

# Management and Analysis of Physics Dataset – 1

## A.A. 2020-21

### Digital Circuits - 5

#### Overview

G.Collazuol

- Digital Circuits properties
  - Asynchronous vs Synchronous circuits
  - Combinational vs Sequential
- Combinational circuits → synthesis and examples
- Sequential circuits → latches and flip-flops

# Asynchronous vs Synchronous Operation

Asynchronous operation serves for diverse specific processing that do not require using a clock (mismatches between delays can impair the results).

More reliable is when timing is controlled by a clock (synchronous operation). The period must be such as to accommodate the most time consuming processing section.

Often, the circuit includes sections where processing is performed in multiple clock periods.

## Sequential Operation

A sequential system relies on a sequence of actions, required to occur in the right order.

Sequential architectures can be synchronous and asynchronous.

Use of a clock to establish the update times of the feedback signals or no time control the circuit sends back outputs continuously as soon they change value.

Sequential circuits, both synchronous and asynchronous architectures enable higher level processing than combinational schemes.

With given inputs possible instability. It persists until a different input configuration takes the output out of unstable conditions.

# Combinational vs Sequential Operation

The task of combinational logics is just to relate signals at input and generate defined logic signals at output.



Input B going high **causes** X to go low

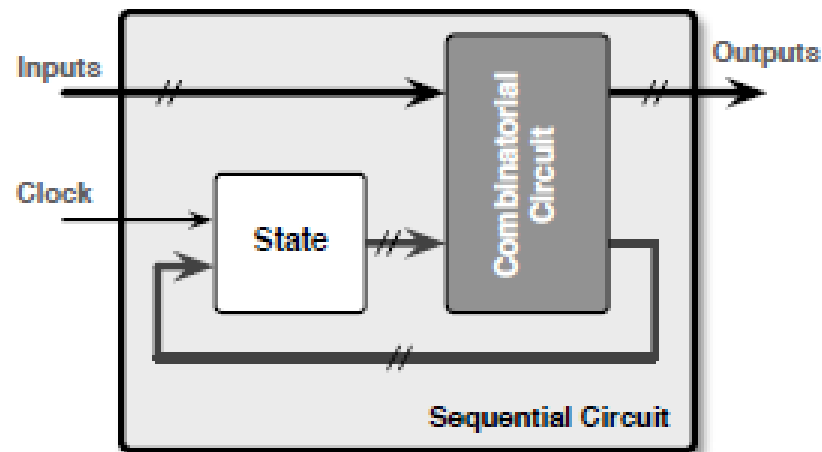
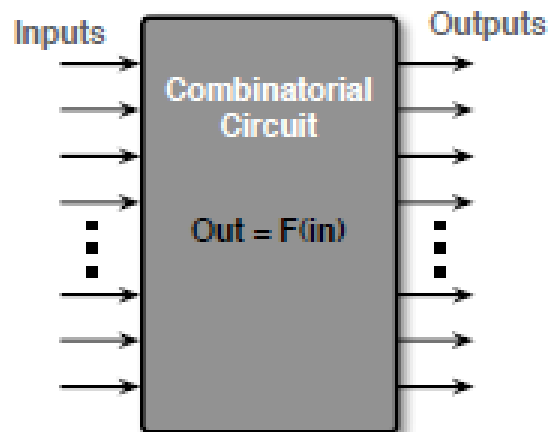
Input A going low **causes** X to go high



The output of combinational circuits depends only on input.

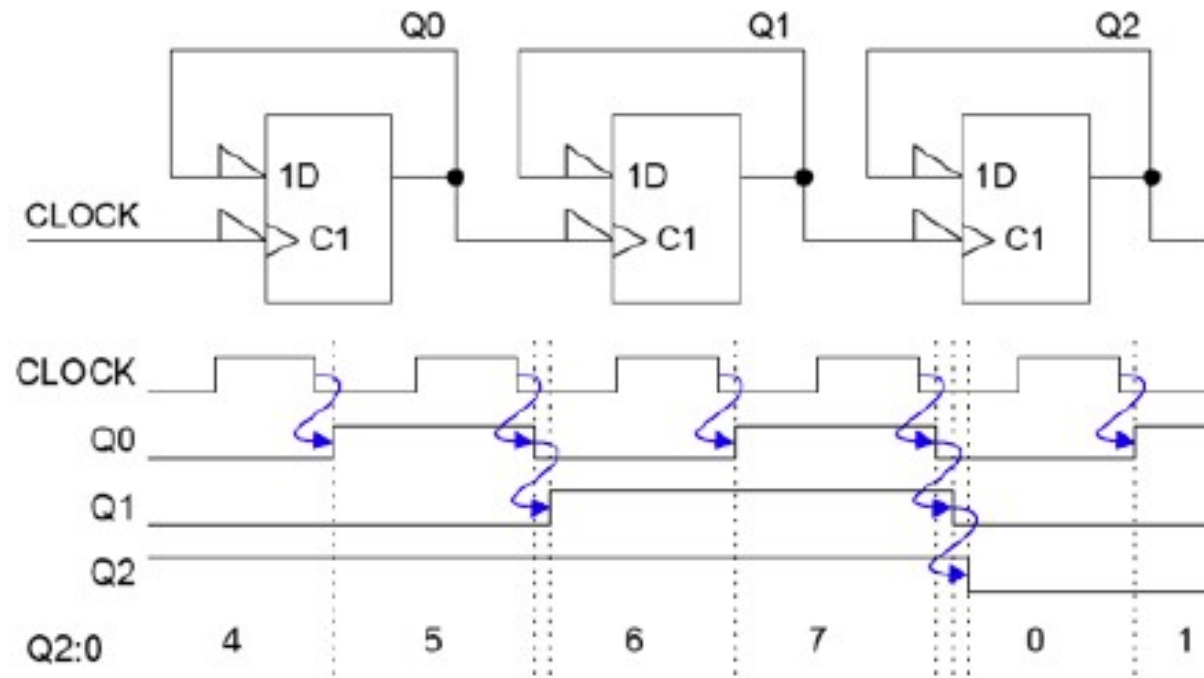
Sequential circuits produce output on the basis of both input and output.

For example, a memory circuit is essentially sequential, because its output depends on the input that occurred in the past. Instead, the addition of two inputs is combinational because the result just depends on changing inputs.



# Ripple Counter

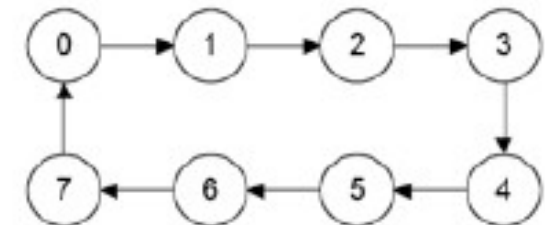
example of a D-FF used in a ripple counter.



- Notice inverters on the CLOCK and DATA inputs
- Least significant bit of a number is always labelled 0

Propagation Delay: CLOCK↓ to Q2 =  $3 \times 1\text{ns} = 3\text{ns}$

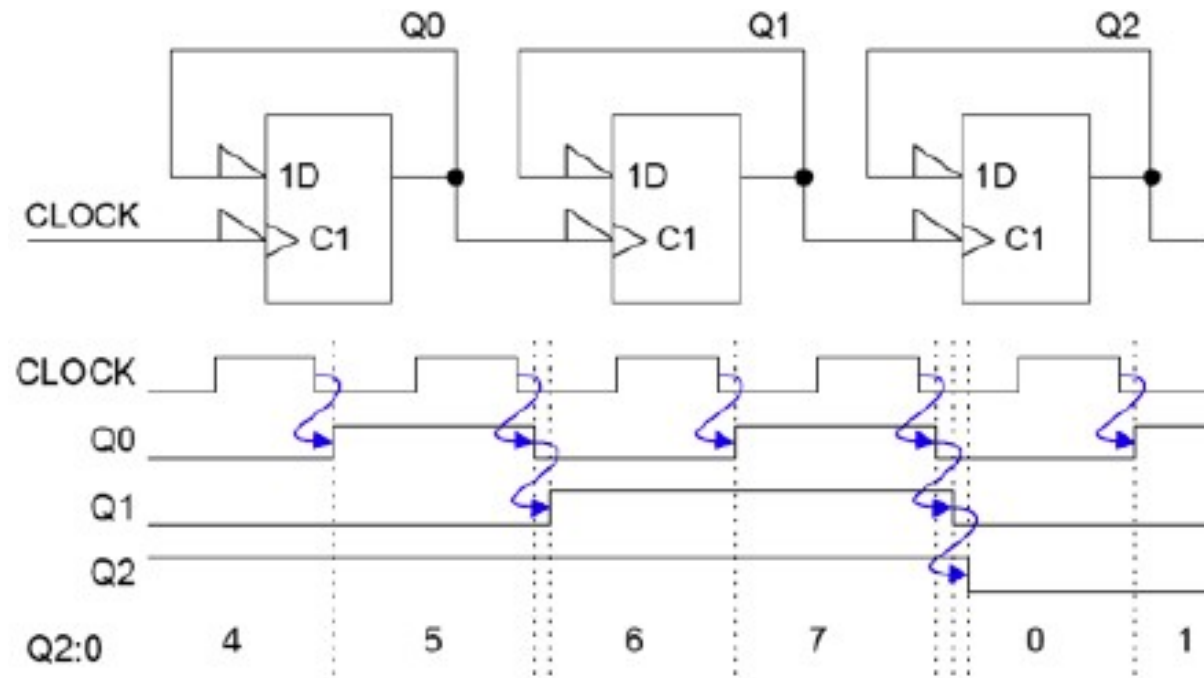
**State Diagram**  
(not including transient states):



This counter is also known as an **asynchronous sequential circuit**. It is “**asynchronous**” because the output signals are NOT synchronised to a single clock signal (since there are many clock signals), and “**sequential**” because its current output value (or state) depends on previous output values in the sequence.

# Ripple Counter

example of a D-FF used in a ripple counter.



- Notice inverters on the CLOCK and DATA inputs
- Least significant bit of a number is always labelled 0

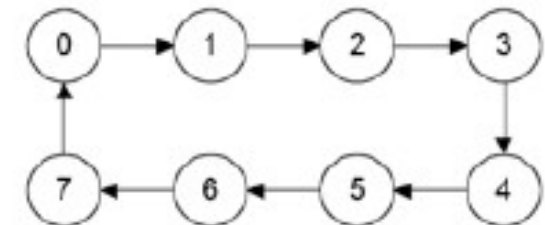
Q0 value is first inverted (represented by the triangle) and then used as D input on the next clock cycle. The flipflop is triggered on the FALLING edge of CLOCK. Therefore the Q output “TOGGLES” on each active edge of the clock (i.e. falling edge). Q0 is therefore changing at half the rate of CLOCK, hence this flipflop acts as a divide-by-2 circuit.

The Q0 signal is now used as clock input to the next D-FF. Hence Q1 is toggling at half the frequency of Q0. The circuit is effectively a binary counter.

This is a simple finite state machine (FSM) because it has 8 states which cycles through in a sequence. FSM will be covered in some later lectures in details and it is a very important topic in digital designs.

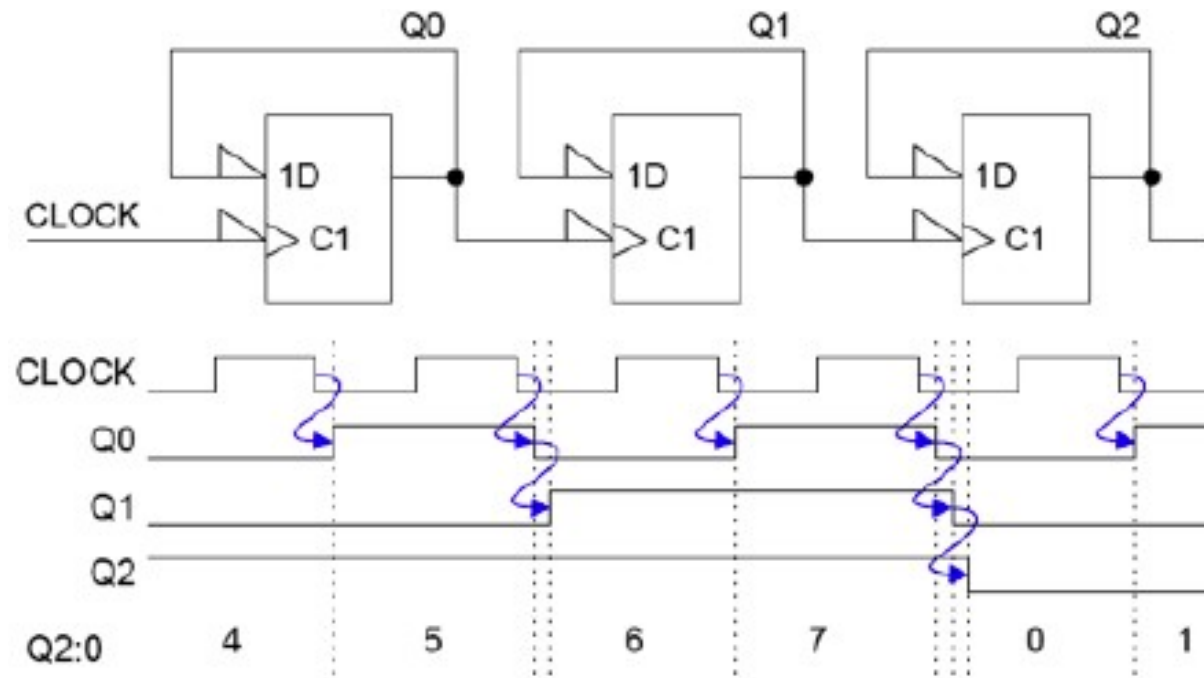
We then use the Q0 output as the clock input the next stage etc. Note that because the 2<sup>nd</sup> stage only starts to work once the first stage is completed, the propagation of effects “ripples” through the circuit – hence we call this a “ripple counter”.

**State Diagram**  
(not including transient states):



# Ripple Counter

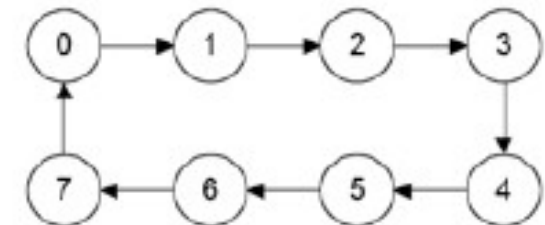
example of a D-FF used in a ripple counter.



- Notice inverters on the CLOCK and DATA inputs
- Least significant bit of a number is always labelled 0

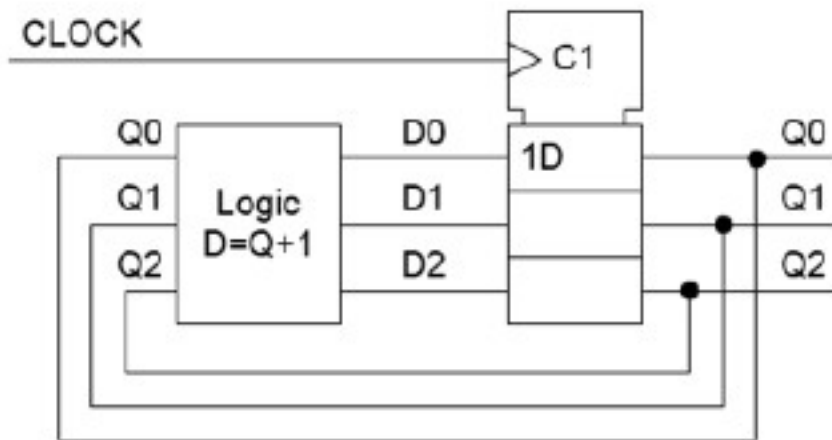
Propagation Delay: CLOCK↓ to Q2 =  $3 \times 1\text{ns} = 3\text{ns}$

**State Diagram**  
(not including transient states):



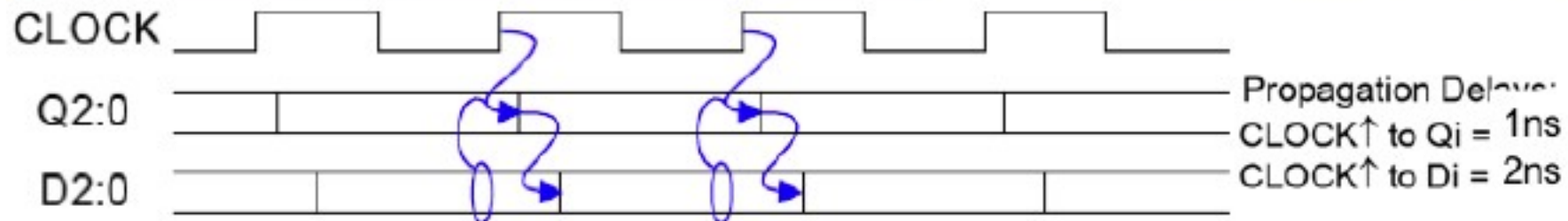
The ripple counter is potentially slow. The delay between the active edge of the clock and the counter output giving the correct value is dependent on the number of flipflops in the circuit and therefore the size of the counter (i.e. how many stages) .

# Synchronous Counter



The logic block must add 1 onto the current value of the counter,  $Q$ , to generate the next value of the counter,  $D$ . Suppose it has a propagation delay of 1ns.

- A register is a bunch of flipflops with the same CLOCK.
- The individual flipflops are rectangles stacked on top of each other. Only the top one is labelled.
- All shared signals (e.g. the CLOCK input) go to the notched common control block at the top of the stack.

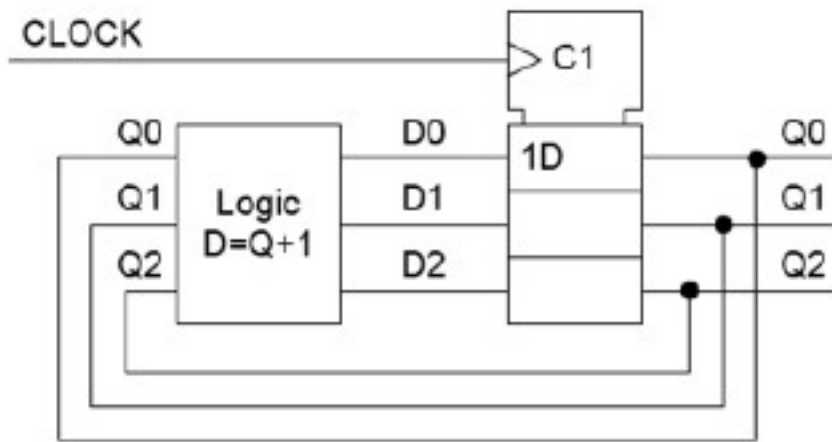


All flipflops change state within a fraction of nanosecond of each other.

A far better approach is to use the flipflops TOGETHER as a group, and clock them using THE SAME CLOCK signal as shown here. The Logic Block is a combinatorial circuit which computes the next D value  $D_{2:0}$  from the current Q value  $Q_{2:0}$ . (D has three bits  $D_0$ ,  $D_1$  and  $D_2$ . We use the notation  $D_{2:0}$  to represent this.) The relationship between D and Q is simple:  $D_{2:0} = Q_{2:0} + 1$ .

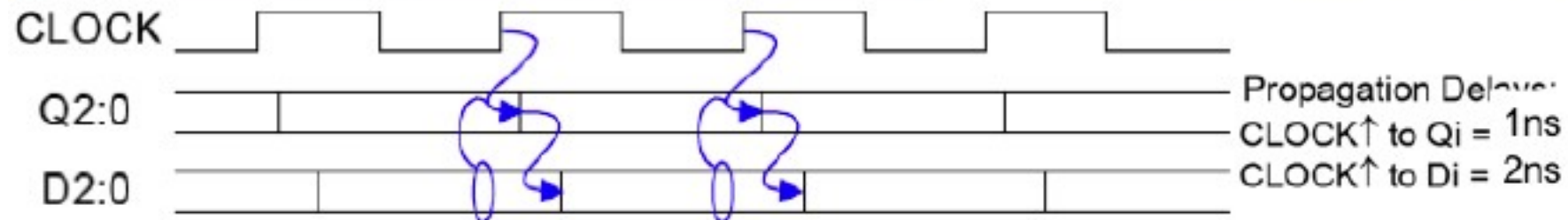


# Synchronous Counter



The logic block must add 1 onto the current value of the counter,  $Q$ , to generate the next value of the counter,  $D$ . Suppose it has a propagation delay of 1ns.

- A register is a bunch of flipflops with the same CLOCK.
- The individual flipflops are rectangles stacked on top of each other. Only the top one is labelled.
- All shared signals (e.g. the CLOCK input) go to the notched common control block at the top of the stack.

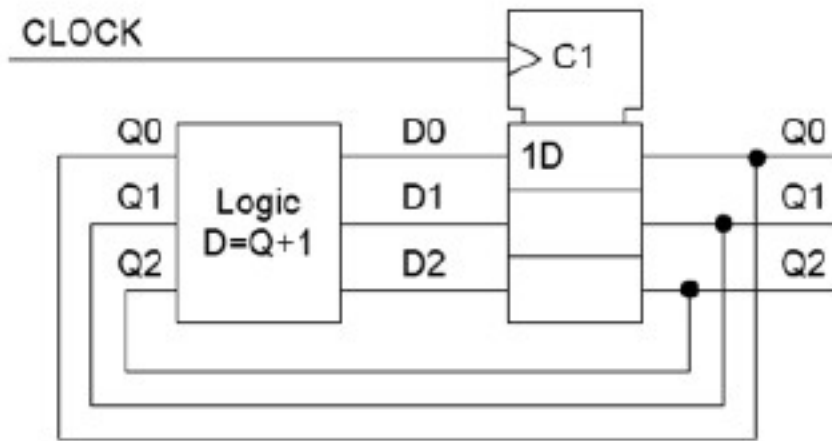


All flipflops change state within a fraction of nanosecond of each other.

Since the three output bits  $Q_{2:0}$  change within a fraction of a nanosecond of each other, this circuit is: 1) faster than the ripple counter; 2) the “delay” is constant instead of dependent on the size of the counter.

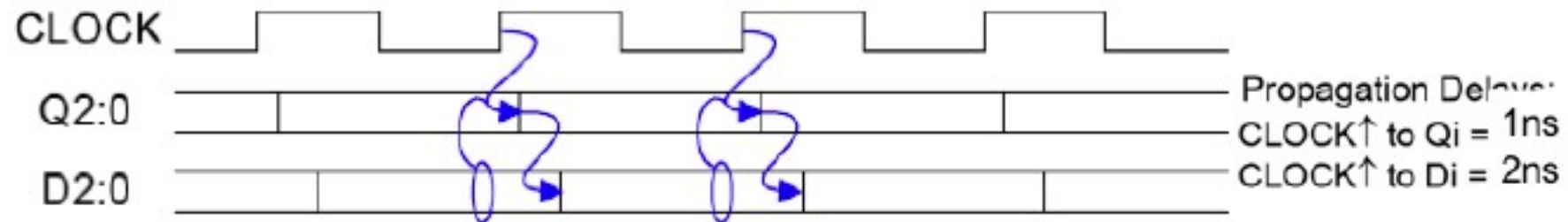


# Synchronous Counter



The logic block must add 1 onto the current value of the counter,  $Q$ , to generate the next value of the counter,  $D$ . Suppose it has a propagation delay of 1ns.

- A register is a bunch of flipflops with the same CLOCK.
- The individual flipflops are rectangles stacked on top of each other. Only the top one is labelled.
- All shared signals (e.g. the CLOCK input) go to the notched common control block at the top of the stack.



All flipflops change state within a fraction of nanosecond of each other.

This circuit is known as a **synchronous sequential circuit** because its function is synchronous to a single clock signal. If you regard the  $Q2:0$  output value as a state value, it follows a finite number of states in a defined sequence. Therefore it is also a form of **Finite State Machine**.

# Combinational Logic vs Sequential Logic

## Combinational circuits

the steady-state outputs are logical functions of the steady-state inputs only,

→ outputs do not depend on the internal condition of the circuit

→ combinational circuits can always be constructed with gates

Examples:

- adding or comparing integers
- deciding whether all the conditions for a decision are satisfied

## Sequential circuits

when memory of past actions is required → combinational circuits do not suffice

Example: if we want to detect the second occurrence of a given action, we must register the fact that at some point there was a first occurrence of this action

# Combinational Logic Circuits

- Synthesis of logical functions
- Examples of combinatorial circuits

# Synthesis of combinatorial logical functions

Logical functions can be written in more than one way

Some versions are more economical than others (in terms of number of gates)  
eg: the function  $F=A+AB$ , for example, can be written more succinctly as  $F=A+B$

Criteria for synthesising

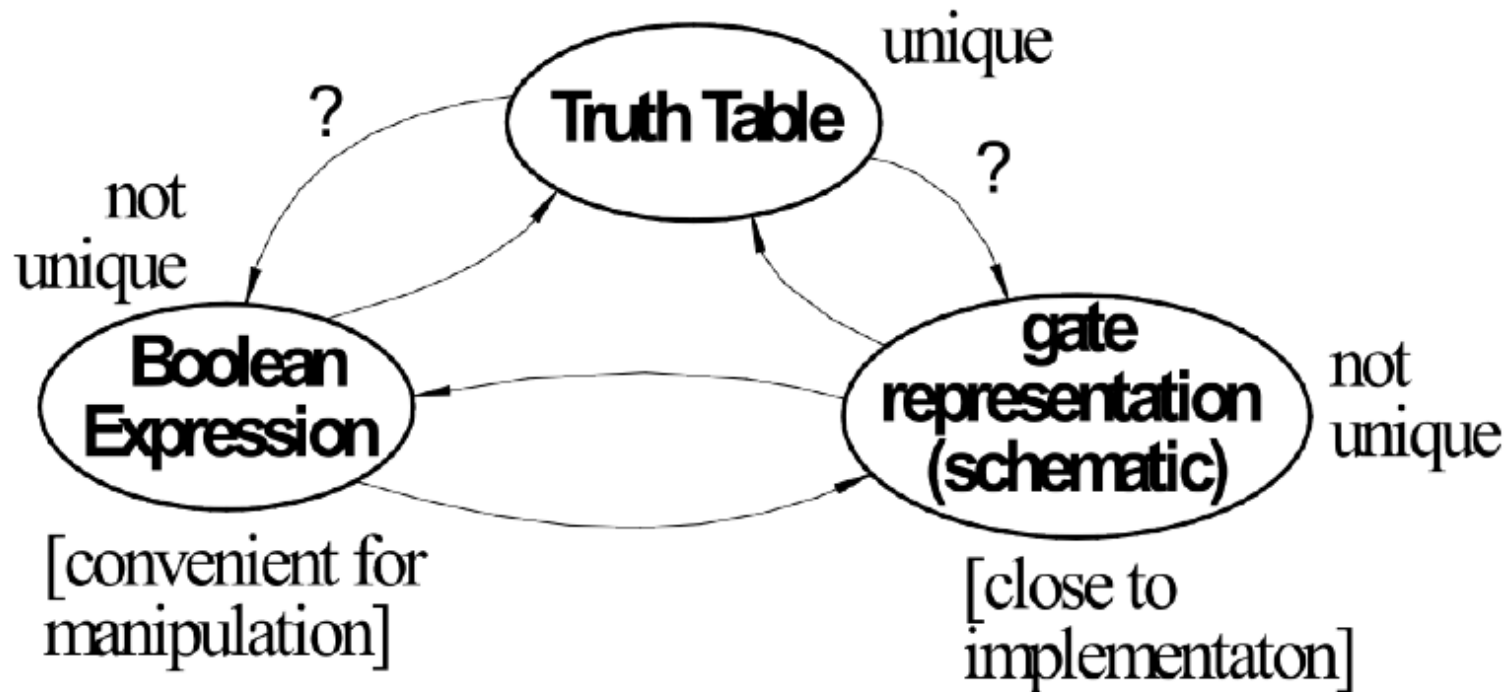
Minimizing the number of gates required to execute a logical function is not always a priority: speed or logical clarity, for example, almost always have higher priority

Programmable Logic Devices (PLD, eg FPGA) can generate arbitrary logical functions of a certain maximum number of variables when the interconnections of their internal gates are suitably programmed.

In PLDs standardization is an additional criterium that has priority over economy: logical functions are therefore implemented in the well-defined canonical forms  
→ Standardization of Boolean equations makes the implementation, evolution and simplification easier and more systematic

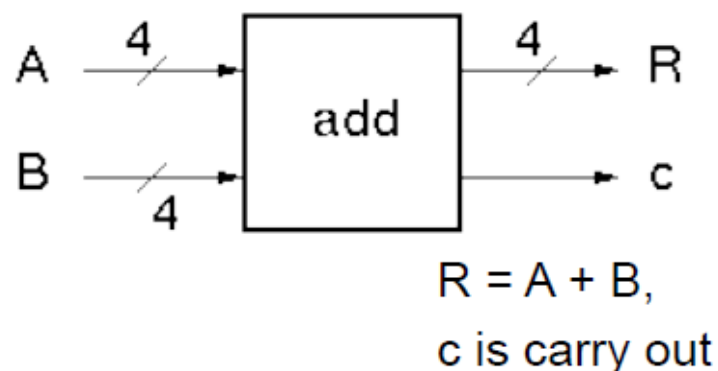
# Representation of combinatorial logical functions

Theorem: Any Boolean function that can be expressed as a truth table can be written as an expression in Boolean Algebra using AND, OR, NOT.



