

CSC 252: Computer Organization

Spring 2020: Lecture 13

Substitute Instructor: Sandhya Dwarkadas

Department of Computer Science
University of Rochester

Announcement

- Programming assignment 3 due soon
 - Details:
<https://www.cs.rochester.edu/courses/252/spring2020/labs/assignment3.html>
 - Due on **Feb. 28**, 11:59 PM
 - You (may still) have 3 slip days

17	18	19	20	21	22
24	25	26	27	28	29
			Today	Due	

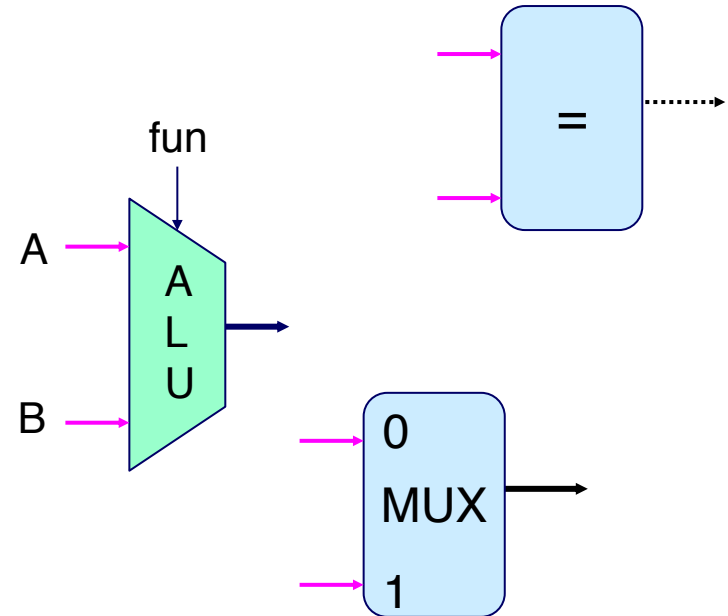
Announcement

- 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.

Building Blocks

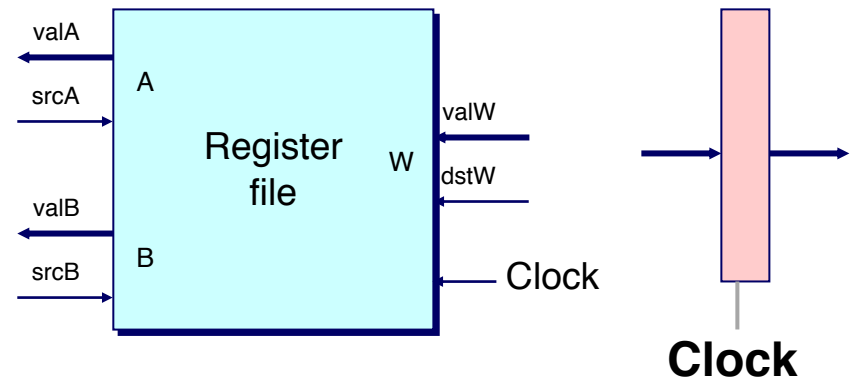
Combinational Logic

- Compute Boolean functions of inputs
- Continuously respond to input changes
- Operate on data and implement control

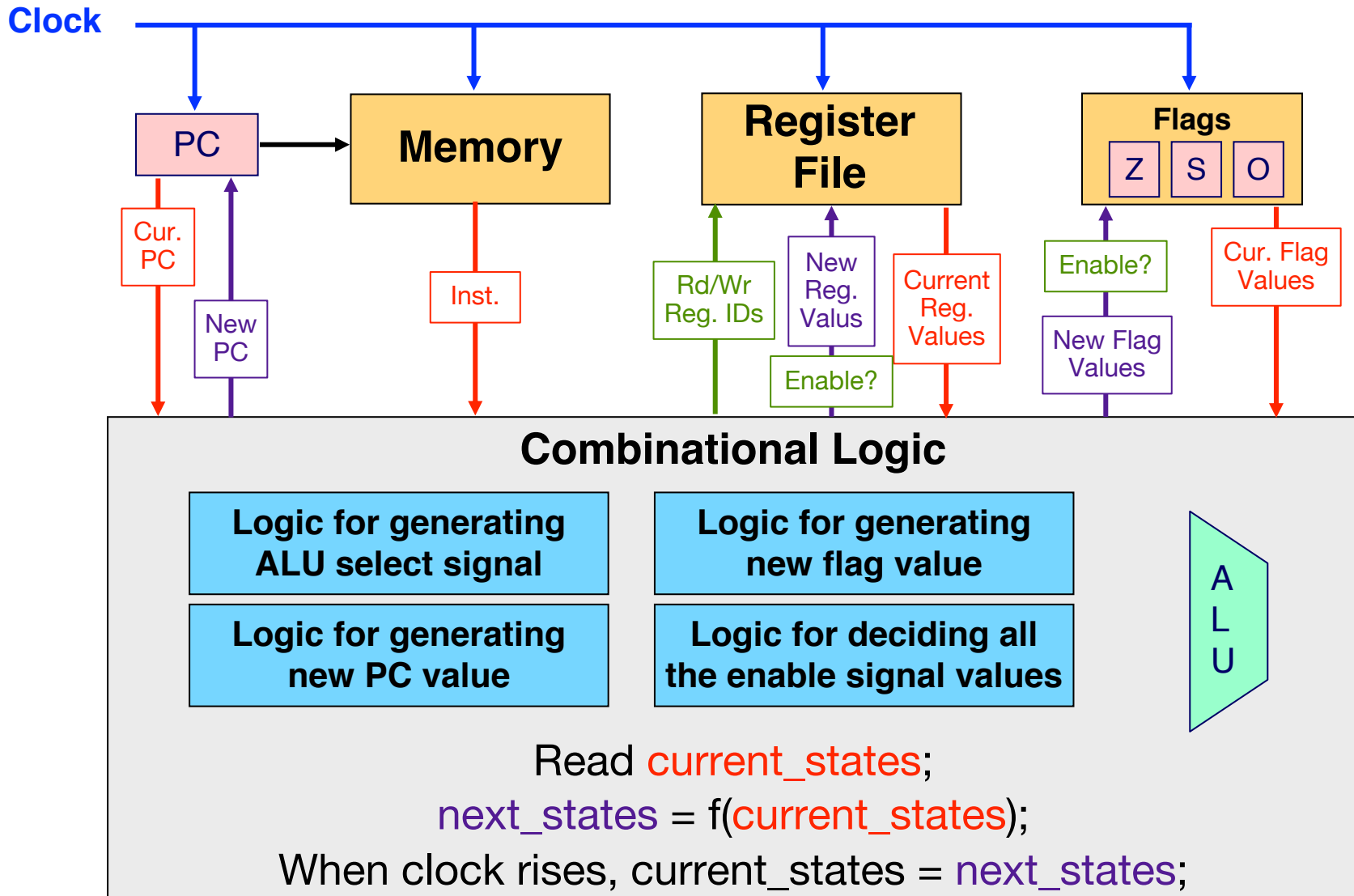


Storage Elements

- Store bits
- Addressable memories
- Non-addressable registers
- Loaded only as clock rises

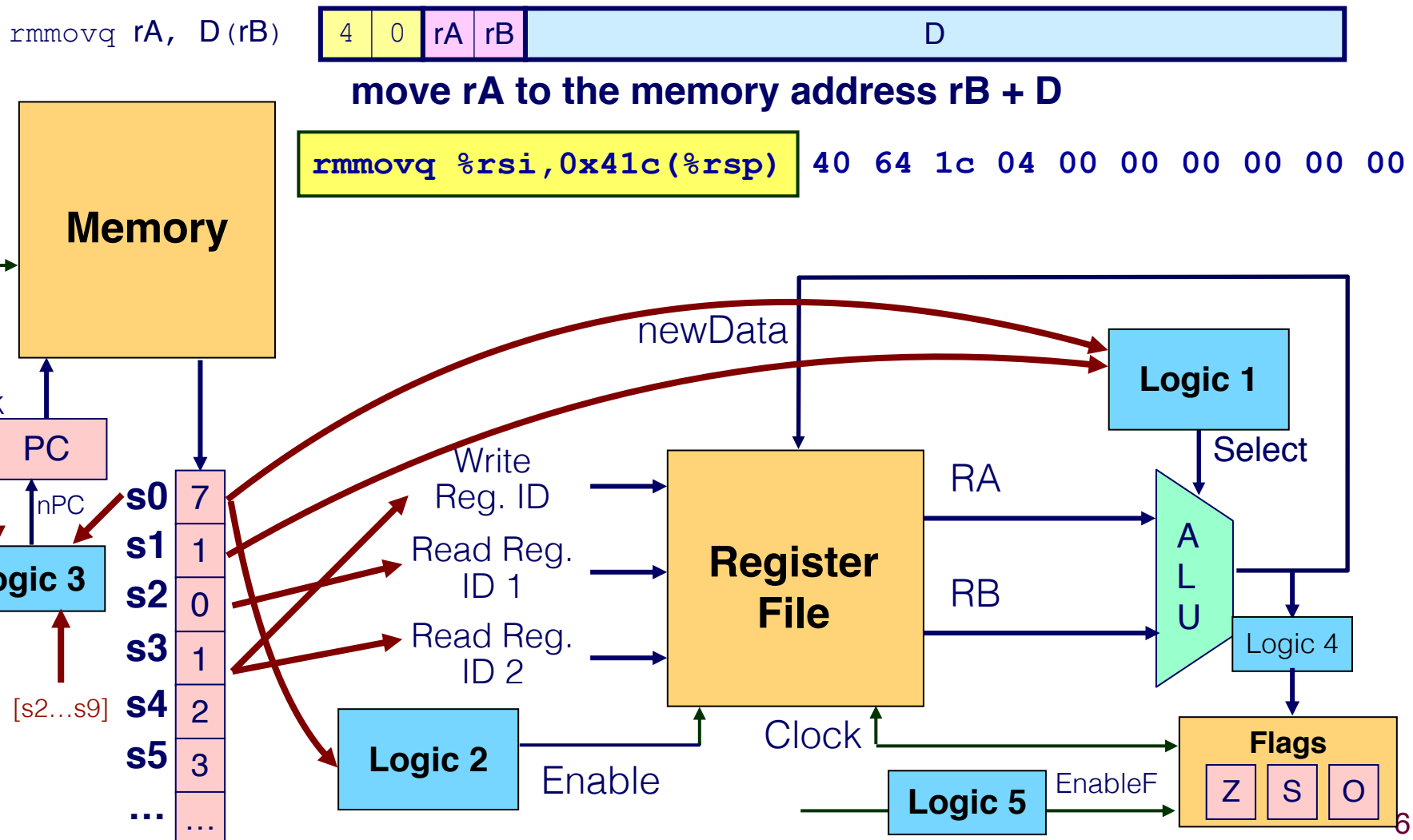


Microarchitecture (So far): Single Cycle



Executing a MOV instruction

- How do we modify the hardware to execute a move instruction?

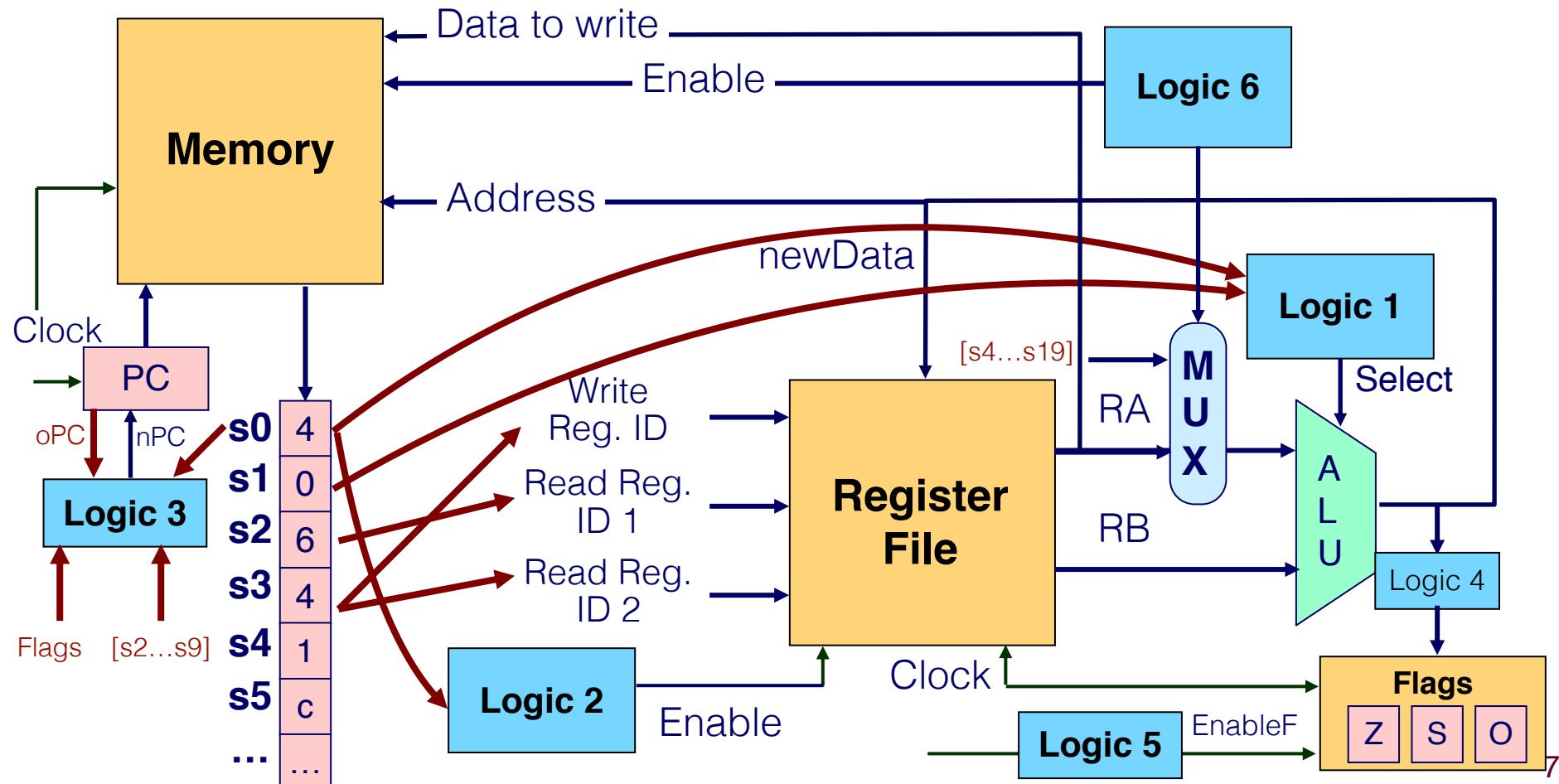


move rA to the memory address rB + D

rmmovq rA, D(rB)



