Sequential Logic

Jinyang Li

What we've learnt so far

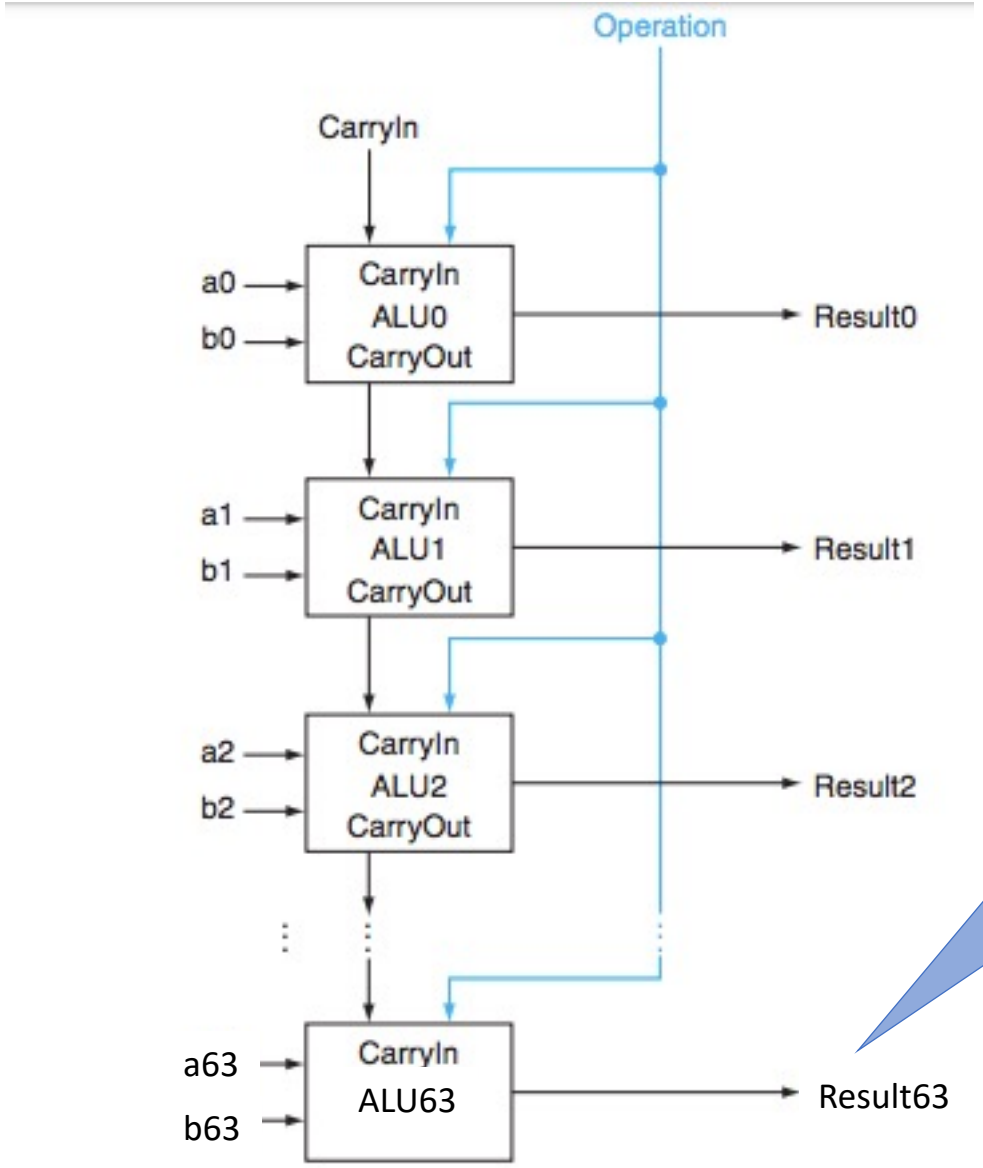
- Combinatorial logic
  - E.g. Multiplexors (Mux), Decoders
  - Two ways of building a CL
    - Truth table → sum of products circuits
    - ROM
- ALU
  - Compute all operations (+, OR, AND, NOR), multiplexer picks the result
  - Building a 64-bit adder
    - Ripple carry chains together 1-bit adders
    - Carry lookahead

Not yet covered

Today’s lesson plan

- Ripple carry vs. Carry lookahead
- Sequential circuit: Memory (state) elements
- Sequential circuit: Finite State Machine

Downside of ripple carry?



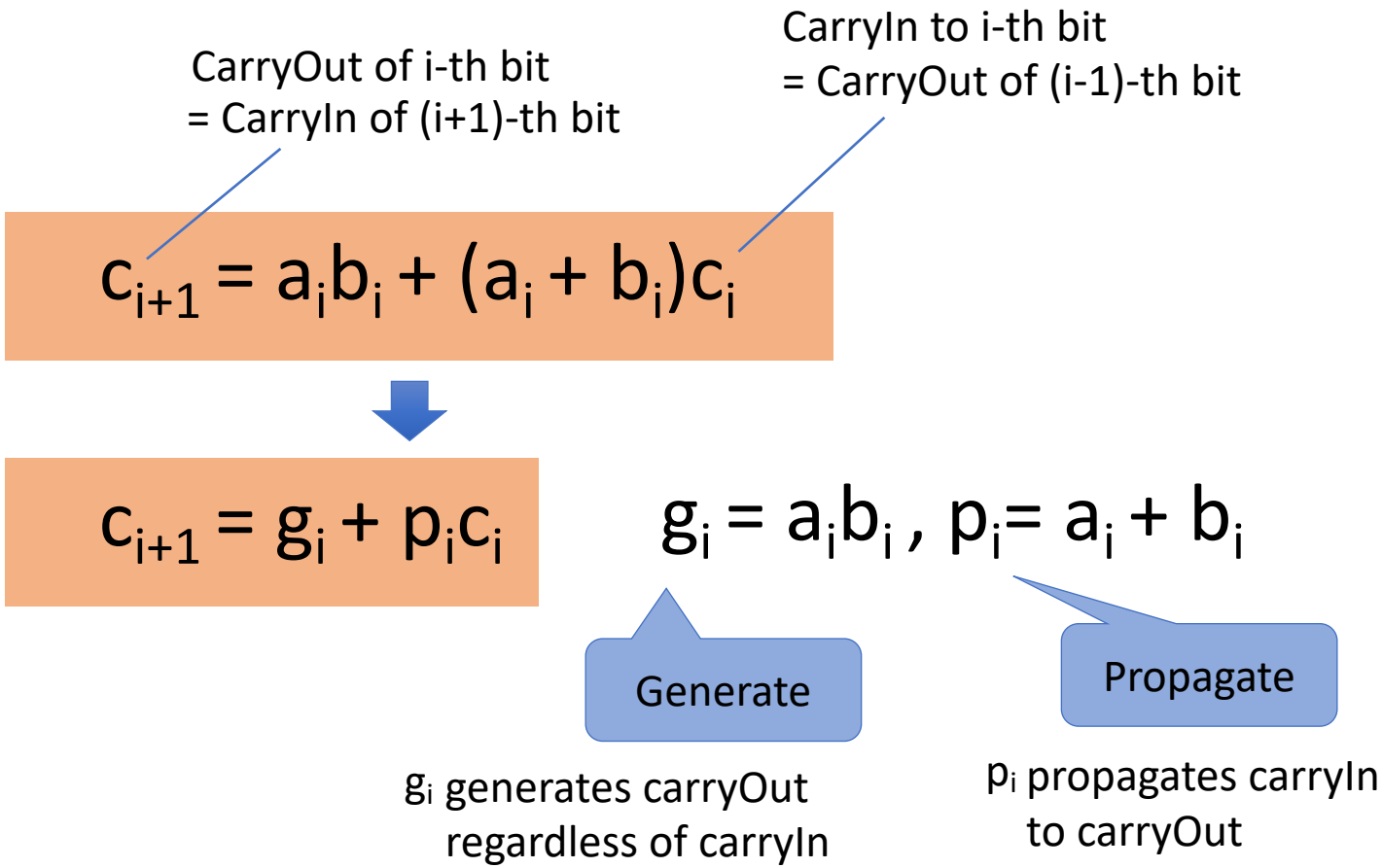
Must wait for sequential evaluation of all 64 1-bit adders

In search of a faster adder

- Ripple carry:
  - Delay: 64*1BitAdderGateDelay, Gate count: 64*c
- Brute-force (truth table->PLA)
  - Delay: 2, Gate count: $O(2^{64+64})$
- Clever designs in between?
- Idea #1: (Carry lookahead) compute multiple carry-bits at a time

