香港中文大學
The Chinese University of Hong Kong

CENG3430 Rapid Prototyping of Digital Systems

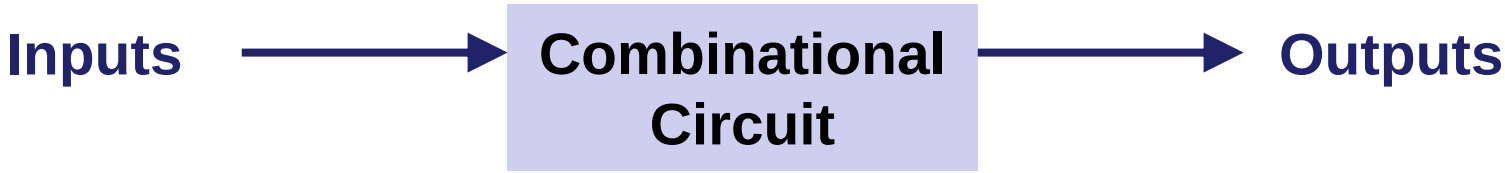
Lecture 03: Combinational Circuit and Sequential Circuit

Ming-Chang YANG

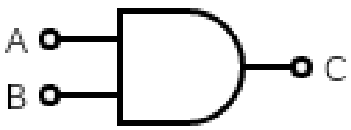
mcyang@cse.cuhk.edu.hk

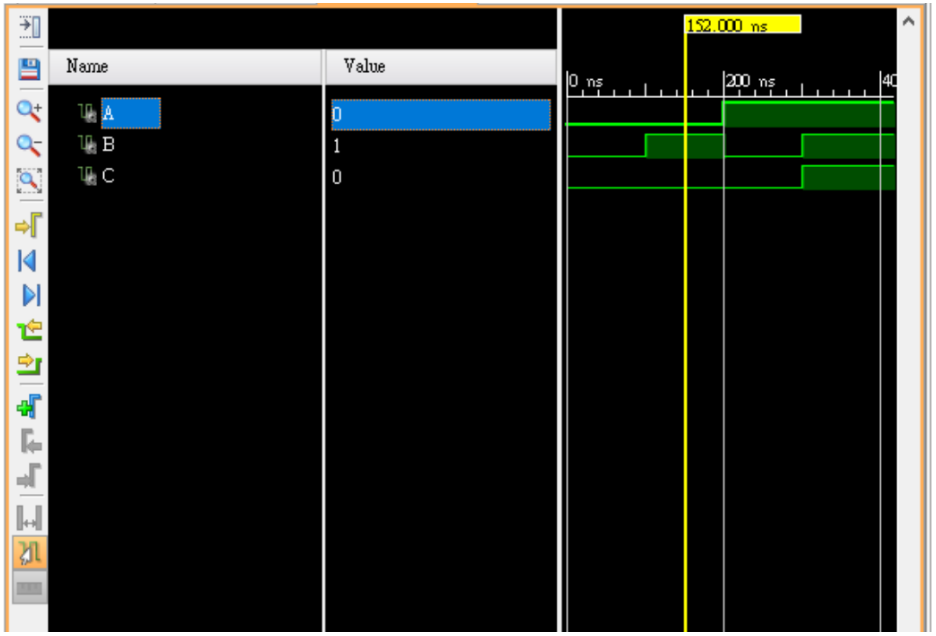
What we have learnt so far

- **Combinational Circuit**: **no memory**
  - Outputs depend on the *present* inputs only.
    - As soon as inputs change, the values of previous outputs are **lost**.



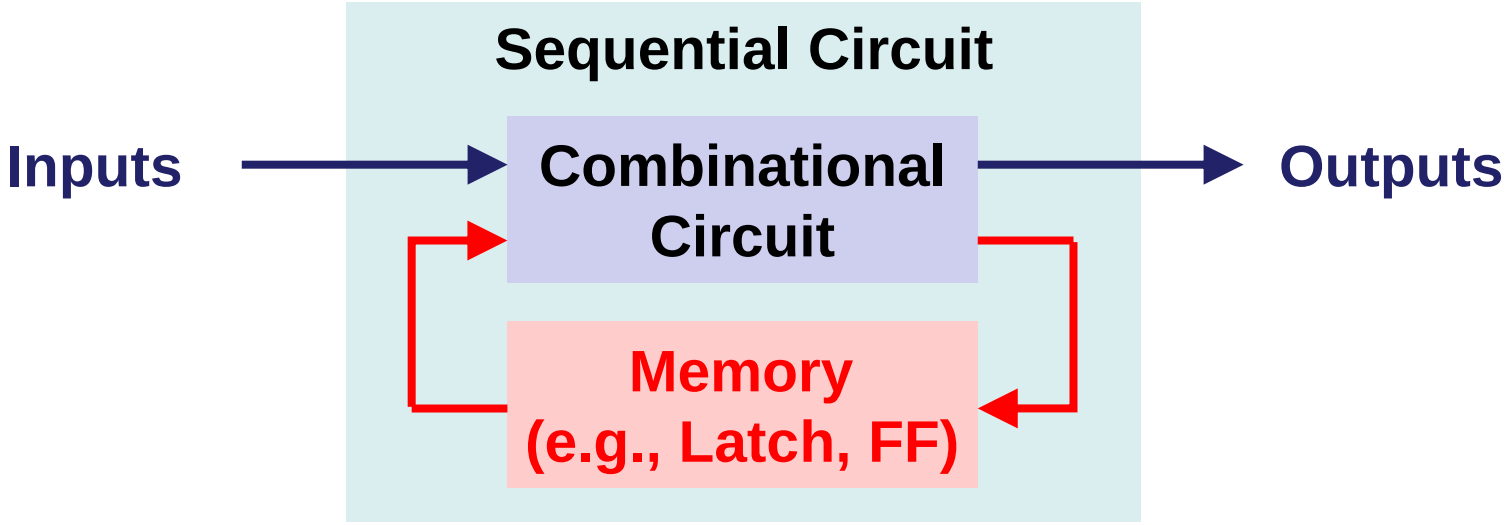

```
entity lab0 is
  port (
    A: in std_logic;
    B: in std_logic;
    C: out std_logic
  );
end lab0;
architecture behavioral of lab0 is
begin
    C <= A AND B;
end behavioral;
```





