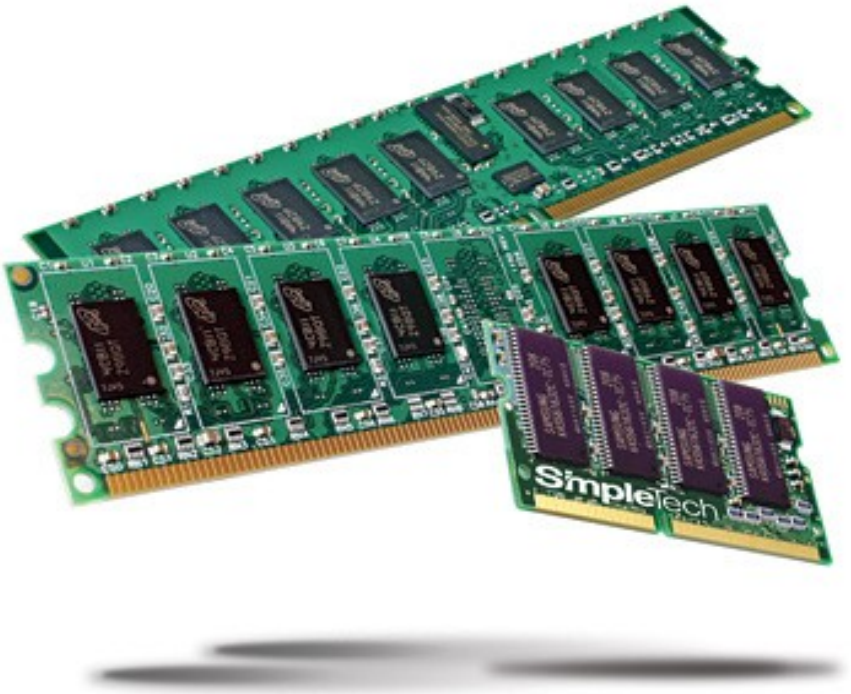
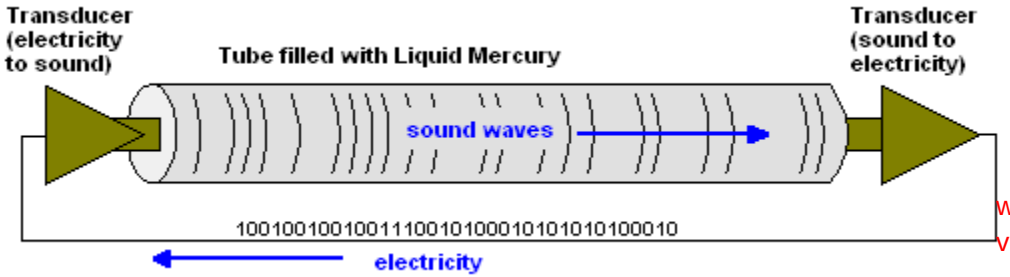
Logic Design:
Implementing **Memory**

Memory: an organism's ability to **store**, **retain** and **recall** information

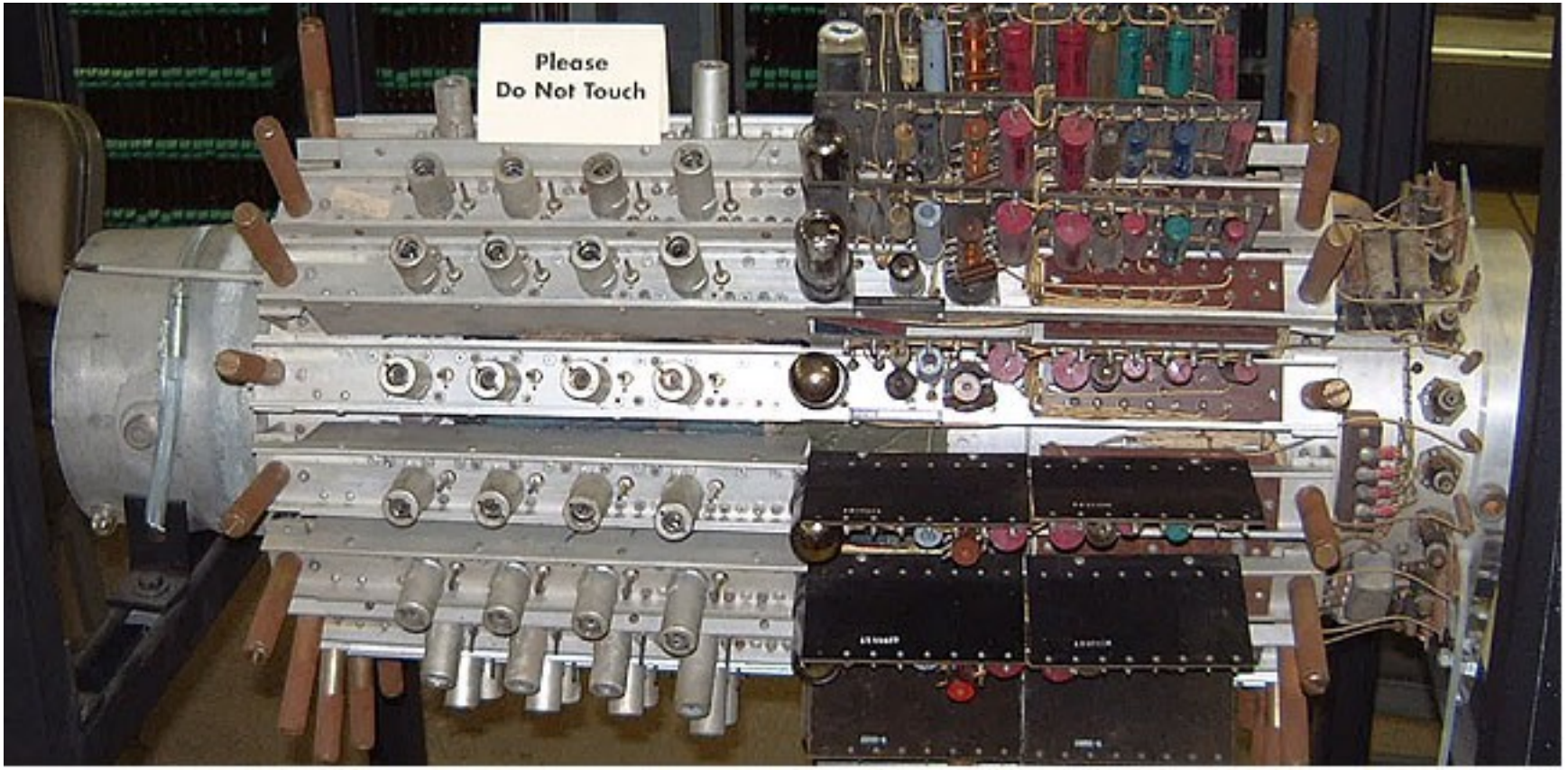


Delay Line memory

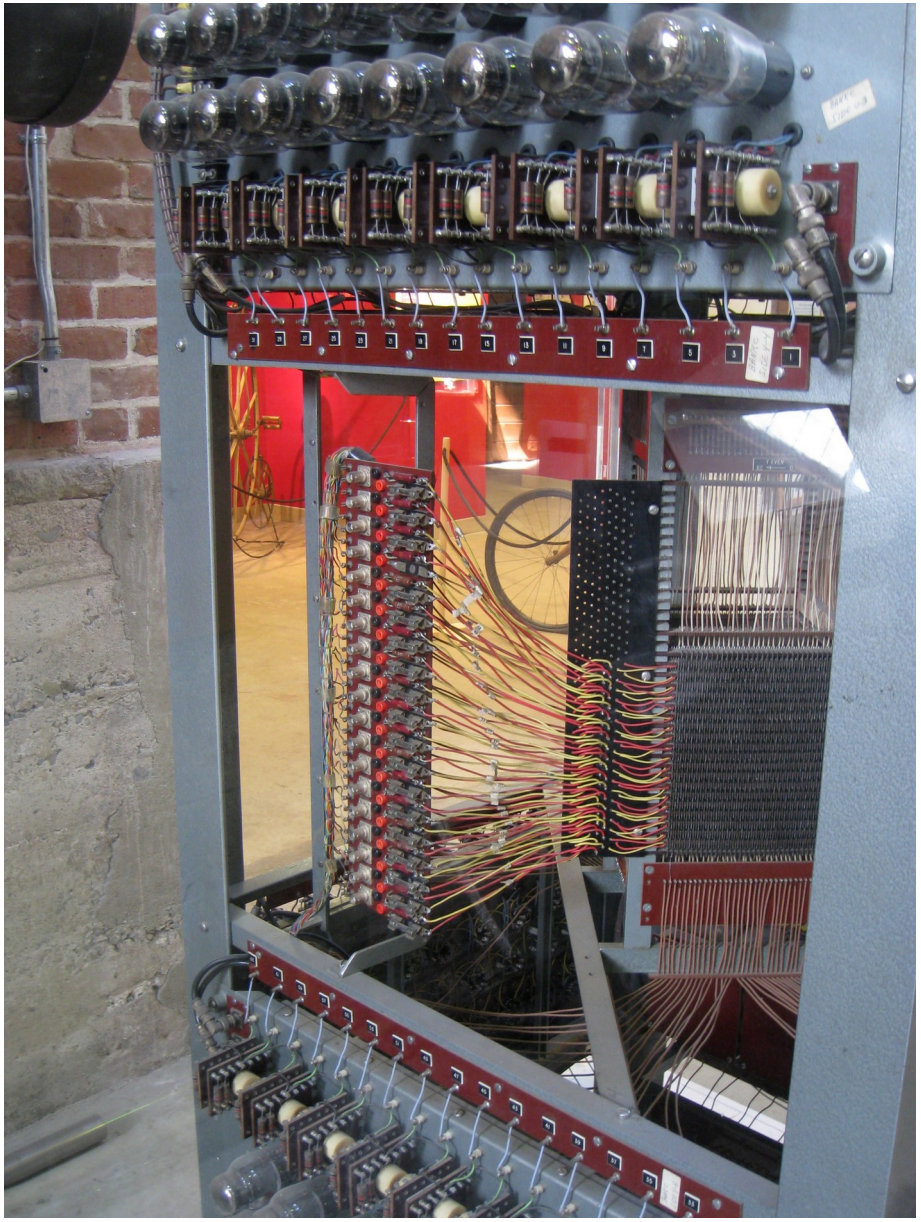


gebruikt van fysica van eigenschapepn van stoffen om iets op te slagen

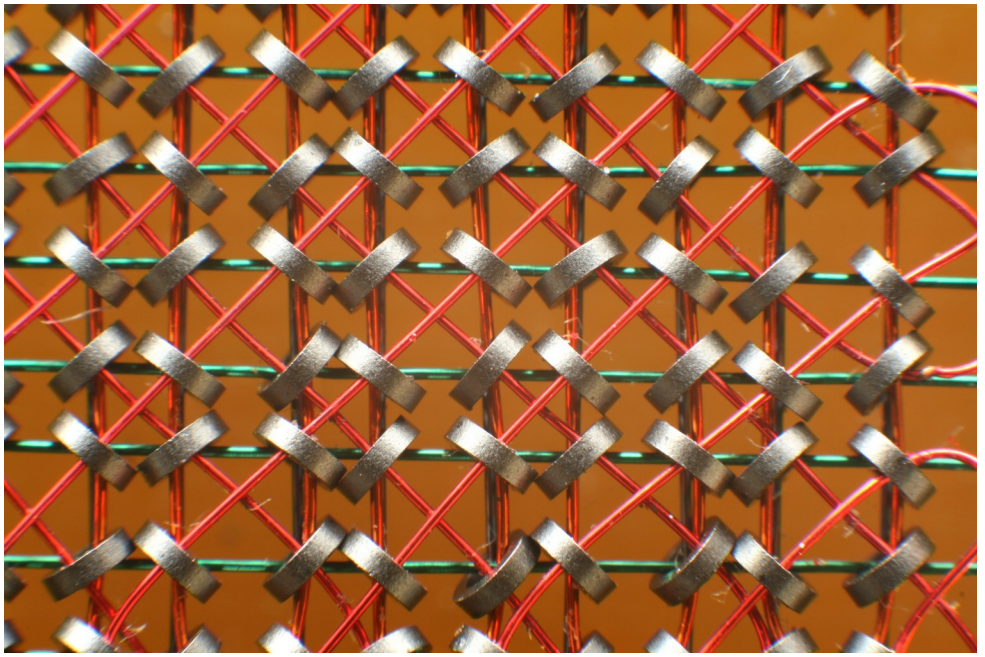
werd erg vertraagd vanbinnen om 'op te slaan' voor paar minuten



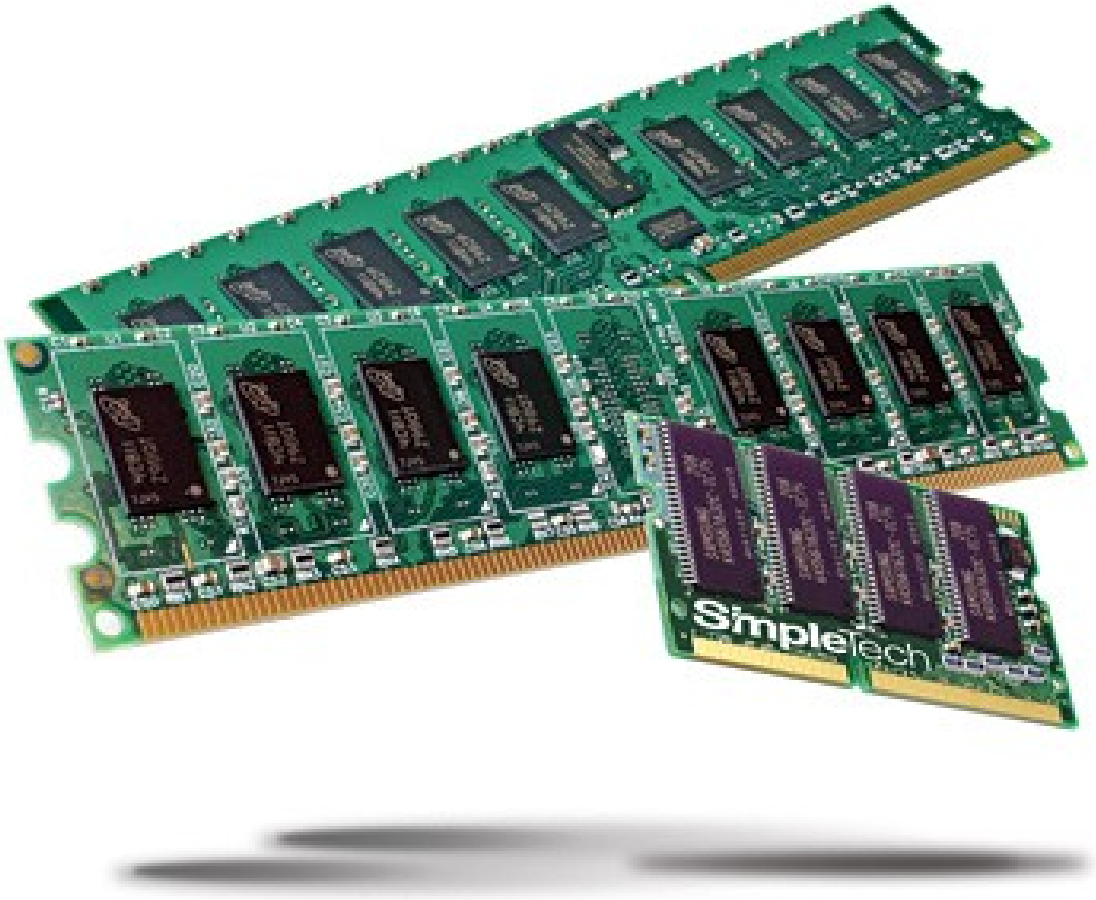
Magnetic Core memory



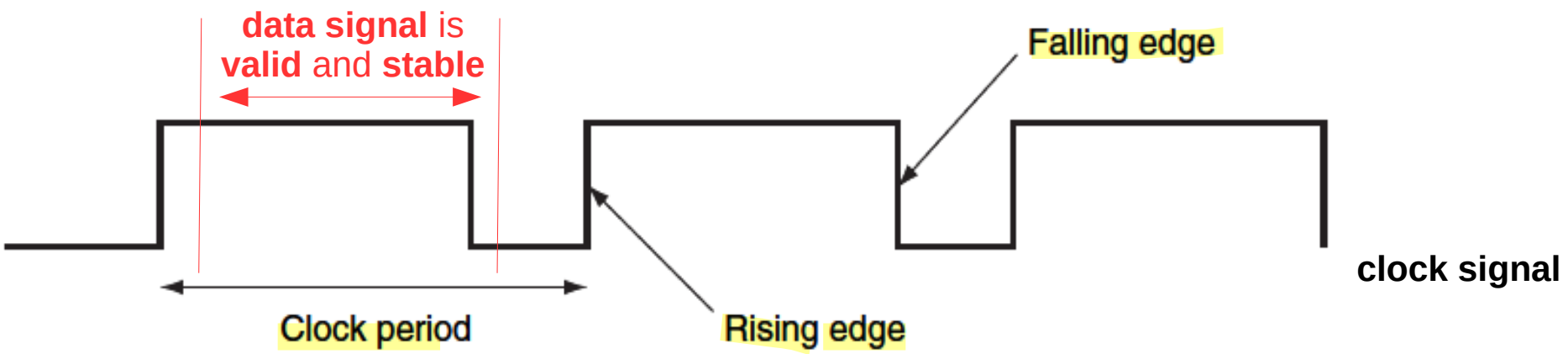
ferromagnetische ringetjes matrix, gemagnetiseerd of nt gemagnetiseerd, elk is 1 bit



