CSC 252: Computer Organization
Fall 2021: Lecture 12

Processor Architecture:
Circuits
Single-cycle Implementation

Instructor: Alan Beadle

Department of Computer Science
University of Rochester

Announcements

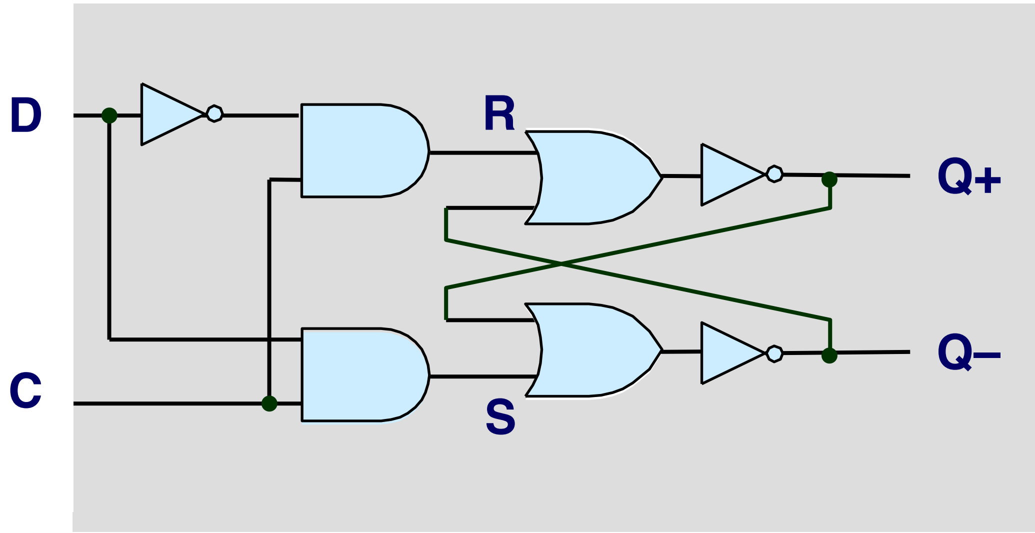
Assignment 3 executable is 32-bit in order to support the modifications to modern Linux stack management

As a result, if you use gcc to generate code for this assignment, you will need to use “-m32” flag

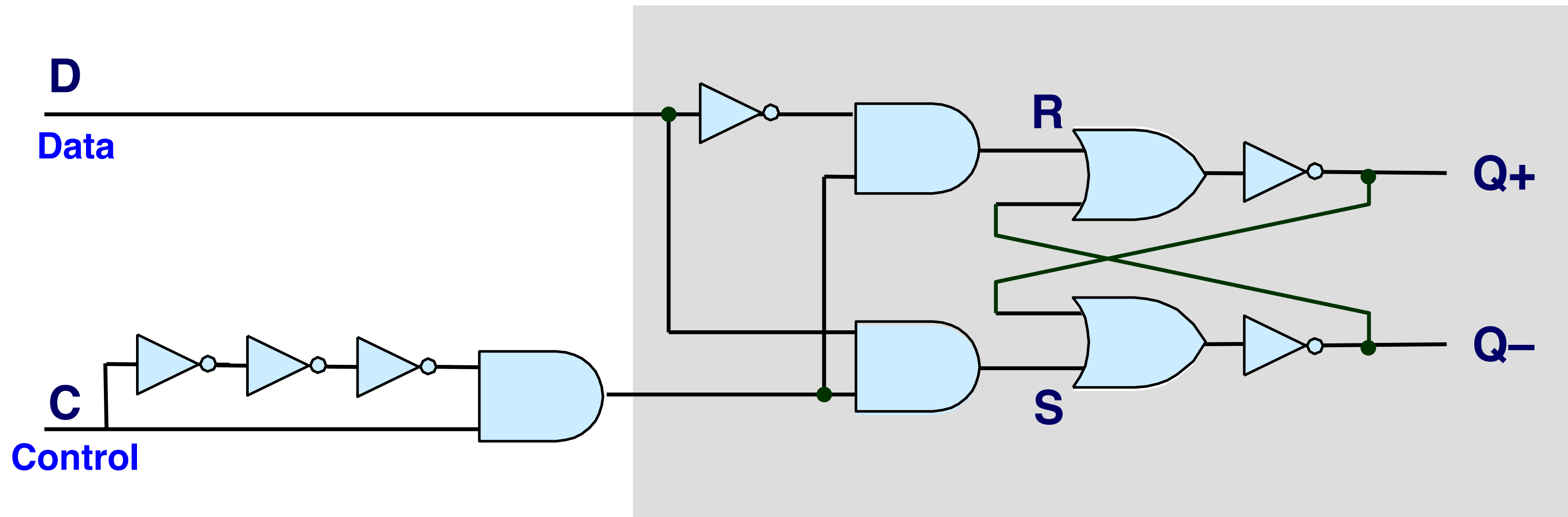
You have about 2 weeks, but the midterm is right after, so make sure you have time for both

No class Monday: Fall break

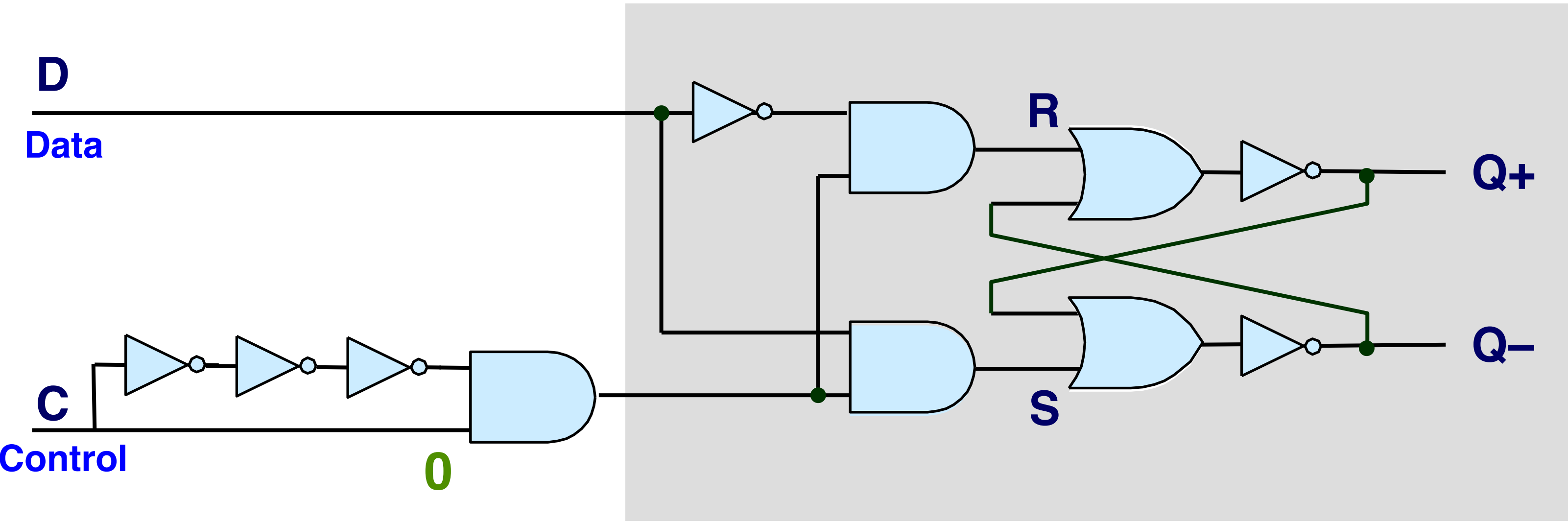
Edge-Triggered Latch (Flip-Flop)



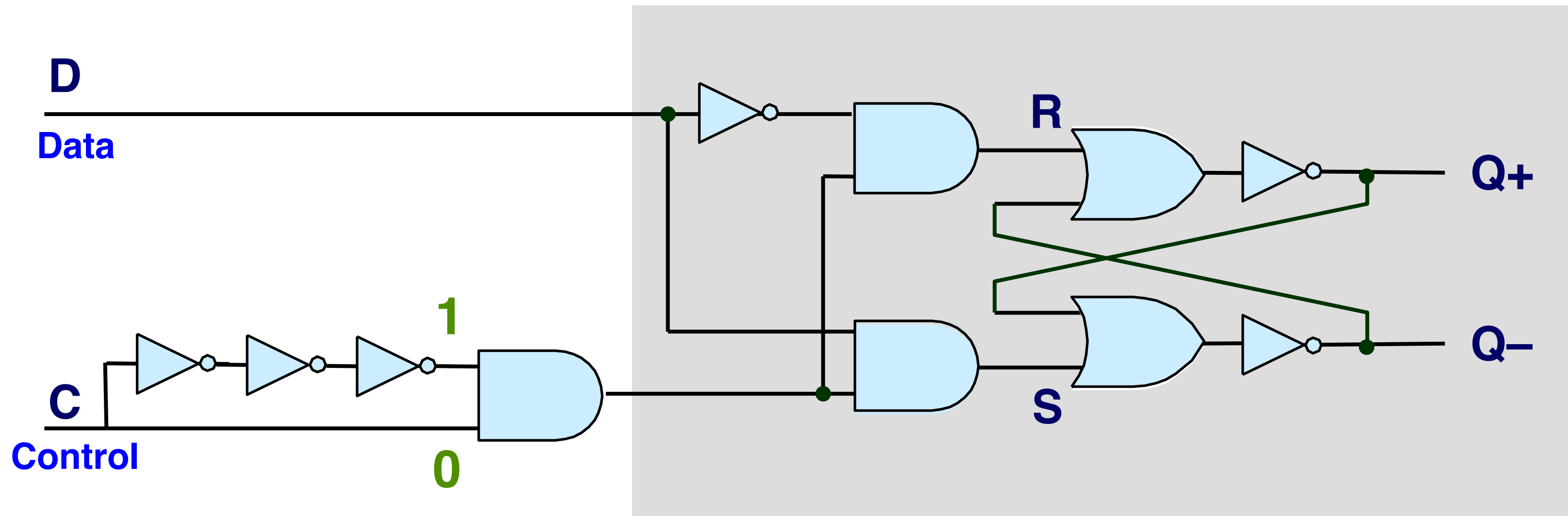
Edge-Triggered Latch (Flip-Flop)



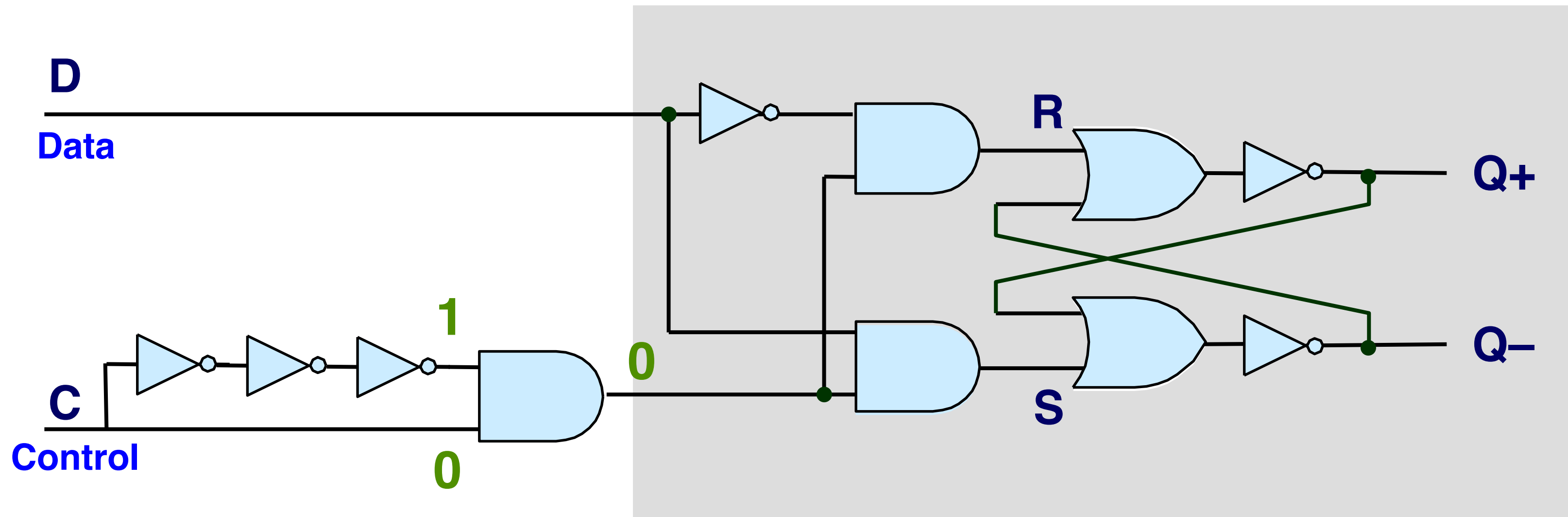
Edge-Triggered Latch (Flip-Flop)



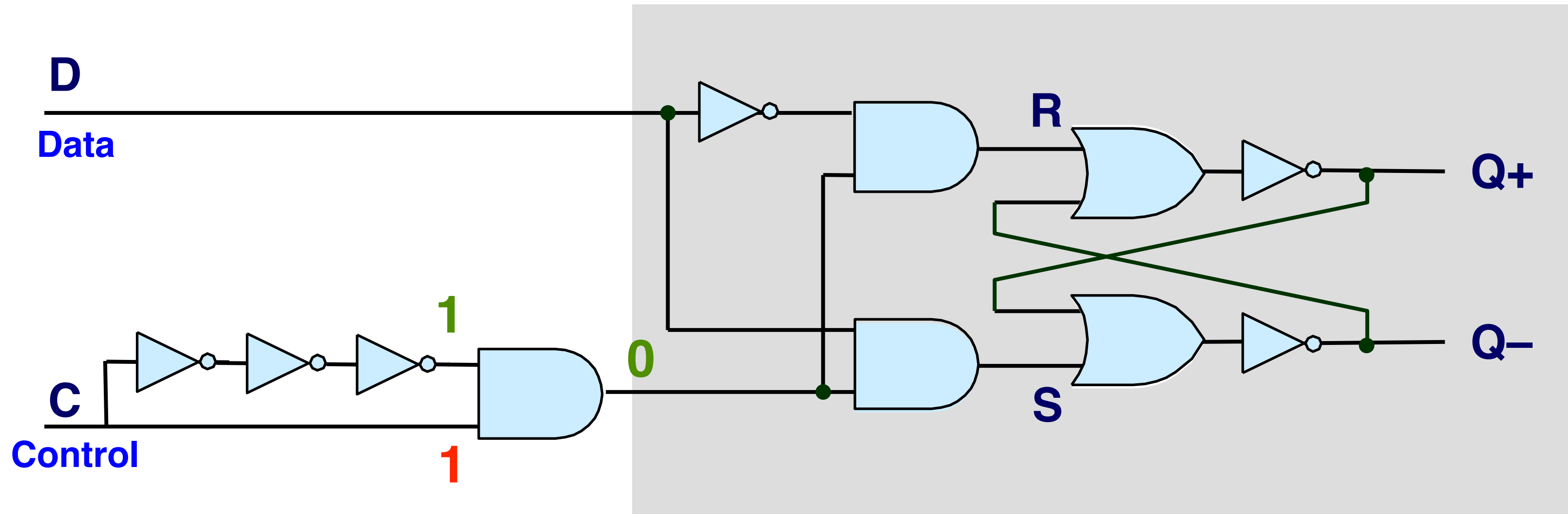
Edge-Triggered Latch (Flip-Flop)



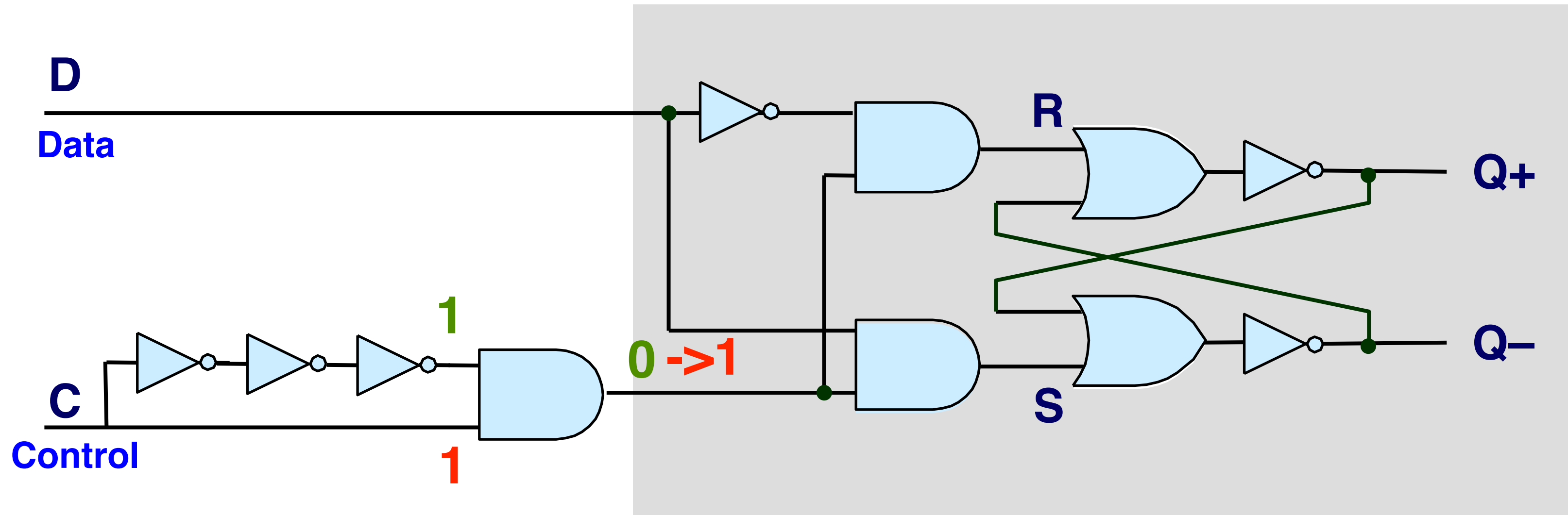
Edge-Triggered Latch (Flip-Flop)



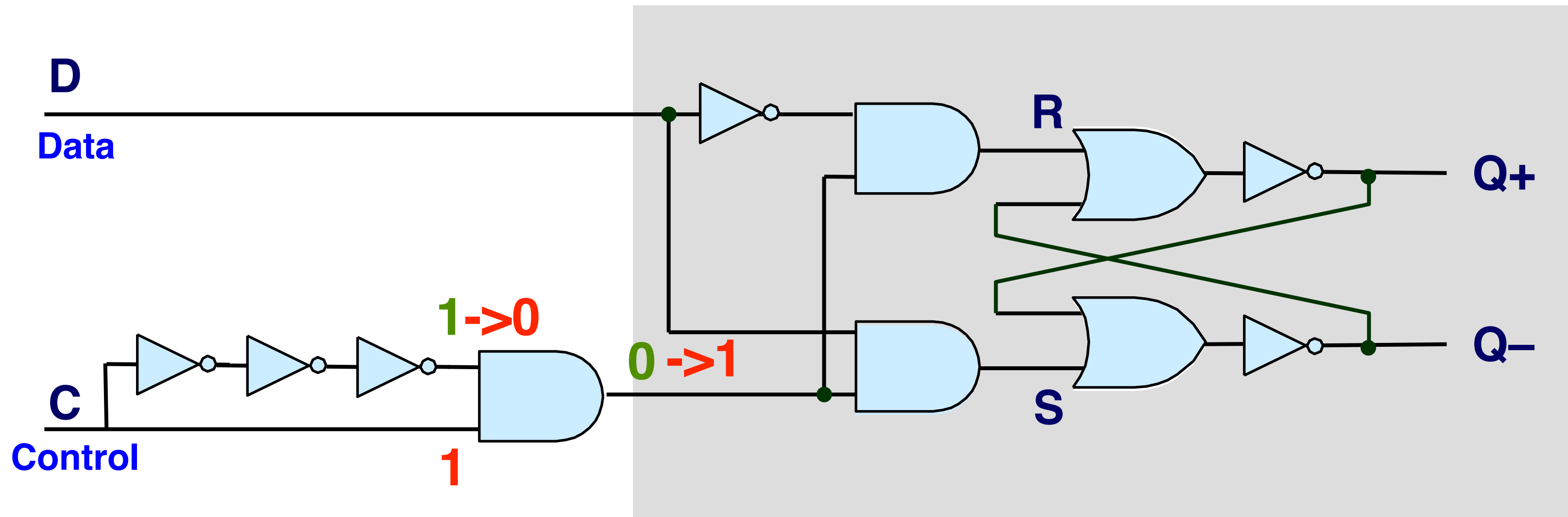
Edge-Triggered Latch (Flip-Flop)



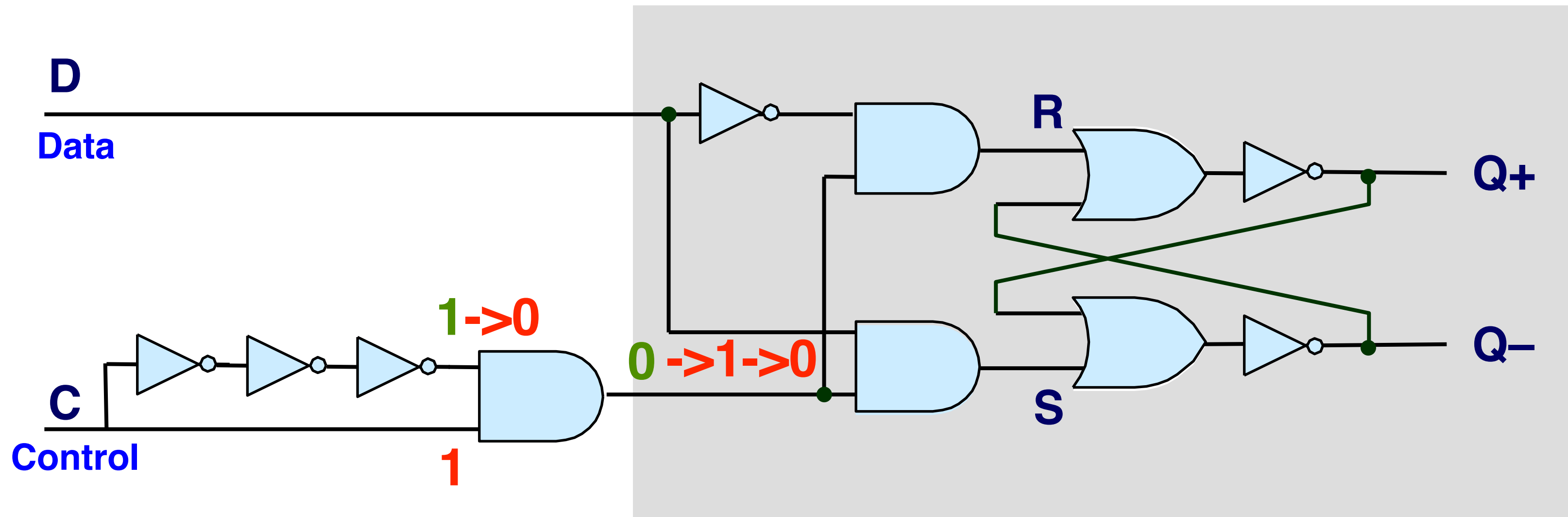
Edge-Triggered Latch (Flip-Flop)



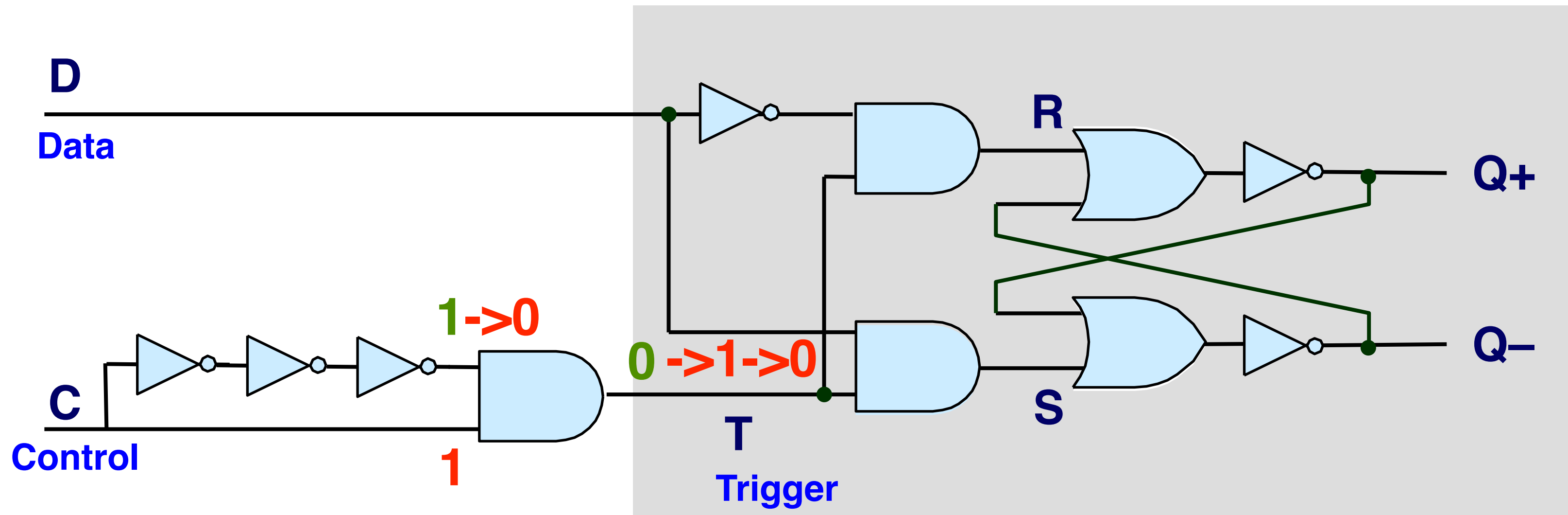
Edge-Triggered Latch (Flip-Flop)

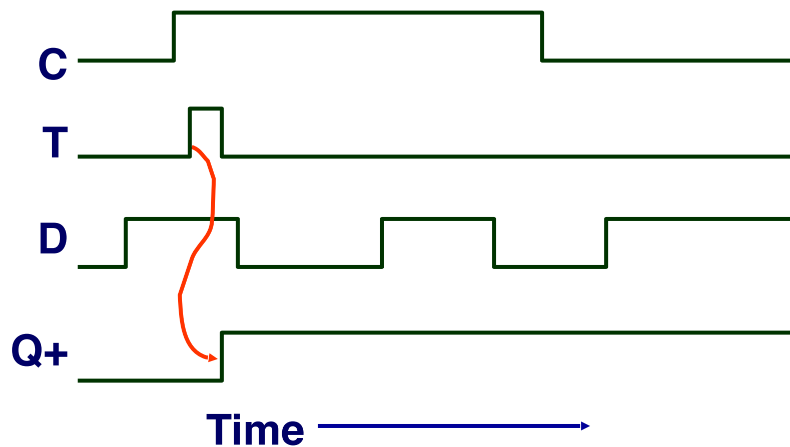


Edge-Triggered Latch (Flip-Flop)

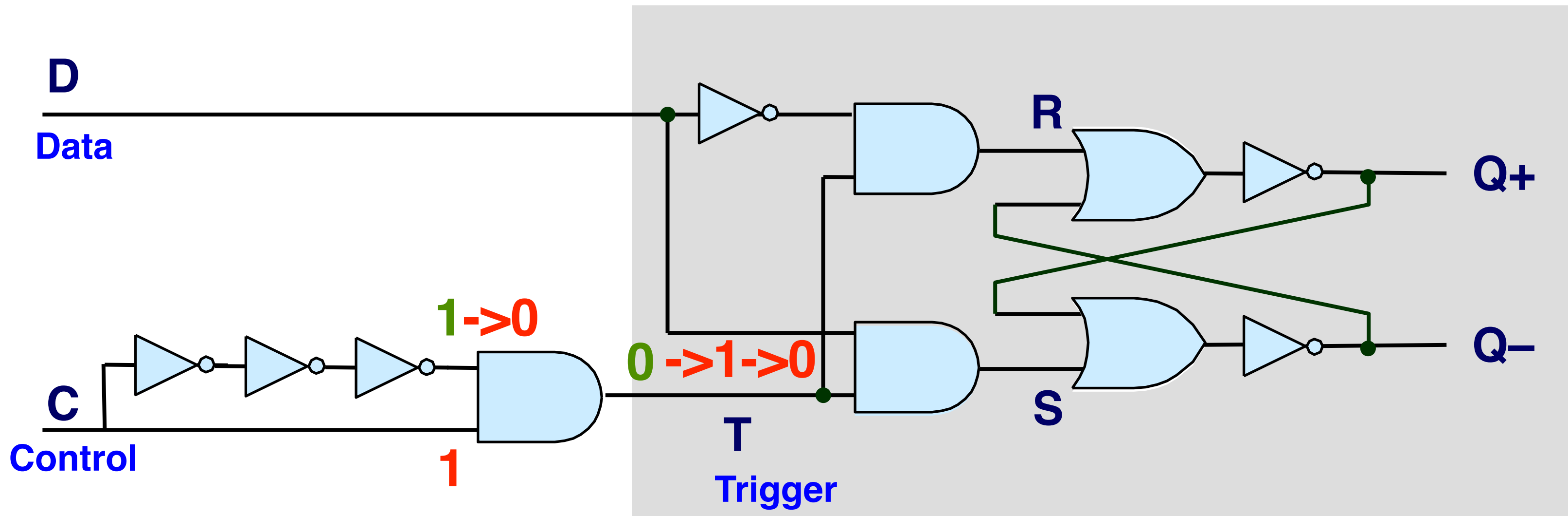


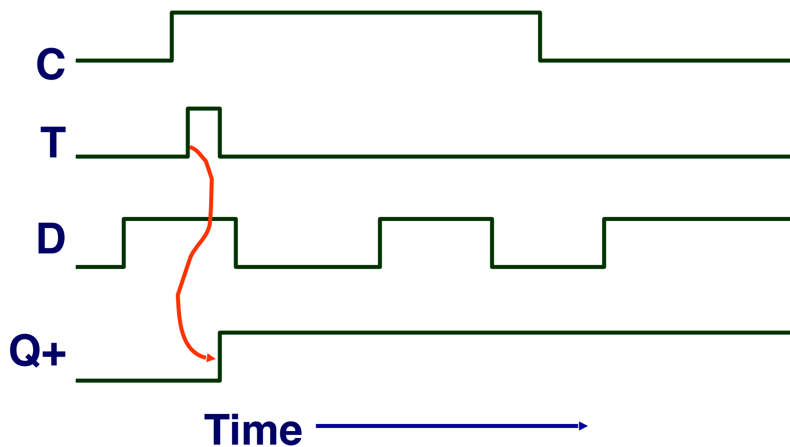
Edge-Triggered Latch (Flip-Flop)





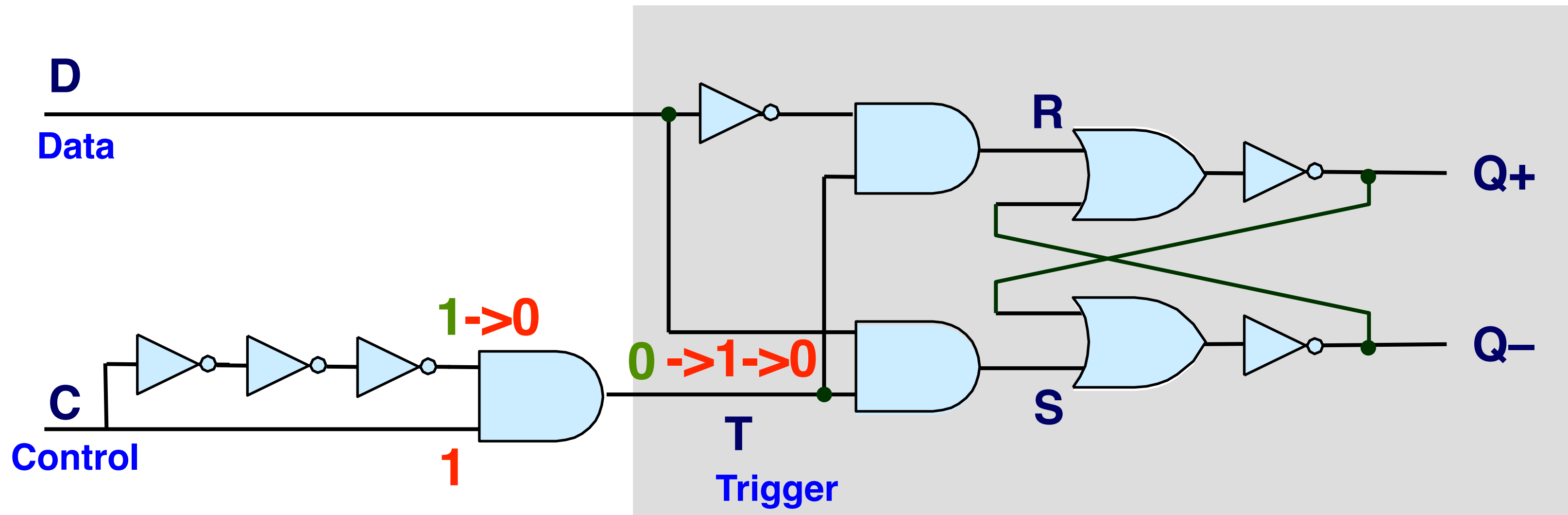
Edge-Triggered Latch (Flip-Flop)

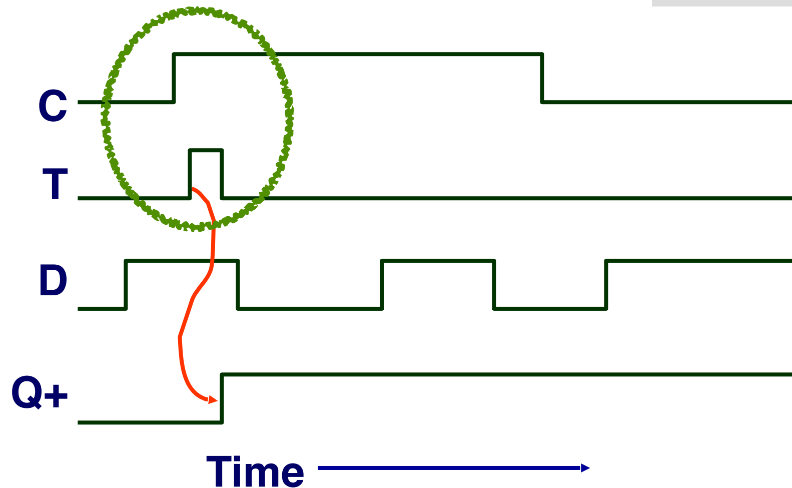




- Flip-flop: Only latches data for a brief period

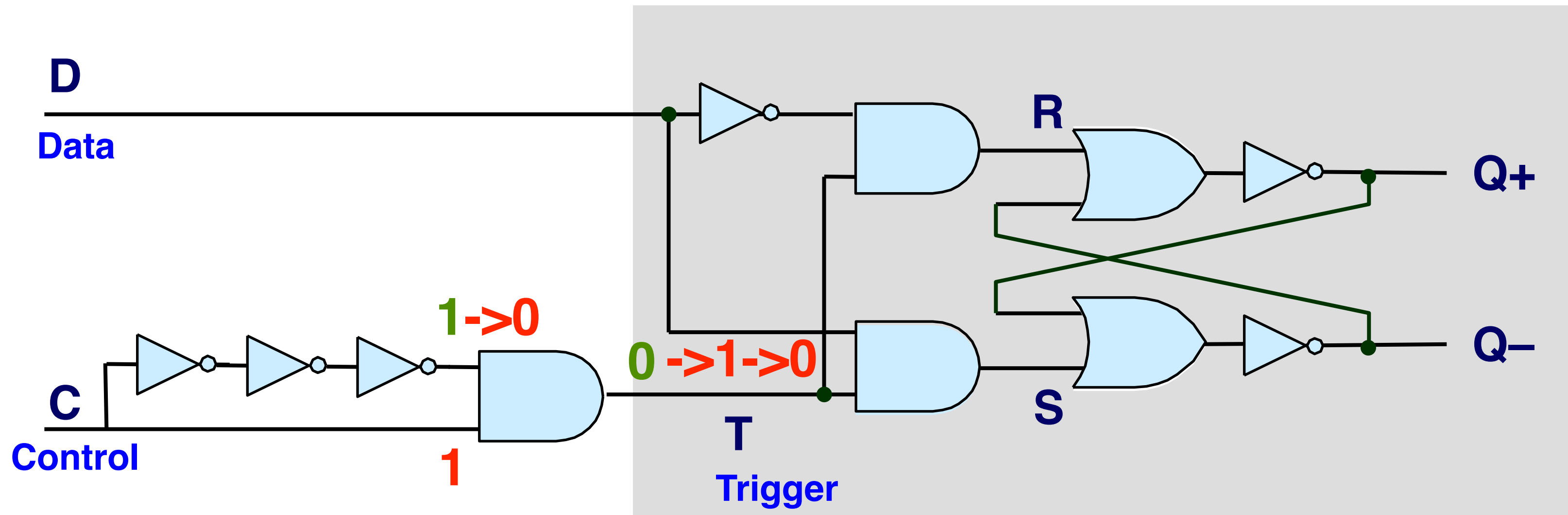
Edge-Triggered Latch (Flip-Flop)

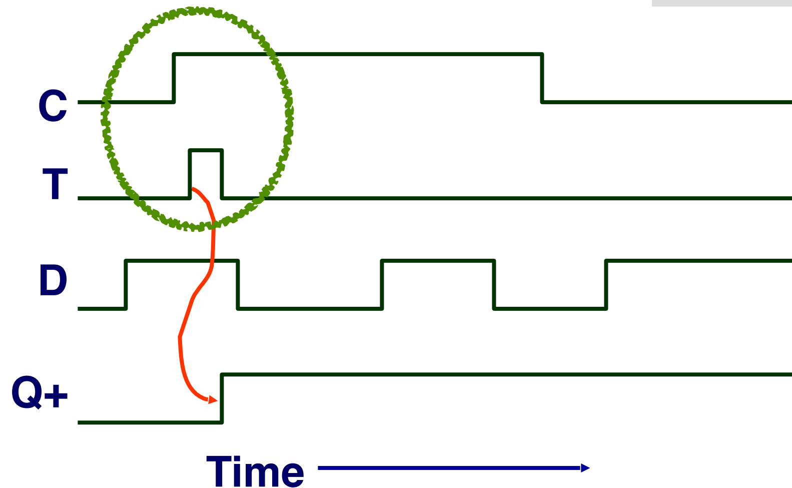




- Flip-flop: Only latches data for a brief period
- Value latched depends on data as C **rises** (i.e., 0–>1); usually called at the **rising edge** of C

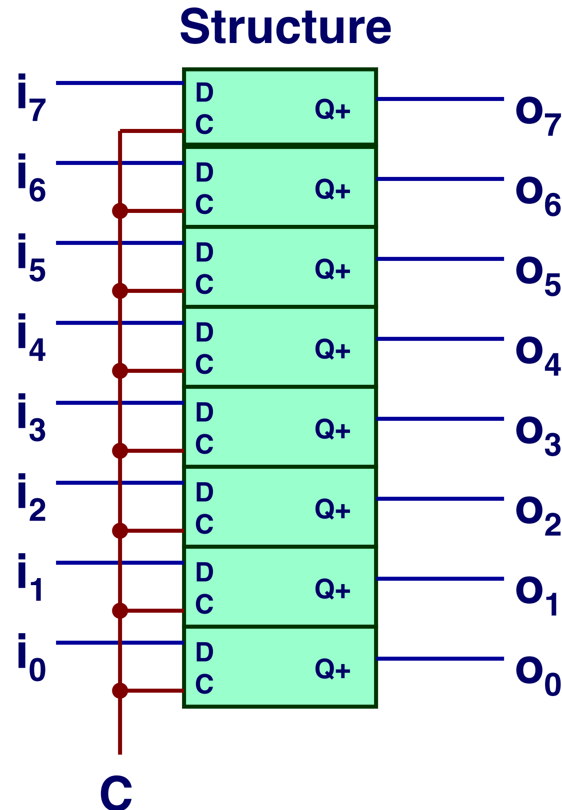
Edge-Triggered Latch (Flip-Flop)





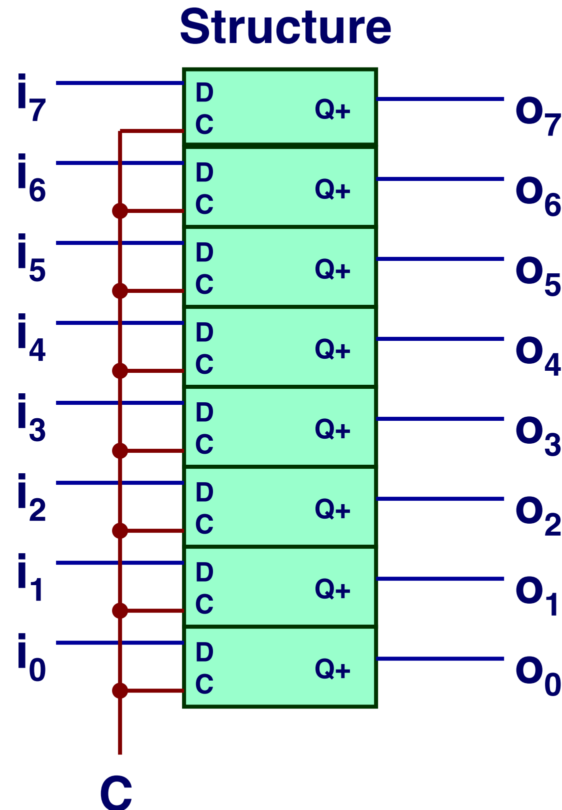
- Flip-flop: Only latches data for a brief period
- Value latched depends on data as C **rises** (i.e., 0–>1); usually called at the **rising edge** of C
- Output remains stable at all other times