Faster adder: carry lookahead

- Idea #1: (Carry lookahead) compute multiple carry-bits at a time



Faster adder: carry lookahead

- Idea #1: (Carry lookahead) compute multiple carry-bits at a time

Computing all carry-bits of a 4-bit adder:

$$c1 = g0 + (p0 \cdot c0)$$

$$c2 = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$

$$c3 = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$

Delay? 3

4-bit ripple carry
delay: 2 * 4

Faster adder: carry lookahead

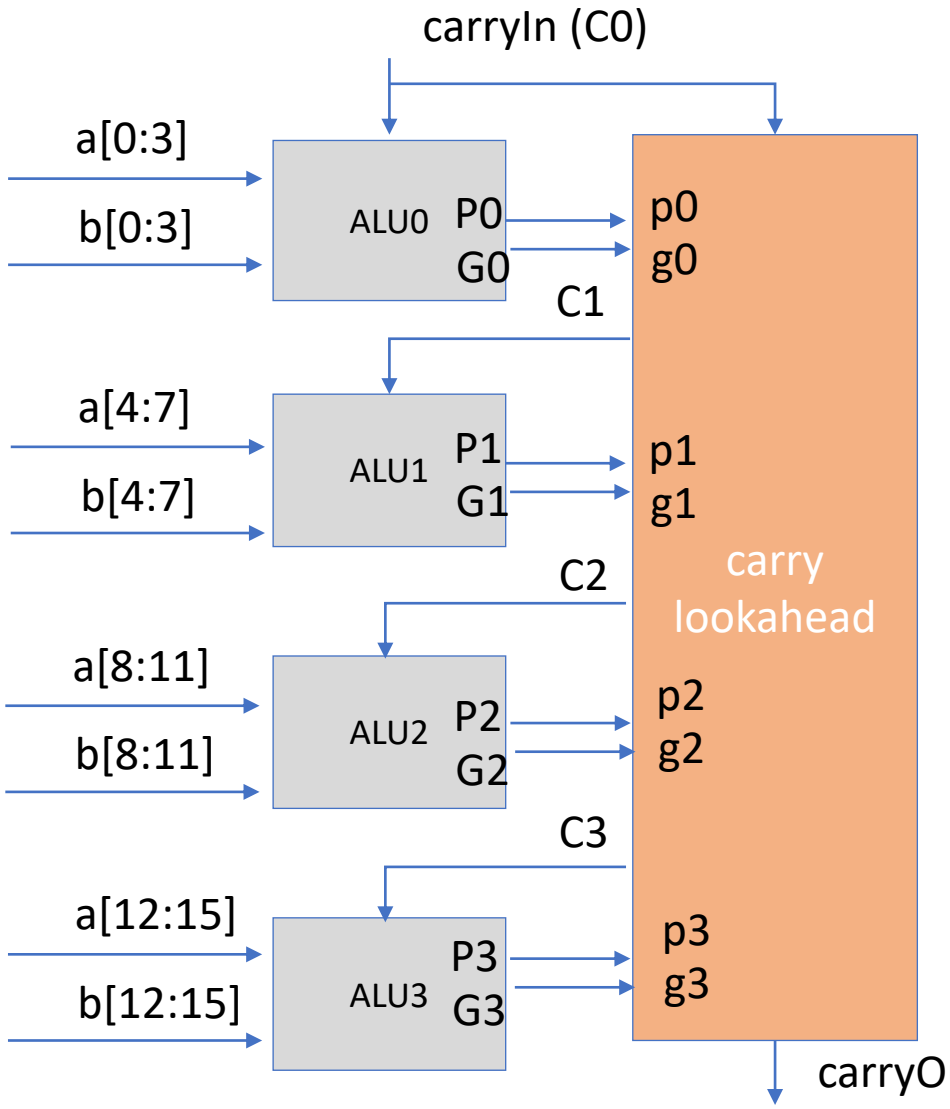
- Idea #1: (Carry lookahead) compute multiple carry-bits at a time

Computing all result bits in a 4-bit adder:

$$s_i = \overline{c_i} \cdot a_i \cdot b_i + c_i \cdot \overline{a_i} \cdot b_i + c_i \cdot a_i \cdot \overline{b_i} \ , \ i = 0,..,3$$

Faster adder: carry lookahead

- Build a 16-bit adder with carry-ahead 4-bit adders



$P0 = p3 \cdot p2 \cdot p1 \cdot p0$

$P1 = p7 \cdot p6 \cdot p5 \cdot p4$

$P2 = p11 \cdot p10 \cdot p9 \cdot p8$

$P3 = p15 \cdot p14 \cdot p13 \cdot p12$

$G0 = g3 + p3 \cdot g2 + p3 \cdot p2 \cdot g1 + p3 \cdot p2 \cdot p1 \cdot g0$

$G1 = g7 + p7 \cdot g6 + p7 \cdot p6 \cdot g5 + p7 \cdot p6 \cdot p5 \cdot g4$

$G2 = g11 + p11 \cdot g10 + p11 \cdot p10 \cdot g9 + p11 \cdot p10 \cdot p9 \cdot g8$

$G3 = g15 + p15 \cdot g14 + p15 \cdot p14 \cdot g13 + p15 \cdot p14 \cdot p13 \cdot g12$

$C1 = G0 + P0 \cdot C0$

$C2 = G1 + P1 \cdot G0 + P1 \cdot P0 \cdot C0$

$C3 = G2 + P2 \cdot G1 + P2 \cdot P1 \cdot G0 + P2 \cdot P1 \cdot P0 \cdot C0$

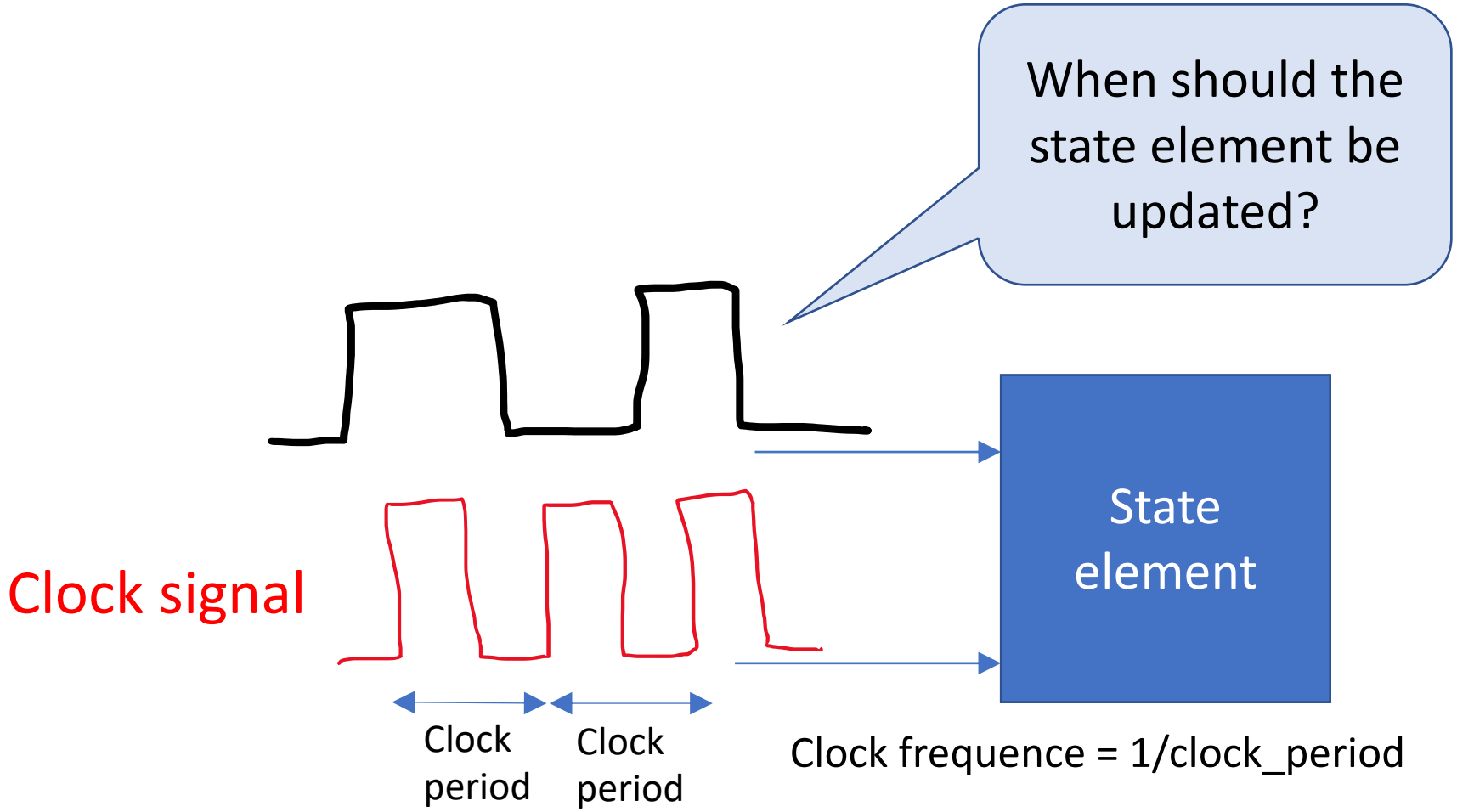
$C3 = G3 + P3 \cdot G2 + P3 \cdot P2 \cdot G1 + P3 \cdot P2 \cdot P1 \cdot G0 + P3 \cdot P2 \cdot P1 \cdot P0 \cdot C0$

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)

