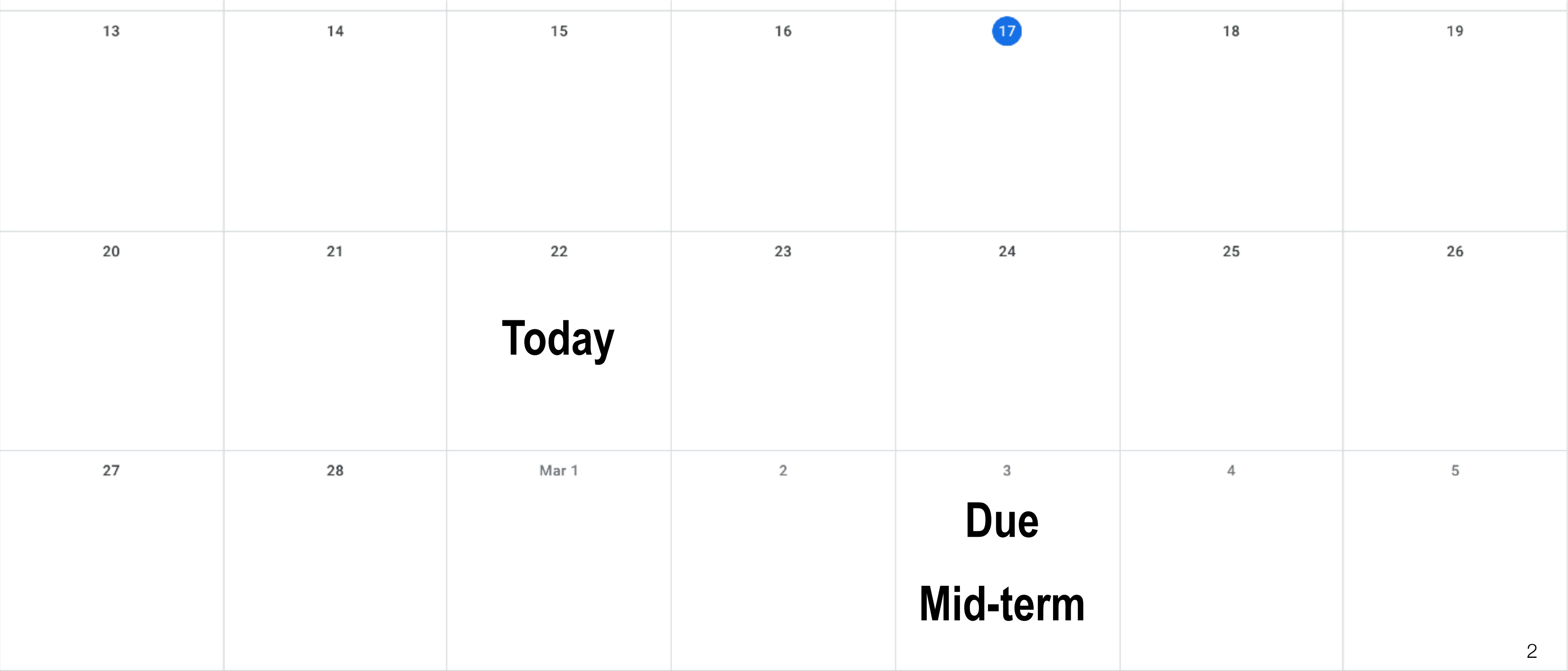
CSC 252: Computer Organization
Spring 2022: Lecture 12

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

- Programming assignment 3 is out
  - Details: https://www.cs.rochester.edu/courses/252/spring2022/labs/assignment3.html
  - Due on **March 3**, 11:59 PM
  - You (may still) have 3 slip days



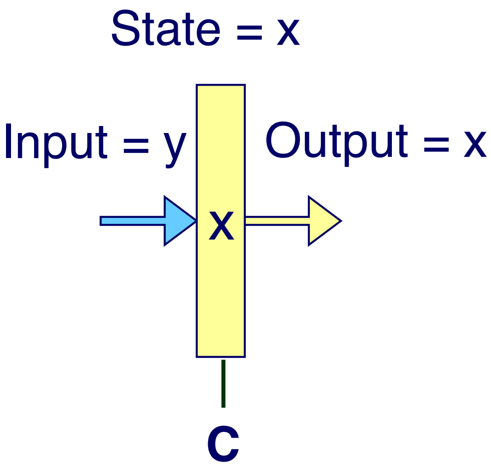
13	14	15	16	17	18	19
20	21	22 **Today**	23	24	25	26
27	28	Mar 1	2	3 **Due** **Mid-term**	4	5

Announcements

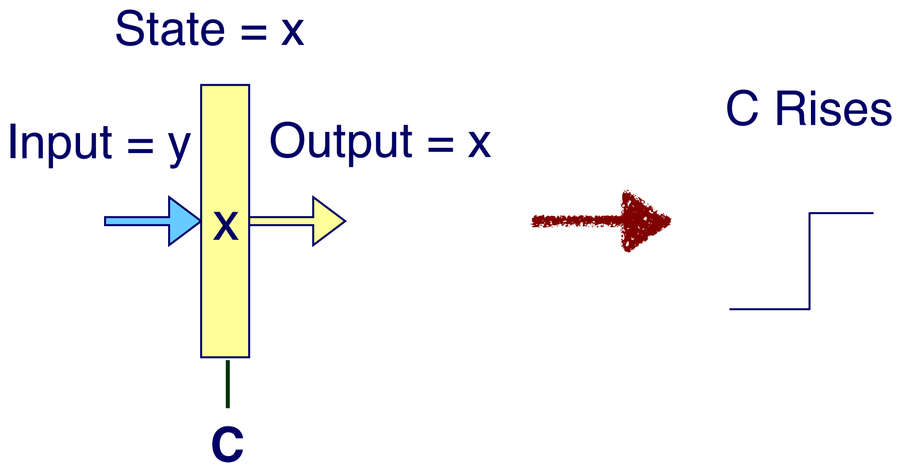
- Grades for Lab 1 are posted.
- Will grade Lab 2 soon.

- 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.

Register Operation



Register Operation



State = x

Input = y

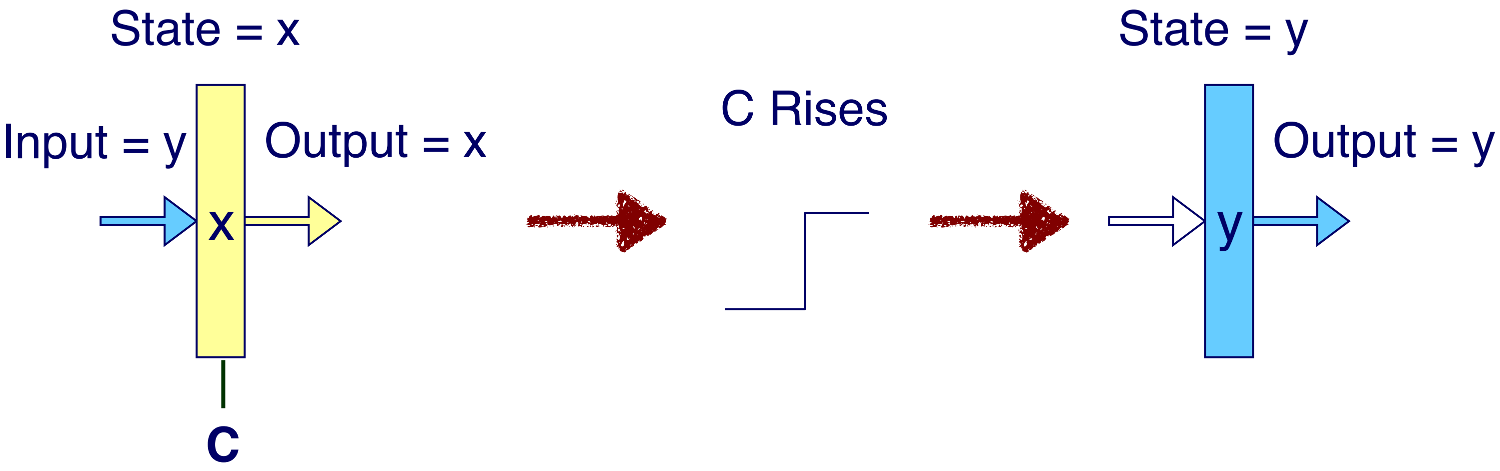
x

Output = x

C

C Rises

Register Operation



State = x

Input = y

x

Output = x

C

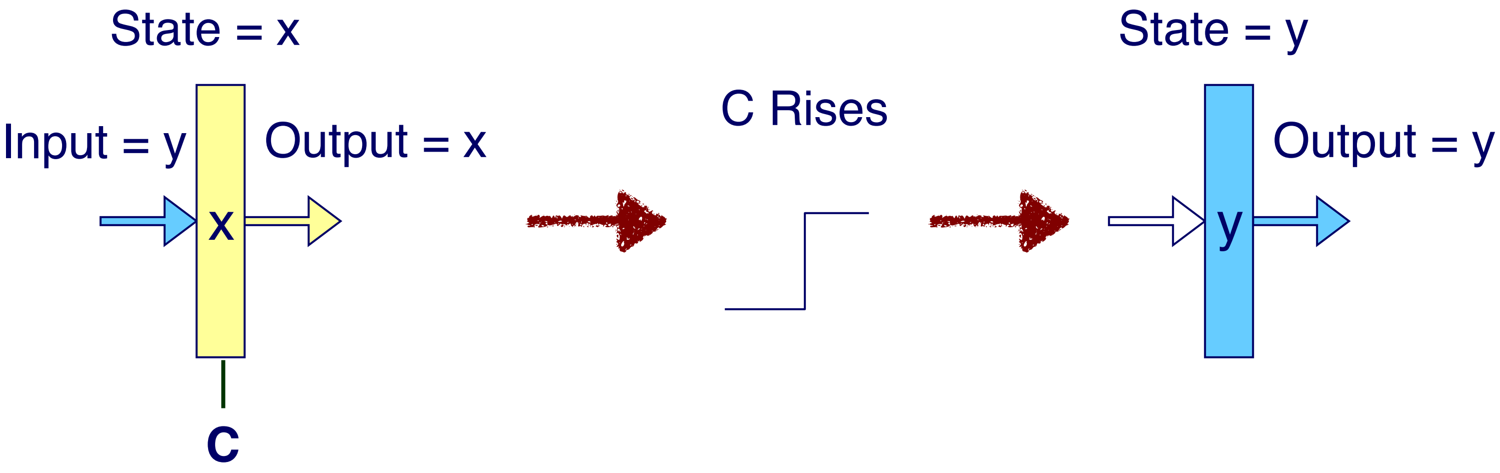
C Rises

State = y

y

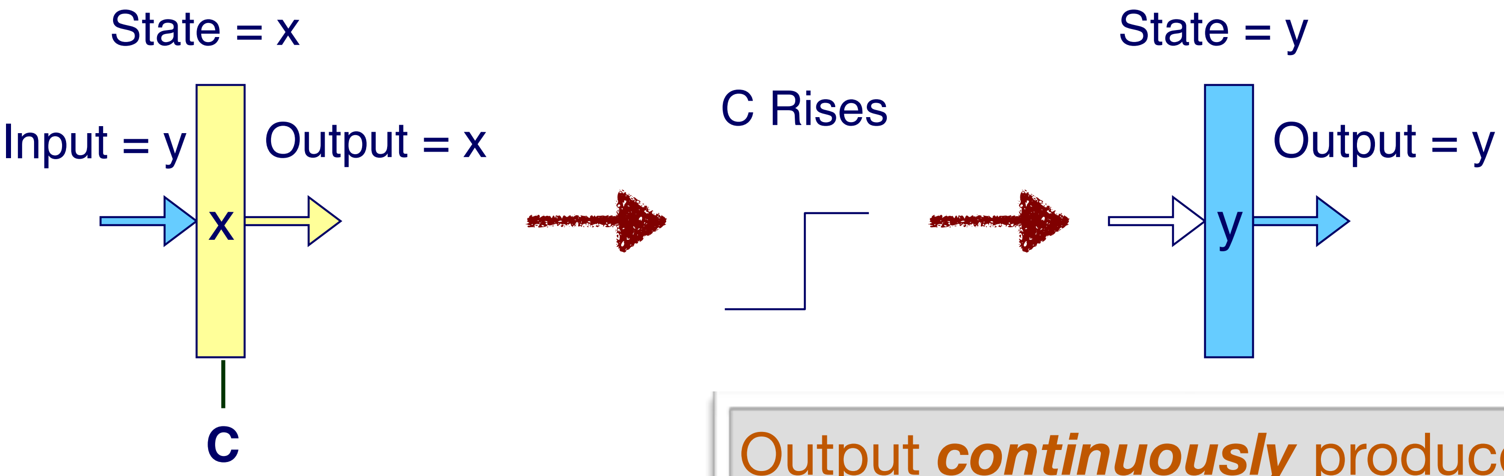
Output = y

Register Operation



- Stores data bits
- For most of time acts as barrier between input and output
- As C rises, loads input
- So you'd better compute the input before the C signal rises if you want to store the input data to the register

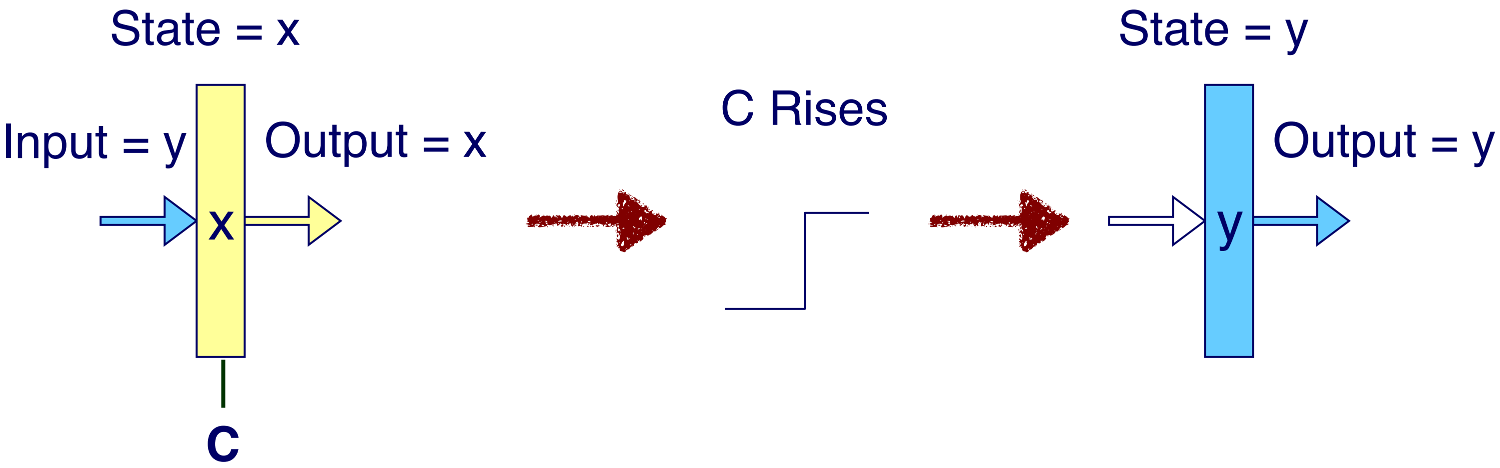
Register Operation



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

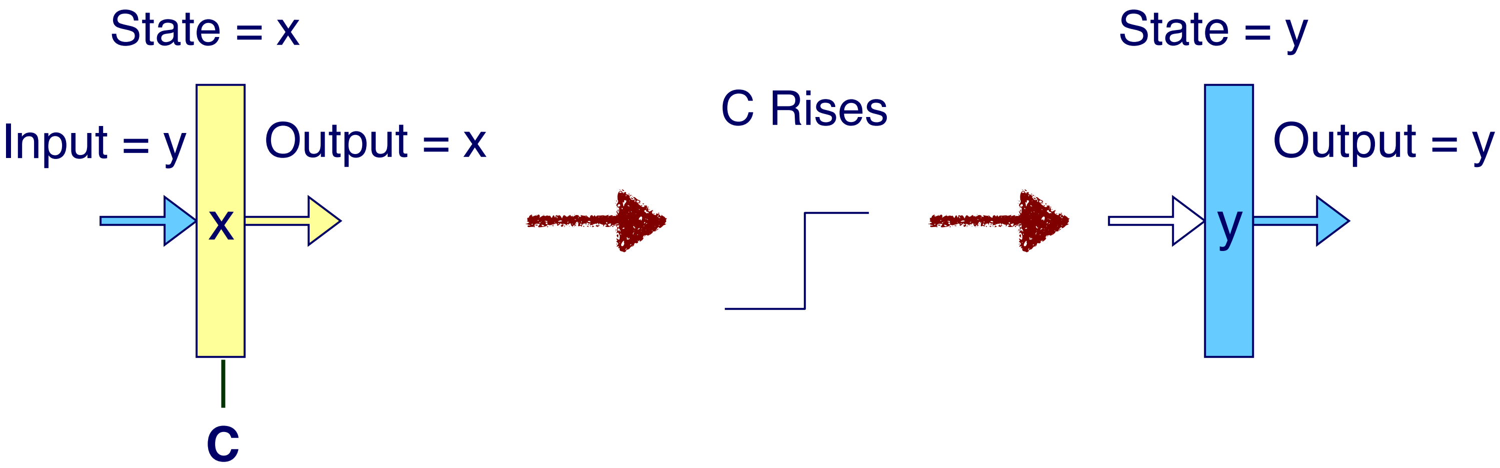
- Stores data bits
- For most of time acts as barrier between input and output
- As C rises, loads input
- So you'd better compute the input before the C signal rises if you want to store the input data to the register

Clock Signal

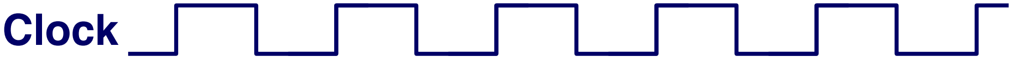


- A special C: periodically oscillating between 0 and 1
- That’s called the **clock** signal. Generated by a crystal oscillator inside your computer.

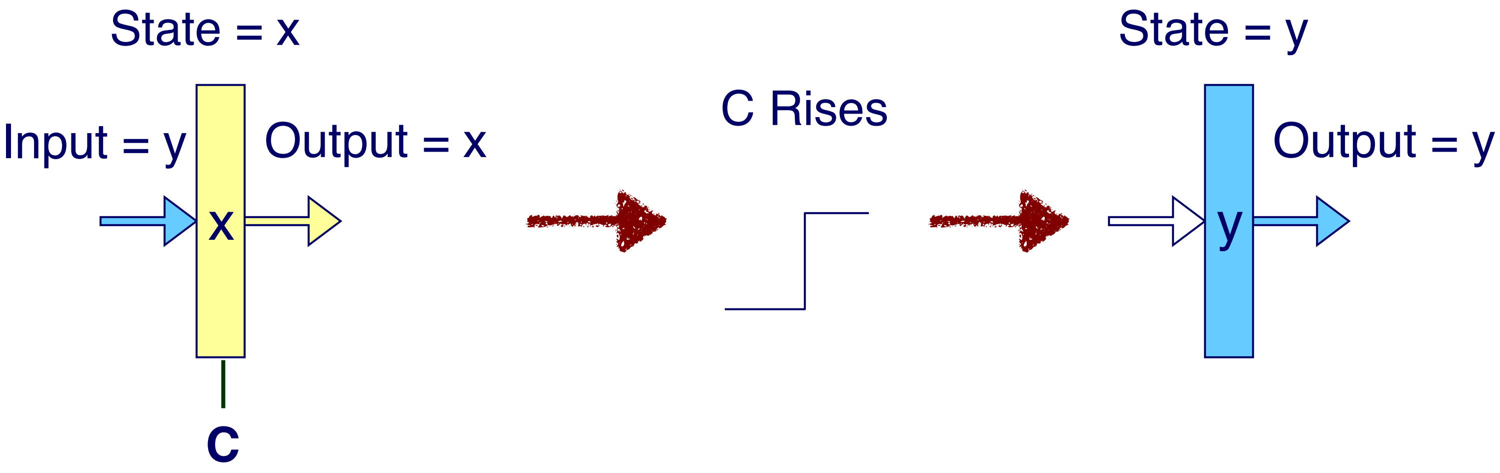
Clock Signal



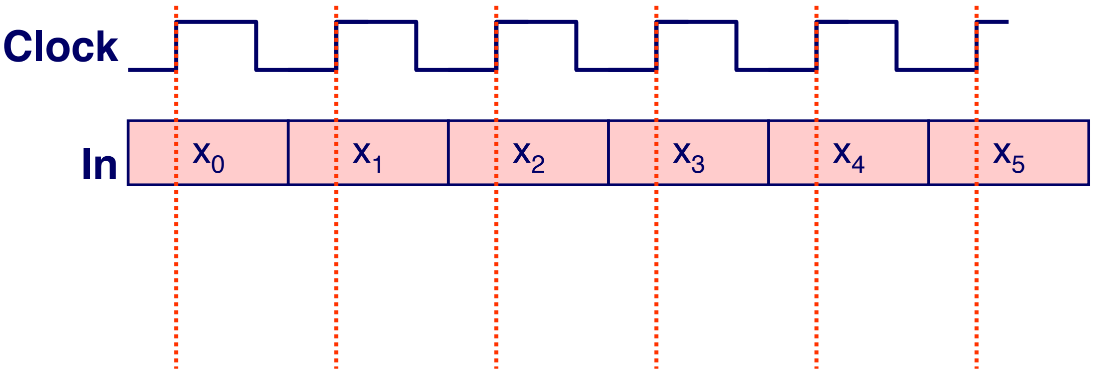
- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.



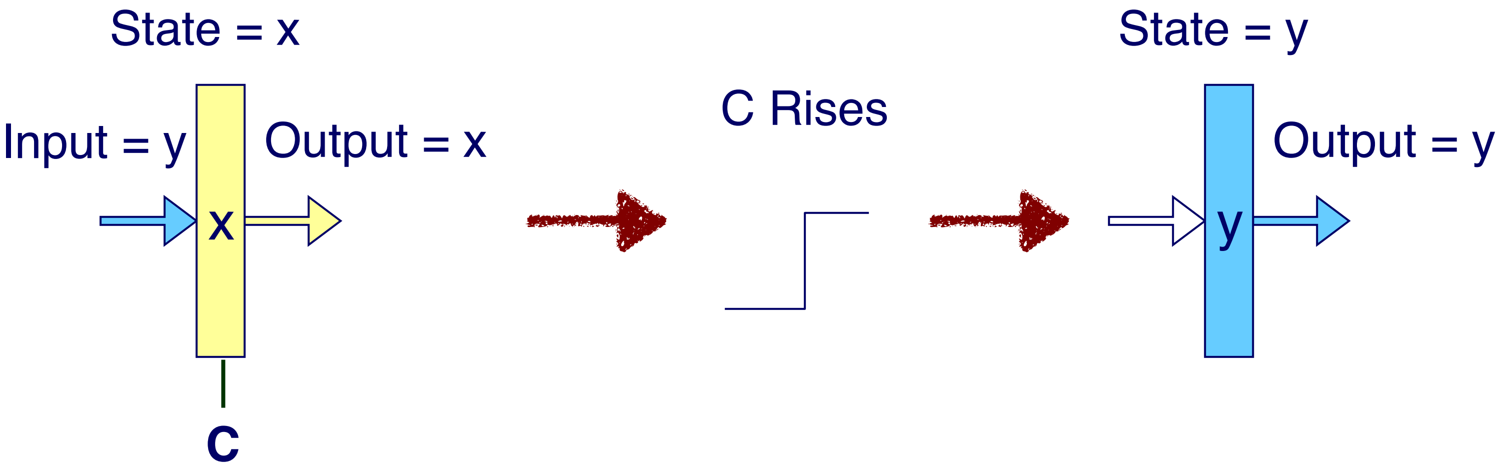
Clock Signal



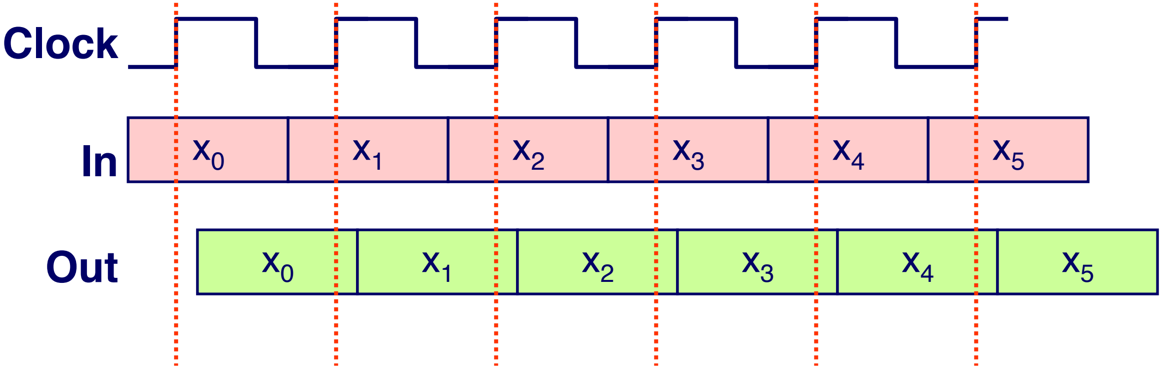
- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.



Clock Signal

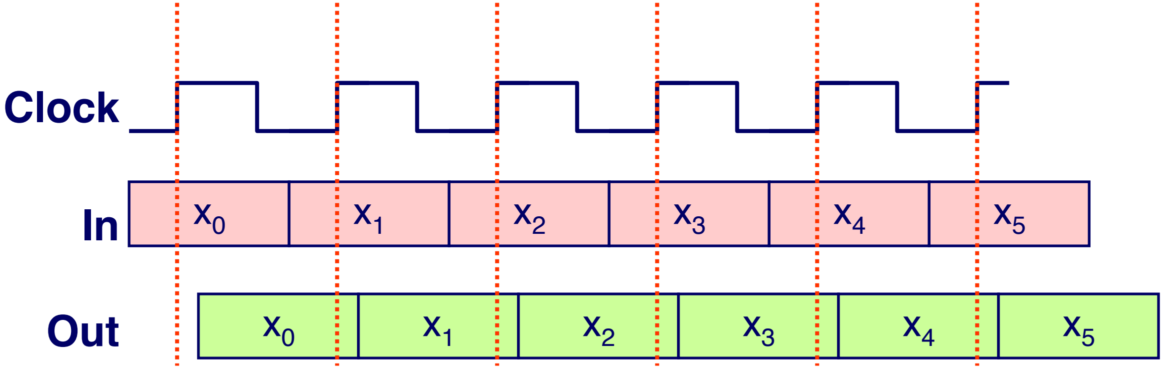


- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.



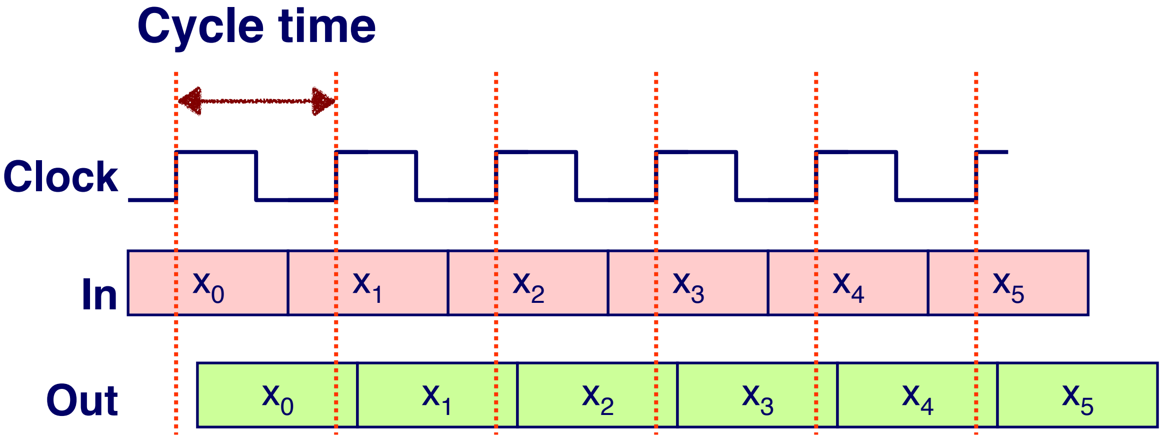
Clock Signal

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



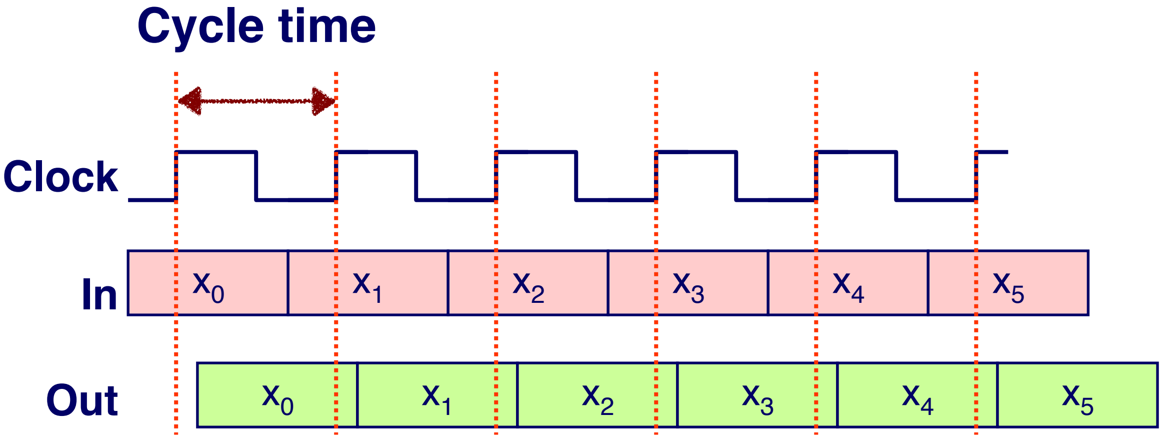
Clock Signal

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



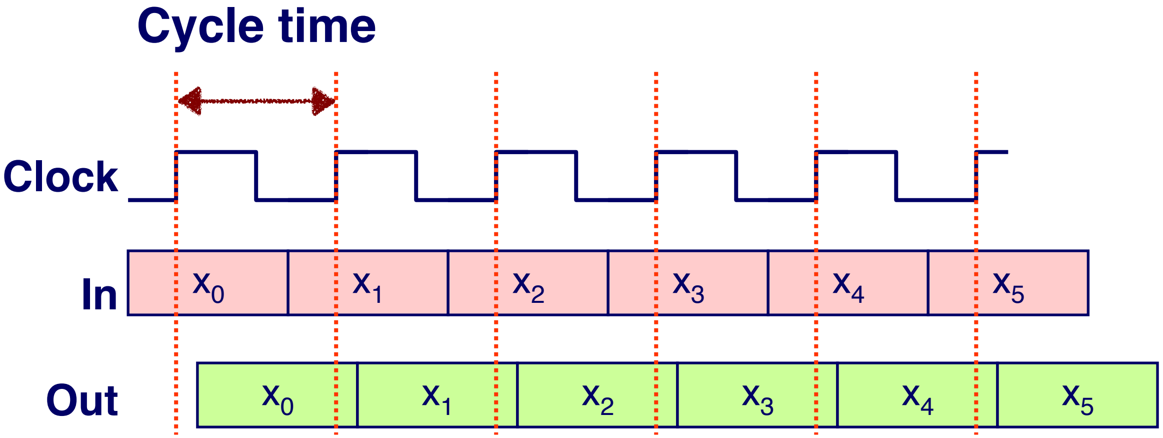
Clock Signal

- 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.



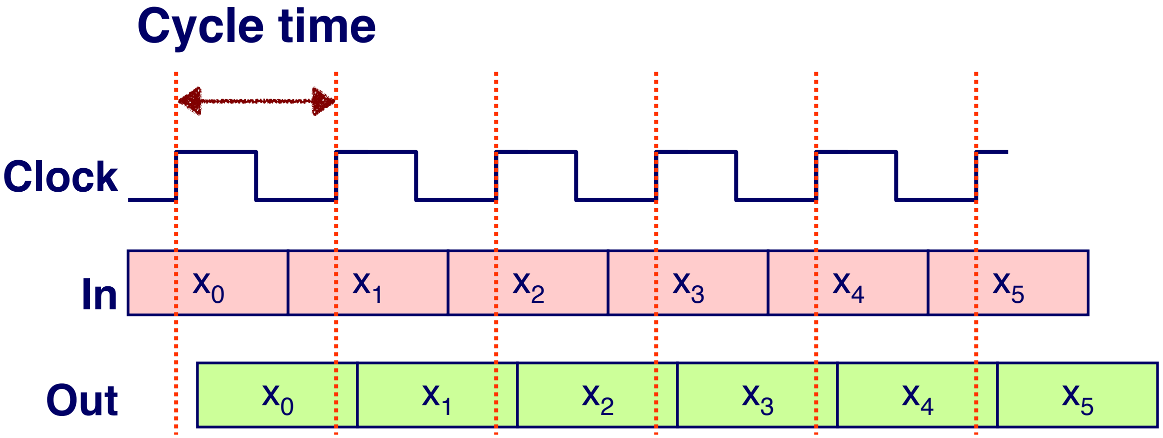
Clock Signal

- 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

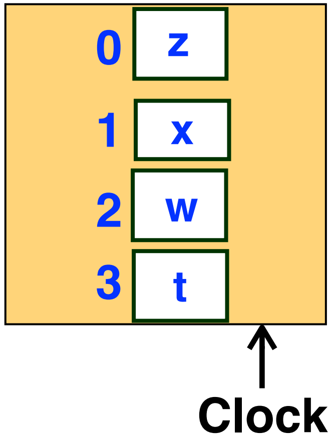
- Cycle time of a clock signal: the time duration between two rising edges.
- Frequency of a clock signal: how many rising (falling) edges in 1 second.
- 1 GHz CPU means the clock frequency is 1 GHz
  - The cycle time is 1/10^9 = 1 ns



Register File

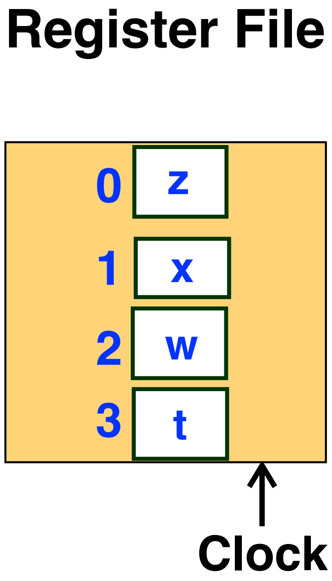
- A register file consists of a set of registers that you can individually read from and write to.

Register File



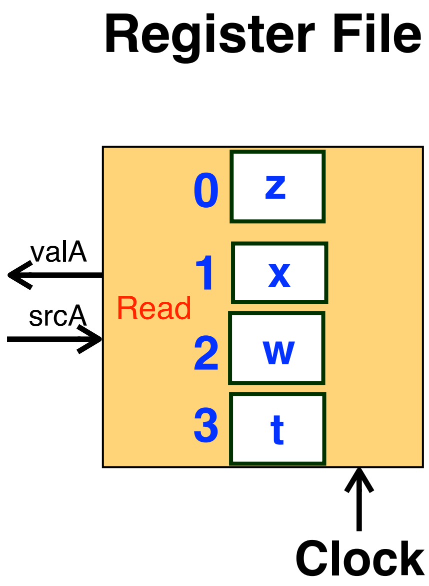
Register File

- 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



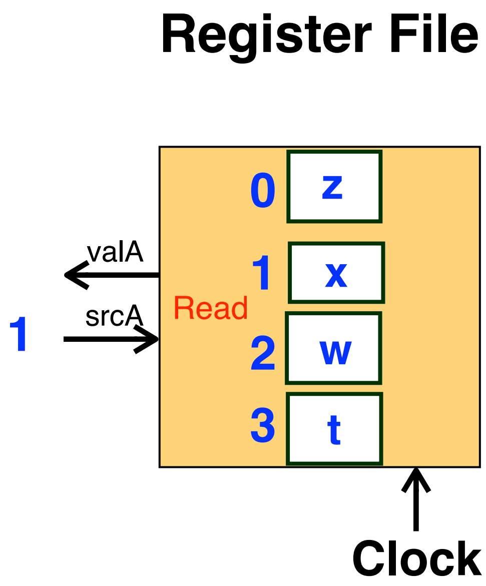
Register File

- 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



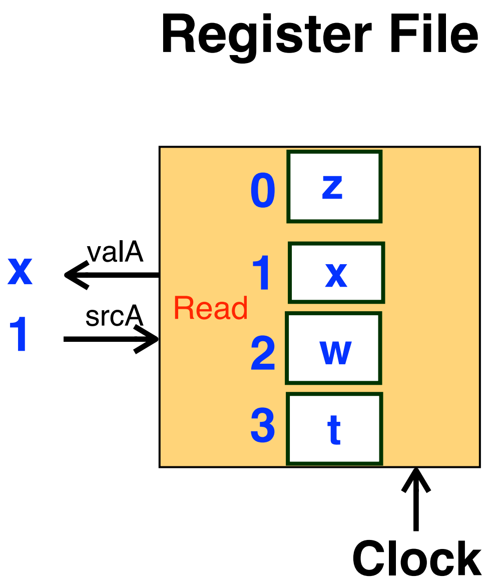
Register File

- 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



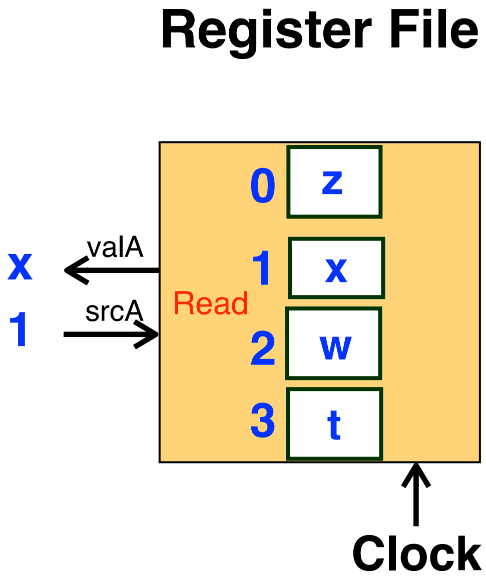
Register File

- 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

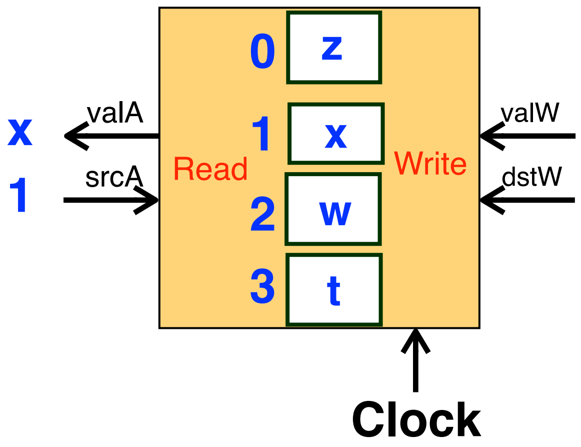
- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value



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

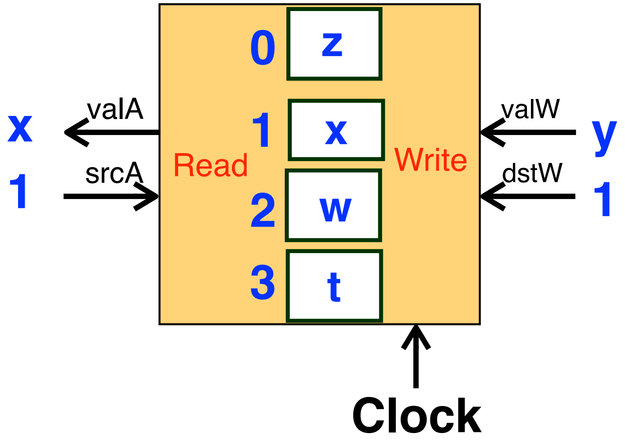
Register File



Register File

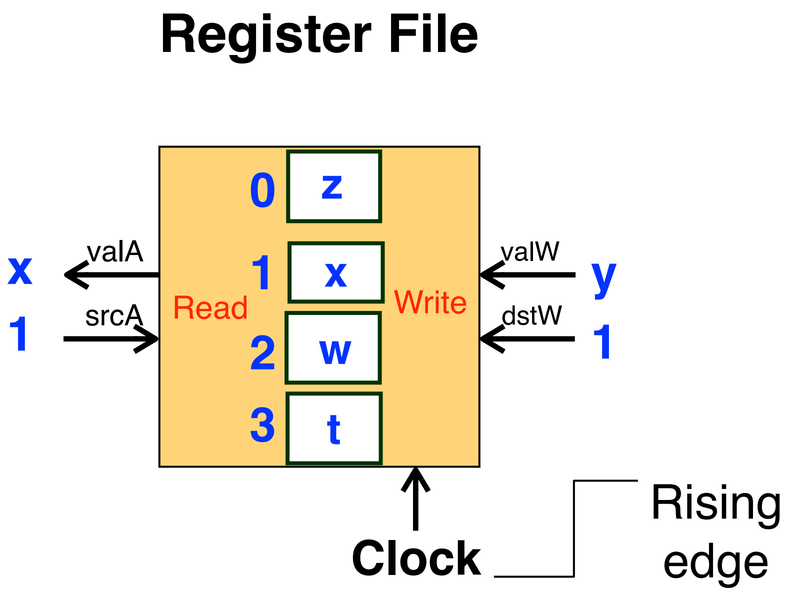
- A register file consists of a set of registers that you can individually read from and write to.
- To read: give a register file ID, and read the stored value out
- To write: give a register file ID, a new value, overwrite the old value

Register File



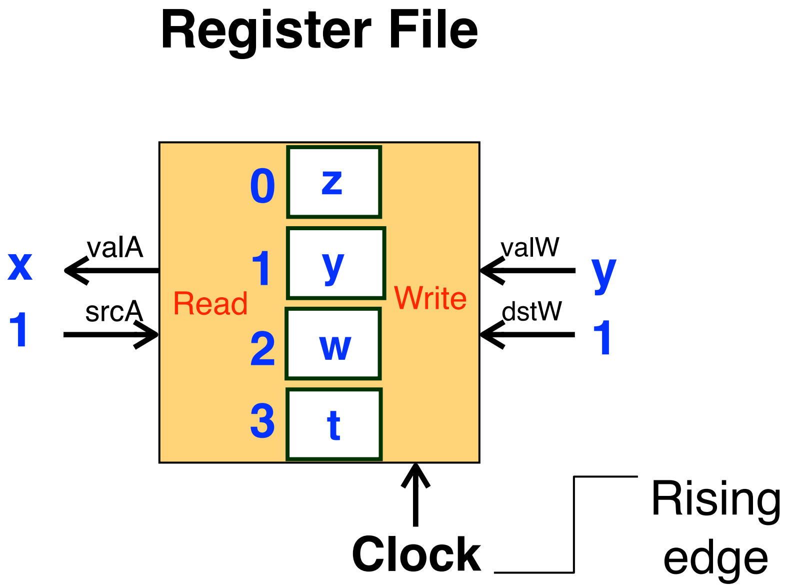
Register File

- 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



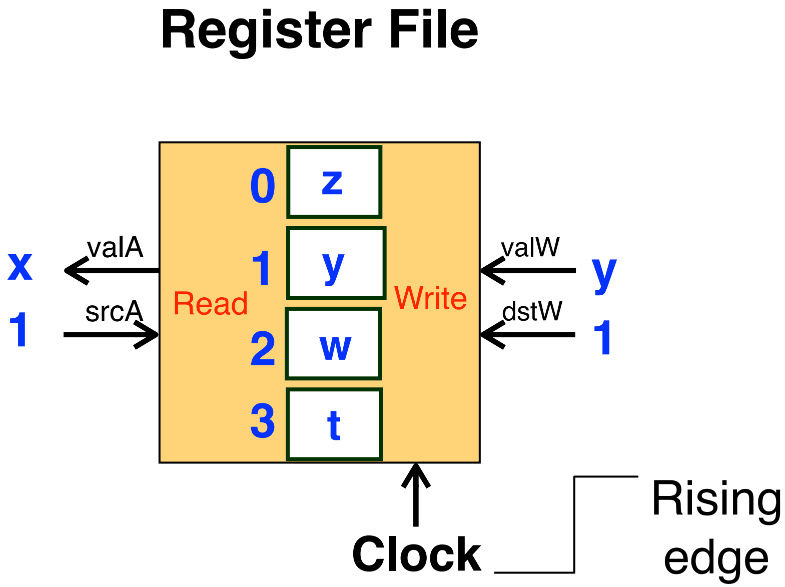
Register File

- 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



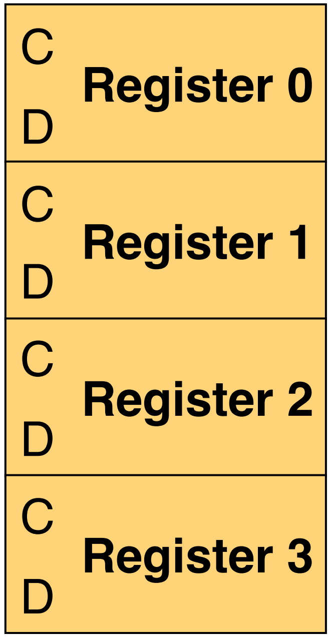
Register File

- 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
- How do we build a register file out of individual registers??



