SEQUENTIAL CIRCUITS

Sequential Circuits: *An Overview*

In sequential circuits, outputs depend on both present & past inputs. We consider sequential circuits and their realizations.

Types of sequential circuits: *Synchronous* and *Asynchronous*:

Synchronous	*Asynchronous*
Clocked: need a clock input	*Unclocked*
responds to inputs at discrete time instants governed by a clock input	responds whenever input signals change

***Asynchronous flip-flops* (FFs): set-reset (SR) FFs & applications**

***Synchronous flip-flops* (FFs) come with *clocked* circuits: J-K, D, and T FFs, and conversions between them.**

Design and realization of *asynchronous* & *synchronous* counters. Storage and shift registers.

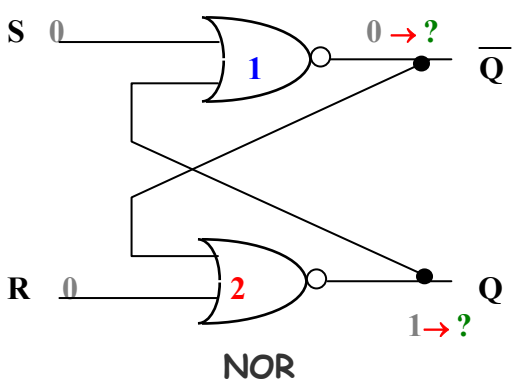
VHDL based modeling of sequential circuits!

Realization of Basic *Flip-flops* & Simple Applications

Objective: **to obtain a basic understanding of FF operations.**

FFs are building blocks of *sequential circuits* and are fundamental *memory devices* which have two stable states: 1(T) *or* 0(F) ⇒ *implies that a* FF *can store 1 bit of info*.

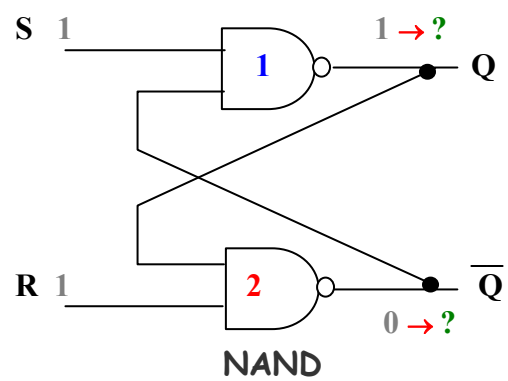
The most basic FF is *unclocked* S-R *Flip-flop* of which there are two varieties: the NOR form and the NAND form.



NOR

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

0 0 is the rest state



NAND

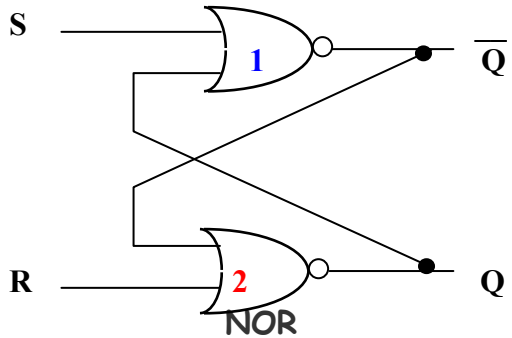
S R	Output
0 0	Invalid
0 1	Q = 1
1 0	Q = 0
1 1	No change

1 1 is the rest state

state tables follow directly from truth tables of NOR and NAND gates.

S-R *Flip-flops* & Simple Applications

S-R FF can *record* and *store* transient events.

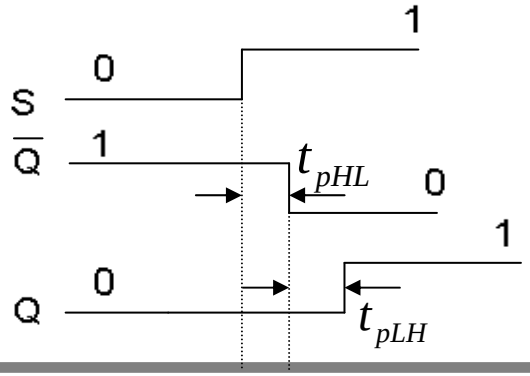


S R	Output
0 0	No change
0 1	Q = 0
1 0	Q = 1
1 1	Invalid

0 0 is the rest state

Assume that the *rest state* is: S = R = 0; and let Q = 0, $\overline{Q}$ = 1.

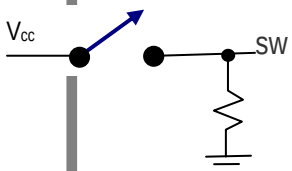
At some point in time, if S →  **while R = 0 ⇒ Q = 1, $\overline{Q}$ = 0,**
i.e., the event (S going *high*) is recorded and stored as Q = 1.

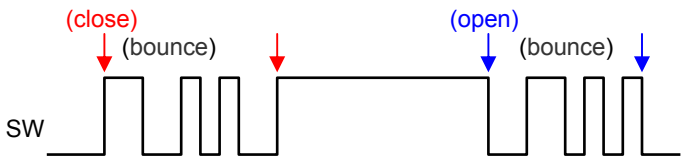


Switching is *not instantaneous* i.e., *propagation delays* are involved

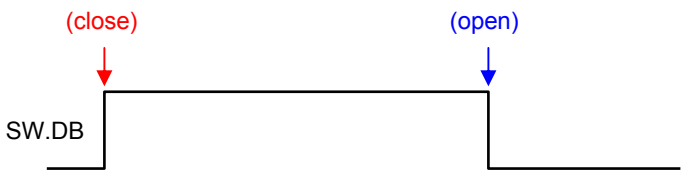
S-R Flip-flops & Simple Applications

If a mechanical switch is used as an input to a digital circuit, it may cause problems – it bounces (see Fig (a)) before settling down.





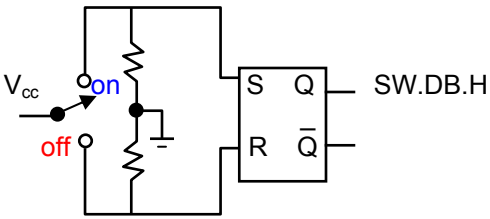
(a) Undebounced switch signal



(b) Debounced switch signal

To obtain a clean signal (see Fig (b)), we can use a S-R FF.

***Switch debouncing* is a common use of S-R FFs.**



Switch debouncing circuit

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

0 0 is the resting state

Analysis:

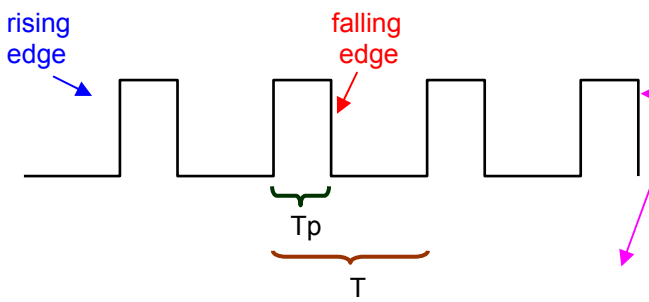
a. Vcc at OFF position: S=0, R=1 ⇒ Q = 0

b. Throw switch to ON: S=  , R=0 ⇒ Q=1

c. Throw switch to OFF: S=0, R=  ⇒ Q= 0

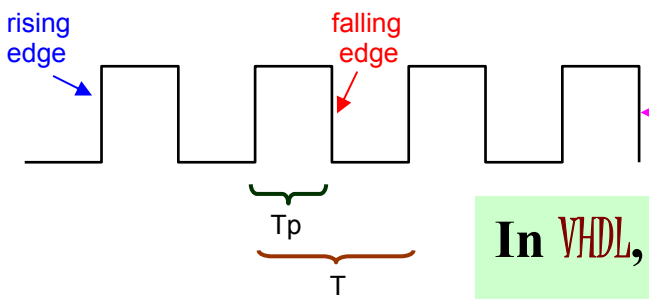
Clock Signal: *a periodic pulse train of equally spaced pulses*

Clock input is a controlling input to a clocked sequential circuit which specifies when FF outputs can change in response inputs.



Clock signals with *pulse width* of Tp & a *period* of T

Duty cycle (% of time where clock is TRUE) = $\frac{Tp}{T} \times 100\%$



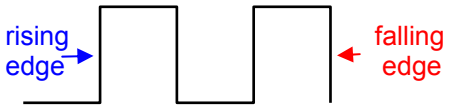
A clock signal with a 50% duty cycle

In VHDL, use "`if`" statement to capture clock edges:

```
if (clk'event and clk='l')    -- rising edge triggered

if (clk'event and clk='0')    -- falling edge triggered
```

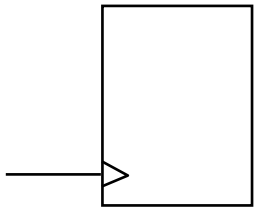
Clocked Sequential Circuits

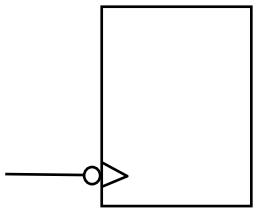
Sequential circuits **respond to inputs only at the active clock edges i.e., HIGH → LOW or LOW → HIGH transitions**



At any other time in the clock cycle, changing inputs have no effect on the output.

Edge triggered **circuit elements with active-high and active-low clock inputs (indicated by bubble 'o'):**





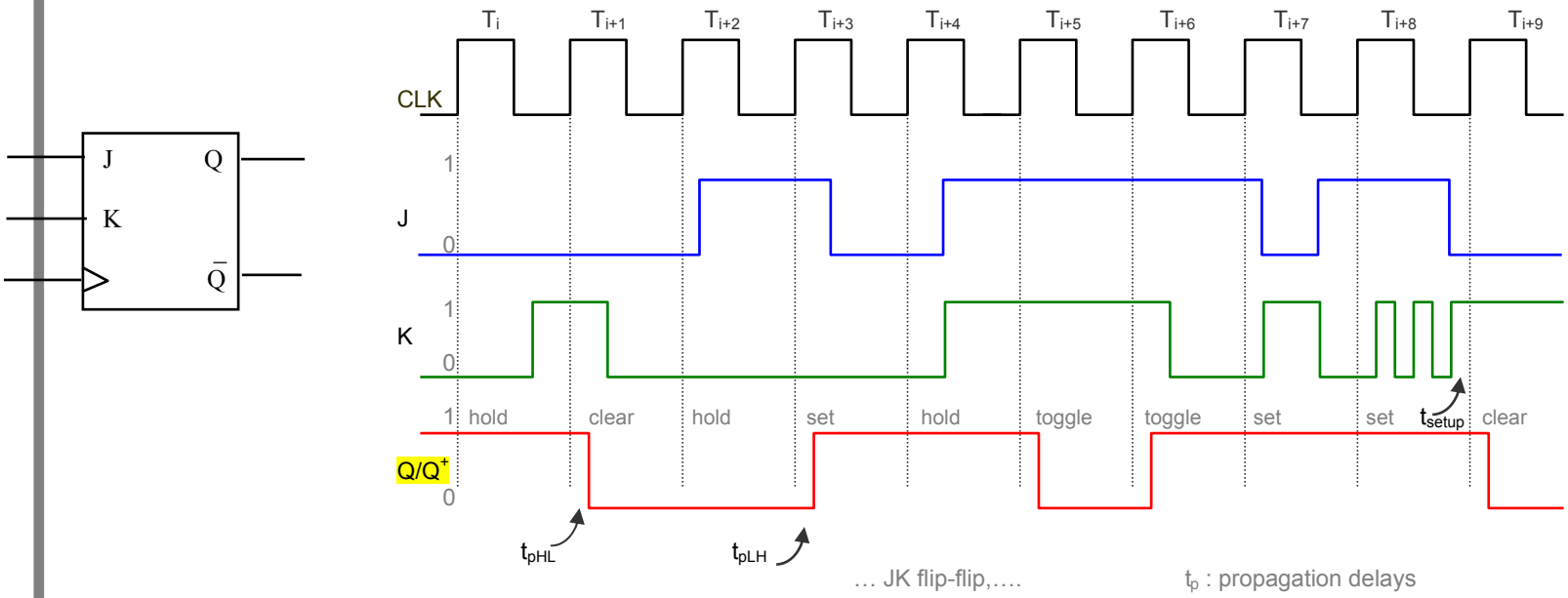
active-high clock input :
responds to inputs "at the moment" clock signal goes from
LOW → HIGH (*rising edge*)

active-low clock input :
responds to inputs at *falling edge* of clock signal i.e., "at the moment" clock signal goes from HIGH → LOW.

