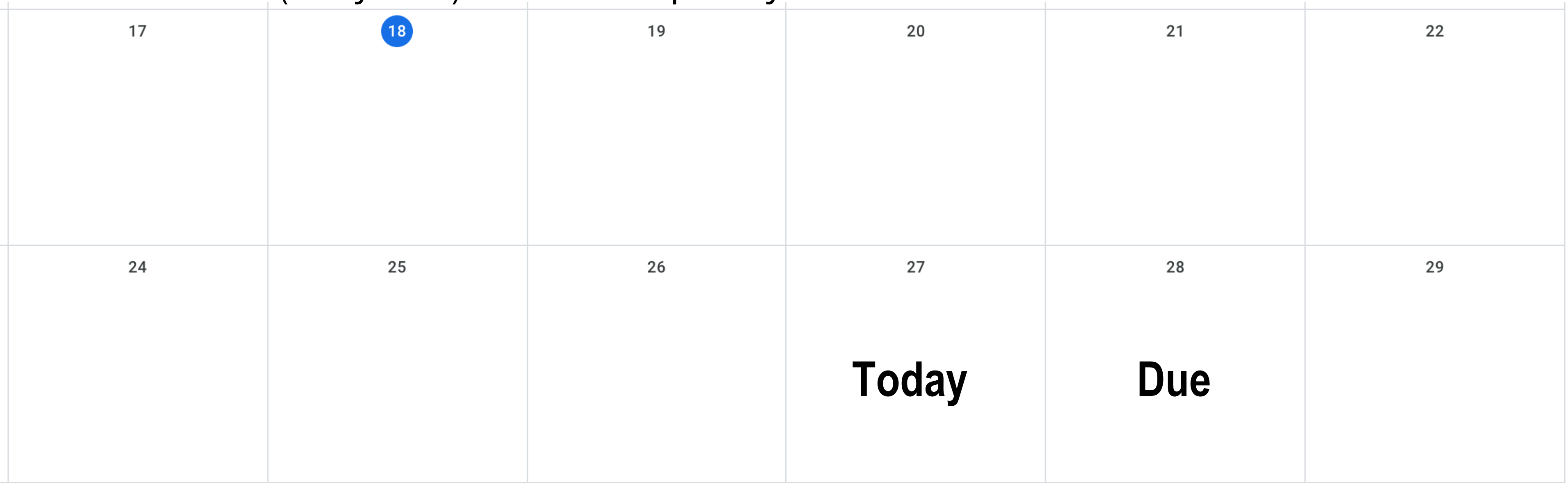
CSC 252: Computer Organization
Spring 2020: Lecture 13

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 **Today**	28 **Due**	29

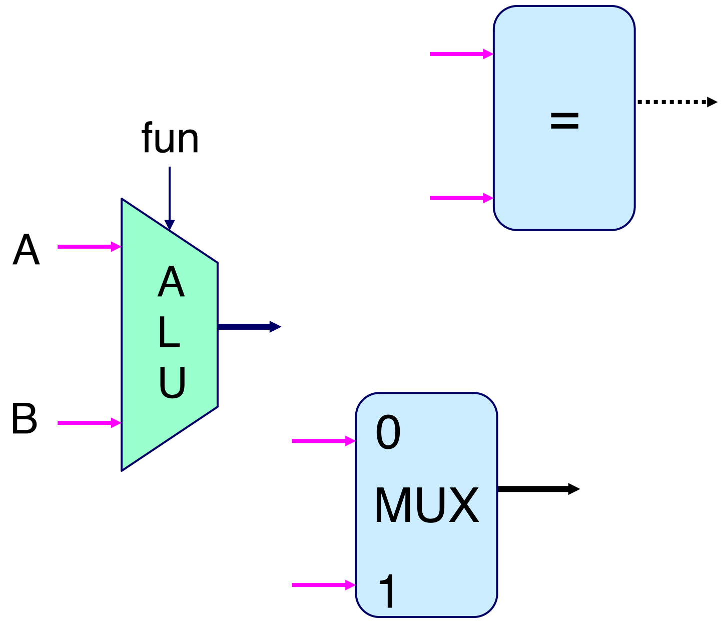
Announcement

- 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

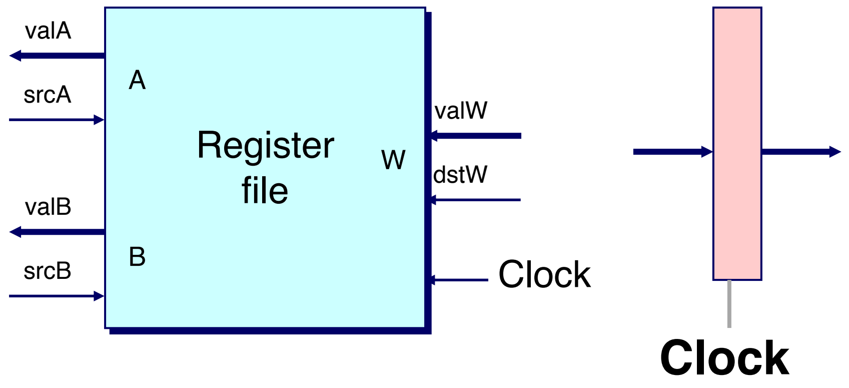
Combinational Logic

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

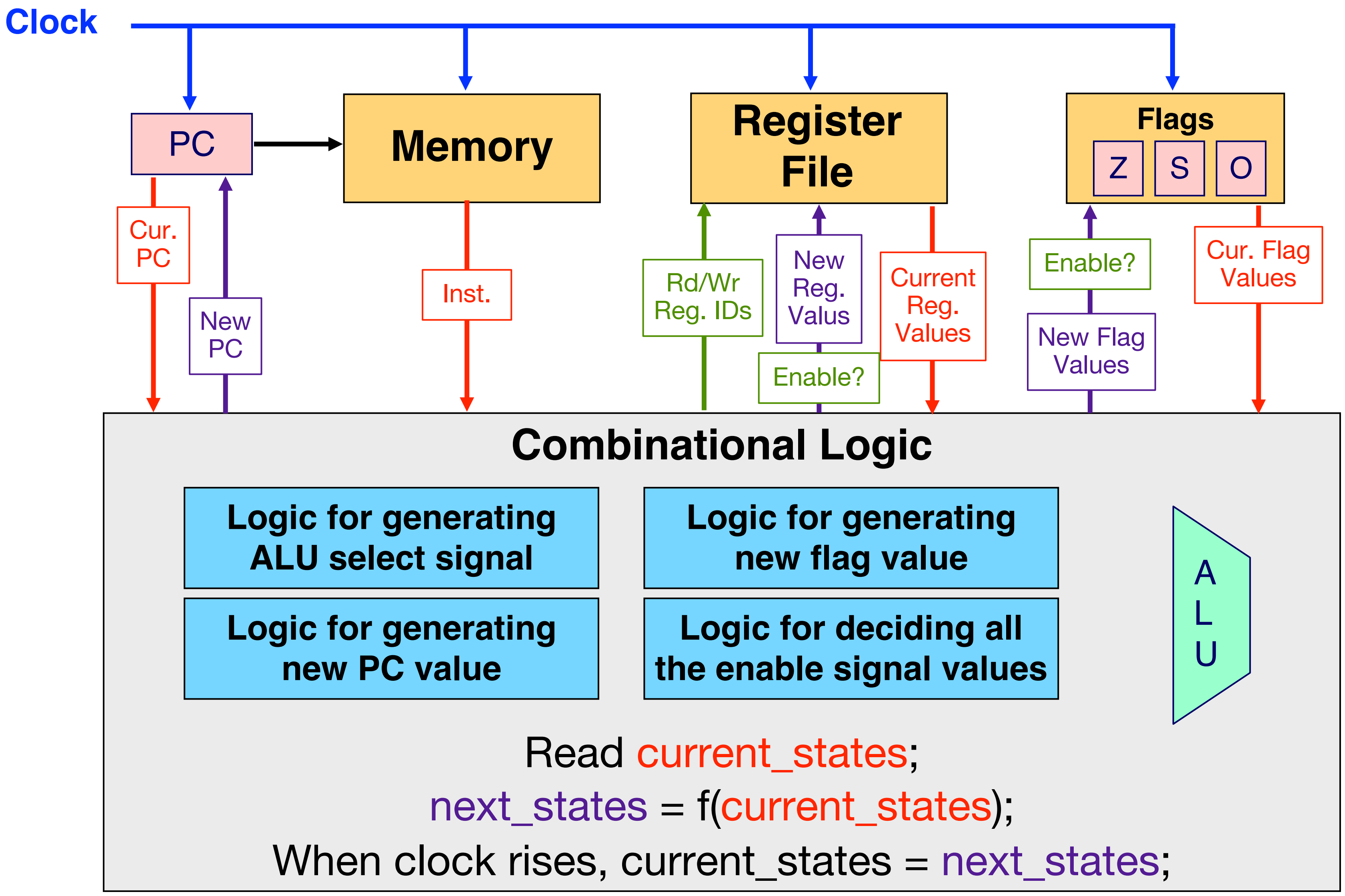


Storage Elements

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



Microarchitecture (So far): Single Cycle



Read current_states;

next_states = f(current_states);

When clock rises, current_states = next_states;

Executing a MOV instruction

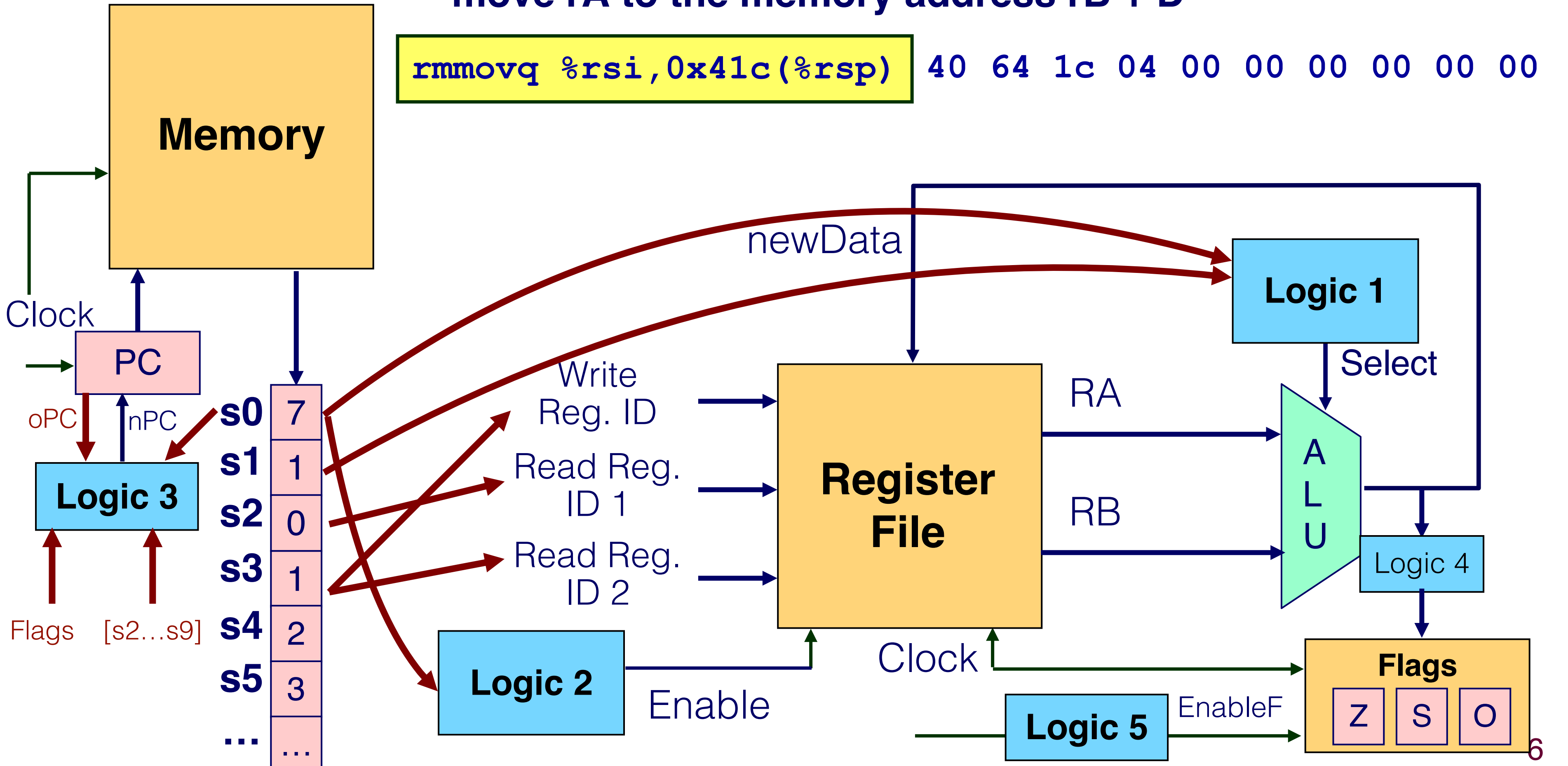
- How do we modify the hardware to execute a move instruction?

