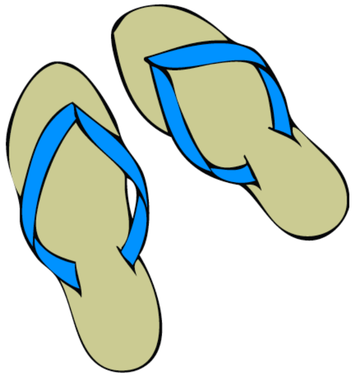
Sequential Logic and State

Latch: CC-BY Rberteig@flickr

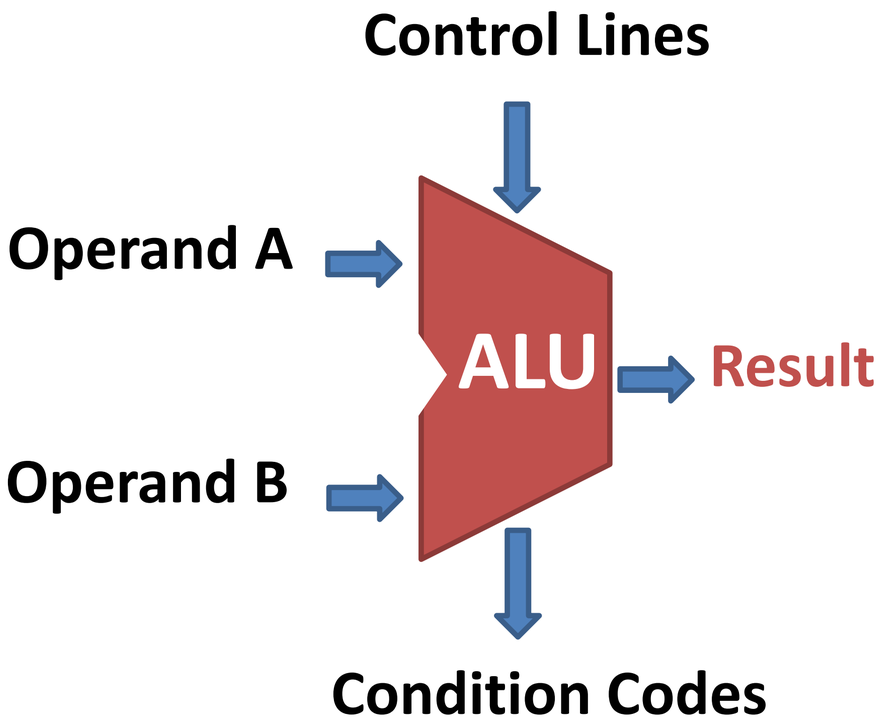


Output depends on inputs *and stored values*.

(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?

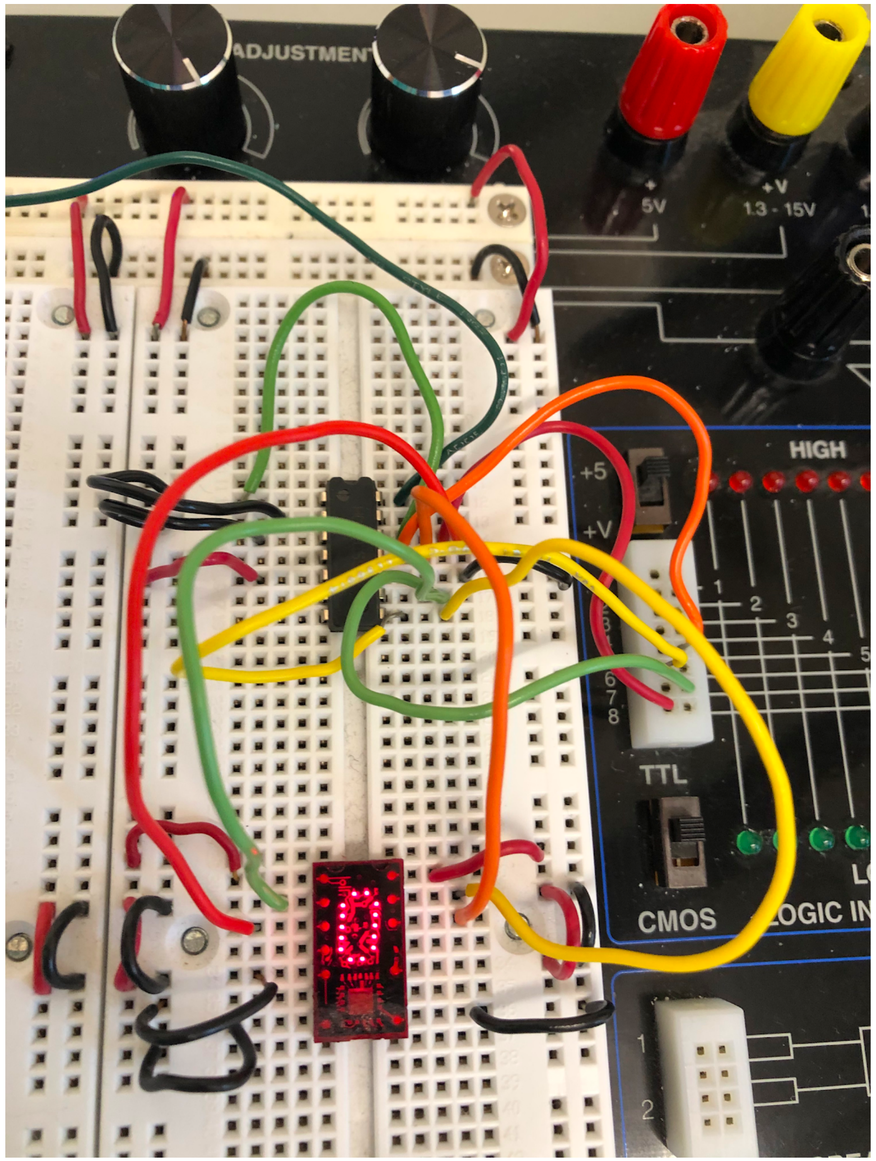
Answer: we need circuits that depend not just on inputs, but also on *prior state*
= Sequential Logic

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

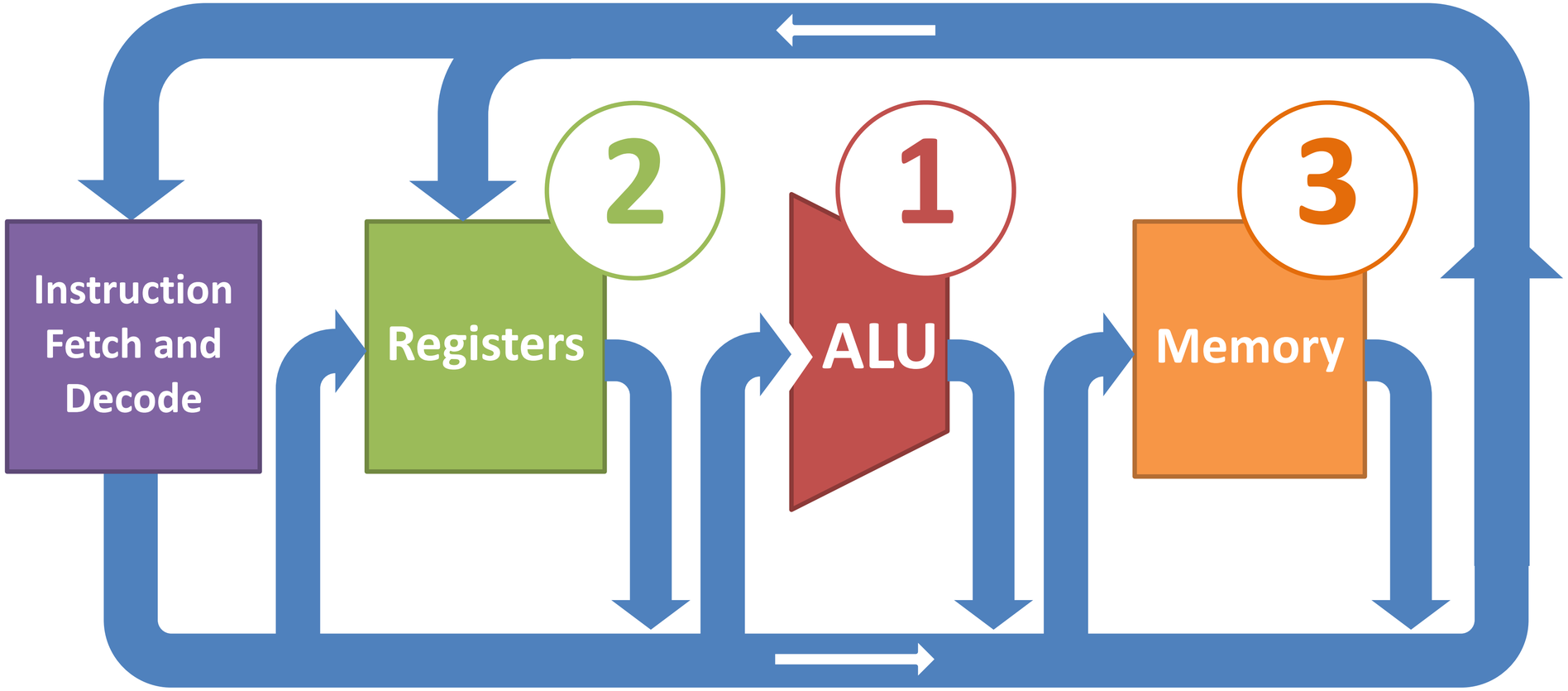
Nobody has responded yet.

Hang tight! Responses are coming in.