*How do we convert from one to the other?*

# Example – a CL block example: 4bit adder



- Truth Table Representation:

a3	a2	a1	a0	b3	b2	b1	b0	r3	r2	r1	r0	c
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	0	1	1	0	0	1	1	0
0	0	0	0	0	1	0	0	0	1	0	0	0
⋮												
0	0	1	0	0	0	1	0	0	1	0	0	0
0	0	1	0	0	0	1	1	0	1	0	1	0
⋮												
0	0	0	1	1	1	1	1	0	0	0	0	1

*In general:  $2^n$  rows for  $n$  inputs.*

*256 rows!*

*Is there a more efficient (compact) way to specify this function?*



# Example – a CL block example: 4bit adder

- Motivate the adder circuit design by hand addition:

$$\begin{array}{r} a_3 \ a_2 \ a_1 \ a_0 \\ + \ b_3 \ b_2 \ b_1 \ b_0 \\ \hline c \ r_3 \ r_2 \ r_1 \ r_0 \end{array}$$

- Add  $a_0$  and  $b_0$  as follows:

a	b	r	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

*carry to next stage*

$$r = a \text{ XOR } b = a \oplus b$$

$$c = a \text{ AND } b = ab$$

$$\begin{array}{r} a_3 \ a_2 \ a_1 \ a_0 \\ + \ b_3 \ b_2 \ b_1 \ b_0 \\ \hline c \ r_3 \ r_2 \ r_1 \ r_0 \end{array}$$

- Add  $a_1$  and  $b_1$  as follows:

$c_i$	a	b	r	$co$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$r = a \oplus b \oplus c_i$$

$$co = ab + ac_i + bc_i$$

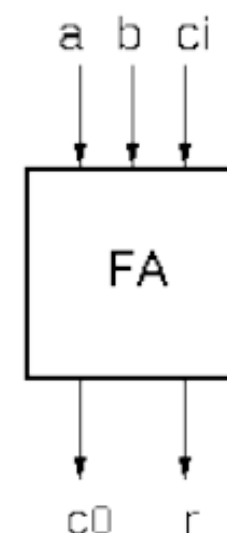
# Example – a CL block example: 4bit adder

- In general:

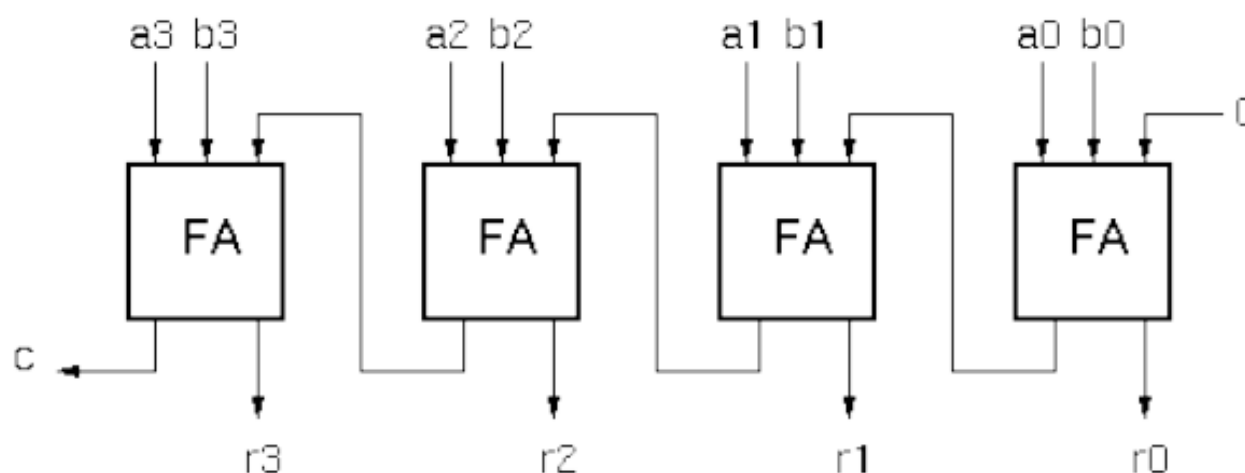
$$r_i = a_i \oplus b_i \oplus c_{in}$$

$$c_{out} = a_i c_{in} + a_i b_i + b_i c_{in} = c_{in}(a_i + b_i) + a_i b_i$$

- Now, the 4-bit adder:



*“Full adder cell”*



*“ripple” adder*

# Example – a CL block example: 4bit adder

## □ Graphical Representation of FA-cell

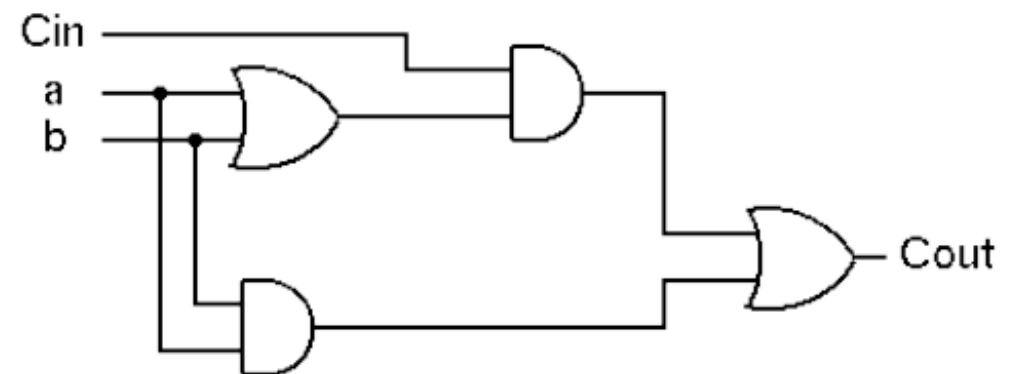
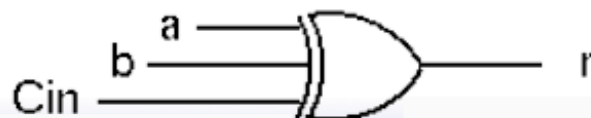
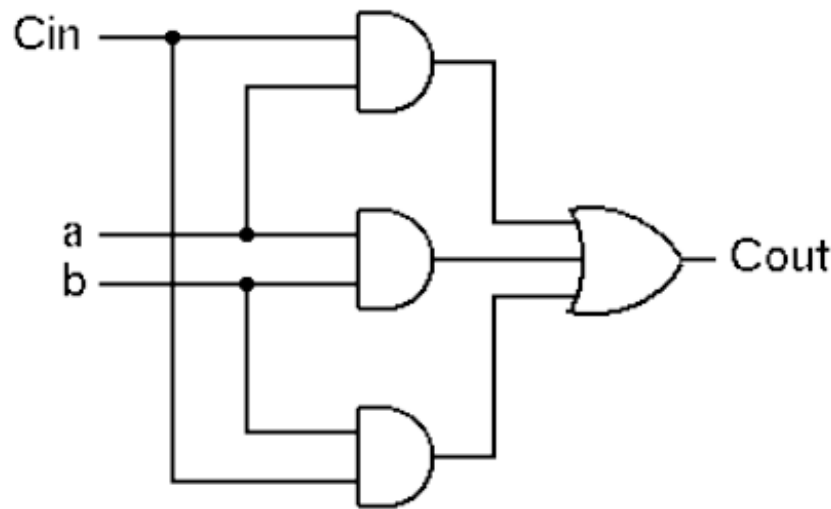
$$r_i = a_i \oplus b_i \oplus c_{in}$$

$$c_{out} = a_i c_{in} + a_i b_i + b_i c_{in}$$

- Alternative Implementation (with only 2-input gates):

$$r_i = [a_i \oplus b_i] \oplus c_{in}$$

$$c_{out} = c_{in}[a_i + b_i] + a_i b_i$$



# Minterms and Canonical form

Let us consider an arbitrary logical function of  $n$  variables  $F(A_1, \dots, A_n)$ , and let us further consider  **$n$ -term products of the form**  $(\overline{A_1}) \dots (\overline{A_n})$  in which each variable is represented once, either negated or not. Such products of this form are called a **minterms**.

Example: in the case of two variables  $(A, B)$  the 4 minterms are:  $\underline{A}\underline{B}, \underline{A}B, A\underline{B}, AB$

Note: for each array of **values** of the variables  $A_1 \dots A_n$

there is **one and only one minterm whose value is 1**:

→ it is the minterm in which  $A_i$  is not negated if  $A_i$  is equal to 1, and is negated if  $A_i$  is equal to 0

The number of minterms is  $2^n$  (ie the number of possible arrays of values)

**The function  $F$  is defined on the set of its minterms**

Note:

the number of the possible boolean functions of  $n$  boolean variables is  $2^{(2^n)}$  (indeed two possible values can be assigned to each minterm)

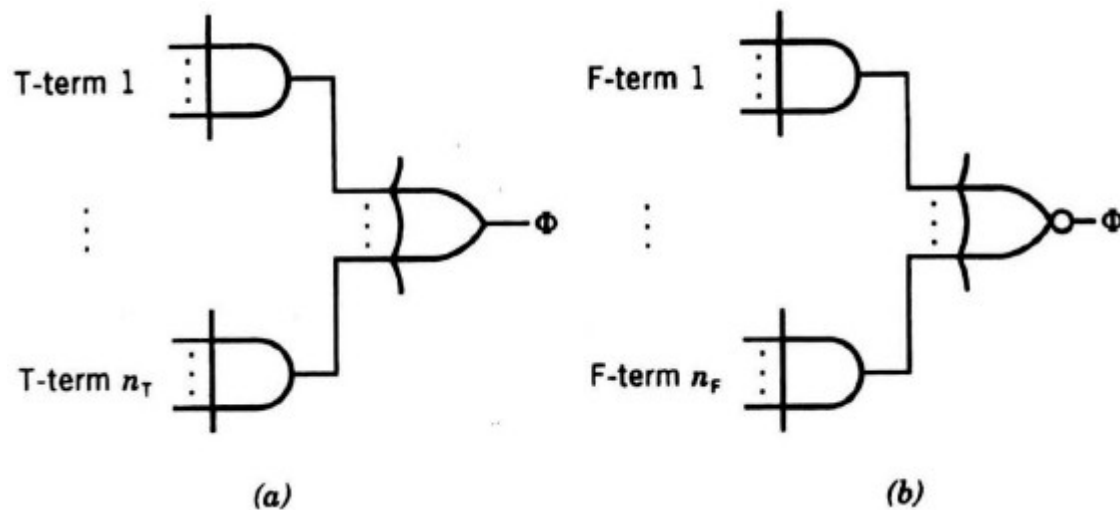
# Miniterms and Canonical form

Let us divide the minterms of  $F$  into two type of terms:

- **T-terms** (corresponding to arrays of values for which  $F = 1$  or TRUE)
- **F-terms** (corresponding to arrays of values for which  $F = 0$  or FALSE)

If the number of T-terms is  $n_T$  and the number of F-terms is  $n_F$  then  $n_T + n_F = 2^n$

→ **F is the sum of its T-terms** and  $F$  is also the negated sum of its F-terms



These are **canonical** representations of  $F$  as **sum of products (SOP)** of the  $n$  input variables

Note: similarly **maxterms** are defined as the  **$n$ -term sum of the  $n$  variables**, where each variable is represented once, either negate or not. For each array of values of the variables  $A_1 \dots A_n$  there is one and only one maxterm = 0

Example: in the case of two variables  $(A, B)$  the 4 maxterms are:  $(\underline{A} + \underline{B})$ ,  $(\underline{A} + B)$ ,  $(A + \underline{B})$ ,  $(A + B)$

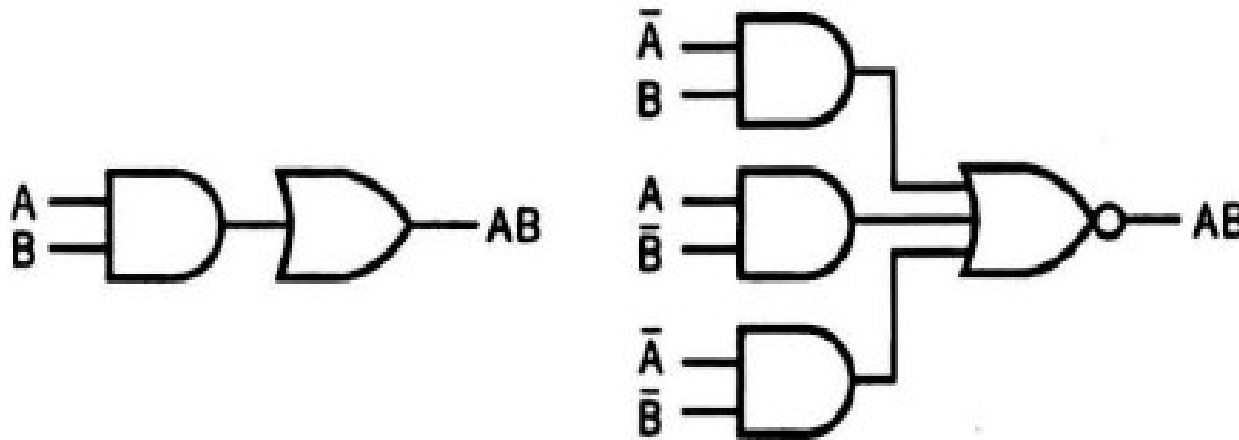
It can be seen that **F can be represented as product of sums (POS)**, namely T-maxterms

## Example: canonical form of $F(A,B) = AB$

The only T-term of  $F$  is  $AB \rightarrow n_T = 1$

The F-terms are  $\underline{A}B, A\underline{B}, \underline{A}\underline{B} \rightarrow n_F = 3$

The canonical representations of  $AB$  in miniterms are thus:



As a check:  $\underline{A}B + A\underline{B} + \underline{A}\underline{B} = \underline{A}(B + \underline{B}) + A\underline{B} = A + A\underline{B} = \underline{A} + \underline{B}$  or  $\underline{A}B + A\underline{B} + \underline{A}\underline{B} = \underline{A}\underline{B}$   
→ negating the last expression yields  $F(A,B) = AB$

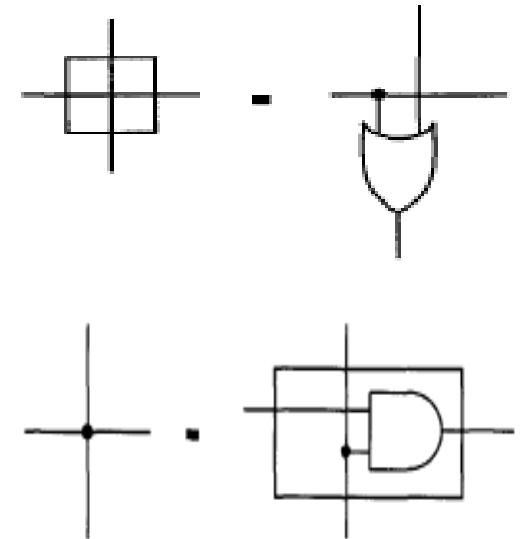
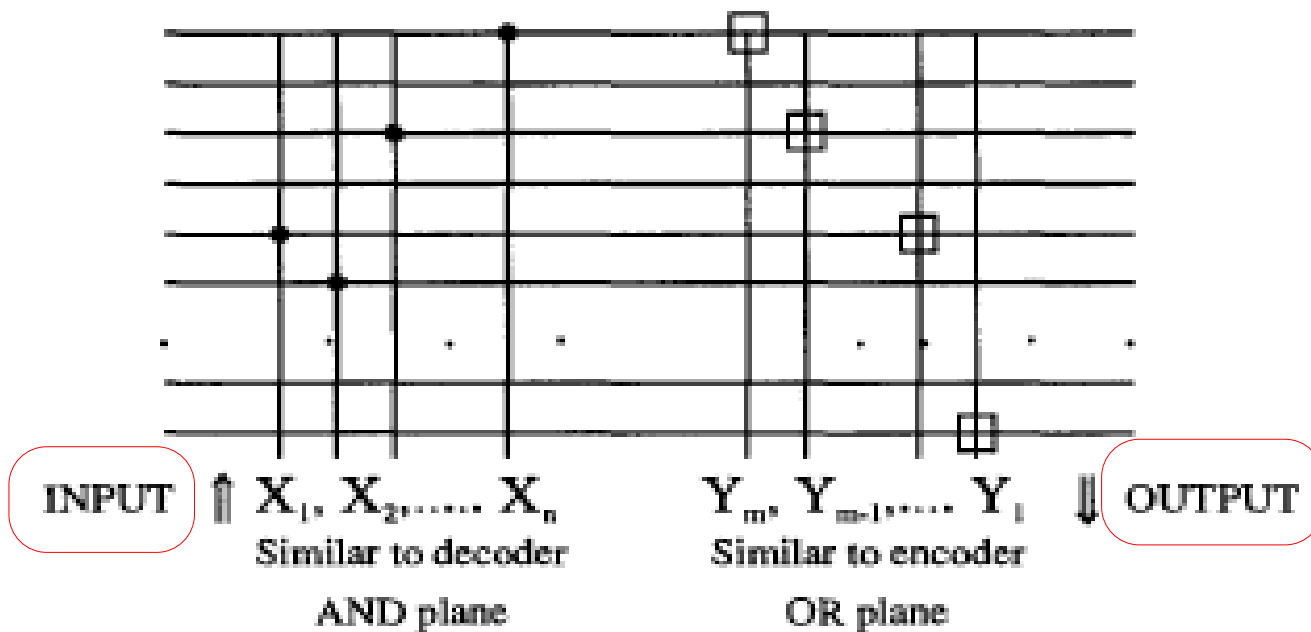


# Note: Complete sets of operators

Given a basic set of Operations (eg {**AND**, **OR**, **FANOUT**, **EXCHANGE**})  
then for any set of  $m$  output functions  $\{F_i, i=1, \dots, m\}$  on  $n$  inputs  $\{X_j, j=1, \dots, n\}$

$$Y_i = F_i(\{X\}), i=1, \dots, m$$

it can be shown that a circuit that can perform that functions can be built



Procedure quite simple (each output is a single value function) ... just many wires !

# Canonical Forms

- Standard form for a Boolean expression - unique algebraic expression directly from a true table (TT) description.
- Two Types:
  - \* **Sum of Products (SOP)** —► Easily implemented by NAND
  - \* **Product of Sums (POS)** —► Easily implemented by NOR
- Sum of Products (disjunctive normal form, minterm expansion). Example:

Minterms	a	b	c	f	f'
a'b'c'	0	0	0	0	1
a'b'c'	0	0	1	0	1
a'bc'	0	1	0	0	1
a'bc	0	1	1	1	0
ab'c'	1	0	0	1	0
ab'c	1	0	1	1	0
abc'	1	1	0	1	0
abc	1	1	1	1	0

One product (and) term for each 1 in f:

$$f = a'bc + ab'c' + ab'c + abc' + abc$$

$$f' = a'b'c' + a'b'c + a'bc'$$

*What is the cost?*

# Canonical Forms - SOP

Canonical Forms are usually not minimal:

Our Example:

$$\begin{aligned}f &= a'bc + ab'c' + ab'c + abc' + abc && (xy' + xy = x) \\&= a'bc + ab' + ab \\&= a'bc + a && (x'y + x = y + x) \\&= a + bc\end{aligned}$$

$$\begin{aligned}f' &= a'b'c' + a'b'c + a'bc' \\&= a'b' + a'bc' \\&= a' ( b' + bc' ) \\&= a' ( b' + c' ) \\&= a'b' + a'c'\end{aligned}$$

→ SOP form can be obtained by

- Writing an AND term for each input combination, which produces HIGH output.
- Writing the input variables if the value is 1, and write the complement of the variable if its value is 0.
- OR the AND terms to obtain the output function.

## Simplify – Algebra of Karnaugh-Maps ...

□ Algebra:  $f = a + bc$

□ K-maps:

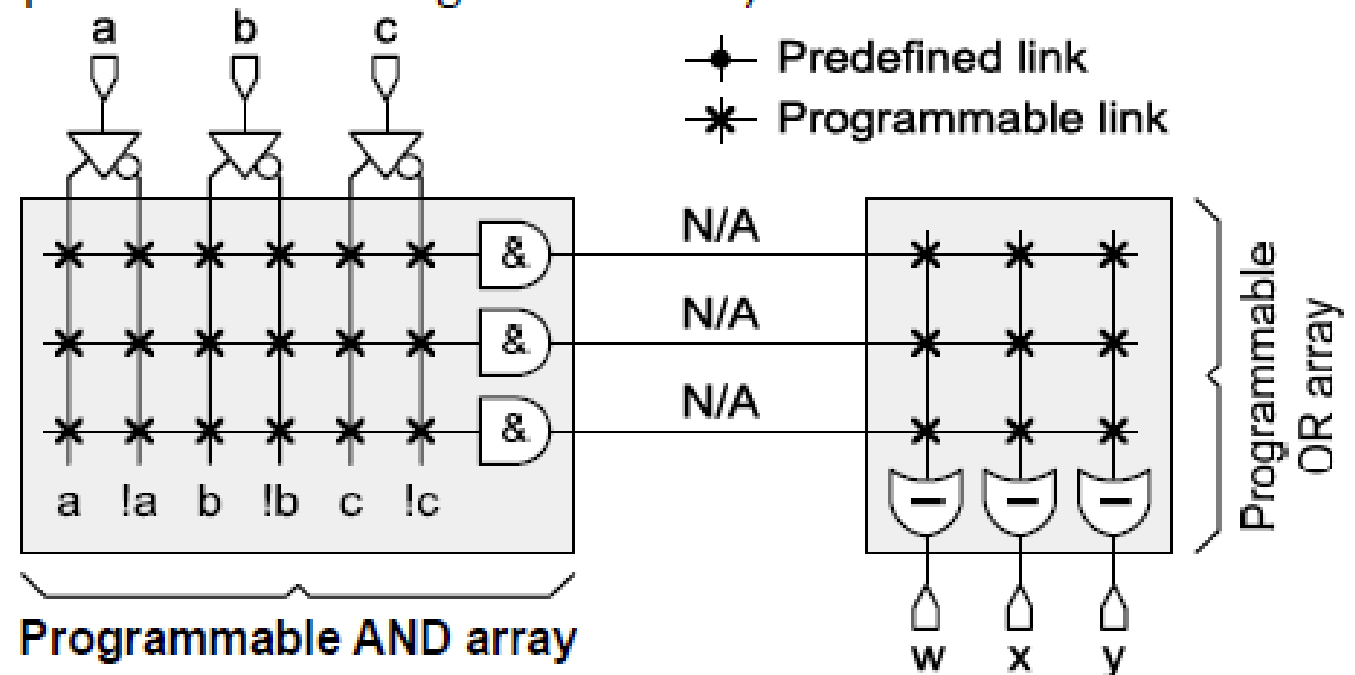
		ab			
		00	01	11	10
c	0	0	0	1	1
	1	0	1	1	1

various methods can be exploited

... FPGAs → directly mapping truth tables

# Programmable Logic Arrays

Una PLA (*programmable logic array*) è simile a una PROM ma ha sia il piano AND che il piano OR programmabili. A differenza delle PROM, la dimensione del piano AND è indipendente dal numero di ingressi. Lo svantaggio principale è una minore velocità perché il segnale in una PLA deve attraversare due livelli di collegamenti programmabili (che sono più lenti di un collegamento fisso).



# Practical examples of combinatorial logic circuits

- Binary adder
- Arithmetic Logic Units
- Magnitude Comparator



# Binary addition with gates

In adding two binary words, at each position we need

- (1) to know whether there is a carry from the next-lowest position (carry-in)
- (2) to generate the carry for the next-highest position (carry-out)

At any given position, let

- A and B be the bits of the words to be added
- C the carry-in
- $\Sigma$  the sum
- K the carry-out

} 1bit adder

$\Sigma$  is the low-order bit of the sum  $A+B+C$ , and it is 1 in only two cases:

- only one of the addends is 1, or
- all three are 1

If we interpret A, B, C, and  $\Sigma$  as logical variables  $\rightarrow \Sigma = A\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C + ABC$

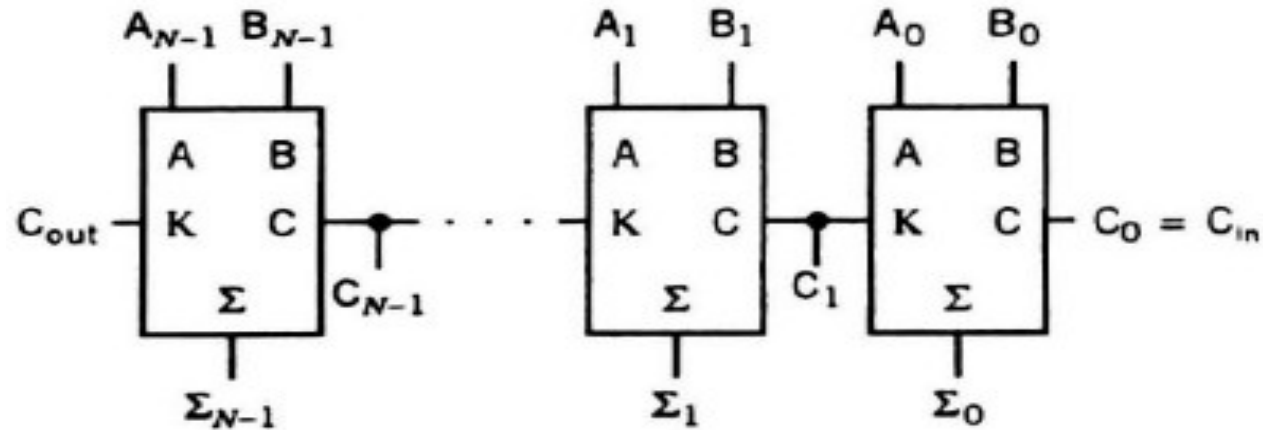
Similarly, K is 1 if any pair of addends is 1, and we obtain  $\rightarrow K = AB + BC + AC$

Note: the term ABC is already included in  $AB+BC+AC$

(indeed  $ABC=1$  implies  $AB=BC=AC=1$ )

# Binary addition with gates

If we want to add  
two  $N$ -bit words we can  
chain  $N$  1-bit adders together



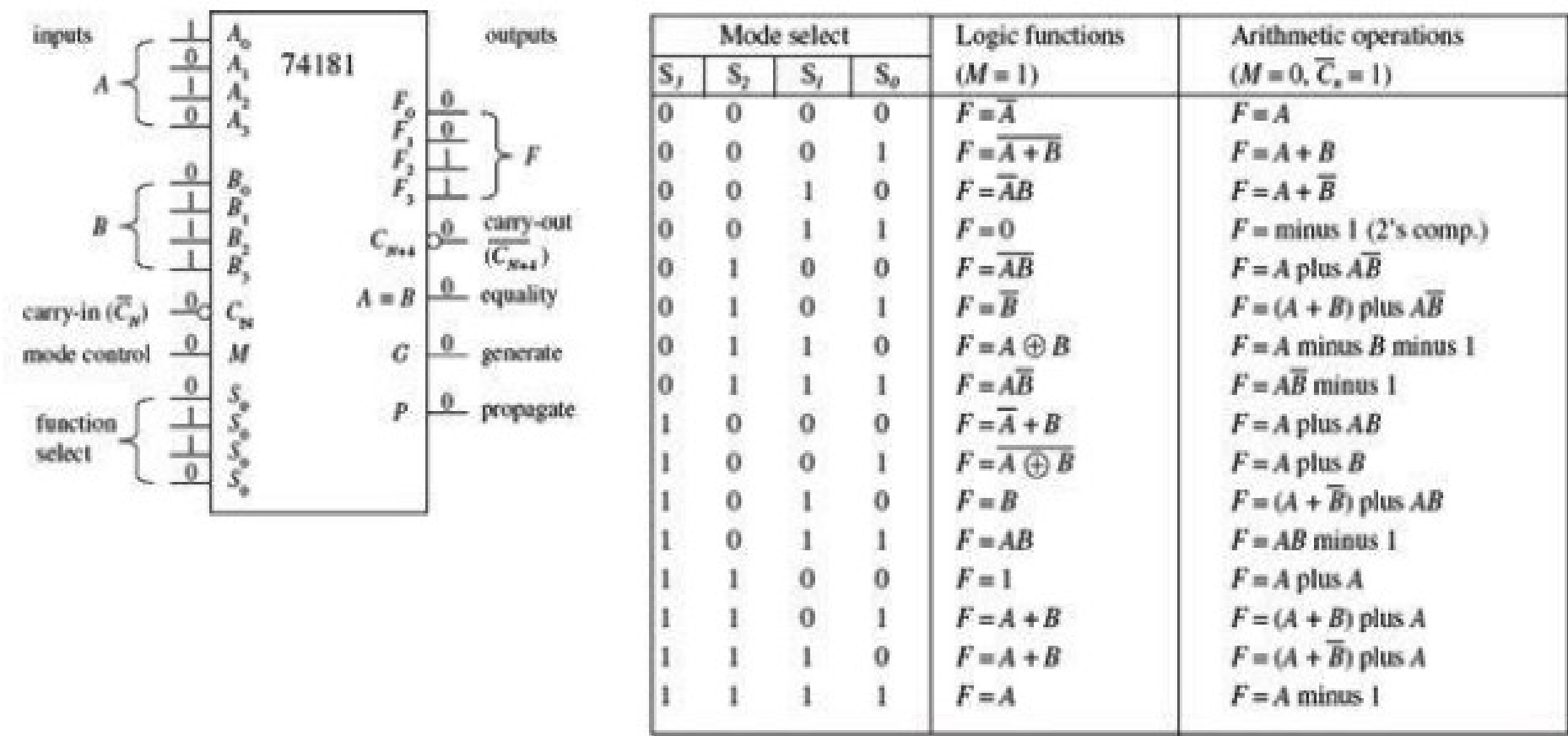
The adder for the lowest-order bit accepts a carry-in  $C_{in}$  from a possible lower-order word, and the adder for the highest-order bit generates a carry-out  $C_{out}$  that can feed a possible higher-order word

Note: the decision in a given bit has to wait for the decisions in all lower-order bits  
→ since each decision takes a finite time, the resulting delay in  $C_{out}$  might be considered excessive. This is the reason for the existence of [look-ahead carry generators](#), circuits that consider all possible combinations of the inputs and decide with minimal delay whether  $C_{out}$  will be generated

# Aritmetic Logic Units

An ALU is a multipurpose integrated circuit capable of performing various arithmetic and logic operations. To choose a specific operation to be performed, a binary code is applied to the IC's mode select inputs.