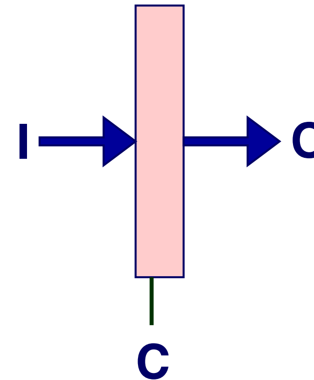
Registers



- Stores several bits of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of the C signal

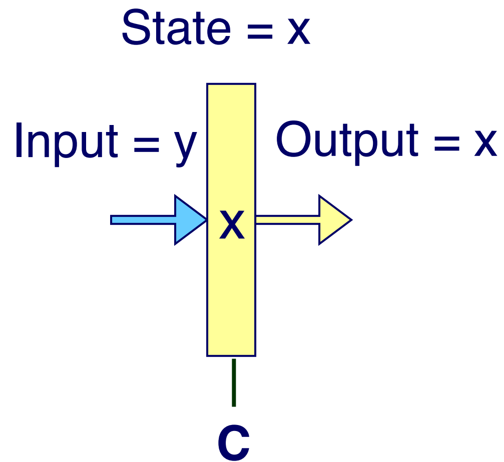
Registers



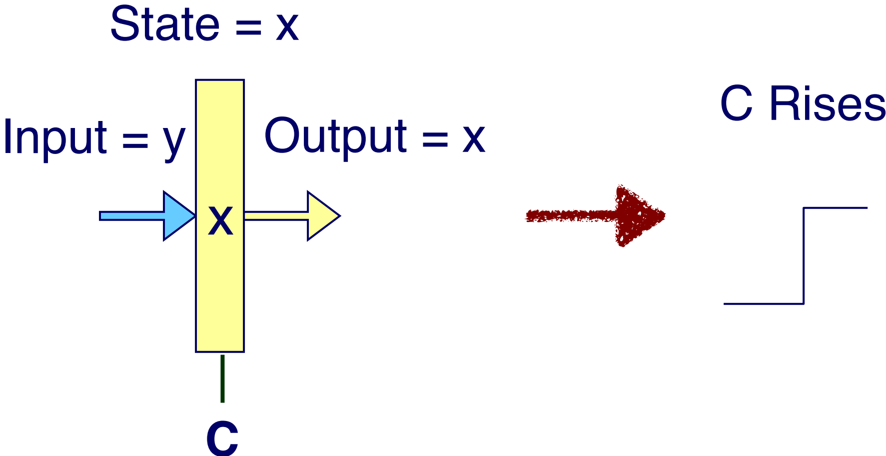


- Stores several bits of data
- Collection of edge-triggered latches (D Flip-flops)
- Loads input on rising edge of the C signal

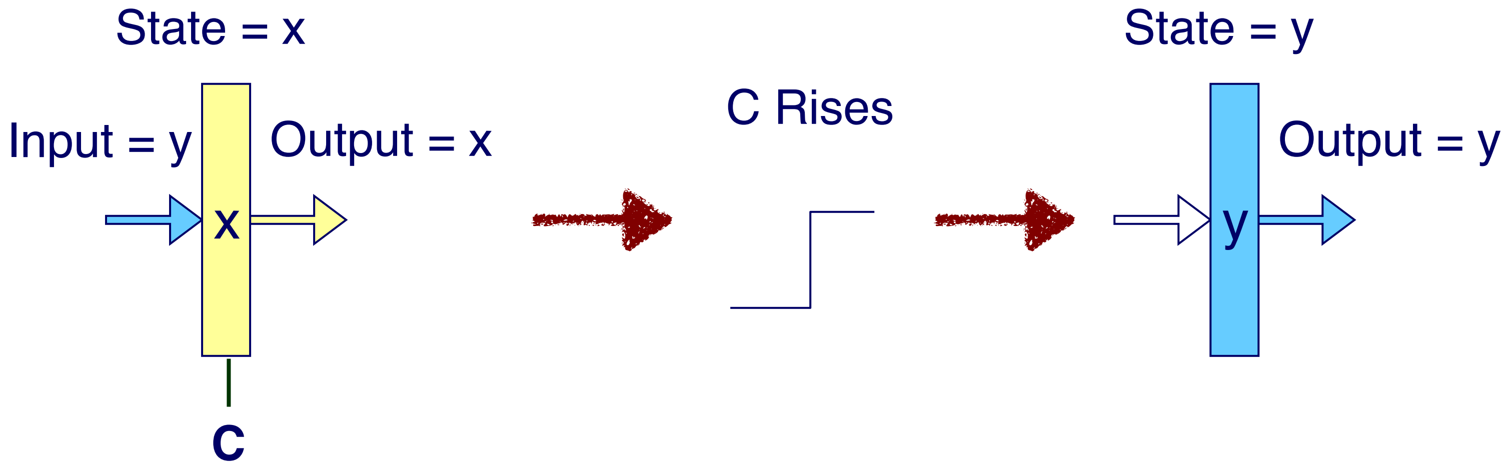
Register Operation



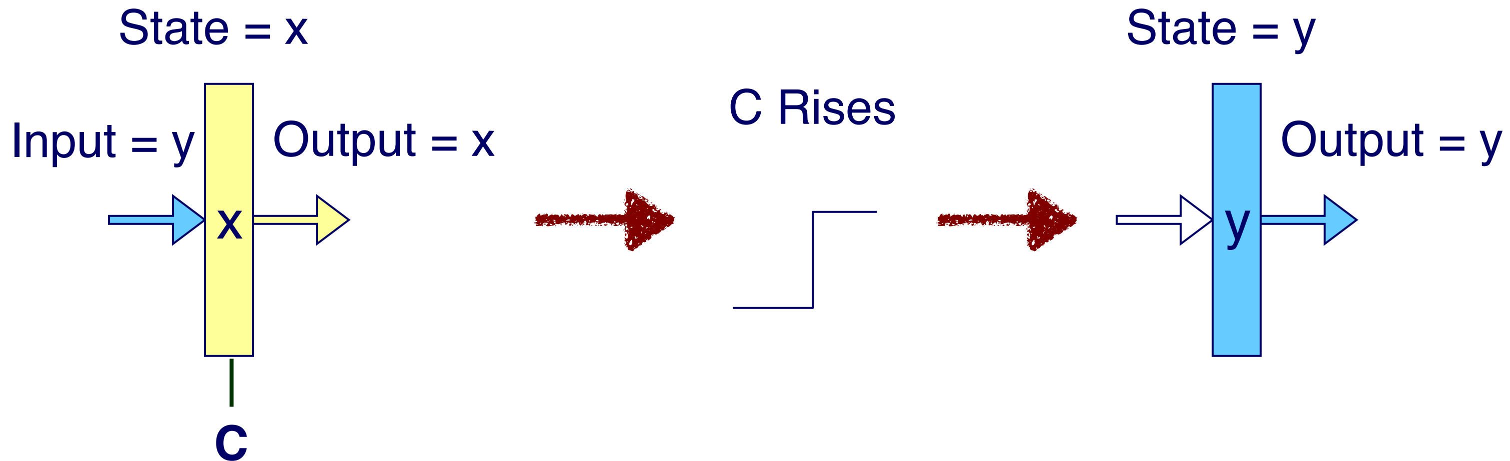
Register Operation



Register Operation

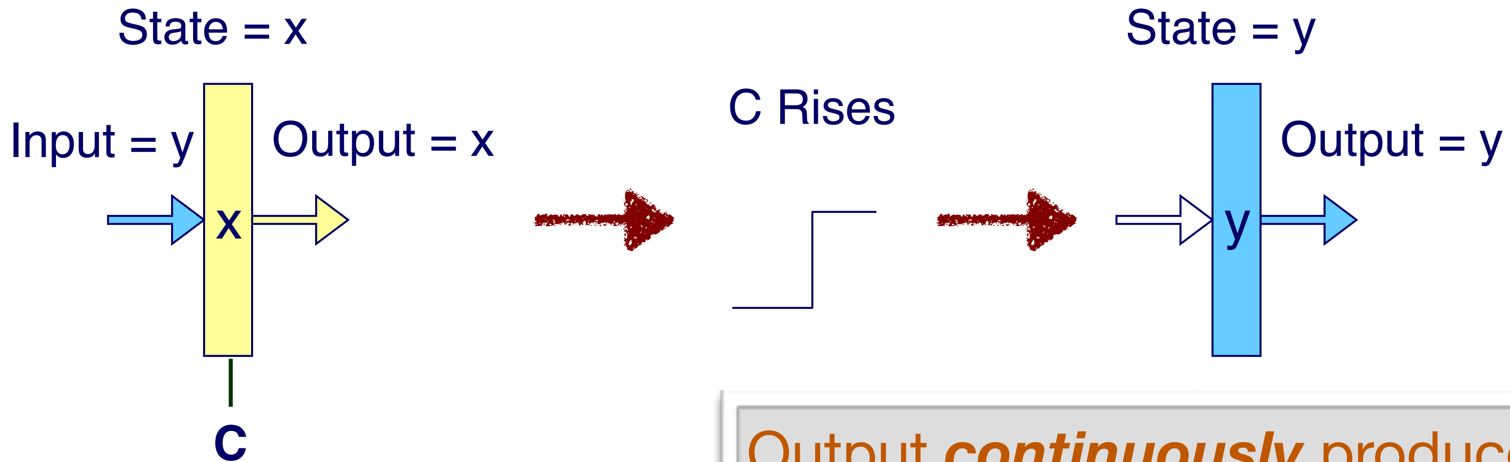


Register Operation



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

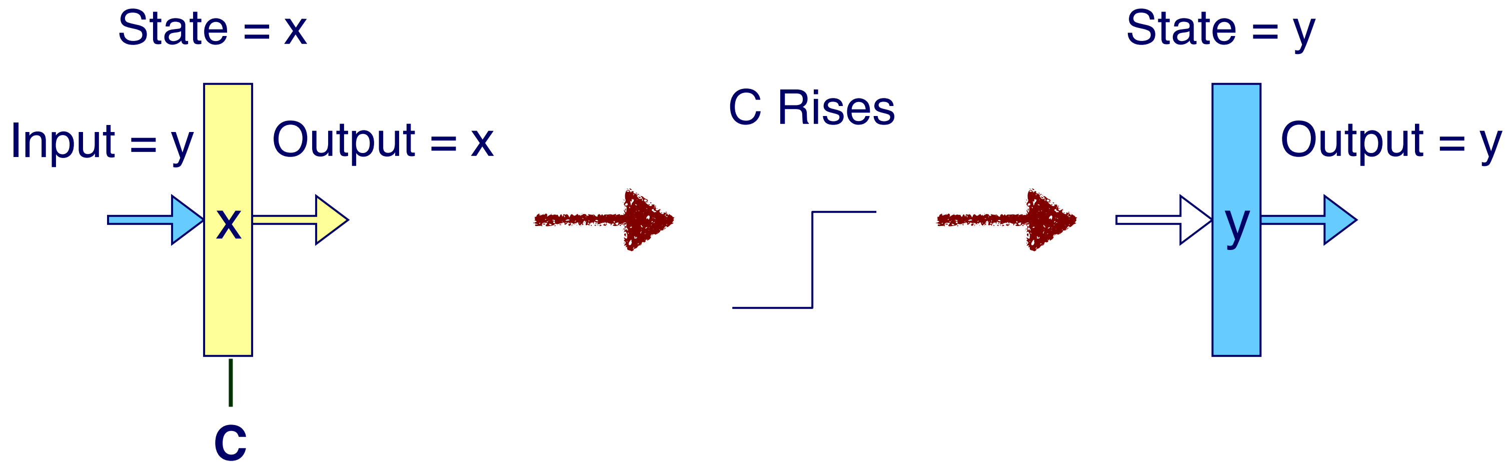
Register Operation



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

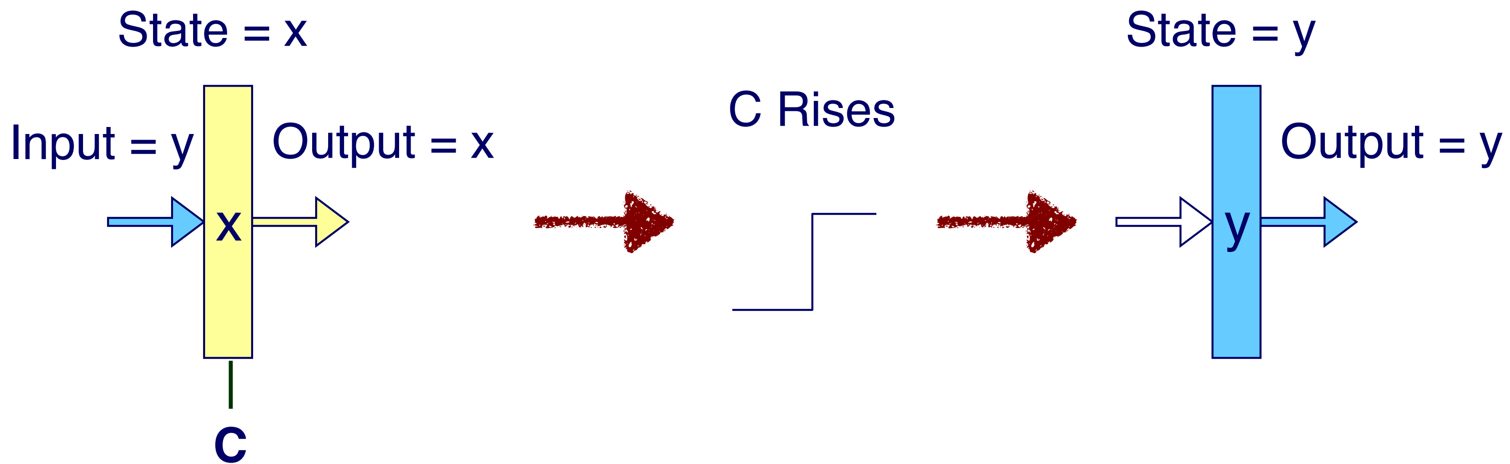
- Stores data bits
- For most of time acts as barrier between input and output
- As C rises, loads input
- So you'd better compute the input before the C signal rises if you want to store the input data to the register

Clock Signal



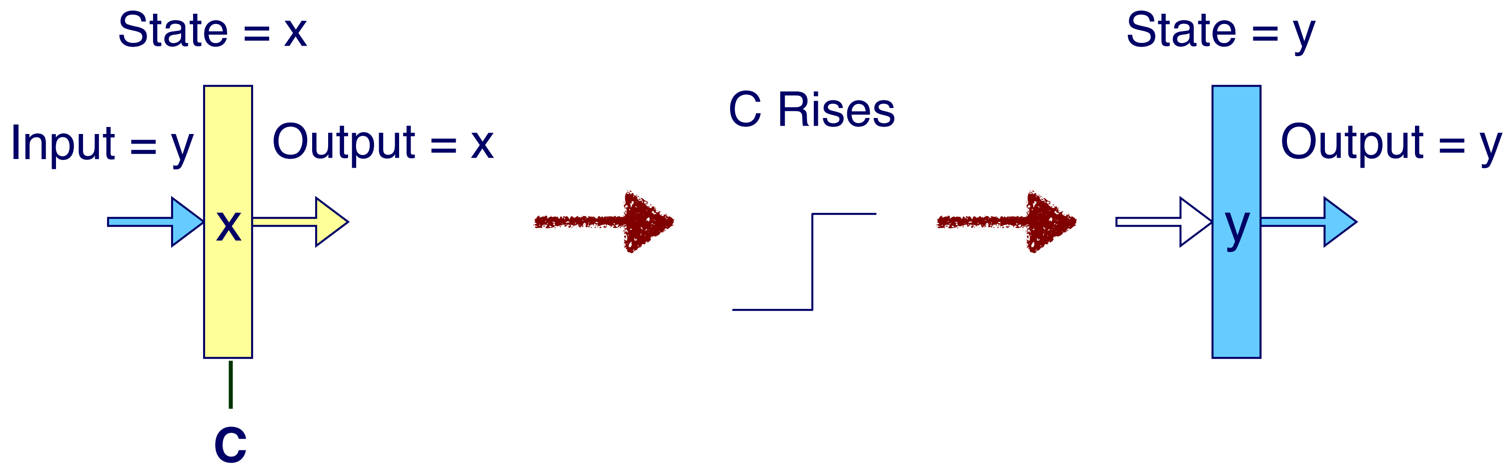
- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.

Clock Signal

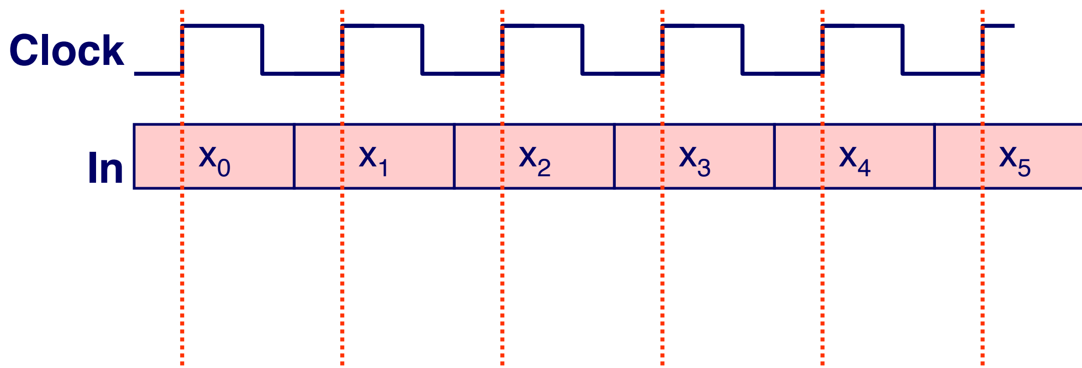


- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.

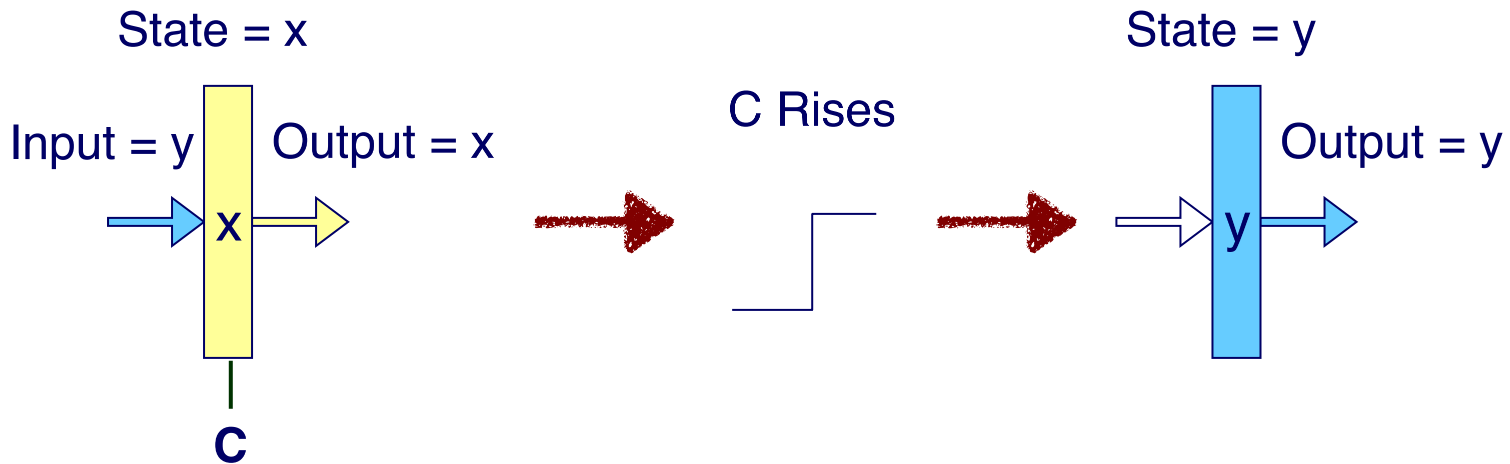
Clock Signal



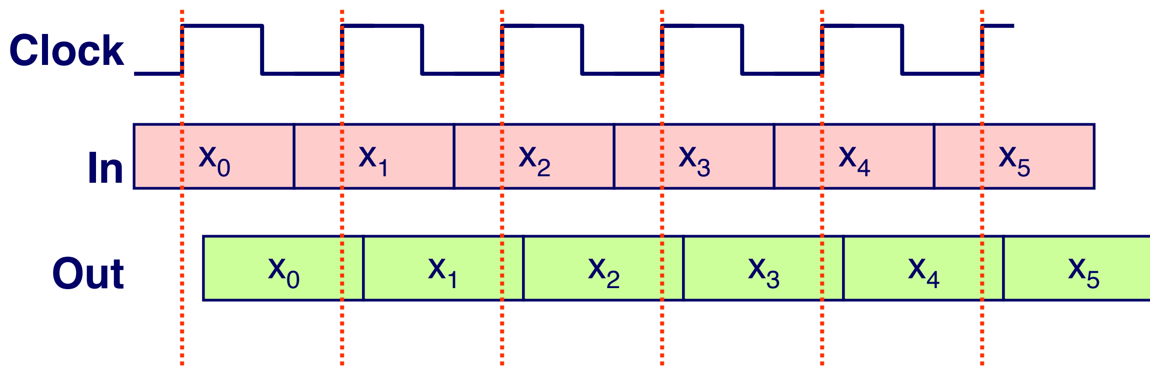
- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.



Clock Signal

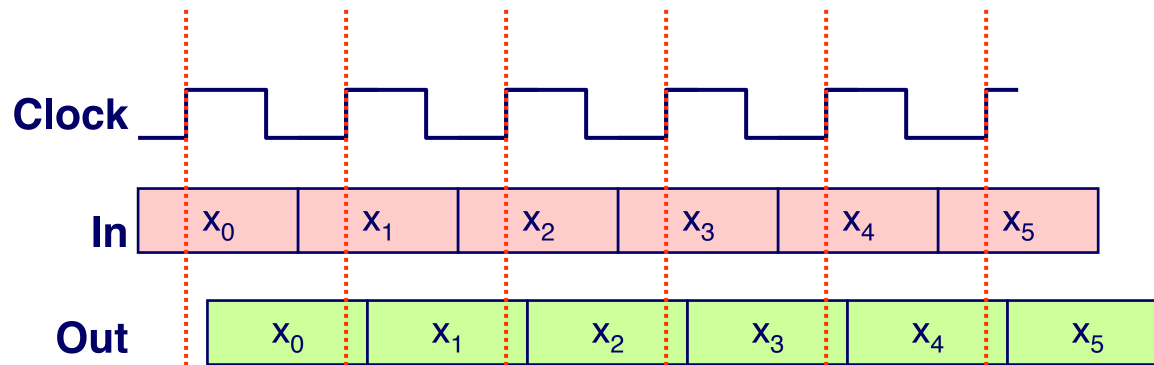


- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer.



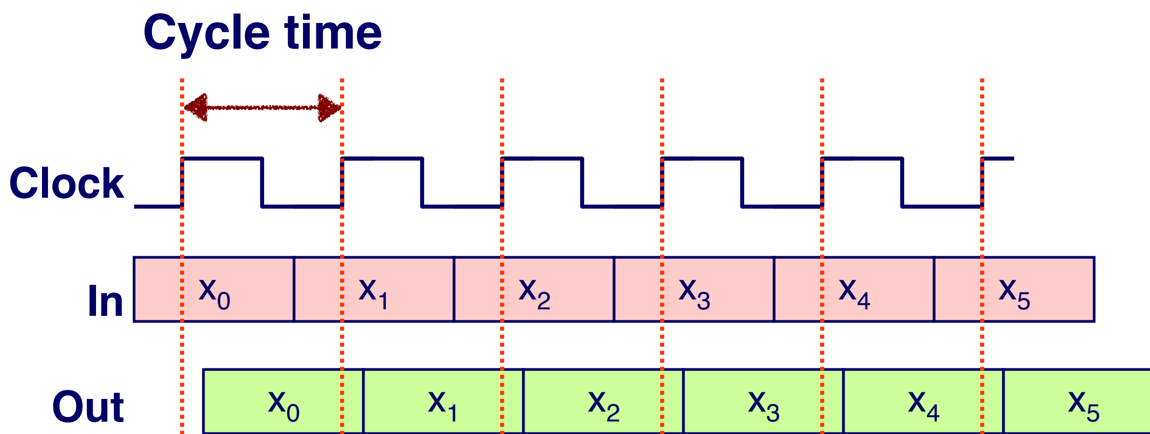
Clock Signal

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



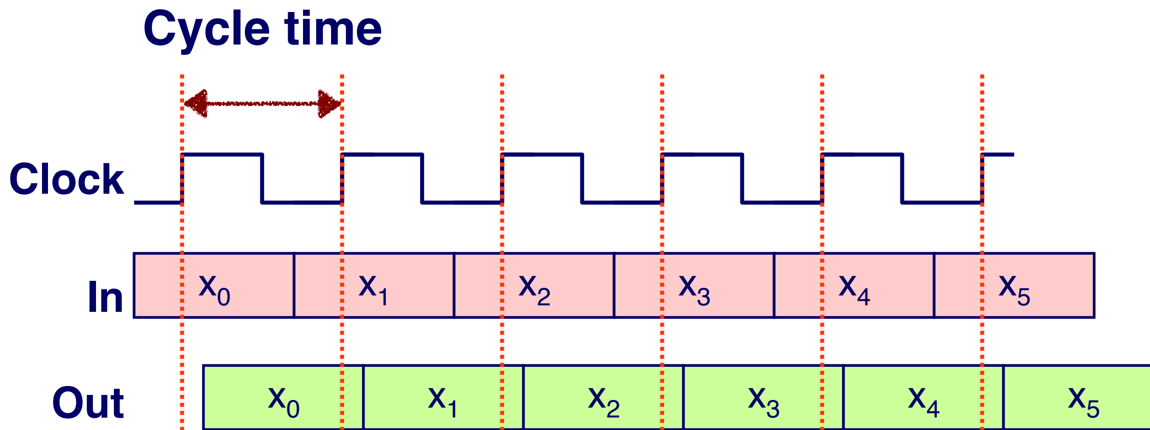
Clock Signal

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



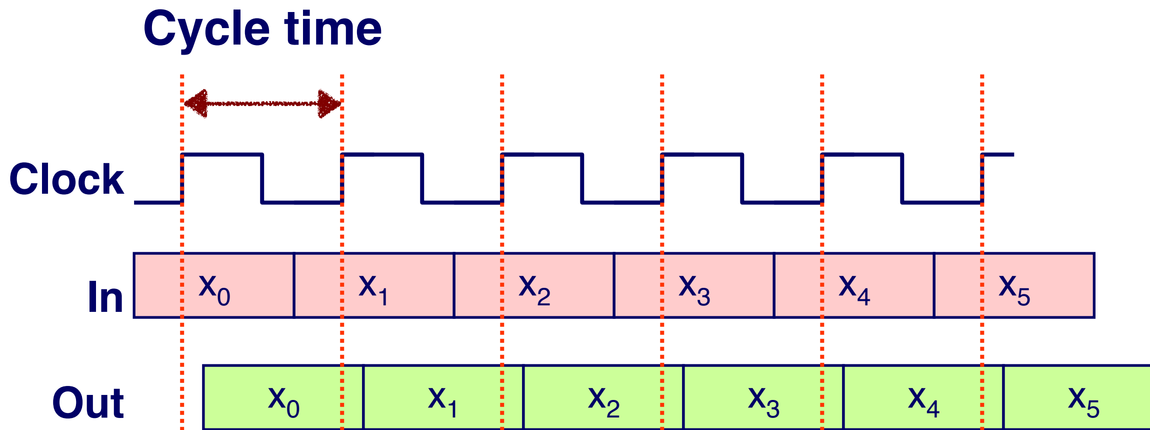
Clock Signal

- 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.



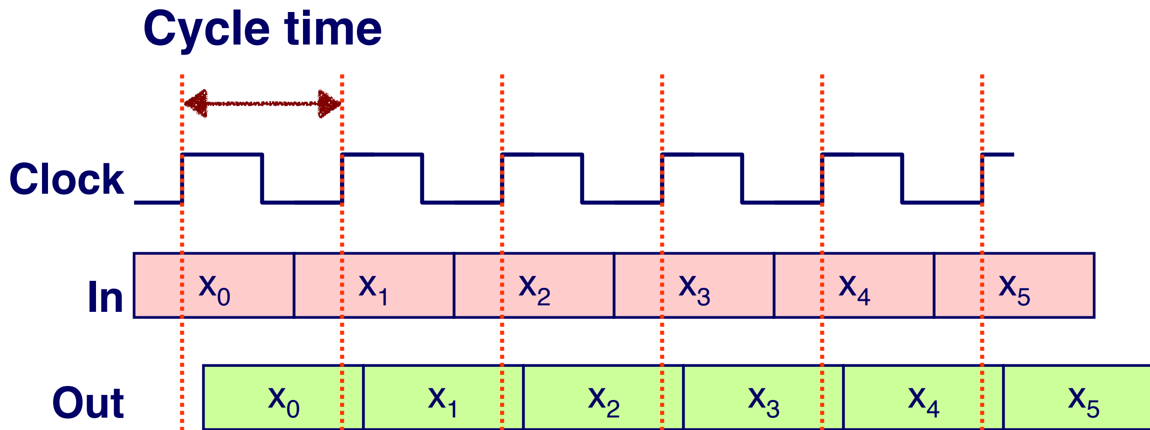
Clock Signal

- 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



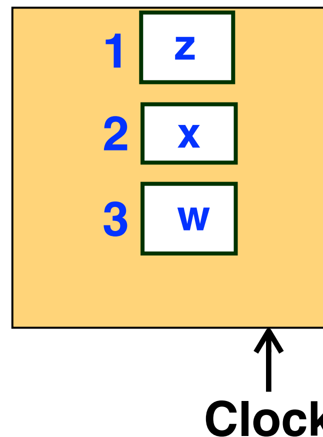
Clock Signal

- 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



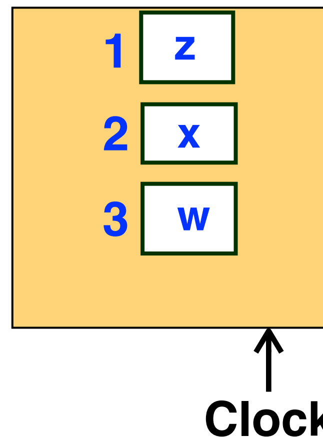
Register File

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



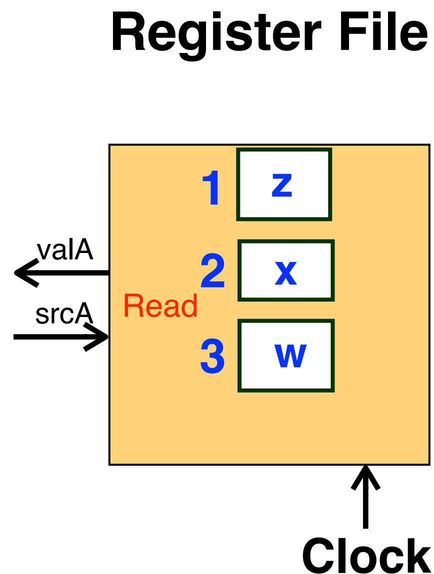
Register File

- 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



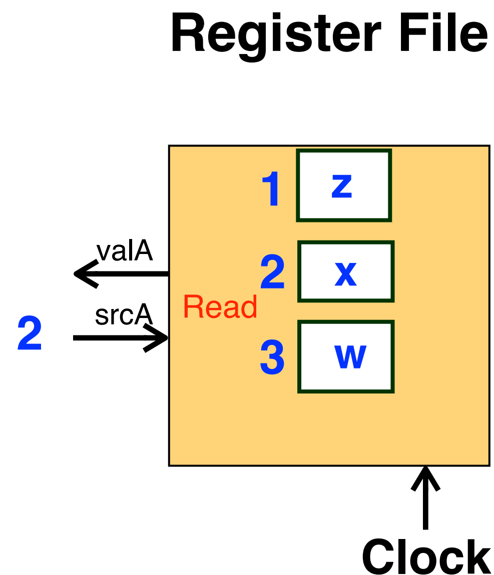
Register File

- 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



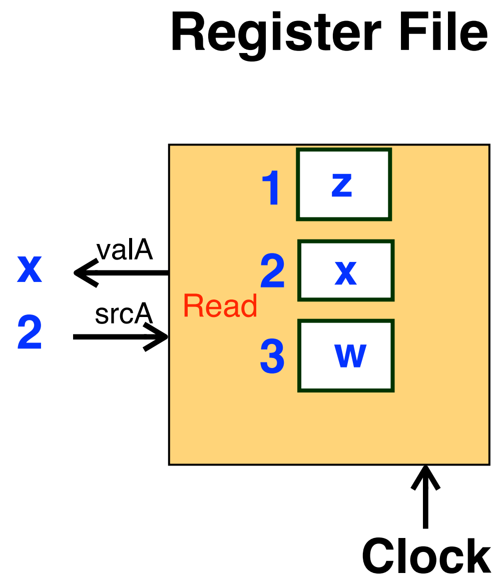
Register File

- 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



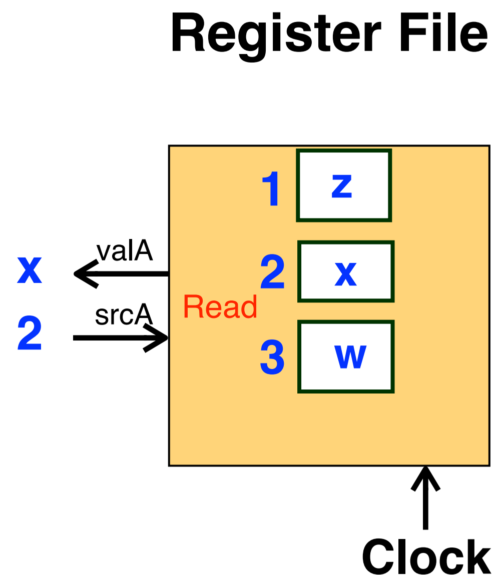
Register File

- 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



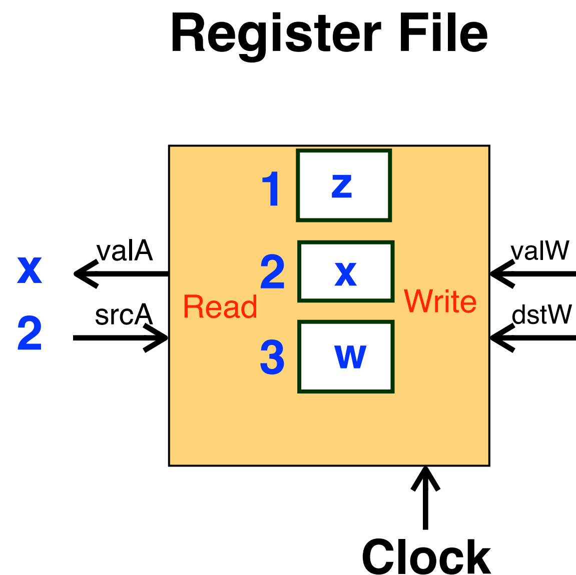
Register File

- 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



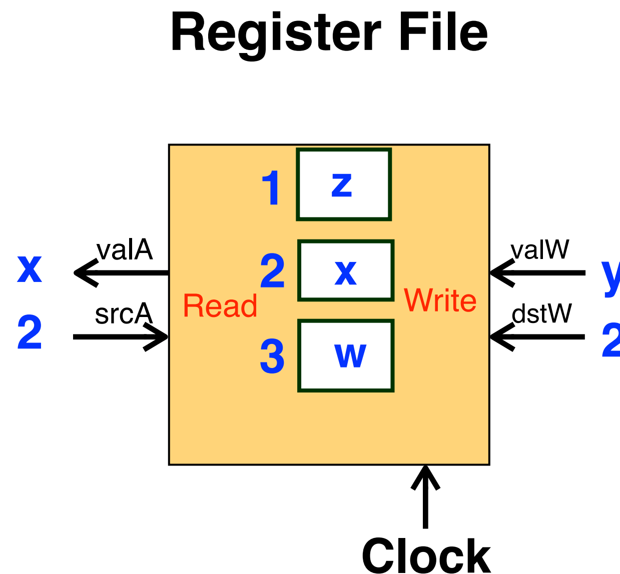
Register File

- 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



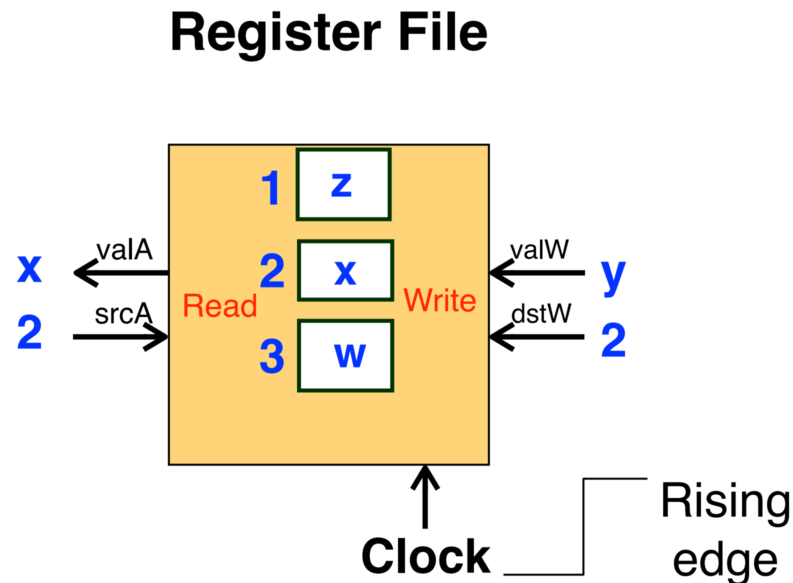
Register File

- 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



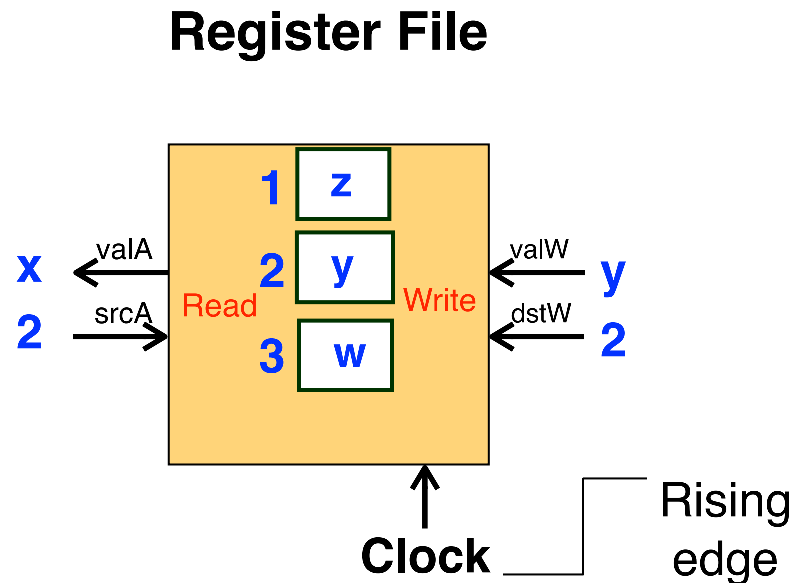
Register File

- 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



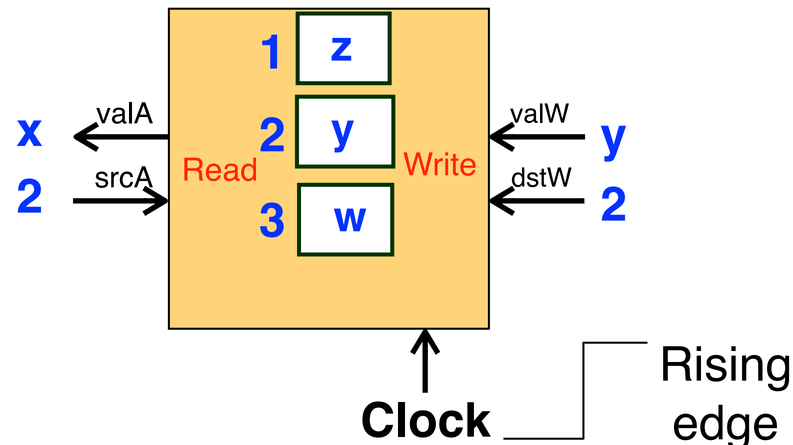
Register File

- 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



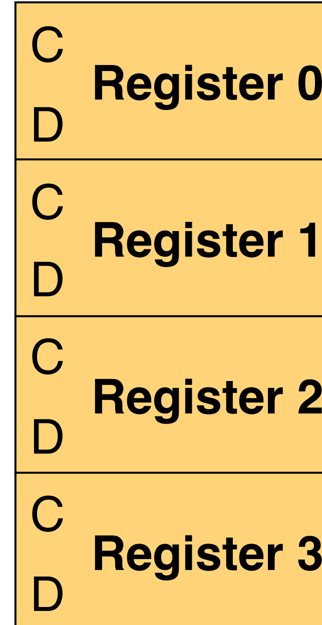
Register File

- 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??