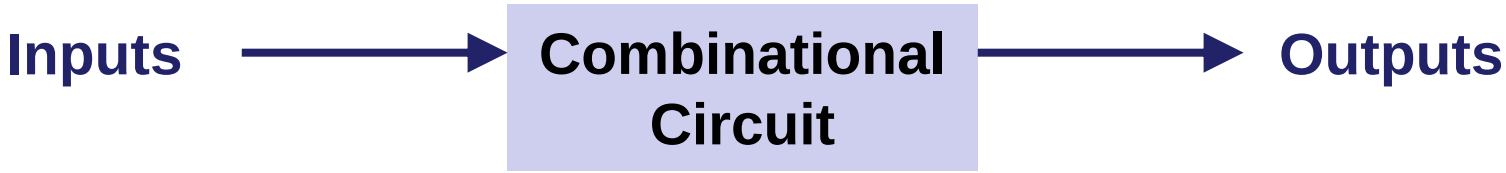
Outline

- **Combinational Circuit**: **no memory**
  - Outputs depend on the *present* inputs only.
  - **Rule**: Use **either** concurrent **or** sequential statements.
- **Sequential Circuit**: **has memory**
  - Outputs depend on *present* inputs and *previous* outputs.
  - **Rule**: **MUST** use sequential statements (i.e., `process`) .



Combinational Circuit

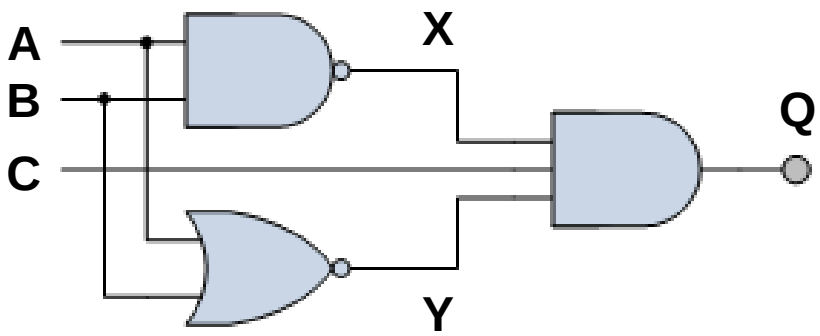
- **Combinational Circuit**: **no memory**
  - Outputs depend on the *present* inputs only.
    - As soon as inputs change, the values of previous outputs are **lost**.
  - **Rule**: You can build a combinational circuit using **either** concurrent statements (i.e., statements outside `process`) **or** sequential statements (i.e., statements inside `process`) .



Modeling Combinational Circuit (1/3)

- Typical ways for modeling a combinational circuit:
  1) **Logic/Schematic Diagram** shows the wiring and connections of each individual logic gate.
  2) **Boolean Expression** is an expression in Boolean algebra that represents the logic circuit.
  3) **Truth Table** provides a concise list that shows all the output states for each possible combination of inputs

Logic/Schematic Diagram



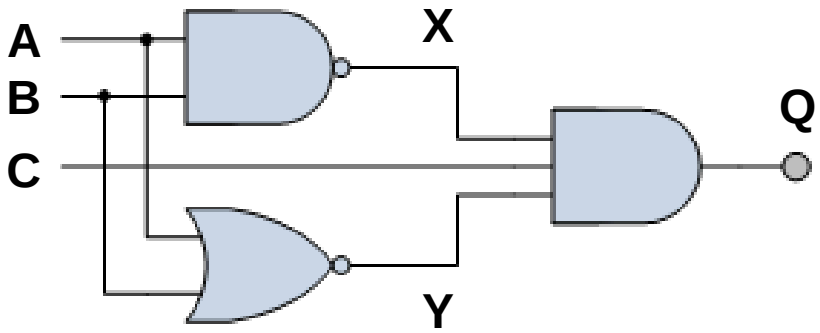

```
architecture com_arch of comb is
signal X, Y: std_logic;
begin
  X <= not (A and B);
  Y <= not (A or B);
  Q <= (X and C) and Y;
end com_arch;
```

https://www.electronics-tutorials.ws/combination/comb_1.html

Modeling Combinational Circuit (2/3)

- Typical ways for modeling a combinational circuit:
  1) **Logic/Schematic Diagram** shows the wiring and connections of each individual logic gate.
  2) **Boolean Expression** is an expression in Boolean algebra that represents the logic circuit.
  3) **Truth Table** provides a concise list that shows all the output states for each possible combination of inputs

Logic/Schematic Diagram



Boolean Expression

$$Q = \overline{(A \cdot B)} \cdot \overline{(A + B)} \cdot C$$

```
architecture com_arch of comb is
begin
  Q <= (not (A and B)) and
       (not (A or B)) and C;
end com_arch;
```

https://www.electronics-tutorials.ws/combination/comb_1.html

Modeling Combinational Circuit (3/3)

- Typical ways for modeling a combinational circuit:
  1) **Logic/Schematic Diagram** shows the wiring and connections of each individual logic gate.
  2) **Boolean Expression** is an expression in Boolean algebra that represents the logic circuit.
  3) **Truth Table** provides a concise list that shows all the output states for each possible combination of inputs

A	B	C	Q
0	0	0	0
0	**0**	**1**	**1**
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

```
process (A, B, C)
begin
   if( A = '0' and B = '0' and C = '1' ) then
     Q <= '1';
   else
     Q <= '0';
   end if;
end process;
```

Comb. Circuit Example: Decoder (1/2)

- A **decoder** is designed to converts binary information from the **n** inputs to a maximum of 2^n unique outputs.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity decoder_ex is
port (in0,in1: in std_logic;
      out00,out01,out10,out11: out std_logic);
end decoder_ex;
architecture decoder_ex_arch of decoder_ex is
begin
  process (in0, in1)
  begin
    if in0 = '0' and in1 = '0' then
      out00 <= '1';
    else
      out00 <= '0';
    end if;
```

out00

in 0	in 1	out 00	out 01	out 10	out 11
0	**0**	**1**	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Comb. Circuit Example: Decoder (2/2)

(cont'd)