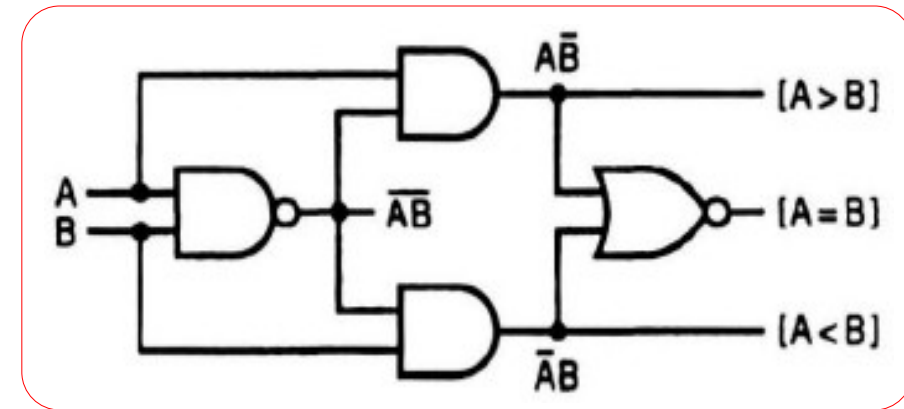
The 74181 fon instance is a 4-bit ALU that provides 16 arithmetic and 16 logic operations



# Magnitude comparator for unsigned integers

First construct **1-bit magnitude comparator**, which is a useful step because magnitude comparators usually have all 3 outputs for  $= > <$

The conditions  $[A > B]$ ,  $[A = B]$ , and  $[A < B]$  can be determined with the logical operations  $\overline{A}B$ ,  $(AB + \overline{A}\overline{B})$  (ie XNOR),  $A\overline{B}$  and correctly generated by the comparator in figure



A **2-bit comparator** is then constructed as follows. As an example, let us build a circuit that decides whether the 2-bit word  $A_1A_0$  is  $>$  (greater than) the 2-bit word  $B_1B_0$ . Note: the procedure for  $<$  (less than) is similar. Both can be used on wider words

We observe first that the condition  $[A_1 > B_1] = 1$  is sufficient. If it is not satisfied, we must require that  $[A_1 = B_1]$  and  $[A_0 > B_0]$  simultaneously be 1

We thus obtain  $\rightarrow$

$$[A_1A_0 > B_1B_0] = [A_1 > B_1] + [A_1 = B_1][A_0 > B_0]$$

# More combinatorial blocks

- Parity Generator / Checker
- Multiplexers / Demultiplexers
- Encoders / Decoders

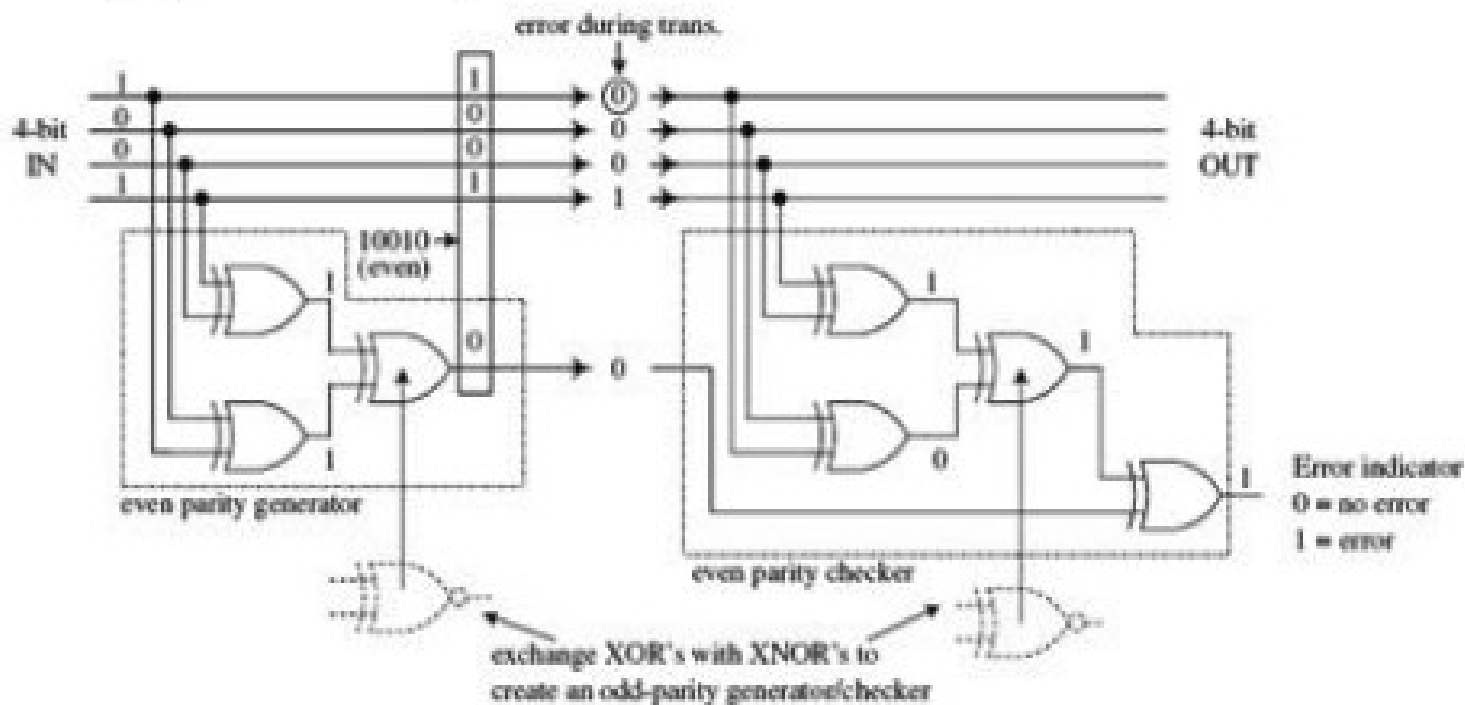
# Parity generator / Checker

Often, external noise can corrupt binary information (cause a bit to flip from one logic state to the other) as it travels along a conductor from one device to the next

4-bit even-parity error-detection system

Even vs. odd parity (examples)

4-bit string	parity bit	parity
1 0 0 1	0	even (two 1's)
1 1 0 0	1	odd (three 1's)
0 0 0 0	1	odd (one 1's)
1 1 1 0	1	even (four 1's)



Note: even (odd) parity generator when the generated parity bit makes the total number of 1s even (odd)



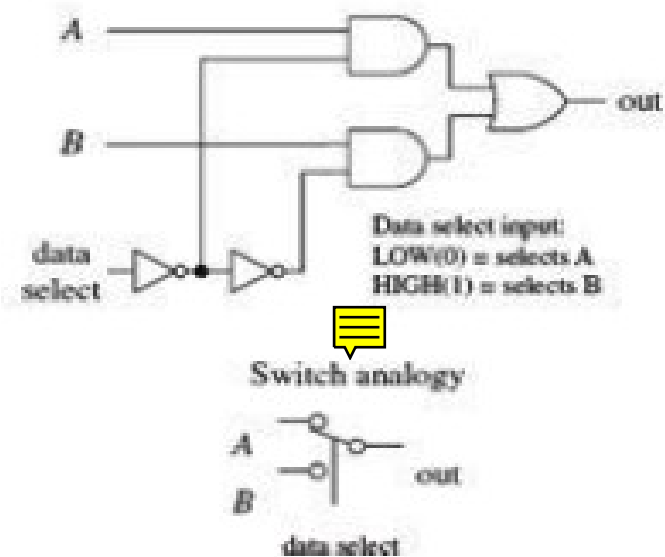


# Multiplexers (Data Selectors)

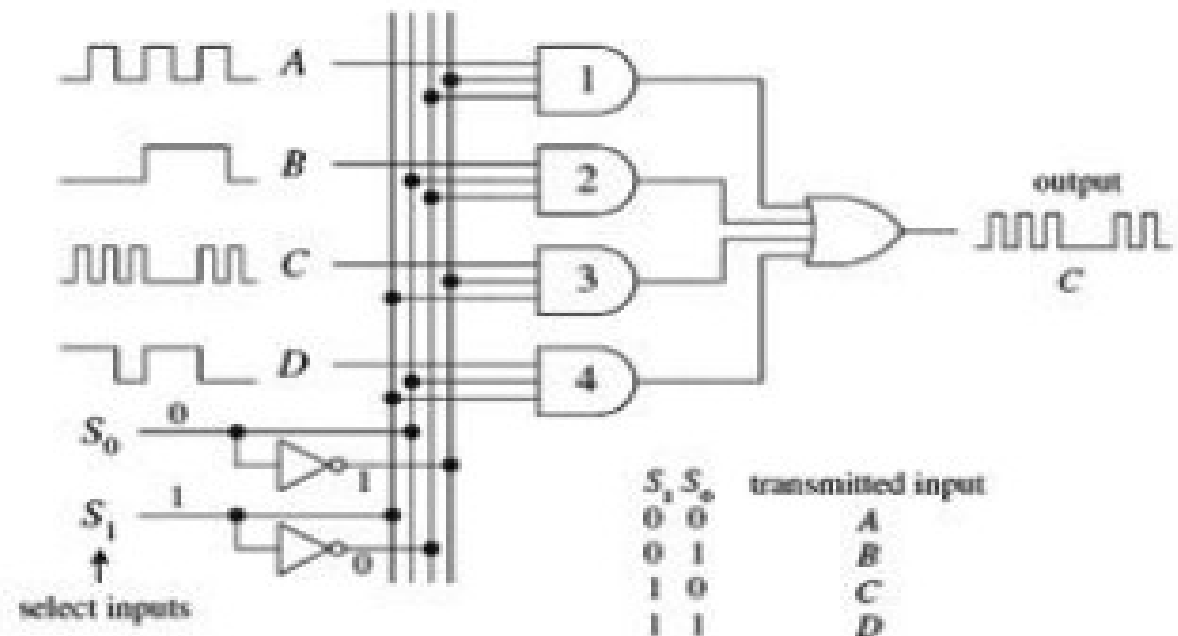
Multiplexing is the technique of selecting and transmitting signals from **multiple input lines** to a **single output line**

Multiplexers work as digitally controlled switches

Simple 2-to-1 data selector



4-line to 1-line multiplexer



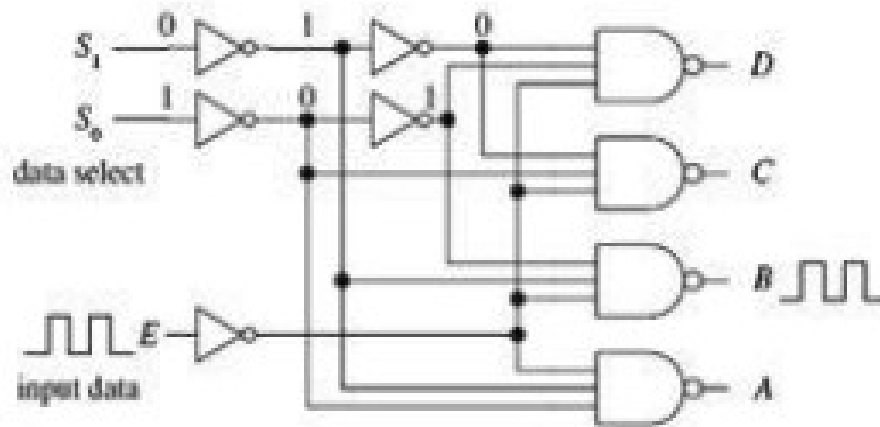
AND gates are enable gates

# Demultiplexer (Data Distributor)

A demultiplexer (or data distributor) is the opposite of a multiplexer: it takes a single data input and routes it to one of several possible outputs



A simple 1-to-4 four-line demultiplexer



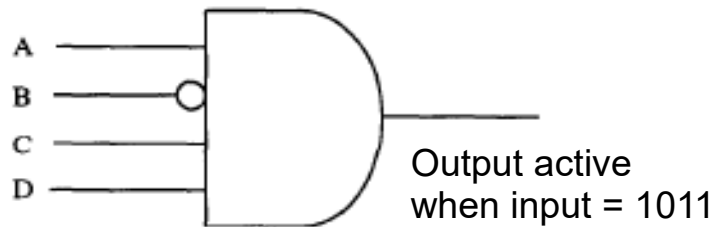
Control logic

$S_1$	$S_0$	input routed to:
0	0	A
0	1	B
1	0	C
1	1	D

disabled outputs  
are held HIGH

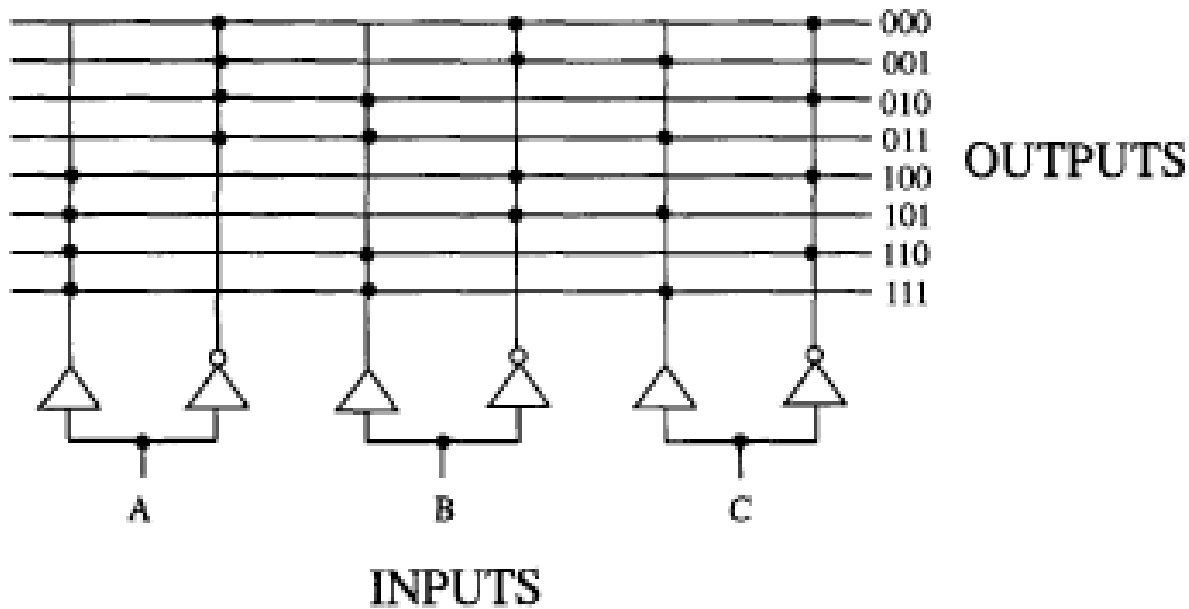
# Binary Decoder

## Simple binary decoder

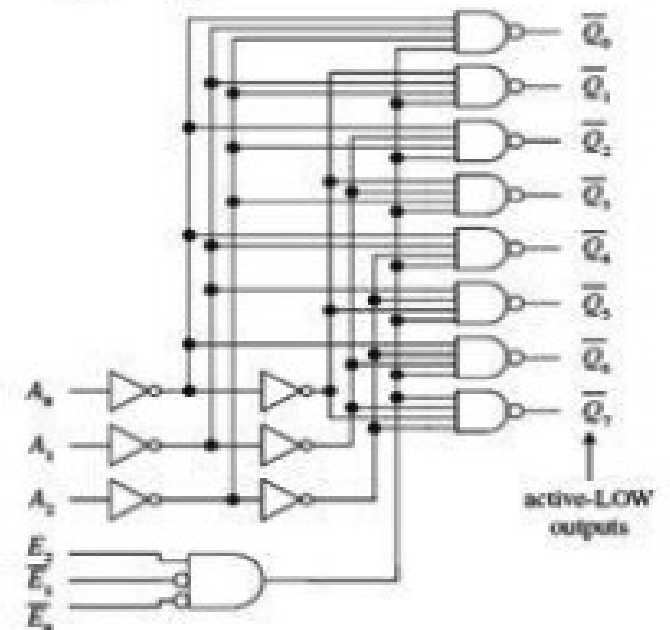


Data select inputs decide which outputs are active

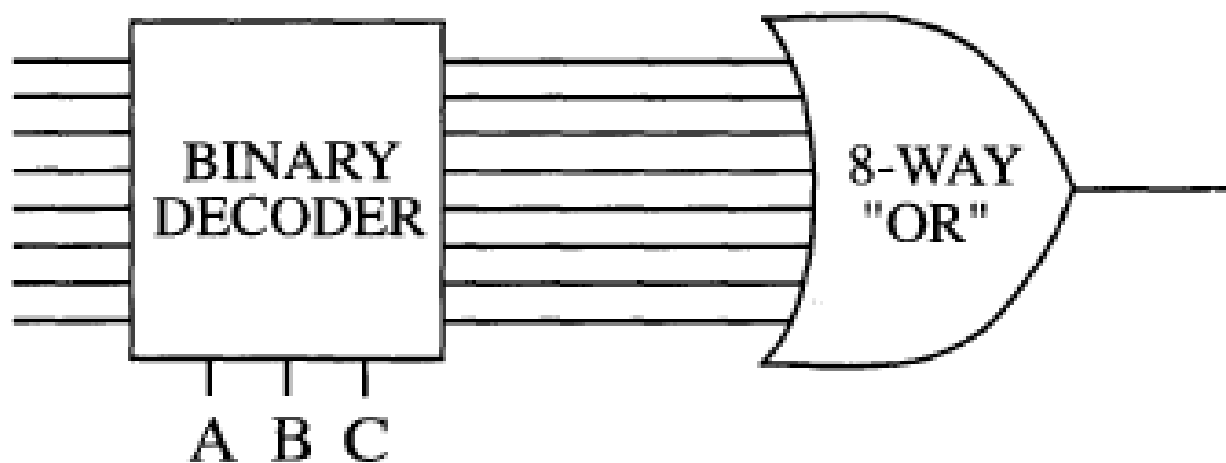
## 3-to-8 binary decoder



Logic diagram 74LS138 1-of-8 decoder



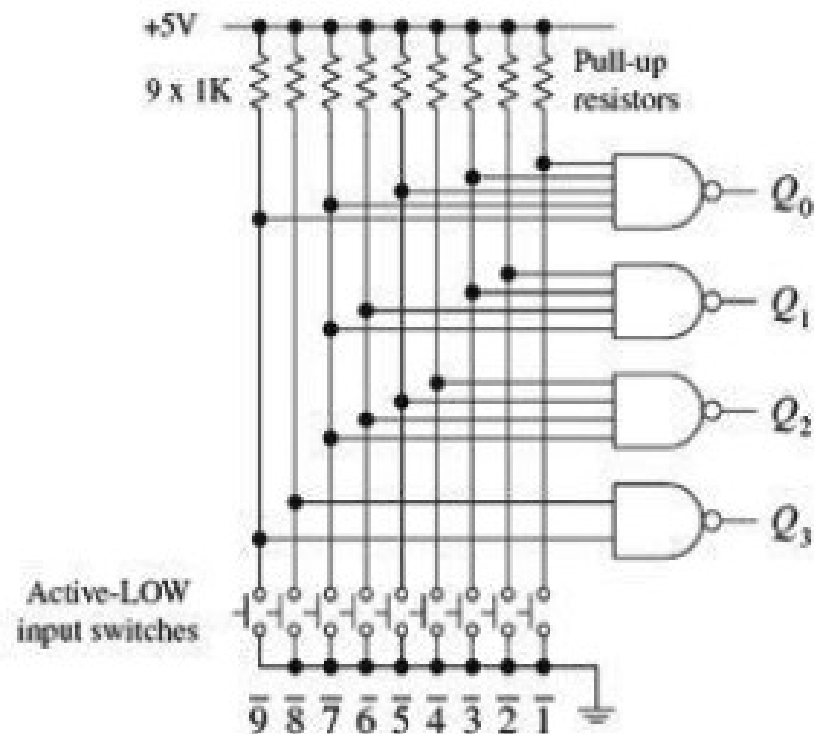
Note: Binary Decoder + OR = Multiplexer



# Encoder

Encoders are the opposite of decoders. They are used to generate a coded output from a single active numeric input

Simple decimal-to-BCD encoder



Truth table

$\bar{1}$	$\bar{2}$	$\bar{3}$	$\bar{4}$	$\bar{5}$	$\bar{6}$	$\bar{7}$	$\bar{8}$	$\bar{9}$	$Q_0$	$Q_1$	$Q_2$	$Q_3$	BCD (pos. logic)
H	H	H	H	H	H	H	H	H	L	L	L	L	0000 ( $0_{10}$ )
L	H	H	H	H	H	H	H	H	L	L	L	H	0001 ( $1_{10}$ )
H	L	H	H	H	H	H	H	H	L	L	H	L	0010 ( $2_{10}$ )
H	H	L	H	H	H	H	H	H	L	L	H	H	0011 ( $3_{10}$ )
H	H	H	L	H	H	H	H	H	L	H	L	L	0100 ( $4_{10}$ )
H	H	H	H	L	H	H	H	H	L	H	L	H	0101 ( $5_{10}$ )
H	H	H	H	H	L	H	H	H	L	H	H	L	0110 ( $6_{10}$ )
H	H	H	H	H	H	L	H	H	L	H	H	H	0111 ( $7_{10}$ )
H	H	H	H	H	H	H	L	H	H	L	L	L	1000 ( $8_{10}$ )
H	H	H	H	H	H	H	H	L	H	L	L	H	1001 ( $9_{10}$ )

H = High voltage level, L = Low voltage level

# Sequential Logic Circuits

- Async / Synchronous systems
- Flip-Flops / Latches
- Counters and Registers
- State Machines
- Synchronizers
- Memories

# Synchronous vs Asynchronous

- Synchronous sequential logic:
  - the time at which transitions between circuit states occurs is controlled by a **common clock** signal
  - changes in all variables occur simultaneously
- Asynchronous sequential logic:
  - state transitions occur independently of any clock, and normally depend on the time at which input variables change
  - outputs do not necessarily change simultaneously
- **Clock**
  - a clock signal is a square wave of a fixed frequency
  - it is used to trigger state transitions at fixed times in synchronous circuits


# Flip-flops and Latches

- Flip-flops and latches are the fundamental elements of sequential circuits
  - bistable (two stable states)
- Flip-flops and latches are essentially 1-bit storage devices
  - outputs can be set to store either 1 or 0 depending on the inputs
  - even when the inputs are deasserted, the outputs retain their prescribed values
- Flip-flops and latches (normally) have 2 complementary outputs
  - usually denoted  $Q$  and  $\bar{Q}$
- Three main types:
  - R-S      J-K      D-type

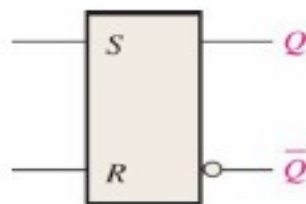


# Name conventions

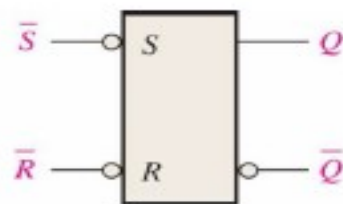
Standard distinction between:

- 1) **Latches** → Flip-Flops which are **sensitive to levels** (high or low) 
- 2) **Flip-flops** → Flip-Flops that are **sensitive to transitions** (signal edges)

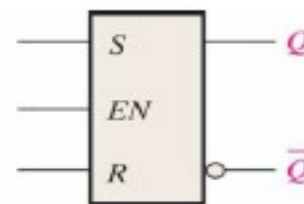
... sometimes both of them are called Flip-Flop in any case ...