Register File Read

- Continuously read a register independent of the clock signal



C
Register 0
D

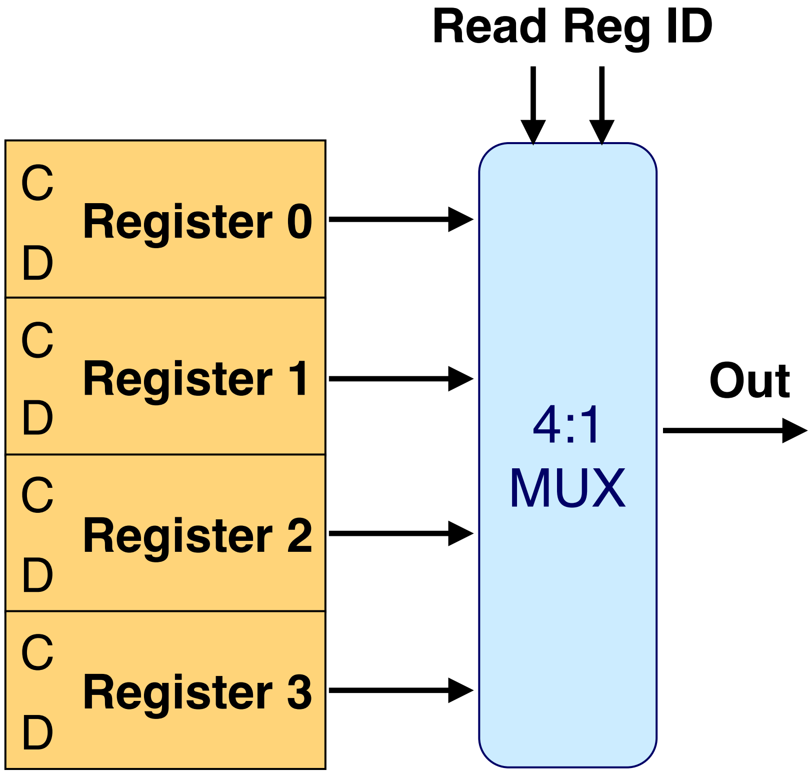
C
Register 1
D

C
Register 2
D

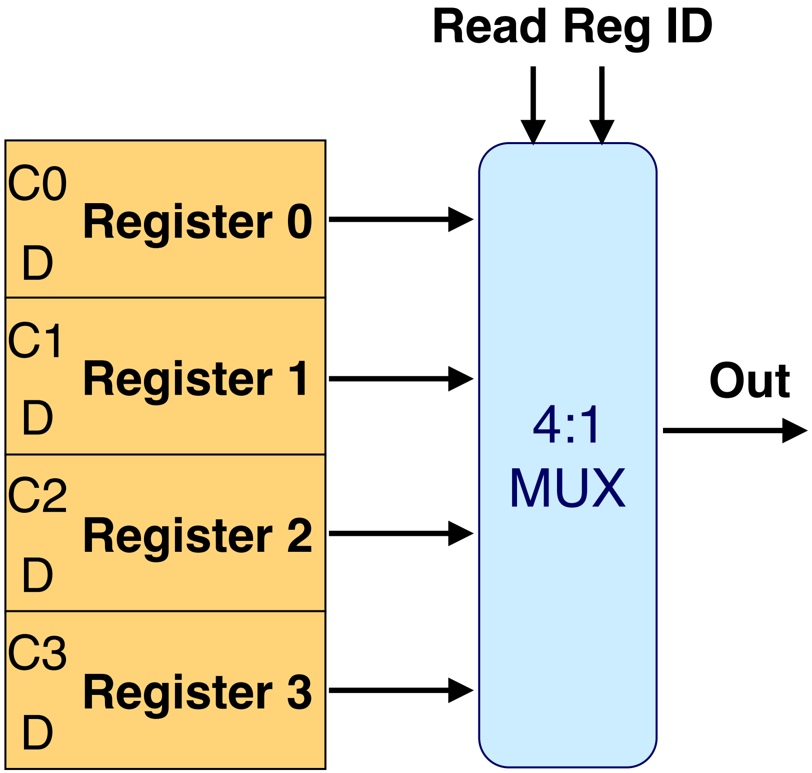
C
Register 3
D

Register File Read

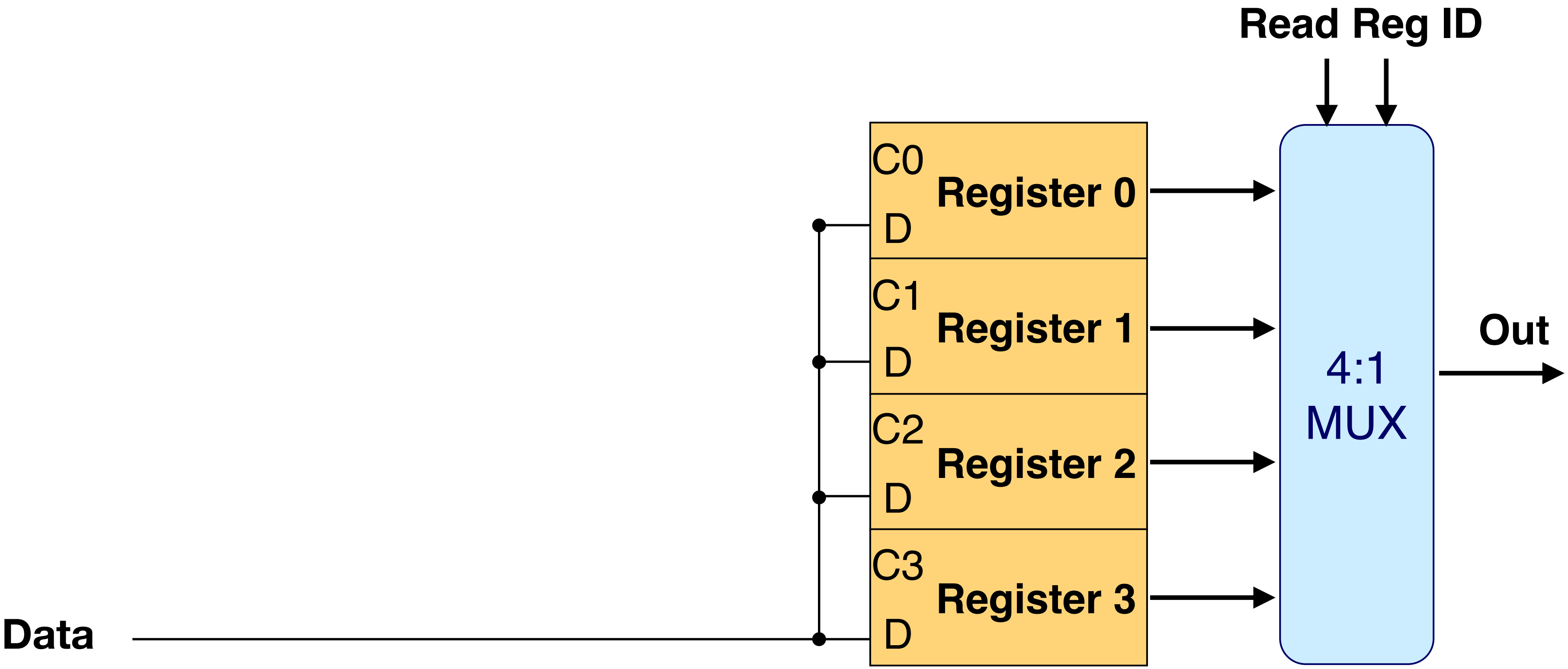
- Continuously read a register independent of the clock signal



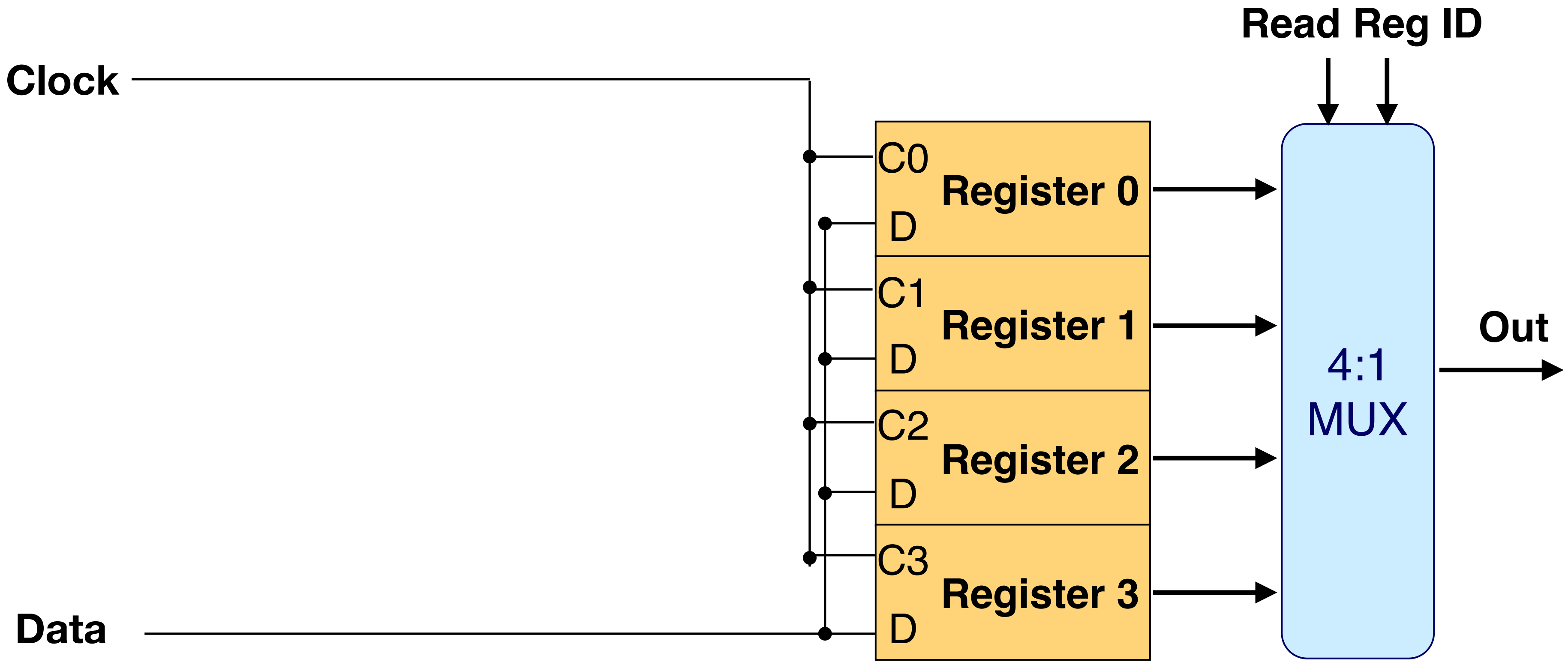
Register File Write



Register File Write

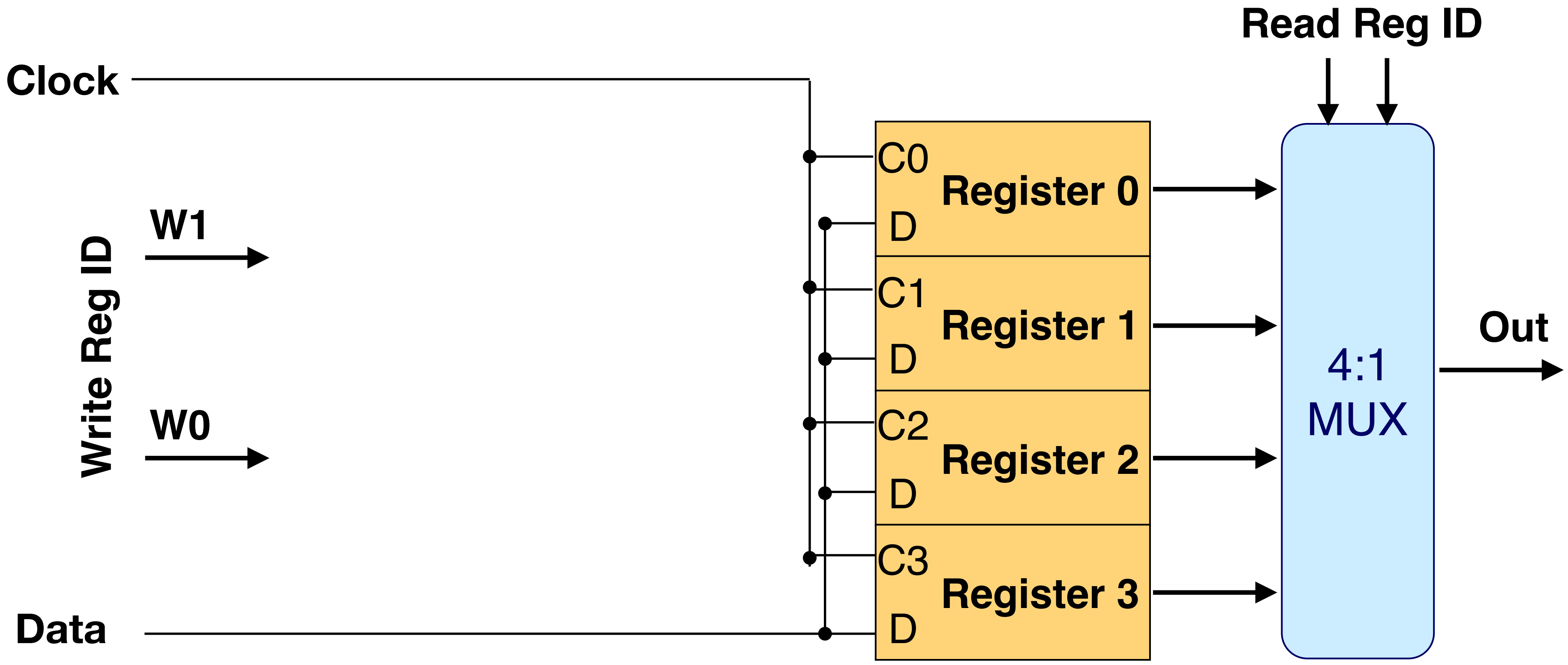


Register File Write



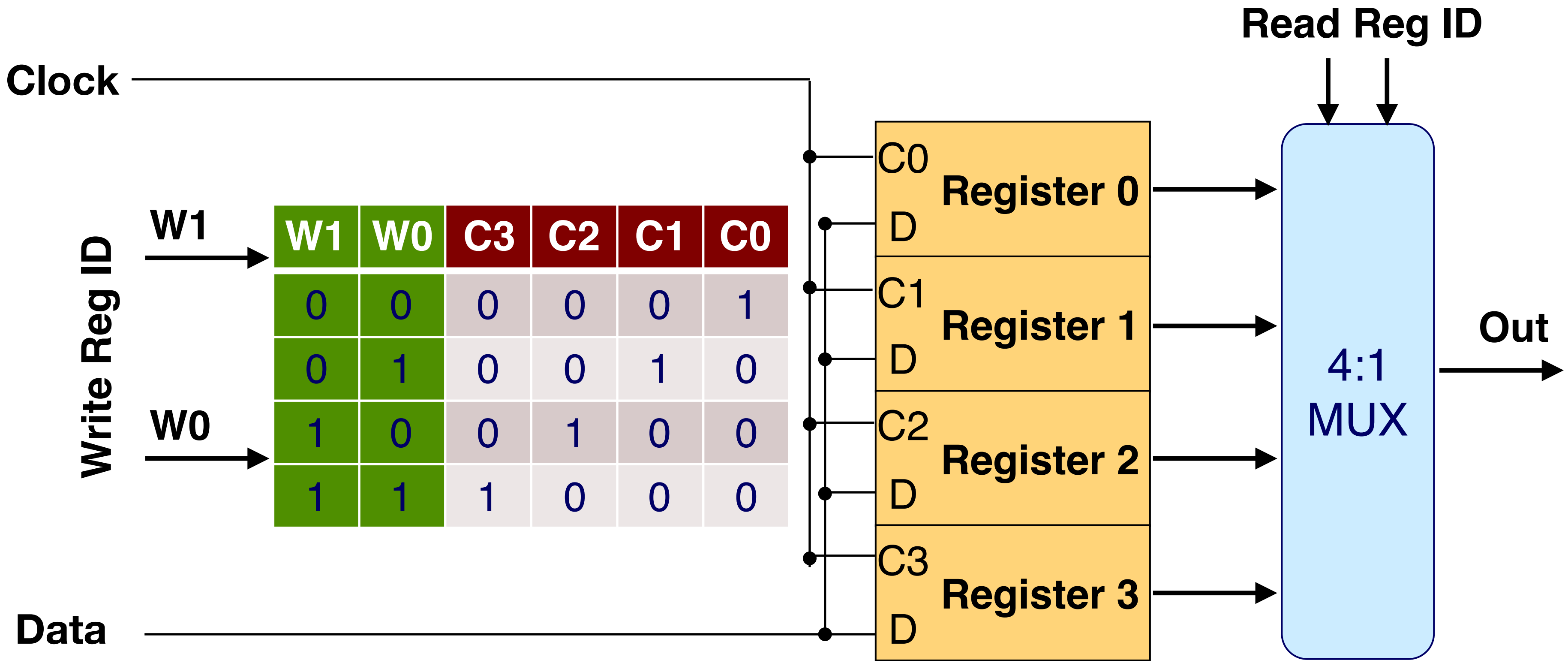
Register File Write

- Only write the a specific register when the clock rises. How??



Register File Write

- Only write the a specific register when the clock rises. How??



W1	W0	C3	C2	C1	C0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Decoder

W1	W0	C3	C2	C1	C0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

W0

W1

C0

C1

C2

C3

Decoder

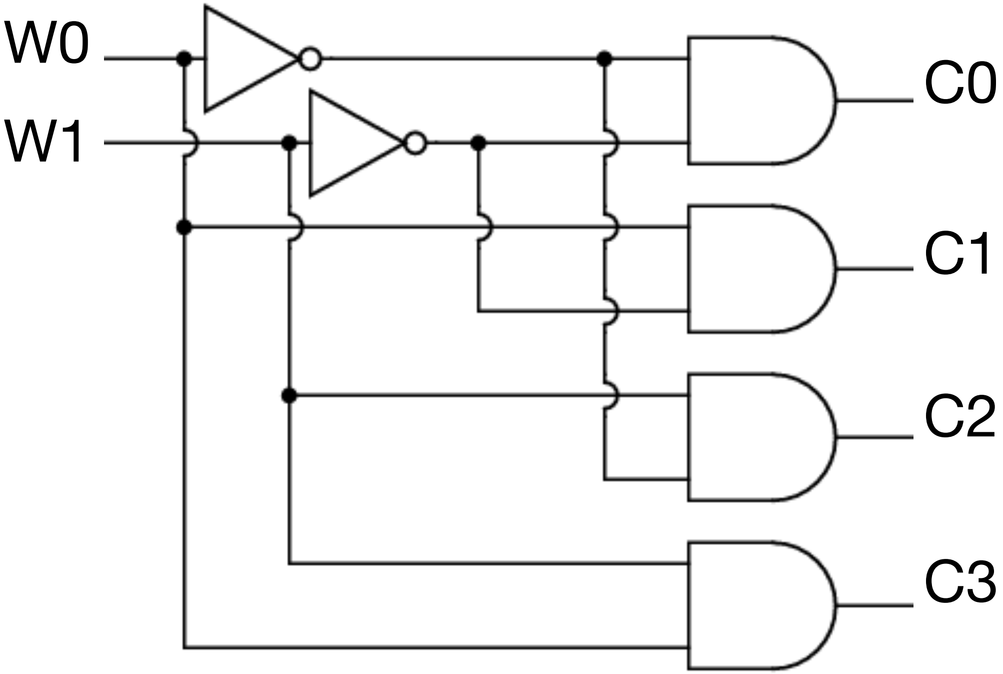
W1	W0	C3	C2	C1	C0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

C0 = !W1 & !W0
C1= !W1 & W0
C2 = W1 & !W0
C3 = W1 & W0

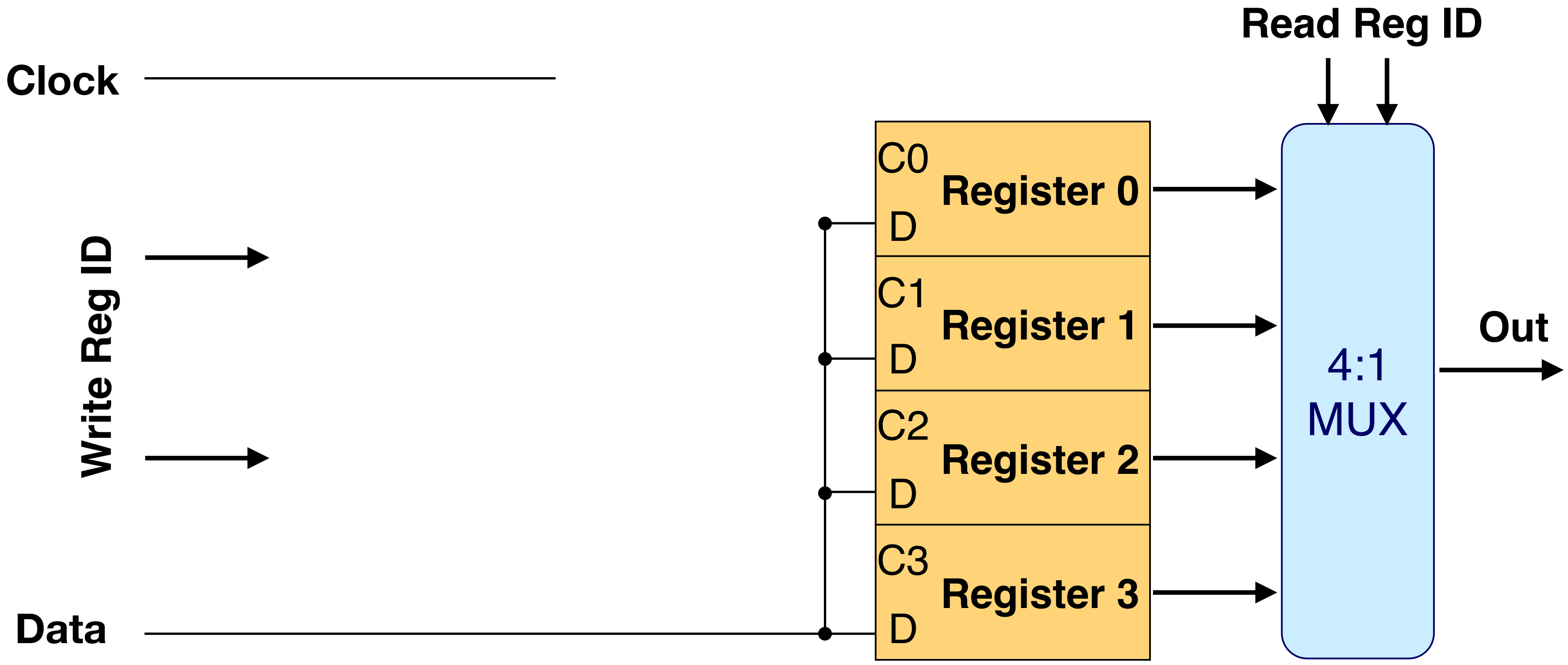
Decoder

W1	W0	C3	C2	C1	C0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

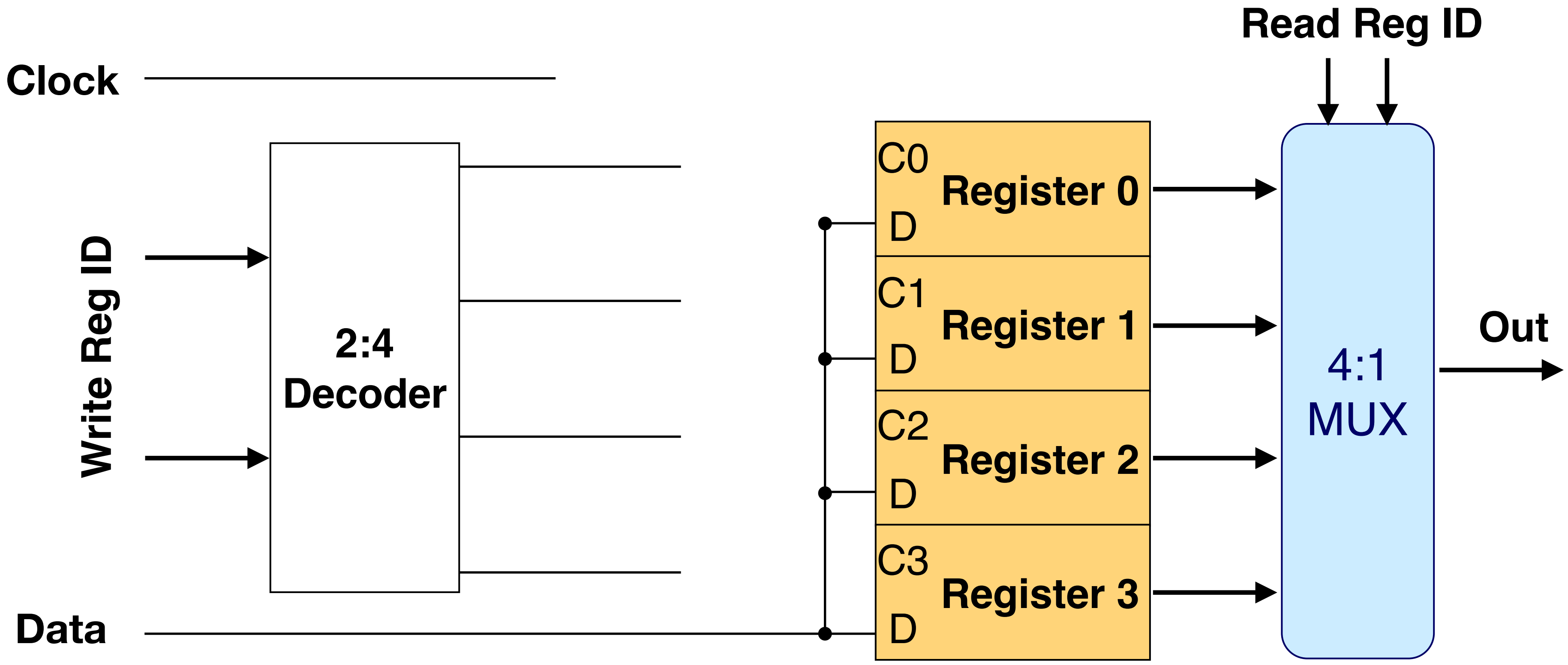
C0 = !W1 & !W0
C1= !W1 & W0
C2 = W1 & !W0
C3 = W1 & W0



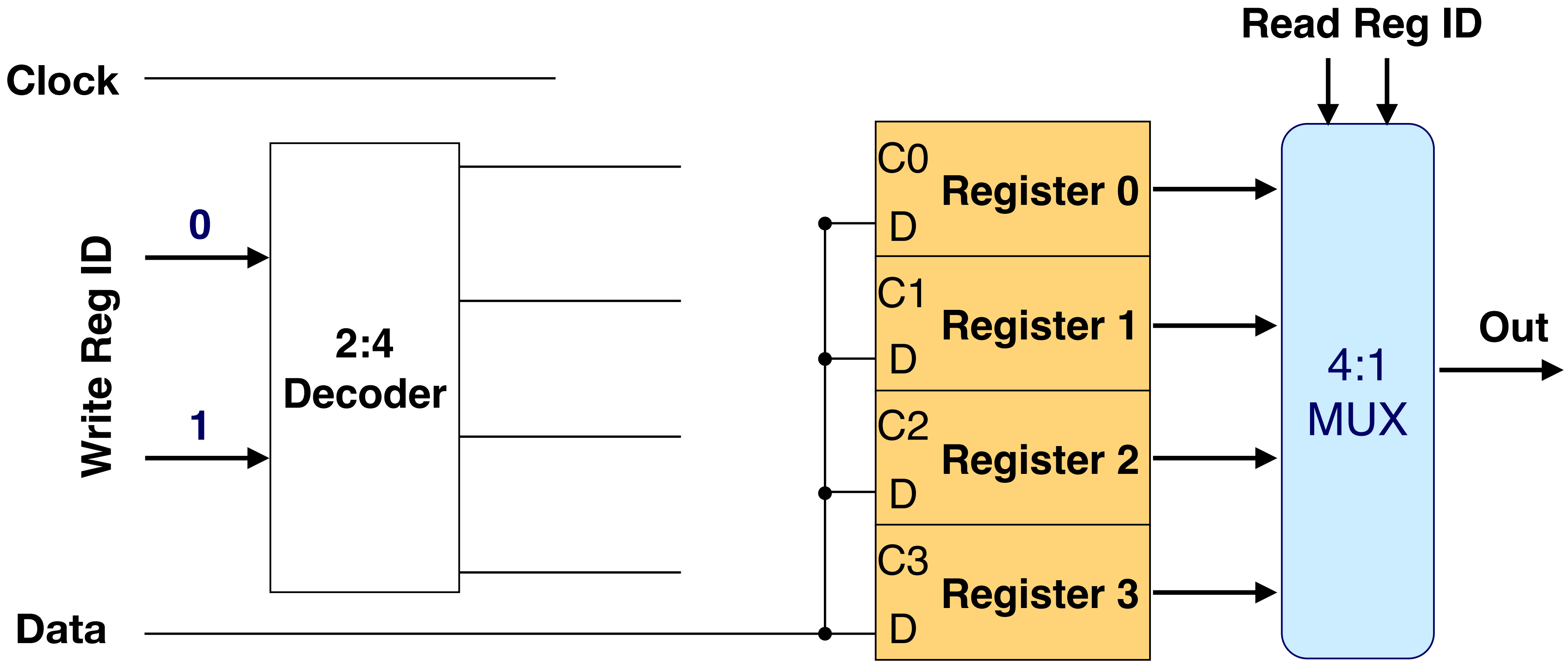
Register File Write



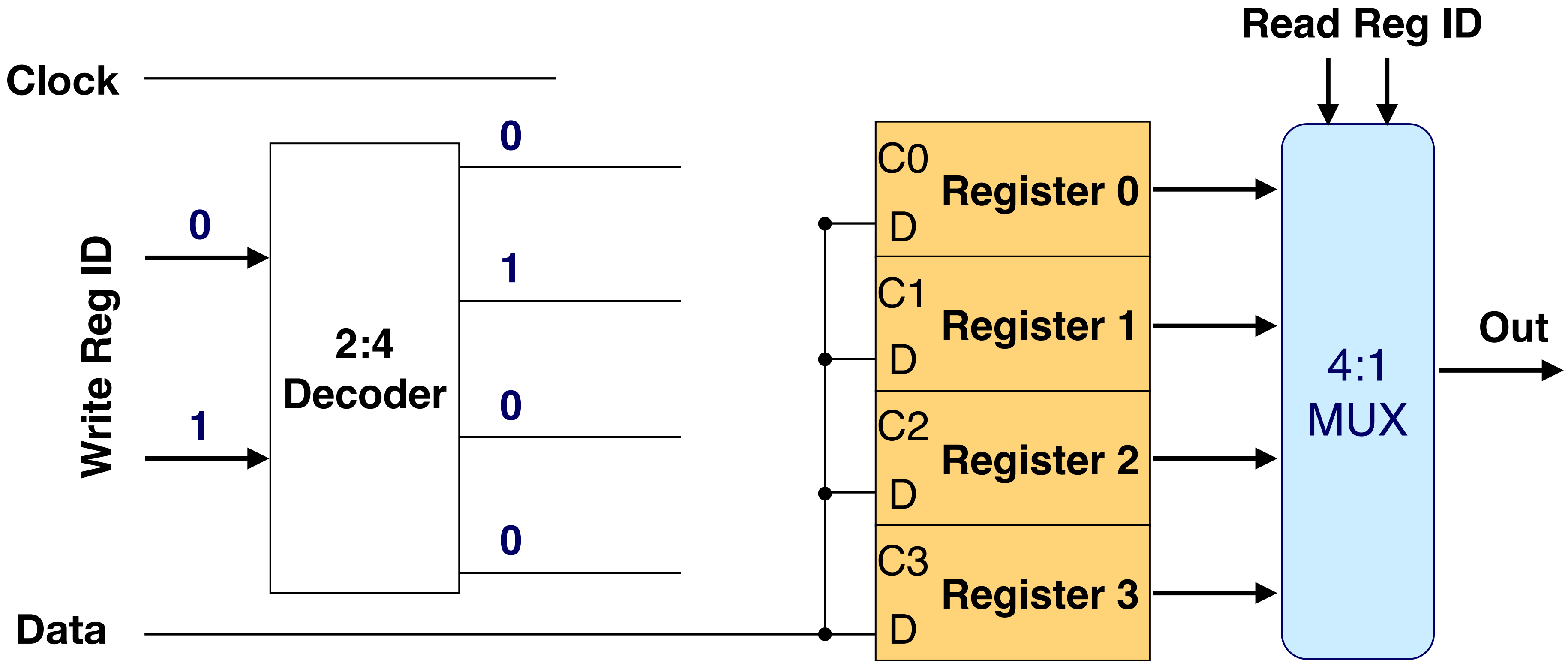
Register File Write



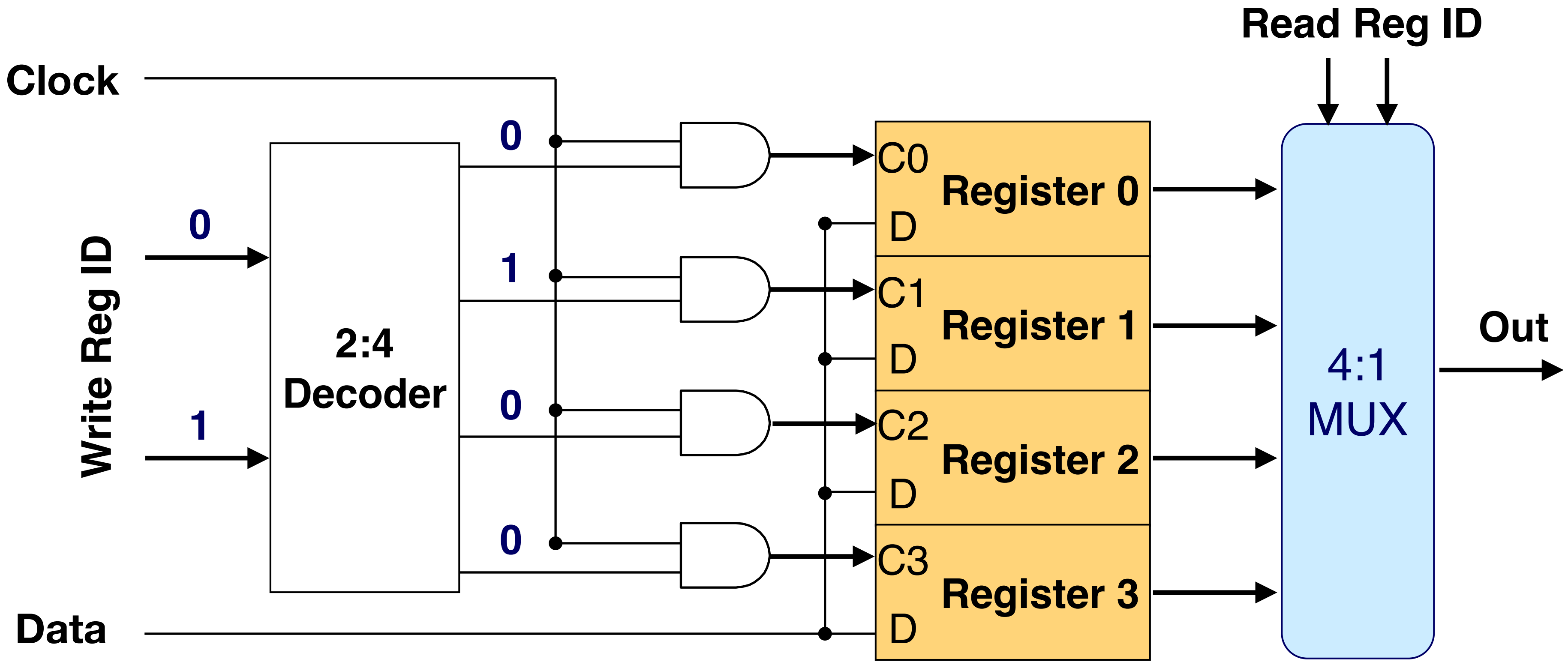
Register File Write



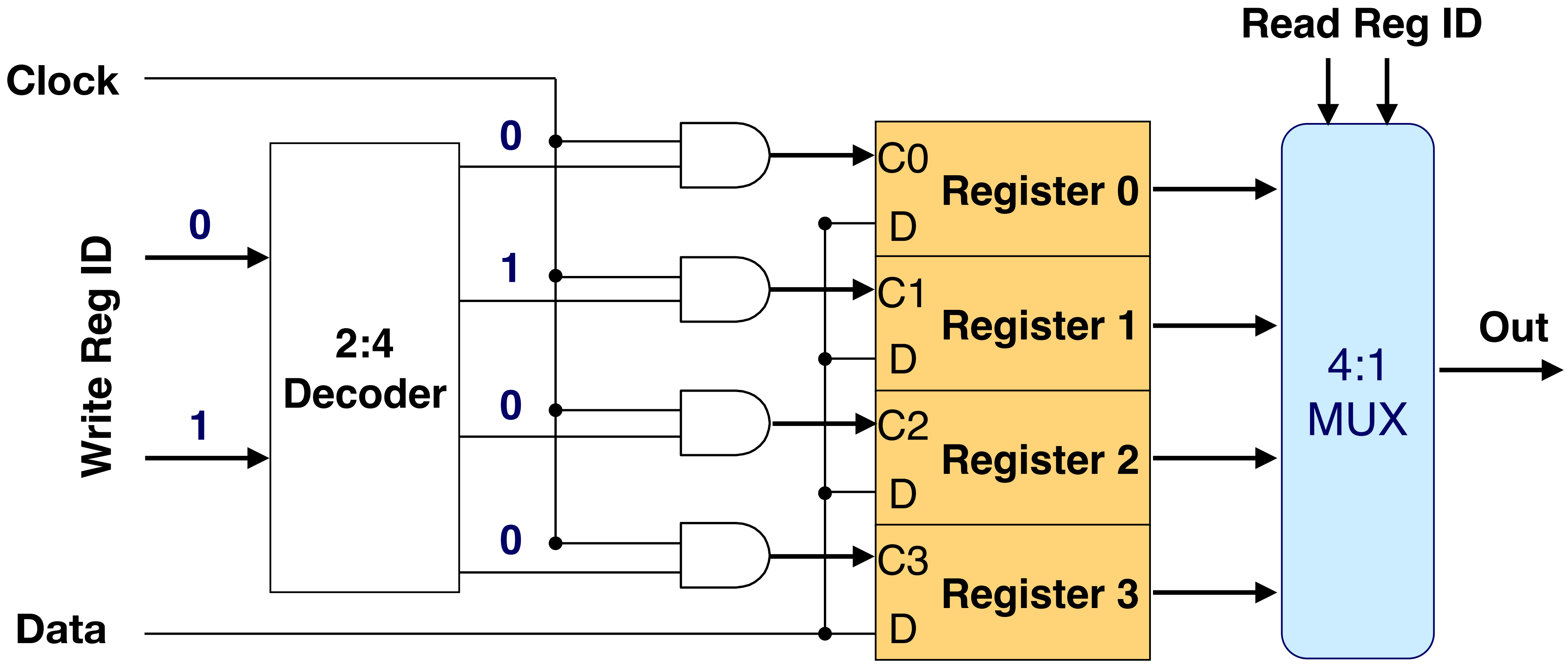
Register File Write



Register File Write



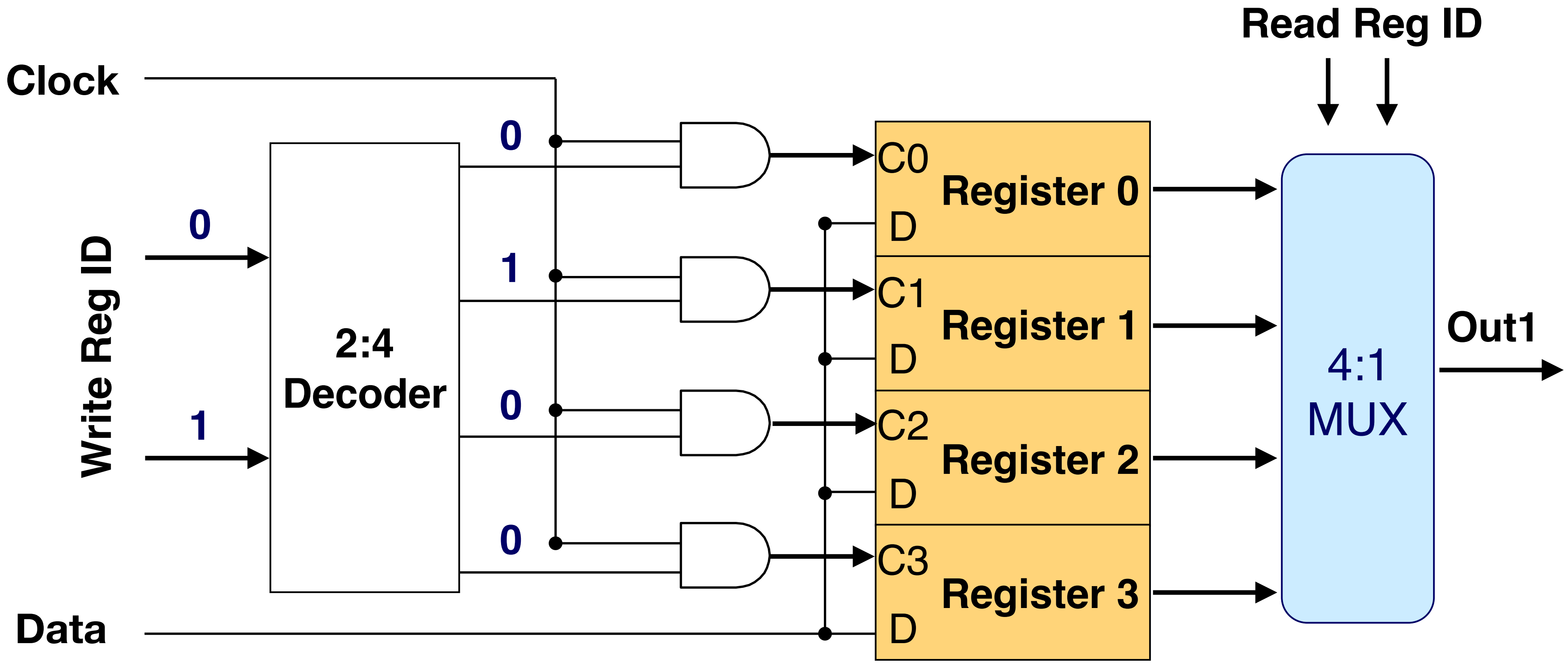
Register File Write



- This implementation can read 1 register and write 1 register at the same time: 1 read port and 1 write port

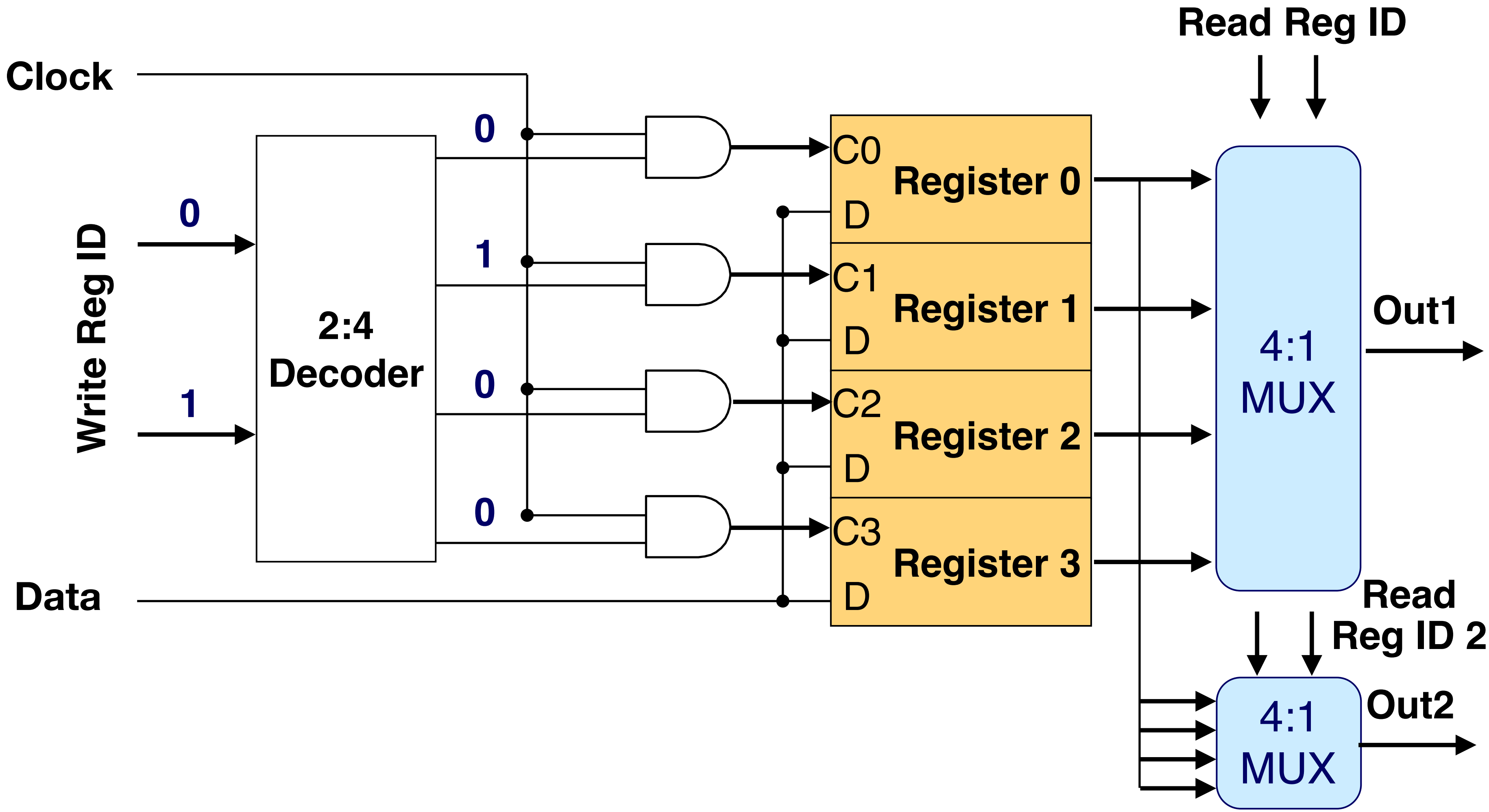
Multi-Port Register File

- What if we want to read multiple registers at the same time?



