Sequential Logic

Jinyang Li

What we’ve learnt so far

- Combinatorial logic
  - Truth table $\rightarrow$ sum of products circuits
  - E.g. Multiplexors (Mux), Decoders
- ALU
  - Ripple carry
  - Carry lookahead

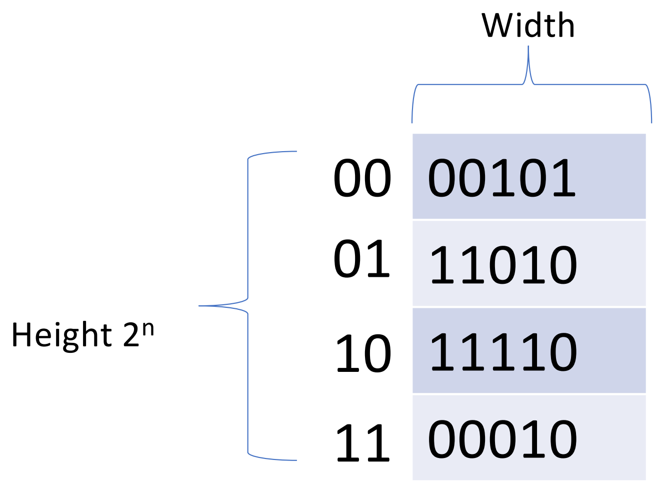
Two types of logic circuits

- Combinatorial circuit
  - Truth table → sum of products
  - ROM
- Sequential circuit
  - output is dependent on both input and state (memory elements)

Today’s lesson

ROM-based implementation of CL

- ROM (read-only memory)



00	00101
01	11010
10	11110
11	00010

ROM of 2 address lines and 5 bits per entry

ROM-based implementation of CL

Example CL : Count # of 1’s in 3-bit inputs

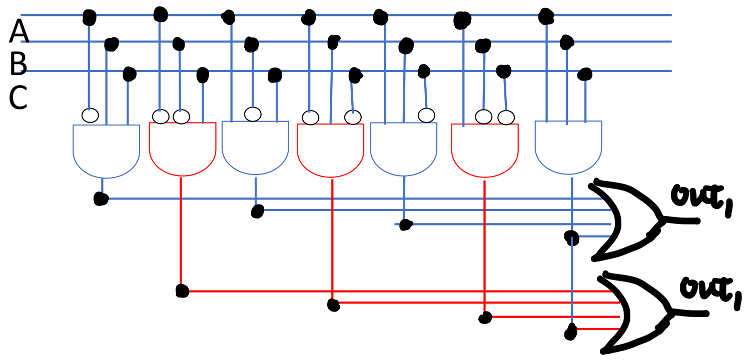
A	B	C	out_1	out_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

000:	00
001:	01
010:	01
011:	10
100:	01
101:	10
110:	10
111:	11

ROM of height 8 width 2

ROM vs. PLA (sum of products)

- ROM contains more entries than PLA (2^n)



7 products (AND)

More efficient!

000:	00
001:	01
010:	01
011:	10
100:	01
101:	10
110:	10
111:	11

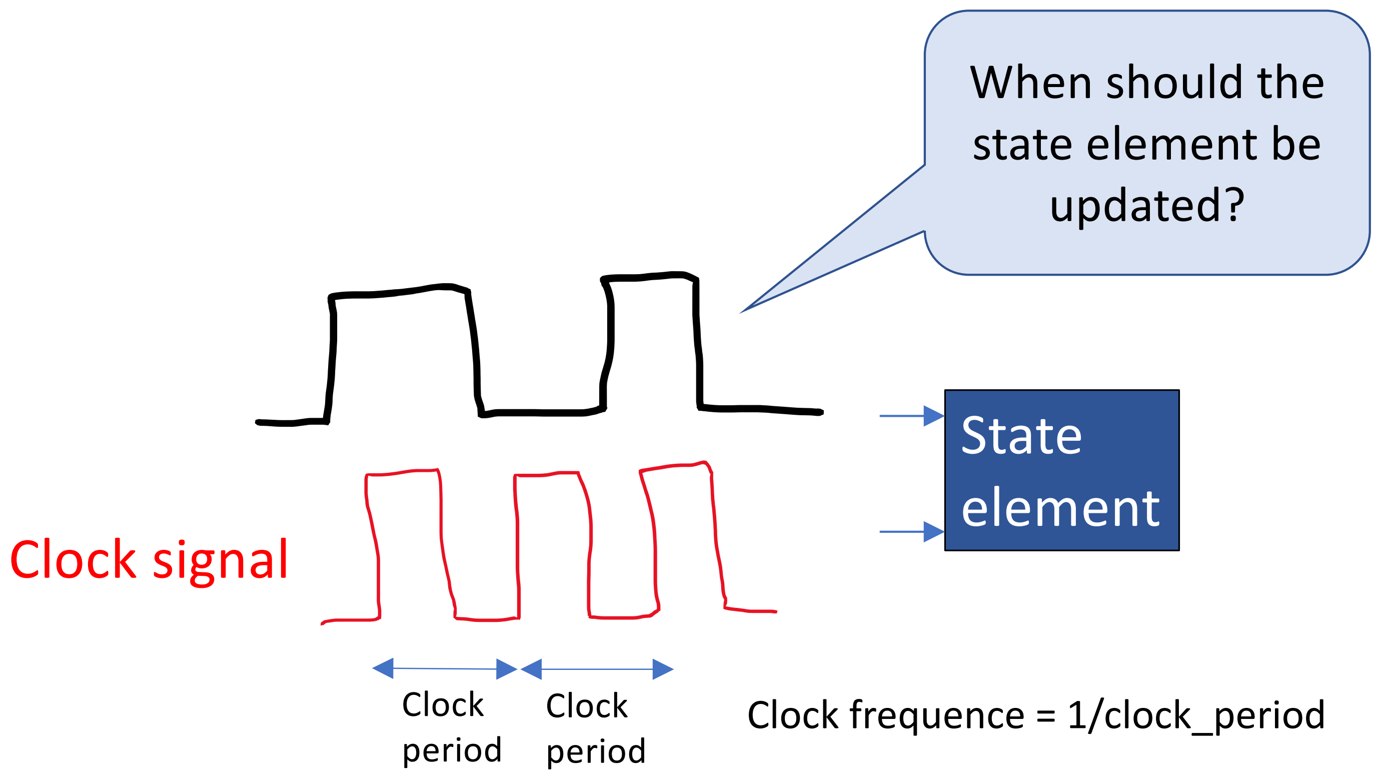
8 entries

Easy!

