

Cours de Systèmes numériques

Ch. 3 : Fonctions combinatoires & bascules

Nicolas.Schroeter@hefr.ch

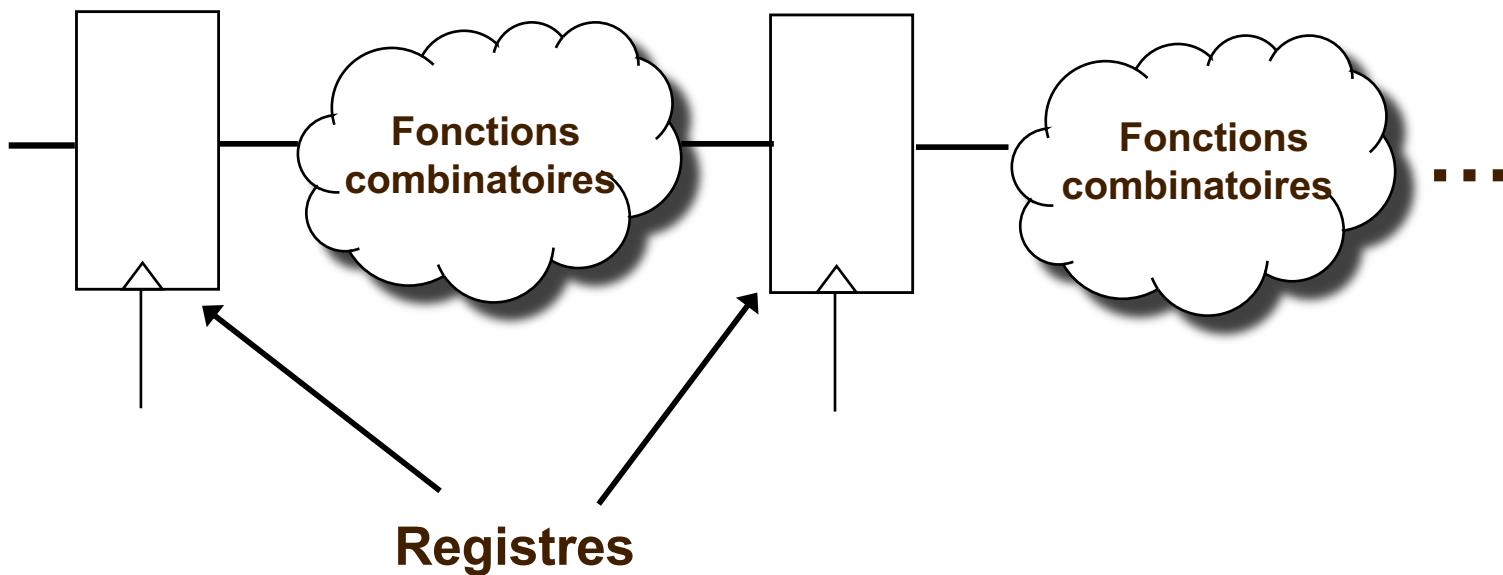


Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

SN-1 : Combinatoire & bascules 1

Register Transfer Level (RTL) Design

- ❑ Pour la conception de systèmes numériques synchrones, RTL est largement utilisé.



- ❑ Chaque instruction concurrente VHDL représente soit un bloc combinatoire, soit un bloc séquentiel (registres).



Fonctions combinatoires

Exemple de fonction combinatoires:

- ❑ Equation logique
- ❑ Table de vérité
- ❑ Décodage X/Y
- ❑ Multiplexage
- ❑ Comparaison
- ❑ Encodage de priorité
- ❑ Opérations arithmétiques
 - Addition/Soustraction



Instructions VHDL concurrentes pour les fonctions combinatoires

❑ Affectation:

- $Y \leq A \text{ and } C;$ -- équation logique sous forme algébrique

❑ Affectation avec condition:

- $Y \leq \dots \text{ when } \dots \text{ else } \dots;$

❑ Affectation de sélection:

- With ... select

❑ process



Processus concurrents implicites

- ❑ Les instructions concurrentes d'affectation simple, conditionnelle et de sélection sont implicitement des processus.
- ❑ Par exemple:

```
-- une affectation simple, hors  
-- d'une declaration de process  
A <= B NAND C ;
```

revient au même que:

```
process (B,C) begin  
    A <= B NAND C ;  
end process;
```



Processus concurrents implicites

Ou encore:

```
A <= '1' when B='1' else
  | '0' when C='1' else '0';
-- ATTENTION : l'affection conditionnelle avec when else
-- ne peut etre utilisee qu'a l'exterieur d'une declaration
-- de process (instruction strictement concurrente)
```

Revient au même que:

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

-- ATTENTION :
-- la structure conditionnelle
-- if then else ne peut etre
-- utilisee qu'a l'interieur
-- d'une declaration de process
-- (strictement sequentielle)



Processus concurrents implicites

Ou encore:

```
with signal_sel select
    Sortie <= Entr_A when "00" | "01",
        '1' when "11",
        '0' when others; -- pour couvrir toutes les autres
                            -- combinaisons "UZ", ...
```

Revient au même que:

```
process(signal_sel, ENTR_A)
begin
    if signal_sel = "00" or signal_sel = "01" then
        Sortie <= Entr_A;
    elsif signal_sel = "11" then
        Sortie <= '1';
    else
        Sortie <= '0';
    end if;
end process;
```



Modélisation d'une table de vérité

Table de vérité du convertisseur

Hexa → ASCII:

Hexa	ascii
0000	0110000
0001	0110001
0010	0110010
0011	0110011
0100	0110100
0101	0110101
0110	0110110
0111	0110111
1000	0111000
1001	0111001
1010	1000001
1011	1000010
1100	1000011
1101	1000100
1110	1000101
1110	1000110



Modélisation d'une table de vérité

Code VHDL:

```
with hexa select
  ascii <= "0110000" when "0000",
              "0110001" when "0001",
              "0110010" when "0010",
              "0110011" when "0011",
              "0110100" when "0100",
              "0110101" when "0101",
              "0110110" when "0110",
              "0110111" when "0111",
              "0111000" when "1000",
              "0111001" when "1001",
              "1000001" when "1010",
              "1000010" when "1011",
              "1000011" when "1100",
              "1000100" when "1101",
              "1000101" when "1110",
              "1000110" when others;
```



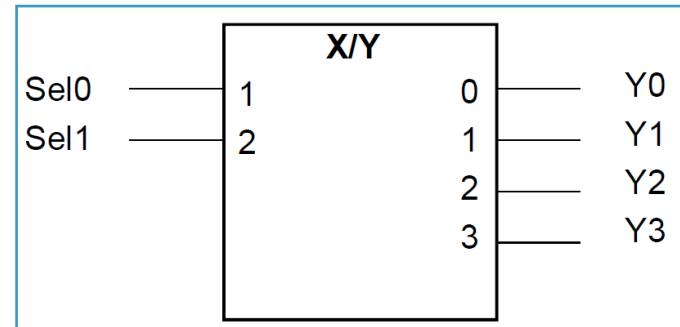
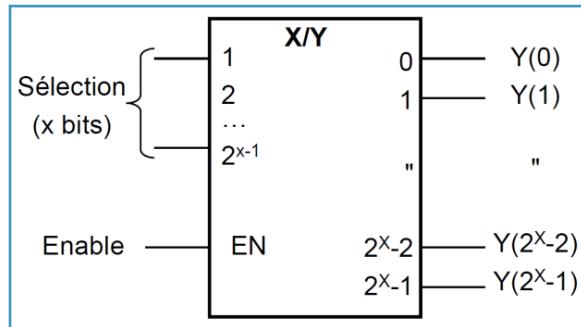
Décodeur X/Y

- ❑ But: activer la sortie dont on donne l'indice sous forme binaire (entier non signé)
 - Décode la valeur binaire donnée en entrée sur n-bits
 - Génère tous les produits de l'entrée n-bits
- ❑ Une seule et unique sortie est active simultanément
- ❑ Comporte souvent une entrée d'activation (Enable), indispensable pour étendre le décodage

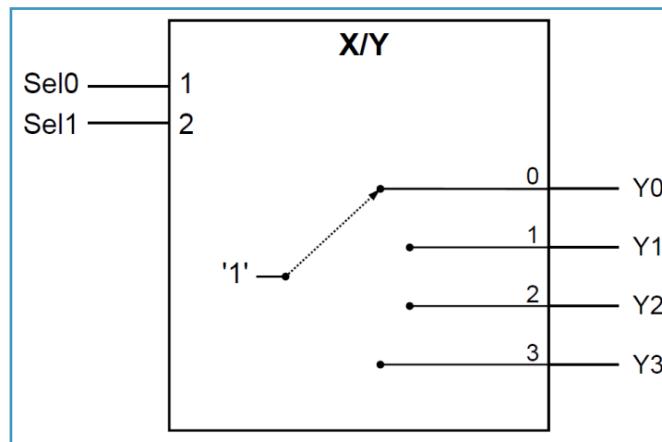


Décodeur X/Y

❑ Symbole du décodeur:



❑ Principe de fonctionnement:

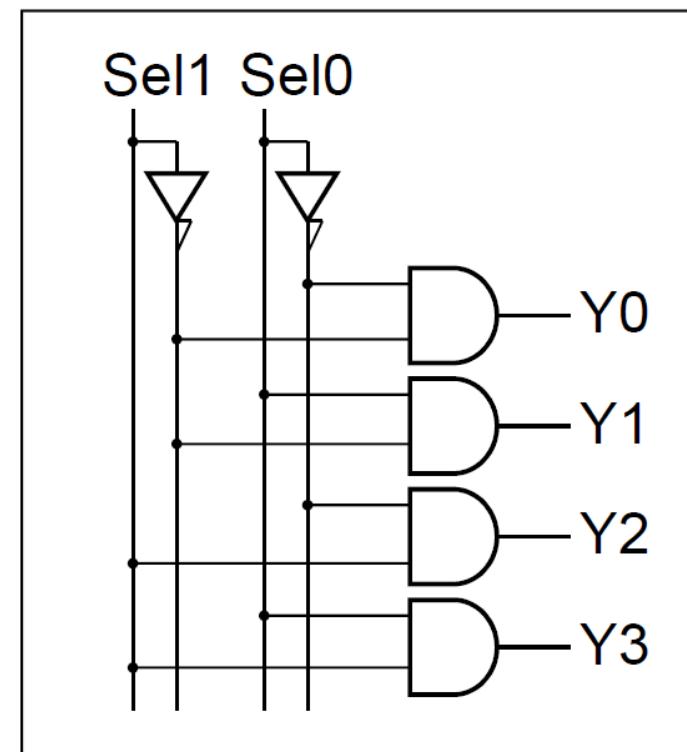


Décodeur 2 entrées et 4 sorties

- Génère tous les produits de Sel1 et Sel0

Table de vérité					
Sel1	Sel0	Y3	Y2	Y1	Y0
'0'	'0'	'0'	'0'	'0'	'1'
'0'	'1'	'0'	'0'	'1'	'0'
'1'	'0'	'0'	'1'	'0'	'0'
'1'	'1'	'1'	'0'	'0'	'0'

$$\begin{array}{ll} Y_0 = (\overline{\text{Sel1}} \cdot \overline{\text{Sel0}}) & Y_1 = (\overline{\text{Sel1}} \cdot \text{Sel0}) \\ Y_2 = (\text{Sel1} \cdot \overline{\text{Sel0}}) & Y_3 = (\text{Sel1} \cdot \text{Sel0}) \end{array}$$



Décodeur 2 à 4 avec Enable

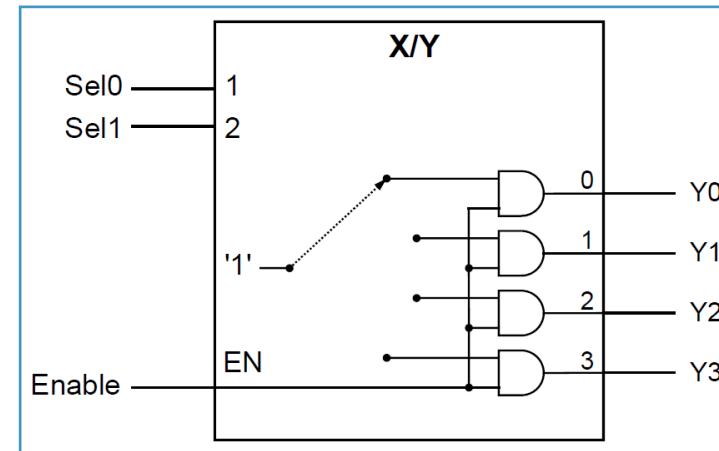
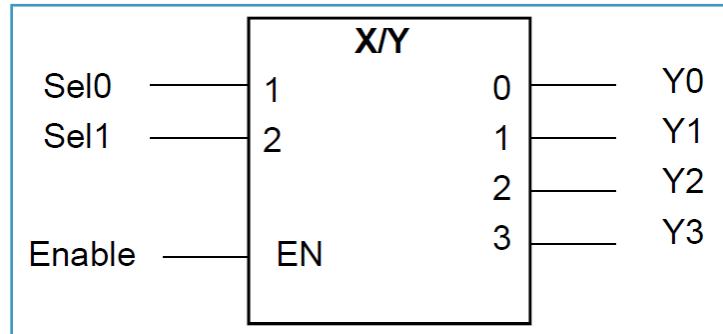
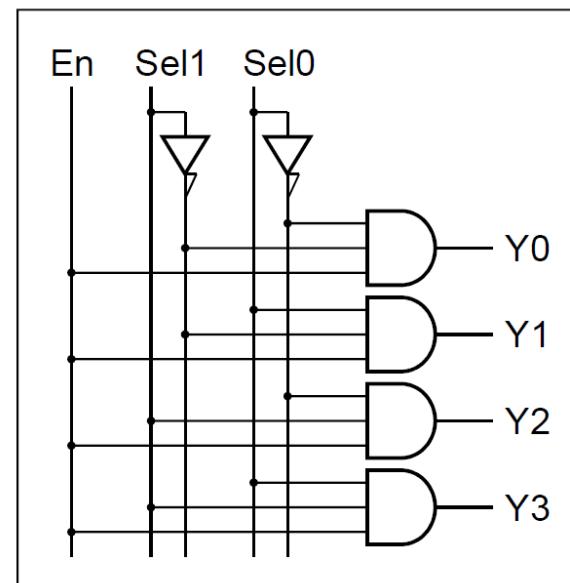


