

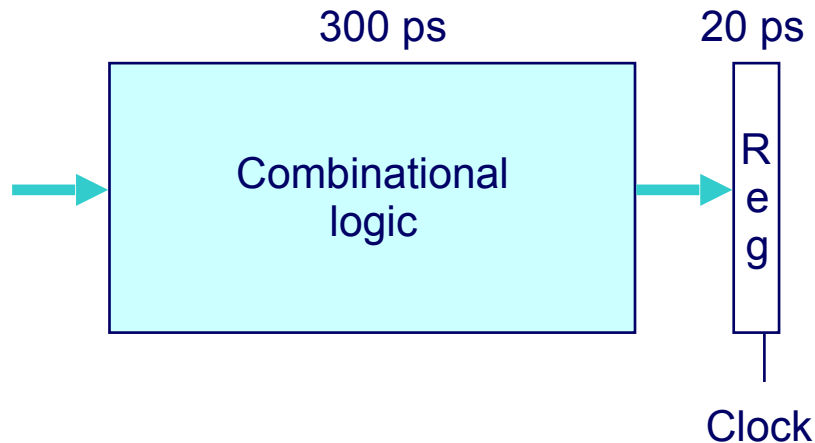
CSC 252: Computer Organization

Spring 2026: Lecture 13

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

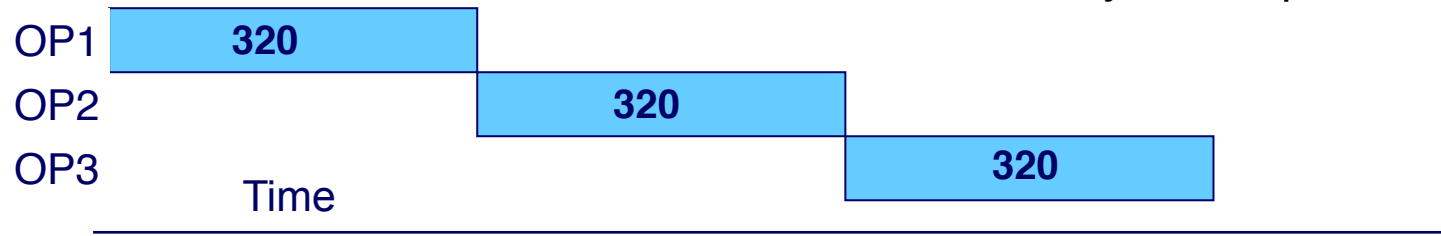
A Motivating Example



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

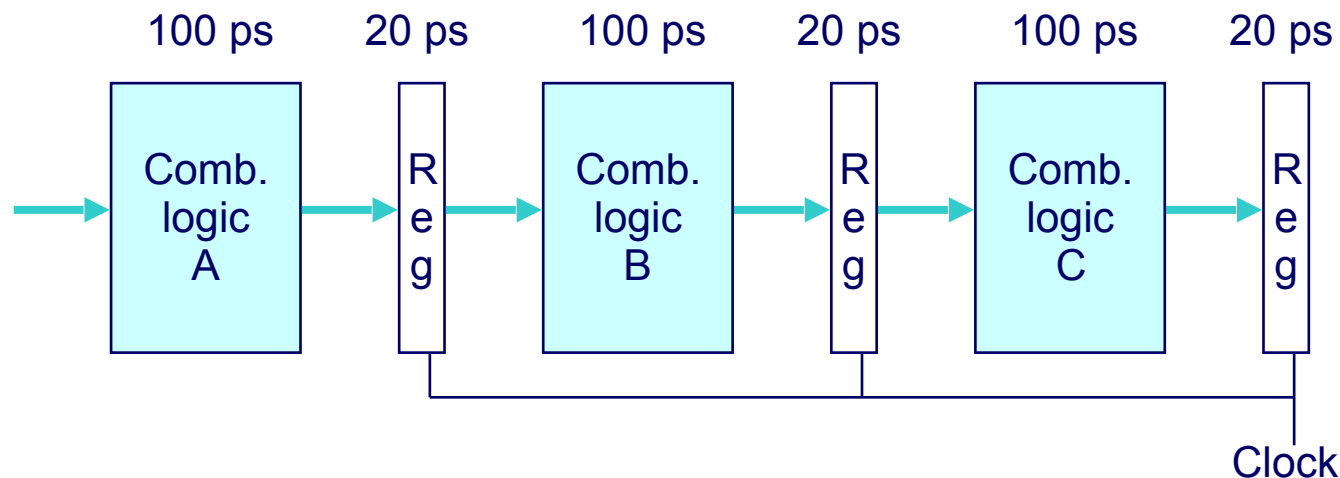
Pipeline Diagrams

- Time to finish 3 insts = 960 ps
- Each inst.'s latency is 320 ps



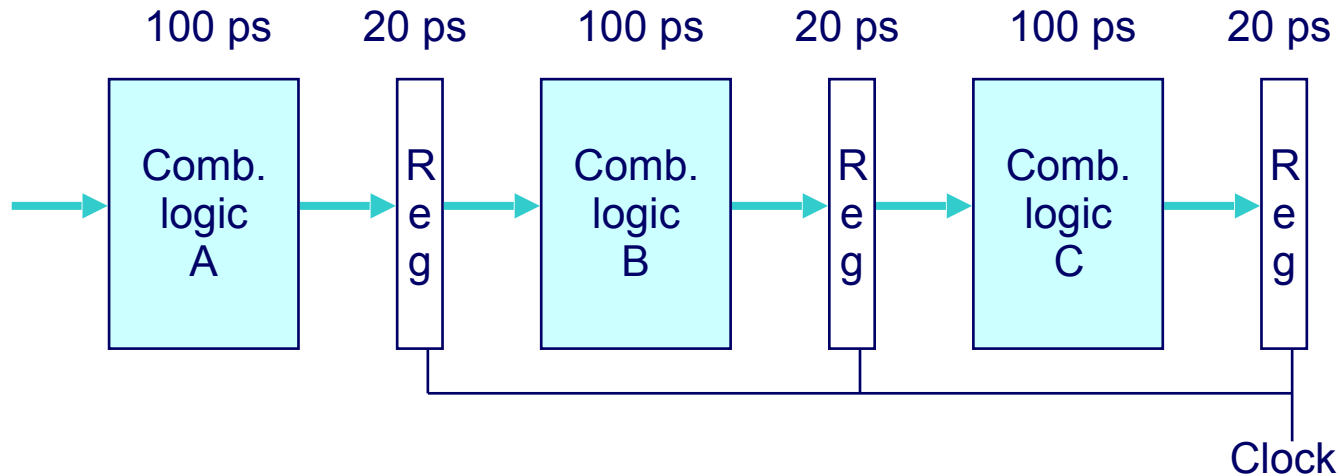
- 3 instructions will take 960 ps to finish
 - First cycle: Inst 1 takes 300 ps to compute new state, 20 ps to store the new states
 - Second cycle: Inst 2 starts; it takes 300 ps to compute new states, 20 ps to store new states
 - And so on...

3-Stage Pipelined Version

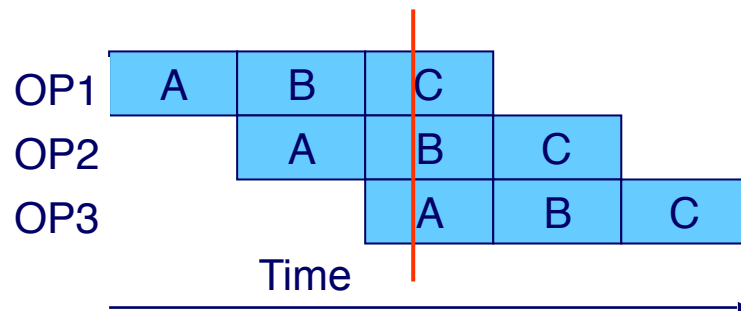


- Divide combinational logic into 3 stages of 100 ps each
- Insert registers between stages to store intermediate data between stages. These are called pipeline registers (ISA-invisible)
- Can begin a new instruction as soon as the previous one finishes stage A and has stored the intermediate data.
 - Begin new operation every **120 ps**
 - **Cycle time can be reduced to 120 ps**

