CSC 252: Computer Organization
Spring 2020: Lecture 12

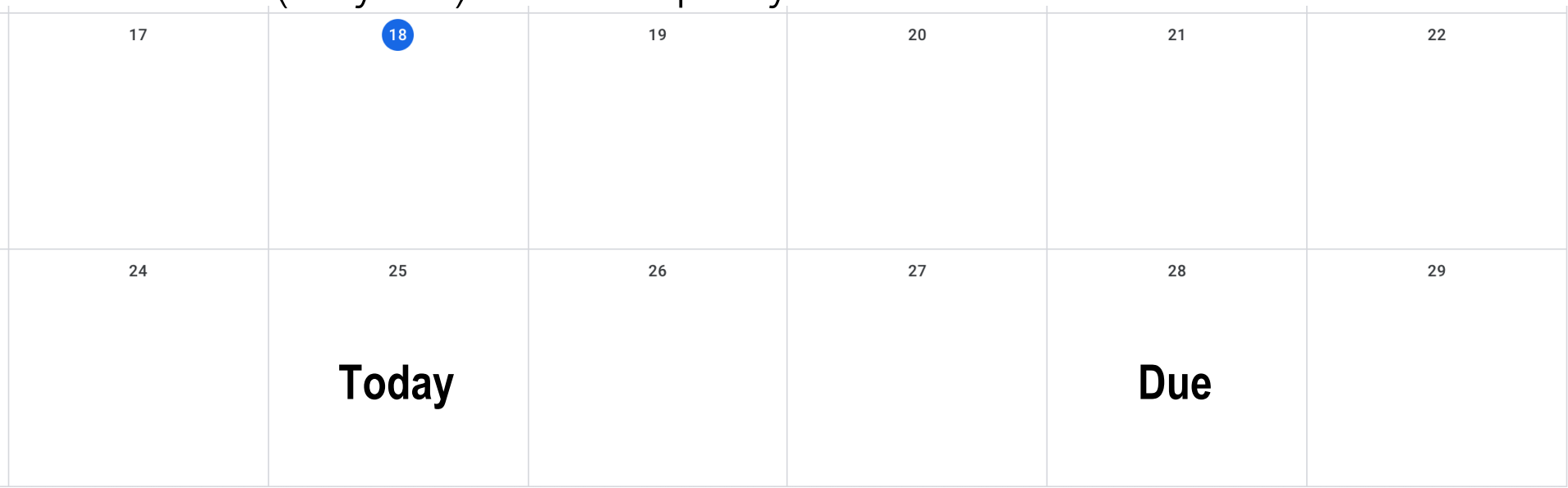
Processor Architecture

Substitute Instructor: Sandhya Dwarkadas

Department of Computer Science
University of Rochester

Announcement

- Programming assignment 3 due soon
  - Details: https://www.cs.rochester.edu/courses/252/spring2020/labs/assignment3.html
  - Due on **Feb. 28**, 11:59 PM
  - You (may still) have 3 slip days



Announcement

- Grades for lab2 are posted.
- If you think there are some problems
  - Take a deep breath
  - Tell yourself that the teaching staff like you, not the opposite
  - Email/go to Shuang or Sudhanshu's office hours and explain to them why you should get more points, and they will fix it for you

Announcement

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

Overview of Logic Design

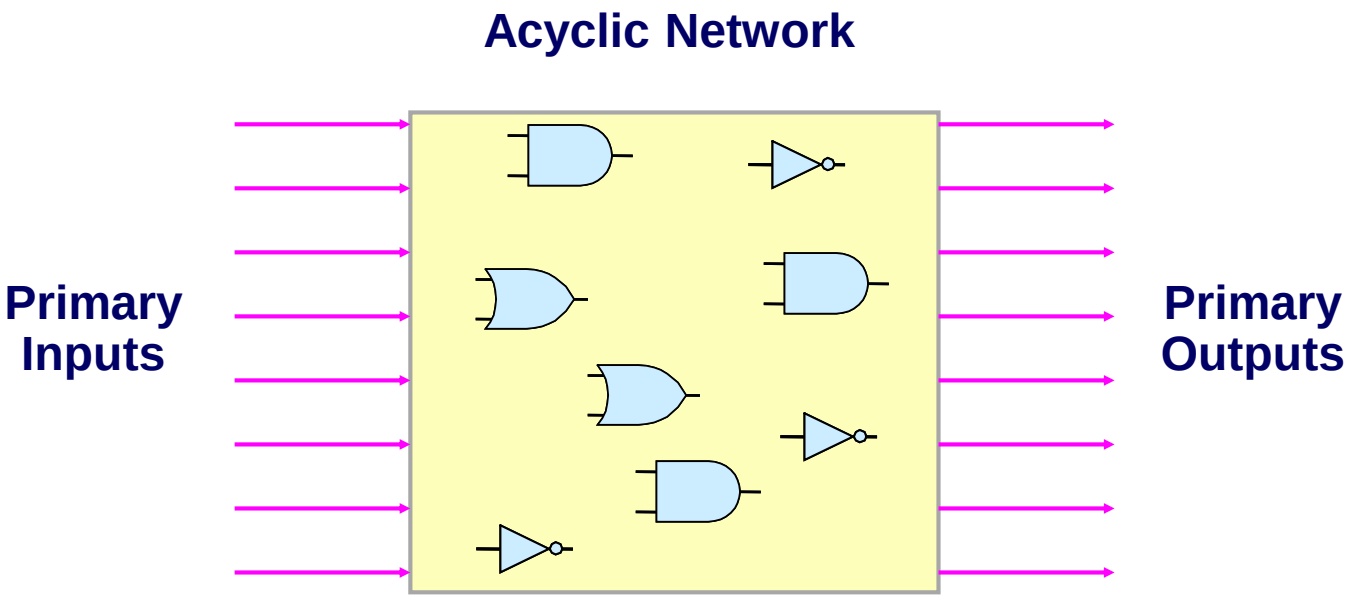
Fundamental Hardware Requirements

- **Communication**
  - **How to get values from one place to another**
- **Computation**
- **Storage**

Bits are Our Friends

- **Everything expressed in terms of values 0 and 1**
- **Communication**
  - **Low or high voltage on wire**
- **Computation**
  - **Compute Boolean functions**
- **Storage**
  - **Store bits of information**

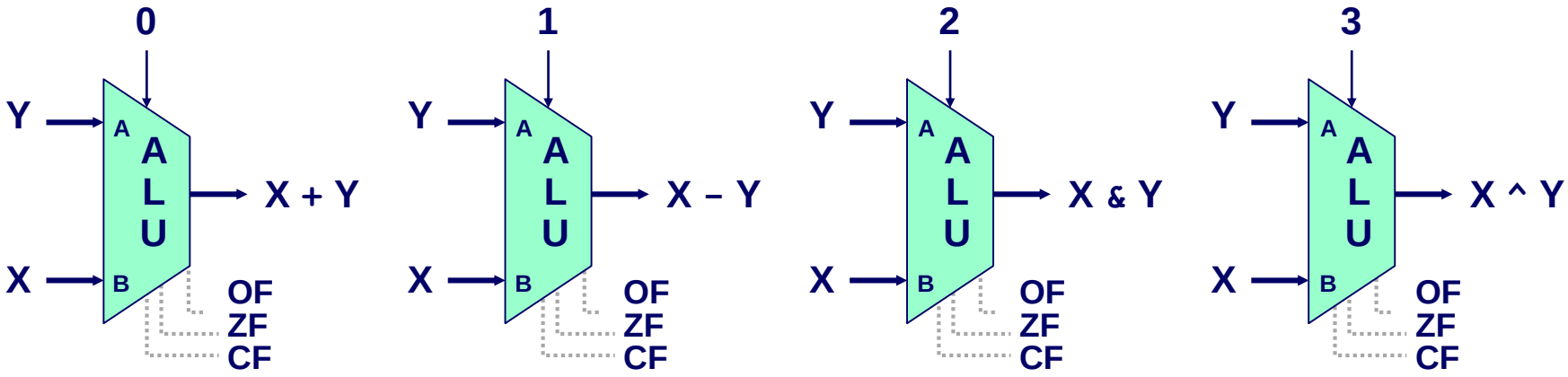
Combinational Circuits



Acyclic Network of Logic Gates

- Continously responds to changes on primary inputs
- Primary outputs become (after some delay) Boolean functions of primary inputs

Arithmetic Logic Unit



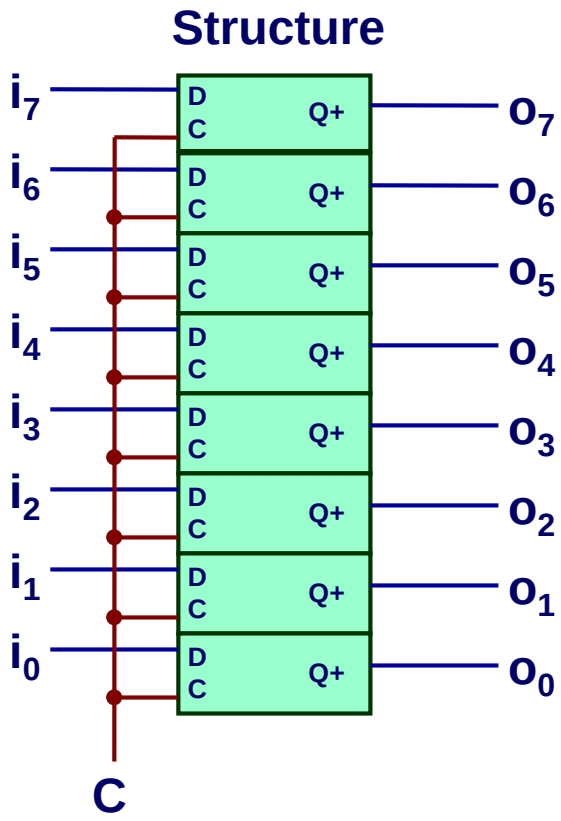
- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86
- Also computes values for condition codes

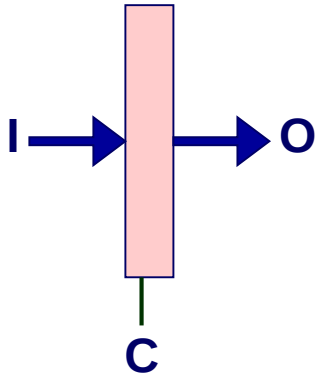
Sequential Logic: Memory and Control

Sequential:

- **Output depends on the *current* input values and the *previous* sequence of input values.**
- **Are Cyclic:**
  - **Output of a gate feeds its input at some future time.**
- **Memory:**
  - **Remember results of previous operations**
  - **Use them as inputs.**
- **Example of use:**
  - **Build registers and memory units.**

