CSC 252: Computer Organization
Spring 2022: Lecture 13

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

- Programming assignment 3 is out
  - Details: https://www.cs.rochester.edu/courses/252/spring2022/labs/assignment3.html
  - Due on **March 3**, 11:59 PM
  - You (may still) have 3 slip days

13	14	15	16	17	18	19
20	21	22	23	24 **Today**	25	26
27	28	Mar 1	2	3 **Due** **Mid-term**	4	5

Announcements

- Mid-term exam: March 3; online. More details next lecture.
- Past exam & Problem set: https://www.cs.rochester.edu/courses/252/spring2022/handouts.html
- Anything on paper is allowed (book, notes, past exams, etc.)

13	14	15	16	17	18	19
20	21	22	23	24 **Today**	25	26
27	28	Mar 1	2	3 **Due** **Mid-term**	4	5

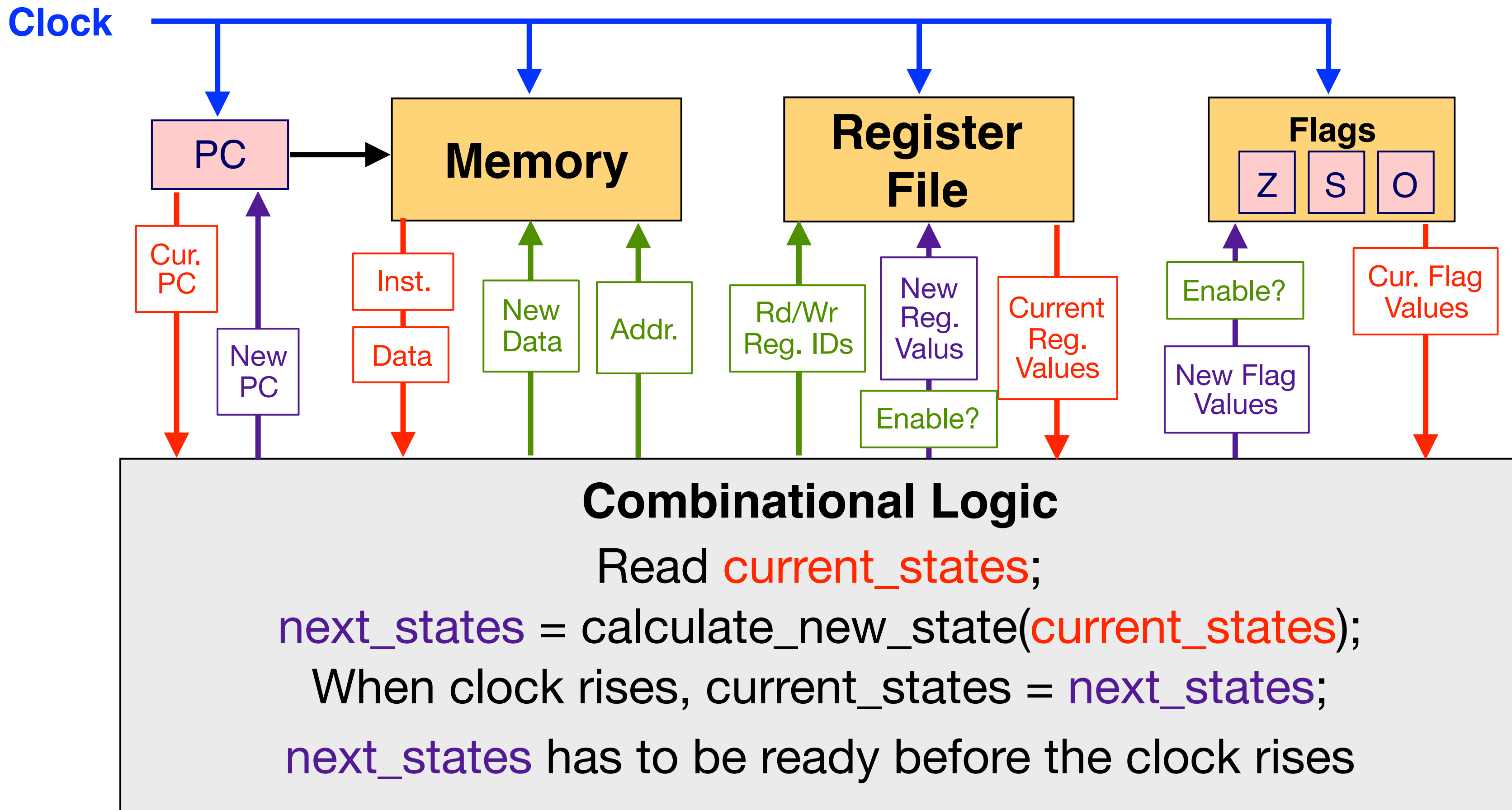
Announcements

- Office hours this week moved to Friday, 3-4pm.

- Grades for Lab 1 are posted.
- Will grade Lab 2 soon.

- 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



Key principles:

States are stored in storage units, e.g., Flip-flops (and SRAM and DRAM, later..)

New states are calculated by combination logic.

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**

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

/ 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

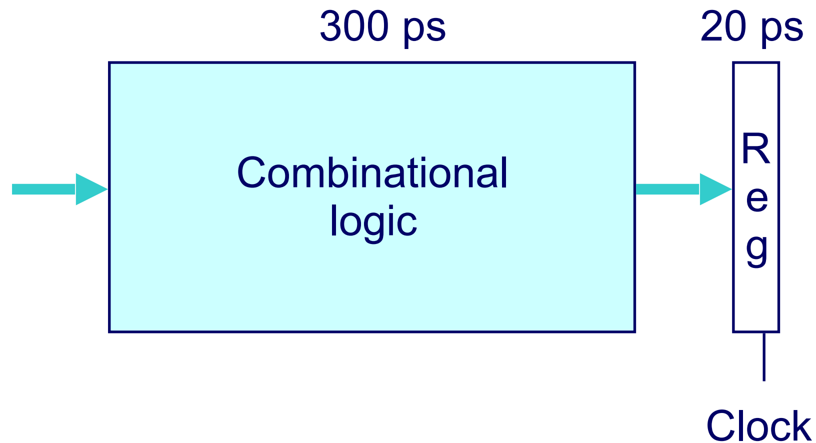
Limitations of a Single-Cycle CPU

- 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

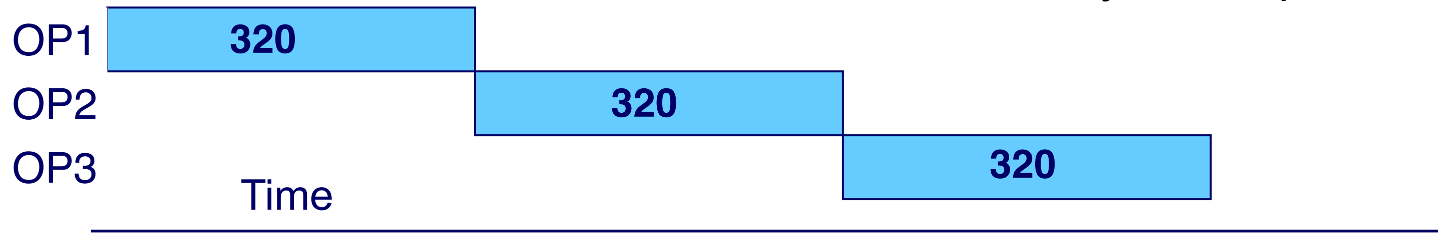
- Cycle time
  - Every instruction finishes in one cycle.
  - The absolute time takes to execute each instruction varies. Consider for instance an ADD instruction and a JMP instruction.
  - But the cycle time is uniform across instructions, so the cycle time needs to accommodate the worst case, i.e.,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



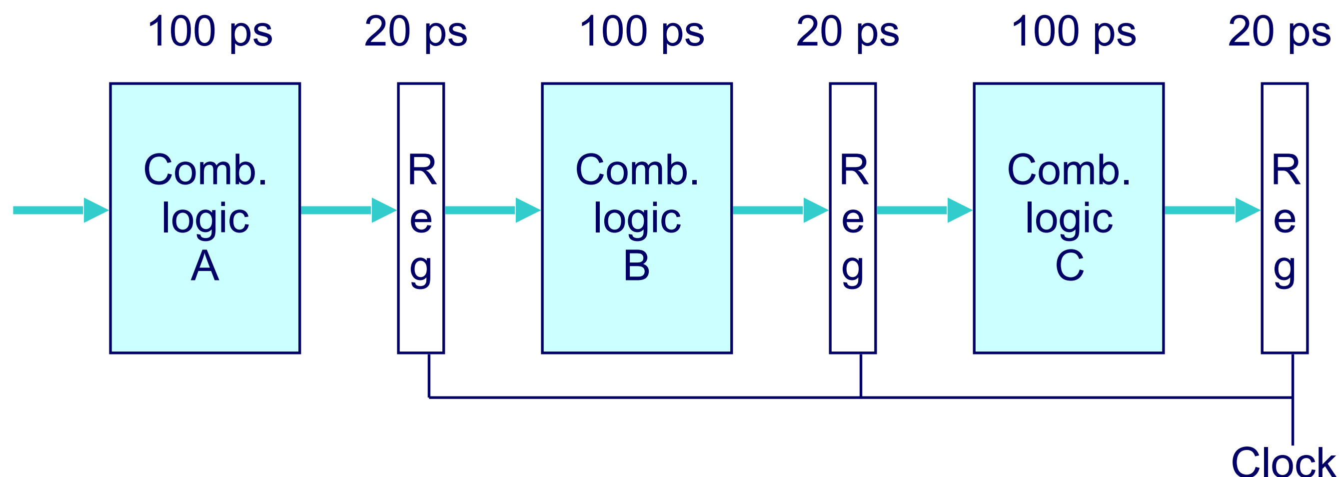
- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle time of at least 320 ps

Pipeline Diagrams



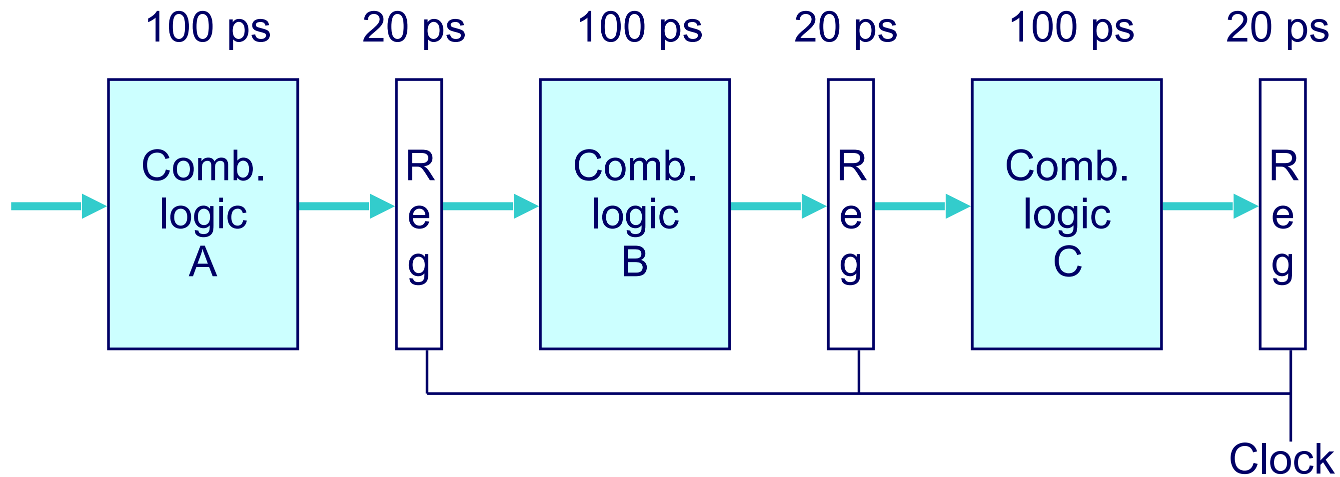
- 3 instructions will take 960 ps to finish
  - First cycle: Inst 1 takes 300 ps to compute new state, 20 ps to store the new states
  - Second cycle: Inst 2 starts; it takes 300 ps to compute new states, 20 ps to store new states
  - And so on…

3-Stage Pipelined Version

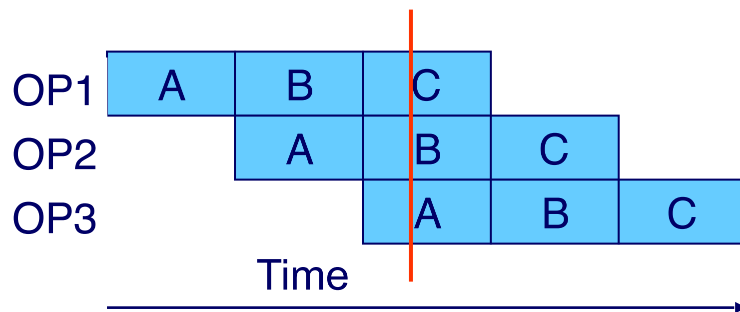


- Divide combinational logic into 3 stages of 100 ps each
- Insert registers between stages to store intermediate data between stages. These are call pipeline registers (ISA-invisible)
- Can begin a new instruction as soon as the previous one finishes stage A and has stored the intermediate data.
  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

