

# **CSC 252: Computer Organization**

## **Spring 2026: Lecture 12**

Instructor: Yuhao Zhu

Department of Computer Science  
University of Rochester

# Announcement

- Programming assignment 3 out.
- Open-book mid-term on Friday at this time
  - Anything on paper is fair, nothing electronically.
- “I don’t know” is given 15% partial credit
  - You need to decide if guessing is worthwhile
  - Saves grading time
  - You have to write “I don’t know” and cross out /erase anything else to get credit: A blank answer doesn’t count

# Processor Microarchitecture

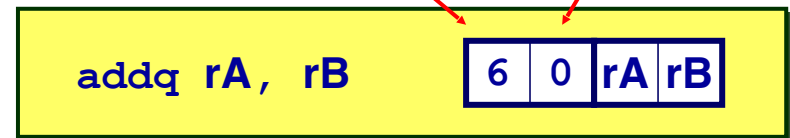
- Sequential, single-cycle microarchitecture implementation
  - Basic idea
  - Hardware implementation
- Pipelined microarchitecture implementation
  - Basic Principles
  - Difficulties: Control Dependency
  - Difficulties: Data Dependency

# Executing an ADD instruction

- How does the processor execute `addq %rax,%rsi`
- The binary encoding is `60 06`

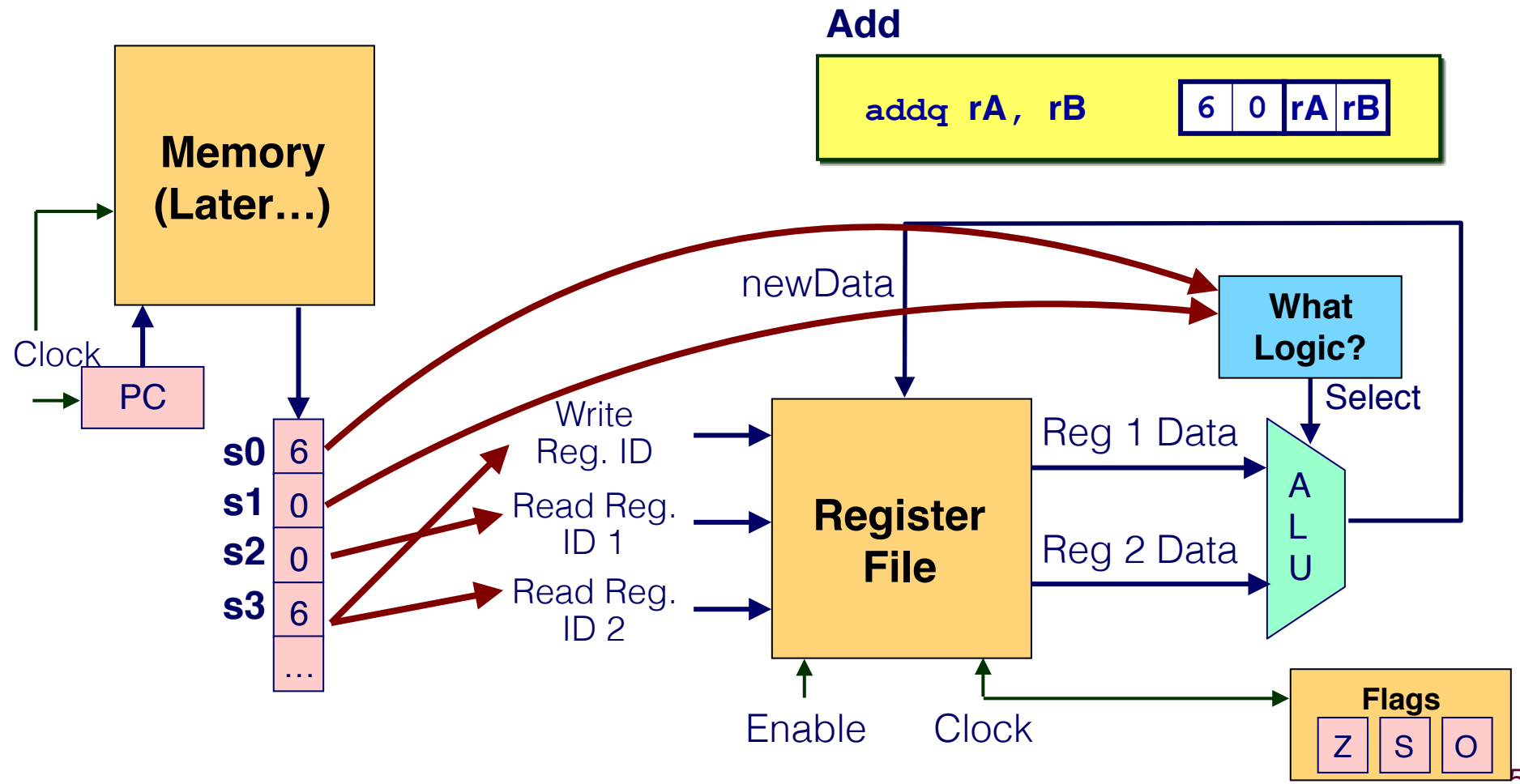
**Instruction Code**  
**Add**

**Function Code**



# Executing an ADD instruction

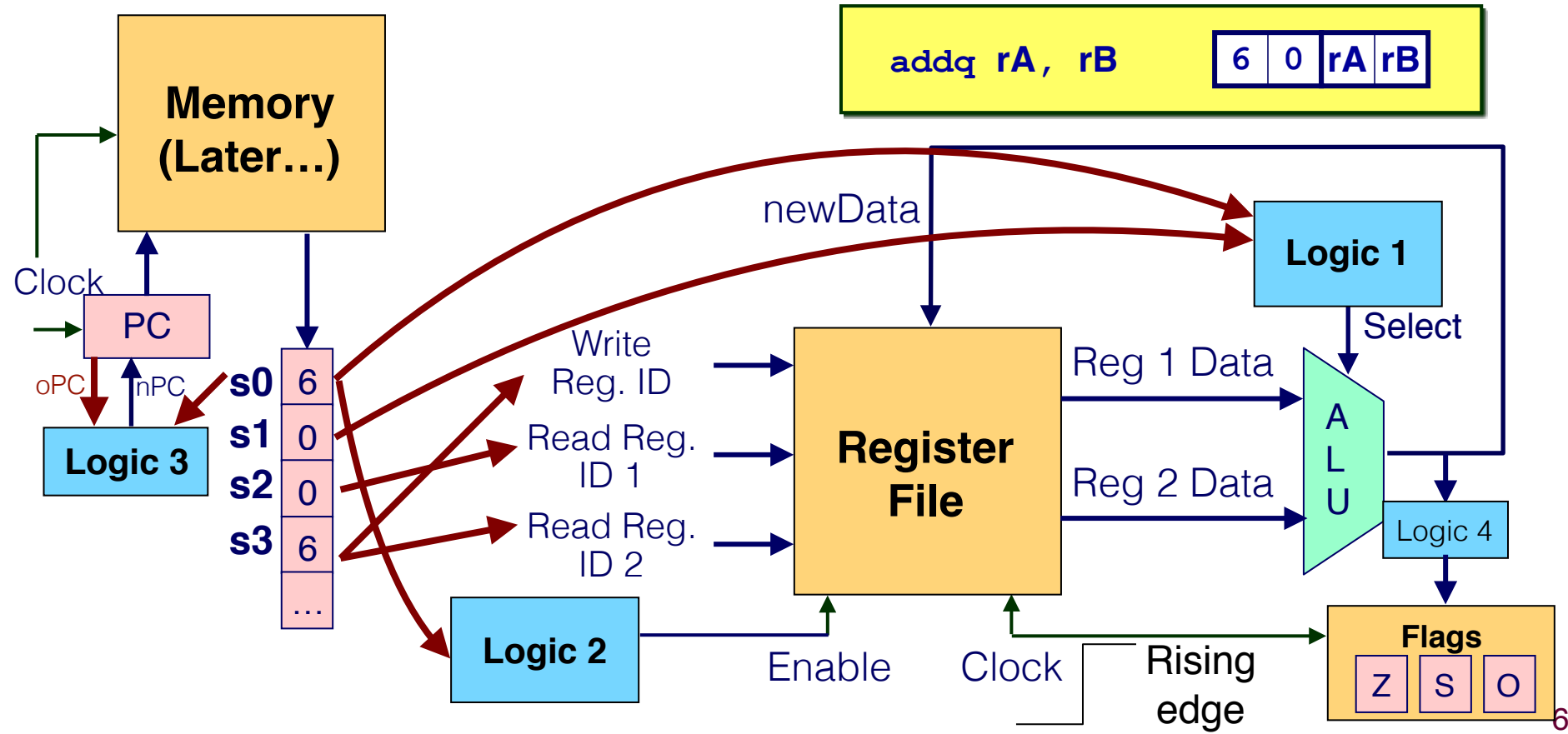
- How does the processor execute `addq %rax,%rsi`
- The binary encoding is `60 06`



# Executing an ADD instruction

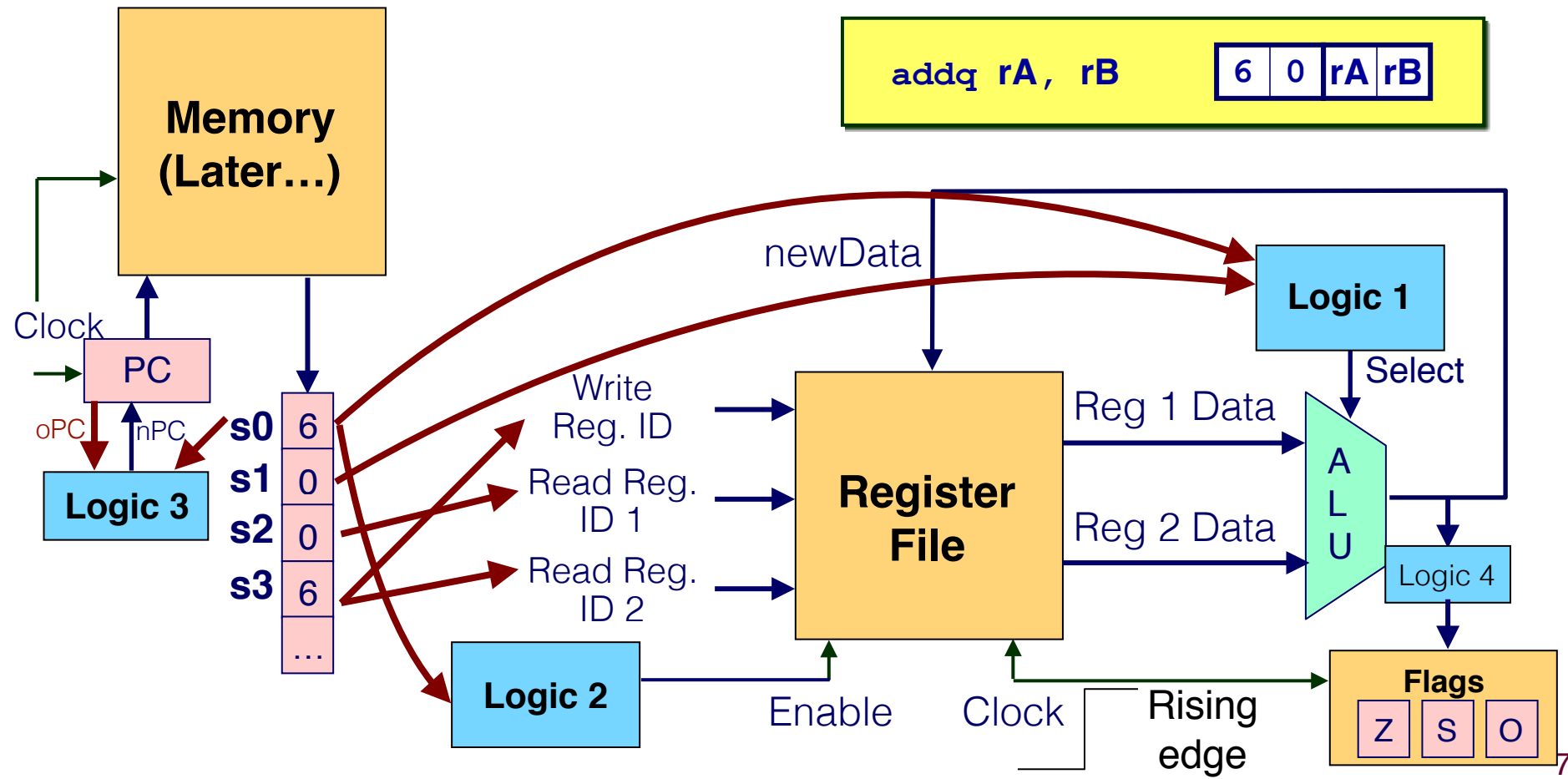
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

How do these logics get implemented?