Clocked FFs only respond to inputs at *active clock edges*

When *inputs don't change* ⇒ FF *outputs don't change*. If *inputs change* ⇒ FF *output changes state* only at the *active* clock edge.

Different kinds of FFs are available which differ by types of inputs, and how the inputs affect FF operation.



Intrinsic timing parameters associated with FF operations.

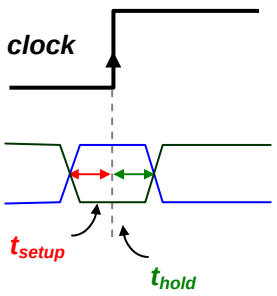
FF timing parameters

t_{setup} : **minimum time** before the ***active*** clock edge by which FF inputs must be stable.

t_{hold} : **minimum time** inputs must be stable after ***active*** clock edge

t_{pHL} : time taken for FF output to change state from **High** to **Low**.

t_{pLH} : time taken for FF output to change state from **Low** to **High**.



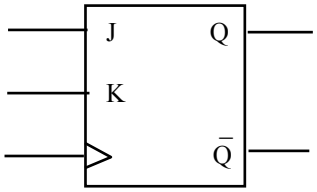
What happens if inputs *change state* right at the ***active clock transition*** (**ACT**)?? *Answer*: **output is *unpredictable***

Thus, inputs must meet the required **setup** & **hold** **times** of device.

Design methods considered later will explicitly guarantee this…

J-K Flip-flop

FF responds at *rising edge* (*positive edge triggered*)



J-K Flip-Flop

next output state

CLK	J	K	Q	Q^+
↑	0	0	0	0
↑	0	0	1	1
↑	0	1	0	0
↑	0	1	1	0
↑	1	0	0	1
↑	1	0	1	1
↑	1	1	0	1
↑	1	1	1	0

characteristic table

J	K	Q^+	Operation
0	0	Q	hold
0	1	0	clear
1	0	1	set
1	1	$\bar{Q}$	Toggle

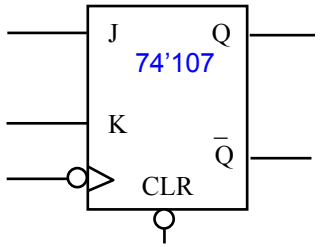
condensed characteristic table

```
entity JK_FF is
   port ( Q, Qbar: buffer STD_LOGIC;
          J,K, CLK:  in      STD_LOGIC);
end JK_FF;
architecture JK_FF_BEH of JK_FF is
begin
 process ( CLK )
  variable tmp: STD_LOGIC;
  begin
    if CLK'event and CLK ='1' then
       if   J/=K then tmp := J;
       elsif J='1' then tmp := not Q;
       else tmp := Q;
       end if;
     Q <= tmp; Qbar <= not tmp;
    end if;
 end process;
end JK_FF _BEH;
```

process executes when there's an event on **CLK**: when **CLK** changes to '1' ⇒ Q changes depending on values of **J** & **K** as shown in the JK FF's characteristic table.

Commercially available J-K Flip-flops

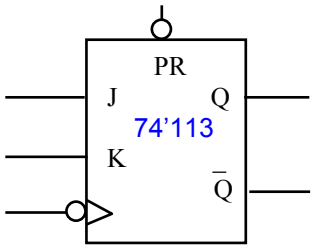
These commercial FFs respond at the falling edge (*negative edge triggered*)



74'107 with asynchronous clear

CLK	CLR	J	K	Q^+
X	L	X	X	L
↓	H	L	L	Q
↓	H	L	H	L
↓	H	H	L	H
↓	H	H	H	$\overline{Q}$

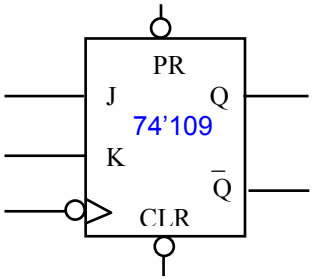
voltage table (X denotes "don't care")



74'113 with asynchronous set

CLK	PR	J	K	Q^+
X	L	X	X	H
↓	H	L	L	Q
↓	H	L	H	L
↓	H	H	L	H
↓	H	H	H	$\overline{Q}$

voltage table



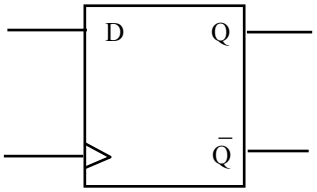
74'109 with direct set & direct clear

CLK	PR	CLR	J	K	Q^+
X	L	H	X	X	H
X	H	L	X	X	L
X	L	L	X	X	not allowed
↓	H	H	L	L	Q
↓	H	H	L	H	L
↓	H	H	H	L	H
↓	H	H	H	H	$\overline{Q}$

voltage table

D Flip-flop (D for delay)

doesn't depend on previous value!



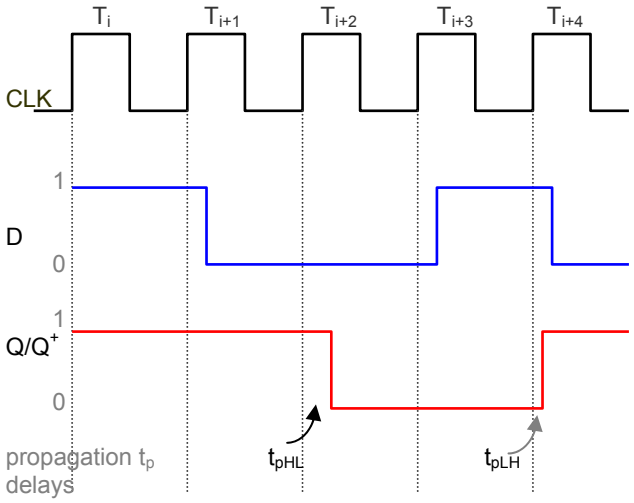
Functional block diagram of a D Flip-Flop

CLK	D	Q	Q^+
↑	0	0	0
↑	0	1	0
↑	1	0	1
↑	1	1	1

characteristic table

CLK	D	Q^+
↑	0	0
↑	1	1

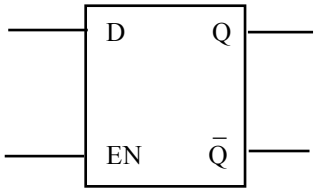
condensed characteristic table



Timing diagram

```
entity D_FF is
    port ( Q, Qbar: out STD_LOGIC;
           D,  CLK: in  STD_LOGIC);
 end D_FF;
 architecture D_FF_BEH of D_FF is
 begin
  process (CLK)
   begin
     if CLK'event and CLK ='1' then
        Q <= D; Qbar <= not D;
     end if;
  end process;
 end D_FF_BEH;
```

D Latch & T Flip-flop



Functional block diagram of a D Latch

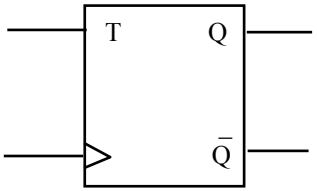
EN	D	Q
0	X	no change
1	0	0
1	1	1

truth table

(X denotes "don't care")

The **D Latch** is basically a level sensitive FF.

Model these using VHDL!



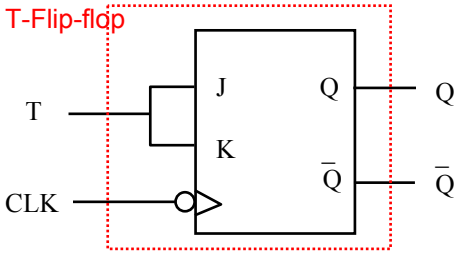
Functional block diagram of a T Flip-flop

CLK	T	Q	Q^+
↑	0	0	0
↑	0	1	1
↑	1	0	1
↑	1	1	0

characteristic table

CLK	T	Q^+
↑	0	Q
↑	1	$\overline{Q}$

condensed characteristic table



A T Flip-flop made from a J-K Flip-flop

J	K	Q^+
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

truth table

Since **T Flip-flops** are easy to construct from other FFs, they are not available commercially.

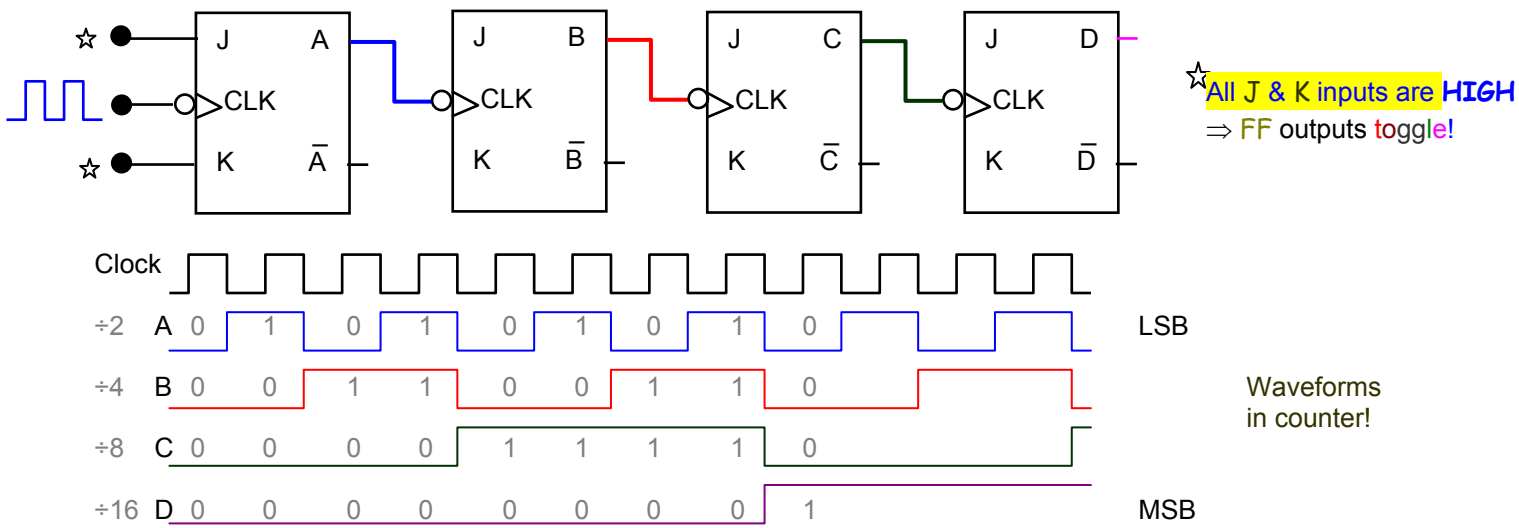
Counters: Asynchronous & Synchronous counters

Asynchronous counters : circuit elements do not get the clock input simultaneously.

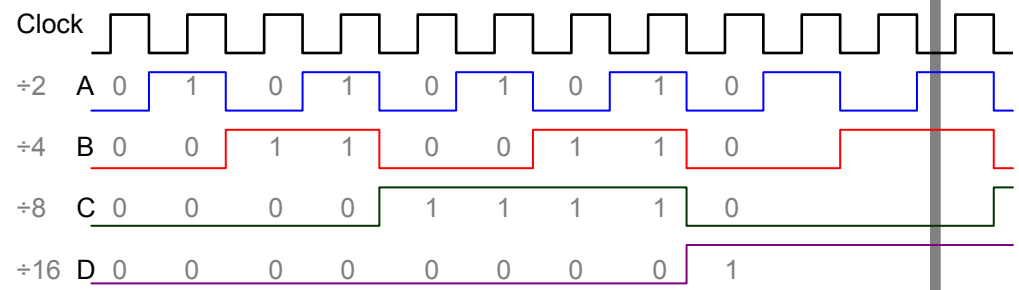
Synchronous counters : circuit elements get the clock input simultaneously.

Asynchronous (*ripple*) counter:

clock connected to first (LSB) FF only while succeeding FFs get their clock input from output of previous FF asynchronous counter



About the Asynchronous Ripple Counter…



Each FF A,B,C, & D successively *halves* the input clock frequency.