3-Stage Pipelined Version

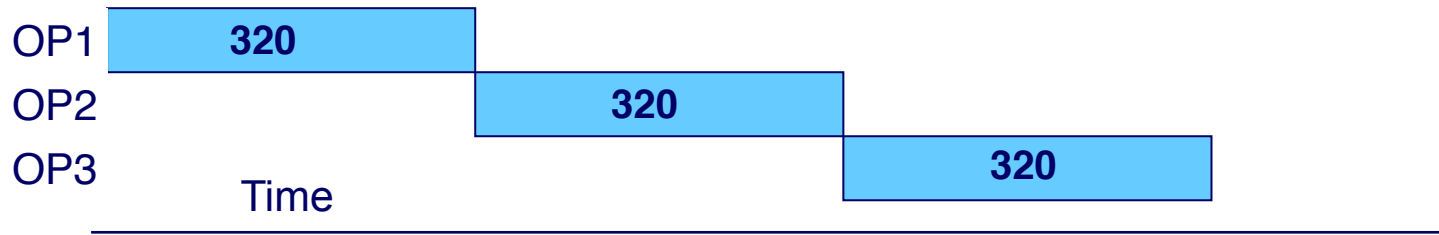


3-Stage Pipelined



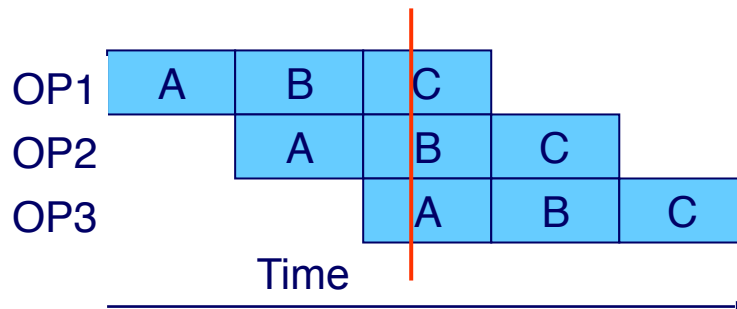
Comparison

Unpipelined



- Time to finish 3 insts = 960 ps
- Each inst.'s latency is 320 ps

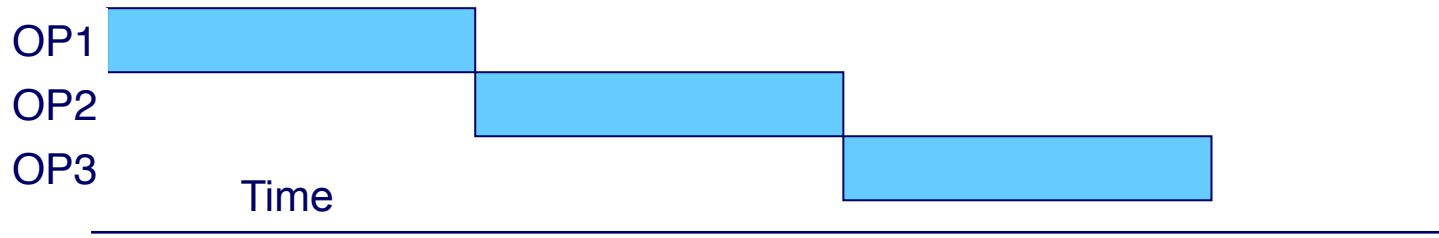
3-Stage Pipelined



- Time to finish 3 insets = $120 * 5 = 600$ ps
- But each inst.'s latency increases: $120 * 3 = 360$ ps

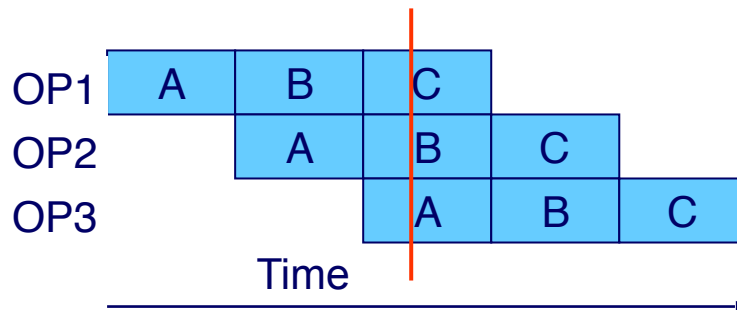
Benefits of Pipelining

- Time to finish 3 insts = 960 ps
- Each inst.'s latency is 320 ps



1. Reduce the cycle time from 320 ps to 120 ps

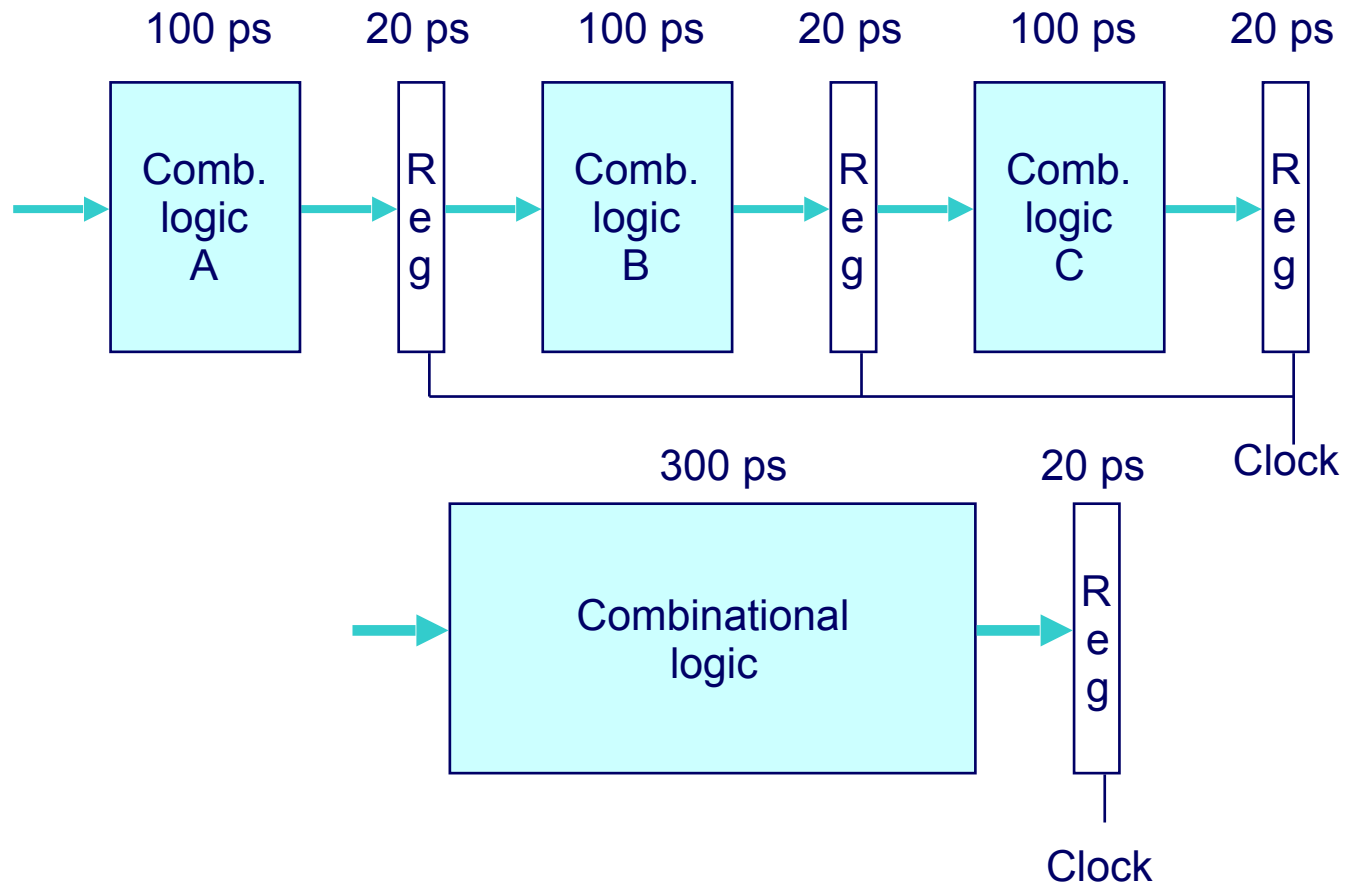
2. CPI reduces from 1 to 1/3 (i.e., executing 3 instruction in one cycle)



- Time to finish 3 insets = $120 * 5 = 600$ ps
- But each inst.'s latency increases: $120 * 3 = 360$ ps