Modern memory bank



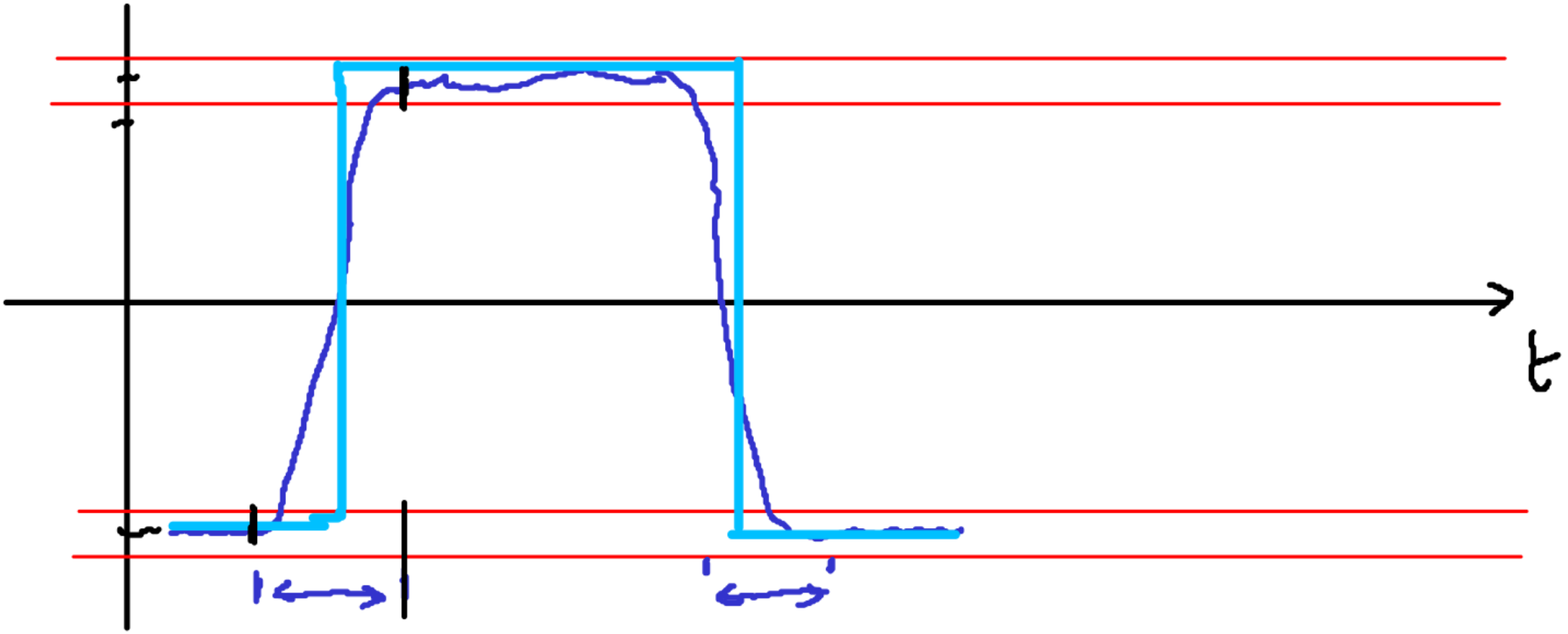
Clock Signal (**memory** retains **data** over **time**)



Clocking **methodology**:
determines **when** data is **valid** and **stable** relative to the **clock**

Edge-triggered clocking:
all **state changes** occur on a **clock edge**

Digital Signals (**analog** reality)



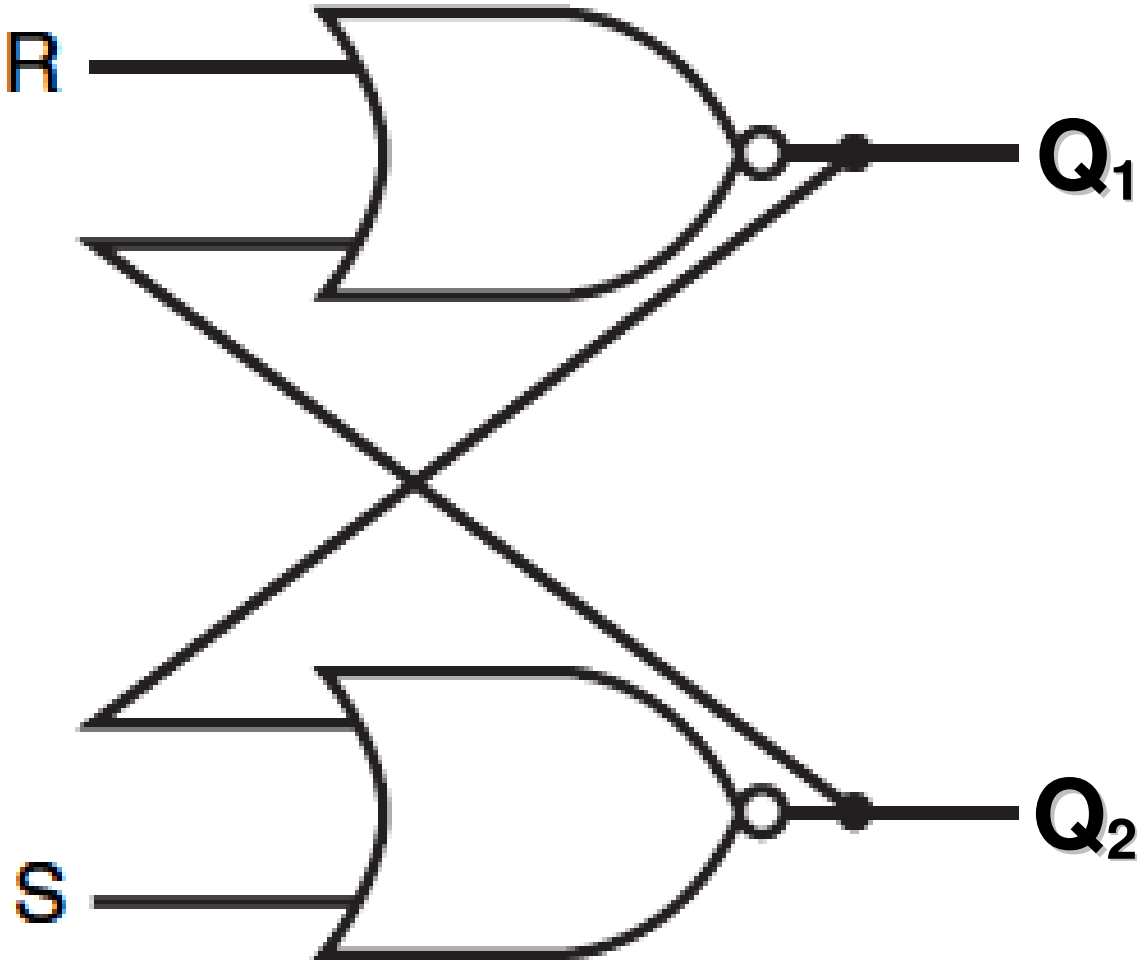
State Elements: “memory”

State Element → Combinational Logic → State Element

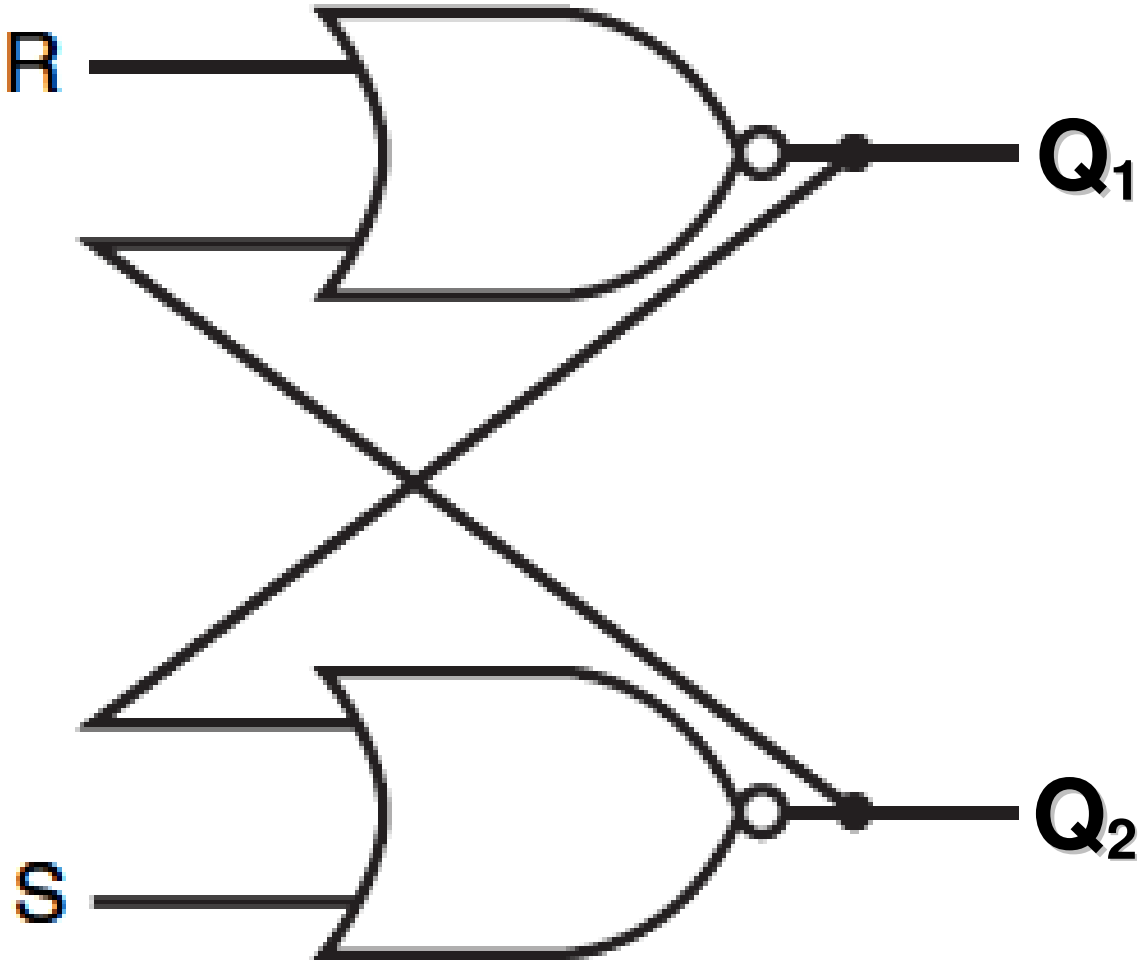
(function/computation)


Latch: store value

S-R Latch (unclocked): store and remember value

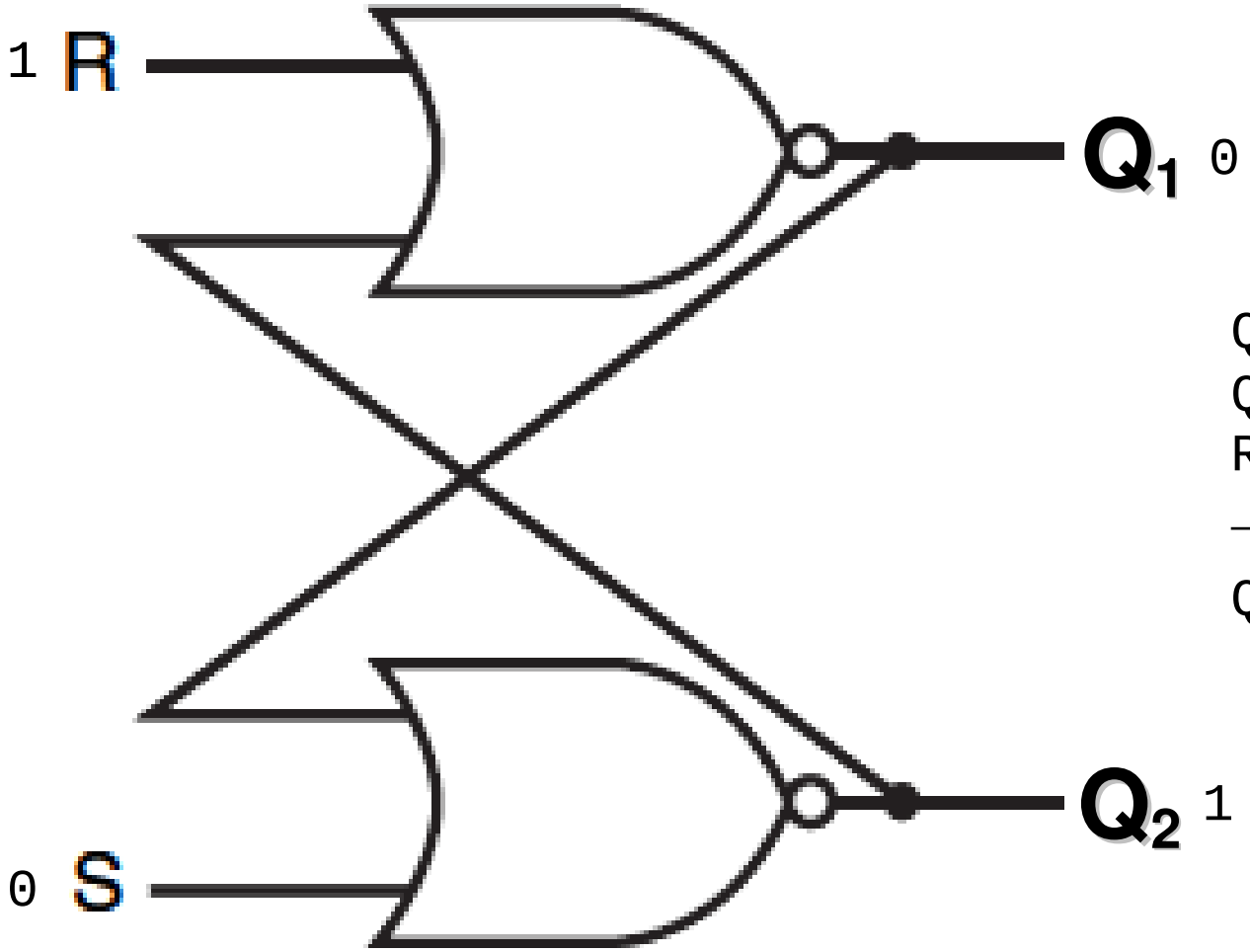


S-R Latch (unclocked): store and remember value



Q_1 = not (R or Q_2)
Q_2 = not (S or Q_1)
→
Q_1 = not (R or not (S or Q_1))
Q_2 = not (S or Q_1)
→
Q_1 = not R and (S or Q_1)
Q_2 = not (S or Q_1)

