

CSC 252/452: Computer Organization

Fall 2025: Lecture 13

Instructor: Yanan Guo

Department of Computer Science
University of Rochester

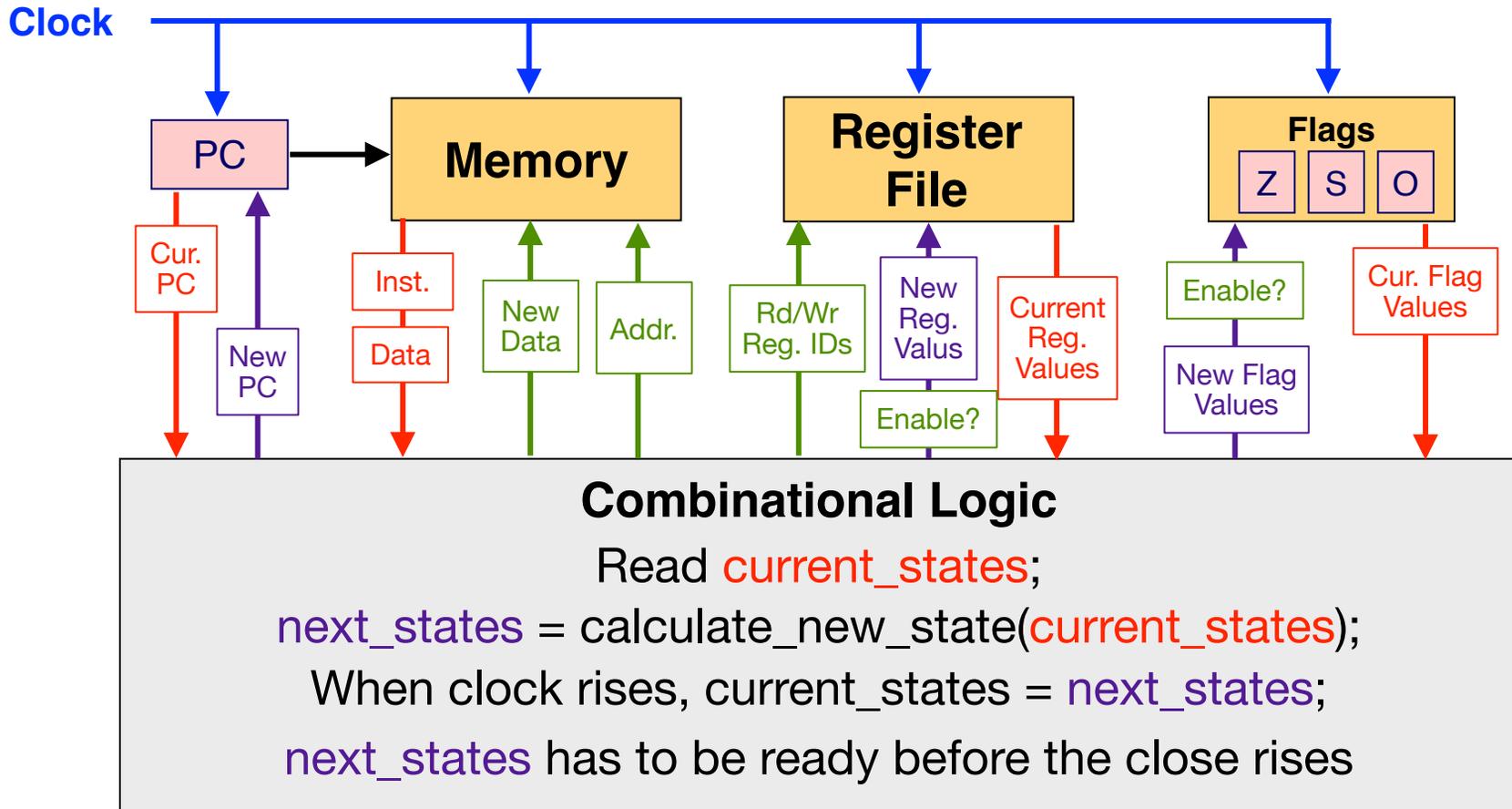
Announcement

- Programming assignment 3 is out
 - Details: <https://www.cs.rochester.edu/courses/252/fall2025/labs/assignment3.html>
 - Due on **Oct. 27th**, 11:59 PM
 - You (may still) have 3 slip days

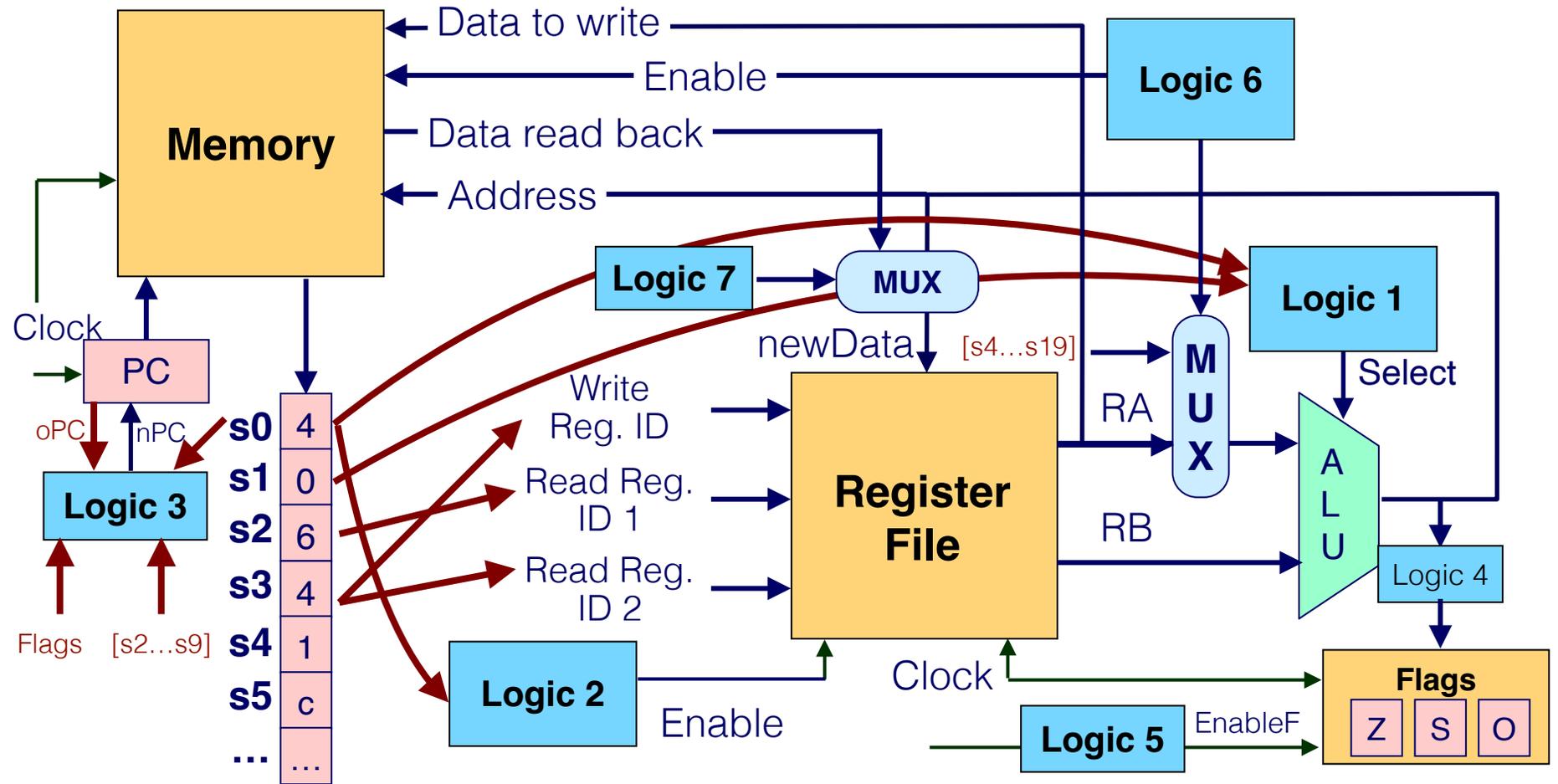
Announcement

- Programming assignment 3 is in x86 assembly language. Seek help from TAs.
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.

Single-Cycle Microarchitecture



Single-Cycle Microarchitecture



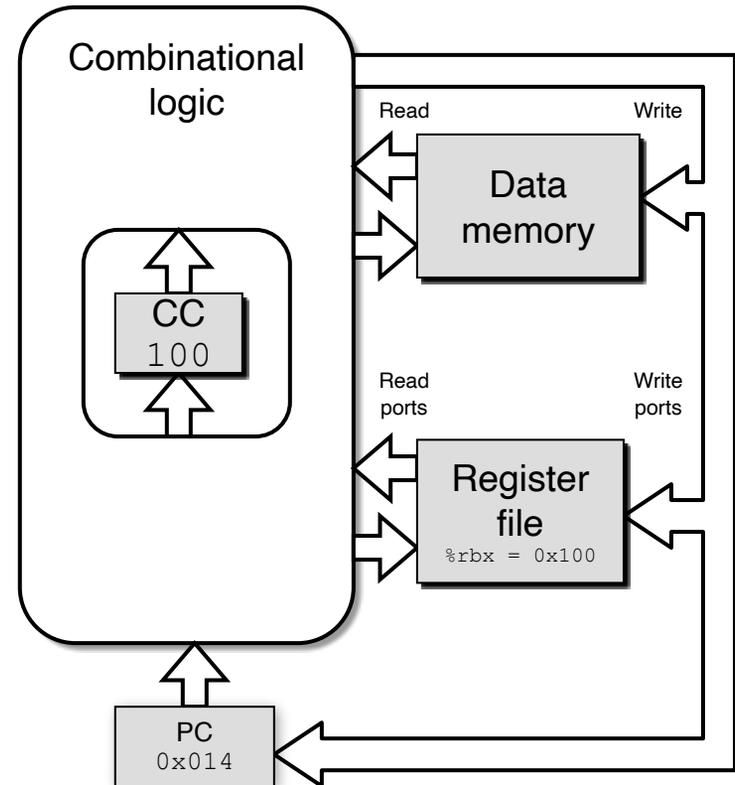
Single-Cycle Microarchitecture: Illustration

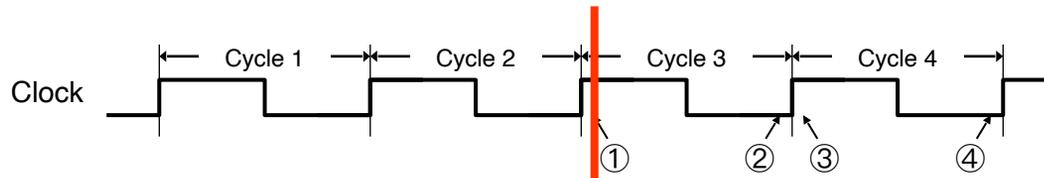
Think of it as a state machine

Every cycle, one instruction gets executed. At the end of the cycle, architecture states get modified.

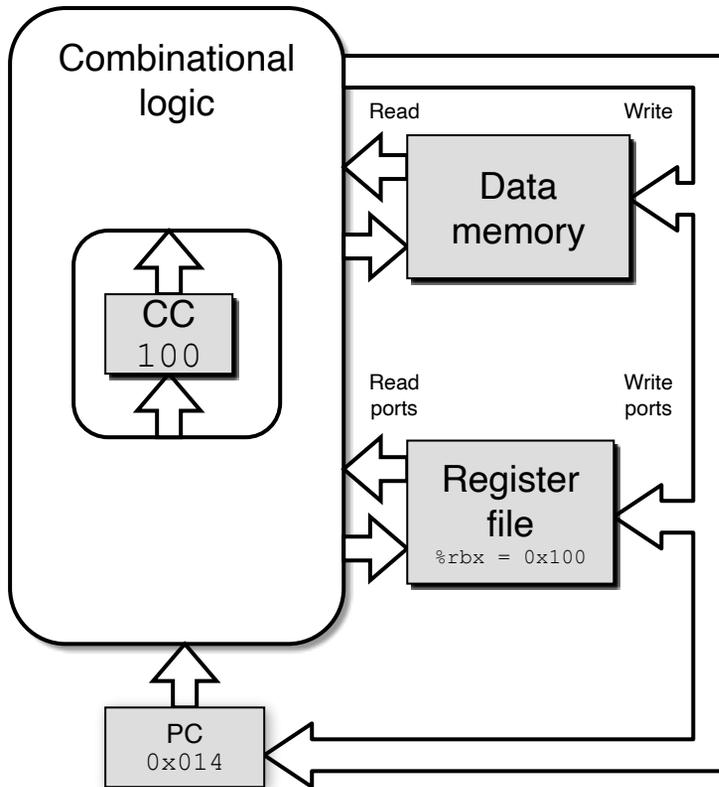
States (All updated as clock rises)

- PC register
- Cond. Code register
- Data memory
- Register file

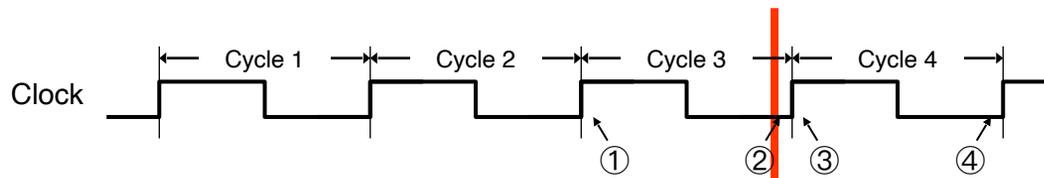




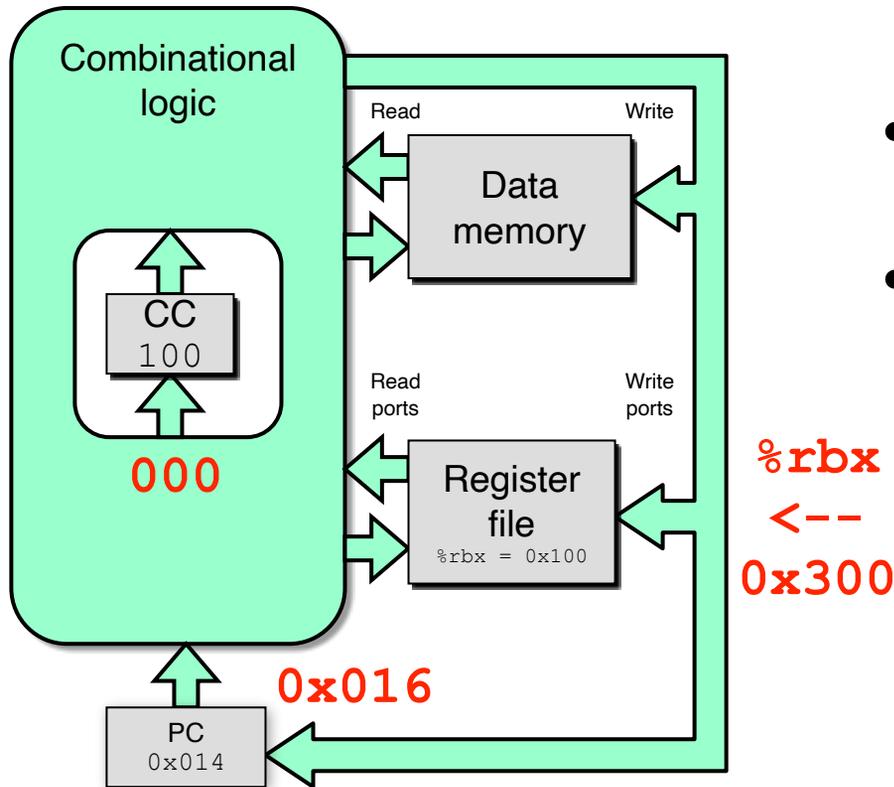
Cycle 1:	0x000:	<code>irmovq \$0x100,%rbx</code>	# %rbx <-- 0x100
Cycle 2:	0x00a:	<code>irmovq \$0x200,%rdx</code>	# %rdx <-- 0x200
Cycle 3:	0x014:	<code>addq %rdx,%rbx</code>	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	<code>je dest</code>	# Not taken
Cycle 5:	0x01f:	<code>rmmovq %rbx,0(%rdx)</code>	# M[0x200] <-- 0x300



- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

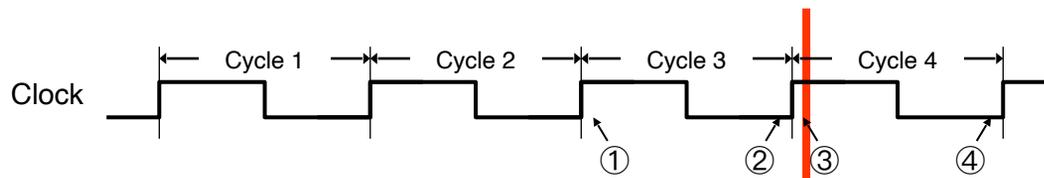


Cycle 1:	0x000:	<code>irmovq \$0x100,%rbx</code>	# %rbx <-- 0x100
Cycle 2:	0x00a:	<code>irmovq \$0x200,%rdx</code>	# %rdx <-- 0x200
Cycle 3:	0x014:	<code>addq %rdx,%rbx</code>	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	<code>je dest</code>	# Not taken
Cycle 5:	0x01f:	<code>rmmovq %rbx,0(%rdx)</code>	# M[0x200] <-- 0x300

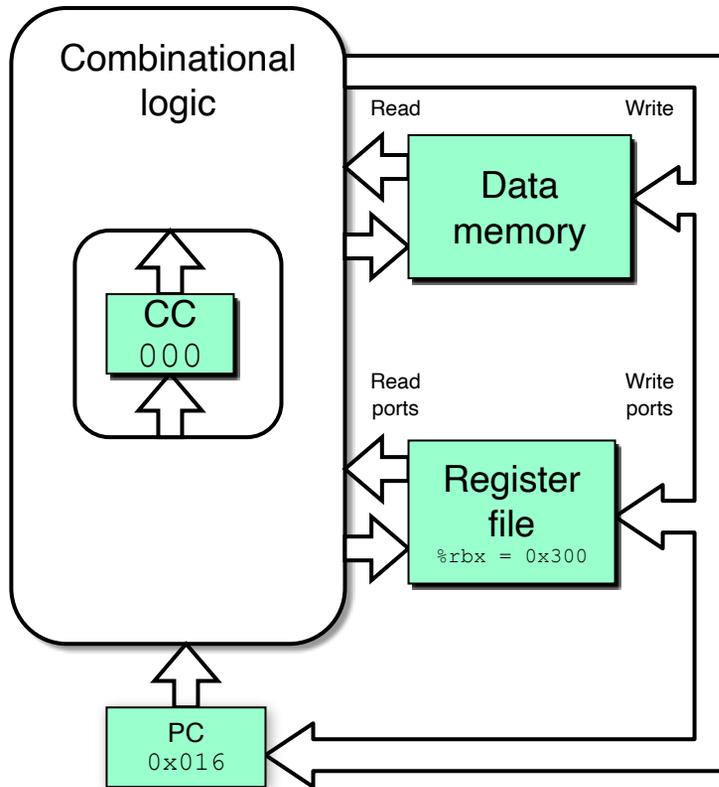


- state set according to second `irmovq` instruction
- combinational logic generates results for `addq` instruction