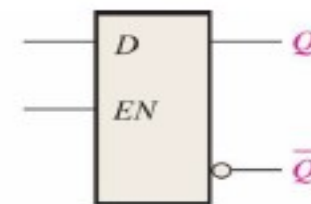
(a) Active-HIGH  
input S-R latch



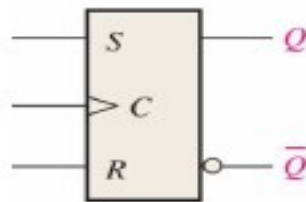
(b) Active-LOW input  
 $\bar{S}$ - $\bar{R}$  latch



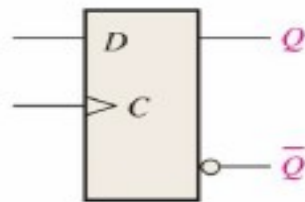
(c) Gated S-R latch



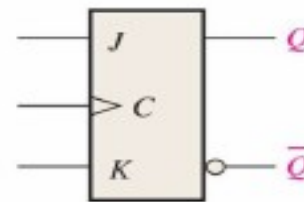
(d) Gated D latch



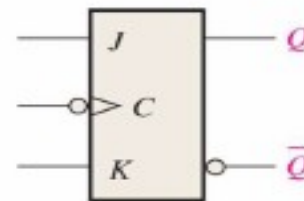
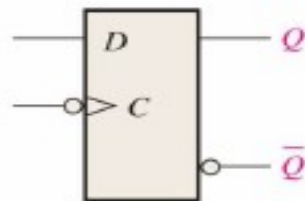
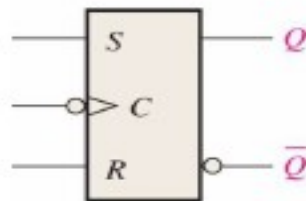
(e) S-R edge-triggered  
flip-flops



(f) D edge-triggered  
flip-flops



(g) J-K edge-triggered  
flip-flops



# FLIP-FLOPs and Latches



A flip-flop is a memory device with two stable states:

→ by an appropriate choice of inputs, either state can be obtained

Flip-flops are needed to execute sequential logic, which requires memory of past actions

Flip-flops come in many forms:

- we focus on those obtained by cross-coupling logic gates
- we will assume that all gates have the same gate delay  $t$

# Asynchronous Flip-Flops

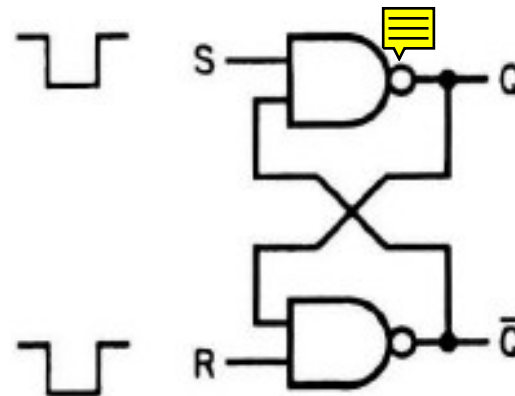
- Set-Reset Flip-Flop
- Arbitrator and metastability
- D Latch

# Set-Reset FLIP-FLOPs - asynchronous

Consider the asynchronous RS NAND flip-flop

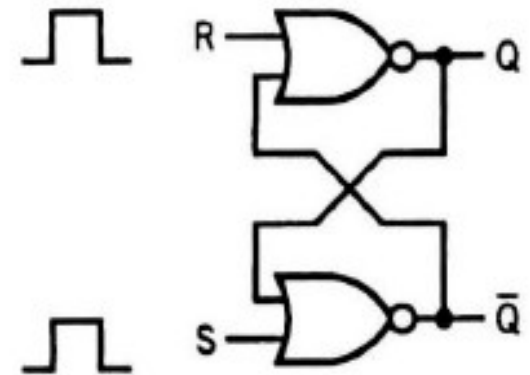
If the S(set) and R(reset) inputs are inactive ( $R = S = 1$ ) → the outputs Q and  $\bar{Q}$  can be in one of two self-consistent states:  
either  $Q = 0$  and  $\bar{Q} = 1$   
or  $Q = 1$  and  $\bar{Q} = 0$

active low



(a) RS NAND flip-flop

active high



(b) RS NOR flip-flop

- If S is made active (ie set to 0) → Q will become 1 after a time t (if it was not already 1) and  $\bar{Q}$  will become 0 after a time 2t. After this time, Q will remain at 1 even if S becomes inactive
- If R is made active (ie set to 0) for a time longer than 2t → Q can be set to 0

Note:

1) an analogous description is valid for the NOR flip-flop, where the only difference is that S and R are active when they are equal to 1

2) NAND FF is active low, NOR FF is active high

# Set-Reset FLIP-FLOPs - asynchronous

Consider the asynchronous RS NAND flip-flop

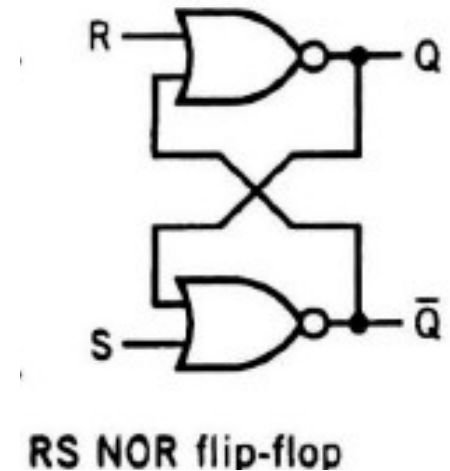
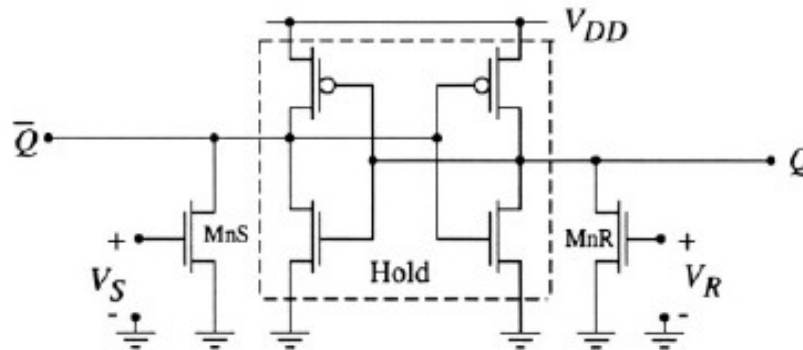
If the S(set) and R(reset) inputs are inactive ( $R = S = 1$ ) → the outputs Q and  $\bar{Q}$  can be one of two self-consistent state either  $Q = 0$  and  $\bar{Q} = 1$  or  $Q = 1$  and  $\bar{Q} = 0$

- If S is made active (ie set to 0) → Q will become 1 after a time t (if it was not already 1) and  $\bar{Q}$  will become 0 after a time 2t. After this time, Q will remain at 1 even if S becomes inactive
- If R is made active (ie set to 0) for a time longer than 2t → Q can be set to 0

Note:

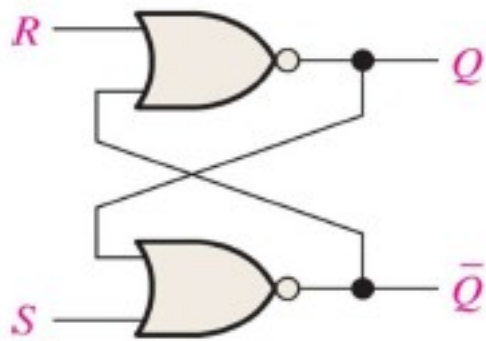
- 1) an analogous description is valid for the NOR flip-flop, where the only difference is that S and R are active when they are equal to 1
- 2) NAND FF is active low, NOR FF is active high

NOR SR Flip-Flop  
(Latch, simplified circuit)

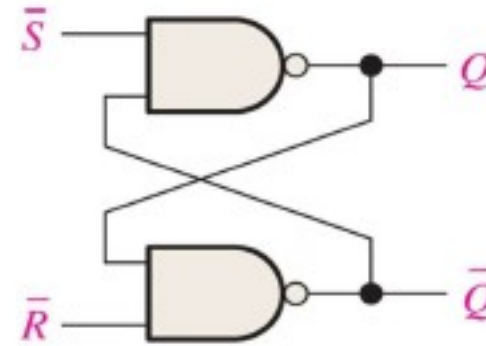


# SR FLIP-FLOPs (two versions)

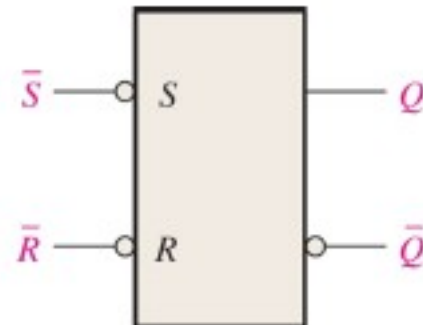
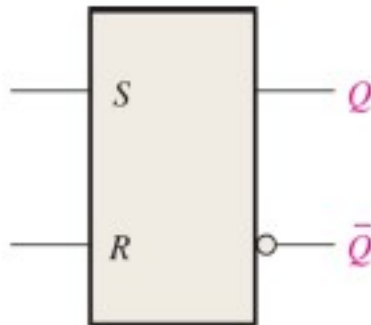
There are two versions of RESET-SET (R-S) Latch



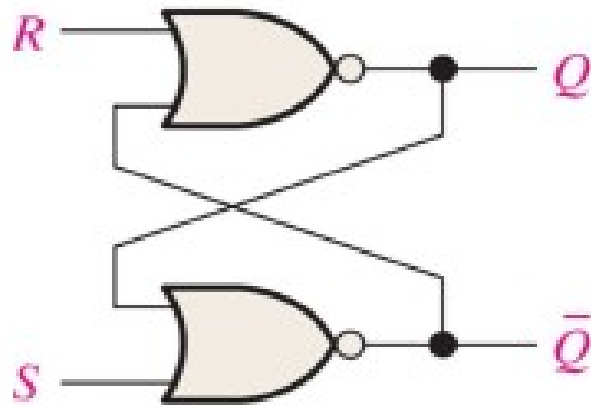
Active HIGH



Active LOW

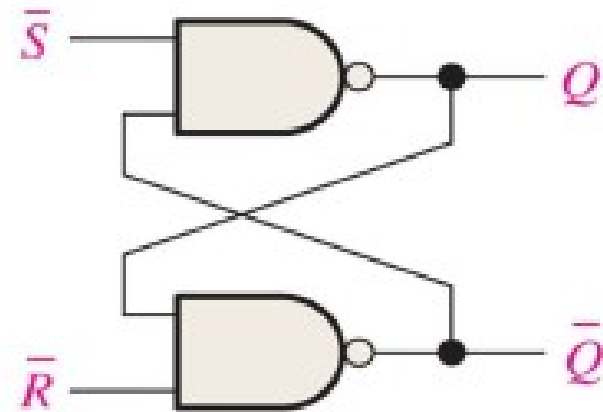


## “Old Q” / “New Q” Analysis



(a) Active-HIGH input S-R latch

Input		Output
S	R	$Q_{\text{new}}$
0	0	$Q_{\text{old}}$
0	1	0
1	0	1
1	1	0

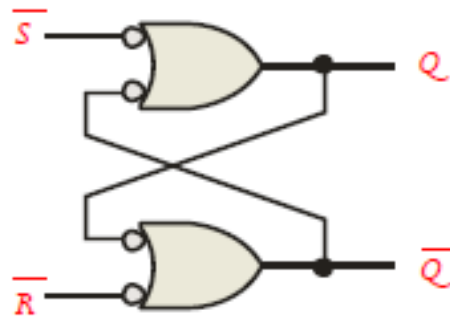


(b) Active-LOW input  $\bar{S}$ - $\bar{R}$  latch

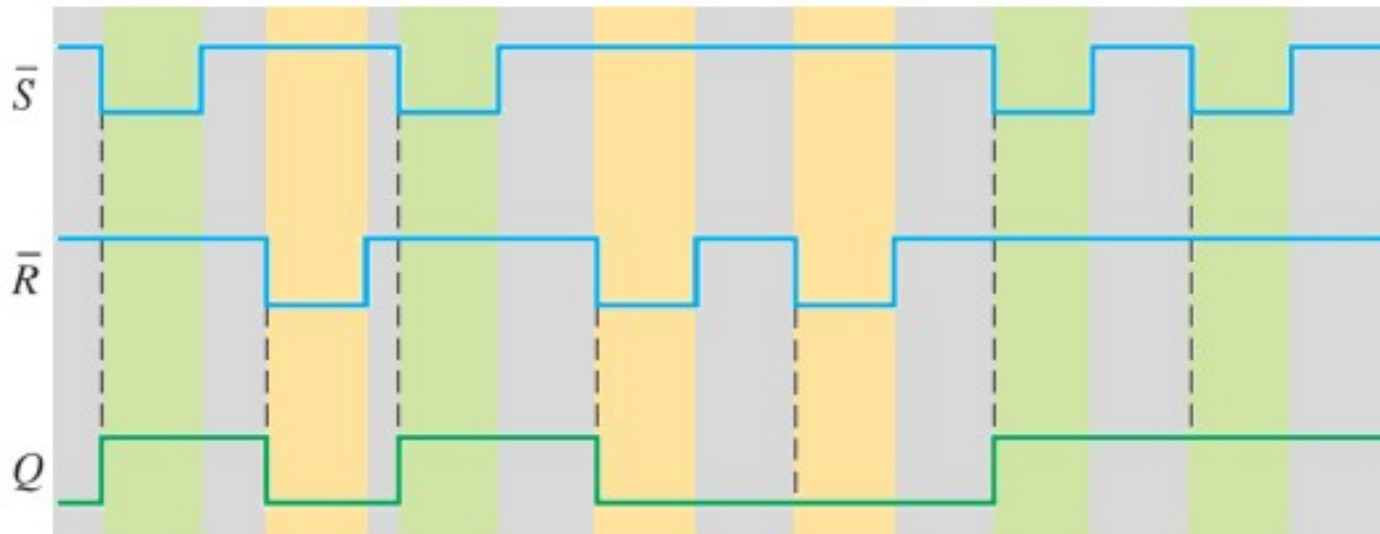
Input		Output
$\bar{S}$	$\bar{R}$	$Q_{\text{new}}$
0	0	1
0	1	1
1	0	0
1	1	$Q_{\text{old}}$

# Example

NAND type → active low

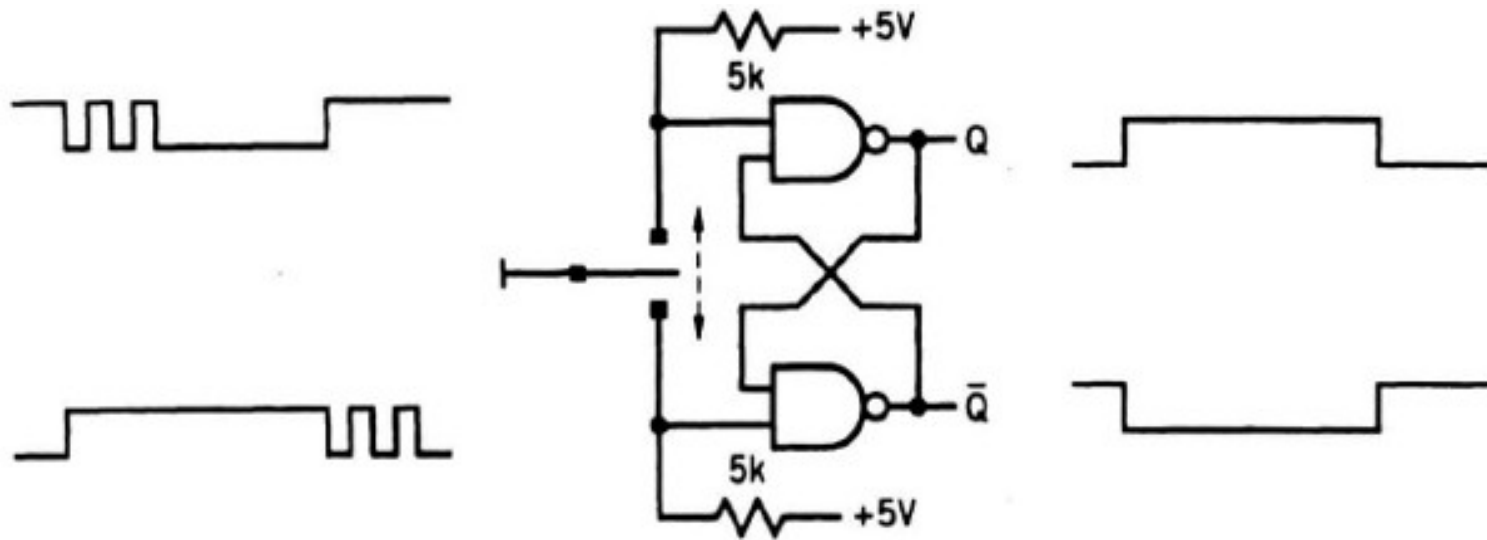


Input s		Mode
$\bar{S}$	$\bar{R}$	
0	1	SET
1	0	RESET
1	1	HOLD





## Example: RS latch debouncer

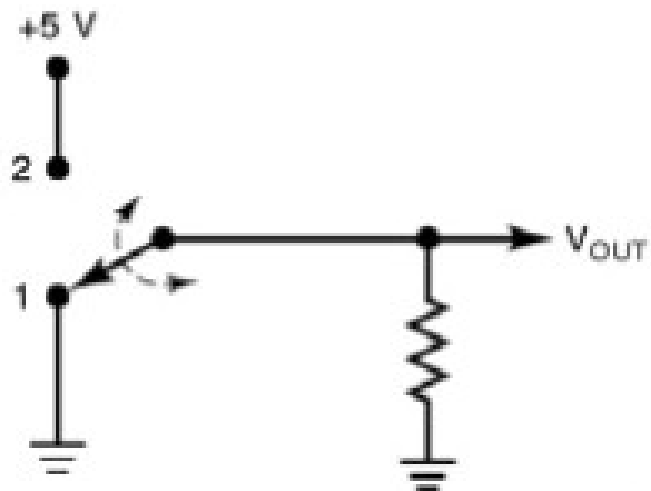


TTL RS flip-flop can be used in a [debouncer](#) to generate a single transition for instance when a mechanical single-pole double-throw switch changes state

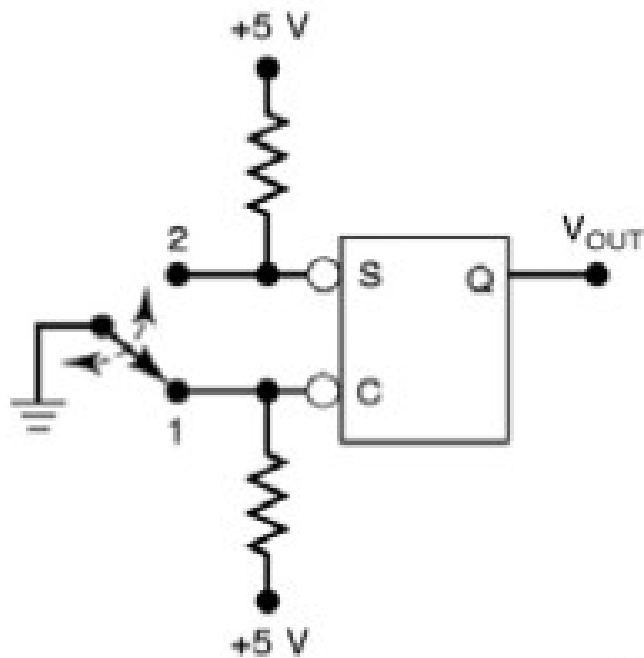
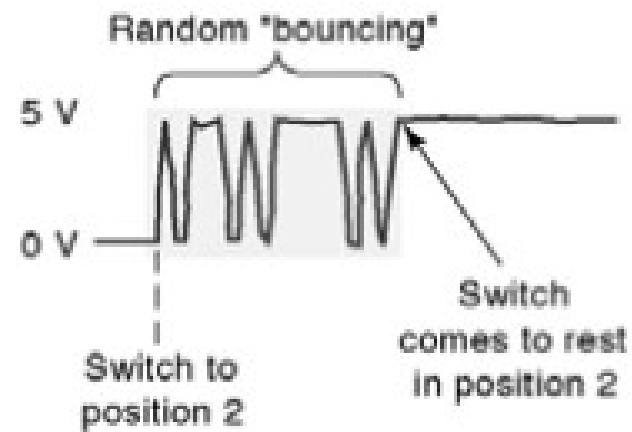
The contacts bounce many times when the switch is thrown toward one pole or the other

The duration of the first closure of the contacts, however, is far larger than any possible gate delay, so that the [flip-flop is sure to change state](#). Furthermore, the bounce is [never so severe that contact is made with the opposite pole](#)

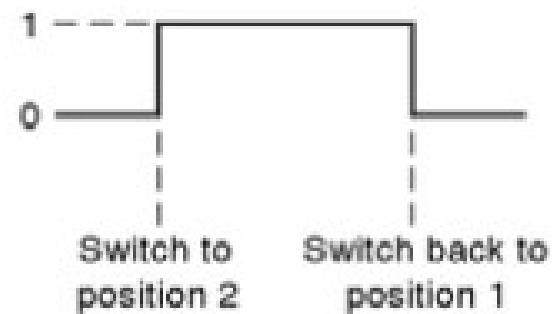
## Example: RS latch debouncer



(a)



(b)



# Set-Reset FLIP-FLOPs – Arbitrator

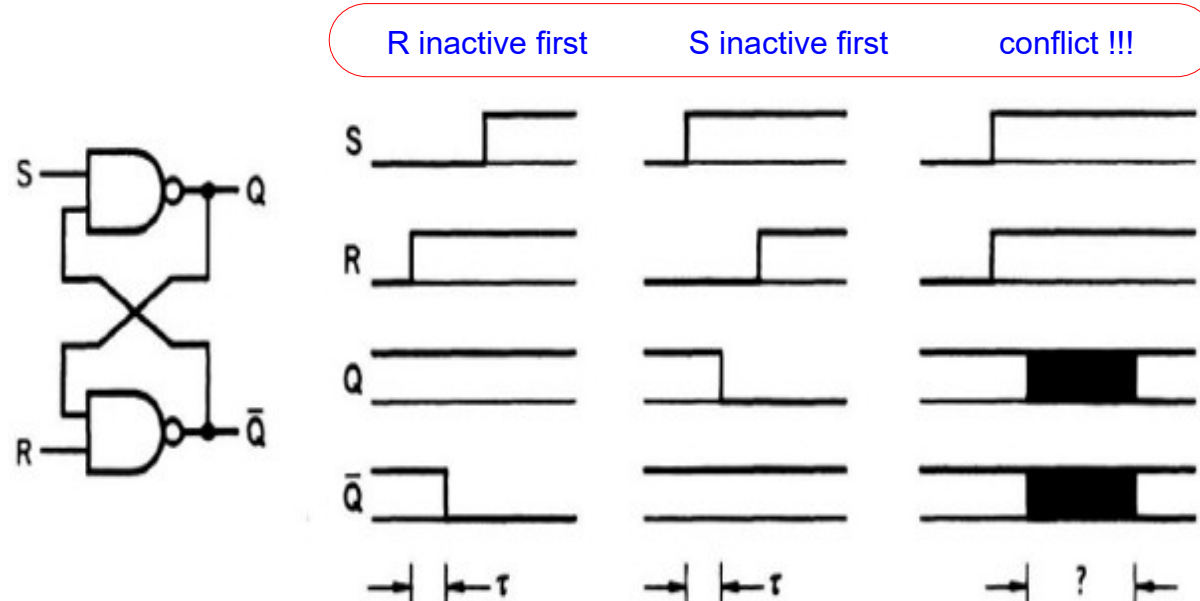
Consider the asynchronous RS NAND flip-flop

- If both the S(set) and R(reset) **inputs are inactive** ( $R = S = 1$ )  
→ the outputs Q and  $\bar{Q}$  can be in one of **two self-consistent states**:  
either  $Q = 0$  and  $\bar{Q} = 1$   
or  $Q = 1$  and  $\bar{Q} = 0$

- If **both inputs are held active** ( $R = S = 0$ ) → **both outputs are equal to 1**  
The flip-flop registers which input first becomes inactive and leaves it in a proper state (“**arbitrator**” function)

Example: if we consider S as a sampling signal when it goes to 1, and we then inquire about the state of the flip-flop, we can establish the value of R at the time of sampling (see clocked flip-flops, later)

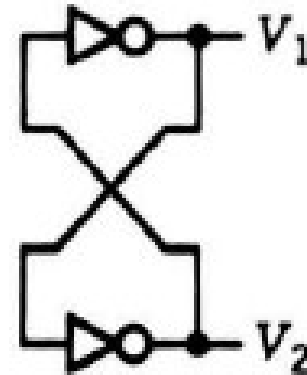
... BUT there is a **problem**: the arbitrator works well if R and S become inactive at times that are **separated by at least one gate delay**: indeed the output that goes to 0 arrives in time to confirm the 0 at the input of the gate that remains at 1. For shorter intervals we must be careful... let us take a closer look at the structure of RS flip-flops then ...



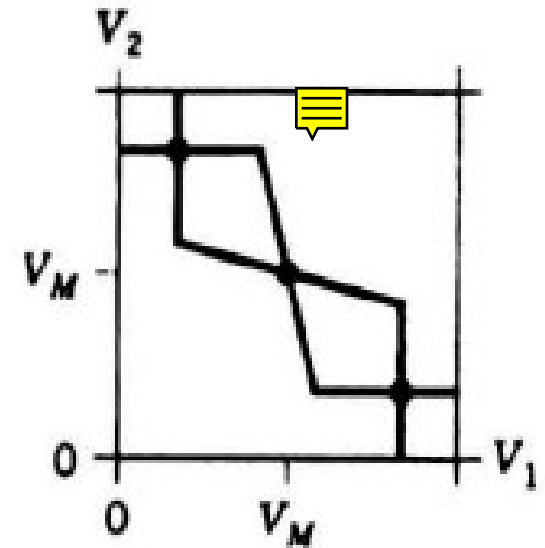
# Set-Reset FLIP-FLOPs - Metastability

When both inputs of an RS flip-flop are inactive, the flip-flop reduces to a pair of cross-coupled inverters

!!! there is an additional **third non-digital equilibrium state** where both the output voltages values between LOW and HIGH  
If the inverters are identical their output voltages are equal ( $V_1 = V_2 = V_M$ )



Cross-coupled inverters



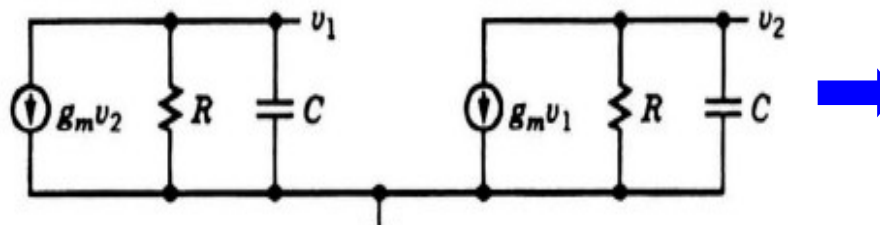
Static transfer characteristics  
The equilibrium point at  $V_1 = V_2 = V_M$  is metastable

This state is **metastable**: any departure from equilibrium drives the flip-flop toward one of the two stable states: this happens because the inverters are on the steep part of their voltage transfer characteristic when they are near the metastable state and thus have a large incremental gain; **since the feedback is positive, their small-signal response is unstable**



# Set-Reset FLIP-FLOPs – Metastability in detail

Assume that the inverters have a first-order response  $\rightarrow$  small signal circuit



$$\begin{aligned} v_1 + RCv_1' &= -g_m R v_2 \\ v_2 + RCv_2' &= -g_m R v_1 \end{aligned} \quad \rightarrow \quad \begin{aligned} (1 + g_m R)(v_1 + v_2) + RC(v_1 + v_2)' &= 0 \\ (1 - g_m R)(v_1 - v_2) + RC(v_1 - v_2)' &= 0 \end{aligned}$$

(assuming  $1 \ll g_m R$ )  $\rightarrow$

$$\begin{aligned} v_1 + v_2 &\approx V_+ \exp(-\omega_T t) \\ v_1 - v_2 &\approx V_- \exp(+\omega_T t) \end{aligned}$$

with  $\omega_T = g_m/C$  gain-bandwidth product of the inverters  
and  $V_+$ ,  $V_-$  being constants

It is clear that  $v_1$  and  $v_2$  will grow exponentially if they differ initially in the least degree, and that a large gain-bandwidth product will result in a quick exit from the metastable state. These conclusions hold for higher-order gains

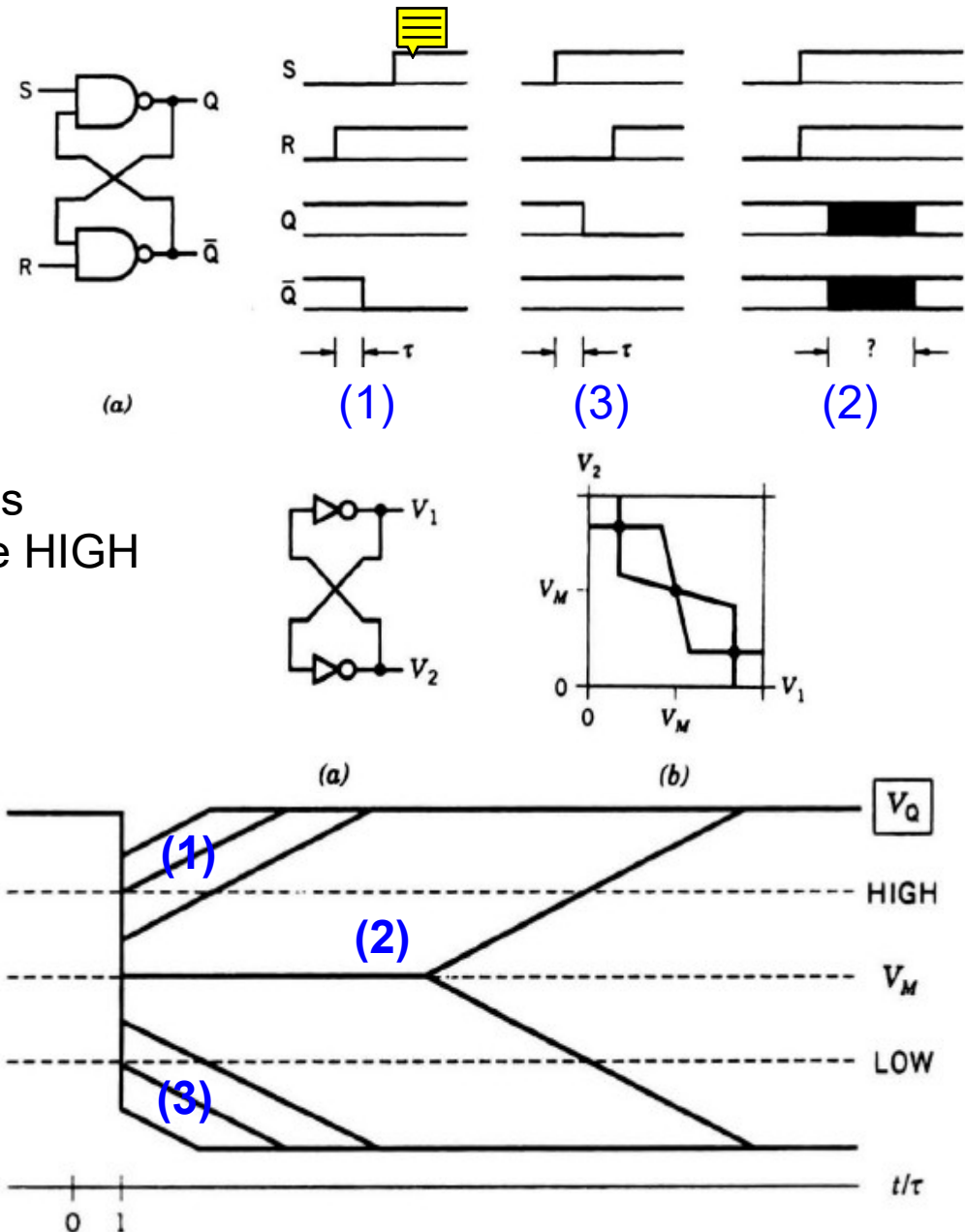
# Set-Reset FLIP-FLOPs – Arbitrator in detail

Back to the arbitrator: let us imagine to **slowly shorten the interval** between the times when R and S become inactive

(1) If we now make **R precedes S** by **less than one gate delay**, nothing critical happens for some time because even though Q drops slightly, it remains above HIGH. Beyond a certain point, however, Q will drop below HIGH for longer times although it will eventually return to a value above HIGH

(2) When the **interval is reduced to zero**, Q and  $\bar{Q}$  are left in the metastable state.