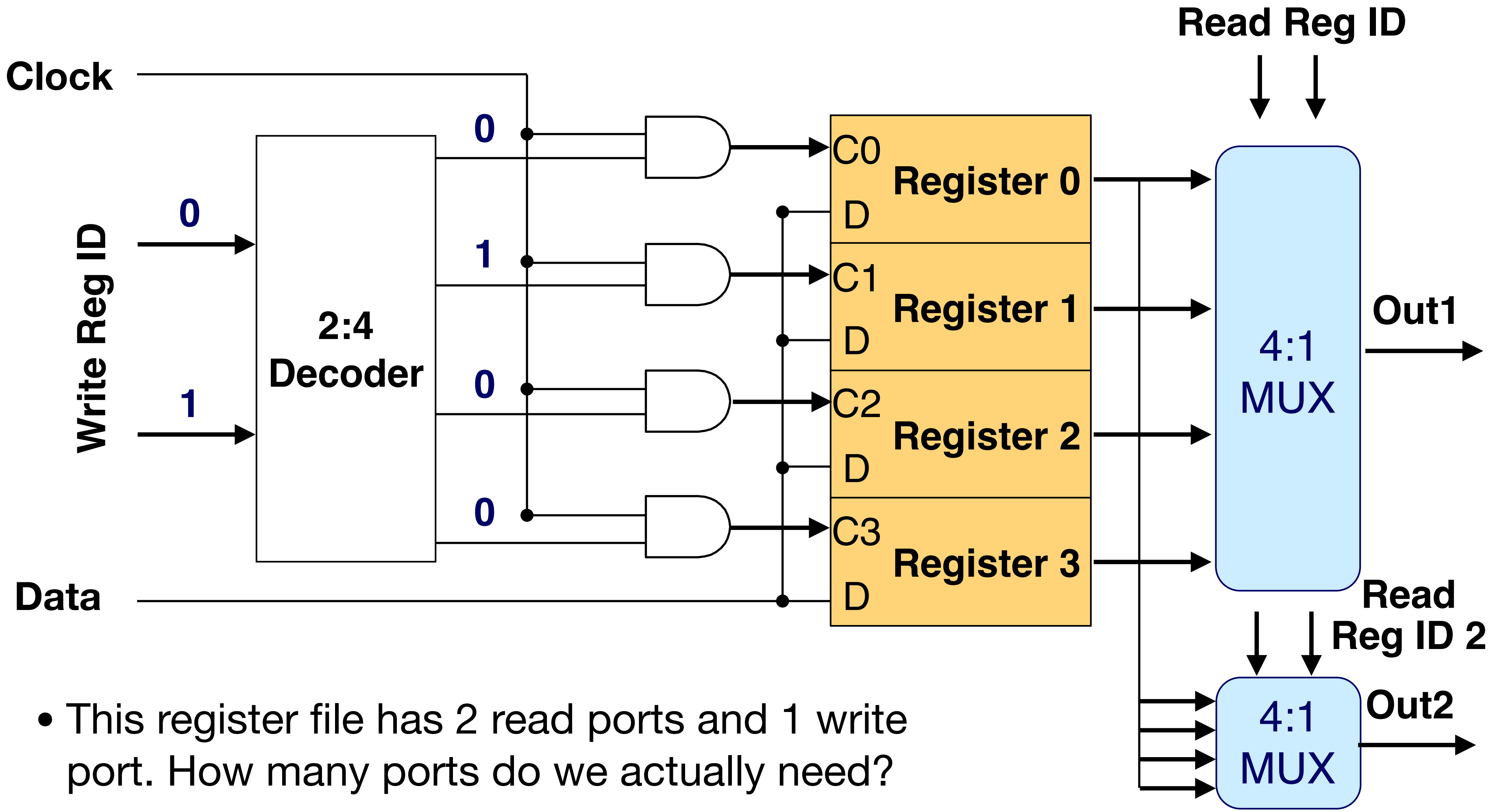
Multi-Port Register File

- What if we want to read multiple registers at the same time?



Multi-Port Register File

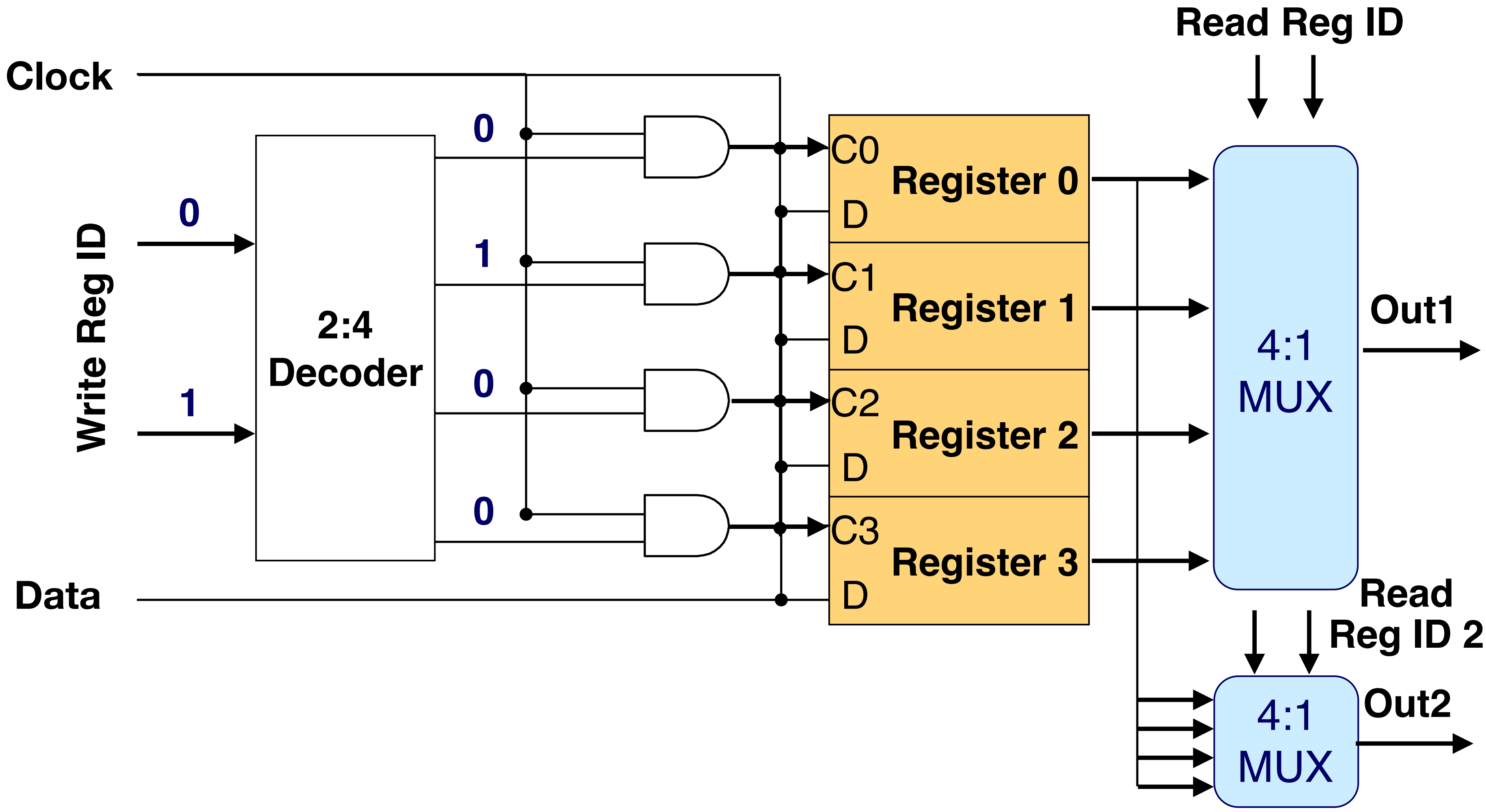
- What if we want to read multiple registers at the same time?



- This register file has 2 read ports and 1 write port. How many ports do we actually need?

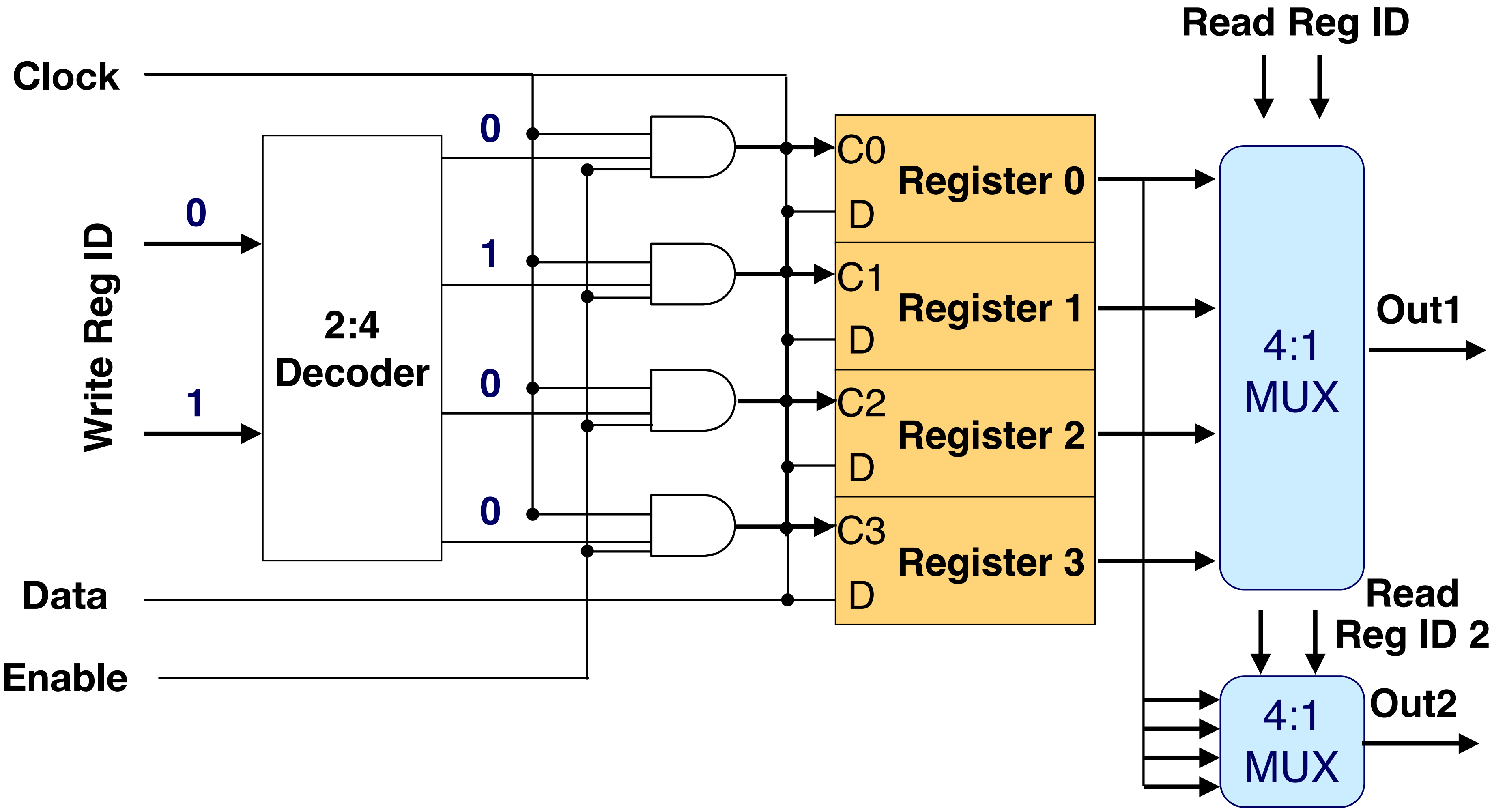
Multi-Port Register File

- Is this correct? What if we don’t want to write anything?



Multi-Port Register File

- Is this correct? What if we don't want to write anything?

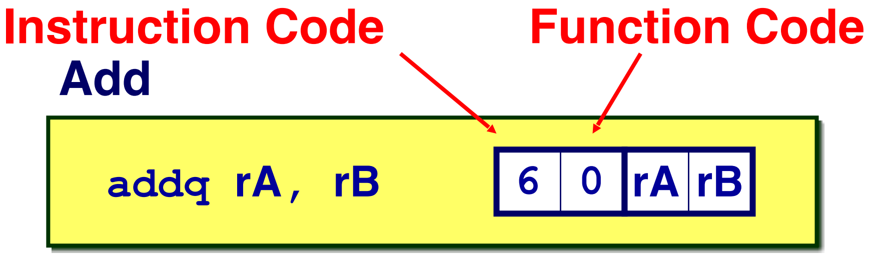


Processor Microarchitecture

- 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

Function Code

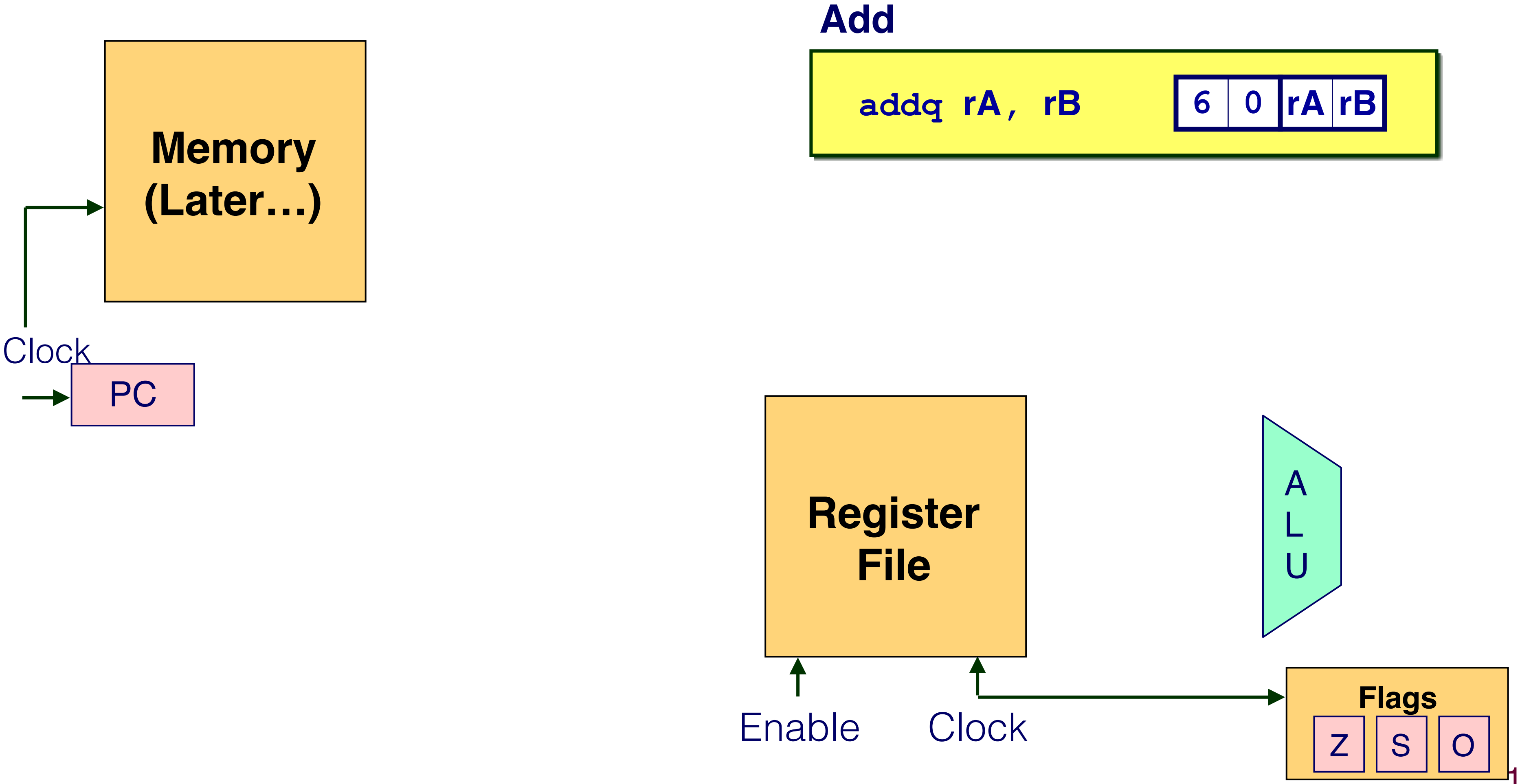
Add

`addq rA, rB`

6	0	rA	rB

Executing an ADD instruction

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



Add

addq rA, rB

6	0	rA	rB

Memory (Later…)