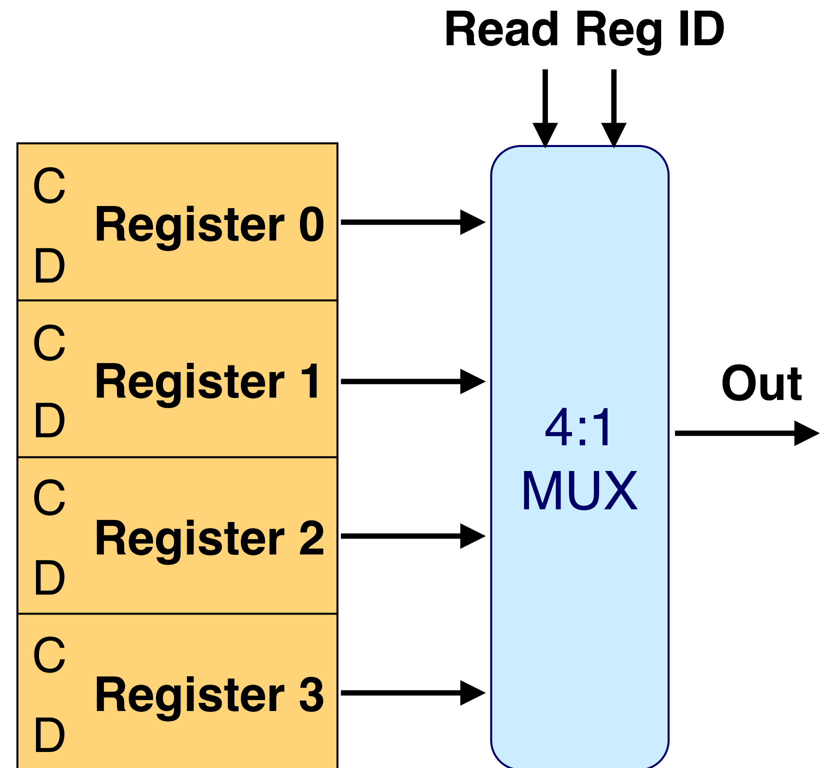
Register File Read

- Continuously read a register independent of the clock signal

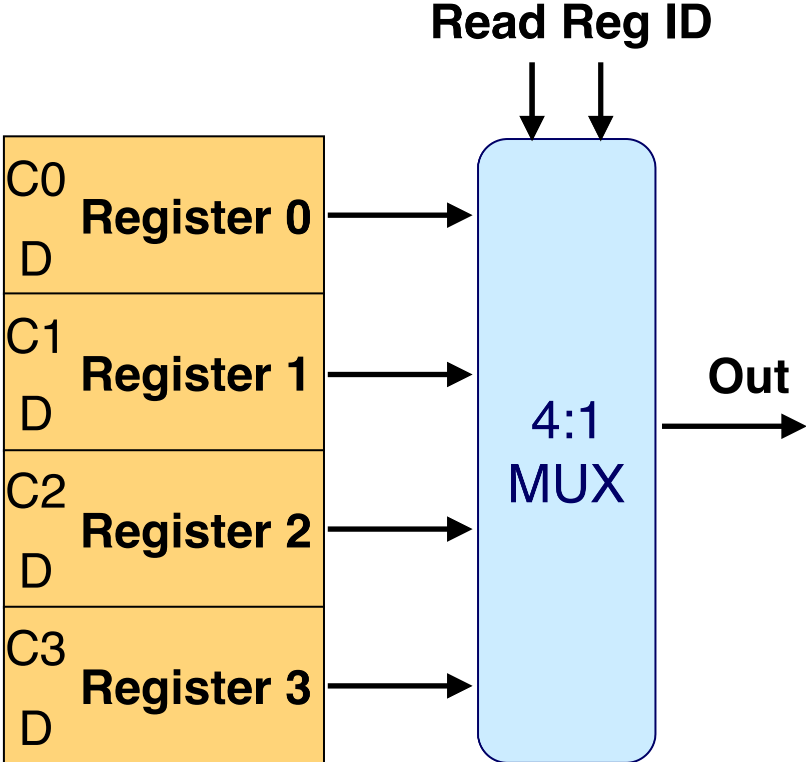


Register File Read

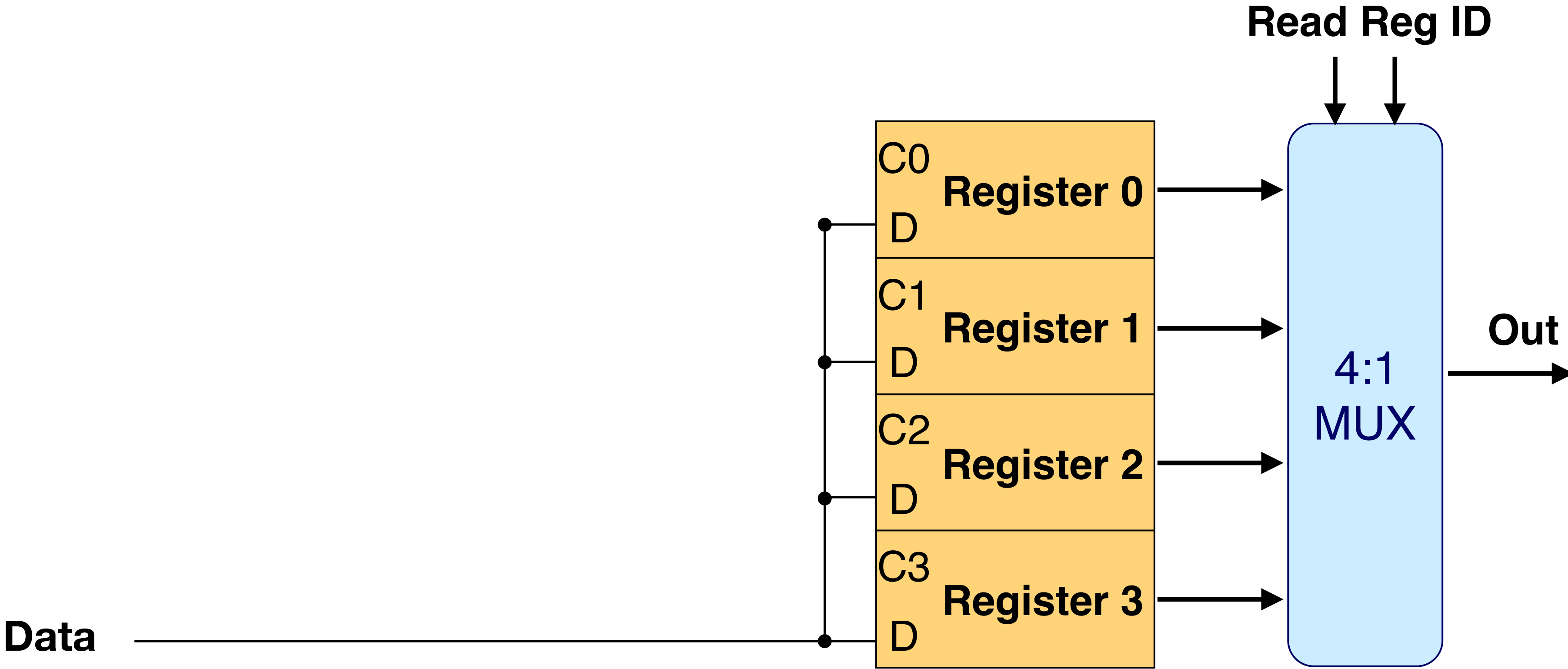
- Continuously read a register independent of the clock signal