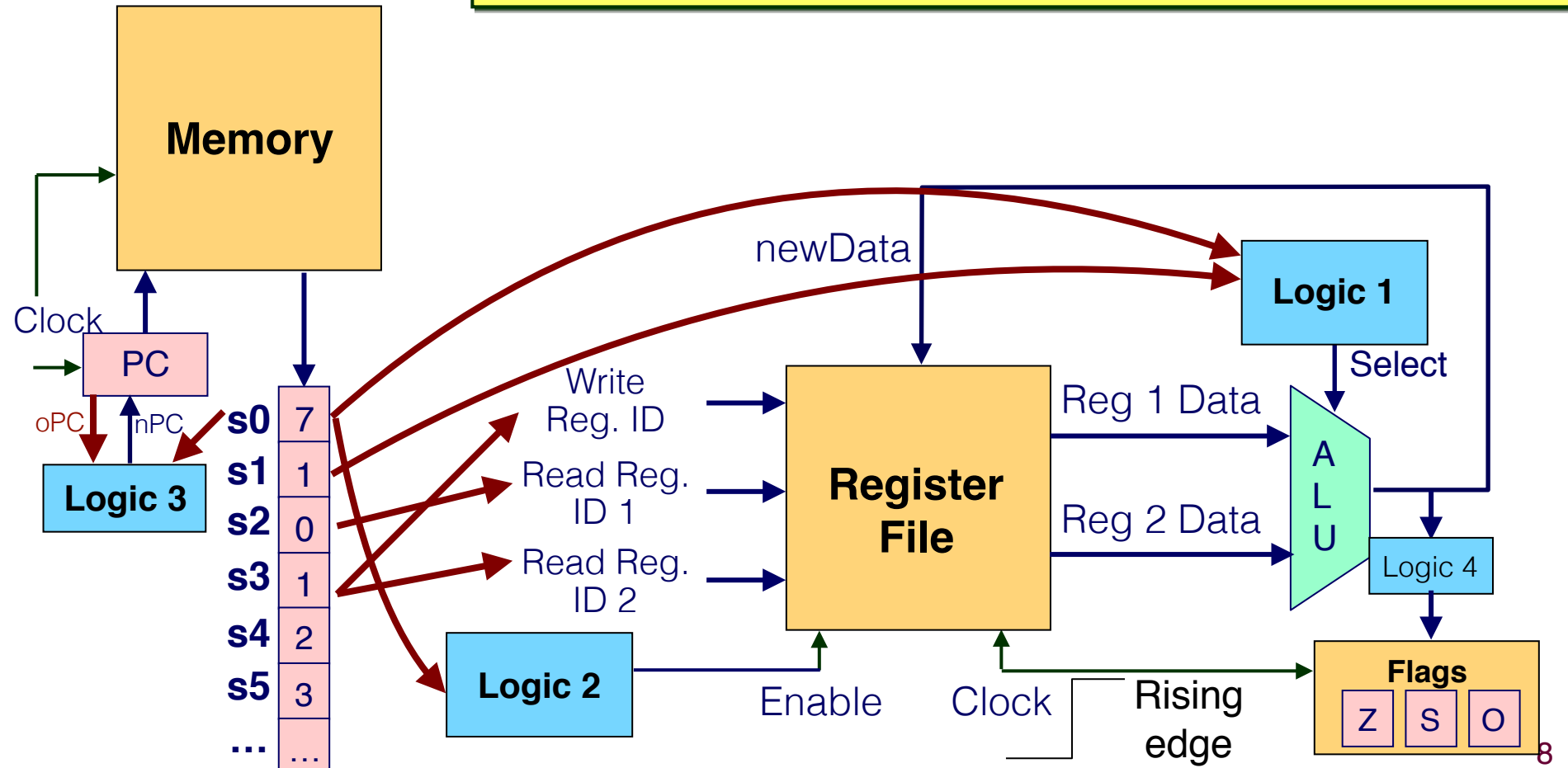
# Executing an ADD instruction

- When the rising edge of the clock arrives, the RF/PC/Flags will be written.
- So the following has to be ready: newData, nPC, which means Logic1, Logic2, Logic3, and Logic4 has to finish.



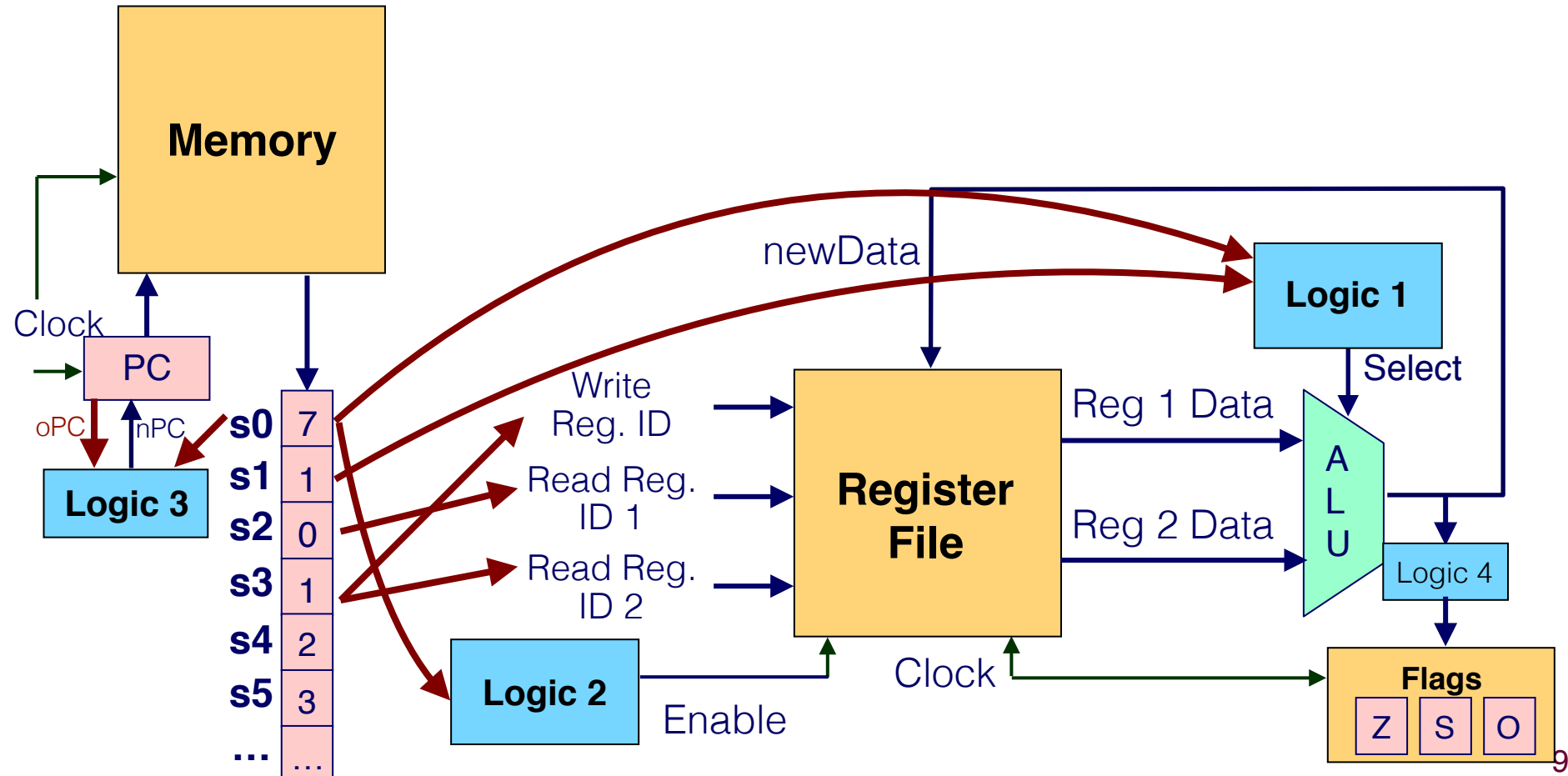
# Executing a JLE instruction

- Let's say the binary encoding for `jle .L0` is `71 0123000000000000`
- What are the logics now?



# Executing a JLE instruction

- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;