The 4-bit counter:

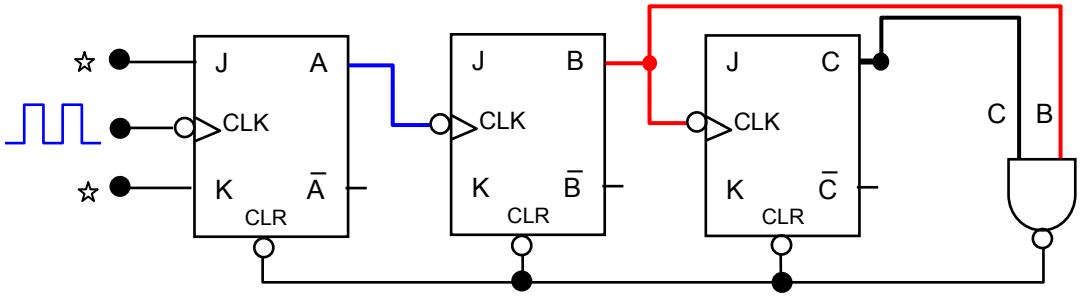
- counts in sequence from 0000 (0) → 1111 (15)
- has 16 distinct count states ⇒ called a mod-16 counter

N FFs connected this way will have 2^N states ⇒ mod-2^N counter

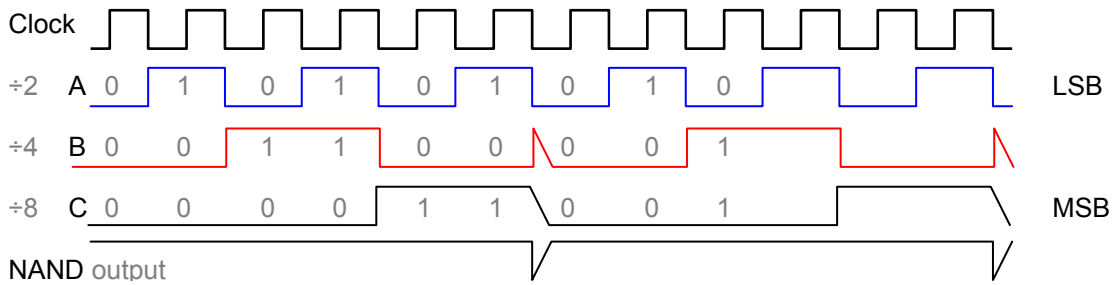
How to obtain a counter with mod-X < 2^N? *Answer*:

- assume counter starts from 0
- identify FFs that will be in HIGH state when count = X
- feed those FFs outputs to a NAND gate
- connect NAND gate output to asynchronous CLR input of FFs

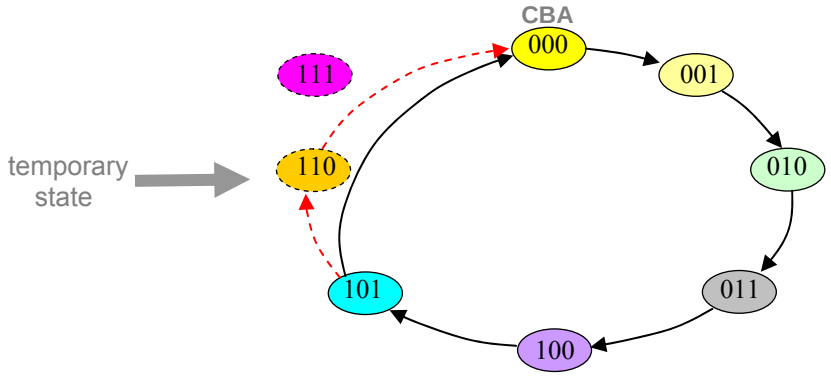
A Mod-6 asynchronous counter… mod-6 < 2^3



☆ All J & K inputs are **HIGH**
⇒ FF outputs toggle!



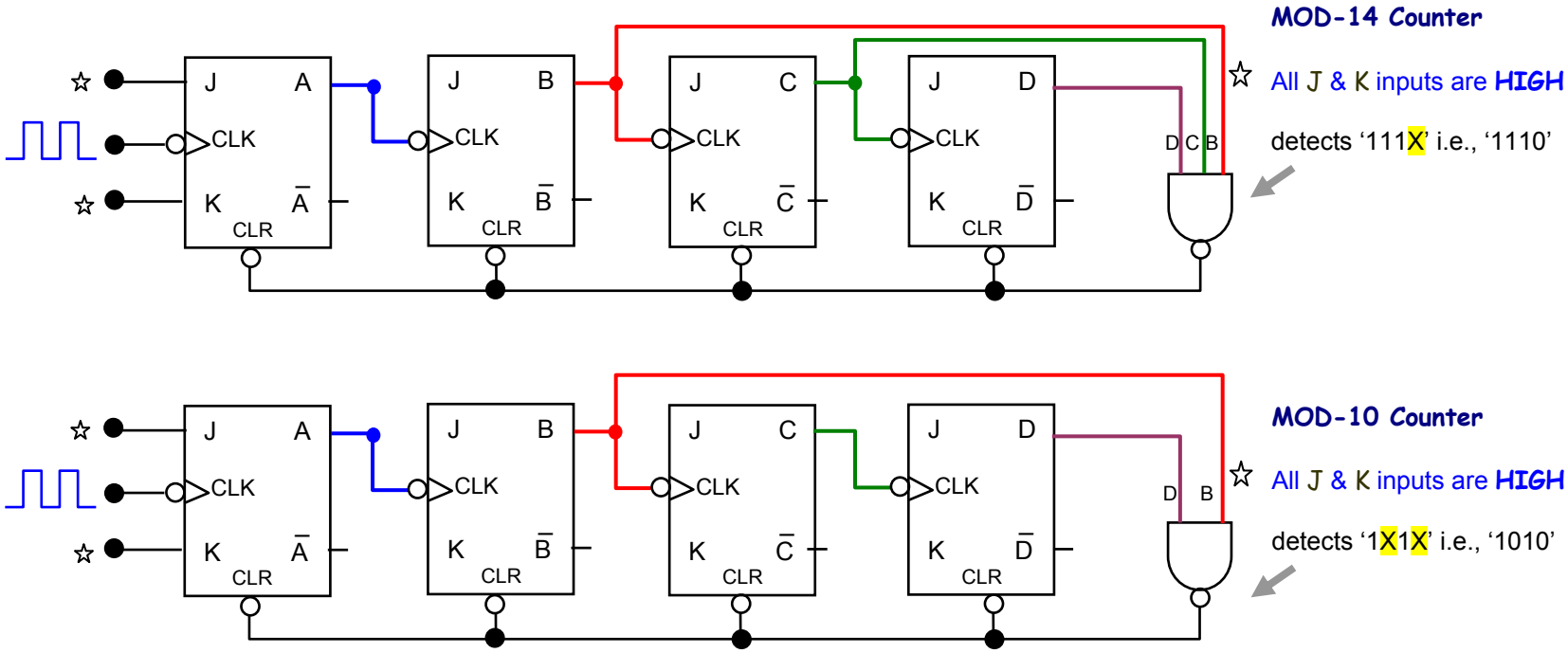
Waveforms in counter!



The state transition diagram for the mod-6 counter!

A Mod-14 & Mod-10 (decade) counters…

D C B A
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 0
1 0 0 0
1 0 0 1
1 0 1 0 →
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0 →
1 1 1 1



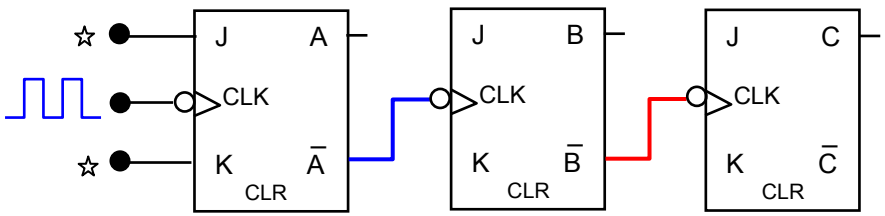
Decade counter: counts up in ordinary binary sequence: 0000 → 1001

Decade counters are also called **BCD counters**.

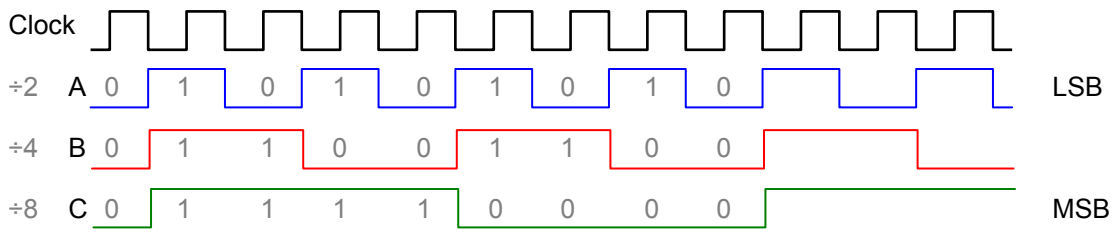
Ripple counters considered so far, *counted up*↑. How to make them *count down*↓?

Count-down ripple counter…

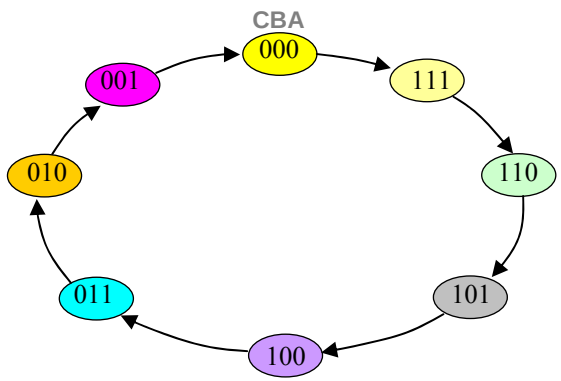
To count down : **connect complements of FF outputs to clock inputs of succeeding FFs**



☆ All J & K inputs are **HIGH**
⇒ FF outputs toggle!



Waveforms in counter!



The state transition diagram for the **mod-8 down counter**!

Asynchronous ripple counter… limiting frequency

Ripple counters are very easy to implement but have a major drawback: *they cannot operate beyond a limiting frequency* and so are primarily useful for low frequency applications.

The limitation arises because propagation delays of the **FFs** in the **chain** add up:

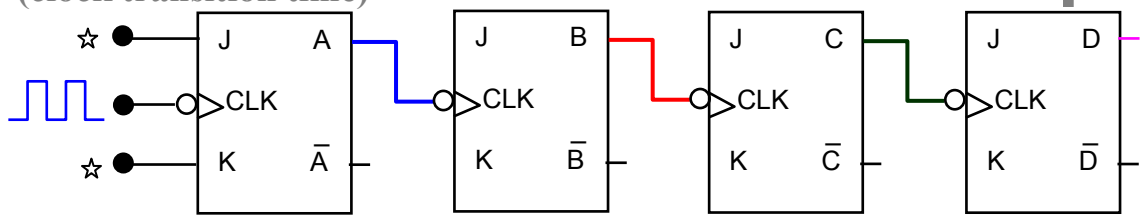
- clock input to FF1: t_0
- clock input to FF2: $t_0 + t_{pd}$
- clock input to FF3: $t_0 + 2t_{pd}$

⋮

- clock input to FFN: $t_0 + (n-1)\ t_{pd}$

  ⇒ the nth FF rather changes state at $n\ t_{pd}$ after t_0

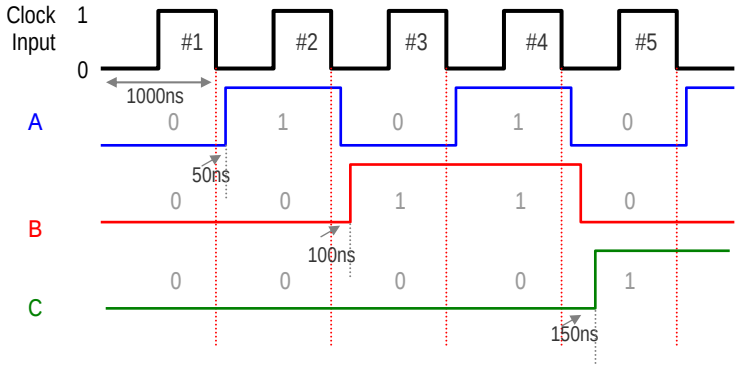
(clock transition time)

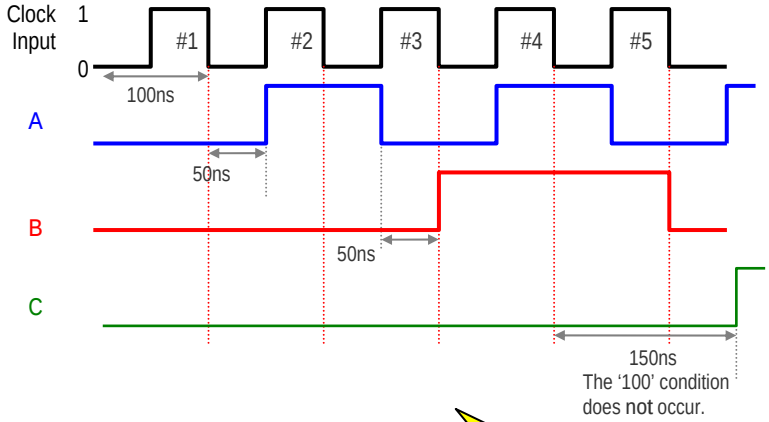


Hence for proper operation: $T_{clock} \geq n\ t_{pd}$ or $f_{max} \leq 1 / n\ t_{pd}$

This is illustrated by waveforms in the next slide!

