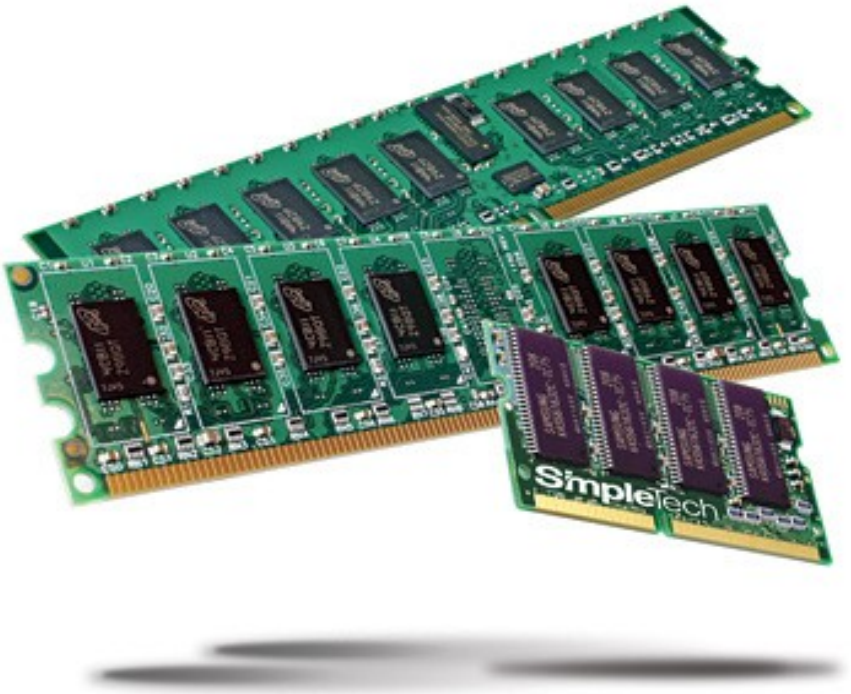
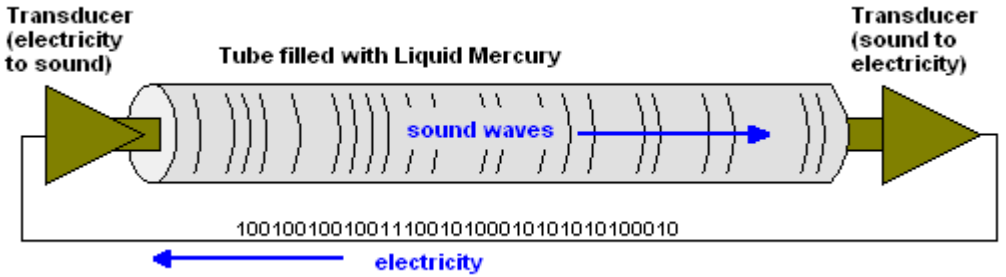
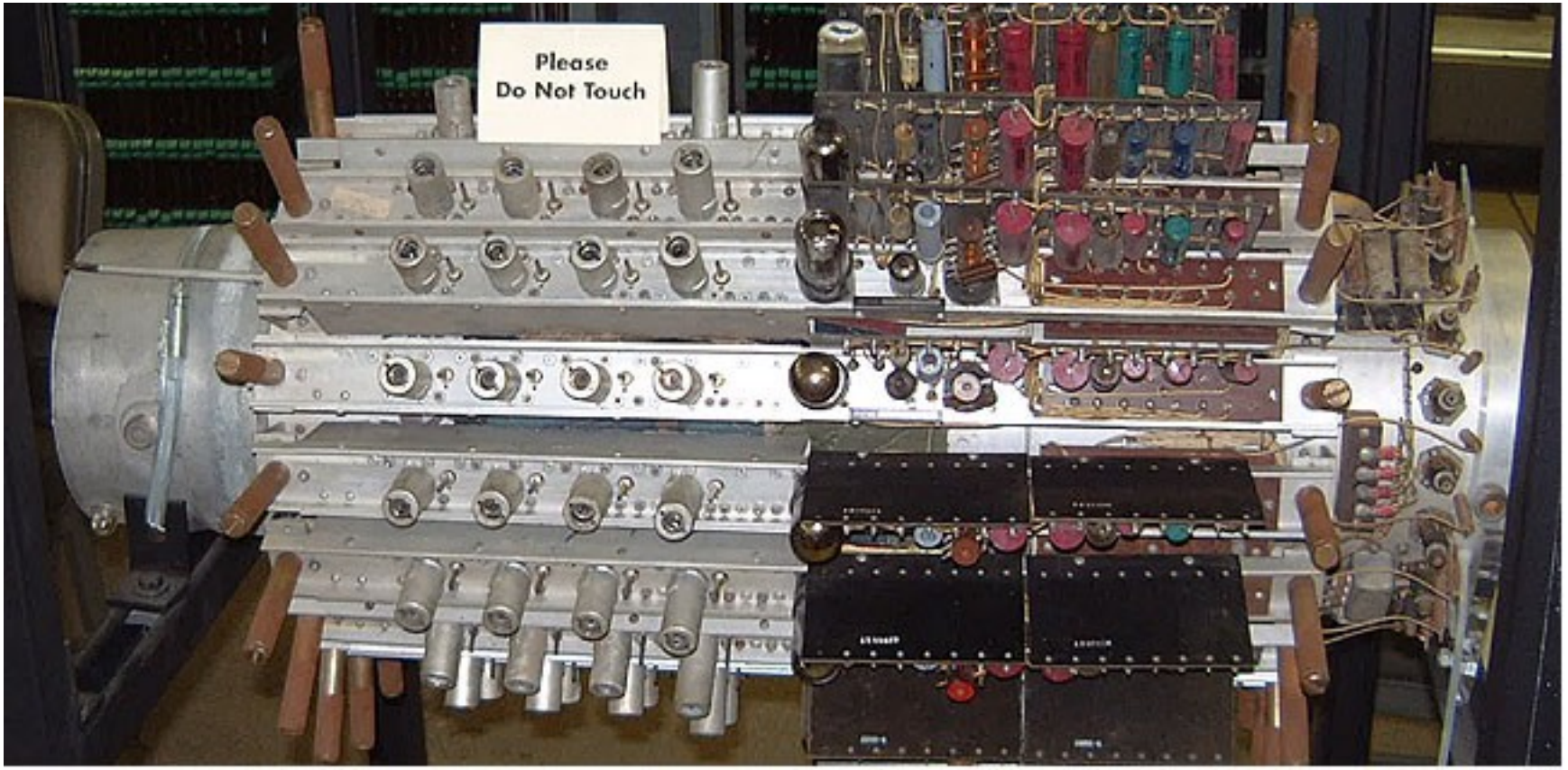
Logic Design:
Implementing **Memory**

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

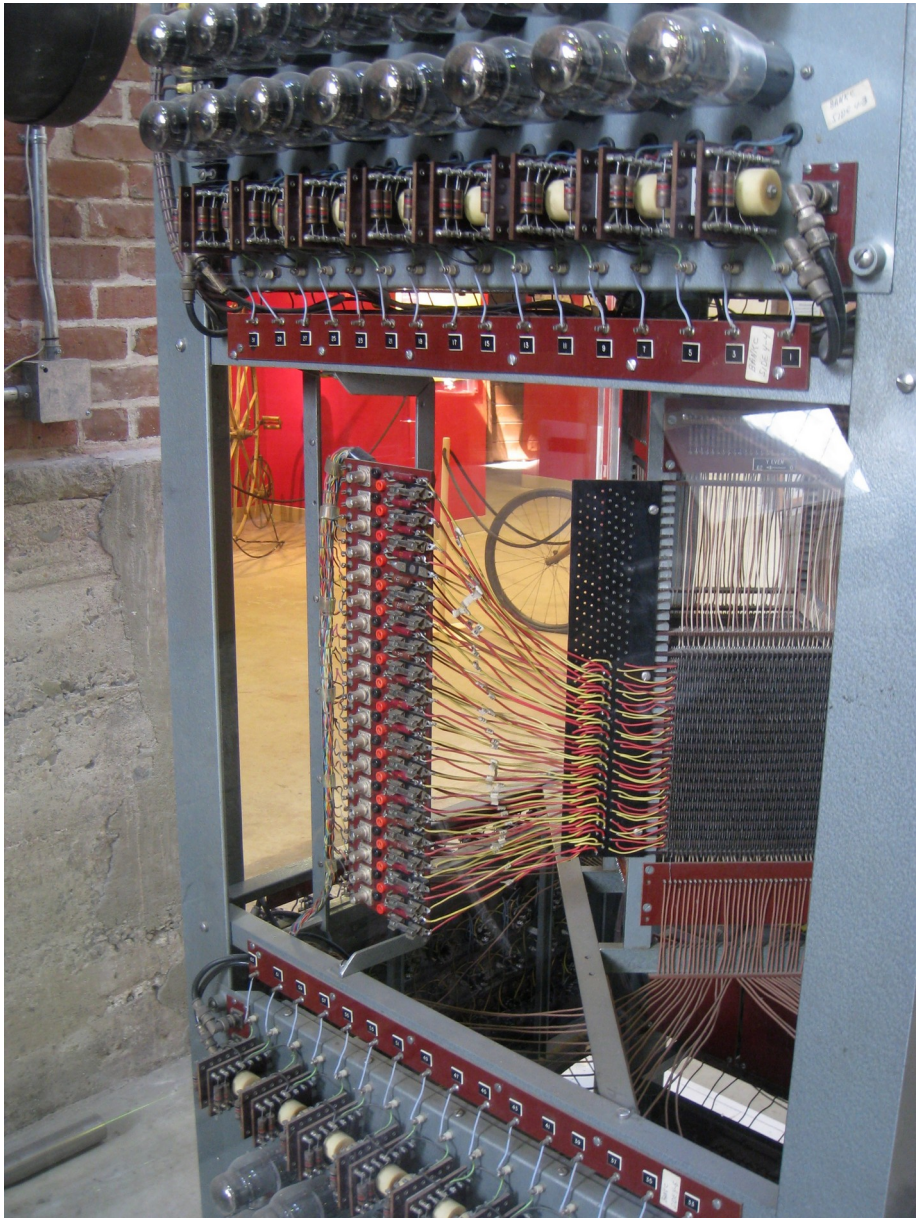


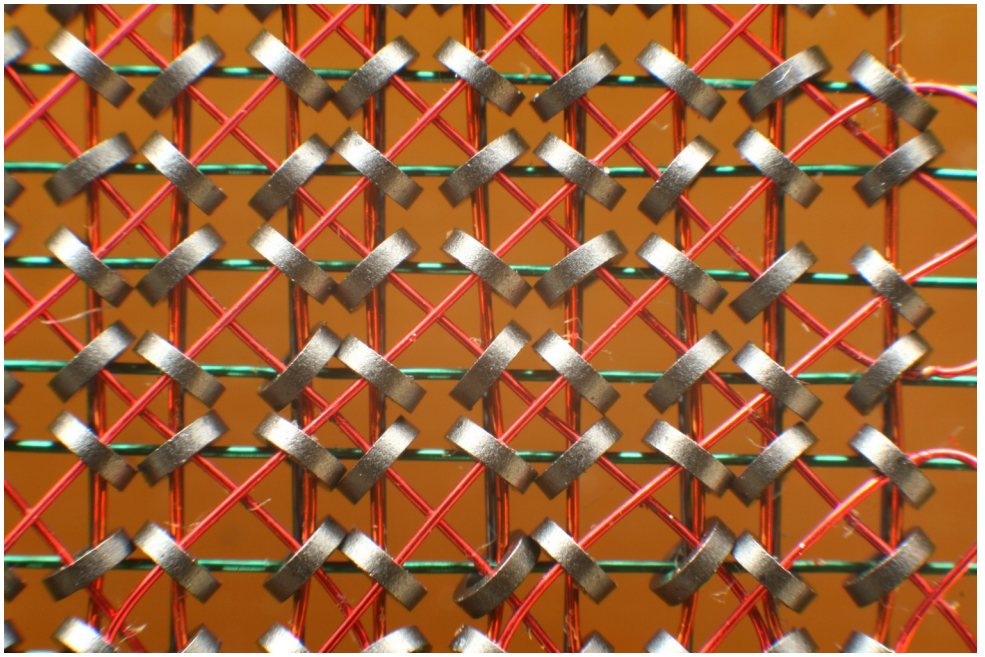
Delay Line memory



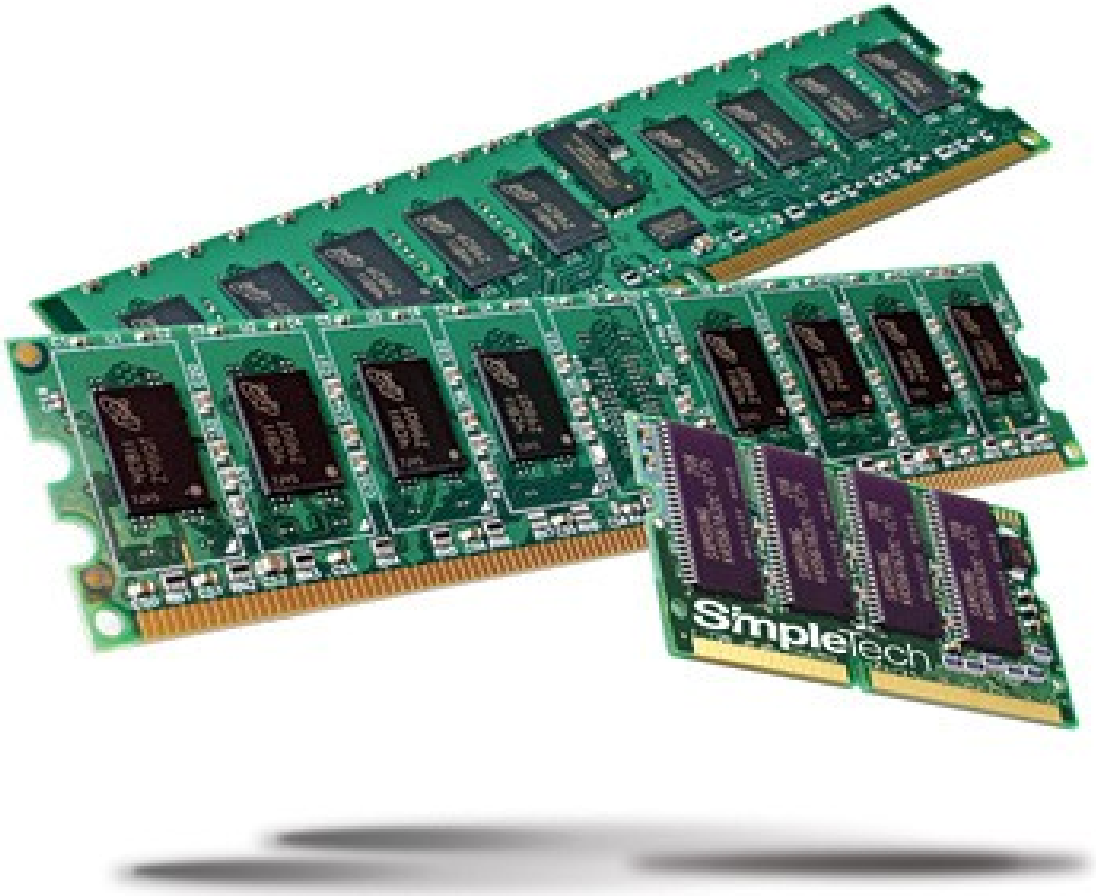


Magnetic Core memory

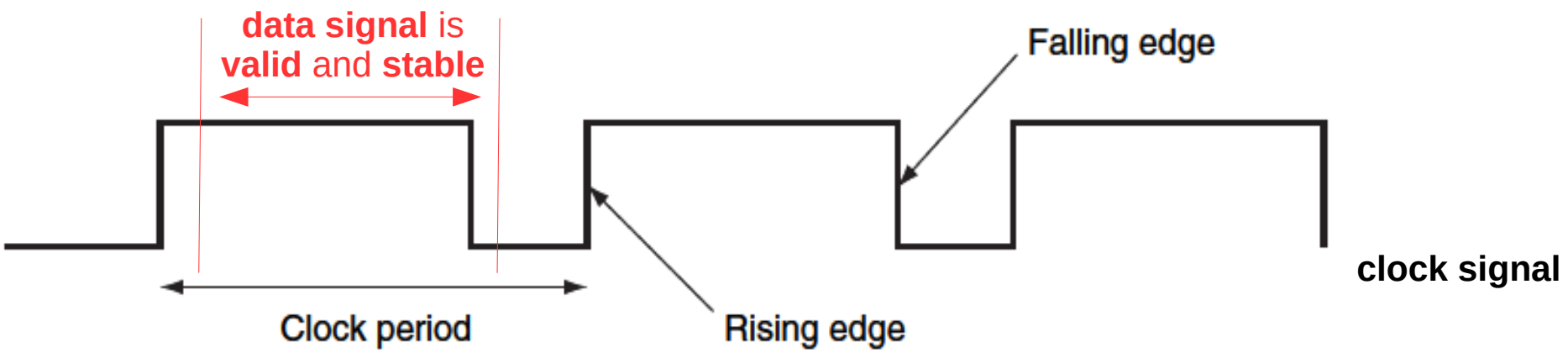




Modern memory bank



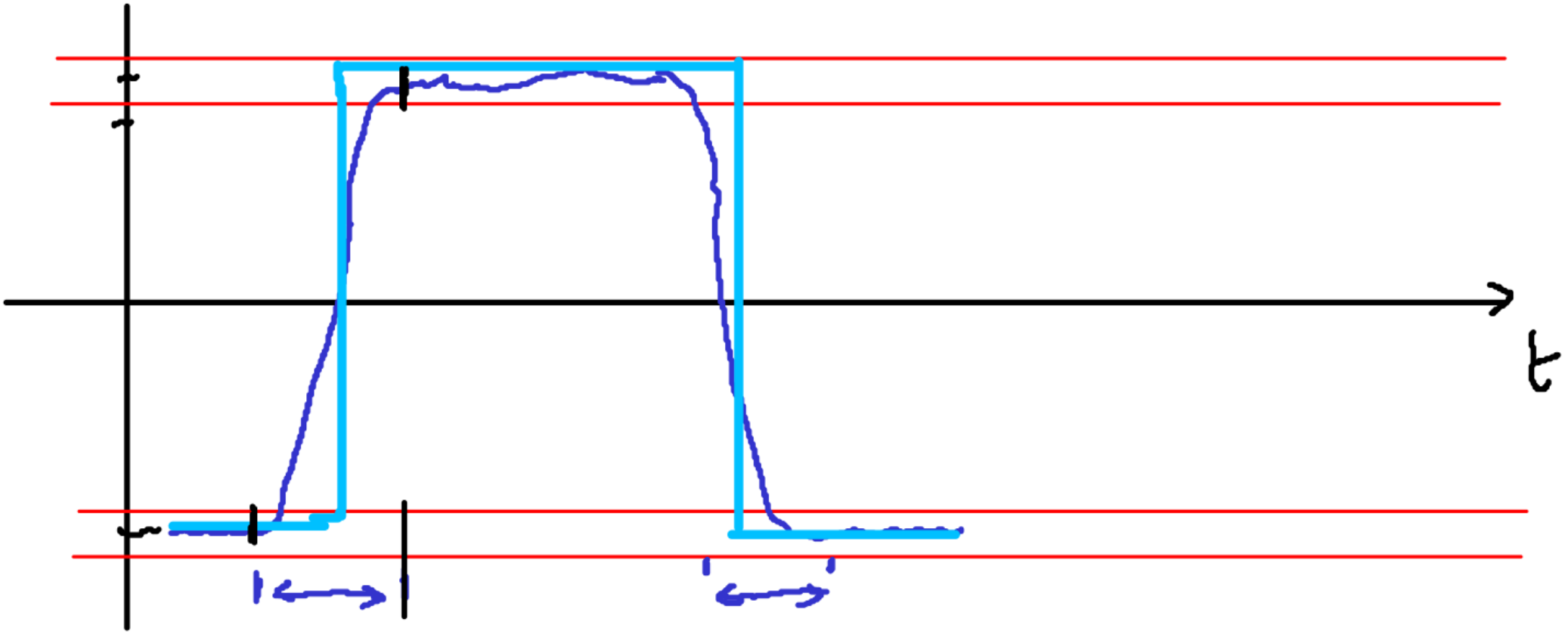
Clock Signal (**memory** retains **data** over **time**)