Table de vérité

En	Sel1	Sel0	Y3	Y2	Y1	Y0
'0'	x	x	'0'	'0'	'0'	'0'
'1'	'0'	'0'	'0'	'0'	'0'	'1'
'1'	'0'	'1'	'0'	'0'	'1'	'0'
'1'	'1'	'0'	'0'	'1'	'0'	'0'
'1'	'1'	'1'	'1'	'0'	'0'	'0'

$$\begin{aligned}
 Y_0 &= \text{En} \cdot (\overline{\text{Sel1}} \cdot \overline{\text{Sel0}}) & Y_1 &= \text{En} \cdot (\overline{\text{Sel1}} \cdot \text{Sel0}) \\
 Y_2 &= \text{En} \cdot (\text{Sel1} \cdot \overline{\text{Sel0}}) & Y_3 &= \text{En} \cdot (\text{Sel1} \cdot \text{Sel0})
 \end{aligned}$$



Décodeur 2 à 4

Expressions des équations VS du comportement

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

entity dec_2a4_flot is
    port(
        Sel_i : in std_logic_vector(1 downto 0);
        en_i: in std_logic;
        y_o : out std_logic_vector(3 downto 0)
    );
end entity;

architecture flot of dec_2a4_flot is
    signal y_s : std_logic_vector(3 downto 0);
begin
    --equations logiques
    y_s(0) <= not Sel_i(1) and not Sel_i(0); --prod. 0
    y_s(1) <= not Sel_i(1) and Sel_i(0); --prod. 1
    y_s(2) <= Sel_i(1) and not Sel_i(0); --prod. 2
    y_s(3) <= Sel_i(1) and Sel_i(0); --prod. 3

    y_o <= Y_s when En_i = '1' else "0000";
end architecture;
```

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

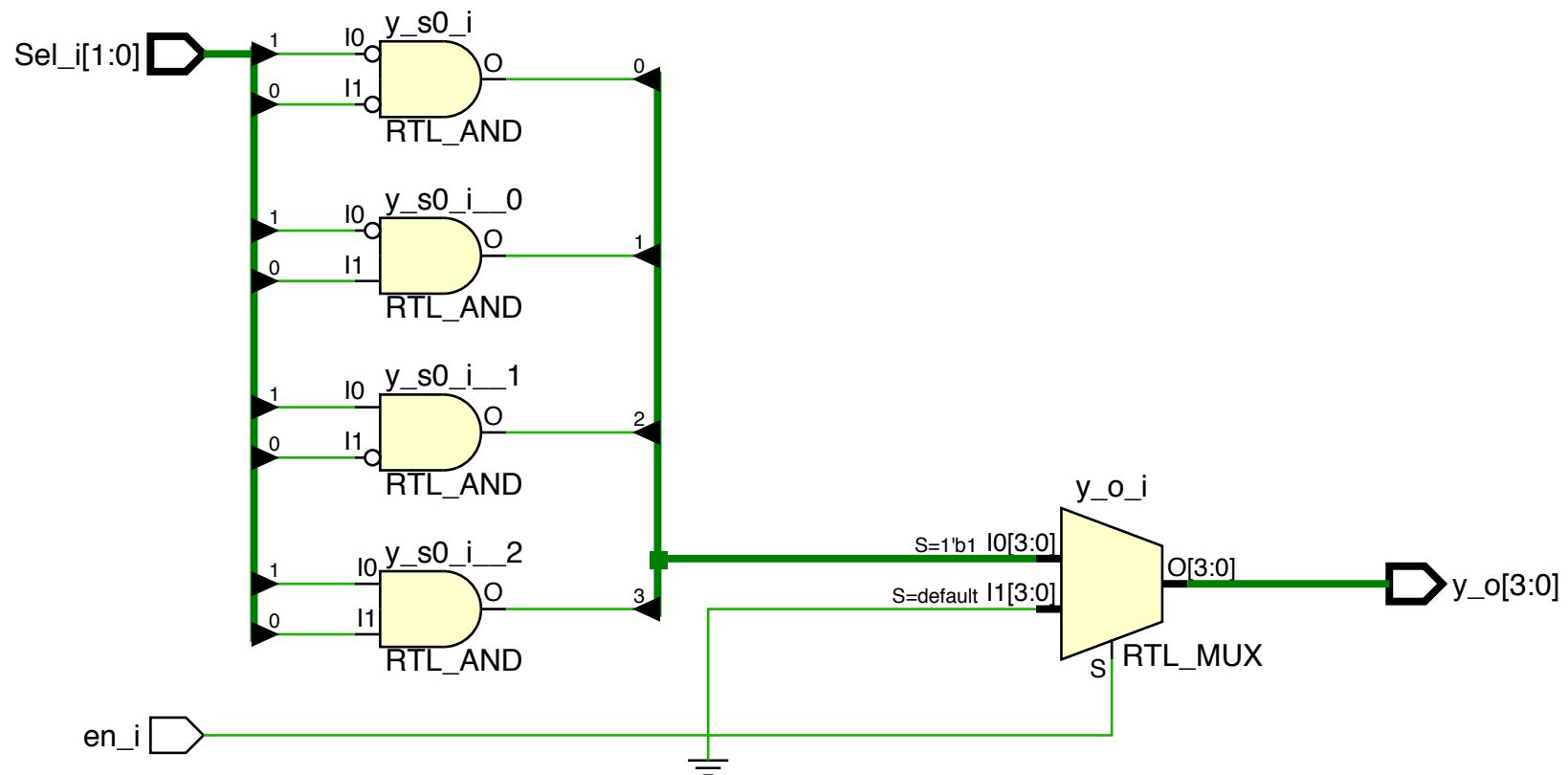
entity dec_2a4_beh is
    port(
        Sel_i : in std_logic_vector(1 downto 0);
        en_i: in std_logic;
        y_o : out std_logic_vector(3 downto 0)
    );
end entity;

architecture behavioral of dec_2a4_beh is
    signal y_s : std_logic_vector(3 downto 0);
begin
    with Sel_i select
        y_s <= "0001" when "00",
                    "0010" when "01",
                    "0100" when "10",
                    "1000" when "11",
                    "XXXX" when others; --simulation

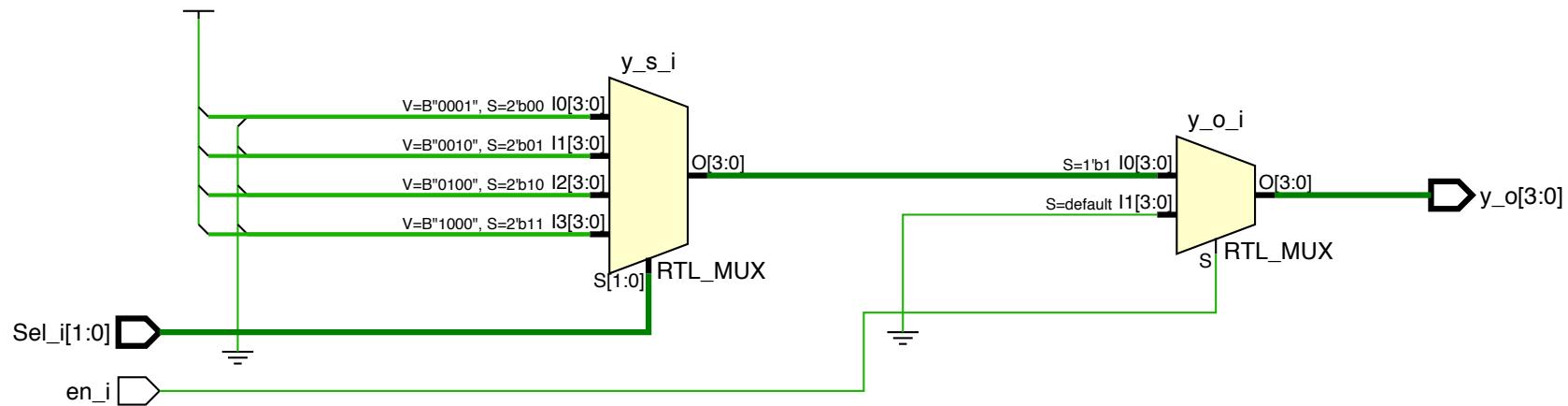
    y_o <= Y_s when En_i = '1' else "0000";
end architecture;
```



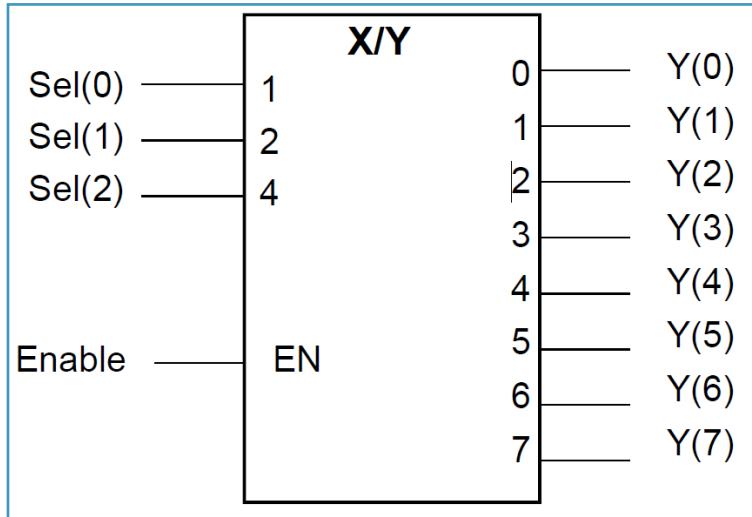
Décodeur flot de données, schéma RTL



Décodeur comportemental, schéma RTL



Décodeur 3 à 8

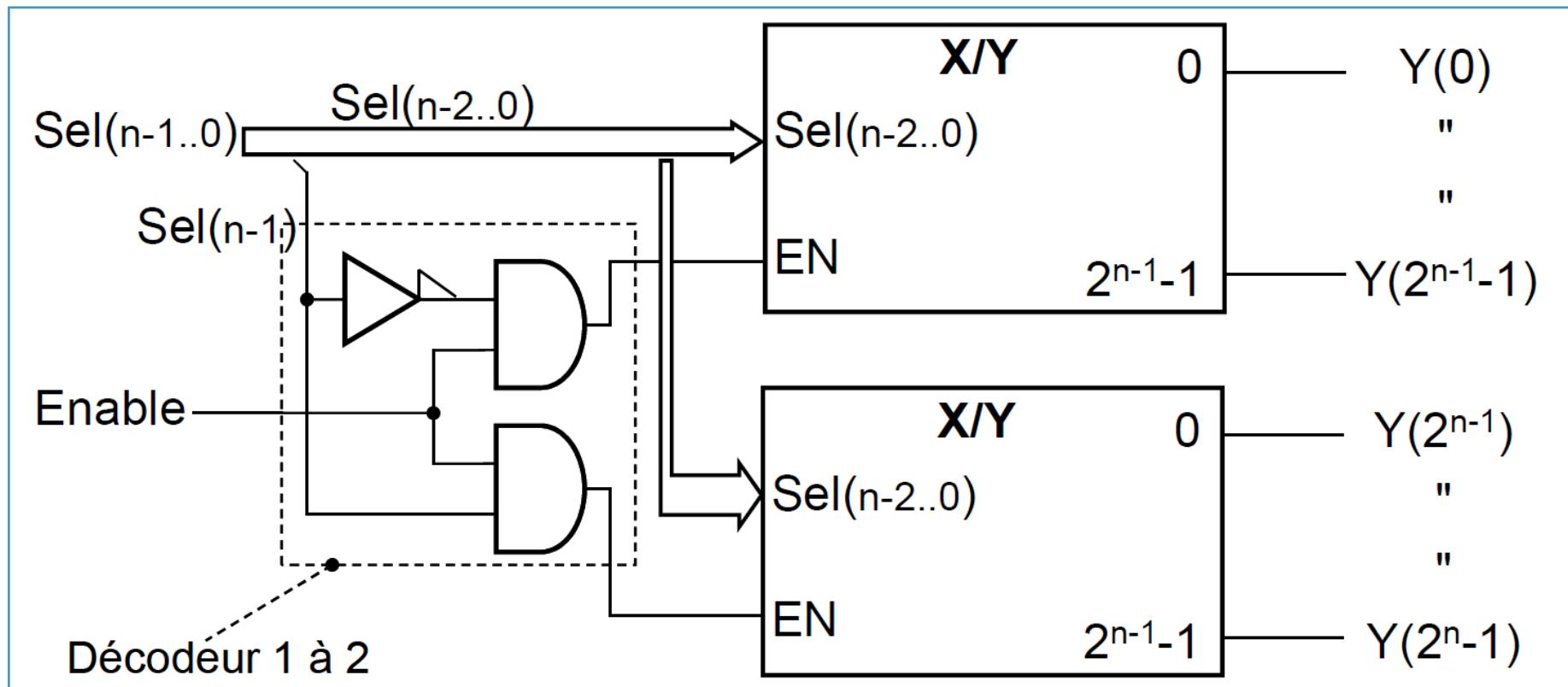


EN	Sel(2)	Sel(1)	Sel(0)	Y(7)	Y(6)	Y(5)	Y(4)	Y(3)	Y(2)	Y(1)	Y(0)
'0'	x	x	x	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'1'
'1'	'0'	'0'	'1'	'0'	'0'	'0'	'0'	'0'	'0'	'1'	'0'
'1'	'0'	'1'	'0'	'0'	'0'	'0'	'0'	'0'	'1'	'0'	'0'
'1'	'0'	'1'	'1'	'0'	'0'	'0'	'0'	'1'	'0'	'0'	'0'
'1'	'1'	'0'	'0'	'0'	'0'	'0'	'1'	'0'	'0'	'0'	'0'
'1'	'1'	'0'	'1'	'0'	'0'	'1'	'0'	'0'	'0'	'0'	'0'
'1'	'1'	'1'	'0'	'0'	'1'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'1'	'1'	'1'	'1'	'0'	'0'	'0'	'0'	'0'	'0'	'0'