3-Stage Pipelined Version



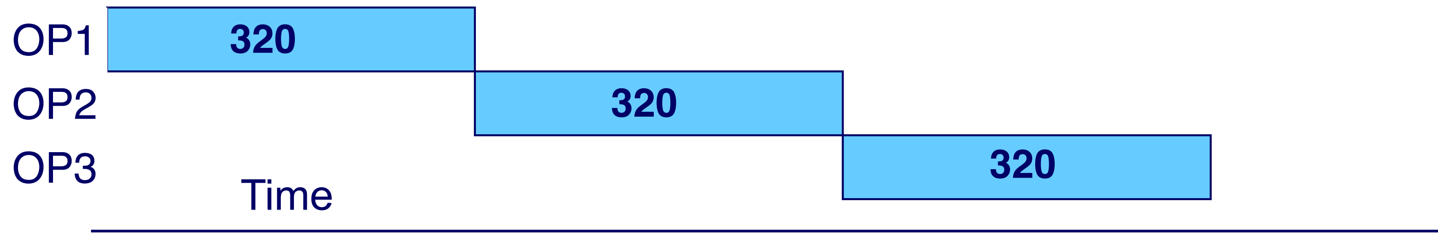
3-Stage Pipelined



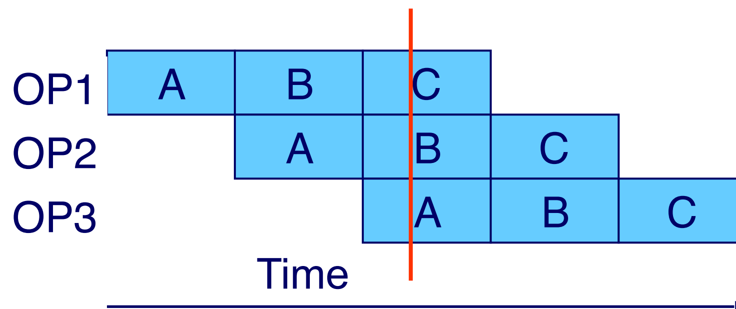
Comparison

Unpipelined

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



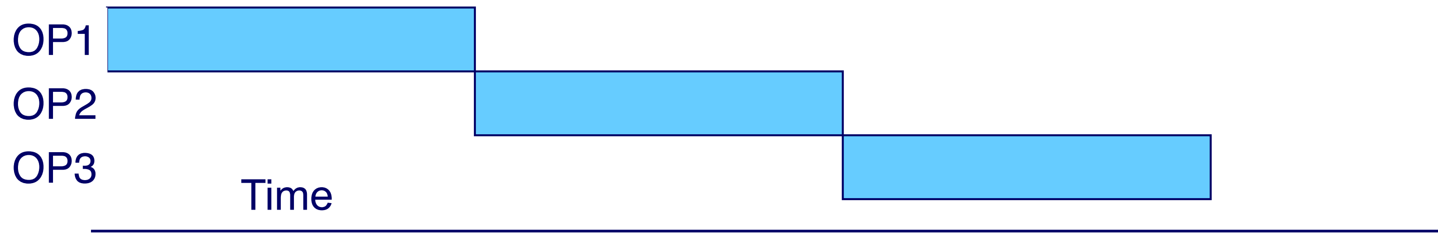
3-Stage Pipelined



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

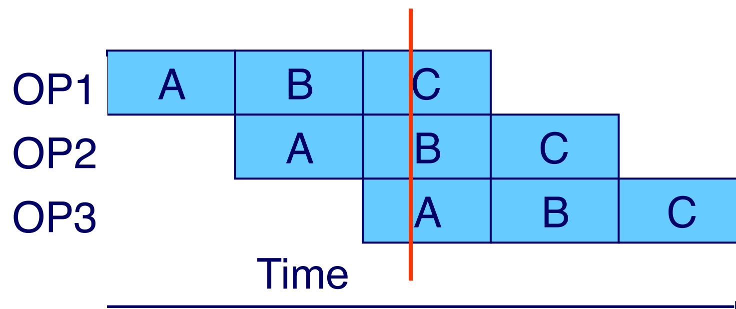
Benefits of Pipelining

- 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

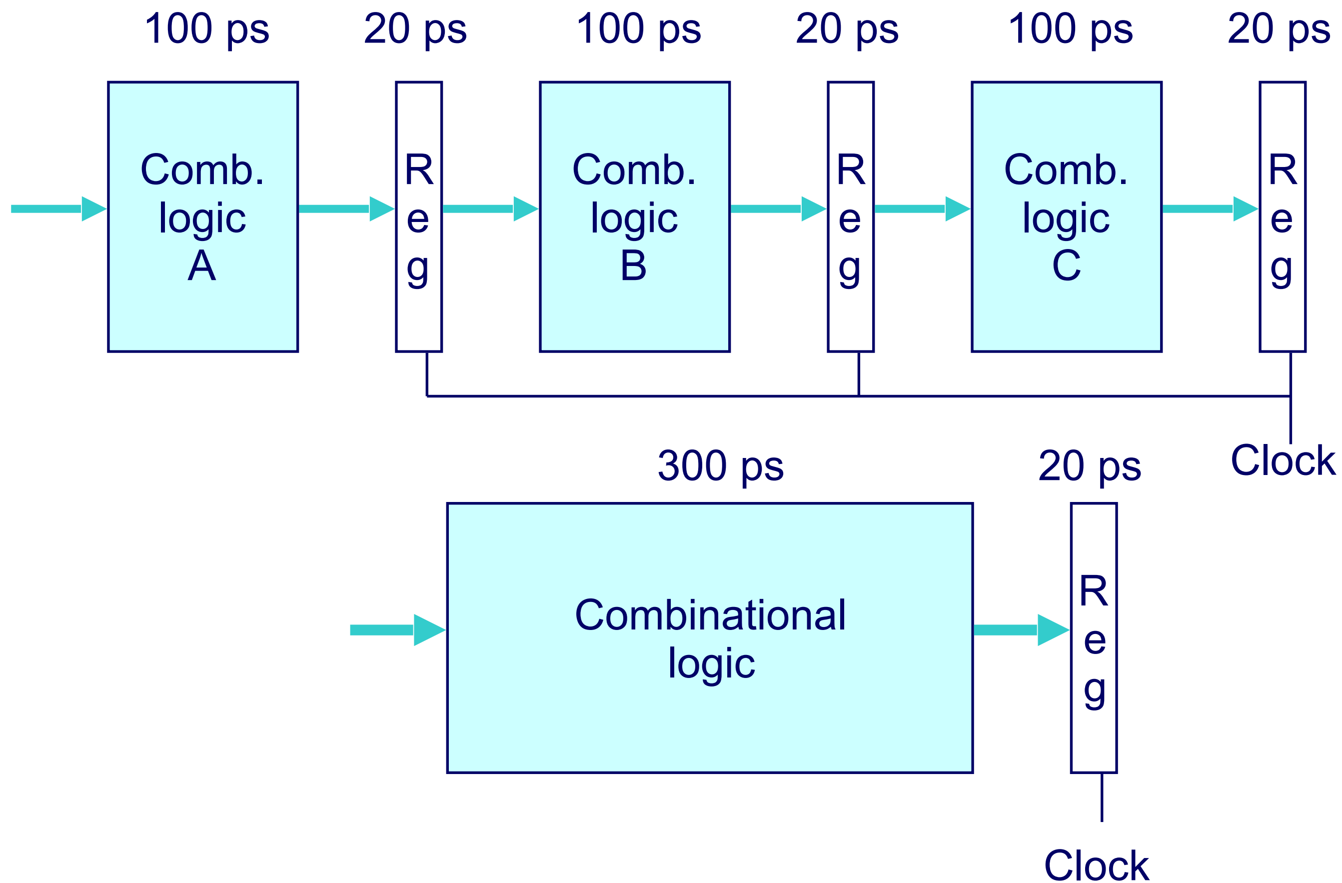
2. CPI reduces from 1 to 1/3 (i.e., executing 3 instruction in one cycle)



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

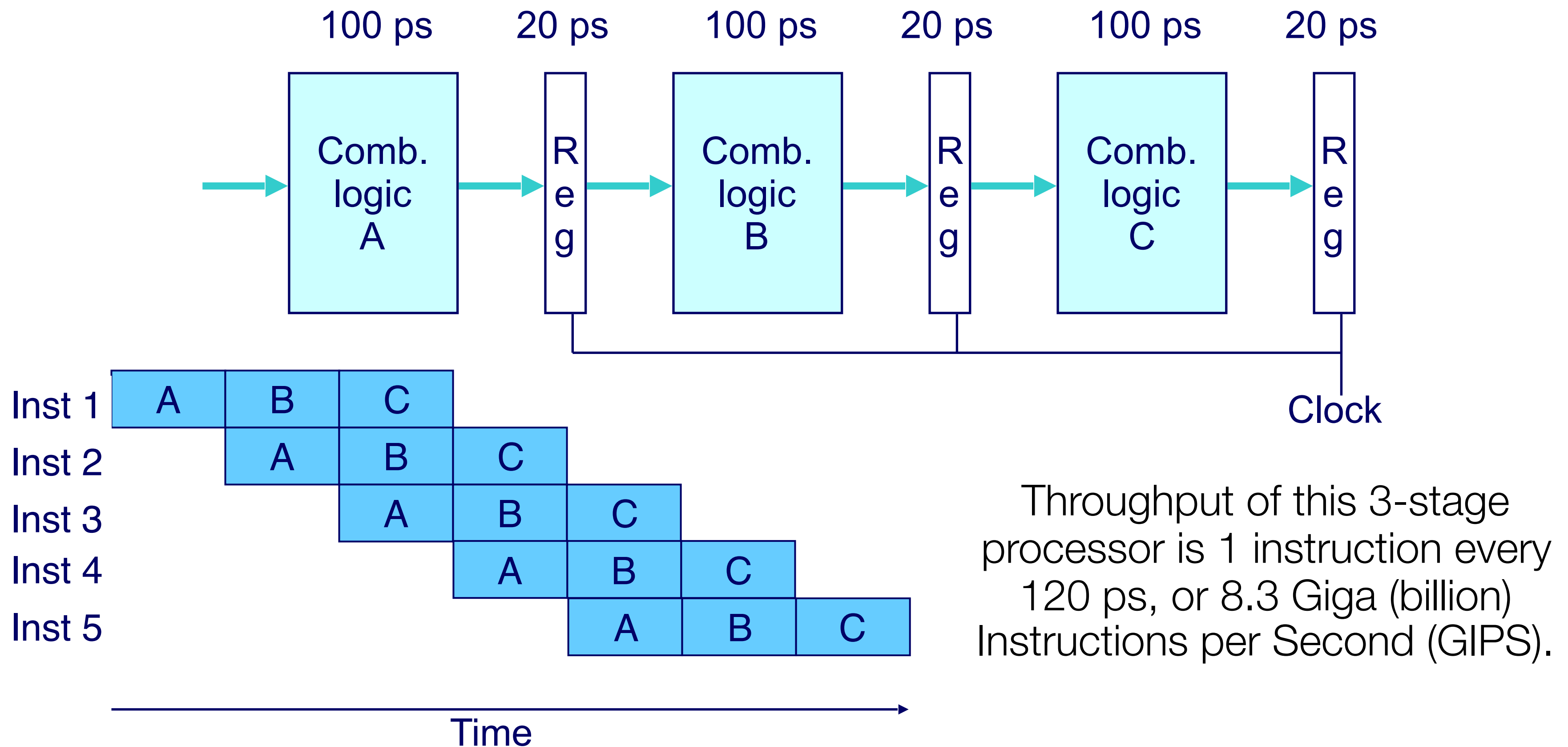
Pipeline Trade-offs

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



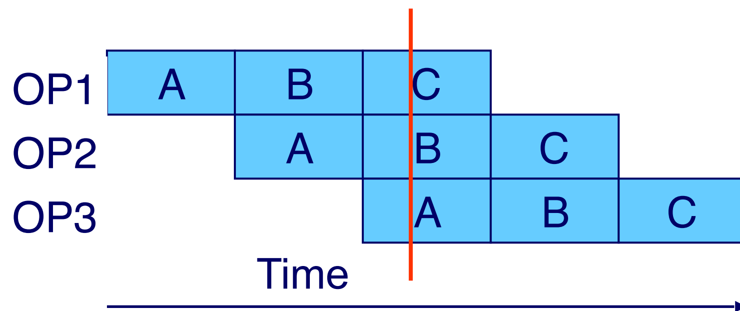
Throughput

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



One Requirement of Pipelining

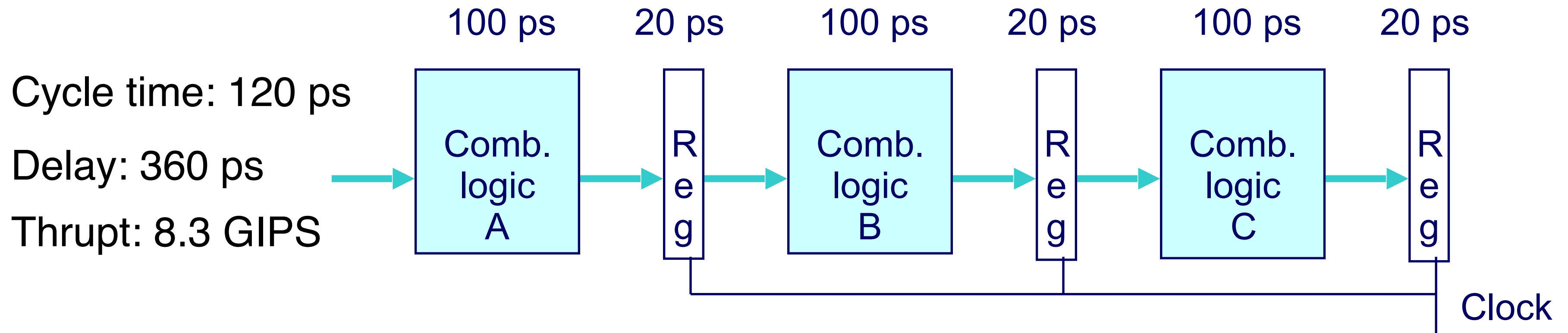
- The stages need to be using different hardware structures.
- That is, Stage A, Stage B, and Stage C need to exercise different parts of the combination logic.



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

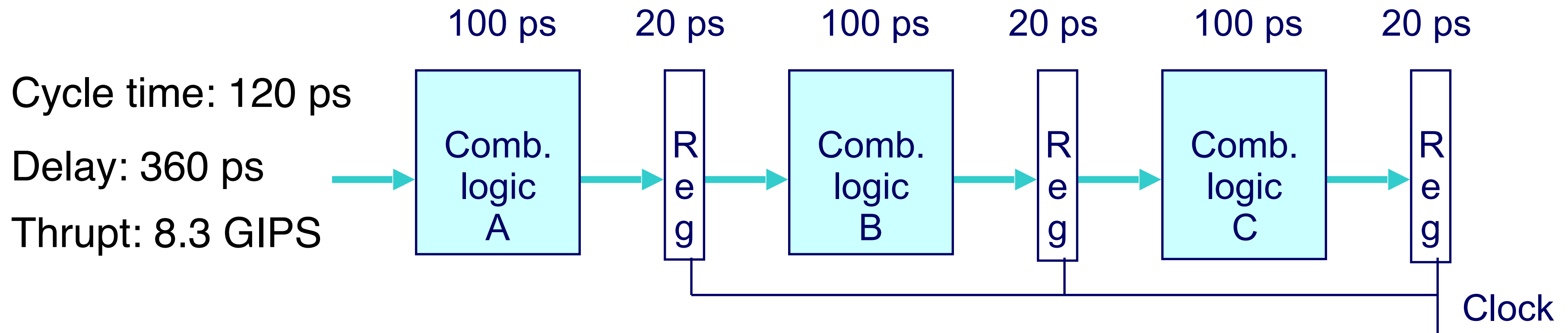
Aside: Unbalanced Pipeline

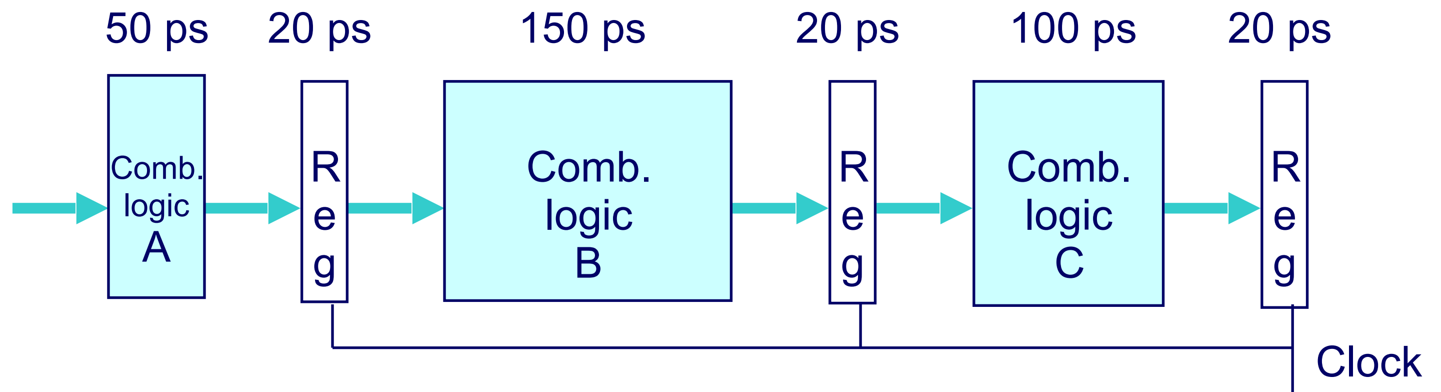
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput



Aside: Unbalanced Pipeline

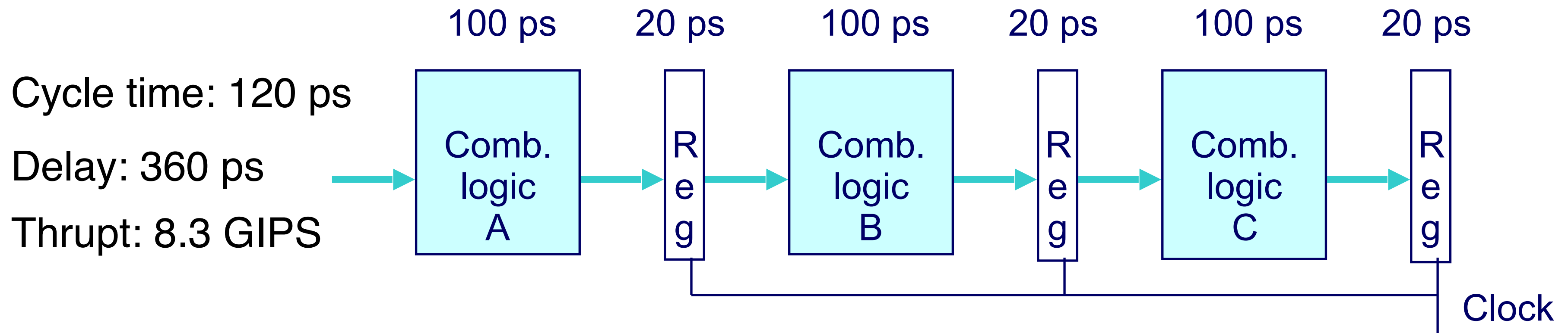
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

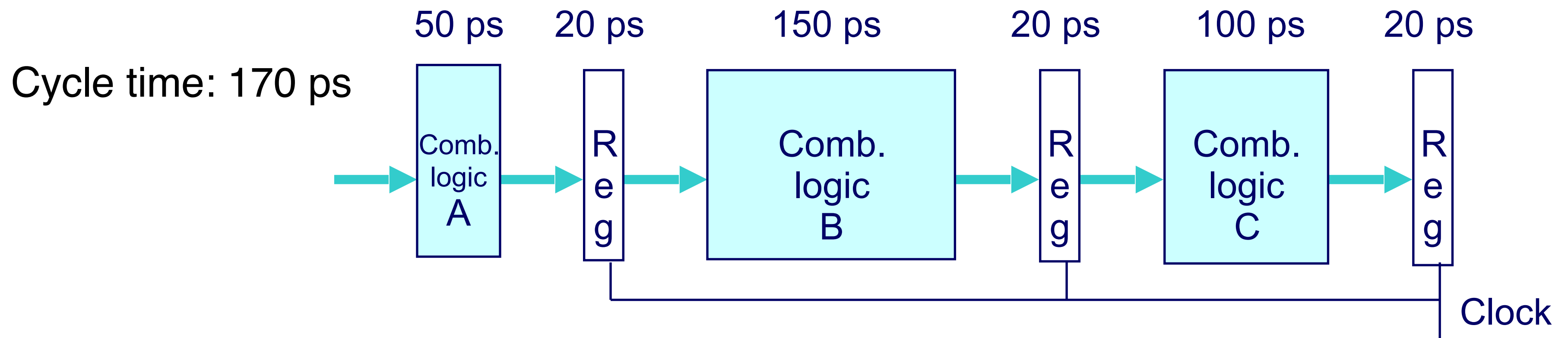




Aside: Unbalanced Pipeline

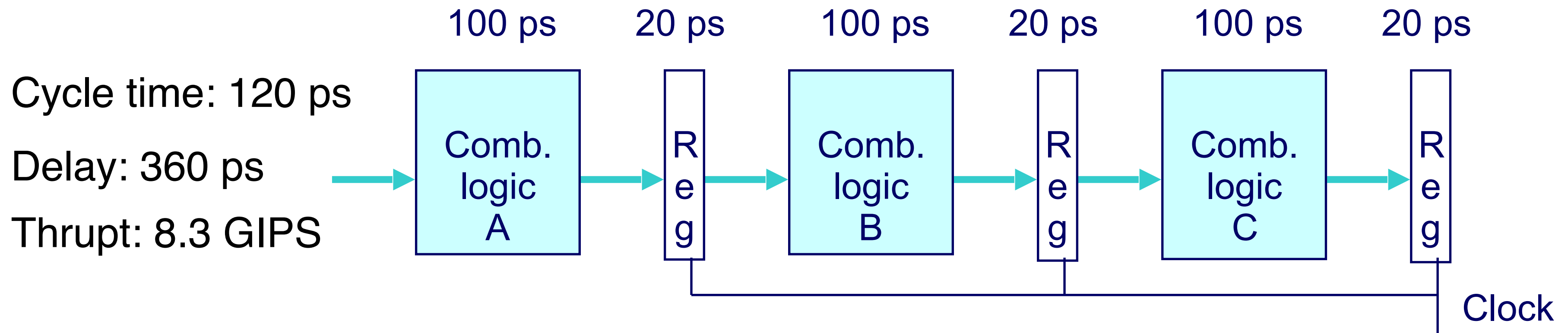
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