(3) If we now make **S precede R** by an **interval shorter than one gate delay**, Q will eventually drop below LOW, but it will take longer and longer to do so as the interval is shortened



Waveforms of the voltage at the Q output

# Set-Reset FLIP-FLOPs – Arbitrator in detail

→ there exists a metastability window  $\tau_w \sim O(\tau)$  such that if S and R are separated by less than  $\tau_w$  the flip-flop will become metastable (ie Q will not settle above HIGH or below LOW within one gate delay  $\tau$  after S becomes inactive)

Now, if we assume that R goes from LOW to HIGH and back with an average frequency  $f$ , the probability that the flip-flop will become metastable when S becomes inactive is  $f\tau_w$

Let us define  $\Pi(T)$  as the probability that the flip-flop is still metastable a time  $T$  after becoming metastable. The probability  $P(T)$  that the flip-flop is still metastable a time  $T+\tau$  after S becomes inactive is the “probability of synchronization failure” or of “arbitration conflict” and is given by

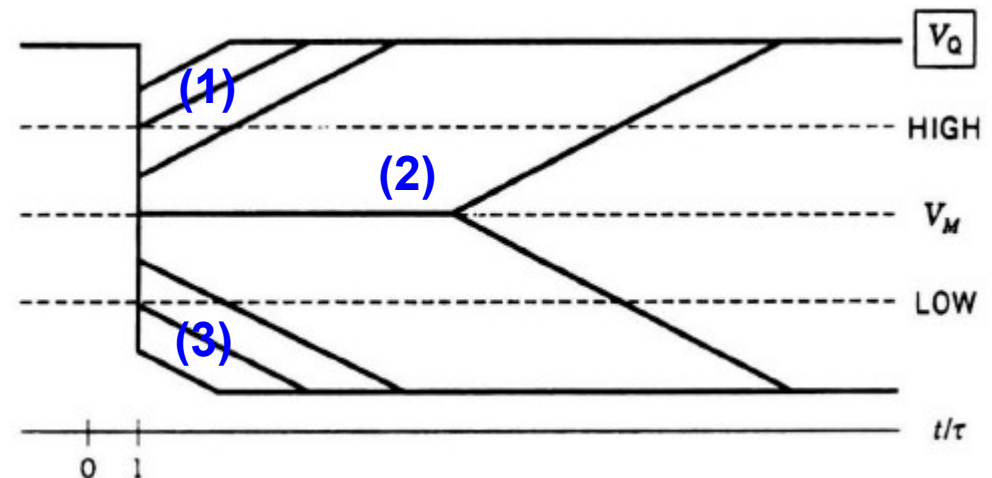
$$P(T) = f\tau_w \Pi(T)$$

$P(T)$  cannot be reduced to zero in an RS flip-flop, but it can be made arbitrarily small by waiting long enough before examining Q because in practice,  $\Pi(T)$  decreases exponentially for large  $T$

$$\Pi(T \gg \tau) \sim \exp(-T/\tau_M)$$

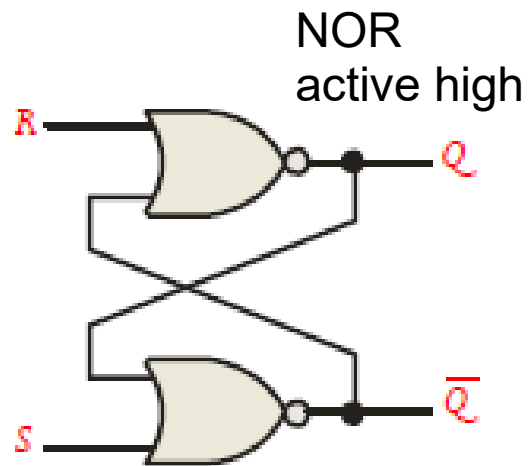
Note:

- 1) choice of  $T$  not easy ... failure to find good solution can result in catastrophes
- 2) synchronization failure can be avoided internally in clocked (or gated) systems



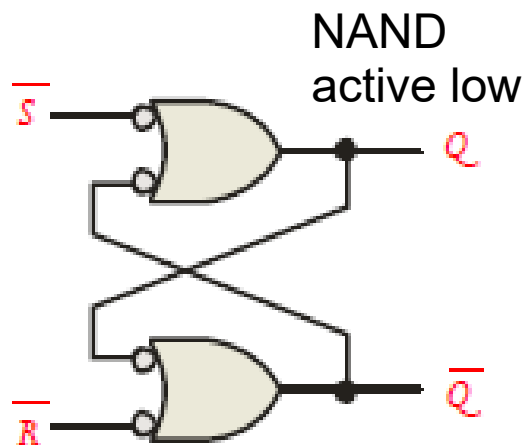
Waveforms of the voltage at the Q output

# SR Flip-Flop – Short summary



Input s		Output s		Mode of Operation	Comment
S	R	Q	$\bar{Q}$		
0	0	NC	NC	Hold	No change.
0	1	0	1	Reset	For RESET ting Q to 0
1	0	1	0	Set	For SET ting Q to 1
1	1	0	0	Prohibited	Invalid Condition

both active

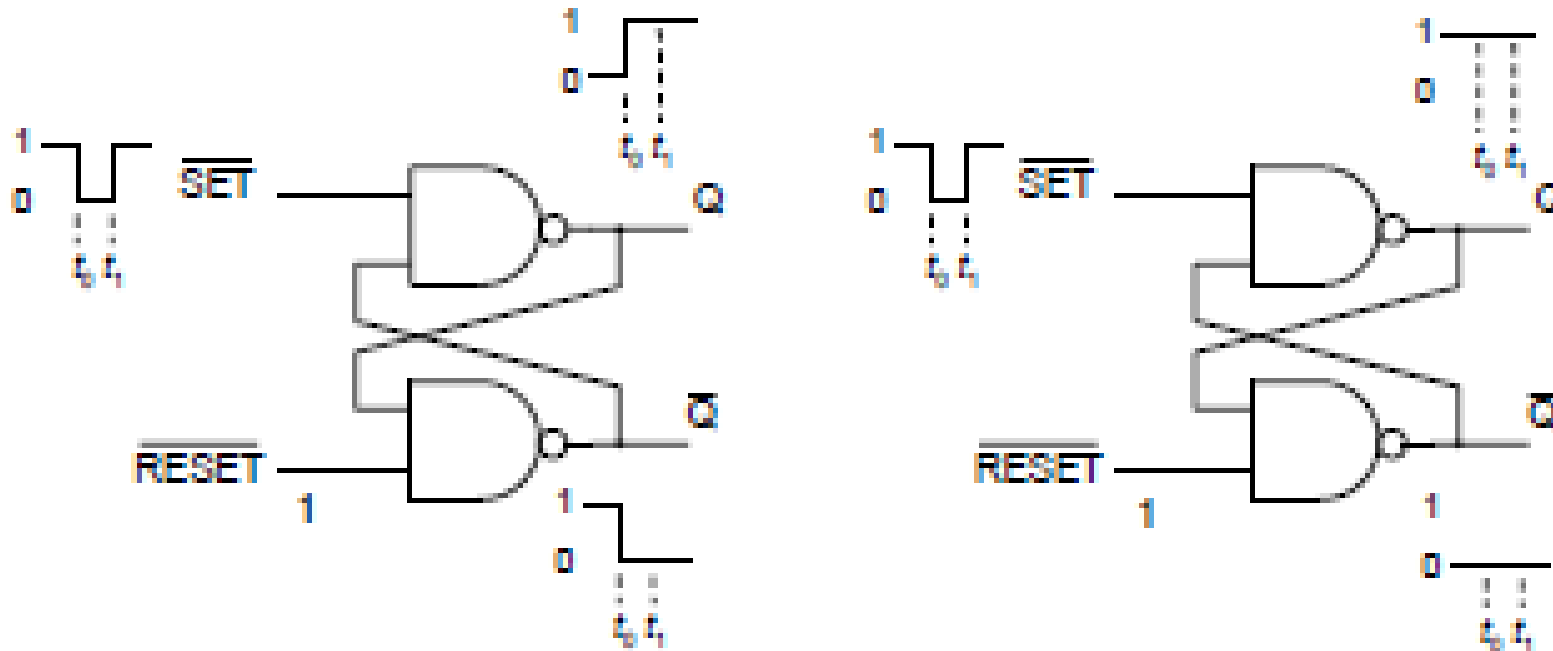


Inputs		Outputs		Mode of Operation	Comment
$\bar{S}$	$\bar{R}$	Q	$\bar{Q}$		
0	0	1	1	Prohibited	Invalid Condition
0	1	1	0	Set	For SETting Q to 1
1	0	0	1	Reset	For RESETting Q to 0
1	1	NC	NC	Hold	No change.

both active

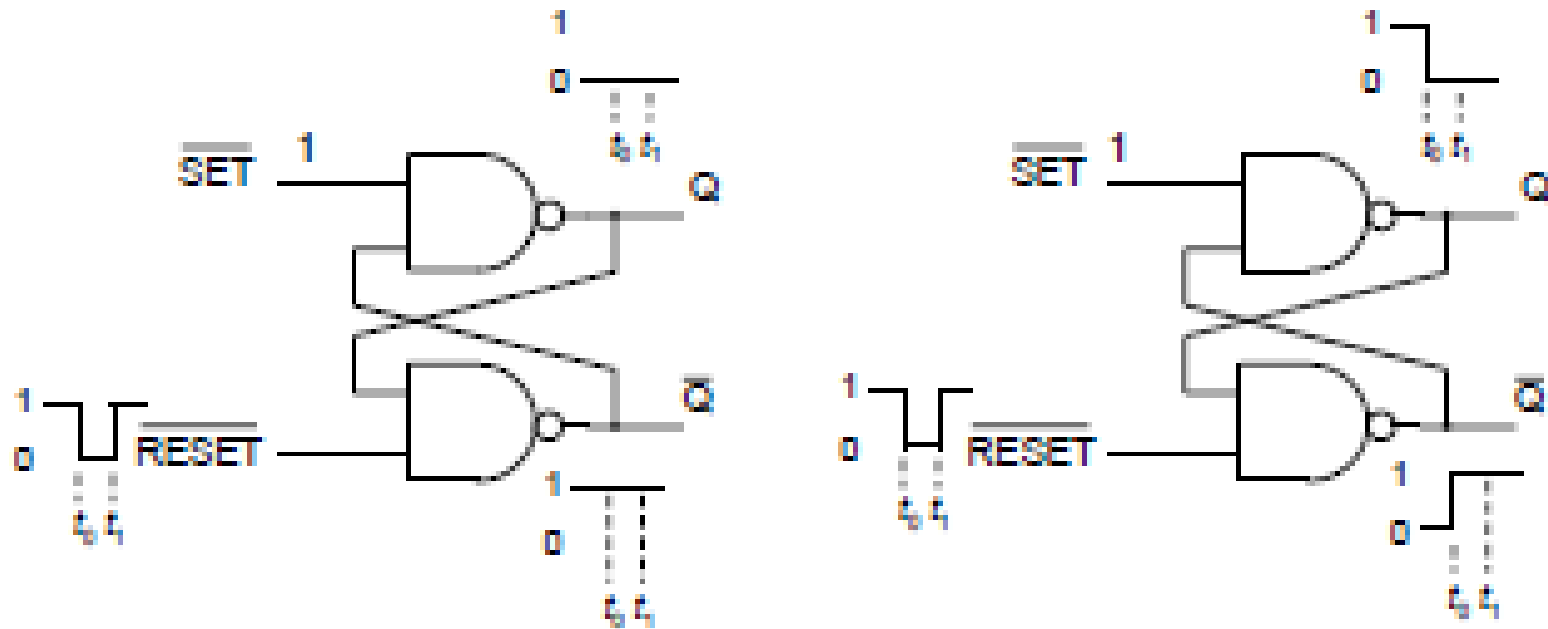


## SR Latch – Set operation



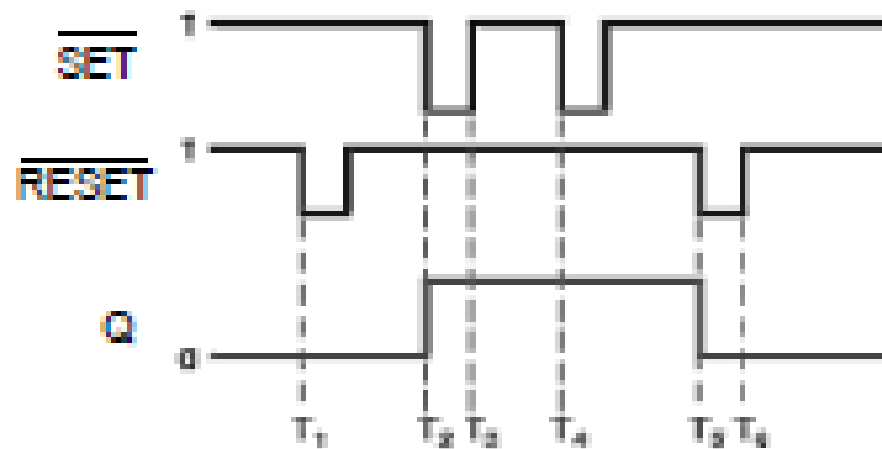
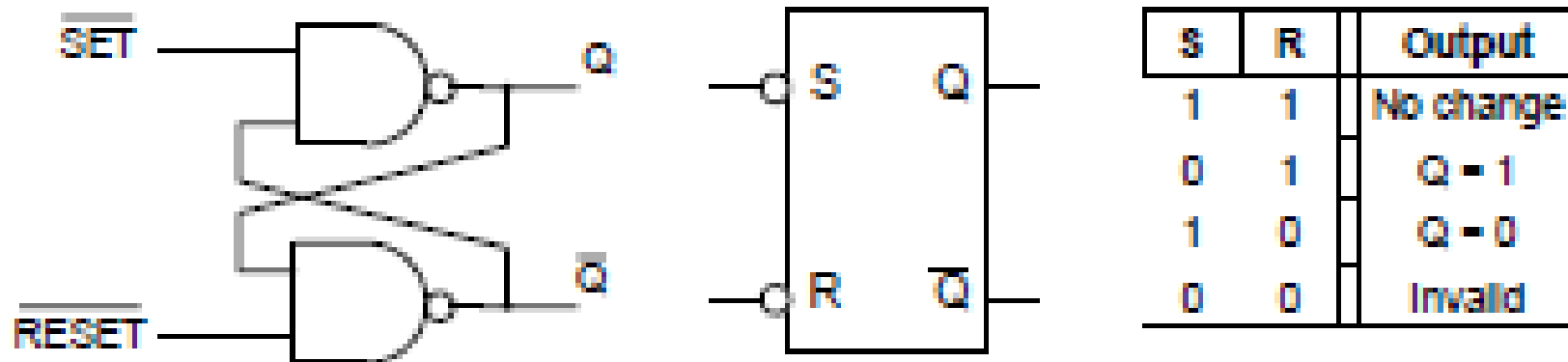
A negative pulse on  $\overline{SET}$  puts the latch in HIGH (SET) state

## SR Latch – ReSet operation



A negative pulse on  $\overline{\text{RESET}}$  puts the latch in LOW (RESET) state

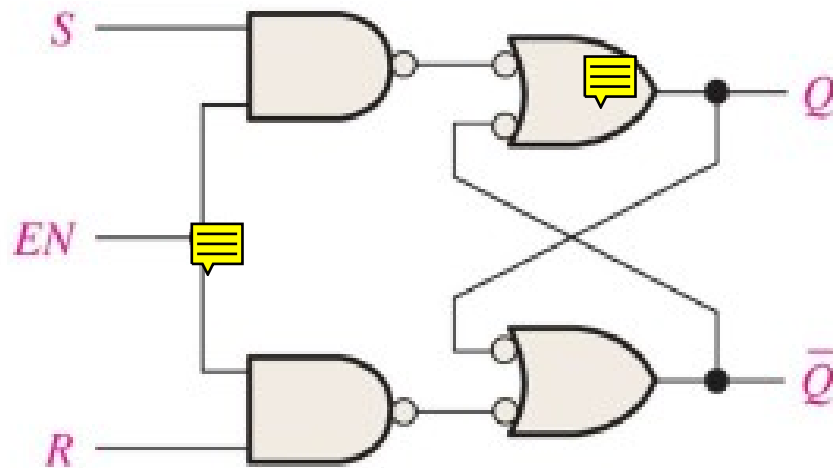
# SR Latch – Set-Reset operation & Truth table



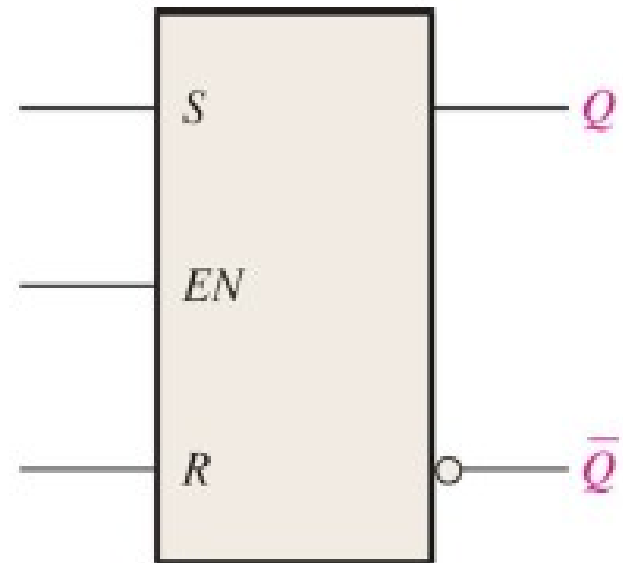
# Gated Latch

A **gated latch** is a variation on the basic flip-flop

The gated latch has an **additional input, called enable (EN)** that must be HIGH in order for the latch to respond to the S and R inputs (active LOW)



(a) Logic diagram



(b) Logic symbol

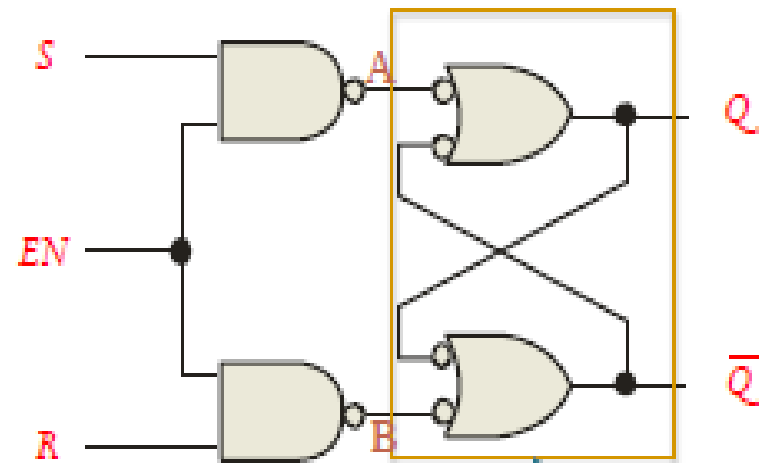
# Gated Latch

Observe that:

$$A = \overline{S \cdot EN} = \overline{S} + \overline{EN}$$

$$B = \overline{R \cdot EN} = \overline{R} + \overline{EN}$$

EN	A	B
0 $\Rightarrow$	1	1
1 $\Rightarrow$	$\overline{S}$	$\overline{R}$

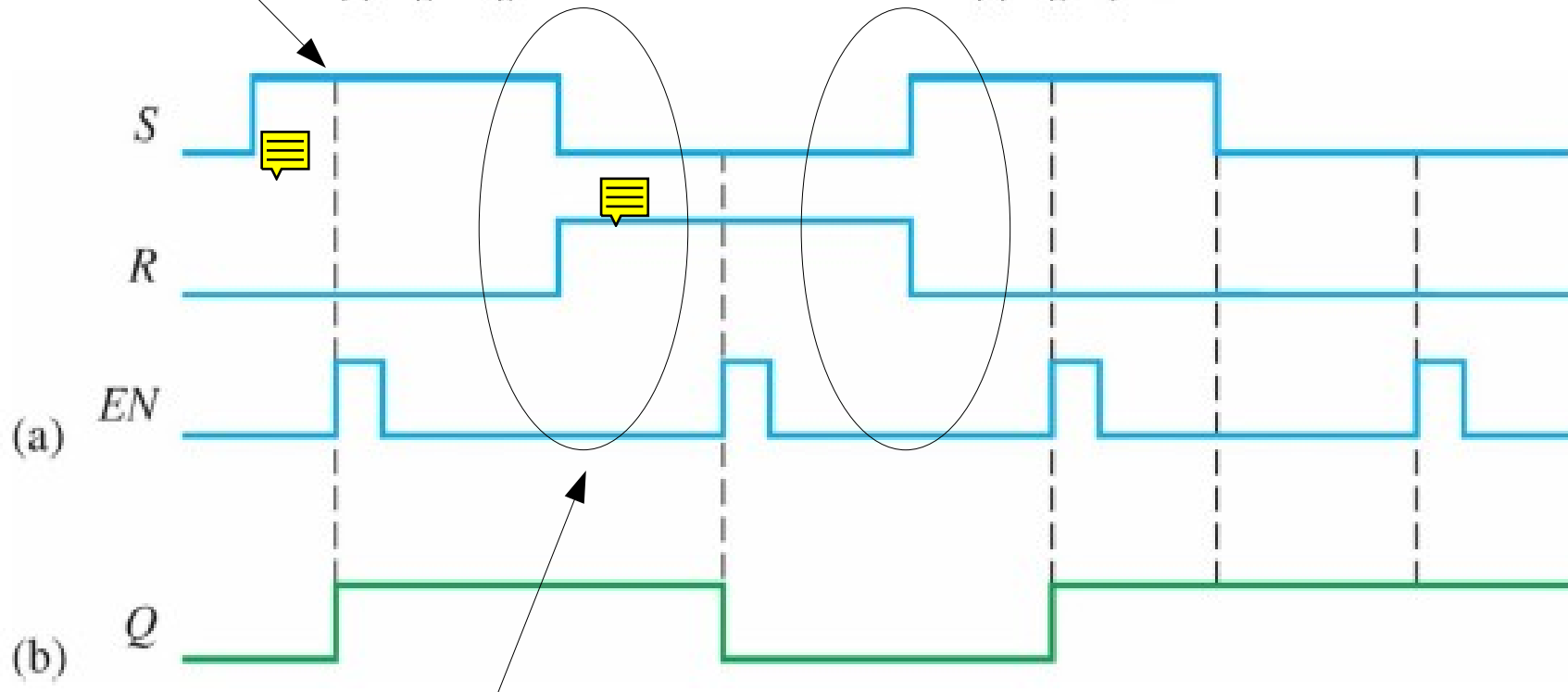
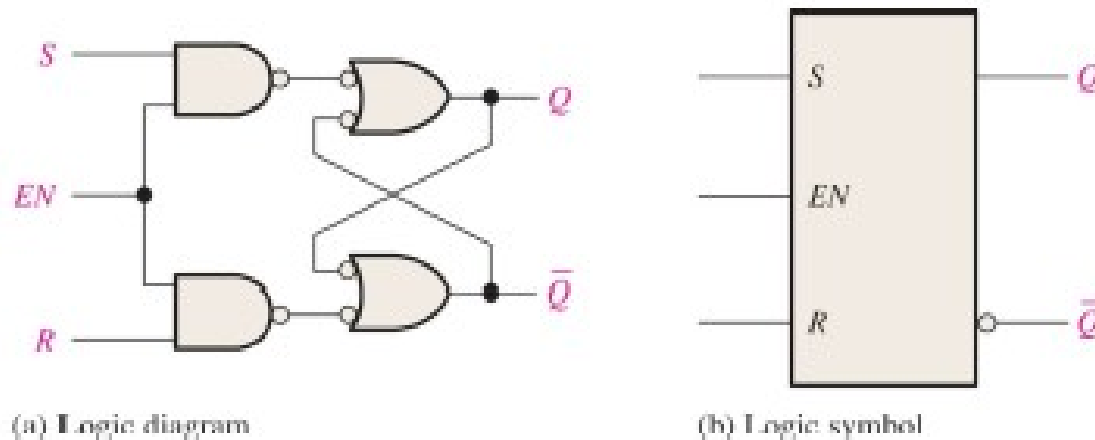


This is the same as  
the active-LOW  
input latch!

# Example - Gated S-R Latch

Note:  
**Latch** is active  
on **levels**

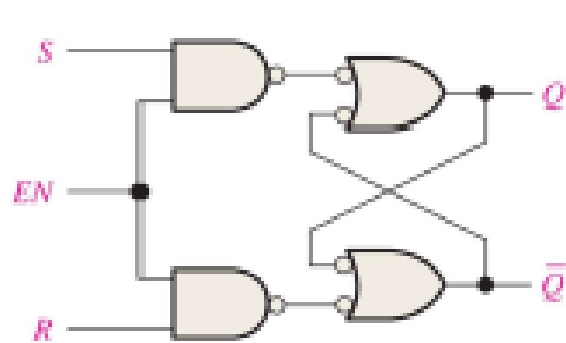
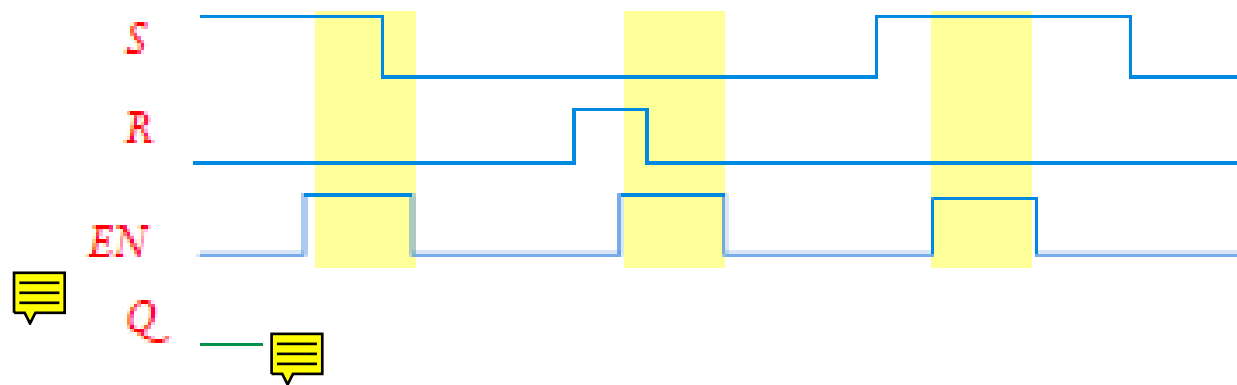
(... **Flip-flop**  
on **edges**)



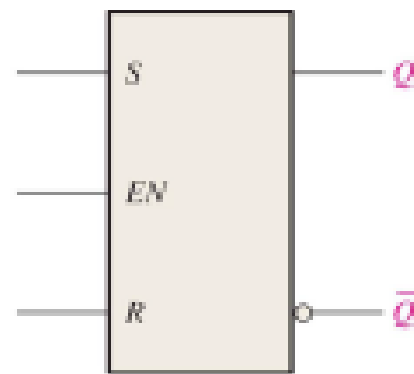
Note: **Arbitration conflicts can be avoided**

## Exercise - Gated S-R Latch

Show the Q output with relation to the input signals. Assume Q starts LOW.



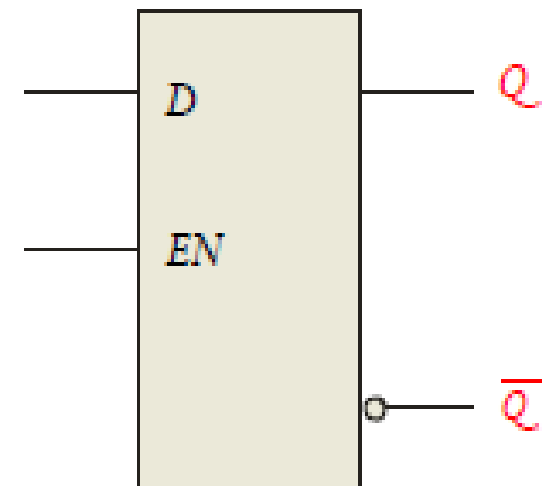
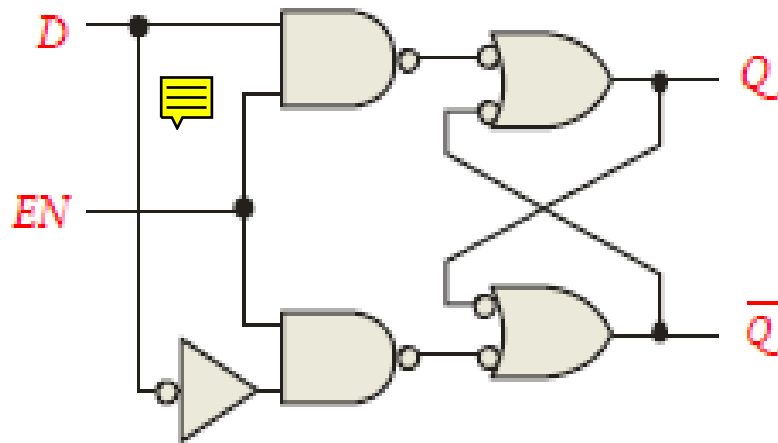
(a) Logic diagram



(b) Logic symbol

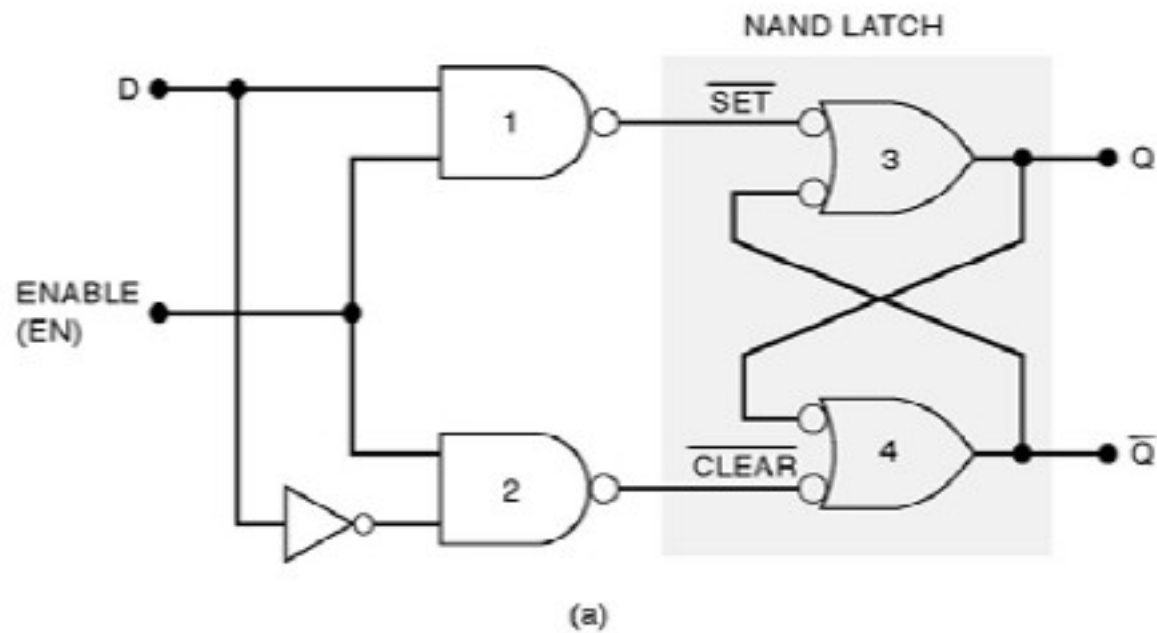
## (Gated) D-Latch – Transparent Latch

- The D latch is a variation of the S-R latch.
- Has only one input in addition to EN.
  - This input is called the D (data) input.
- Combine the S and R inputs into a single D input.





