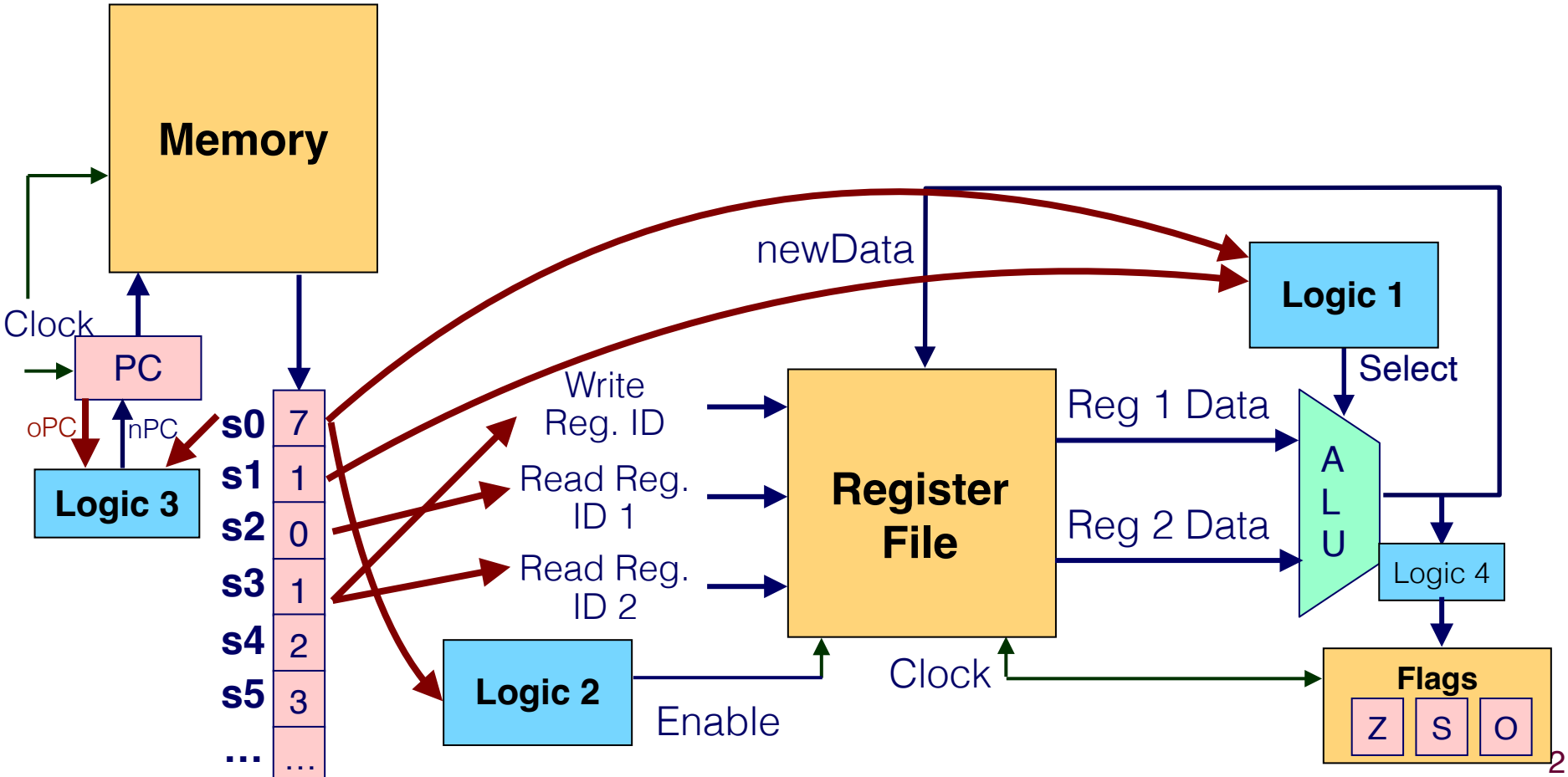
CSC 252: Computer Organization
Spring 2024: Lecture 14

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Executing a JLE instruction

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

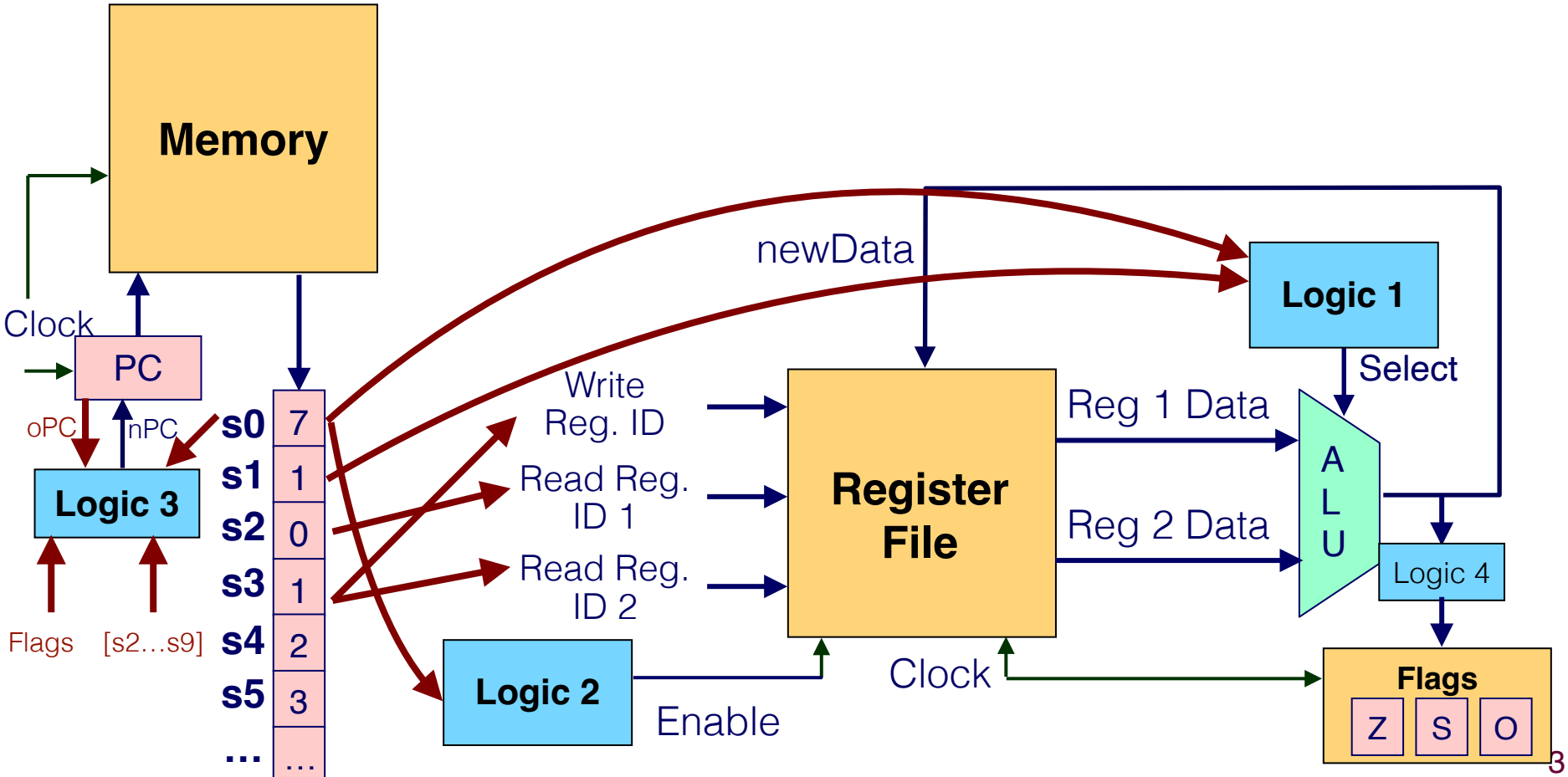


jle Dest	7	1	Dest

Executing a JLE instruction

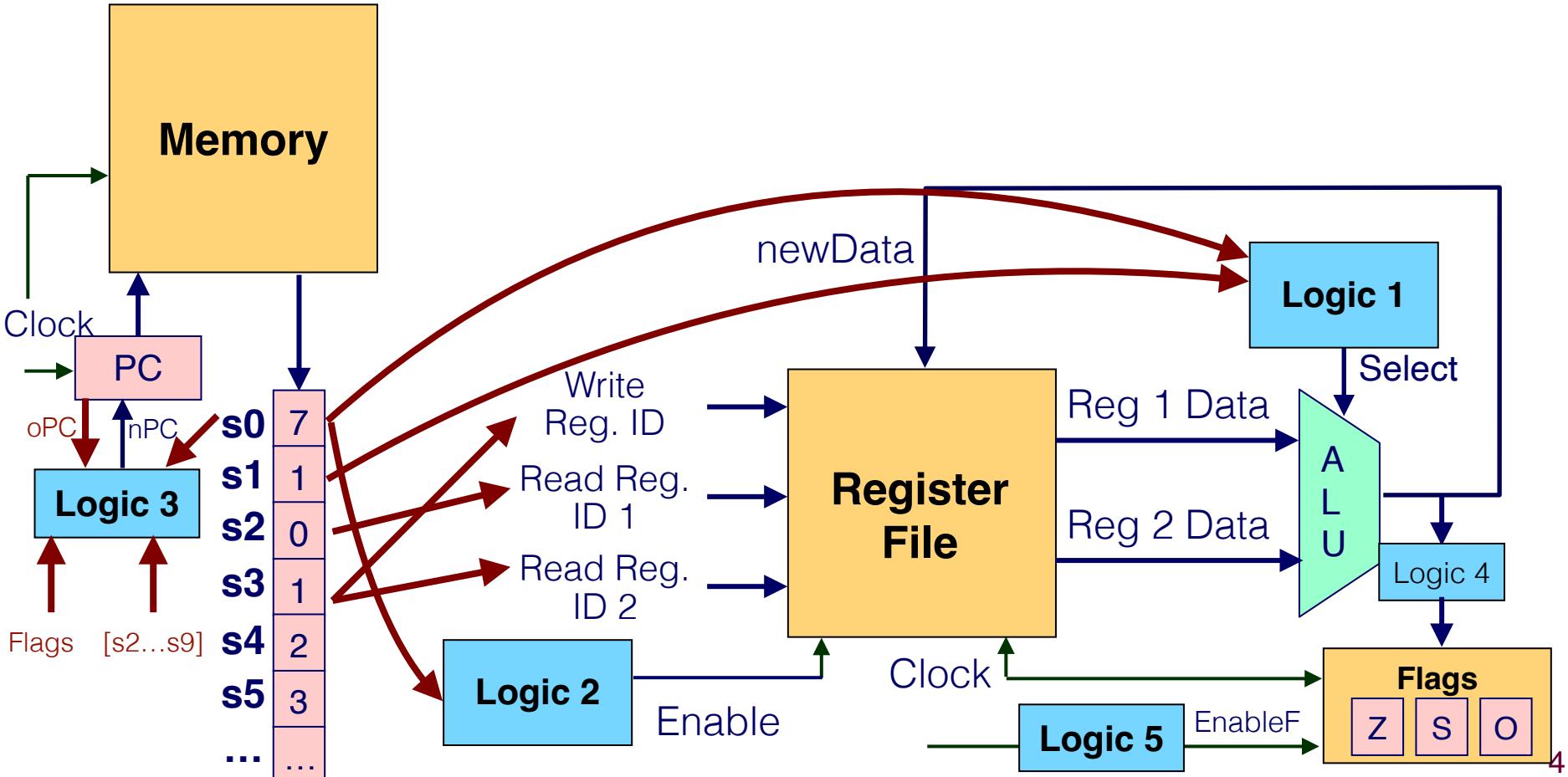
- Logic 3??