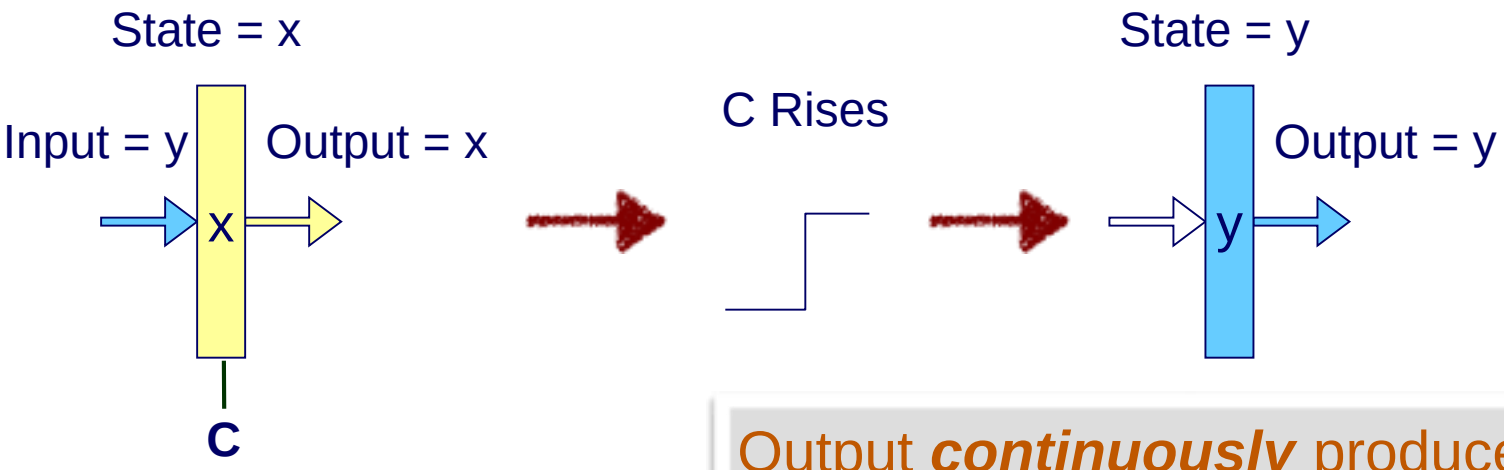
Registers





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

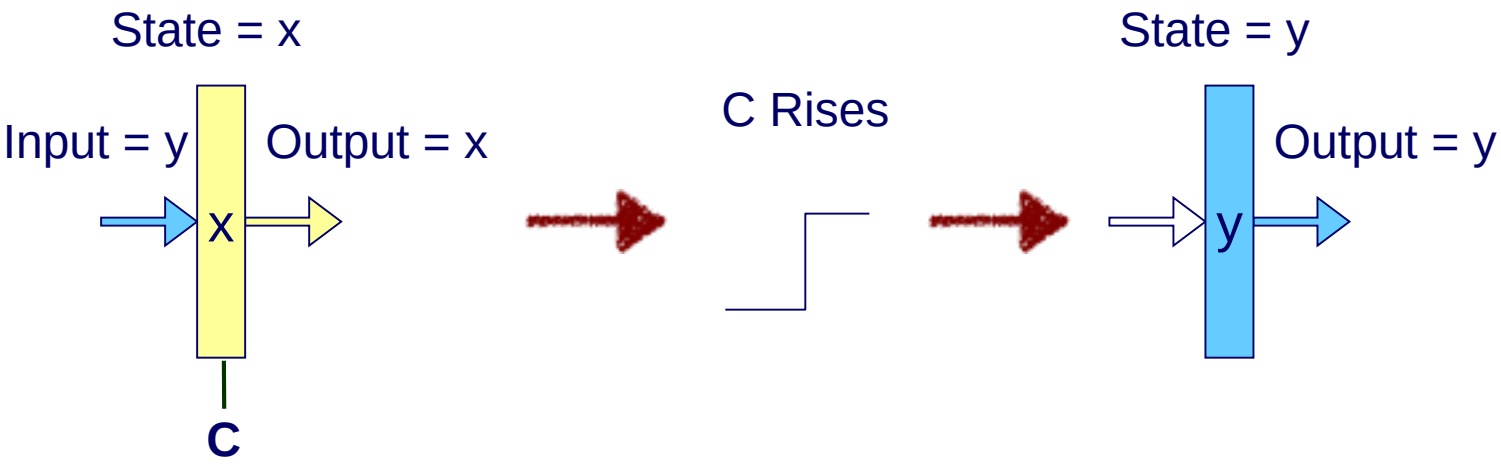
Register Operation



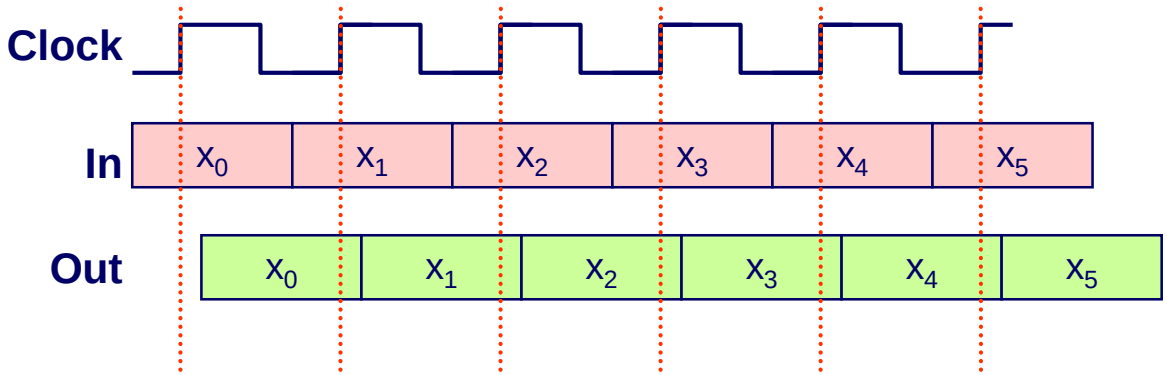
Output ***continuously*** produces y after the rising edge unless you cut off power.

- 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

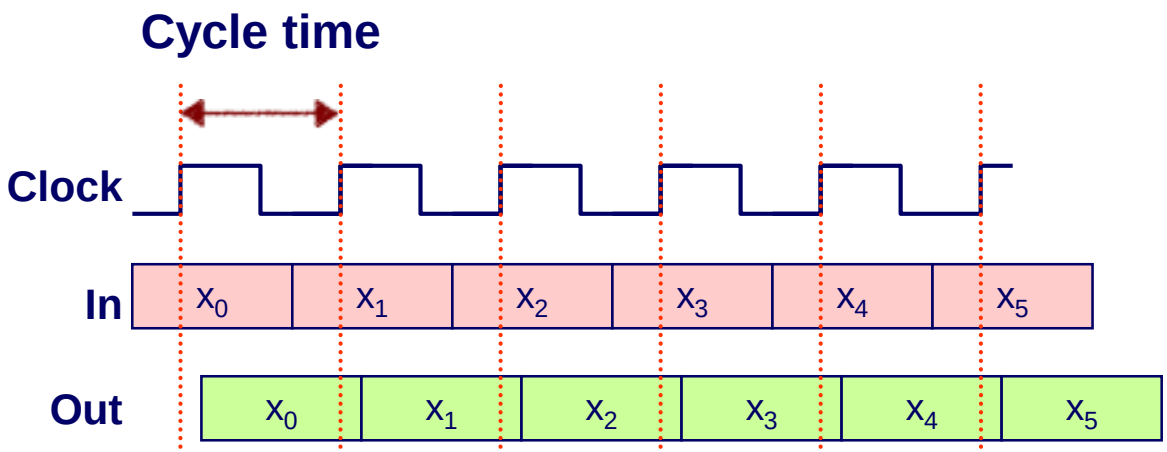


- A special C: periodically oscillating between 0 and 1
- That's called the **clock** signal. Generated by a crystal oscillator inside your computer



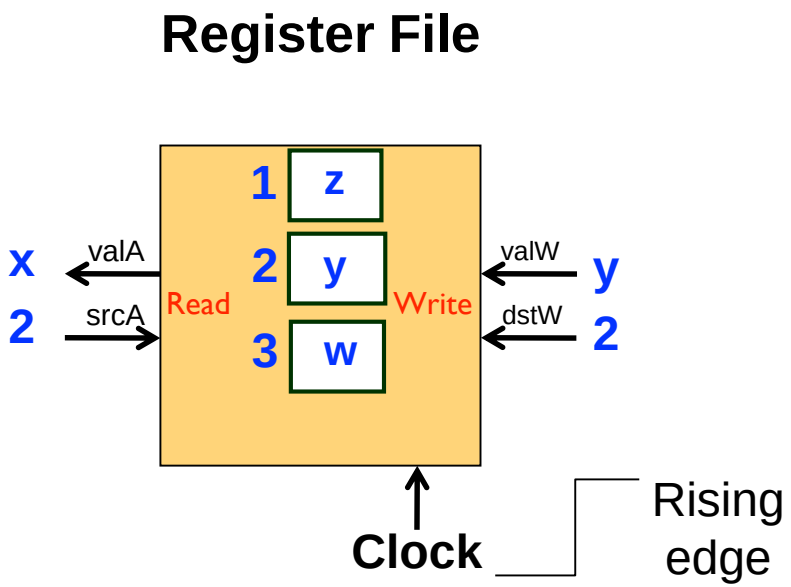
Clock Signal

- 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



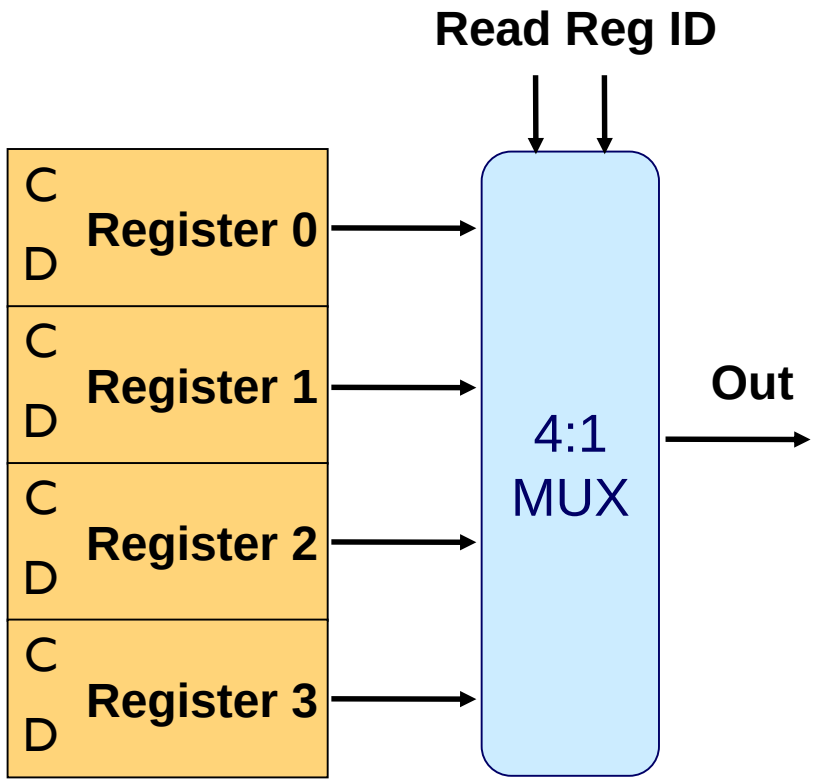
Register File

- A register file consists of a set of registers that you can individual 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??