Décodeur 3 à 8 décomposé

Un décodeur n bits peut se réaliser au moyen de deux décodeurs $n-1$ bits:

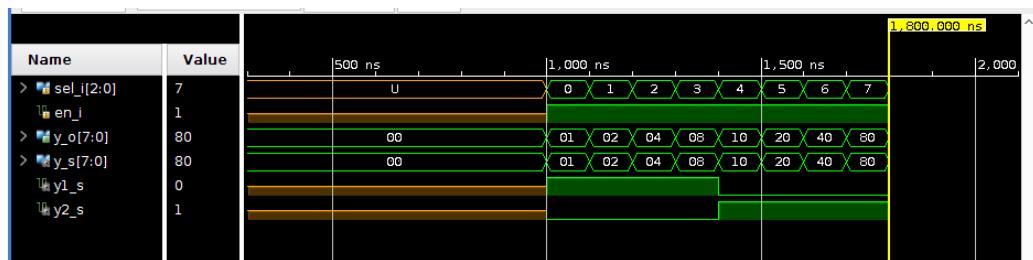
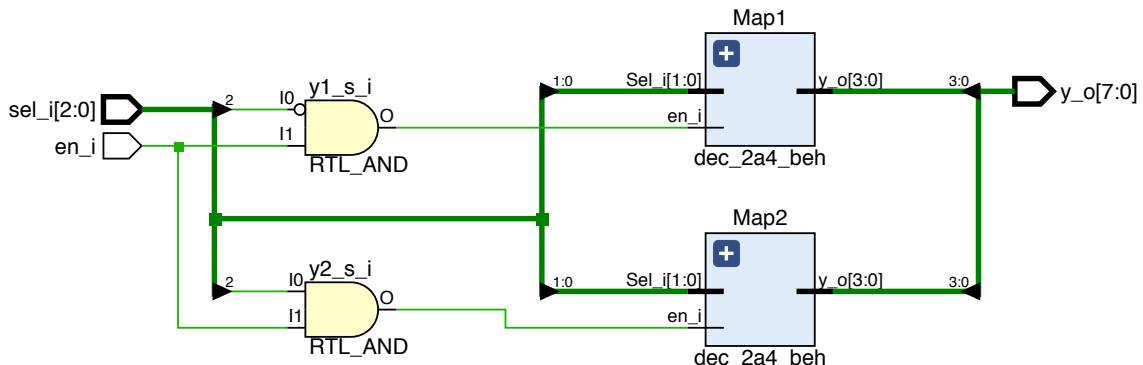


Exercice 1: Décodeur

- ❑ Réaliser le composant VHDL d'un décodeur 3 à 8 en combinant:
 - 2 décodeurs 2 à 4
 - 1 décodeur 1 à 2
- ❑ Simuler pour vérifier le bon fonctionnement du décodeur
- ❑ Produire le schéma RTL



Solution 1:Décodeur 3 à 8



```

library ieee;
use ieee.std_logic_1164.all;
entity dec_3a8 is
    port(
        sel_i: in std_logic_vector(2 downto 0);
        en_i : in std_logic;
        y_o : out std_logic_vector(7 downto 0)
    );
end entity;
architecture behav of dec_3a8 is
    component dec_2a4_beh is
        port(
            Sel_i : in std_logic_vector(1 downto 0);
            en_i: in std_logic;
            y_o : out std_logic_vector(3 downto 0)
        );
    end component;
    signal y_s : std_logic_vector(7 downto 0);
    signal y1_s, y2_s : std_logic;
begin
    Map1:dec_2a4_beh
    port map(
        sel_i => sel_i(1 downto 0),
        en_i => y1_s,
        y_o => y_s(3 downto 0)
    );
    Map2:dec_2a4_beh
    port map(
        sel_i => sel_i(1 downto 0),
        en_i => y2_s,
        y_o => y_s(7 downto 4)
    );
    y1_s <= not(sel_i(2)) and en_i;
    y2_s <= sel_i(2) and en_i;
    y_o <= y_s ;
end architecture;

```



Décodage: génération de minterme

- ❑ Chaque sortie d'un décodeur n'est active que pour une seul et unique valeur du code binaire en entrée
 - Chaque sortie correspond à un minterme
- ❑ Avec un décodeur à N entrées et une porte OU à $2^{**}N-1$ entrées au maximum, on peut implémenter n'importe quelle fonction combinatoire à N entrées, sous la forme d'une somme de mintermes.



Décodeur en générateur de fonction

- Table de vérité et équation de la fonction F

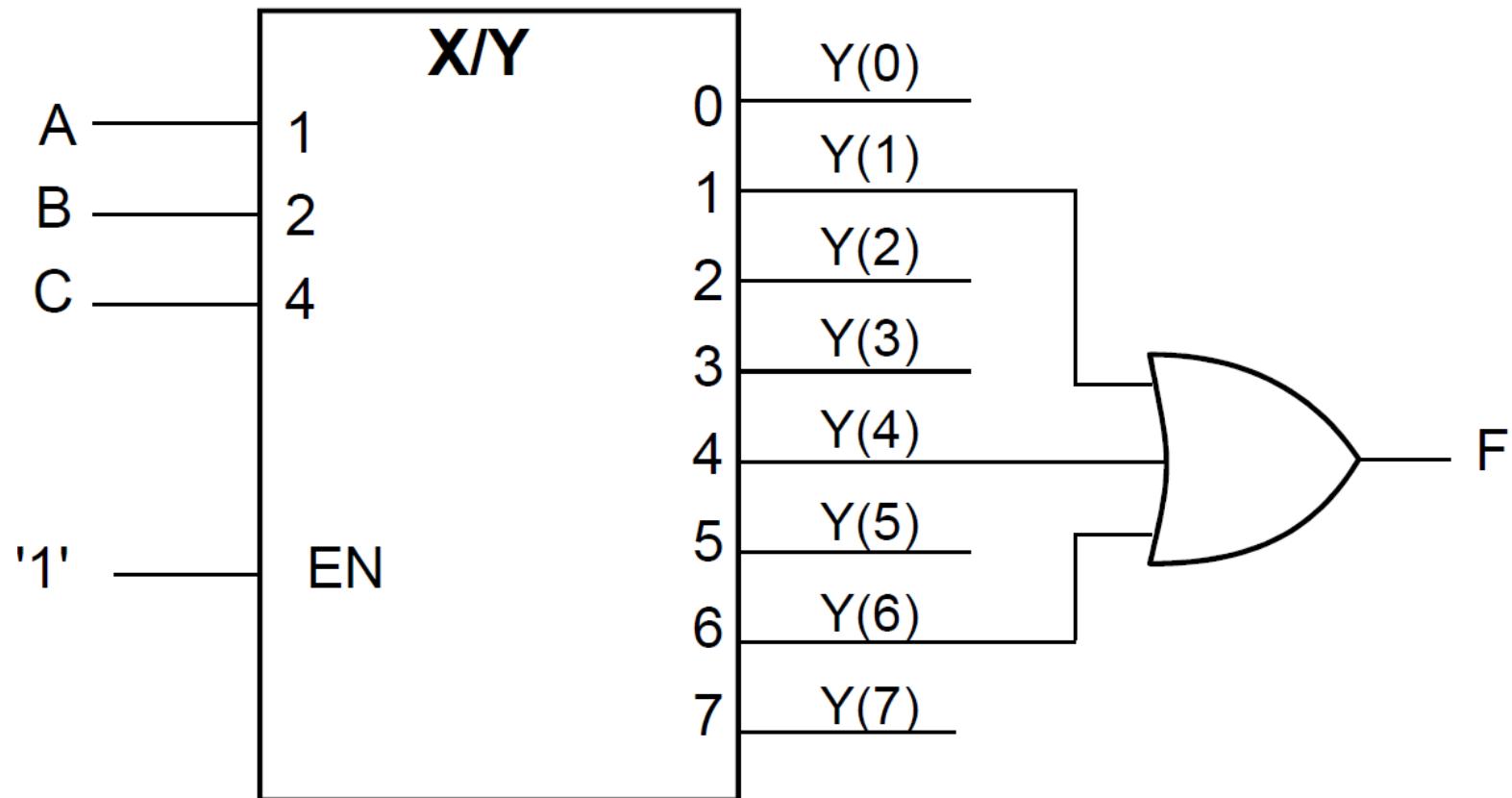
C	B	A	F
'0'	'0'	'0'	'0'
'0'	'0'	'1'	'1'
'0'	'1'	'0'	'0'
'0'	'1'	'1'	'0'
'1'	'0'	'0'	'1'
'1'	'0'	'1'	'0'
'1'	'1'	'0'	'1'
'1'	'1'	'1'	'0'

$$F(C, B, A) = \sum 1, 4, 6$$



Décodeur en générateur de fonction

$$F(C, B, A) = \sum 1, 4, 6$$



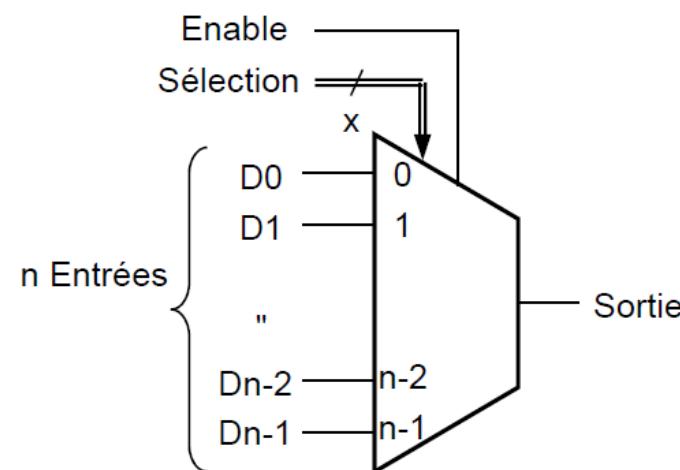
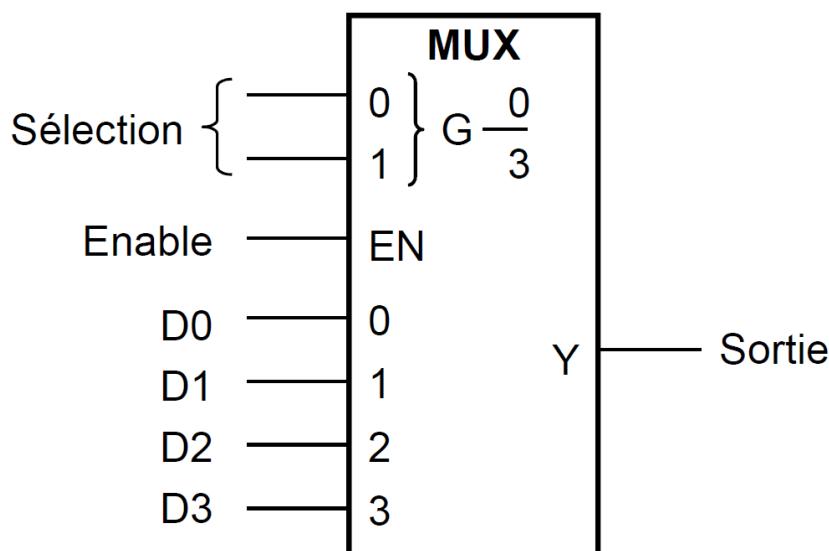
Multiplexeur

- ❑ But: Transmettre sur la sortie l'entrée sélectionnée par son indice
- ❑ Une entrée supplémentaire «Enable» est souvent présente pour faciliter l'extension.
- ❑ Les multiplexeurs sont utilisables à la place de portes logiques pour réaliser des fonctions combinatoires quelconques.
- ❑ L'utilisation judicieuse des multiplexeurs permet d'économiser du matériel.