Two types of logic circuits

- Combinatorial circuit
  - Truth table $\rightarrow$ sum of products
  - ROM
- Sequential circuit
  - output is dependent on both input and state (memory elements)

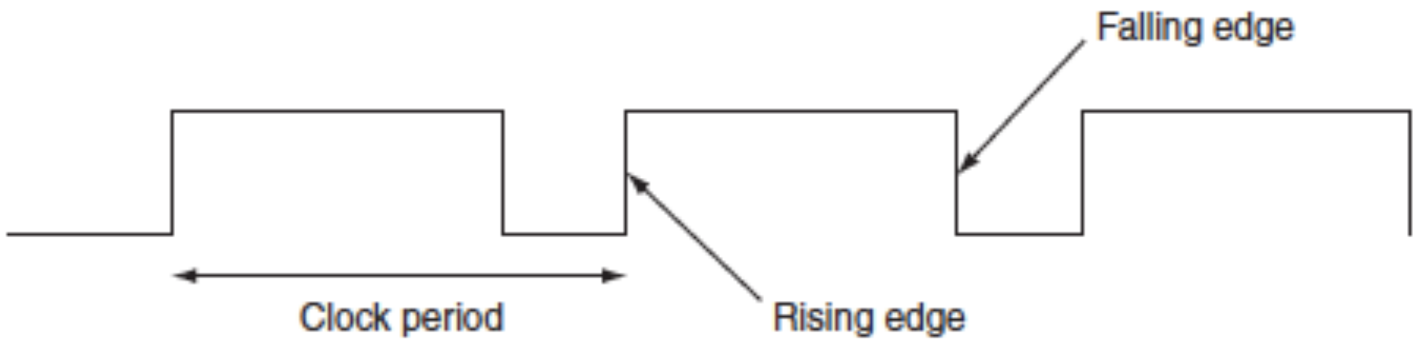
Sequential logic require clocks



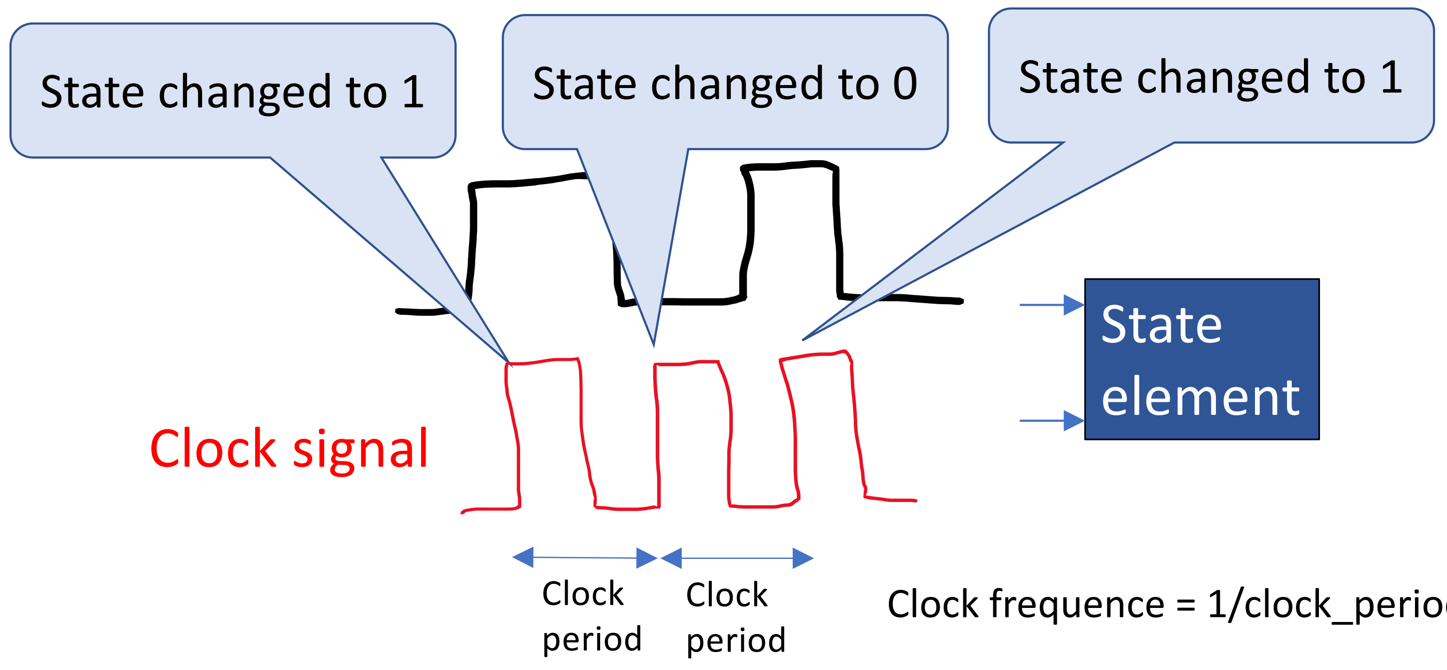
Clock frequence = 1/clock_period

Clocks