Example from previous lab



Processor: Data Path Components

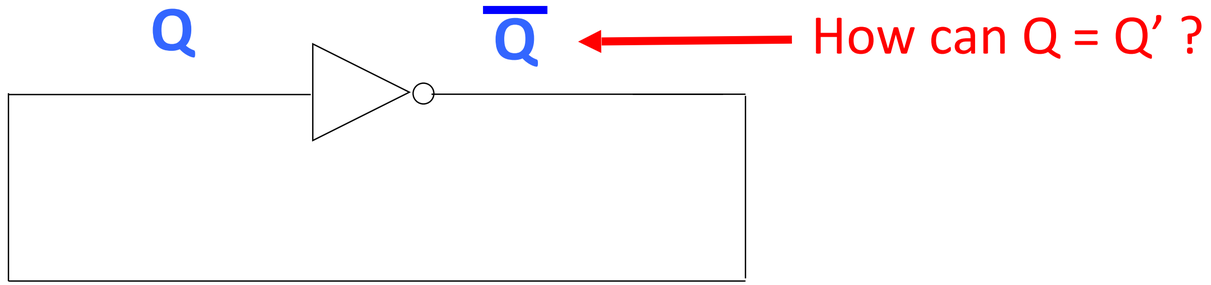


Goal for this section

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

First attempt: Unstable circuit

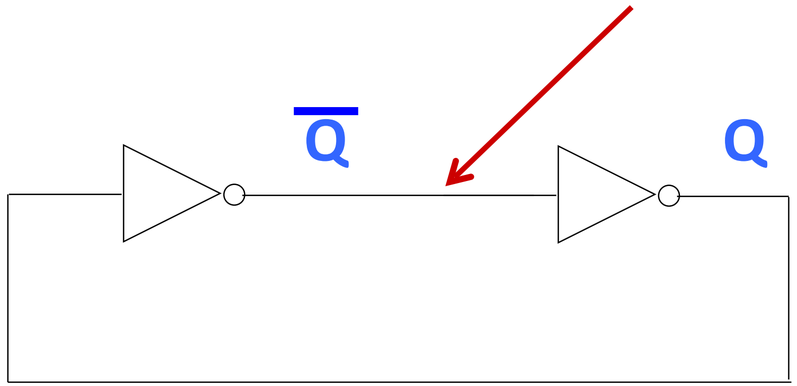


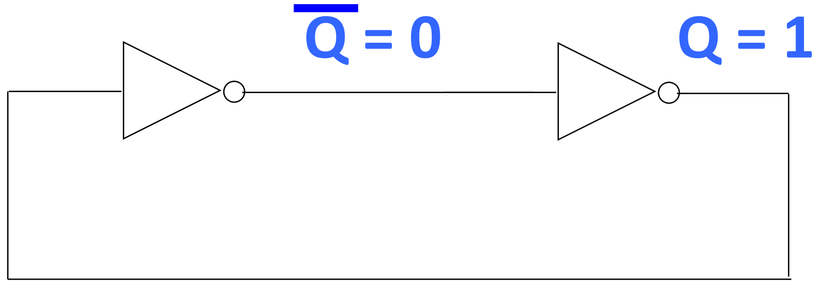
Have this issue with any odd number of inverters in a loop.

Second attempt: stable circuit?

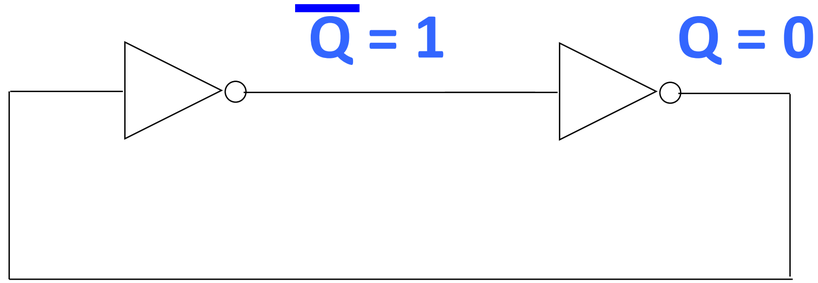
Things are more sensible with an even number of inverters in a loop.

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





or

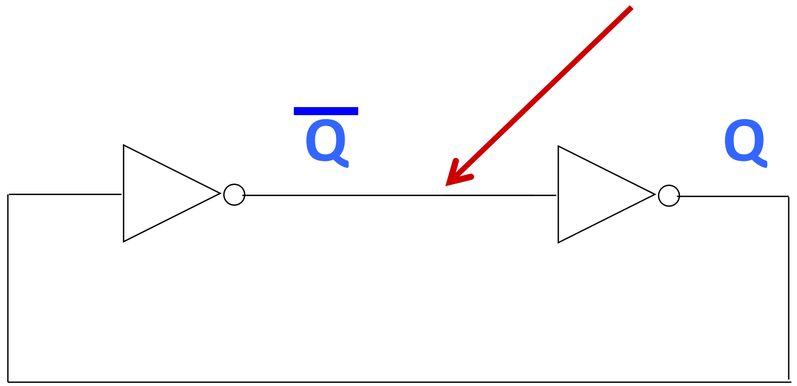


Now stable, but how do we set the value?

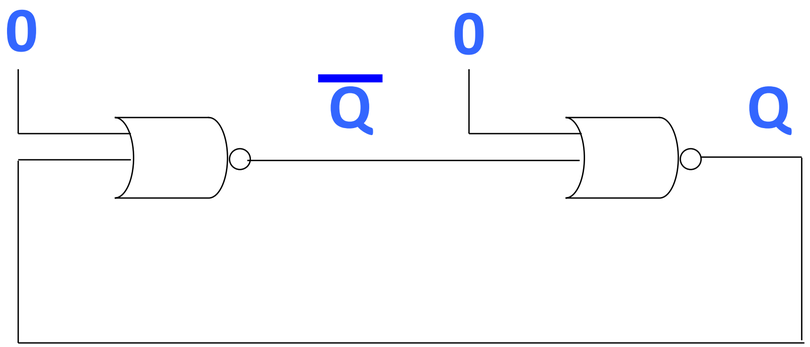
Bistable latches

Things are more sensible with an even number of inverters in a loop.

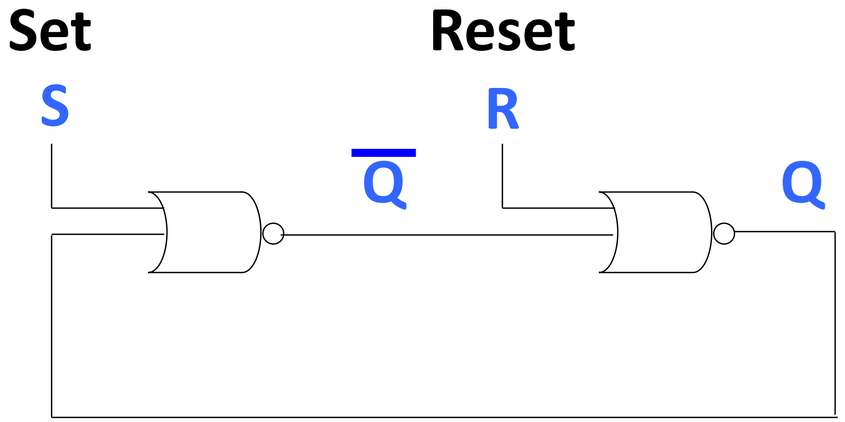
Suppose we somehow get a 1 (or a 0?) on here.



=



SR latch

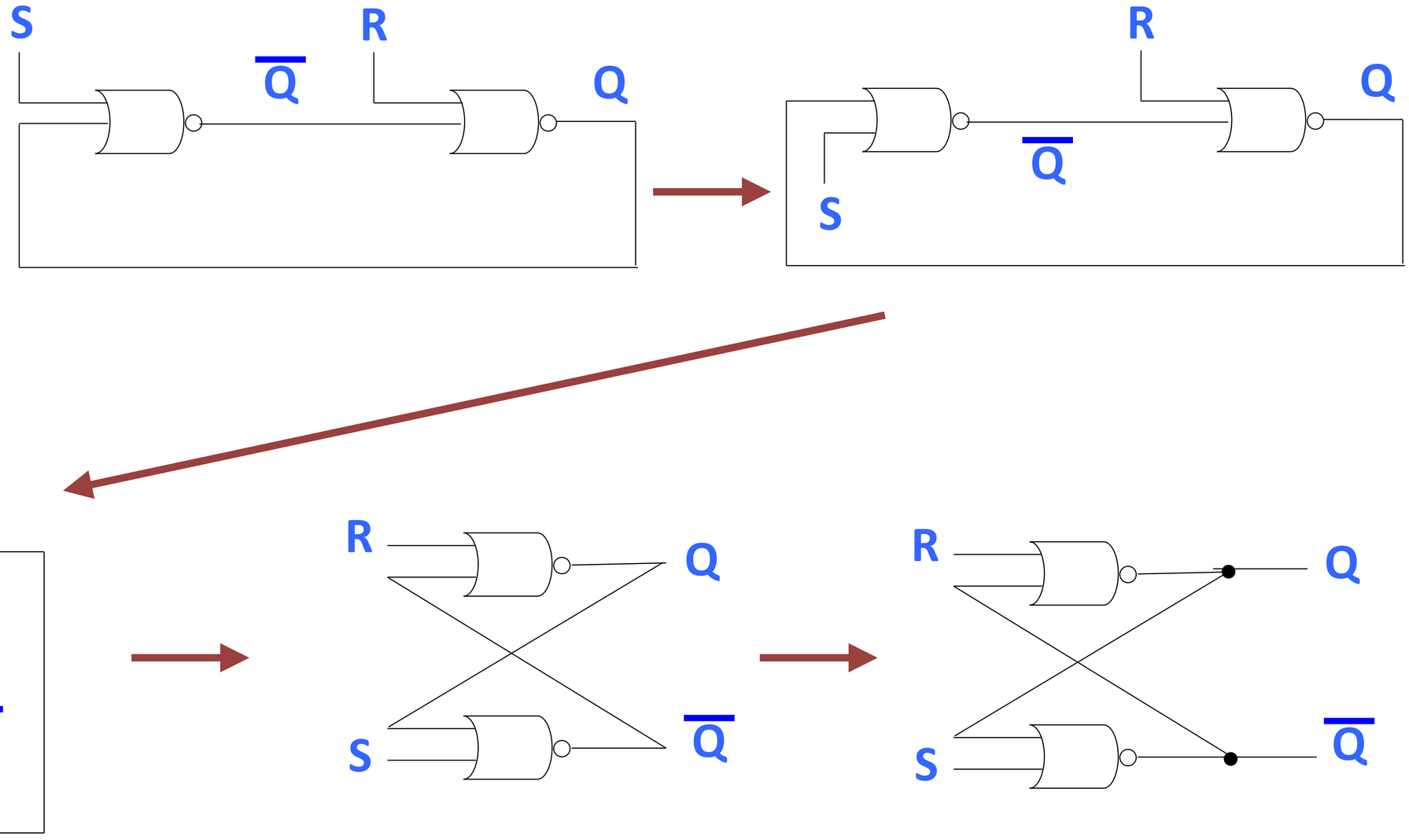


S	R	Q_{prev}	Q'_{prev}	Q_{next} (stable)	Q'_{next} (stable)
0	0	0	1	0	1
0	0	1	0	1	0
1	0	*any*	*any*	1	0
0	1	*any*	*any*	0	1
1	1	*any*	*any*	0	0

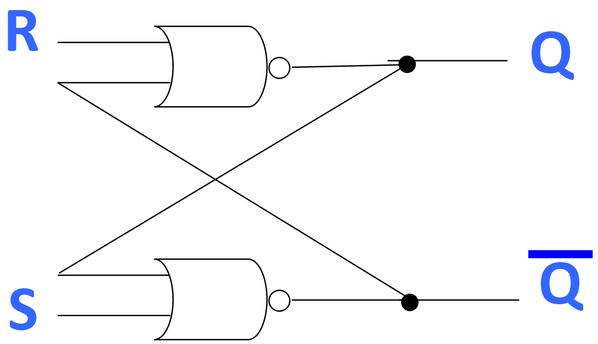
← Violates invariant that Q and Q' are inverses!