S-R Latch set

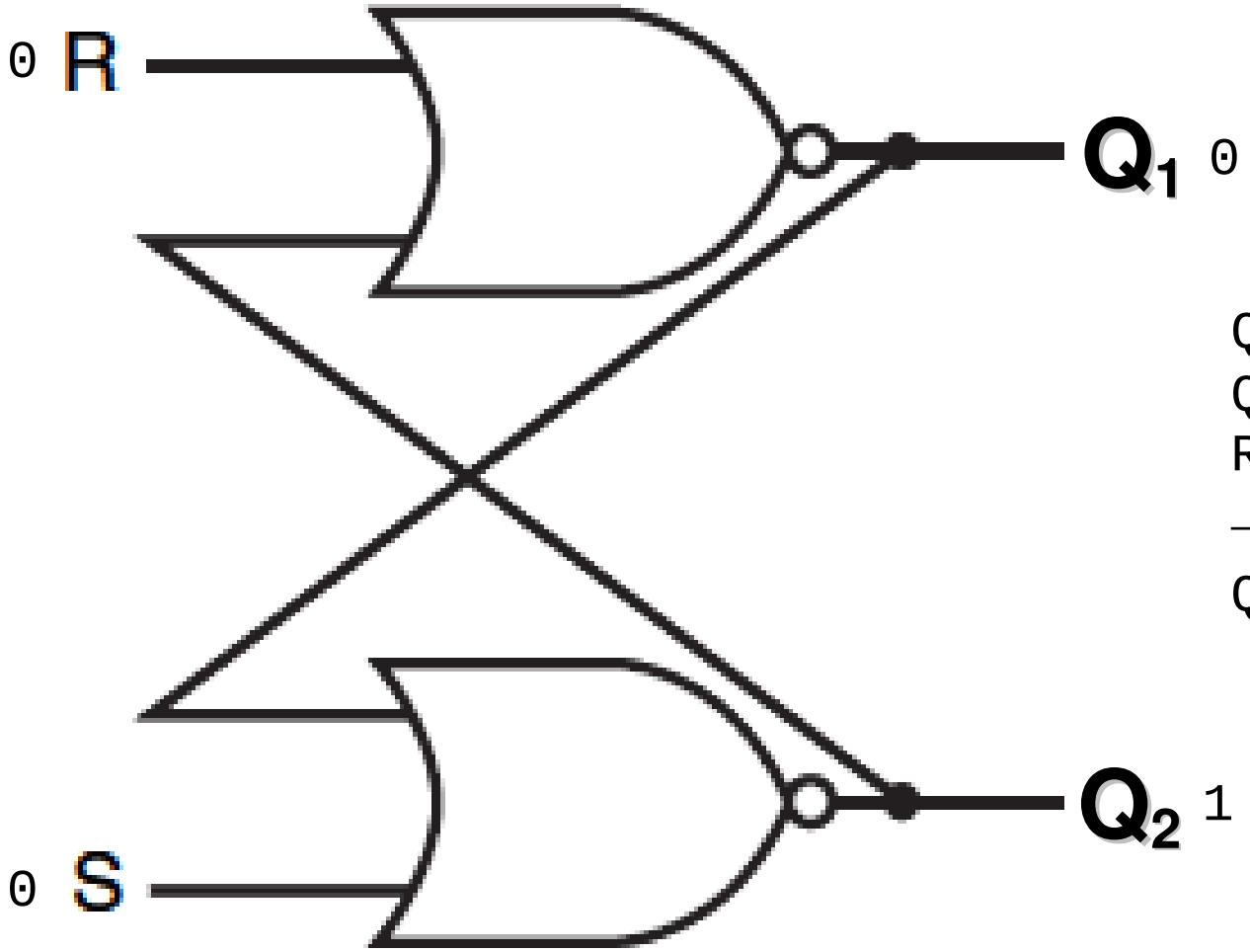


Q_1 = not R and (S or Q_1)
Q_2 = not (S or Q_1)
R = 1, S = 0

→

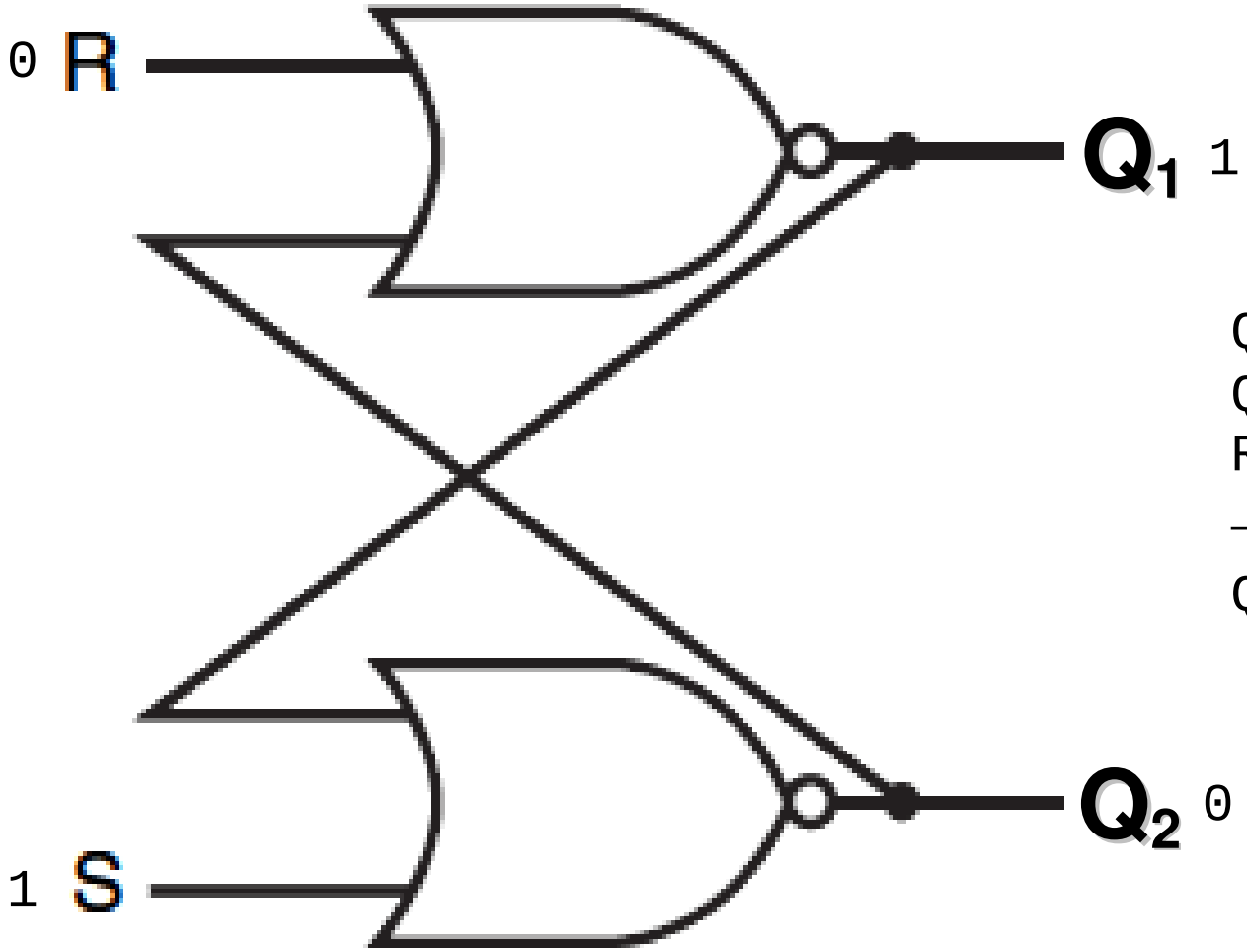
Q_1 = 0, Q_2 = not Q_1 = 1

S-R Latch remember output value



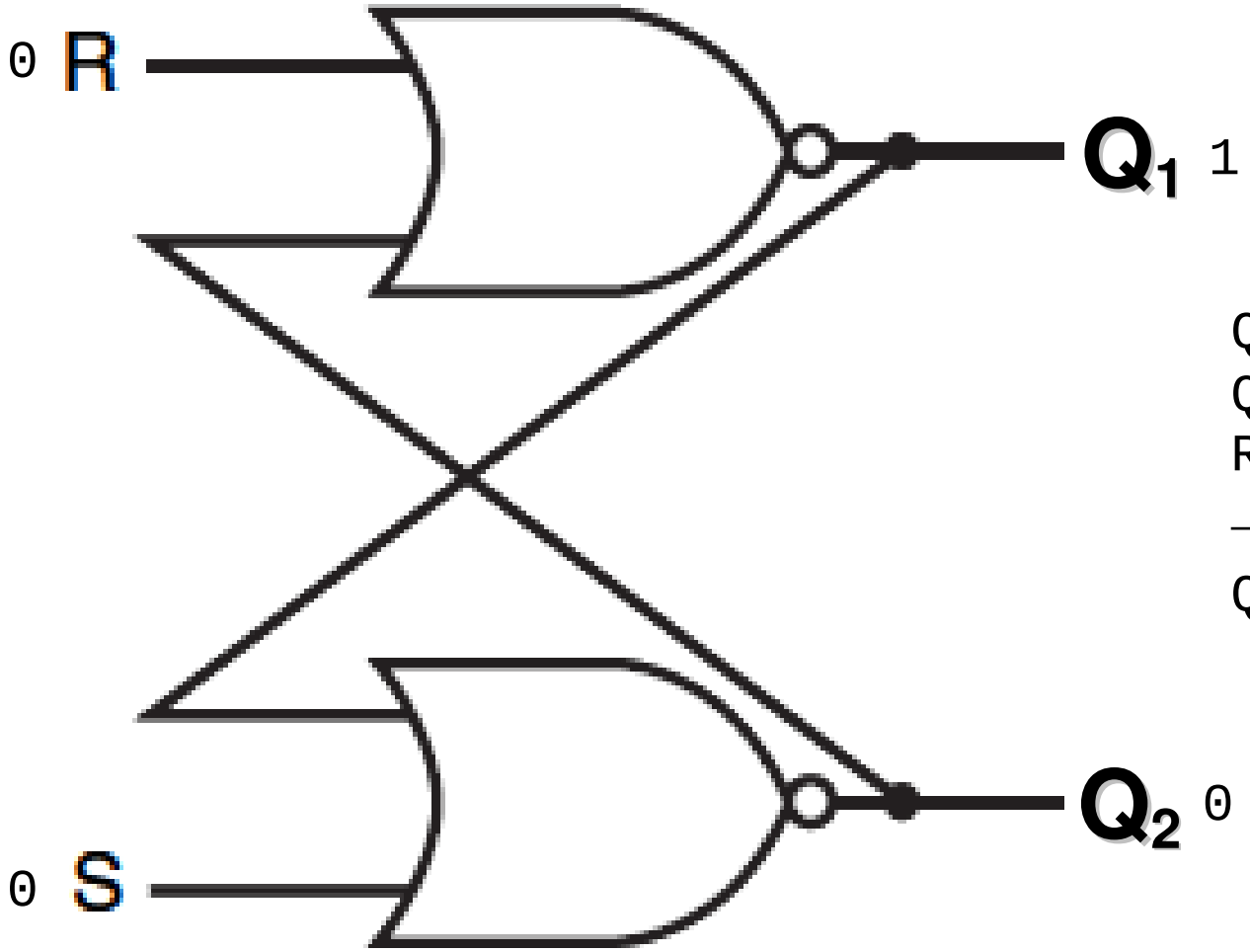
Q_1 = not R and (S or Q_1)
Q_2 = not (S or Q_1)
R = 0, S = 0
→
Q_1 = Q_1, Q_2 = not Q_1

S-R Latch set



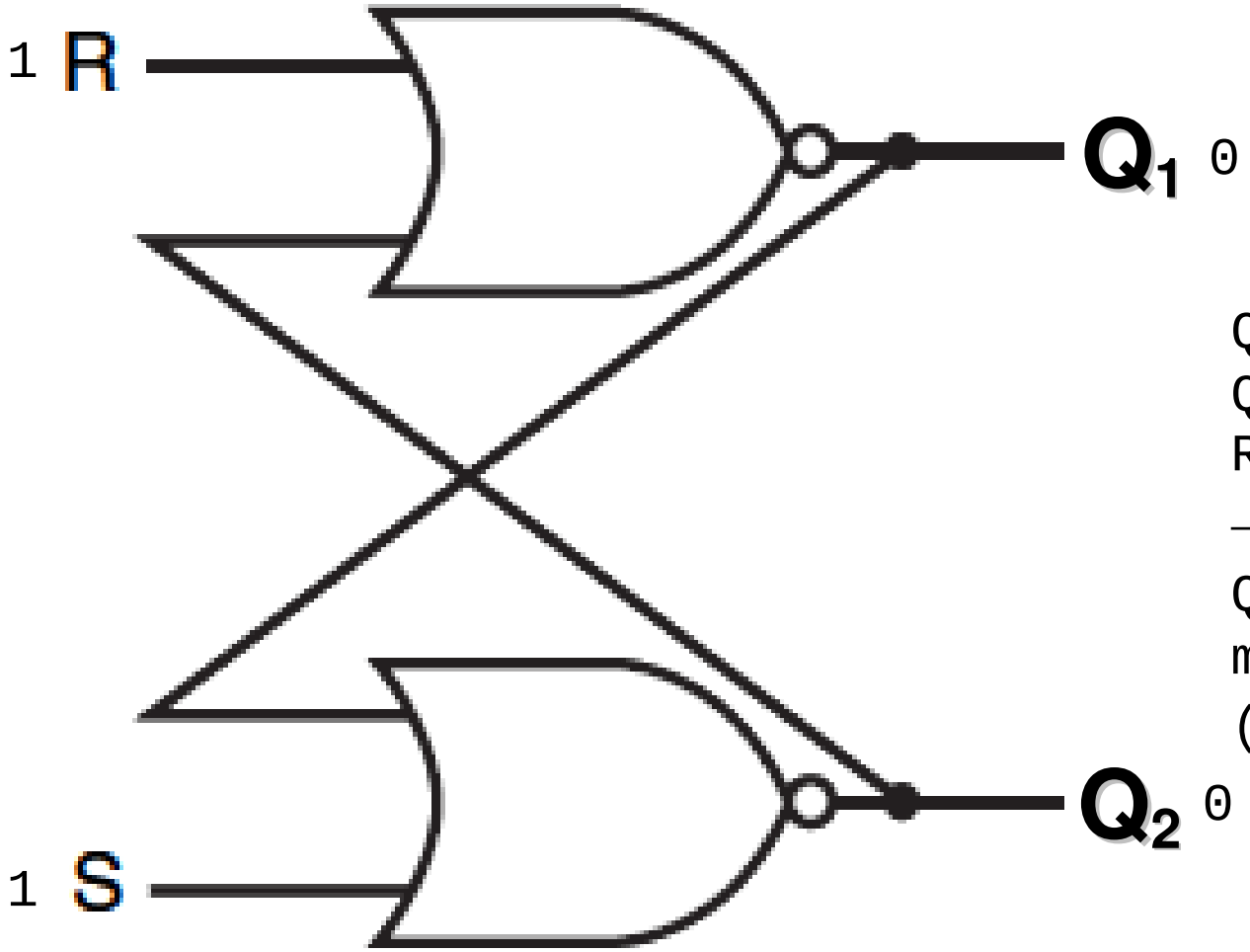
Q_1 = not R and (S or Q_1)
Q_2 = not (S or Q_1)
R = 0, S = 1
→
Q_1 = 1, Q_2 = 0 = not Q_1

S-R Latch remember output value



Q_1 = not R and (S or Q_1)
Q_2 = not (S or Q_1)
R = 0, S = 0
→
Q_1 = Q_1, Q_2 = not Q_1

S-R Latch – no longer Q_2 = not Q_1



Q_1 = not R and (S or Q_1)
Q_2 = not (S or Q_1)
R = 1, S = 1
→
Q_1 = 0, Q_2 = 0
may become meta-stable
(**analog** reality: delays)

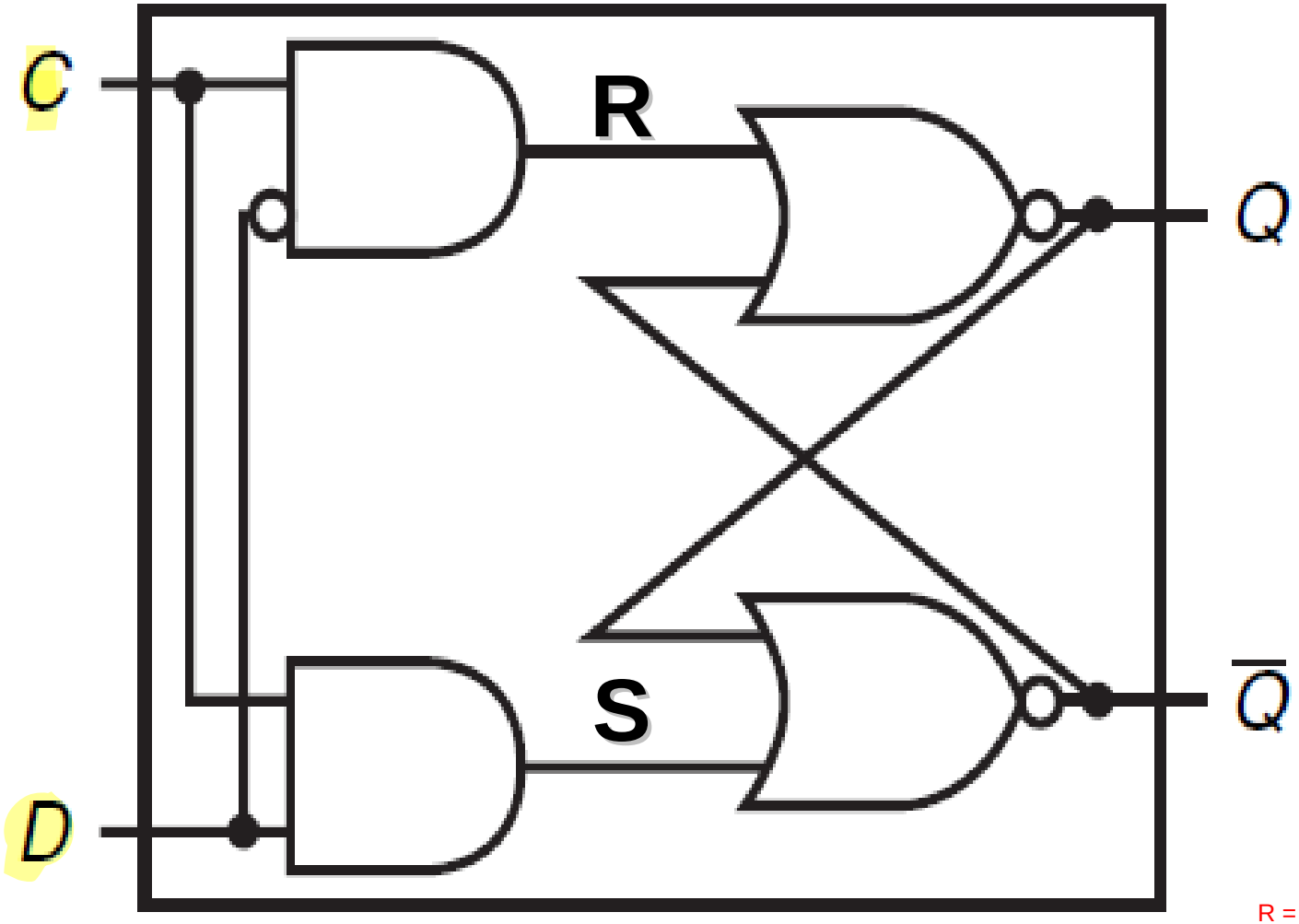
Latch:
change **while** clock asserted
(clock connected to R or S)