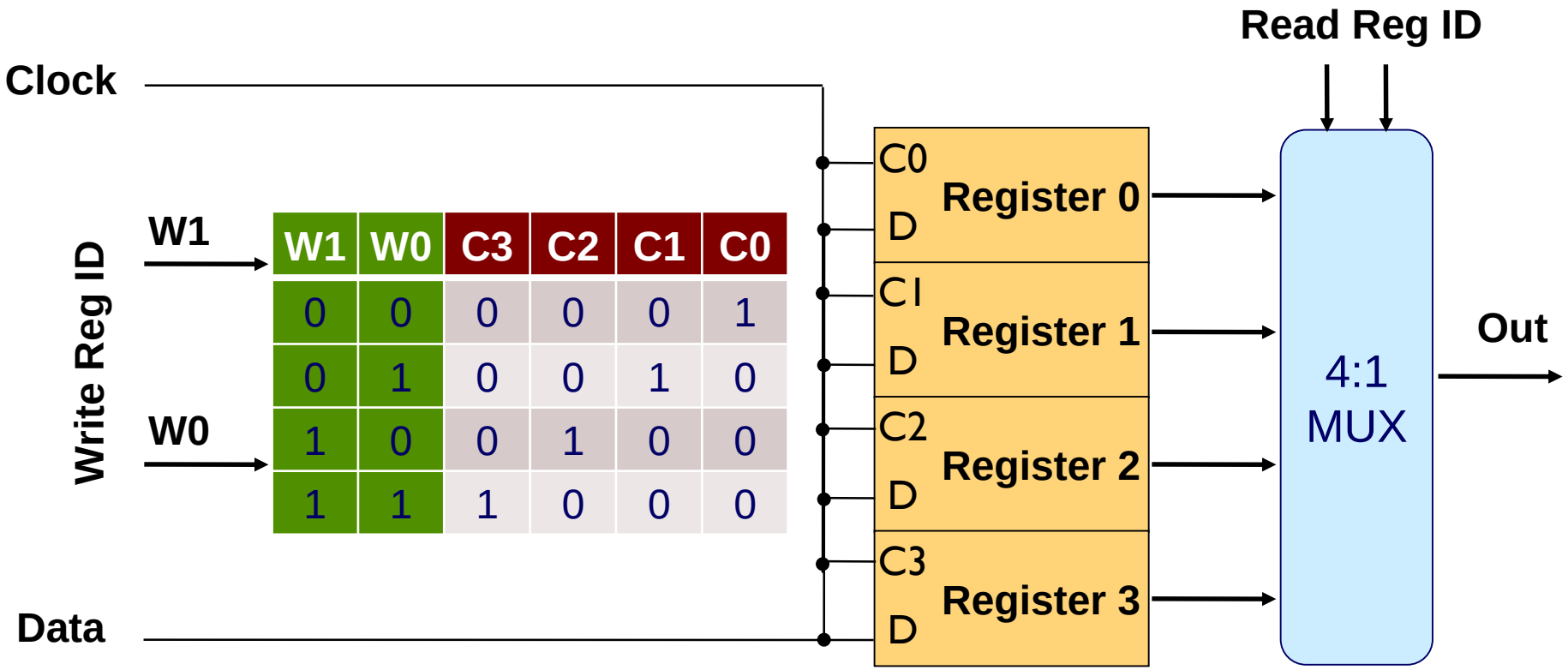
Register File Read

- Continuously read a register independent of the clock signal



Register File Write

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

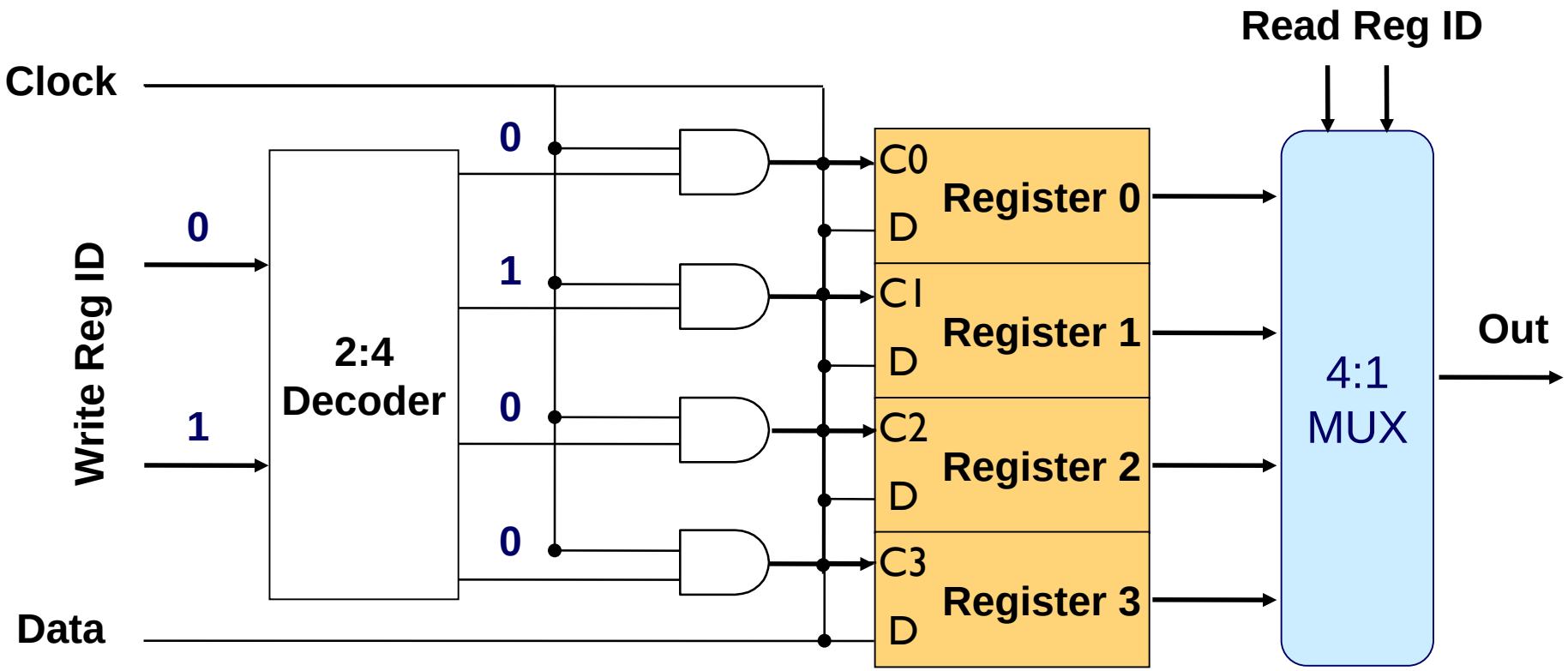


Decoder

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

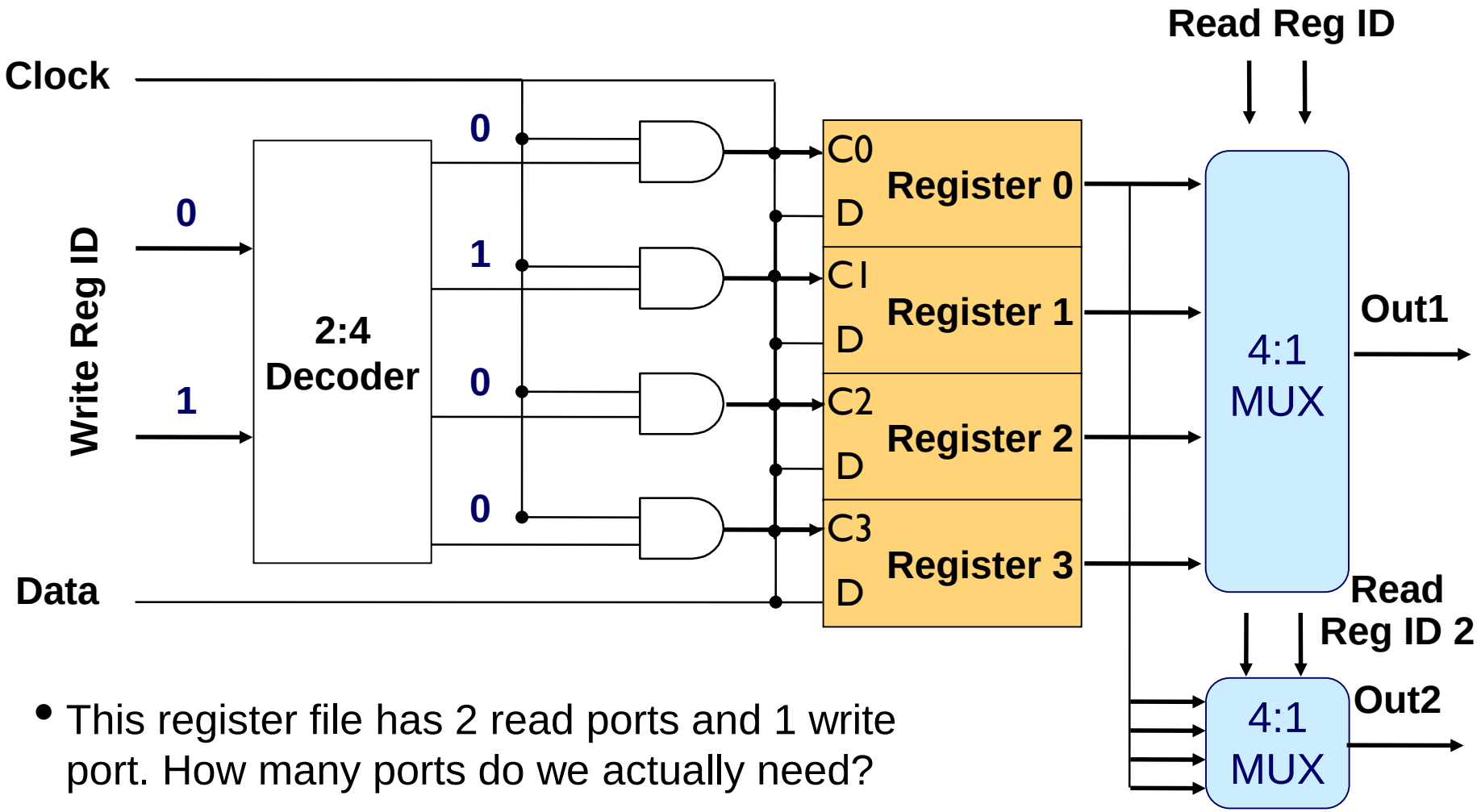
Register File Write



- 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

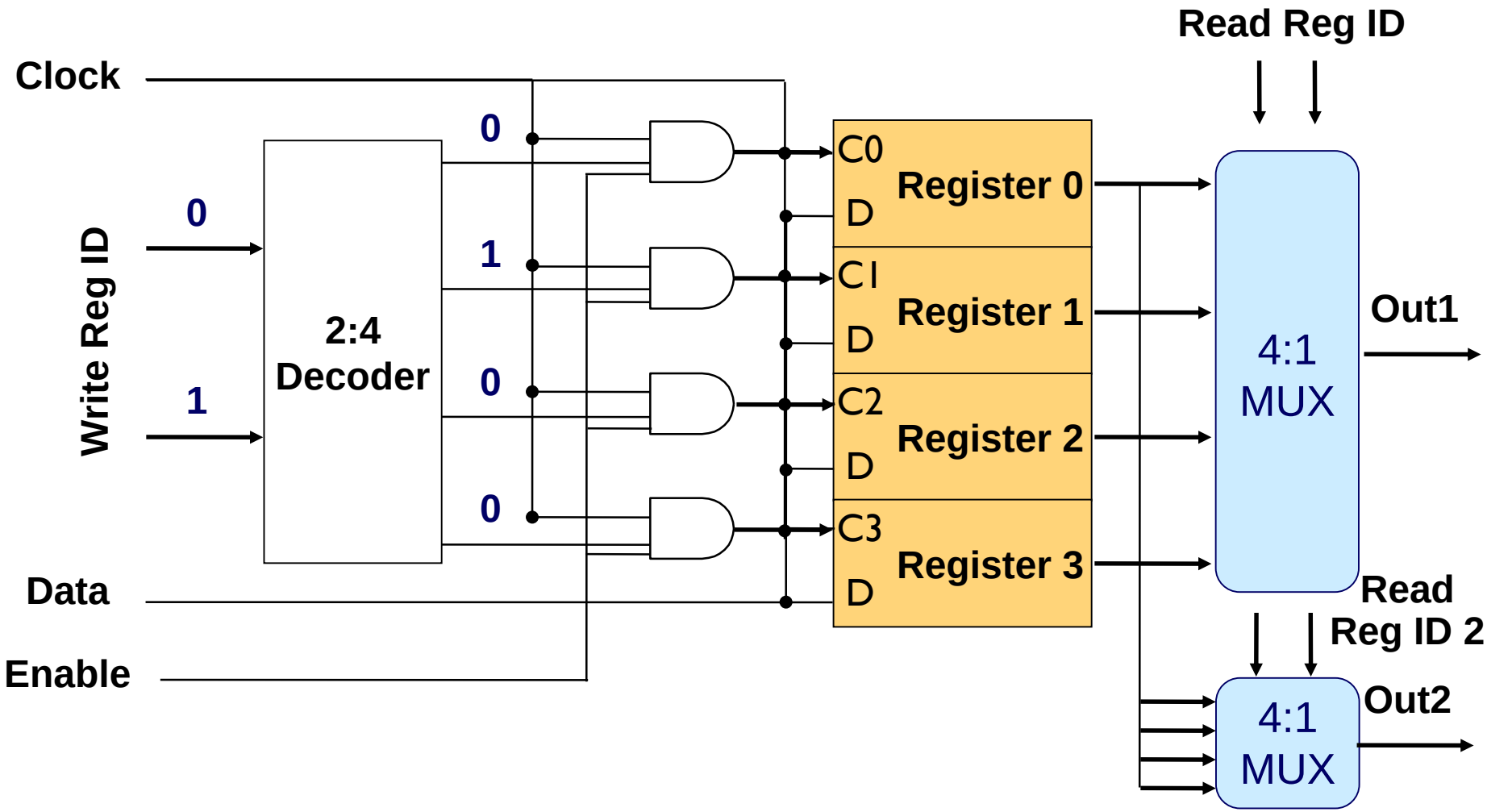
- What if we want to read multiple registers at the same time?



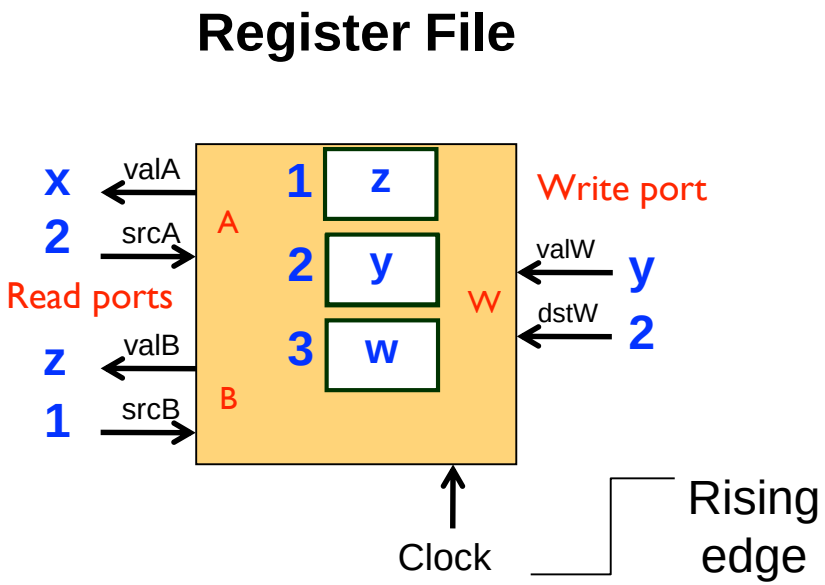
- This register file has 2 read ports and 1 write port. How many ports do we actually need?

Multi-Port Register File

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

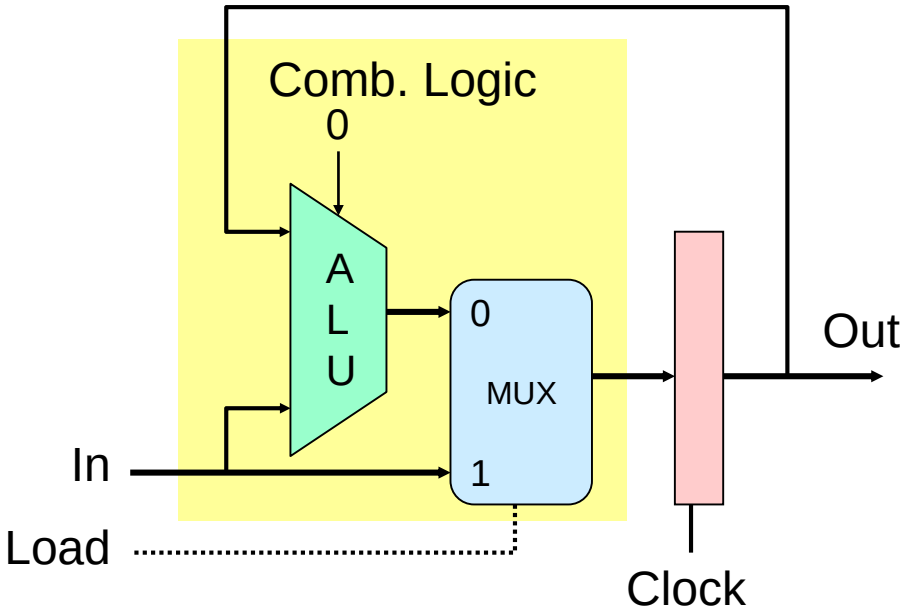


Register File

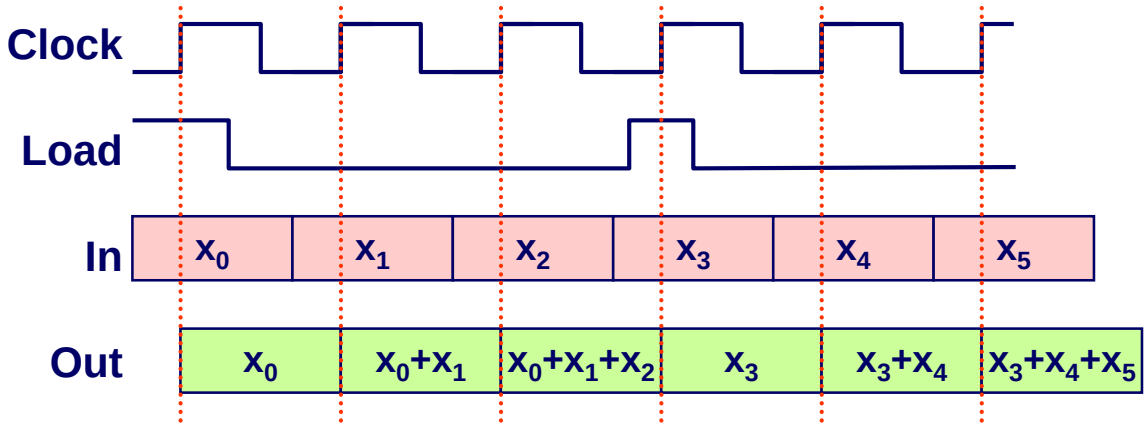


- Stores multiple registers of data
  - Address input specifies which register to read or write
- Register file is a form of Random-Access Memory (RAM)
- Multiple Ports: Can read and/or write multiple words in one cycle. Each port has separate address and data input/output