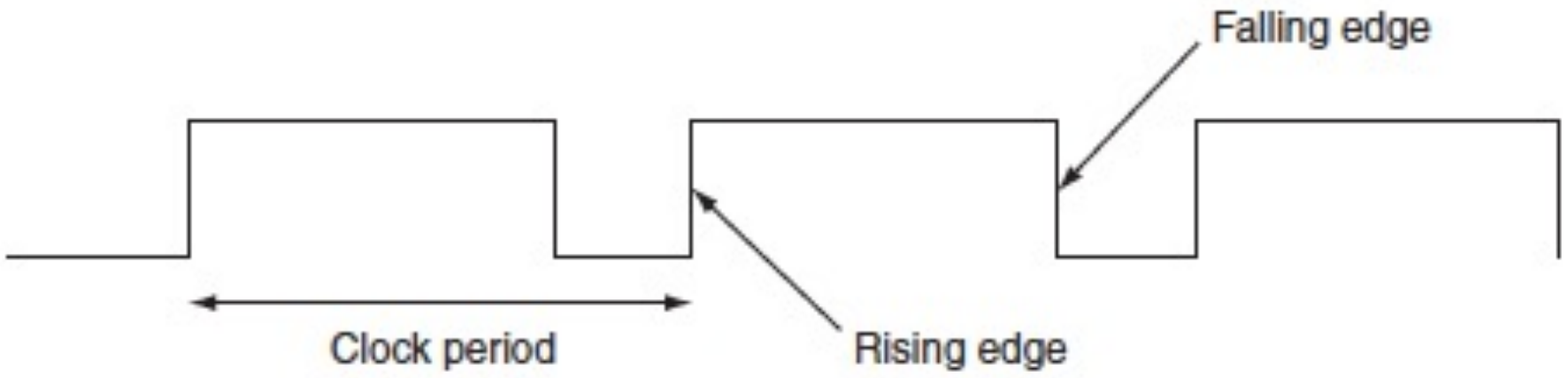
Today's lesson

Sequential logic requires clock

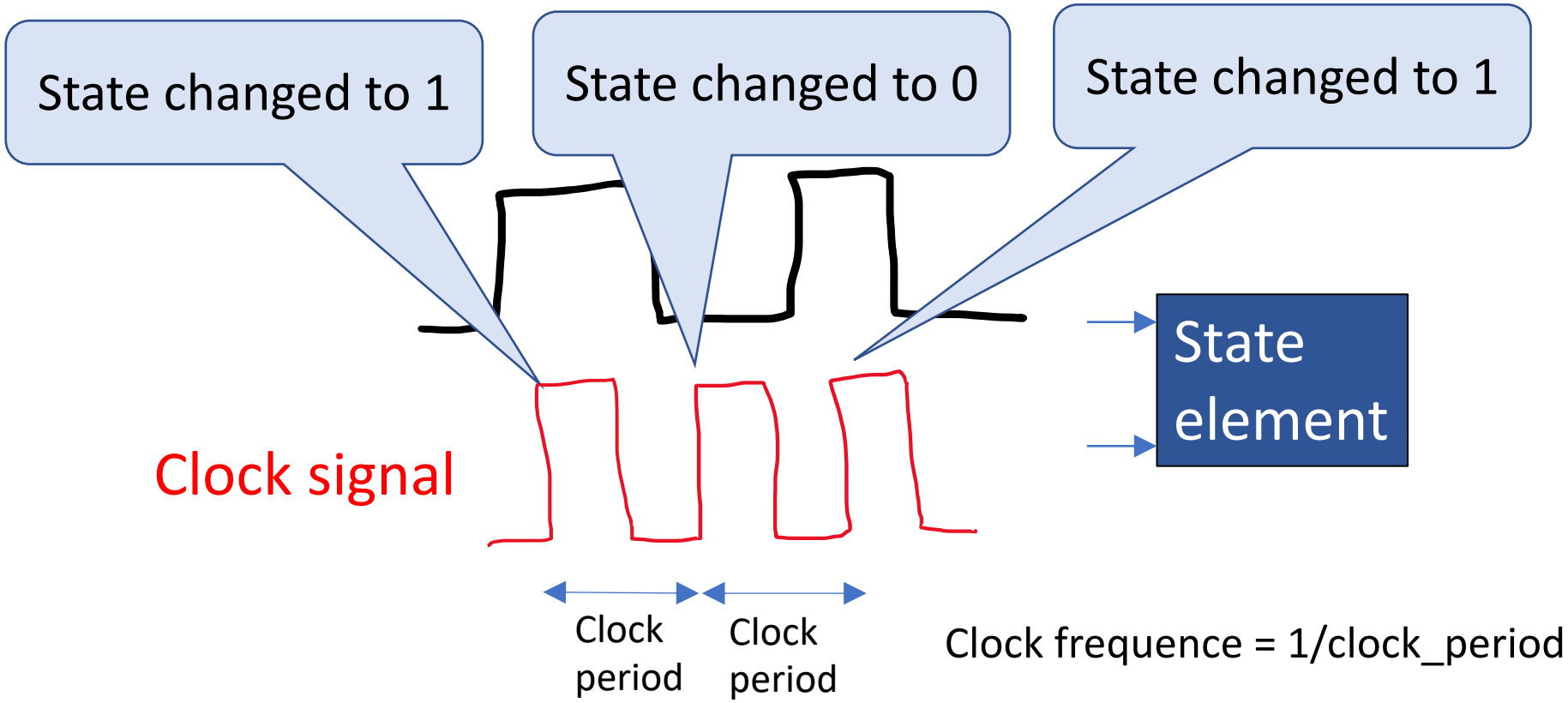


Clocks

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

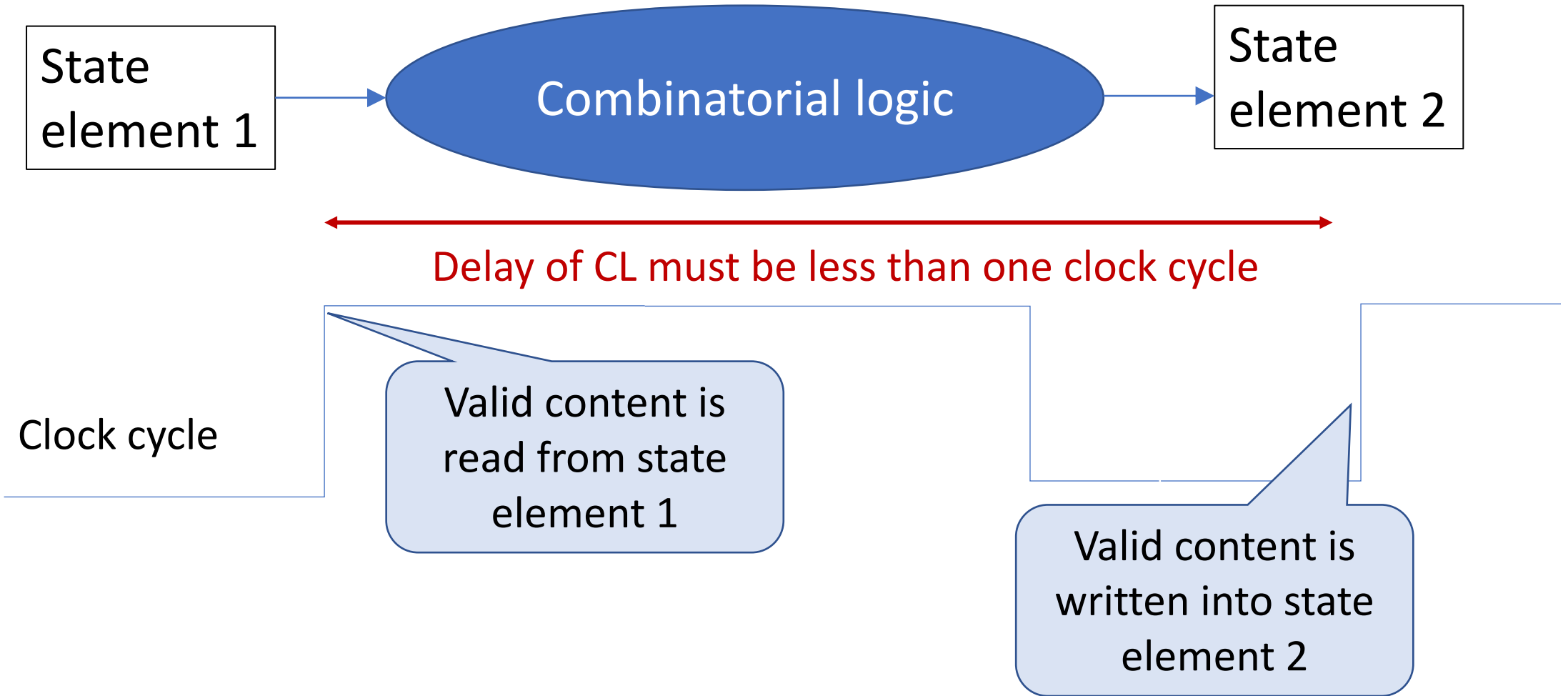


Sequential logic requires clock

