CSC 252: Computer Organization
Spring 2023: Lecture 13

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

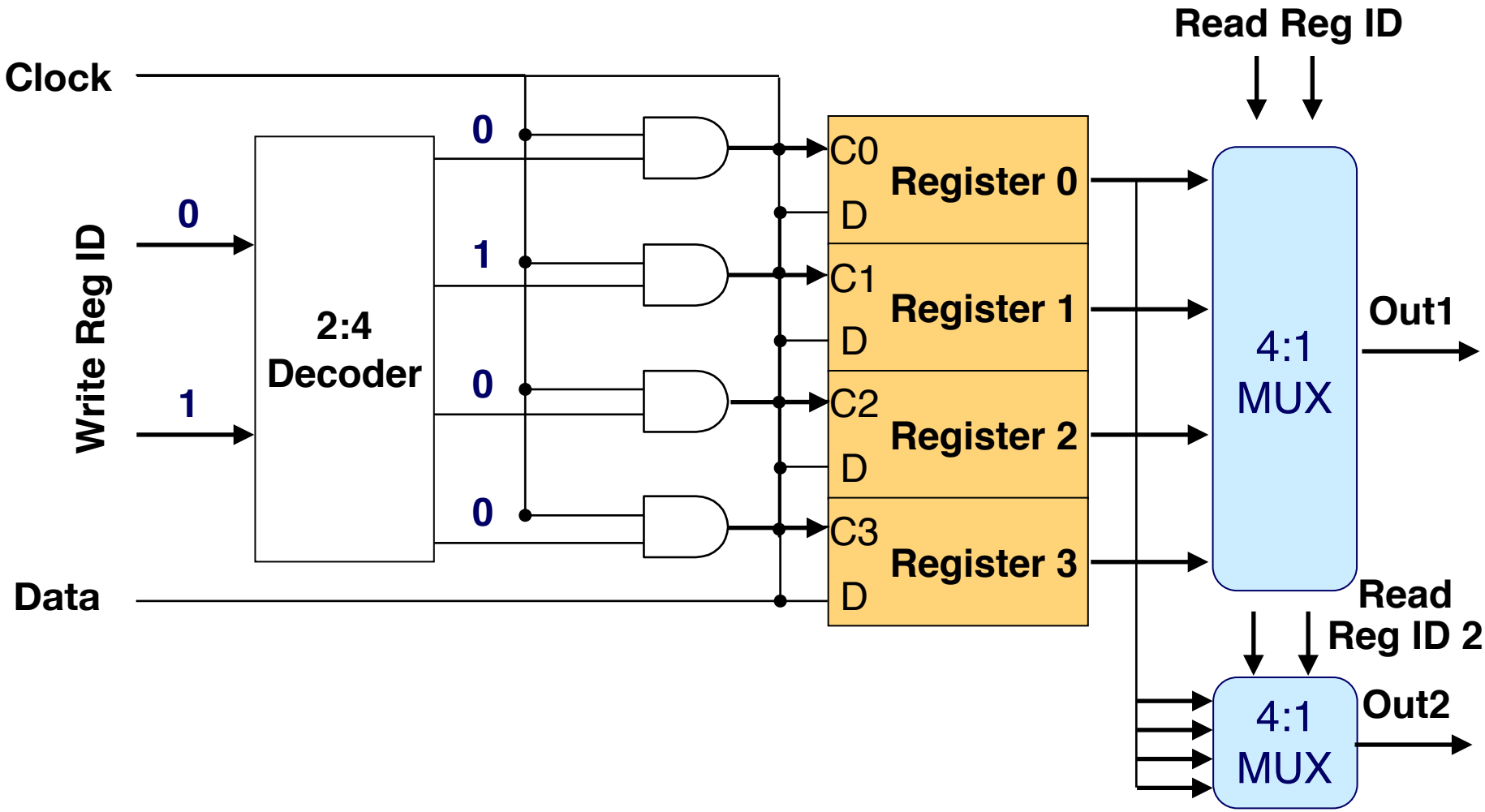
Announcement

- Programming assignment 3 out.
- If you don’t see your lab2 score on the scoreboard, talk to a TA.

12	13	14	15	16	17	18
19	20	21	22 **Today**	23	24	25
26	27	28	Mar 1 **Due**	2	3 **Mid-term**	4

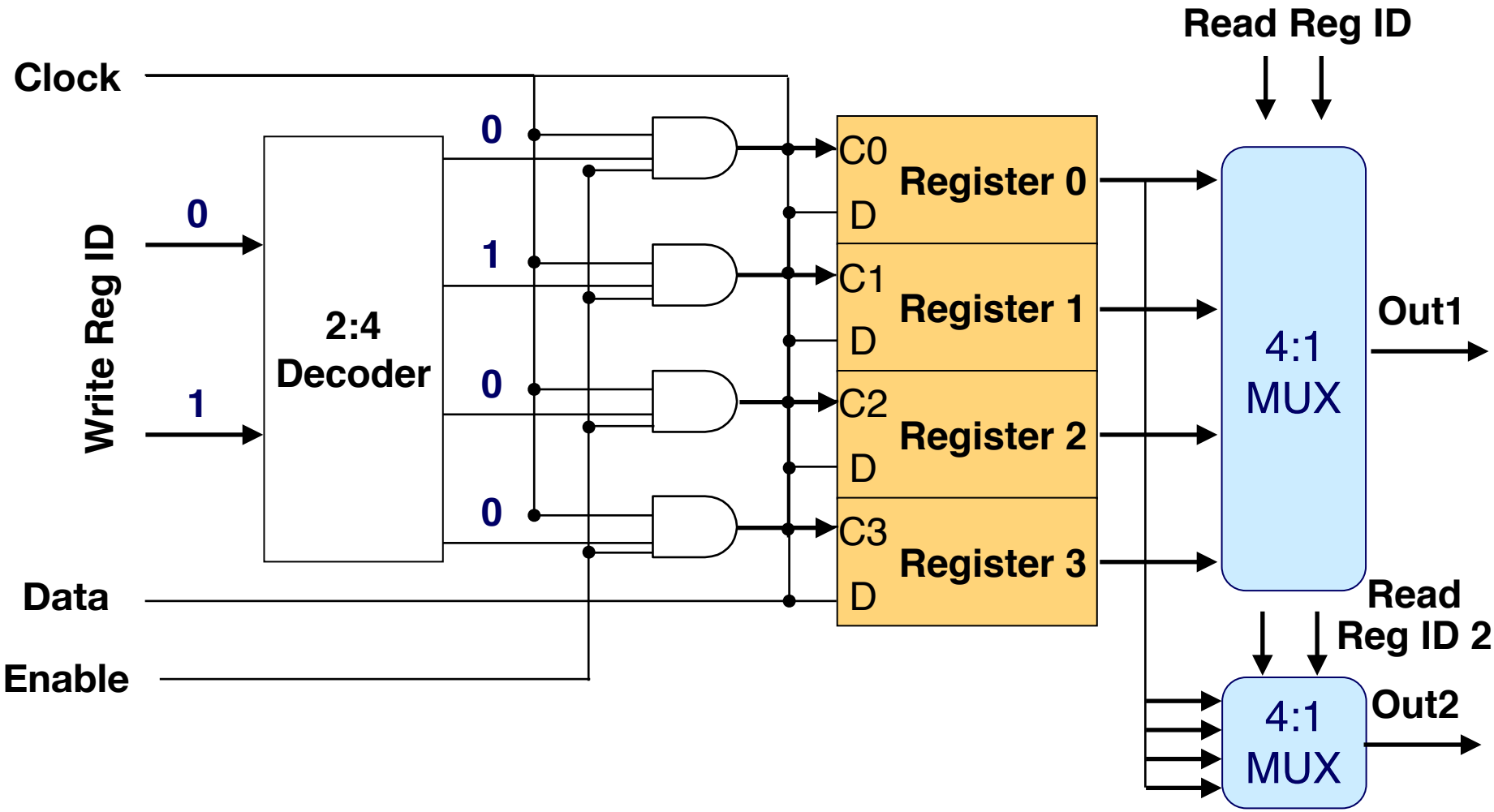
Multi-Port Register File

- 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?