```
if in0 = '0' and in1 = '1' then
  out01 <= '1';
else                                  out01
  out01 <= '0';
end if;
if in0 = '1' and in1 = '0' then
  out10 <= '1';
else                                  out10
  out10 <= '0';
end if;
if in0 = '1' and in1 = '1' then
  out11 <= '1';
else                                  out11
  out11 <= '0';
end if;
  end process;
end decoder_ex_arch;
```

in 0	in 1	out 00	out 01	out 10	out 11
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Class Exercise 3.1

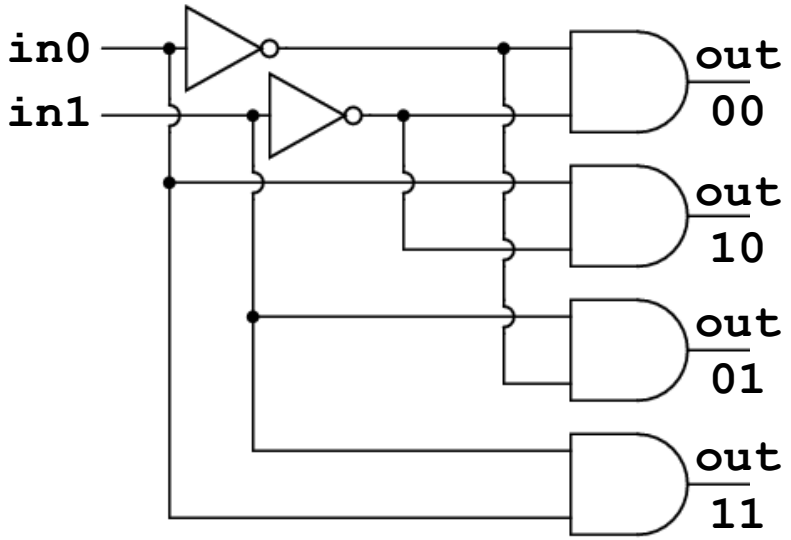
- Re-implement the 2-to-4 decoder by referring the provided schematic diagram:

in 0	in 1	out 00	out 01	out 10	out 11
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity decoder_ex is
  port( in0,in1: in std_logic;
        out00,out01,out10,out11:
        out std_logic);
end decoder_ex;
architecture decoder_ex_arch of
decoder_ex is
begin




end decoder_ex_arch;
```



Comb. Circuit Example: Multiplexer

- A **multiplexer** is designed to switch one of several input lines through to a single common output line.

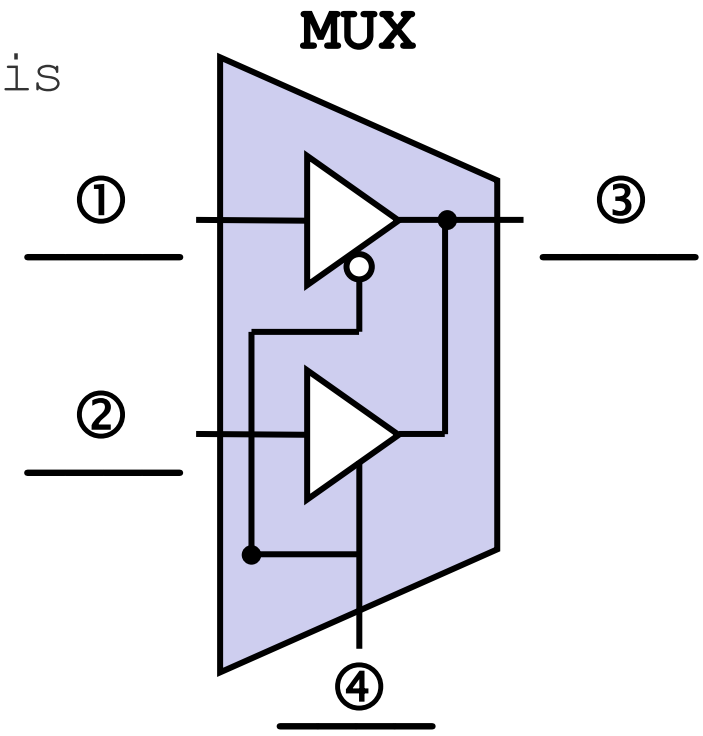
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mux_ex is
port (in1,in2,sel: in std_logic;
                out1: out std_logic);
end mux_ex;
architecture mux_ex_arch of mux_ex is
begin
  process (in1, in2, sel)
  begin
    if sel = '0' then
      out1 <= in1; -- select in1
    else
      out1 <= in2; -- select in2
    end if;
  end process;
end mux_ex_arch;
```

sel	in1	in2	out1
0	0	0	0
	0	1	0
	1	0	1
	1	1	1
1	0	0	0
	0	1	1
	1	0	0
	1	1	1

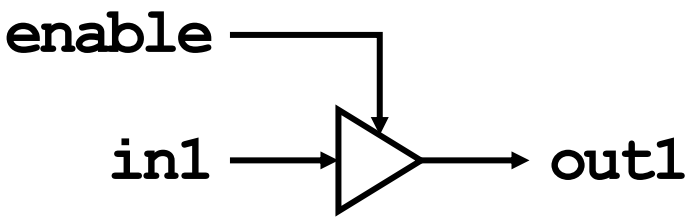
Class Exercise 3.2

- Specify the I/O signals in the schematic diagram:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mux_ex is
port (in1,in2,sel: in std_logic;
                out1: out std_logic);
end mux_ex;
architecture mux_ex_arch of mux_ex is
begin
  process (in1, in2, sel)
  begin
    if sel = '0' then
      out1 <= in1; -- select in1
    else
      out1 <= in2; -- select in2
    end if;
  end process;
end mux_ex_arch;
```