jle Dest

7

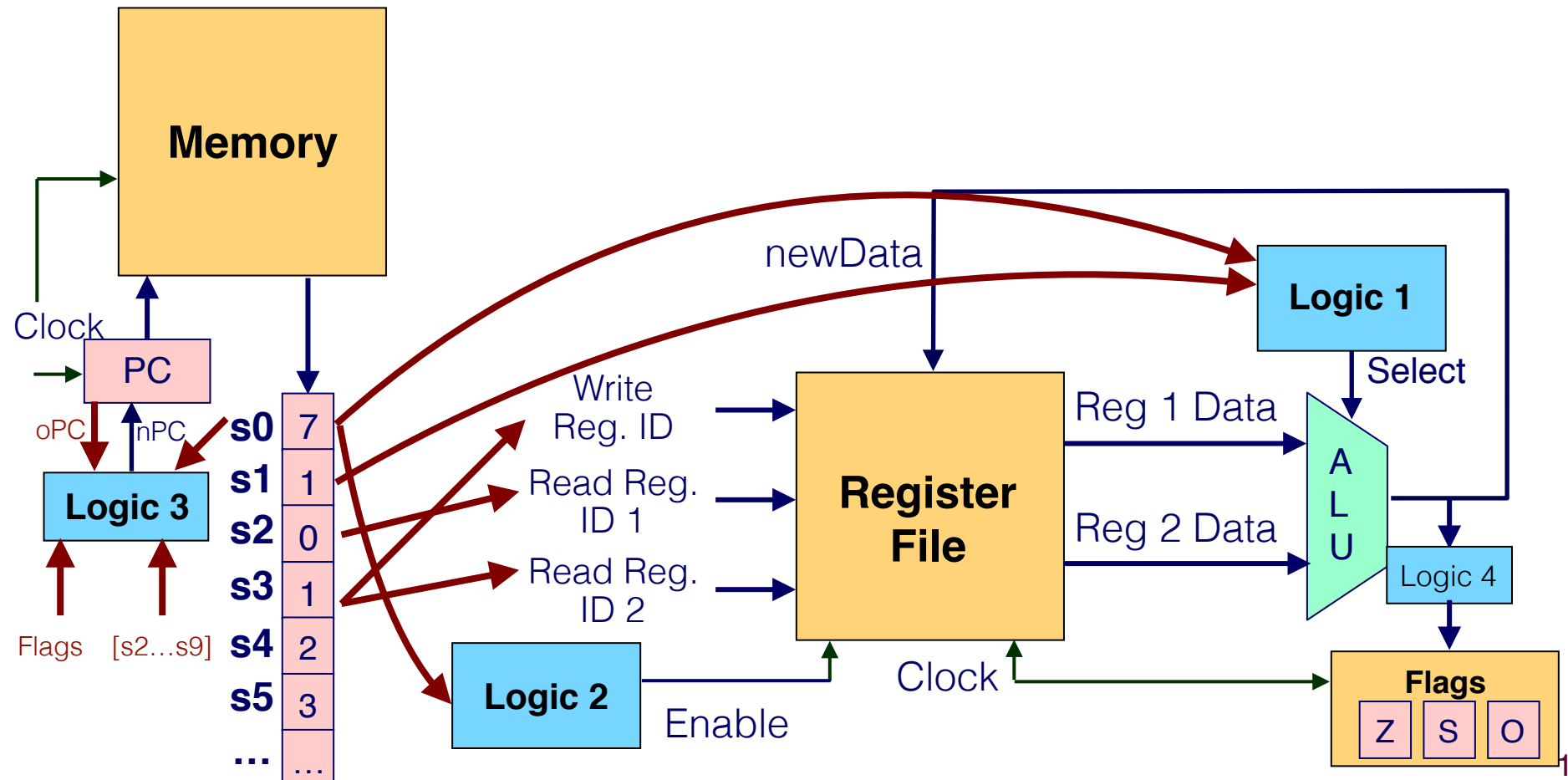
1

Dest

# Executing a JLE instruction

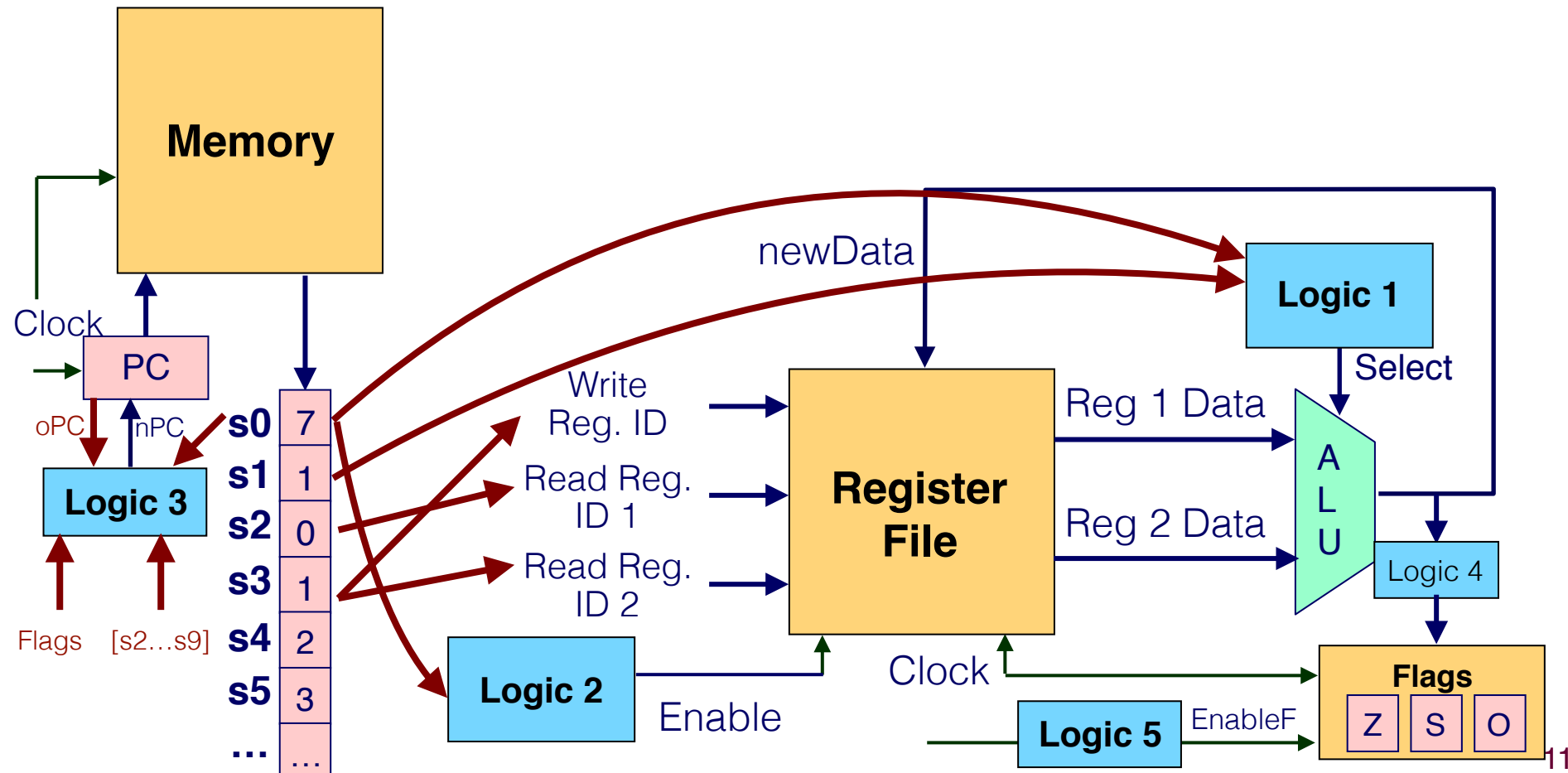
- Logic 3??

if (s0 == 6) nPC = oPC + 2;

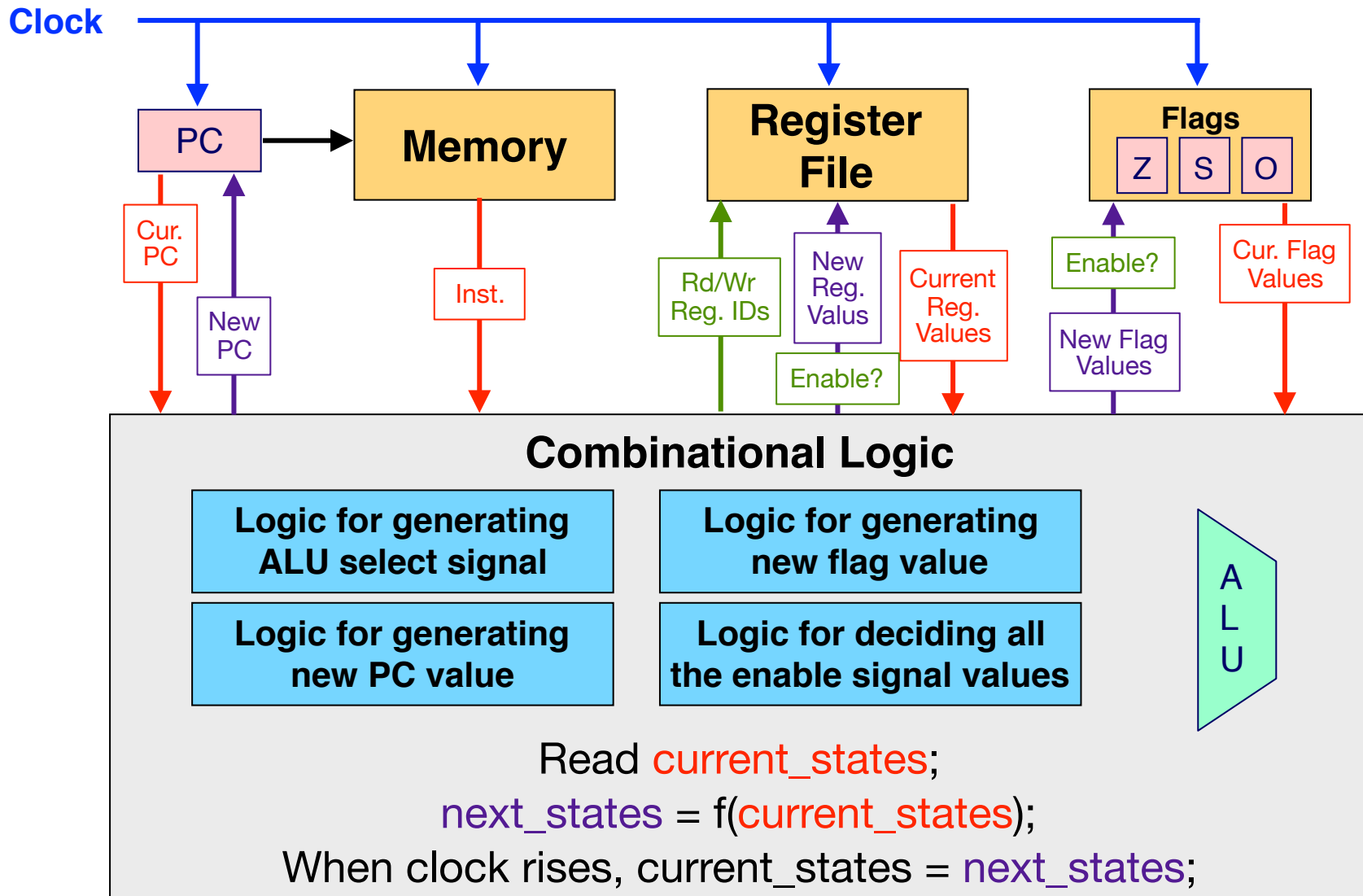


# Executing a JLE instruction

- Logic 4? Does JLE write flags?
- Need another piece of logic.
- Logic 5: if (s0 == 7) EnableF = 0; else if (s0 == 6) EnableF = 1;