`rmmovq rA, D(rB)`

4	0	rA	rB	D

move rA to the memory address rB + D

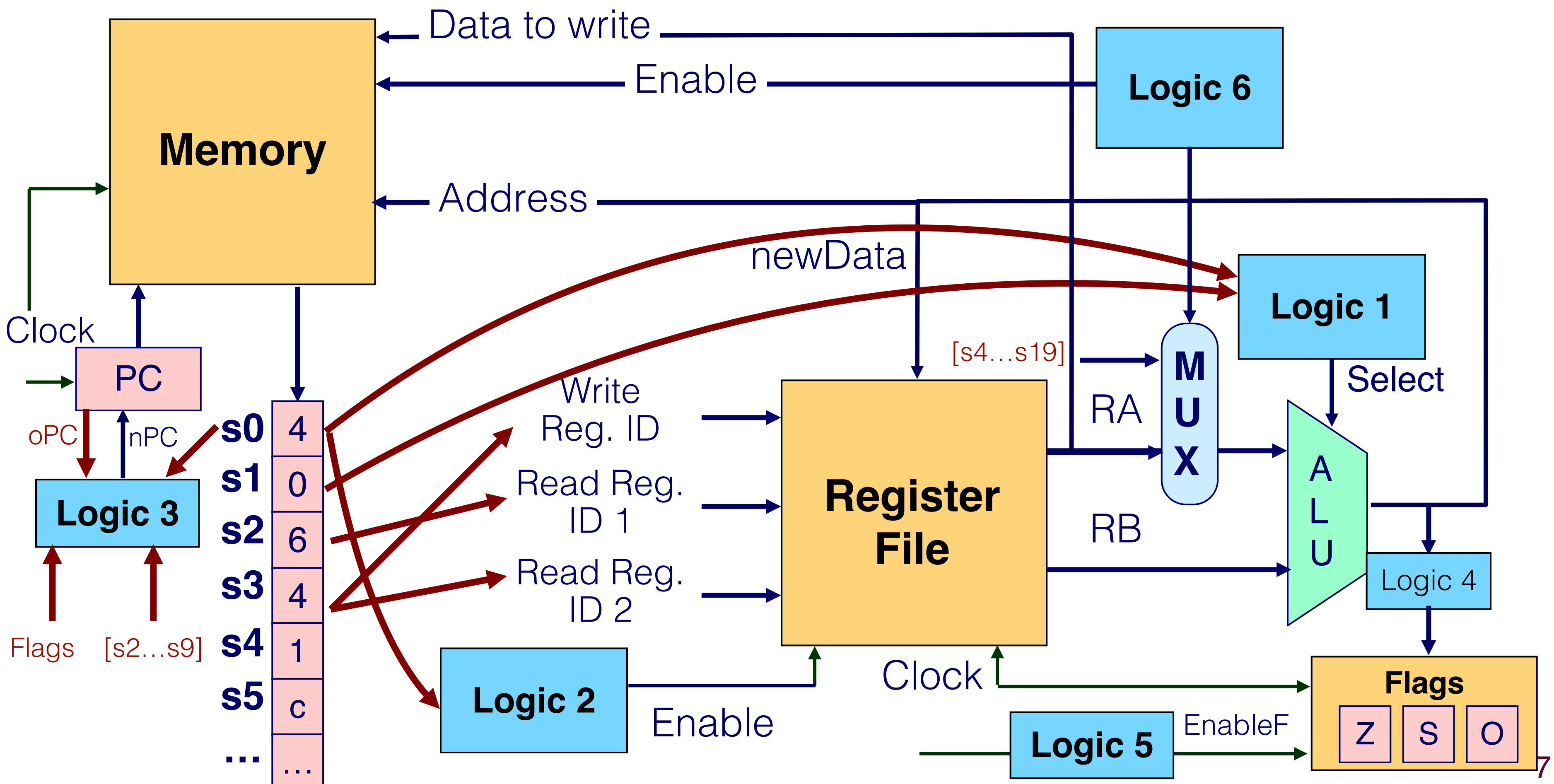
`rmmovq %rsi,0x41c(%rsp)` **40 64 1c 04 00 00 00 00 00 00**



move rA to the memory address rB + D

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

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

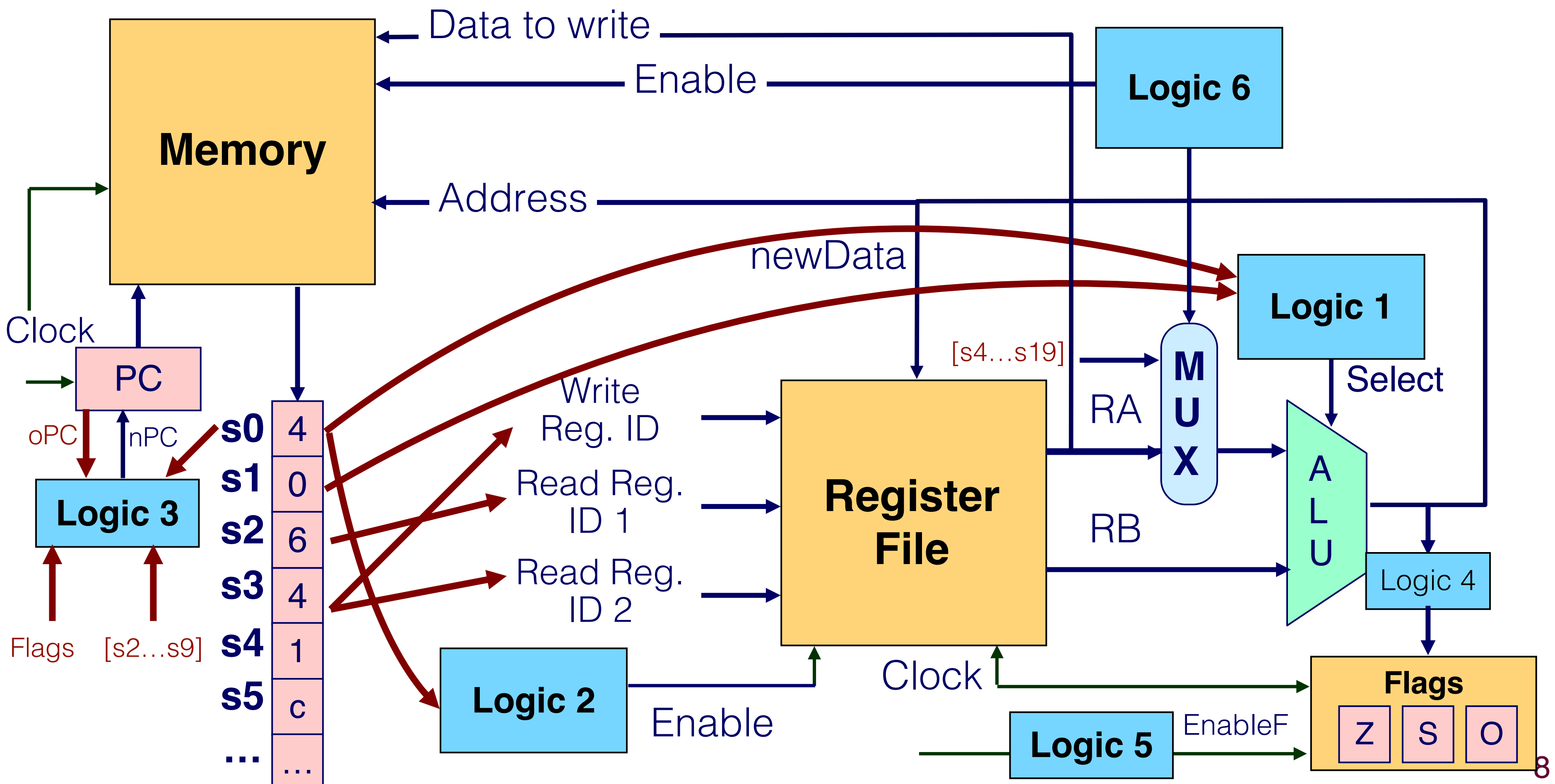


How About Memory to Register MOV?

move data at memory address rB + D to rA

`mrmovq D(rB), rA`

4	0	rA	rB	D

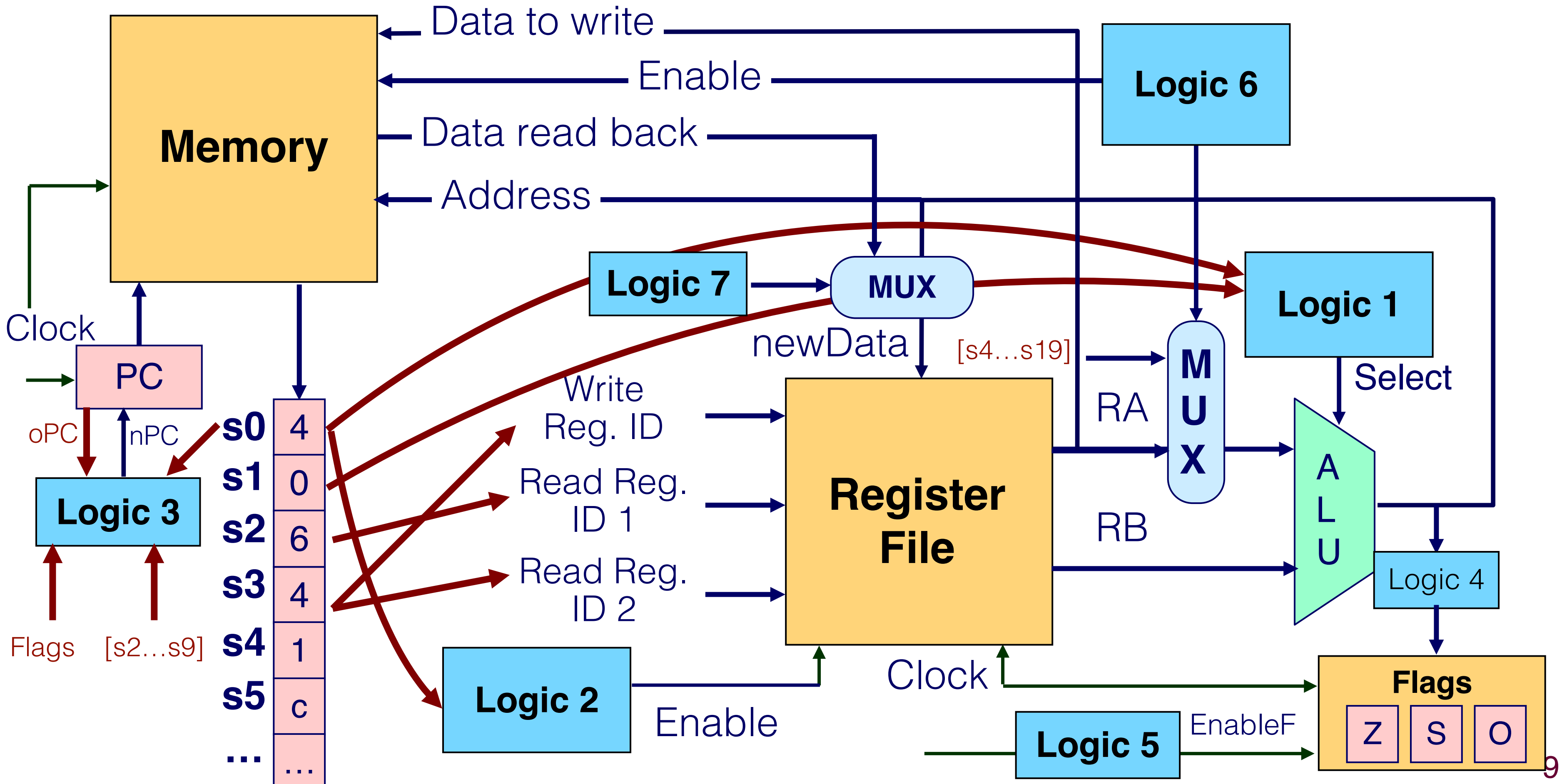


How About Memory to Register MOV?