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)



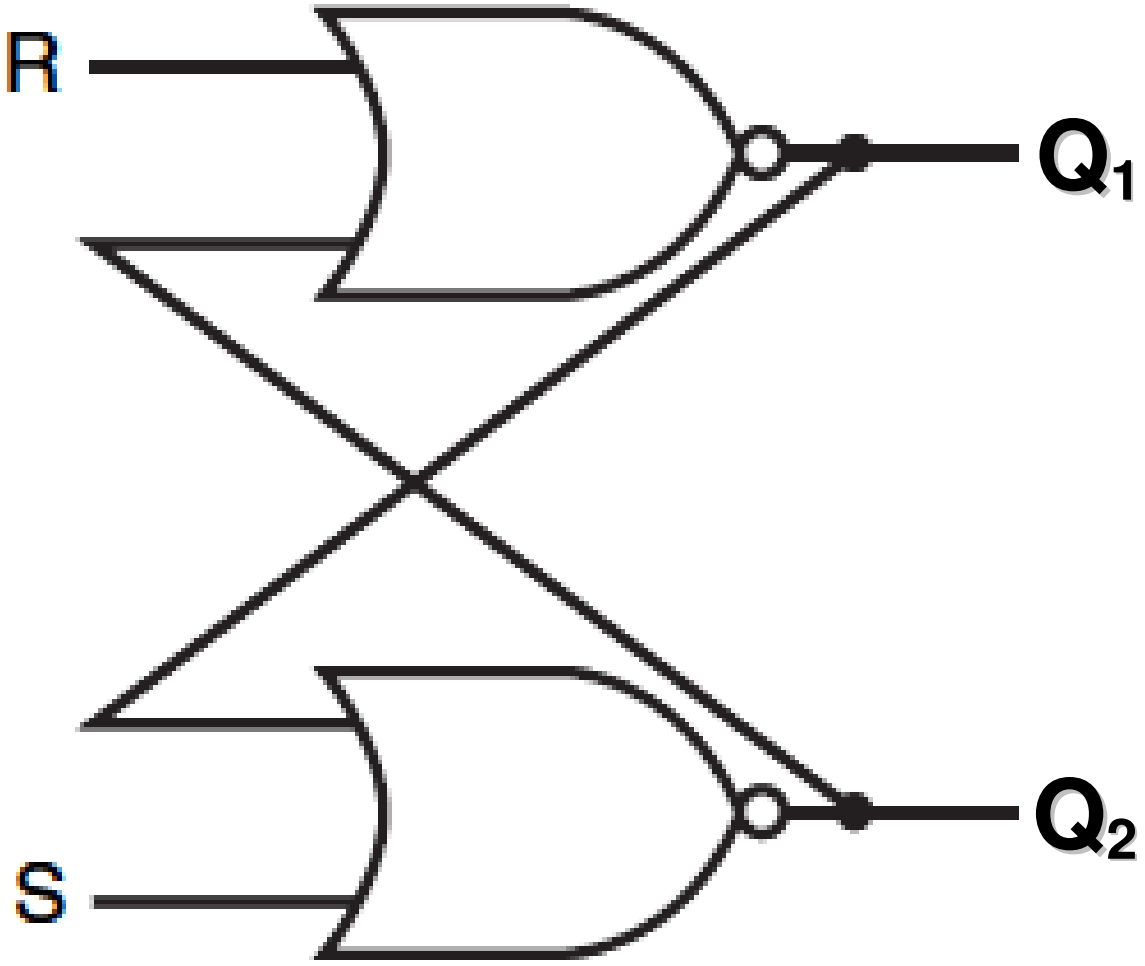
State Elements: “memory”

State Element → Combinational Logic → State Element

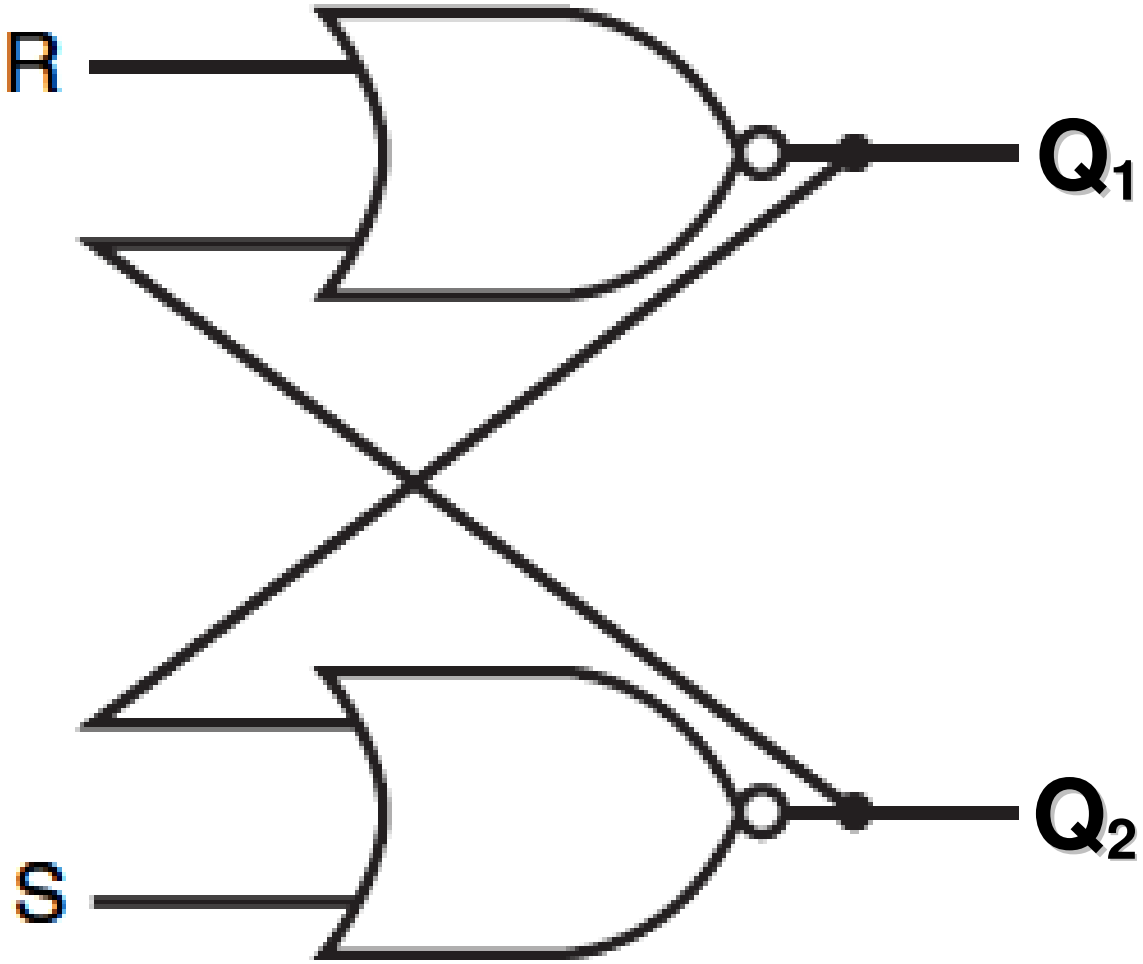
(function/computation)


