

# **CSC 252/452: Computer Organization**

## **Fall 2024: Lecture 11**

Instructor: Yanan Guo

Department of Computer Science  
University of Rochester

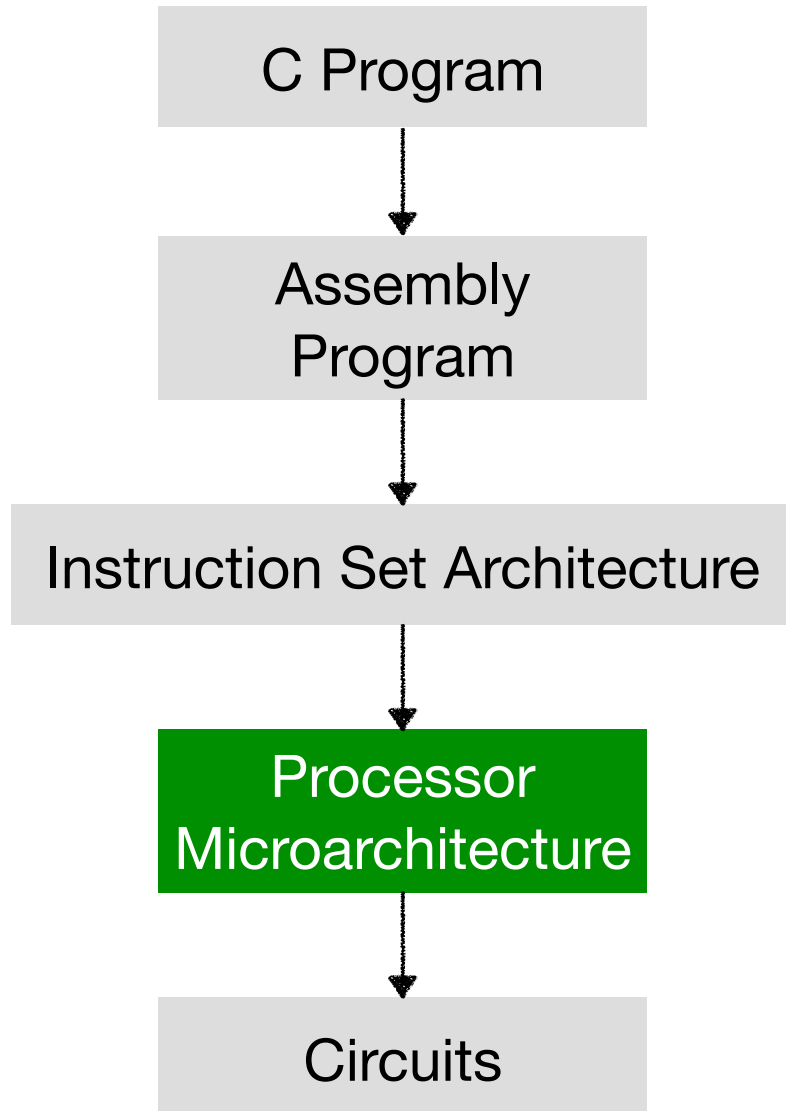
# Announcement

- Programming assignment 3 will be out
  - Details: <https://www.cs.rochester.edu/courses/252/fall2024/labs/assignment3.html>
  - Due on **Oct. 25th**, 11:59 PM
  - You (may still) have 3 slip days

# Announcement

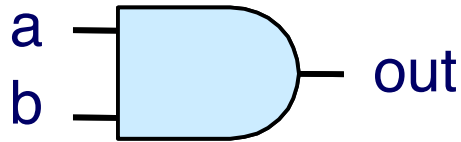
- Programming assignment 3 is in x86 assembly language. Seek help from TAs.
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.

# So far in 252...



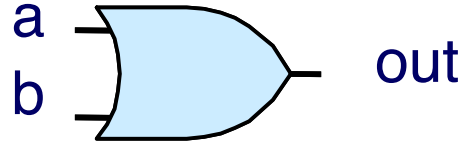
# Computing with Logic Gates

And



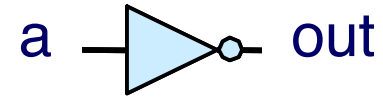
$\text{out} = a \ \&\& \ b$

Or



$\text{out} = a \ || \ b$

Not

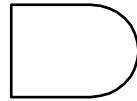


$\text{out} = !a$

# Bit Equality

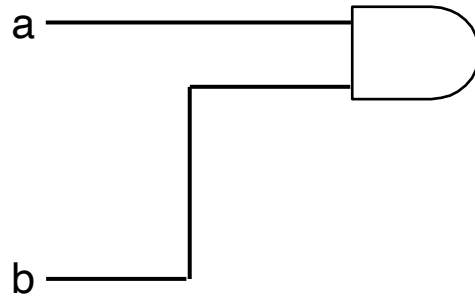
Combinational Circuit: Circuit for computation

# Bit Equality



Combinational Circuit: Circuit for computation

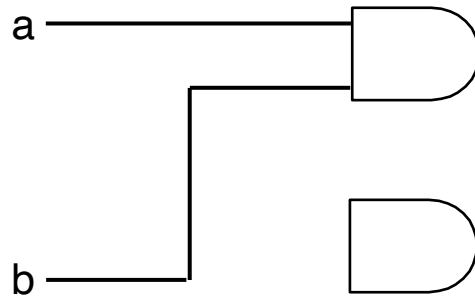
# Bit Equality



Combinational Circuit: Circuit for computation

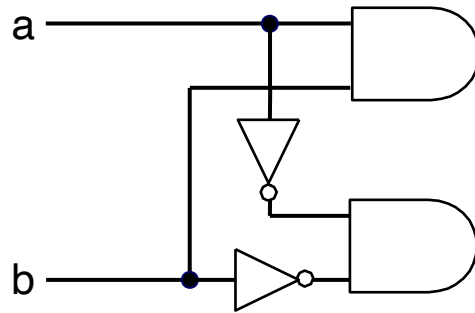


# Bit Equality



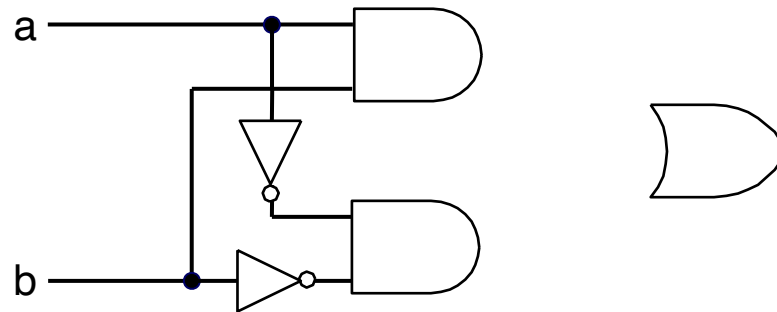
Combinational Circuit: Circuit for computation

# Bit Equality



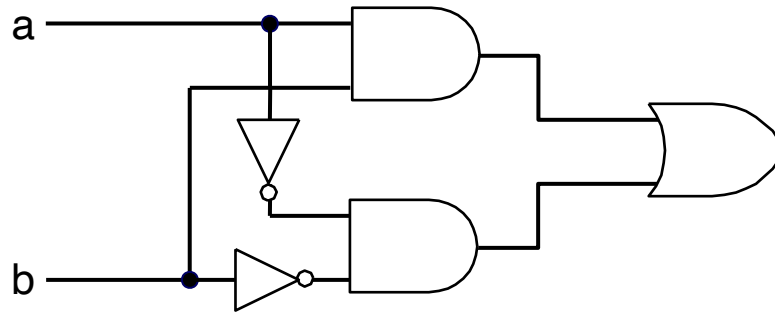
Combinational Circuit: Circuit for computation

# Bit Equality



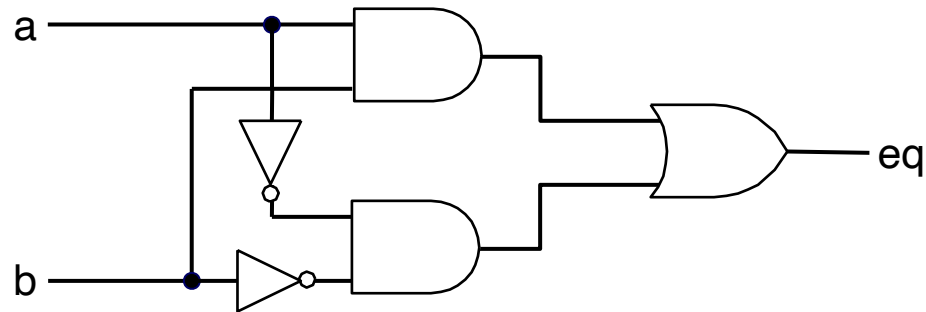
Combinational Circuit: Circuit for computation

# Bit Equality



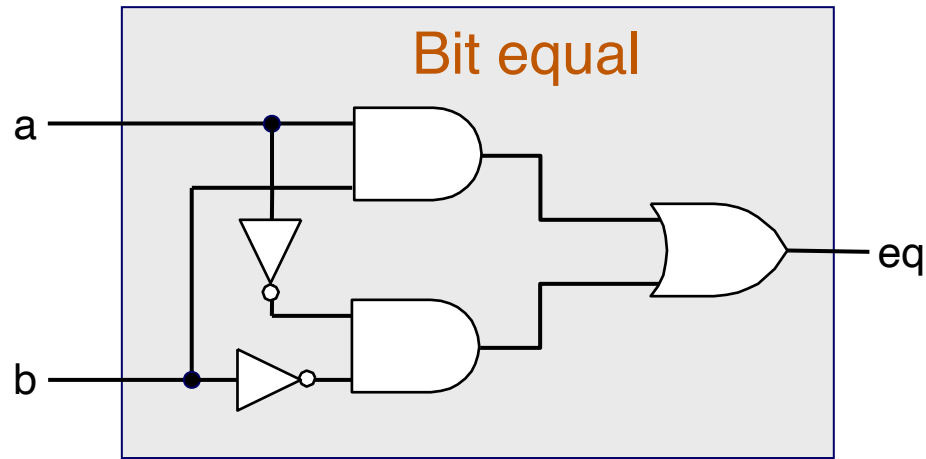
Combinational Circuit: Circuit for computation

# Bit Equality



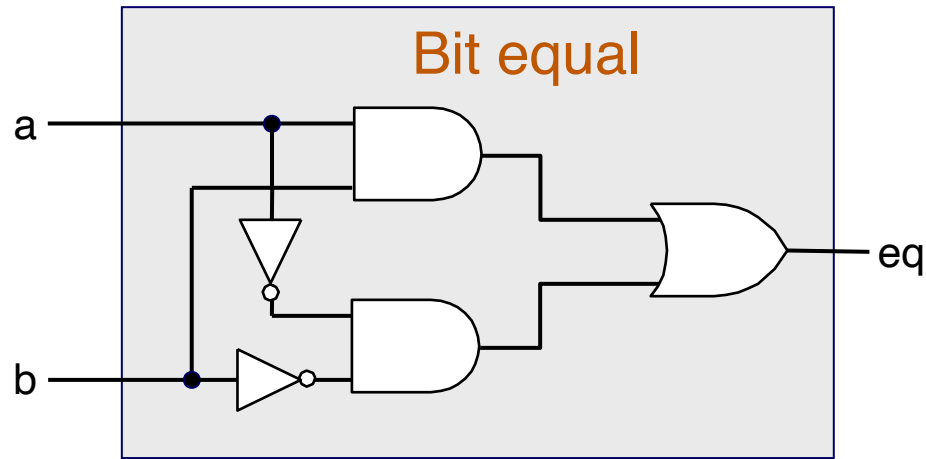
Combinational Circuit: Circuit for computation

# Bit Equality



Combinational Circuit: Circuit for computation

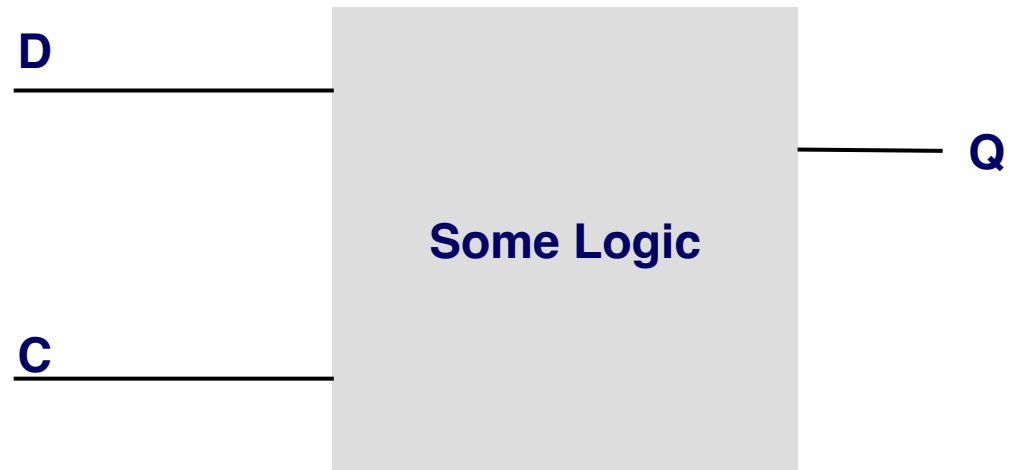
# Bit Equality



Combinational Circuit: Circuit for computation

Sequential Circuit: Circuit for storage

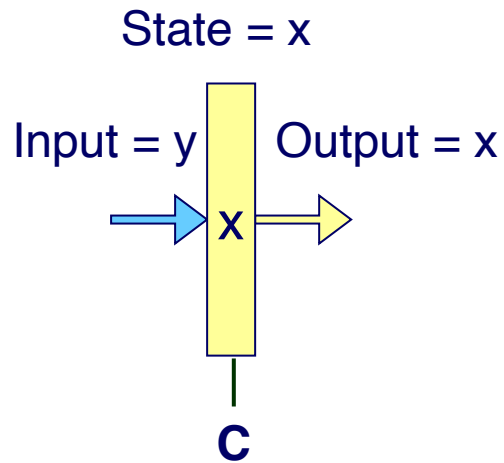
# Sequential Circuit



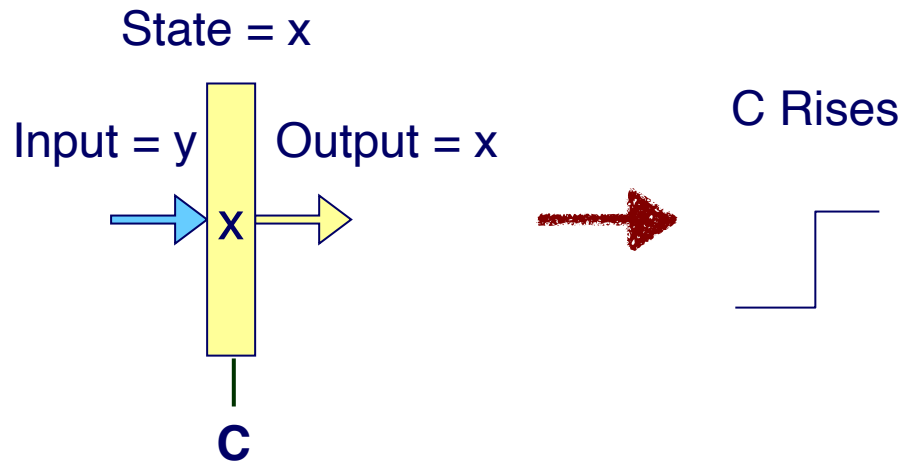
- 1-bit storage:
  - D is the data I want to store (0 or 1)
  - C is the control signal
    - When C is 1, Q becomes D (i.e., storing the data)
    - When C is 0, Q doesn't change with D (data stored)



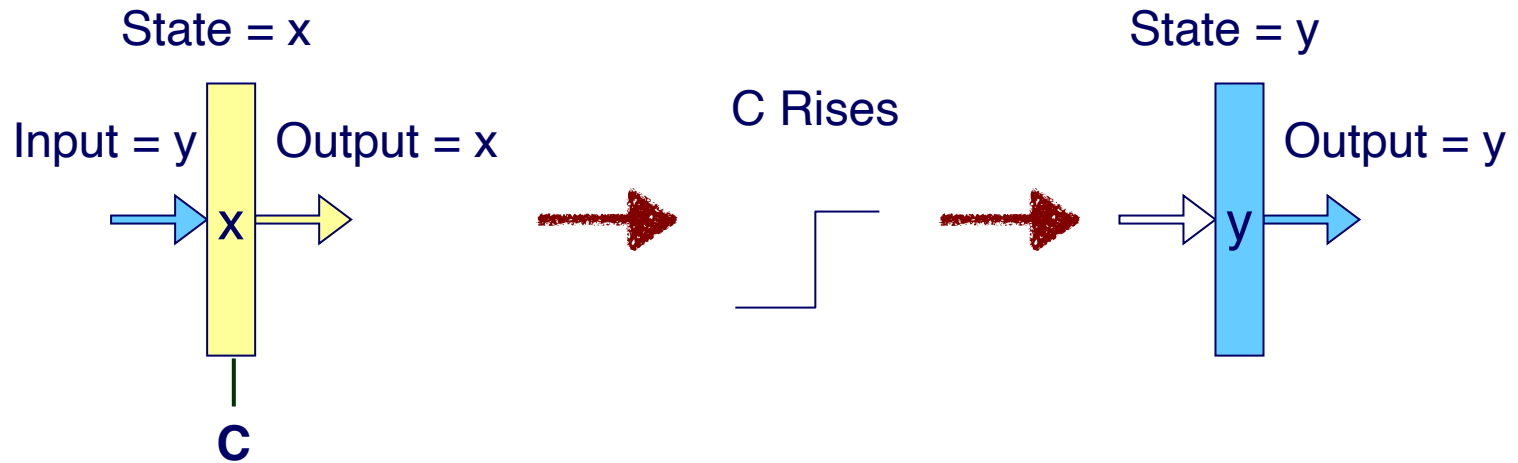
# Register Operation



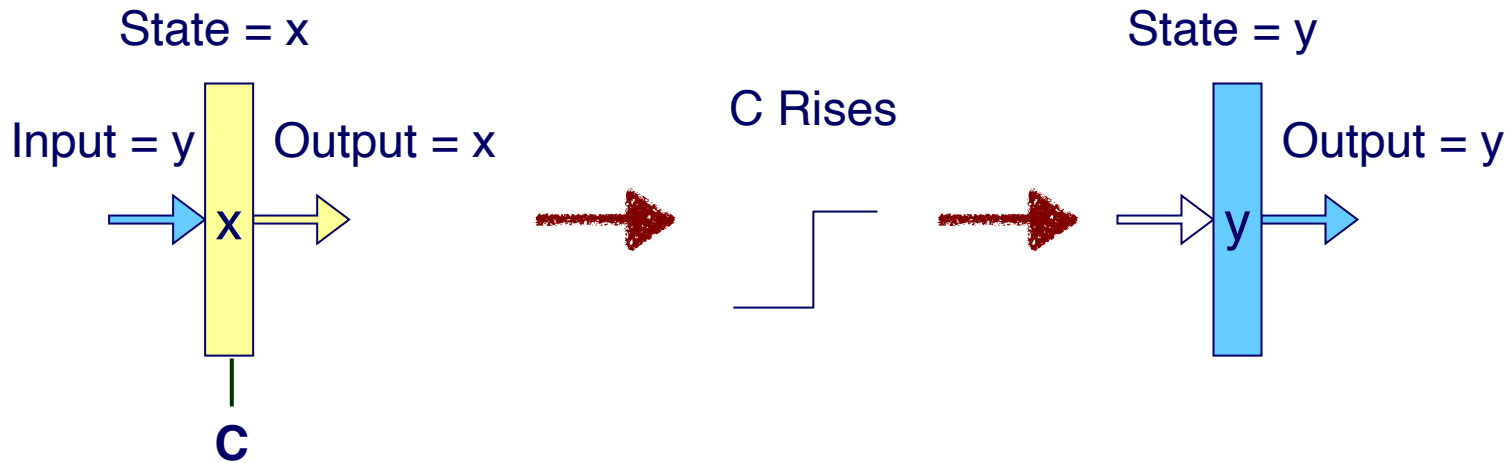
# Register Operation



# Register Operation

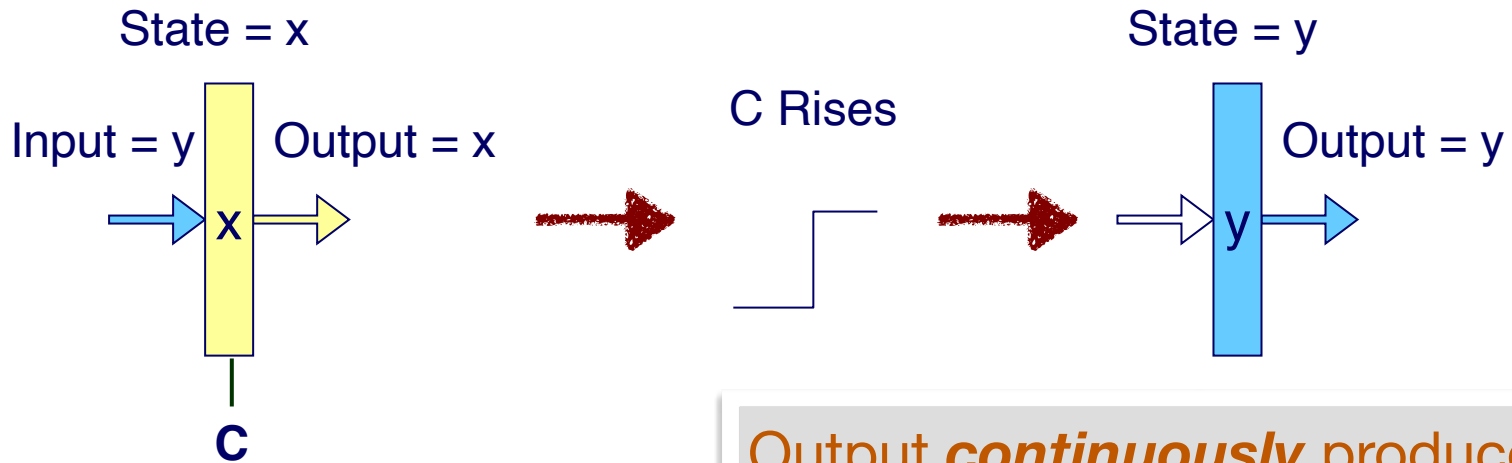


# Register Operation



- Stores data bits
- For most of time acts as barrier between input and output
- As C rises, loads input

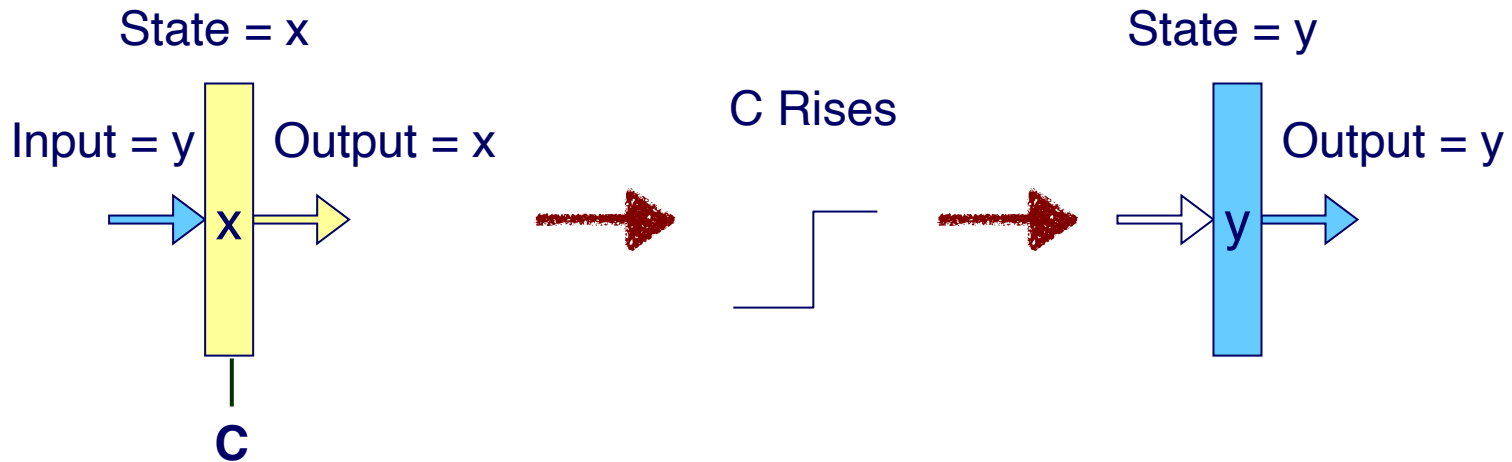
# Register Operation



Output ***continuously*** produces y after the rising edge unless you cut off power.

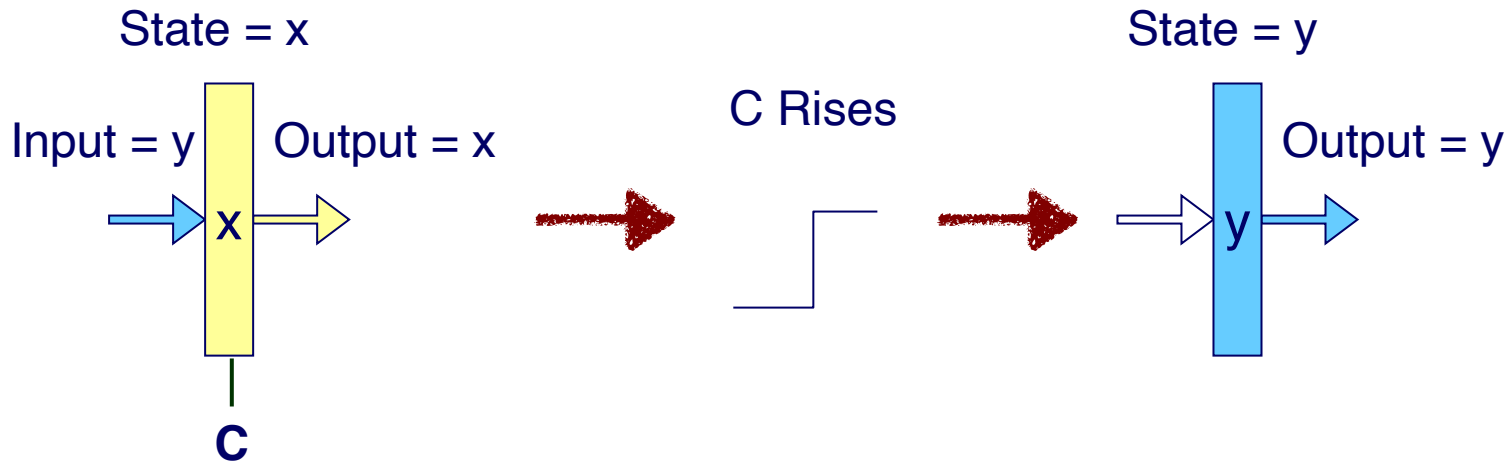
- Stores data bits
- For most of time acts as barrier between input and output
- As C rises, loads input

# Clock Signal



- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.

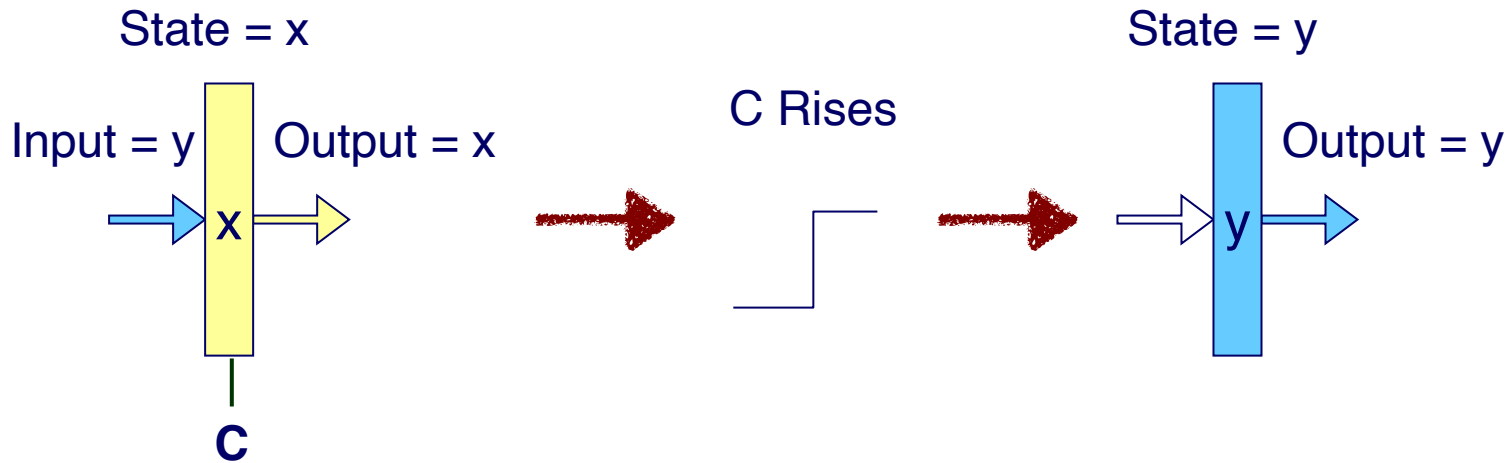
# Clock Signal



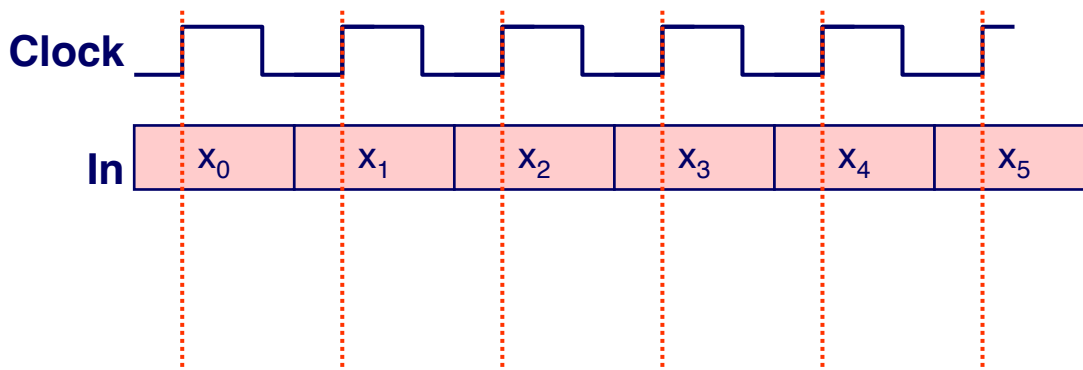
- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.



# Clock Signal

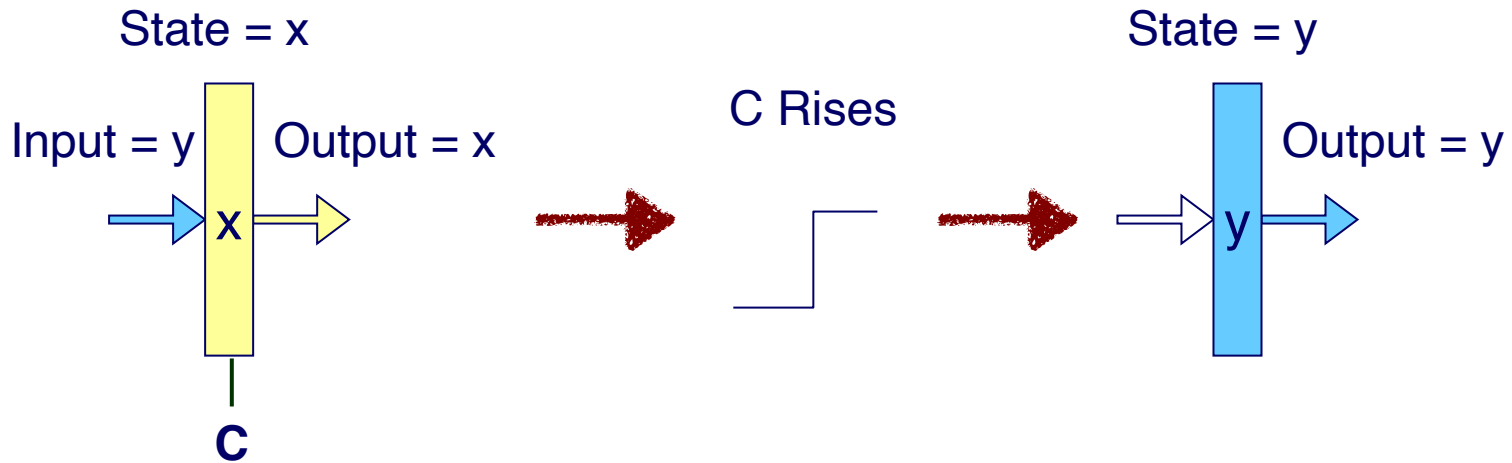


- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.

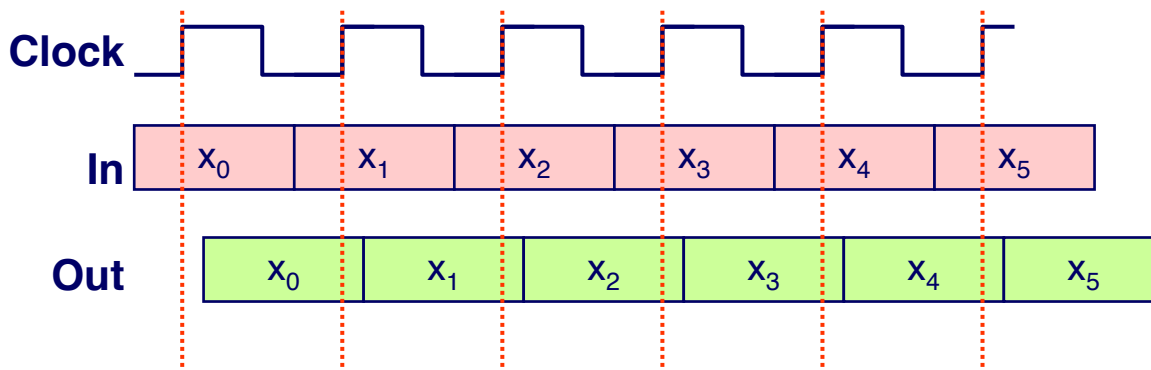




# Clock Signal

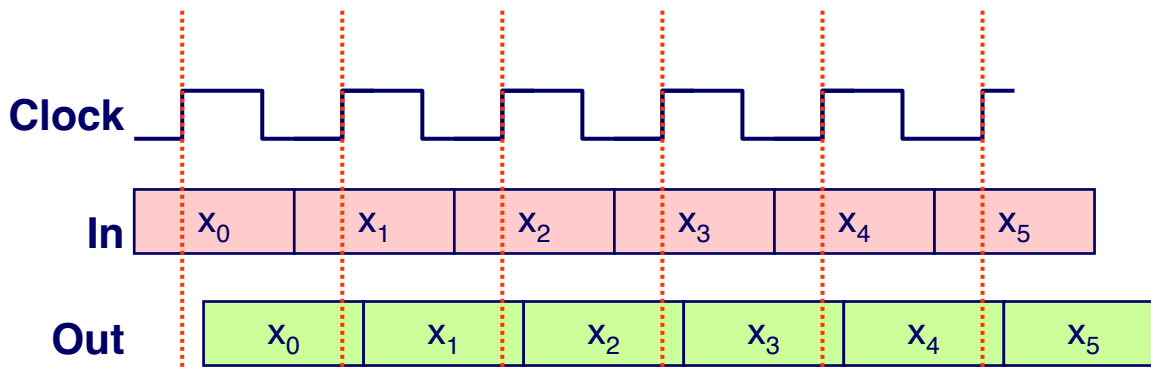


- A special  $C$ : periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.



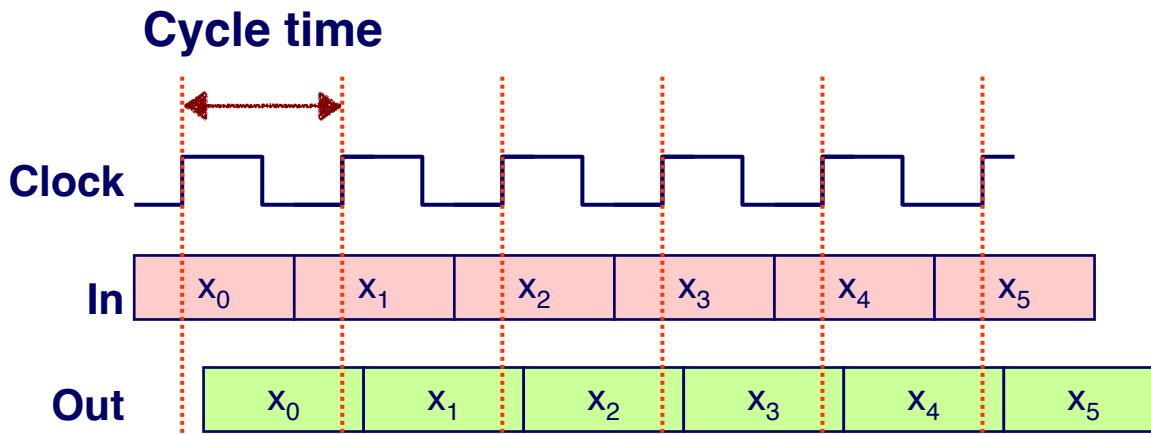
# Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.



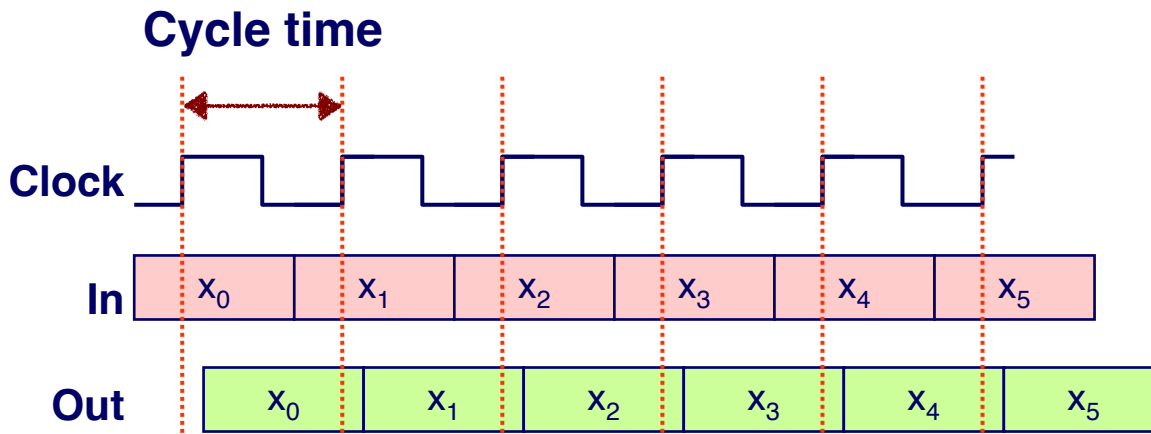
# Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.



