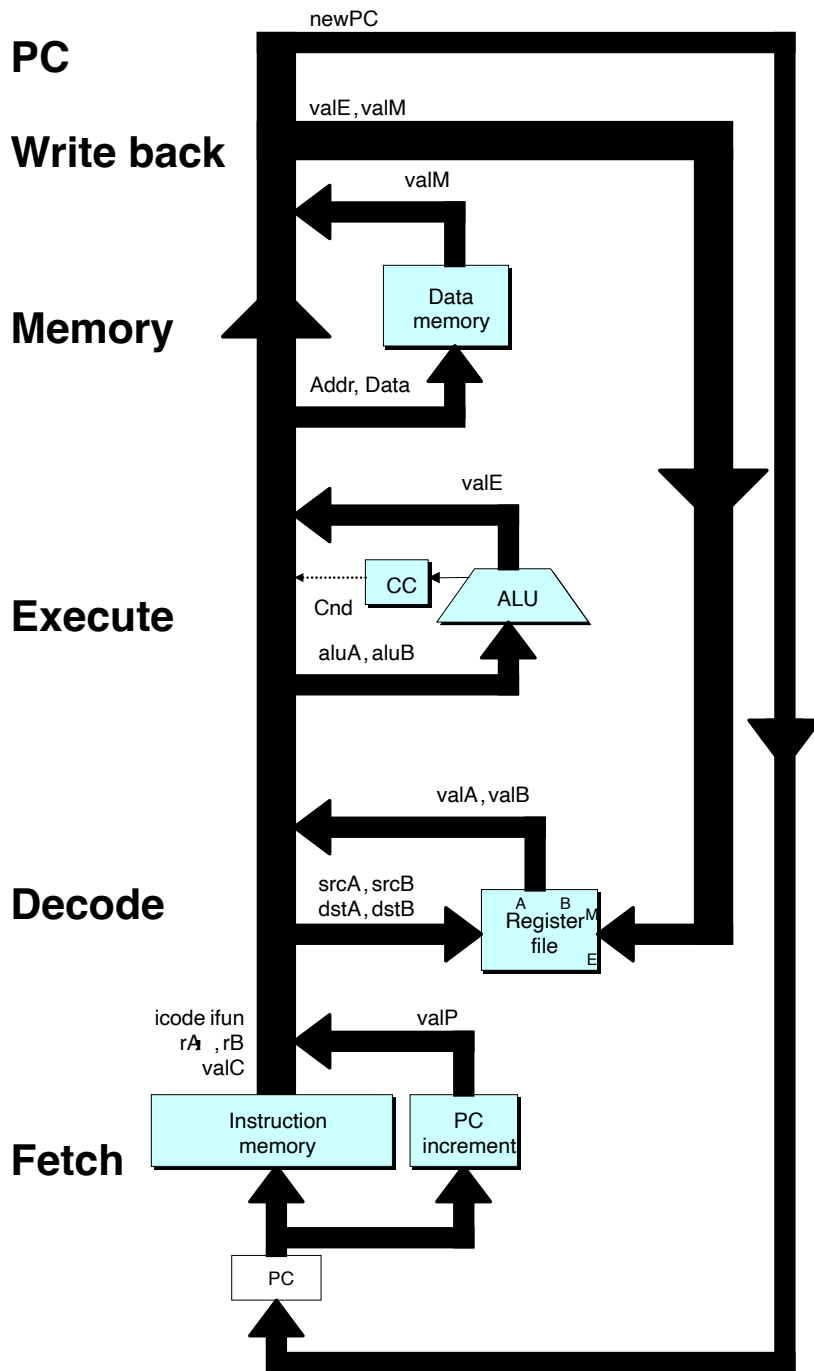
**Decode:** Read program registers

**Execute:** Compute value or address

**Memory:** Read or write data

**Write Back:** Write program registers

**PC:** Update program counter



## Fetch

- Read instruction from instruction memory

## Decode

- Read program registers

## Execute

- Compute value or address

## Memory

- Read or write data

## Write Back

- Write program registers

## PC

- Update program counter

# Stage Computation: Arith/Log. Ops



	OPq rA, rB	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valP \leftarrow PC+2$	Read instruction byte Read register byte  Compute next PC
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	Read operand A Read operand B
Execute	$valE \leftarrow valB \ OP \ valA$ Set CC	Perform ALU operation Set condition code register
Memory		
Write back	$R[rB] \leftarrow valE$	Write back result
PC update	$PC \leftarrow valP$	Update PC

# Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



	<code>rmmovq rA, D(rB)</code>	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valC} \leftarrow M_8[\text{PC}+2]$ $\text{valP} \leftarrow \text{PC}+10$	Read instruction byte Read register byte Read displacement D Compute next PC
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	Read operand A Read operand B
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Compute effective address
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	Write value to memory
Write back		
PC update	$\text{PC} \leftarrow \text{valP}$	Update PC

# Stage Computation: Jumps

	jXX Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Read instruction byte
	$\text{valC} \leftarrow M_8[\text{PC}+1]$	Read destination address
	$\text{valP} \leftarrow \text{PC}+9$	Fall through address
Decode		
Execute	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	Take branch?
Memory		
Write back		
PC update	$\text{PC} \leftarrow \text{Cnd} ? \text{valC} : \text{valP}$	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