Latch: store value

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

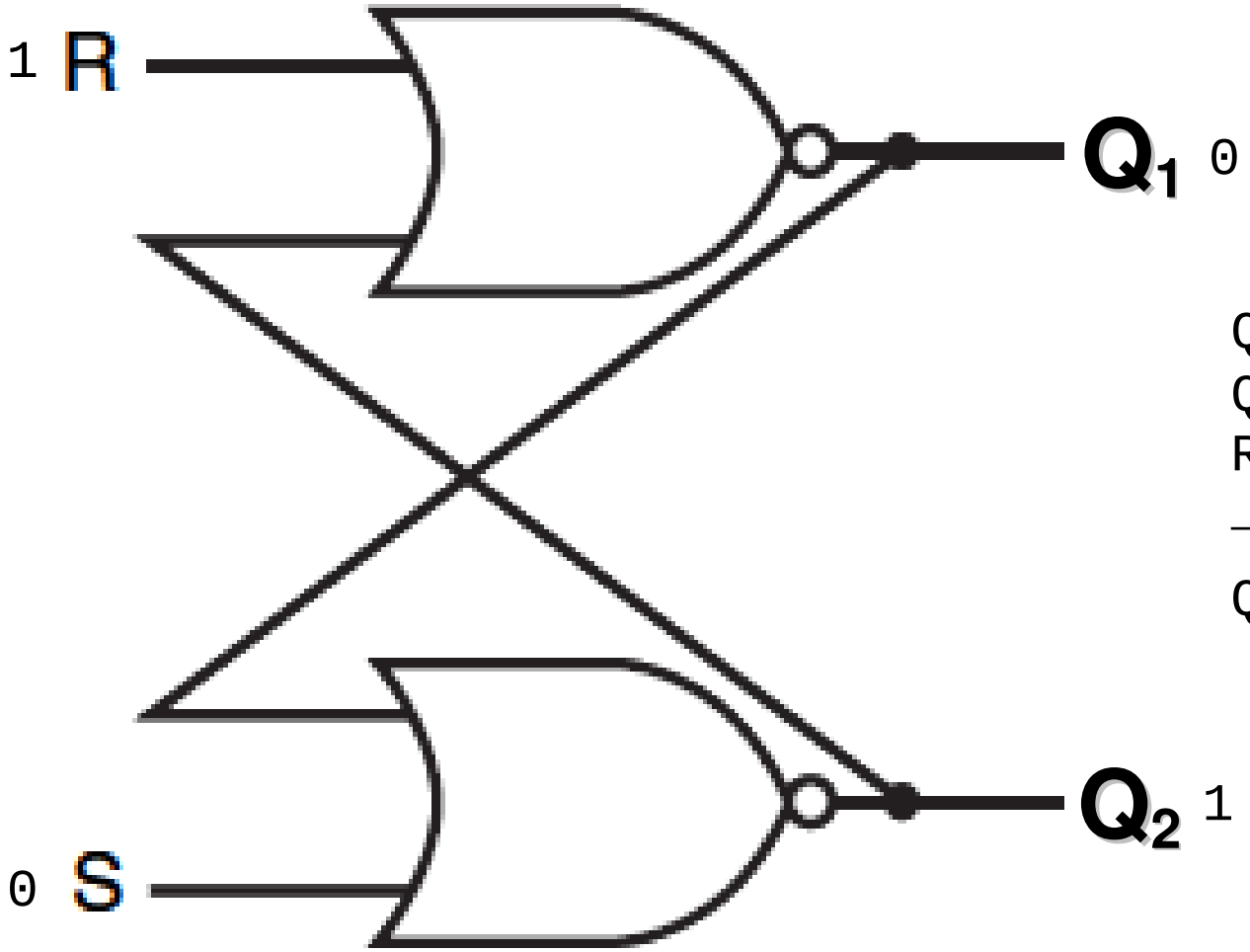


S-R Latch (unclocked): store and remember value



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

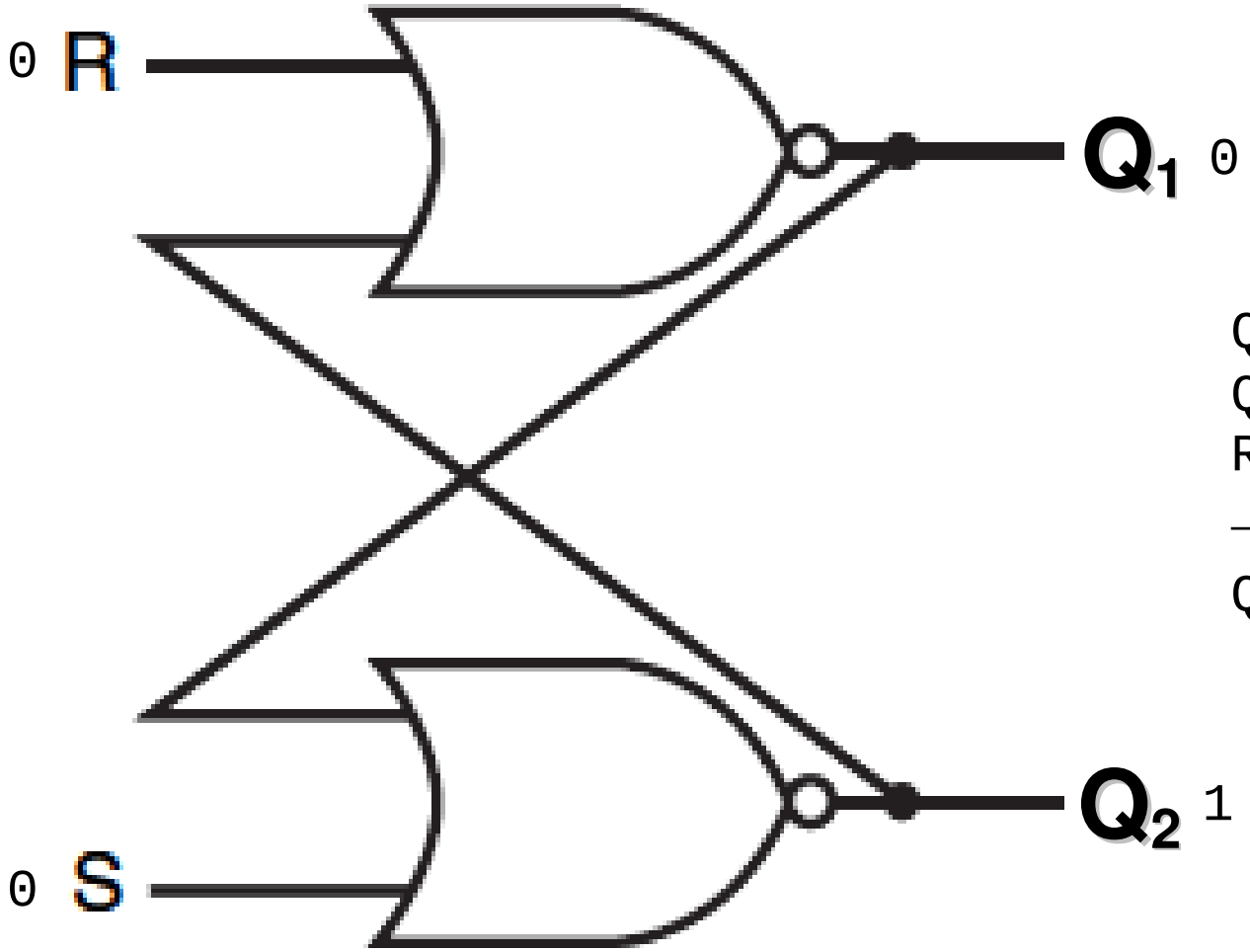


Q_1 = not R and (S or Q_1)
Q_2 = not (S or Q_1)
R = 1, S = 0

→

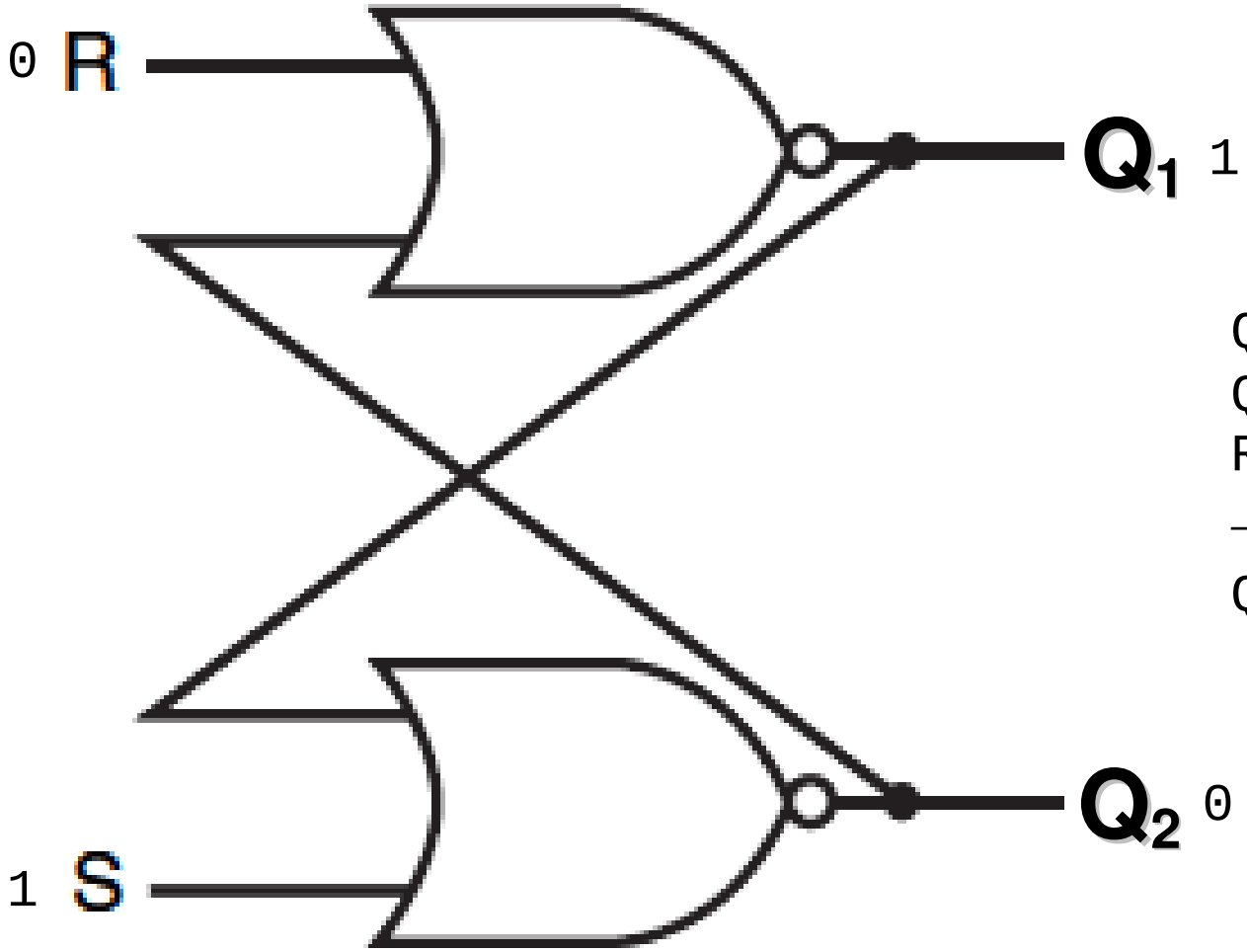
Q_1 = 0, Q_2 = not Q_1 = 1

S-R Latch remember output value



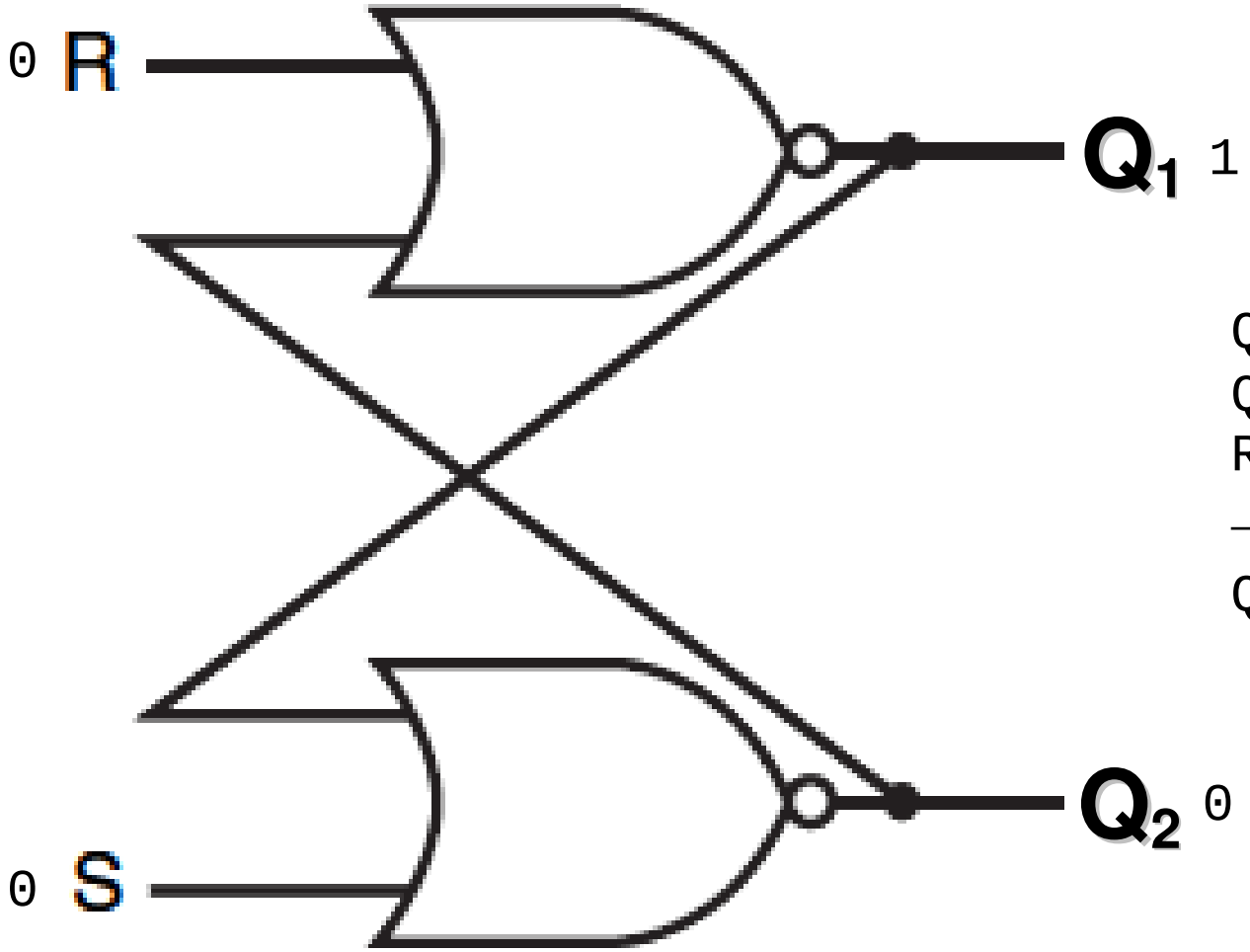
Q_1 = not R and (S or Q_1)
Q_2 = not (S or Q_1)
R = 0, S = 0
→
Q_1 = Q_1, Q_2 = not Q_1

S-R Latch set



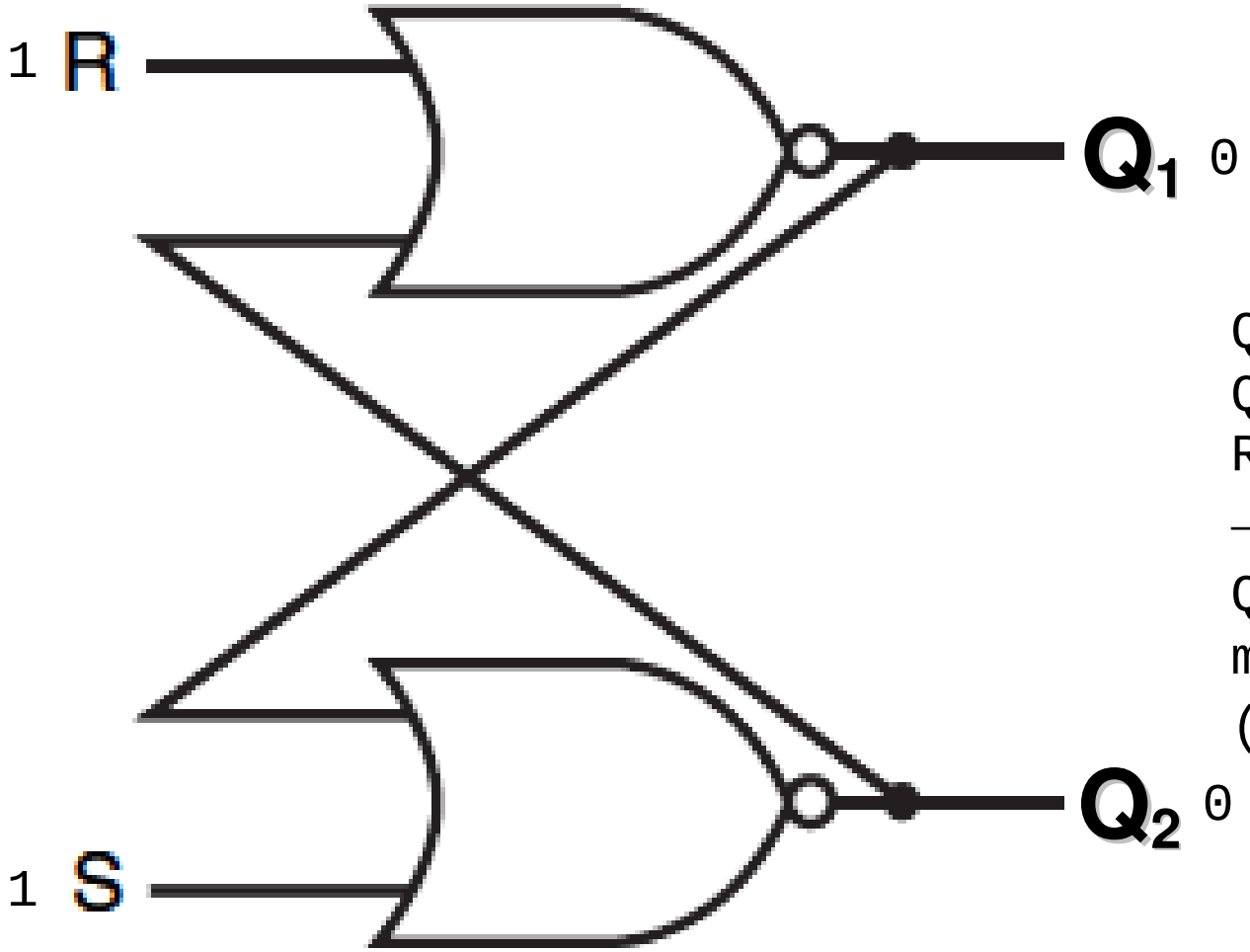
Q_1 = not R and (S or Q_1)
Q_2 = not (S or Q_1)
R = 0, S = 1
→
Q_1 = 1, Q_2 = 0 = not Q_1

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

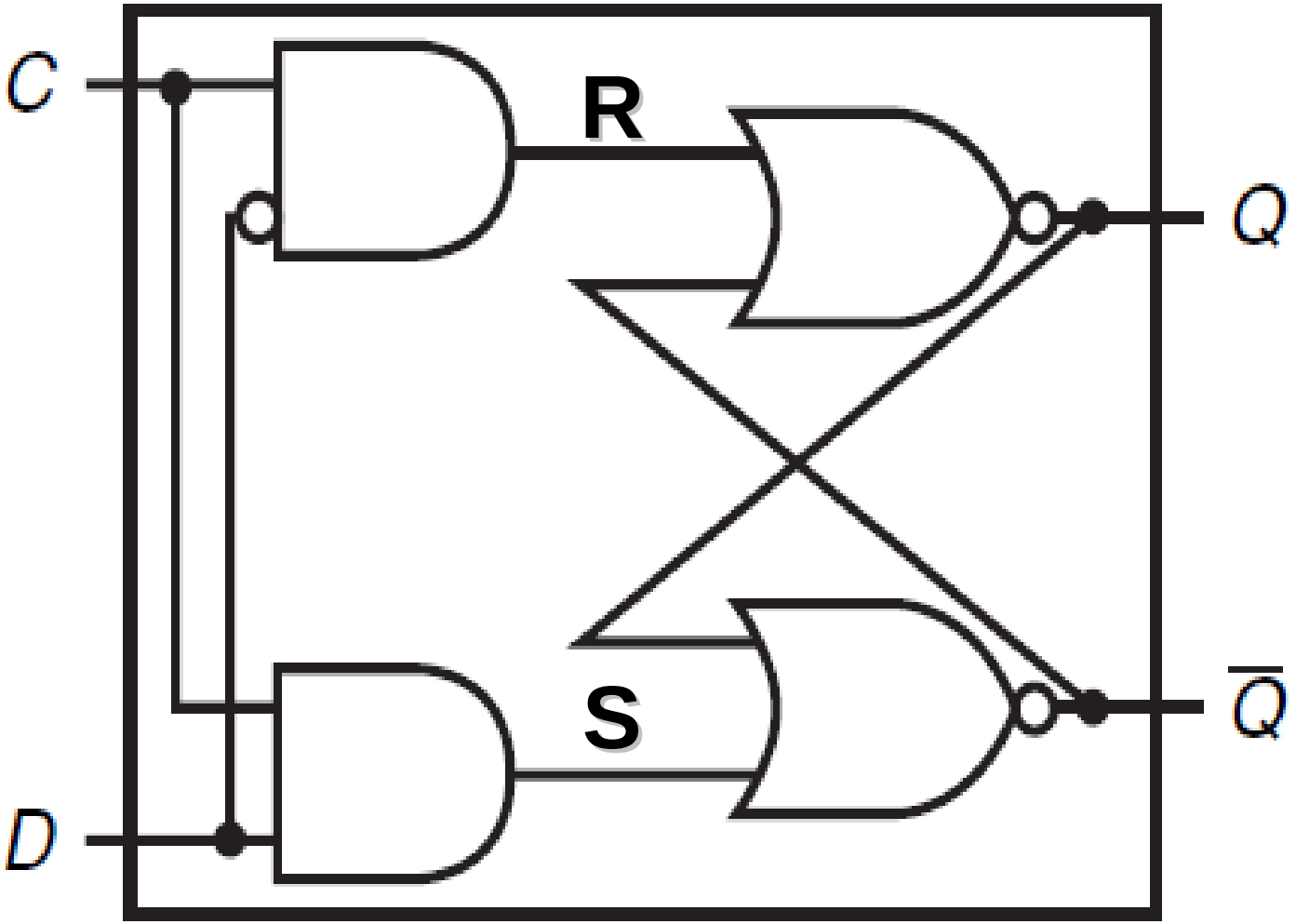


Q_1 = not R and (S or Q_1)
Q_2 = not (S or Q_1)
R = 1, S = 1
→
Q_1 = 0, Q_2 = 0
may become meta-stable
(**analog** reality: delays)

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

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

D latch (“transparent”)



C: “Clock”
D: “Data”

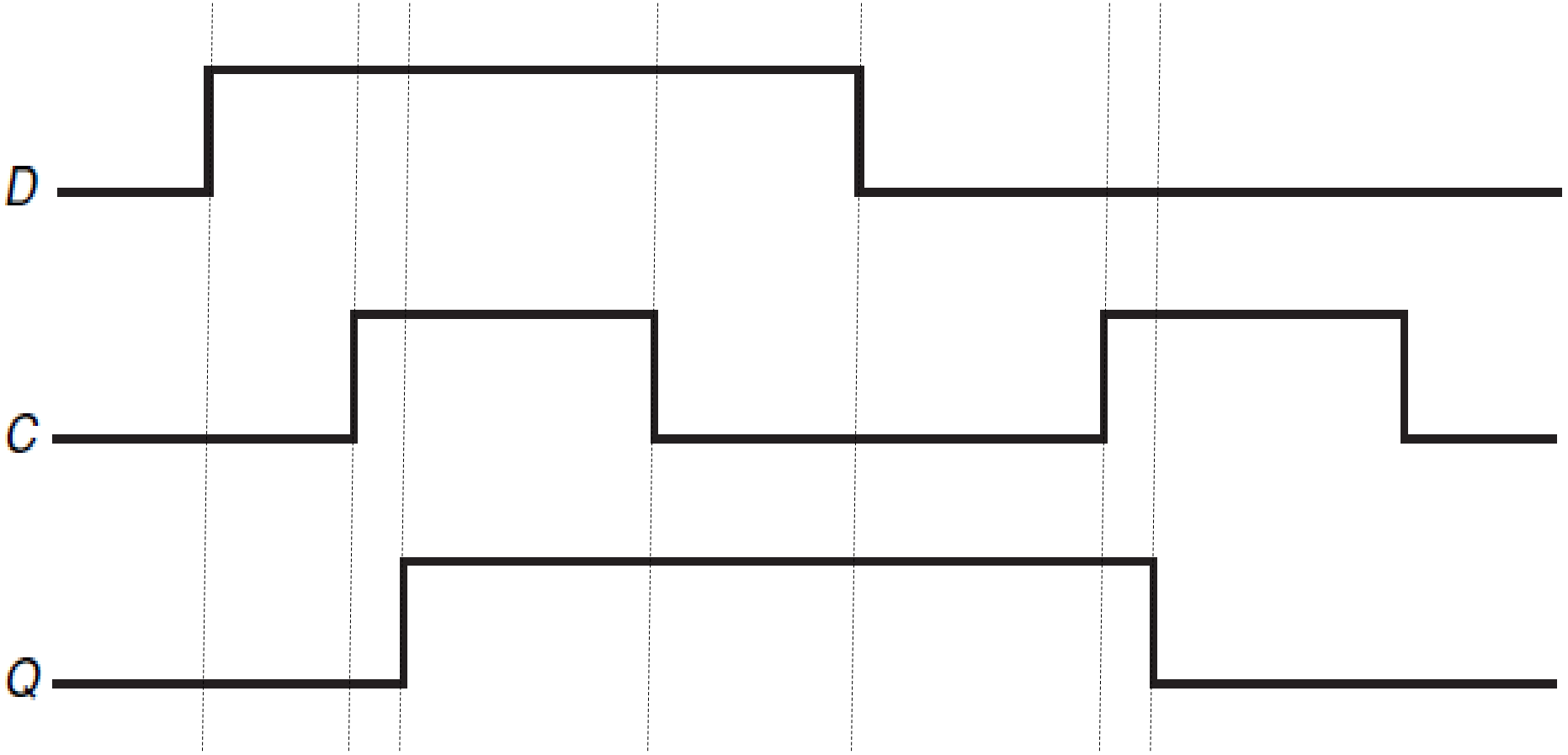
R = C and not D
S = C and D
→

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

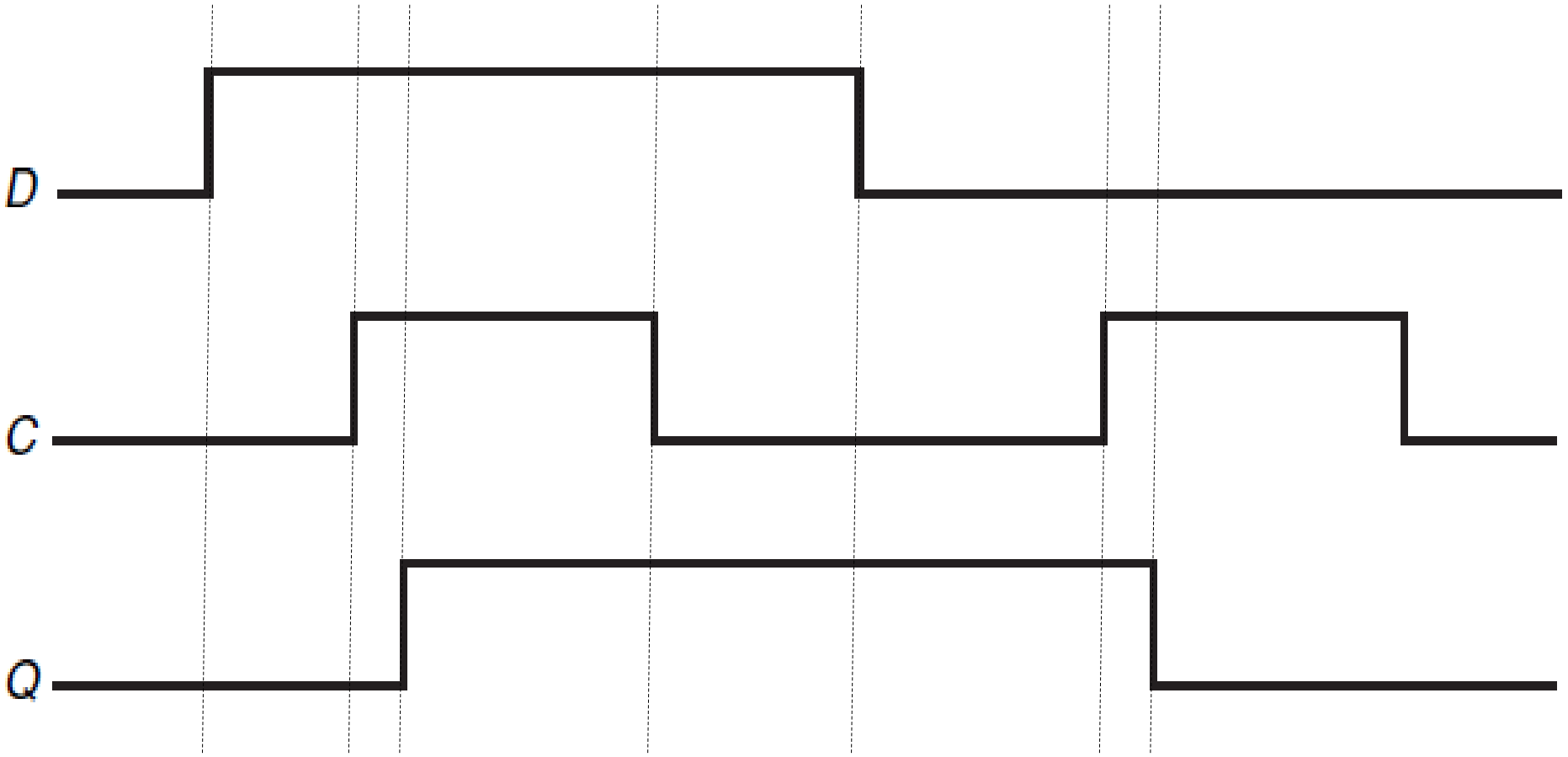
when C = 0: “closed”
R = 0
S = 0
→ Q is not changed