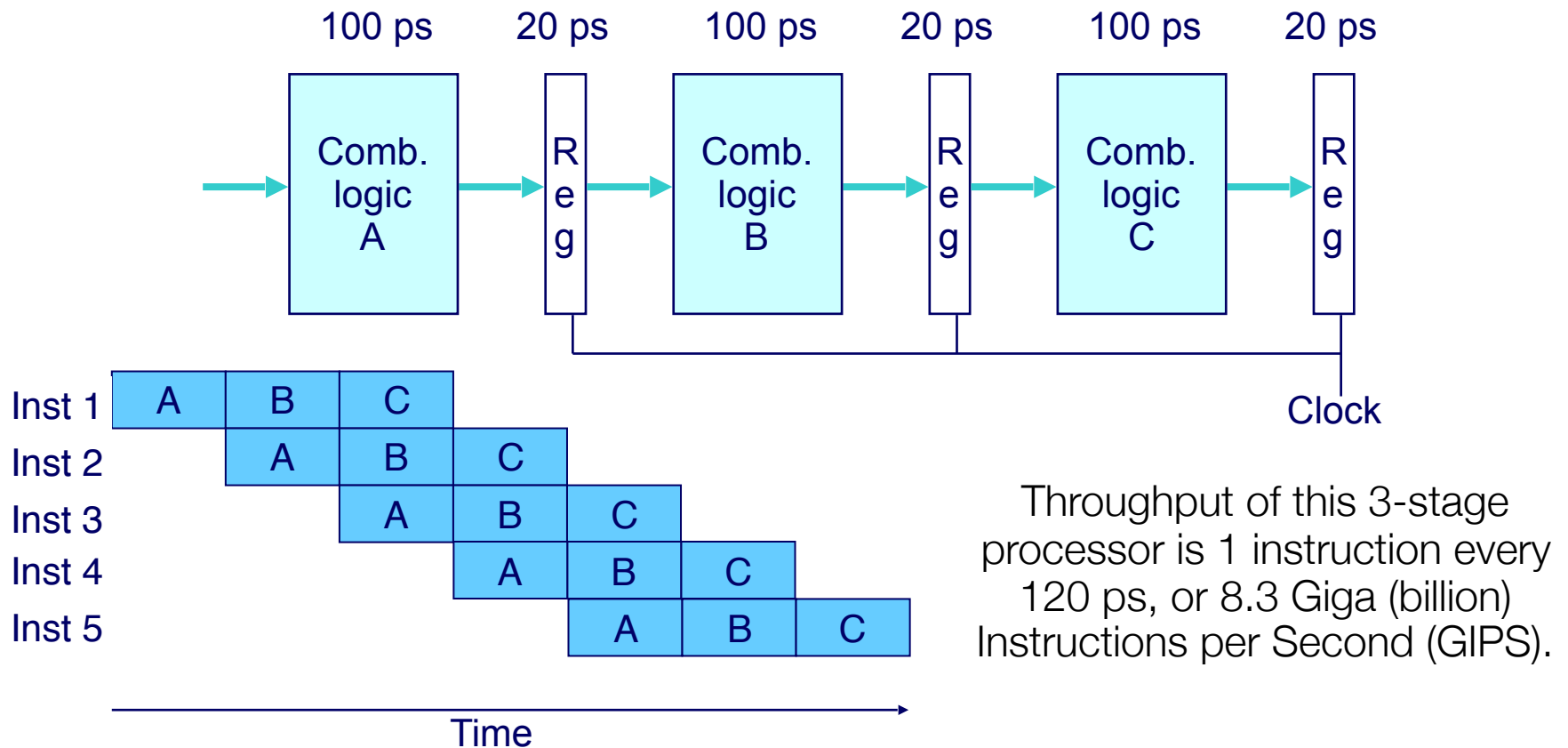
Pipeline Trade-offs

- Pros: Decrease the total execution time (Increase the “**throughput**”).
- Cons: Increase the latency of each instruction as new registers are needed between pipeline stages.



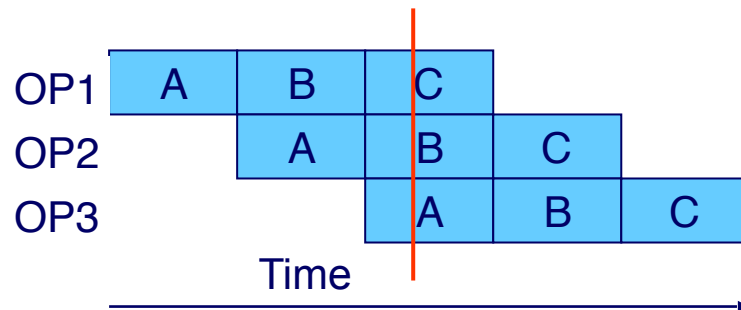
Throughput

- The rate at which the processor can finish executing an instruction (at the steady state).



One Requirement of Pipelining

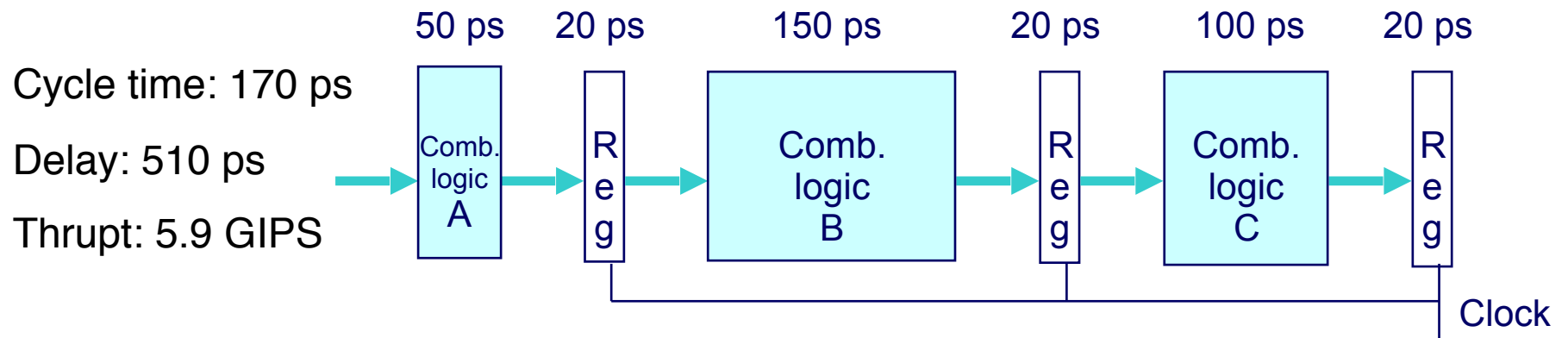
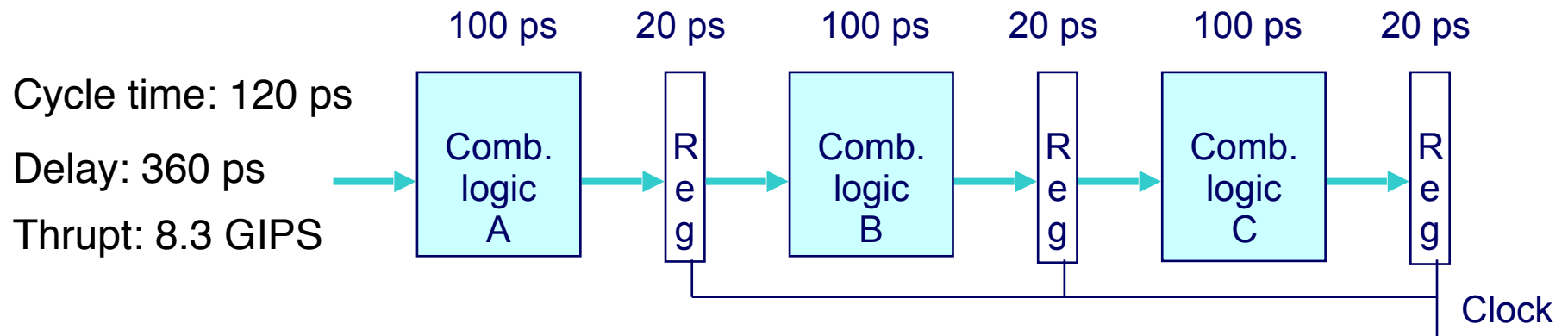
- The stages need to be using different hardware structures.
- That is, Stage A, Stage B, and Stage C need to exercise different parts of the combination logic.



- Time to finish 3 insets = $120 * 5 = 600$ ps
- But each inst.'s latency increases: $120 * 3 = 360$ ps