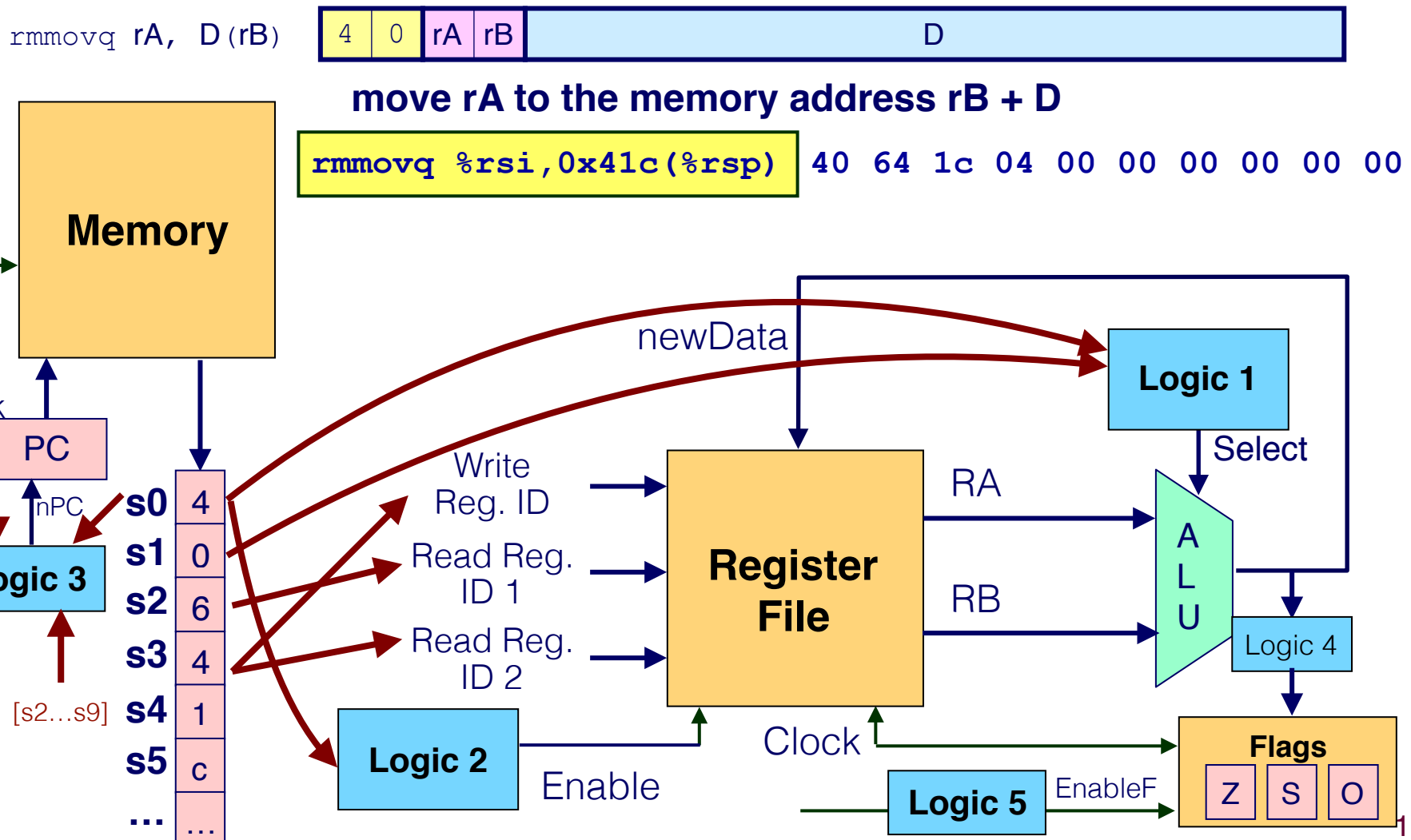


# Microarchitecture (So far)



# Executing a MOV instruction

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

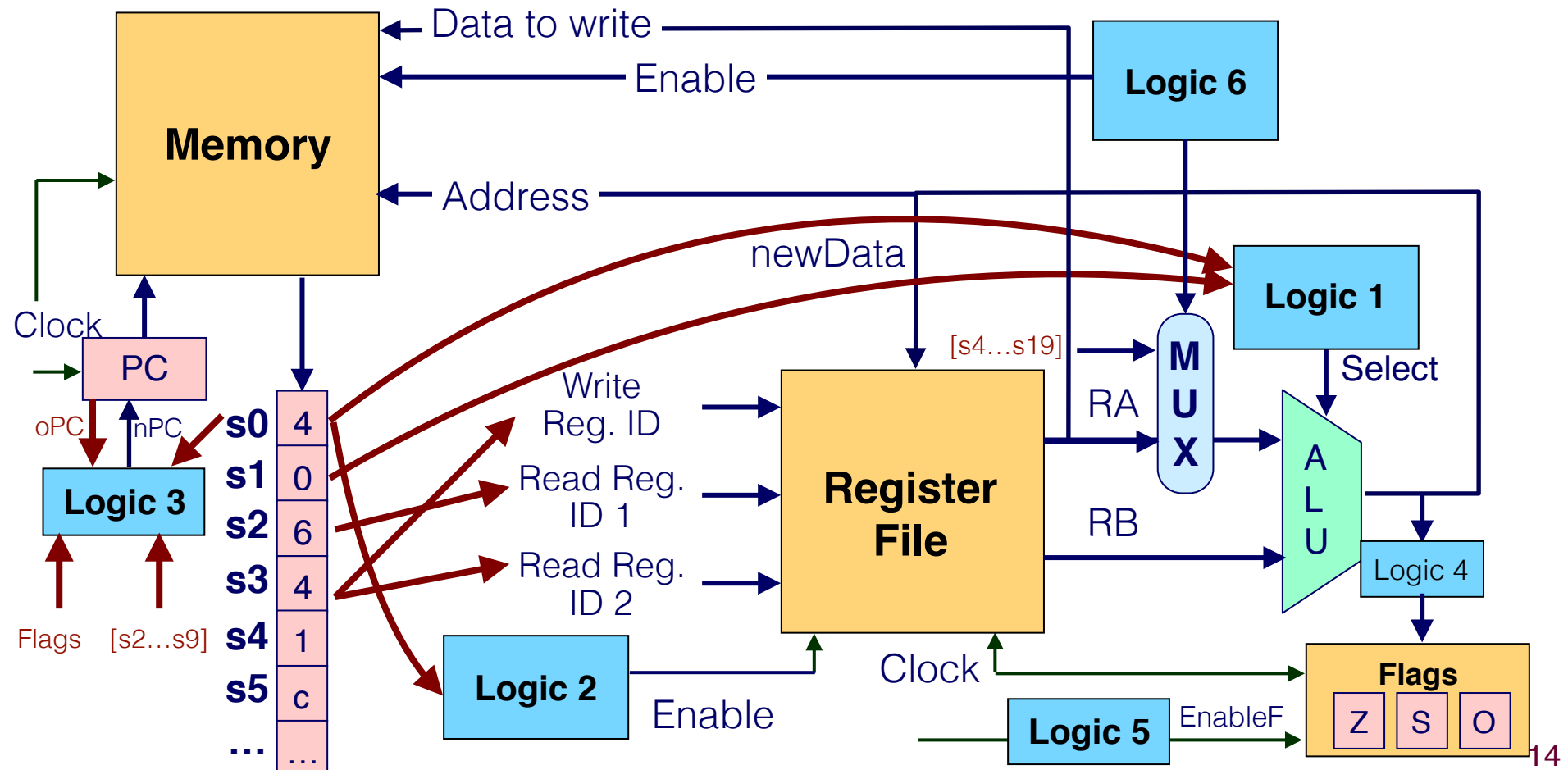


## move rA to the memory address rB + D

`rmmovq rA, D(rB)`



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

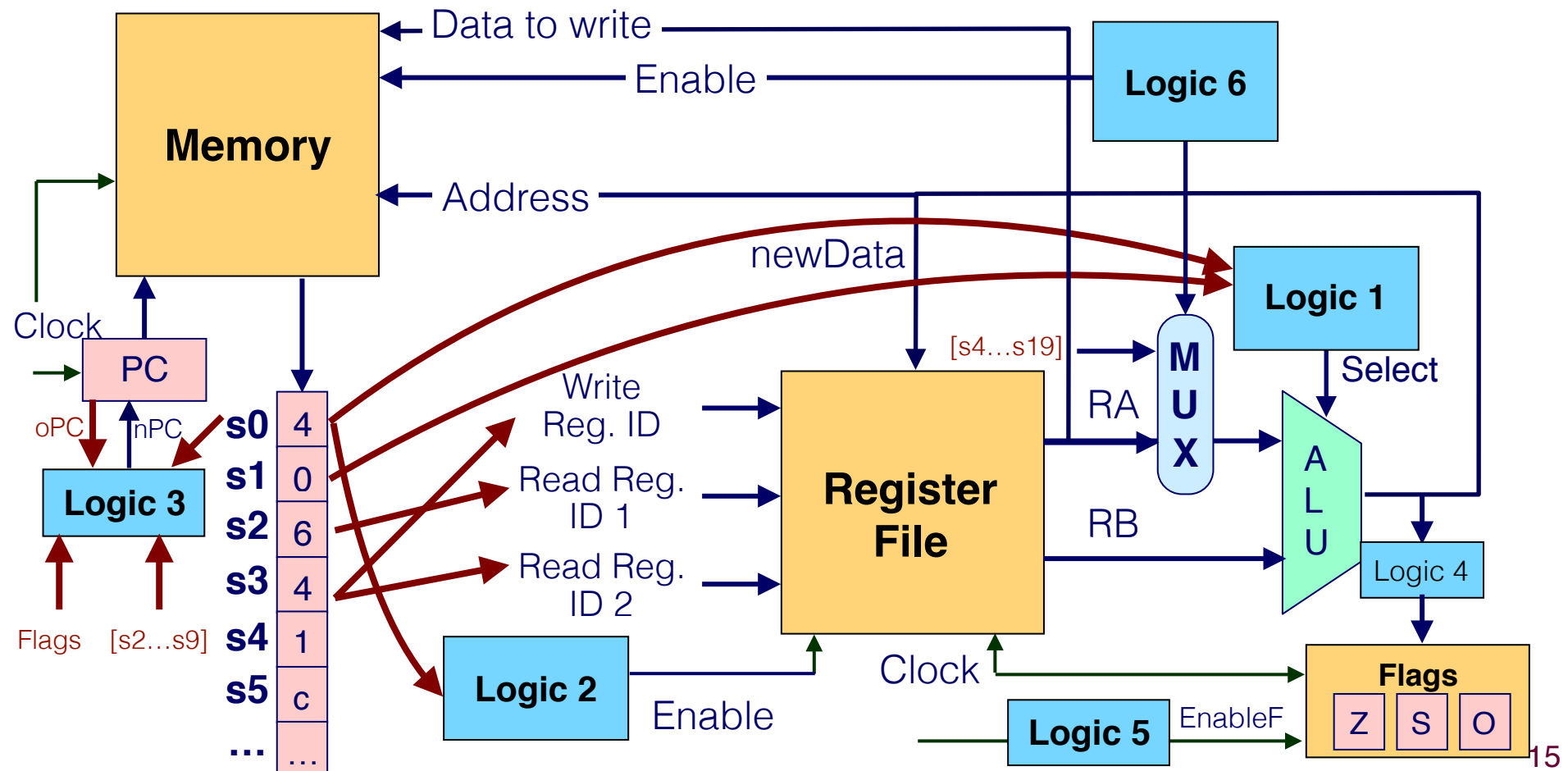


# How About Memory to Register MOV?

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

`mrmovq D(rB), rA`

|   |   |    |    |   |  |  |  |  |  |  |  |  |  |  |  |
|---|---|----|----|---|--|--|--|--|--|--|--|--|--|--|--|
| 4 | 0 | rA | rB | D |  |  |  |  |  |  |  |  |  |  |  |
|---|---|----|----|---|--|--|--|--|--|--|--|--|--|--|--|

