

CSC 252: Computer Organization

Spring 2019: Lecture 13

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:

- **Assignment 3 is due March 1, midnight *****

Announcement

- Lab 3 is due on **March 1, midnight**
- If you had finished Lab3 without cookie issues you don't have to do it again.
- If you had cookie issues, most likely you worked in a pair, then generate a new cookie, and you have until March 11 midnight to submit Lab 3.

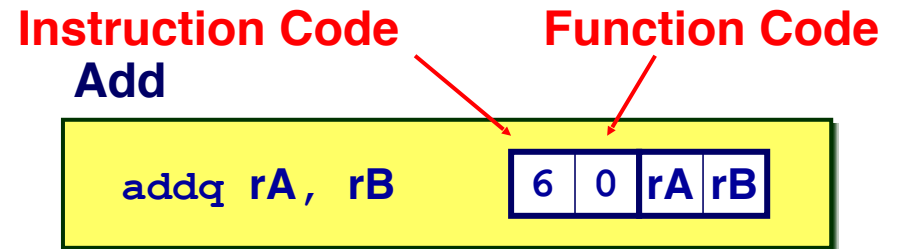
Announcement

- Mid-term exam: **March 7**; in class
- Past exam & Problem set: <http://www.cs.rochester.edu/courses/252/spring2019/handouts.html>
- Anything on paper is allowed (book, notes, past exams, etc.)

MON 25	TUE 26	WED 27	THU 28	FRI Mar 1
4	5	6	7	8
	Lecture		Today ↓ Midterm	A3 due*

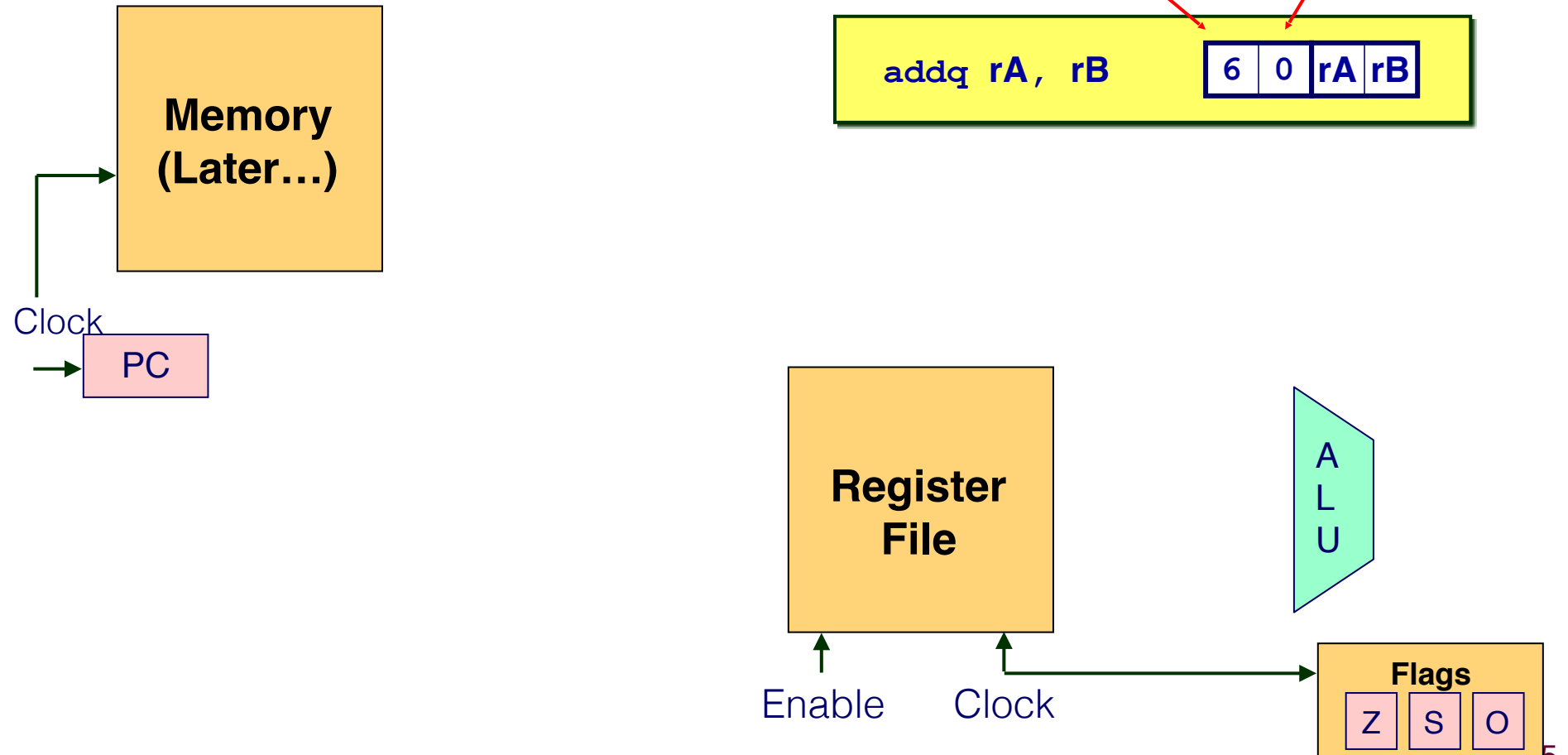
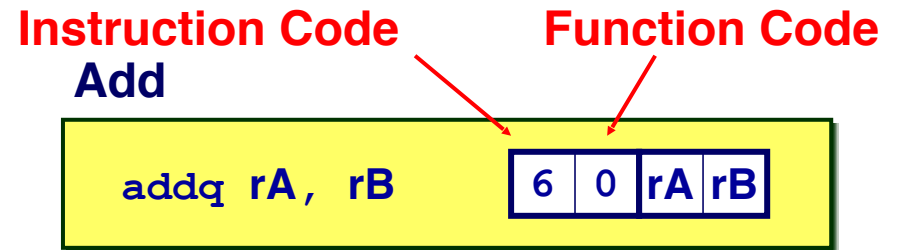
Executing an ADD instruction

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



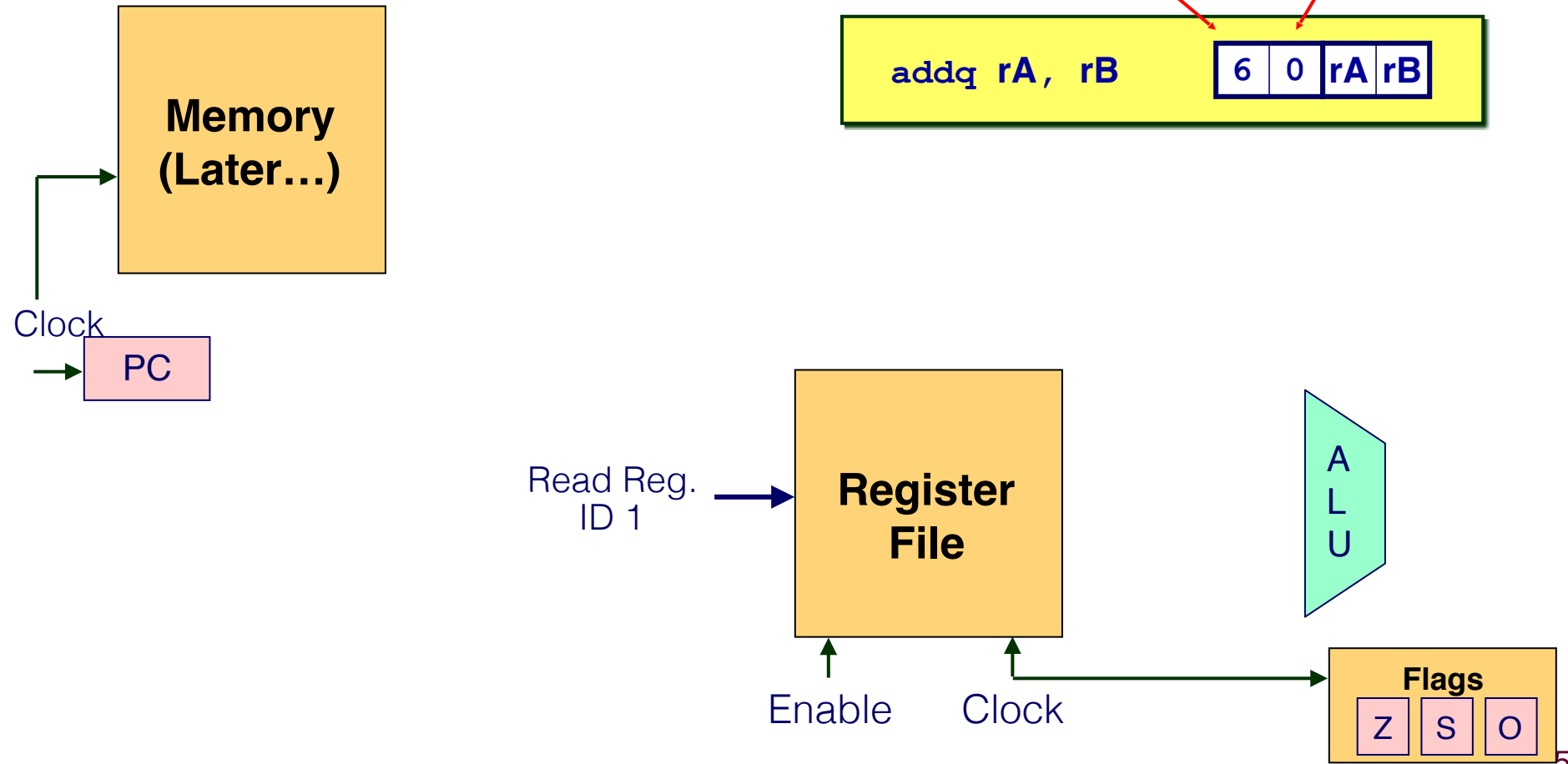
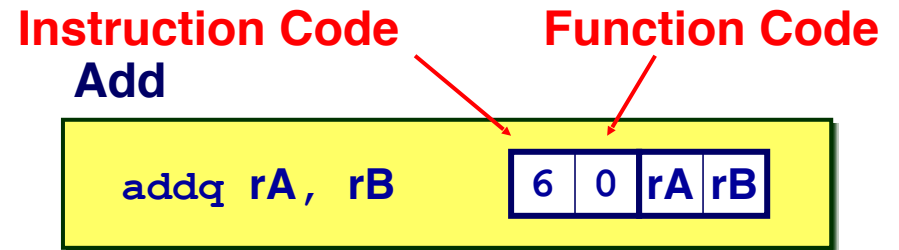
Executing an ADD instruction

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



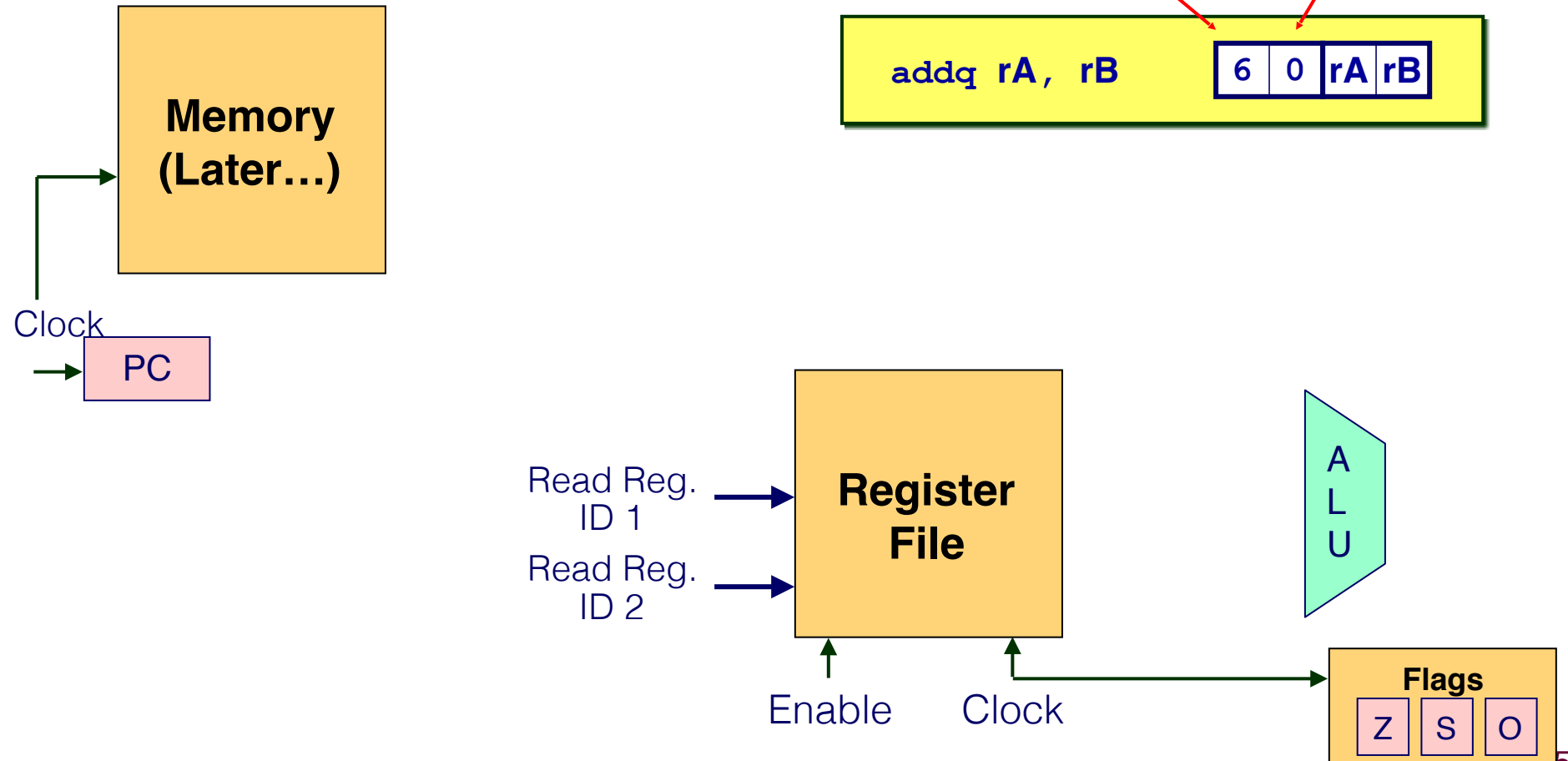
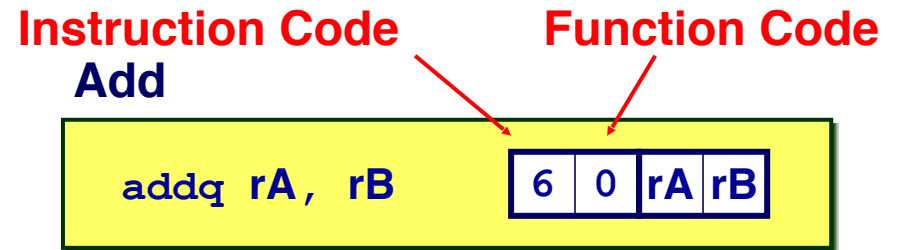
Executing an ADD instruction

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



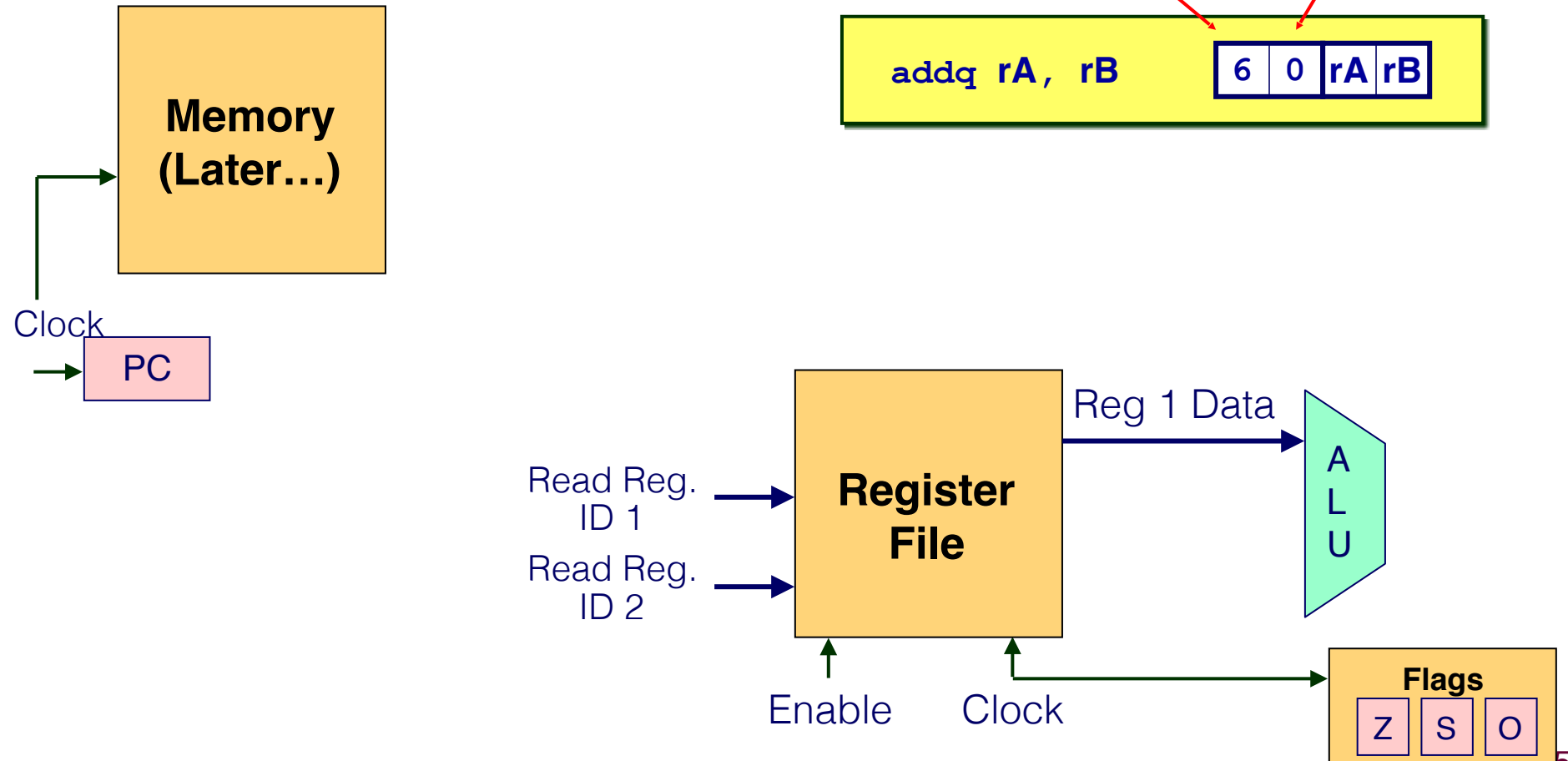
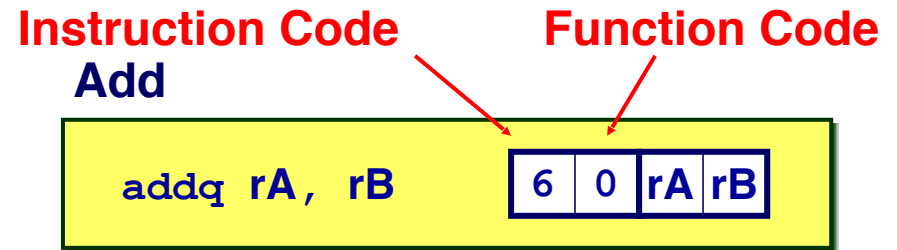
Executing an ADD instruction

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



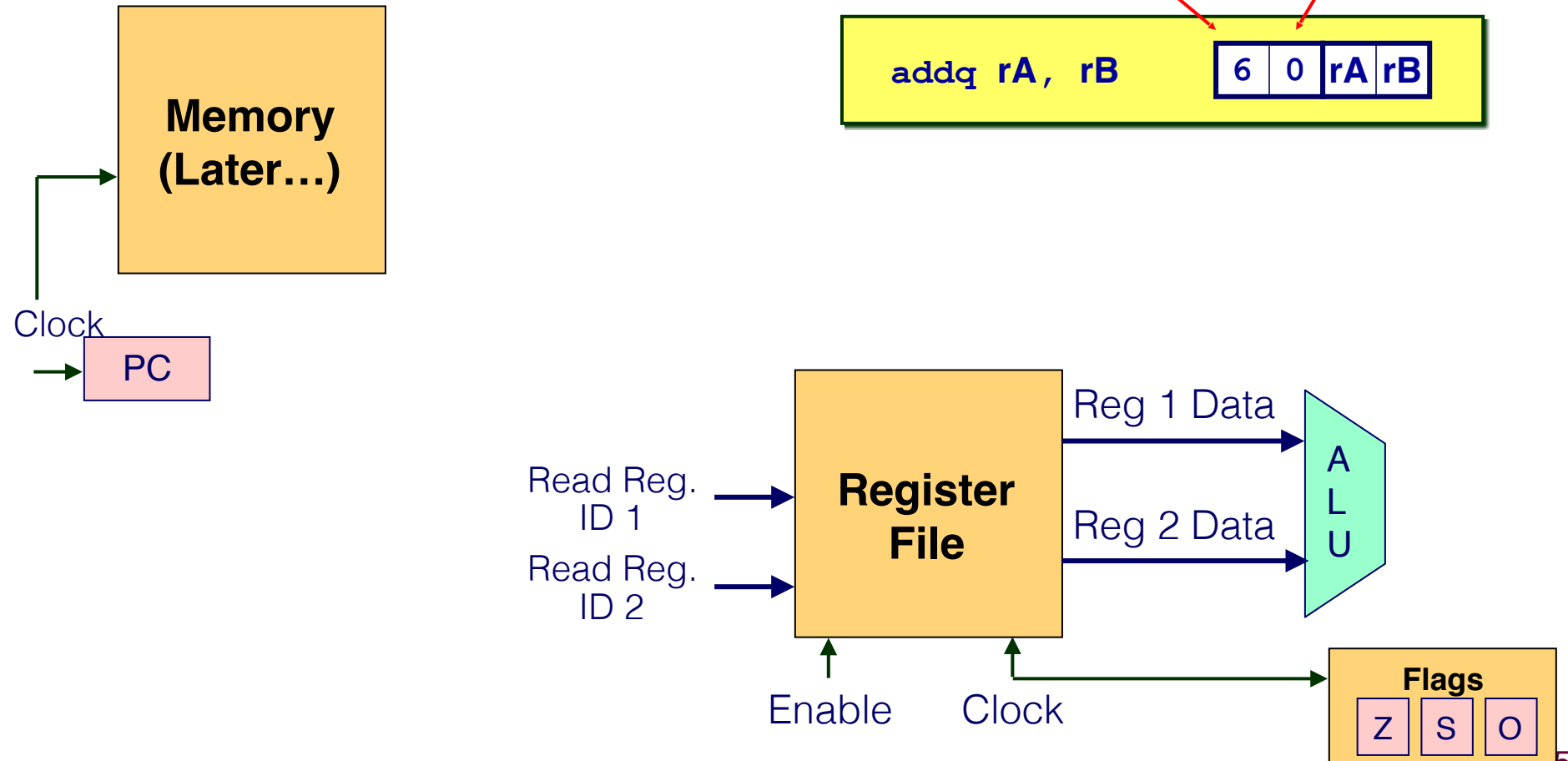
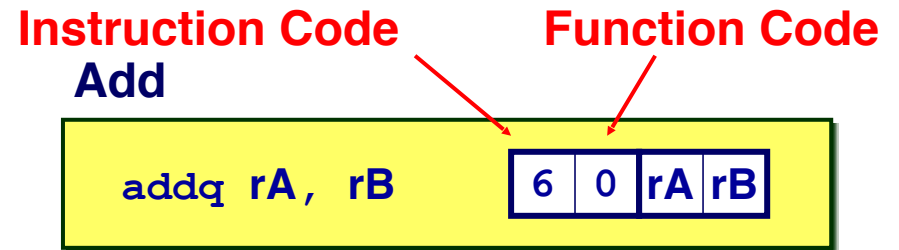
Executing an ADD instruction

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



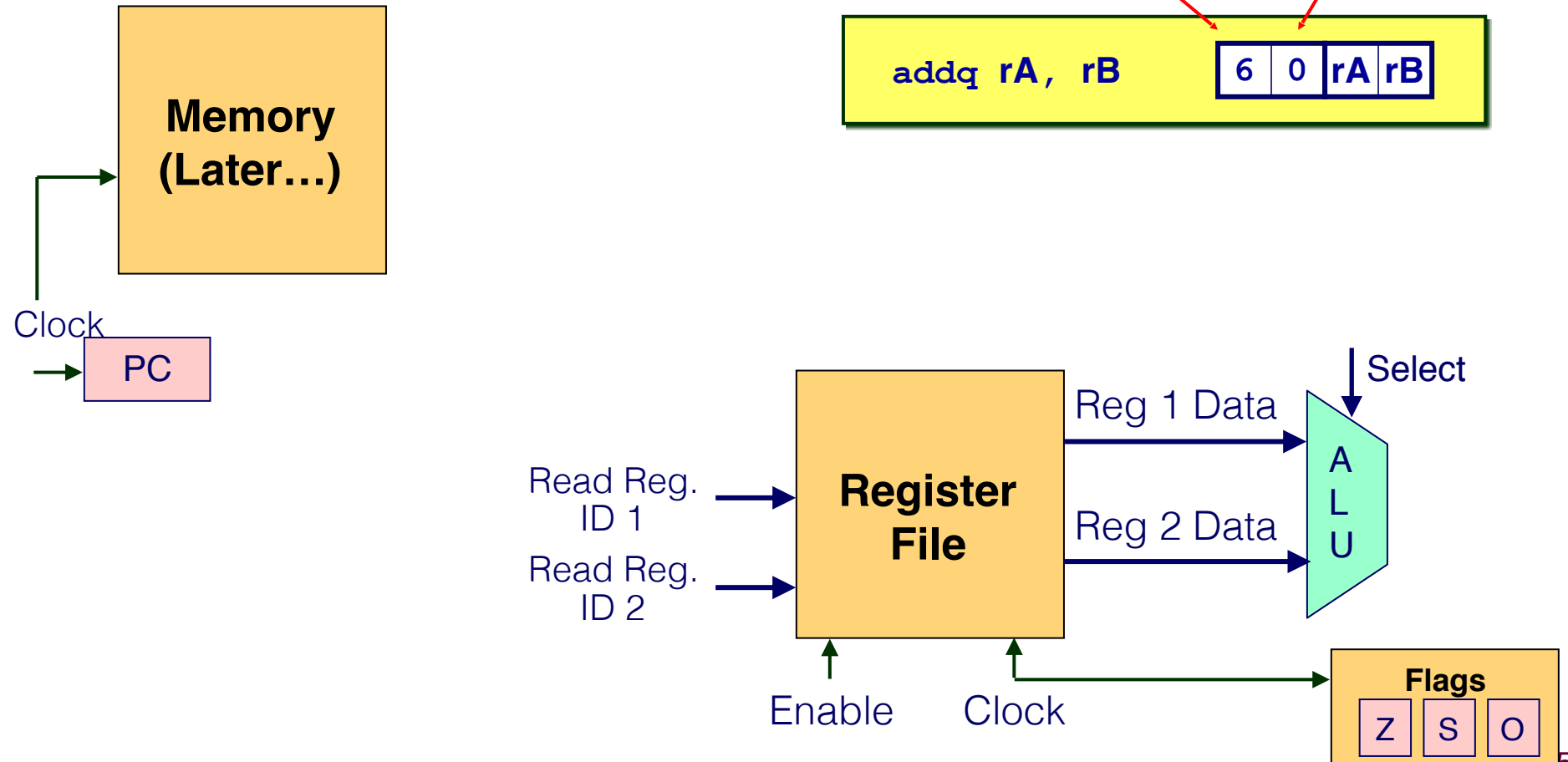
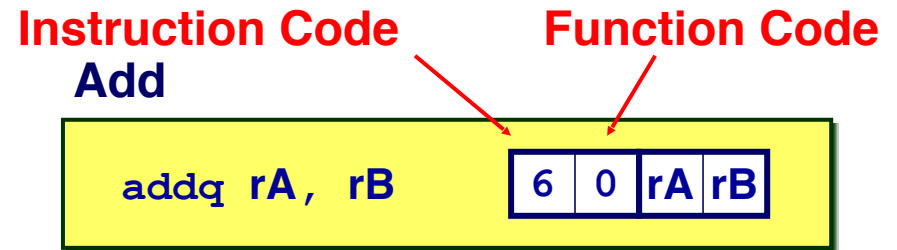
Executing an ADD instruction