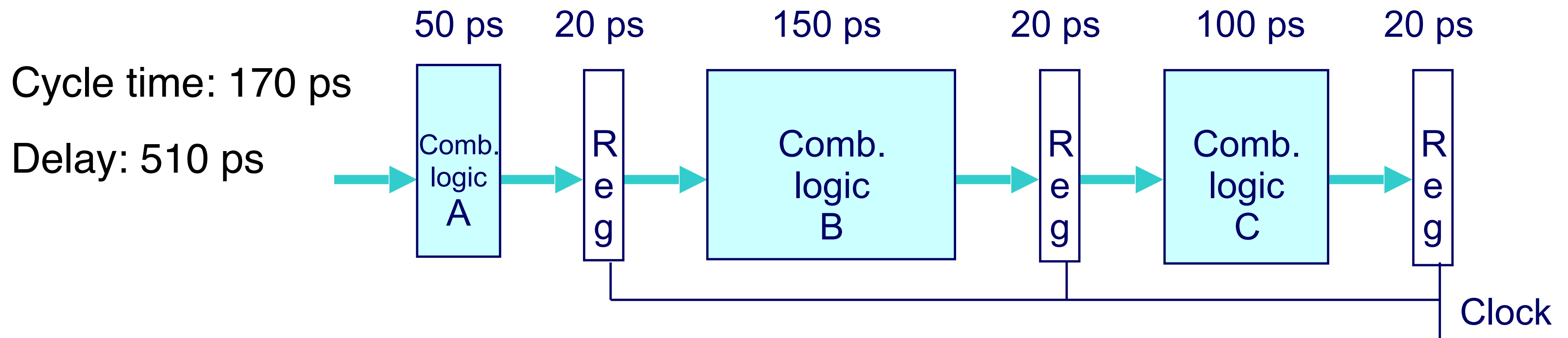




Aside: Unbalanced Pipeline

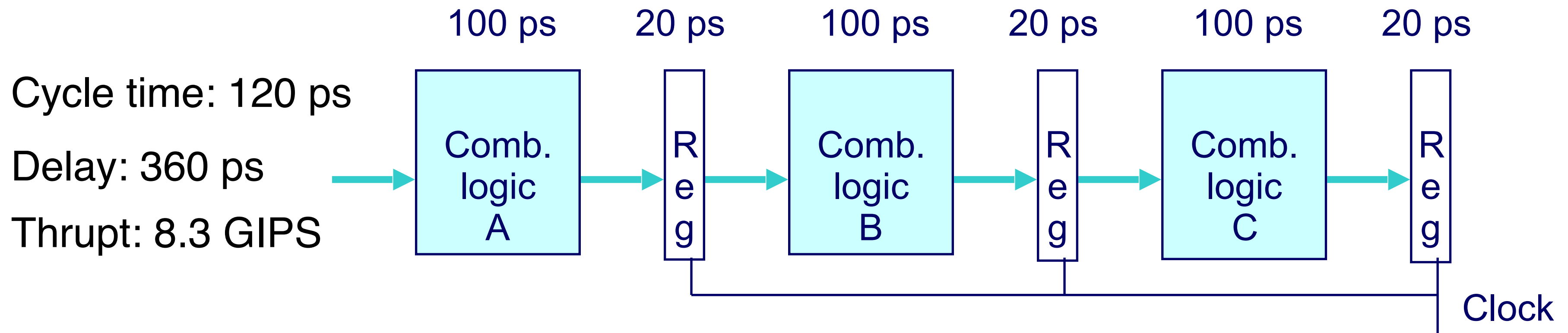
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

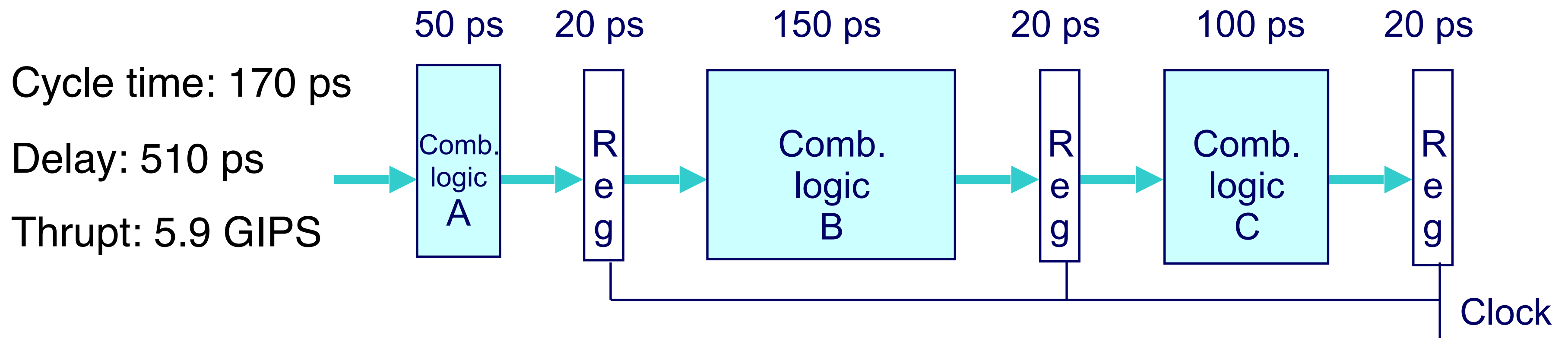




Aside: Unbalanced Pipeline

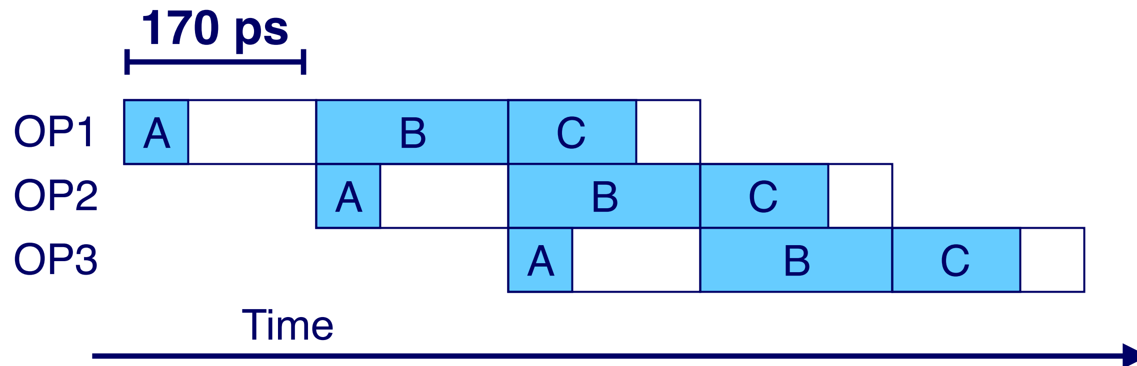
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

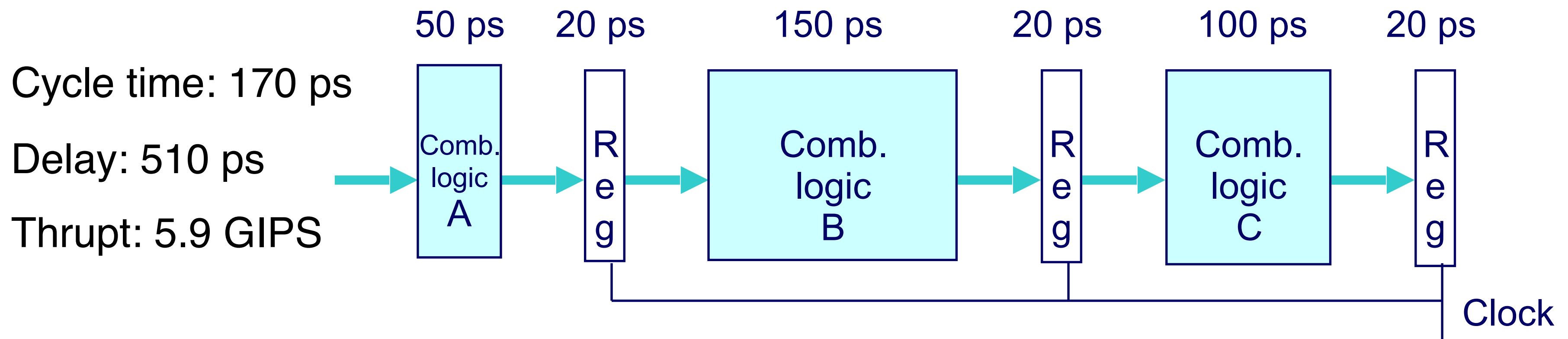




Aside: Unbalanced Pipeline

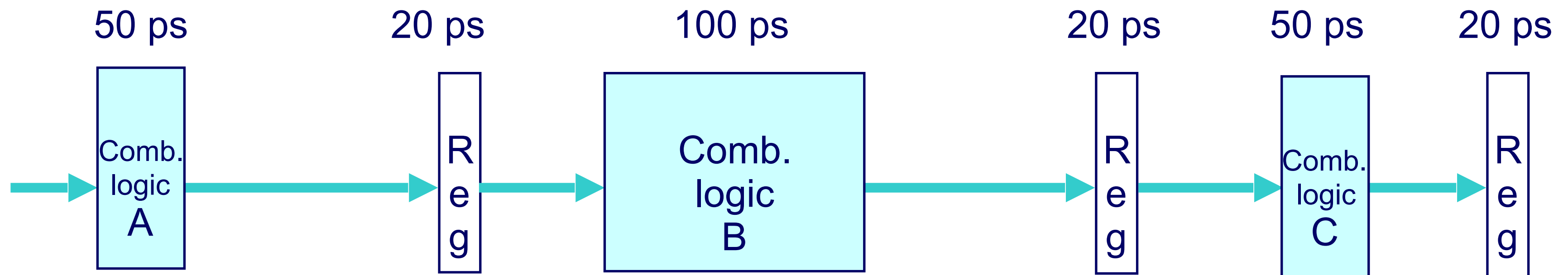
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput





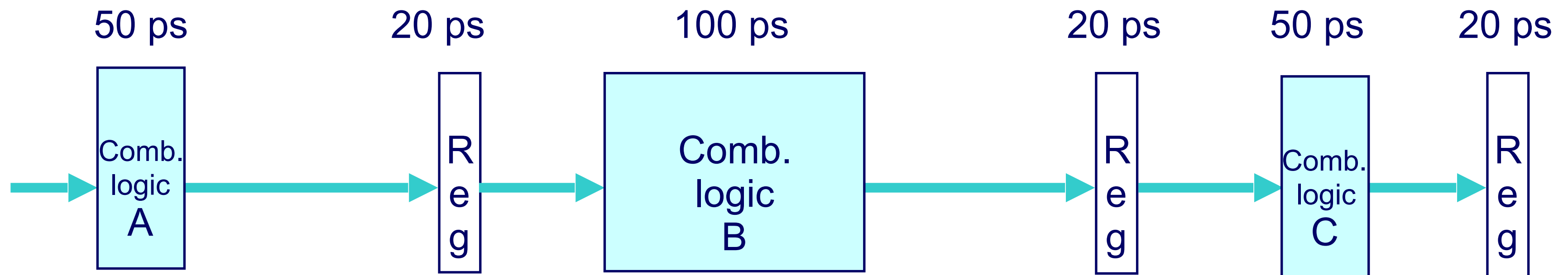
Aside: Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages