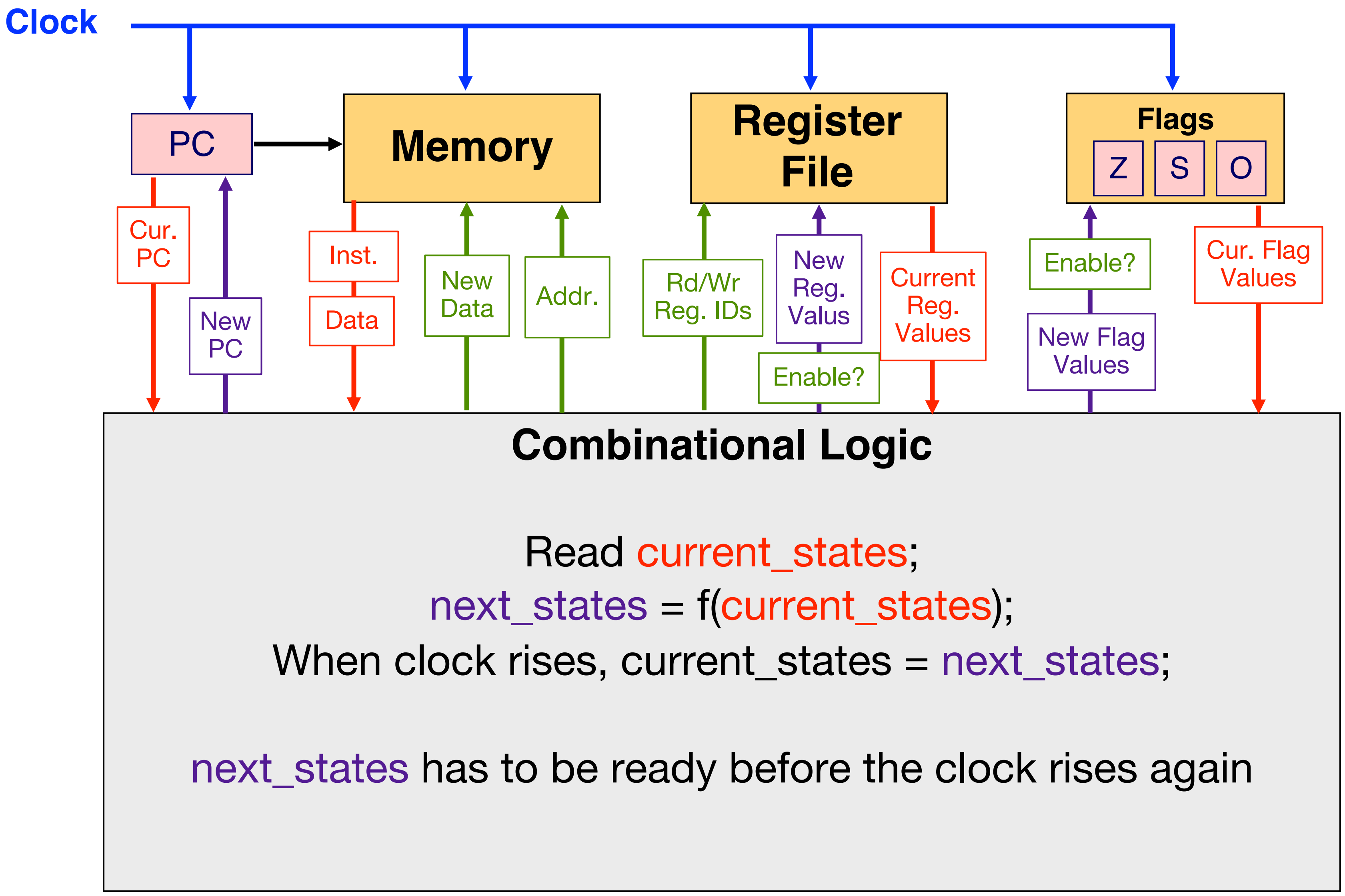
move data at memory address rB + D to rA

`mrmovq D(rB), rA`

4	0	rA	rB	D



Microarchitecture (with MOV)



Read current_states;

next_states = f(current_states);

When clock rises, current_states = next_states;

next_states has to be ready before the clock rises again

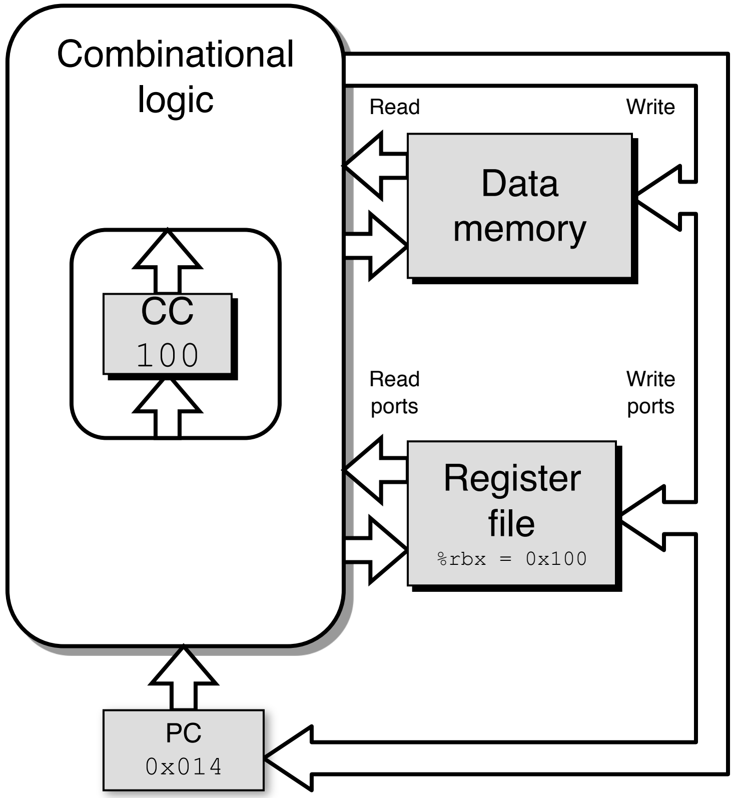
Microarchitecture Overview

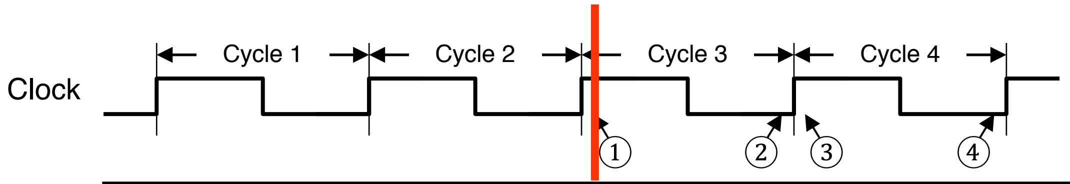
Think of it as a state machine

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

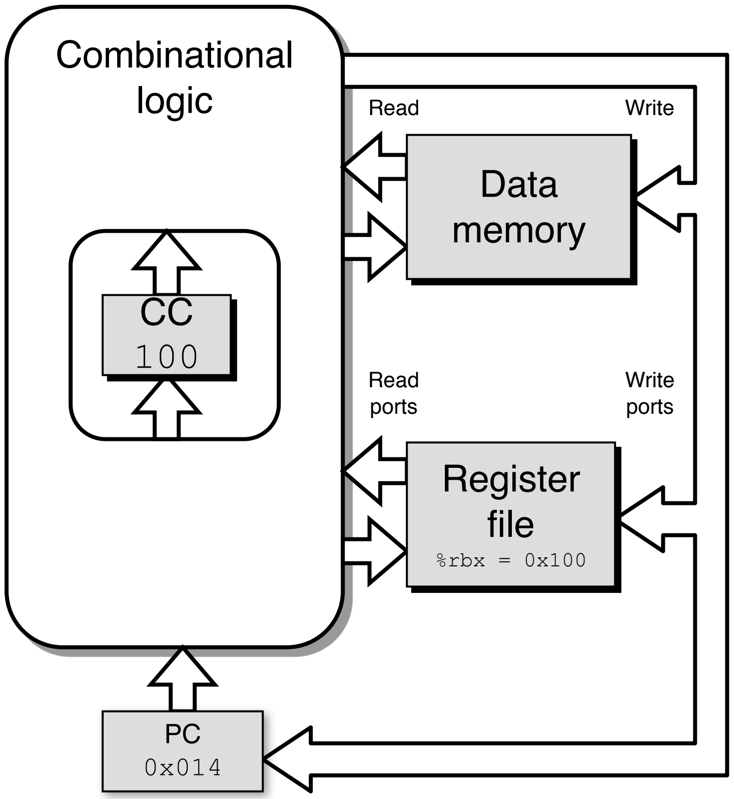
States (All updated as clock rises)

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

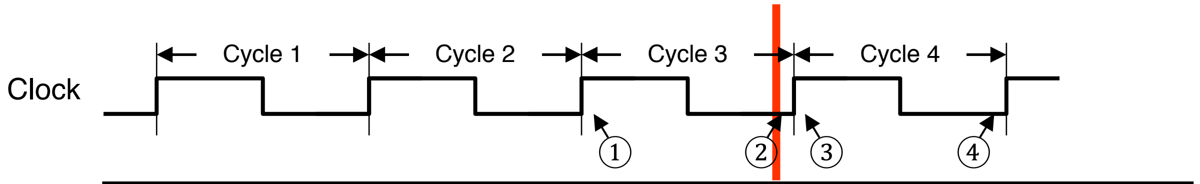




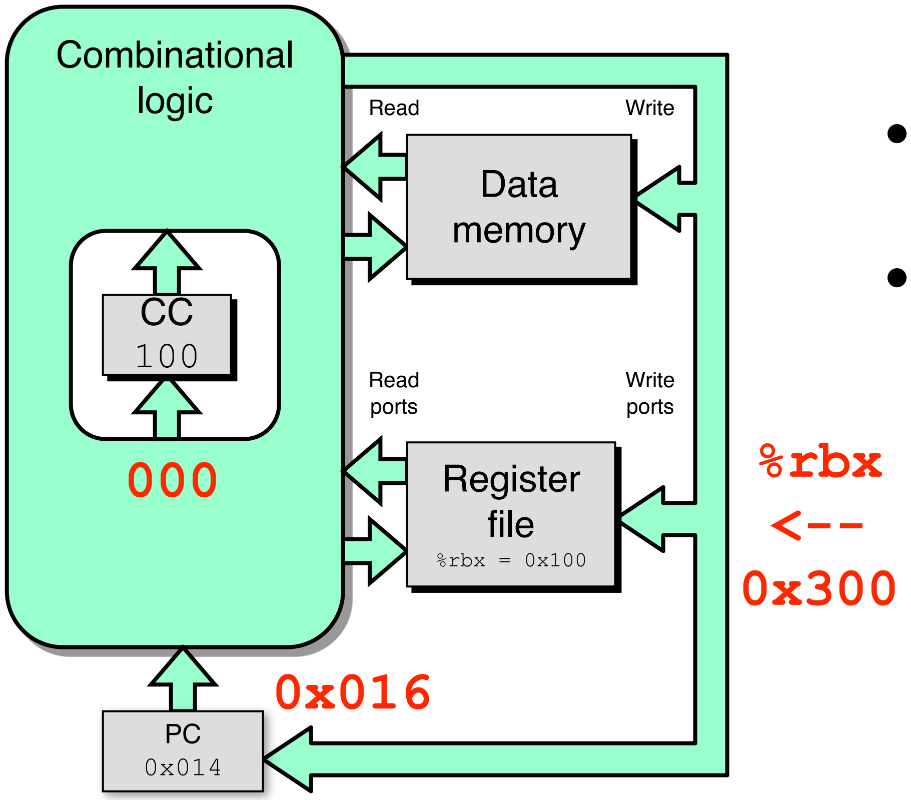
```
Cycle 1:  0x000:   irmovq $0x100,%rbx   # %rbx <-- 0x100
Cycle 2:  0x00a:   irmovq $0x200,%rdx   # %rdx <-- 0x200
Cycle 3:  0x014:   addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000
Cycle 4:  0x016:   je dest              # Not taken
Cycle 5:  0x01f:   rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300
```



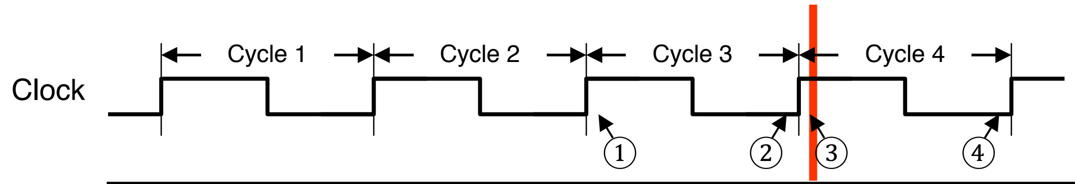
- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes



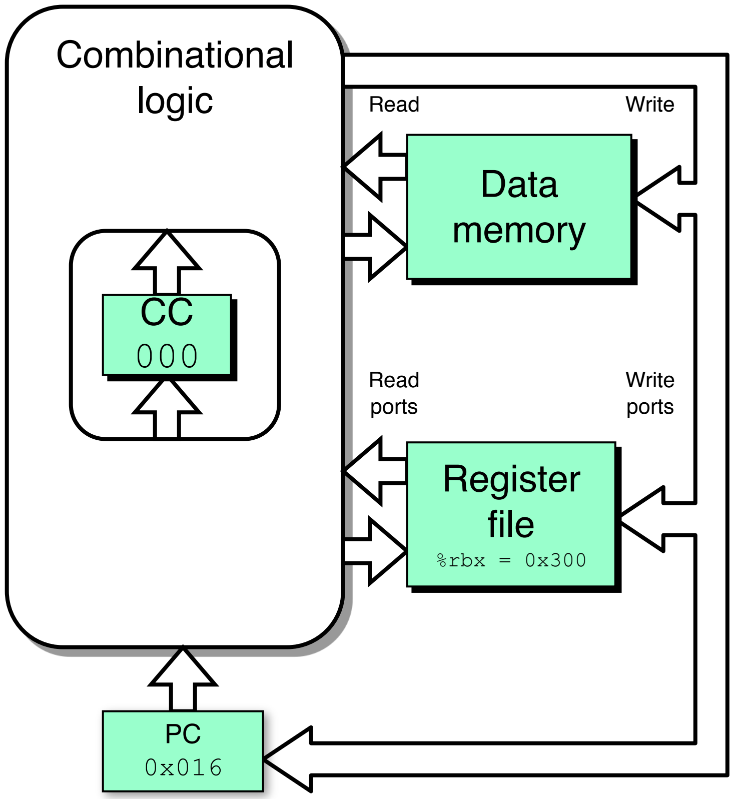
Cycle 1:	`0x000:  irmovq $0x100,%rbx  # %rbx <-- 0x100`
Cycle 2:	`0x00a:  irmovq $0x200,%rdx  # %rdx <-- 0x200`
Cycle 3:	`0x014:  addq %rdx,%rbx      # %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:  je dest             # Not taken`
Cycle 5:	`0x01f:  rmmovq %rbx,0(%rdx) # M[0x200] <-- 0x300`



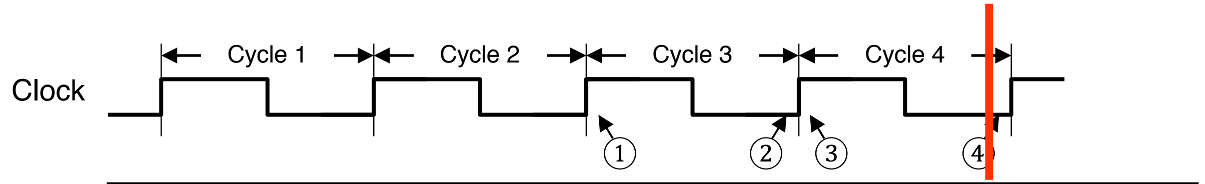
- state set according to second `irmovq` instruction
- combinational logic generates results for addq instruction



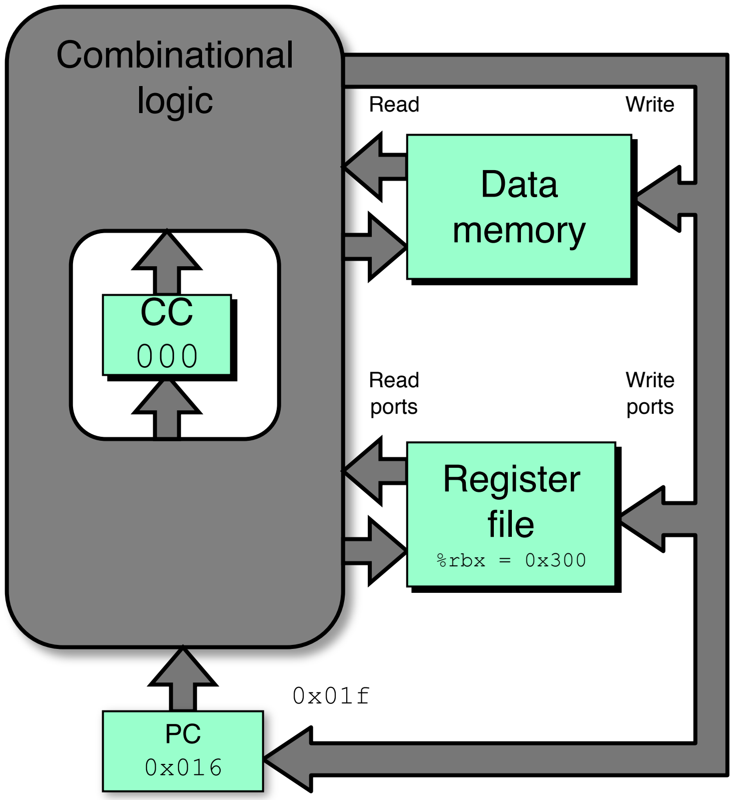
Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



- 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

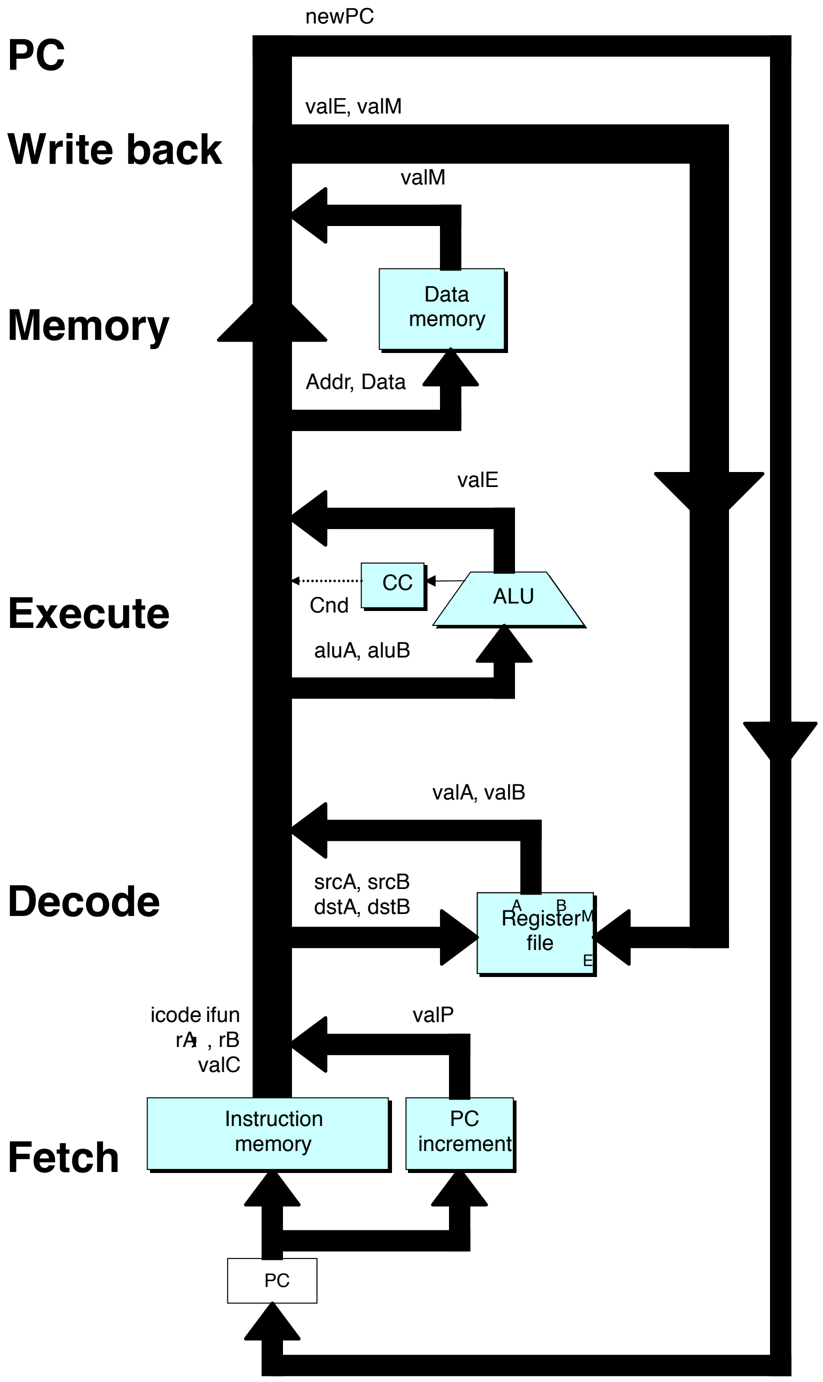
Decode: Read program registers

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`

6	fn	rA	rB

	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)	
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	
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		
Write back		
PC update	PC $\leftarrow$ Cnd ? valC : 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

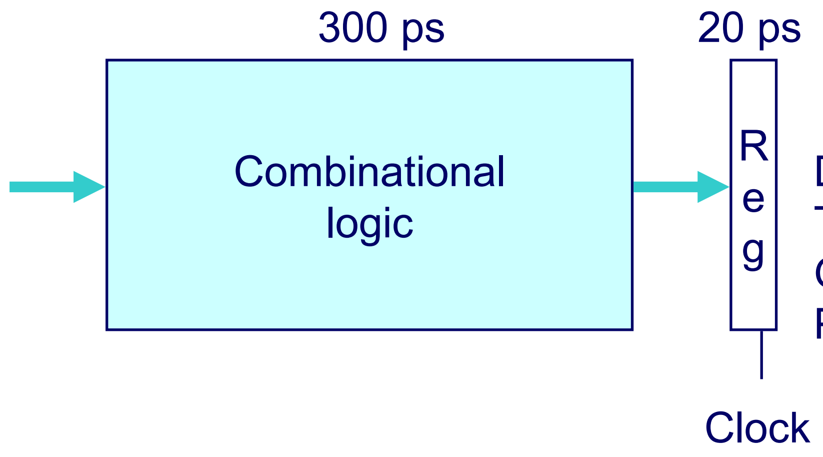
Sequential

Pipelined

Idea

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

Computational Example

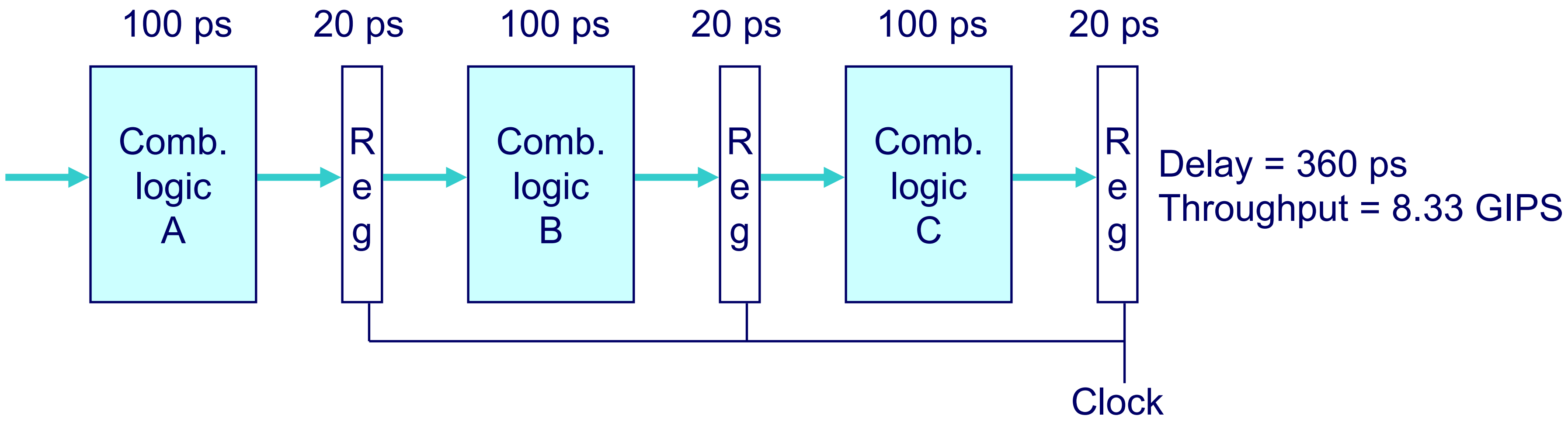


Delay = 320 ps
Throughput = 3.12 Giga Instructions Per second (GIPS)