Clock

PC

Register File

Enable

Clock

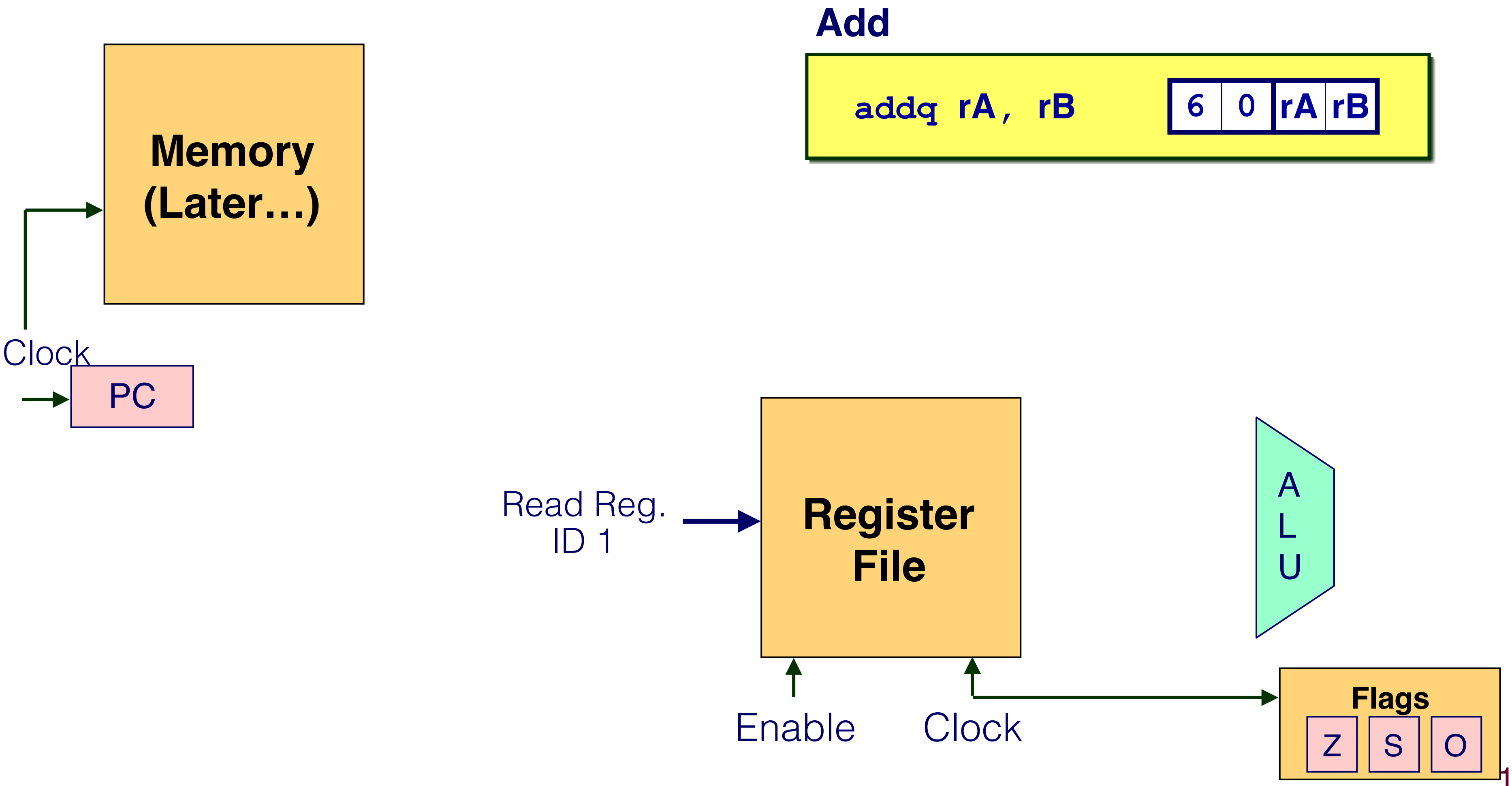
ALU

Flags

Z S O

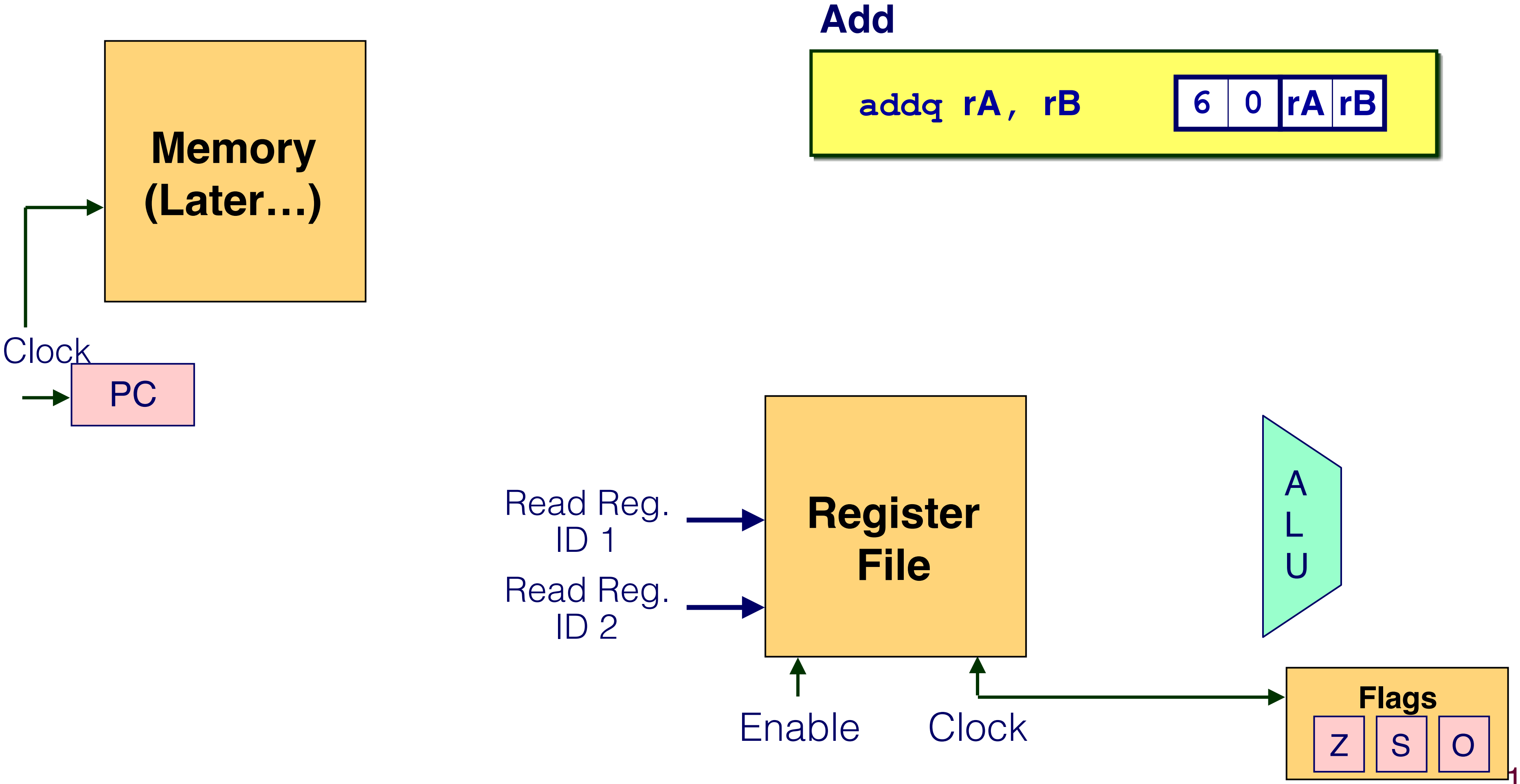
Executing an ADD instruction

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



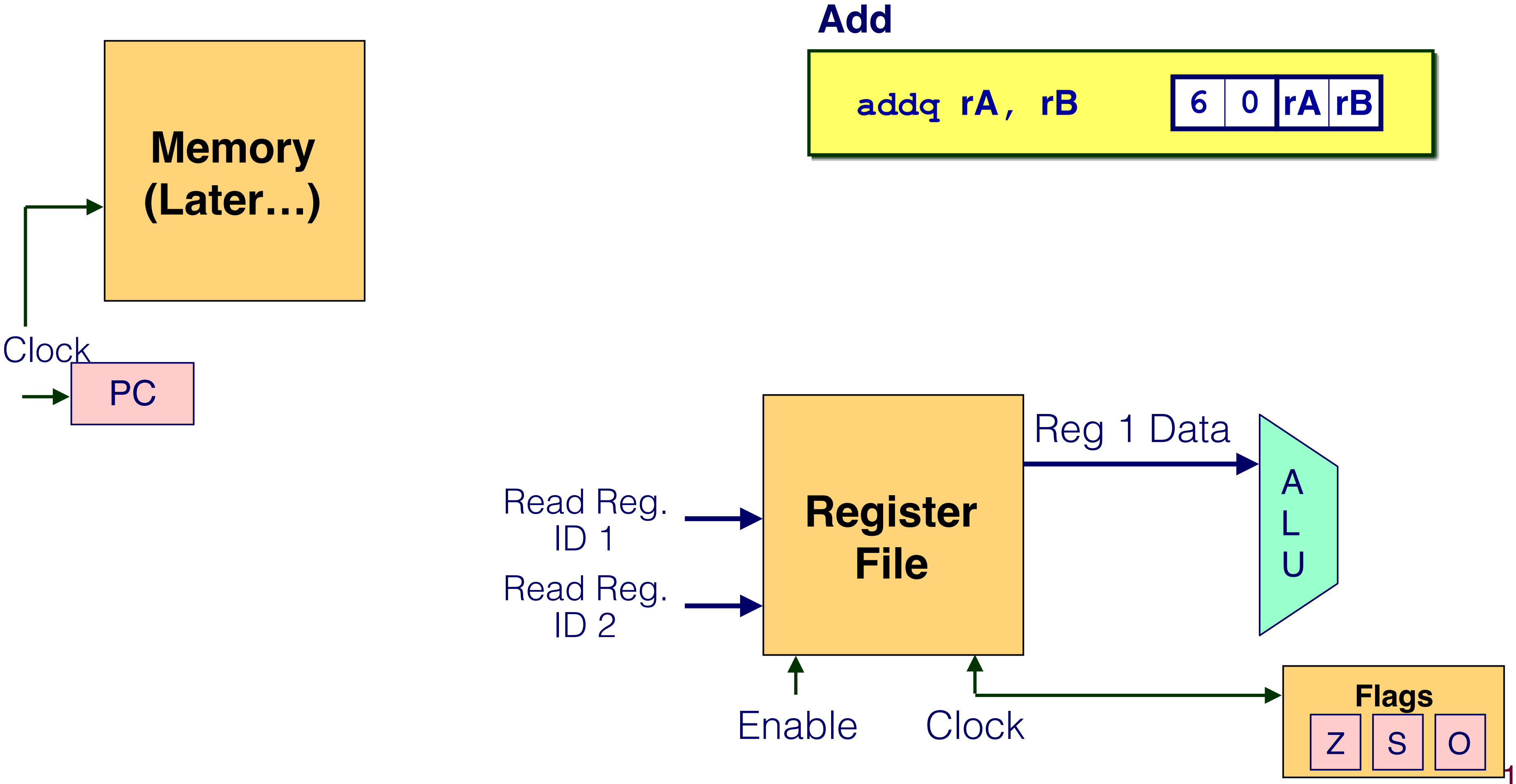
Executing an ADD instruction

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



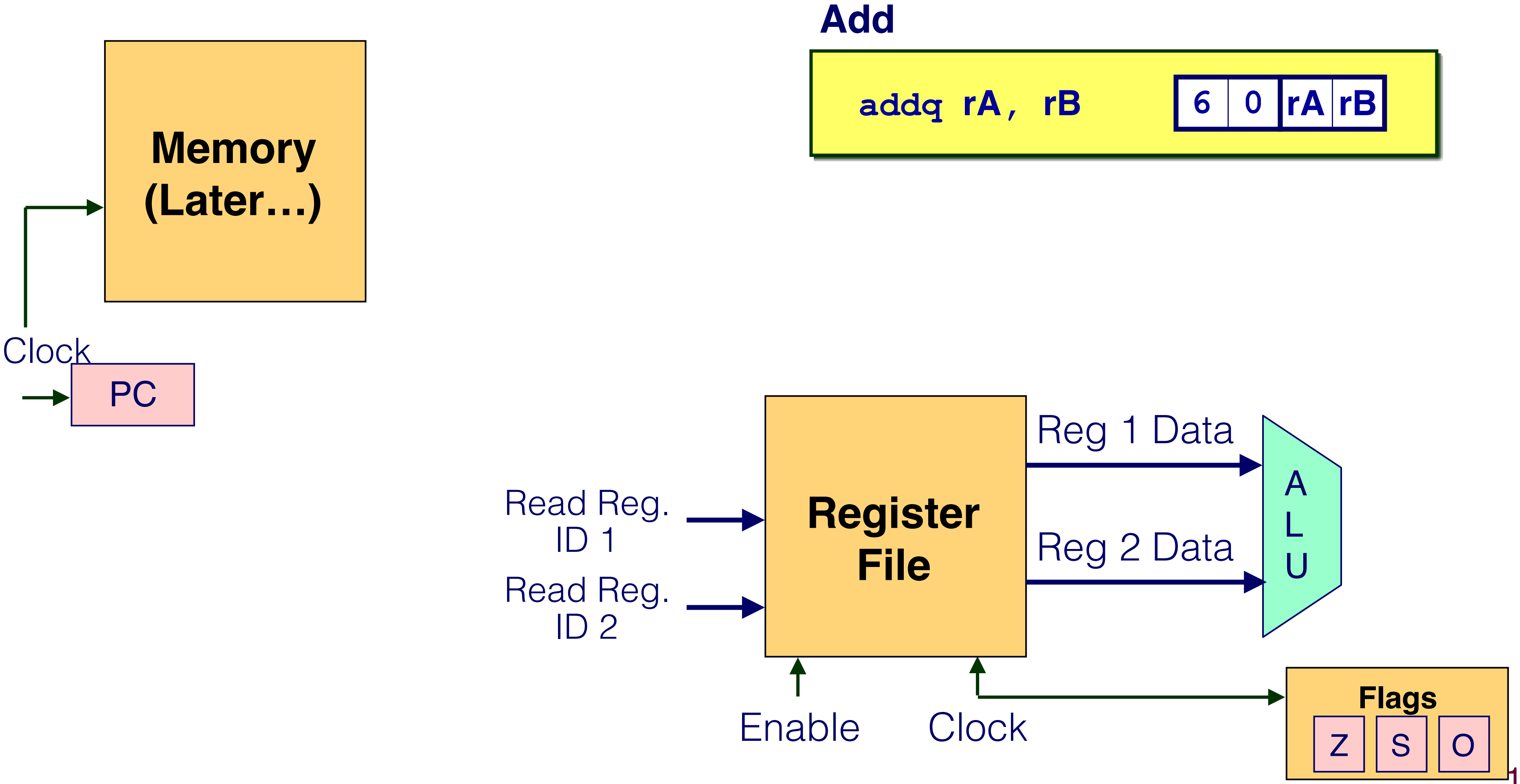
Executing an ADD instruction

- 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`



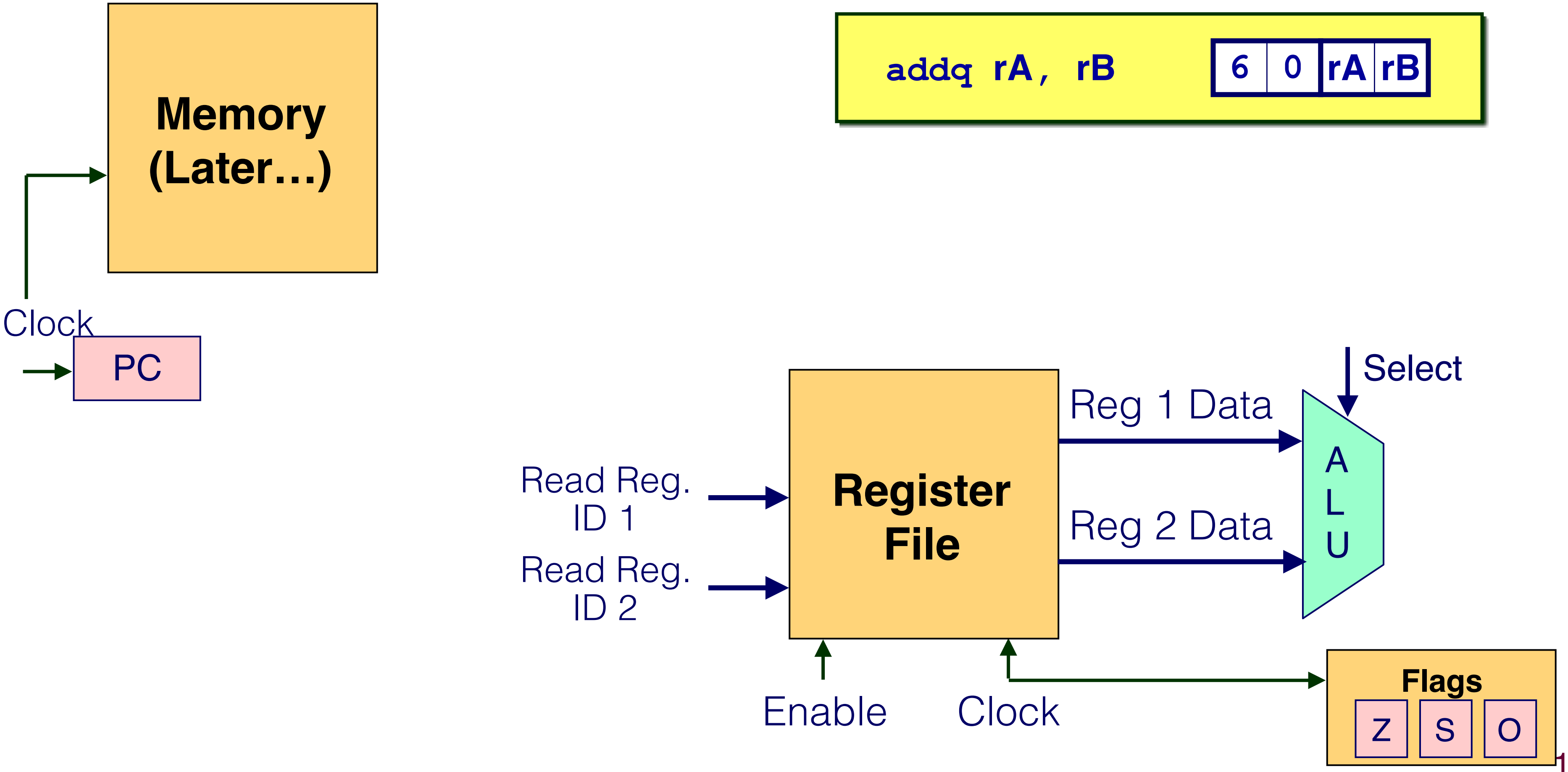
Executing an ADD instruction

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

Add

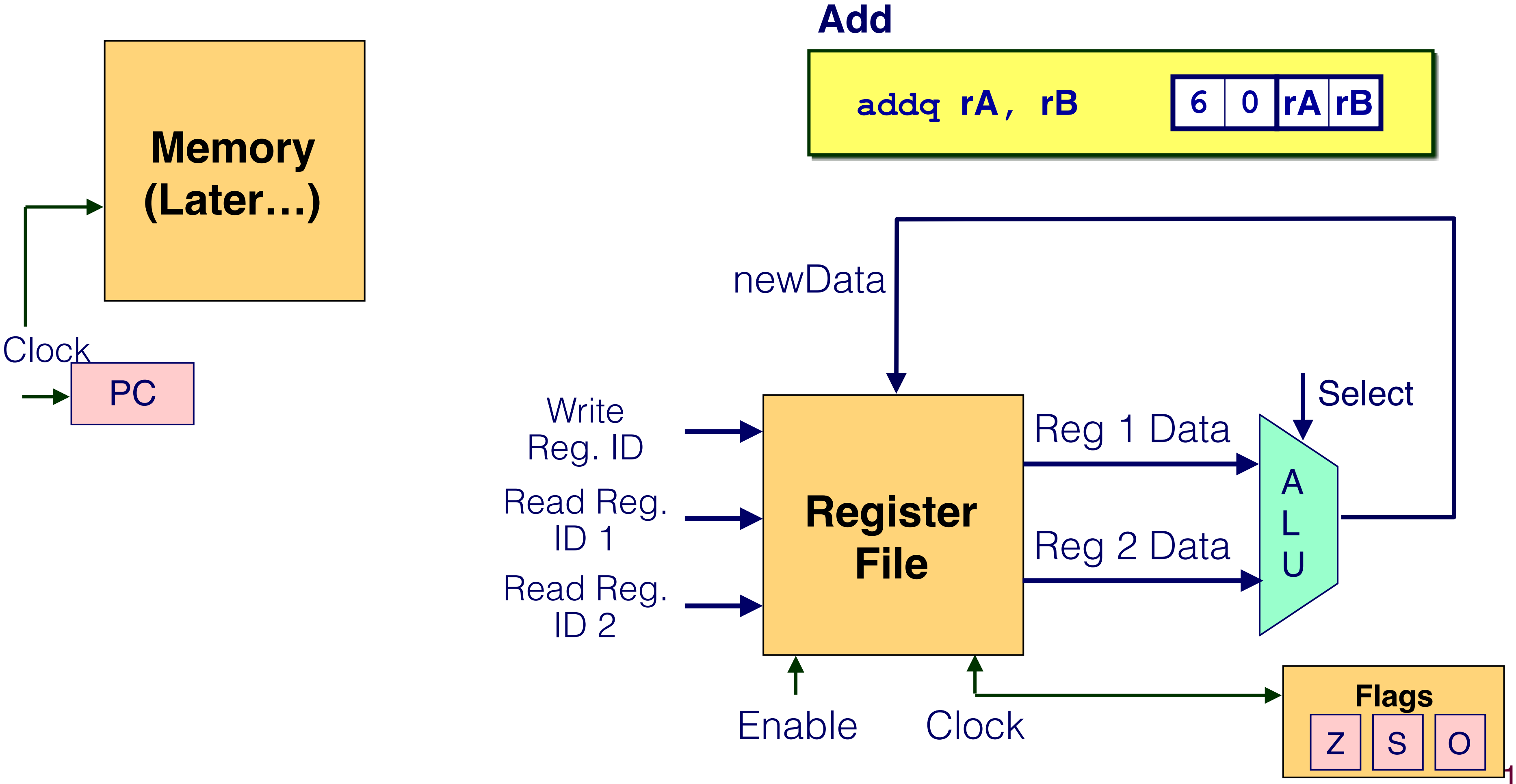
`addq rA, rB`

6	0	rA	rB



Executing an ADD instruction

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



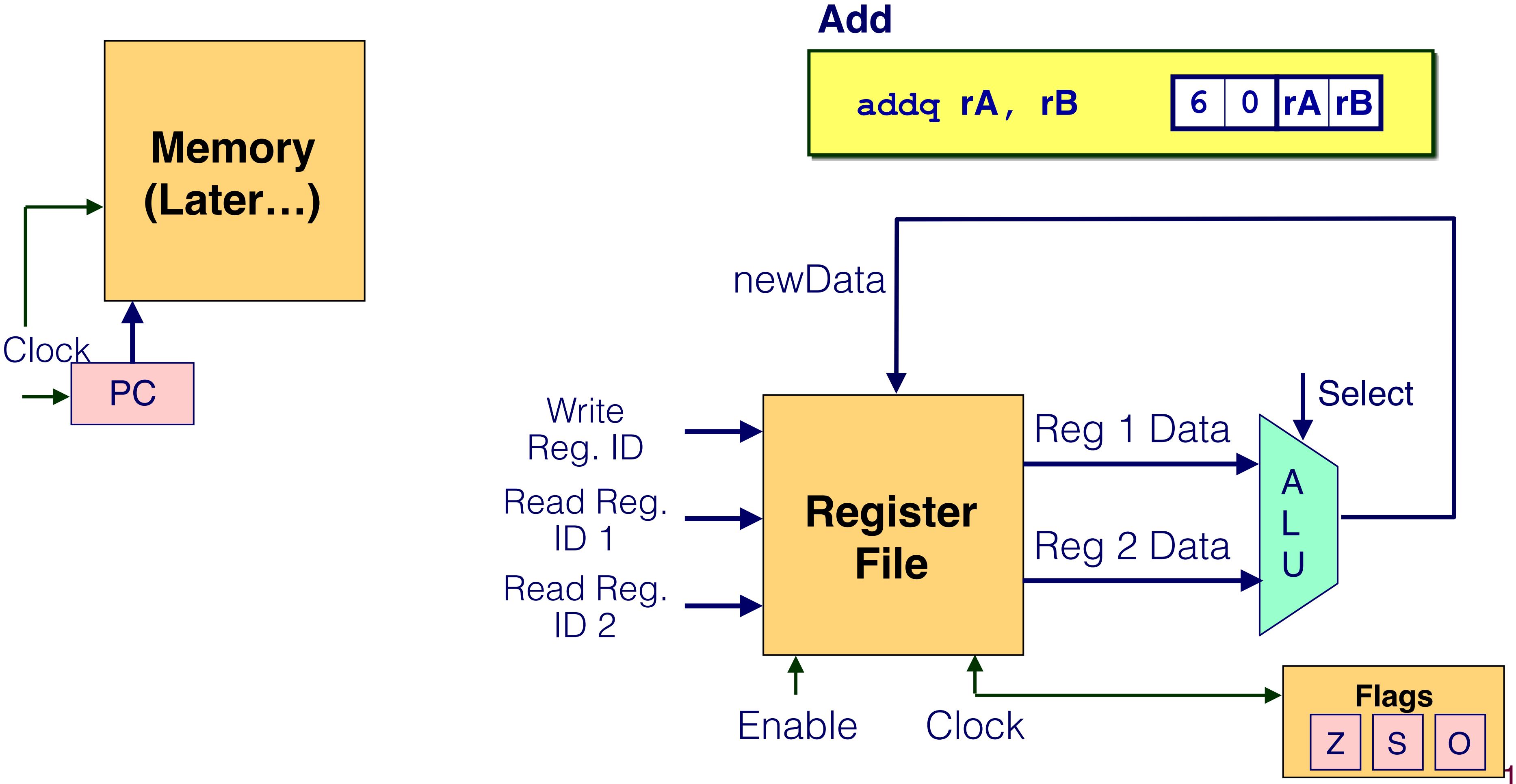
Add

`addq rA, rB`

6	0	rA	rB

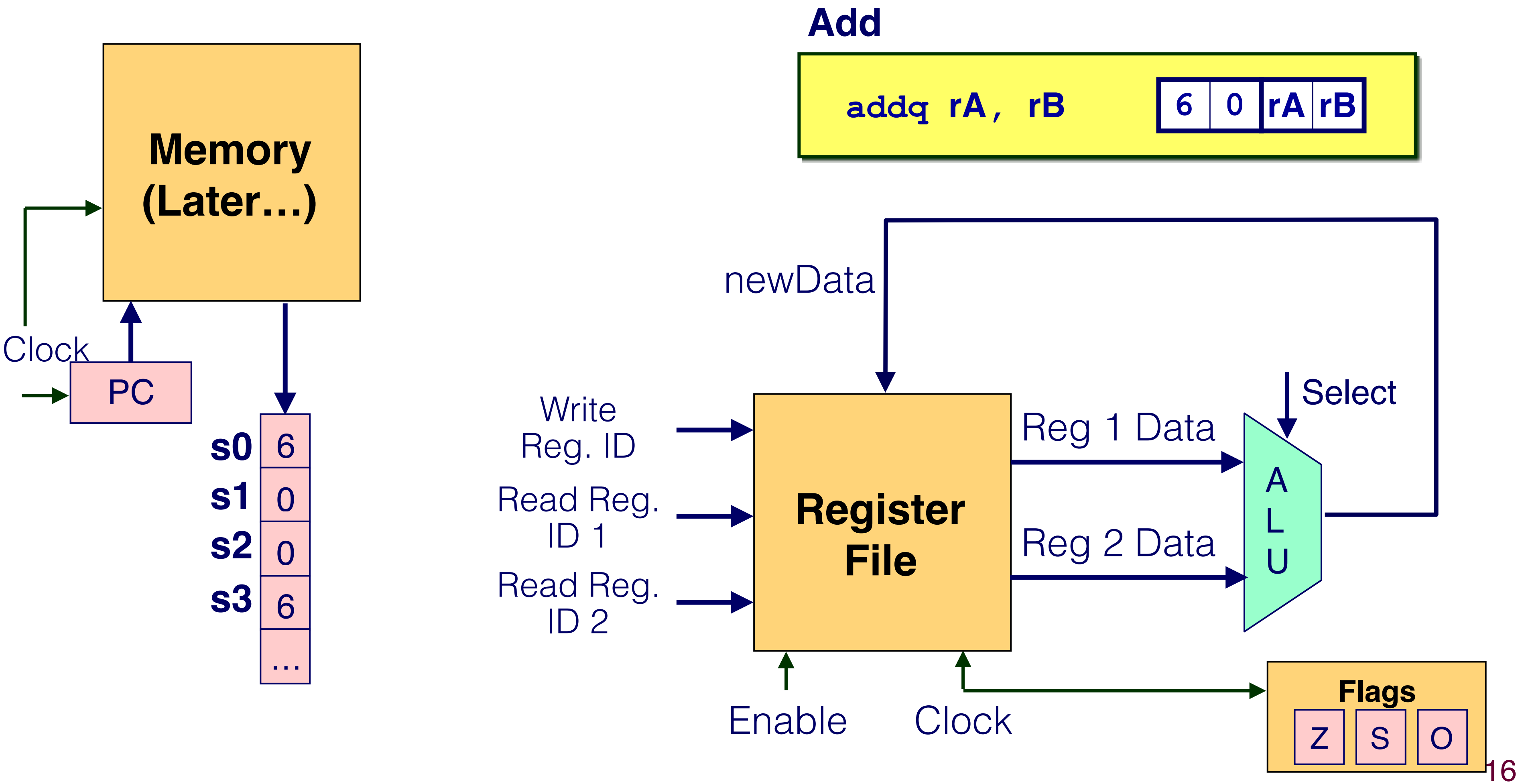
Executing an ADD instruction

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



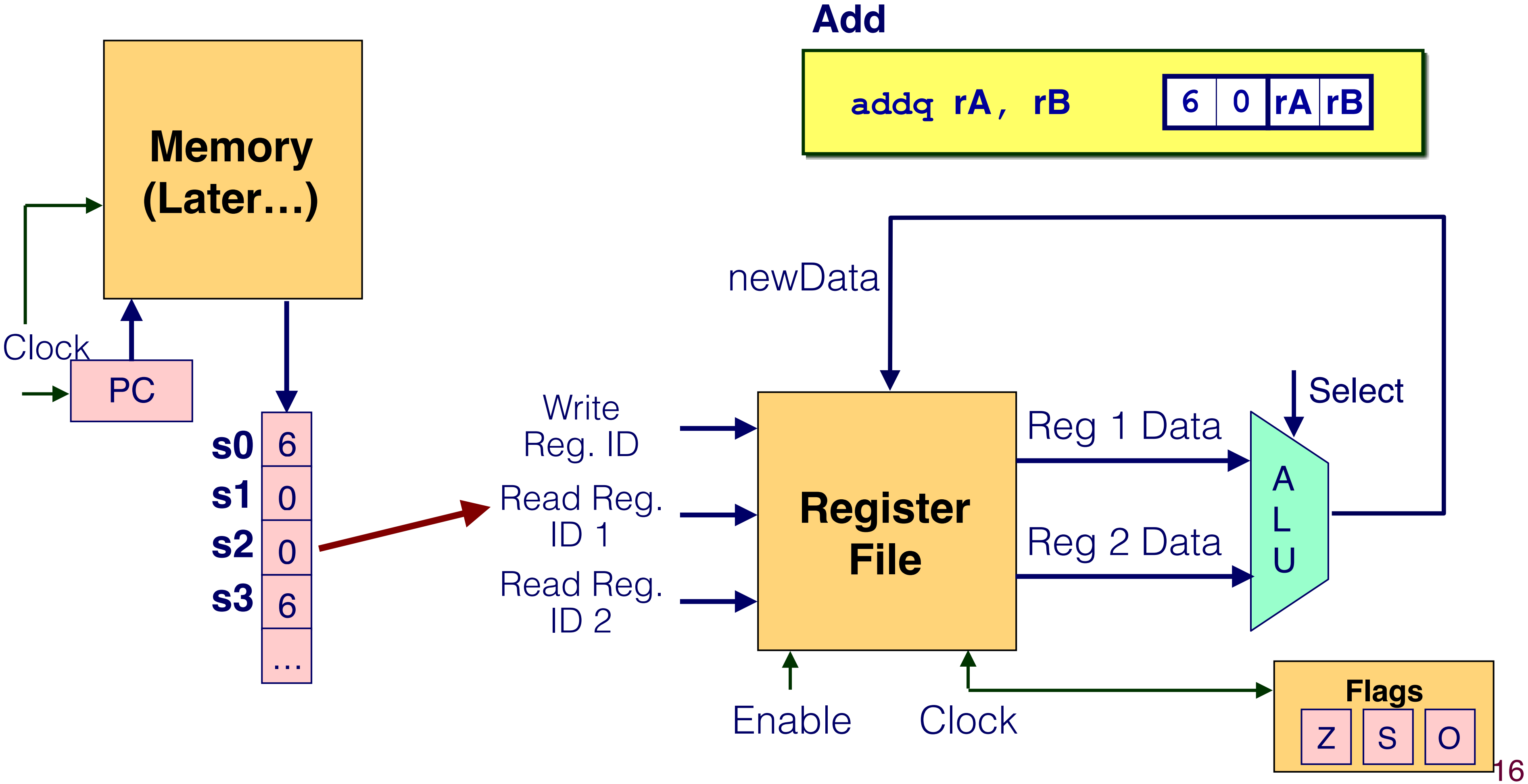
Executing an ADD instruction

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



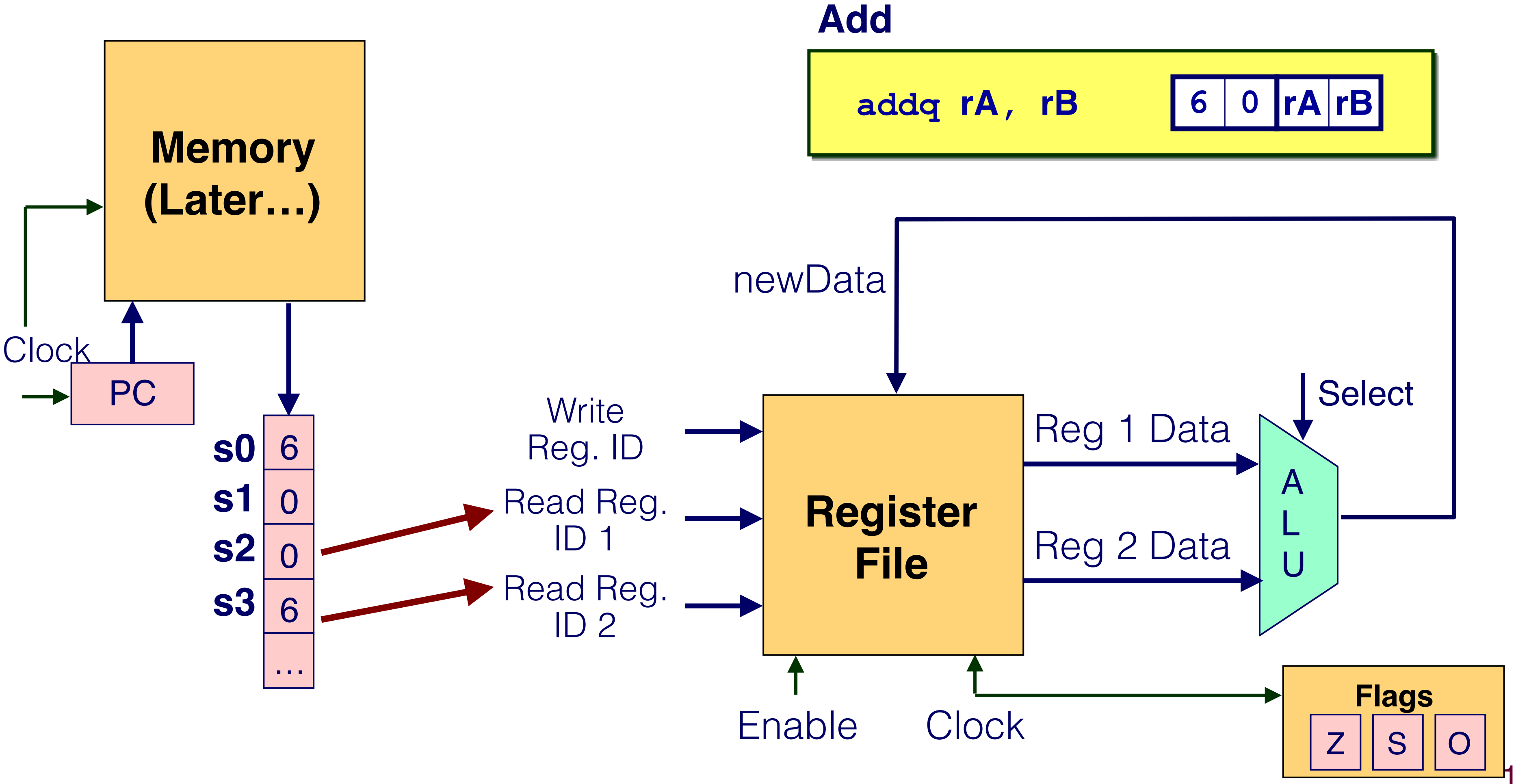
Executing an ADD instruction

- 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`



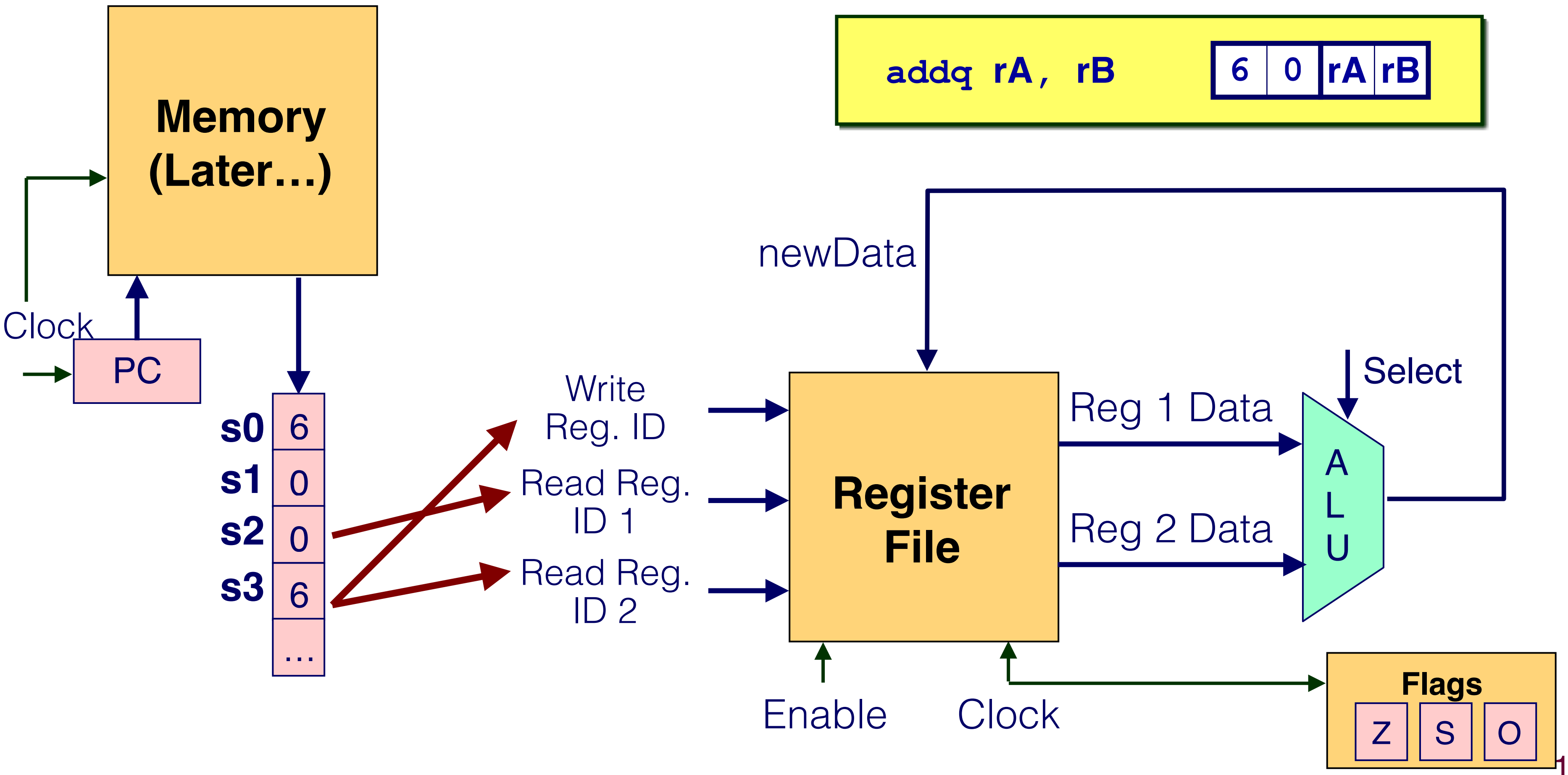
Executing an ADD instruction

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

Add

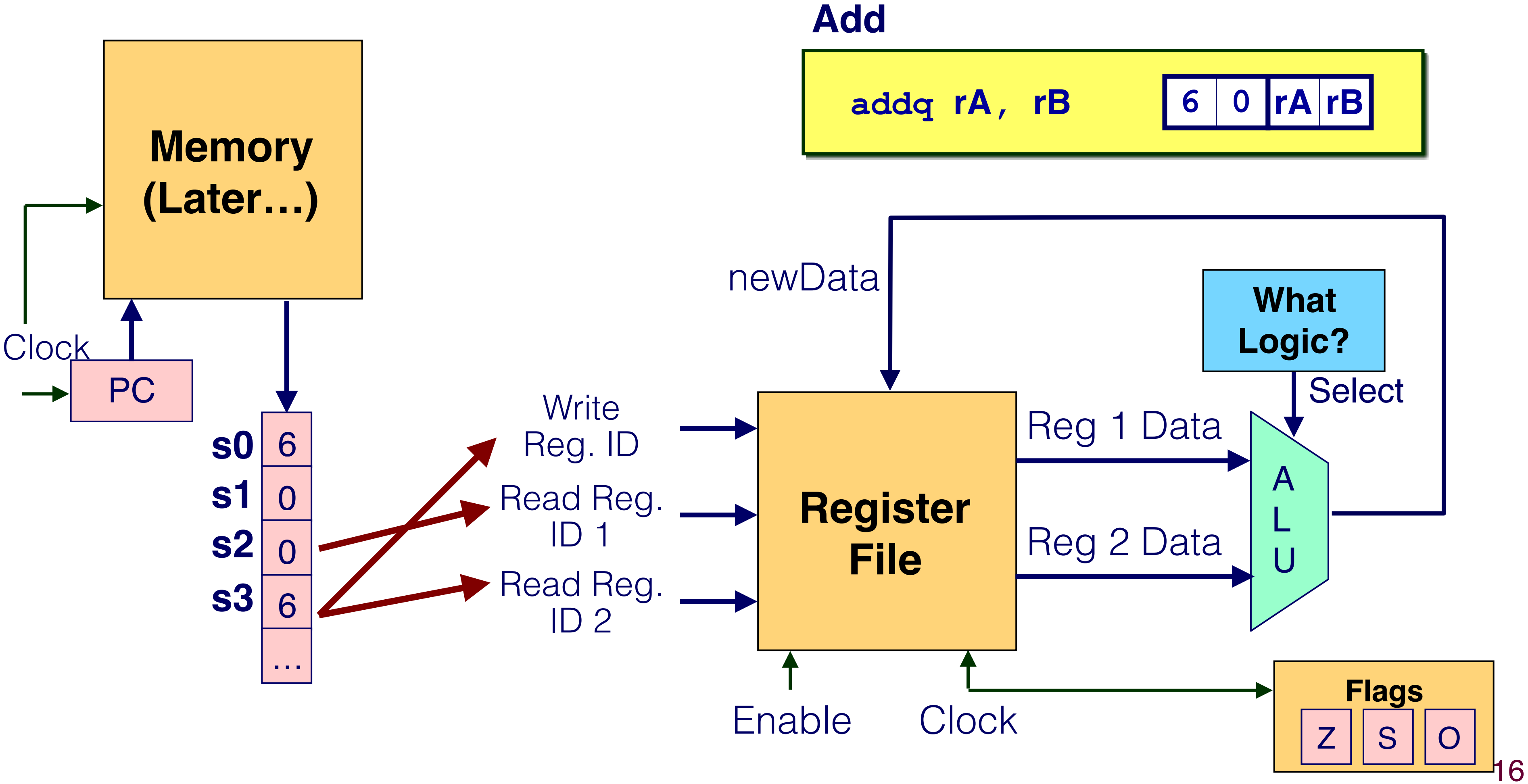
`addq rA, rB`

6	0	rA	rB



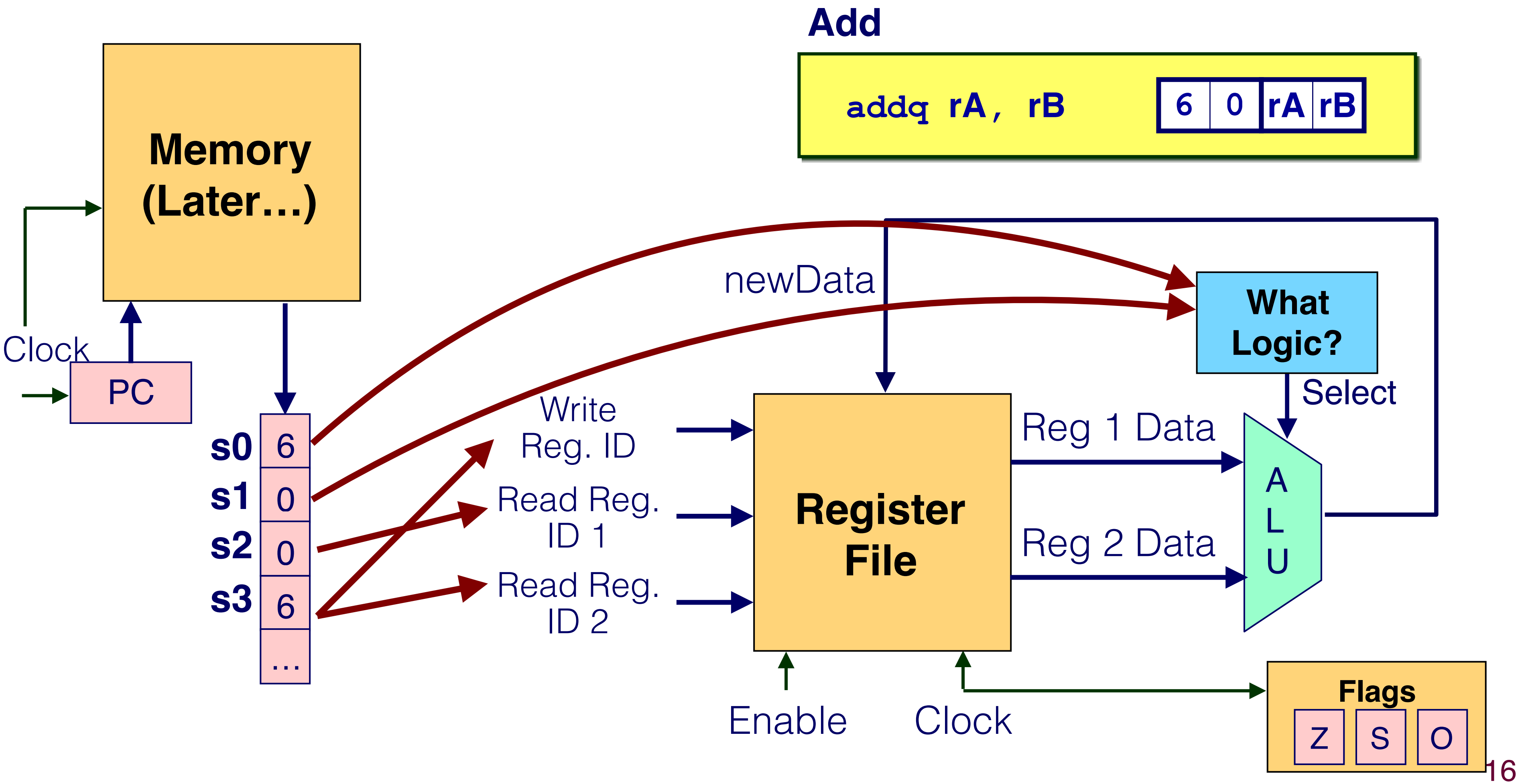
Executing an ADD instruction

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



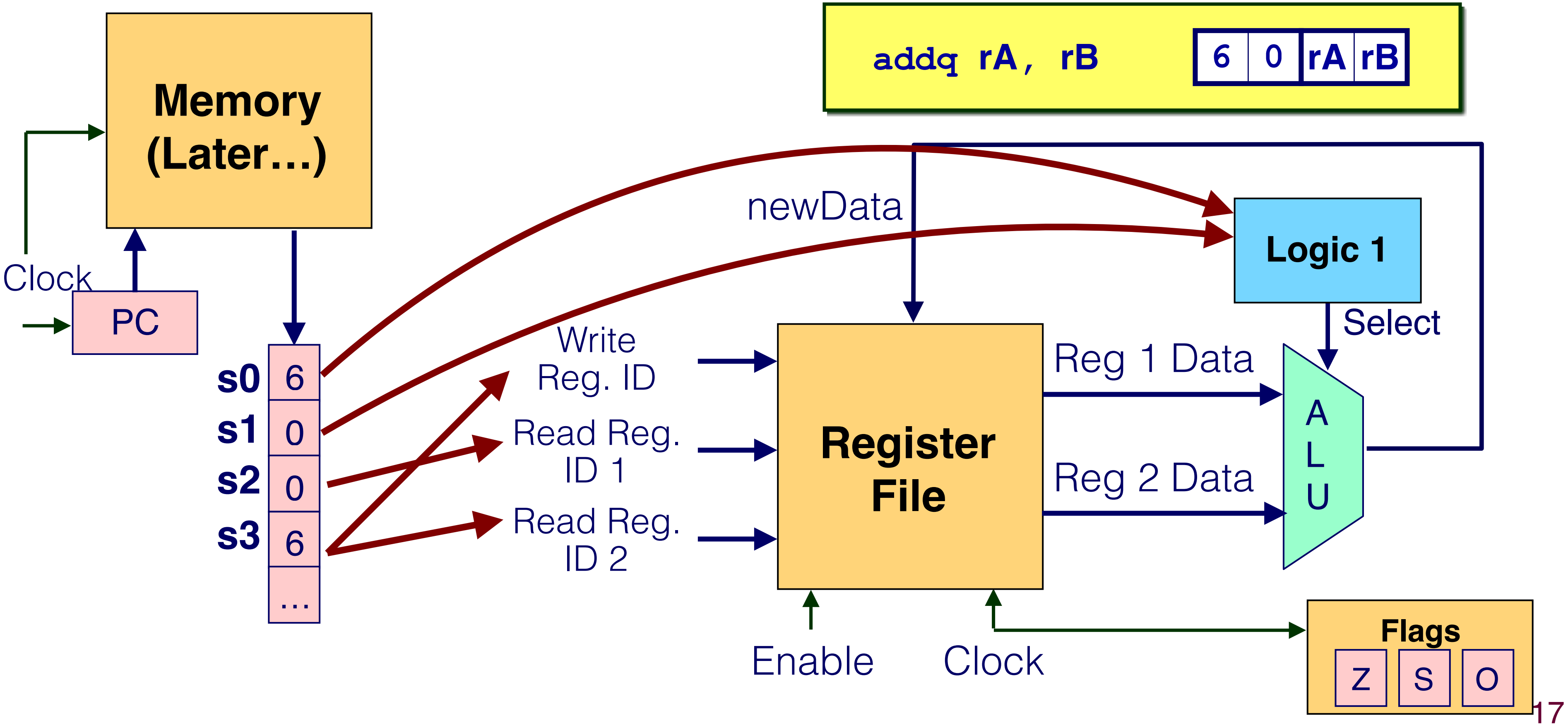
Executing an ADD instruction

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



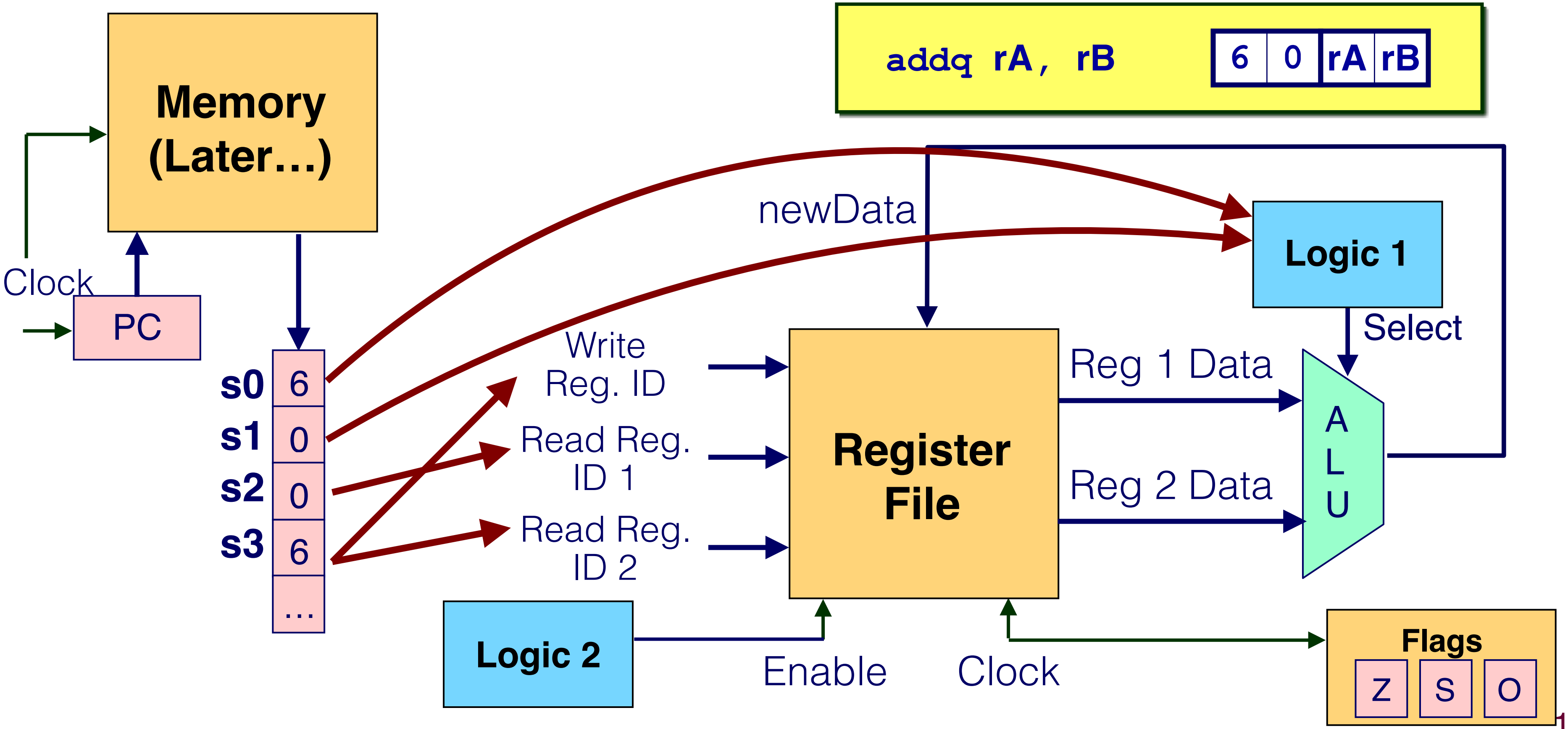
Executing an ADD instruction

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



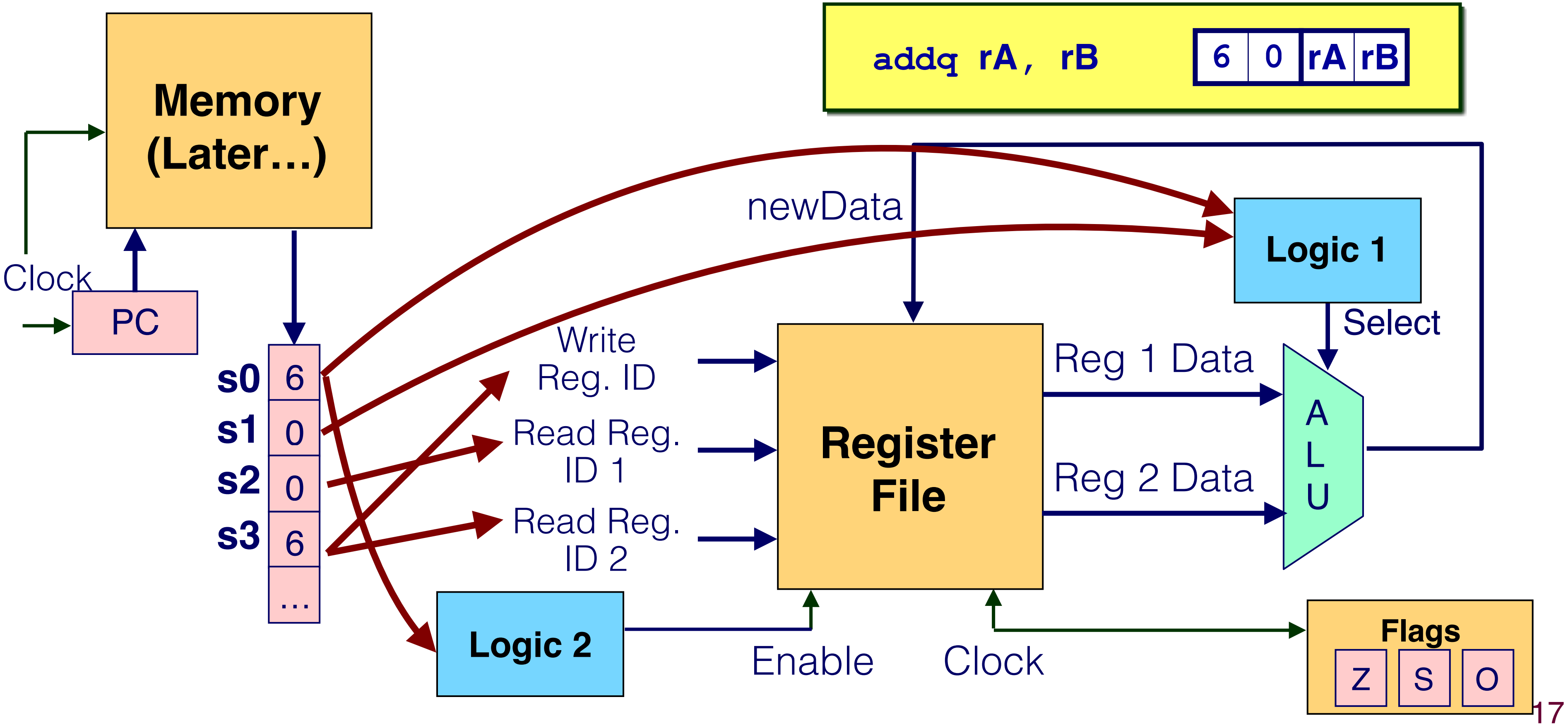
Executing an ADD instruction

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



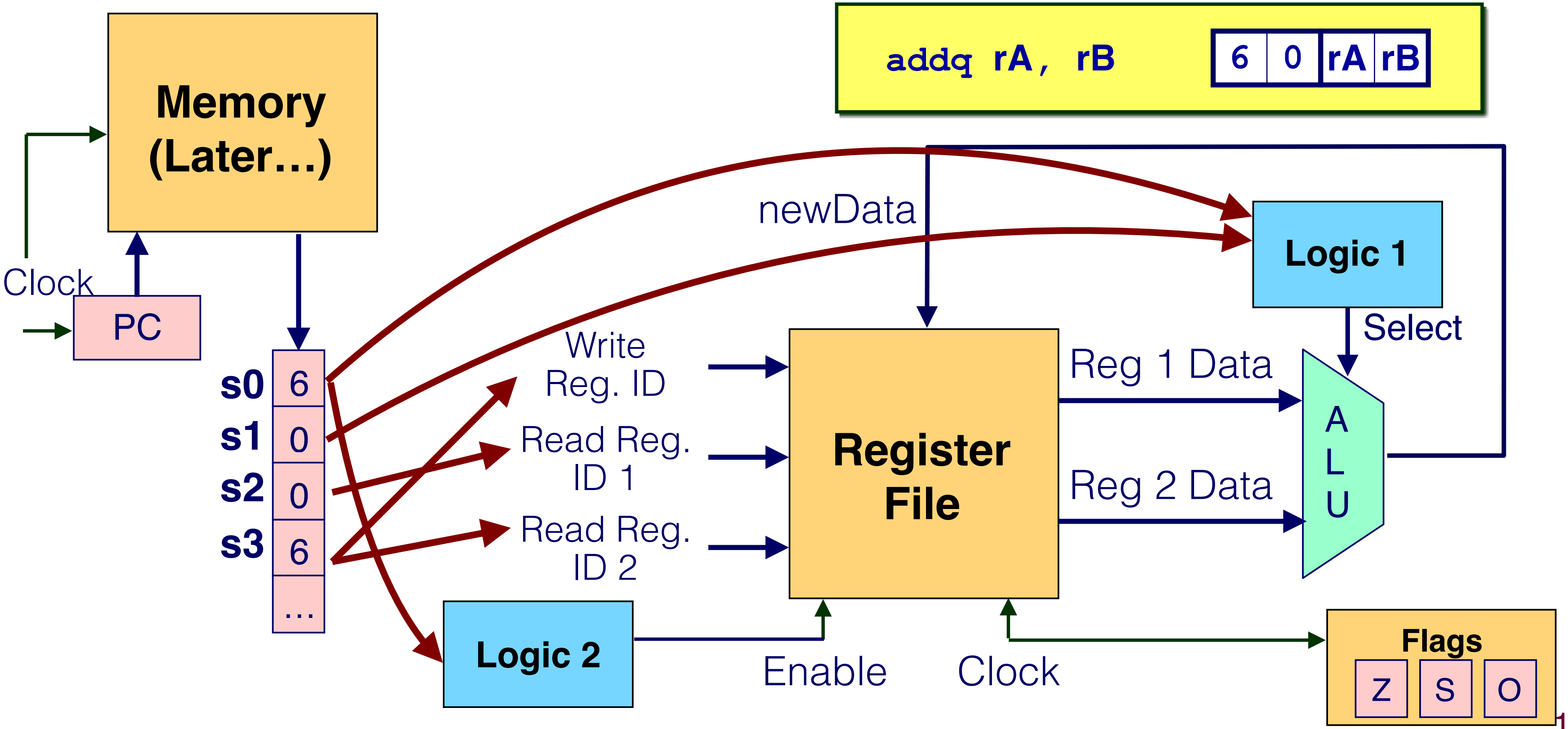
Executing an ADD instruction