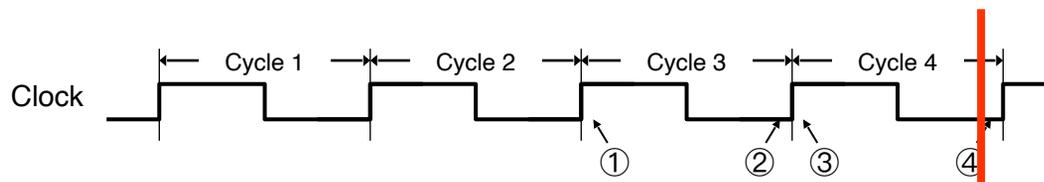
%rbx
<--
0x300



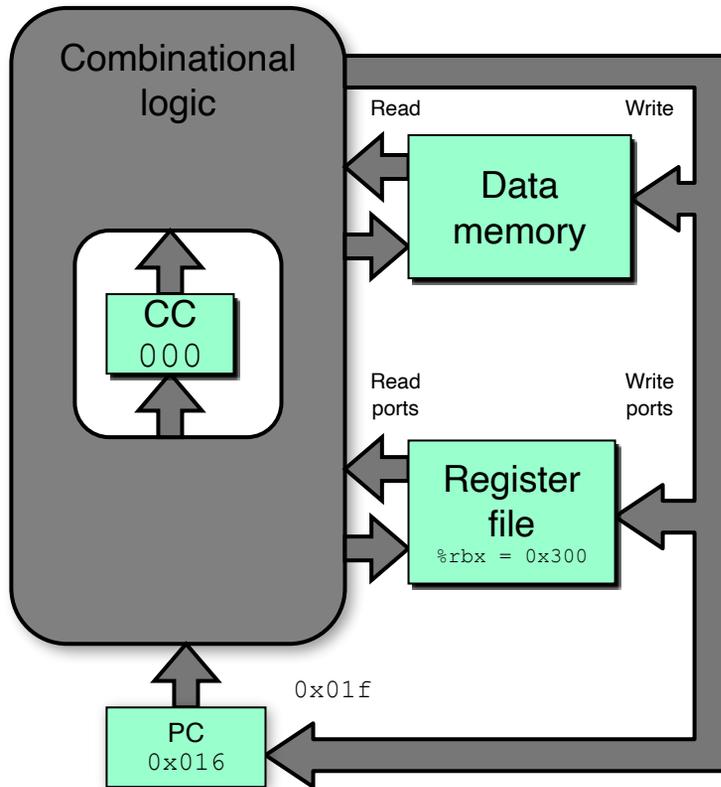
Cycle 1:	0x000:	<code>irmovq \$0x100,%rbx</code>	# %rbx <-- 0x100
Cycle 2:	0x00a:	<code>irmovq \$0x200,%rdx</code>	# %rdx <-- 0x200
Cycle 3:	0x014:	<code>addq %rdx,%rbx</code>	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	<code>je dest</code>	# Not taken
Cycle 5:	0x01f:	<code>rmmovq %rbx,0(%rdx)</code>	# M[0x200] <-- 0x300



- state set according to `addq` instruction
- combinational logic starting to react to state changes



Cycle 1:	0x000:	<code>irmovq \$0x100,%rbx</code>	# %rbx <-- 0x100
Cycle 2:	0x00a:	<code>irmovq \$0x200,%rdx</code>	# %rdx <-- 0x200
Cycle 3:	0x014:	<code>addq %rdx,%rbx</code>	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	<code>je dest</code>	# Not taken
Cycle 5:	0x01f:	<code>rmmovq %rbx,0(%rdx)</code>	# M[0x200] <-- 0x300



- state set according to `addq` instruction
- combinational logic generates results for `je` instruction

Processor Microarchitecture

- Sequential, single-cycle microarchitecture implementation
 - Basic idea
 - Hardware implementation
- Pipelined microarchitecture implementation
 - Basic Principles
 - Difficulties: Control Dependency
 - Difficulties: Data Dependency

Performance Model

Execution time
of a program
(in seconds) = # of **Dynamic** Instructions

X # of cycles taken to execute an instruction (on average)

/ number of cycles per second

Performance Model

Execution time
of a program
(in seconds)

= # of **Dynamic** Instructions

CPI

X # of cycles taken to execute an instruction (on average)

/ number of cycles per second

Performance Model

Execution time
of a program
(in seconds)

= # of **Dynamic** Instructions

CPI

X # of cycles taken to execute an instruction (on average)

/ number of cycles per second

Clock Frequency
(1/cycle time)

Improving Performance

Execution time
of a program
(in seconds) = # of **Dynamic** Instructions

X # of cycles taken to execute an instruction (on average)

/ number of cycles per second

- 1. Reduce the total number of instructions executed (mainly done by the compiler and/or programmer).

Improving Performance

Execution time
of a program
(in seconds) = # of **Dynamic** Instructions

X # of cycles taken to execute an instruction (on average)

/ number of cycles per second

- 1. Reduce the total number of instructions executed (mainly done by the compiler and/or programmer).
- 2. Increase the clock frequency (reduce the cycle time). Has huge power implications.

Improving Performance

Execution time
of a program
(in seconds) = # of **Dynamic** Instructions

X # of cycles taken to execute an instruction (on average)

/ number of cycles per second

- 1. Reduce the total number of instructions executed (mainly done by the compiler and/or programmer).
- 2. Increase the clock frequency (reduce the cycle time). Has huge power implications.
- 3. Reduce the CPI, i.e., execute more instructions in one cycle.

Improving Performance

Execution time
of a program
(in seconds) = # of **Dynamic** Instructions

X # of cycles taken to execute an instruction (on average)

/ number of cycles per second

- 1. Reduce the total number of instructions executed (mainly done by the compiler and/or programmer).
- 2. Increase the clock frequency (reduce the cycle time). Has huge power implications.
- 3. Reduce the CPI, i.e., execute more instructions in one cycle.
- We will talk about one technique that simultaneously achieves 2 & 3.

Limitations of a Single-Cycle CPU

Limitations of a Single-Cycle CPU

- Cycle time

Limitations of a Single-Cycle CPU

- Cycle time
 - Every instruction finishes in one cycle.

Limitations of a Single-Cycle CPU

- Cycle time
 - Every instruction finishes in one cycle.
 - The absolute time takes to execute each instruction varies.
Consider for instance an ADD instruction and a JMP instruction.

Limitations of a Single-Cycle CPU

- Cycle time
 - Every instruction finishes in one cycle.
 - The absolute time takes to execute each instruction varies. Consider for instance an ADD instruction and a JMP instruction.
 - But the cycle time is uniform across instructions, so the cycle time needs to accommodate the worst case, i.e., the slowest instruction.

Limitations of a Single-Cycle CPU

- Cycle time
 - Every instruction finishes in one cycle.
 - The absolute time takes to execute each instruction varies. Consider for instance an ADD instruction and a JMP instruction.
 - But the cycle time is uniform across instructions, so the cycle time needs to accommodate the worst case, i.e., the slowest instruction.
 - How do we shorten the cycle time (increase the frequency)?

Limitations of a Single-Cycle CPU

- Cycle time
 - Every instruction finishes in one cycle.
 - The absolute time takes to execute each instruction varies. Consider for instance an ADD instruction and a JMP instruction.
 - But the cycle time is uniform across instructions, so the cycle time needs to accommodate the worst case, i.e., the slowest instruction.
 - How do we shorten the cycle time (increase the frequency)?
- CPI

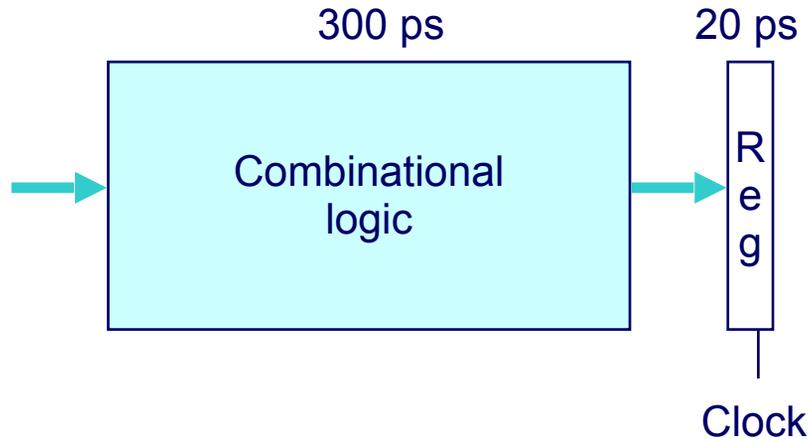
Limitations of a Single-Cycle CPU

- Cycle time
 - Every instruction finishes in one cycle.
 - The absolute time takes to execute each instruction varies. Consider for instance an ADD instruction and a JMP instruction.
 - But the cycle time is uniform across instructions, so the cycle time needs to accommodate the worst case, i.e., the slowest instruction.
 - How do we shorten the cycle time (increase the frequency)?
- CPI
 - The entire hardware is occupied to execute one instruction at a time. Can't execute multiple instructions at the same time.

Limitations of a Single-Cycle CPU

- Cycle time
 - Every instruction finishes in one cycle.
 - The absolute time takes to execute each instruction varies. Consider for instance an ADD instruction and a JMP instruction.
 - But the cycle time is uniform across instructions, so the cycle time needs to accommodate the worst case, i.e., the slowest instruction.
 - How do we shorten the cycle time (increase the frequency)?
- CPI
 - The entire hardware is occupied to execute one instruction at a time. Can't execute multiple instructions at the same time.
 - How do execute multiple instructions in one cycle?

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

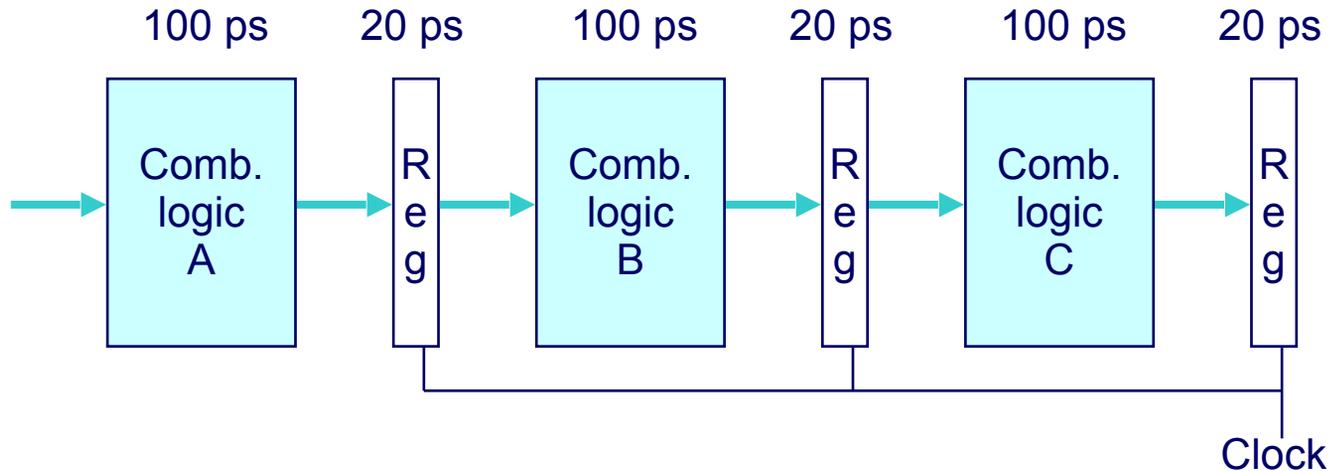
Pipeline Diagrams

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



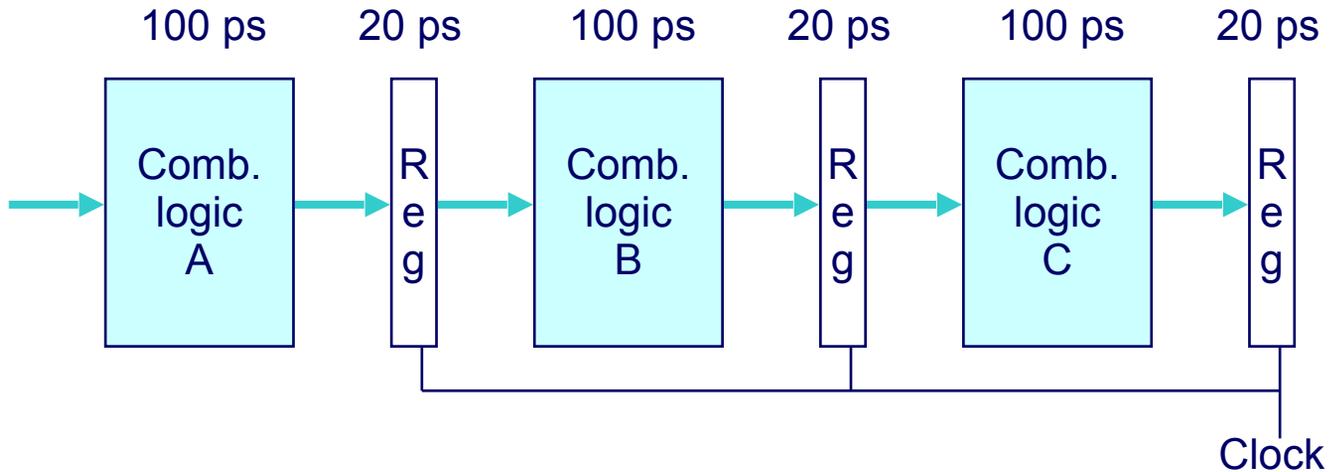
- 3 instructions will take 960 ps to finish
 - First cycle: Inst 1 takes 300 ps to compute new state, 20 ps to store the new states
 - Second cycle: Inst 2 starts; it takes 300 ps to compute new states, 20 ps to store new states
 - And so on...

3-Stage Pipelined Version

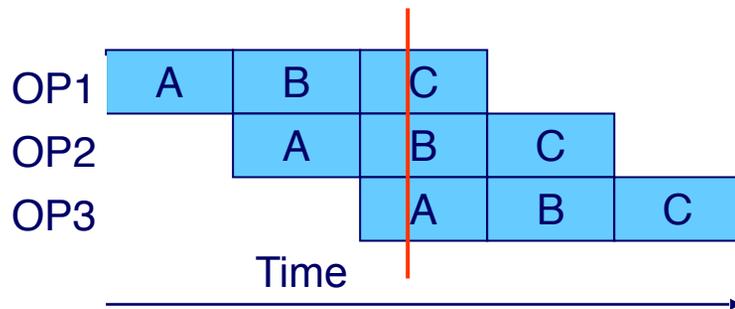


- Divide combinational logic into 3 stages of 100 ps each
- Insert registers between stages to store intermediate data between stages. These are called pipeline registers (ISA-invisible)
- Can begin a new instruction as soon as the previous one finishes stage A and has stored the intermediate data.
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

3-Stage Pipelined Version



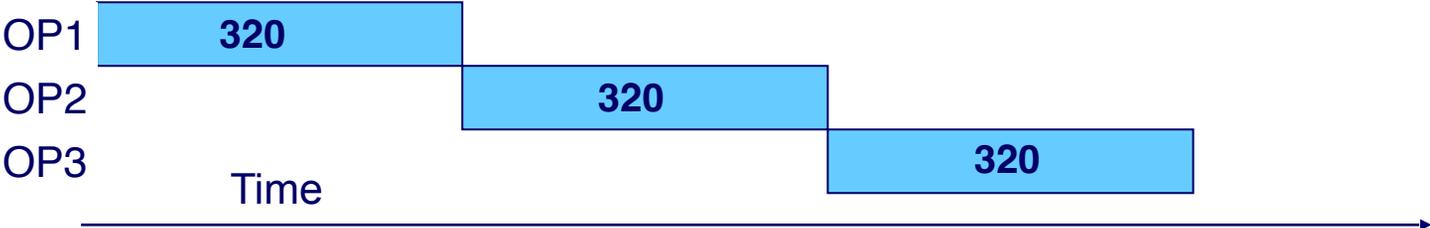
3-Stage Pipelined



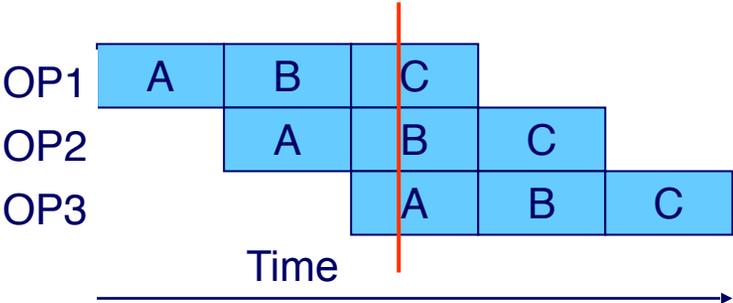
Comparison

Unpipelined

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



3-Stage Pipelined



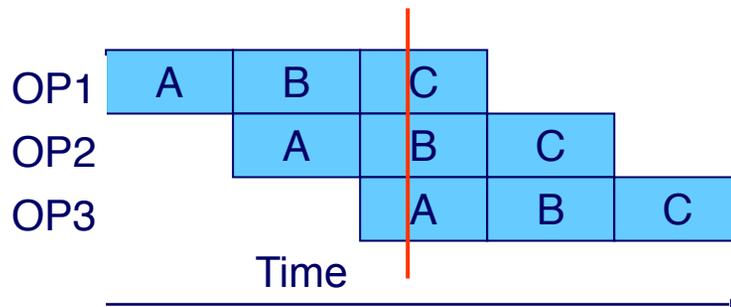
- Time to finish 3 insets = $120 * 5 = 600$ ps
- But each inst.'s latency increases: $120 * 3 = 360$ ps

Benefits of Pipelining

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



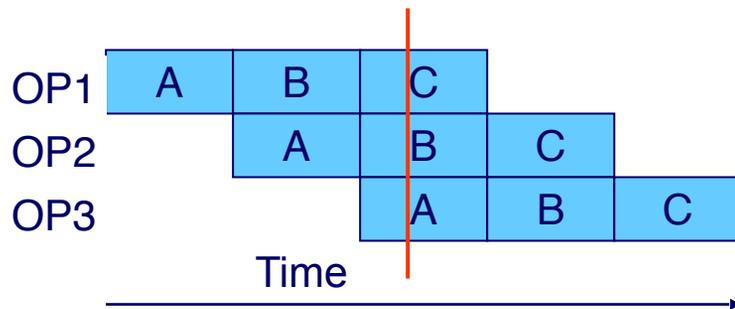
Reduce the cycle time from 320 ps to 120 ps