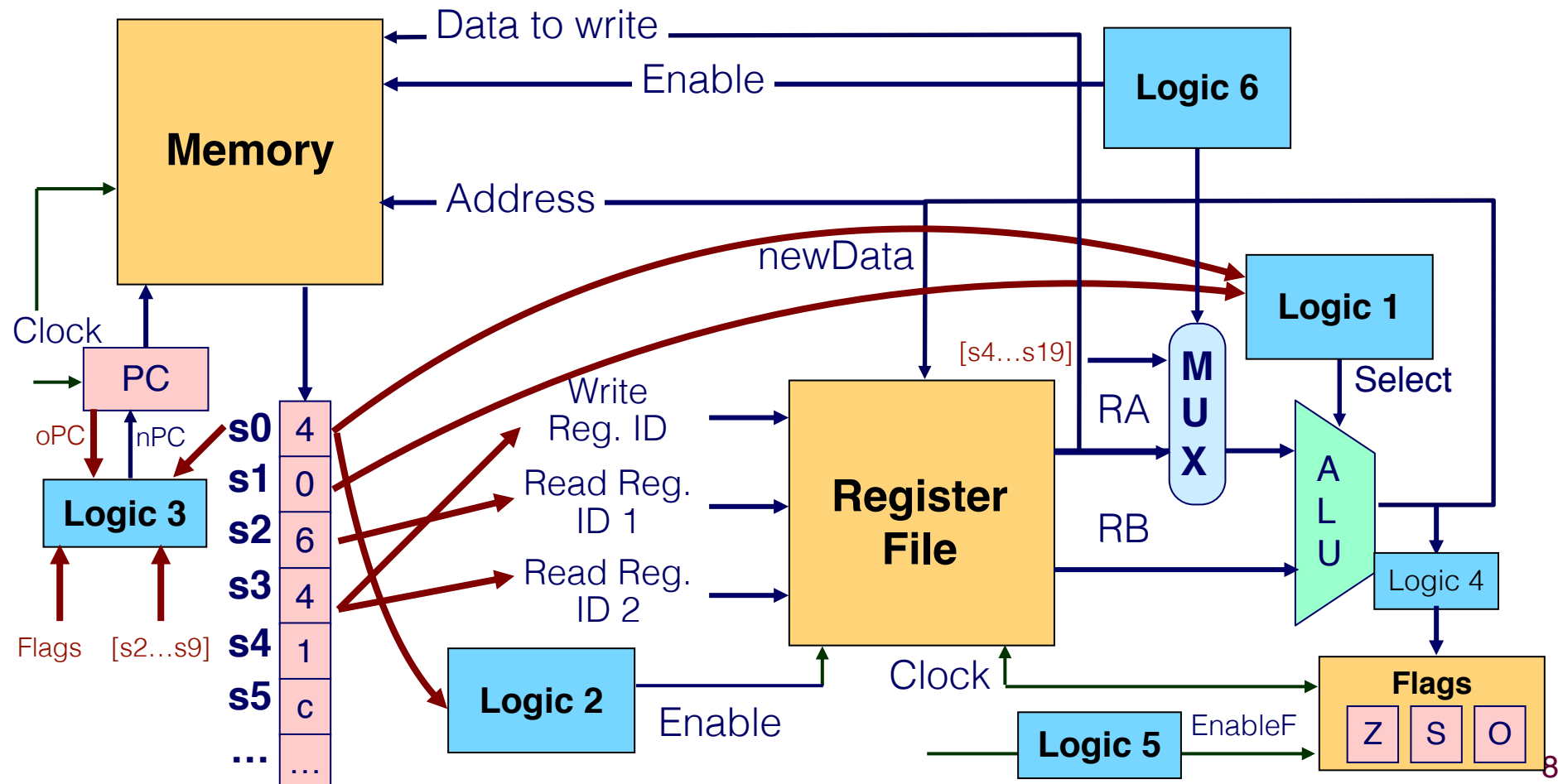
- Need new logic (Logic 6) to select the input to the ALU for Enable.
- How about other logics?



How About Memory to Register MOV?

move data at memory address $rB + D$ to rA

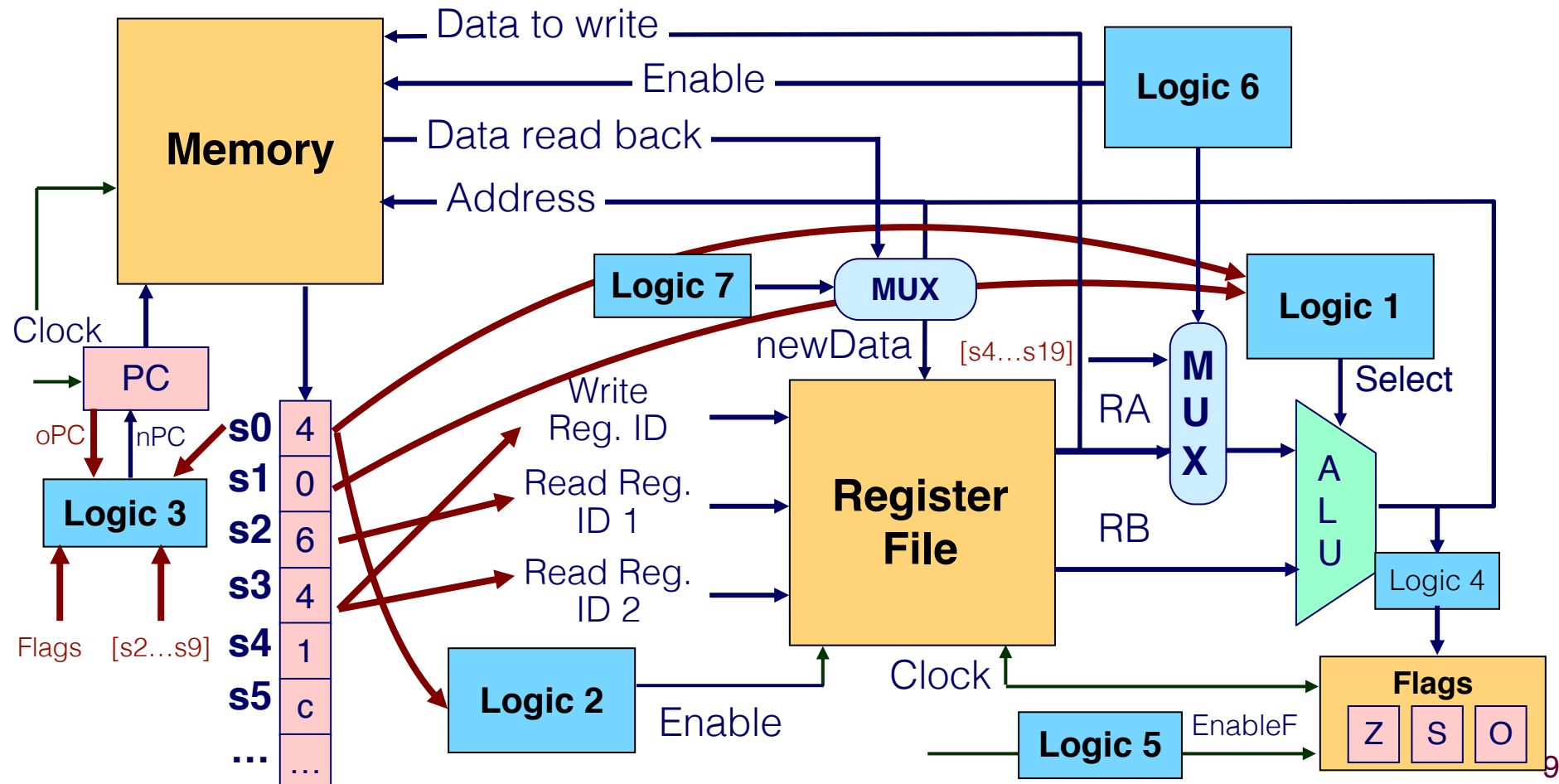
`mrmovq D(rB), rA`



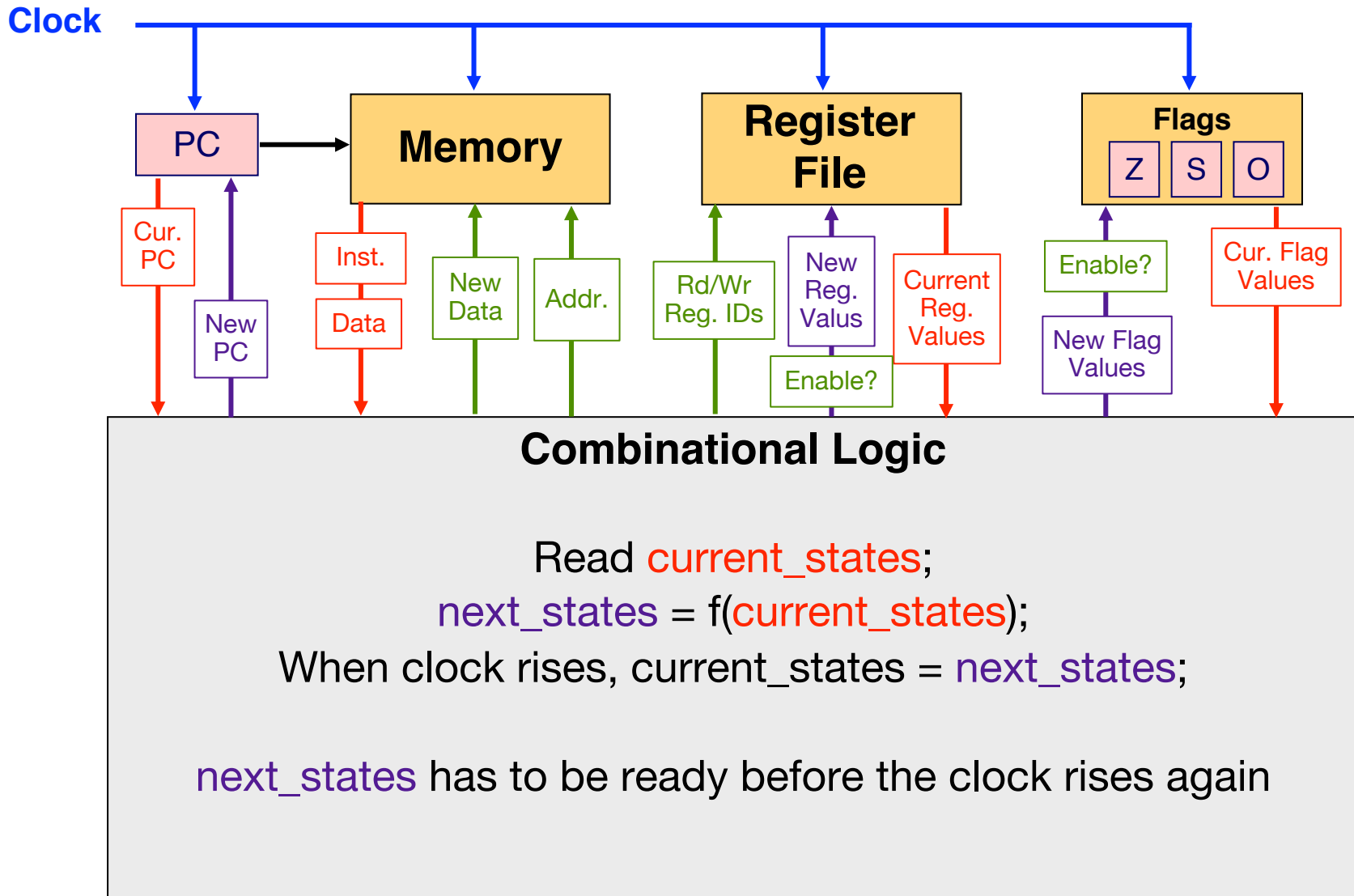
How About Memory to Register MOV?

move data at memory address $rB + D$ to rA

`mrmovq D(rB), rA`



Microarchitecture (with MOV)



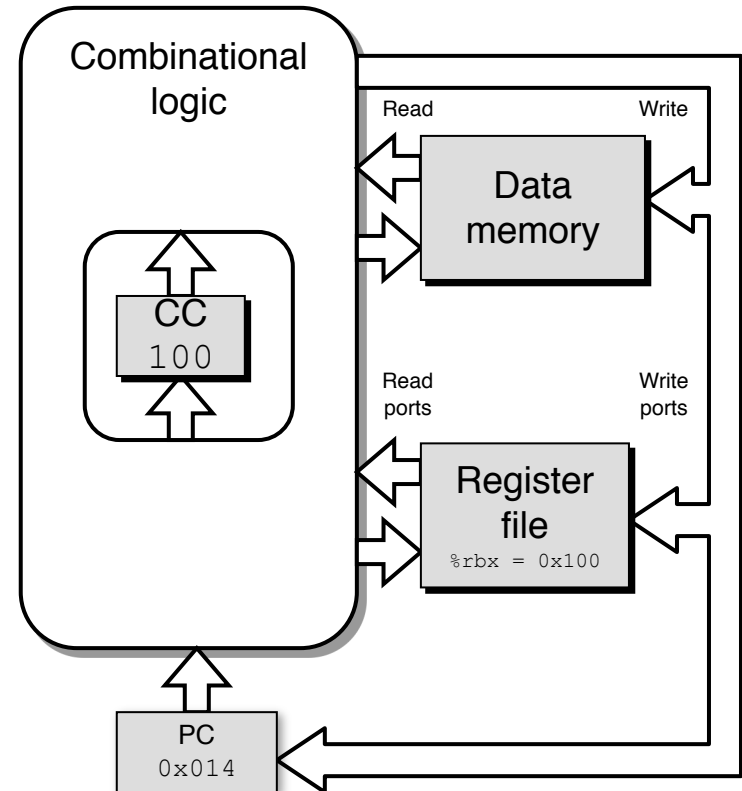
Microarchitecture Overview

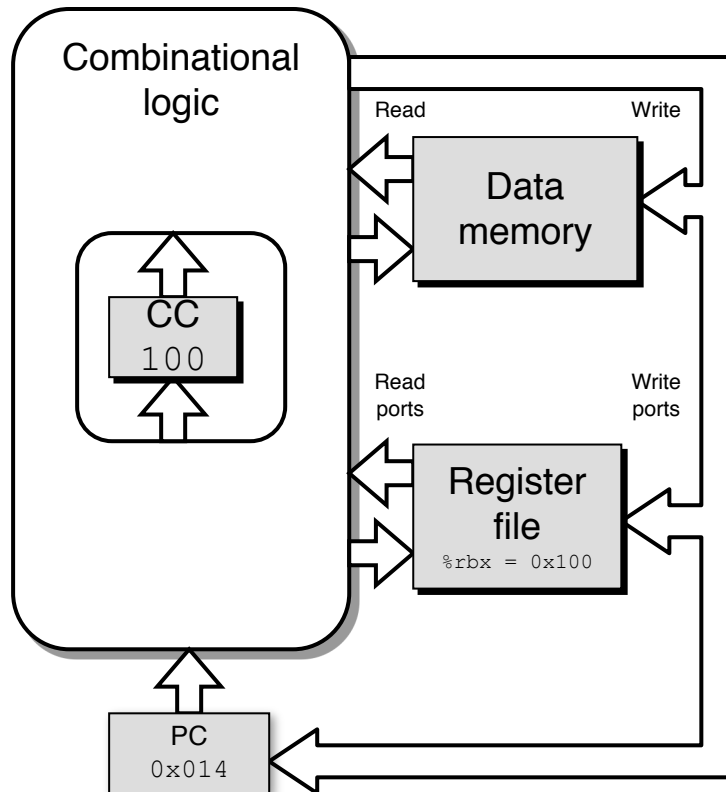
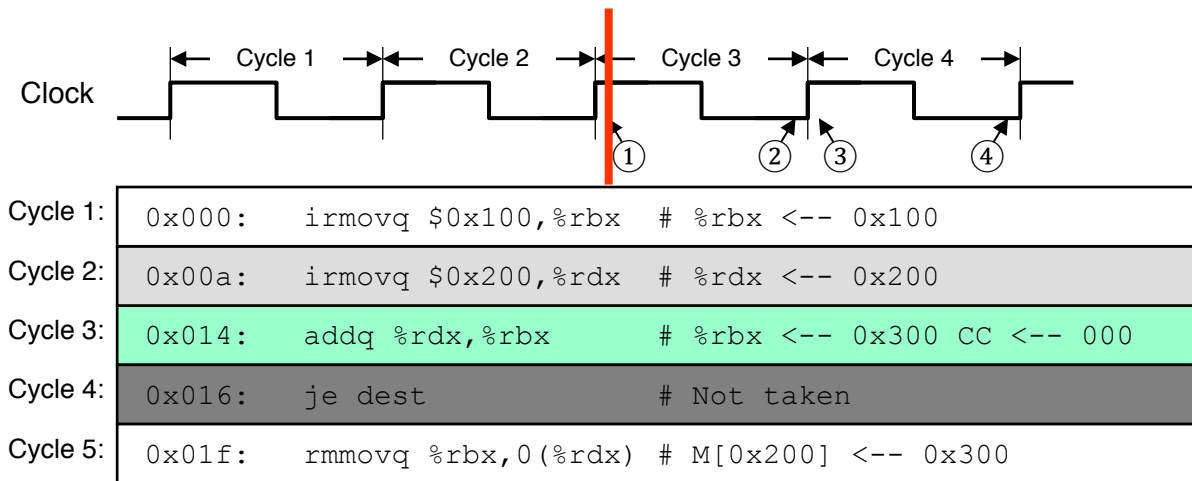
Think of it as a state machine