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

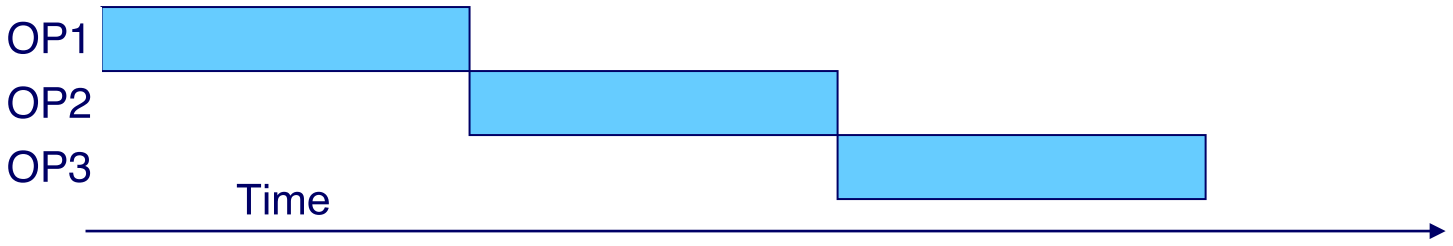


System

- 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

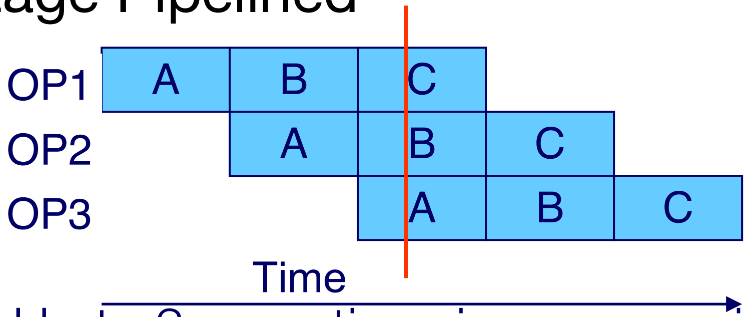
Pipeline Diagrams

Unpipelined



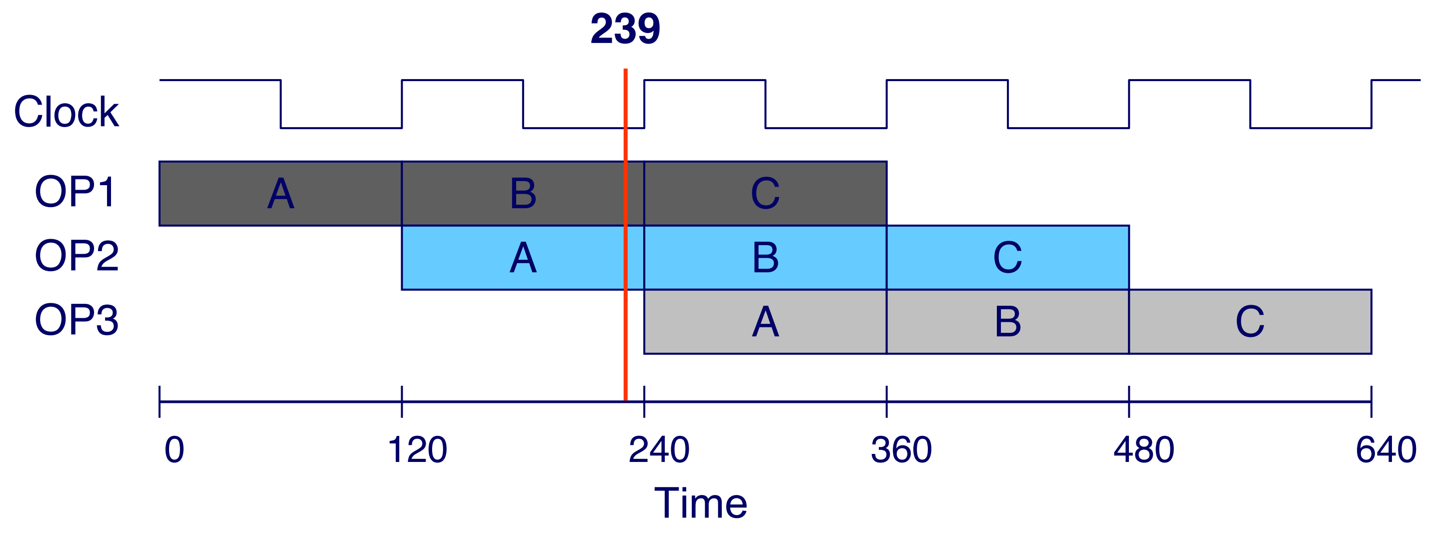
- Cannot start new operation until previous one completes

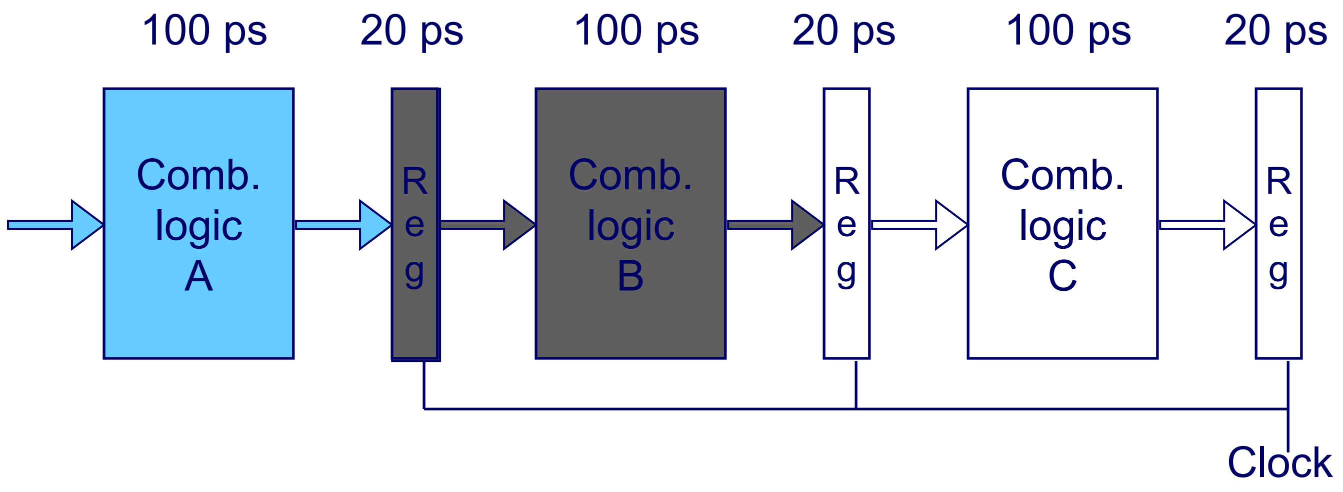
3-Stage Pipelined



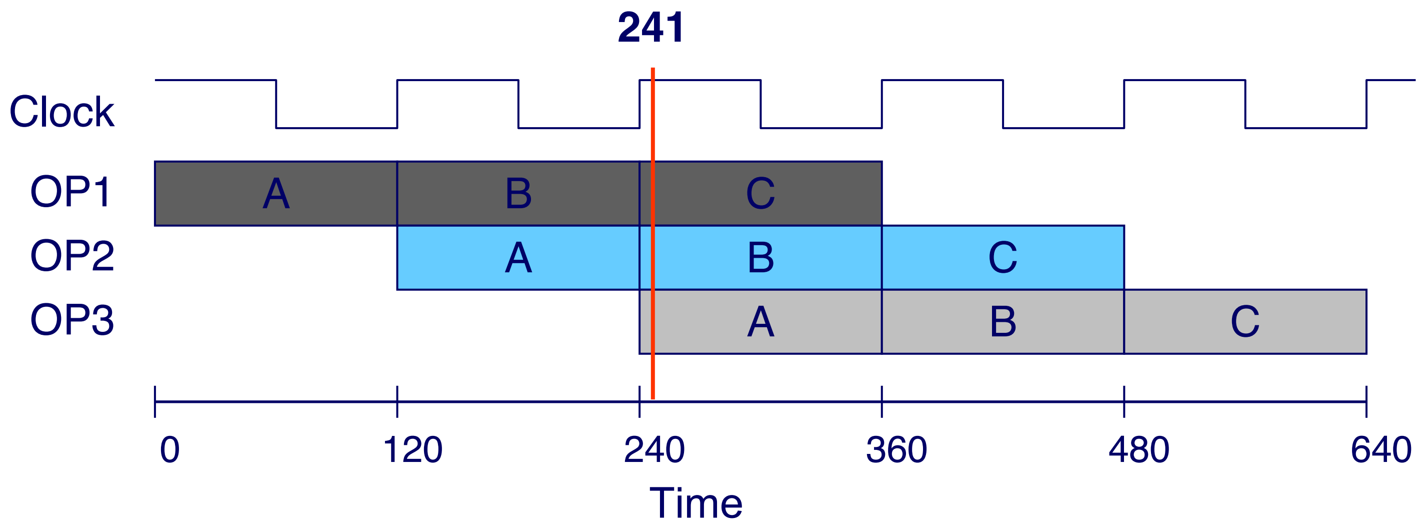
- Up to 3 operations in process simultaneously