- Edge-triggered clocking: state content only changes on active clock edge

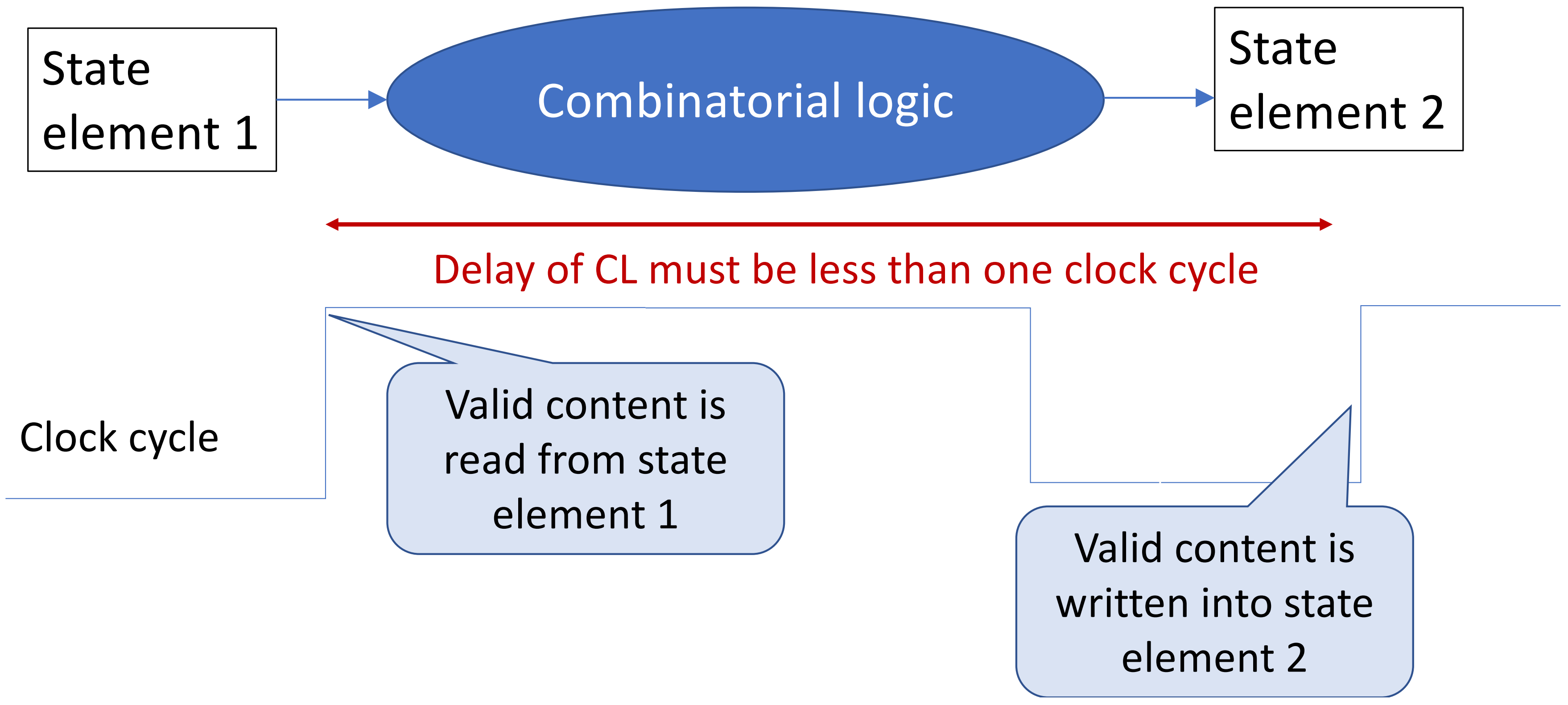


Sequential logic require clocks

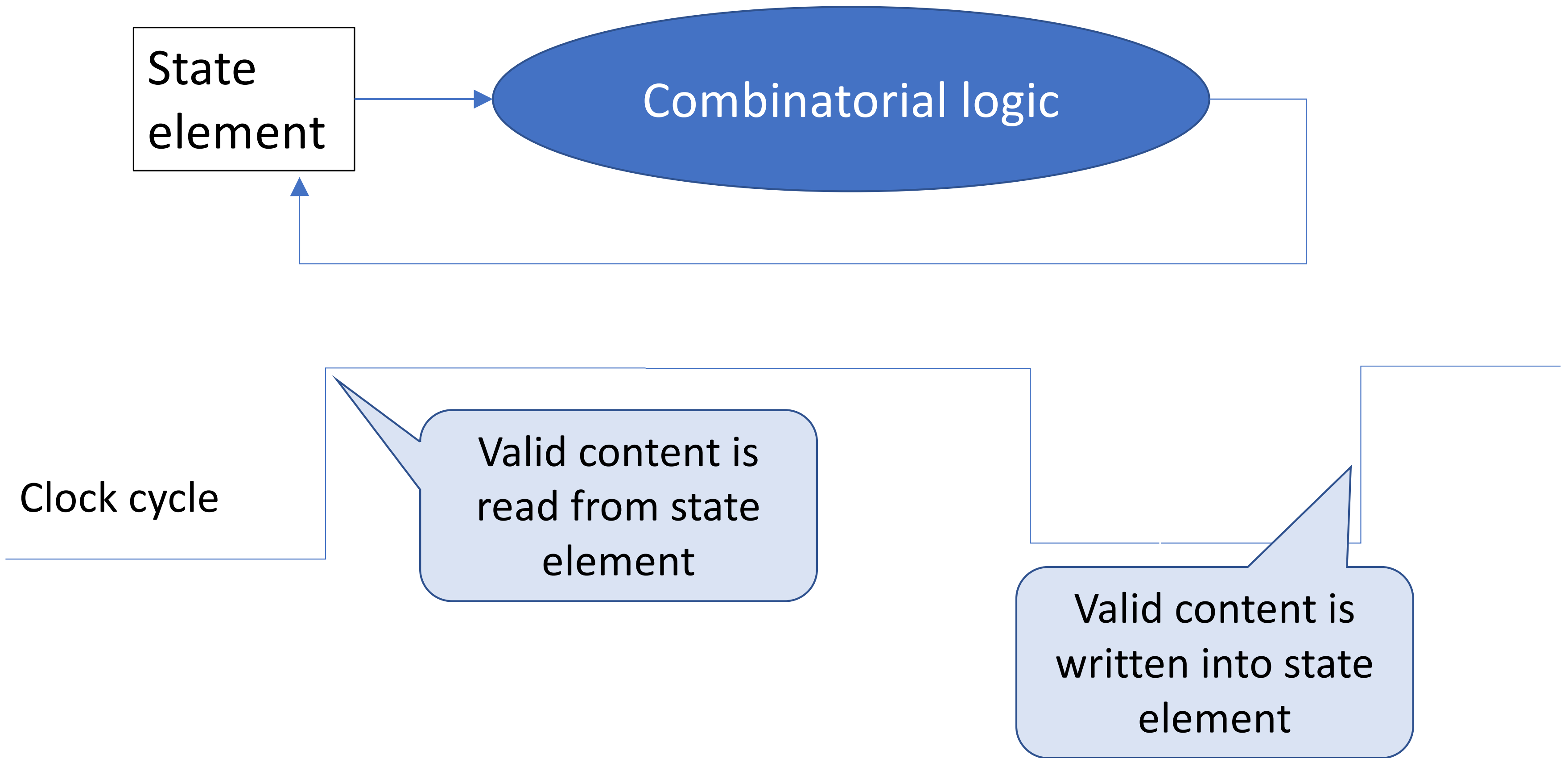


Clock frequence = 1/clock_period

Sequential logic require clocks

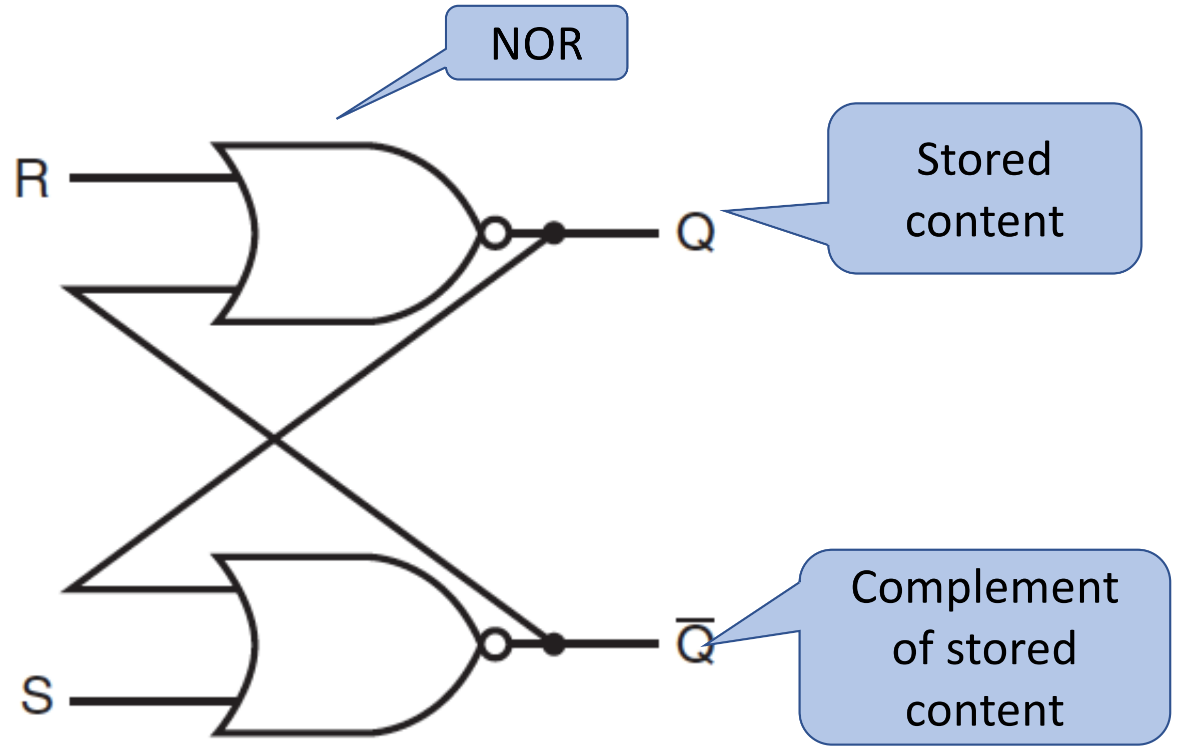

State element 1
Combinatorial logic
State element 2
Delay of CL must be less than one clock cycle