Asynchronous ripple counter… limiting frequency.. cont.





Worst case design for a ripple counter:

Here propagation delays imply f_{max} = 1/(3 x 50ns) = 6.67 MHz. But the clock frequency is 10 MHz!

Consider a FF: $t_{pLH} \cong 16$ ns., $t_{pHL} \cong 24$ ns.

Hence, choose $t_{pd} = 24$ ns for a **worst case design**.

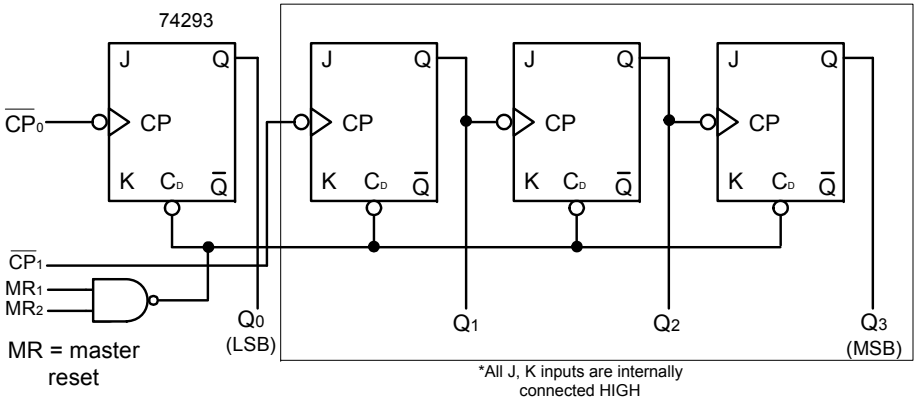
For a **4-bit ripple counter**:

$f_{max} = 1./(4 \times 24 \text{ ns.}) = 10.4$ MHz.

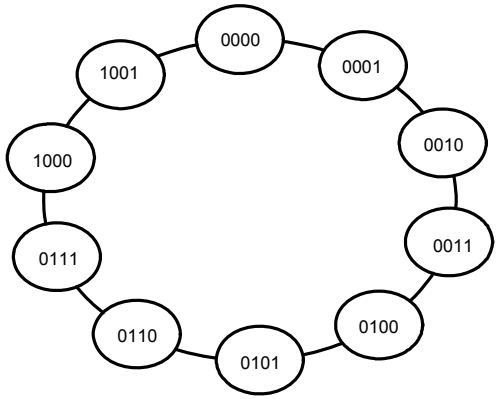
For a **6-bit ripple counter**:

$f_{max} = 1./(6 \times 24 \text{ ns.}) = 6.9$ MHz.

Commercial MSI ripple counter… 74'293

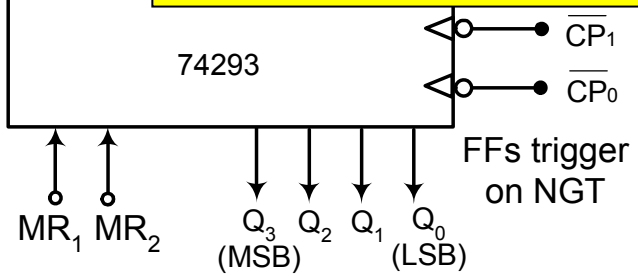


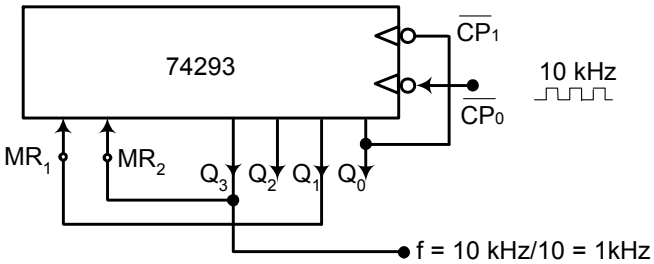
(a)



Practice questions:

1. Wire 74'293 for a mod-14 ctr.
2. Wire 2 74'293's for a mod-60 ctr.



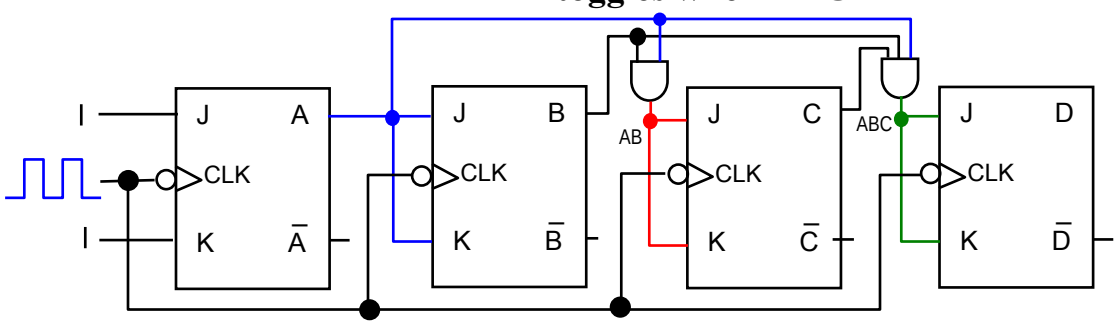


Synchronous (*parallel*) Counters ...

These counters can operate at higher frequencies than ripple counters because all FFs are simultaneously triggered by the clock.

Circuit **below is a mod-16 counter where all FFs are triggered by the same clock & toggles at every ACT (since J=K=1) as follows:**

- **B toggles when A = 1**
- **C toggles when AB = 1**
- **D toggles when ABC = 1**

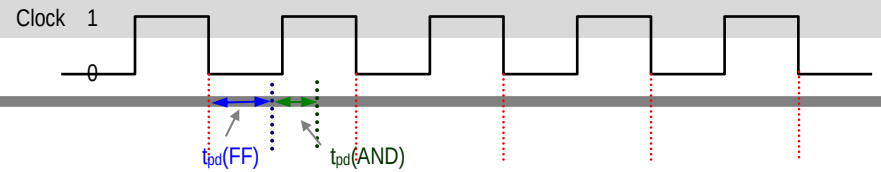


Count	D	C	B	A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
0	0	0	0	0
	c	o	n	t

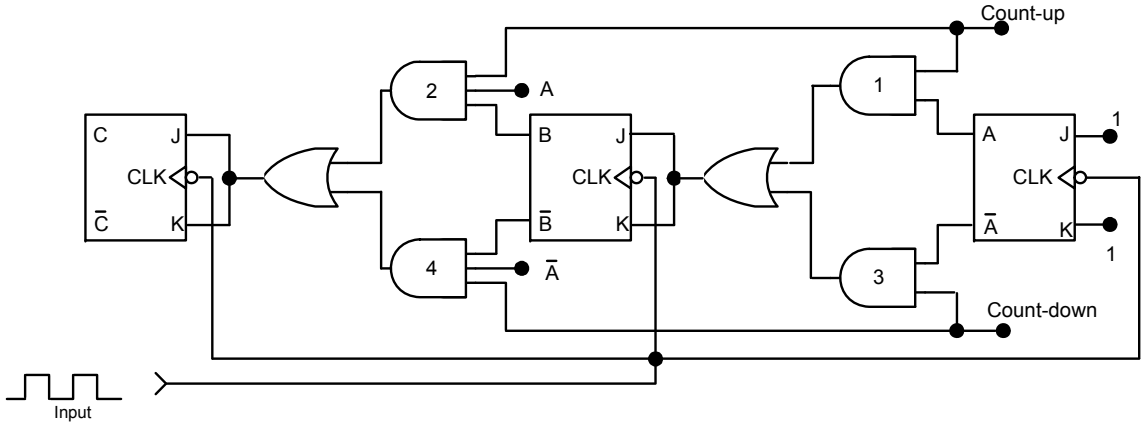
Total delay involved in this synchronous counter: $t_{pd} = t_{pd}(\text{FF}) + t_{pd}(\text{AND})$

Hence it operates at higher frequency than corresponding ripple counter. Also, it is better behaved with respect to the decoding process.

If $t_{pd}(\text{FF}) = 50\text{ns}$ **&** $t_{pd}(\text{AND}) = 20\text{ns} \Rightarrow t_{pd} = 50+20 = 70\text{ns} \Rightarrow T_{clock} \geq 70\text{ns} \Rightarrow$ ***operating frequency*** $f_{max} \leq 1 / 70\text{ns} = 14.3\text{MHz}$ (*irrespective of no. of* FFs). **However, for a 4-bit ripple counter** $f_{max} = 1/(4 \times 50\text{ns}) = 5\ \text{MHz}$.



Up/Down Synchronous Counters ...



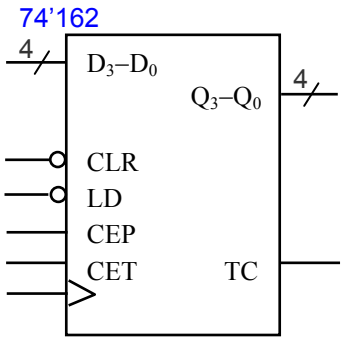
Count-up & **Count-down** signals control the count sequence

Count sequence is **up** if **Count-up** = **1** & **Count-down** = **0** ⇒ FF inputs (*like before*): $J(A)=K(A)=1$, $J(B)=K(B)=A$, and $J(C)=K(C)=AB$.

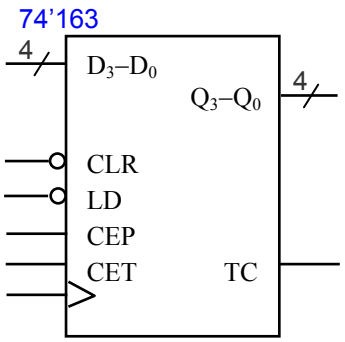
Count sequence is **down** if **Count-up** = **0** & **Count-down** = **1** ⇒ FF inputs: $J(A)=K(A)=1$, $J(B)=K(B)=\bar{A}$, and $J(C)=K(C)=\bar{A}\bar{B}$.

Verify that this indeed makes the counter count down!