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

One Requirement of Pipelining

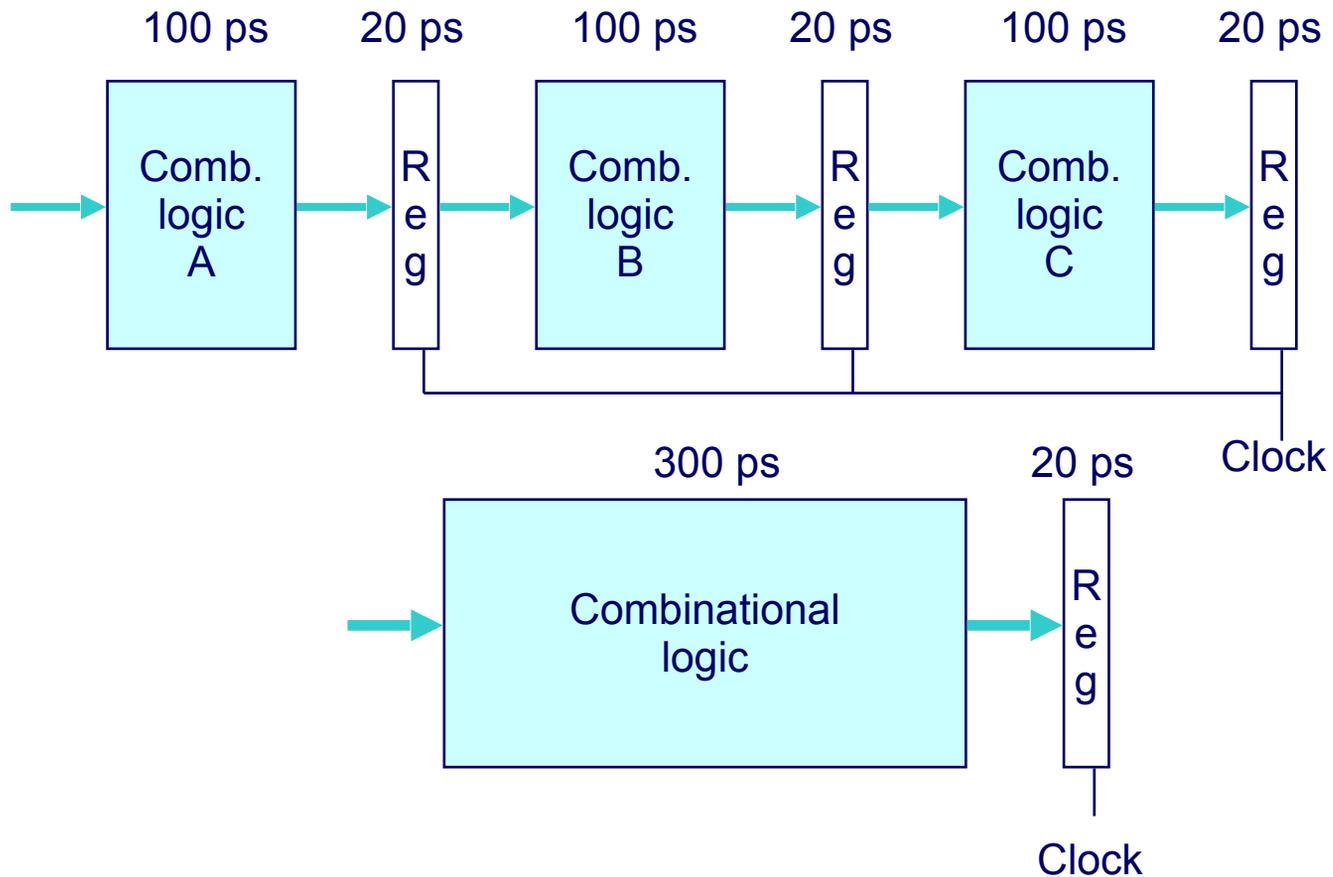
- The stages need to be using different hardware structures.
- That is, Stage A, Stage B, and Stage C need to exercise different parts of the combination logic.



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

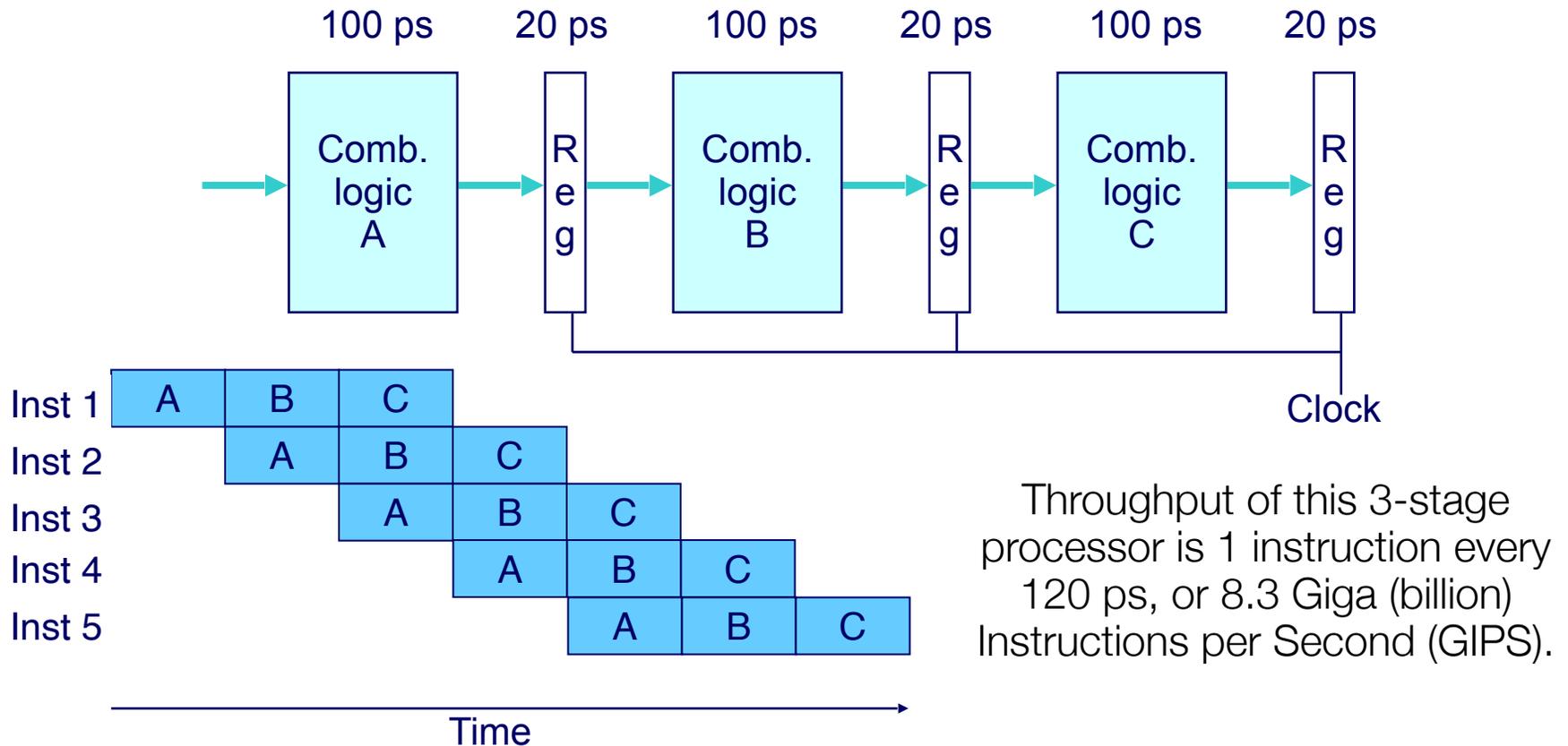
Pipeline Trade-offs

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



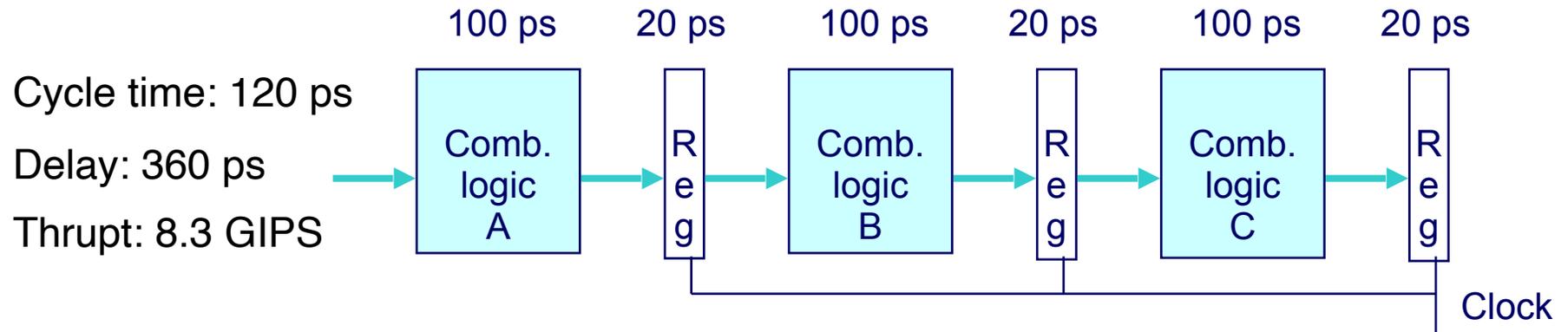
Throughput

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



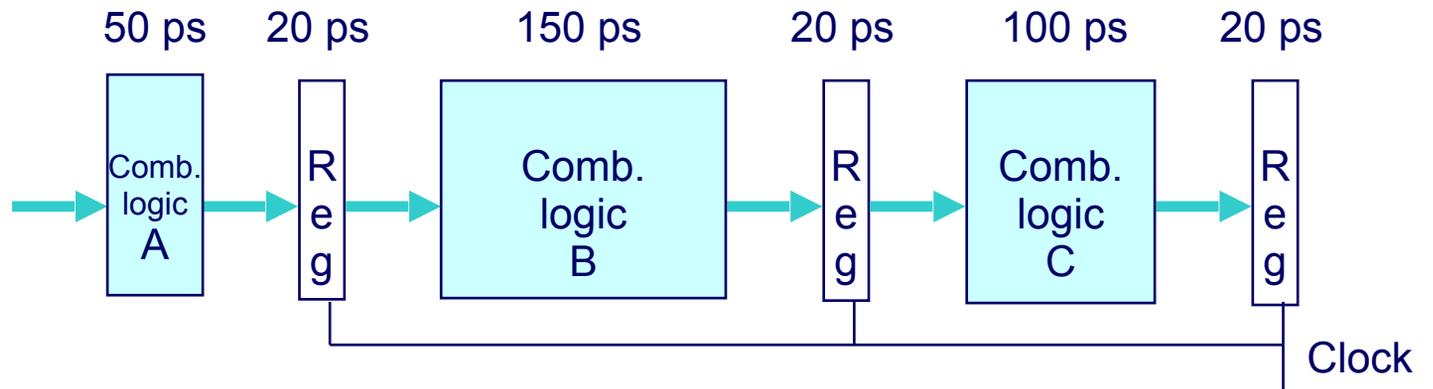
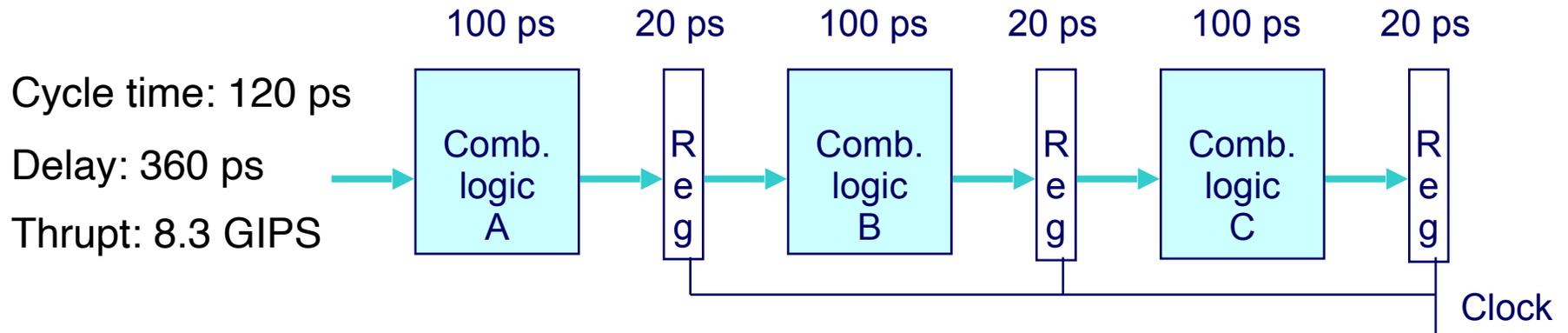
Aside: Unbalanced Pipeline

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



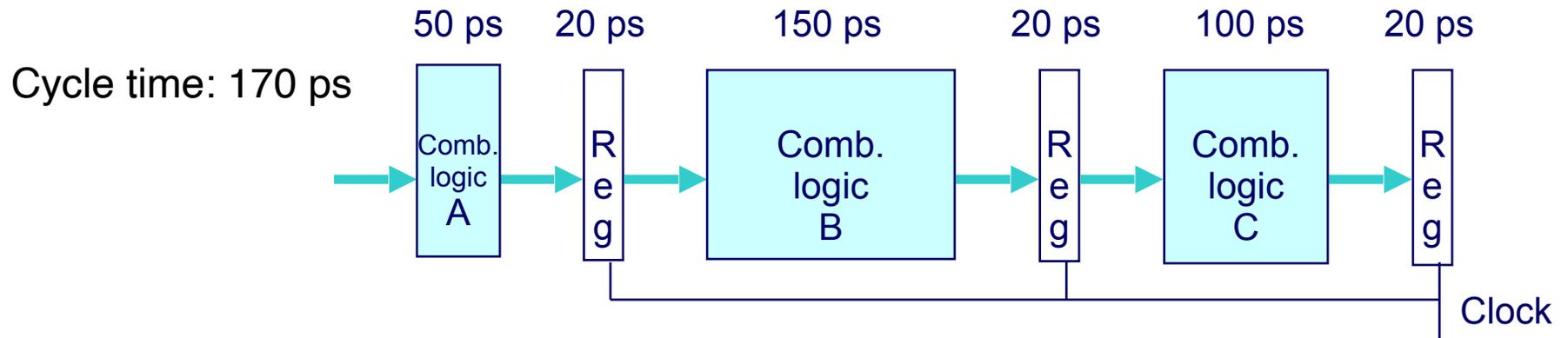
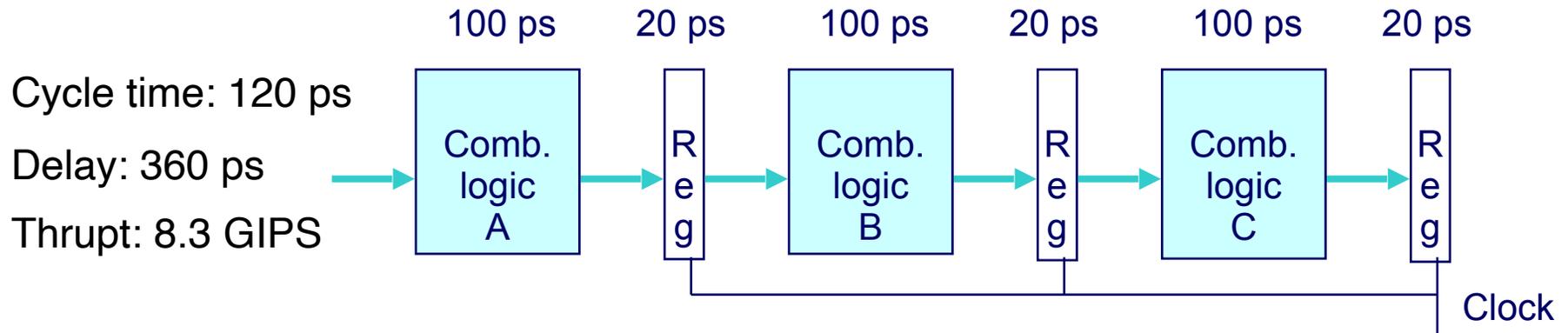
Aside: Unbalanced Pipeline

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



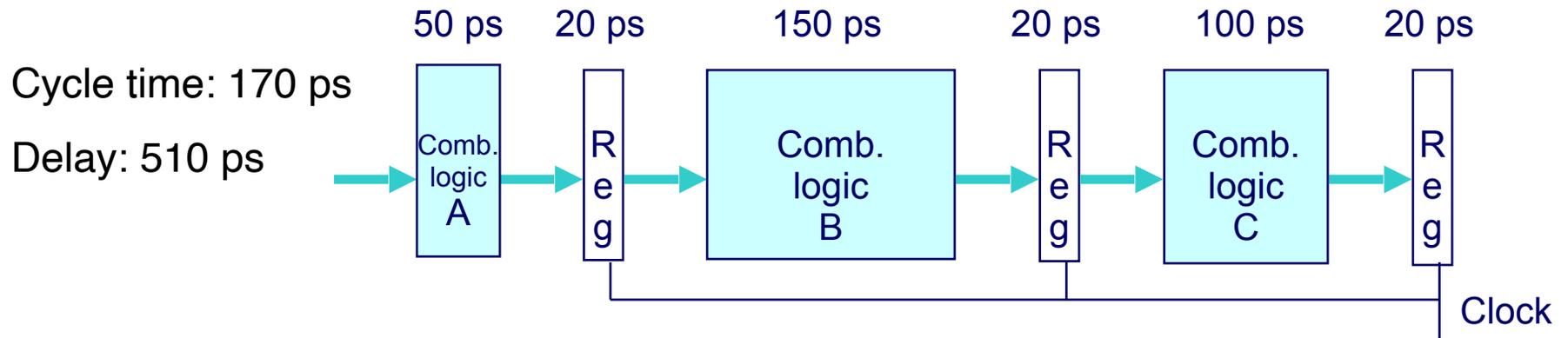
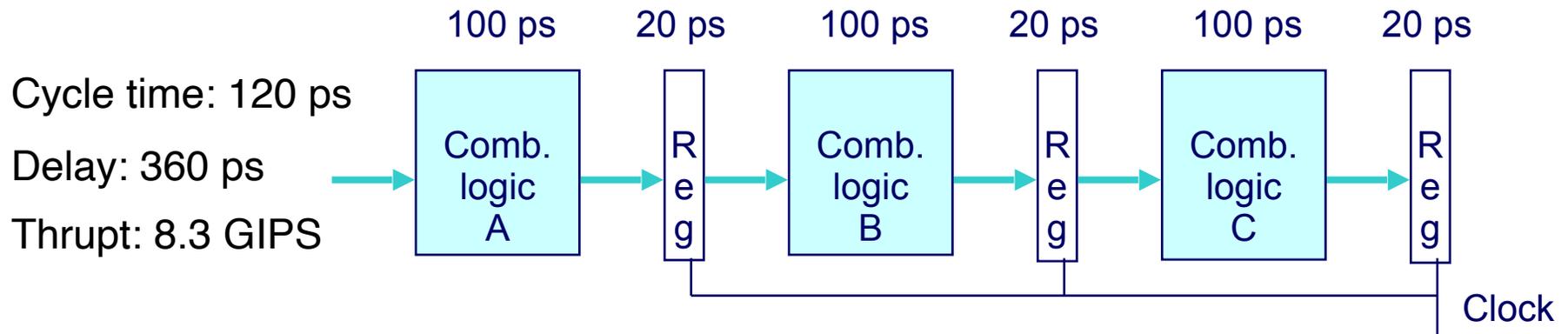
Aside: Unbalanced Pipeline