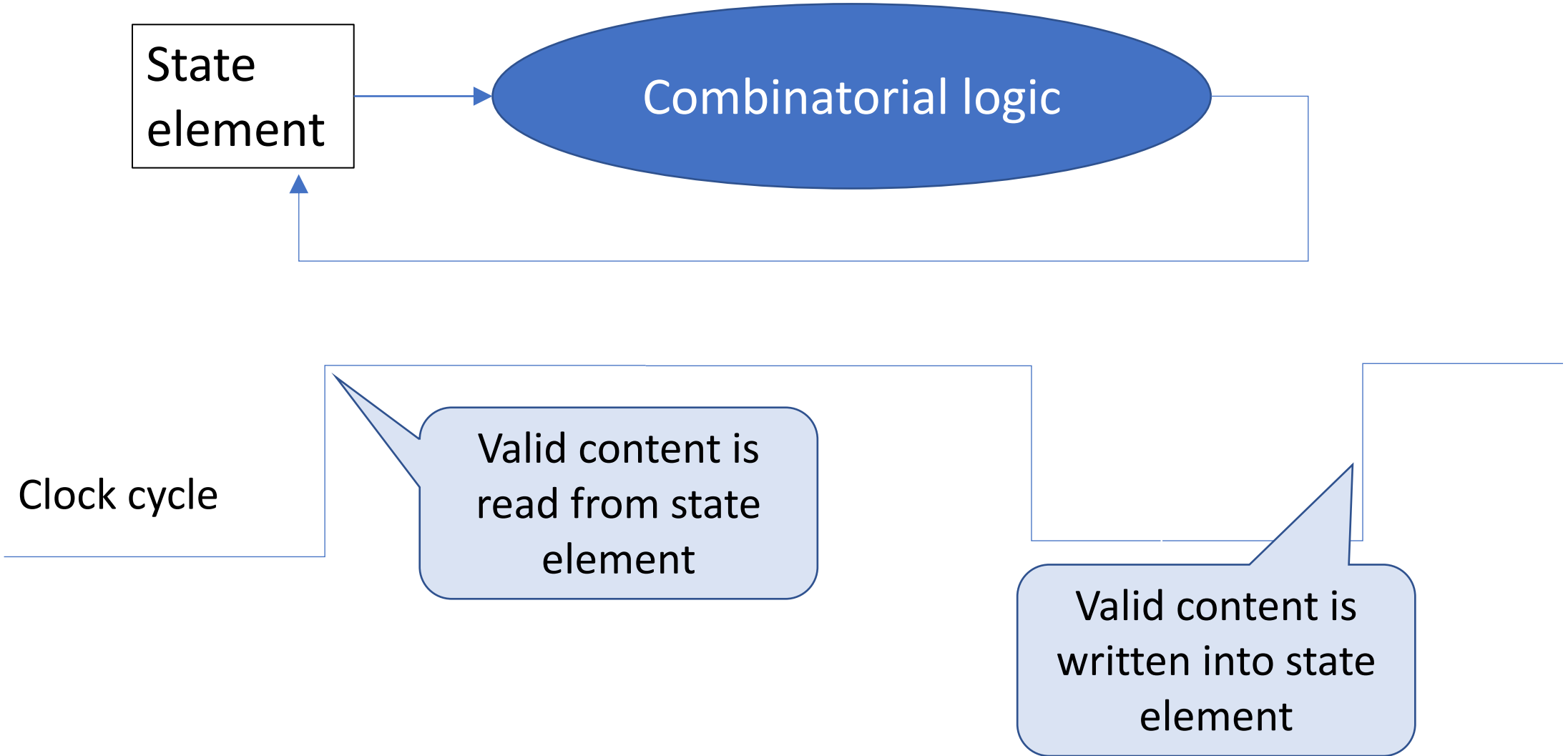


Clock frequence = 1/clock_period

Sequential logic requires clock

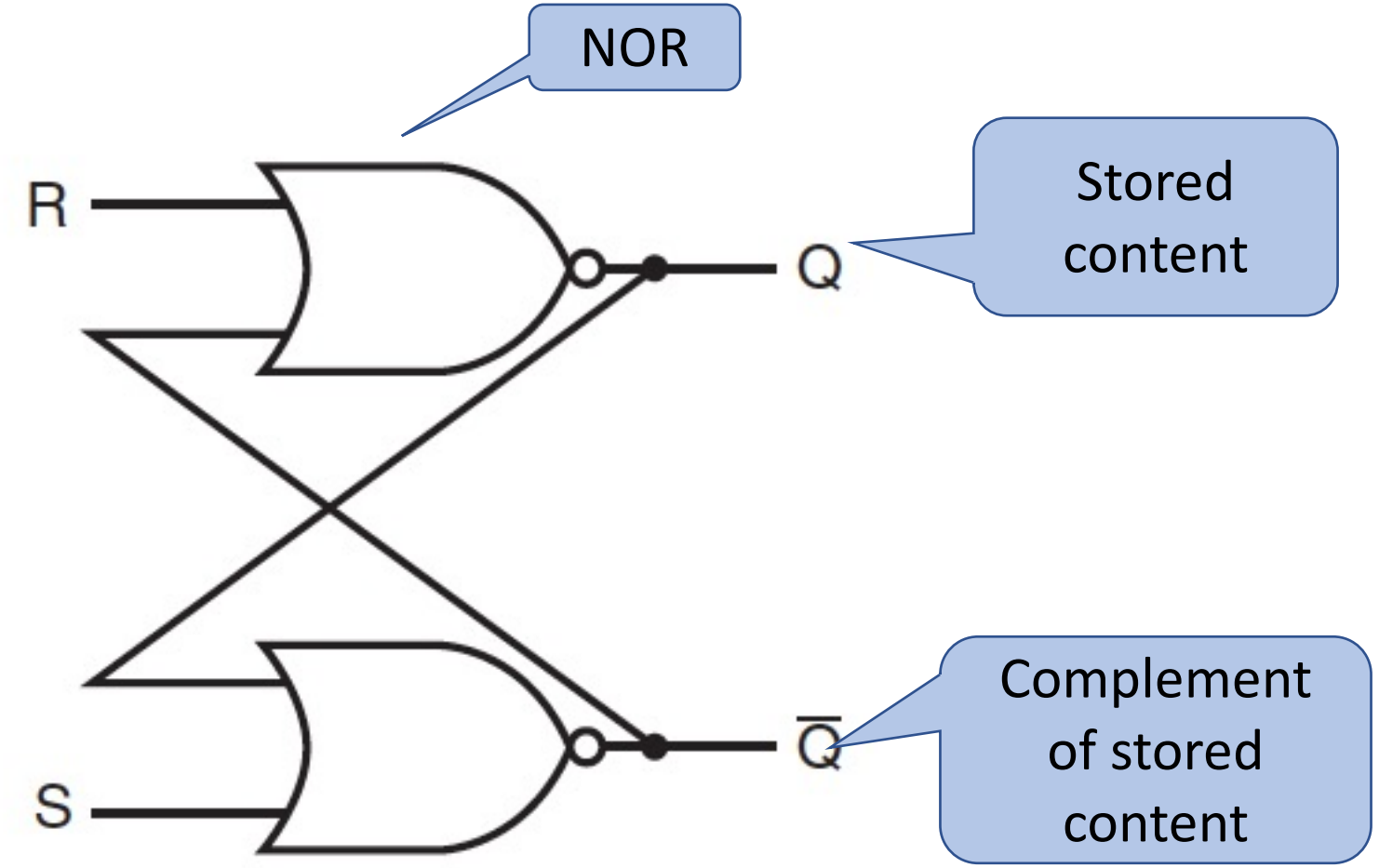


Sequential logic requires clock



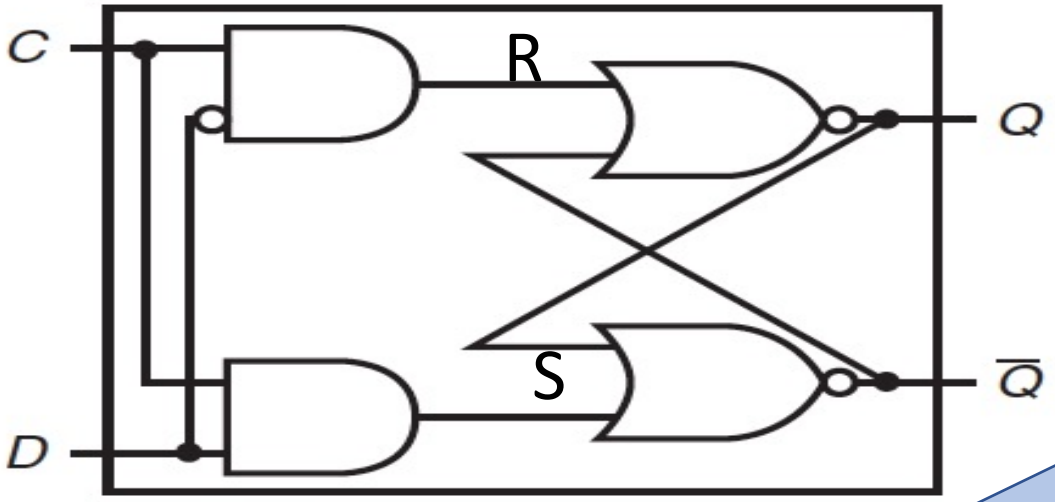
Memory (state) elements: unlocked S-R Latch