R = S = 1
not possible

D latch operation

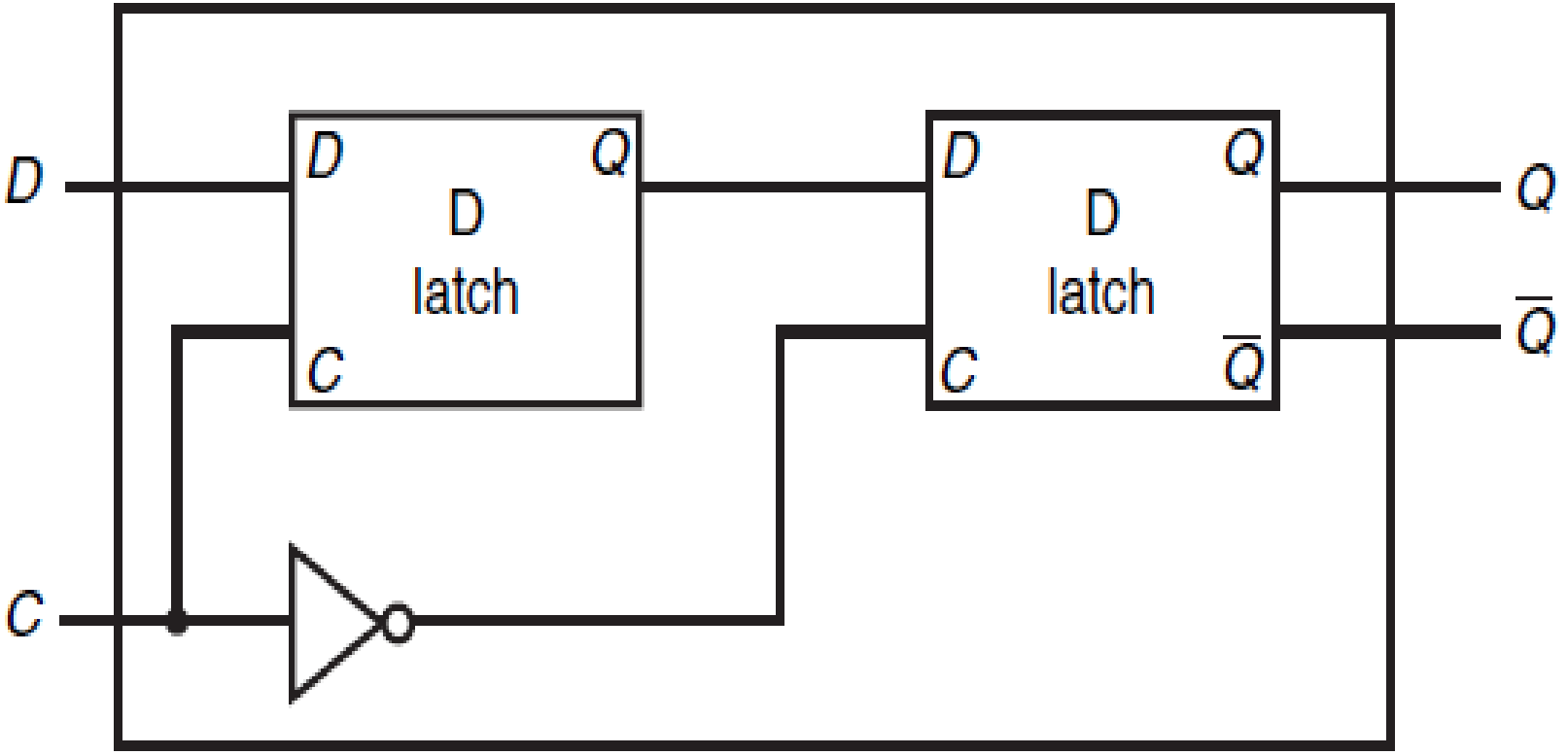


D latch operation

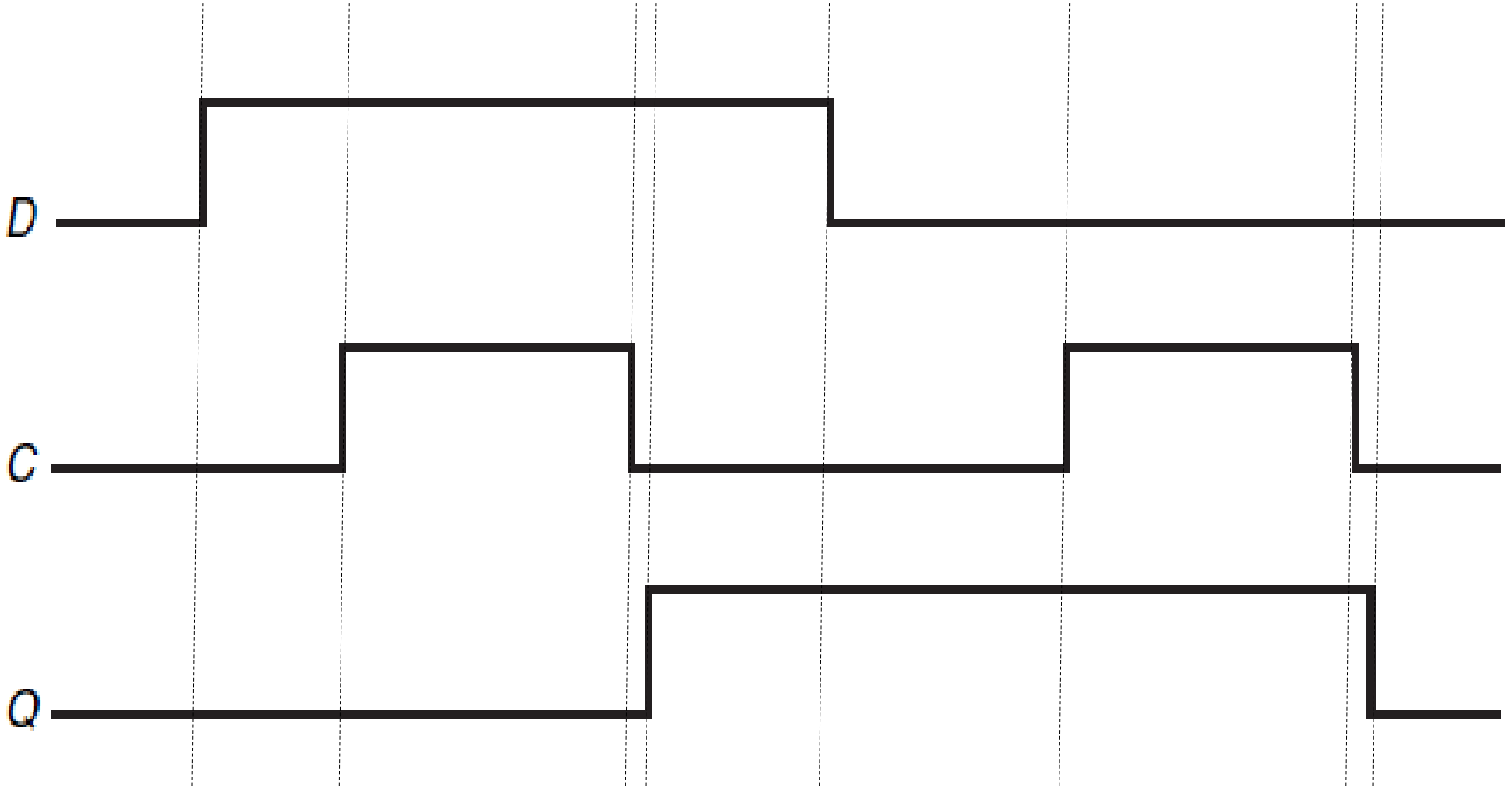


What if D changes during C asserted?

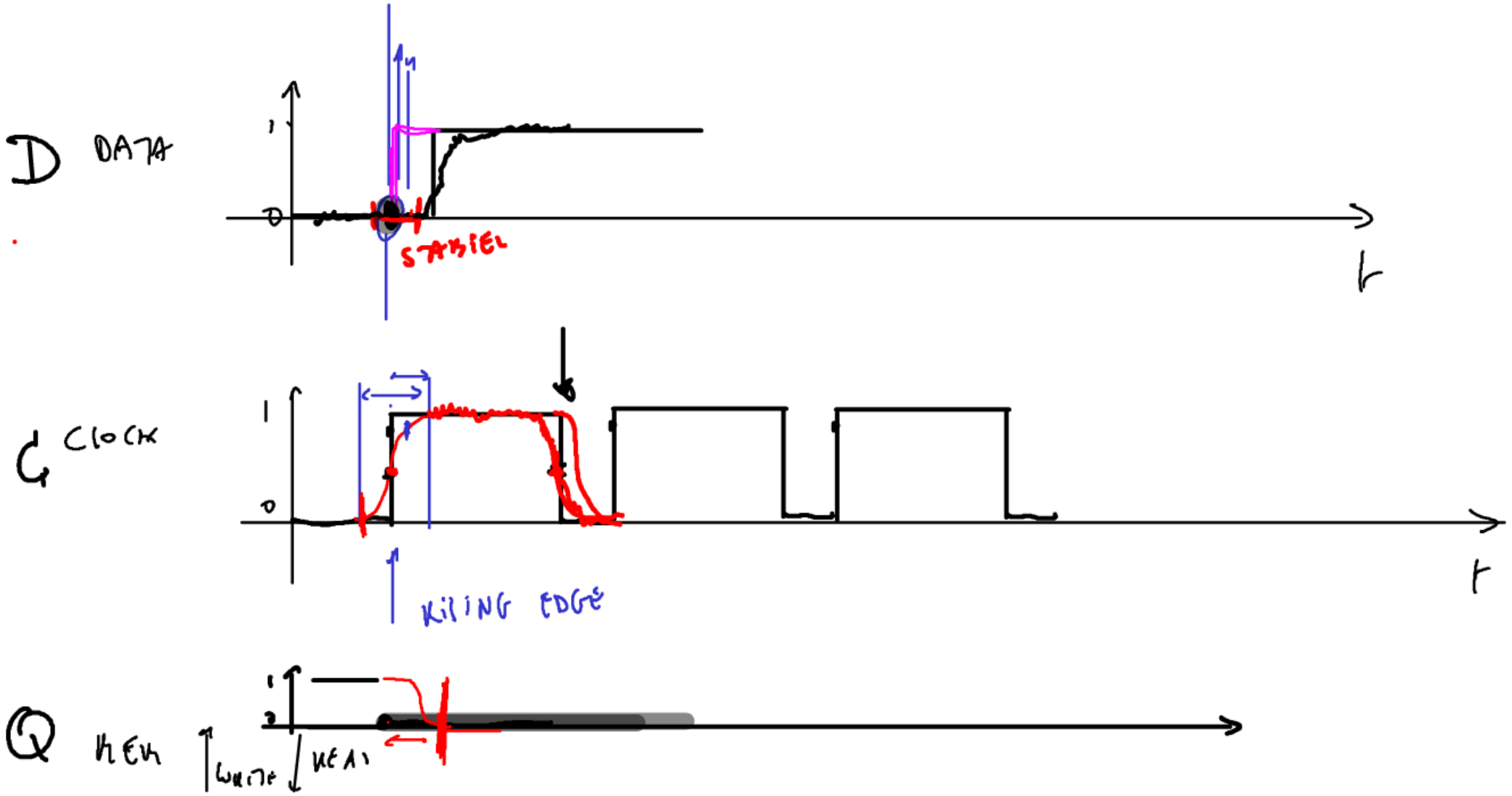
D flip-flop (not “transparent”) with falling-edge trigger



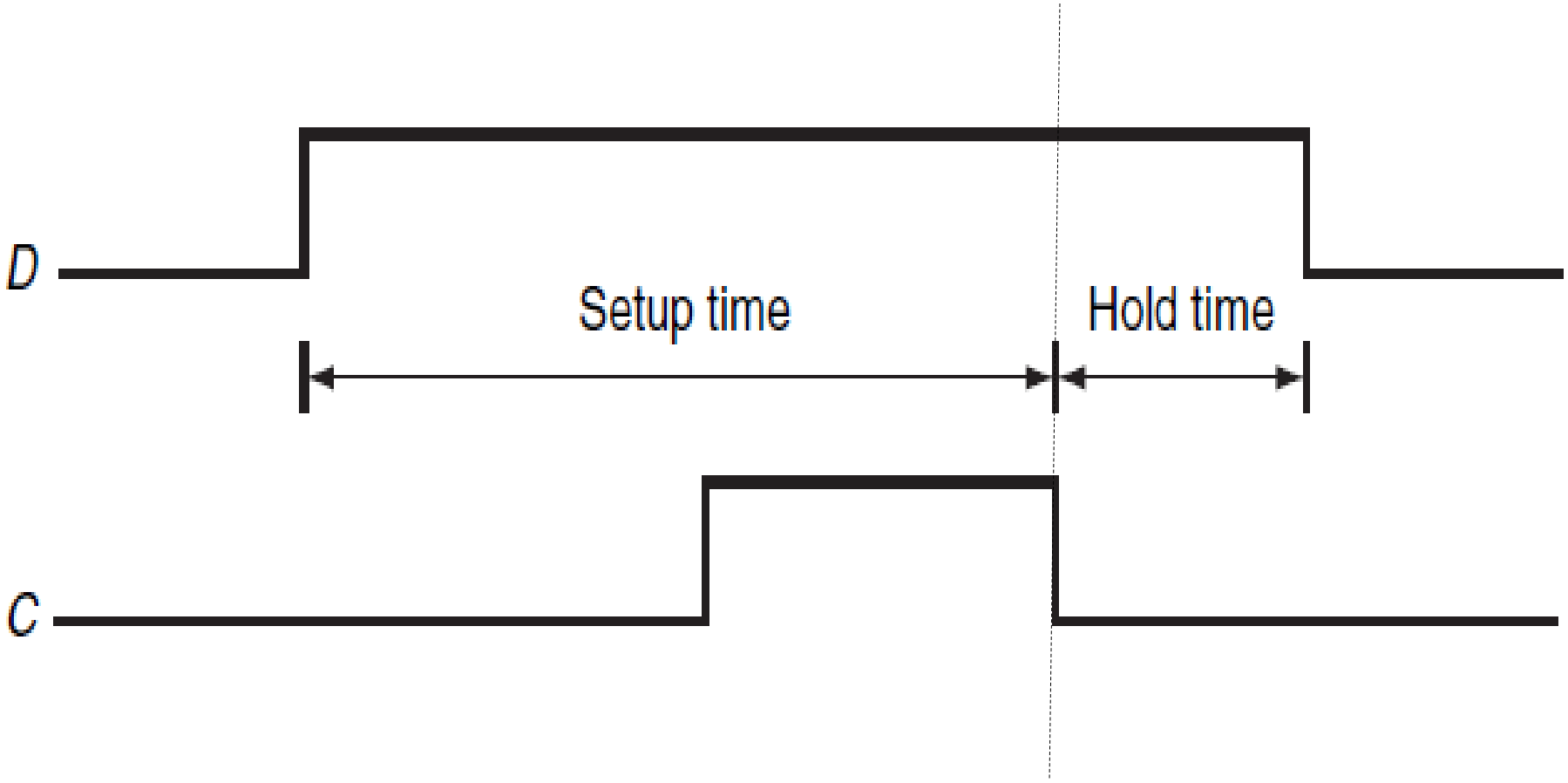
D flip-flop
with falling-edge trigger operation



timing constraints: “leaky” abstraction (rising edge)