# Pipeline Stages

## Fetch

- Select current PC
- Read instruction
- Compute incremented PC

## Decode

- Read program registers

## Execute

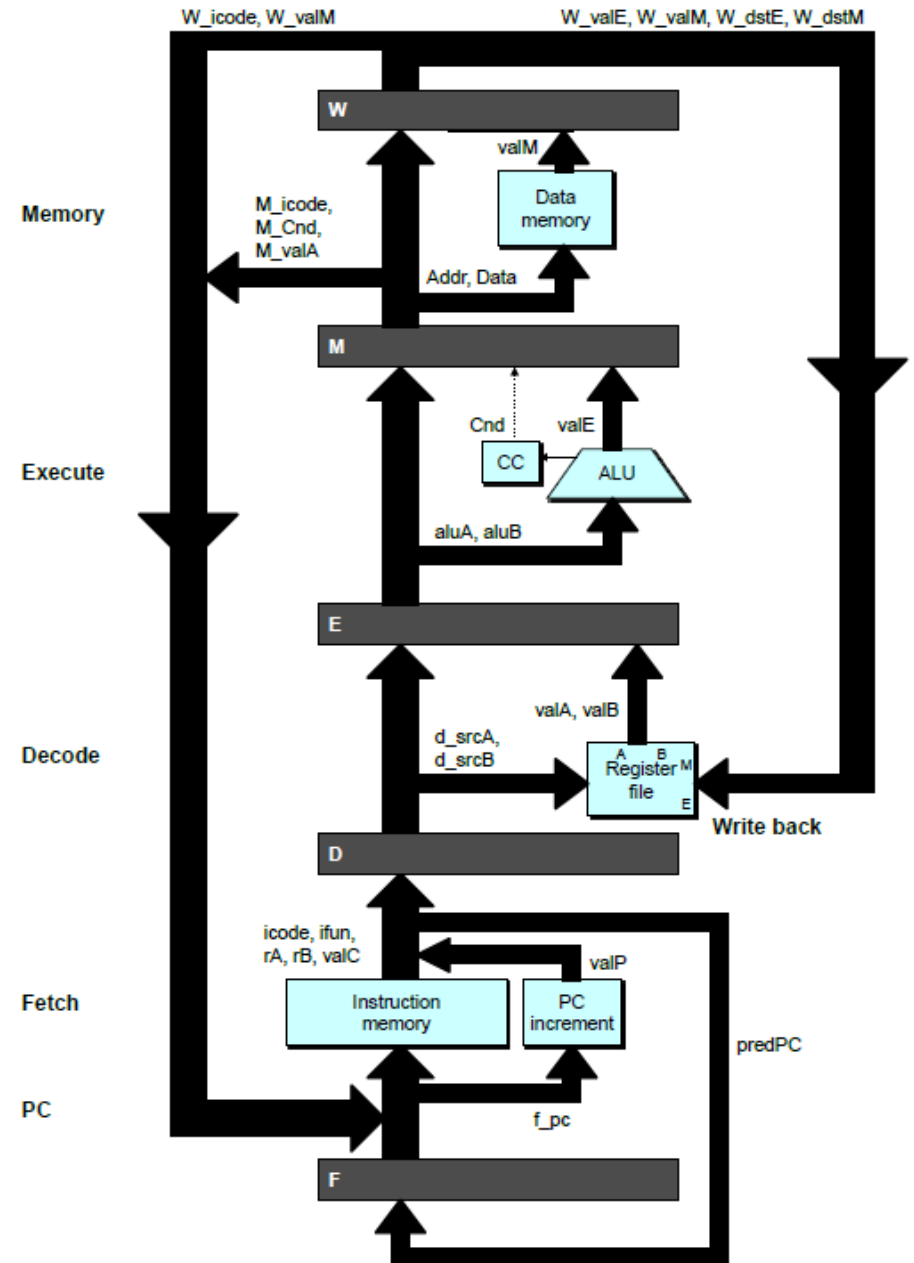
- Operate ALU

## Memory

- Read or write data memory

## Write Back

- Update register file



# Real-World Pipelines: Car Washes

**Sequential**



**Pipelined**



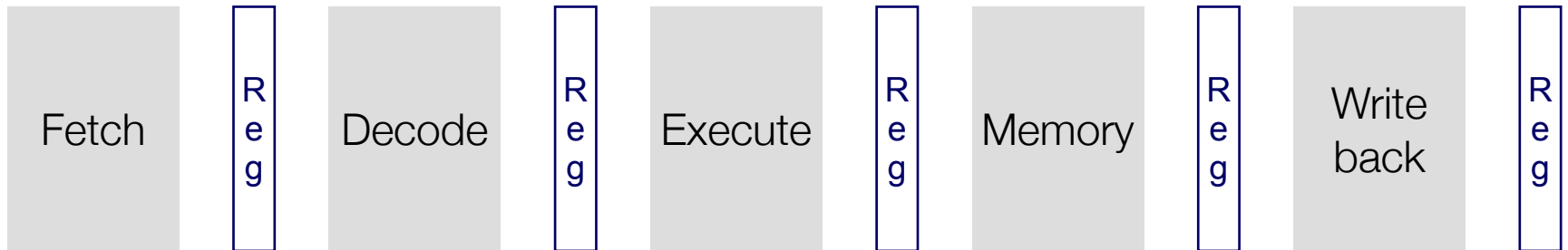
## Idea

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

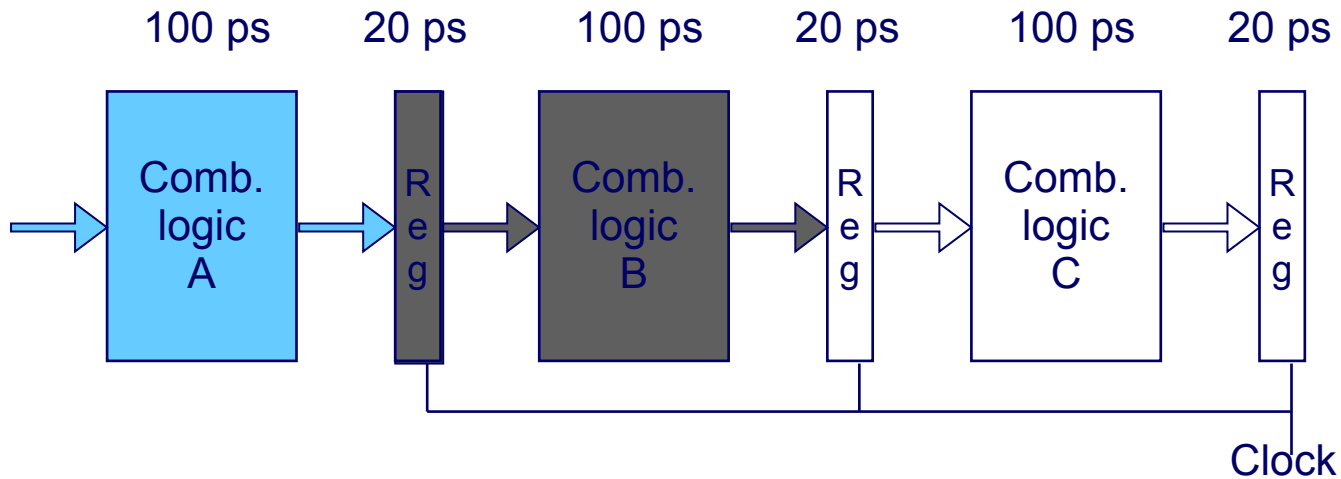
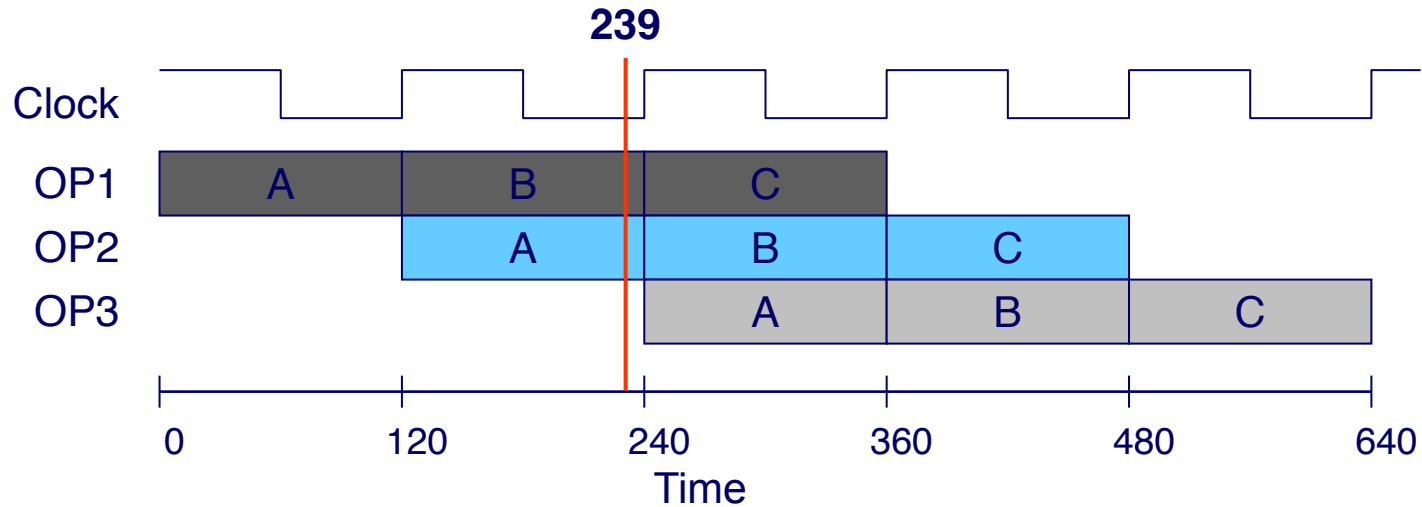


# Pipeline Illustration

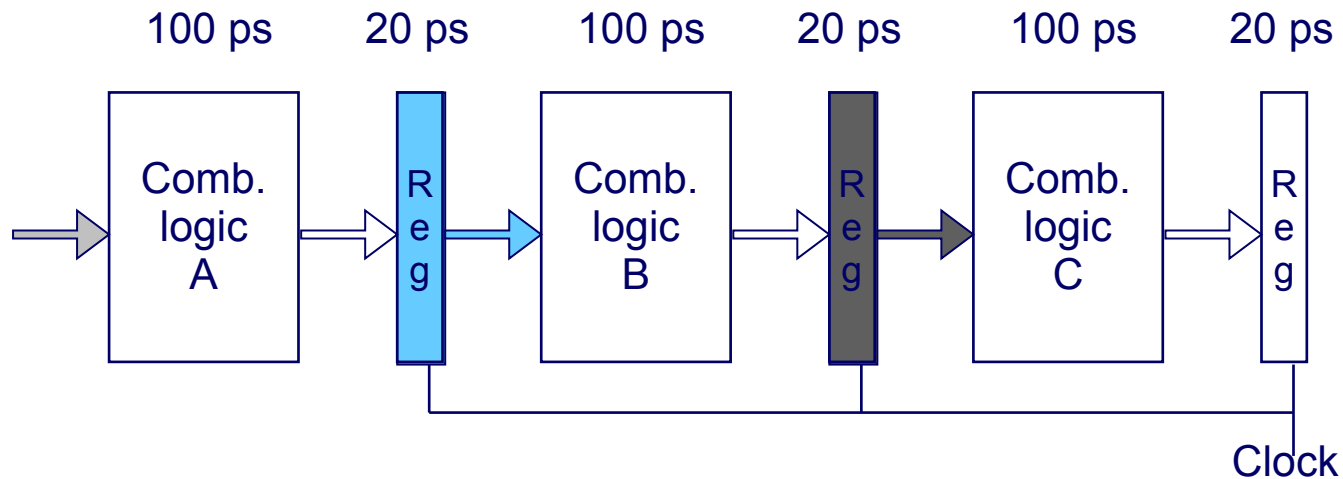
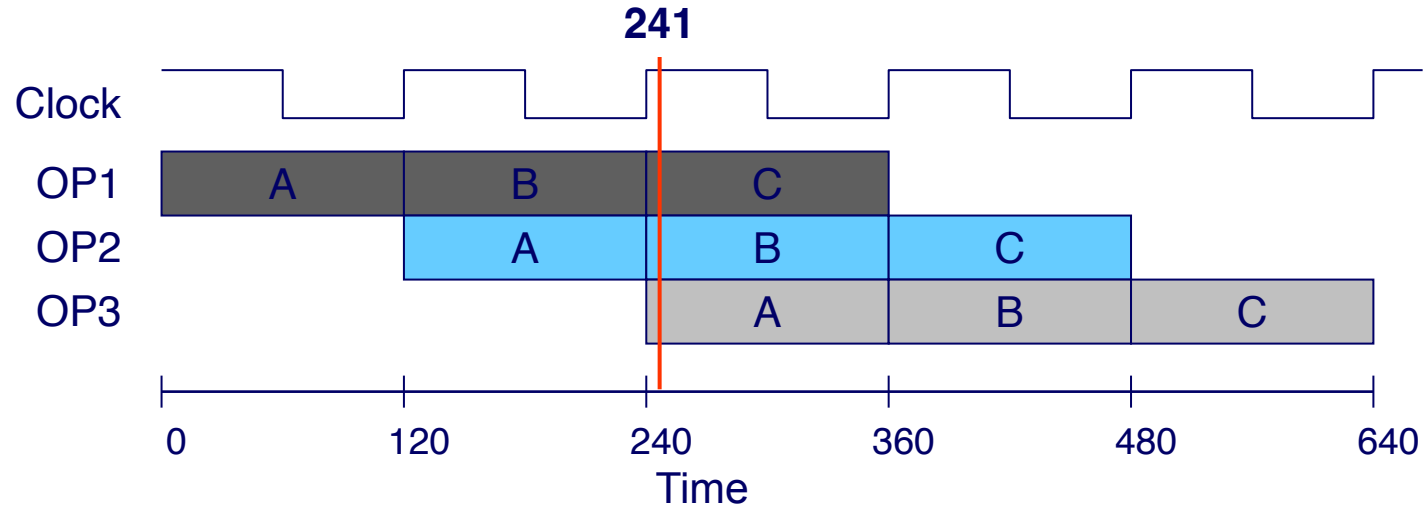
**Inst0**