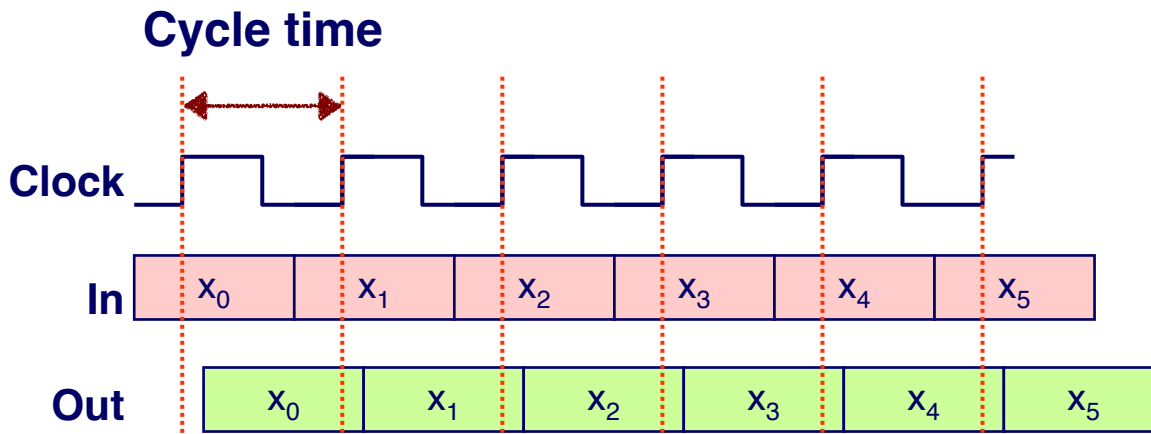
# Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.
- Frequency of a clock signal: how many rising (falling) edges in 1 second.



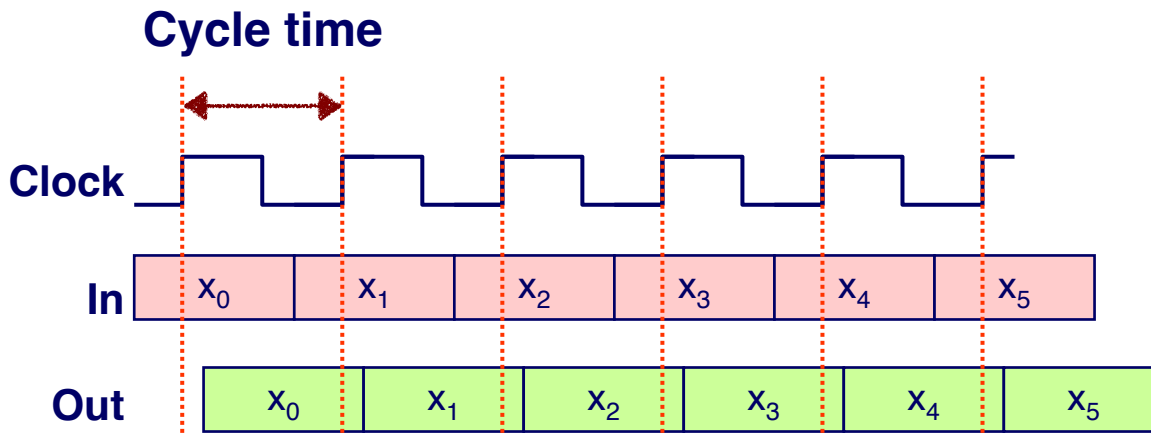
# Clock Signal

- Cycle time of a clock signal: the time duration between two rising edges.
- Frequency of a clock signal: how many rising (falling) edges in 1 second.
- 1 GHz CPU means the clock frequency is 1 GHz



# Clock Signal

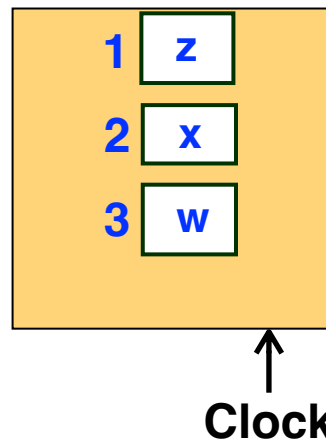
- Cycle time of a clock signal: the time duration between two rising edges.
- Frequency of a clock signal: how many rising (falling) edges in 1 second.
- 1 GHz CPU means the clock frequency is 1 GHz
  - The cycle time is  $1/10^9 = 1 \text{ ns}$



# Register File

- A register file consists of a set of registers that you can individually read from and write to.

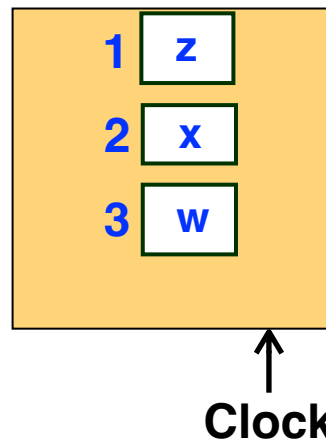
## Register File



# Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out

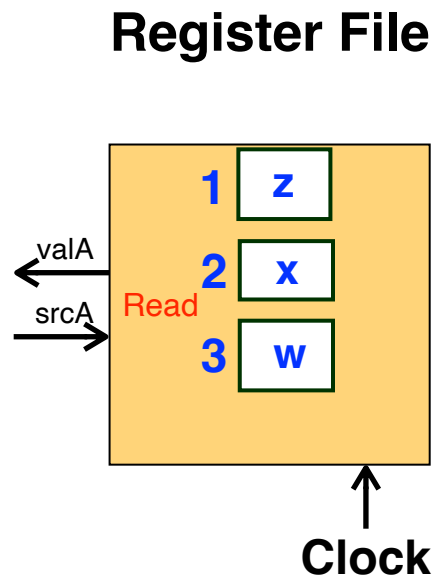
## Register File





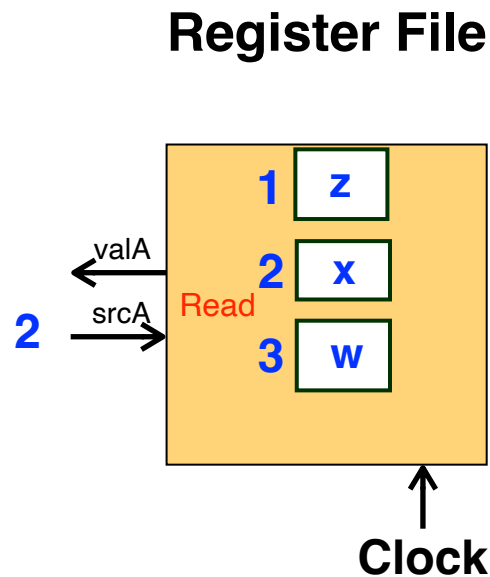
# Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out



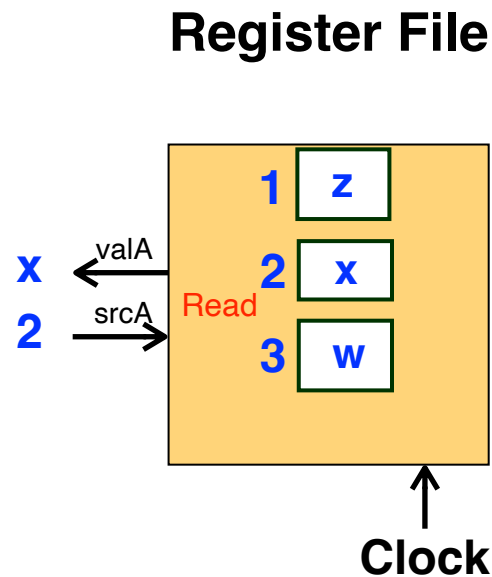
# Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out



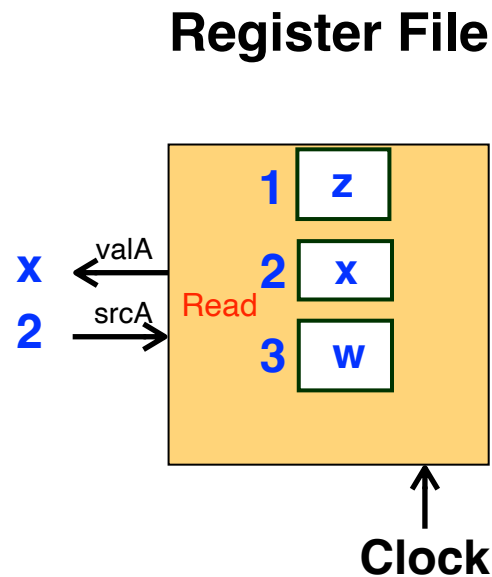
# Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out



# Register File

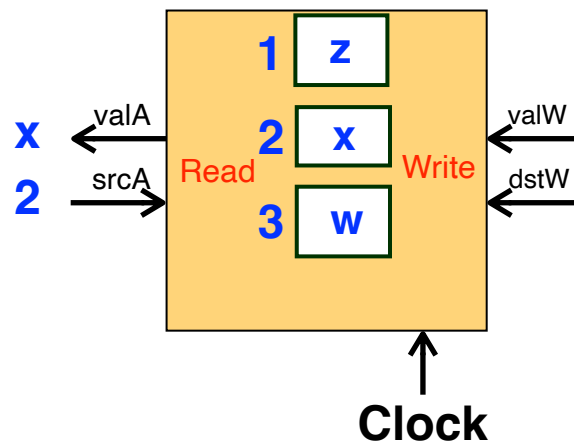
- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value



# Register File

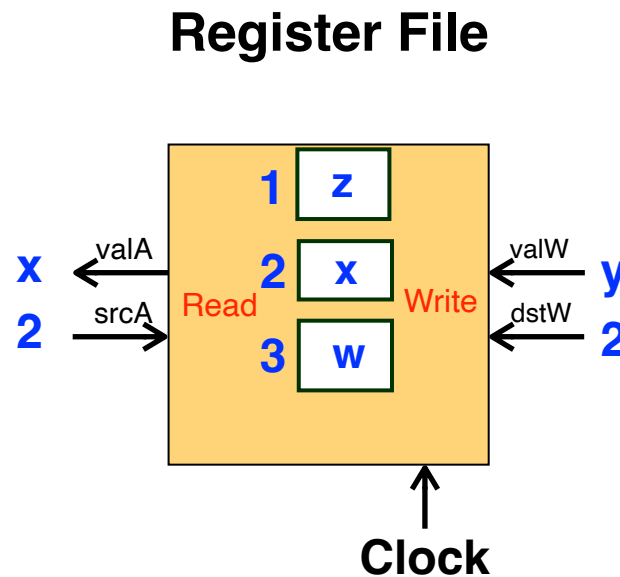
- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value

## Register File



# Register File

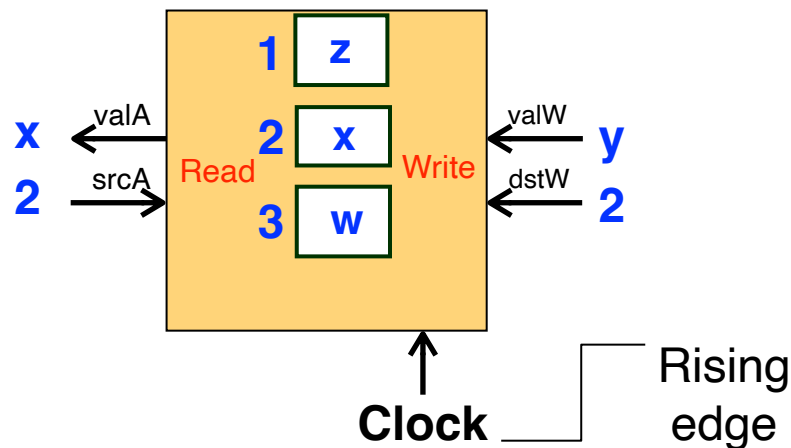
- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value



# Register File

- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value

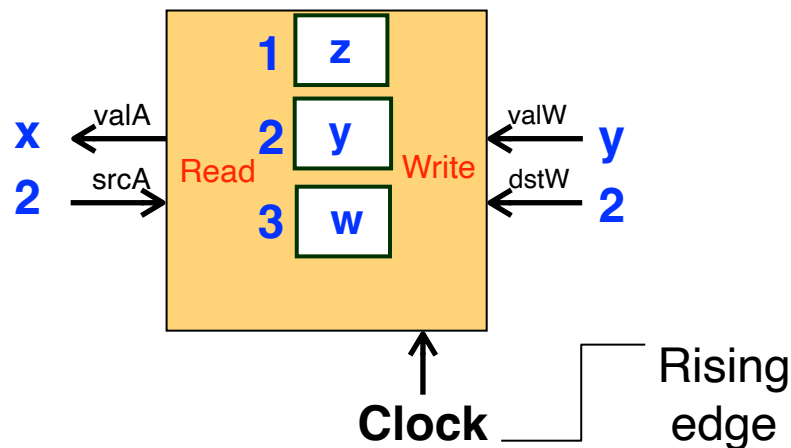
## Register File



# Register File

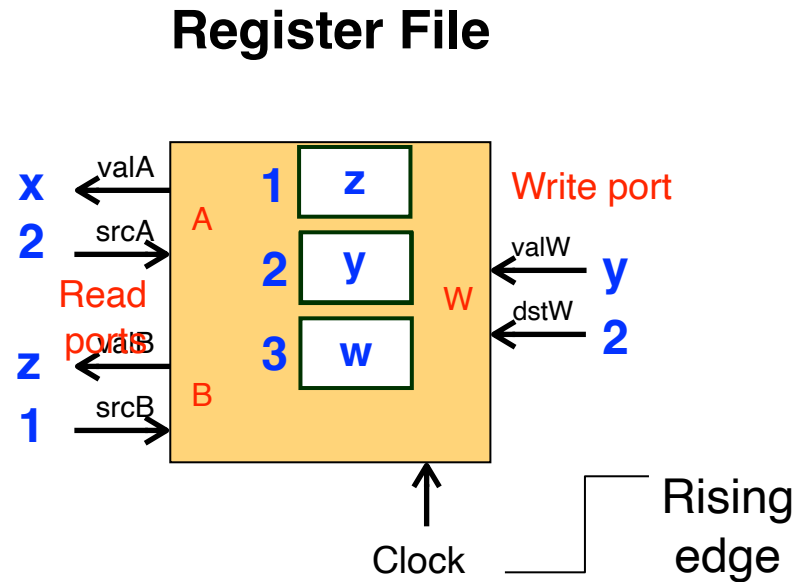
- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value

## Register File





# Register File



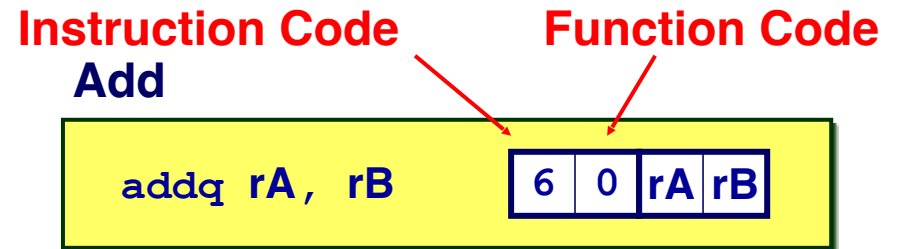
- Stores multiple registers of data
  - Address input specifies which register to read or write
- **Multiple Ports:** Can read and/or write multiple words in one cycle. Each port has separate address and data input/output

# Processor Microarchitecture

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

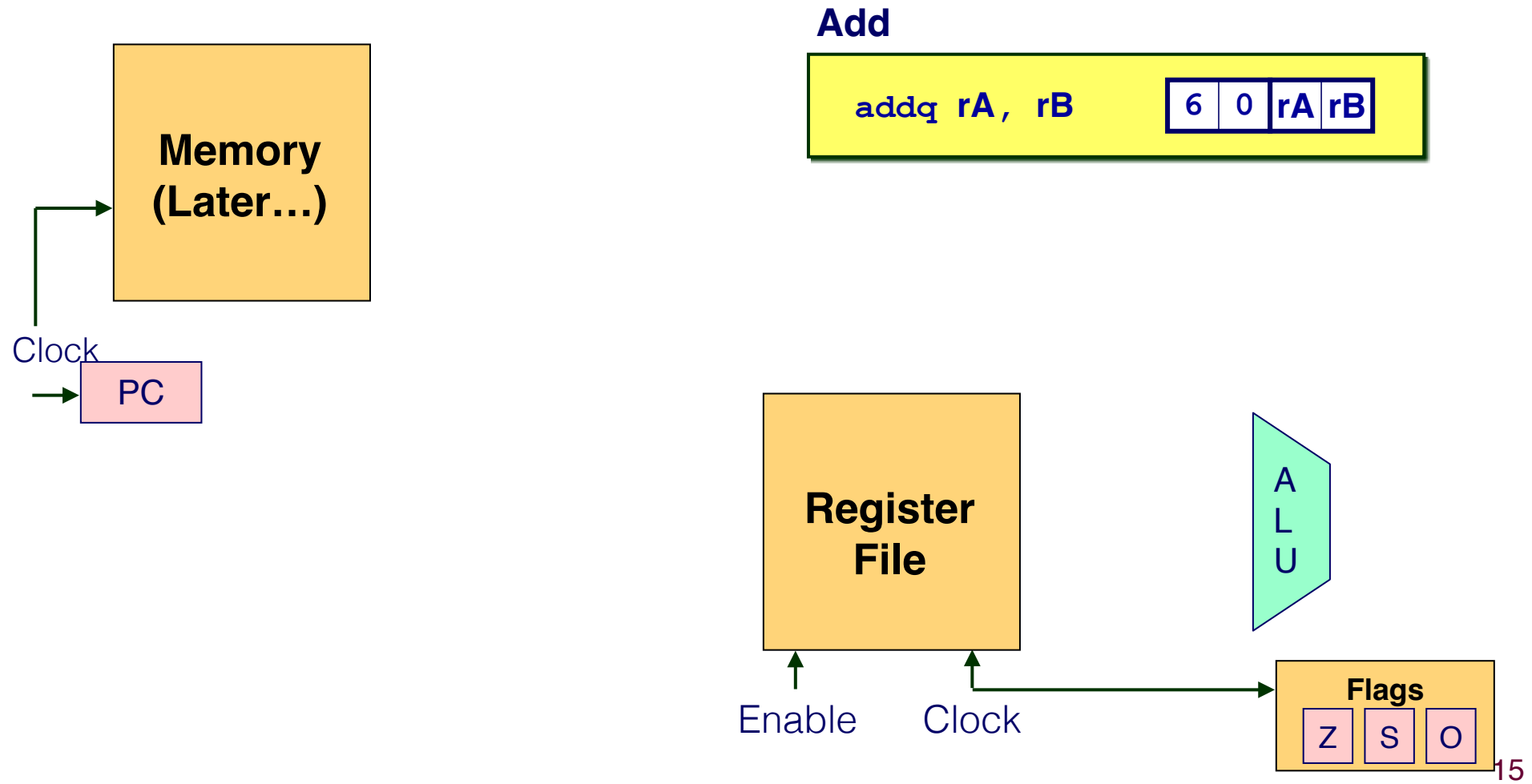
# Executing an ADD instruction

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



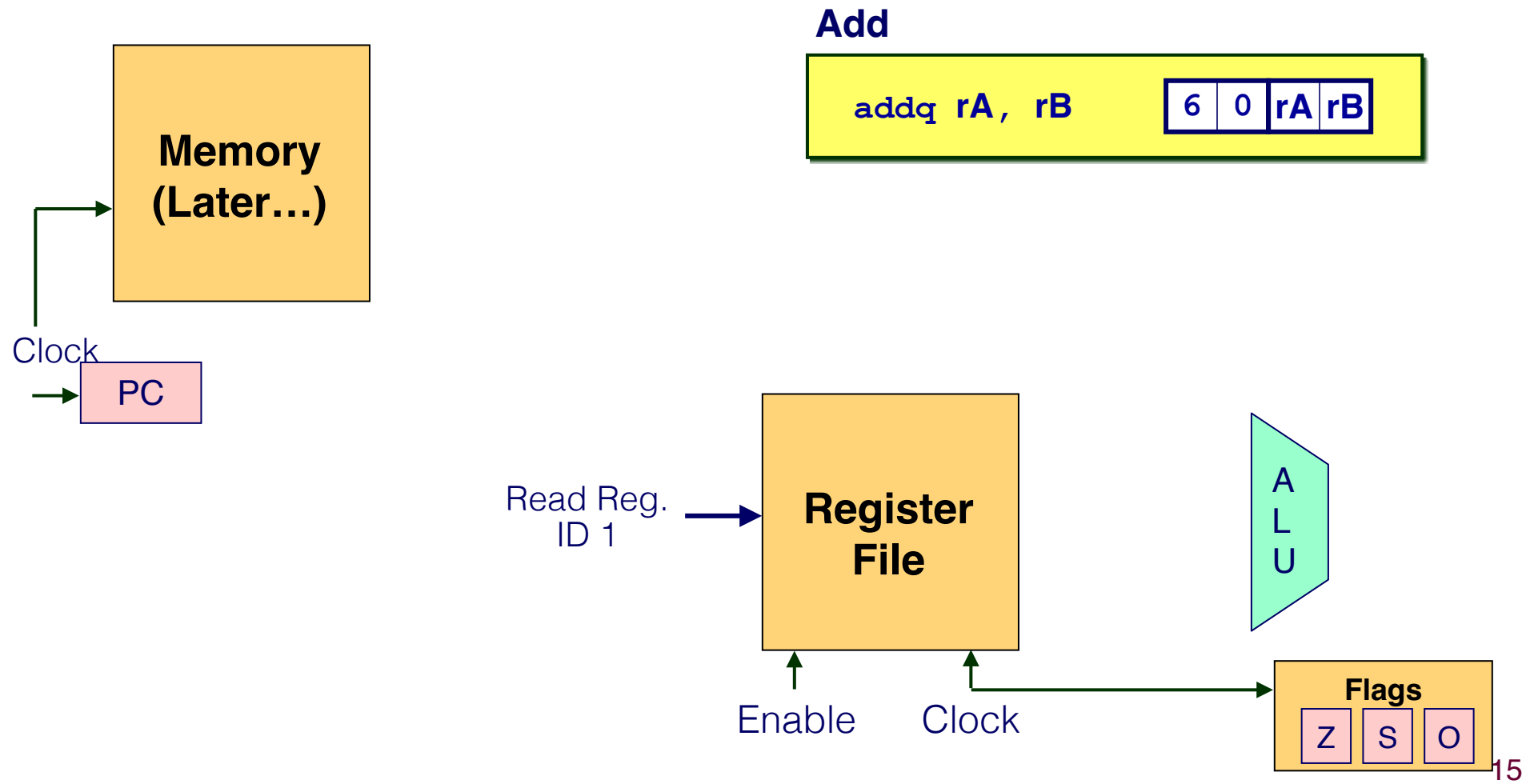
# Executing an ADD instruction

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



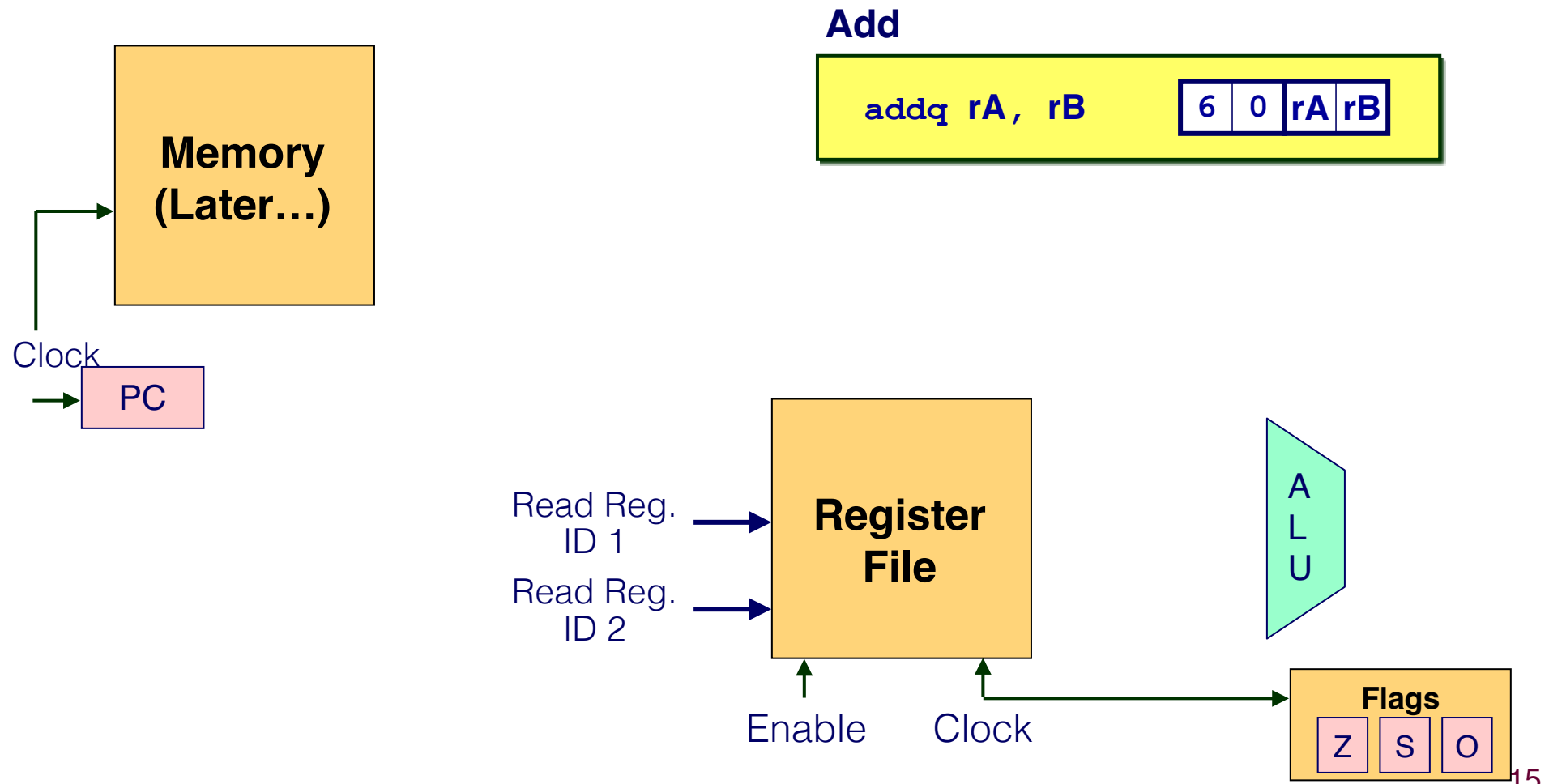
# Executing an ADD instruction

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



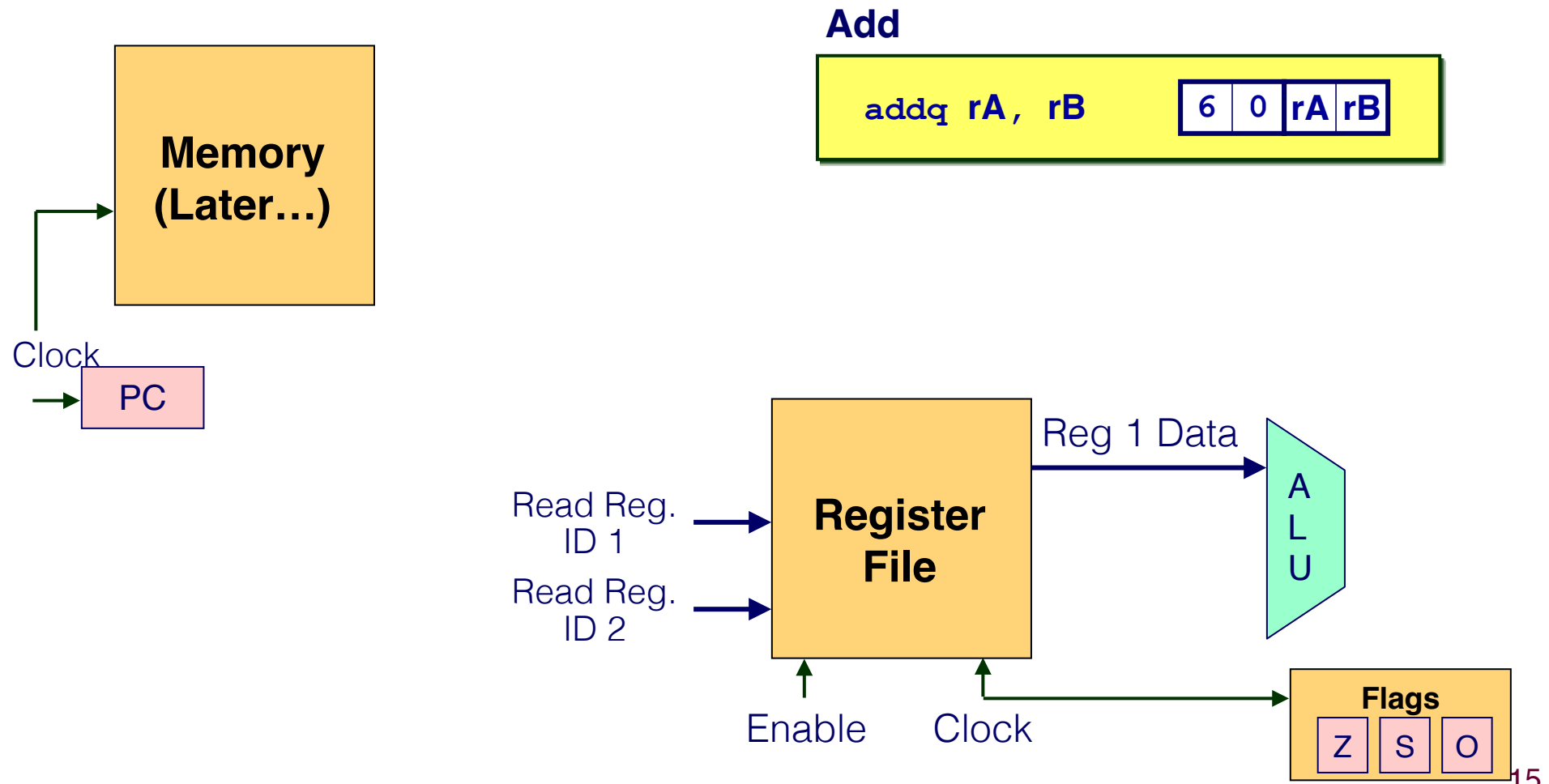
# Executing an ADD instruction

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



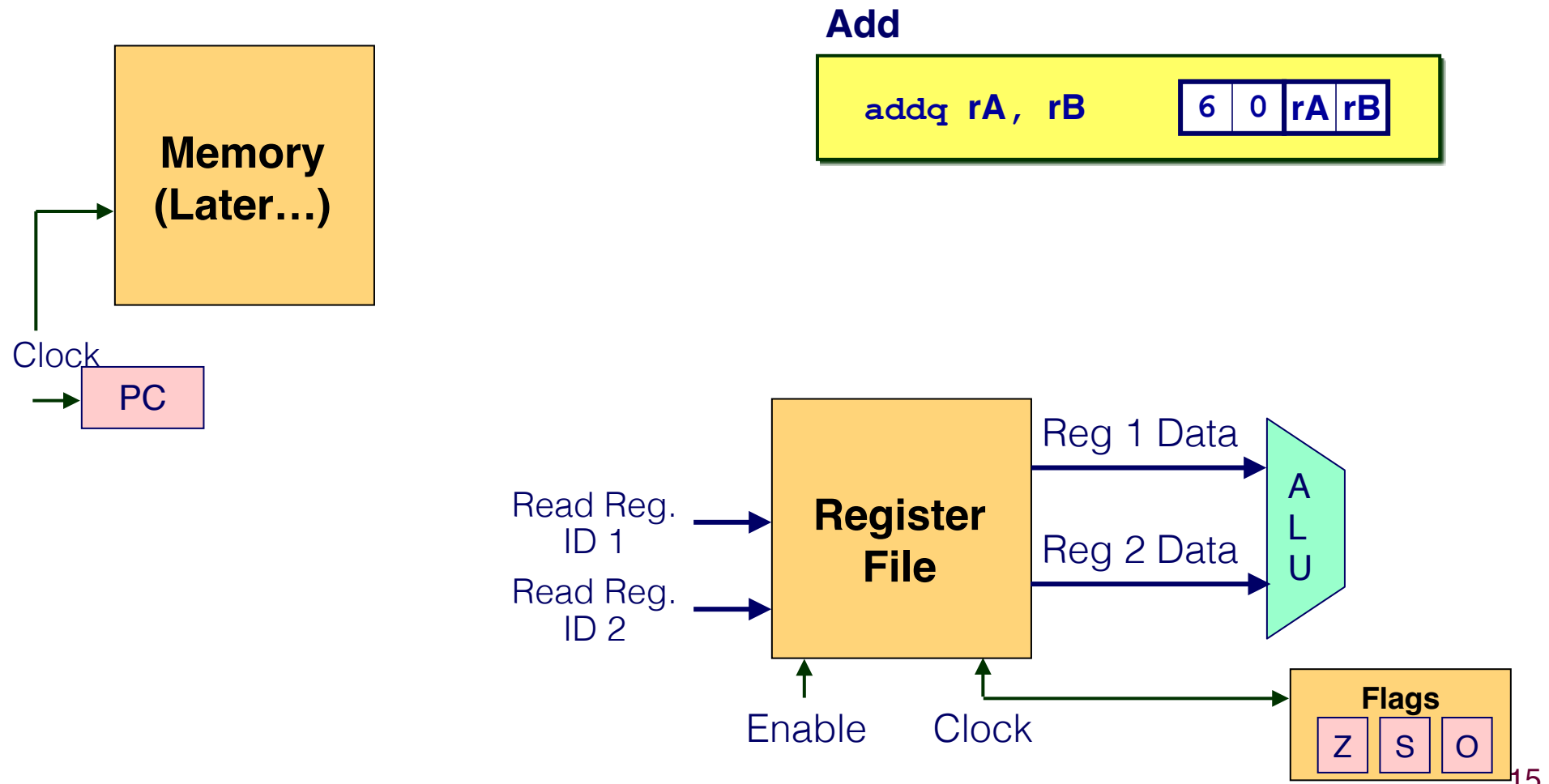
# Executing an ADD instruction

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



# Executing an ADD instruction

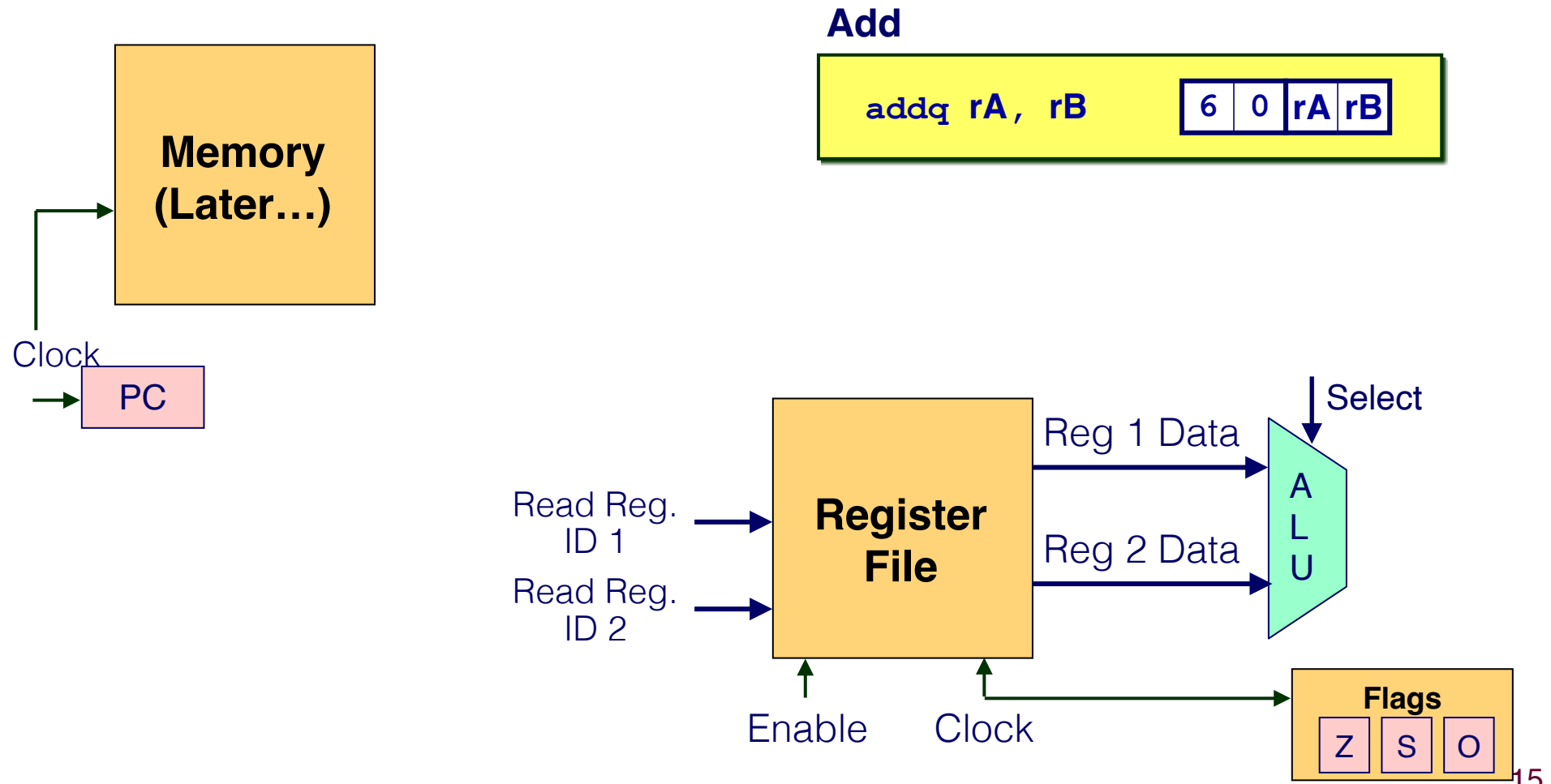
- How does the processor execute `addq %rax, %rsi`
- The binary encoding is `60 06`