Flip-Flop:
change **on** clock **edge**

D latch (“transparent”)

data

C: “Clock”
D: “Data”



R = C and not D
S = C and D
→

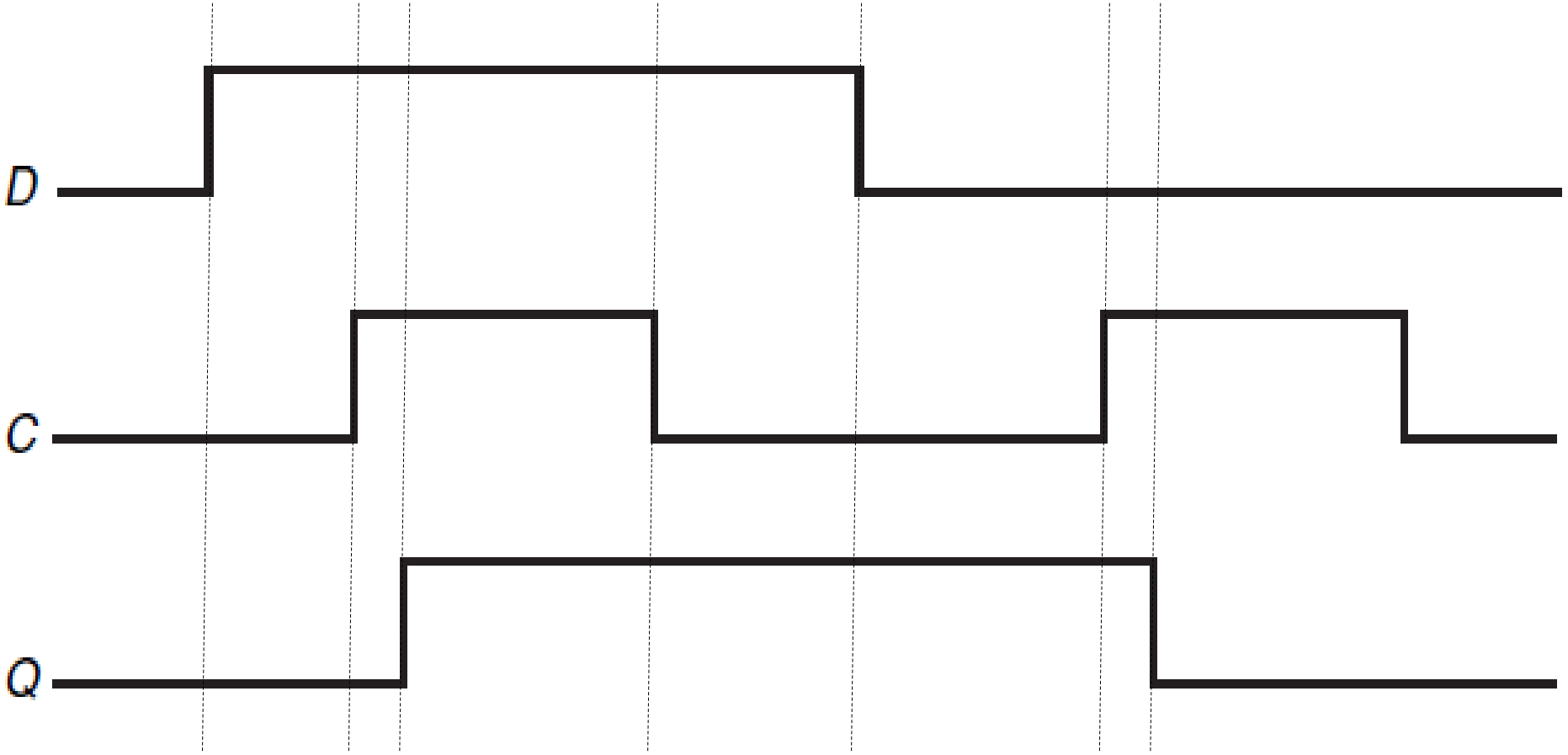
when C = 1: “open”
R = not D
S = D
→ Q = S = D

when C = 0: “closed”
R = 0
S = 0
→ Q is not changed

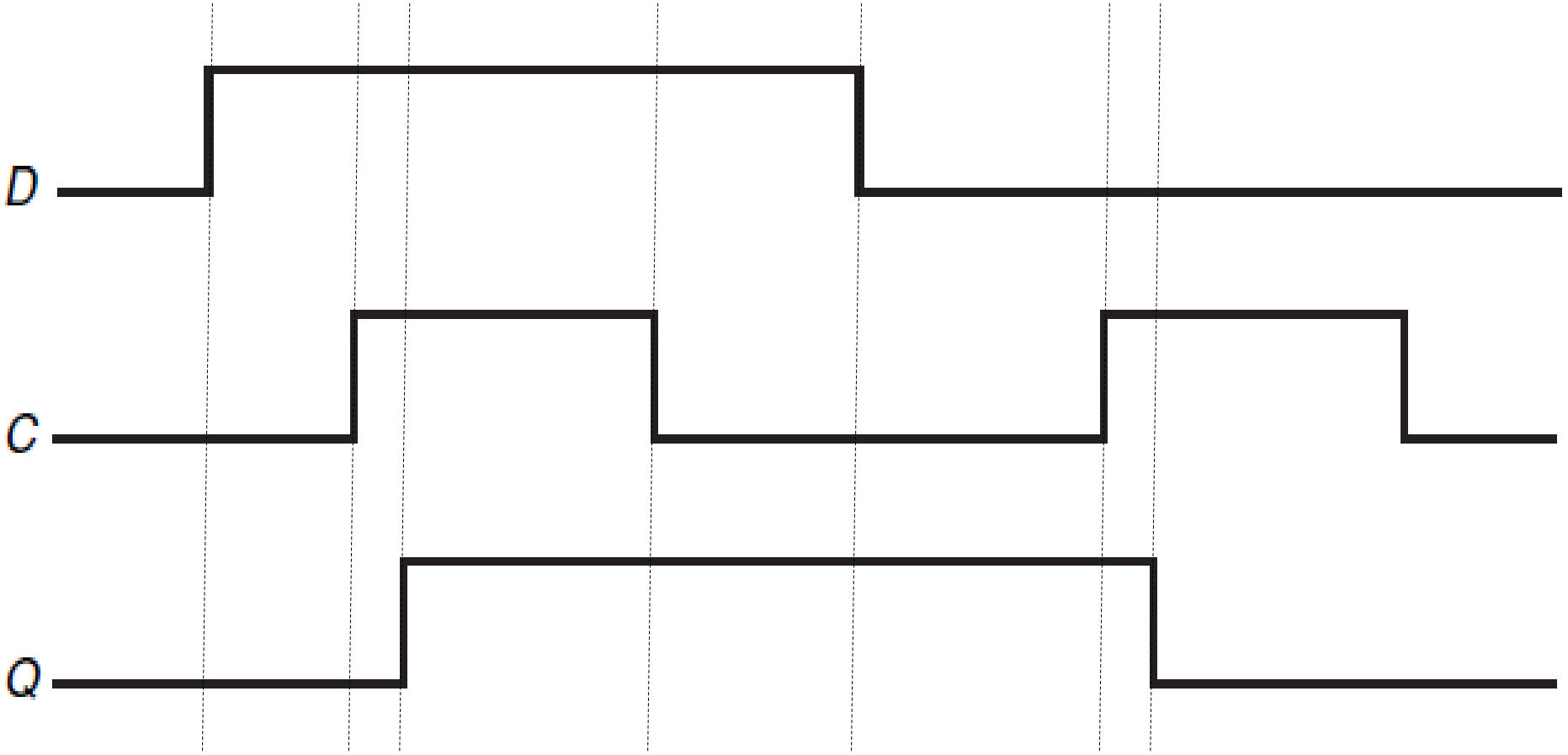
R = S = 1
not possible

R = 1 en S = 1 not possible
will destablise and ruin the latch

D latch operation



D latch operation



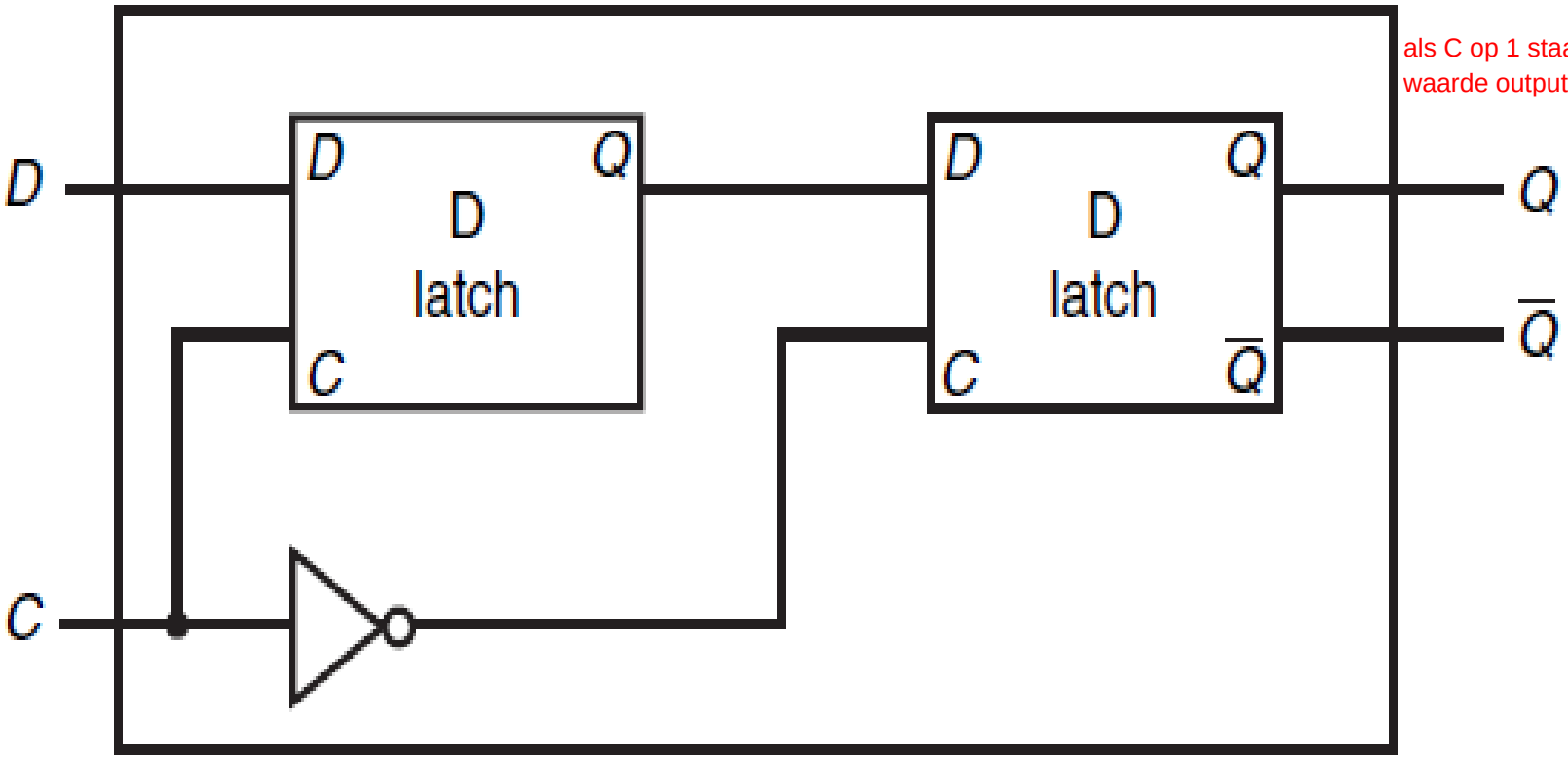
What if D changes during C asserted?

When **C** is asserted (high), the output **Q** follows the input **D**. If **D** changes while **C** is high, **Q** will update to match **D**.

D flip-flop (not “transparent”) with falling-edge trigger

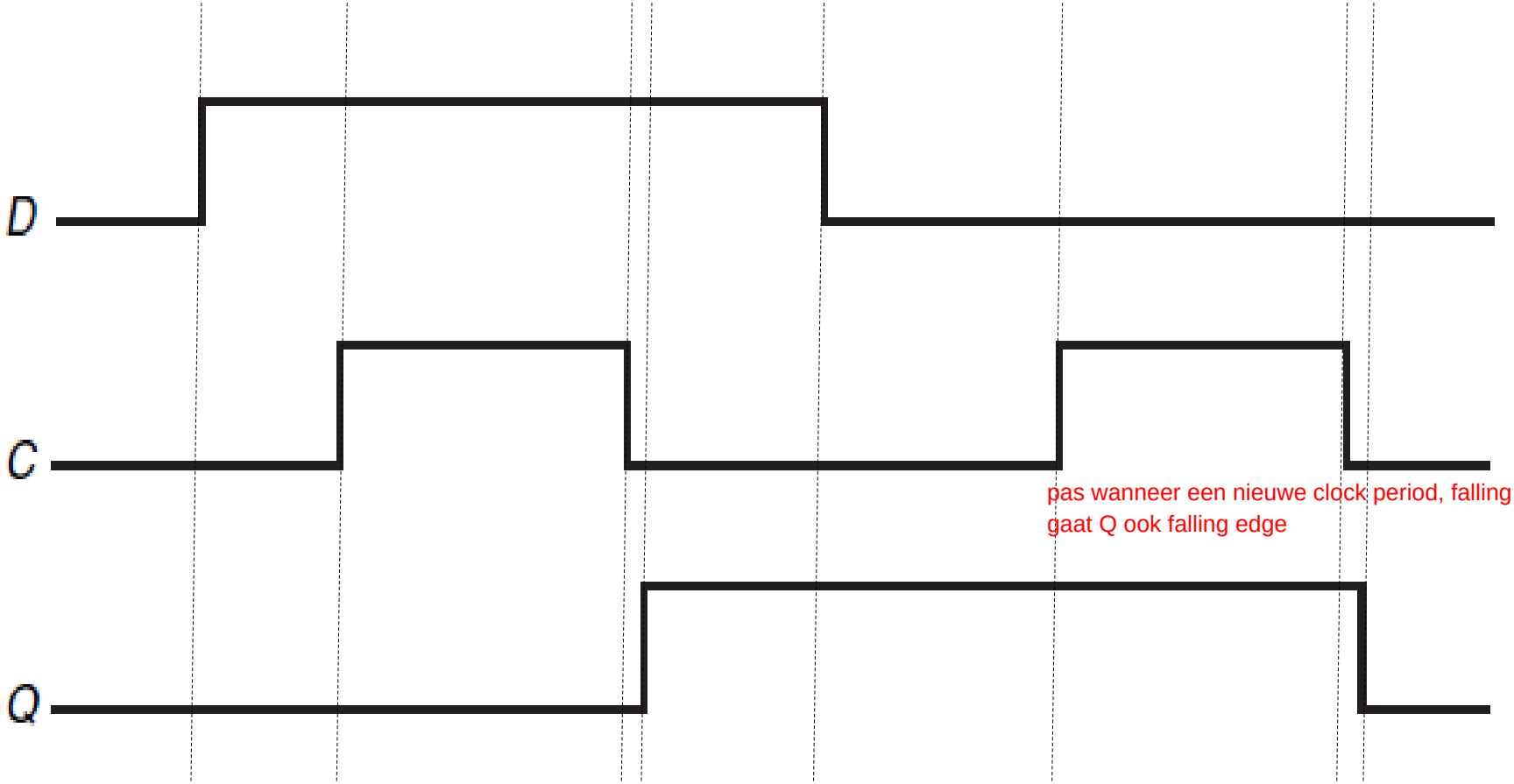
ongeacht wat er eerst veranderd, als C op 0 staat, gaat het terug nr oude Q

als C op 1 staat, zal nieuwe Q waarde output zijn



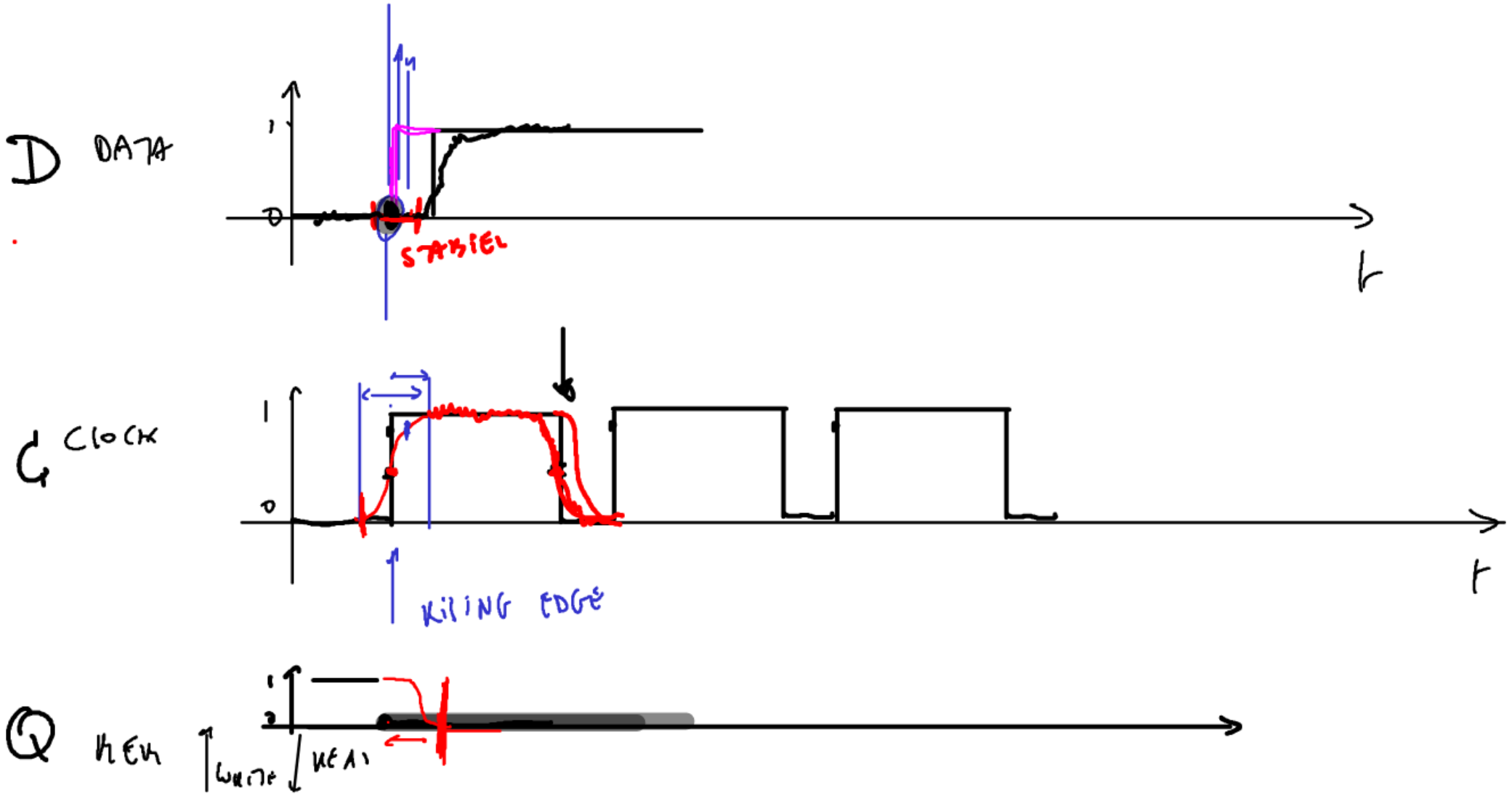
data komt binnen, C staat op 1 , data wijzigt, maar op moment van falling edge van clock, wordt laatste waarde bewaard