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



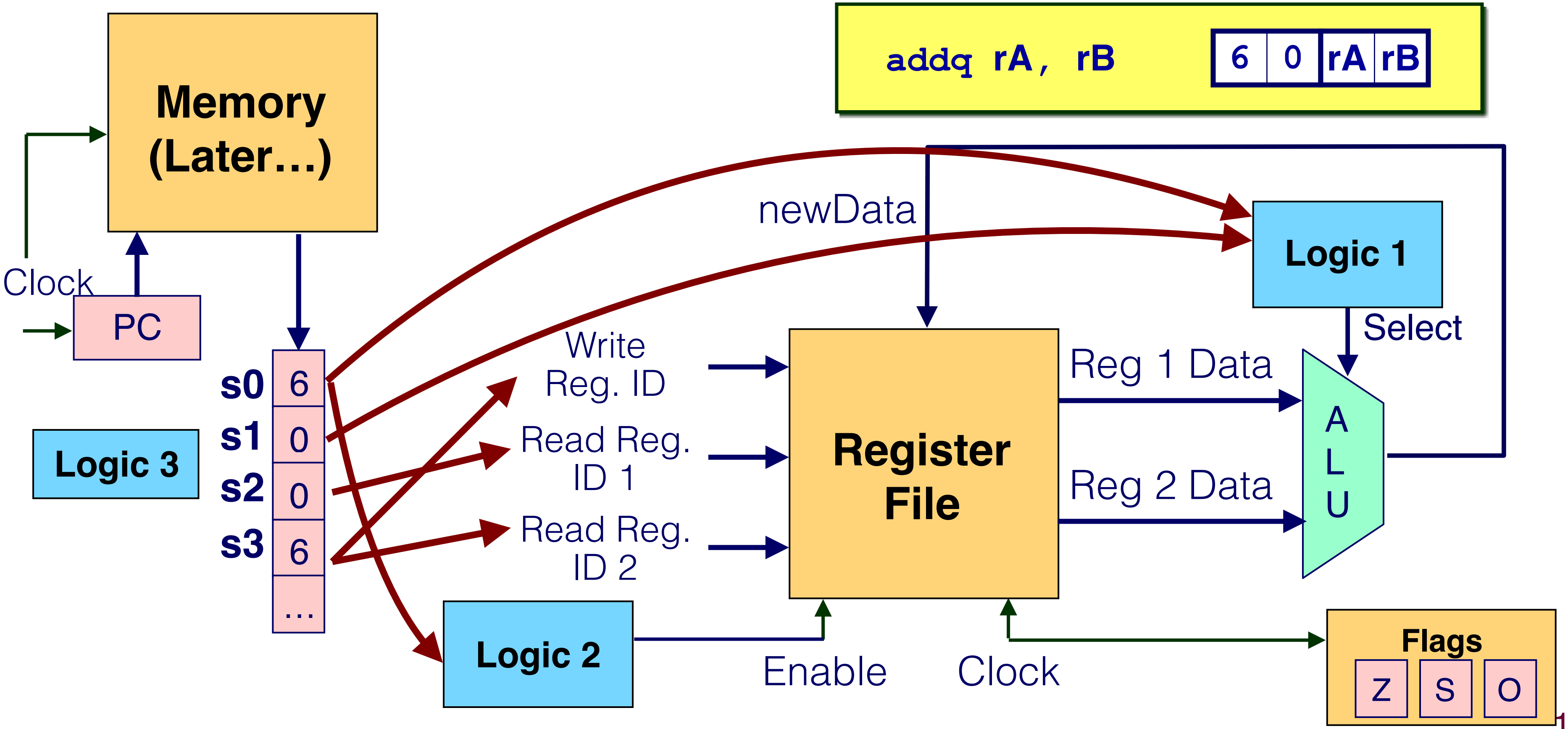
Executing an ADD instruction

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



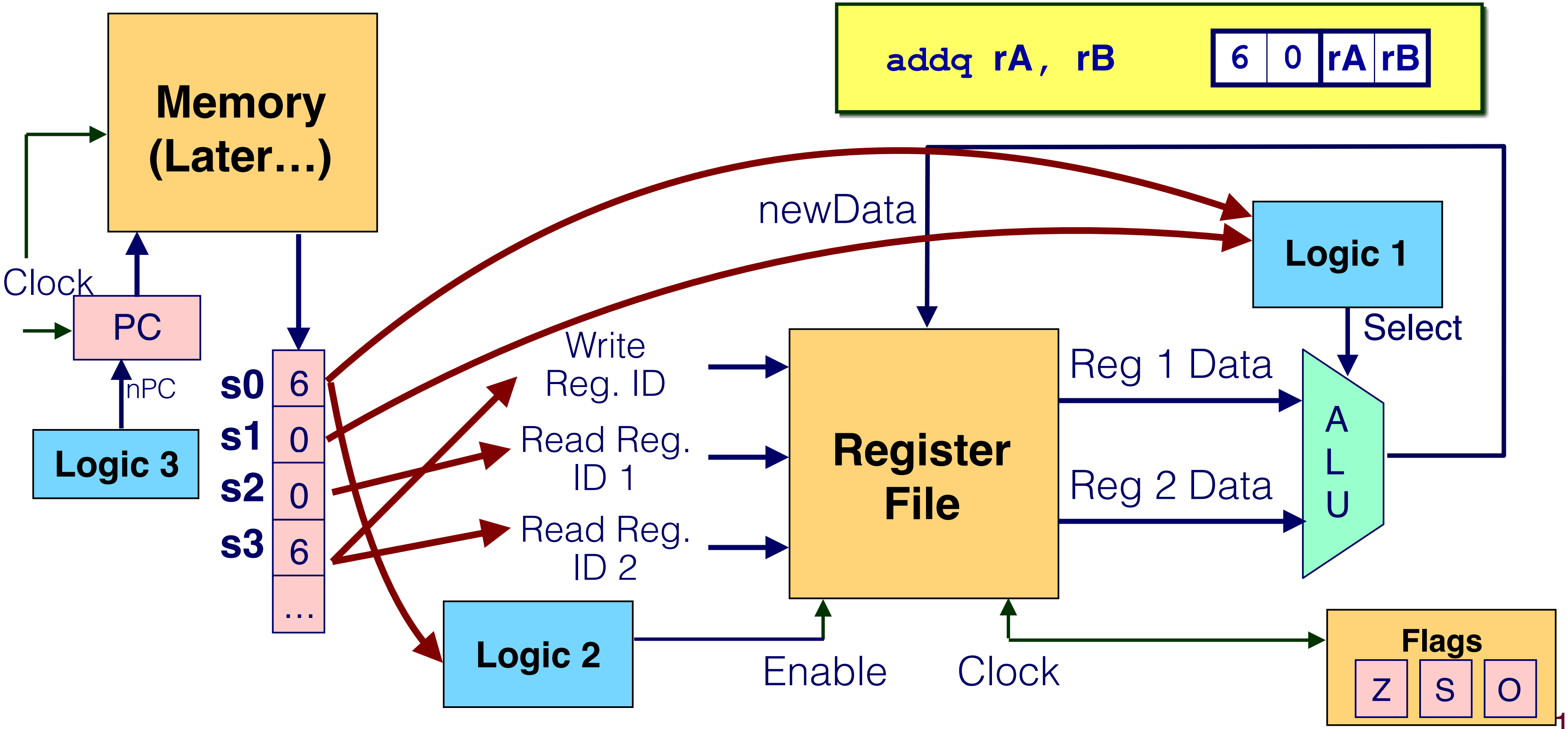
Executing an ADD instruction

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



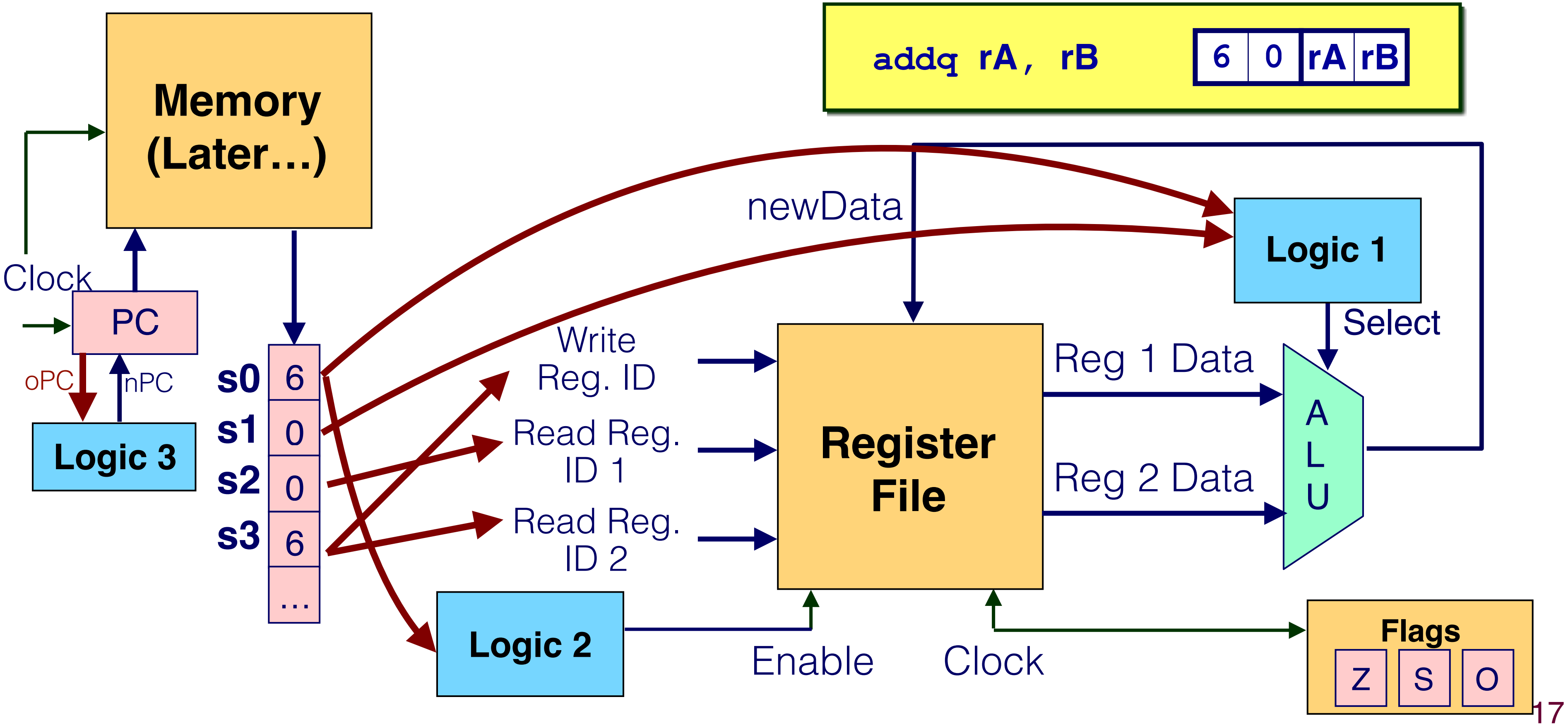
Executing an ADD instruction

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



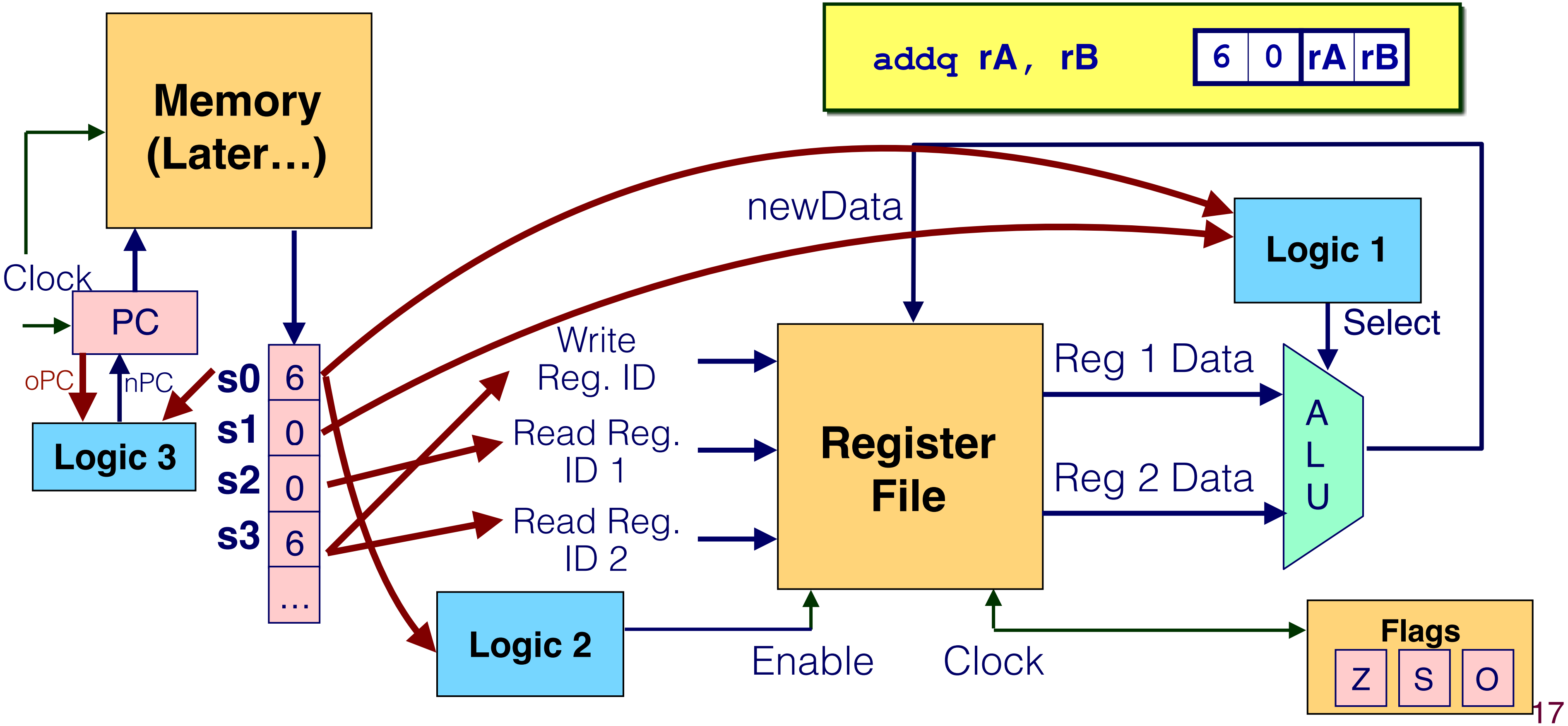
Executing an ADD instruction

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



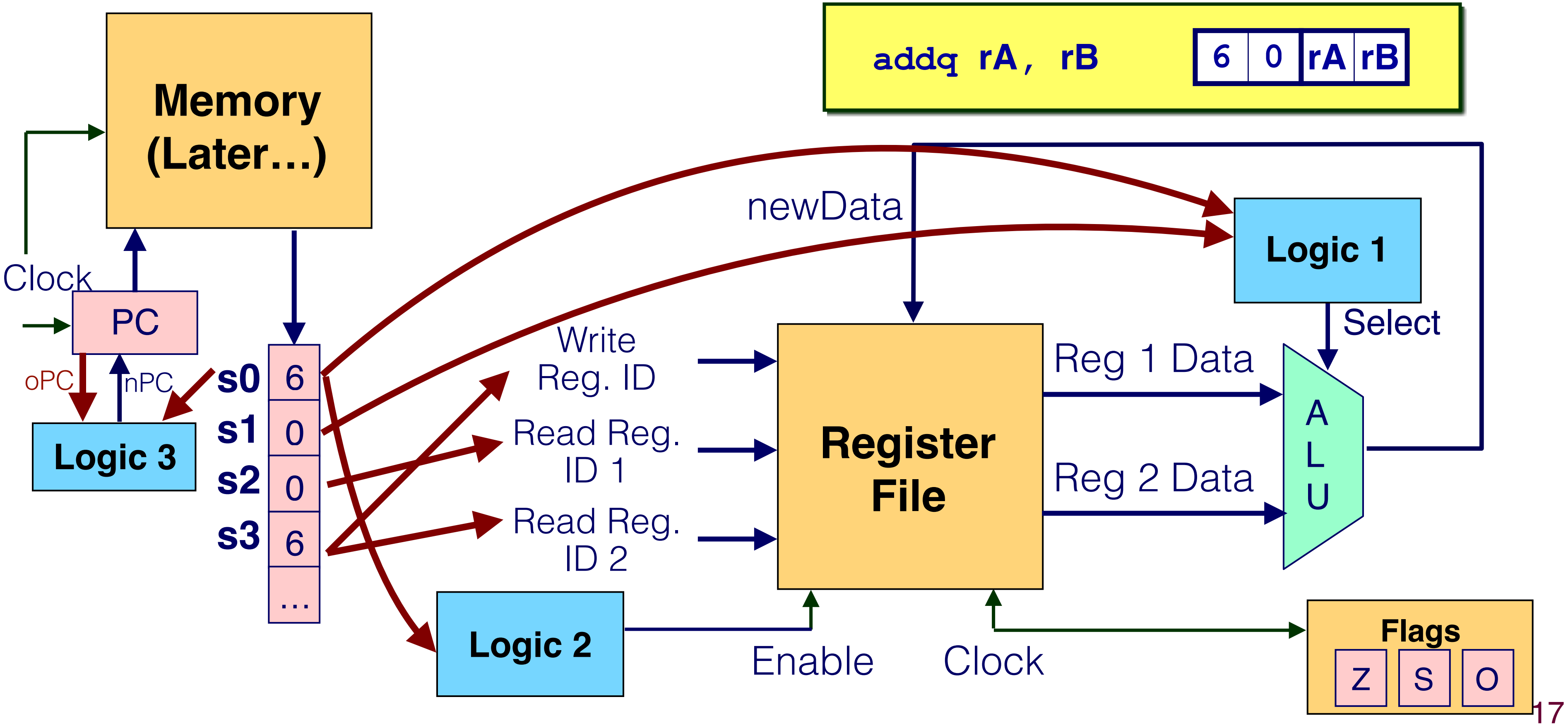
Executing an ADD instruction

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



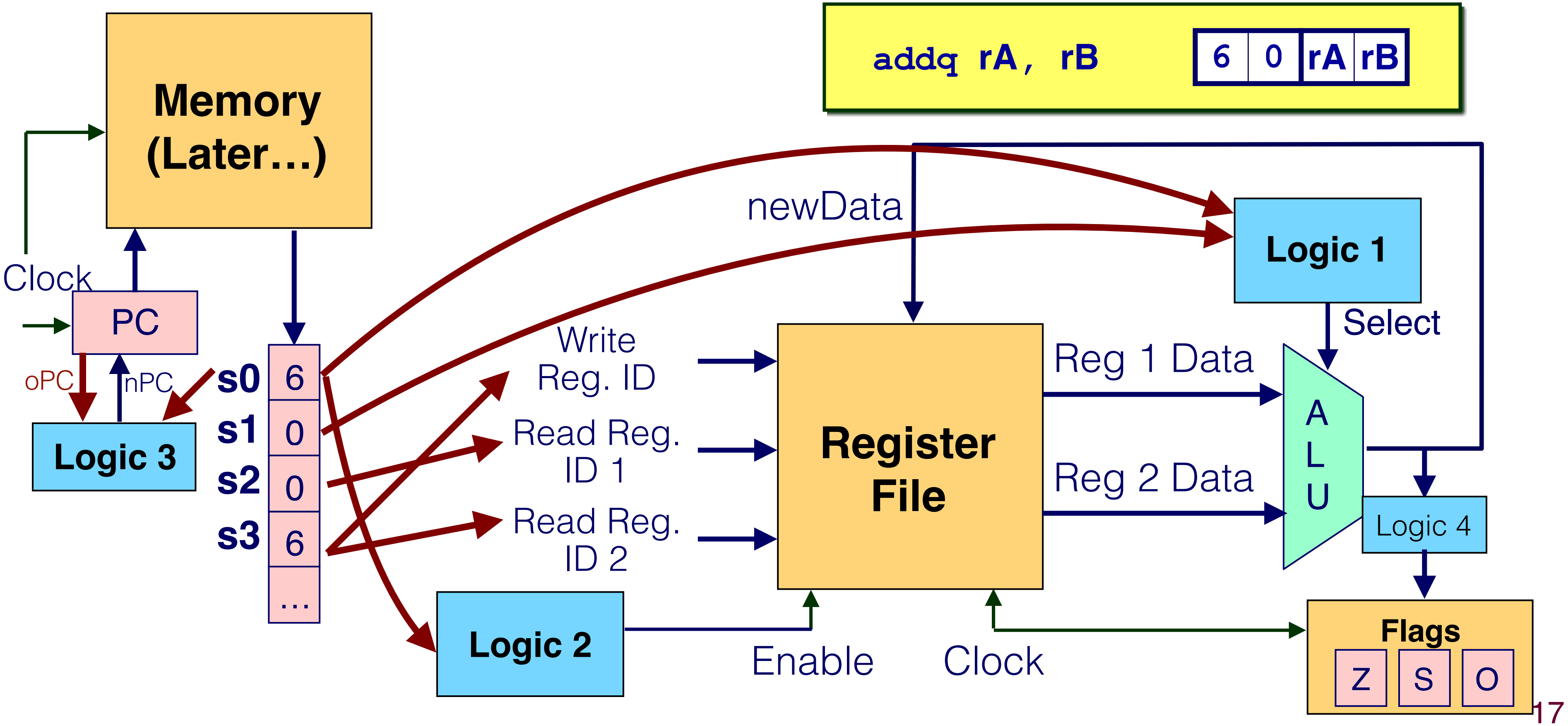
Executing an ADD instruction

- 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

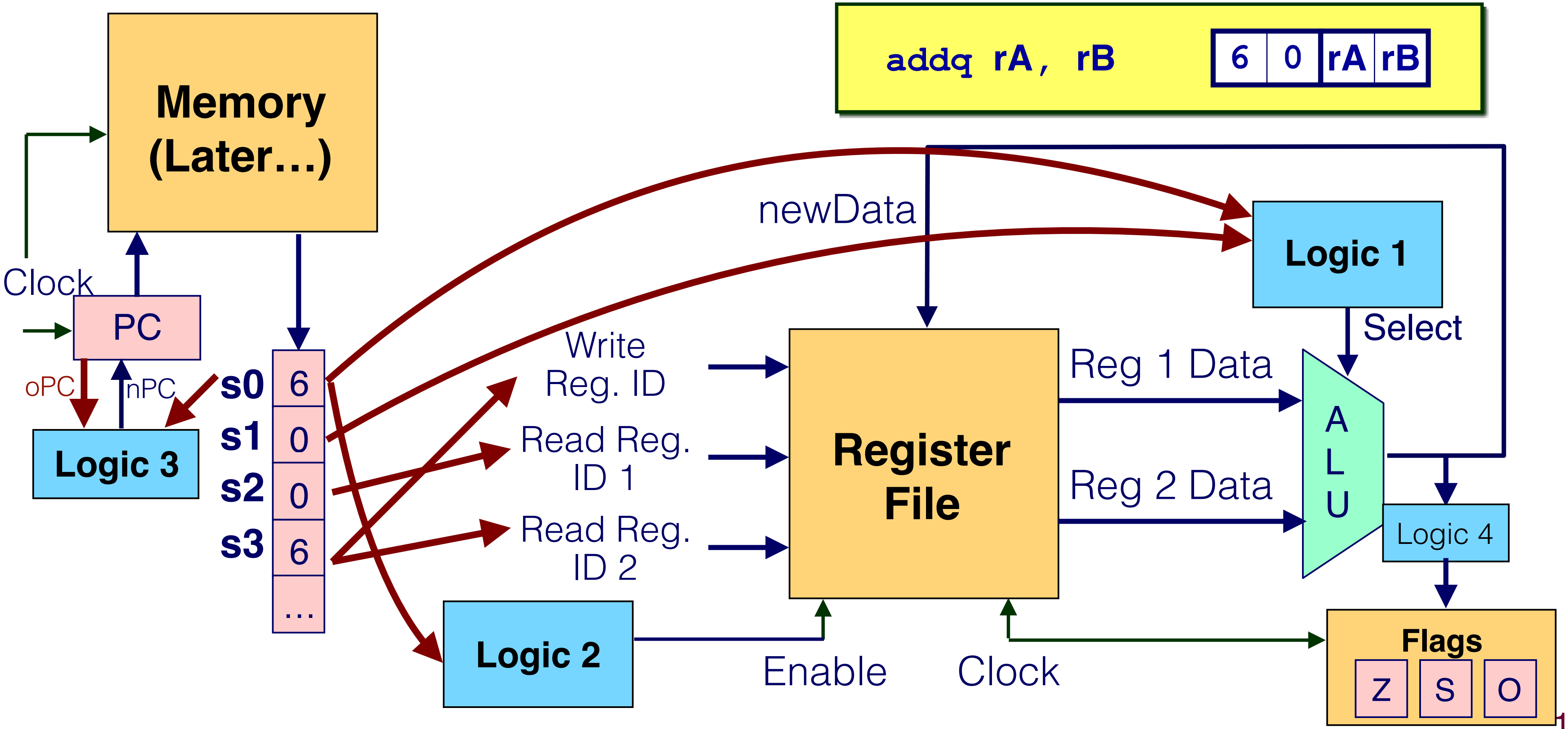
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?



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?

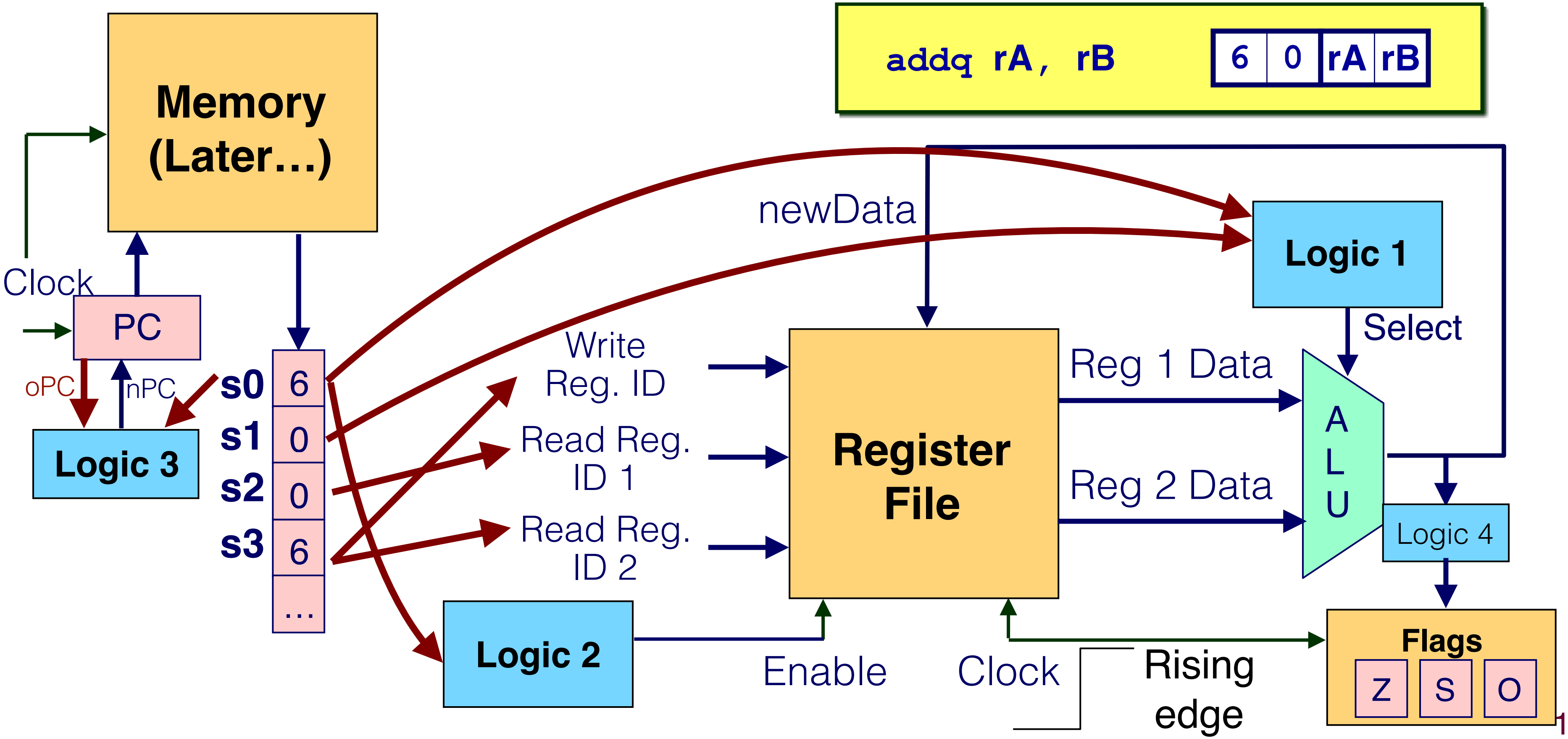
How do these logics get implemented?



Executing an ADD instruction

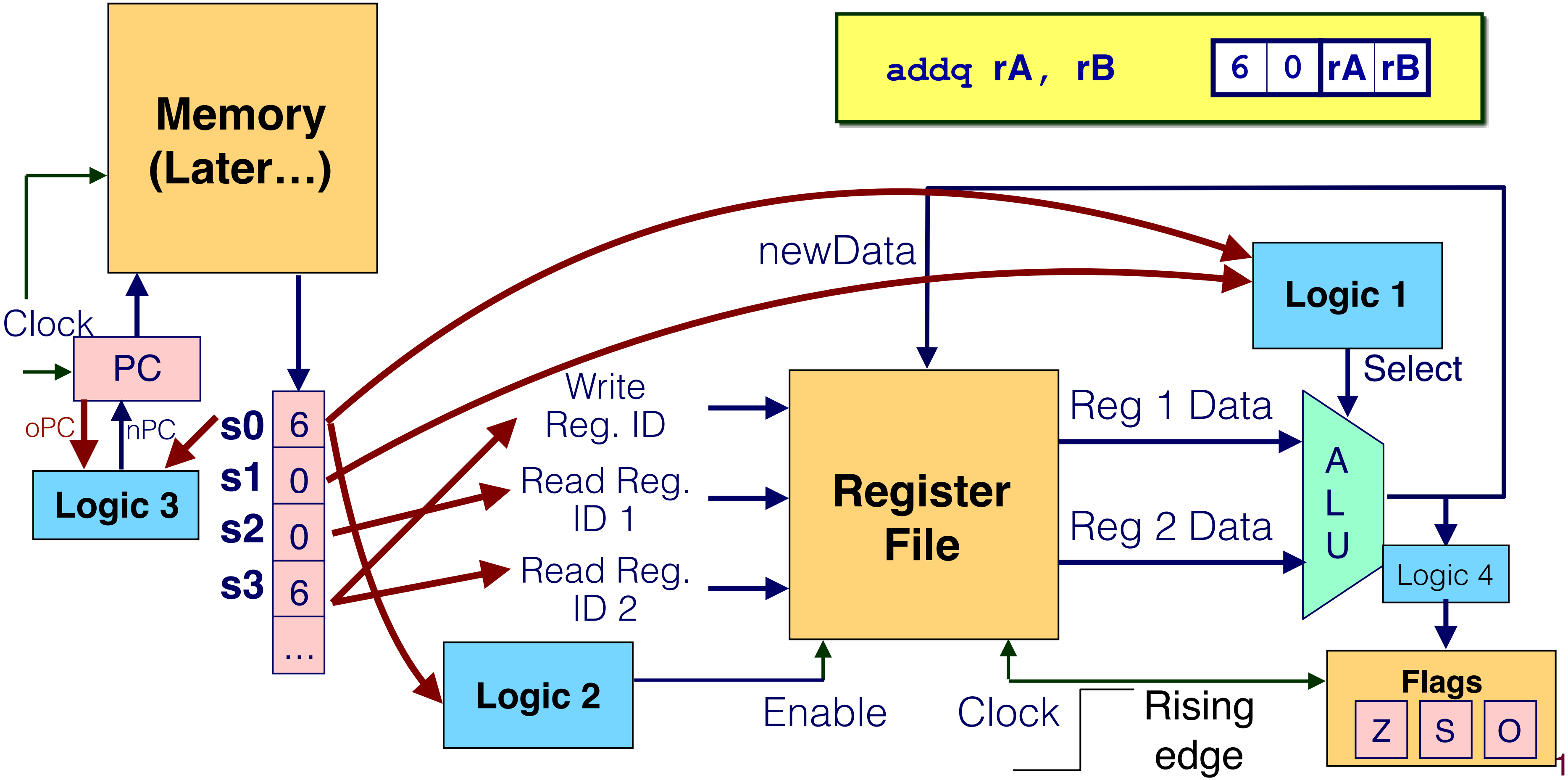
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

How do these logics get implemented?



Executing an ADD instruction

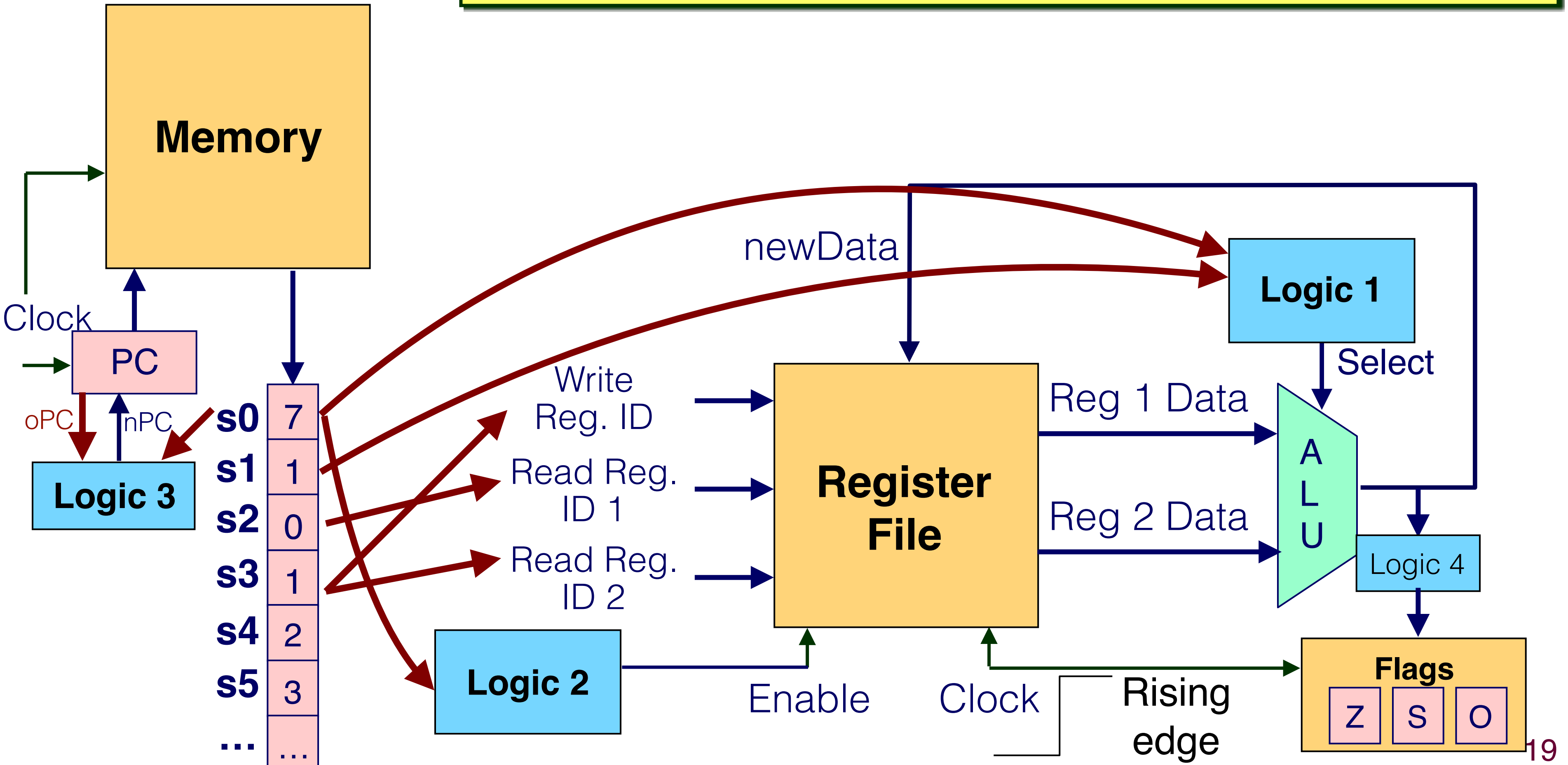
- When the rising edge of the clock arrives, the RF/PC/Flags will be written.
- So the following has to be ready: newData, nPC, which means Logic1, Logic2, Logic3, and Logic4 has to finish.



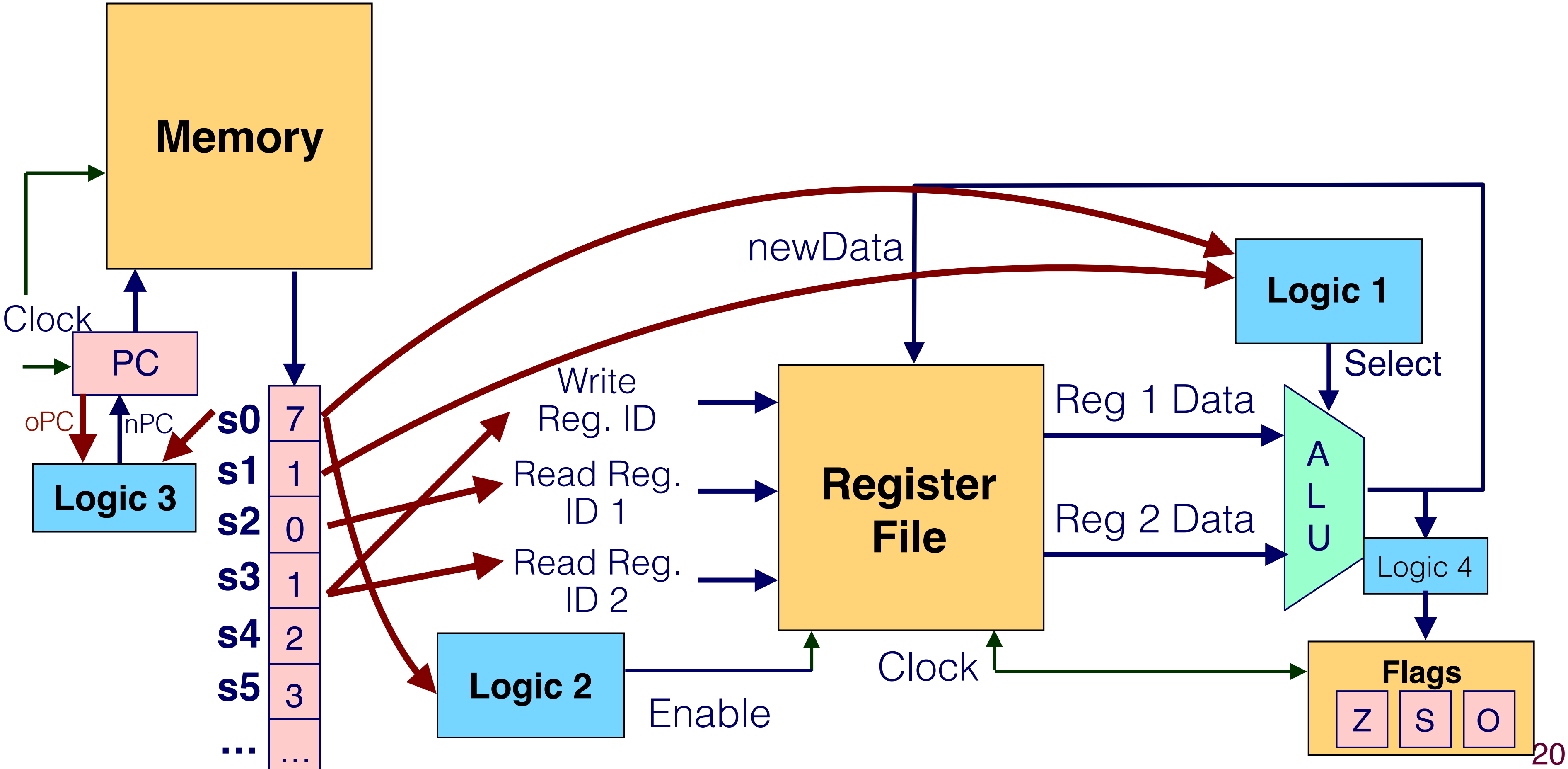
Executing a JLE instruction

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

`jle` Dest	**7**	**1**	Dest

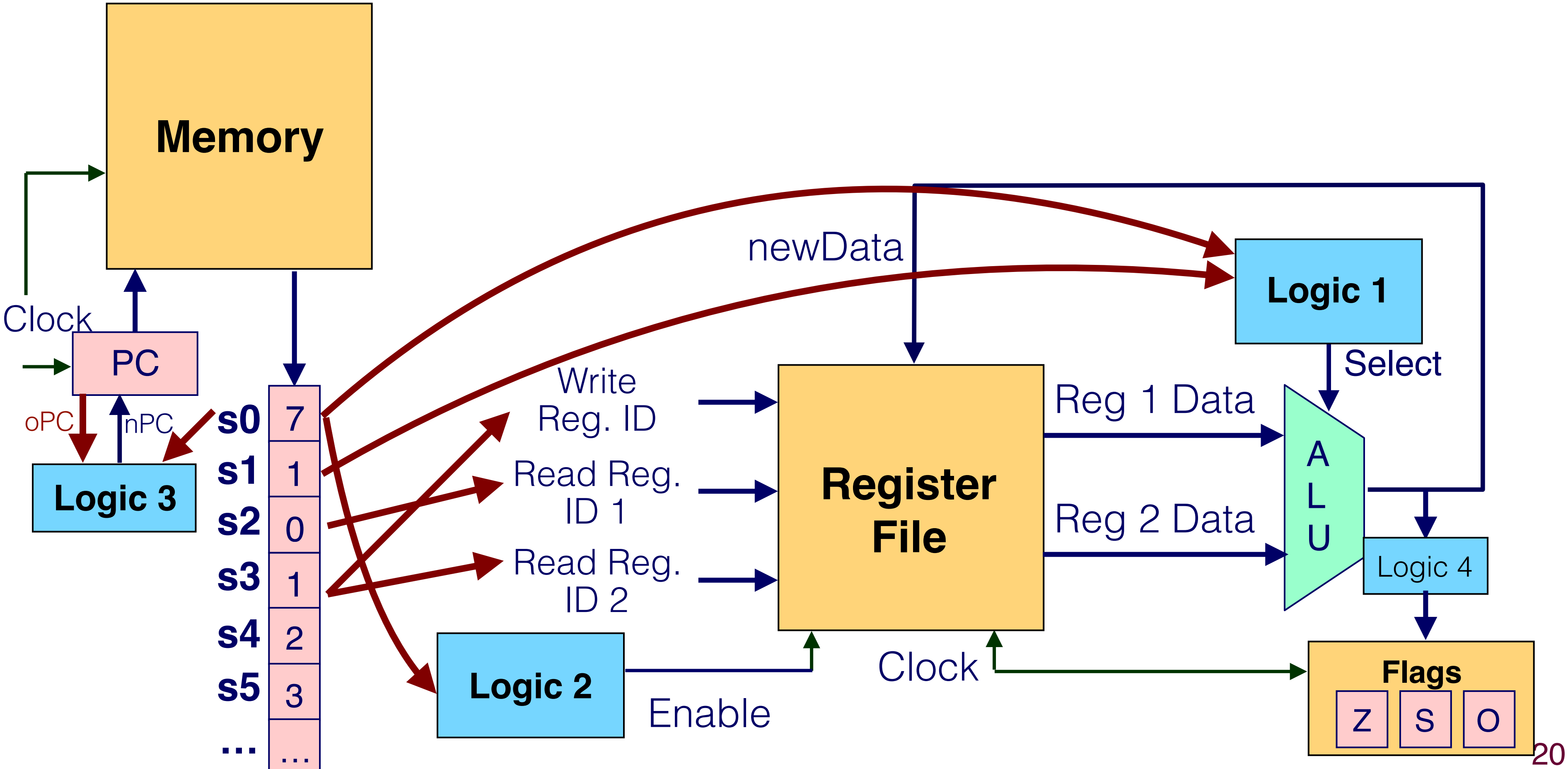


Executing a JLE instruction



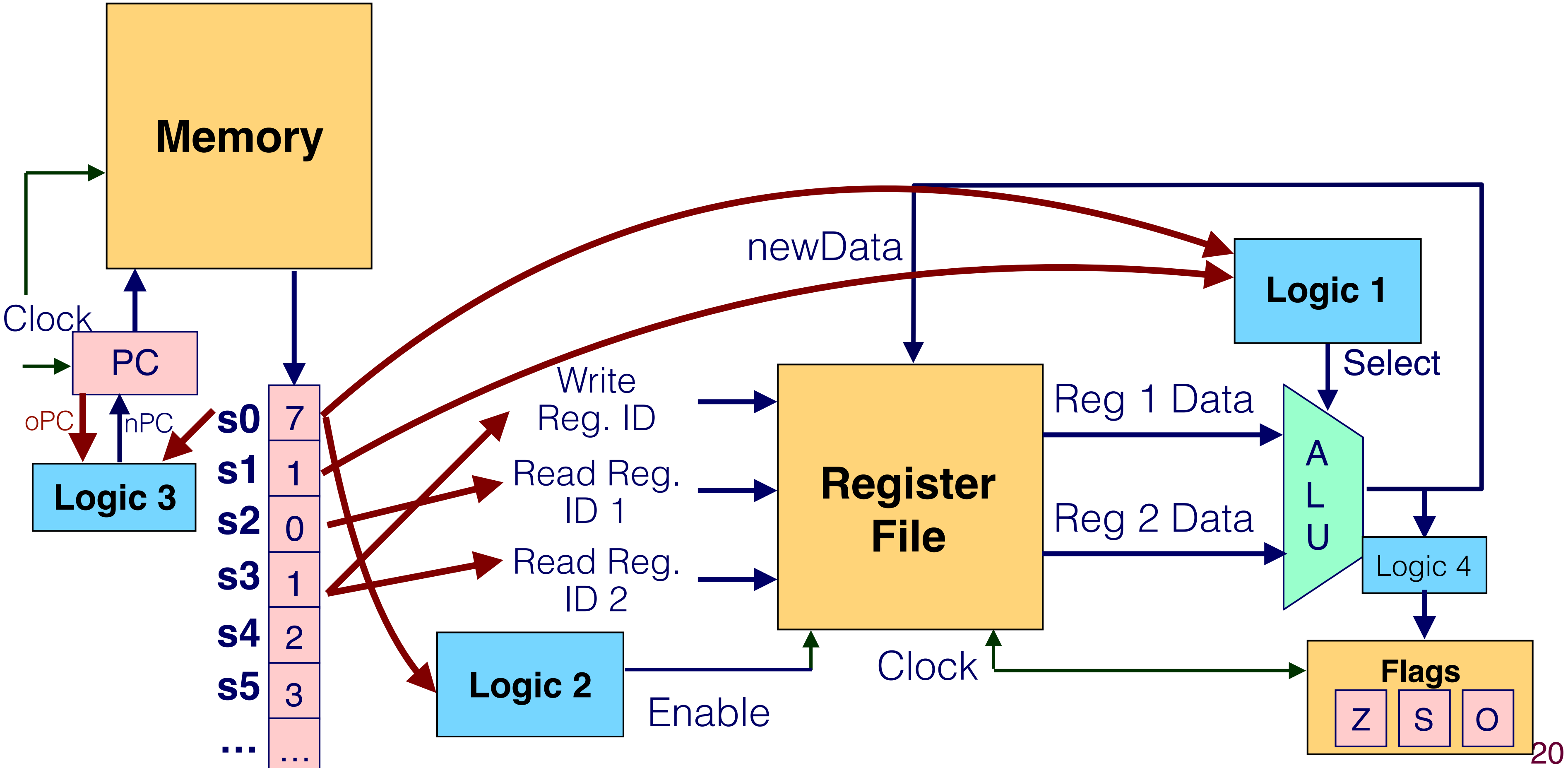
Executing a JLE instruction

- 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;

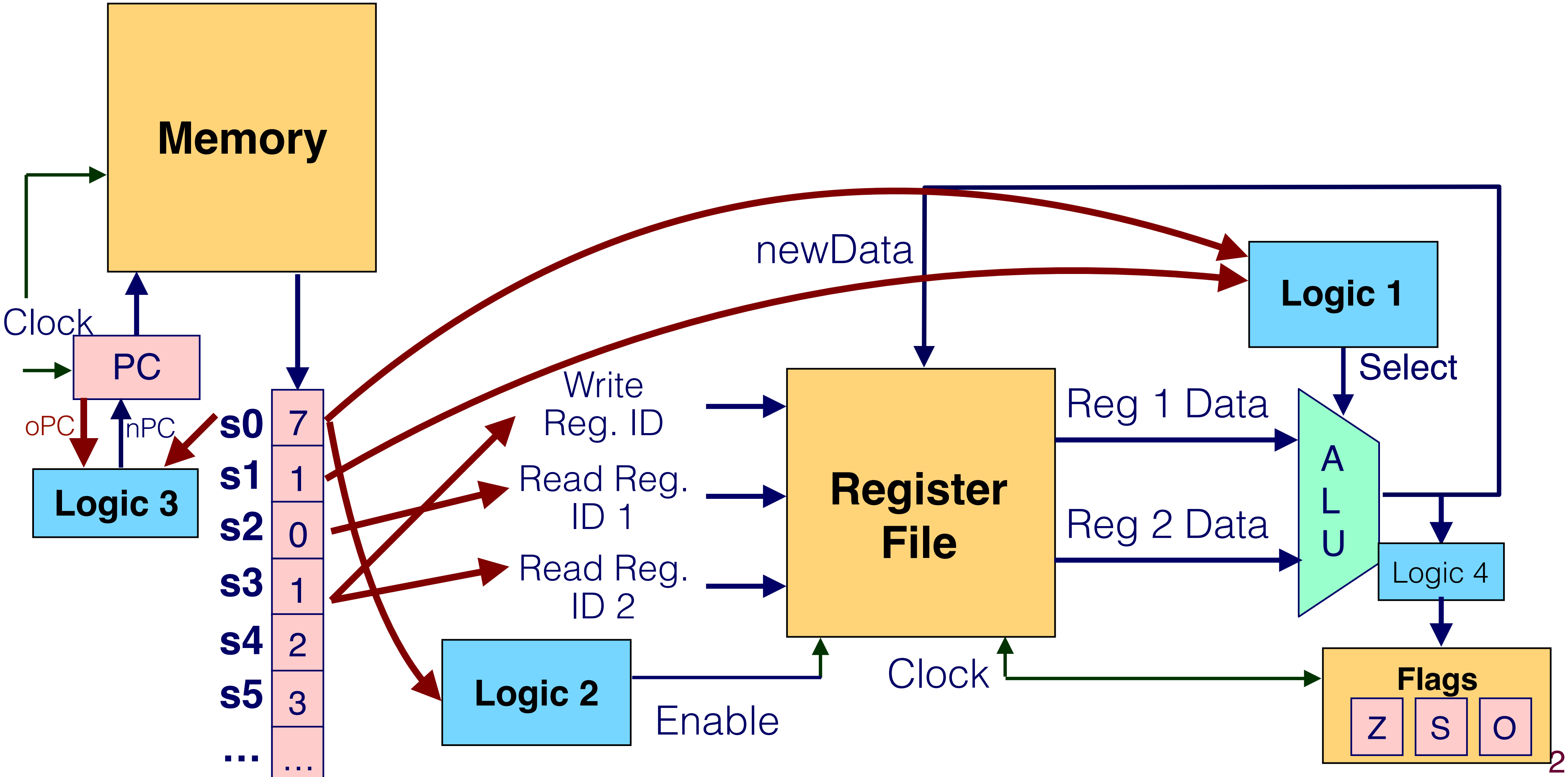


`jle Dest`

7	1	Dest

Executing a JLE instruction

- Logic 3??



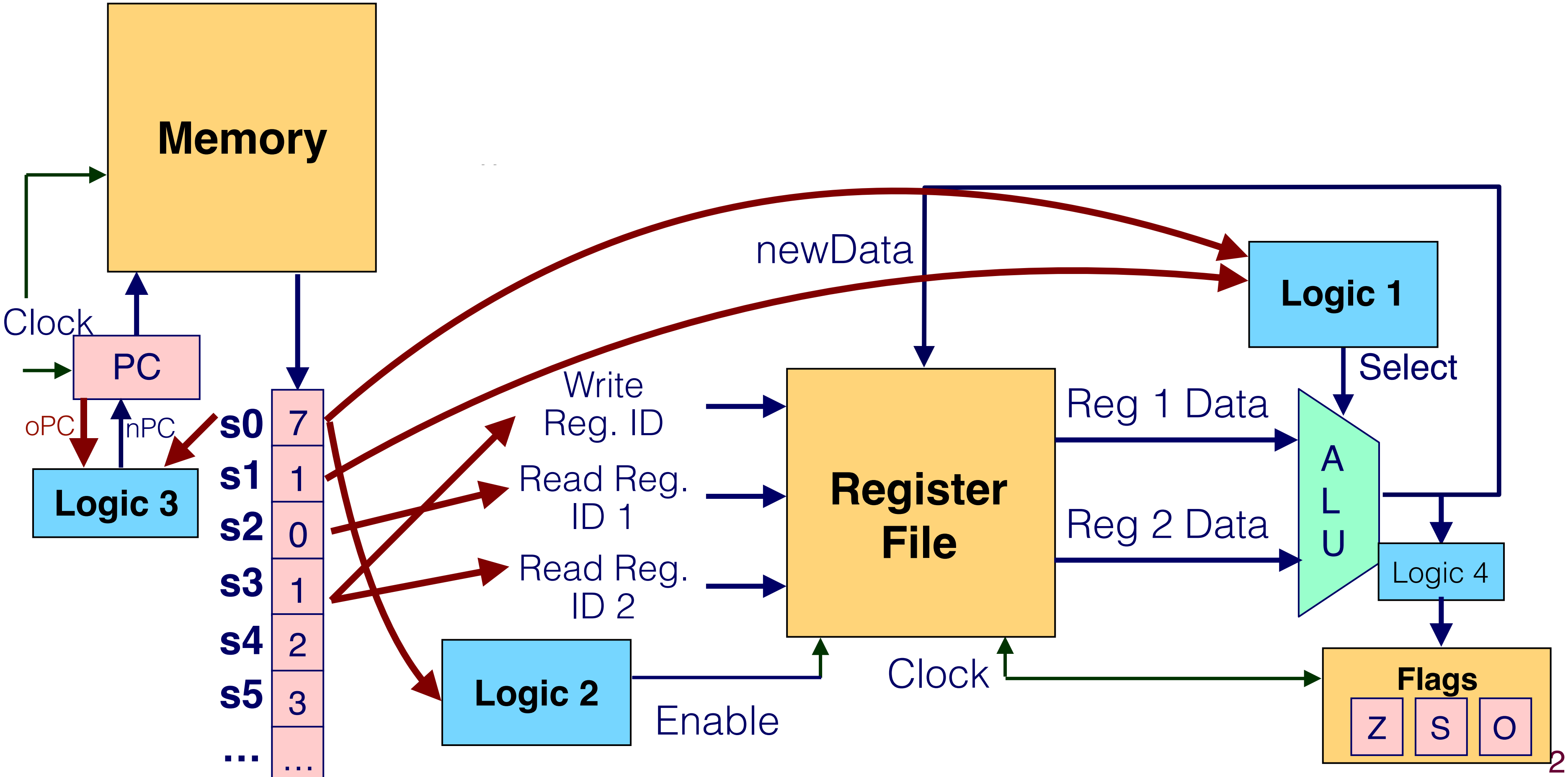
jle Dest

7	1	Dest

Executing a JLE instruction

- Logic 3??