- 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`

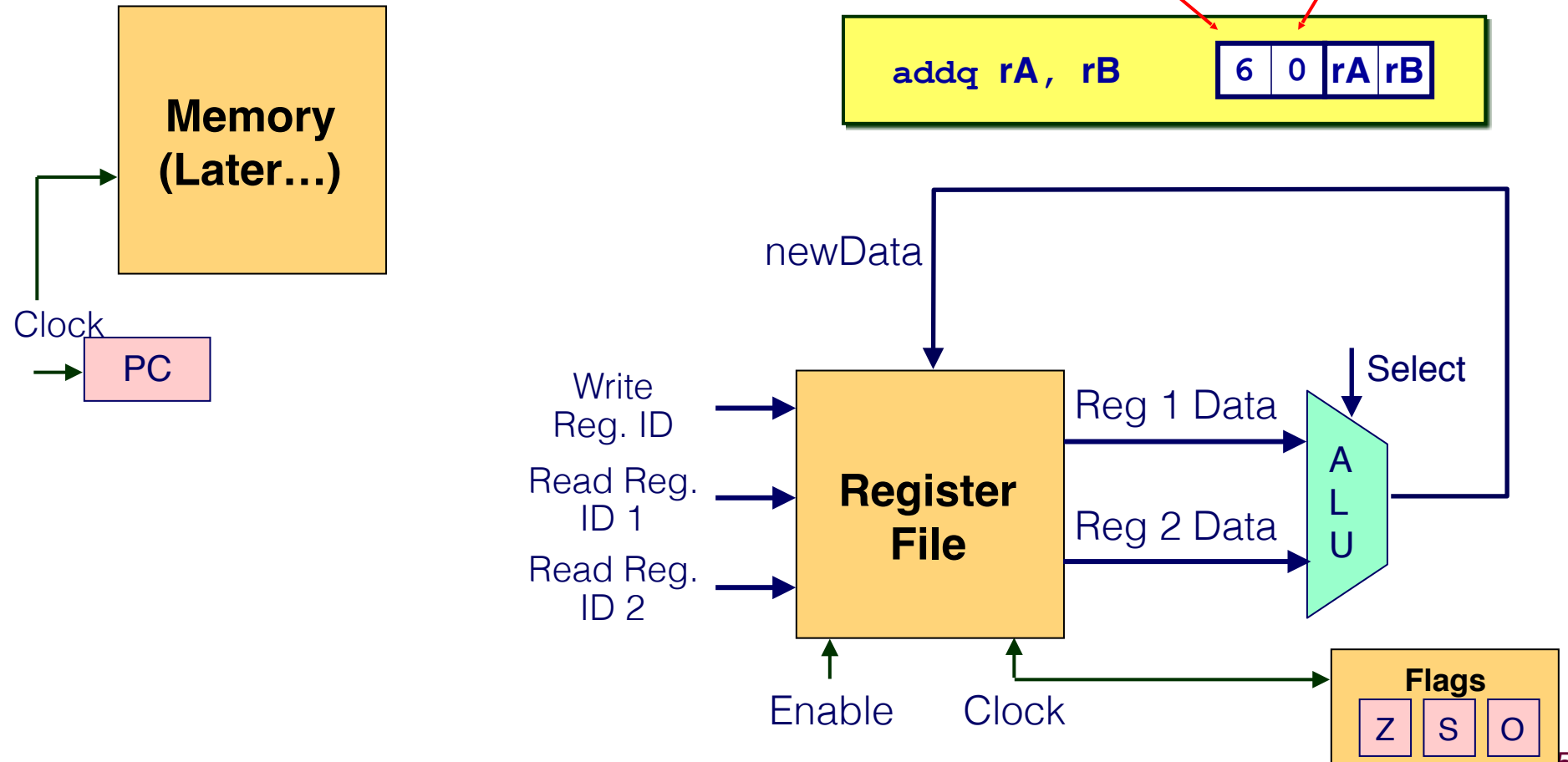


Executing an ADD instruction

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

Instruction Code
Add

Function Code

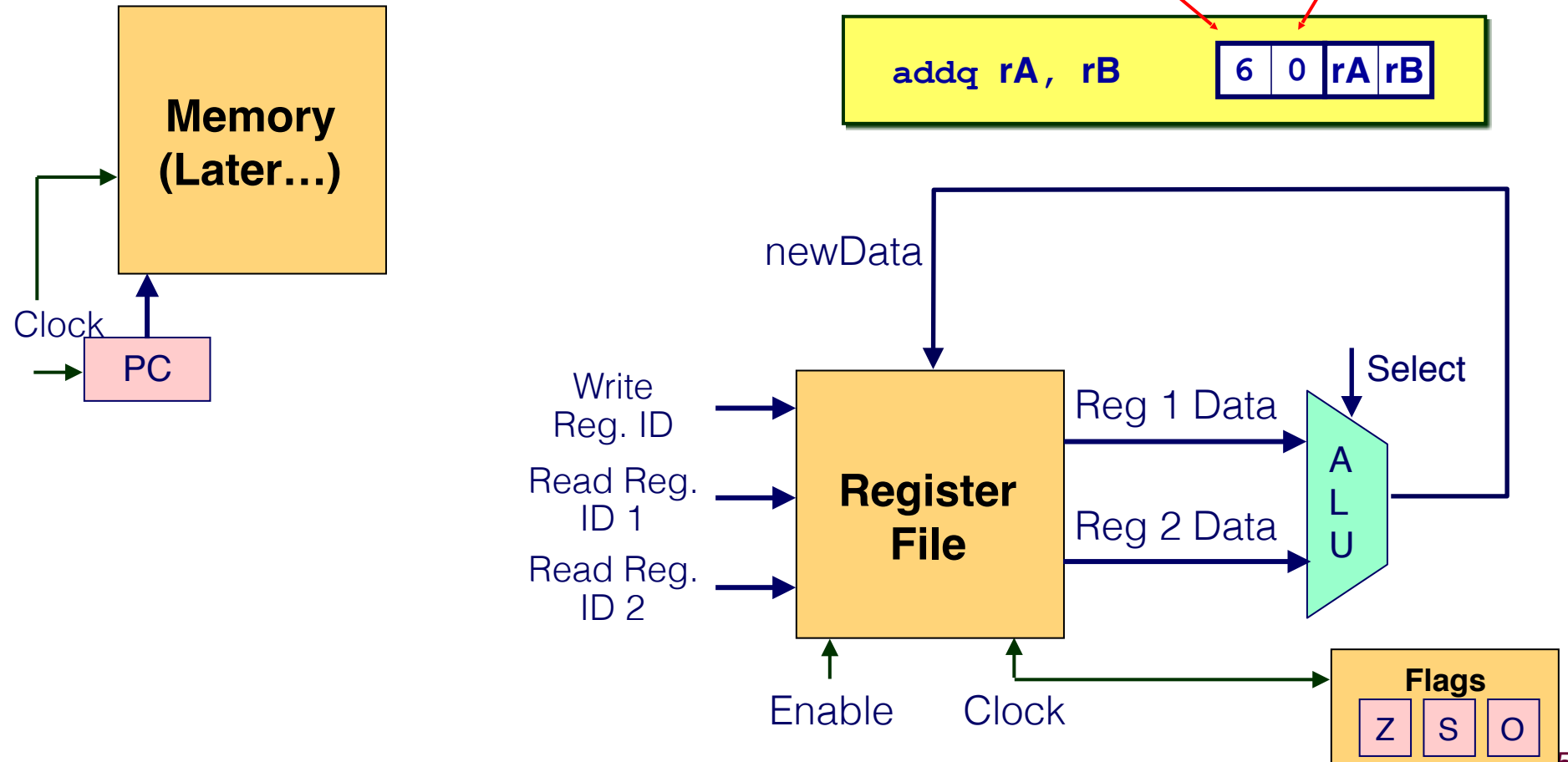


Executing an ADD instruction

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

Instruction Code
Add

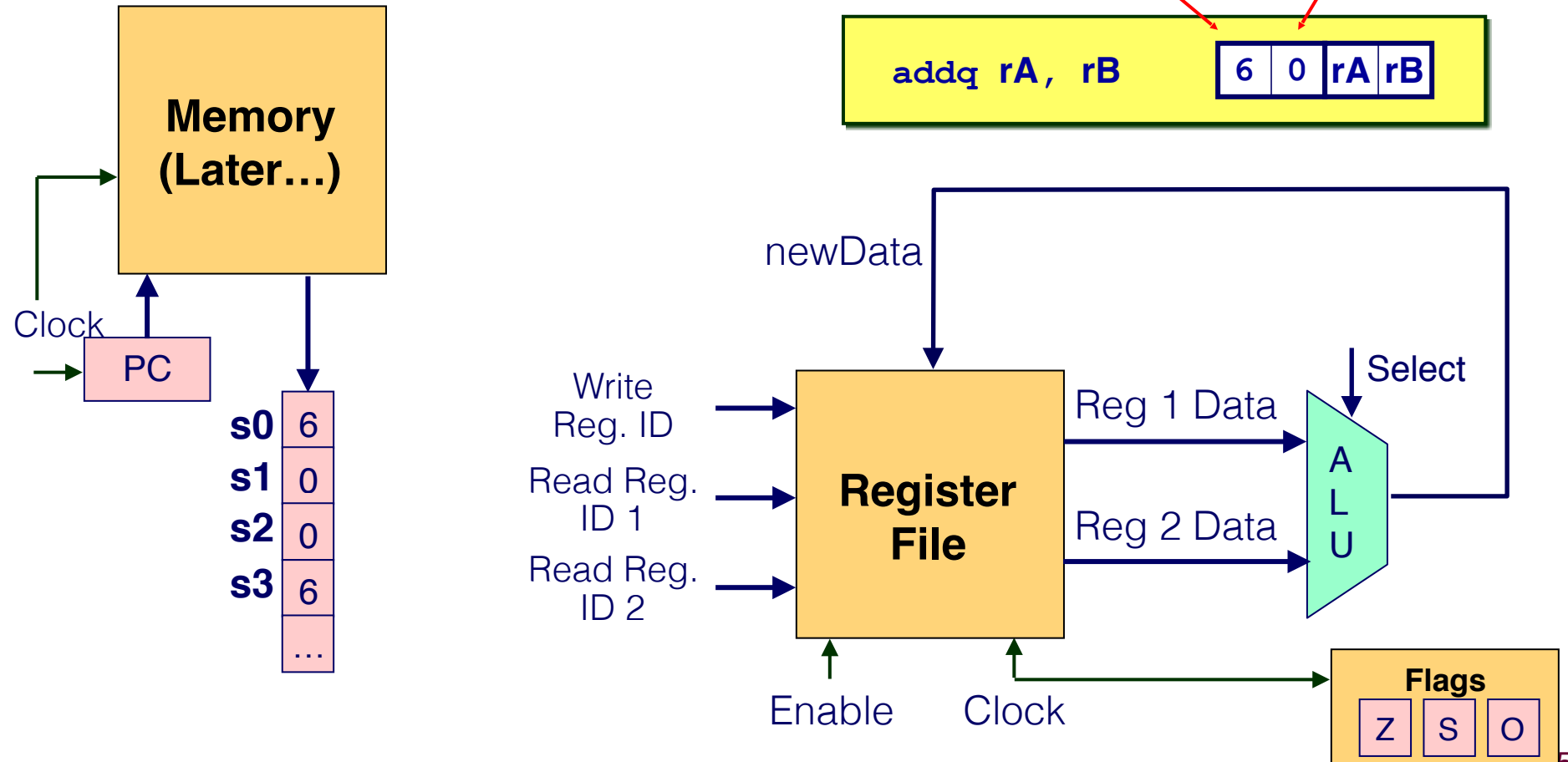
Function Code



Executing an ADD instruction

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

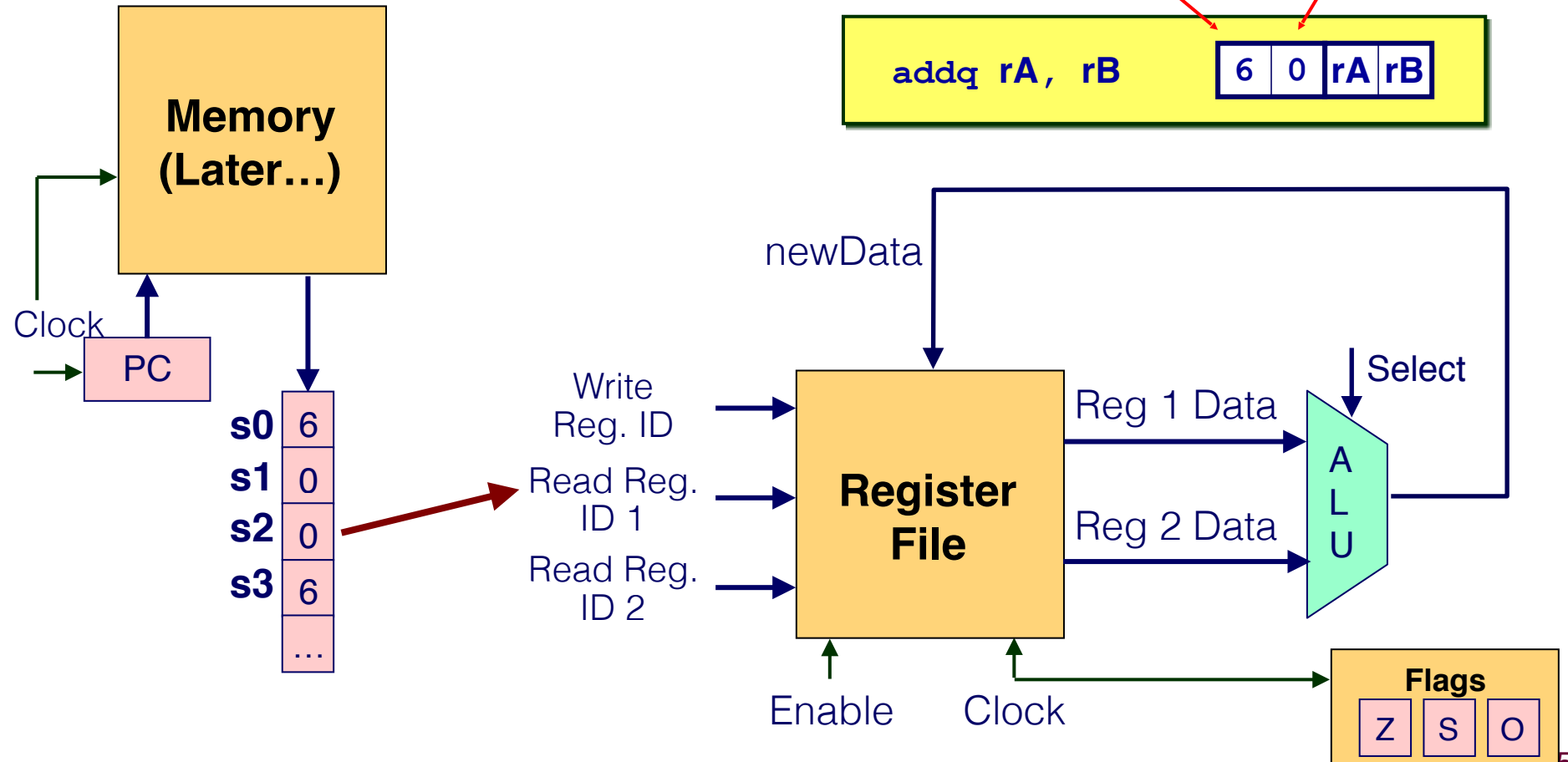
Instruction Code
Add



Executing an ADD instruction

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

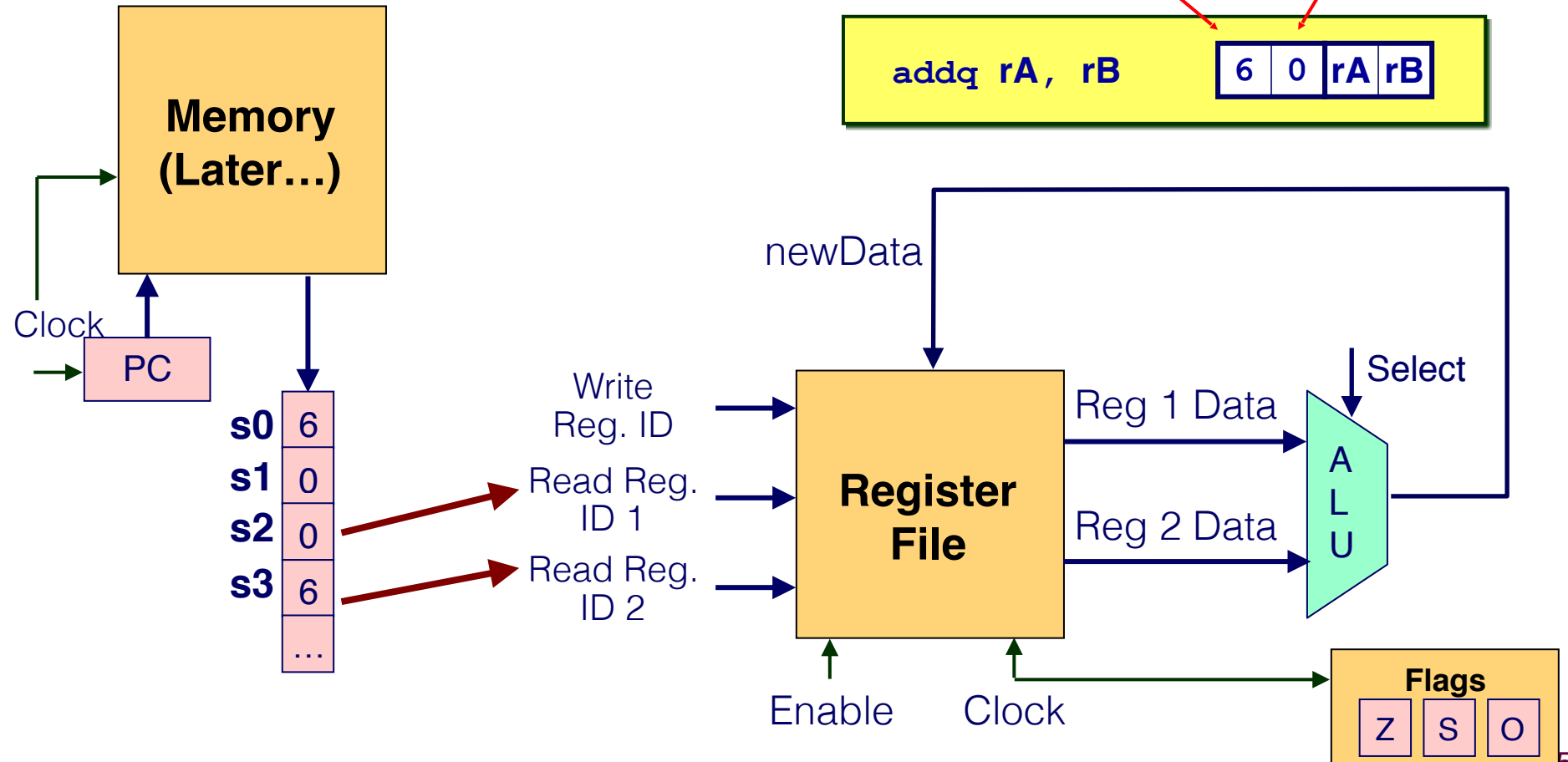
Instruction Code
Add



Executing an ADD instruction

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

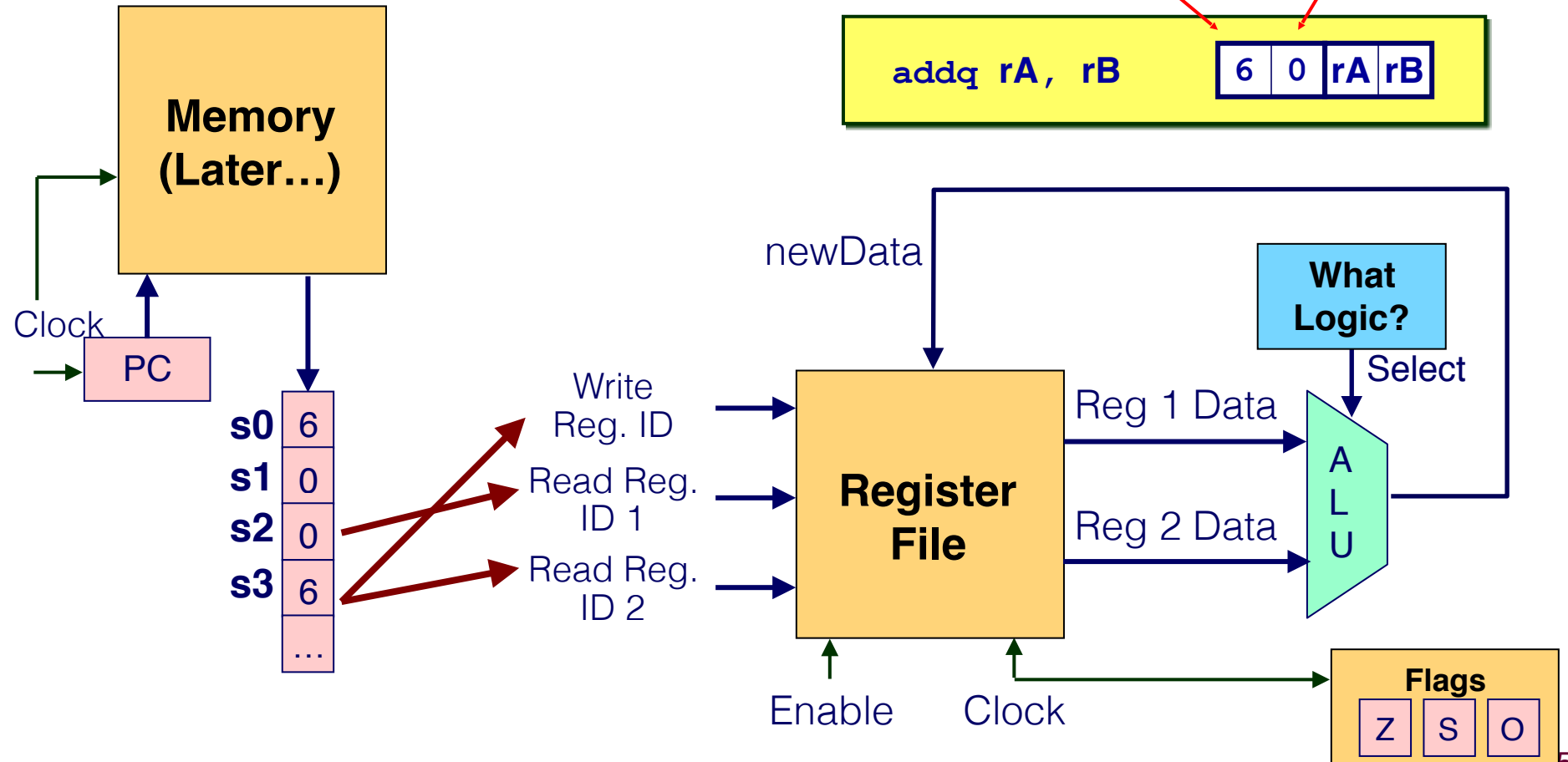
Instruction Code
Add



Executing an ADD instruction

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

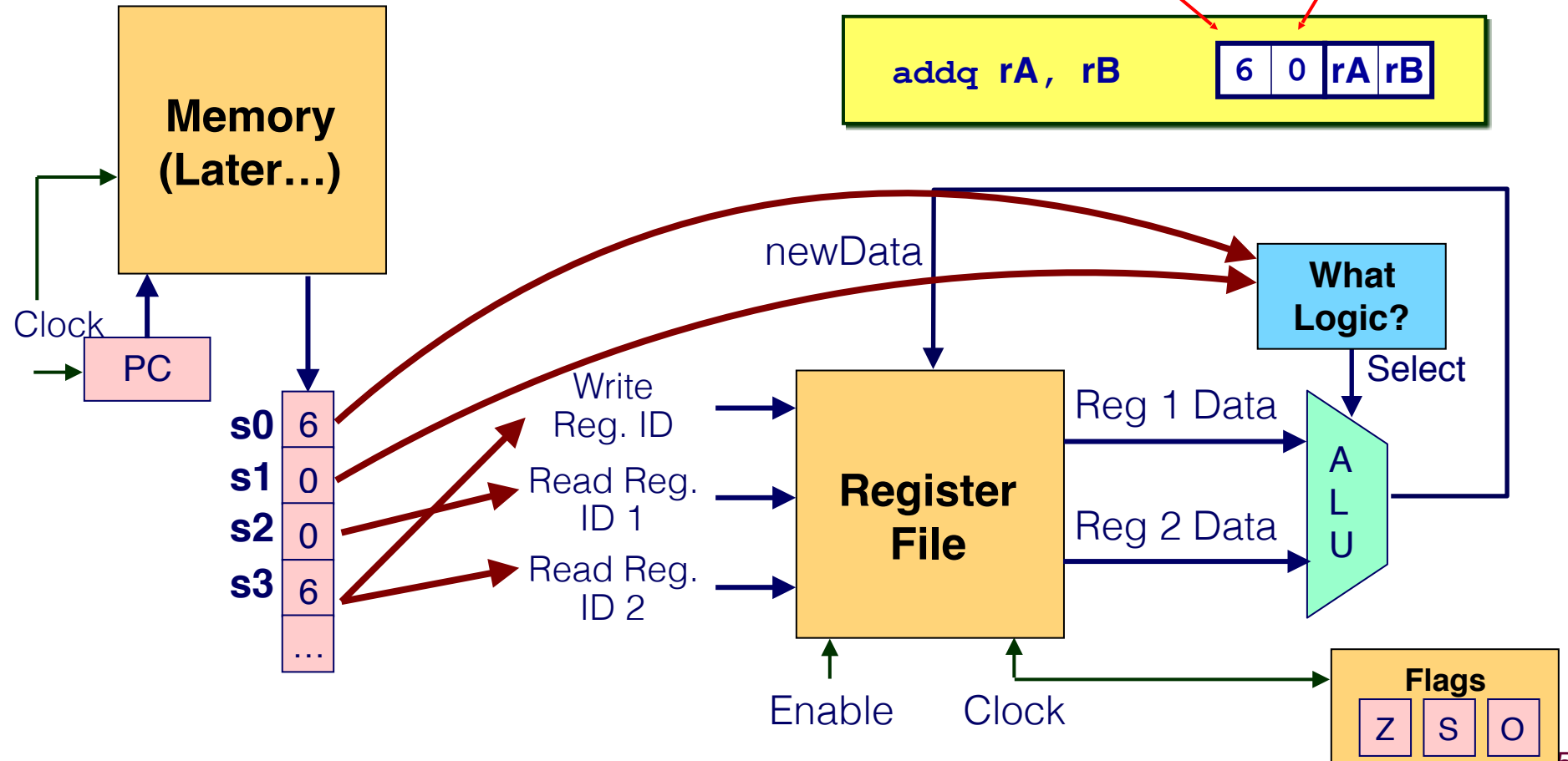
Instruction Code
Add



Executing an ADD instruction

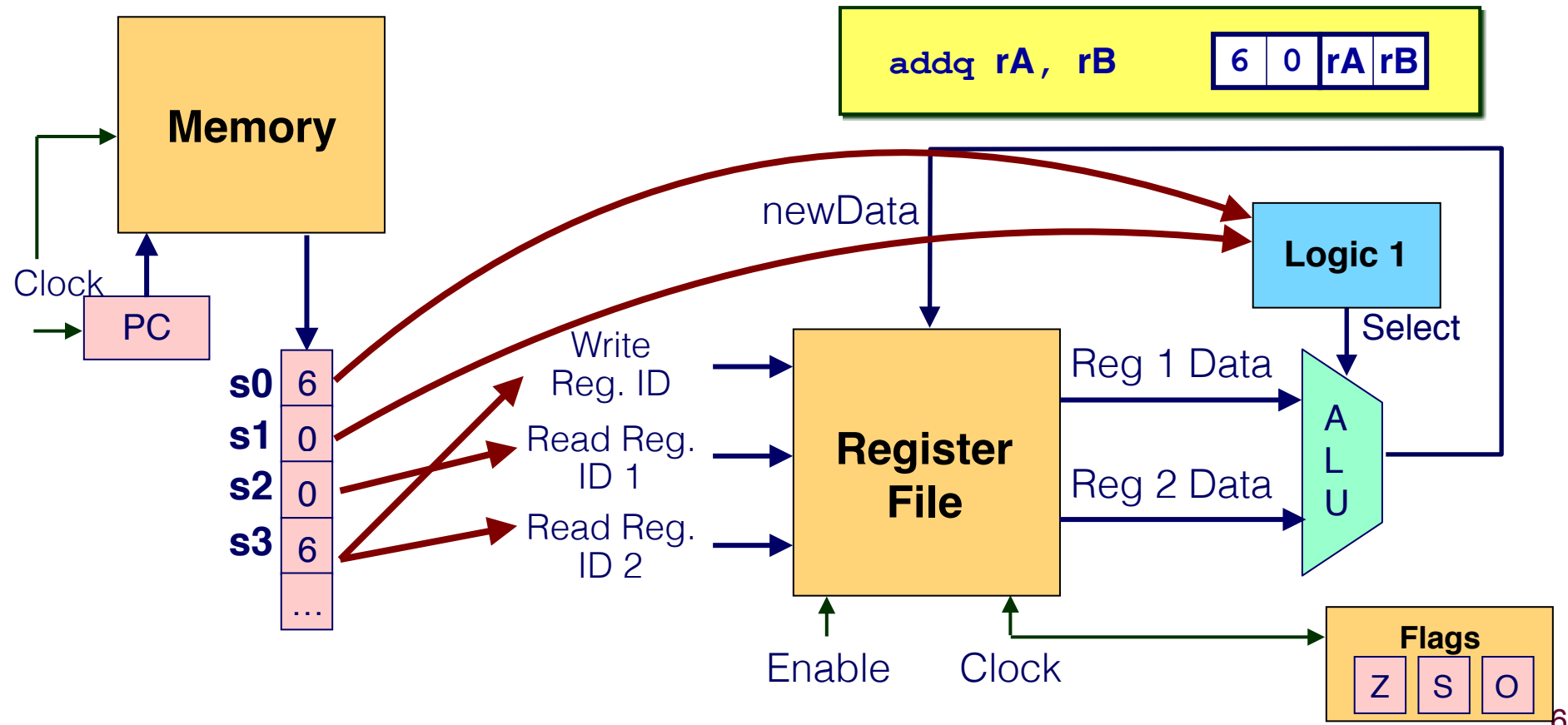
- How does the processor execute `addq %rax, %rsi`
- The binary encoding is `60 06`