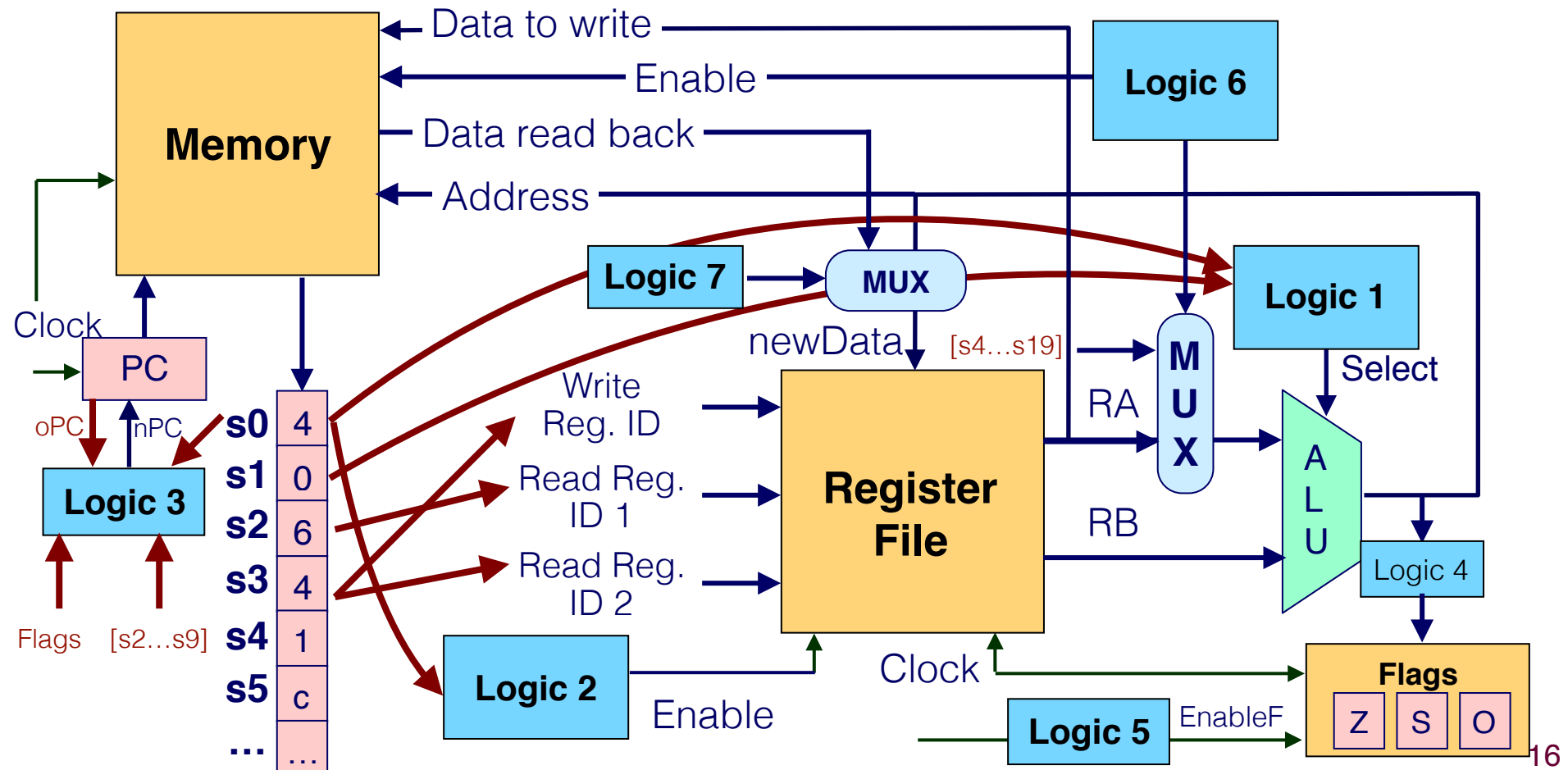


# How About Memory to Register MOV?

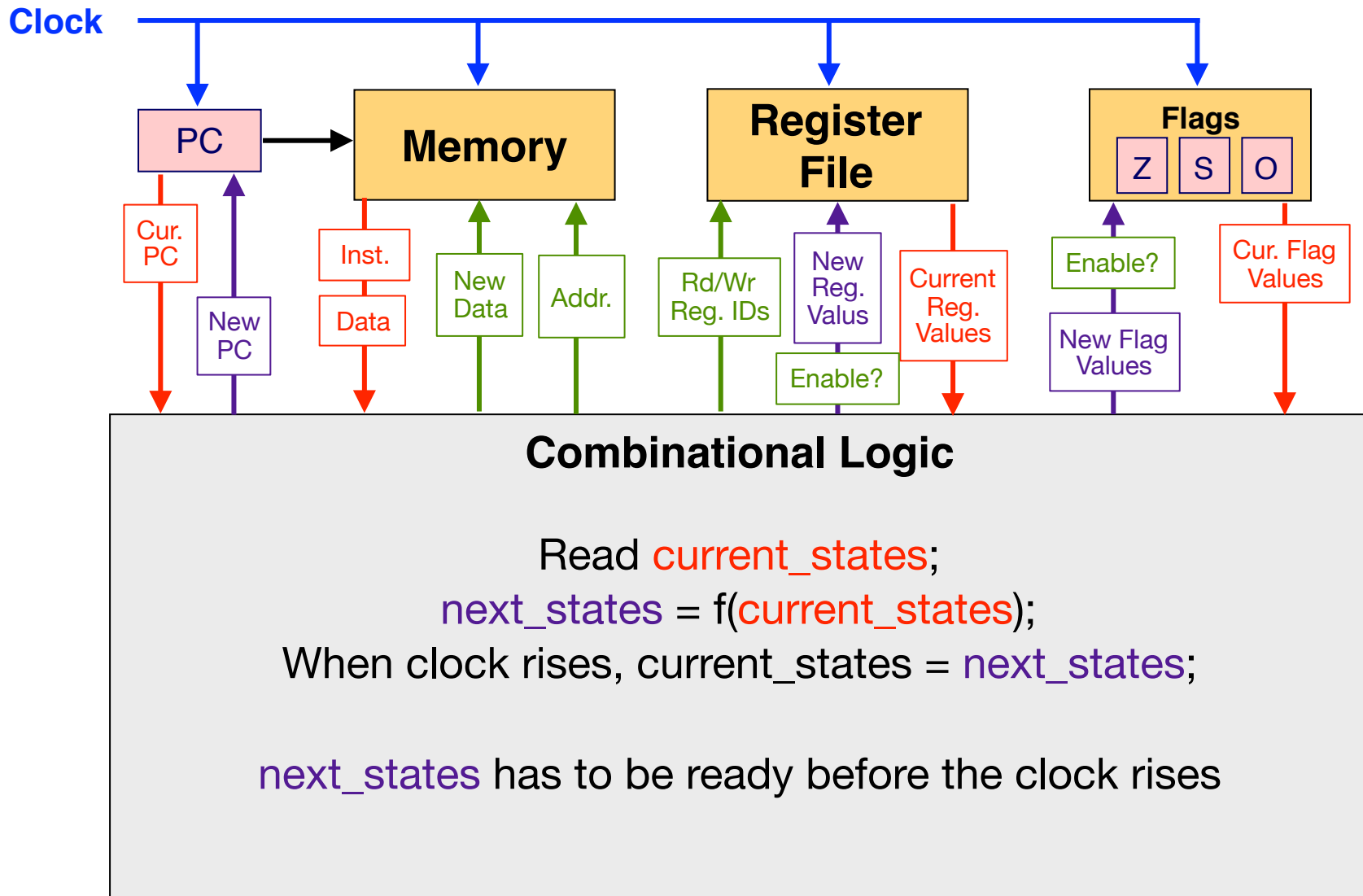
move data at memory address  $rB + D$  to  $rA$

`mrmovq D(rB), rA`

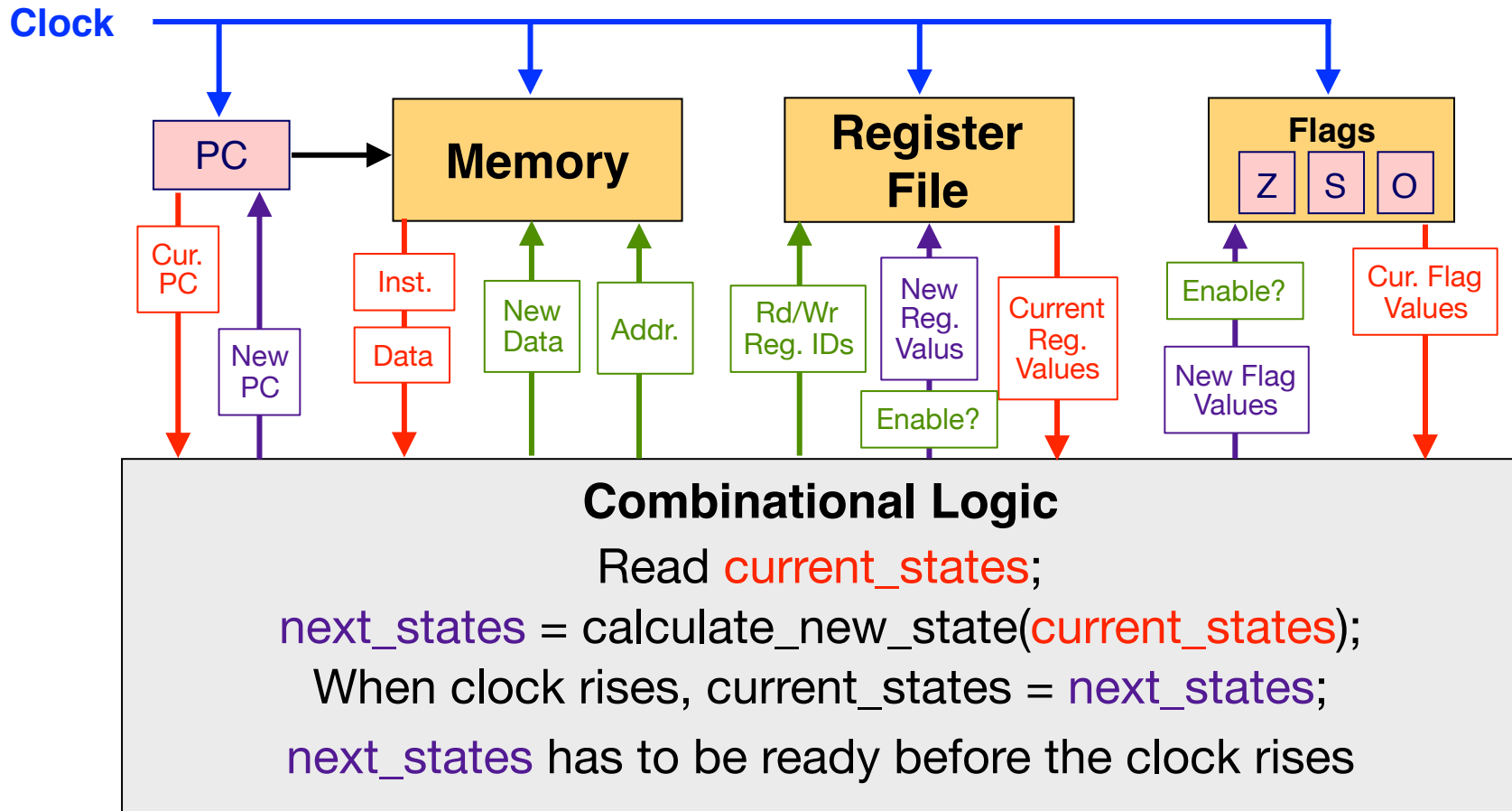
|   |   |    |    |   |  |  |  |  |  |  |  |  |  |  |  |
|---|---|----|----|---|--|--|--|--|--|--|--|--|--|--|--|
| 4 | 0 | rA | rB | D |  |  |  |  |  |  |  |  |  |  |  |
|---|---|----|----|---|--|--|--|--|--|--|--|--|--|--|--|



# Microarchitecture (with MOV)



# 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.

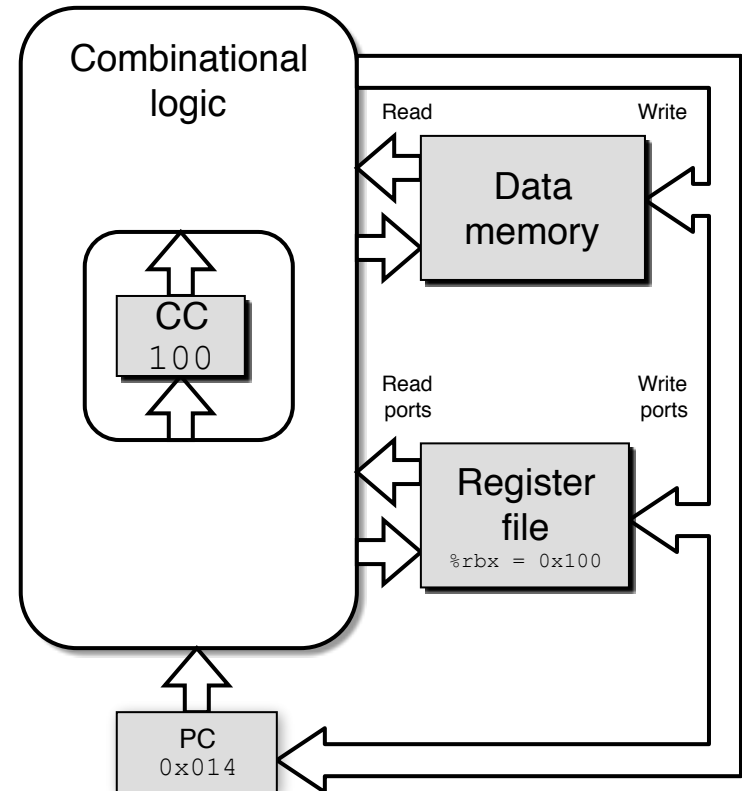
# Single-Cycle Microarchitecture: Illustration