S (set)	R (reset)	
0	0	
1	0	
0	1	
1	1	



Memory element: clocked D latch

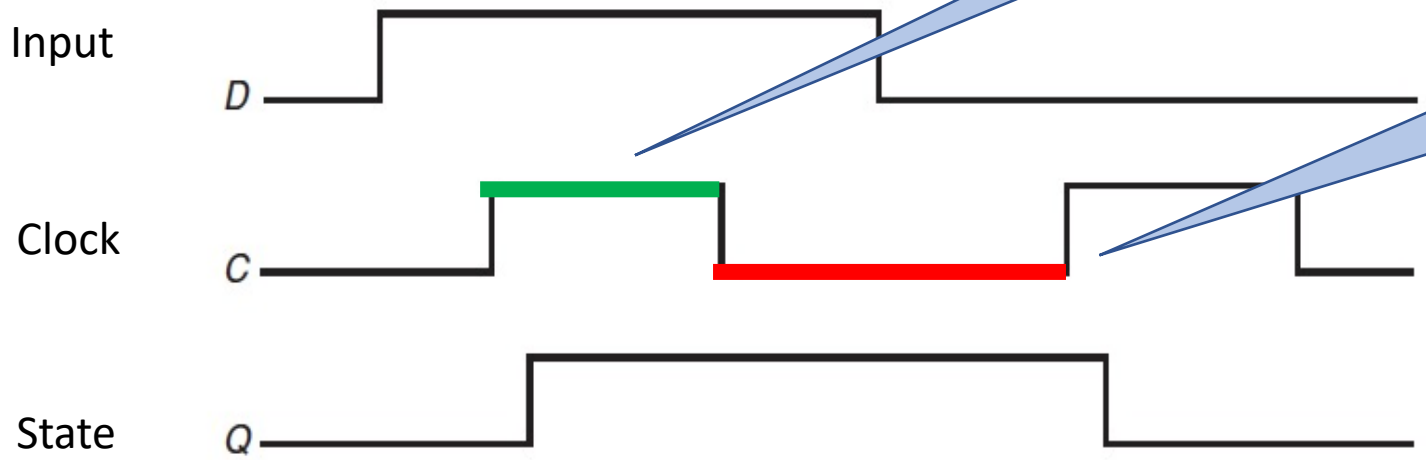
- D latch: state is changed as long as clock is asserted



Clock signal

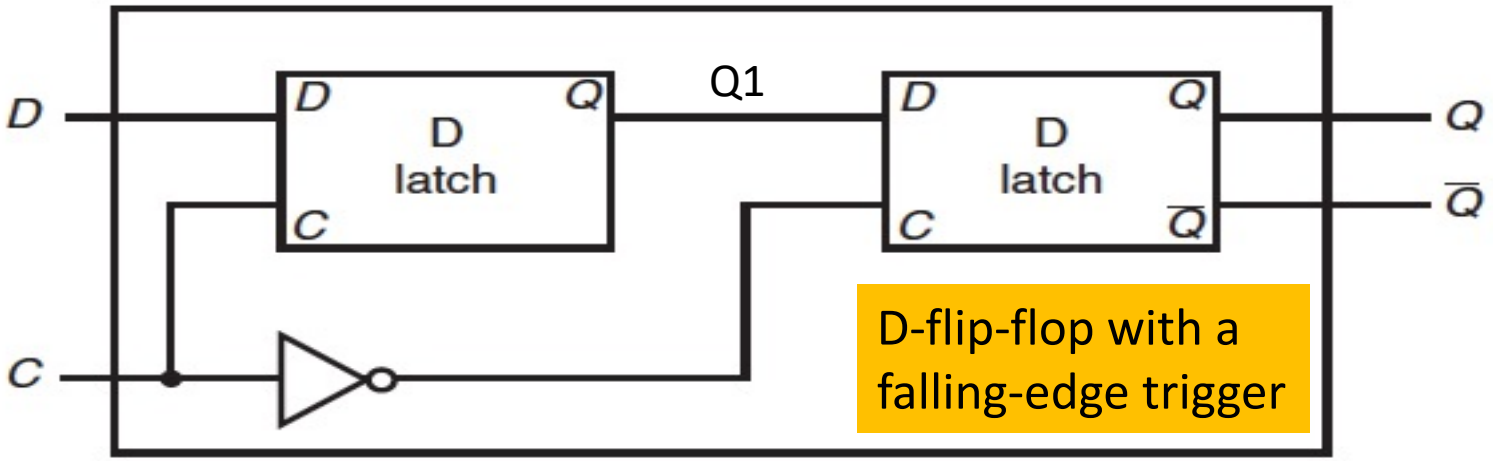
Latch is open; input is written to state

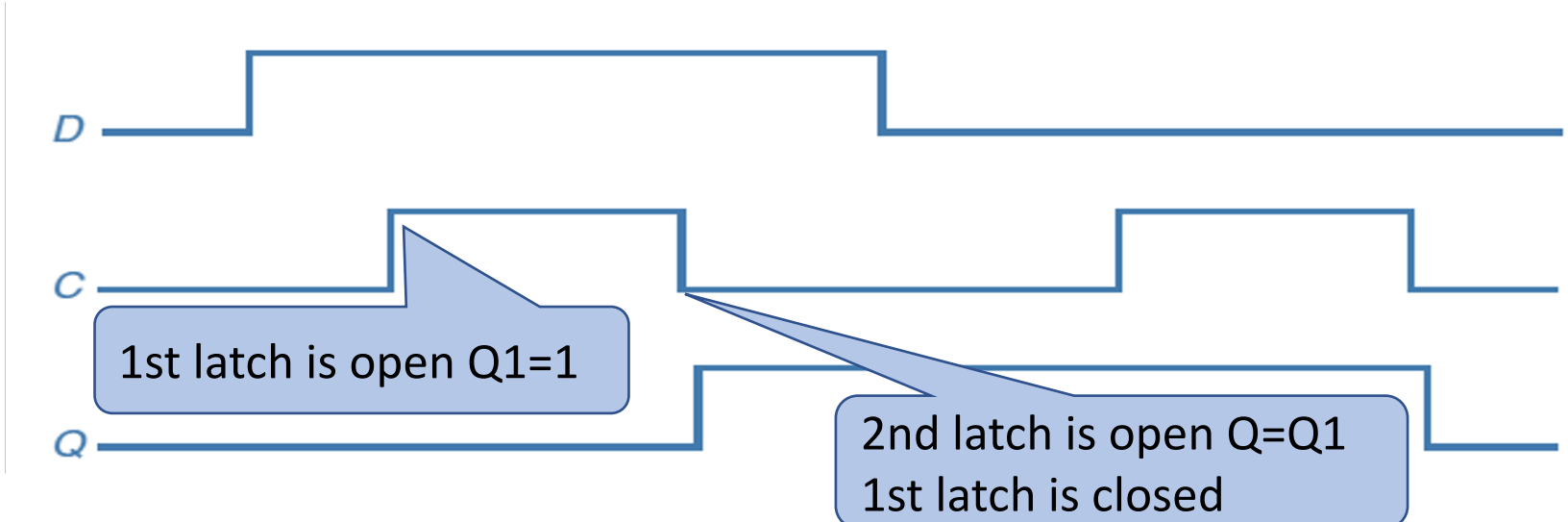
Latch is closed; state remains unchanged



