How are you feeling about `Bits`?

Haven’t started to be honest.

>25% done, feeling good.

>25% done, feeling confused.

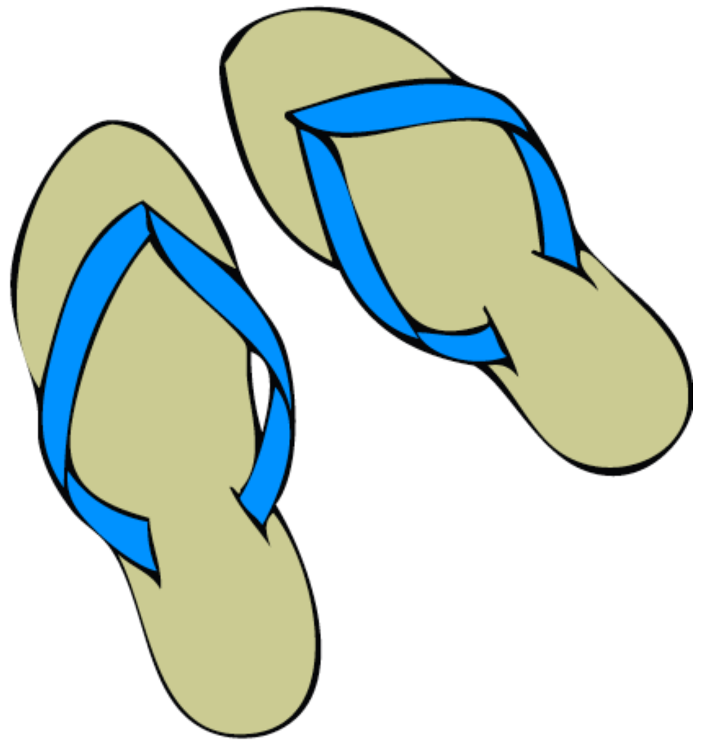
>50% done, feeling good.

>50% done, feeling confused.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Latch: CC-BY Rberteig@flickr

Sequential Logic and State

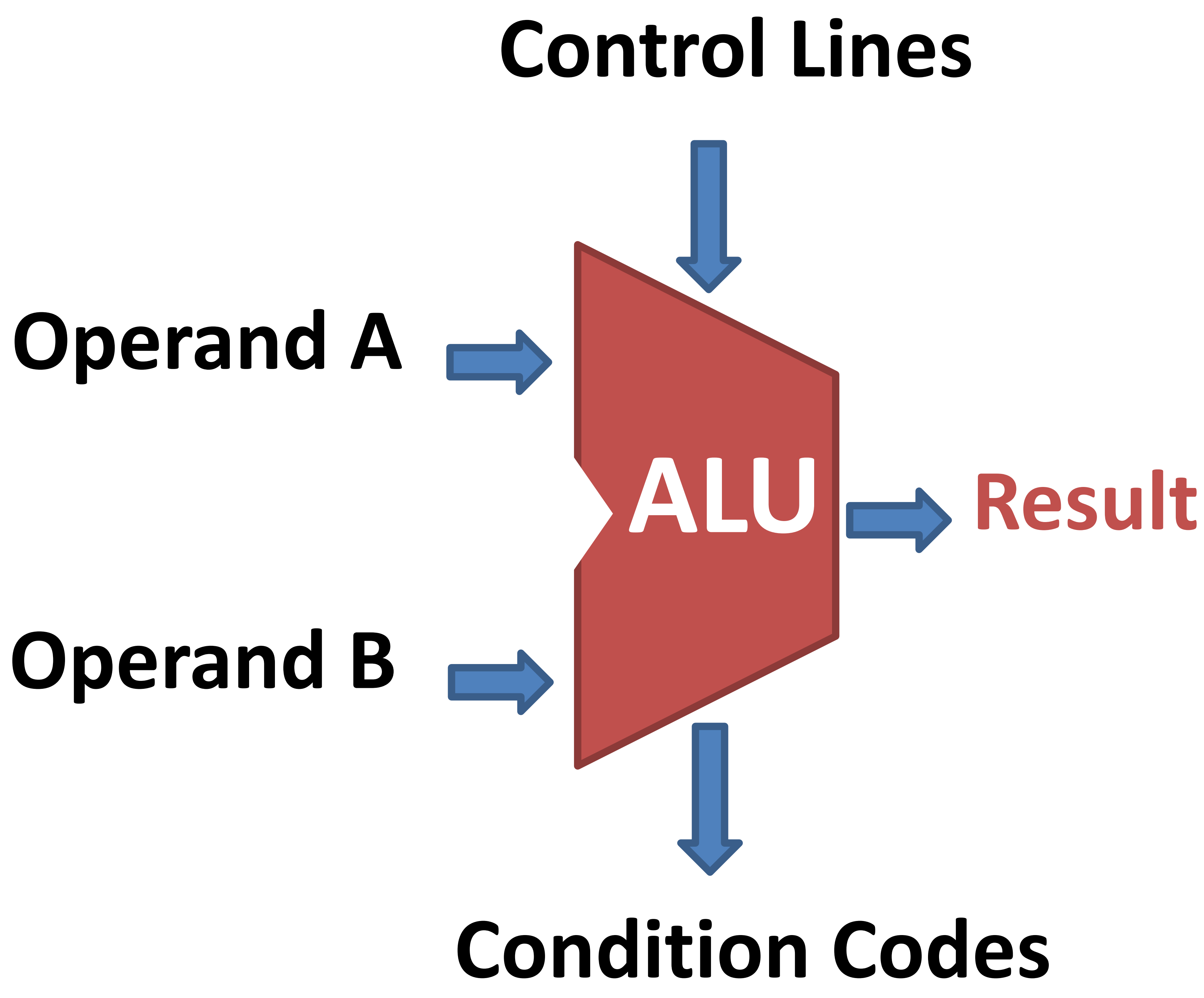


Output depends on inputs *and stored values*.

(vs. combinational: output depends only on inputs)

Elements to store values: latches, flip-flops, registers, memory

Motivation



Now that we have ALUs to perform computations, how do we store the results?

How do we calculate different results over time?

Answer: we need circuits that depend not just on inputs, but also on *prior state*
= Sequential Logic

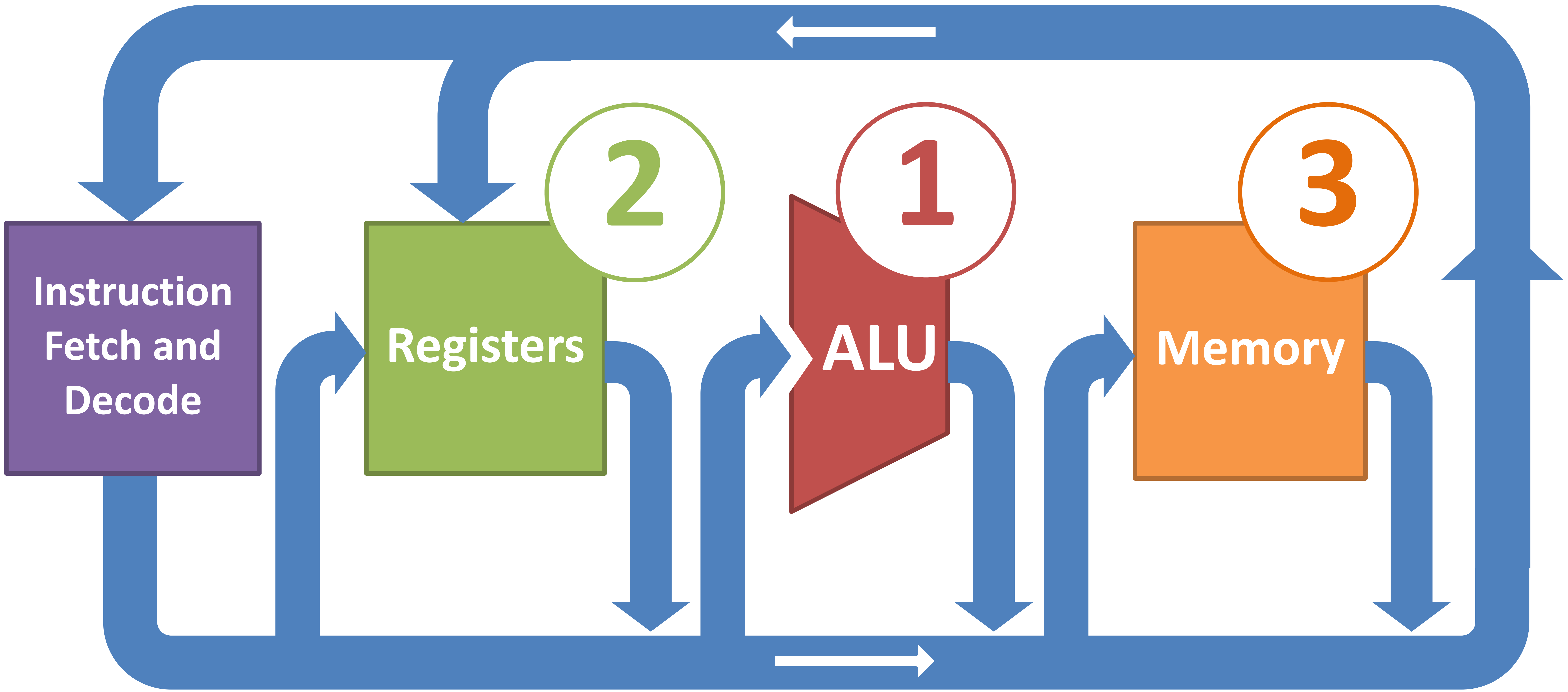
Can you think of an example from lab of a sequential circuit you used?
Hint: previous button pushes are past state.

Nobody has responded yet.

Hang tight! Responses are coming in.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Processor: Data Path Components



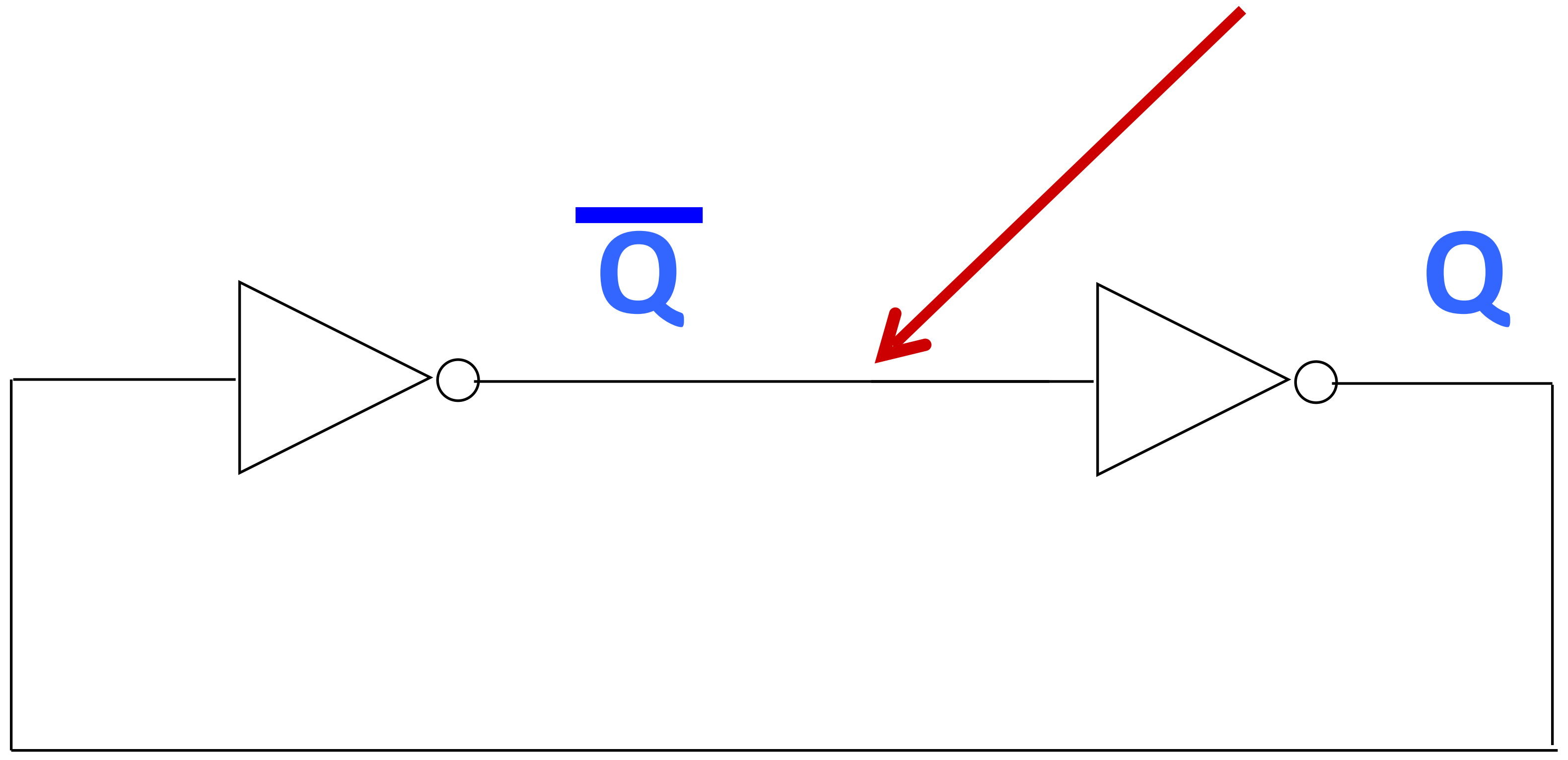
Goal for this section

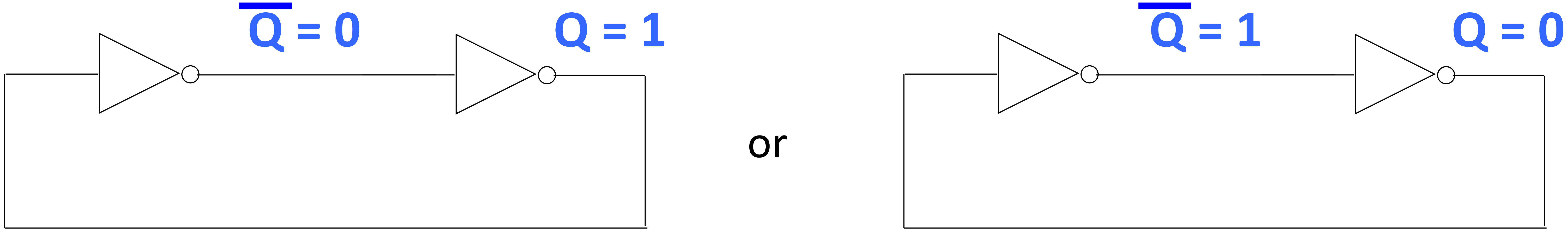
Design a circuit state that **holds** a state over time

- We should be able to set the value to 0 or 1
- We should be able to read the value off the circuit

Building a stable circuit?

Suppose we somehow get a 1 (or a 0?) on here.