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



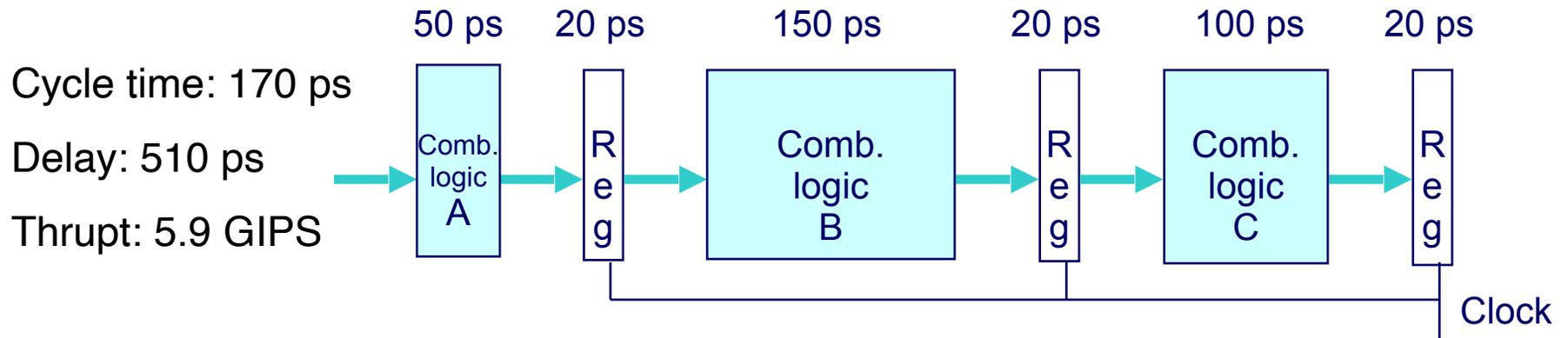
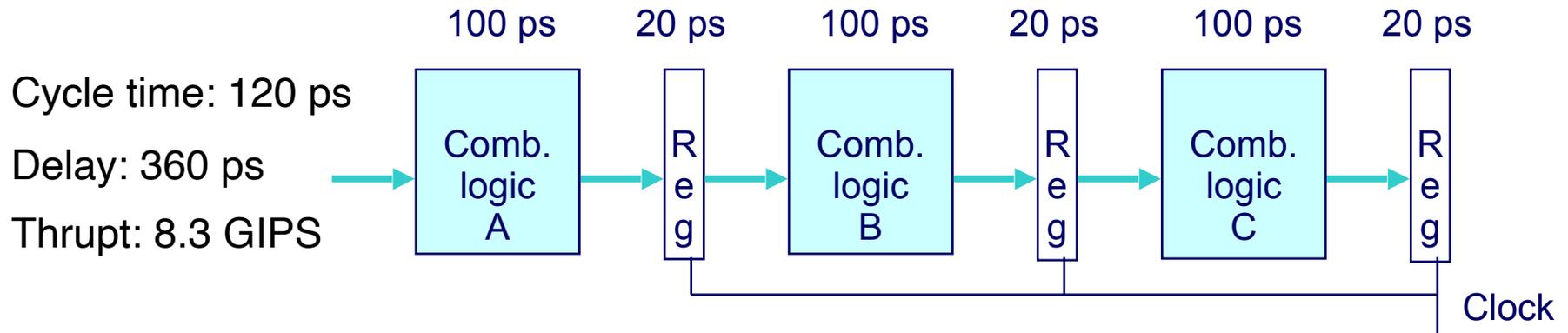
Aside: Unbalanced Pipeline

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



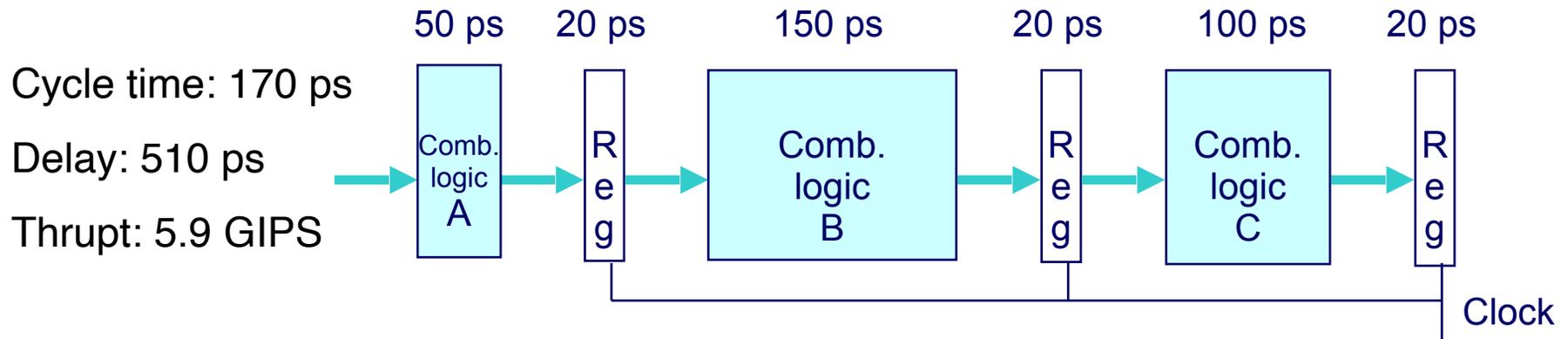
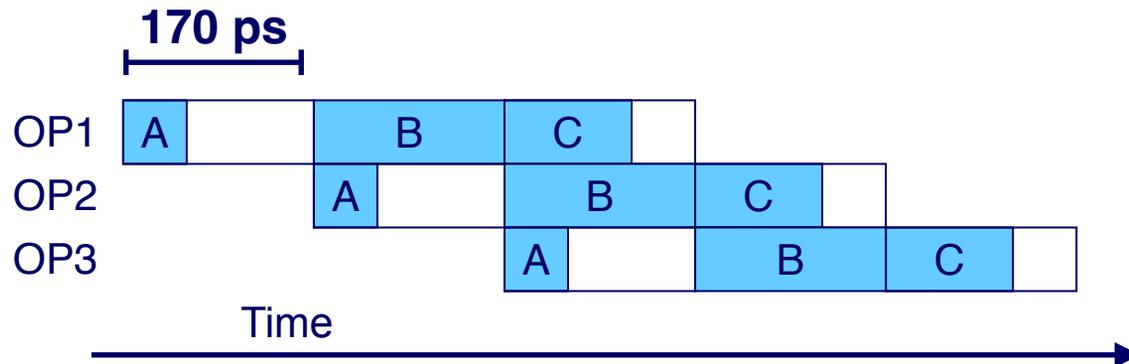
Aside: Unbalanced Pipeline

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



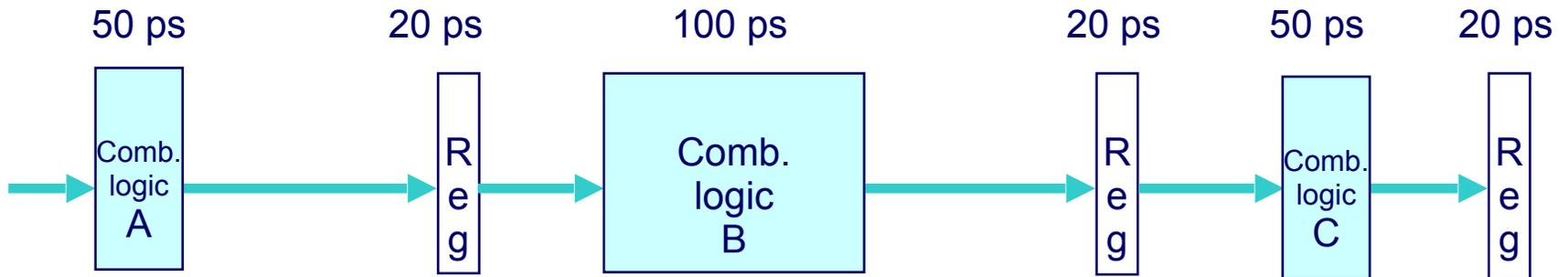
Aside: Unbalanced Pipeline

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



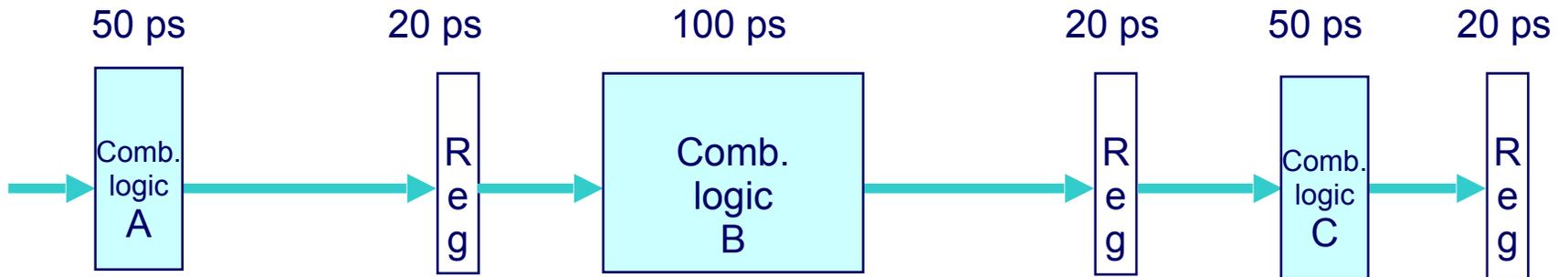
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages