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

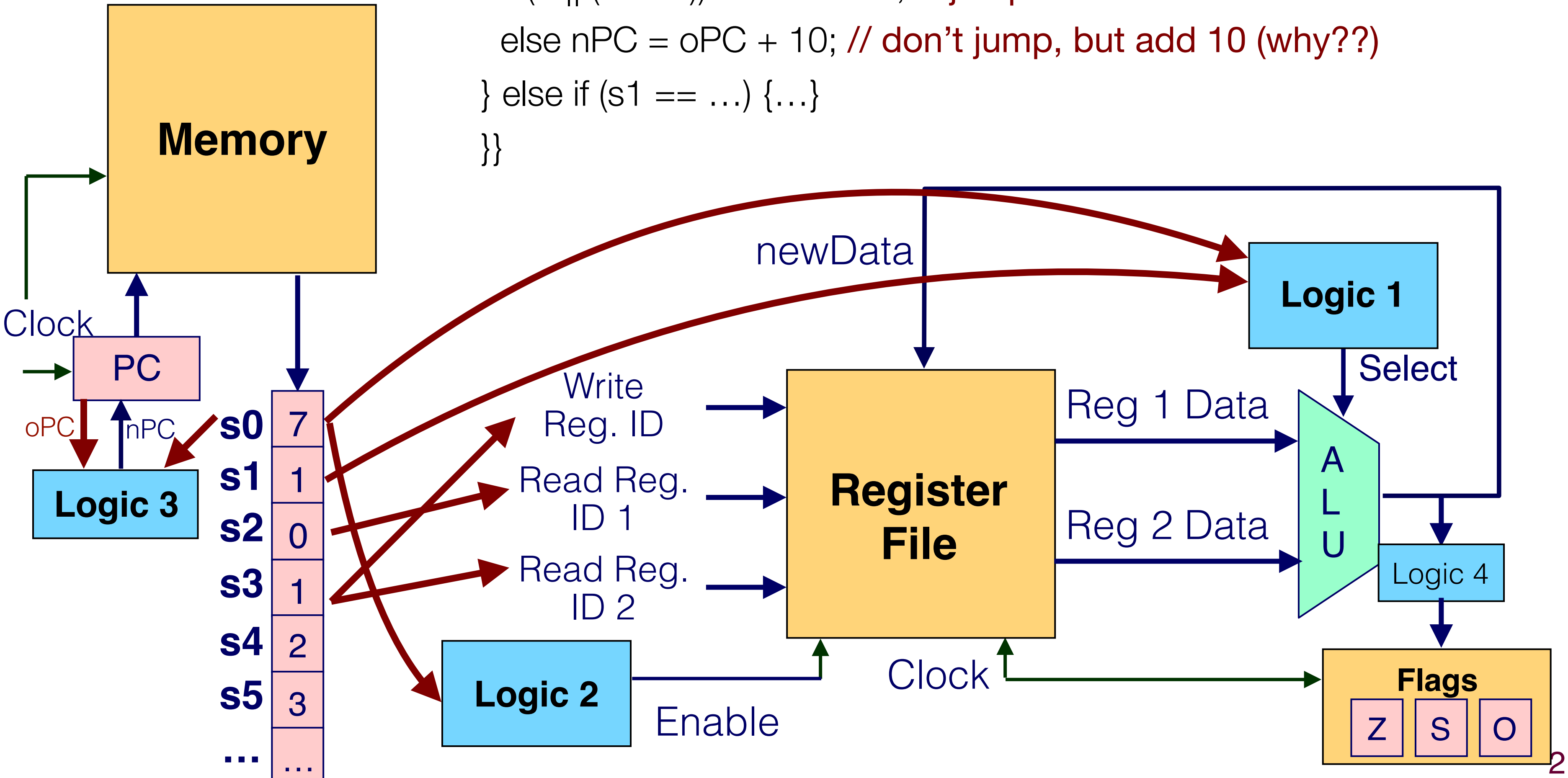


jle Dest	7	1	Dest

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 == …) {…}
  }}
```

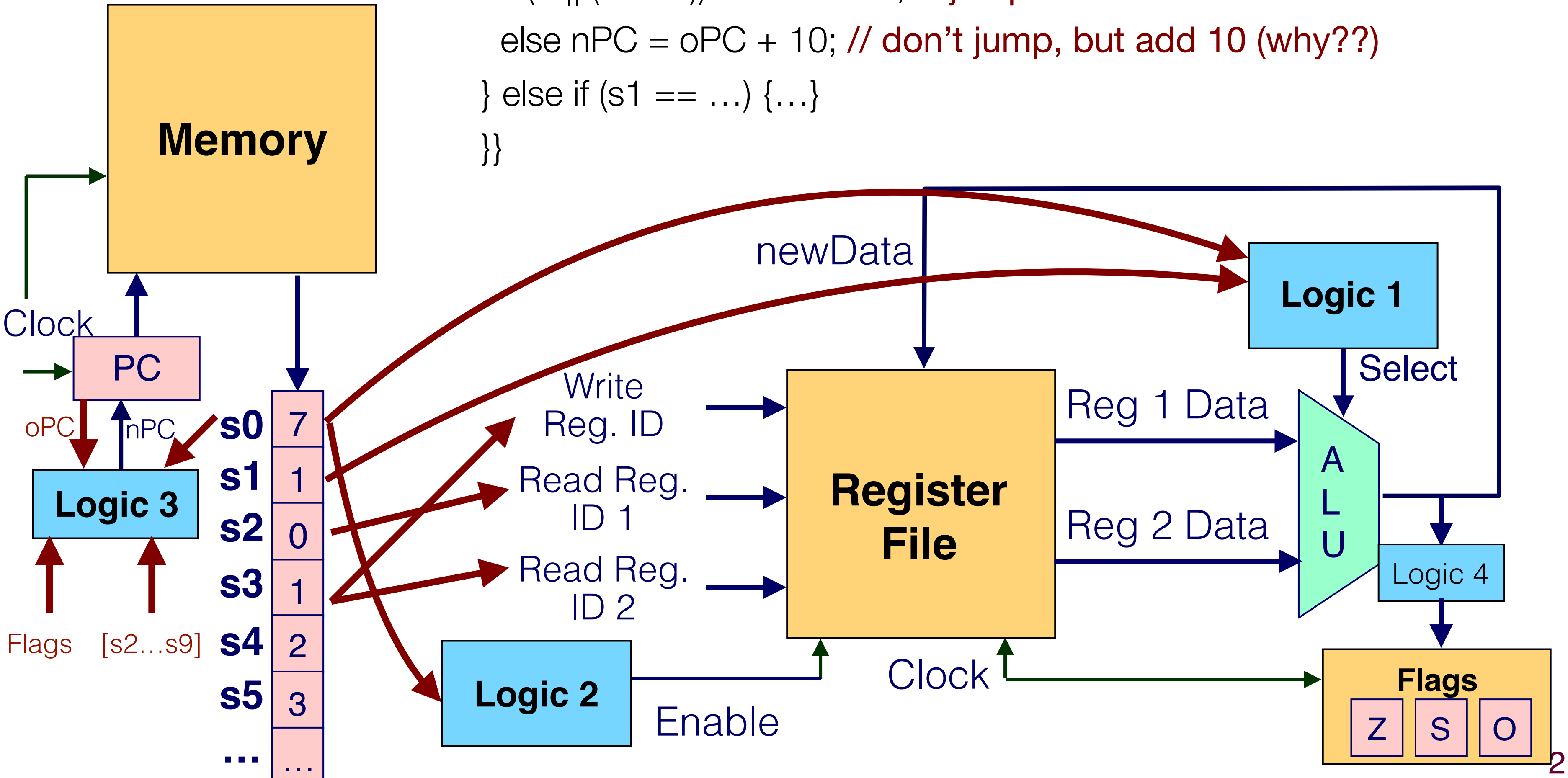


jle Dest	7	1	Dest

Executing a JLE instruction

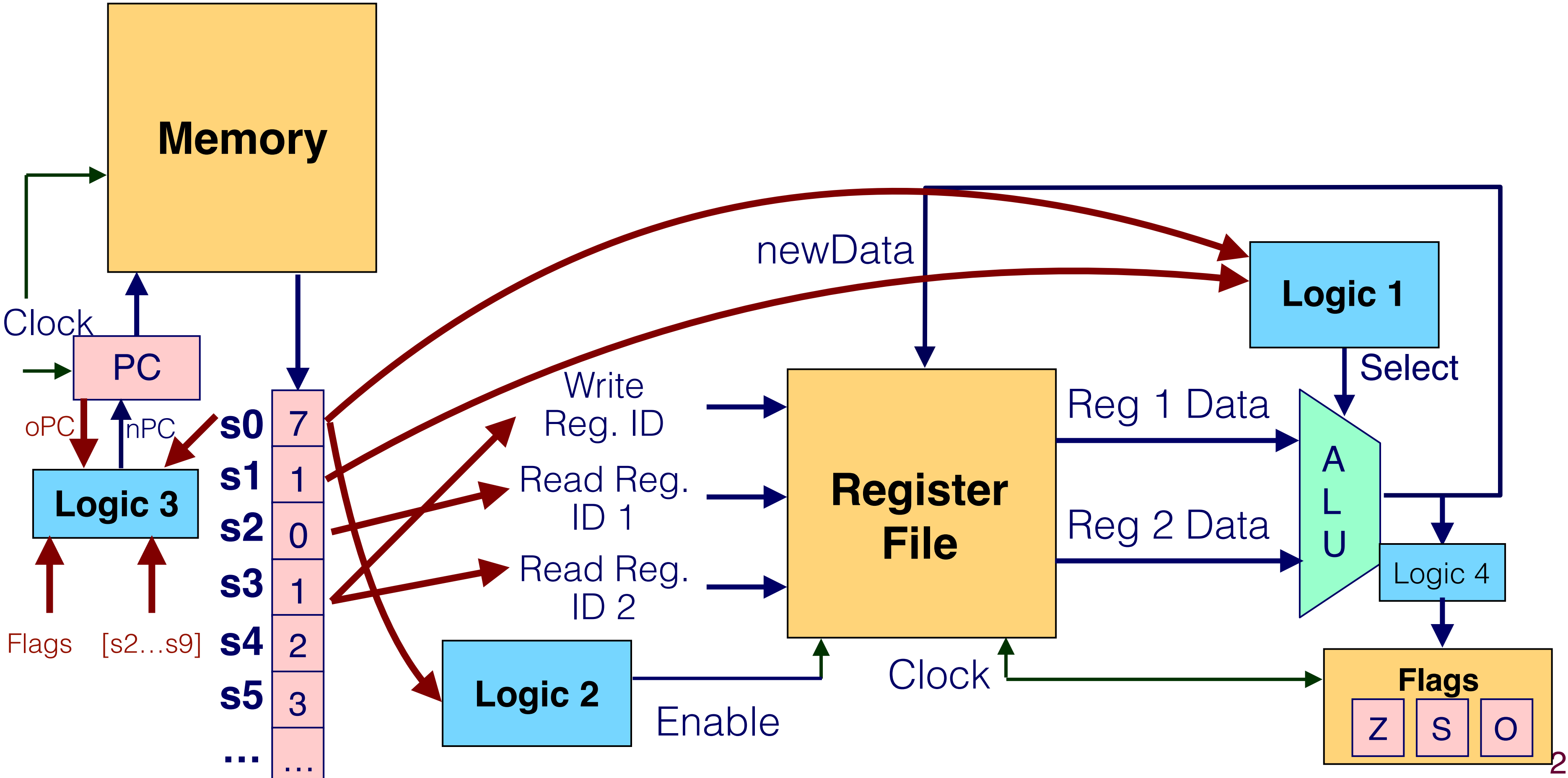
- Logic 3??