D flip-flop (falling edge) timing constraints

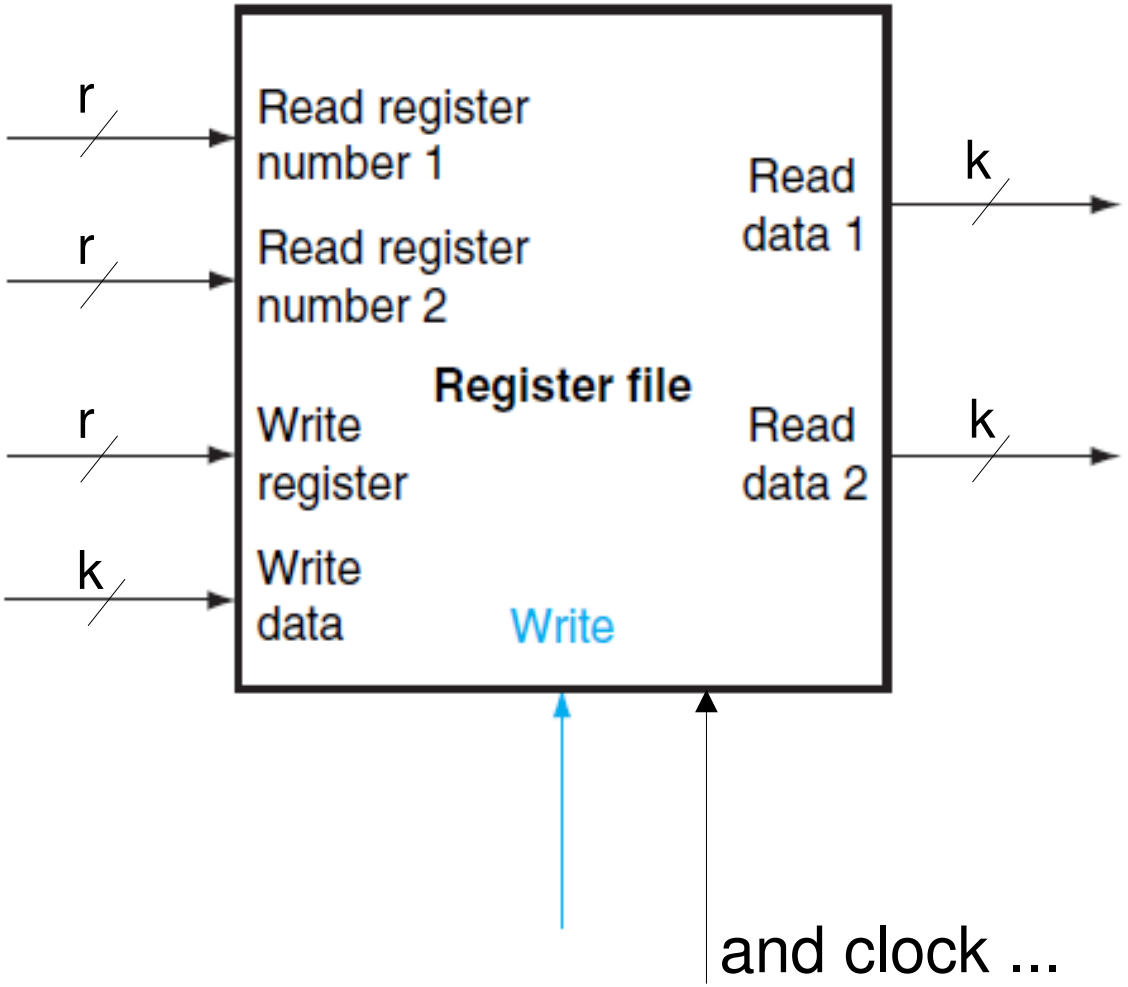


Register File

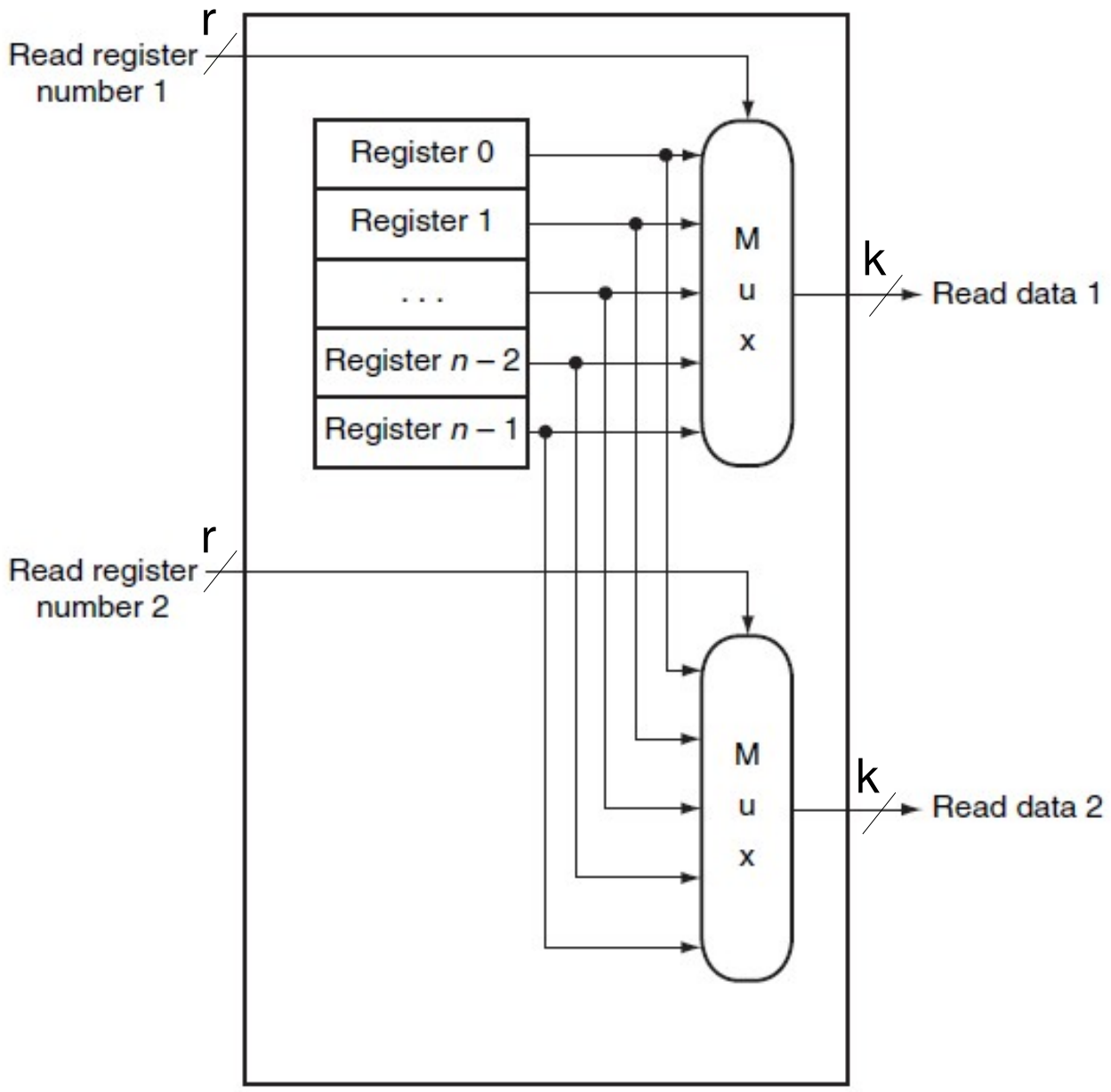
k-bit registers
(using k D flip-flops, see later: 4x2 SRAM)

number of registers n

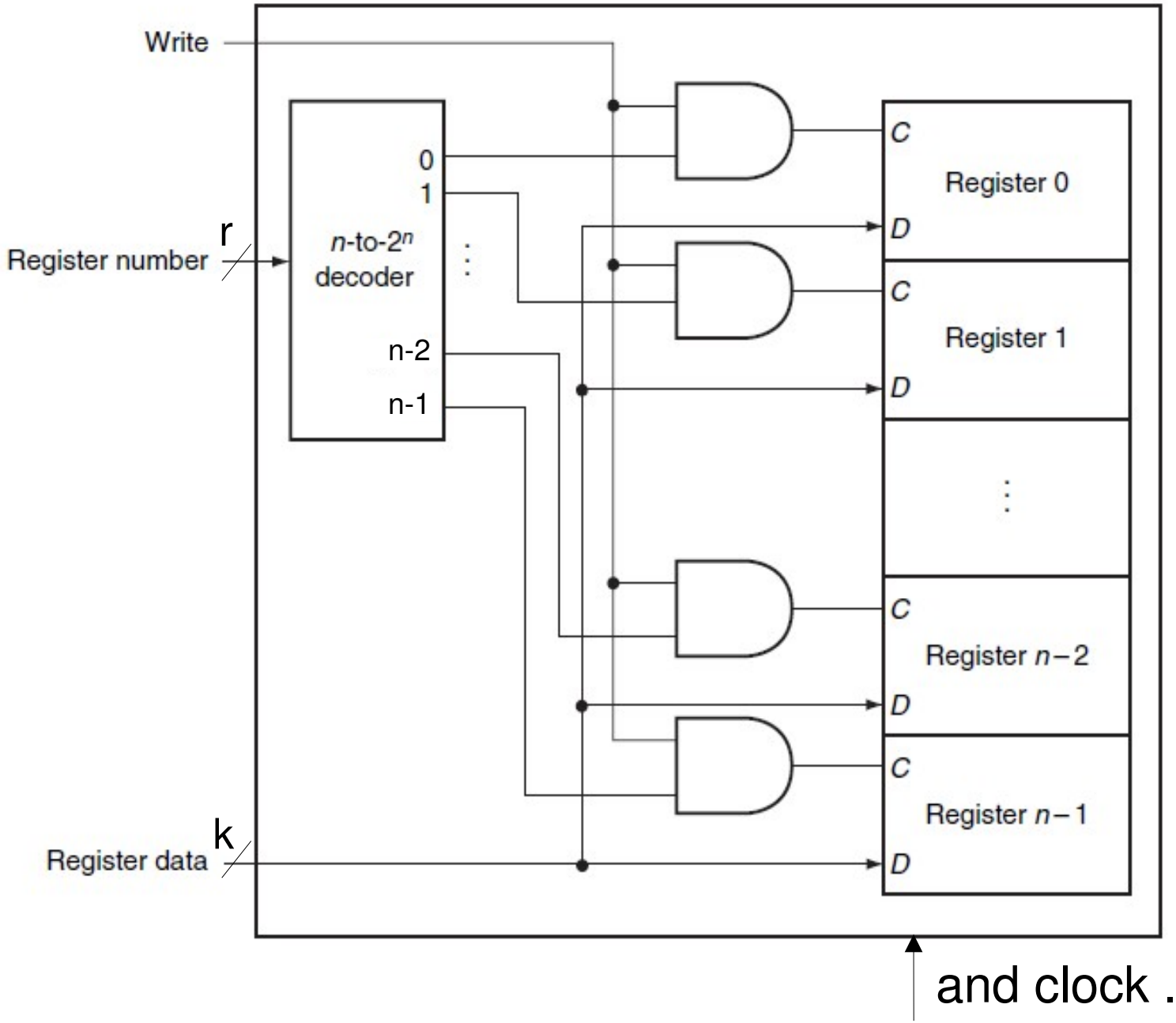
$\rightarrow r = \lceil \log_2 n \rceil$



Register File: read

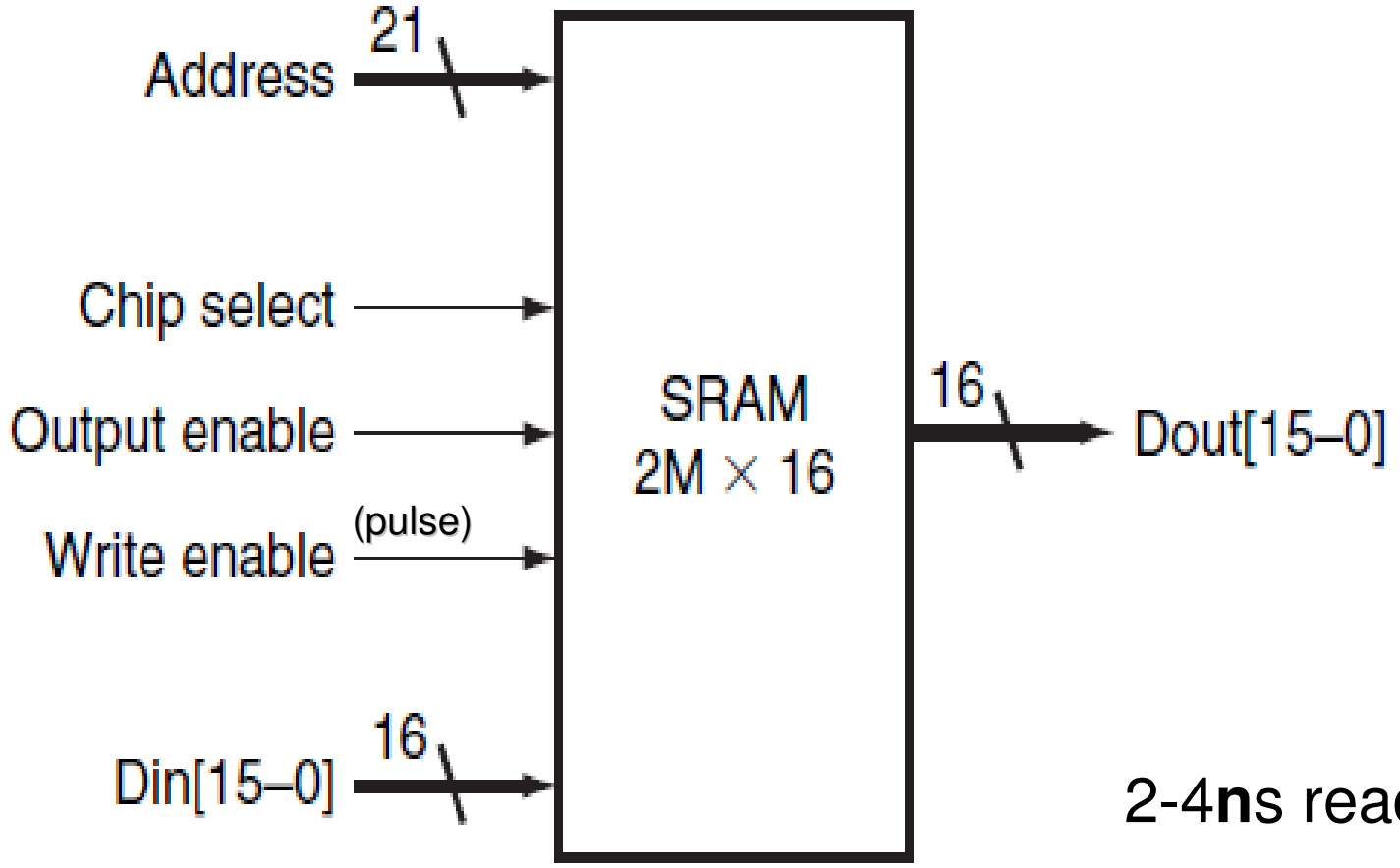


Register File: write



Large Memory:

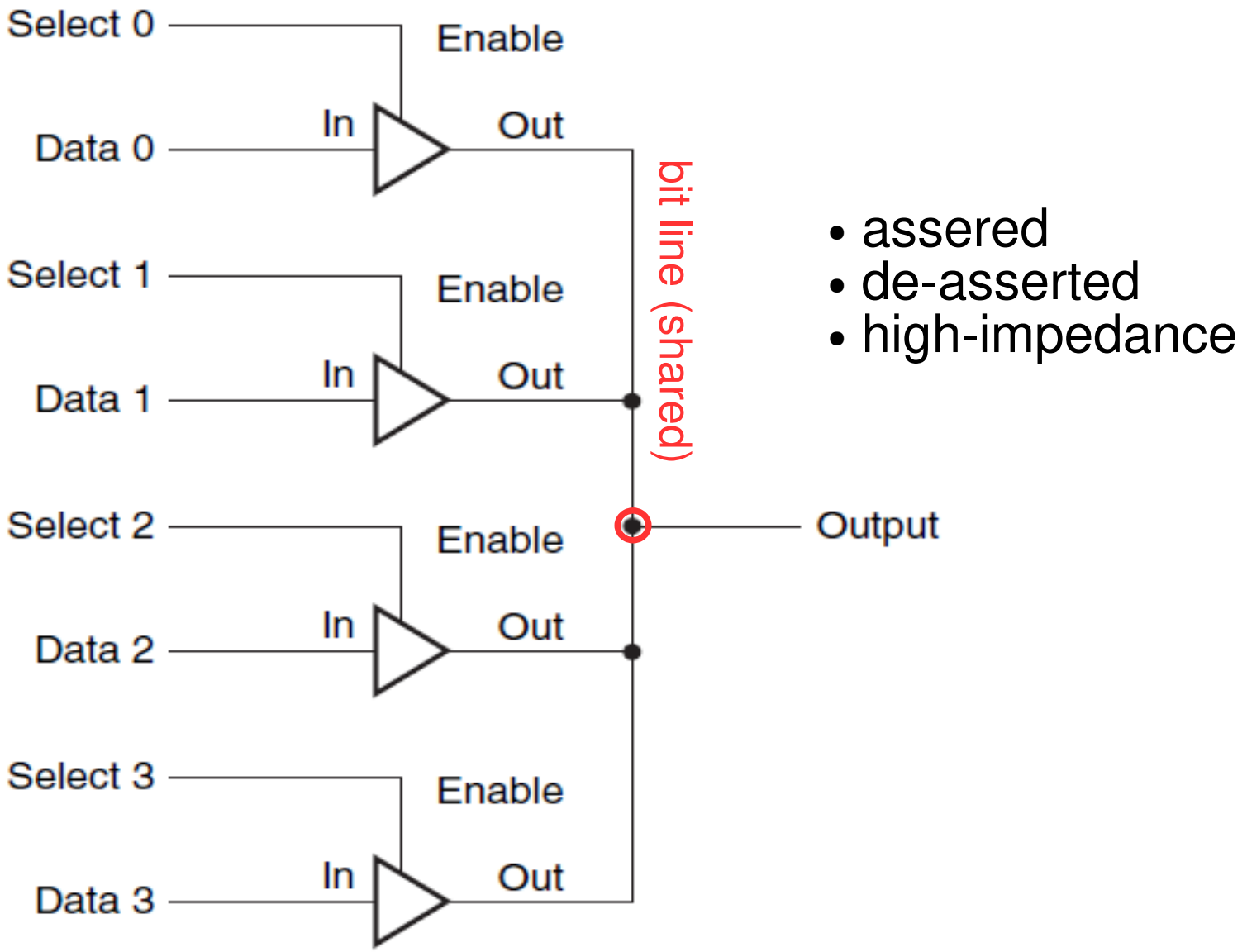
Static Random Access Memory (SRAM)
vs. Dynamic Random Access Memory (DRAM)