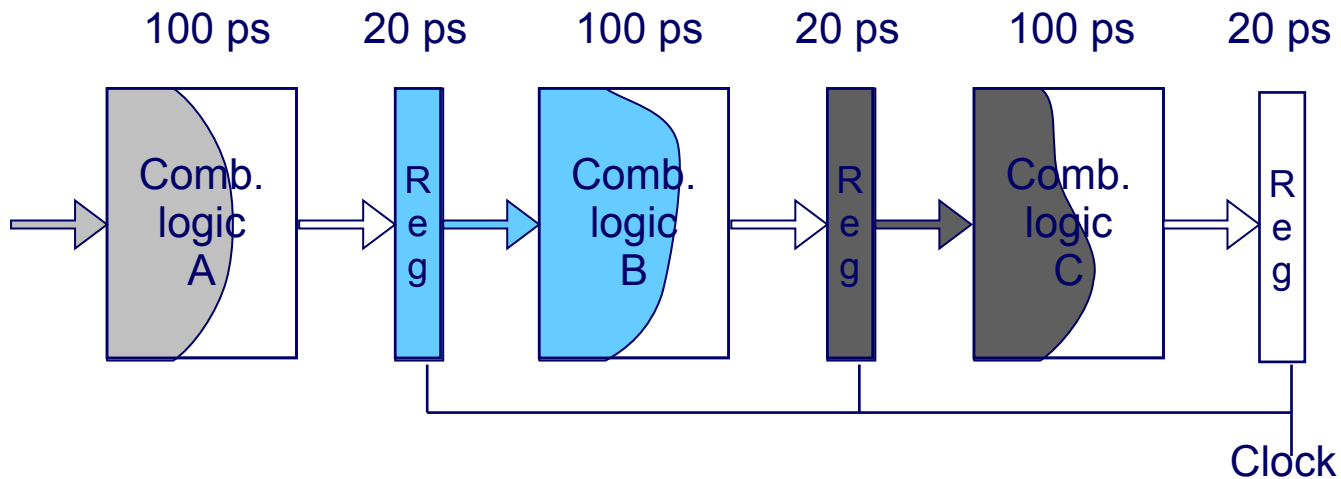
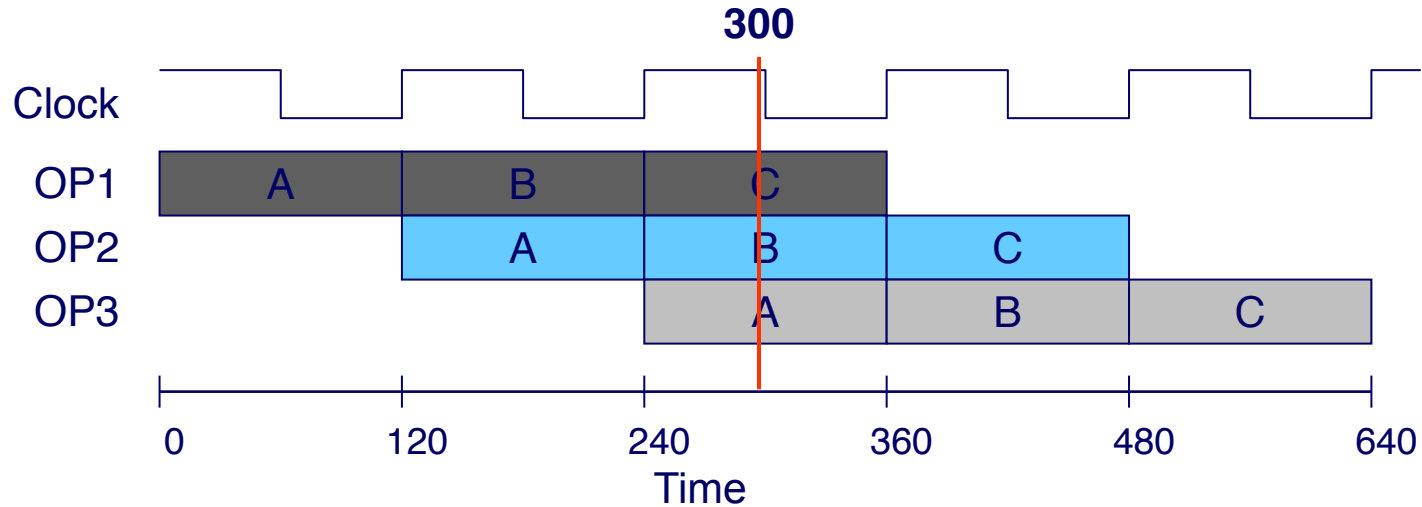
# Another Illustration



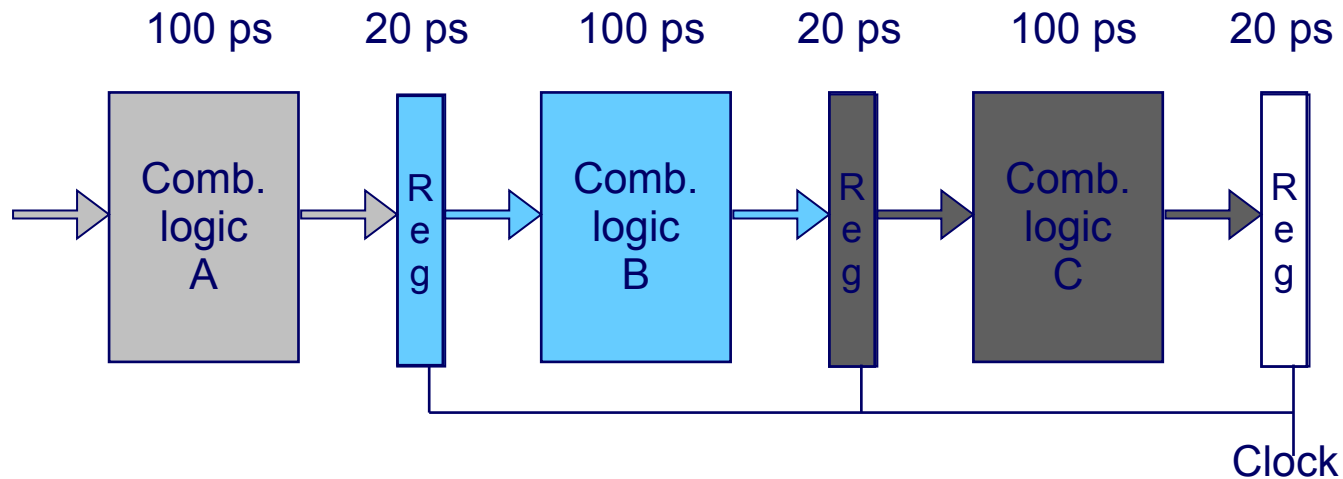
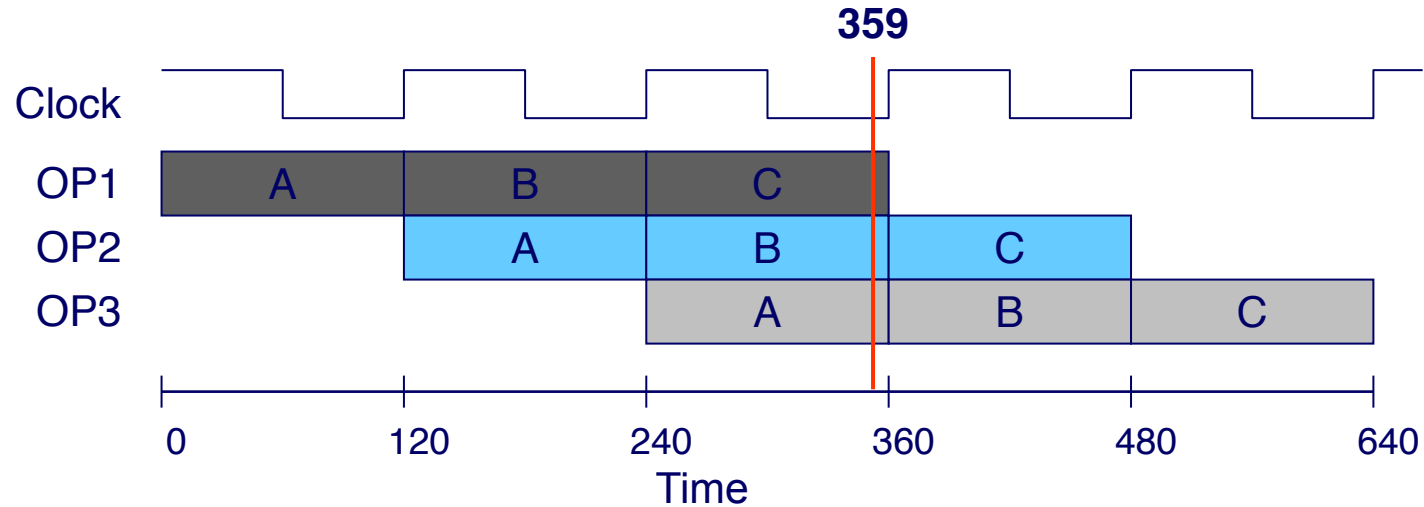
# Another Illustration



# Another Illustration



# Another Illustration

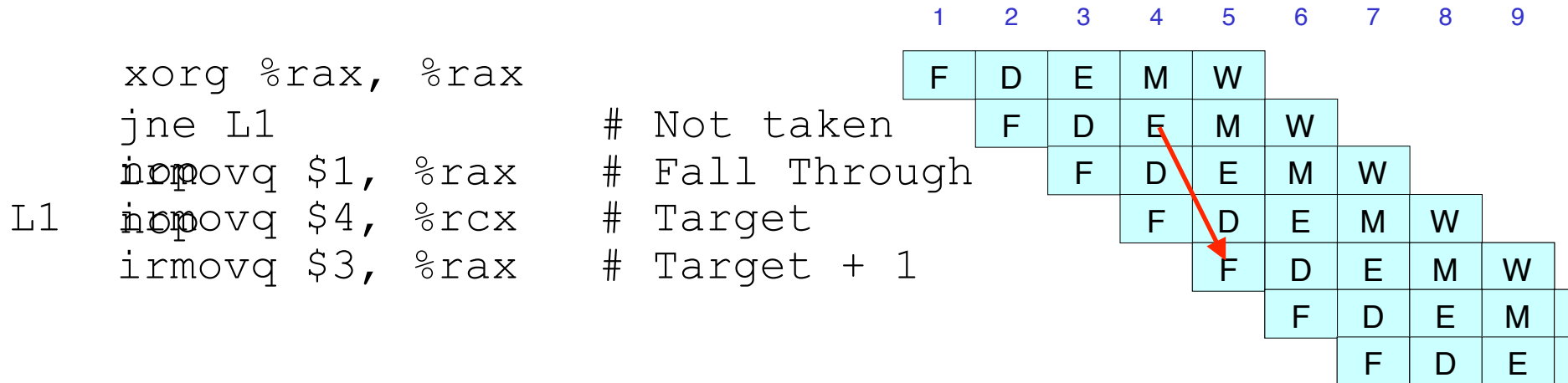


# Making the Pipeline Really Work

- Control Dependencies
  - What is it?
  - Software mitigation: Inserting Nops
  - Software mitigation: Delay Slots
- Data Dependencies
  - What is it?
  - Software mitigation: Inserting Nops

# Control Dependency

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage



# Delay Slots

```
xorg %rax, %rax
jne L1
nop
nop
irmovq $1, %rax    # Fall Through
L1: irmovq $4, %rcx  # Target
    irmovq $3, %rax  # Target + 1
```

Can we make use of the 2 wasted slots?