D flip-flop
with falling-edge trigger operation

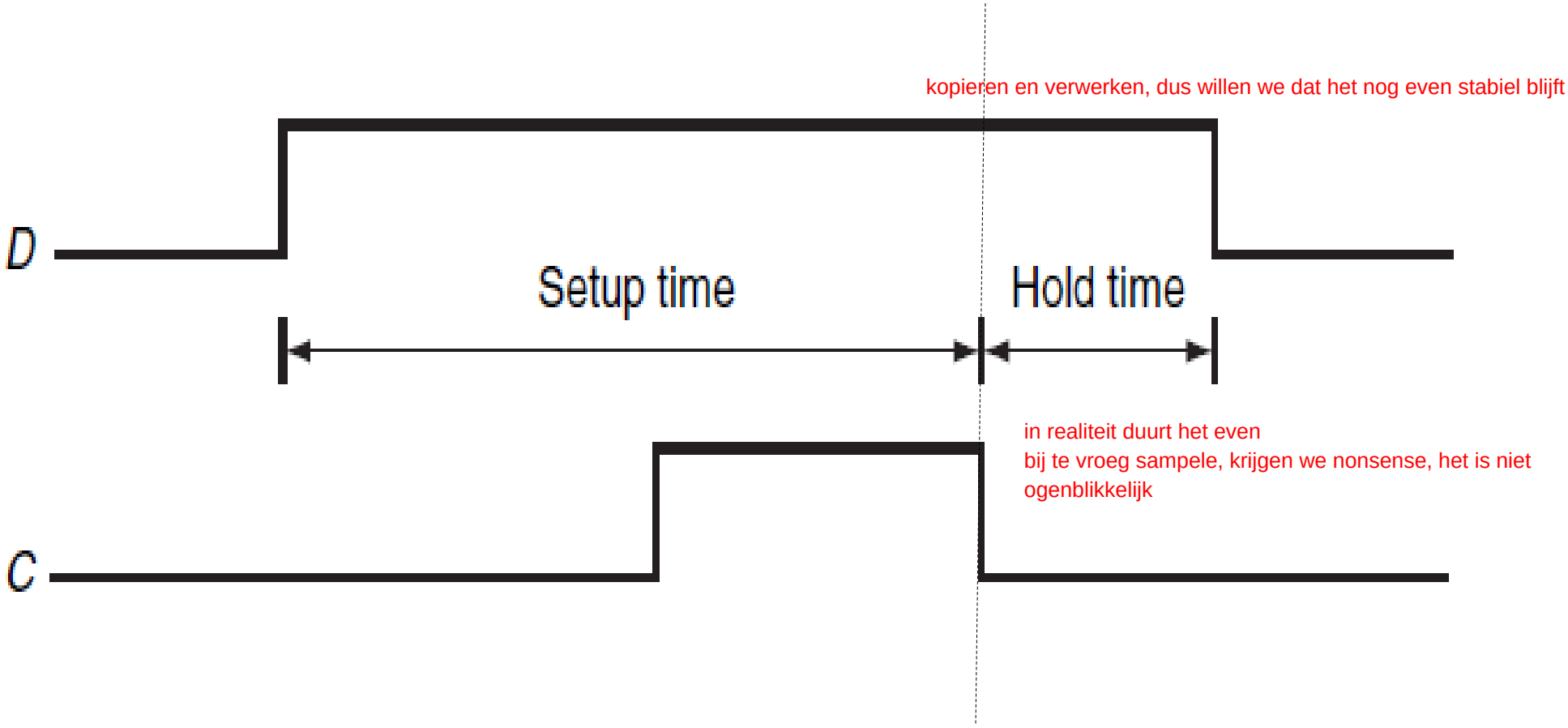


pas wanneer een nieuwe clock period, falling edge gebeurt, gaat Q ook falling edge

timing constraints: “leaky” abstraction (rising edge)



D flip-flop (falling edge) timing constraints

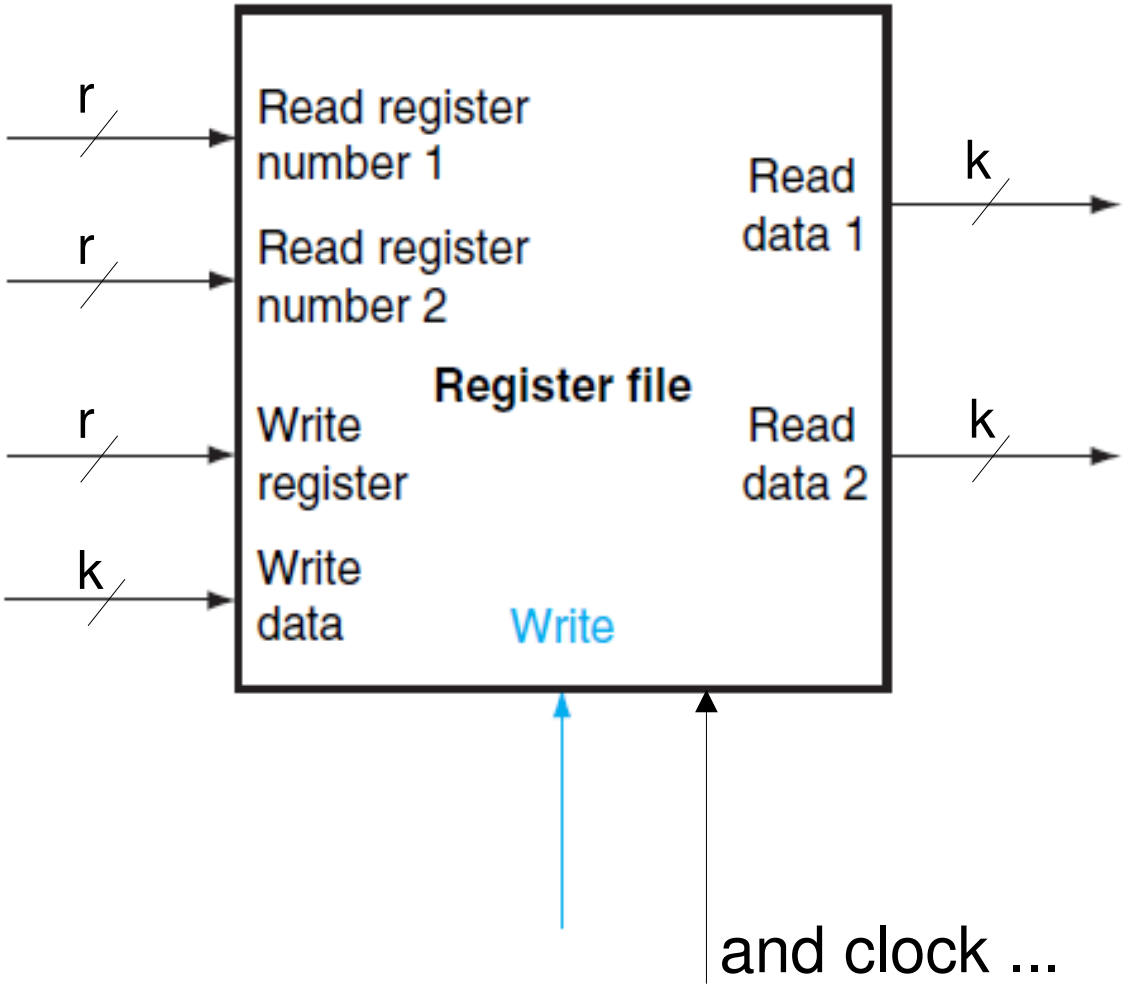


Register File

k-bit registers
(using k D flip-flops, see later: 4x2 SRAM)

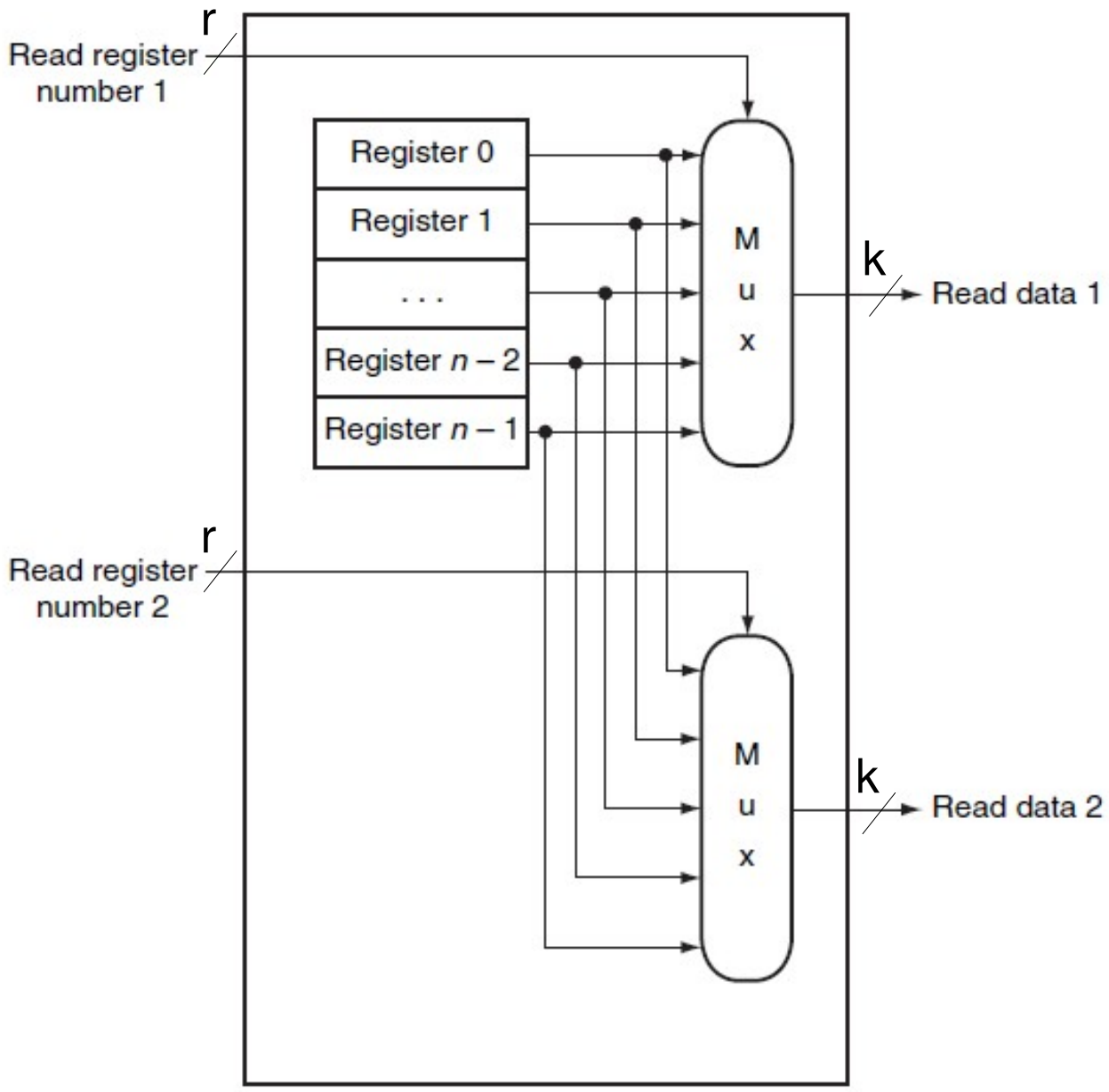
number of registers n

→ $r = \lceil \log_2 n \rceil$

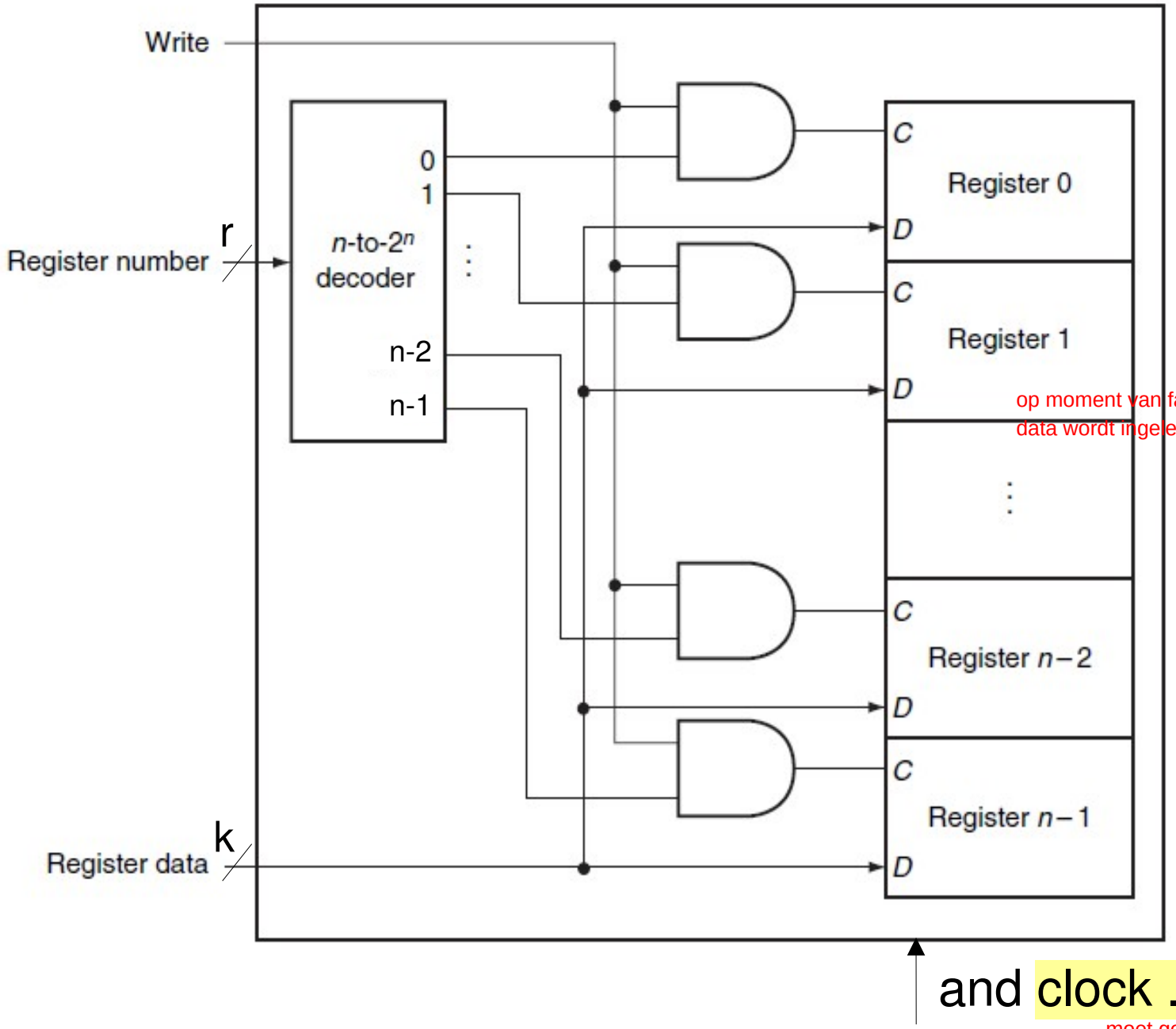


Register File: read

selecteren van een bep. adres
=> vb 2 milj, multiplexor met 2 milj wordt moeilijk