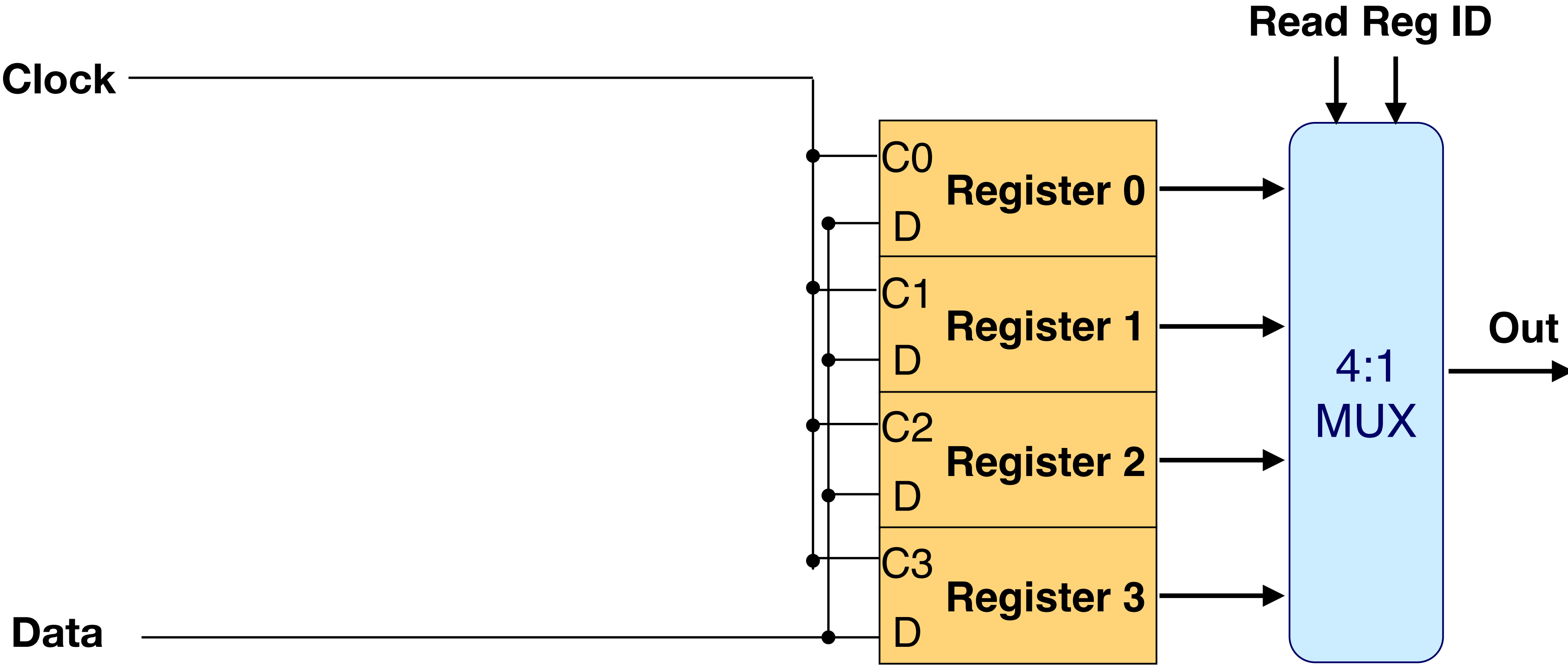
Register File Write



Register File Write

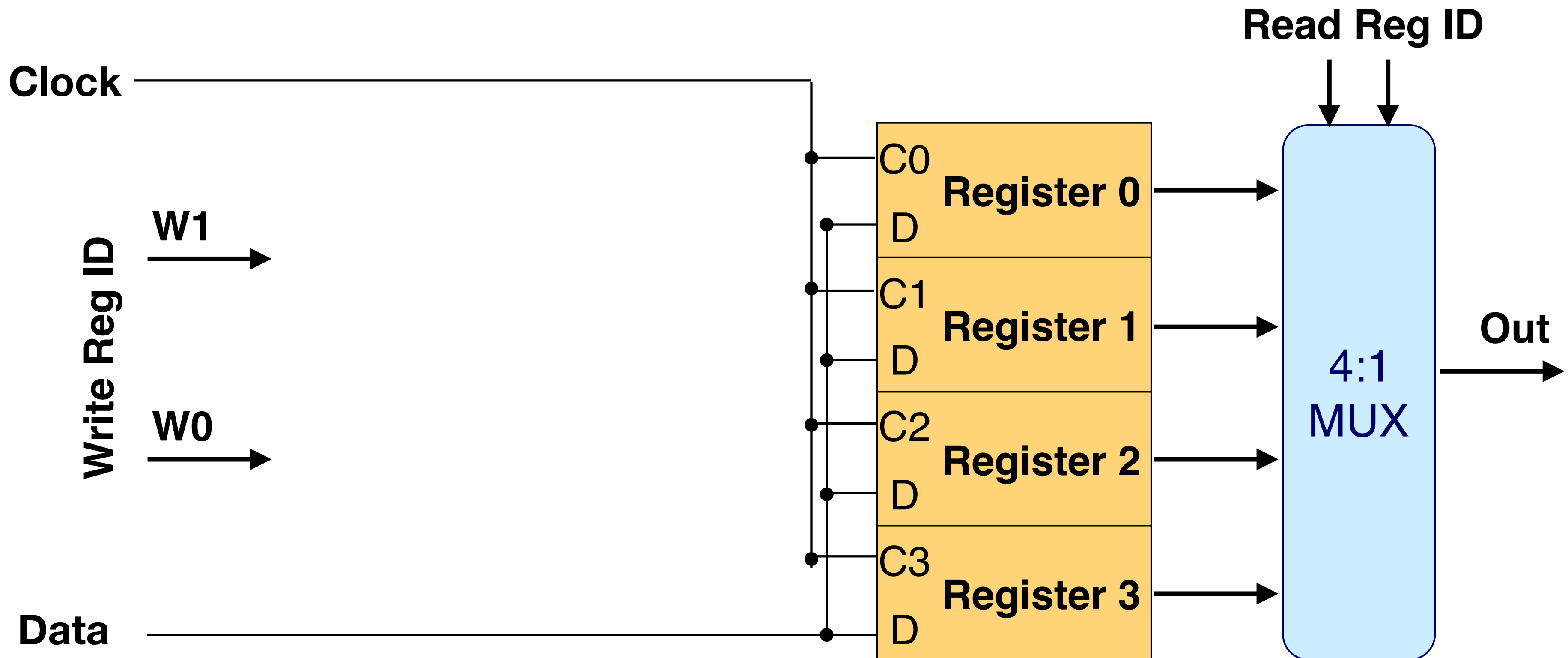


Register File Write



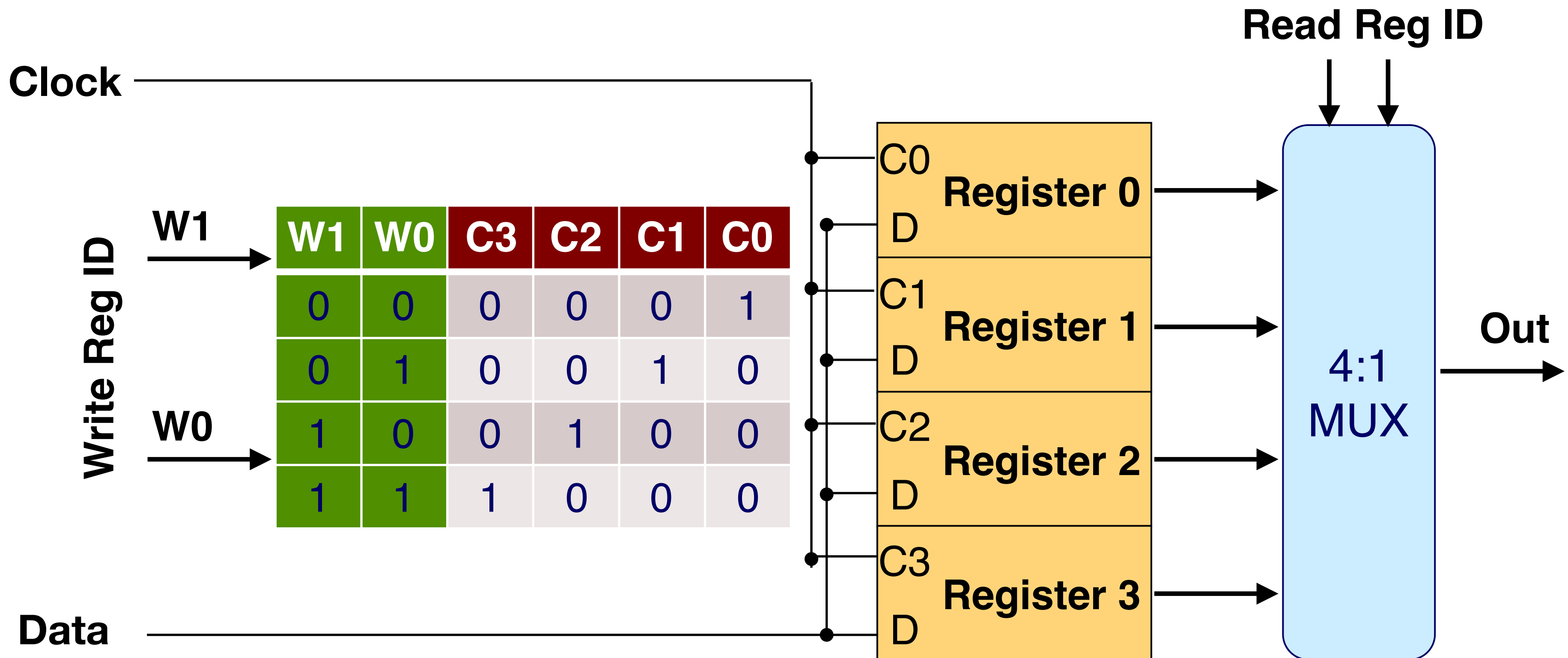
Register File Write

- 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

W0 _

W1 _

_ C0

_ C1

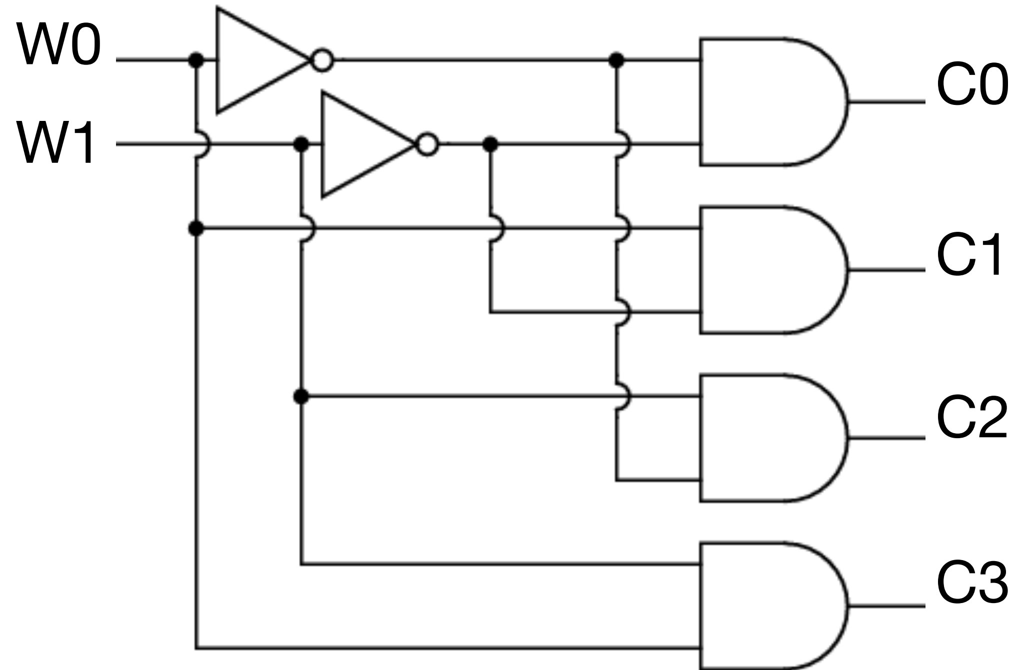
_ C2

_ C3

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

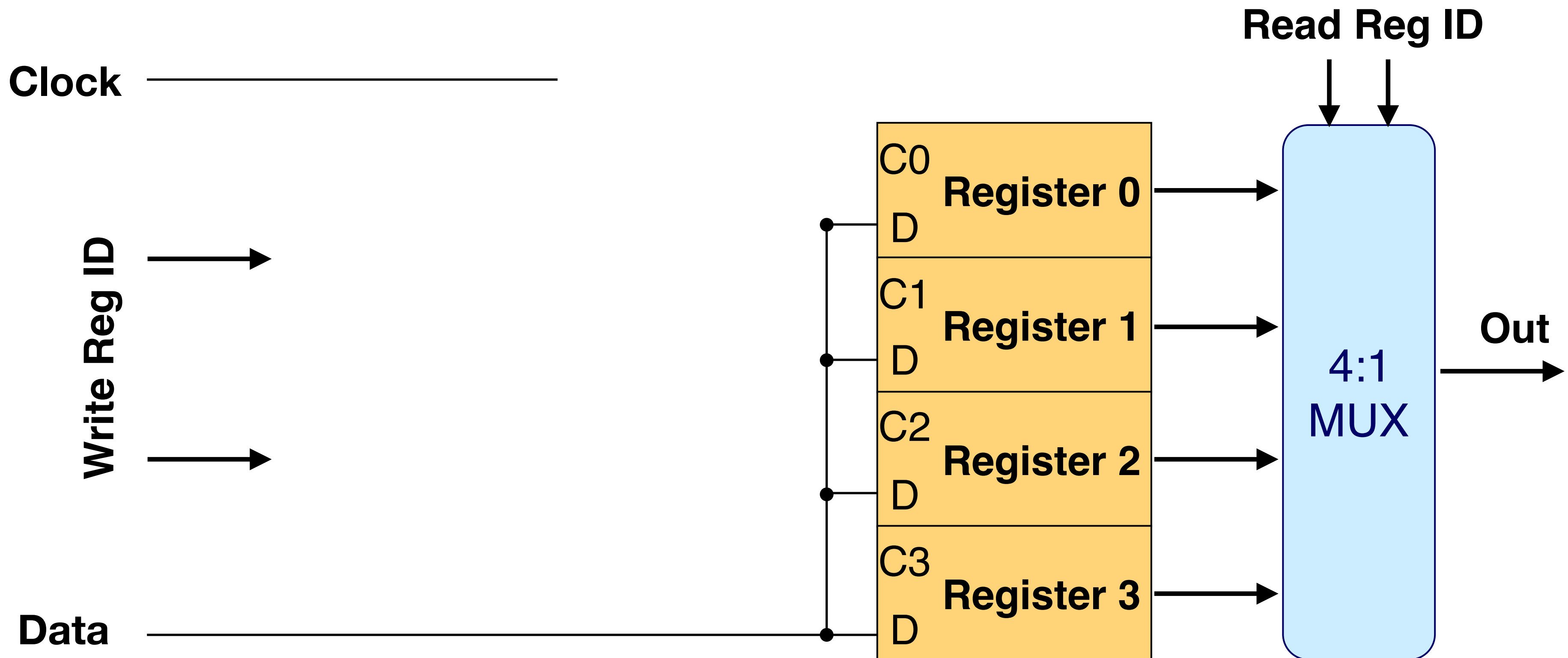
Decoder

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

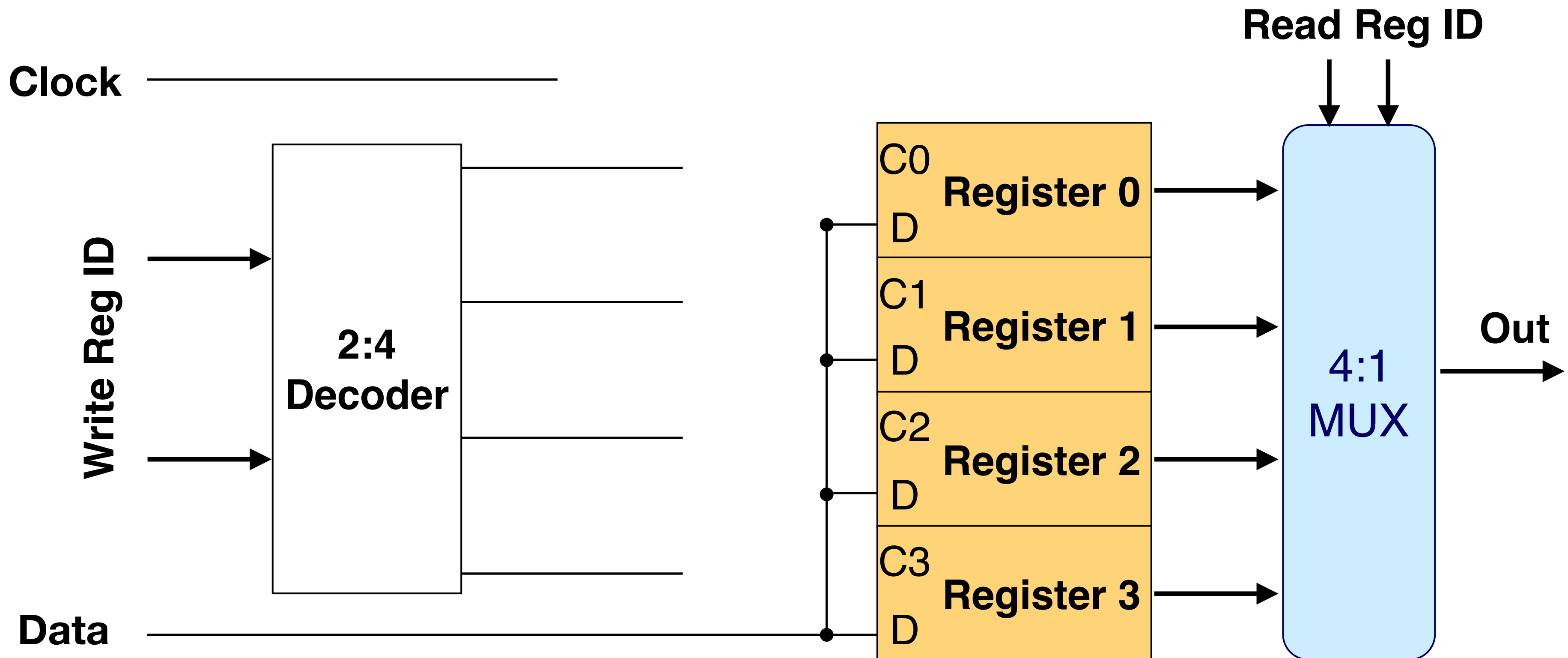


C0 = !W1 & !W0
C1= !W1 & W0
C2 = W1 & !W0
C3 = W1 & W0

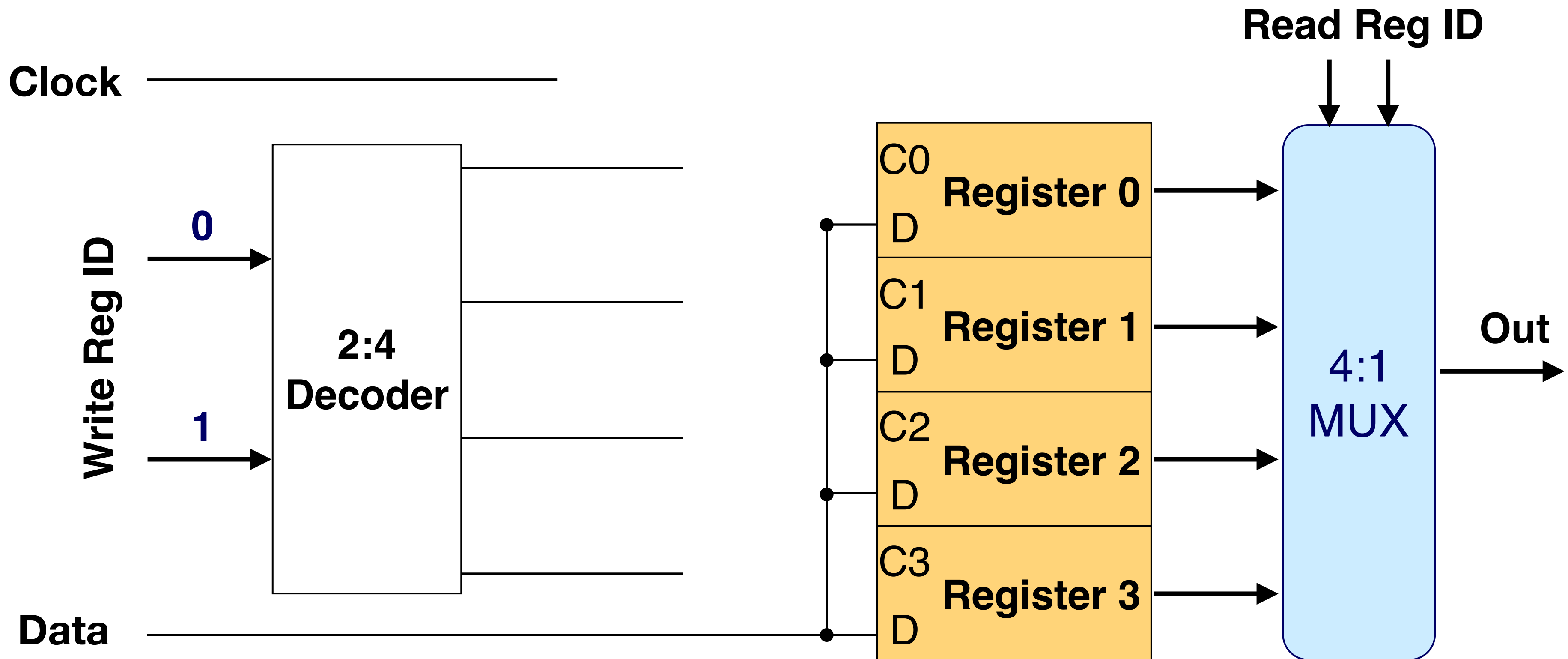
Register File Write



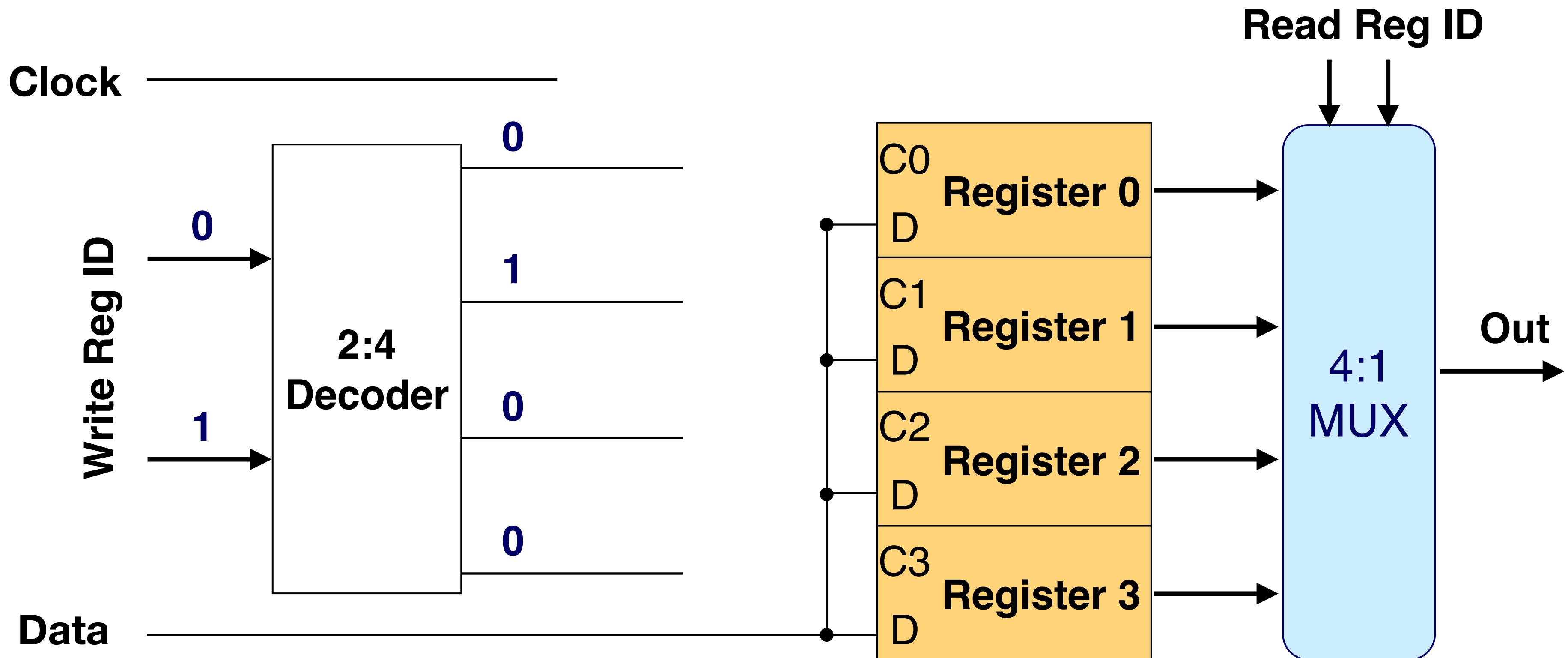
Register File Write



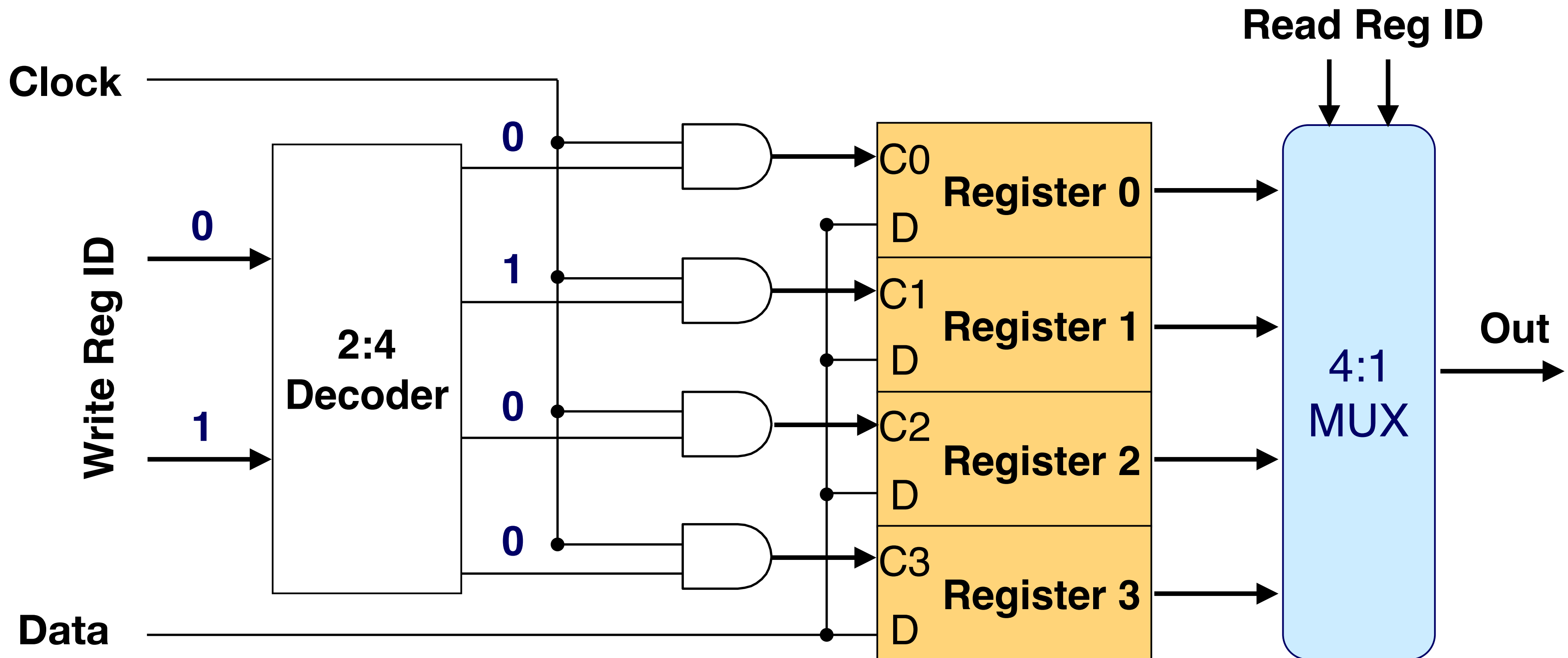
Register File Write



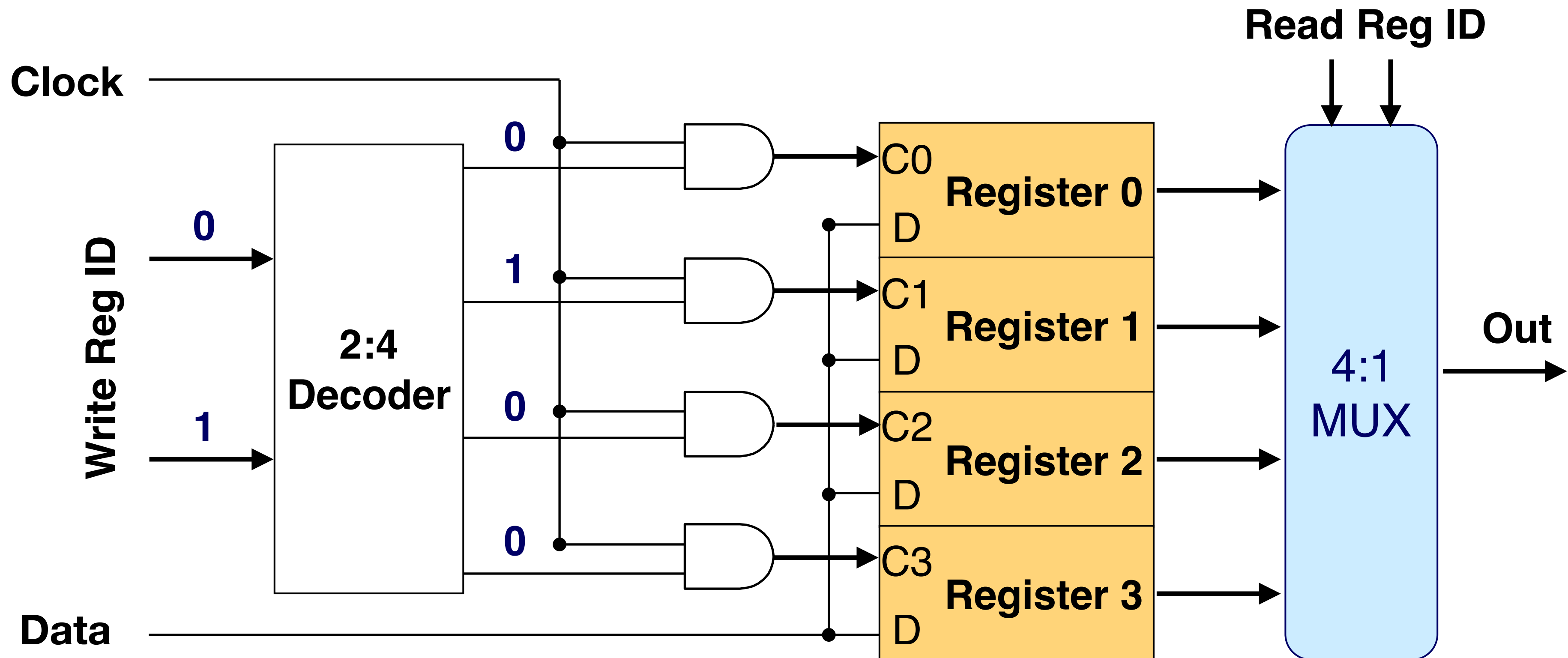
Register File Write



Register File Write



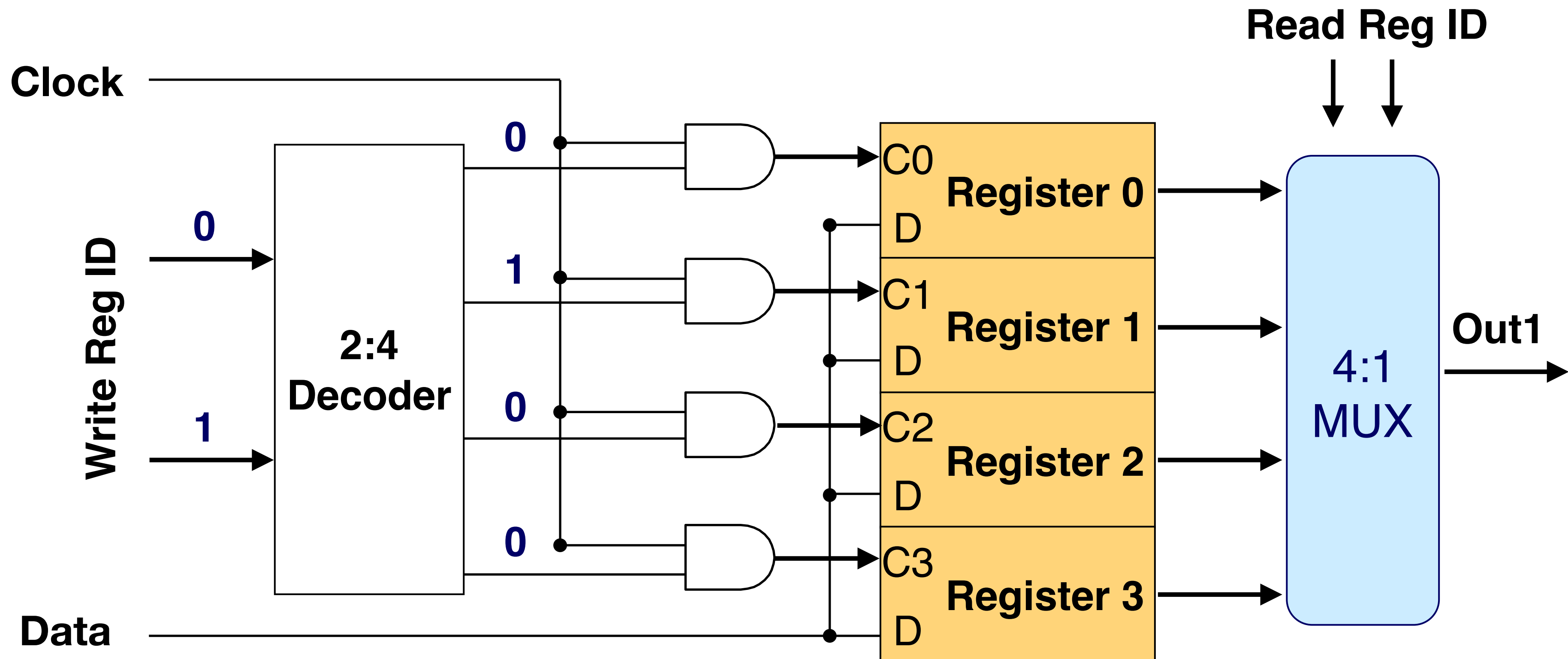
Register File Write



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

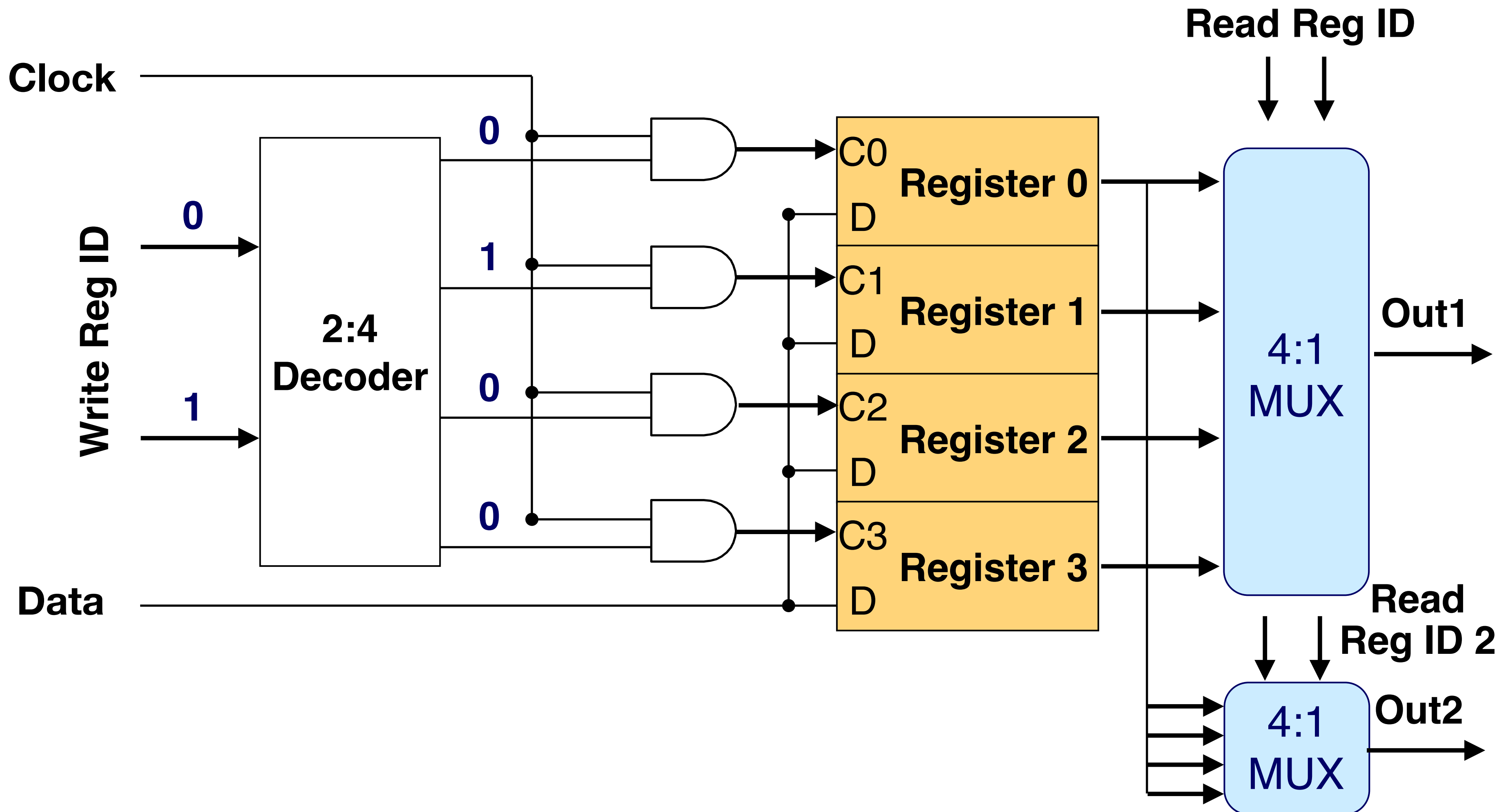
Multi-Port Register File

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



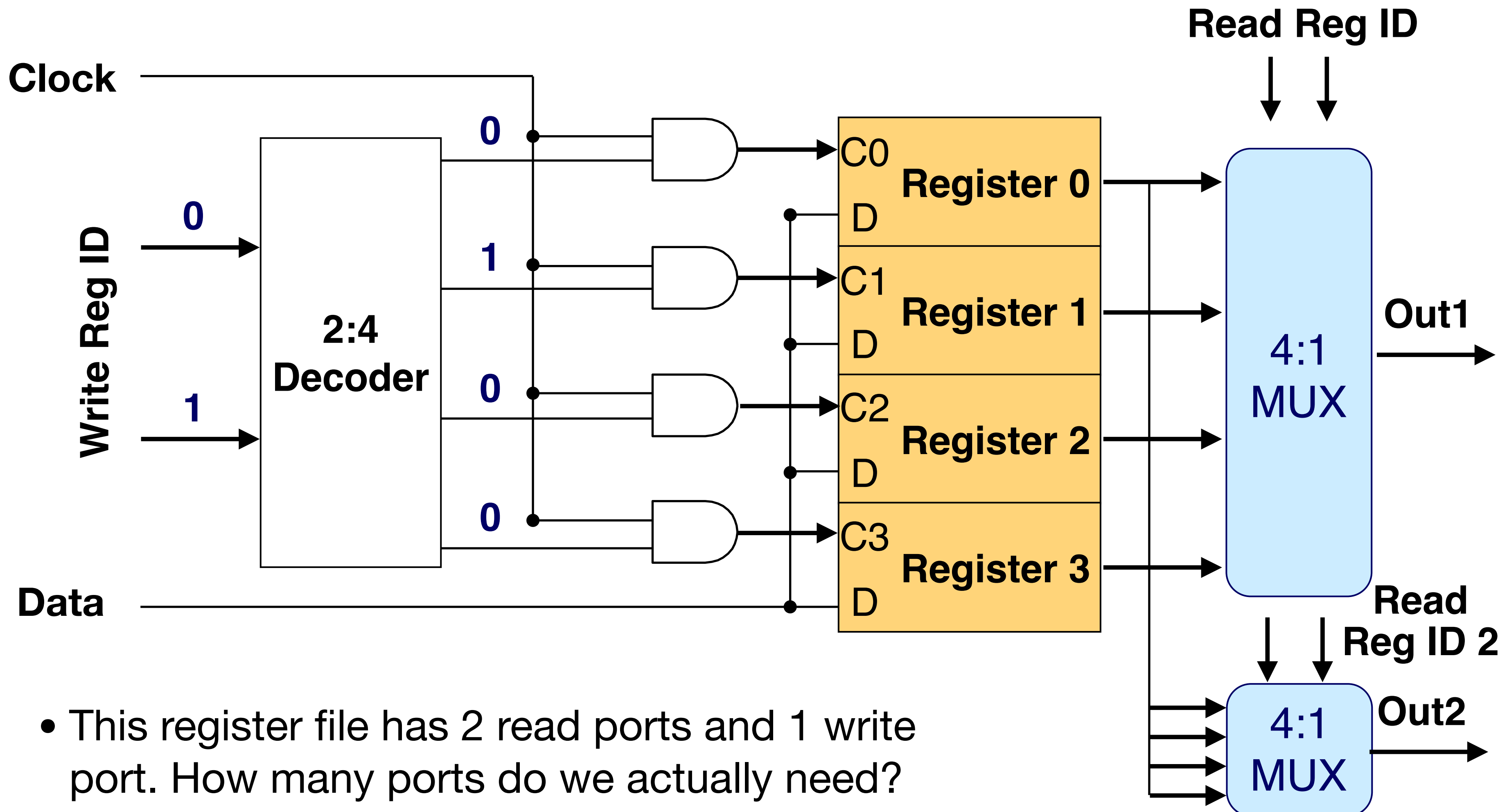
Multi-Port Register File

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



Multi-Port Register File

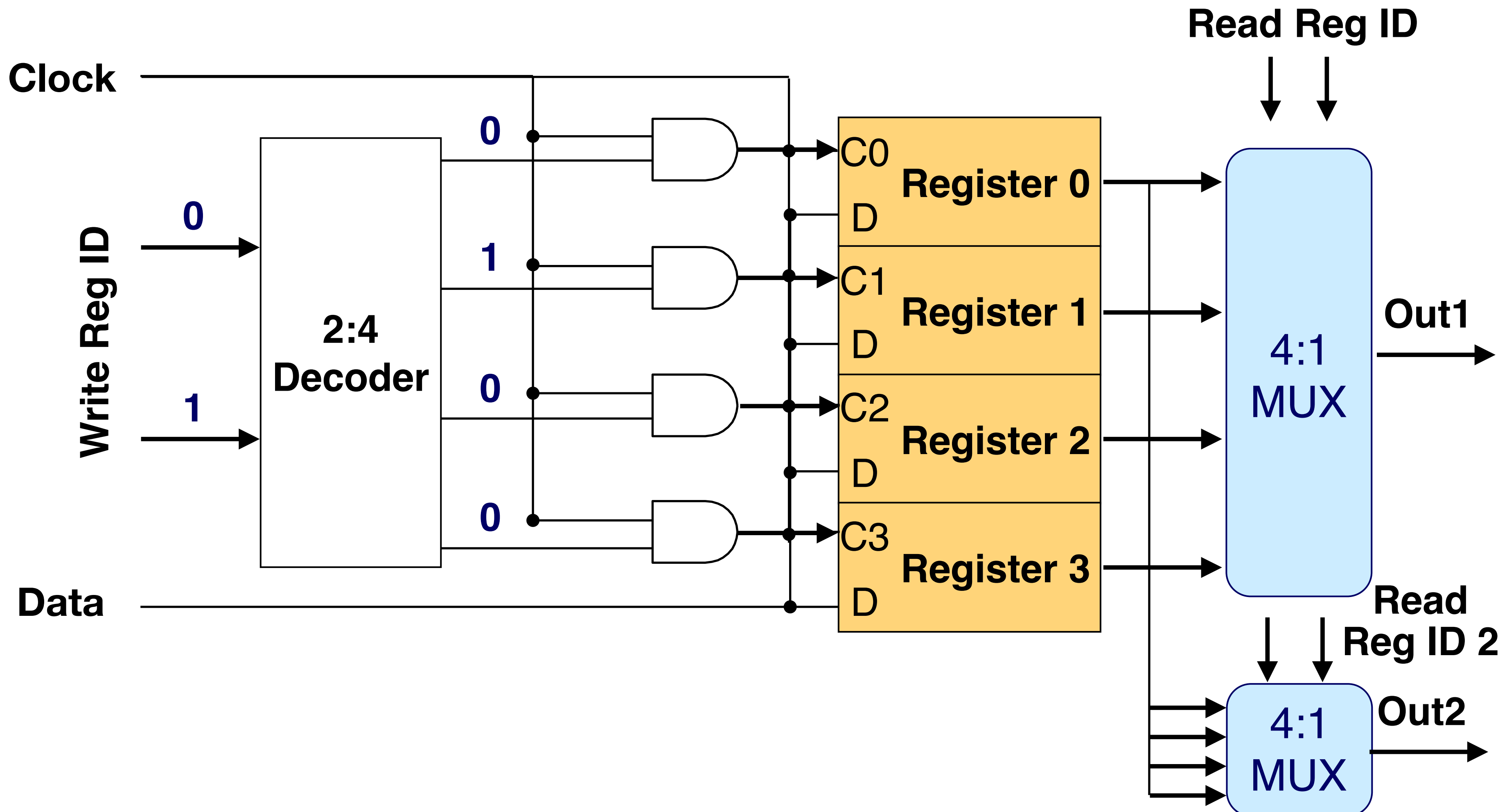
- What if we want to read multiple registers at the same time?



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

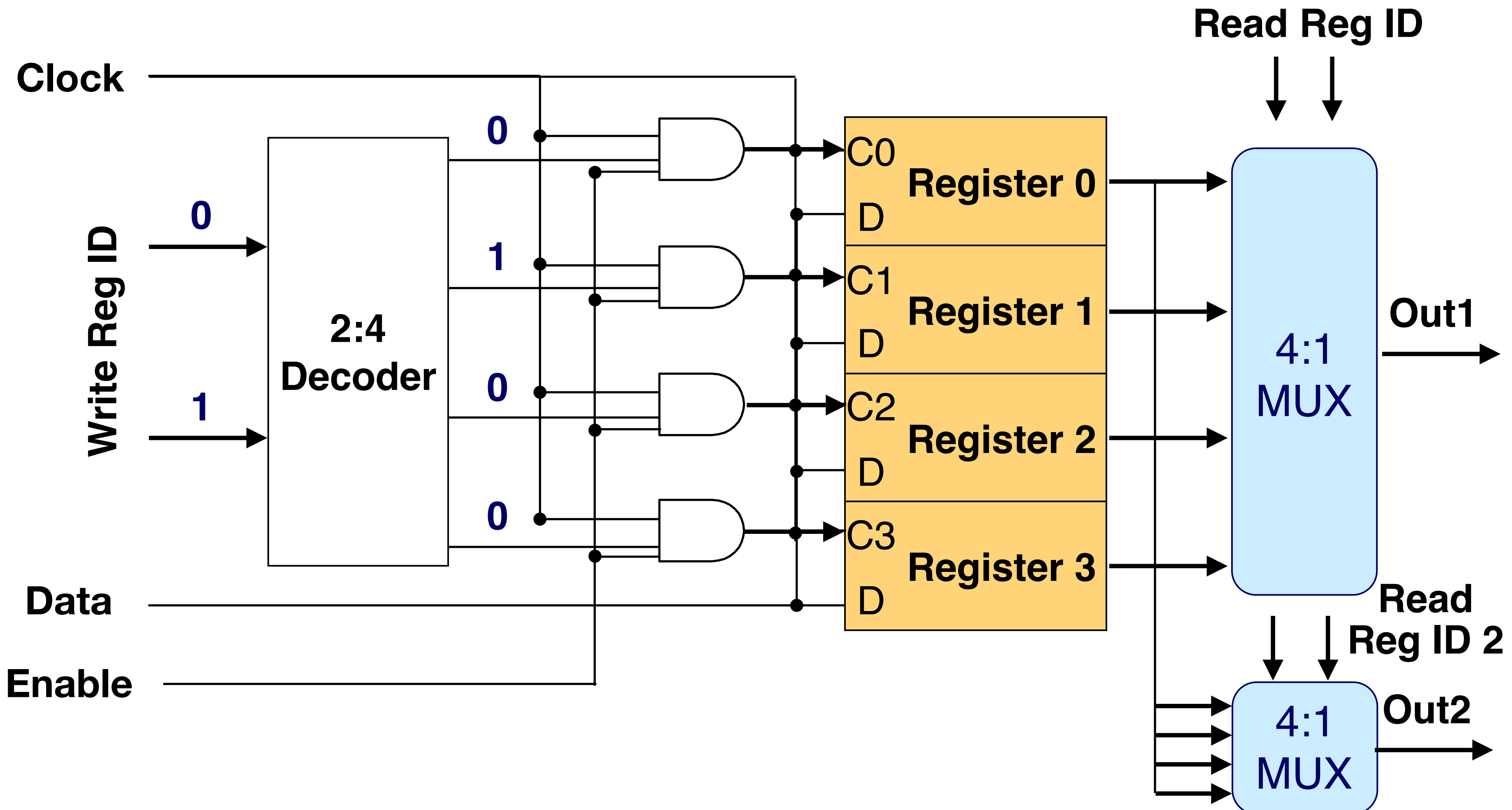
Multi-Port Register File

- 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?