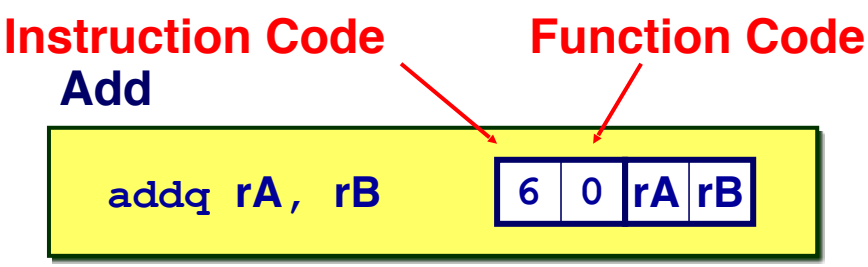


Processor Microarchitecture

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

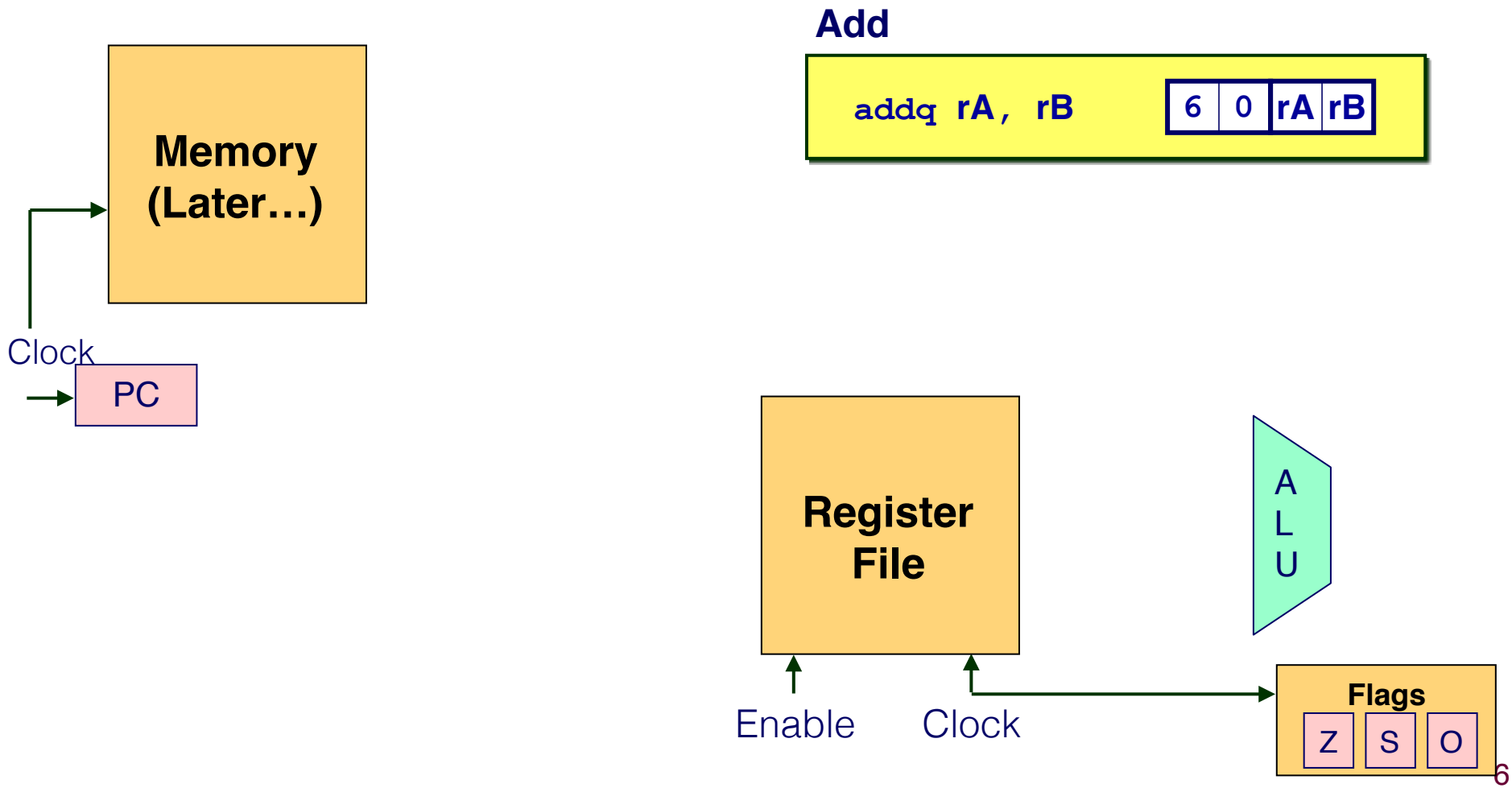
Executing an ADD instruction

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



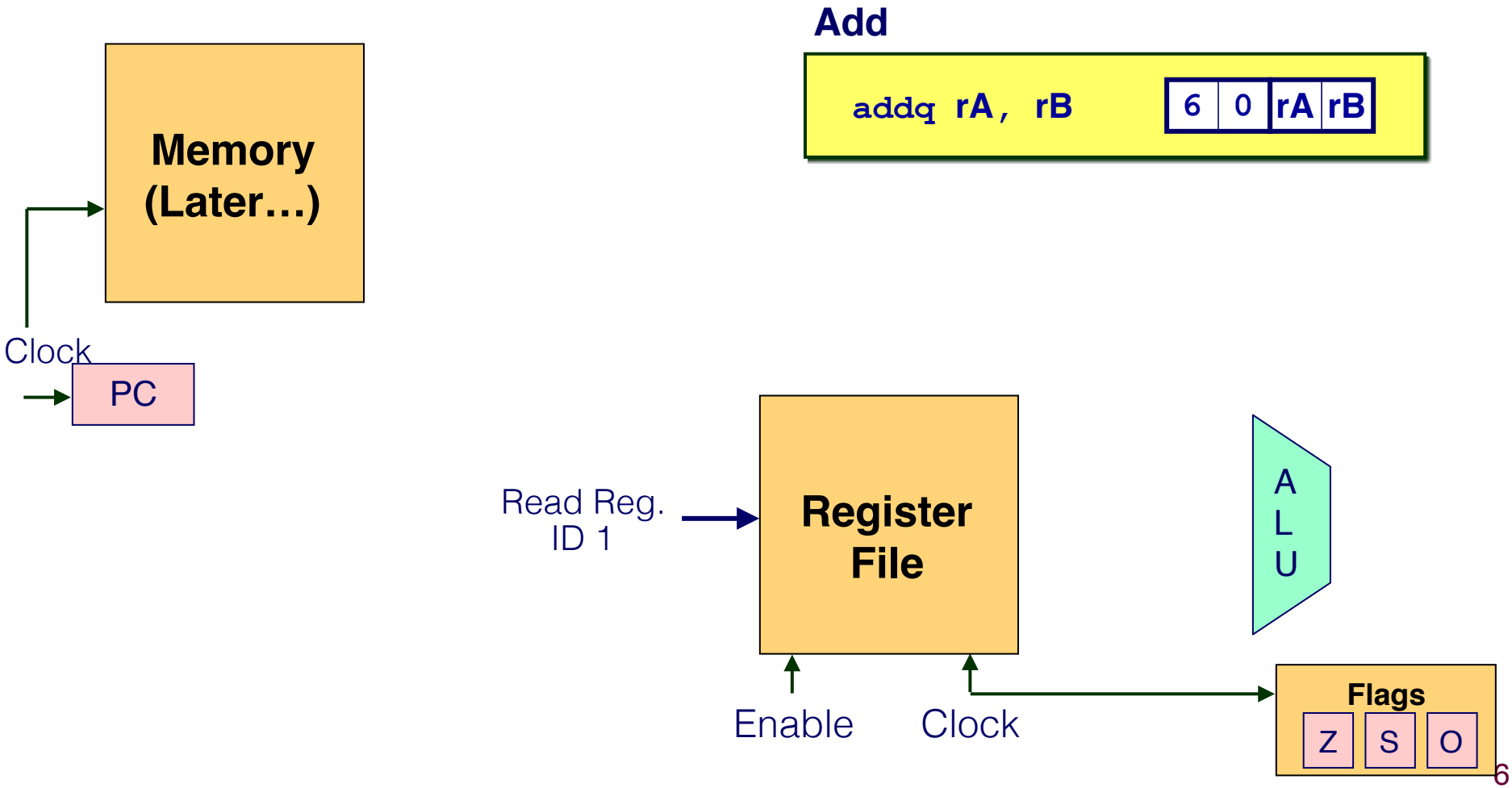
Executing an ADD instruction

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



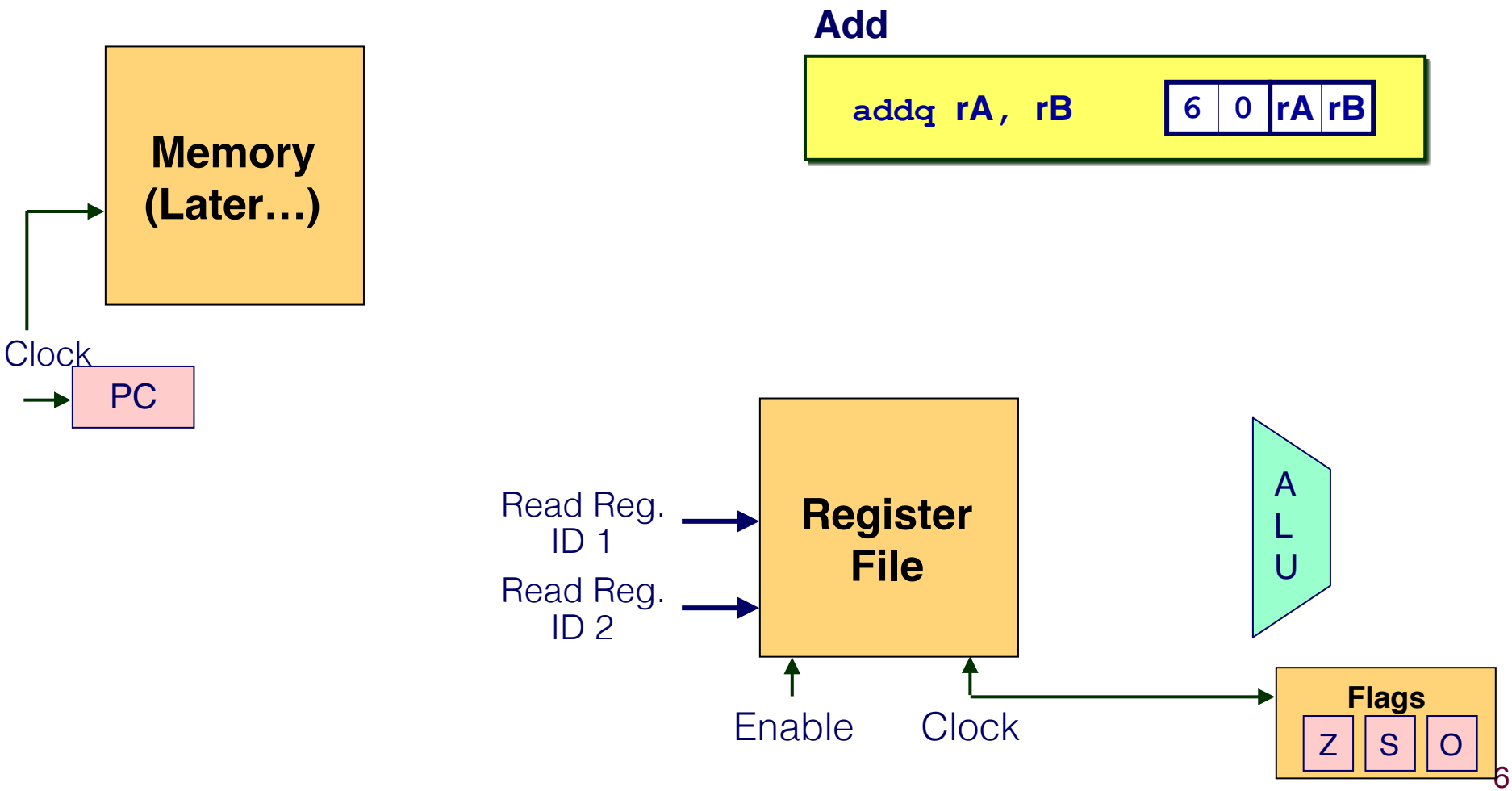
Executing an ADD instruction

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



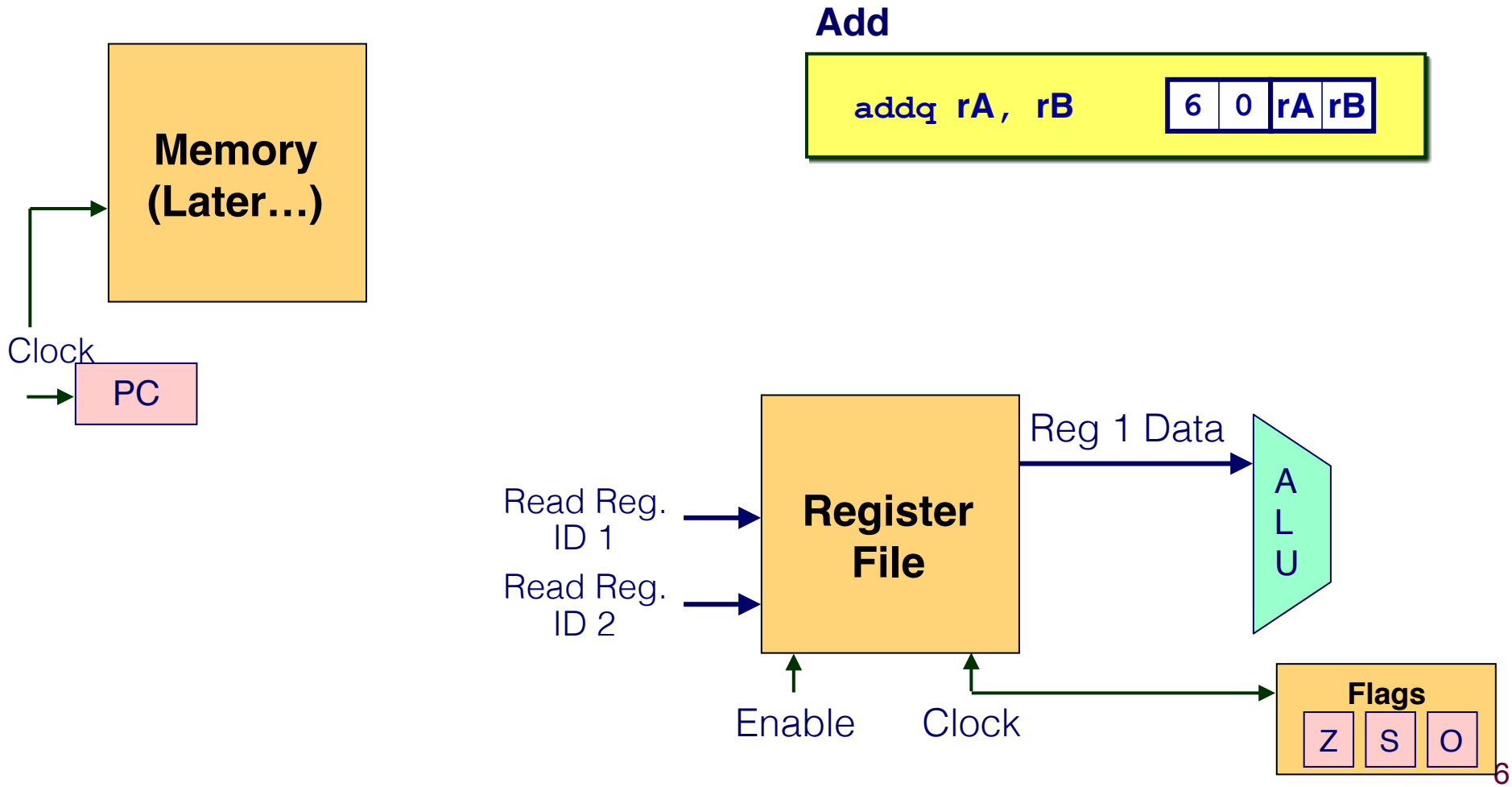
Executing an ADD instruction

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



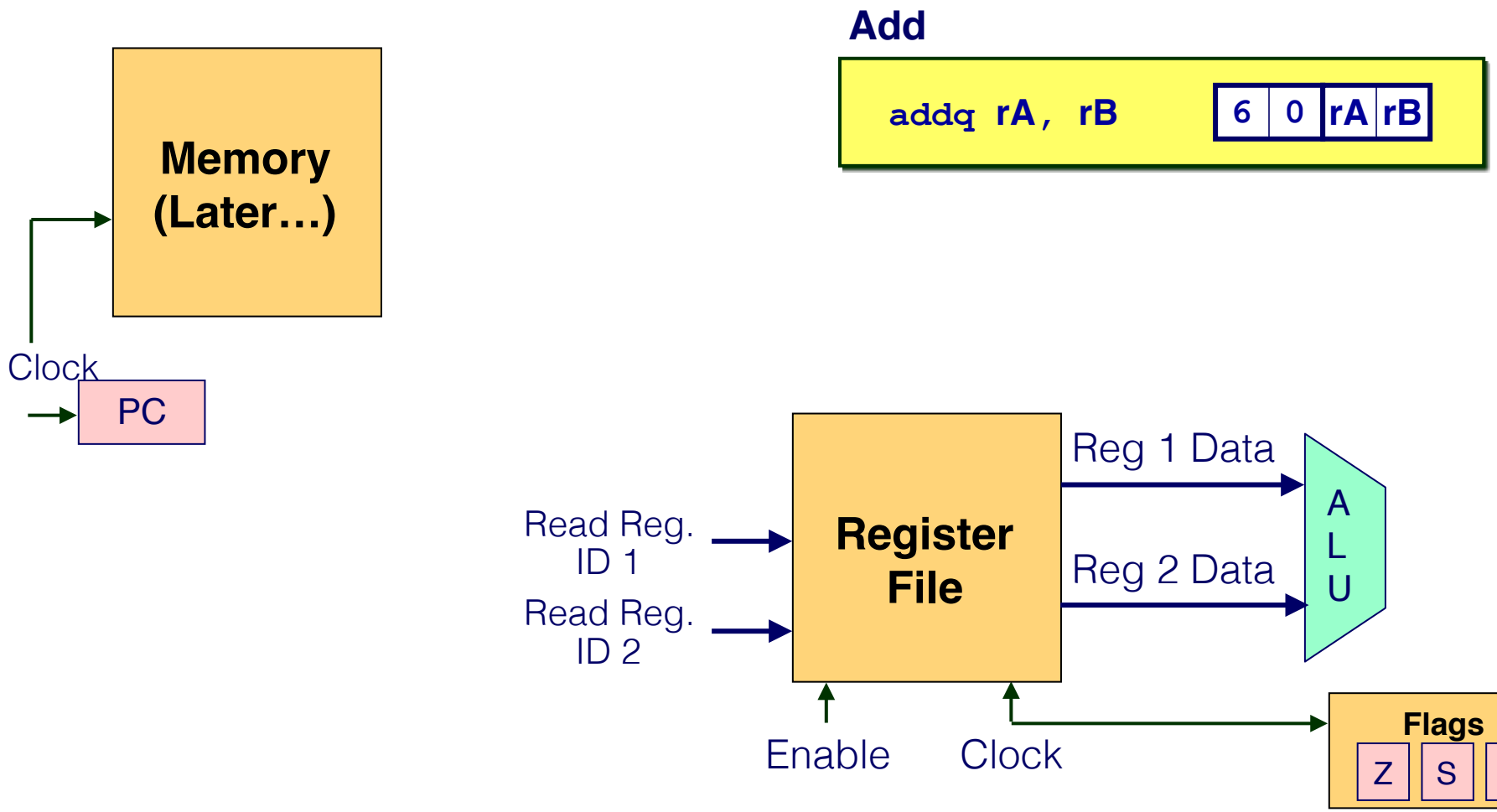
Executing an ADD instruction

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



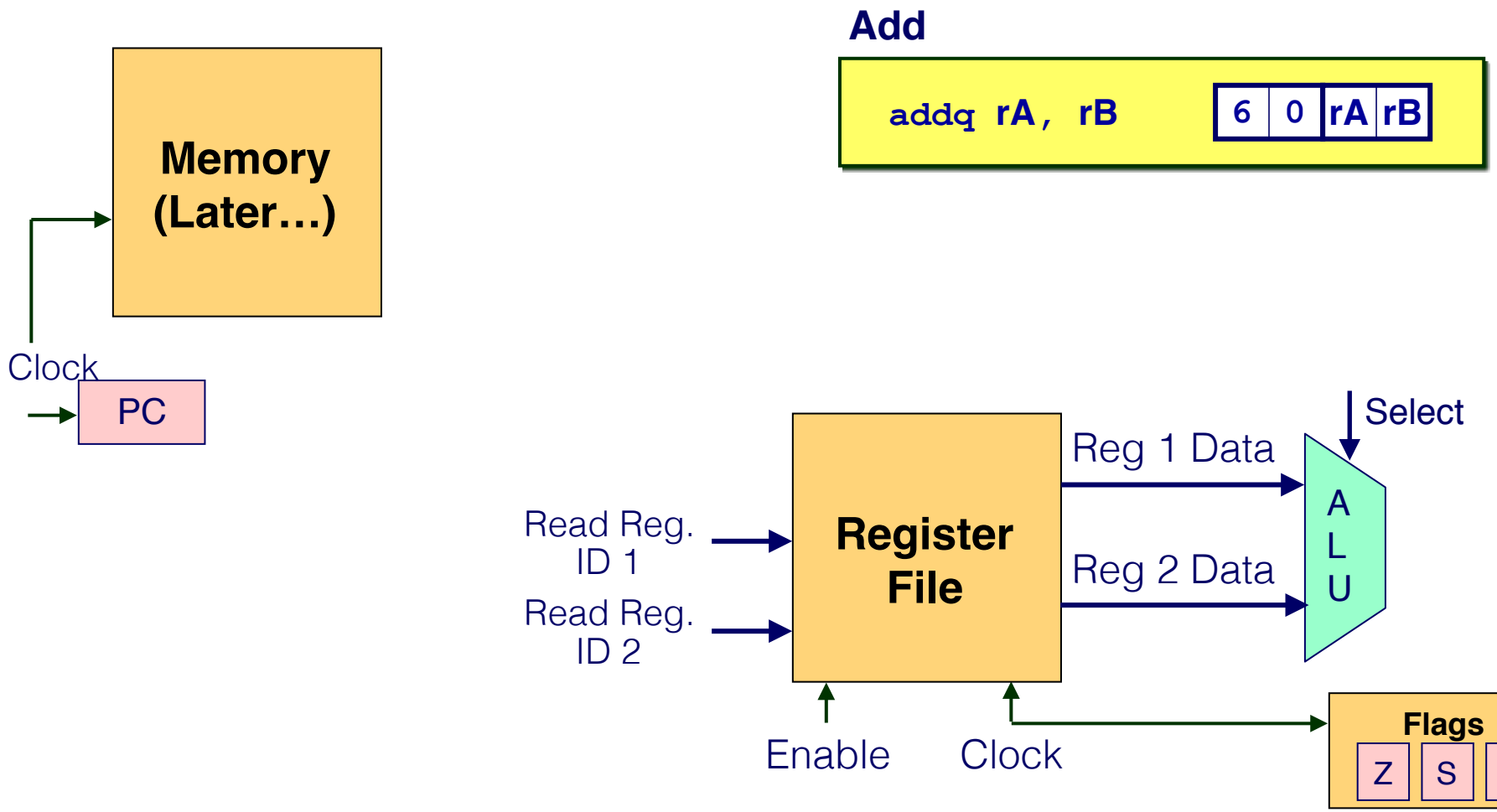
Executing an ADD instruction

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



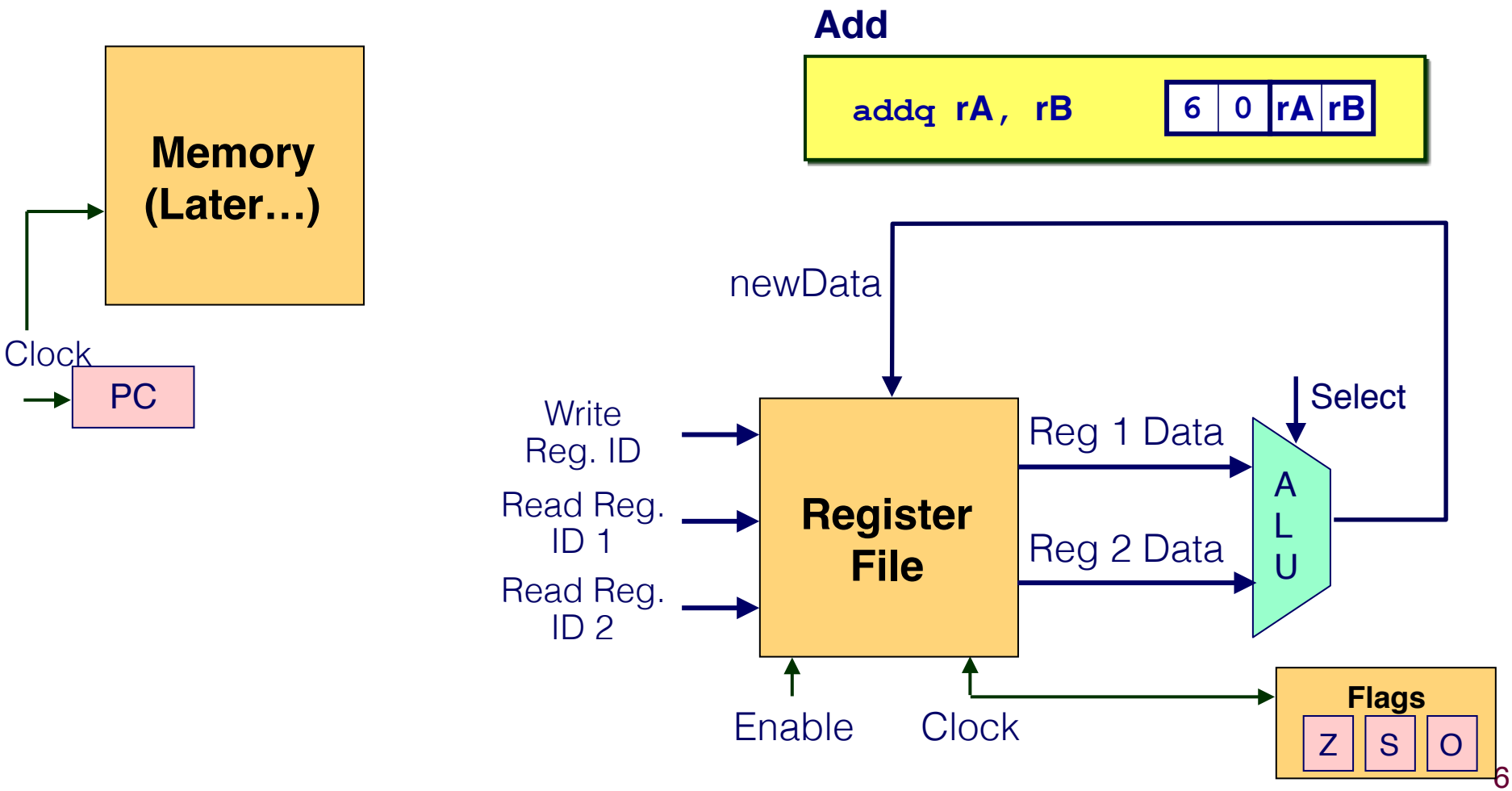
Executing an ADD instruction

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



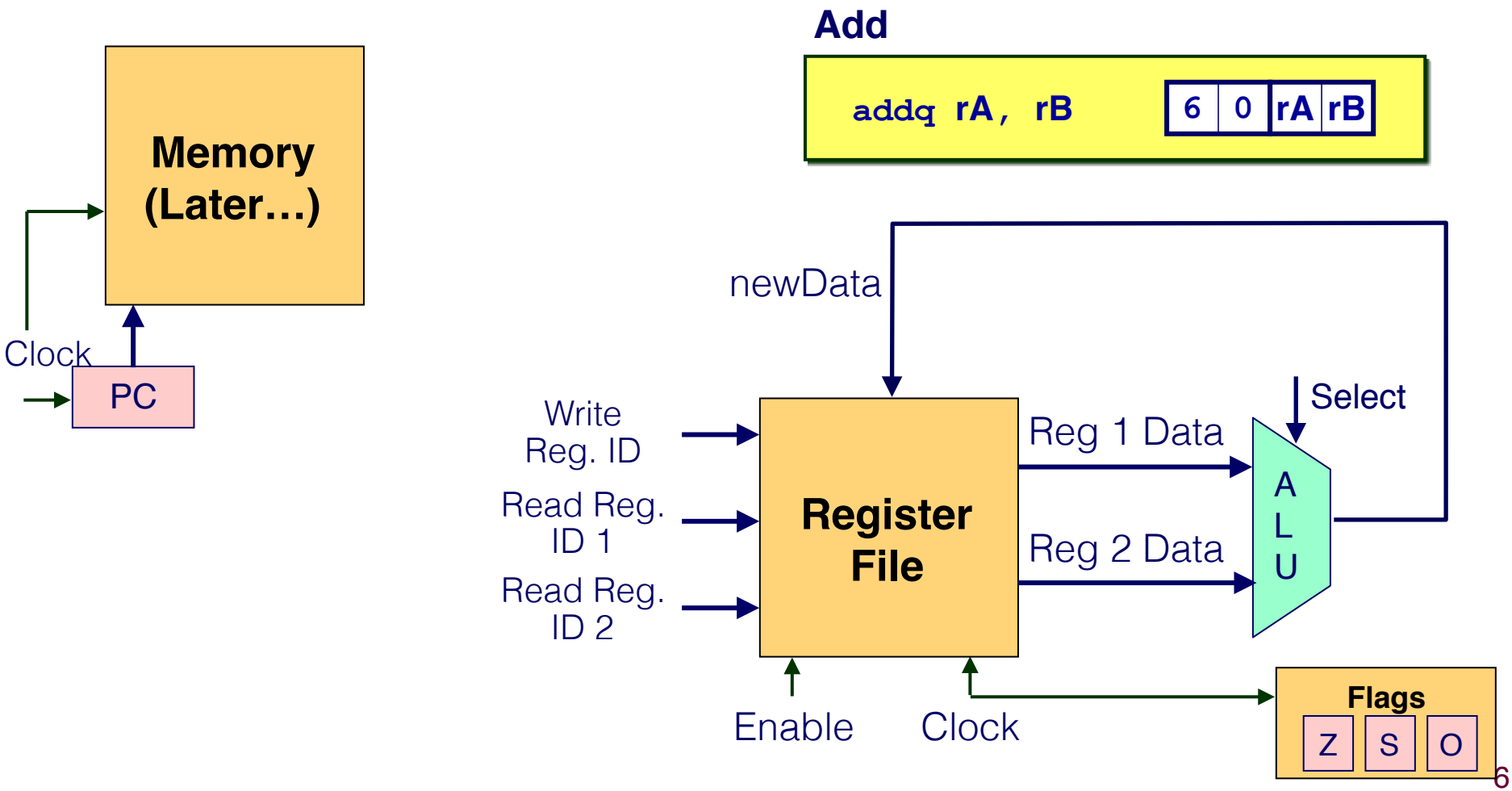
Executing an ADD instruction

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



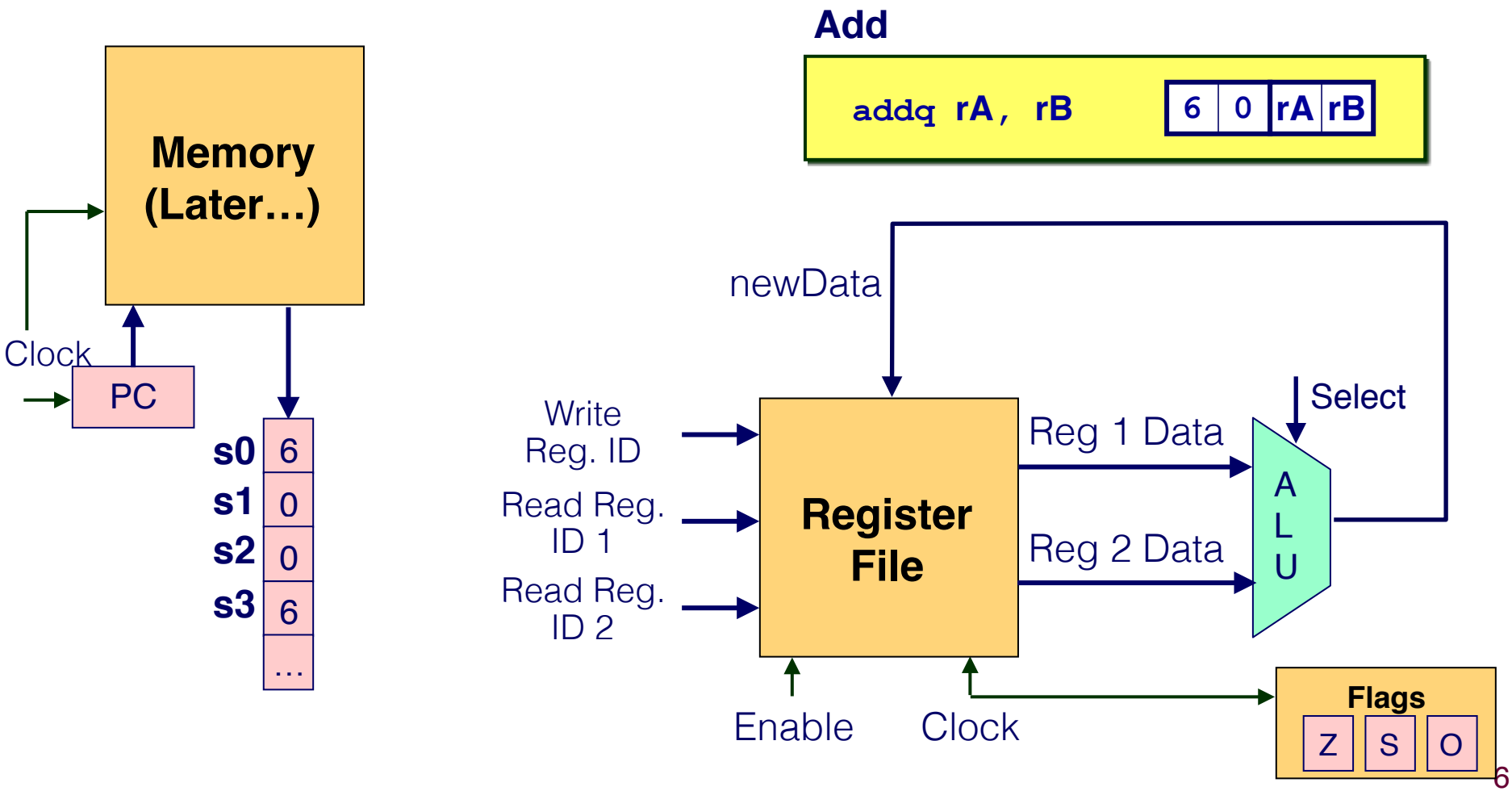
Executing an ADD instruction

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



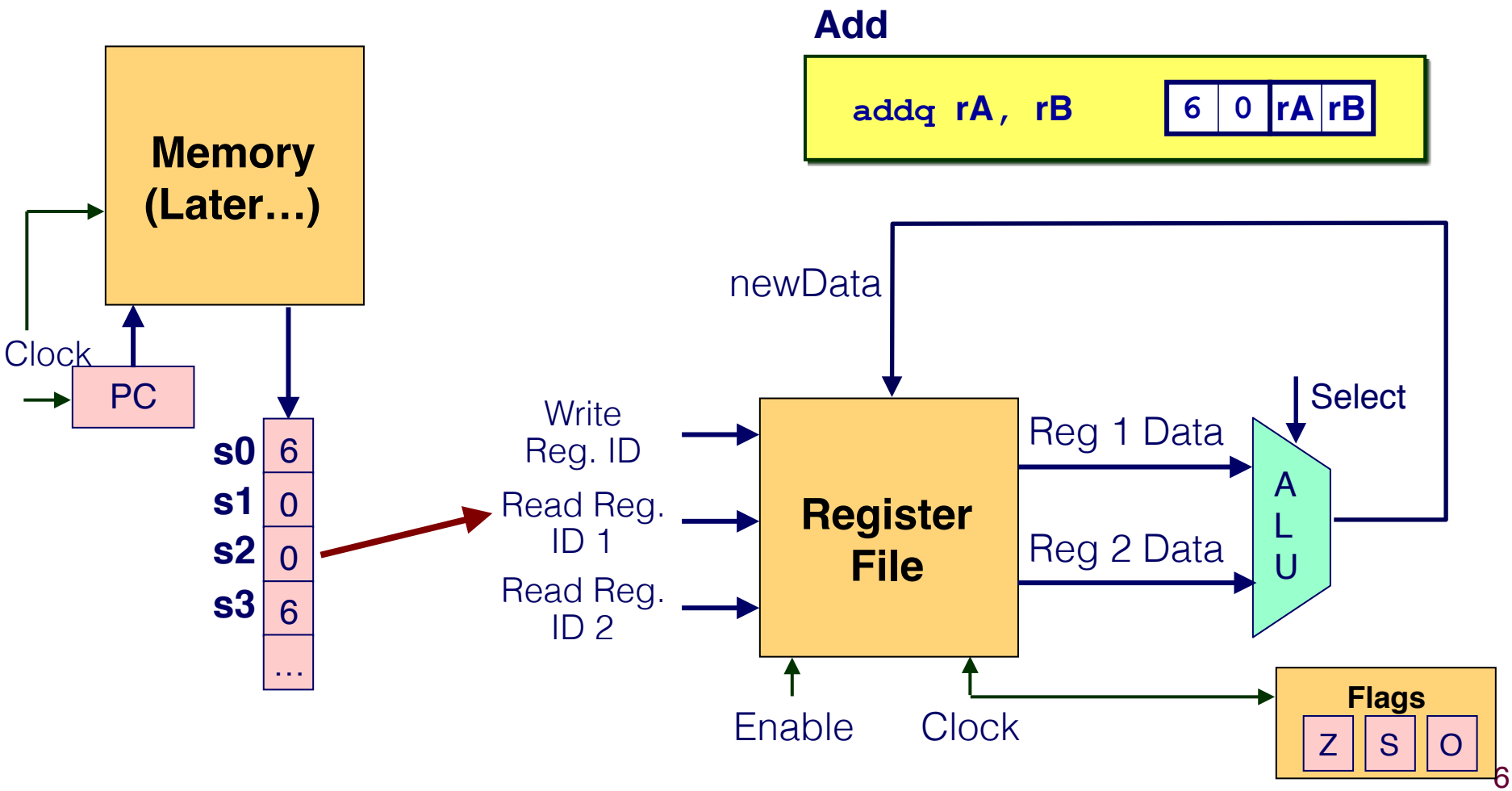
Executing an ADD instruction

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



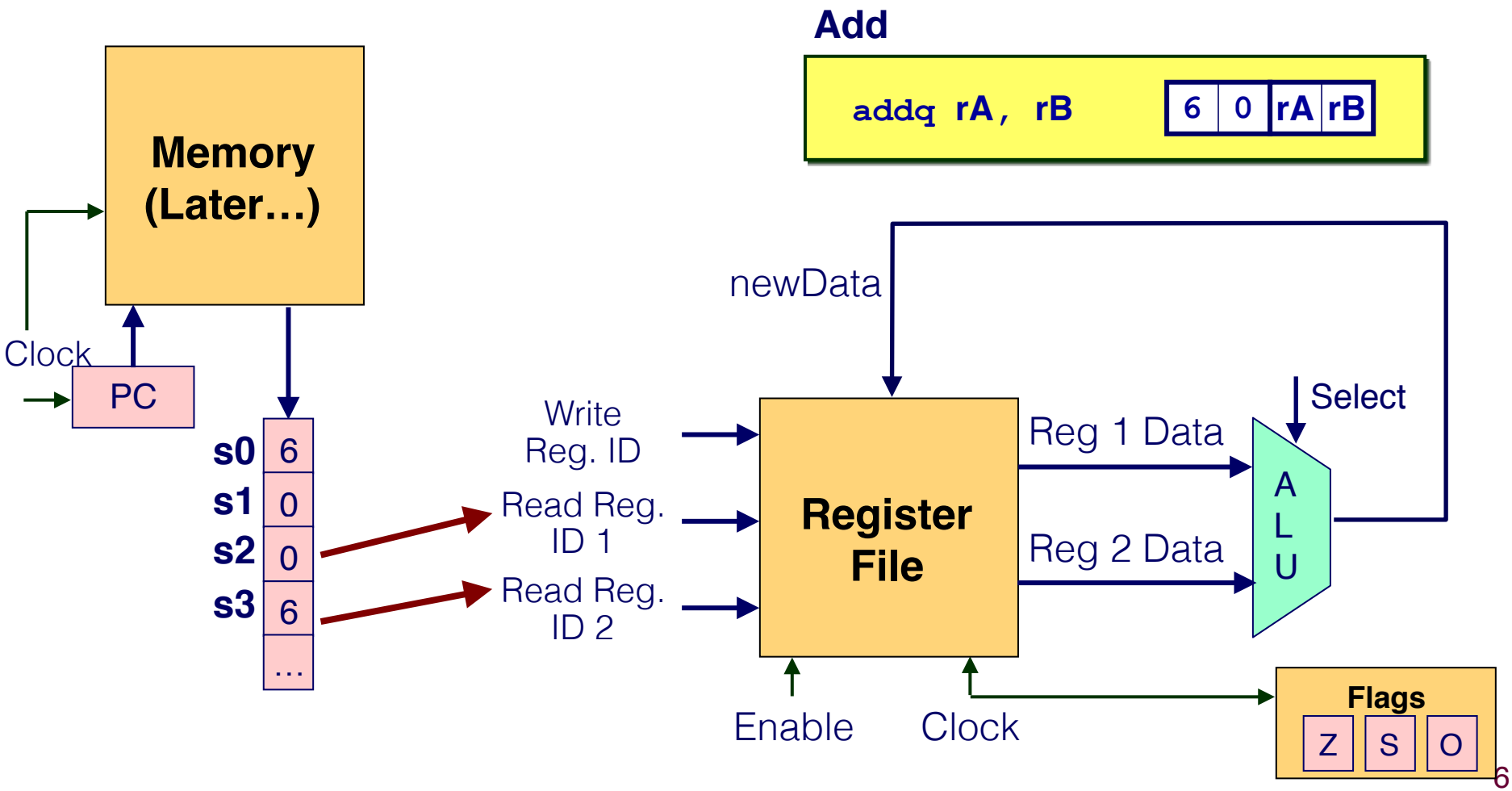
Executing an ADD instruction

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



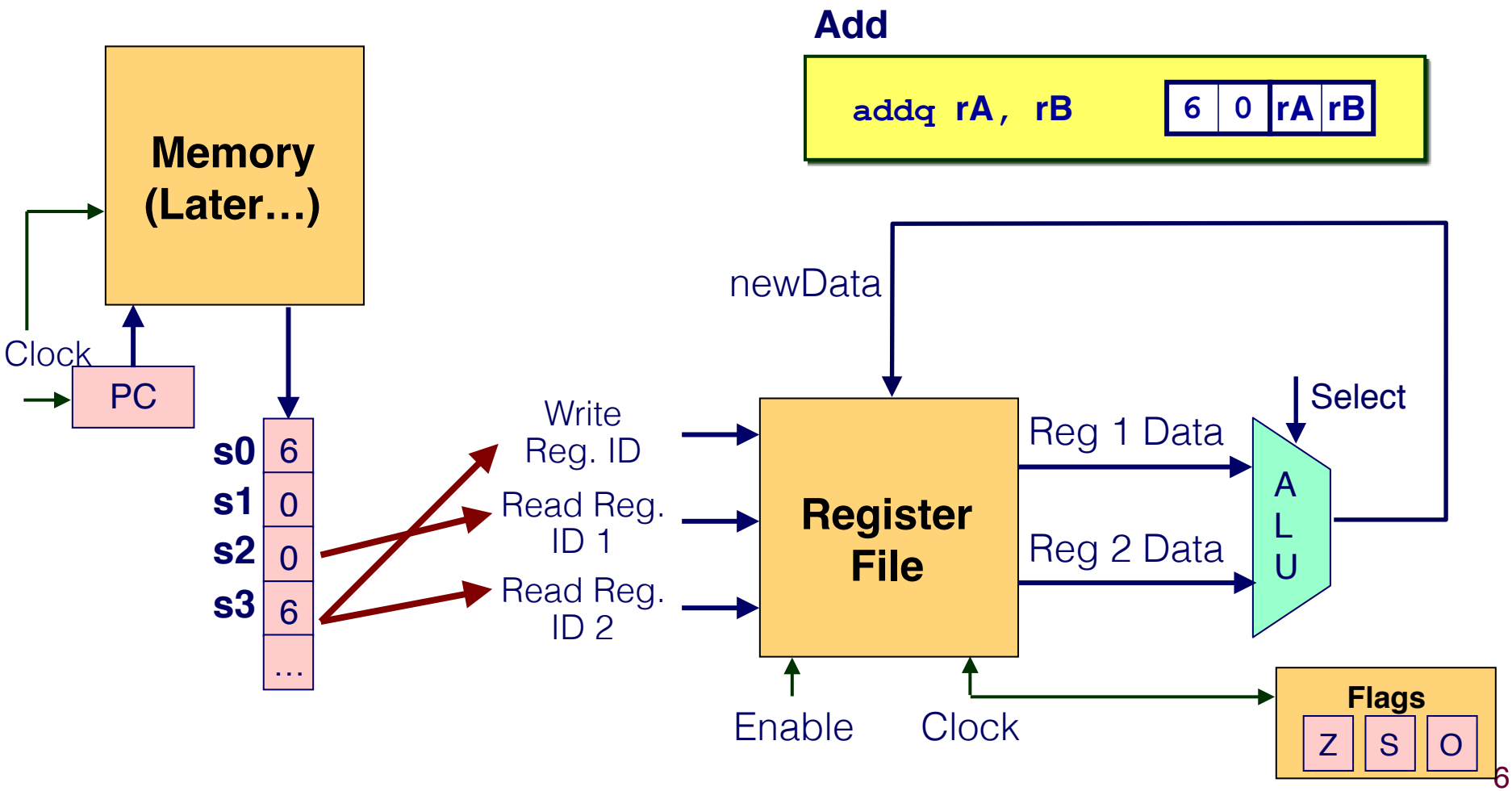
Executing an ADD instruction

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



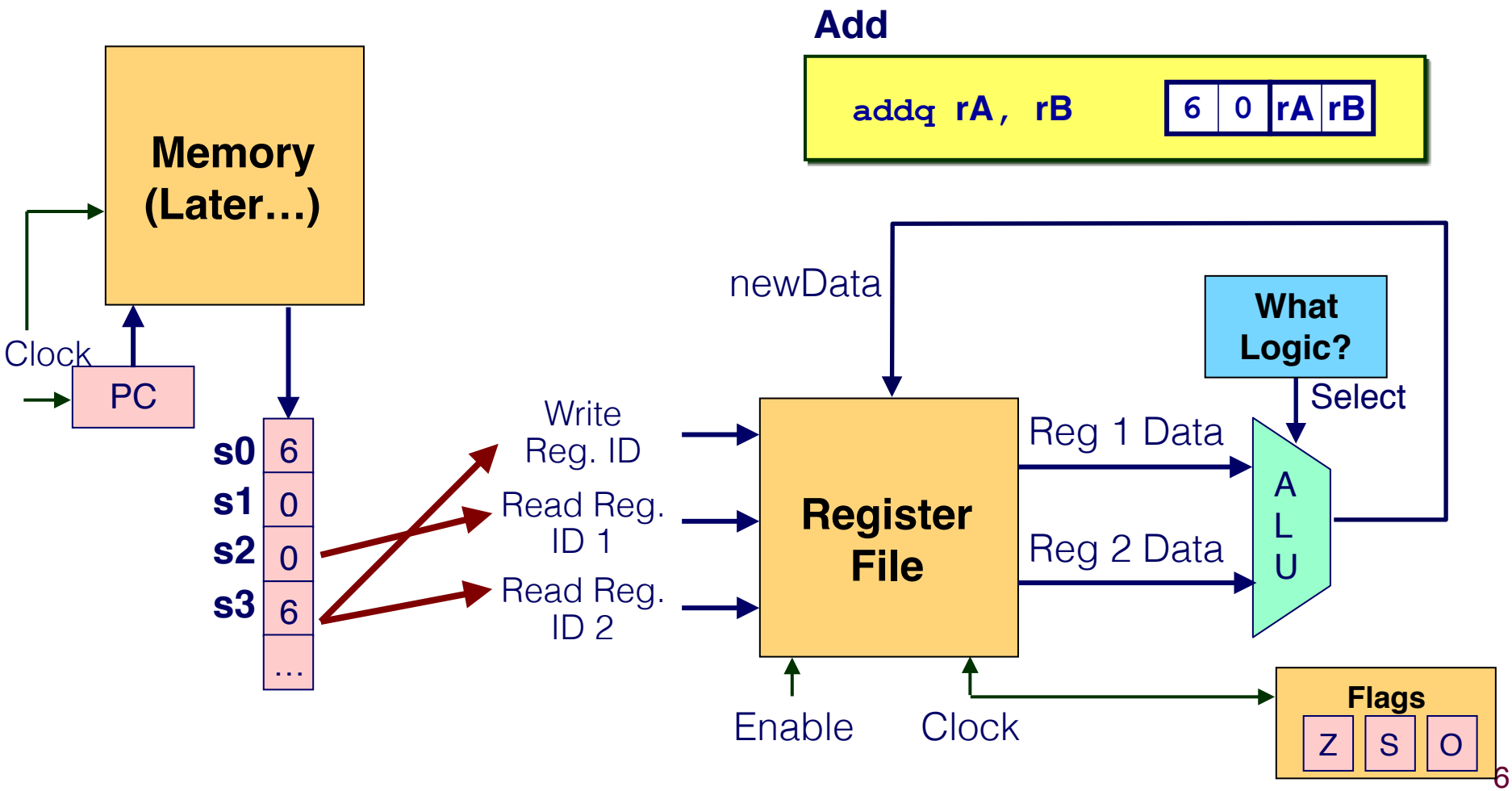