Clock cycle
Valid content is read from state element 1
Valid content is written into state element 2

Sequential logic require clocks


State element
Combinatorial logic
Clock cycle
Valid content is read from state element
Valid content is written into state element

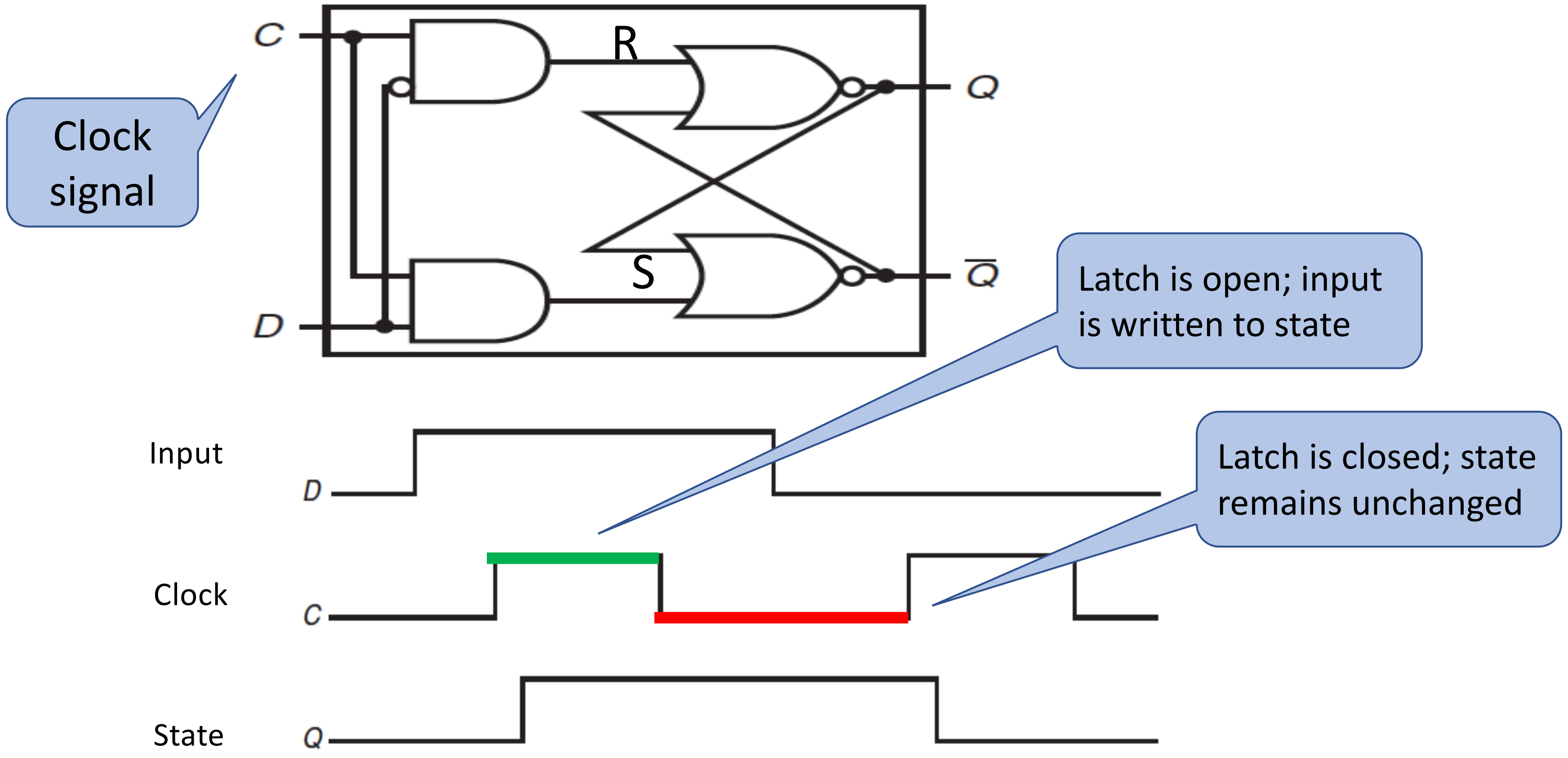
Memory (state) elements: unlocked S-R Latch