Recall: Tri-state Buffer

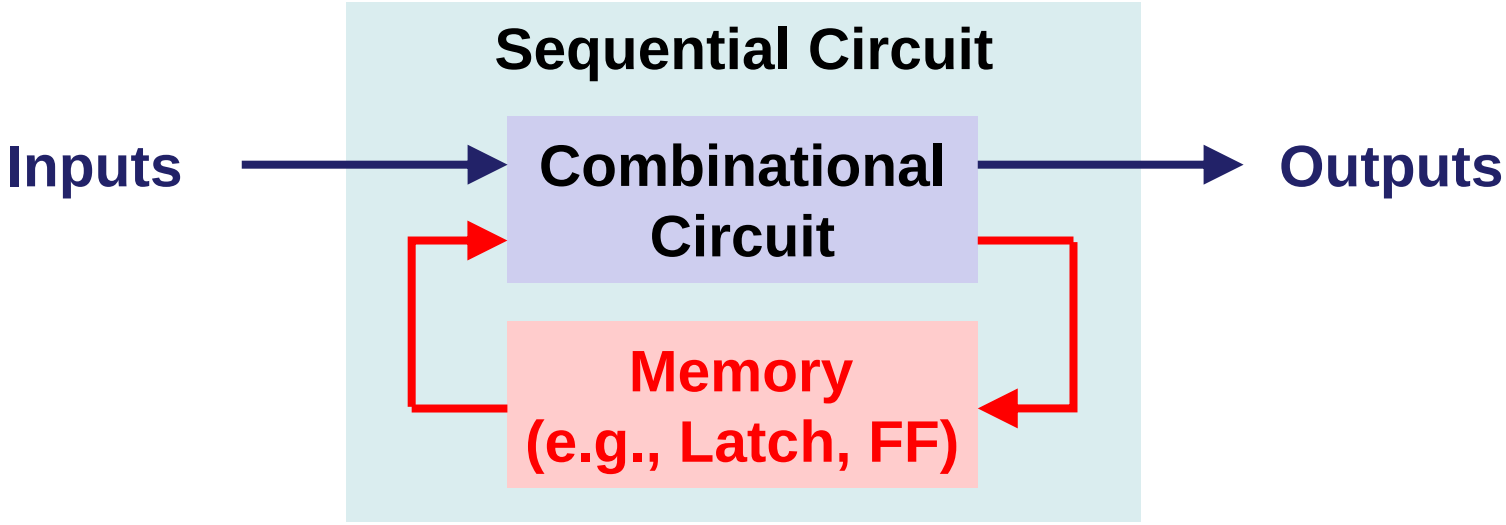


in1	enable	out1
0	0	Z
1	0	Z
0	1	0
1	1	1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity tri_ex is
port (in1, enable: in std_logic;
                out1: out std_logic);
end tri_ex;
architecture tri_ex_arch of tri_ex is
begin
    out1 <= in1 when enable = '1' else 'Z';
end tri_ex_arch;
```

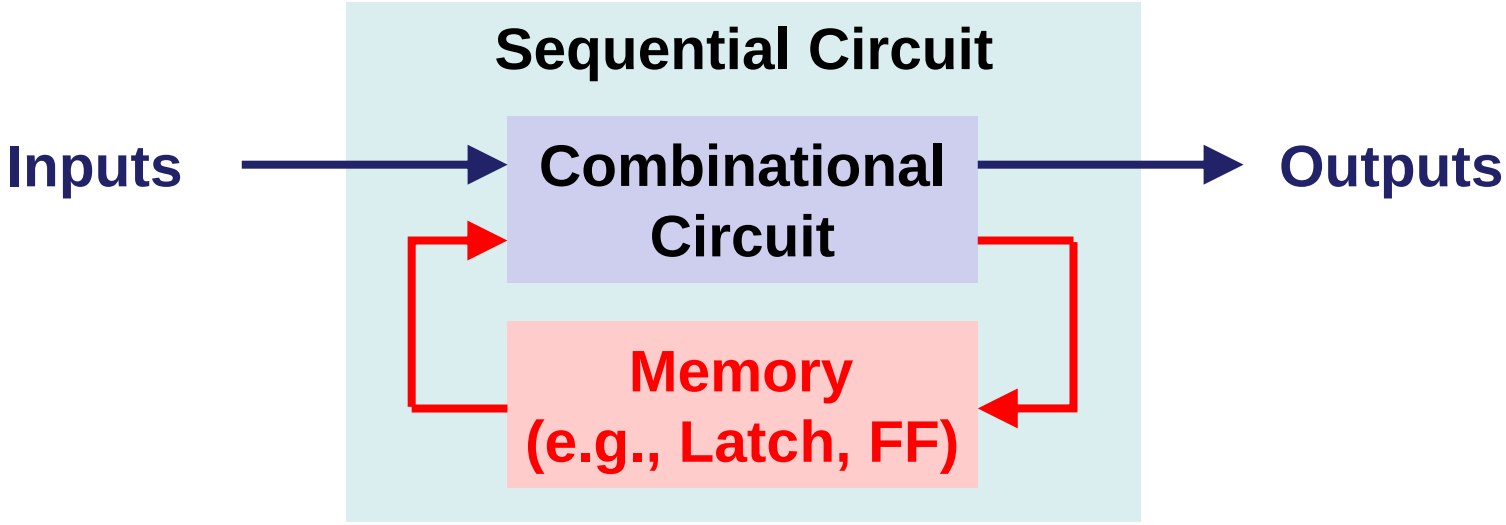
Outline

- **Combinational Circuit: no memory**
  - Outputs depend on the *present* inputs only.
  - **Rule**: Use **either** concurrent **or** sequential statements.
- **Sequential Circuit**: **has memory**
  - Outputs depend on *present* inputs and *previous* outputs.
  - **Rule**: **MUST** use sequential statements (i.e., `process`) .



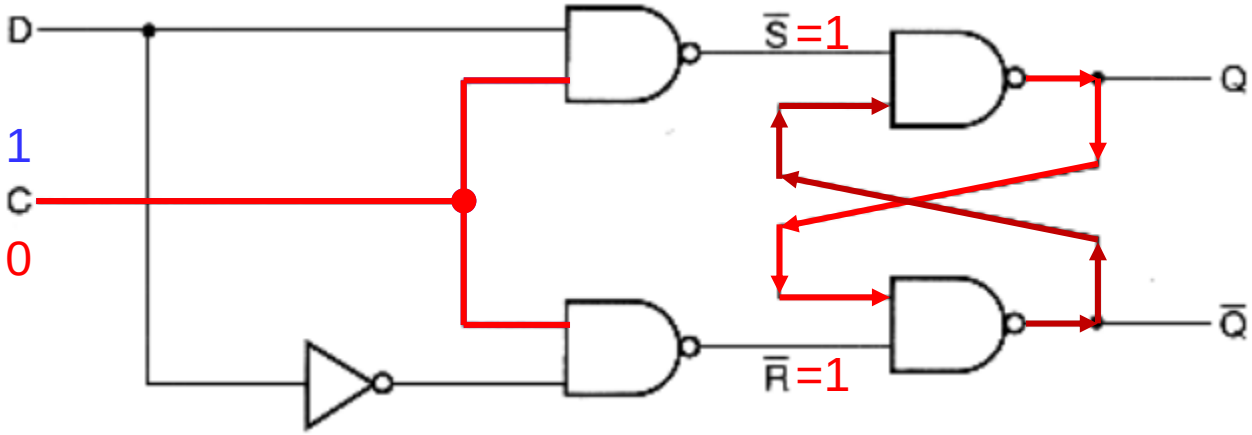
Sequential Circuit

- **Sequential Circuit**: **has memory**
  - Outputs depend on *present* inputs and *previous* outputs.
    - The previous output(s) are kept in the memory device(s) and treated as the present state.
    - Two most common memory devices: Latch and Flip-flop (FF), both can hold one bit of data (i.e., either low or high).
  - **Rule**: You **MUST** build a sequential circuit with **only** sequential statements (i.e., statements inside `process`).



Memory Device: Latch

- **Latch** has **no** CLOCK signal.
  - It changes the output in accordance with the **enable signal**.
- **Case Study**: **D Latch**
  - When enable signal C is high, the output Q follows input D.
    - That is why D latch is also called as transparent latch: When enable line C is asserted, the latch is said to be transparent.
  - When C falls, the last state of D is held (i.e., has memory)!

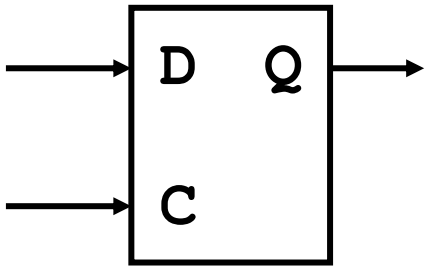


C	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

https://www.edgefx.in/digital-electronics-latches-and-flip-flops/

D Latch in VHDL