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
					F	D	E	M
						F	D	E

Have to make sure `do_C` doesn't  
depend on `do_A` and `do_B`!!!

```
if (cond) {
    do_A();
} else {
    do_B();
}
do_C();
```



# Delay Slots

```

    xorg %rax, %rax
    jne L1
    nop
    nop
    irmovq $1, %rax    # Fall Through
L1:  irmovq $4, %rcx    # Target
     irmovq $3, %rax    # Target + 1

```

Can we make use of the 2 wasted slots?

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
					F	D	E	M
						F	D	E

A less obvious example

```

do_C();
if (cond) {
    do_A();
} else {
    do_B();
}

```

add A, B	add A, B
<b>or C, D</b>	sub E, F
sub E, F	jle 0x200
jle 0x200	<b>or C, D</b>
add A, C	add A, C

Why don't we move the sub instruction?

# Resolving Control Dependencies

- **Software Mechanisms**

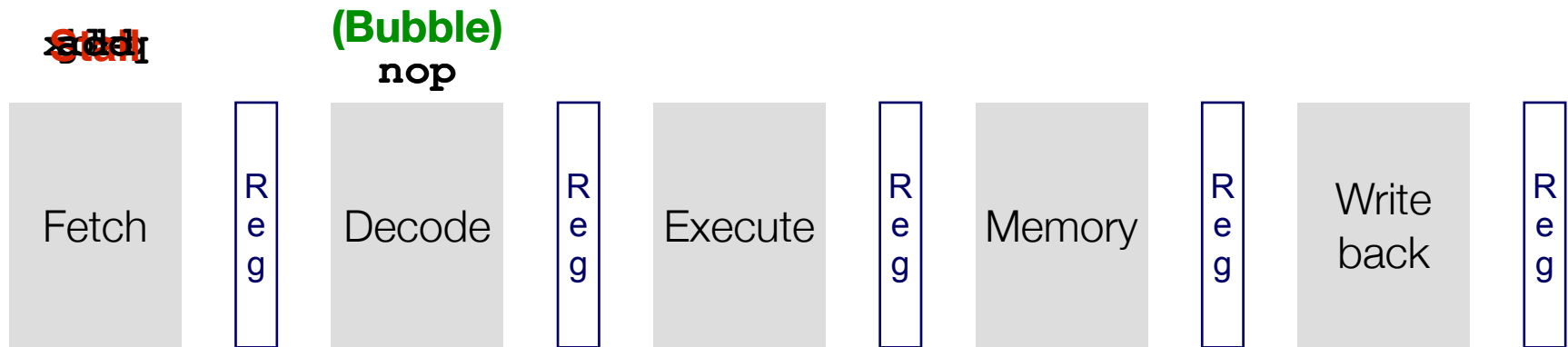
- Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
- Delay slot: insert instructions that do not depend on the effect of the preceding instruction. These instructions will execute even if the preceding branch is taken — old RISC approach

- **Hardware mechanisms**

- Stalling (Think of it as hardware automatically inserting nops)
- Branch Prediction
- Return Address Stack

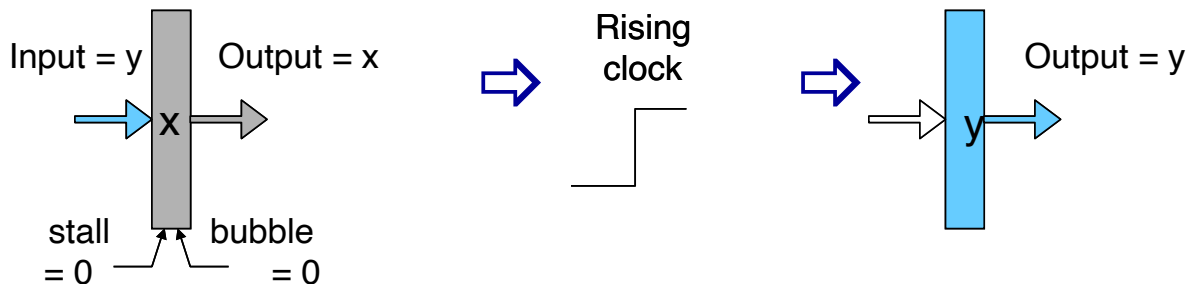
# Hardware Generated Nops (Bubble and Stalling)

- **Stall**: the pipeline register shouldn't be written
- **Bubble**: signals correspond to a nop
- Why is it good for the hardware to do so anyways?

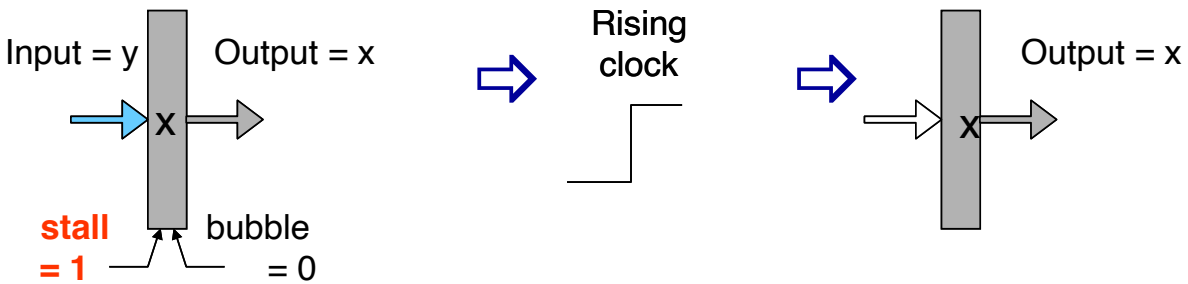


# How are Stall and Bubble Implemented in Hardware?

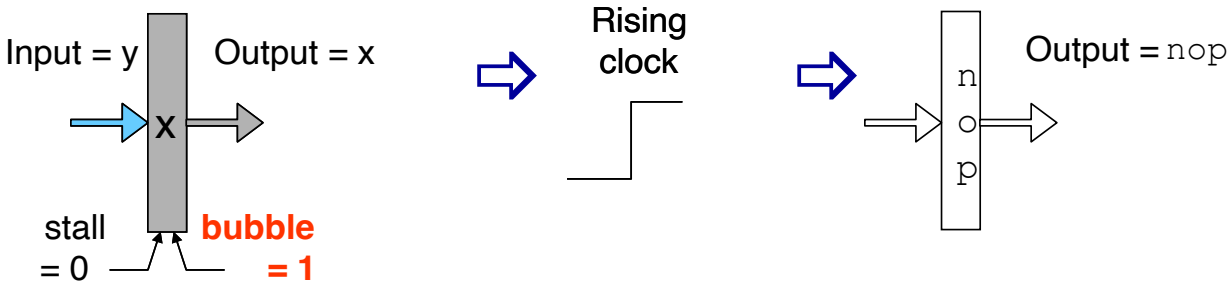
Normal



Stall



Bubble



# Branch Prediction

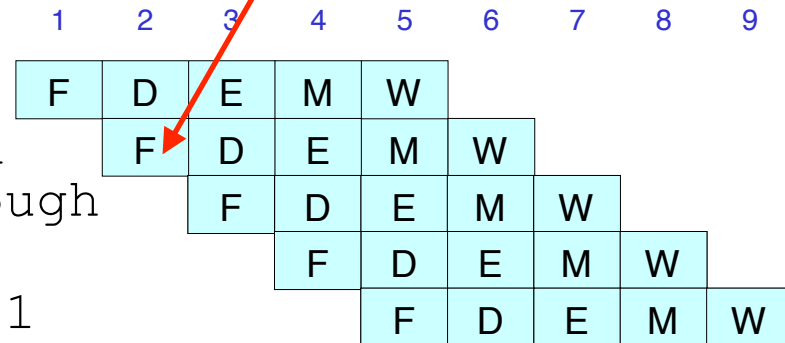
Idea: instead of waiting, why not just guess the direction of jump?