S (set)	R (reset)	
0	0	keeps existing value
1	0	Q=1
0	1	Q=0
1	1	Invalid



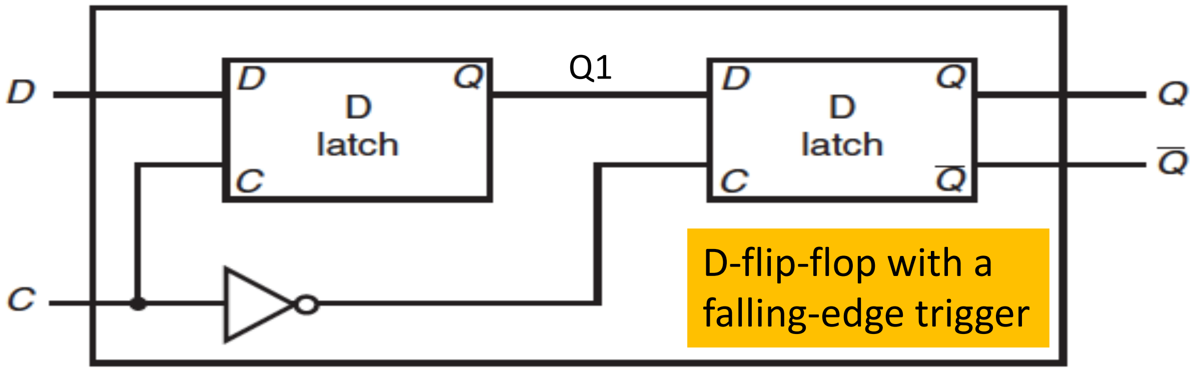
Memory element: clocked D latch

- D latch: state is changed as long as clock is asserted

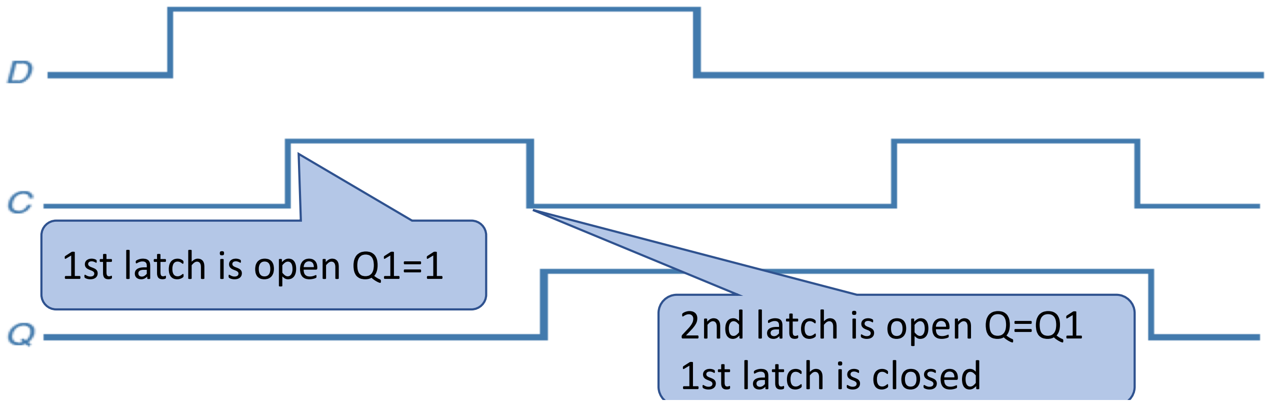

Clock signal
C
D
R
S
Q
$\overline{Q}$
Latch is open; input is written to state
Latch is closed; state remains unchanged
Input
D
Clock
C
State
Q