2-4**n**s read time

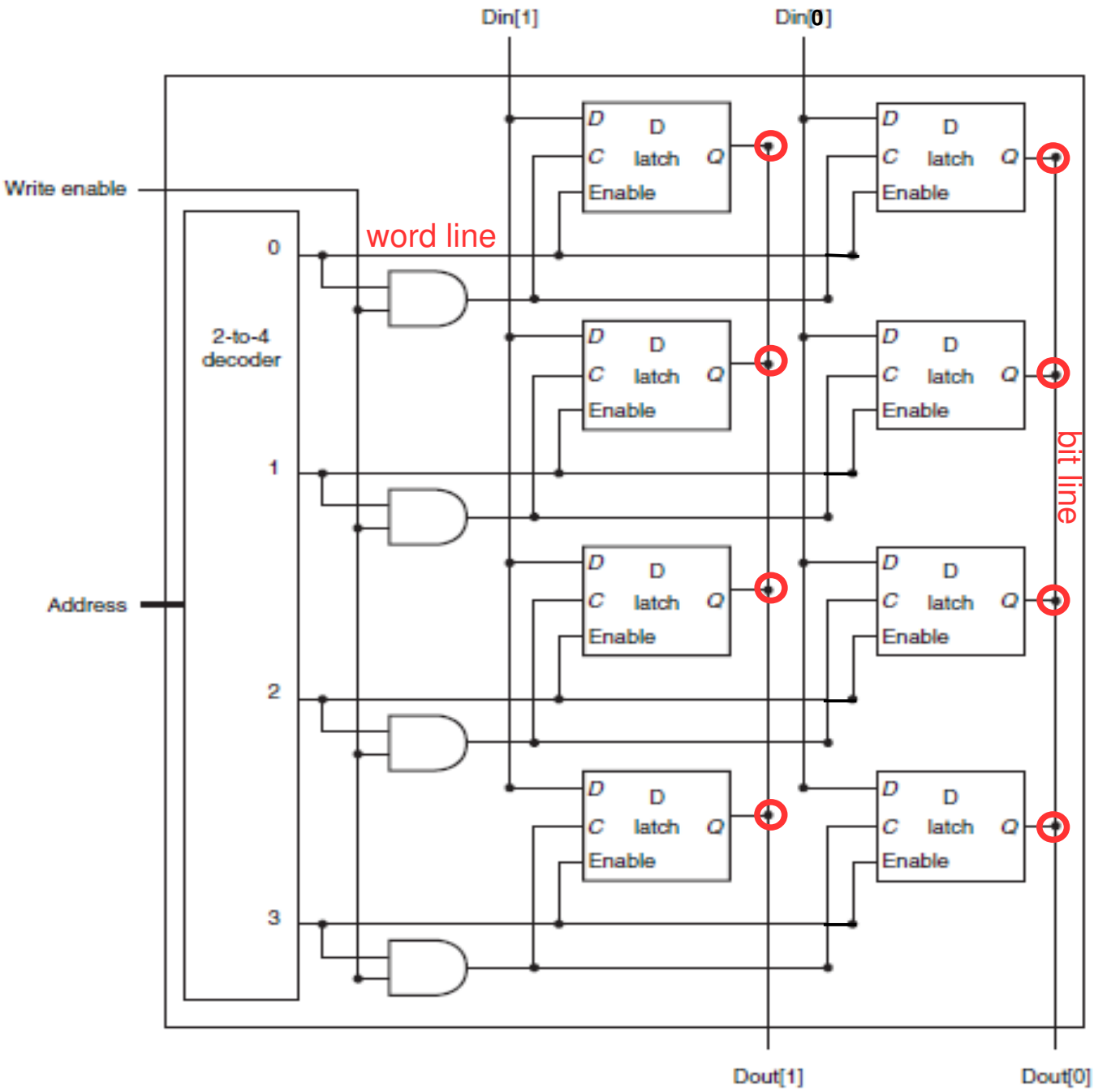
Random Access vs. Sequential Access: access time
RAM vs. ROM (Read Only Memory)

Three-state buffers (replace multiplexer)



- assered
- de-asserted
- high-impedance

4x2 SRAM (output enable, chip select omitted)



No large multiplexer at output!

SRAM **array** (read)

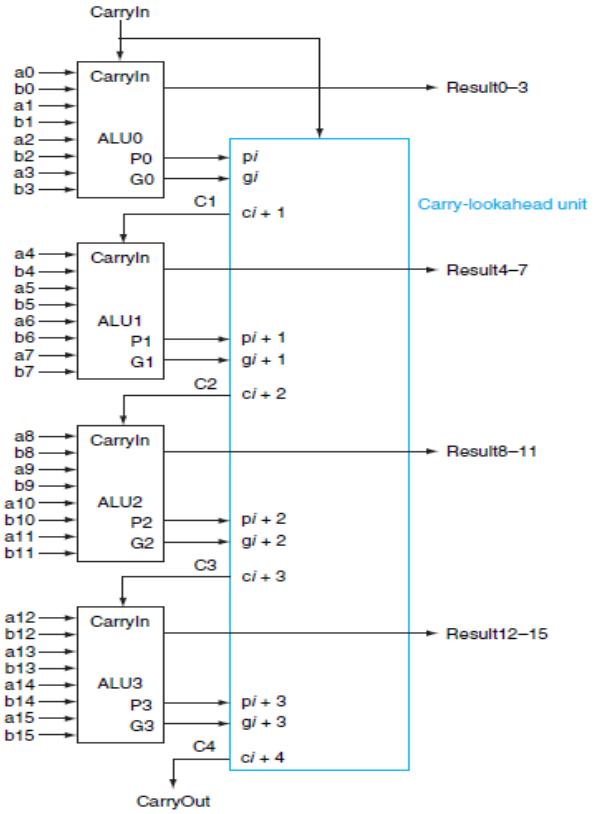
4Mx8

problem: need **huge** decoder !

solution: in analogy with carry lookahead adder
where two levels were used

use a two-level decoding process

→ slower
→ but can use a reasonably-sized decoder



ripple adder teveel delay ==> caryr look ahead

nu decoder te groot ==> 2 level decoding (volgende slide)

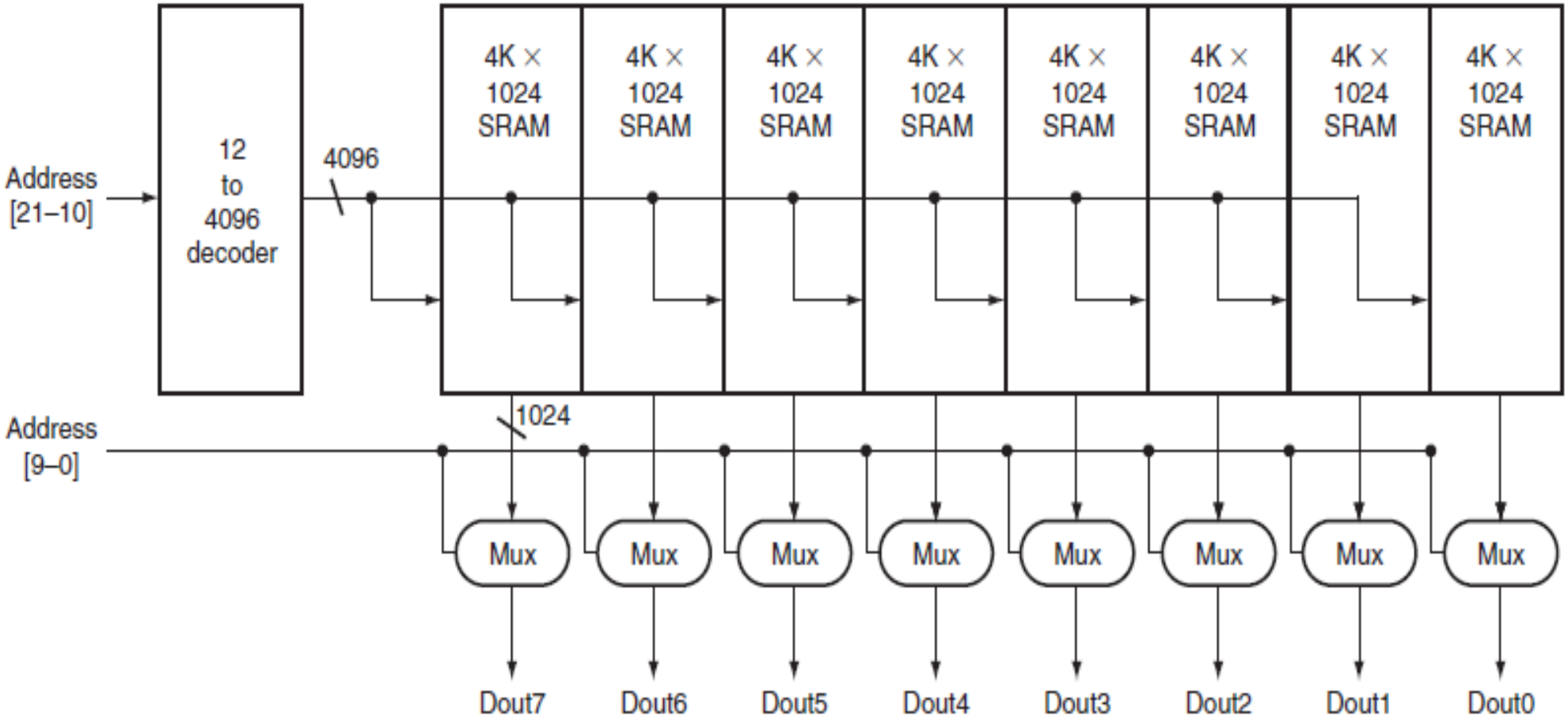
SRAM **array** (read)

4Mx8

2-level decoding process
to remove need for huge decoder

STATIC RAM !

verschillende geheugenbanken elk 1024 breed, allemaal flipflops
we gaan in parellel info eruit halen



met mux alle 8 bits kiezen

Both register memory and cache RAM memory
use SRAM technology: based on **flip-flops**.

Register memory:
+ 1-level decoder and
+ no three-state buffers but multiplexer

Cache memory
+ fast “buffer” between registers and main memory (data cache)
+ also instruction cache
+ different speeds and sizes: L1, L2, L3

snelste, iets minder snel, iets minder snek
==> natuurlijk L1 duurder

DRAM: Dynamic, use capacitor instead of flip-flop

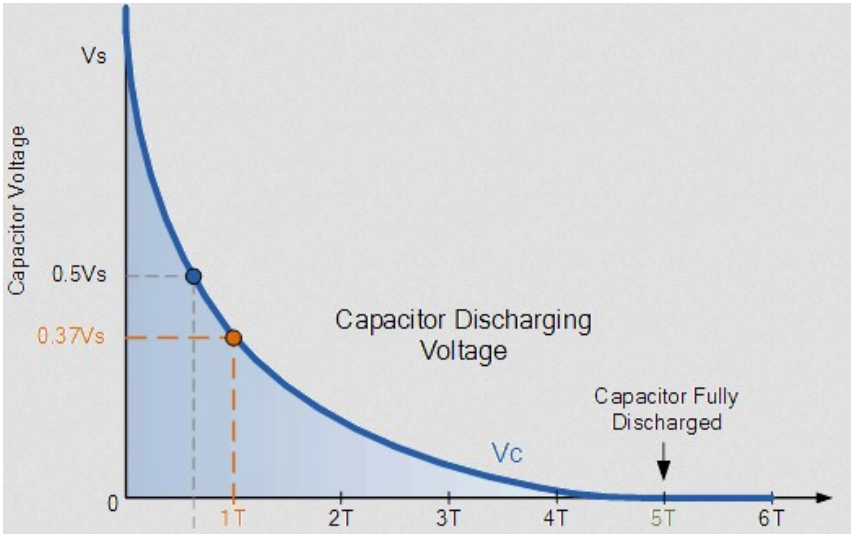
store information in capacitor →

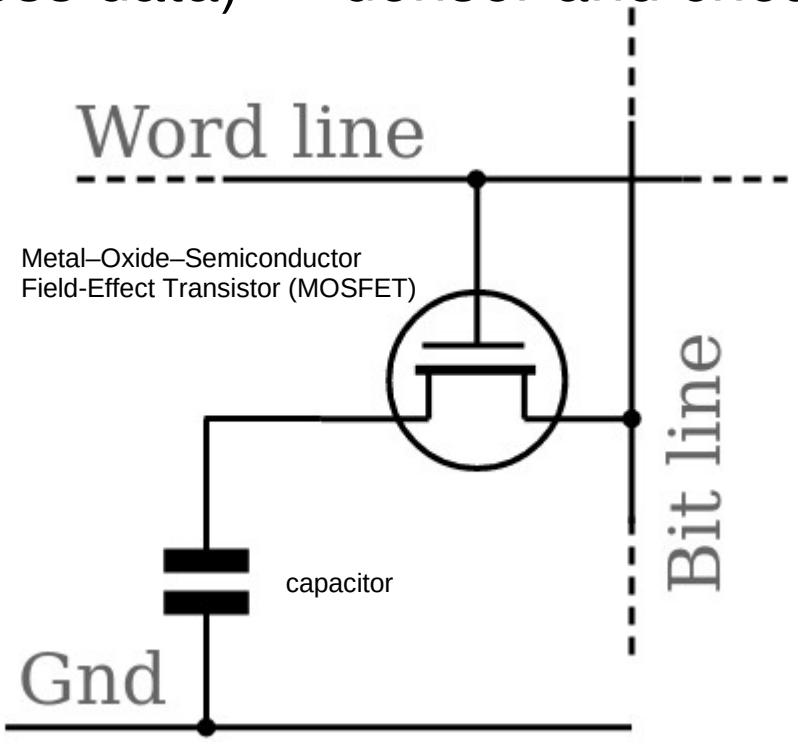
volatile (even when powered), needs **refresh**

(andere is alleen volatile wanneer stroom uitvalt, hier ook refreshen wnr stroom er is)

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

dit is normale RAM van pc





maar 1 transistor ipv 24

read/write/**refresh**

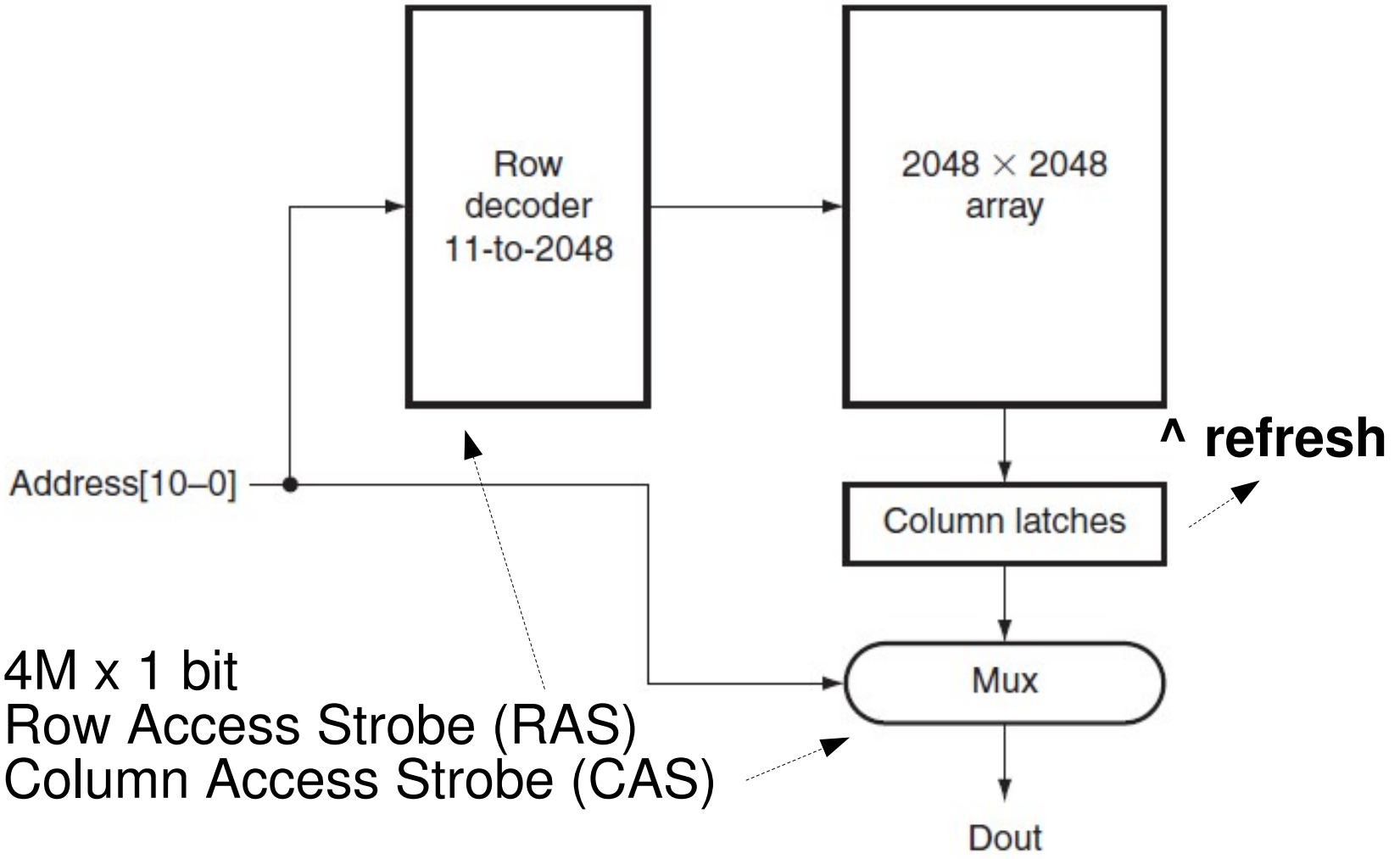
(rate = **m**s, 64ms row refresh)

refresh by read and write-back

moet om de 64ms refreshen om te zeggen wat de waarde is

DRAM array

niet op ingaan



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

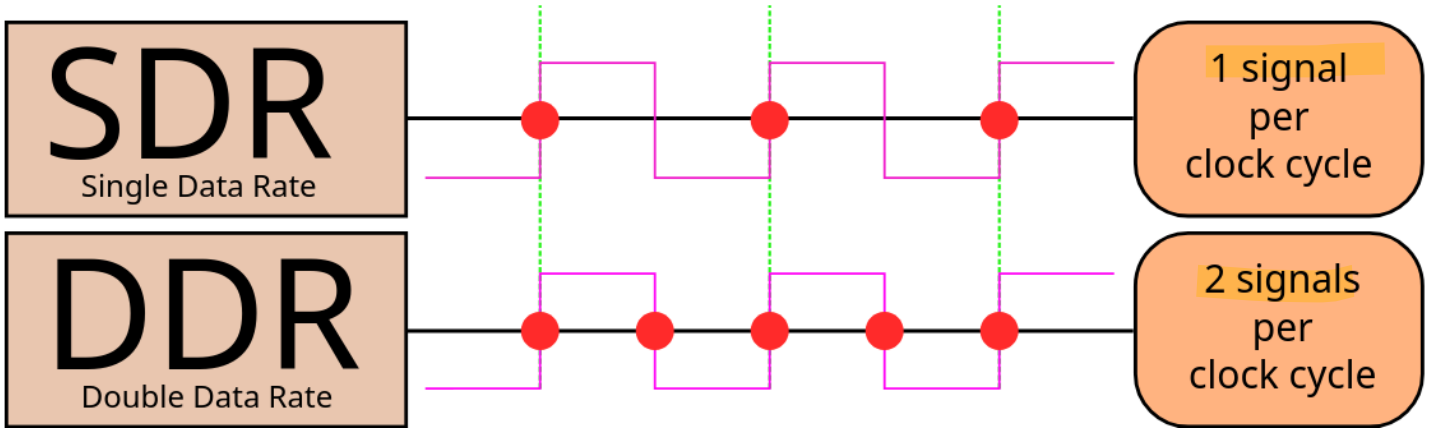
45-65**n**s read time

Asynchronous DRAM (ADRAM) vs.
Synchronous RAM
SDRAM (for general memory) and **S**SRAM (for cache memory):