```
 1 library IEEE;--(ok vivado 2014.4)
 2 use IEEE.STD_LOGIC_1164.ALL;
 3 entity latch_ex is
 4 port (C, D: in std_logic;
 5           Q: out std_logic);
 6 end latch_ex;
 7 architecture latch_ex_arch of latch_ex is
 8 begin
 9   process(C, D) -- sensitivity list
10   begin
11     if ( C = '1' ) then
12       Q <= D;
13     end if;
       -- no change (memory)
14   end process;
15 end latch_ex_arch;
```

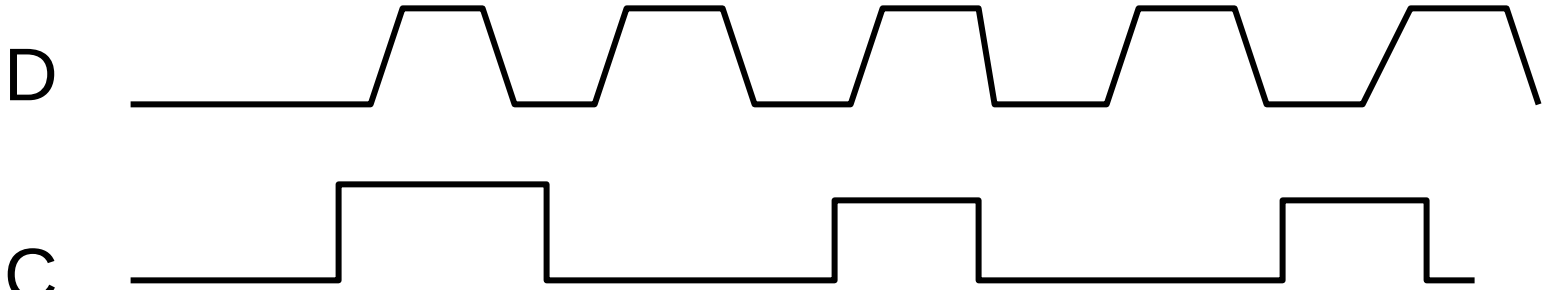


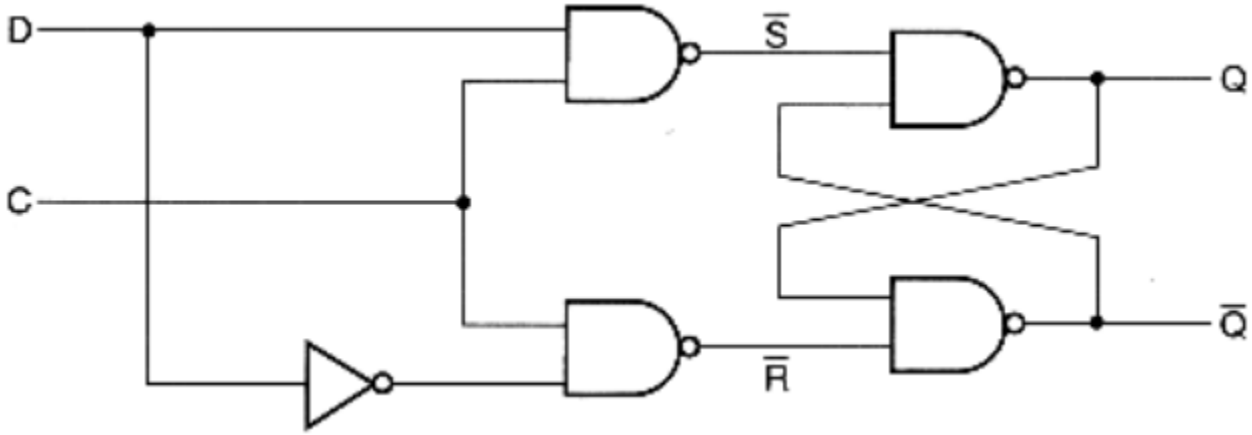
C	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

https://www.edgefx.in/digital-electronics-latches-and-flip-flops/

Class Exercise 3.3

- Given a D latch, draw Q in the following figure:

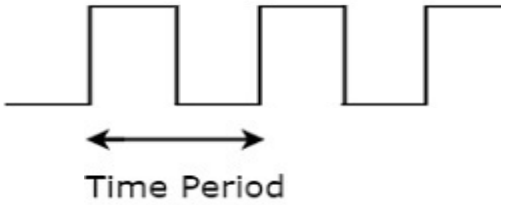




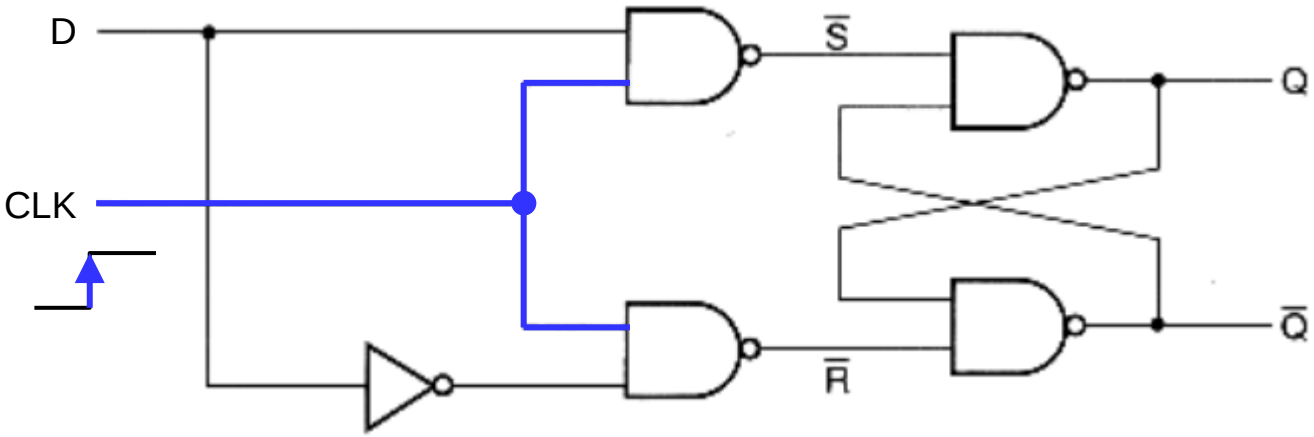
C	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

Memory Device: Flip-flop (FF)

- Flip-flop **has** a clock signal (i.e., **CLK**).
  - It changes the output at clock edges.
    - **Positive-Edge-Triggered**: At every **low to high** of CLOCK.
    - **Negative-Edge-Triggered**: At every **high to low** of CLOCK.

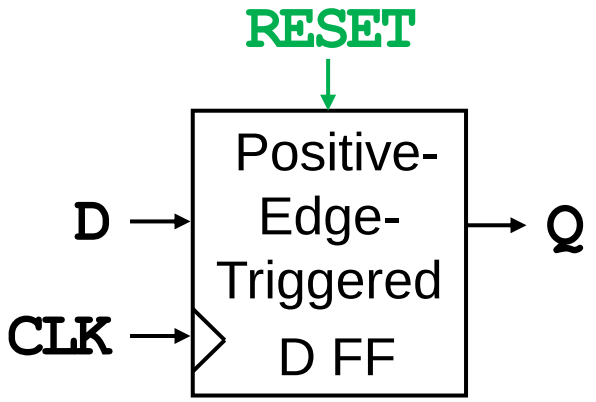


- **Case Study**: **Positive-Edge-Triggered D Flip Flop**
  - Whenever the clock rises, the output Q follows input D.
  - Otherwise, the last state of D is held (i.e., has memory)!
  - The held value can be **reset** asynchronously (i.e., anytime) or synchronously (i.e., only at clock edges).



CLK	D	Q_{next}
rising	0	0
	1	1
non-rising	X	Q

D Flip-flop with Async. Reset in VHDL



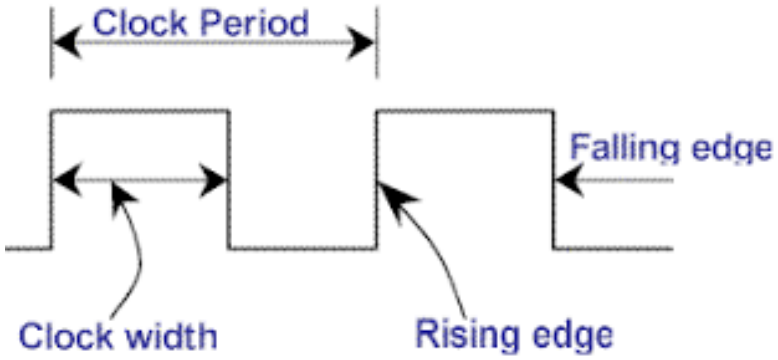