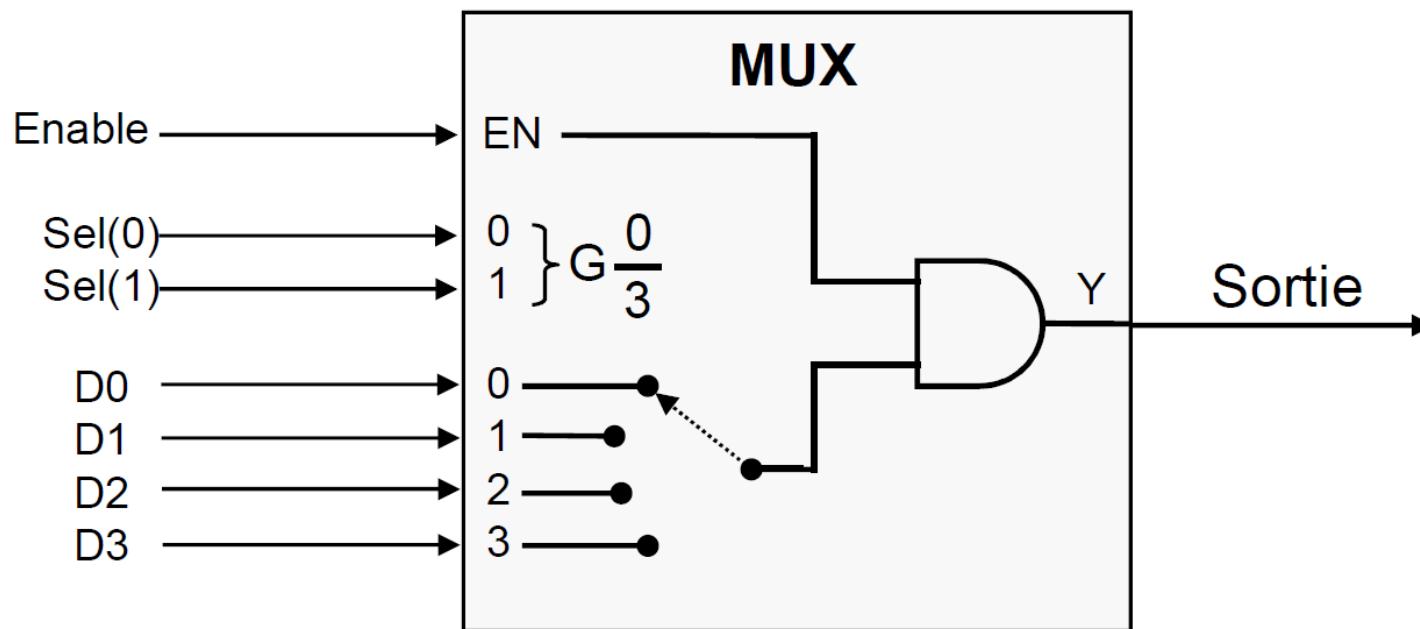
Multiplexeur

- ❑ Un multiplexeur à N to 1 est un commutateur à N entrées et une sortie, avec $N = 2^x$ (x nombre de bits de sélection)



Multiplexeur 4 à 1

- Il s'agit d'un sélecteur à :
 - 2 lignes de sélection Sel(1..0), choix de l'entrée



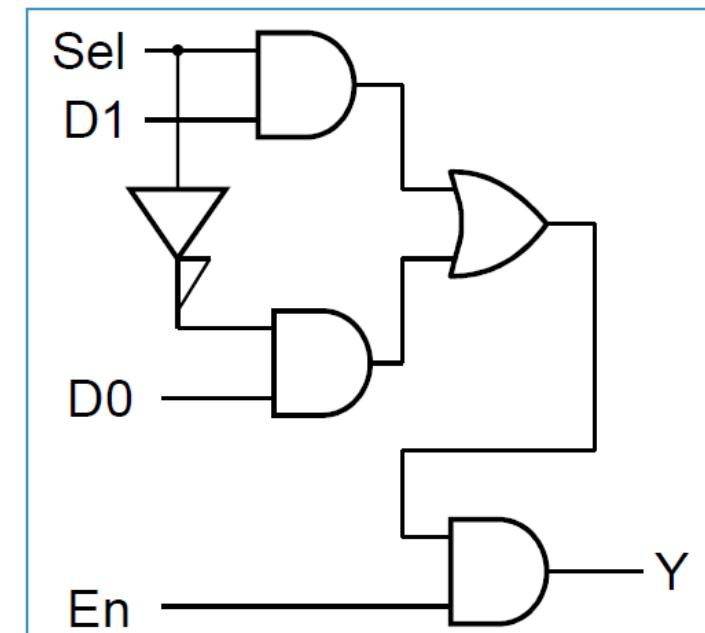
Multiplexeur 2 à 1

- Logiques combinatoires à 4 entrées (En, Sel, D0 et D1) et une sortie (Y)

Table de vérité

En	Sel	D1	D0	Y
'0'	x	x	x	'0'
'1'	'0'	'0'	'0'	'0'
'1'	'0'	'0'	'1'	'1'
'1'	'0'	'1'	'0'	'0'
'1'	'0'	'1'	'1'	'1'
'1'	'1'	'0'	'0'	'0'
'1'	'1'	'0'	'1'	'0'
'1'	'1'	'1'	'0'	'1'
'1'	'1'	'1'	'1'	'1'

		En	Sel	00	01	11	10
D1	D0	00		'0'	'0'	'0'	'0'
		01		'0'	'0'	'0'	'1'
		11		'0'	'0'	'1'	'1'
		10		'0'	'0'	'1'	'0'

$$Y = (\overline{Sel} \cdot D0 + Sel \cdot D1) \cdot EN$$


Multiplexeur 4 à 1

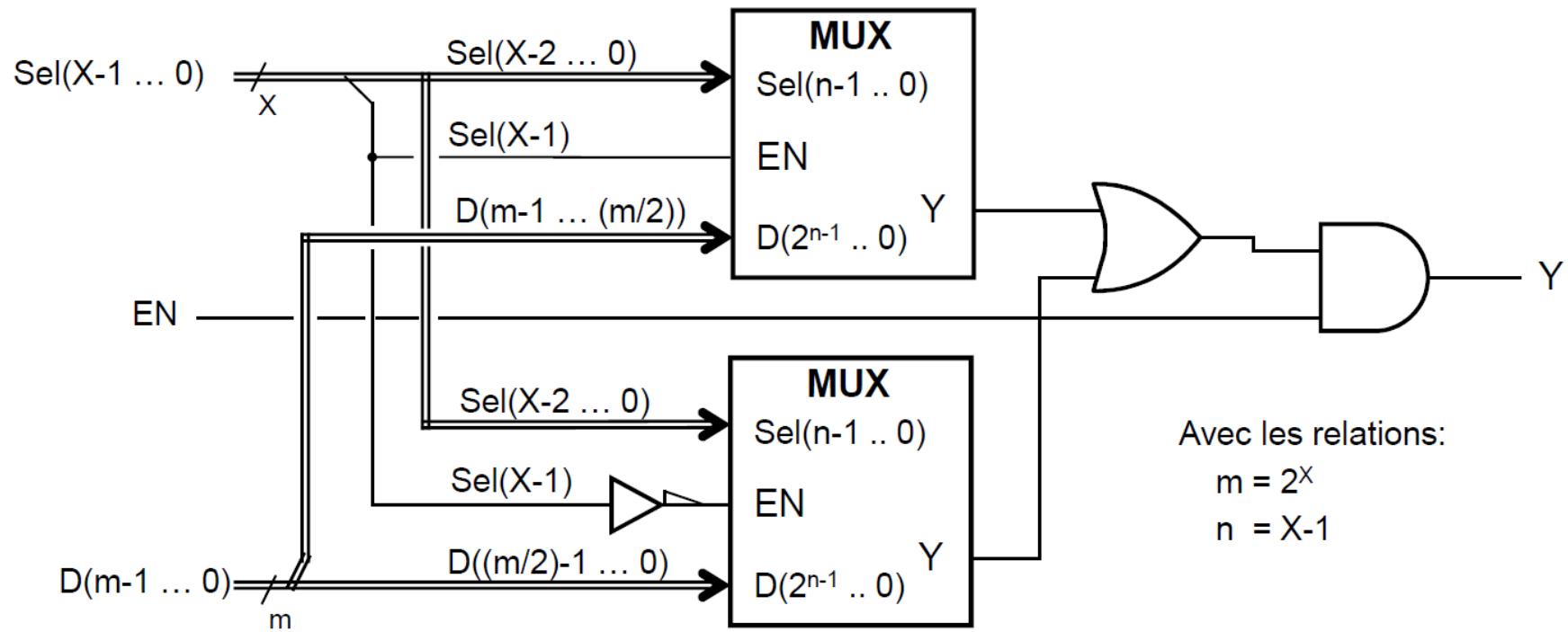
- ❑ Table de vérité compactée:

EN	Sel(1)	Sel(0)	Y
0	X	X	0
1	0	0	D0
1	0	1	D1
1	1	0	D2
1	1	1	D3



Multiplexeur M à 1 décomposé

- Toujours possible de réaliser un multiplexeur M to 1, au moyen de deux multiplexeurs M/2 à 1 et des portes logiques



Multiplexeur 4 to 1

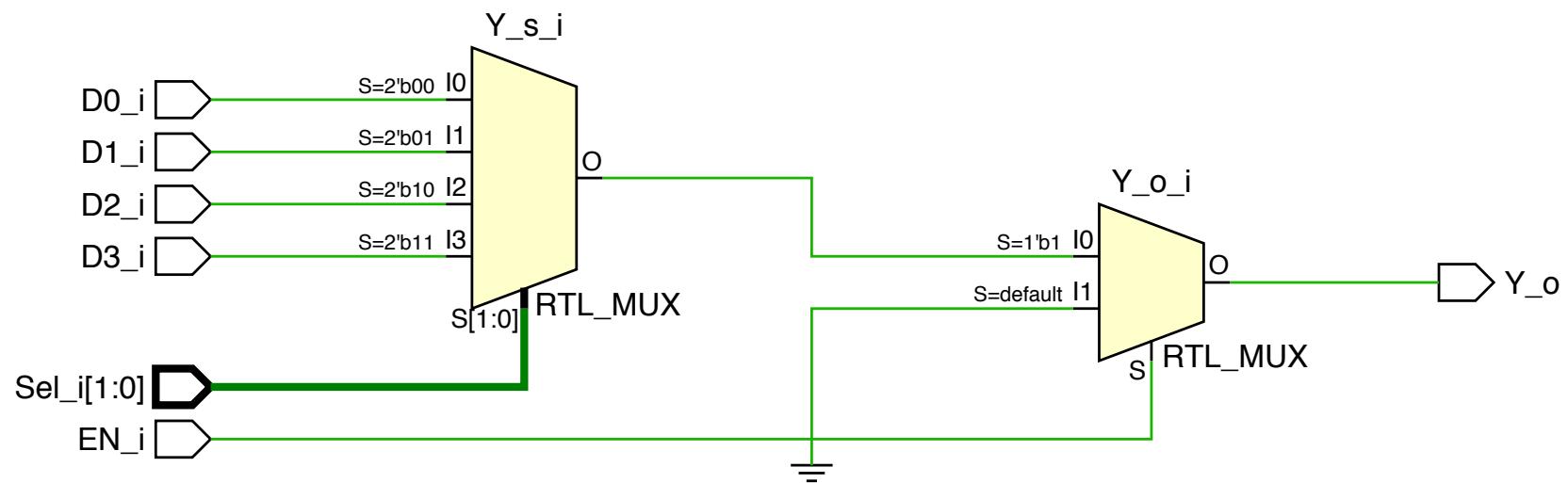
□ Description VHDL pour un Mux 4 à 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Mux_4x1 is
    Port ( Sel_i : in STD_LOGIC_VECTOR (1 downto 0);
            EN_i : in STD_LOGIC;
            D0_i, D1_i, D2_i, D3_i : in STD_LOGIC;
            Y_o : out STD_LOGIC);
end Mux_4x1;