SR latch

Move from the circuit we built to the canonical form



SR latch



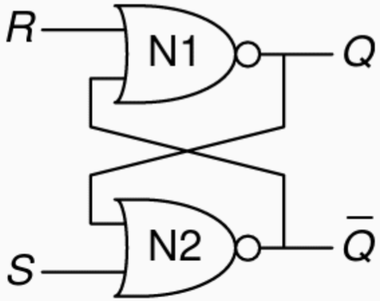


Figure 3.3 SR latch schematic

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?

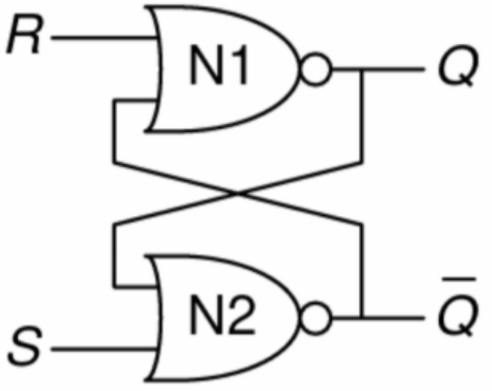


Figure 3.3 SR latch schematic

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

How do we set Q to 1?

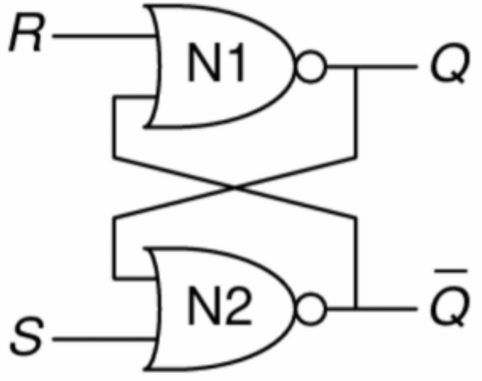


Figure 3.3 SR latch schematic

S = 0; R = 0 — 0%

S = 1; R = 0 — 0%

S = 0; R = 1 — 0%

S = 1; R = 1 — 0%

None of the above — 0%

How do we set Q to 1?

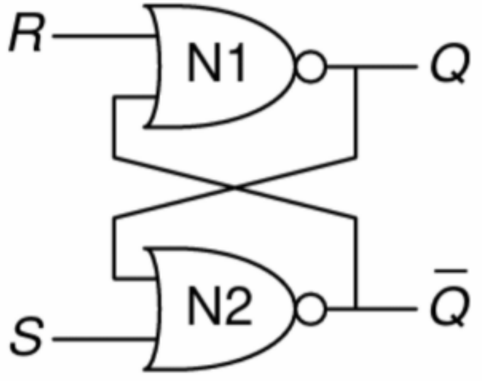


Figure 3.3 SR latch schematic

S = 0; R = 0 — 0%

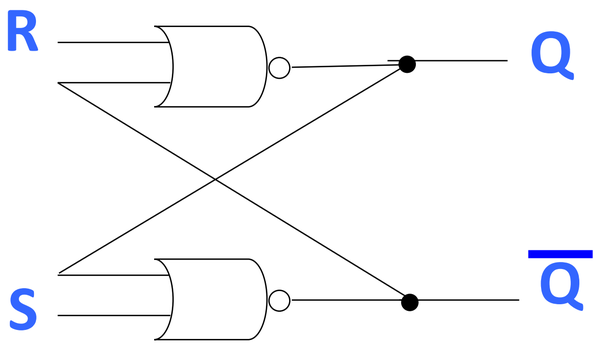
S = 1; R = 0 — 0%

S = 0; R = 1 — 0%

S = 1; R = 1 — 0%

None of the above — 0%

SR latch



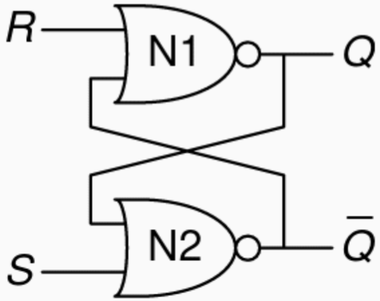


Figure 3.3 SR latch schematic

Meets our goals:

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

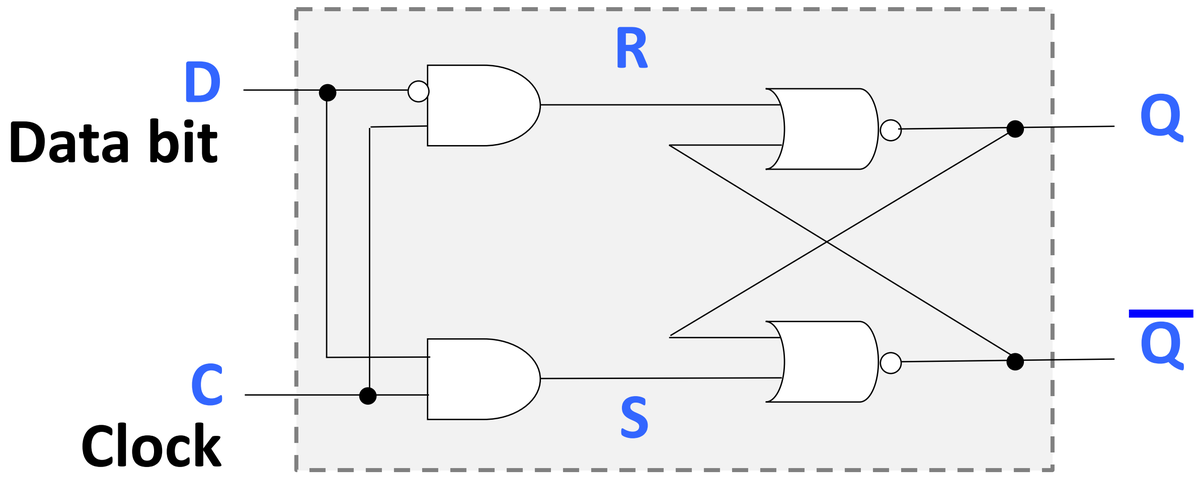
But:

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

D latch

Goals:

- 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:

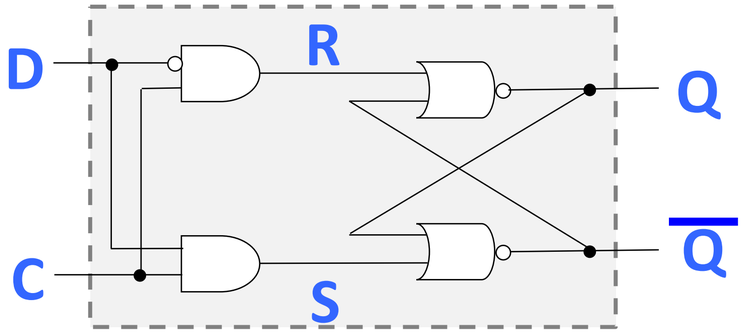
if D = 0, then R = 1 and S = 0, Q = 0

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

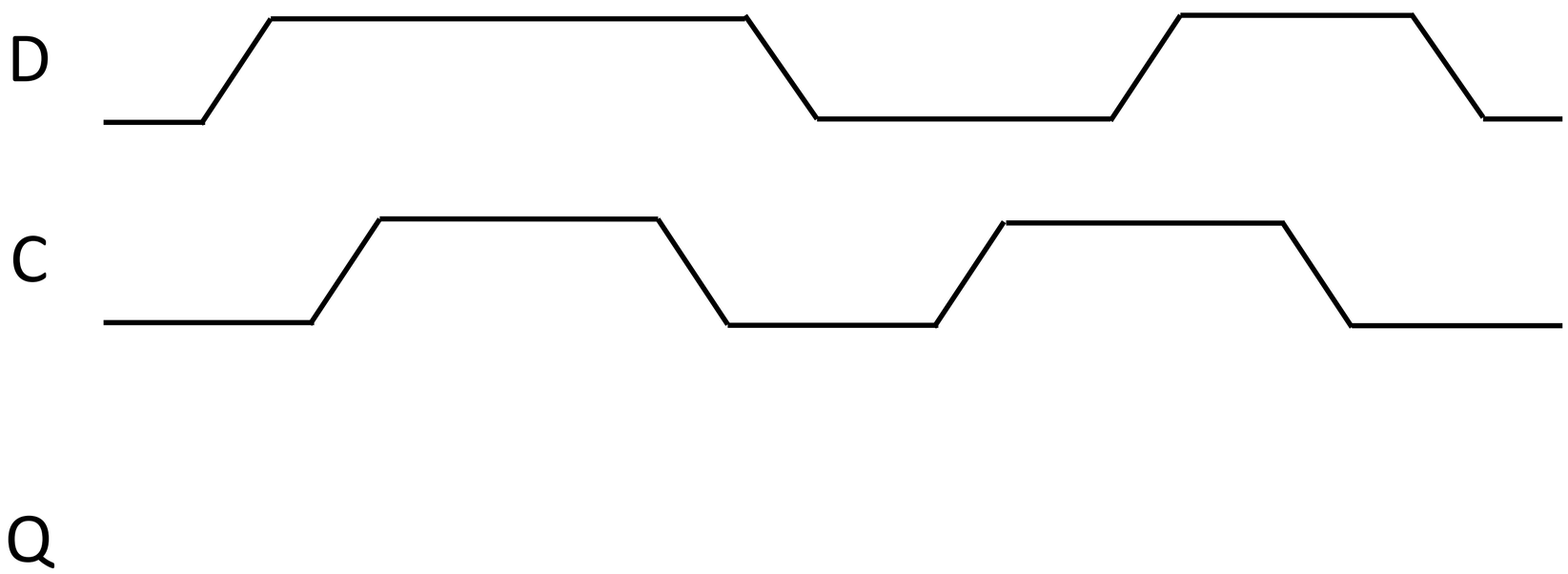
Notes:

- 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!



ex



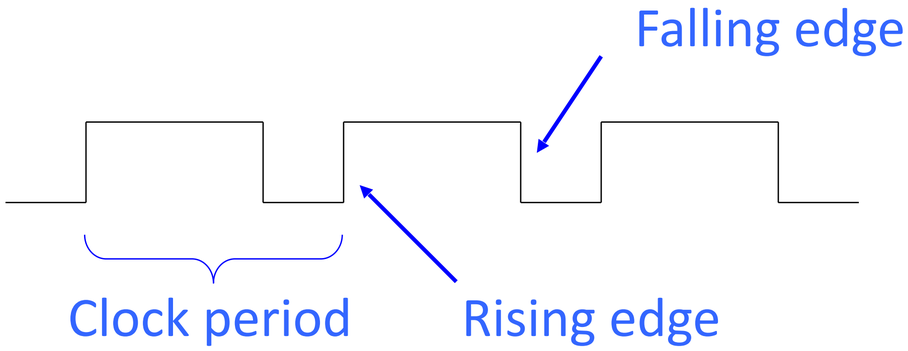
Assume Q has an initial state of 0

In general: clocks

Clock: free-running signal

with fixed cycle time = clock period = T.