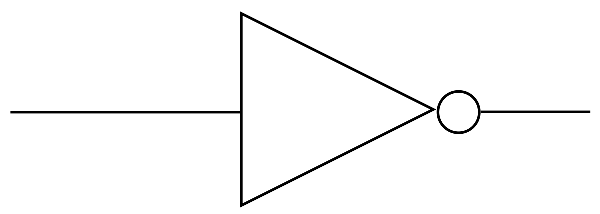




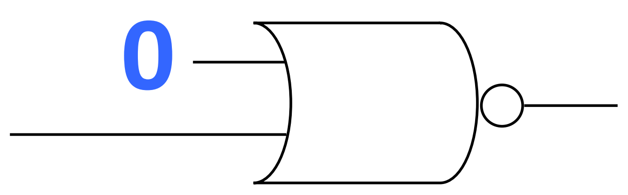
Stable, but how do
we set the value?

Bistable latches

Recall from Gates:

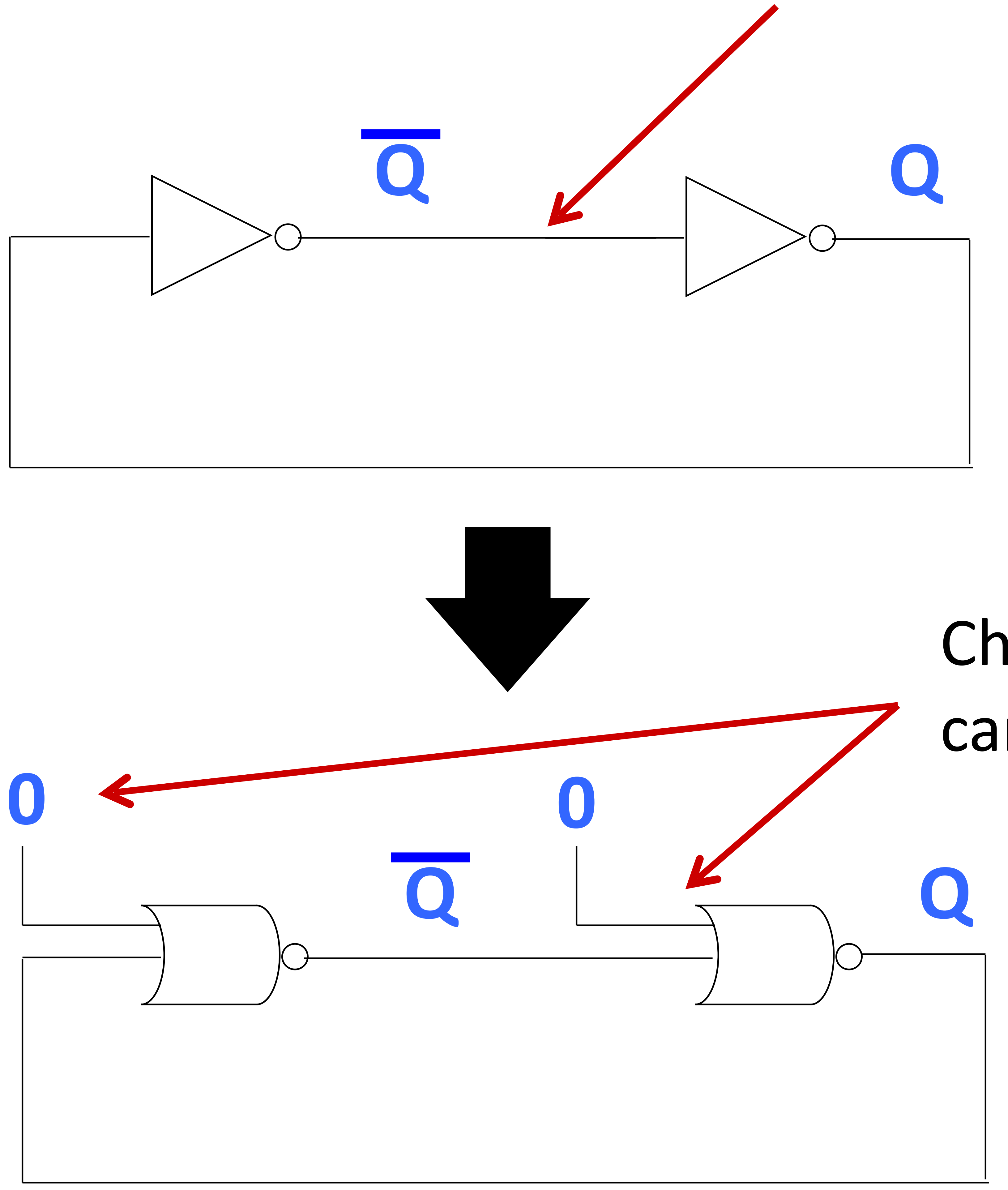


is equivalent to:



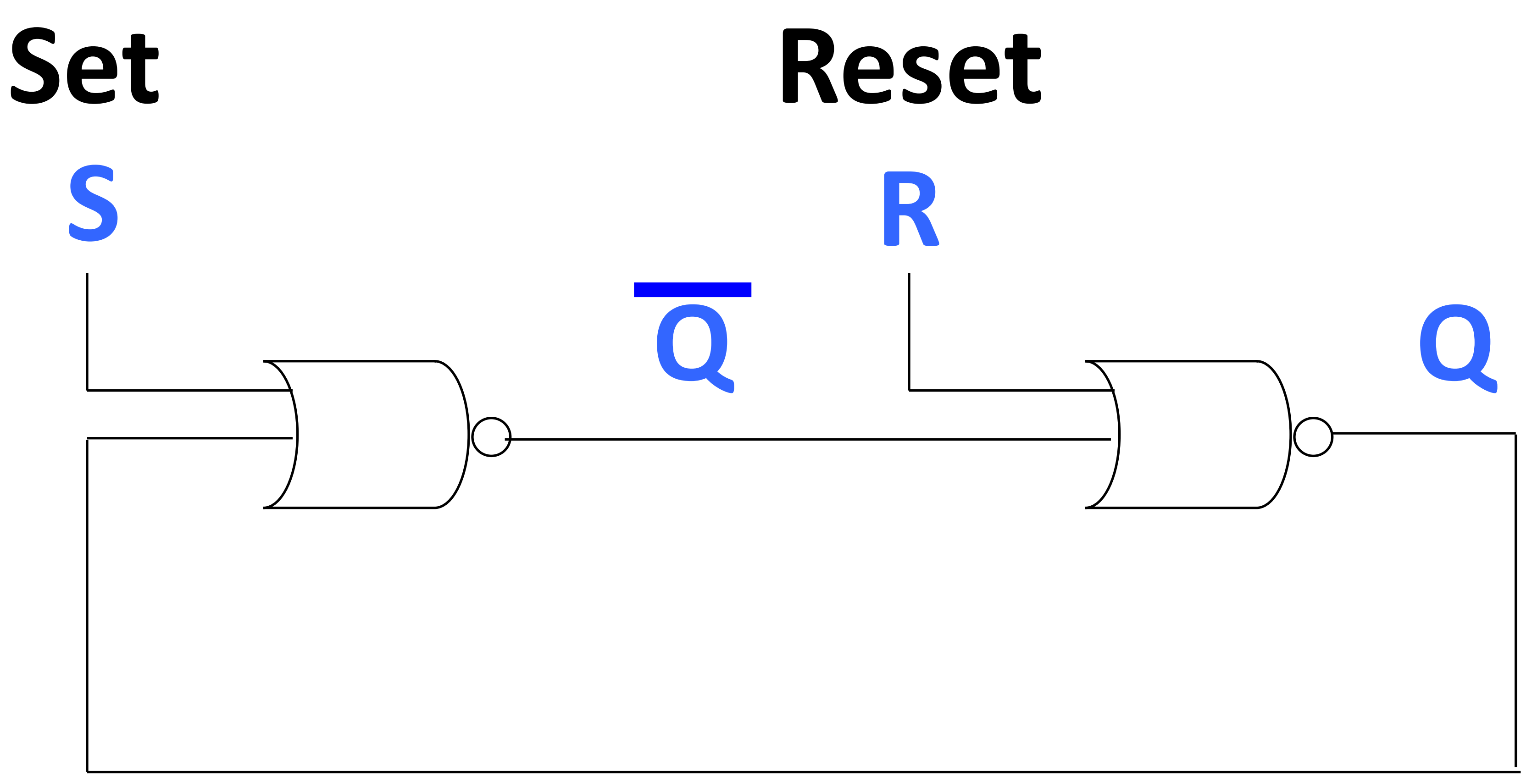
Stable, but how do we set the value?

Suppose we somehow get a 1 (or a 0?) on here.



Change to a 2-input gates so that we can **set** updated values to be stored

SR latch

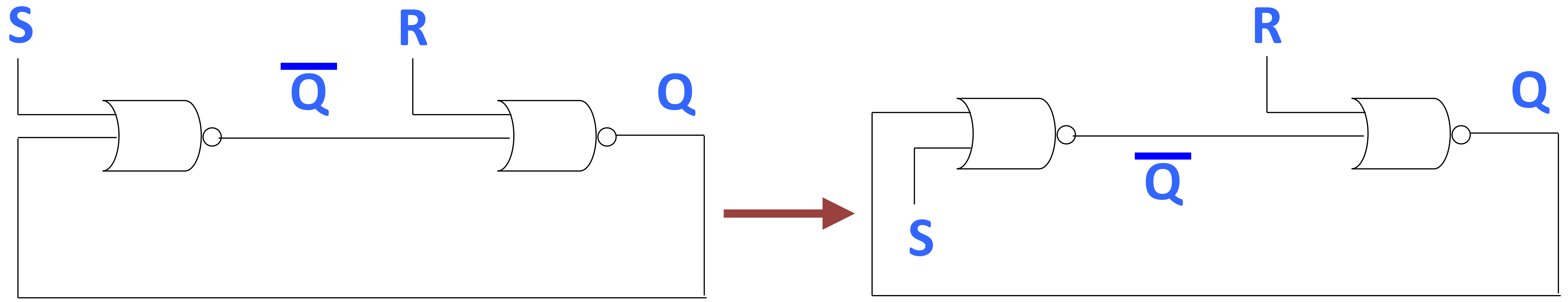


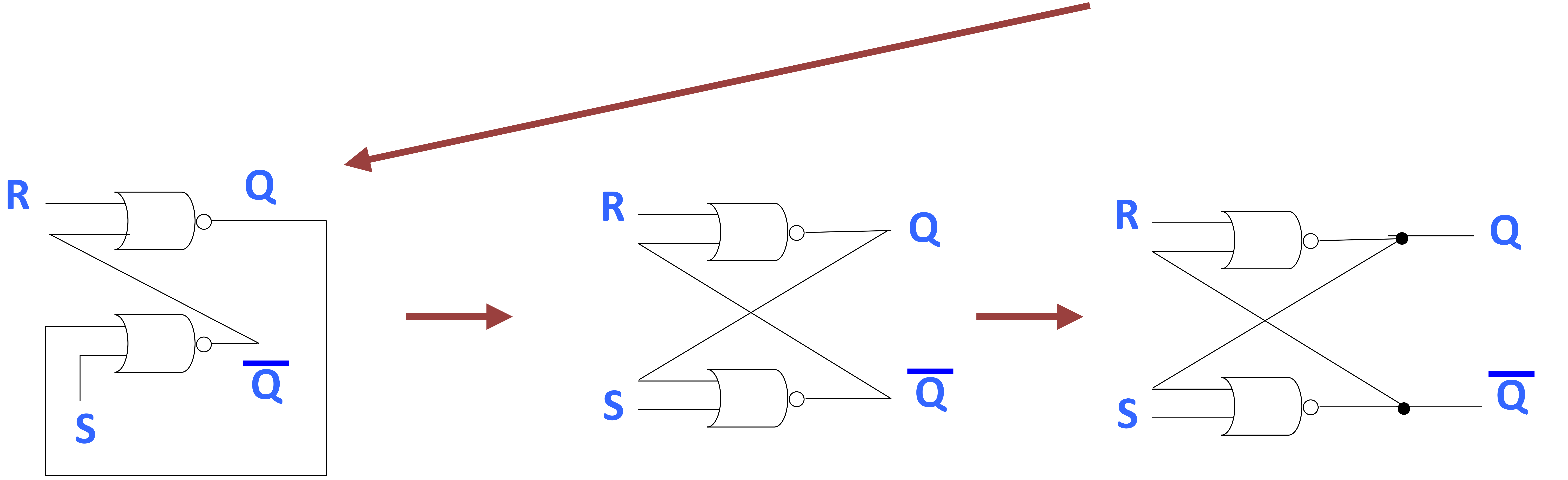
S	R	Q_{prev}	Q'_{prev}	Q_{next} (stable)	Q'_{next} (stable)
0	0	0	1		
0	0	1	0		
1	0	*any*	*any*		
0	1	*any*	*any*		
1	1	*any*	*any*		

← Violates invariant that Q and Q' are inverses!

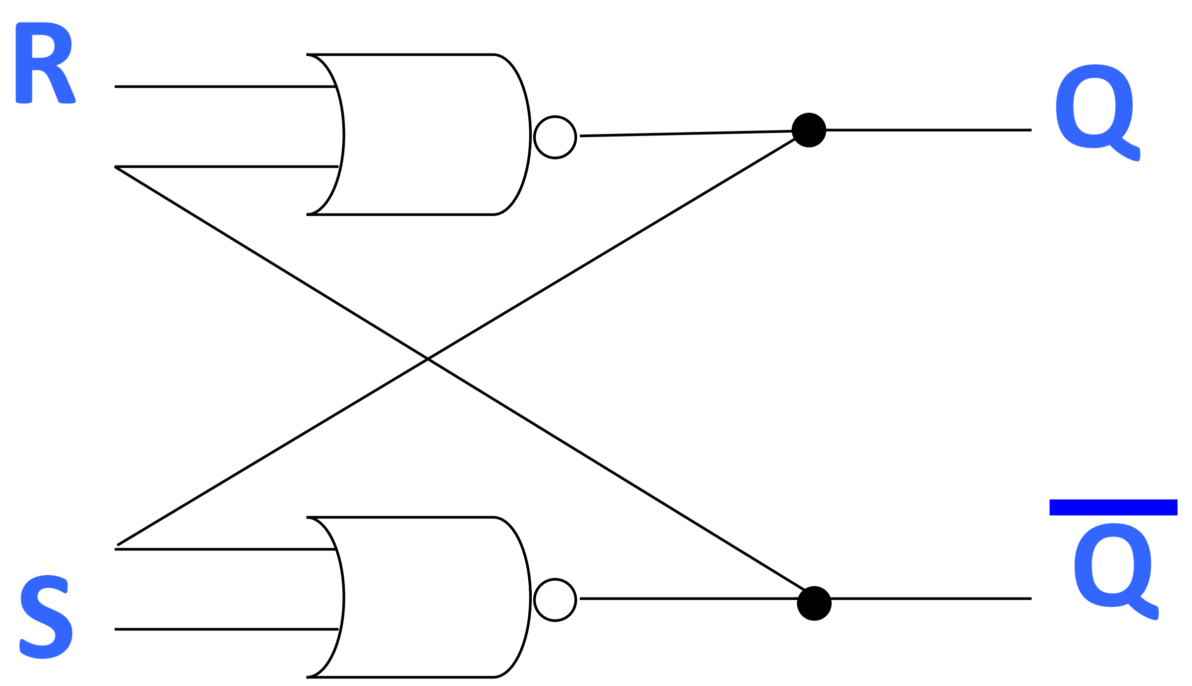
SR latch

Move from the circuit we built to the canonical form





SR latch



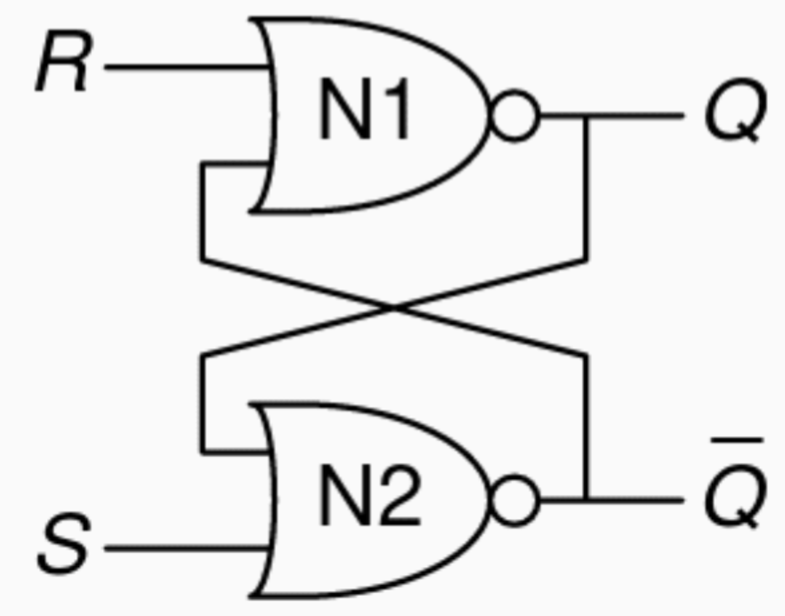


Figure 3.3 SR latch schematic

Meets our goals:

- Able to set the value to 0 or 1
- Able to read the value off the circuit

How do we set Q to 1?