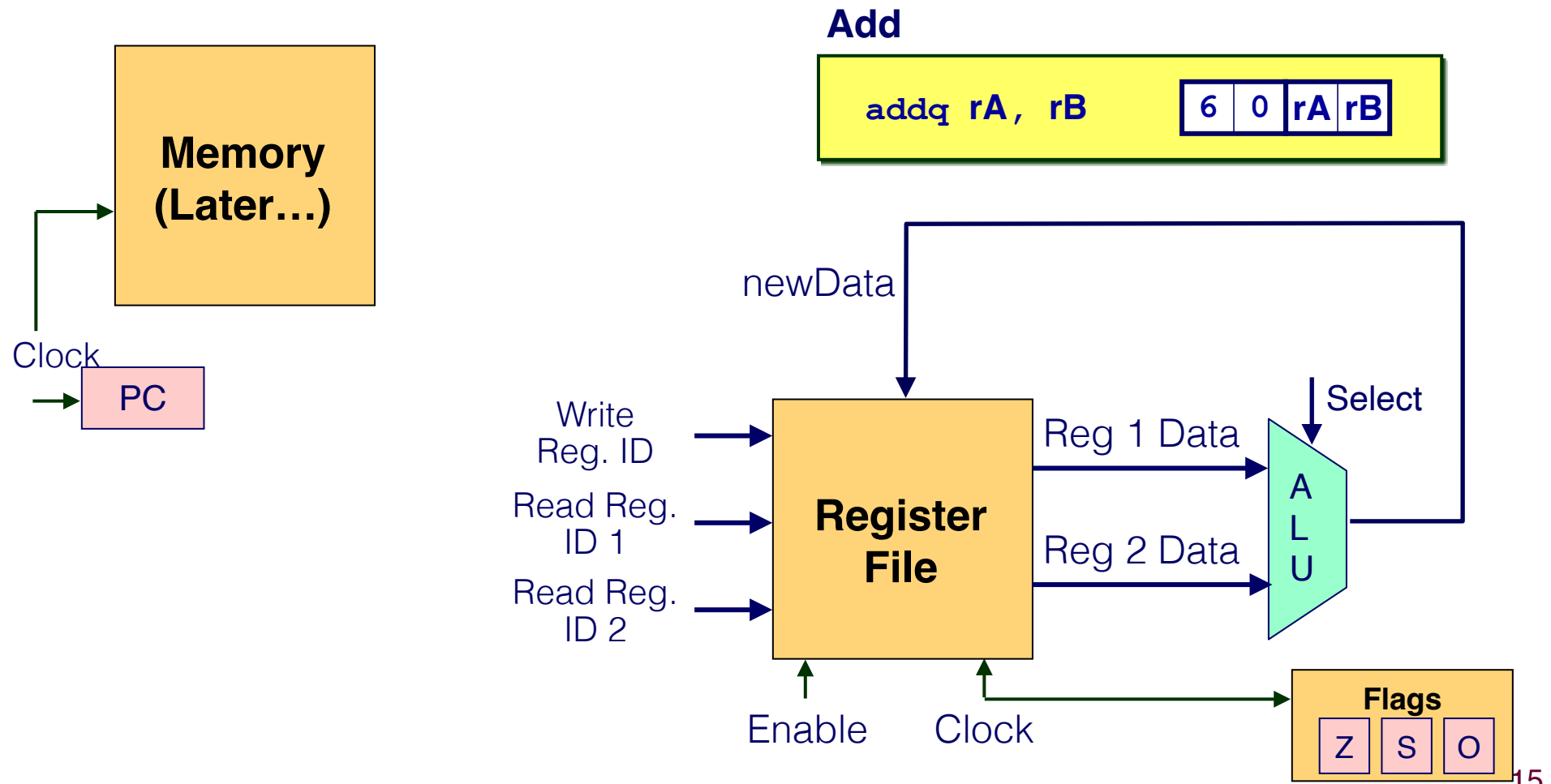
# Executing an ADD instruction

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



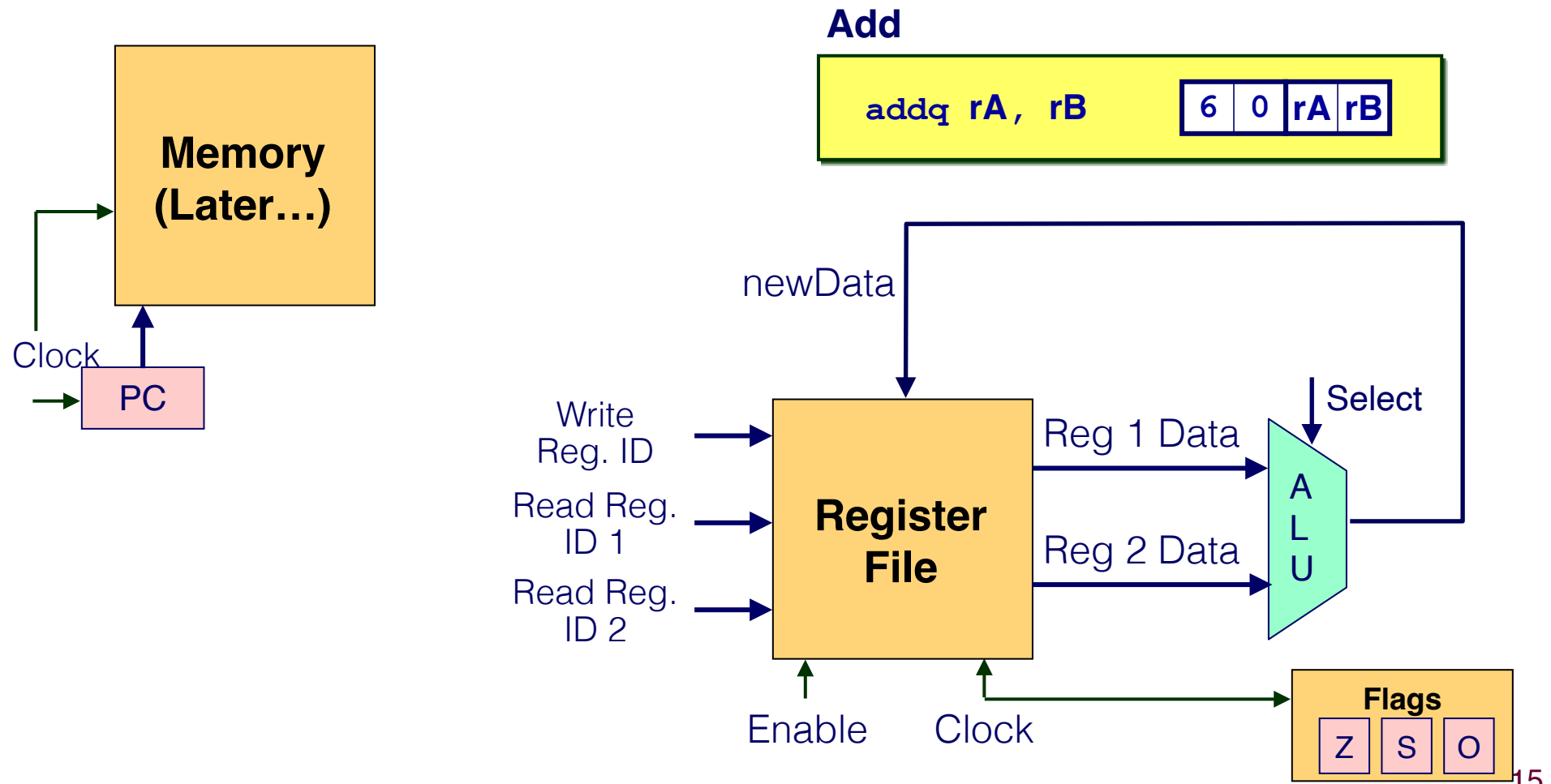
# Executing an ADD instruction

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



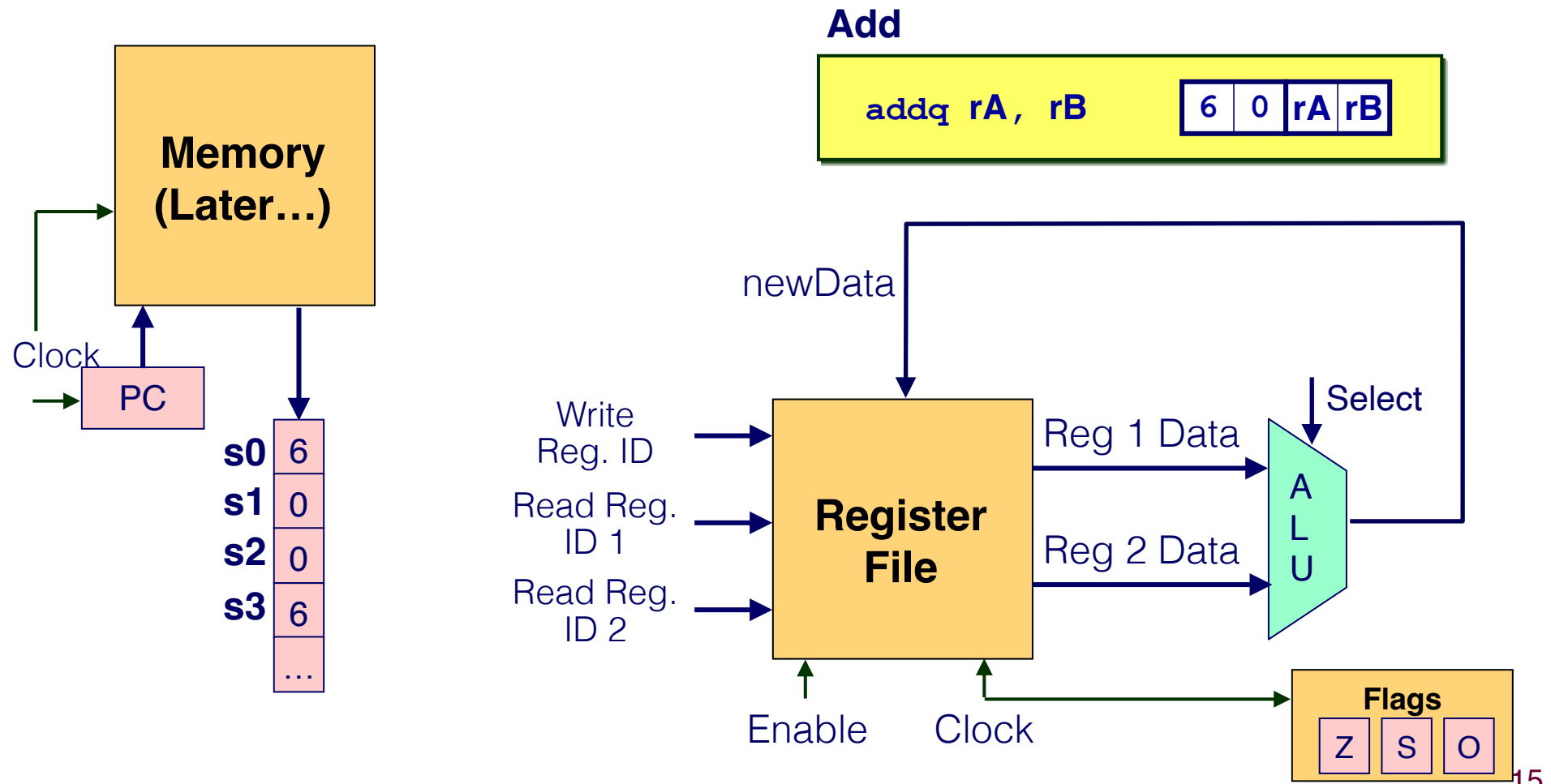
# Executing an ADD instruction

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



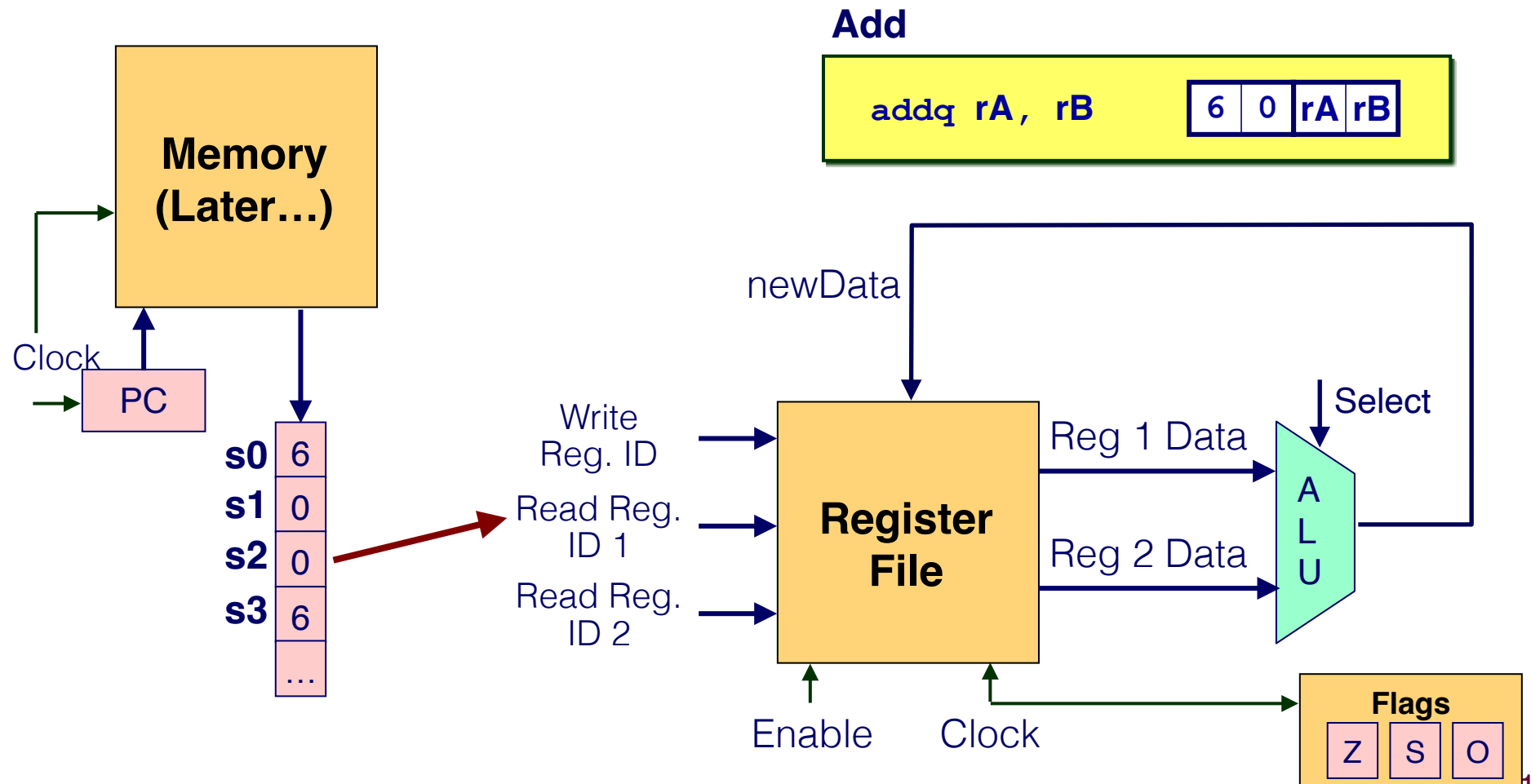
# Executing an ADD instruction

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



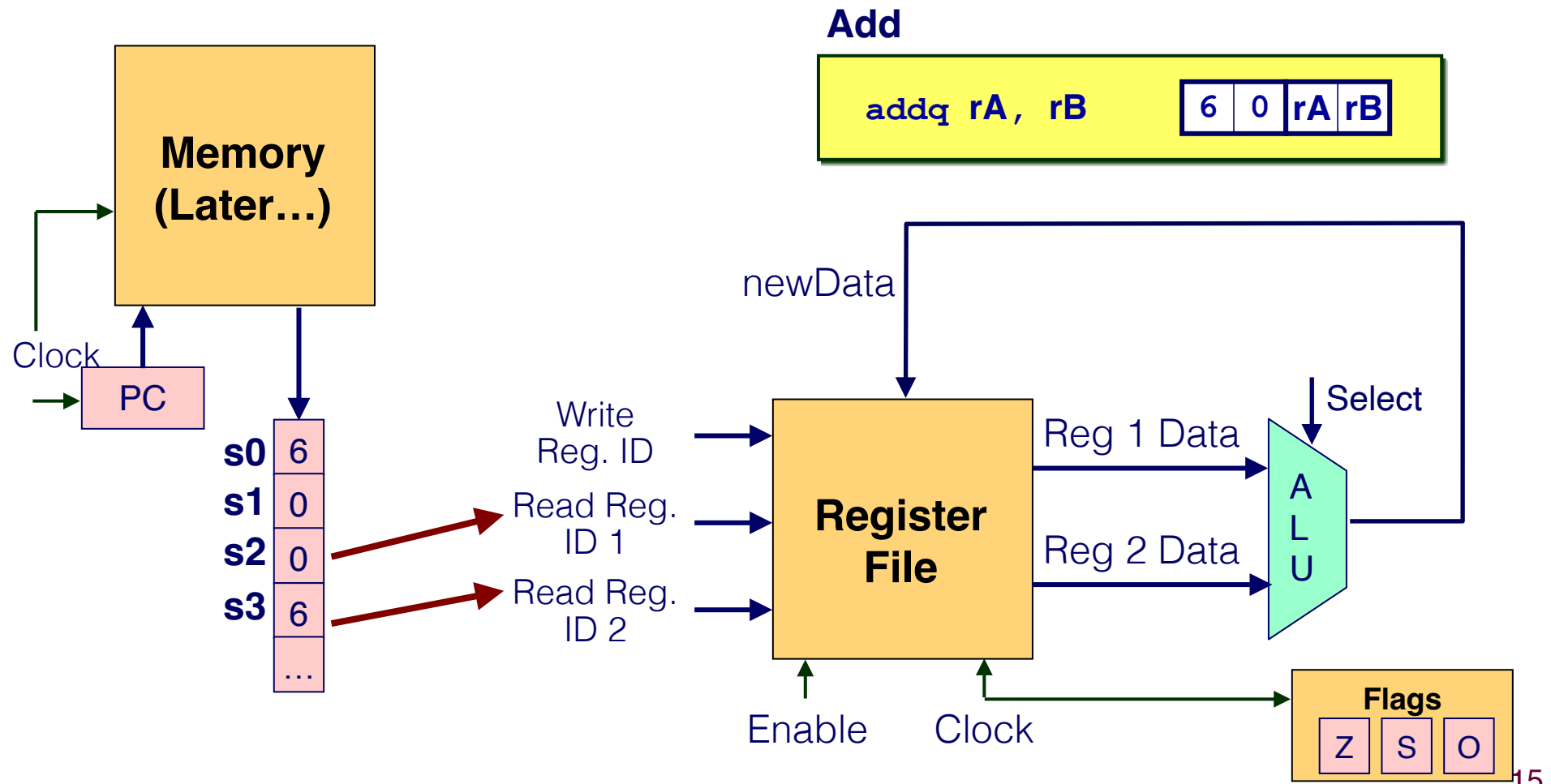
# Executing an ADD instruction

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



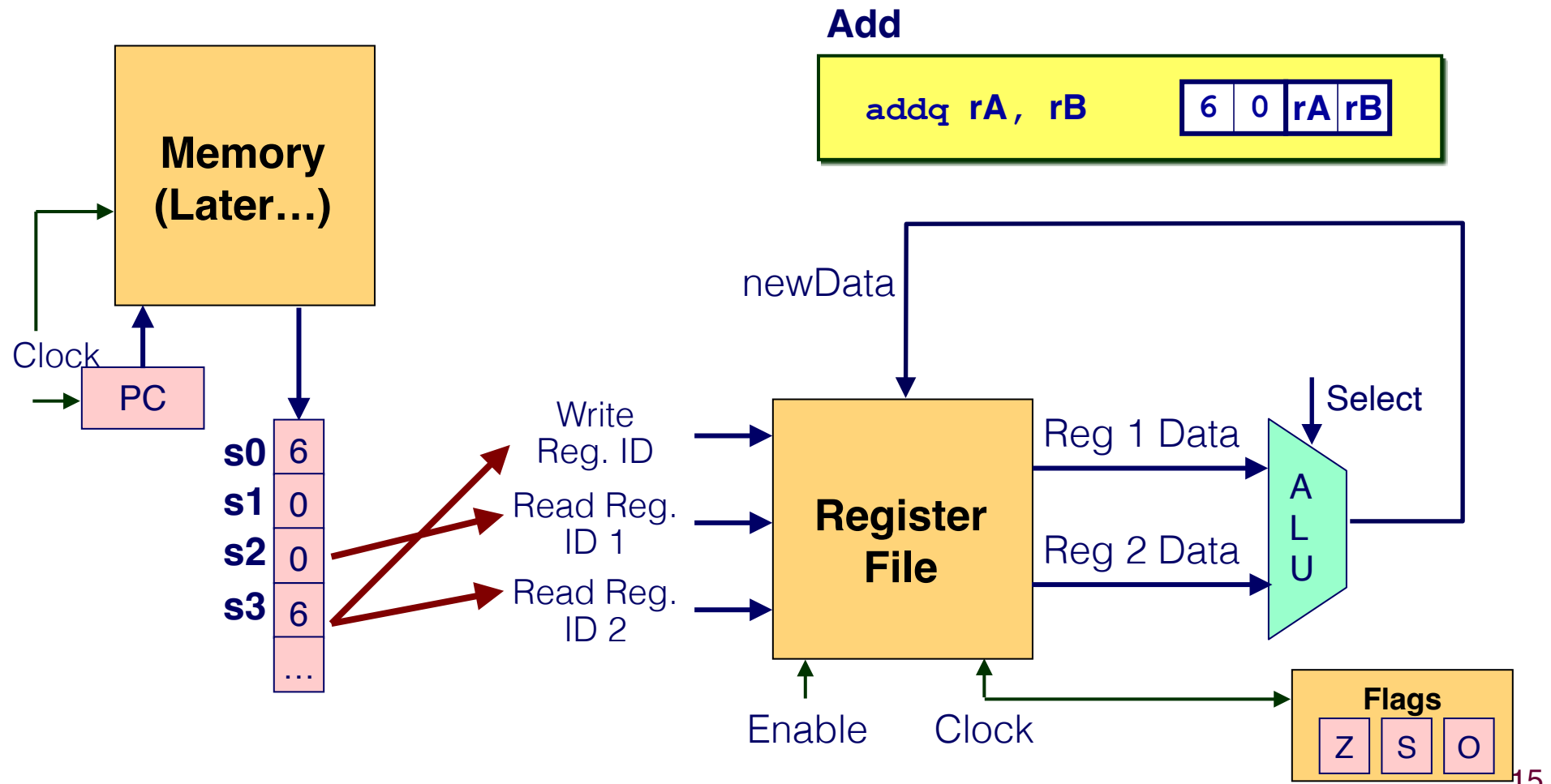
# Executing an ADD instruction

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



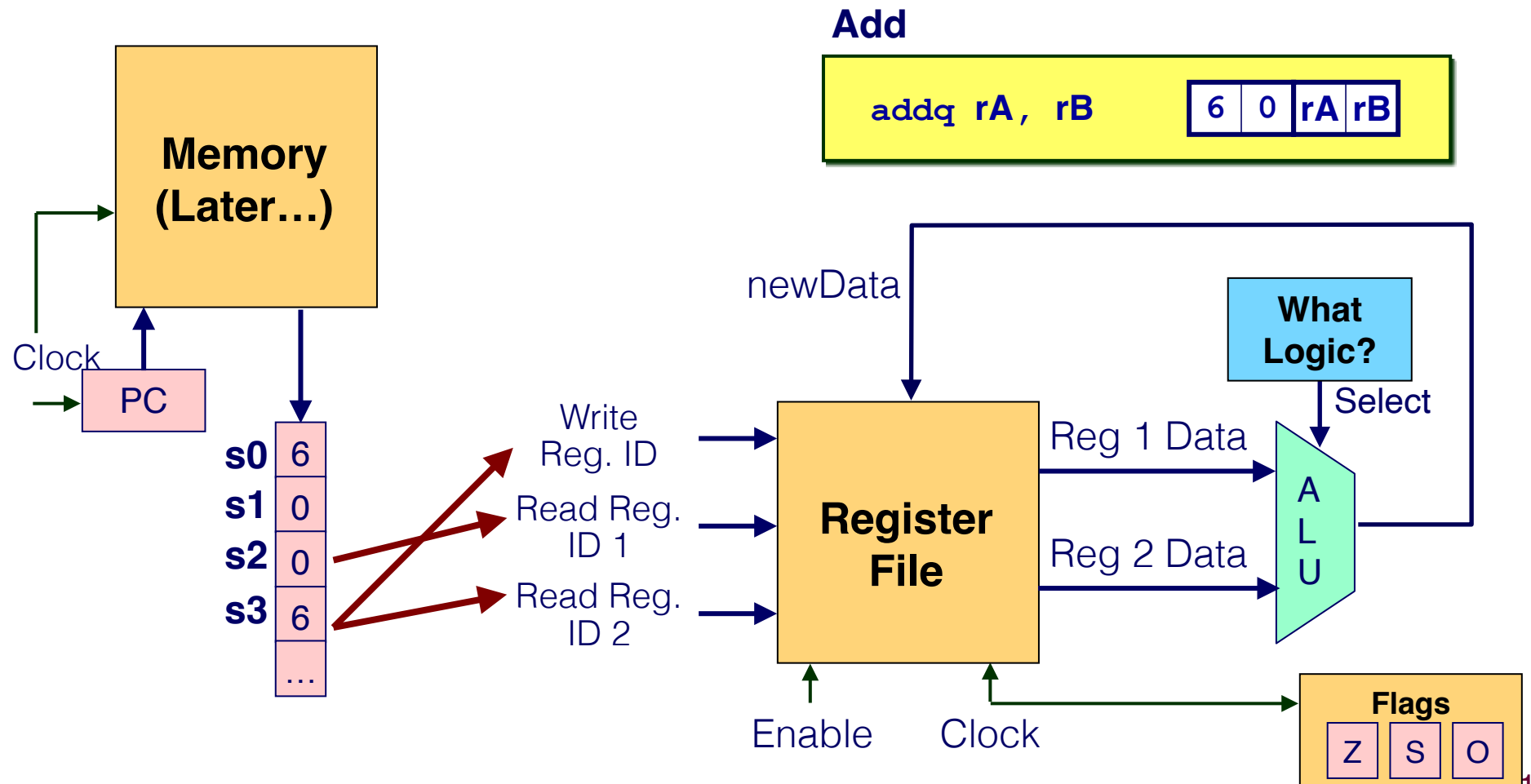
# Executing an ADD instruction

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



# Executing an ADD instruction

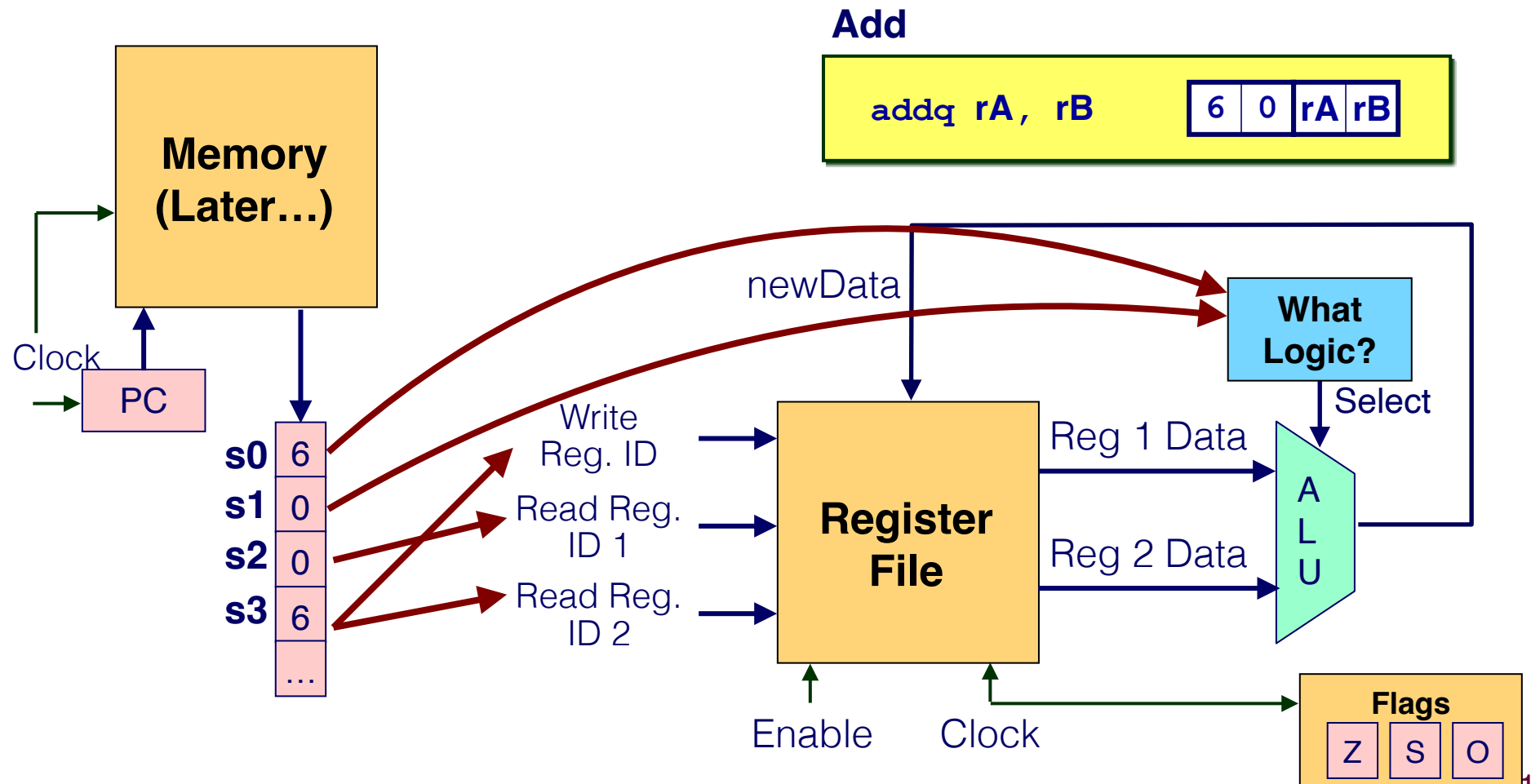
- How does the processor execute `addq %rax, %rsi`
- The binary encoding is `60 06`





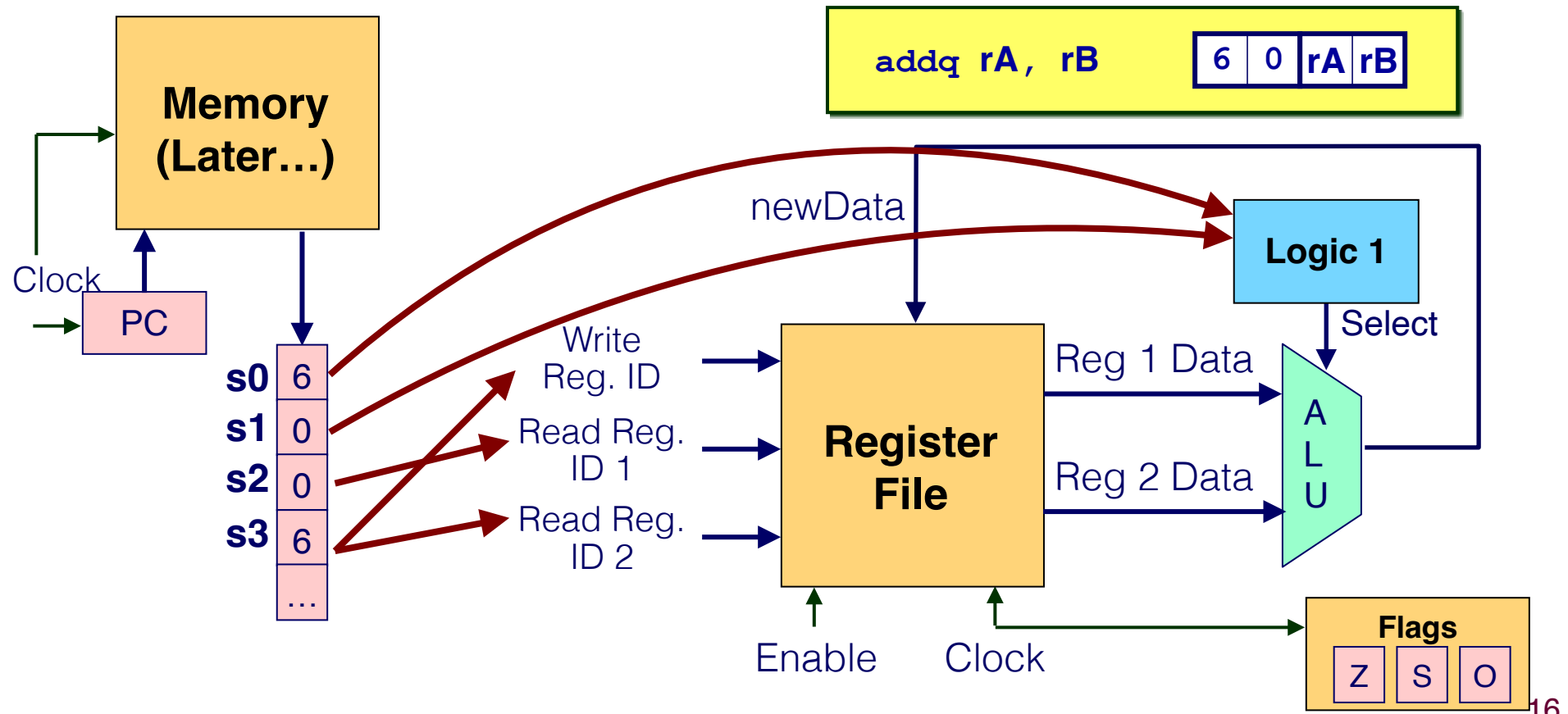
# Executing an ADD instruction

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



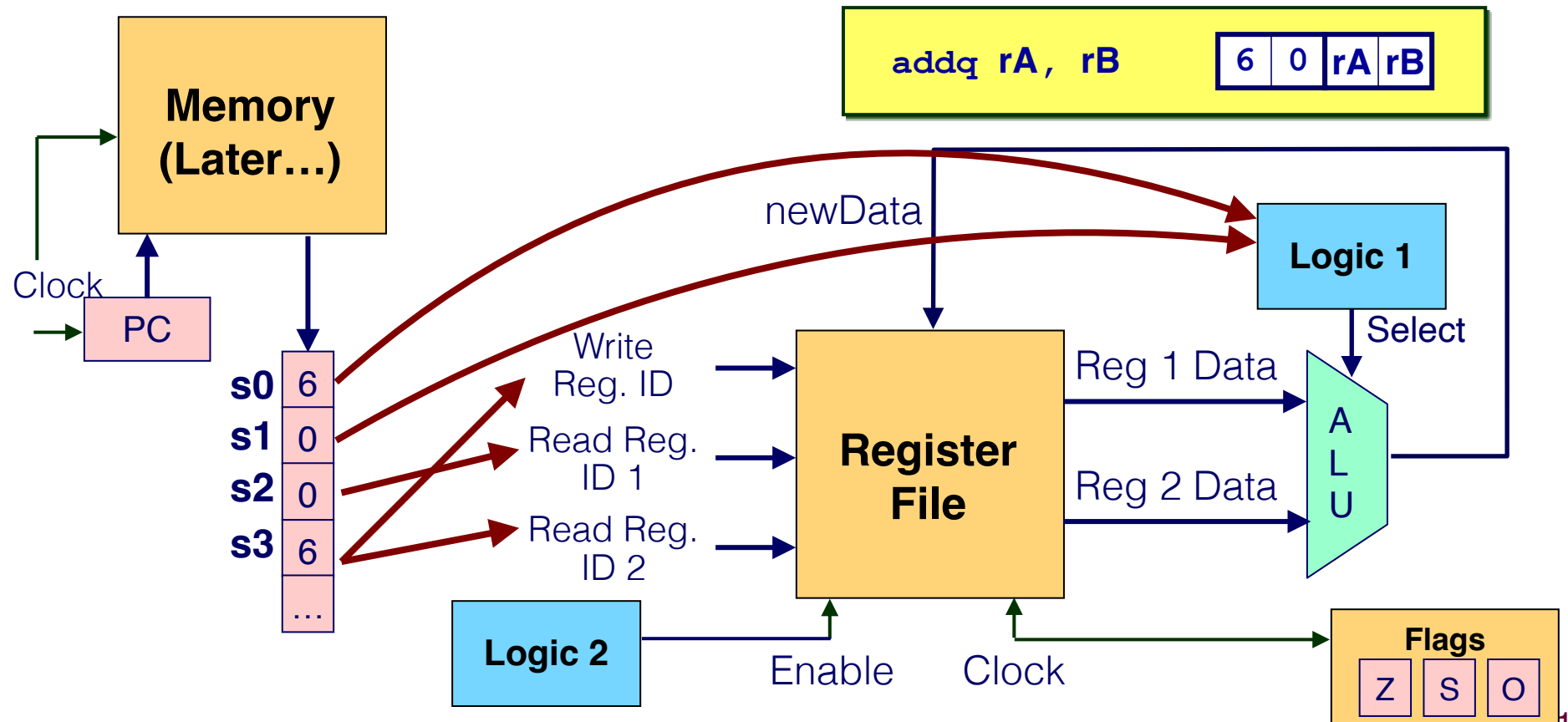
# Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;



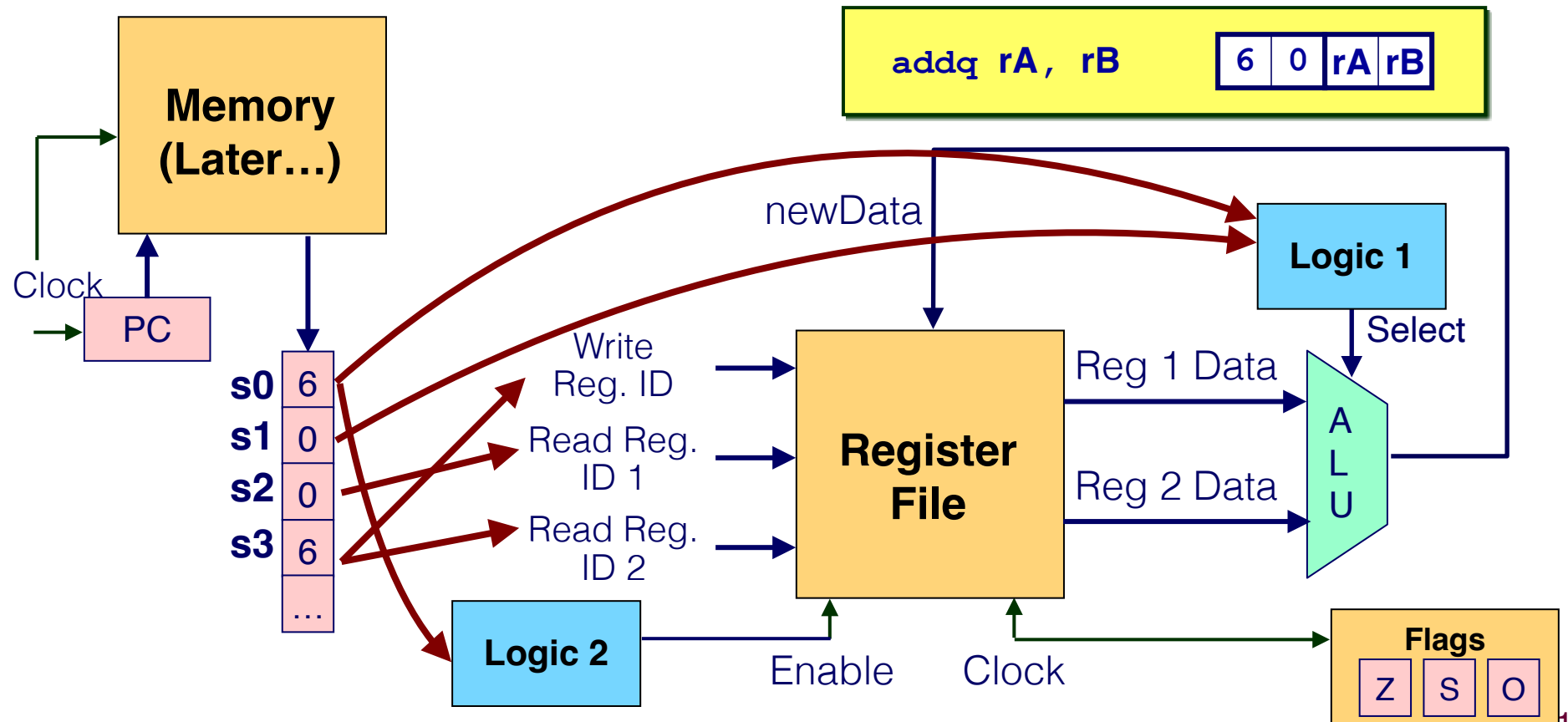
# Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;