Clock frequency = 1 / clock period



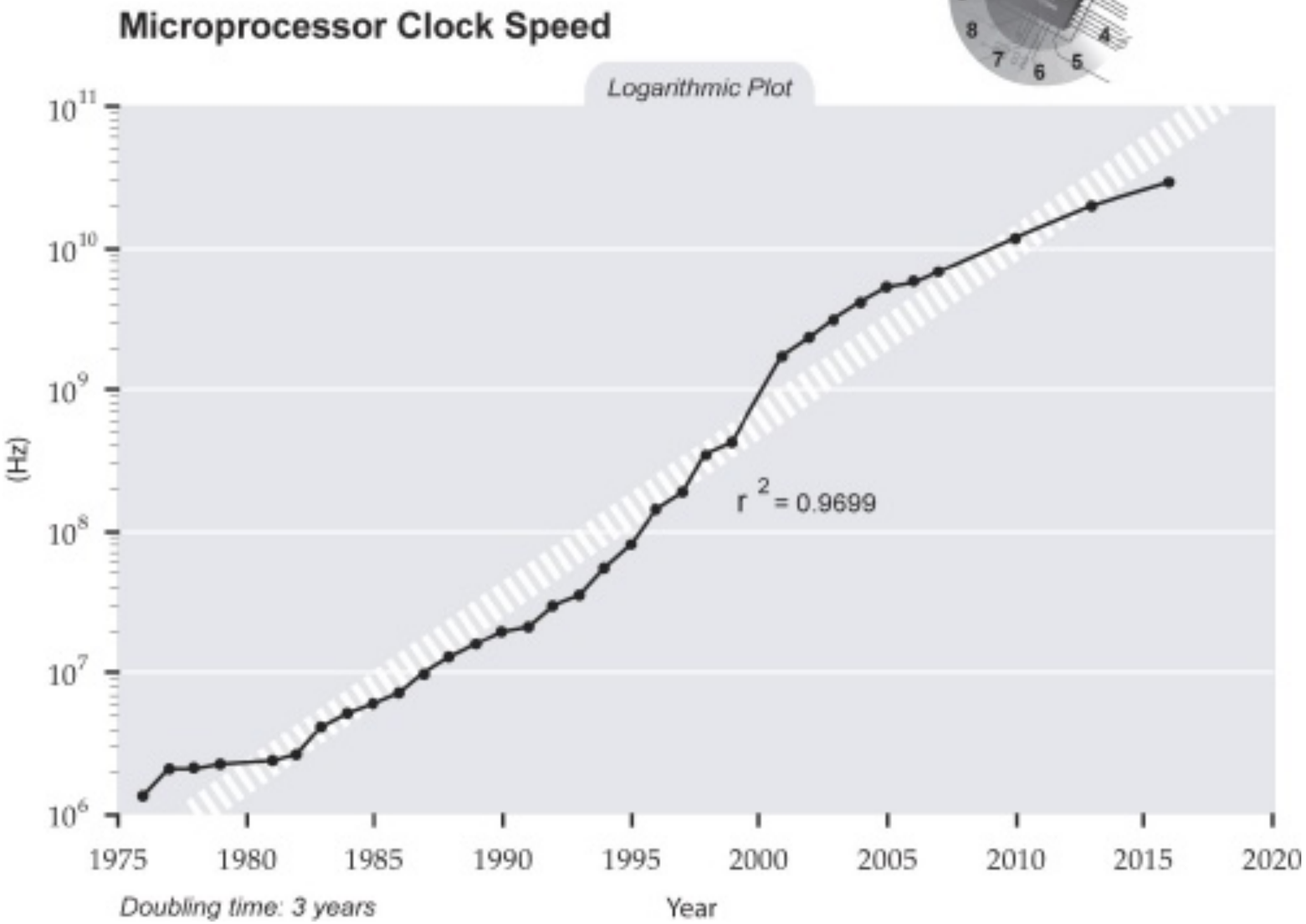
A clock controls when to update a sequential logic element's state.

Aside: “Clock frequency”