"burst" of data from a series of **sequential addresses**
(use **clock**, not address)

Synchronized (on rising clock edge) with datapath

SDR vs. DDR

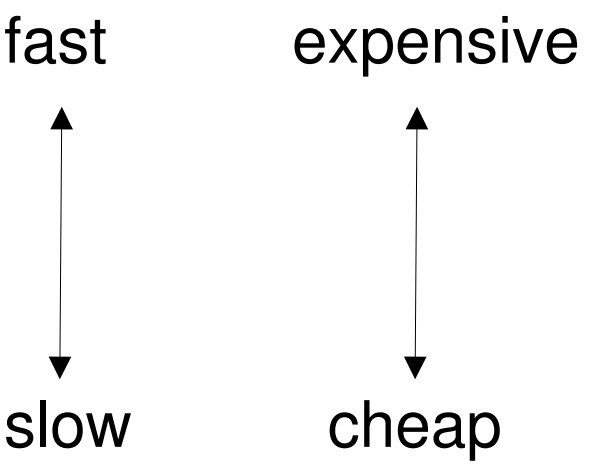


"double pumping": transfer data on both rising **and** falling edge

DDR, DDR2, DDR3, DDR4, DDR5

Memory Technology

- Static RAM (SRAM)
  0.5ns – 2.5ns, $2000 – $5000 per GiB
- Dynamic RAM (DRAM)
  50ns – 70ns, $20 – $75 per GiB
- Magnetic disk
  5ms – 20ms, $0.20 – $2 per GiB



caching

virtueel geheugen

Ideal memory:

- Access time of SRAM
- Capacity and cost/GiB of magnetic disk

register spilling, hetgene da we ni meer nodig hebbe in STATiC
==> register spillin gnaar DYNAMIC

Principle of **Locality**

- Programs access a **small proportion** of their address space (instruction/data) at any time

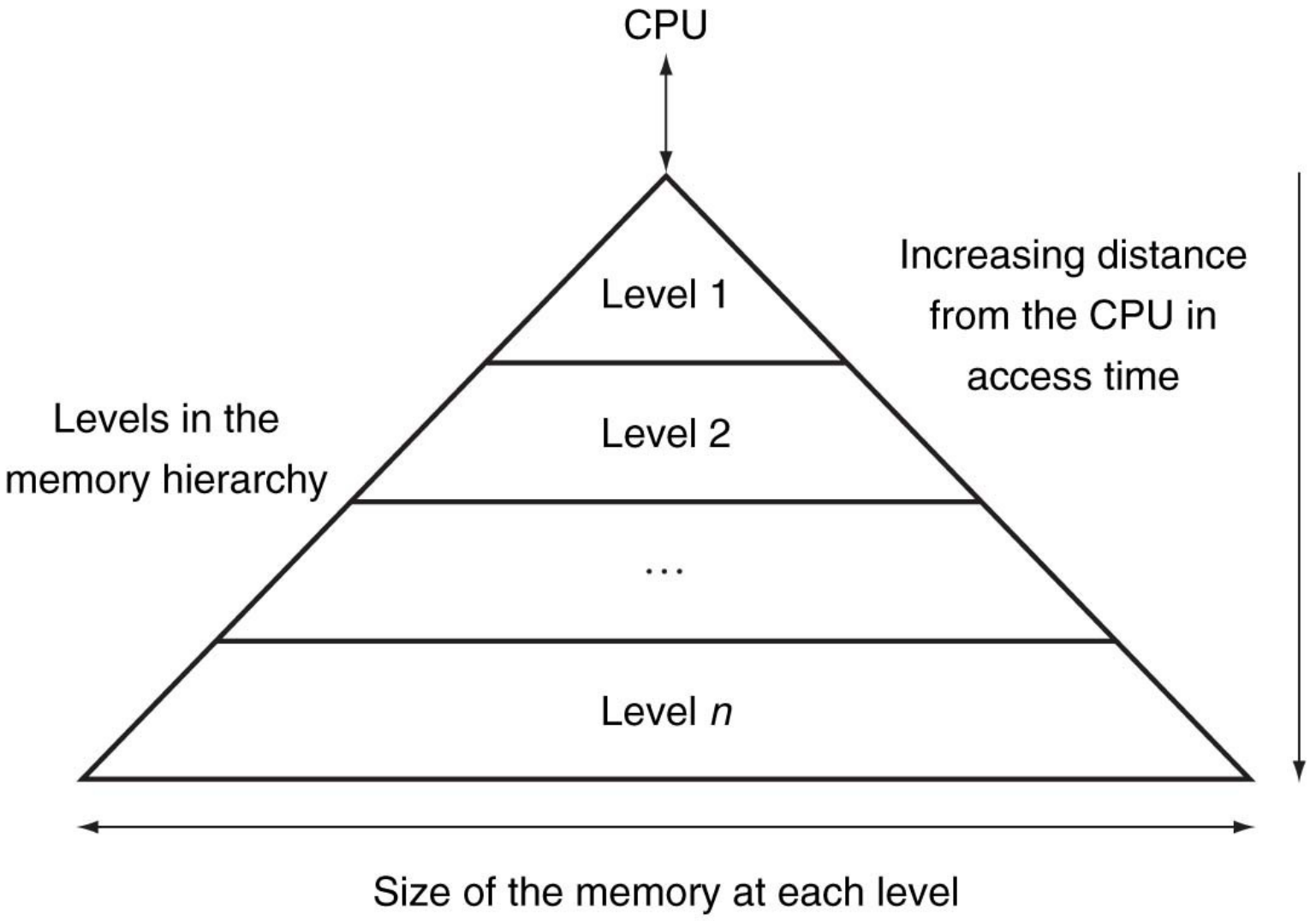
- **Temporal** locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables

- **Spatial** locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

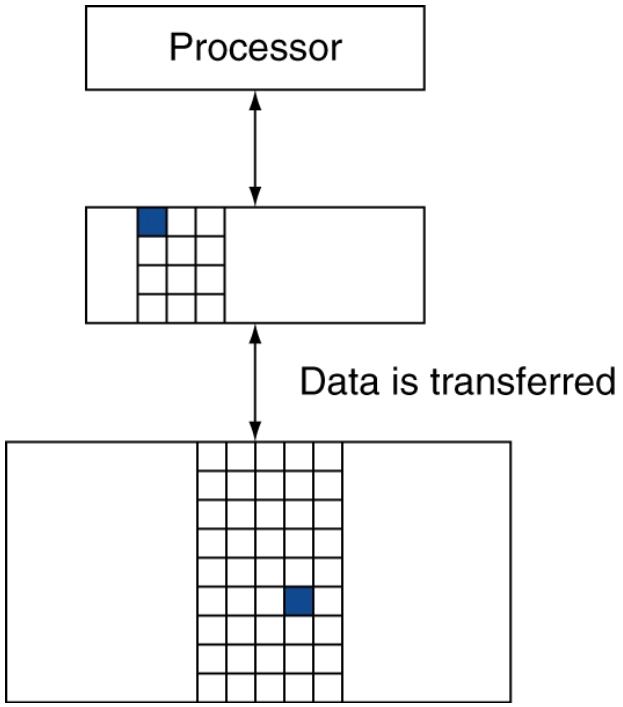
Using Locality: **Memory Hierarchy**

- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU

Memory Hierarchy Levels



Memory Hierarchy Levels



- **Block** (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - **Hit**: access satisfied by upper level
    - **Hit ratio**: hits/accesses
- If accessed data is absent
  - **Miss**: block copied from lower level
    - Time taken: **miss penalty**
    - **Miss ratio**: misses/accesses = 1 – hit ratio
  - Then accessed data supplied from upper level

probeer iets te vinden, is er niet
naar volgend niveau

more about memory hierarchy algorithms in the Operating Systems course

- between cpu/registers and main memory (“cache”)
- between main memory and disk memory (“virtual memory”)

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`

manieren waarop het fout kan gaan
1 bit error

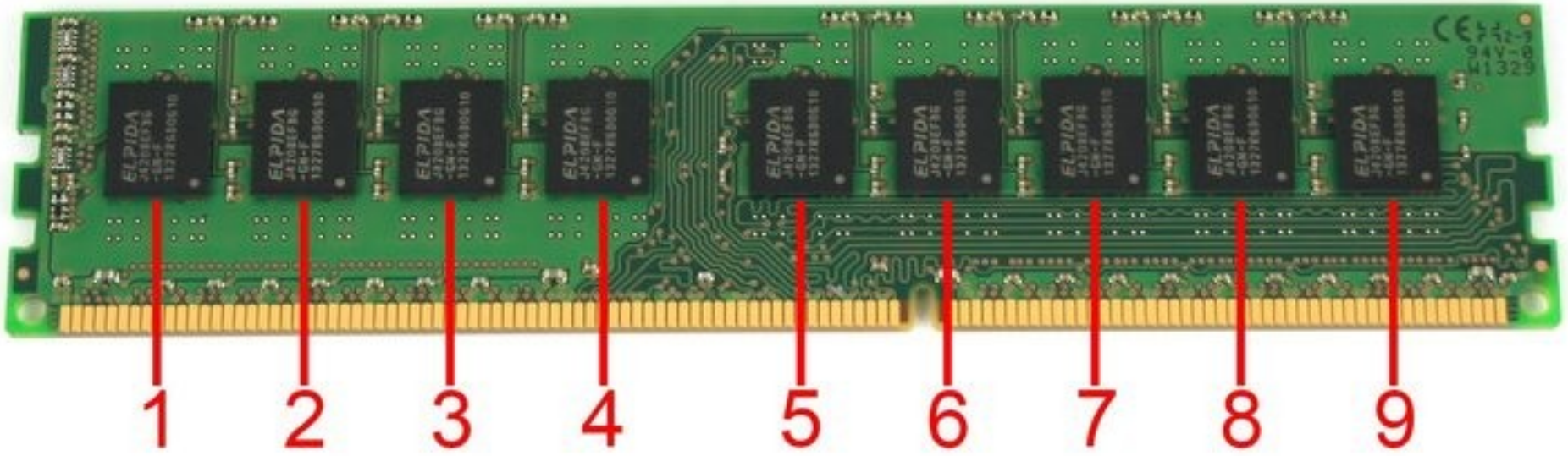
ECCode `011` appears in table for

- `0110` (**edit distance** 1) ← most likely
- `0001` (edit distance 3, 2, 2, 2 respectively)

in een server wil je echt geen fouten, dan ook meer chips
8 chips + 1 chip voor error correction bij te houden

ECC RAM

aka “server class” memory

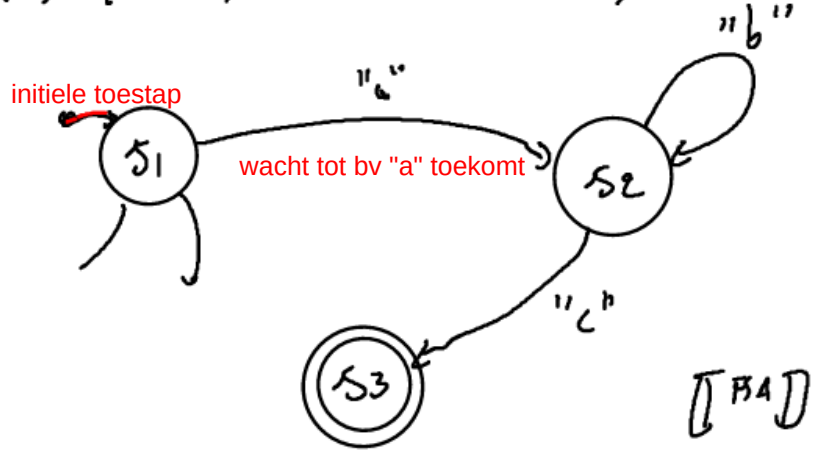


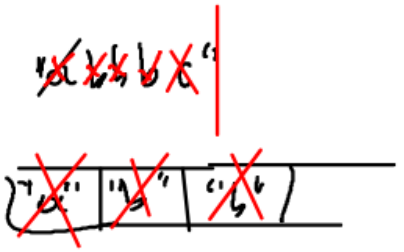
Non-ECC RAM

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

[FSA]: { "abc", "abbbc", ... }

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

$s_1 = s_1$

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

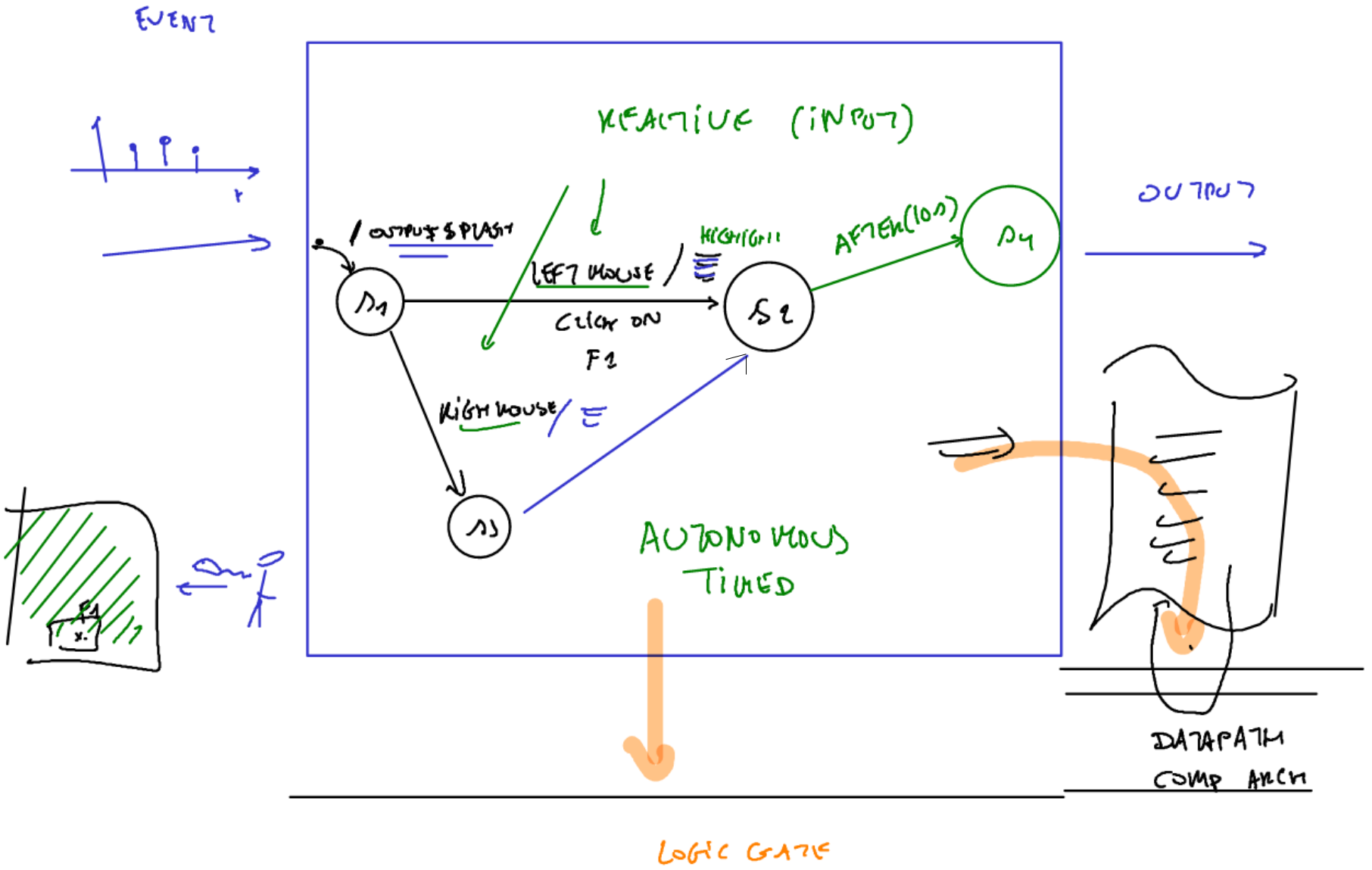
$L = \{ "a", "b", "c" \}$

$Acc = \{ s_3 \}$

for “language” recognition

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

getimede, reactieve en autonomie systemen



luisterd naar input van buitenwereld en geeft output
ook heeft de output een notie van tijd, "wacht tot ..."

reageren op input, outputs geven, ...

for system (behaviour) specification (and synthesis)