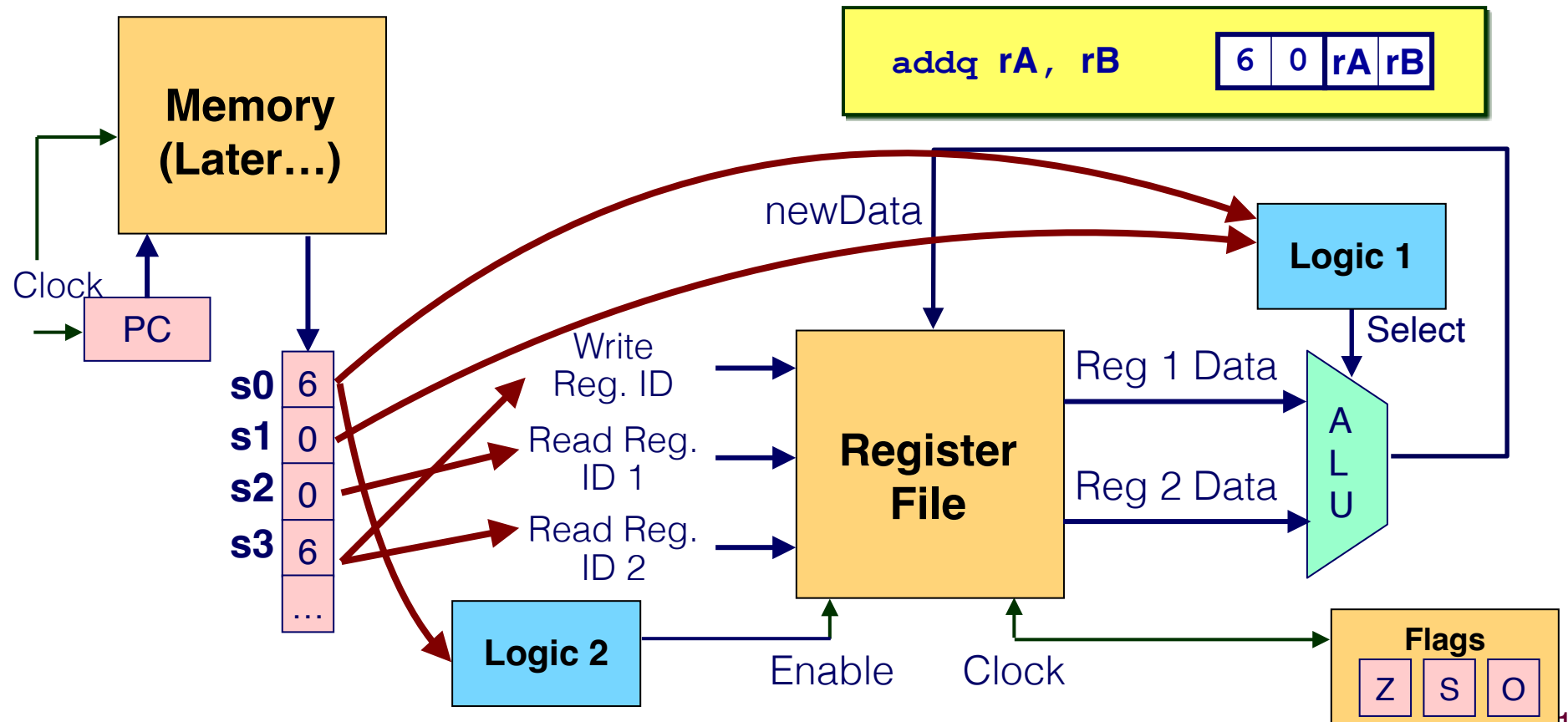
# Executing an ADD instruction

- Logic 1: if (s0 == 6) select = s1;



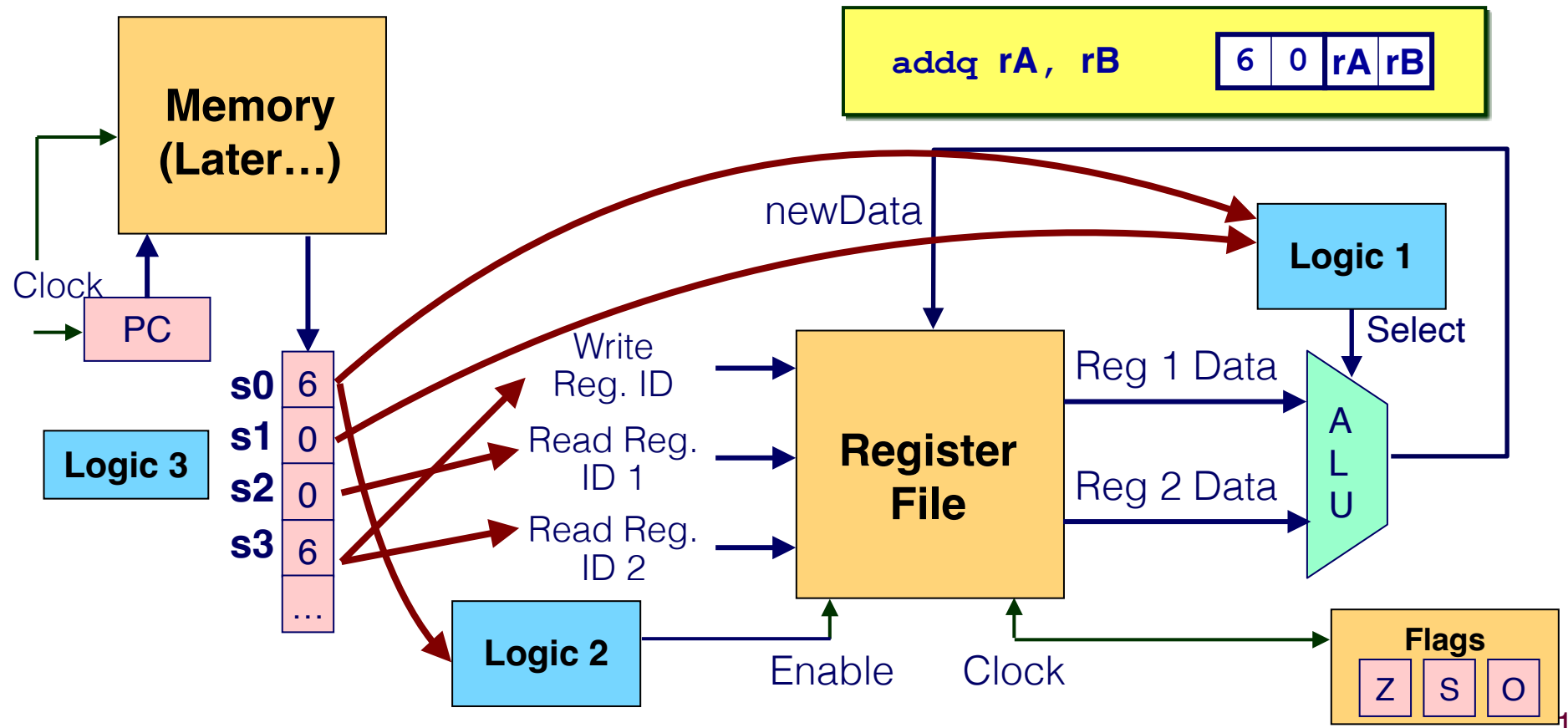
# Executing an ADD instruction

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



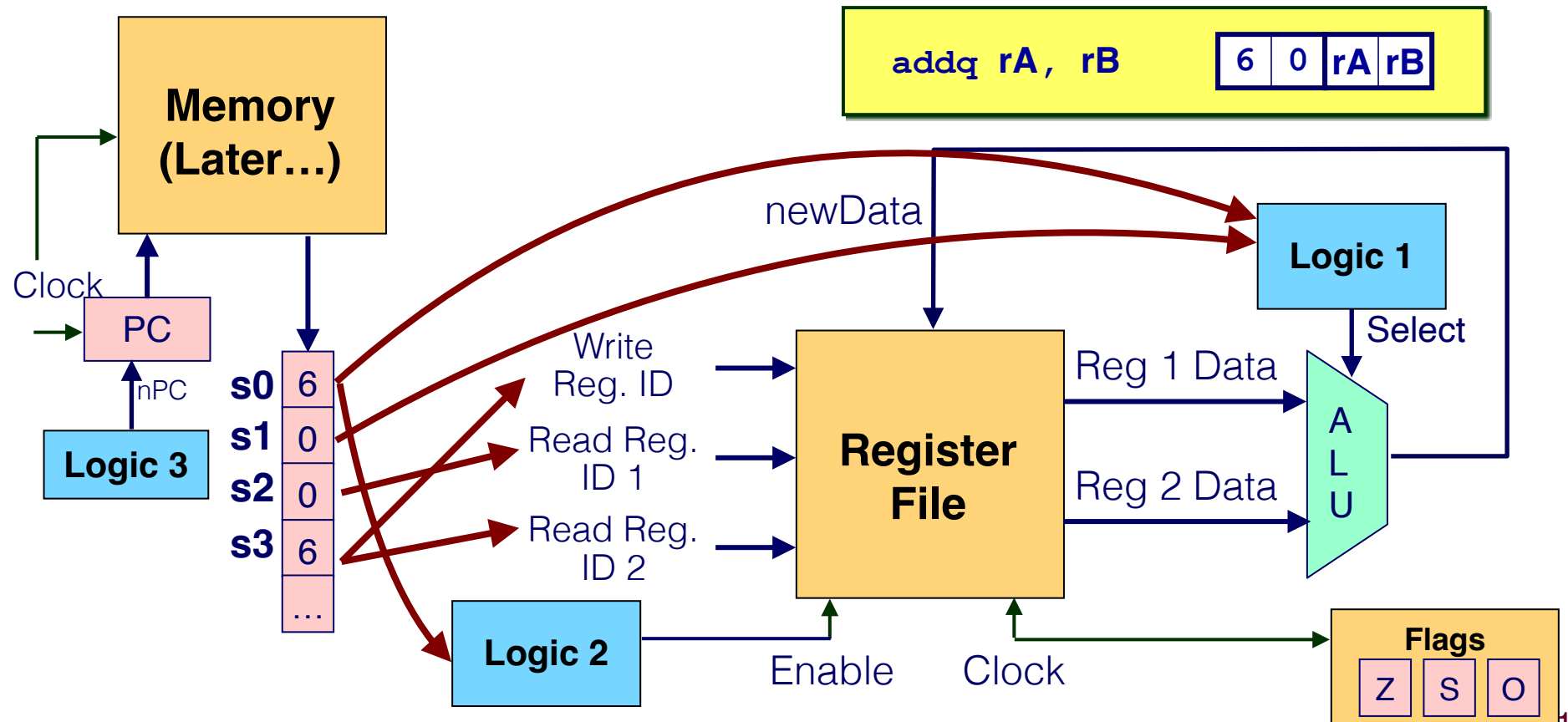
# Executing an ADD instruction

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



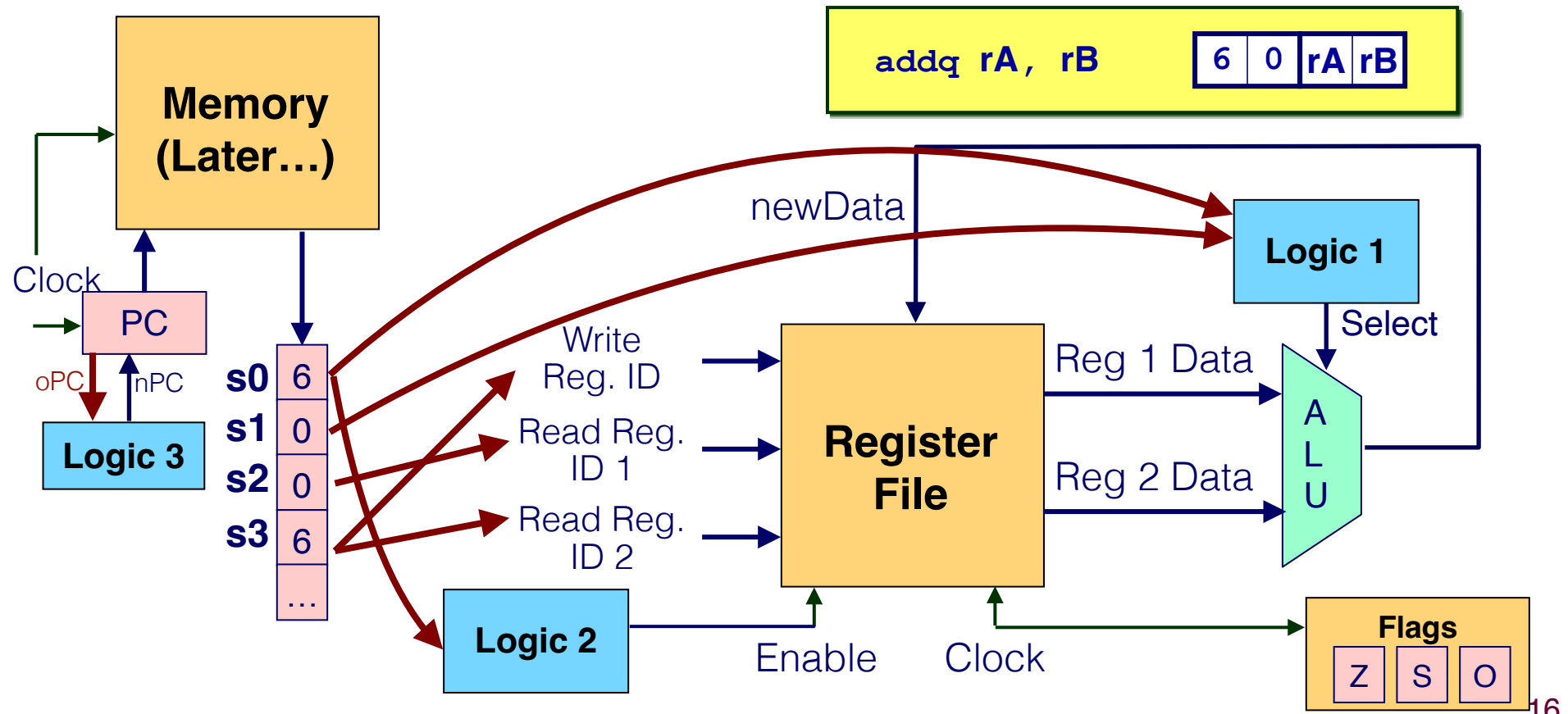
# Executing an ADD instruction

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



# Executing an ADD instruction

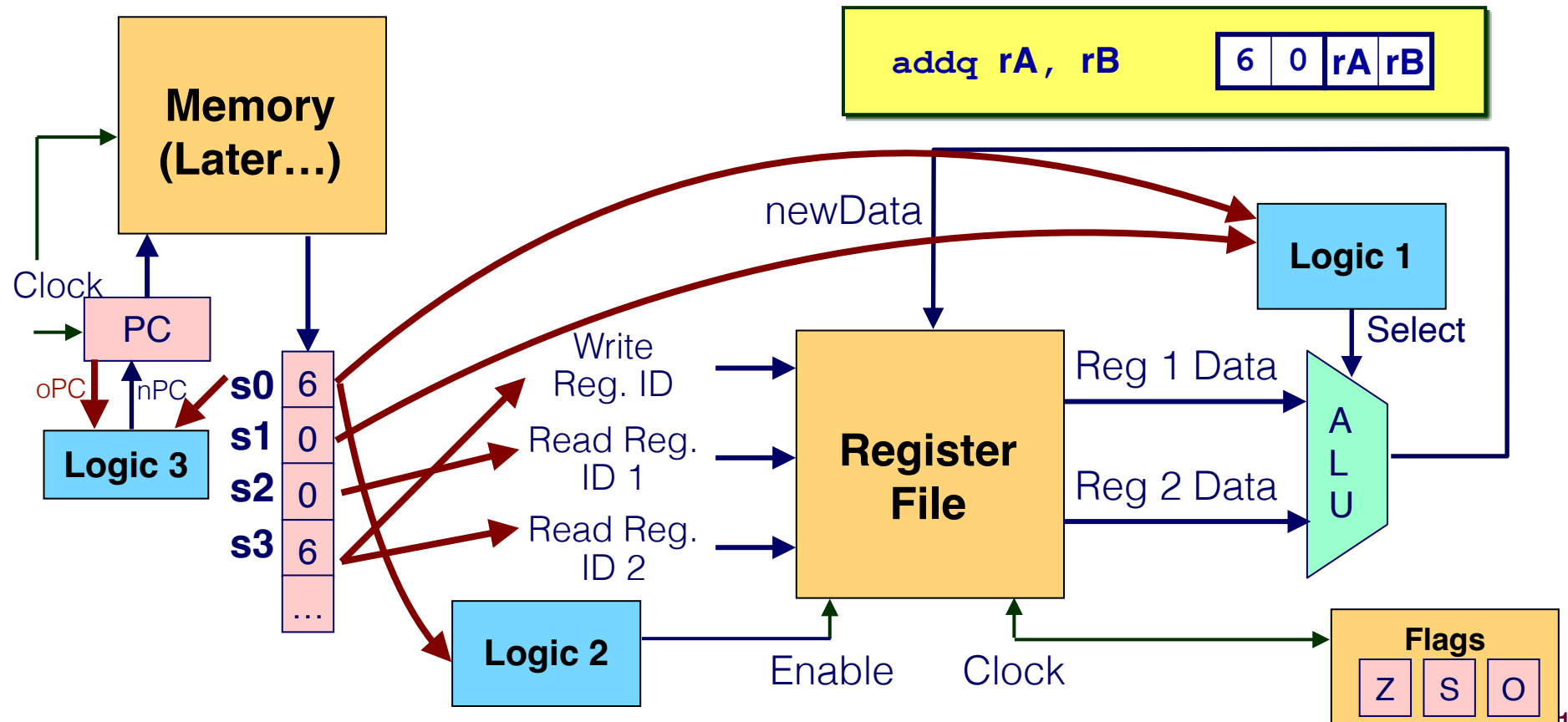
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;





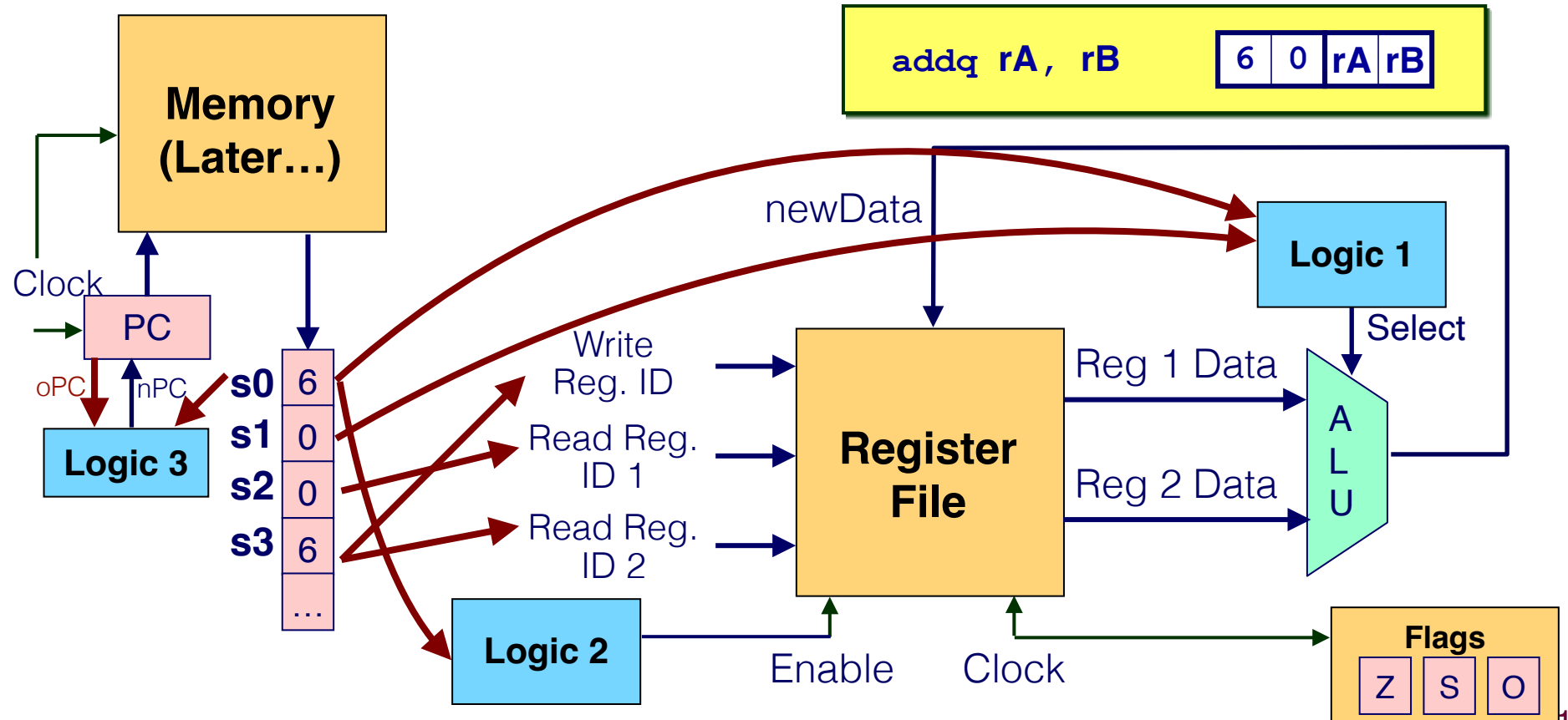
# Executing an ADD instruction

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



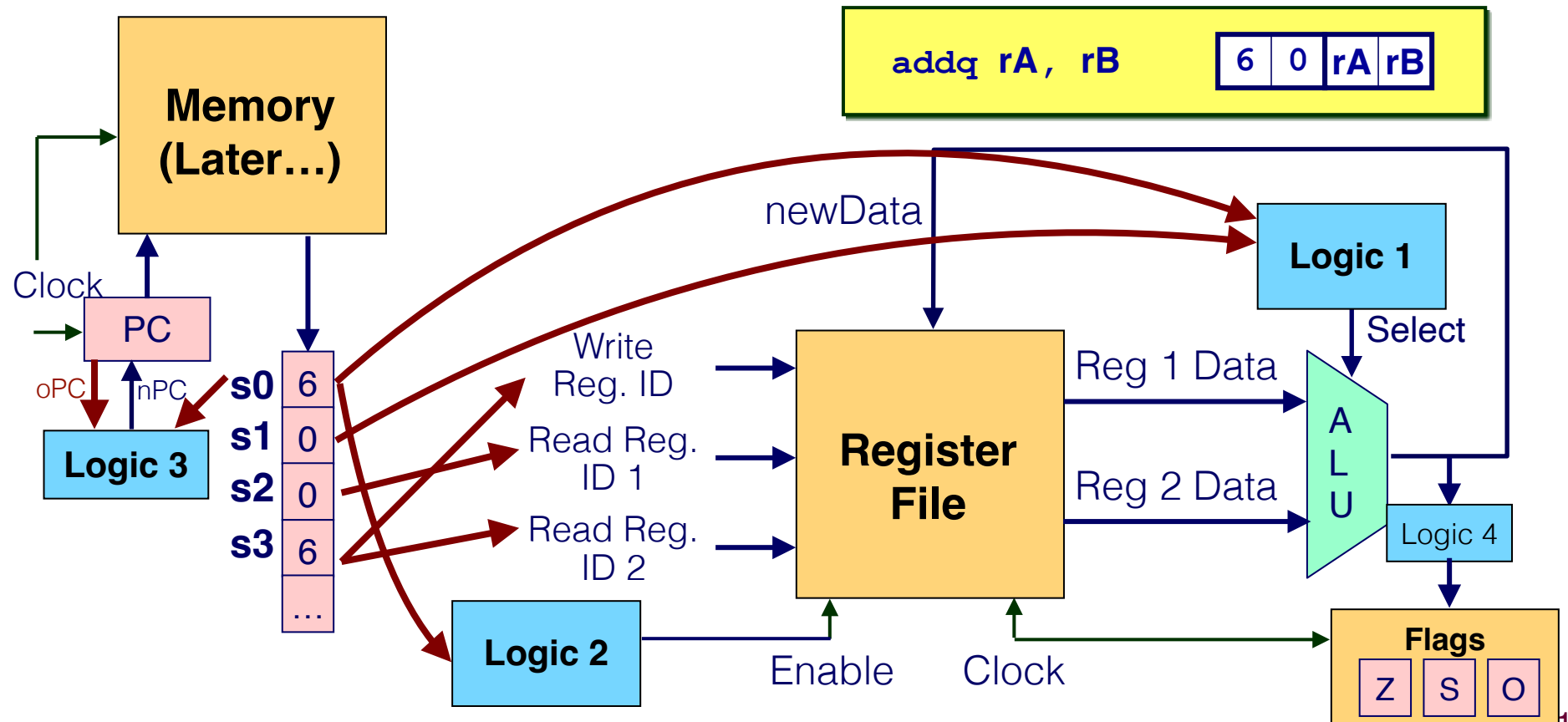
# Executing an ADD instruction

- 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;



# Executing an ADD instruction

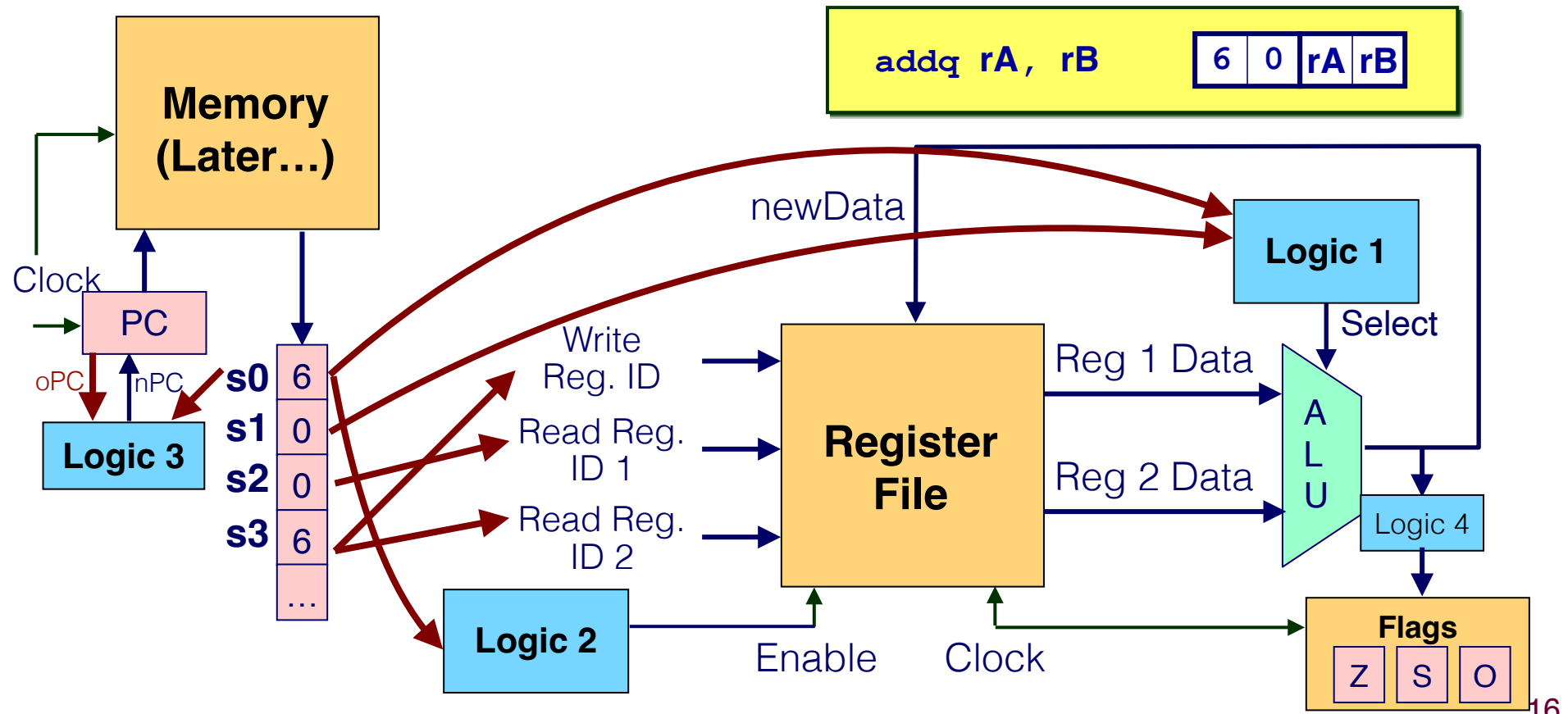
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?



# 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?

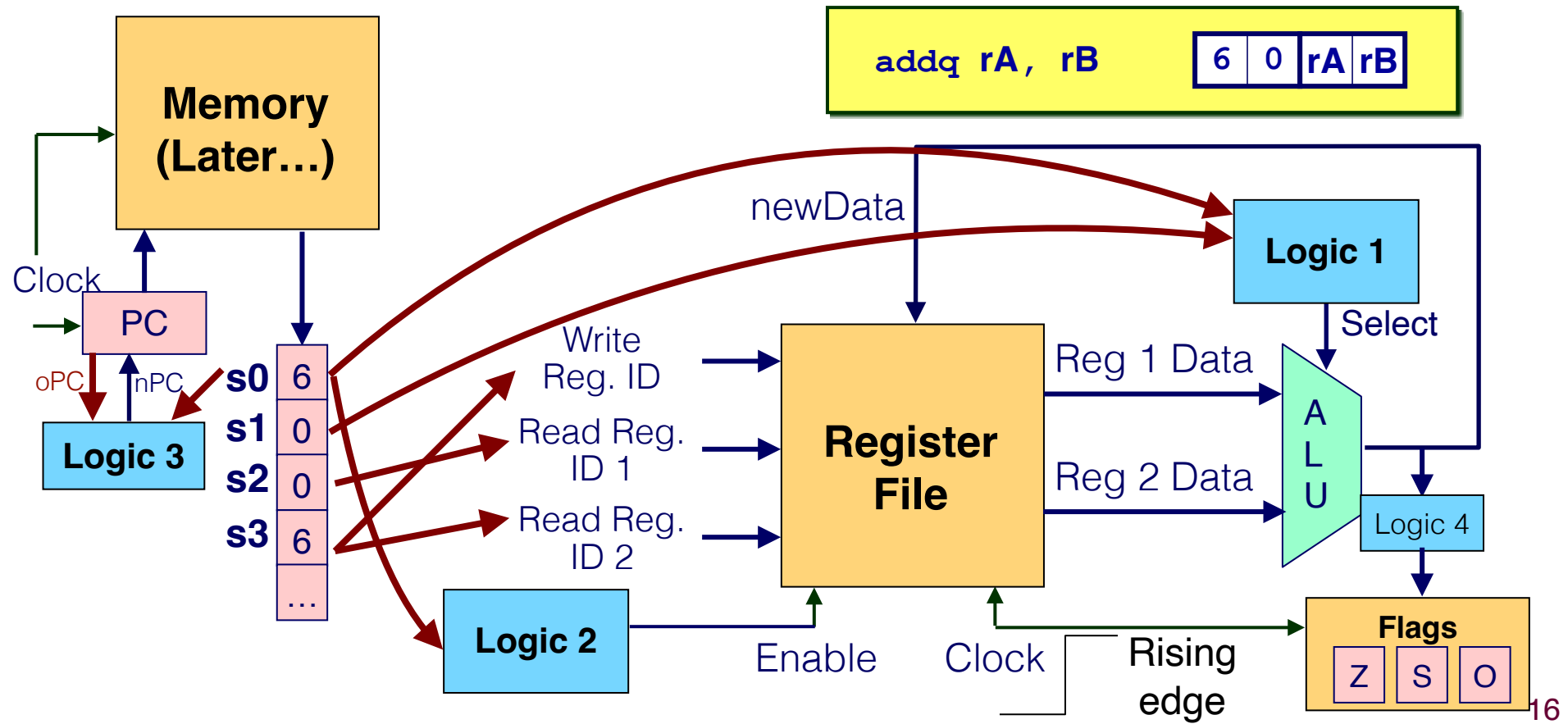
How do these logics get implemented?



# Executing an ADD instruction

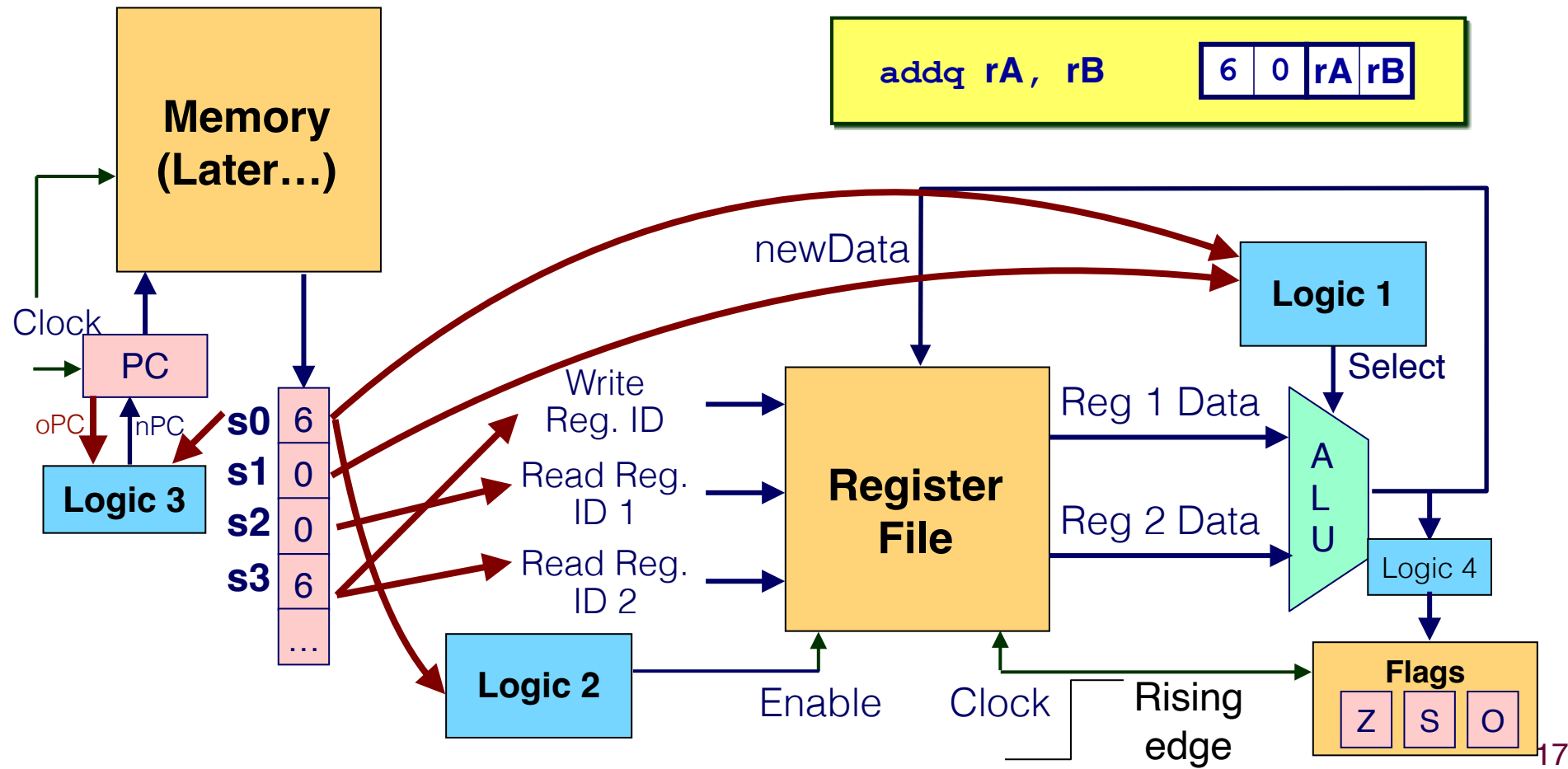
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

How do these logics get implemented?

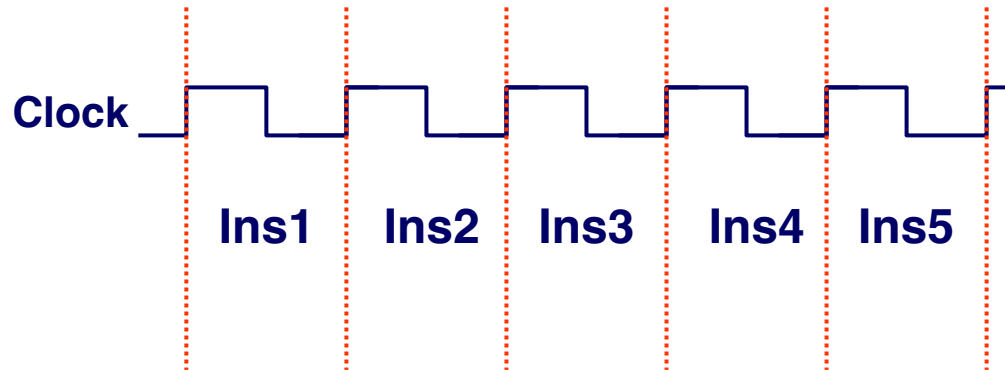


# Executing an ADD instruction

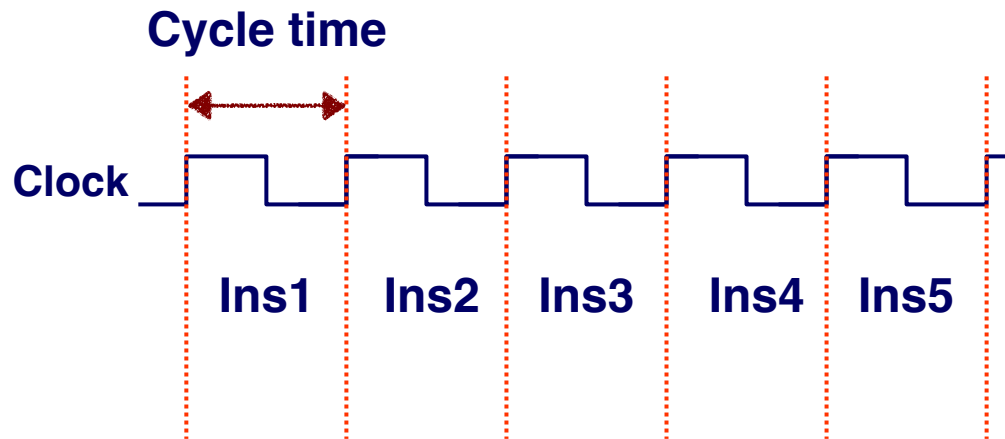
- When the rising edge of the clock arrives, the RF/PC/Flags will be written.
- So the following has to be ready: newData, nPC, which means Logic1, Logic2, Logic3, and Logic4 has to finish.