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

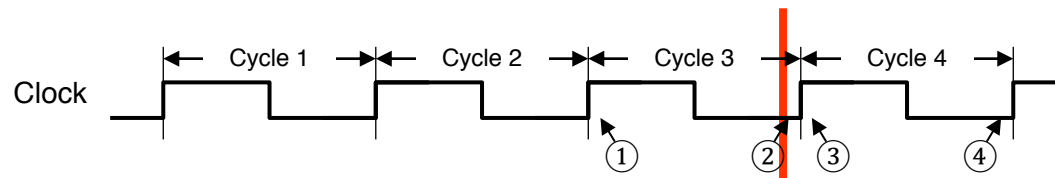
States (All updated as clock rises)

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

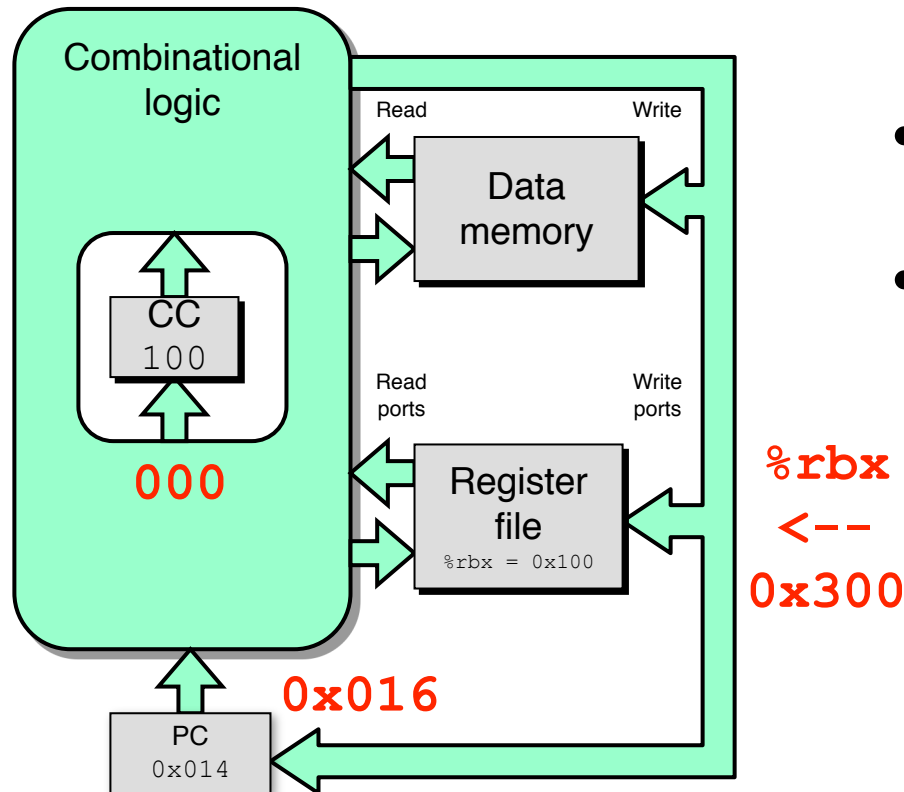




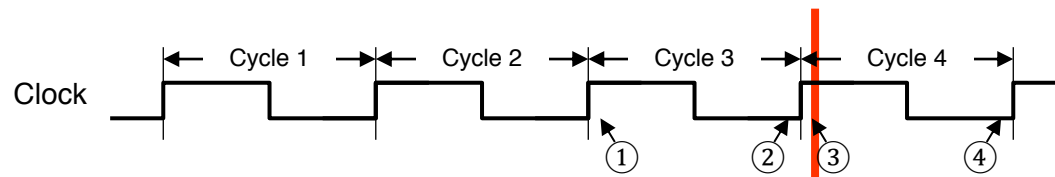
- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes



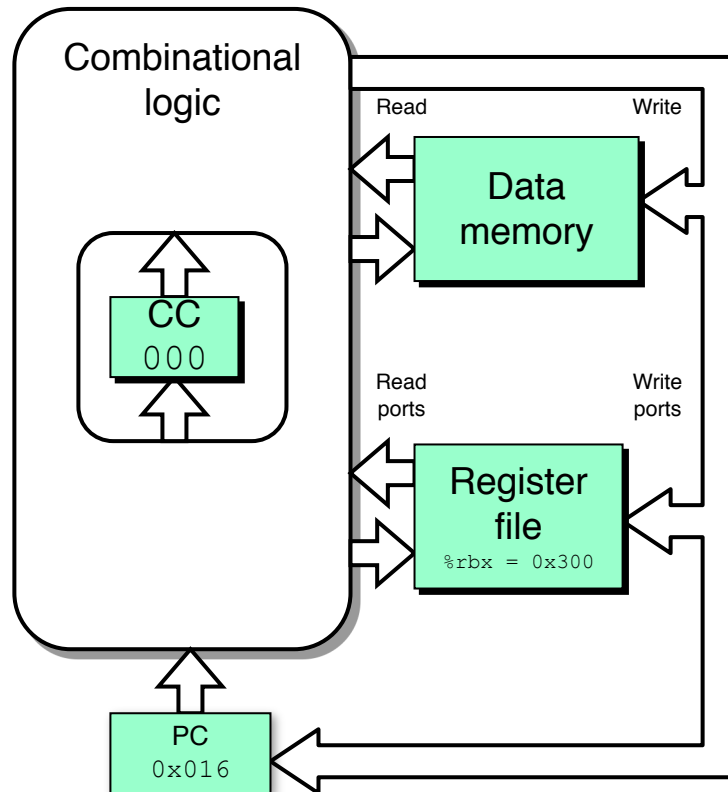
Cycle 1:	0x000:	<code>irmovq \$0x100,%rbx</code>	# %rbx <-- 0x100
Cycle 2:	0x00a:	<code>irmovq \$0x200,%rdx</code>	# %rdx <-- 0x200
Cycle 3:	0x014:	<code>addq %rdx,%rbx</code>	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	<code>je dest</code>	# Not taken
Cycle 5:	0x01f:	<code>rmmovq %rbx,0(%rdx)</code>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



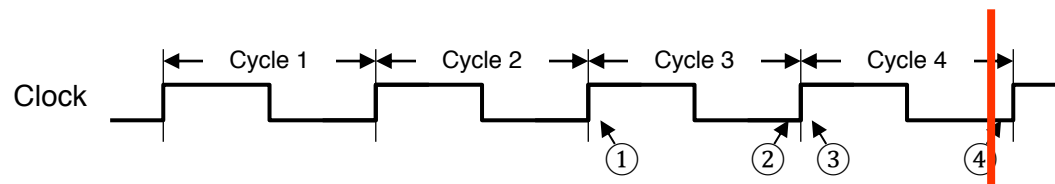
- state set according to second `irmovq` instruction
- combinational logic generates results for `addq` instruction



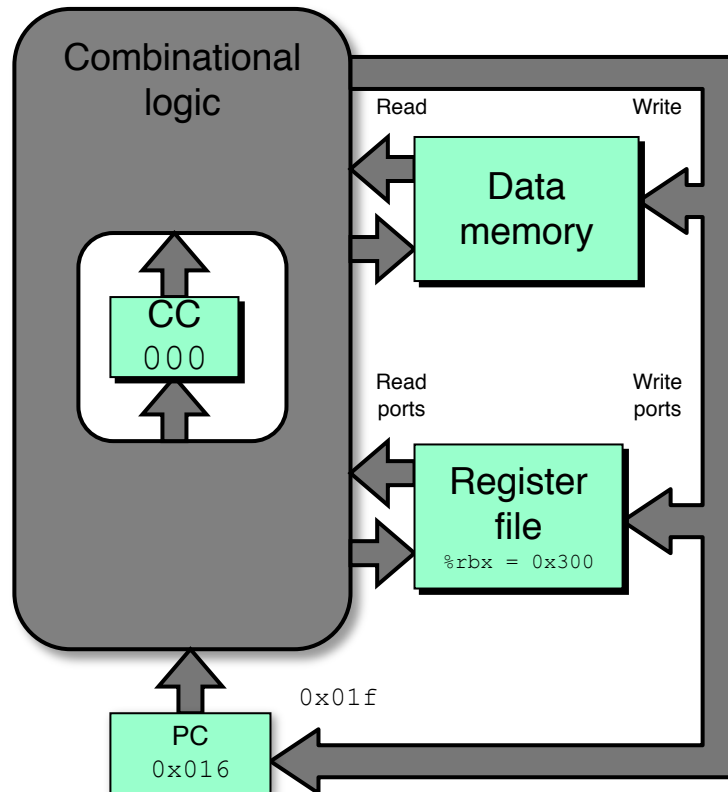
Cycle 1:	0x000:	irmovq \$0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq \$0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



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



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



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

Another Way to Look At the Microarchitecture

Principles:

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

Fetch: Read instruction from instruction memory

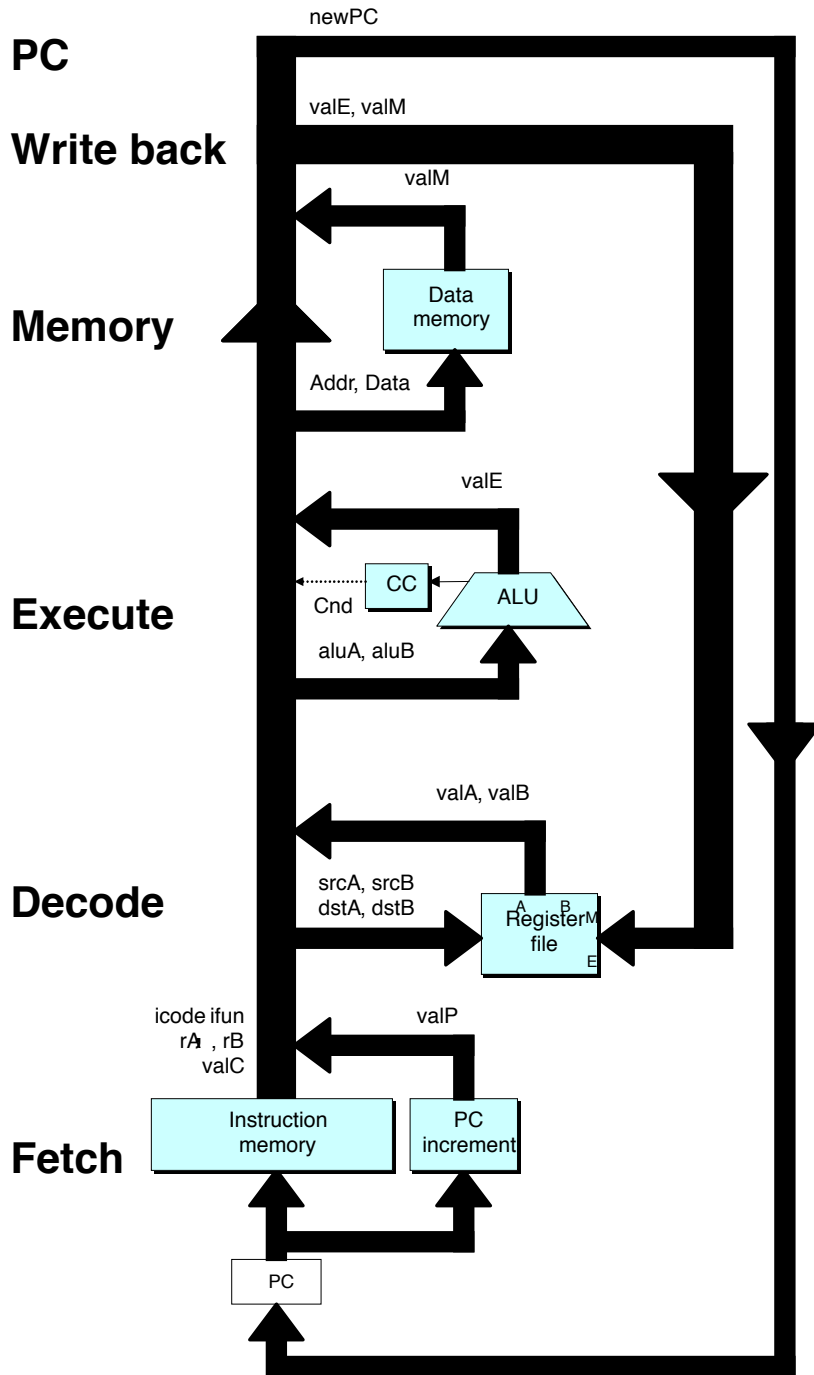
Decode: Read program registers

Execute: Compute value or address

Memory: Read or write data

Write Back: Write program registers

PC: Update program counter



Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

- Read or write data

Write Back

- Write program registers

PC

- Update program counter

Stage Computation: Arith/Log. Ops



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

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



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

Stage Computation: Jumps

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

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

Processor Microarchitecture

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

Real-World Pipelines: Car Washes

Sequential



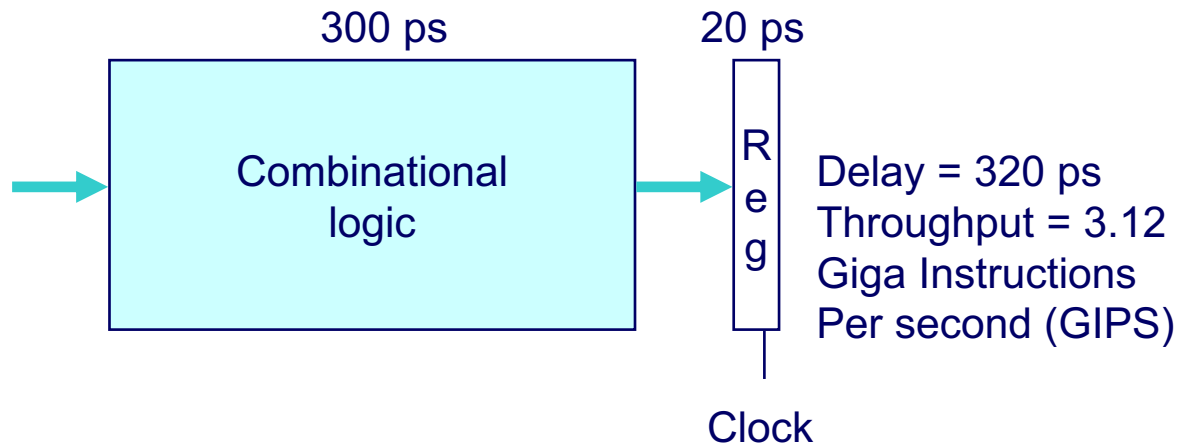
Pipelined



Idea

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

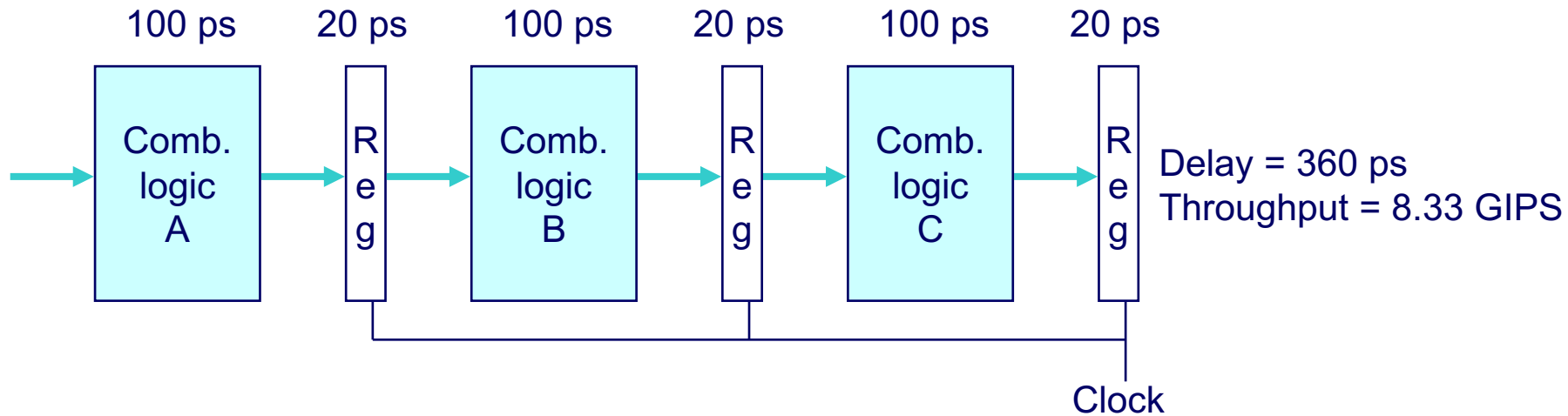
Computational Example



System

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

3-Stage Pipelined Version

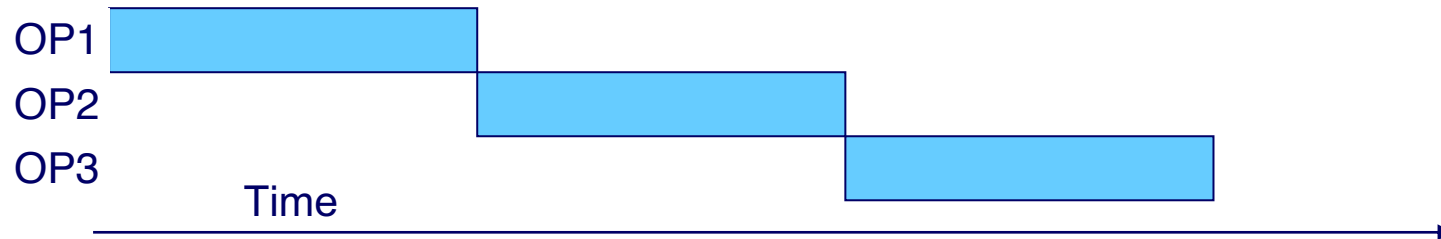


System

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

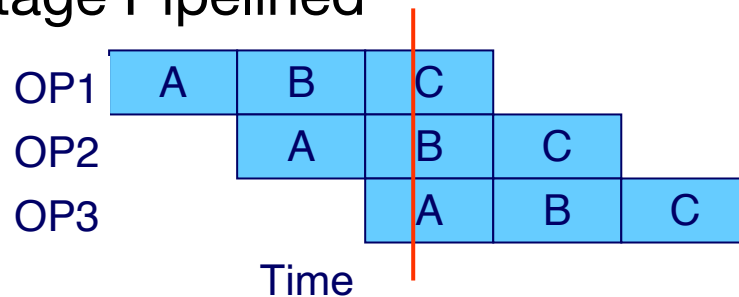
Pipeline Diagrams

Unpipelined



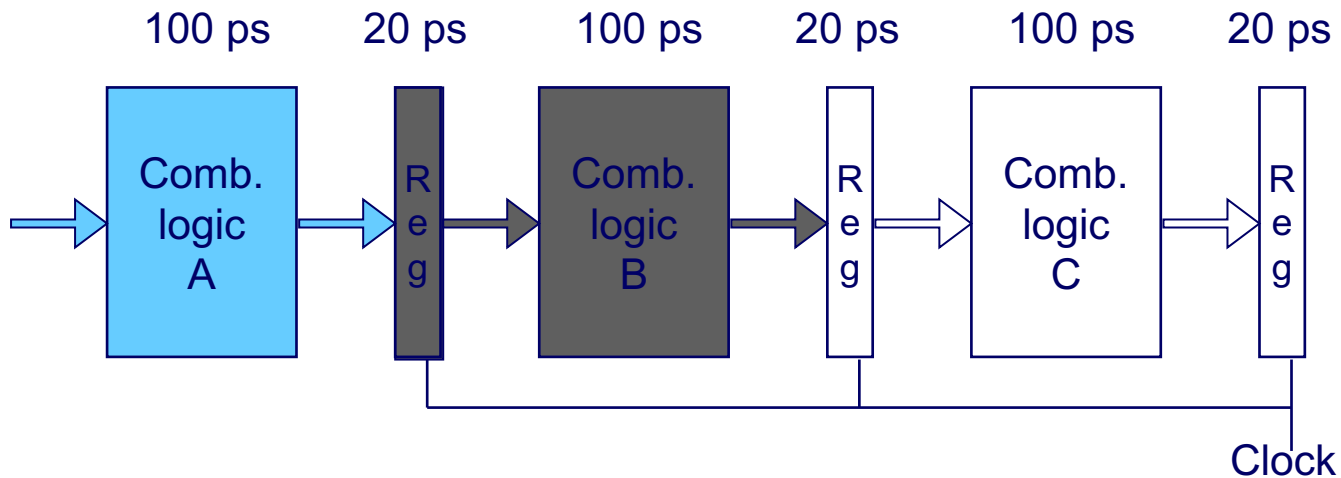
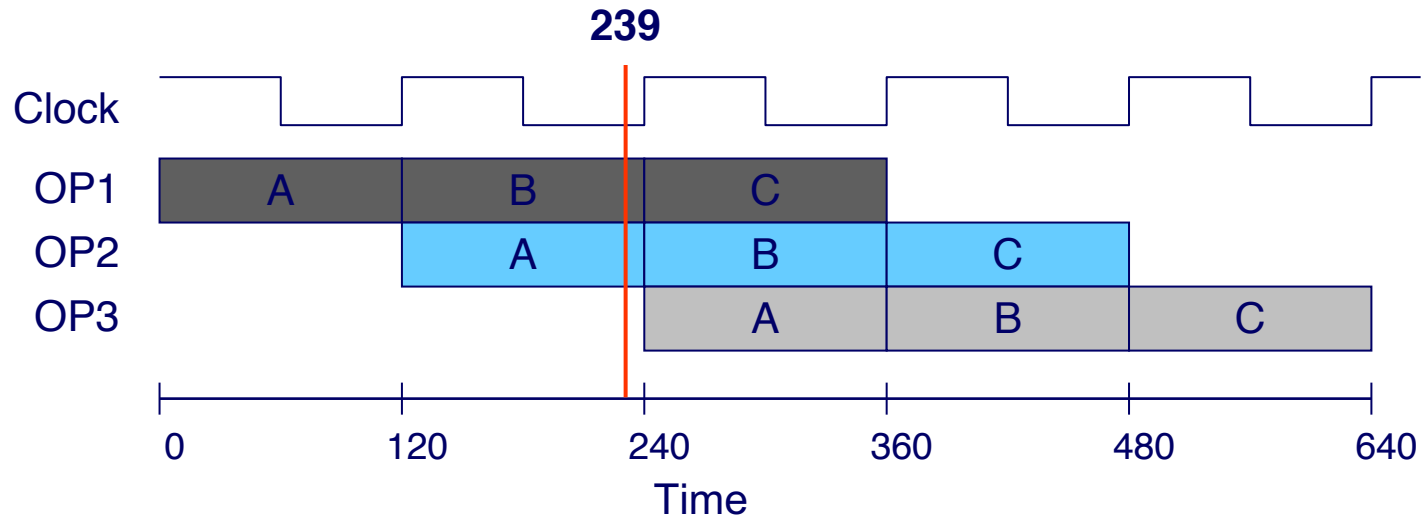
- Cannot start new operation until previous one completes

3-Stage Pipelined

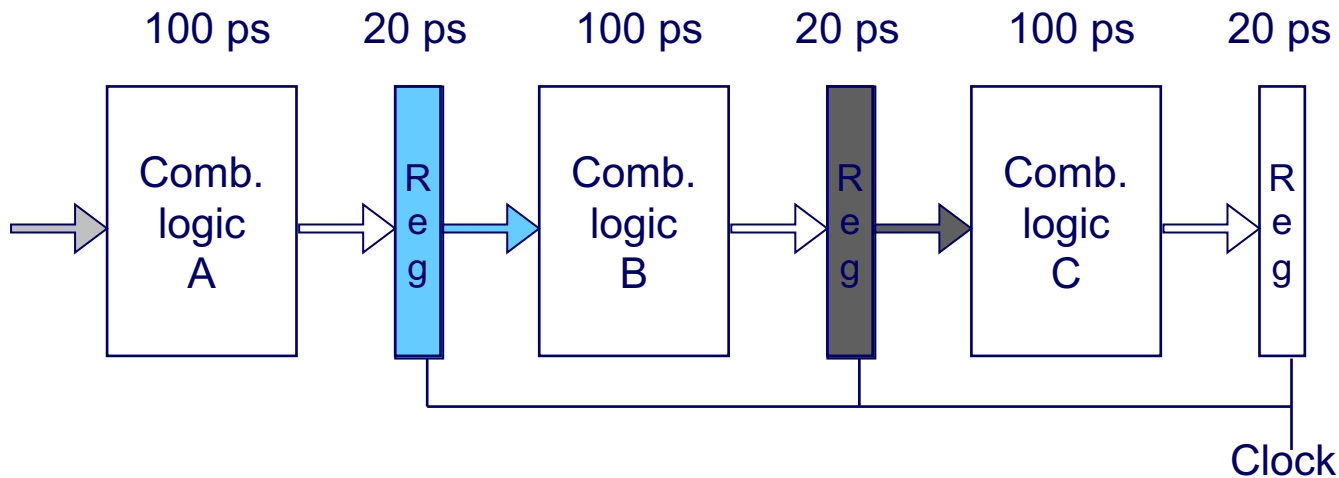
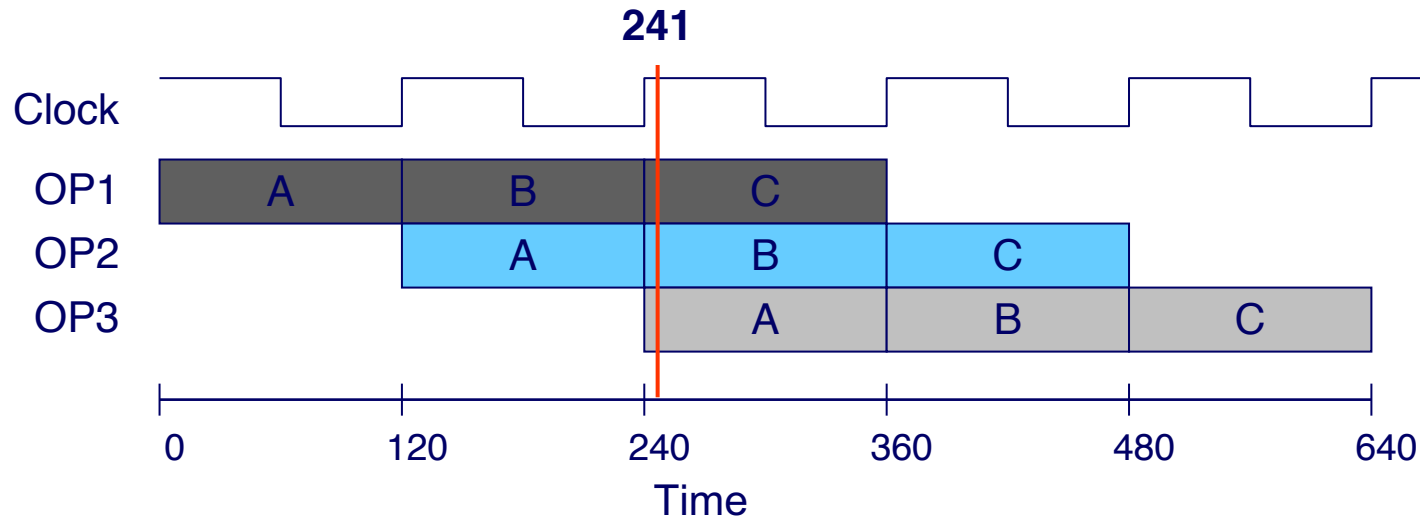