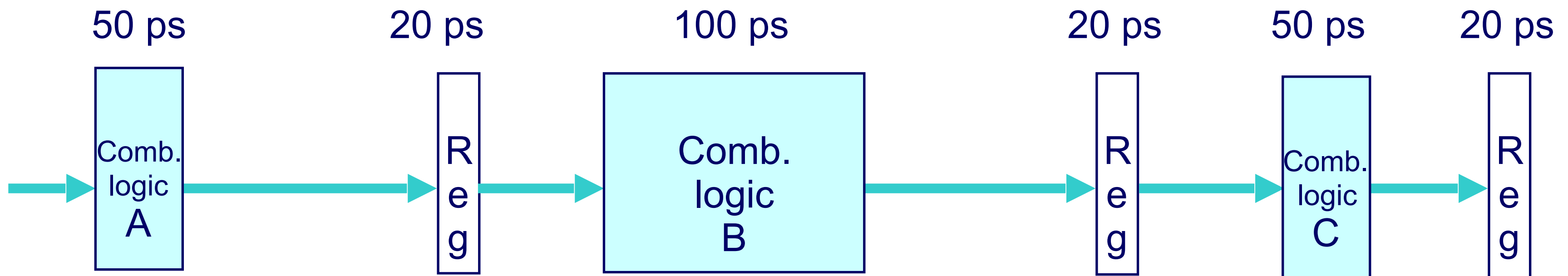
Aside: Mitigating Unbalanced Pipeline

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



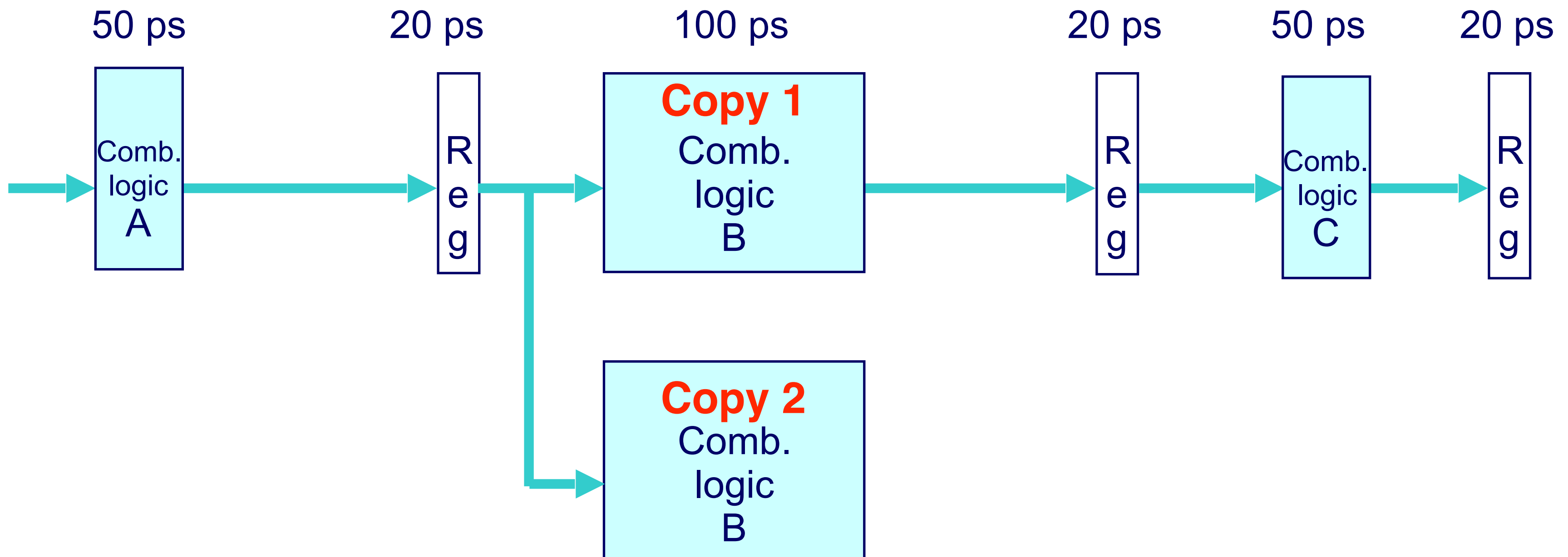
Aside: Mitigating Unbalanced Pipeline

- 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



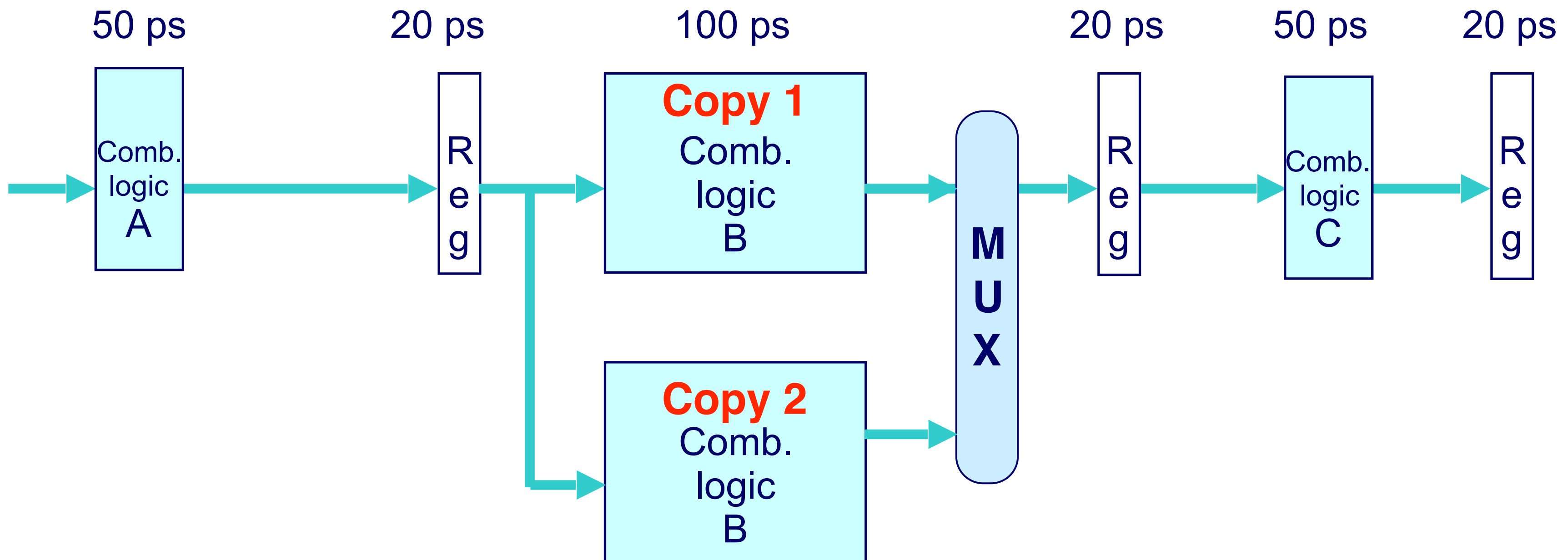
Aside: Mitigating Unbalanced Pipeline

- 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



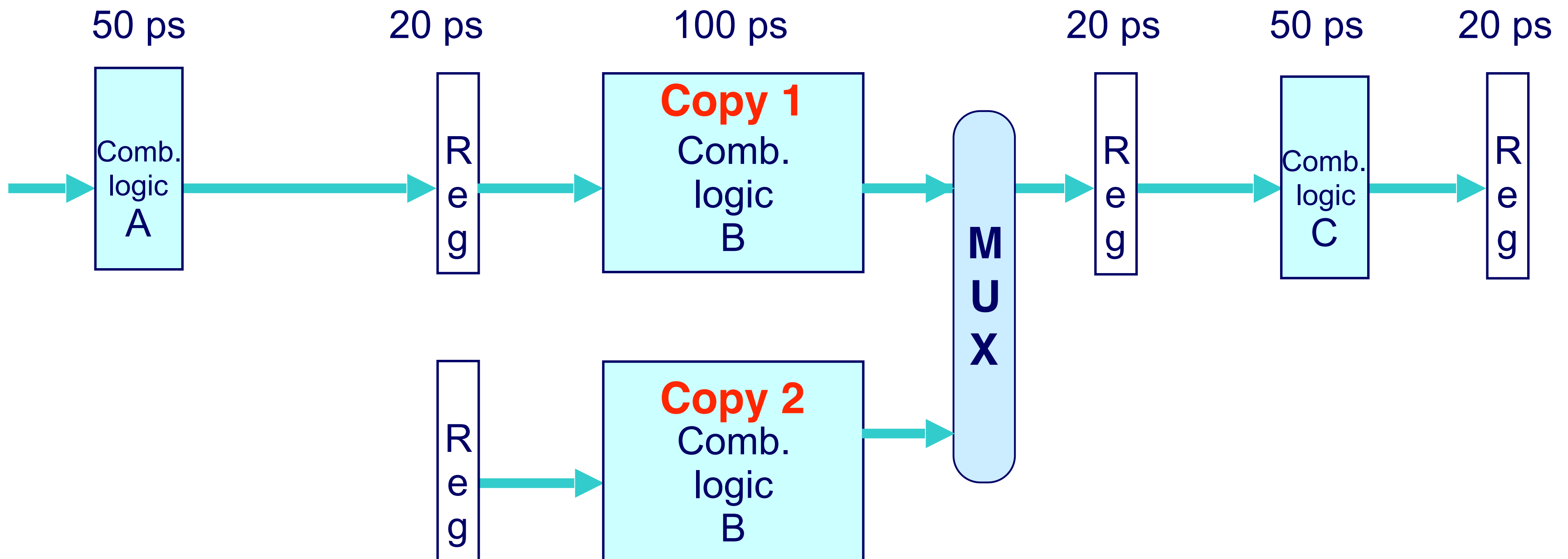
Aside: Mitigating Unbalanced Pipeline

- 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



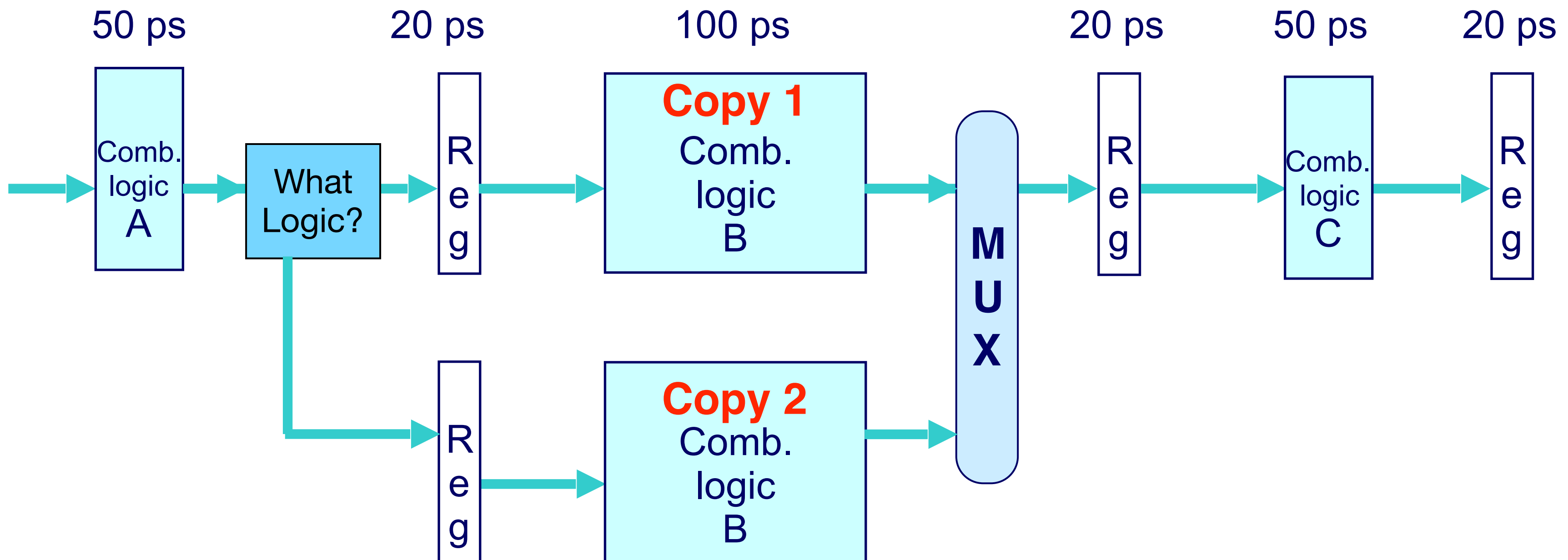
Aside: Mitigating Unbalanced Pipeline

- 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



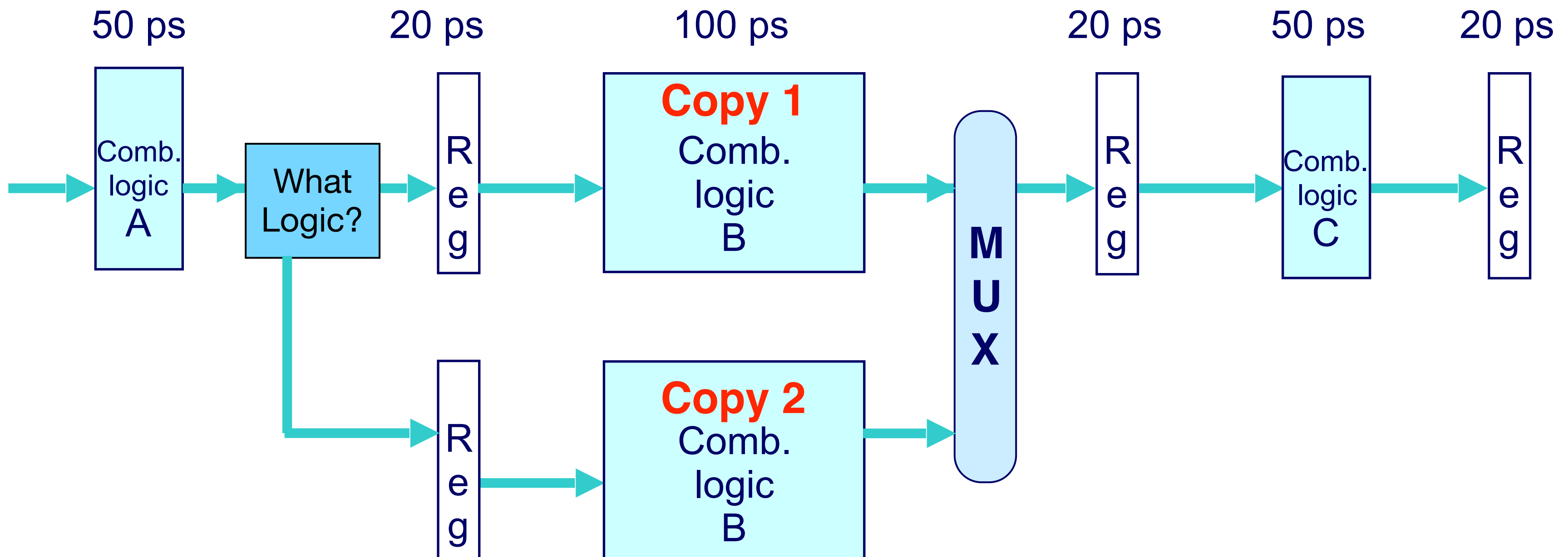
Aside: Mitigating Unbalanced Pipeline

- 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

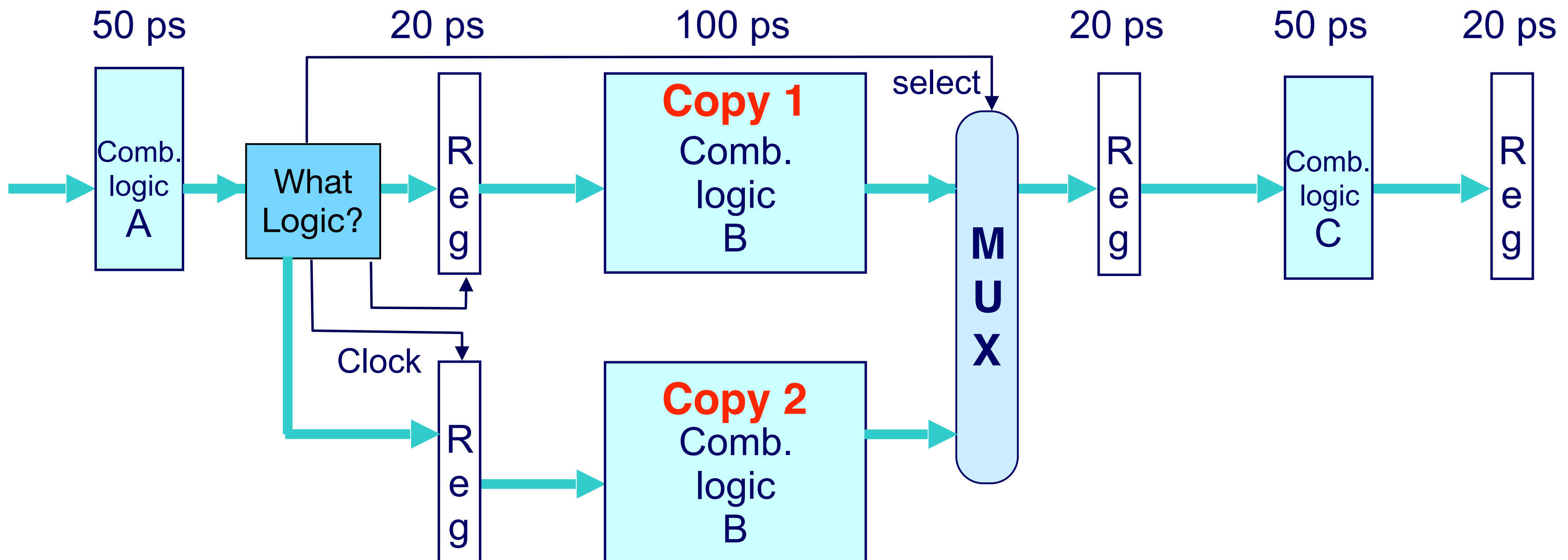


What logic do you need there?

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

Aside: Mitigating Unbalanced Pipeline

- 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

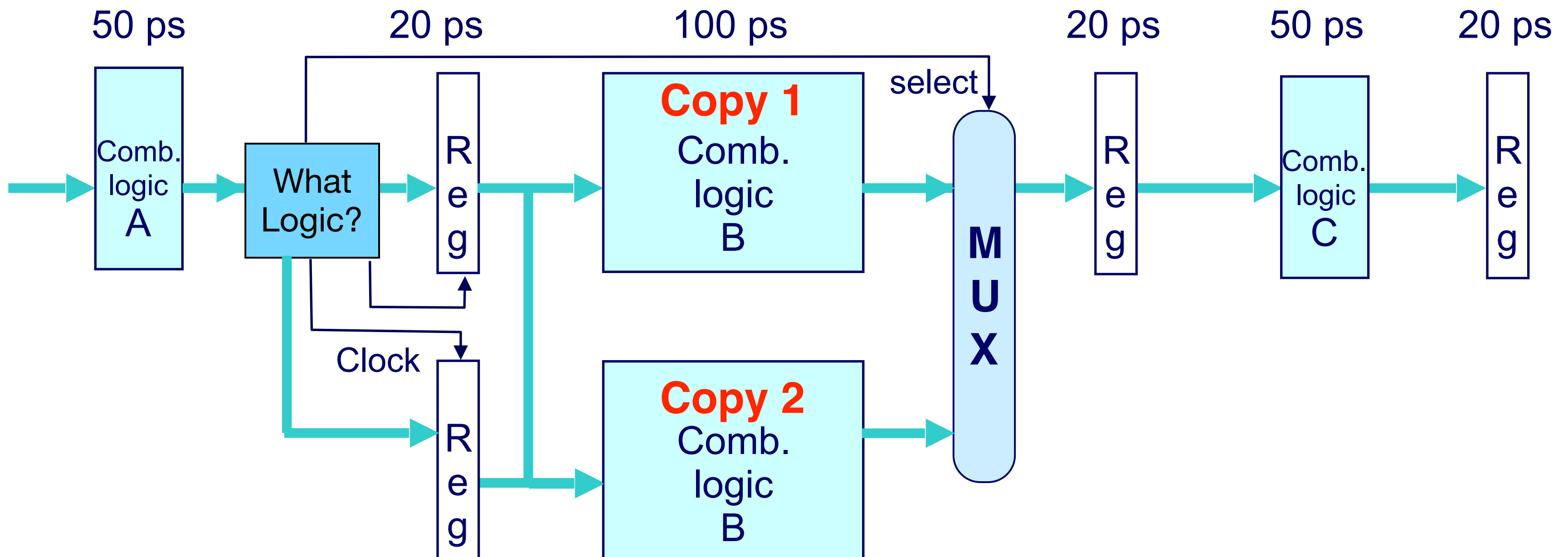


What logic do you need there?