MSI counters: 74'162 (decade) & 74'163 (mod-16)



Functional block diagram for the 74'162



Functional block diagram for the 74'163

CLR	LD	CEP	CET	Function
H	H	H	H	Normal count
H	L	X	X	Parallel load
L	X	X	X	Clear

For the 74'162:

$$TC = CET \cdot Q_3 \cdot \overline{Q}_2 \cdot \overline{Q}_1 \cdot Q_0$$

For the 74'163

$$TC = CET \cdot Q_3 \cdot Q_2 \cdot Q_1 \cdot Q_0$$

Functional descriptions

…, 1111, 0000, 0001, 0010, 1011, 1100, 1101, 1110, 1111, 0000, …

↑

during this clock cycle:

LD = 1 and $D_3 - D_0$ = 1011

Synchronous Presetting

←**Example of the LD input function**

…, 1111, 0000, 0001, 0010, 0000, 0001, 0010, 0011, 0100, 0101, …

↑

during this clock cycle:

CLR = 1

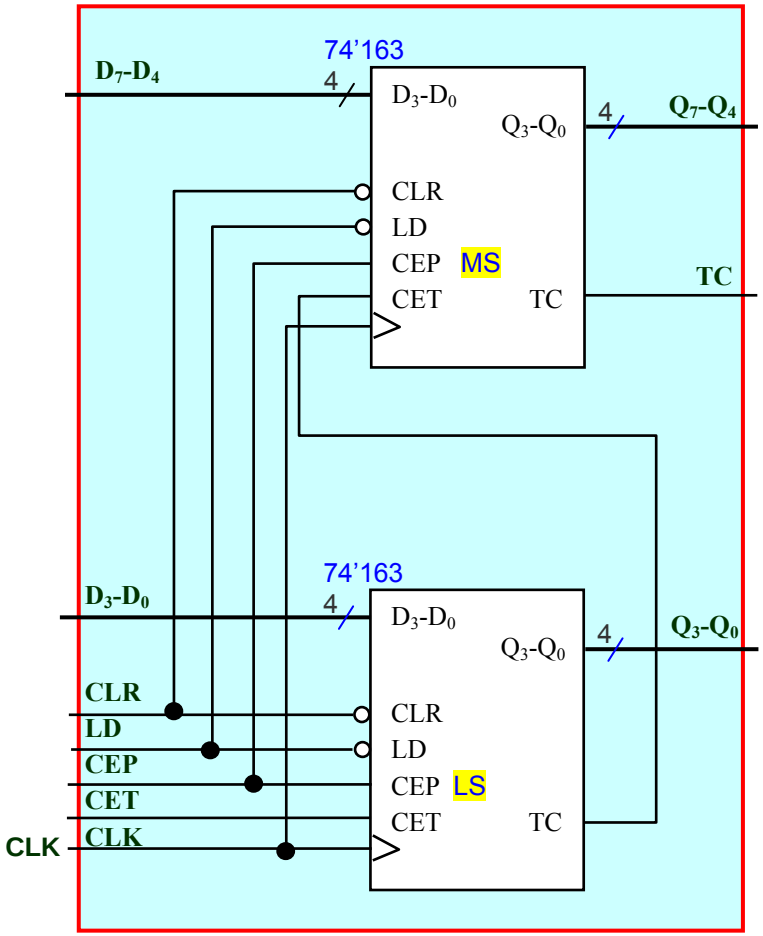
Synchronous Clear

← **Example of the CLR input function**

TC (*Terminal count output*): becomes TRUE at the counts of `1001` (**74'162**) and `1111` (**74'163**) for one clock period ⇒ useful for *cascading* counters.

Cascading connection for two 74'163's to make an 8-bit counter.

8-bit counter formed by cascading two 74'163s …



Notes:

- TC output of the bottom 74'163 is connected to the CET input of the top 74'163.
- Q_7, Q_6,…, Q_3, Q_2,…, Q_0 are the counter bits ordered from MSB to LSB.

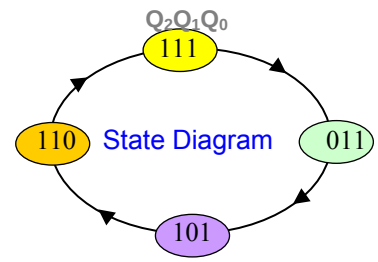
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Dev_74163 is
port (LD, CLR, CEP, CET, CLK: in std_logic;
      D : in std_logic_vector (3 downto 0);
      count : out std_logic_vector (3 downto 0);
      TC : out std_logic);
end Dev_74163;

architecture BEH of Dev_74163 is
 signal Q: std_logic_vector (3 downto 0):= "0000";
 begin
  process (CLK)
   begin
      if CLK'event and CLK ='1' then
        if    CLR = '0' then Q <= "0000";
        elsif LD  = '0' then Q <= D;
        elsif (CEP and CET) = '1' then Q<= Q + 1;
        end if;
      end if;
  end process;
   count <= Q;  -- concurrent outputs
   TC <= Q(3) and Q(2) and Q(1) and Q(0) and CET;
 end BEH;
```

Modeling counters using VHDL … example I

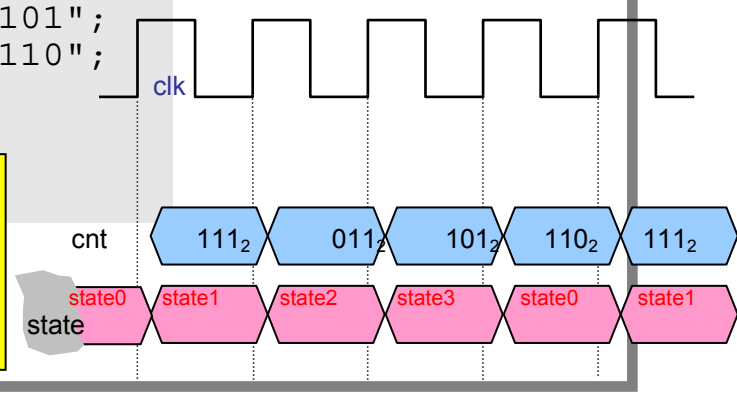
```
library ieee;
use ieee.std logic 1164.all,
use ieee.std logic unsigned.all;
                    -- this works with the Xilinx...
entity counter is
 port (CLK, reset: in std_logic;
        cnt : out std_logic_vector(4 downto 0));
end counter;

architecture beh of counter is
signal  temp : std_logic_vector(4 downto 0);
begin
process(CLK , reset)is  -- synchronous counter with asynchronous reset
 begin                  -- '+' can be applied on std_logic with use of the unsigned function
  if reset='1' then temp <= "00000";
  elsif CLK'event and CLK='1' then
      temp <= temp + 1;  -- '+' does not produce a carry-out!
  end if;
end process;
cnt <= temp;  -- concurrent output
end beh;
```

Modeling counters using VHDL. example II (*using states*)

```
-- OBJECTIVE:  To design a counter with sequence:  ... 111 , 011 , 101 , 110 , 111 , 011 , ....
library ieee;
use ieee.std logic 1164.all,

entity sctr is
 port (clk, rst: in std_logic;
       cnt : out std_logic_vector (2 downto 0));
end sctr;