```
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 == ...) {...}
  }}
```



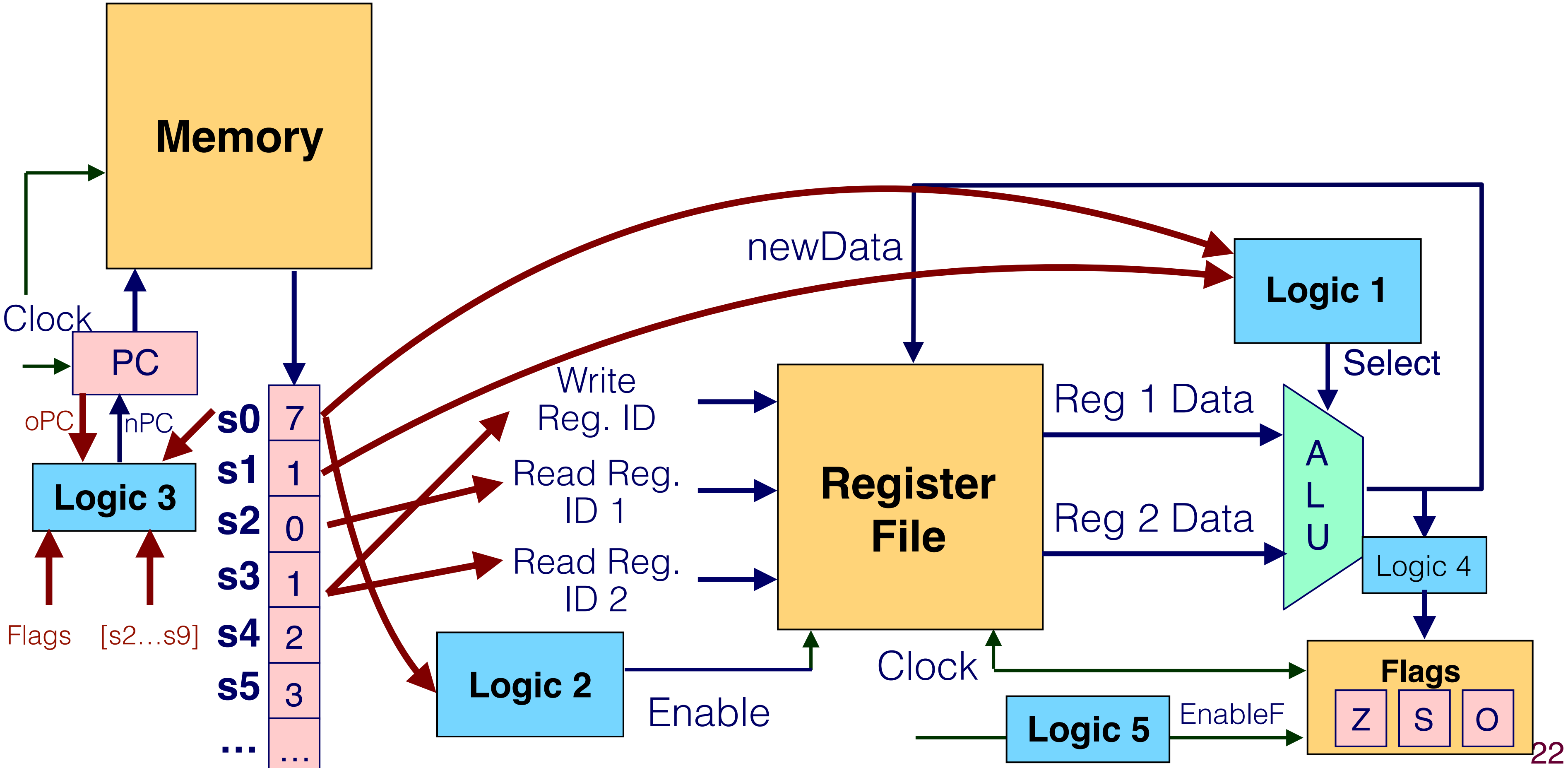
Executing a JLE instruction

- Logic 4? Does JLE write flags?



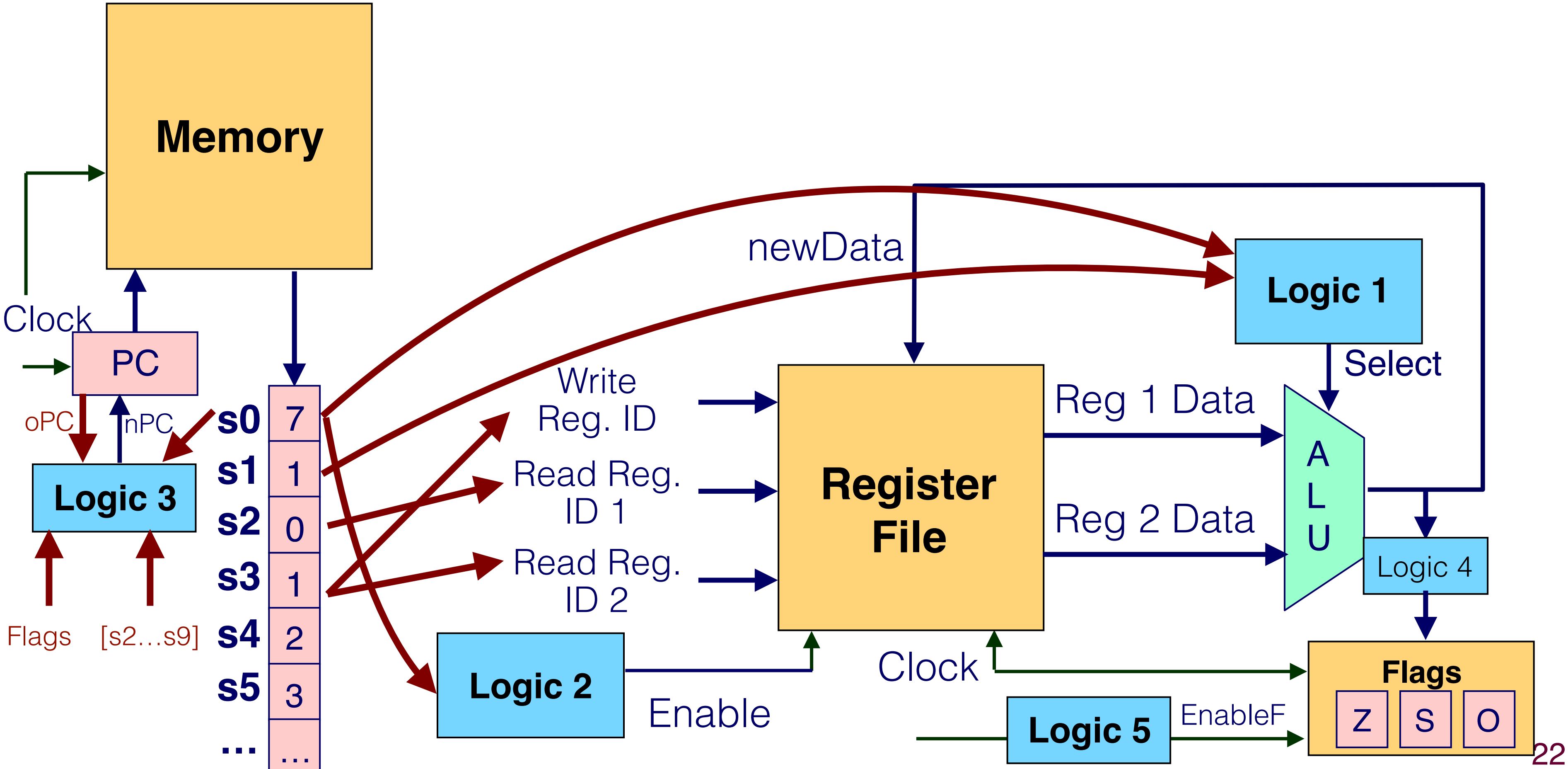
Executing a JLE instruction

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

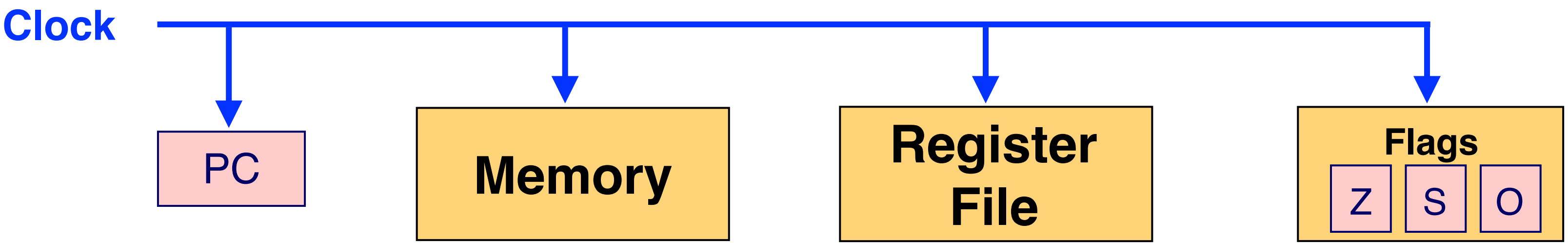


Executing a JLE instruction

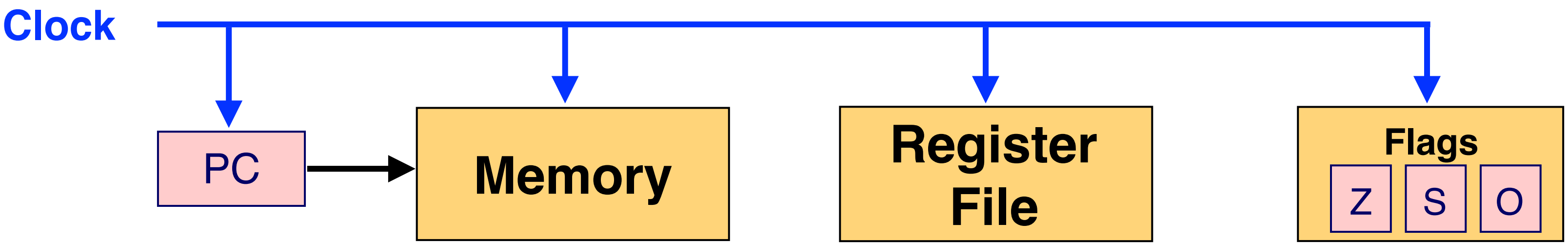
- Logic 4? Does JLE write flags?
- Need another piece of logic.
- Logic 5: if (s0 == 7) EnableF = 0; else if (s0 == 6) EnableF = 1;



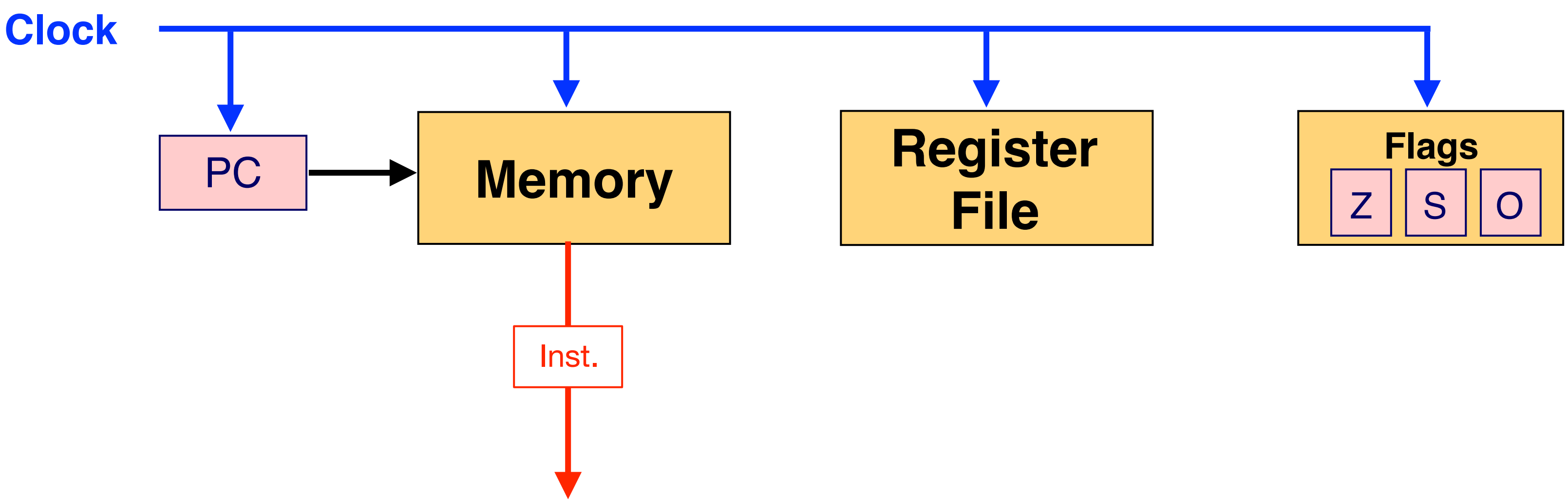
Microarchitecture (So far)



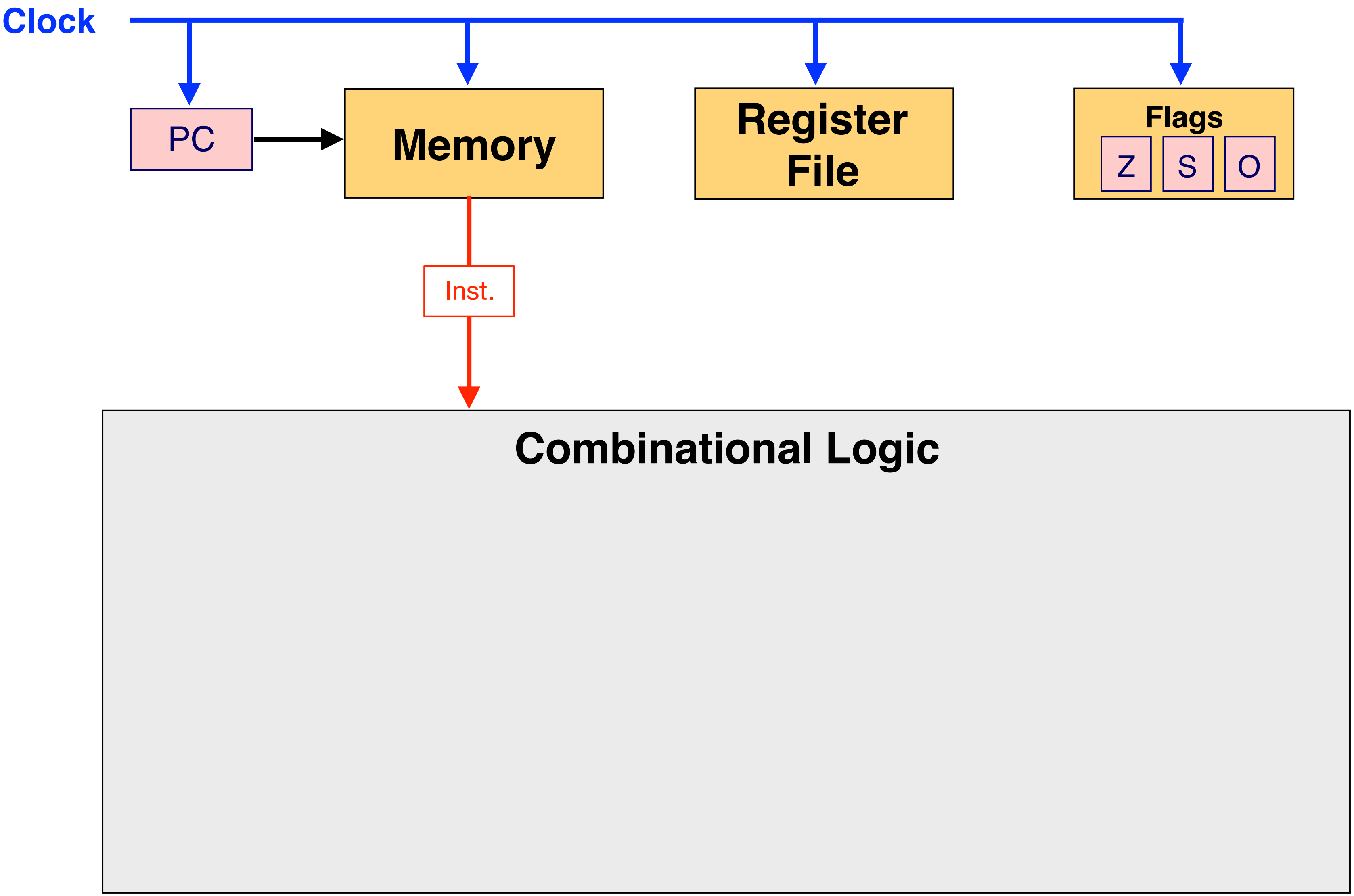
Microarchitecture (So far)



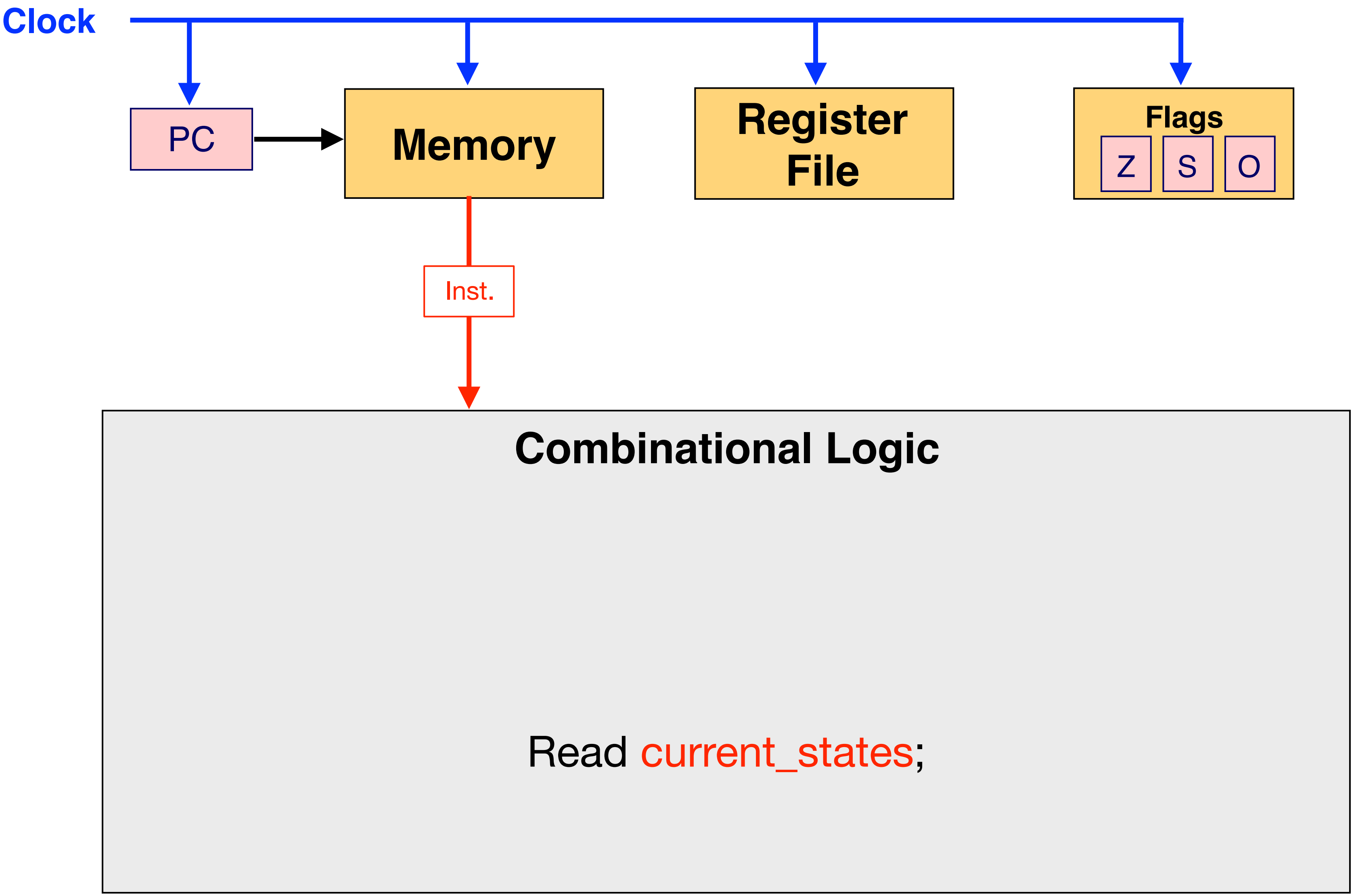
Microarchitecture (So far)



Microarchitecture (So far)

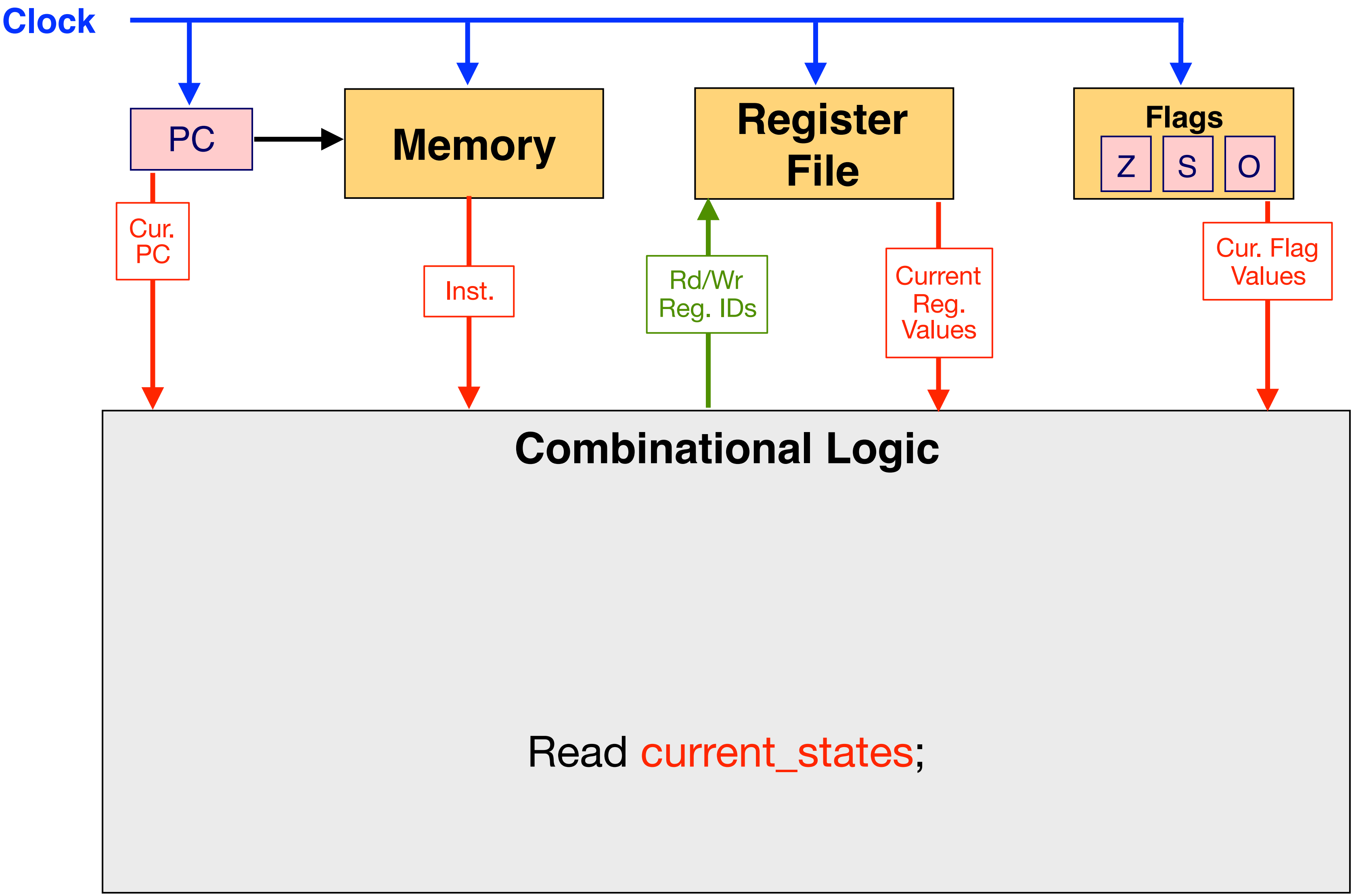


Microarchitecture (So far)



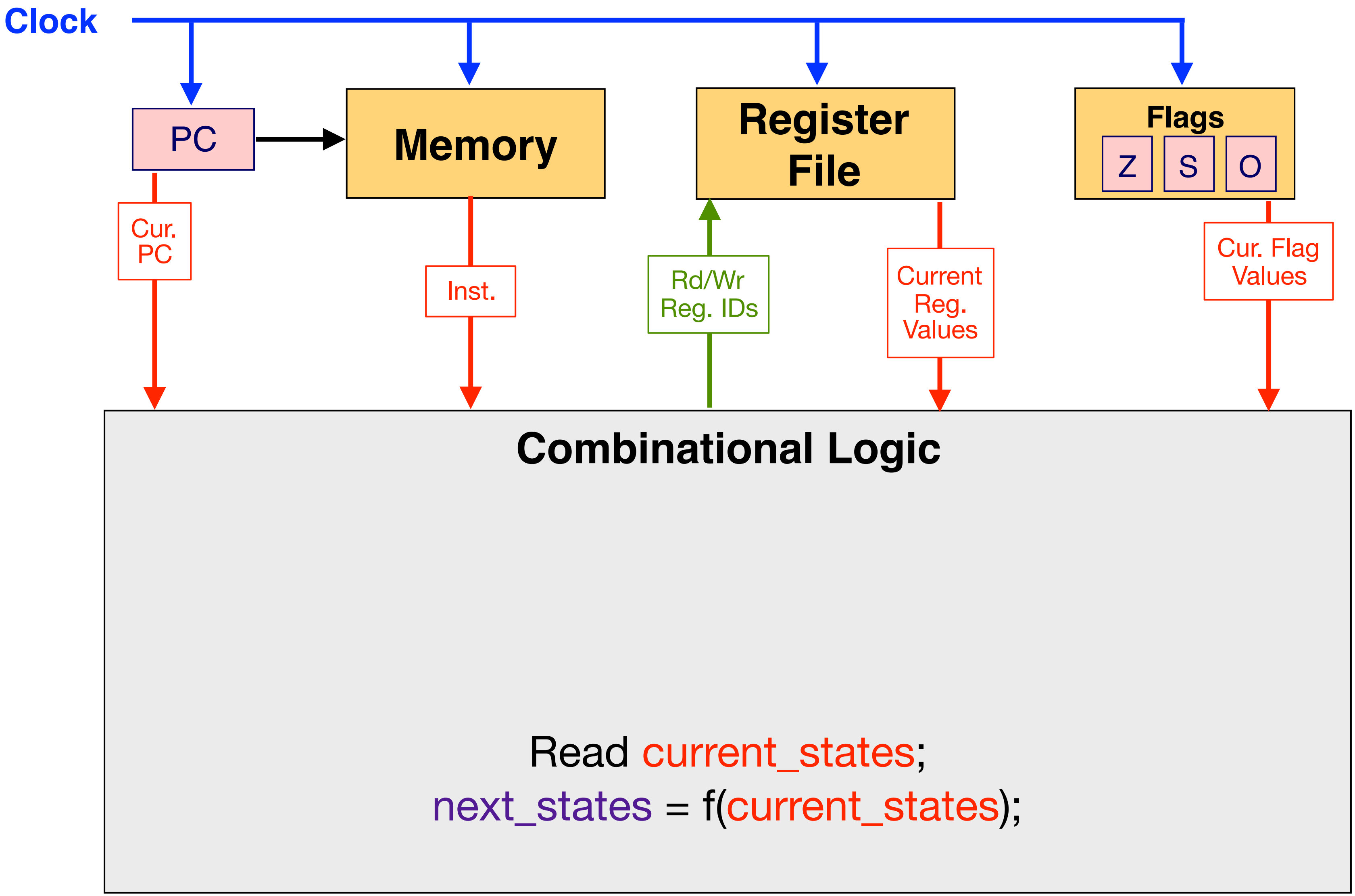
Read current_states;

Microarchitecture (So far)



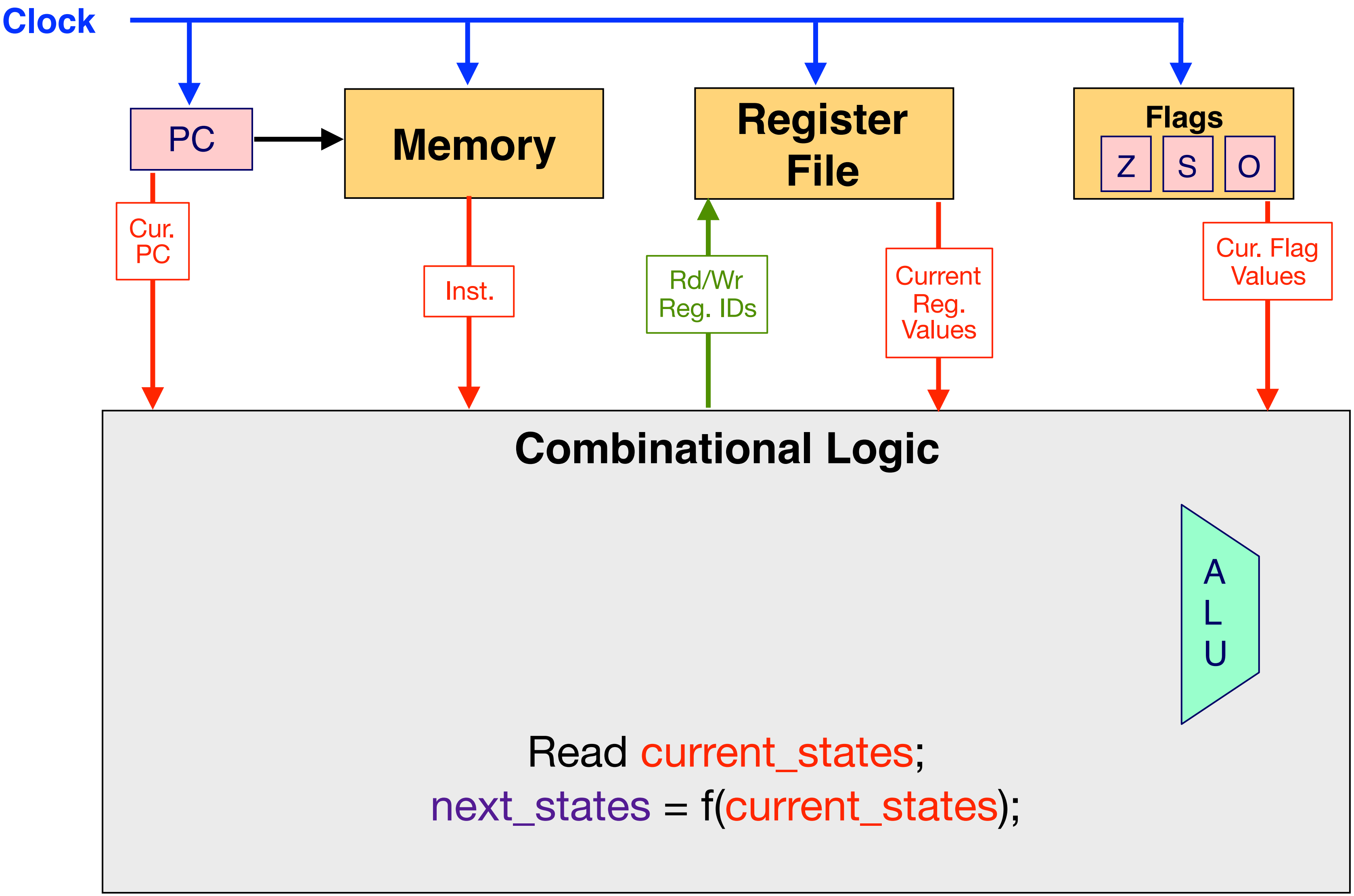
Read current_states;

Microarchitecture (So far)



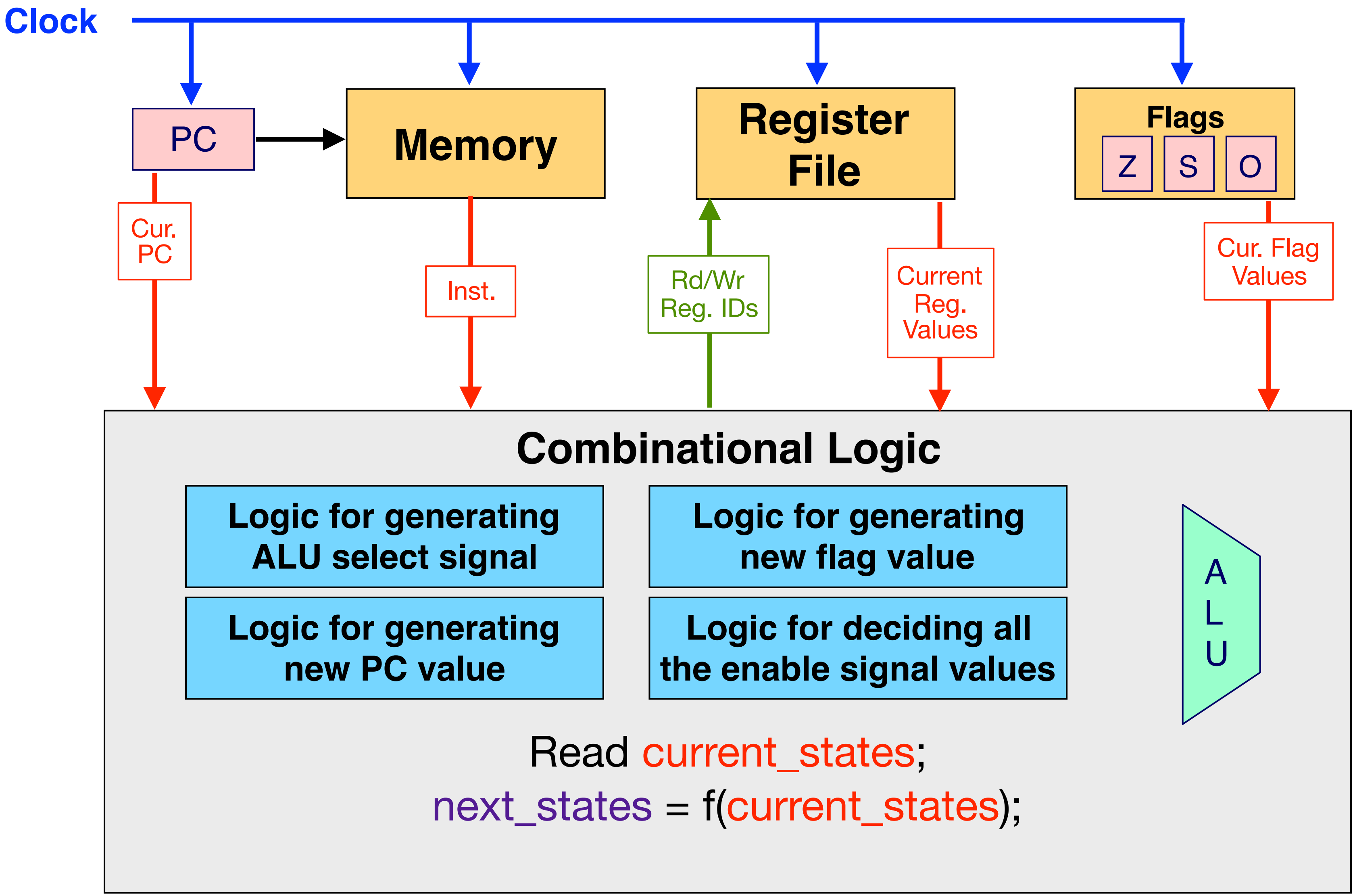
Read current_states;
next_states = f(current_states);

Microarchitecture (So far)



Read current_states;
next_states = f(current_states);

Microarchitecture (So far)



Clock

PC

Memory

Register File

Flags

Z S O

Cur. PC

Inst.

Rd/Wr Reg. IDs

Current Reg. Values

Cur. Flag Values

Combinational Logic

Logic for generating ALU select signal

Logic for generating new flag value

Logic for generating new PC value

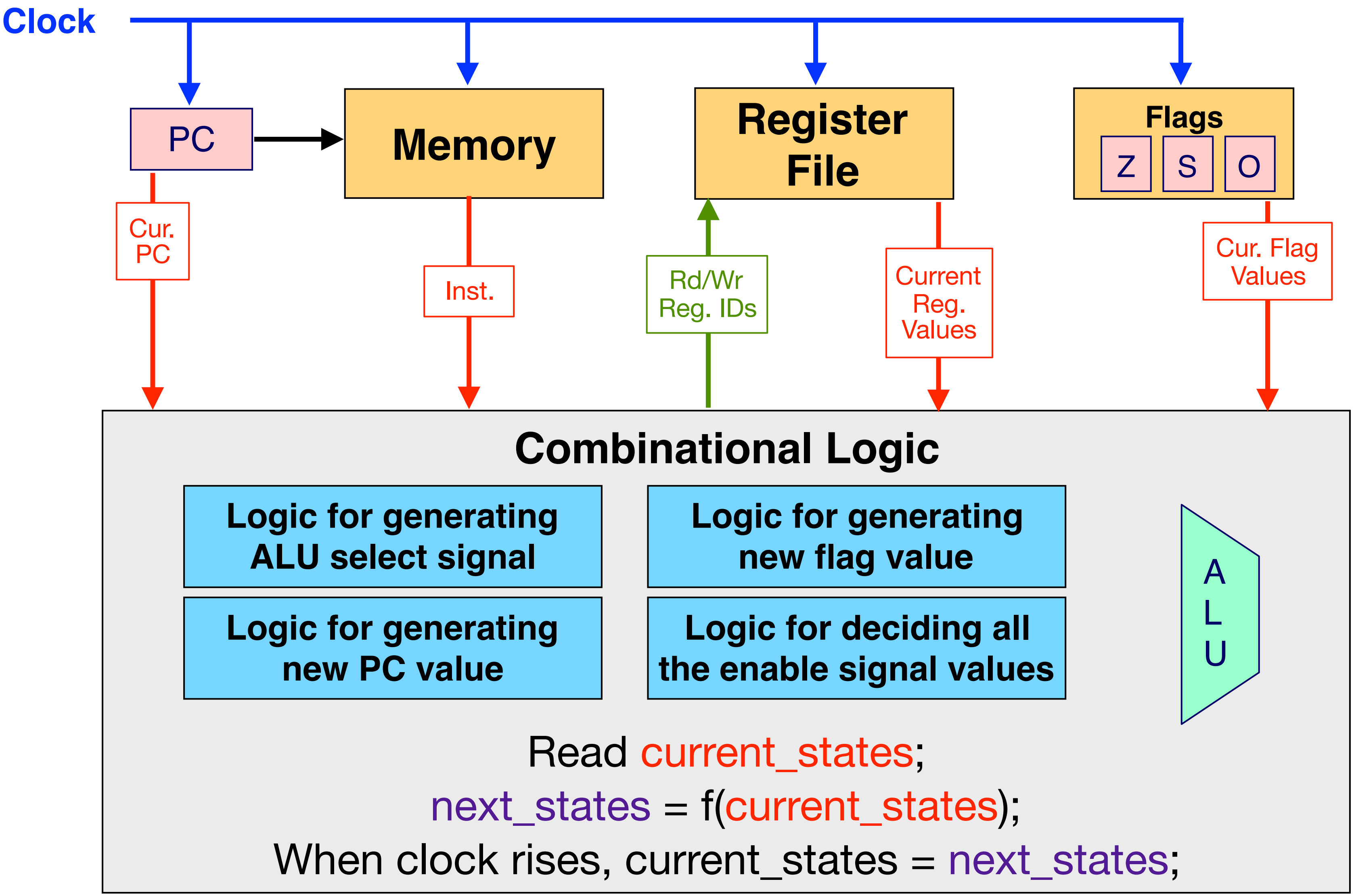
Logic for deciding all the enable signal values

ALU

Read current_states;

next_states = f(current_states);

Microarchitecture (So far)

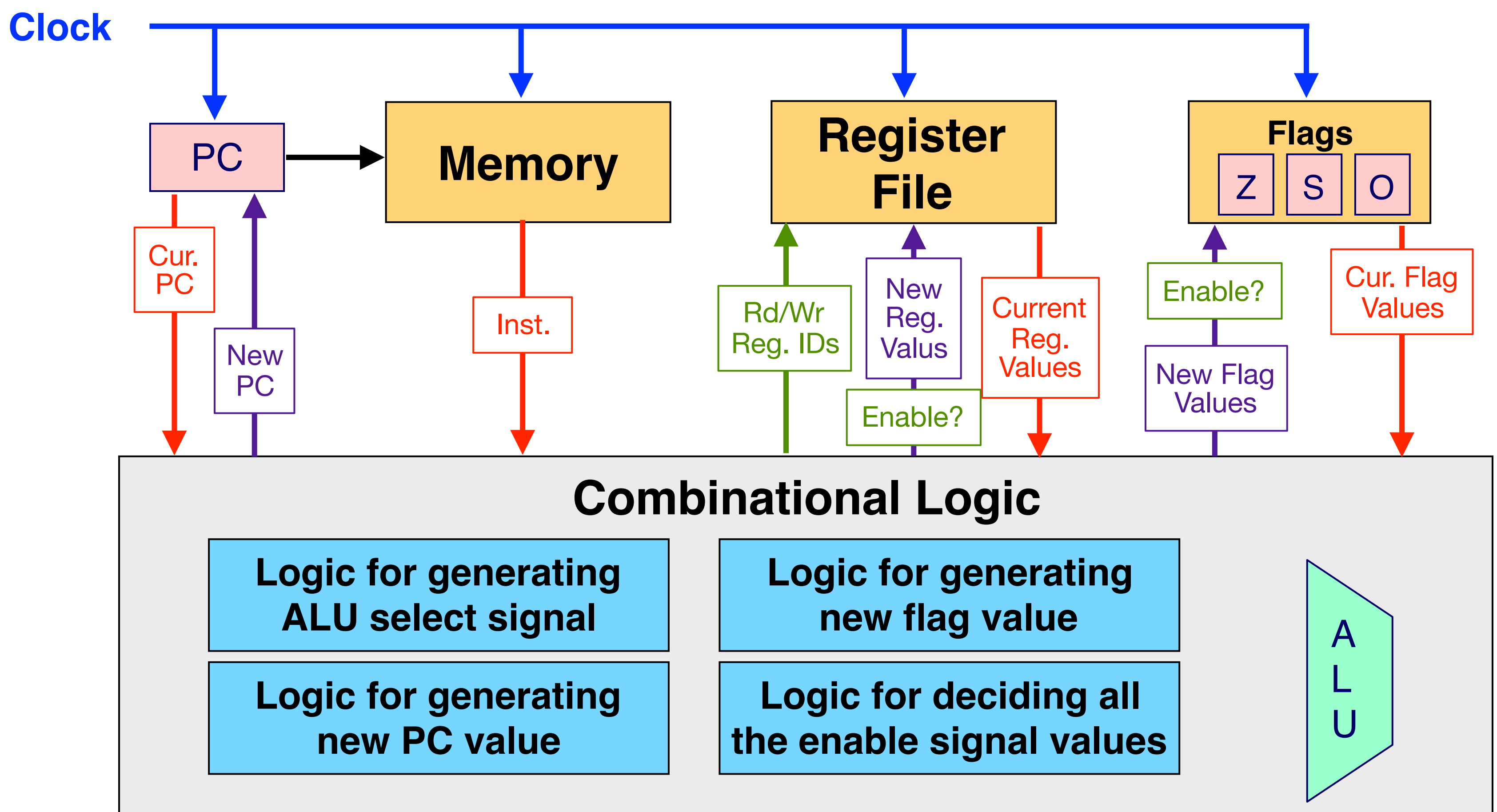

Clock
PC
Memory
Register File
Flags
Z
S
O
Cur. PC
Inst.
Rd/Wr Reg. IDs
Current Reg. Values
Cur. Flag Values
Combinational Logic
Logic for generating ALU select signal
Logic for generating new flag value
Logic for generating new PC value
Logic for deciding all the enable signal values
ALU

Read current_states;

next_states = f(current_states);

When clock rises, current_states = next_states;

Microarchitecture (So far)



Read current_states;

next_states = f(current_states);

When clock rises, current_states = next_states;

Executing a MOV instruction

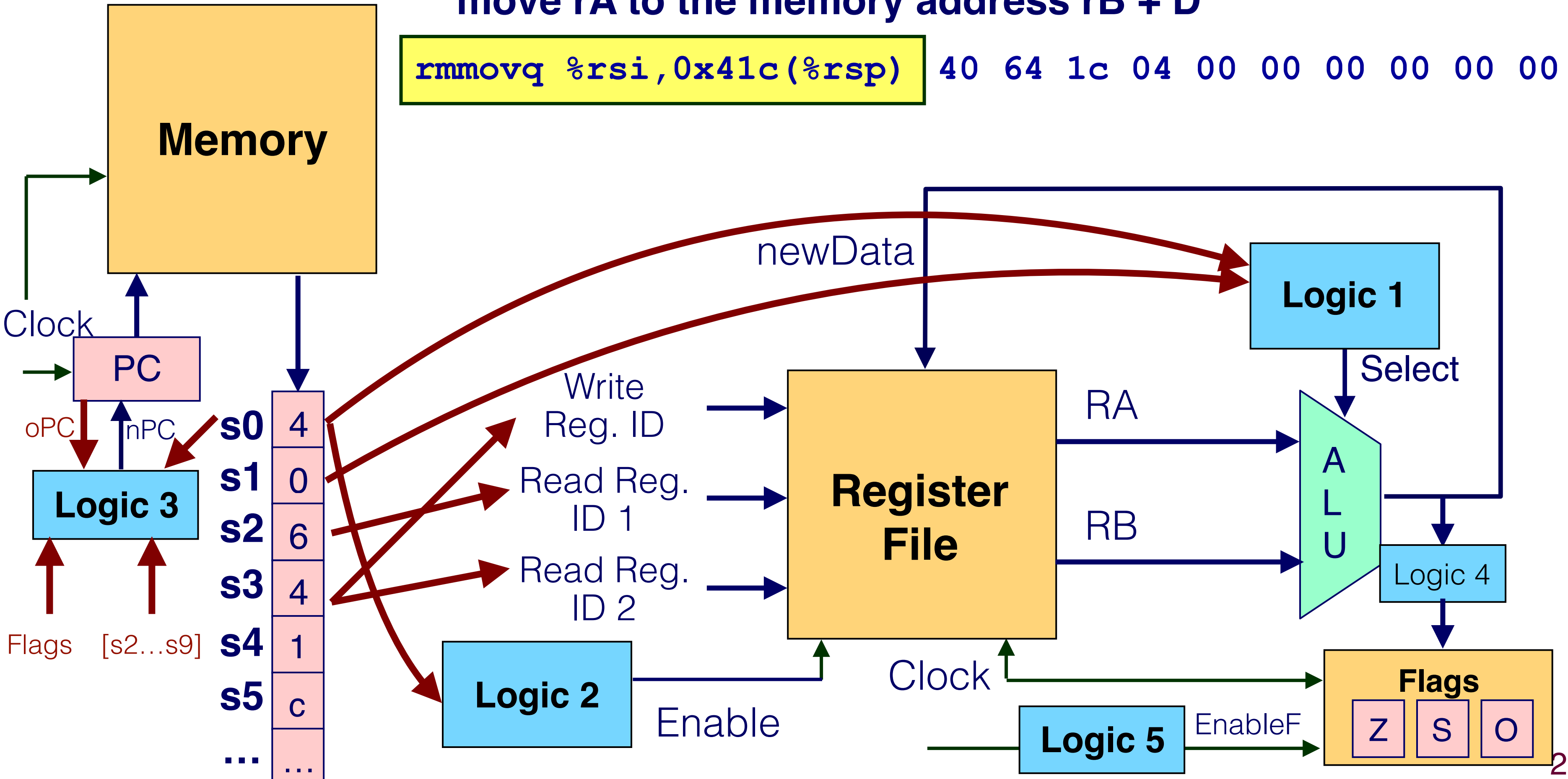
- How do we modify the hardware to execute a move instruction?

`rmmovq rA, D(rB)`

4	0	rA	rB	D

move rA to the memory address rB + D

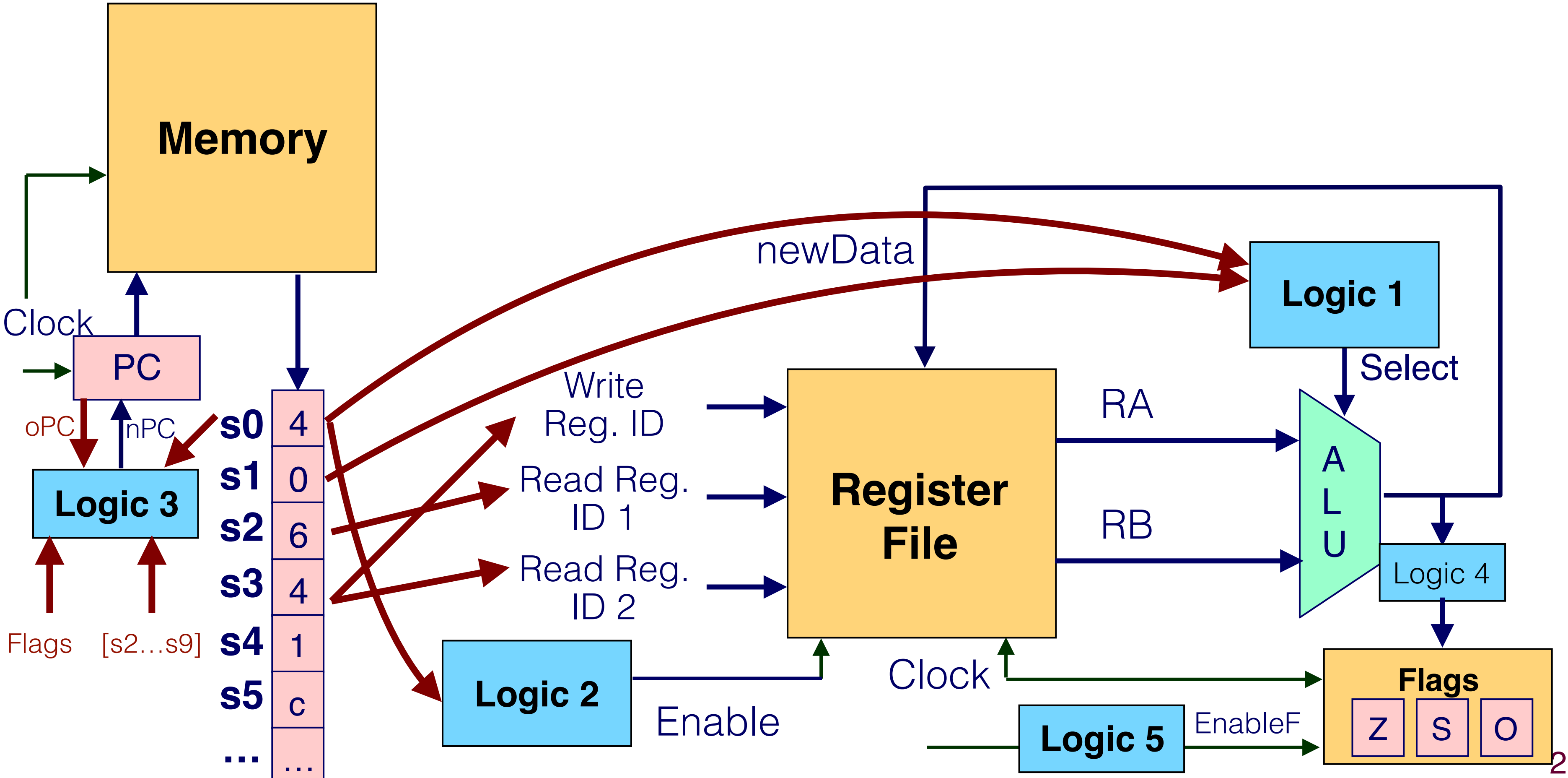
`rmmovq %rsi,0x41c(%rsp)` **40 64 1c 04 00 00 00 00 00 00**



move rA to the memory address rB + D

`rmmovq rA, D(rB)`

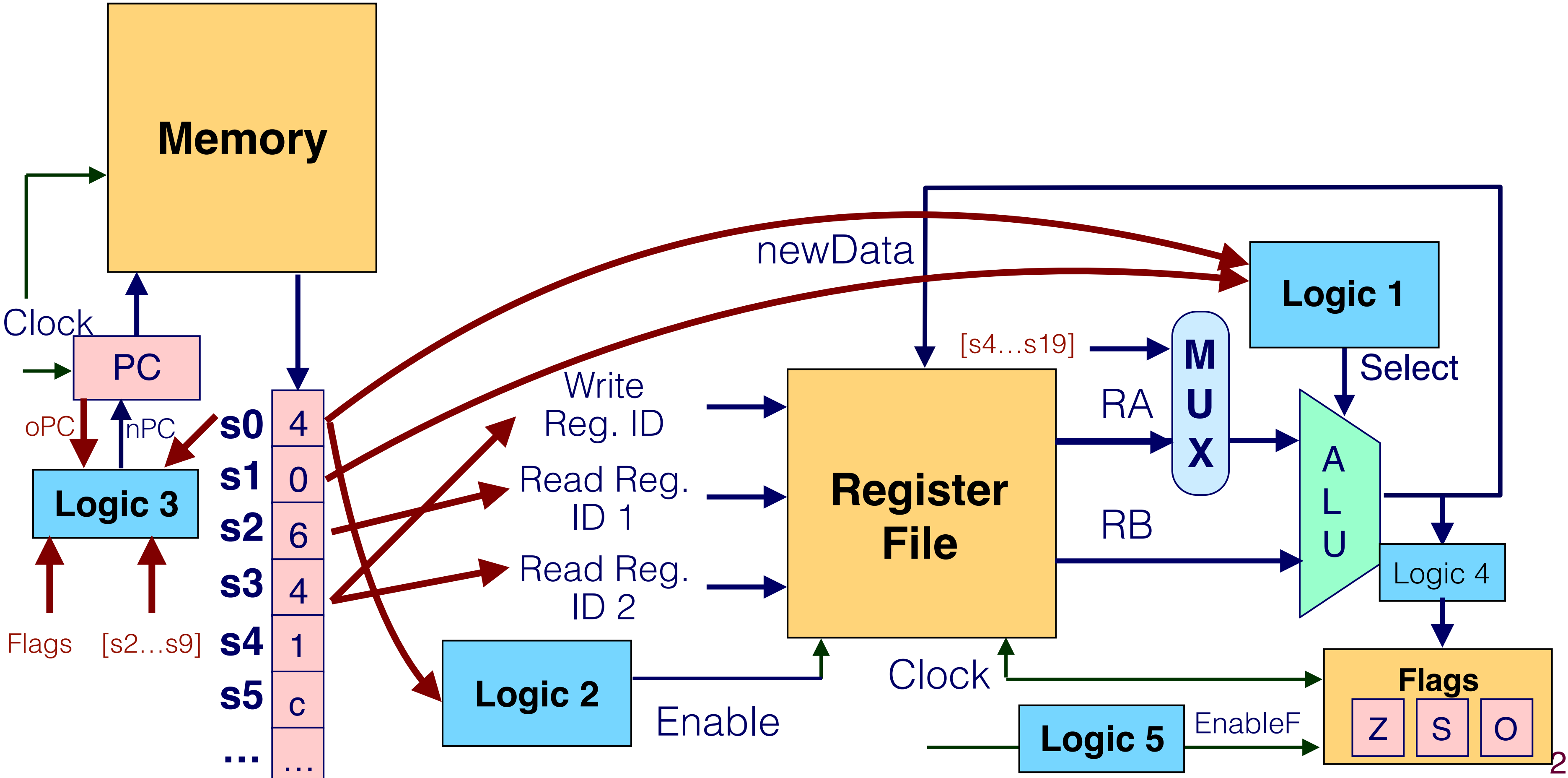
4	0	rA	rB	D



move rA to the memory address rB + D

`rmmovq rA, D(rB)`

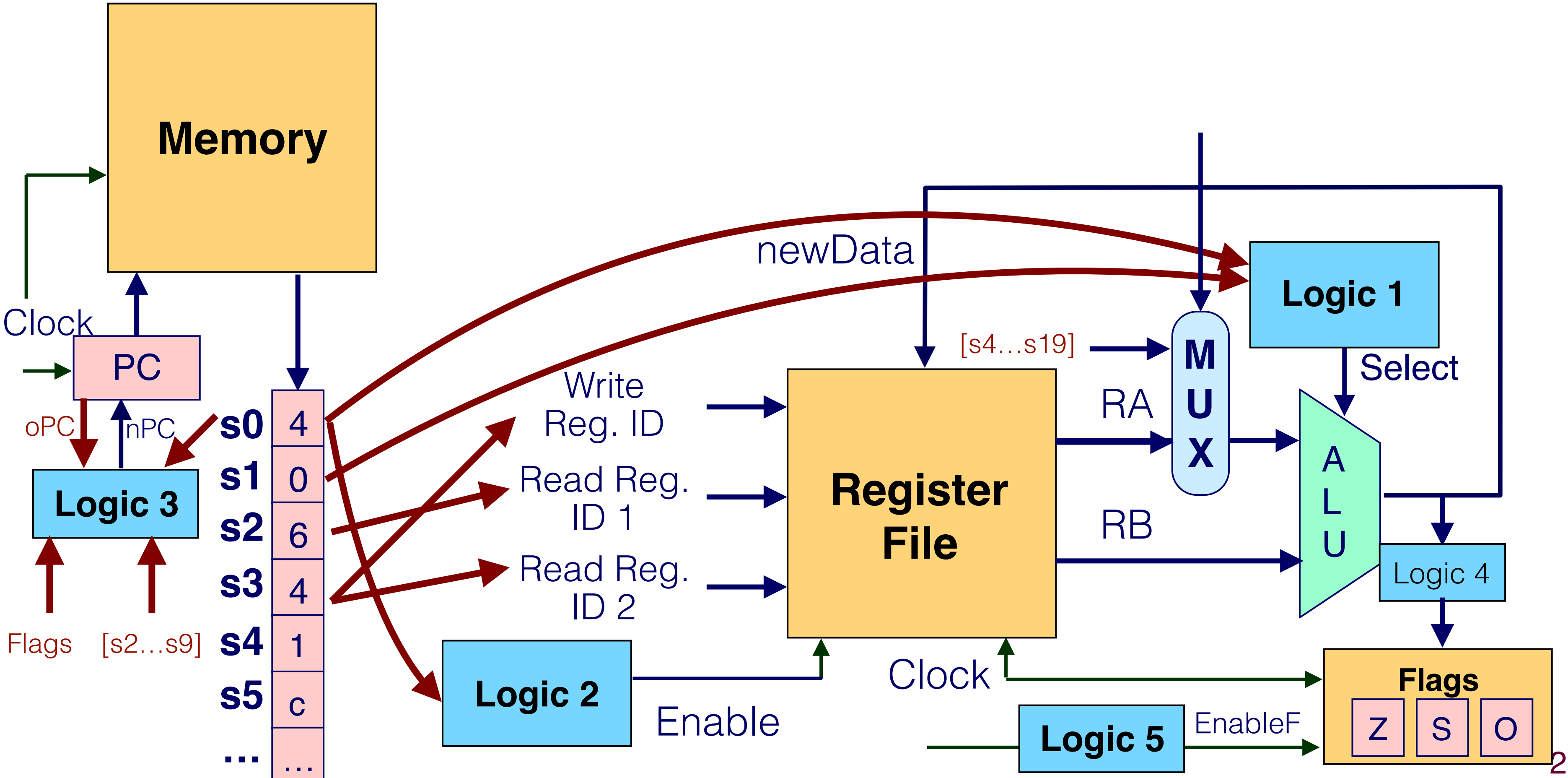
4	0	rA	rB	D



move rA to the memory address rB + D

`rmmovq rA, D(rB)`

4	0	rA	rB	D

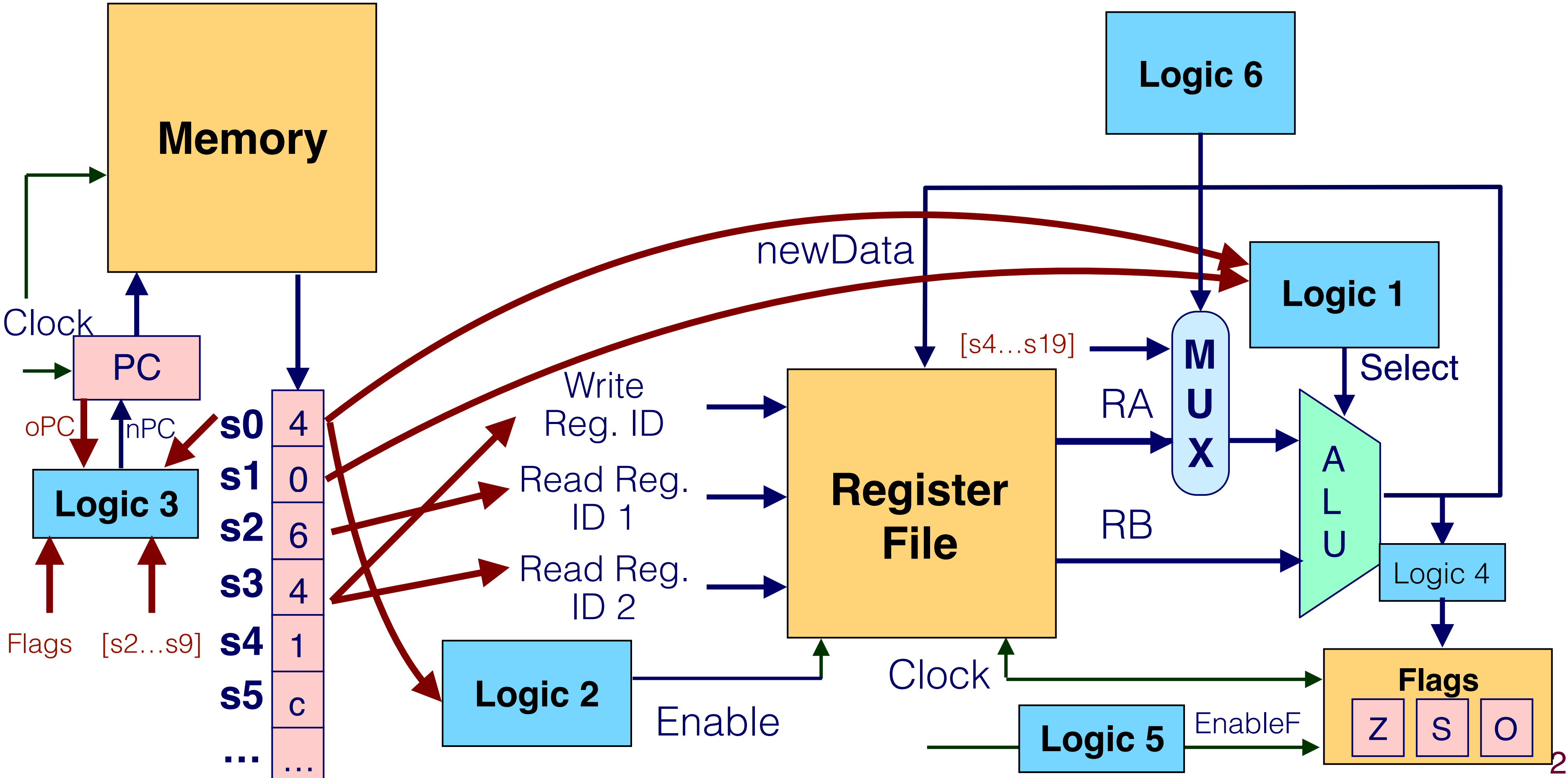


move rA to the memory address rB + D

`rmmovq rA, D(rB)`

4	0	rA	rB	D

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

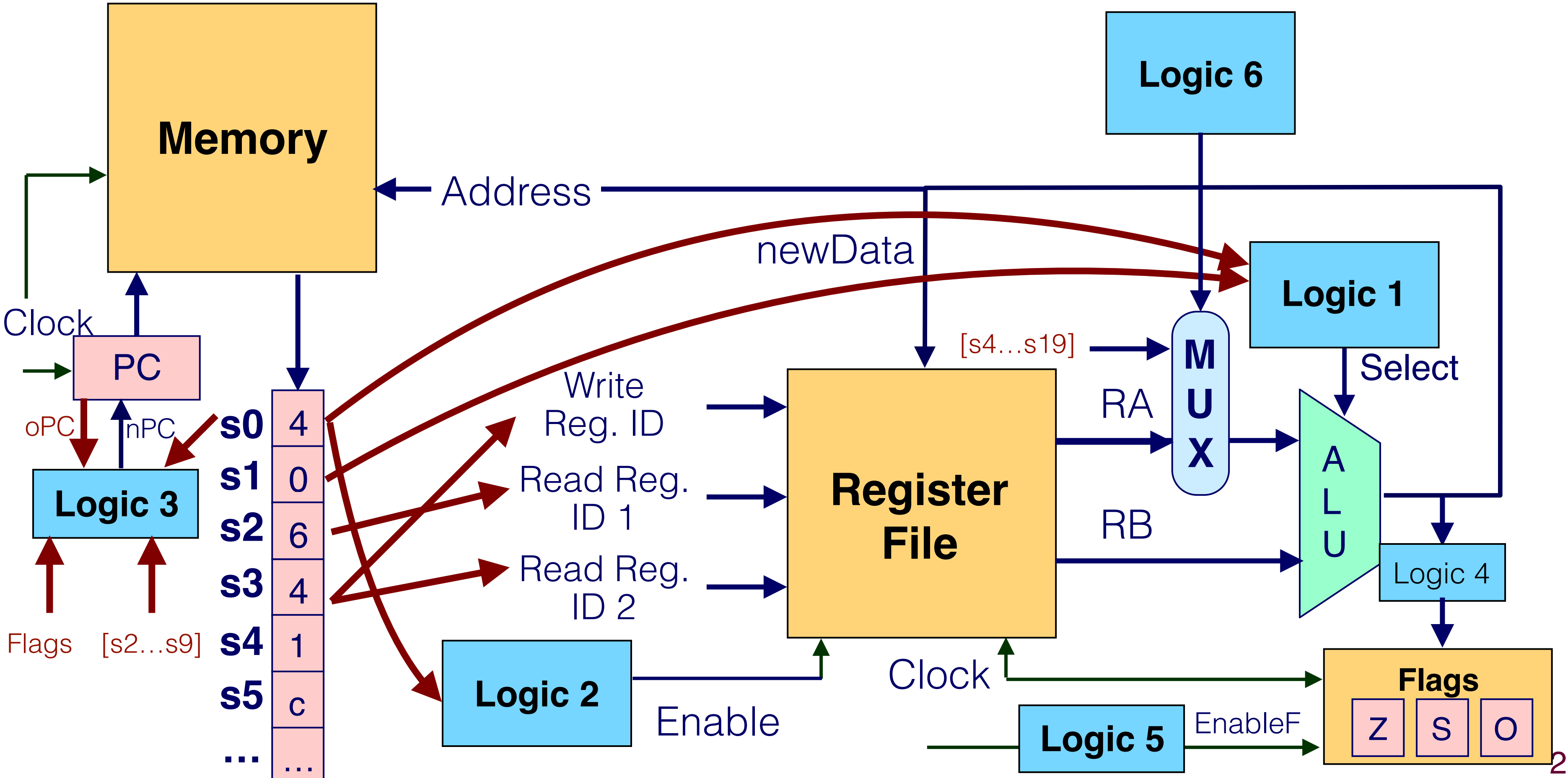


move rA to the memory address rB + D

`rmmovq` rA, D(rB)

4	0	rA	rB	D

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

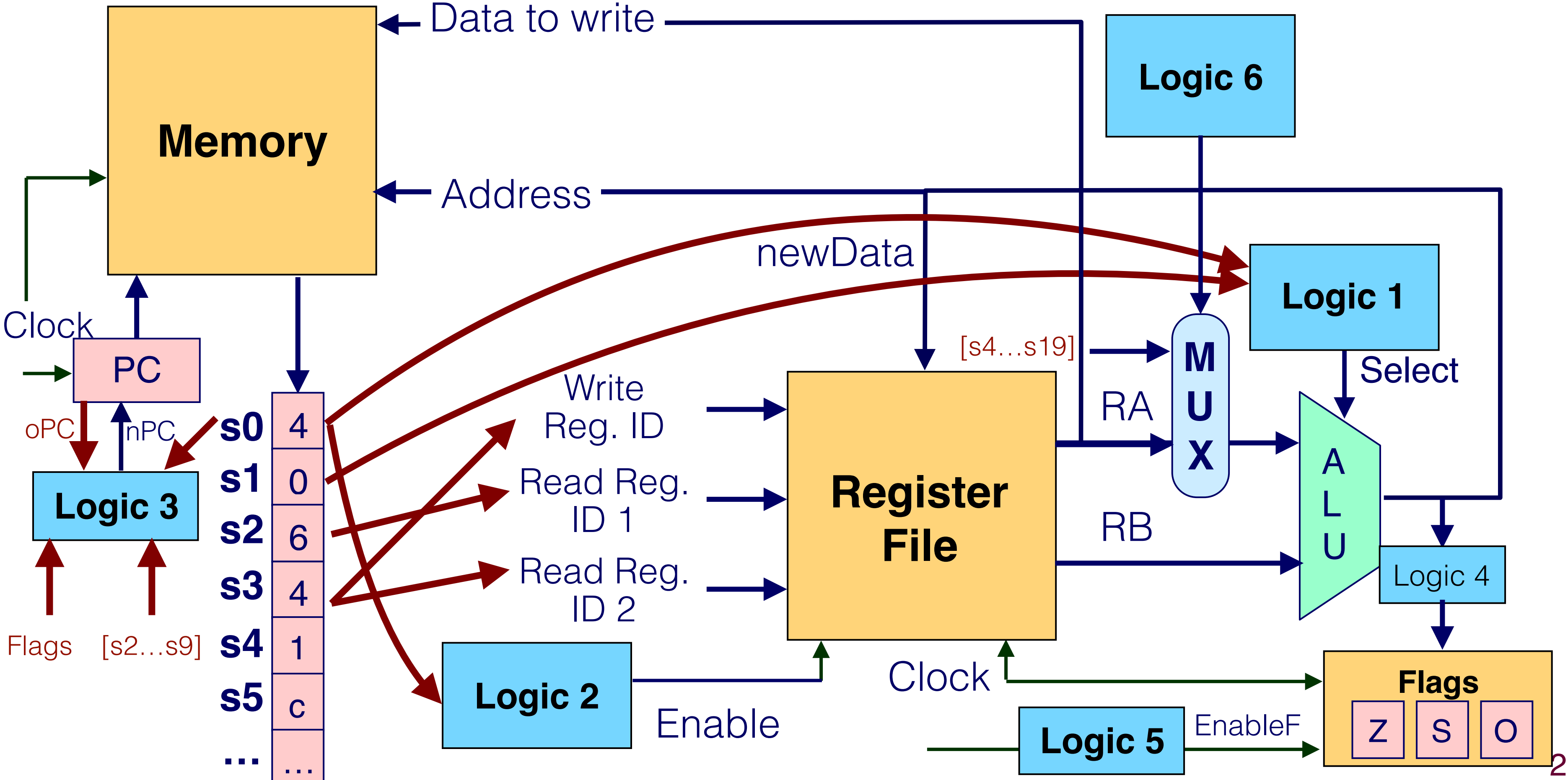


move rA to the memory address rB + D

rmmovq rA, D(rB)

4	0	rA	rB	D

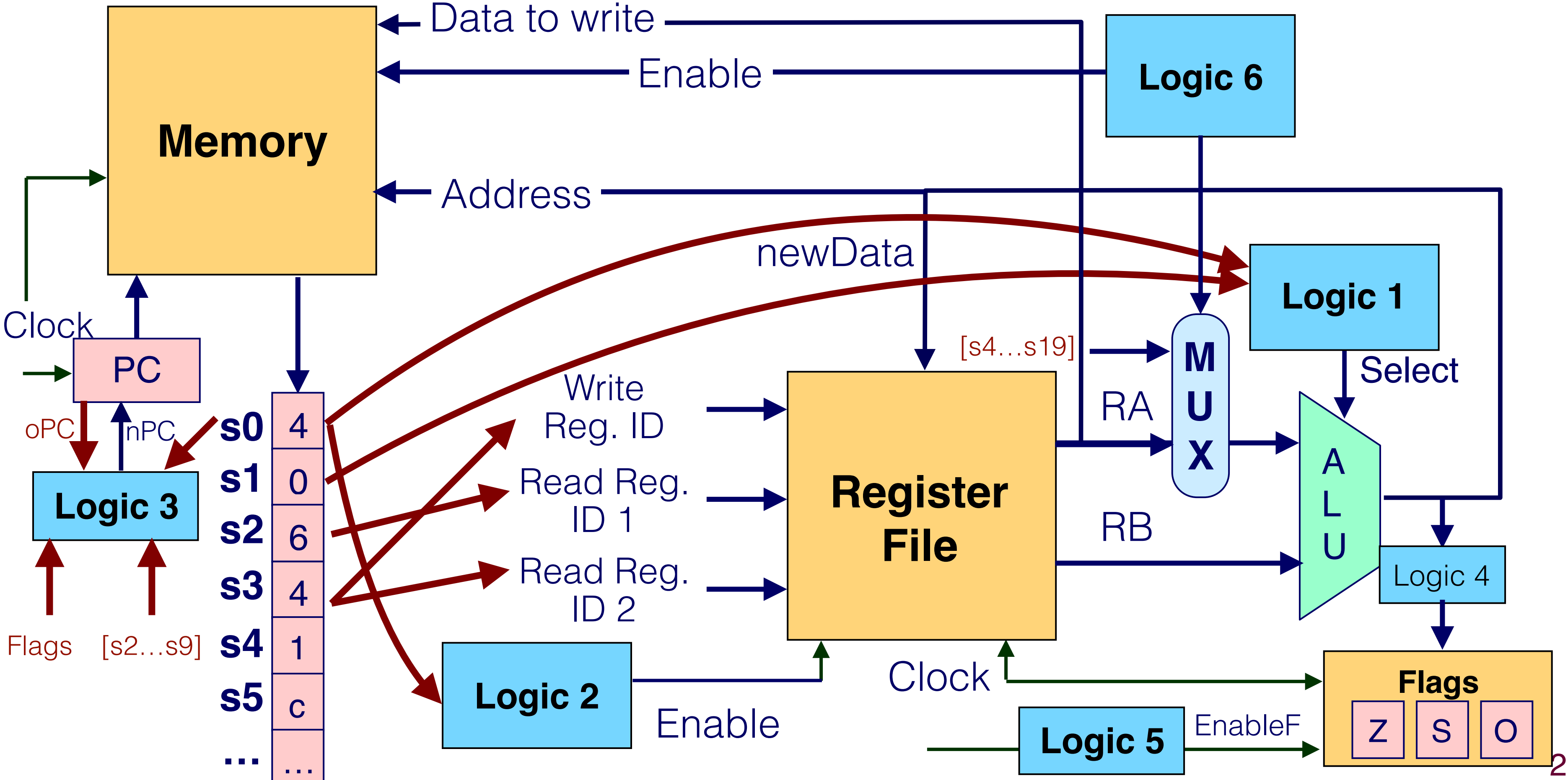
- Need new logic (Logic 6) to select the input to the ALU for Enable.



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.

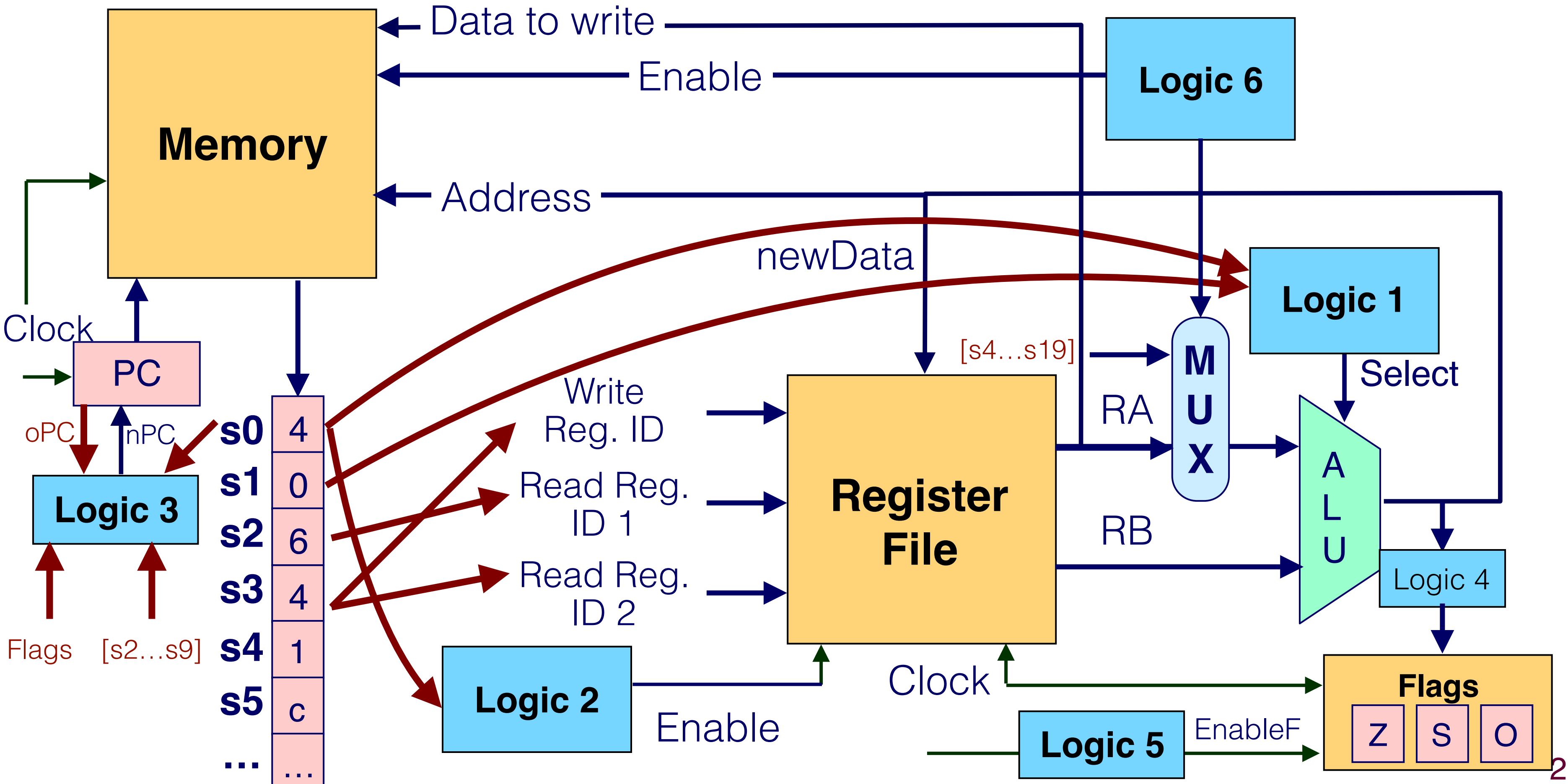


move rA to the memory address rB + D

`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?

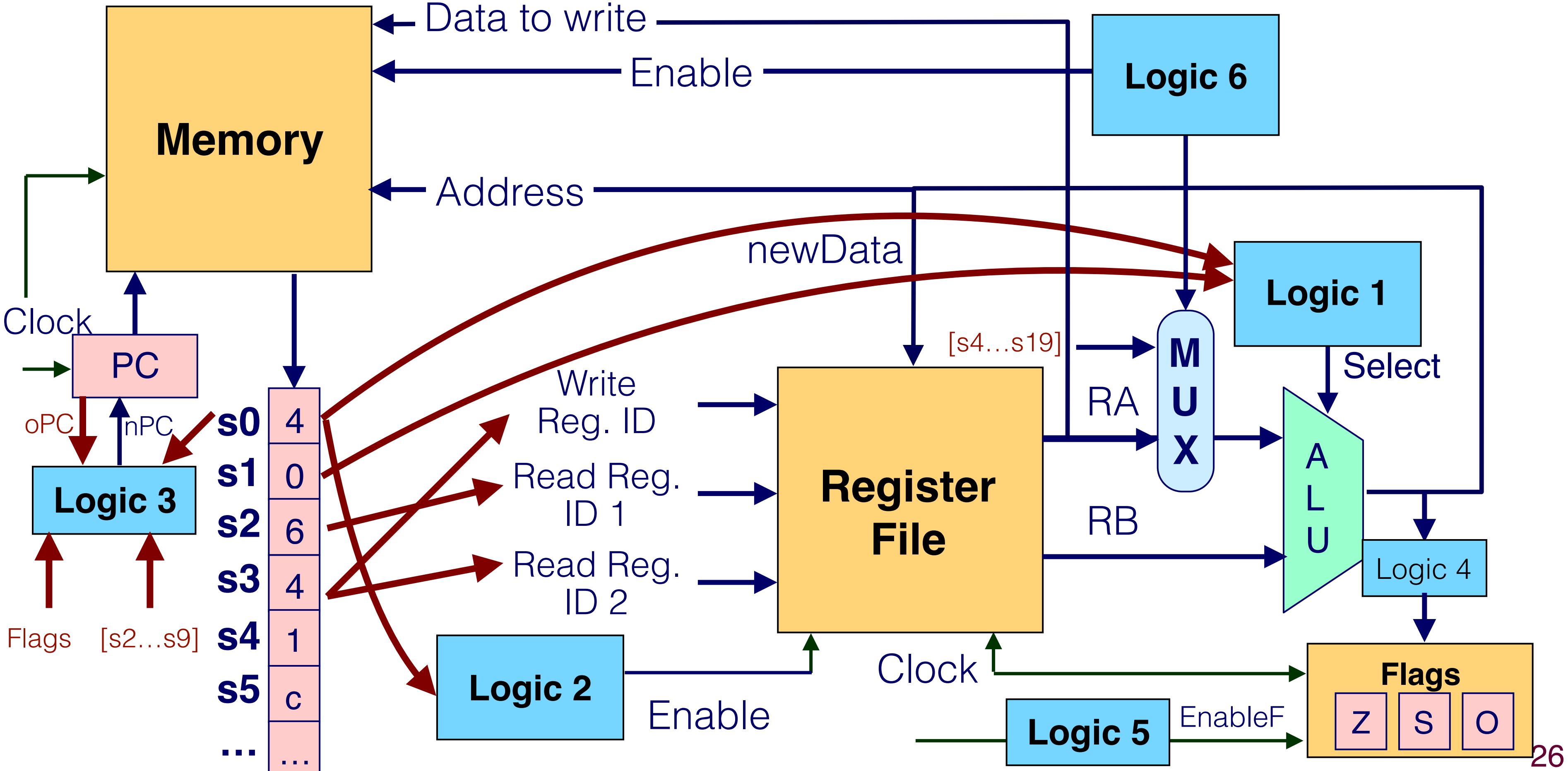