Register File: write

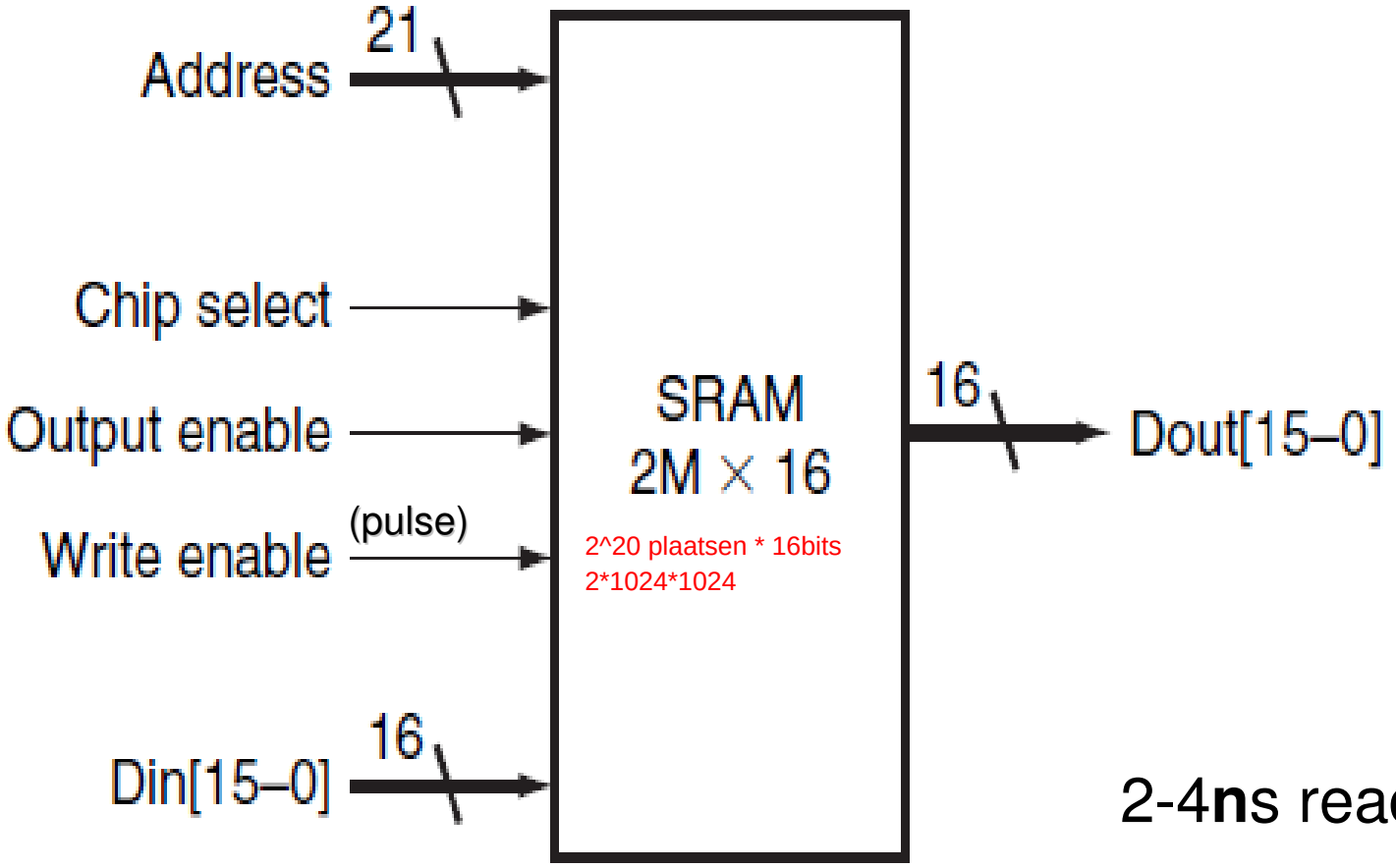


op moment van falling edge data wordt ingelezen

moet ge-and worden met data

Large Memory:

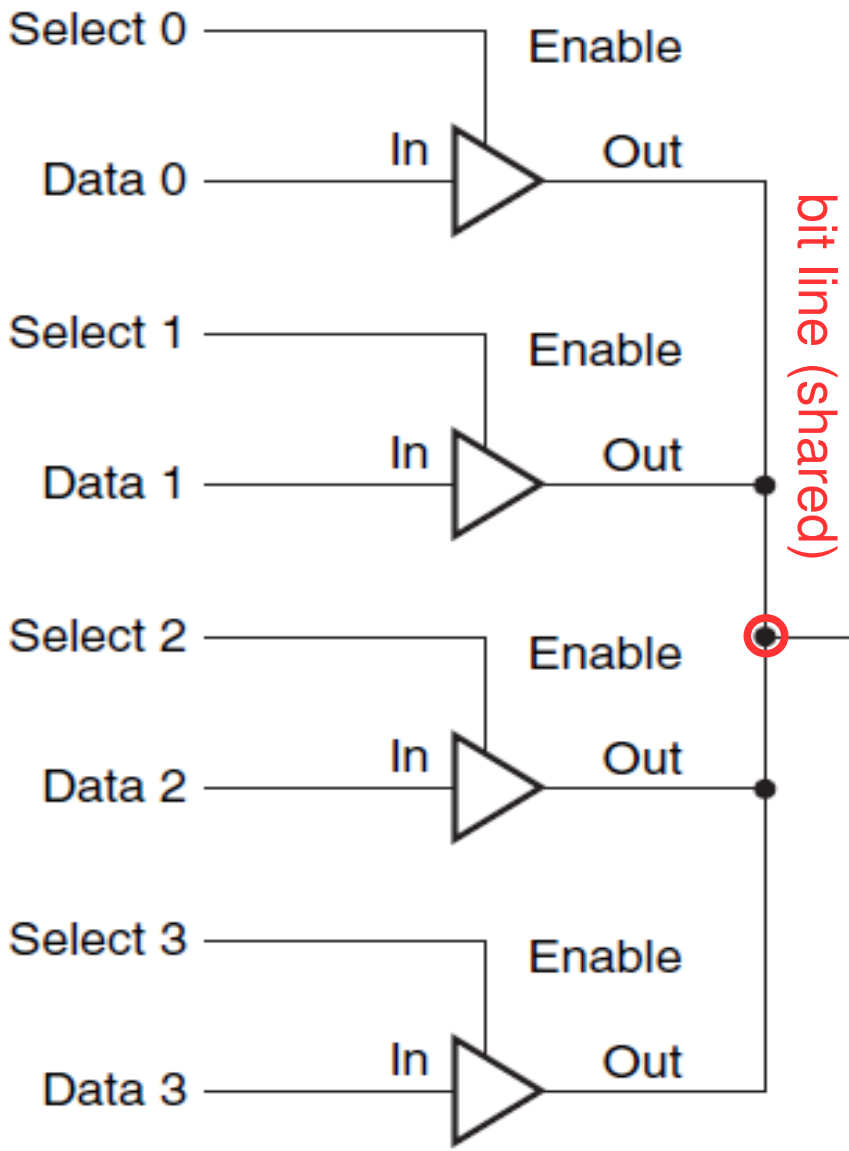
Static Random Access Memory (SRAM)
vs. Dynamic Random Access Memory (DRAM)



2-4**n**s read time

Random Access vs. Sequential Access: access time
RAM vs. ROM (Read Only Memory)

Three-state buffers (replace multiplexer)



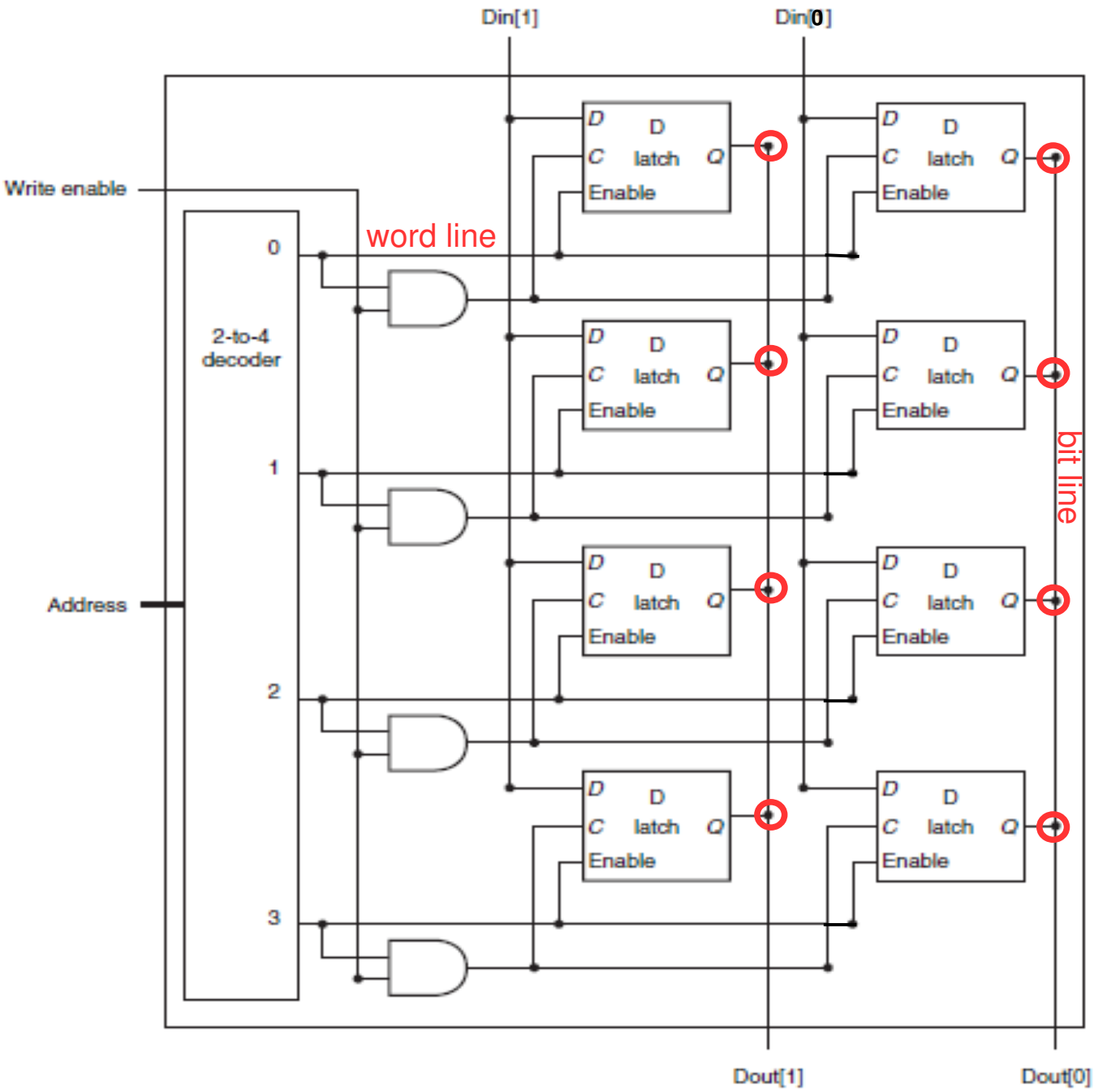
- assered
- de-asserted
- high-impedance

niet asserted niet de-asserted, gewoon doorgelaten
want kortsluiting als het ene 0V is en de andere 5V

4x2 SRAM (output enable, chip select omitted)

4 locaties met 2 bits

multiplexor al weg,
maar wel grote decoder

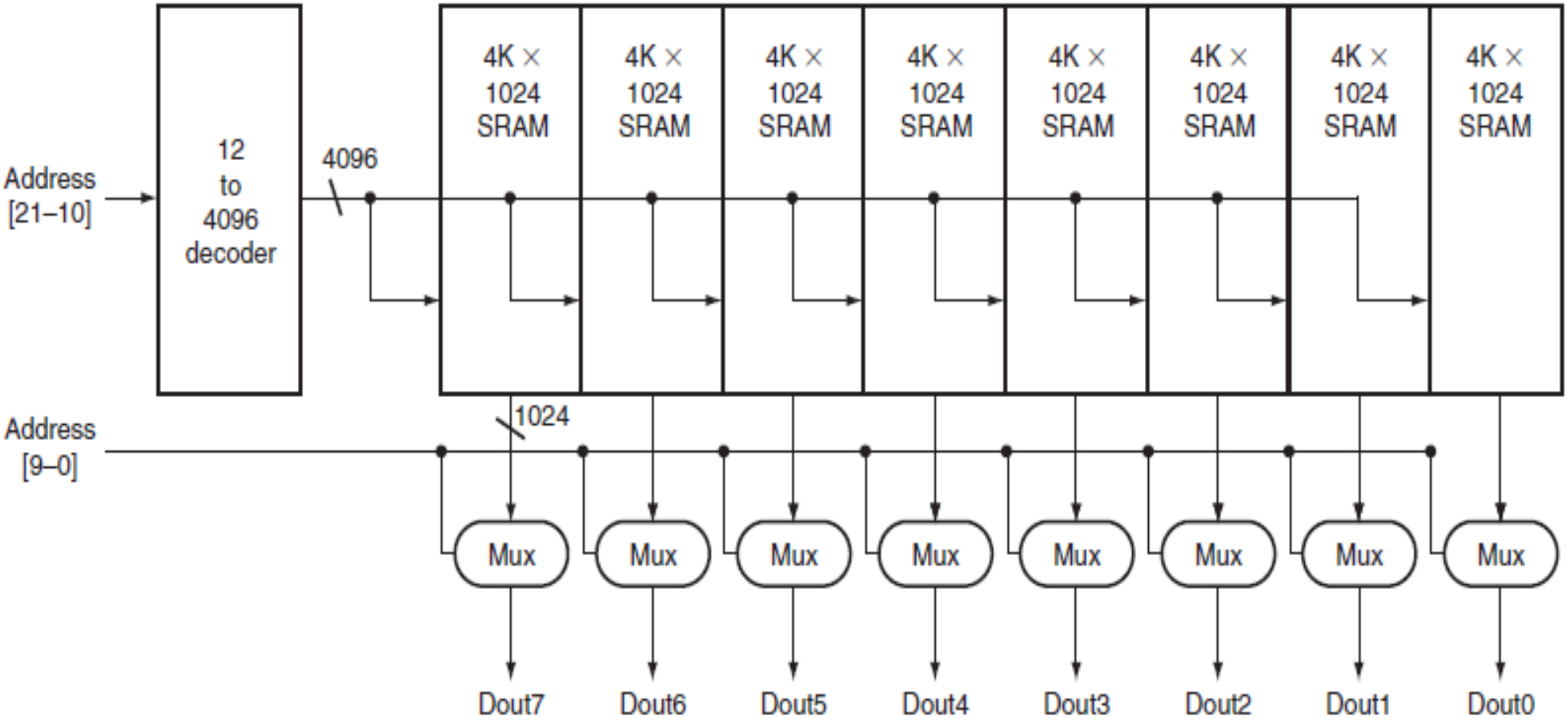


No large multiplexer at output!