Instruction Code
Add



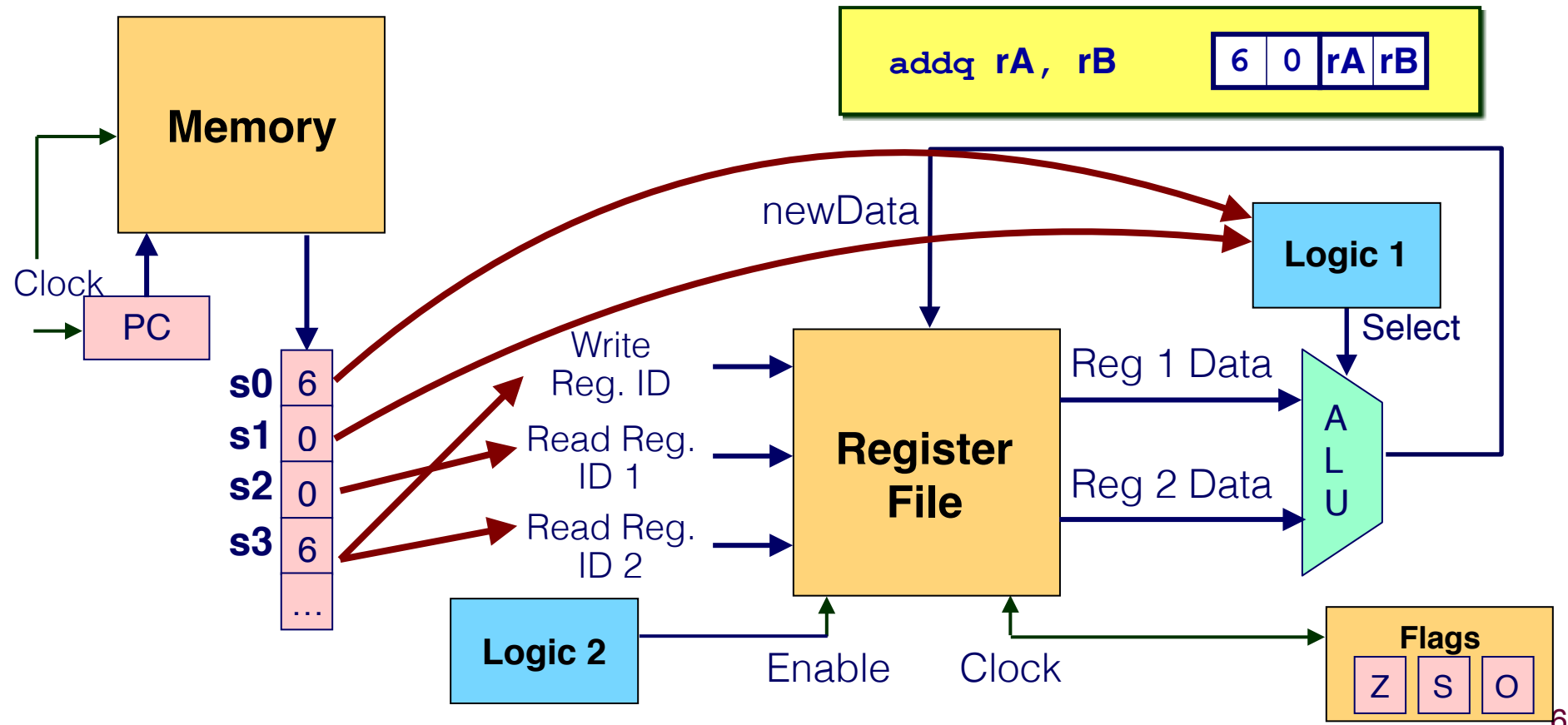
Executing an ADD instruction

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



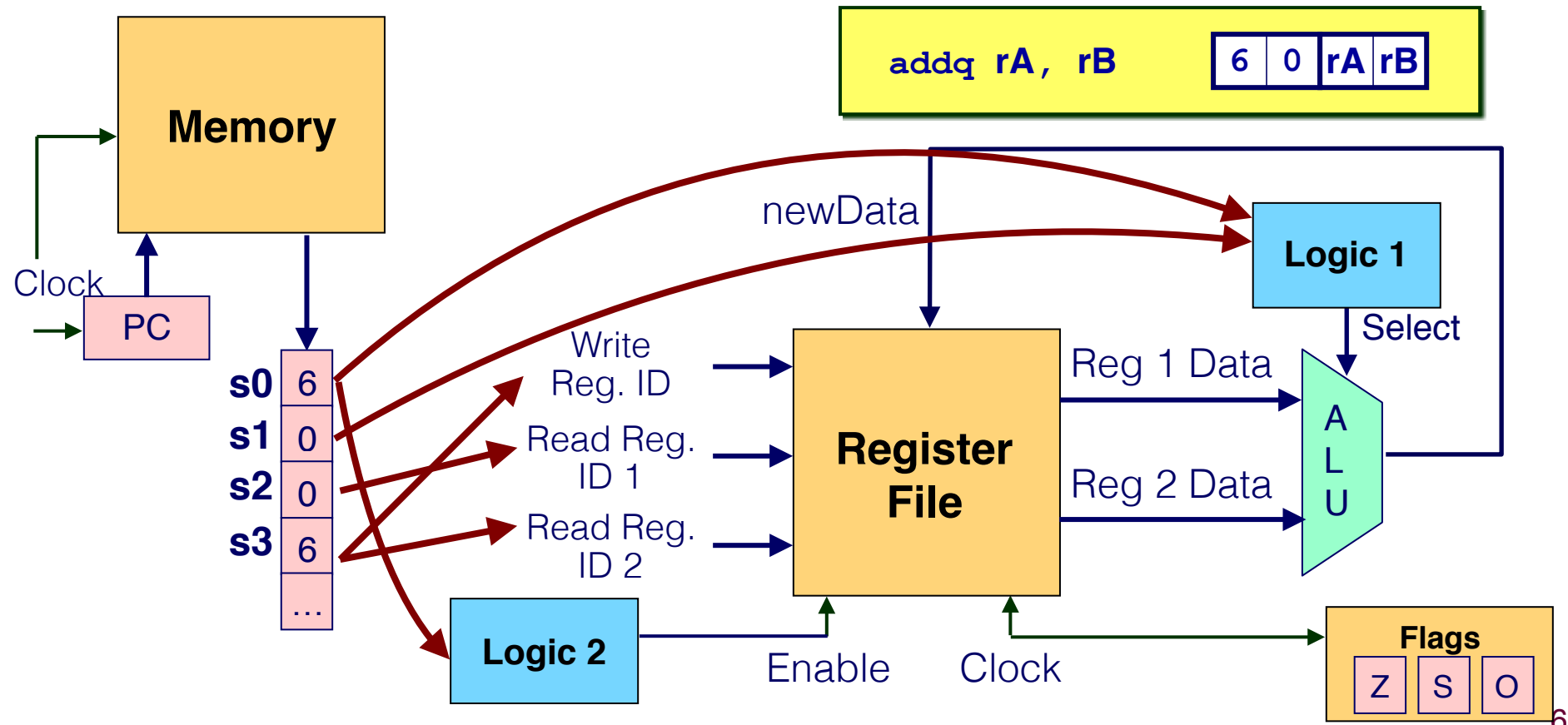
Executing an ADD instruction

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



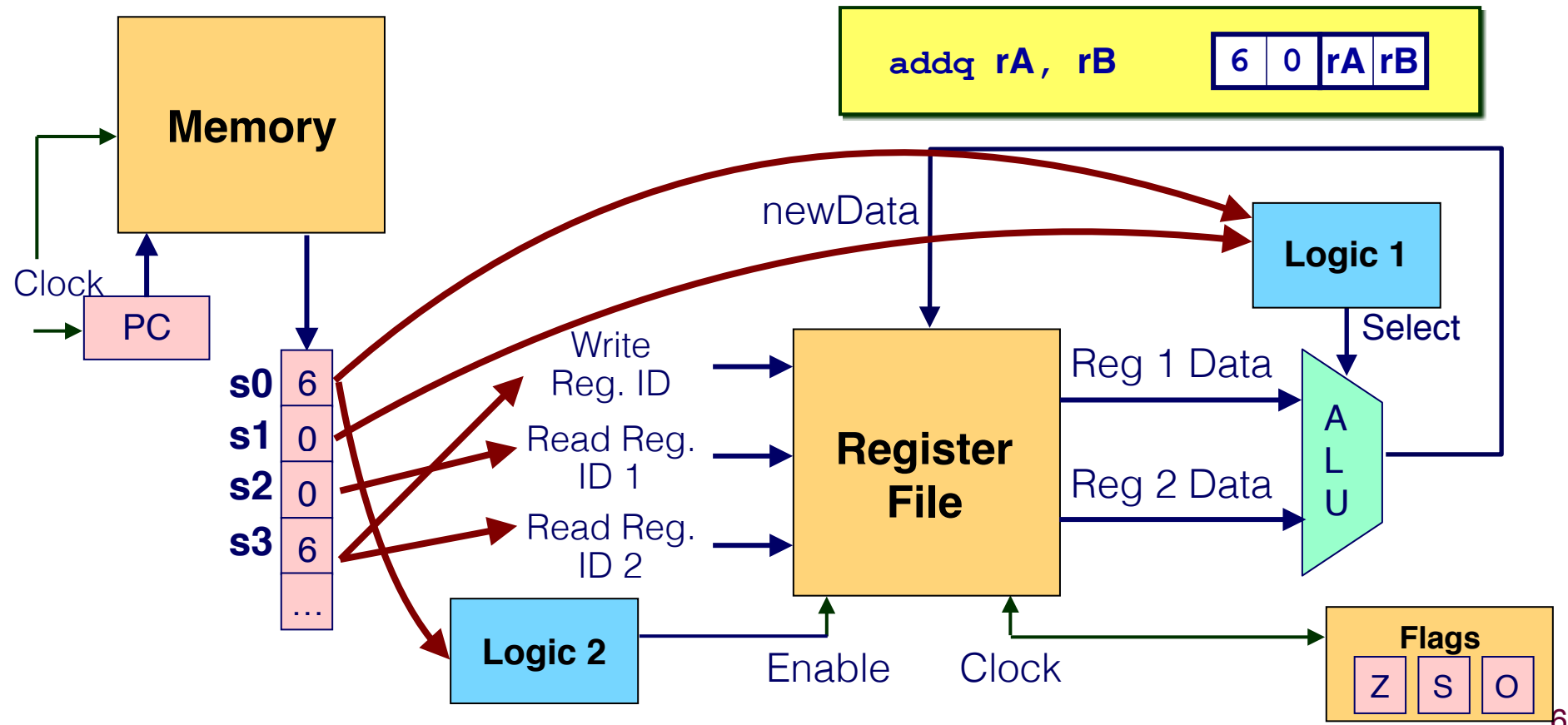
Executing an ADD instruction

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



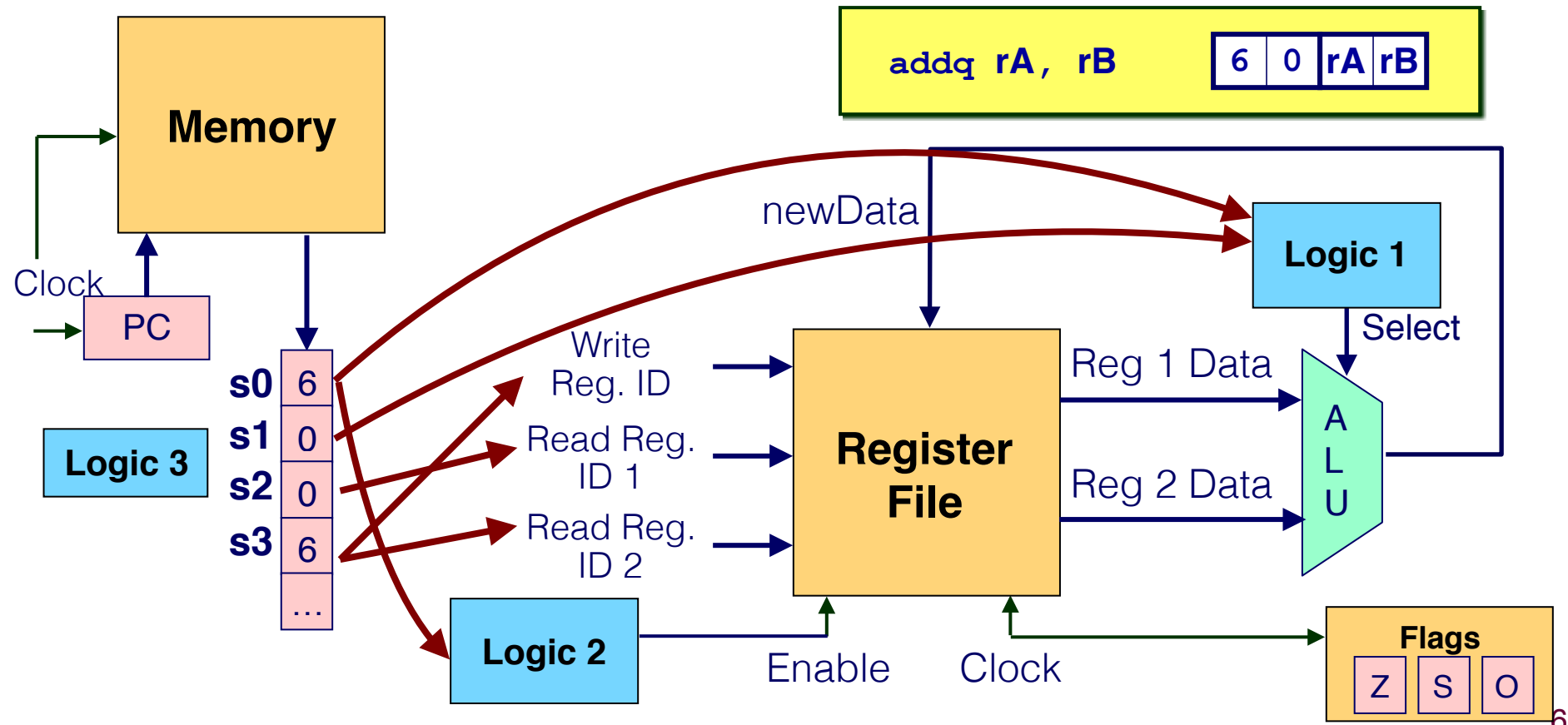
Executing an ADD instruction

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



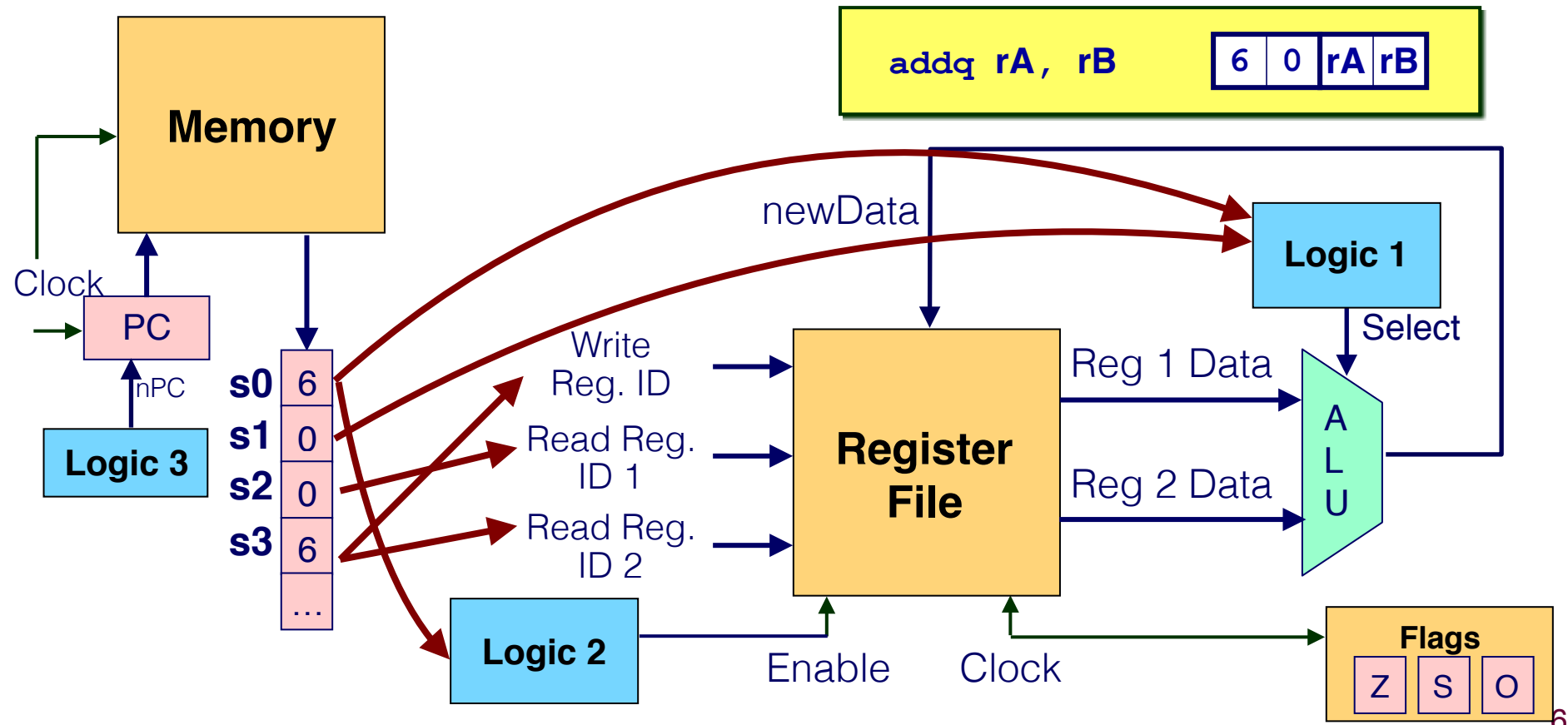
Executing an ADD instruction

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



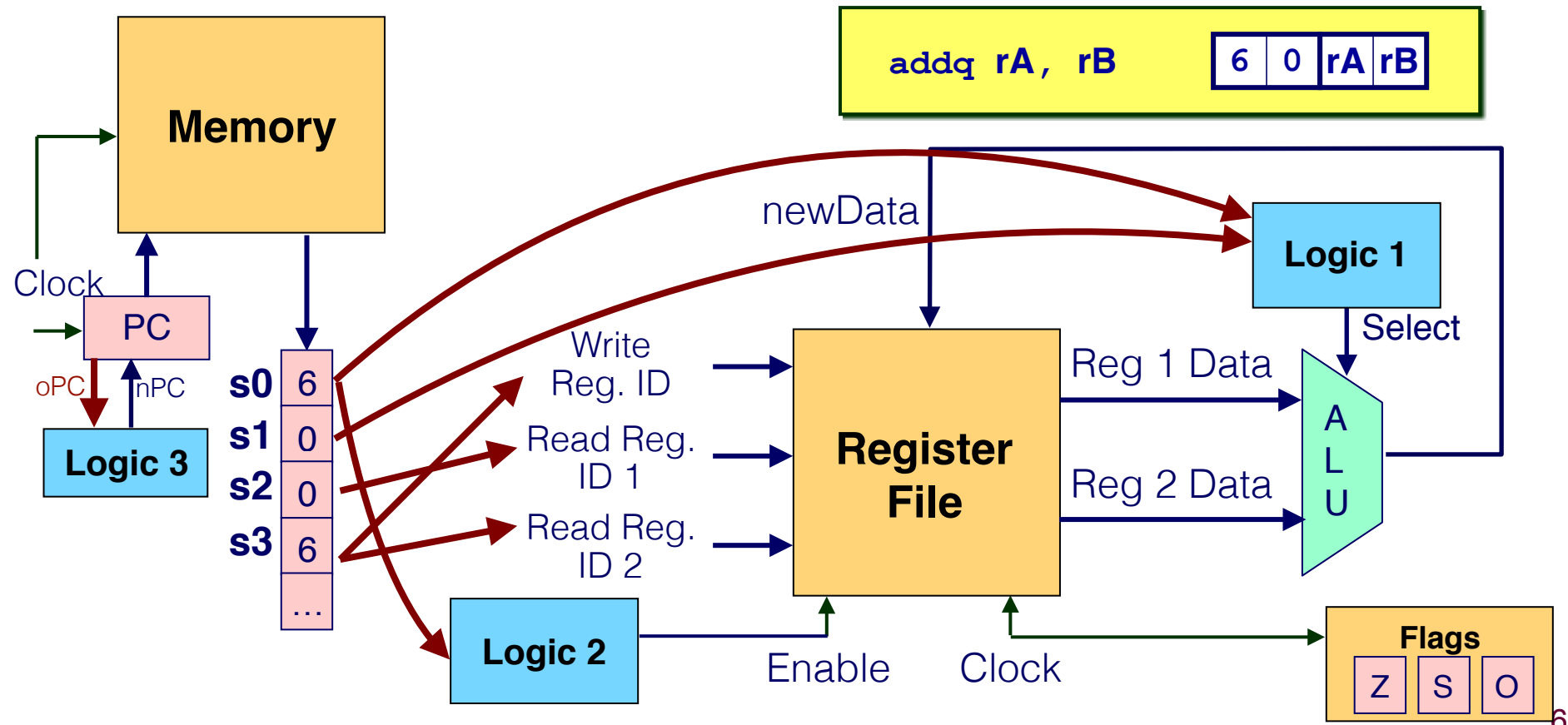
Executing an ADD instruction

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



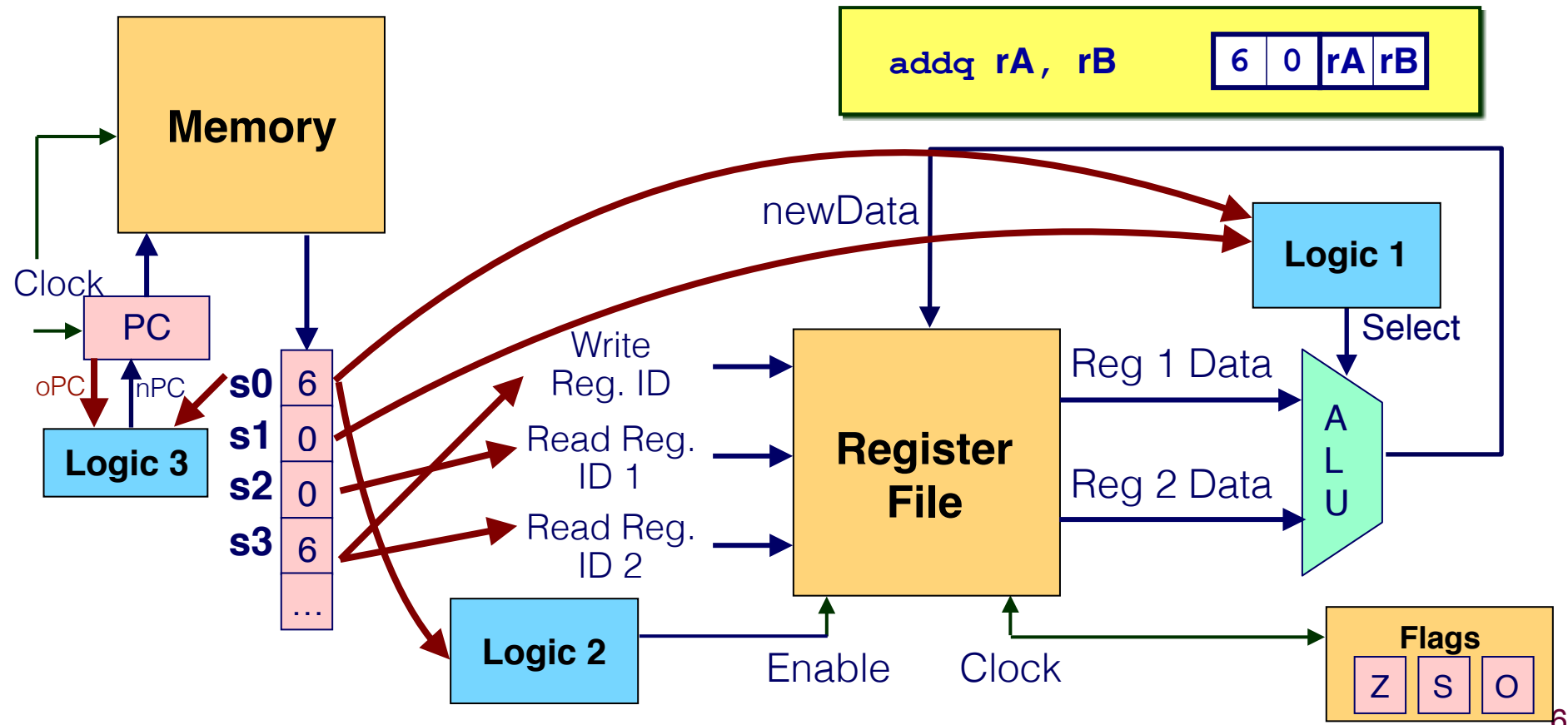
Executing an ADD instruction

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



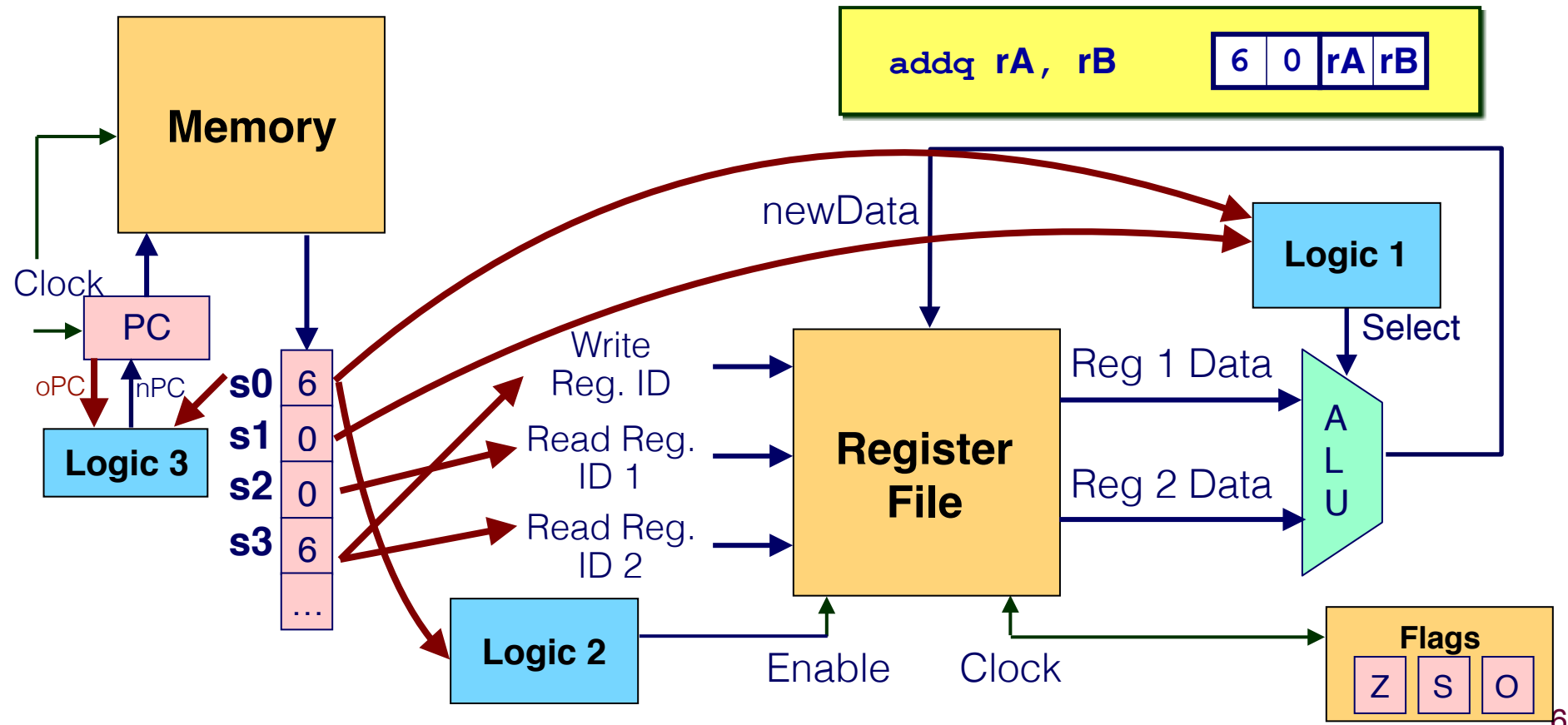
Executing an ADD instruction

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



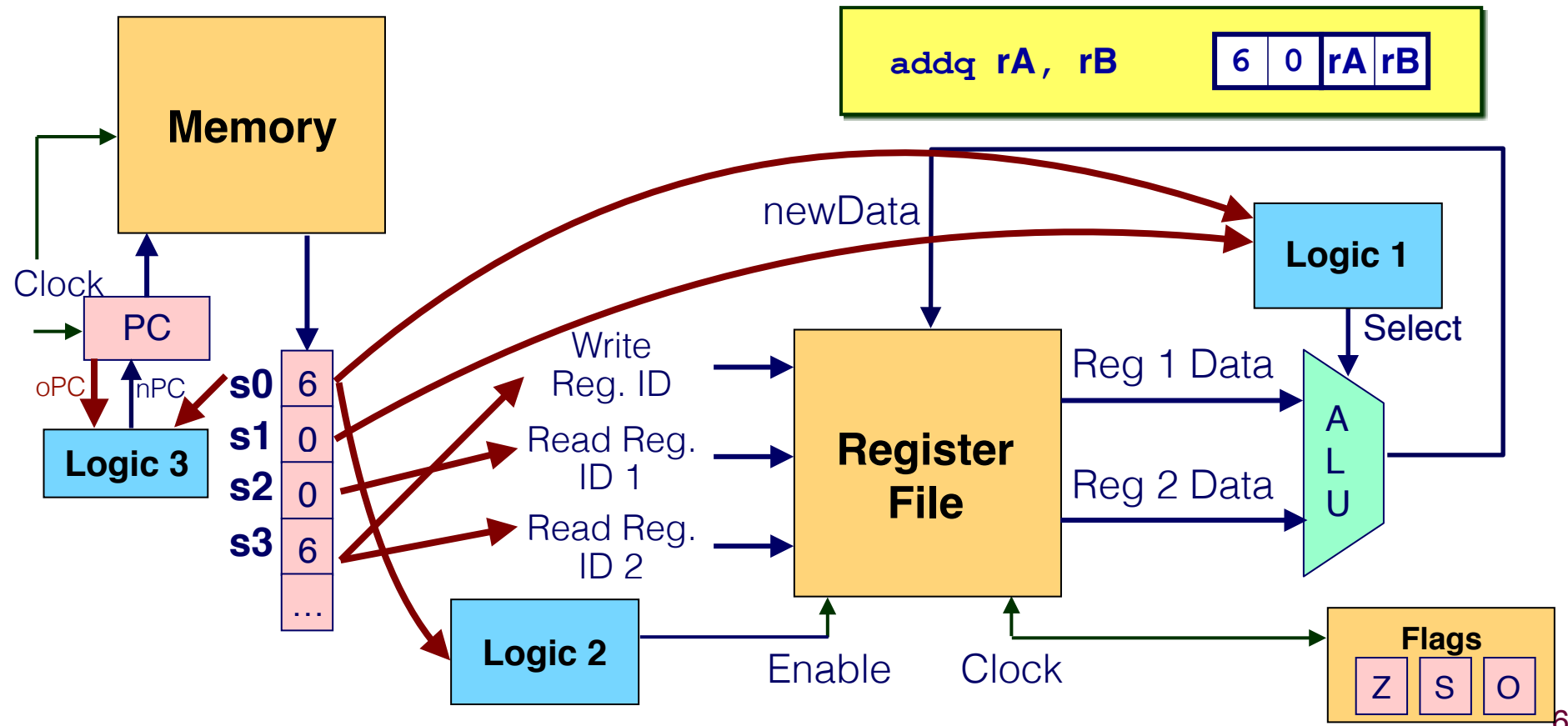
Executing an ADD instruction

- 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;



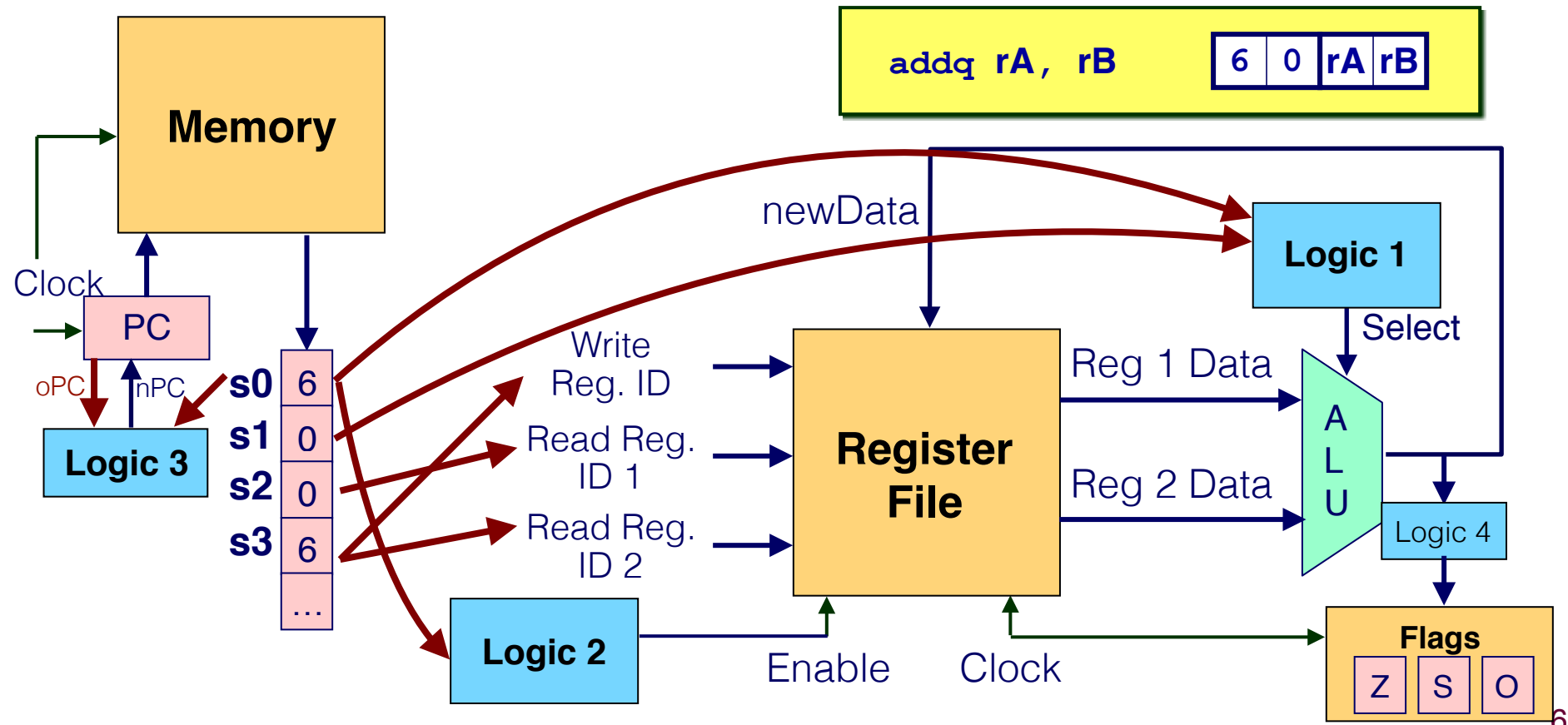
Executing an ADD instruction

- 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

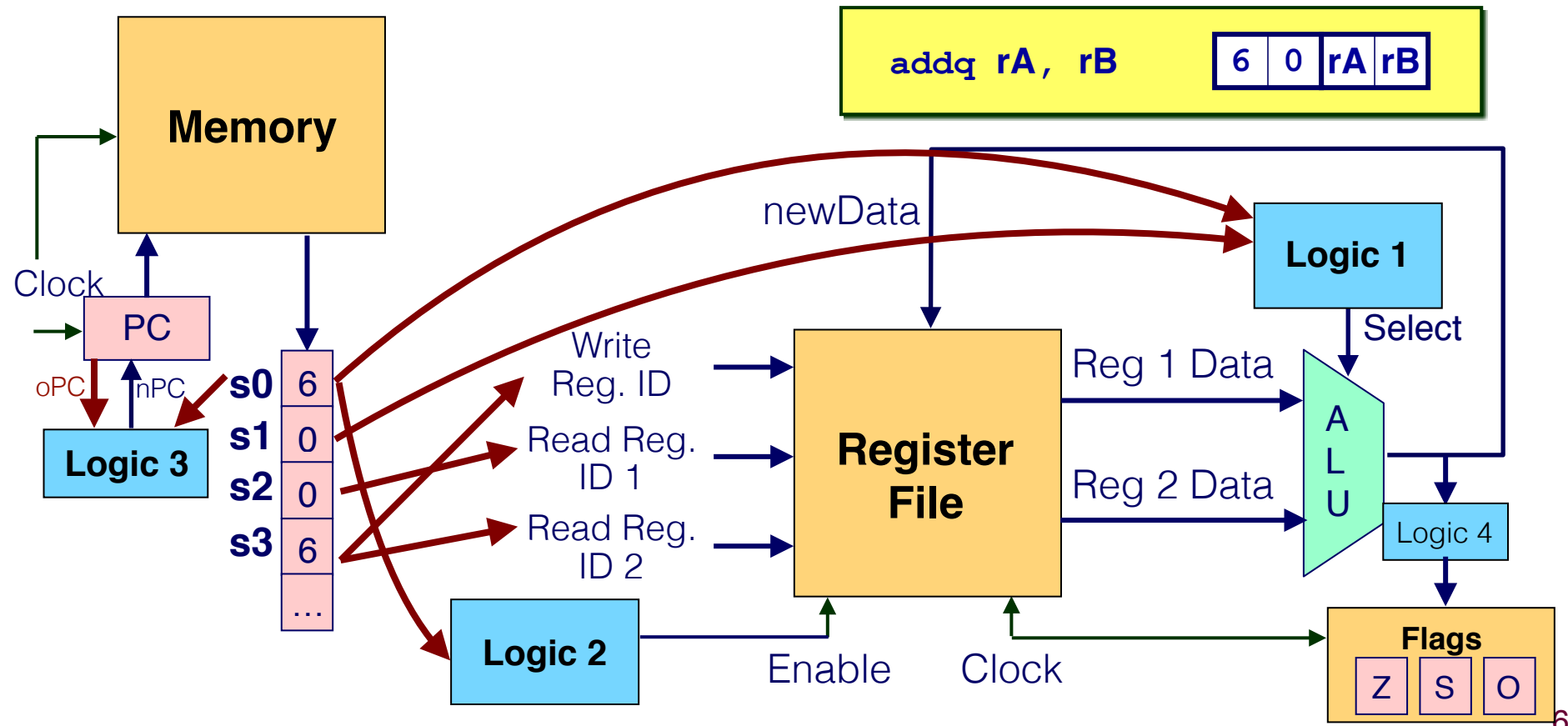
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?



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?

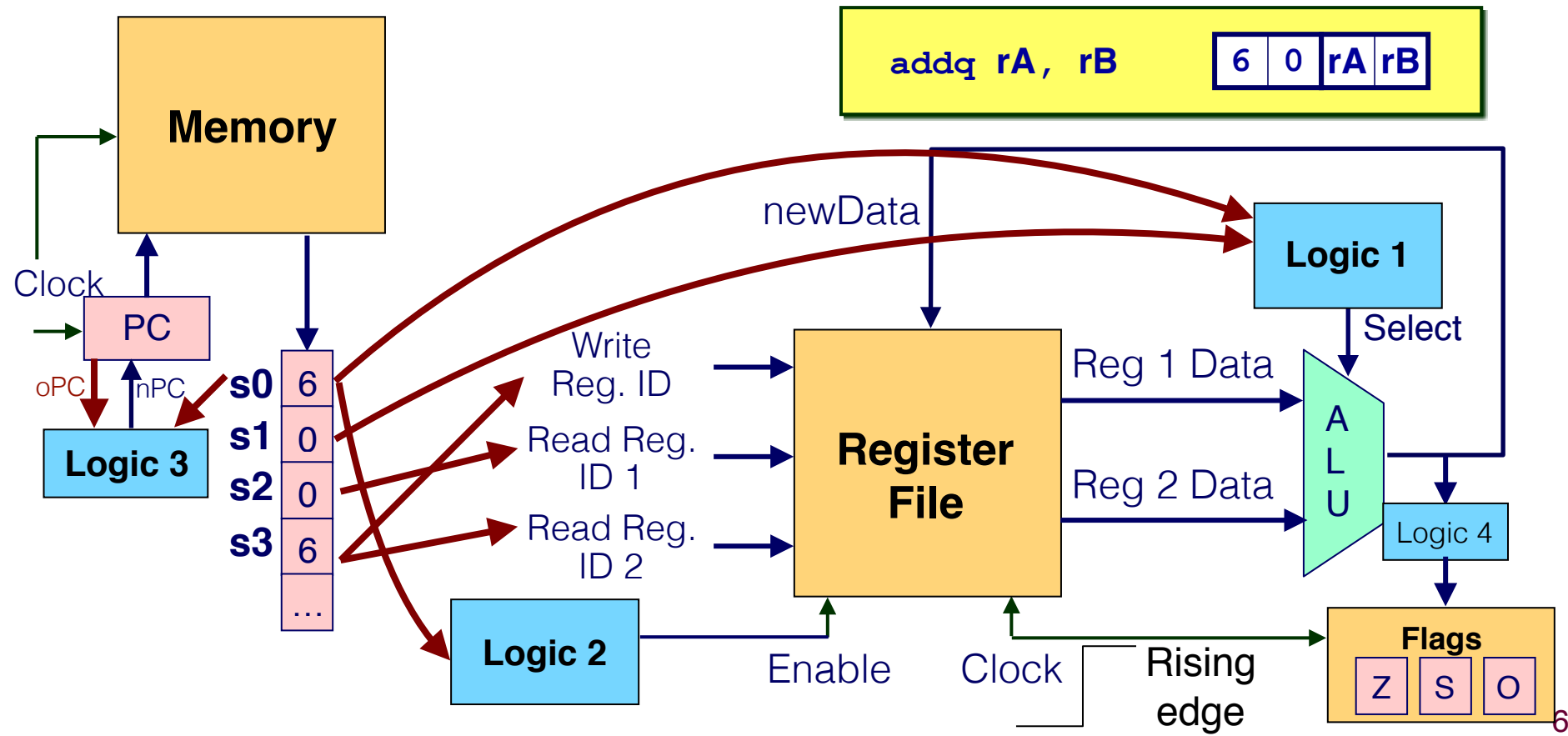
How do these logics get implemented?