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

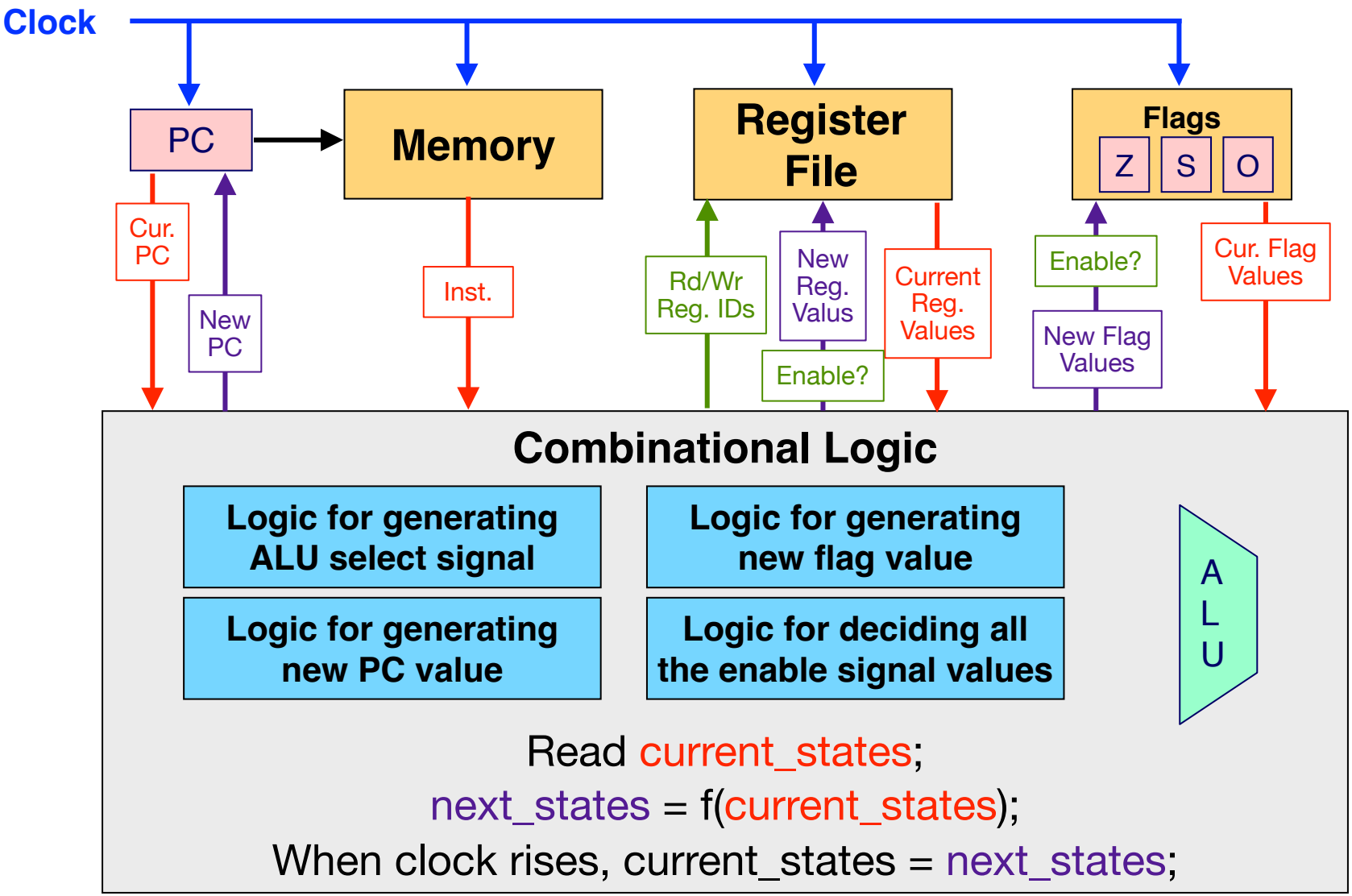


Executing a JLE instruction

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

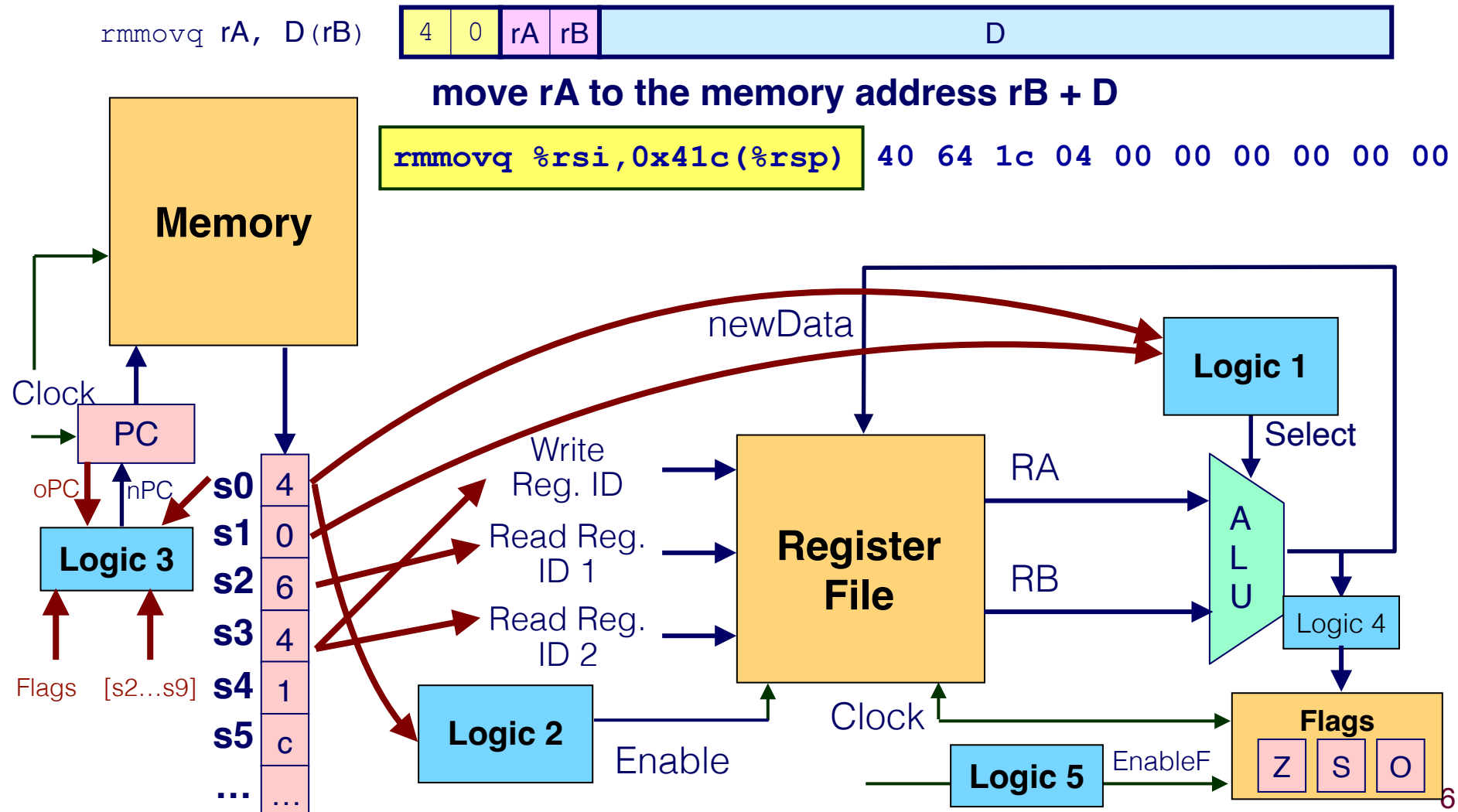


Microarchitecture (So far)



Executing a MOV instruction

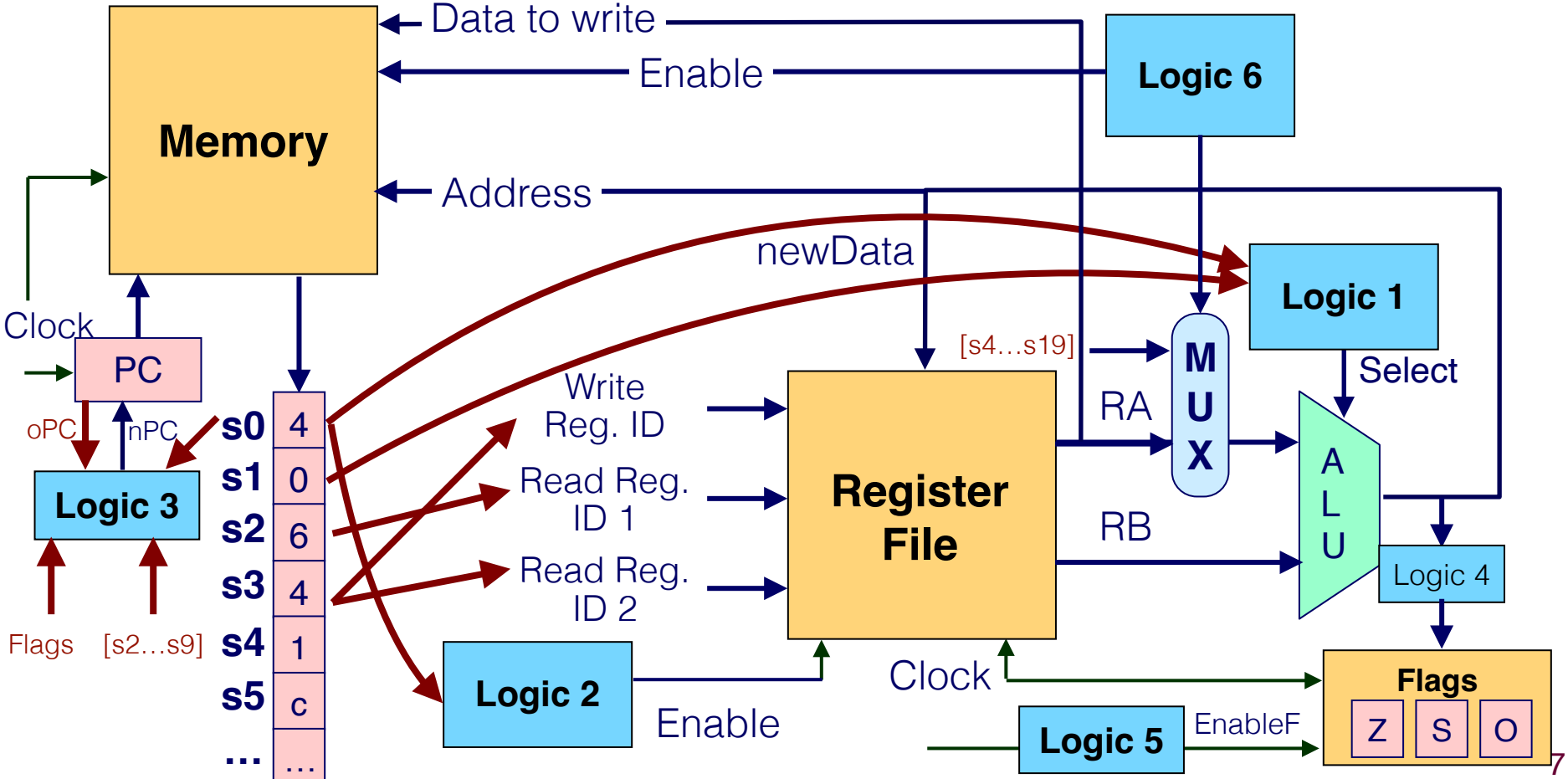
- How do we modify the hardware to execute a move instruction?