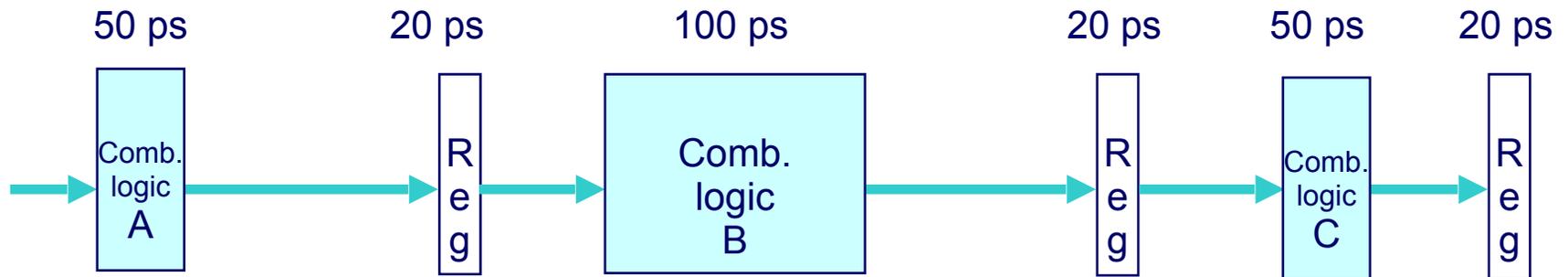
Aside: Mitigating Unbalanced Pipeline

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



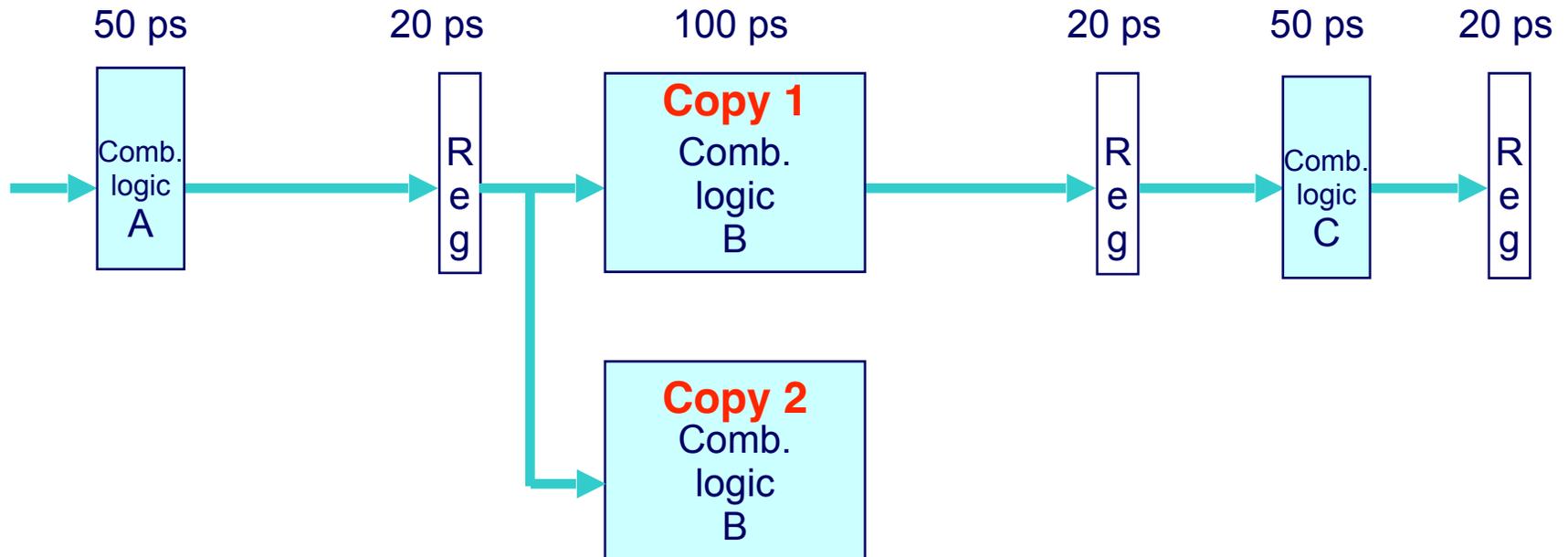
Aside: Mitigating Unbalanced Pipeline

- 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



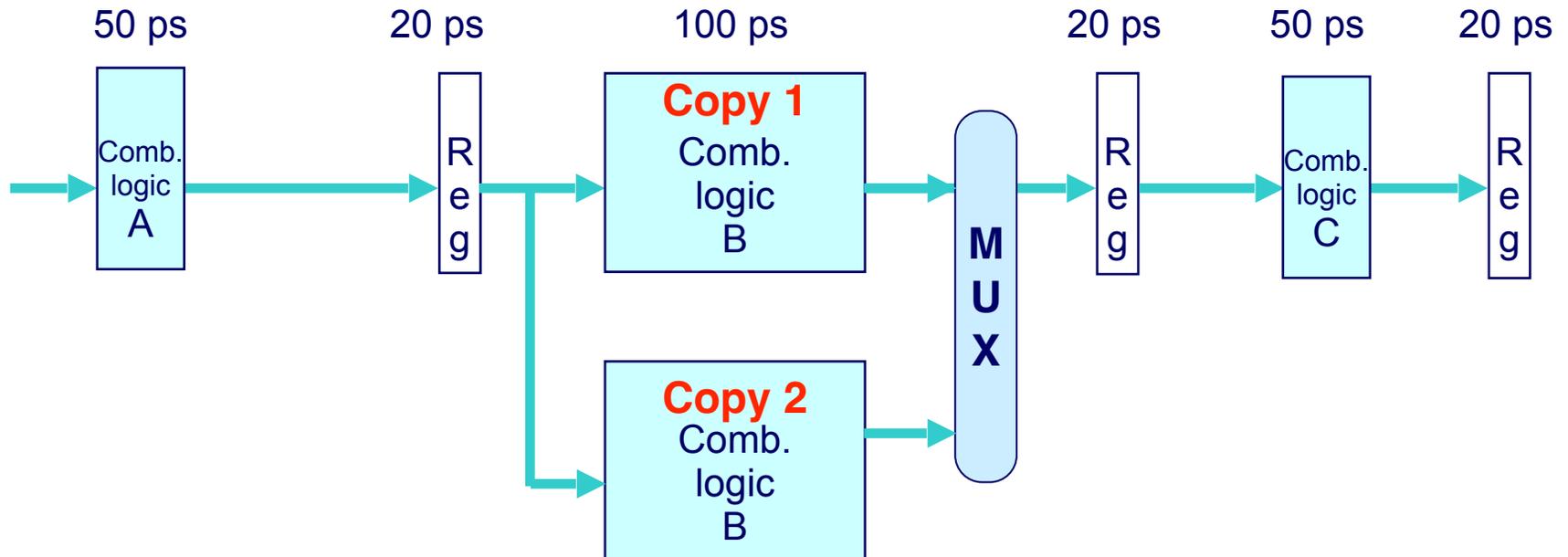
Aside: Mitigating Unbalanced Pipeline

- 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



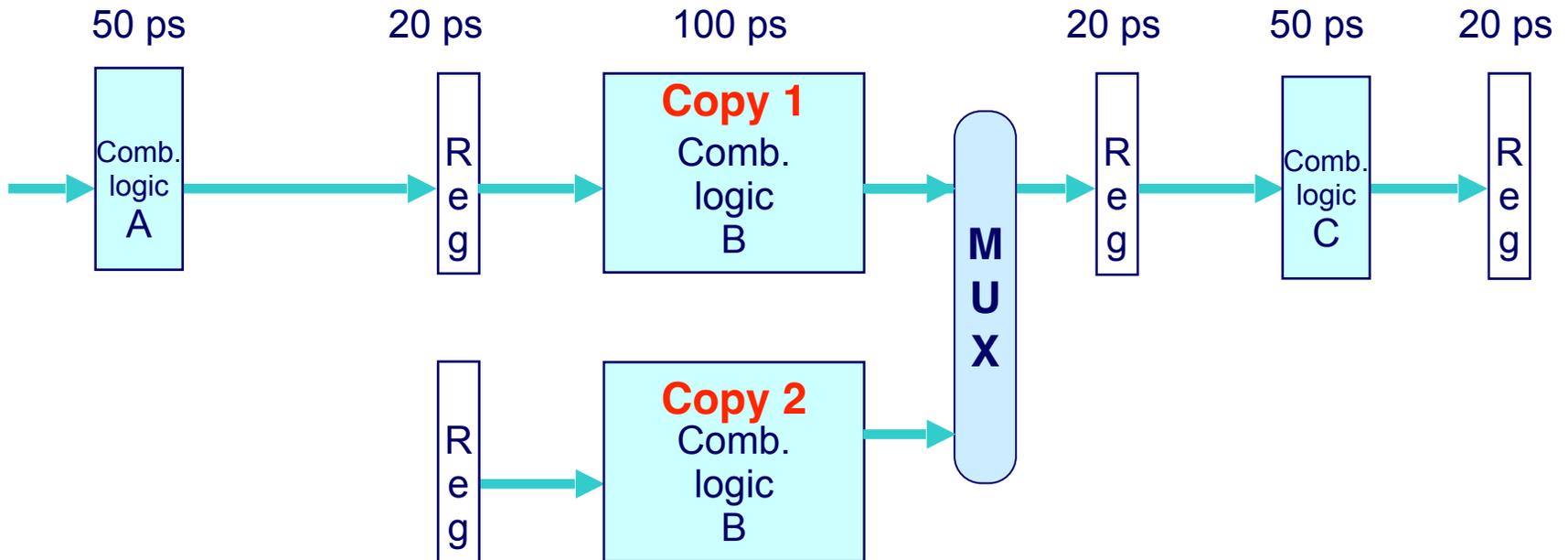
Aside: Mitigating Unbalanced Pipeline

- 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



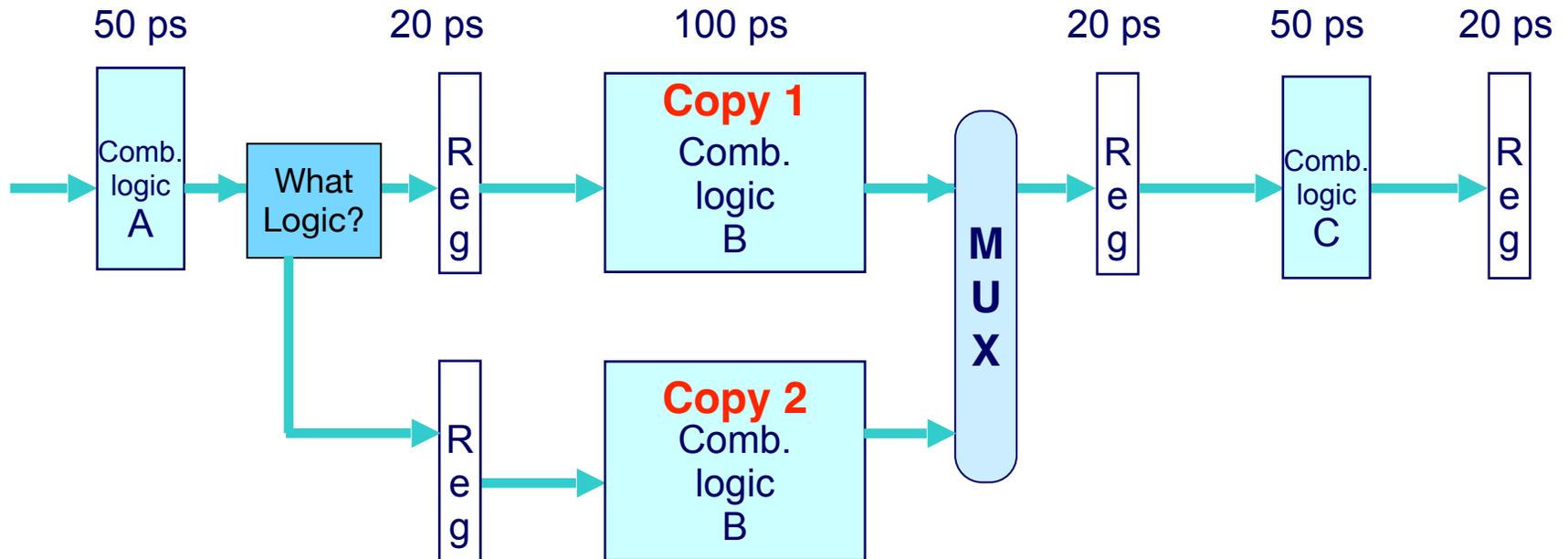
Aside: Mitigating Unbalanced Pipeline

- 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



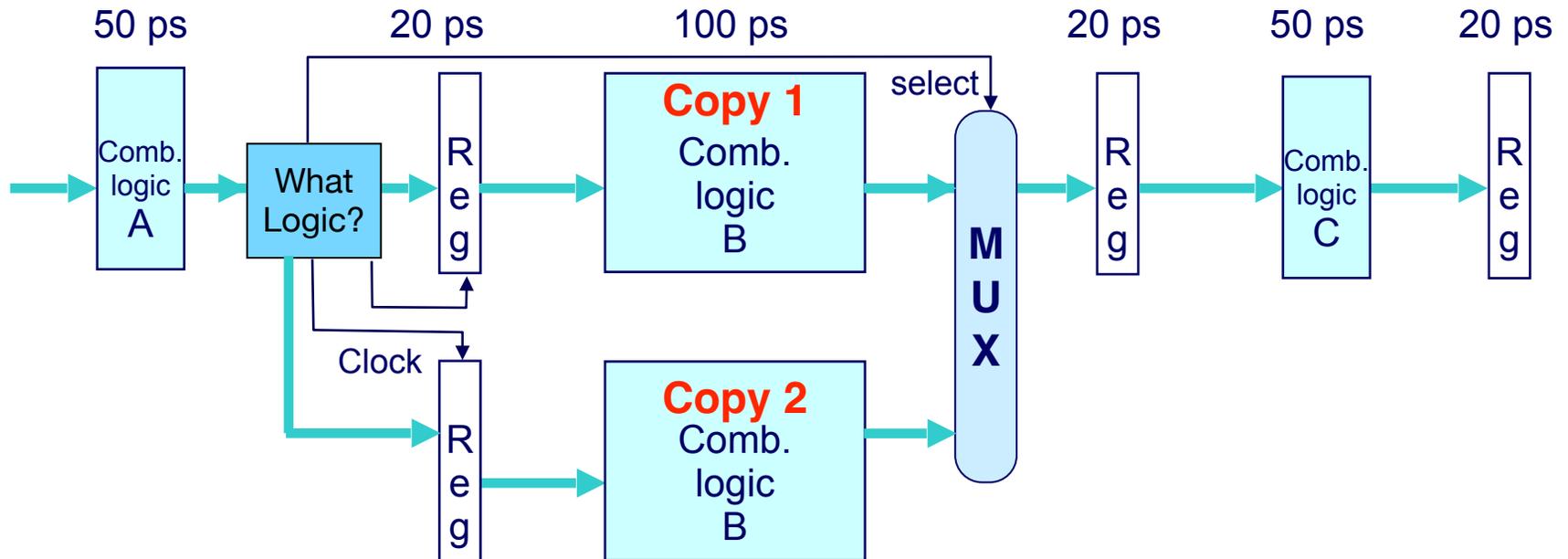
Aside: Mitigating Unbalanced Pipeline

- 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



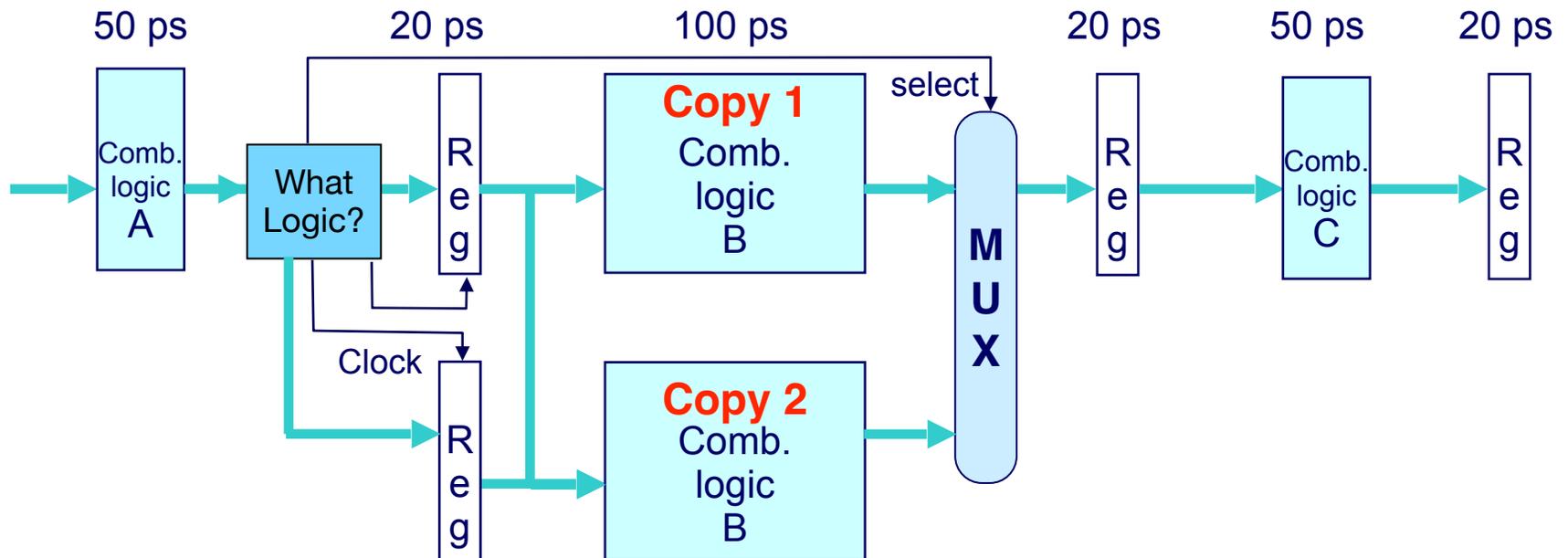
Aside: Mitigating Unbalanced Pipeline

- 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



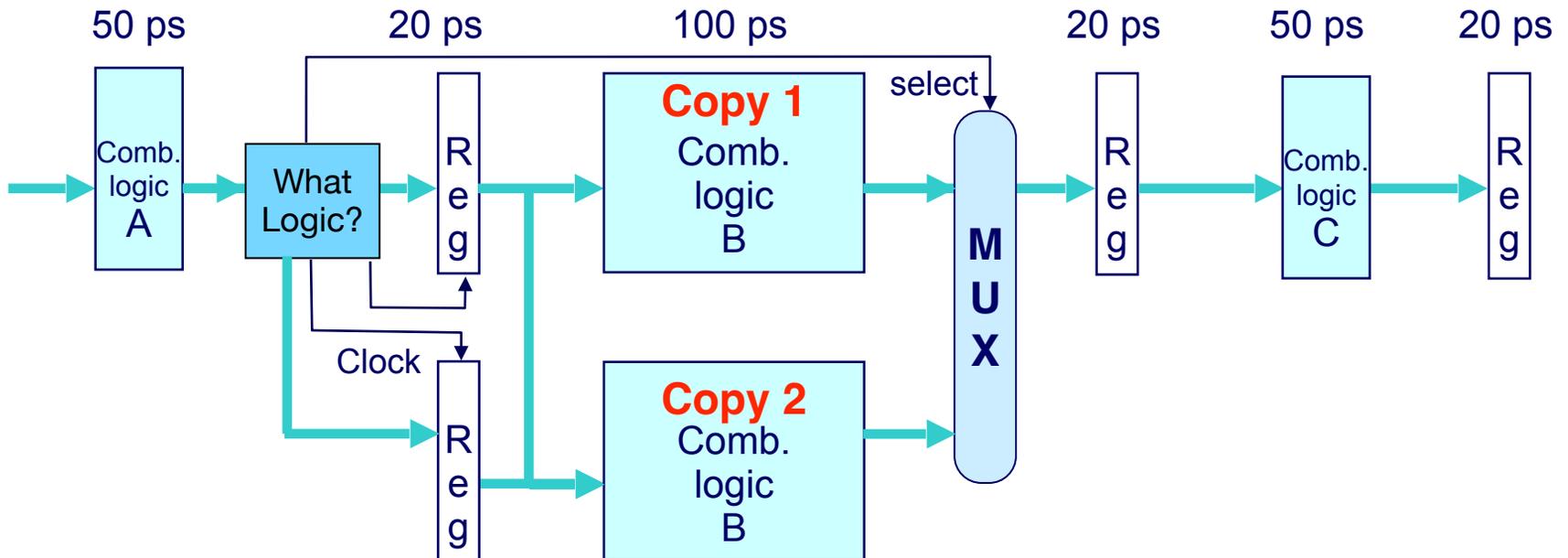
Aside: Mitigating Unbalanced Pipeline

- Data sent to copy 1 in odd cycles and to copy 2 in even cycles.



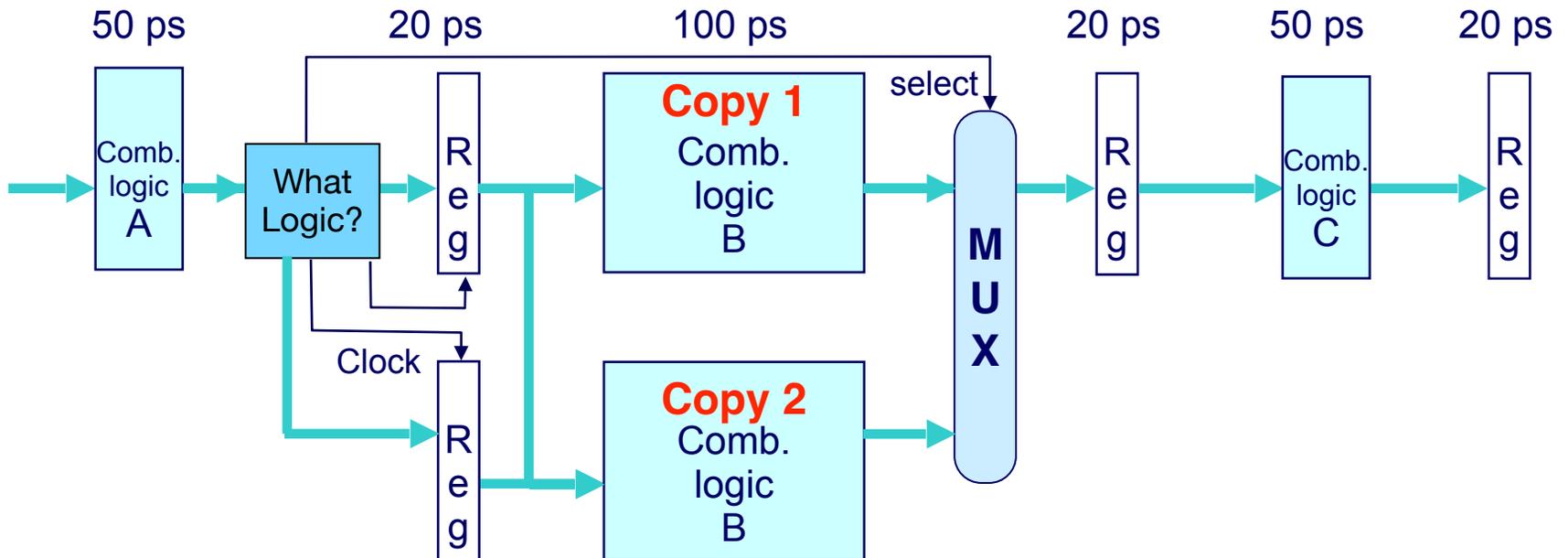
Aside: Mitigating Unbalanced Pipeline

- 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.



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

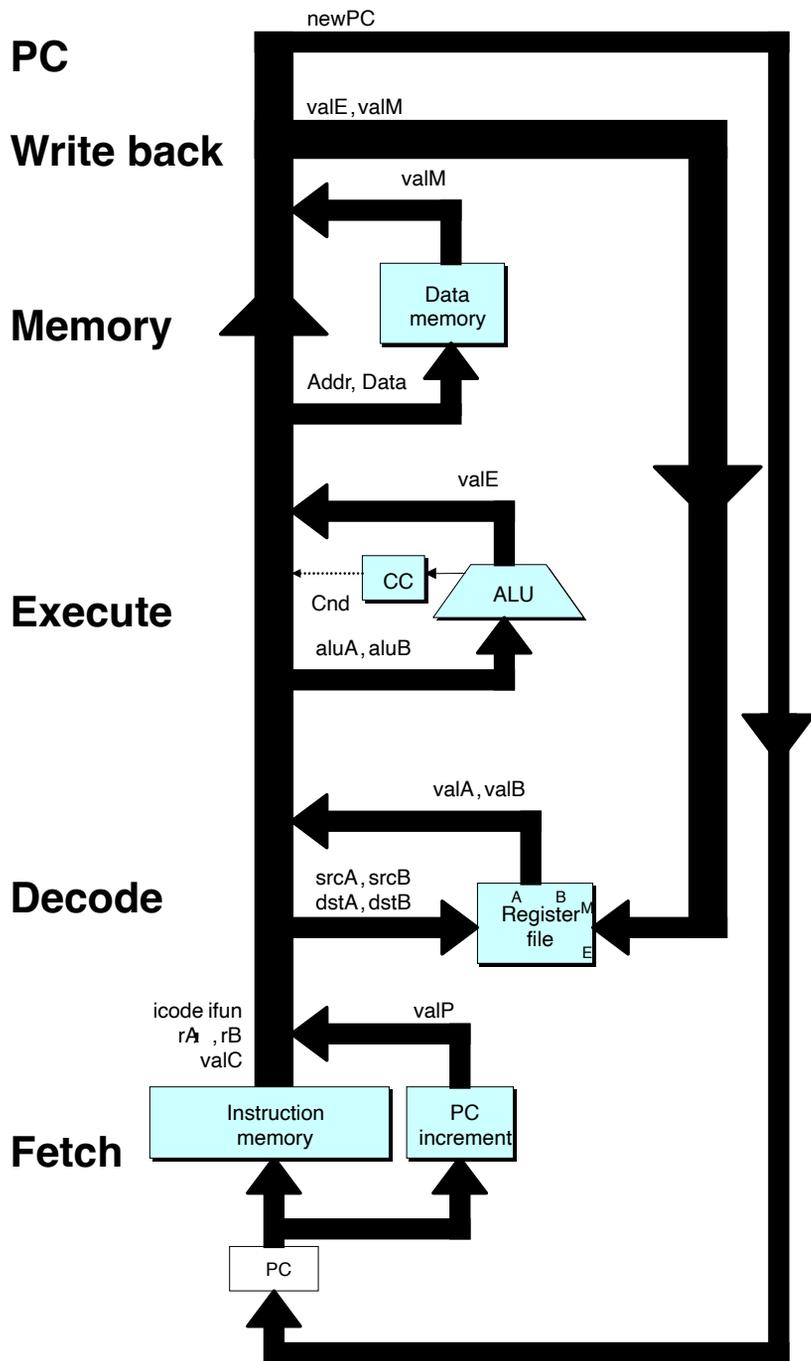
Decode: Read program registers

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



Stage Computation: Arith/Log. Ops



	$OPq\ rA,\ rB$	
Fetch	$icode:ifun \leftarrow M_1[PC]$	Read instruction byte
	$rA:rB \leftarrow M_1[PC+1]$	Read register byte
	$valP \leftarrow PC+2$	Compute next PC

Stage Computation: Arith/Log. Ops



	$OPq \ rA, \ rB$	
Fetch	$icode:ifun \leftarrow M_1[PC]$	Read instruction byte
	$rA:rB \leftarrow M_1[PC+1]$	Read register byte
	$valP \leftarrow PC+2$	Compute next PC
Decode	$valA \leftarrow R[rA]$	Read operand A
	$valB \leftarrow R[rB]$	Read operand B

Stage Computation: Arith/Log. Ops



	OPq rA, rB	
Fetch	icode:ifun \leftarrow M ₁ [PC] rA:rB \leftarrow M ₁ [PC+1]	Read instruction byte Read register byte
	valP \leftarrow PC+2	Compute next PC
Decode	valA \leftarrow R[rA] valB \leftarrow R[rB]	Read operand A Read operand B
	valE \leftarrow valB OP valA Set CC	Perform ALU operation Set condition code register

Stage Computation: Arith/Log. Ops



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

Stage Computation: Arith/Log. Ops



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

Stage Computation: Arith/Log. Ops



	OPq rA, rB	
Fetch	icode:ifun \leftarrow M ₁ [PC] rA:rB \leftarrow M ₁ [PC+1]	Read instruction byte Read register byte
	valP \leftarrow PC+2	Compute next PC
Decode	valA \leftarrow R[rA]	Read operand A
	valB \leftarrow R[rB]	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)`

4	0	rA	rB
---	---	----	----

D

<code>rmmovq rA, D(rB)</code>

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



	<code>rmmovq rA, D(rB)</code>
Fetch	<code>icode:ifun ← M₁[PC]</code> <code>rA:rB ← M₁[PC+1]</code> <code>valC ← M₈[PC+2]</code> <code>valP ← PC+10</code>

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



<code>rmmovq rA, D(rB)</code>	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



<code>rmmovq rA, D(rB)</code>	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB + valC$

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B
- Compute effective address

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



<code>rmmovq rA, D(rB)</code>	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB + valC$
Memory	$M_8[valE] \leftarrow valA$

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B
- Compute effective address
- Write value to memory

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



<code>rmmovq rA, D(rB)</code>	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB + valC$
Memory	$M_8[valE] \leftarrow valA$
Write back	

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B
- Compute effective address
- Write value to memory

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



<code>rmmovq rA, D(rB)</code>	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB + valC$
Memory	$M_8[valE] \leftarrow valA$
Write back	
PC update	$PC \leftarrow valP$

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B
- Compute effective address
- Write value to memory
- Update PC

Stage Computation: Jumps

jXX Dest

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

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

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

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		

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

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?