# Executing many ADD instructions



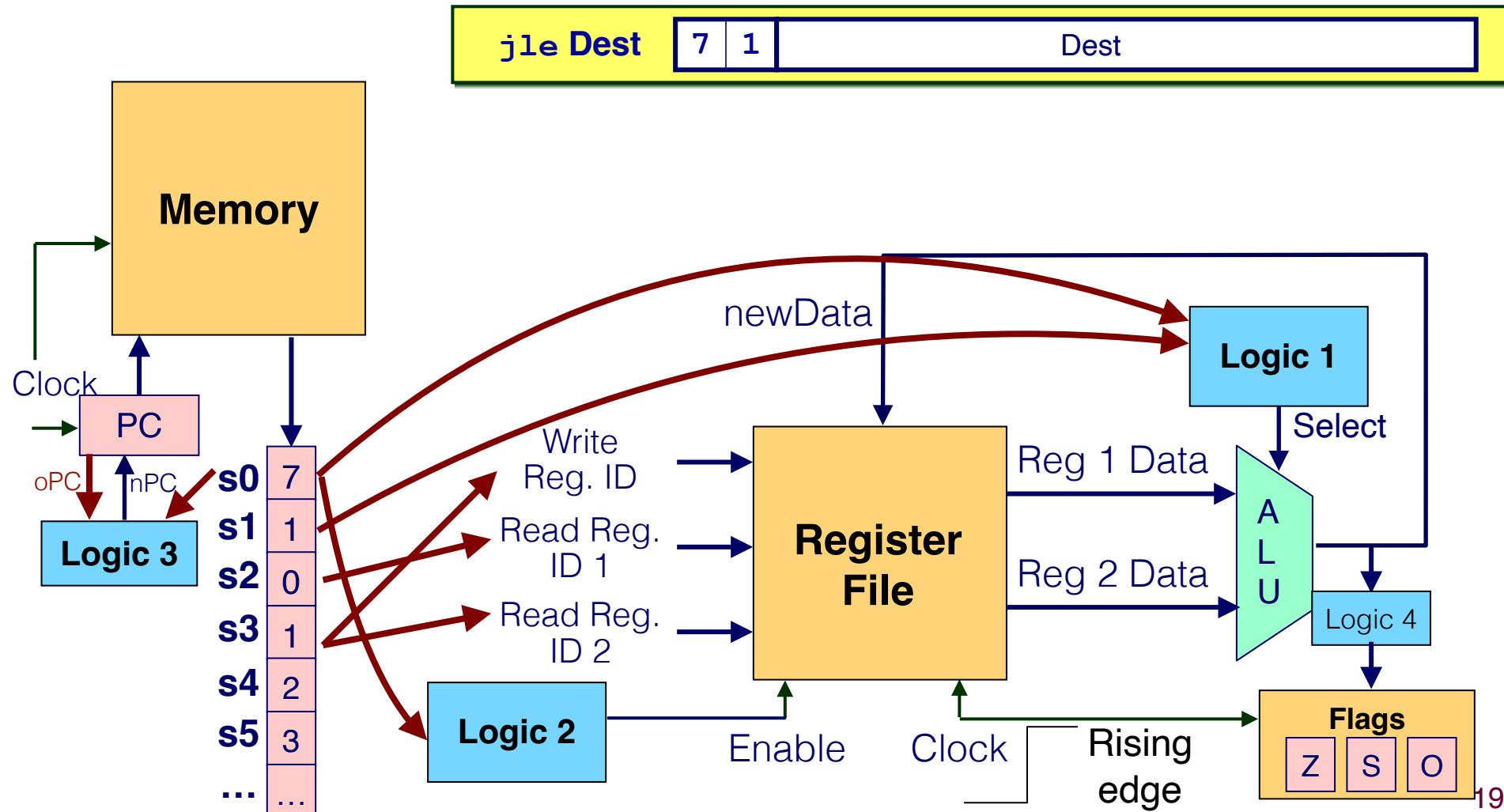
# Executing many ADD instructions



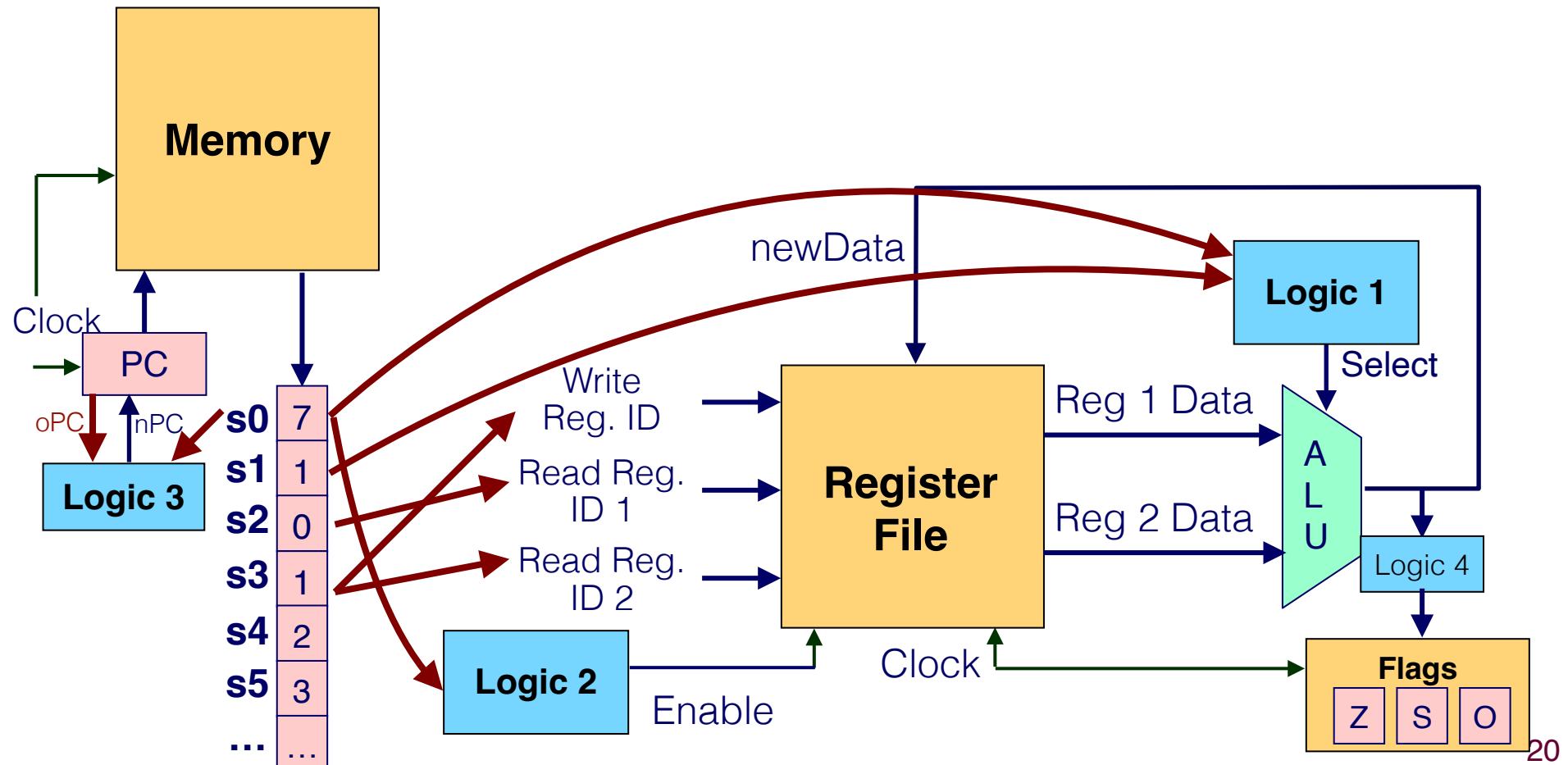


# Executing a JLE instruction

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

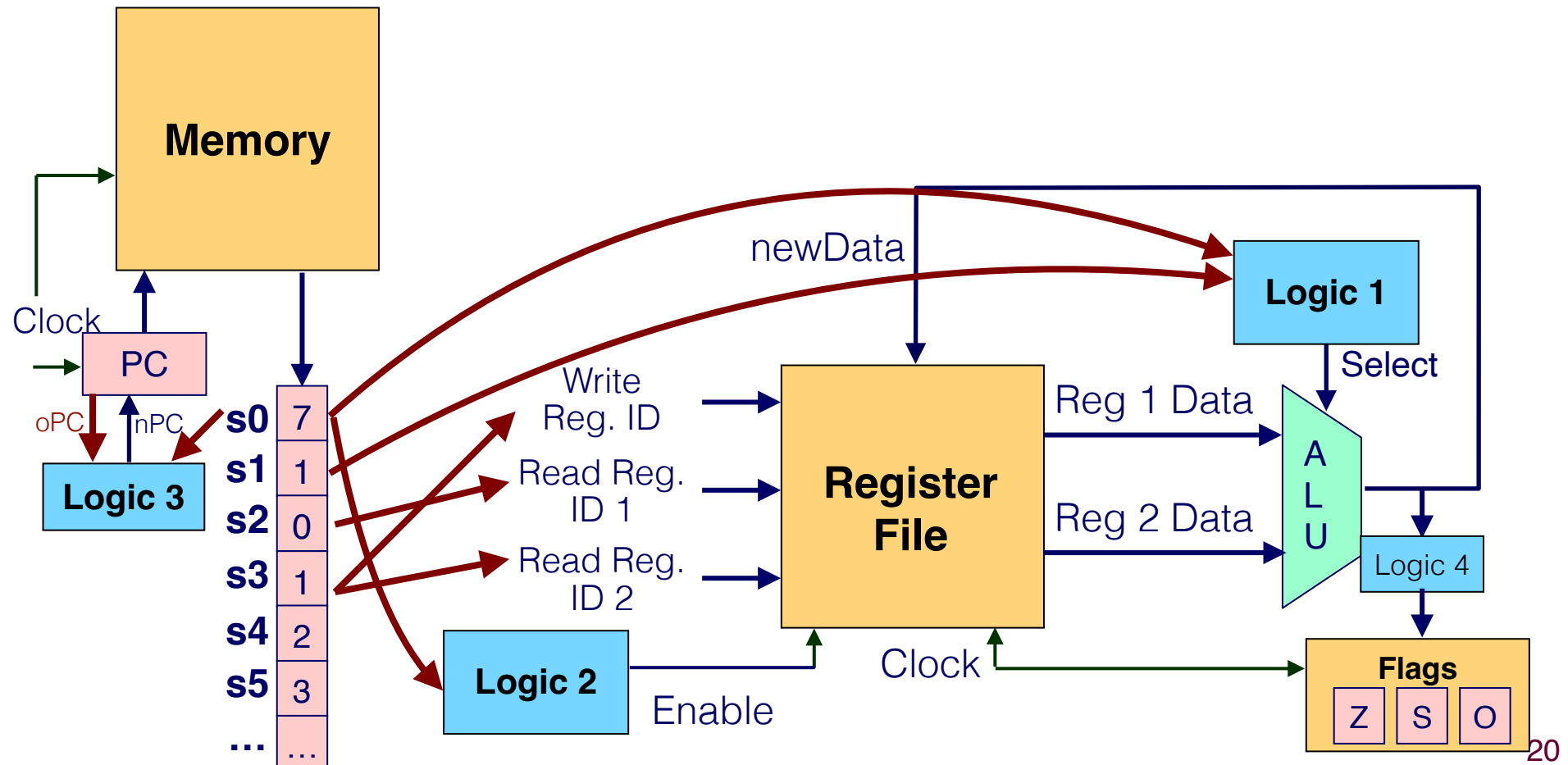


# Executing a JLE instruction



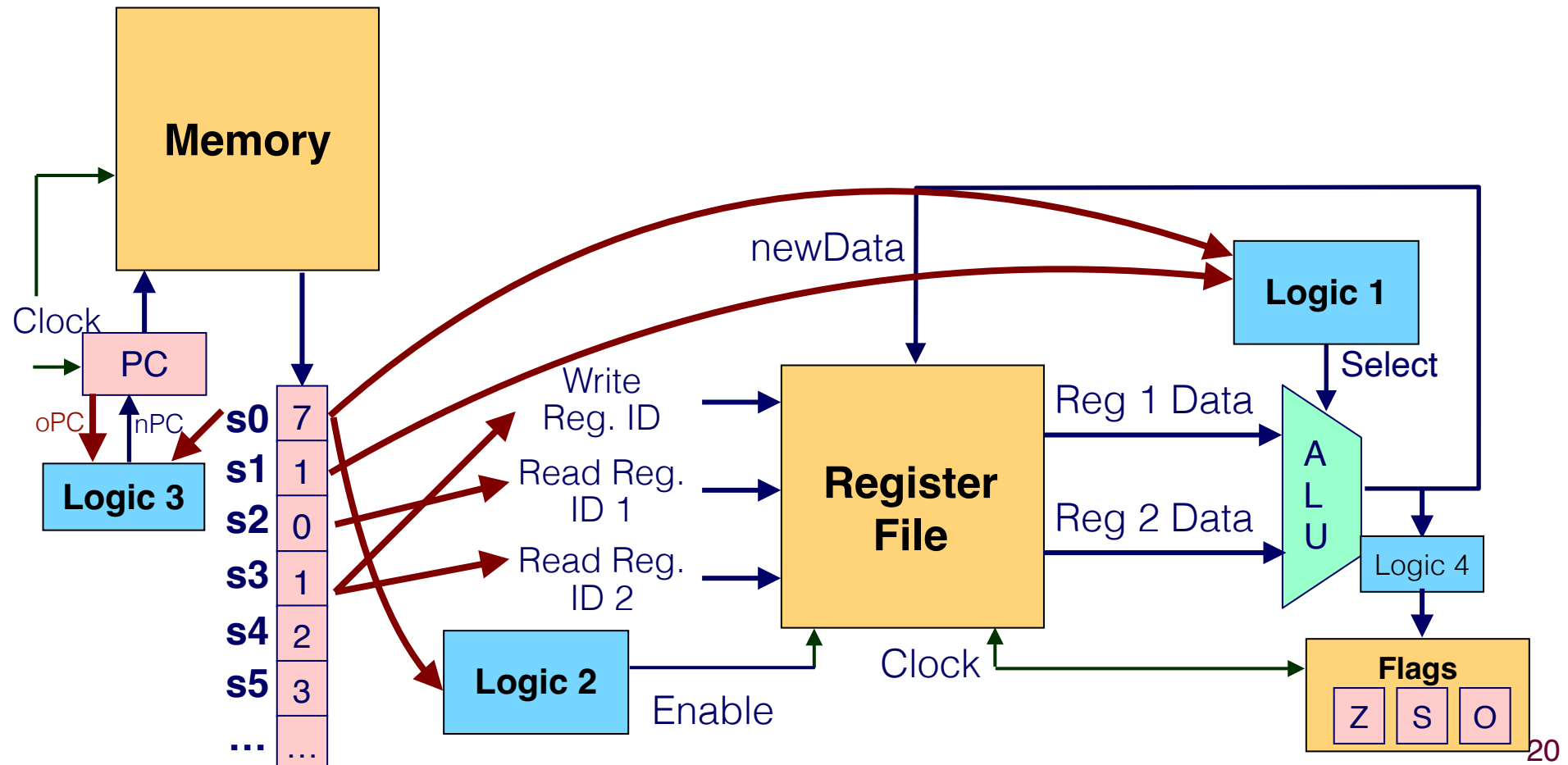
# Executing a JLE instruction

- Logic 1: if (s0 == 6) select = s1;



# Executing a JLE instruction

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



jle Dest

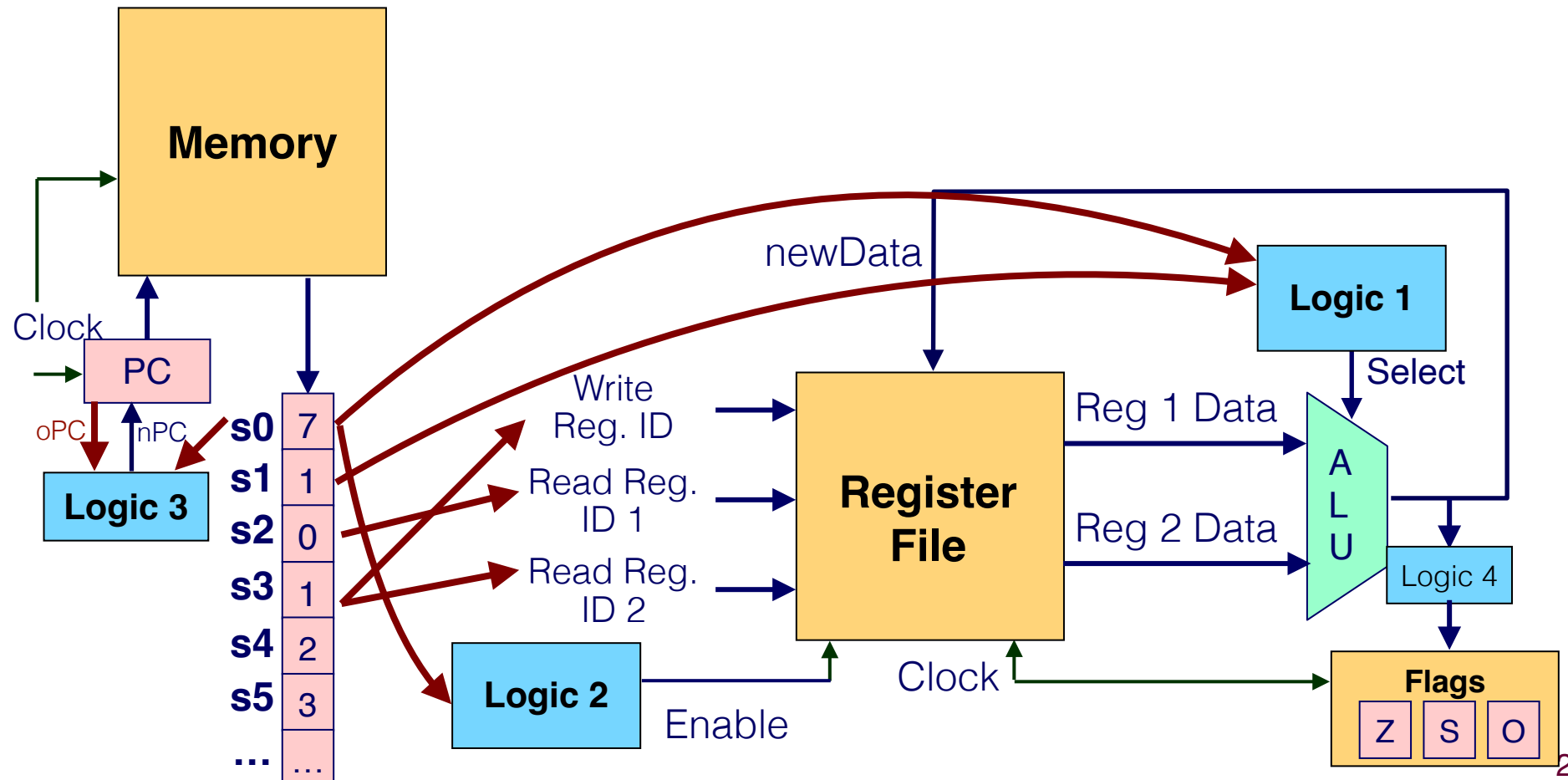
7

1

Dest

# Executing a JLE instruction

- Logic 3??



jle Dest

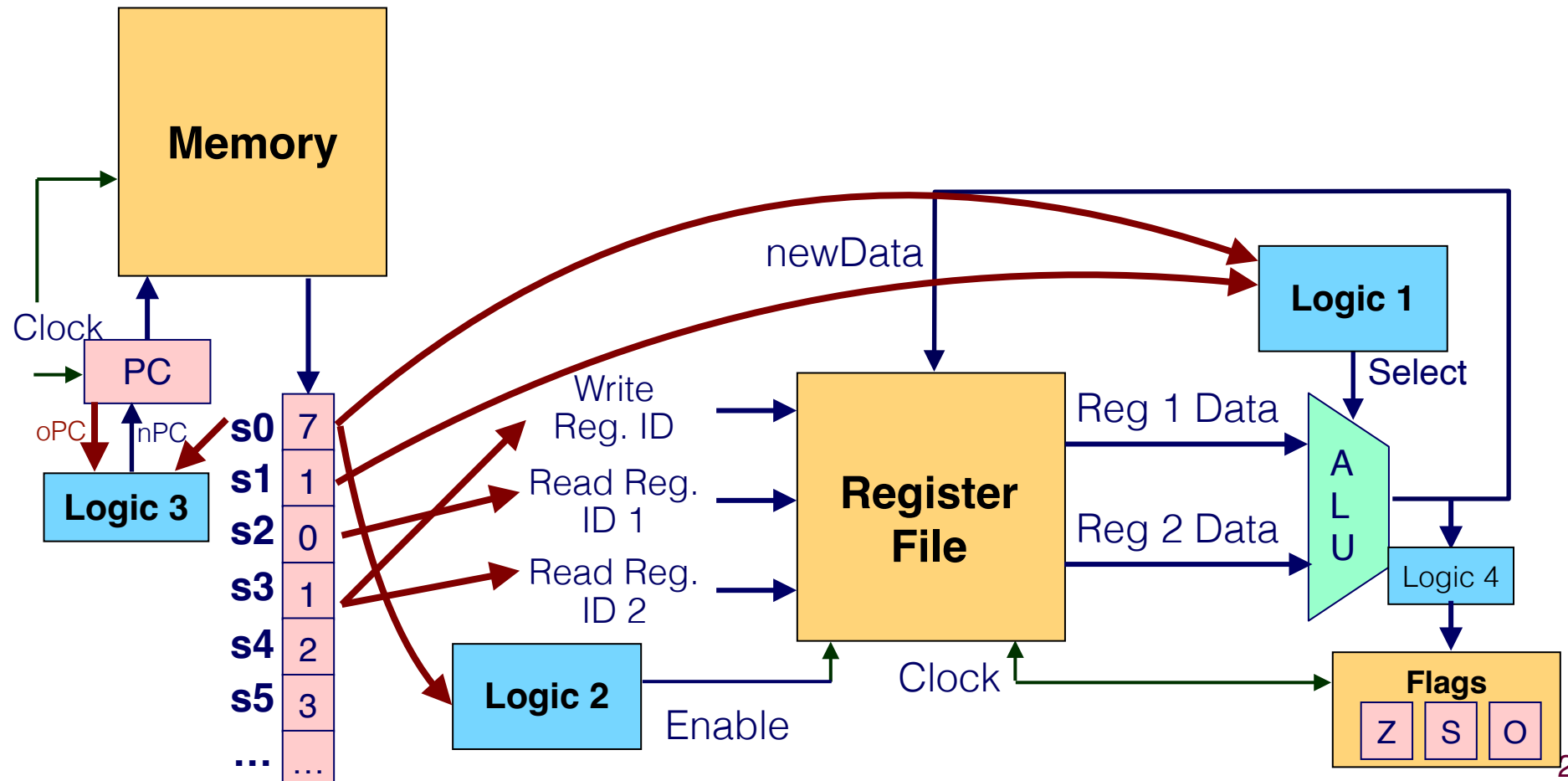
7

1

Dest

# Executing a JLE instruction

- Logic 3??  $\text{if } (s0 == 6) \text{ nPC} = \text{oPC} + 2;$



jle Dest

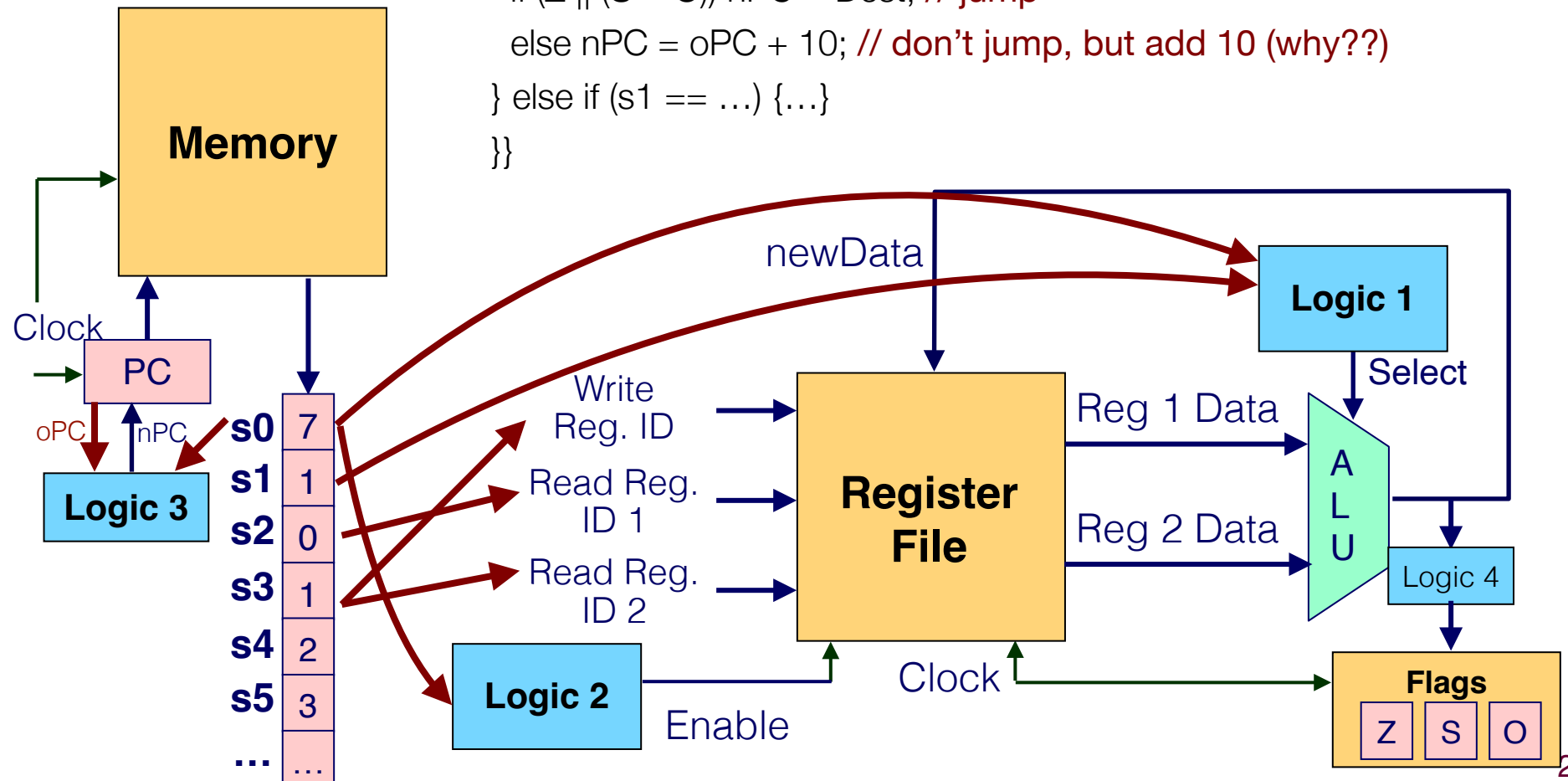
7 1

Dest

# Executing a JLE instruction

- Logic 3??

```
if (s0 == 6) nPC = oPC + 2;  
else if (s0 == 7) {  
  if (s1 == 1) { // jLE  
    if (Z || (S ^ O)) nPC = Dest; // jump  
    else nPC = oPC + 10; // don't jump, but add 10 (why??)  
  } else if (s1 == ...) {...}  
}
```



jle Dest

7

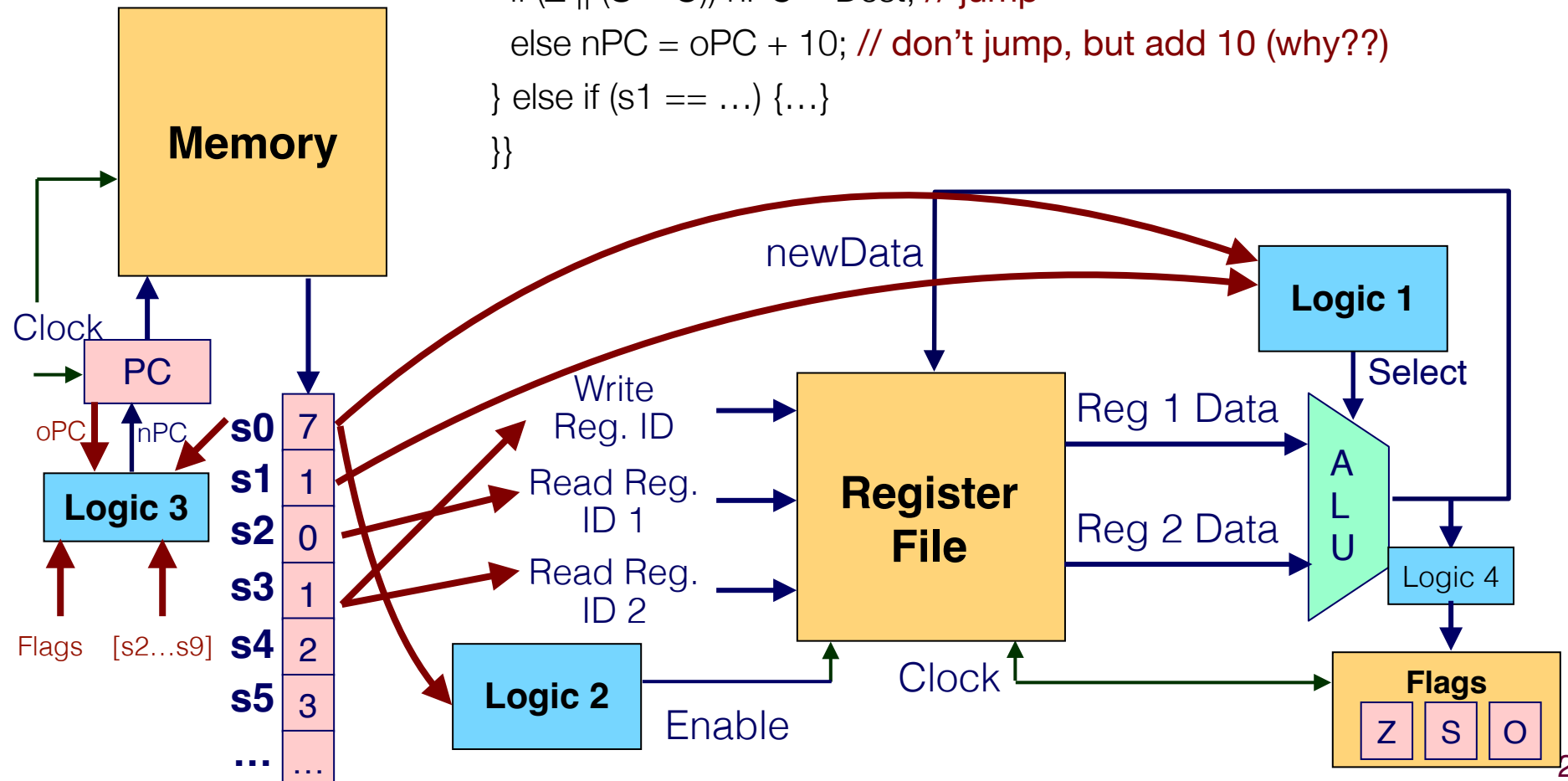
1

Dest

# Executing a JLE instruction

- Logic 3??

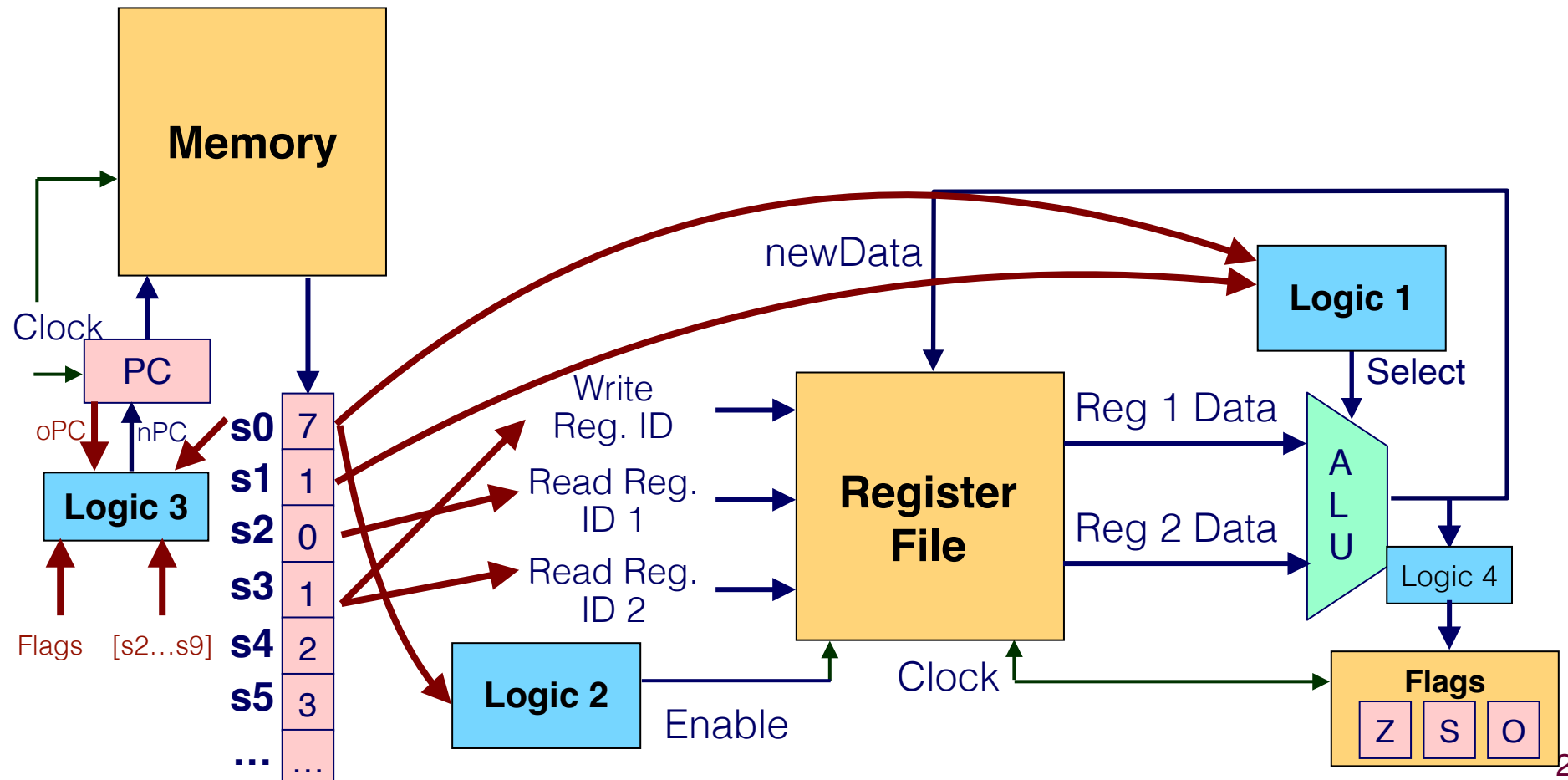
```
if (s0 == 6) nPC = oPC + 2;  
else if (s0 == 7) {  
  if (s1 == 1) { // jLE  
    if (Z || (S ^ O)) nPC = Dest; // jump  
    else nPC = oPC + 10; // don't jump, but add 10 (why??)  
  } else if (s1 == ...) {...}  
}
```





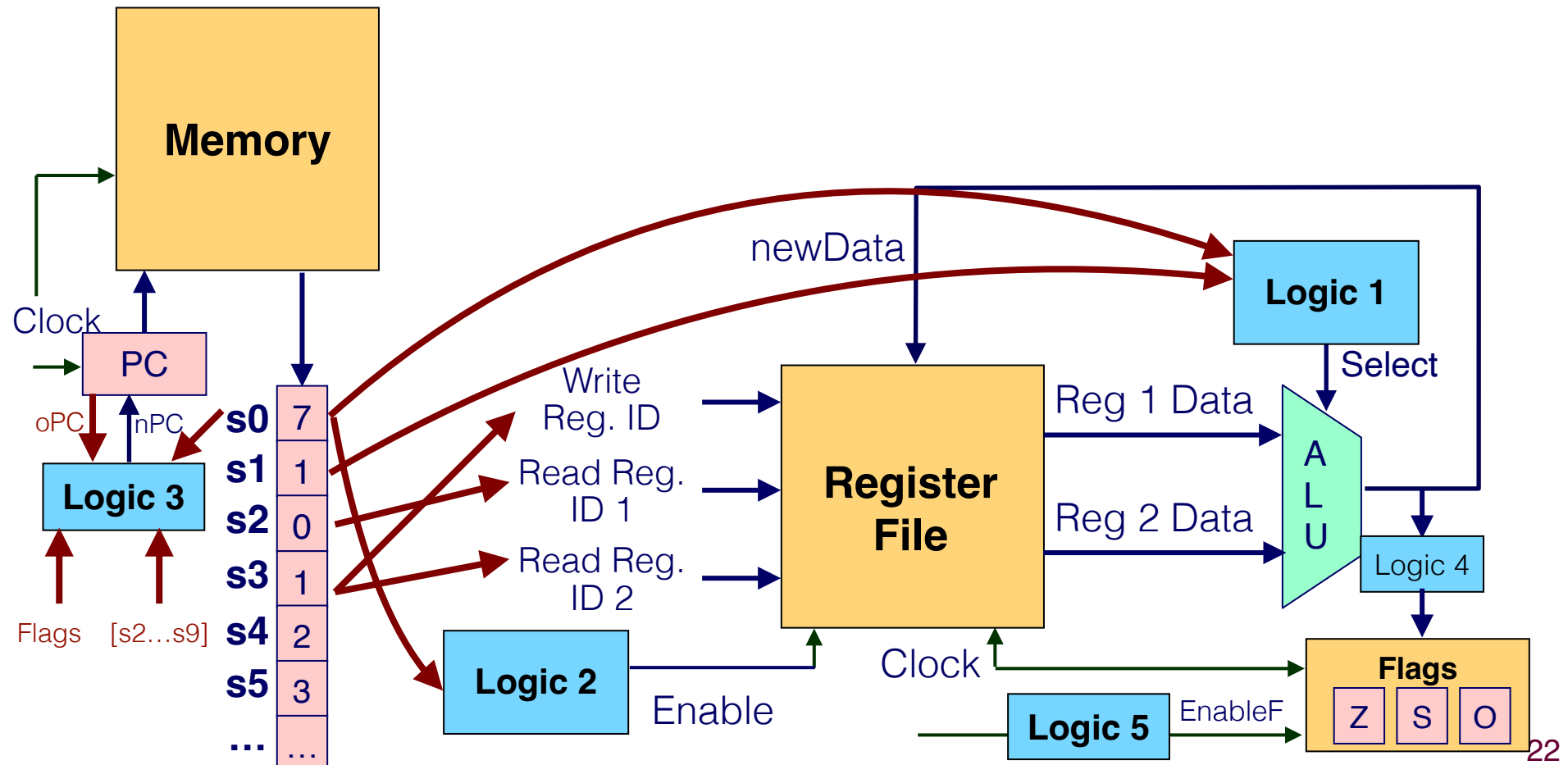
# Executing a JLE instruction

- Logic 4? Does JLE write flags?