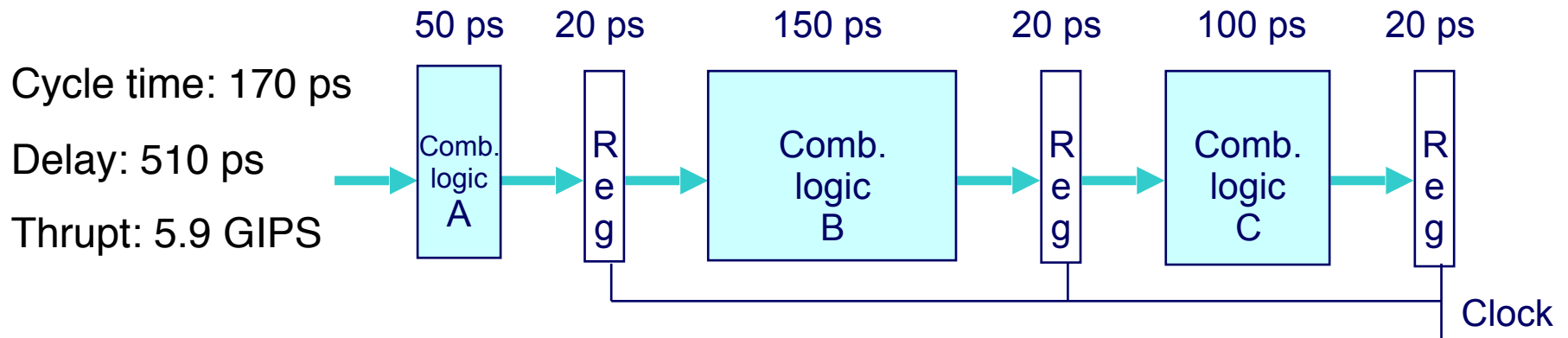
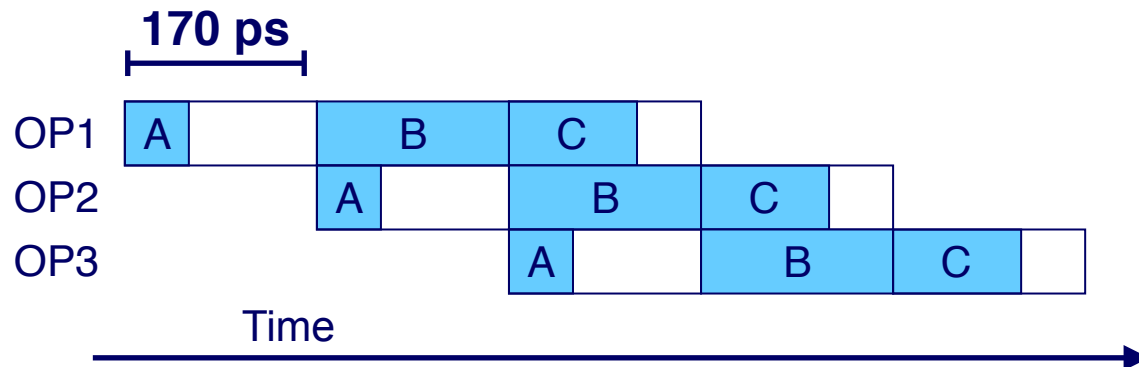
Aside: Unbalanced Pipeline

- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput



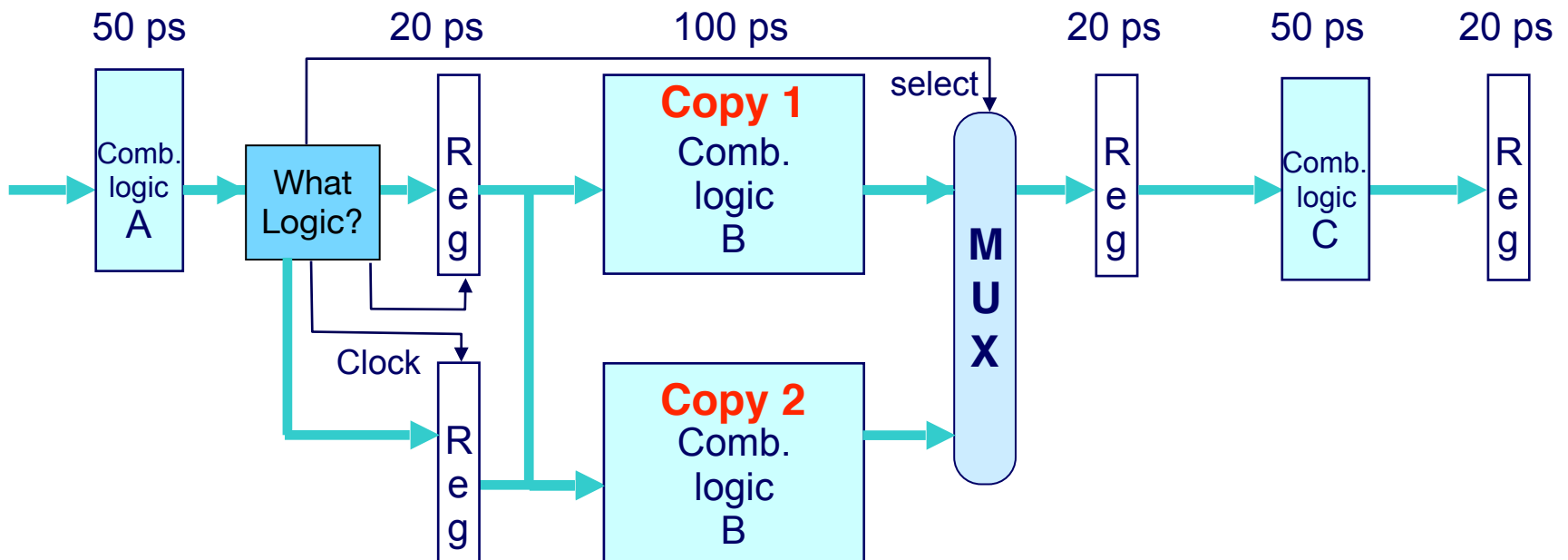
Aside: Unbalanced Pipeline

- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput



Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
 - Not always possible. What to do if we can't further pipeline a stage?
- Solution 2: Use multiple copies of the slow component

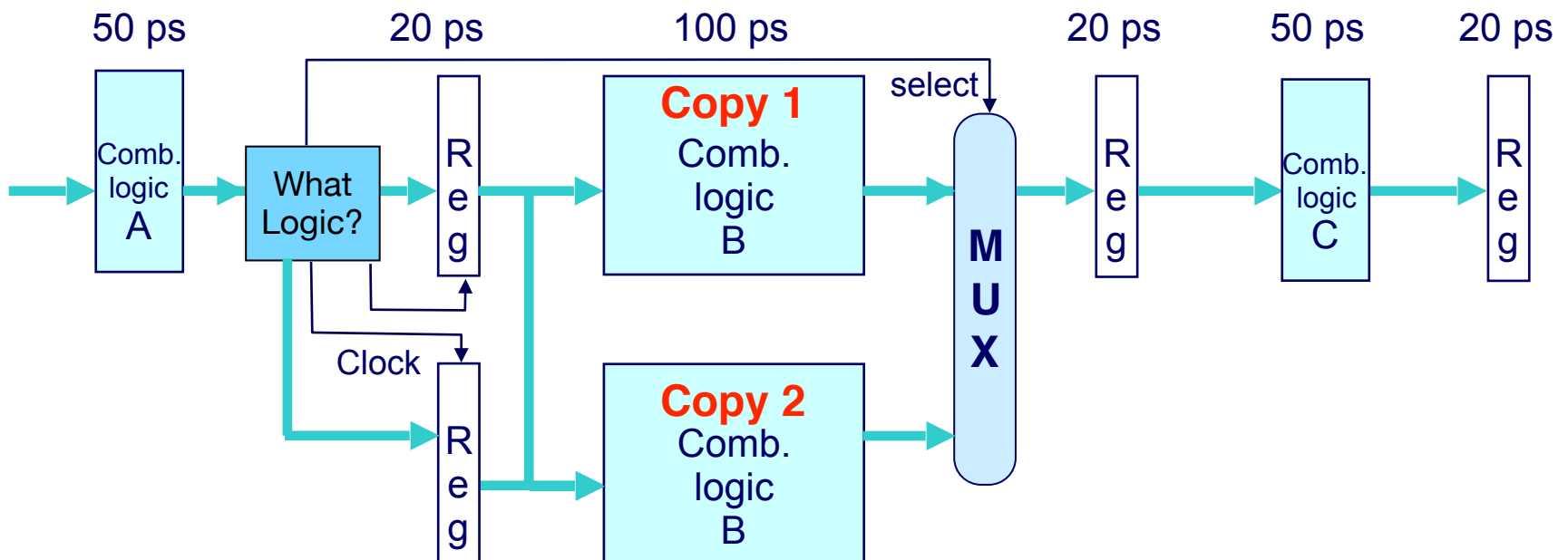


What logic do you need there?

Hint: it needs to control the clock signals of the two registers and the select signal of the MUX.

Aside: Mitigating Unbalanced Pipeline

- Data sent to copy 1 in odd cycles and to copy 2 in even cycles.
- This is called 2-way interleaving. Effectively the same as pipelining Comb. logic B into two sub-stages.
- The cycle time is reduced to 70 ps (as opposed to 120 ps) at the cost of extra hardware.