- Up to 3 operations in process simultaneously

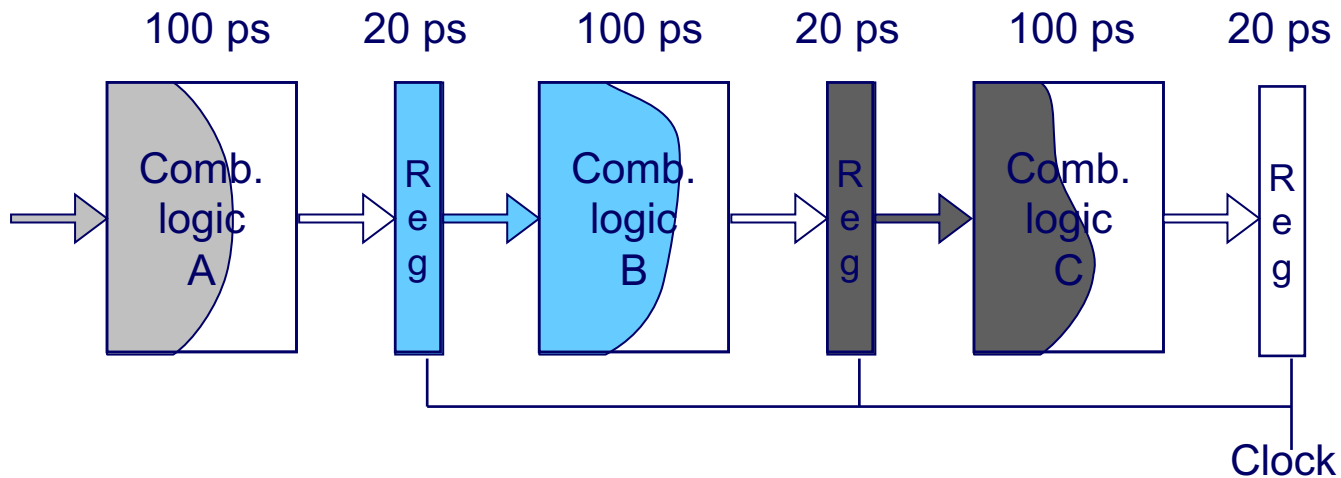
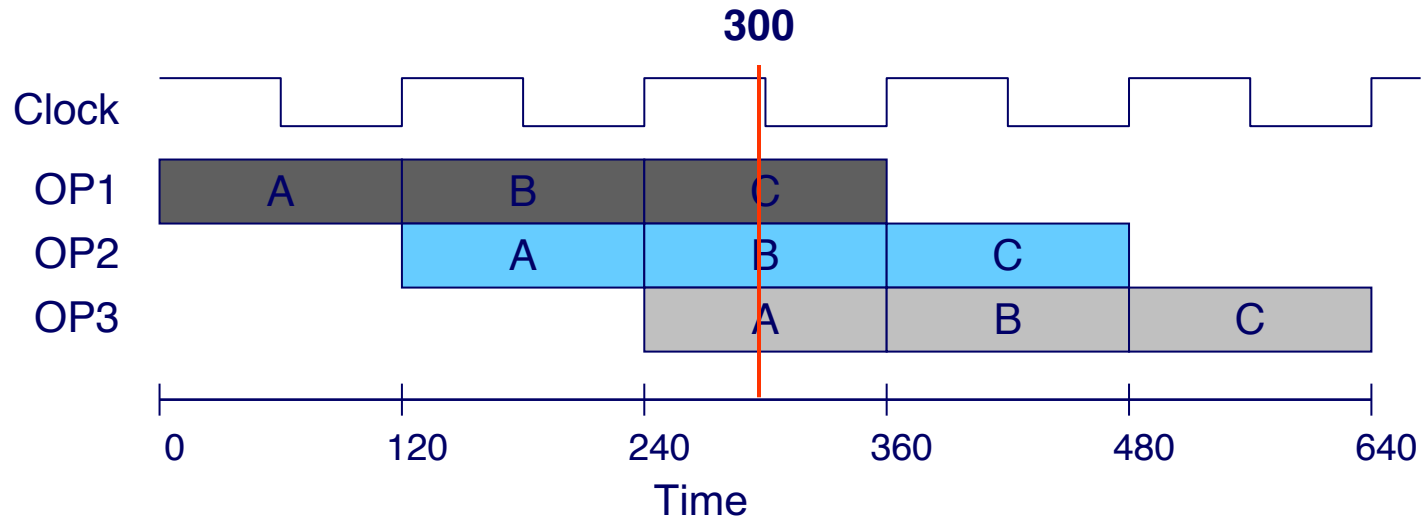
Operating a Pipeline



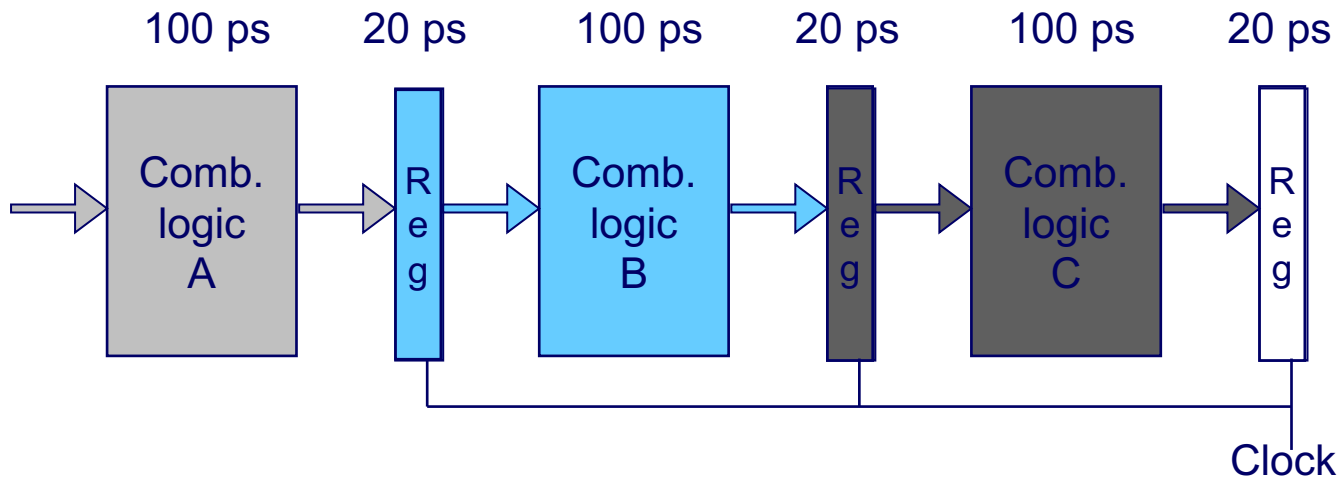
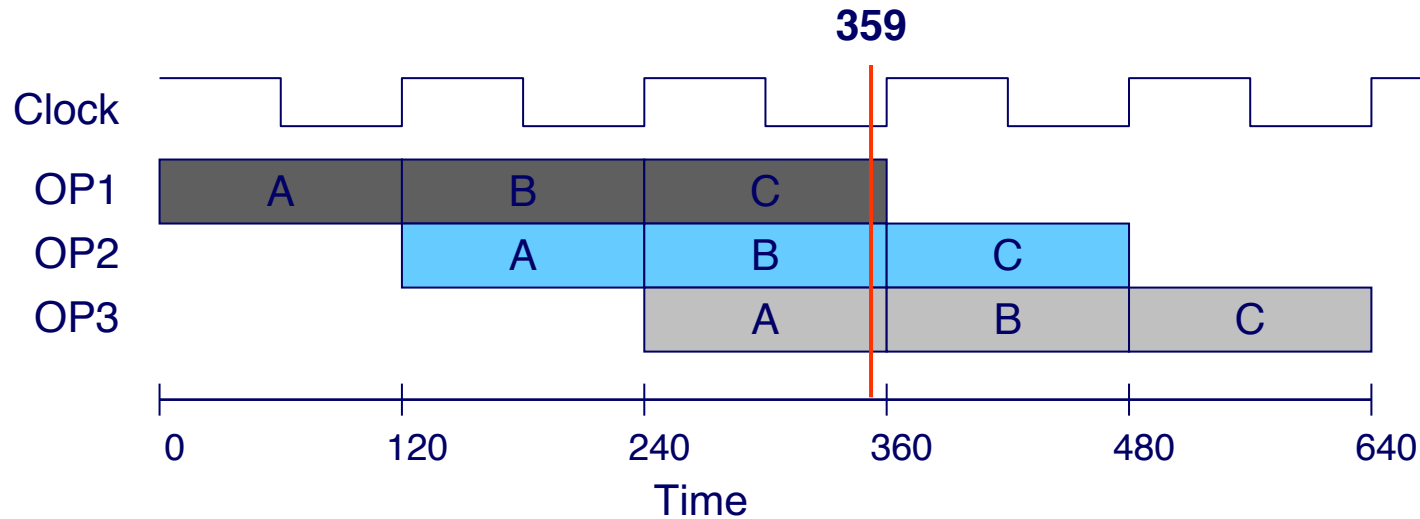
Operating a Pipeline



Operating a Pipeline



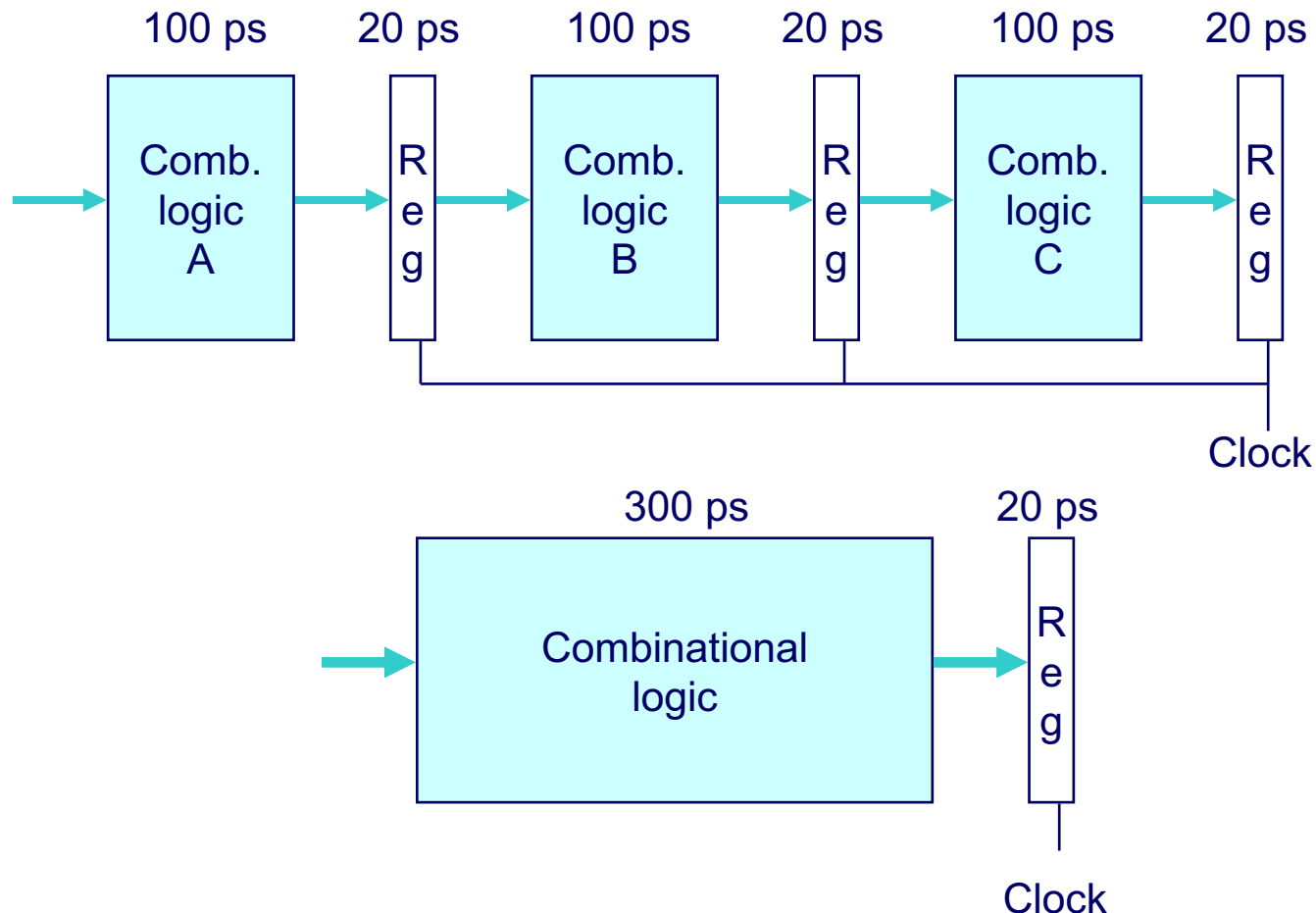
Operating a Pipeline



Pipeline Trade-offs

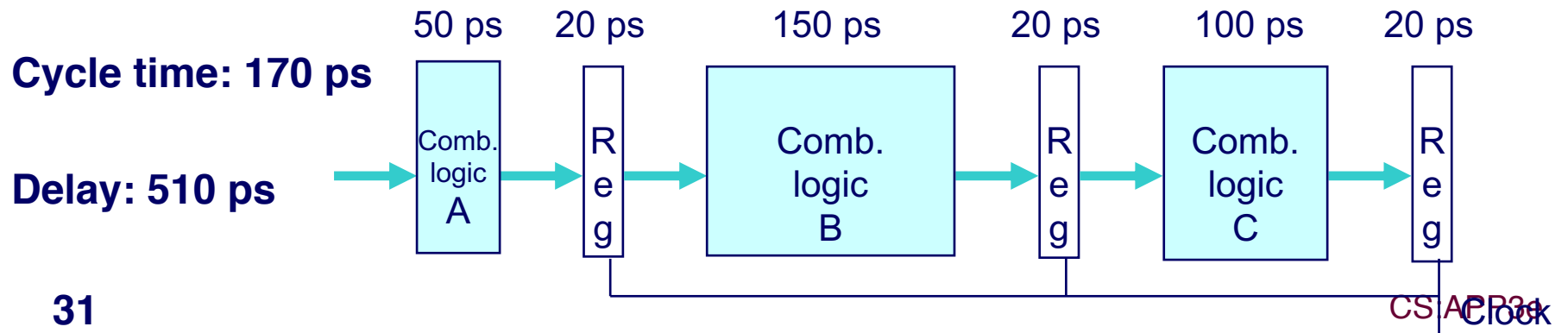
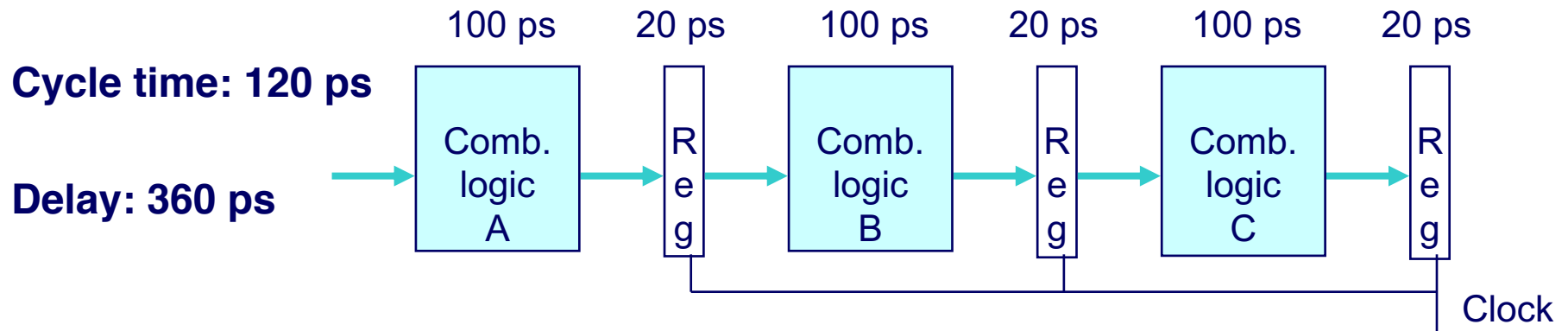
Pros: Increase throughput. Can process more instructions in a given time span.

Cons: Increase latency as new registers are needed between pipeline stages.