Executing an ADD instruction

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



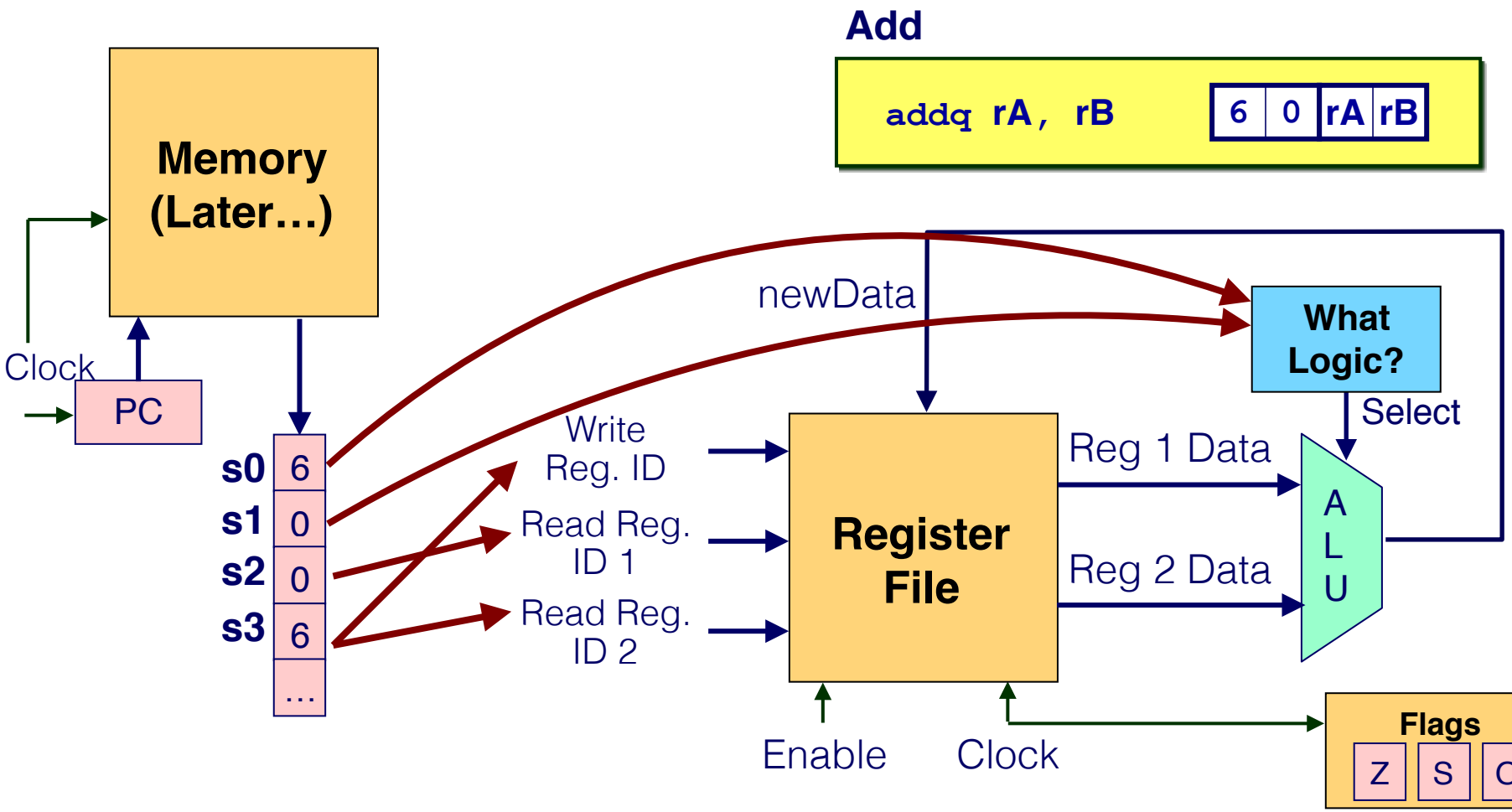
Executing an ADD instruction

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



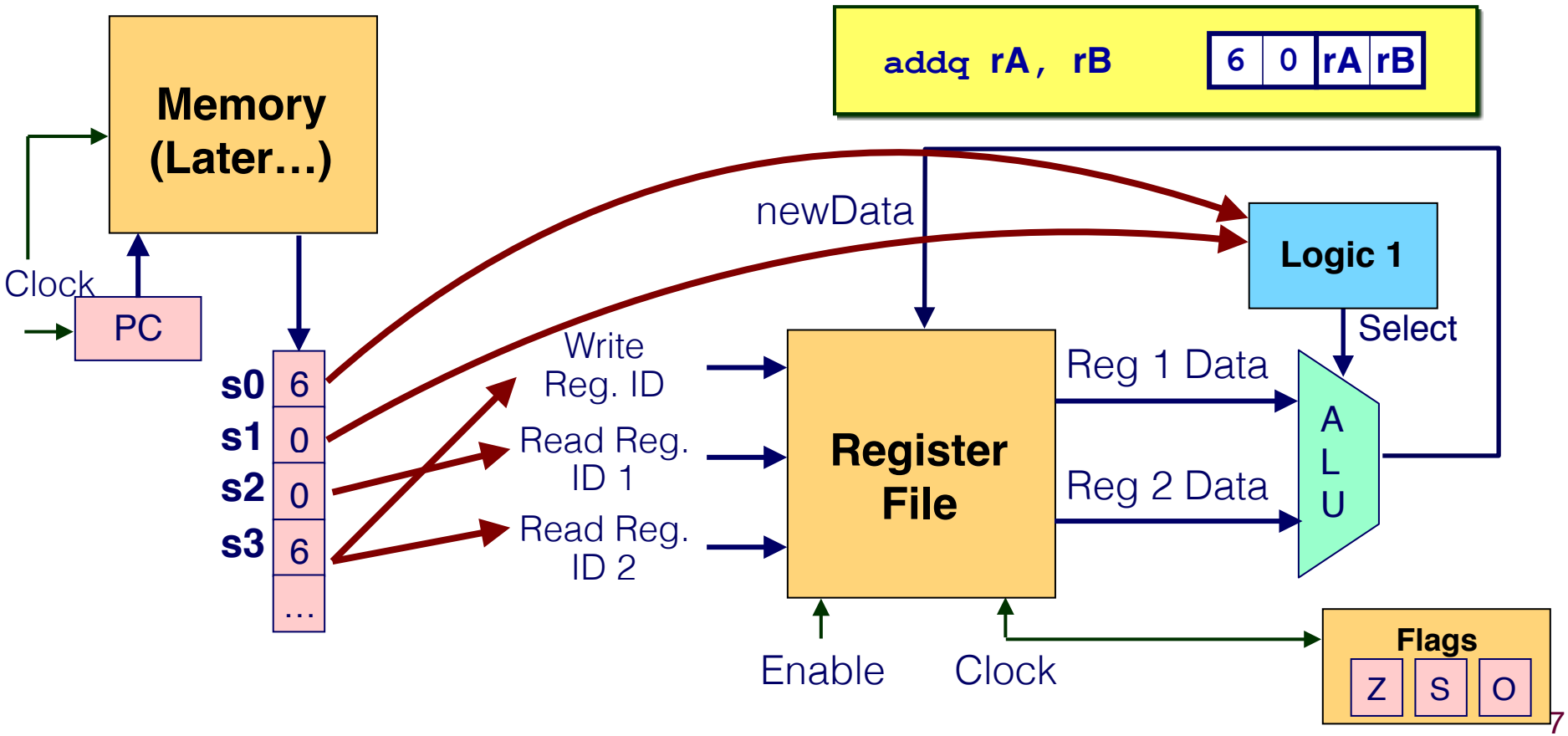
Executing an ADD instruction

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



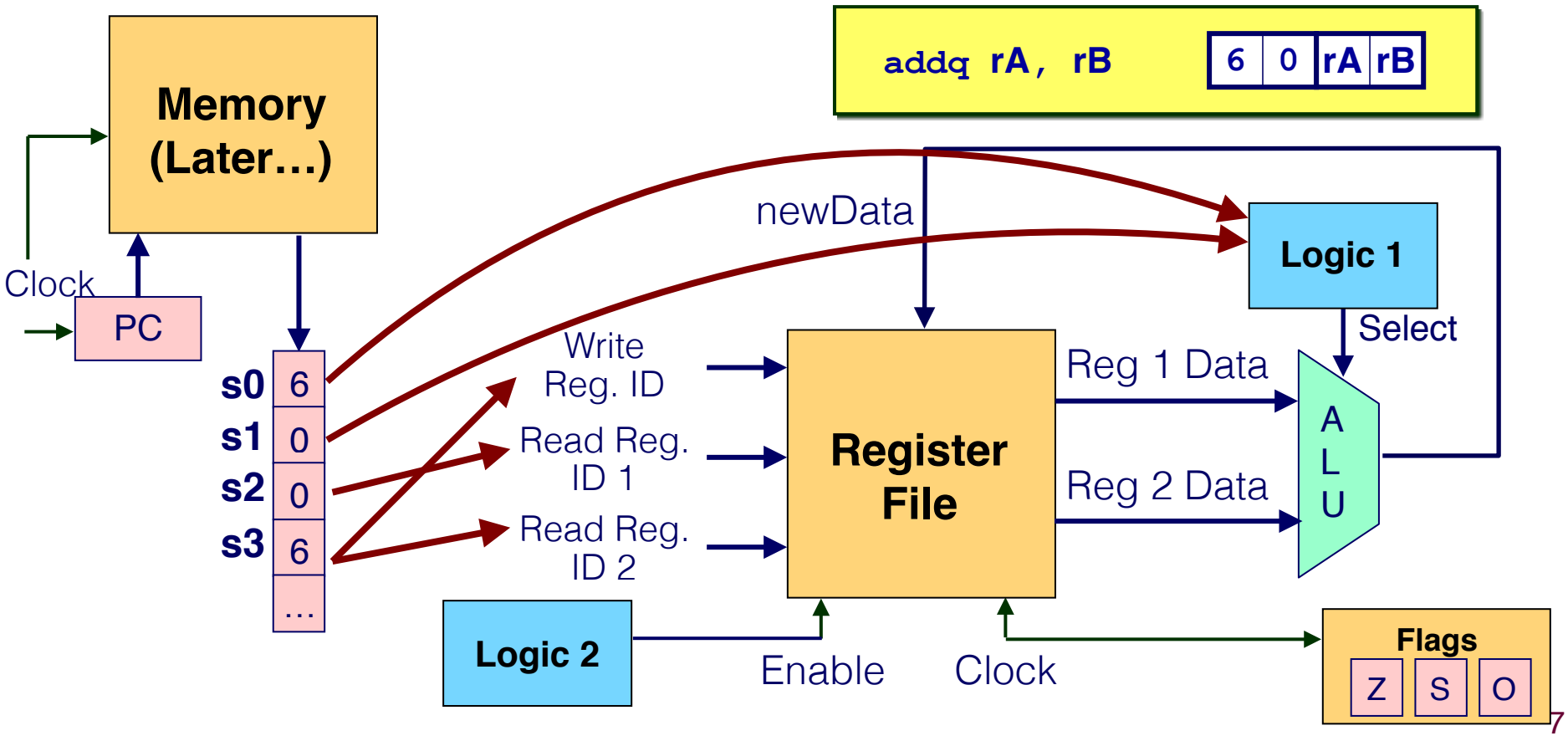
Executing an ADD instruction

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



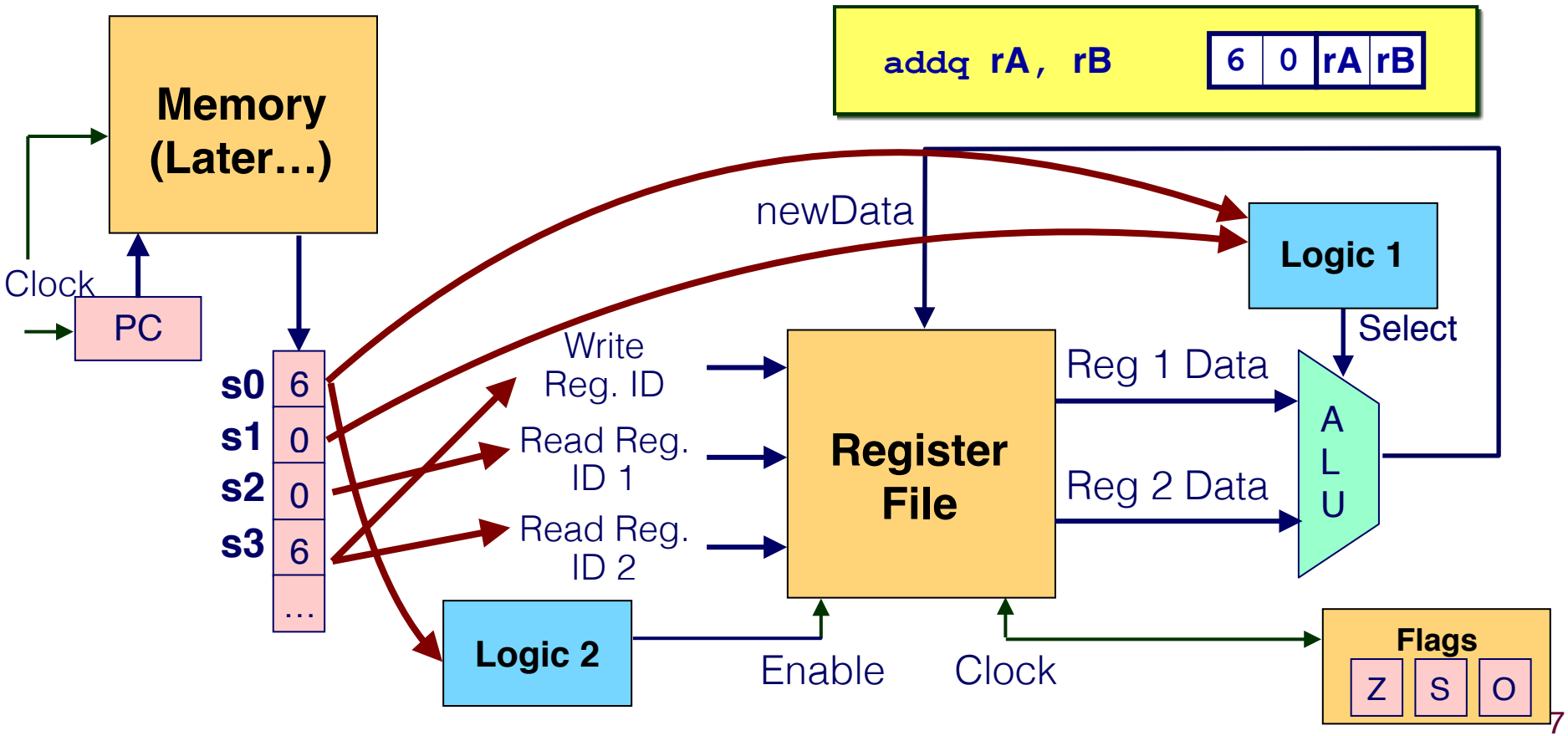
Executing an ADD instruction

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



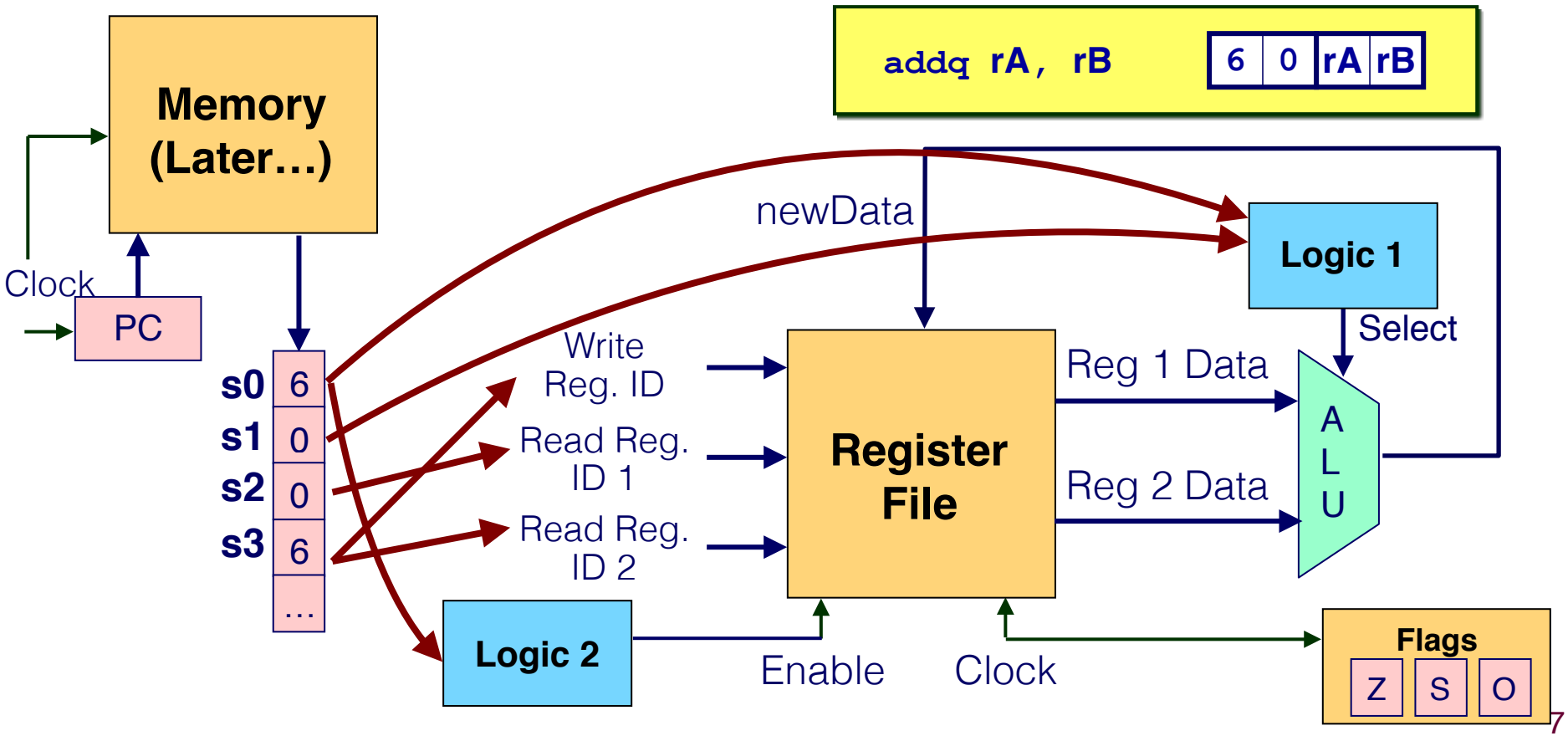
Executing an ADD instruction

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



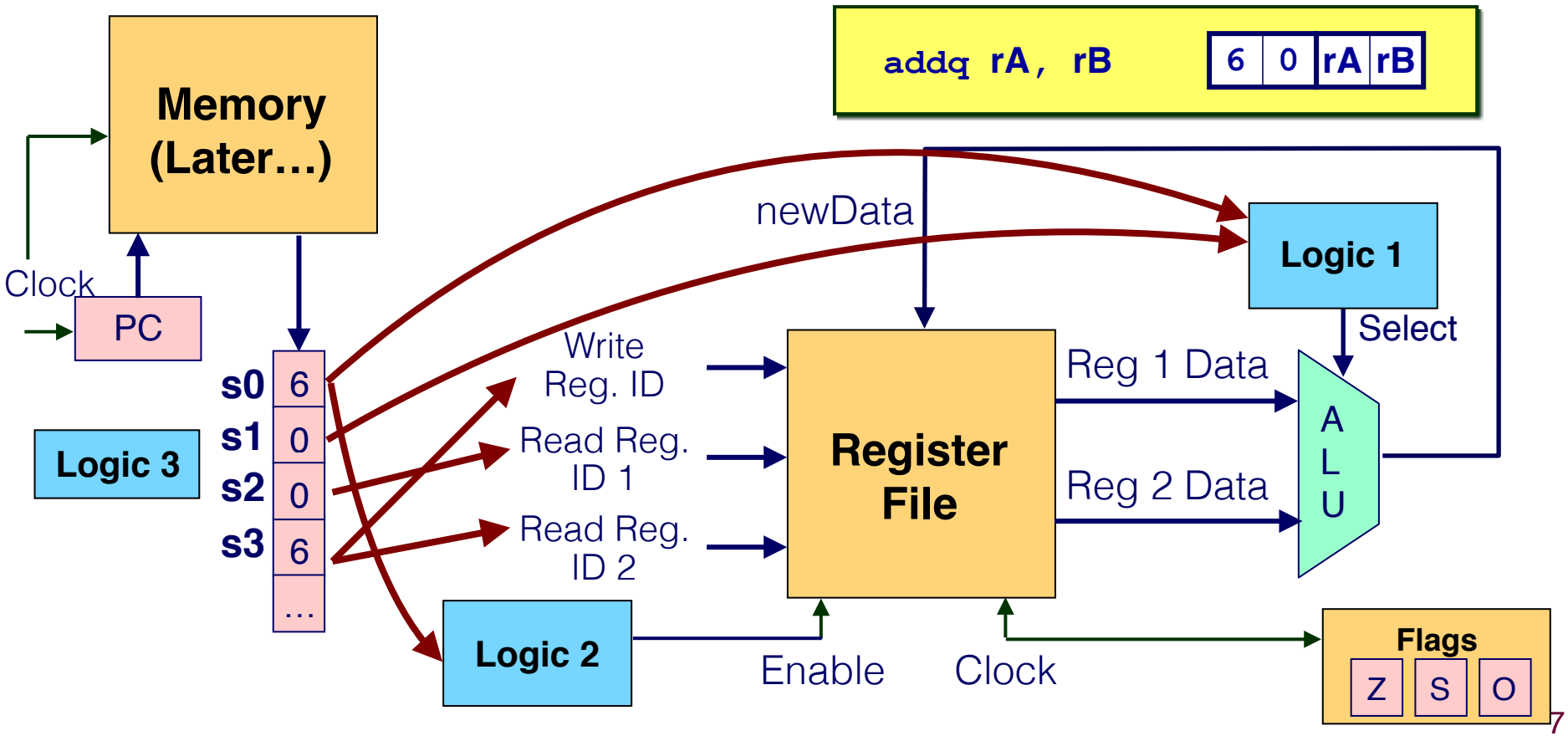
Executing an ADD instruction

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



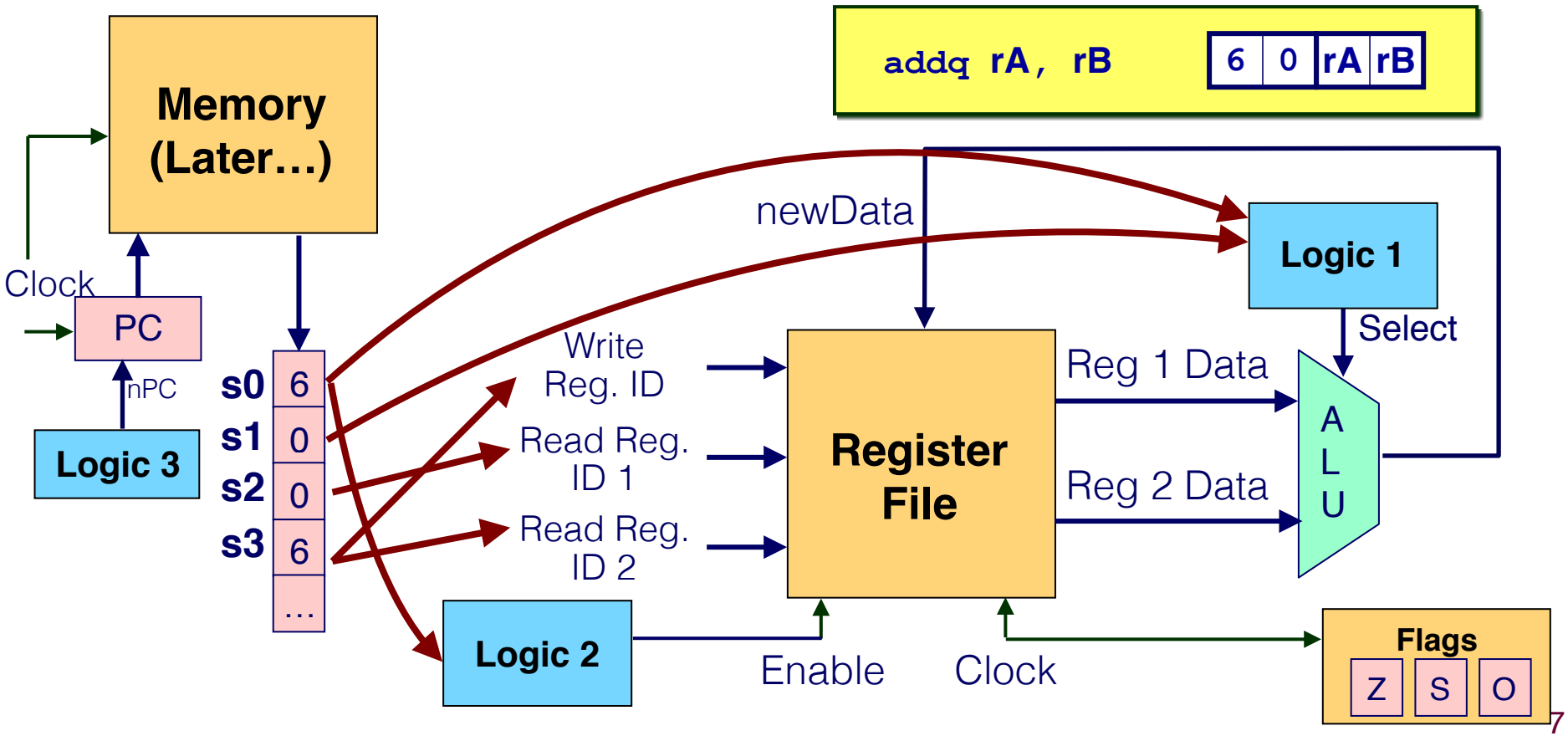
Executing an ADD instruction

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



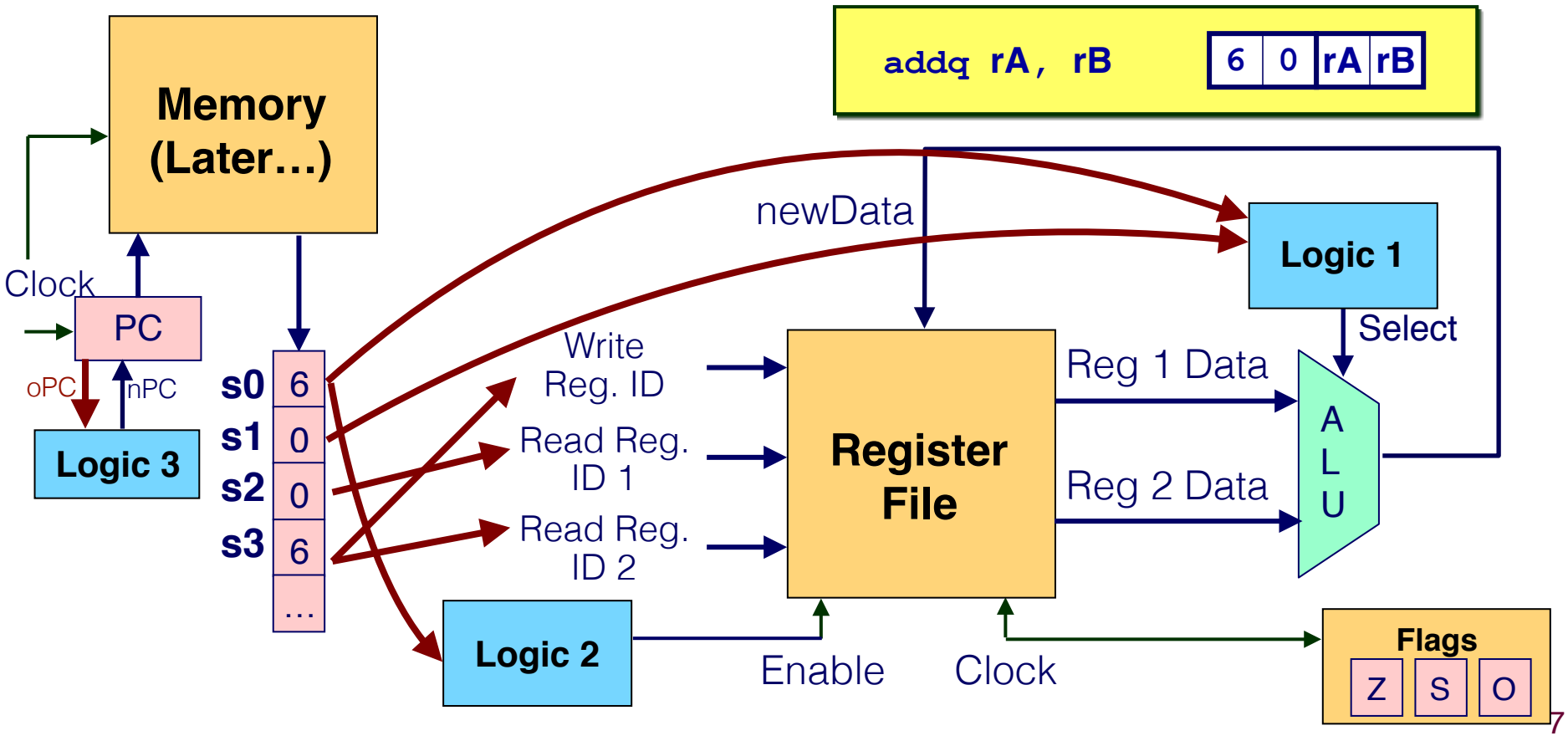
Executing an ADD instruction

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



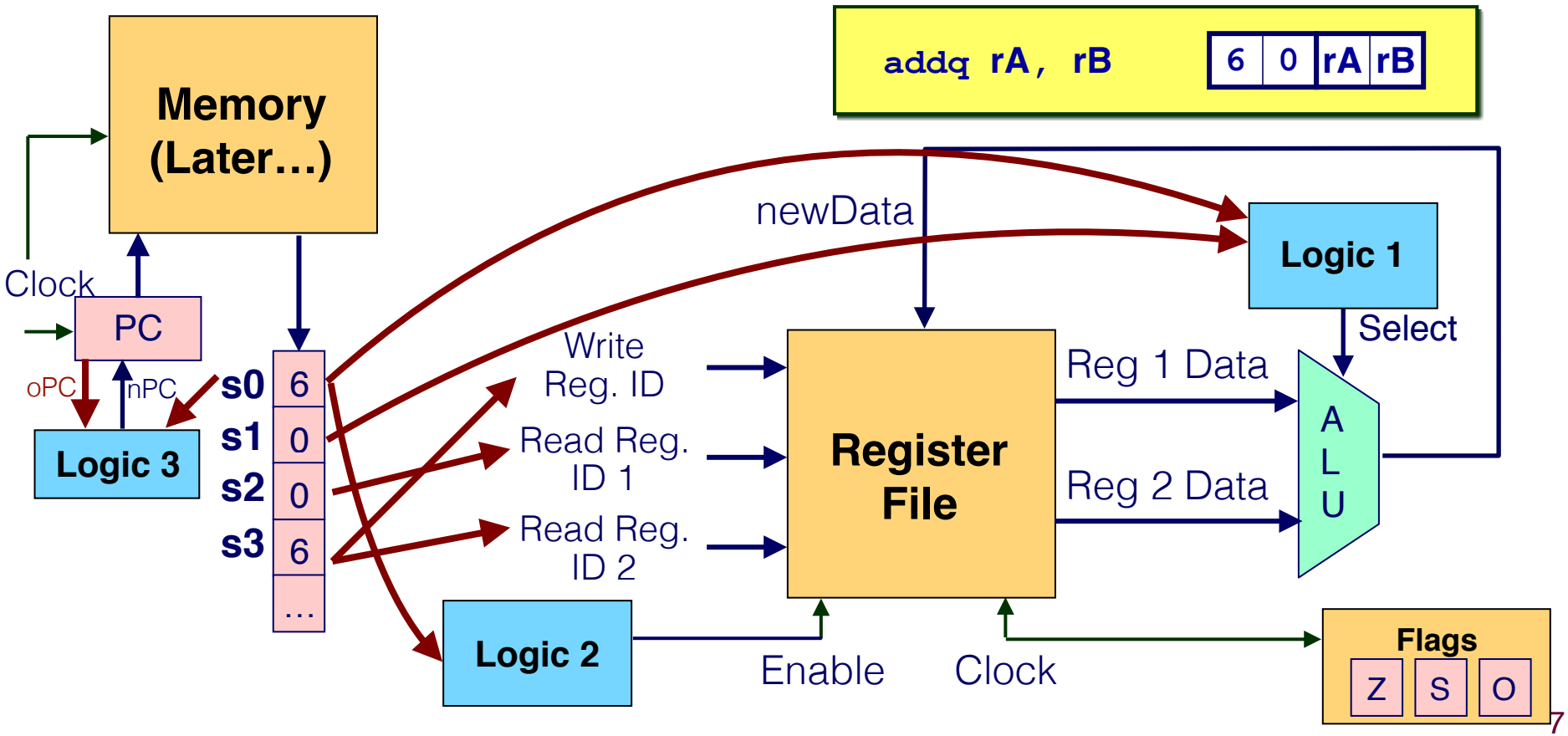
Executing an ADD instruction

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



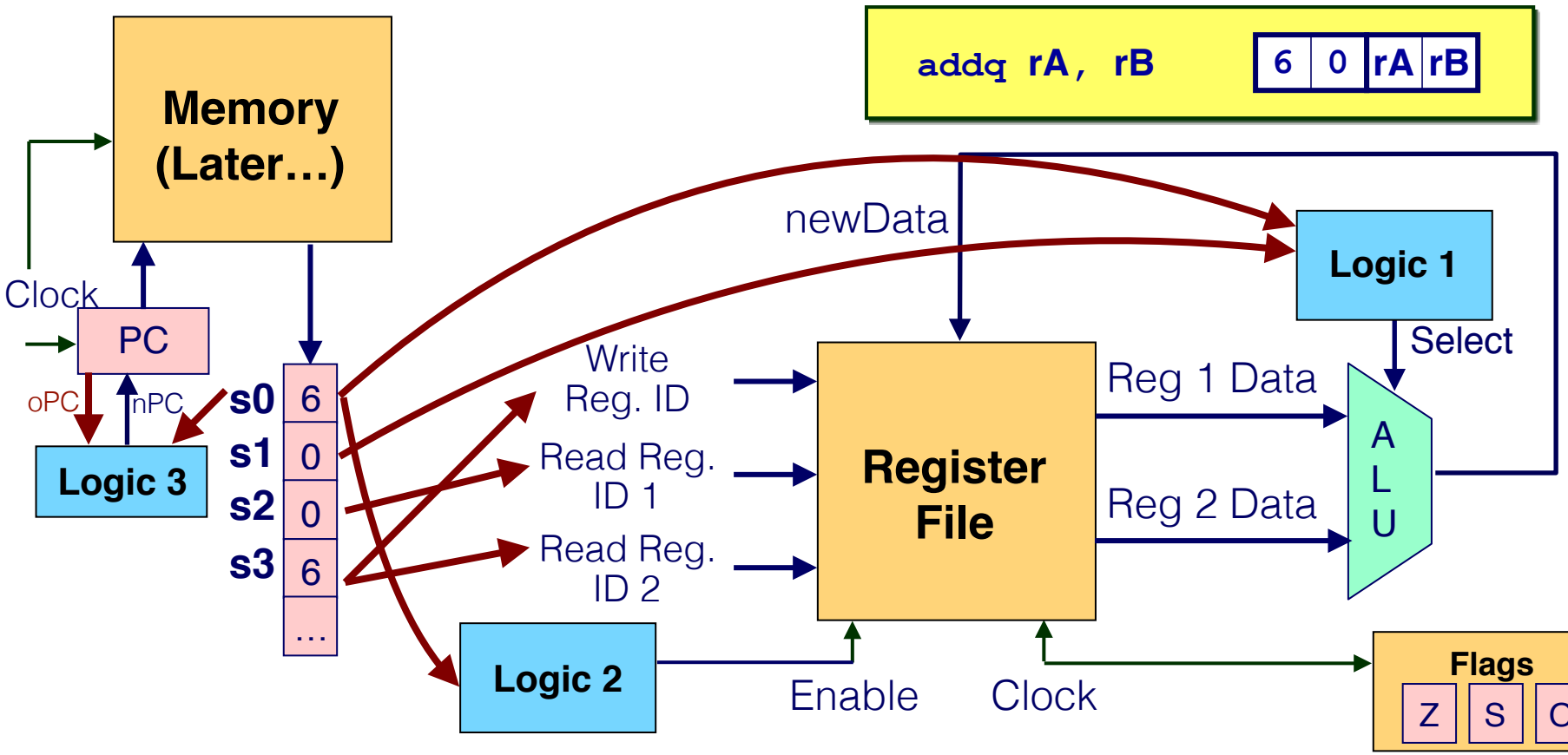
Executing an ADD instruction

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



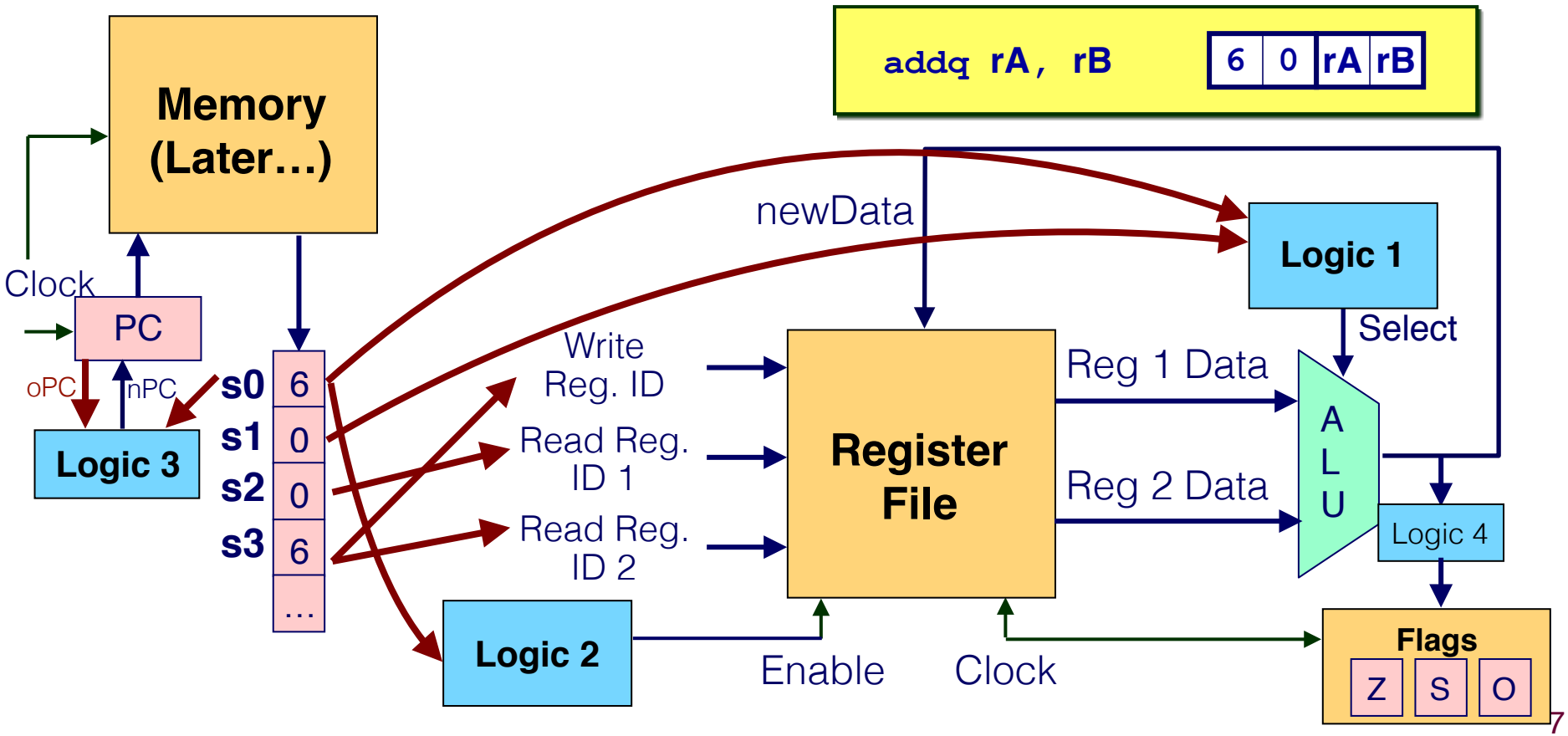
Executing an ADD instruction

- 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

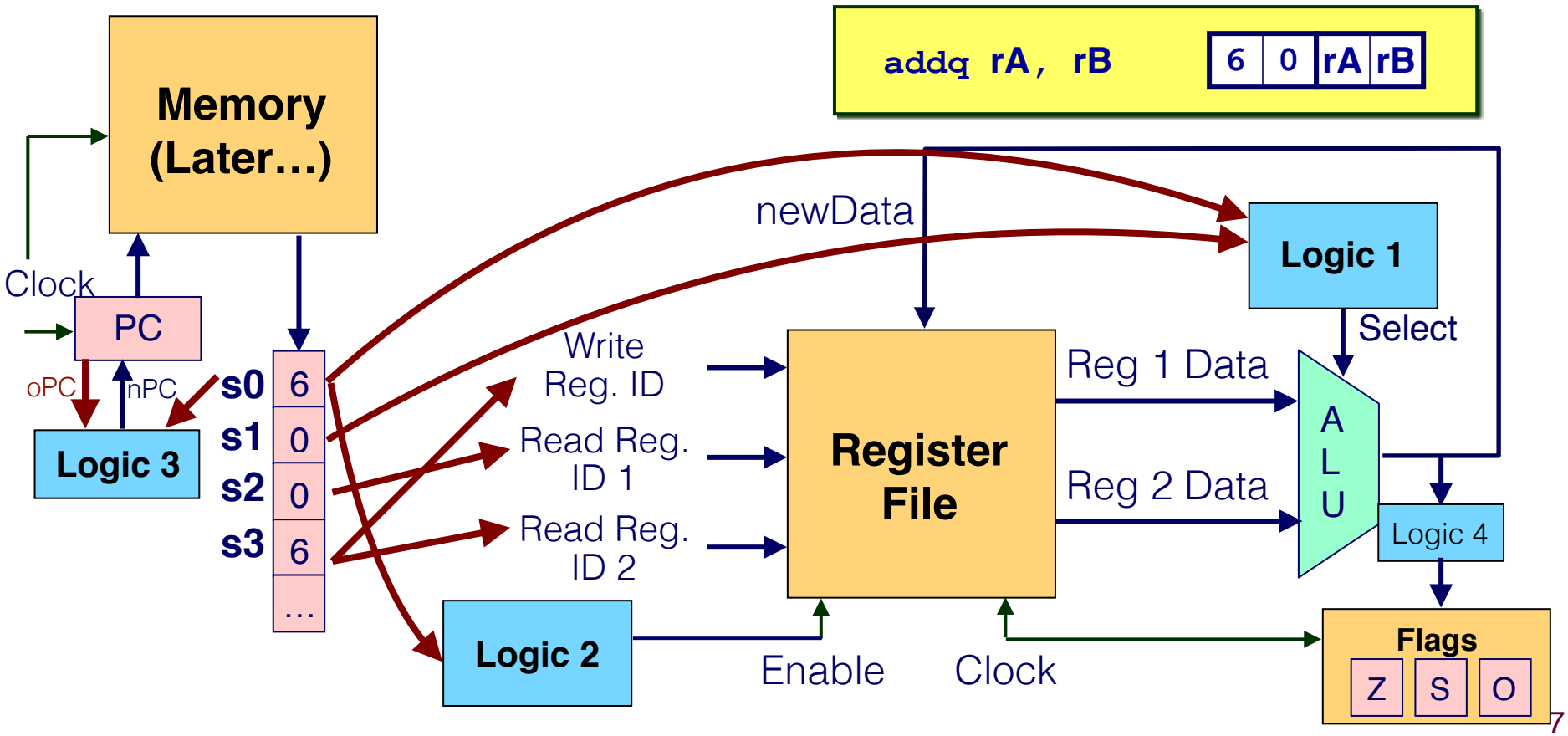
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?



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?

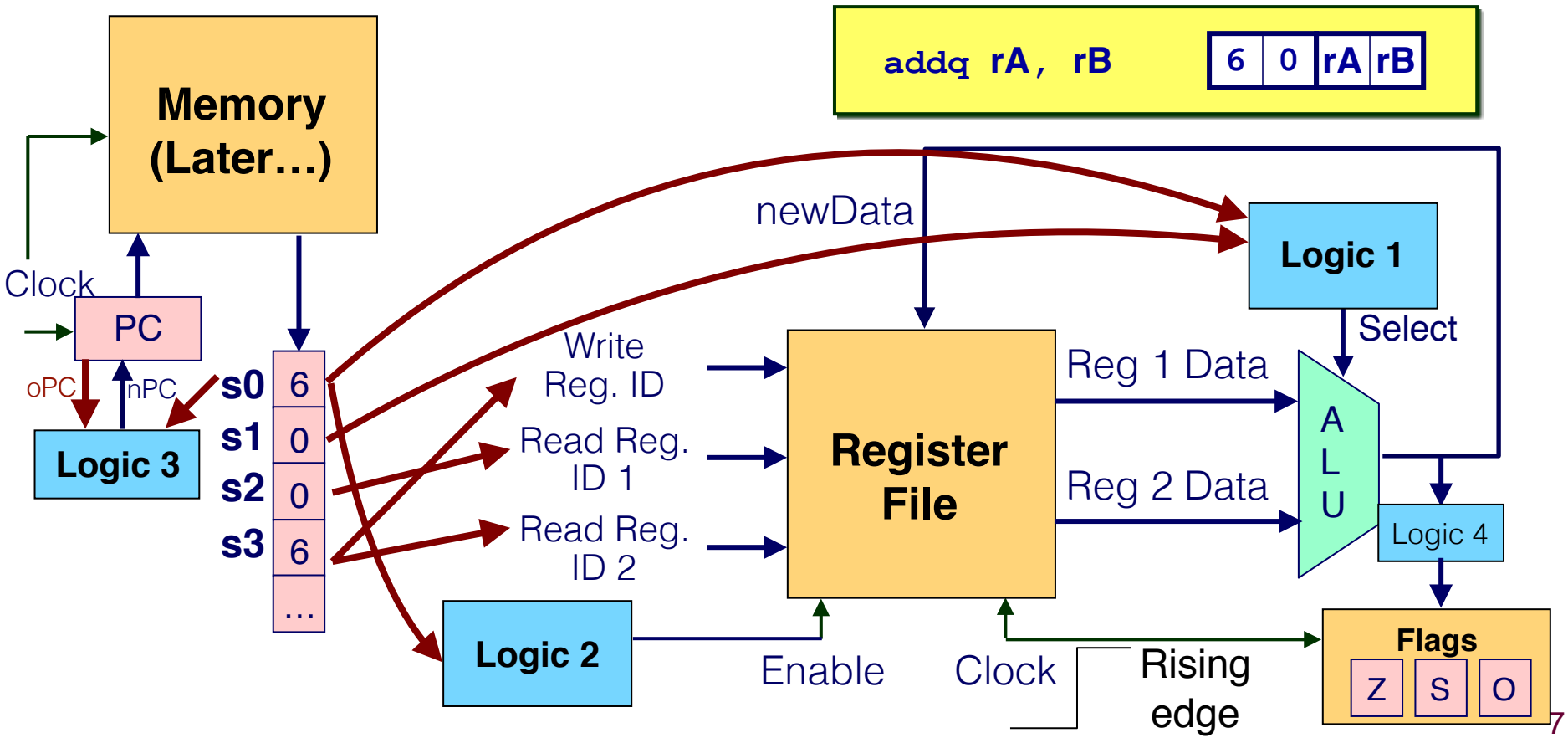
How do these logics get implemented?