```
 1 library IEEE;
 2 use IEEE.STD_LOGIC_1164.ALL;
 3 entity DFF_ASYNC is
 4 port(D,CLK,RESET: in std_logic;
 5                   Q: out std_logic);
 6 end dff_async;
 7 architecture DFF_ASYNC_ARCH of DFF_ASYNC is
 8 begin
 9   process(CLK, RESET) -- sensitivity list
10   begin
11     if (RESET = '1') then
12       Q <= '0'; -- Reset Q anytime
13     elsif CLK = '1' and CLK'event then
14       Q <= D; -- Q follows input D
15     end if;
       -- no change (so has memory)
16   end process;
17 end DFF_ASYNC_ARCH;
```

← Positive-edge-triggered

Attribute

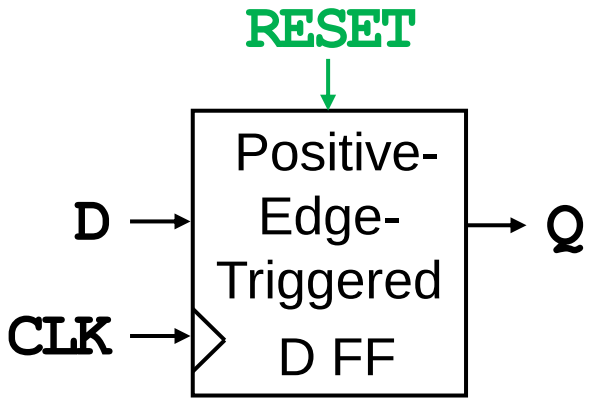
- An attribute provides information about items such as entities, architecture, signals, and types.
  - The syntax is an apostrophe (`'`) and the attribute name.
  - There are several useful predefined value, signal, and range attributes (see VHDL Predefined Attributes).

- An important signal attribute is the **`'event`**.
  - This attribute yields a Boolean value of TRUE if an event has just occurred on the signal.
  - It is used primarily to determine if a **clock** has transitioned (i.e., either low-to-high or high-to-low).