Memory element: Flip-flop

- Flip-flop: state is changed only on (rising or falling) clock edge

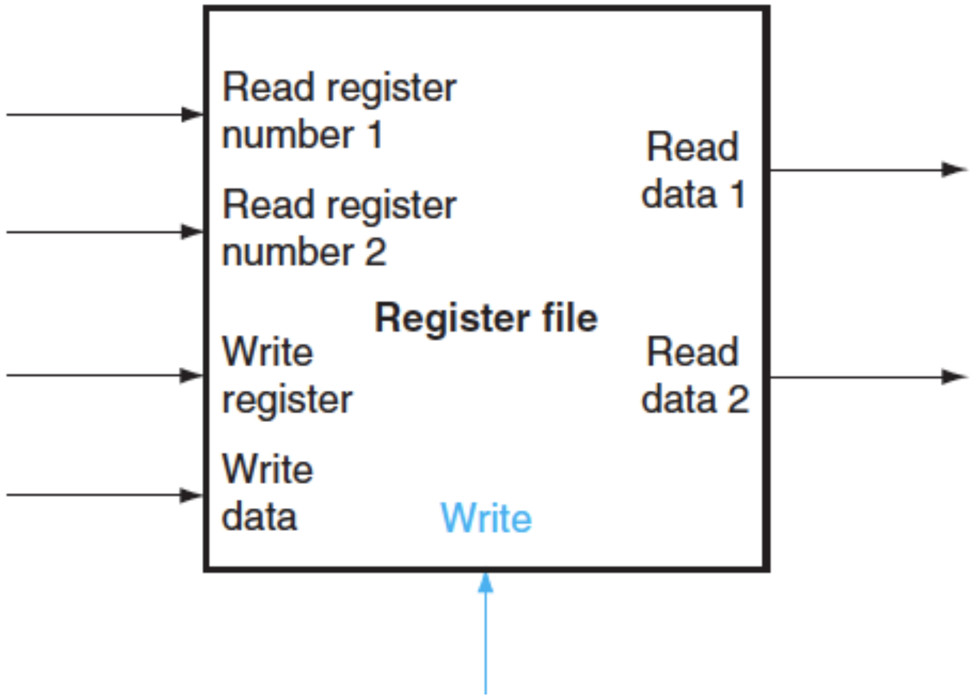


D-flip-flop with a falling-edge trigger

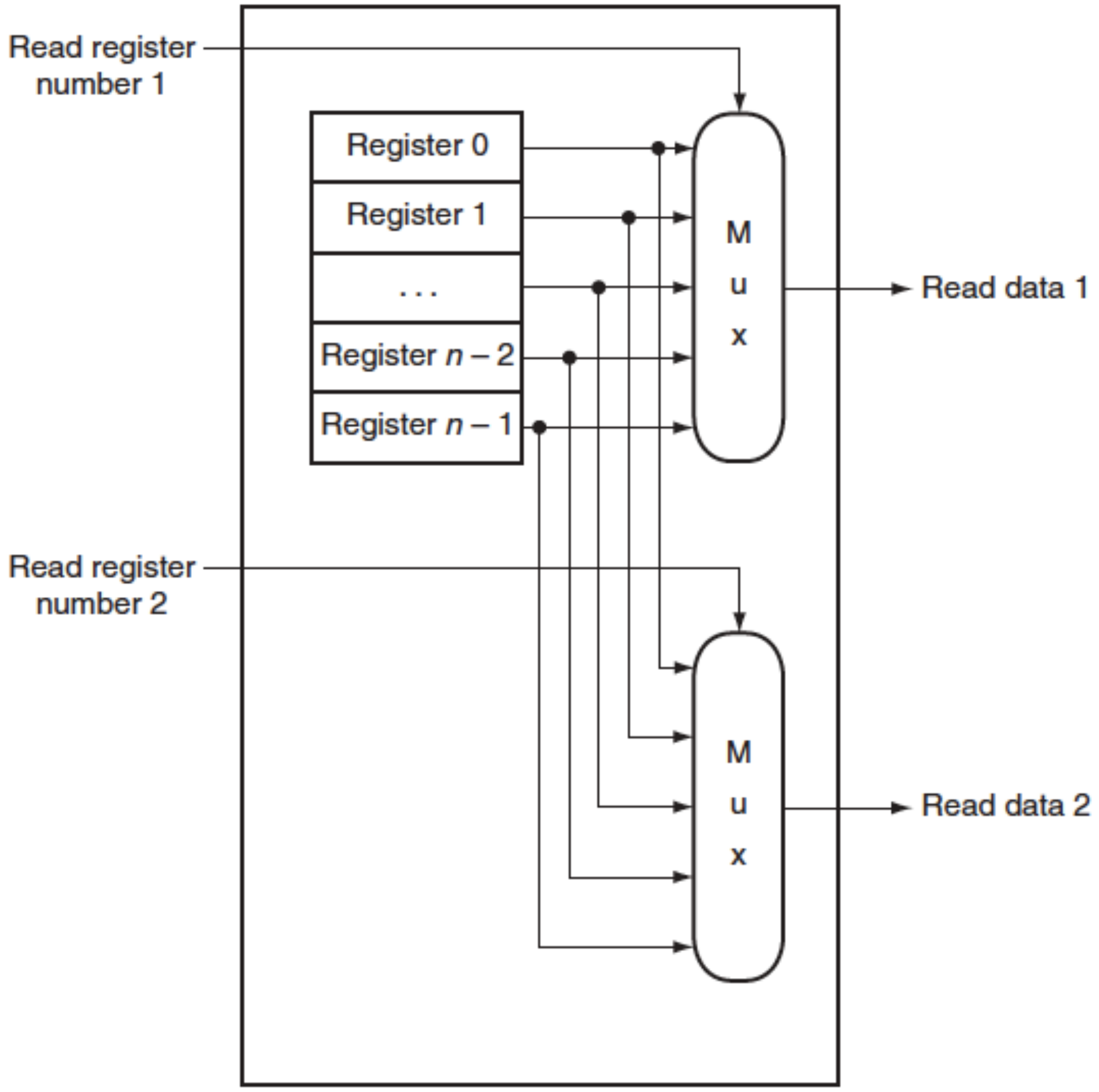


Memory element: Register file

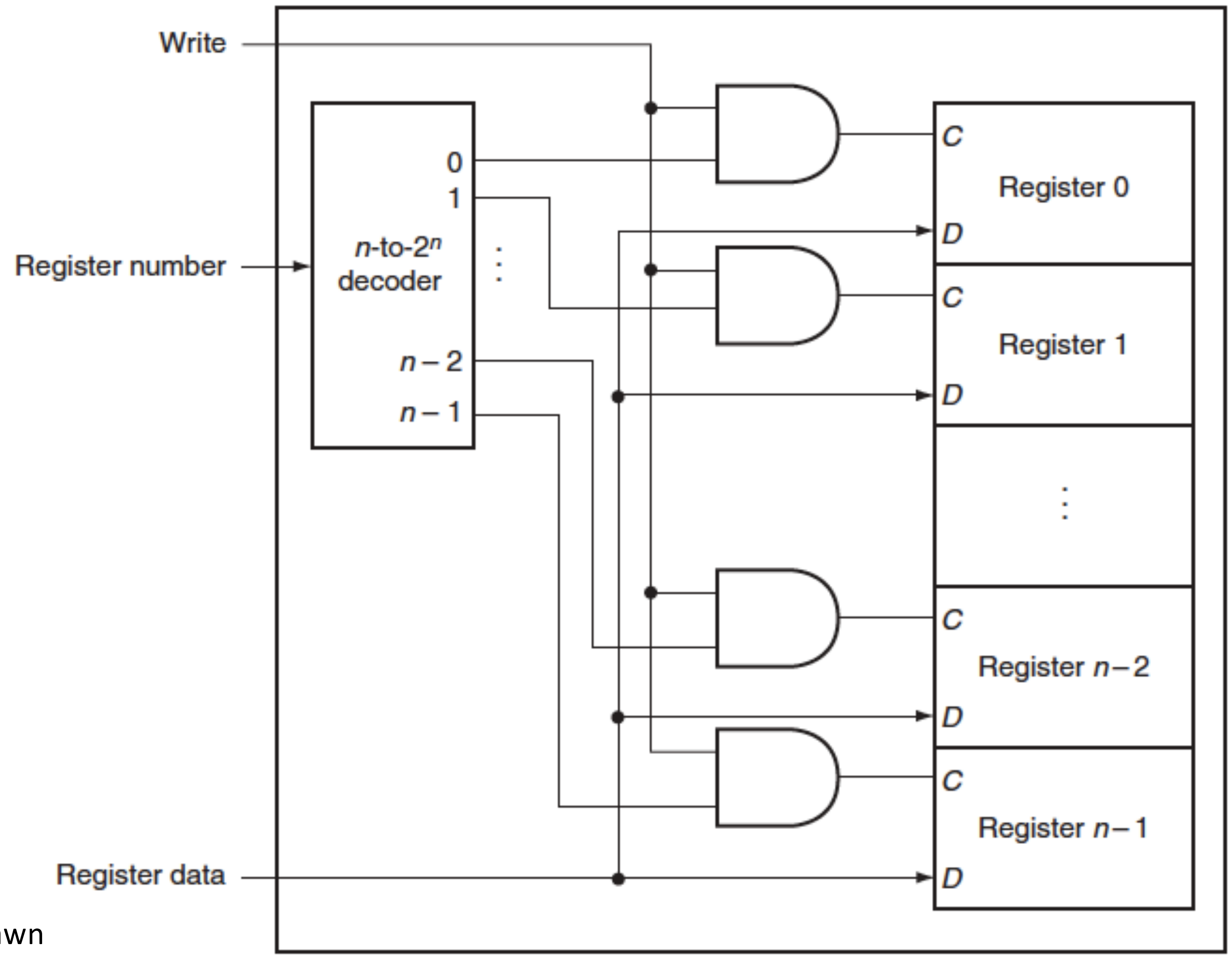
- Register file: a set of registers that can be read and written