move rA to the memory address rB + D

`rmmovq rA, D(rB)`

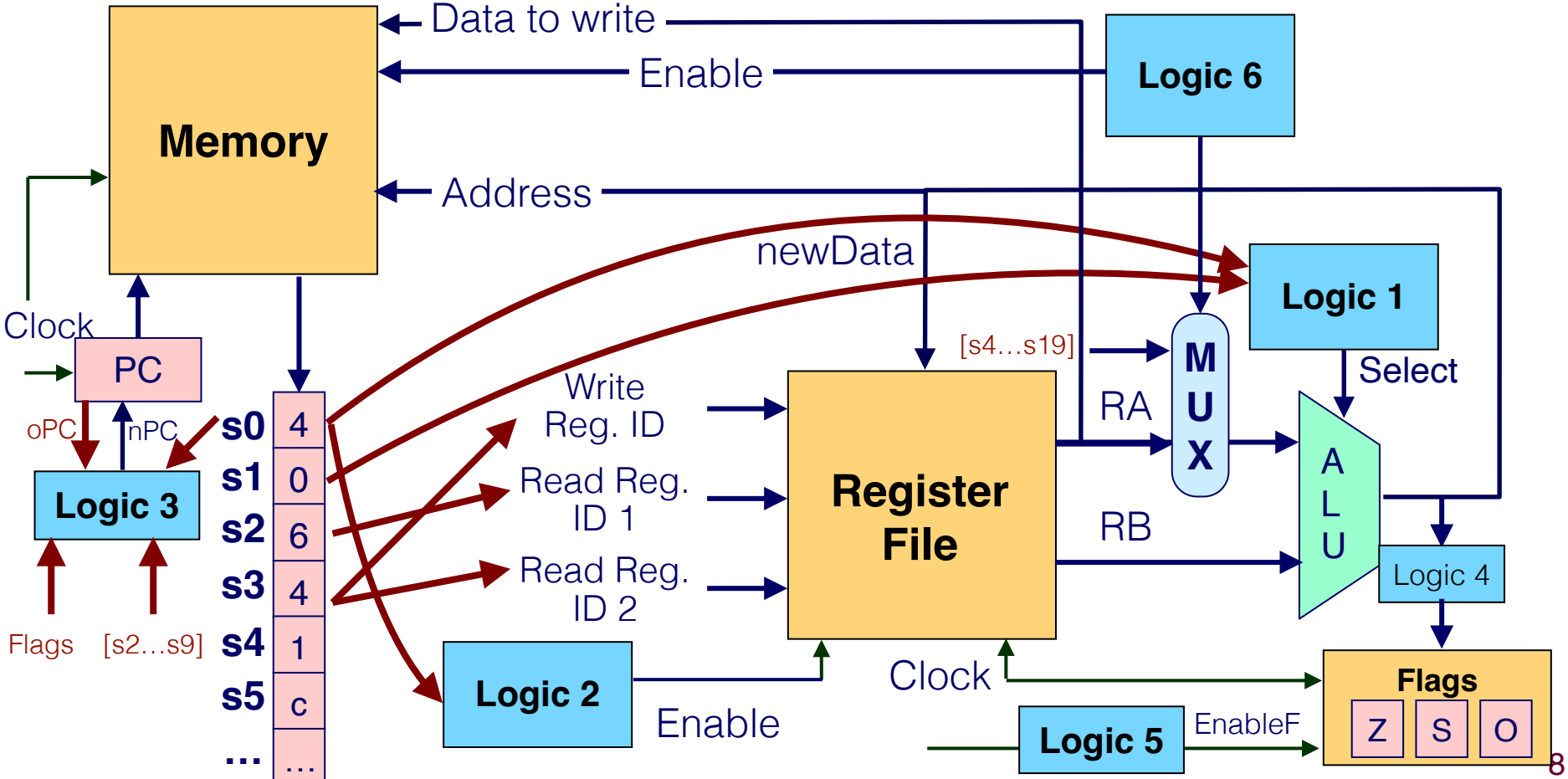
- Need new logic (Logic 6) to select the input to the ALU for Enable.
- How about other logics?



How About Memory to Register MOV?

move data at memory address rB + D to rA

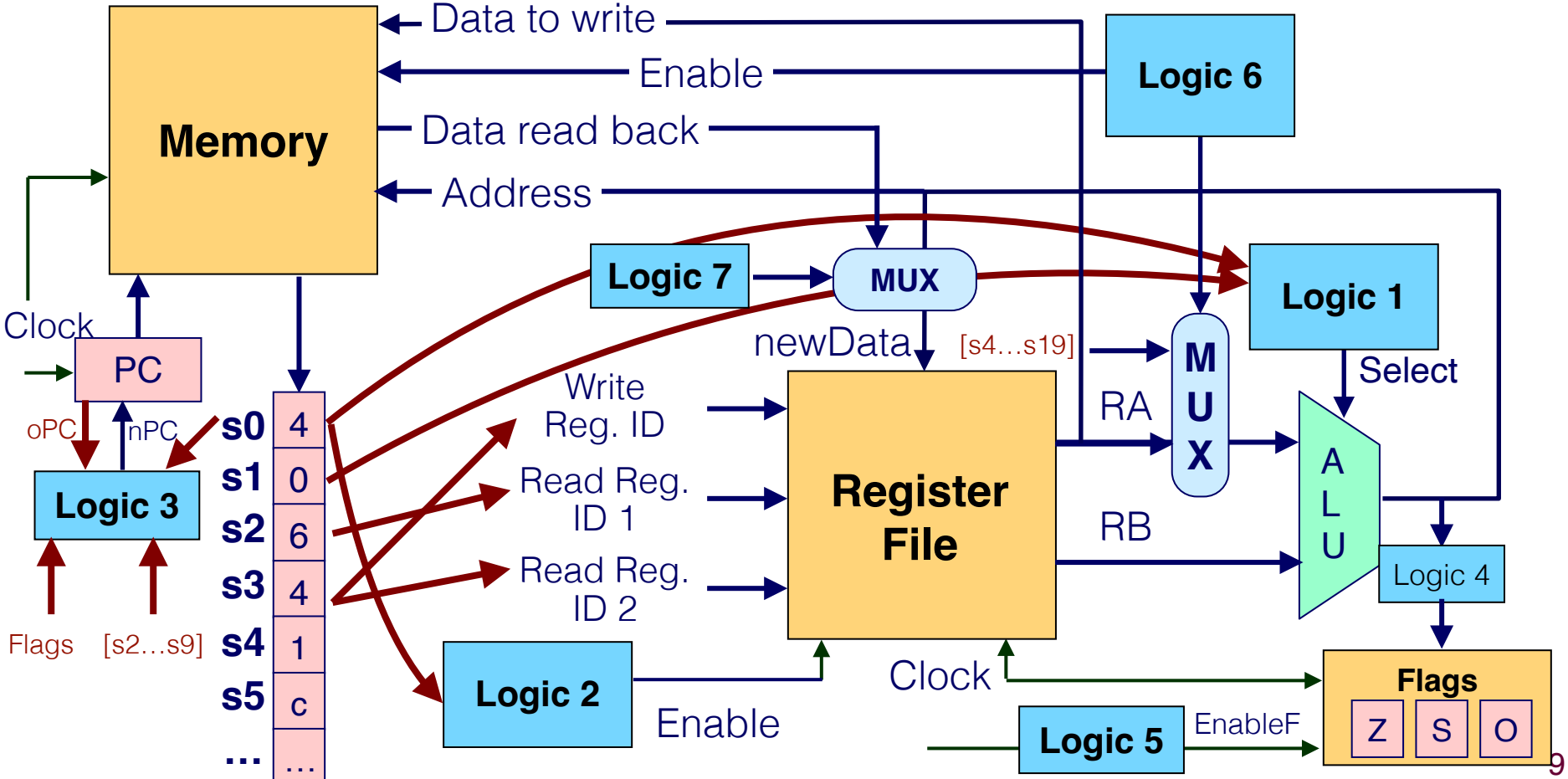
`mrmovq D(rB), rA`



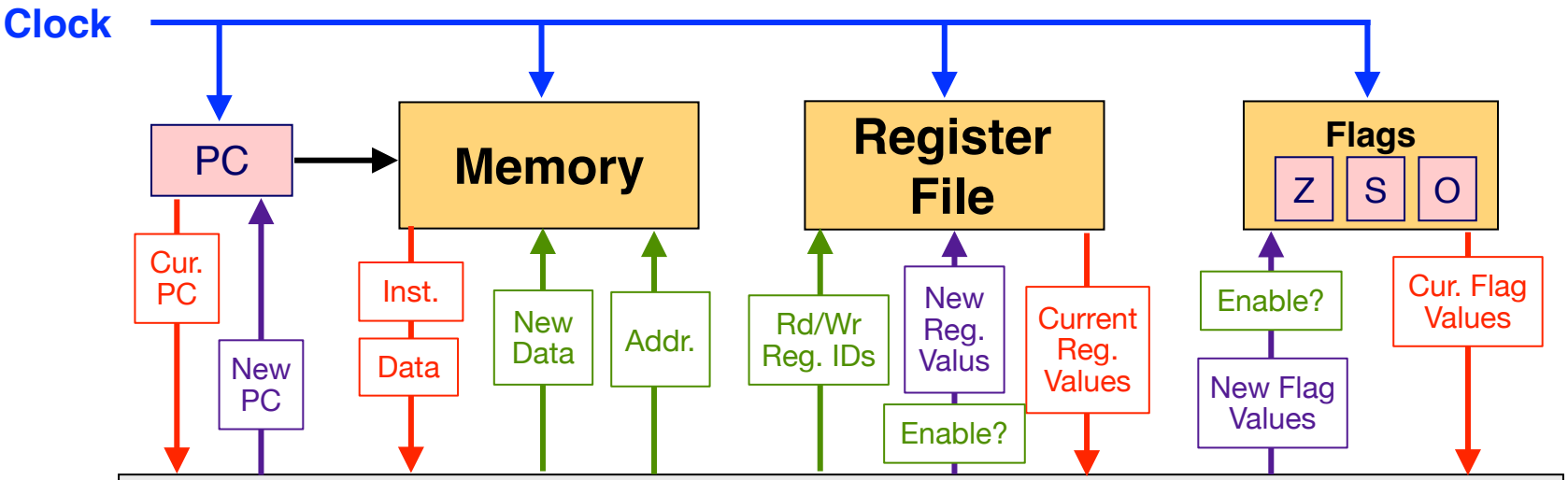
How About Memory to Register MOV?