Clock frequency

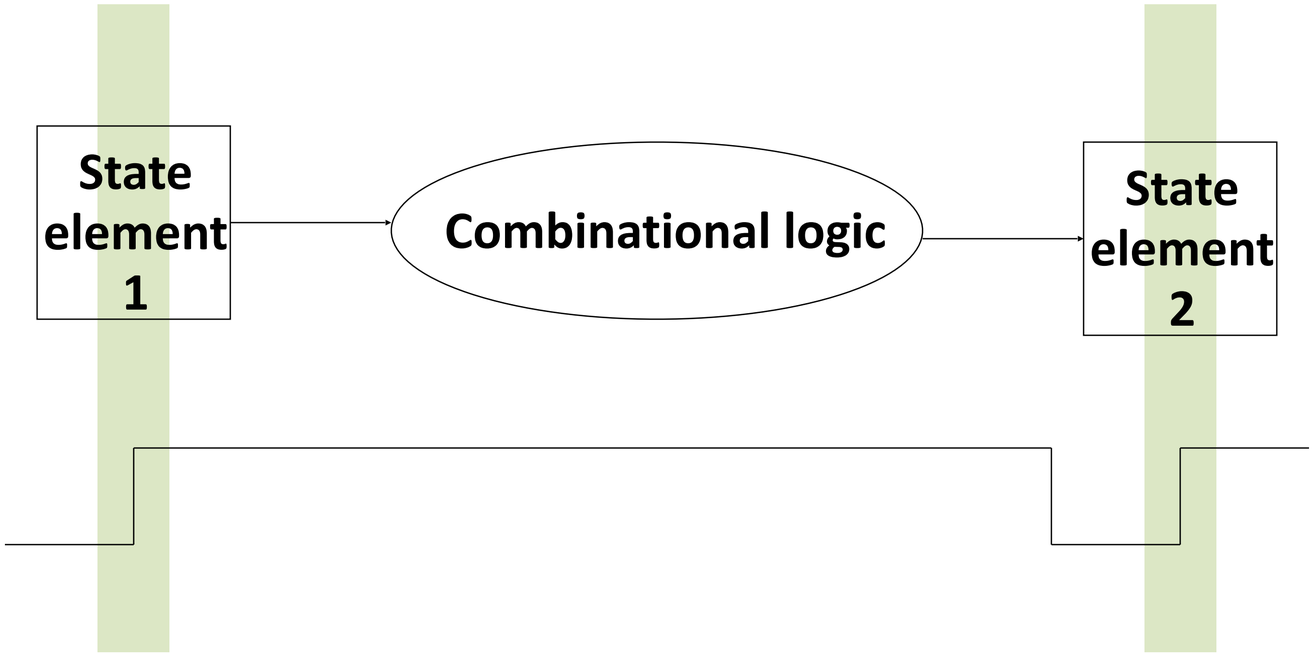
= 1 / period = 1 / s = Hz

Typical CPU: 3-4 GHz

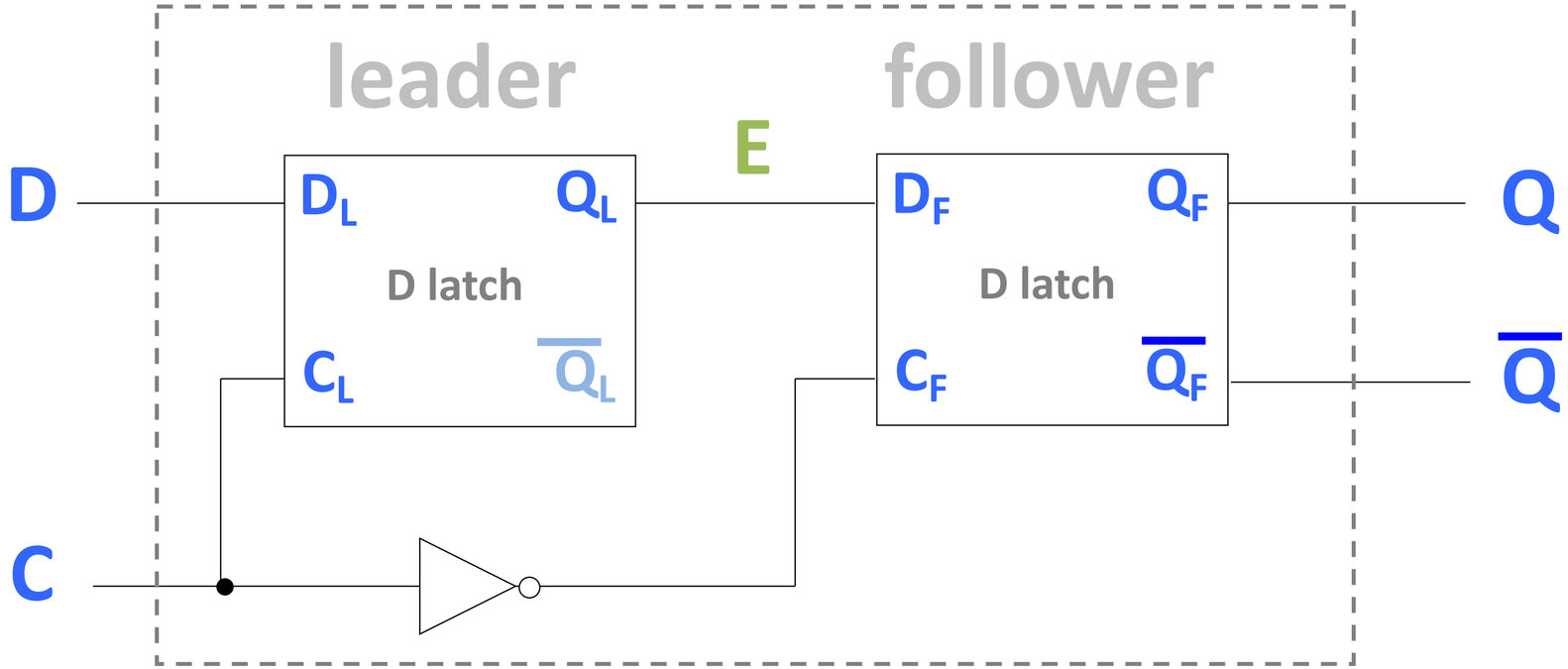


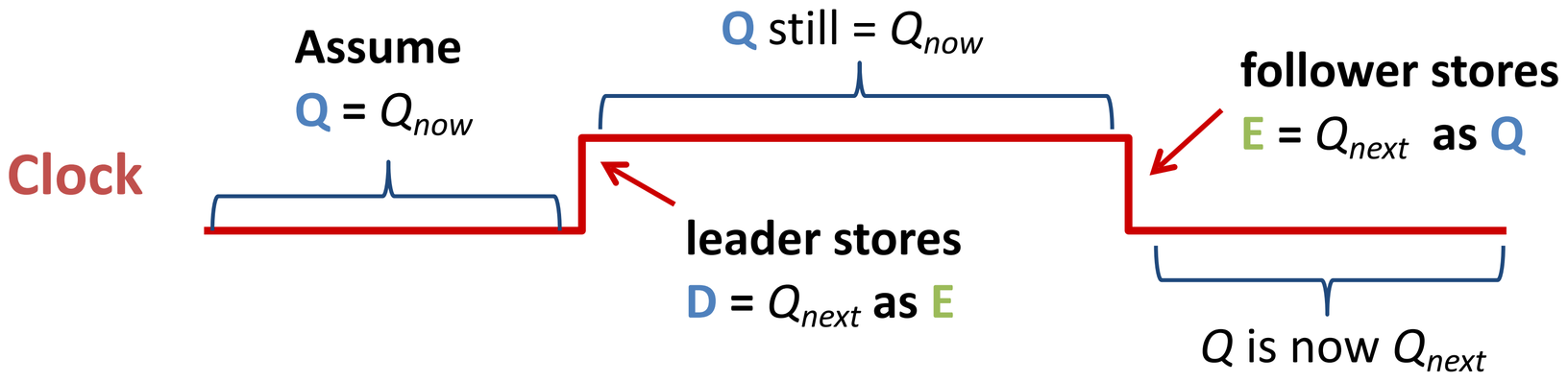
Synchronous systems

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



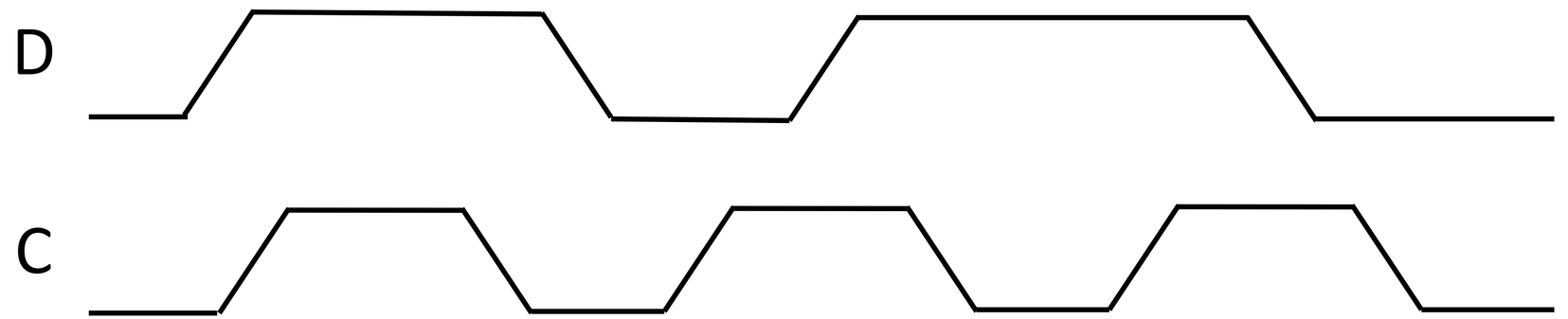
D flip-flop with *falling*-edge trigger





Time matters! D flip-flop with *falling*-edge trigger

ex

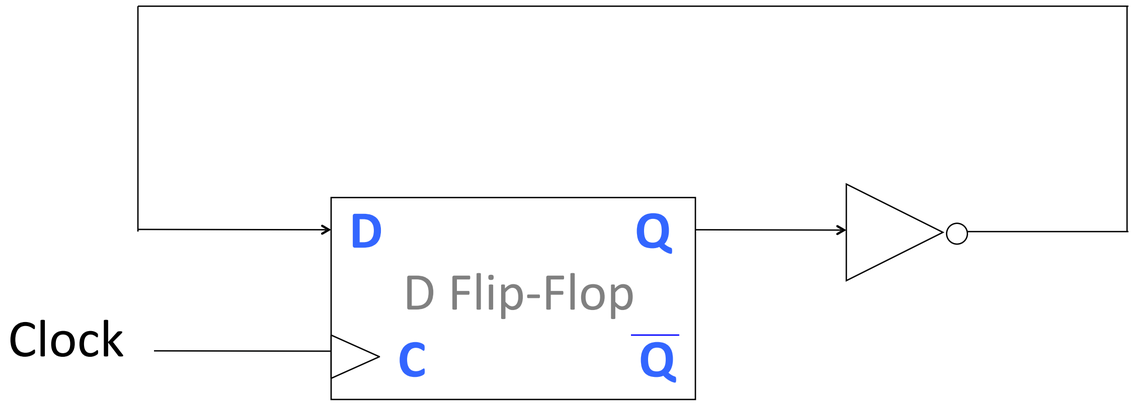


E

Q

Assume Q and E have an initial state of 0

Reading and writing in the same cycle



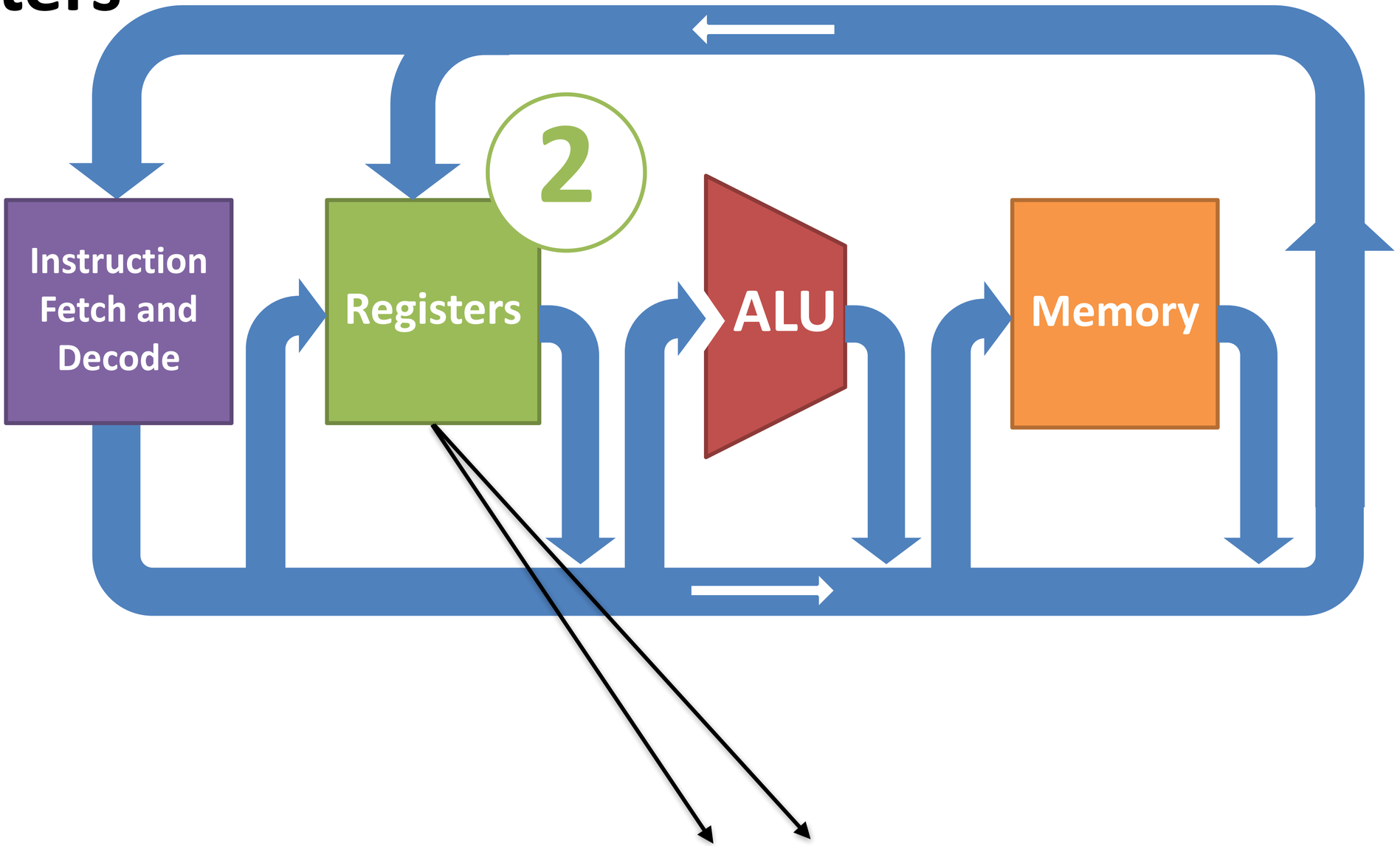
Assume Q is initially 0.