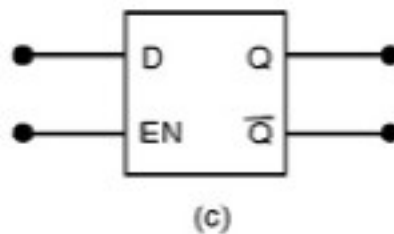
# Gated D Latch



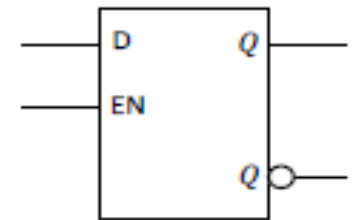
Inputs		Output
EN	D	Q
0	X	$Q_0$ (no change)
1	0	0
1	1	1

\*X\* indicates 'don't care'  
 $Q_0$  is state Q just prior to EN going LOW

(b)



# Gated D Latch



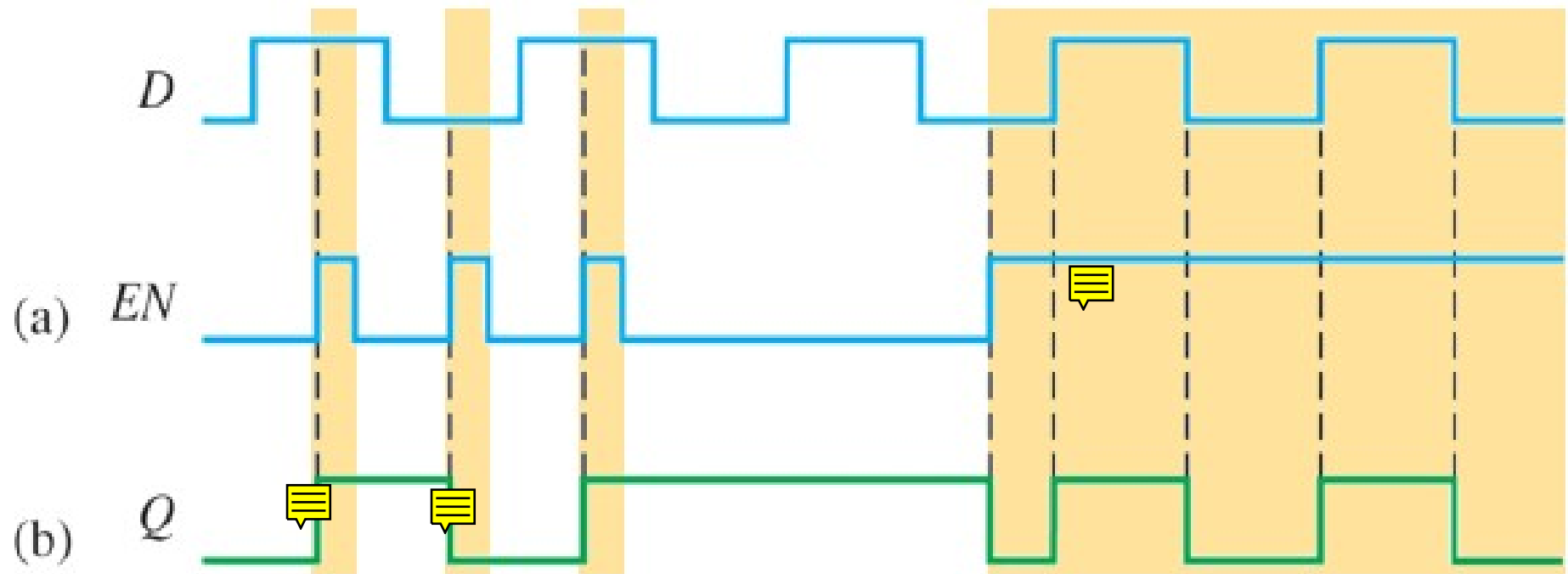
- A simple rule for the D latch is:
  - Q follows D when the Enable is active / asserted.
    - In this situation, the latch is said to be “open” and the path from D input to Q output is “transparent”.
    - The circuit is often called a transparent latch for this reason.
- When EN is LOW, the state of the latch is not affected by the D input.
  - In this situation, the latch is said to be “close”
  - The Q output retains its last value and no longer changes in response to D, as long as EN remains negated.
- Output is “latched” at the last value when the enable signal becomes not asserted.

- Truth Table: →

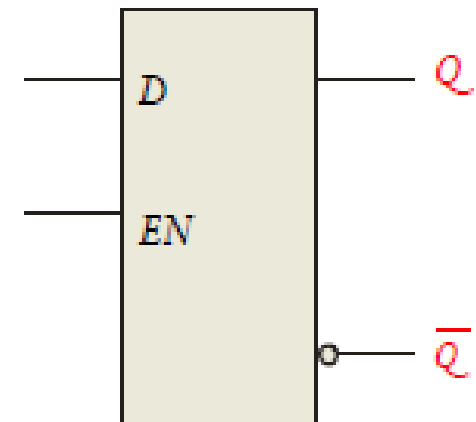
$Q_0$  is the prior output level before the indicated input conditions were established.

Inputs		Outputs		Comments
$D$	$EN$	$Q$	$\bar{Q}$	
0	1	0	1	RESET
1	1	1	0	SET
X	0	$Q_0$	$\bar{Q}_0$	No change

## Example - Gated D Latch

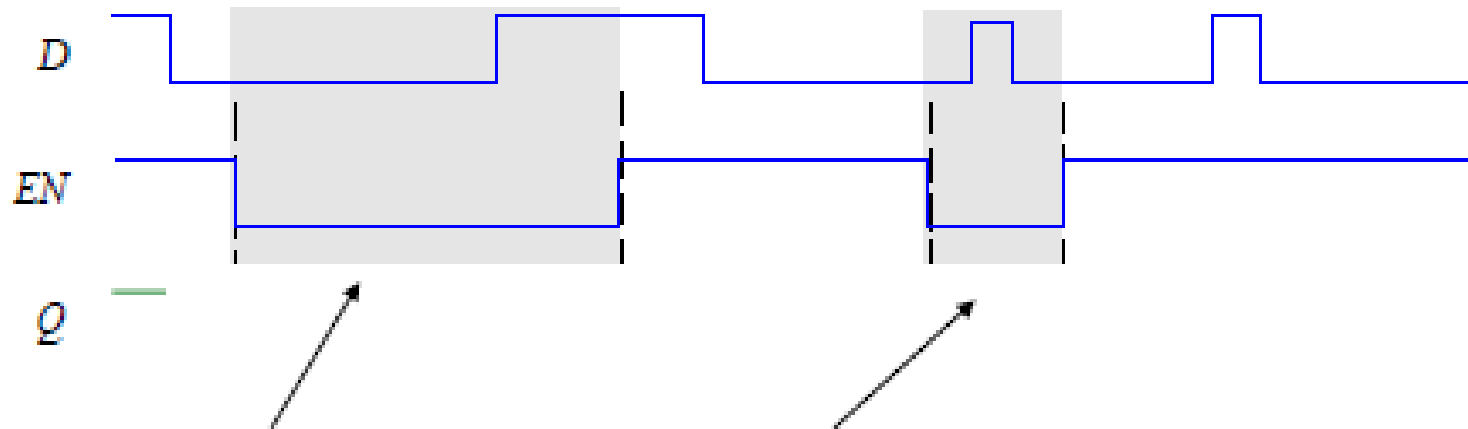
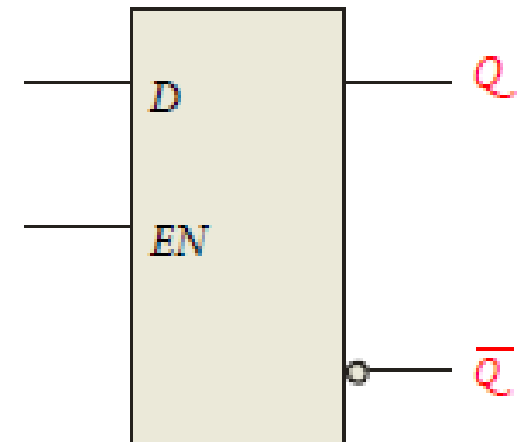


$Q$  follows  $D$  when the Enable is active.



## Exercise - Gated D Latch

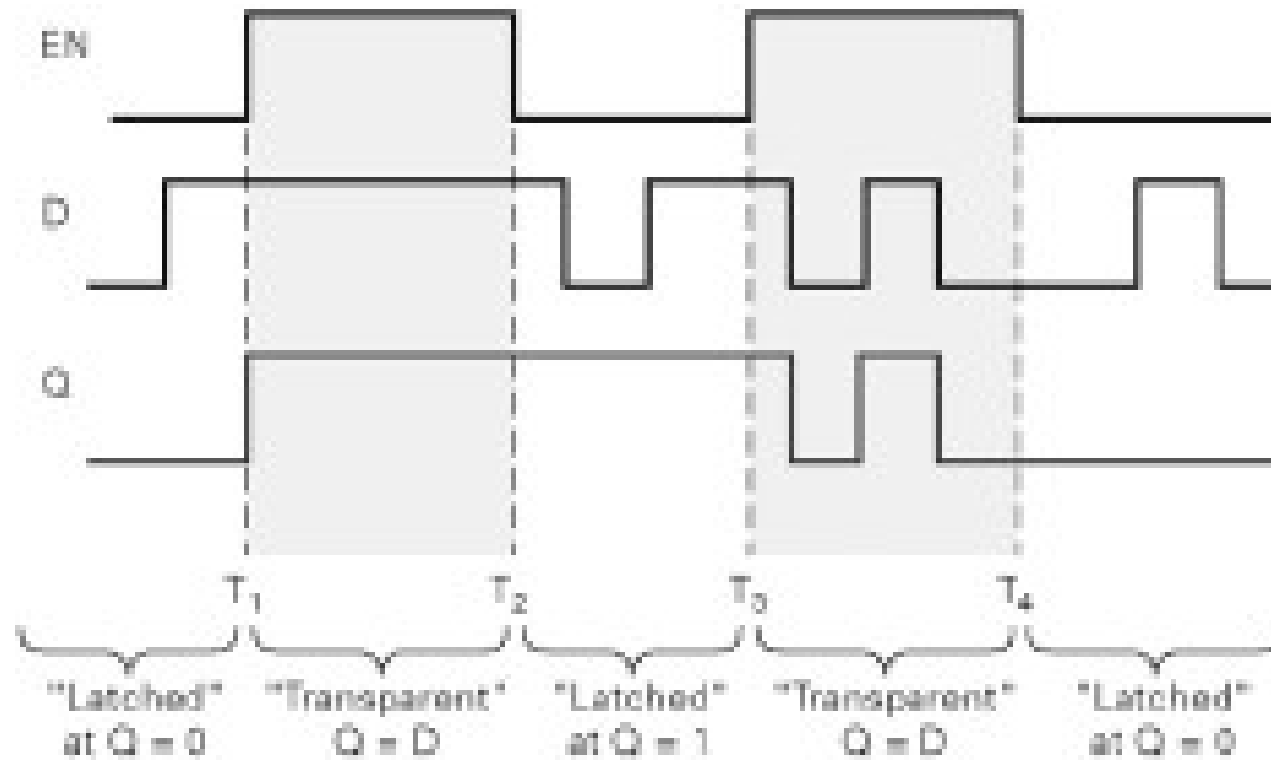
Determine the Q output for the D latch, given the inputs shown.



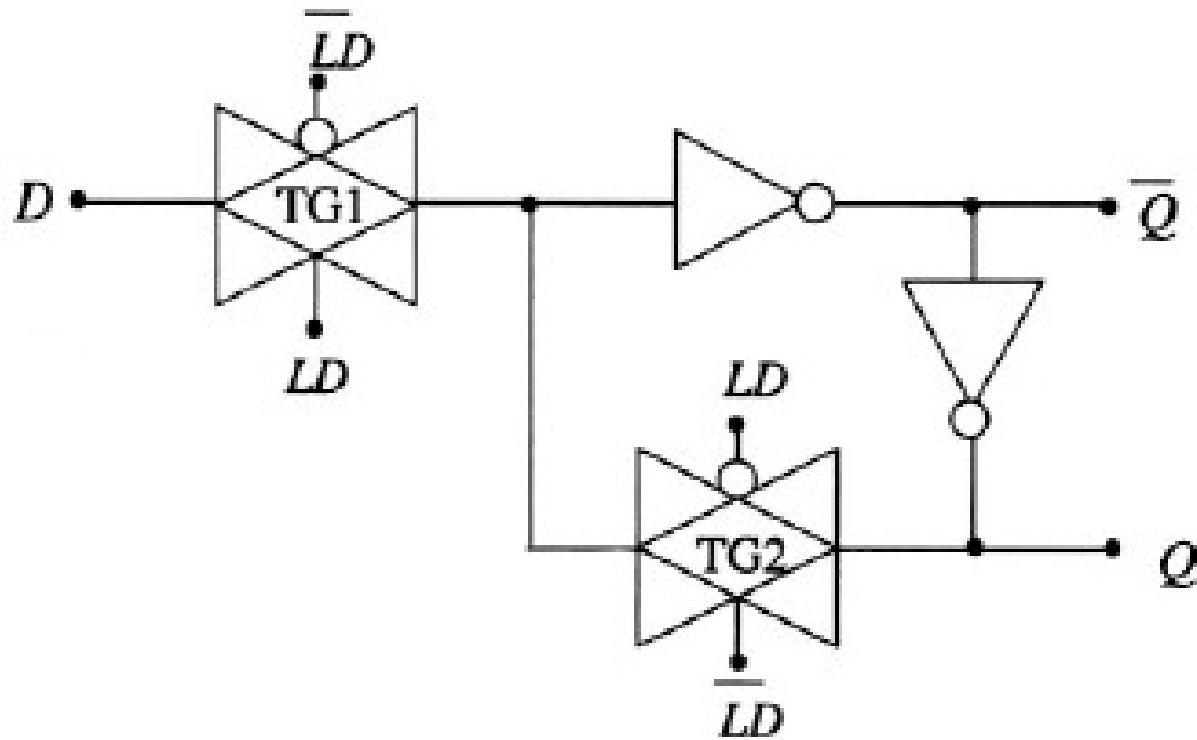
Notice that the Enable is not active during these times, so the output is latched.

# Gated D Latch timing

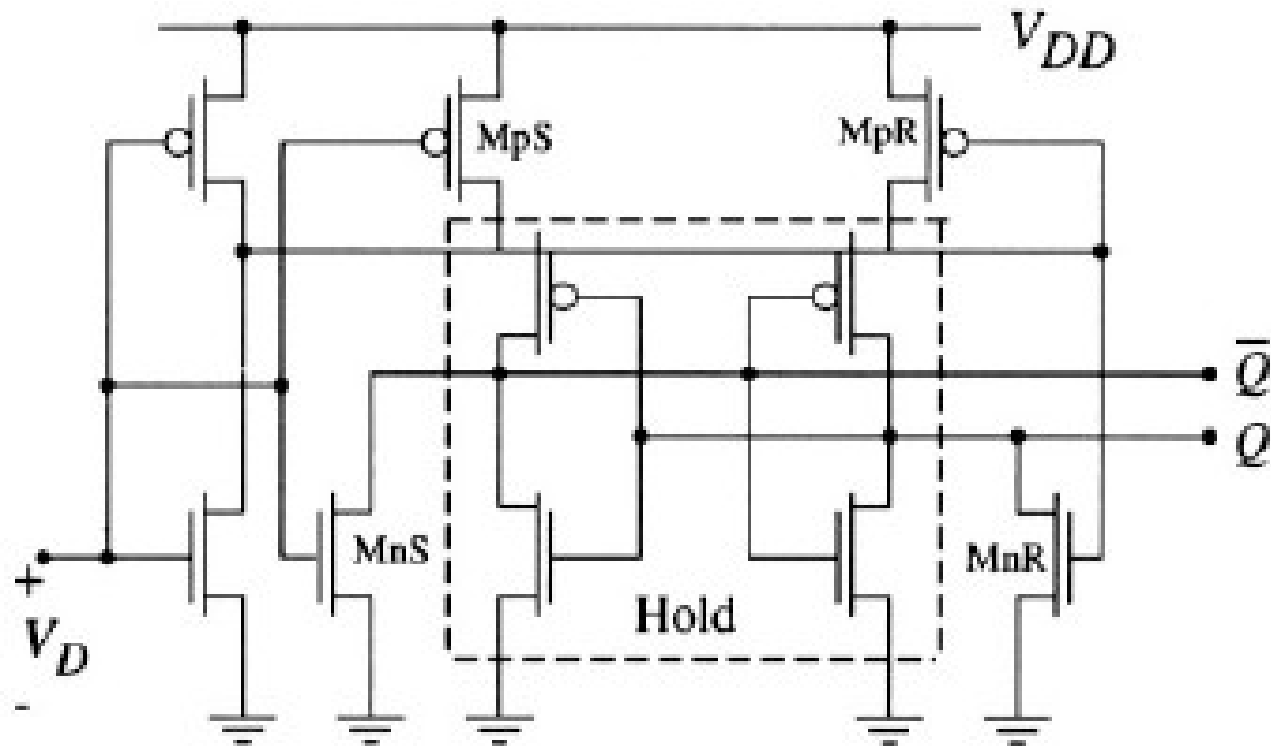
## D latch timing



## Gated D Latch circuit – Pass-Transistor Logic



## Gated D Latch circuit - CMOS



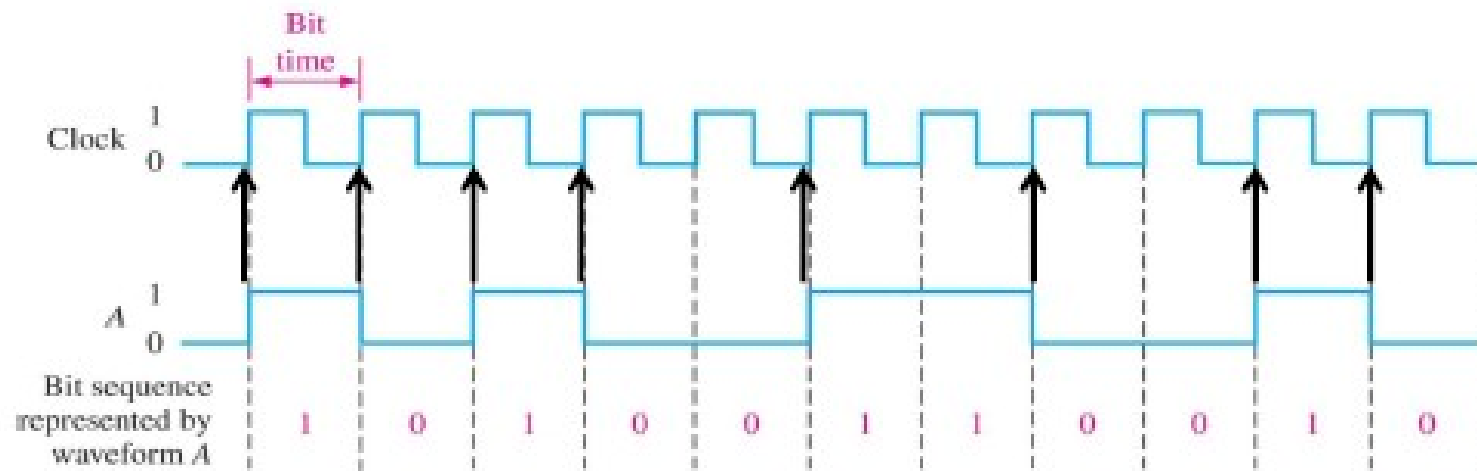
# Synchronous Flip-Flops

- Clocked Flip-Flops and Synchronous systems
- Edge-Triggered Flip-Flops
- D Flip-Flops
- Shift Registers
- JK Flip-Flop
- Counters



# Clocks (CLK)

- In digital synchronous systems, all waveforms are synchronized with a clock.
  - The clock waveform itself does not carry information.
- The **clock** is a periodic waveform in which each interval between pulses (the period) equals the time for one bit.



- Notice that change in level of waveform A occurs at the **rising edge** of the clock waveform.

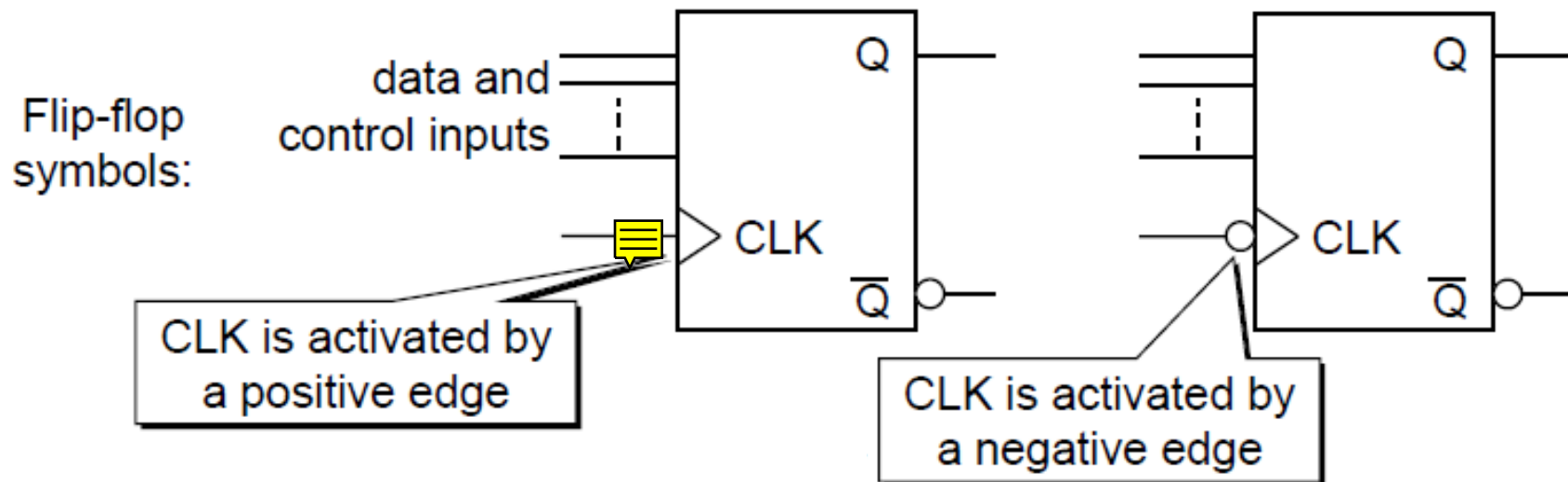
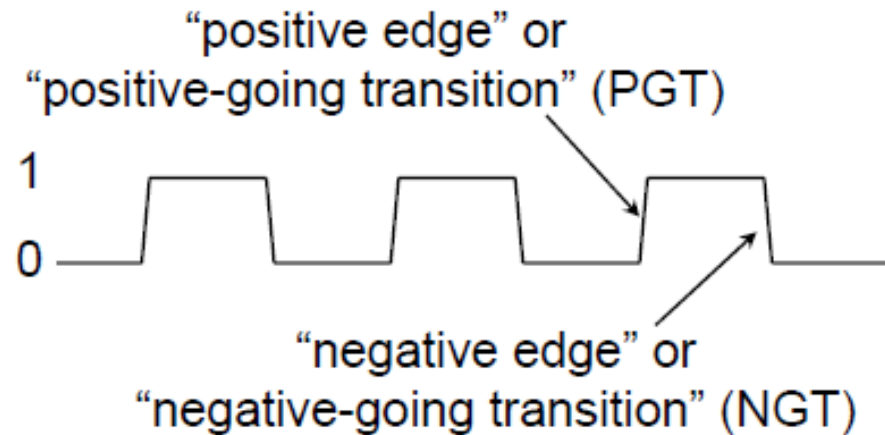
## Note on “async” clocks

In some cases simple square pulses (either in regular trains or even in irregular sequences) can be used for synchronization ...