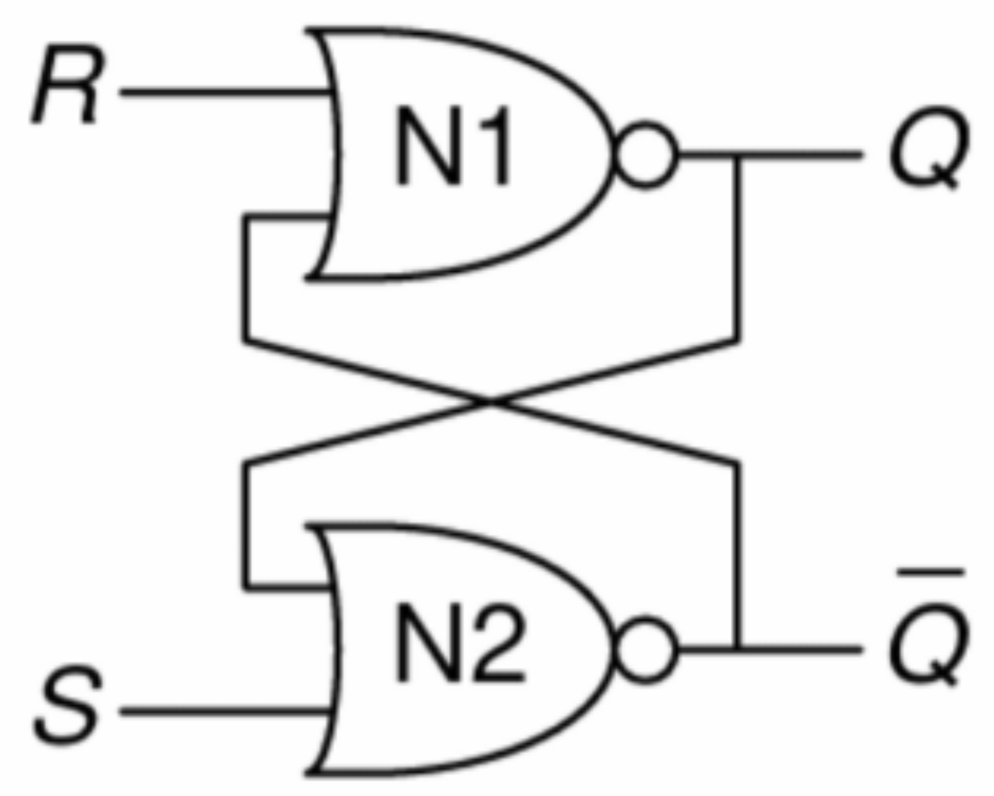
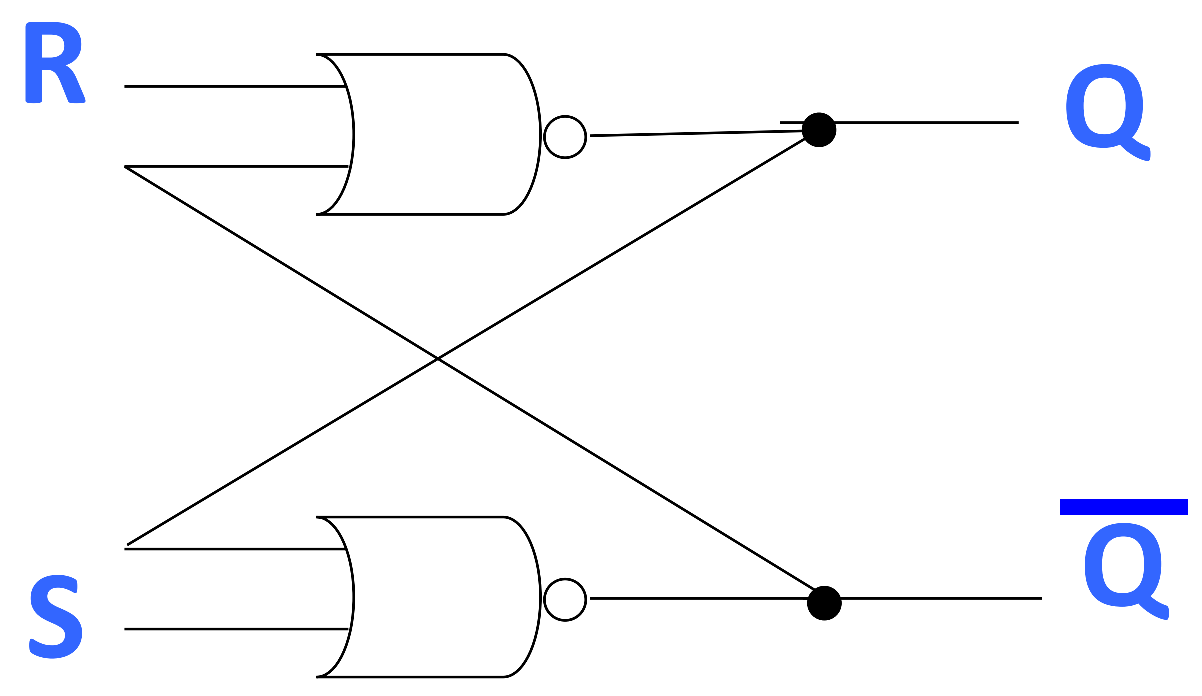


Figure 3.3 SR latch schematic

- S = 0; R = 0
- S = 1; R = 0
- S = 0; R = 1
- S = 1; R = 1
- None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

SR latch



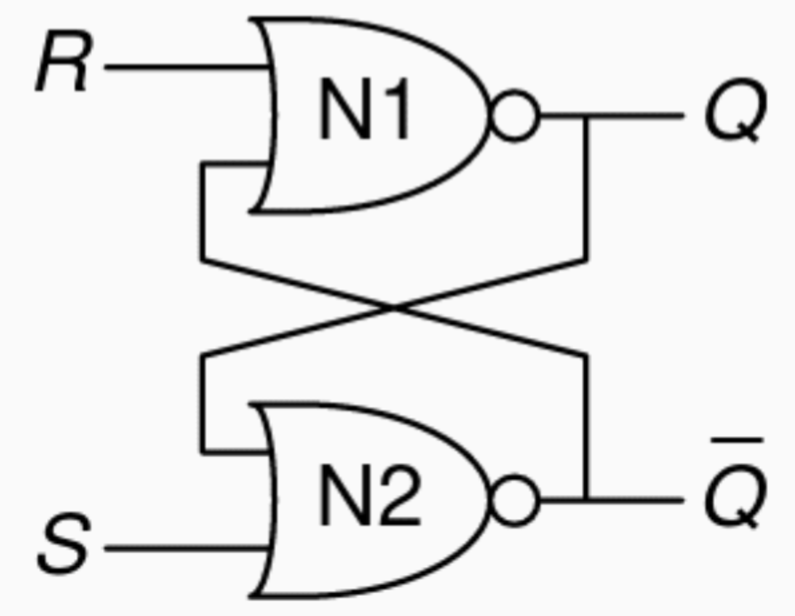


Figure 3.3 **SR latch schematic**

Meets our goals:

- Able to set the value to 0 or 1
- Able to read the value off the circuit

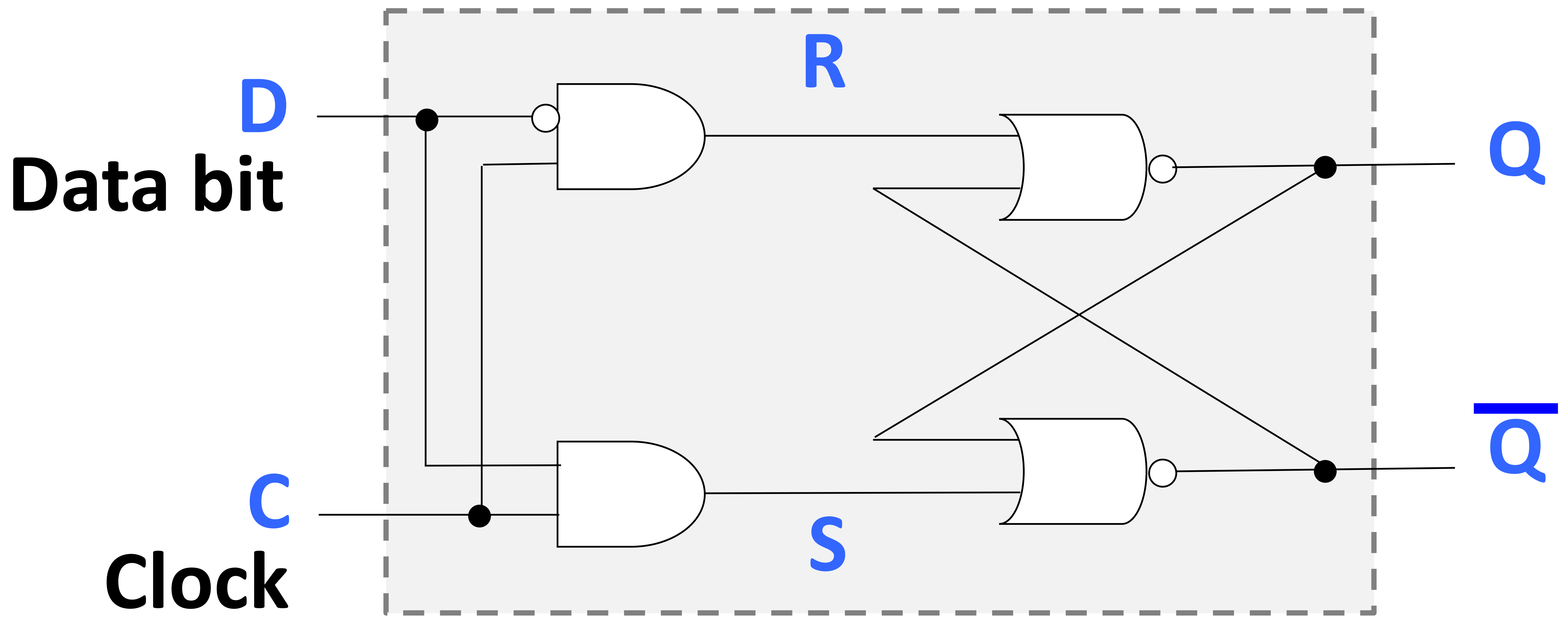
But:

- Broken invariant when S = 1 and R = 1
- No distinction between new value and timing

D latch

Goals:

- No invalid states
- Only 1 bit for data
- Control over timing



if C = 0, then SR latch stores current value of Q.

if C = 1, then D flows to Q:

    if D = 0, then R = 1 and S = 0, Q = 0

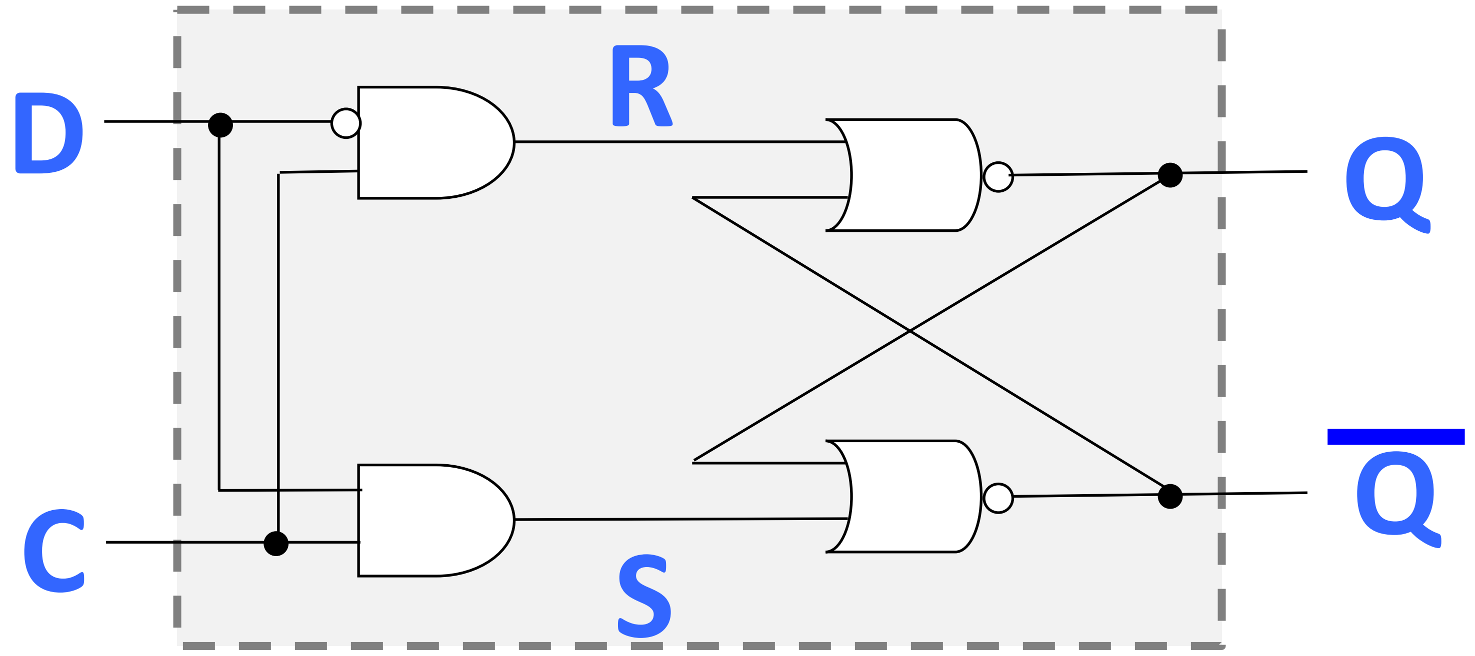
    if D = 1, then R = 0 and S = 1, Q = 1