architecture archsctr of sctr is
    type states is (state0, state1, state2, state3);
    signal state: states; -- user defined type
begin
 process (clk, rst)
    begin
     if rst='1' then state <= state0; cnt <= "110";
      elsif (clk'event and clk='1') then
       case state is
        when state0 => state <= state1; cnt <= "111";
        when state1 => state <= state2; cnt <= "011";
        when state2 => state <= state3; cnt <= "101";
        when state3 => state <= state0; cnt <= "110";
       end case;
      end if;
    end process
end archsctr;
```



Practice question:
Modify this counter such that it accepts an input ID and when ID='1', the sequence is ....'111', '110', '101', '011'... and when ID='0', the sequence is reversed.



FLIP-FLOPS

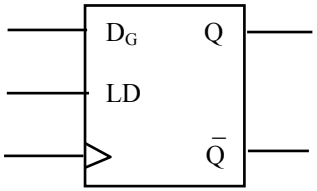
Flip-flop conversions.....

Conversion from J-K FF to T FF was easy and intuitive. How about more *complex* conversions?

Example: **Build a "*gated*" D FF from a J-K FF.**

When a gating signal LD is TRUE the circuit behaves as an ordinary D FF. When LD is FALSE, the D FF does not change state regardless of changing inputs (hold).

What's the functional block diagram of circuit, and truth table?



A gated D Flip-flop functional block diagram

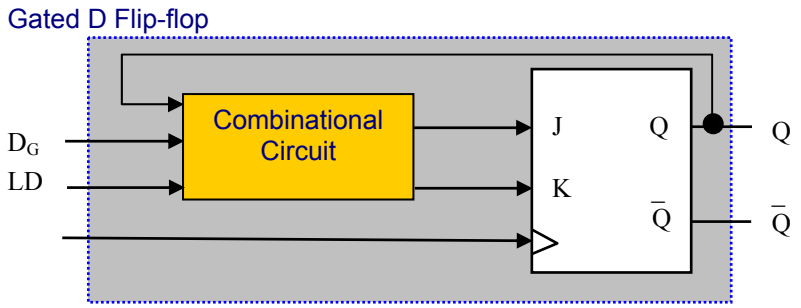
CLK	LD	D_G	Q	Q^+
↑	0	0	0	0
↑	0	0	1	1
↑	0	1	0	0
↑	0	1	1	1
↑	1	0	0	0
↑	1	0	1	0
↑	1	1	0	1
↑	1	1	1	1

truth table

This circuit allows more control over FF operations.

Flip-flop conversions..... continued

The *Design Problem* is as follows:



Block diagram of the *flip-flop* conversion

Given a J-K FF, need to design a combinational circuit that:

- **has inputs D_G, LD, and Q**
- **provides appropriate output values to be used as inputs to the J-K FF which will in turn provide correct output state Q at the next ACT.**

Next, we introduce a *systematic way* to do these kinds of designs.

Flip-flop conversions..... a systematic procedure

Step#1: **Determine the functional block diagram of the combinational circuit.**

Step#2: **Determine the truth table for the combinational circuit.** (*two steps*)

LD	D_G	Q	J	K
0	0	0		
0	0	1		
0	1	0	??	
0	1	1		
1	0	0		
1	0	1		
1	1	0	??	
1	1	1		

Step#2A: **Transform the characteristic table of the source FF into its excitation table.**

Specifies what the FF inputs should be for a specific $Q \rightarrow Q^+$ transition to occur.

Characteristic Table

J	K	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Excitation Table

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Flip-flop conversions..... a systematic procedure. cont

Step#2B: Use the **excitation table** of the source FF to determine the output values for the **truth table** of the combinational circuit.

LD	D_G	Q	Q^+	J	K
0	0	0	0	0	X
0	0	1	1	X	0
0	1	0	0	0	X
0	1	1	1	X	0
1	0	0	0	0	X
1	0	1	0	X	1
1	1	0	1	1	X
1	1	1	1	X	0

Determination of the J and K values for the combinational circuit

LD	D_G	Q	J	K
0	0	0	0	X
0	0	1	X	0
0	1	0	0	X
0	1	1	X	0
1	0	0	0	X
1	0	1	X	1
1	1	0	1	X
1	1	1	X	0

Final truth table for the combinational circuit

The combinational circuit is now completely specified in terms of its truth table!

Flip-flop conversions..... a systematic procedure. cont

Step#3: **Realize the combinational circuit.**

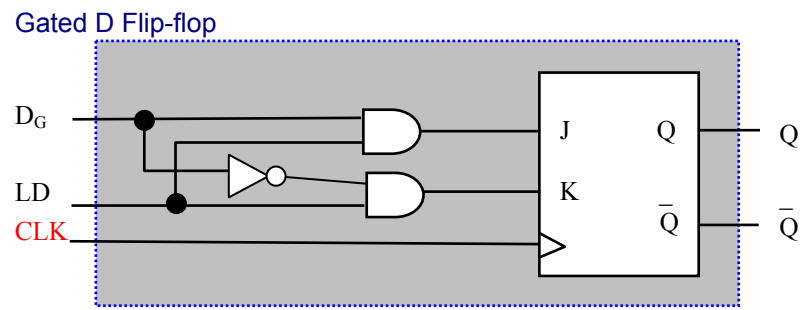
D_G Q \ LD	0	1
0 0	0	0
0 1	X	X
1 1	X	X
1 0	0	1

$J = LD \cdot D_G$

D_G Q \ LD	0	1
0 0	X	X
0 1	0	1
1 1	0	0
1 0	X	X

$K = LD \cdot \overline{D_G}$

K-maps for the **J** & **K** outputs



Final circuit diagram

Once circuit is realized, the circuit operation should be verified with respect to given specifications.

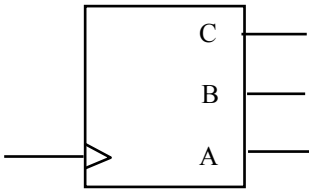
Procedure can be used to convert any type of FF to any other *type*.

This basic procedure can be used to design more complex sequential circuits.

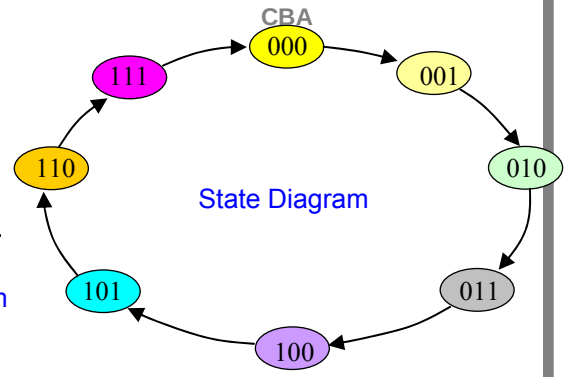
Design Method for Synchronous Counters …

Goal: **Given the state diagram of a counter realize it using common FFs (and *combinational logic*).**

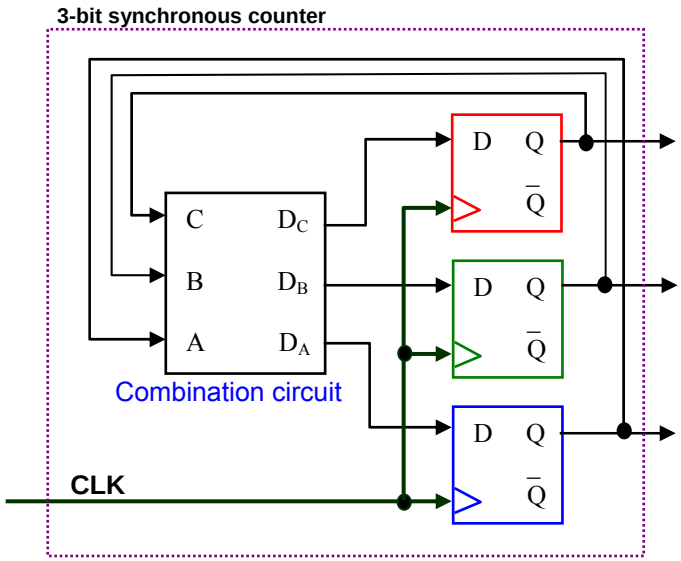
Example: **Design a 3-bit counter having the following state diagram. Use D FFs.**



Functional block diagram of 3-bit counter.



The functional block diagram can be expanded out as shown below.



FF outputs are **fed back** to combinational circuit inputs.

Combinational circuit outputs D_A, D_B, & D_C are connected to D FF inputs and at next ACT, these will be transferred to the output.

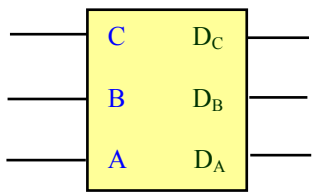
Key: Design combinational circuit to **take previous counter outputs** & **produce the next state**.

Systematic design method is similar to that used for FF conversion considered before.

Design Method for Synchronous Counters … cont

Design Steps :

1. Determine the ***count sequence*** from the state diagram.
2. Determine the ***functional block diagram*** of the ***N***-bit counter to be designed.
   - o consists of ***N*** **flip-flops**.
   - o a **combinational circuit** for generating valid **FF** inputs.
3. Determine the ***functional block diagram*** of the **combinational circuit** (readily extracted from step 2).



Combinational circuit:

Inputs: Present-state counter outputs (A,B,C).

Outputs: D_A, D_B, D_C to connect to FF inputs.

4. Obtain the **truth-table** of the **combinational circuit**. This is done in 2 parts.
   - o determine the ***next state table*** for the counter.
   - o from the ***next state table*** determine the required **FF** inputs using the ***excitation table*** of the chosen **FF**s.