SRAM **array** (read)

4Mx8

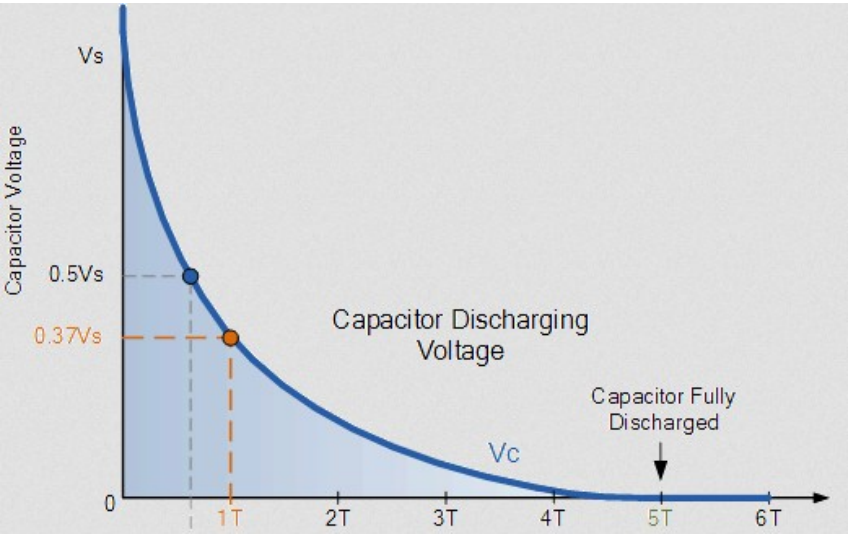
2-level decoding process
to remove need for huge decoder

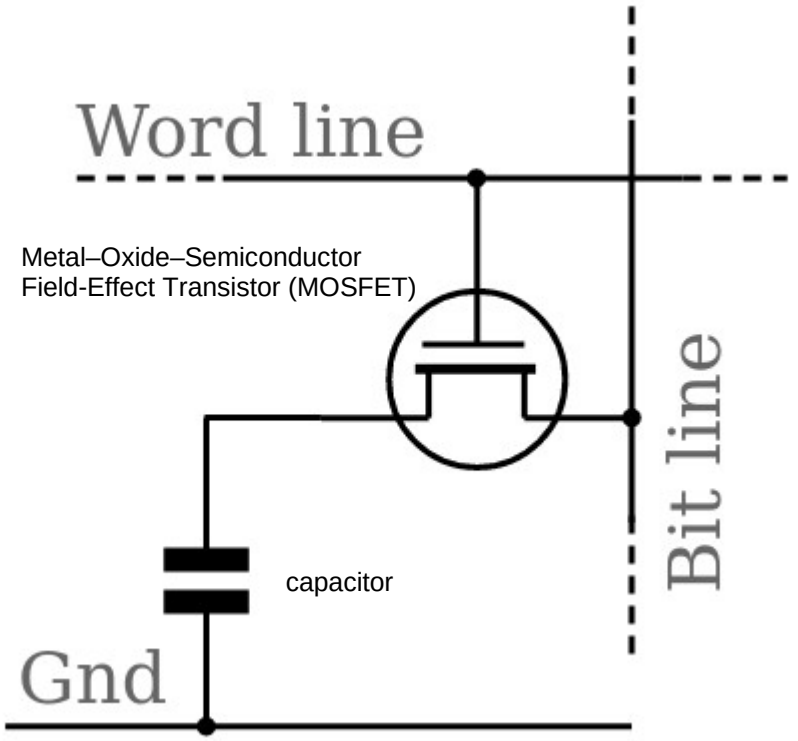


DRAM: Dynamic, use capacitor instead of flip-flop

store information in capacitor → **volatile**, needs **refresh**

only 1 transistor per bit (to access data) → denser and cheaper



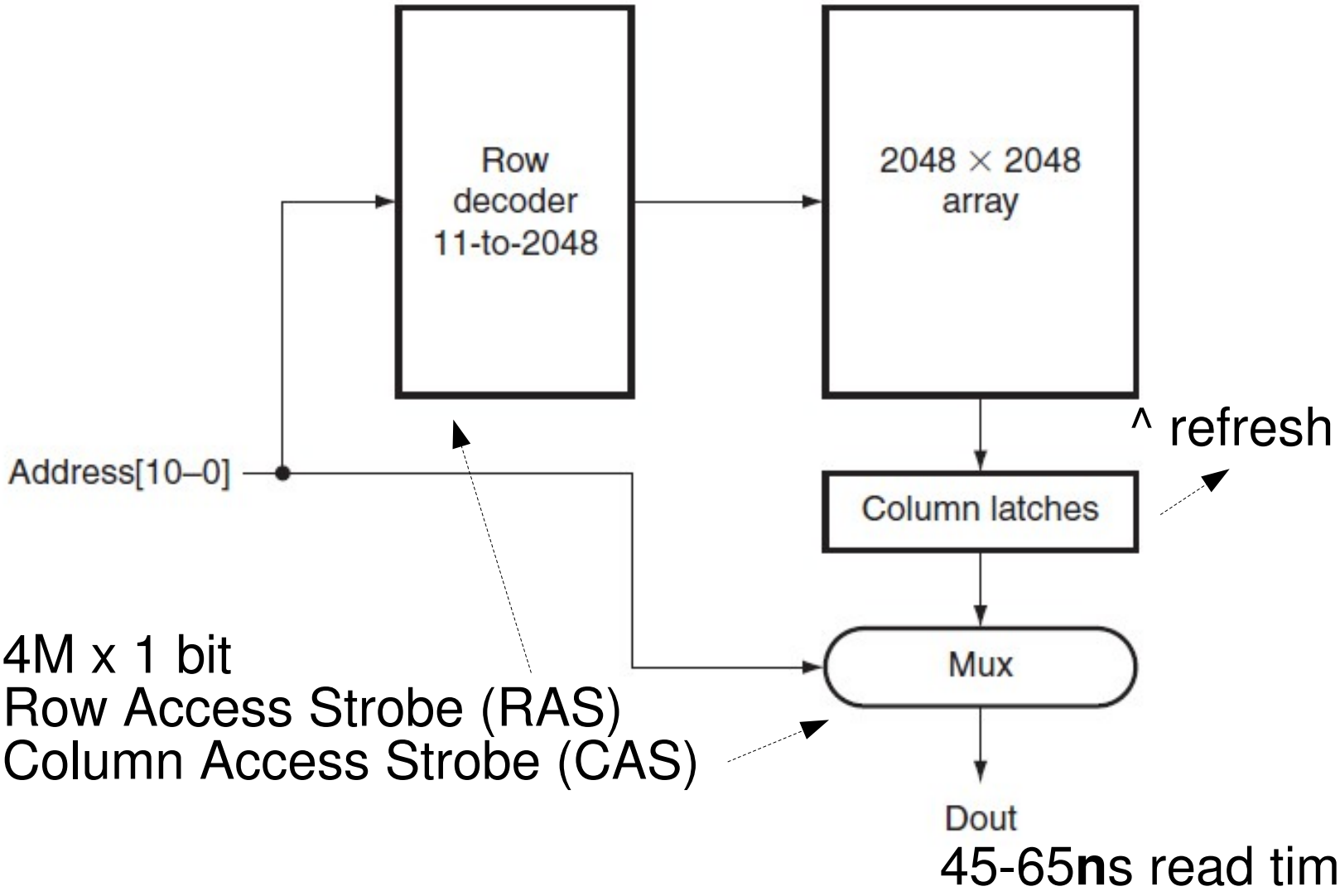


read/write/**refresh**

(rate = **m**s, 64ms row refresh)

refresh by read and write-back

DRAM array



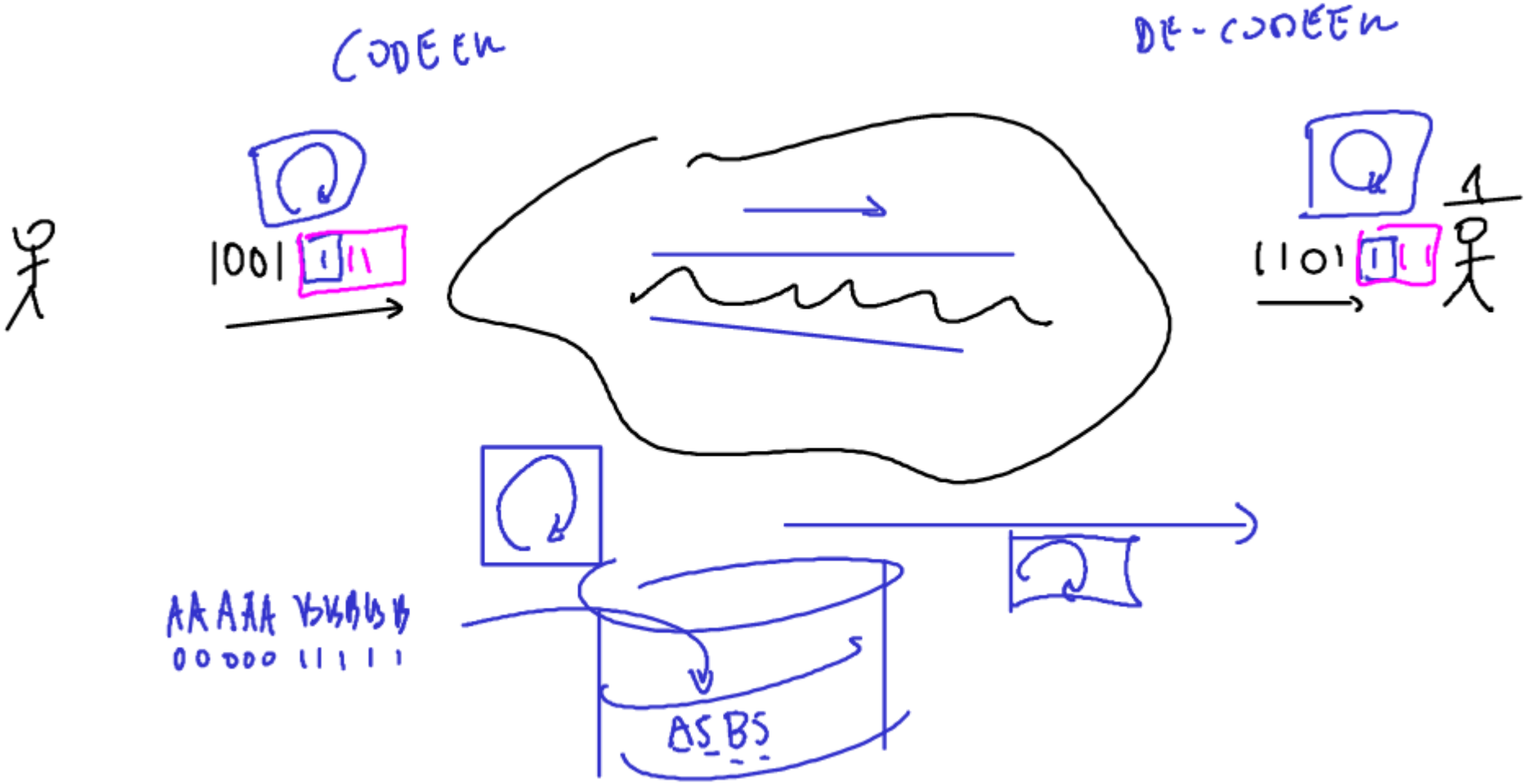
4M x 1 bit
Row Access Strobe (RAS)
Column Access Strobe (CAS)

45-65**n**s read time

Synchronous RAM (**S**SRAM and **S**DRAM):
"burst" of data from a series of sequential addresses (use **clock**, not address)
Synchronized (on rising clock edge) with datapath

Errors (in information transmission/storage)

→ en**co**ding/**dec**oding (hence the name “codec”, e.g., H.264)



Error Detection Code
(in information transmission/storage)

1 bit **parity**: **detect** 1 bit error
= distance-2 code

even/odd parity

Error Correction Code (ECC)

Data Word	Code bits	Data	Code bits
0000	000	1000	111
0001	011	1001	100
0010	101	1010	010
0011	110	1011	001
0100	110	1100	001
0101	101	1101	010
0110	011	1110	100
0111	000	1111	111

`0110 011` with 1 bit data error gives one of :

`1110 011, 0010 011, 0100 011, 0111 011`

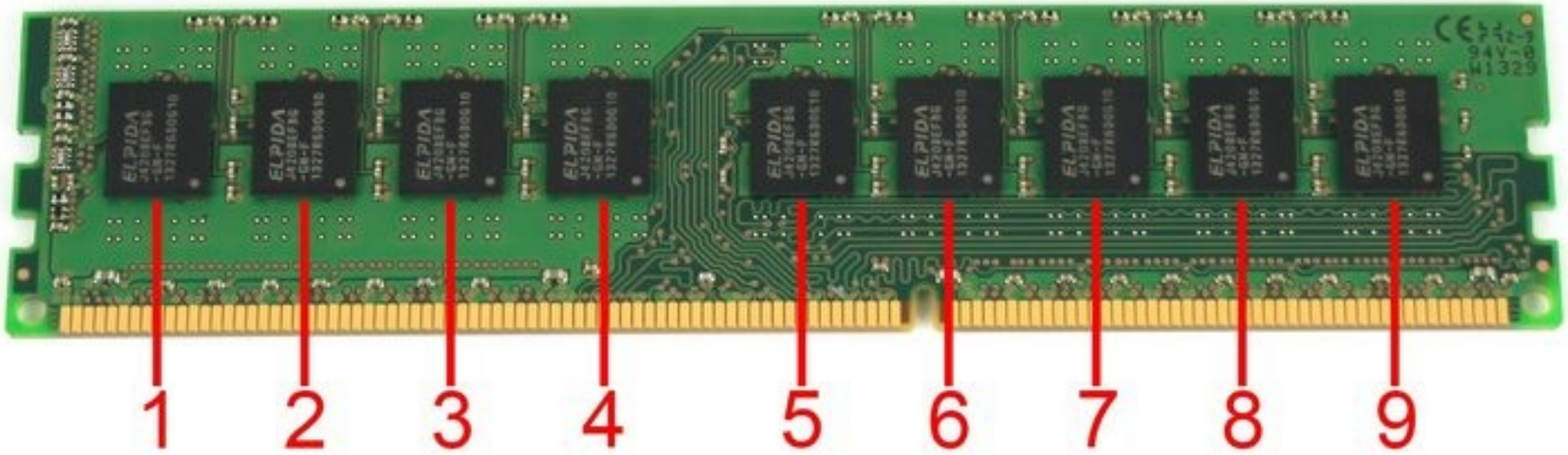
ECCode `011` appears in table for

`0110` (**edit distance** 1) ← most likely

`0001` (edit distance 3, 2, 2, 2 respectively)

ECC RAM

aka “server class” memory



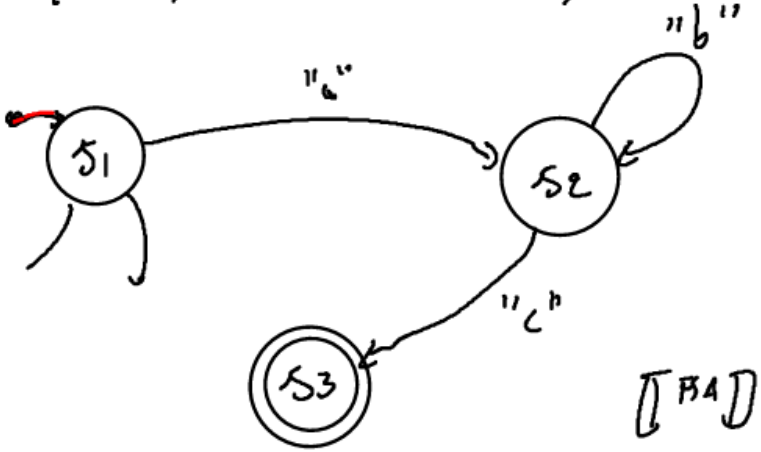
Non-ECC RAM

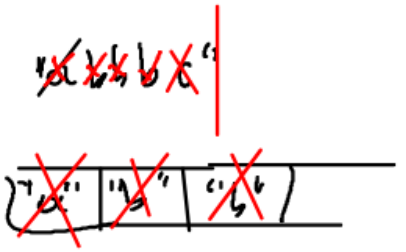
Finite **State** Machines (DFA, FSA, FSM)

DETERMINISTIC

DFA

$FSA = \langle S, s_i \in S, T \subseteq S \times S \times L, Acc \}$





"a b* c"

↑ REGEXPR

$[\![FA]\!] = \{$ "abc", "abbbc", ... $\}$

$S = \{ s_1, s_2, s_3 \}$

$s_1 = s_1$

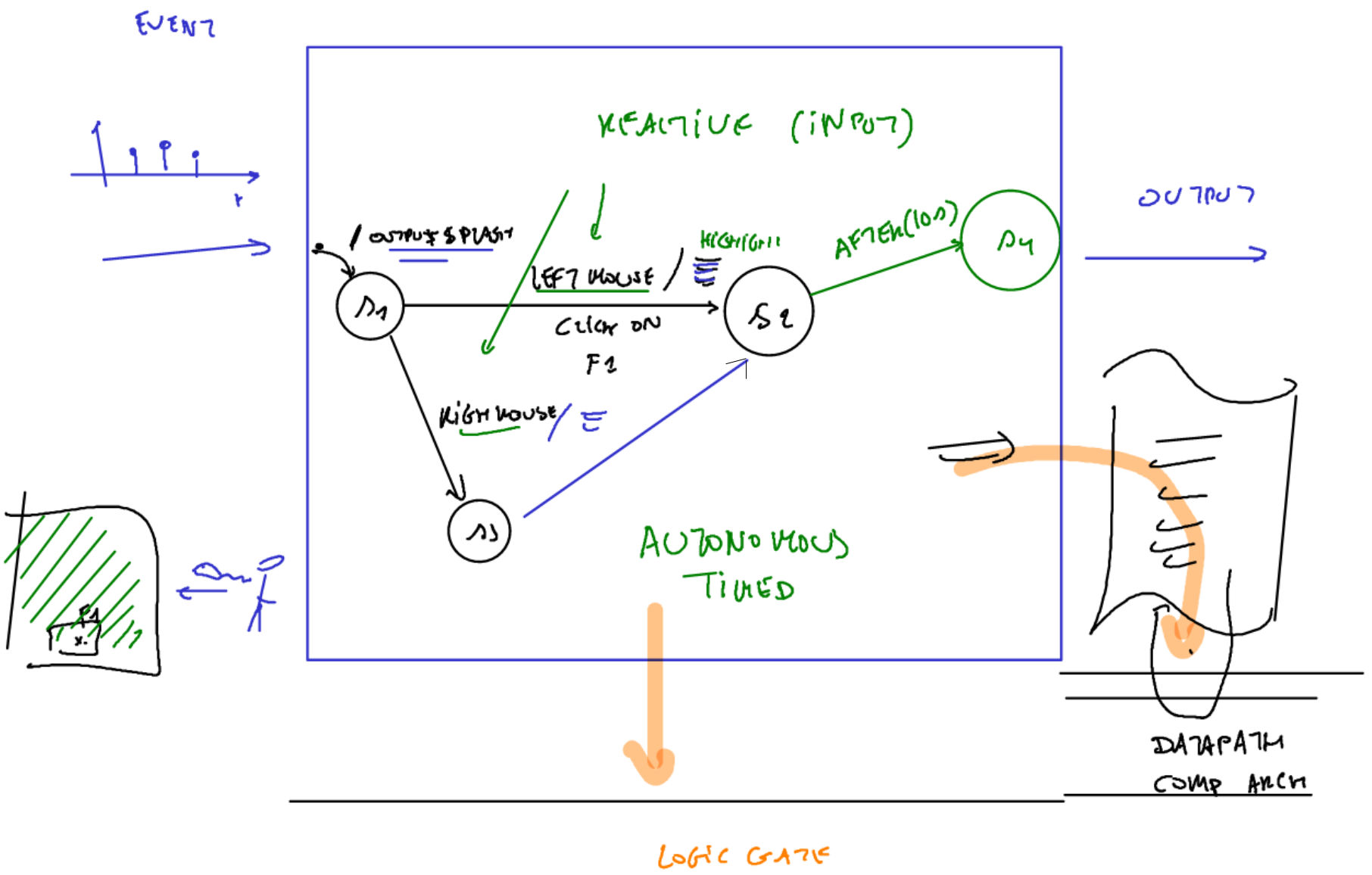
$T = \{ ((s_1, s_2), "a"), ((s_2, s_2), "b"), ((s_2, s_3), "c") \}$