architecture Behavioral of Mux_4x1 is
    signal Y_s : std_logic;
begin
    with Sel_i select
        Y_s <=  D0_i when "00",
                  D1_i when "01",
                  D2_i when "10",
                  D3_i when "11",
                  'X'  when others; -- simulation
    --affectation de la sortie
    Y_o <= Y_s when EN_i = '1' else '0';
end Behavioral;
```



Multiplexeur schéma RTL

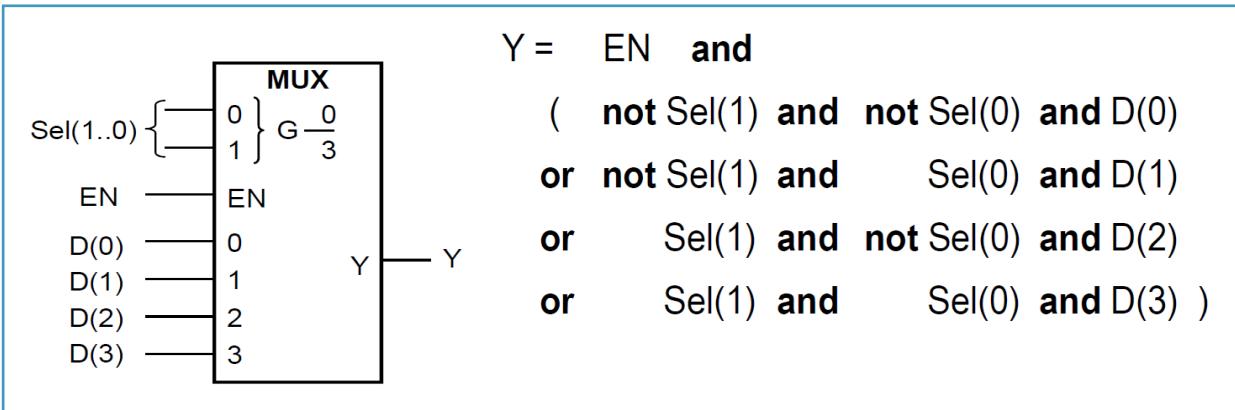


MUX en générateur de fonction

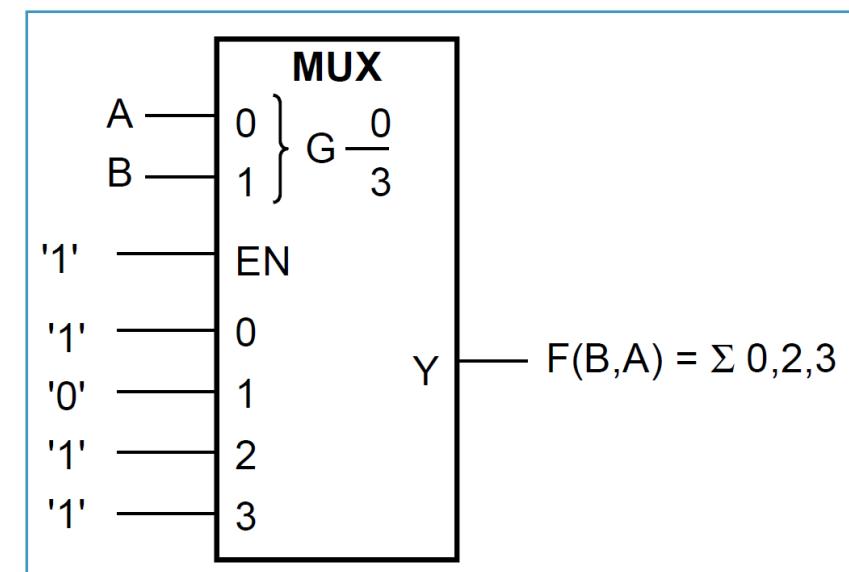
- ❑ On peut donc décomposer le MUX en :
 - Un décodeur
 - Un aiguillage
- ❑ Il est possible de **générer n'importe quelles fonctions** de 2 variables ($\text{Sel}(1), \text{Sel}(0)$) en plaçant sur les entrées D(3)...D(0) les états logiques des sorties souhaitées.
- ❑ Généralisable à n variables.
- ❑ Cette **technique** ressemble aux principes de **Look-up-table**, utilisée dans les circuits programmables **FPGA** → vue plus tard dans le cours



Mux en générateur de fonction



			$F(B,A) = \Sigma 0,2,3$
'1'	B EN	A Sel(1)	F Y
0	X	X	0 pas utilisé
1	0	0	D(0) '1'
1	0	1	D(1) '0'
1	1	0	D(2) '1'
1	1	1	D(3) '1'

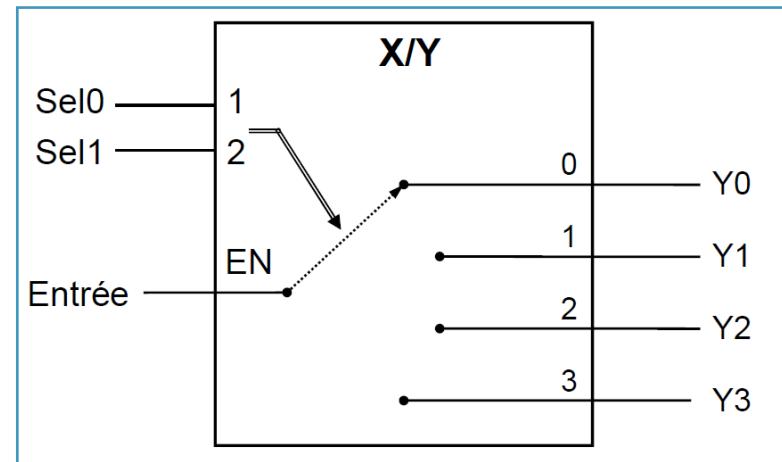


Décodeur - Démultiplexeur

- Un décodeur peut aussi être utilisé comme démultiplexeur

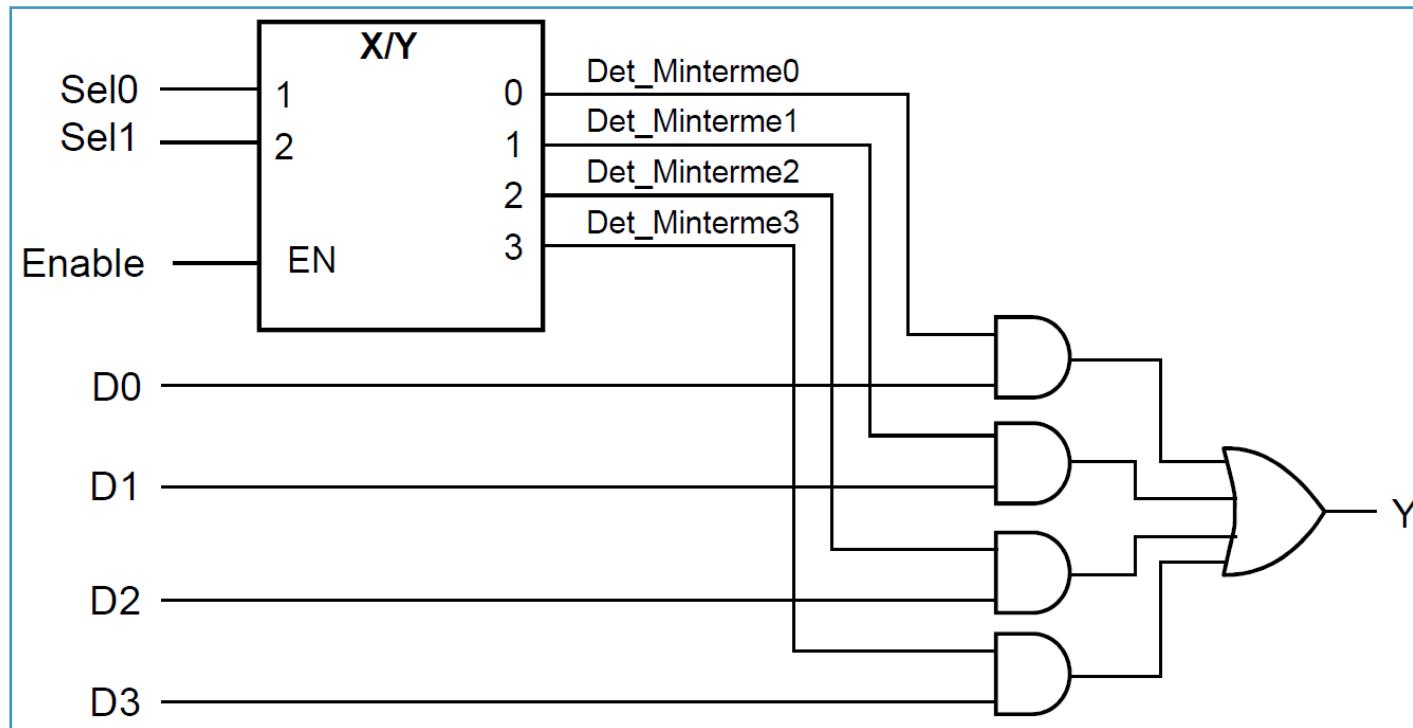
- Pour cela, il faut que:

- L'entrée «Enable» comme entrée de données
- Les entrées de sélection indique la sortie sur laquelle l'entrée de données (EN) doit être aiguillée



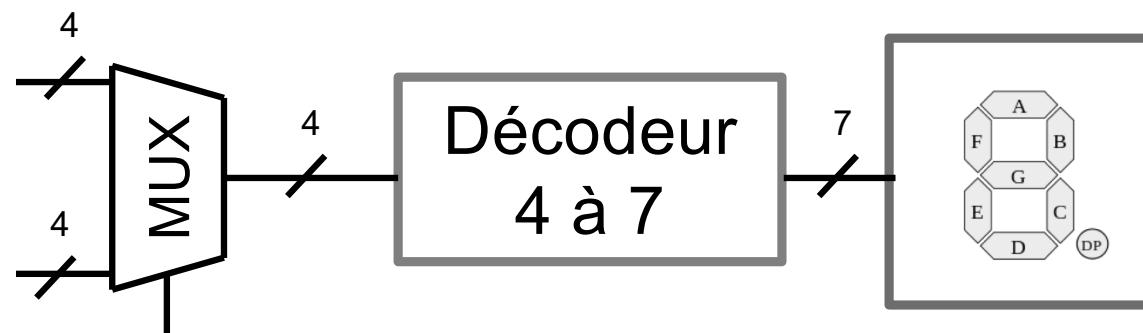
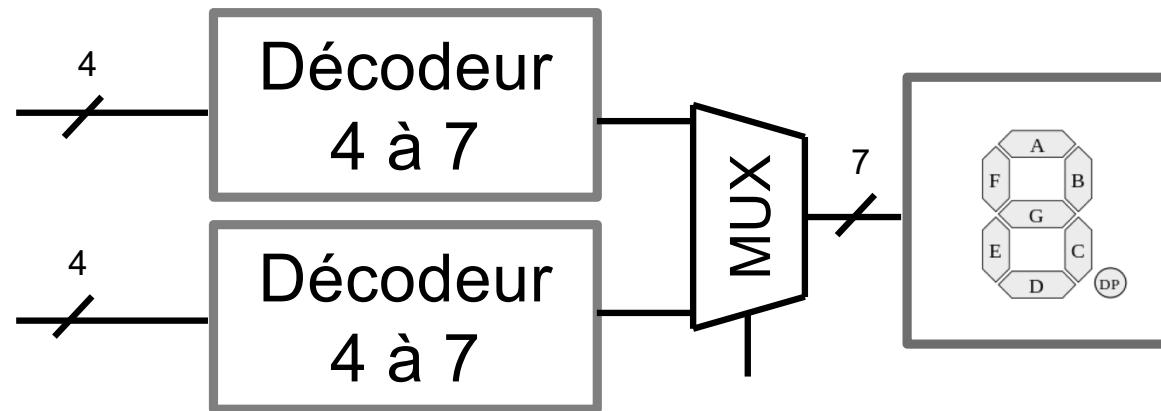
Multiplexeur et décodeur

- Le multiplexeur peut être construit sur la base d'un décodeur



Multiplexage

- ❑ L'utilisation judicieuse des multiplexeurs permet d'économiser du matériel :



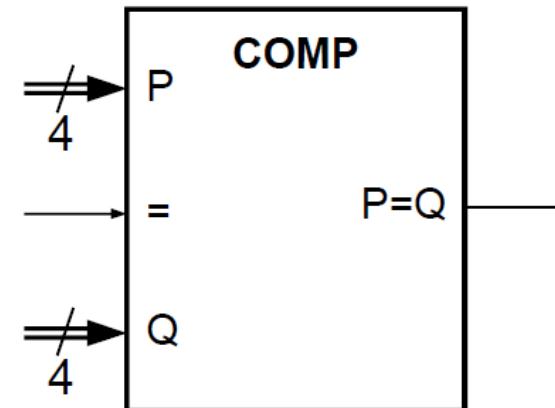
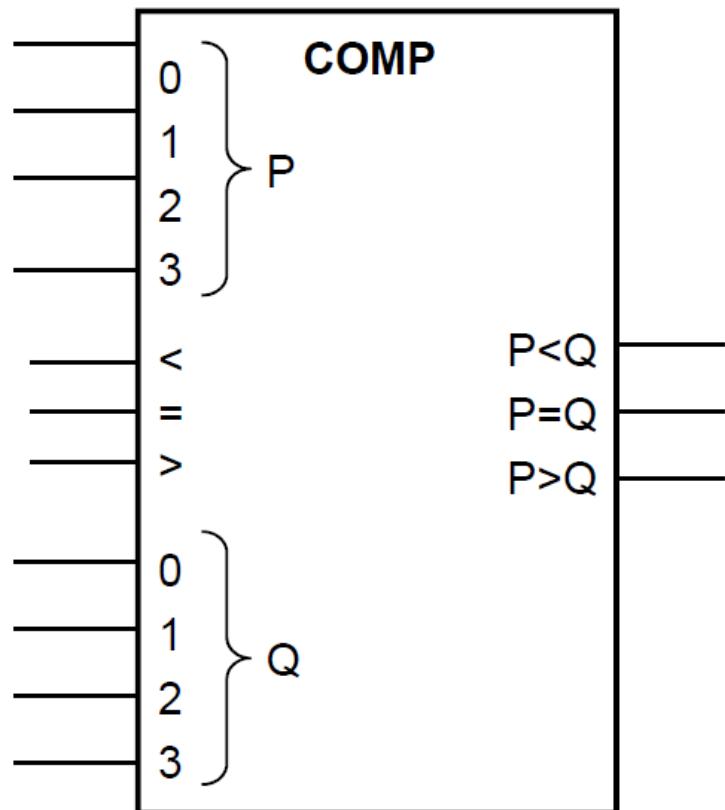
Comparateur

- ❑ But: indiquer si deux nombres binaires sont égaux
- ❑ Les sorties < ou > sont utiles pour les applications numériques
- ❑ Modulaire si dispose d'entrées \leq , \geq



Comparateur

□ Symbole



Décomposition du comparateur

- ❑ Tout comparateur à N bits peut être décomposer en N comparateurs à 1 bit
- ❑ La décomposition peut être de type cascade ou parallèle:
 - Cascade → moins de matériel, plus lent, moins coûteux
 - Parallèle → plus de matériel, plus rapide, plus performant

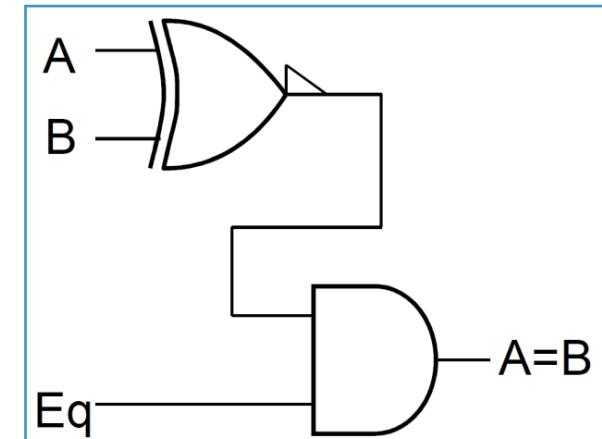


Comparateur 1 bit

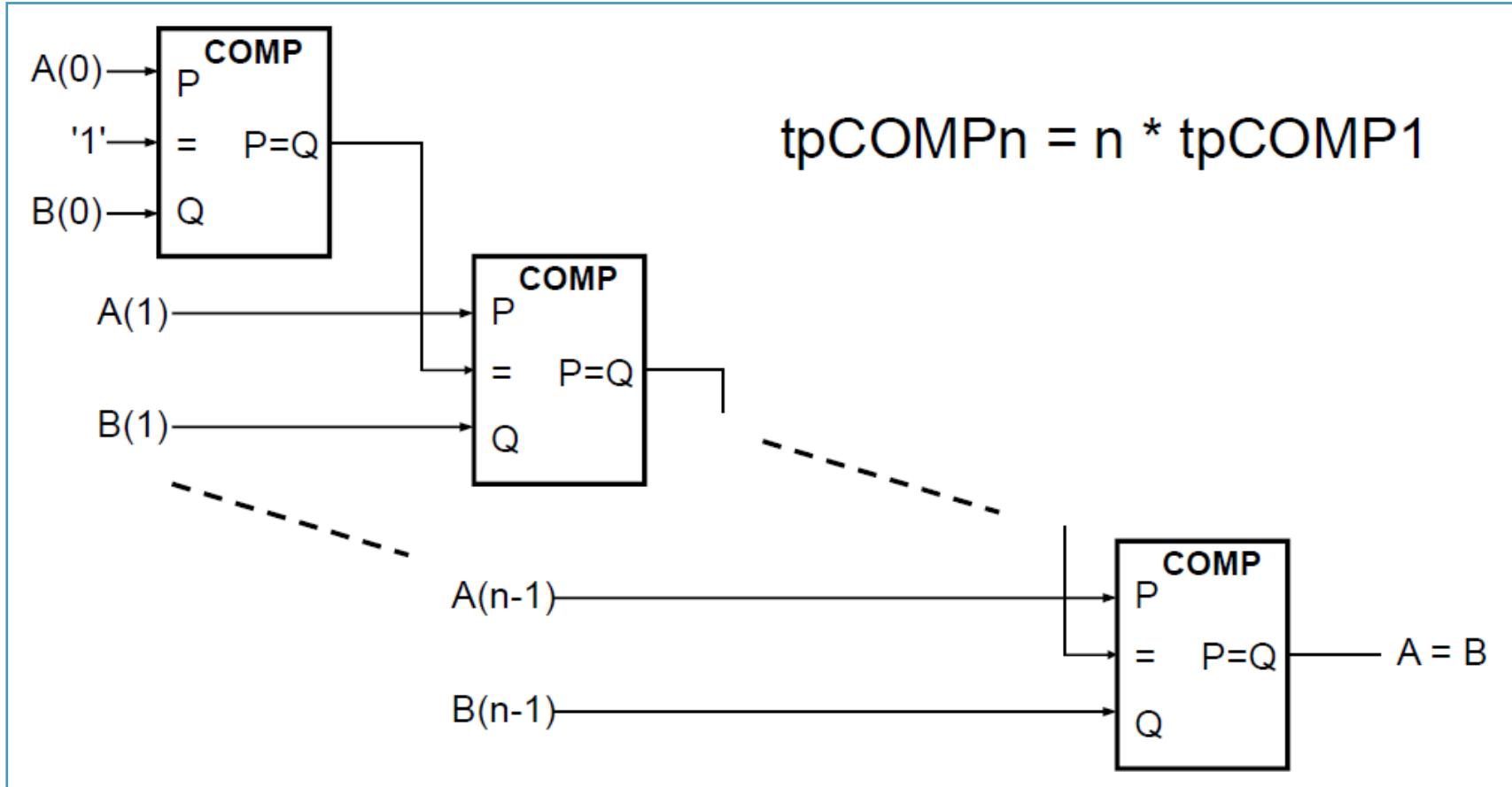
- Un comparateur à 1 bit se réalise au moyen d'une porte ou-exclusive inversée (xnor)

Table de vérité			A=B
Eq	B	A	
'0'	'.'	'.'	'0'
'1'	'0'	'0'	'1'
'1'	'0'	'1'	'0'
'1'	'1'	'0'	'0'
'1'	'1'	'1'	'1'

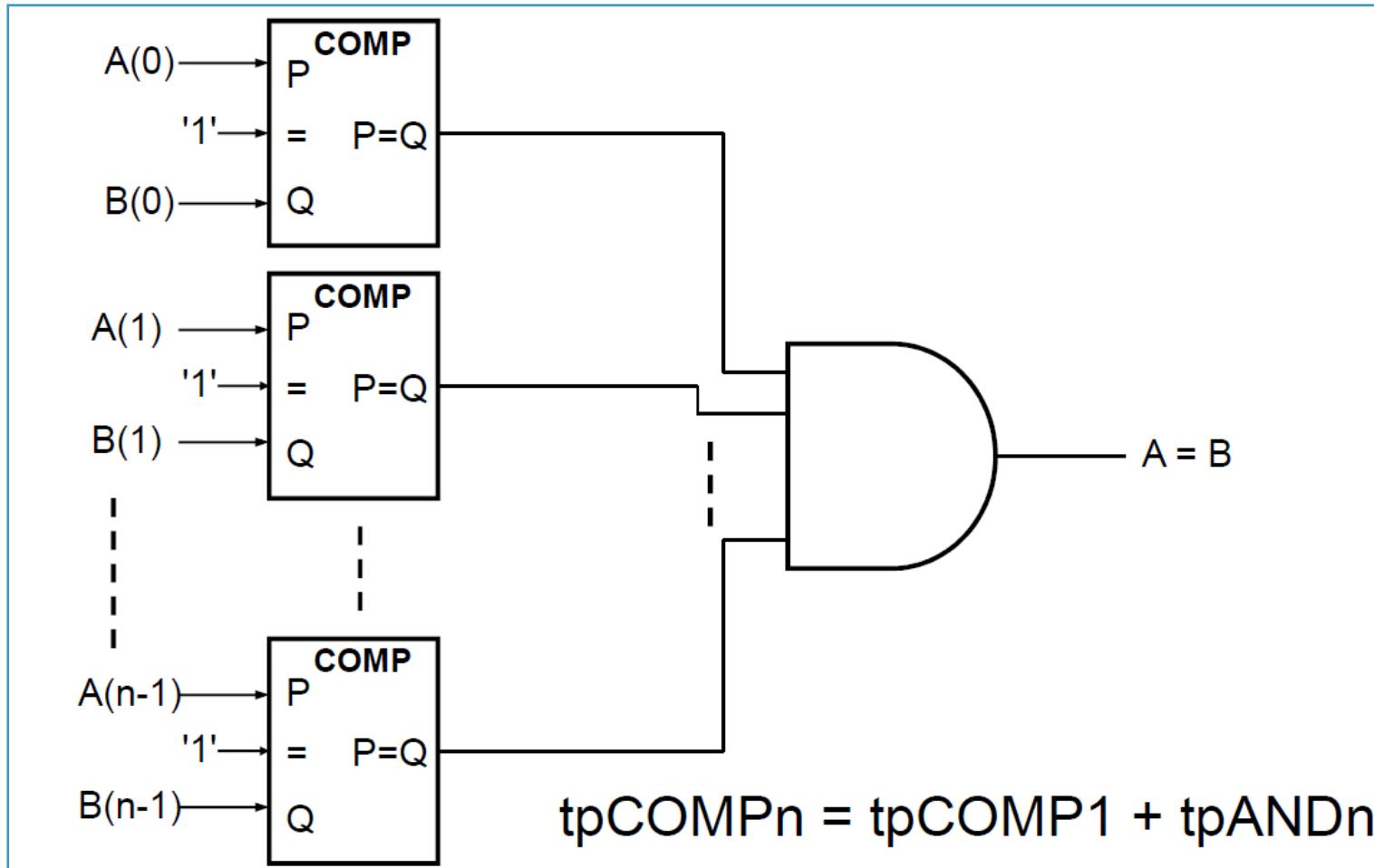
Eq	B		A	
	00	01	11	10
0	'0'	'0'	'0'	'0'
	'1'	'0'	'1'	'0'

$$\begin{aligned} A=B &= (\bar{A} \cdot \bar{B} + A \cdot B) \cdot Eq \\ &= (\overline{A \oplus B}) \cdot Eq \end{aligned}$$


Décomposition en cascade



Décomposition en parallèle



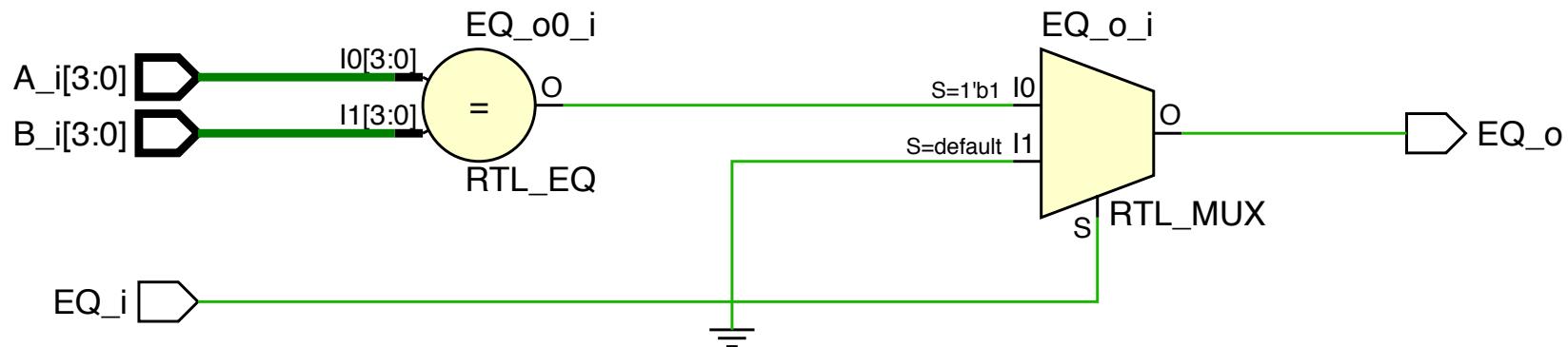
Comparateur 4 bits

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Comp_4 is
    Port ( A_i : in STD_LOGIC_VECTOR (3 downto 0);
            B_i : in STD_LOGIC_VECTOR (3 downto 0);
            EQ_i : in STD_LOGIC;
            EQ_o : out STD_LOGIC);
end Comp_4;

architecture Behavioral of Comp_4 is
    signal EQ_s : std_logic;
begin
    -- test de l'égalité
    EQ_s <= '1' when A_i = B_i else '0';
    -- sortie avec l'entrée de chainage
    EQ_o <= EQ_s when EQ_i = '1' else '0';
end Behavioral;
```