Memory element: Flip-flop

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

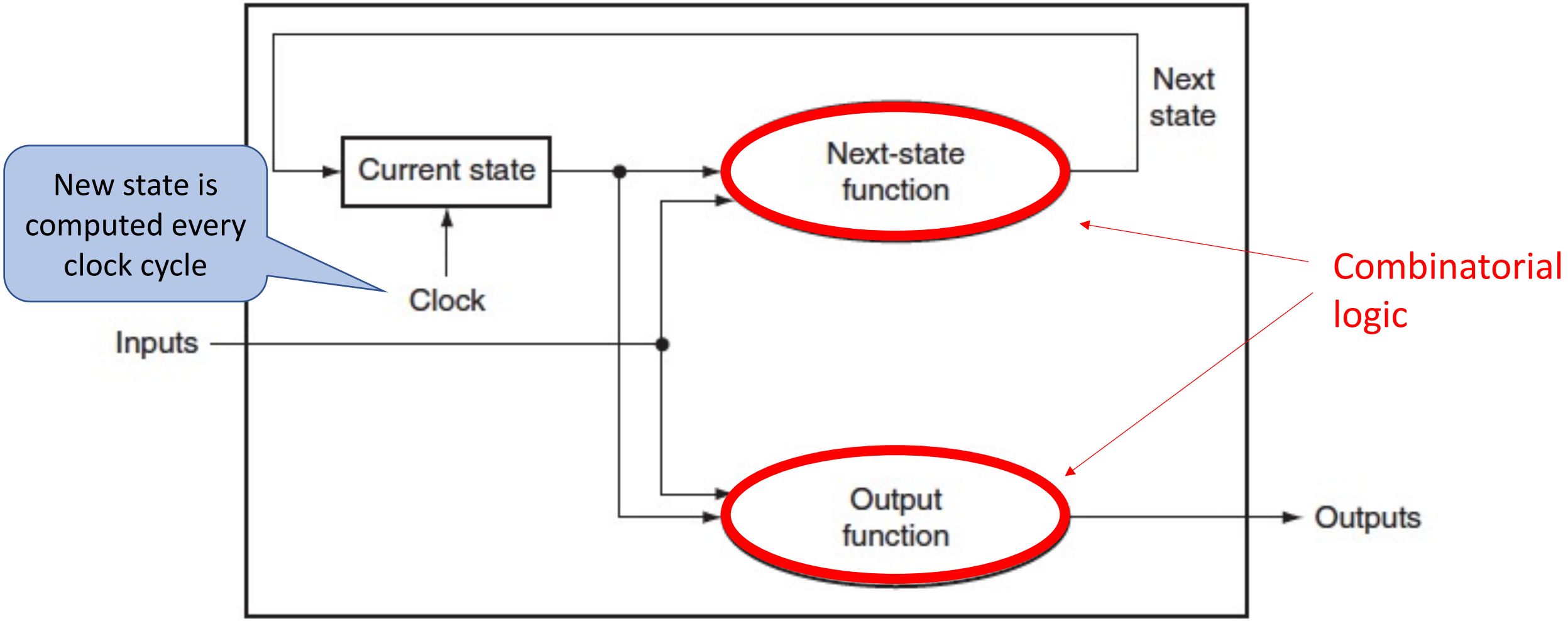




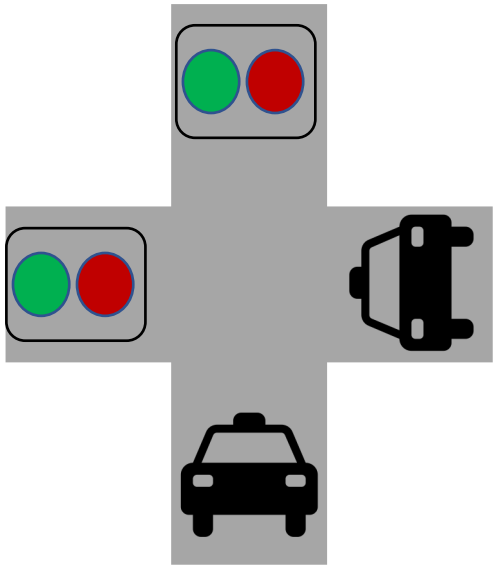
Finite State Machine

- 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

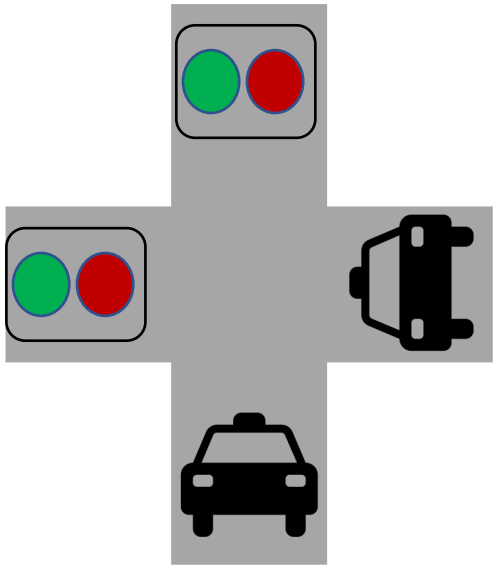
EWcar: car detected in E-W

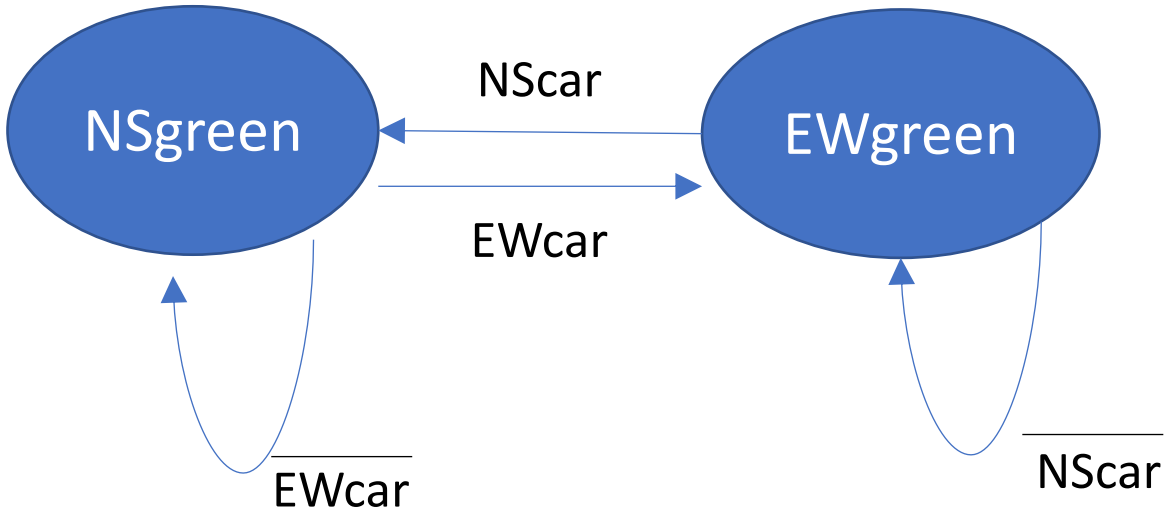
Outputs:

NSlite: 1 if state=NSgreen

EWlite: 1 if state=EWgreen

FSM example: traffic light control





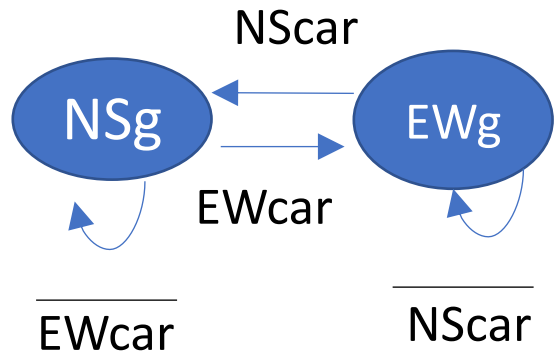
Clock cycles once every 30 seconds

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

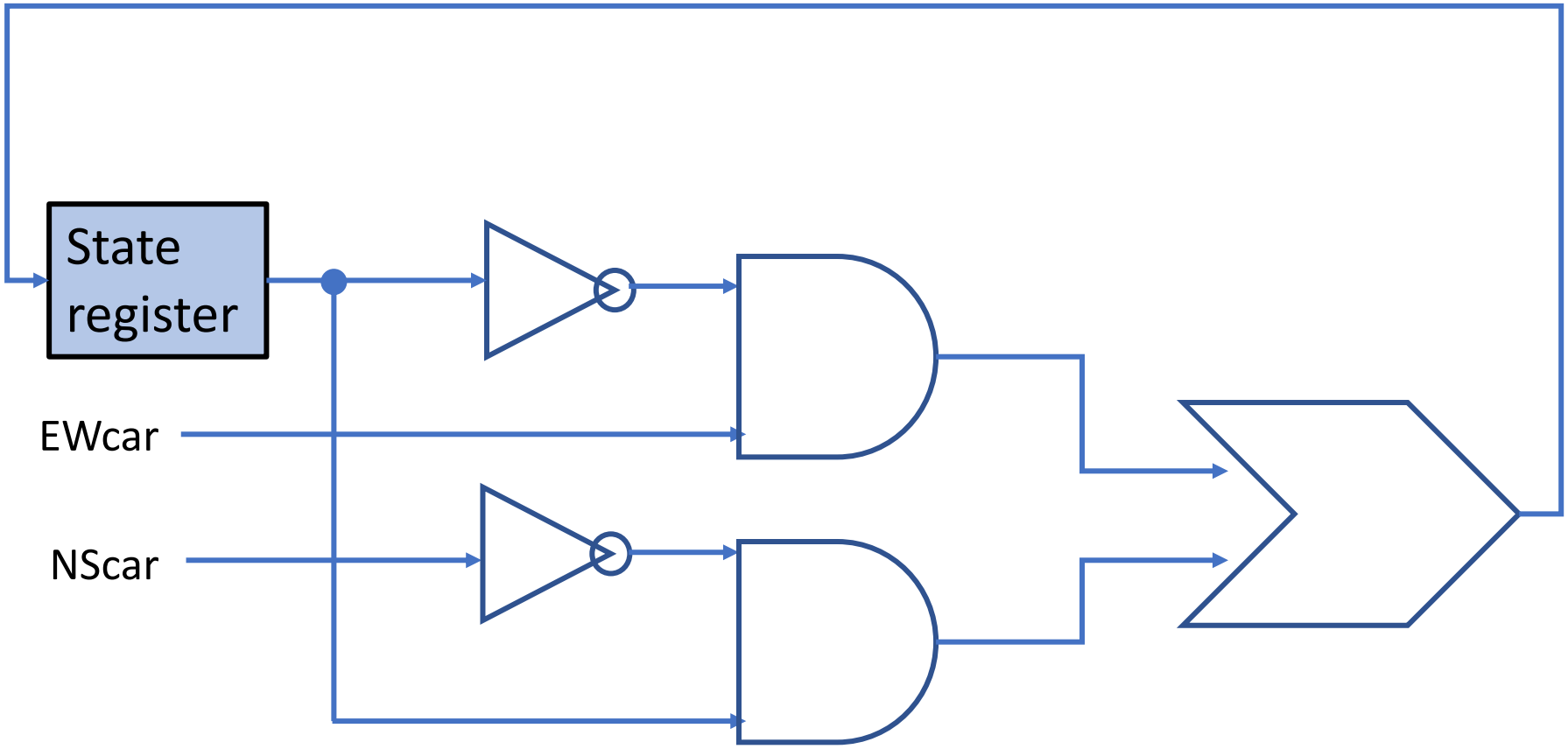
- FSM is determined by NextState function and Output function