move data at memory address rB + D to rA

`mrmovq D(rB), rA`



Microarchitecture (with MOV)



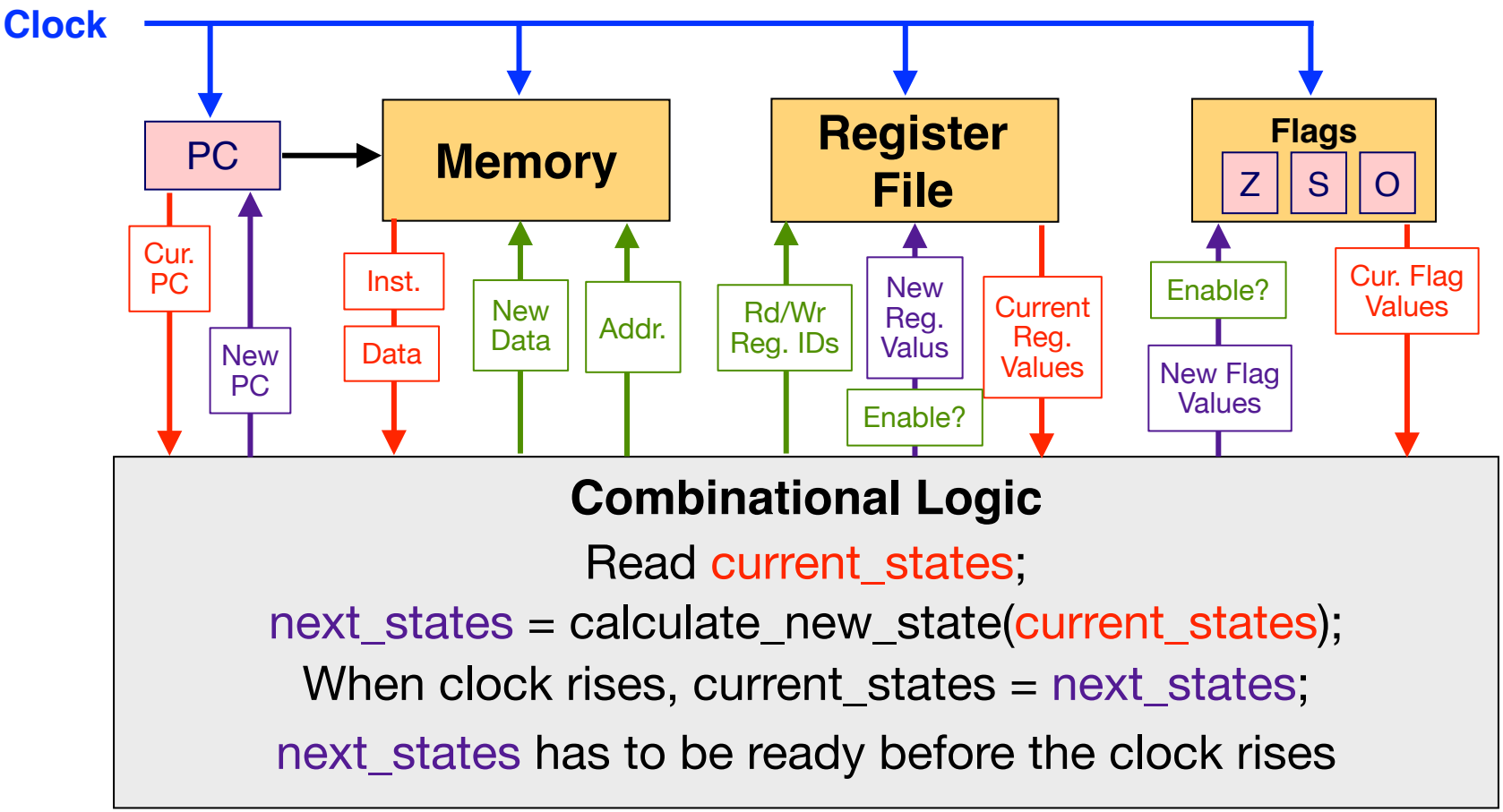
Read current_states;

next_states = f(current_states);

When clock rises, current_states = next_states;

next_states has to be ready before the clock rises

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.

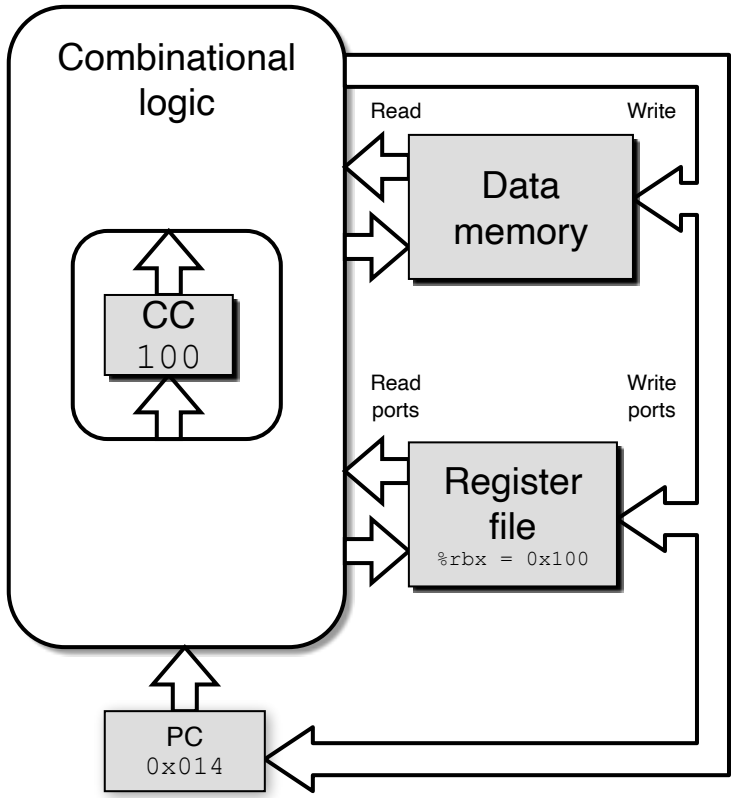
Single-Cycle Microarchitecture: Illustration

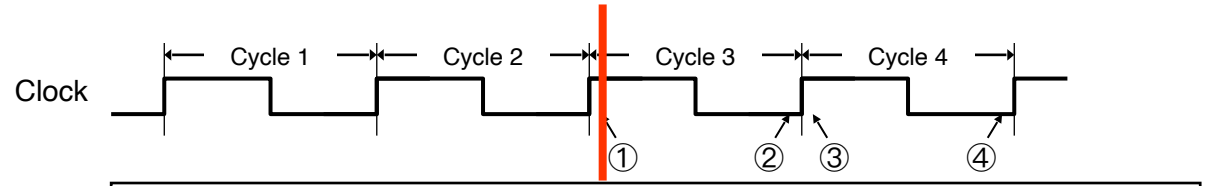
Think of it as a state machine

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

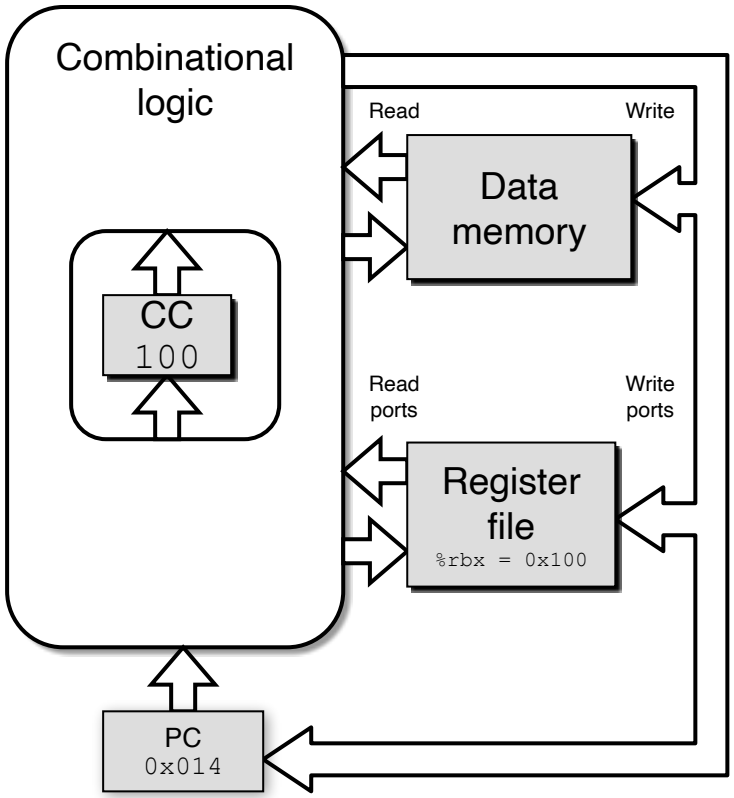
States (All updated as clock rises)

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

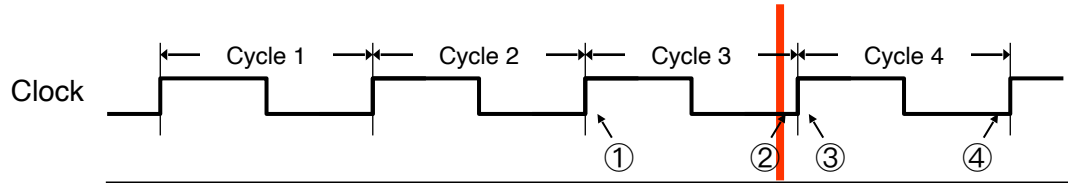




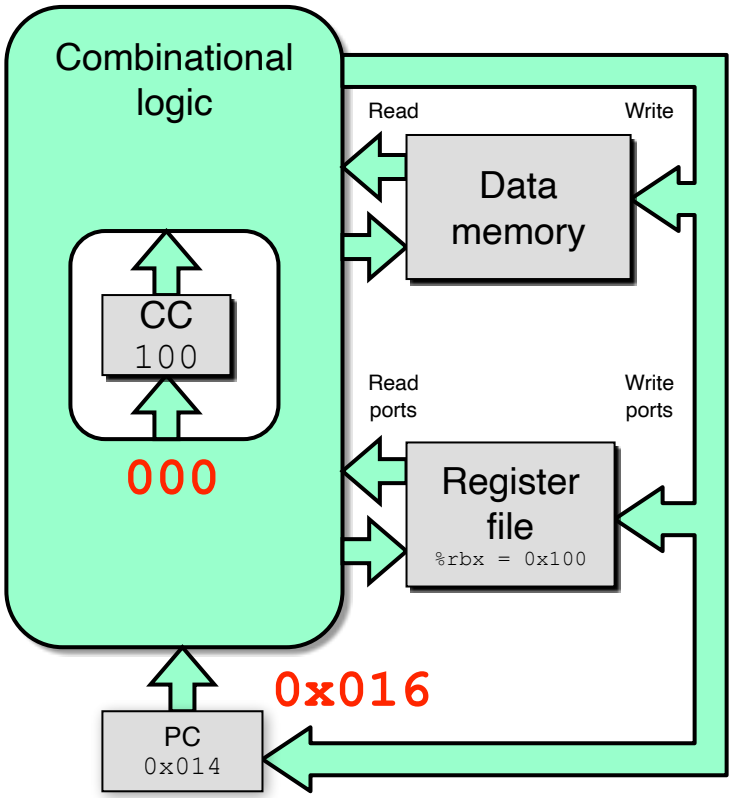
Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