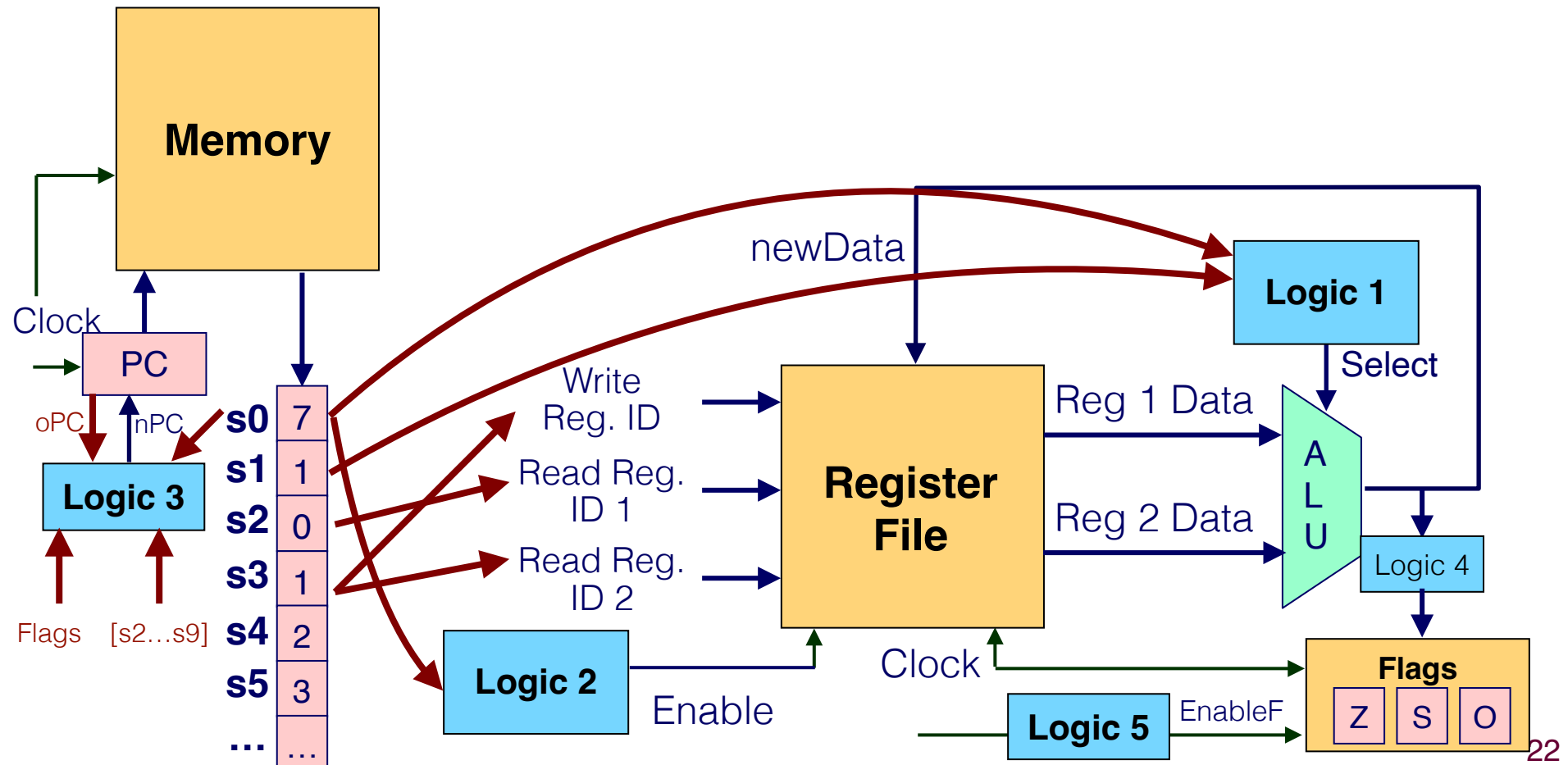
# Executing a JLE instruction

- Logic 4? Does JLE write flags?
- Need another piece of logic.

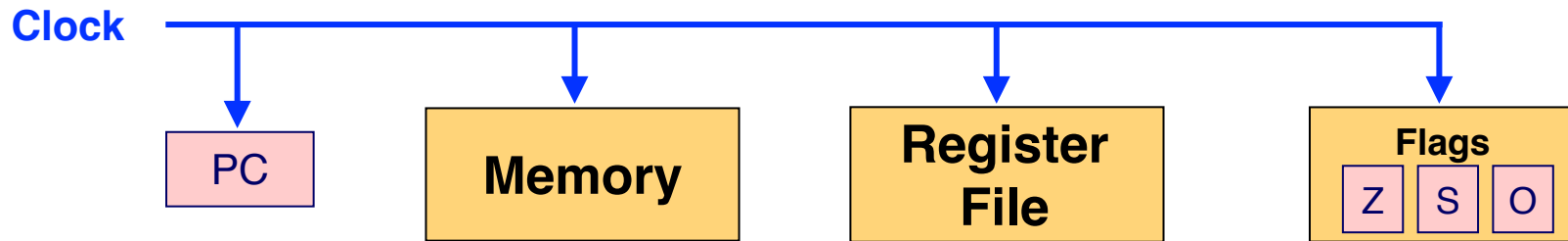


# Executing a JLE instruction

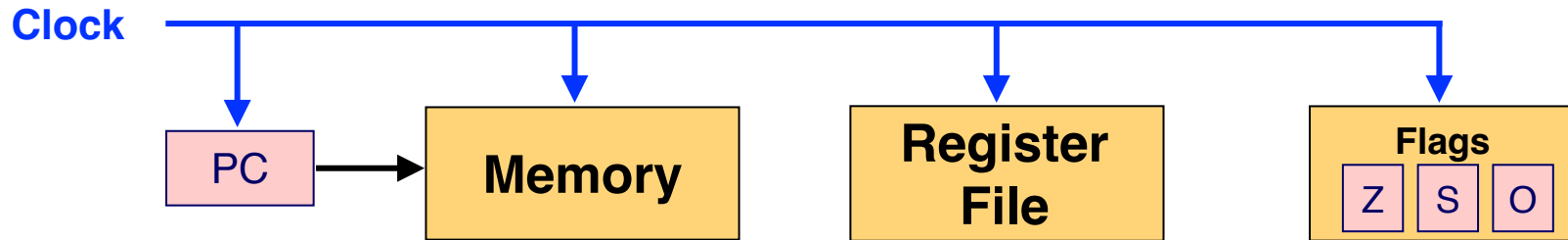
- Logic 4? Does JLE write flags?
- Need another piece of logic.
- Logic 5: if (s0 == 7) EnableF = 0; else if (s0 == 6) EnableF = 1;



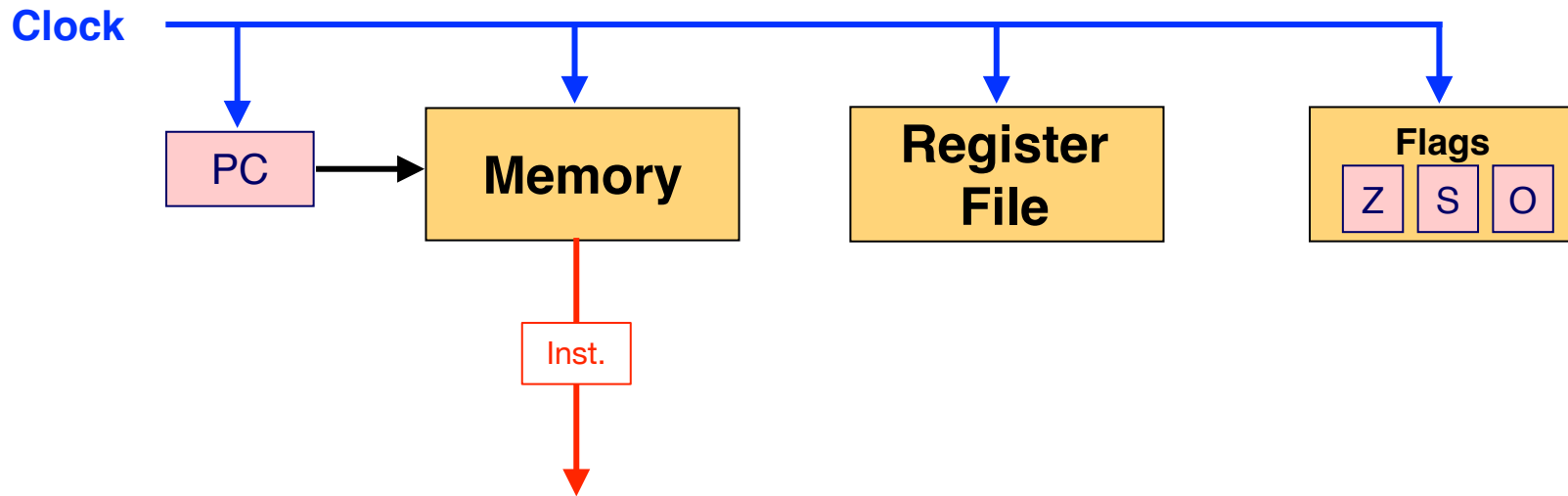
# Microarchitecture (So far)



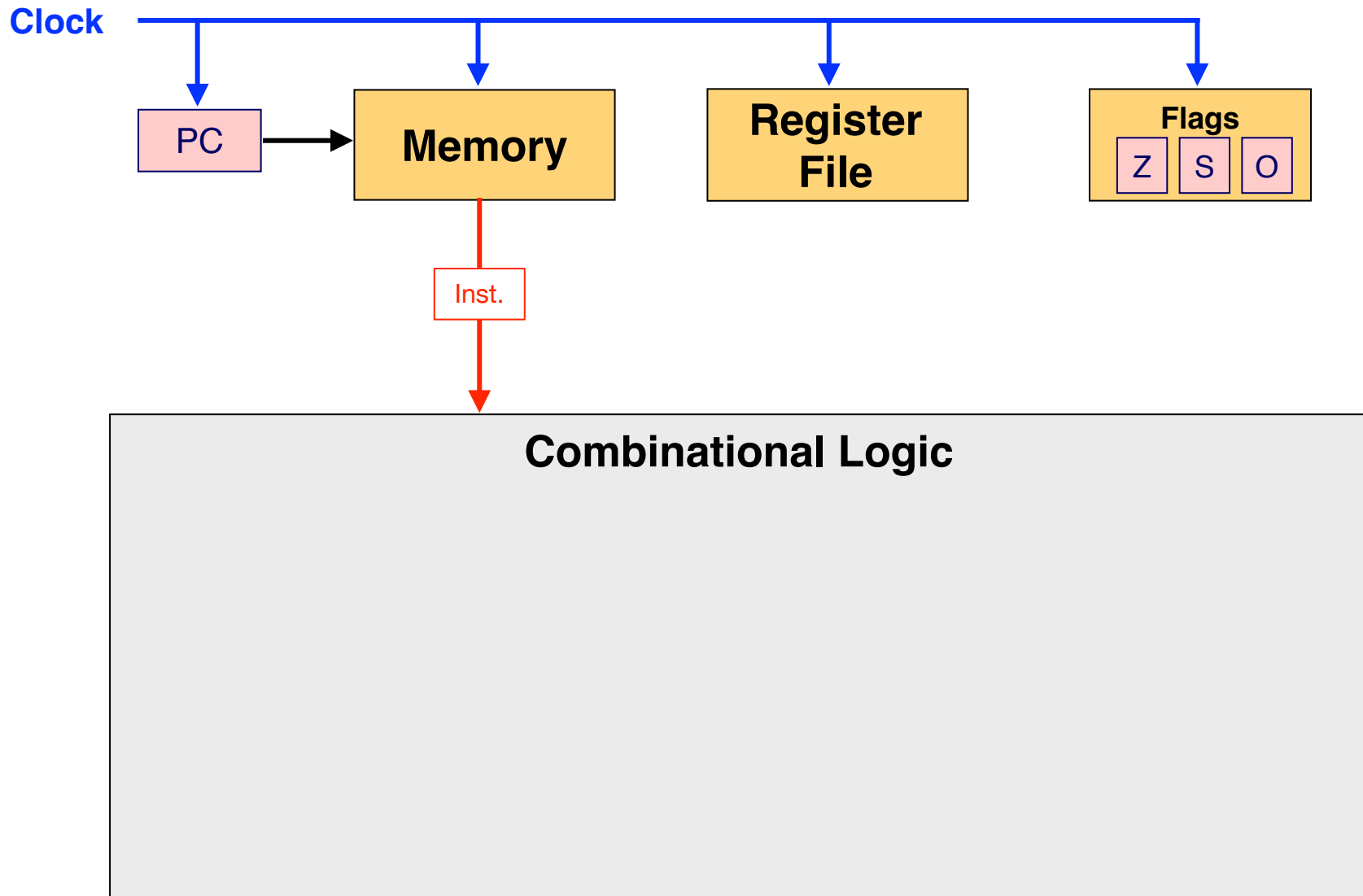
# Microarchitecture (So far)



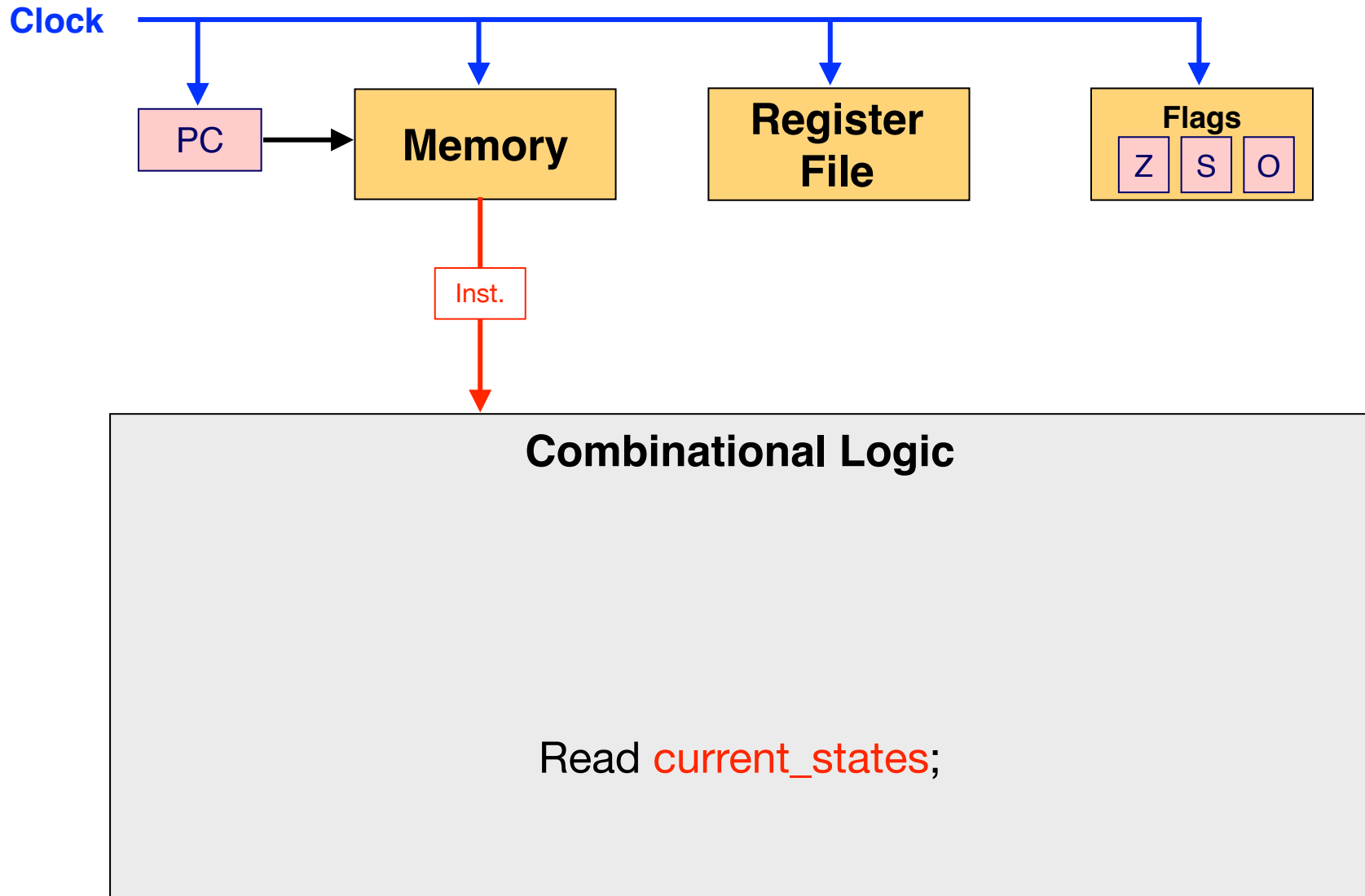
# Microarchitecture (So far)



# Microarchitecture (So far)

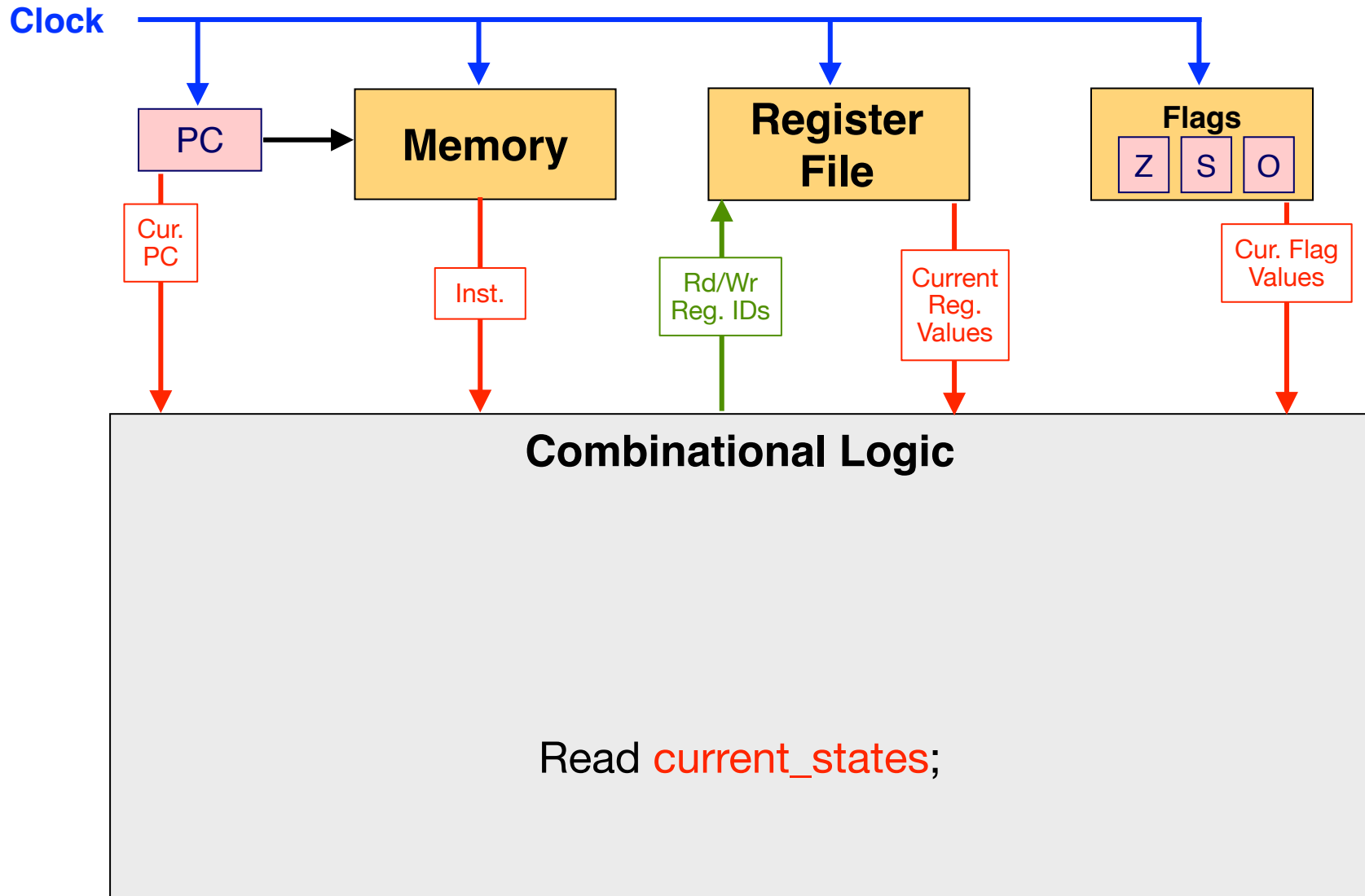


# Microarchitecture (So far)

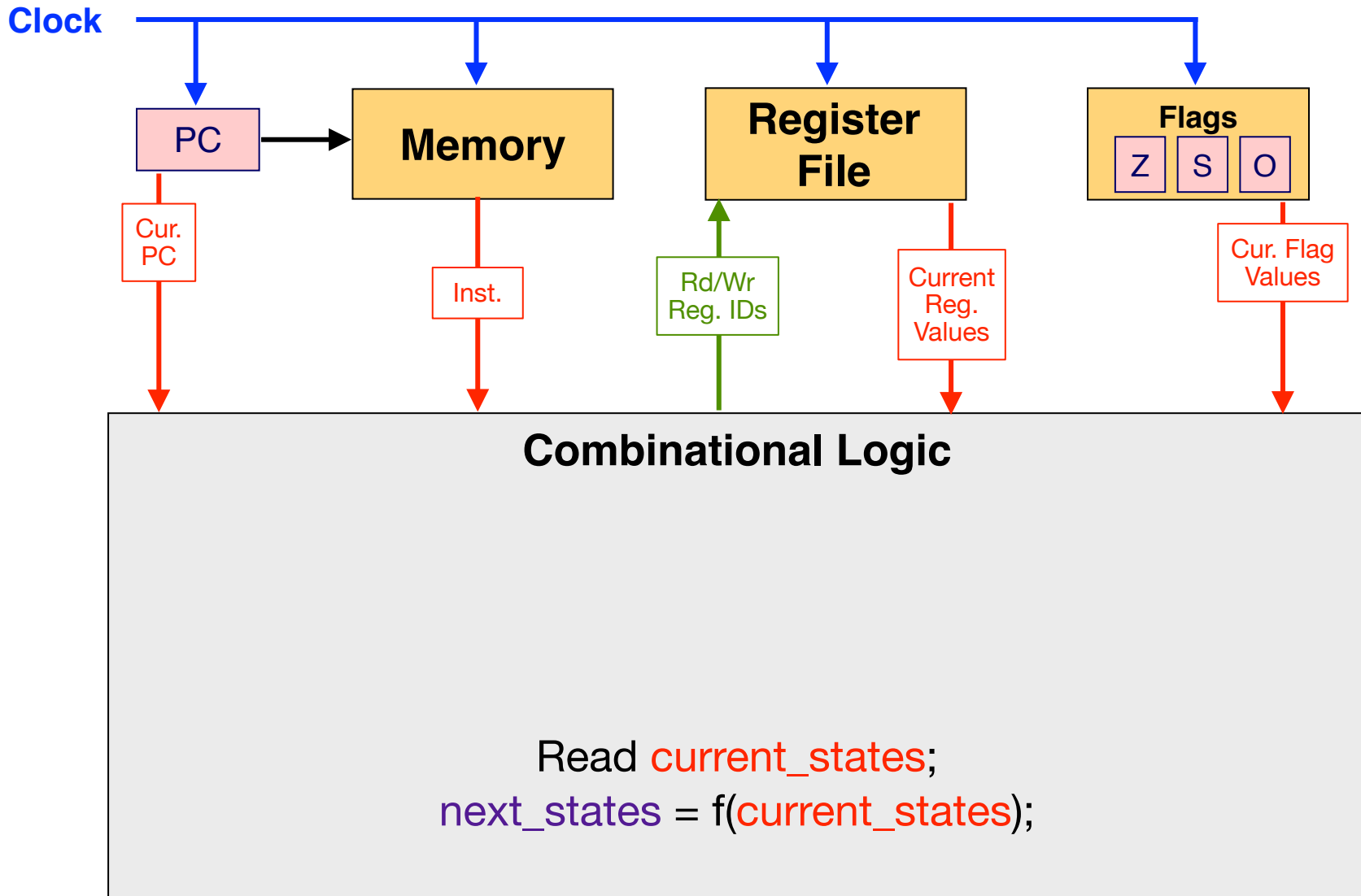




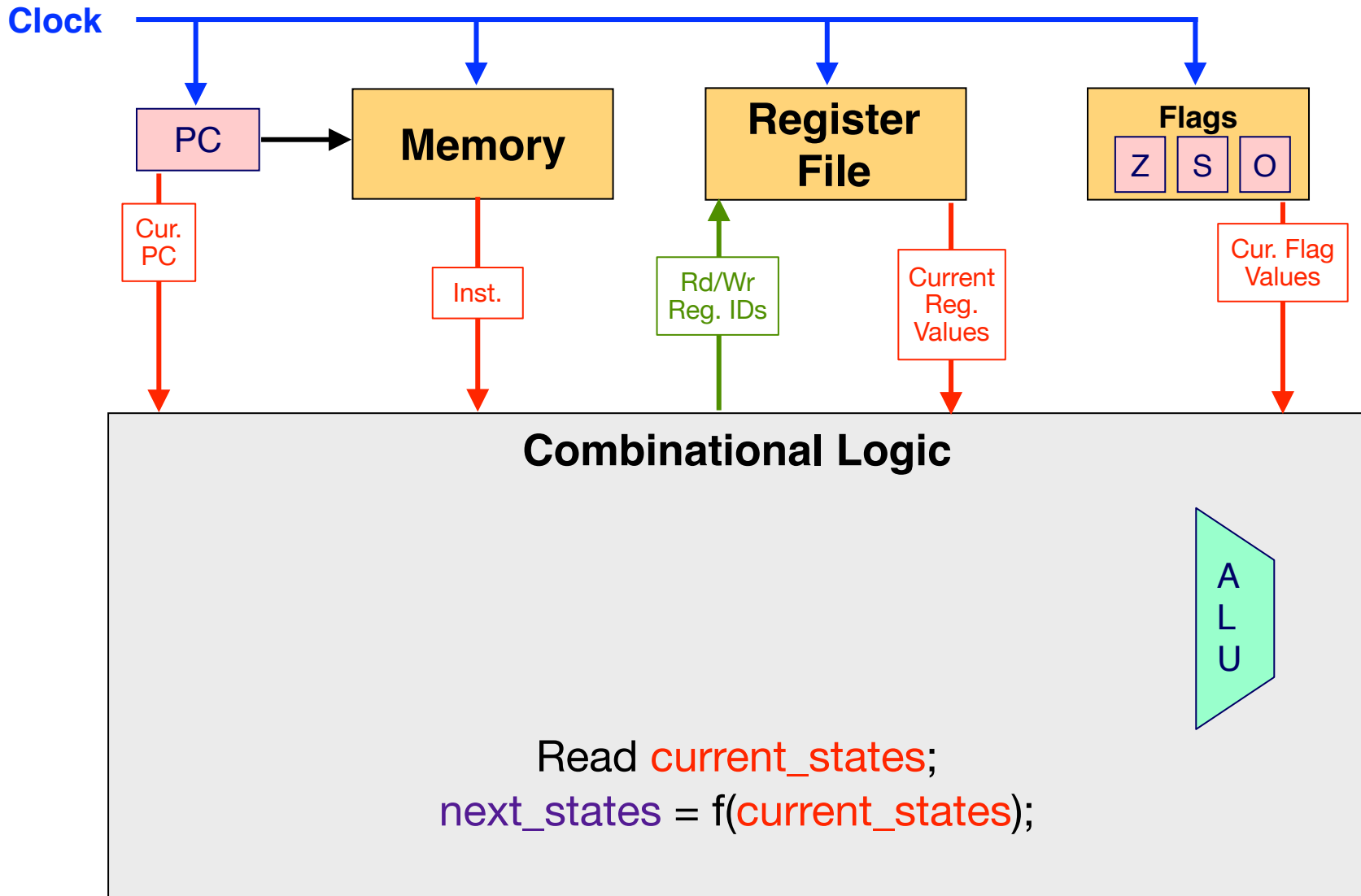
# Microarchitecture (So far)



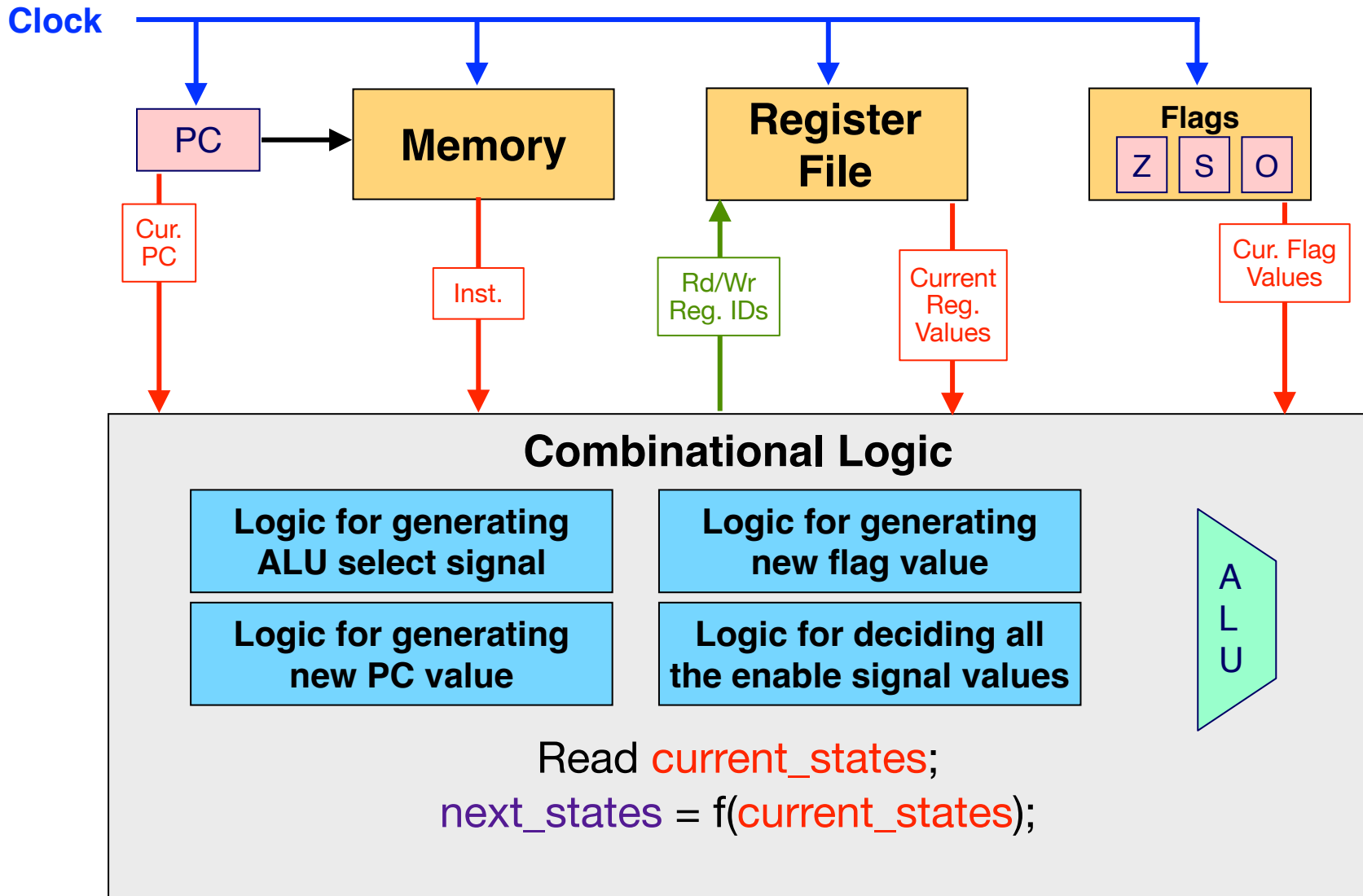
# Microarchitecture (So far)



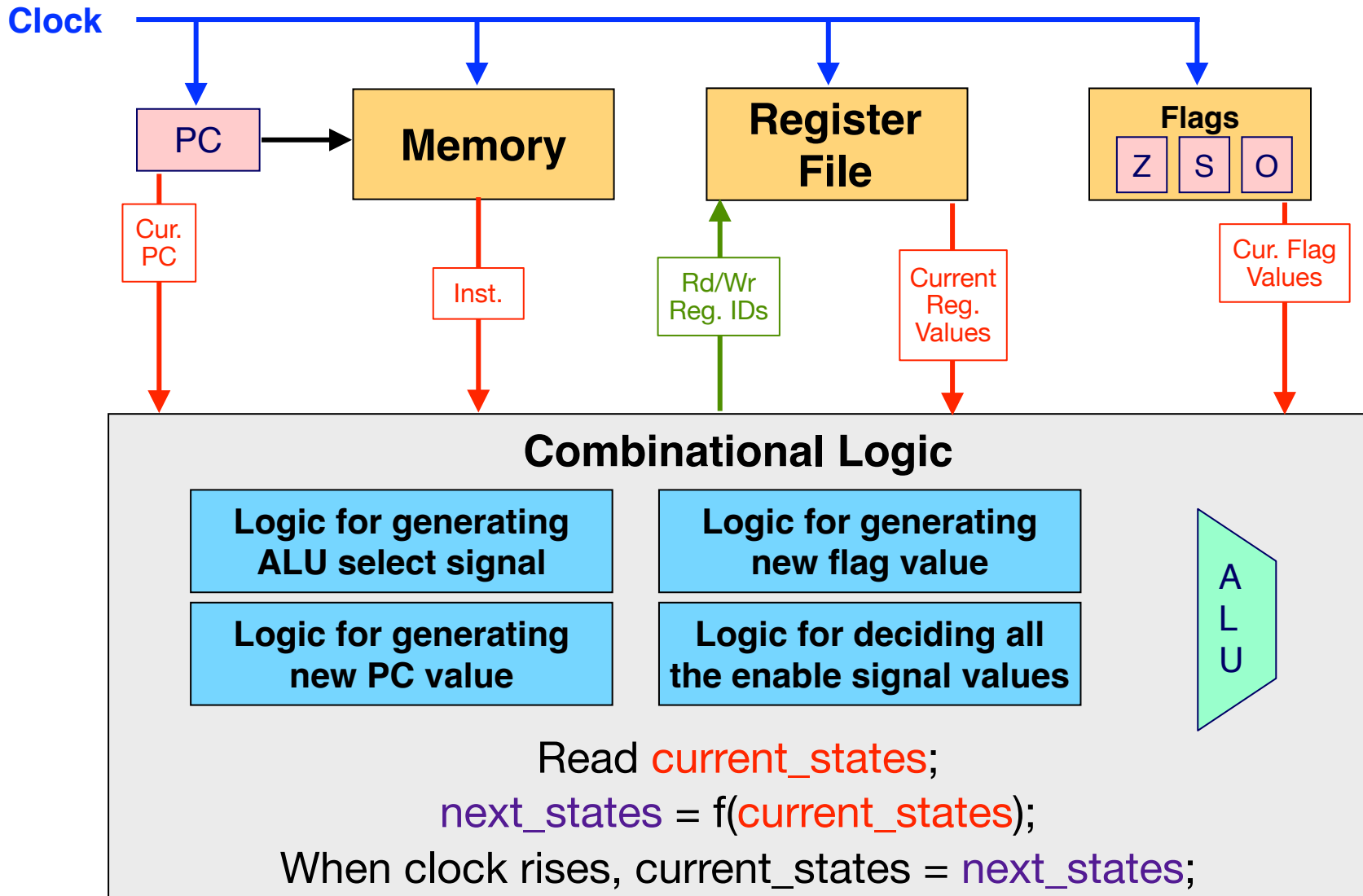
# Microarchitecture (So far)