If prediction is correct: pipeline moves forward without stalling

If mispredicted: kill mis-executed instructions, start from the correct target

```
L1  xorg %rax, %rax
    jne L1          # Not taken
    irmovq $1, %rax # Fall Through
    irmovq $4, %rcx # Target
    irmovq $3, %rax # Target + 1
```

Also takes a guess of  
the jump direction



# Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

If prediction is correct: pipeline moves forward without stalling

If mispredicted: kill mis-executed instructions, start from the correct target

## Static Prediction

- Always Taken
- Always Not-taken

## Dynamic Prediction

- Dynamically predict taken/not-taken for each specific jump instruction

# Static Prediction


## Observation (Assumption really): Two uses of jumps


- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

## Strategy:

- Forward jumps (i.e., `if-else`): always predict not-taken
- Backward jumps (i.e., `loop`): always predict taken

```
cmpq    %rsi, %rdi
jle     .corner_case    <before>
<do_A>
.corner_case:          .L1: <body>
<do_B>                cmpq B, A
ret                   jl  .L1
                        <after>
```

 **Mostly not taken**

 **Mostly taken**

# Static Prediction

Knowing branch prediction strategy helps us write faster code

- Any difference between the following two code snippets?
- What if you know that hardware uses the always non-taken branch prediction?

```
if (cond) {  
    do_A()  
} else {  
    do_B()  
}
```

```
if (!cond) {  
    do_B()  
} else {  
    do_A()  
}
```




# Dynamic Prediction

- Simplest idea:
  - If last time taken, predict taken; if last time not-taken, predict not-taken
  - Called 1-bit branch predictor
  - Works nicely for loops

```
for (i=0; i <5; i++) {...}
```

Iteration #1	0	1	2	3	4
Predicted Outcome	N	T	T	T	T
Actual Outcome	T	T	T	T	N



# Dynamic Prediction

- With 1-bit prediction, we change our mind instantly if mispredict
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

```
for (i=0; i <5; i++) {...}
```

Predict with 1-bit	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T
Actual Outcome	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N
Predict with 2-bit	N	N	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

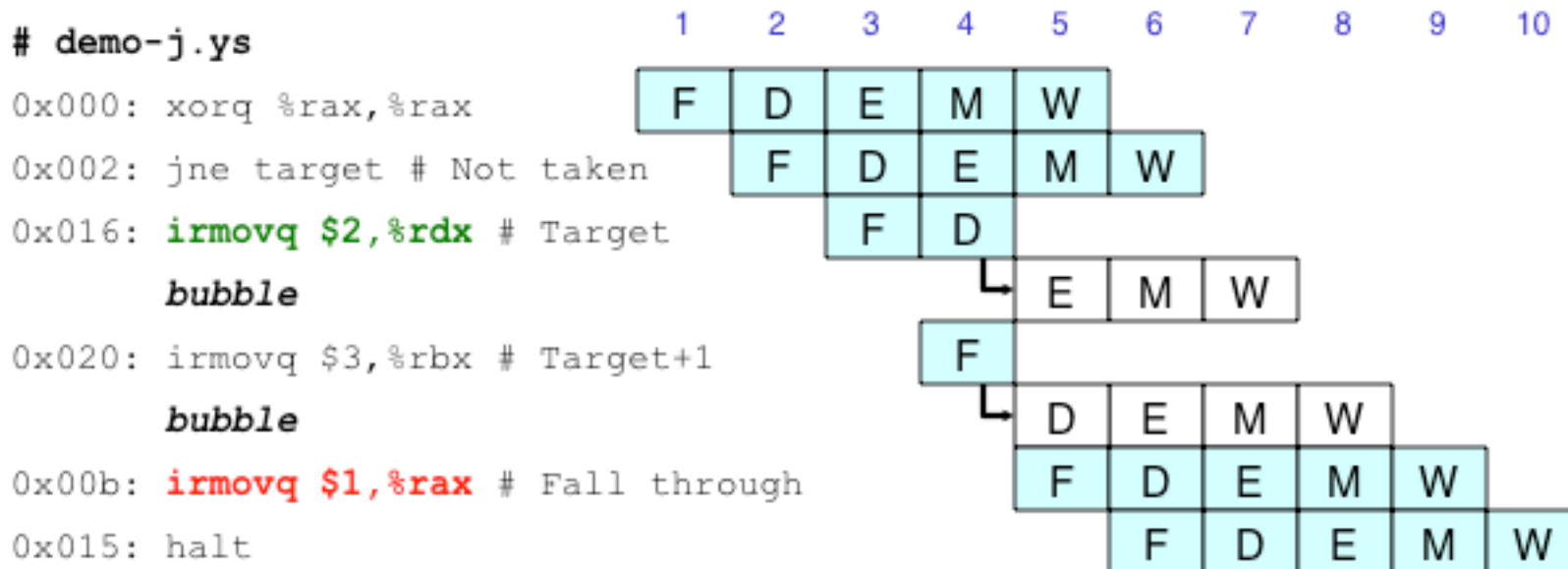
# More Advanced Dynamic Prediction

- Look for past histories *across instructions*
- Branches are often correlated
  - Direction of one branch determines another

cond1 branch not-  
taken means  $(x \leq 0)$   
branch taken

```
x = 0
if (cond1) x = 3
if (cond2) y = 19
if (x <= 0) z = 13
```

# What Happens If We Mispredict?



## Cancel instructions when mispredicted

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

# Return Instruction

```
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
```

# Stalling for Return

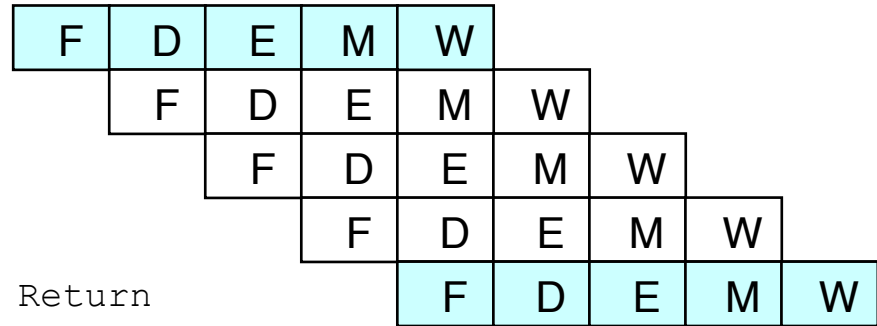
0x026:     ret

      nop

      nop

      nop

0x013:     irmovq \$5,%rsi   # Return

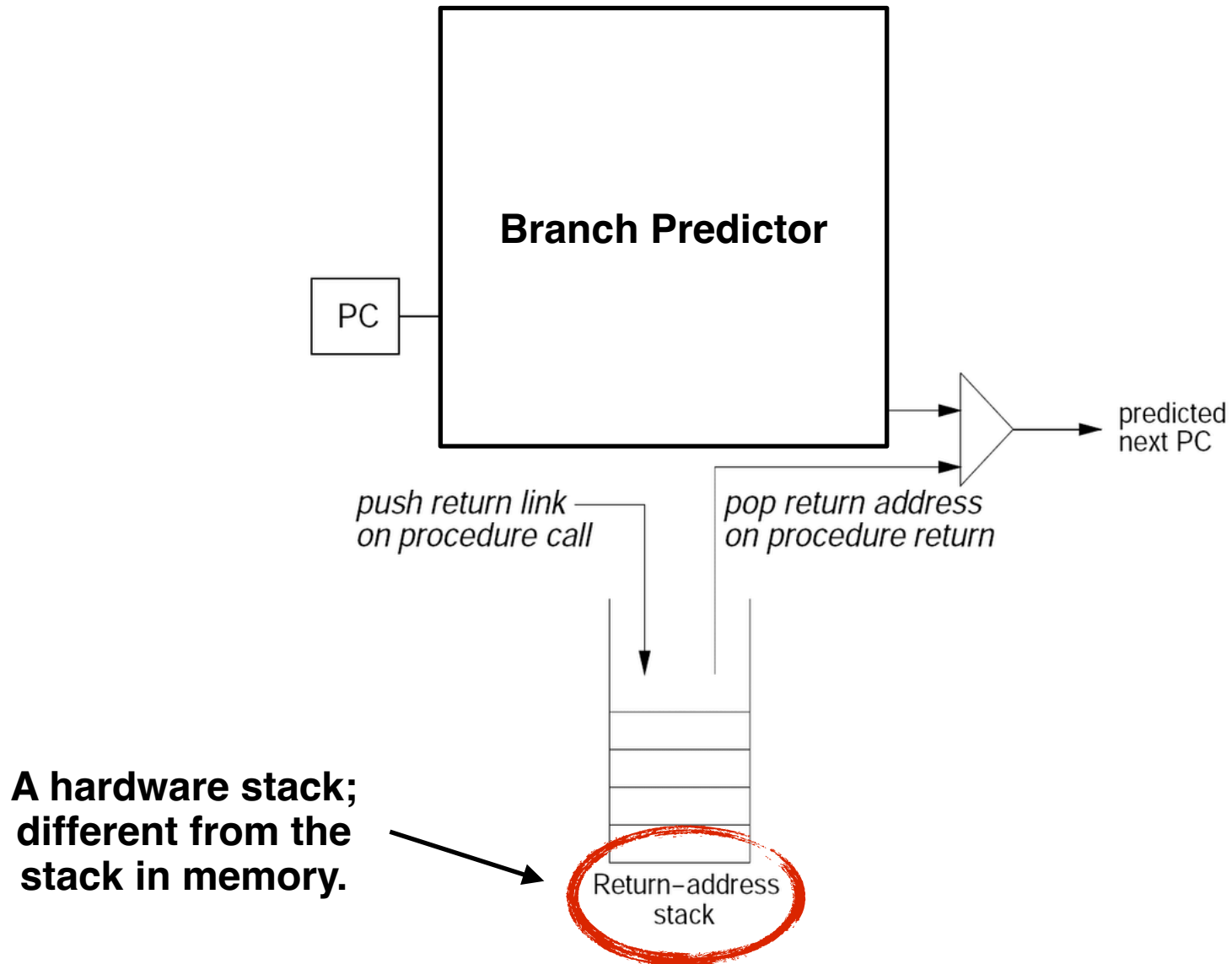


- **As ret passes through pipeline, stall at fetch stage**
  - While in decode, execute, and memory stage
- **Inject bubble into decode stage**
- **Release stall when reach write-back stage**

# Return Address Stack (RAS)

- Stalling for return is silly since we know where exactly we need to jump to, except the jump target is retrieved later in the memory stage.
- Can we get that sooner? Where should we get it?