Executing an ADD instruction

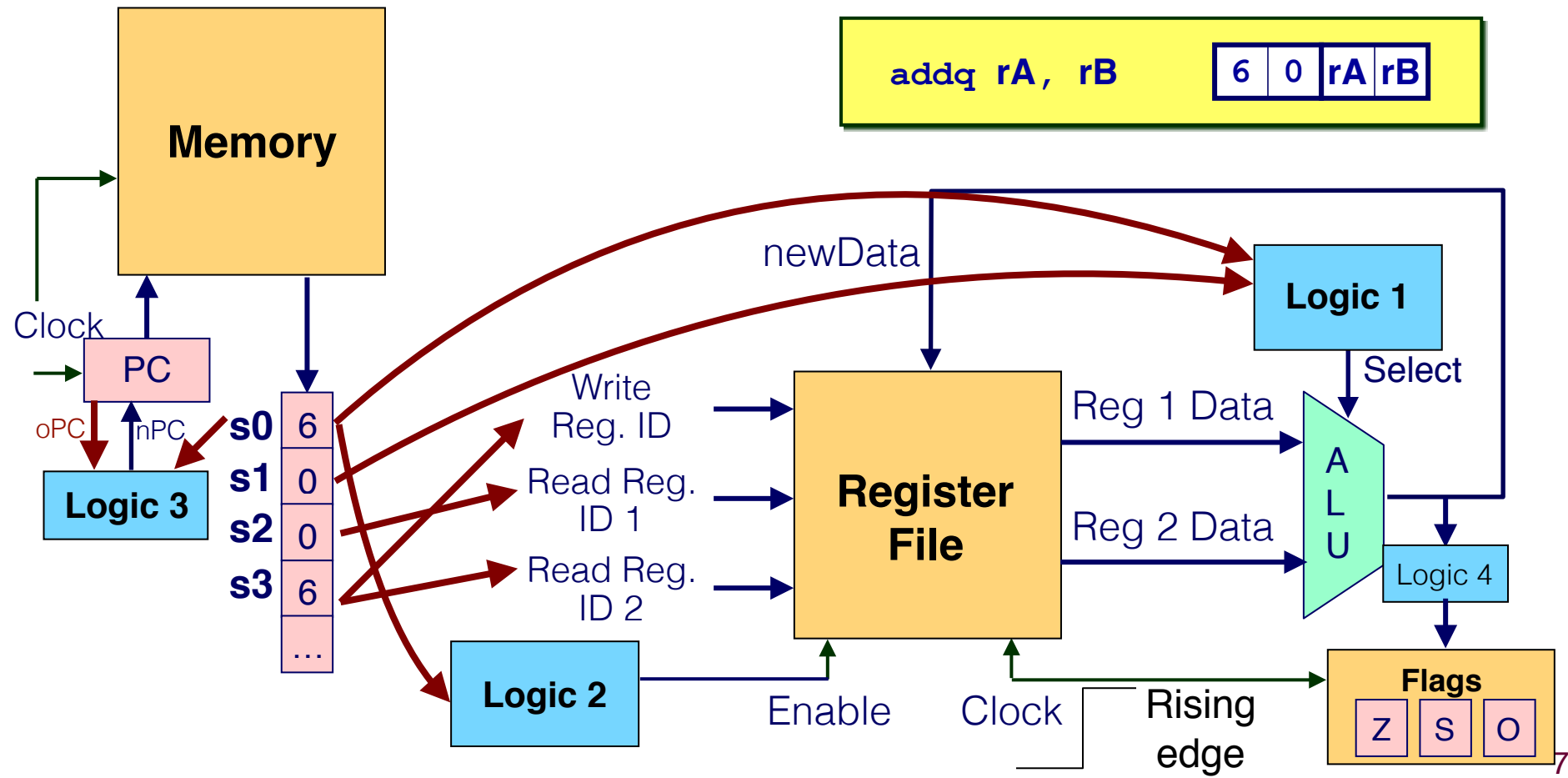
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

How do these logics get implemented?



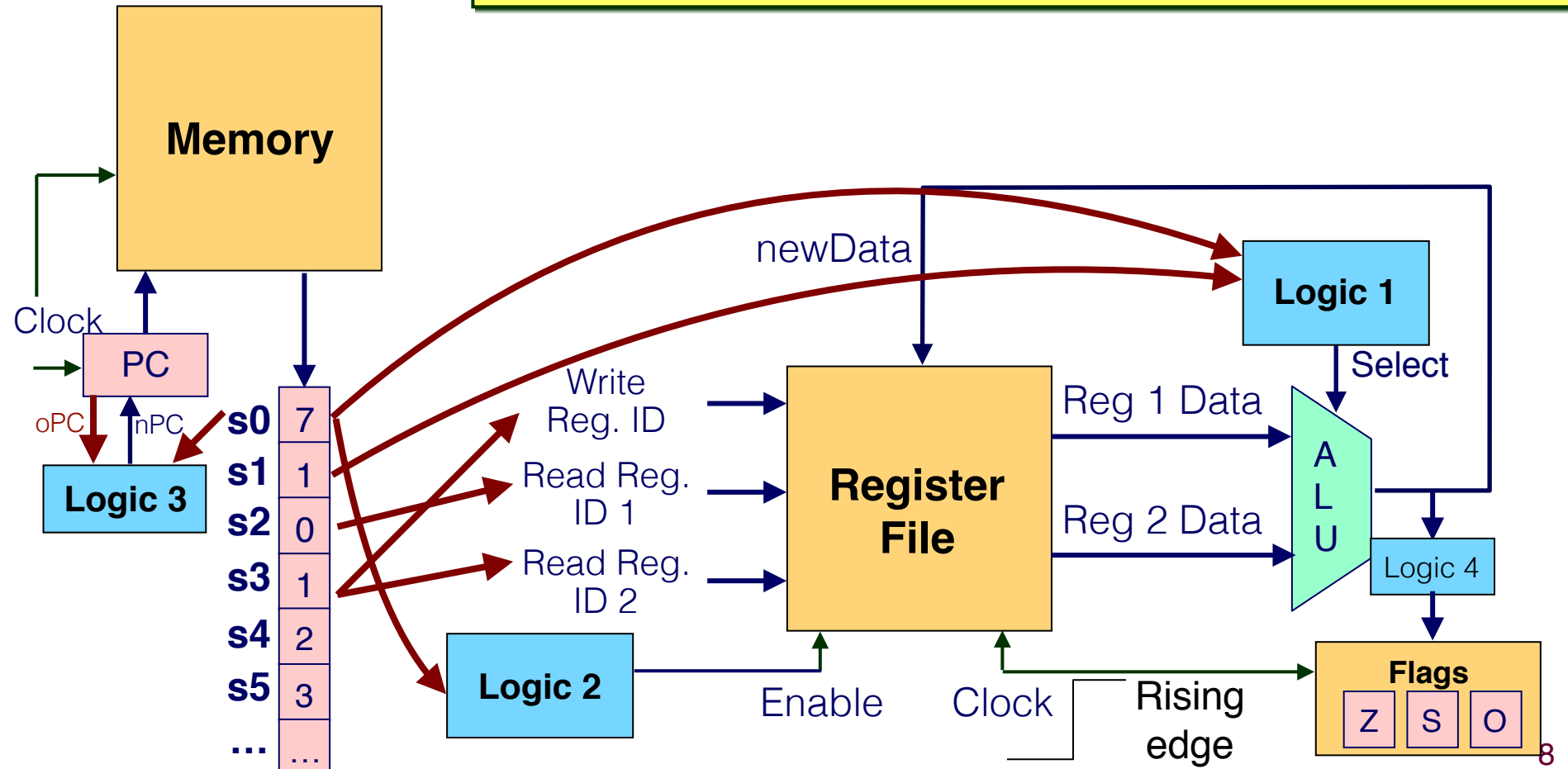
Executing an ADD instruction

- 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.

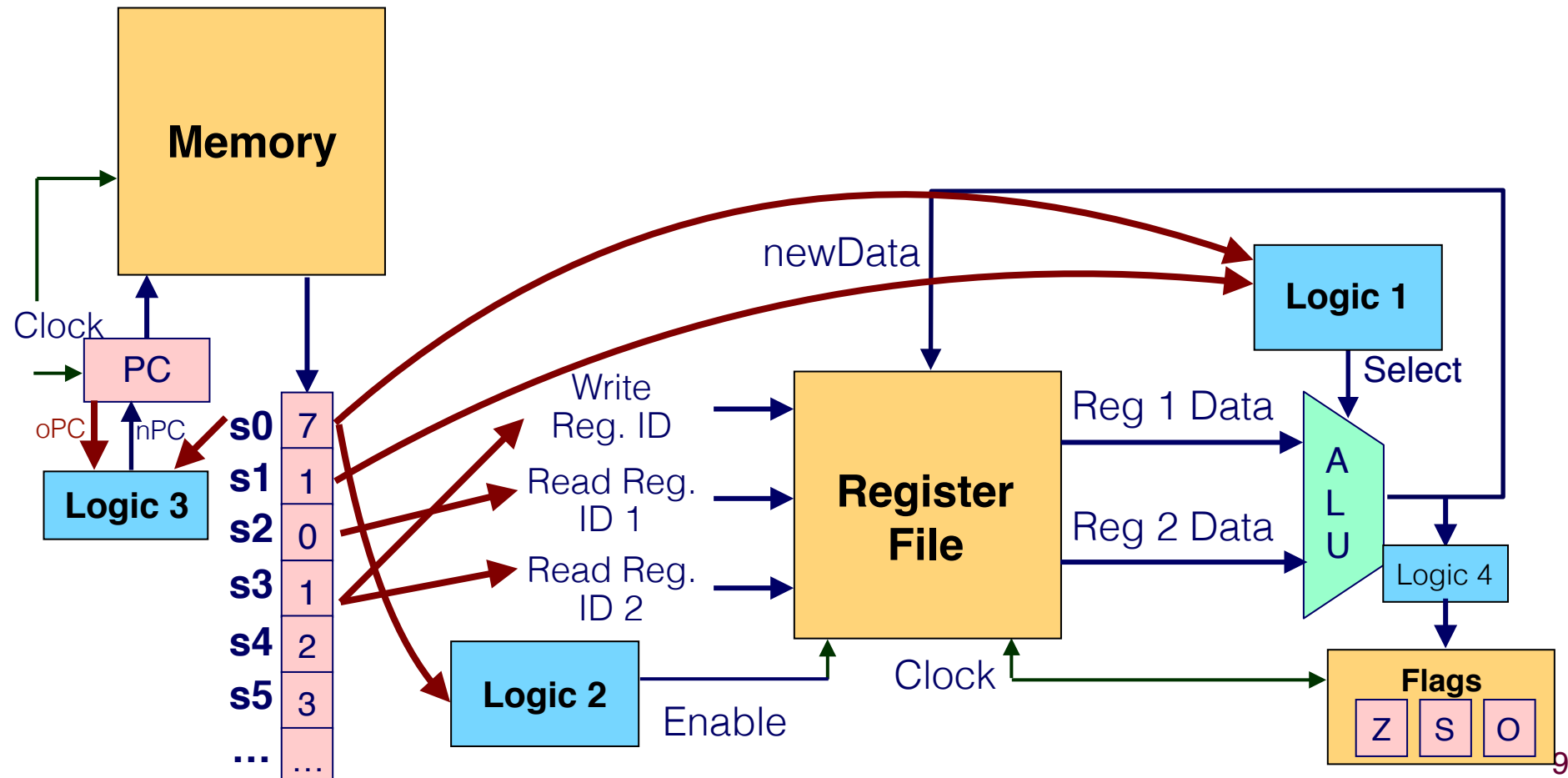


Executing a JLE instruction

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

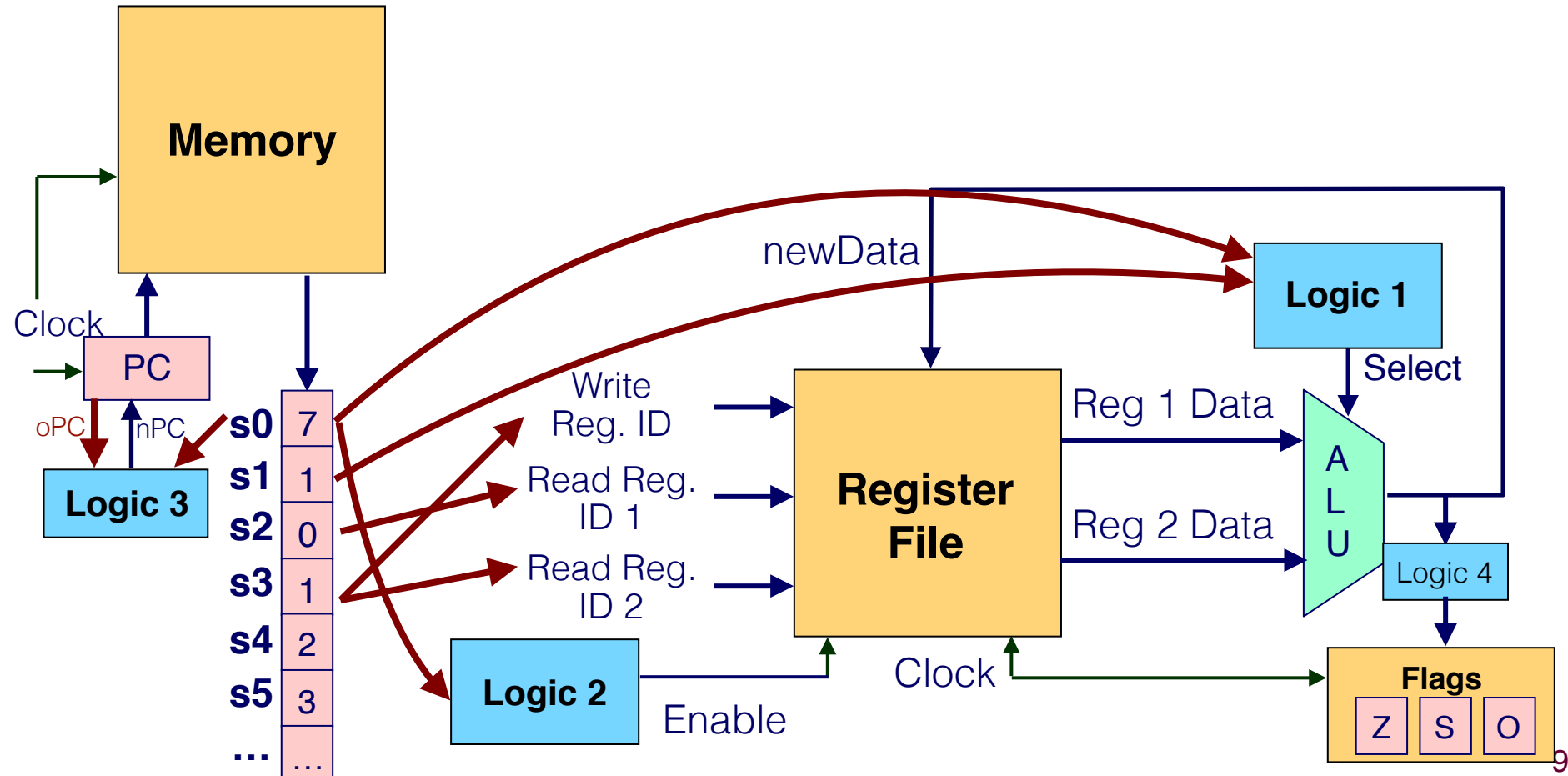


Executing a JLE instruction



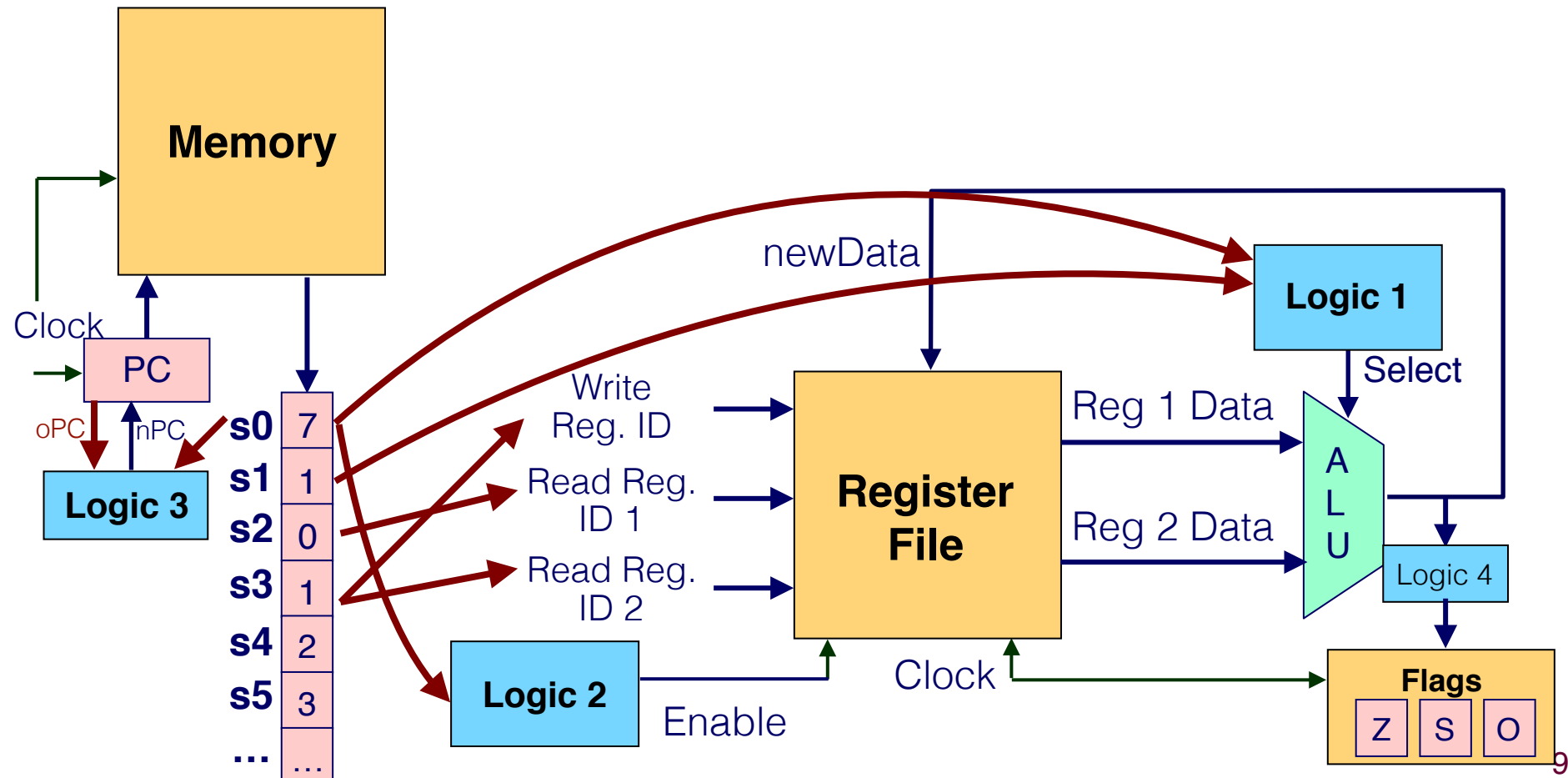
Executing a JLE instruction

- 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;



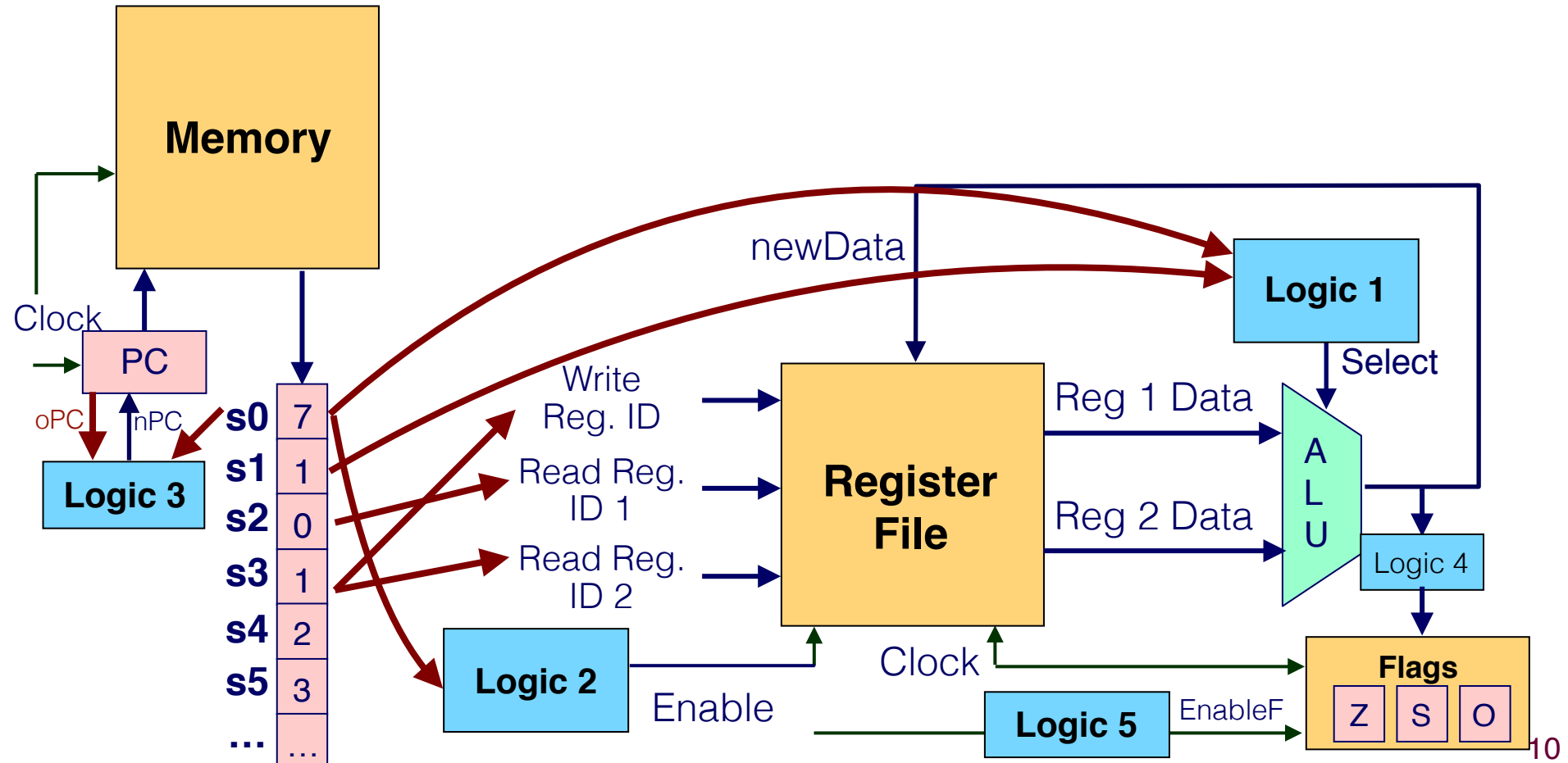
jle Dest

7 1

Dest

Executing a JLE instruction

- Logic 3??



jle Dest

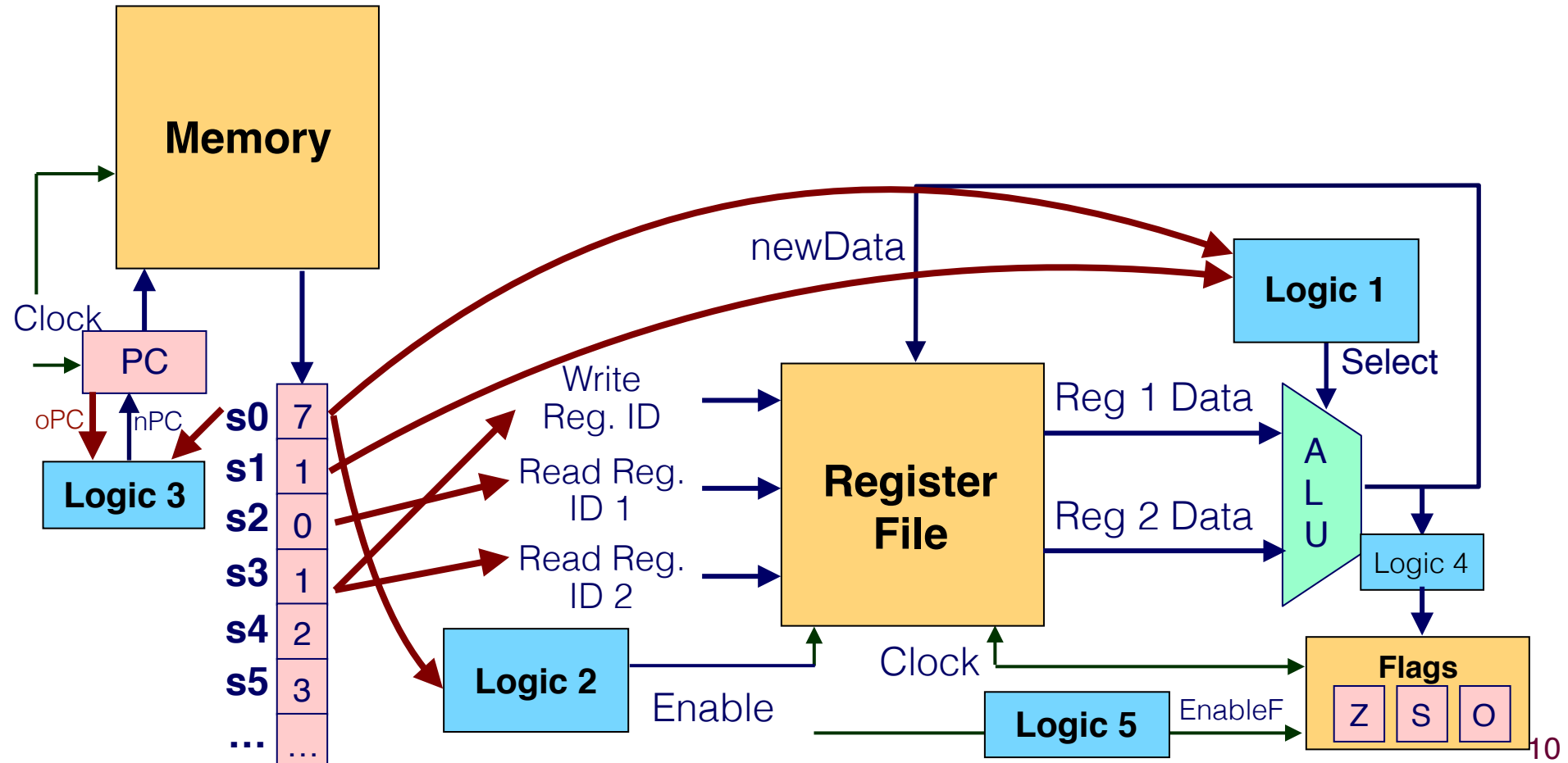
7 1

Dest

Executing a JLE instruction

- Logic 3???