Executing an ADD instruction

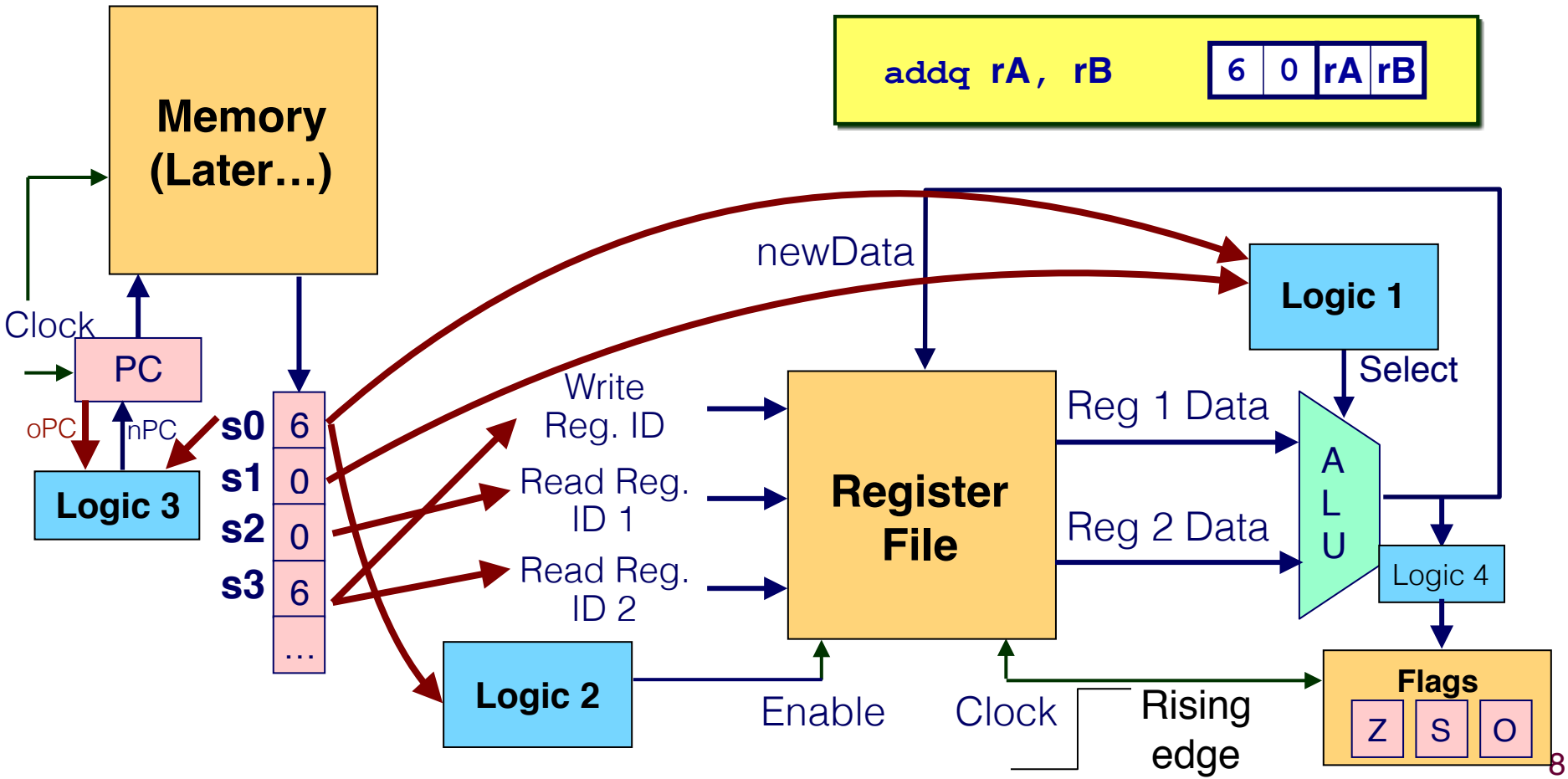
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

How do these logics get implemented?



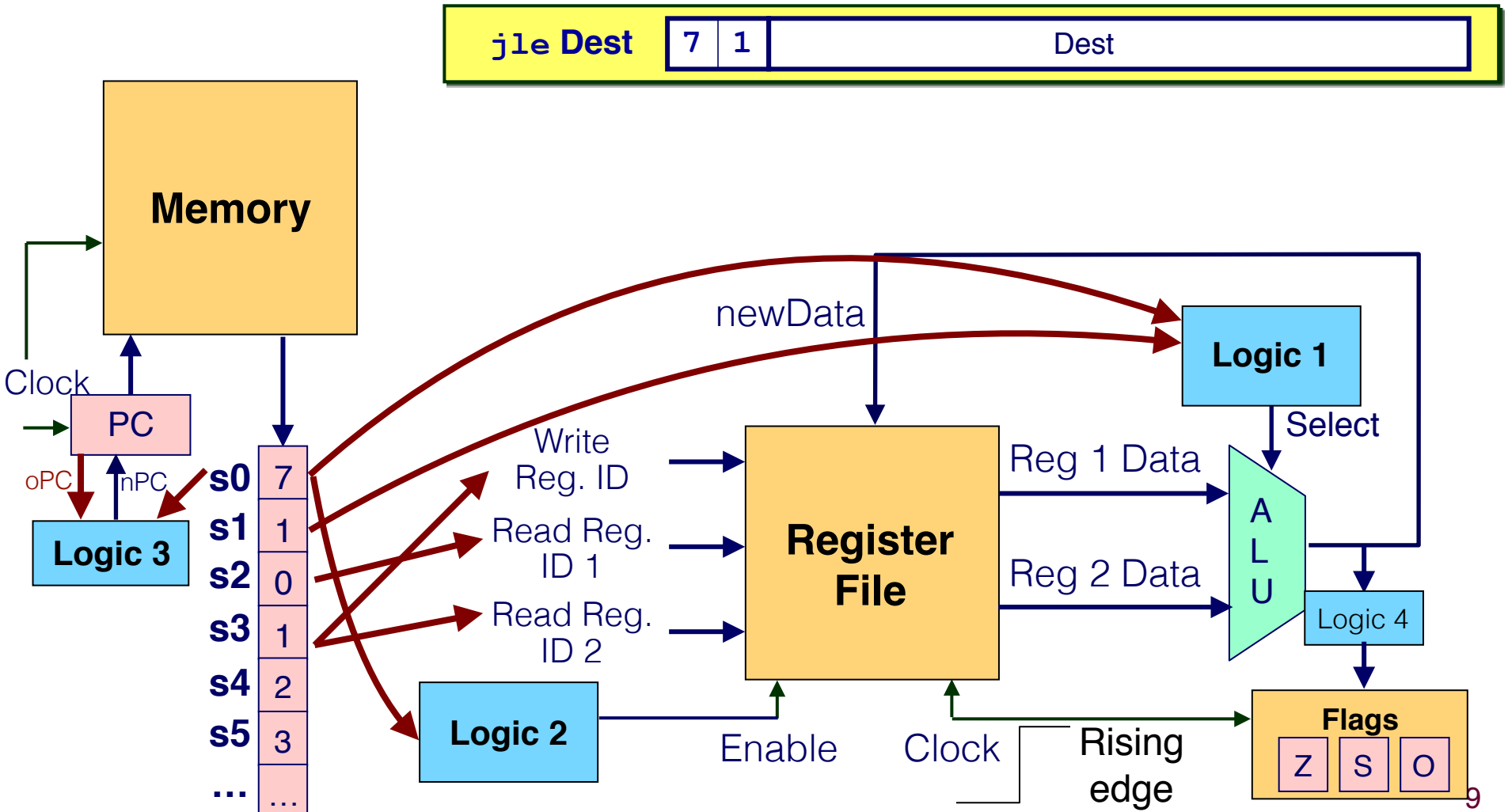
Executing an ADD instruction

- 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.

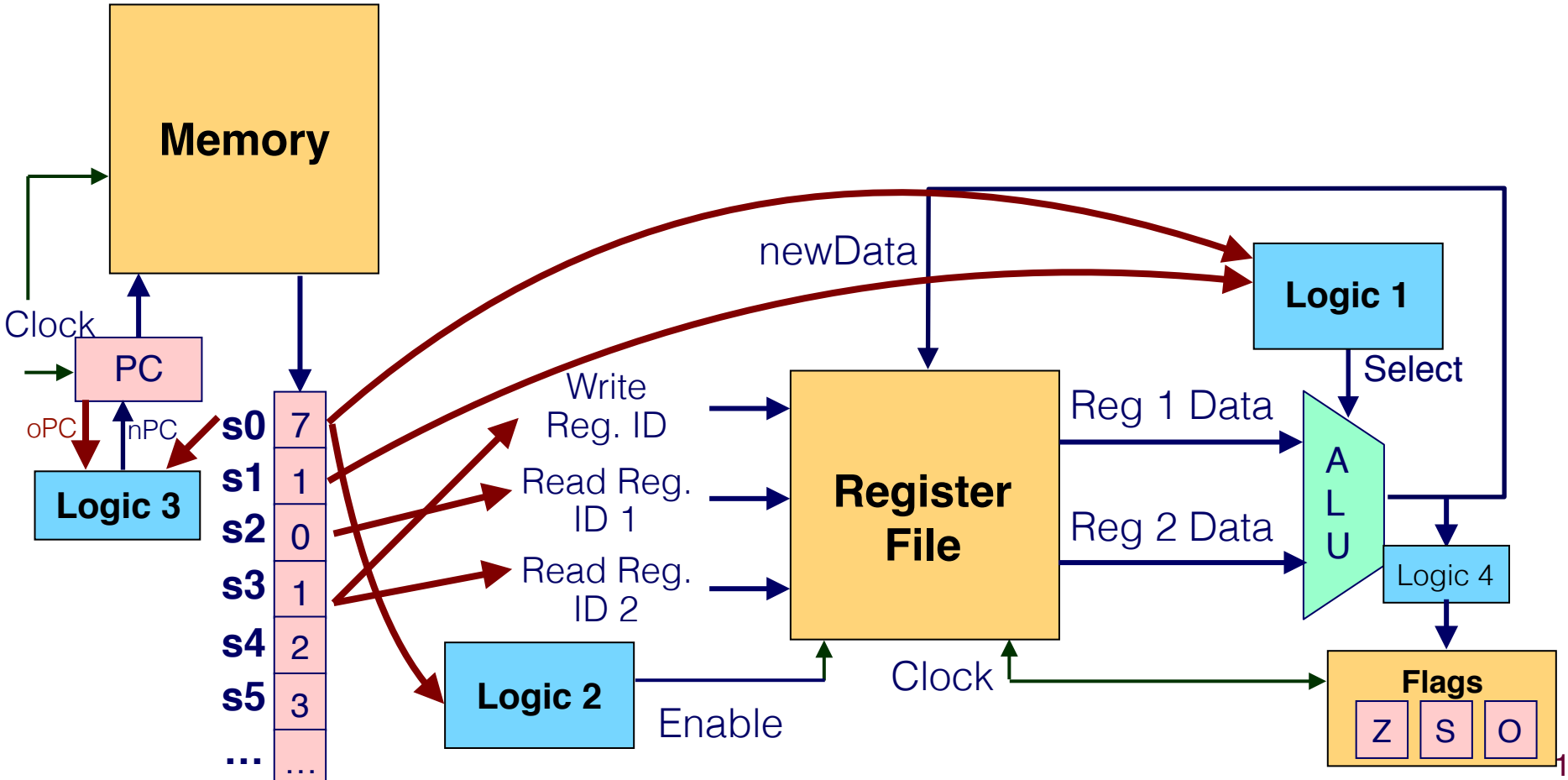


Executing a JLE instruction

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

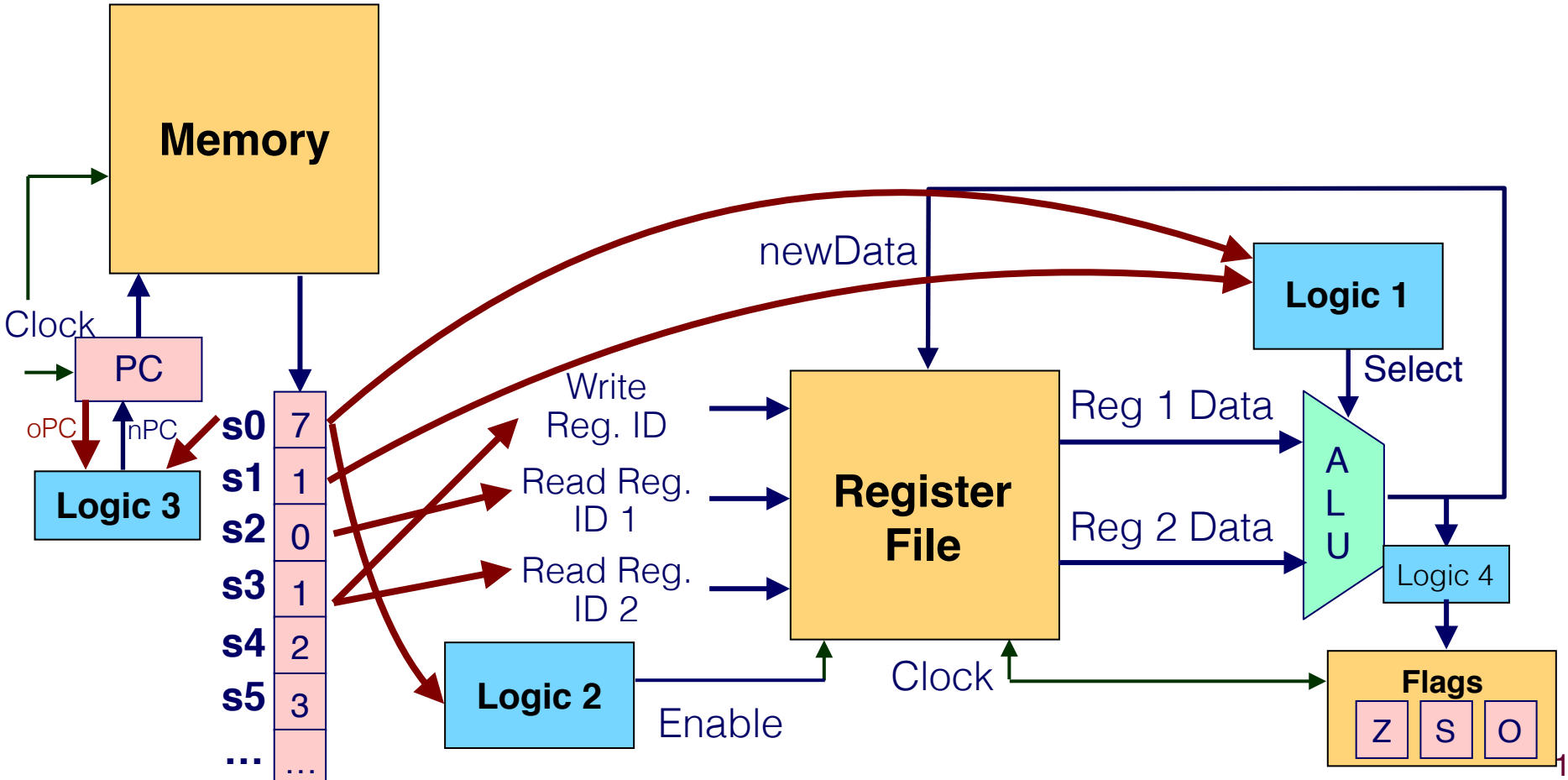


Executing a JLE instruction



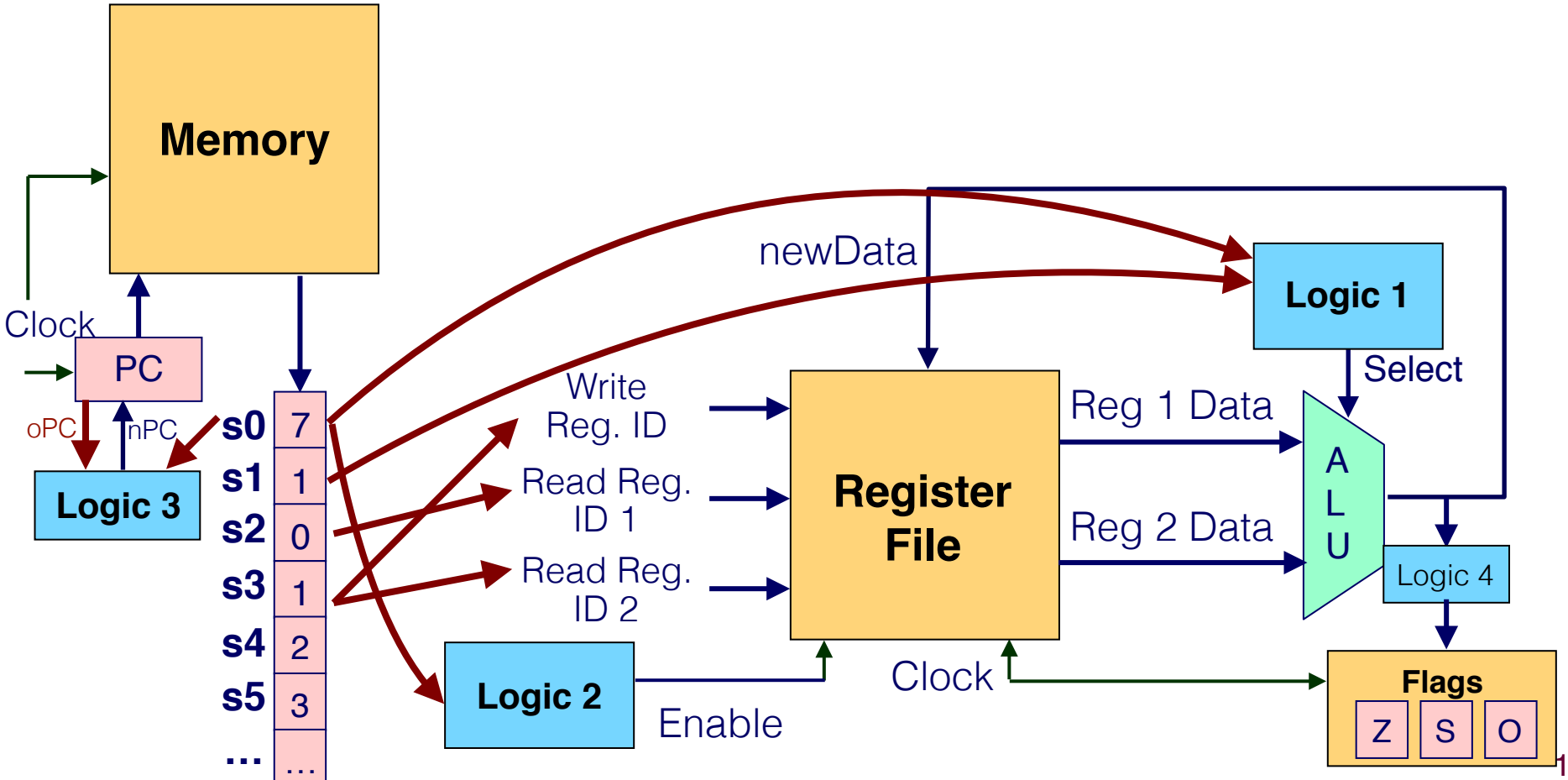
Executing a JLE instruction

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



Executing a JLE instruction

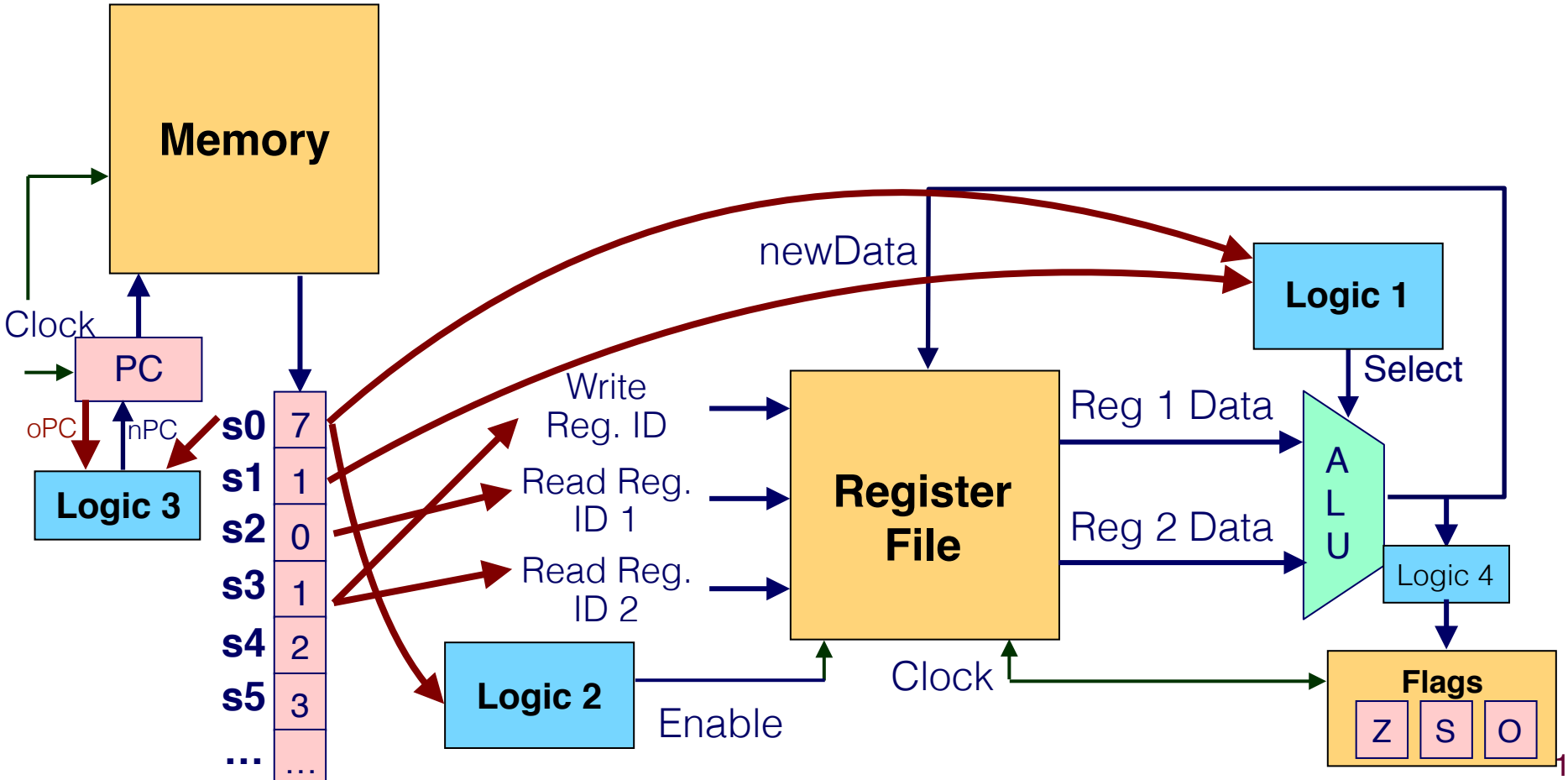
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;



jle Dest	7	1	Dest

Executing a JLE instruction

- Logic 3??

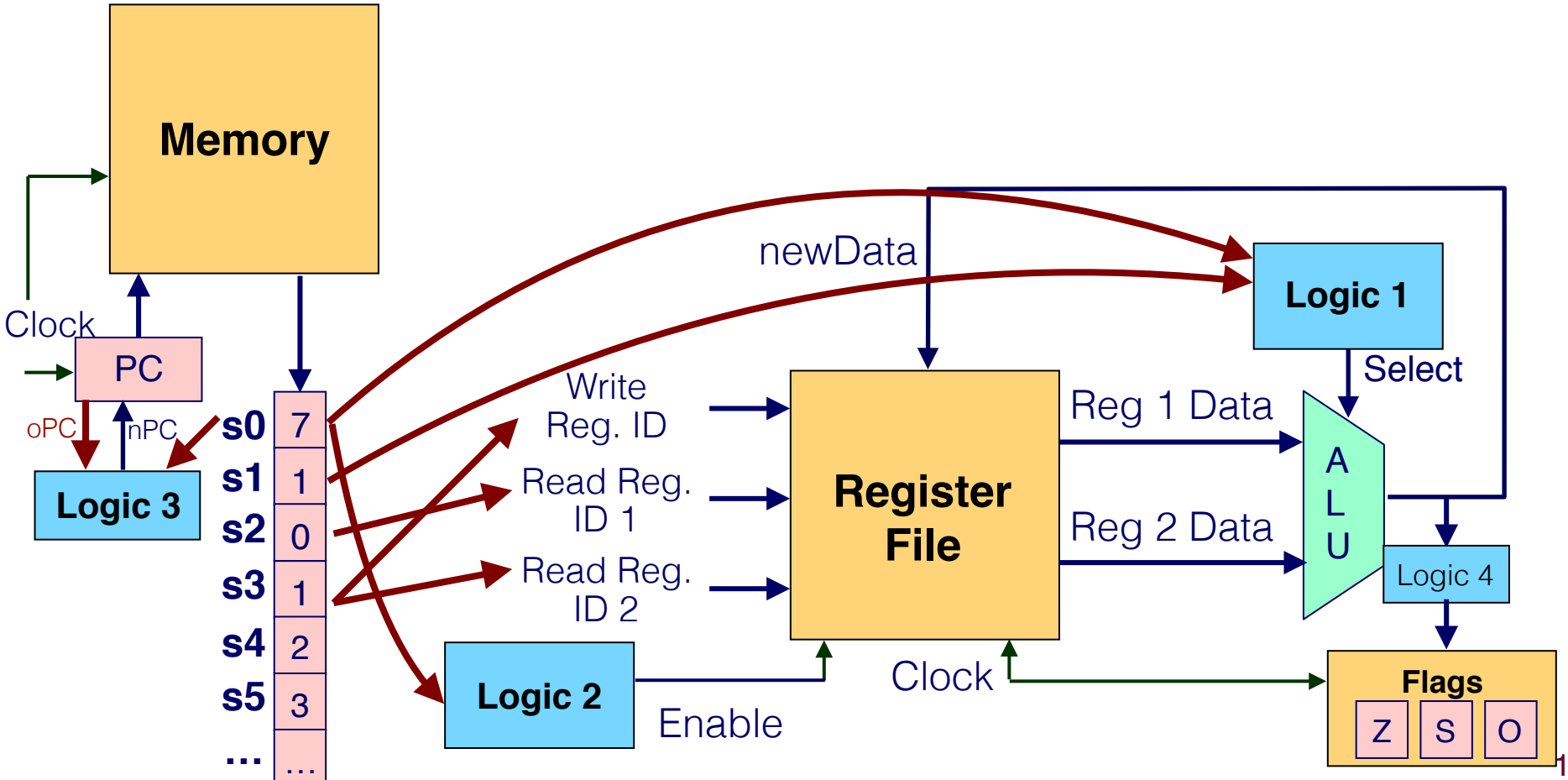


jle Dest	7	1	Dest

Executing a JLE instruction

- Logic 3??