# Return Address Stack (RAS)





# Today: Making the Pipeline Really Work

- Control Dependencies

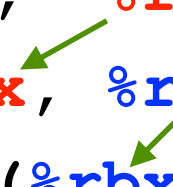
- Inserting Nops
- Stalling
- Delay Slots
- Branch Prediction

- Data Dependencies

- Inserting Nops
- Stalling
- Out-of-order execution

# Data Dependencies

```
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

# A Subtle Data Dependency

- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome **until after its Execute stage**.

## Why?

- There is a data dependency between `xorg` and `jne`. The “data” is the status flags.

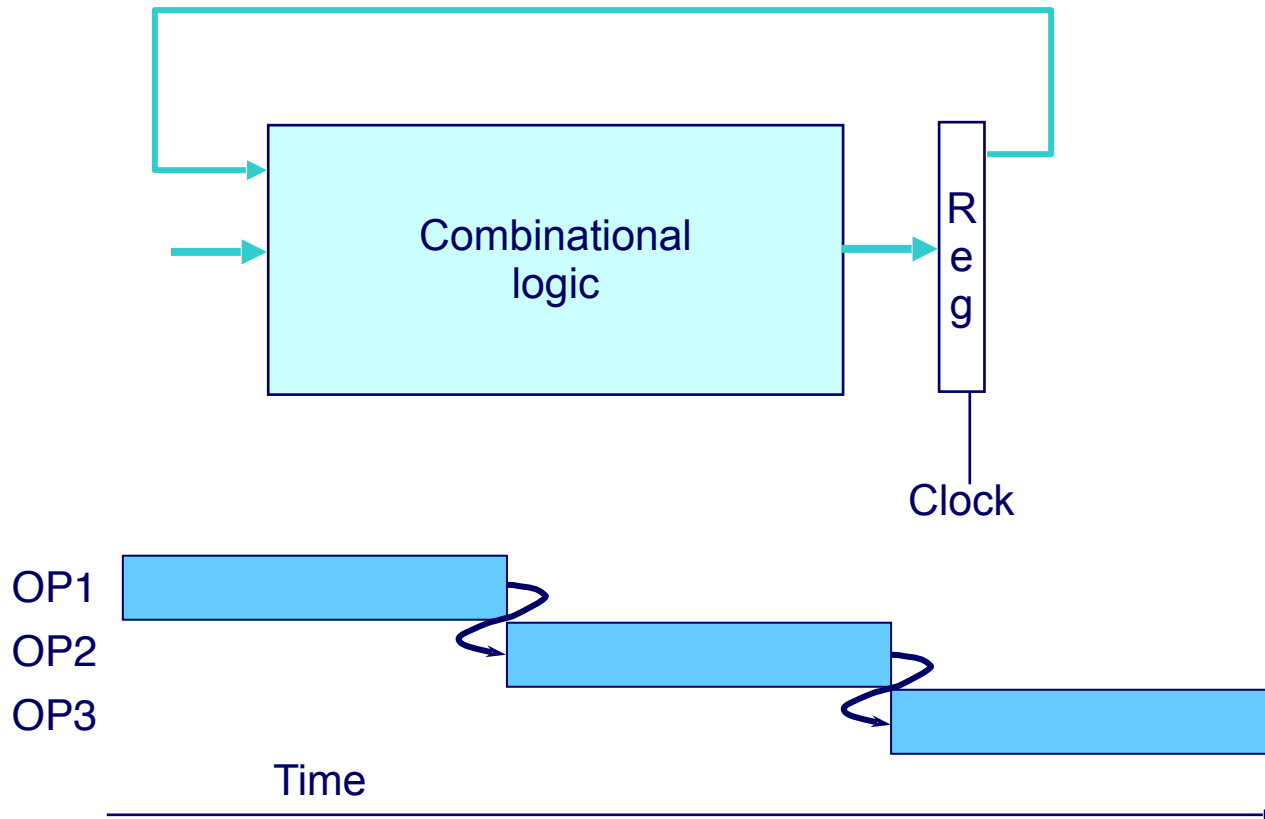
```

graph TD
    L1((L1)) --> Node1
    subgraph LoopBody
        Node1["xorg %rax, %rax"] --> Node2["jne L1"]
        Node2 --> Node3["nop"]
        Node3 --> Node4["nop"]
    end
    Node4 --> Node5["irmovq $1, %rax"]
    Node5 --> Node6["irmovq $4, %rcx"]
    Node6 --> Node7["irmovq $3, %rax"]
    Node7 --> Node8["Fall Through"]
    Node2 --> L1

```

Control flow graph showing the execution flow of the assembly code. The graph is annotated with the number of times each instruction is executed (1-9).

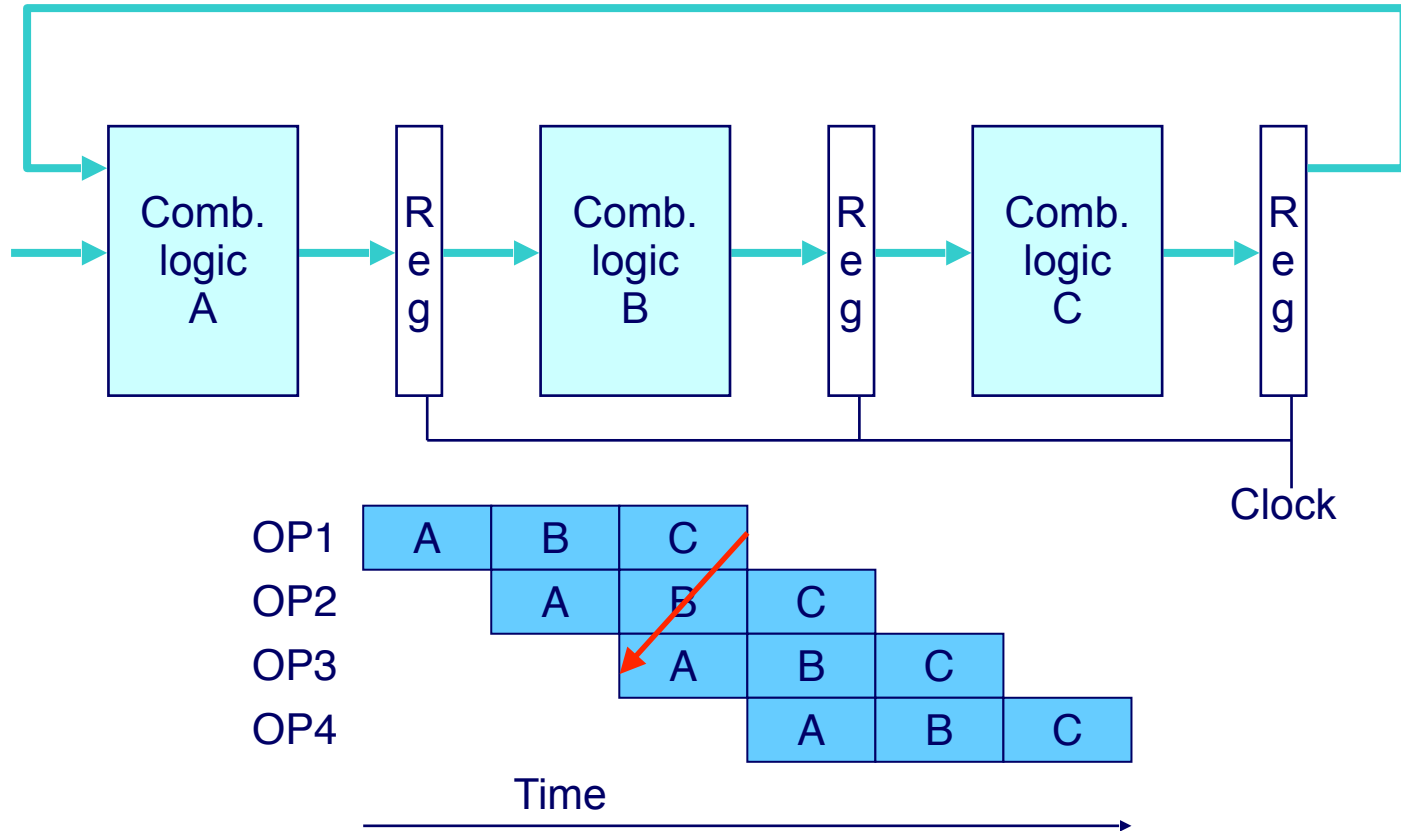
# Data Dependencies in Single-Cycle Machines



## In Single-Cycle Implementation:

- Each operation starts only after the previous operation finishes. Dependency always satisfied.

# Data Dependencies in Pipeline Machines



Data Hazards happen when:

- Result does not feed back around in time for next operation

# Data Dependencies: No Nop

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

Remember registers get  
updated in the Write-back stage

**addq reads wrong %rdx and %rax**

# Data Dependencies: 1 Nop

0x000: `irmovq $10,%rdx`

0x00a: `irmovq $3,%rax`

0x014: `nop`

0x015: `addq %rdx,%rax`

0x017: `halt`

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

**addq still reads wrong %rdx and %rax**

# Data Dependencies: 2 Nop's

	1	2	3	4	5	6	7	8	9	10
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		
0x016: addq %rdx,%rax					F	D	E	M	W	
0x018: halt						F	D	E	M	W

**addq reads the correct %rdx,  
but %rax still wrong**



# Data Dependencies: 3 Nop's

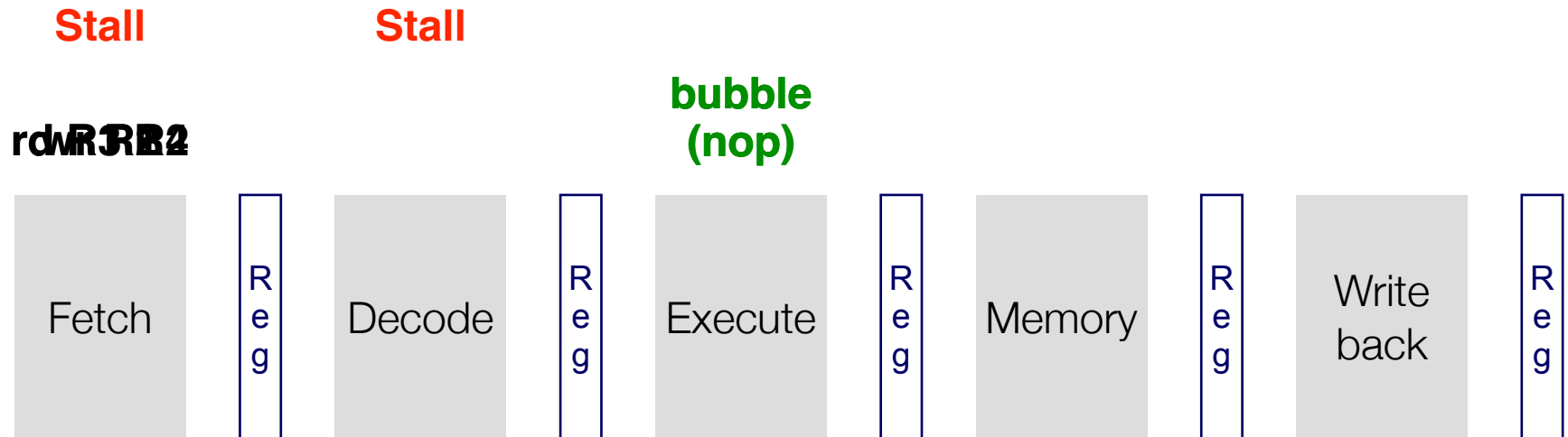
	1	2	3	4	5	6	7	8	9	10	11
0x000: irmovq \$10,%rdx	F	D	E	M	W						
0x00a: irmovq \$3,%rax		F	D	E	M	W					
0x014: nop			F	D	E	M	W				
0x015: nop				F	D	E	M	W			
0x016: nop					F	D	E	M	W		
0x017: addq %rdx,%rax						F	D	E	M	W	
0x019: halt							F	D	E	M	W

**addq reads the correct %rdx  
and %rax**

# Resolving Data Dependencies

- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
- Hardware mechanisms
  - Stalling
  - Forwarding
  - Out-of-order execution

# Hardware Generated Nops (Bubble and Stalling)



# Detecting Stall Condition

- Using a “**scoreboard**”. Each register has a bit.
- Every instruction that writes to a register sets the bit.
- Every instruction that reads a register would have to check the bit first.
  - If the bit is set, then generate a bubble
  - Otherwise, free to go!!

# Detecting Stall Condition