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



jle Dest

7 1

Dest

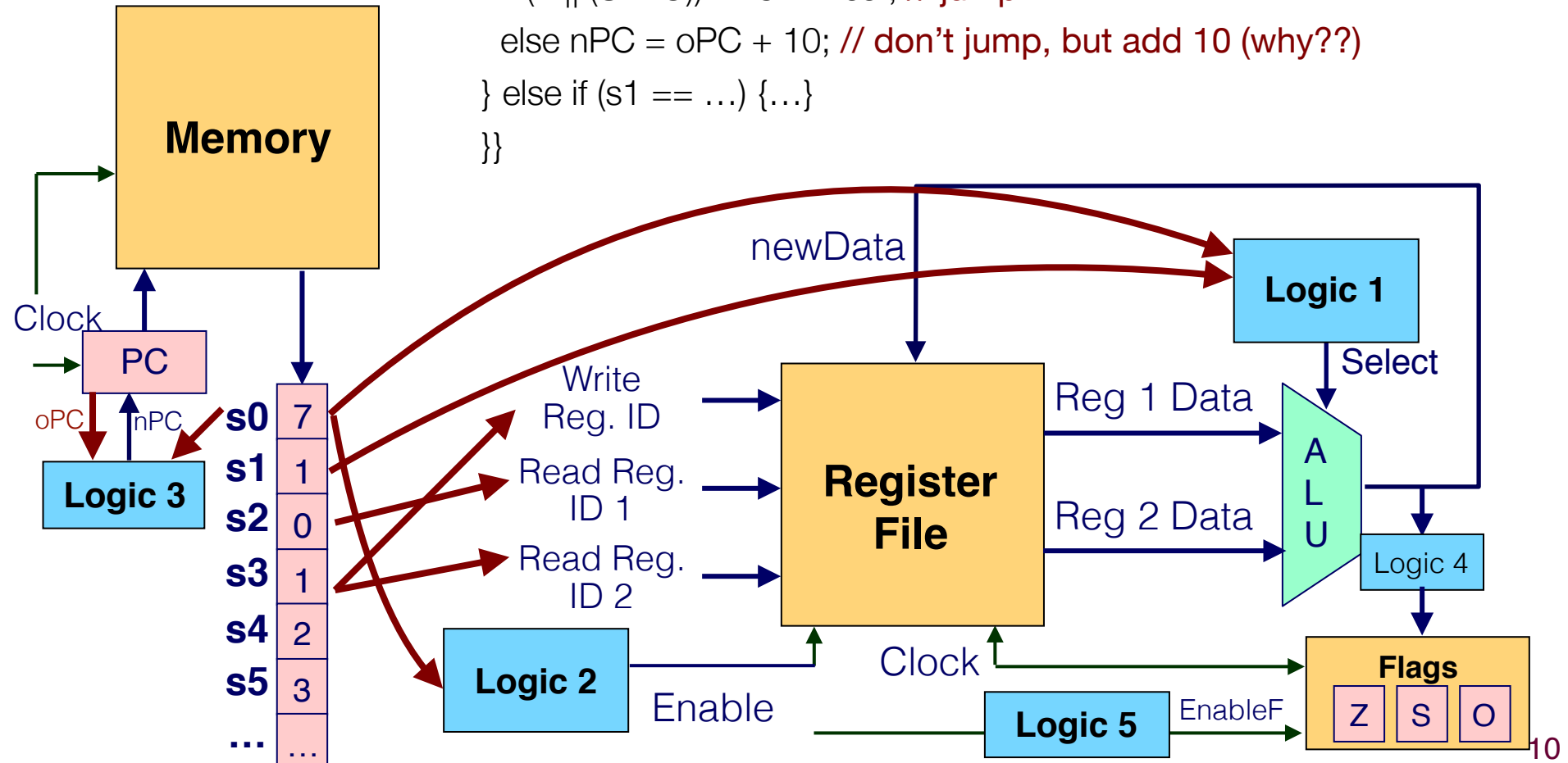
Executing a JLE instruction

- 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

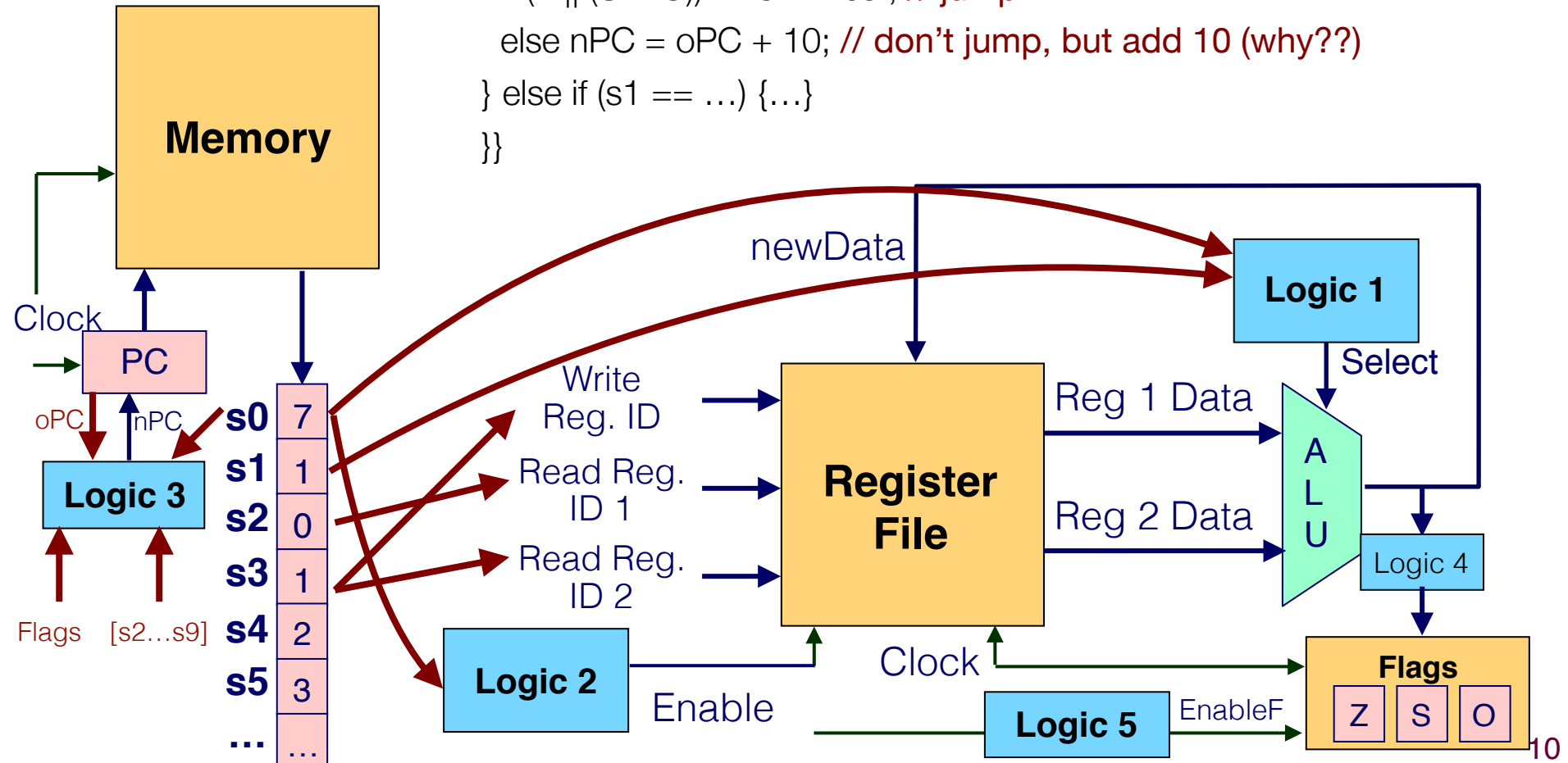
Executing a JLE instruction

- Logic 3???

```

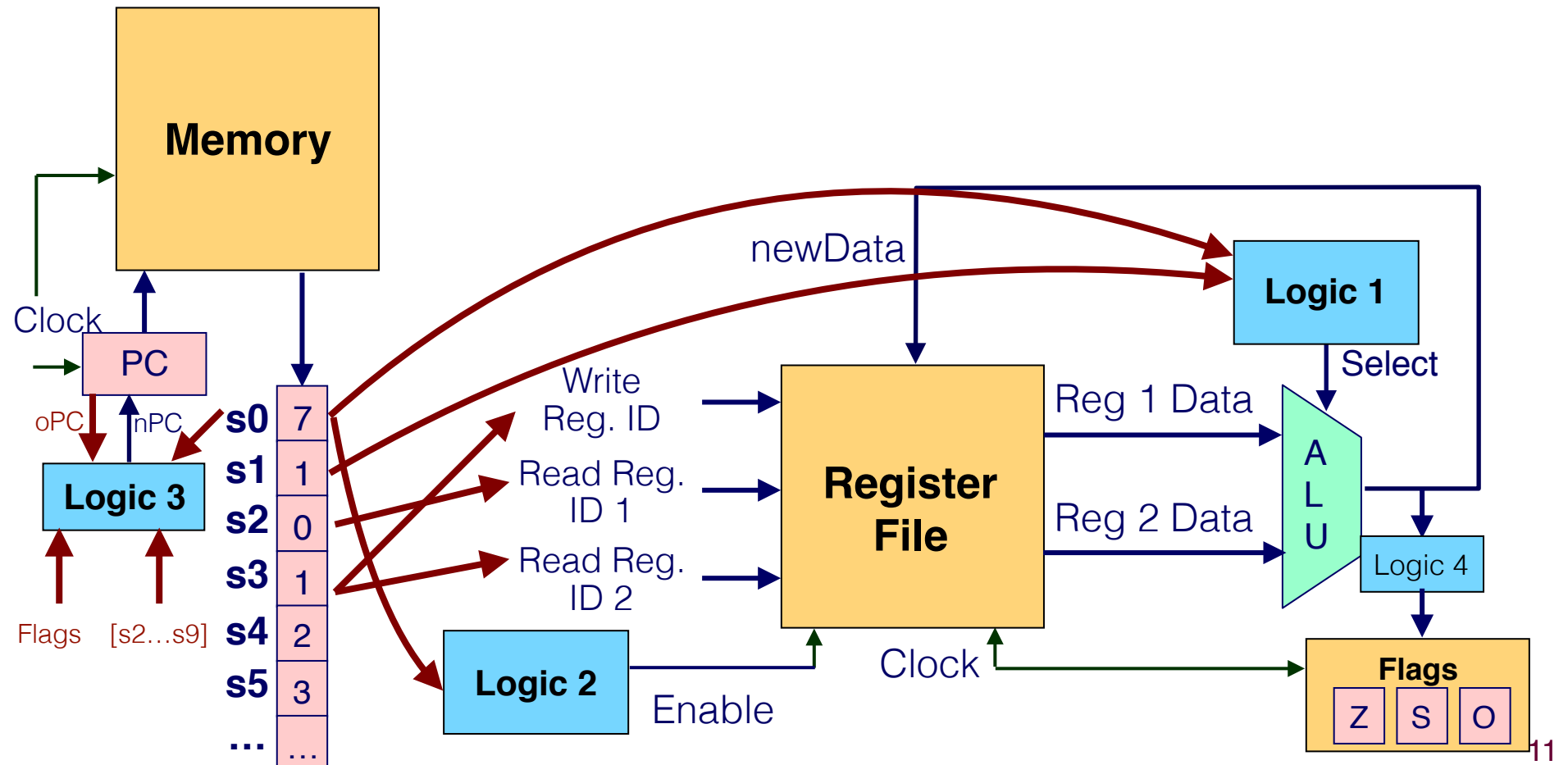
if (s0 == 6) nPC = oPC + 2;
else if (s0 == 7) {
  if (s1 == 1) { // jLE
    if (Z || (S ^ O)) nPC = Dest; // jump
    else nPC = oPC + 10; // don't jump, but add 10 (why??)
  } else if (s1 == ...) {...}
}

```



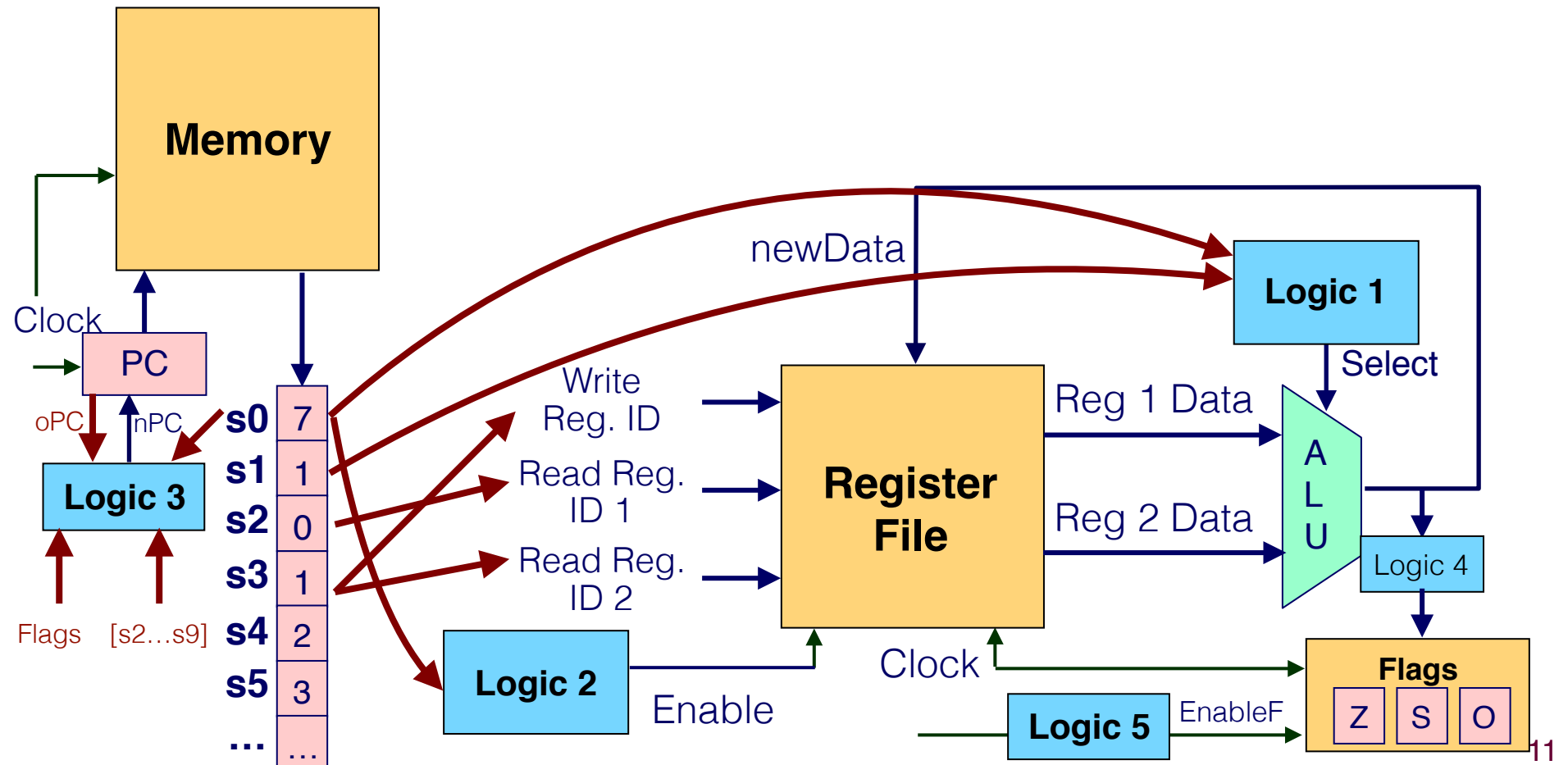
Executing a JLE instruction

- Logic 4? Does JLE write flags?



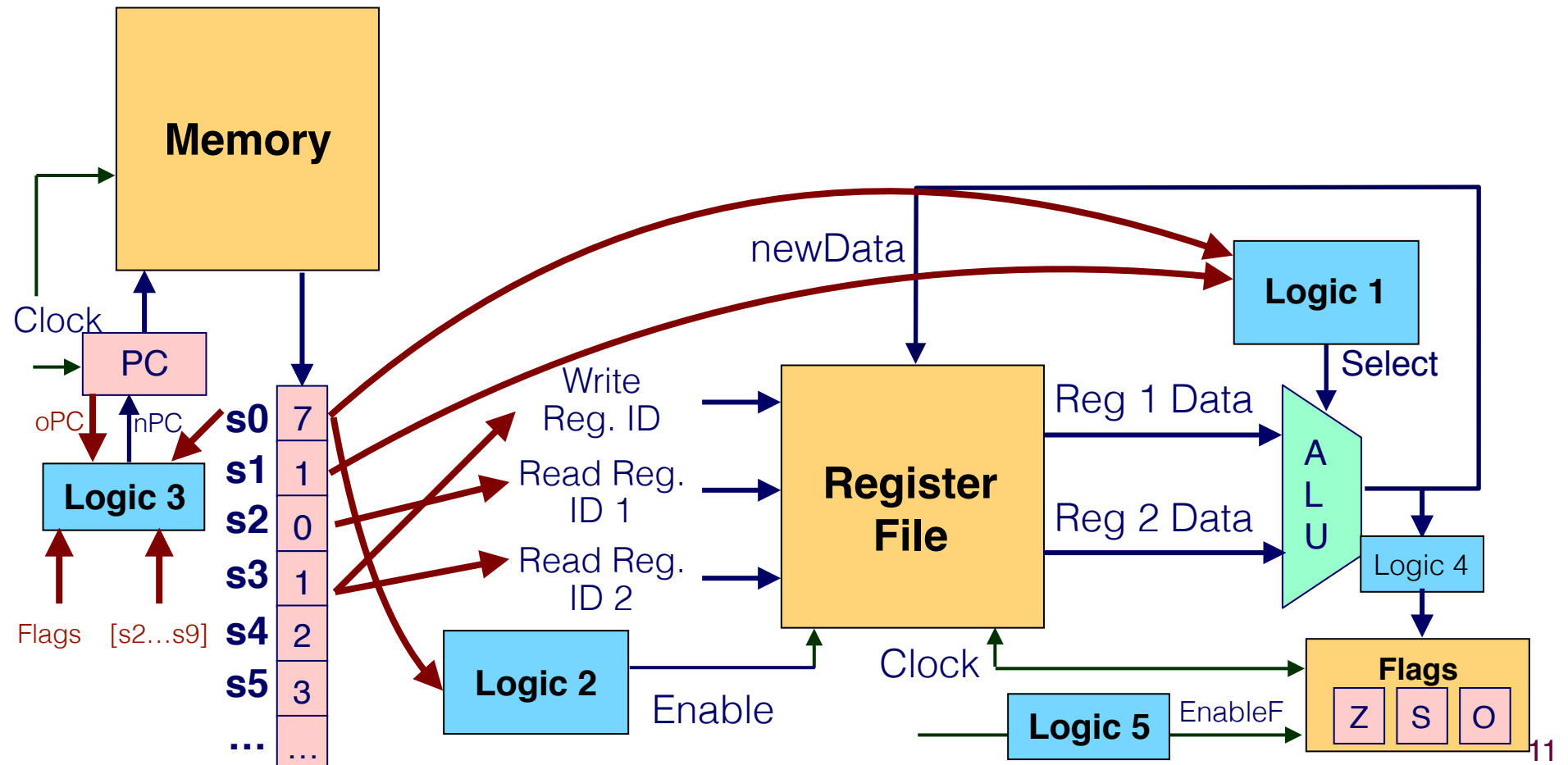
Executing a JLE instruction

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

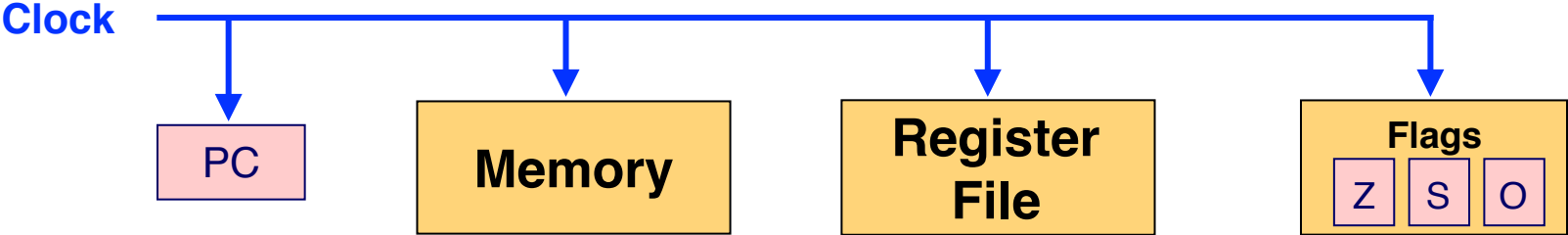


Executing a JLE instruction

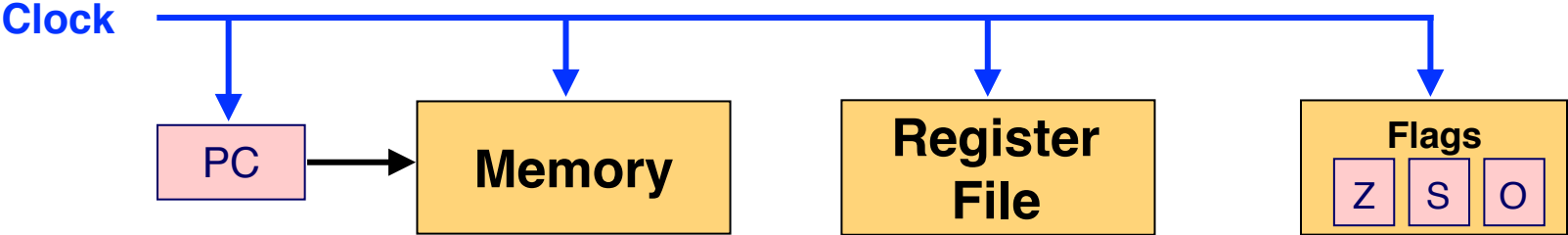
- Logic 4? Does JLE write flags?
- Need another piece of logic.
- Logic 5: if (s0 == 7) EnableF = 0; else if (s0 == 6) EnableF = 1;



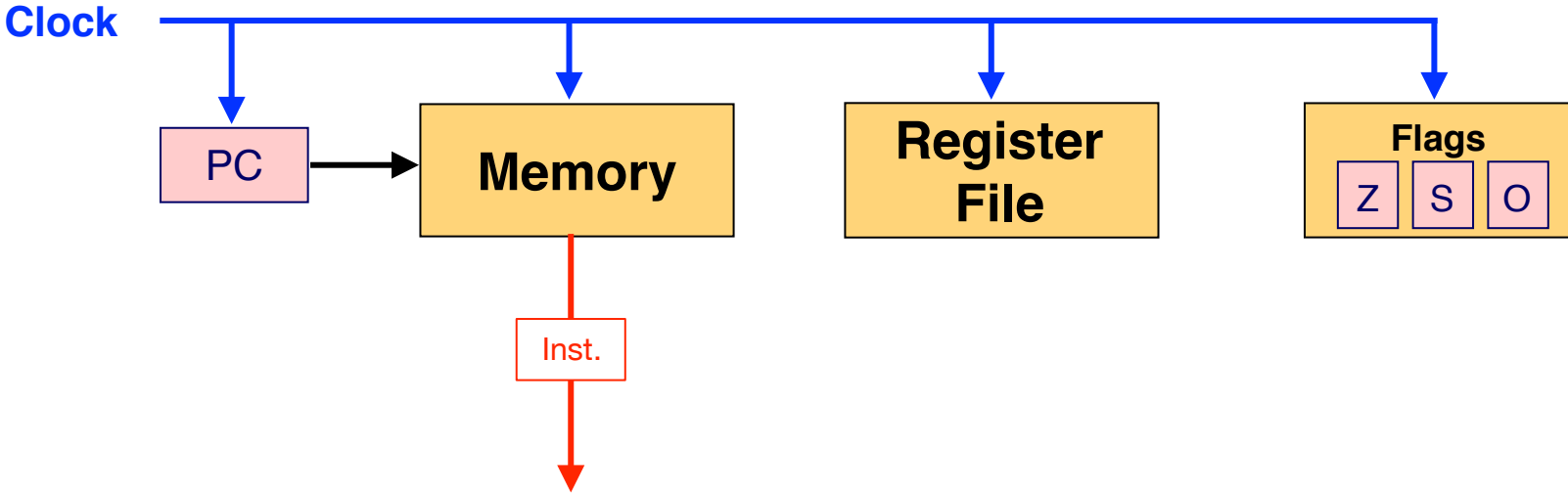
Microarchitecture (So far)



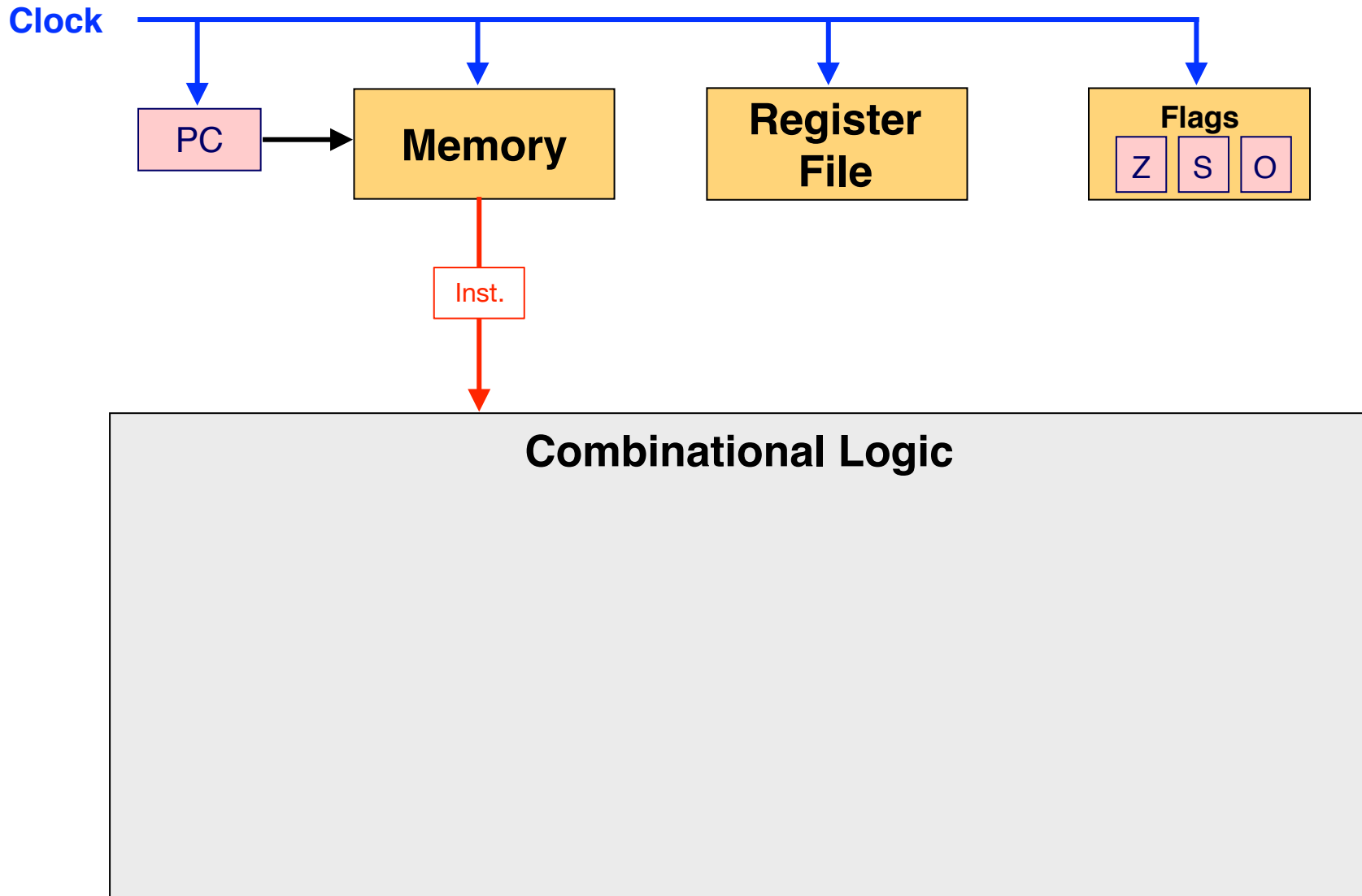
Microarchitecture (So far)



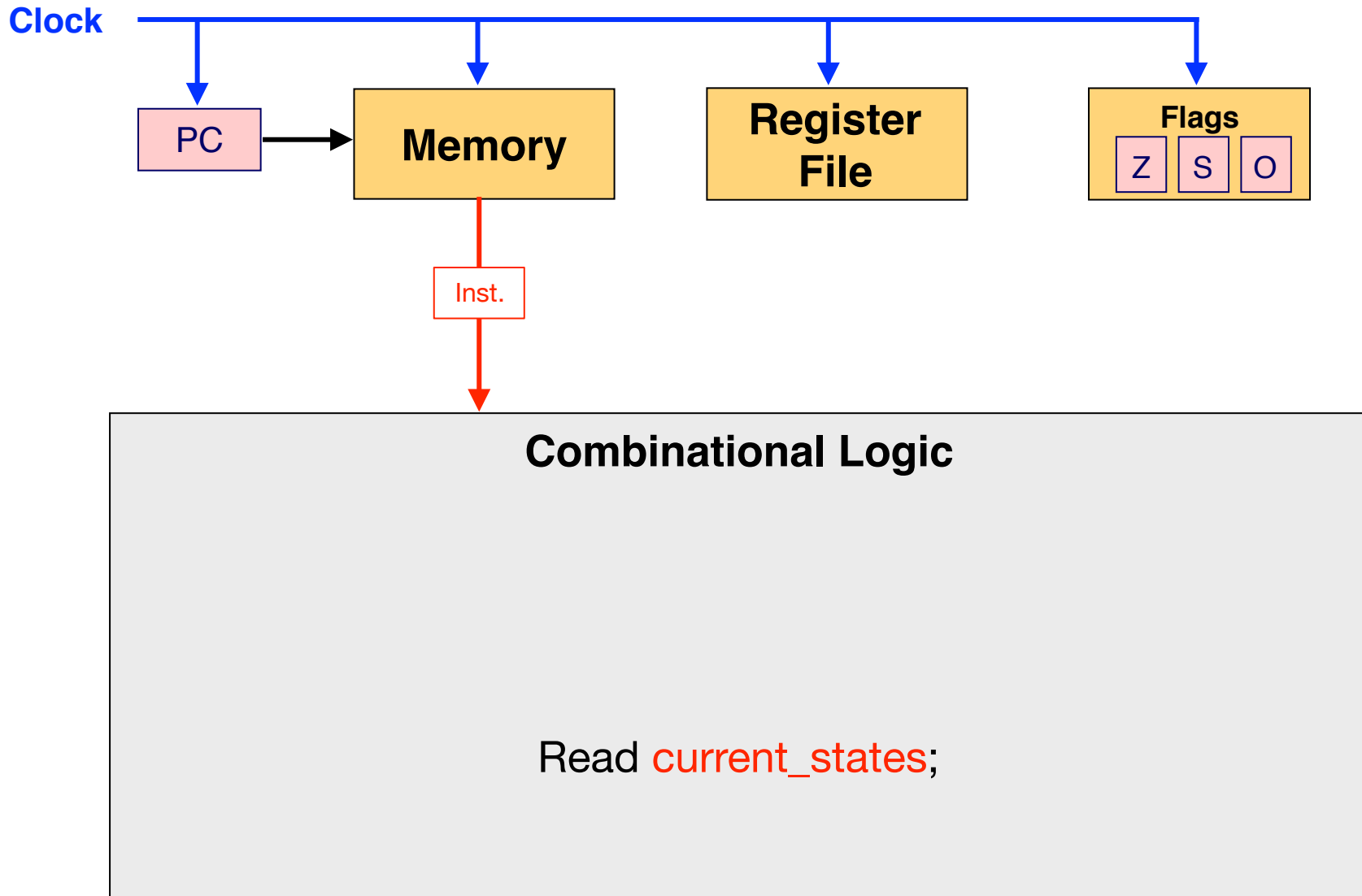
Microarchitecture (So far)



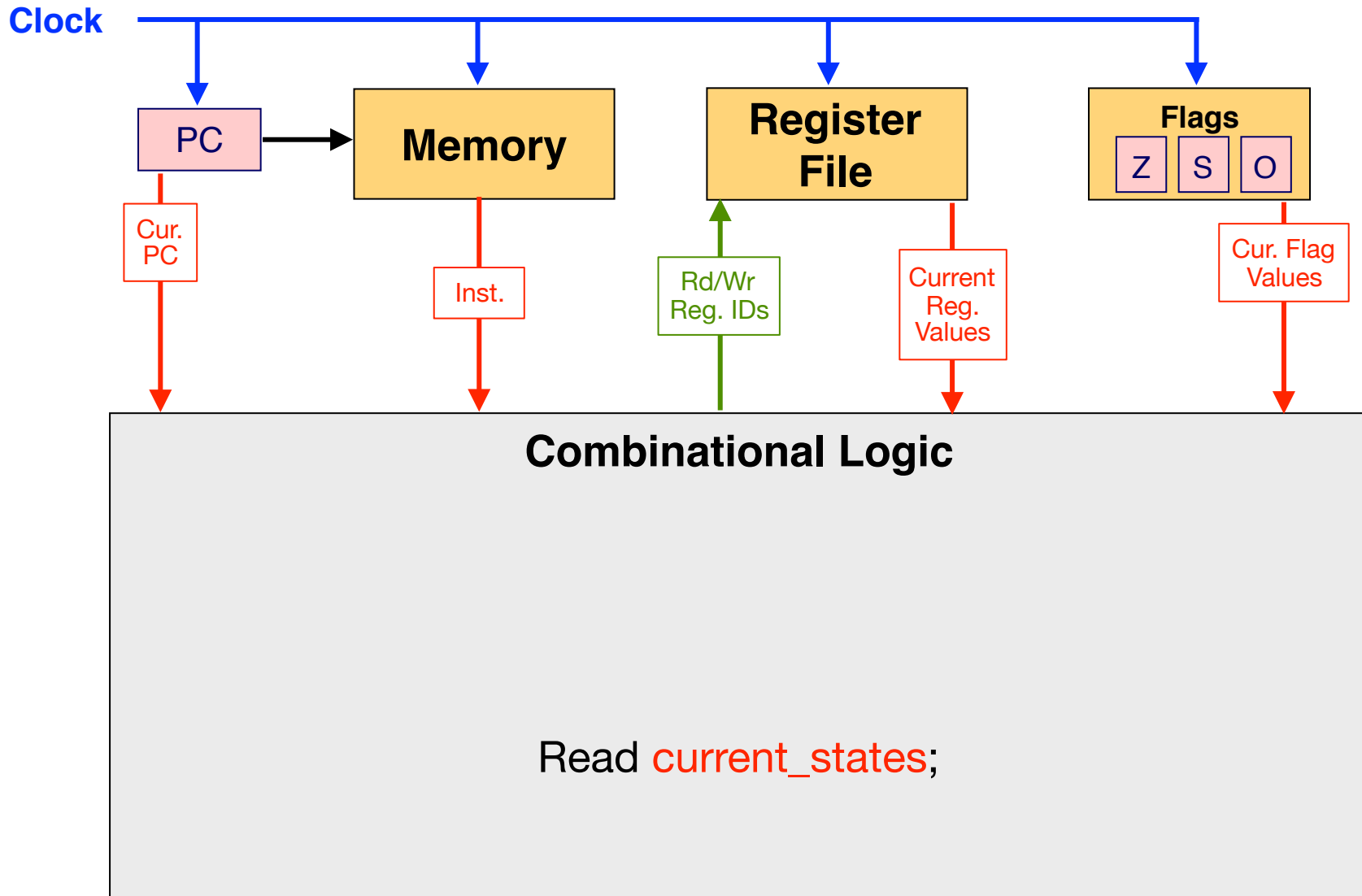
Microarchitecture (So far)



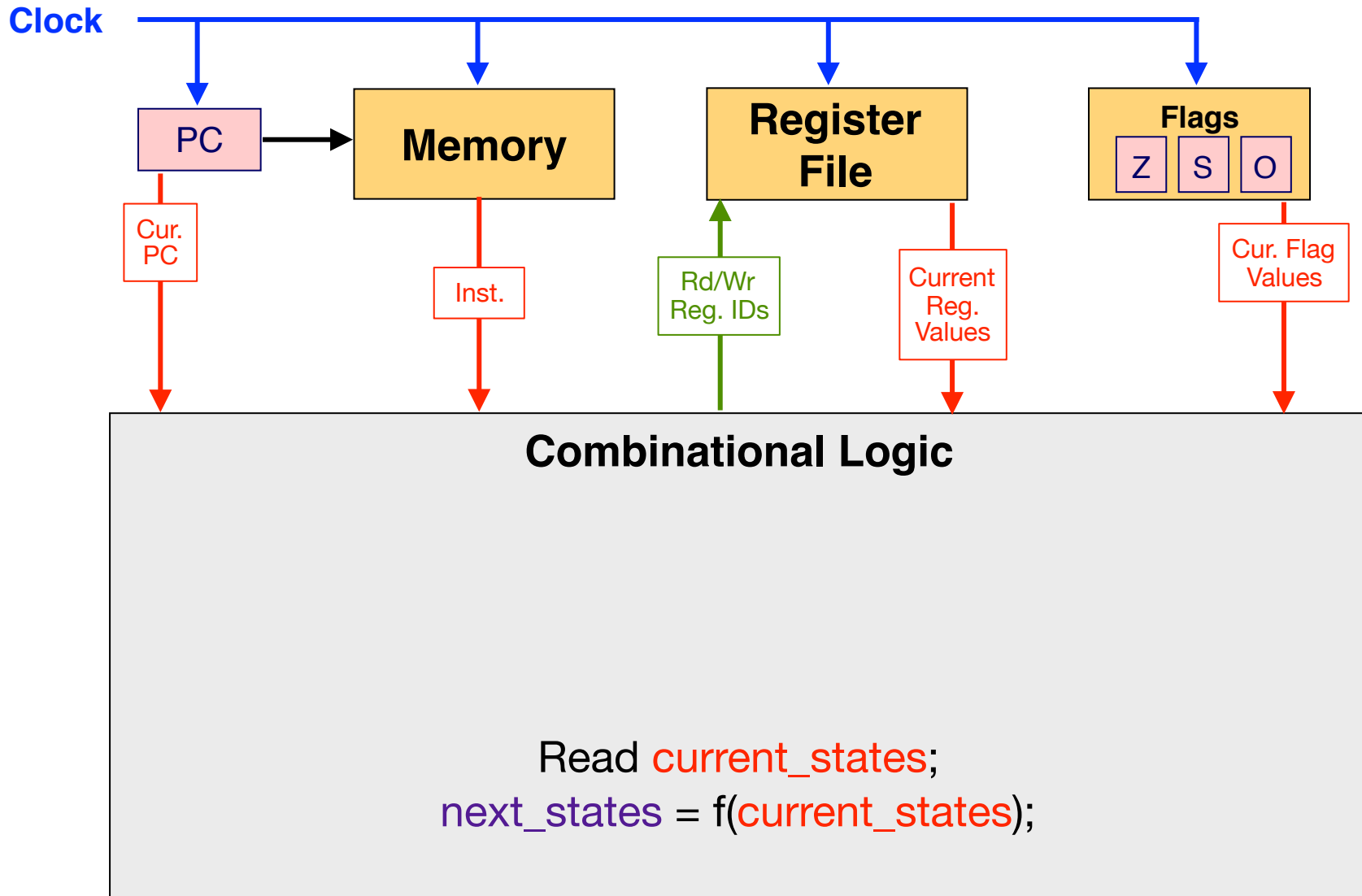
Microarchitecture (So far)