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

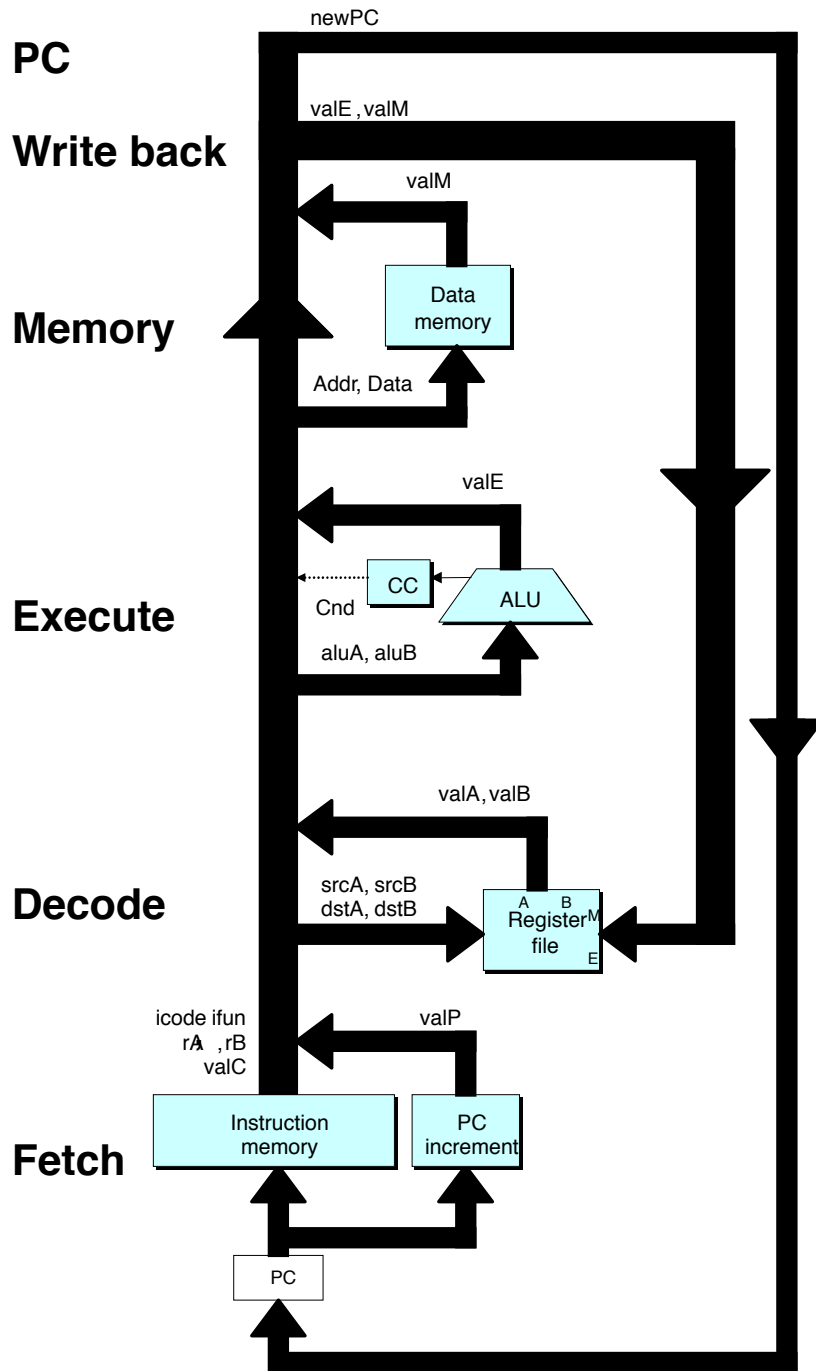
Decode: Read program registers

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

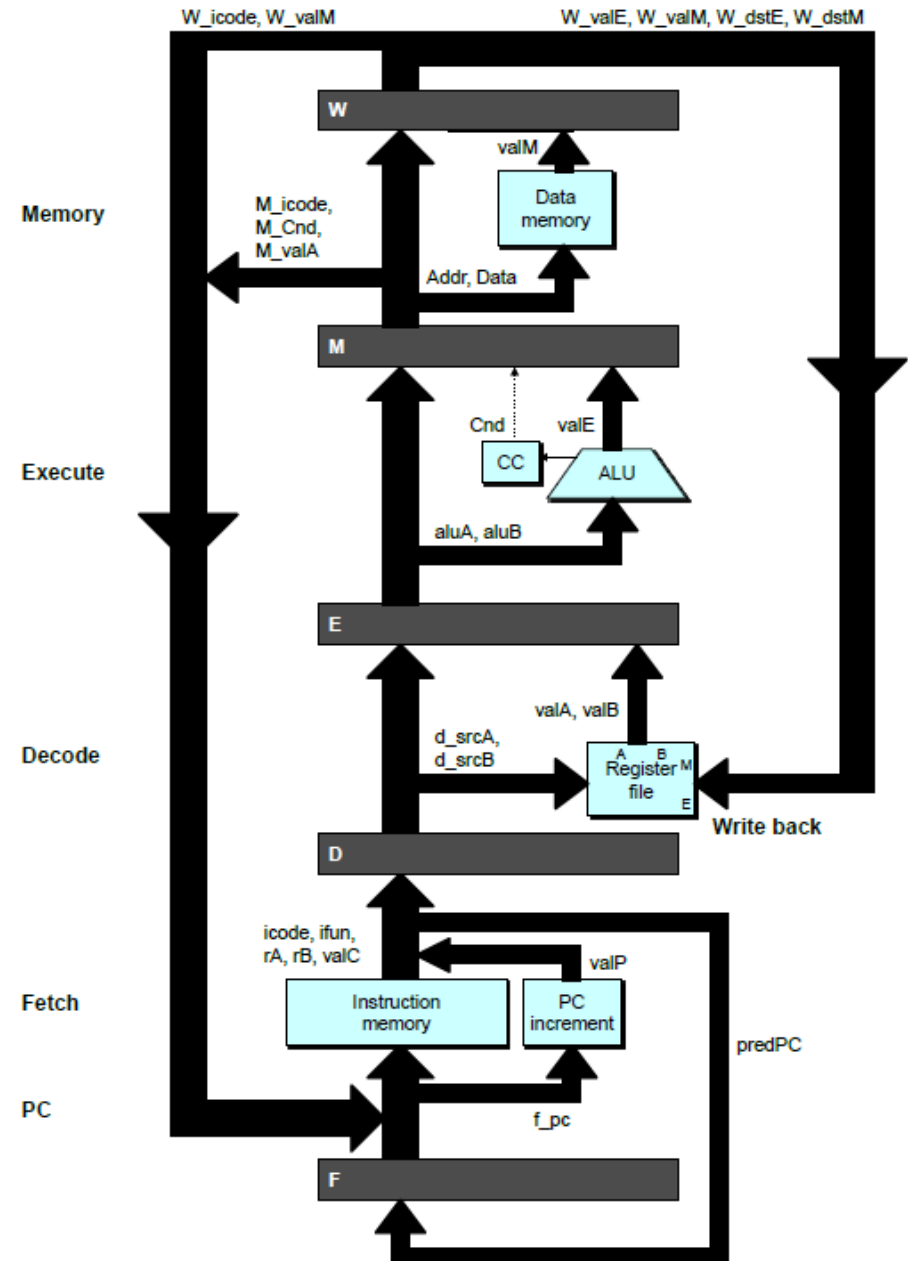
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file