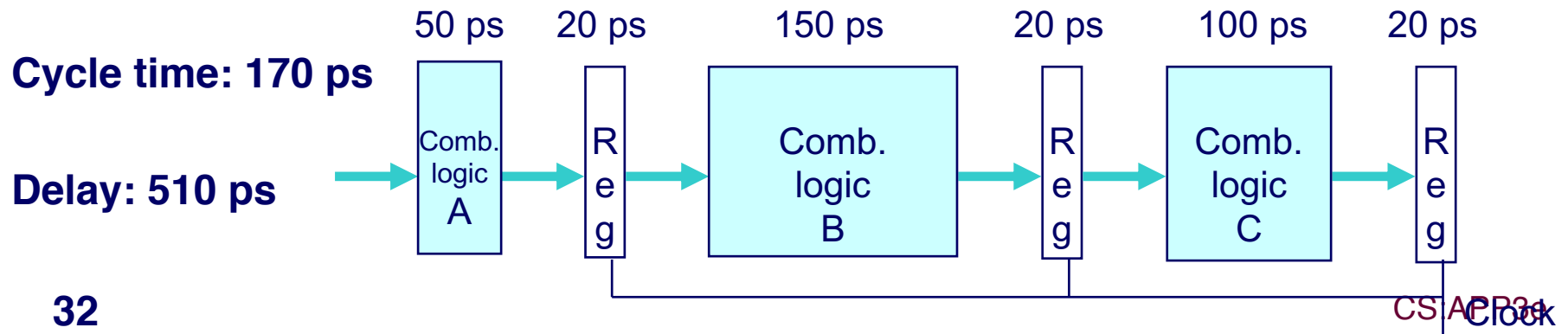
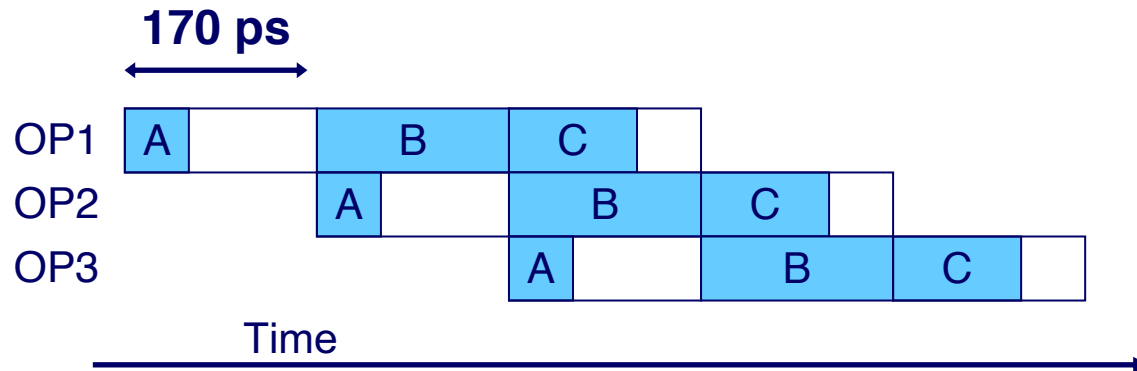
Unbalanced Pipeline

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



Unbalanced Pipeline

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

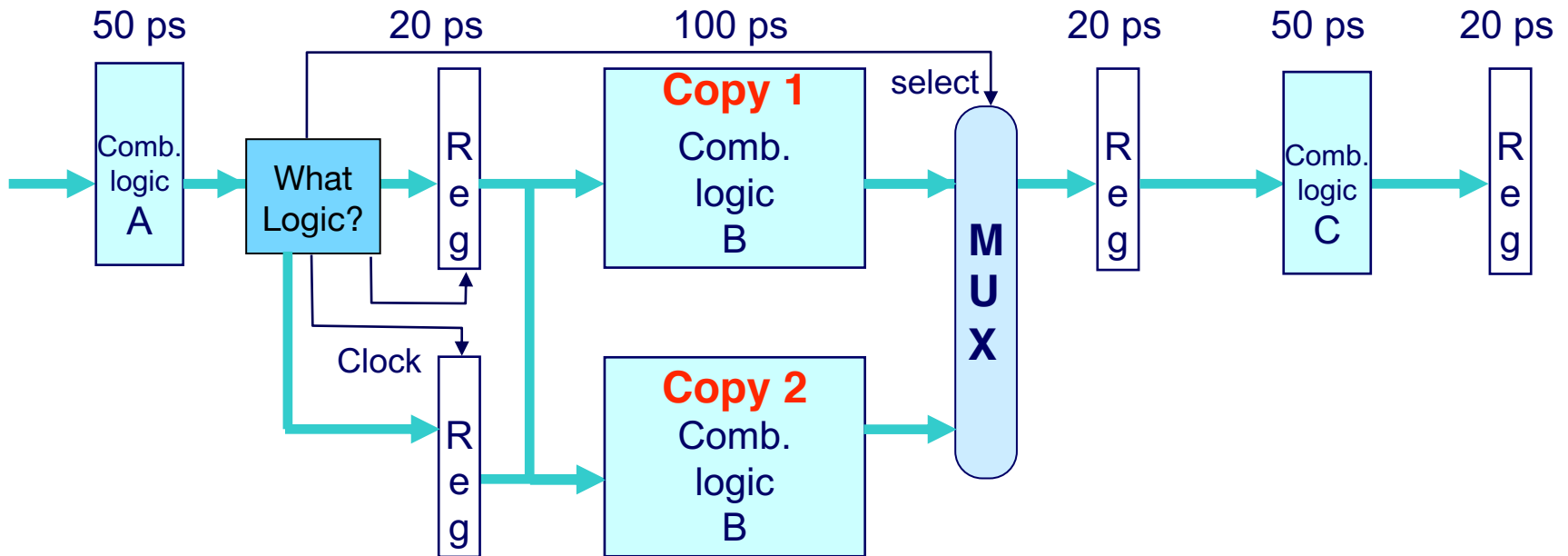


Mitigating Unbalanced Pipeline

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?

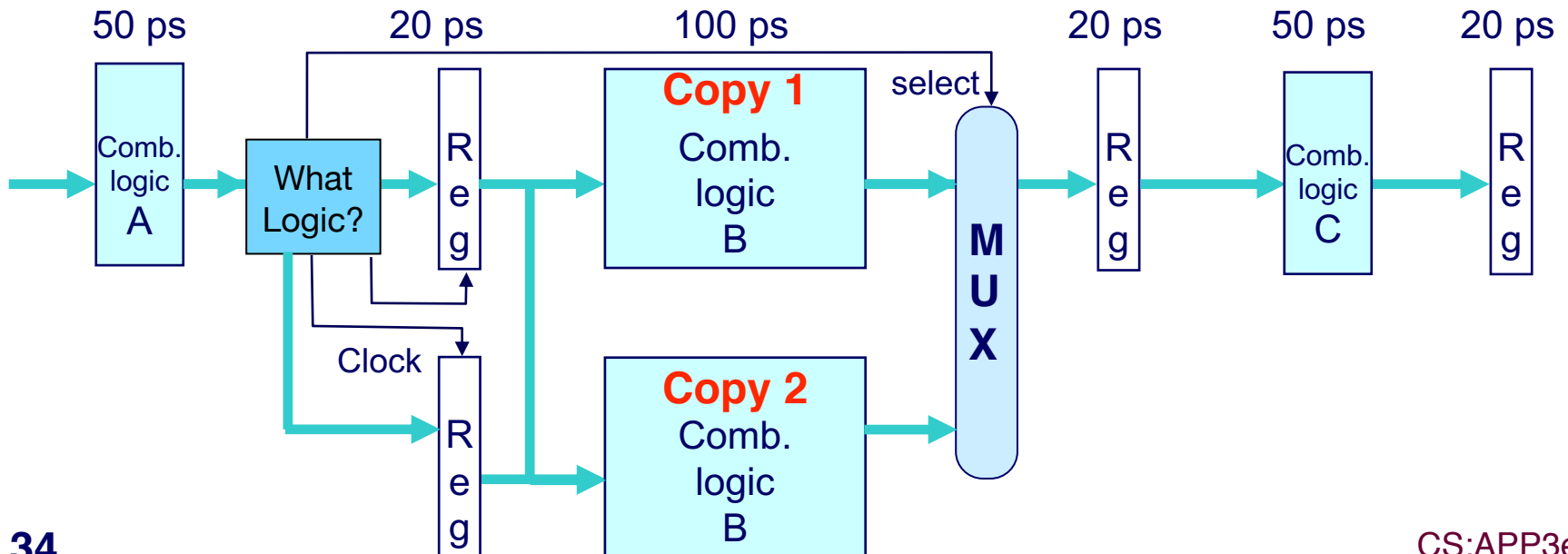
Hint: it needs to control the clock signals of the two registers and the select signal of the MUX.

Mitigating Unbalanced Pipeline

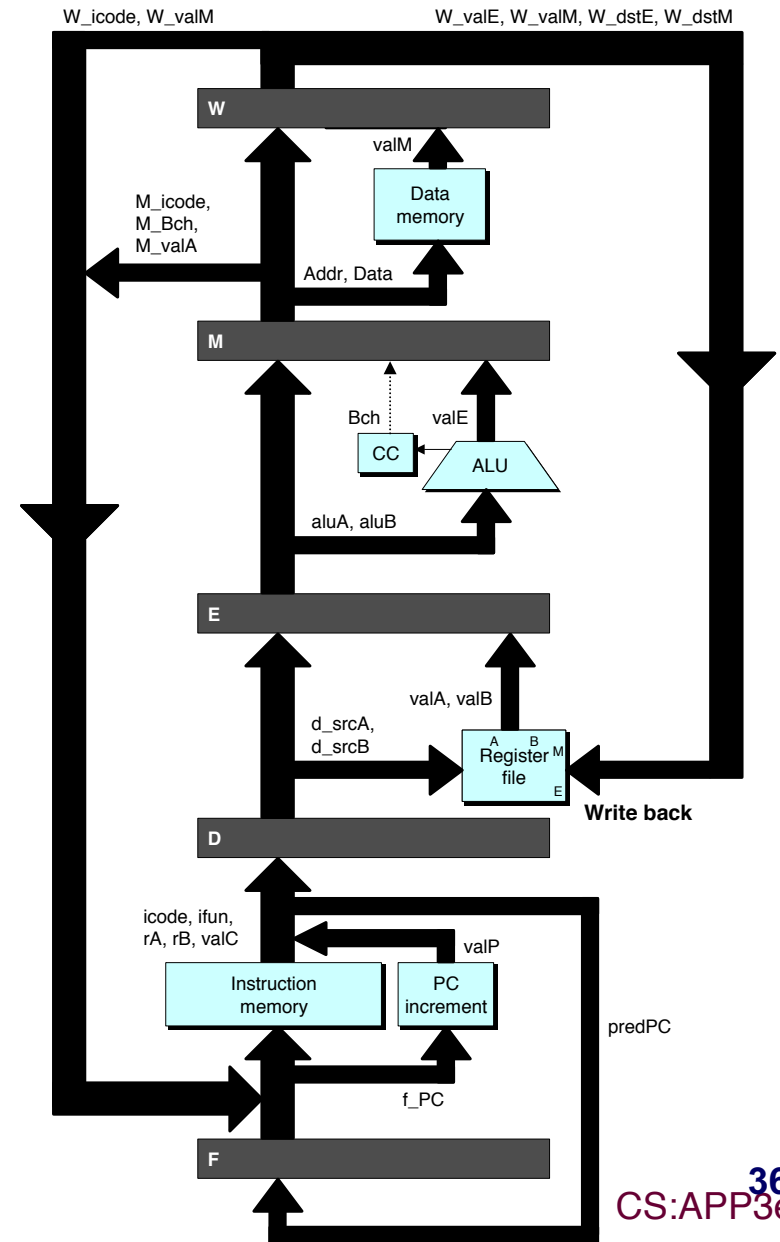
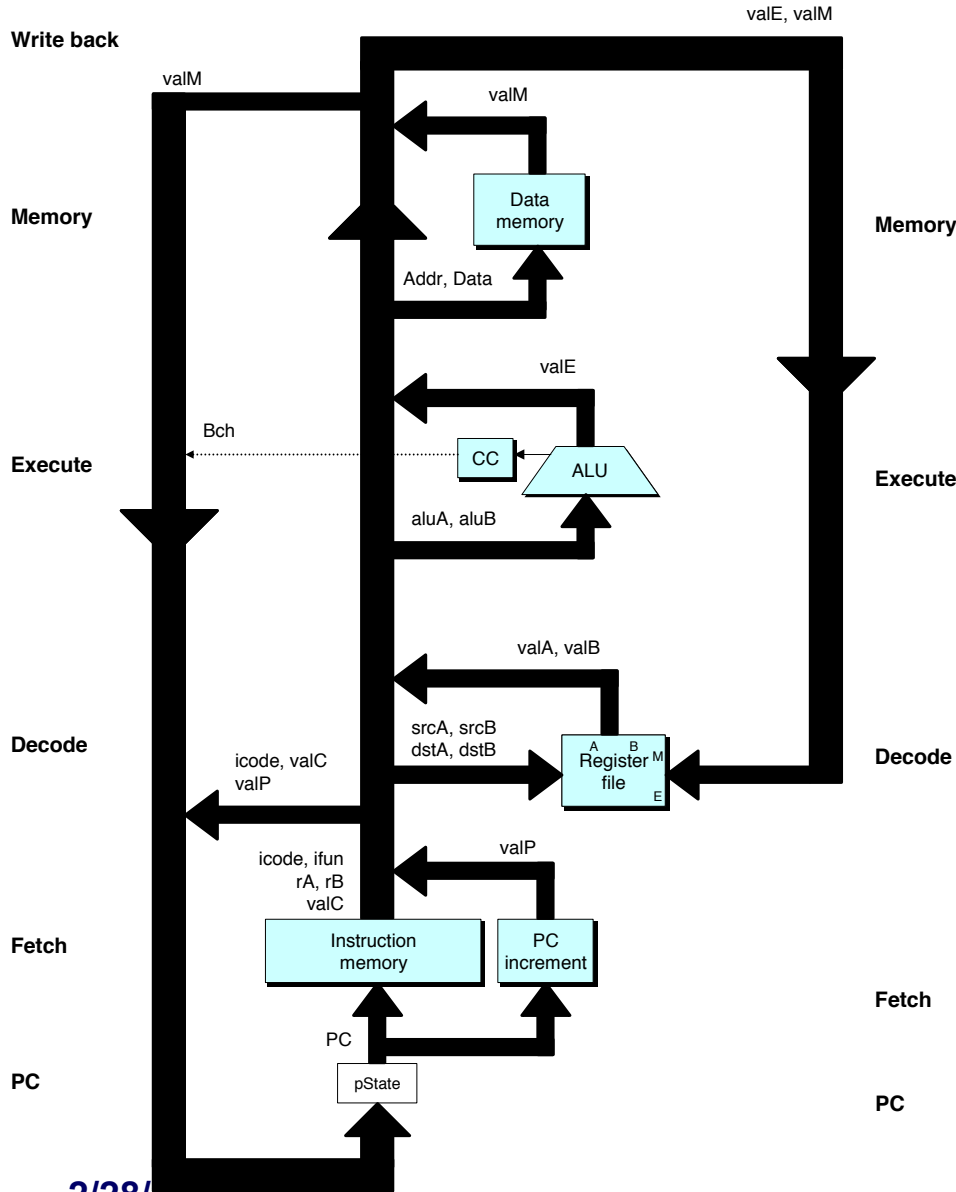
Data sent to copy 1 in odd cycles and to copy 2 in even cycles.

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.