Design Method for Synchronous Counters … cont

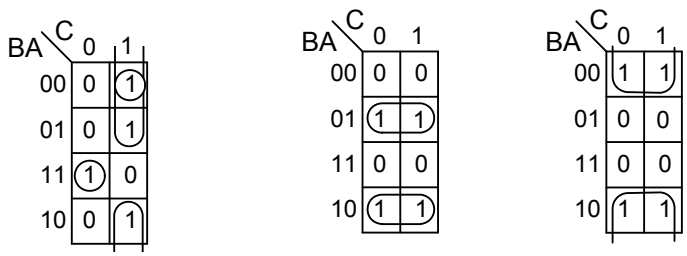
Present-state outputs			Next-state outputs			Required flip-flop inputs		
C	B	A	C^+	B^+	A^+	D_C	D_B	D_A
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0

Next-state table

Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

D flip-flop excitation table

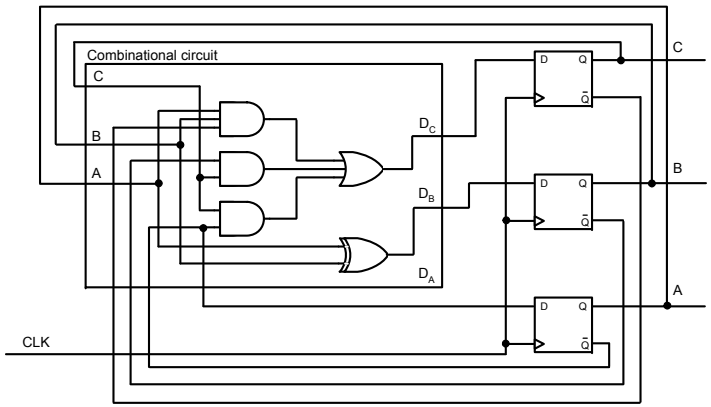
5. Realize the combinational circuit.



$D_C = AB\overline{C} + \overline{B}C + \overline{A}C$ $\quad D_B = A\overline{B} + \overline{A}B = A \oplus B$ $\quad D_A = \overline{A}$

K-maps for flip-flop inputs

Synchronous 3-bit counter



Circuit Diagram

A **synchronous counter** can be realized with **J-K FFs** or with any other **FF**

Design Method for Synchronous Counters … example.

Design a synchronous counter with count sequence:

101,001,000,010,110,100,101,… (mod-6).

The counter also has an external CLEAR input, which when TRUE will clear the counter to 000 at next ACT.

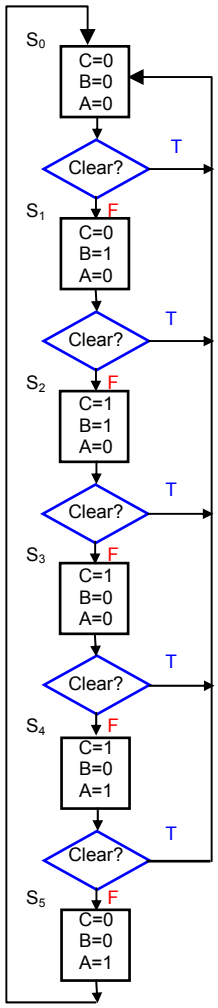
e.g. 101,001,000,010,110,100,000,010..

↑ CLEAR

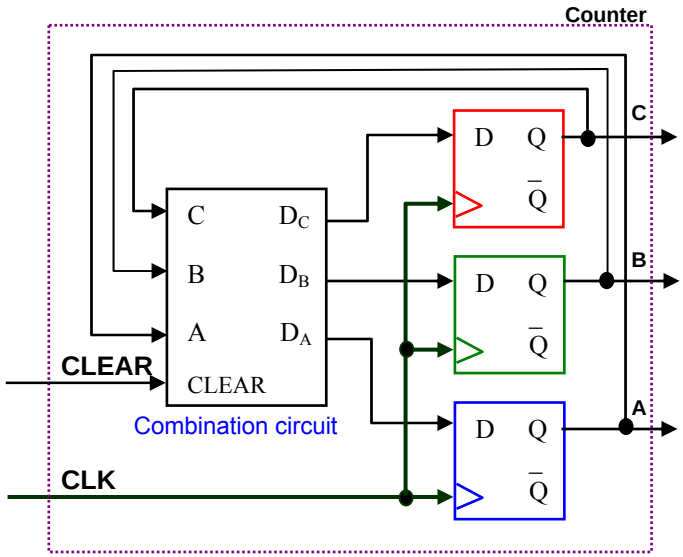
Step1: Write state diagram.

Step2: Determine functional block diagram of counter.

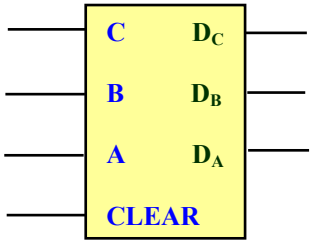
Step3 : Functional block diagram of combinational cct– extract from above.



STATE DIAGRAM



Functional block diagram of counter



Step4 : Get TT of combinational circuit using FF excitation table.

Step5 : Realize circuit.

CLEAR	C	B	A	C^+	B^+	A^+	D_C	D_B	D_A
0	0	0	0	0	1	0	0	1	0
0	0	0	1	0	0	0	0	0	0
0	0	1	0	1	1	0	1	1	0
0	0	1	1	X	X	X	X	X	X
0	1	0	0	1	0	1	1	0	1
0	1	0	1	0	0	1	0	0	1
0	1	1	0	1	0	0	1	0	0
0	1	1	1	X	X	X	X	X	X
1	X	X	X	0	0	0	0	0	0

Next-state table with required flip-flop inputs

$$D_C = \overline{CLEAR} \bullet B + \overline{CLEAR} \bullet C \bullet \overline{A}$$
$$D_B = \overline{CLEAR} \bullet \overline{C} \bullet \overline{A}$$
$$D_A = \overline{CLEAR} \bullet C \bullet \overline{B}$$

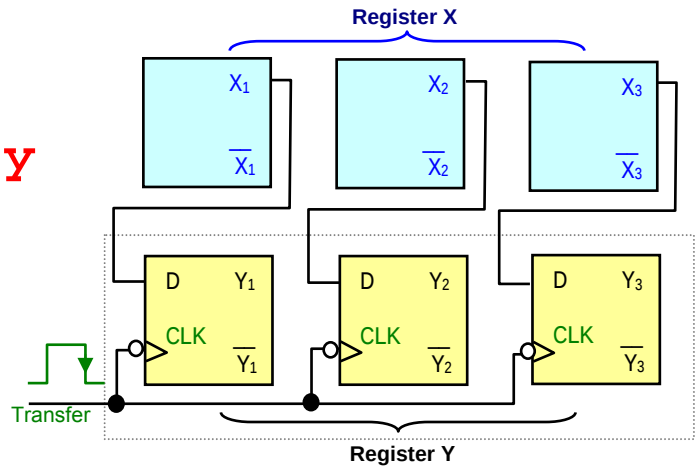
Logic equations for flip-flop inputs

Registers … consists of FFs

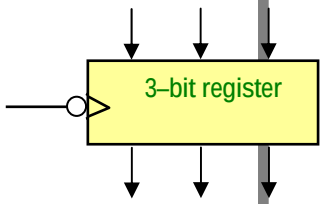
A **register** is a circuit element which consists of a number of FFs (*storage elements*) that are connected together.

Their main functions are to *store data* (*storage register*) or to *convert data from one form to another*, e.g., `parallel to serial` or vice-versa (*shift register*).

Data can be loaded into a **register** either `serially` or in `parallel`.



Data from FFs in register X is transferred (synchronously) in `parallel` to FFs in register Y.

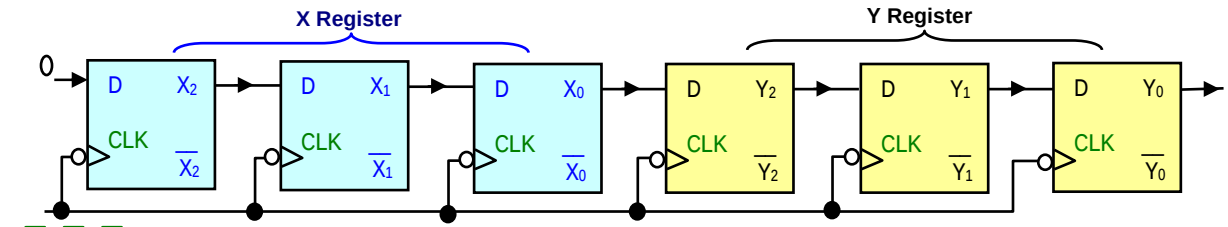


Functional block diagram of 3-bit register.

Registers … continued

Model this using VHDL!

Synchronous serial data transfer:
(from register X to register Y):



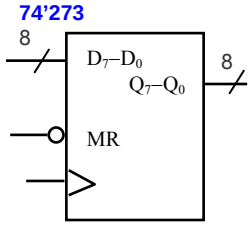
X_2	X_1	X_0	Y_2	Y_1	Y_0	
1	0	1	0	0	0	Before pulses applied
0	1	0	1	0	0	After first pulse
0	0	1	0	1	0	After second pulse
0	0	0	1	0	1	After third pulse

Registers are available with a number of different features, but basic classifications are ⇒

Serial in	Serial out
Serial in	Parallel out
Parallel in	Parallel out
Parallel in	Serial out

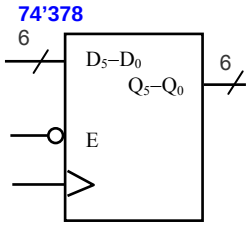
Registers … commercially available registers..

Some commercially available MSI registers:



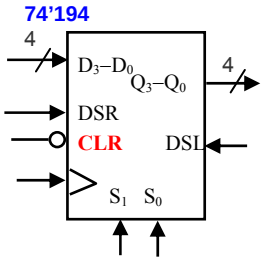
CLK	MR	D_i	Q_i	Q_i^+
X	L	X	X	L
↑	H	H	X	H
↑	H	L	X	L

74'273 8-bit storage register



CLK	E	D_i	Q_i	Q_i^+
X	H	X	L	L
X	H	X	H	H
↑	L	H	X	H
↑	L	L	X	L

74'378 6-bit storage register

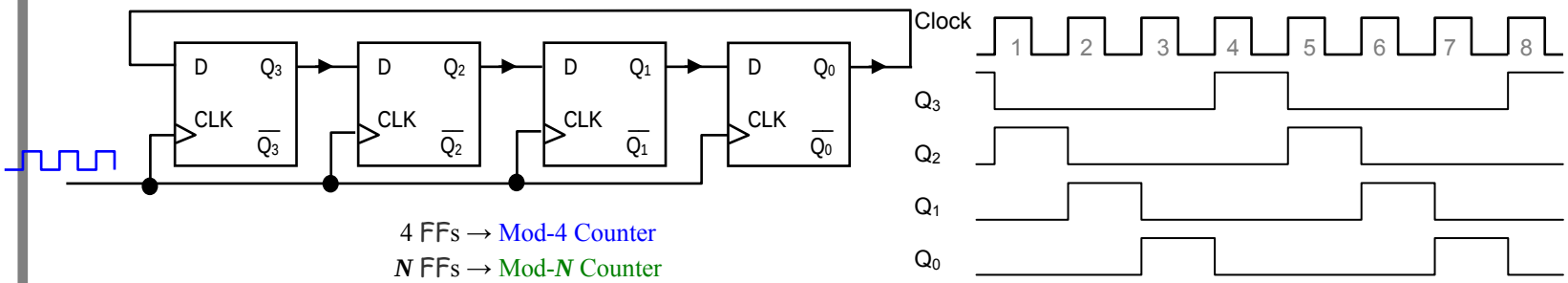


Operation	CLK	CLR	S_1	S_0	DSR	DSL	Q_3^+	Q_2^+	Q_1^+	Q_0^+
Clear	X	L	X	X	X	X	L	L	L	L
Hold	↑	H	L	L	X	X	Q_3	Q_2	Q_1	Q_0
Shift right	↑	H	L	H	L	X	L	Q_3	Q_2	Q_1
	↑	H	L	H	H	X	H	Q_3	Q_2	Q_1
Shift left	↑	H	H	L	X	L	Q_2	Q_1	Q_0	L
	↑	H	H	L	X	H	Q_2	Q_1	Q_0	H
parallel load	↑	H	H	H	X	X	D_3	D_2	D_1	D_0

Bidirectional universal shift register.

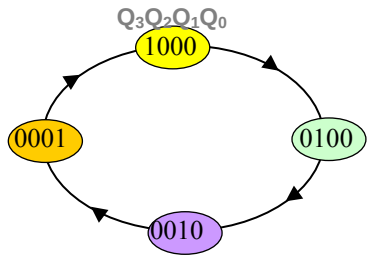
Register Based Counters … *Ring Counter*

Ring Counter: a circulating arrangement where a single 1 moves from FF to FF.



4 FFs → Mod-4 Counter
N FFs → Mod-*N* Counter

Q_3	Q_2	Q_1	Q_0	Clock pulse
1	0	0	0	0
0	1	0	0	1
0	0	1	0	2
0	0	0	1	3
1	0	0	0	4
0	1	0	0	5
0	0	1	0	6
0	0	0	1	7
.	.	.	.	.
.	.	.	.	.



In most instances, only a single 1 circulates.

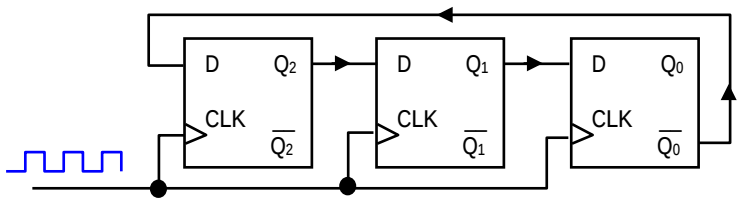
Initialize counter by *presetting* a 1 into one FF and *clearing* the rest.

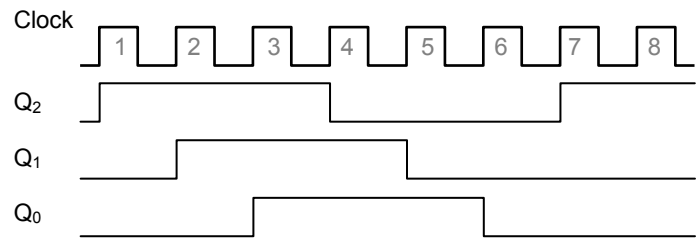
Mod-*N* counter needs *N* FFs ⇒ more hardware *than* other counters for the same mod-#.

On the other hand, **this counter does not need any decoding gates at all (savings).**

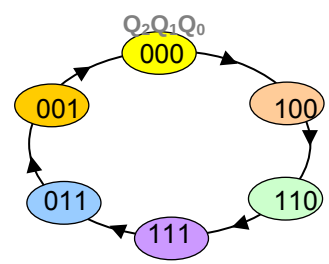
Register Based Counters … *Johnson counter*

Johnson counter: a ring counter with a twist. This is a mod-6 counter that does not count in the ordinary binary sequence.





Q_2	Q_1	Q_0	Clock pulse
0	0	0	0
1	0	0	1
1	1	0	2
1	1	1	3
0	1	1	4
0	0	1	5
0	0	0	6
1	0	0	7
1	1	0	8
.	.	.	.
.	.	.	.



Mod-# = 2 x number of FFs, i.e. Mod-N Johnson counter needs $^{N}/_{2}$ FFs ⇒ *half the number that a ring counter needs*.

But a Johnson counter needs decoders to decode its output

Self Study Section

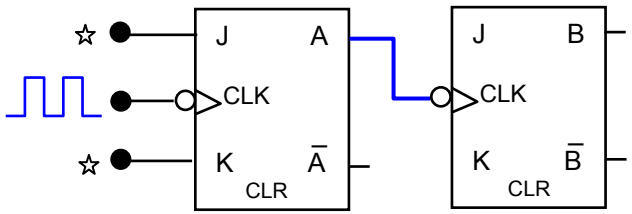
More counters..

Timing considerations..

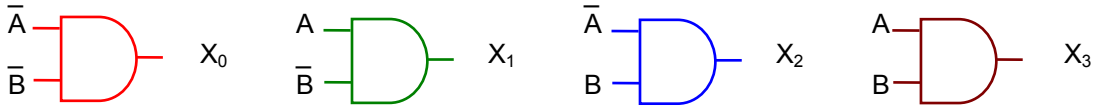
Asynchronous ripple counter… decoding glitches

Propagation delays associated with ripple counters could give rise to decoding glitches as shown in the example below.

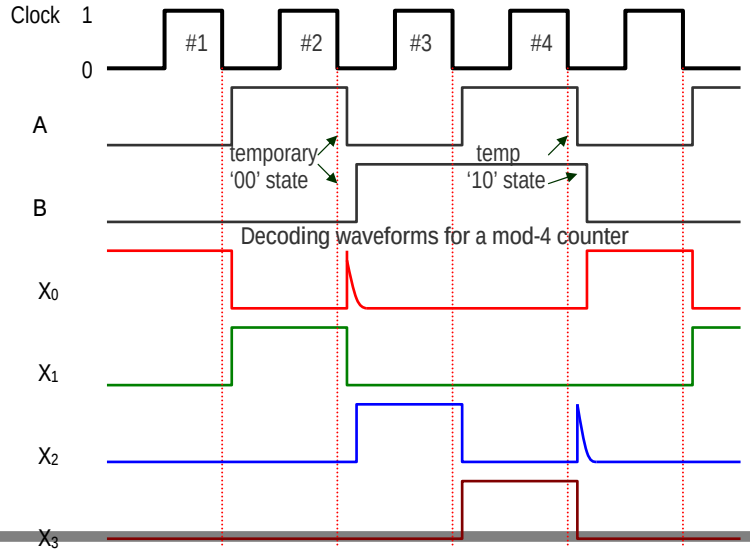
Mod-4 Counter