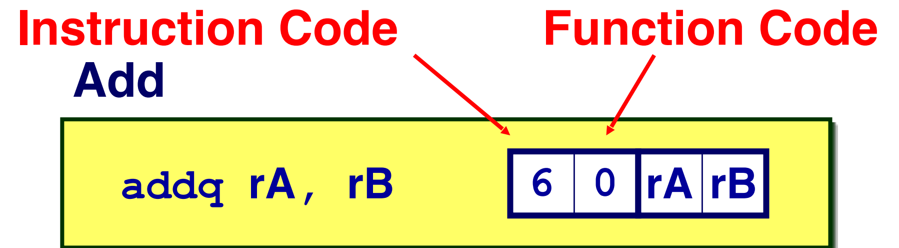


Processor Microarchitecture

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

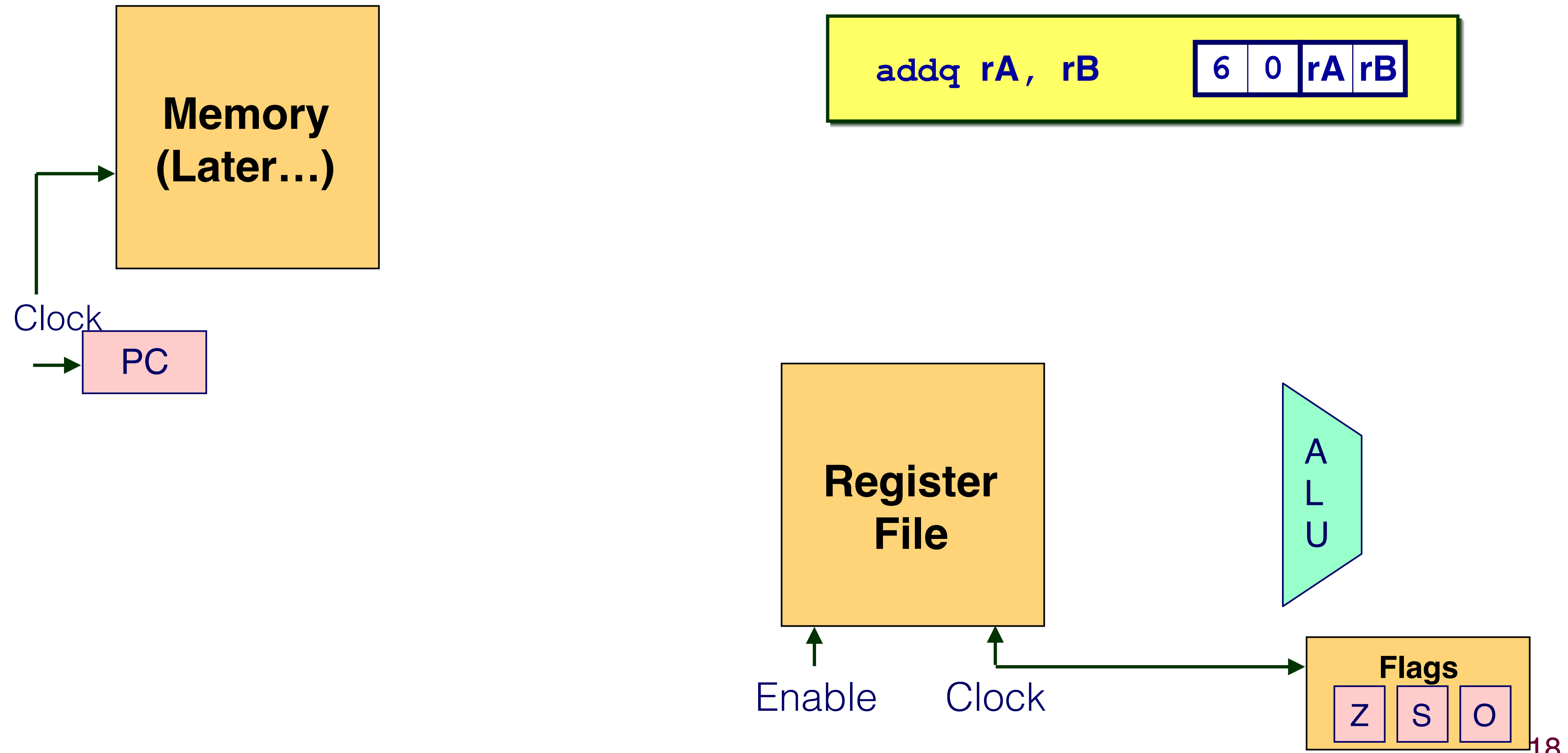
Executing an ADD instruction

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



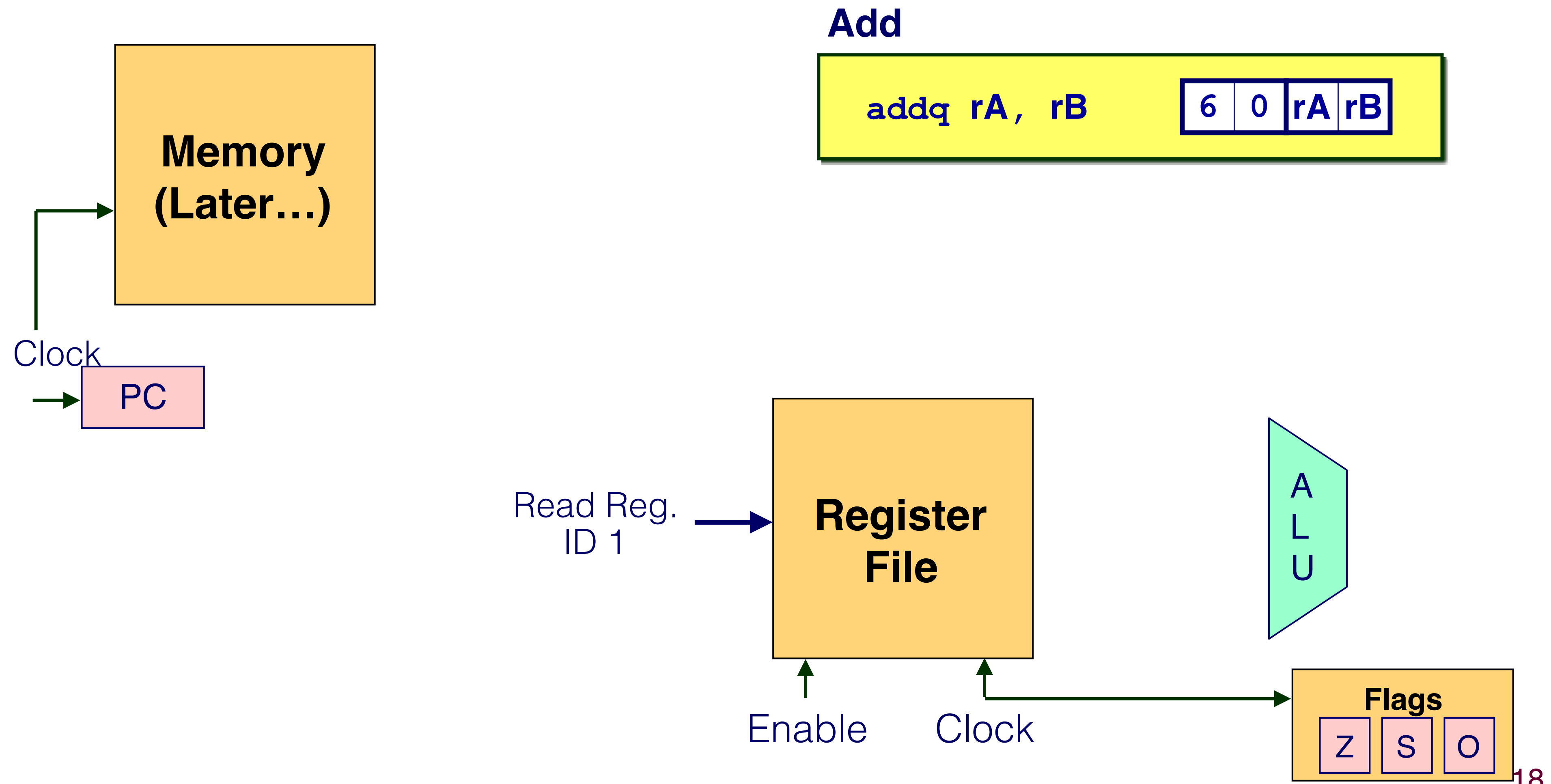
Executing an ADD instruction

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



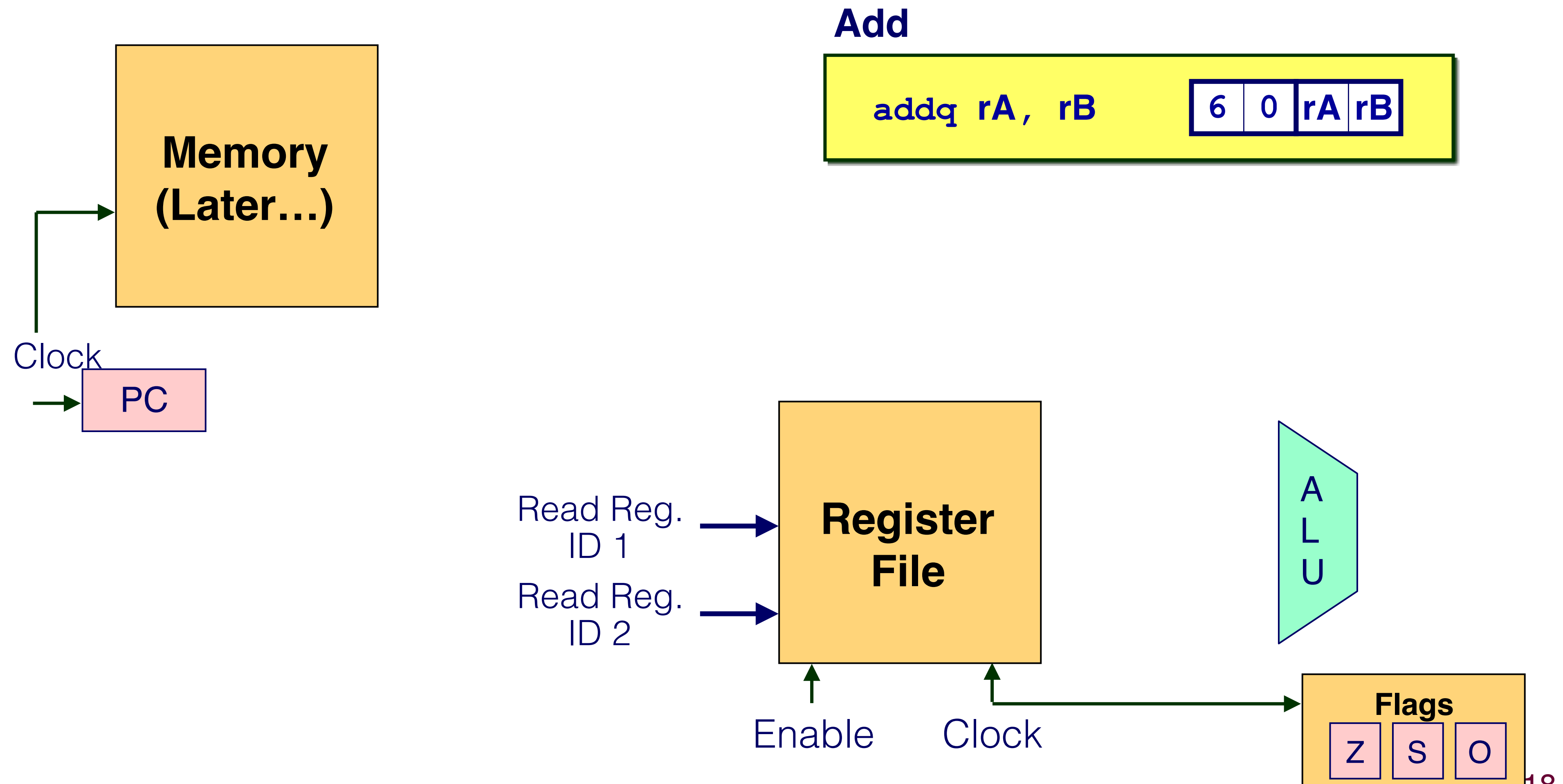
Executing an ADD instruction

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



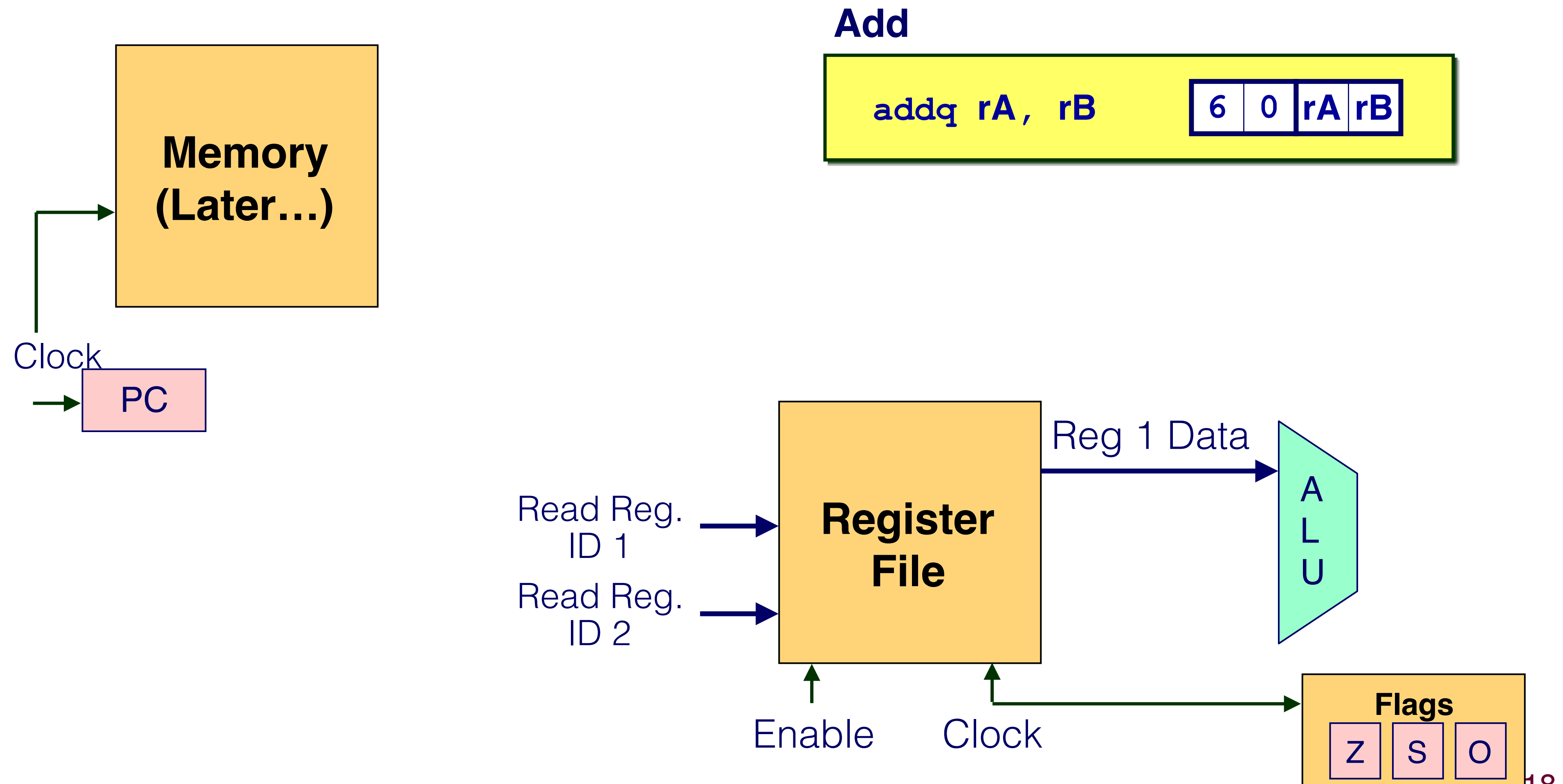
Executing an ADD instruction

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



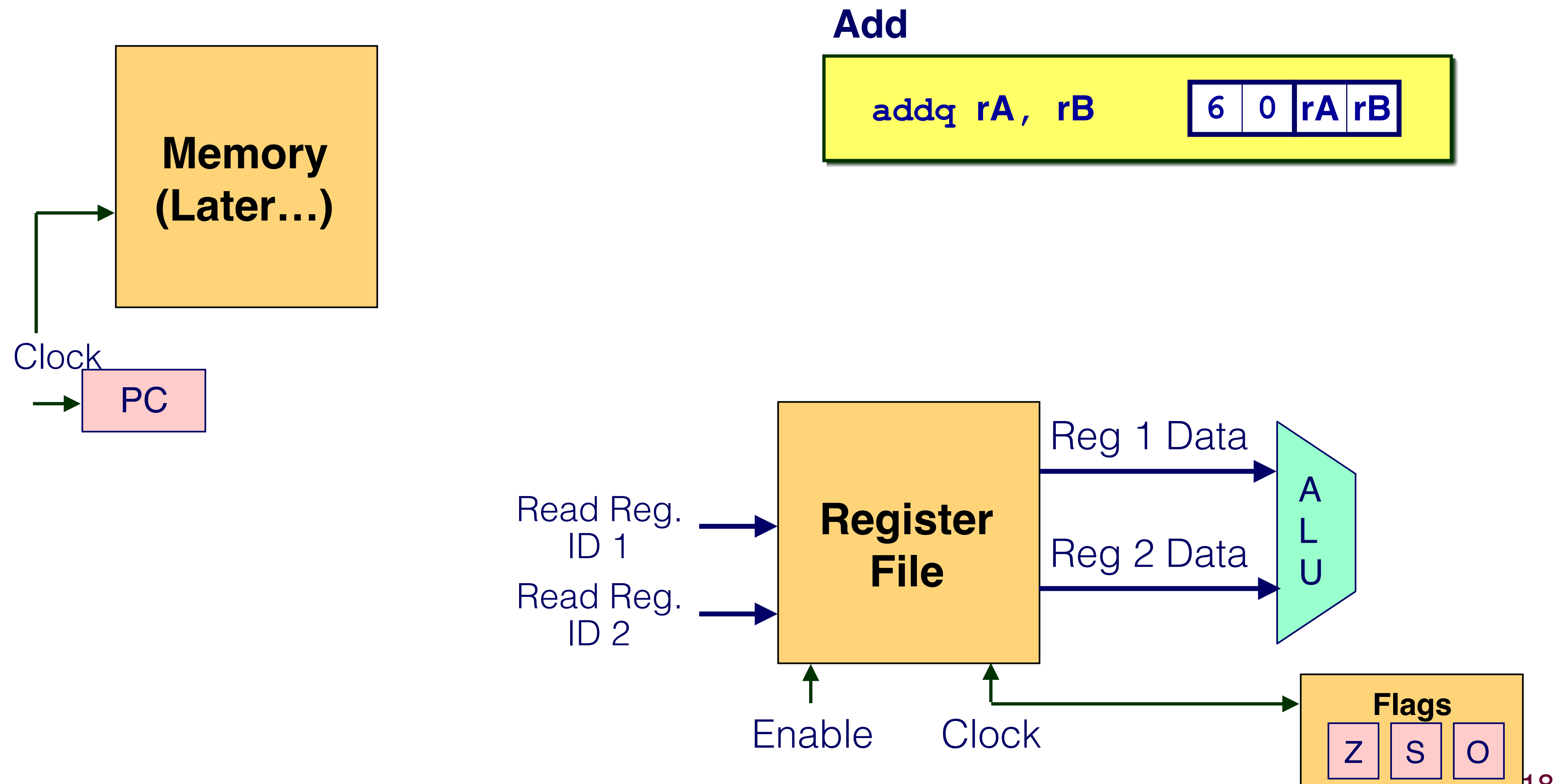
Executing an ADD instruction

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



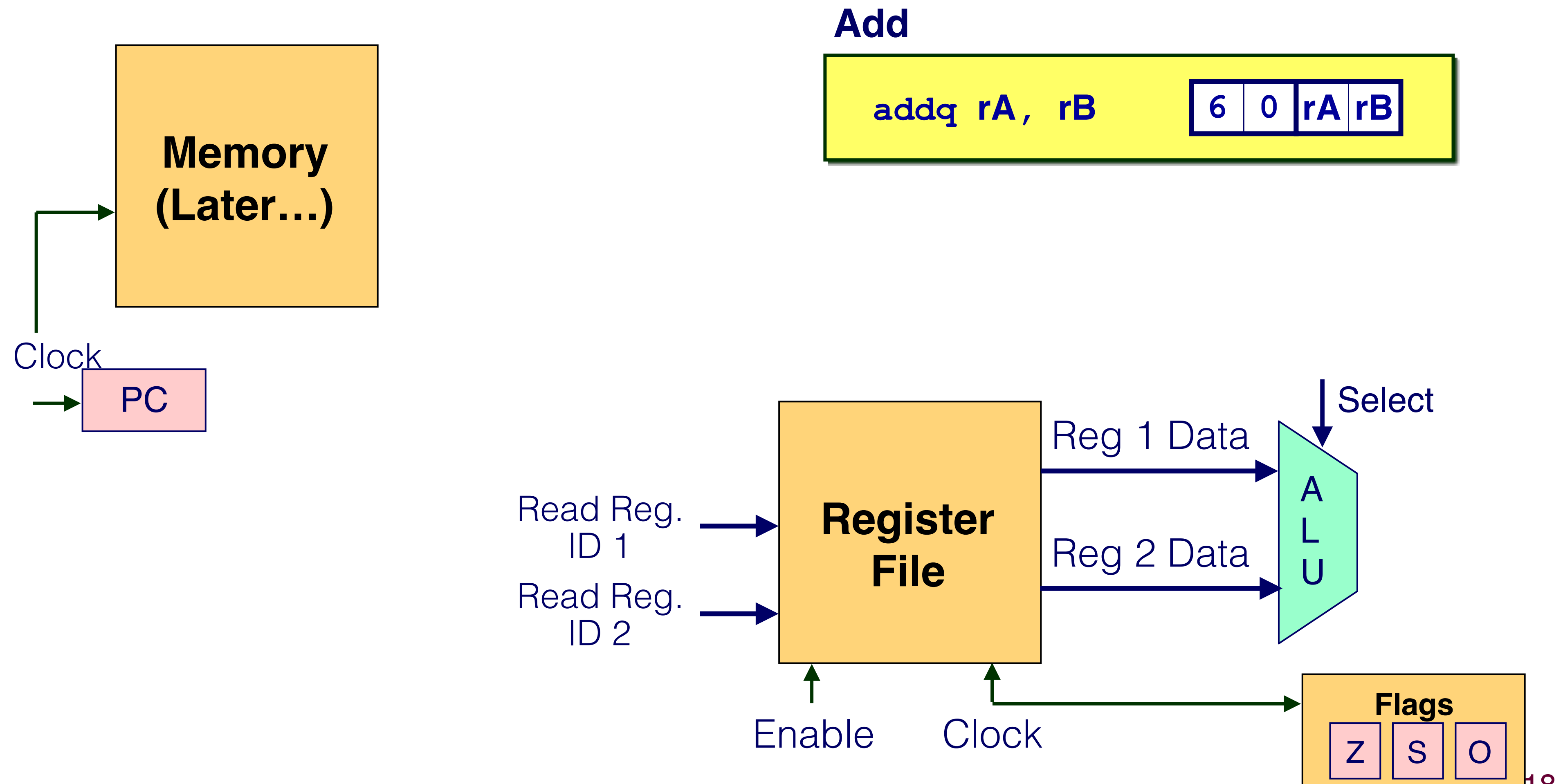
Executing an ADD instruction

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



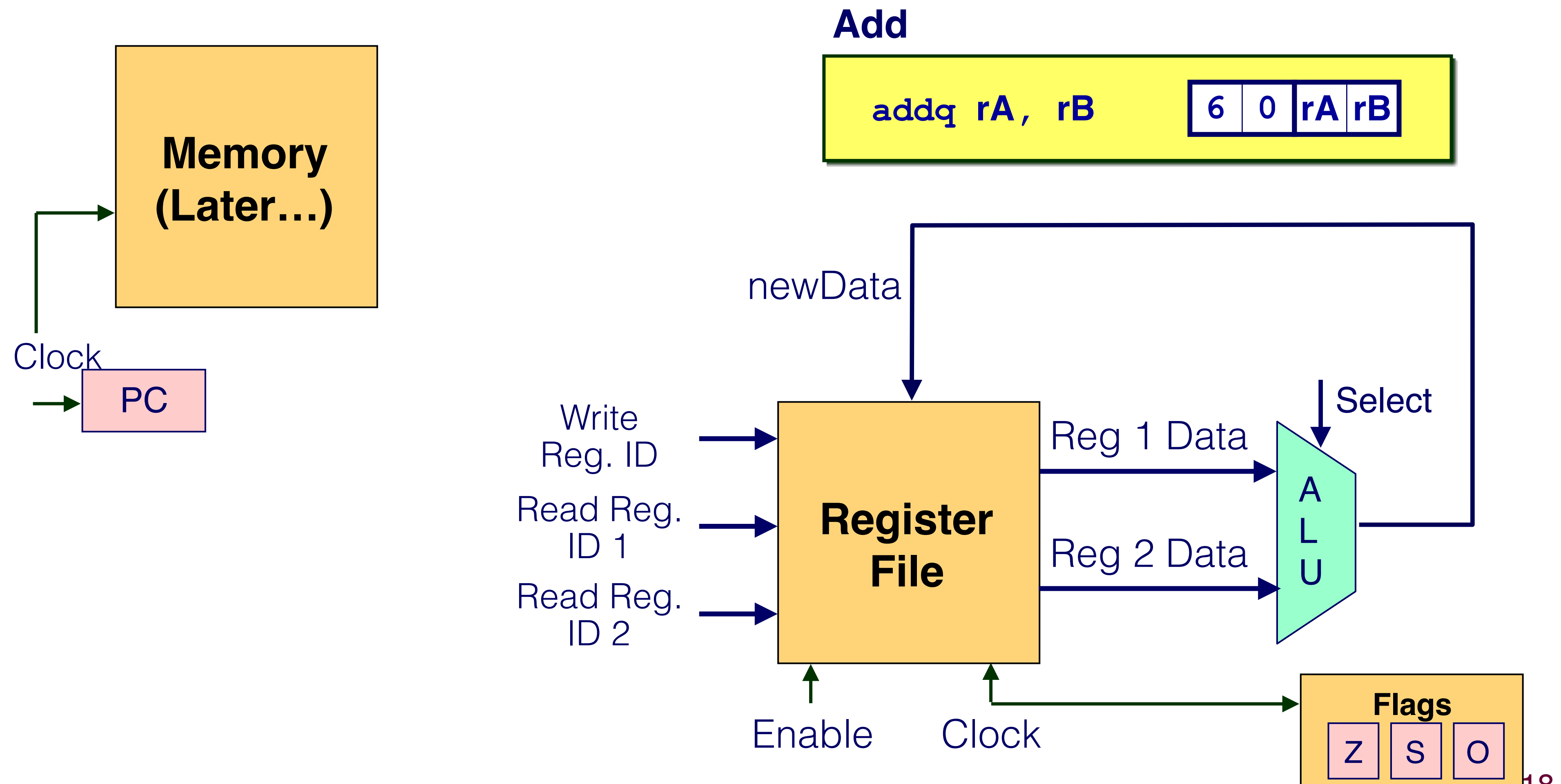
Executing an ADD instruction

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



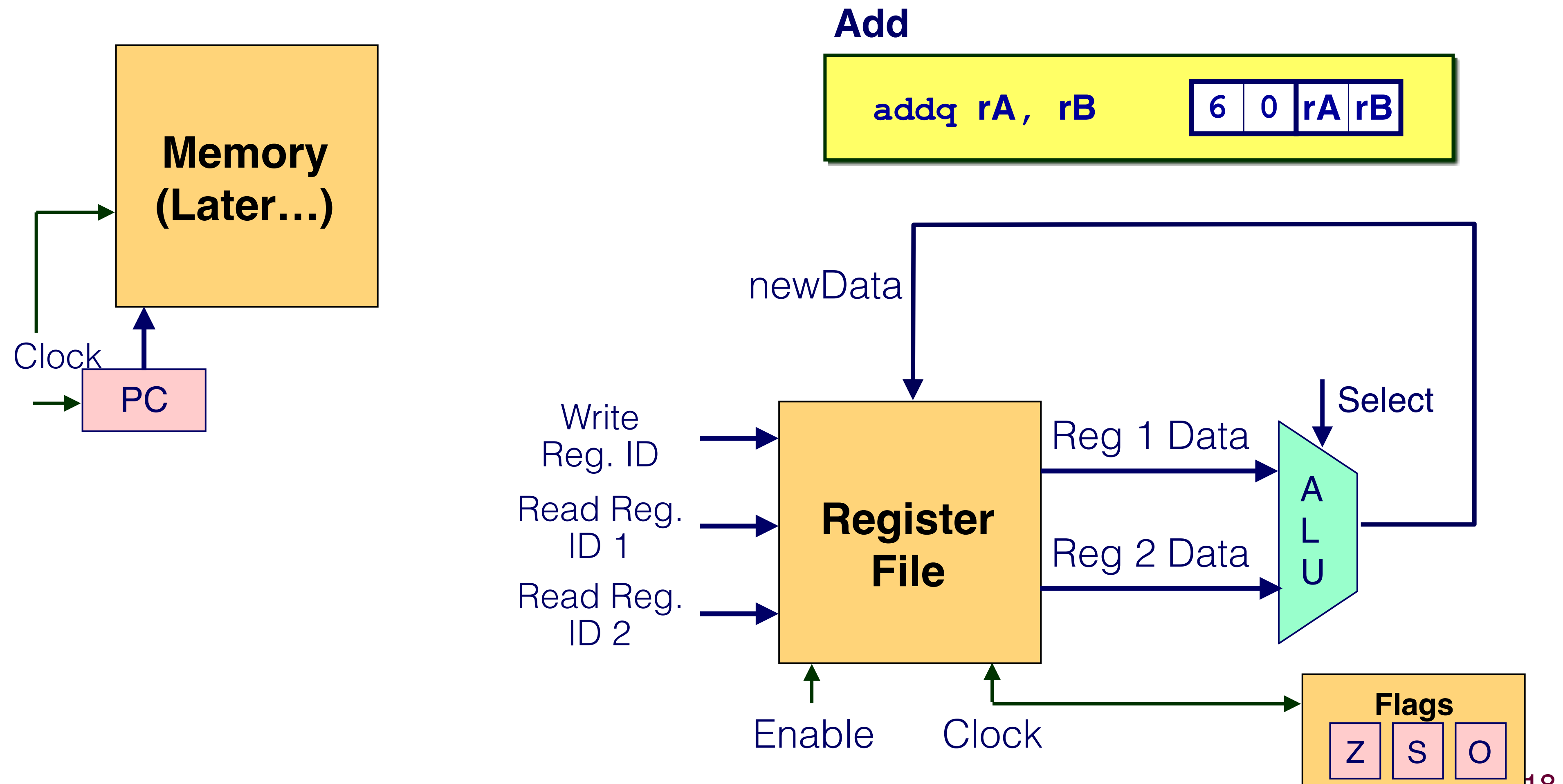
Executing an ADD instruction

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



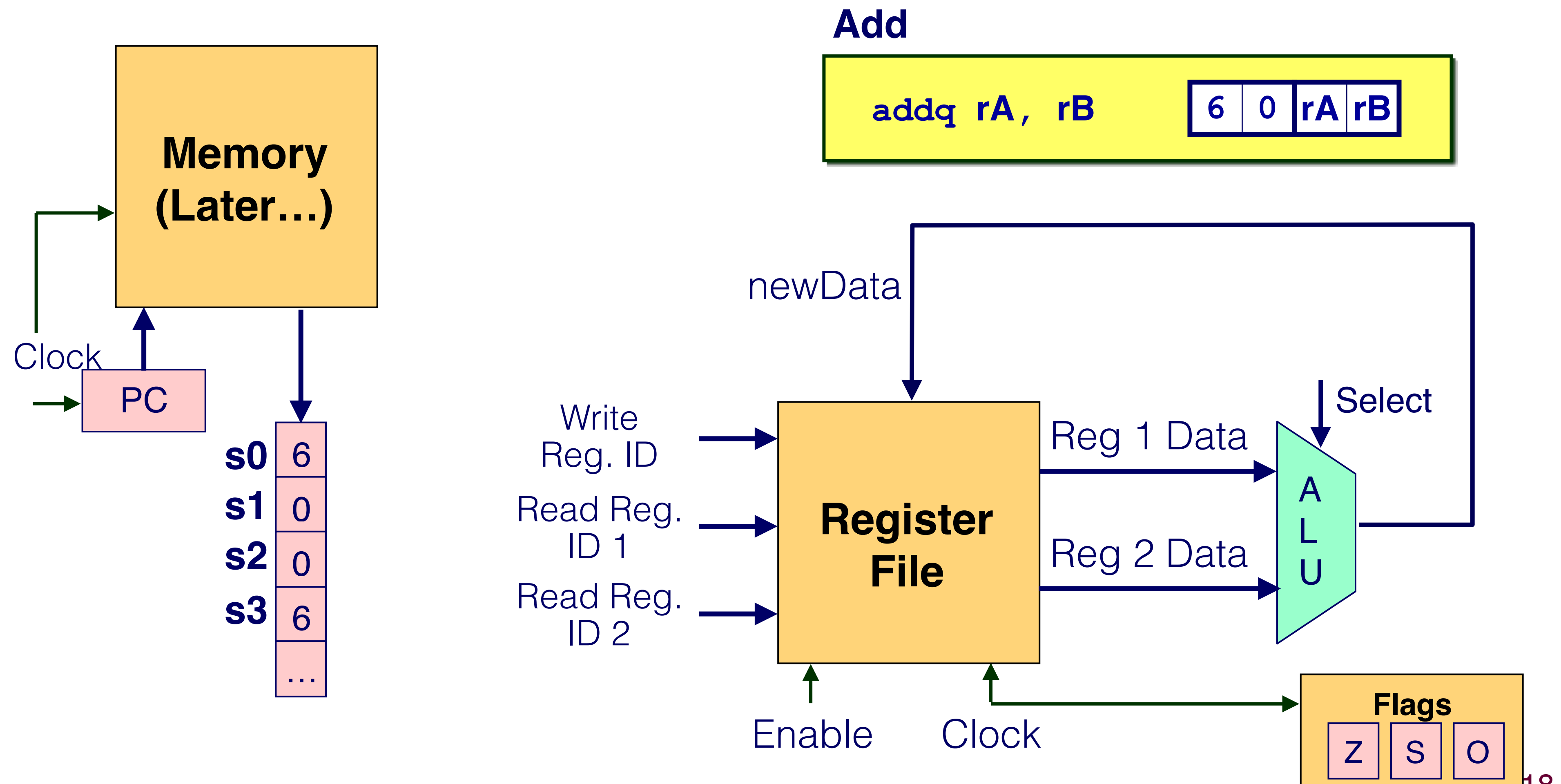
Executing an ADD instruction

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



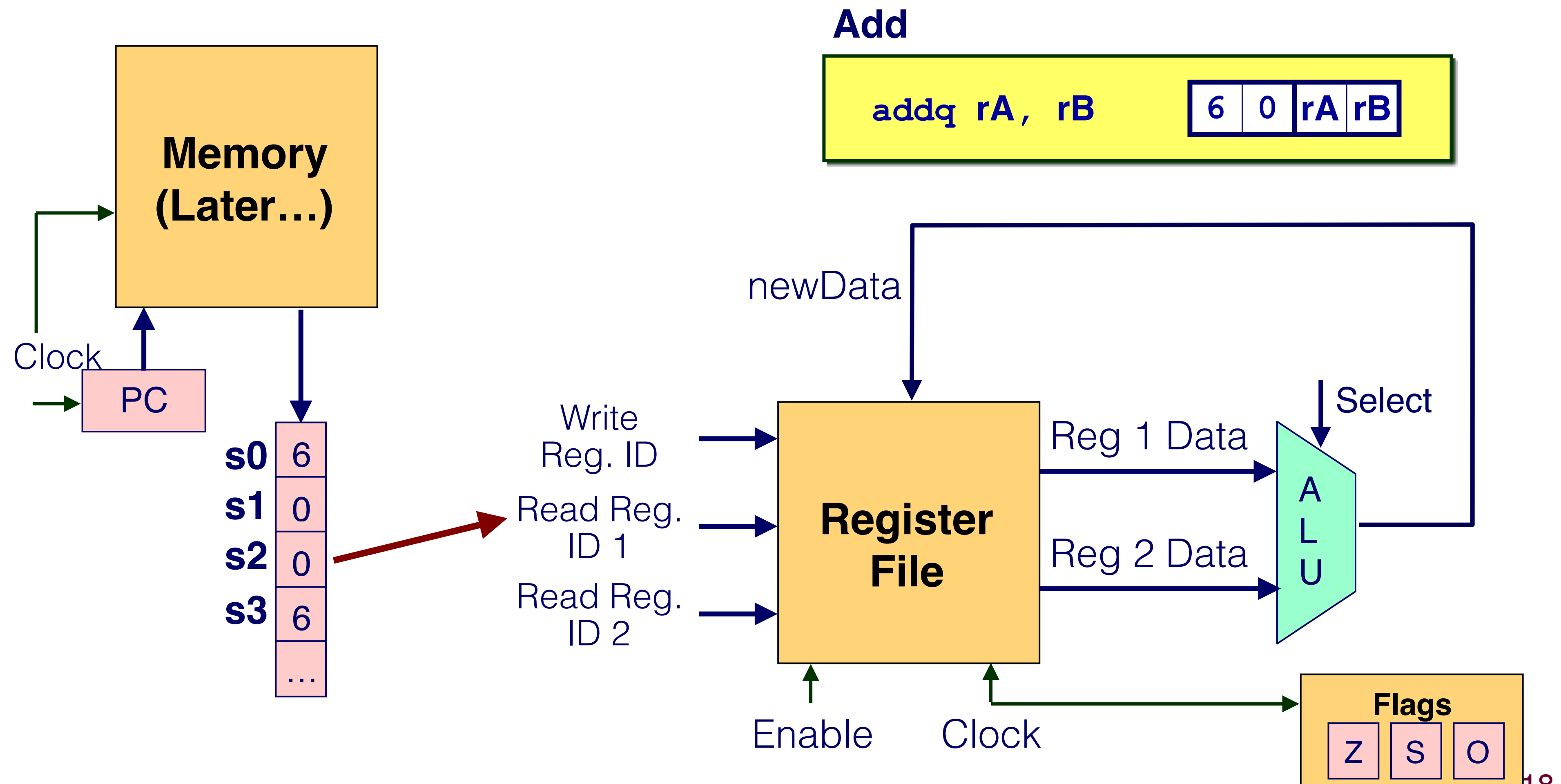
Executing an ADD instruction

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



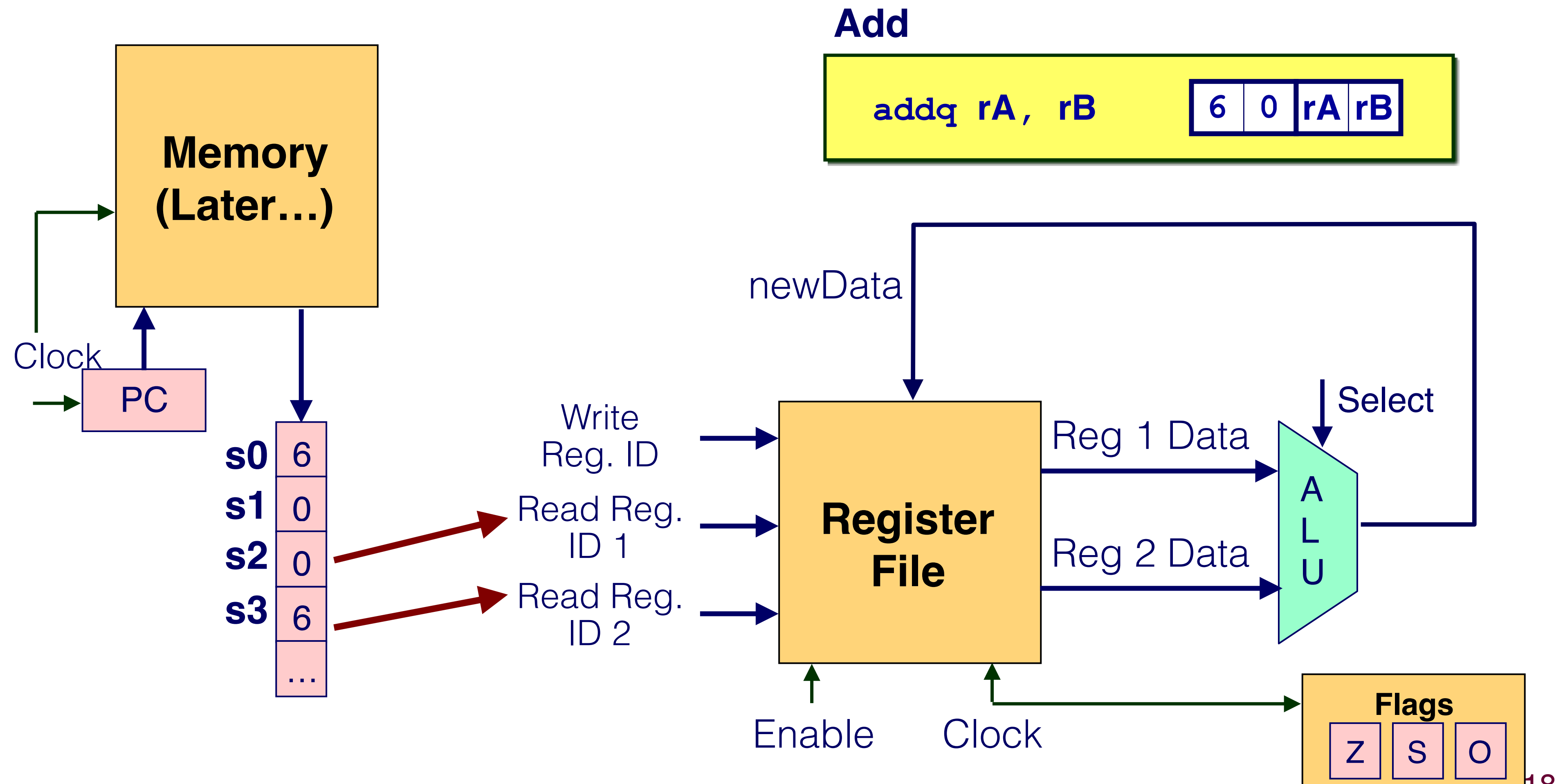
Executing an ADD instruction

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



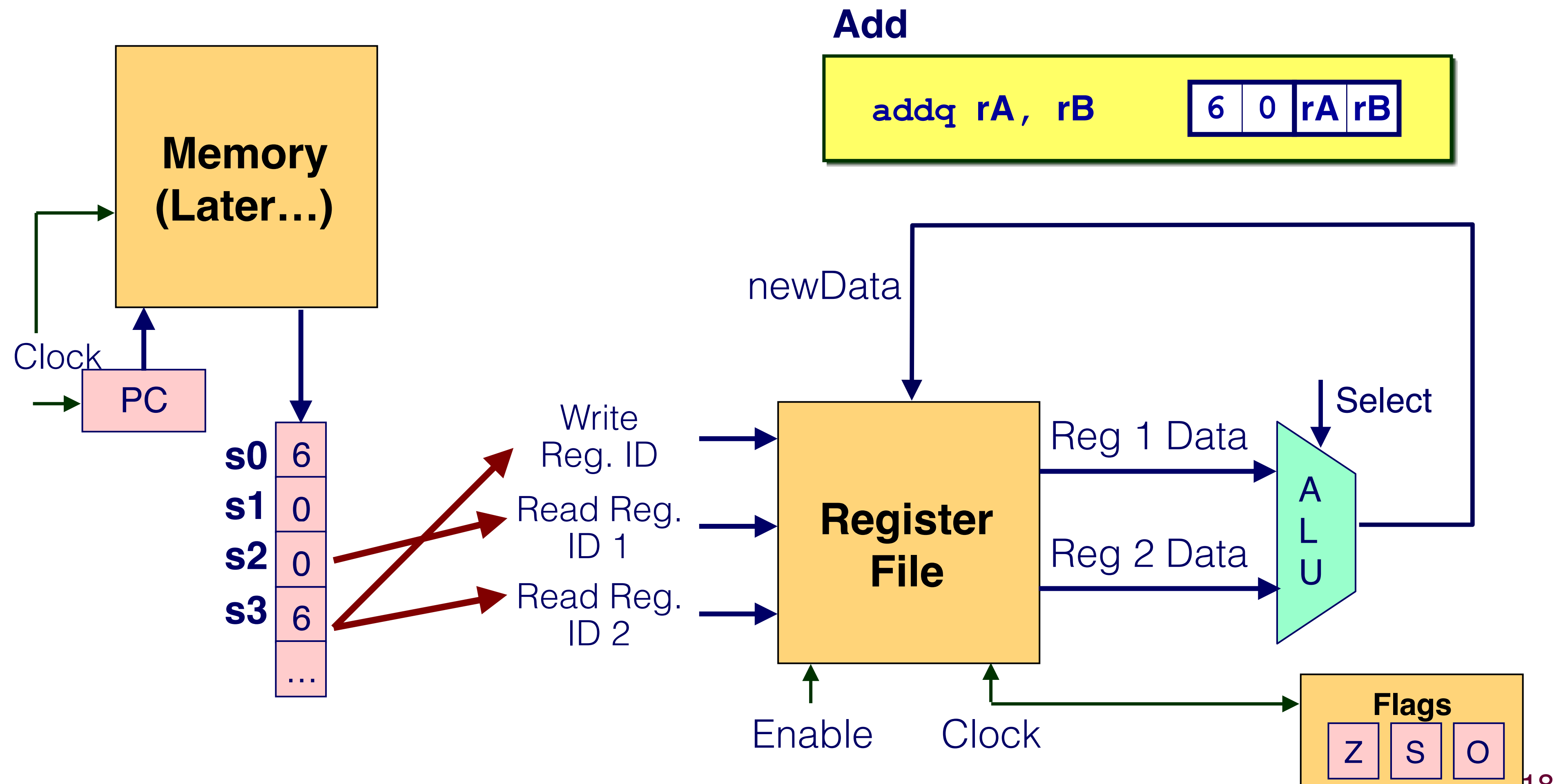
Executing an ADD instruction

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



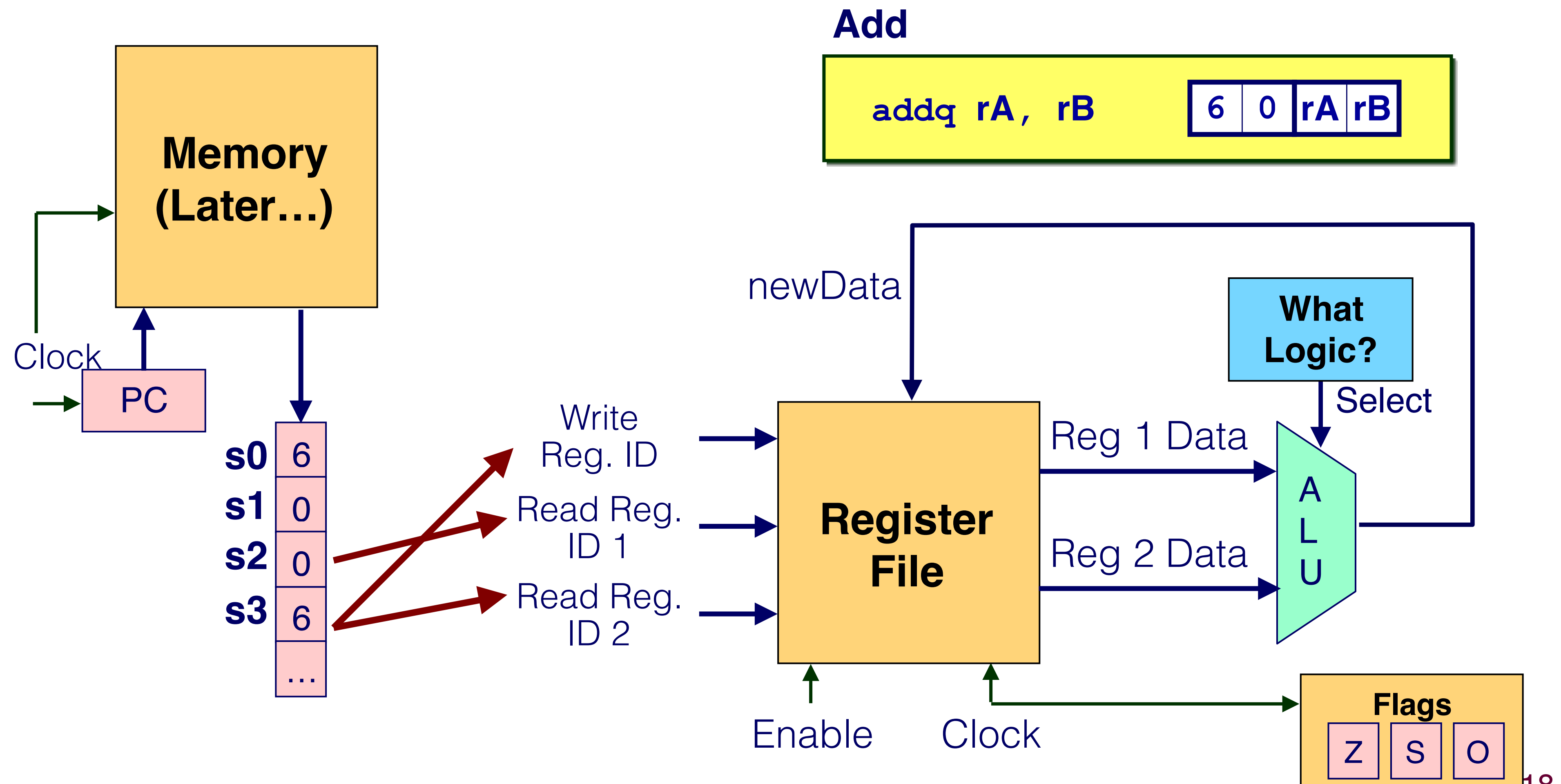
Executing an ADD instruction

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



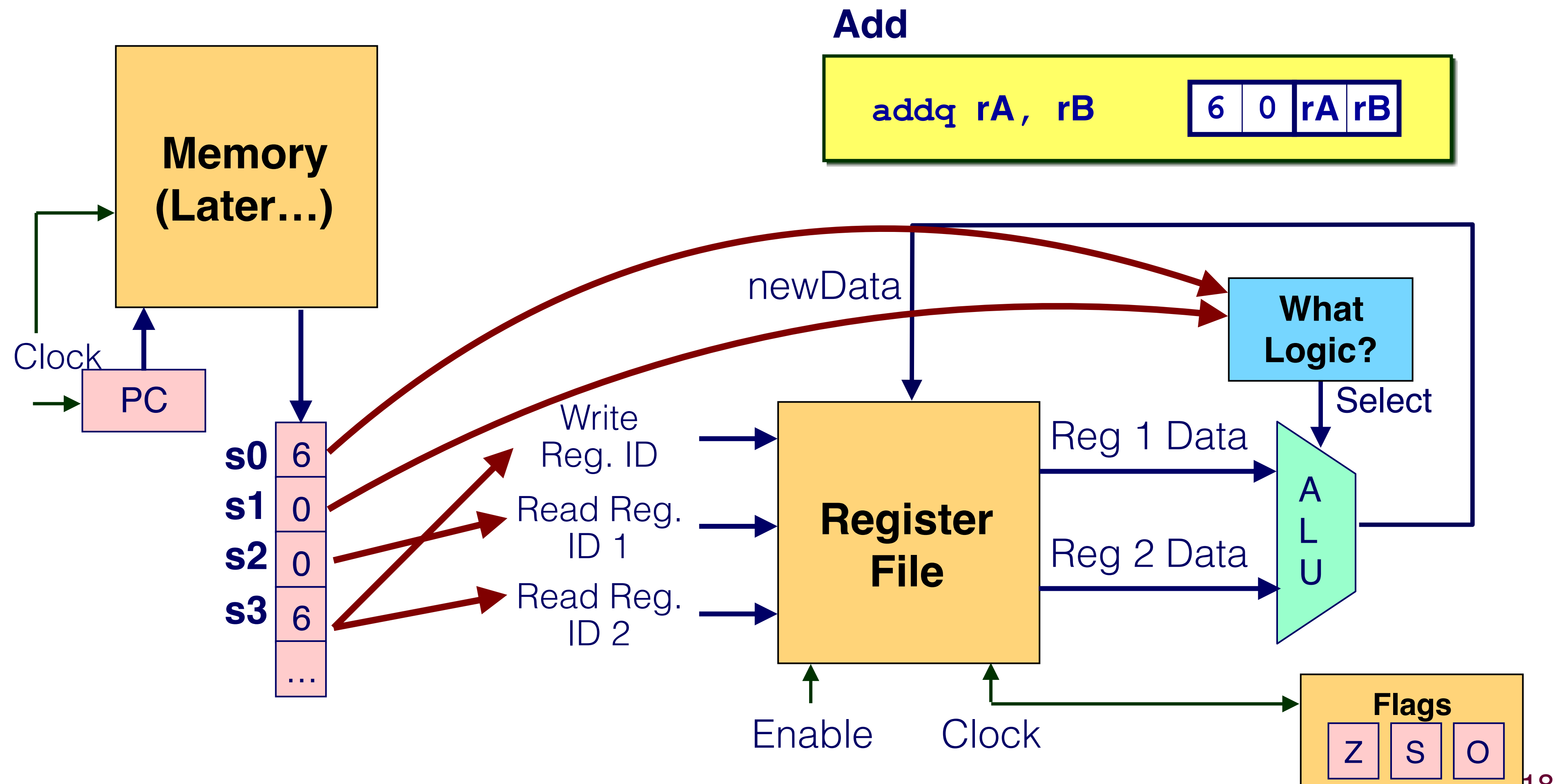
Executing an ADD instruction

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



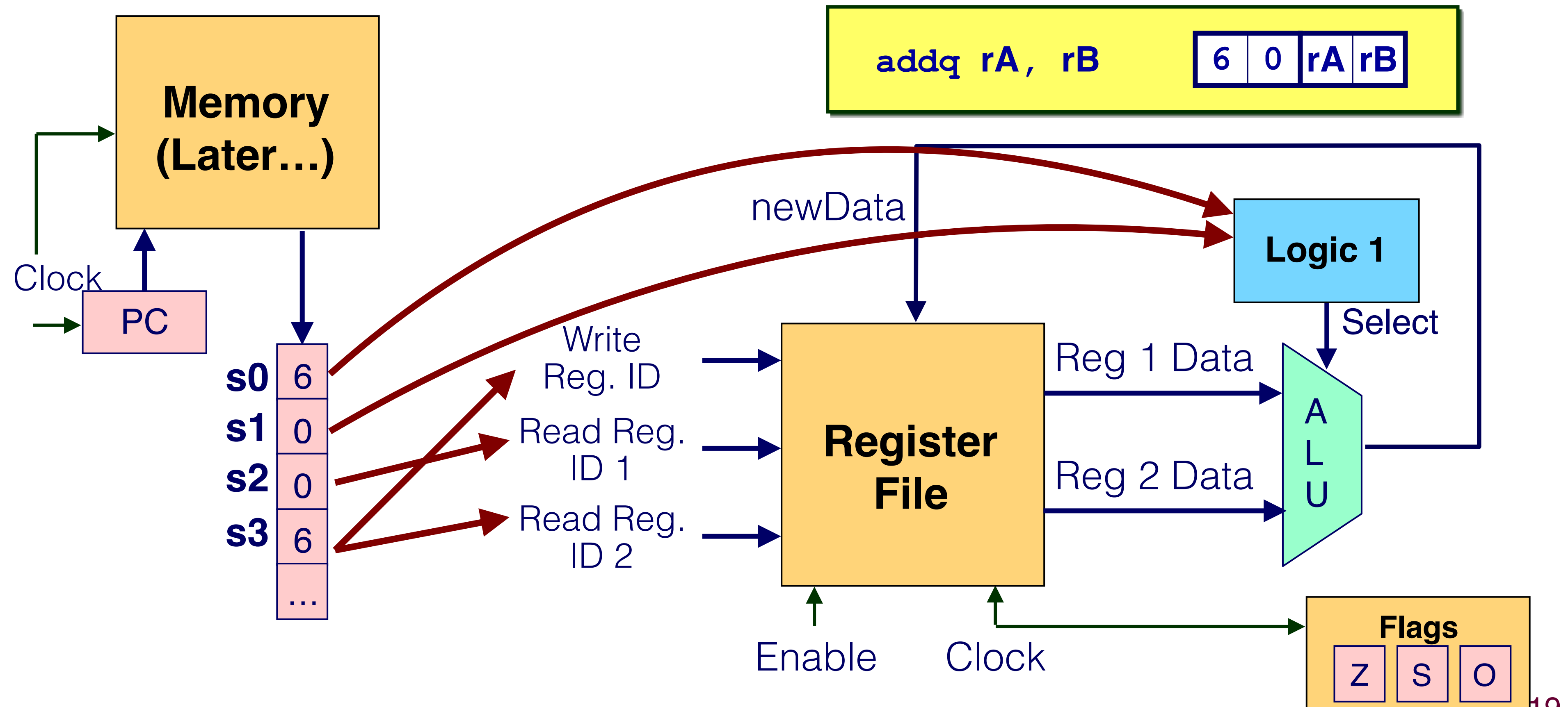
Executing an ADD instruction

- 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;

Memory (Later…)

Clock

PC

s0 6
s1 0
s2 0
s3 6
…

`addq rA, rB` | 6 | 0 | rA | rB |

newData

Logic 1

Select

Write Reg. ID

Read Reg. ID 1

Read Reg. ID 2

Register File

Reg 1 Data

Reg 2 Data

A L U

Logic 2

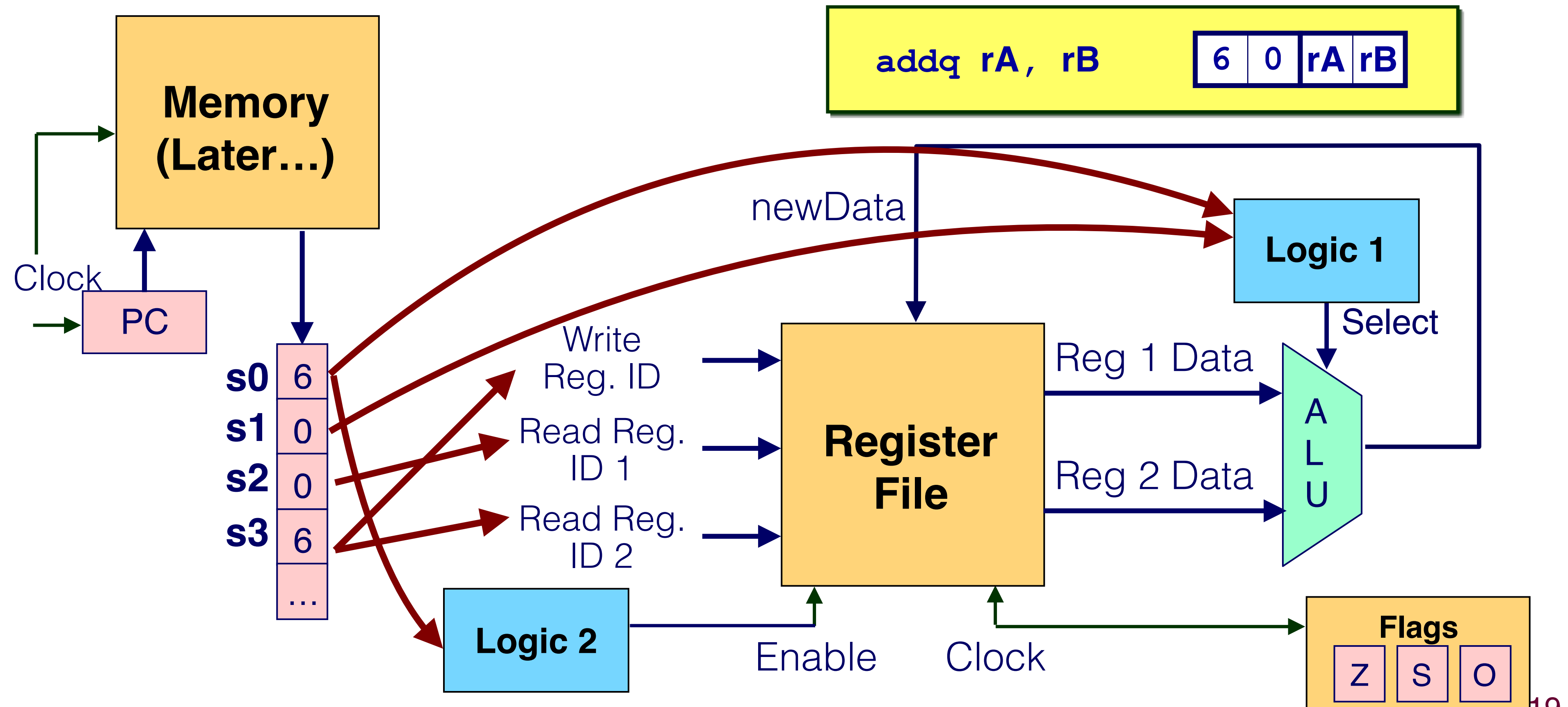
Enable

Clock

Flags
Z S O

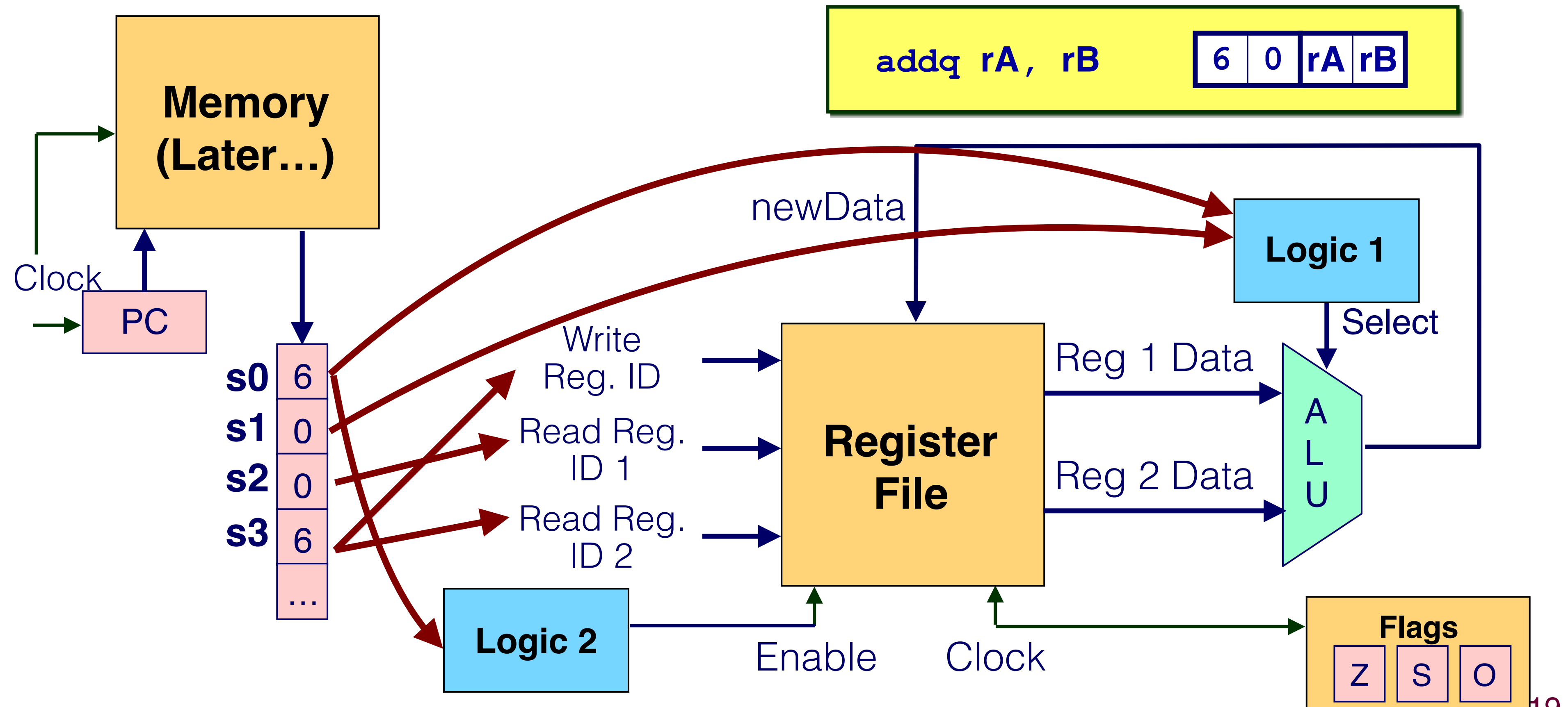
Executing an ADD instruction

- 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;

addq rA, rB

6	0	rA	rB

Memory (Later…)