State Machine Example

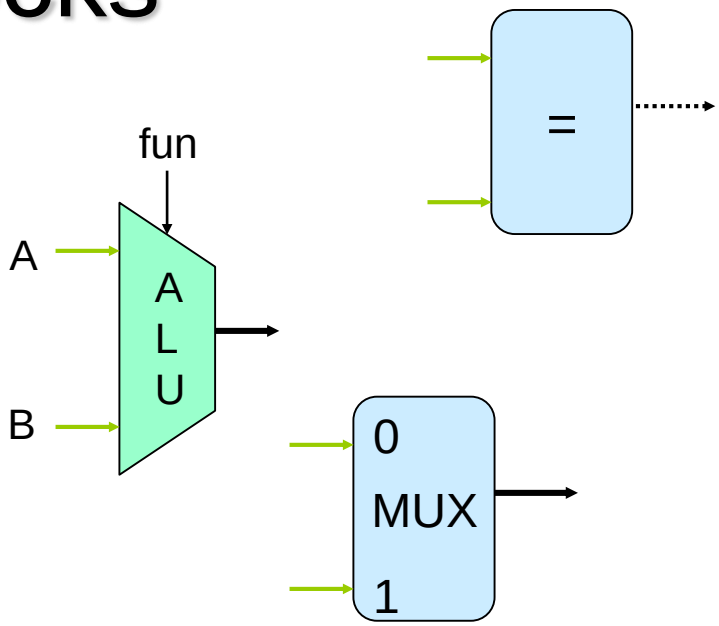


- Accumulator circuit
- Load or accumulate on each cycle

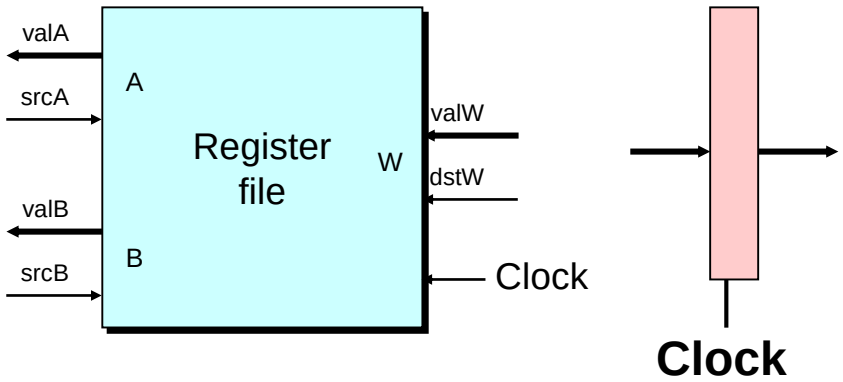


Building Blocks

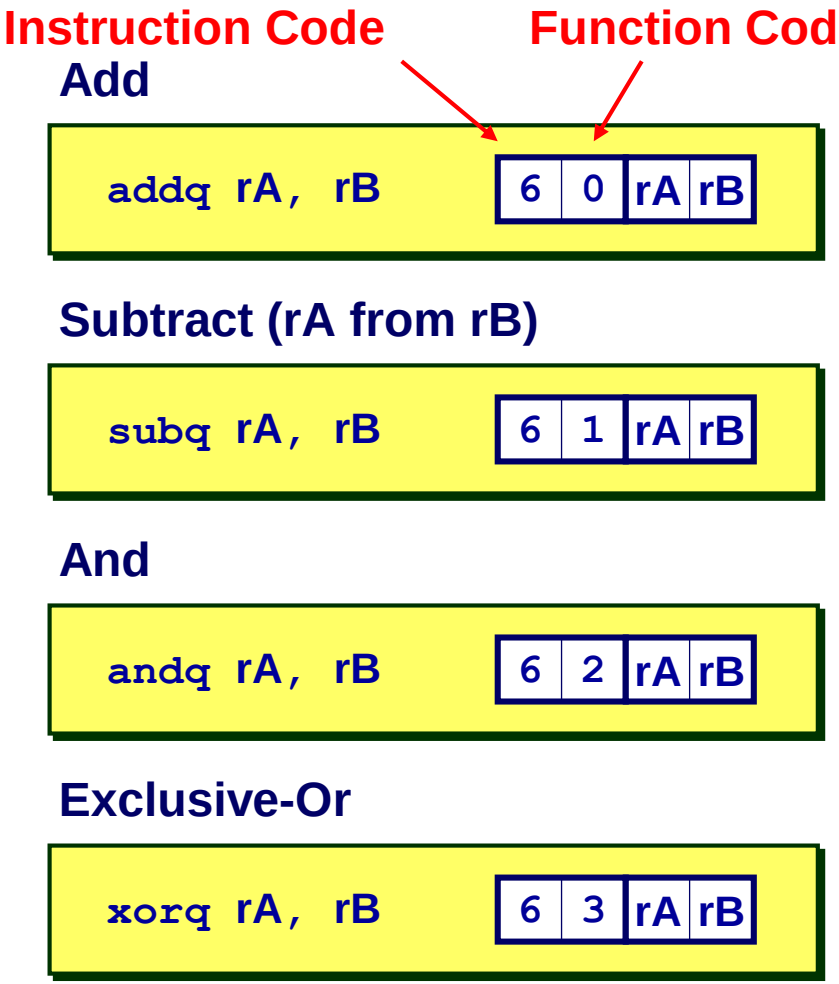
- Combinational Logic
  - Compute Boolean functions of inputs
  - Continuously respond to input changes
  - Operate on data and implement control



- Storage Elements
  - Store bits
  - Addressable memories
  - Non-addressable registers
  - Loaded only as clock rises



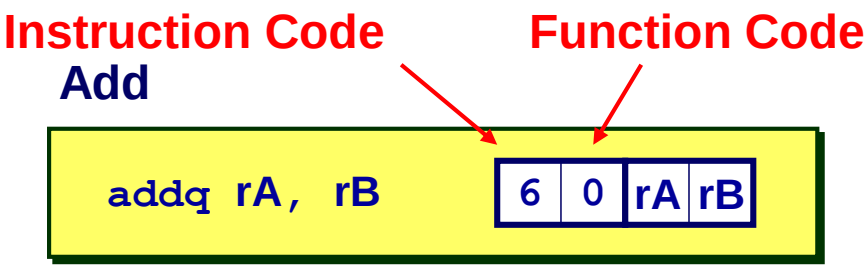
Arithmetic and Logical Operations



- Refer to generically as "`OPq`"
- Encodings differ only by "function code"
  - Low-order 4 bits in first instruction word
- Set condition codes as side effect

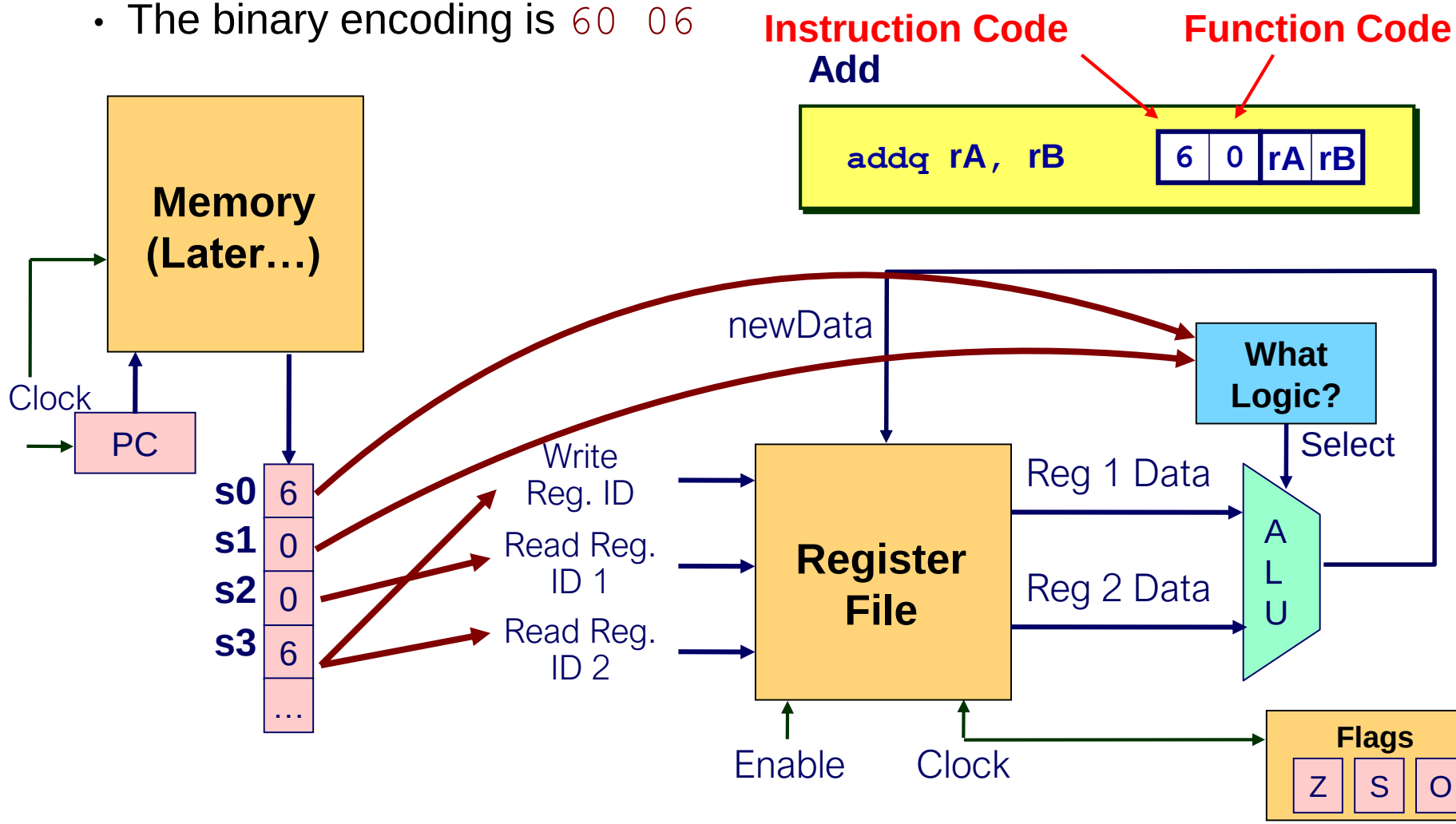
Executing an ADD instruction

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



Executing an ADD instruction

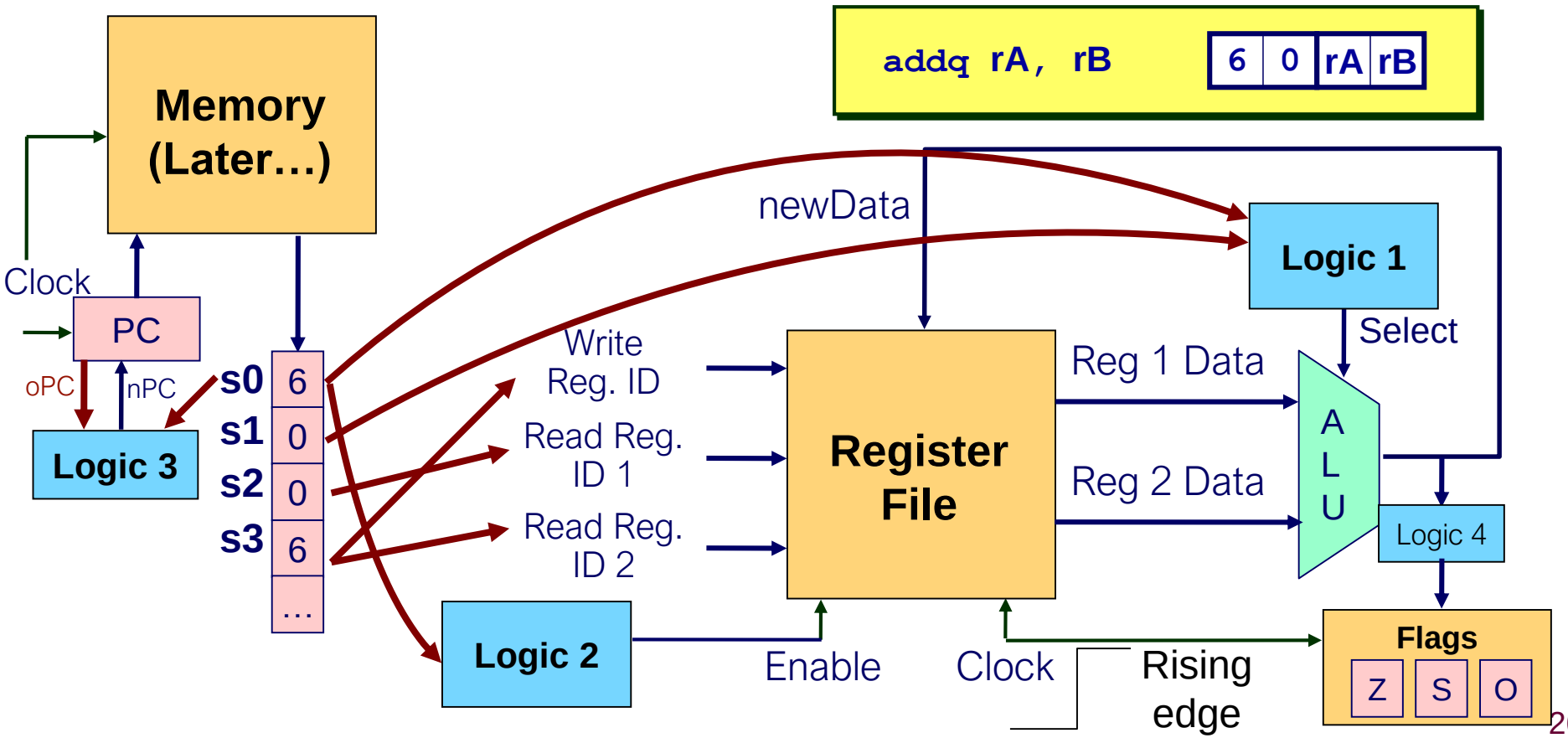
- How does the processor execute `addq %rax,%rsi`
- The binary encoding is `60 06`