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

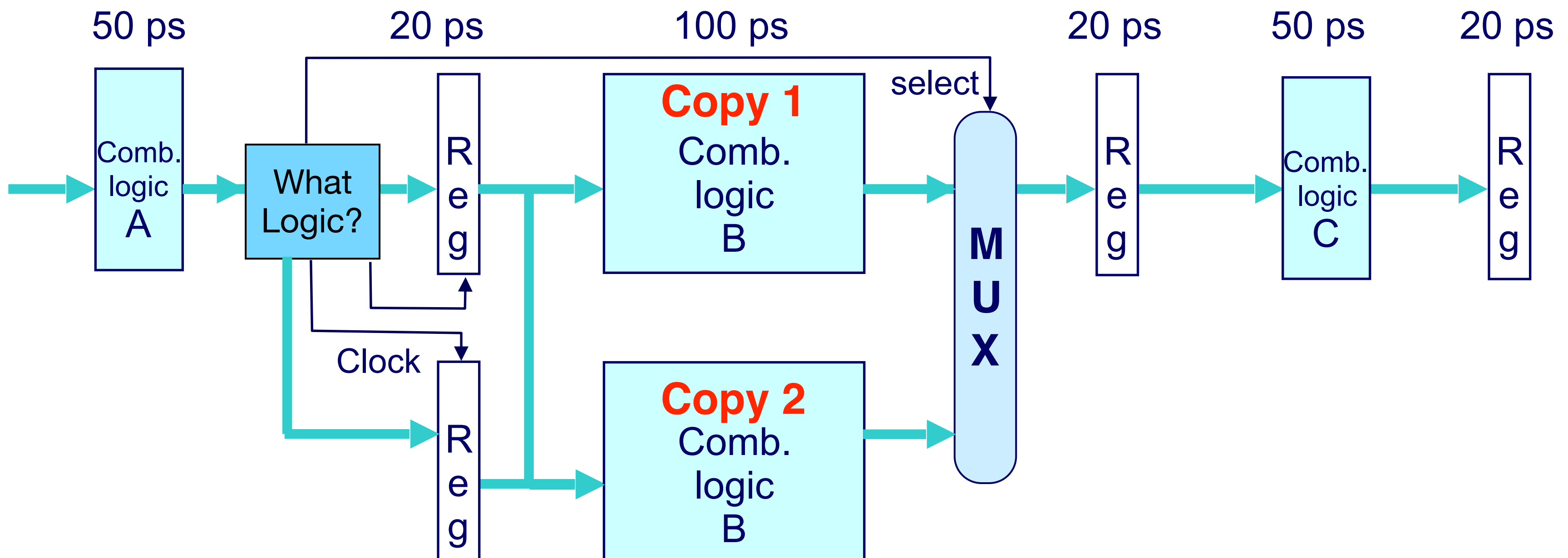
Aside: Mitigating Unbalanced Pipeline

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



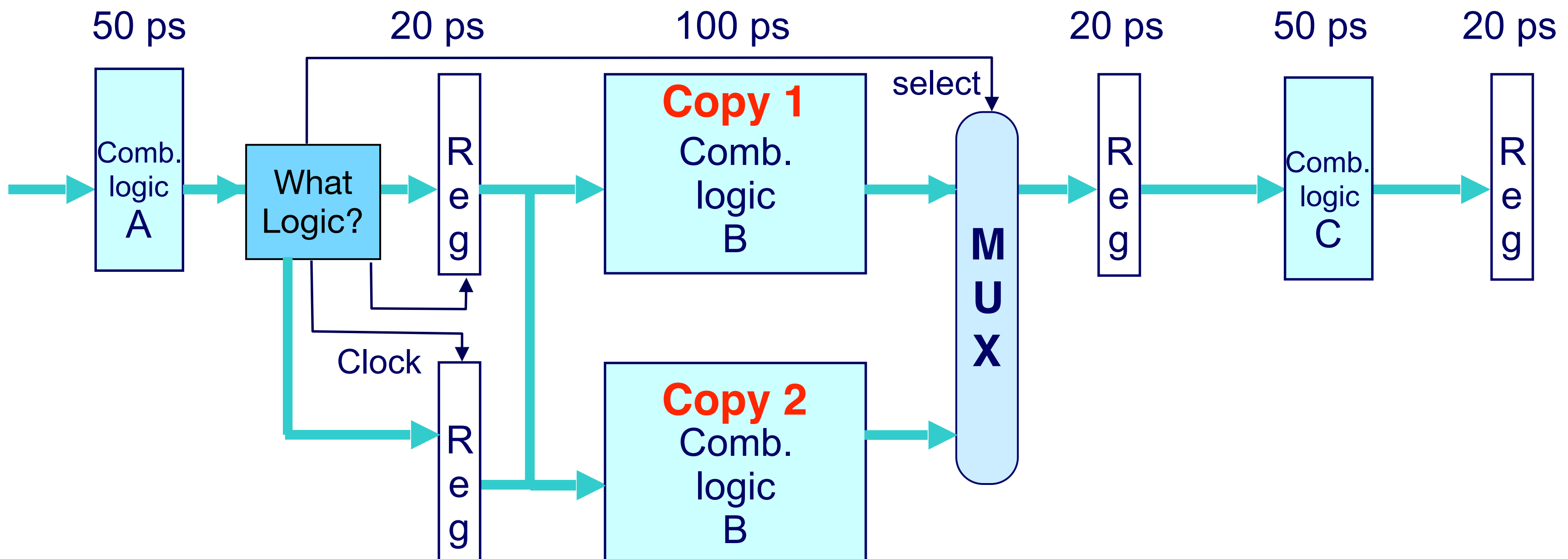
Aside: Mitigating Unbalanced Pipeline

- 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

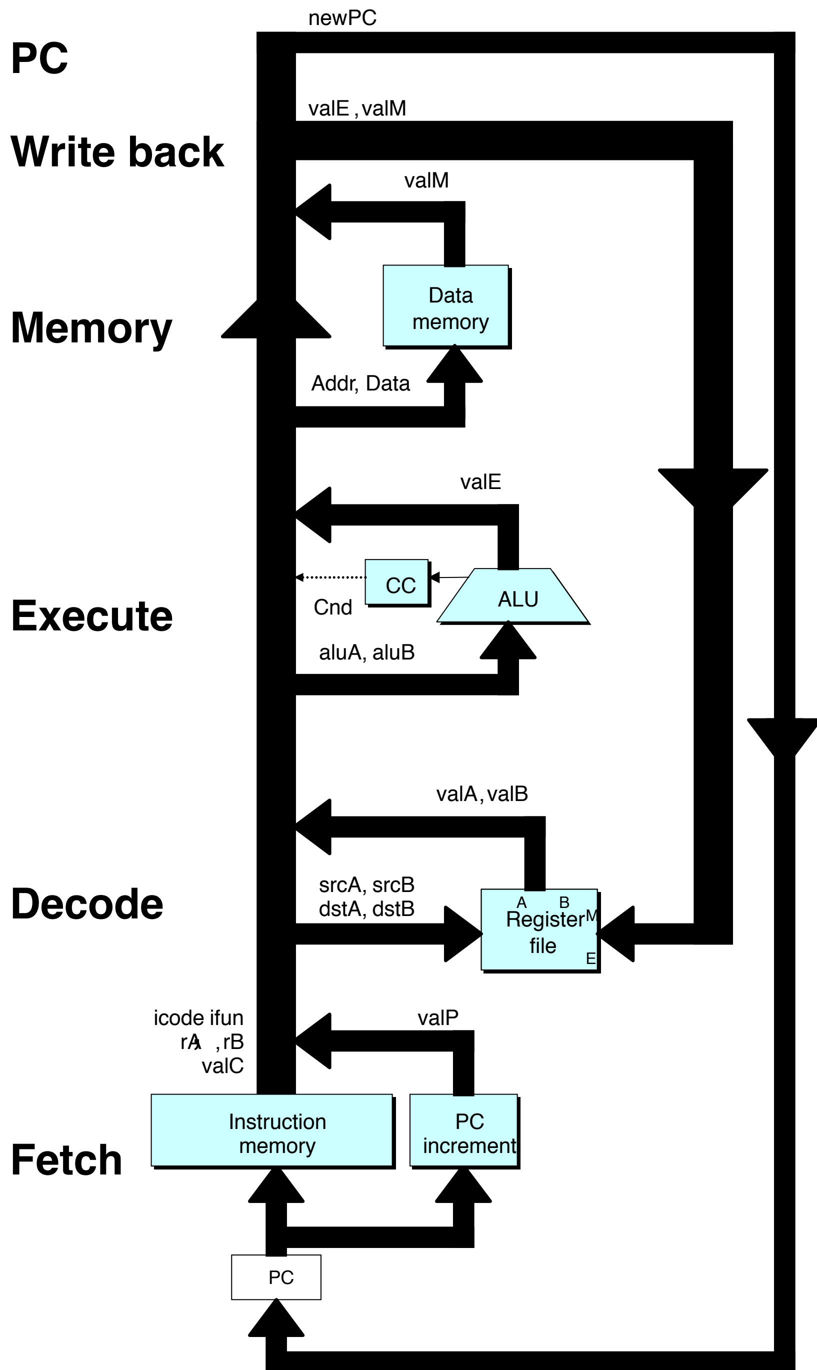
Decode: Read program registers

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

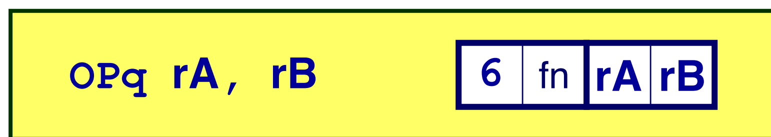
- Write program registers

PC

- Update program counter

Stage Computation: Arith/Log. Ops

Stage Computation: Arith/Log. Ops



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

Stage Computation: Arith/Log. Ops

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

Stage Computation: Arith/Log. Ops

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

Stage Computation: Arith/Log. Ops

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

Stage Computation: Arith/Log. Ops

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

Stage Computation: Arith/Log. Ops

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

Stage Computation: `rmmovq`

`rmmovq` rA , D (rB)	4	0	rA	rB	D

`rmmovq` rA, D(rB)

Stage Computation: `rmmovq`

	rmmovq rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_8[PC+2]	Read displacement D
	valP ← PC+10	Compute next PC

Stage Computation: `rmmovq`

	rmmovq rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_8[PC+2]	Read displacement D
	valP ← PC+10	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B

Stage Computation: `rmmovq`

	rmmovq rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_8[PC+2]	Read displacement D
	valP ← PC+10	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address

Stage Computation: `rmmovq`

	`rmmovq` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_8[PC+2]	Read displacement D
	valP ← PC+10	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_8[valE] ← valA	Write value to memory

Stage Computation: `rmmovq`

	rmmovq rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_8[PC+2]	Read displacement D
	valP ← PC+10	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_8[valE] ← valA	Write value to memory
Write back		

Stage Computation: `rmmovq`

	`rmmovq` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_8[PC+2]	Read displacement D
	valP ← PC+10	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_8[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	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	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_8[PC+1]	Read destination address
	valP ← PC+9	Fall through address

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

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_8[PC+1]	Read destination address
	valP ← 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	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_8[PC+1]	Read destination address
	valP ← PC+9	Fall through address
Decode		
Execute	Cnd ← Cond(CC,ifun)	Take branch?

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

Stage Computation: Jumps

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

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

Stage Computation: Jumps

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

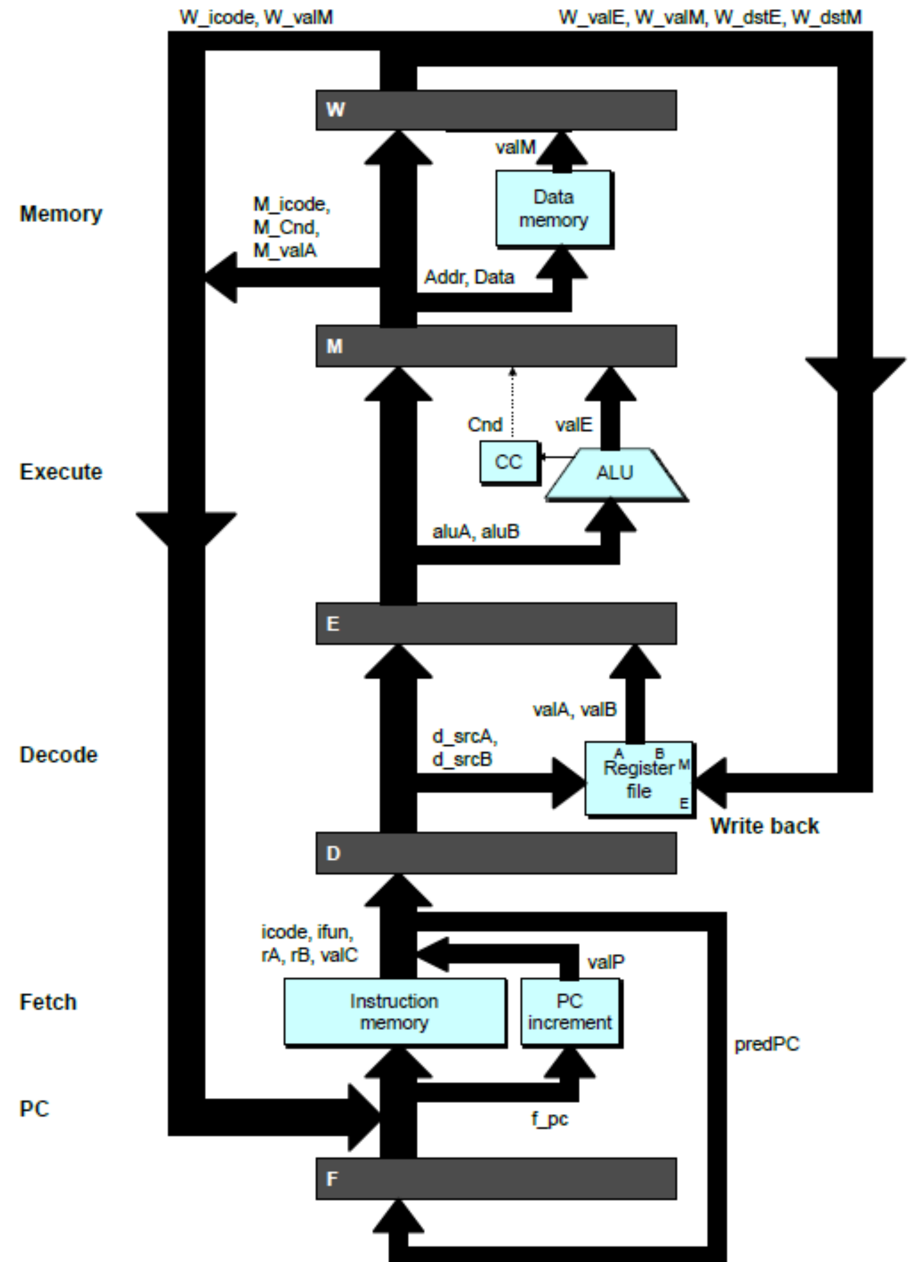
- Operate ALU

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

(AP PHOTO)