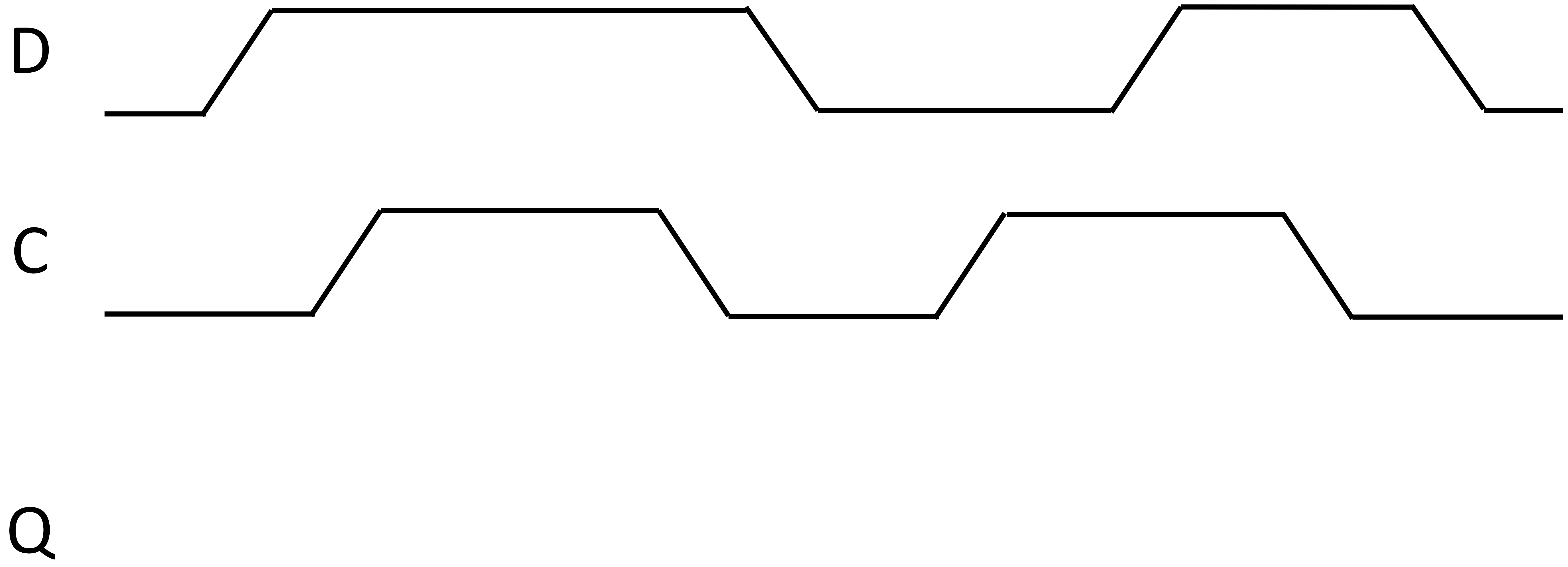
Notes:

- Data bit D replaces S & R: it's the bit value we want to store when Clock = 1
  - Internally, Data bit D prevents bad case of S = R = 1
- This logic is **level-triggered**; as long as Clock = 1, changes to D flow to outputs

Time matters!

Transparent D latch





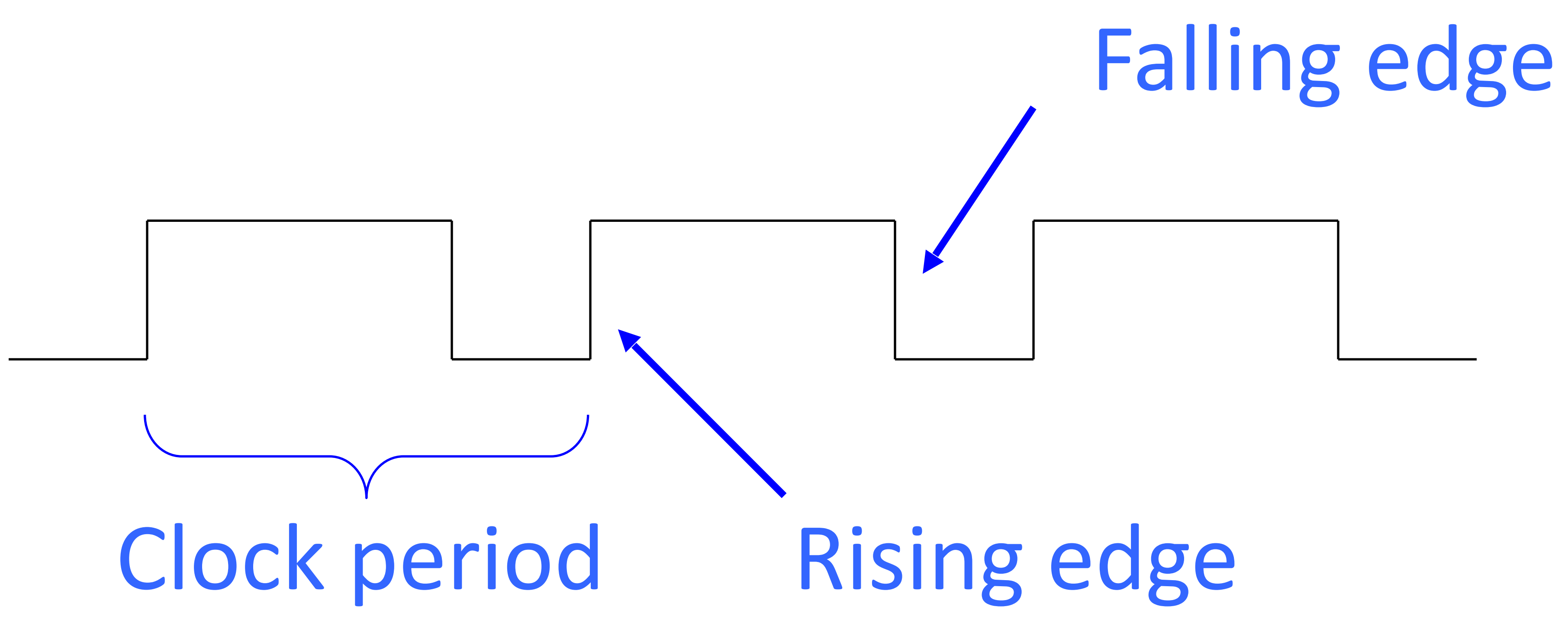
Assume Q has an initial state of 0

In general: clocks

Clock: free-running signal

with fixed cycle time = clock period = T.

Clock frequency = 1 / clock period



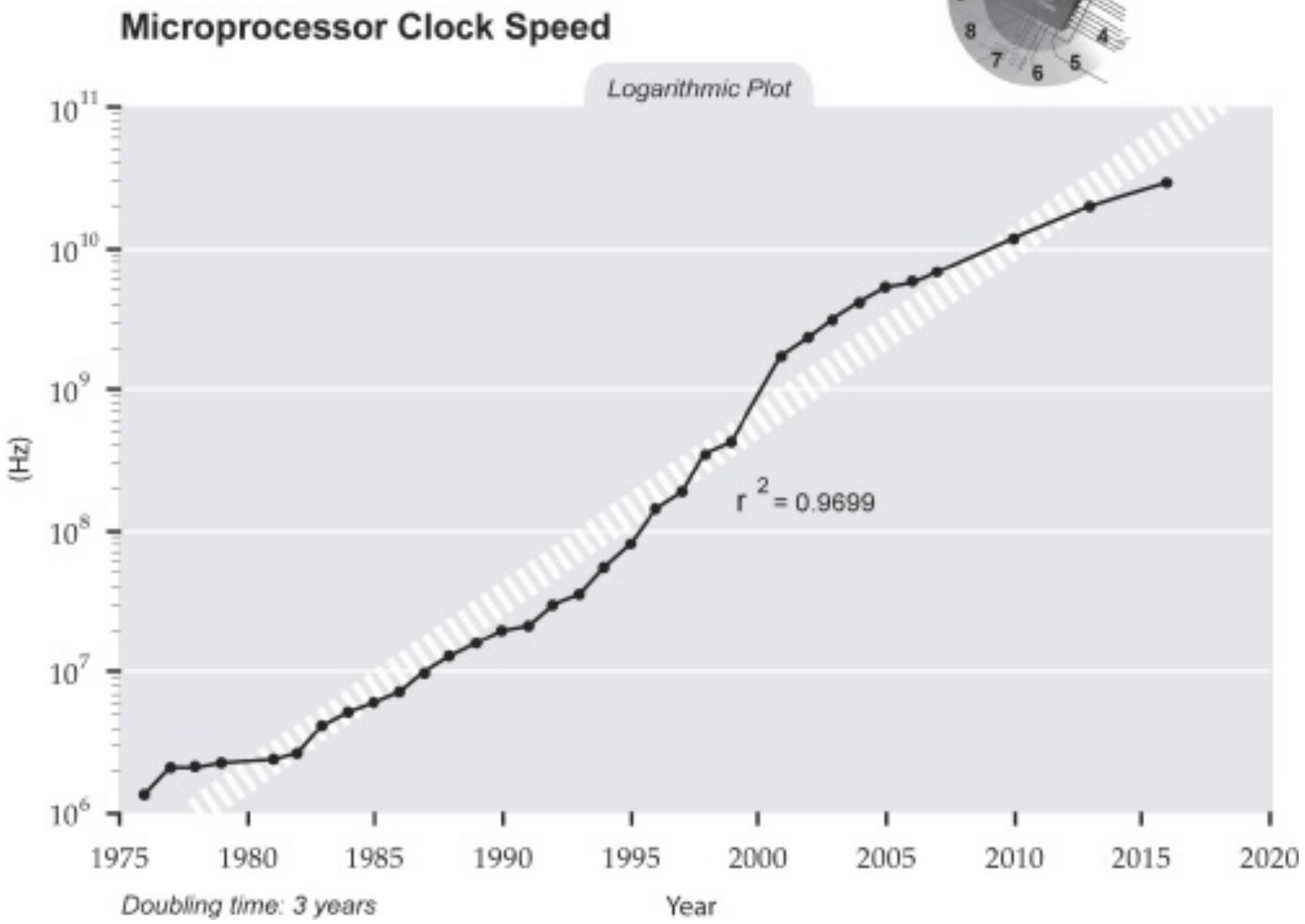
A clock controls when to update a sequential logic element's state.

Real CPU context: “Clock frequency”