Comparateur 4-bits vue RTL



Encodeur de priorité

- ❑ But: déterminer l'indice de l'entrée active ayant le degré de priorité le plus élevé
- ❑ Module comporte souvent en plus une entrée d'activation
- ❑ Module comporte une sortie indiquant qu'une des entrées au moins est active (nécessaire pour identifier une action sur l'entrée 0)



Encodeur à 4 entrées sans chaînage

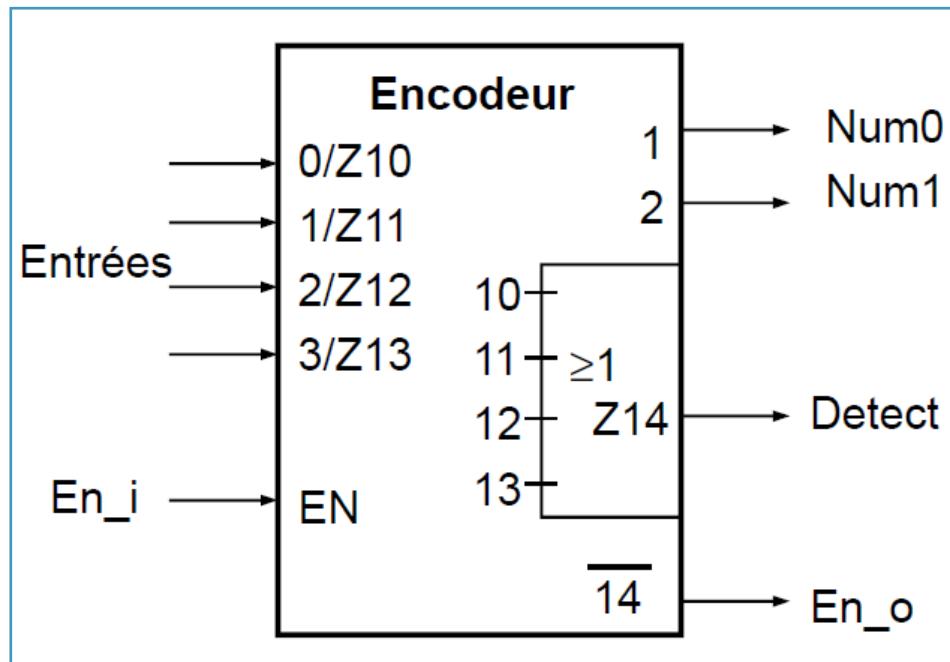


Table de vérité (TDV)

In3	In2	In1	In0	Detect	Num1	Num0
'0'	'0'	'0'	'0'	'0'	'.'	'.'
'0'	'0'	'0'	'1'	'1'	'0'	'0'
'0'	'0'	'1'	x	'1'	'0'	'1'
'0'	'1'	x	x	'1'	'1'	'0'
'1'	x	x	x	'1'	'1'	'1'

Equations logiques

$$\text{Detect} = \text{In3} + \text{In2} + \text{In1} + \text{In0}$$

$$\text{Num1} = \text{In3} + \text{In2}$$

$$\text{Num0} = \text{In3} + \overline{\text{In2}} \cdot \text{In1}$$



Description de l'encodeur

- Description textuelle du fonctionnement de l'encodeur de priorité:
 - Si l'entrée In3, la plus prioritaire, est active alors:
 - Detect est activé & Num = 3
 - sinon...
 - Si l'entrée In2 est active alors:
 - Detect est activé & Num = 2
 - sinon...