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

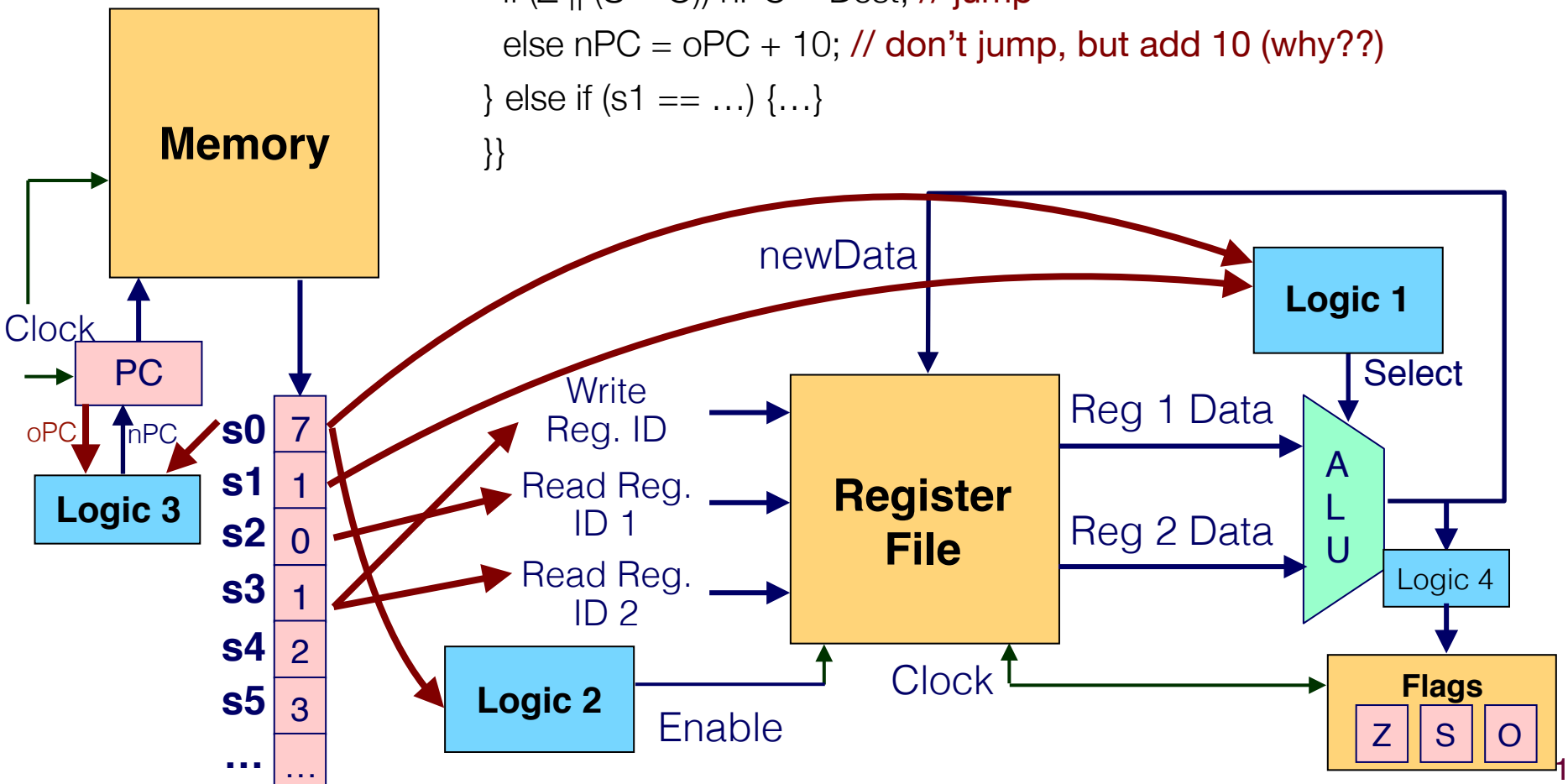


jle Dest	7	1	Dest

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 == …) {…}
  }}
```

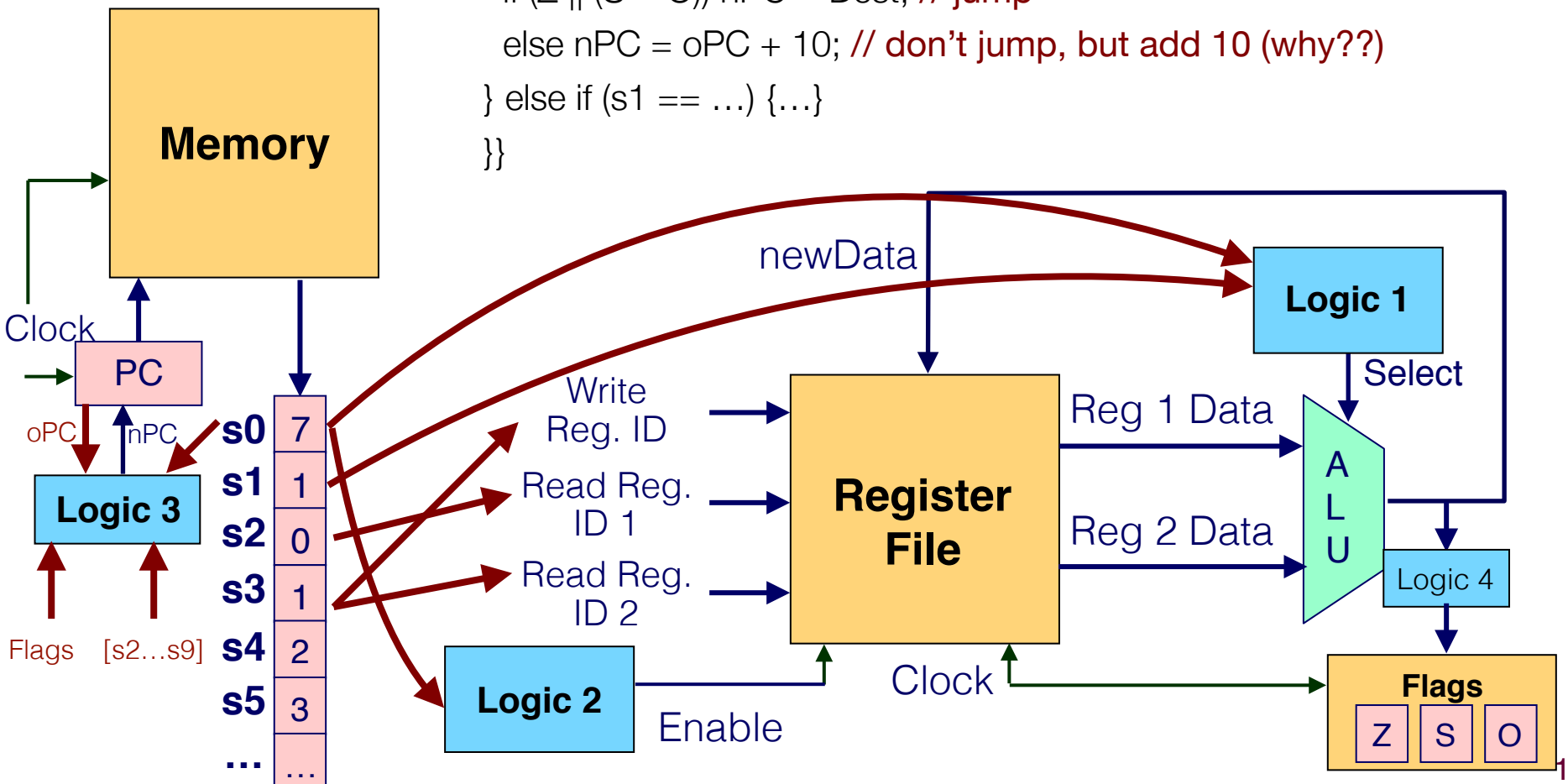


jle Dest	7	1	Dest

Executing a JLE instruction

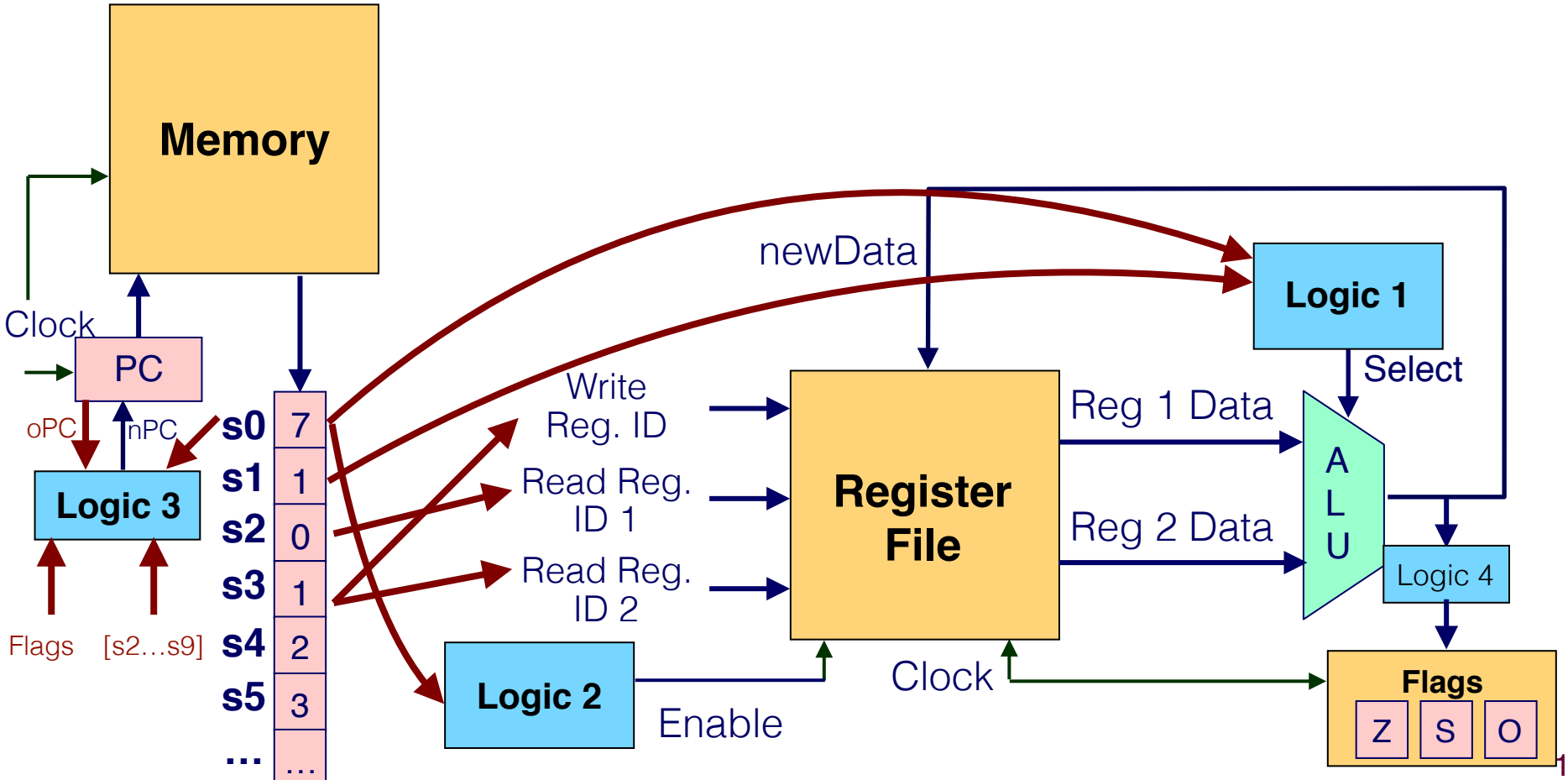
- Logic 3??