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

Registers



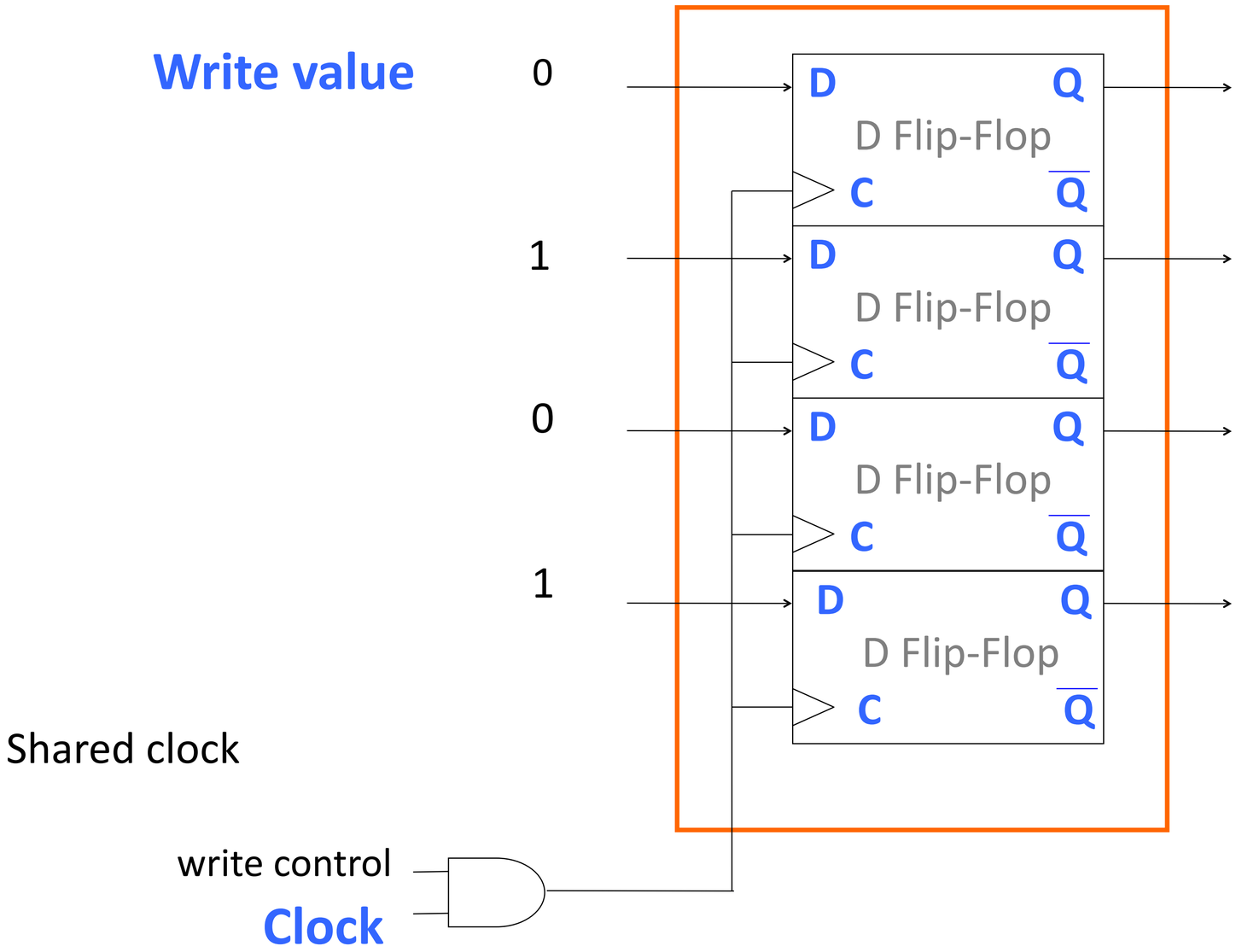
Assembly code (later this semester):

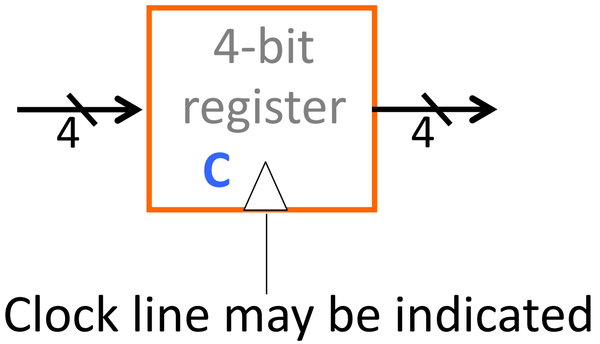
```
addq %rdi,%rsi
```

A 1-nybble* register

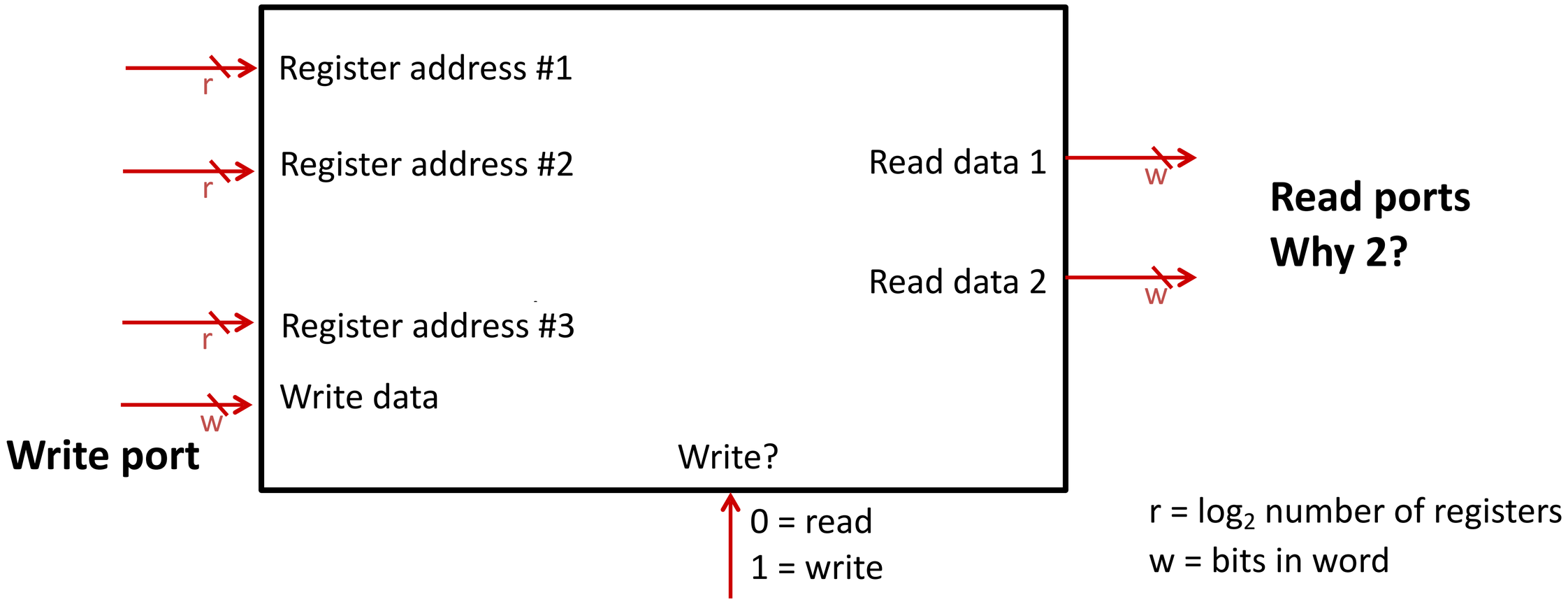
*Half a byte!

(a 4-bit hardware storage cell)