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

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		

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

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		

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

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

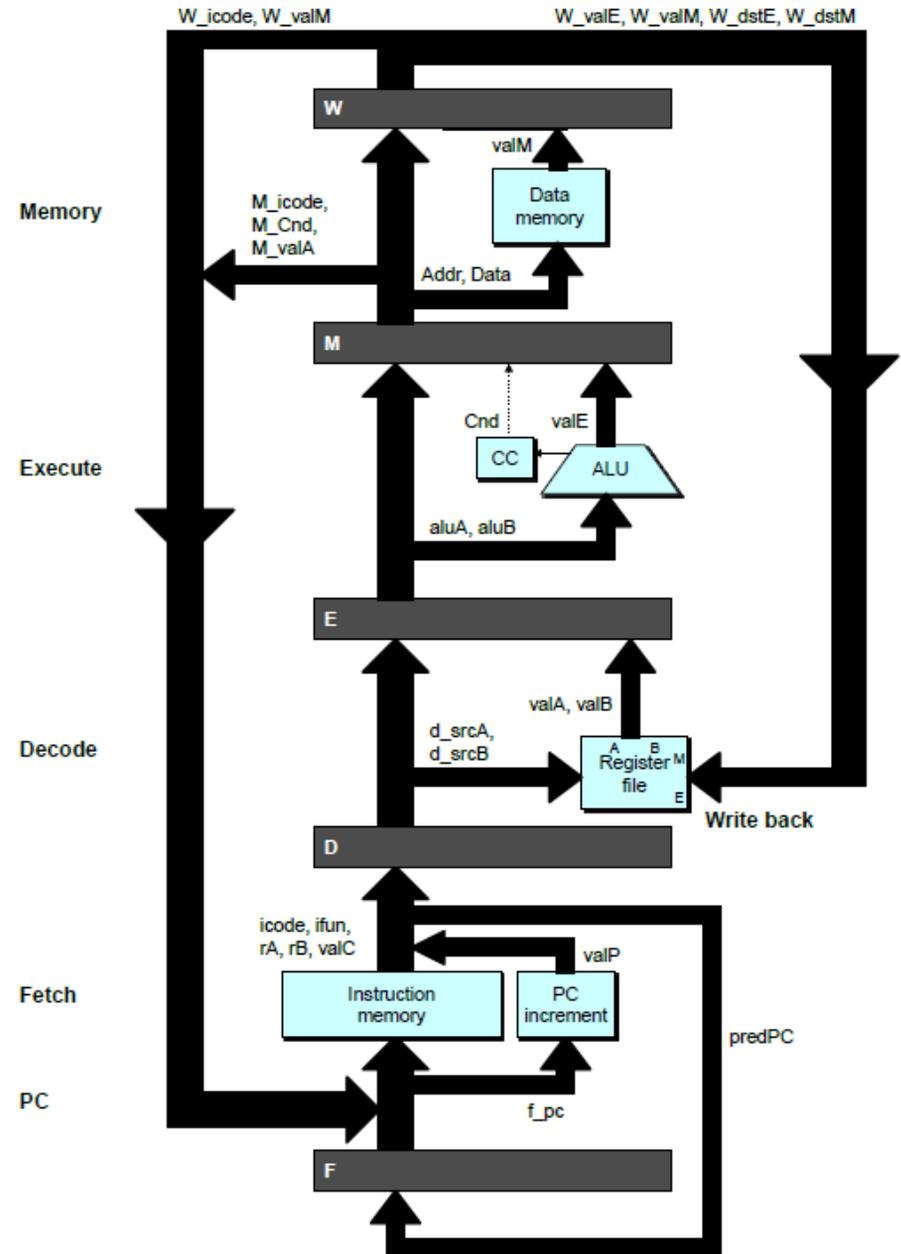
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file