Executing an ADD instruction

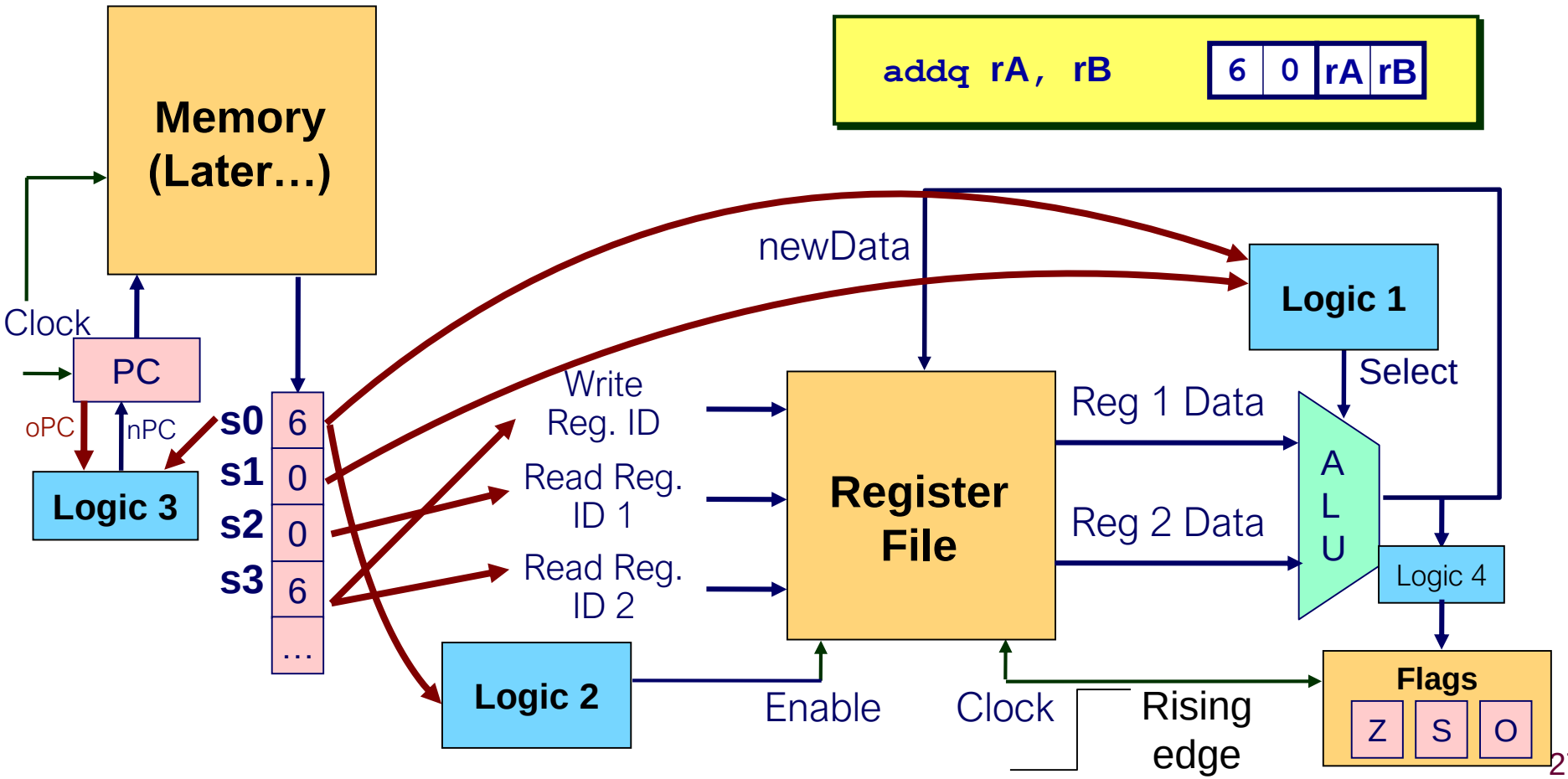
- Logic 1: if (s0 == 6) select = s1;
- Logic 2: if (s0 == 6) Enable = 1; else Enable = 0;
- Logic 3: if (s0 == 6) nPC = oPC + 2;
- How about Logic 4?

How do these logics get implemented?



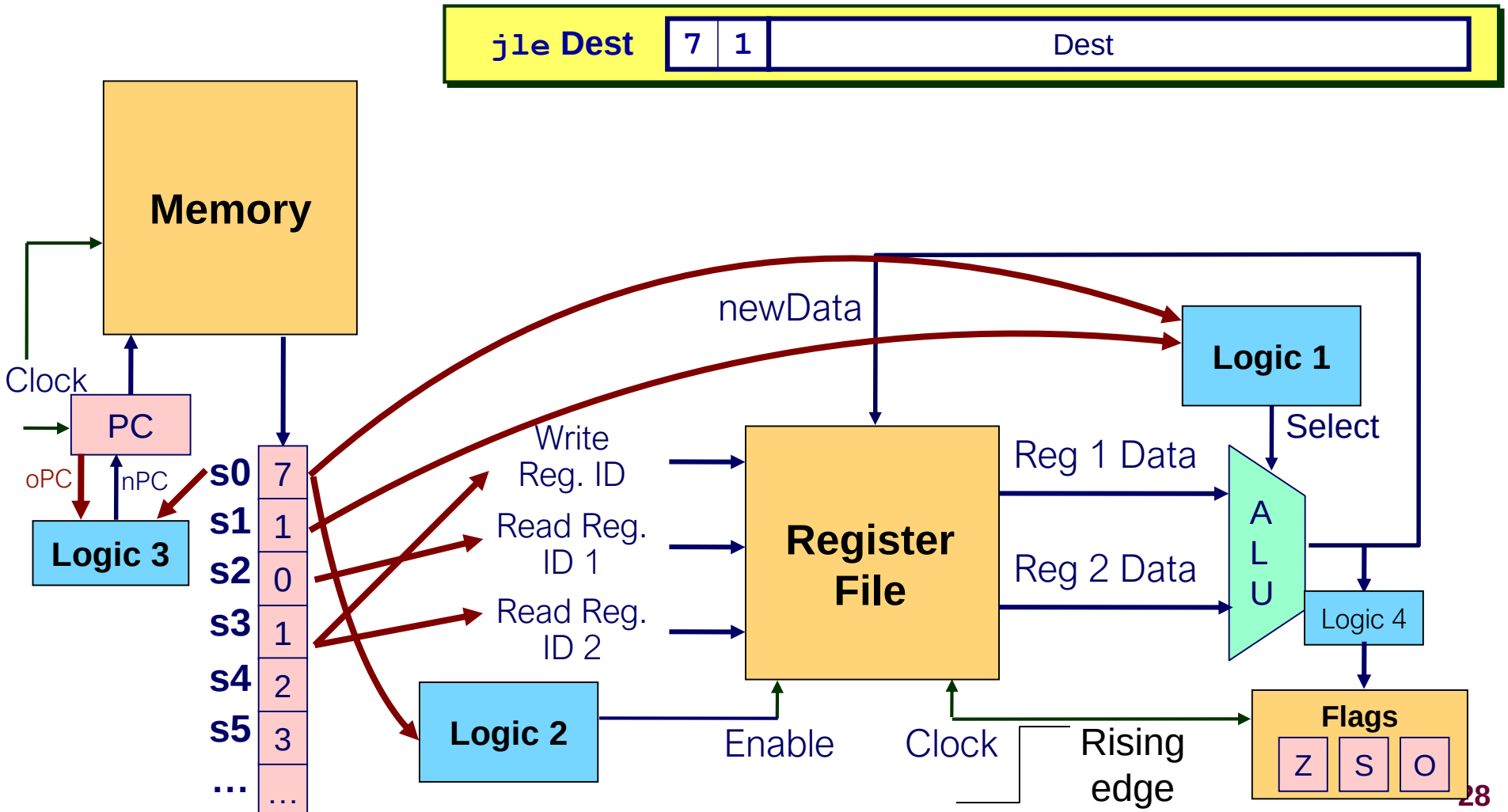
Executing an ADD instruction

- 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.