Register file: Read



Register file: Write



Clock signal is assumed and not drawn

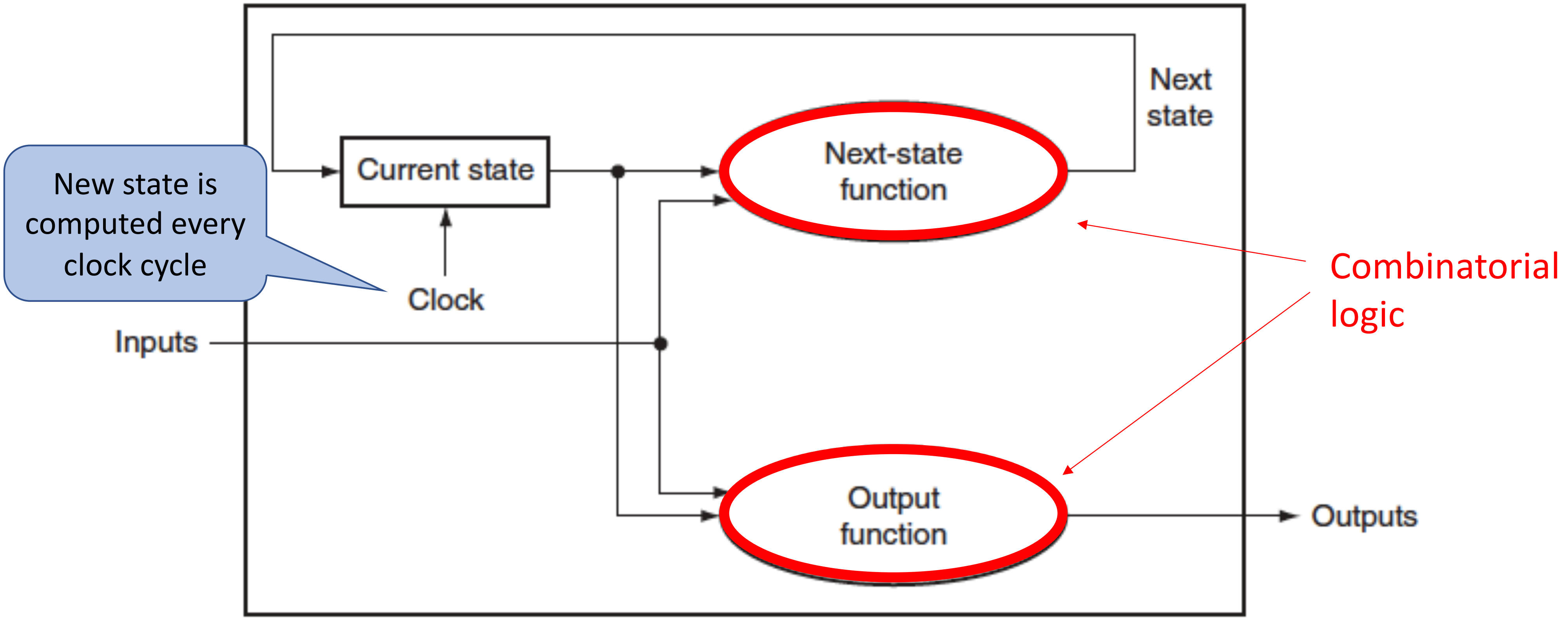
Register file

- What if the same register is read and written in the same clock cycle?
  - Return value written in an earlier cycle
  - Write of new value occurs on the clock edge (at the end of the current cycle)
- Some register file can read value currently being written
  - Requires additional logic in the register file

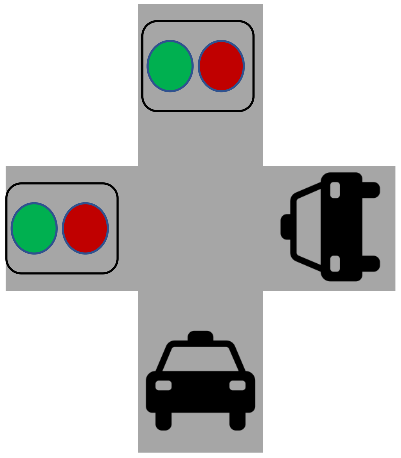
Finite State Machine

- Combinatorial logic → truth table
- Sequential logic → F(inite) S(tate) M(achine)
  - Input and current state determine next state and outputs

Finite State Machine



FSM example: traffic light control

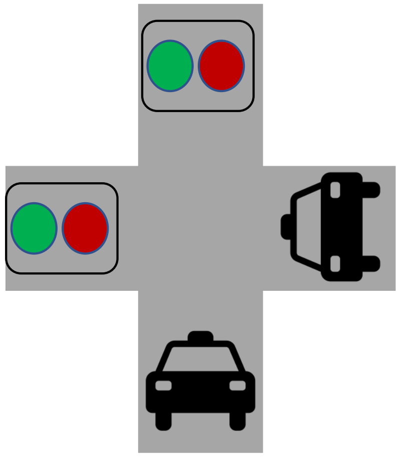


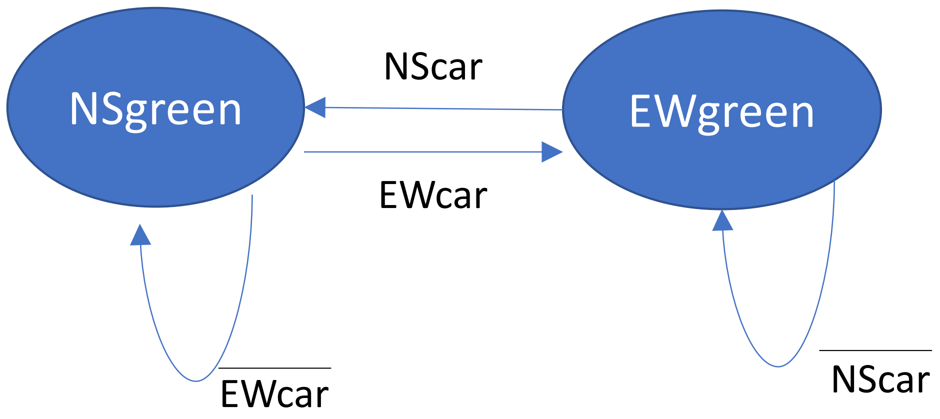
State:
NSgreen: traffic light is green in N-S (red in E-W)
EWgreen: traffic light is green in E-W (red in N-S)