```
0x000: irmovq $10,%rdx
```

```
0x00a: irmovq $3,%rax
```

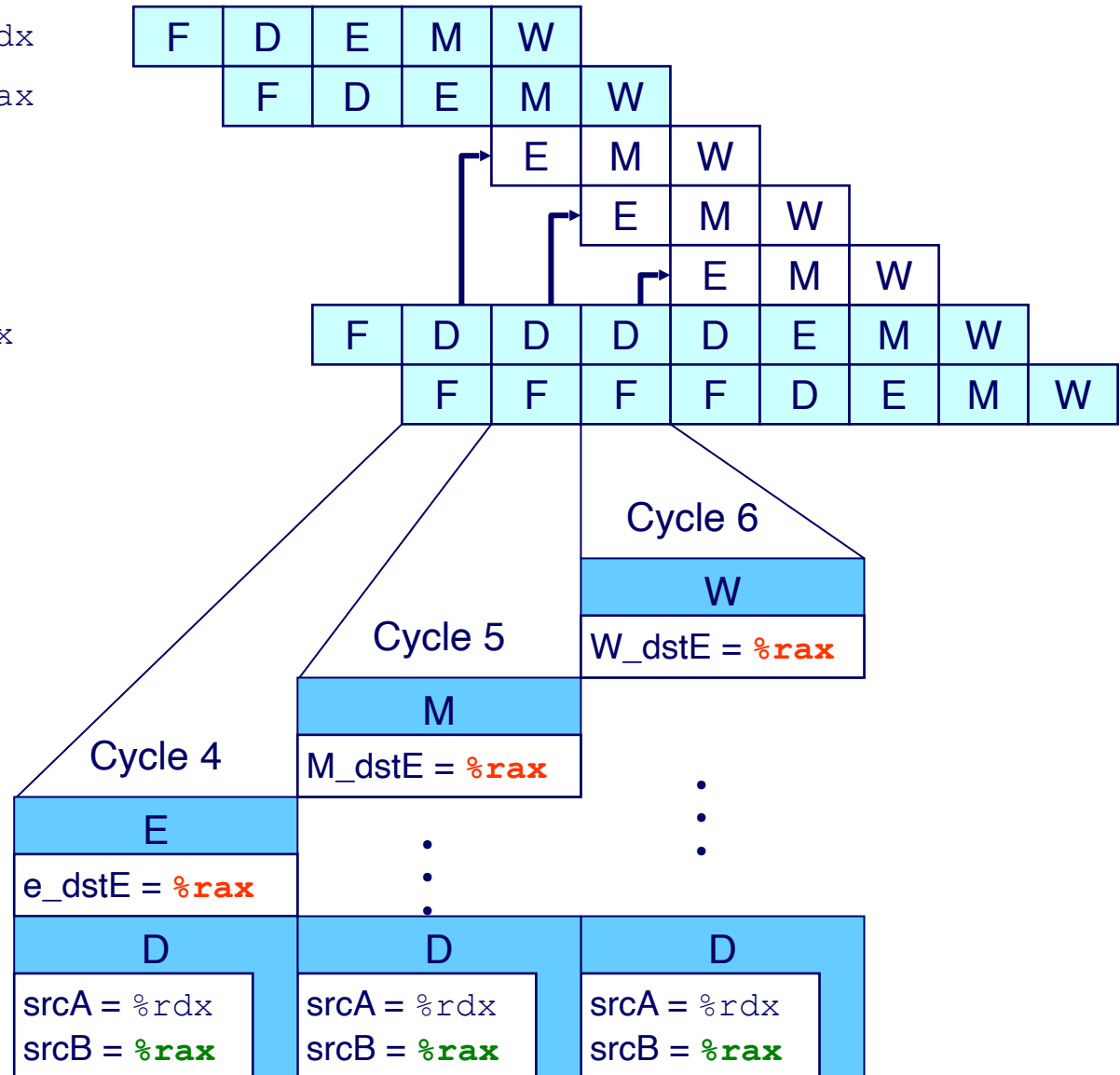
***bubble***

***bubble***

***bubble***

```
0x014: addq %rdx,%rax
```

```
0x016: halt
```



# Data Forwarding

## Naïve Pipeline

- Register isn't written until completion of write-back stage
- Source operands read from register file in decode stage
- The decode stage can't start until the write-back stage finishes

## Observation

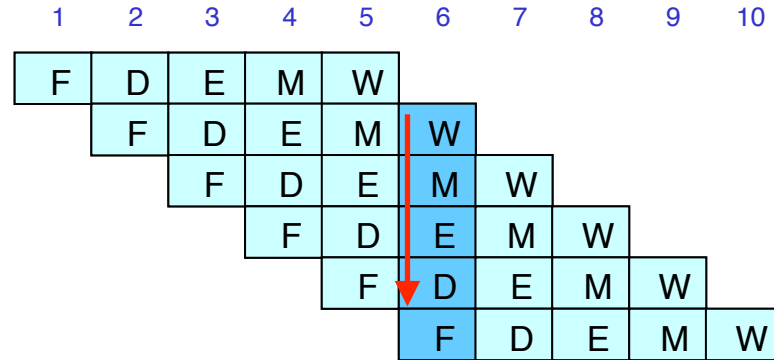
- Value generated in execute or memory stage

## Trick

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

# Data Forwarding Example

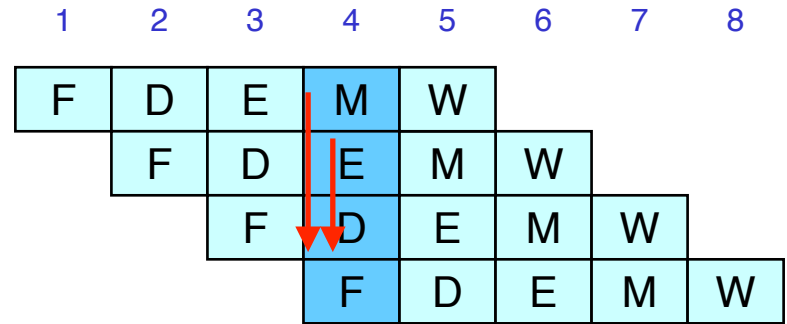
```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



- `irmovq` writes `%rax` to the register file at the end of the write-back stage
- But the value of `%rax` is already available at the beginning of the write-back stage
- Forward `%rax` to the decode stage of `addq`.