```
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 == …) {…}
  }}
```



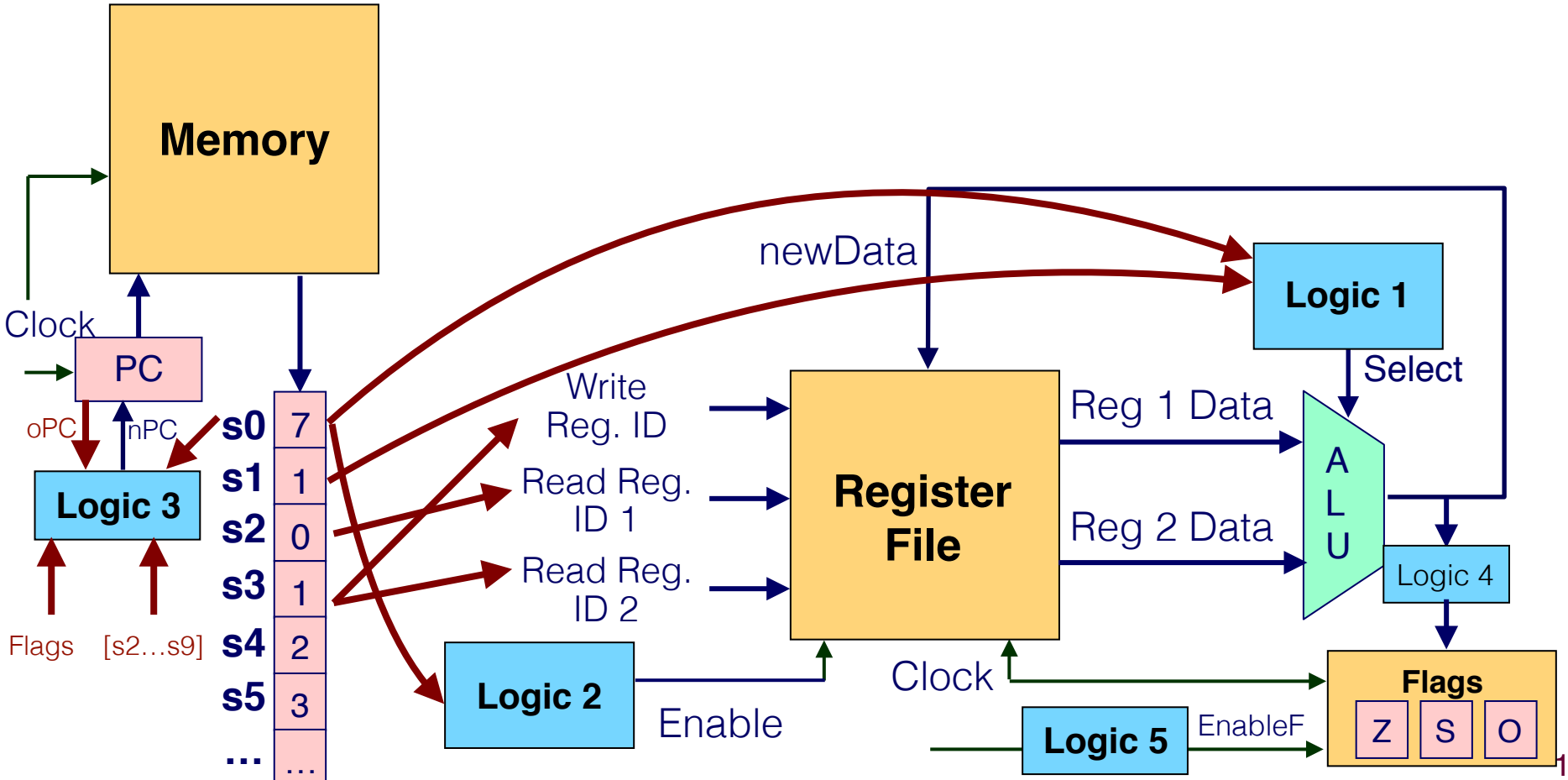
Executing a JLE instruction

- Logic 4? Does JLE write flags?



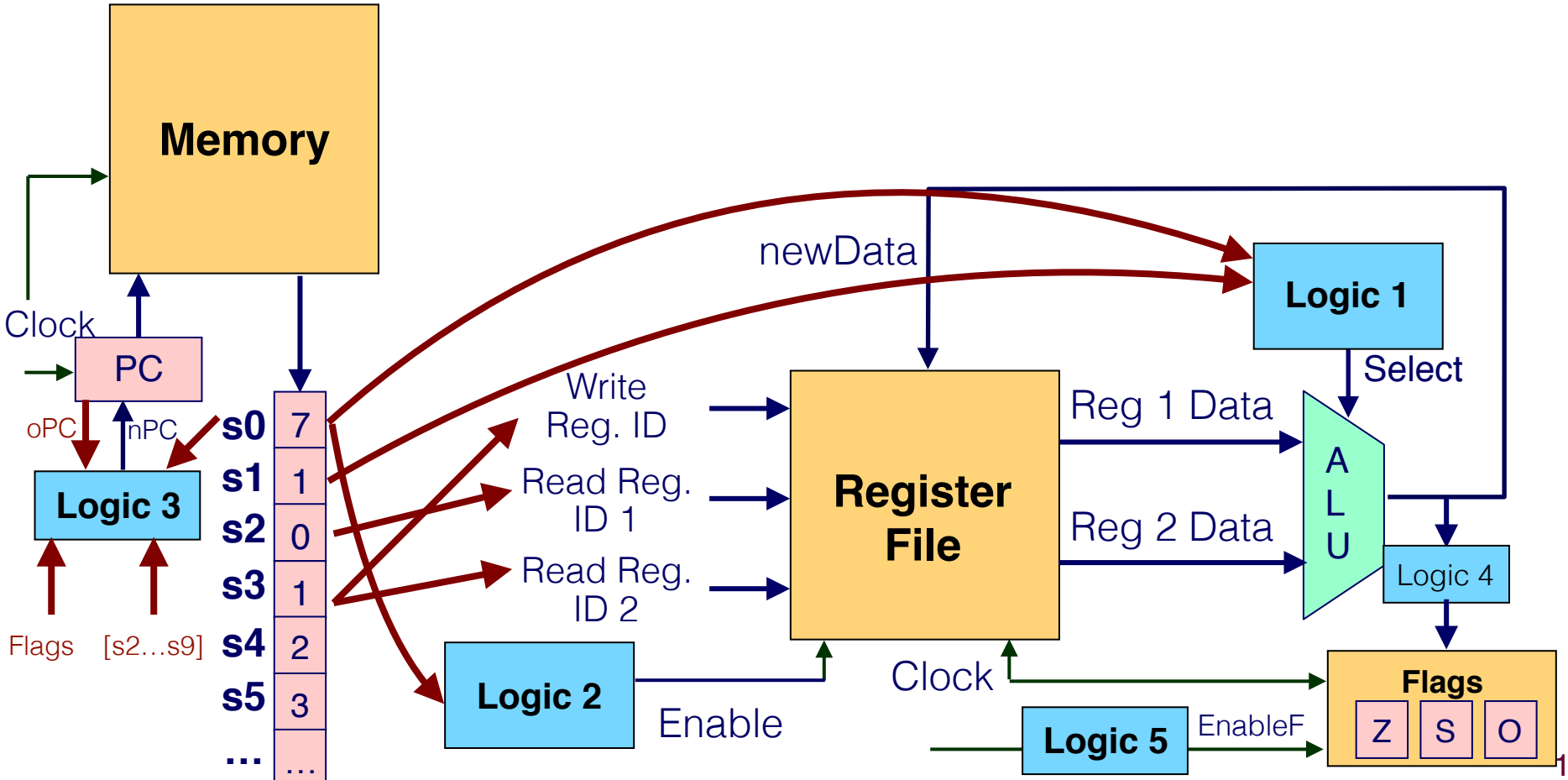
Executing a JLE instruction

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

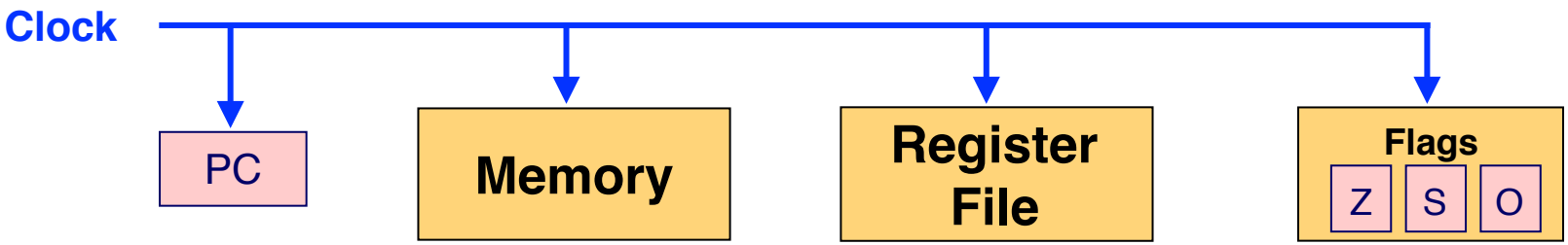


Executing a JLE instruction

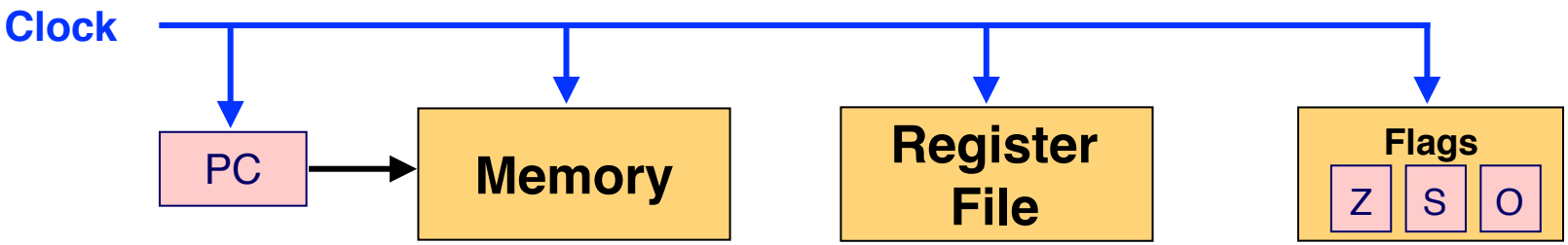
- Logic 4? Does JLE write flags?
- Need another piece of logic.
- Logic 5: if (s0 == 7) EnableF = 0; else if (s0 == 6) EnableF = 1;



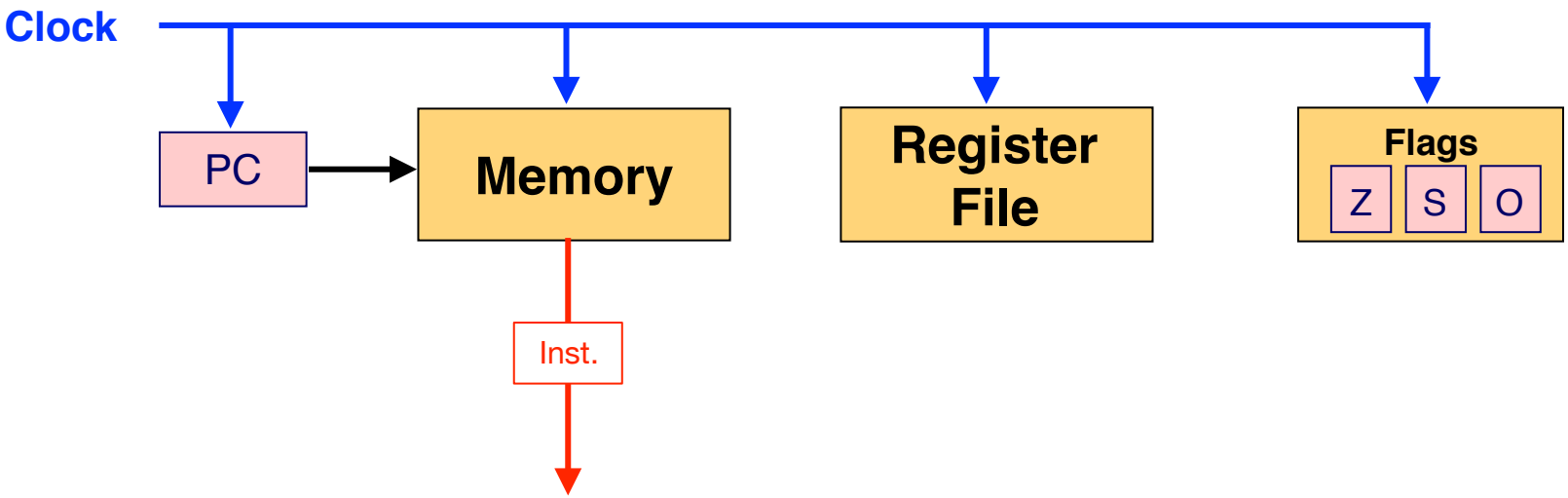
Microarchitecture (So far)



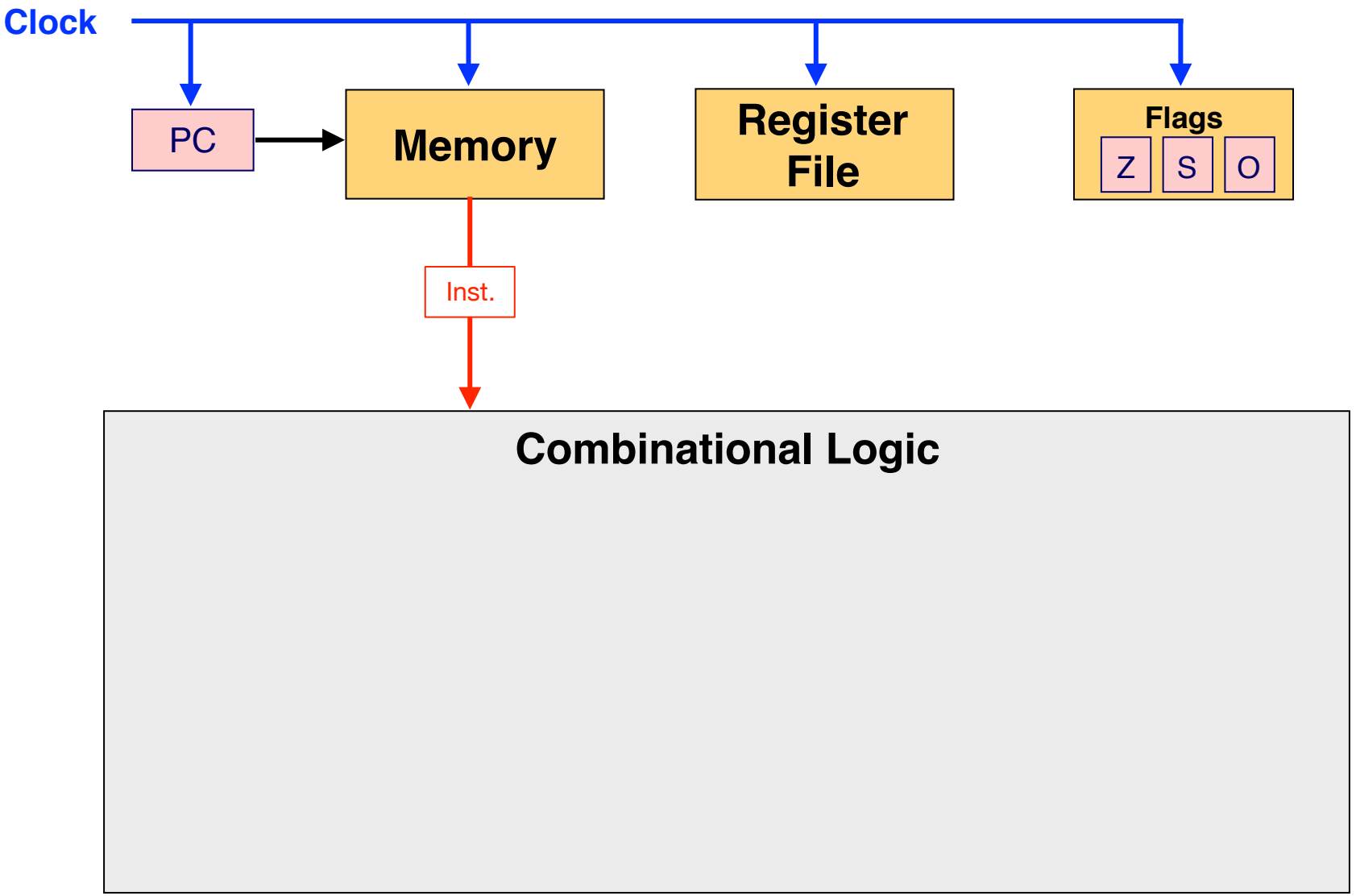
Microarchitecture (So far)



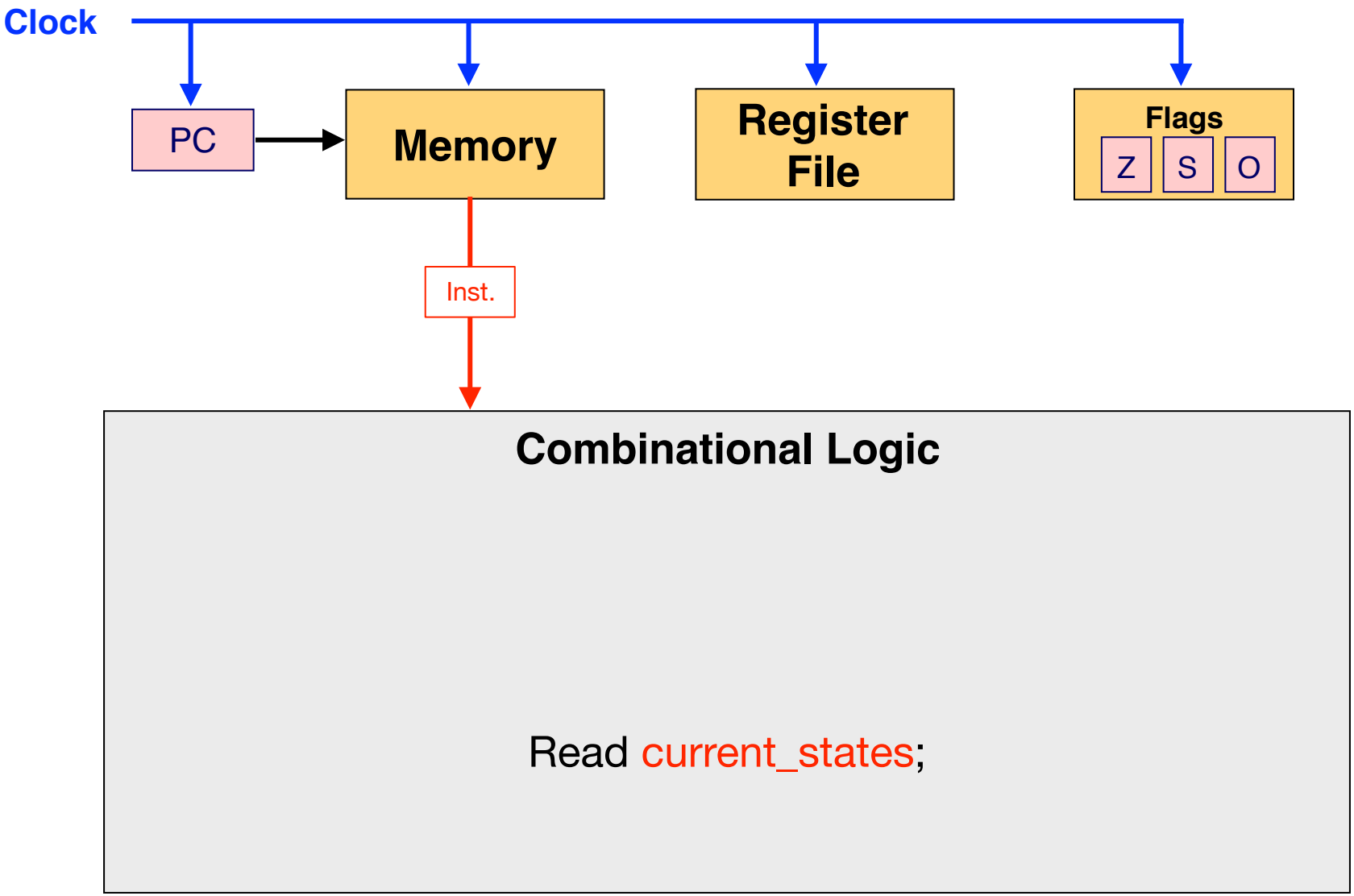
Microarchitecture (So far)



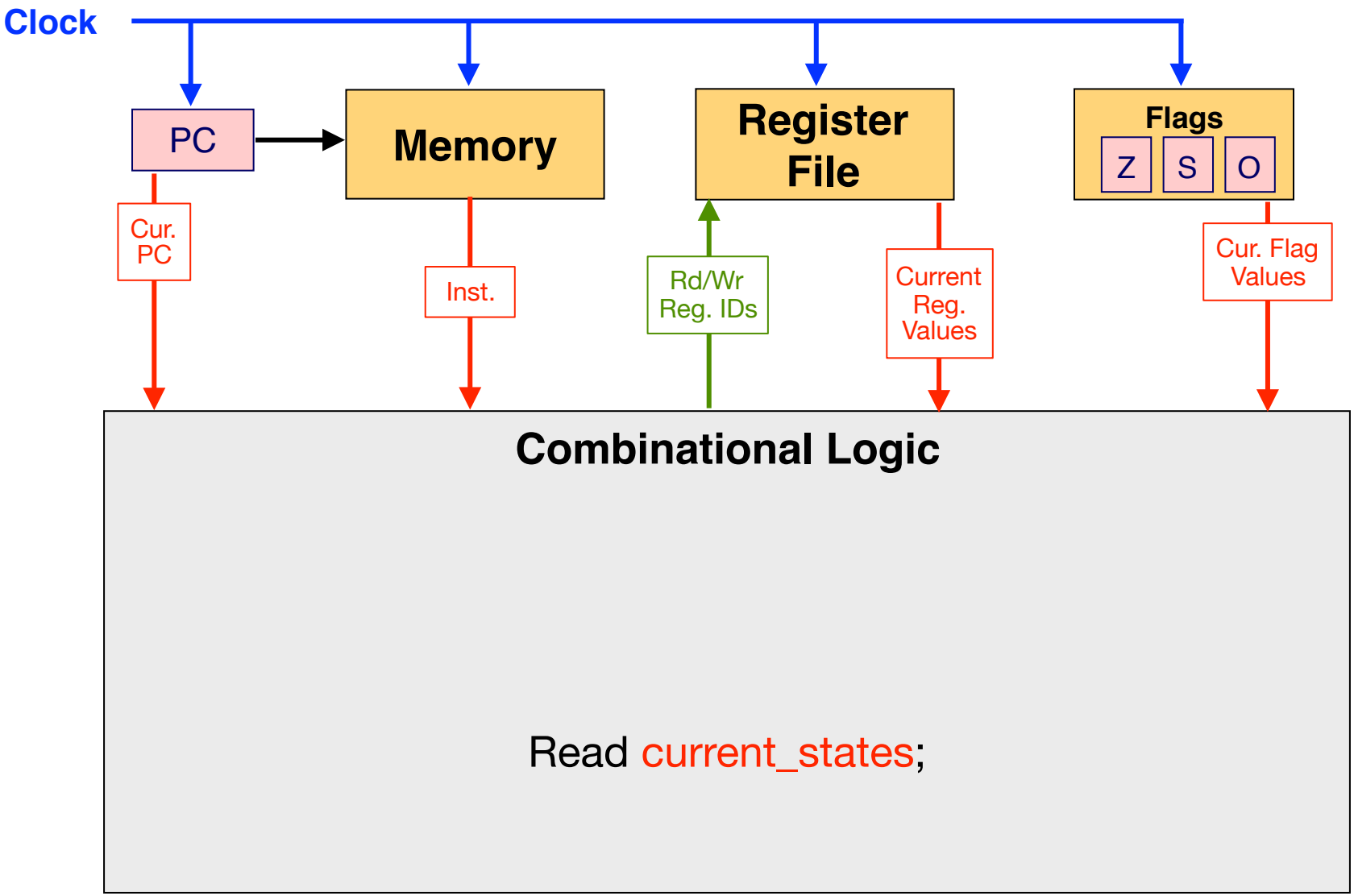
Microarchitecture (So far)



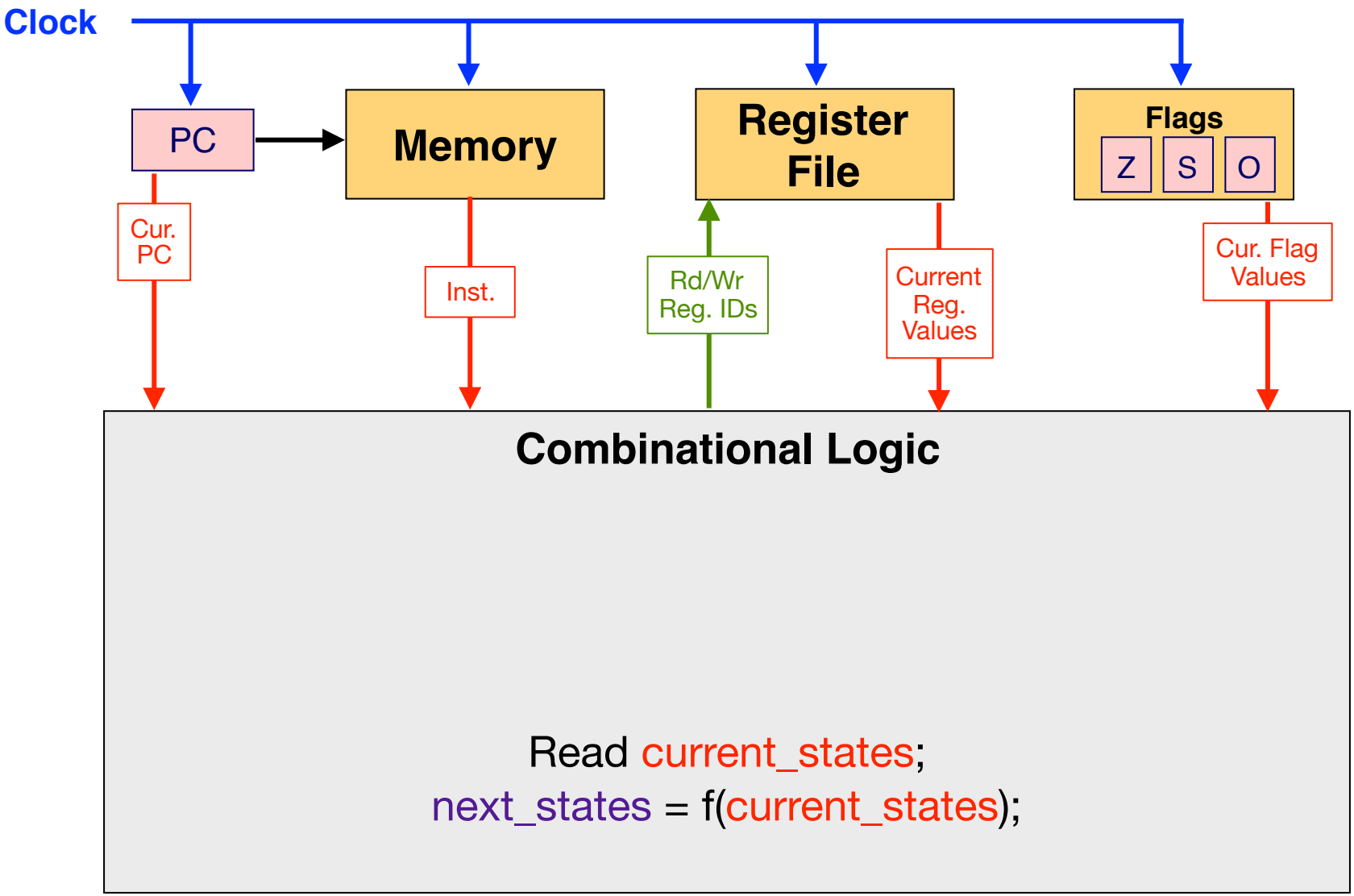
Microarchitecture (So far)



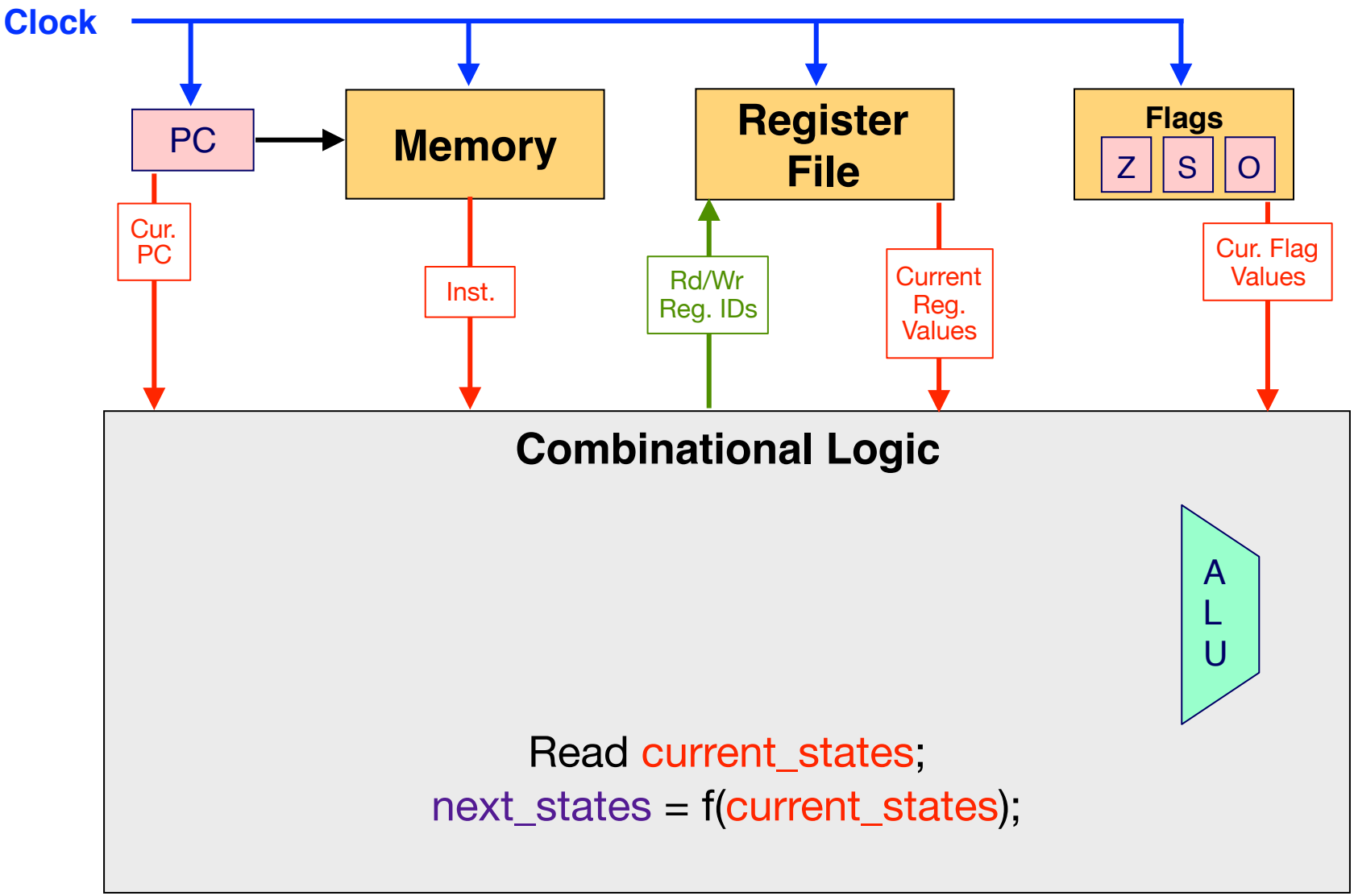
Microarchitecture (So far)



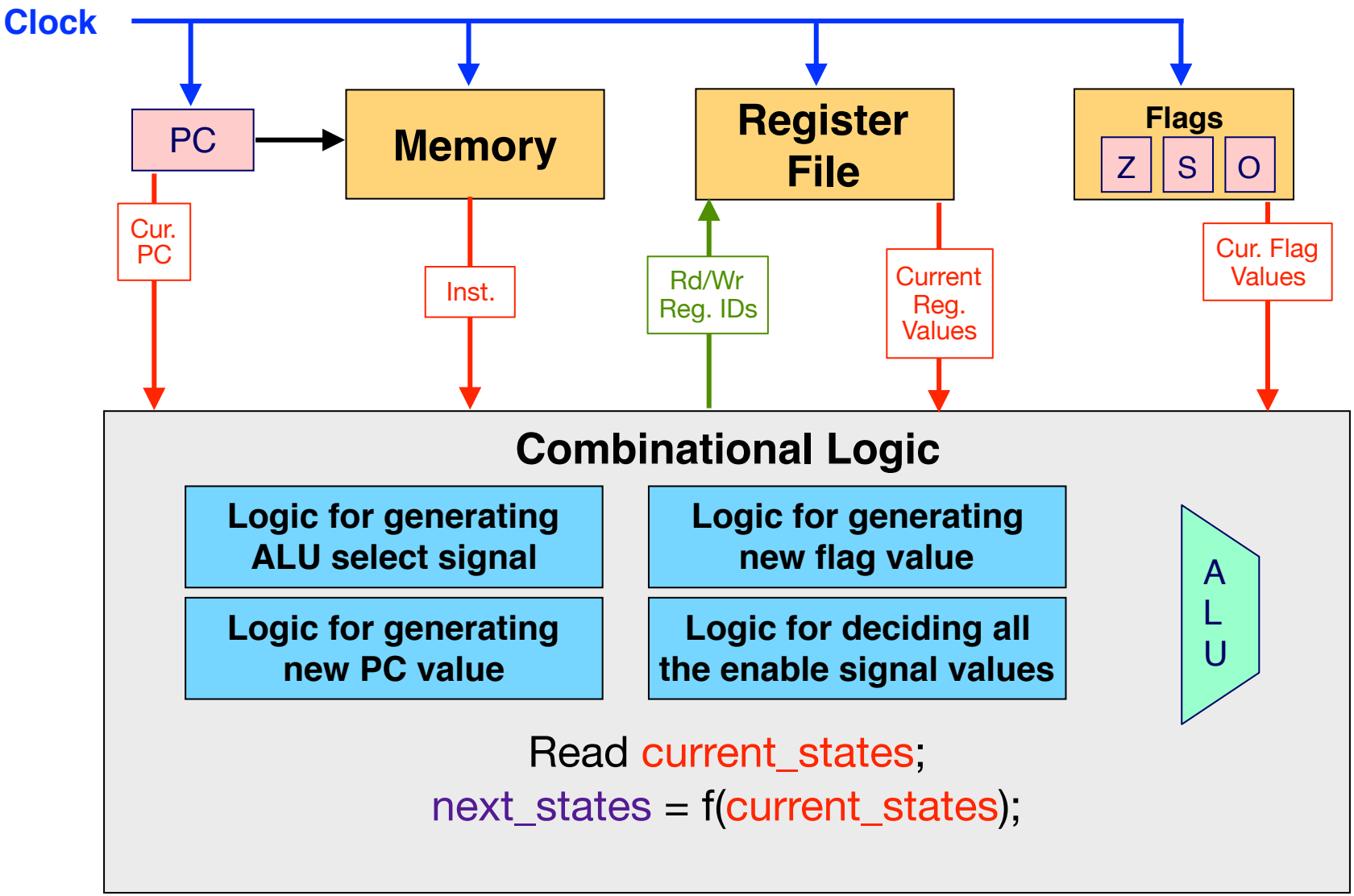
Microarchitecture (So far)



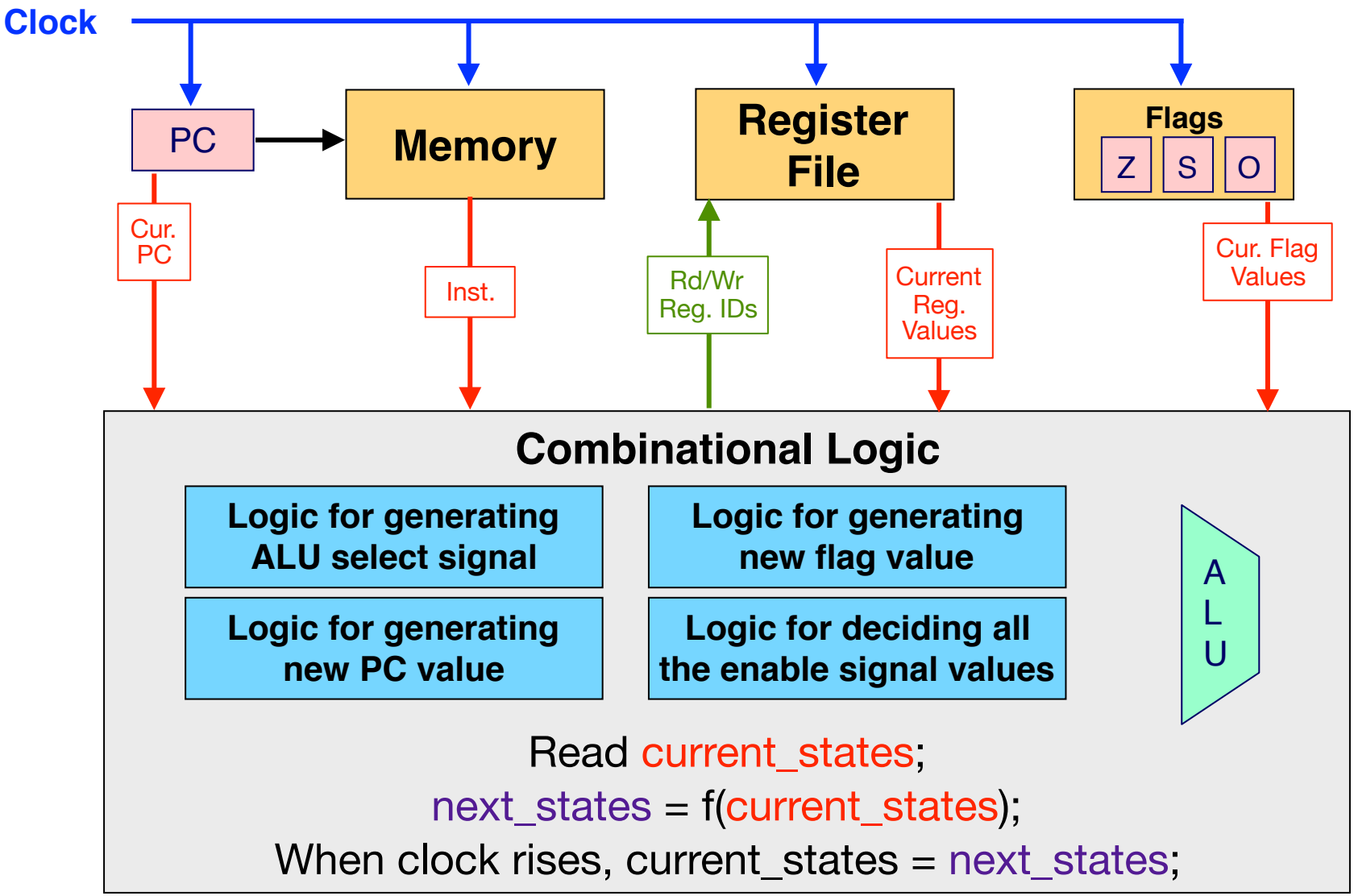
Microarchitecture (So far)



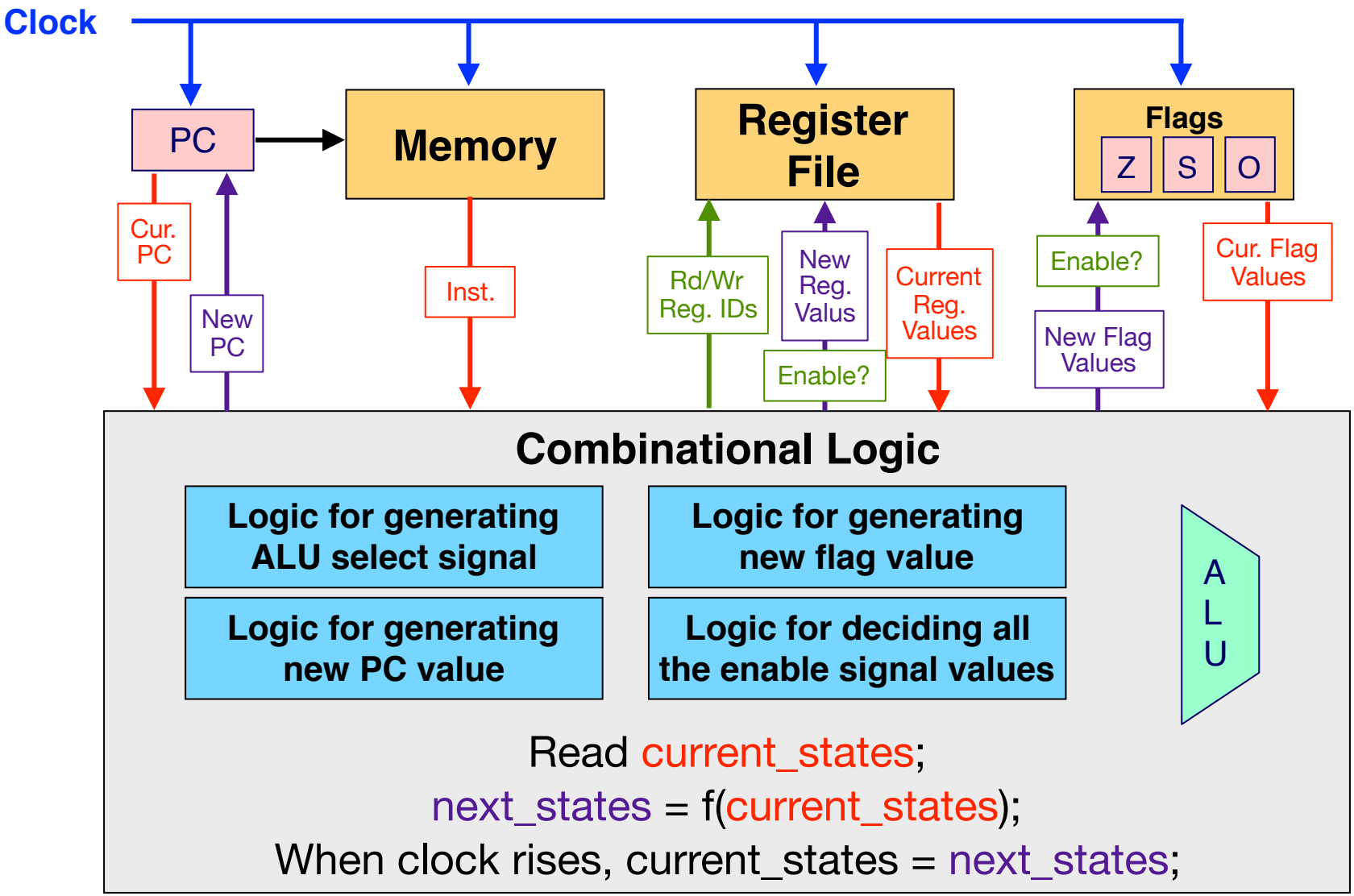
Microarchitecture (So far)



Microarchitecture (So far)

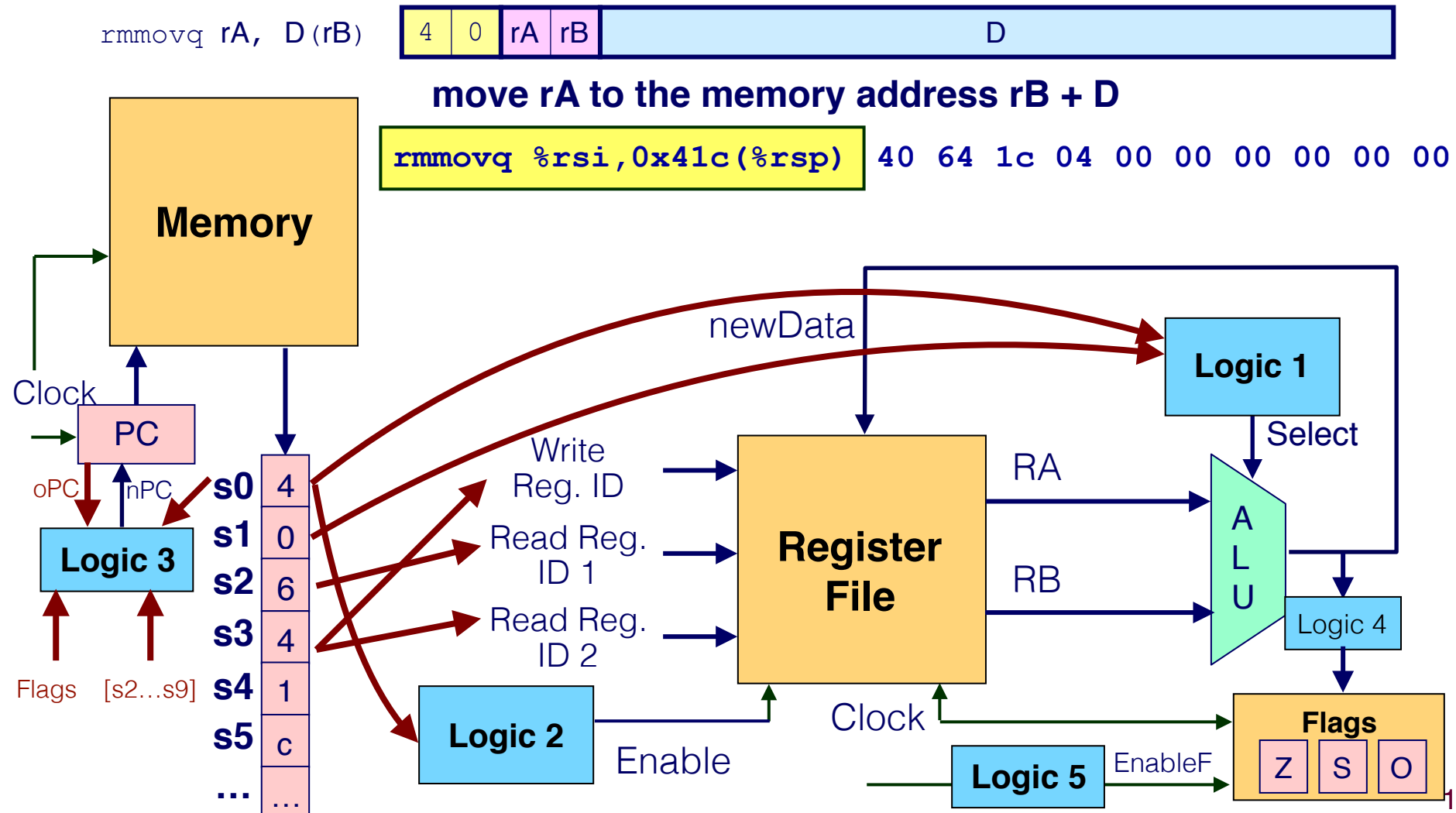


Microarchitecture (So far)



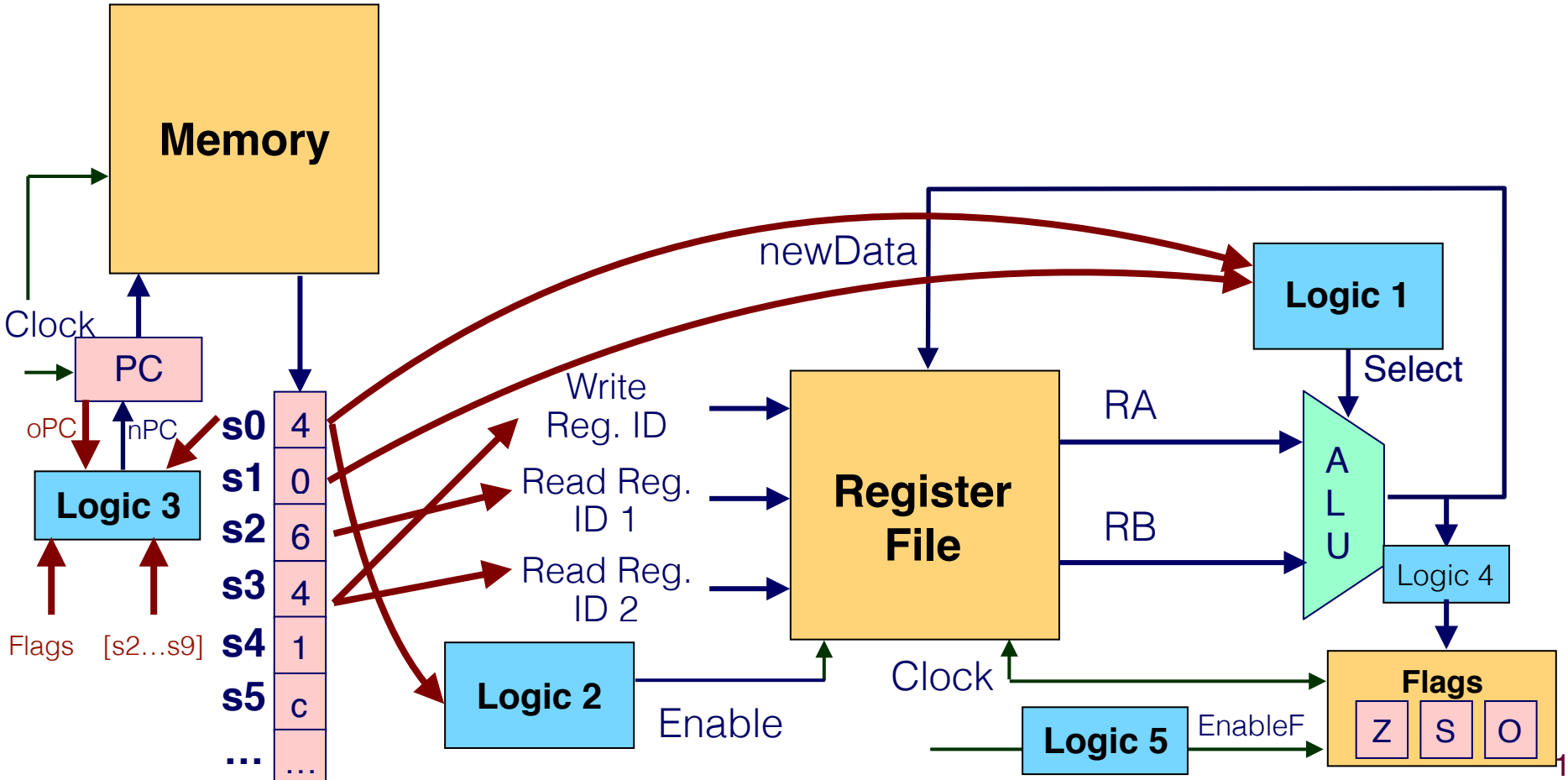
Executing a MOV instruction

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



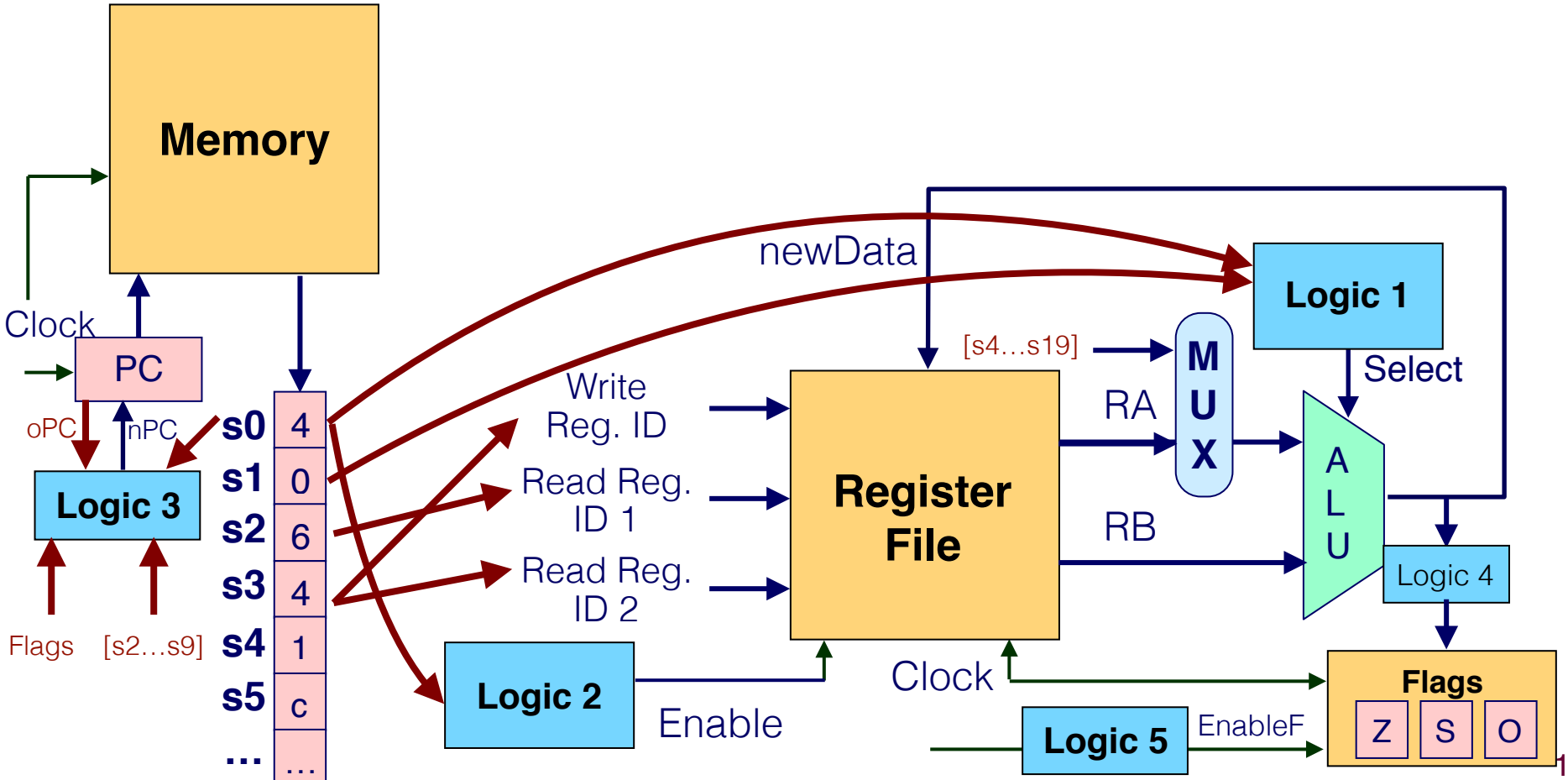
move rA to the memory address rB + D

`rmmovq rA, D(rB)`



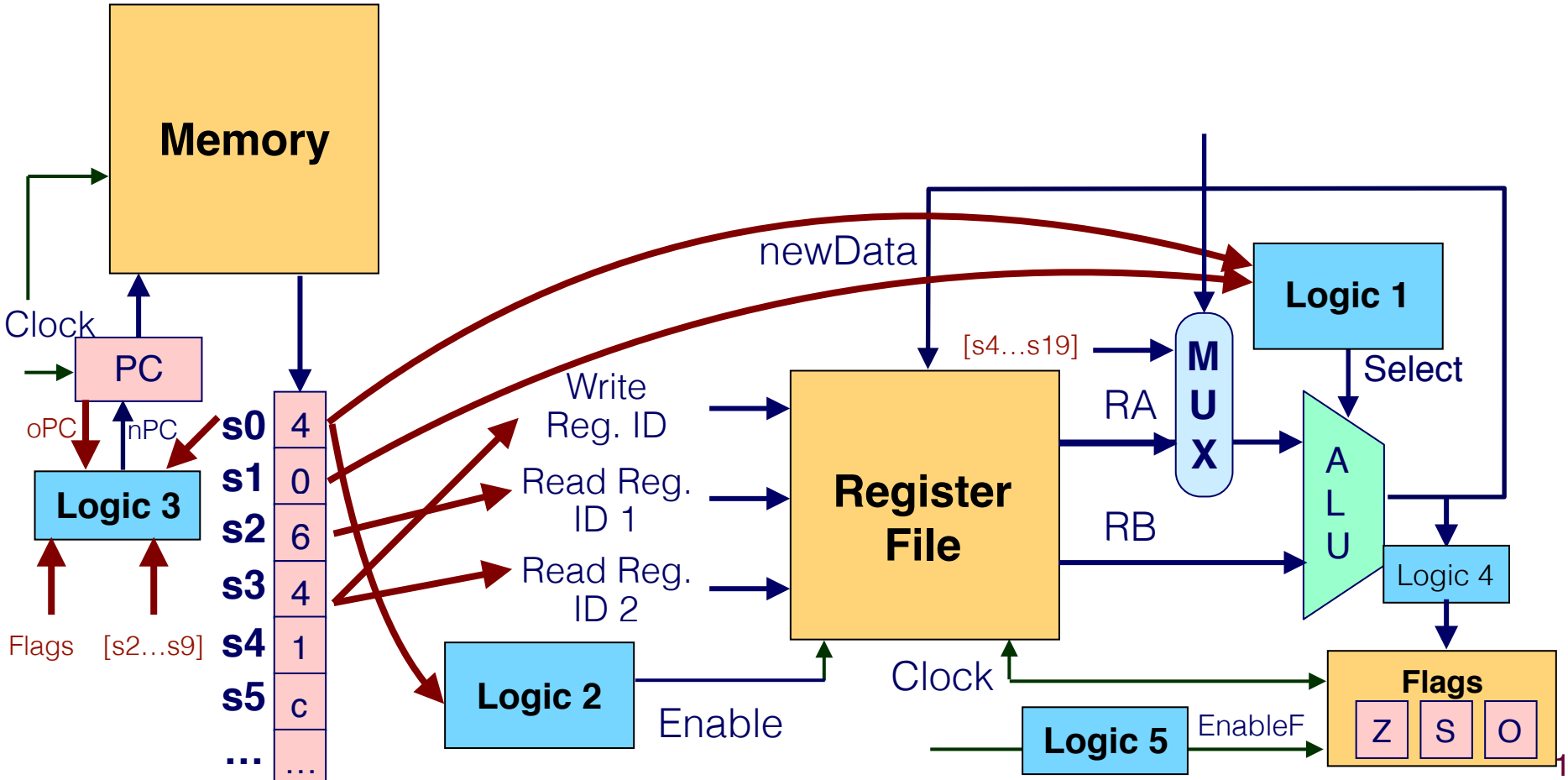
move rA to the memory address rB + D

`rmmovq rA, D(rB)`



move rA to the memory address rB + D

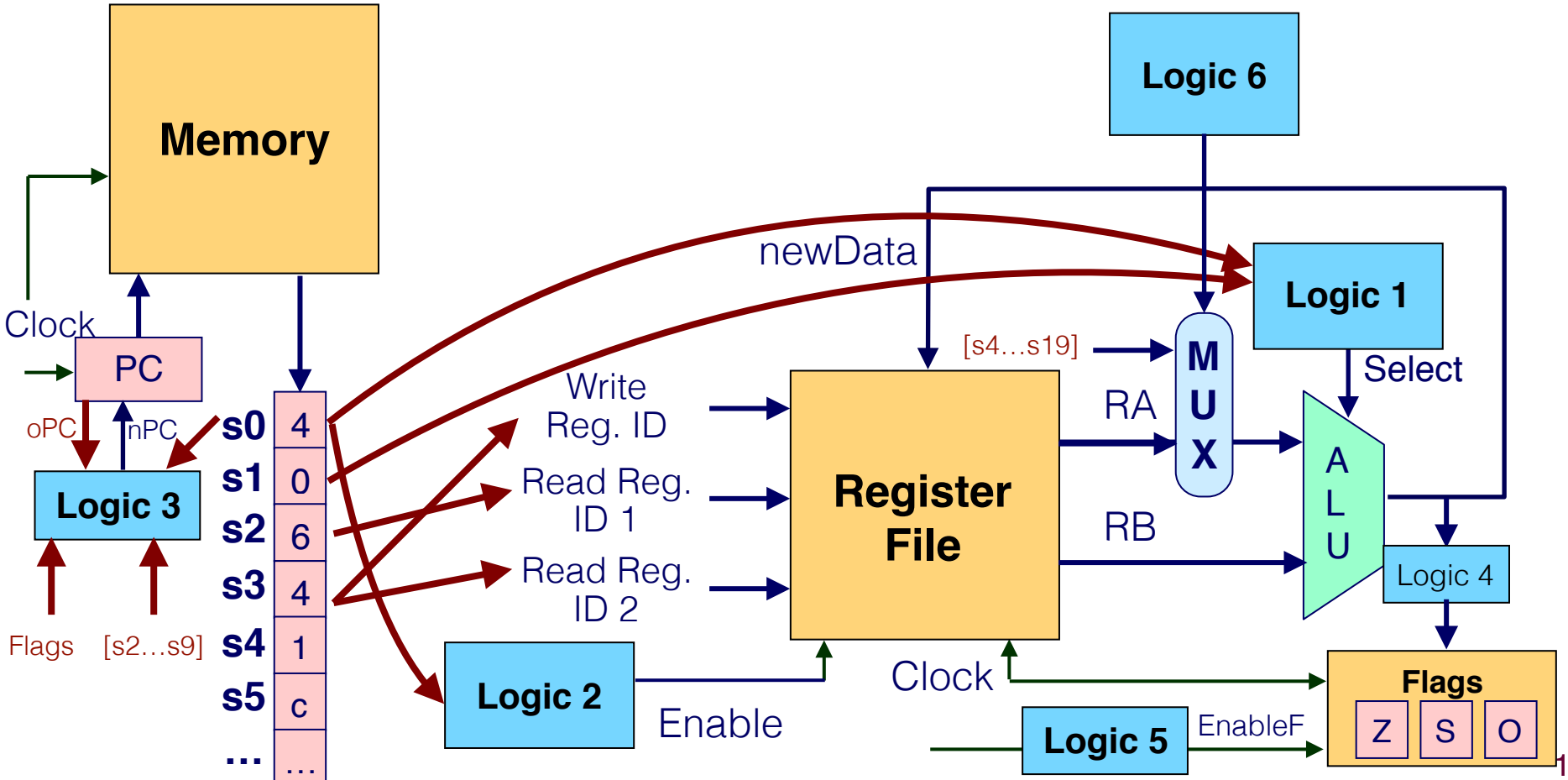
`rmmovq rA, D(rB)`



move rA to the memory address rB + D

`rmmovq rA, D(rB)`

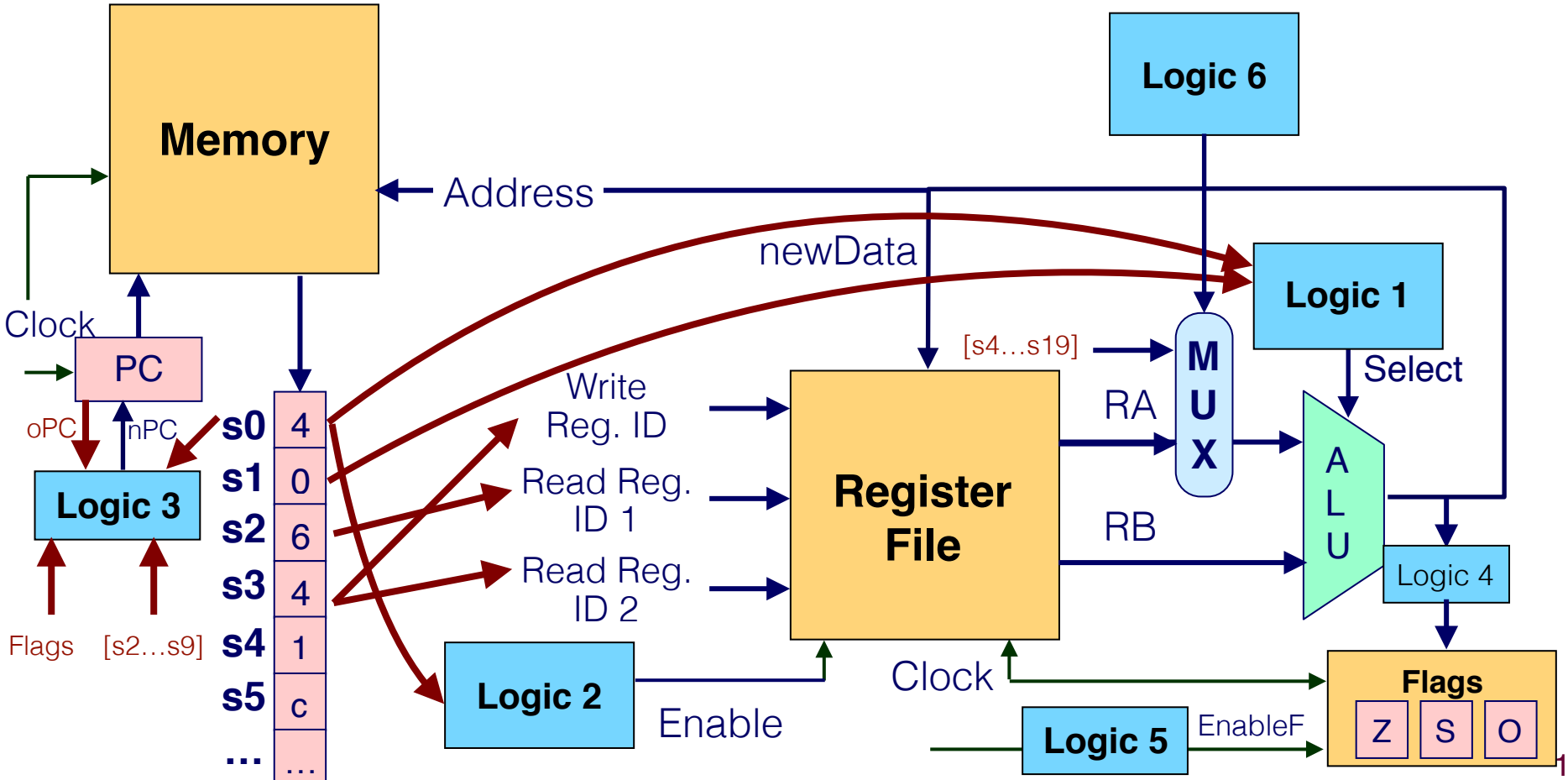
- Need new logic (Logic 6) to select the input to the ALU for Enable.



move rA to the memory address rB + D

`rmmovq rA, D(rB)`

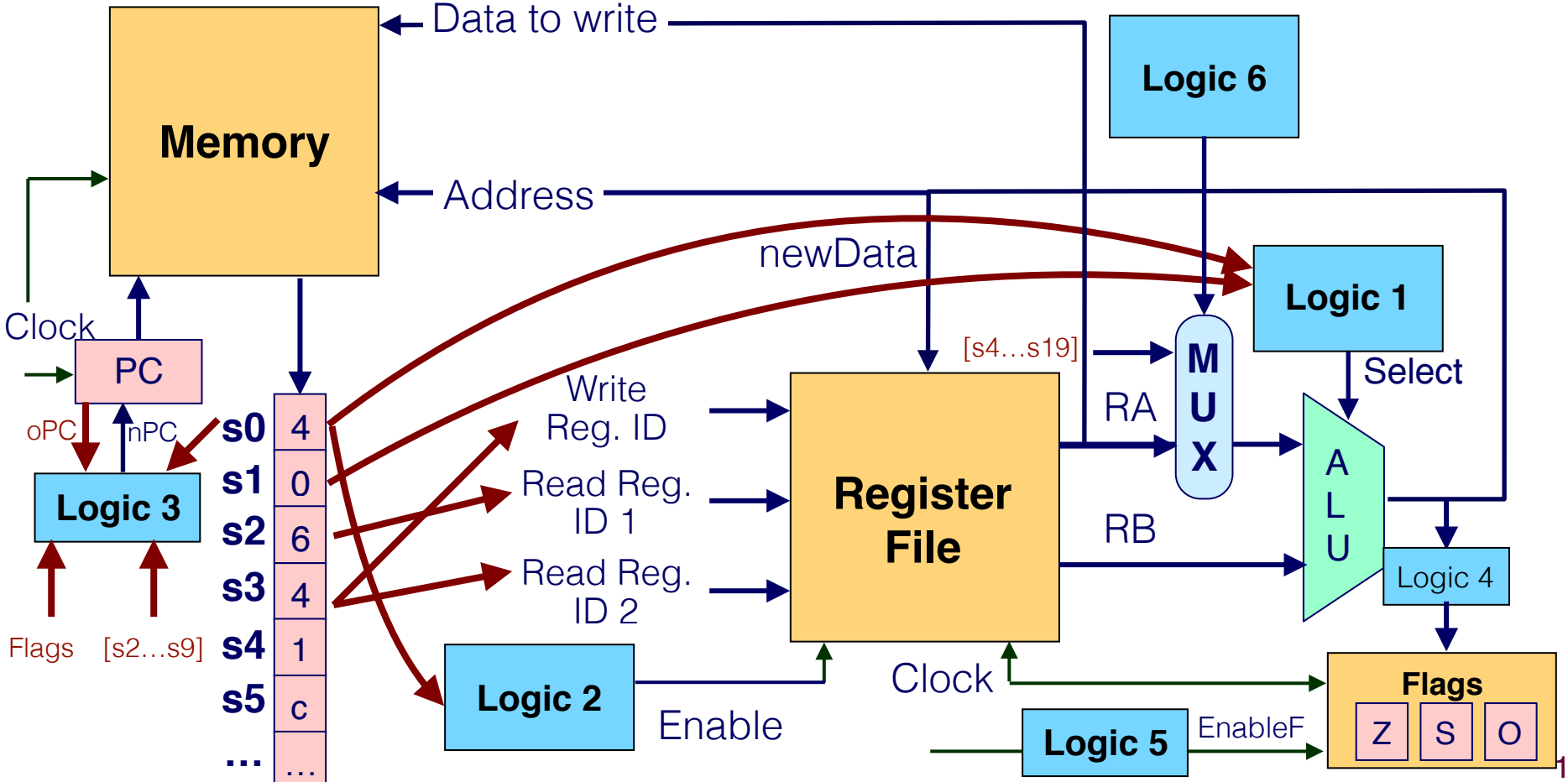
- Need new logic (Logic 6) to select the input to the ALU for Enable.



move rA to the memory address rB + D

`rmmovq rA, D(rB)`

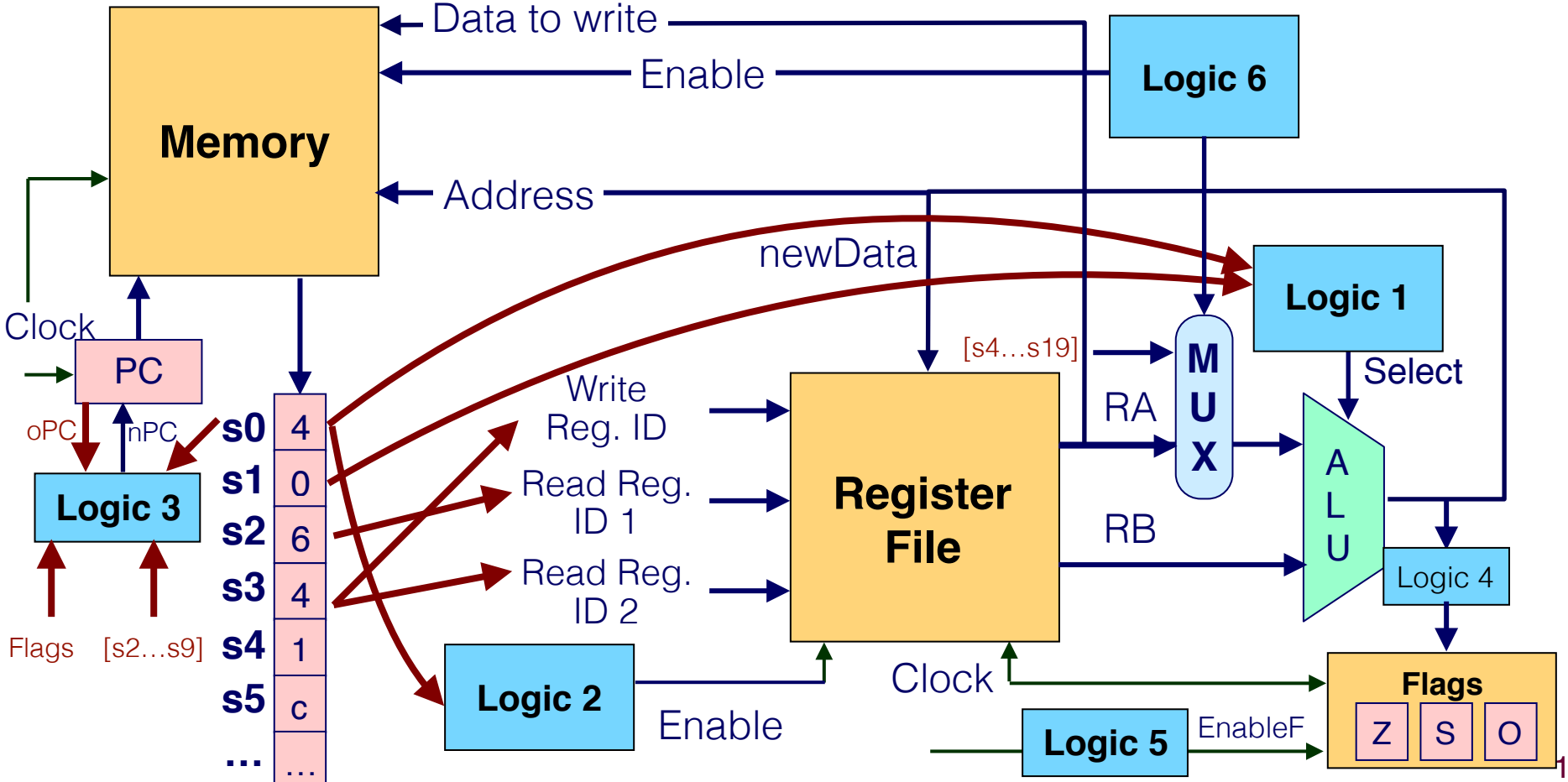
- Need new logic (Logic 6) to select the input to the ALU for Enable.