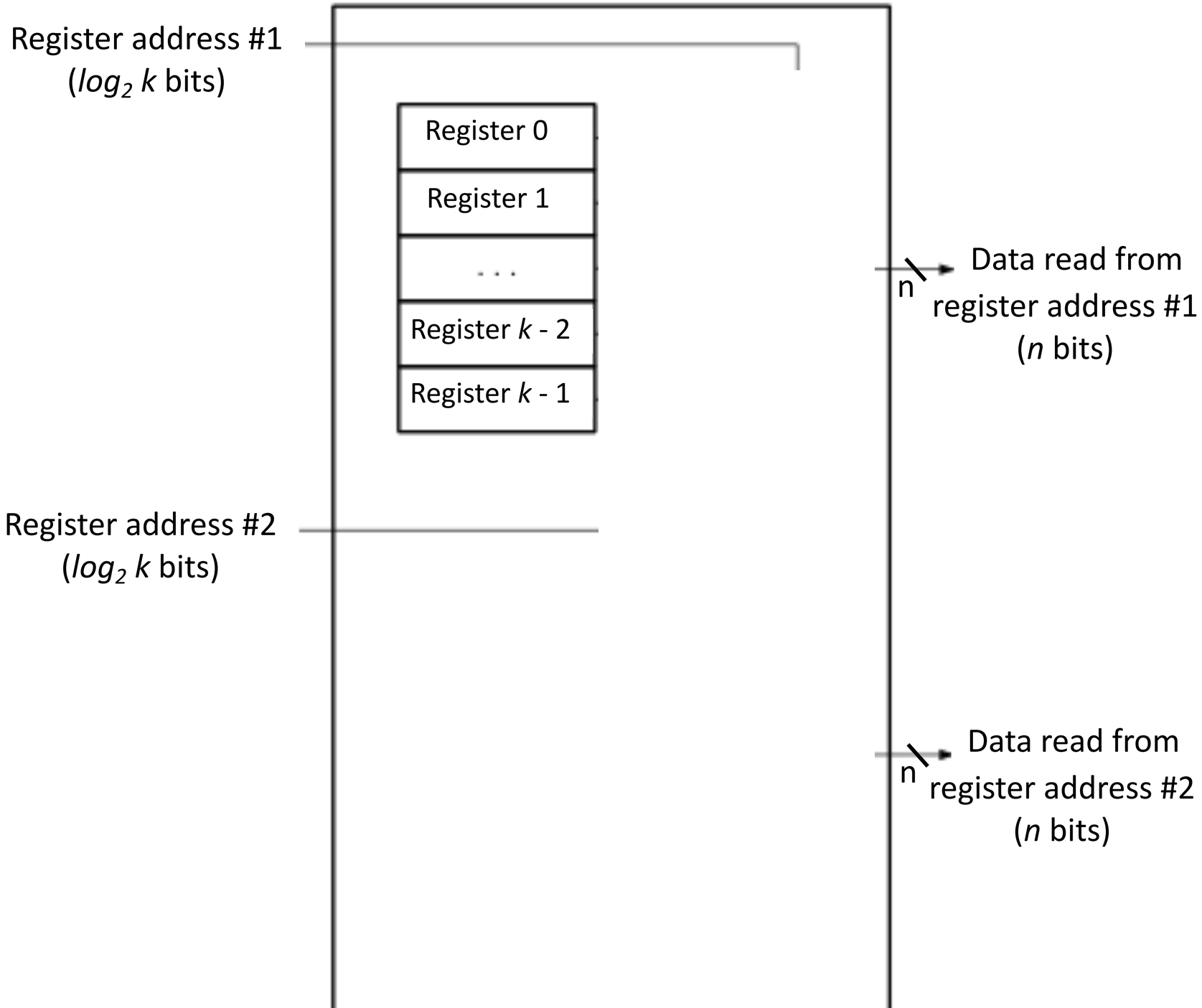


Register file

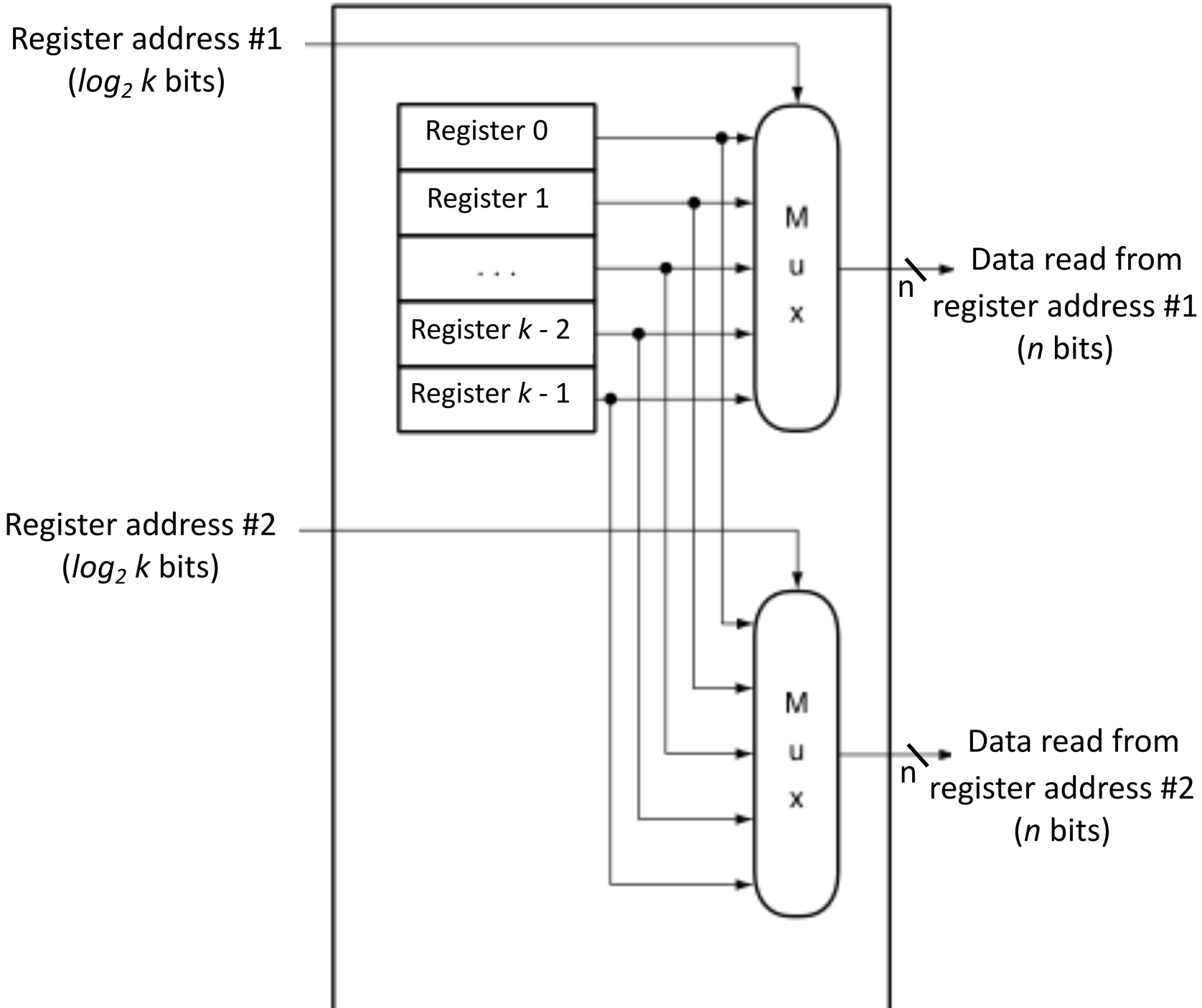


Array of registers, with register selectors, write/read control, input port for writing data, output ports for reading data.

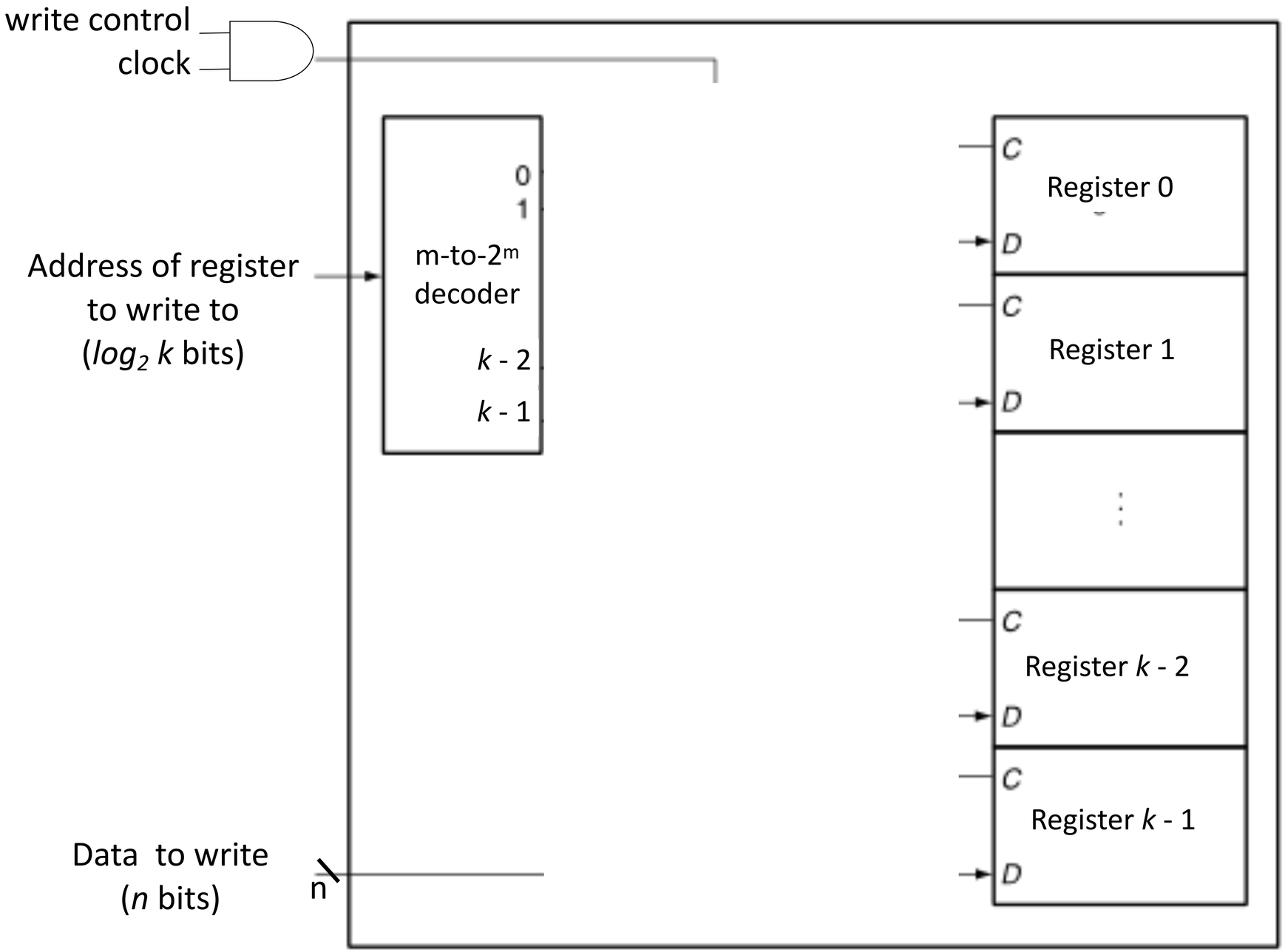
Read ports
(data out)



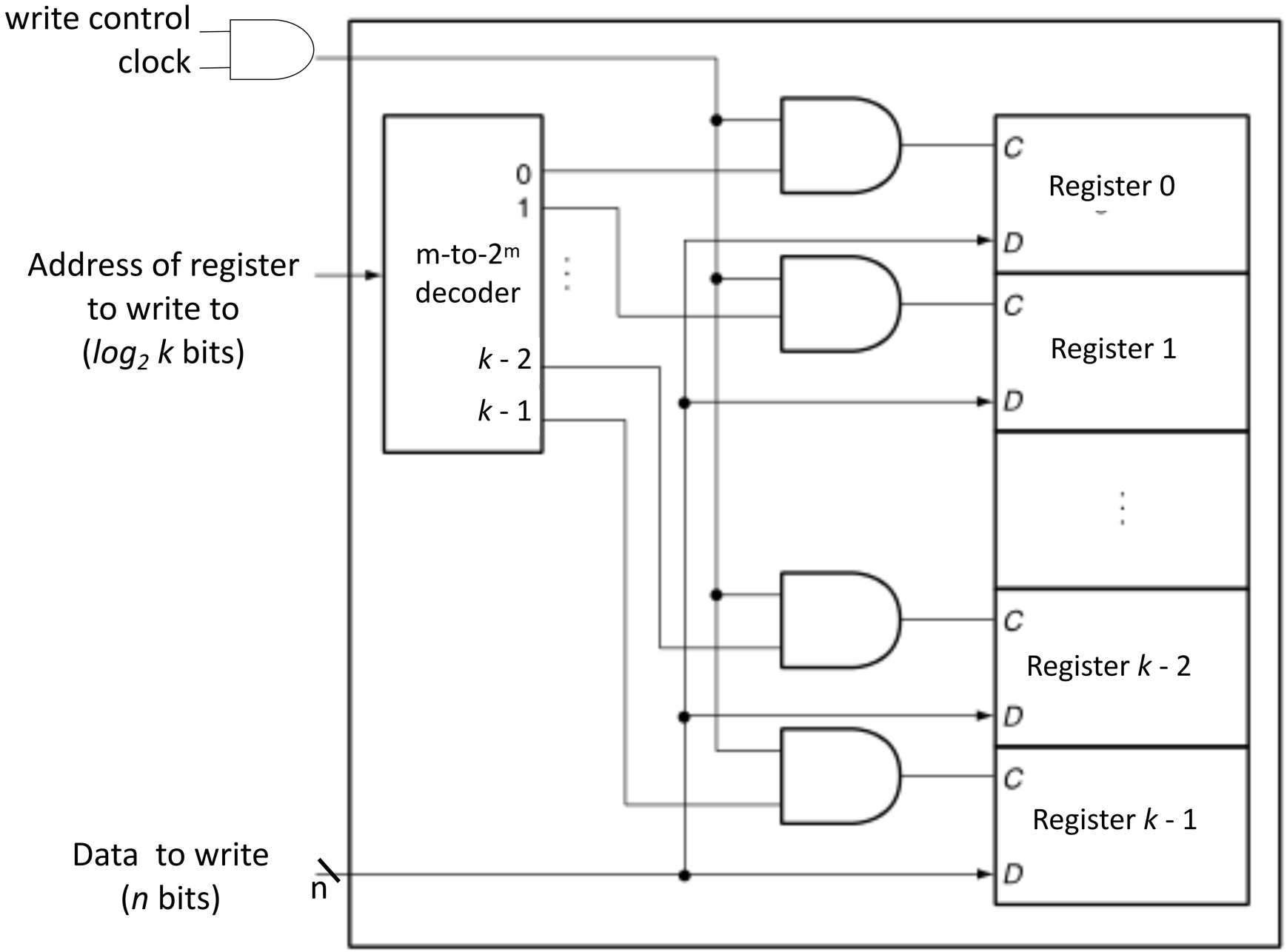
Read ports
(data out)