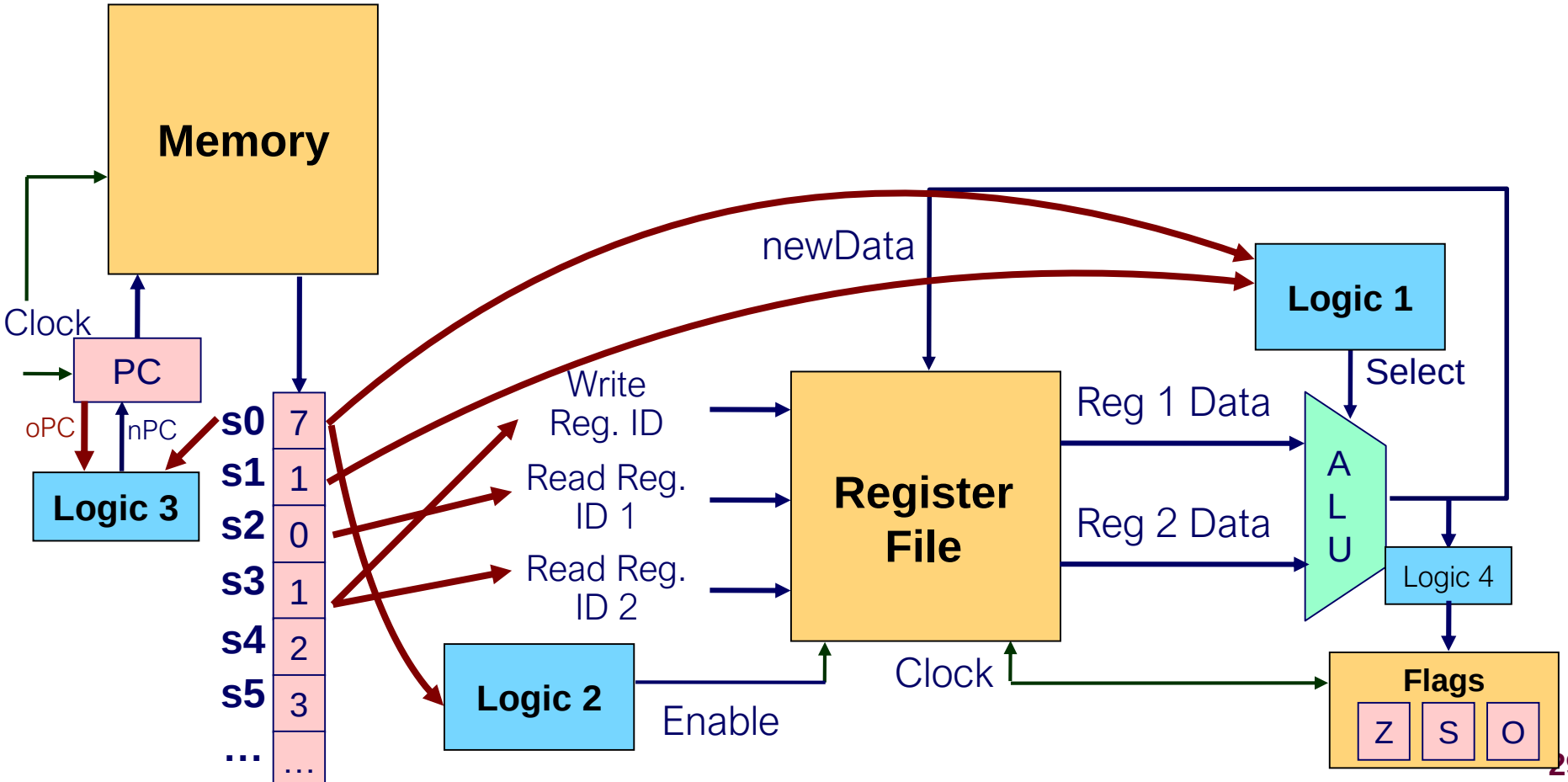
Executing a JLE instruction

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



Executing a JLE instruction

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

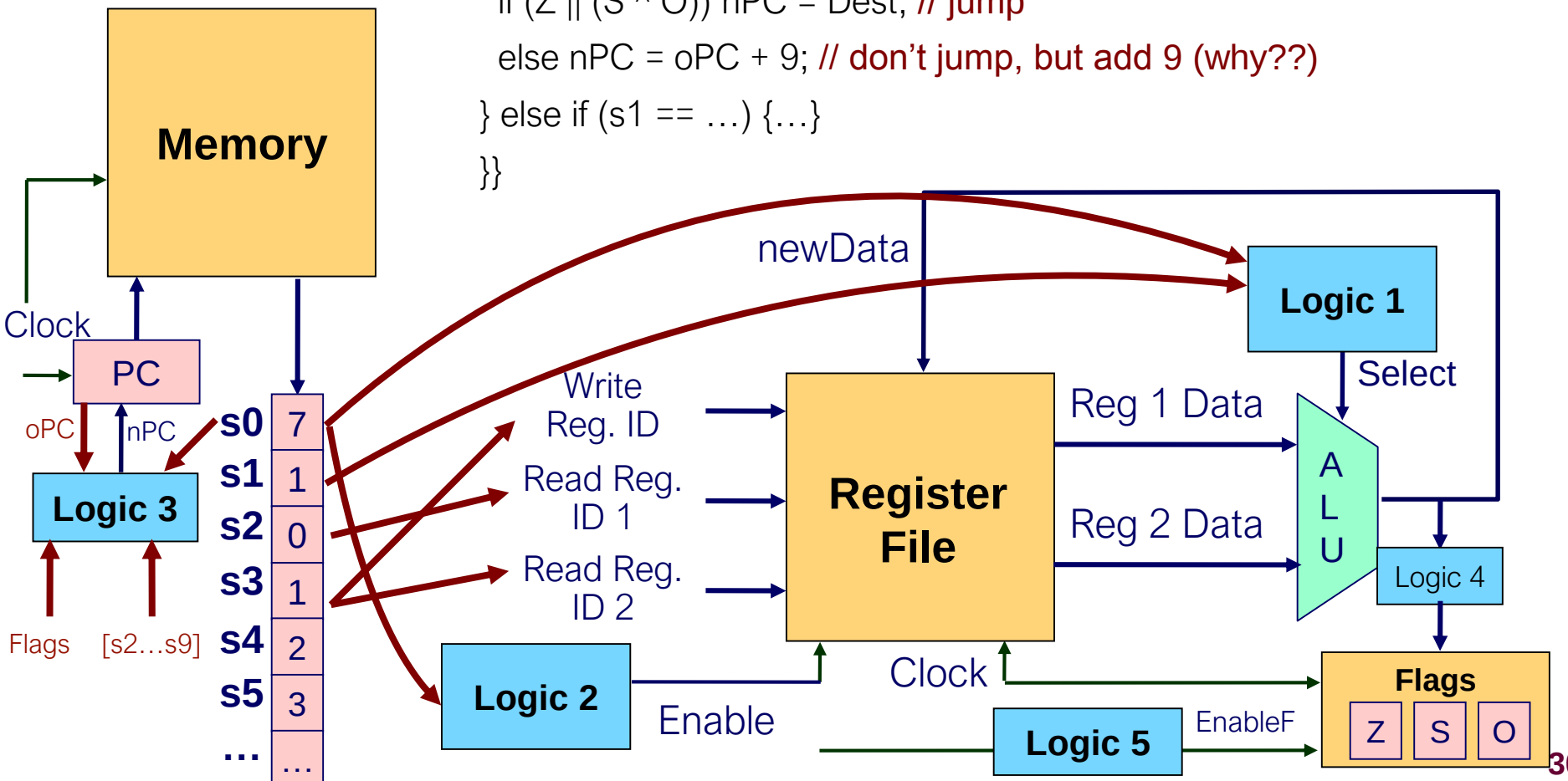


Executing a JLE instruction

jle Dest	7	1	Dest

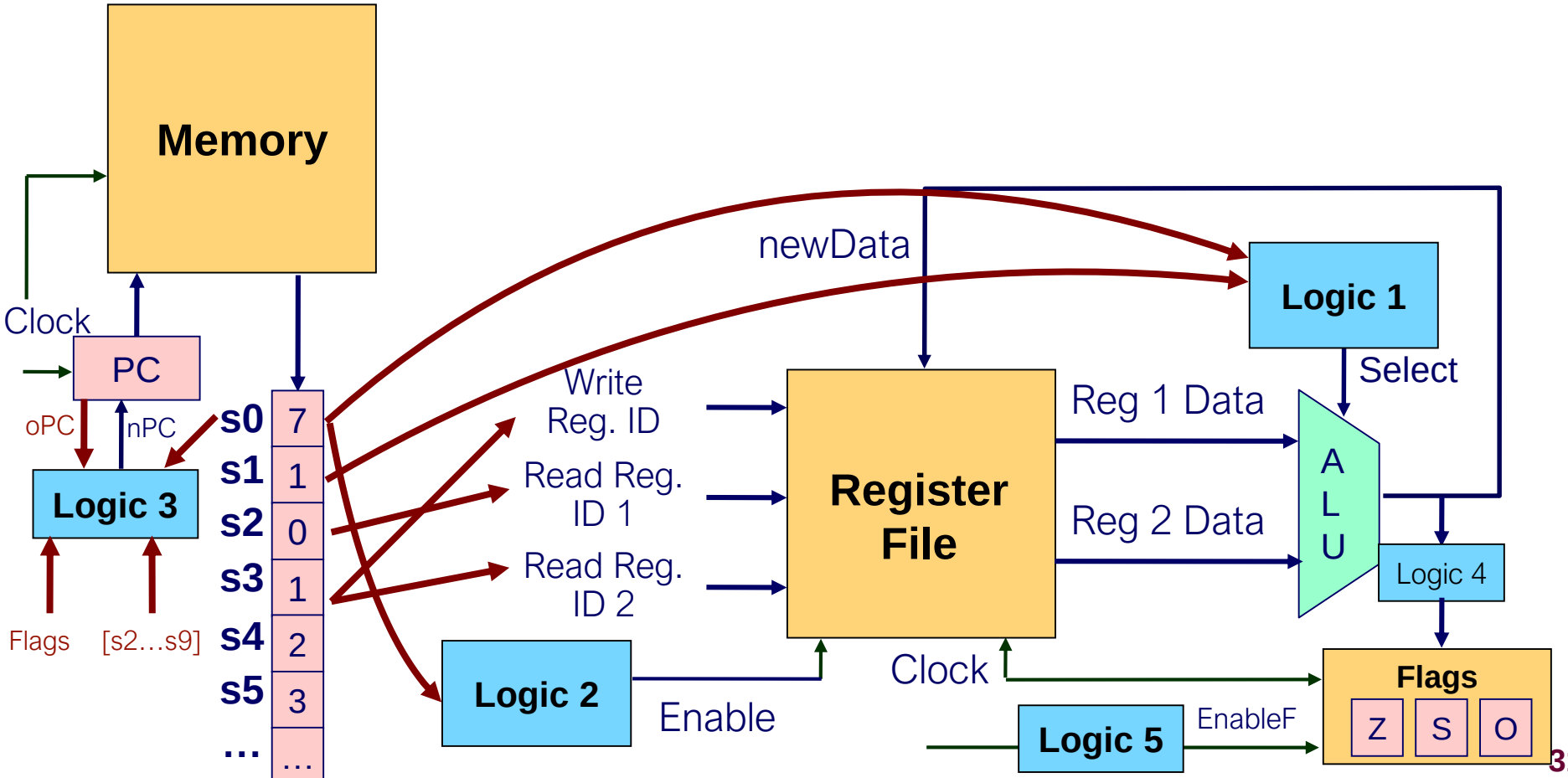
- **Logic 3??**

```
if (s0 == 6) nPC = oPC + 2;
else if (s0 == 7) {
  if (s1 == 1) { // jLE
    if (Z || (S ^ O)) nPC = Dest; // jump
    else nPC = oPC + 9; // don't jump, but add 9 (why??)
  } else if (s1 == …) {…}
  }}
```

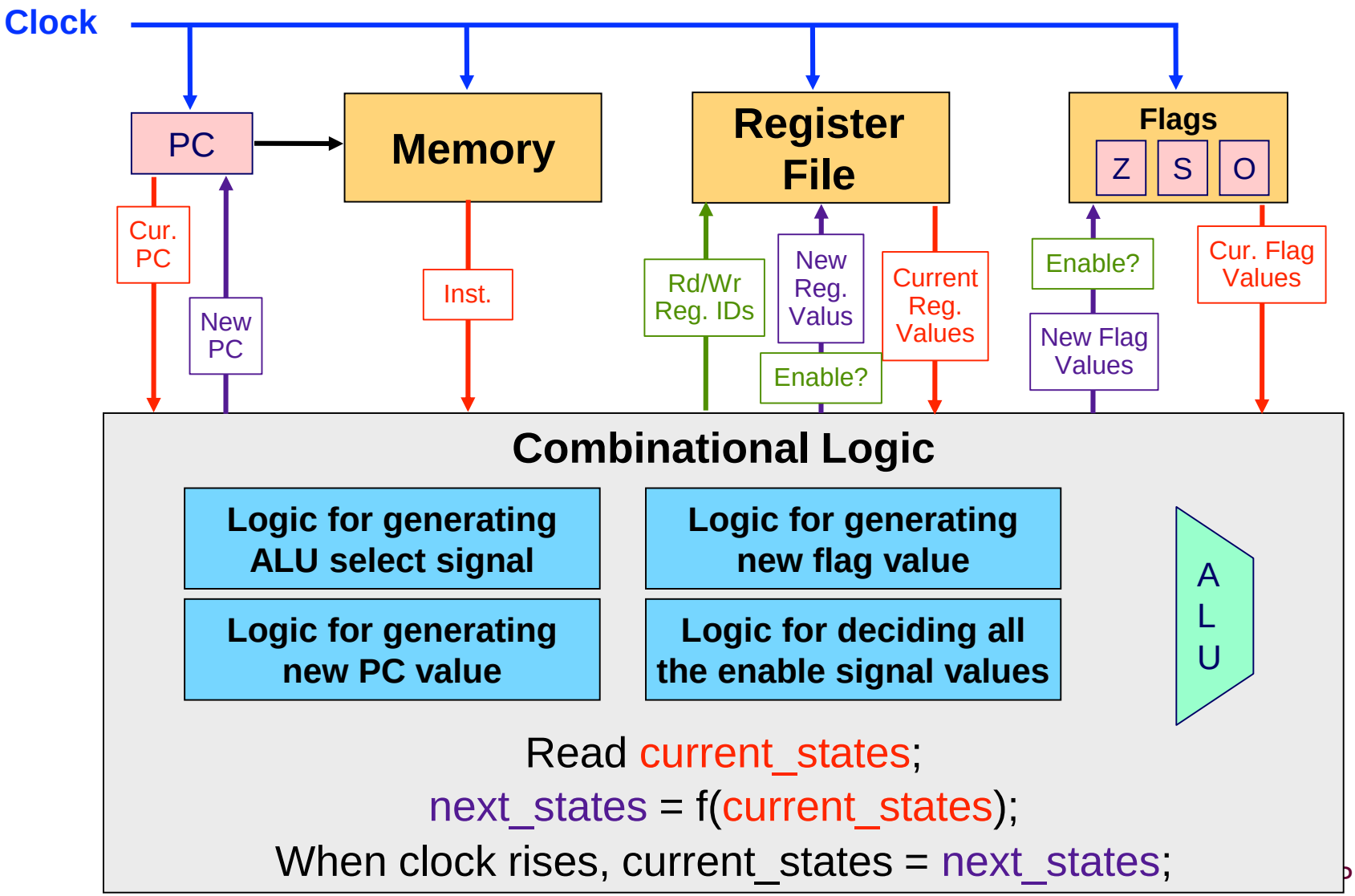


Executing a JLE instruction

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

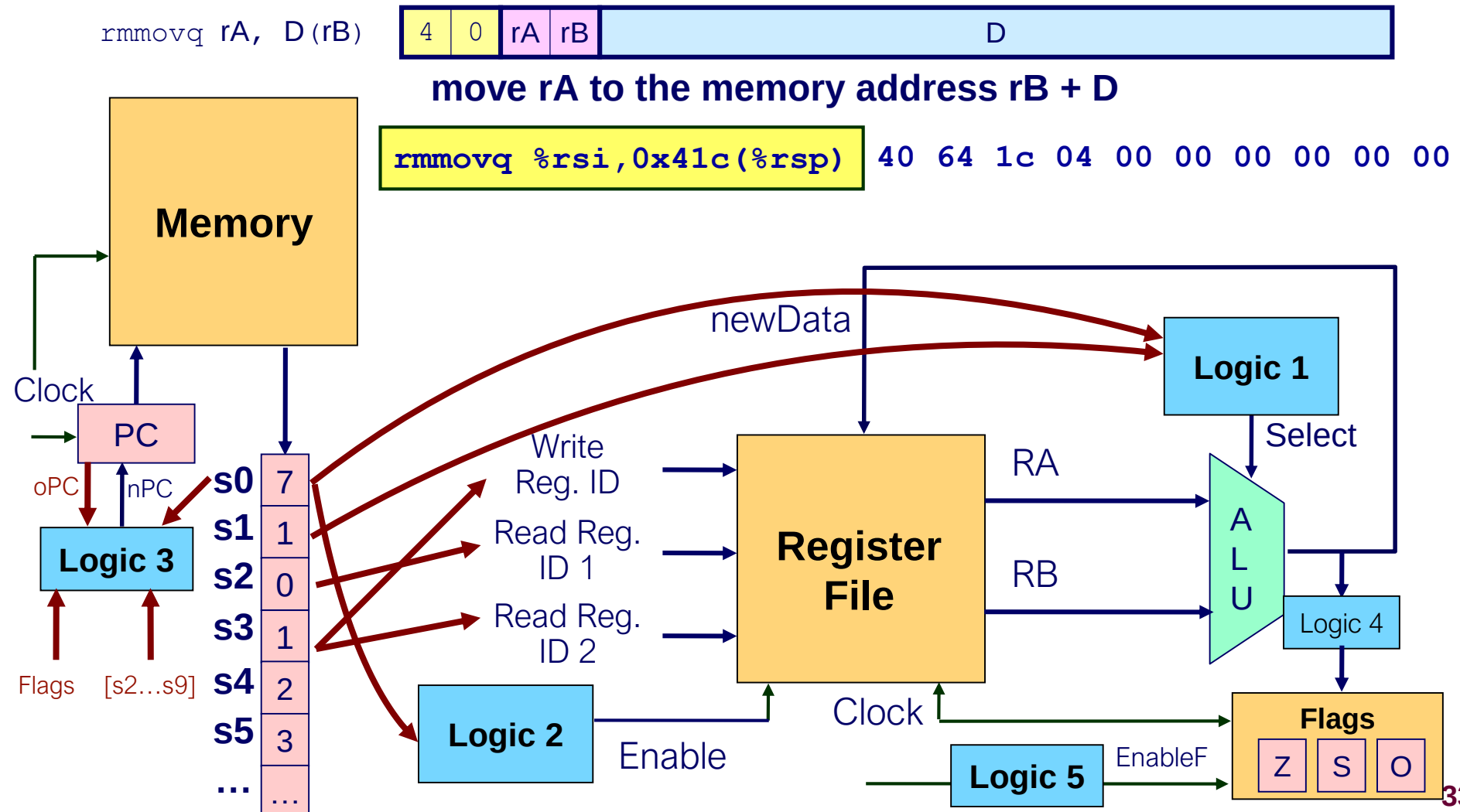


Microarchitecture (So far)