Clock

PC

newData

Logic 1

Select

s0 6
s1 0
s2 0
s3 6
…

Logic 3

Write Reg. ID

Read Reg. ID 1

Read Reg. ID 2

Register File

Reg 1 Data

Reg 2 Data

A L U

Logic 2

Enable

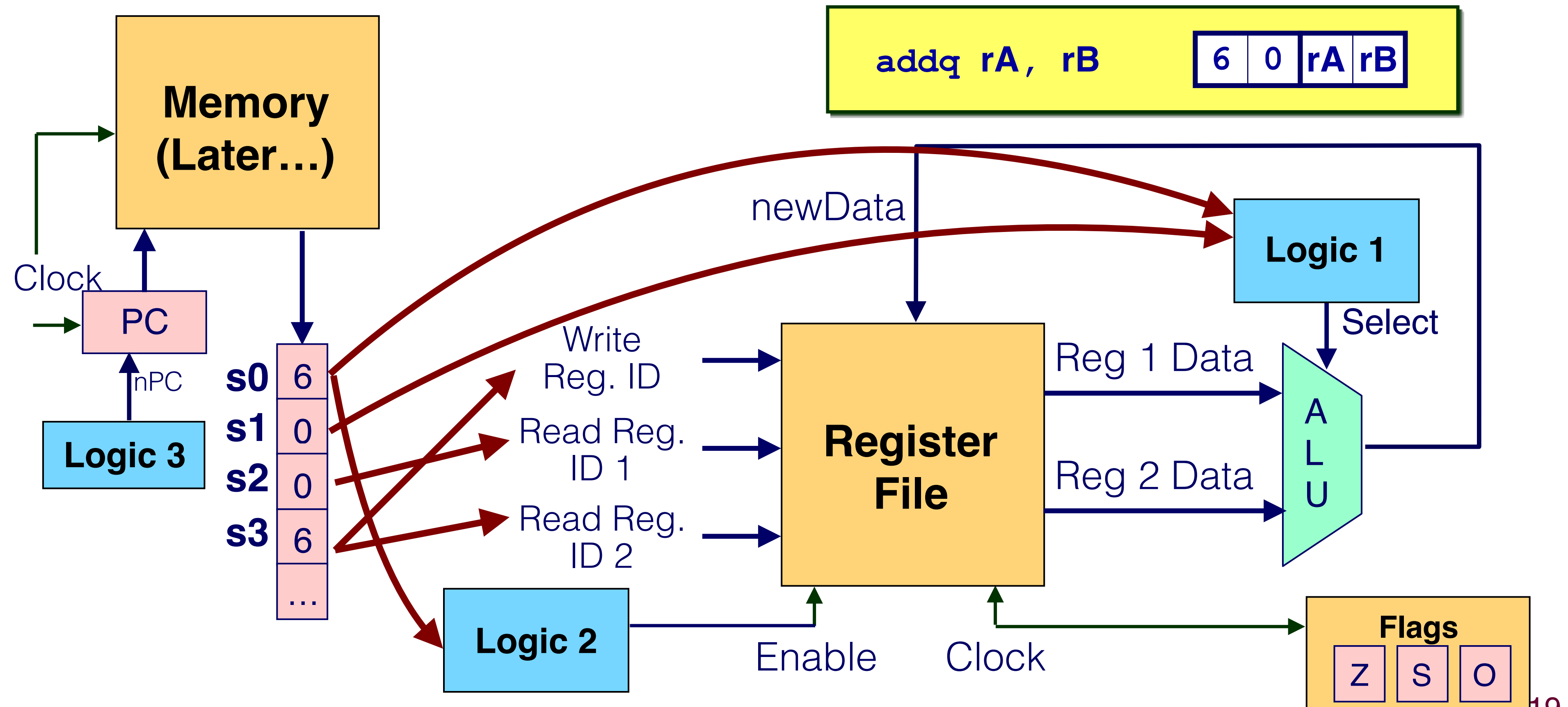
Clock

Flags

Z S O

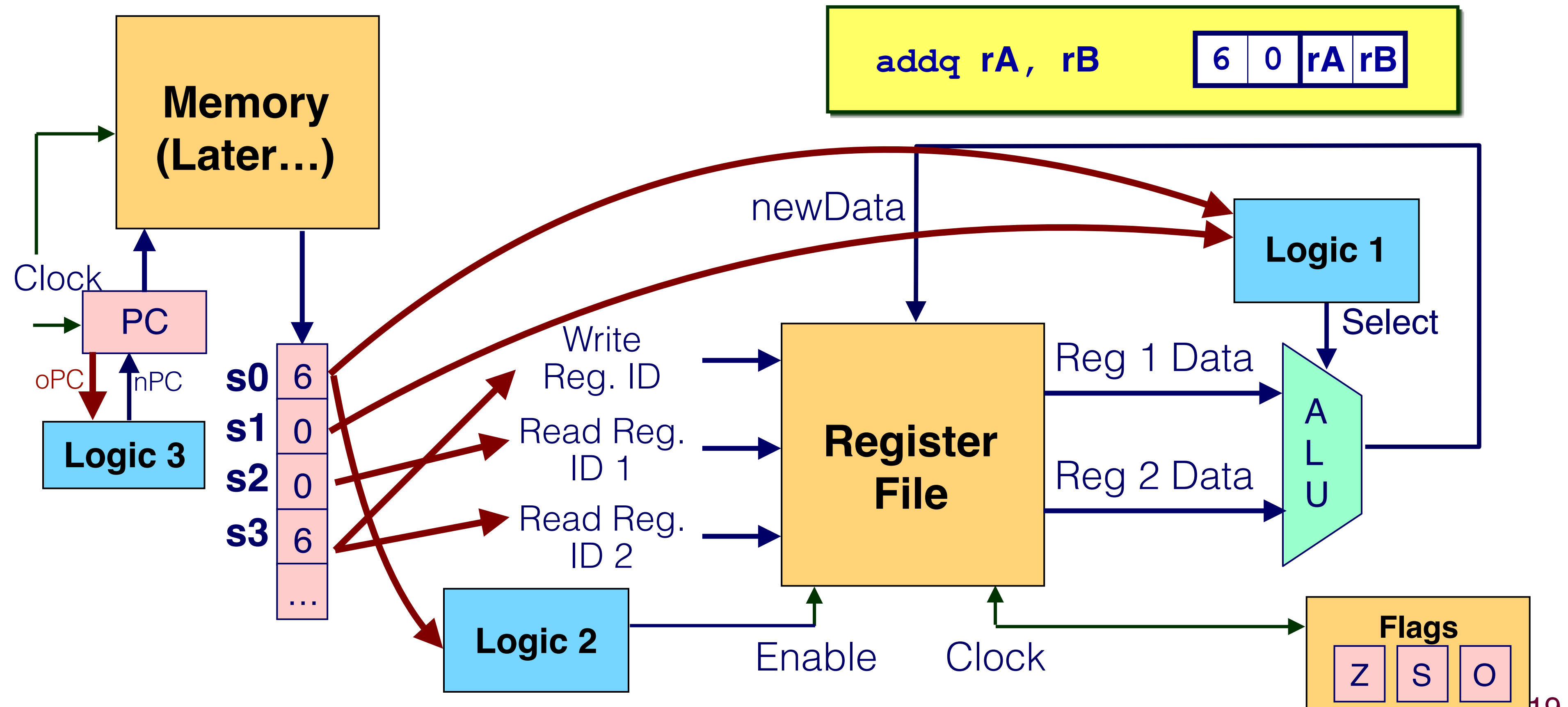
Executing an ADD instruction

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



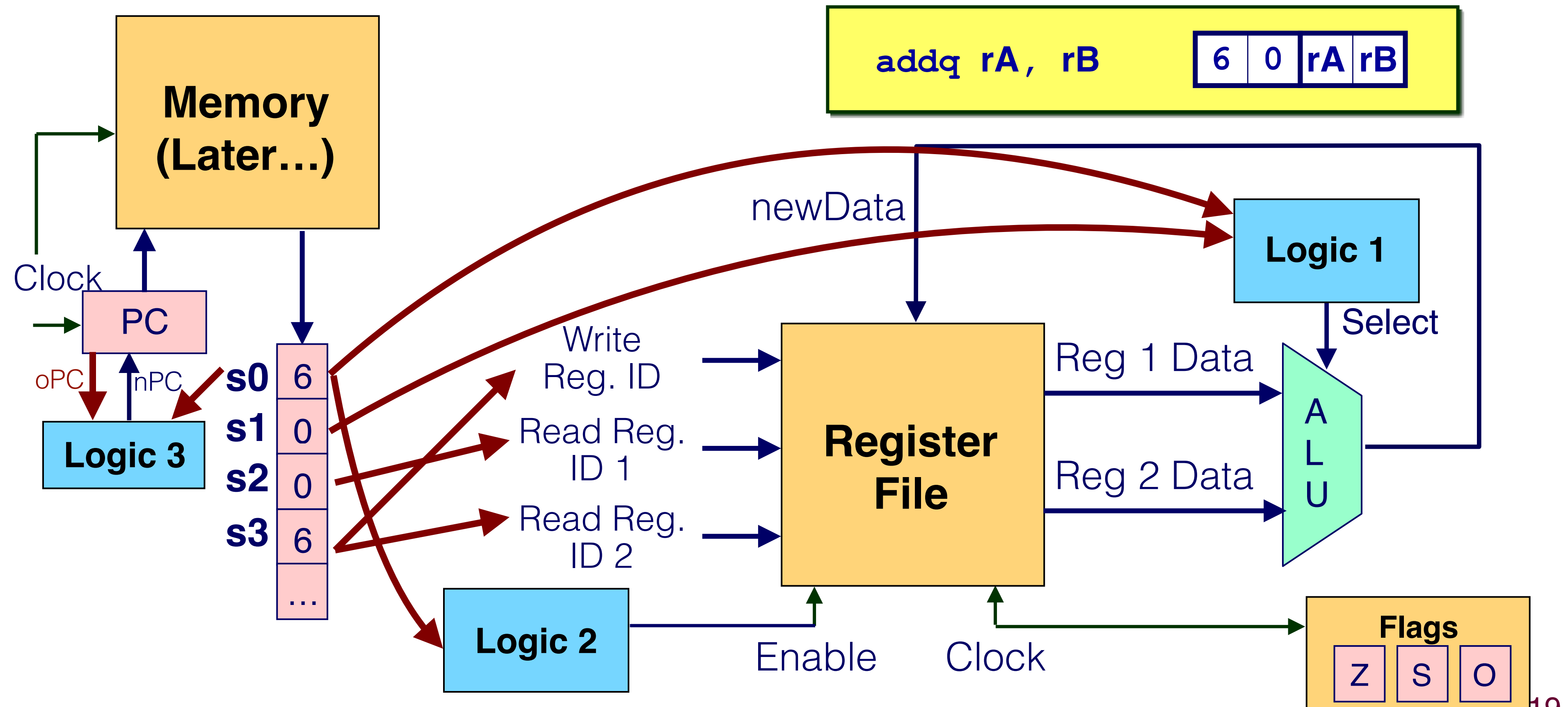
Executing an ADD instruction

- 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;

`addq rA, rB`

6	0	rA	rB

Memory (Later...)

Clock

PC

oPC

nPC

Logic 3

s0 6

s1 0

s2 0

s3 6

...

newData

Logic 1

Select

Write Reg. ID

Read Reg. ID 1

Read Reg. ID 2

Register File

Reg 1 Data

Reg 2 Data

ALU

Logic 2

Enable

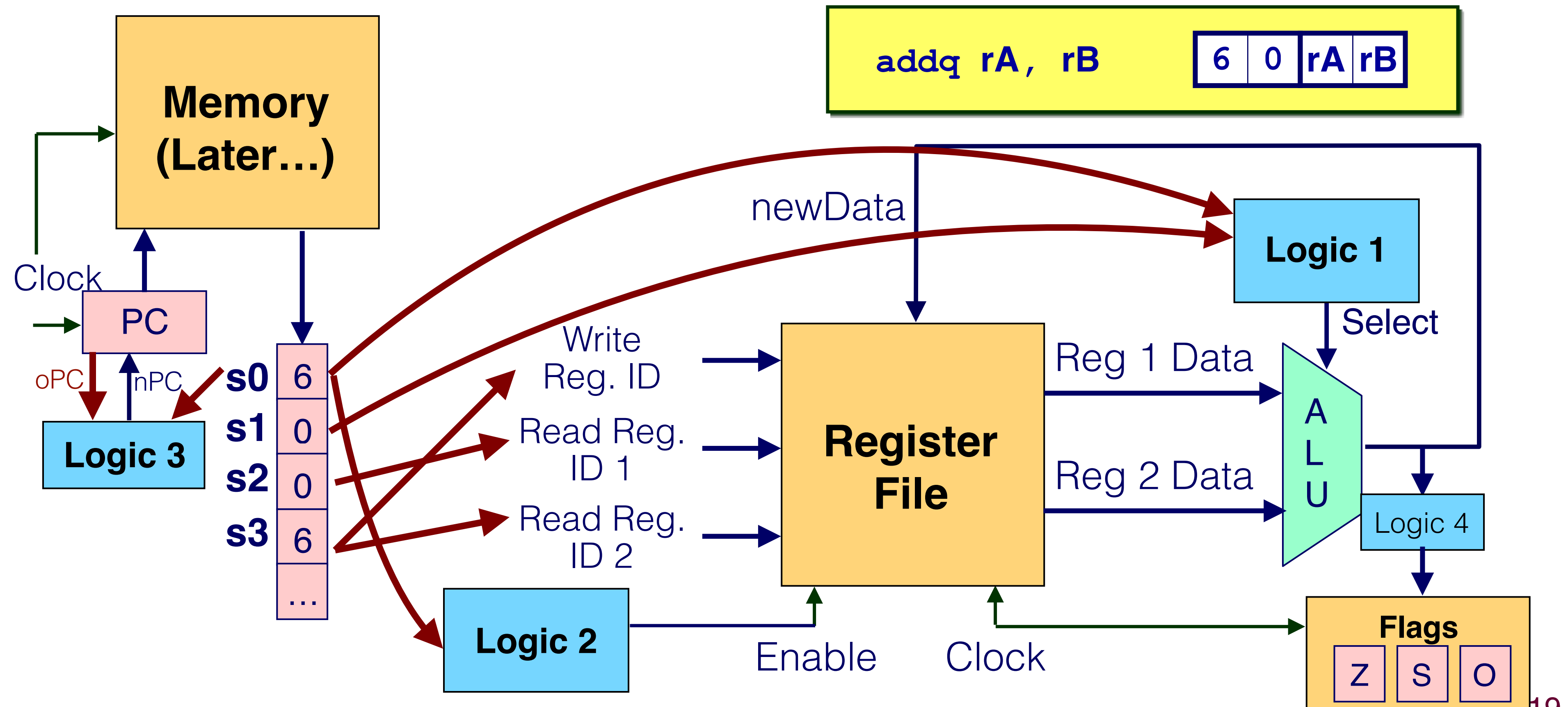
Clock

Flags

Z S O

Executing an ADD instruction

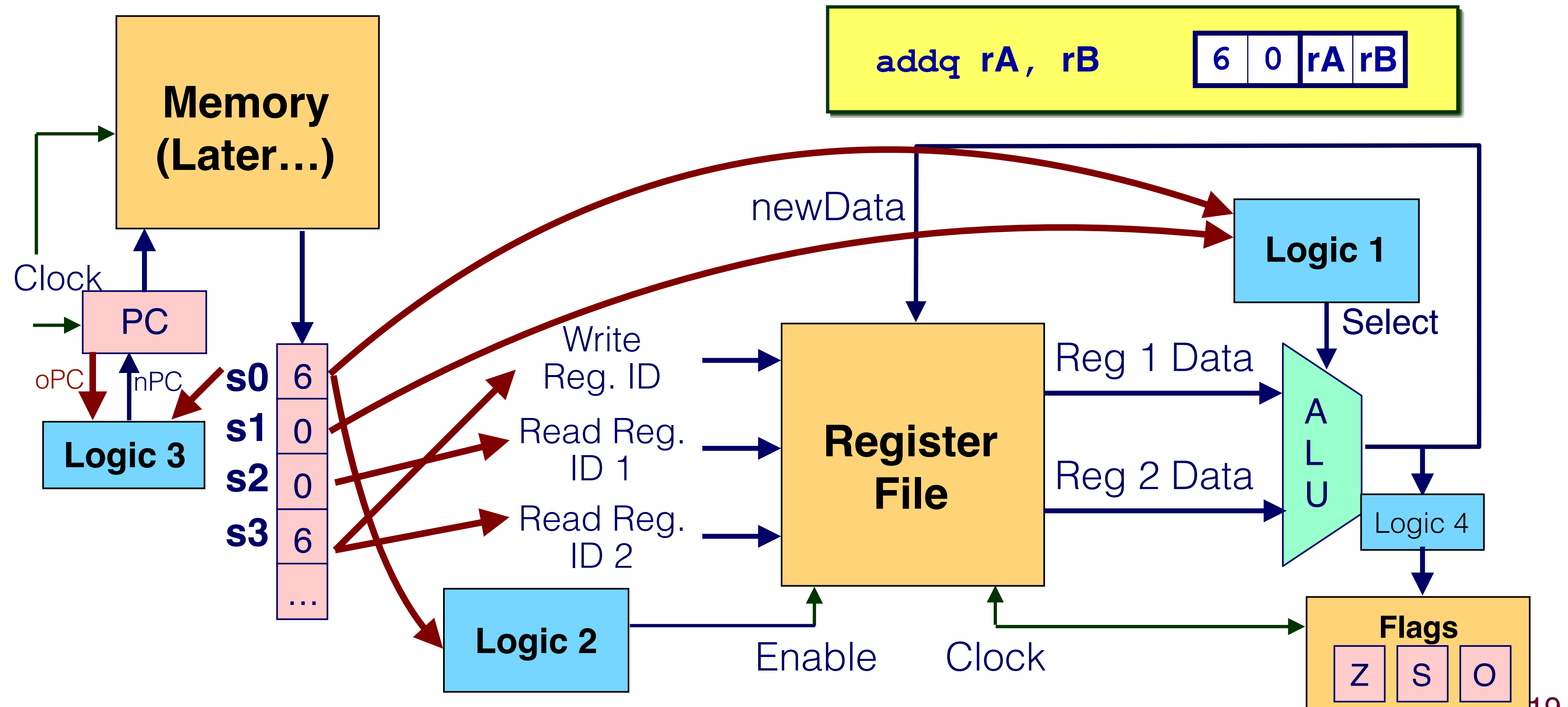
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?



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?

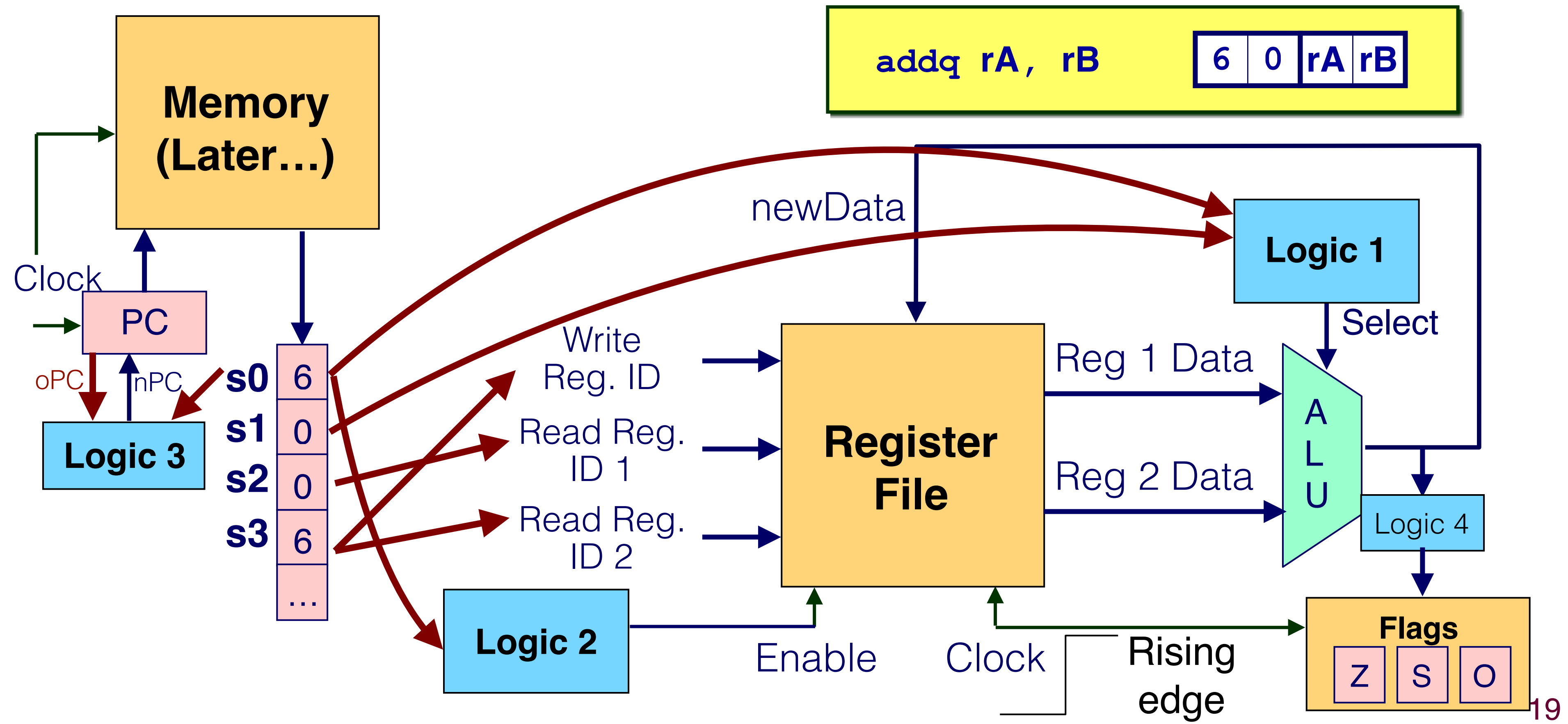
How do these logics get implemented?



Executing an ADD instruction

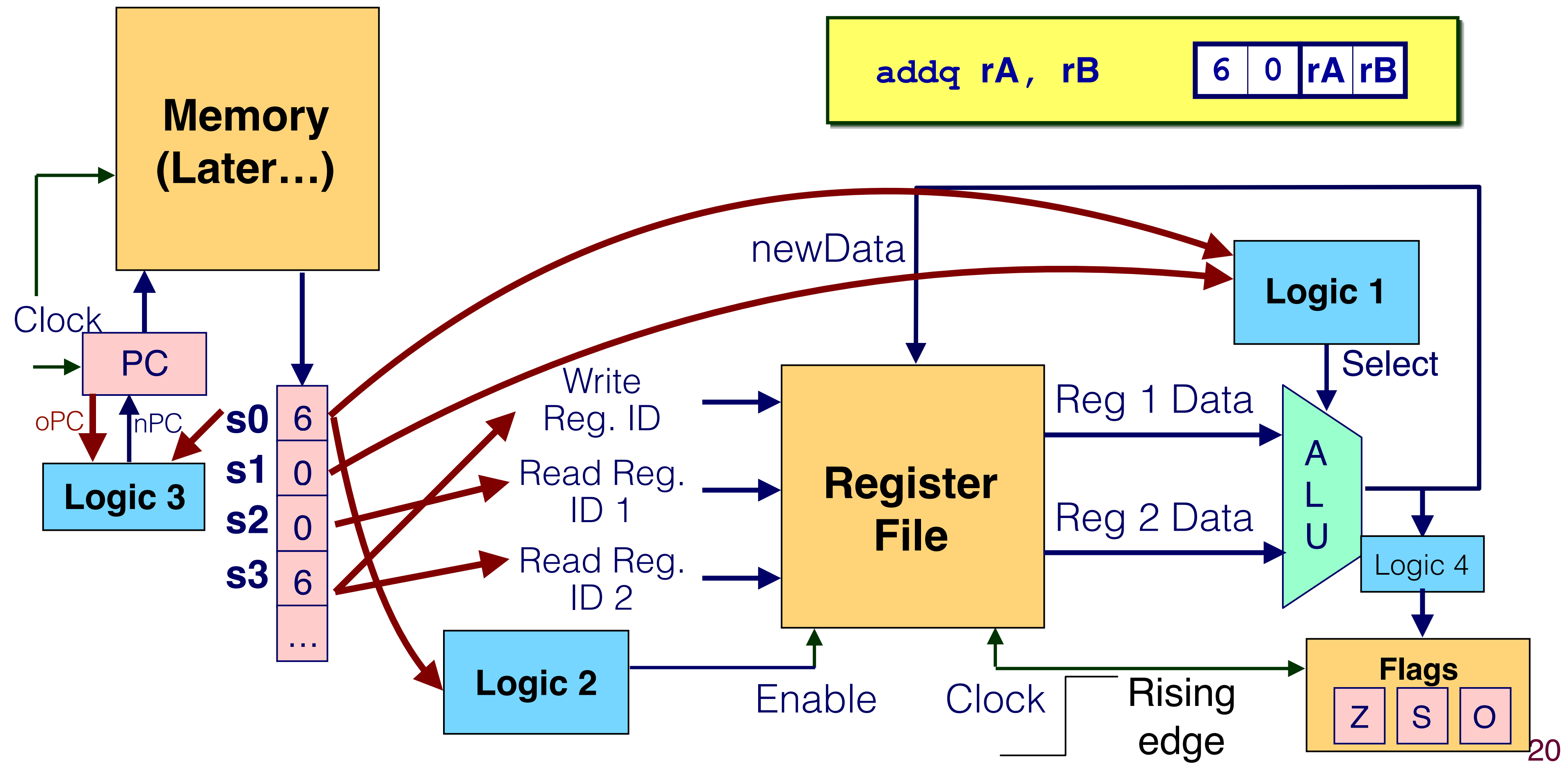
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

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.

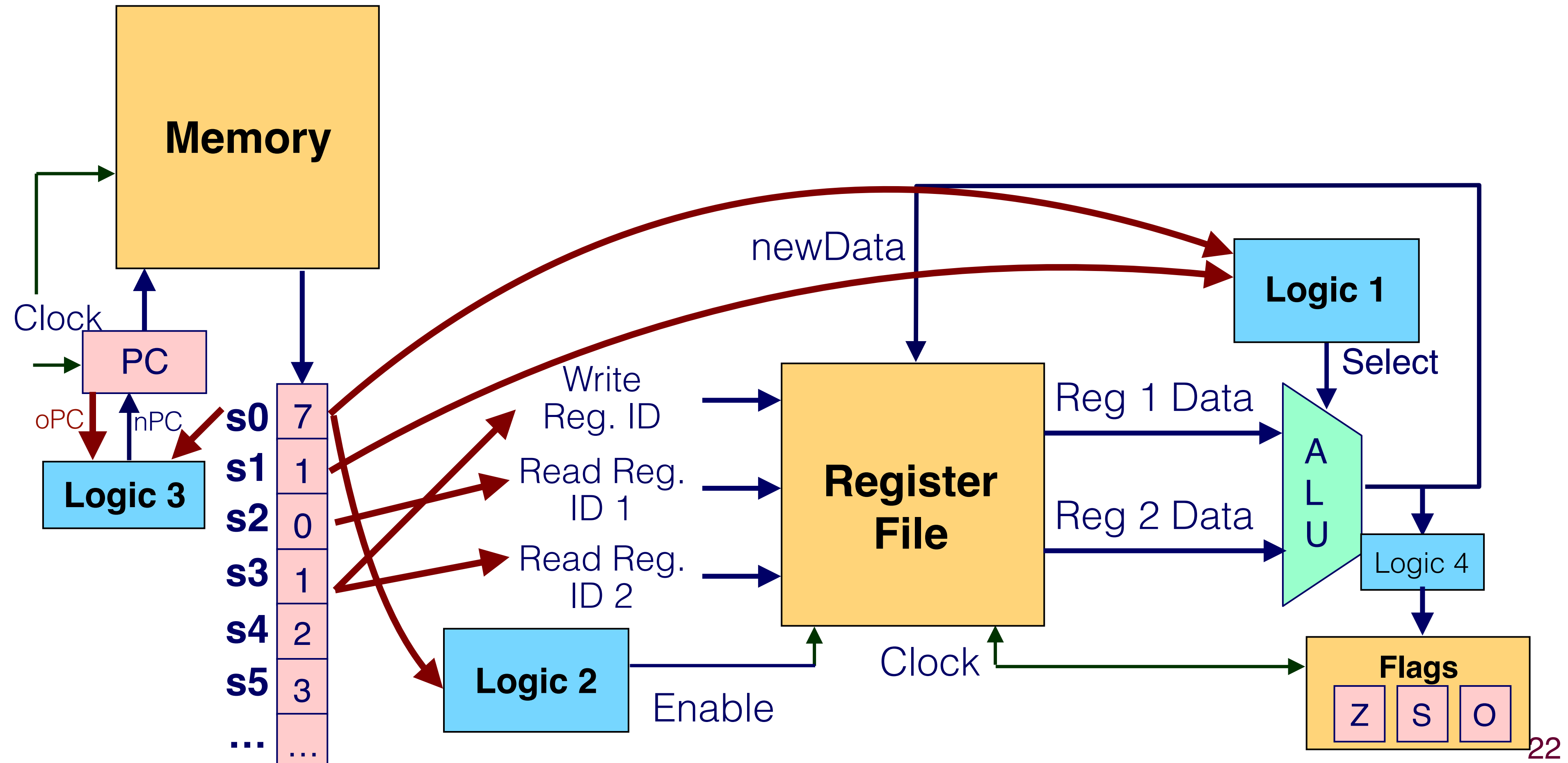


Executing a JLE instruction

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

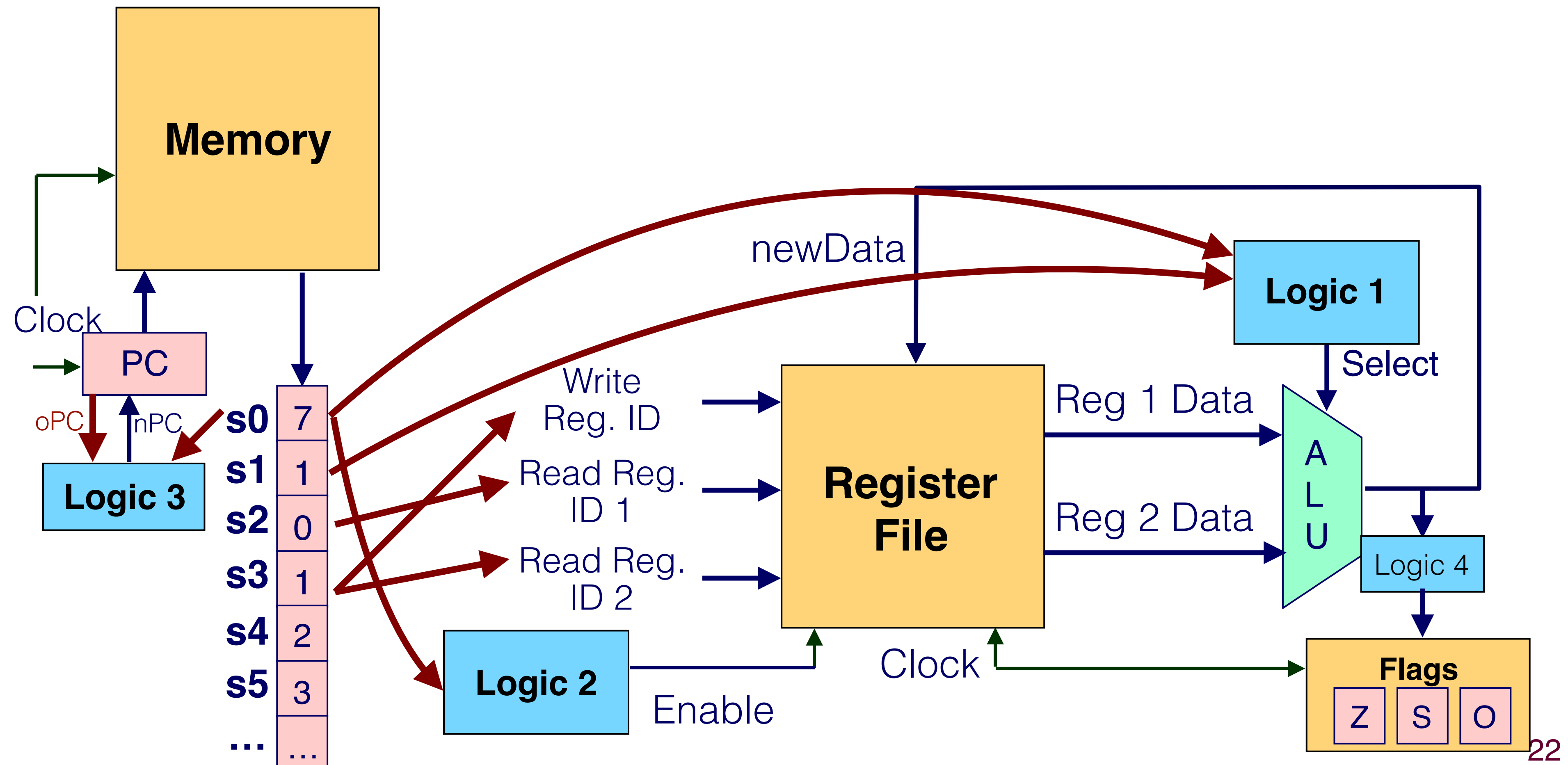
Memory
Clock
PC
oPC
nPC
Logic 3
s0 7
s1 1
s2 0
s3 1
s4 2
s5 3
… …
newData
Logic 1
Select
Write
Reg. ID
Read Reg.
ID 1
Read Reg.
ID 2
Register
File
Reg 1 Data
Reg 2 Data
A
L
U
Logic 4
Logic 2
Enable
Clock
Rising
edge
Flags
Z
S
O

Executing a JLE instruction



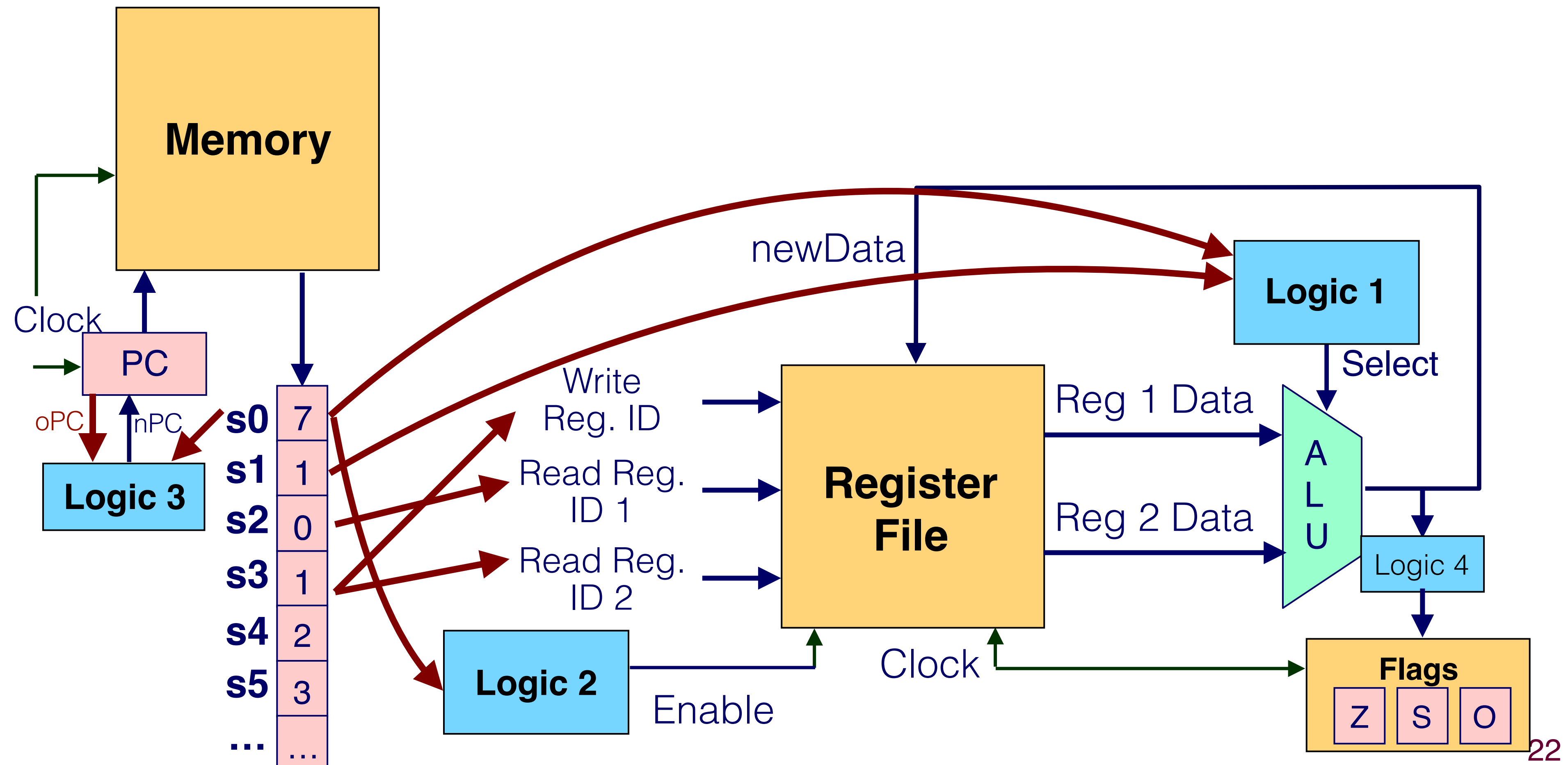
Executing a JLE instruction

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



Executing a JLE instruction

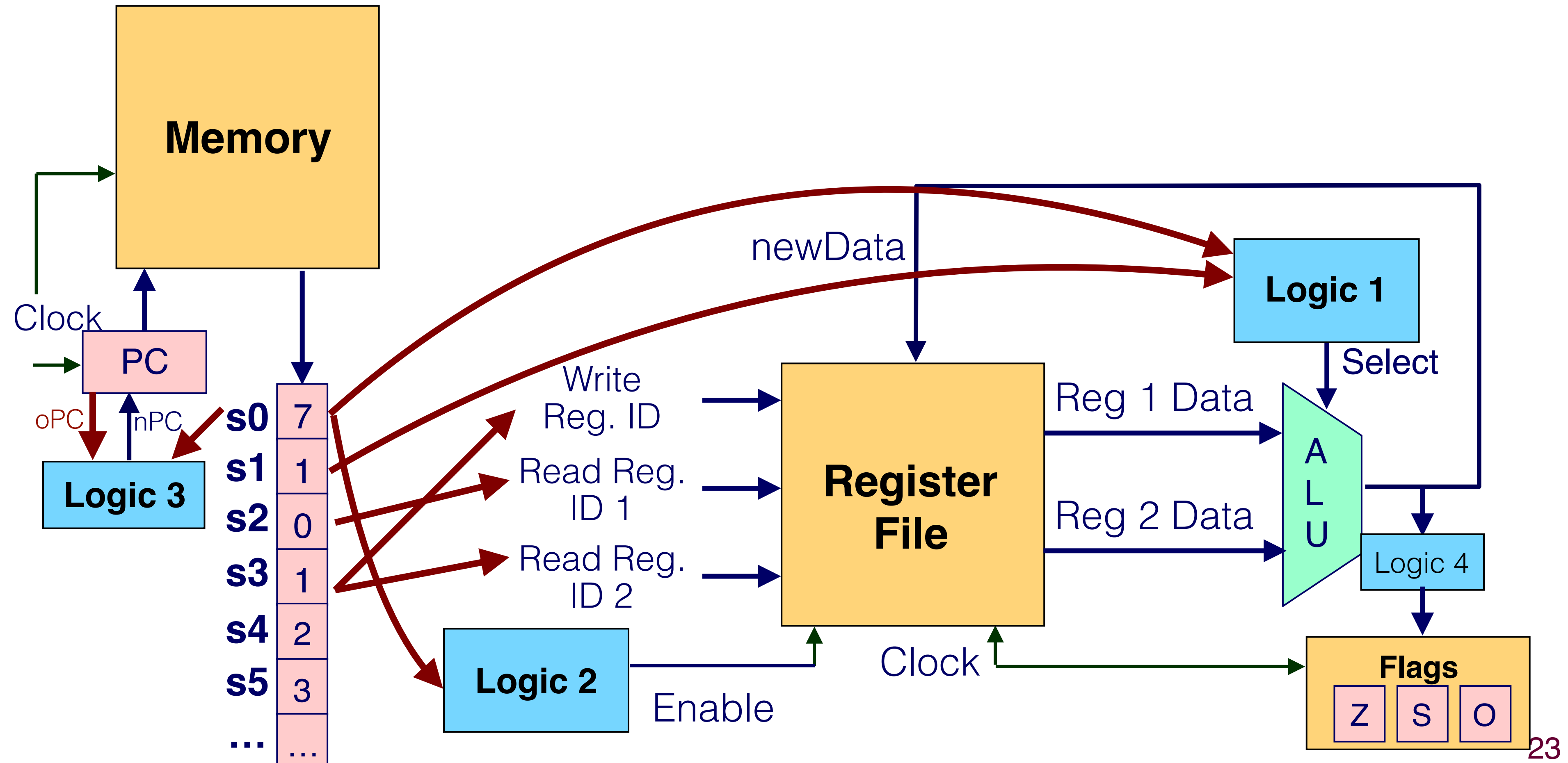
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;



```
jle Dest    7 | 1 | Dest
```

Executing a JLE instruction

- Logic 3??