move rA to the memory address rB + D

`rmmovq rA, D(rB)`

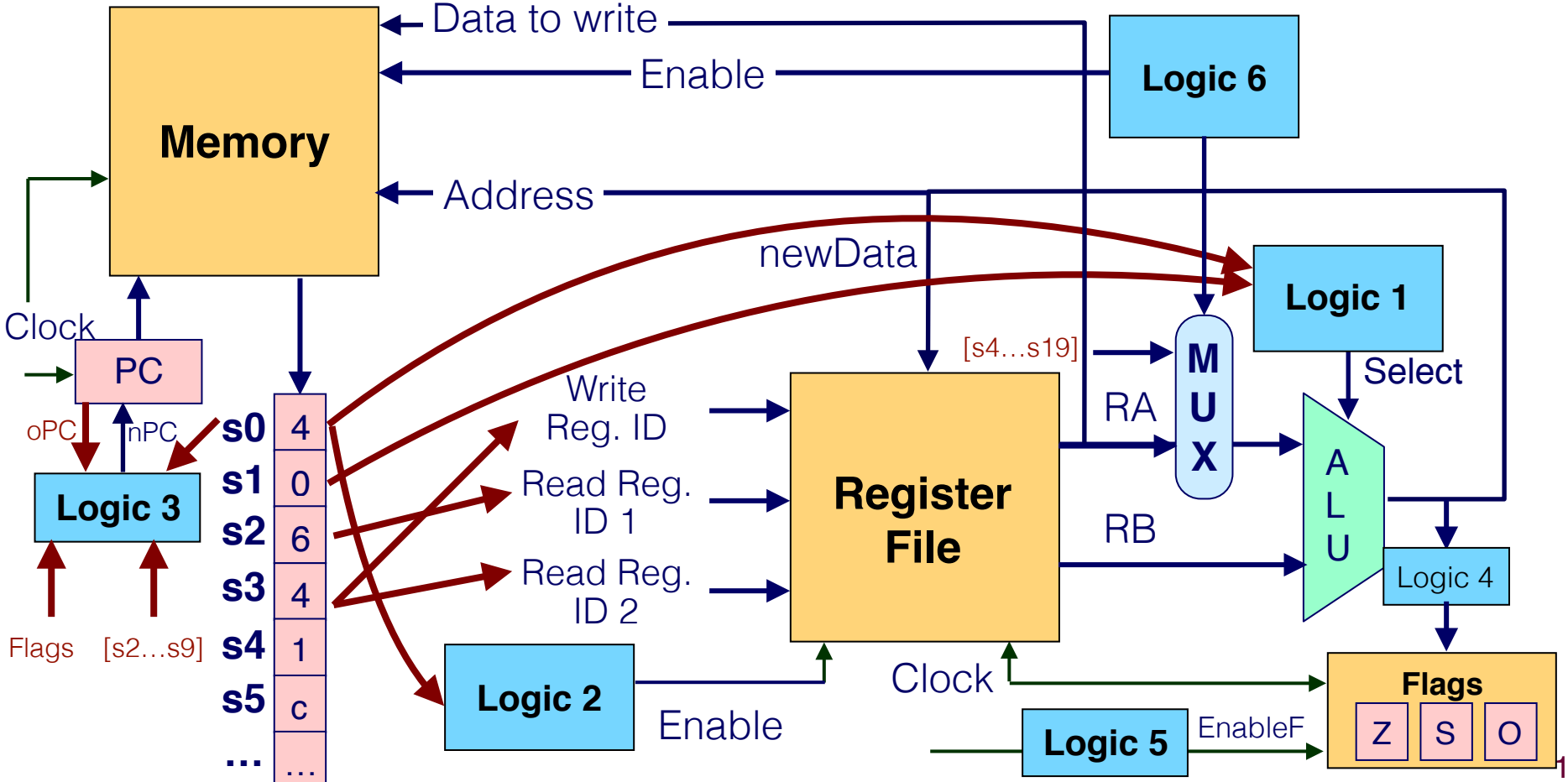
- Need new logic (Logic 6) to select the input to the ALU for Enable.



move rA to the memory address rB + D

`rmmovq rA, D(rB)`

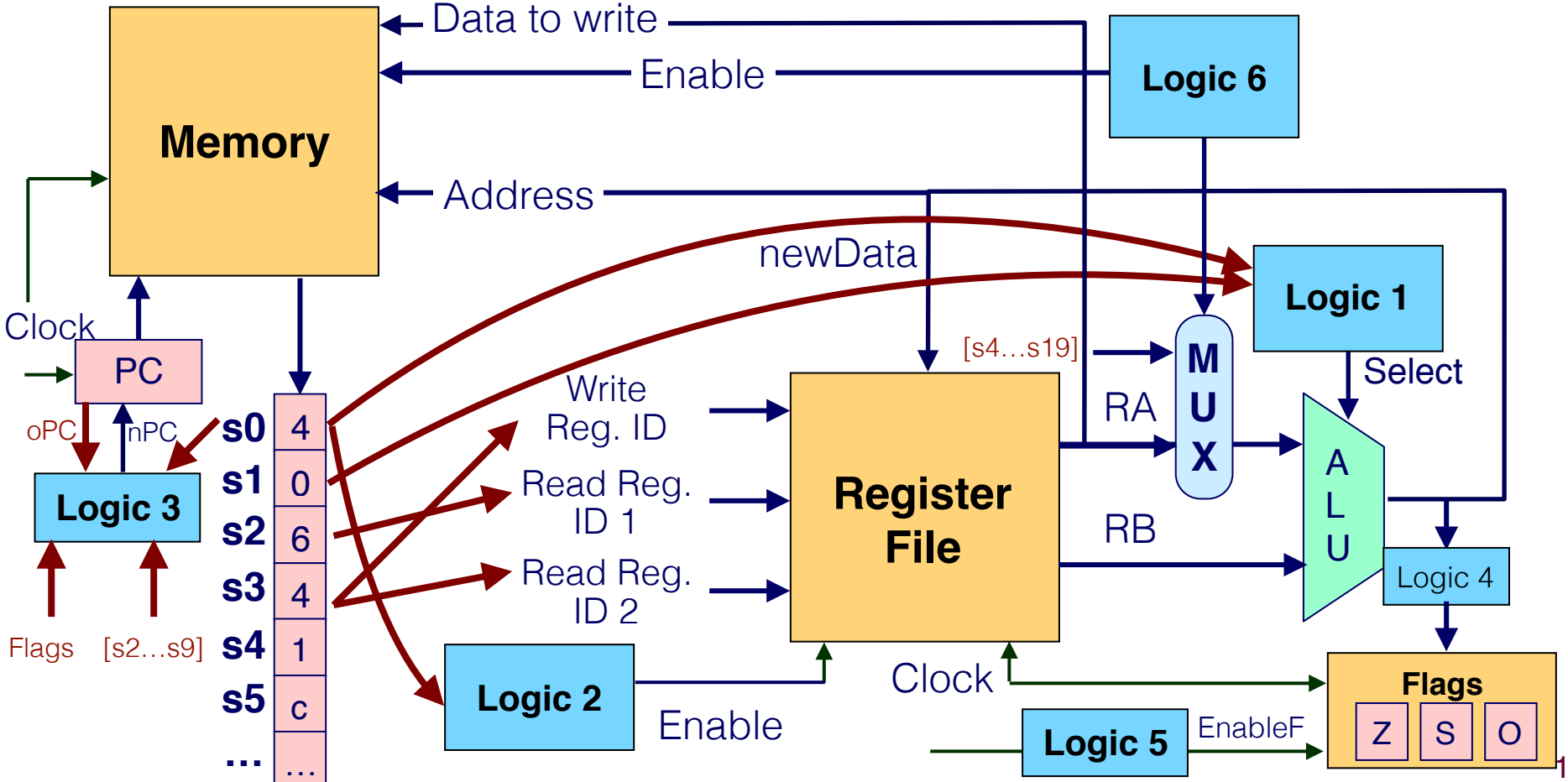
- Need new logic (Logic 6) to select the input to the ALU for Enable.
- How about other logics?



How About Memory to Register MOV?

move data at memory address rB + D to rA

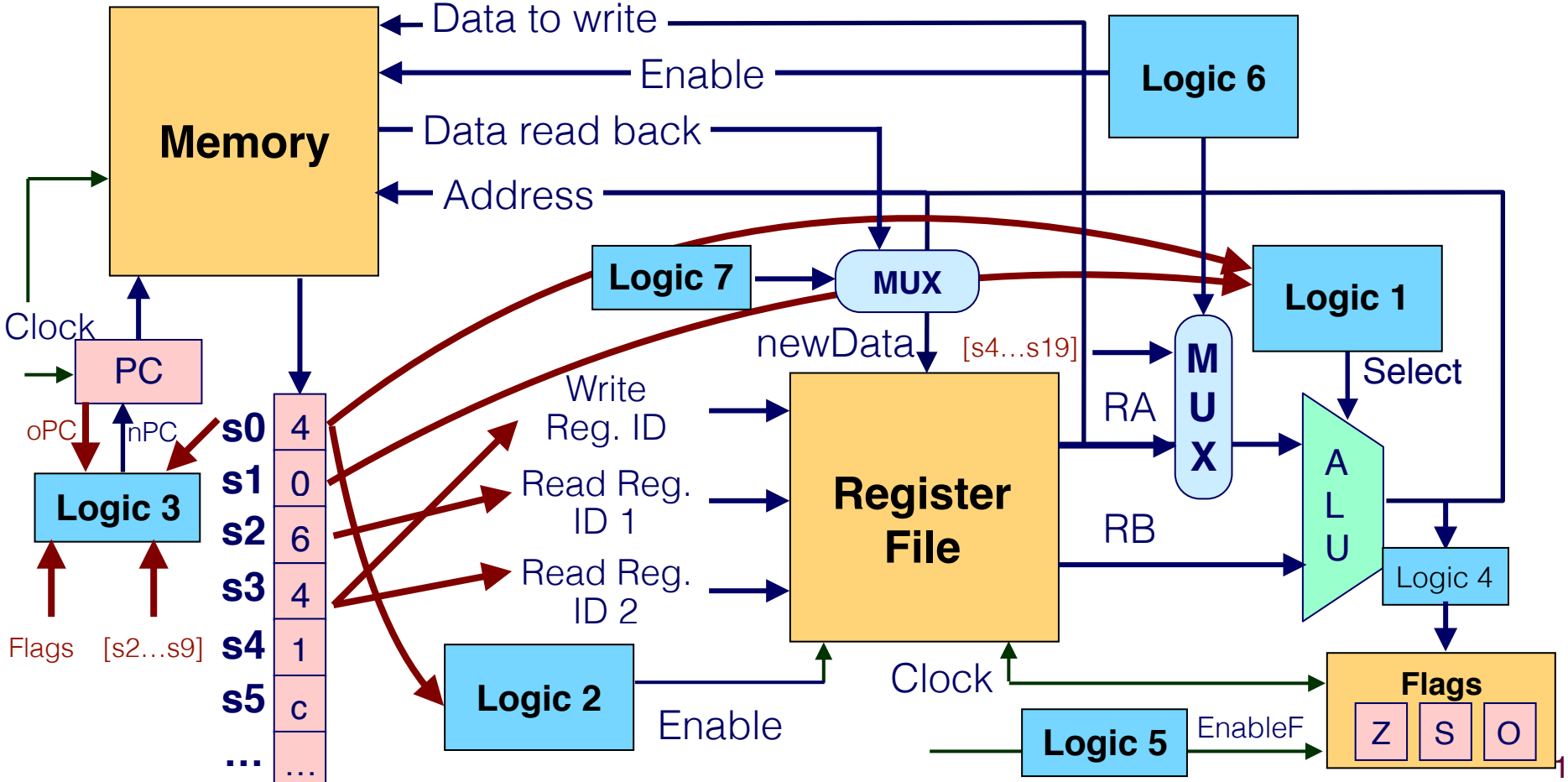
`mrmovq D(rB), rA`



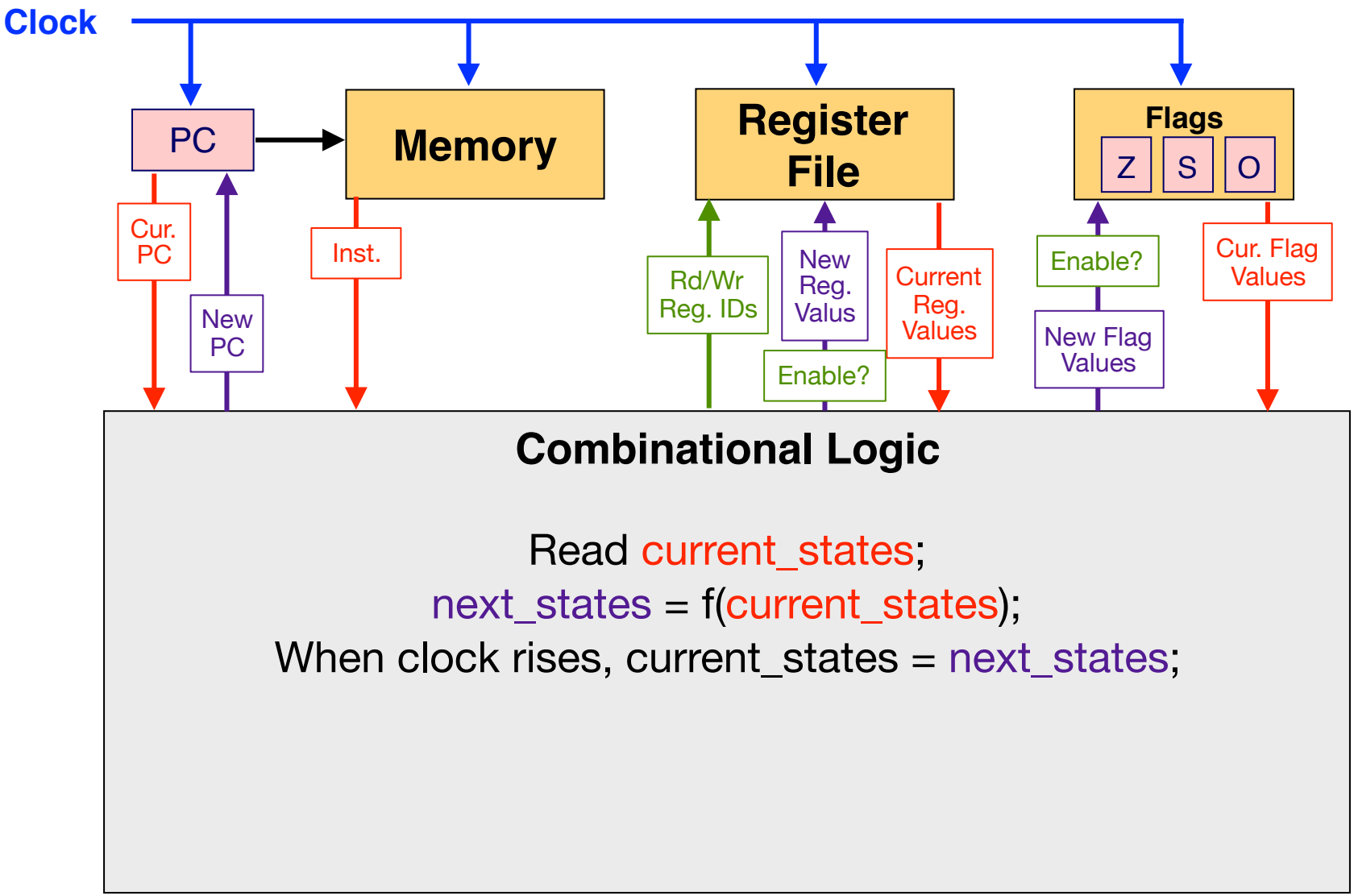
How About Memory to Register MOV?

move data at memory address rB + D to rA

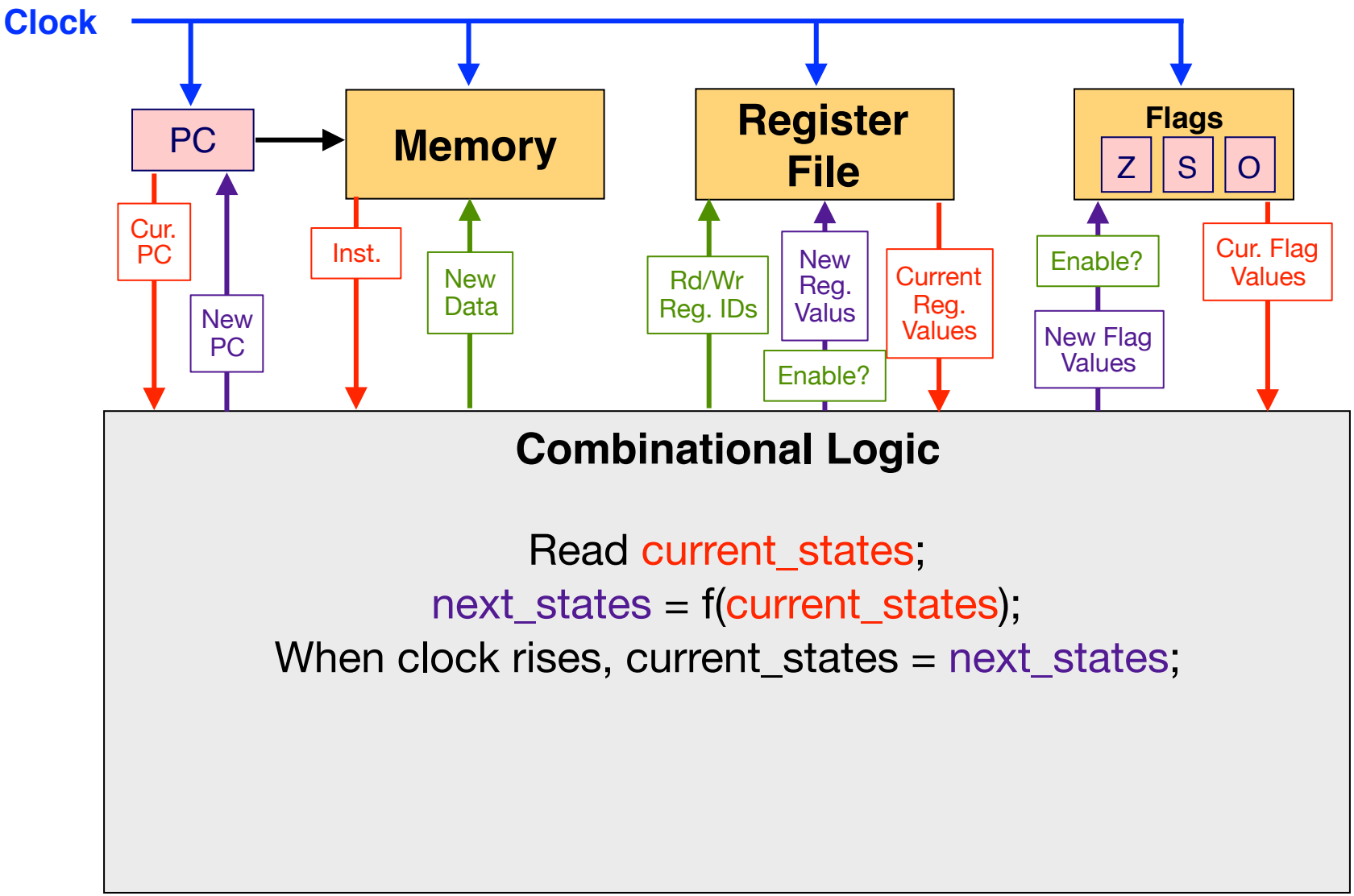
`mrmovq D(rB), rA`



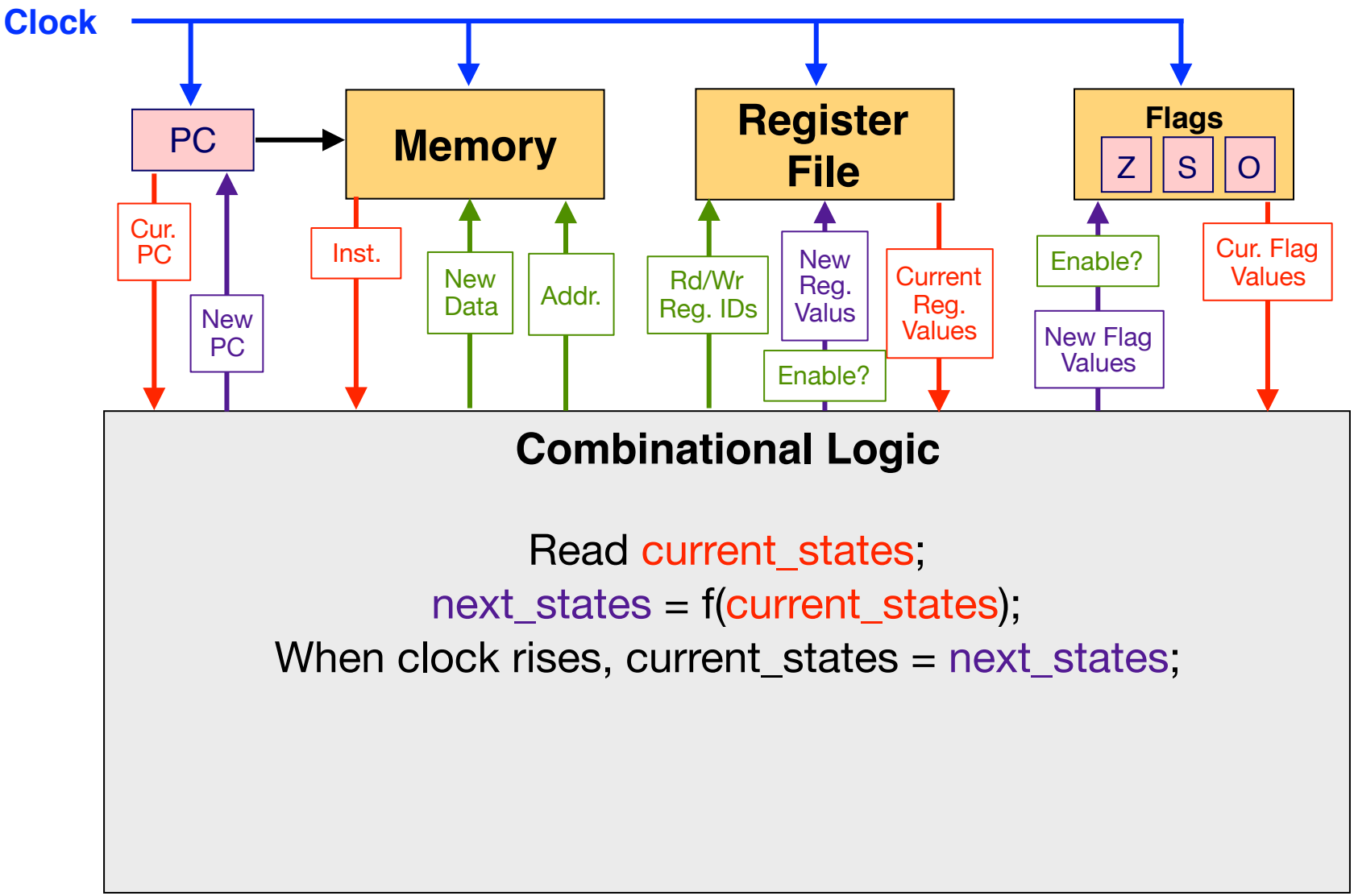
Microarchitecture (with MOV)



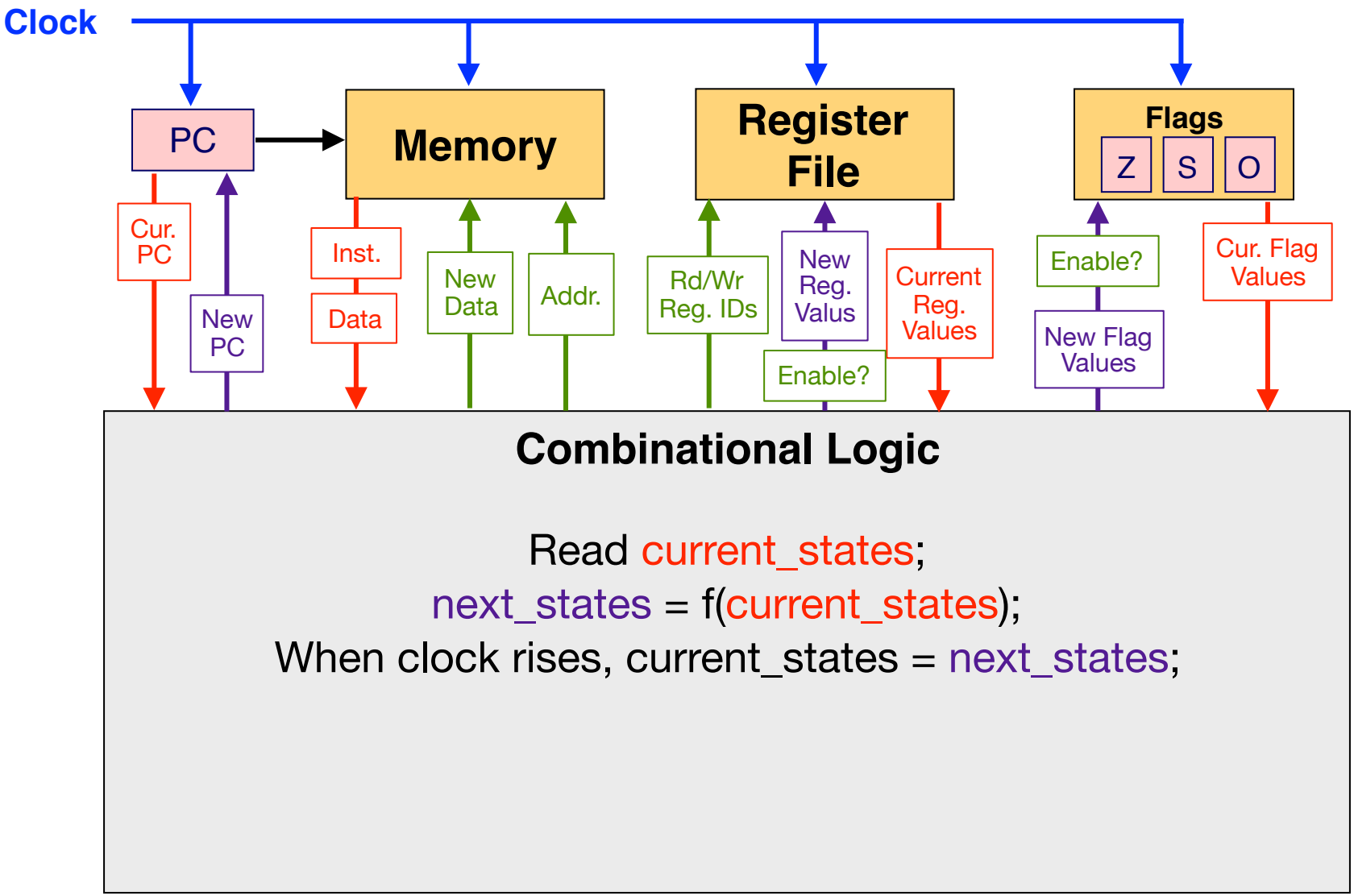
Microarchitecture (with MOV)



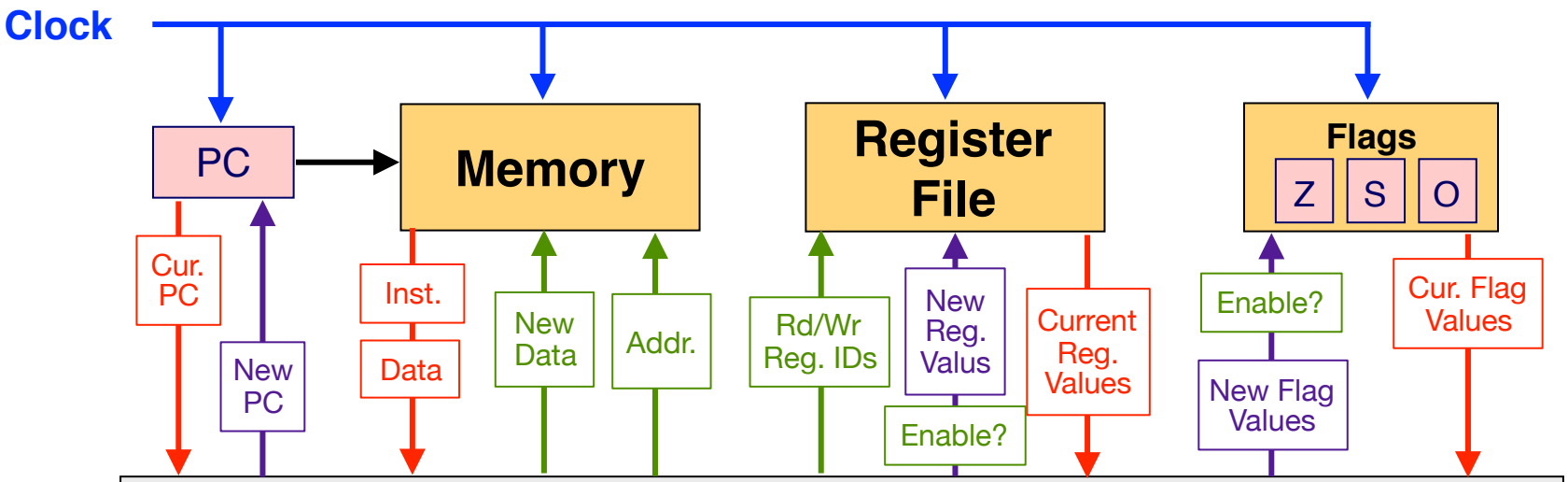
Microarchitecture (with MOV)



Microarchitecture (with MOV)



Microarchitecture (with MOV)



Combinational Logic

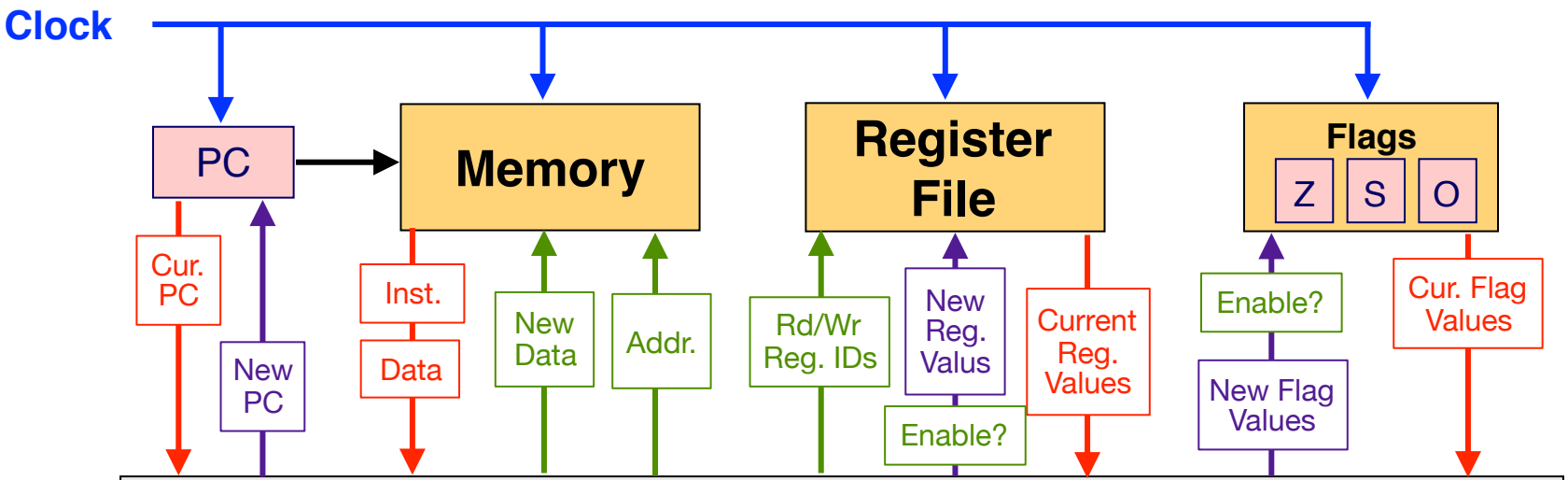
Read current_states;

next_states = f(current_states);

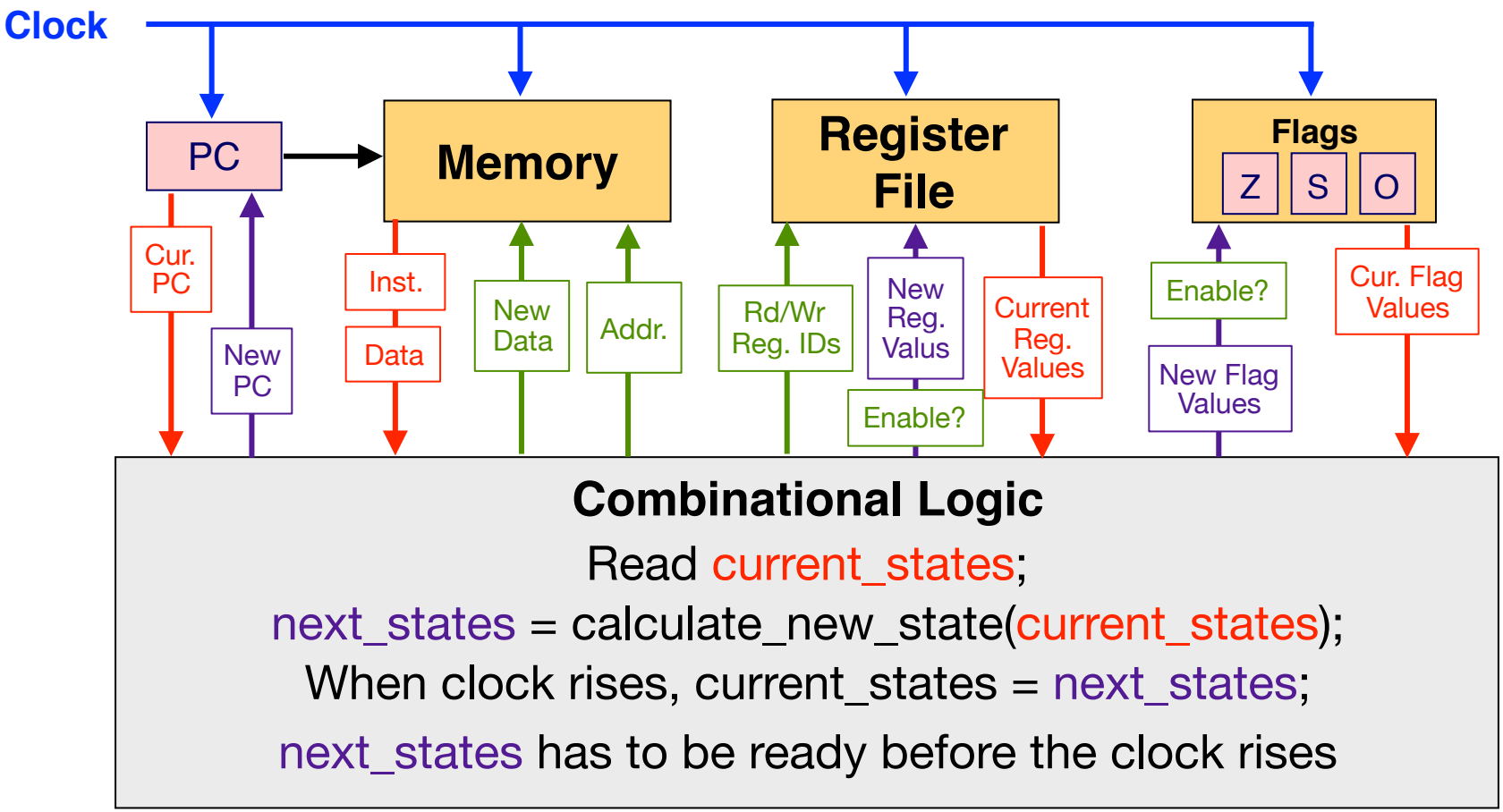
When clock rises, current_states = next_states;

next_states has to be ready before the clock rises

Single-Cycle Microarchitecture



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.

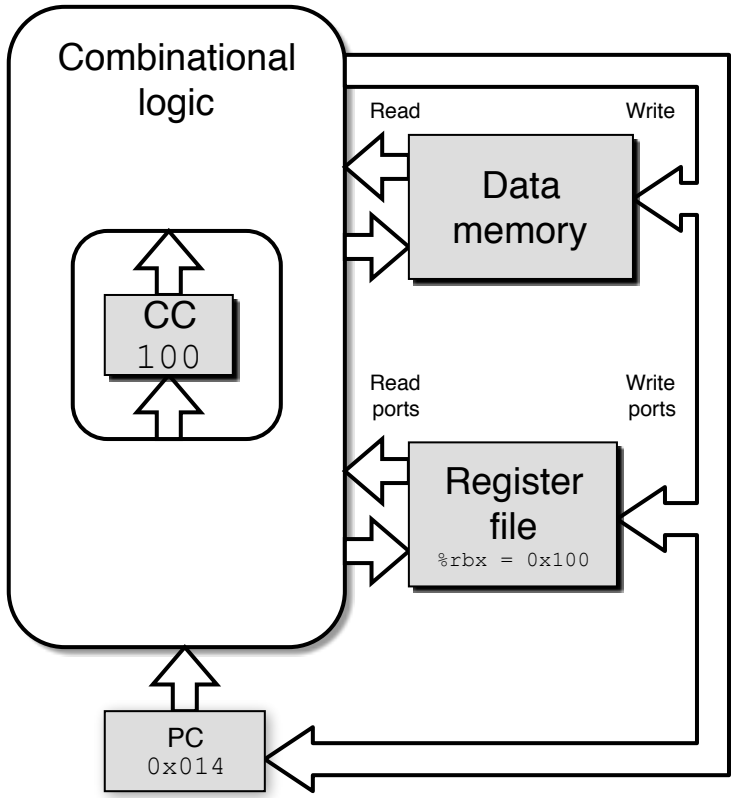
Single-Cycle Microarchitecture: Illustration

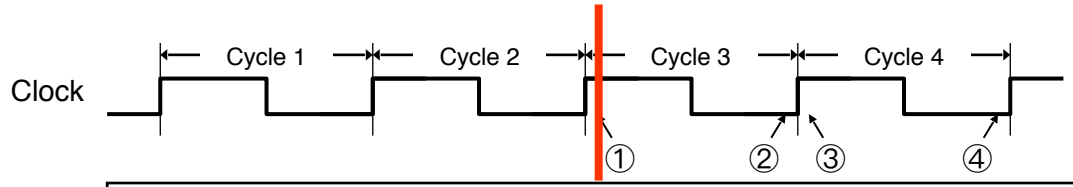
Think of it as a state machine

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

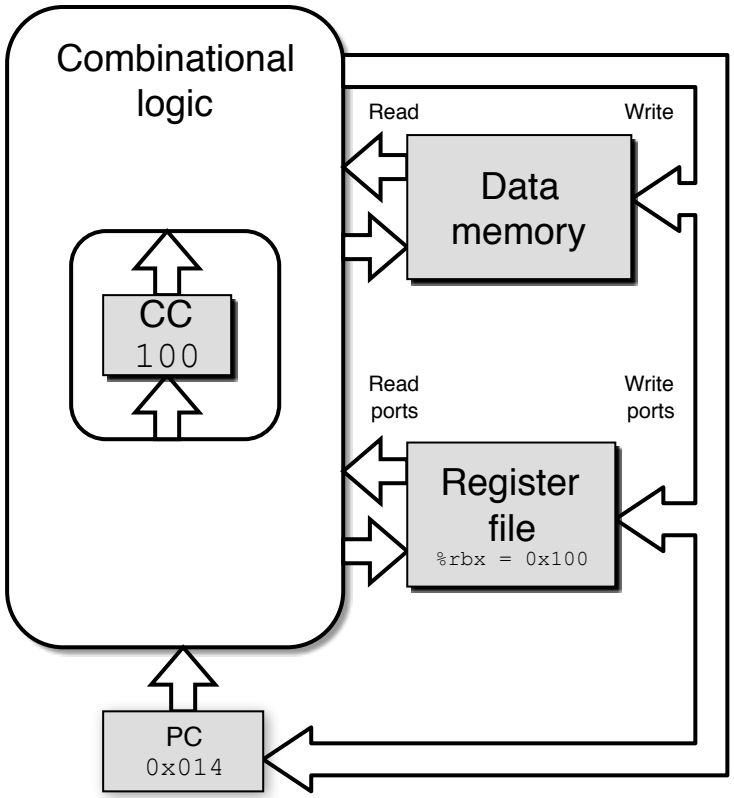
States (All updated as clock rises)

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

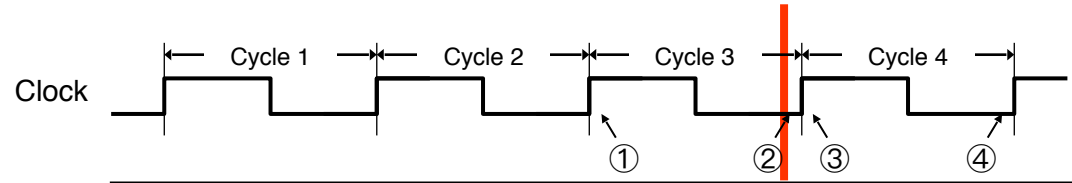




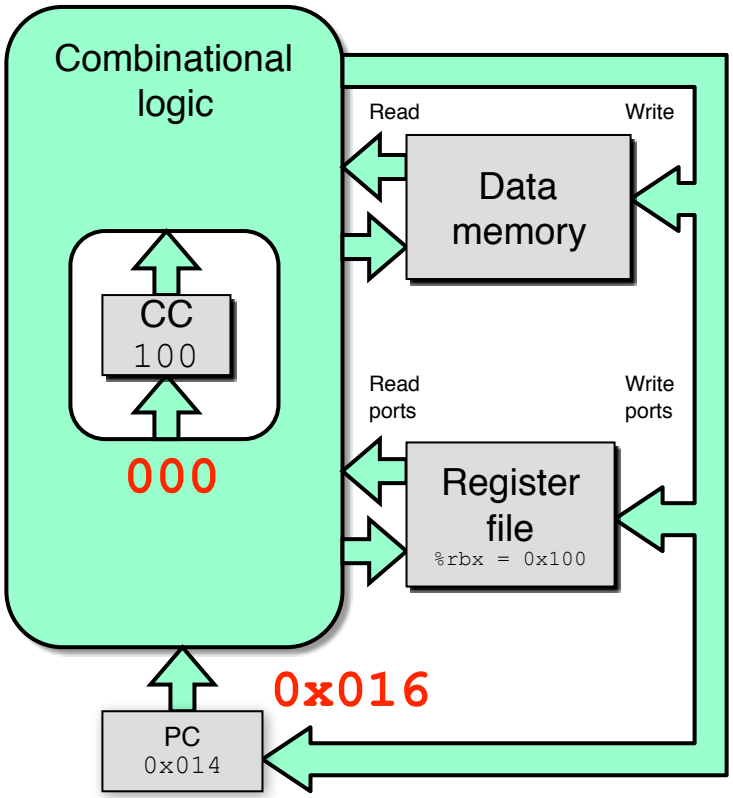
Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



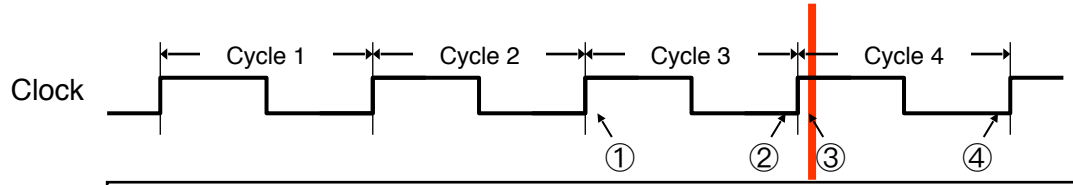
- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes



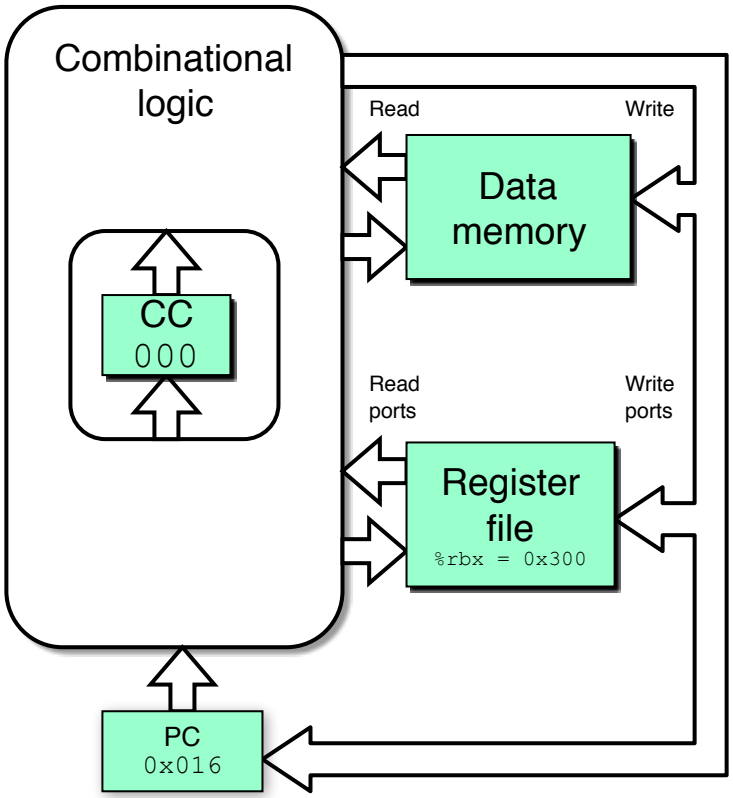
Cycle 1:	`0x000:` `irmovq $0x100,%rbx`	`# %rbx <-- 0x100`
Cycle 2:	`0x00a:` `irmovq $0x200,%rdx`	`# %rdx <-- 0x200`
Cycle 3:	`0x014:` `addq %rdx,%rbx`	`# %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:` `je dest`	`# Not taken`