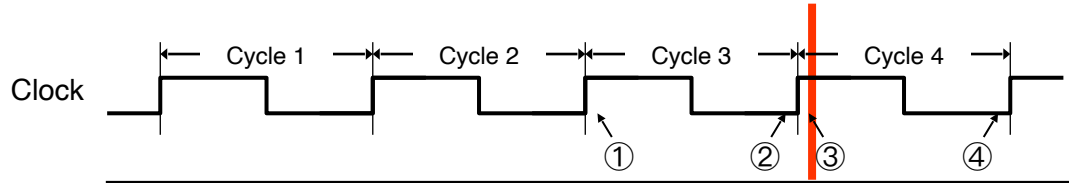
- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes



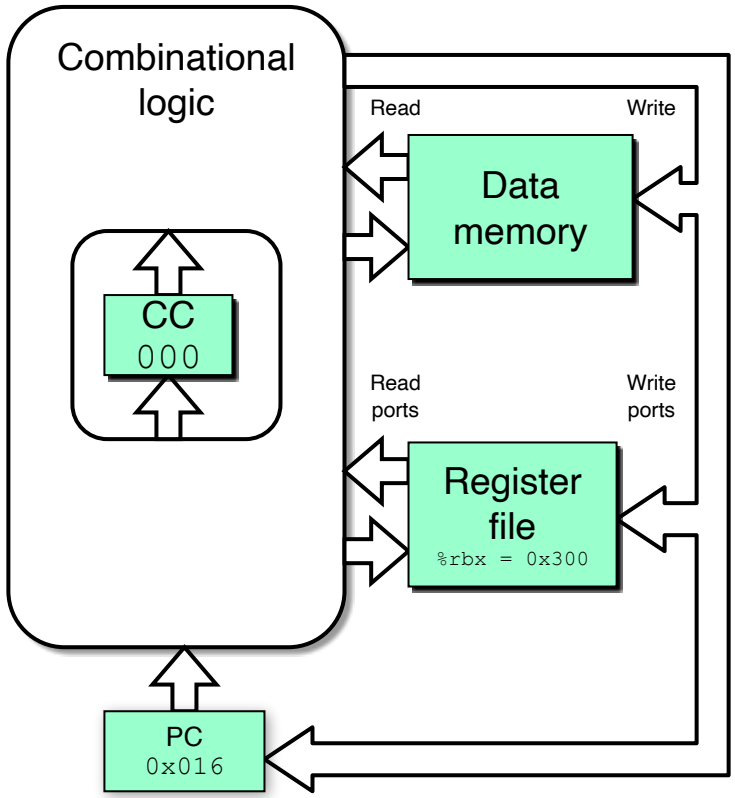
Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



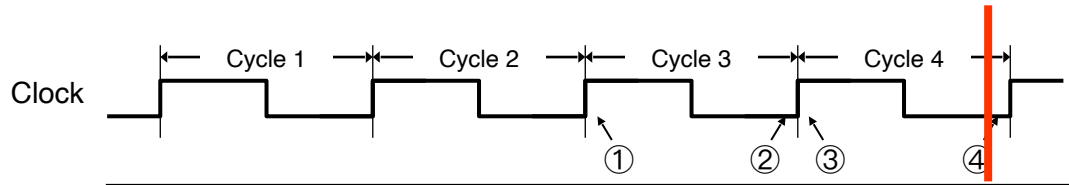
- state set according to second `irmovq` instruction
- combinational logic generates results for addq instruction



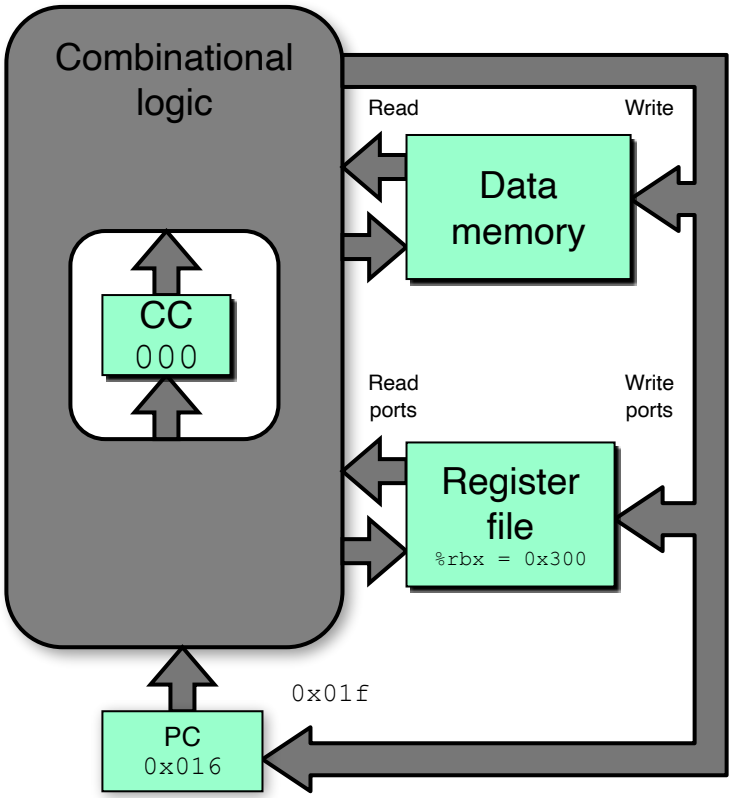
Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



- 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

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)

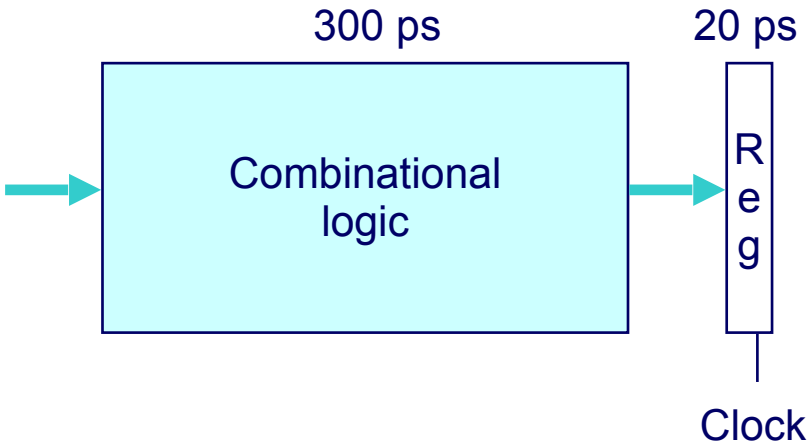
/ number of cycles per second

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

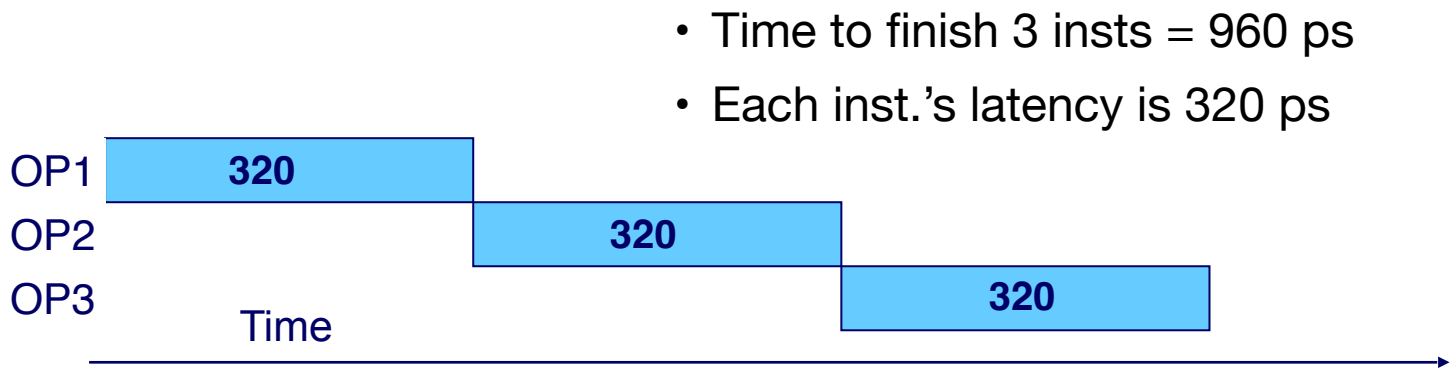
- Cycle time
  - Every instruction finishes in one cycle.
  - The absolute time takes to execute each instruction varies. Consider for instance an ADD instruction and a JMP instruction.
  - But the cycle time is uniform across instructions, so the cycle time needs to accommodate the worst case, i.e.,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



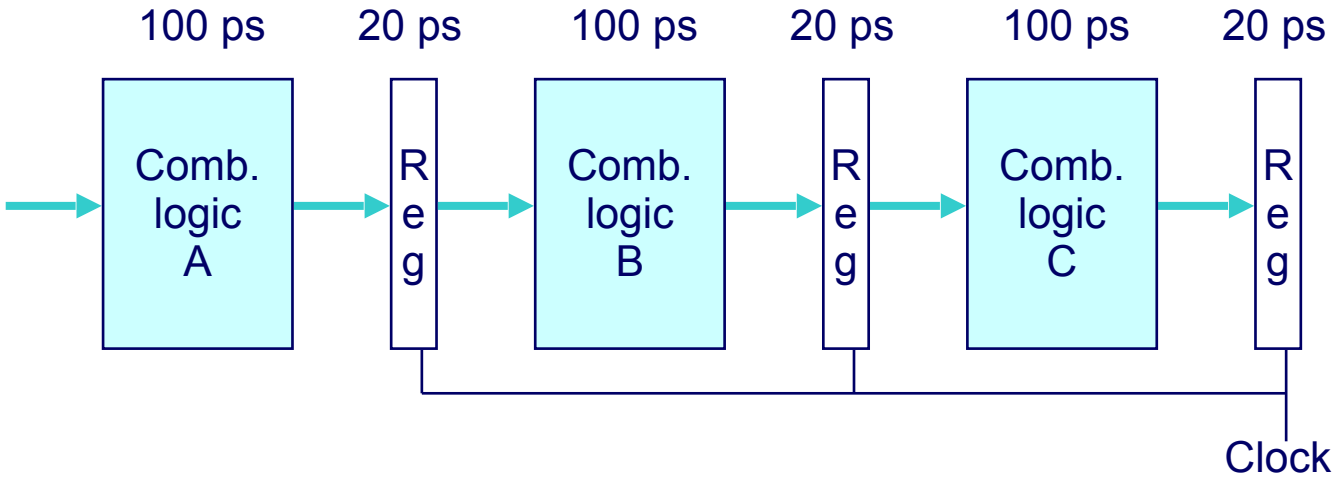
- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle time of at least 320 ps