Real-World Pipelines: Car Washes

Real-World Pipelines: Car Washes

Sequential



Real-World Pipelines: Car Washes

Sequential



Pipelined



Real-World Pipelines: Car Washes

Sequential



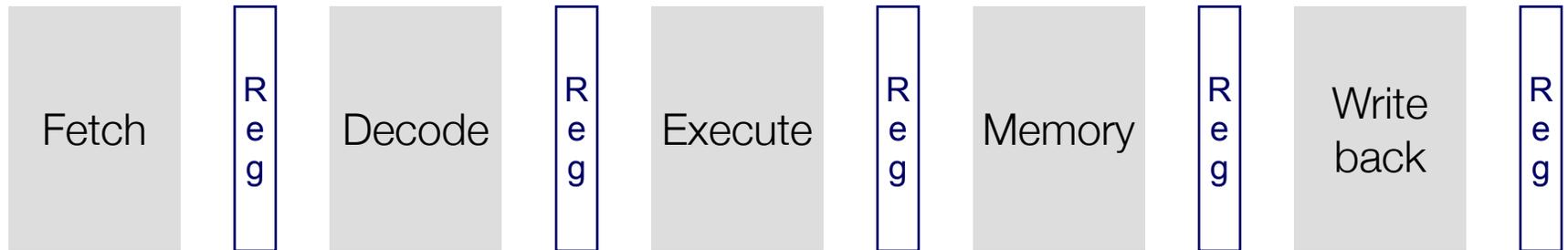
Pipelined



Idea

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

Pipeline Illustration

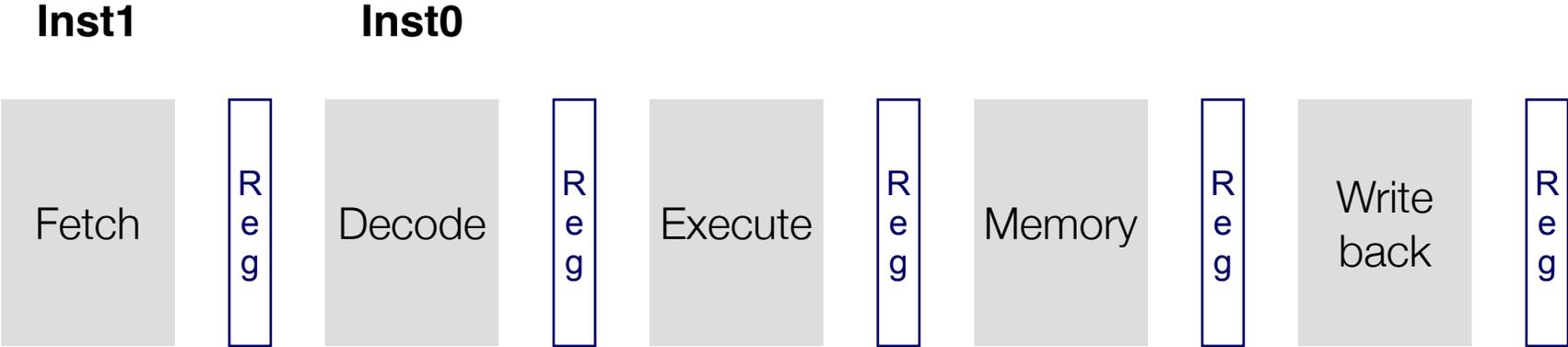


Pipeline Illustration

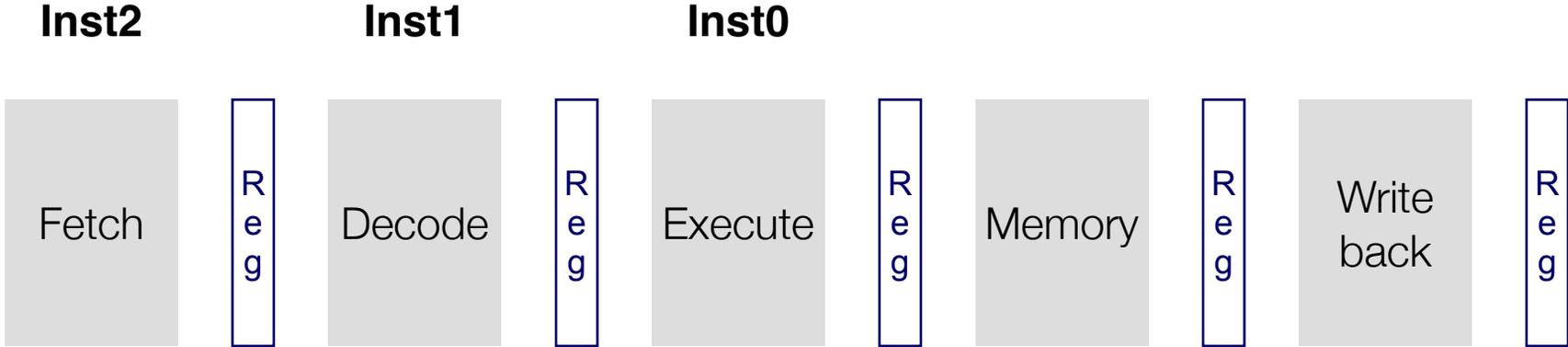
Inst0



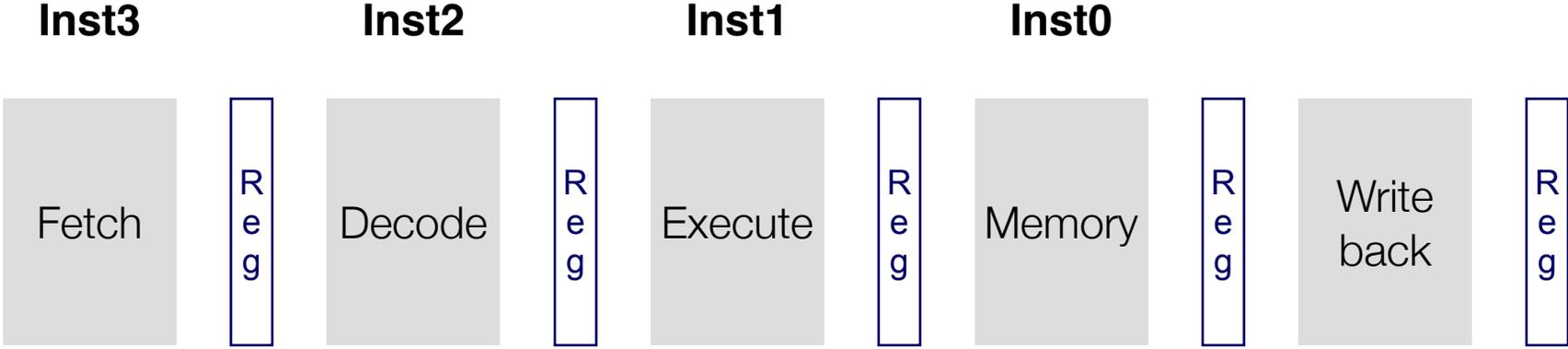
Pipeline Illustration



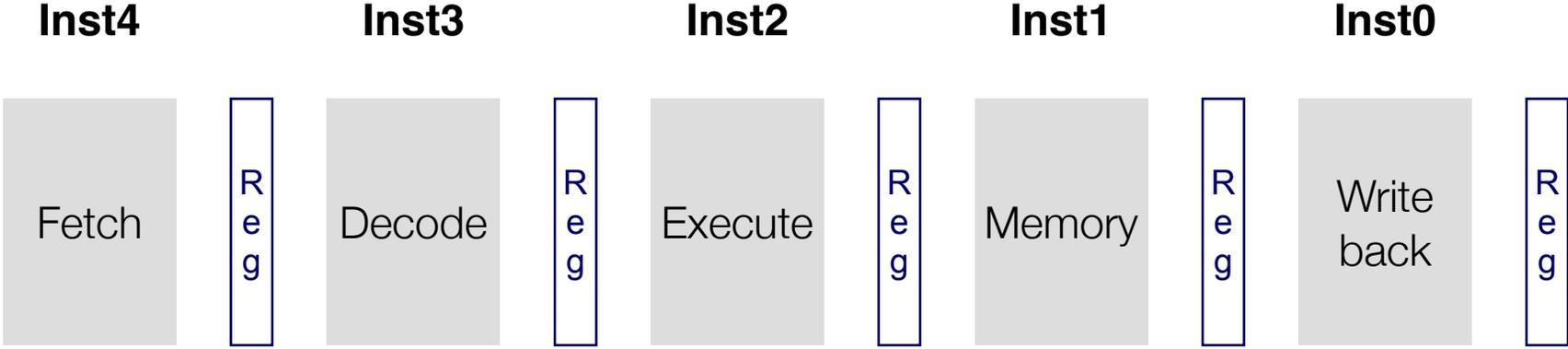
Pipeline Illustration



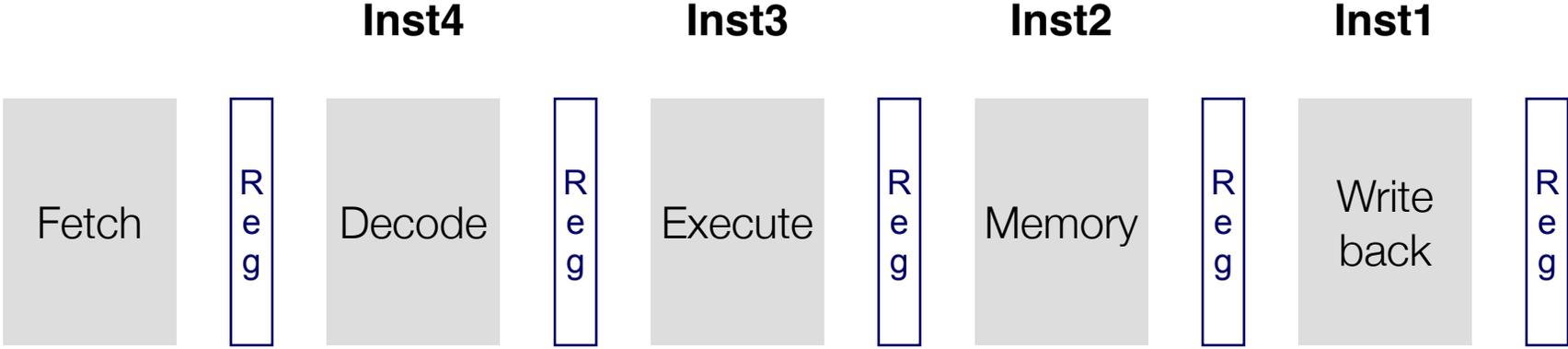
Pipeline Illustration



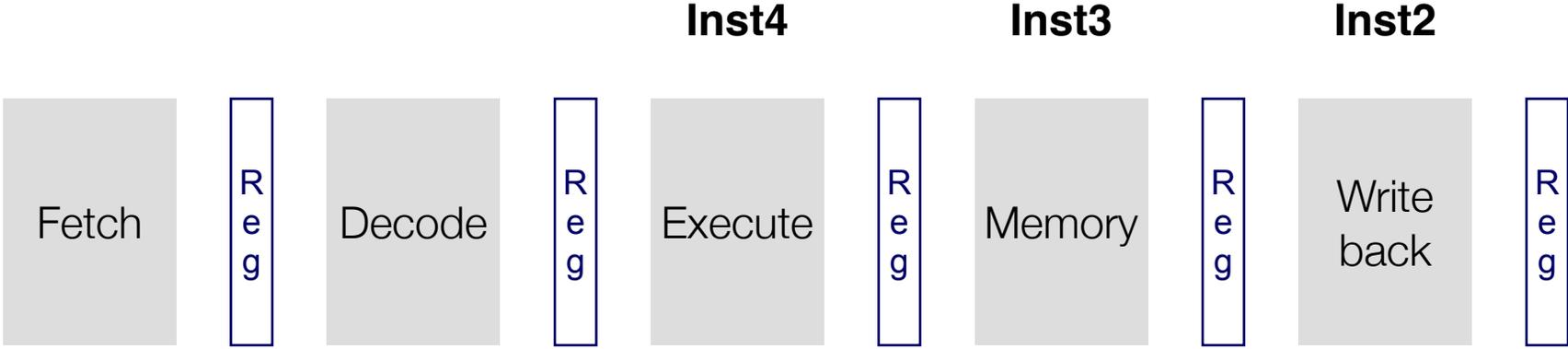
Pipeline Illustration



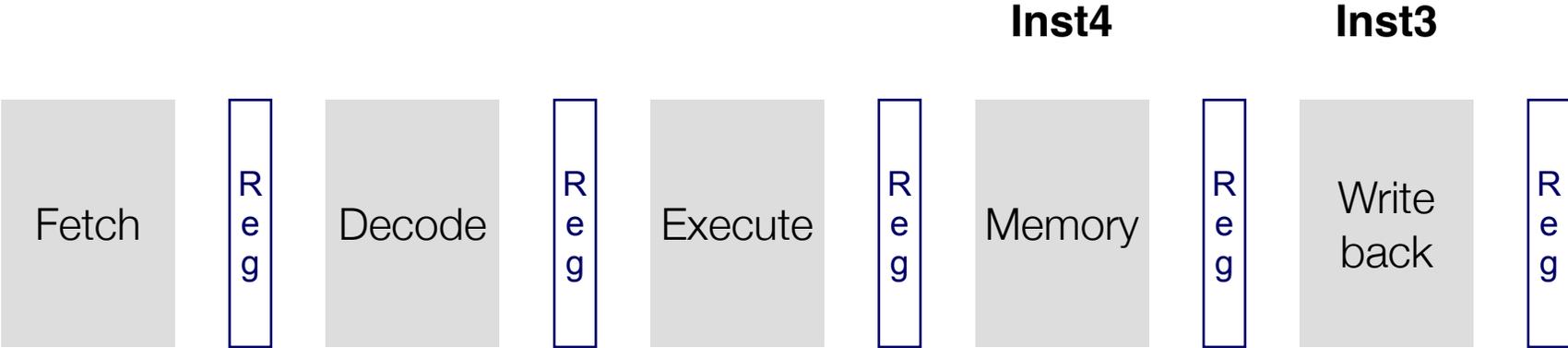
Pipeline Illustration



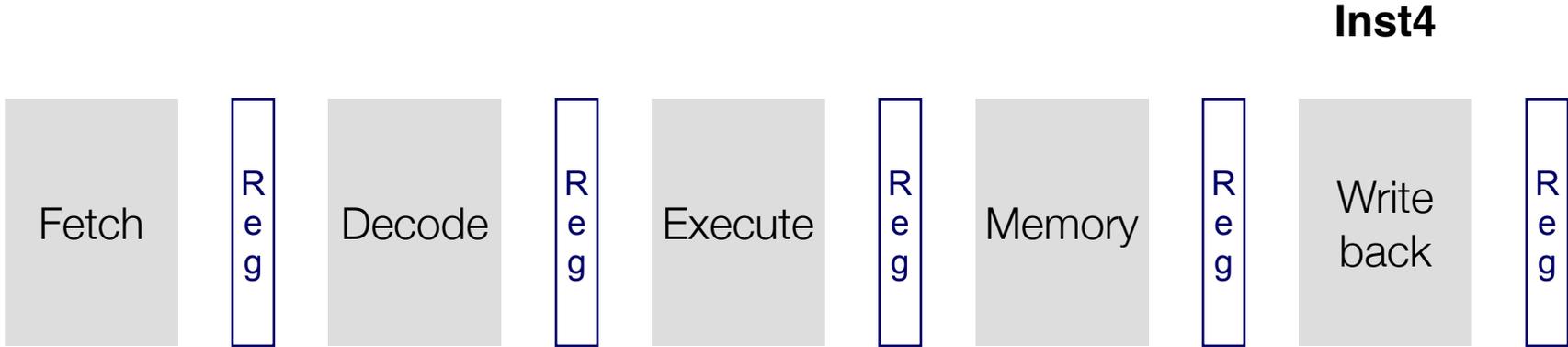
Pipeline Illustration



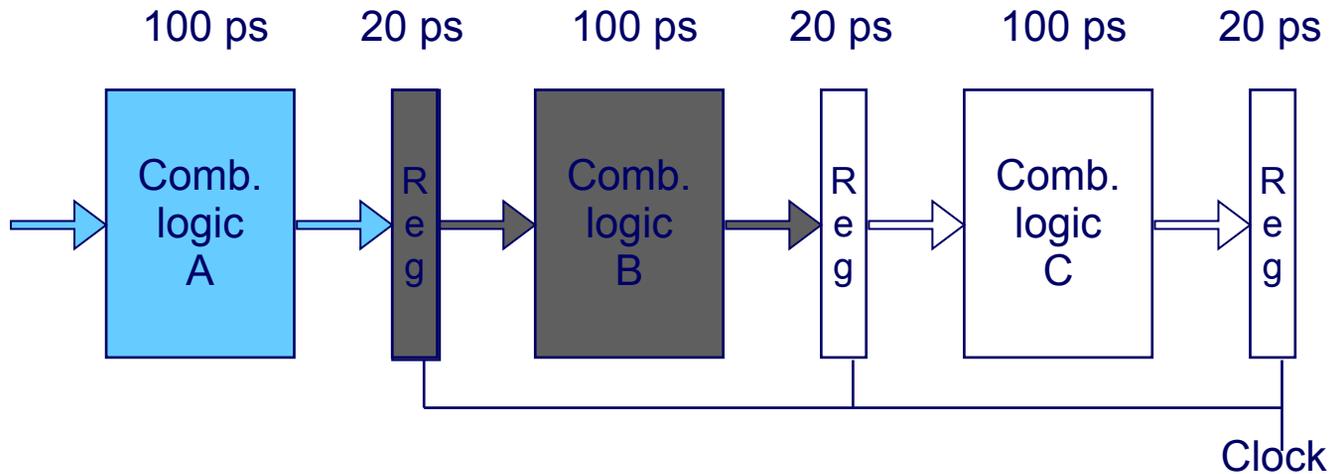
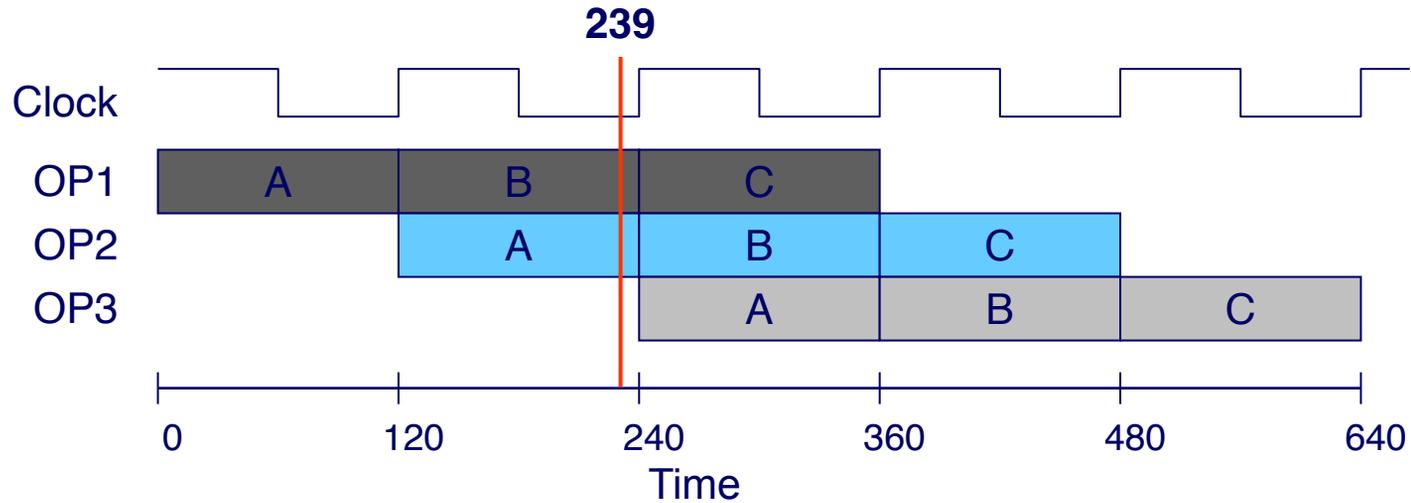
Pipeline Illustration



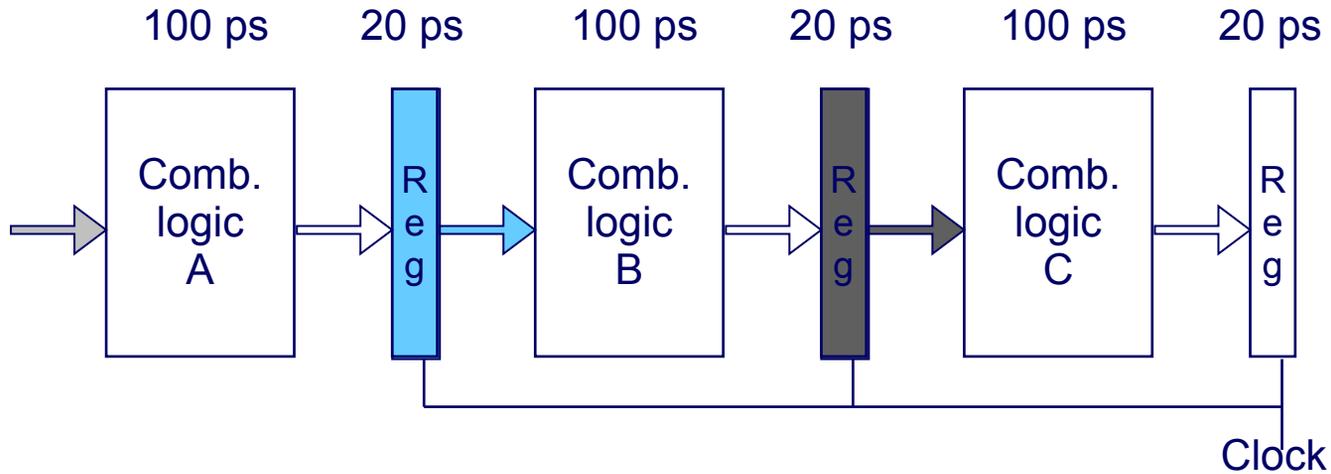
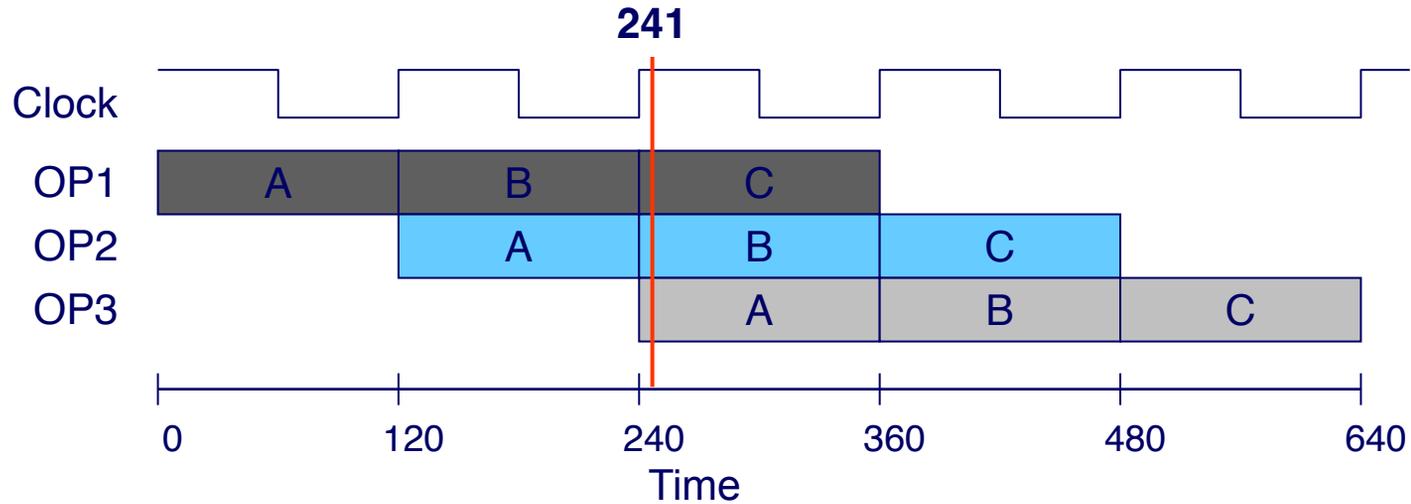
Pipeline Illustration



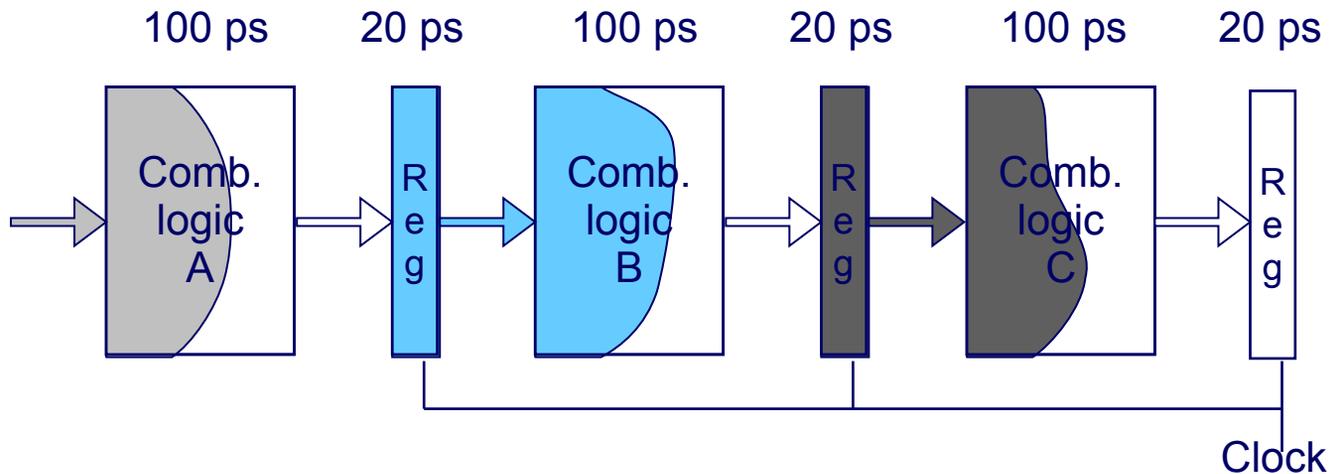
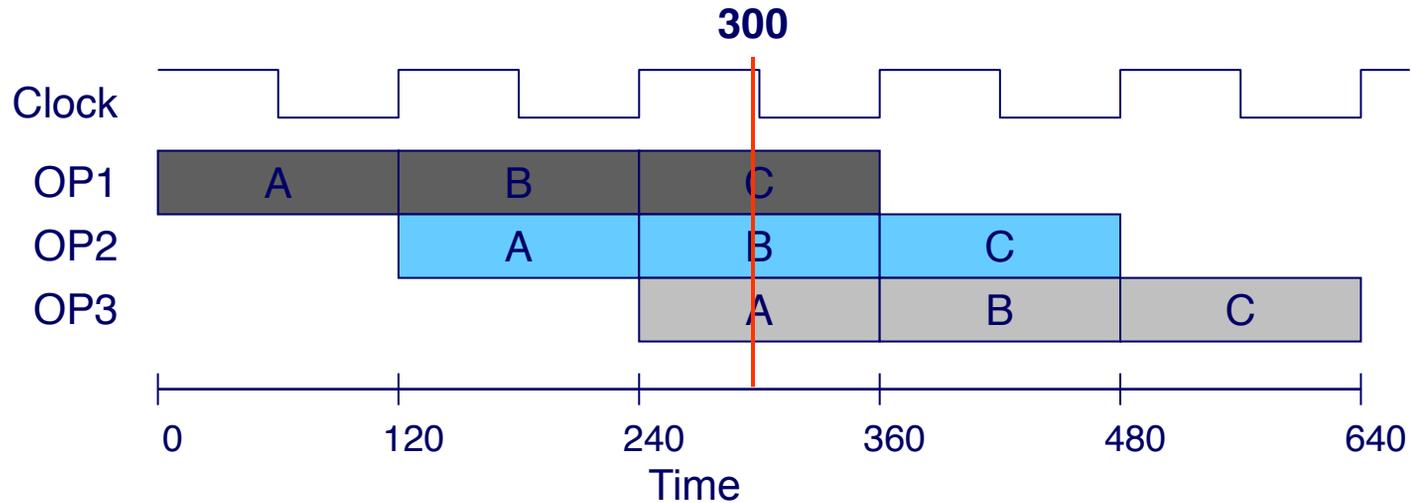
Another Illustration



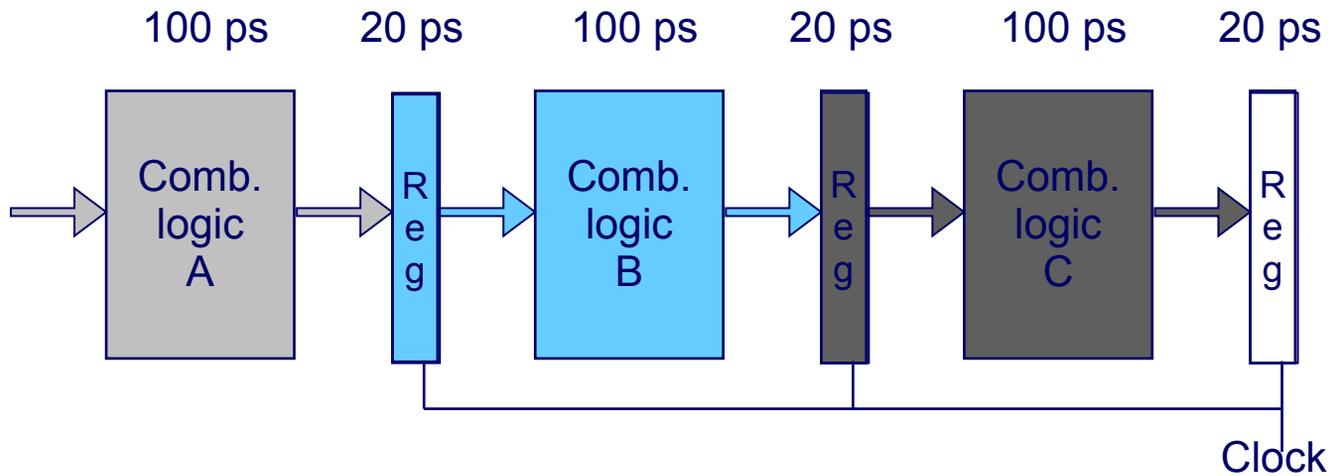
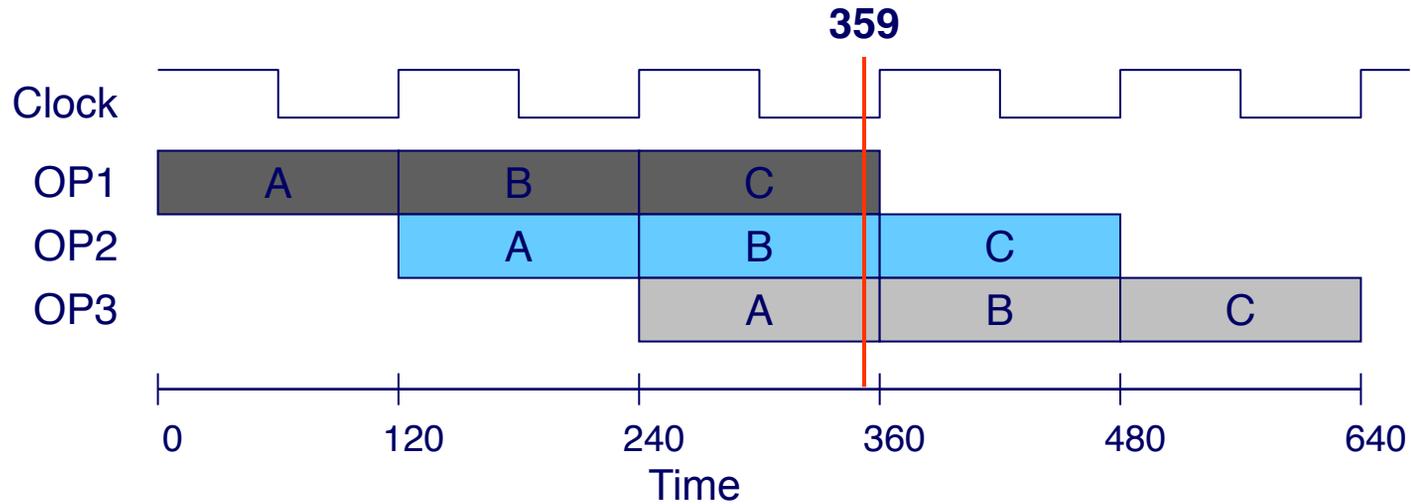
Another Illustration