# Clock signals and Flip-Flops

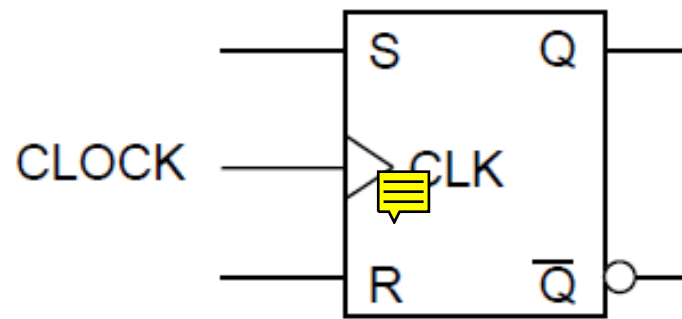
Synchronous digital systems:

- the state bits of the circuit all change simultaneously
- the changes occur at fixed points in time
- the control signal which indicates it is time to change is called the **clock**



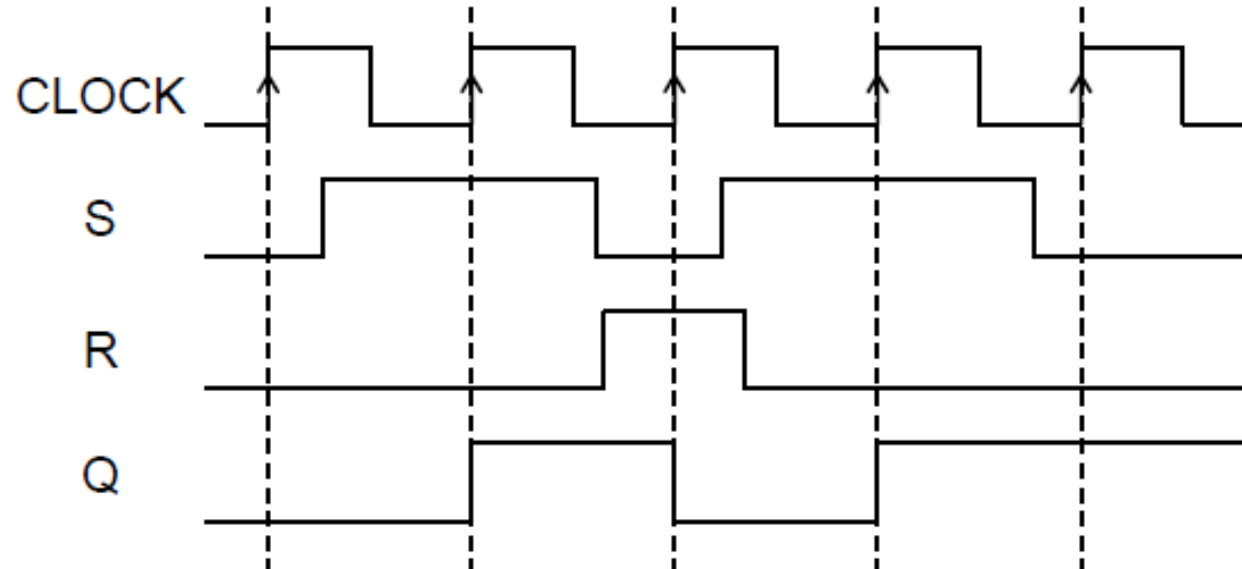
# Set-Reset Flip-Flop

**Clocked flip-flop:**  
the output only  
changes at the  
positive edges of  
the clock



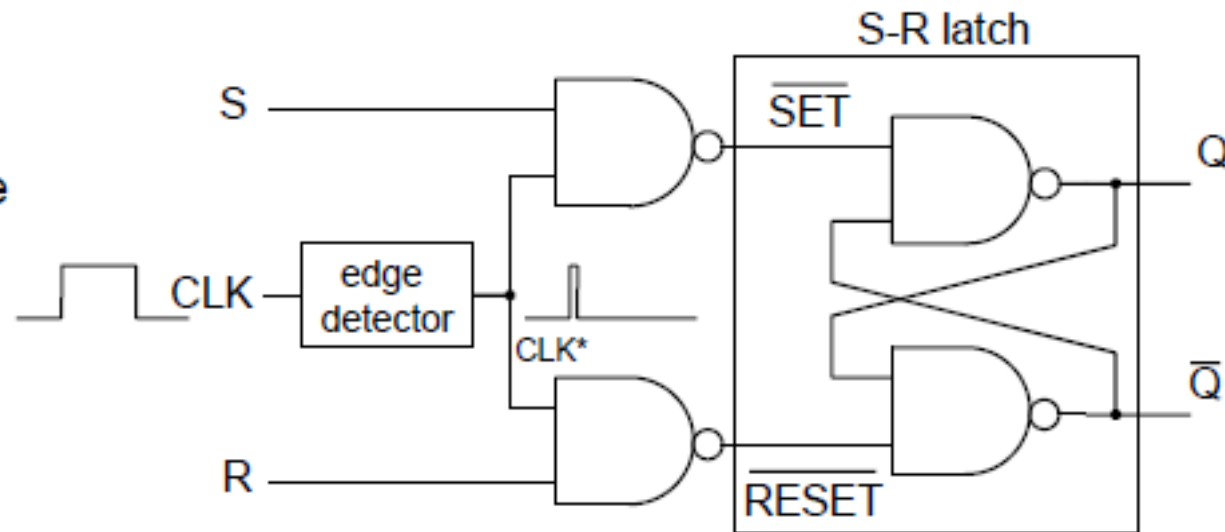
S	R	CLK	Output
0	0	↑	No change
1	0	↑	Q = 1
0	1	↑	Q = 0
1	1	↑	Invalid

Flip-flops are  
sometimes called  
**edge-triggered**

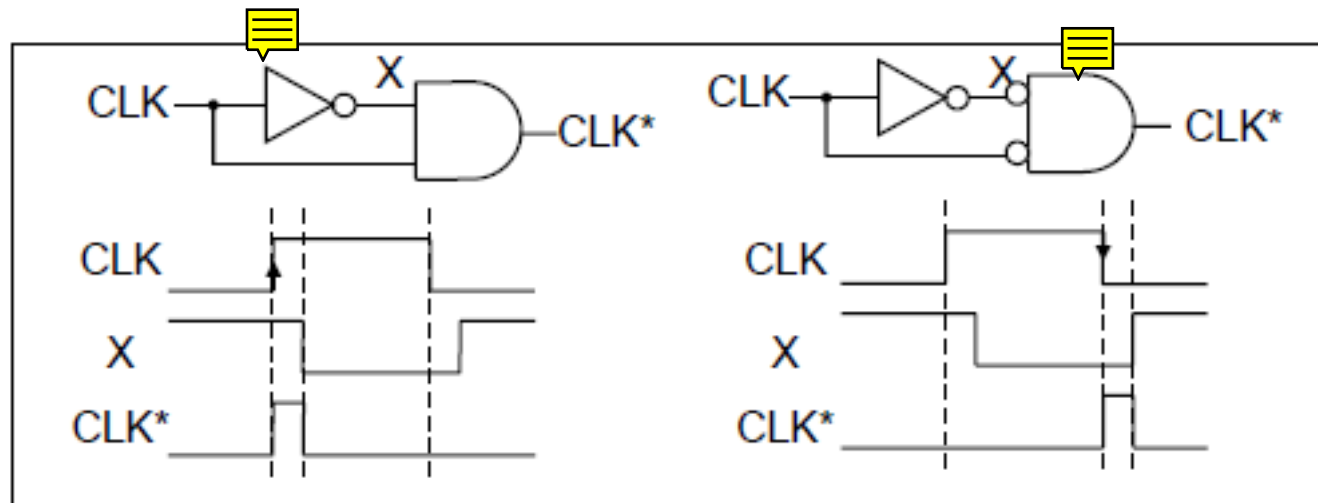


# Set-Reset Flip-Flop – internal circuitry

This is a simplified diagram of the inside of a S-R flip-flop



Edge detector circuits for both positive-edge and negative-edge flip-flops

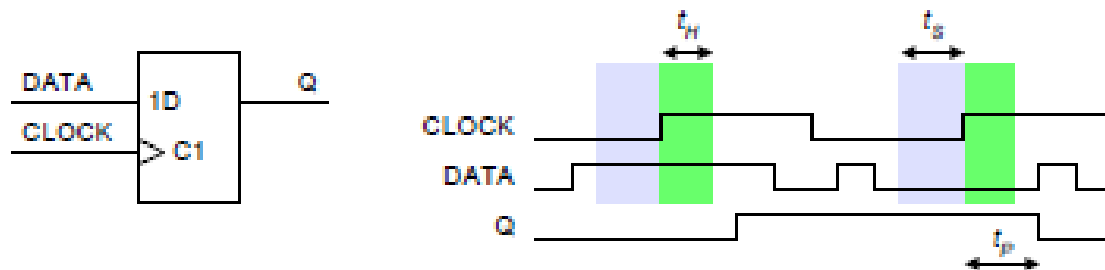


# Clocked Flip-Flops – time constraints

Synchronization failure is avoided in **clocked digital systems** (**synchronous**) by

- **gating** all flip-flops with an (ideally) square-wave clock signal C and
- demanding that **inputs** other than the clock be kept steady for a **setup time**  $t_s$  before a clock transition that might cause a metastable condition
- demanding that inputs remain steady for a **hold time**  $t_H$  after the clock transition

The DATA input to a flipflop or register must not change at the same time as the CLOCK.



Typical values for a register:  $t_s = 5 \text{ ns}$ ,  $t_H = 3 \text{ ns}$

The setup and hold time define a window around each CLOCK  $\uparrow$  edge within which the DATA must not change.

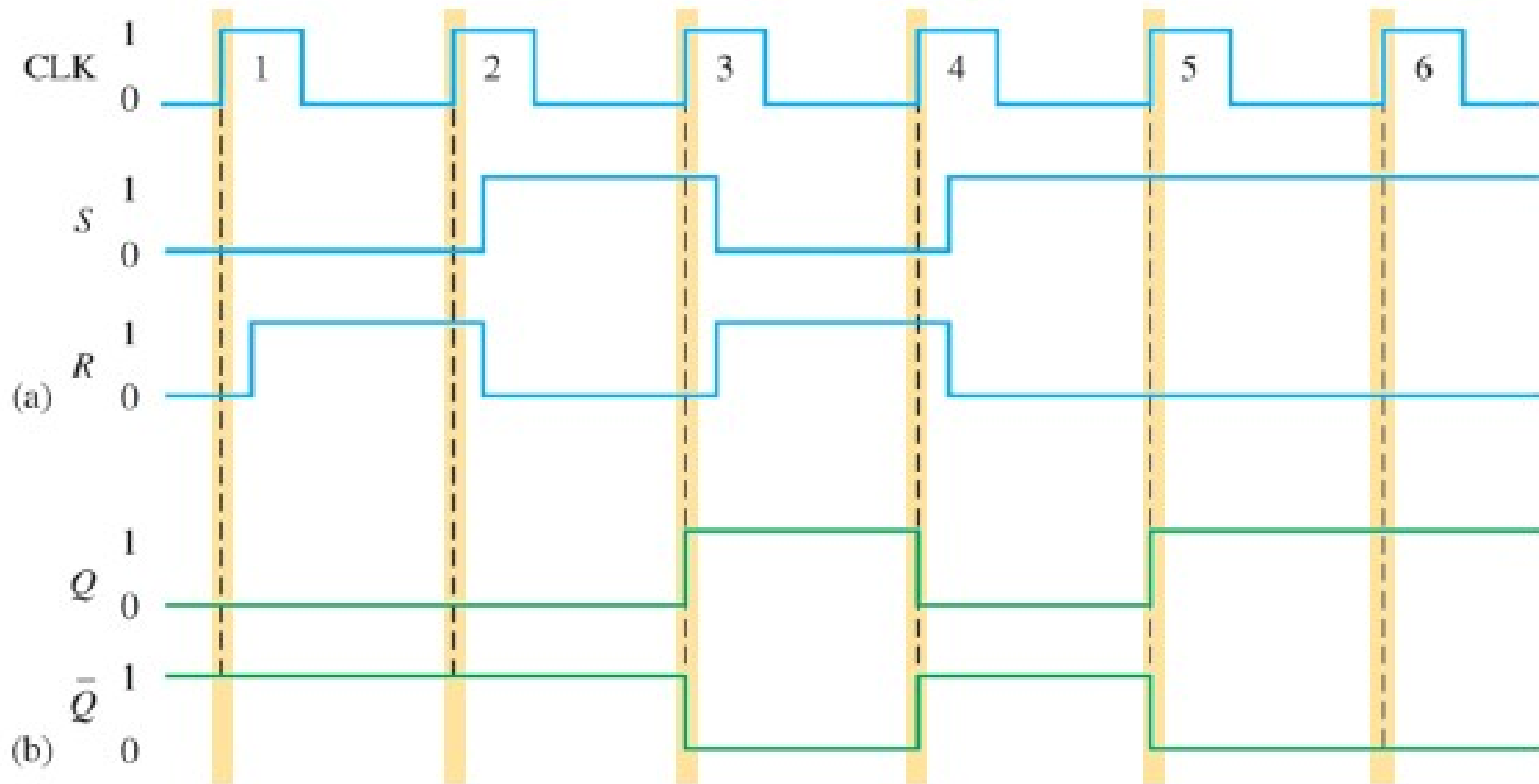
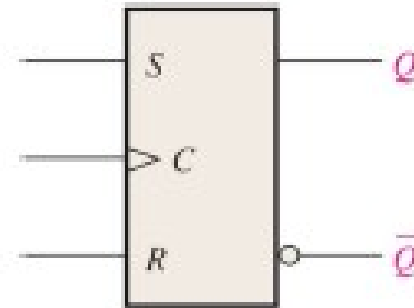
If these requirements are not met, the Q output may oscillate for many nanoseconds before settling to a stable value.

**Setup Time:** DATA must reach its new value at least  $t_s$  before the CLOCK  $\uparrow$  edge.

**Hold Time:** DATA must be held constant for at least  $t_H$  after the CLOCK  $\uparrow$  edge.

# S-R type Flip-Flop

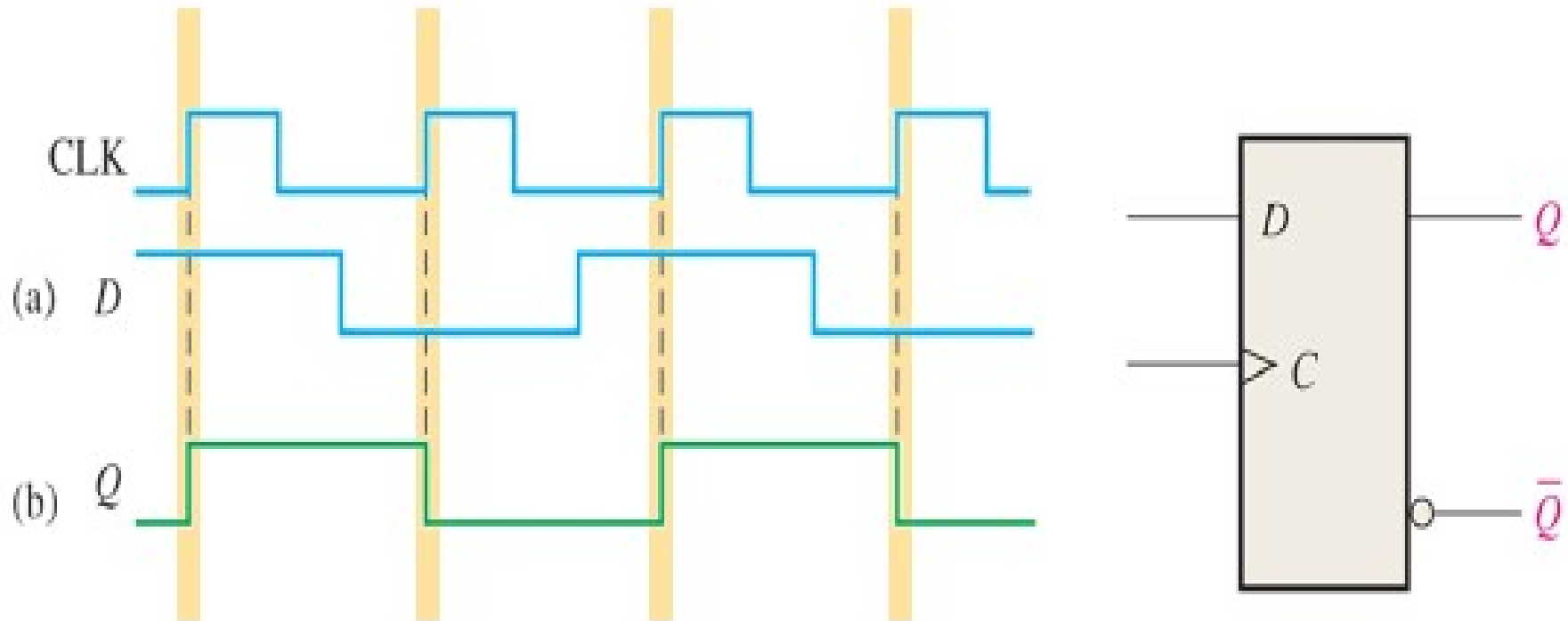
SR-Latch gated with a clock  
and including edge detector circuit  
→ gives a SR-Flip-Flop



# D type Flip-Flop

D-Latch gated with a clock and including edge detector circuit → gives a D-Flip-Flop

Q follows D on the rising edge of the clock



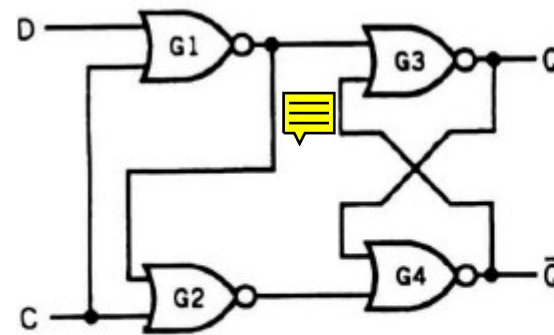
Let's discuss in more detail about timing issues ...

# Example of clocked flip-flop: NOR / NAND D-Latch

## NOR D-Latch:

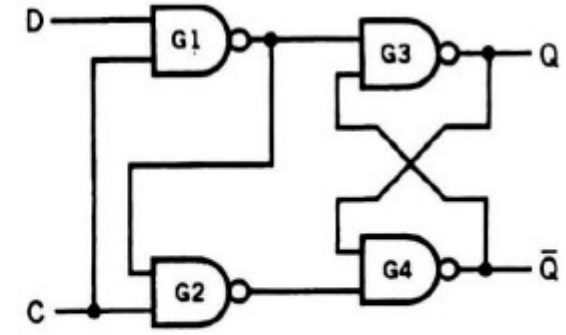
When the clock (C) is 0,  
G1 and G2 act like inverters  
→ the output Q of RS flip-flop  
G3/G4 follows the data input D

When the clock goes to 1 (C=1)  
the outputs of G1 and G2 → 0  
and Q registers the current state of D  
(latch for C=1)



(a) Latch for C = 1

NOR



(b) Latch for C = 0

NAND

- The setup time is  $t_s = 2t$  when D goes  $1 \rightarrow 0$  because the output of G1 must be kept at 1 for a time  $2t$  to ensure that G3/G4 is internally locked at  $Q = 0$   
In contrast,  $t_s = 3t$  when D goes  $0 \rightarrow 1$  because of the extra delay in G2
- The hold time is  $t_H = 0$  because D loses control over G1 and G2 as soon as  $C=1$

## NAND D-Latch:

Note: the NAND latch registers the current state of D when C=0 (latch for C=0)

- $t_s = 3t$  when D goes from  $1 \rightarrow 0$ , and  $t_s = 2t$  when D goes  $0 \rightarrow 1$
- $t_H = 0$

Note: behaves similar to a Gated D-latch (but different timing diagrams)



# Example of clocked flip-flop: edge-triggered FF

Edge-triggered flip-flops **change state only at one of the clock signal transitions**  
(We will normally take the **transition  $0 \rightarrow 1$  as the active one** – rising edge)

The state of the output after the clock depends on  
the **state of the output and the inputs just prior to the clock**

- **Time constraints**
  - (1) **Clock must remain active for a time  $t_w$**   
(minimum clock-pulse width defined by the edge detection circuitry)  
**Outside the interval defined by the setup and hold times, the inputs can change arbitrarily**
  - (2) **Changes of state if any, occur a propagation time  $t_p$  after the clock**  
Must be  $t_w < t_p$  otherwise it would be **impossible** to build systems in which the response of a given flip-flop depends on the **state of the flip-flop itself or other flip-flops at the time of the clock transition**

• An edge-triggered flip-flop is described by an **equation (or a truth table)** that gives the output during **clock period  $n + 1$**  as a function of the values of the **inputs and the output during clock period  $n$**

$$Q_{n+1} = F(\text{inputs}_n)$$

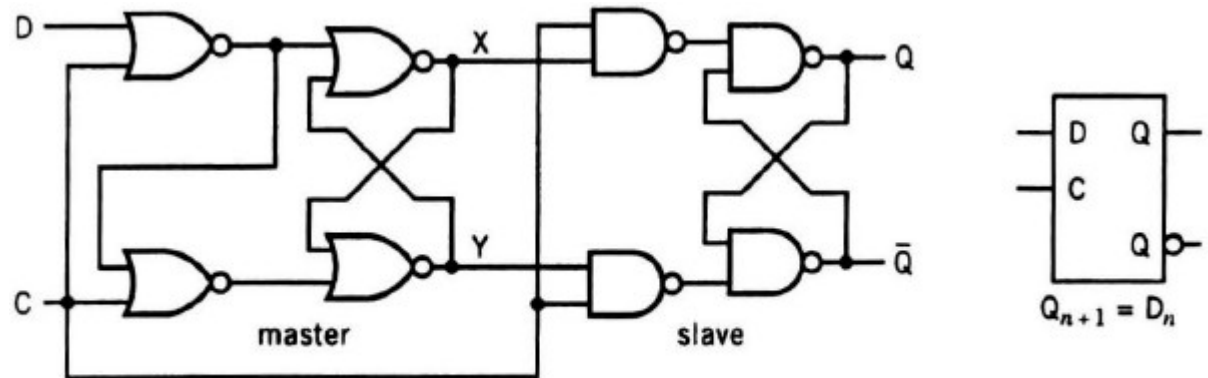
Example : the ubiquitous “D” type flip-flop,  
is governed only by its single data input D:  $Q_{n+1} = D_n$

# Example of clocked flip-flop: edge-triggered FF

## the “master-slave D FF”

The master–slave edge-triggered D flip-flop consists of a **master NOR latch** followed by a **slave gated NAND latch**

- When  $C=0$  the **master follows D** → and we have  $X = D$  and  $Y = \underline{D}$
- When  $C$  flips  $0 \rightarrow 1$ , these values are held by the master and **transmitted to the slave** →  $Q$  thus takes on the value that  $D$  had just prior to the clock



Master–slave edge-triggered D flip-flop, its symbol and equation

- When  $C$  flips back  $1 \rightarrow 0$ , the slave continues to hold the current value of  $Q$  while the master accepts new data

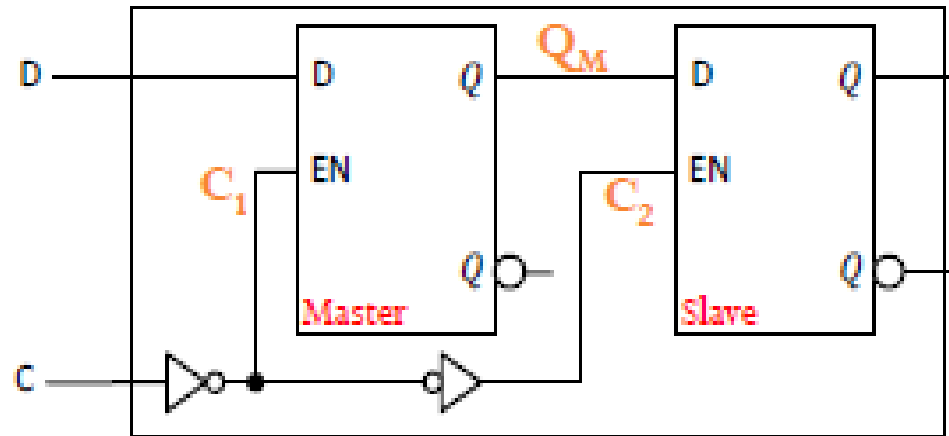
The **setup** and **hold times** are the same as in the NOR latch

The **propagation time** is  $2t$  for the output when  $0 \rightarrow 1$  and  $3t$  for the output when  $1 \rightarrow 0$

Note: two (gated) D-latches make a Master-Slave D-type Flip-Flop

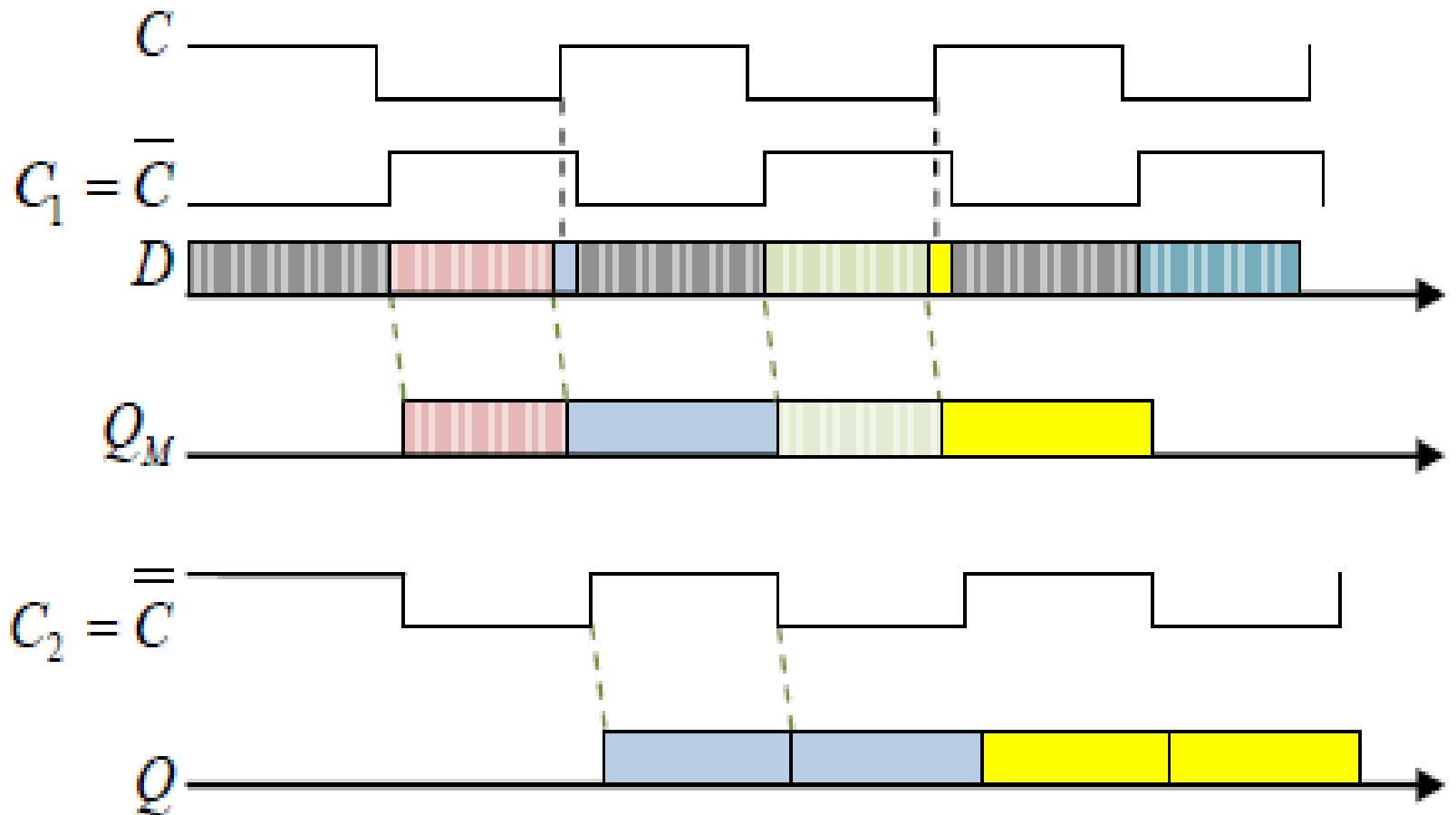
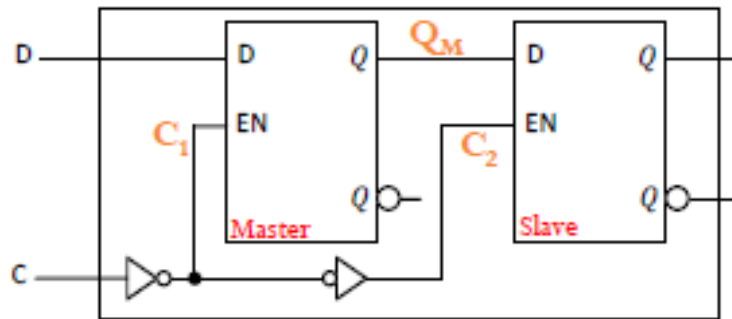
# Master-Slave D Flip-Flop

- Tie two D-latches together to make a D flip-flop



- When C is 0 ( $C_1 = 1$ ), the master latch is open and follows the D input.
- When C is 1 ( $C_1 = 0$ ,  $C_2 = 1$ ), the master latch is closed and its output is transferred to the slave latch.
  - The slave latch is open all the while that C is 1, but changes only at the beginning of this interval, because the master is closed and unchanging during the rest of the interval.

# Master-Slave D Flip-Flop



## Example of clocked flip-flop: edge-triggered FF the “toggle flip-flop”

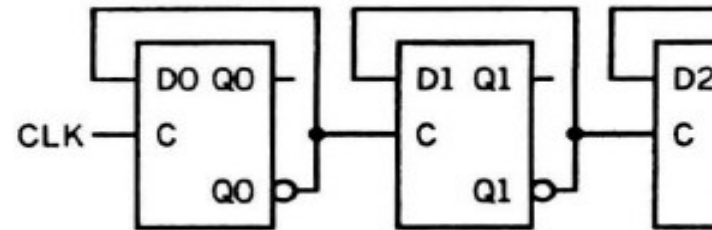
- If we connect the  $Q$  output of a D flip-flop to the D input, we obtain a **toggle flip-flop** that **changes state at every (active) clock transition** and therefore **divides the frequency of the clock signal by 2**.