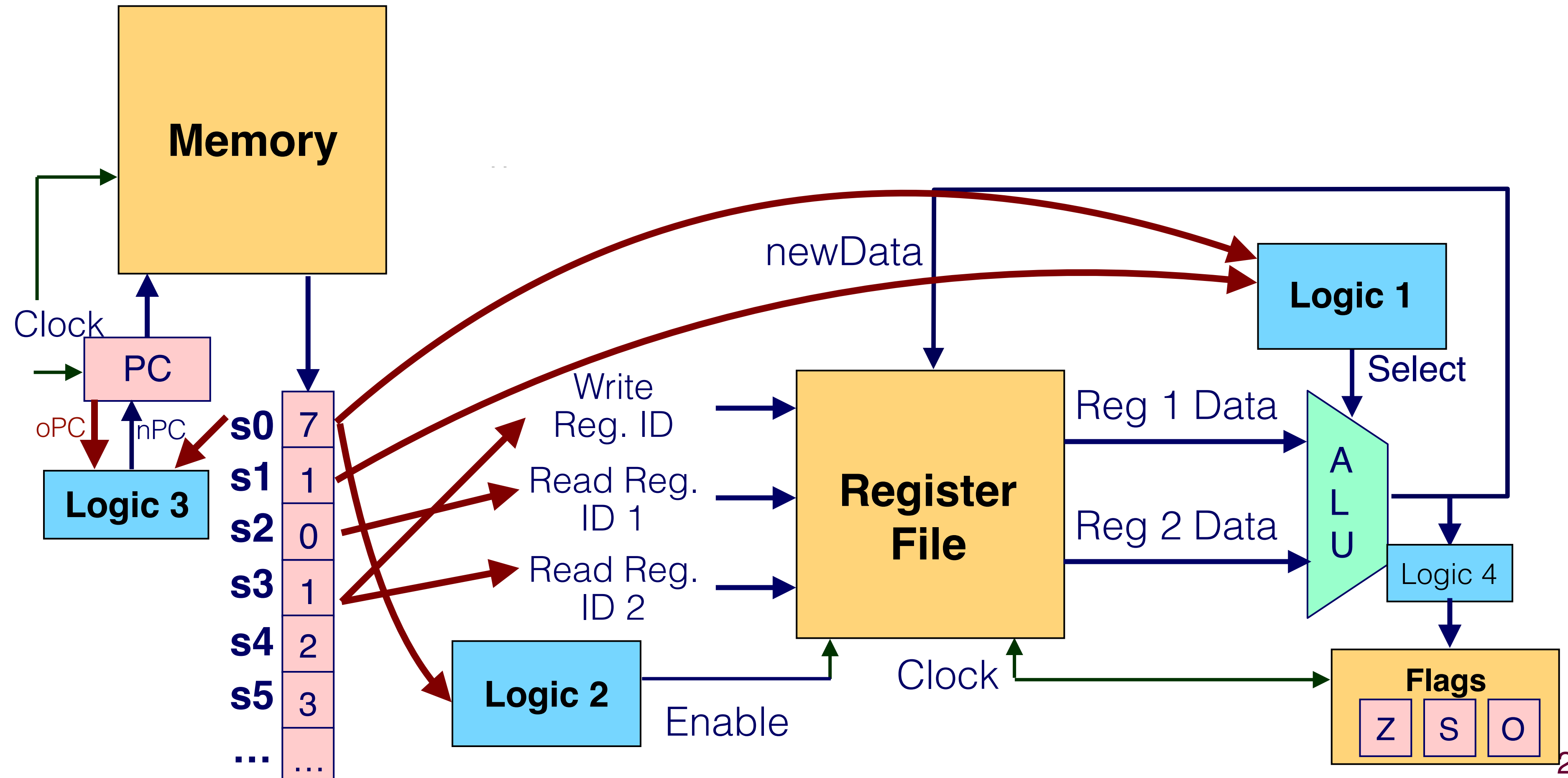


```
jle Dest | 7 | 1 | Dest
```

Executing a JLE instruction

- Logic 3??

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

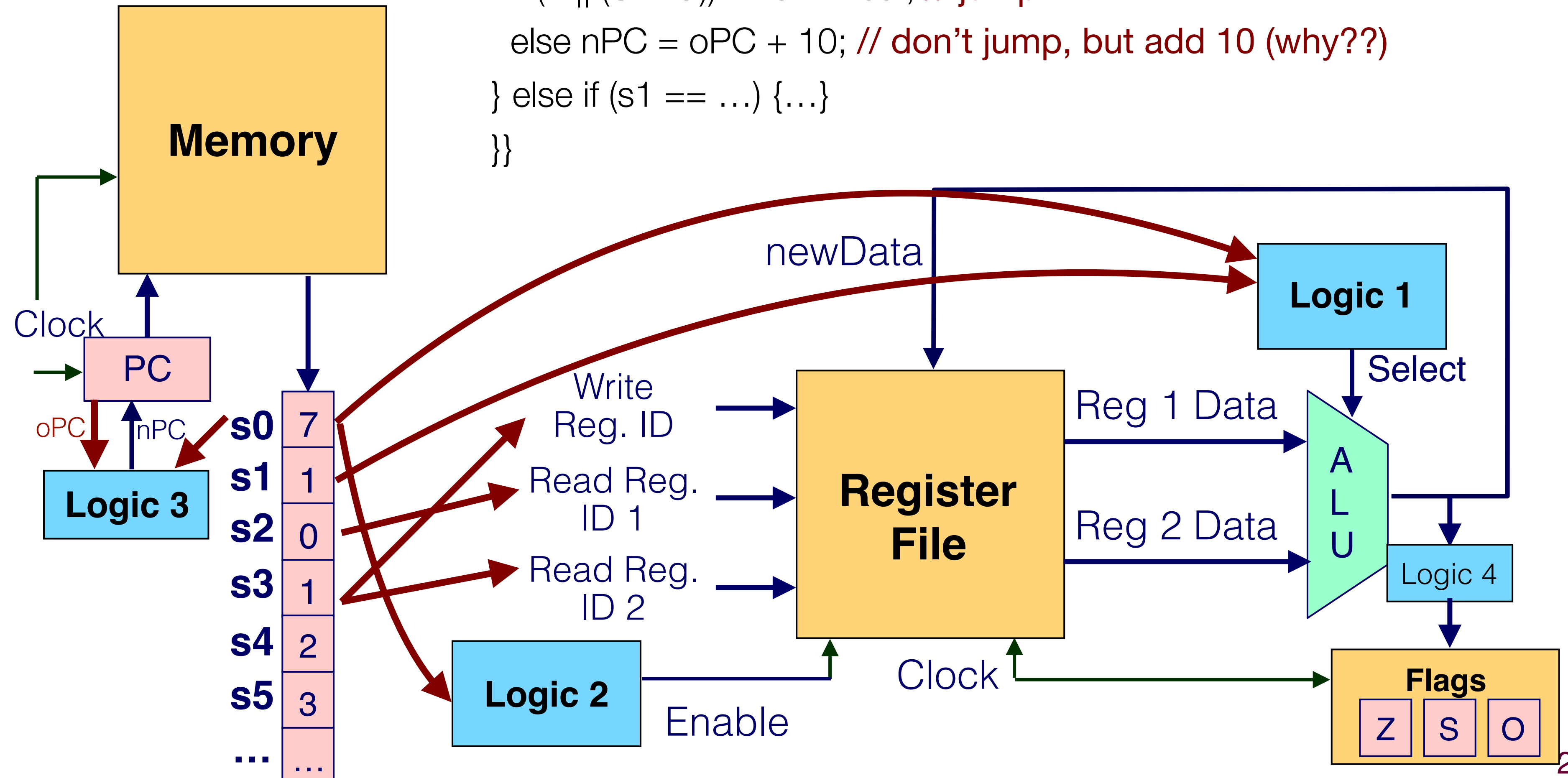


jle Dest | 7 | 1 | Dest

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 == …) {…}
}}
```

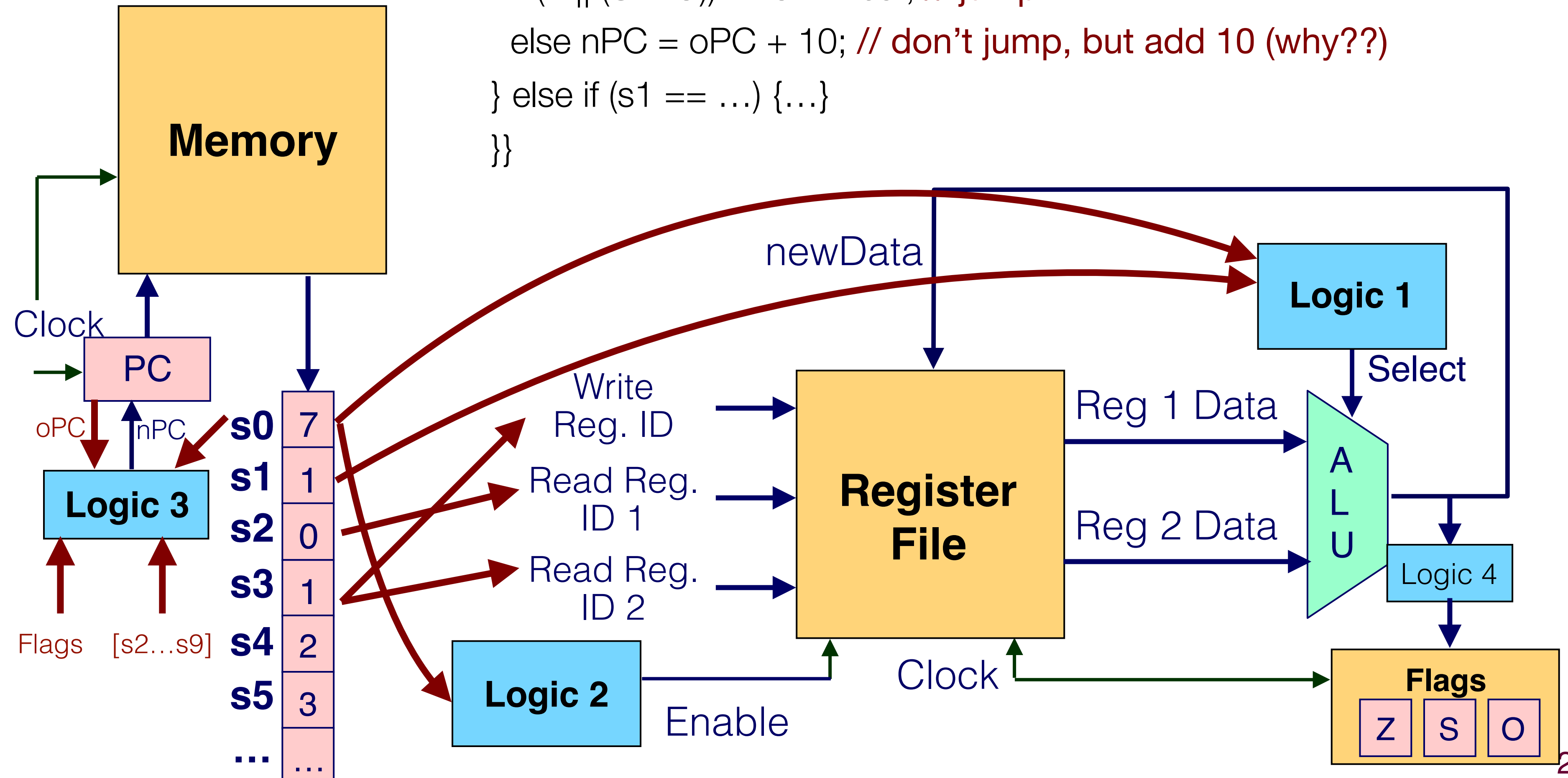


Executing a JLE instruction

jle Dest	7	1	Dest

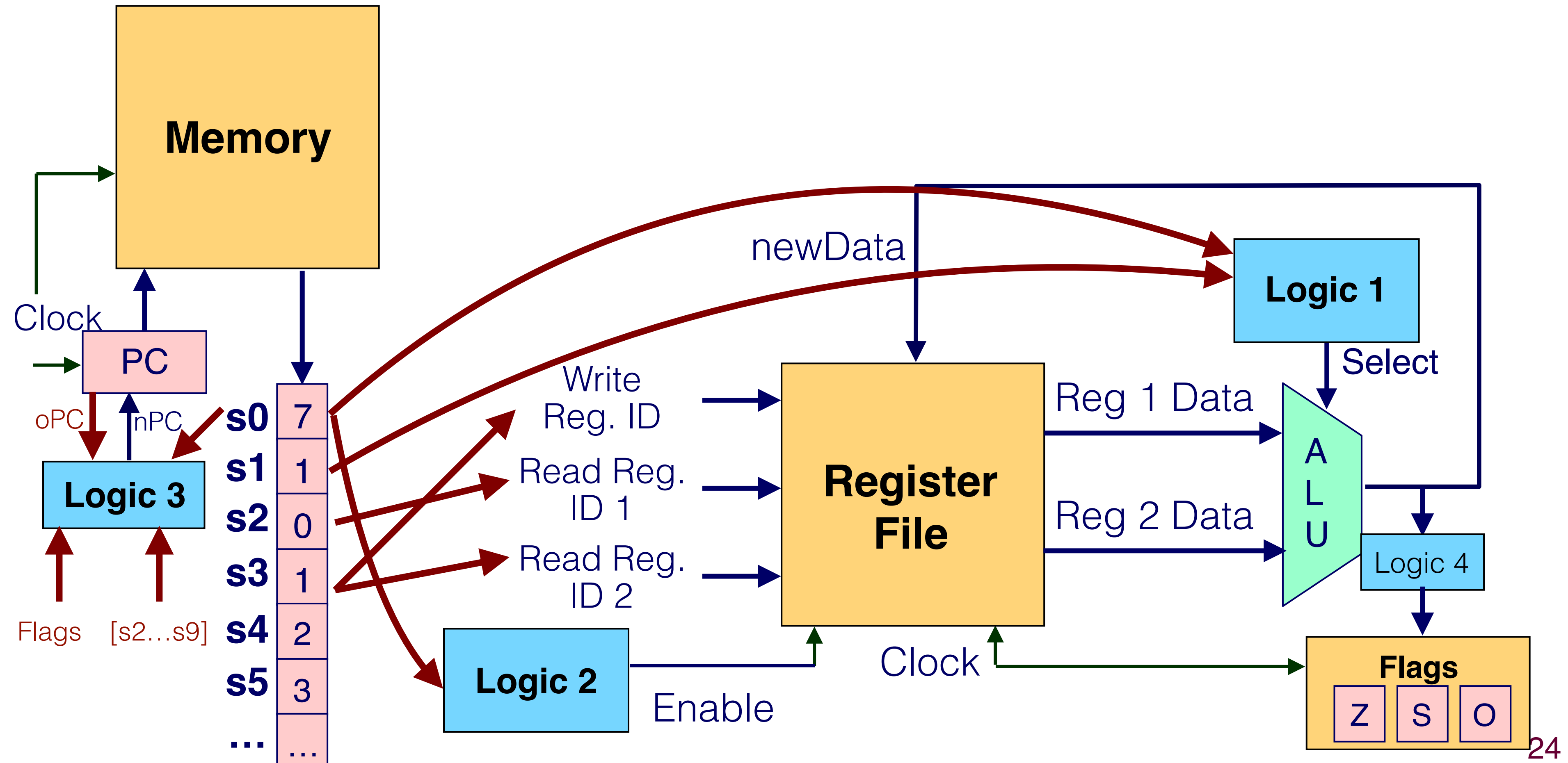
- Logic 3??