Pipeline Diagrams



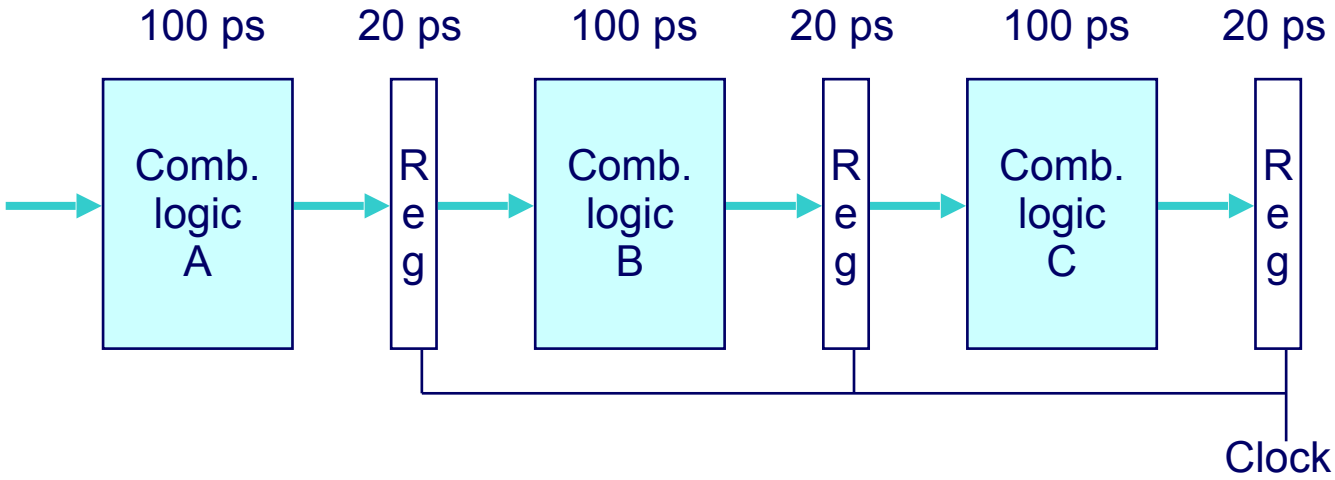
- 3 instructions will take 960 ps to finish
  - First cycle: Inst 1 takes 300 ps to compute new state, 20 ps to store the new states
  - Second cycle: Inst 2 starts; it takes 300 ps to compute new states, 20 ps to store new states
  - And so on…

3-Stage Pipelined Version

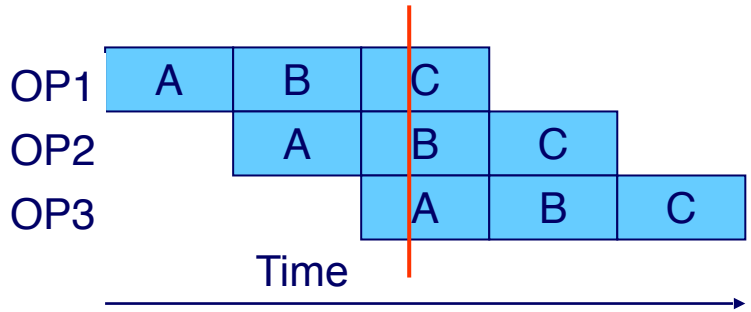


- Divide combinational logic into 3 stages of 100 ps each
- Insert registers between stages to store intermediate data between stages. These are call pipeline registers (ISA-invisible)
- Can begin a new instruction as soon as the previous one finishes stage A and has stored the intermediate data.
  -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

3-Stage Pipelined Version



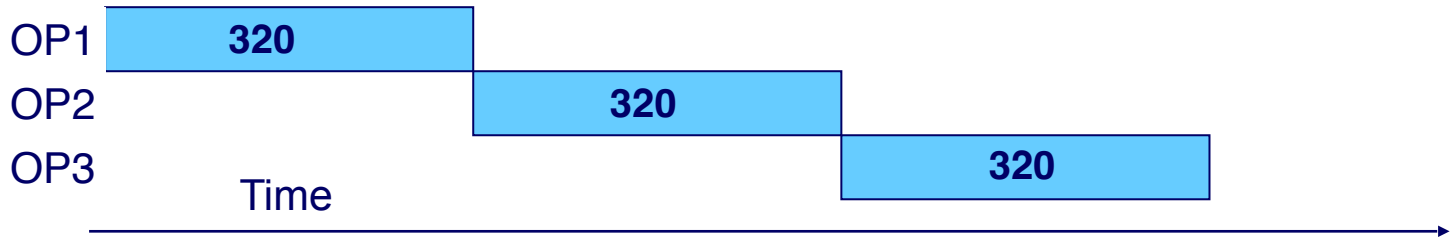
3-Stage Pipelined



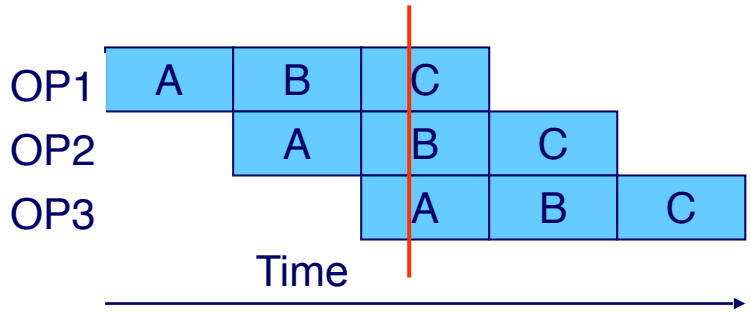
Comparison

Unpipelined

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



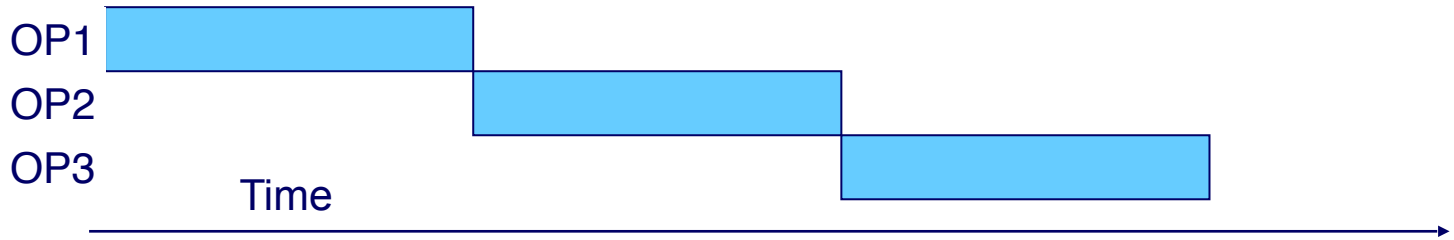
3-Stage Pipelined



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

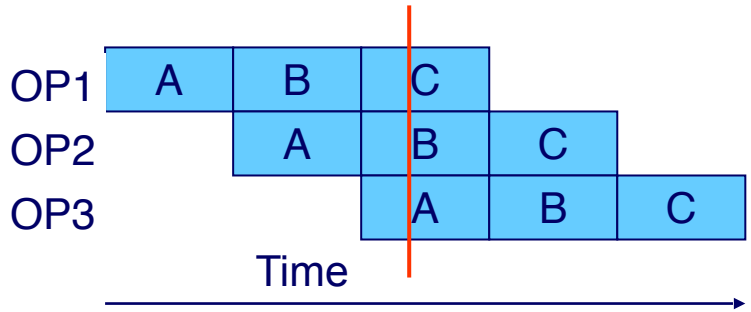
Benefits of Pipelining

- 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

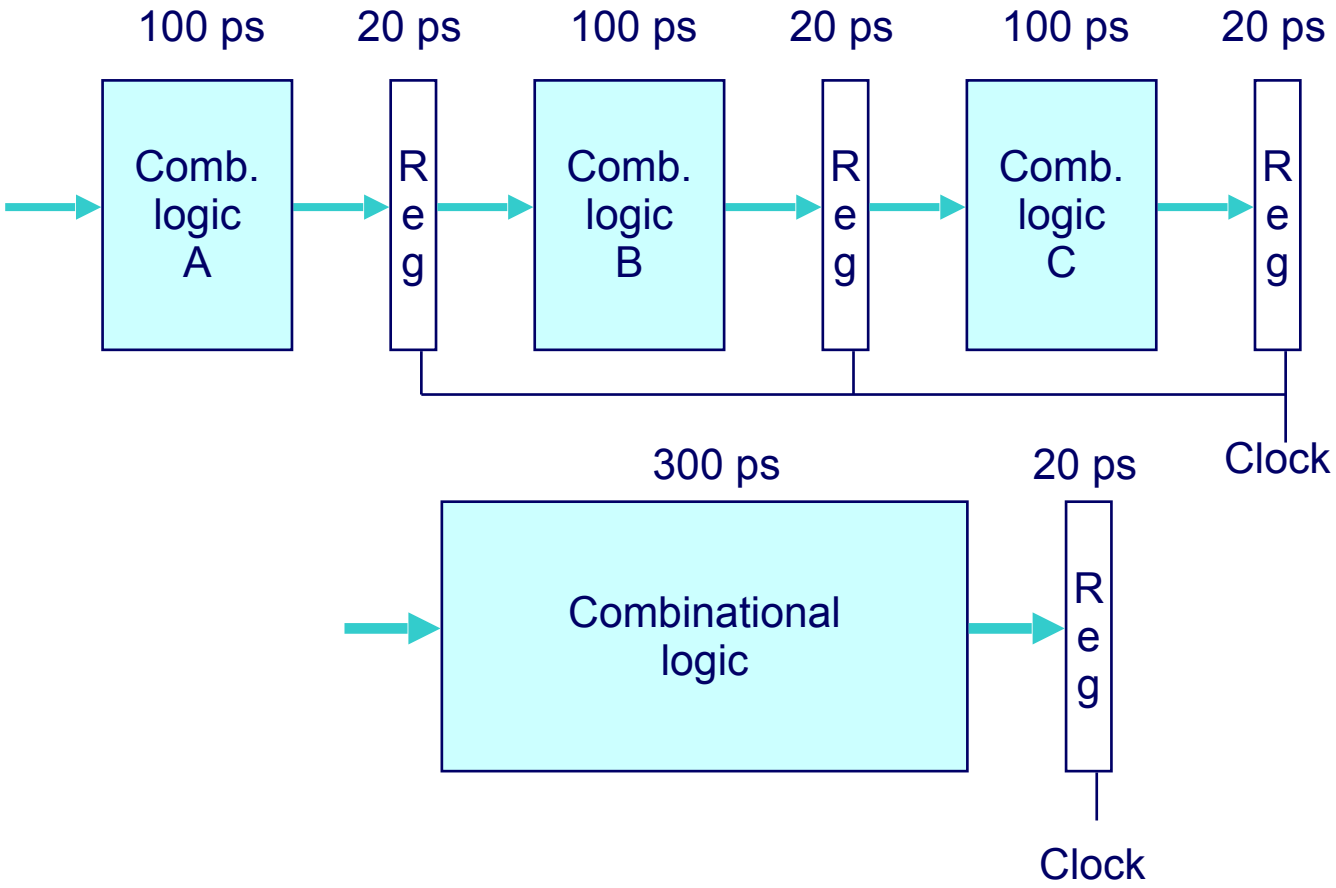
2. CPI reduces from 1 to 1/3 (i.e., executing 3 instruction in one cycle)