Adding Pipeline Registers



Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

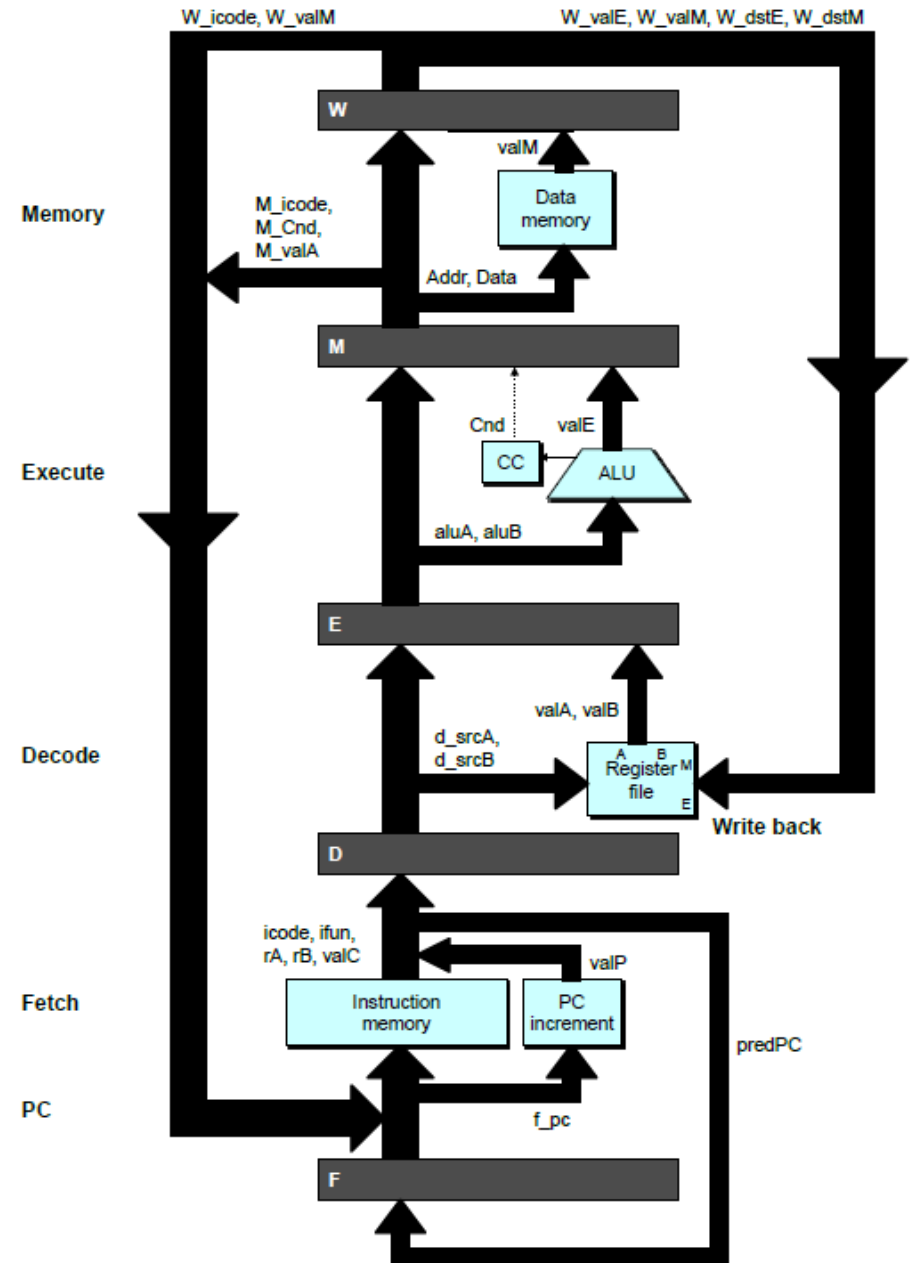
- Operate ALU

Memory

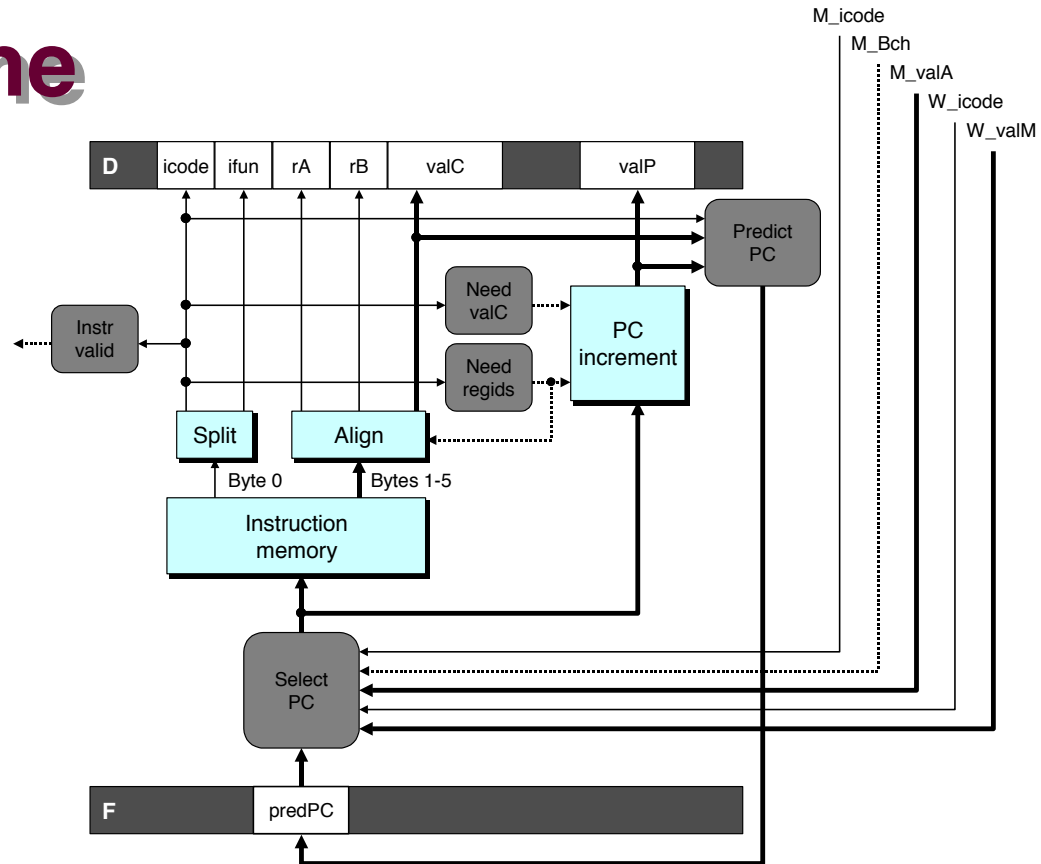
- Read or write data memory

Write Back

- Update register file



Predicting the PC



- Start fetch of new instruction after current one has completed fetch stage
 - Not enough time to reliably determine next instruction
- Guess which instruction will follow
 - Recover if prediction was incorrect

One Prediction Strategy

Instructions that Don't Transfer Control

- Predict next PC to be valP
- Always reliable

Call and Unconditional Jumps

- Predict next PC to be valC (destination)
- Always reliable

Conditional Jumps

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

Return Instruction

- Don't try to predict

Today: 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

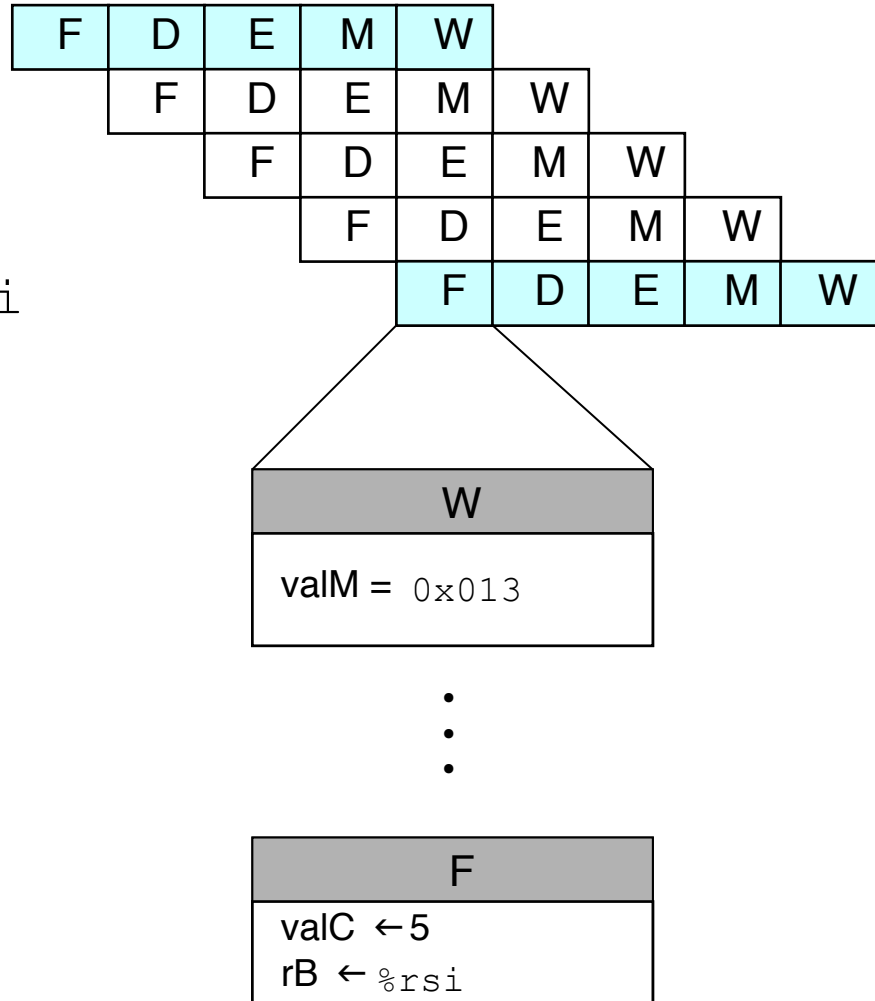
	1	2	3	4	5	6	7	8	9
xorg %rax, %rax	F	D	E	M	W				
jne L1		F	D	E	M	W			
irmovq \$1, %rax			F	D	E	M	W		
L1: irmovq \$4, %rcx				F	D	E	M	W	
irmovq \$3, %rax					F	D	E	M	W

Control Dependency: Return Example

```
0x000:      irmovq Stack,%rsp      # Intialize stack pointer
0x00a:      call p                  # Procedure call
0x013:      irmovq $5,%rsi         # Return point
0x01d:      halt
0x020: p:   irmovq $-1,%rdi        # procedure
0x02a:      ret
0x02b:      irmovq $1,%rax         # Should not be executed
0x035:      irmovq $2,%rcx         # Should not be executed
0x03f:      irmovq $3,%rdx         # Should not be executed
0x049:      irmovq $4,%rbx         # Should not be executed
0x100: Stack:                     # Stack: Stack pointer
```

Control Dependency: Correct Return

0x026: `ret`
 nop
 nop
 nop
0x013: `irmovq $5,%rsi`



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?

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`!!!

```

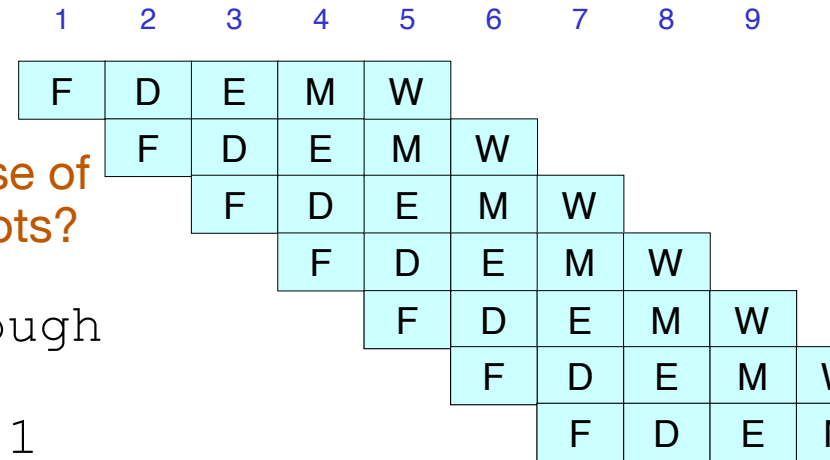
if (cond) {
    do_A();
} else {
    do_B();
}
do_C();
  
```

Delay Slots

```

xorg %rax, %rax
jne L1
nop
nop
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();
}
  
```

```

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

add A, B
sub E, F
jle 0x200
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

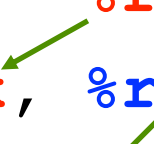
Hardware mechanisms

- Stalling
- Branch Prediction
- Return Address Stack

We will discuss them more later

Data Dependencies