$L = \{ "a", "b", "c" \}$

$Acc = \{s_3\}$

for “language” recognition

Finite **State** Machines (Discrete Event – DE)



for system (behaviour) specification (and synthesis)

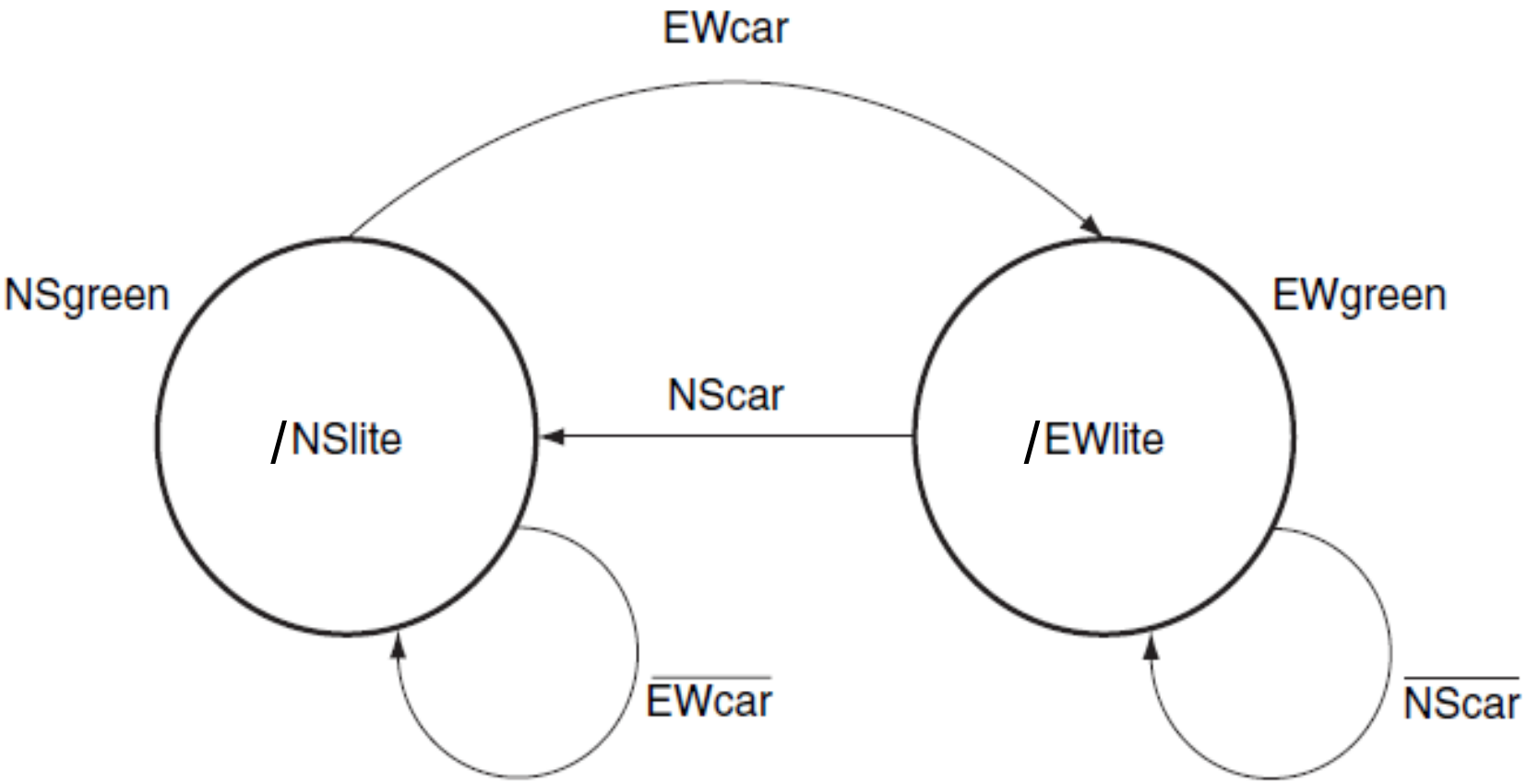
Traffic Light (Discrete-Time – DT)

	Inputs		
	NScar	EWcar	Next state
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

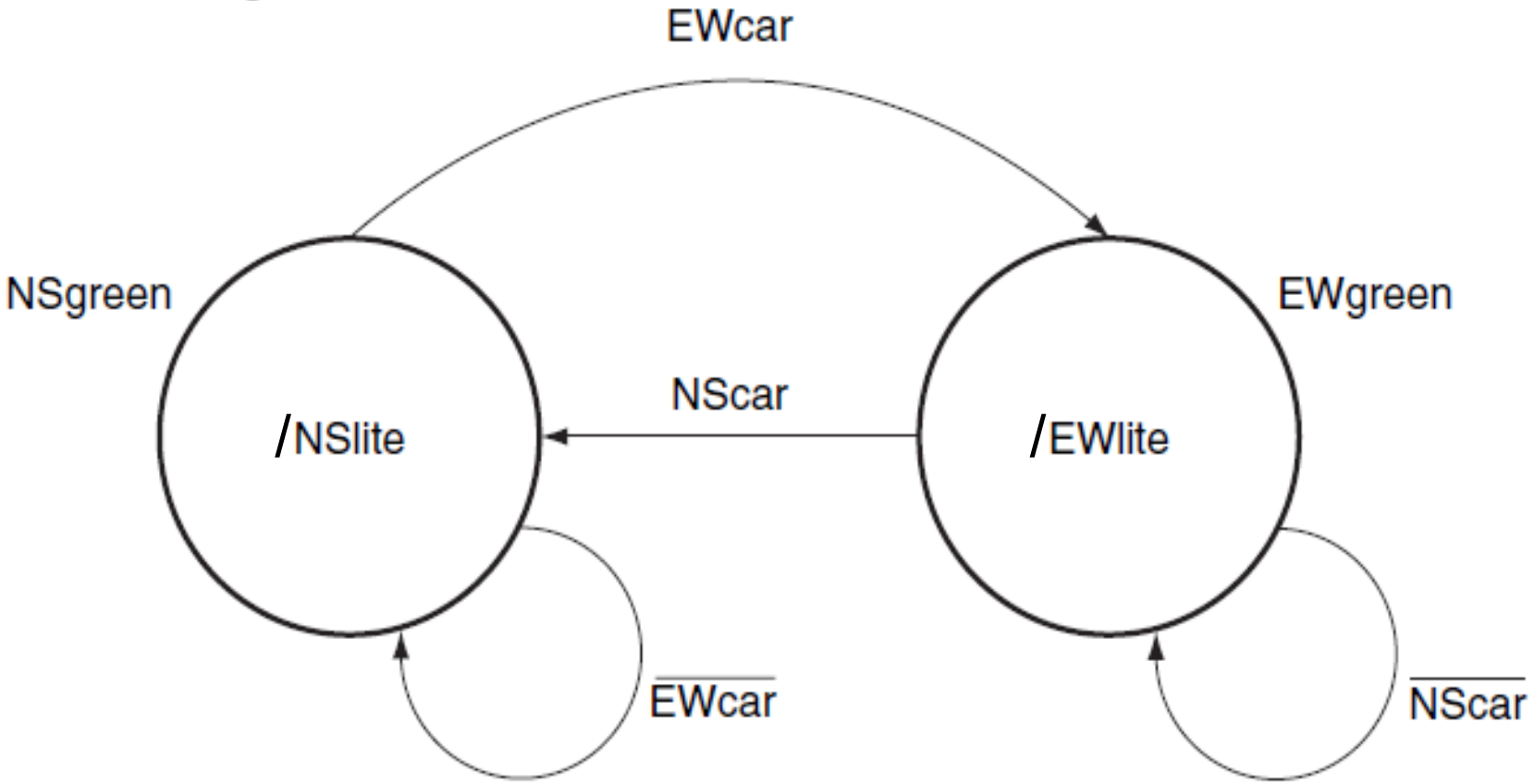
fairness (in case of conflict)

	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

Traffic Light



Traffic Light

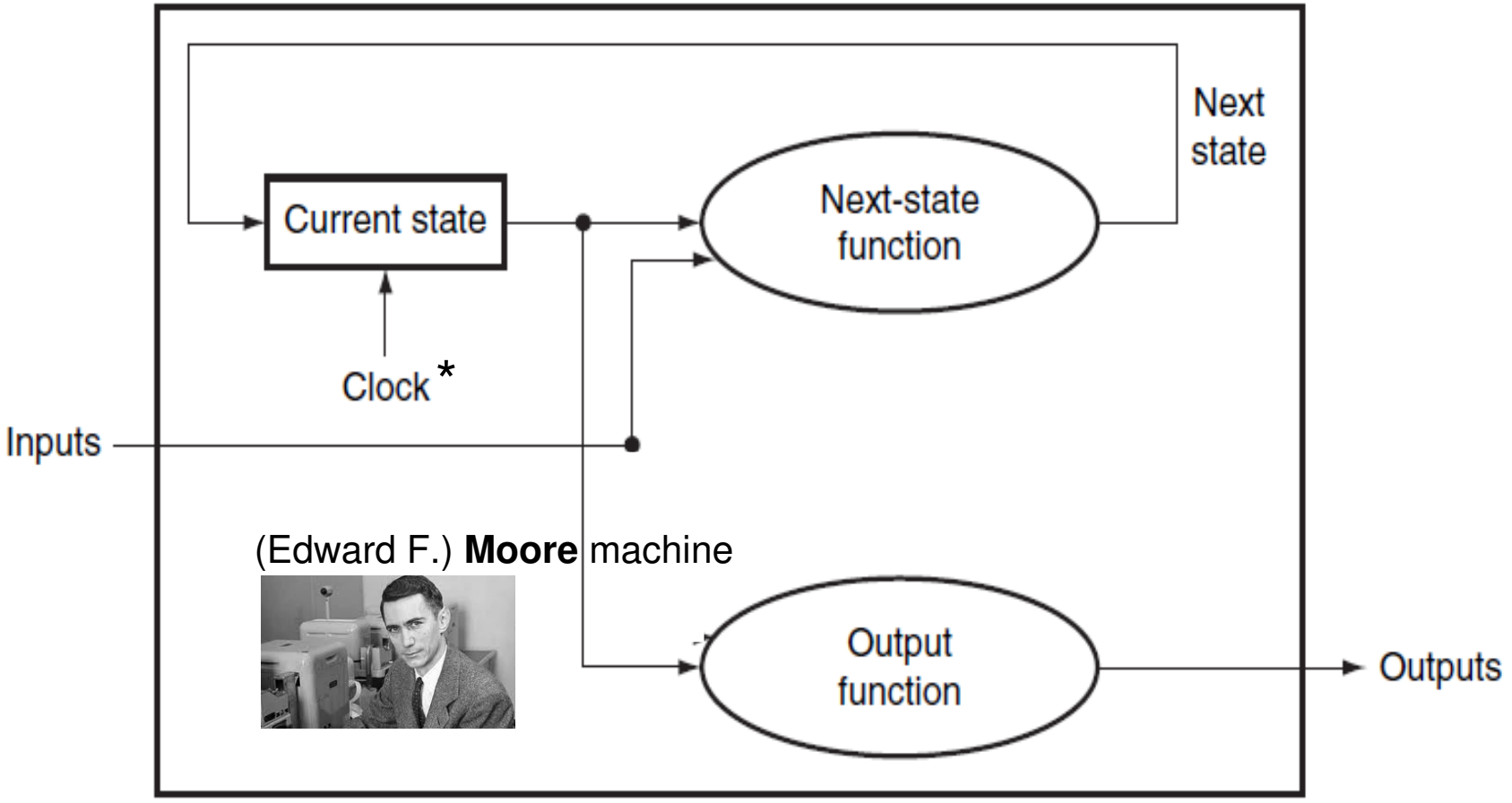


encode `CurrentState (NSgreen or Ewgreen)` in 1 bit

```
NextState = (~CurrentState . EWcar) + (CurrentState . ~NScar)
NSlite    = ~CurrentState
EWlite    =  CurrentState
```

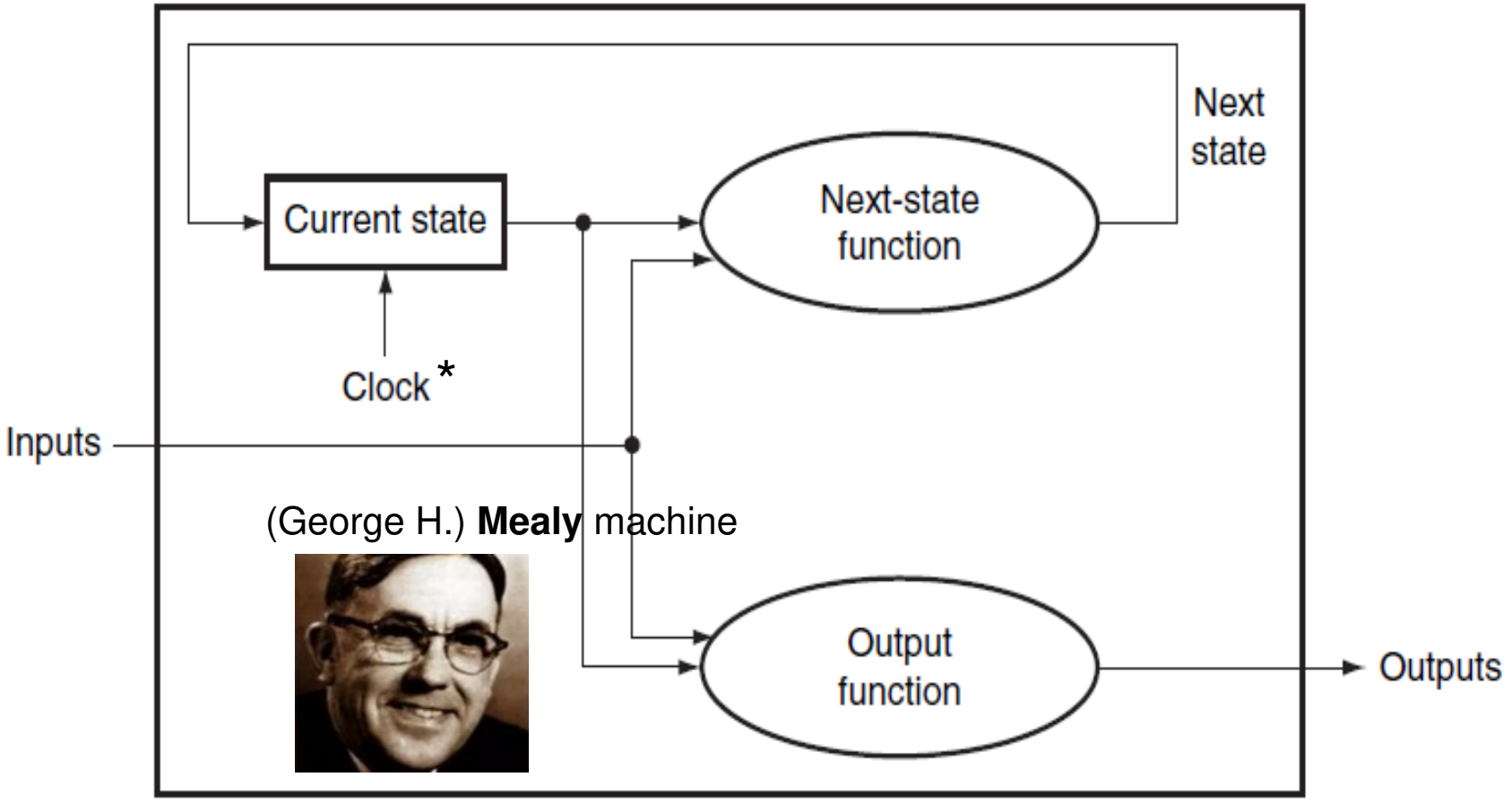
... evaluated every clock cycle … add `after(delay)` ... ?

Finite **State** Machines



* Discrete-Time (DT) realization (vs. Discrete-Event (DE))

Finite **State** Machines



* Discrete-Time (DT) realization (vs. Discrete-Event (DE))

Traffic Light