Real-World Pipelines: Car Washes

Sequential



Pipelined

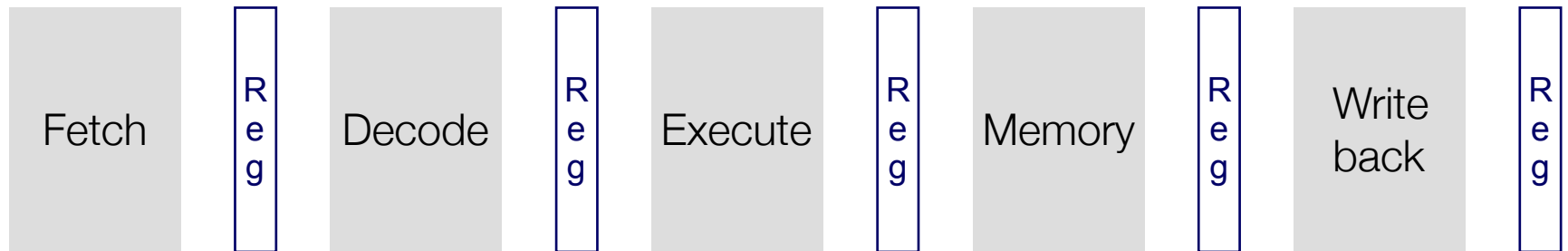


Idea

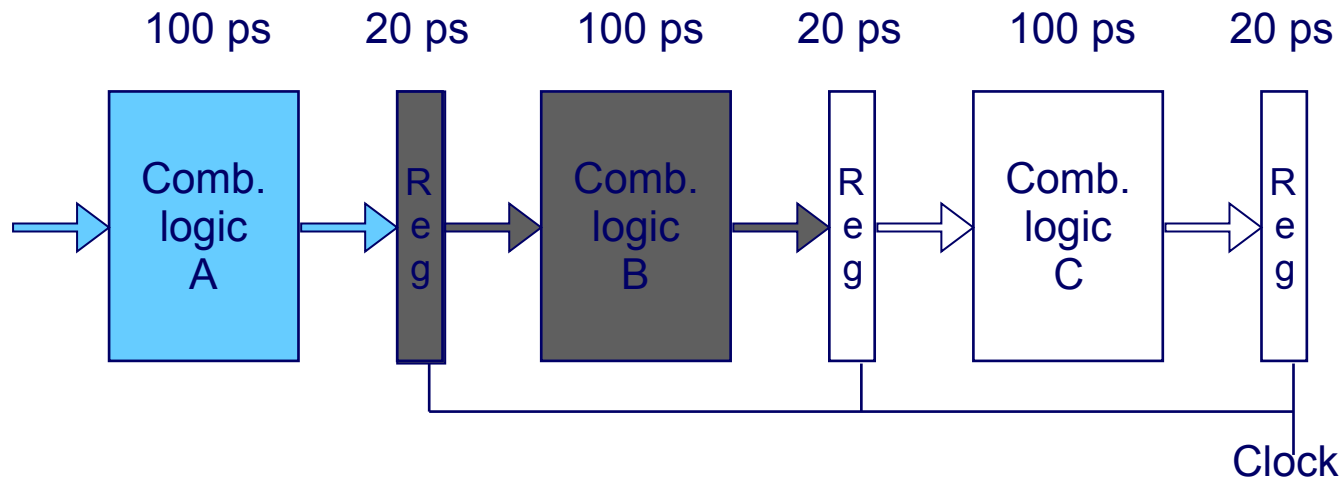
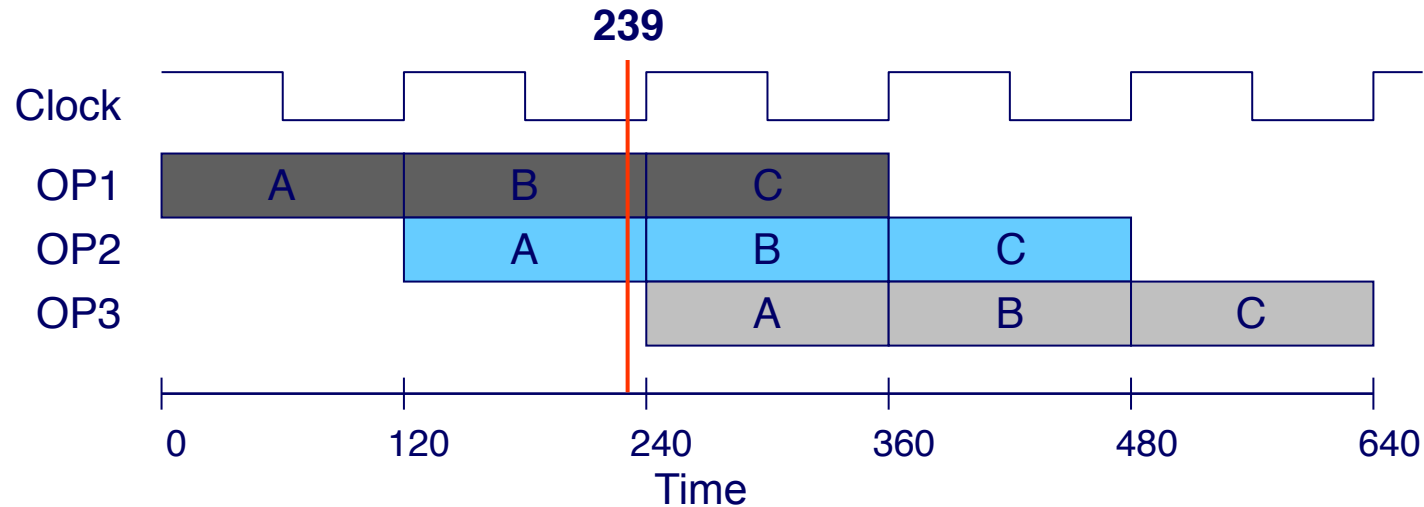
- Divide process into independent stages
- Move objects through stages in sequence
- At any given times, multiple objects being processed

Pipeline Illustration

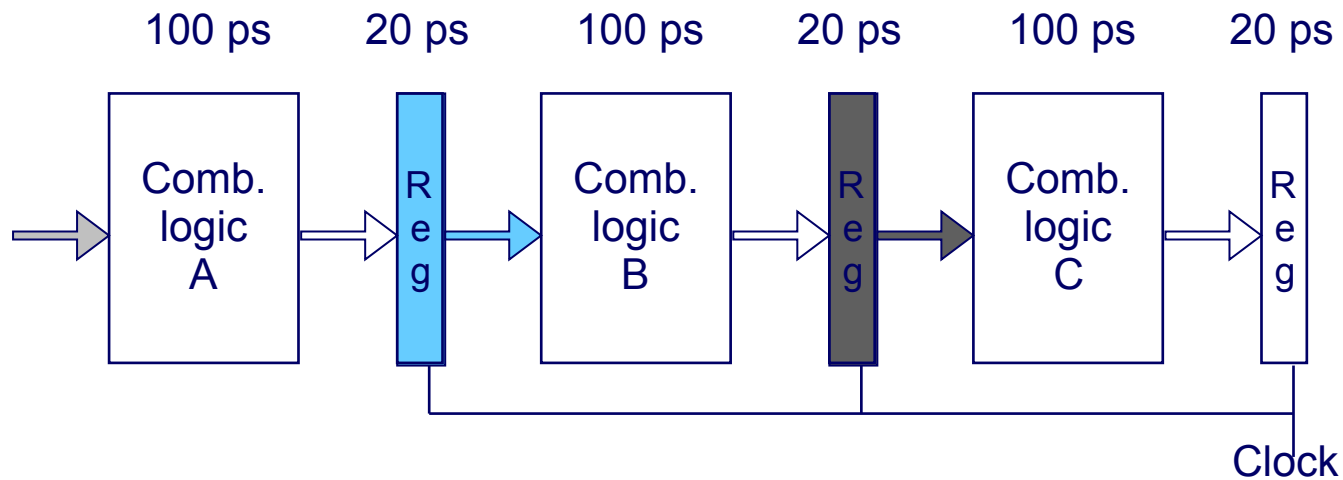
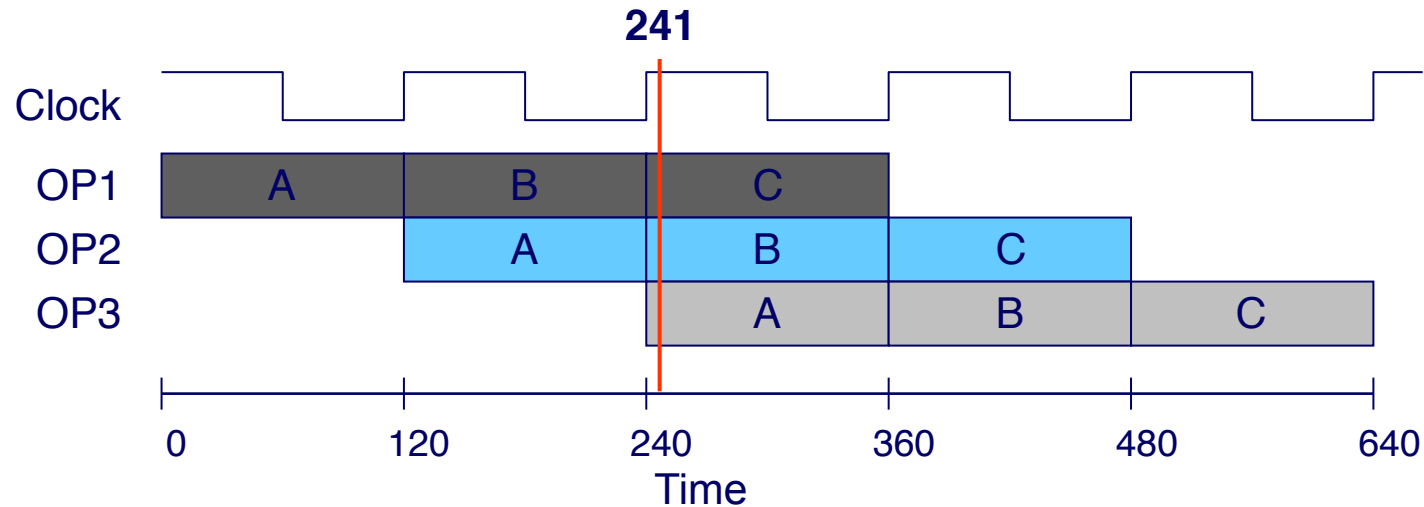
Inst0