```
1    irmovq $50, %rax
2    addq   %rax, %rbx
3    mrmovq 100(%rbx), %rdx
```



Result from one instruction used as operand for another

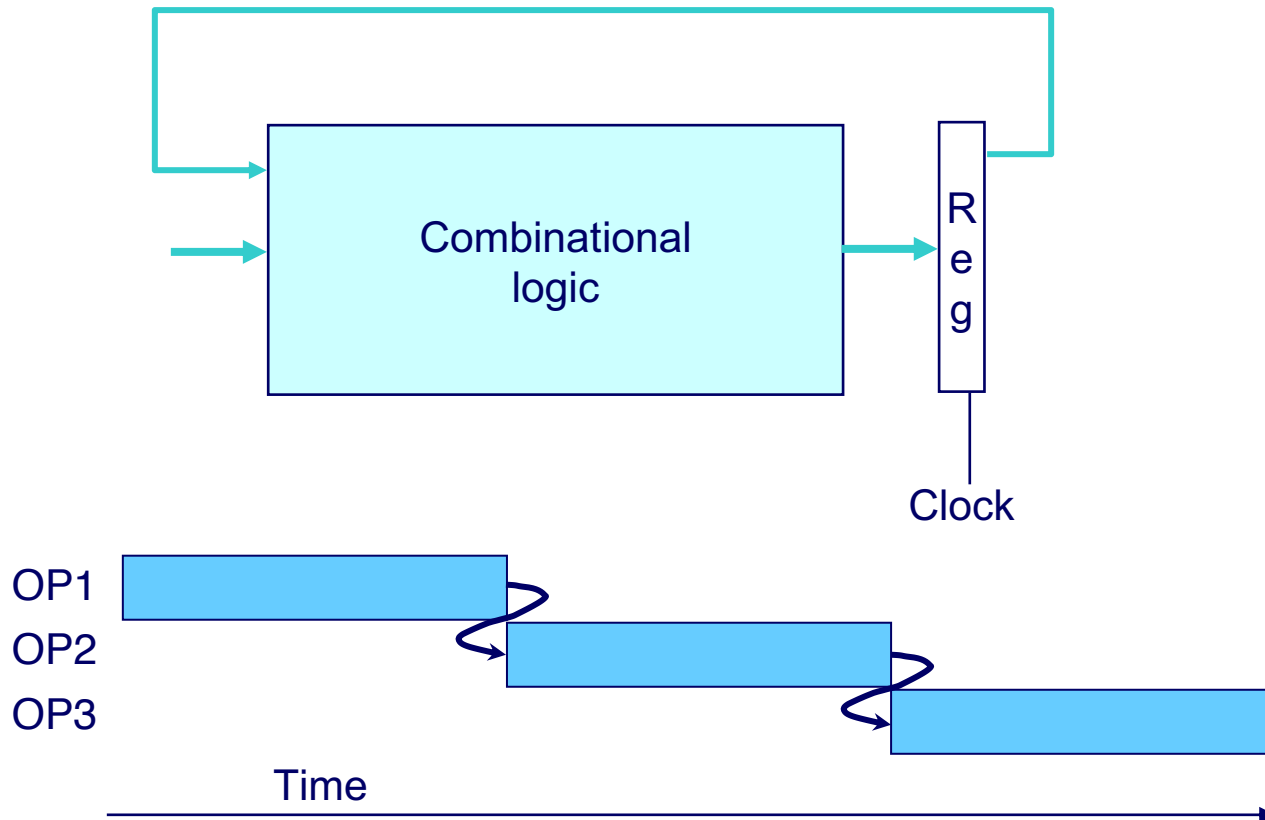
- **Read-after-write (RAW) dependency**

Very common in actual programs

Must make sure our pipeline handles these properly

- **Get correct results**
- **Minimize performance impact**

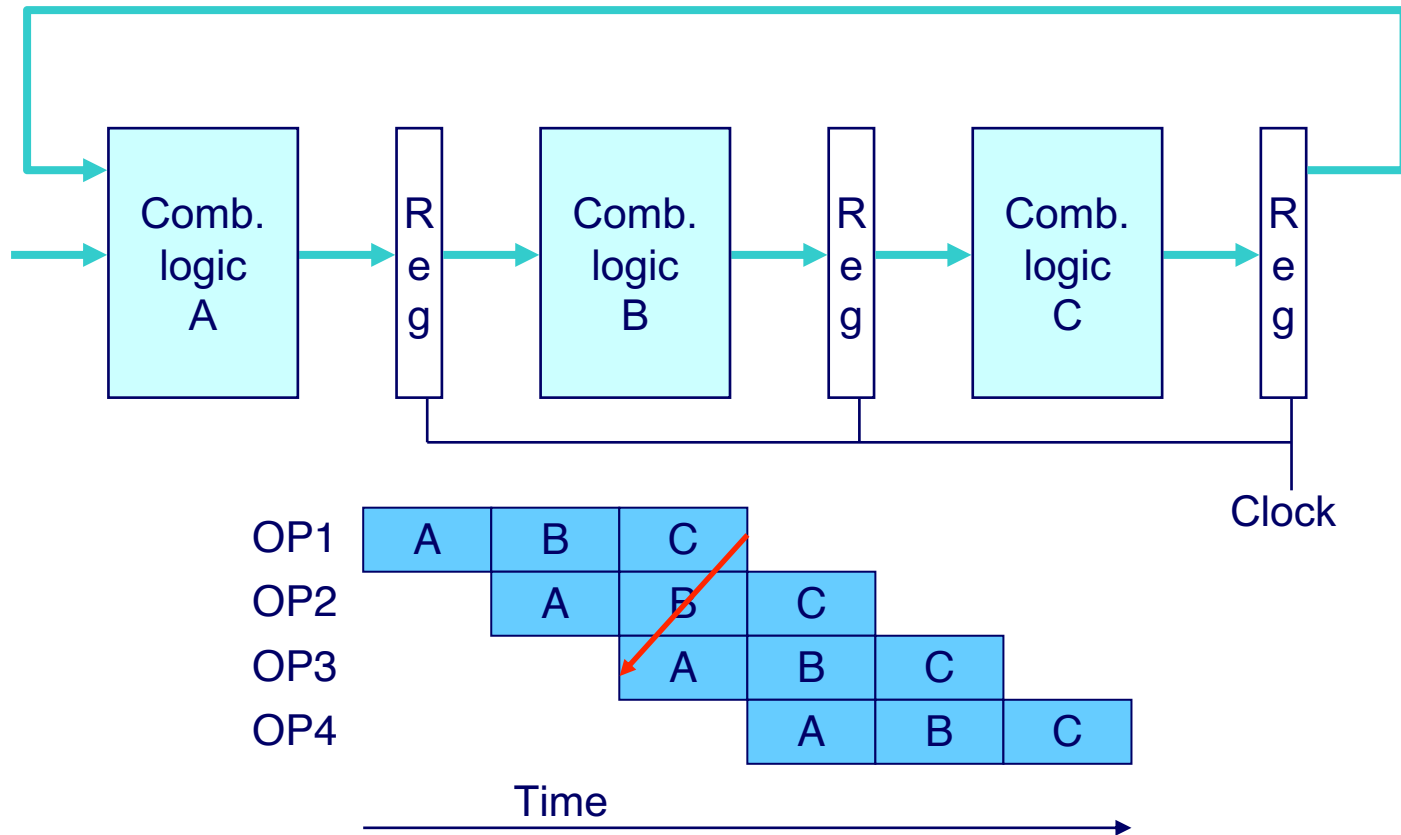
Data Dependencies in Single-Cycle Machines



In Single-Cycle Implementation:

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

Data Dependencies in Pipeline Machines



Data Hazards happen when:

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

Data Dependencies: No Nop

0x000: irmovq \$10,%rdx

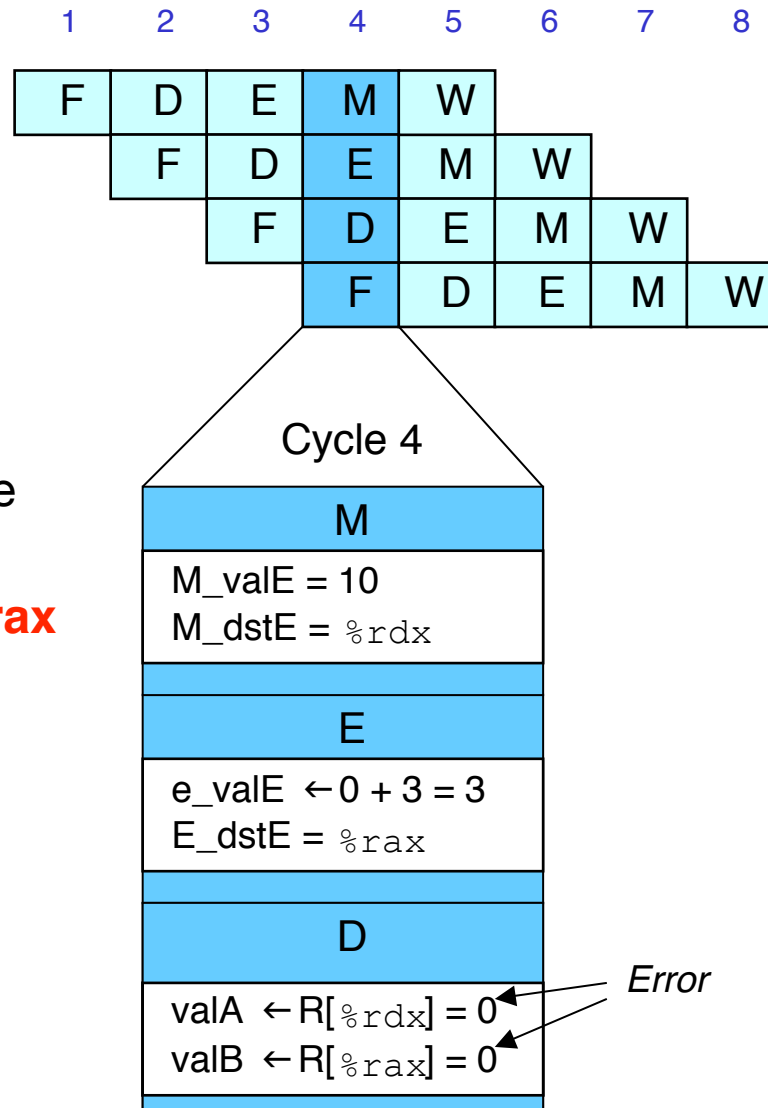
0x00a: irmovq \$3,%rax

0x014: addq %rdx,%rax

0x016: halt

Remember registers get
updated in the Write-back stage

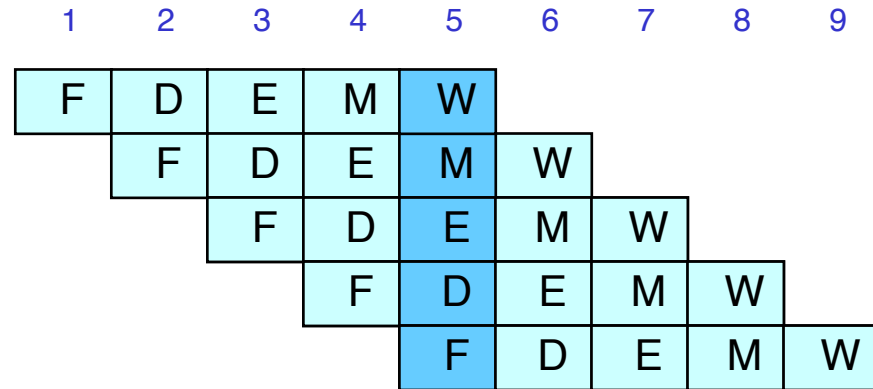
addq reads wrong %rdx and %rax



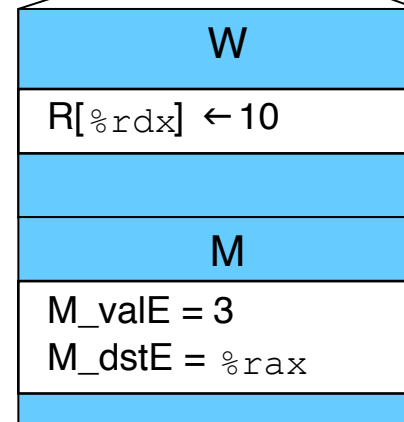
Data Dependencies: 1 Nop

```

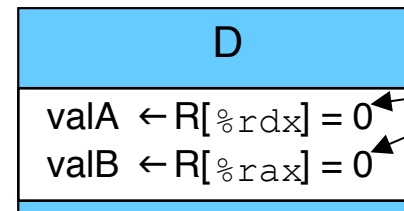
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq  %rdx,%rax
0x017: halt
    
```



addq still reads wrong %rdx and %rax



⋮

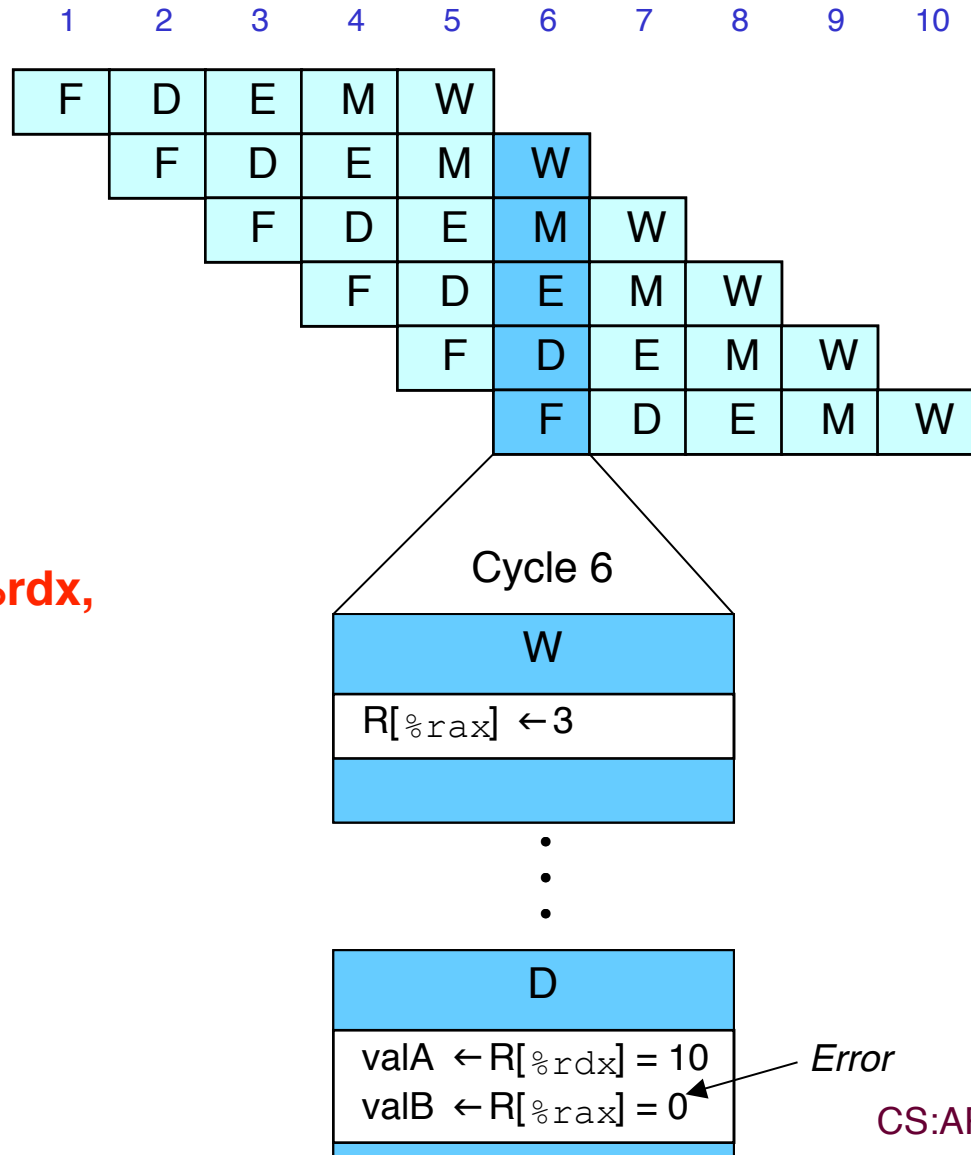


Error

Data Dependencies: 2 Nop's

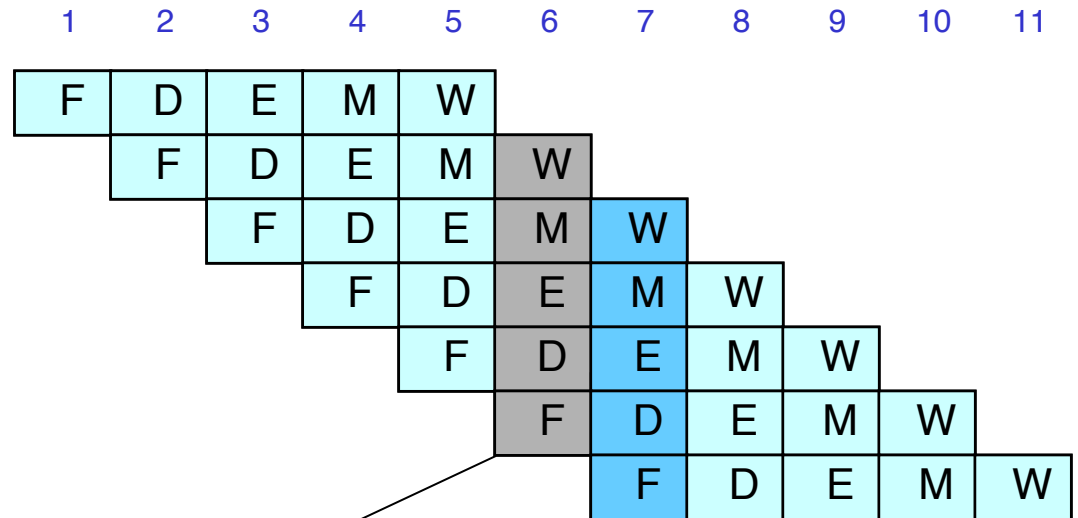
```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

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



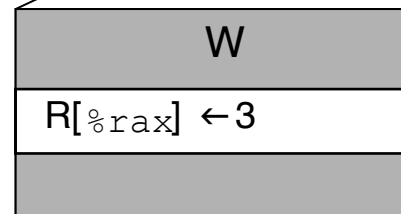
Data Dependencies: 3 Nop's

```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```

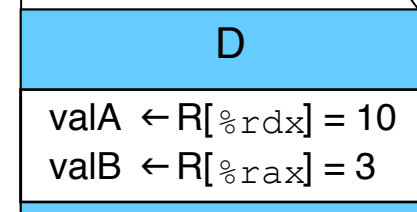


**addq reads the correct %rdx
and %rax**

Cycle 6



Cycle 7



APP3e

Resolving Data Dependencies

Software Mechanisms

- Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea

Hardware mechanisms

- Stalling
- Forwarding
- Out-of-order execution

We will discuss them more later

Branch Prediction

Static Prediction

- Always Taken
- Always Not-taken

Dynamic Prediction

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

If prediction is correct: pipeline moves forward without stalling

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

Static Prediction


Observation: 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
<do_A>
.corner_case:
<do_B>
ret
```

 **Mostly not taken**

```
<before>
.L1:    <body>
        cmpq B, A
        jl  .L1
<after>
```

 **Mostly taken**

Static Prediction

Knowing branch prediction strategy helps us write faster code

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

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

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



Dynamic Prediction

Simplest idea:

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

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

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



Dynamic Prediction

With 1-bit prediction, we change our mind instantly if mispredict

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

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

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