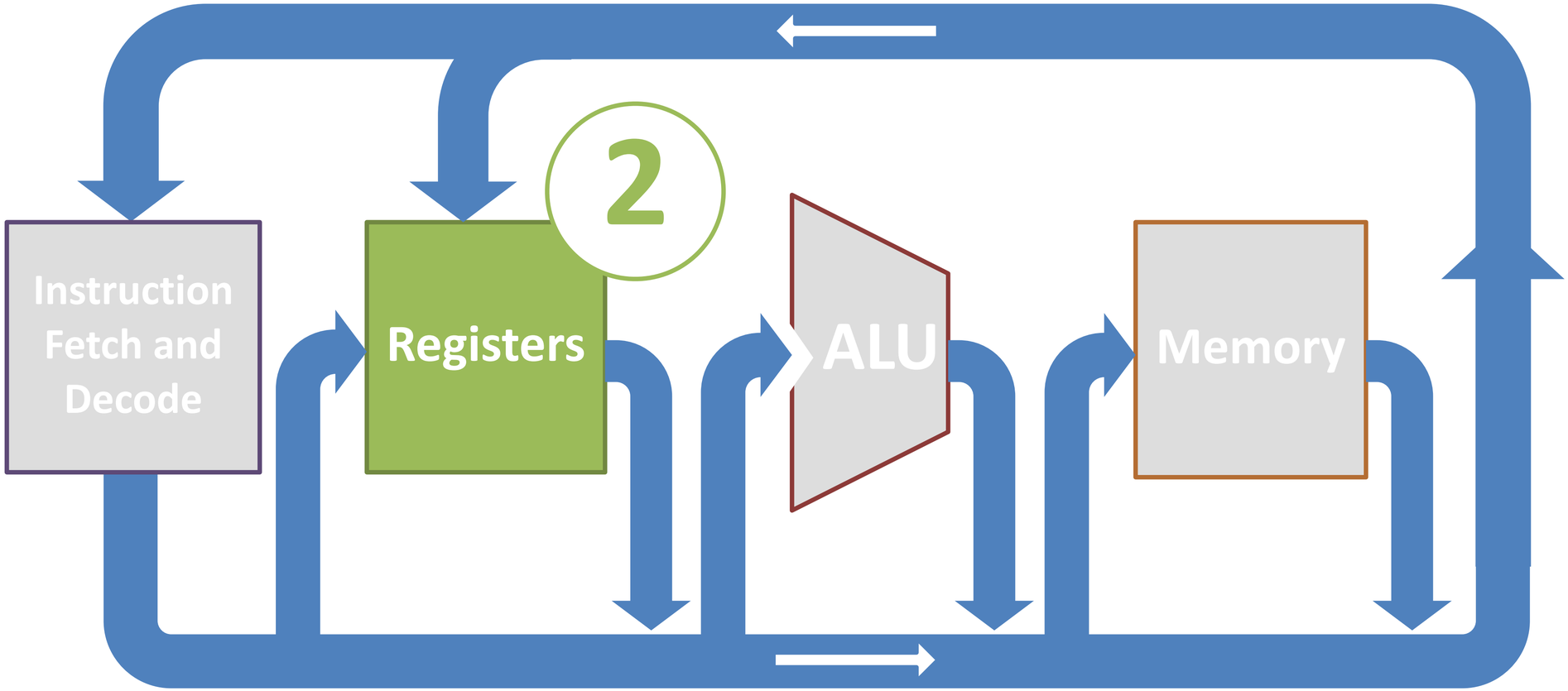
Write port
(data in)



Write port
(data in)

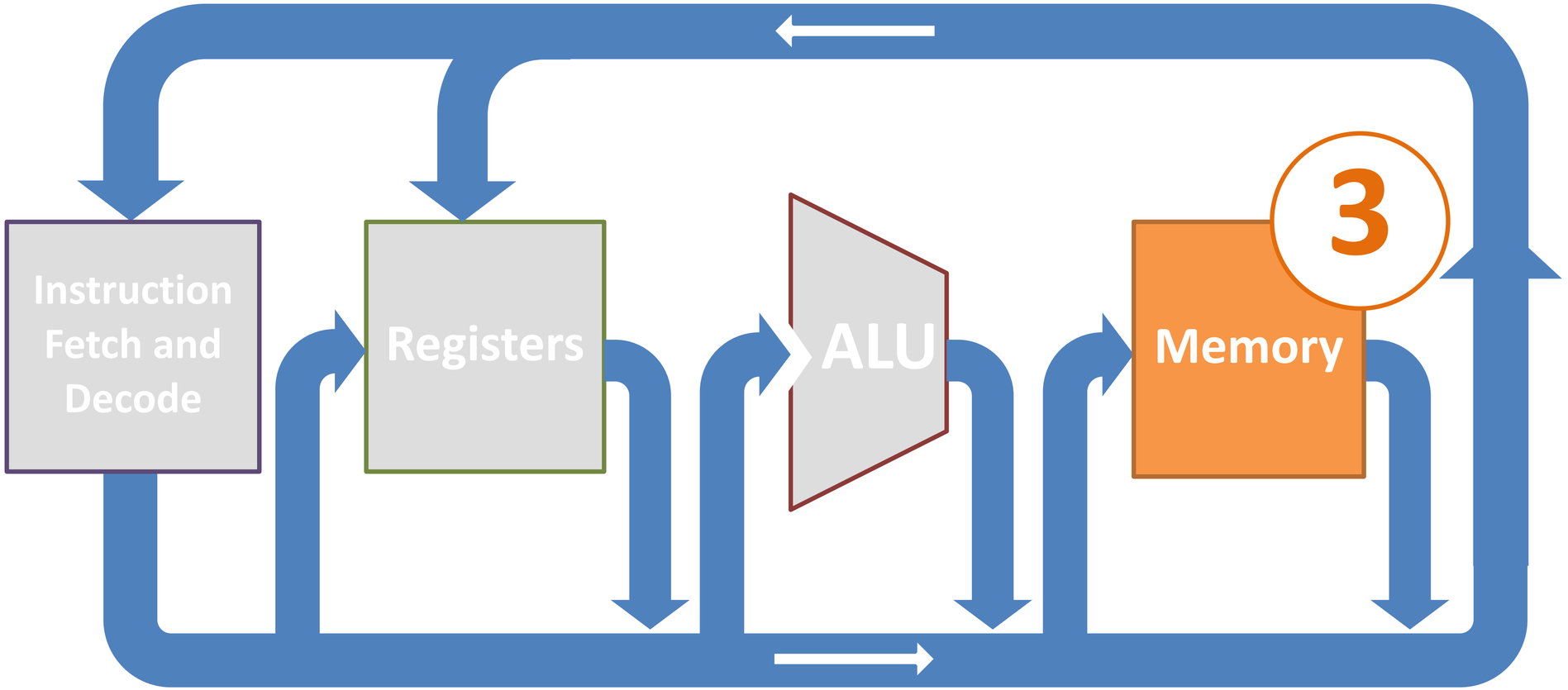


Registers summary



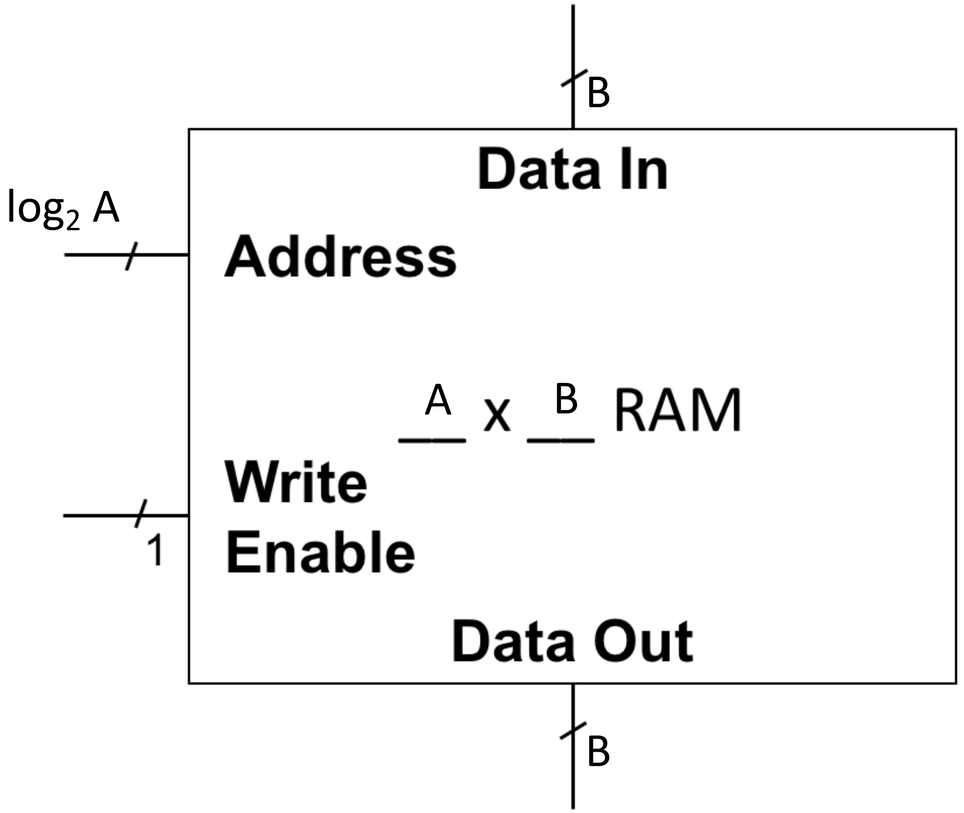
- For our purposes: implemented with flip-flops
- Very fast access
- Limited in size:
  - Need an m-to-2^m decoder
  - CPUs typically have ~10s of words of register storage

Registers summary



- We'll think about at a higher level of abstraction
- Designed to handle a much larger amount of data
  - CPUs can have millions-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