```
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 == …) {…}
}}
```



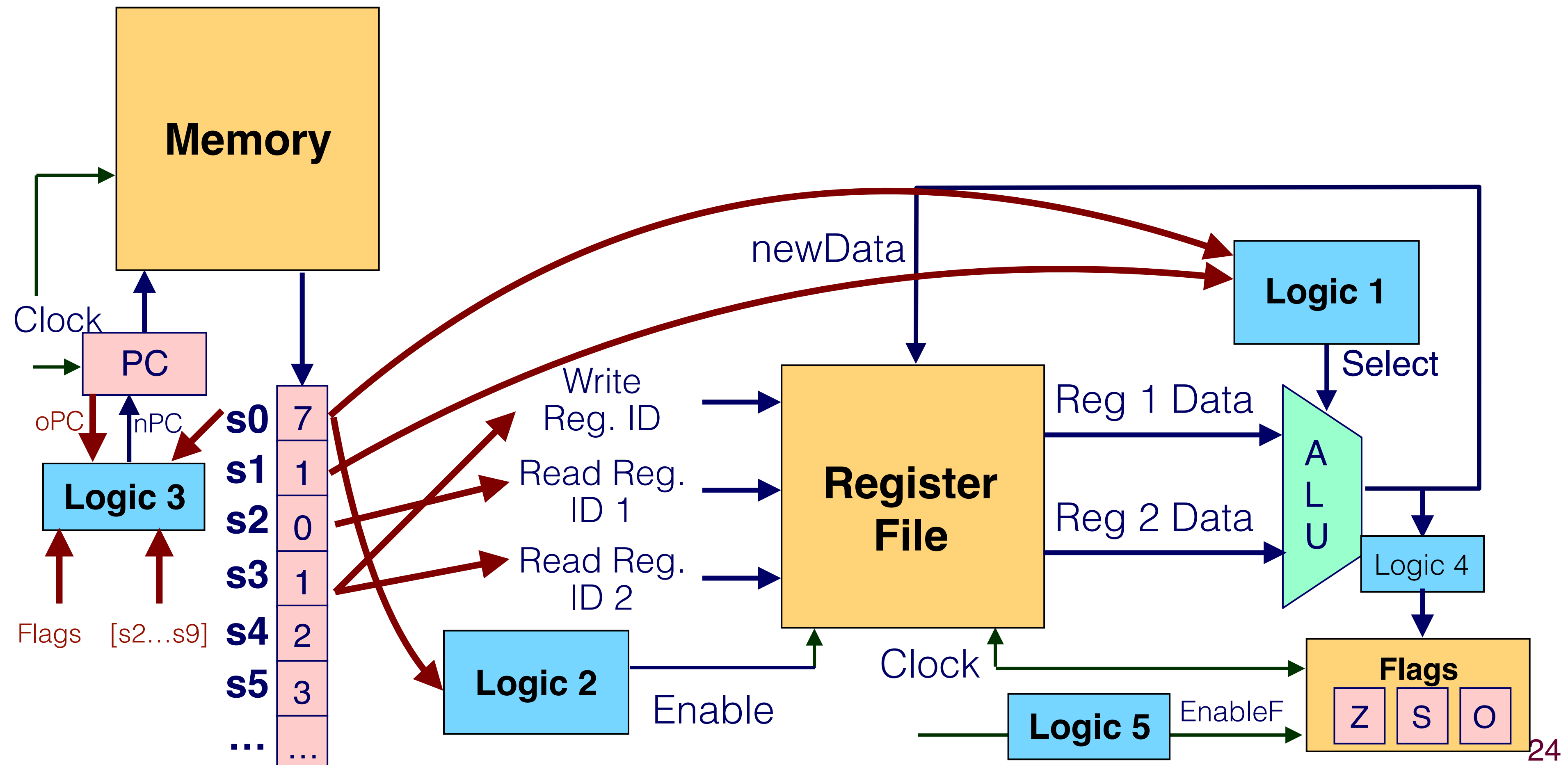
Executing a JLE instruction

- Logic 4? Does JLE write flags?



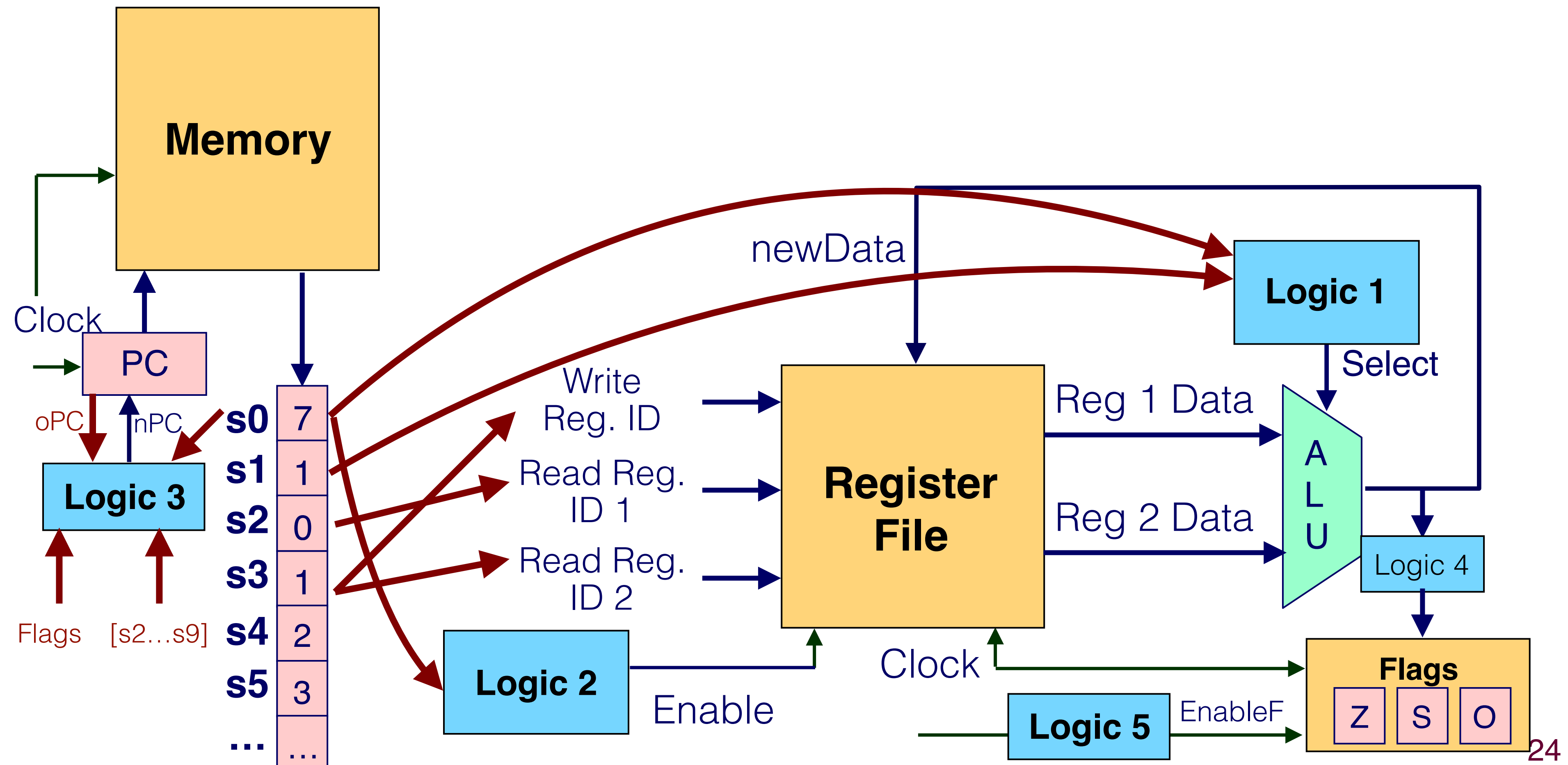
Executing a JLE instruction

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

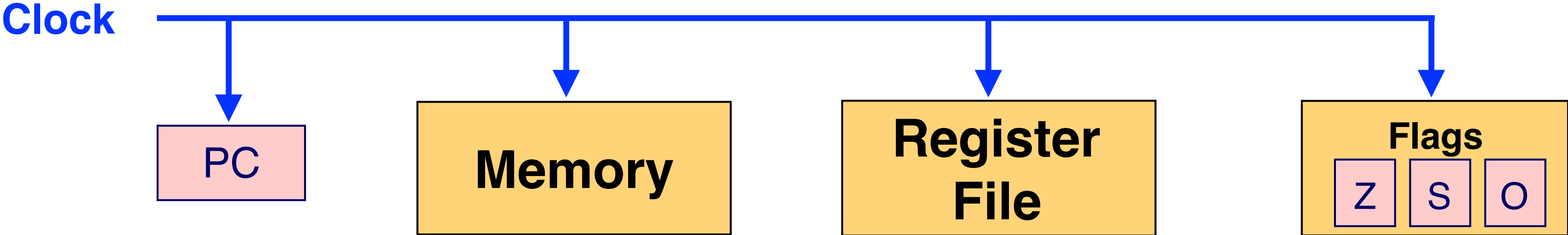


Executing a JLE instruction

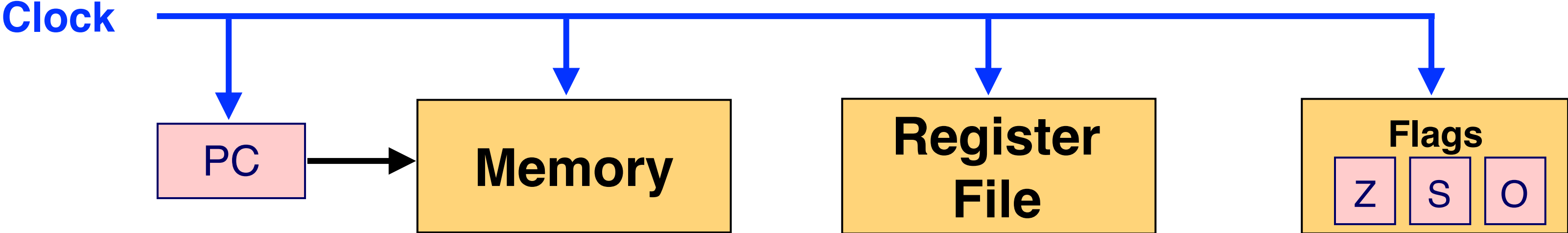
- Logic 4? Does JLE write flags?
- Need another piece of logic.
- Logic 5: if (s0 == 7) EnableF = 0; else if (s0 == 6) EnableF = 1;



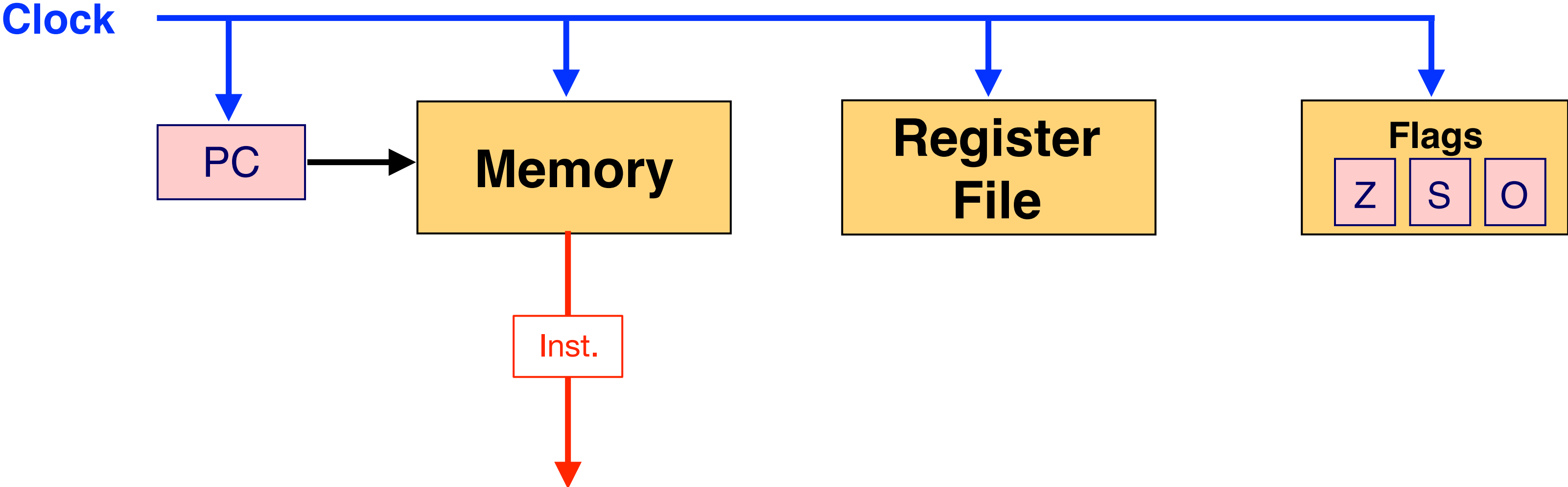
Microarchitecture (So far)



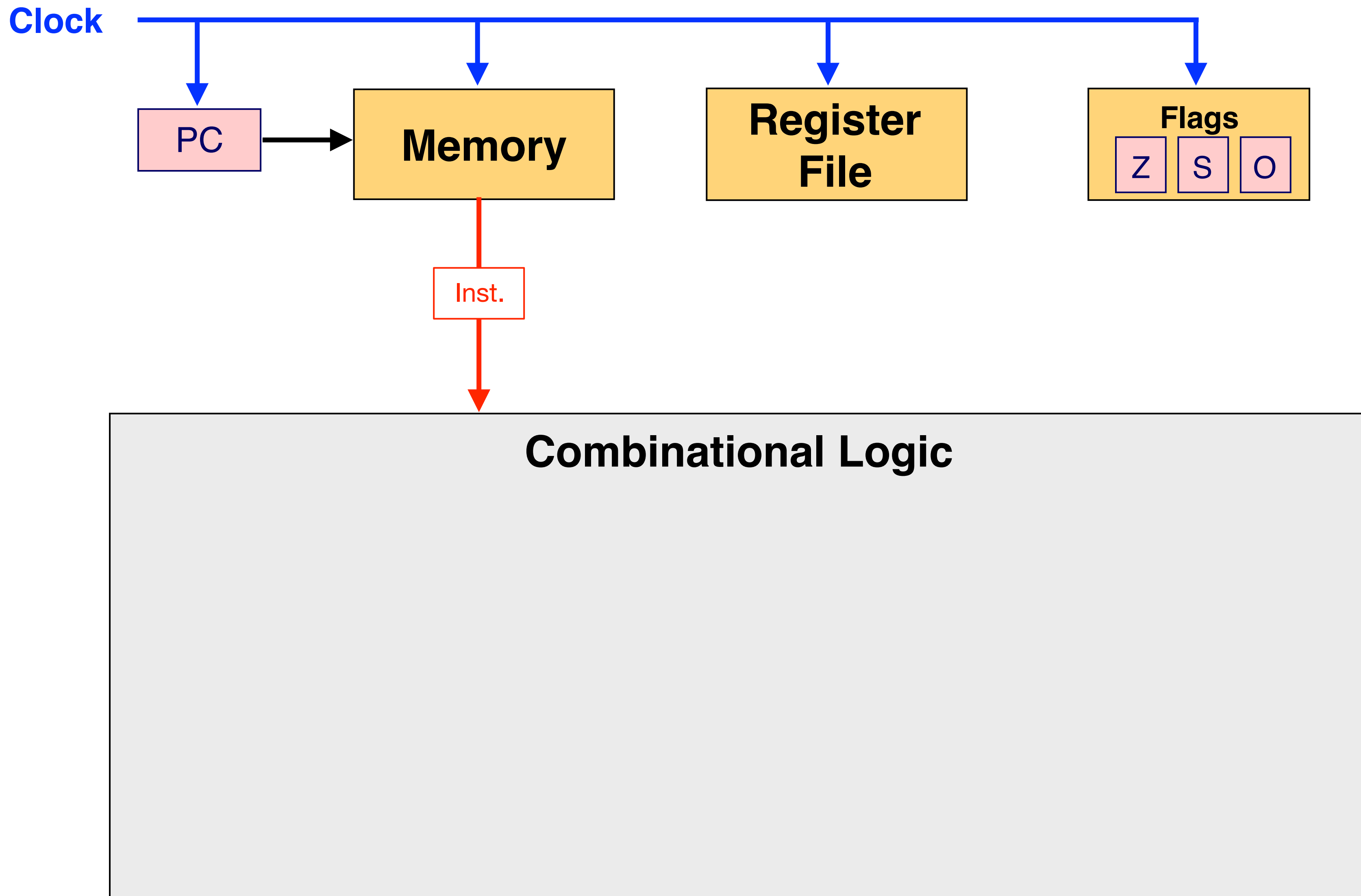
Microarchitecture (So far)



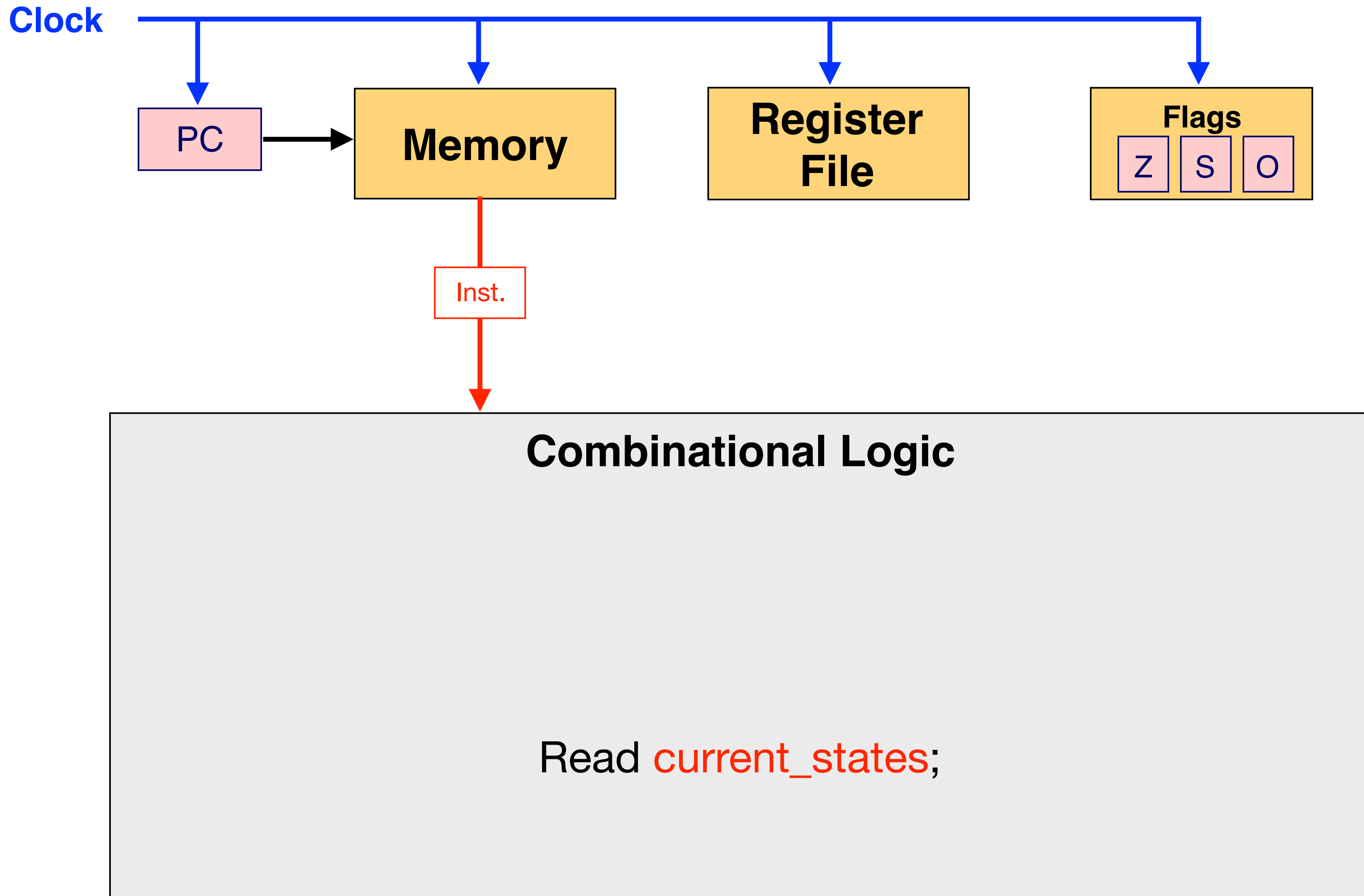
Microarchitecture (So far)



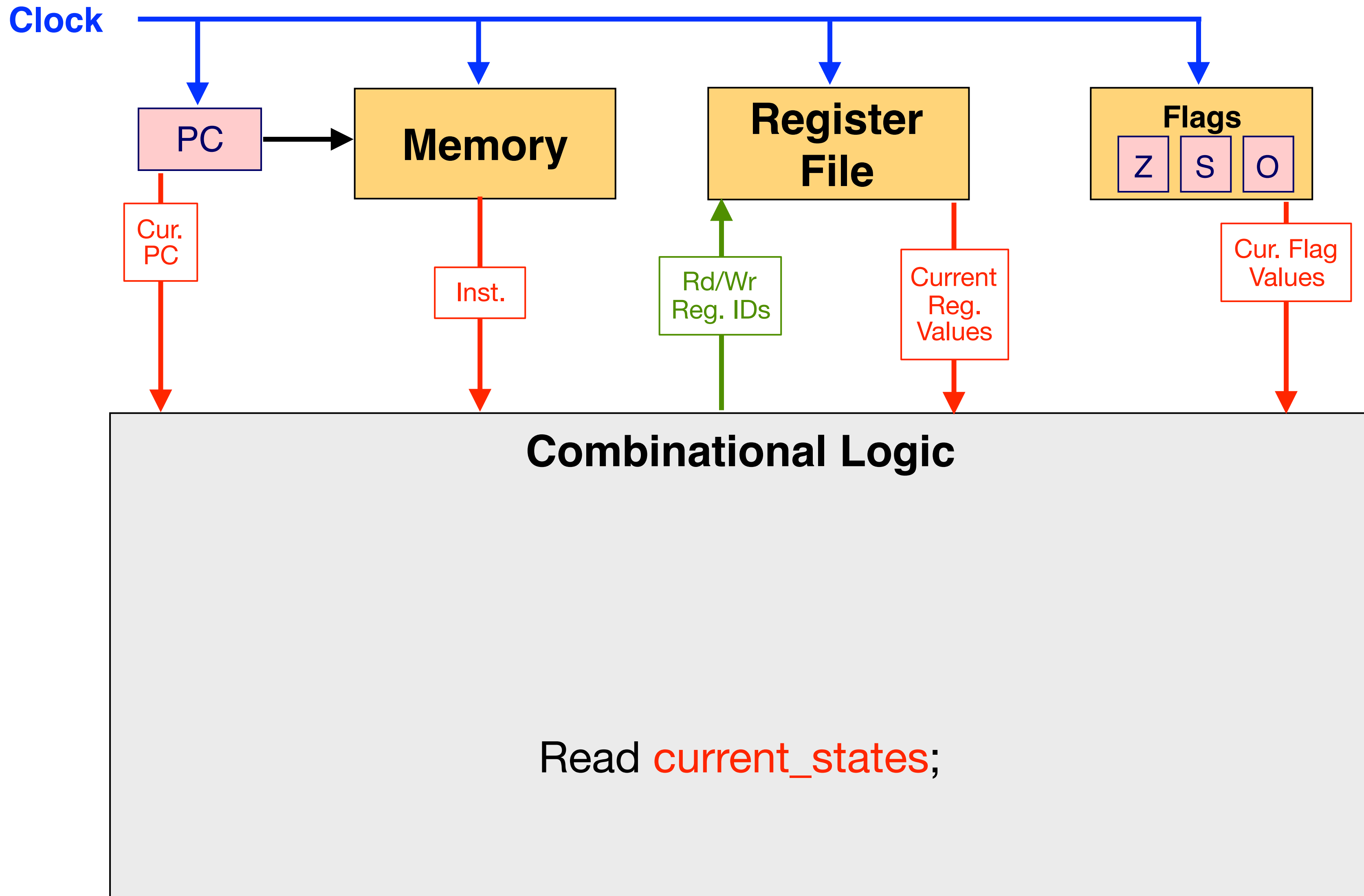
Microarchitecture (So far)



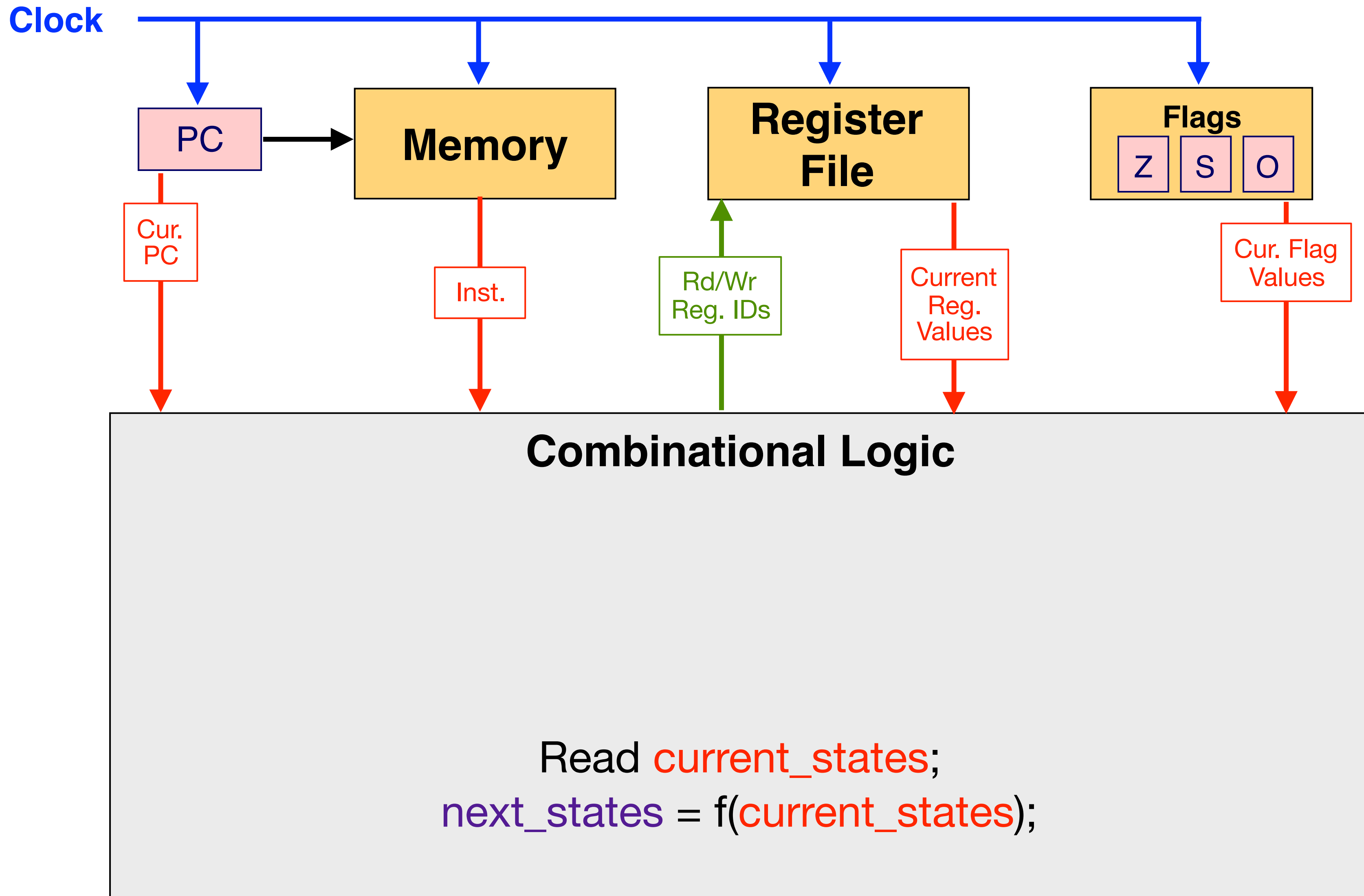
Microarchitecture (So far)



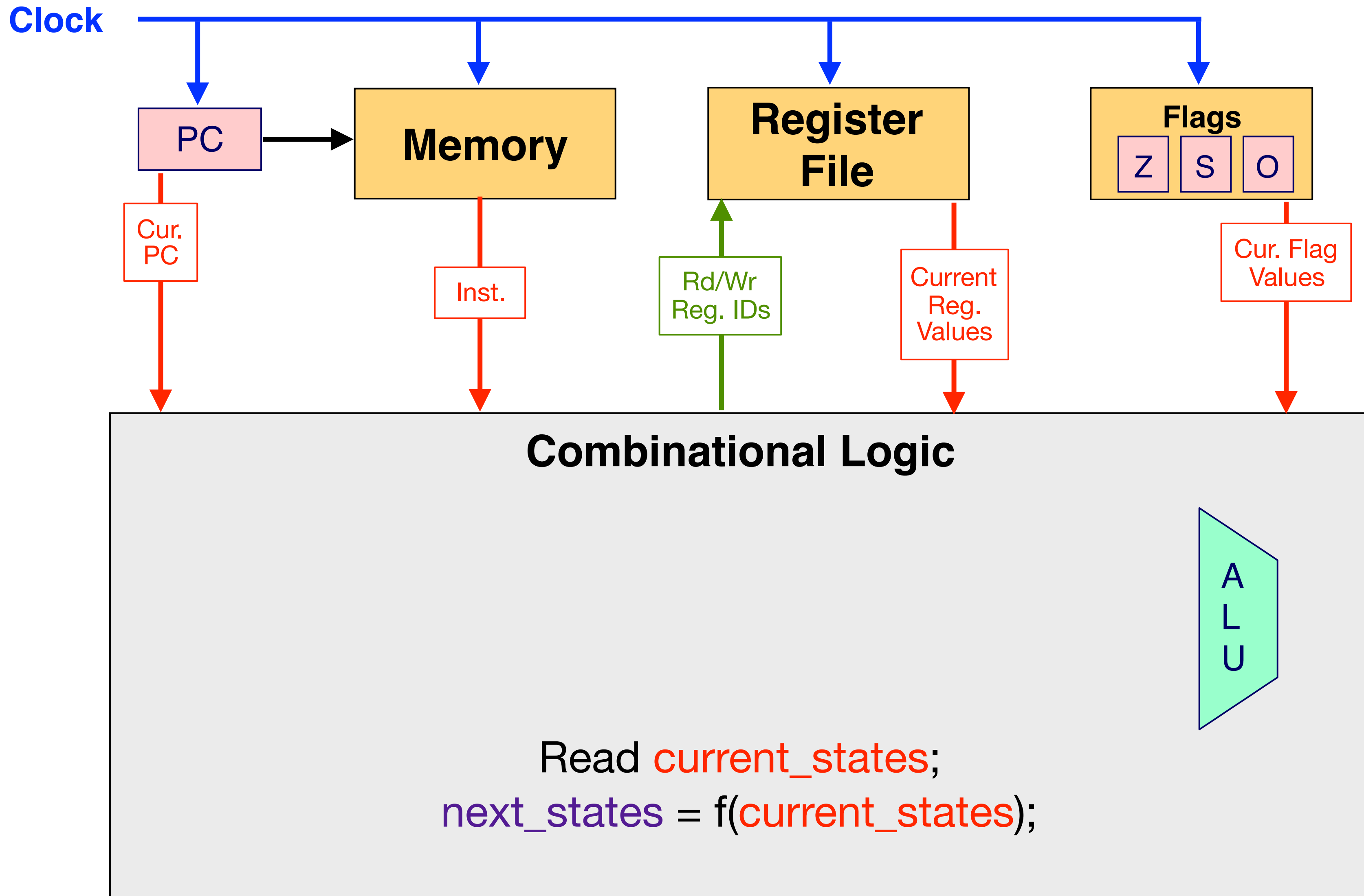
Microarchitecture (So far)



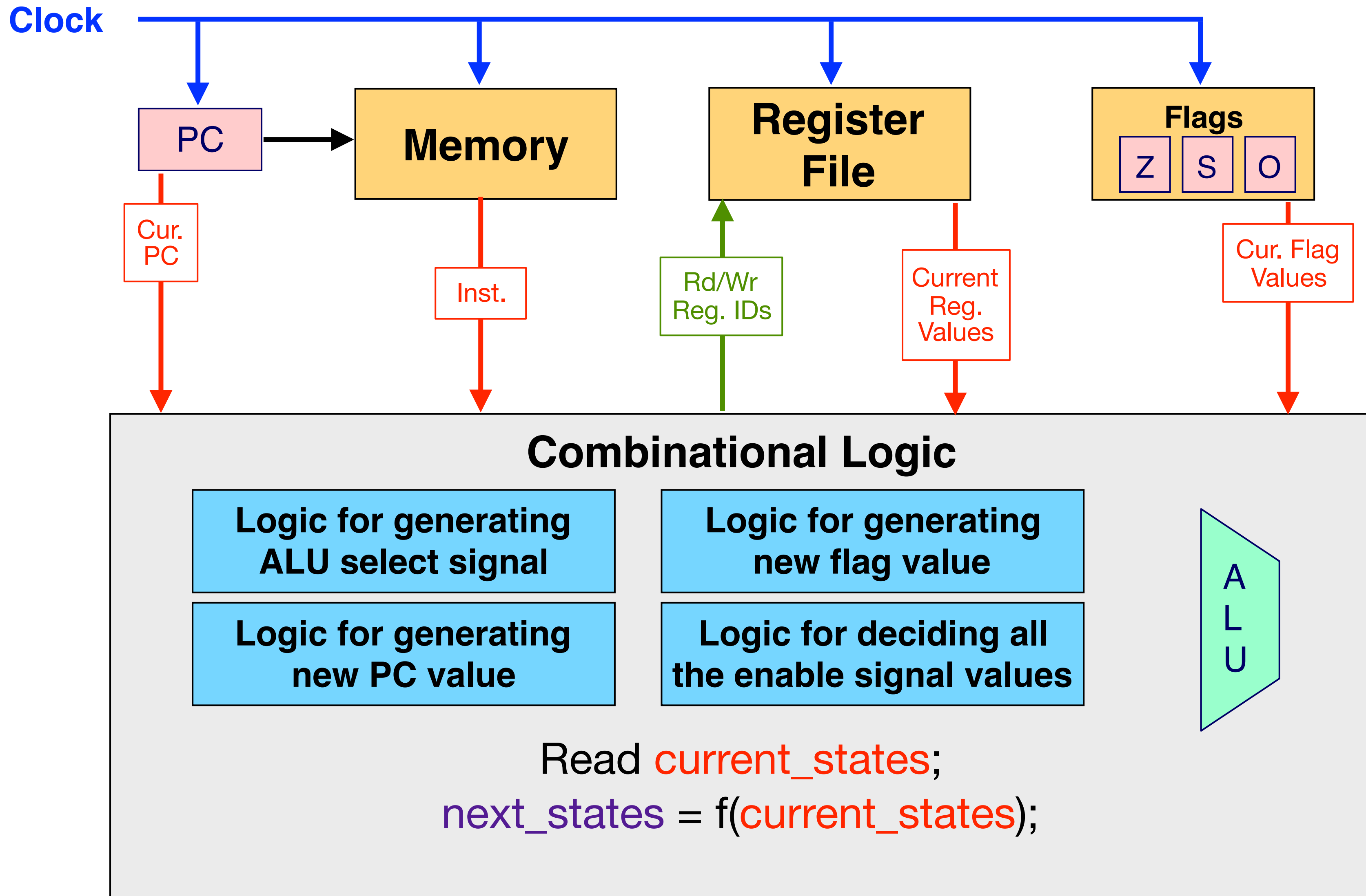
Microarchitecture (So far)



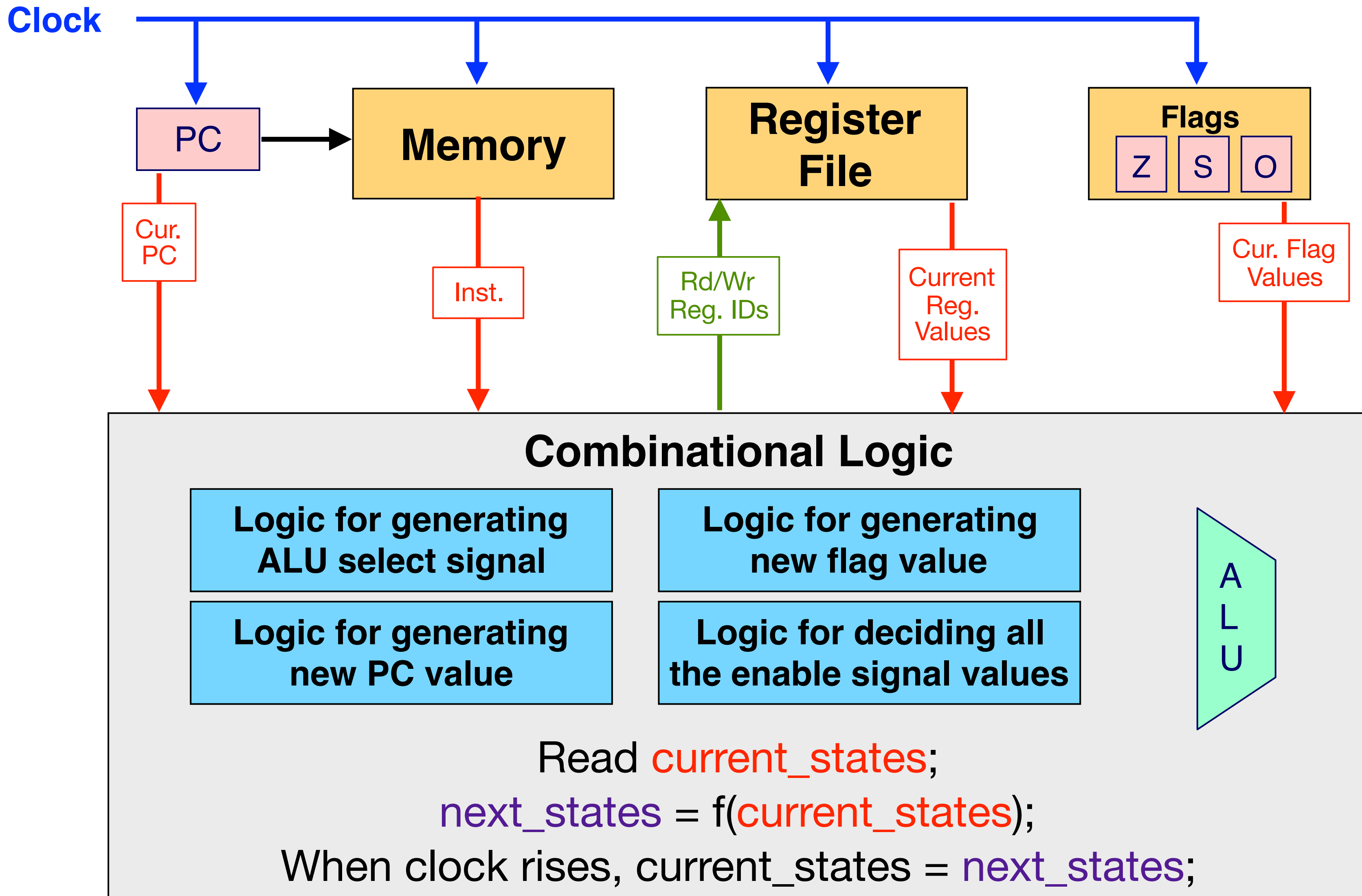
Microarchitecture (So far)



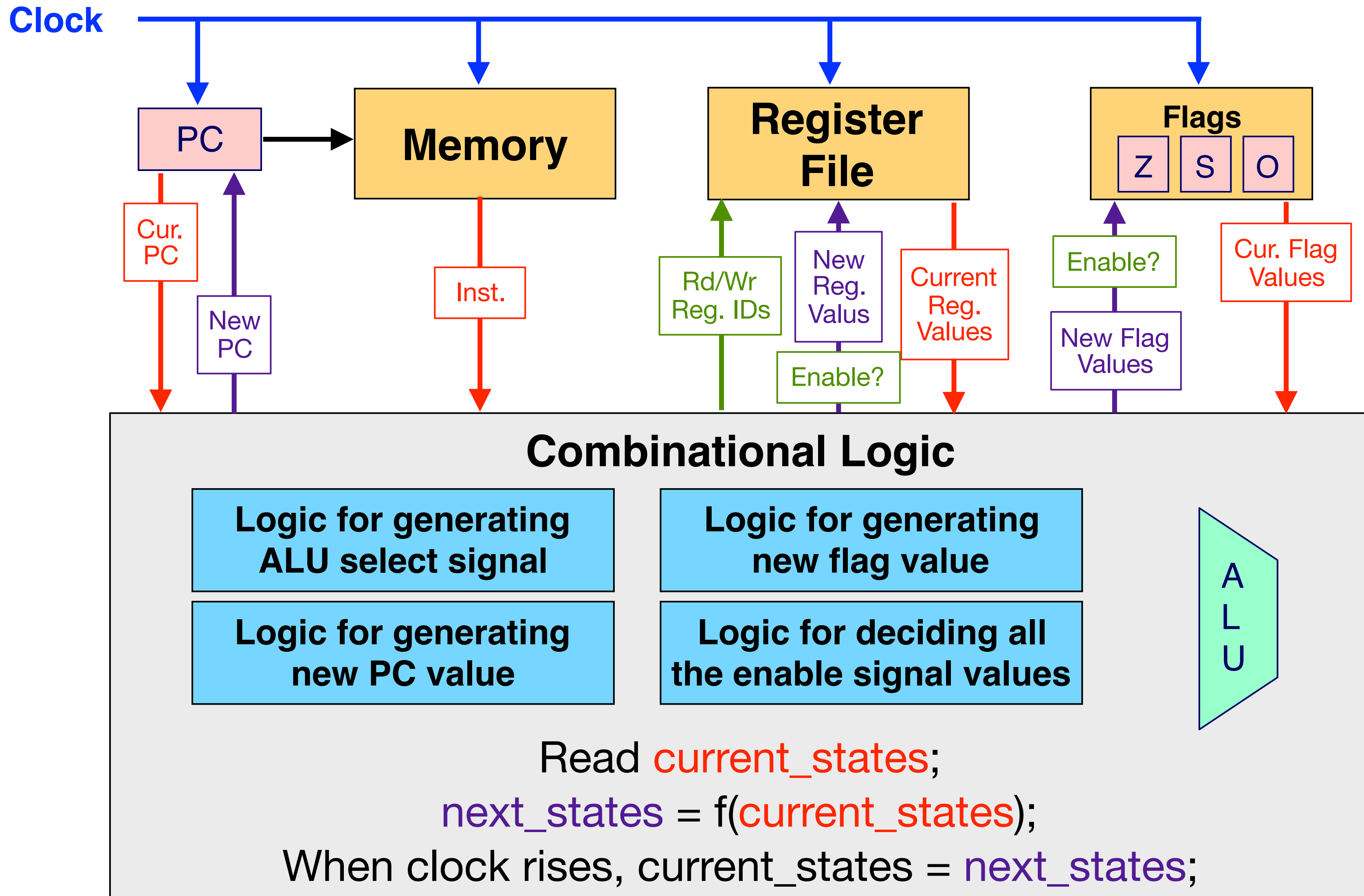
Microarchitecture (So far)



Microarchitecture (So far)

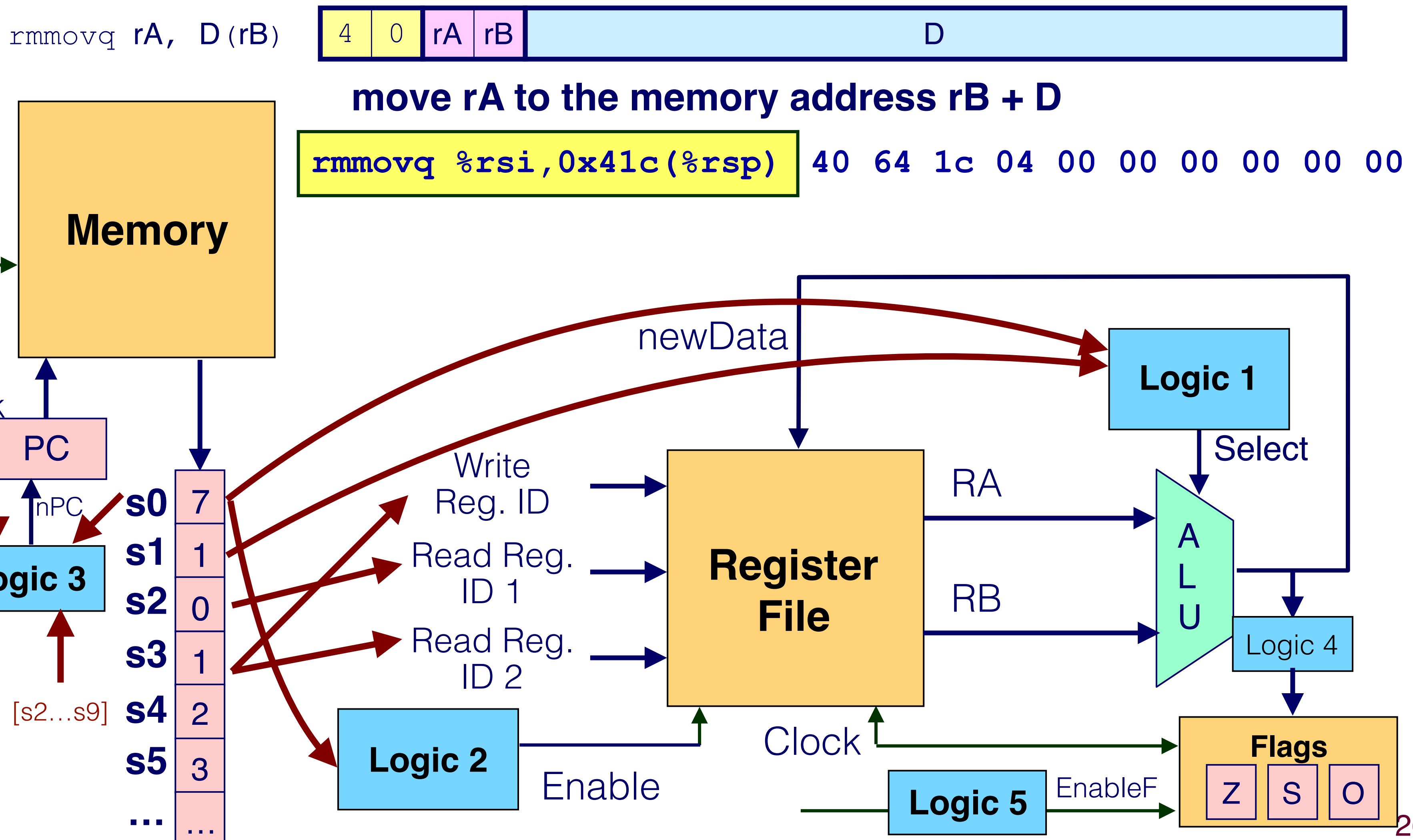


Microarchitecture (So far)



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)`

Memory

Clock

PC

oPC

nPC

Logic 3

Flags

[s2…s9]

s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
… …

newData

Logic 1

Select

Write Reg. ID

Read Reg. ID 1

Read Reg. ID 2

Register File

RA

RB

A L U

Logic 4

Logic 2

Enable

Clock

Flags

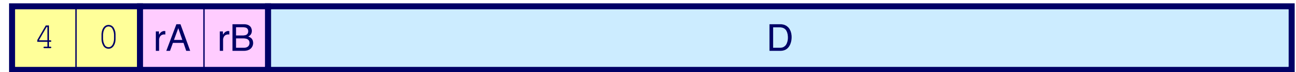
Z S O

Logic 5

EnableF

move rA to the memory address rB + D

rmmovq rA, D(rB)



Memory

Clock

PC

oPC

nPC

Logic 3

Flags

[s2…s9]

s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
… …

newData

Logic 1

Select

[s4…s19]

M U X

Write Reg. ID

Read Reg. ID 1

Read Reg. ID 2

Register File

RA

RB

A L U

Logic 4

Logic 2

Enable

Clock

Flags

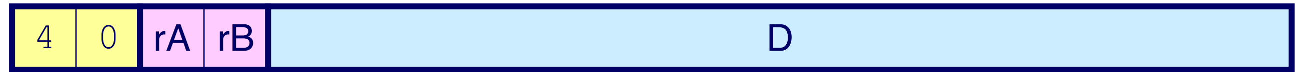
Z S O

Logic 5

EnableF

move rA to the memory address rB + D

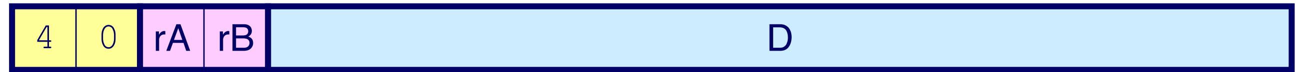
`rmmovq rA, D(rB)`



Memory
Clock
PC
oPC
nPC
Logic 3
Flags
[s2…s9]
s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
… …
newData
Write Reg. ID
Read Reg. ID 1
Read Reg. ID 2
Register File
Logic 2
Enable
Clock
[s4…s19]
MUX
RA
RB
Logic 1
Select
ALU
Logic 4
Flags
Z S O
Logic 5
EnableF

move rA to the memory address rB + D

`rmmovq rA, D(rB)`

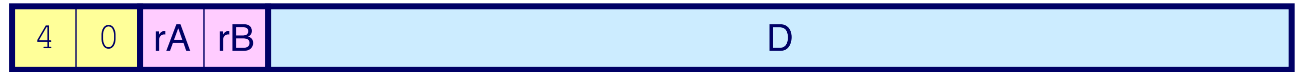


- Need new logic (Logic 6) to select the input to the ALU for Enable.

Memory
Logic 6
newData
Logic 1
Clock
PC
[s4…s19]
M U X
Select
oPC
nPC
Write Reg. ID
RA
s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
… …
Logic 3
Read Reg. ID 1
Register File
A L U
RB
Read Reg. ID 2
Logic 4
Flags
[s2…s9]
Logic 2
Clock
Flags
Z S O
Enable
EnableF
Logic 5

move rA to the memory address rB + D

`rmmovq rA, D(rB)`

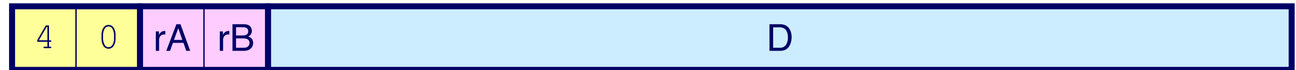


- Need new logic (Logic 6) to select the input to the ALU for Enable.

Memory
Address
newData
Logic 6
Logic 1
Select
Clock
PC
oPC
nPC
Logic 3
Flags
[s2…s9]
s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
… …
Write Reg. ID
Read Reg. ID 1
Read Reg. ID 2
Register File
[s4…s19]
MUX
RA
RB
ALU
Logic 4
Logic 2
Enable
Clock
Flags
Z S O
Logic 5
EnableF

move rA to the memory address rB + D

`rmmovq rA, D(rB)`

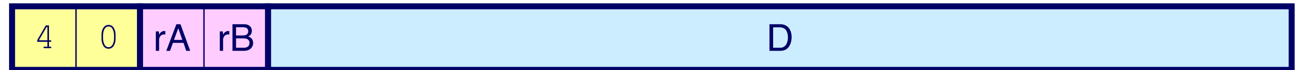


- Need new logic (Logic 6) to select the input to the ALU for Enable.

Memory
Data to write
Address
Logic 6
newData
Logic 1
Clock
PC
[s4…s19]
M U X
Select
Write Reg. ID
RA
A L U
oPC
nPC
s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
… …
Logic 3
Read Reg. ID 1
Register File
RB
Read Reg. ID 2
Logic 4
Flags
[s2…s9]
Logic 2
Clock
Enable
Flags
Z S O
Logic 5
EnableF

move rA to the memory address rB + D

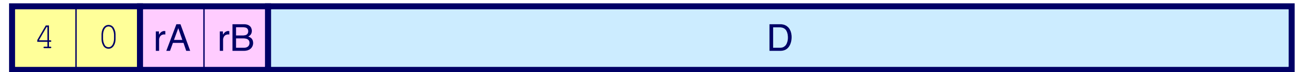
`rmmovq rA, D(rB)`



- 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?

Memory
Data to write
Enable
Address
Logic 6
newData
Logic 1
Select
Clock
PC
oPC
nPC
Logic 3
Flags
[s2…s9]
s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
… …
Write Reg. ID
Read Reg. ID 1
Read Reg. ID 2
Register File
[s4…s19]
RA
RB
MUX
ALU
Logic 4
Logic 2
Enable
Clock
Flags
Z S O
Logic 5
EnableF

How About Memory to Register MOV?

move data at memory address rB + D to rA

`mrmovq D(rB), rA`

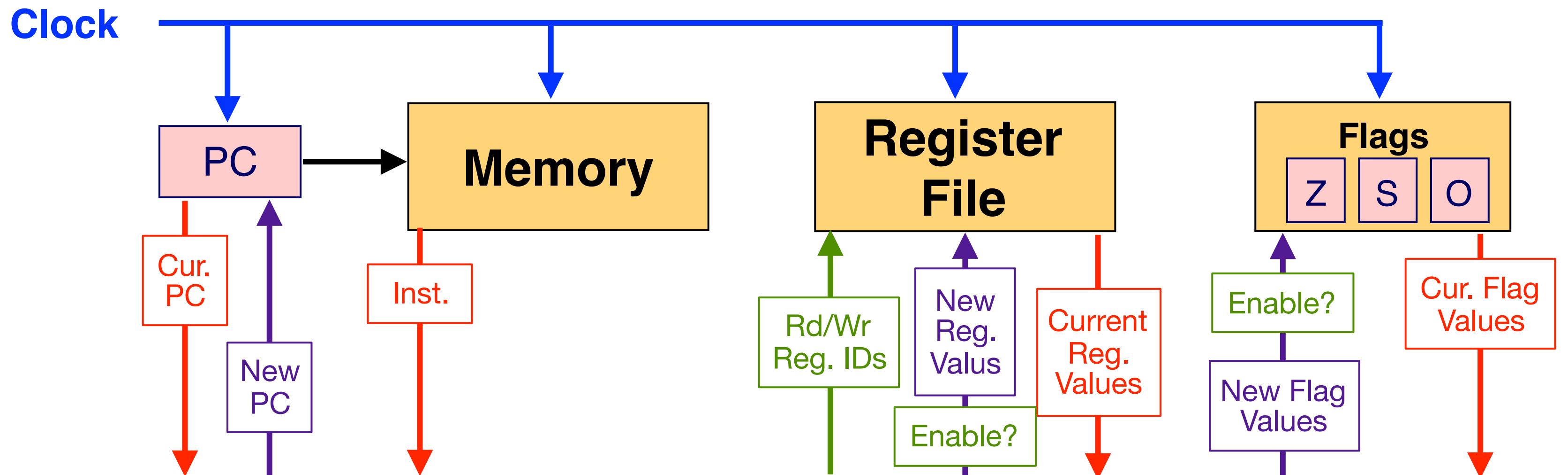
Memory
Data to write
Enable
Address
Logic 6
newData
Logic 1
[s4…s19]
M
U
X
Select
RA
RB
A
L
U
Logic 4
Register File
Write Reg. ID
Read Reg. ID 1
Read Reg. ID 2
Clock
PC
oPC
nPC
Logic 3
Flags
[s2…s9]
s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
… …
Logic 2
Enable
Clock
Logic 5
EnableF
Flags
Z
S
O

How About Memory to Register MOV?

move data at memory address rB + D to rA

Memory
Data to write
Enable
Data read back
Address
Logic 6
Logic 7
MUX
Logic 1
Clock
PC
oPC
nPC
Logic 3
Flags
[s2…s9]
s0 4
s1 0
s2 6
s3 4
s4 1
s5 c
… …
newData
[s4…s19]
Write
Reg. ID
Read Reg.
ID 1
Read Reg.
ID 2
Register
File
RA
RB
MUX
Select
ALU
Logic 4
Logic 2
Enable
Clock
Logic 5
EnableF
Flags
Z S O

Microarchitecture (with MOV)



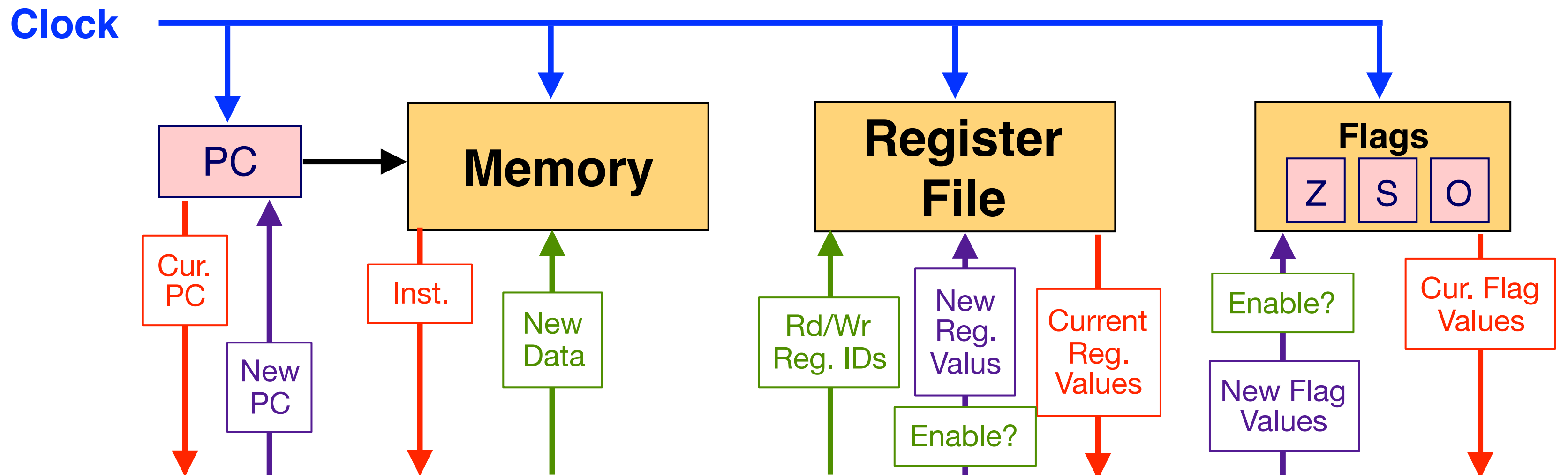
Combinational Logic

Read current_states;

next_states = f(current_states);

When clock rises, current_states = next_states;

Microarchitecture (with MOV)



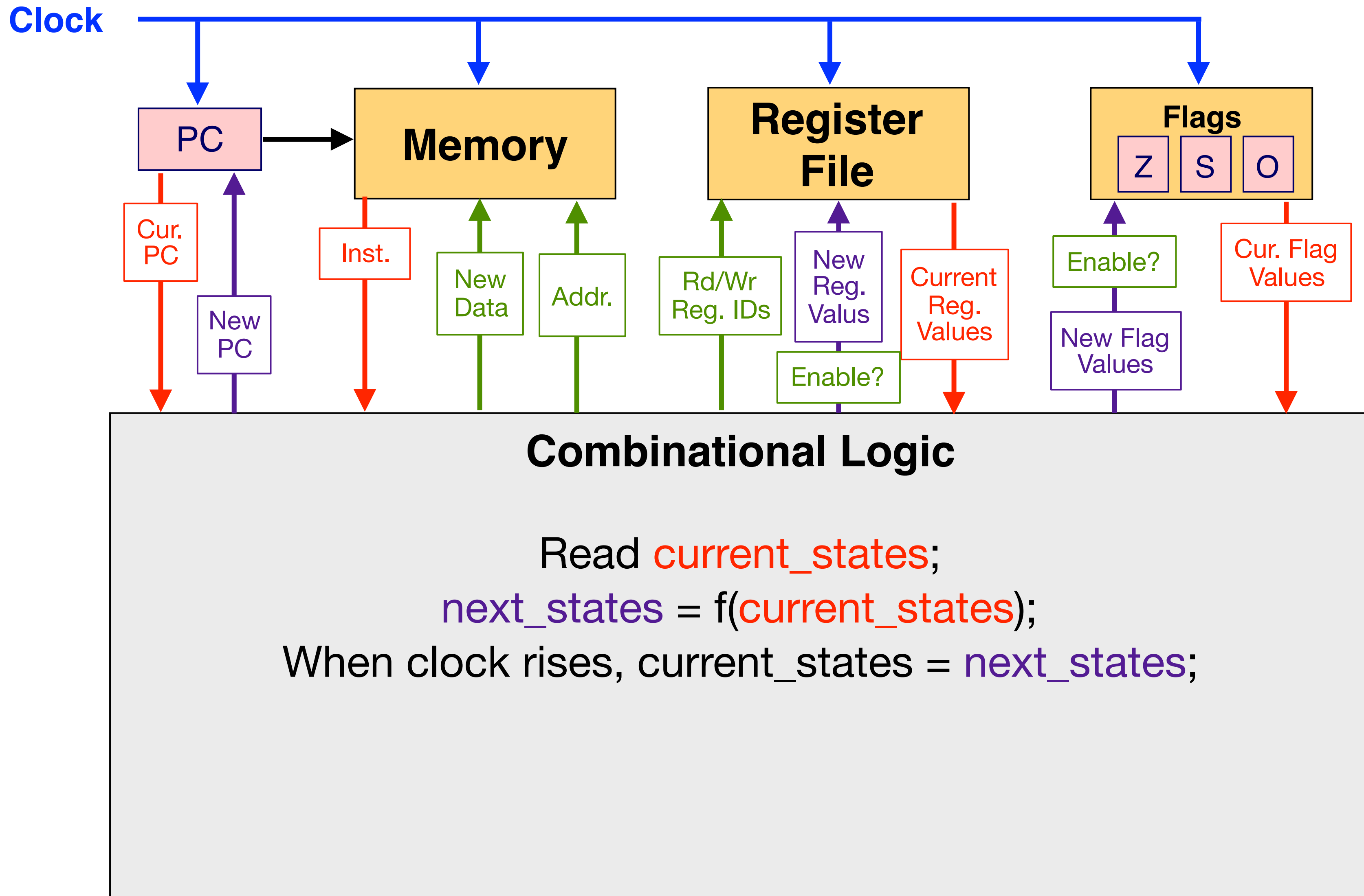
Combinational Logic

Read current_states;

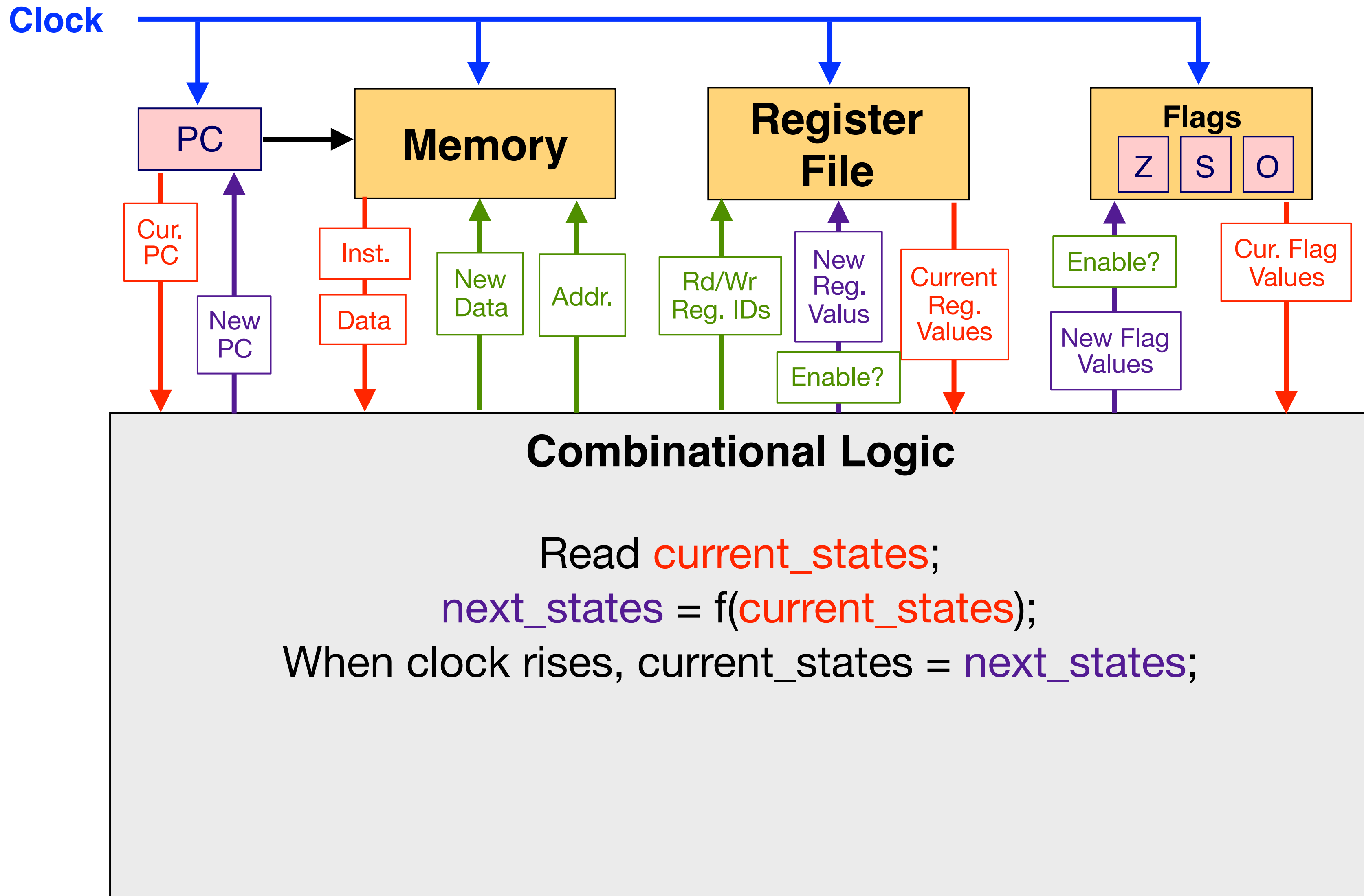
next_states = f(current_states);

When clock rises, current_states = next_states;

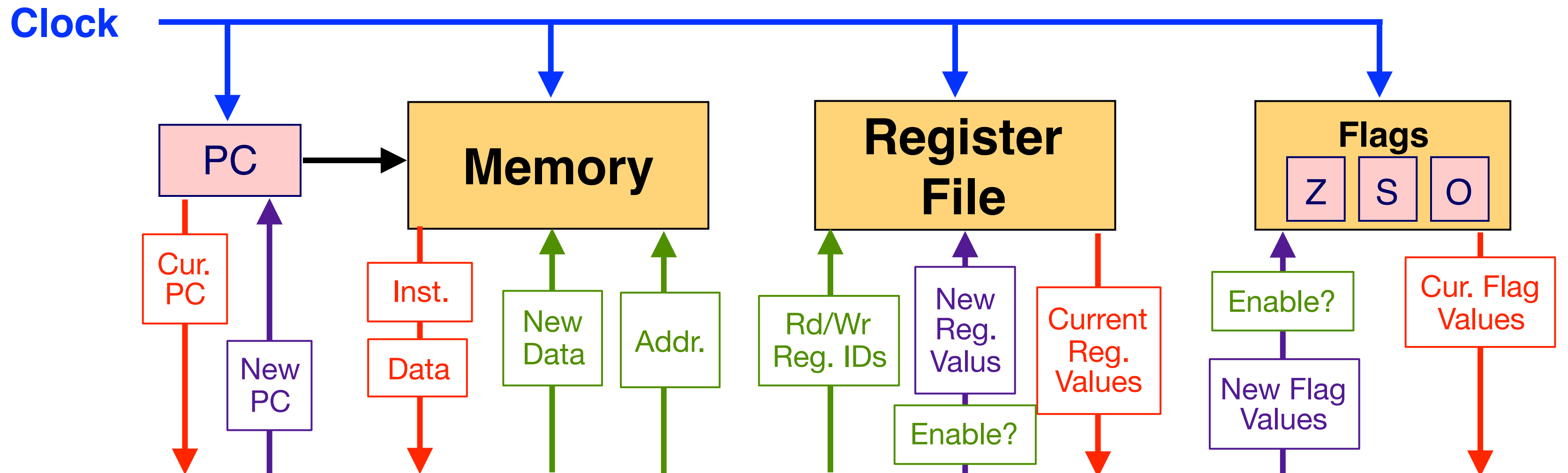
Microarchitecture (with MOV)



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