Pipelined

Real-World Pipelines: Car Washes

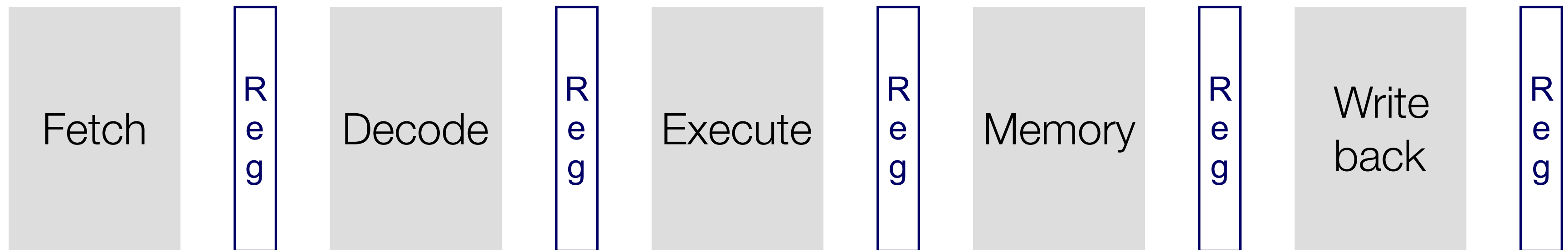
Sequential

Pipelined

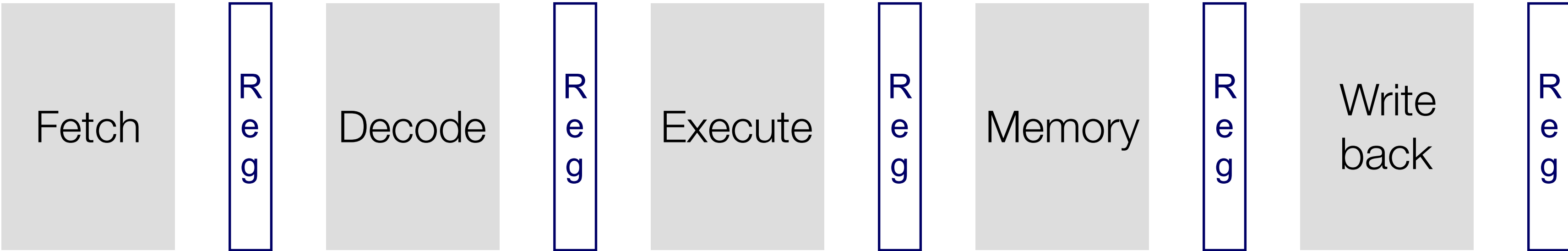
Idea

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

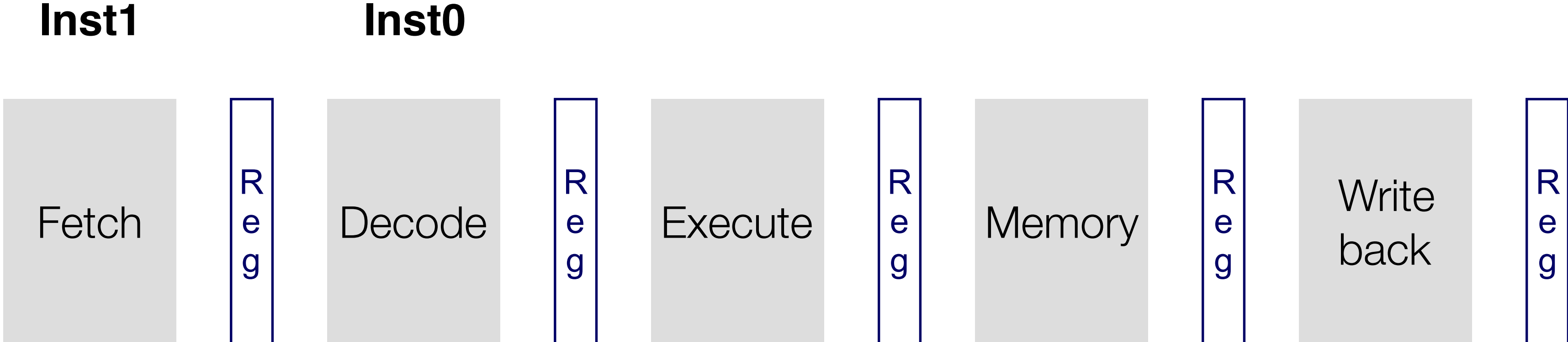
Pipeline Illustration



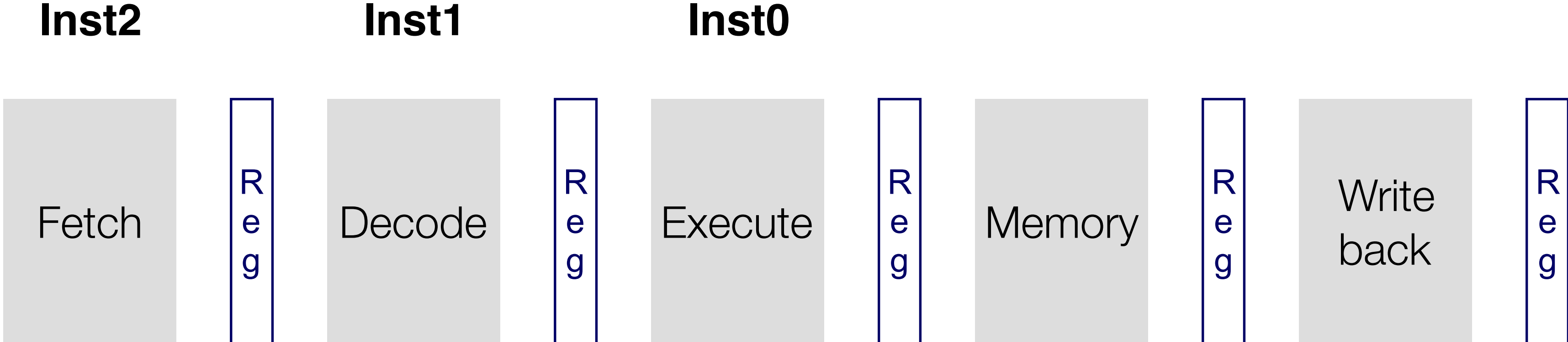
Pipeline Illustration



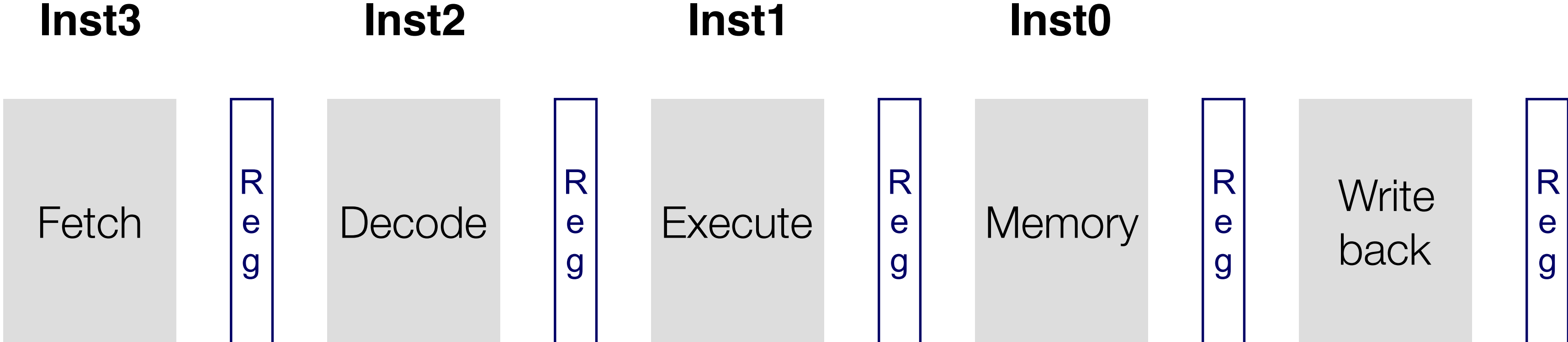
Pipeline Illustration



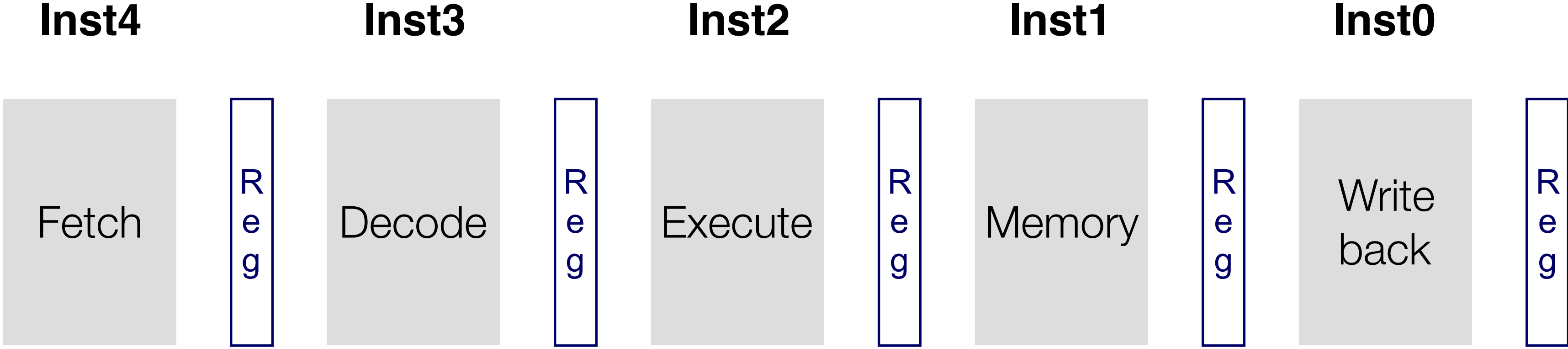
Pipeline Illustration



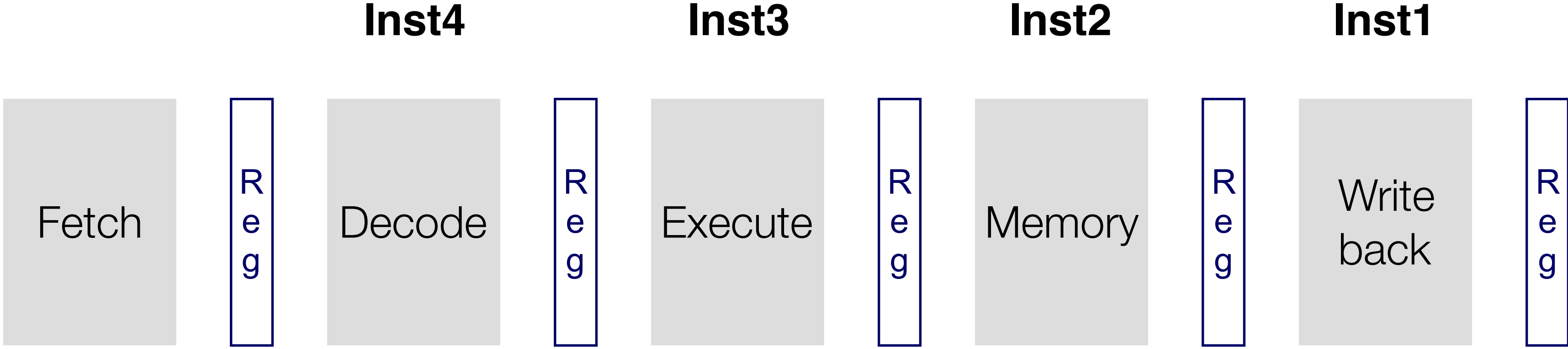
Pipeline Illustration



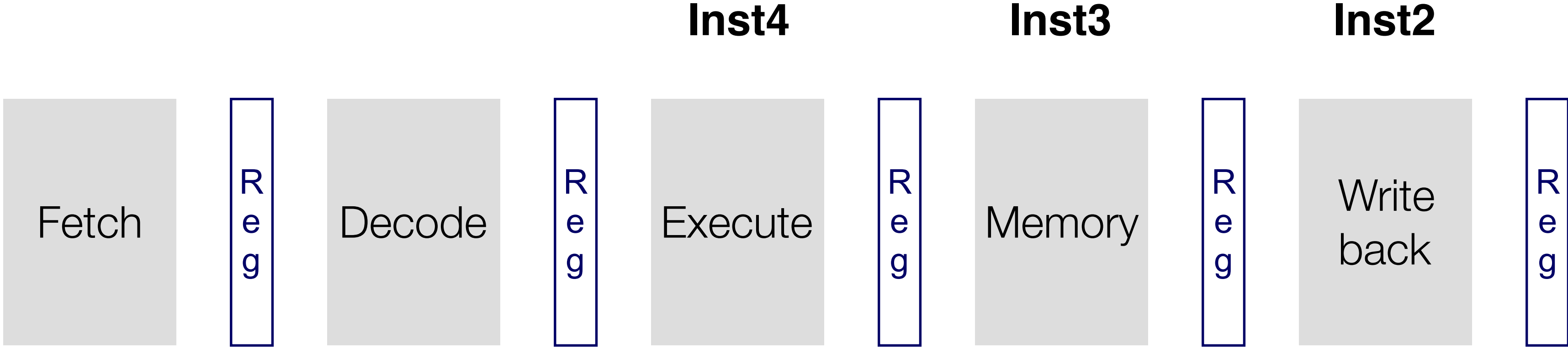
Pipeline Illustration



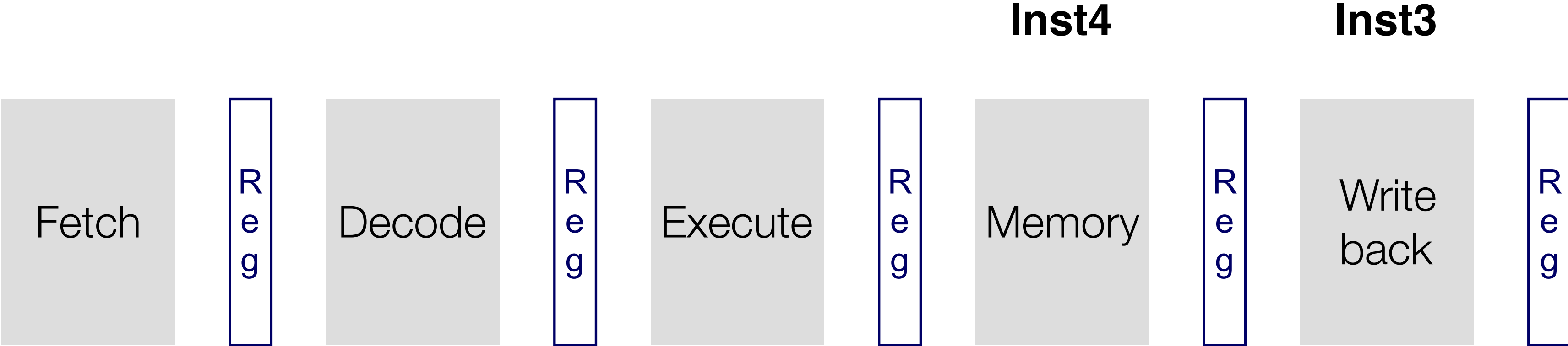
Pipeline Illustration



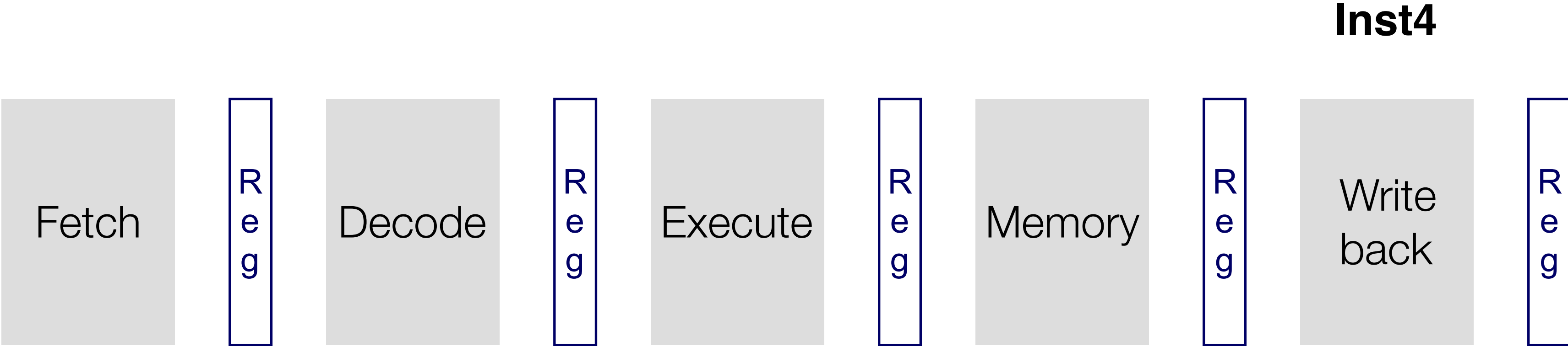
Pipeline Illustration



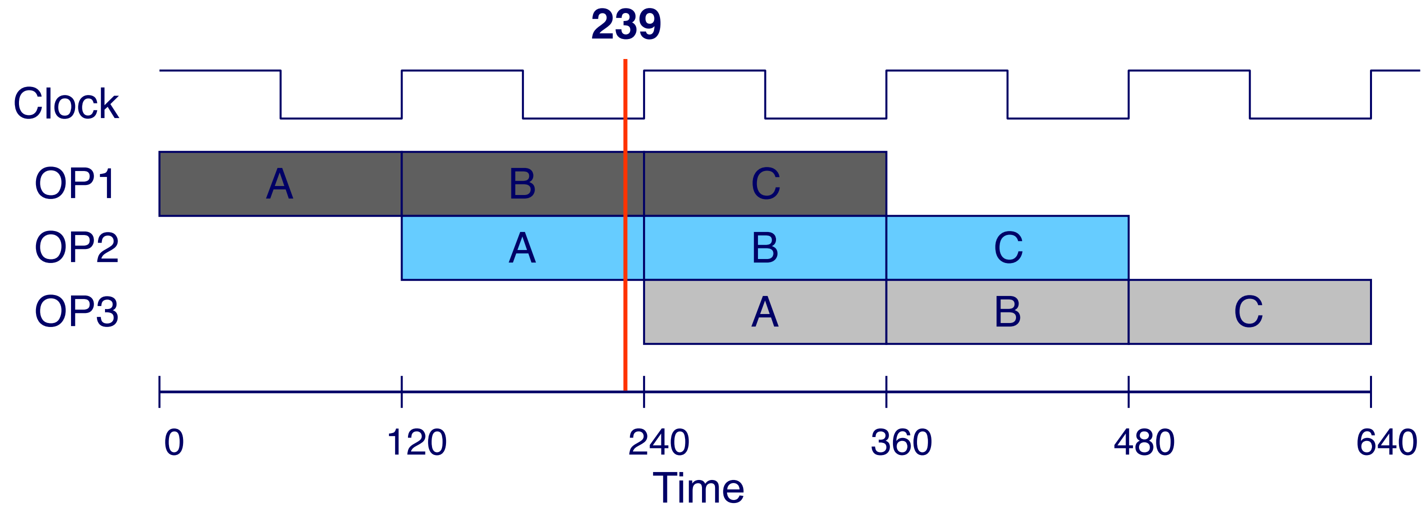
Pipeline Illustration

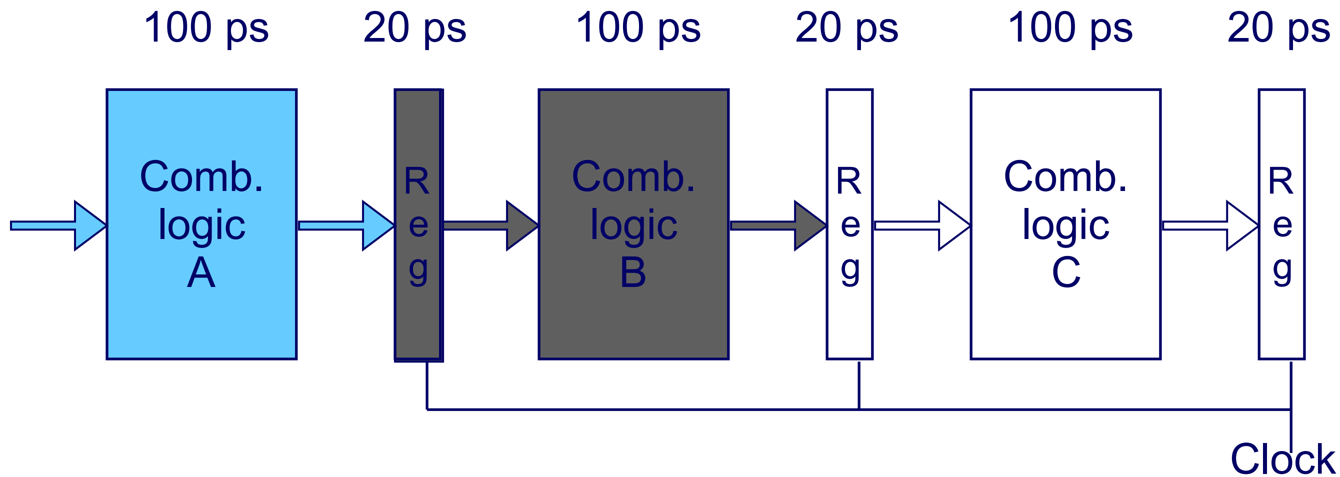


Pipeline Illustration

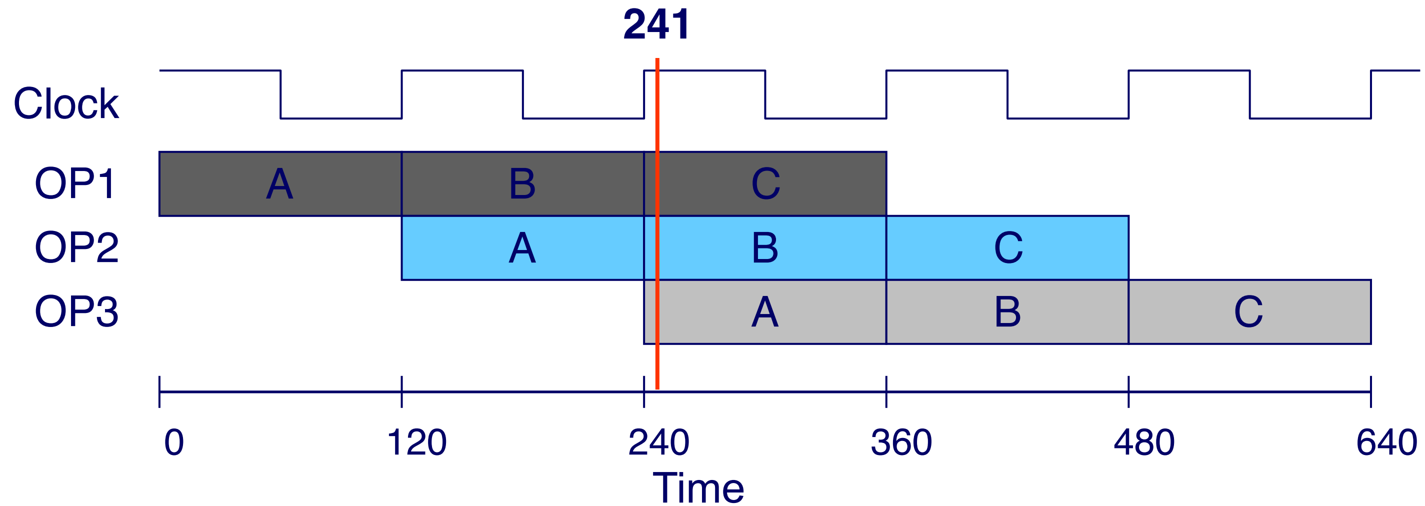


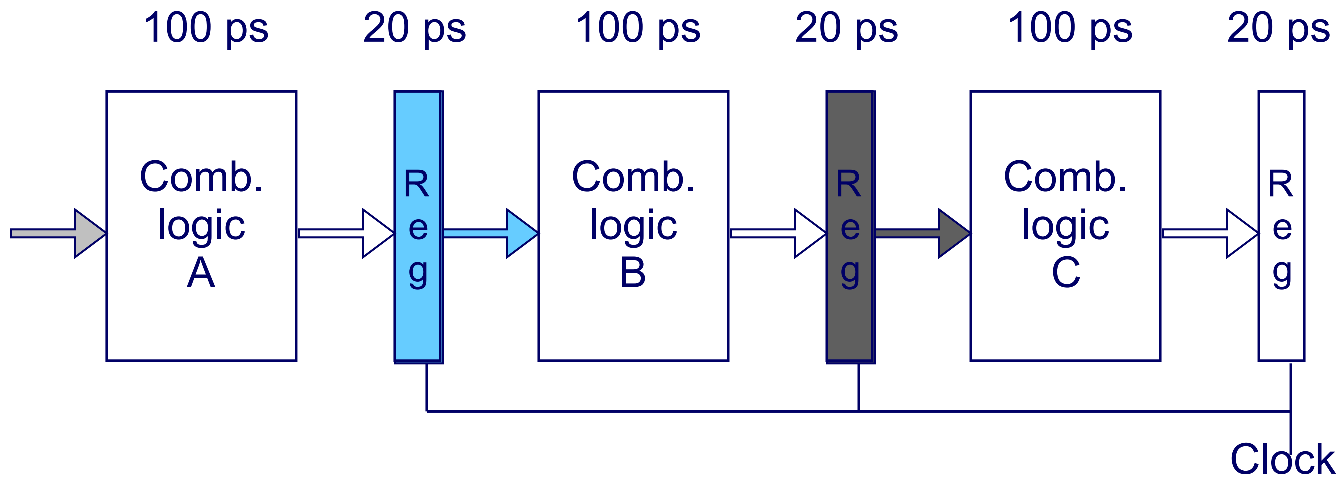