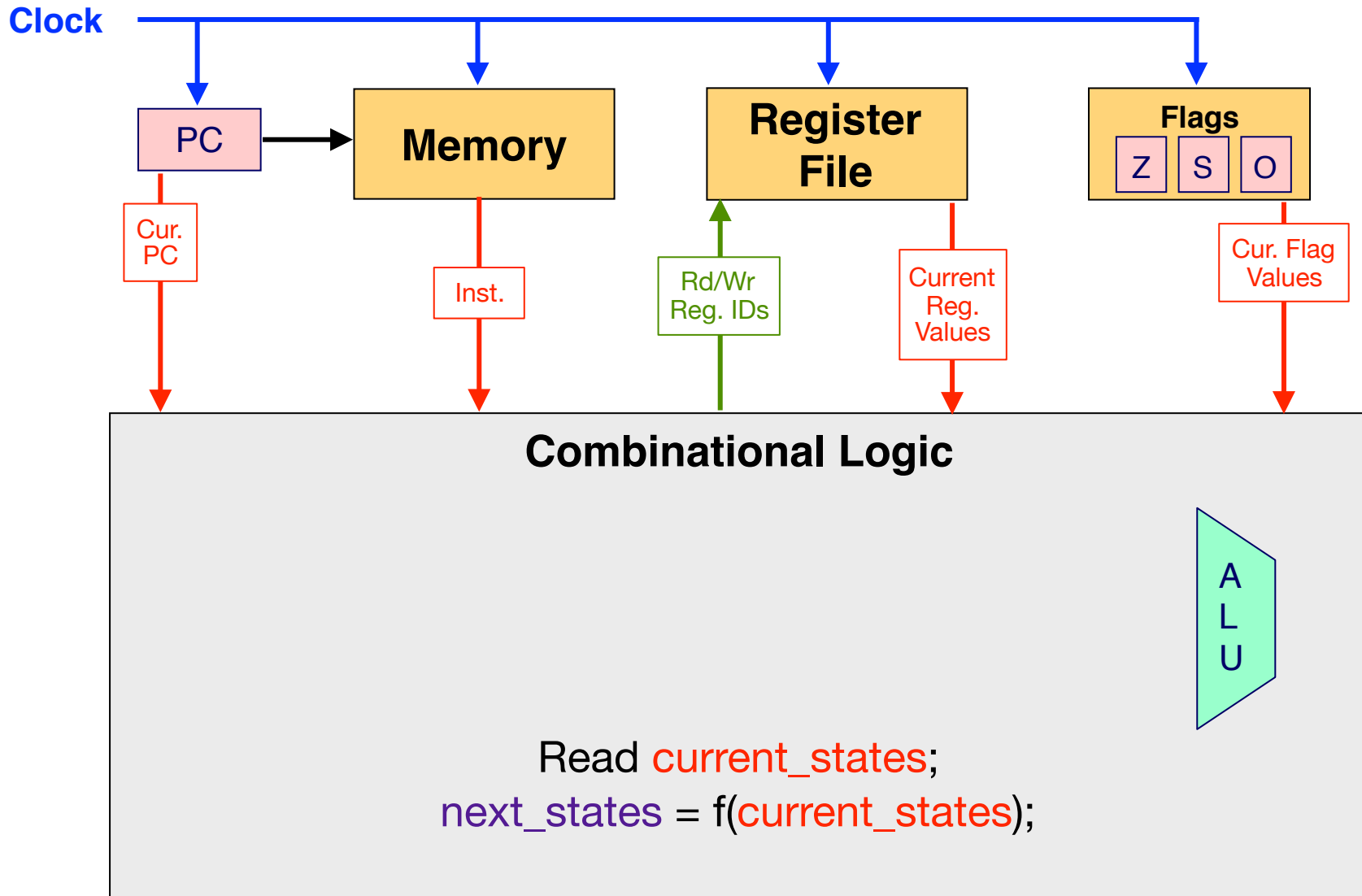
Microarchitecture (So far)



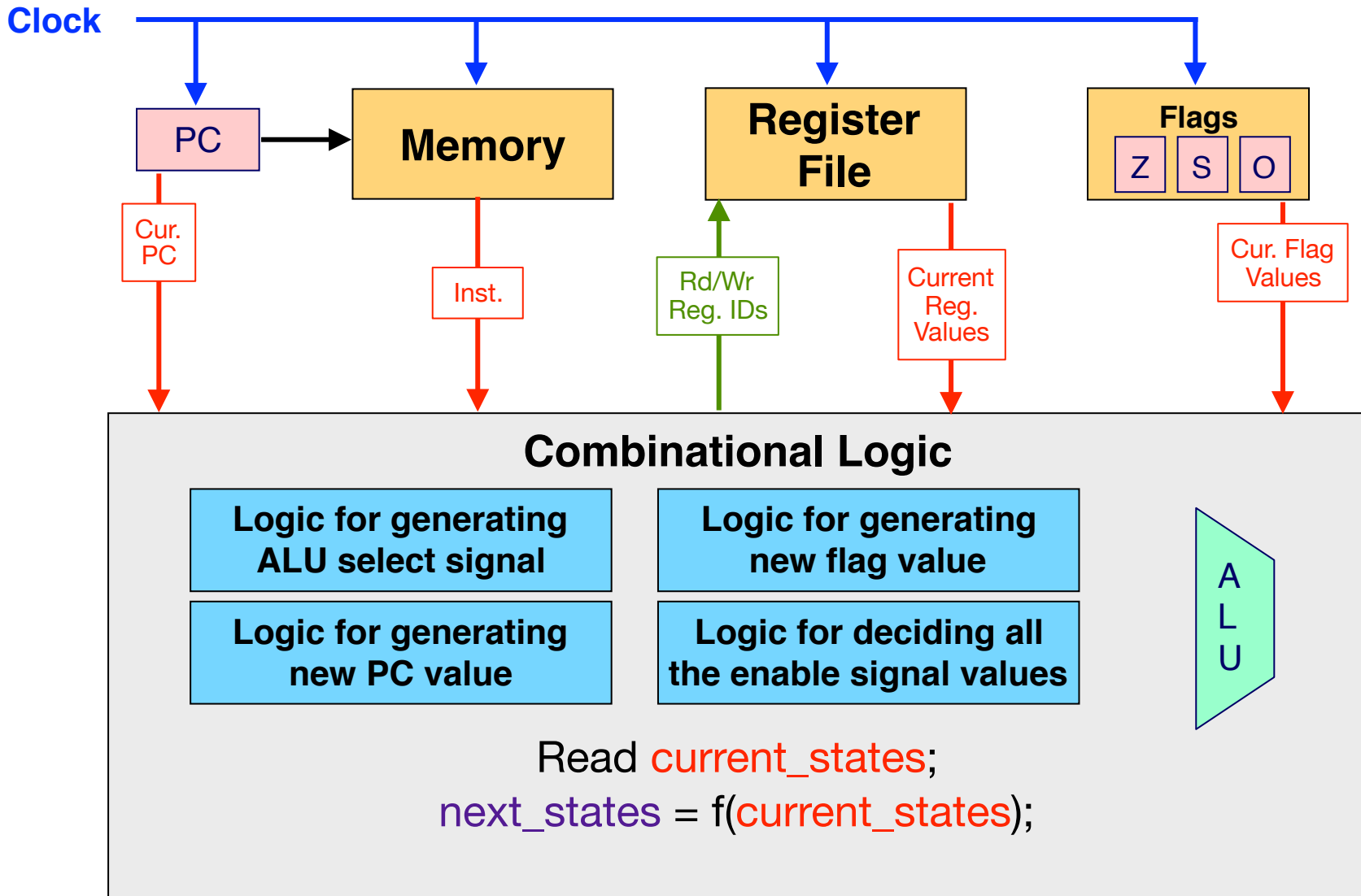
Microarchitecture (So far)



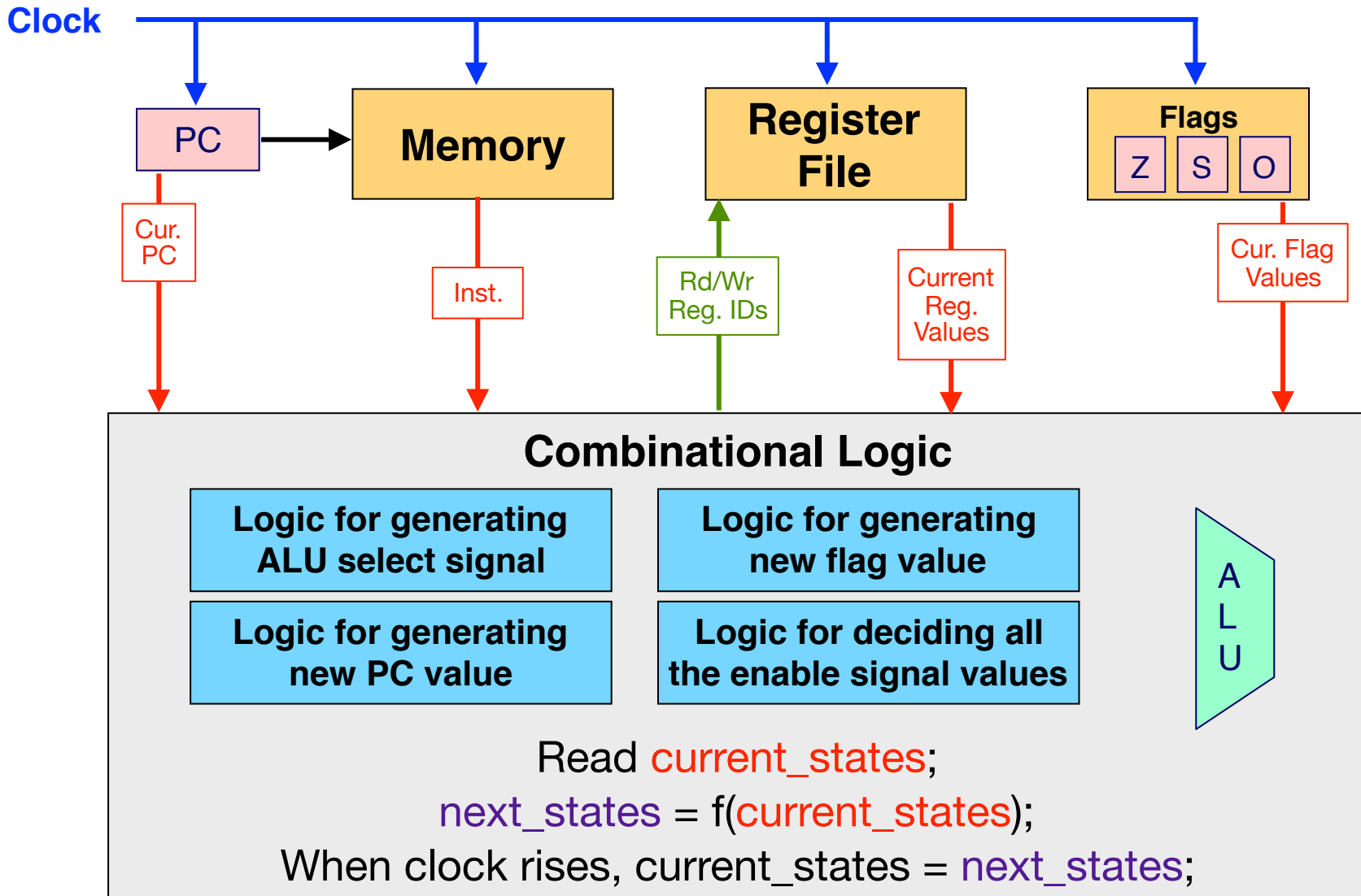
Microarchitecture (So far)



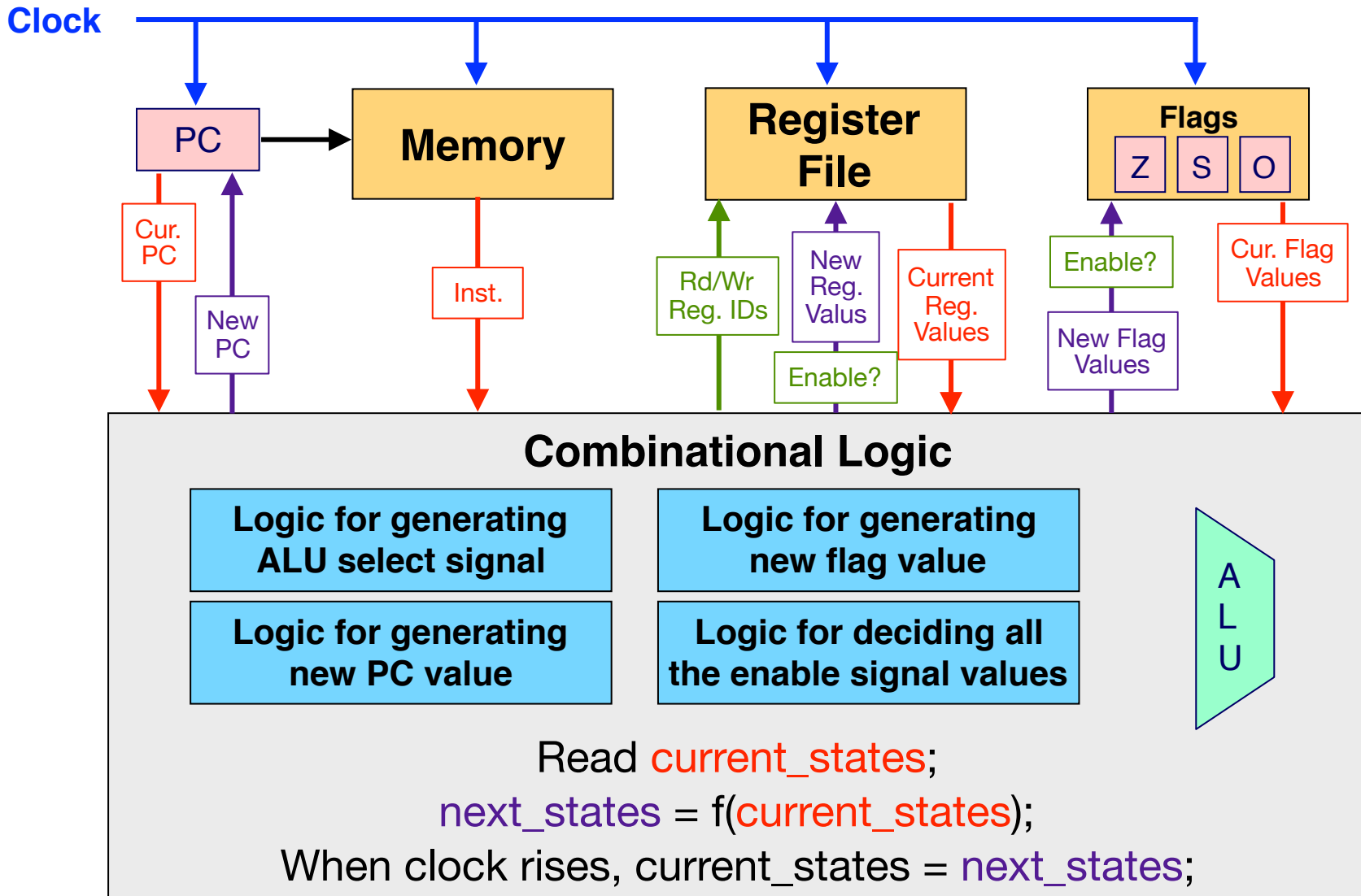
Microarchitecture (So far)



Microarchitecture (So far)

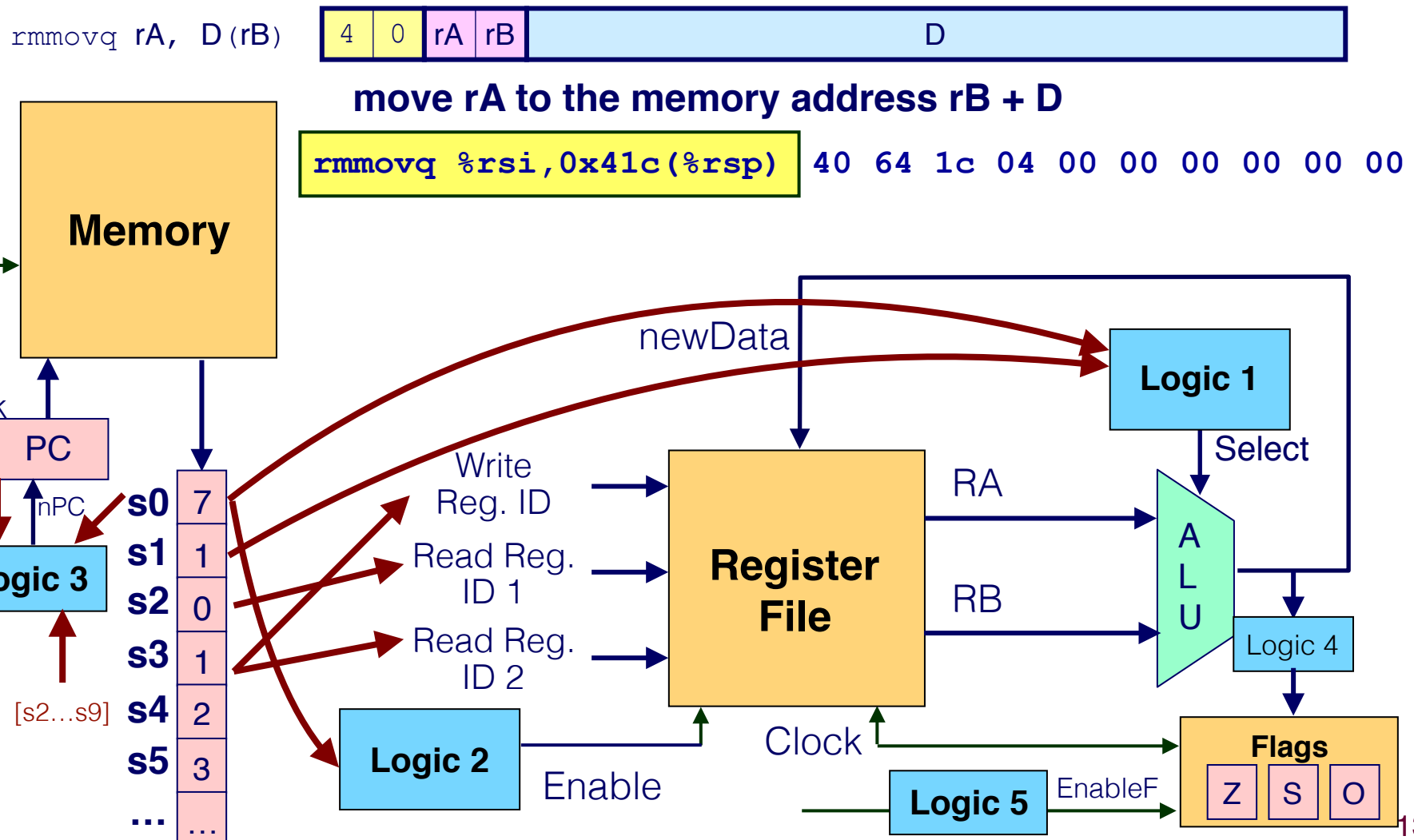


Microarchitecture (So far)



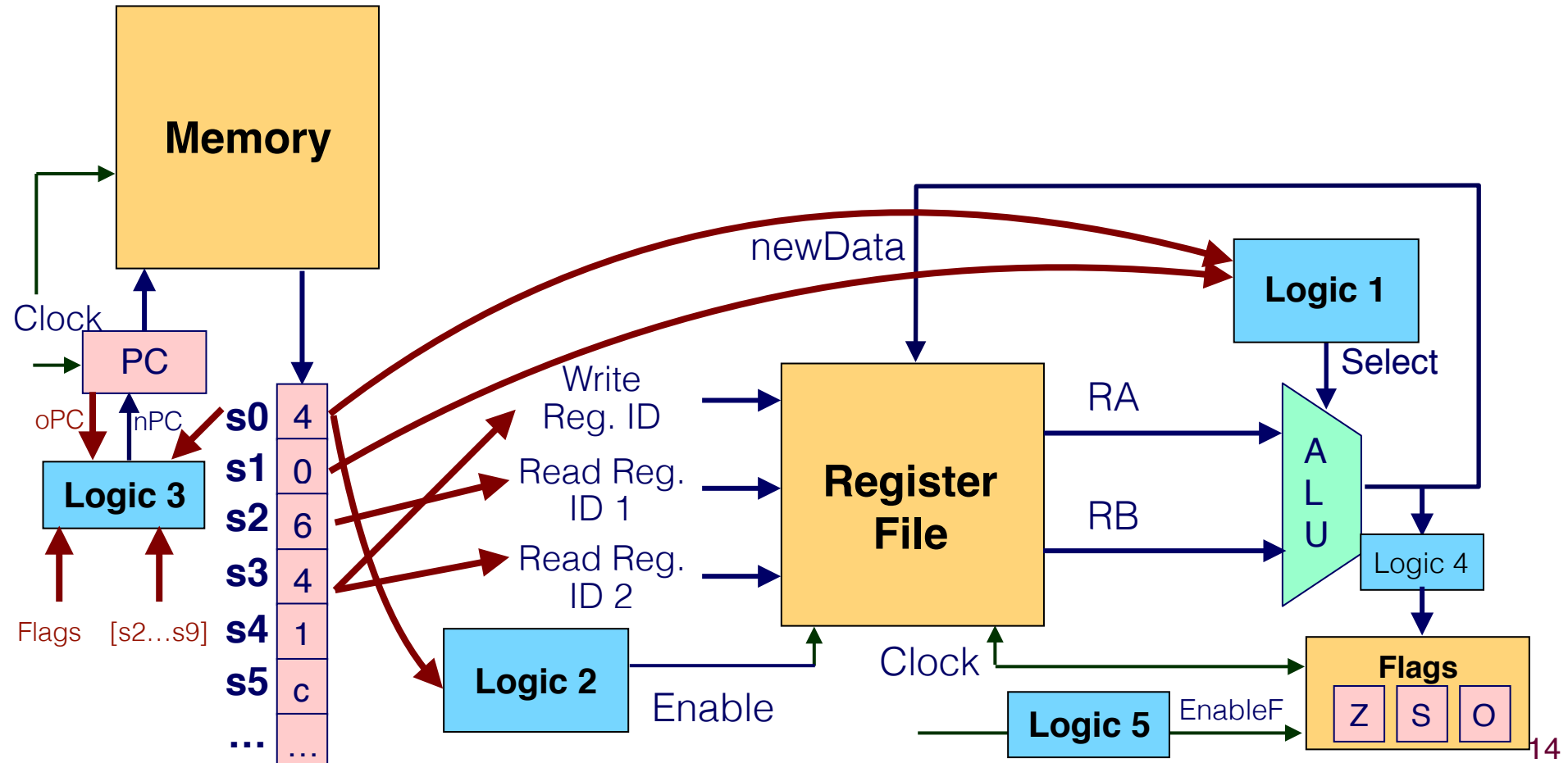
Executing a MOV instruction

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



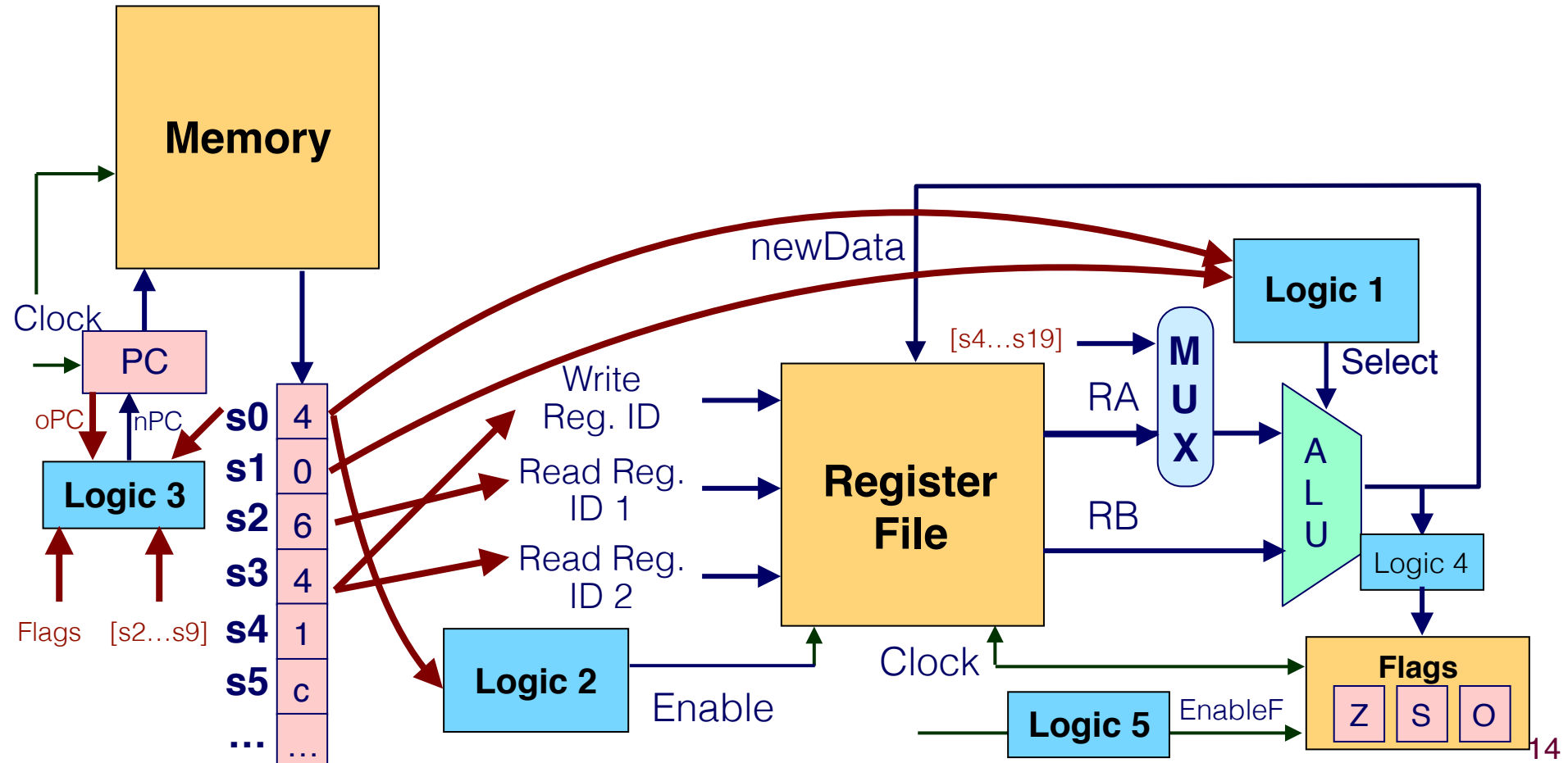
move rA to the memory address rB + D

rmmovq rA, D(rB)



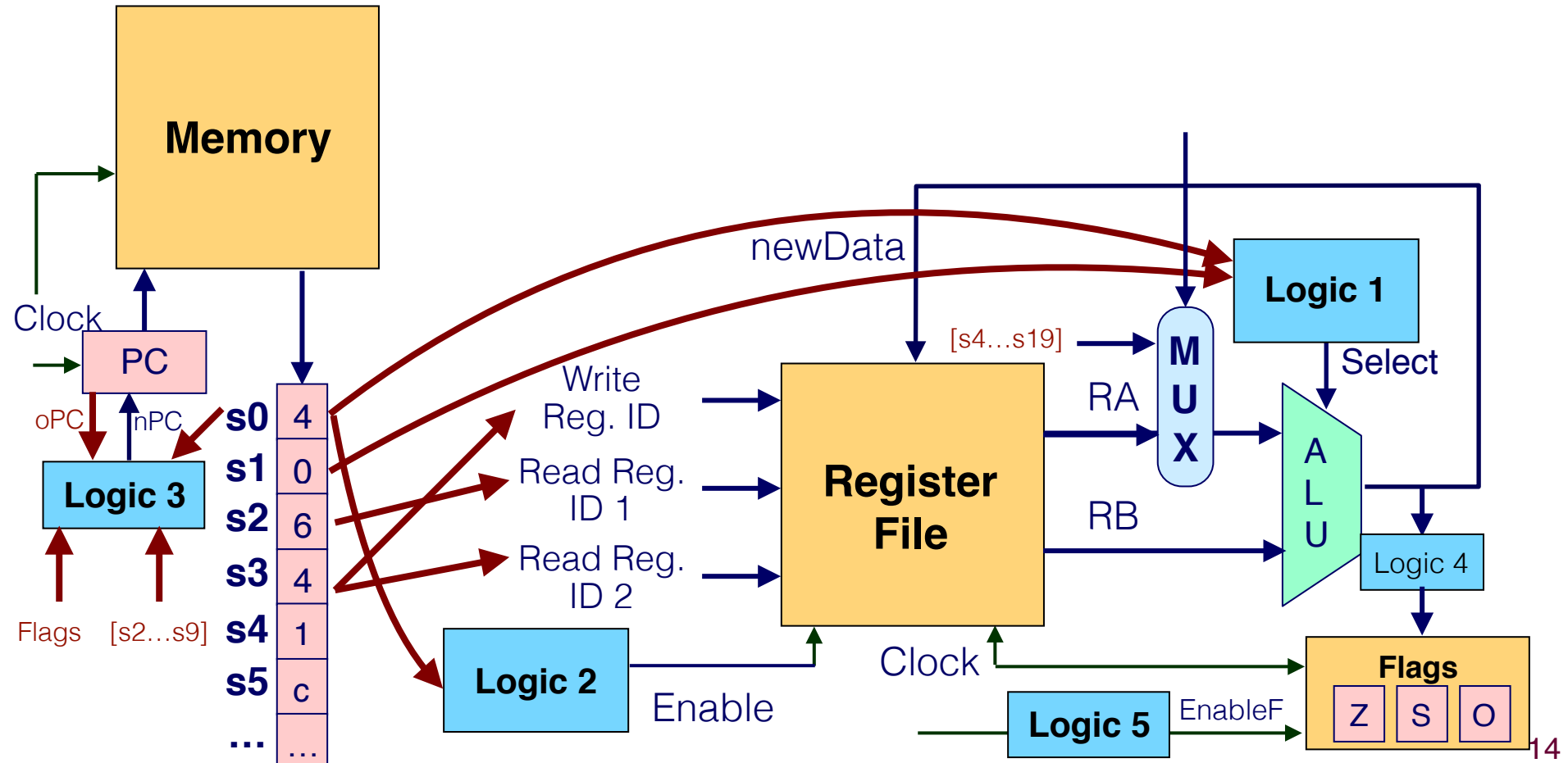
move rA to the memory address rB + D

rmmovq rA, D(rB)



move rA to the memory address rB + D

rmmovq rA, D(rB)

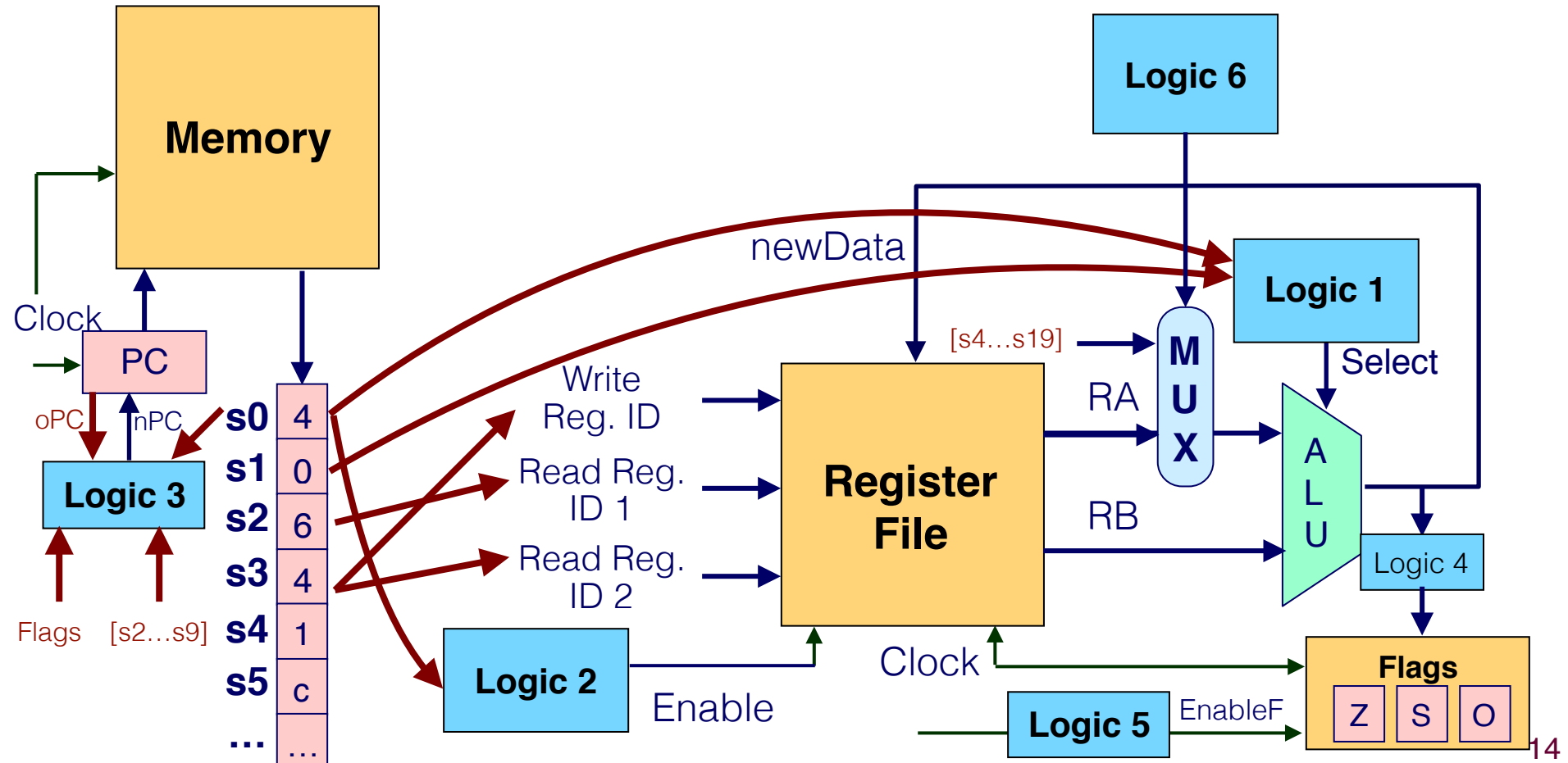


move rA to the memory address rB + D

rmmovq rA, D(rB)