Inputs:
NScar: car detected in N-S
EWcar: car detected in E-W

Outputs:
NSlite: 1 if state=NSgreen
EWlite: 1 if state=EWgreen

FSM example: traffic light control



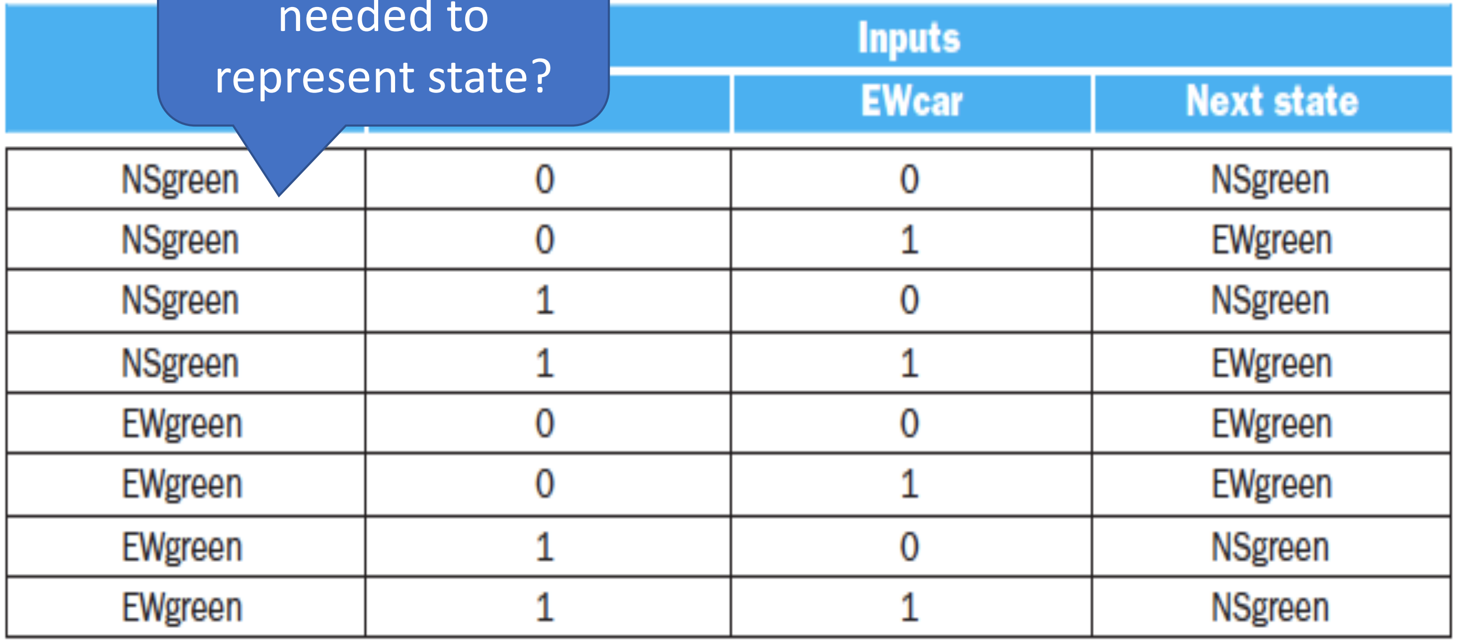


Clock cycles once every 30 seconds

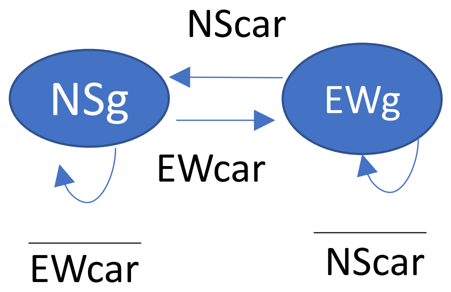
FSM example: traffic light

- FSM is determined by NextState function and Output function

How many bits needed to represent state?



	Inputs		
		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



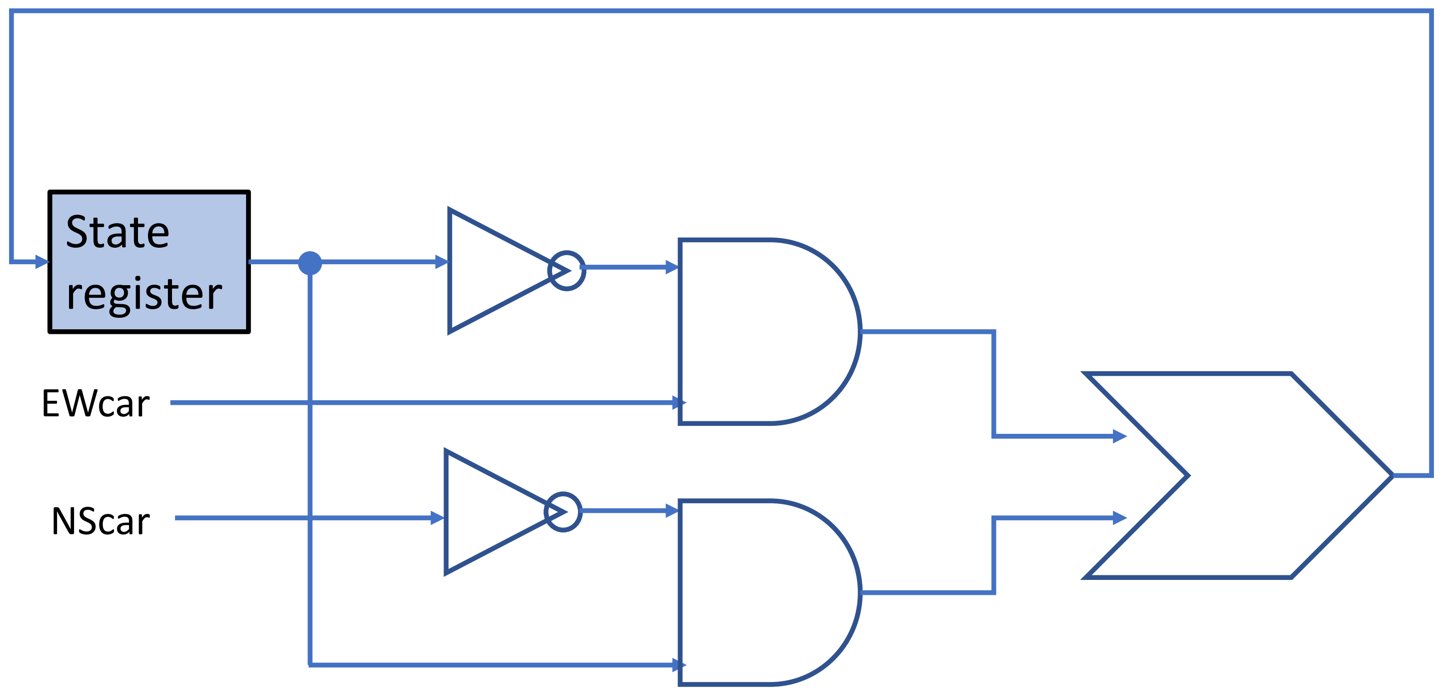
FSM example: traffic light

- FSM is determined by NextState function and Output function

Current State	Inputs		
	NScar	EWcar	Next state
0 (Nsgreen)	0	0	0 (Nsgreen)
0 (Nsgreen)	0	1	1 (Ewgreen)
0 (Nsgreen)	1	0	0 (Nsgreen)
0 (Nsgreen)	1	1	1 (Ewgreen)
1 (Ewgreen)	0	0	1 (Ewgreen)
1 (Ewgreen)	0	1	1 (Ewgreen)
1 (Ewgreen)	1	0	0 (Nsgreen)
1 (Ewgreen)	1	1	0 (Nsgreen)

$$\begin{aligned}
Next &= \overline{Curr} \cdot \overline{NScar} \cdot EWcar \\
&\quad + \overline{Curr} \cdot NScar \cdot EWcar \\
&\quad + Curr \cdot \overline{NScar} \cdot \overline{EWcar} \\
&\quad + Curr \cdot \overline{NScar} \cdot EWCar \\
&= \overline{Curr} \cdot EWcar + Curr \cdot \overline{NScar}
\end{aligned}$$

FSM traffic light: next state function

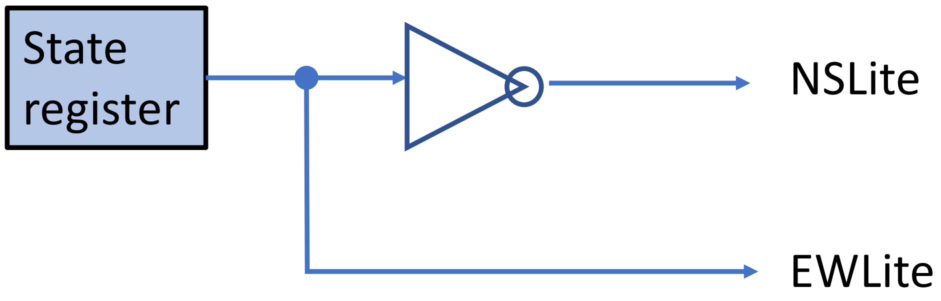


FSM traffic light: output function

	Outputs	
	NSlite	EWlite
0 NSgreen	1	0
1 EWgreen	0	1

NSLite = $\overline{Curr}$
EWLite = $Curr$

FSM traffic light: output function


State register
NSLite
EWLite