Executing a MOV instruction

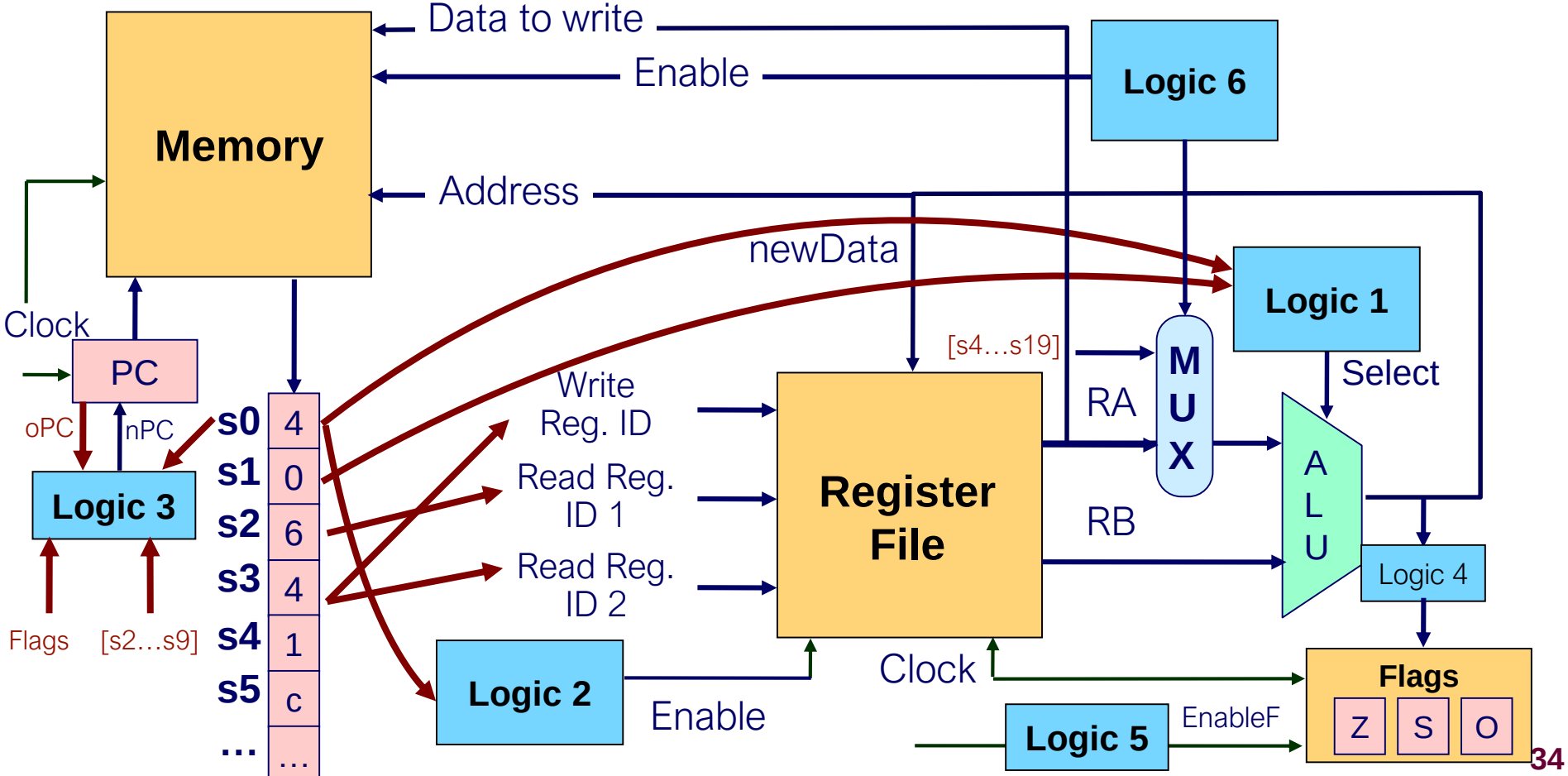
- How do we modify the hardware to execute a move instruction?



move rA to the memory address rB + D

`rmmovq rA, D(rB)`

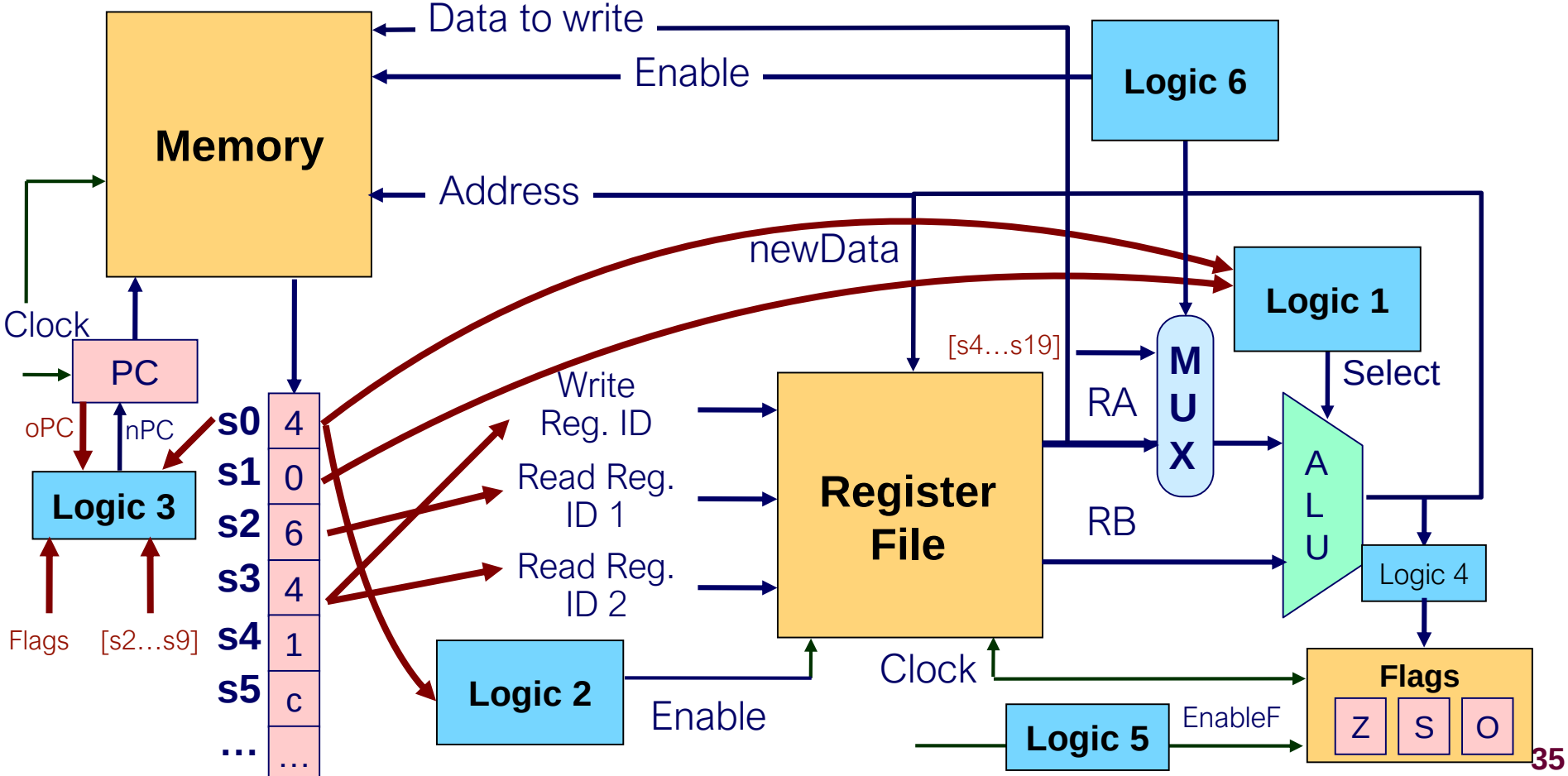
- **Need new logic (Logic 6) to select the input to the ALU for Enable.**
- **How about other logics?**



How About Memory to Register MOV?

move data at memory address rB + D to rA

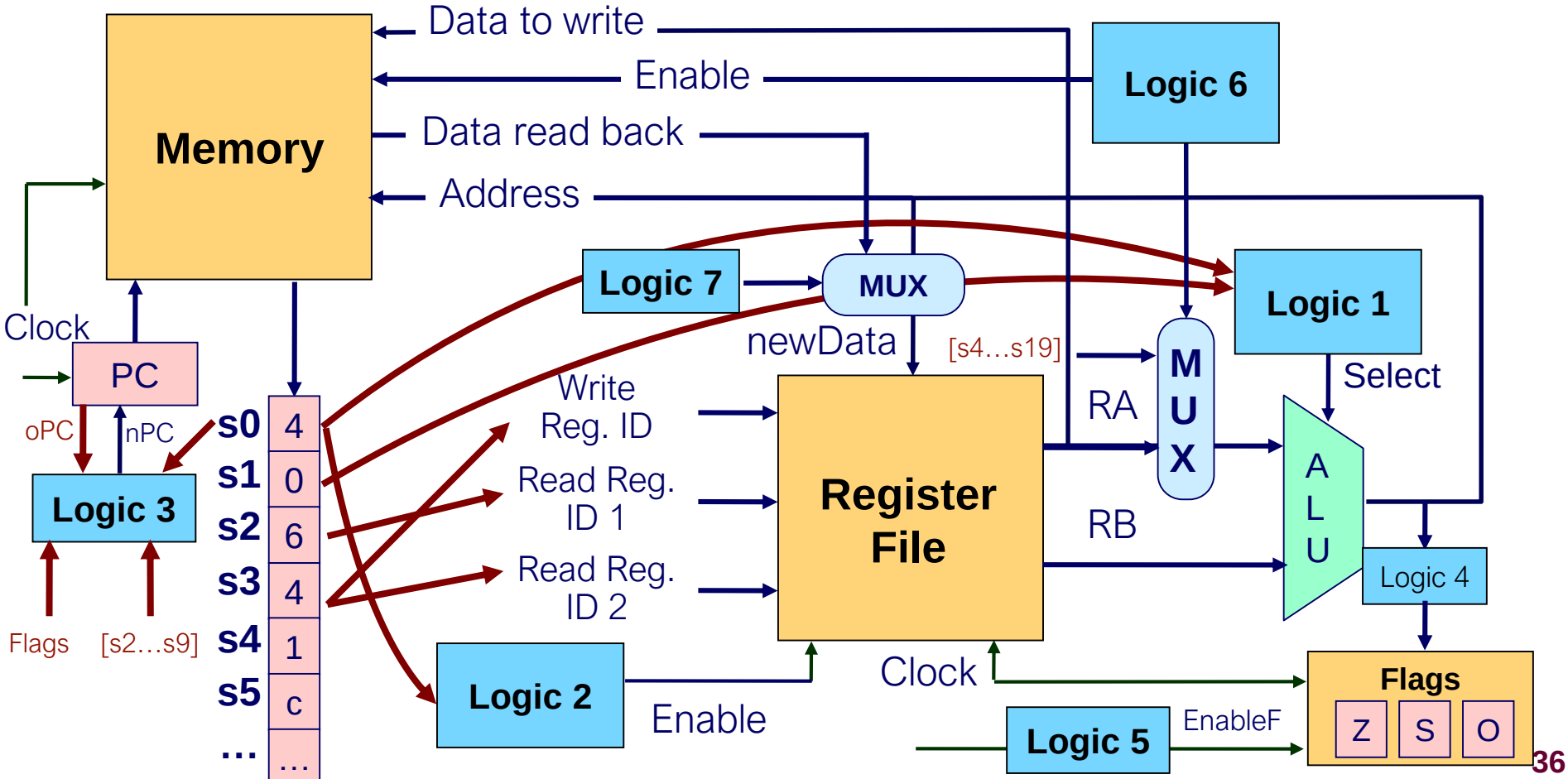
`mrmovq D(rB), rA`



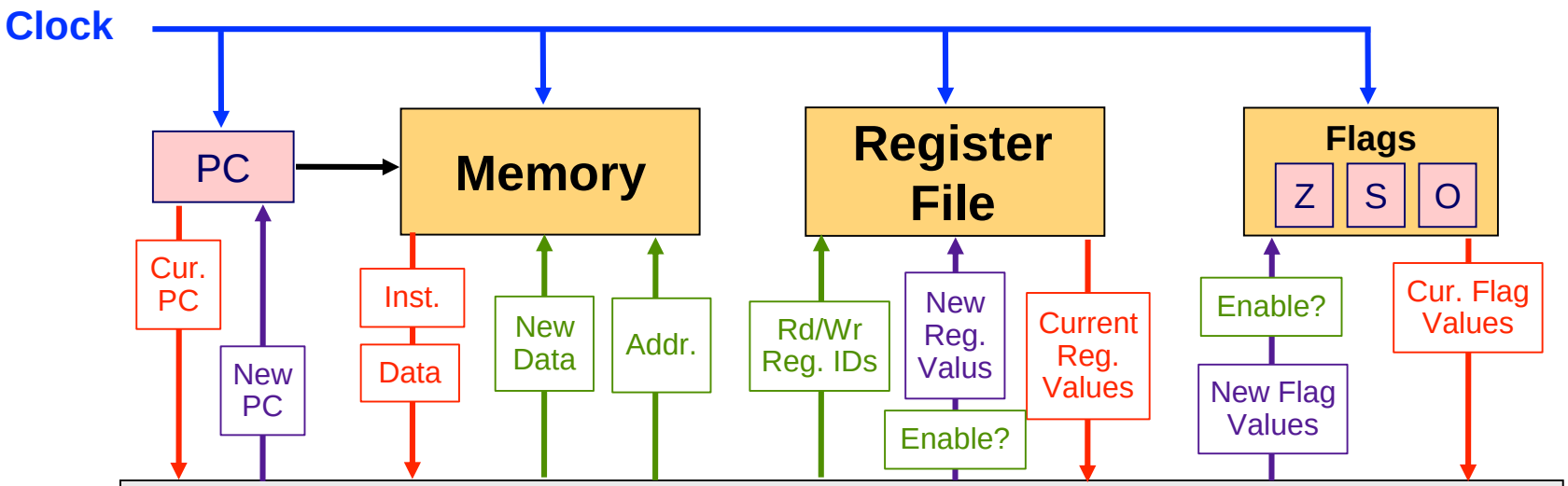
How About Memory to Register MOV?