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

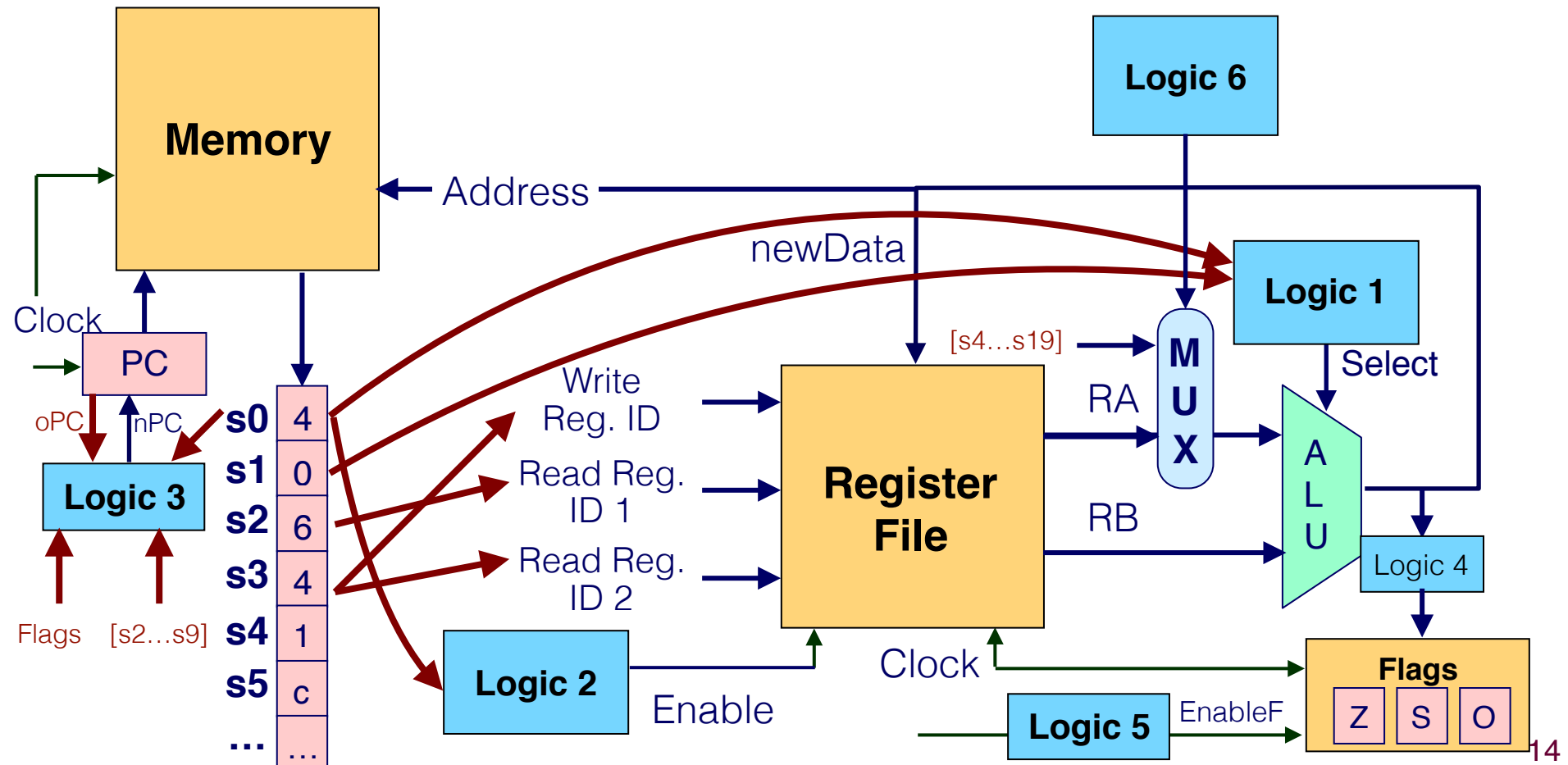


move rA to the memory address rB + D

`rmmovq rA, D(rB)`



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

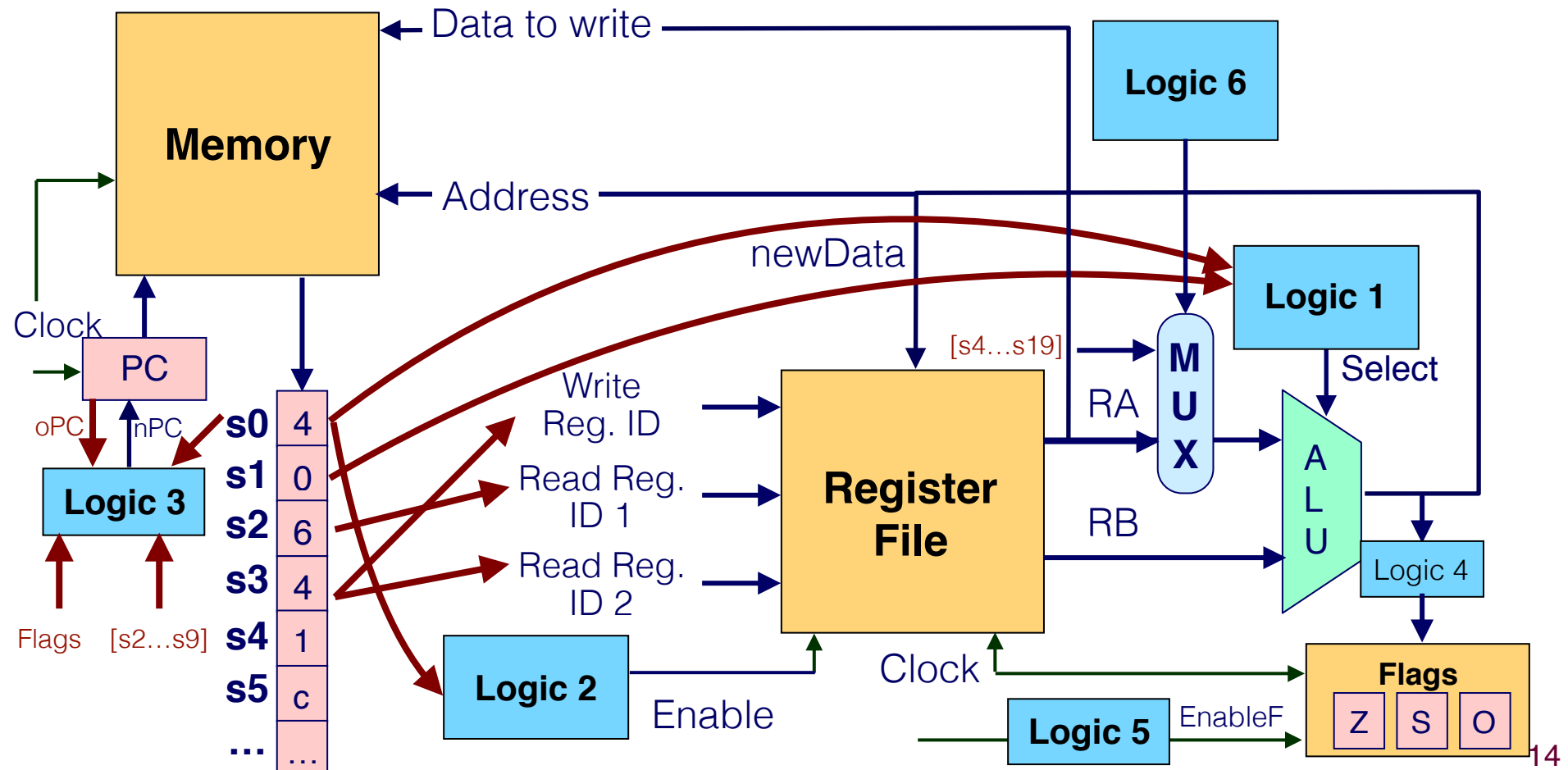


move rA to the memory address rB + D

`rmmovq rA, D(rB)`



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

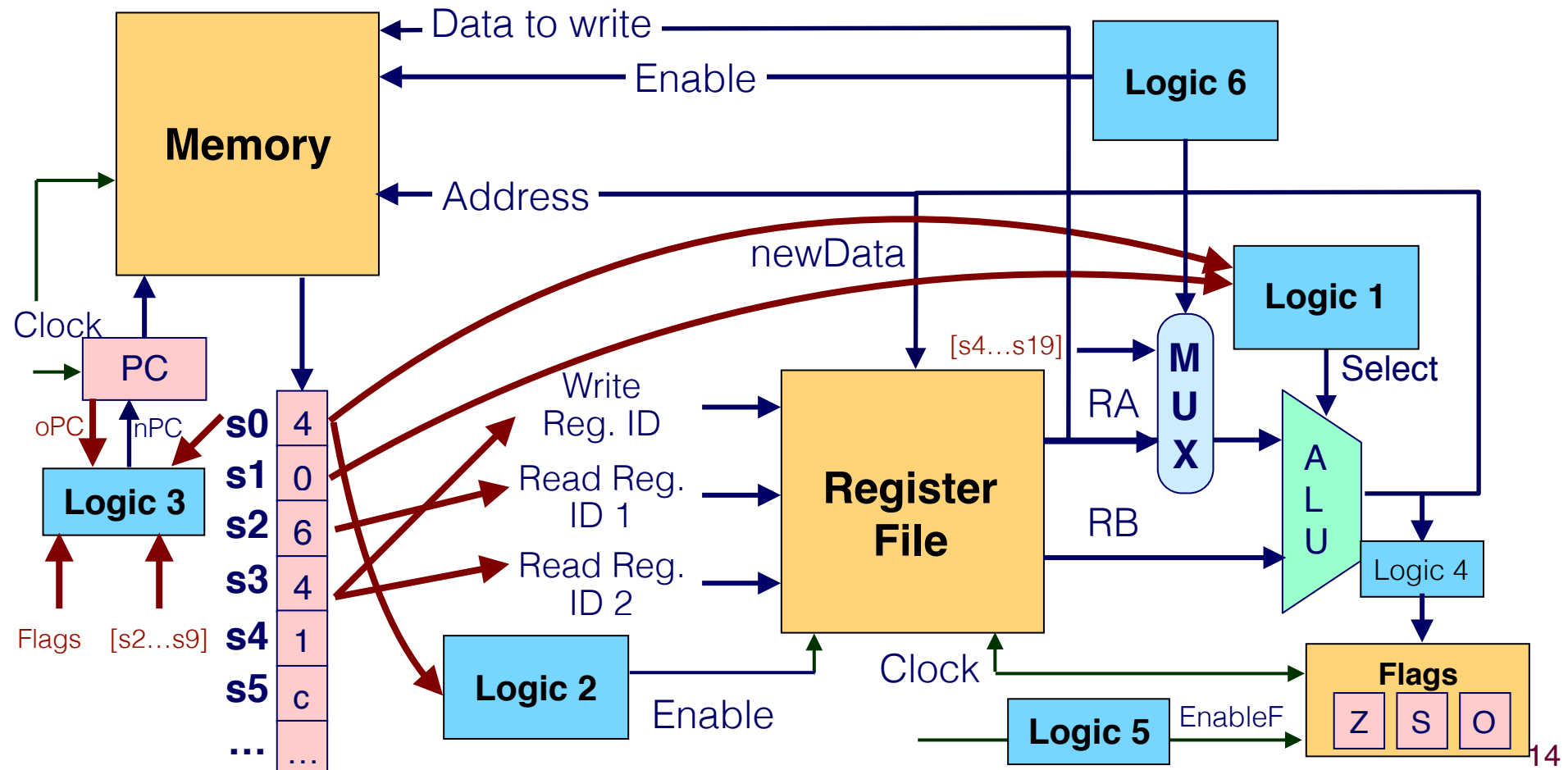


move rA to the memory address rB + D

rmmovq rA, D(rB)



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

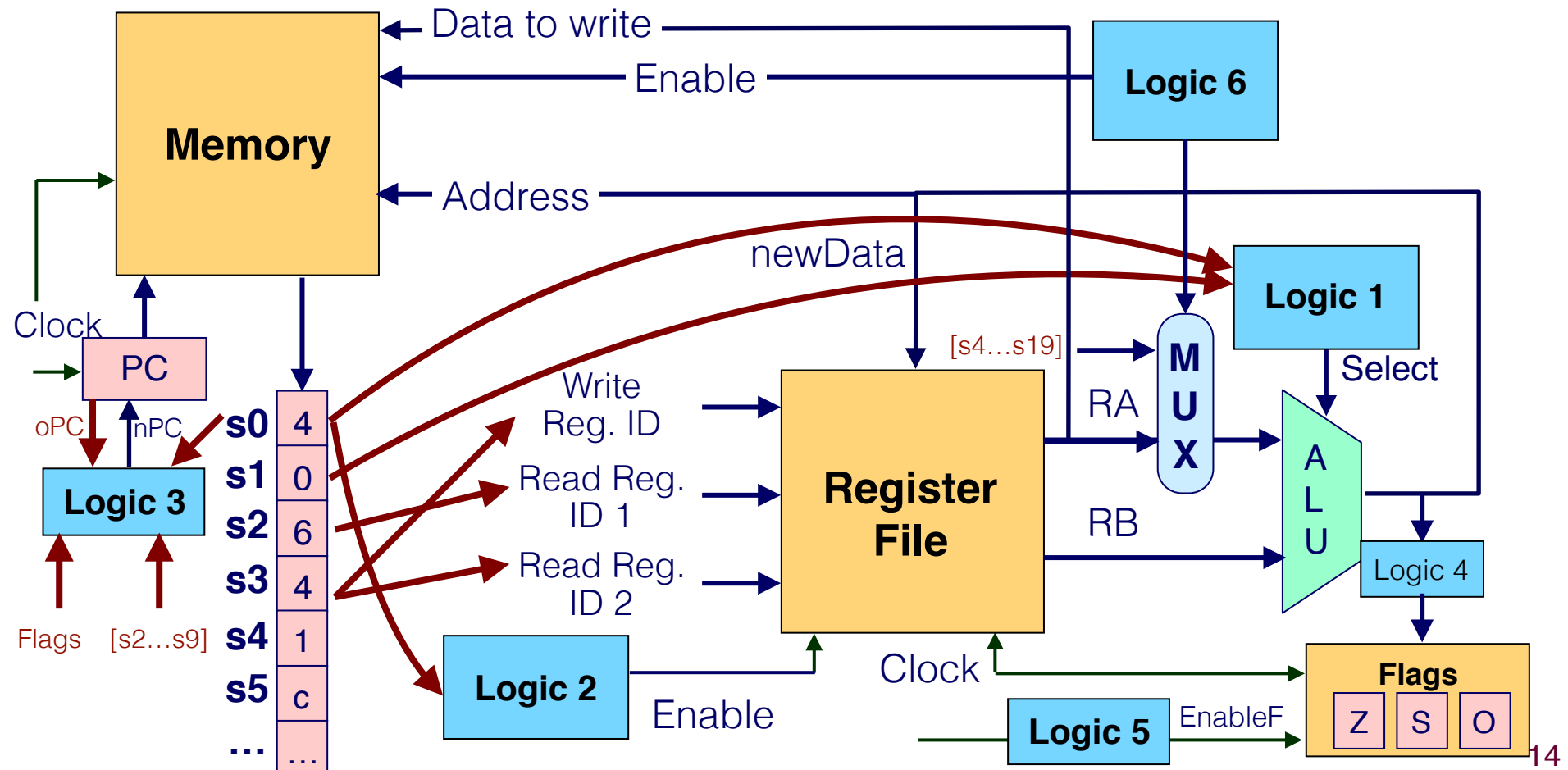


move rA to the memory address rB + D

`rmmovq rA, D(rB)`

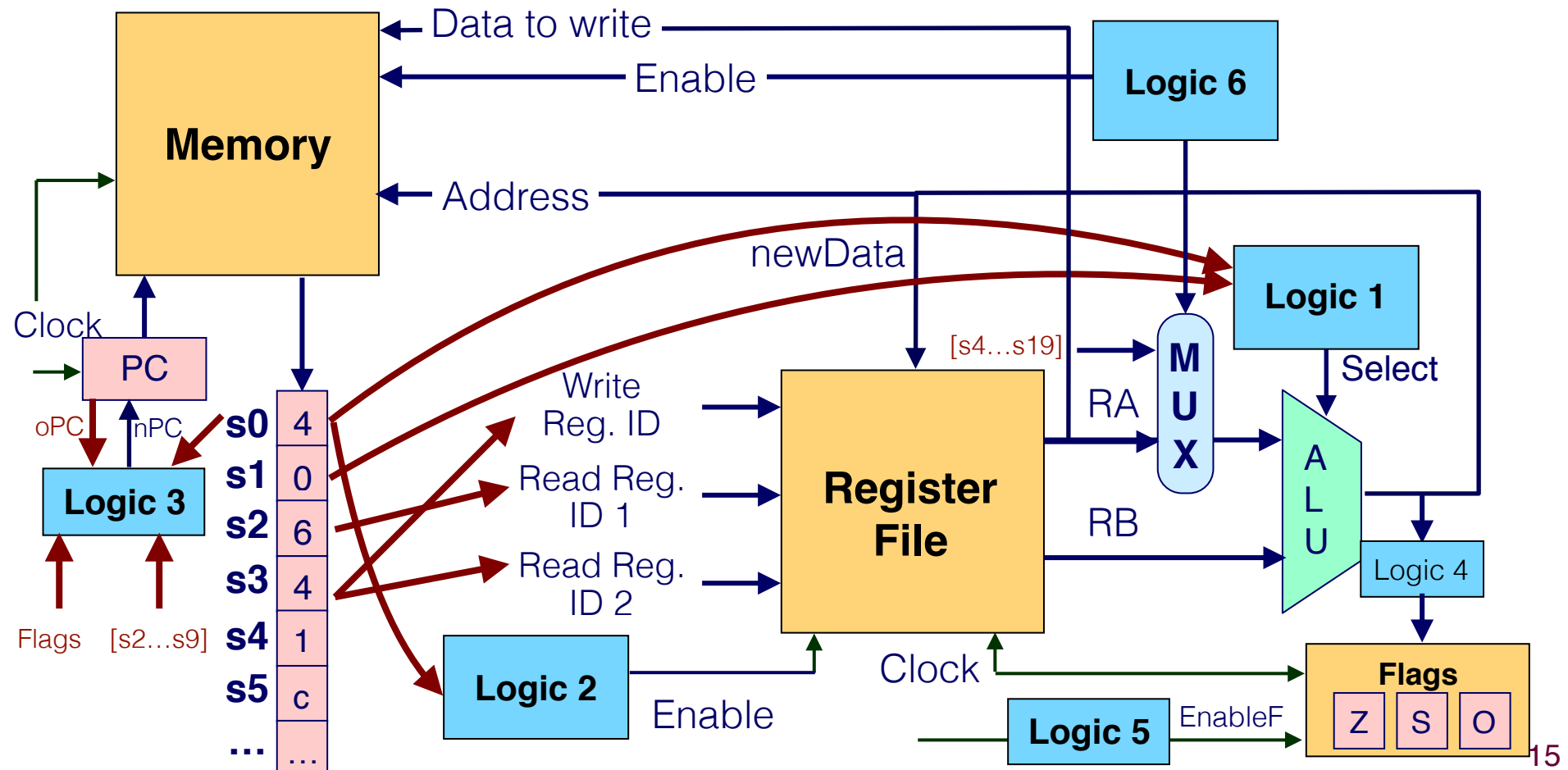


- Need new logic (Logic 6) to select the input to the ALU for Enable.
- How about other logics?



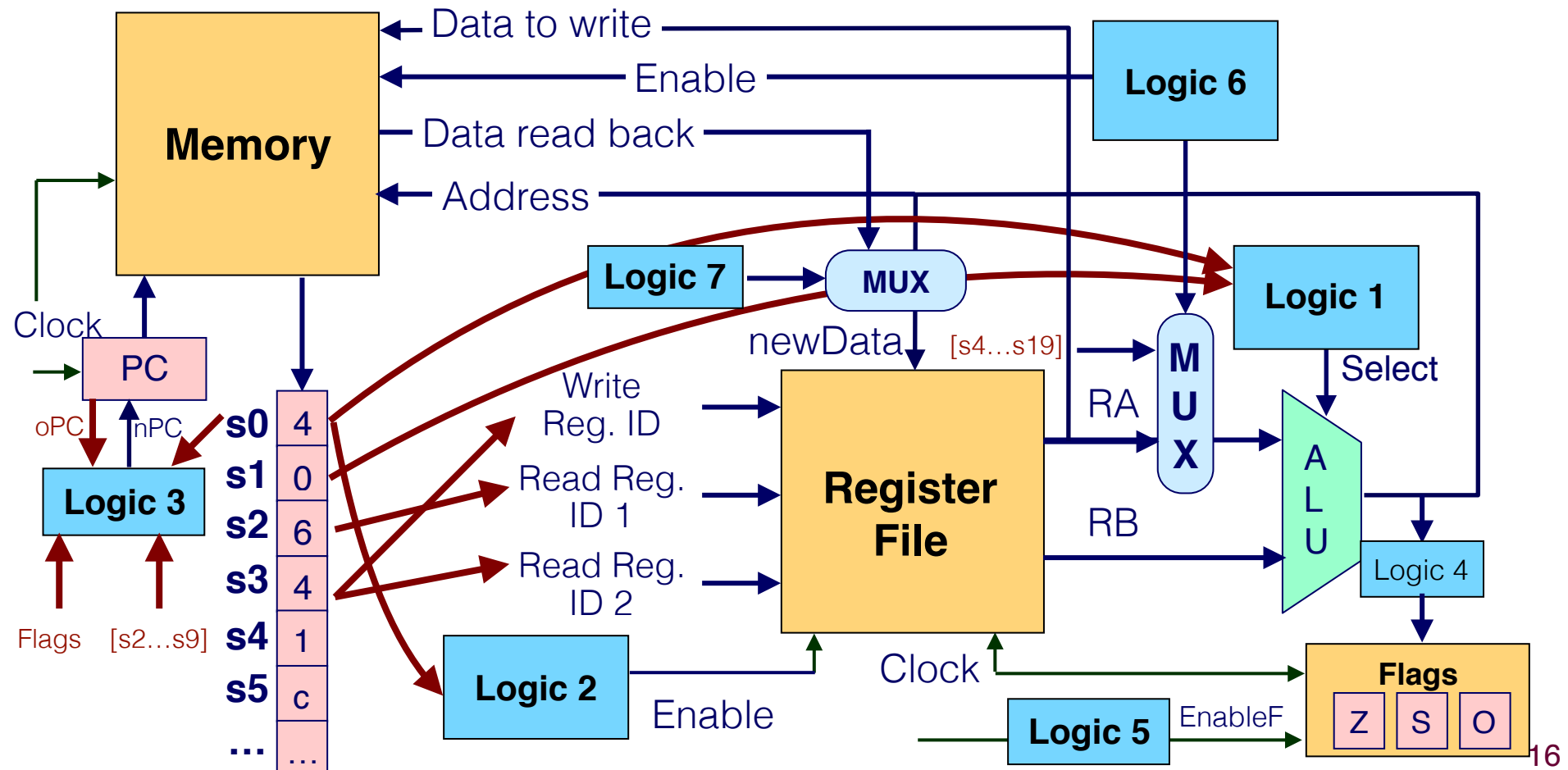
How About Memory to Register MOV?

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

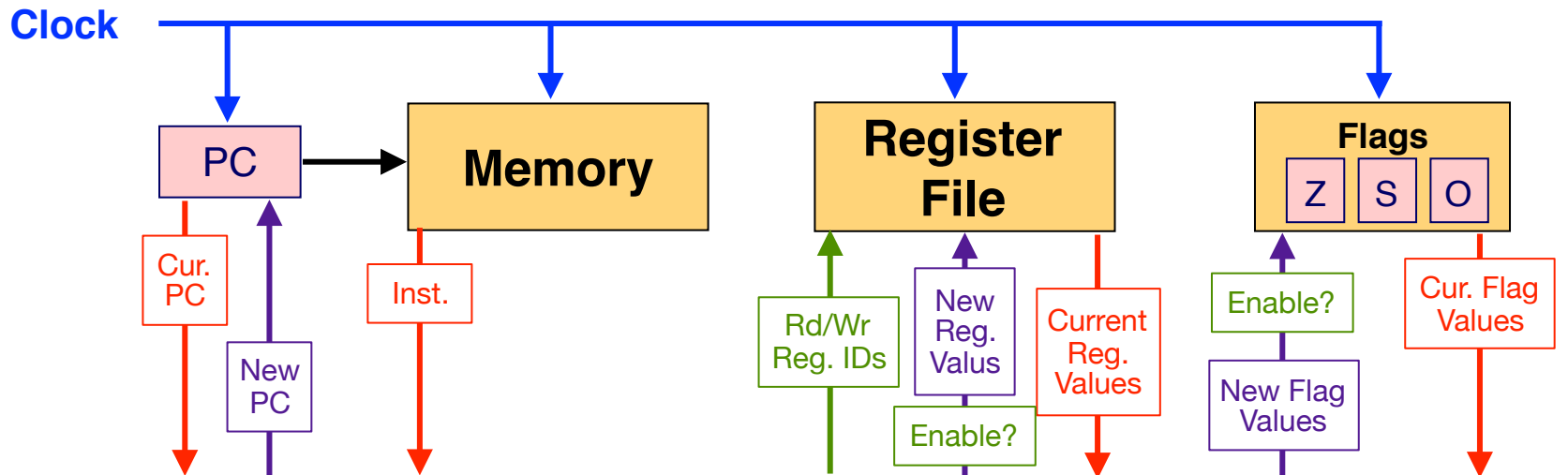


How About Memory to Register MOV?

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



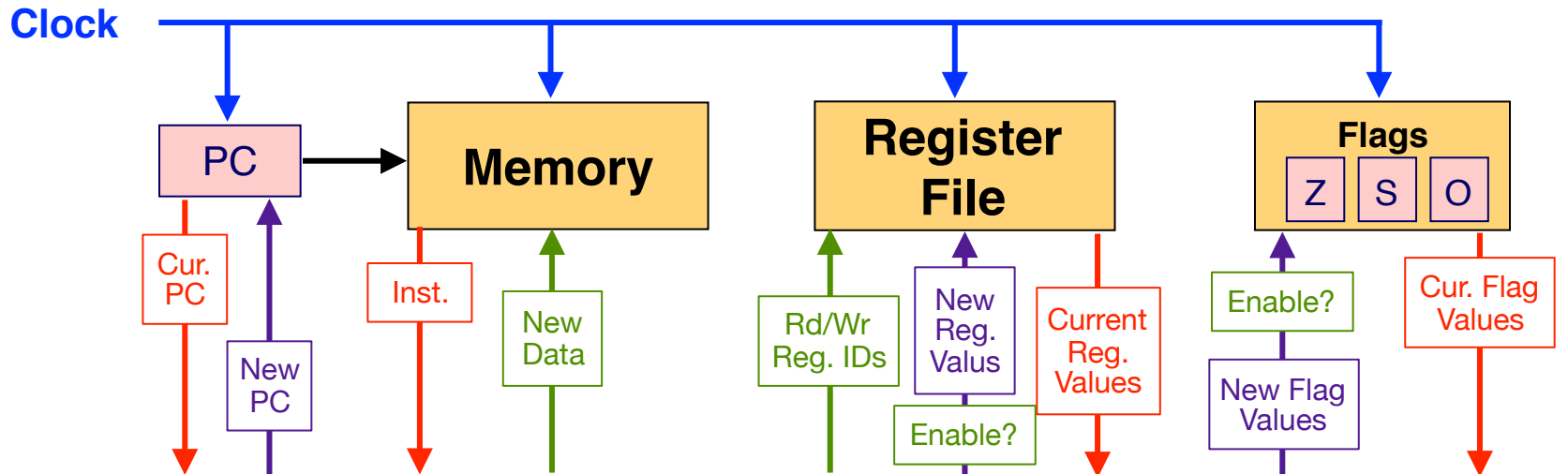
Microarchitecture (with MOV)



Combinational Logic

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

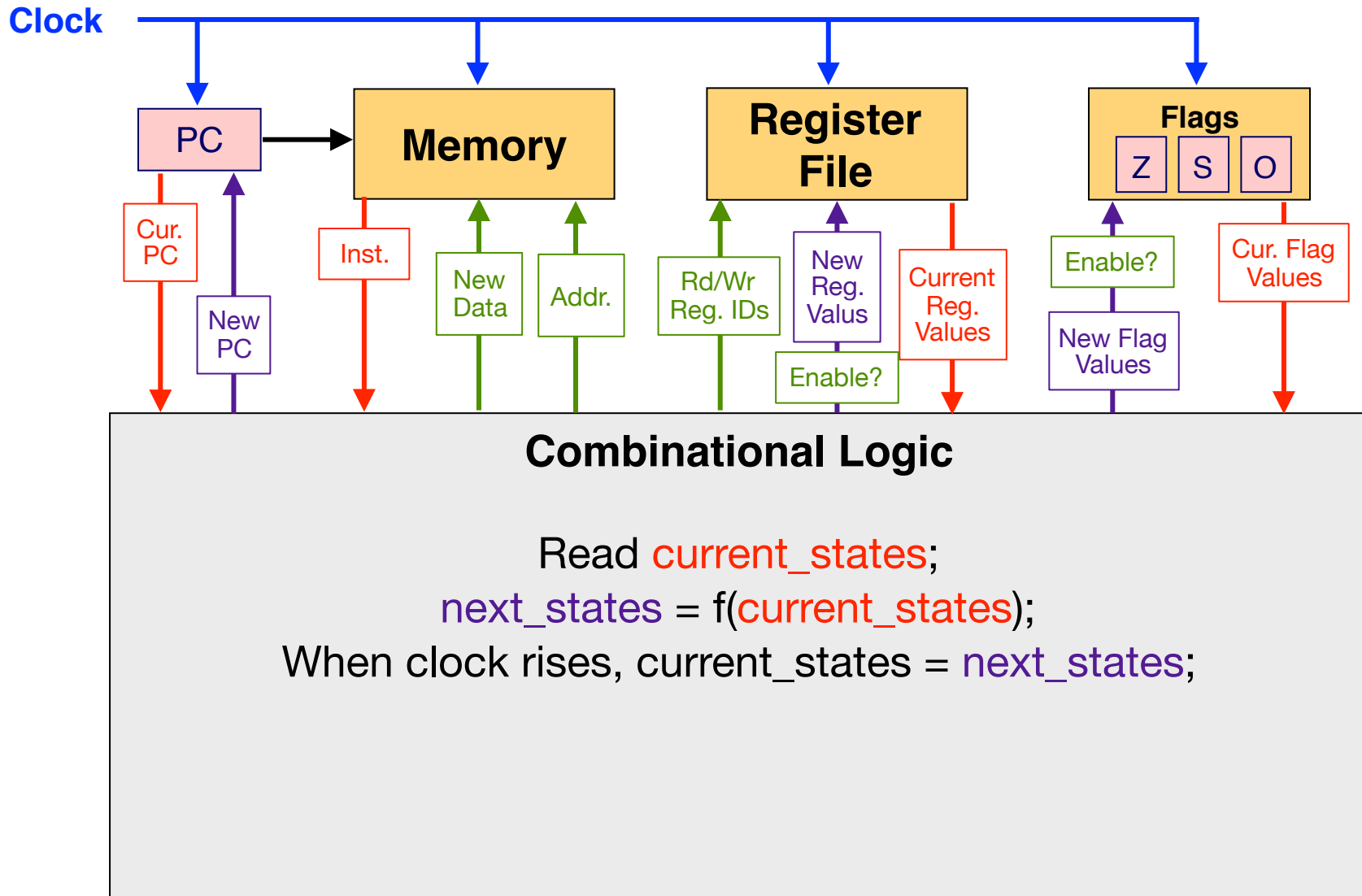
Microarchitecture (with MOV)