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)

$$Next = \overline{Curr} \cdot \overline{NScar} \cdot EWcar + \overline{Curr} \cdot NScar \cdot EWcar + Curr \cdot \overline{NScar} \cdot \overline{EWcar} + Curr \cdot \overline{NScar} \cdot EWCar$$

FSM traffic light: next state function

$$Next = \overline{Curr} \cdot EWcar + Curr \cdot \overline{NScar}$$



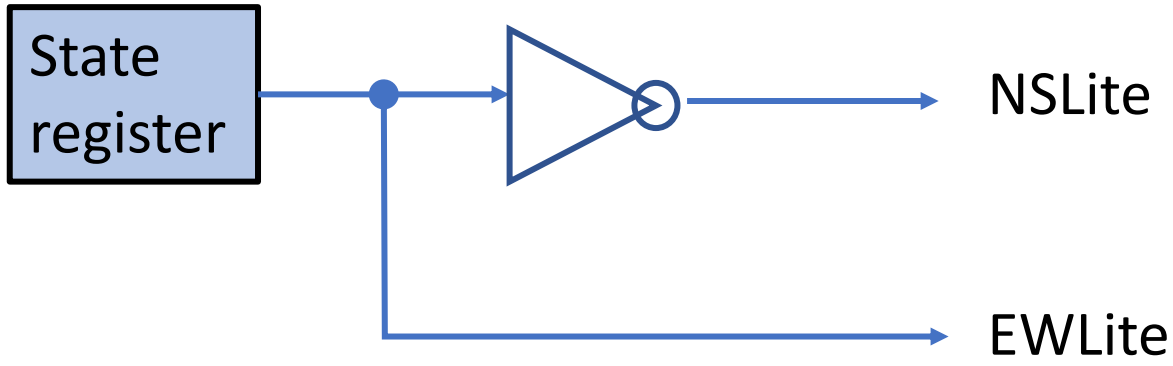
FSM traffic light: output function

	Outputs	
	NSlite	EWlite
0 NSgreen	1	0
1 EWgreen	0	1

NSLite $= \overline{Curr}$

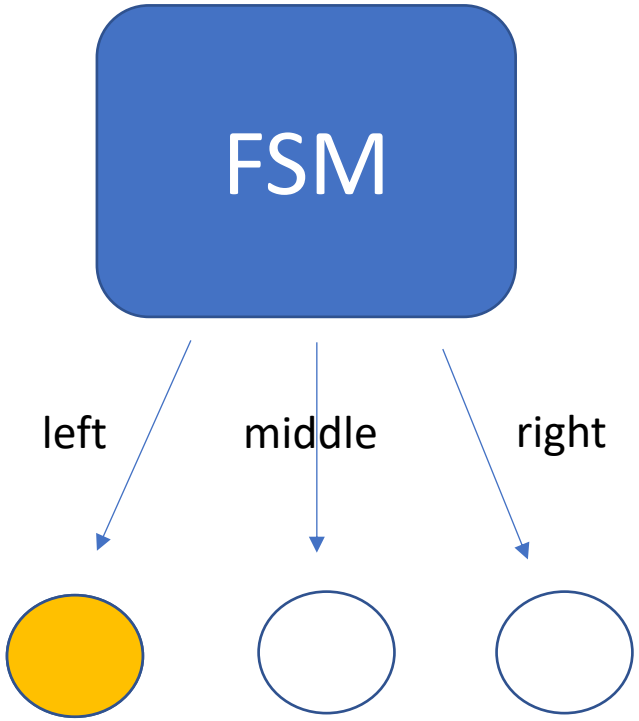
EWLite = $Curr$

FSM traffic light: output function



Another FSM example: Electronic eye

State transition diagram?

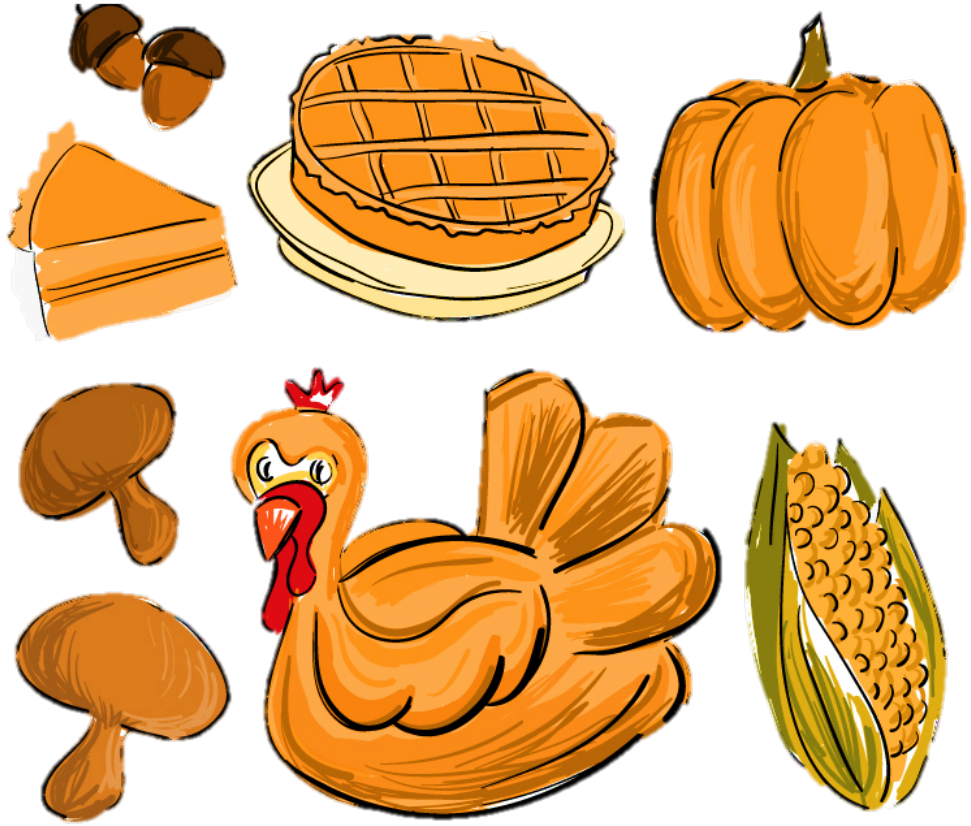


Lights are lit from left to right, then right to left and so on

Summary

- Memory (state) elements
  - Requires a clock signal to know when to update state value
  - Unclocked S-R latch → Clocked D latch → Flip-flop
- Sequential logic
  - Finite state machine
  - Decompose into two CL functions
    - Next state function: compute next state value based on current state value and inputs
    - Output function: compute output based on current state value and inputs

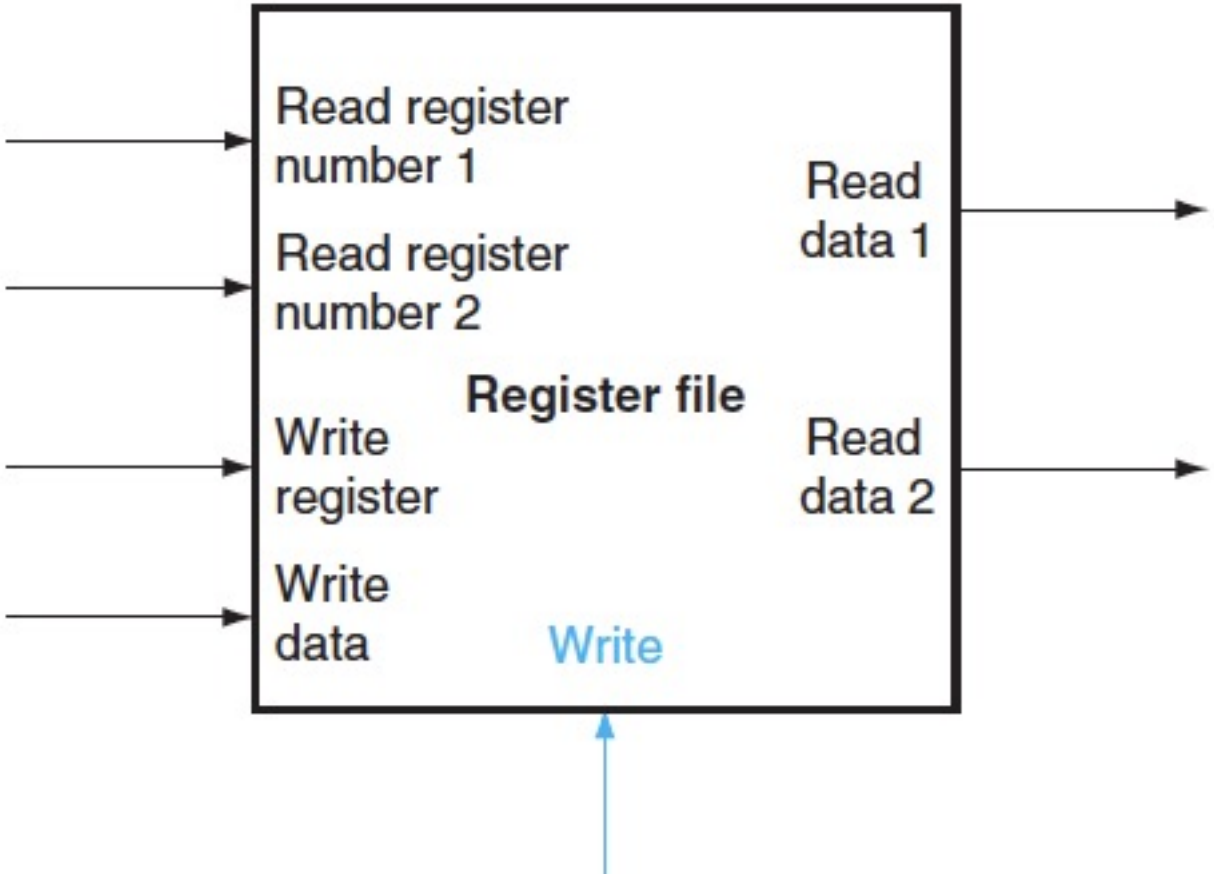
Happy Thanksgiving!