Another Illustration



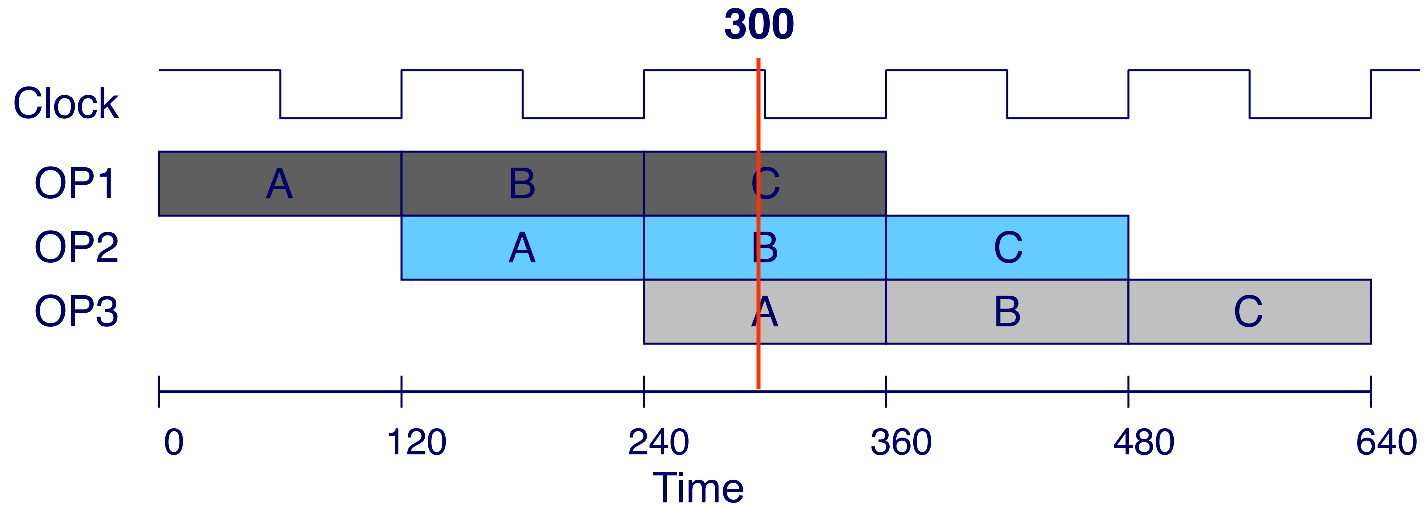


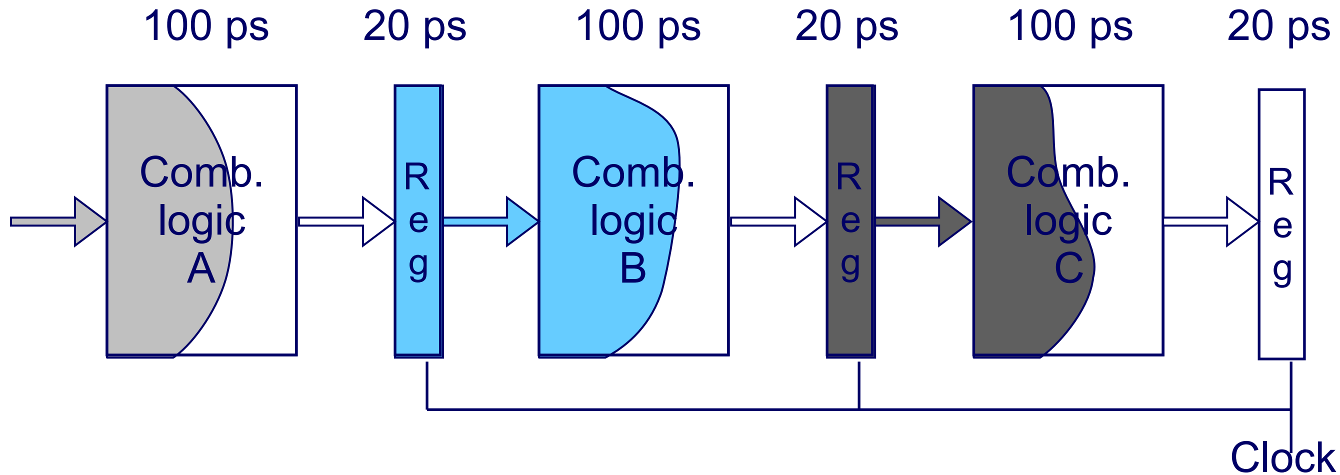
Another Illustration



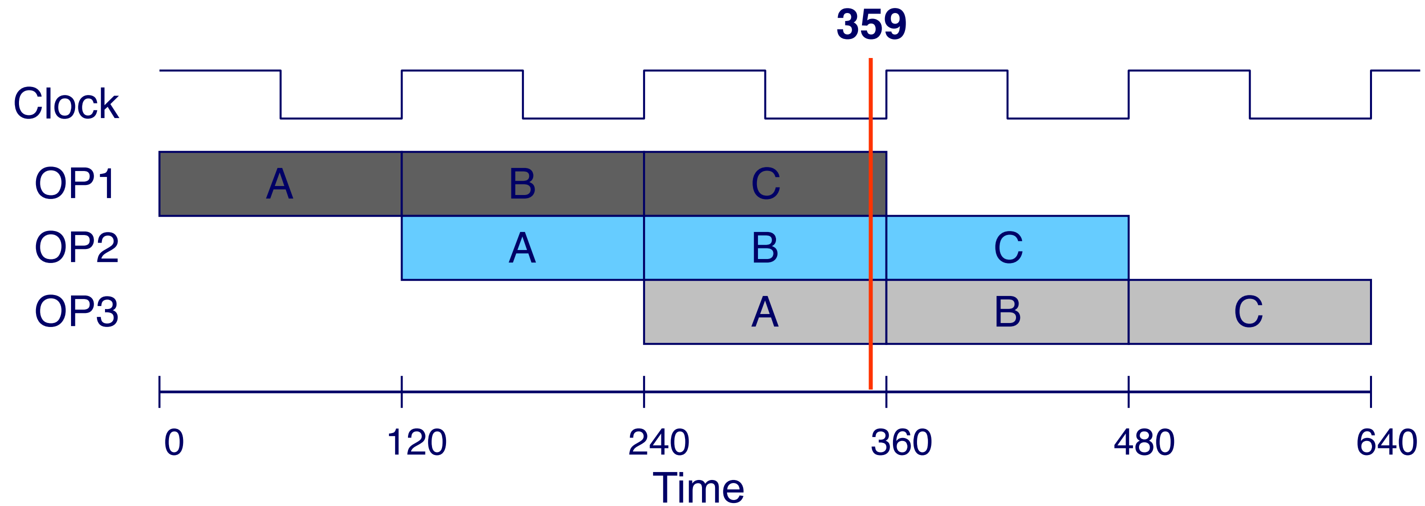


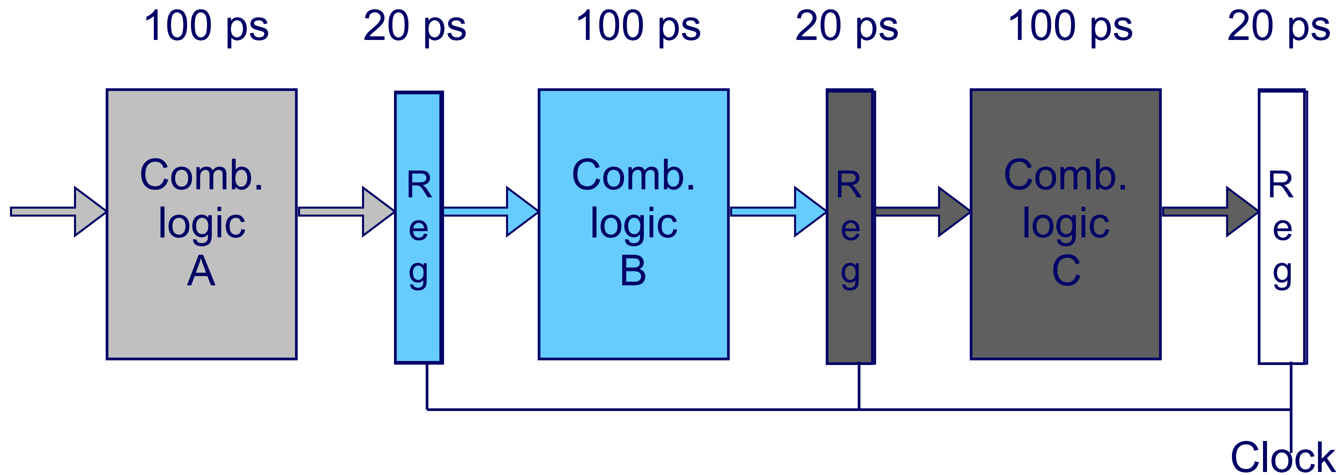
Another Illustration





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 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

```
     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 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

```
     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 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

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

	1	2
xorg %rax, %rax	F	D
jne L1		F

Control Dependency

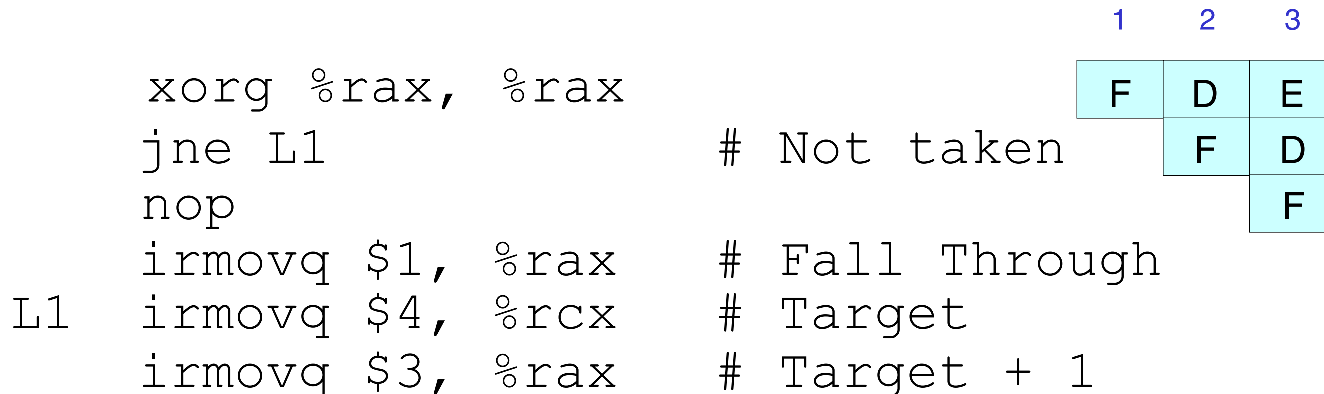
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
    - `jne L1` determines whether `irmovq $1, %rax` should be executed
    - But `jne` doesn't know its outcome until after its Execute stage