Clock frequency

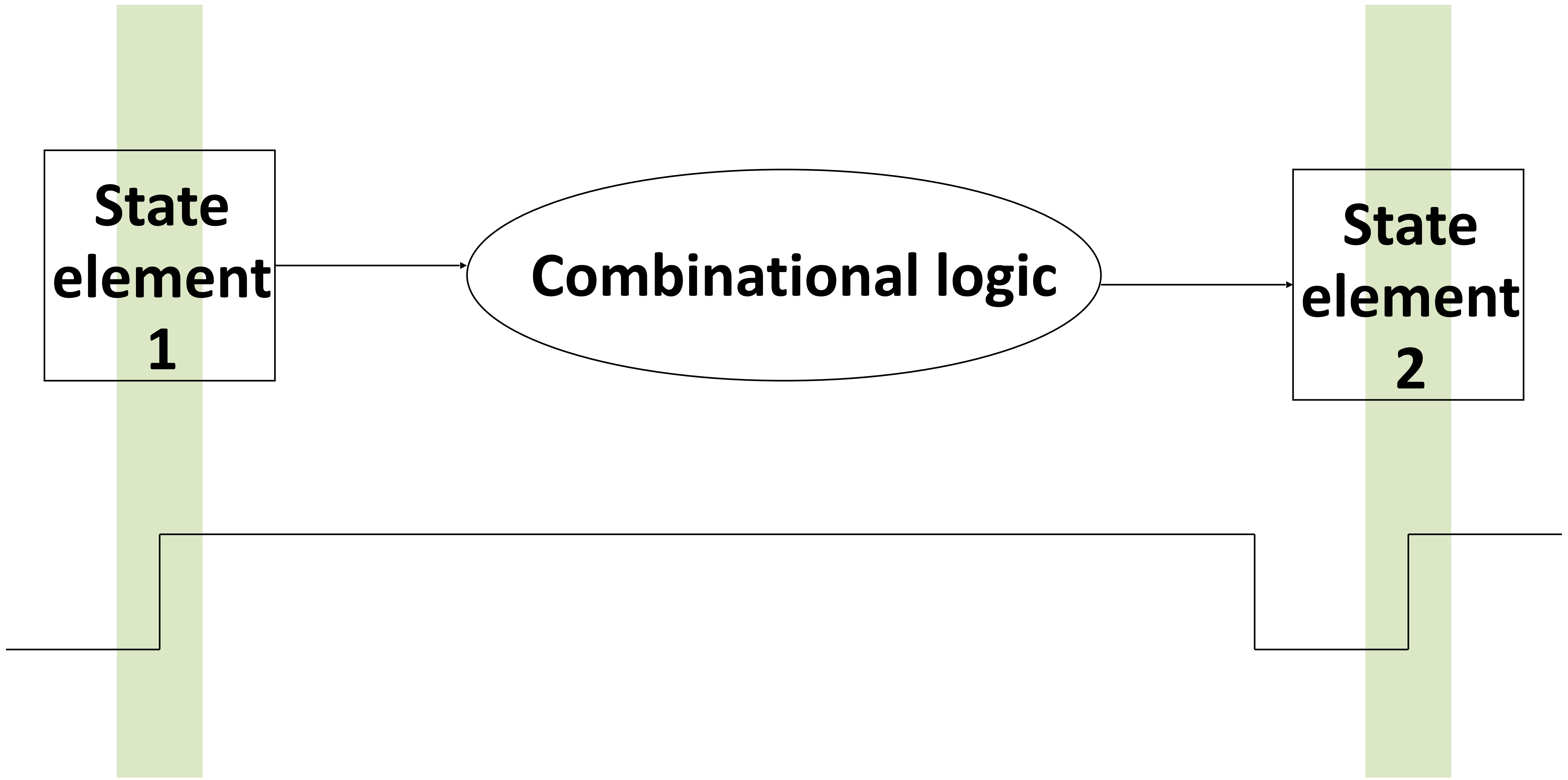
= 1 / period = 1 / s = Hz

Typical CPU: 3-4 GHz

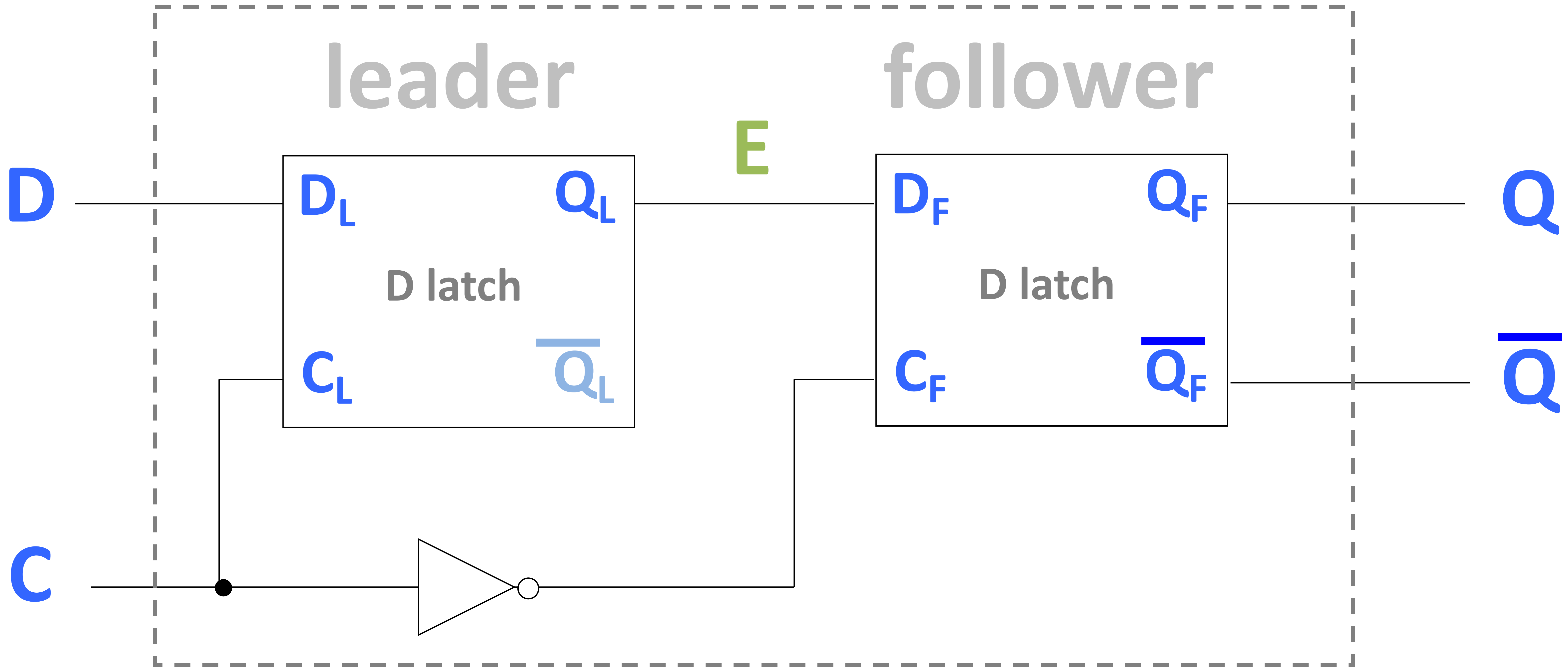


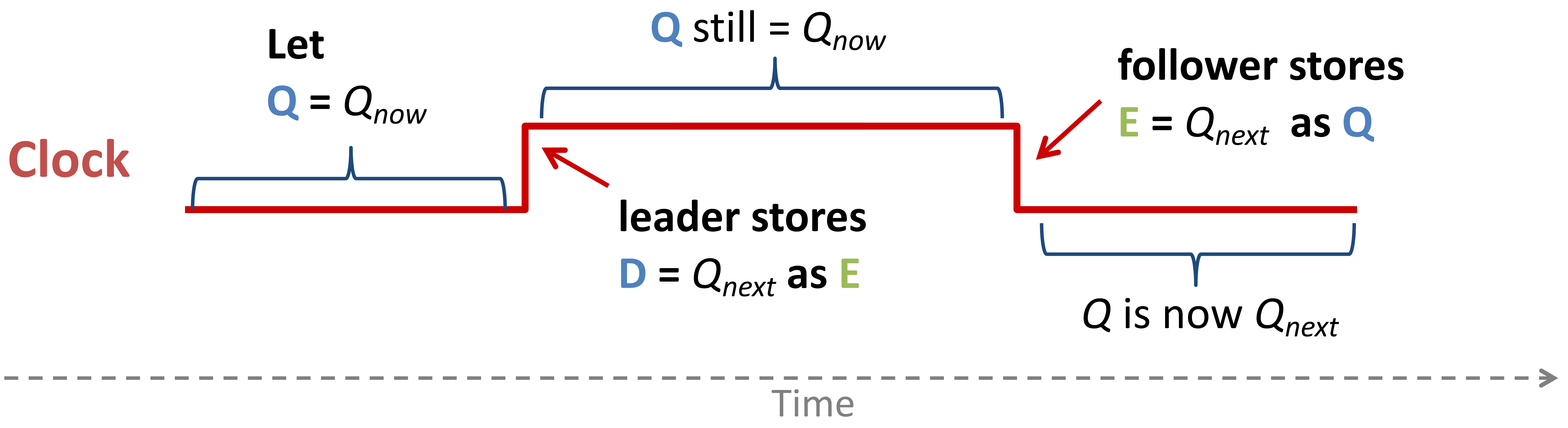
Synchronous systems

Inputs to state elements must be valid on active clock edge.



D flip-flop with *falling*-edge trigger

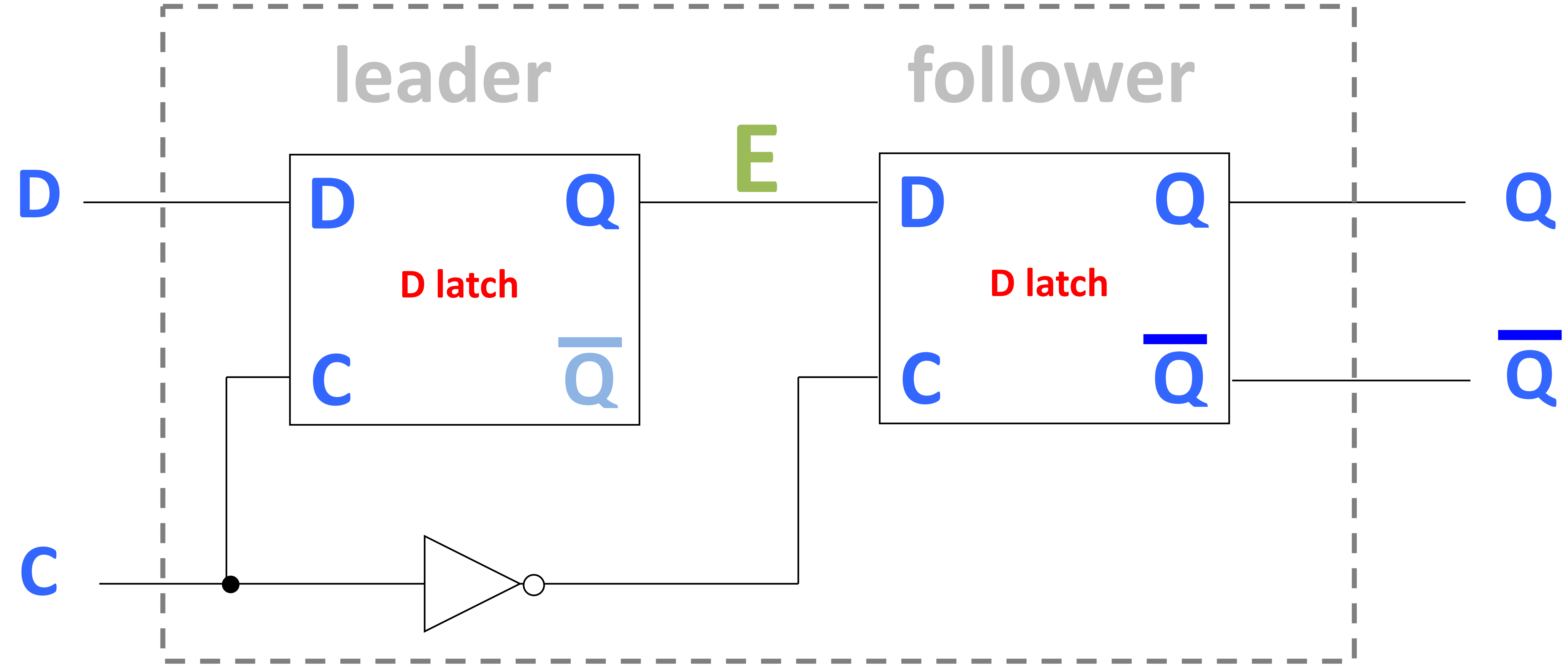




Time matters!

D flip-flop

with *falling*-edge trigger



ex

Recall, for each D latch:

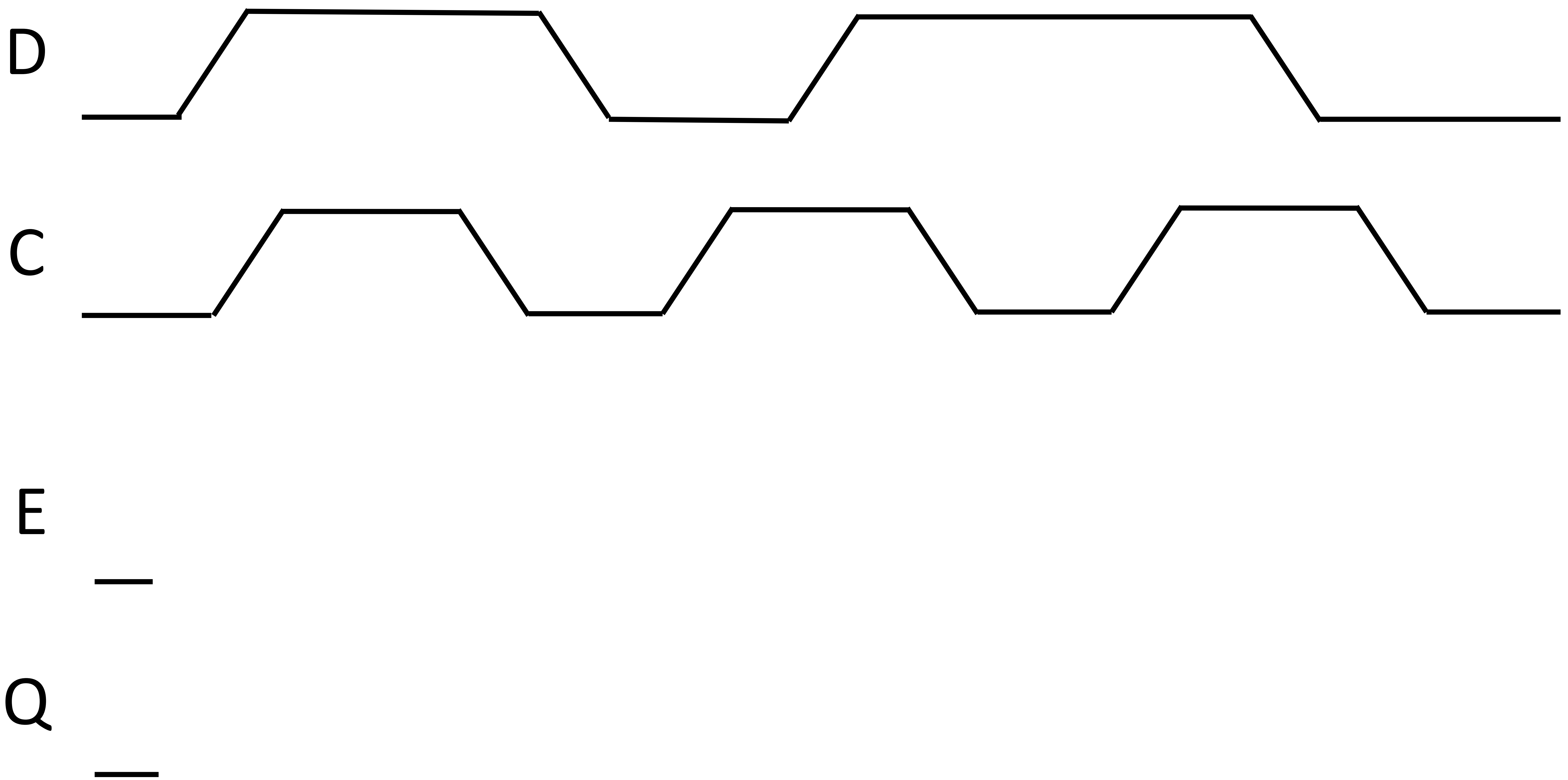
if their C = 0,

then their output holds.

if their C = 1,

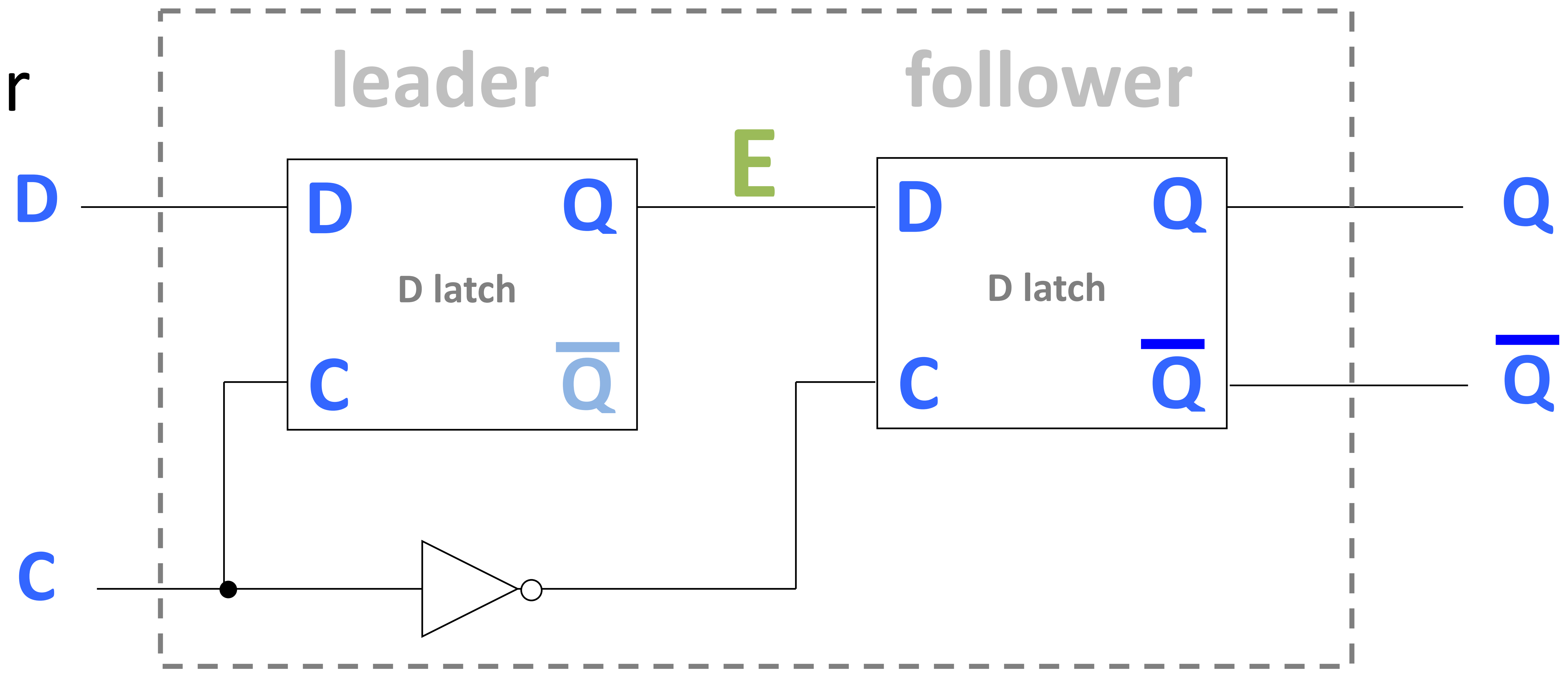
then the data input flows

to the output



Assume Q and E have an initial state of 0

D flip-flop with *falling*-edge trigger



the moment C “falls” from 1 to 0:

if D = 0, then Q = 0 and $\overline{Q}$ = 1

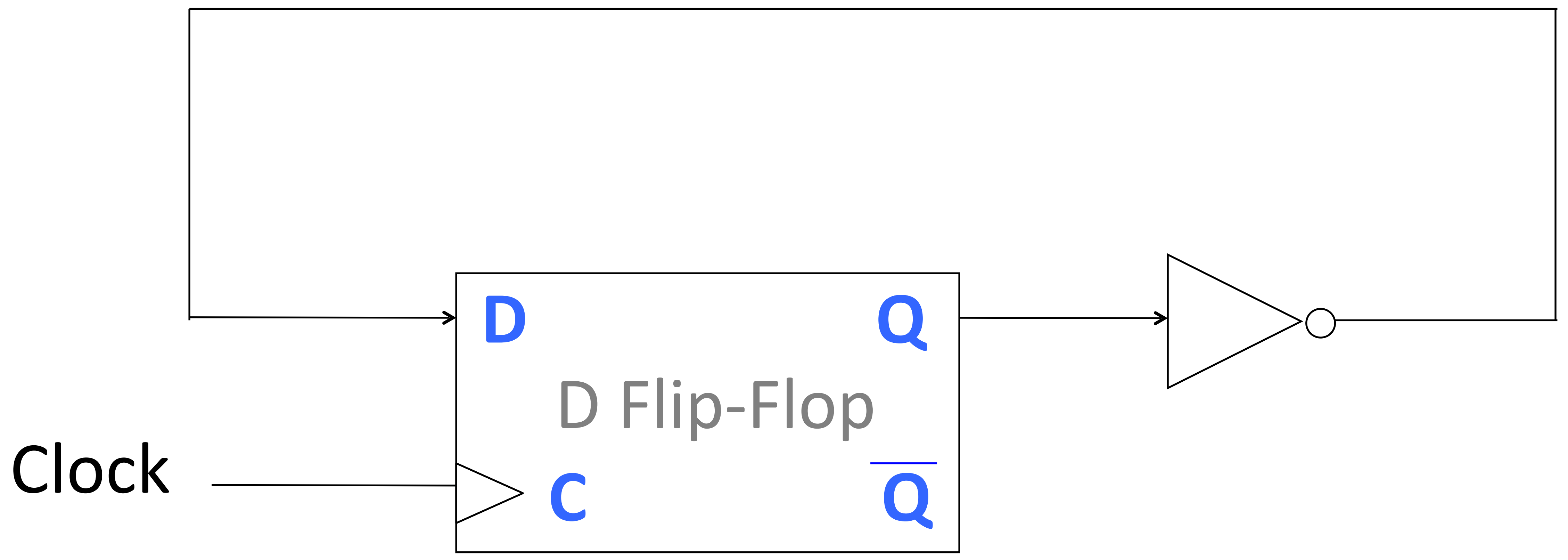
if D = 1, then Q = 1 and $\overline{Q}$ = 0

Q holds the value of D had until the next falling edge

Notes:

- The book covers rising-edge D flip-flops, which omit the NOT gate

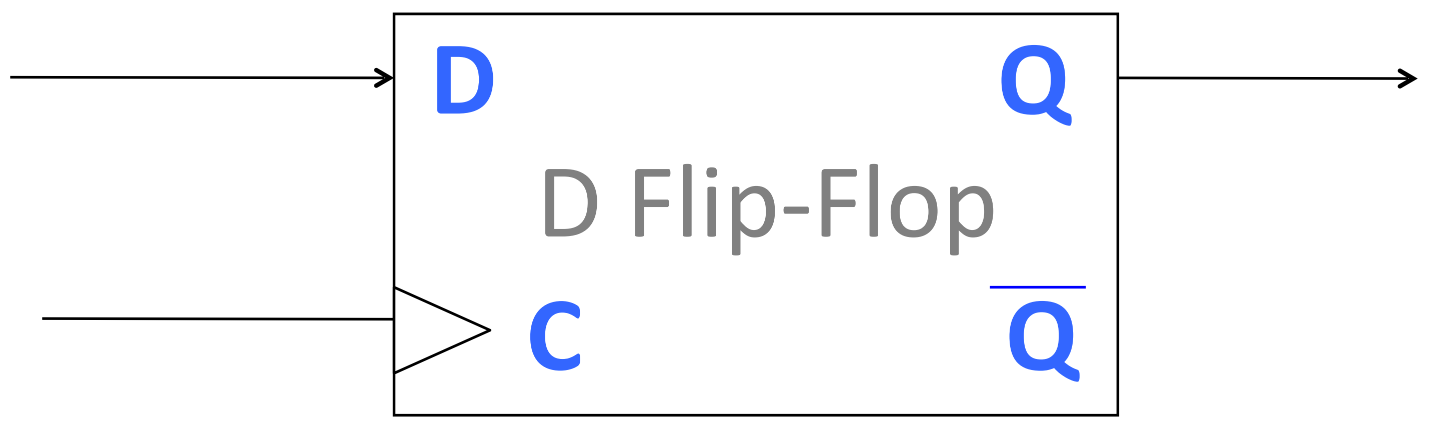
Reading and writing in the same cycle



Assume Q is initially 0.

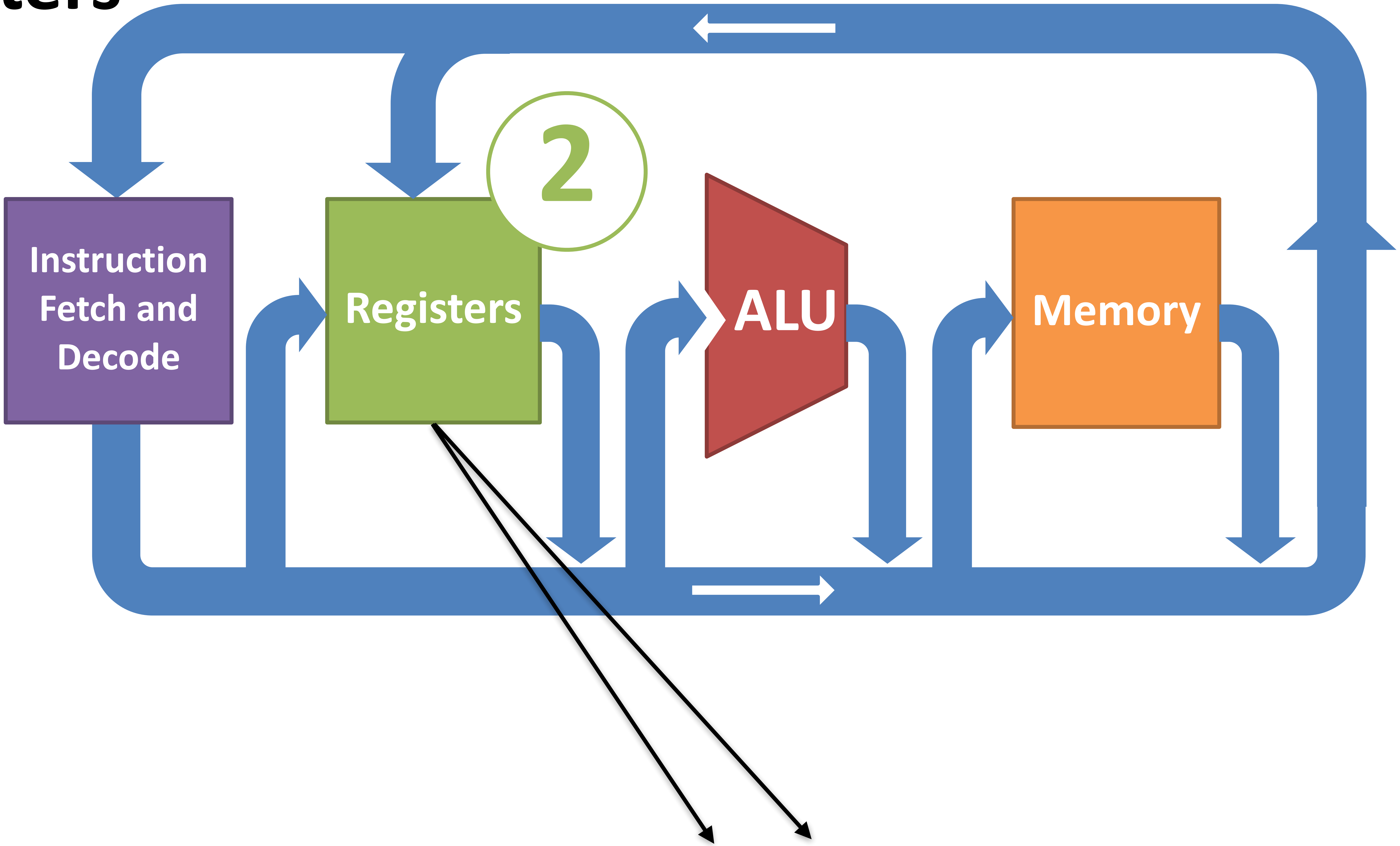
Moral: It’s OK to use the current output Q of a flip-flop as part of the the next data input D to the same flip-flop.

D flip-flop = one bit of storage



The value of D when C has a falling edge is remembered at Q until the next falling edge of C.

Registers



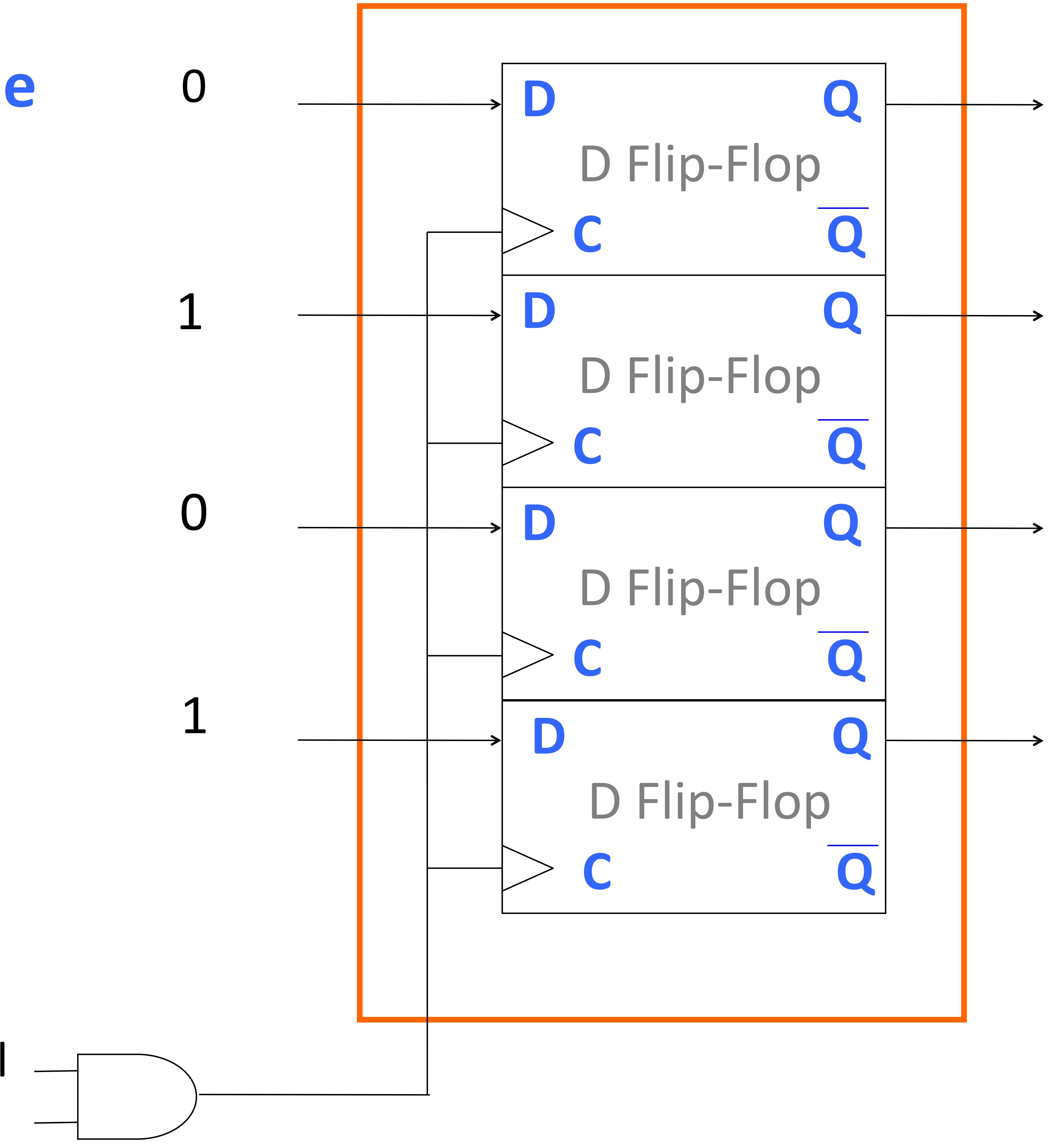
Assembly code (later this semester): `addq %rdi,%rsi`

A 1-nybble* register

**Half a byte!*

(a 4-bit hardware storage cell)