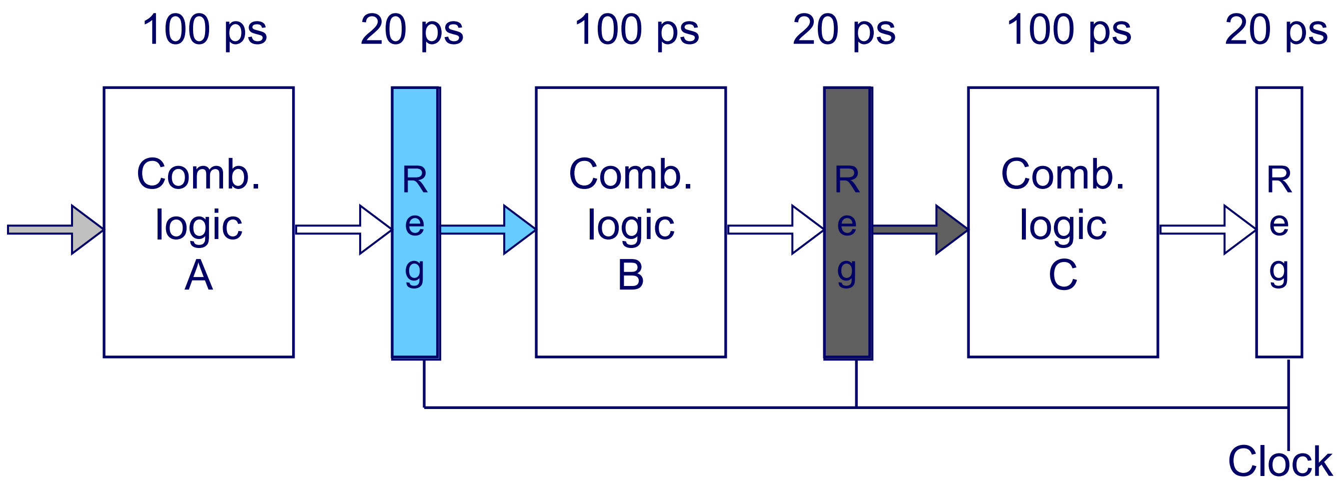
Operating a Pipeline



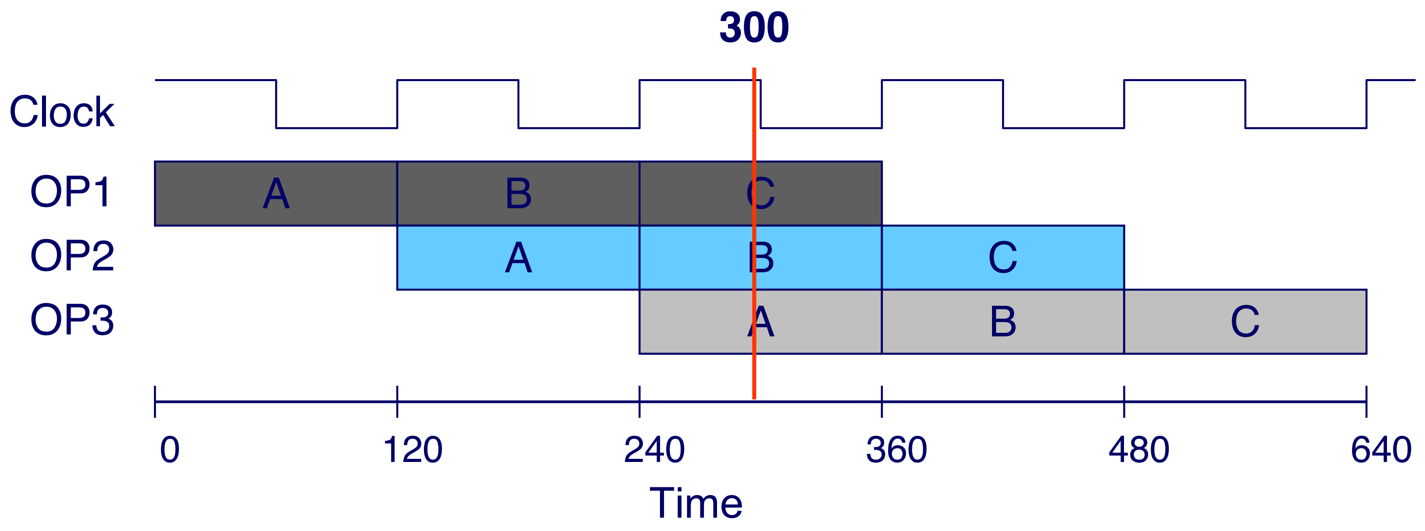


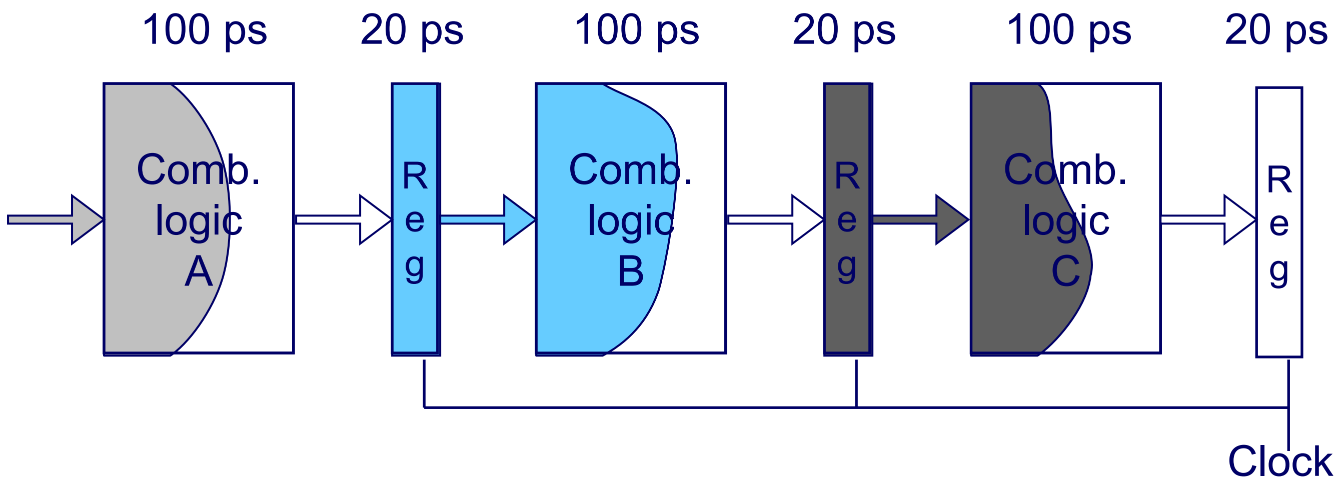
Operating a Pipeline



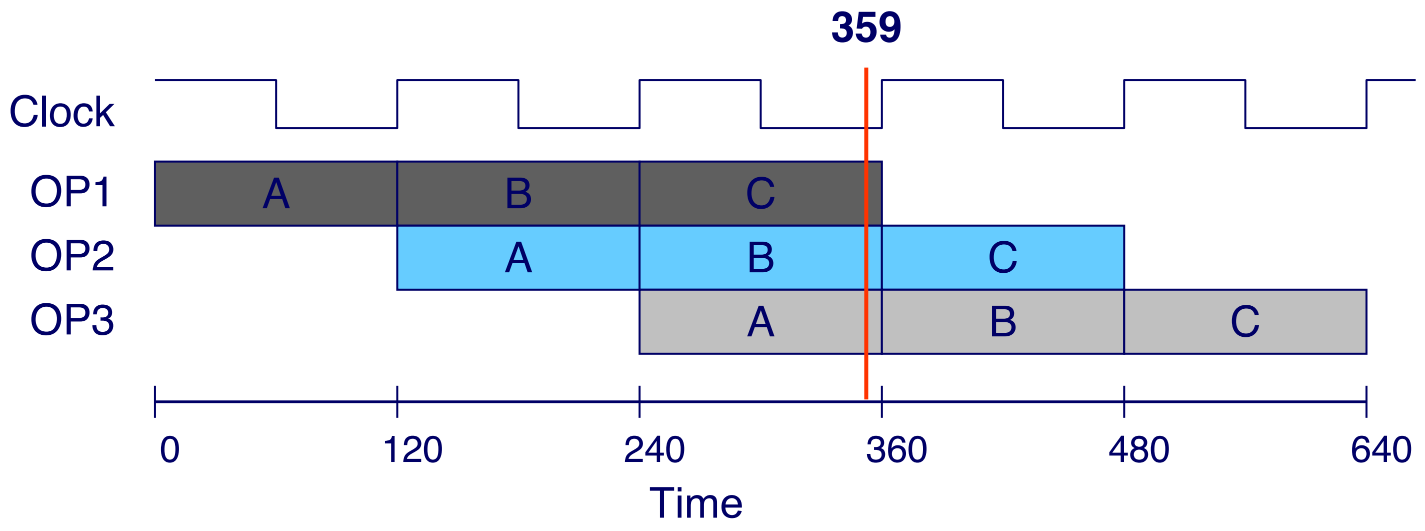


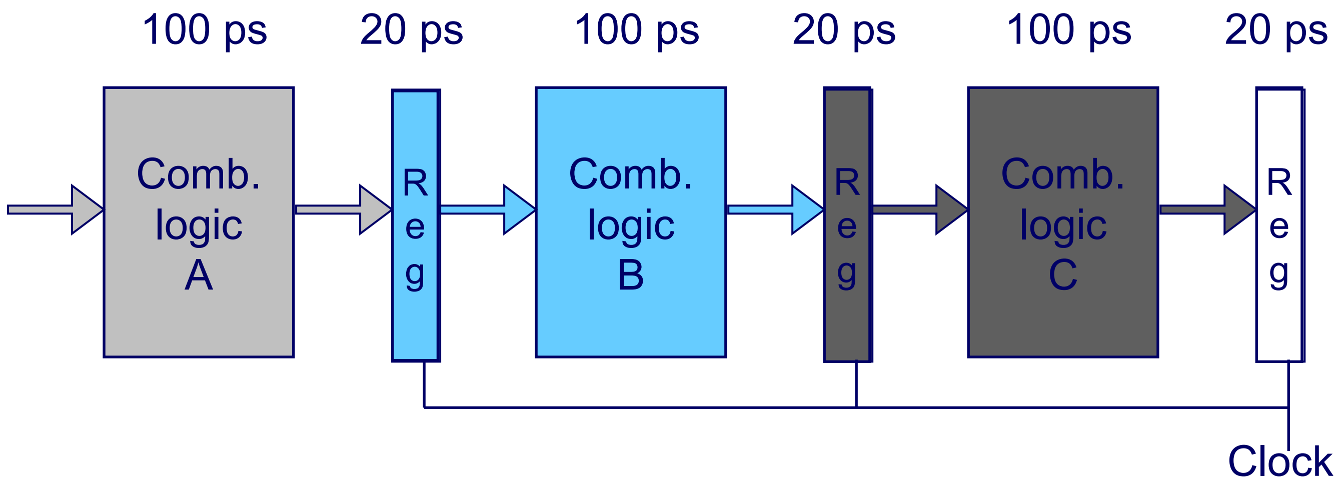
Operating a Pipeline





Operating a Pipeline

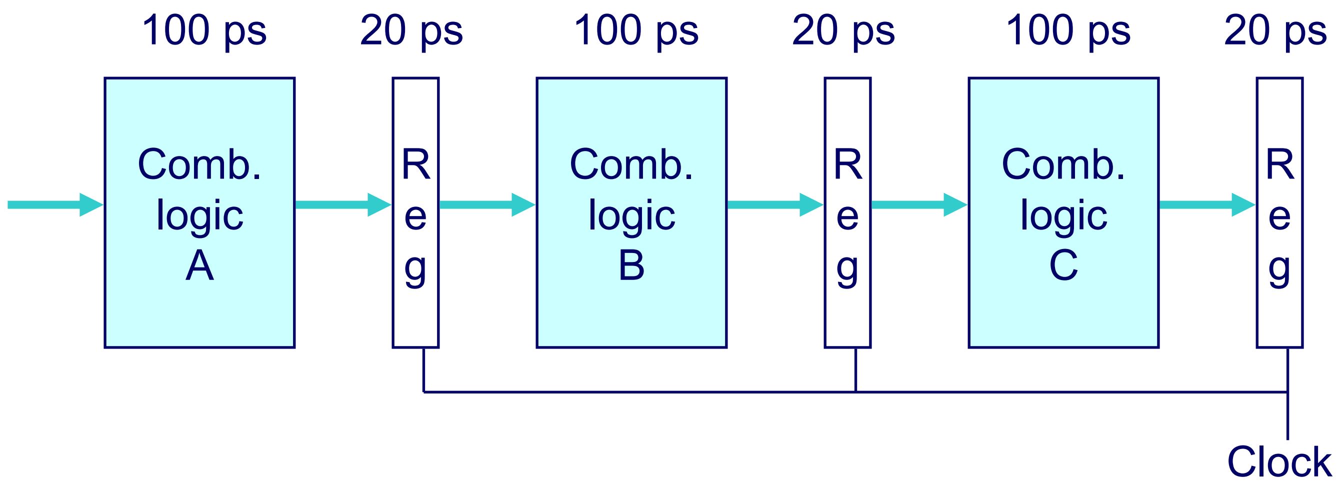


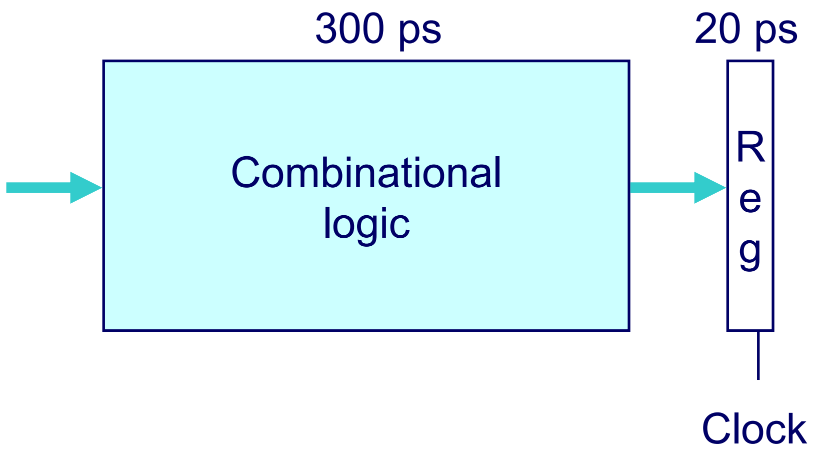


Pipeline Trade-offs

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

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



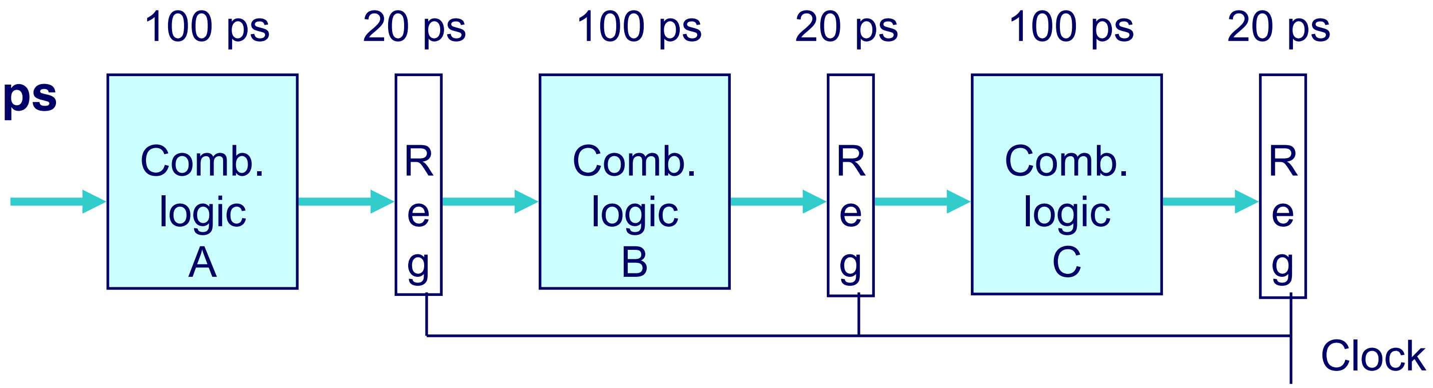


Unbalanced Pipeline

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

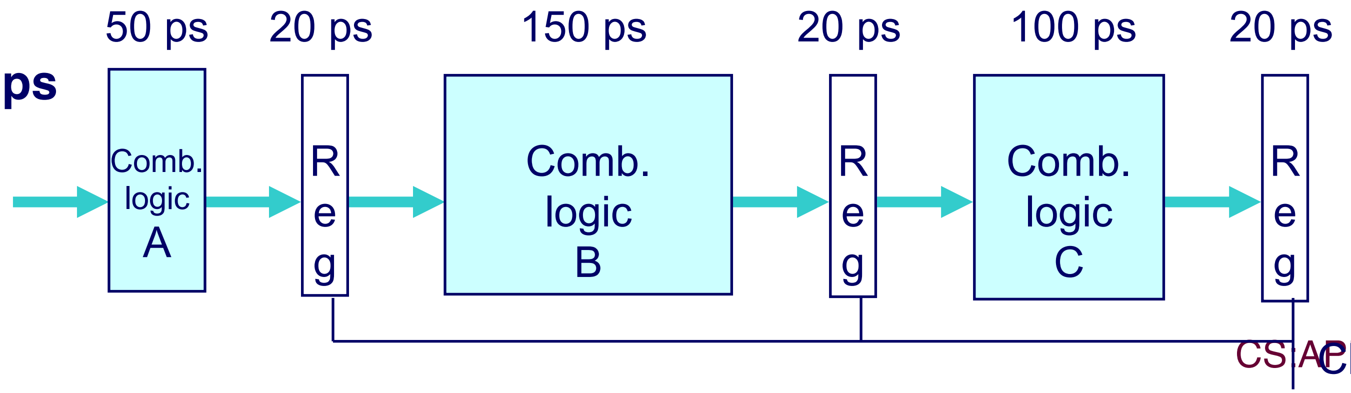