How About Memory to Register MOV?

move data at memory address rB + D to rA

`mrmovq D(rB), rA`

4	0	rA	rB	D

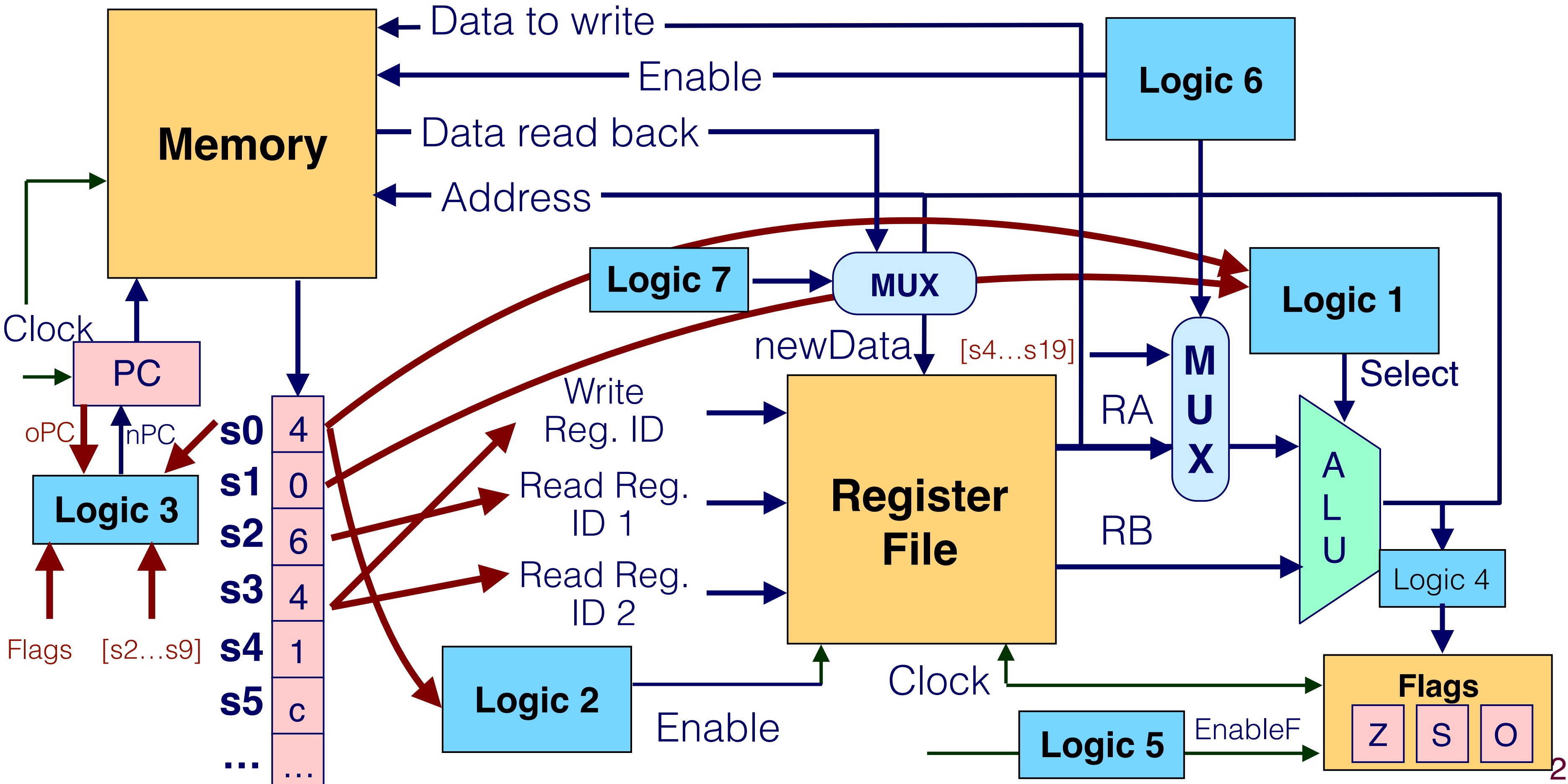


How About Memory to Register MOV?

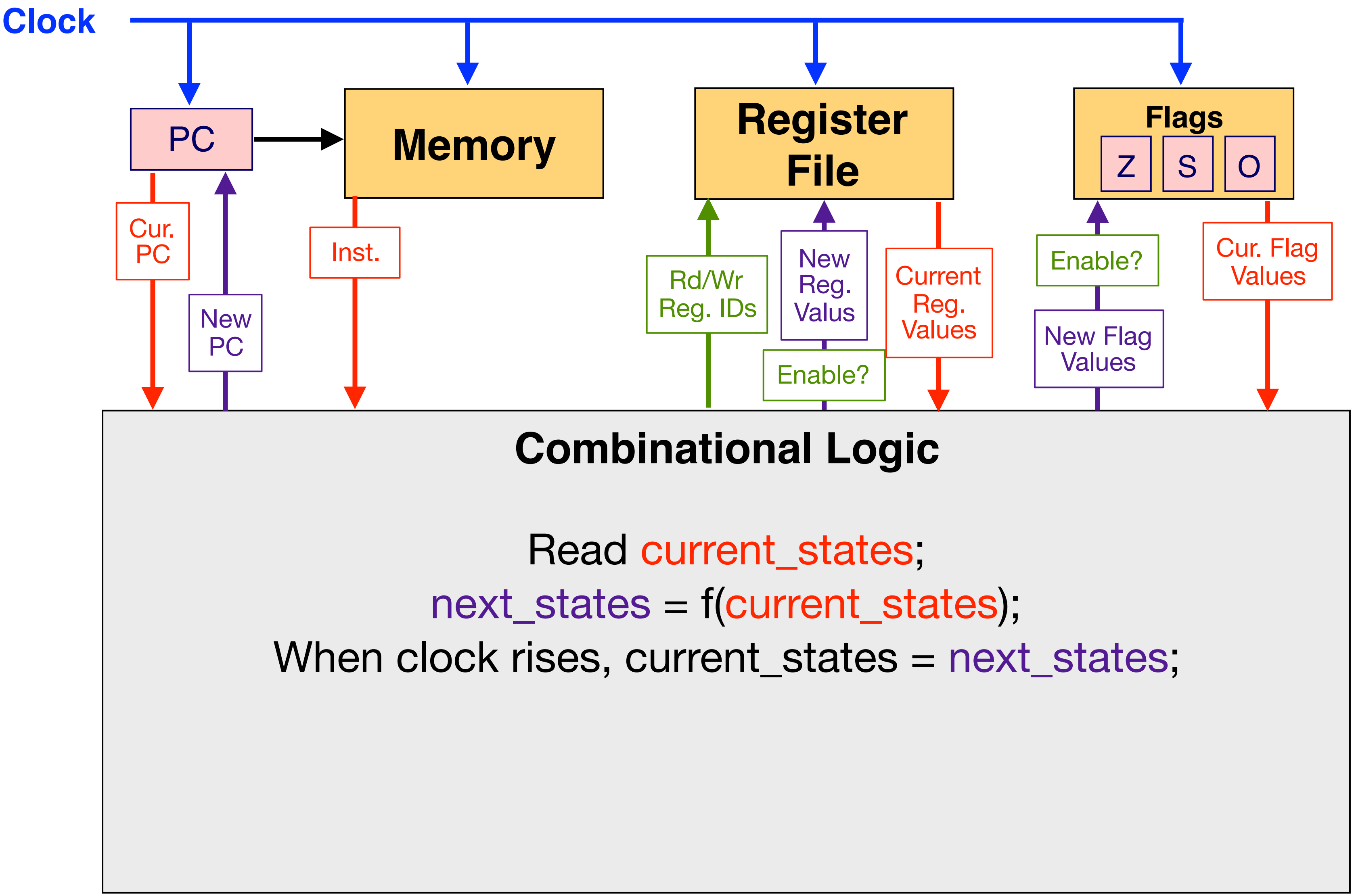
move data at memory address rB + D to rA

`mrmovq D(rB), rA`

4	0	rA	rB	D



Microarchitecture (with MOV)

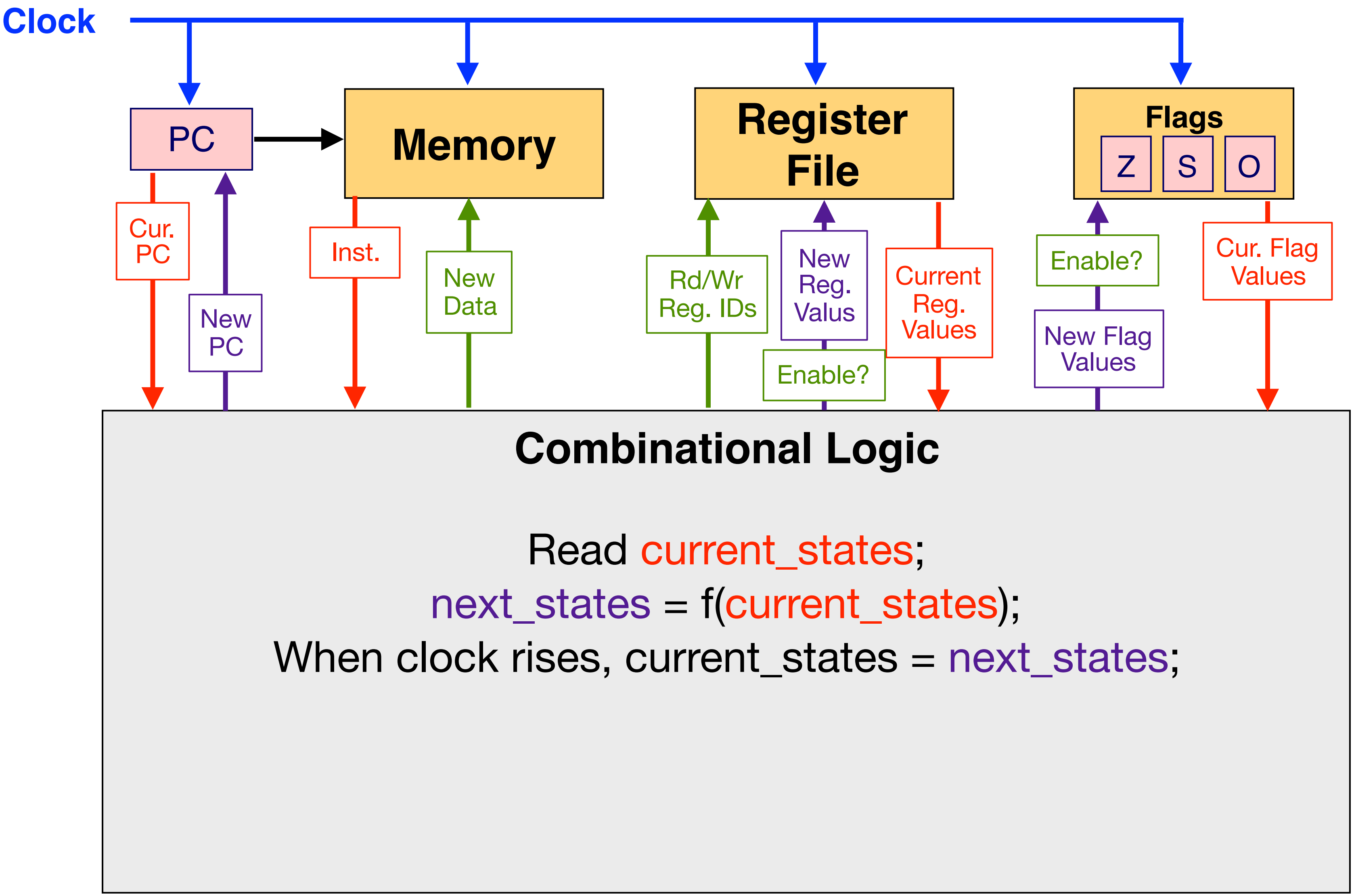


Read current_states;

next_states = f(current_states);

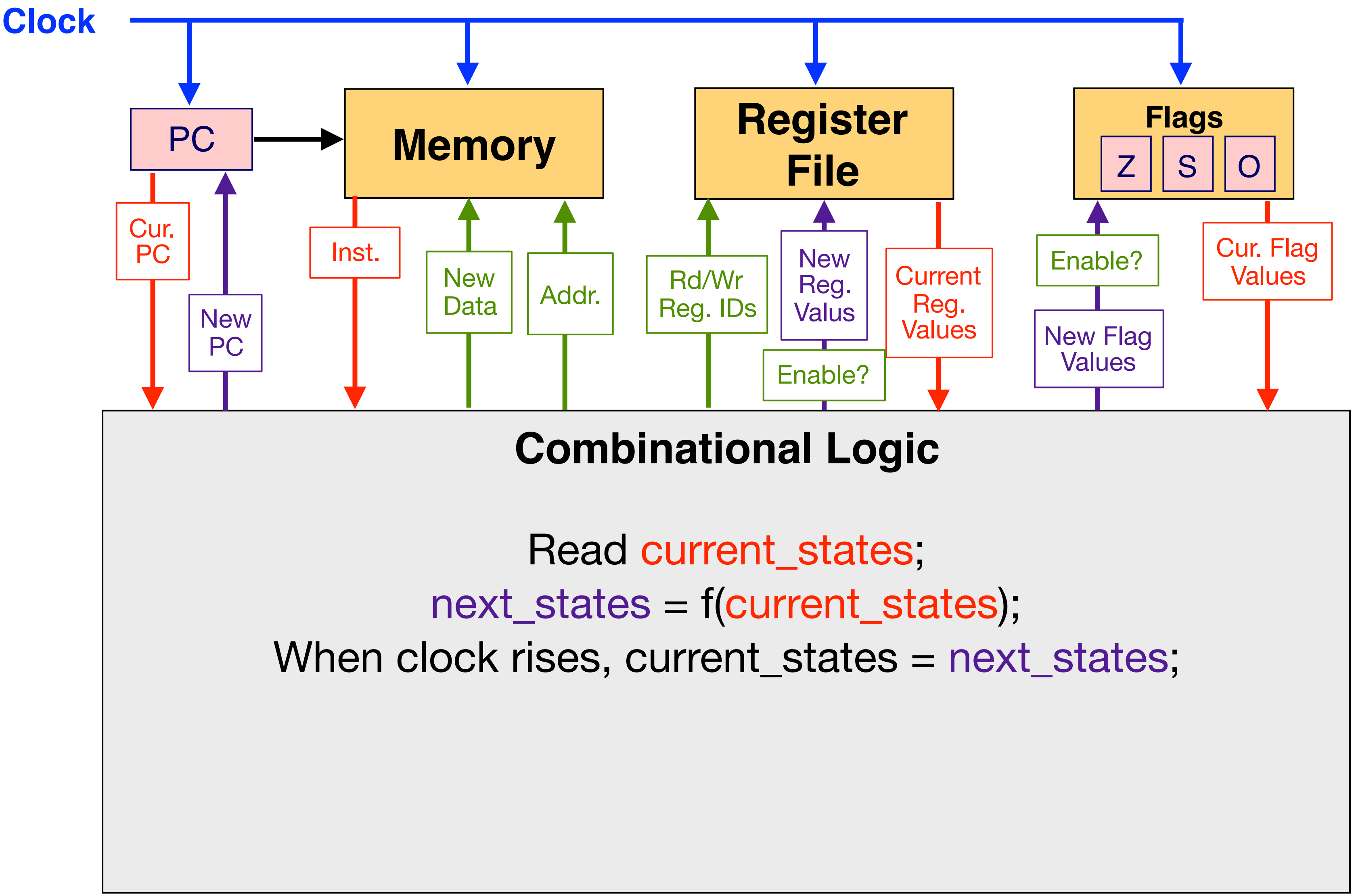
When clock rises, current_states = next_states;

Microarchitecture (with MOV)



Read current_states;
next_states = f(current_states);
When clock rises, current_states = next_states;

Microarchitecture (with MOV)

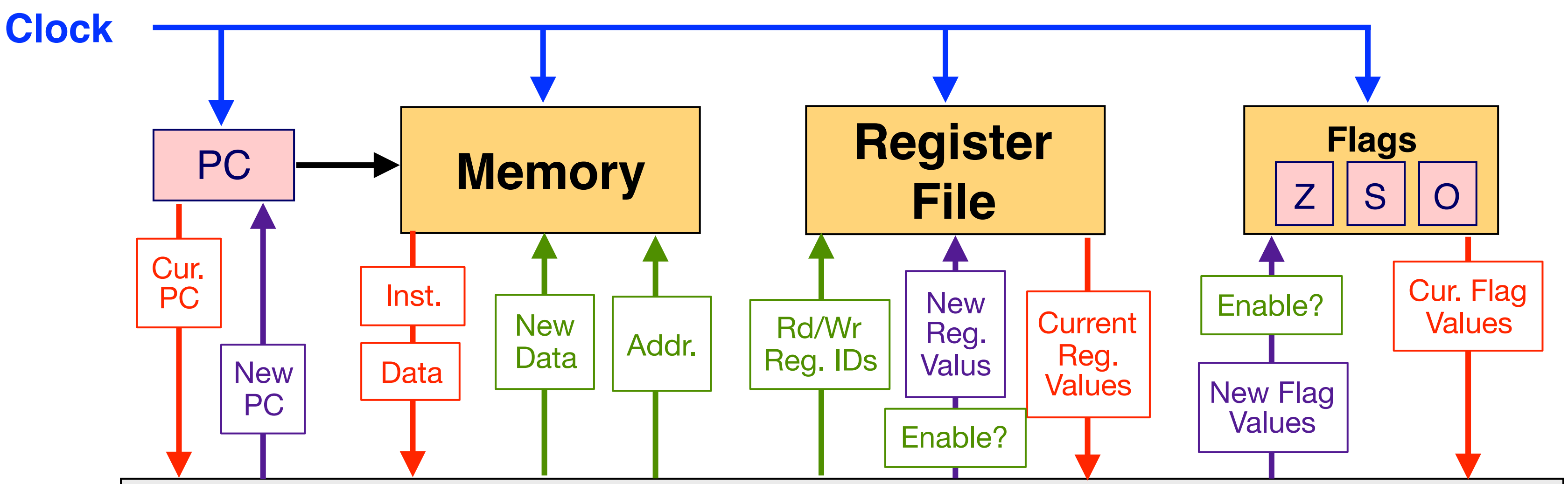


Read current_states;

next_states = f(current_states);

When clock rises, current_states = next_states;

Microarchitecture (with MOV)



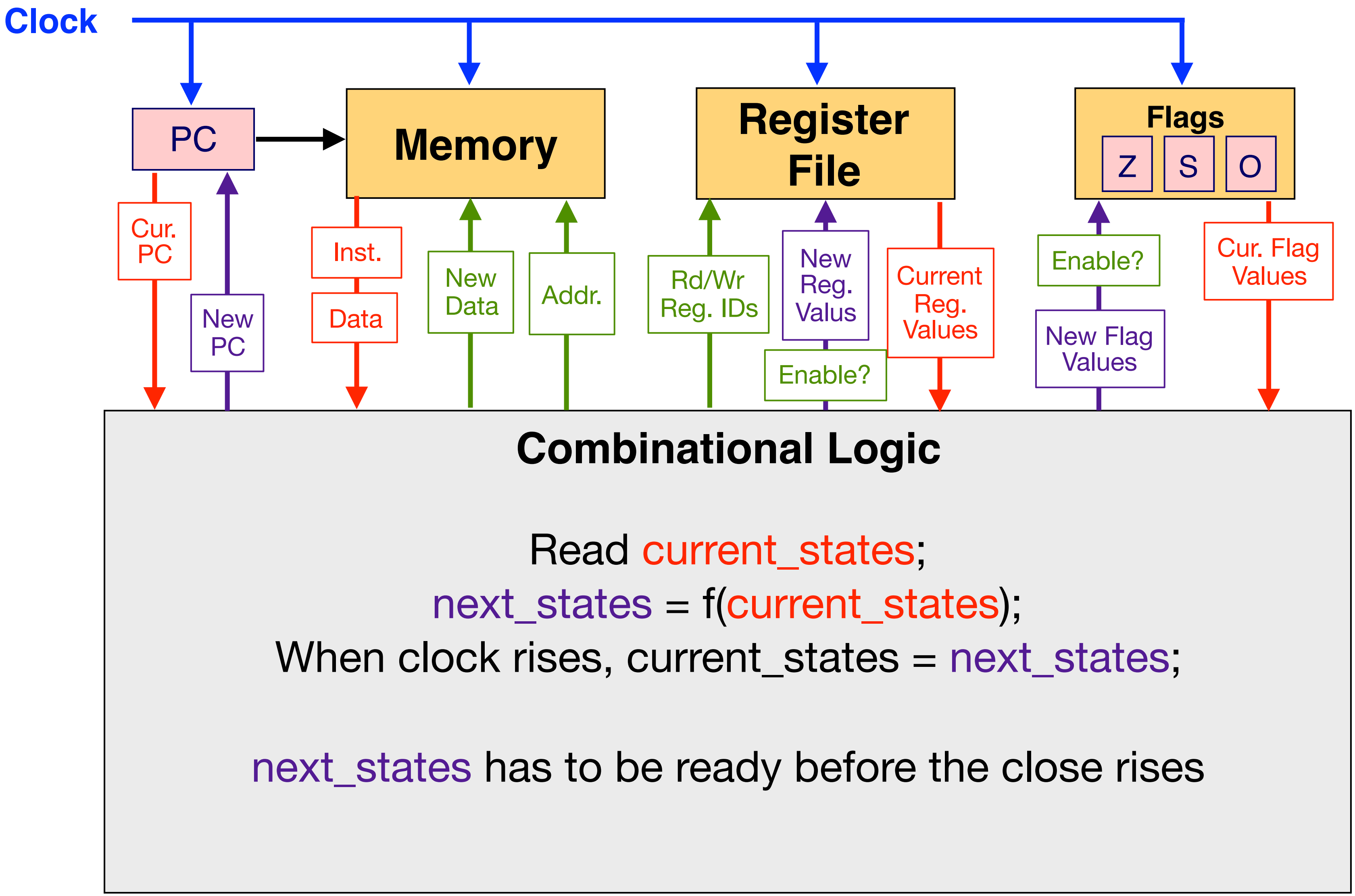
Combinational Logic

Read current_states;

next_states = f(current_states);

When clock rises, current_states = next_states;

Microarchitecture (with MOV)



Combinational Logic

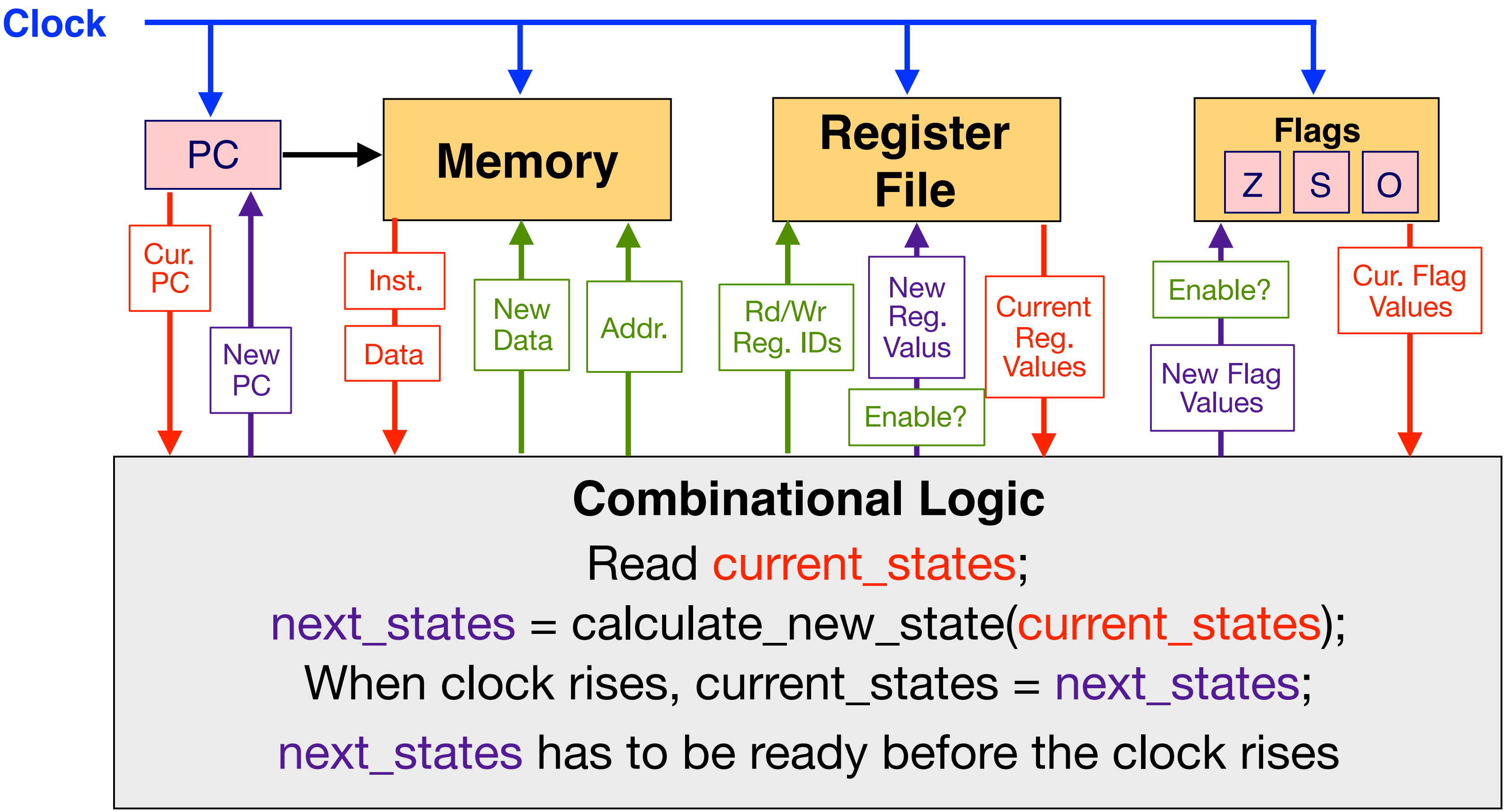
Read current_states;

next_states = f(current_states);

When clock rises, current_states = next_states;

next_states has to be ready before the close rises

Single-Cycle Microarchitecture



Combinational Logic

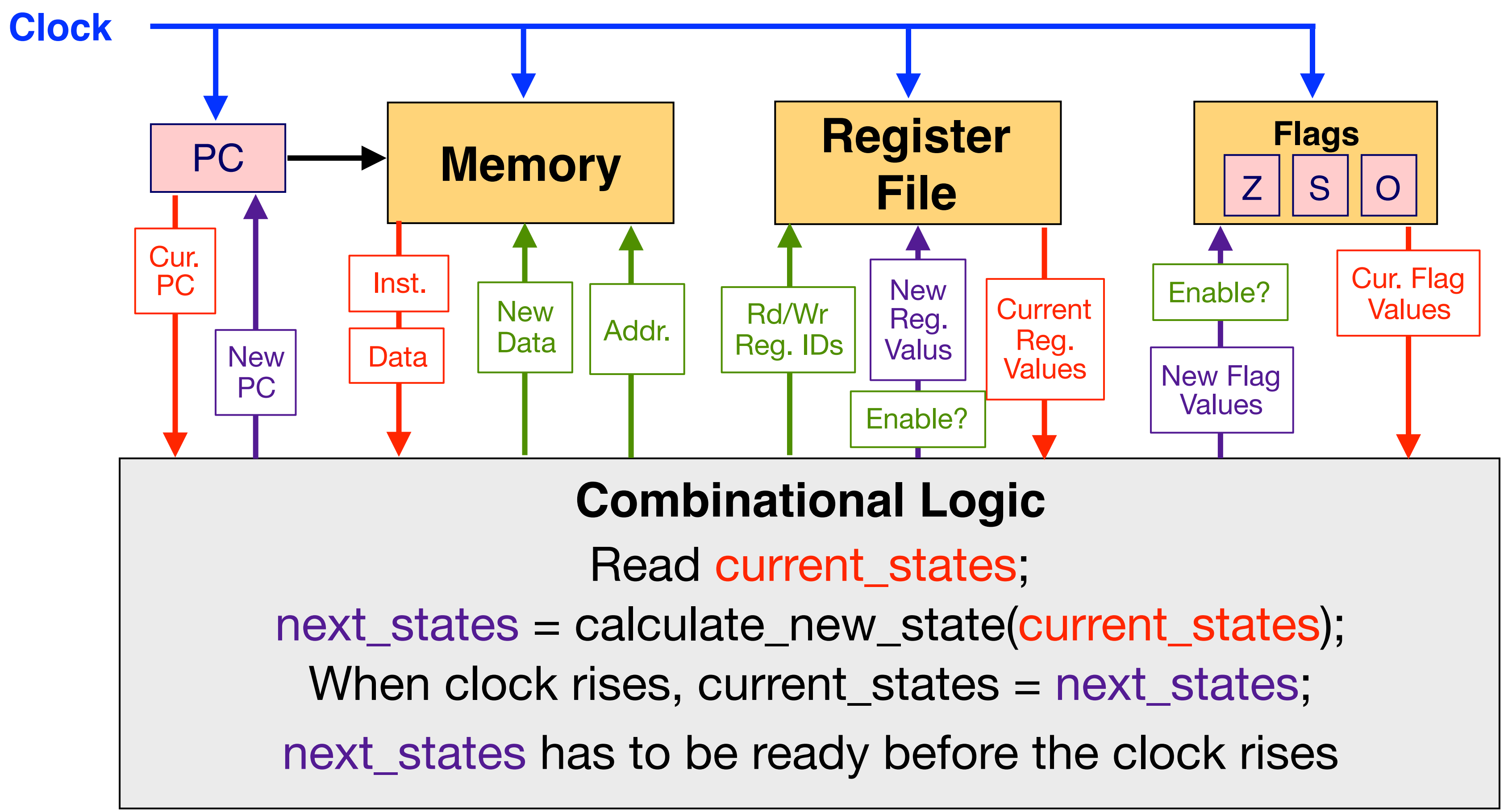
Read current_states;

next_states = calculate_new_state(current_states);

When clock rises, current_states = next_states;

next_states has to be ready before the clock rises

Single-Cycle Microarchitecture



Combinational Logic

Read current_states;

next_states = calculate_new_state(current_states);

When clock rises, current_states = next_states;

next_states has to be ready before the clock rises

Key principles:

States are stored in storage units, e.g., Flip-flops (and SRAM and DRAM, later..)

New states are calculated by combination logic.

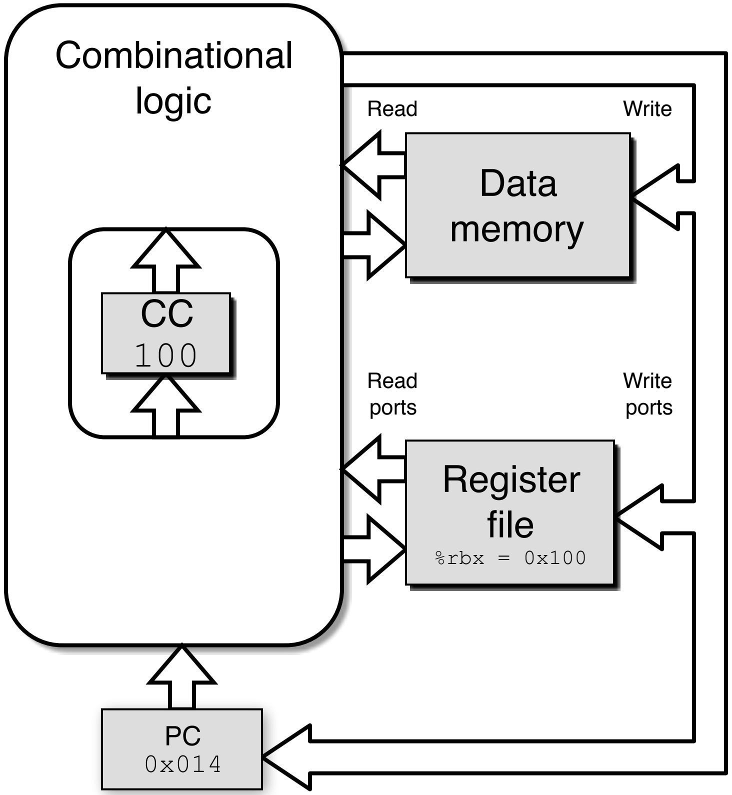
Single-Cycle Microarchitecture: Illustration

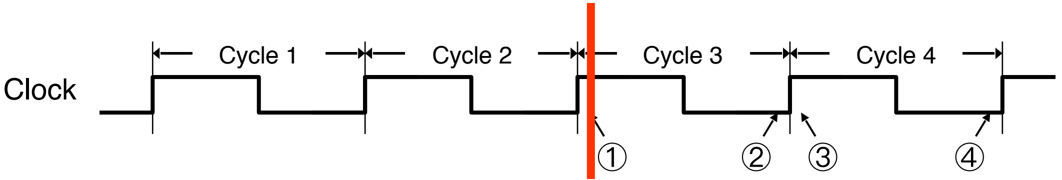
Think of it as a state machine

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

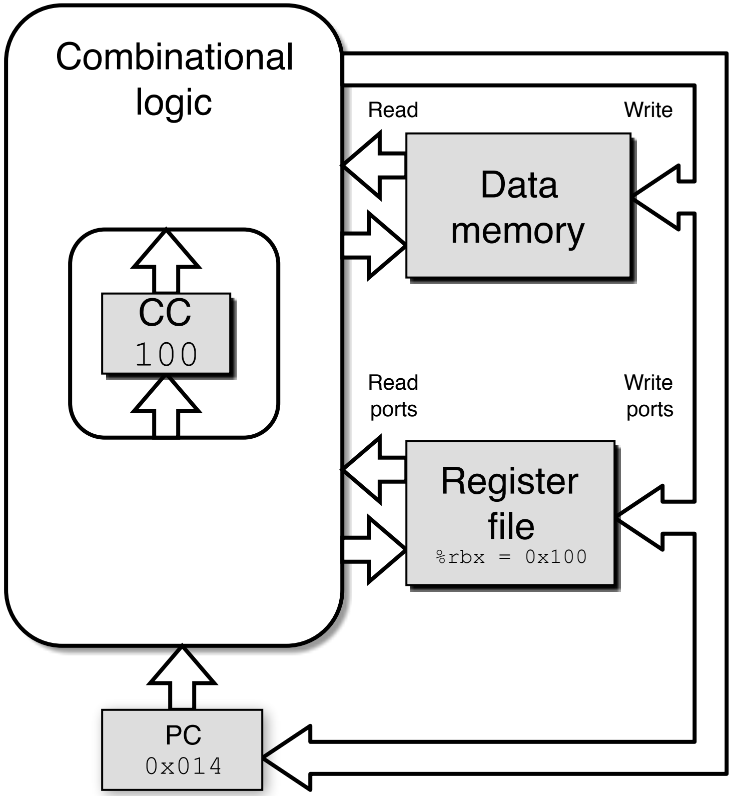
States (All updated as clock rises)

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

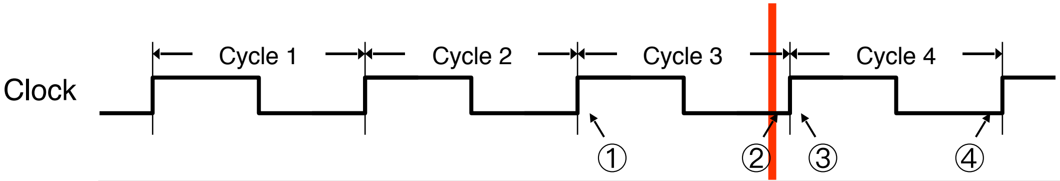




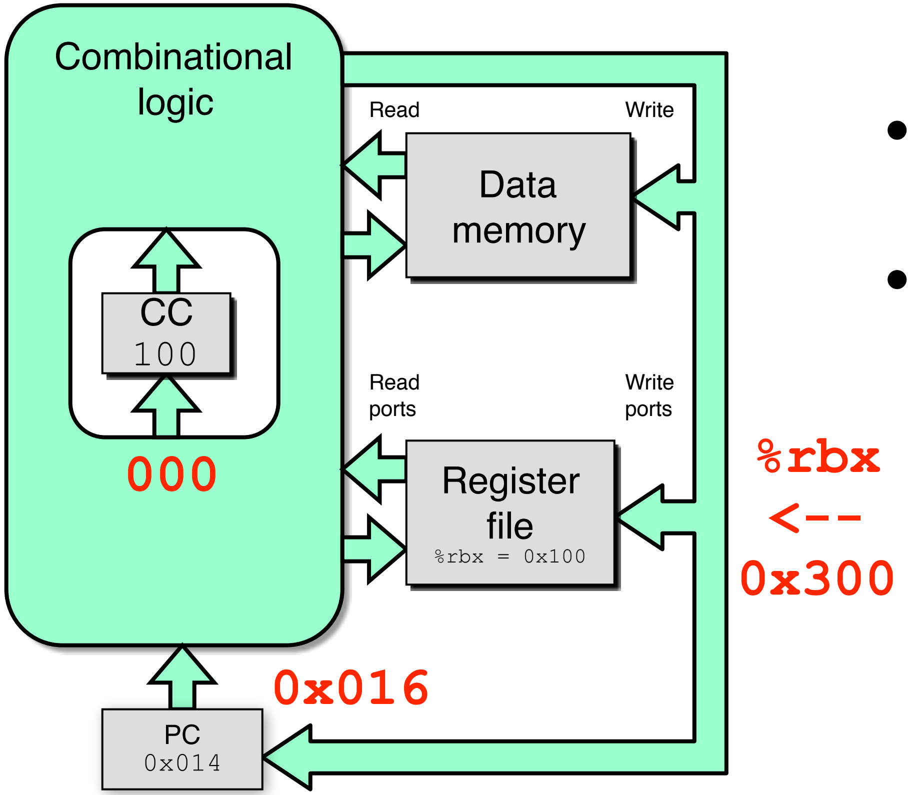
Cycle 1:	`0x000:`	`irmovq $0x100,%rbx`	`# %rbx <-- 0x100`
Cycle 2:	`0x00a:`	`irmovq $0x200,%rdx`	`# %rdx <-- 0x200`
Cycle 3:	`0x014:`	`addq %rdx,%rbx`	`# %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:`	`je dest`	`# Not taken`
Cycle 5:	`0x01f:`	`rmmovq %rbx,0(%rdx)`	`# M[0x200] <-- 0x300`



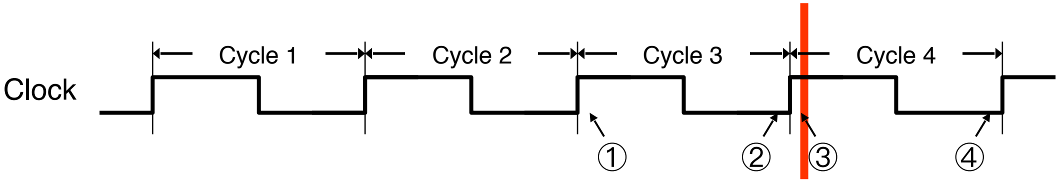
- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes



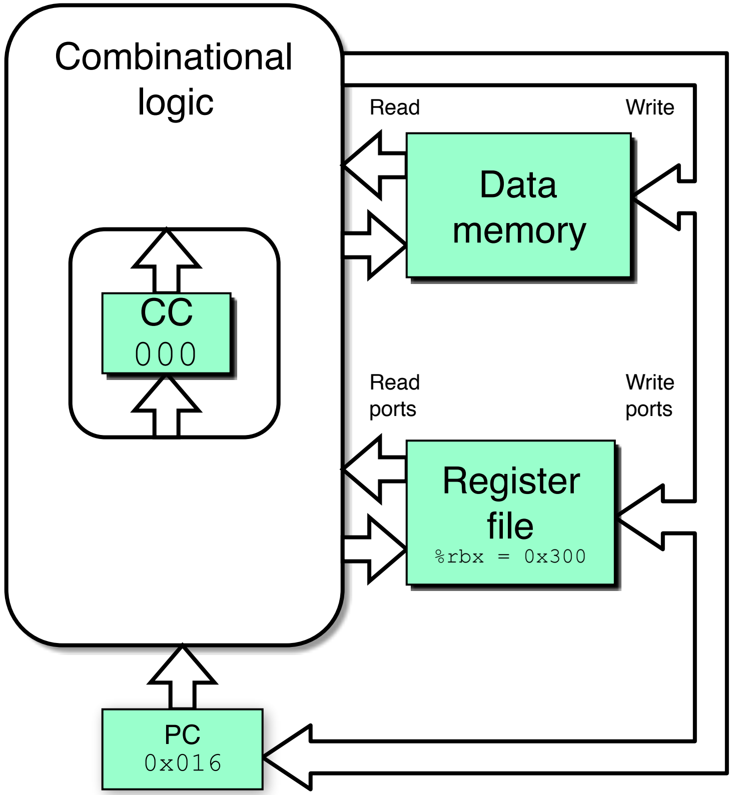
Cycle 1:	`0x000:`	`irmovq $0x100,%rbx`	`# %rbx <-- 0x100`
Cycle 2:	`0x00a:`	`irmovq $0x200,%rdx`	`# %rdx <-- 0x200`
Cycle 3:	`0x014:`	`addq %rdx,%rbx`	`# %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:`	`je dest`	`# Not taken`
Cycle 5:	`0x01f:`	`rmmovq %rbx,0(%rdx)`	`# M[0x200] <-- 0x300`



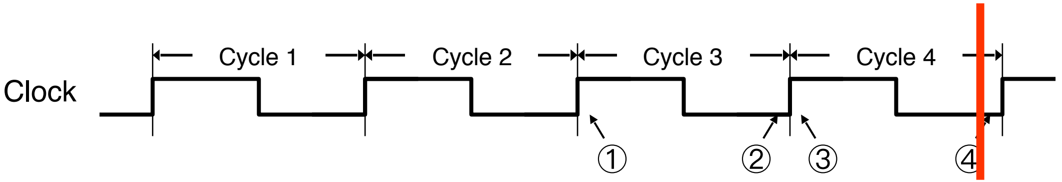
- state set according to second `irmovq` instruction
- combinational logic generates results for addq instruction



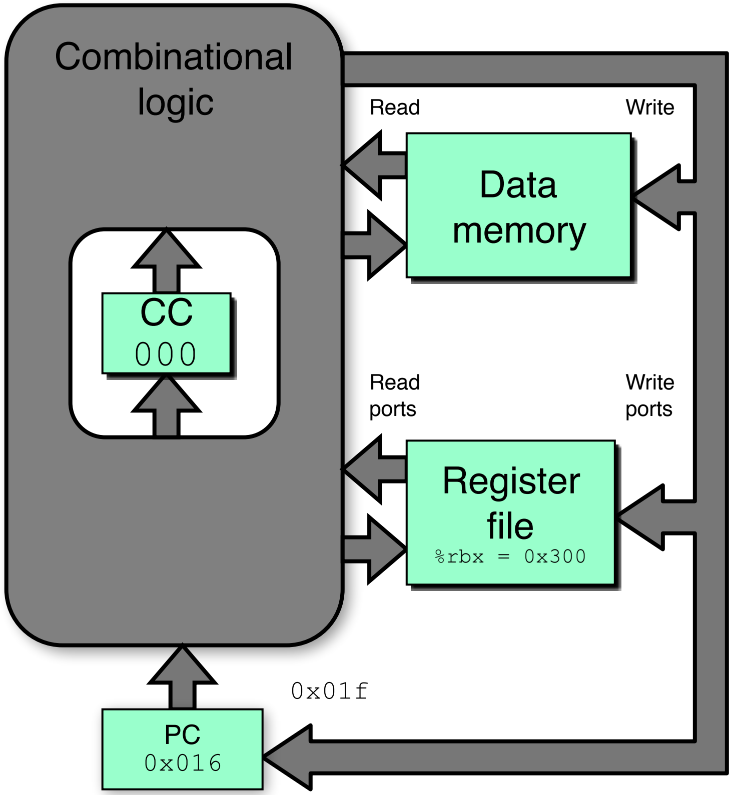
Cycle 1:	`0x000:`	`irmovq $0x100,%rbx`	`# %rbx <-- 0x100`
Cycle 2:	`0x00a:`	`irmovq $0x200,%rdx`	`# %rdx <-- 0x200`
Cycle 3:	`0x014:`	`addq %rdx,%rbx`	`# %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:`	`je dest`	`# Not taken`
Cycle 5:	`0x01f:`	`rmmovq %rbx,0(%rdx)`	`# M[0x200] <-- 0x300`



- 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