Decoder gates:



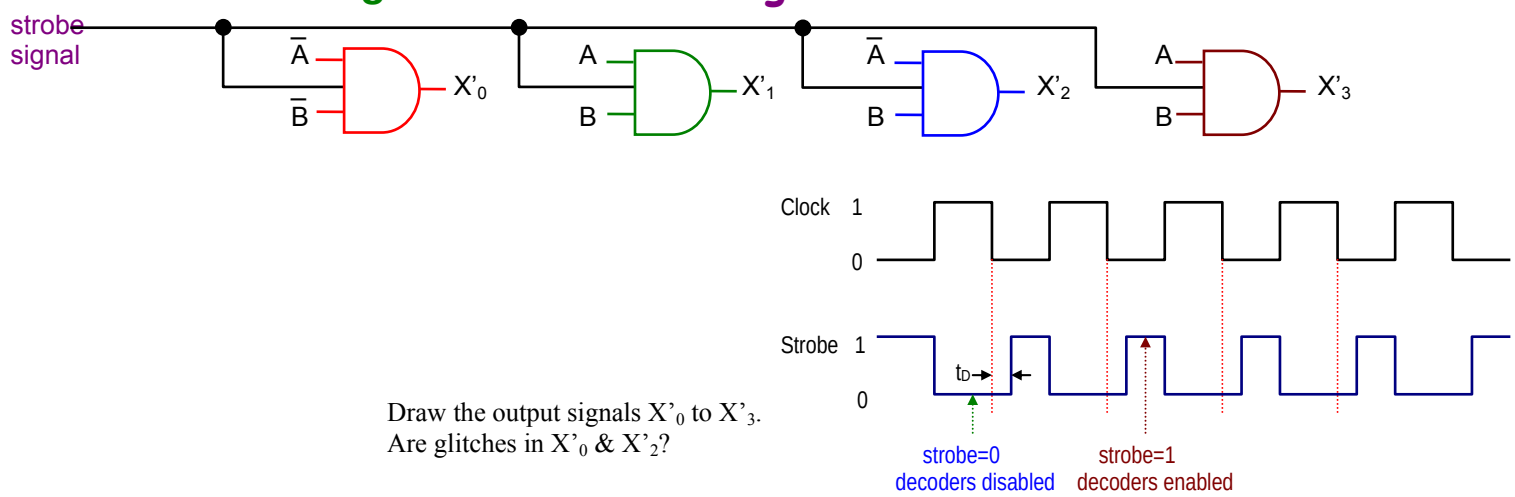
Glitches in X_0 & X_2:



Asynchronous ripple counter… decoding glitches

Decoding glitches can be eliminated by a strobe signal with pulse width t_D which is greater than the total time it takes the counter to reach a stable count (depends on FF delays & # of FFs).

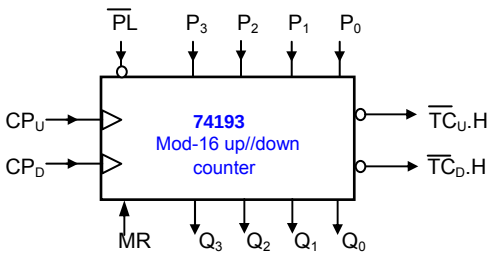
Decoder gates with strobe signal:



Draw the output signals X'_0 to X'_3.
Are glitches in X'_0 & X'_2?

This method need not be used when decoder drives a display – glitch is not visible. But must be used when decoder drives other circuitry.

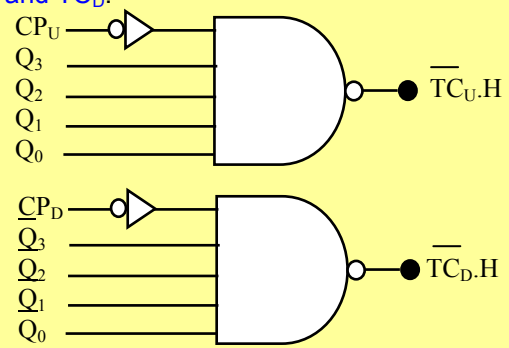
Synchronous Pre-settable Up/Down counter: 74'193



Pin names	Description
CP_U	Count-Up clock input (active rising edge)
CP_D	Count-down clock input (active rising edge)
MR	Asynchronous master reset input (active HIGH)
$\overline{PL}$.H	Asynchronous parallel load input (active LOW)
P_0–P_3	Parallel data inputs
Q_0–Q_3	Flip-flop outputs
$\overline{TC_D}$.H	Terminal count-down (borrow) output (active LOW)
$\overline{TC_u}$.H	Terminal count-up (carry) output (active LOW)

Notes:

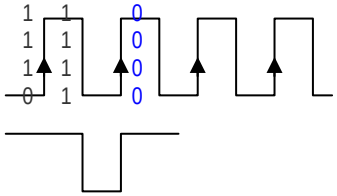
- MR resets counter to `0000` state. It is a DC reset, i.e. it will hold counter at `0` as long as MR = `1`, overriding all other inputs.
- PL: when it pulses `TRUE` (active low), it presets counter to value of P_3, P_2, P_1, P_0, asynchronously.
- TC_U and TC_D:



Mode Select

MR	$\overline{PL}$	CP_U	CP_D	Mode
H	X	X	X	Asynch. reset
L	L	X	X	Asynch. preset
L	H	H	H	No change
L	H	↑	H	Count up
L	H	H	↑	Count down

H = HIGH; L = LOW
X = Don't care; ↑ = PGT



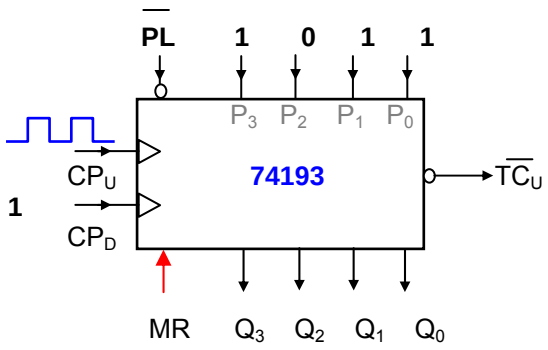
Clock & States

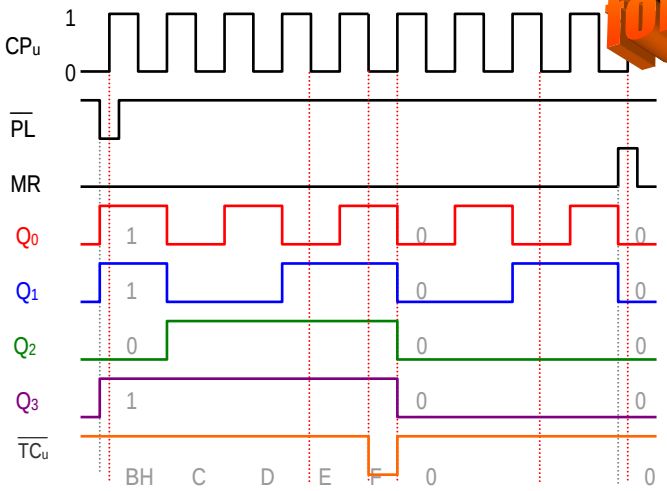
$\overline{TC_U}$ to clock another counter synchronously

Pre-settable Up/Down counter: 74'193 .. configurations

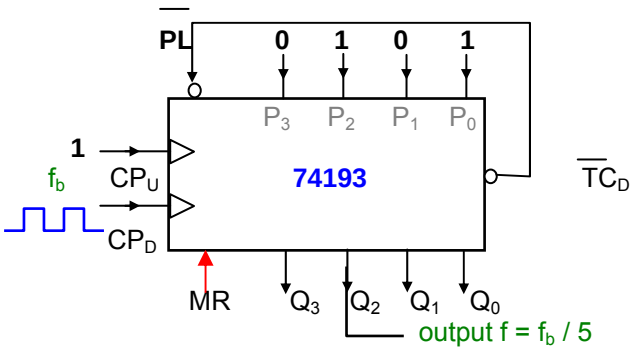
Another counter for self-study !

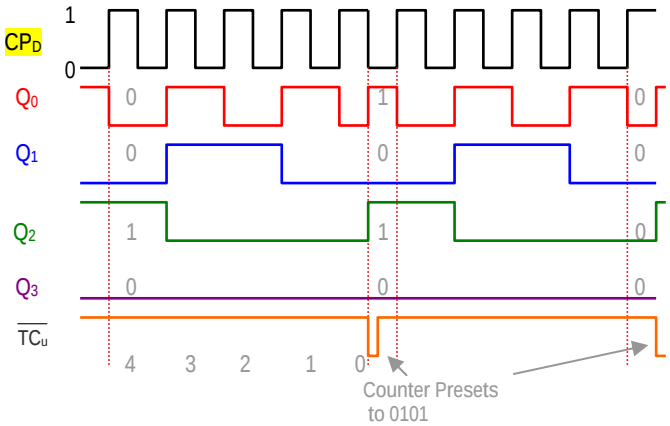
The **74'193** wired up as an Up counter with parallel inputs at **1011**.



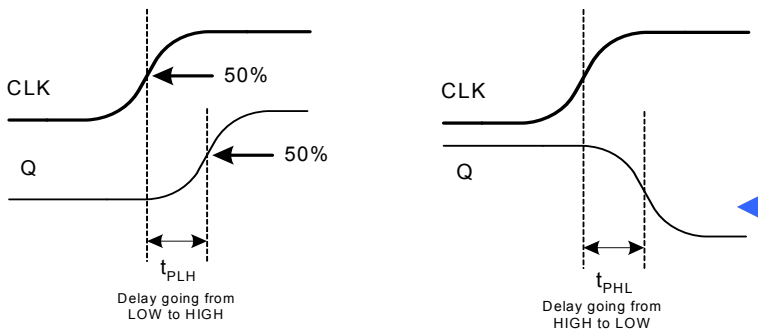


The **74'193** wired up as a mod-5 counter:



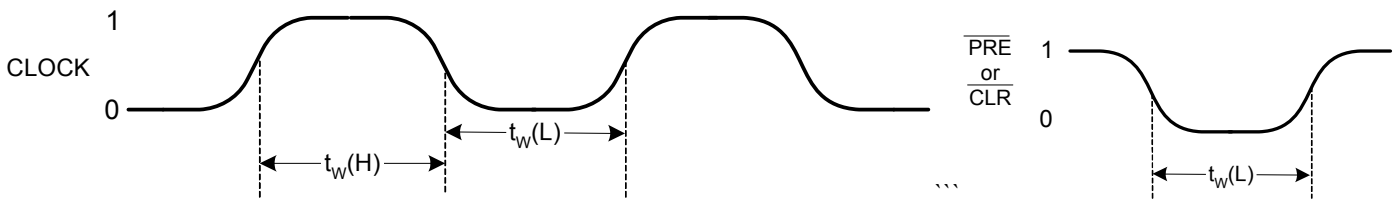


FF Timing Considerations …



Setup & hold times were considered earlier. Data books provide minimum values ≈10's of ns

Propagation delays **(maximum values specified)**



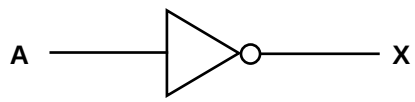
Maximum clocking frequency**: highest clocking frequency that may be used & still have the FFs trigger reliably.**

Clock pulse **HIGH/LOW** ***times*****: the minimum time period for clock to remain LOW before going HIGH, $t_{W(L)}$. Also, clock to stay HIGH before going LOW, $t_{W(H)}$.**

Asynchronous active pulse width**: minimum time duration for which the asynchronous PRESET/CLEAR signals must be active for reliable triggering.**

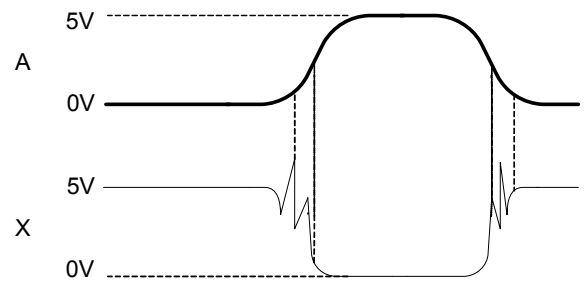
Timing Considerations … cont.

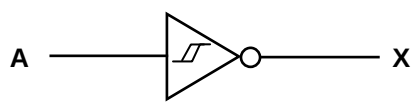
***Clock transition times*: transition time from one level to another must not exceed maximum value specified. If there is a problem with transition times, it can be rectified using Schmitt triggers.**



Standard INVERTER

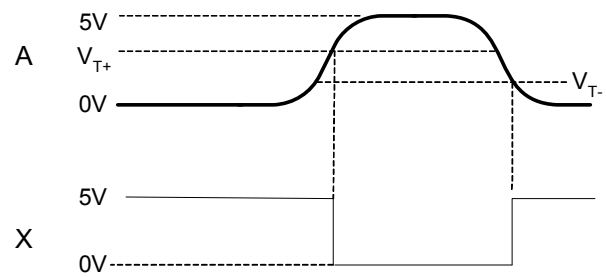
Oscillations may occur on output if input transition times are too slow

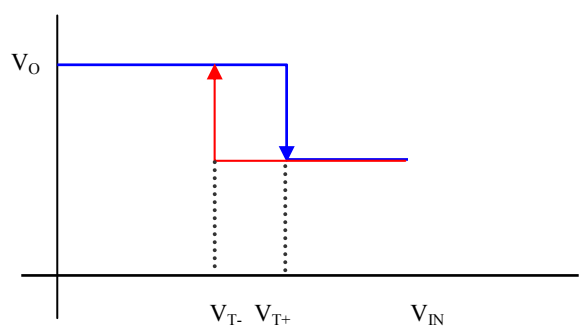




Schmitt-trigger INVERTER

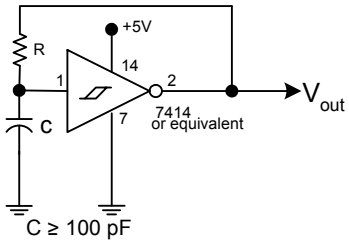
Output has clean, fast transitions independent of input transition times





Device hysterisis characteristics.

Clock Generation (with *Schmitt trigger* inverter)



Values of R and C determine clock frequency in above Schmitt trigger-based clock generator.

IC	Frequency	
7414	≈ 0.8/RC	(R ≤ 500Ω)
74LS14	≈ 0.8/RC	(R ≤ 2kΩ)
74HC14	≈ 1.2/RC	(R ≤10MΩ)

Clock generation with *Schmitt trigger* inverter

Clocks can also be conveniently generated using the `LM555 chip`.

Clock Skew:

- **Clock** arrives at different times at different sequential circuit elements, when it should really be present simultaneously at all elements ⇒ *can cause problems.*
- Main reason for this is that **clock** may be going through additional gates before it reaches the circuit elements ⇒ *can add unequal propagation delays.*
- *Solution* (?) – *try to equalize delays using dummy elements* (*such as inverters*). But this is a bad fix for a badly designed system.
- The **clock** is probably the most important element in a system, and *must be delivered to all circuit elements at the same time.*
- As much as possible design process must ensure that this is the case ⇒ *don't gate the clock!*

Synchronous vs. Asynchronous Circuit Designs

Synchronous operations generally preferred to asynchronous operations, since latter require great care to address problems such as:

- *Races due to unequal path delays.*
- *Transients and glitches which can cause incorrect operation.*
- *Output changes that depend on order of asynchronous input changes*

Synchronous circuits bypass these problems by use of the clock which allows outputs to change only at discrete time instants.

- *This allows time for transients and glitches to settle down, races to be resolved etc.*
- *Design procedures considered later will isolate signal changes in the early part of the clock cycle to ensure stable predictable operation.*

Handling asynchronous inputs (more on this in ASMs):

- *Never connect asynchronous inputs to more than one* **FF***; synchronize as soon as possible and then treat as synchronous signal*