Cycle time: 120 ps

Delay: 360 ps



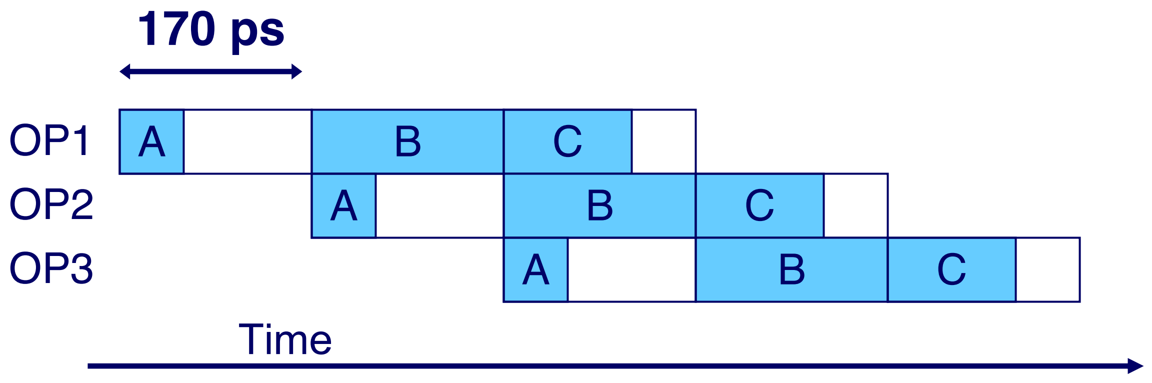
Cycle time: 170 ps

Delay: 510 ps



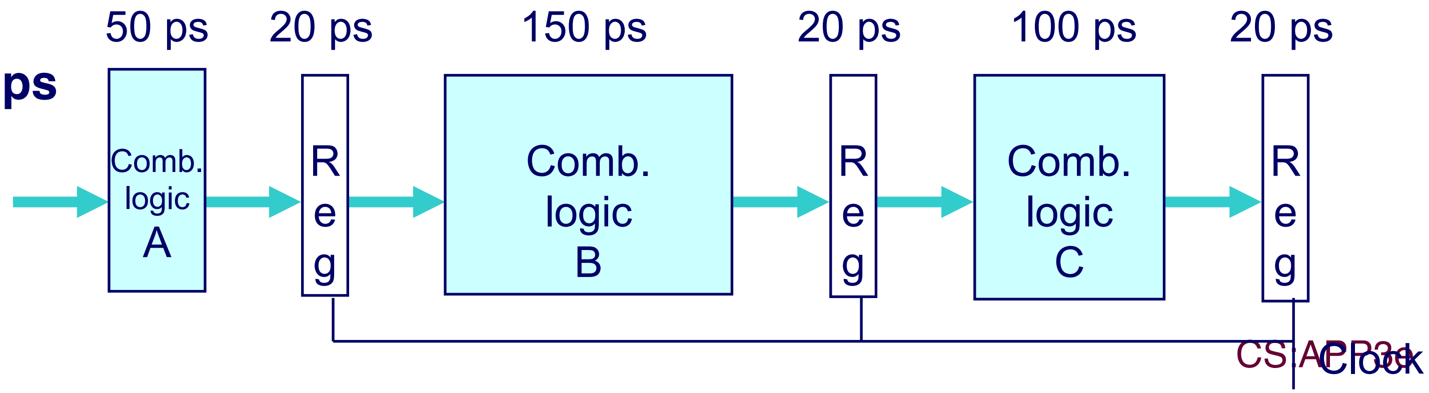
Unbalanced Pipeline

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



Cycle time: 170 ps

Delay: 510 ps

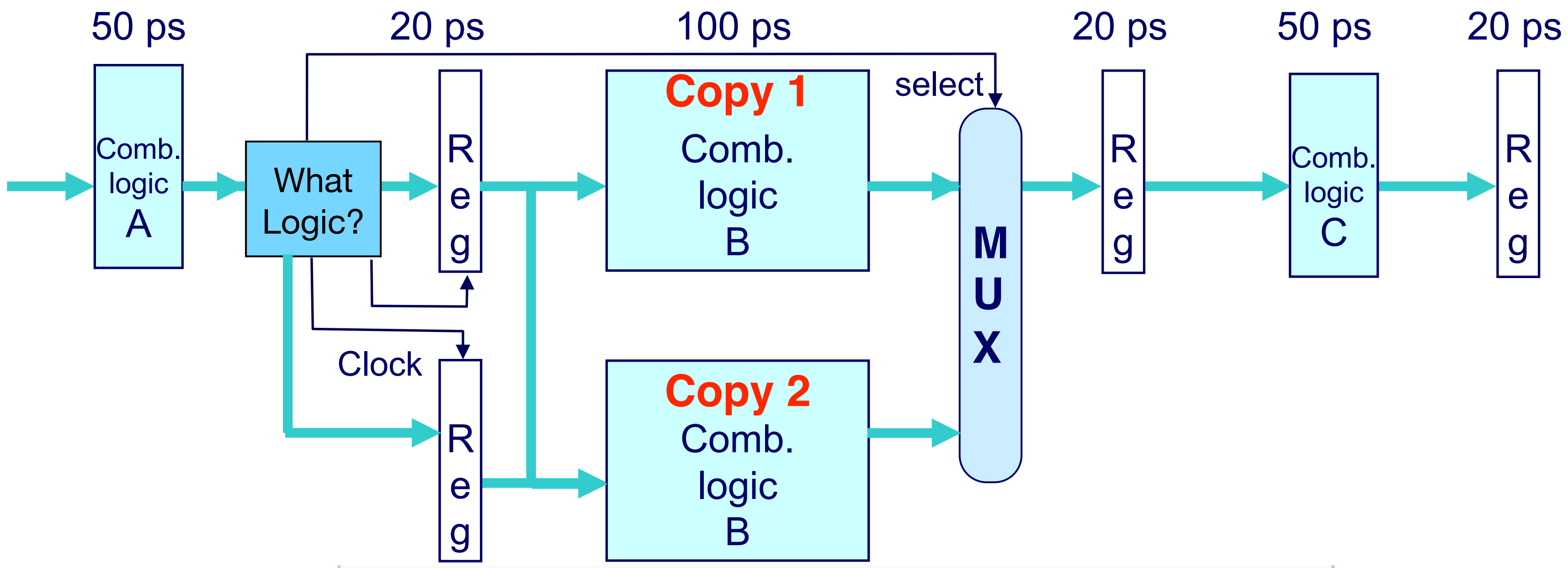


Mitigating Unbalanced Pipeline

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?

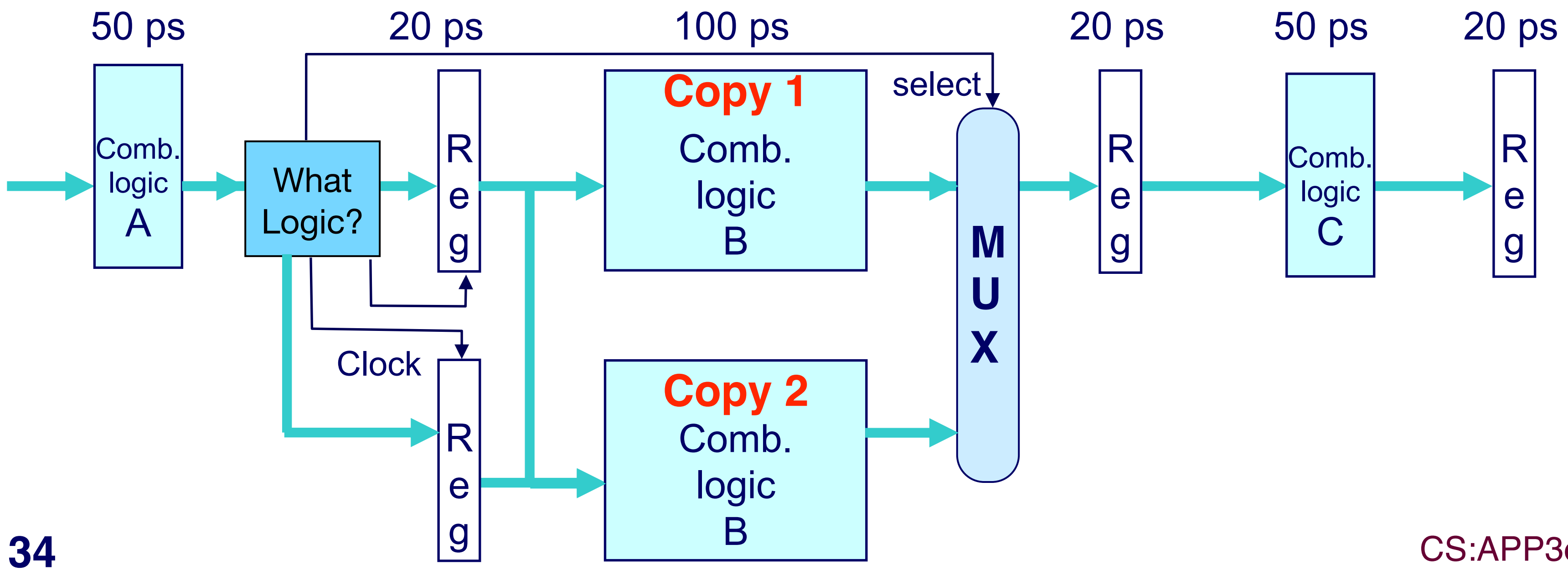
Hint: it needs to control the clock signals of the two registers and the select signal of the MUX.

Mitigating Unbalanced Pipeline

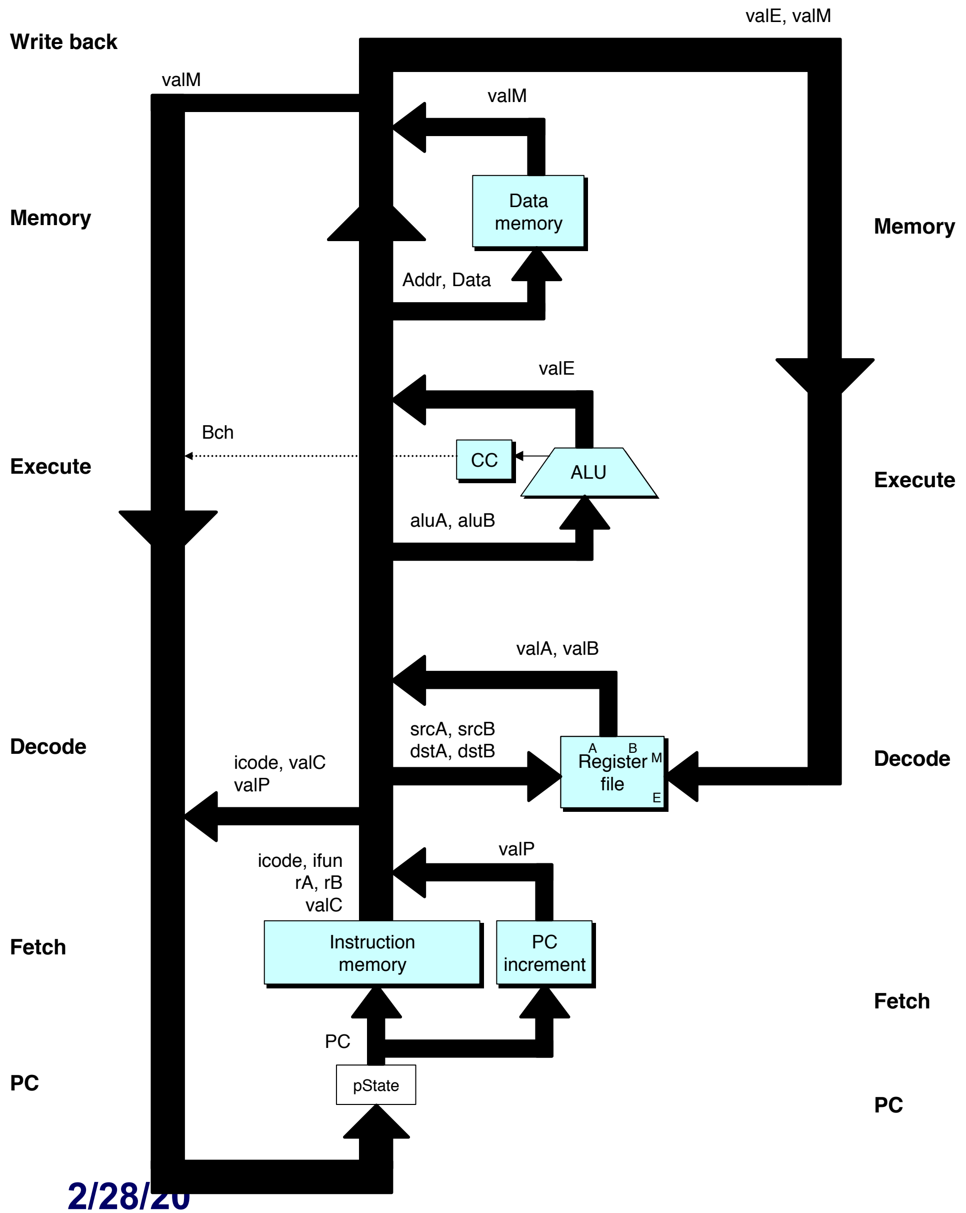
Data sent to copy 1 in odd cycles and to copy 2 in even cycles.