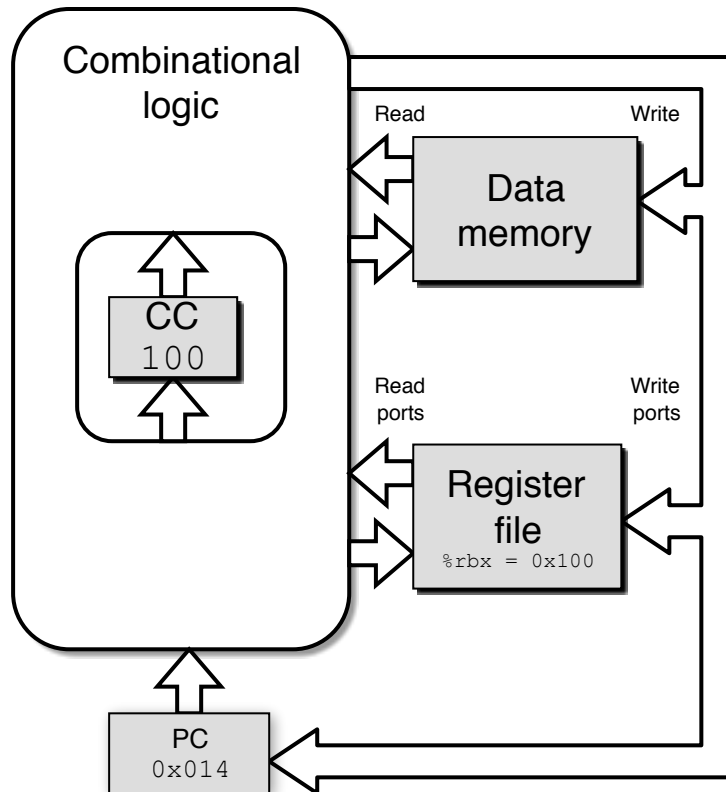
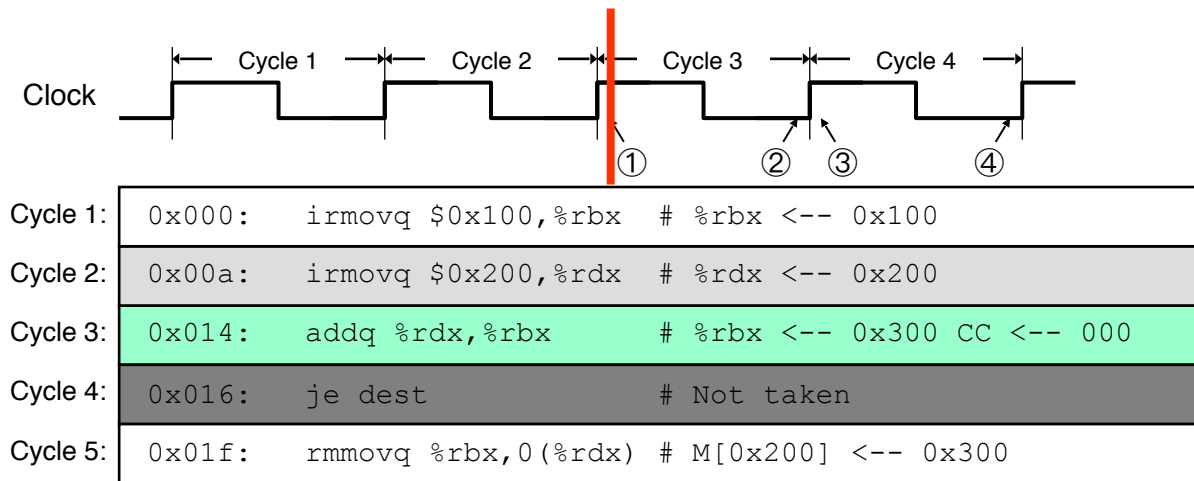
Think of it as a state machine

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

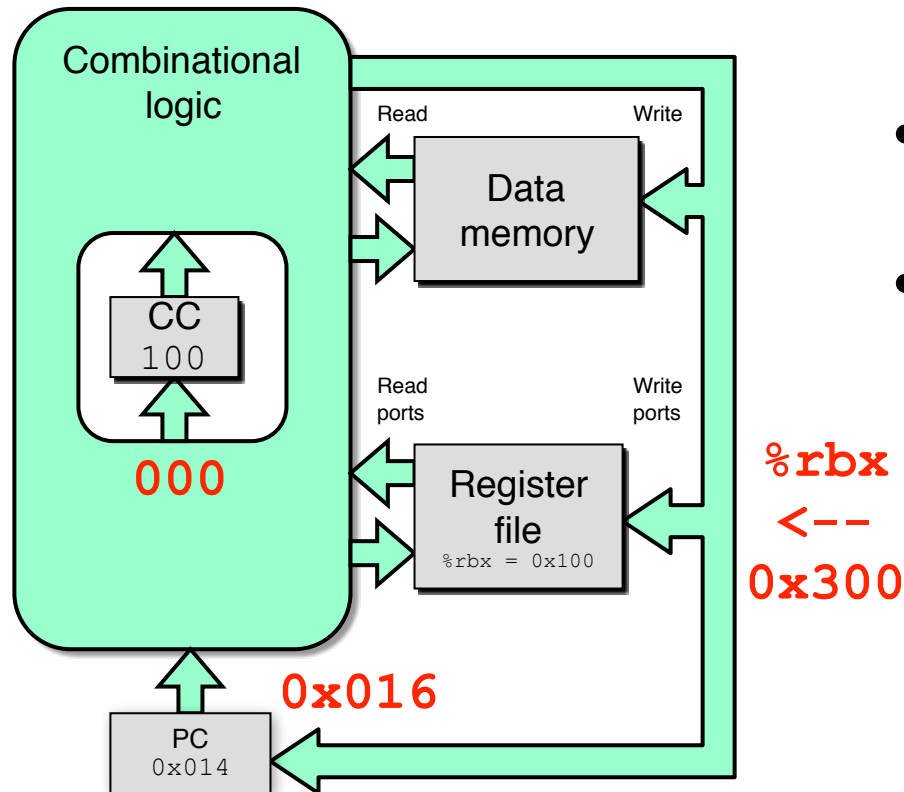
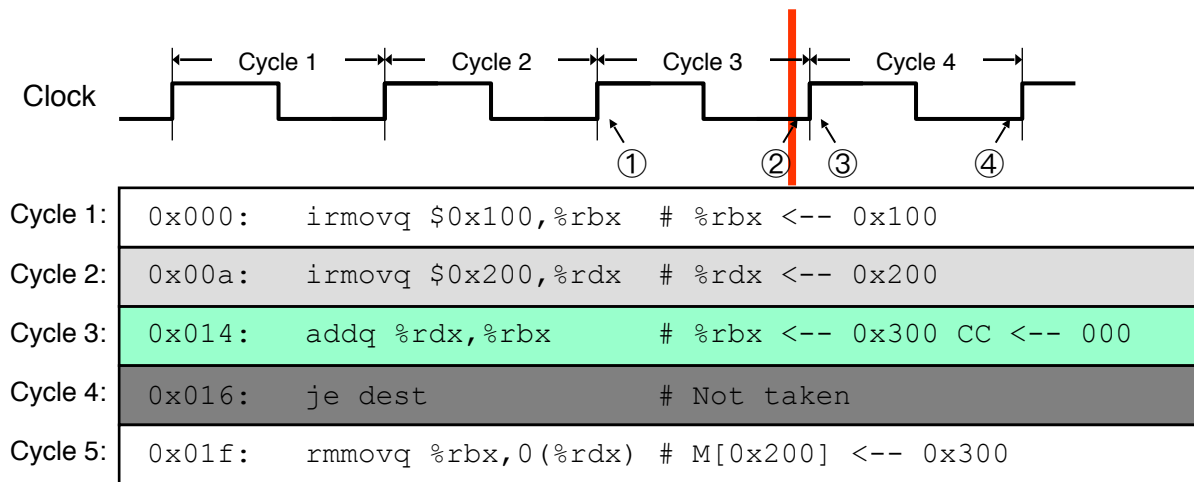
States (All updated as clock rises)

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

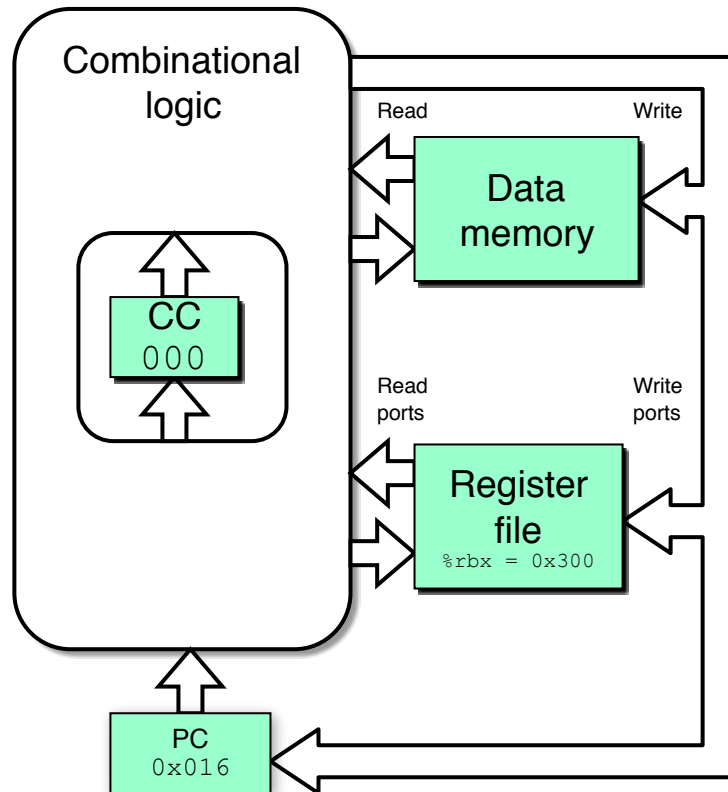
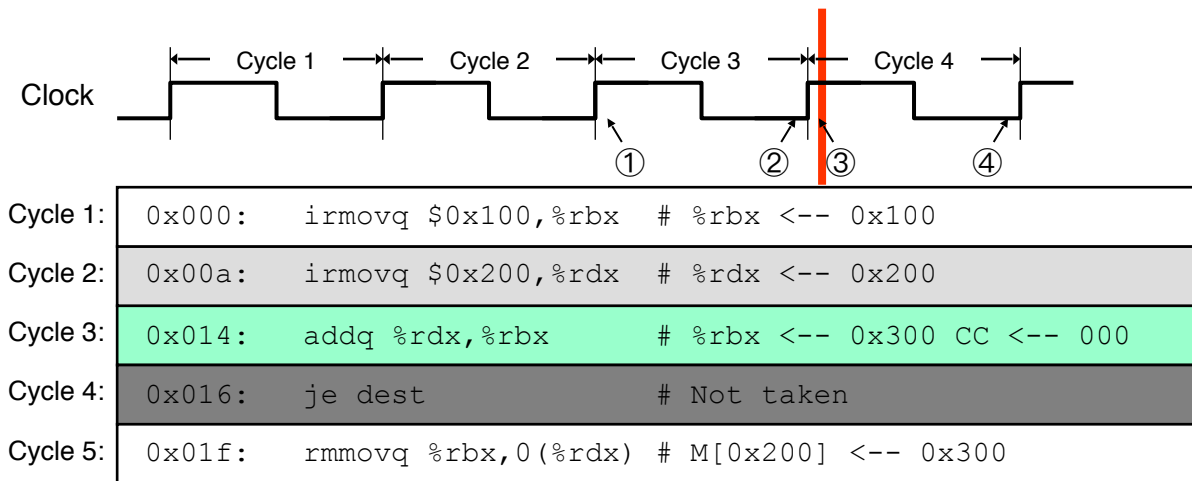