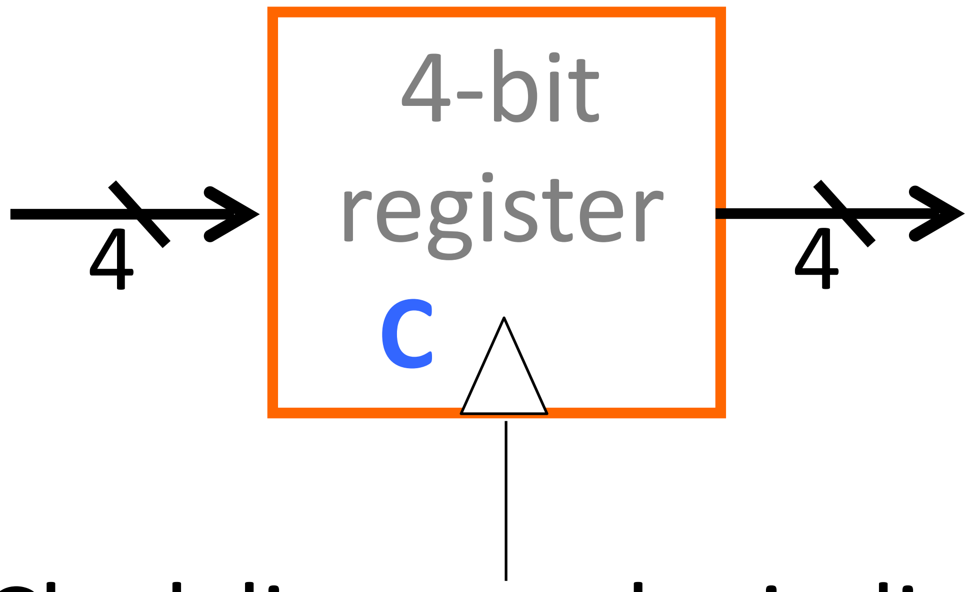
Write value



Shared clock

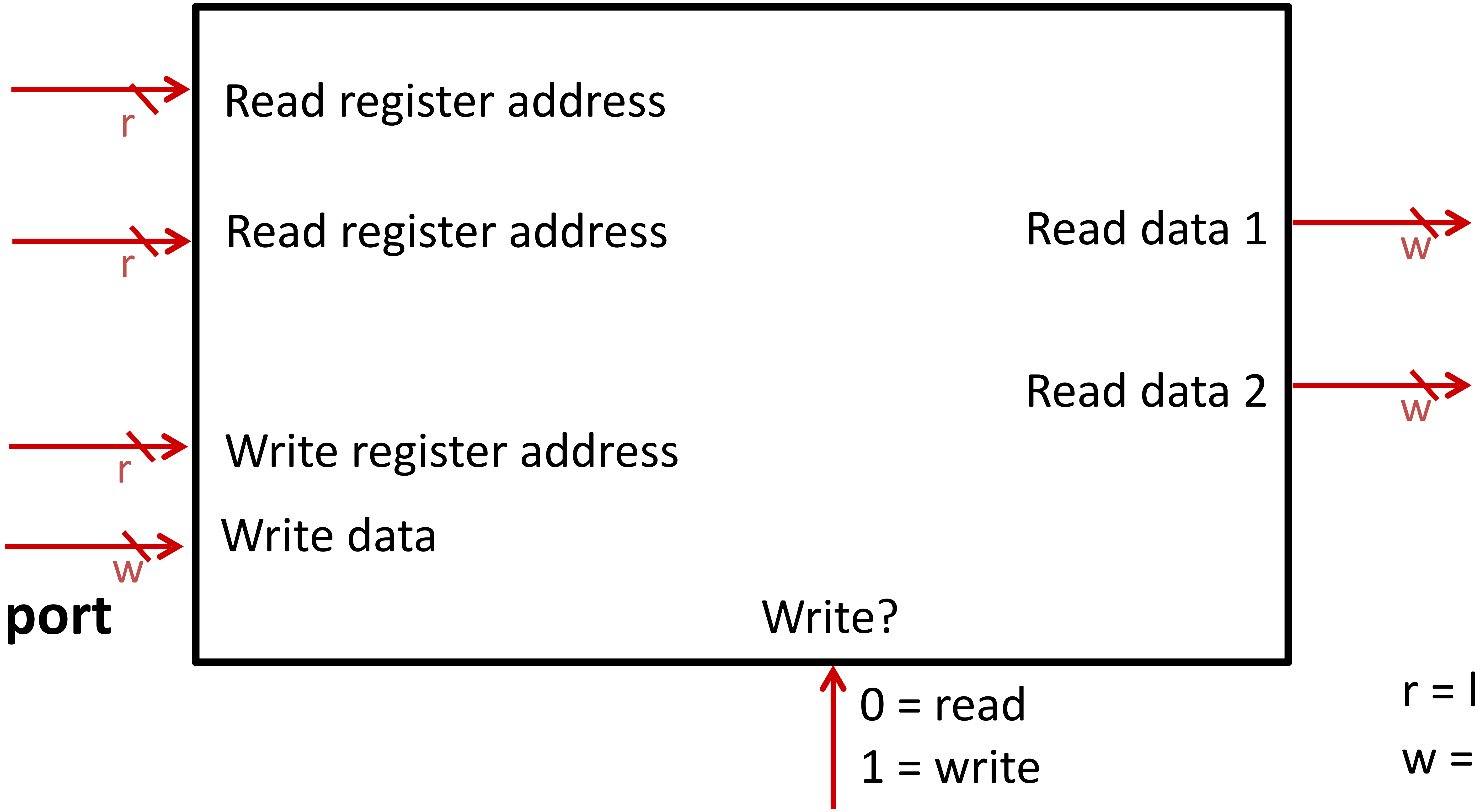
write control

Clock



Clock line may be indicated

Register file



Write port

Read ports

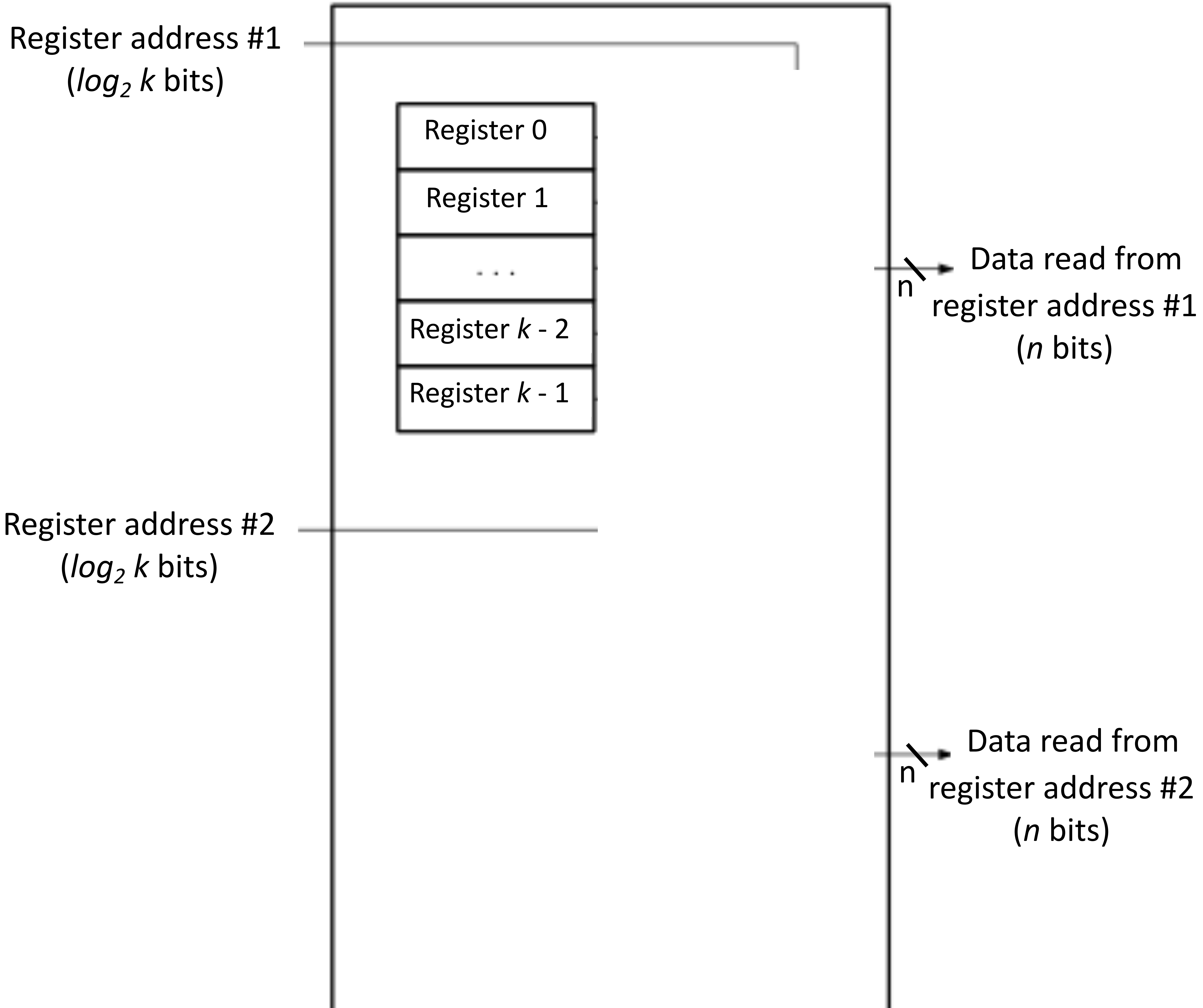
Why 2?

r = $\log_2$ number of registers

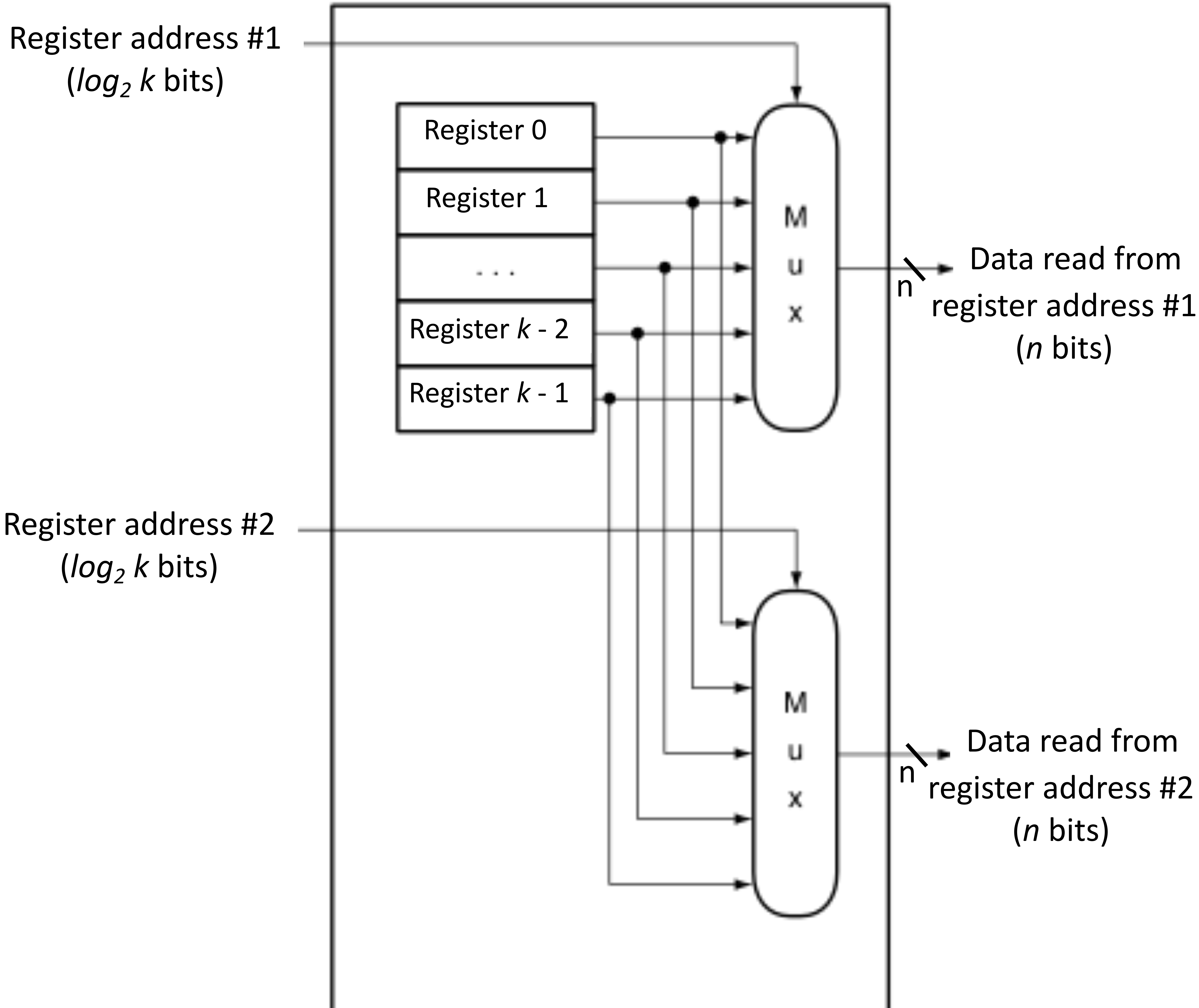
w = bits in word

Array of registers, with register selectors, write/read control, input port for writing data, output ports for reading data.

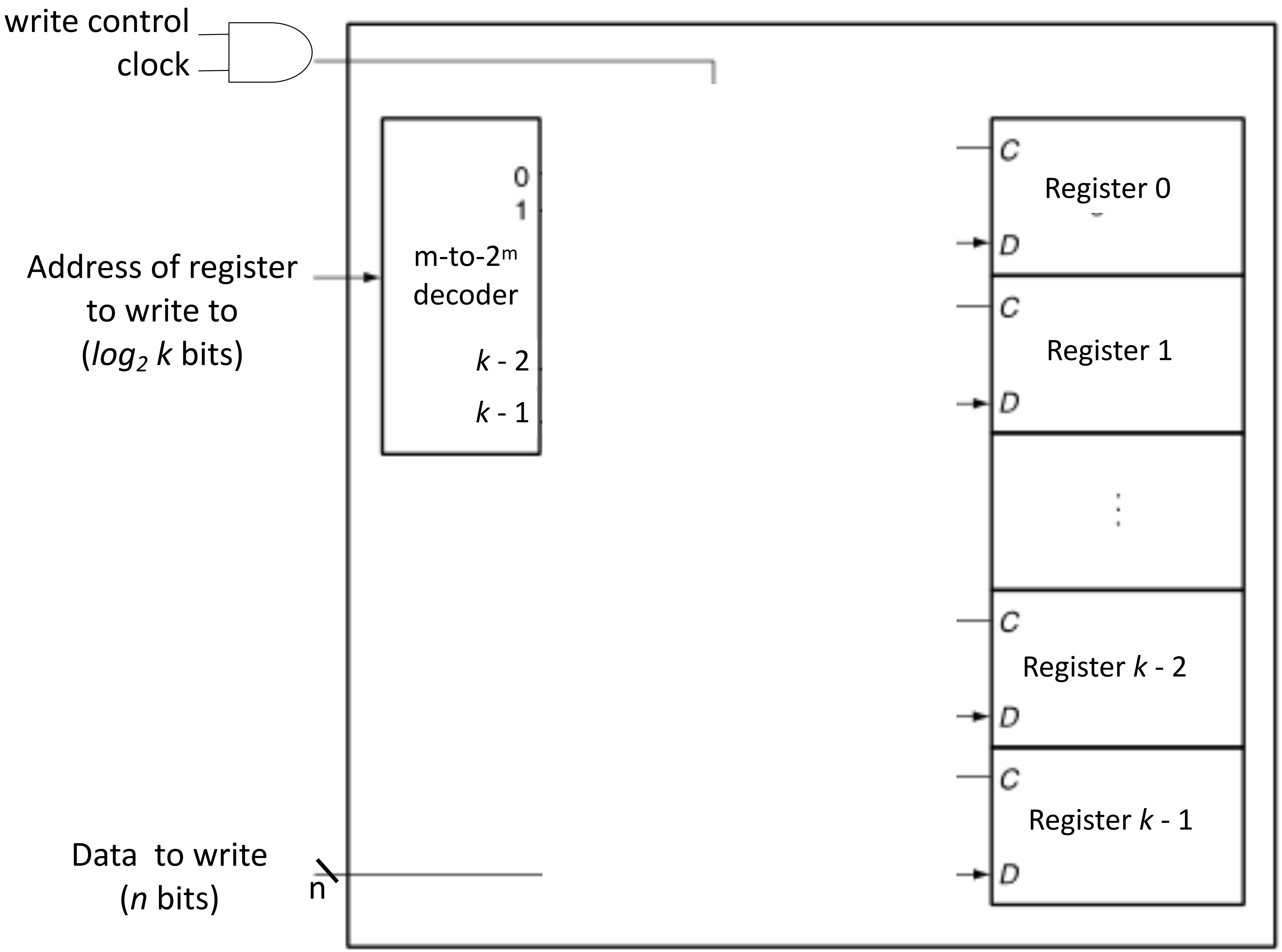
Read ports
(data out)