```
CLK = '1' and CLK'event
```

D Flip-flop with Sync. Reset in VHDL




```
 1 library IEEE;
 2 use IEEE.STD_LOGIC_1164.ALL;
 3 entity DFF_SYNC is
 4 port(D,CLK,RESET: in std_logic;
 5                   Q: out std_logic);
 6 end DFF_SYNC;
 7 architecture DFF_SYNC_ARCH of DFF_SYNC is begin
 8   process(CLK)
 9   begin
10     if CLK = '1' and CLK'event then
11       if (RESET = '1') then
12         Q <= '0'; -- Reset Q at edges
13       else
14         Q <= D; -- Q follows input D
15       end if;
16     end if;
       -- no change (so has memory)
17   end process;
18 end DFF_SYNC_ARCH;
```

← *Shall we put* **RESET** *in the sensitivity list?*

← **Positive-edge-triggered**

Async. Reset vs. Sync. Reset

- The order of the statements inside the process determines asynchronous reset or synchronous reset.
  - **Asynchronous Reset** (check **RESET** first!)

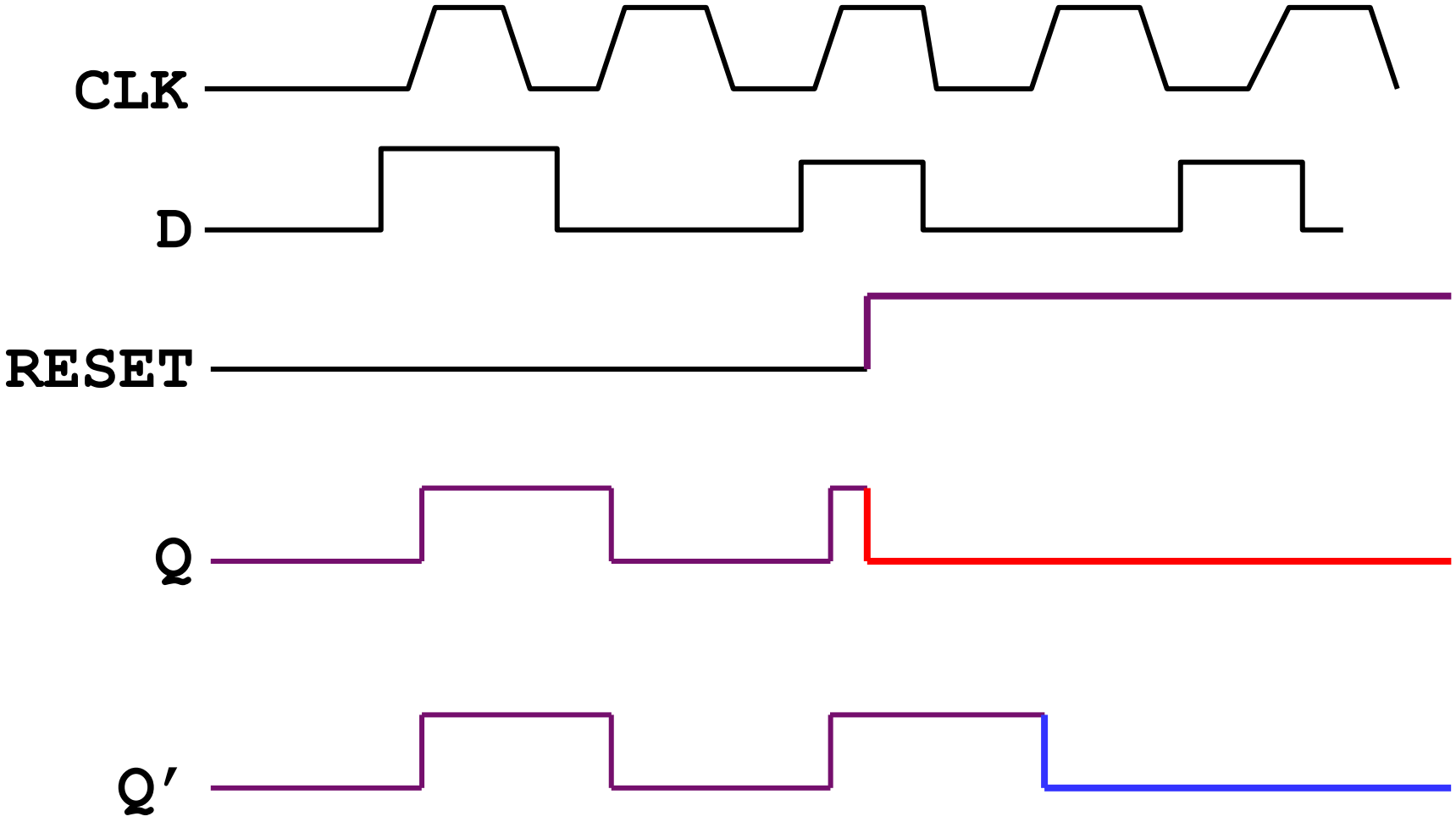
```
11      if (RESET = '1') then
12         Q <= '0'; -- Reset Q anytime
13      elsif CLK = '1' and CLK'event then
14         Q <= D; -- Q follows input D
15      end if;
```

  - **Synchronous Reset** (check **CLK** first!)