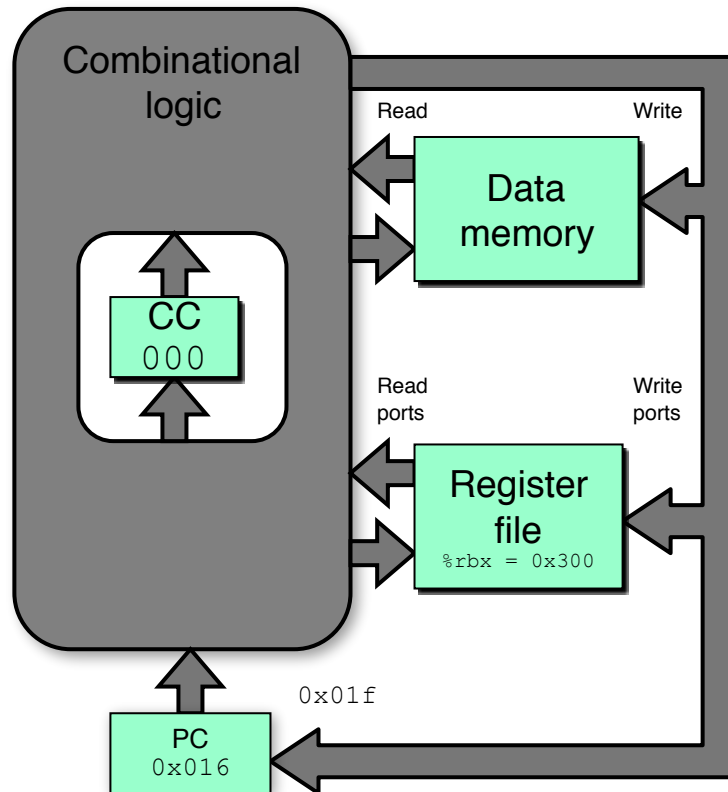
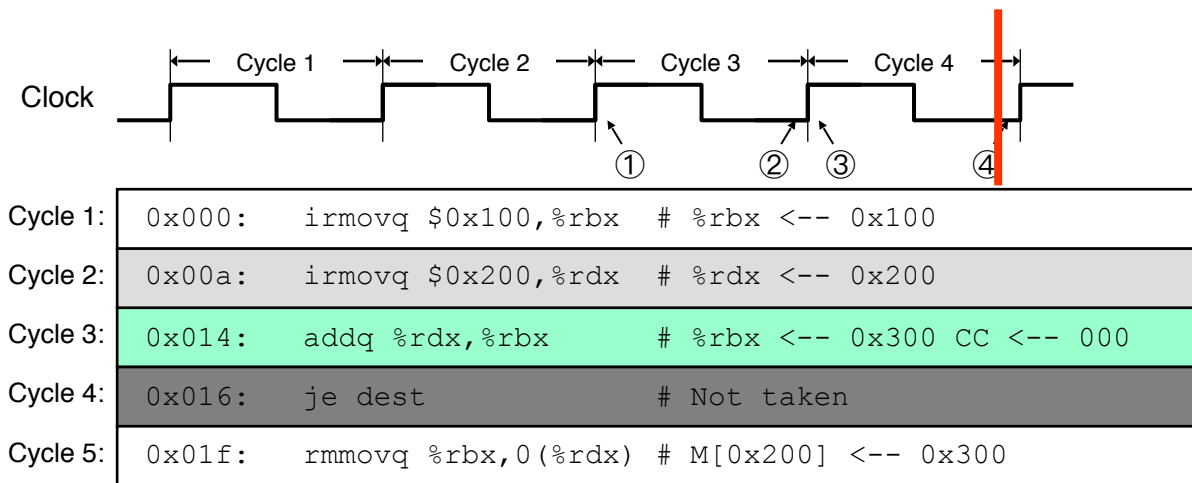
- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes



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



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



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

# Performance Model

$$\begin{aligned} \text{Execution time} &= \text{\# of } \textcolor{green}{\text{Dynamic}} \text{ Instructions} && \text{CPI} \\ \text{of a program} &&& \\ \text{(in seconds)} &&& \\ &\times \boxed{\text{\# of cycles taken to execute an instruction (on average)}} && \\ &/ \boxed{\text{number of cycles per second}} && \text{Clock Frequency} \\ &&& \text{(1/cycle time)} \end{aligned}$$

# Improving Performance

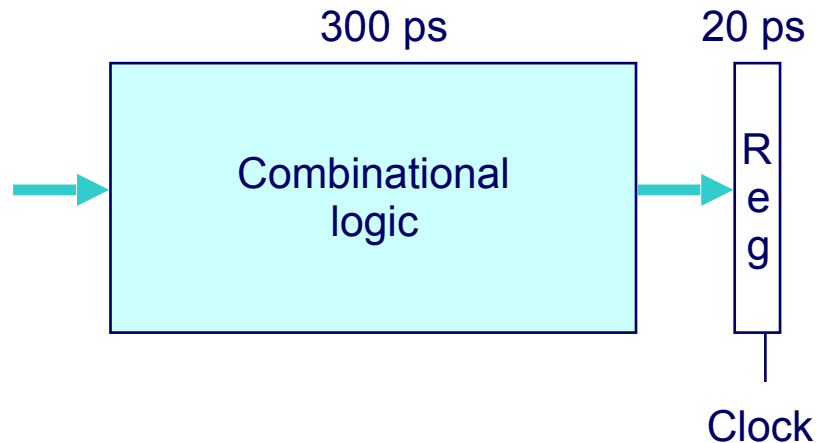
$$\begin{aligned} \text{Execution time} \\ \text{of a program} \\ \text{(in seconds)} &= \# \text{ of } \text{Dynamic} \text{ Instructions} \\ &\quad \times \# \text{ of cycles taken to execute an instruction (on average)} \\ &\quad / \text{ number of cycles per second} \end{aligned}$$

- 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

- 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?

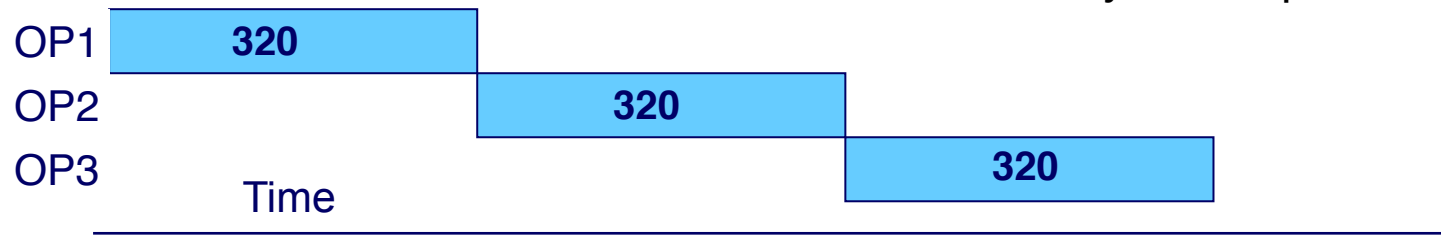
# A Motivating Example



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

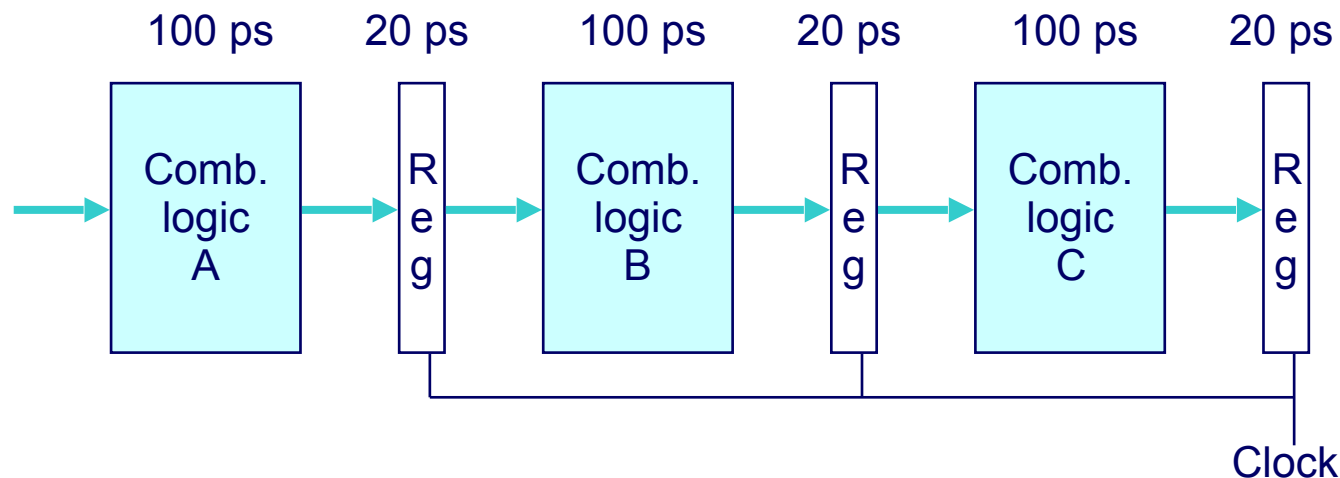
# Pipeline Diagrams

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



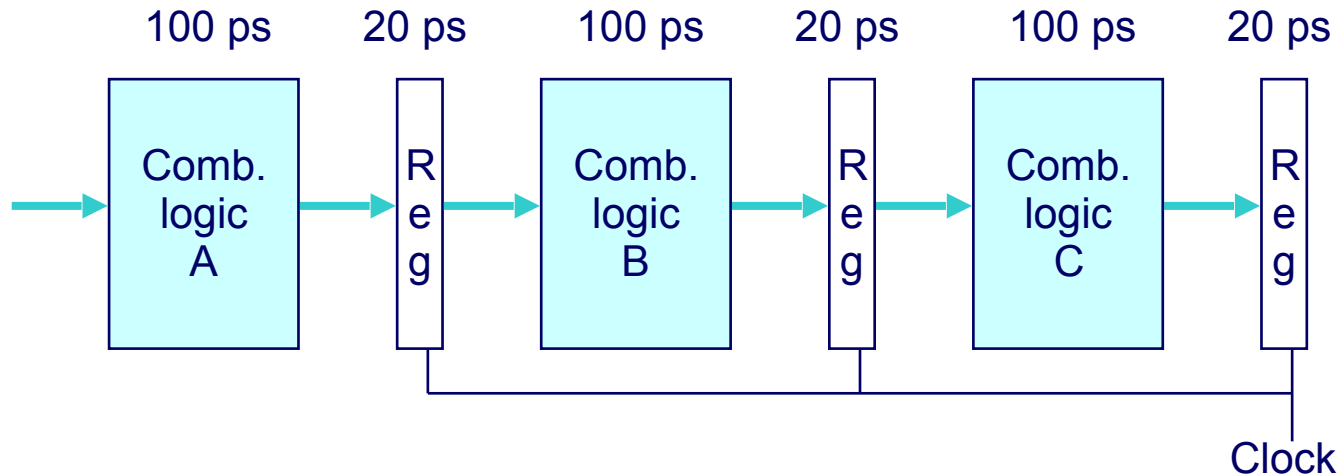
- 3 instructions will take 960 ps to finish
  - First cycle: Inst 1 takes 300 ps to compute new state, 20 ps to store the new states
  - Second cycle: Inst 2 starts; it takes 300 ps to compute new states, 20 ps to store new states
  - And so on...