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.
- 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

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

Control Dependency

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed.
- 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

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

1

F

Control Dependency

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed.
- 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

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

	1	2
	F	D
		F

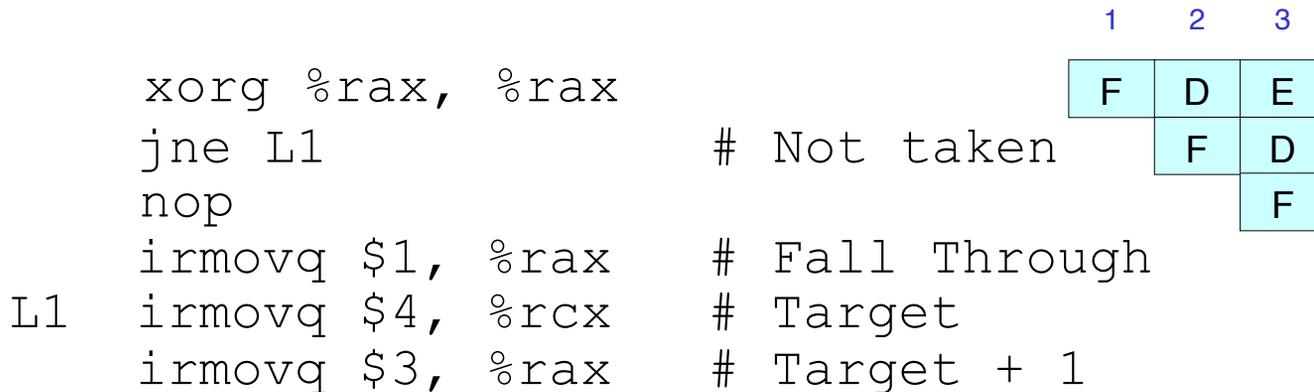
Control Dependency

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed.
- 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
			F	D	E
				F	D
	<code>xorg %rax, %rax</code>				
	<code>jne L1</code>	# Not taken			
	<code>irmovq \$1, %rax</code>	# Fall Through			
L1	<code>irmovq \$4, %rcx</code>	# Target			
	<code>irmovq \$3, %rax</code>	# Target + 1			

Control Dependency

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed.
- 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



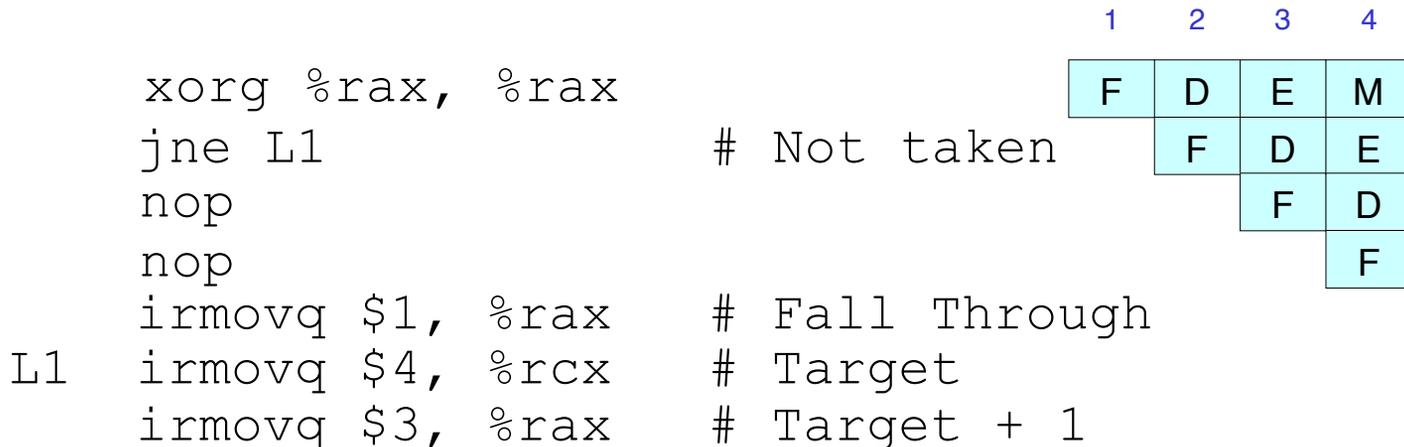
Control Dependency

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed.
- 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
	<code>xorg %rax, %rax</code>	F	D	E	M
	<code>jne L1</code> # Not taken		F	D	E
	<code>nop</code>			F	D
	<code>irmovq \$1, %rax</code> # Fall Through				
L1	<code>irmovq \$4, %rcx</code> # Target				
	<code>irmovq \$3, %rax</code> # Target + 1				

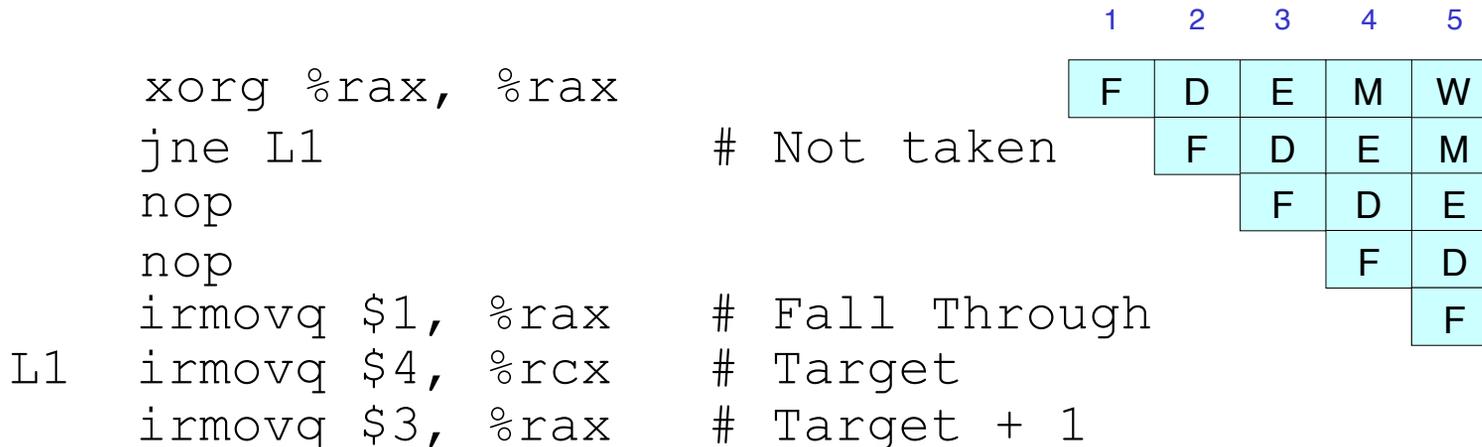
Control Dependency

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed.
- 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



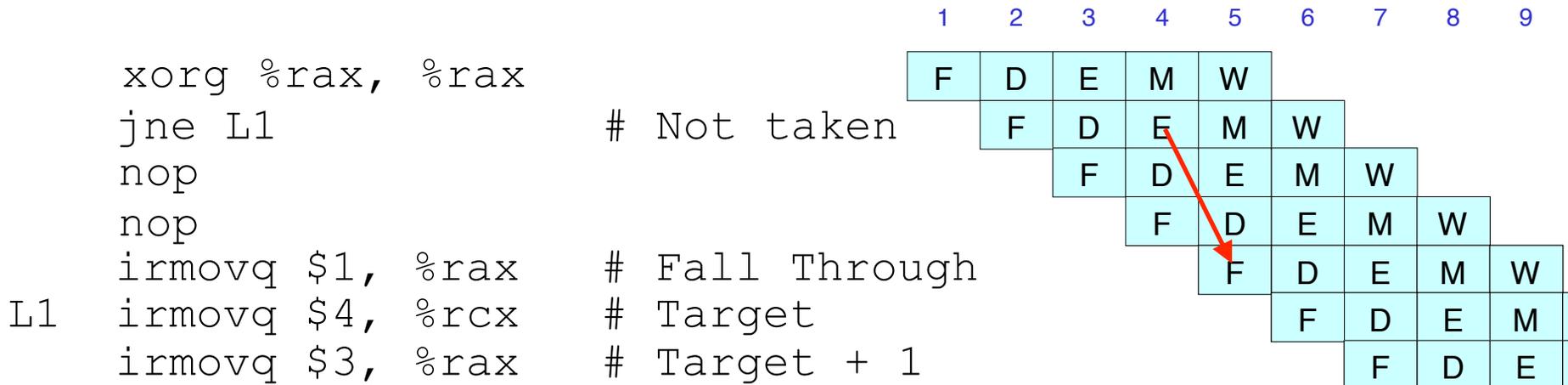
Control Dependency

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed.
- 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



Control Dependency

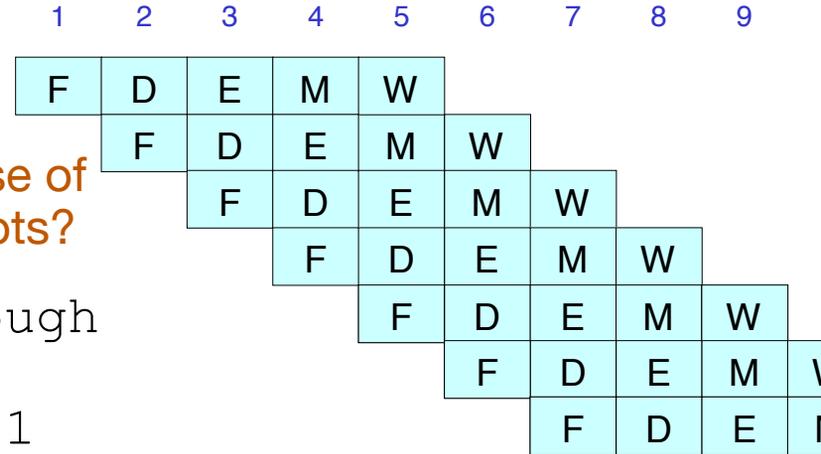
- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
 - `jne L1` determines whether `irmovq $1, %rax` should be executed
 - But `jne` doesn't know its outcome until after its Execute stage



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?

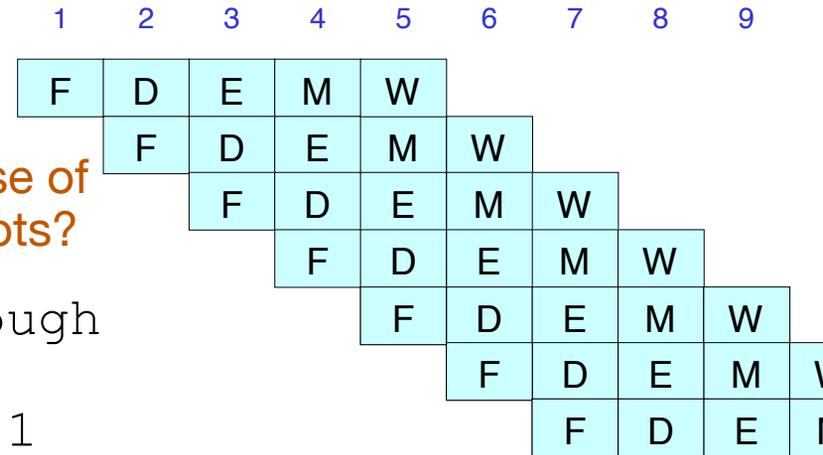


Delay Slots

```
xorg %rax, %rax
jne L1
nop
nop
L1: irmovq $1, %rax
    irmovq $4, %rcx
    irmovq $3, %rax
```

Can we make use of the 2 wasted slots?

```
# Fall Through
# Target
# Target + 1
```



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

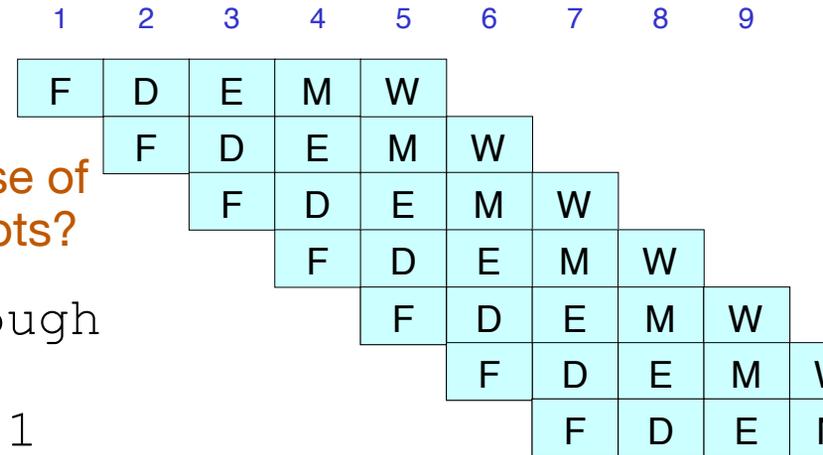
Delay Slots

```

    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?



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

```

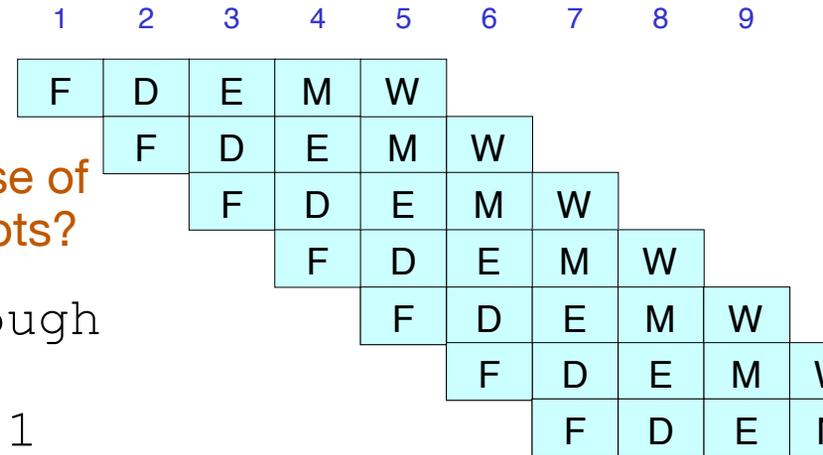
if (cond) {
    do_A();
} else {
    do_B();
}
do_C();

```

Delay Slots

```
xorg %rax, %rax
jne L1
nop
nop
L1: irmovq $1, %rax    # Fall Through
    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();
}
```

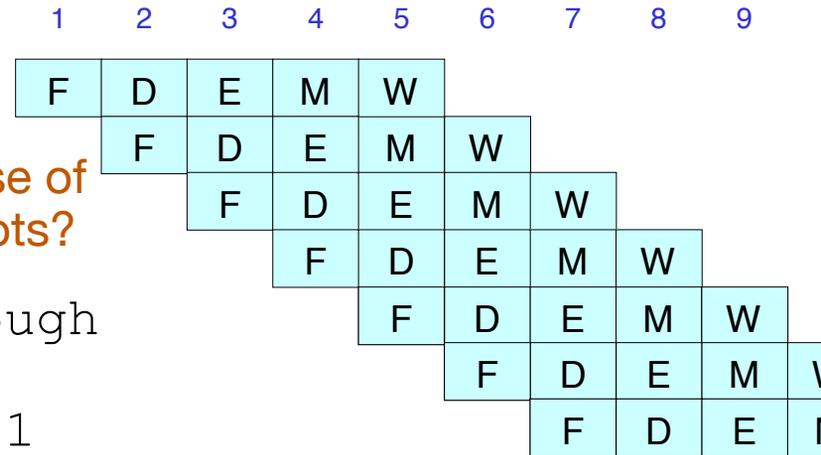
Delay Slots

```

    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

```

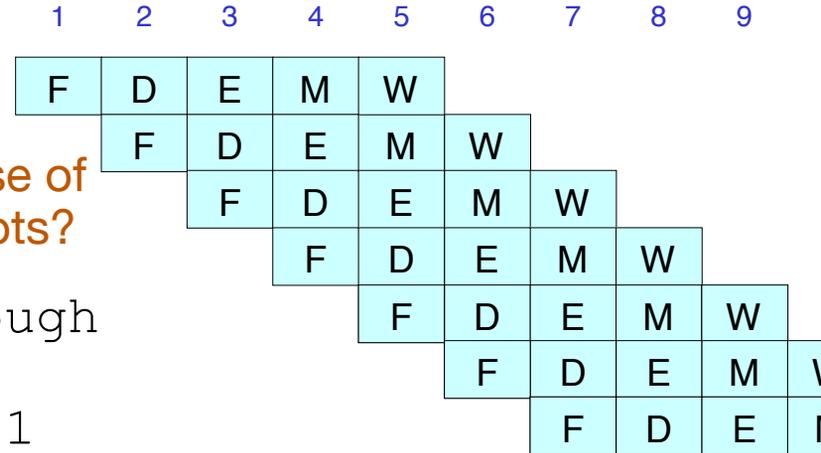
Delay Slots

```

    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

```

A green arrow points from the **or C, D** instruction in the first code block to the **or C, D** instruction in the second code block, indicating that the branch delay slot is filled with the branch target instruction.

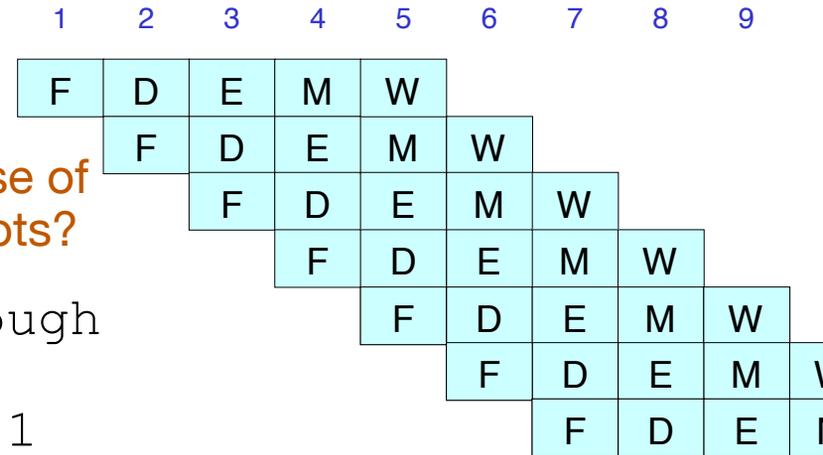
Delay Slots

```

    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