Its equation is  $Q_{n+1} = \overline{Q}_n$

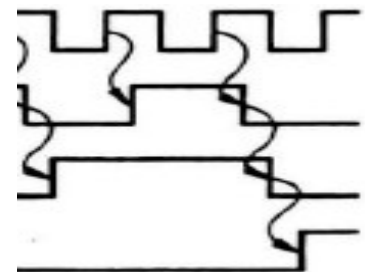
Note: that a toggle flip-flop is the ultimate example of **why propagation times must be longer than hold times ( $t_w$  clock)** – otherwise metastable state for D

- If we now chain together **N toggle flip-flops** by tying the images output of one flip-flop to the C input of the next, we obtain the **divide-by-2N ripple counter** (see figure for  $N = 3$ )

Although not strictly synchronous, the ripple counter is nonetheless widely used because it is simple and, having no inputs, has **no problems with setup or hold times**



The **drawback** it does have, however, is that it can take up to **N propagation times** to change from one state to the next



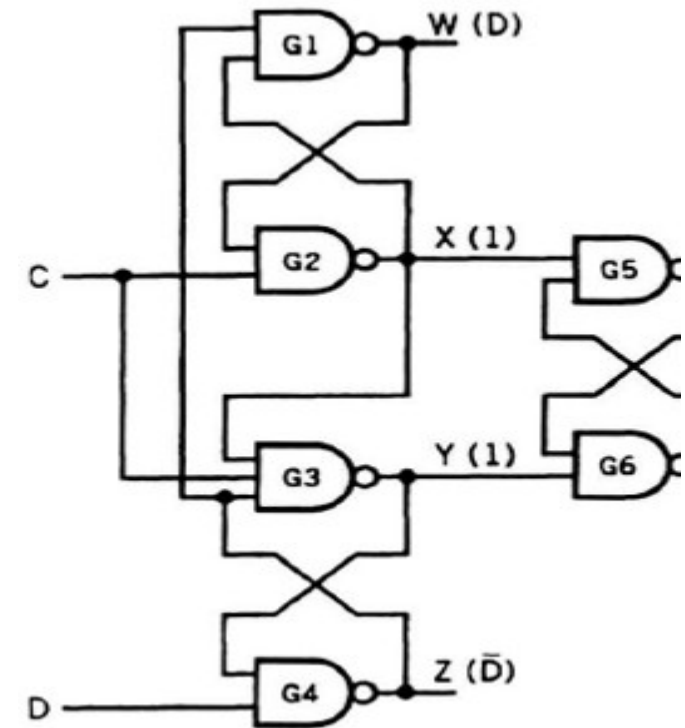
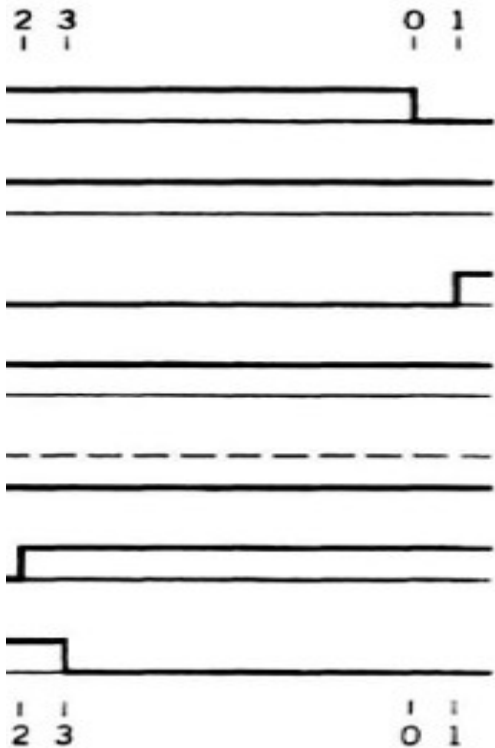
# Example of clocked flip-flop: edge-triggered FF

## the “D flip-flop” LS-TTL 74LS74

Flip-flops G1|G2 and G3|G4 are both arbiters which **register the value of D when  $C \rightarrow 1$**

- The values in brackets indicate the state of the internal nodes when  $C=0$
- The waveforms (below) correspond to the case  $D = 1$  and  $Q = 0$

Note: **drawing waveforms** is often the easiest way to understand how digital circuits work, particularly when feedback is involved



If  **$D = 1$** , we have  **$W = 1$  and  $Z = 0$**  when  $C = 0$ ;  
 then  $X \rightarrow 0$  one gate delay  $\tau$  after  $C$  changes  $0 \rightarrow 1$   
 and confirms  $W = 1$  and  $Y = 1$ . After this time,  
 $Z$  can change without affecting  $X$  and  $Y$  ( $\rightarrow t_H = 0$ )  
 $X$  must remain  $= 0$  for at least two gate delays in order  
 to lock output  $Q$  of flip-flop G5|G6 at 1

**$\rightarrow$  The minimum clock-pulse width  $t_w$  is thus  $2t$**

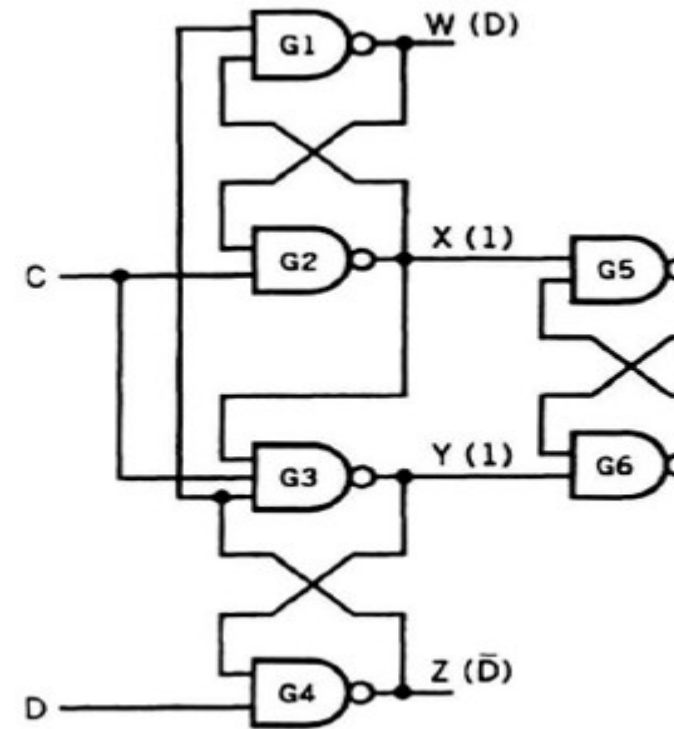
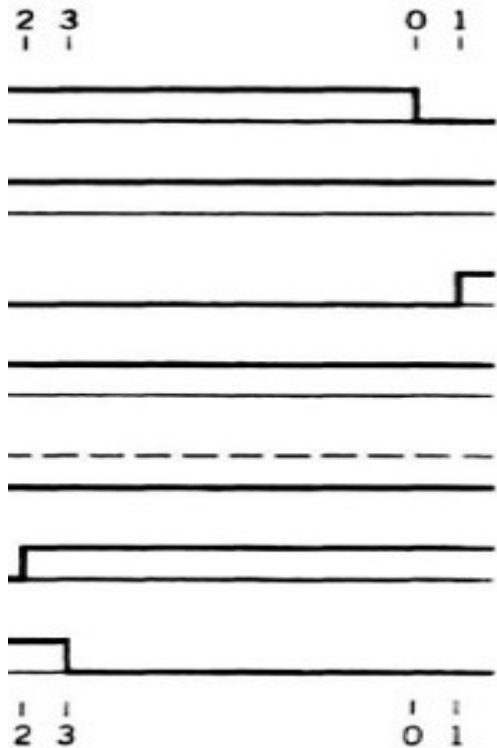
# Example of clocked flip-flop: edge-triggered FF

## the “D flip-flop” LS-TTL 74LS74

Flip-flops G1|G2 and G3|G4 are both arbiters which **register the value of D when  $C \rightarrow 1$**

- The values in parentheses indicate the state of the internal nodes when  $C=0$
- The waveforms (below) correspond to the case  $D = 1$  and  $Q = 0$

Note: **drawing waveforms** is often the easiest way to understand how digital circuits work, particularly when feedback is involved

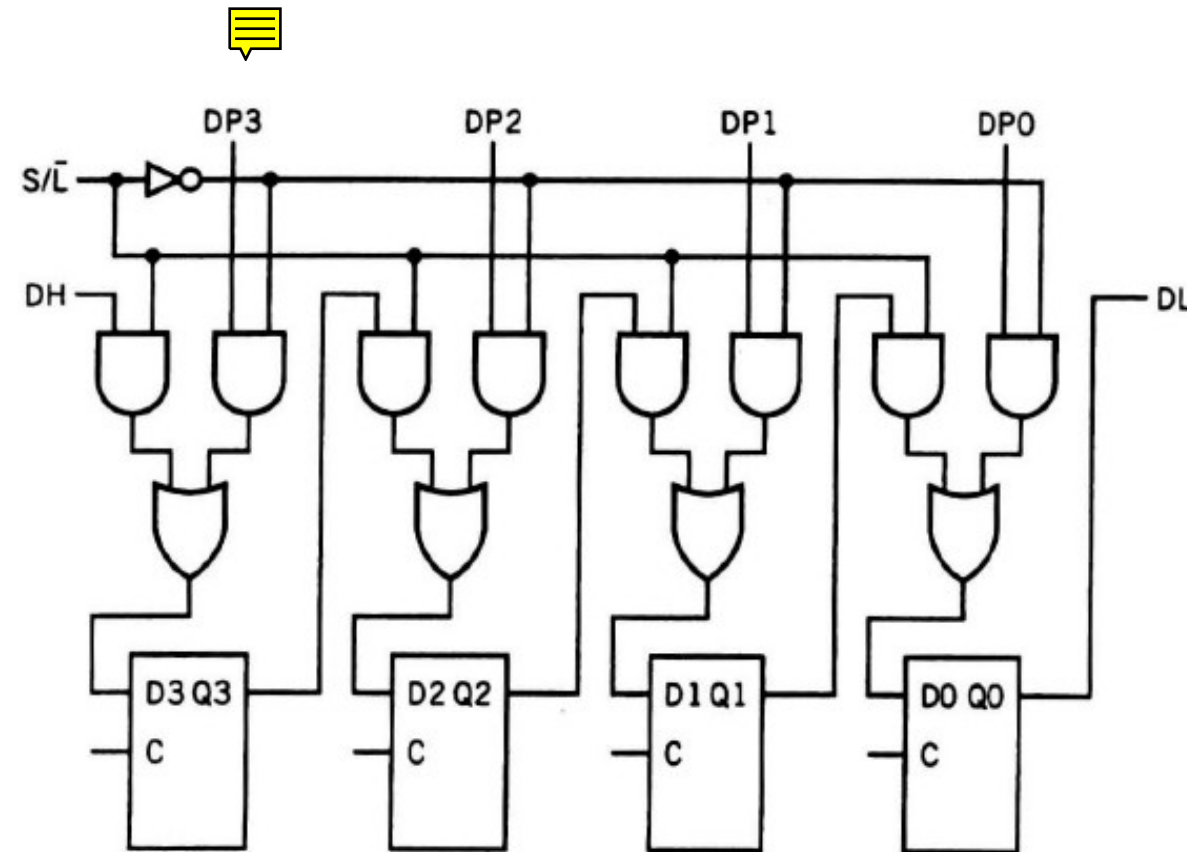


If  **$D = 0$** , we have  **$W = 0$  and  $Z = 1$**  when  $C = 0$ ;  
in this case,  $Y \rightarrow 0$  one gate delay  $\tau$  after the clock transition  
and confirms  $Z = 1$ ,  $W = 0$  and  $X = 1$

The hold time  $t_H$  is  $\tau$  because  $D$  must remain steady until  $Y$  becomes 0. The setup time  $t_s$  is  $2\tau$ , and the propagation time  $t_p$  to the  $Q$  output is  $2\tau$  if  $Q$  goes from  $0 \rightarrow 1$  and  $3\tau$  if  $Q$  goes from  $1 \rightarrow 0$

## Example use of D flip-flop in synchronous system

### a 4-bit shift-register



The AND–OR gates are simple examples of **multiplexers**, that is, of circuits that route one of several inputs to their output according to the setting of their address or control inputs.

In this case, the only **control input** is **S/L (SHIFT/LOAD)**: it determines whether the D input of a given flip-flop is the Q output of the flip-flop of the next-highest order (DH in the case of D3), or whether it is the corresponding bit of the 4-bit word DP3 ... DP0

- If  $S/L=1$ , the 4-bit word in the shift register is shifted one position to the right at the clock transition; DH is copied into Q3, and DL can be copied into the DH input of a lower-order word. Q3 is copied into itself if it is tied to DH; we can thus obtain an arithmetic shift-right or divide-by-2 circuit.
- If  $S/L=0$ , new parallel data are copied into the shift register (register load)

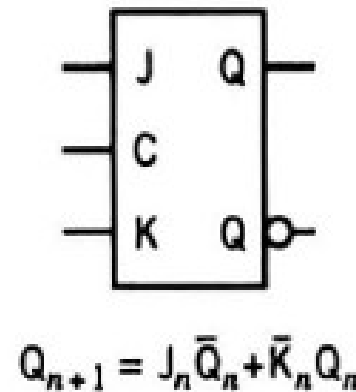
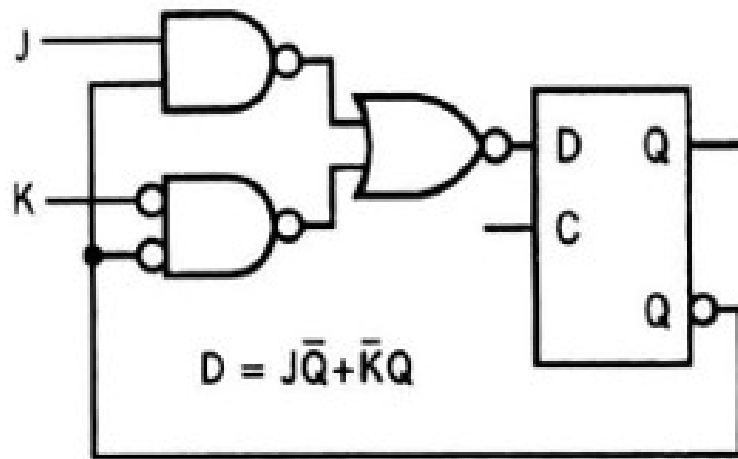


# JK flip-flop – yet another clocked flip-flop

Less common than D flip-flops, but highly useful because of their versatility, are JK flip-flops with inputs J and K; their equation is

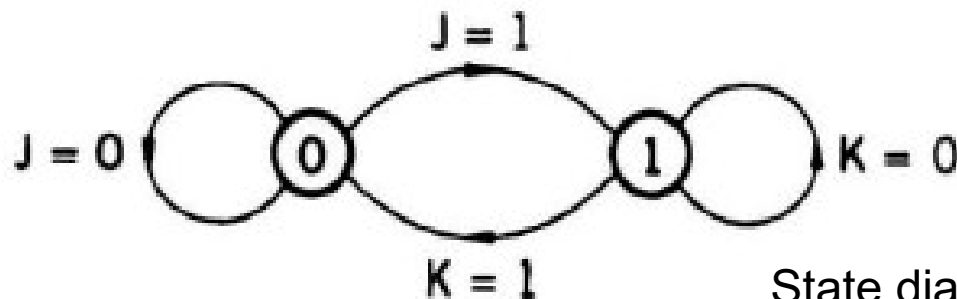
$$Q_{n+1} = J_n \bar{Q}_n + \bar{K}_n Q_n$$

A JK flip-flop can be obtained by making the D input of a D flip-flop =  $J\bar{Q} + \bar{K}Q$



$J_n$	$K_n$	$Q_{n+1}$
0	0	$Q_n$
0	1	0
1	0	1
1	1	$\bar{Q}_n$

A JK flip-flop changes state if J and K are both equal to 1 and remains in the same state if J and K are both equal to 0



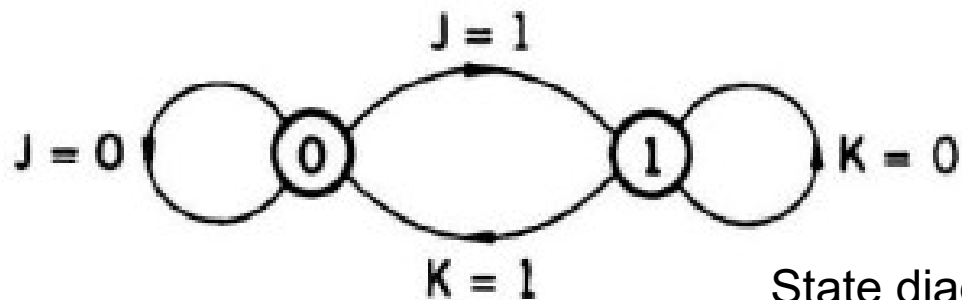
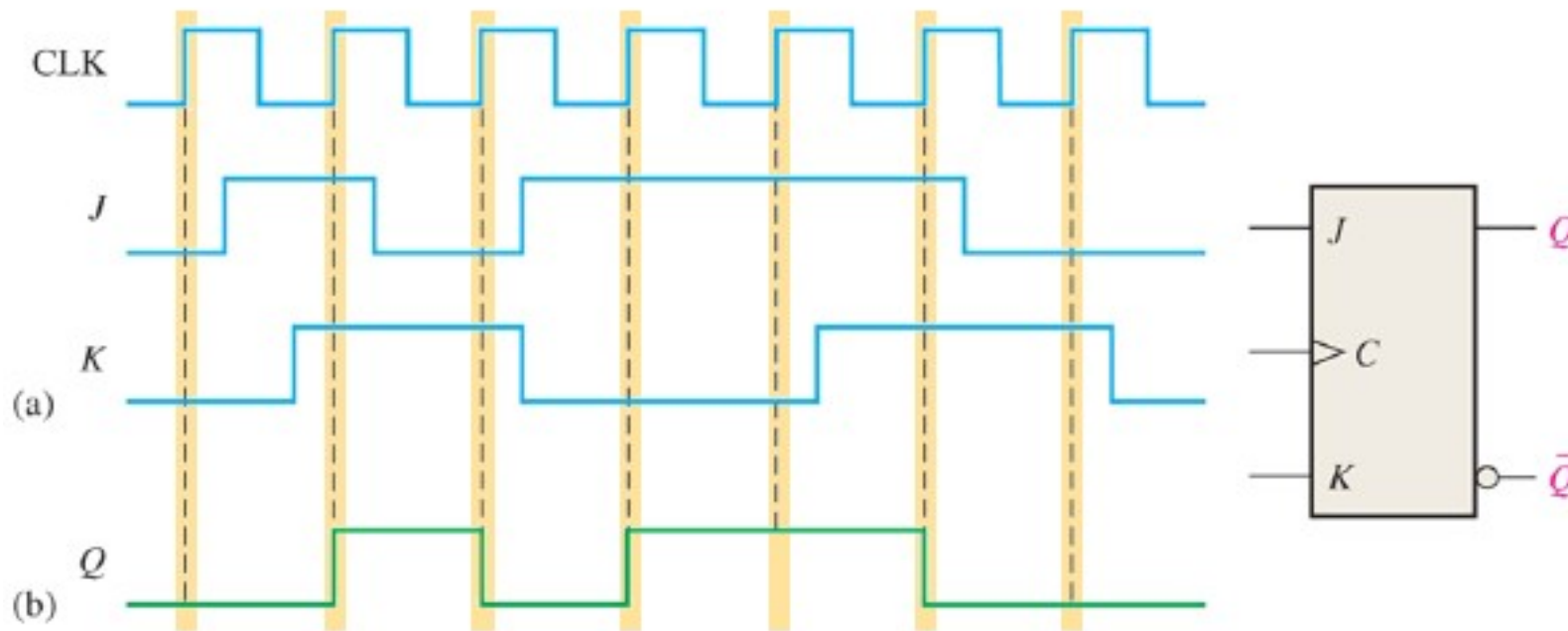
State diagram for the JK flip-flop

# JK flip-flop – yet another clocked flip-flop

Less common than D flip-flops, but highly useful because of their versatility, are JK flip-flops with inputs J and K; their equation is

$$Q_{n+1} = J_n \bar{Q}_n + \bar{K}_n Q_n$$

A JK flip-flop can be obtained by making the D input of a D flip-flop =  $J\bar{Q} + KQ$



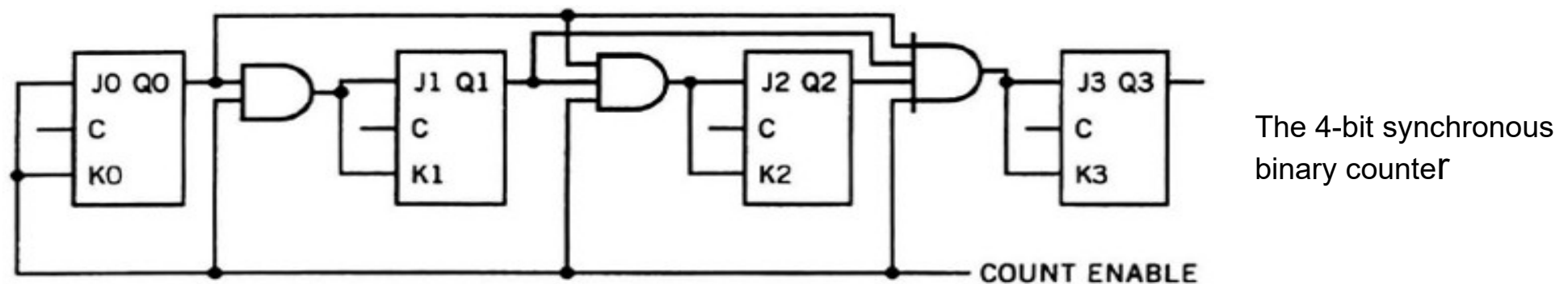
State diagram for the JK flip-flop

## Example of use of JK flip-flop: 4-bit counter

If COUNT ENABLE = 0, the J and K inputs to all flip-flops are 0, and the counter remains wherever it is

If COUNT ENABLE = 1, the lowest-order bit Q0 toggles at every clock, and in each succeeding position the AND gates allow the bit to toggle only if all preceding bits are equal to 1

→ the counter thus cycles through the sequence 0, 1, ..., 15, 0, and so on

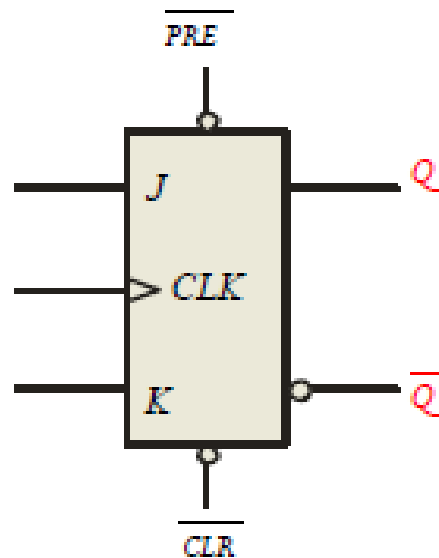


By exploiting a more complicated gating scheme, the counter can also **count down**, and (like the 4-bit shift register see previously) and can be **set to an arbitrary value**

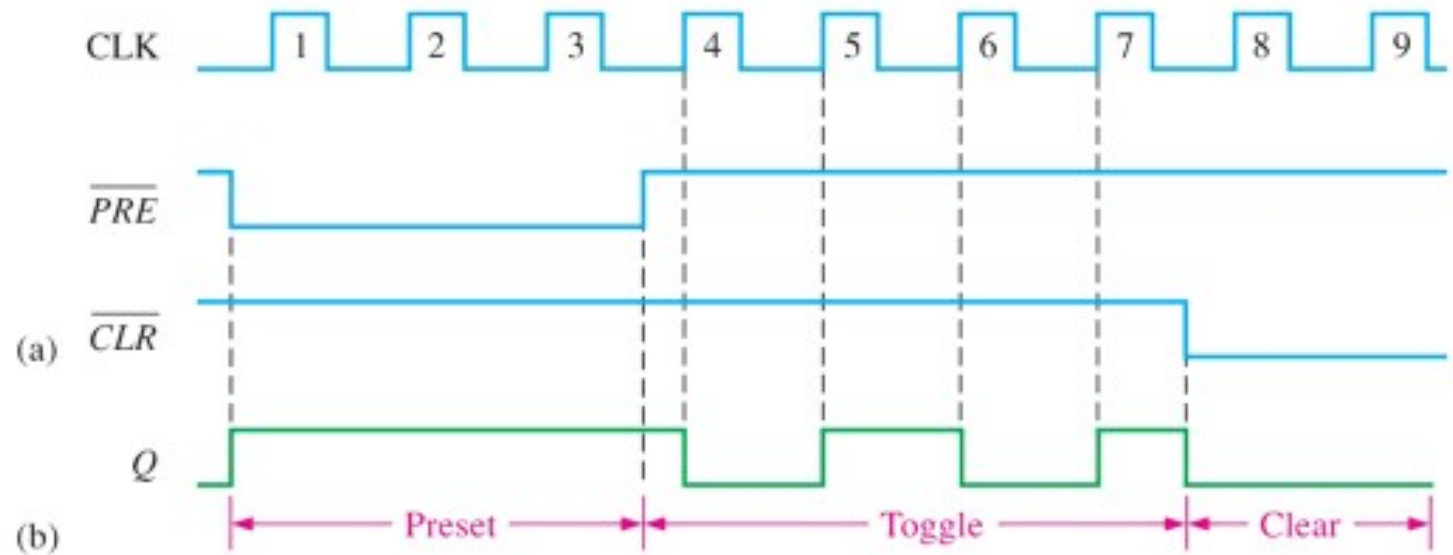
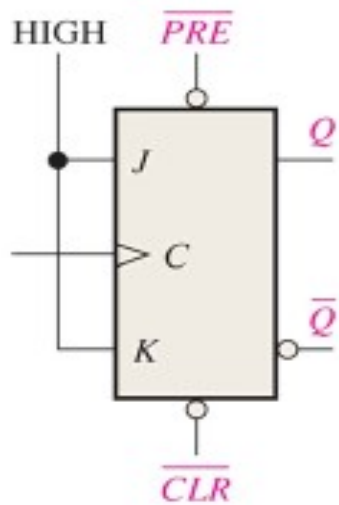
Note: loading a parallel word and then **counting down to zero** is useful when generating pulses of variable length because **only one gate is needed** to detect when the counter arrives at zero... in contrast, **counting up to a given value** requires a **complete magnitude comparator** (much more gates needed !)

## JK flip-flop with synchronous inputs

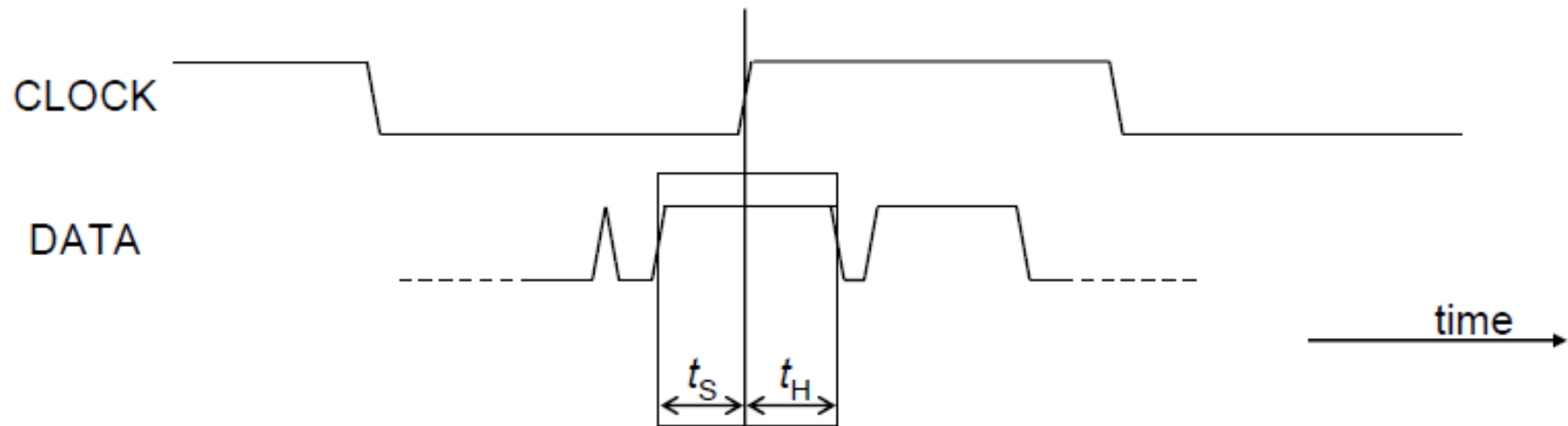
- Most flip-flops have other inputs that are *asynchronous*, meaning they affect the output independent of the clock.
- Two such inputs are normally labeled **preset (PRE)** and **clear (CLR)**.
- These inputs are usually active-LOW.
- A J-K flip flop with active-LOW preset and CLR is shown.



# JK flip-flop with synchronous inputs: example



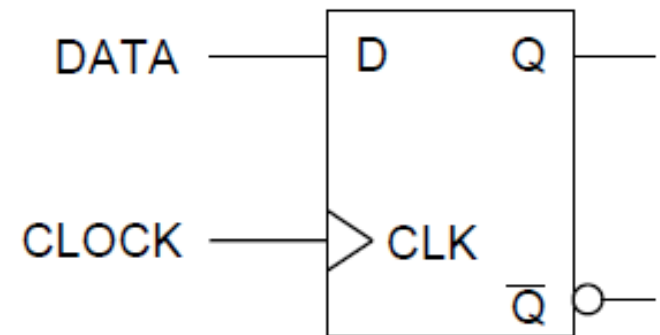
# Setup and Hold times



The data/control inputs to the flip-flop must be stable:

For a time  $t_S$  before the clock edge (the **setup time**)

For a time  $t_H$  after the clock edge (the **hold time**)



Applies to all types of flip-flop

## Note: Clock Skew

We learned that input must not change values at the moment that the clock is rising as this is the time that the flip-flops **read** the input values. Ok, but ...

... we have so far made the implicit assumption that there is no clock skew, that is, that the **clock transition is fast**, that it **arrives simultaneously** at all flip-flops and all of them have the **same threshold voltage**

Clock skew is never zero, but it must nonetheless be small enough to prevent one flip-flop from firing so far ahead of another that it changes state before the other has made a decision; in the worst of cases, a **late clock can make a flip-flop become metastable**

A slow clock **rise time** can result in a large clock skew if the scatter in the **threshold voltages of different** flip-flops is large; this problem particularly affects **CMOS circuits**, whose threshold voltages can lie anywhere between one-third and two-thirds of the power-supply voltage.

Clock skew also occurs if there are delays in the **clock distribution** circuit due either to gates or to transmission lines; this problem is of particular concern in **ECL circuits**