# 3-Stage Pipelined Version

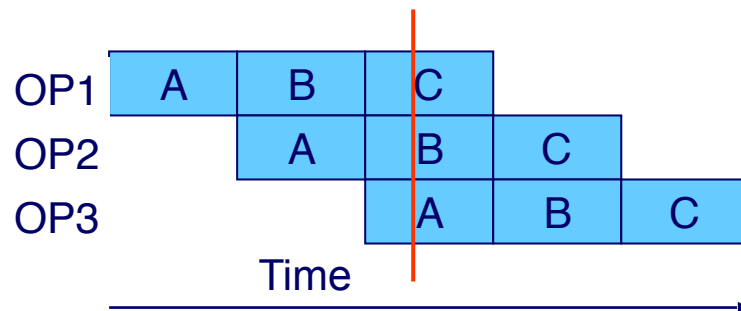


- Divide combinational logic into 3 stages of 100 ps each
- Insert registers between stages to store intermediate data between stages. These are called pipeline registers (ISA-invisible)
- Can begin a new instruction as soon as the previous one finishes stage A and has stored the intermediate data.
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

# 3-Stage Pipelined Version

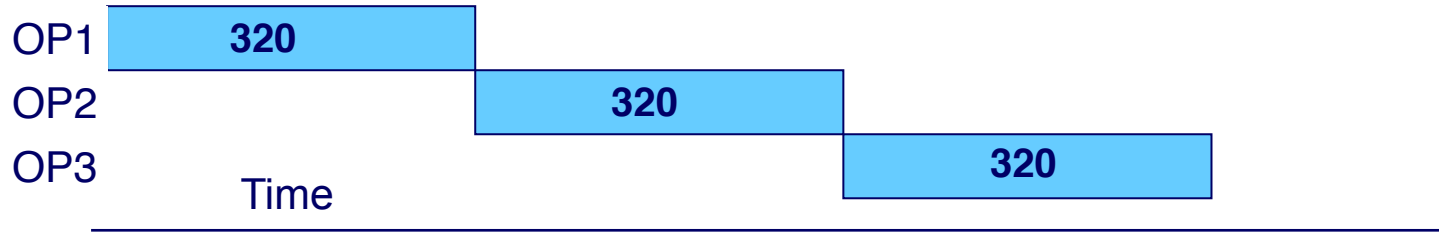


## 3-Stage Pipelined



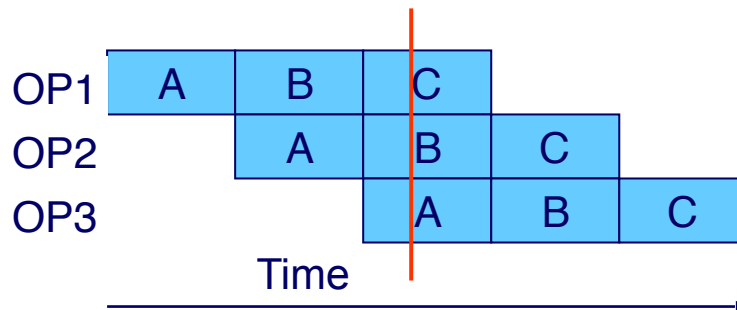
# Comparison

## Unpipelined



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

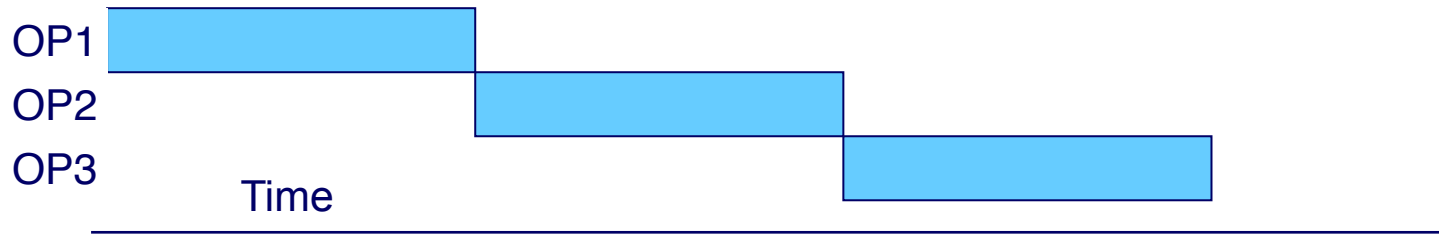
## 3-Stage Pipelined



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

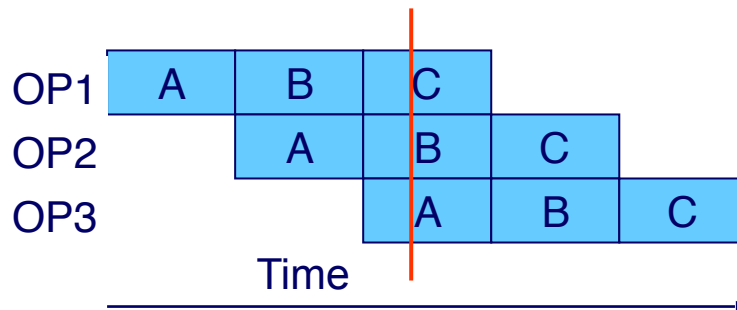
# Benefits of Pipelining

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



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

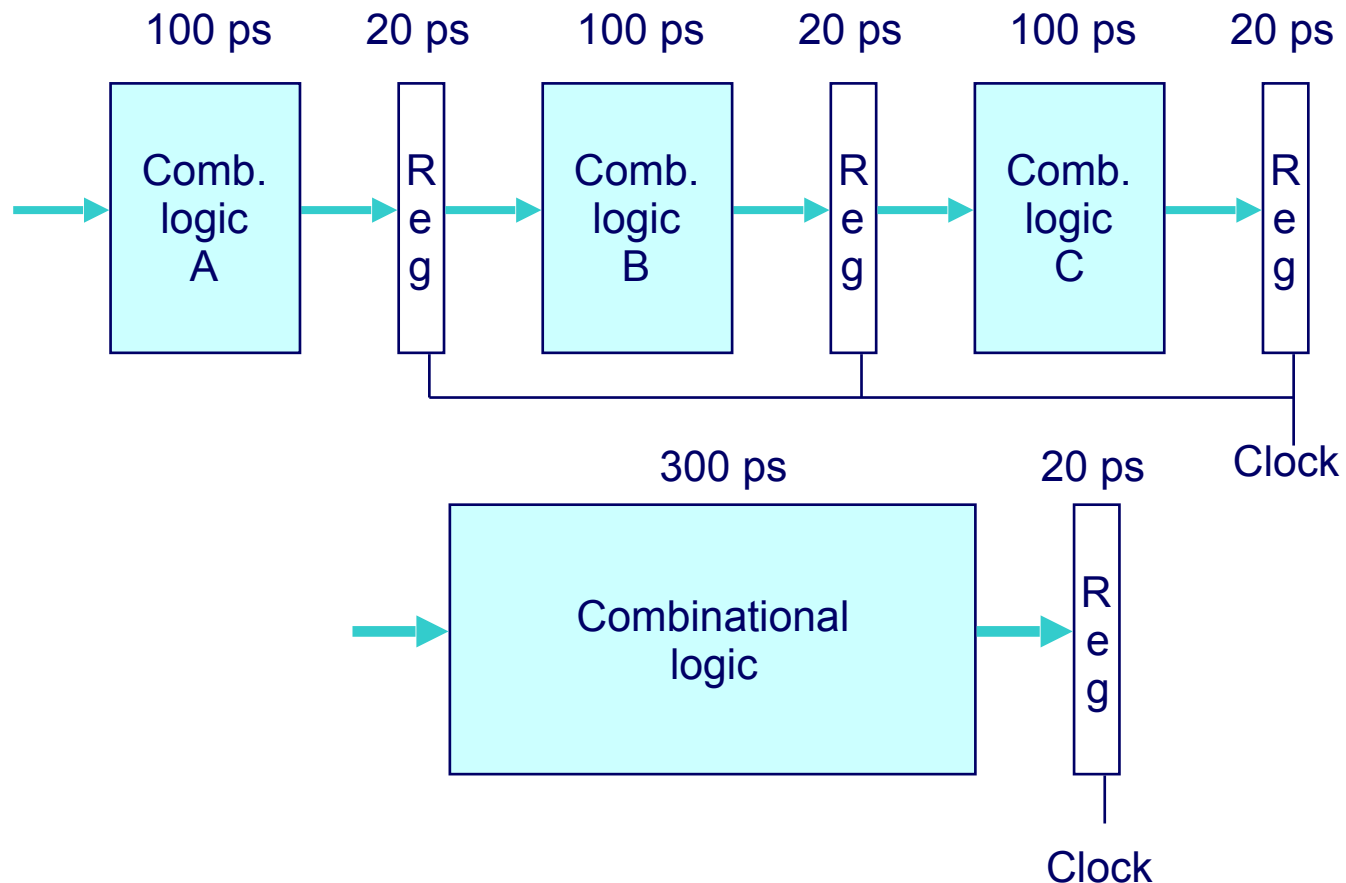
**2. CPI reduces from 1 to 1/3 (i.e., executing 3 instruction in one cycle)**



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

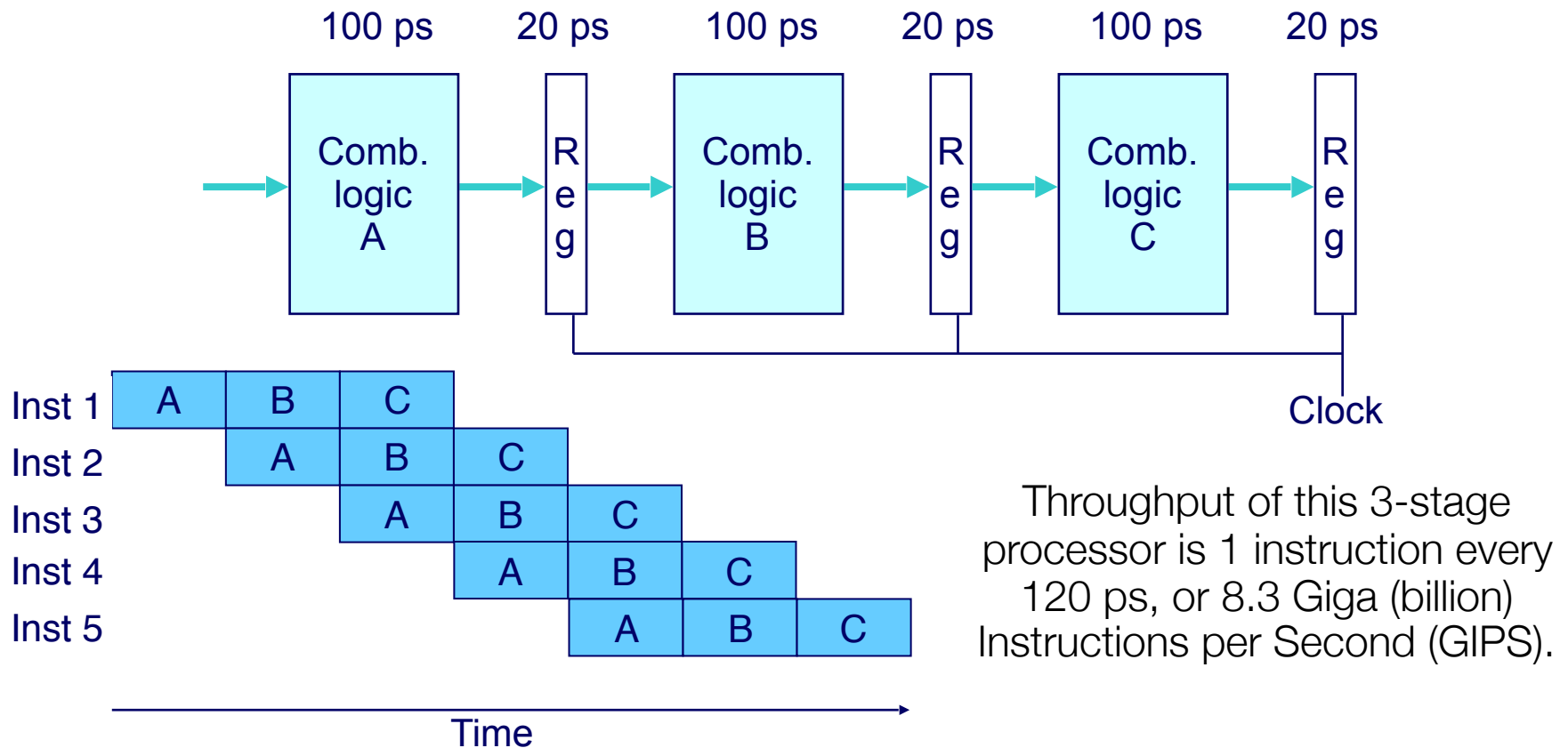
# Pipeline Trade-offs

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



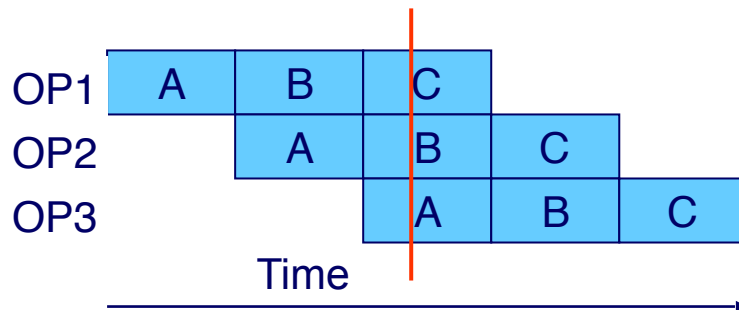
# Throughput

- 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