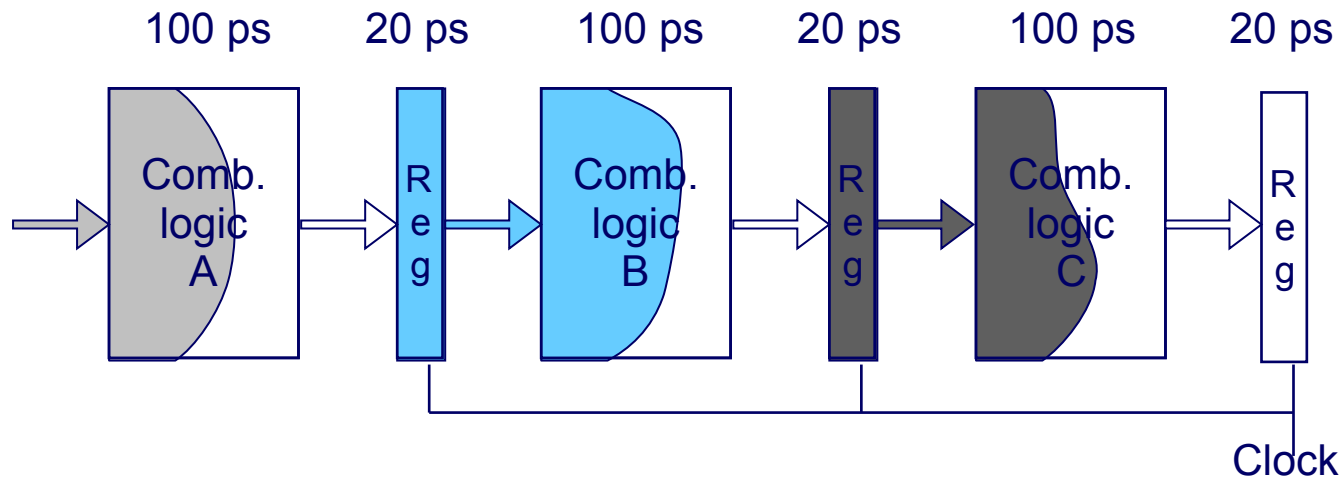
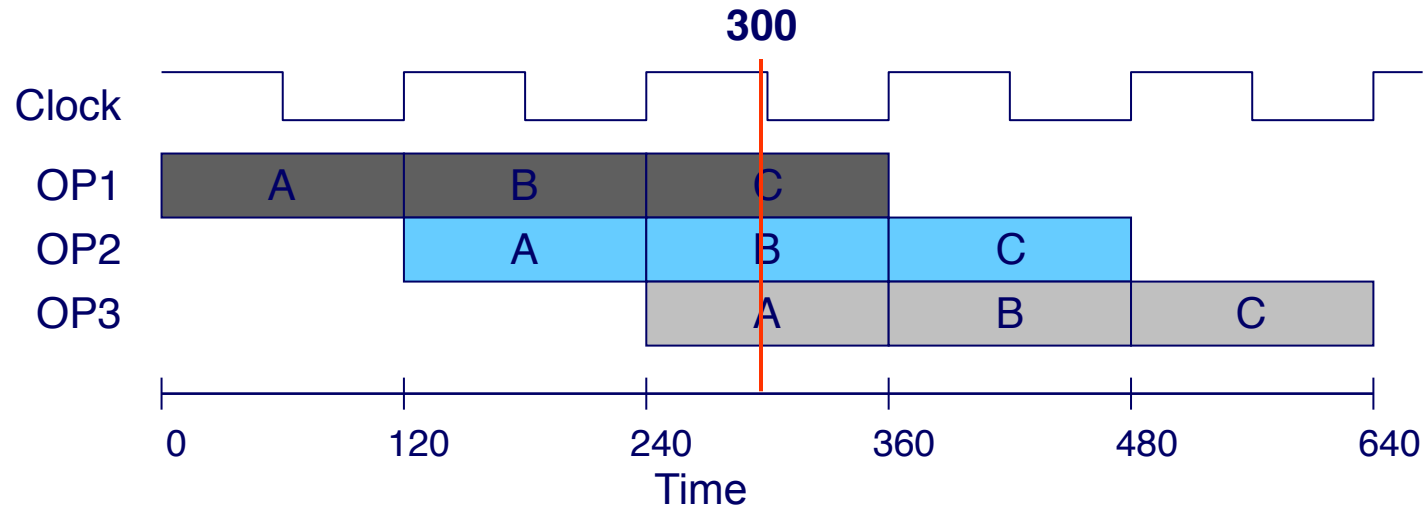
Another Illustration



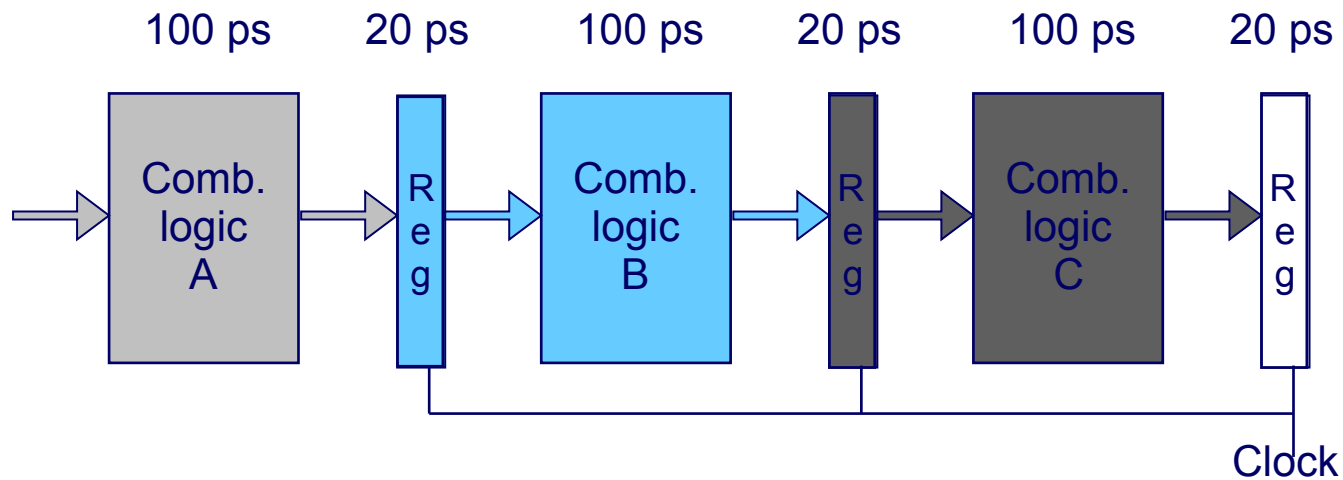
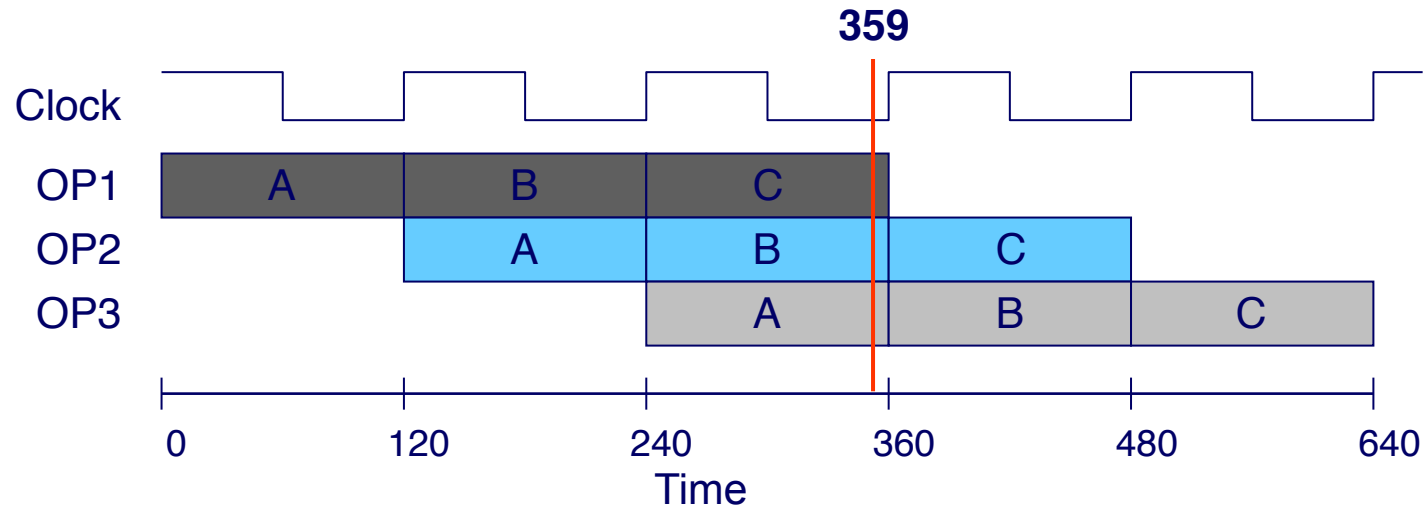
Another Illustration



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

		1	2	3	4	5	6	7	8	9
	xorg %rax, %rax	F	D	E	M	W				
	jne L1		F	D	E	M	W			
	nop			F	D	E	M	W		
L1	movq \$1, %rax				F	D	E	M	W	
	movq \$4, %rcx					F	D	E	M	W
	irmovq \$3, %rax						F	D	E	M
								F	D	E
									F	D
										F

Diagram illustrating the control flow graph (CFG) for the assembly code. The CFG is a grid of nodes (F, D, E, M, W) representing instructions. The flow starts at the top-left node (F) and proceeds through the sequence of instructions. The flow ends at the bottom-right node (F). The flow is marked as "Not taken" for the jump instruction (jne L1) and "Fall Through" for the subsequent instructions.