# Data Forwarding Example #2

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



Register `%rdx`

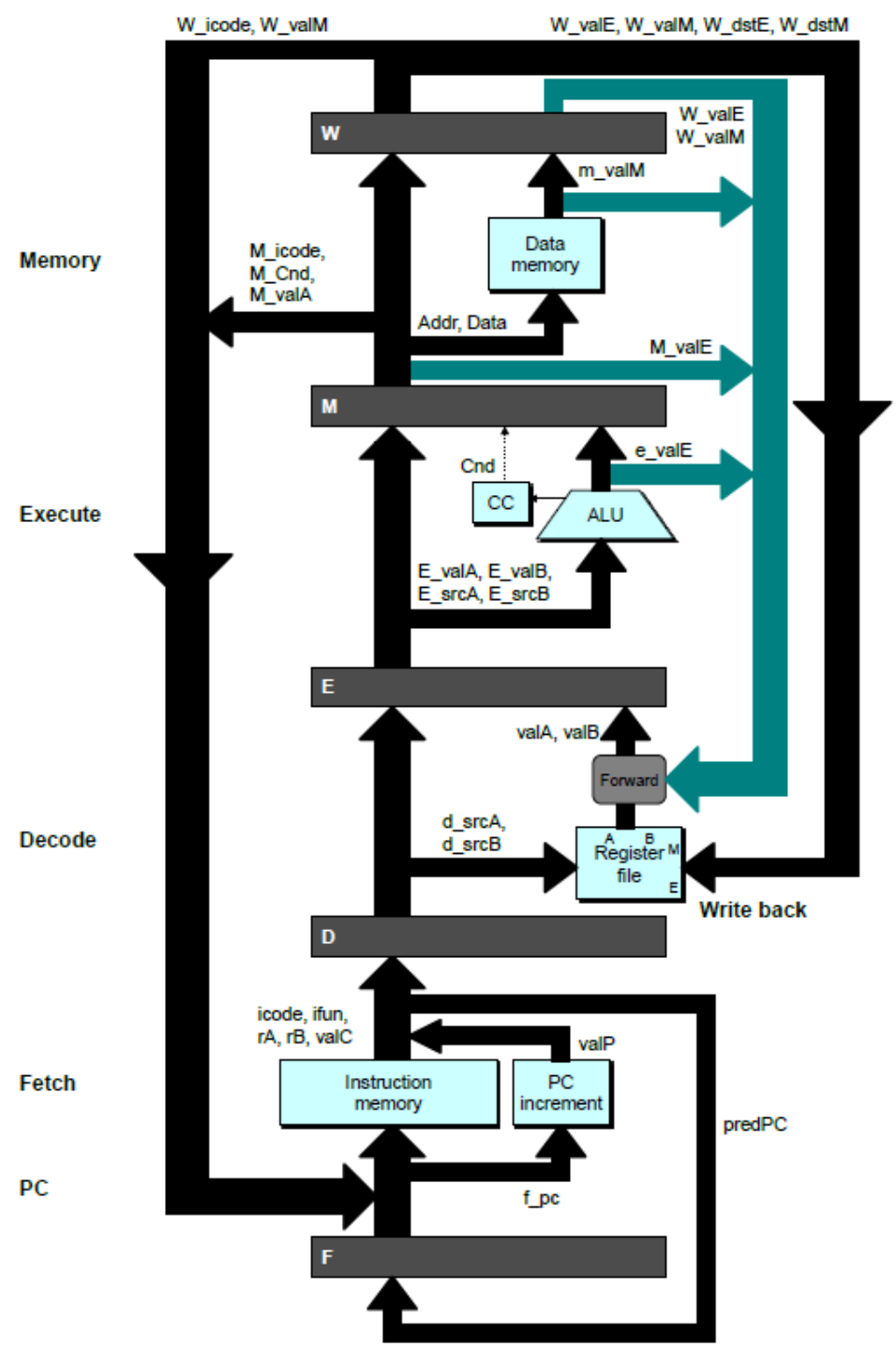
- Forward from the memory stage

Register `%rax`

- Forward from the execute stage



# Hardware Design



# Limitation of Forwarding

# demo-luh.js

0x000: irmovq \$128,%rdx

0x00a: irmovq \$3,%rcx

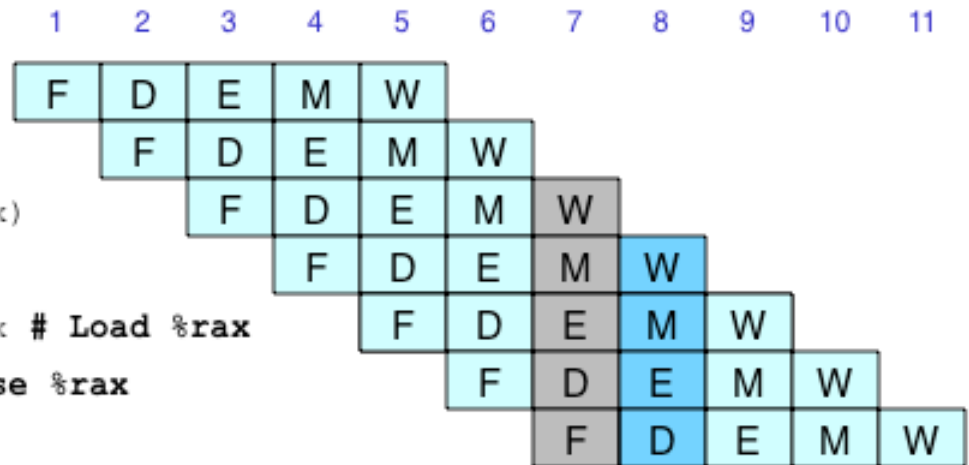
0x014: rmmovq %rcx, 0(%rdx)

0x01e: irmovq \$10,%rbx

0x028: mrmovq 0(%rdx),%rax # Load %rax

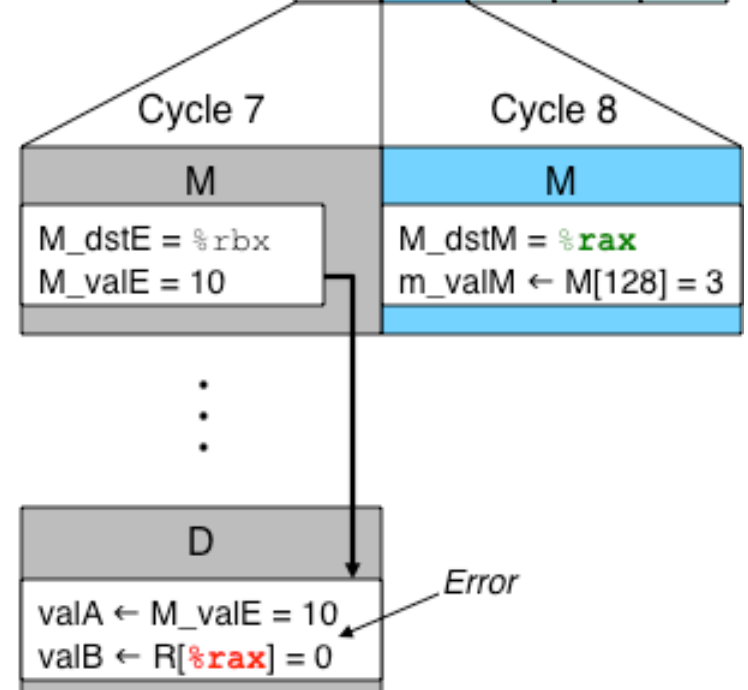
0x032: addq %rbx,%rax # Use %rax

0x034: halt

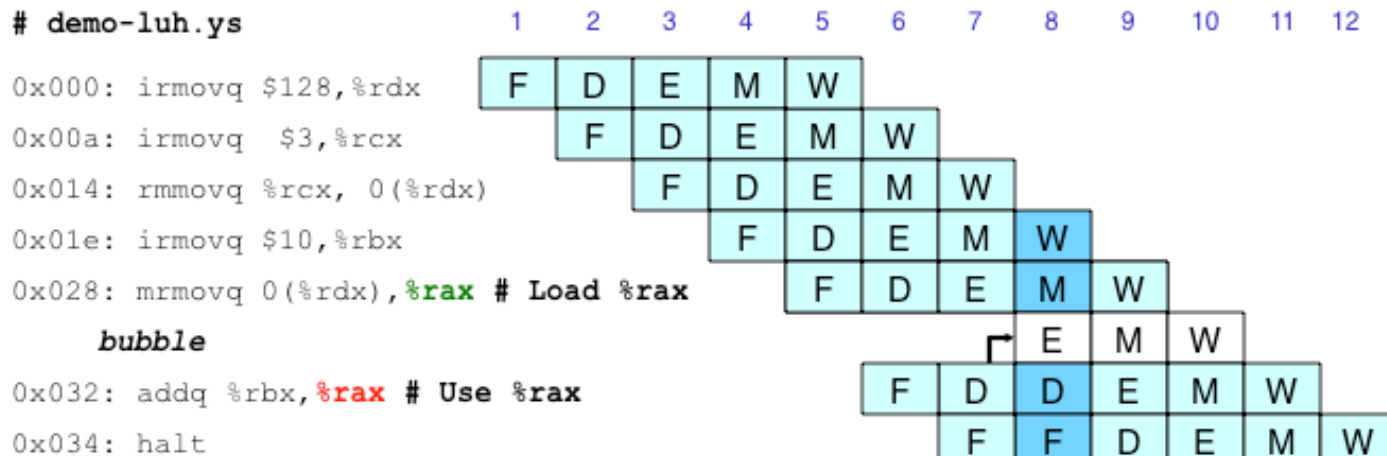


## Load-use dependency

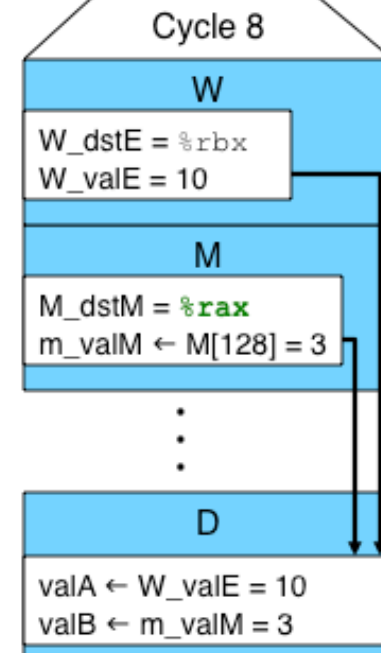
- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8



# Avoiding Load/Use Hazard



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

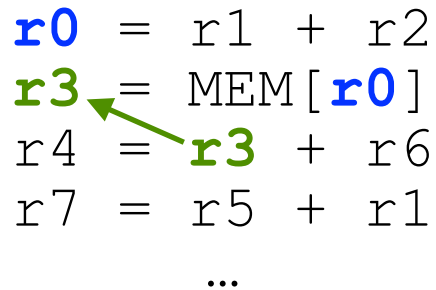


# Out-of-order Execution

- Compiler could do this, but has limitations
- Generally done in hardware

Long-latency instruction.  
Forces the pipeline to stall.

**r0** = r1 + r2  
**r3** = MEM[**r0**]  
r4 = **r3** + r6  
r7 = r5 + r1  
...



**r0** = r1 + r2  
**r3** = MEM[**r0**]  
r7 = r5 + r1  
...  
r4 = **r3** + r6

# Out-of-order Execution

$r0 = r1 + r2$   
 $r3 = \text{MEM}[r0]$   
 $r4 = r3 + \mathbf{r6}$   
 $\mathbf{r6} = r5 + r1$   
...

Is this correct?



$r0 = r1 + r2$   
 $r3 = \text{MEM}[r0]$   
 $\mathbf{r6} = r5 + r1$   
...  
 $r4 = r3 + \mathbf{r6}$

$r0 = r1 + r2$   
 $r3 = \text{MEM}[r0]$   
 $\mathbf{r4} = r3 + r6$   
 $\mathbf{r4} = r5 + r1$   
...

Is this correct?



$r0 = r1 + r2$   
 $r3 = \text{MEM}[r0]$   
 $\mathbf{r4} = r5 + r1$   
...  
 $\mathbf{r4} = r3 + r6$

“**Tomasolu Algorithm**” is the algorithm that is most widely implemented in modern hardware to get out-of-order execution right.