```
10     if CLK = '1' and CLK'event then
11       if (RESET = '1') then
12         Q <= '0'; -- Reset Q at edges
13       else
14         Q <= D; -- Q follows input D
15       end if;
16     end if;
```

Class Exercise 3.4

- Given a pos-edge-triggered D FF, determine whether Q and Q' are based on async. reset or sync. reset:



Latch vs. Flip-flop (FF)

- Latch and Flip-flop (FF) are both typical memory devices which can store one bit of data.

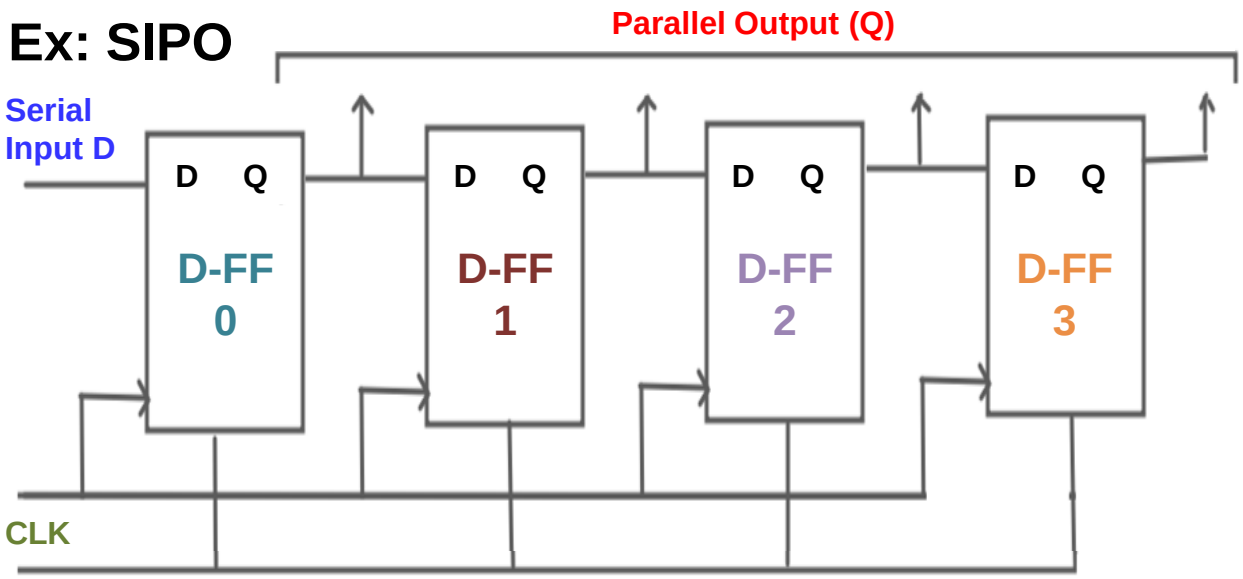
- Latch has **no** clock signal and is level-triggered.
  - The output of a latch can be changed only when the level of enable signal C is high.

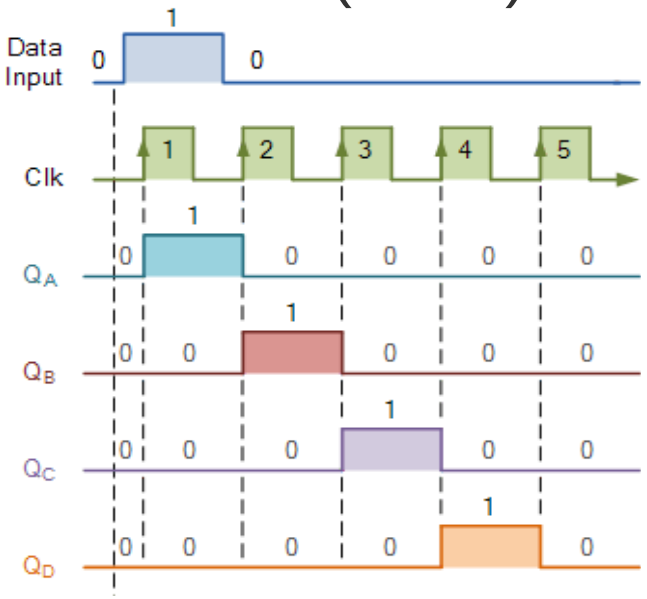
- FF **has** a clock signal and is edge-triggered.
  - The output of an FF can only be changed whenever the clock signal goes from low to high and high to low (i.e., clock edges).
  - FF can be found in a wide range of sequential circuits where time plays an essential role.

Seq. Circuit Example: Shift Register

- A **register** is a device that can be composed of a group of FFs to store multiple bits of data.
- A **shift register** allows the stored data being moved from one FF to another.
  - There are basically **four** types: ① Serial-In-Serial-Out (SISO), ② Serial-In-Parallel-Out (SIPO), ③ Parallel-In-Serial-Out (PISO), and ④ Parallel-In-Parallel-Out (PIPO).

Ex: SIPO





https://www.geeksforgeeks.org/shift-registers-in-digital-logic/ & https://www.electronics-tutorials.ws/sequential/seq_5.html

SIPO Shift Register in VHDL

```
entity SIPO_ASYNC is
  port (D, CLK, RST : IN STD_LOGIC;
        Q : OUT STD_LOGIC_VECTOR(3 downto 0) );
end SIPO_ASYNC;
architecture SIPO_ASYNC_ARCH of SIPO_ASYNC is
component DFF_ASYNC is
  port(D, clk, reset : in STD_LOGIC;
       Q : out STD_LOGIC );
end component;
signal dout : STD_LOGIC_VECTOR(3 downto 0);
begin
DFF0: DFF_ASYNC port map(D, CLK, RST, dout(0));
DFF1: DFF_ASYNC port map(dout(0), CLK, RST, dout(1));
DFF2: DFF_ASYNC port map(dout(1), CLK, RST, dout(2));
DFF3: DFF_ASYNC port map(dout(2), CLK, RST, dout(3));
Q <= dout;
end SIPO_ASYNC_ARCH;
```

← *necessary?*

Recall: Modes of I/O Pins

- Modes of I/O pin must be explicitly specified in **port** of entity declaration:

  Example:

```
entity eqcomp4 is
  port (a, b: in std_logic_vector(3 downto 0);
        equals: out std_logic);
end eqcomp4;
```

- There are 4 available modes of I/O pins:
  1) **in**: Data flows **in** only
  2) **out**: Data flows **out** only (cannot be read back by the entity)
  3) **inout**: Data flows **bi-directionally** (i.e., in or out)
  4) **buffer**: Similar to **out** but it can be **read back** by the entity

Summary

- **Combinational Circuit**: **no memory**
  - Outputs depend on the *present* inputs only.
  - **Rule**: Use **either** concurrent **or** sequential statements.
- **Sequential Circuit**: **has memory**
  - Outputs depend on *present* inputs and *previous* outputs.
  - **Rule**: **MUST** use sequential statements (i.e., `process`) .