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();

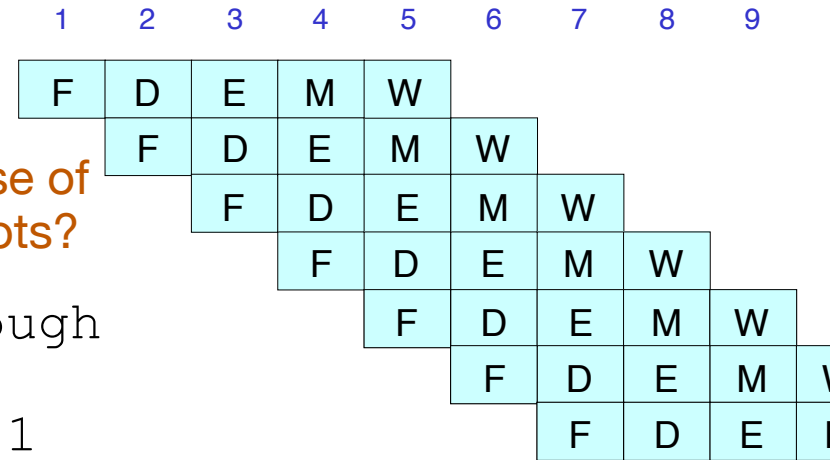
```

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?



A less obvious example

```

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

```

add A, B
or C, D
sub E, F
jle 0x200
add A, C
  
```

→

```

add A, B
sub E, F
jle 0x200
or C, D
add A, C
  
```

Why don't we move the `sub` instruction?

Resolving Control Dependencies

- **Software Mechanisms**

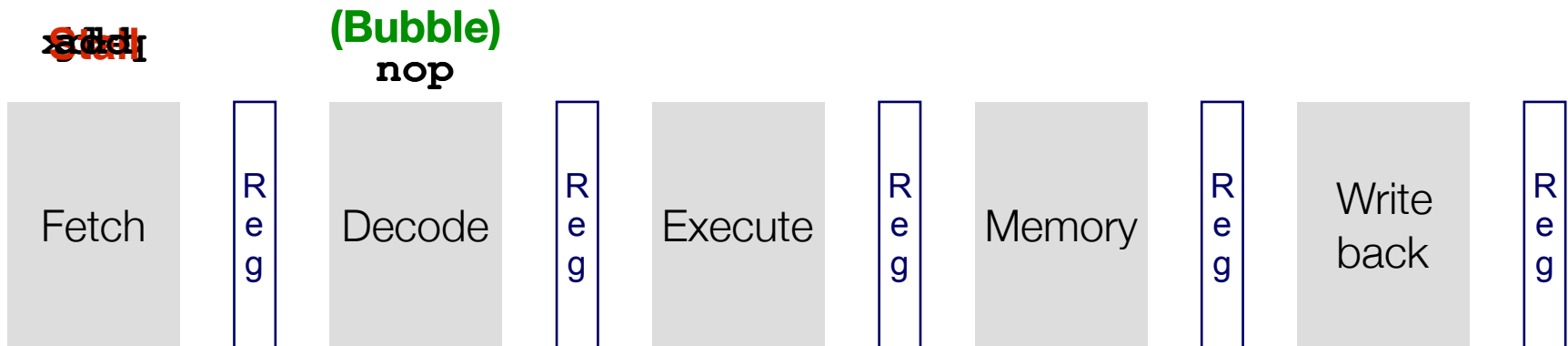
- Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
- Delay slot: insert instructions that do not depend on the effect of the preceding instruction. These instructions will execute even if the preceding branch is taken — old RISC approach

- **Hardware mechanisms**

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

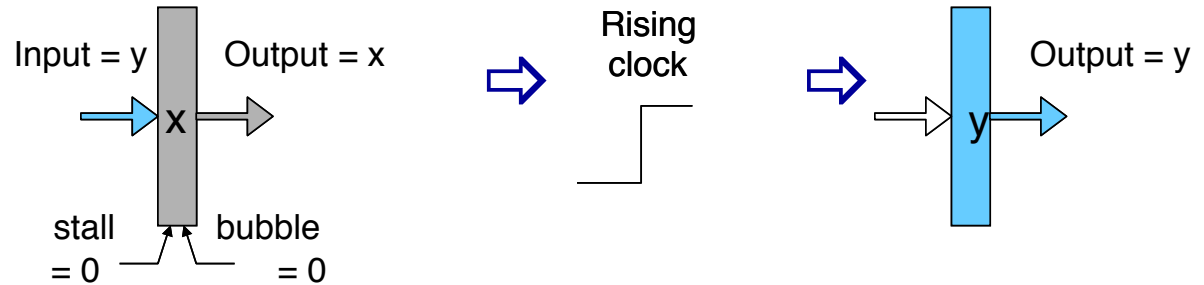
Hardware Generated Nops (Bubble and Stalling)

- **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?

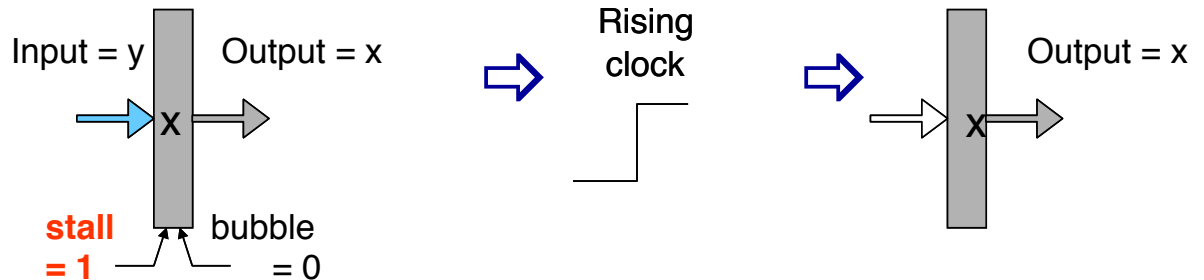


How are Stall and Bubble Implemented in Hardware?

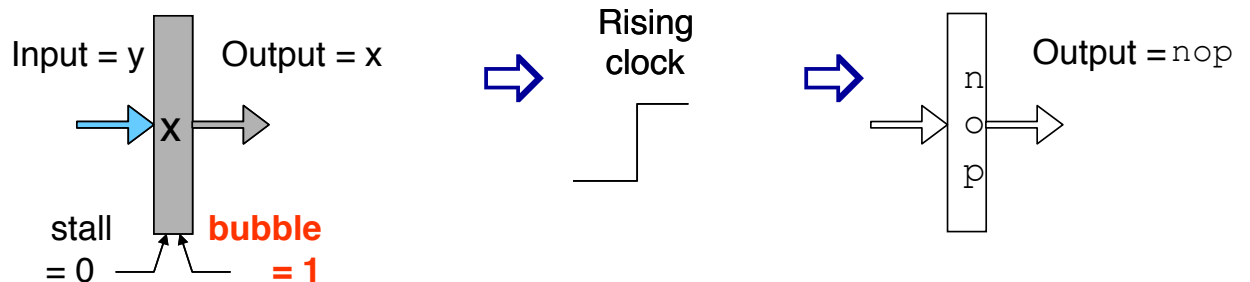
Normal



Stall



Bubble



Branch Prediction

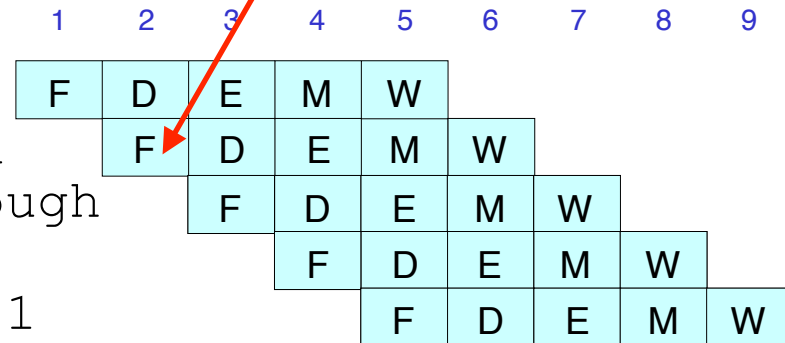
Idea: instead of waiting, why not just guess the direction of jump?

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()  
}
```