move data at memory address rB + D to rA

`mrmovq D(rB), rA`



Microarchitecture (with MOV)



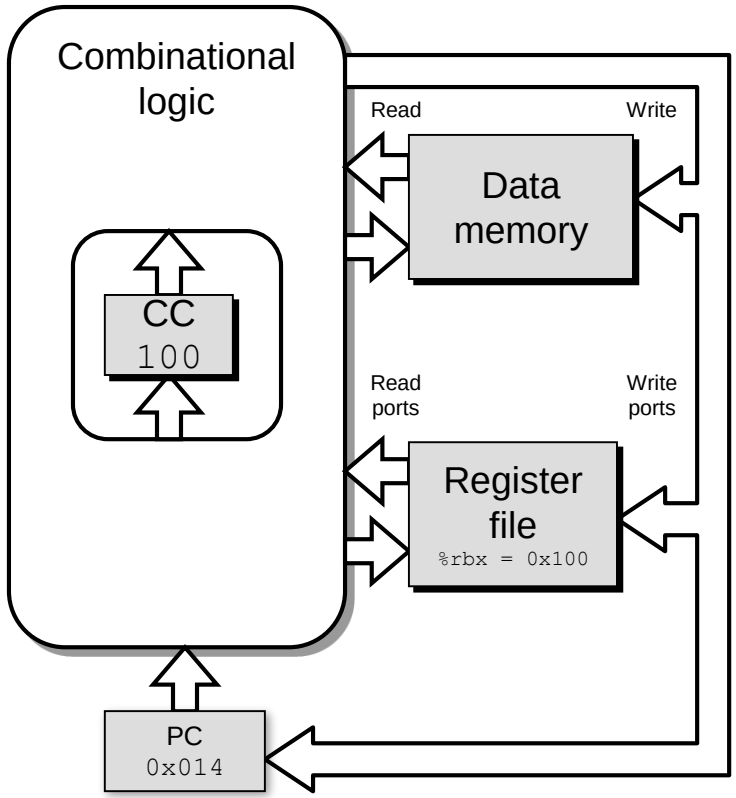
Microarchitecture Overview

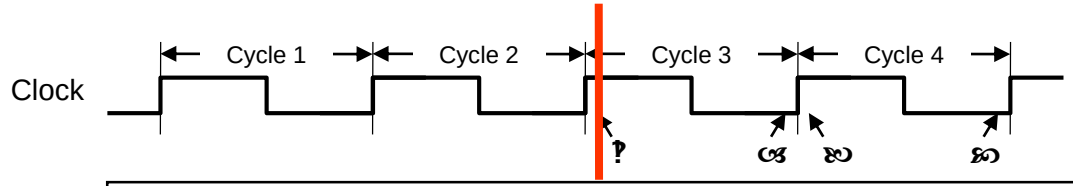
Think of it as a state machine

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

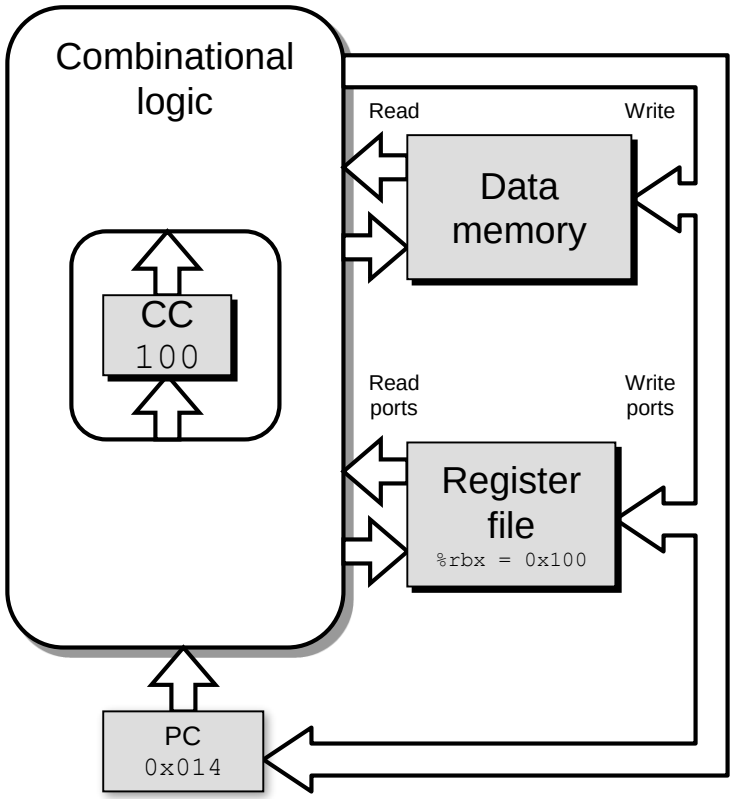
States (All updated as clock rises)

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



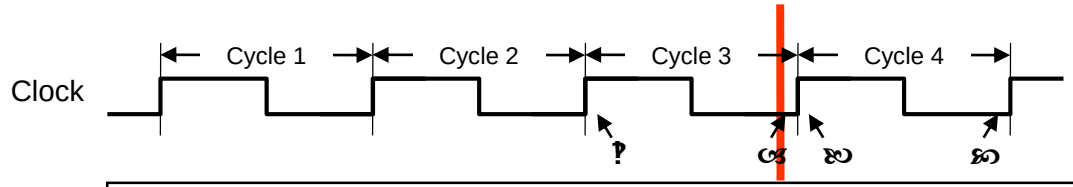


Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

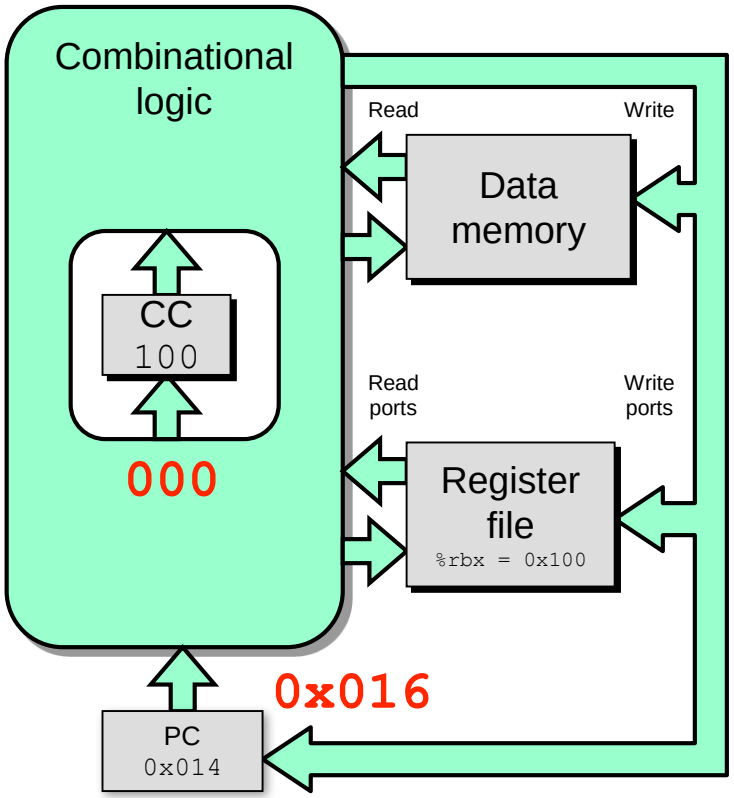


state set according to second **`irmovq`** instruction

combinational logic starting to react to state changes



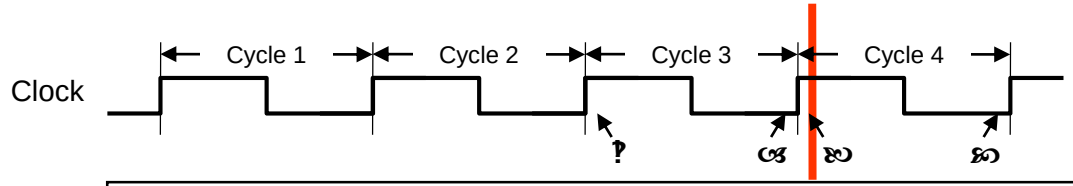
Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



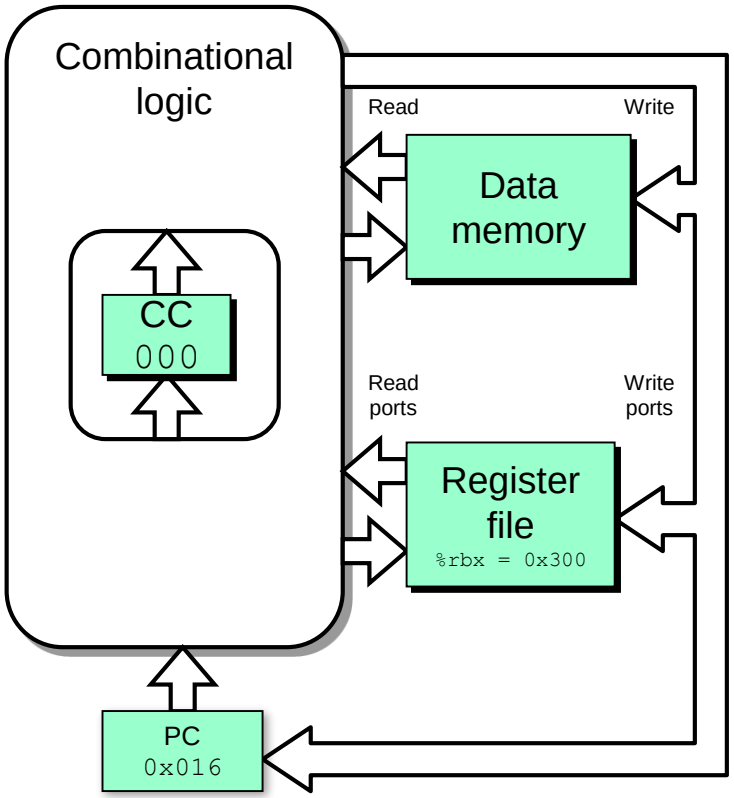
%rbx <-- 0x300

state set according to second **irmovq** instruction

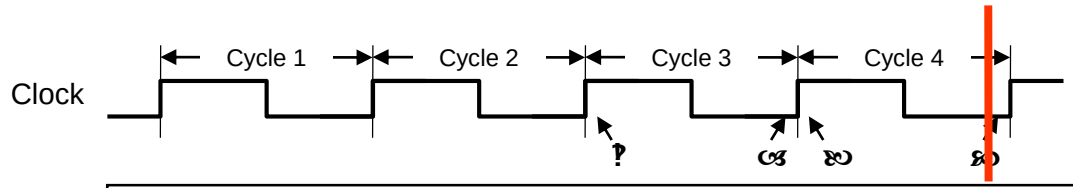
combinational logic generates results for addq instruction



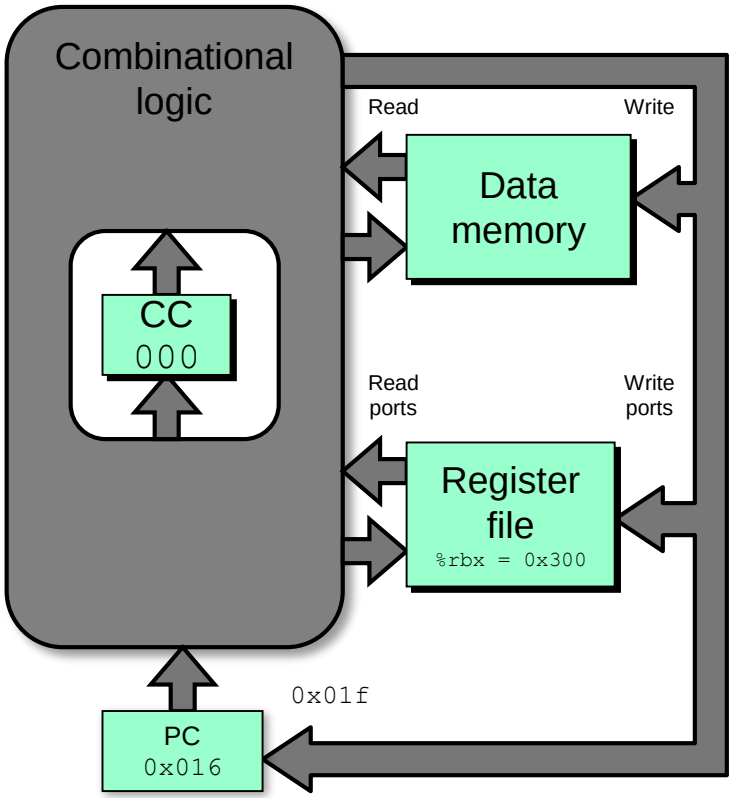
Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



- 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