Stay safe!

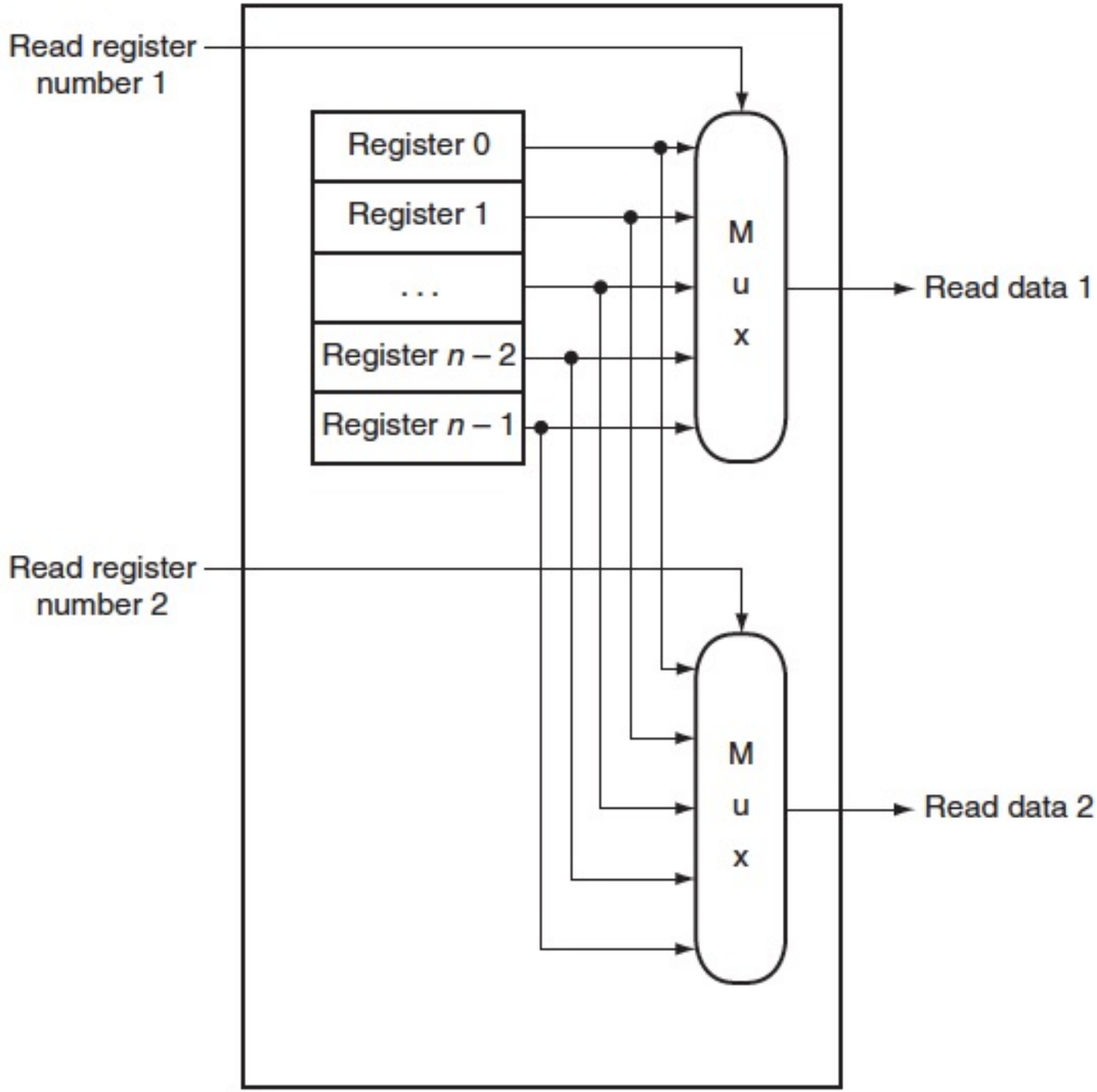
Memory element: Register file

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



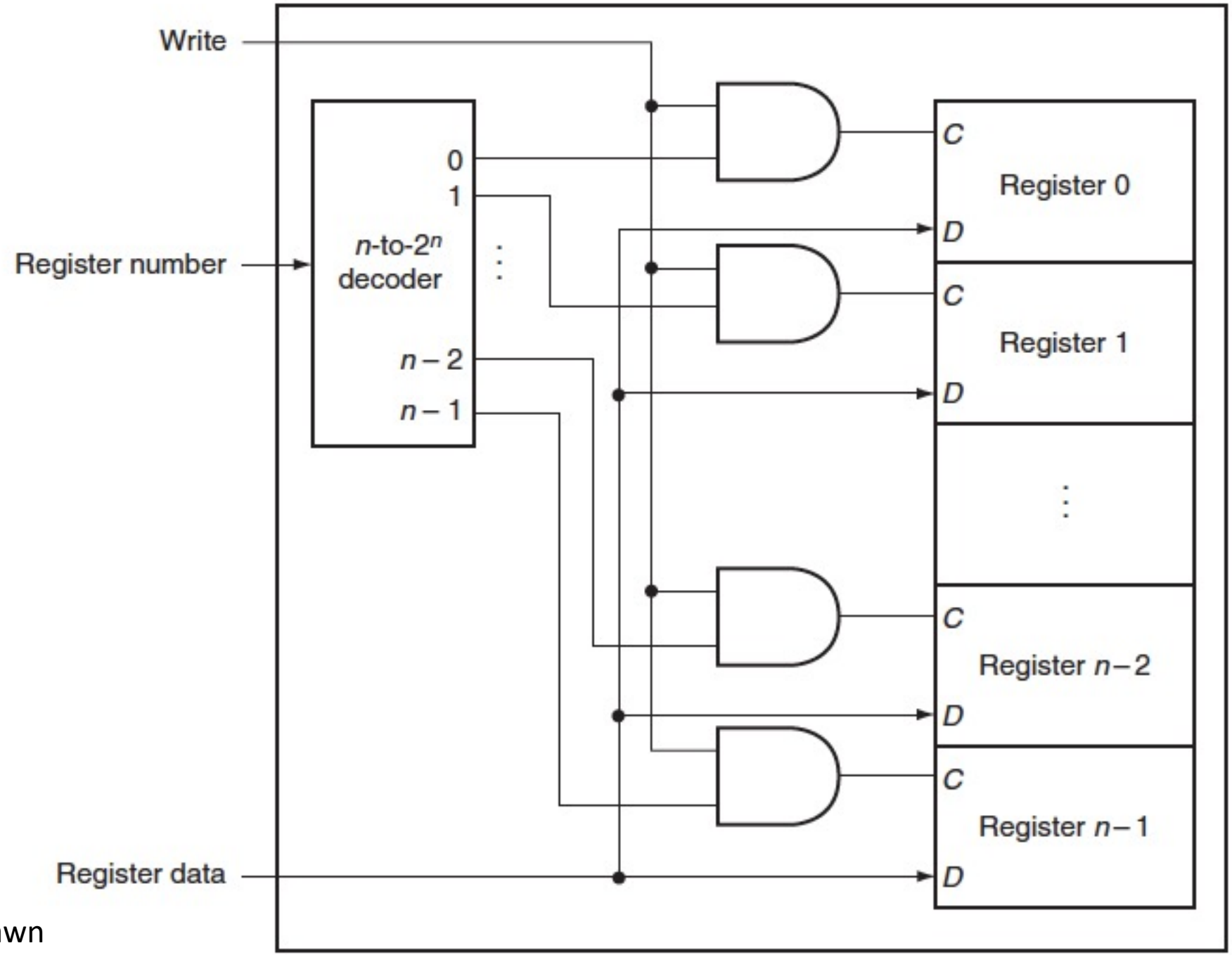
😕Why reading two registers at a time?

Register file: Read



Clock signal is assumed and not drawn

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 register 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