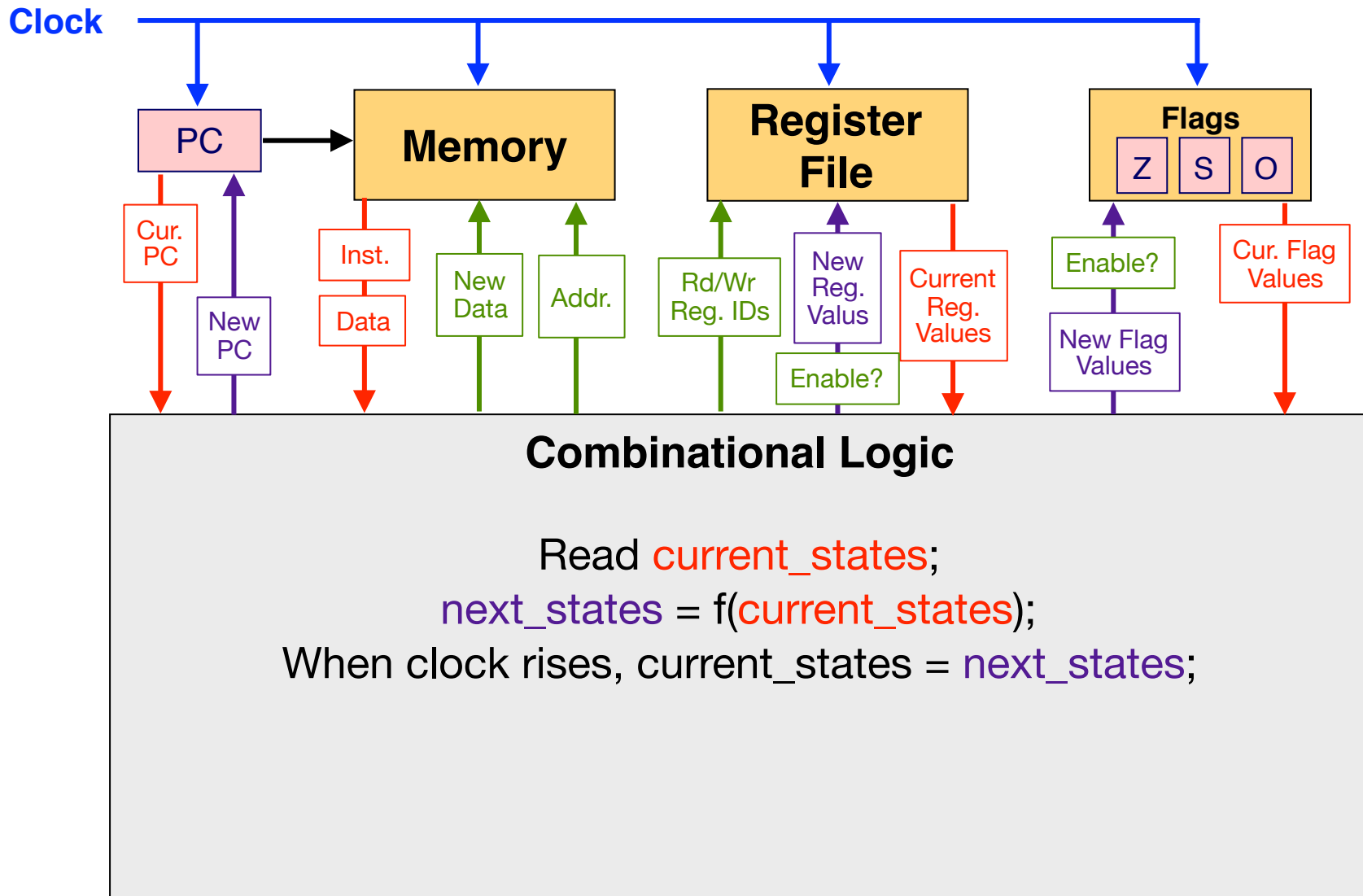
Combinational Logic

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

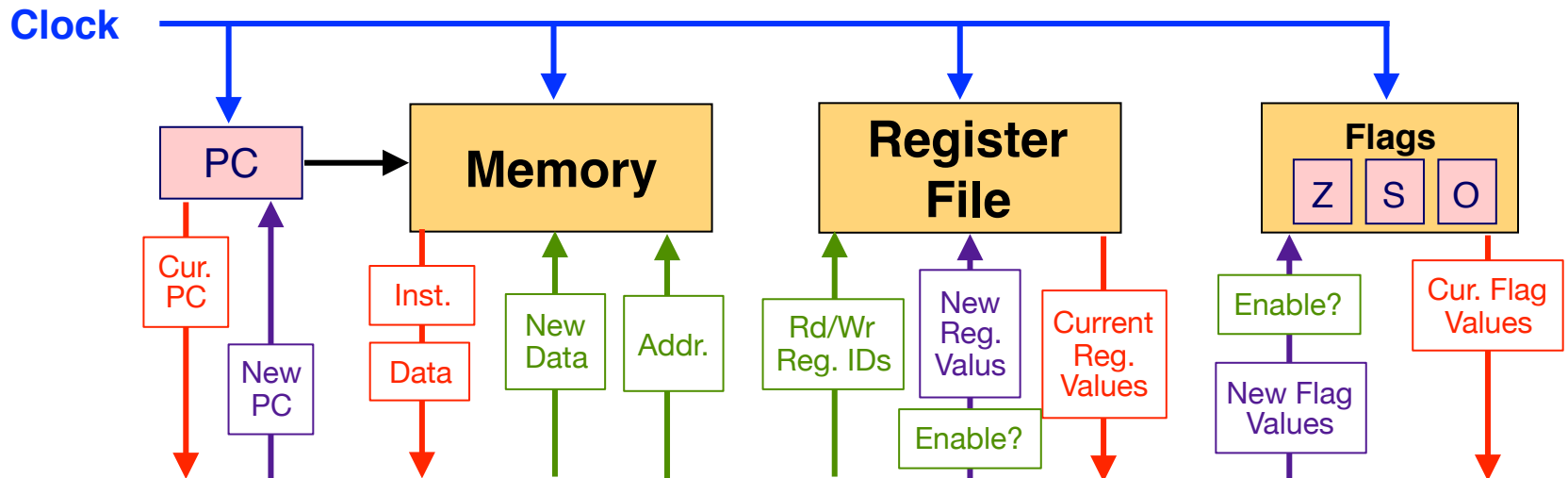
Microarchitecture (with MOV)



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 close rises
```

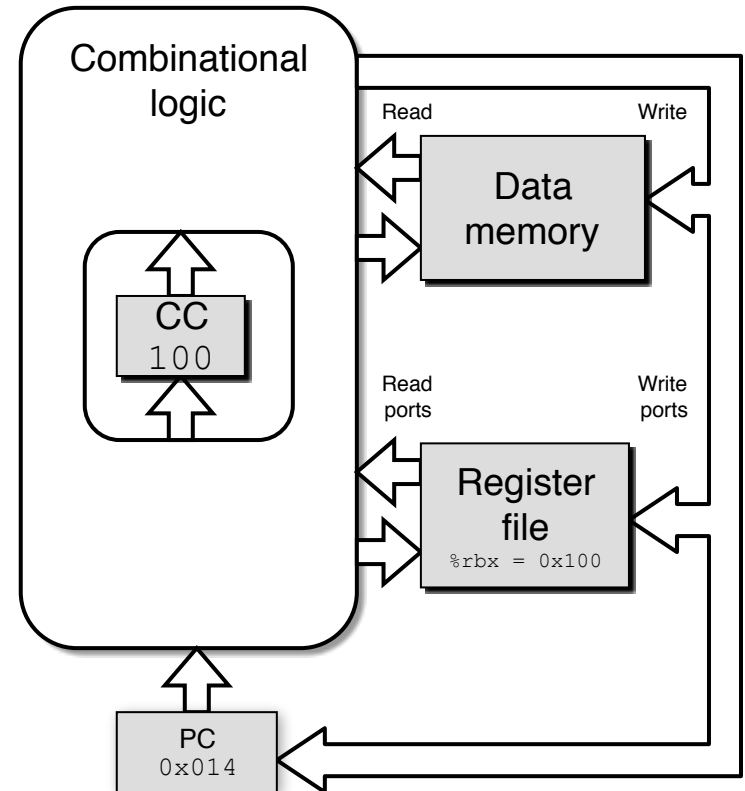
Microarchitecture Overview

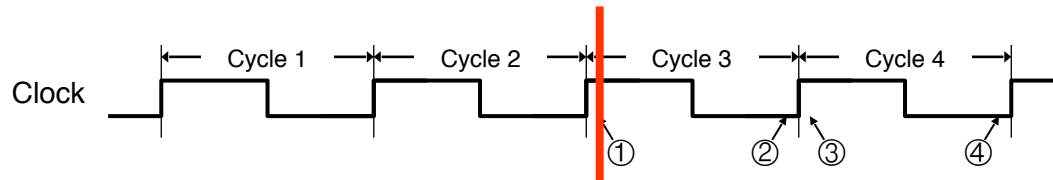
Think of it as a state machine

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

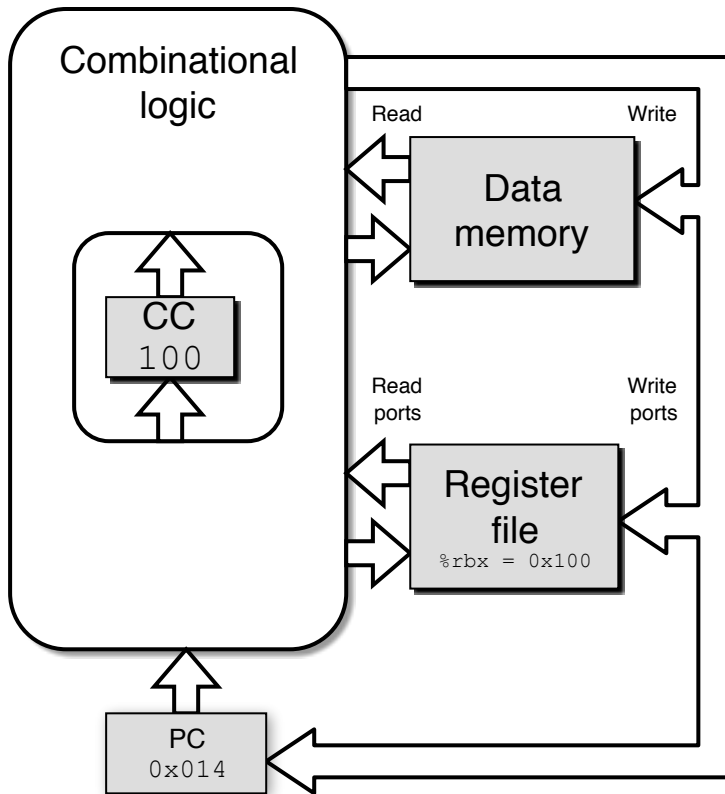
States (All updated as clock rises)

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

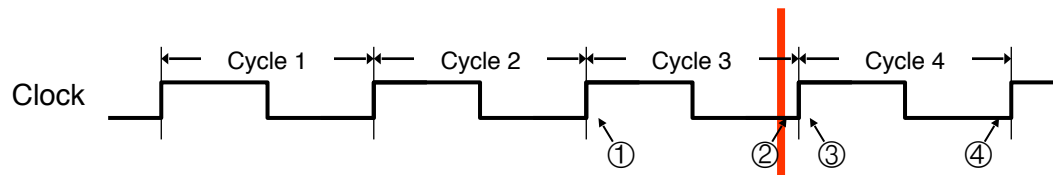




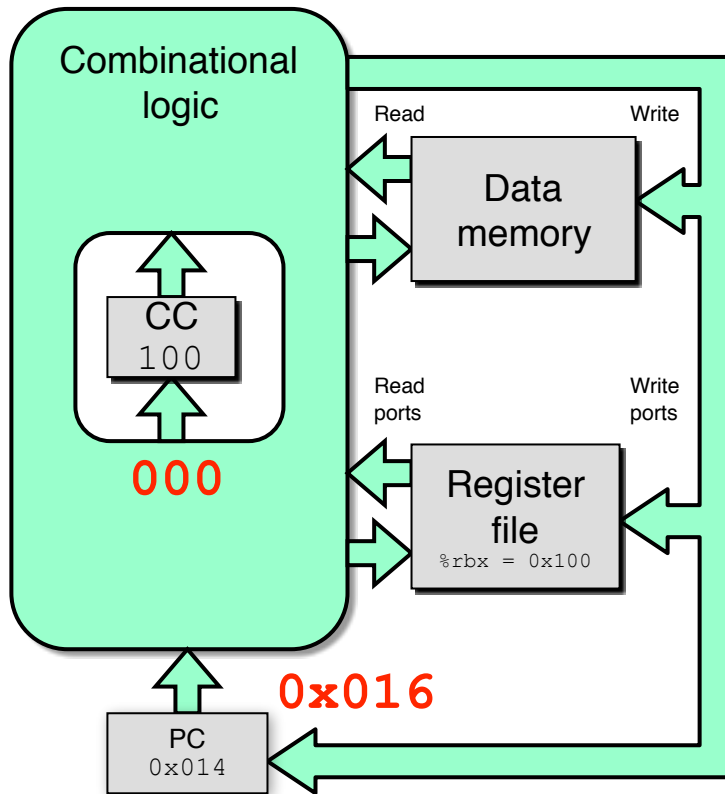
Cycle 1:	0x000:	<code>irmovq \$0x100,%rbx</code>	# %rbx <-- 0x100
Cycle 2:	0x00a:	<code>irmovq \$0x200,%rdx</code>	# %rdx <-- 0x200
Cycle 3:	0x014:	<code>addq %rdx,%rbx</code>	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	<code>je dest</code>	# Not taken
Cycle 5:	0x01f:	<code>rmmovq %rbx,0(%rdx)</code>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



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

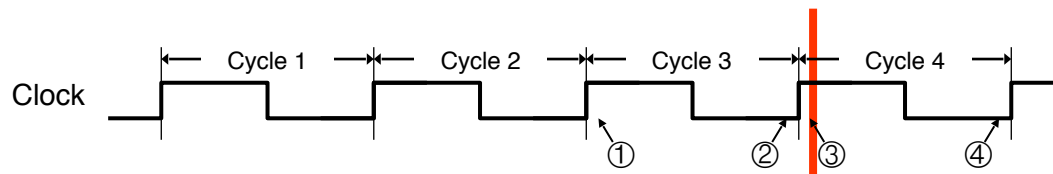


Cycle 1:	0x000:	<code>irmovq \$0x100,%rbx</code>	# %rbx <-- 0x100
Cycle 2:	0x00a:	<code>irmovq \$0x200,%rdx</code>	# %rdx <-- 0x200
Cycle 3:	0x014:	<code>addq %rdx,%rbx</code>	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	<code>je dest</code>	# Not taken
Cycle 5:	0x01f:	<code>rmmovq %rbx,0(%rdx)</code>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

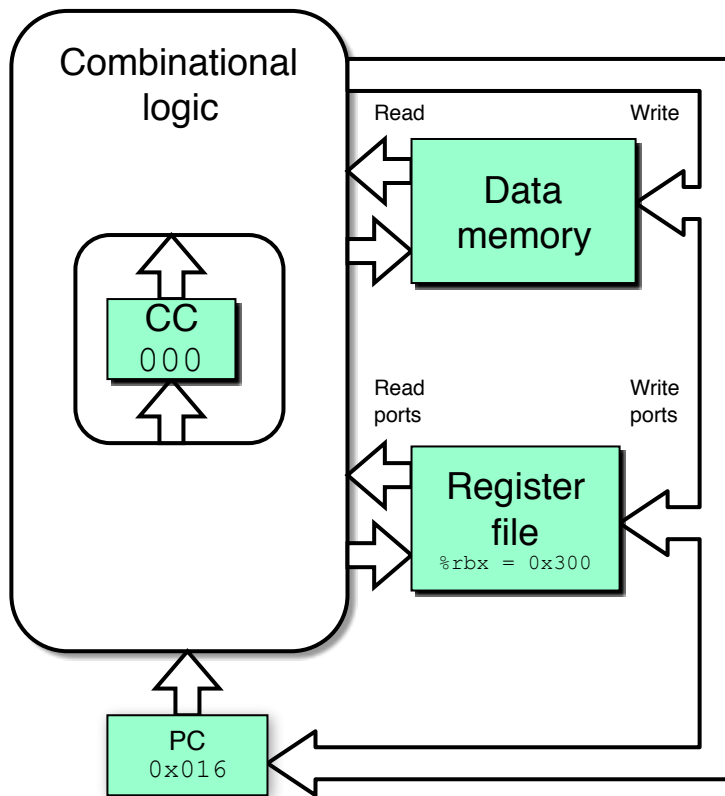


%rbx
<--
0x300

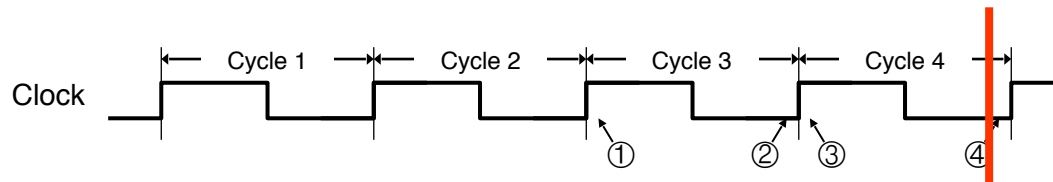
- state set according to second `irmovq` instruction
- combinational logic generates results for `addq` instruction



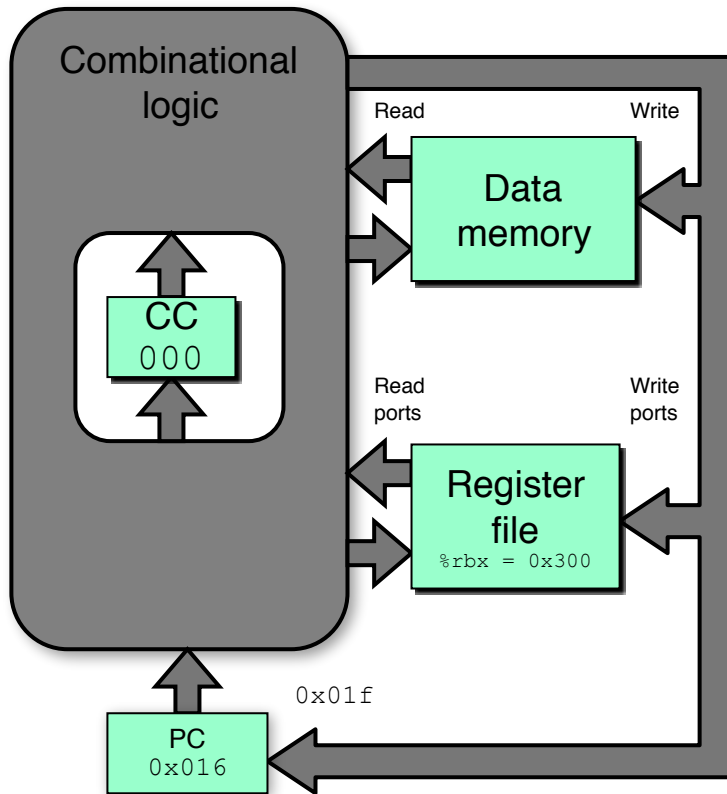
Cycle 1:	0x000:	<code>irmovq \$0x100,%rbx</code>	# %rbx <-- 0x100
Cycle 2:	0x00a:	<code>irmovq \$0x200,%rdx</code>	# %rdx <-- 0x200
Cycle 3:	0x014:	<code>addq %rdx,%rbx</code>	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	<code>je dest</code>	# Not taken
Cycle 5:	0x01f:	<code>rmmovq %rbx,0(%rdx)</code>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



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

Another Way to Look At the Microarchitecture

Principles:

- Execute each instruction one at a time, one after another
- Express every instruction as series of **simple steps**
- Dedicated hardware structure for completing each step
- Follow same general flow for each instruction type

Fetch: Read instruction from instruction memory

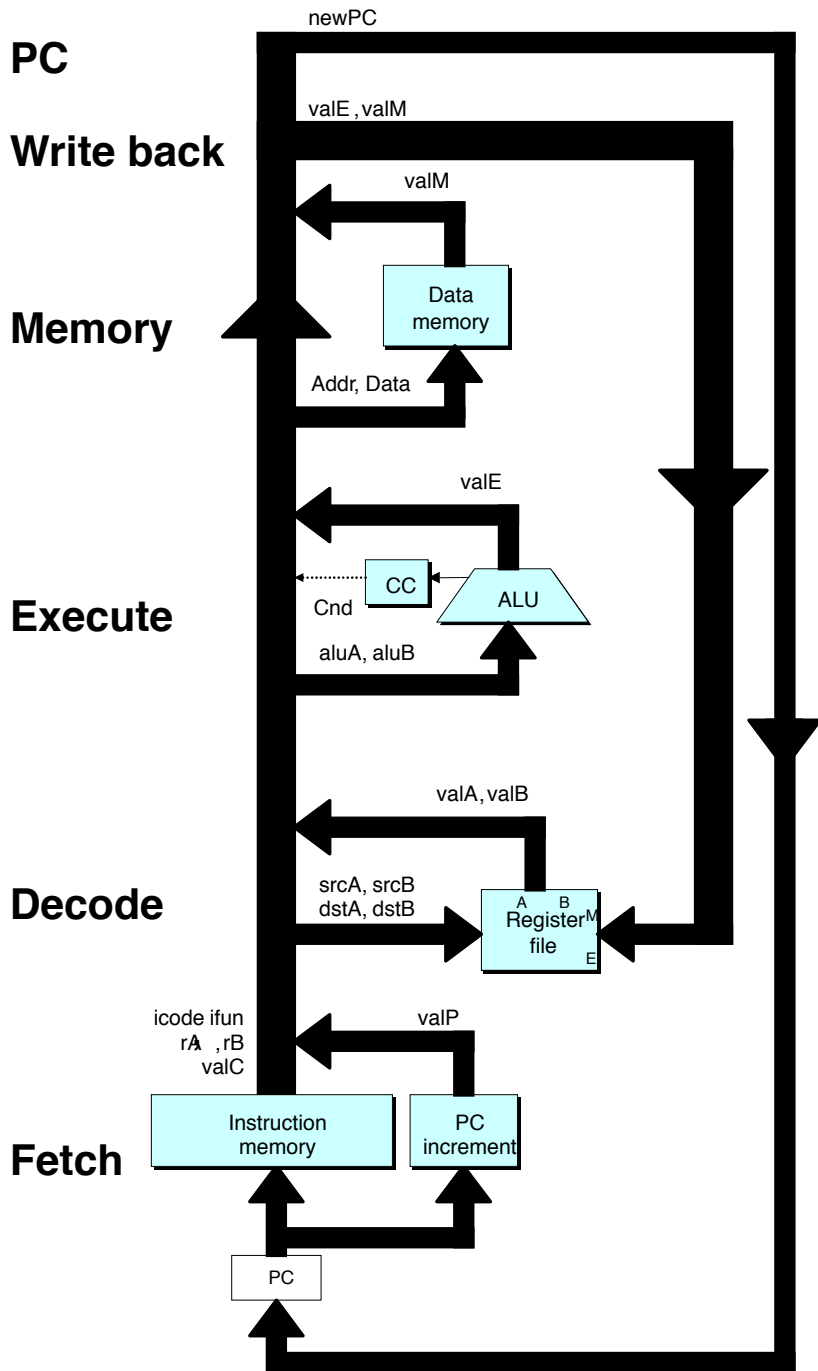
Decode: Read program registers

Execute: Compute value or address

Memory: Read or write data

Write Back: Write program registers

PC: Update program counter



Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

- Read or write data

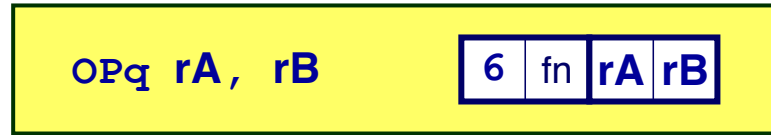
Write Back

- Write program registers

PC

- Update program counter

Stage Computation: Arith/Log. Ops



Stage Computation: Arith/Log. Ops



	$OPq\ rA, rB$
Fetch	$icode:ifun \leftarrow M_1[PC]$
	$rA:rB \leftarrow M_1[PC+1]$
	$valP \leftarrow PC+2$

Read instruction byte

Read register byte

Compute next PC

Stage Computation: Arith/Log. Ops



	$OPq\ rA,\ rB$	
Fetch	$icode:ifun \leftarrow M_1[PC]$	Read instruction byte
	$rA:rB \leftarrow M_1[PC+1]$	Read register byte
	$valP \leftarrow PC+2$	Compute next PC
Decode	$valA \leftarrow R[rA]$	Read operand A
	$valB \leftarrow R[rB]$	Read operand B

Stage Computation: Arith/Log. Ops



	$OPq\ rA,\ rB$	
Fetch	$icode:ifun \leftarrow M_1[PC]$	Read instruction byte
	$rA:rB \leftarrow M_1[PC+1]$	Read register byte
	$valP \leftarrow PC+2$	Compute next PC
Decode	$valA \leftarrow R[rA]$	Read operand A
	$valB \leftarrow R[rB]$	Read operand B
Execute	$valE \leftarrow valB\ OP\ valA$	Perform ALU operation
	Set CC	Set condition code register

Stage Computation: Arith/Log. Ops



	$OPq\ rA, rB$	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$	Read instruction byte Read register byte
	$valP \leftarrow PC+2$	Compute next PC
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	Read operand A Read operand B
	$valE \leftarrow valB\ OP\ valA$ Set CC	Perform ALU operation Set condition code register
Memory		

Stage Computation: Arith/Log. Ops



	OPq rA, rB	
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$	Read instruction byte Read register byte
	valP $\leftarrow PC+2$	Compute next PC
Decode	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	Read operand A Read operand B
	valE $\leftarrow valB \text{ OP } valA$ Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] $\leftarrow valE$	Write back result

Stage Computation: Arith/Log. Ops



	OPq rA, rB	
Fetch	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$	Read instruction byte Read register byte
	valP $\leftarrow PC+2$	Compute next PC
Decode	valA $\leftarrow R[rA]$ valB $\leftarrow R[rB]$	Read operand A Read operand B
	valE $\leftarrow valB \text{ OP } valA$ Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] $\leftarrow valE$	Write back result
PC update	PC $\leftarrow valP$	Update PC

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`

4

0

rA

rB

D

`rmmovq rA, D(rB)`

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



	<code>rmmovq rA, D(rB)</code>
Fetch	<code>icode:ifun ← M₁[PC]</code> <code>rA:rB ← M₁[PC+1]</code> <code>valC ← M₈[PC+2]</code> <code>valP ← PC+10</code>

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



	<code>rmmovq rA, D(rB)</code>
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



<code>rmmovq rA, D(rB)</code>	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB + valC$

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B
- Compute effective address

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



	<code>rmmovq rA, D(rB)</code>
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB + valC$
Memory	$M_8[valE] \leftarrow valA$

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B
- Compute effective address
- Write value to memory

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



	<code>rmmovq rA, D(rB)</code>
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB + valC$
Memory	$M_8[valE] \leftarrow valA$
Write back	

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B
- Compute effective address
- Write value to memory

Stage Computation: `rmmovq`

`rmmovq rA, D(rB)`



<code>rmmovq rA, D(rB)</code>	
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB + valC$
Memory	$M_8[valE] \leftarrow valA$
Write back	
PC update	$PC \leftarrow valP$

- Read instruction byte
- Read register byte
- Read displacement D
- Compute next PC
- Read operand A
- Read operand B
- Compute effective address
- Write value to memory
- Update PC

Stage Computation: Jumps

jXX Dest

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Stage Computation: Jumps

	jXX Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Read instruction byte
	$\text{valC} \leftarrow M_8[\text{PC}+1]$	Read destination address
	$\text{valP} \leftarrow \text{PC}+9$	Fall through address

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Stage Computation: Jumps

	jXX Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Read instruction byte
	$\text{valC} \leftarrow M_8[\text{PC}+1]$	Read destination address
	$\text{valP} \leftarrow \text{PC}+9$	Fall through address
Decode		

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Stage Computation: Jumps

	jXX Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Read instruction byte
	$\text{valC} \leftarrow M_8[\text{PC}+1]$	Read destination address
	$\text{valP} \leftarrow \text{PC}+9$	Fall through address
Decode		
Execute	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	Take branch?

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Stage Computation: Jumps

	jXX Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Read instruction byte
	$\text{valC} \leftarrow M_8[\text{PC}+1]$	Read destination address
	$\text{valP} \leftarrow \text{PC}+9$	Fall through address
Decode		
Execute	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	Take branch?
Memory		

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Stage Computation: Jumps

	jXX Dest	
Fetch	$icode:ifun \leftarrow M_1[PC]$	Read instruction byte
	$valC \leftarrow M_8[PC+1]$	Read destination address
	$valP \leftarrow PC+9$	Fall through address
Decode		
Execute	$Cnd \leftarrow Cond(CC,ifun)$	Take branch?
Memory		
Write back		

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Stage Computation: Jumps

	jXX Dest	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Read instruction byte
	$\text{valC} \leftarrow M_8[\text{PC}+1]$	Read destination address
	$\text{valP} \leftarrow \text{PC}+9$	Fall through address
Decode		
Execute	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	Take branch?
Memory		
Write back		
PC update	$\text{PC} \leftarrow \text{Cnd} ? \text{valC} : \text{valP}$	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Processor Microarchitecture

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

Real-World Pipelines: Car Washes

Real-World Pipelines: Car Washes

Sequential



Real-World Pipelines: Car Washes

Sequential



Pipelined



Real-World Pipelines: Car Washes

Sequential



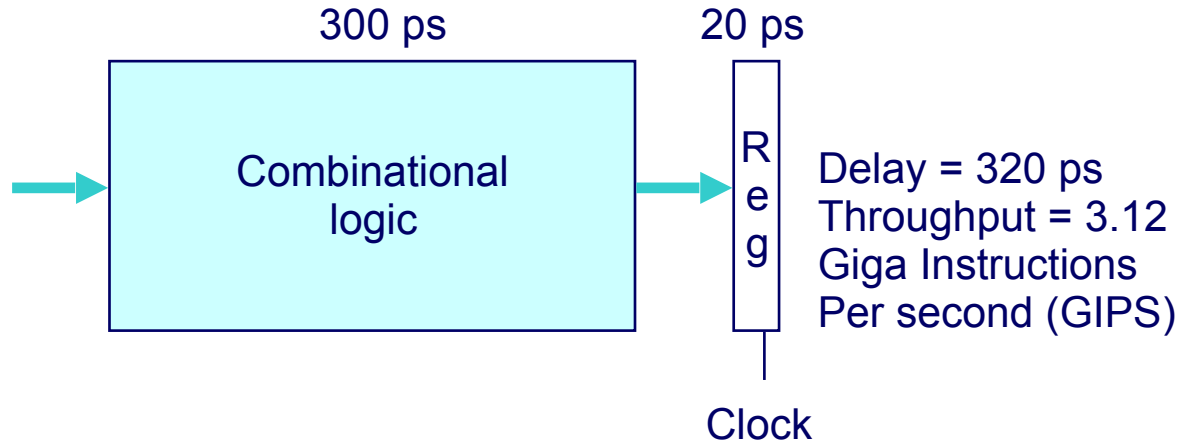
Pipelined



Idea

- Divide process into independent stages
- Move objects through stages in sequence
- At any given times, multiple objects being processed

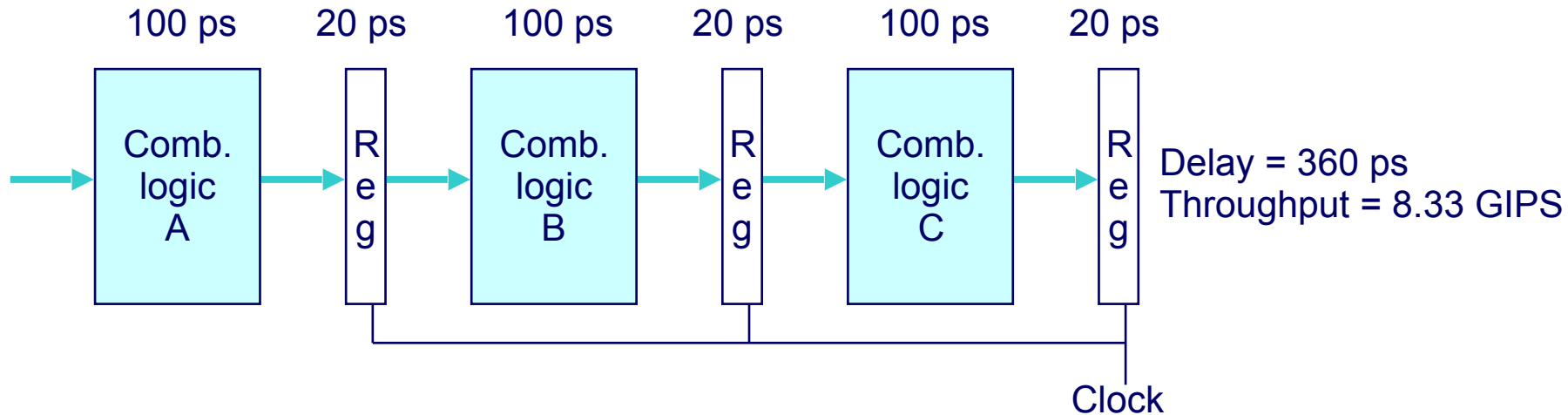
Computational Example



System

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

3-Stage Pipelined Version

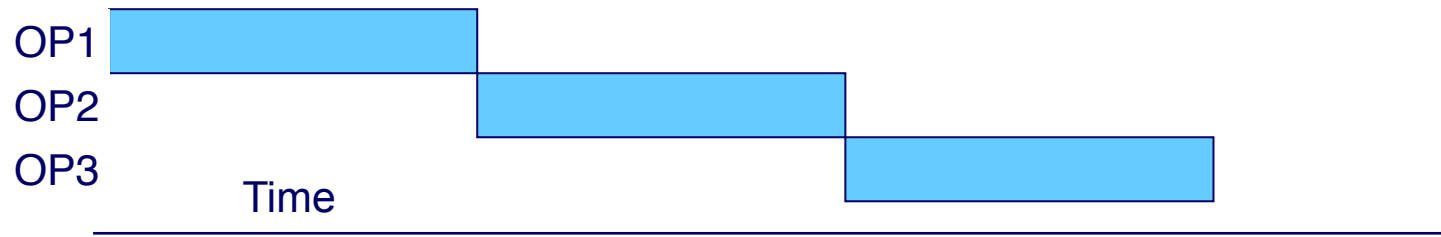


System

- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
 - Begin new operation every 120 ps
- Overall latency increases
 - 360 ps from start to finish

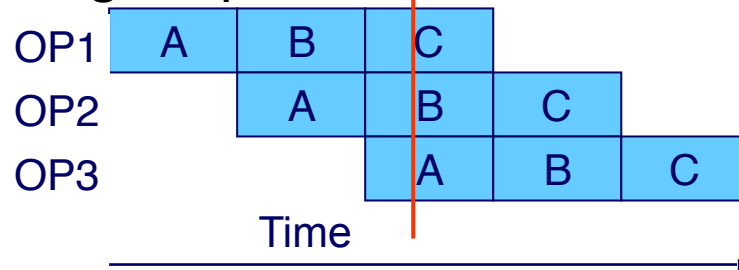
Pipeline Diagrams

Unpipelined



- Cannot start new operation until previous one completes

3-Stage Pipelined



- Up to 3 operations in process simultaneously