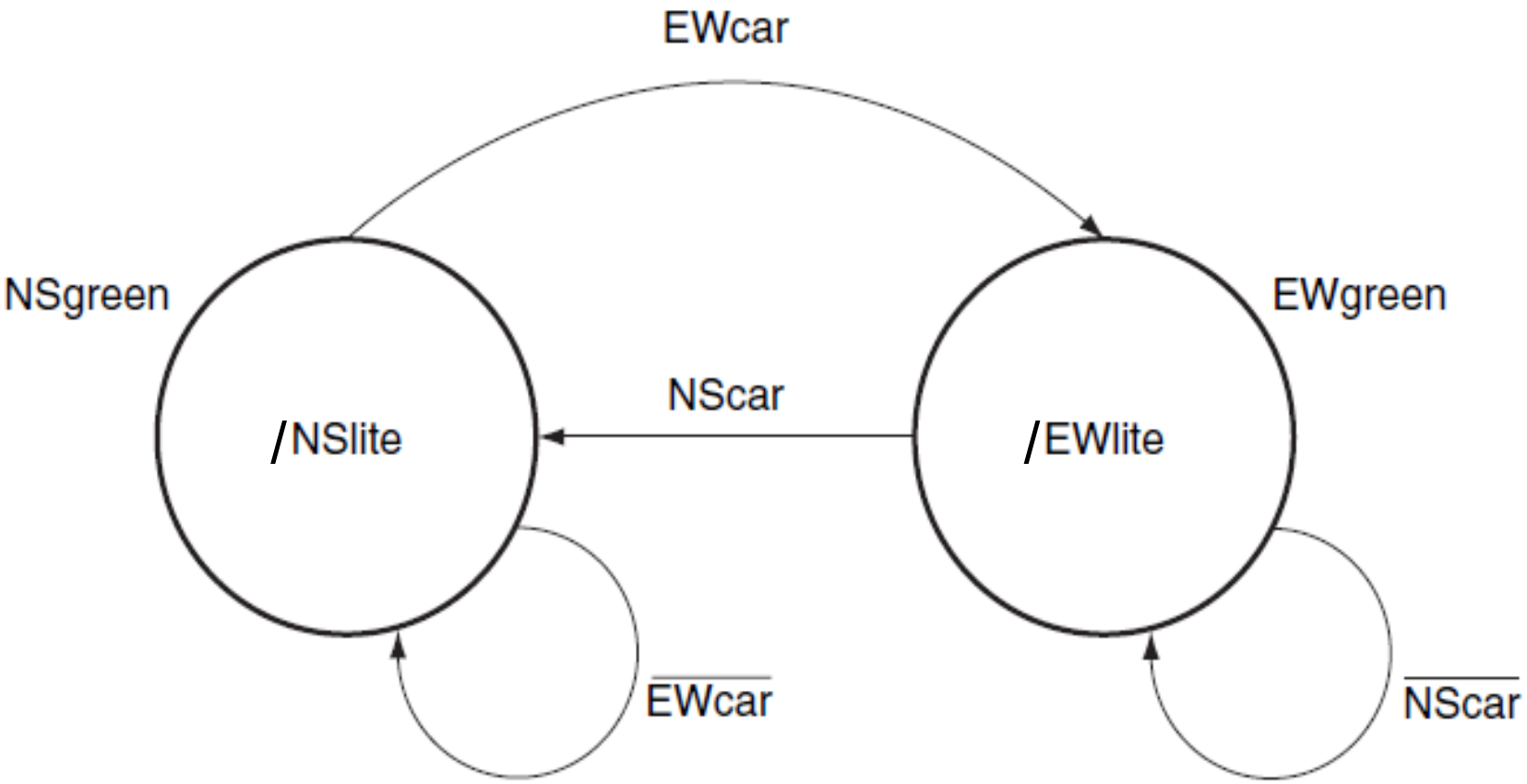
Traffic Light (Discrete-Time – DT)

	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

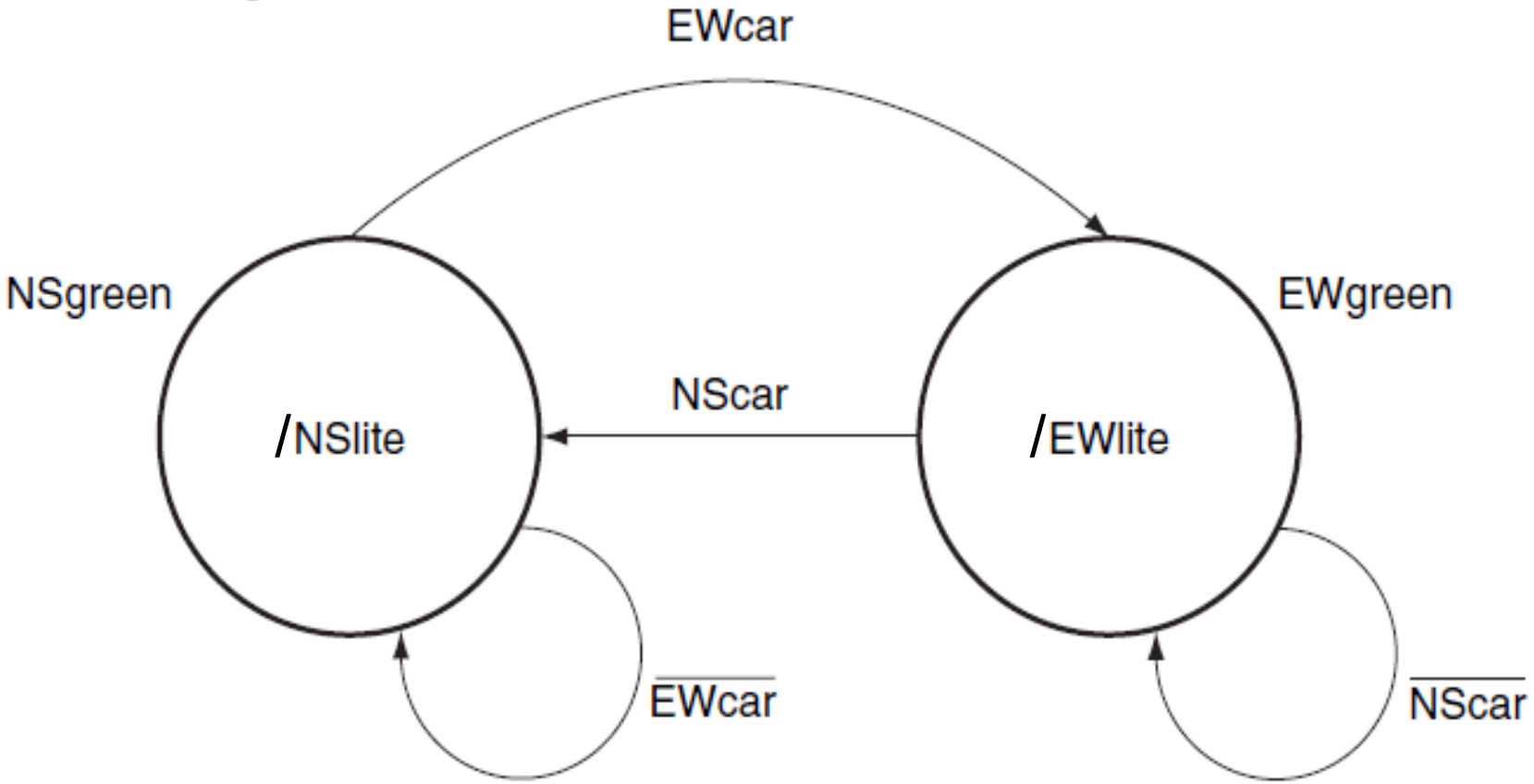
fairness (in case of conflict)

	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

Traffic Light



Traffic Light

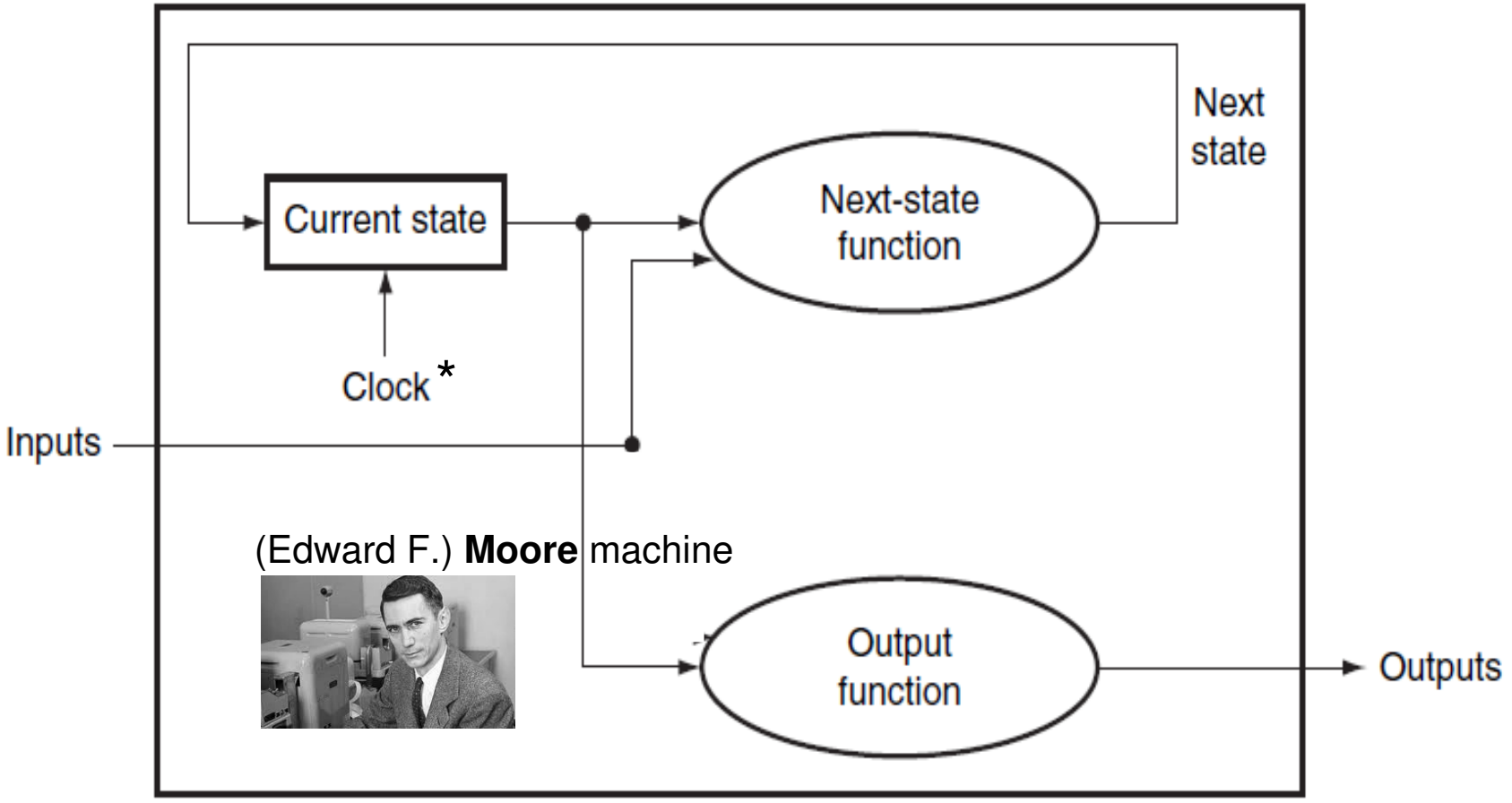


encode `CurrentState (NSgreen or Ewgreen)` in 1 bit

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

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

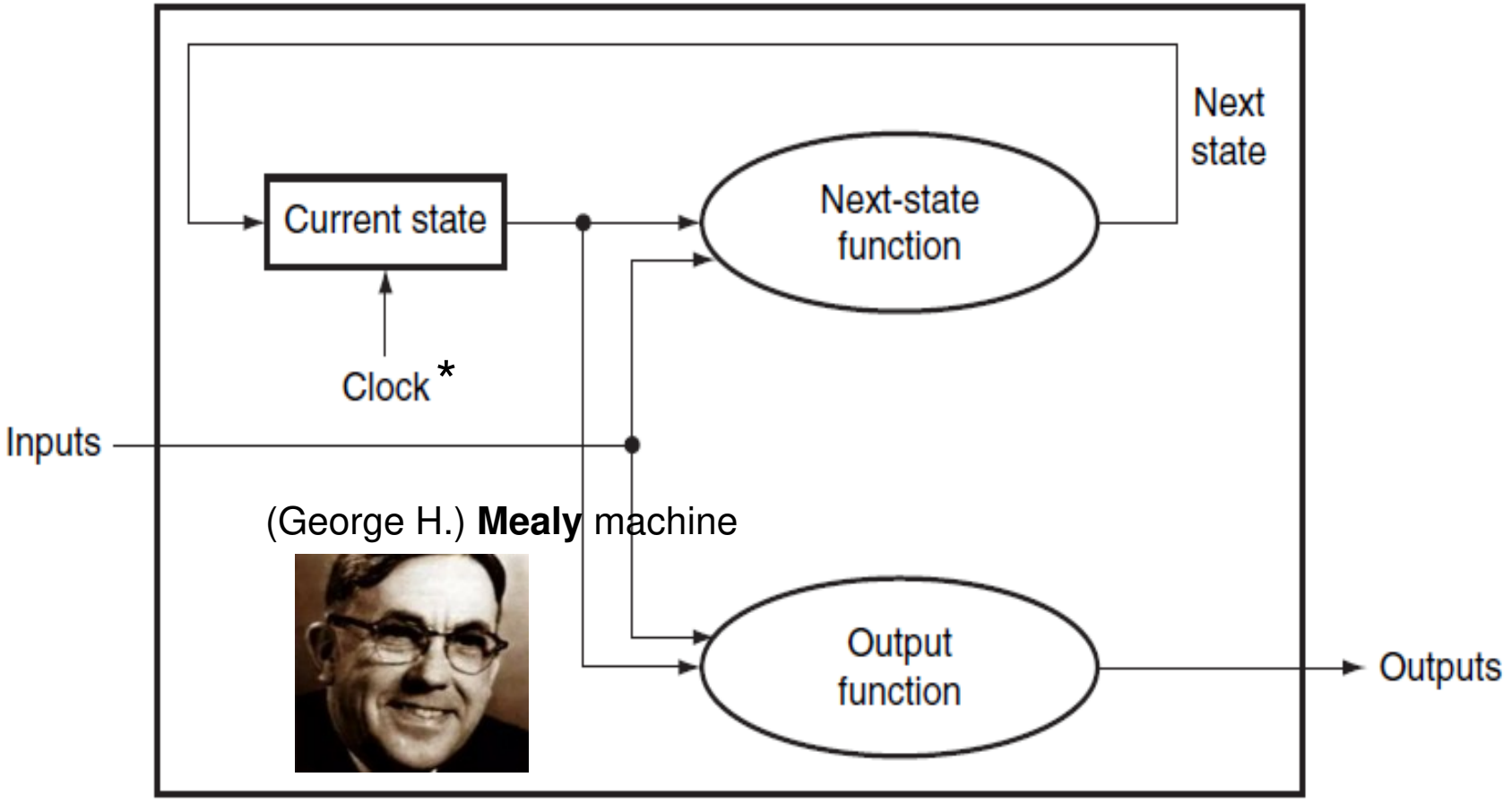
Finite **State** Machines



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

output enkel gebaseerd op huidige toestand

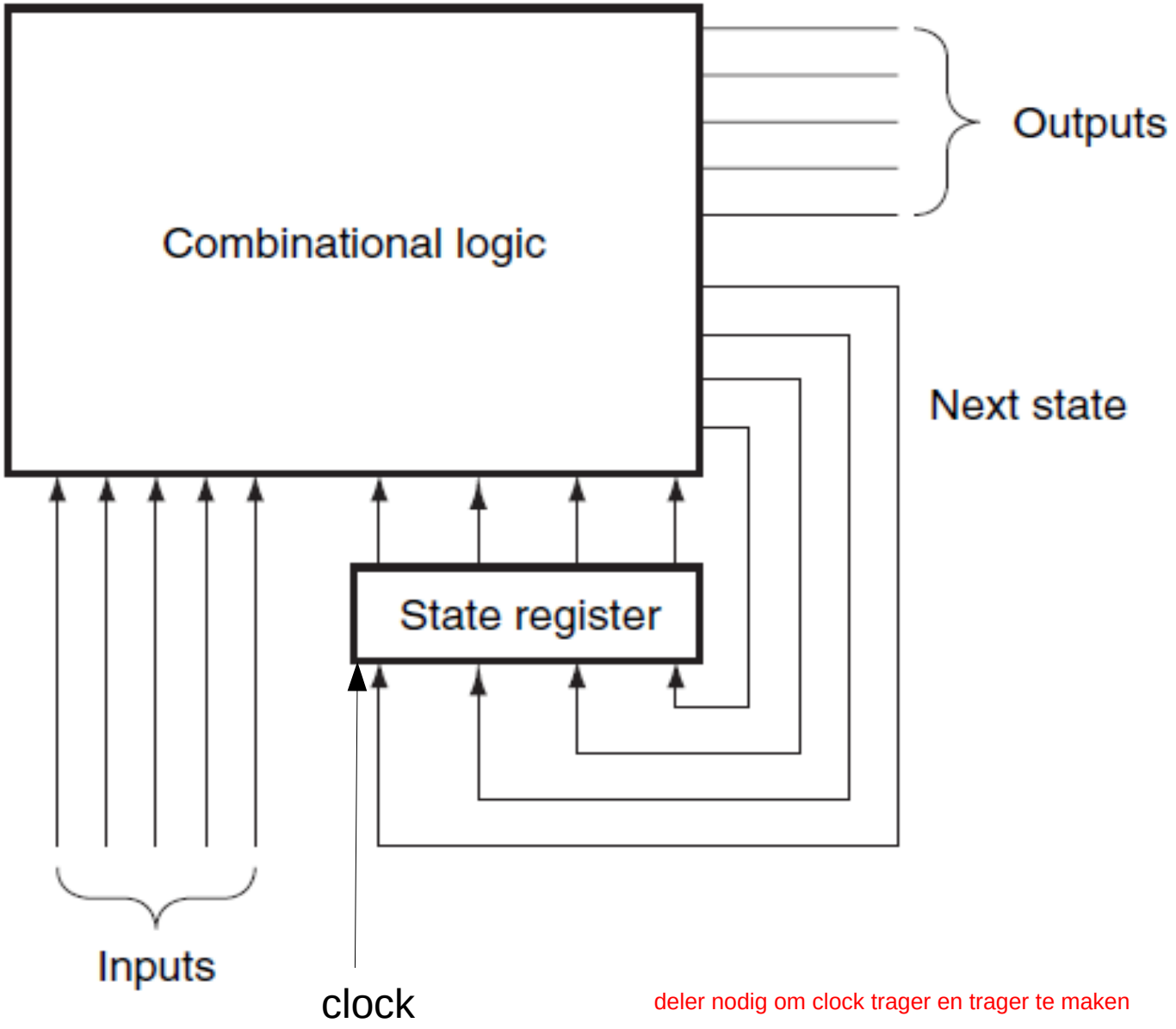
Finite **State** Machines



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

output gebaseerd op huidige toestand en input

Traffic Light



deler nodig om clock trager en trager te maken