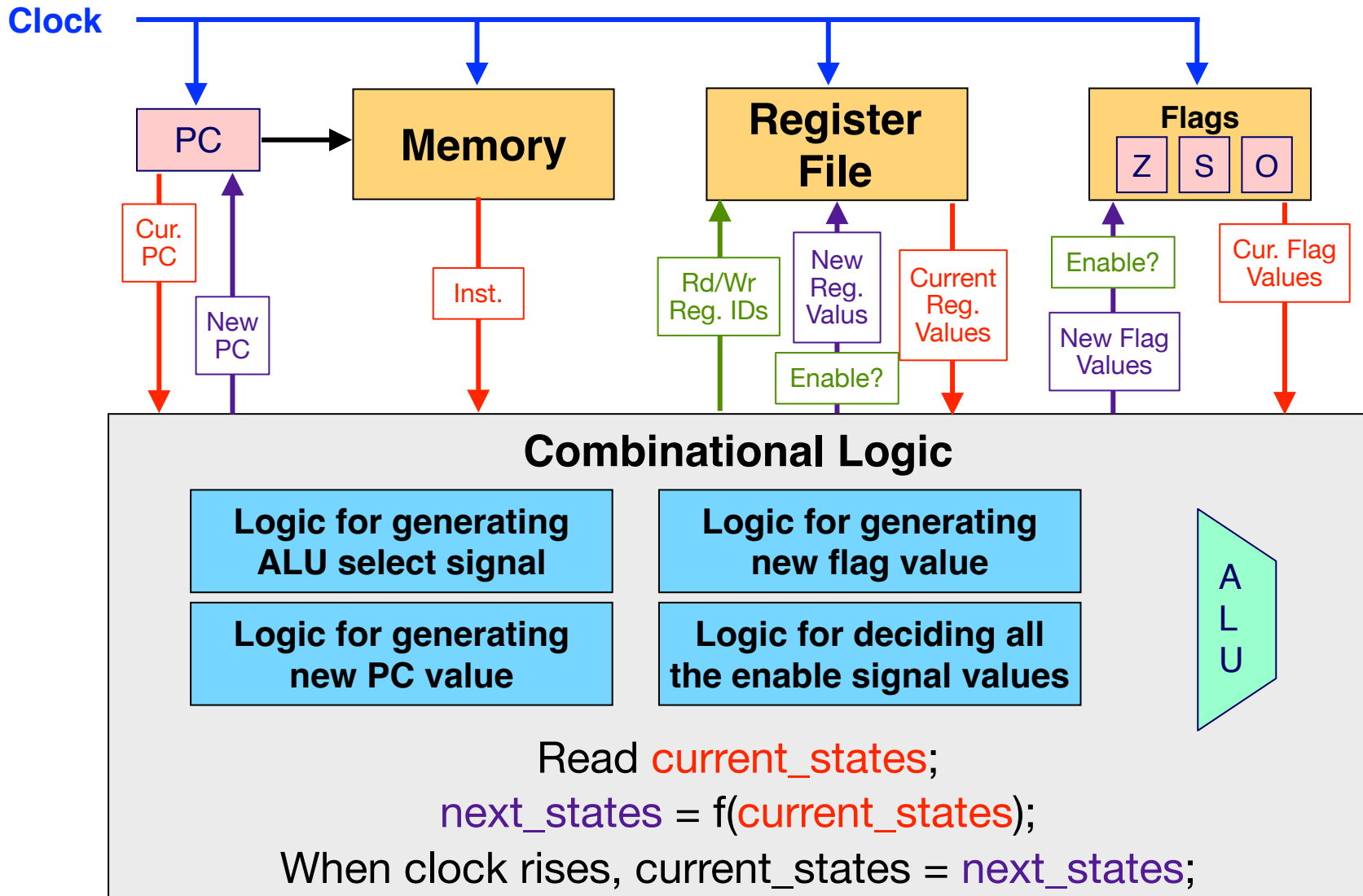
# Microarchitecture (So far)



# Microarchitecture (So far)

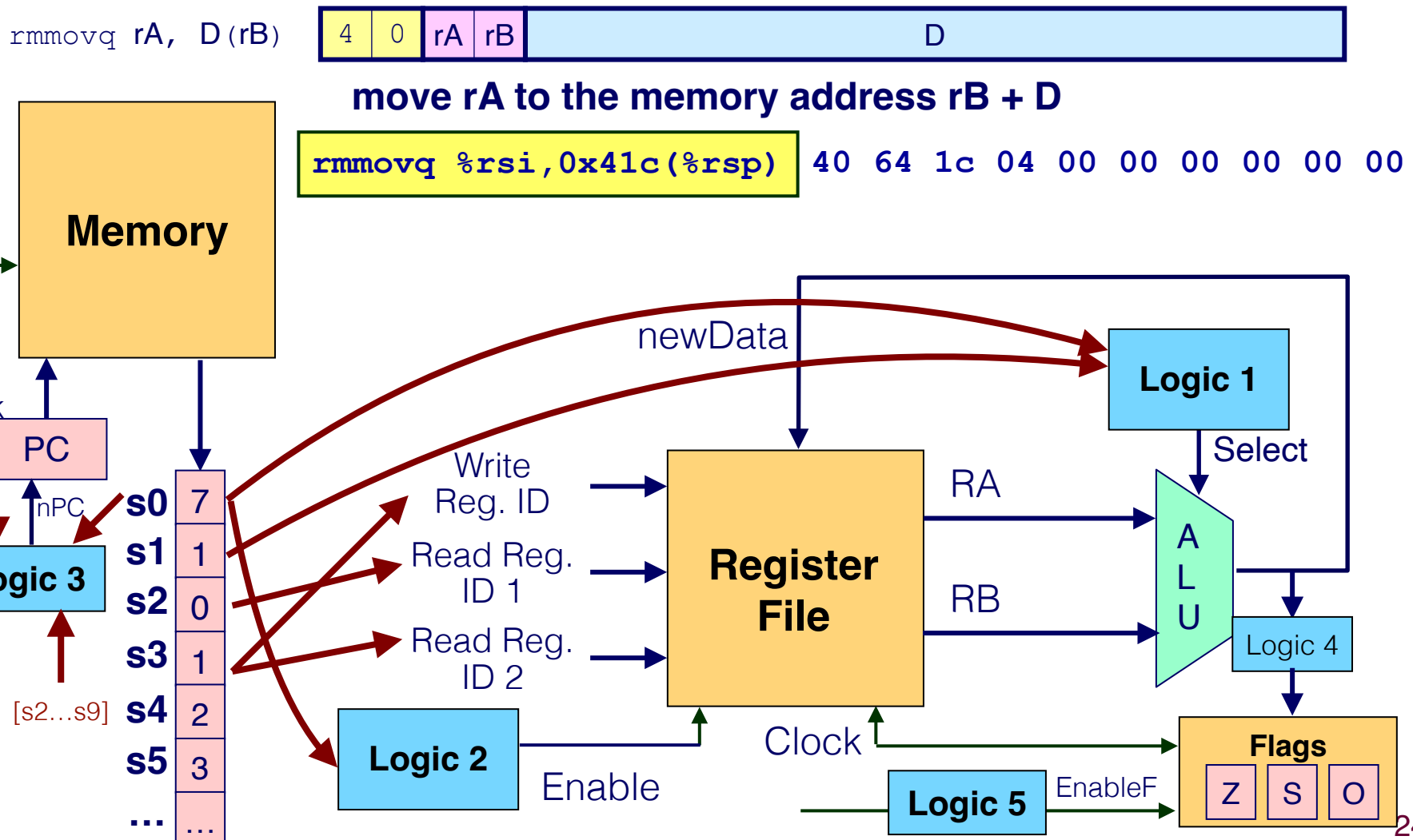


# Microarchitecture (So far)



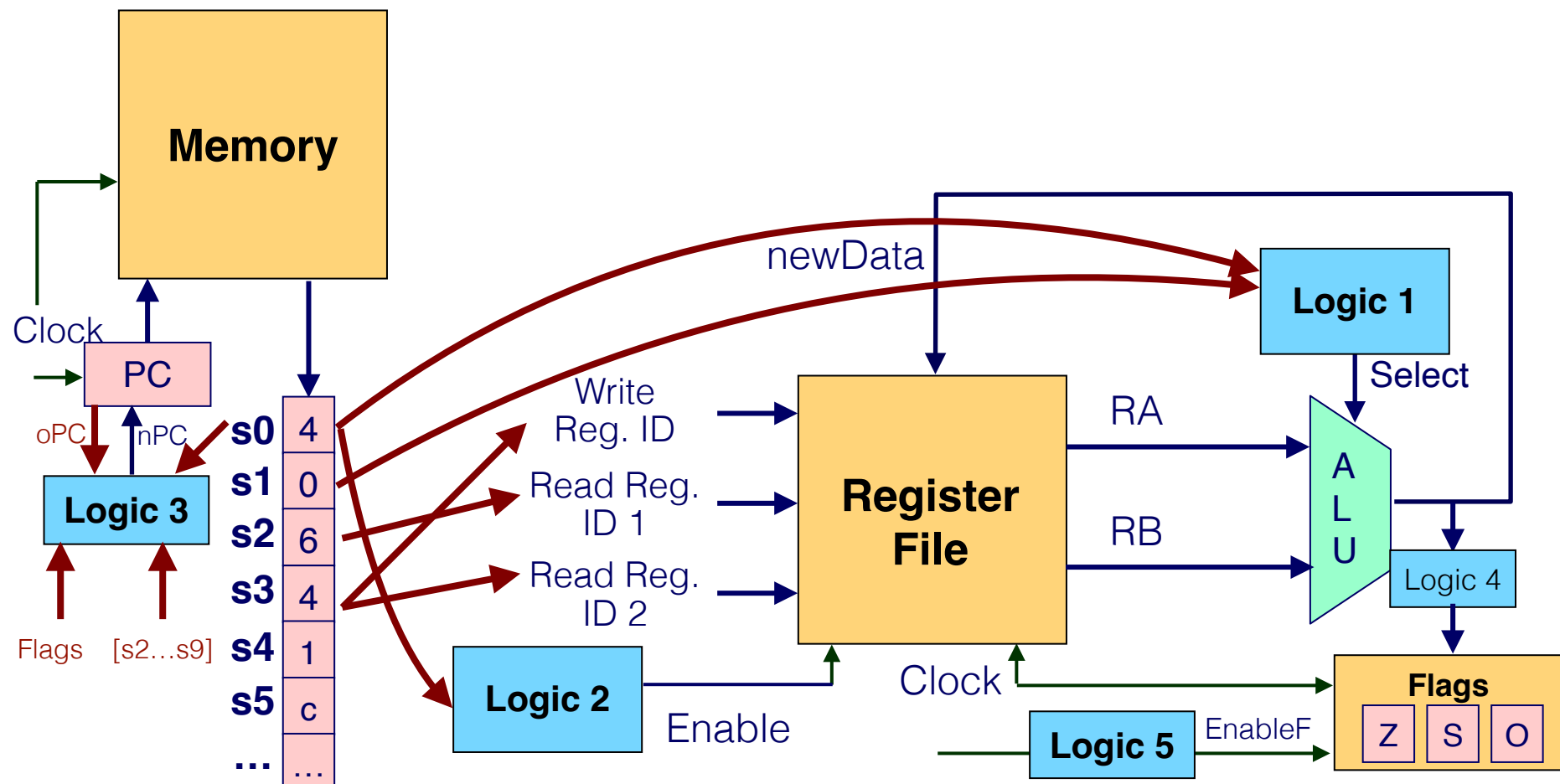
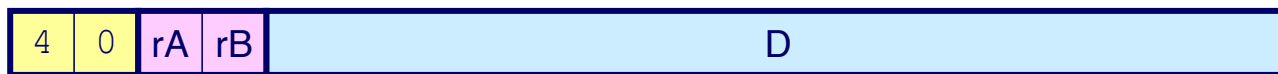
# Executing a MOV instruction

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



# move rA to the memory address rB + D

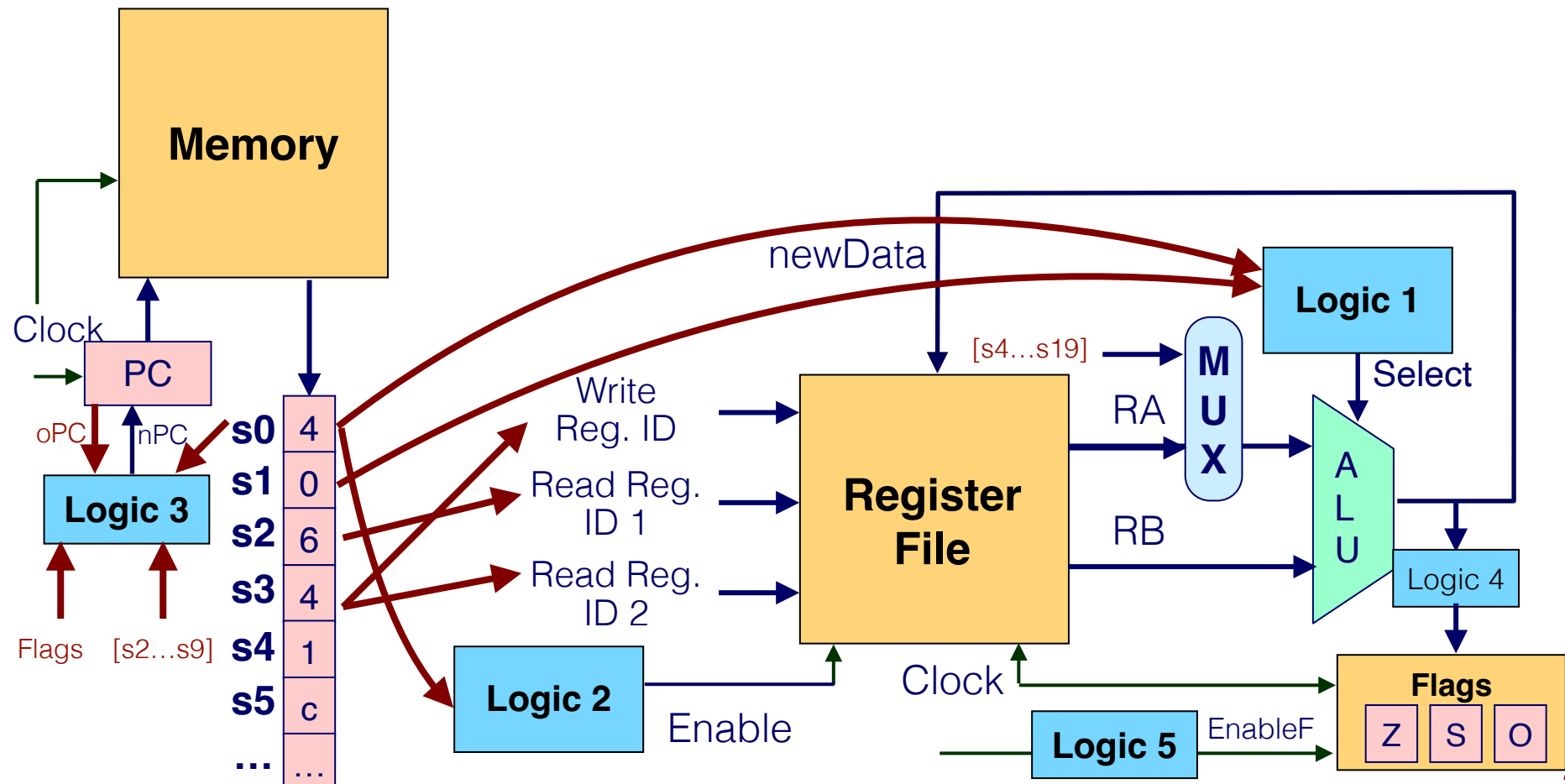
rmmovq rA, D(rB)





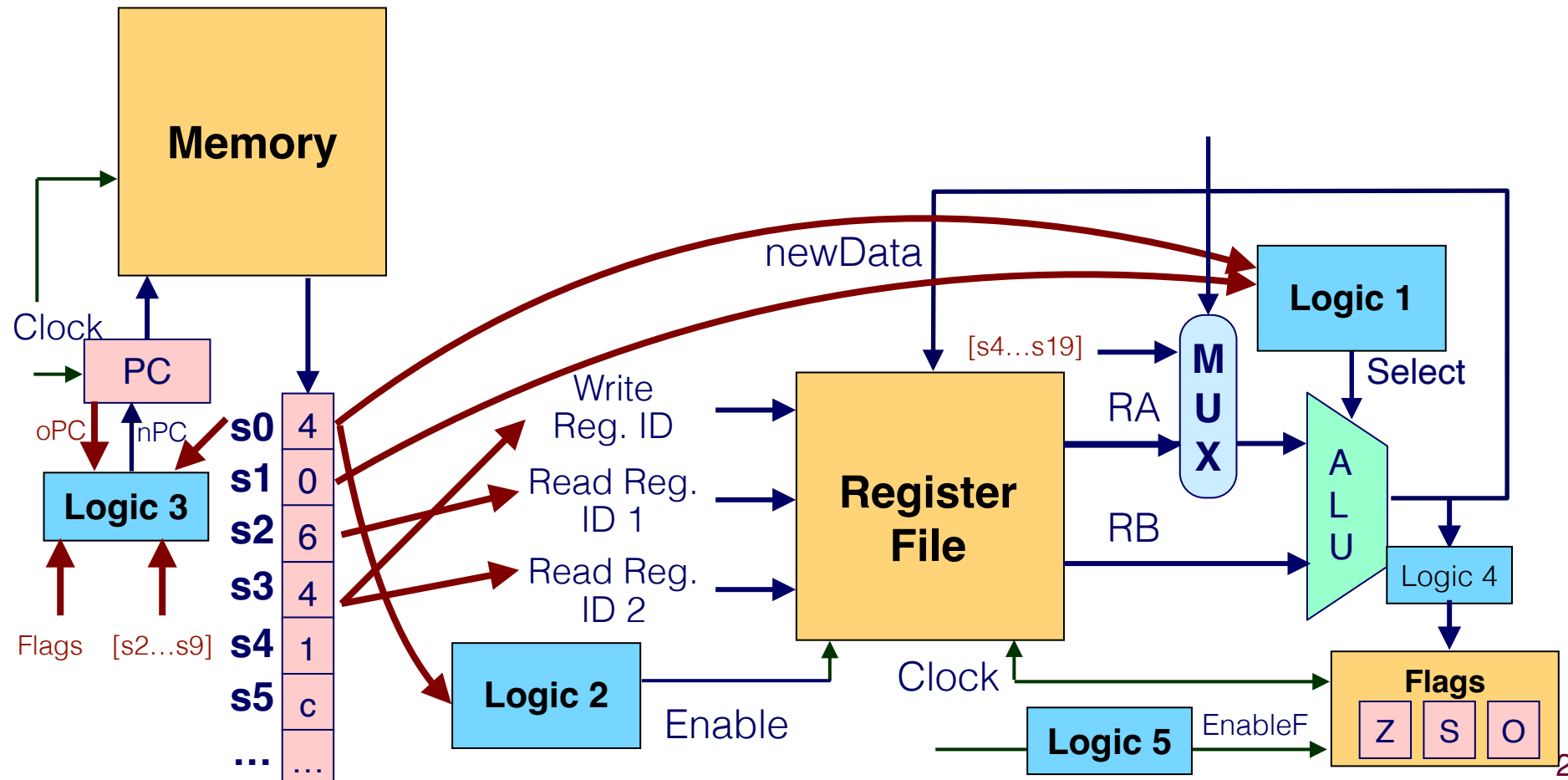
# move rA to the memory address rB + D

rmmovq rA, D(rB)



## move rA to the memory address rB + D

rmmovq rA, D(rB)

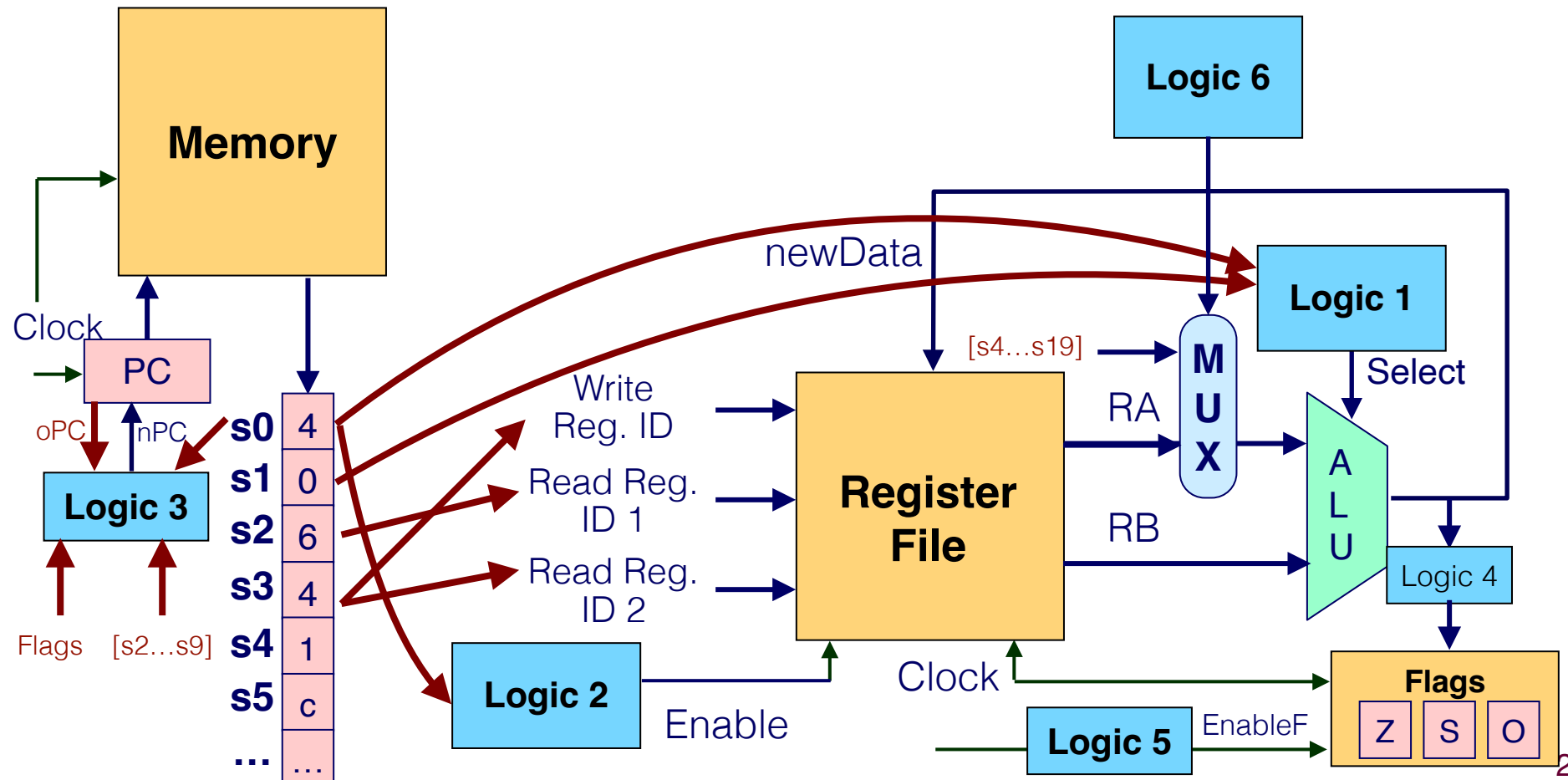


## move rA to the memory address rB + D

`rmmovq rA, D(rB)`



- Need new logic (Logic 6) to select the input to the ALU for Enable.

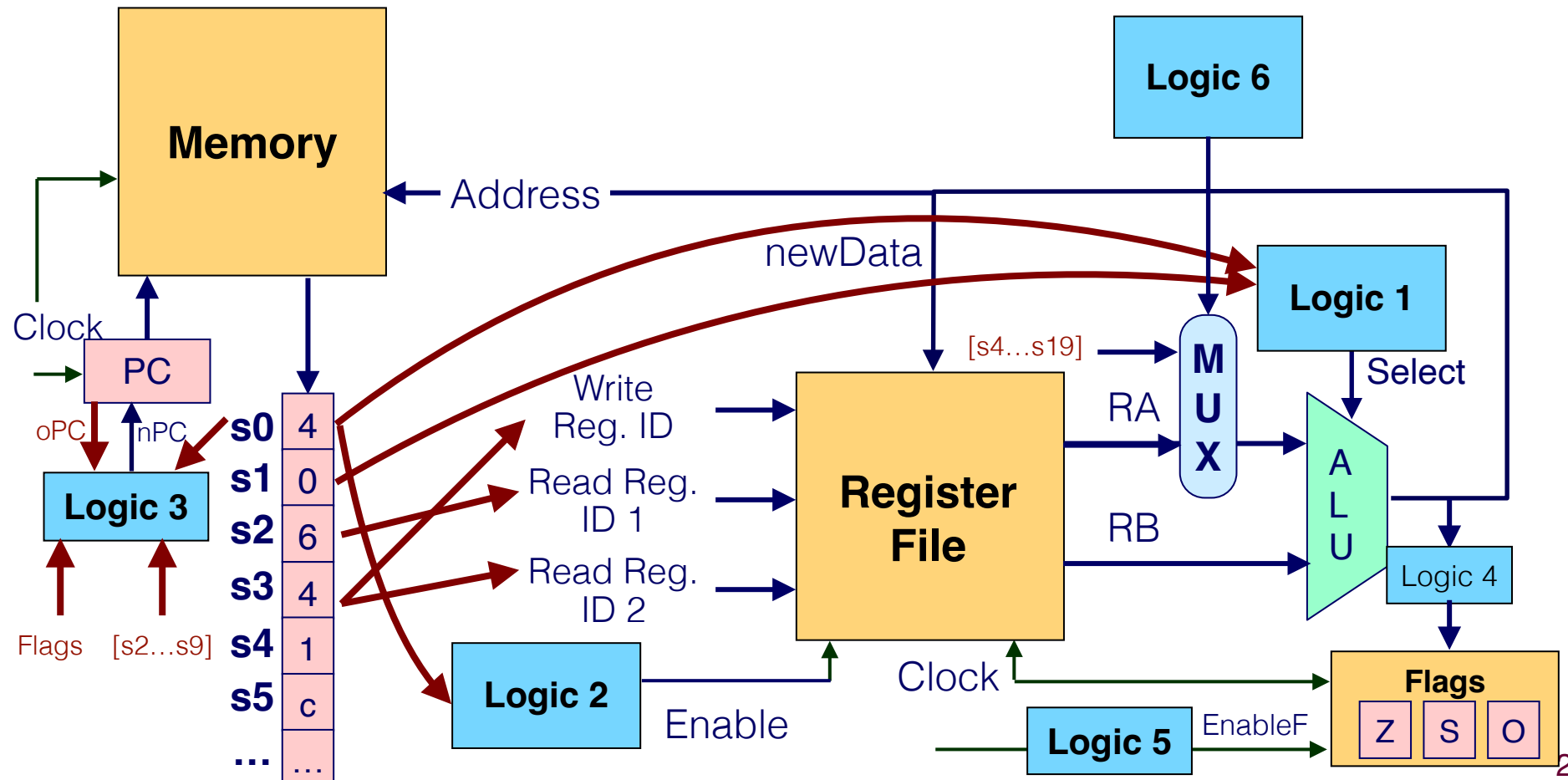


## move rA to the memory address rB + D

`rmmovq rA, D(rB)`



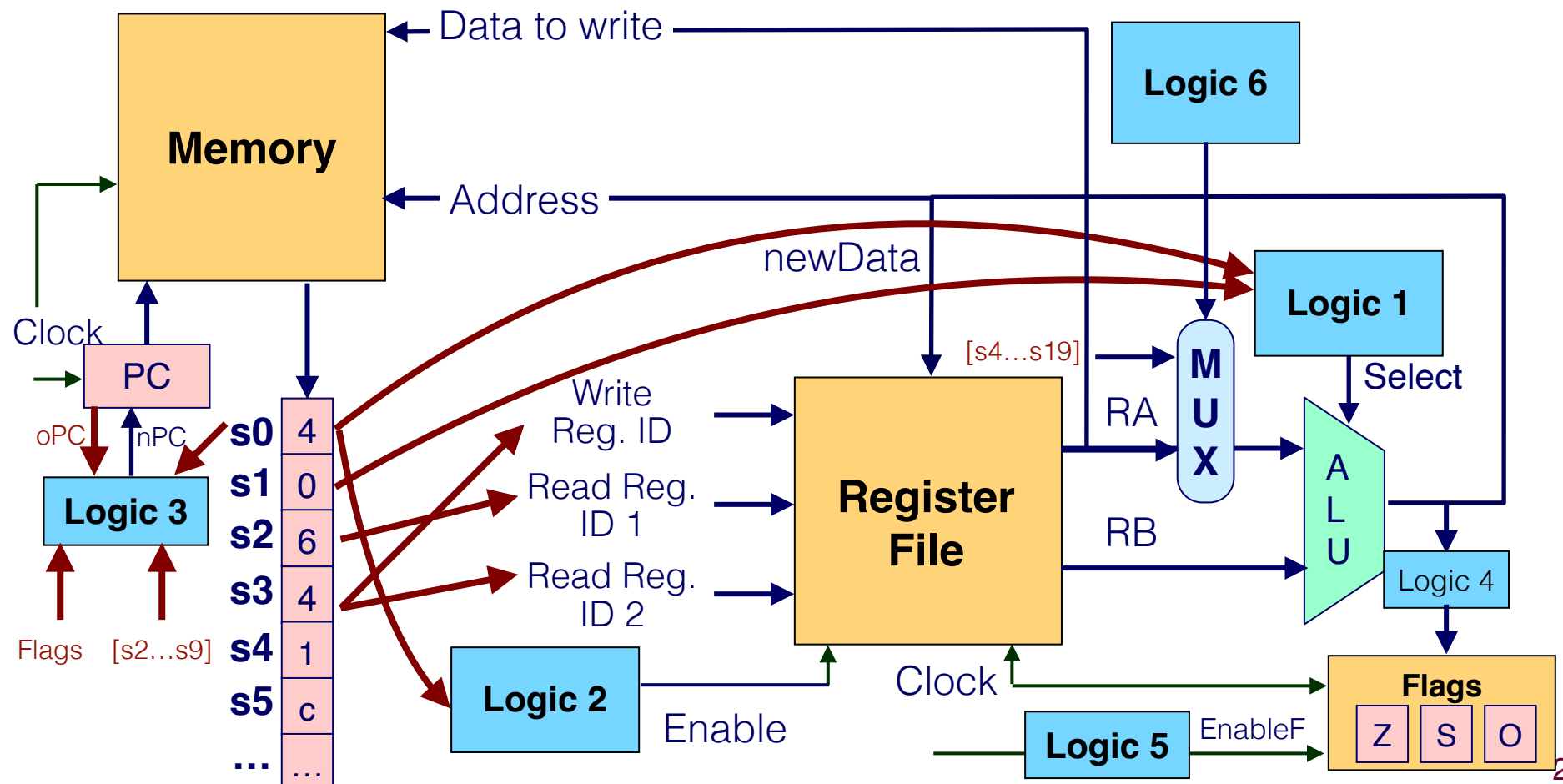
- Need new logic (Logic 6) to select the input to the ALU for Enable.



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

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

- Need new logic (Logic 6) to select the input to the ALU for Enable.

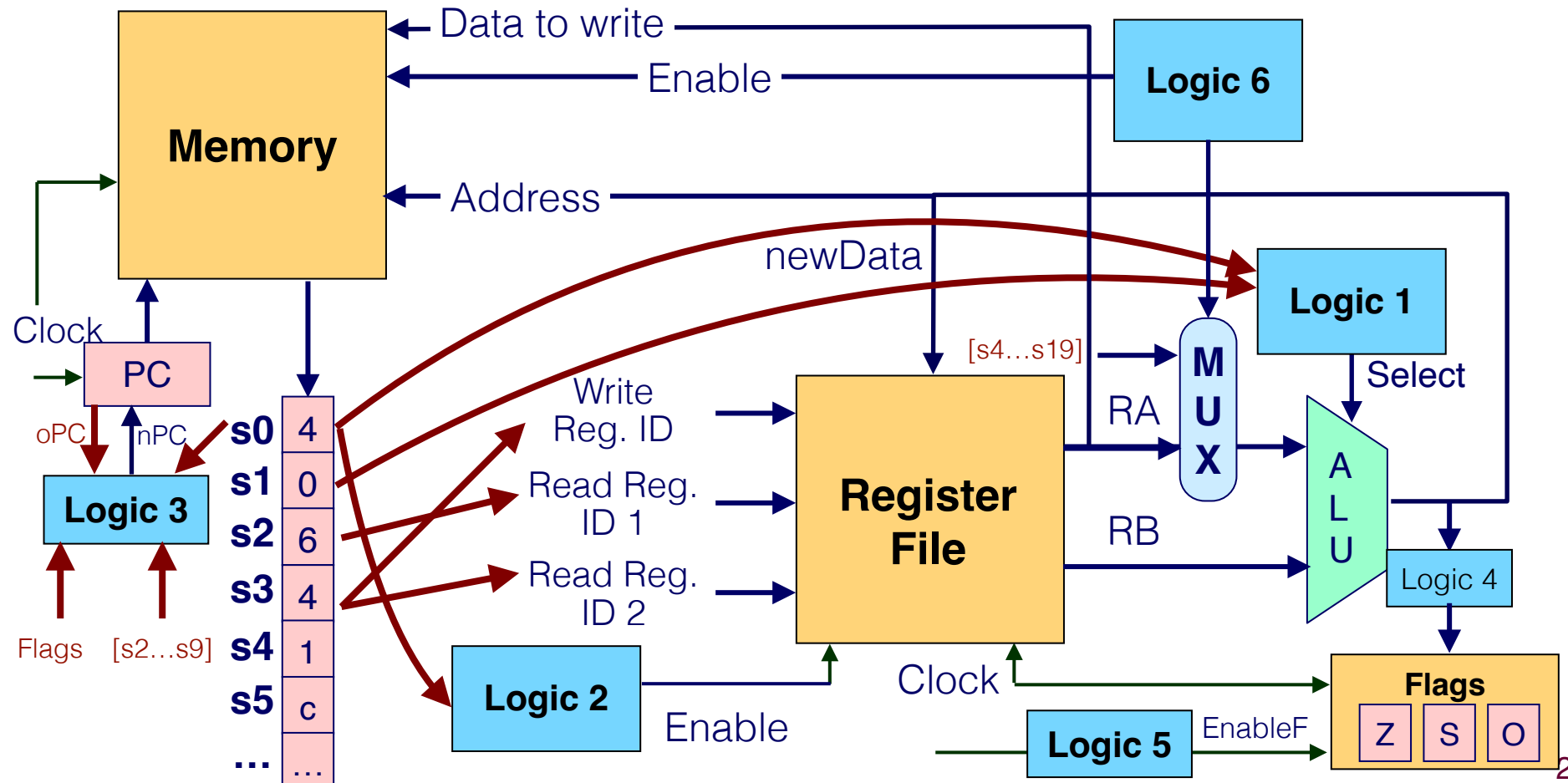


## move rA to the memory address rB + D

`rmmovq rA, D(rB)`



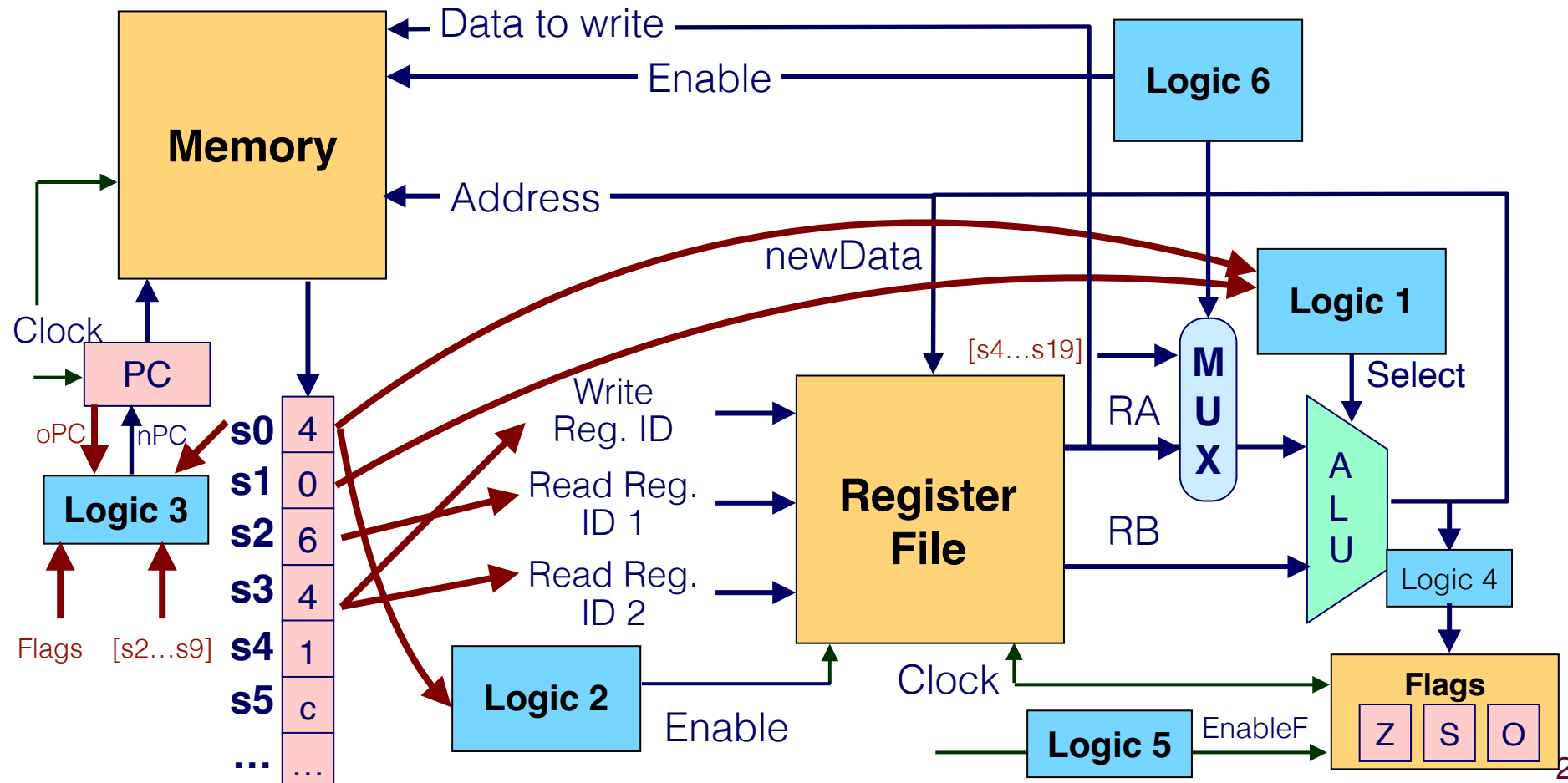
- Need new logic (Logic 6) to select the input to the ALU for Enable.