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

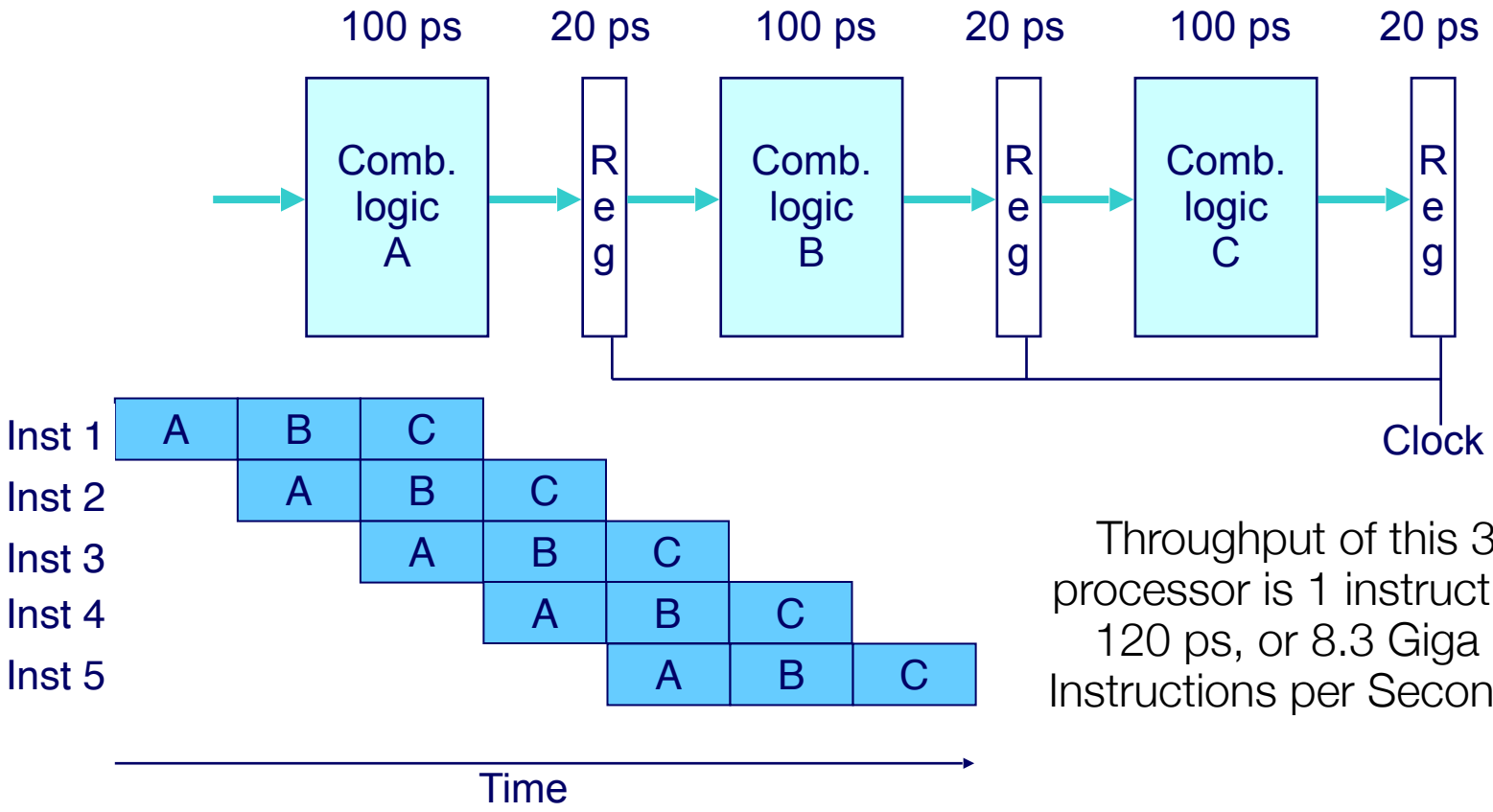
Pipeline Trade-offs

- 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

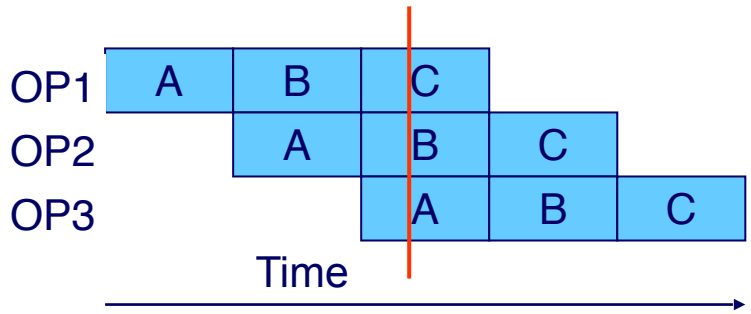
- The rate at which the processor can finish executing an instruction (at the steady state).



Throughput of this 3-stage processor is 1 instruction every 120 ps, or 8.3 Giga (billion) Instructions per Second (GIPS).

One Requirement of Pipelining

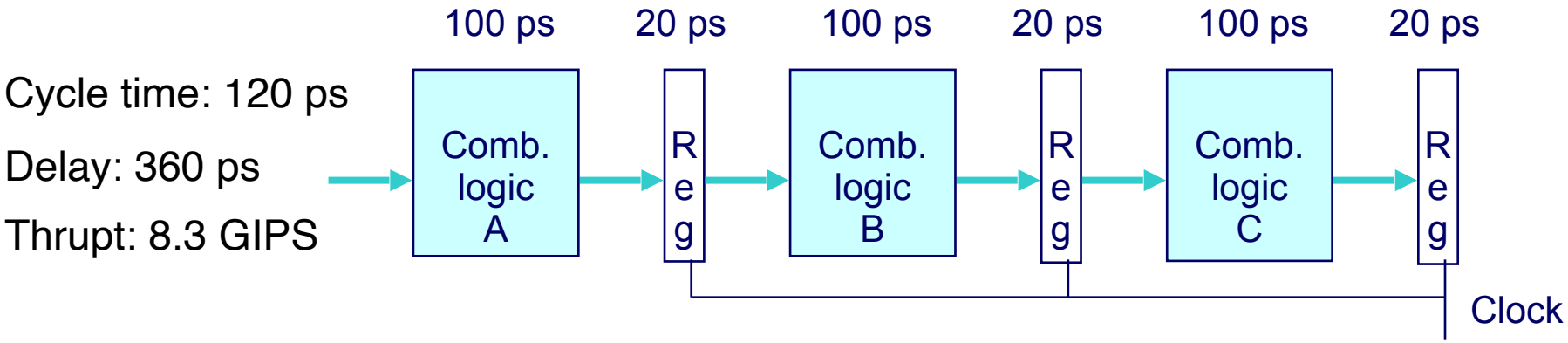
- The stages need to be using different hardware structures.
- That is, Stage A, Stage B, and Stage C need to exercise different parts of the combination logic.

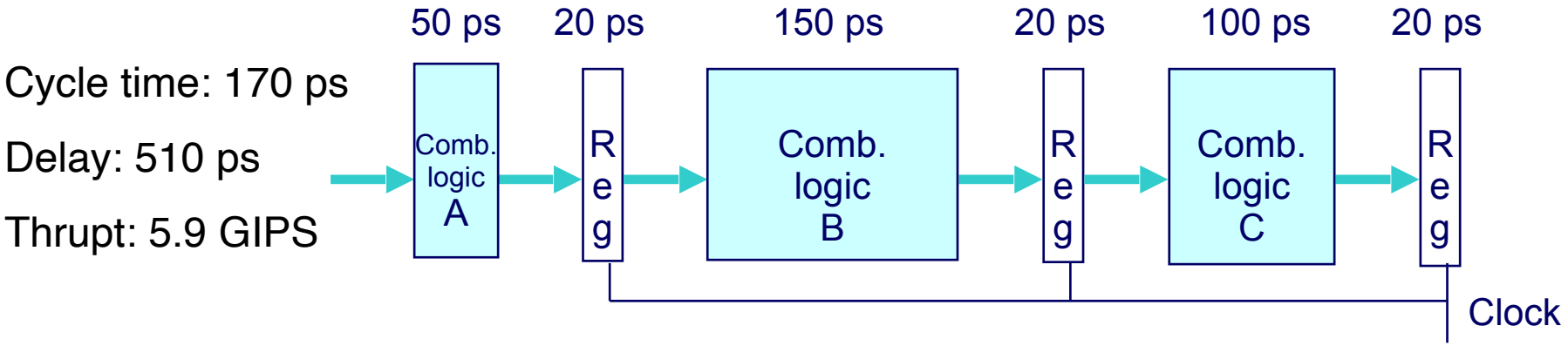


- Time to finish 3 insets = 120 * 5 = 600 ps
- But each inst.'s latency increases: 120 * 3 = 360 ps

Aside: Unbalanced Pipeline

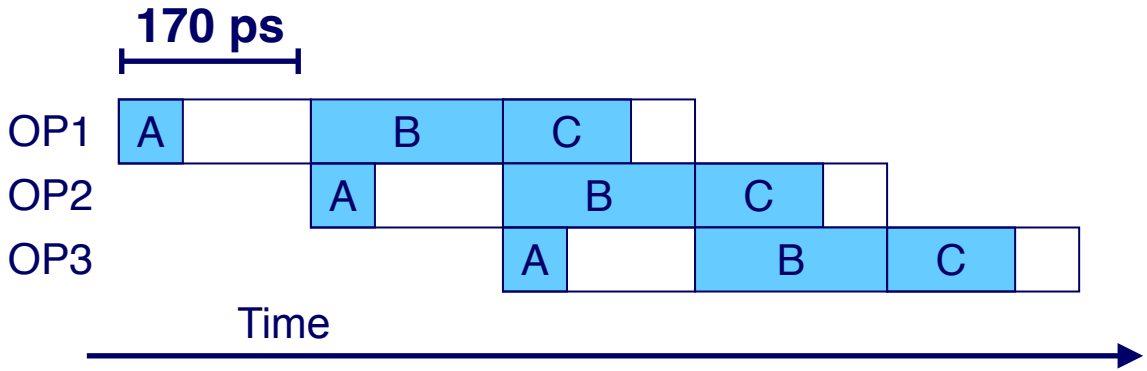
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