Cycle 5:	`0x01f:` `rmmovq %rbx,0(%rdx)`	`# M[0x200] <-- 0x300`



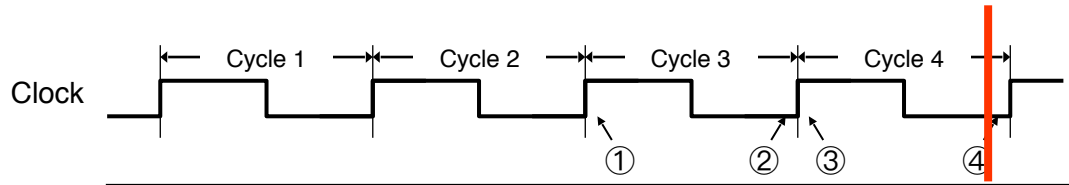
- state set according to second `irmovq` instruction
- combinational logic generates results for addq instruction



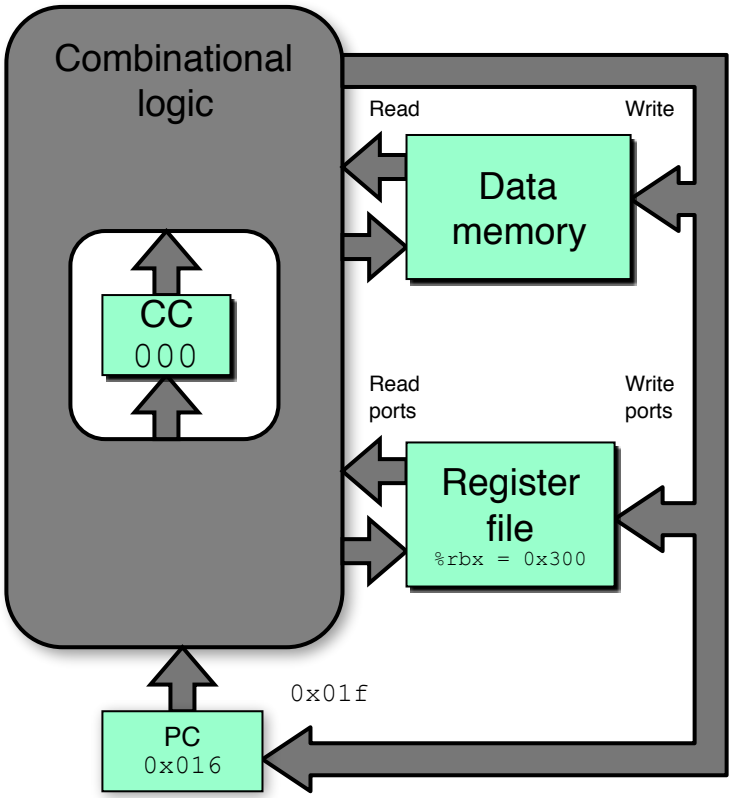
Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



- 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**

X # of cycles taken to execute an instruction (on average)

/ number of cycles per second

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**

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).

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

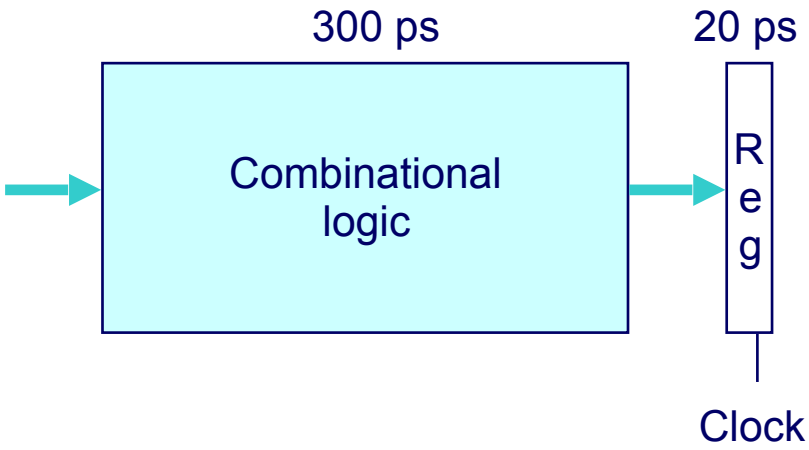
Limitations of a Single-Cycle CPU

- 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

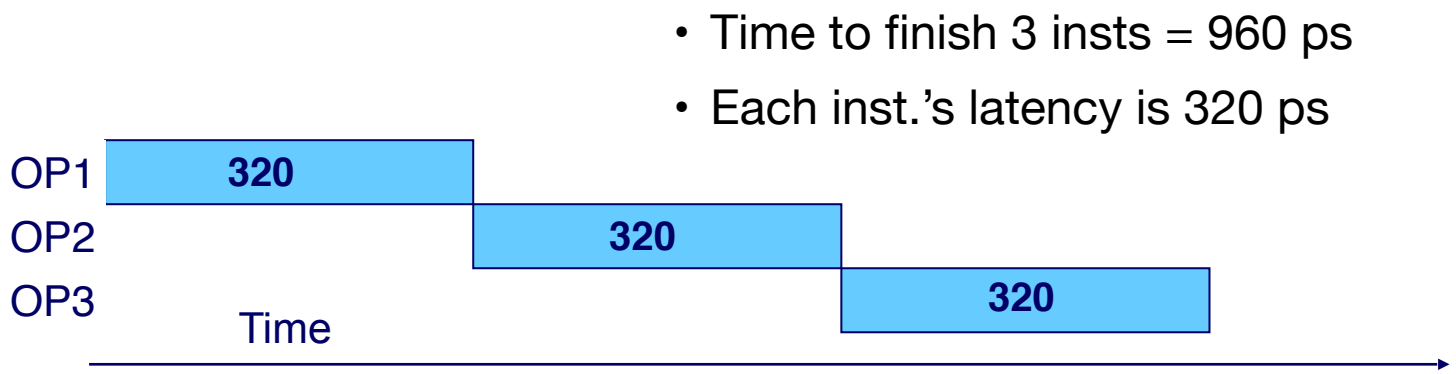
- Cycle time
  - Every instruction finishes in one cycle.
  - The absolute time takes to execute each instruction varies. Consider for instance an ADD instruction and a JMP instruction.
  - But the cycle time is uniform across instructions, so the cycle time needs to accommodate the worst case, i.e.,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



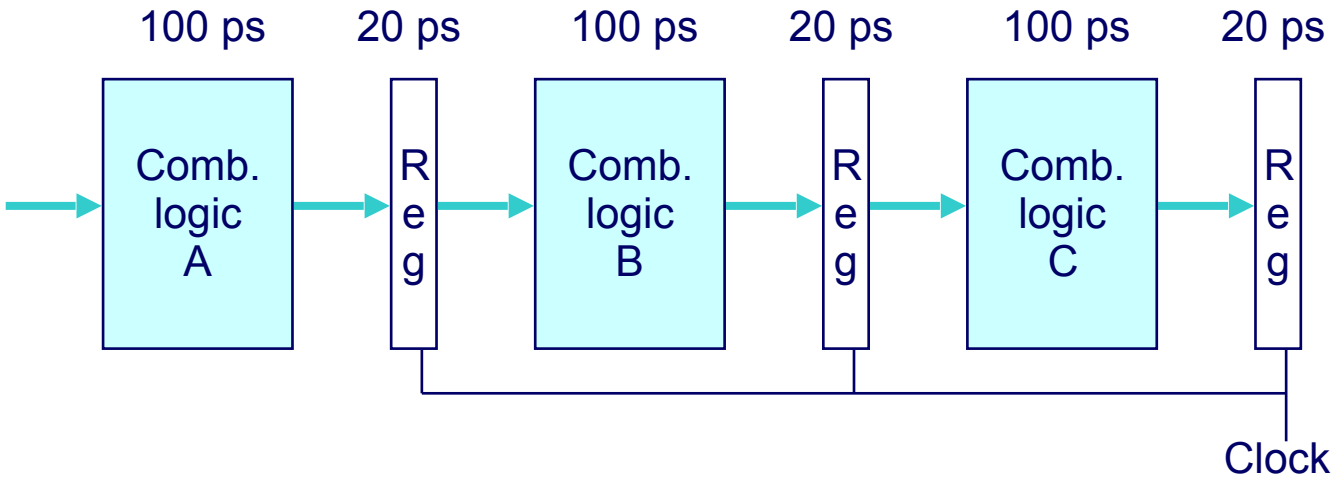
- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle time of at least 320 ps

Pipeline Diagrams



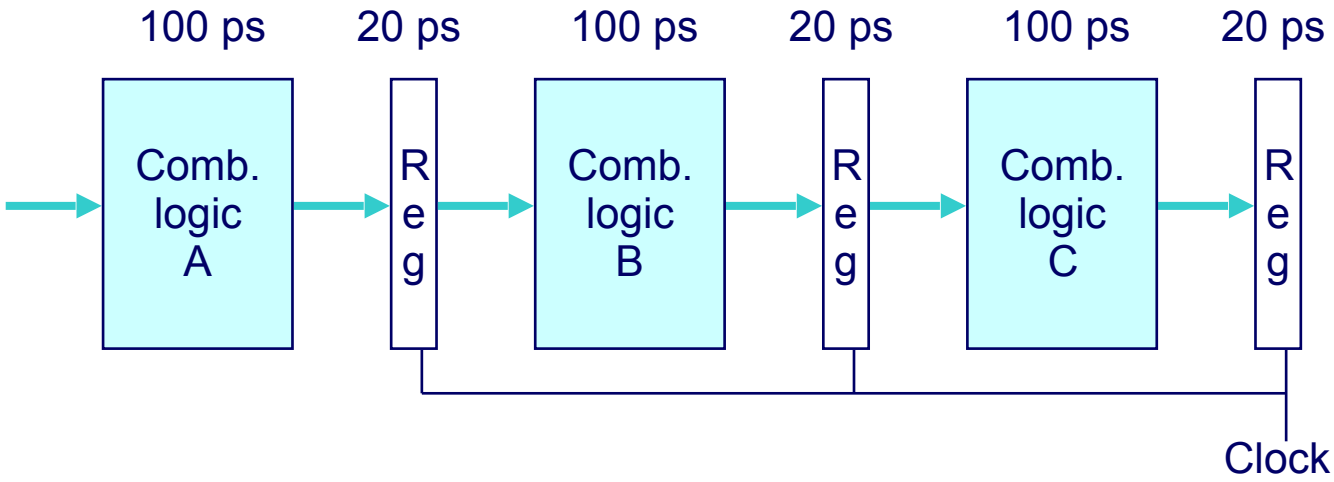
- 3 instructions will take 960 ps to finish
  - First cycle: Inst 1 takes 300 ps to compute new state, 20 ps to store the new states
  - Second cycle: Inst 2 starts; it takes 300 ps to compute new states, 20 ps to store new states
  - And so on…

3-Stage Pipelined Version

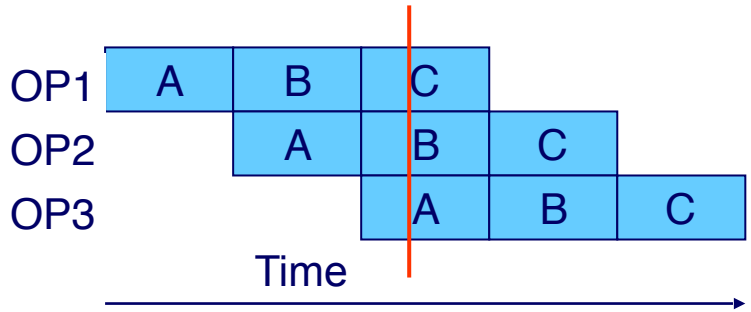


- Divide combinational logic into 3 stages of 100 ps each
- Insert registers between stages to store intermediate data between stages. These are call pipeline registers (ISA-invisible)
- Can begin a new instruction as soon as the previous one finishes stage A and has stored the intermediate data.
  -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

3-Stage Pipelined Version



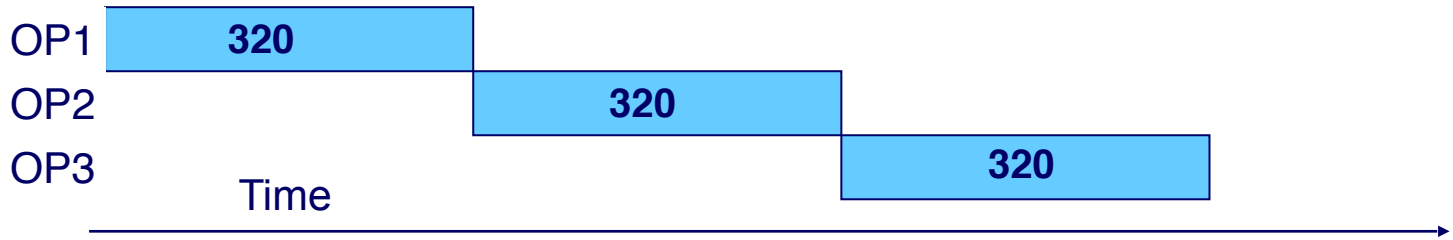
3-Stage Pipelined



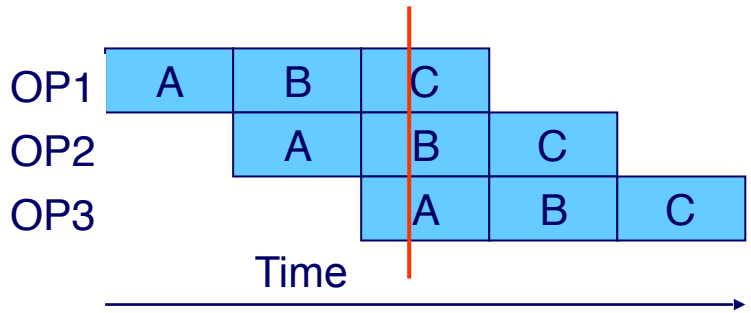
Comparison

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