# How About Memory to Register MOV?

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

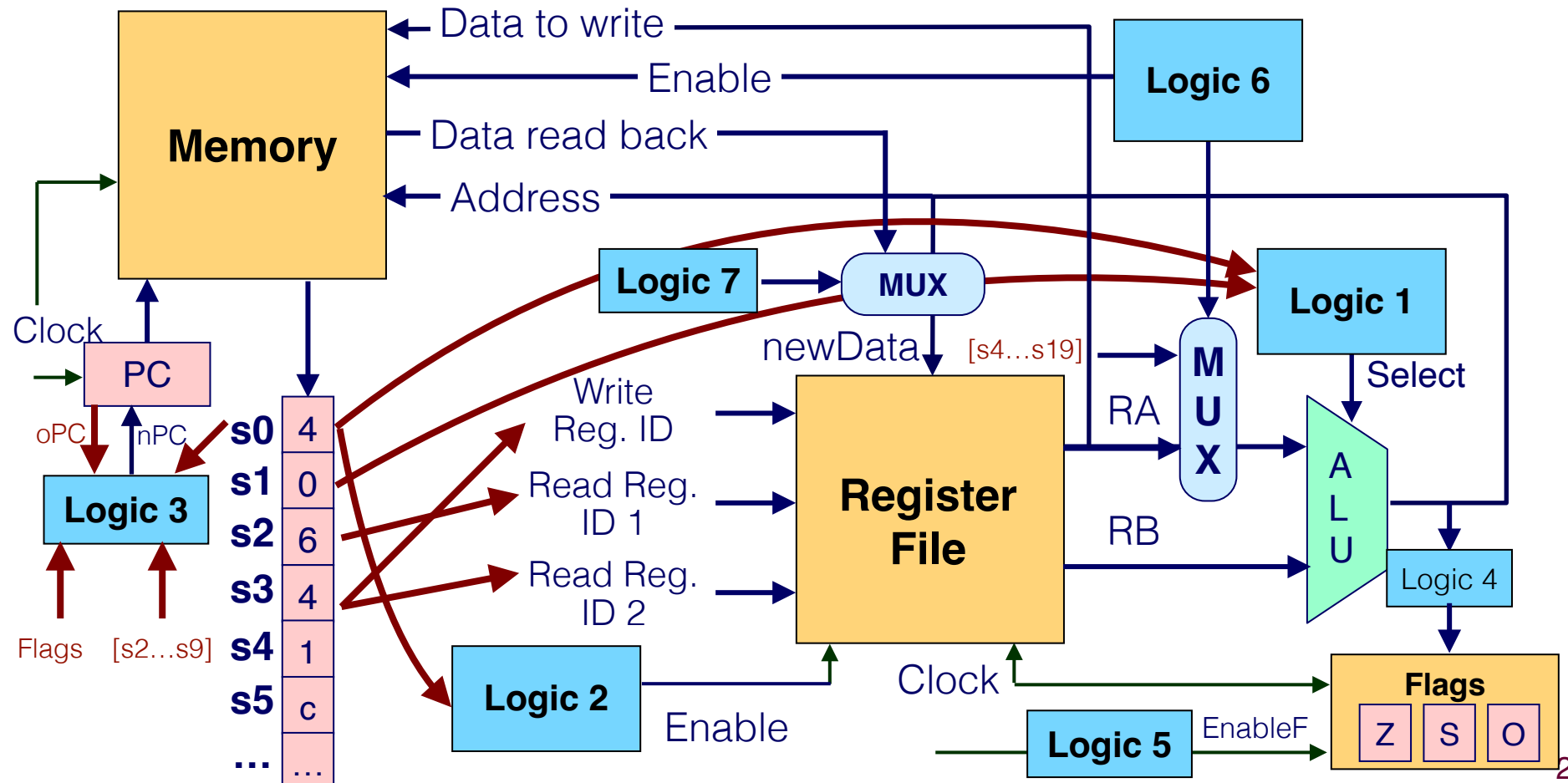
`mrmovq D(rB), rA`



# How About Memory to Register MOV?

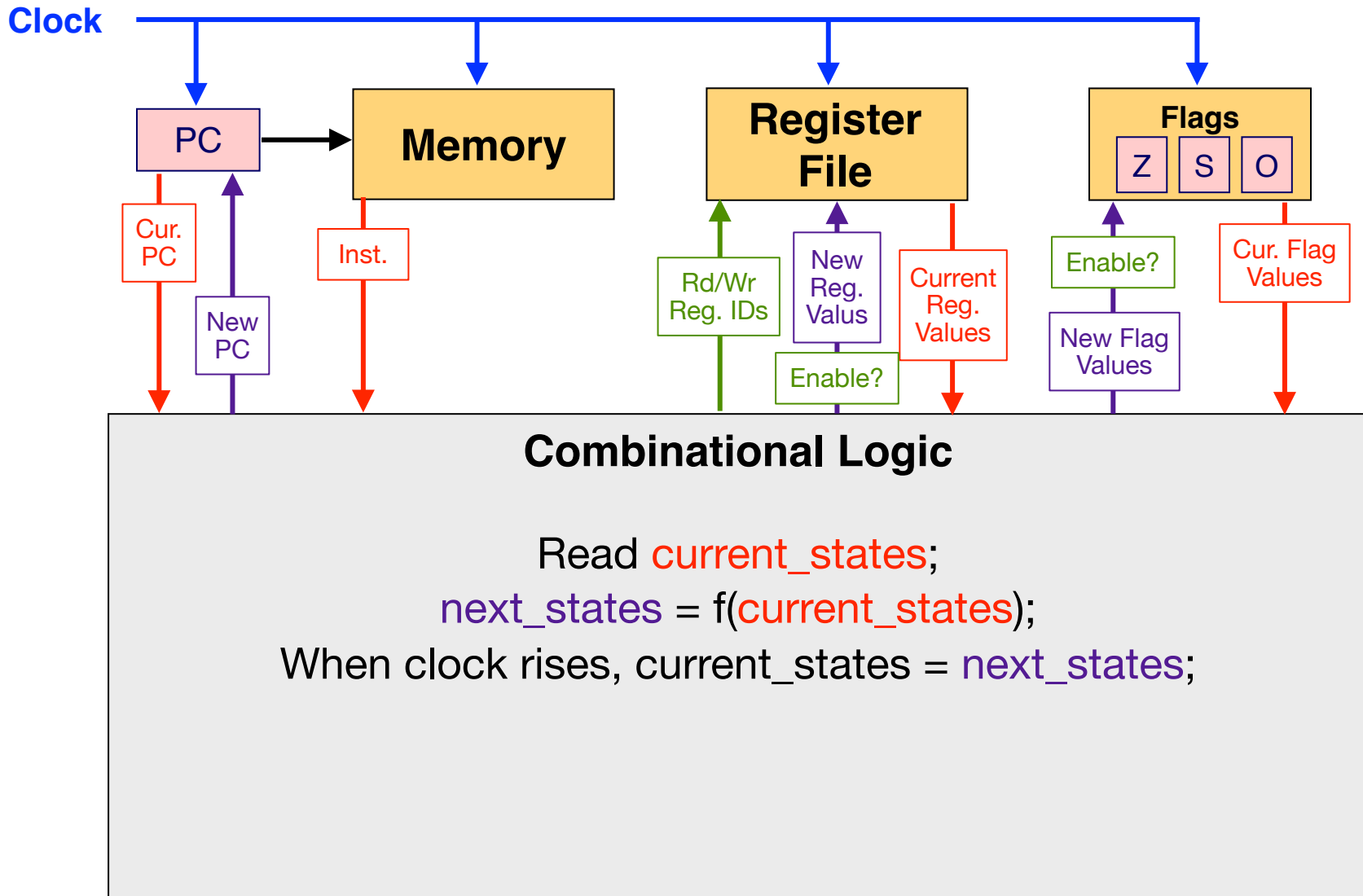
move data at memory address  $rB + D$  to  $rA$

`mrmovq D(rB), rA`

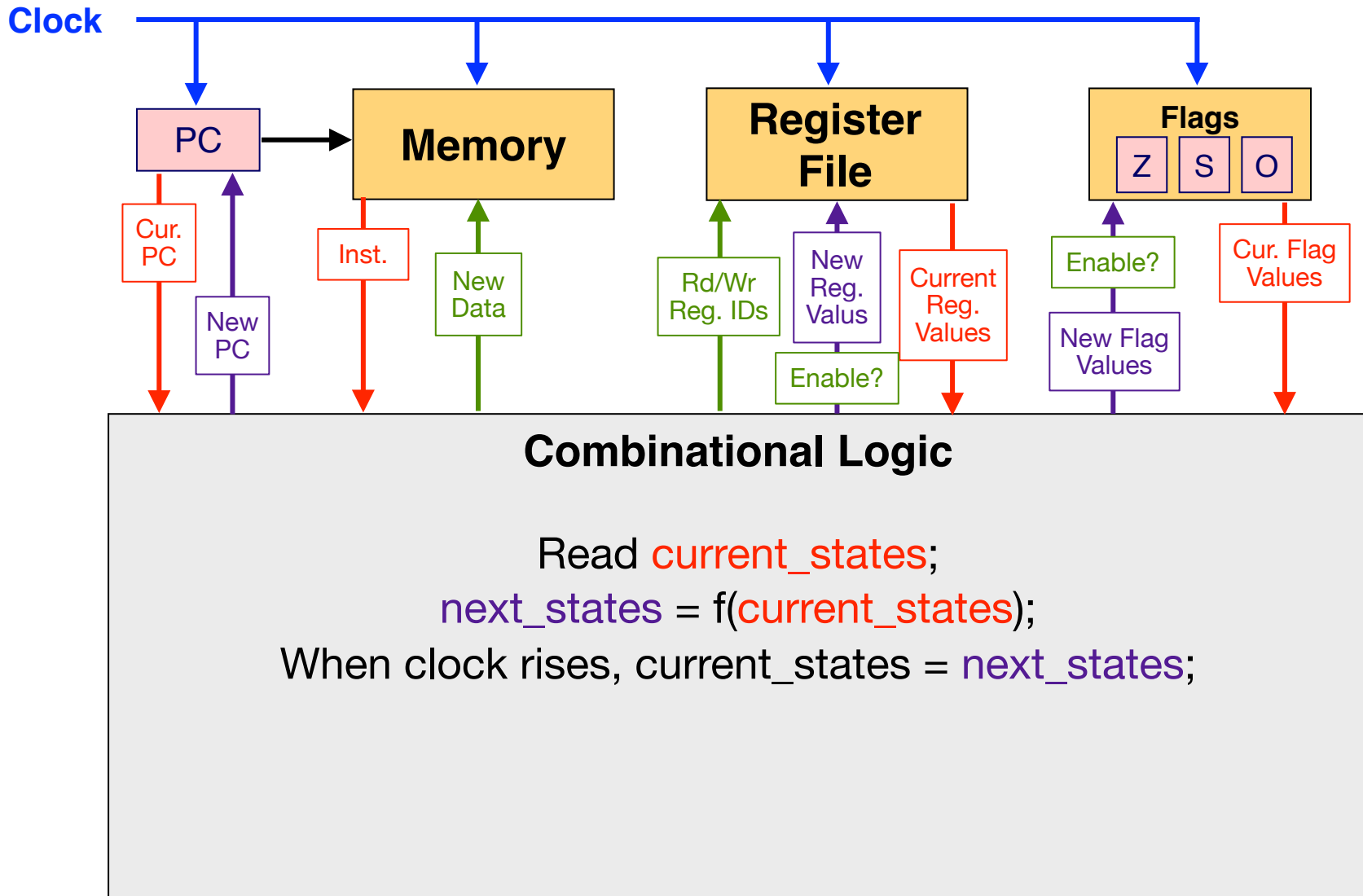




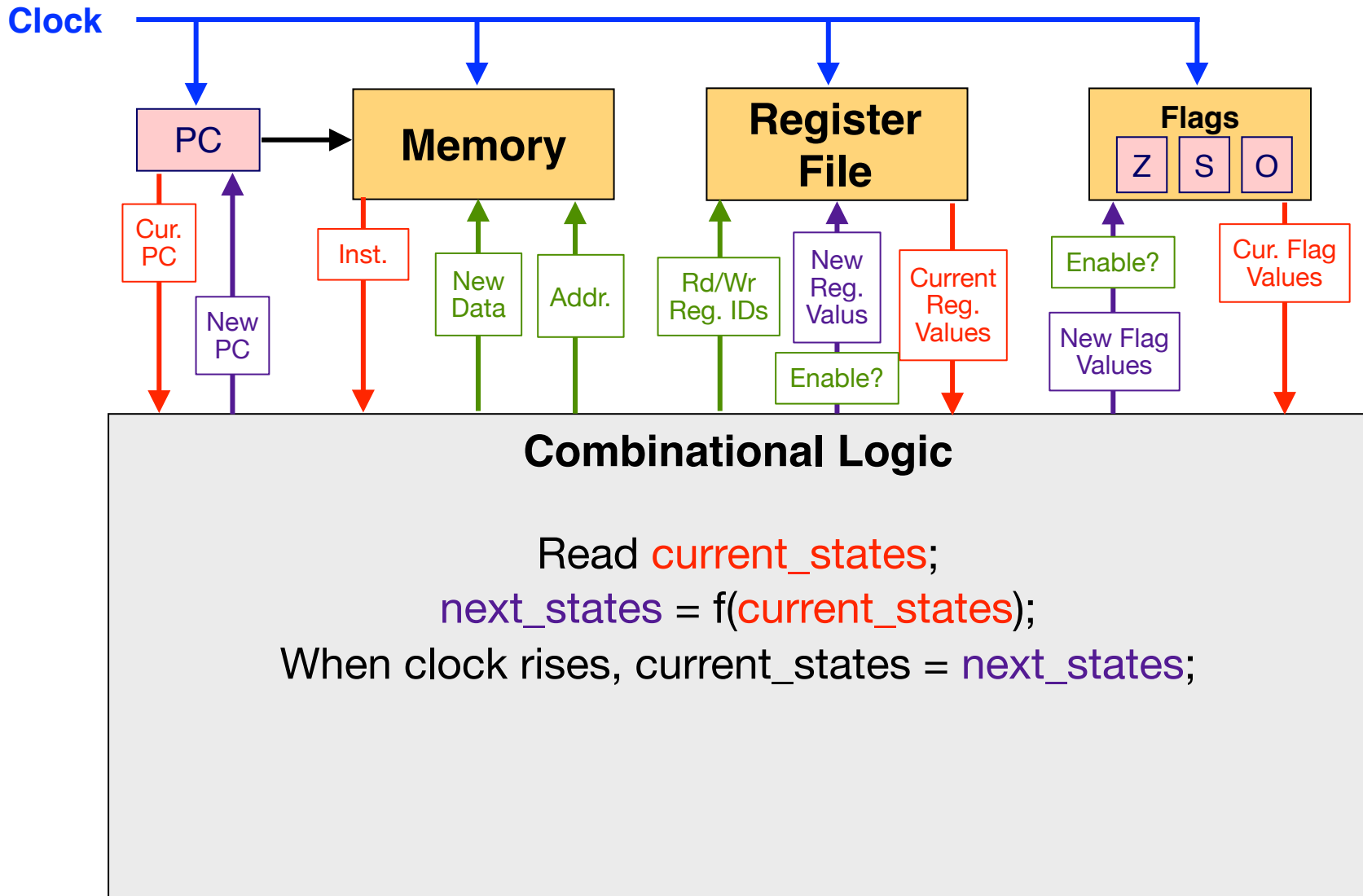
# Microarchitecture (with MOV)



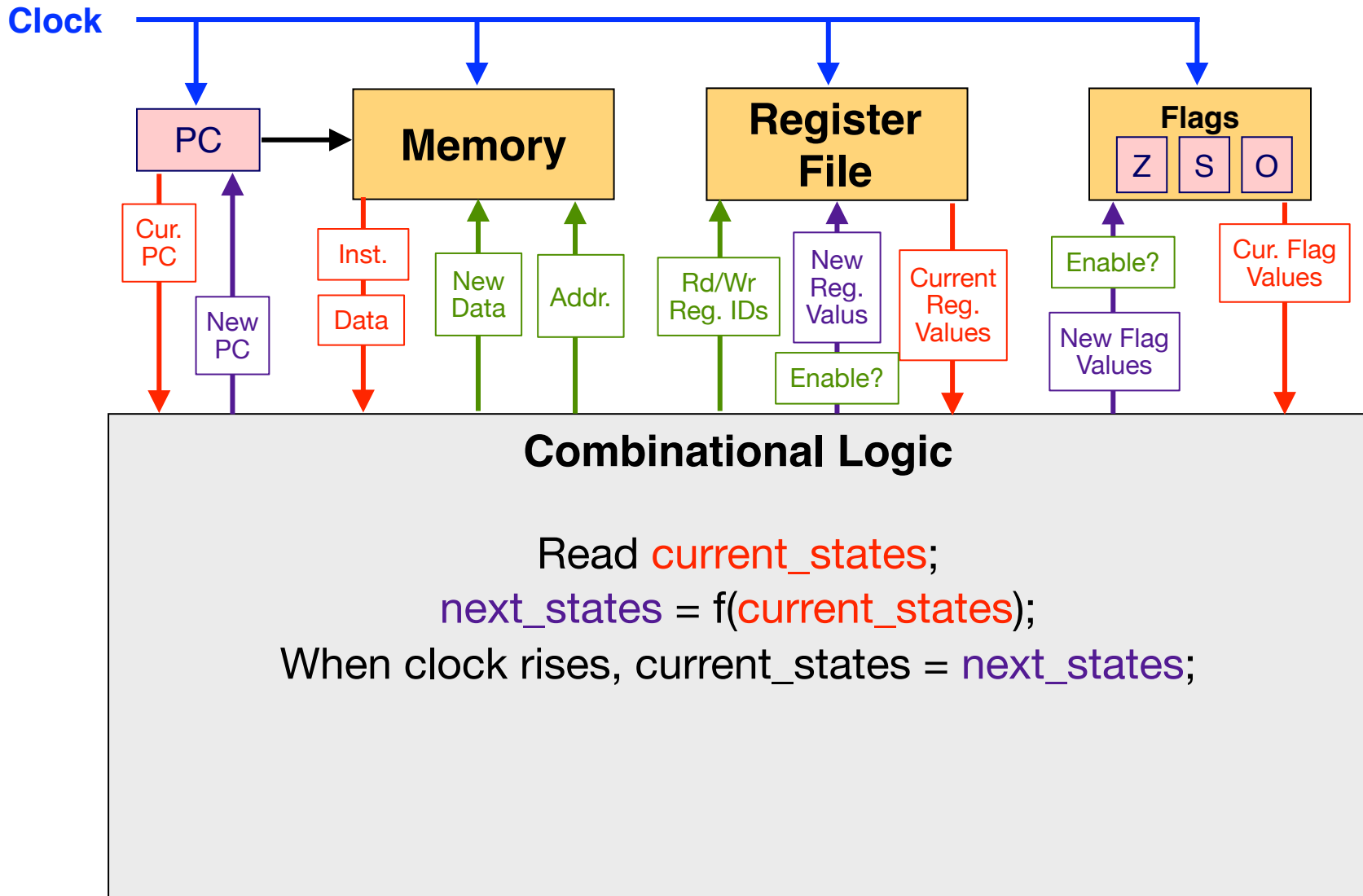
# Microarchitecture (with MOV)



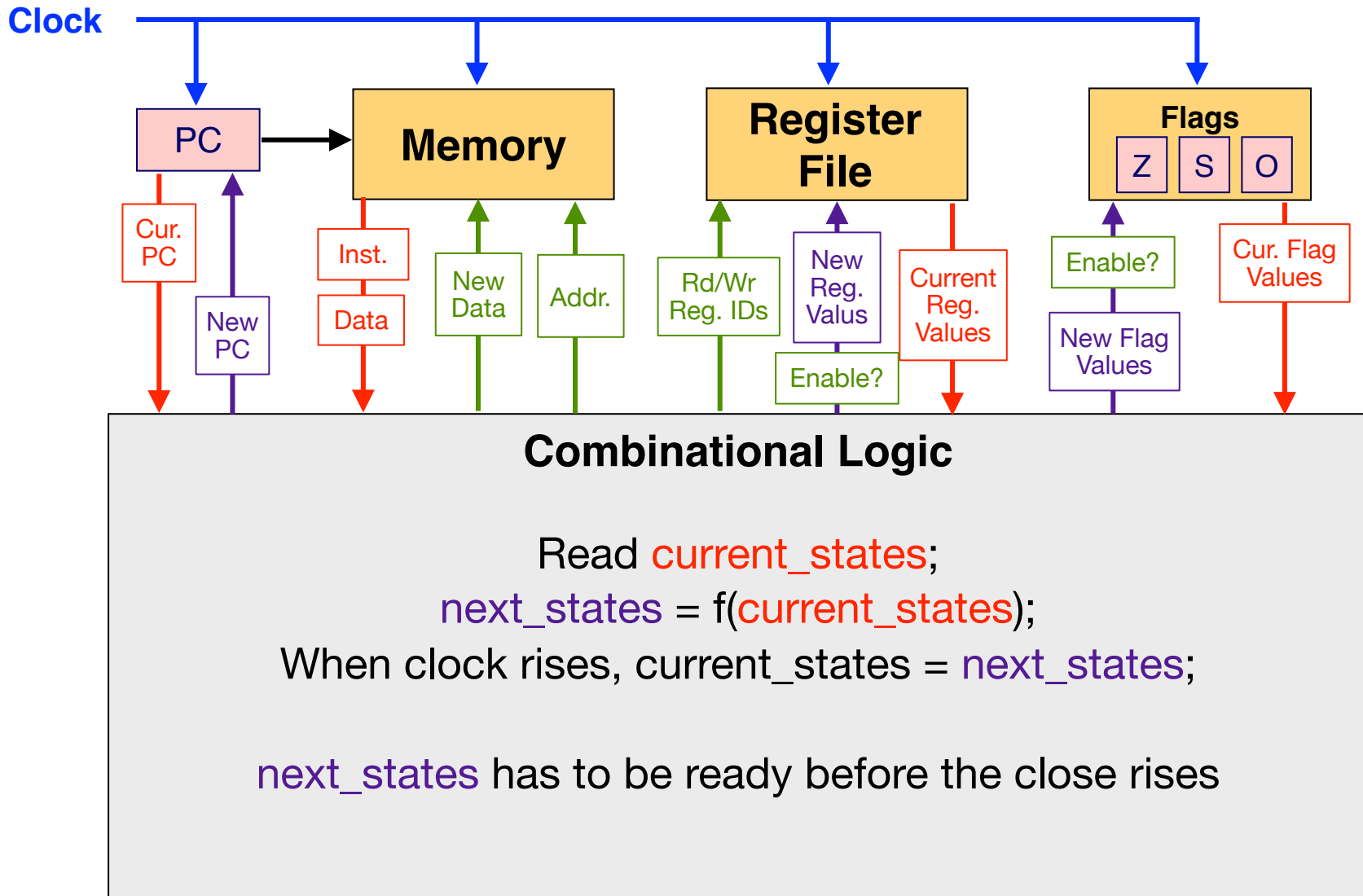
# Microarchitecture (with MOV)



# Microarchitecture (with MOV)



# Microarchitecture (with MOV)



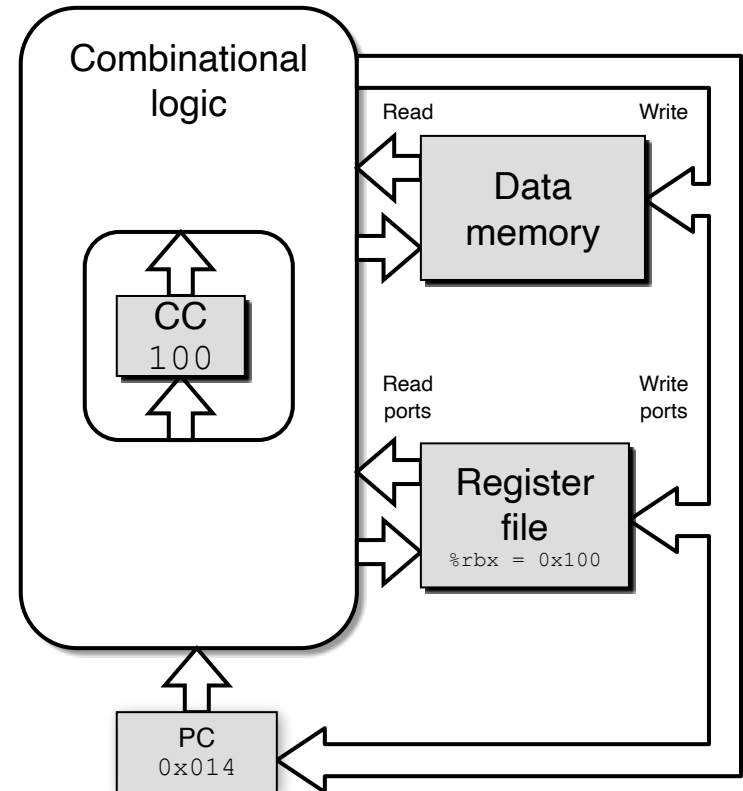
# Single-Cycle Microarchitecture: Illustration

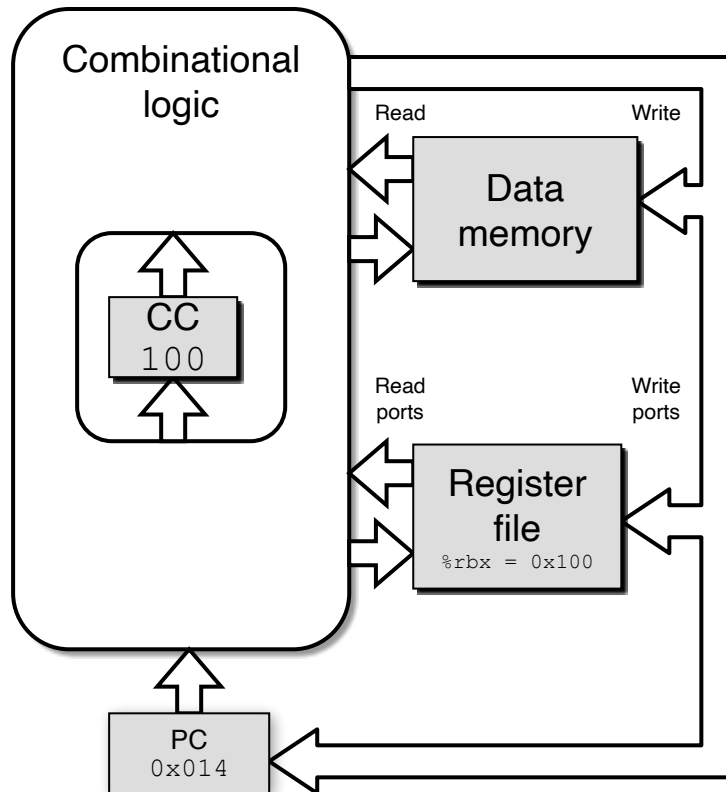
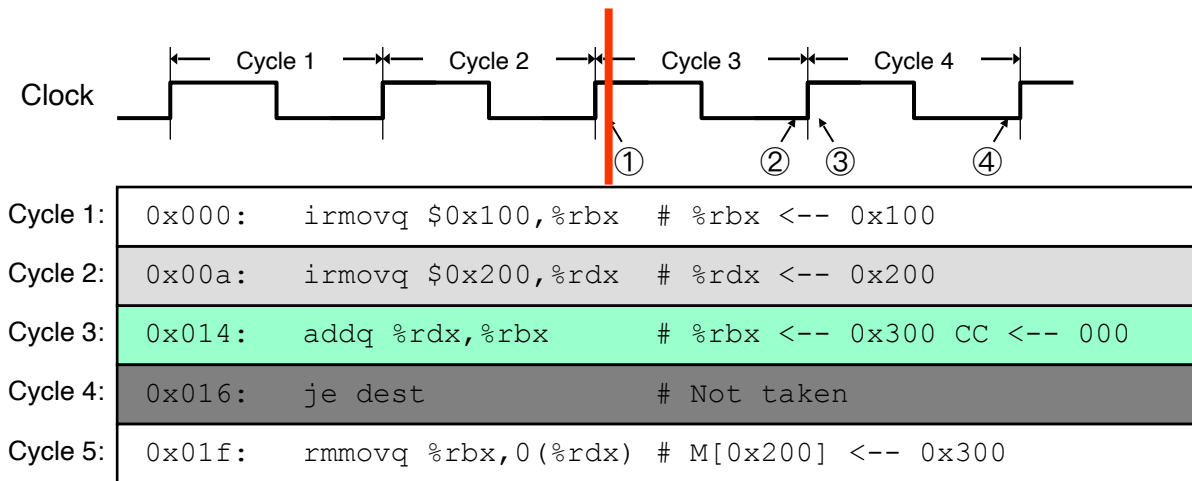
Think of it as a state machine

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

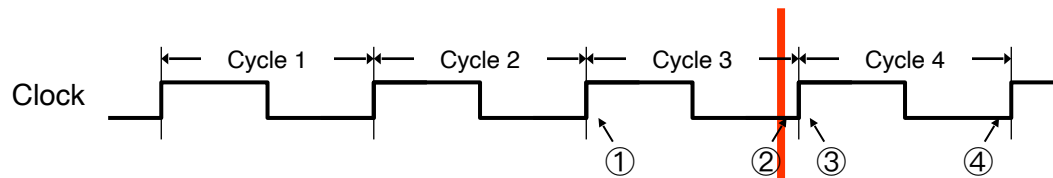
States (All updated as clock rises)

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

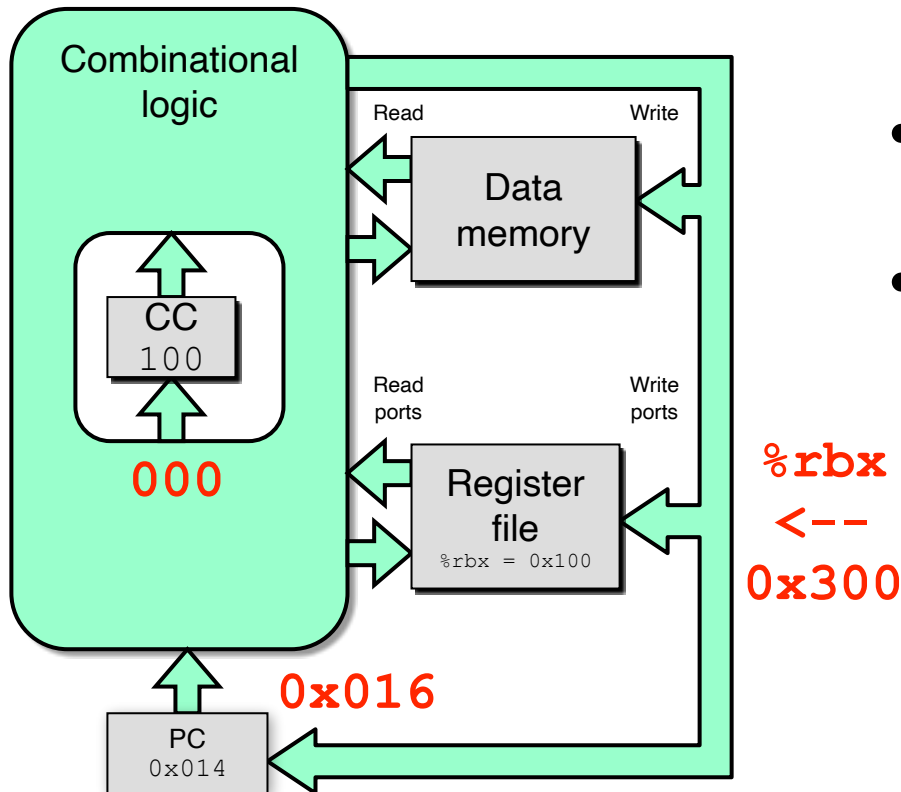




- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

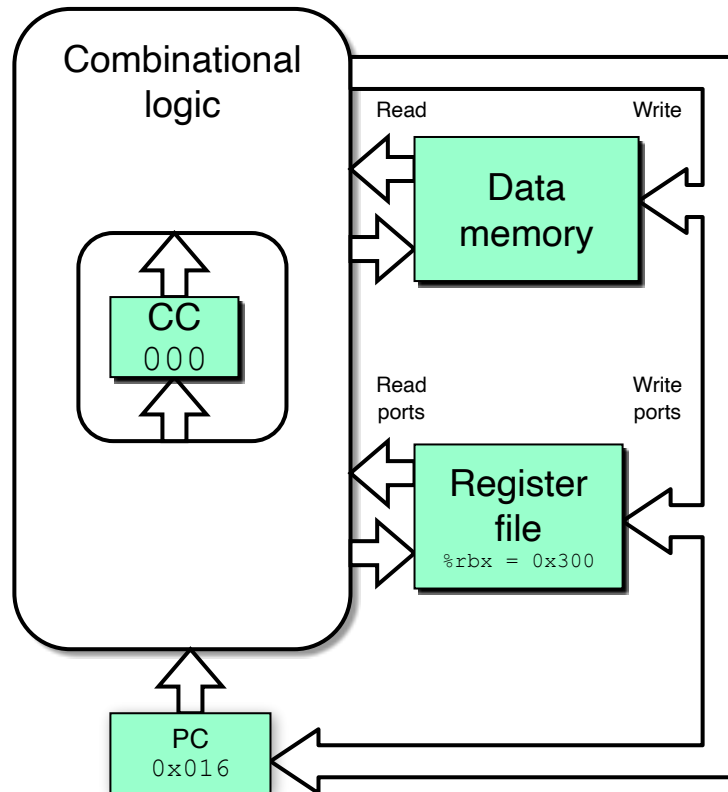
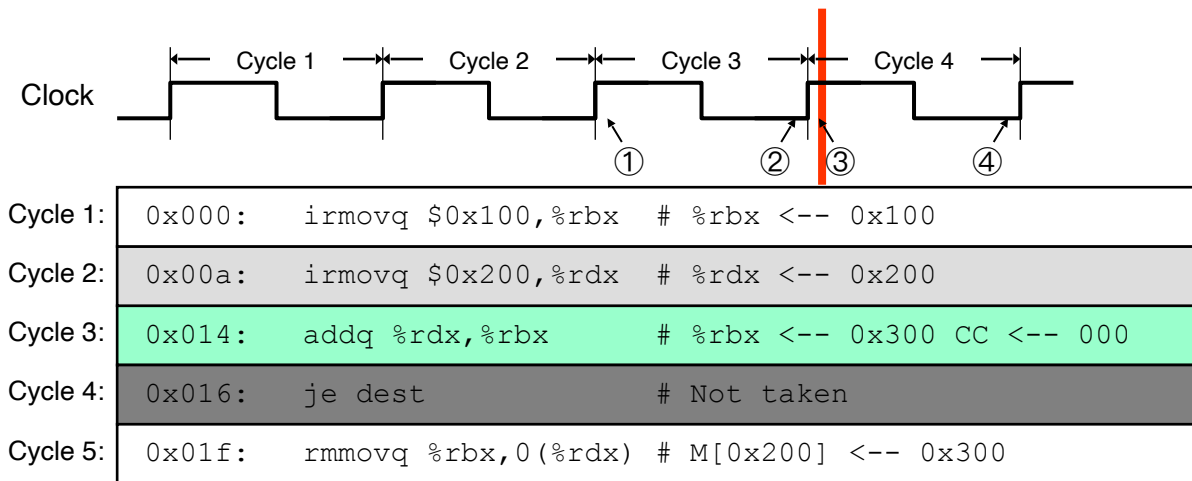


|          |        |                                  |                             |
|----------|--------|----------------------------------|-----------------------------|
| Cycle 1: | 0x000: | <code>irmovq \$0x100,%rbx</code> | # %rbx <-- 0x100            |
| Cycle 2: | 0x00a: | <code>irmovq \$0x200,%rdx</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        |

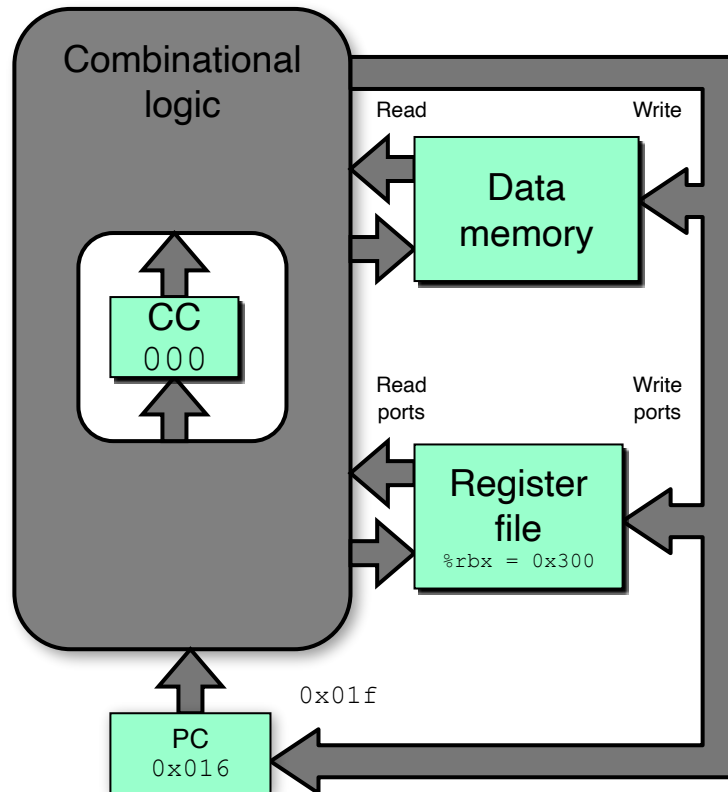
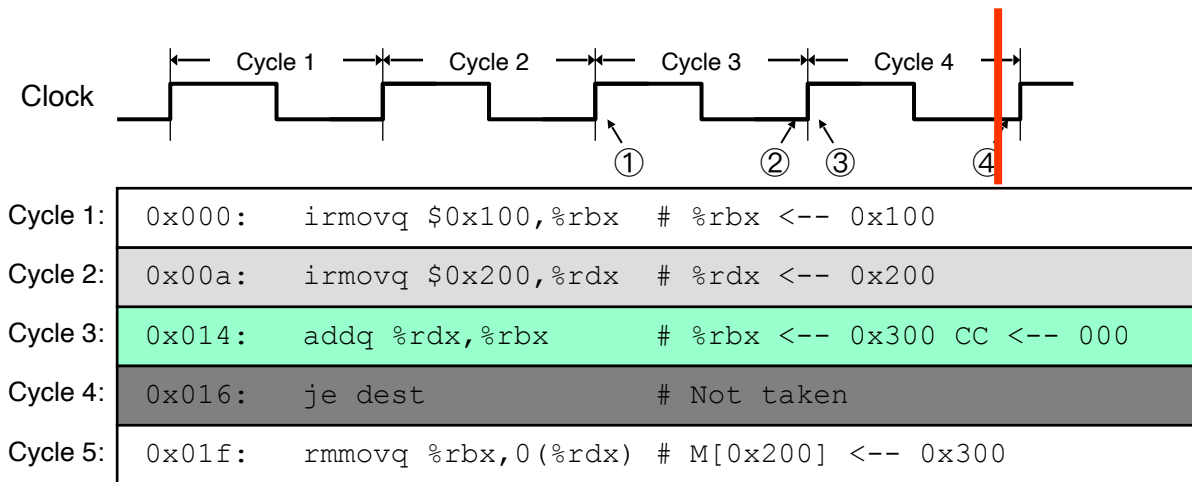


- state set according to second `irmovq` instruction
- combinational logic generates results for `addq` instruction





- 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

# Processor Microarchitecture

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

# 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

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

**CPI**

# Performance Model

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

# Improving Performance

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

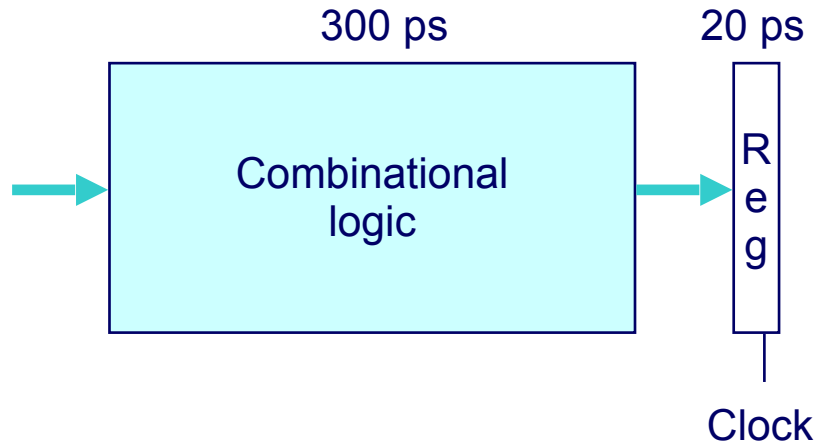
# Limitations of a Single-Cycle CPU

- 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