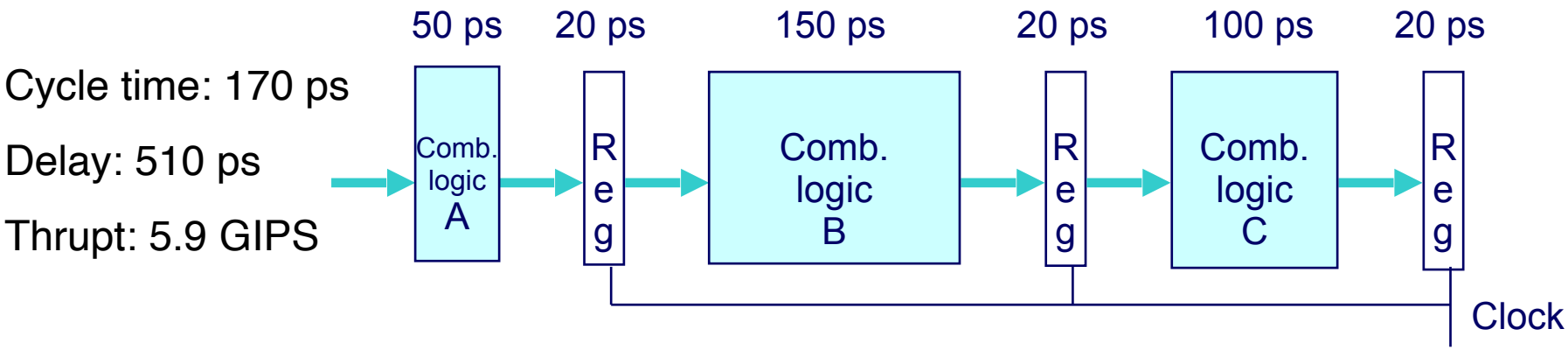




Aside: Unbalanced Pipeline

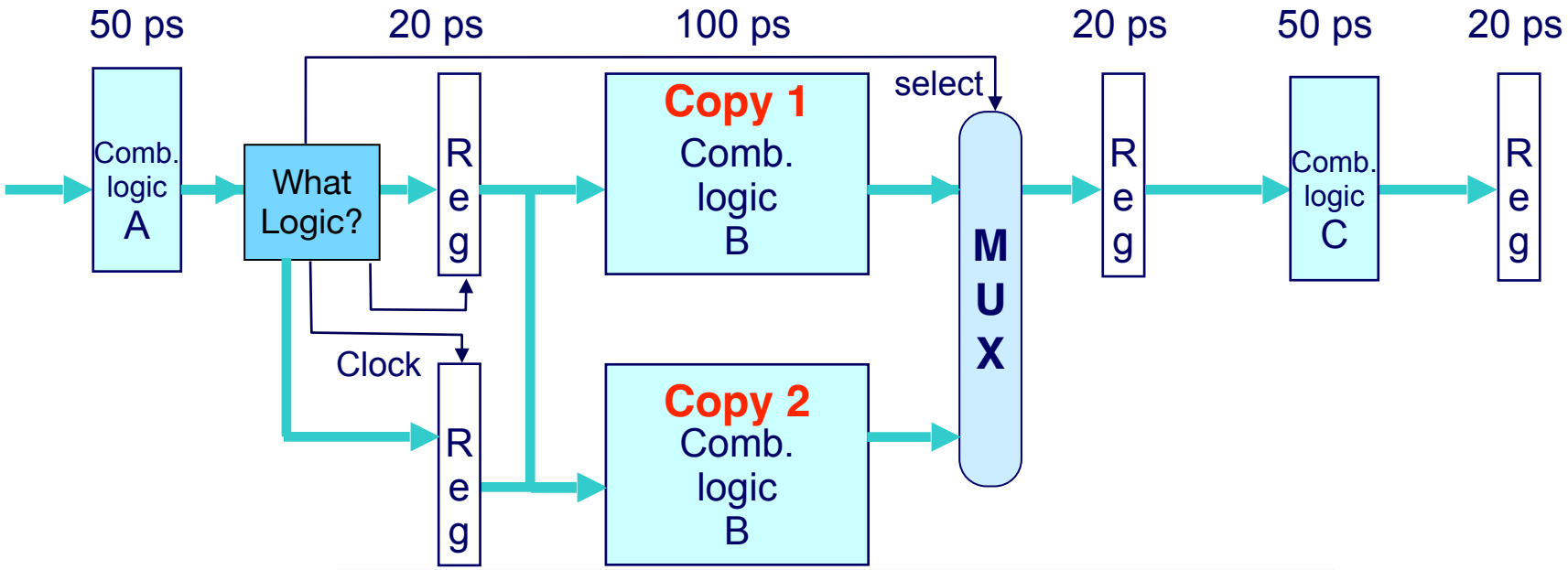
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput





Aside: 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

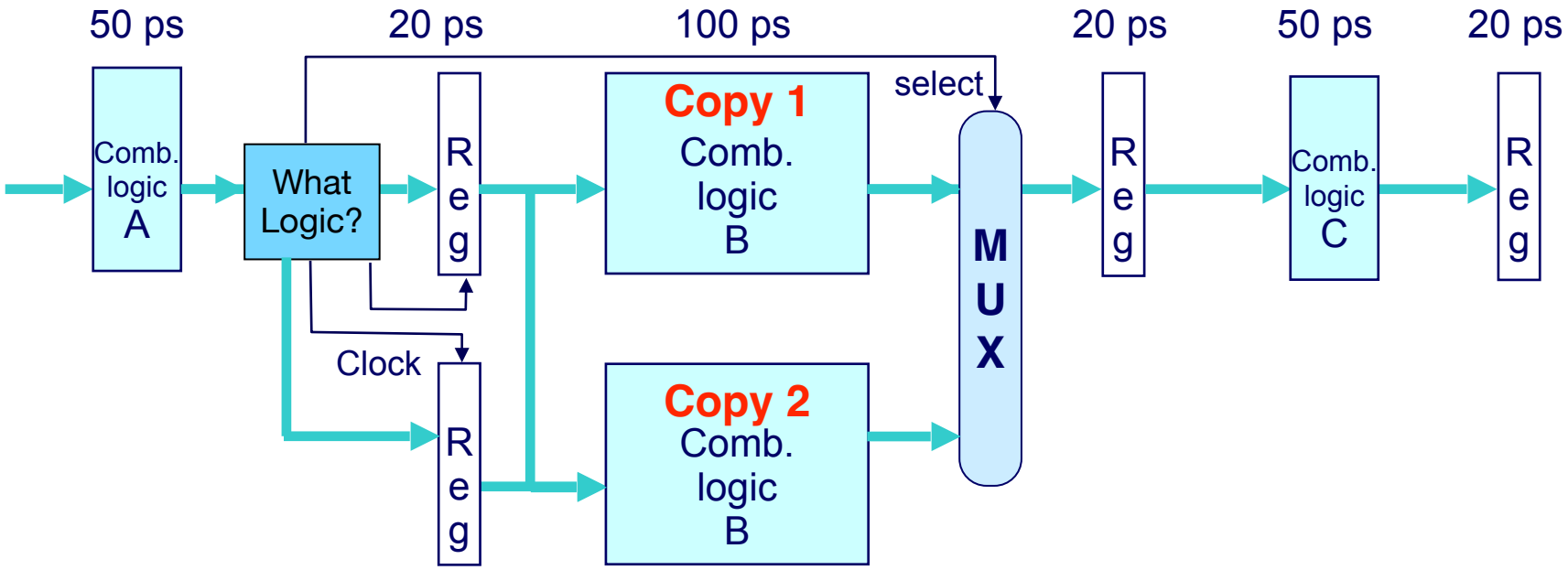


What logic do you need there?

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.
- 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

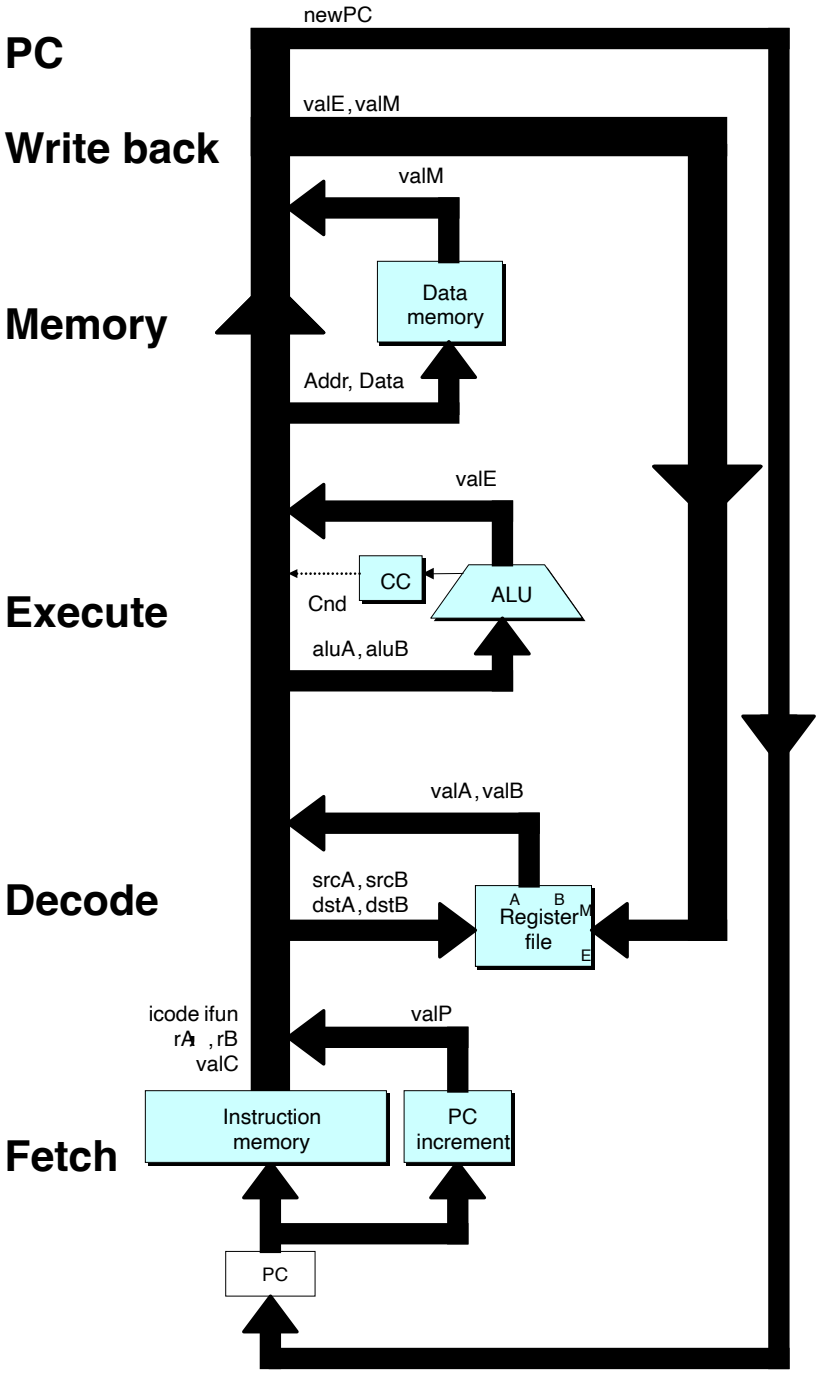
Decode: Read program registers

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

	OPq rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation 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)	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	Read instruction byte Read register byte Read displacement D Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A 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 ← 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