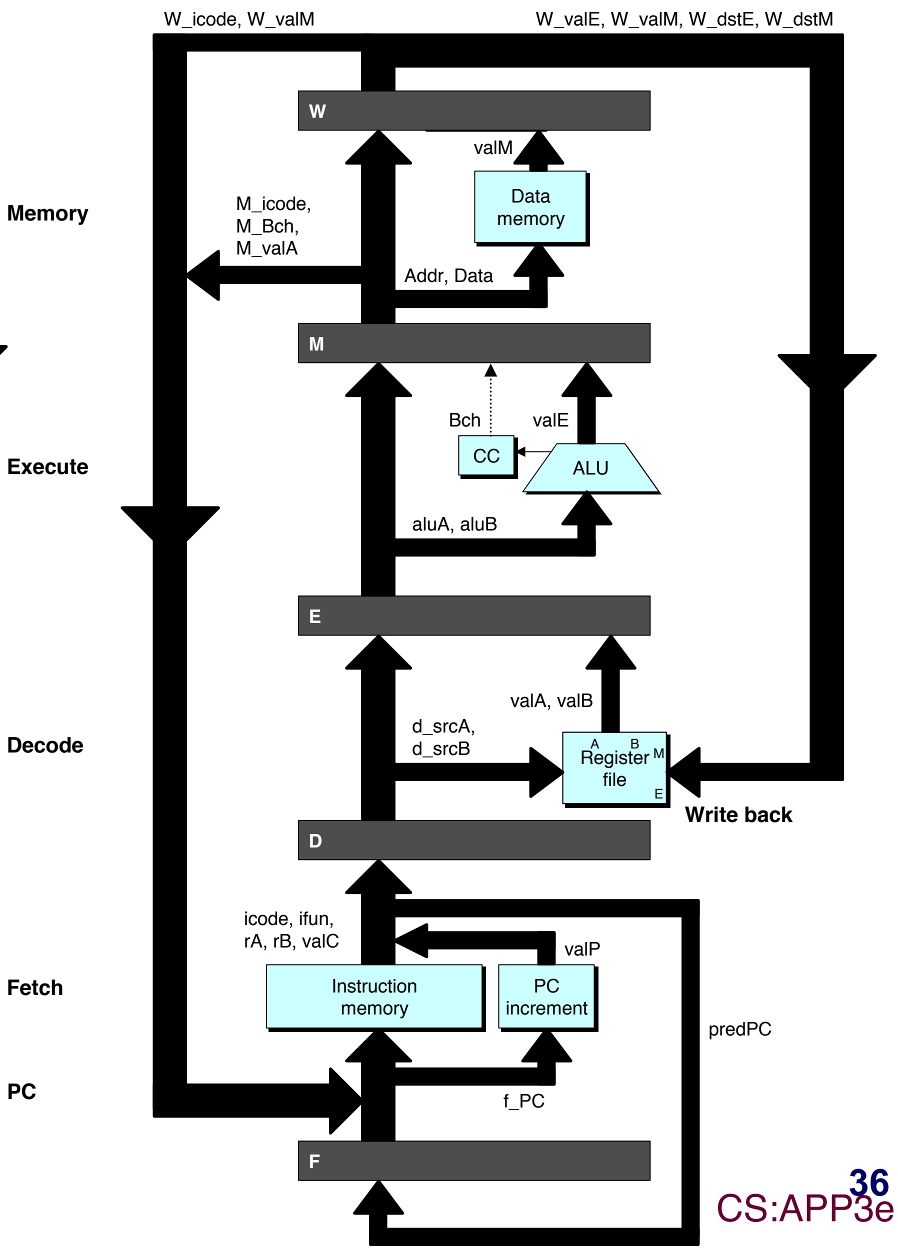
This is called 2-way interleaving. Effectively the same as pipelining Comb. logic B into two sub-stages.

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

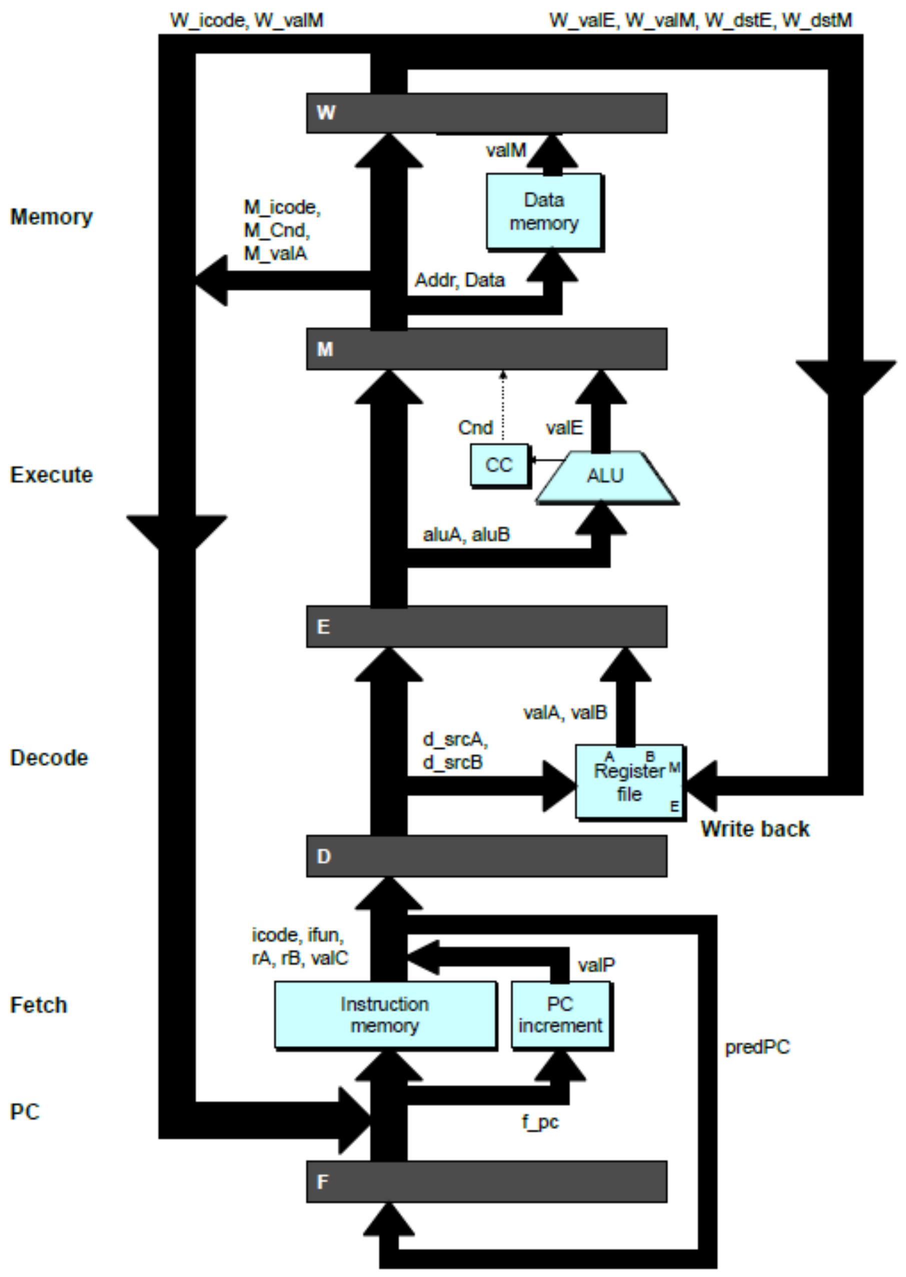
- Operate ALU

Memory

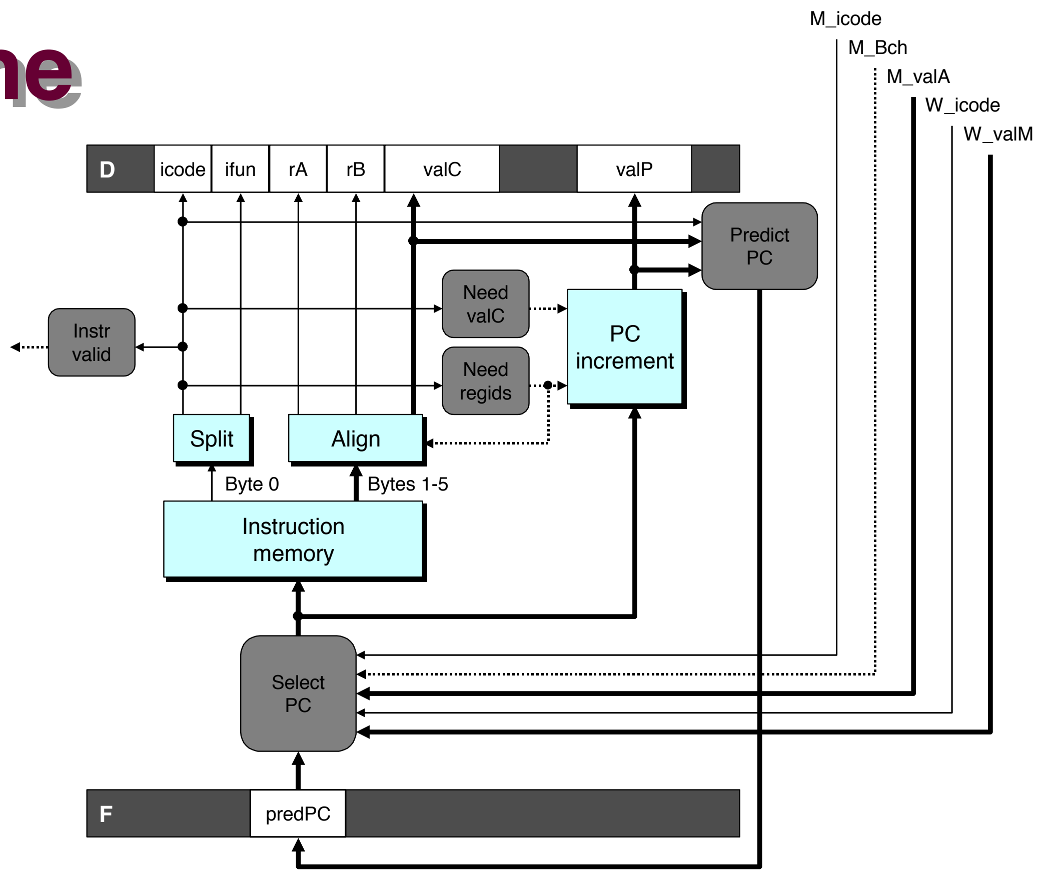
- Read or write data memory

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

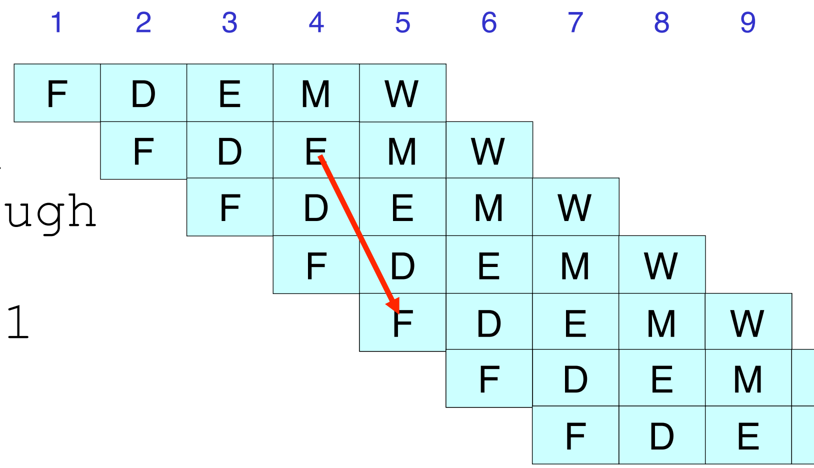
Control Dependency

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**

```
     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 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