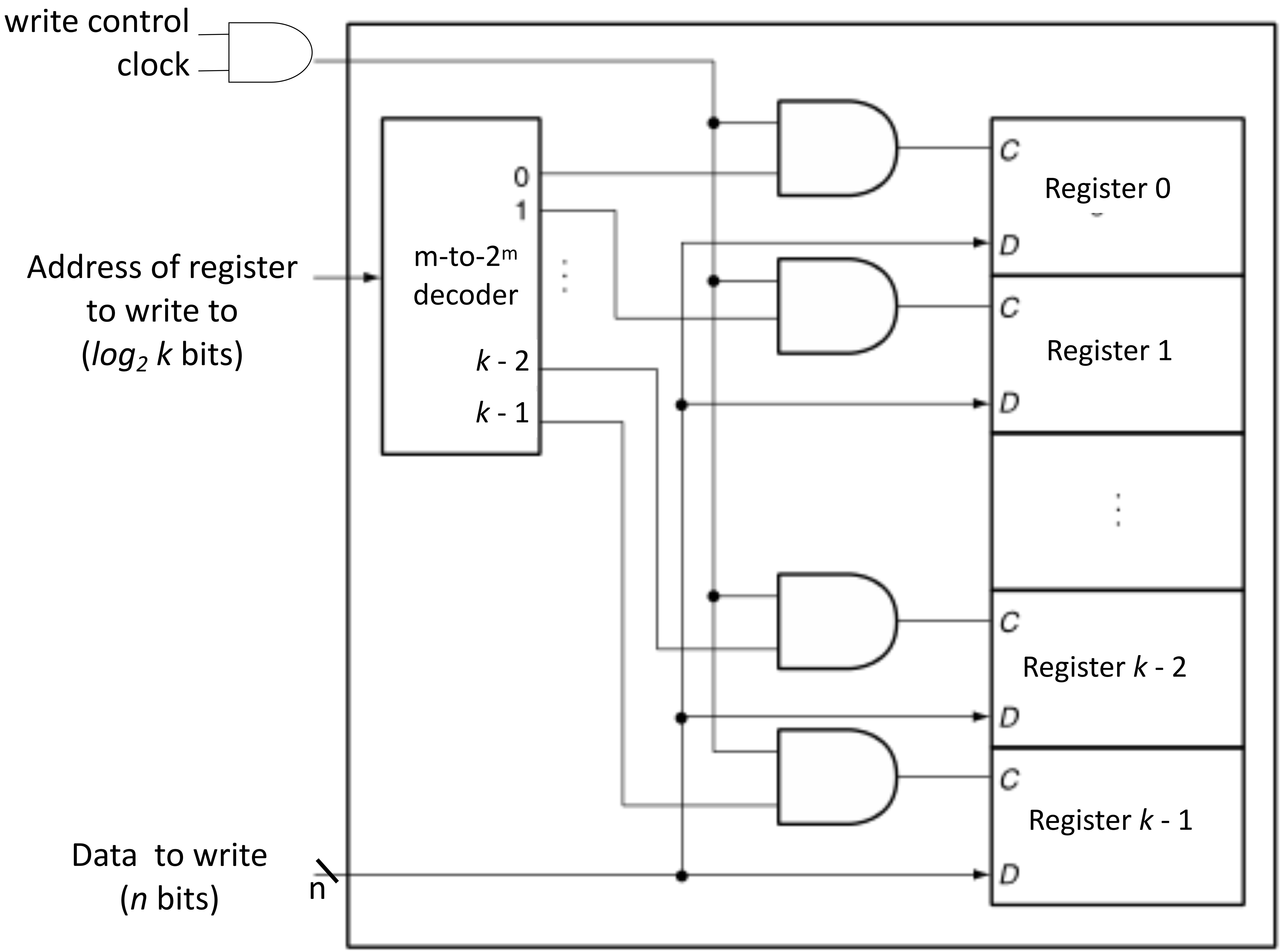
Read ports
(data out)



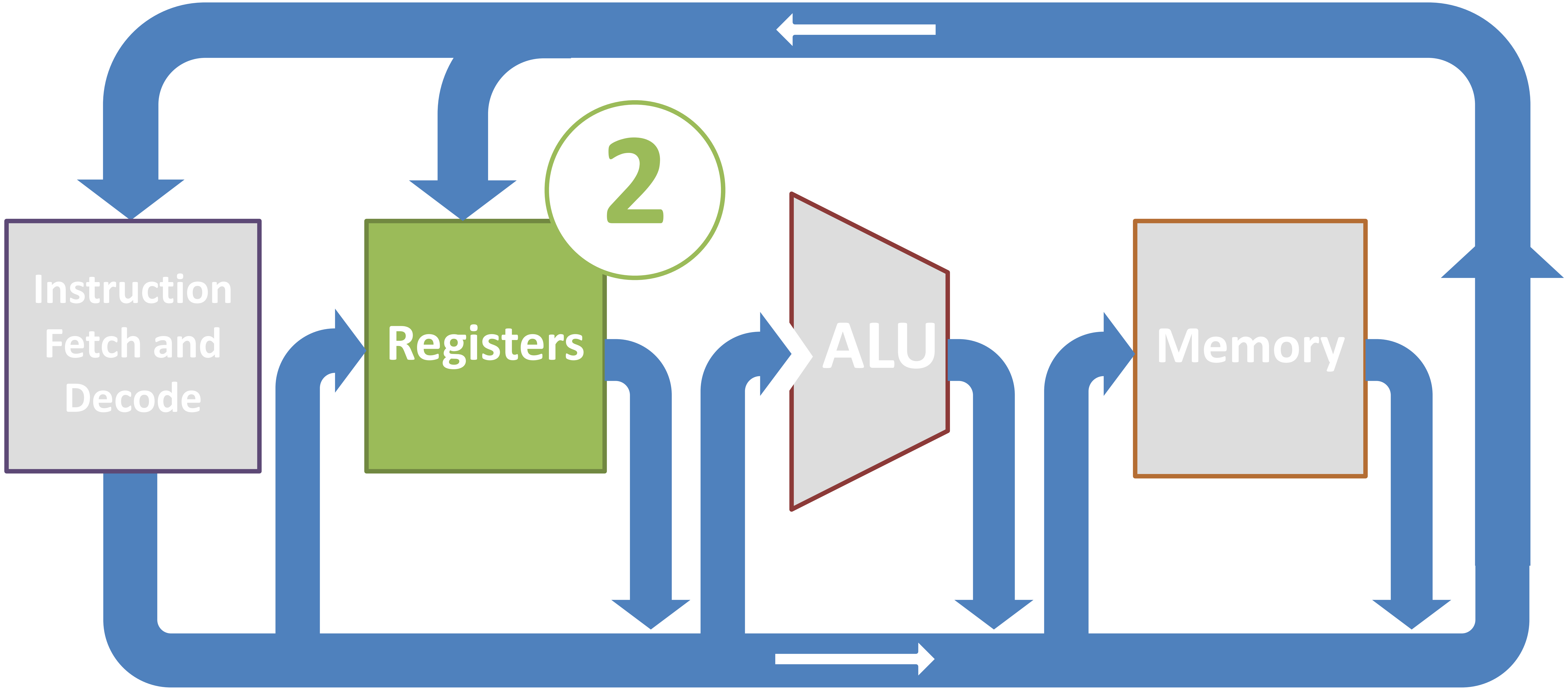
Write port (data in)



Write port (data in)

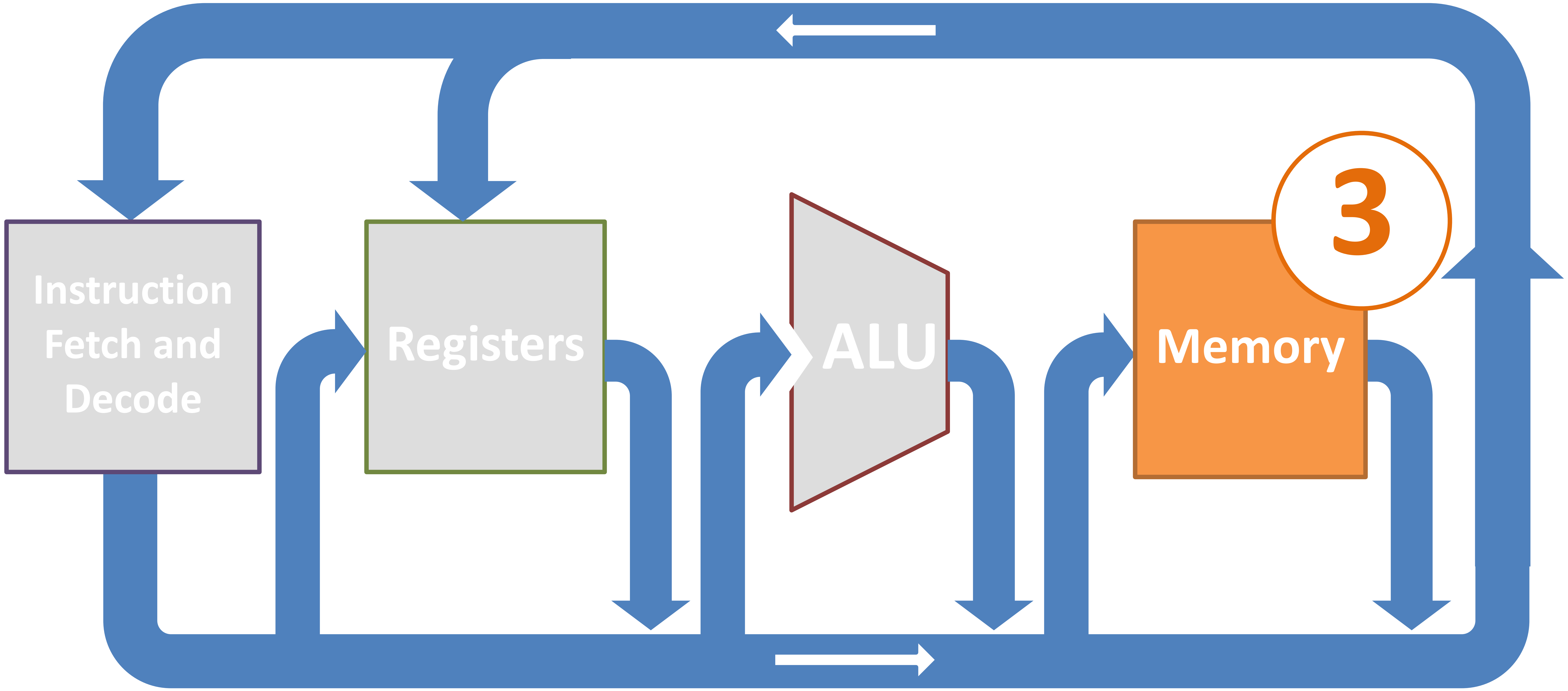


Registers summary



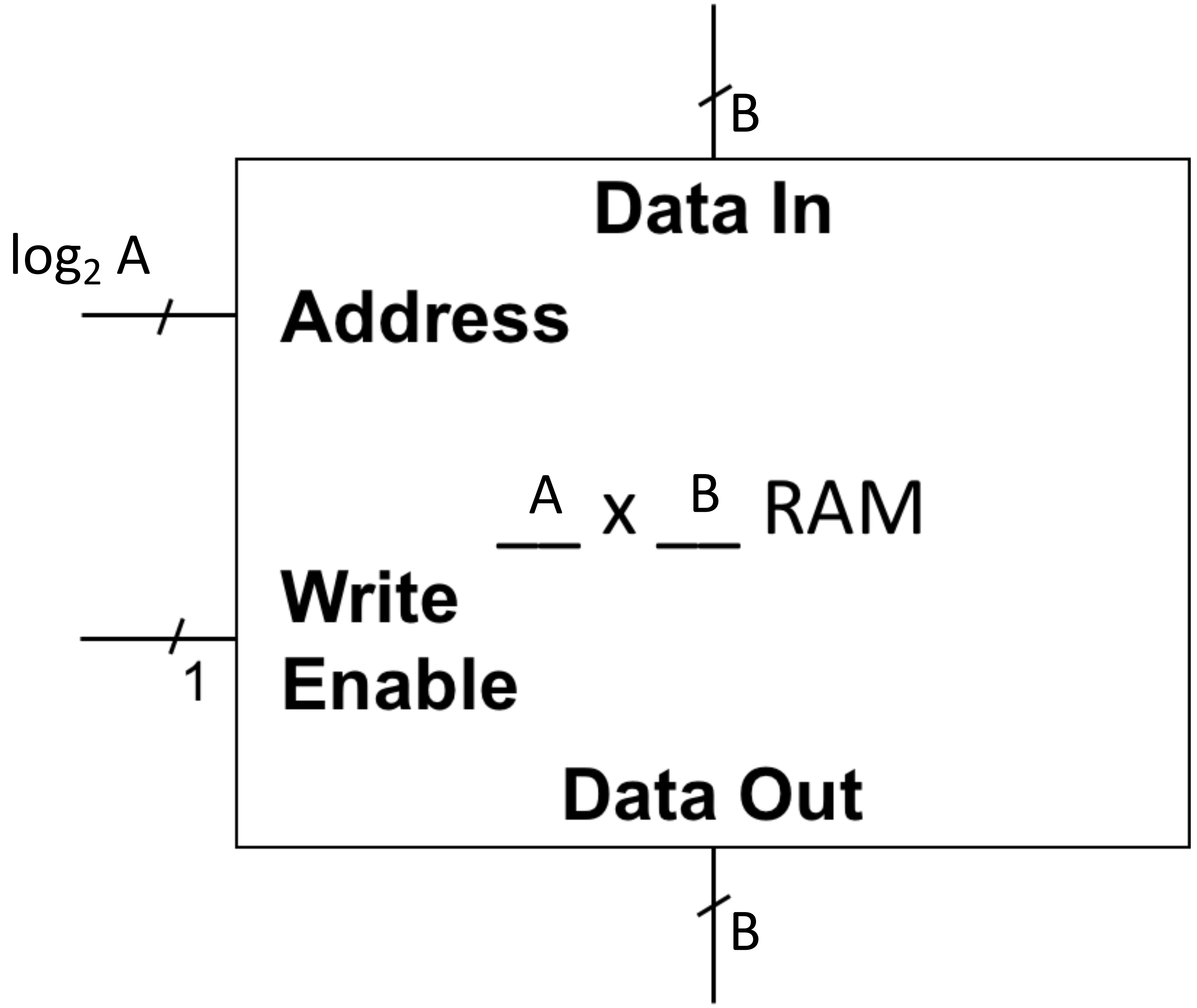
- For our purposes: implemented with flip-flops
- Very fast access
- Limited in size:
  - Need an m-to-2^m decoder
  - CPUs typically have ~10s of words of register storage

Memory summary



- We’ll think about at a higher level of abstraction
- Designed to handle a much larger amount of data
  - CPUs can have millions to billions of words of memory storage

RAM (Random Access Memory)



- A is number of words in RAM
- Specify the desired word by an address of size $\log_2$ A
- B is the width of each word (in bits)

16 x 4 RAM