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

1	2	3
F	D	E
	F	D

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



Control Dependency

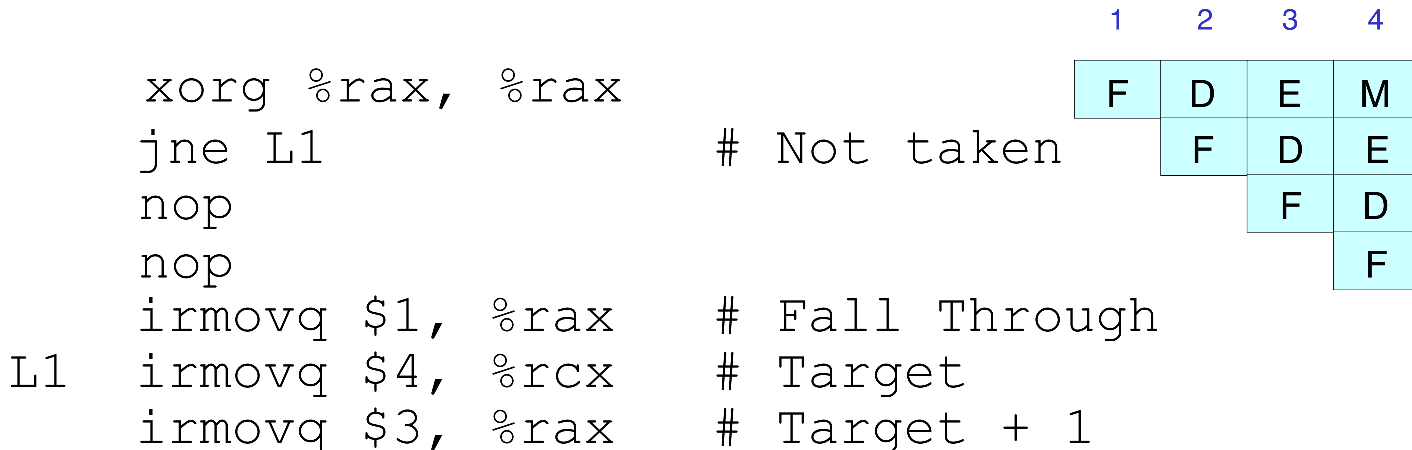
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

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

1	2	3	4
F	D	E	M
	F	D	E
		F	D

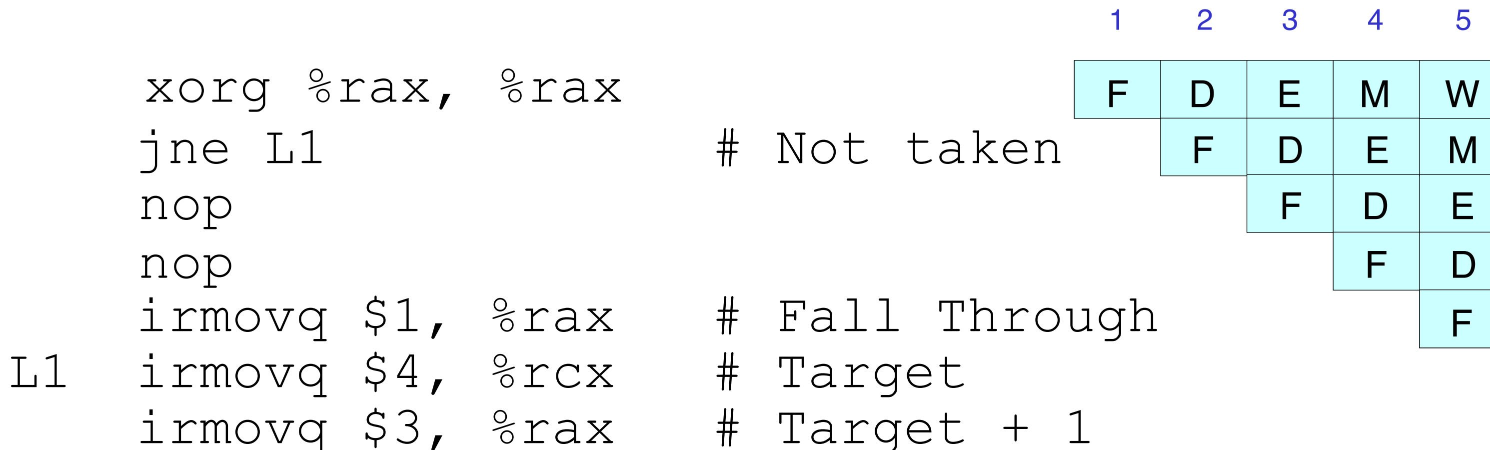
Control Dependency

- **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



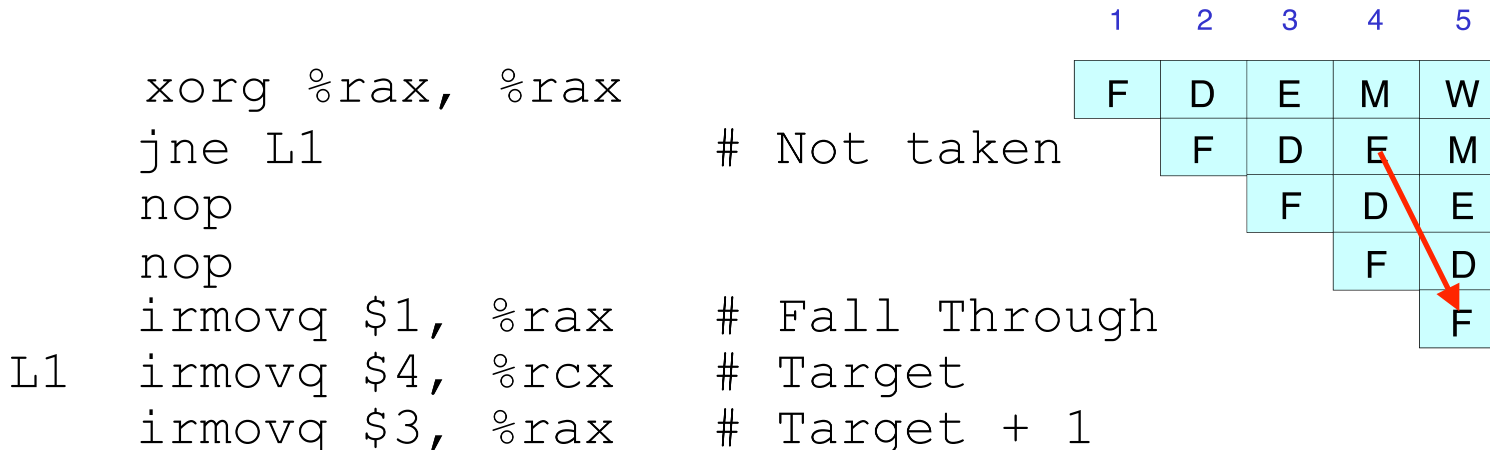
Control Dependency

- **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



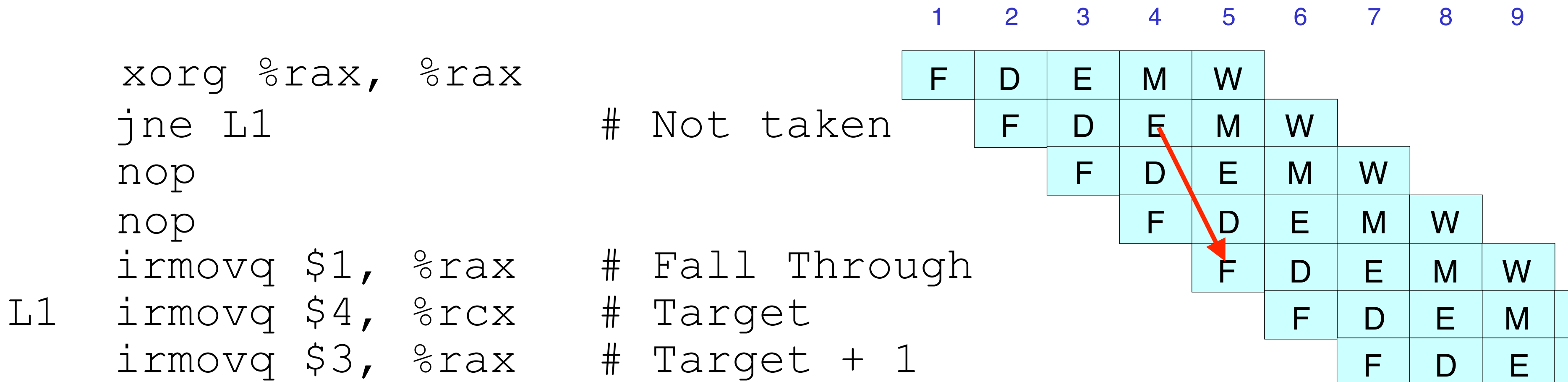
Control Dependency

- **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



Control Dependency

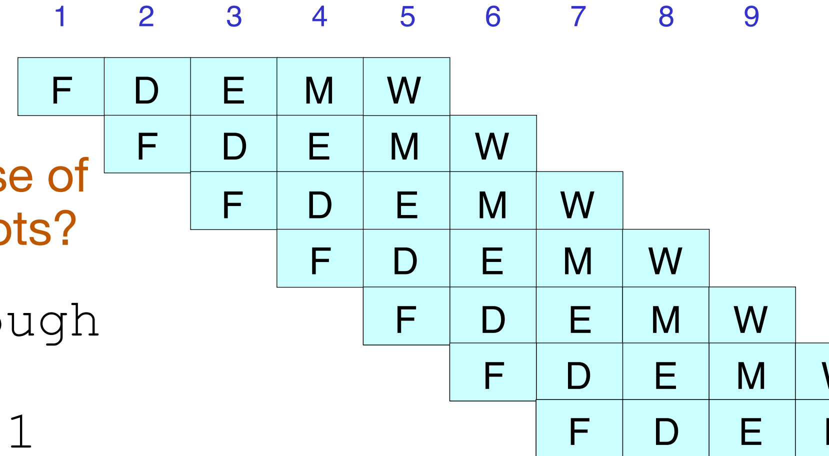
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage



Delay Slots

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

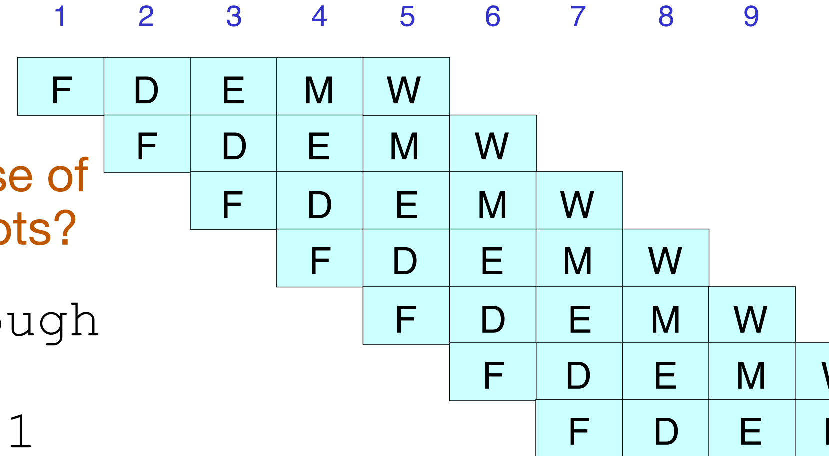
Can we make use of the 2 wasted slots?



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?

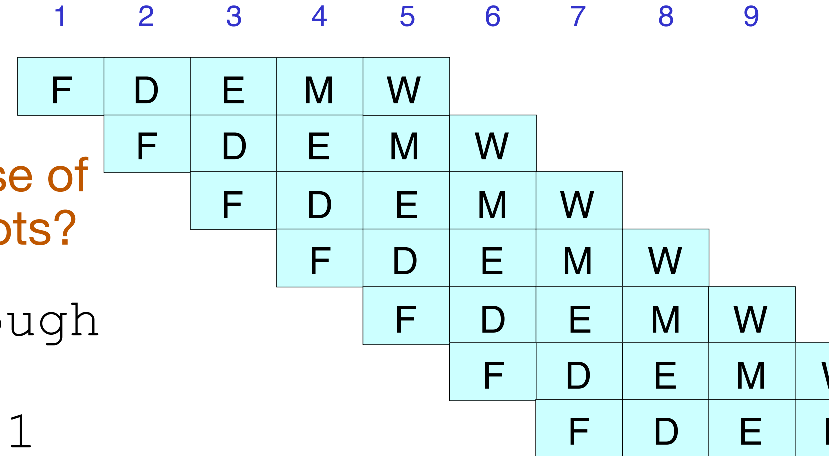



```
if (cond) {
  do_A();
} else {
  do_B();
}
do_C();
```

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?



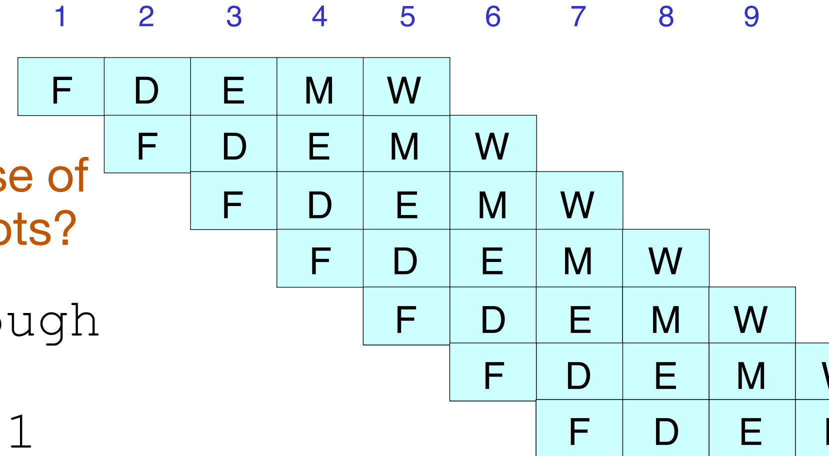

```
if (cond) {
  do_A();
} else {
  do_B();
}
do_C();
```

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



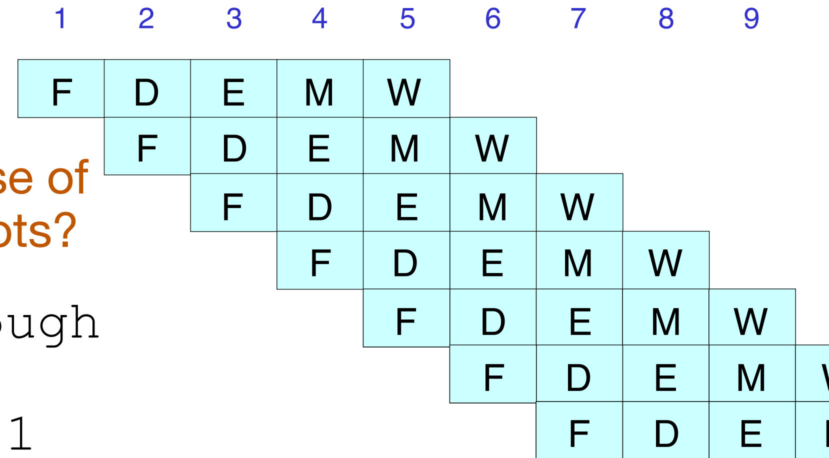
A less obvious example

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

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

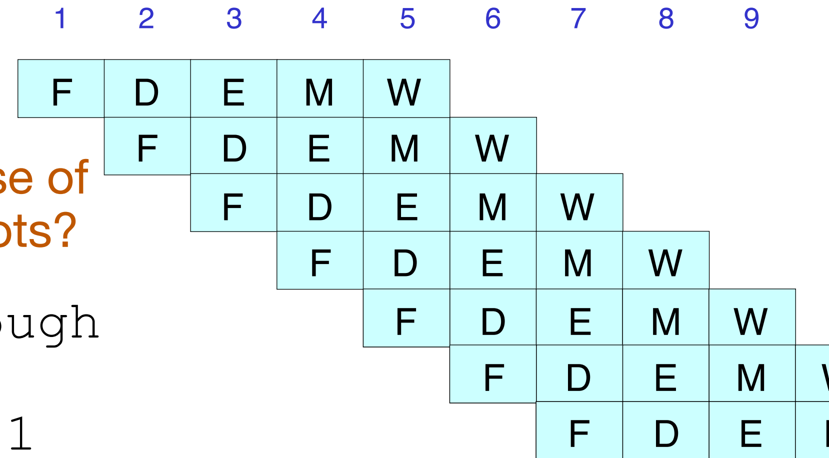
```
do_C();
if (cond) {
   do_A();
} else {
   do_B();
}
```

```
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
```

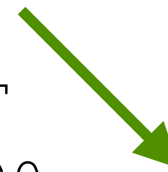
Can we make use of the 2 wasted slots?



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
```

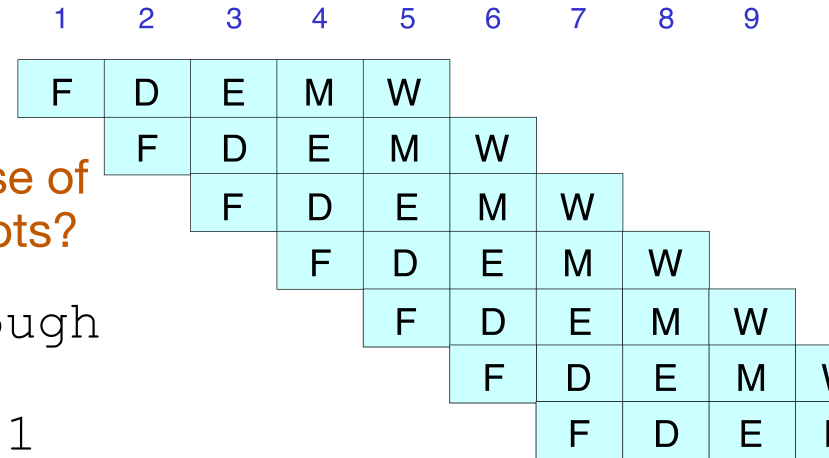


```
add A, B
sub E, F
jle 0x200
or C, D
add A, C
```

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