Encodeur de priorité

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Encode_4 is
    Port ( In_i : in STD_LOGIC_VECTOR(3 downto 0);
            En_i : in STD_LOGIC;
            Num_o : out STD_LOGIC_VECTOR(1 downto 0);
            Detect_o : out STD_LOGIC;
            En_o : out STD_LOGIC);
end Encode_4;

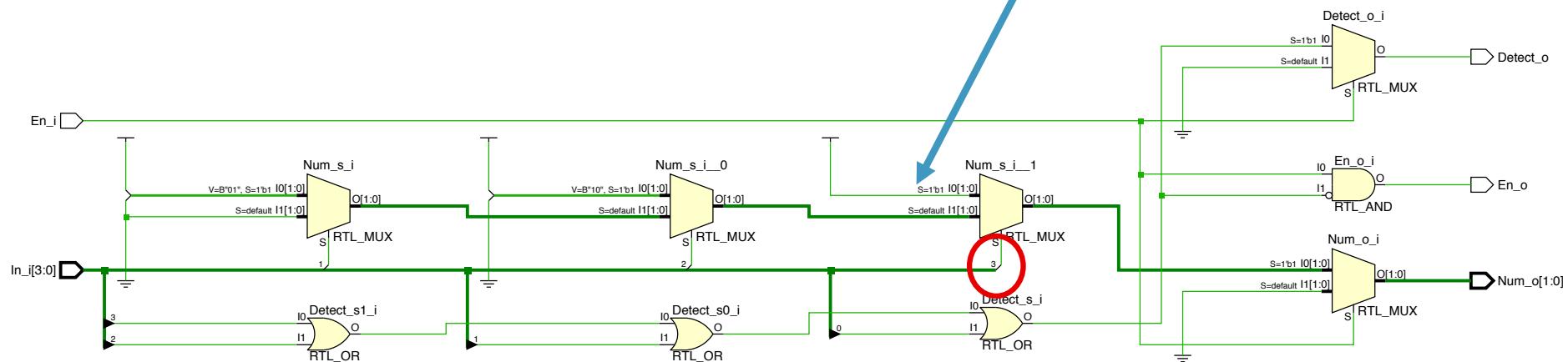
architecture Behavioral of Encode_4 is
    signal Num_s : std_logic_vector(1 downto 0);
    signal Detect_s : std_logic;
begin
    Num_s <= "11" when In_i(3) = '1' else
                "10" when In_i(2) = '1' else
                "01" when In_i(1) = '1' else
                "00" when In_i(0) = '1' else
                "XX"; -- simulation

    Detect_s <= In_i(3) or In_i(2) or In_i(1) or In_i(0);
    Num_o <= Num_s when En_i = '1' else (others => '0');
    Detect_o <= Detect_s when En_i = '1' else '0';
    En_o <= En_i and not Detect_s;
end Behavioral;
```



Encodeur, schéma RTL

11



Additionneur

- ❑ But: réaliser l'addition de deux nombres binaires
- ❑ Ce type de module se réalise fréquemment par une structure en chaîne :
il comporte, en plus de l'entrée des deux nombres, une entrée et une sortie pour les retenues



Addition: cas général

- Un additionneur complet réalise l'addition de deux bits X_i et Y_i , plus le report C_i , en produisant le bit de résultat S_i et le bit de report C_{i+1}

$C(i)$	$B(i)$	$A(i)$	$C(i+1)$	$S(i)$
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Karnaugh → Équations logiques



Addition: cas général

- Un additionneur complet réalise l'addition de deux bits X_i et Y_i , plus le report C_i , en produisant le bit de résultat S_i et le bit de report C_{i+1}

$C(i)$	$B(i)$	$A(i)$	$C(i+1)$	$S(i)$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = \overline{A}\overline{B}C_{IN} + \overline{A}B\overline{C}_{IN} + A\overline{B}\overline{C}_{IN} + ABC_{IN}$$

Factoring out \overline{A} and A :

$$S = \overline{A}(\overline{B}C_{IN} + B\overline{C}_{IN}) + A(\overline{B}\overline{C}_{IN} + BC_{IN})$$

which is

$$S = \overline{A}(B \oplus C_{IN}) + A(\overline{B} \oplus \overline{C}_{IN})$$

Karnaugh → Équations logiques

$$S_i = A_i \oplus B_i \oplus C_i$$

$$C_{i+1} = A_i B_i + C_i(A_i \oplus B_i)$$

$$\begin{aligned}C_{out} &= \Sigma m(3,5,6,7) \\&= X' Y C_{in} + X Y' C_{in} + X Y C_{in}' + X Y C_{in} \\&= (X' Y + X Y') C_{in} + XY(C_{in}' + C_{in}) \\&= (X \oplus Y) C_{in} + XY\end{aligned}$$



Additionneur 1 bit

- Schéma du demi-additionneur (Half-Adder) et d'un additionneur complet 1 bit (Full-Adder):

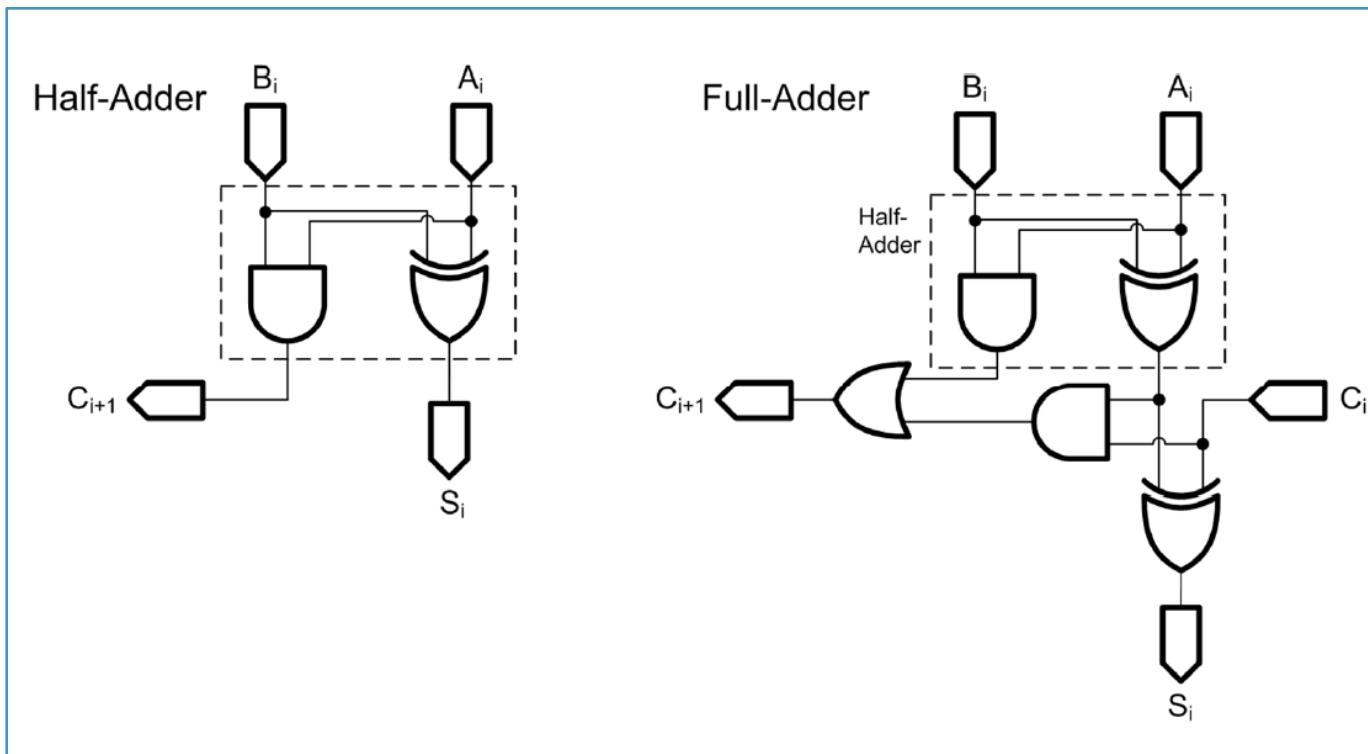
Half-Adder

Full-Adder



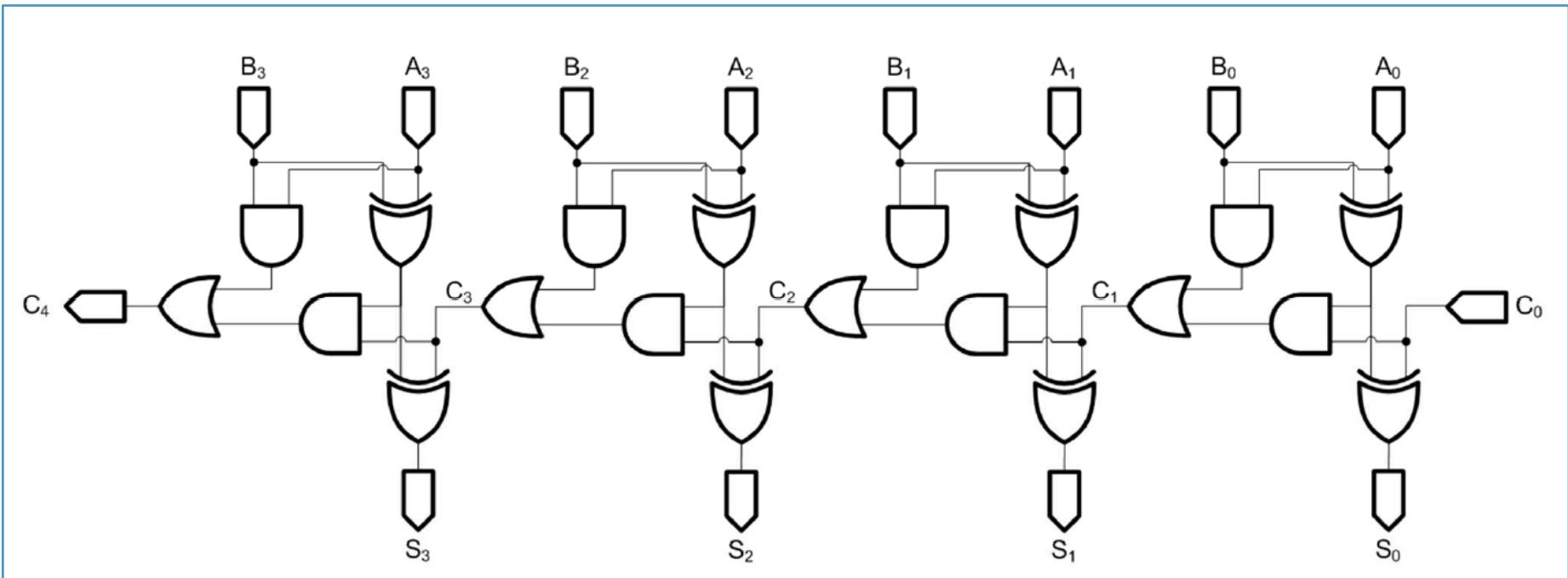
Additionneur 1 bit

- Schéma du demi-additionneur (Half-Adder) et d'un additionneur complet 1 bit (Full-Adder):



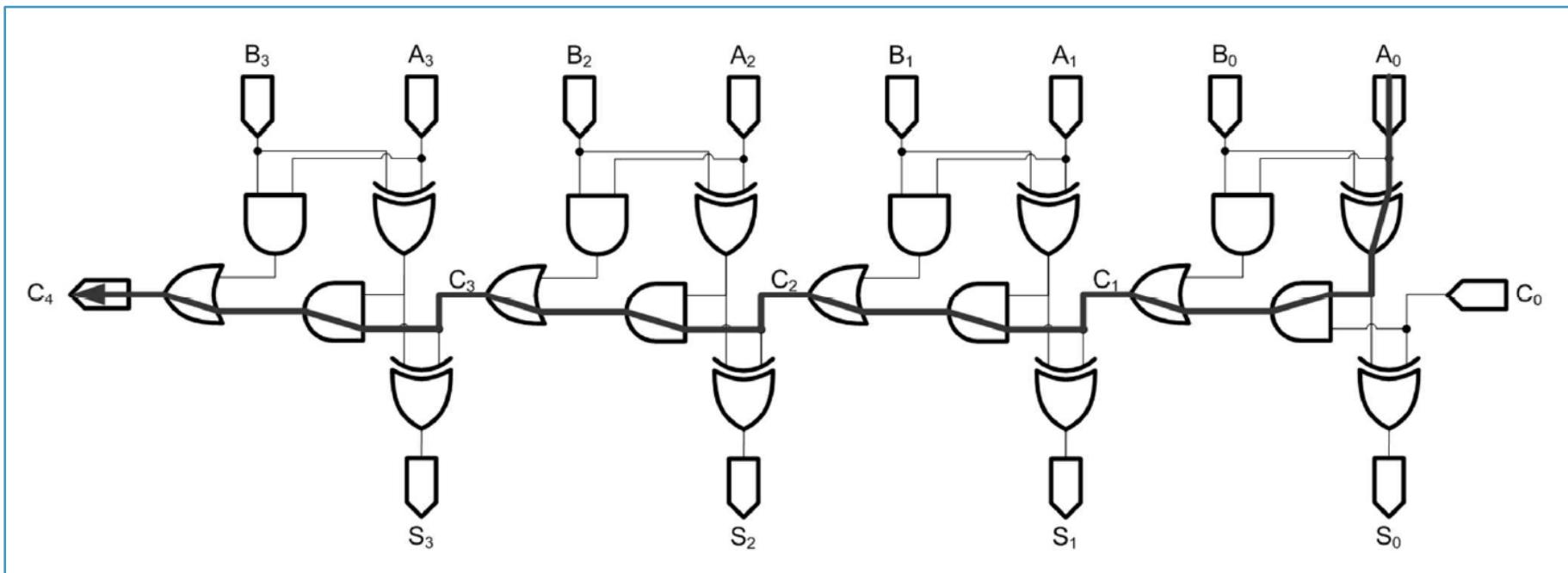
Additionneur 4 bits...

- Schéma additionneur 4 bits avec 4 Full-Adder:



Additionneur 4 bits...

□ Chemin critique additionneur 4 bits:



Additionneur 1 bit

□ Composant additionneur de 1 bit:

```
library IEEE;
use IEEE.Std_Logic_1164.all;
entity Add_1 is
    port( A_i : in Std_Logic; -- Bit du nombre A
          B_i : in Std_Logic; -- Bit du nombre B
          R_i : in Std_Logic; -- Retenue d'entree
          S_o : out Std_Logic; -- Somme
          R_o : out Std_Logic -- Retenue de sortie
    );
end Add_1;

architecture Logique of Add_1 is
begin
    S_o <= A_i xor B_i xor R_i;
    R_o <= ((A_i xor B_i) and R_i) or (A_i and B_i);
end Logique;
```



Additionneur 2 bits:

```
library IEEE;
use IEEE.Std_Logic_1164.all;
entity Add_2 is
    port( NA_i : in Std_Logic_Vector(1 downto 0); -- Nombre A
          NB_i : in Std_Logic_Vector(1 downto 0); -- Nombre B
          R_i : in Std_Logic;                               -- Retenue d'entree
          S_o : out Std_Logic_Vector(1 downto 0); -- Somme
          R_o : out Std_Logic                           -- Retenue de sortie
        );
end Add_2 ;

architecture Struct of Add_2 is
    signal R_0a1_s : Std_logic;
    component Add_1 is
        port( A_i : in Std_Logic; -- Bit du nombre A
              B_i : in Std_Logic; -- Bit du nombre B
              R_i : in Std_Logic; -- Retenue d'entree
              S_o : out Std_Logic; -- Somme
              R_o : out Std_Logic -- Retenue de sortie
            );
    end component;
    for all : Add_1 use entity work.Add_1(Logique);
begin
    Add0 : Add_1 port map(
        A_i => NA_i(0),
        B_i => NB_i(0),
        R_i => R_i,
        S_o => S_o(0),
        R_o => R_0a1_s
      );
    Add1 : Add_1 port map(
        A_i => NA_i(1),
        B_i => NB_i(1),
        R_i => R_0a1_s,
        S_o => S_o(1),
        R_o => R_o
      );
end Struct;
```



Opérations et dépassement

❑ Carry

- Dépassement de capacité pour les additions de nombres
- Généralement abrégé: C

❑ Borrow

- Dépassement de capacité pour les soustractions de nombres

❑ Overflow

- Dépassement de capacité pour les additions et soustractions de nombres
- Généralement abrégé: V



Méthodologie de conception des systèmes combinatoires

- A partir du cahier des charges, l'ingénieur...
 - décrit le **comportement structurel** du système
 - identifie clairement les **entrées** et les **sorties** du système combinatoire
 - **décompose** en sous-systèmes = plusieurs blocs (si nécessaire, selon la complexité)
 - décrit clairement chaque bloc (fonctions et entrées/sorties), puis en VHDL **synthétisable**
 - simule chaque description VHDL afin de valider leur bon fonctionnement
 - intègre les blocs
 - vérifie que le circuit final remplit le cahier des charges.



Description VHDL des bascules

- ❑ Le VHDL n'a pas d'instruction spécifique pour déclarer des bascules
- ❑ Le synthétiseur doit reconnaître la bascule d'après la description → **Inference**
- ❑ Il faut donc utiliser des descriptions standards pour éviter de produire des circuits dont le fonctionnement est erroné
- ❑ La description d'une bascule se fait toujours dans un processus séquentiel.



Comportement par défaut du VHDL

- ❑ Lors de l'utilisation d'instructions d'affectation:

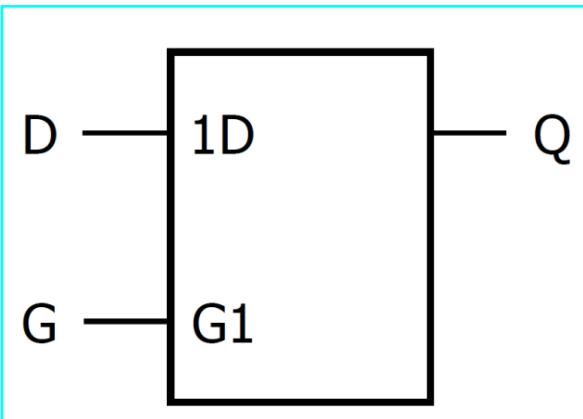
Cas non traité → VHDL maintient l'état du signal

- ❑ Ce principe permet la description des éléments mémoires.
- ❑ Cette fonctionnalité est aussi un piège car elle peut générer des latchs non désirés.



Verrou/Latch D

□ Symbole CEI:



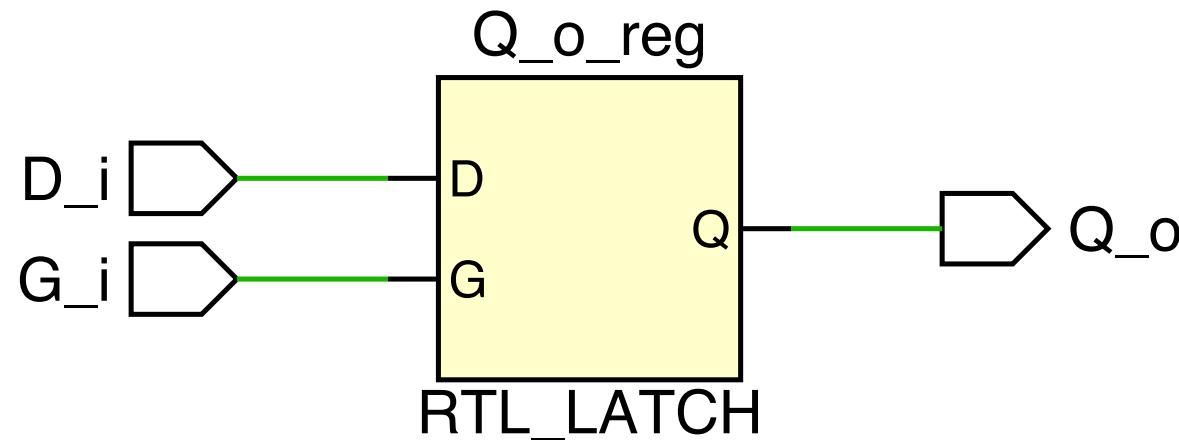
```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity D_Latch is
    port( D_i : in Std_Logic ;
          -- Entrée D
          G_i : in Std_Logic ;
          -- Enable
          Q_o : out Std_Logic
          -- sortie de la bascule
    );
end D_latch ;

architecture Comport of D_Latch is
begin
    Mem: process(G_i,D_i)
    begin
        if (G_i = '1') then
            Q_o <= D_i ;
            -- else implicite => maintien par defaut
        end if;
    end process ;
end Comport;
```

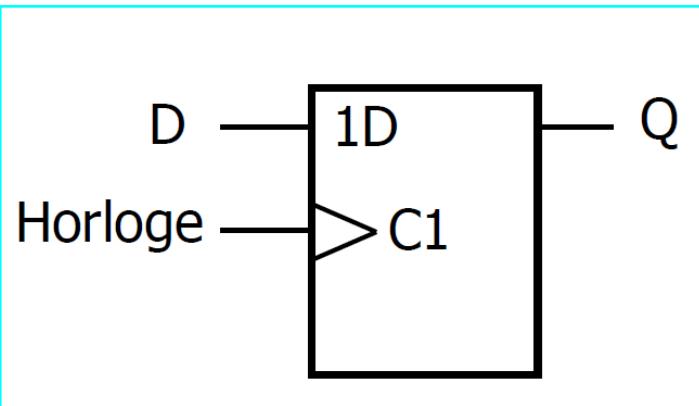


Verrou/Latch D: vue RTL



Bascule D avec Reset (DFF)

Symbole CEI



Fonctionnement:

Horloge	D	Q+
↓	0	0
↑	1	1

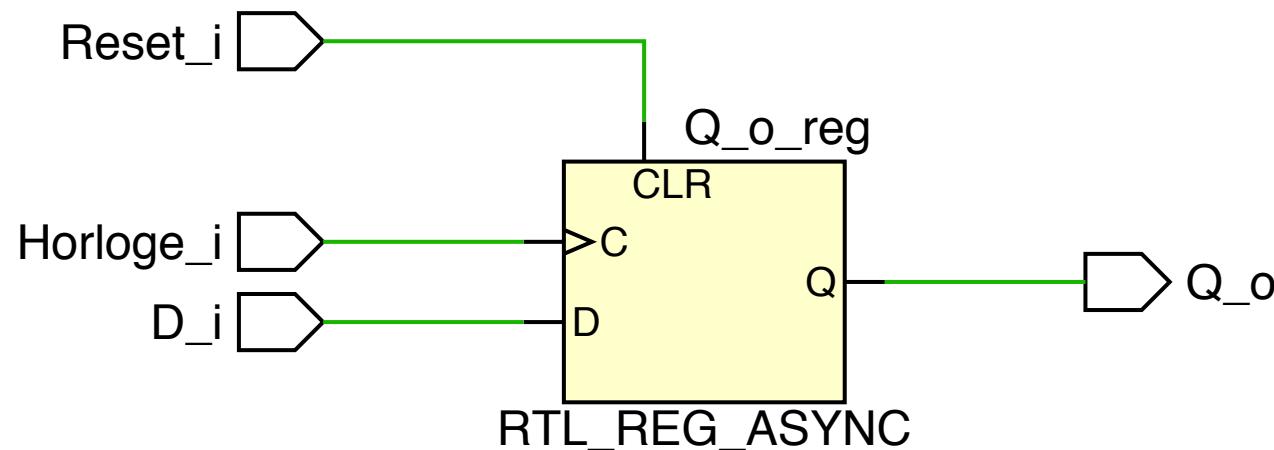
```
library IEEE;
use IEEE.Std_Loic_1164.all;

entity DFF is
    port(Horloge_i : in Std_Loic ;
          -- Entrée de commande
          Reset_i : in Std_Loic ;
          -- Mise à zero asynchrone
          D_i : in Std_Loic ;
          -- Entrée de donnée
          Q_o : out Std_Loic
          -- sortie de la bascule
        );
end DFF ;

architecture Comport of DFF is
begin
    Mem: process (Reset_i, Horloge_i)
    begin
        if (Reset_i = '1') then
            -- comportement asynchrone prioritaire
            Q_o <= '0' ;
        elsif Rising_Edge(Horloge_i) then
            -- comportement synchrone
            Q_o <= D_i ;
            -- else implicite => maintien par défaut
        end if;
    end process ;
end Comport ;
```



DFF avec Reset: vue RTL



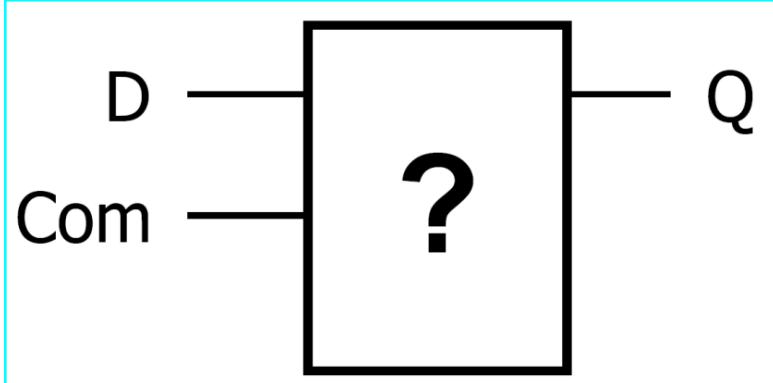
Exercice

- Quel est le type de cette mémoire?

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity Bascule is
    port (D : in Std_Logic;
          Com : in Std_Logic;
          Q : out Std_Logic
        );
end Bascule;

architecture Comport of Bascule is
begin
    process(Com)
    begin
        if Com = '1' then
            Q <= D;
        end if;
    end process;
end Comport;
```



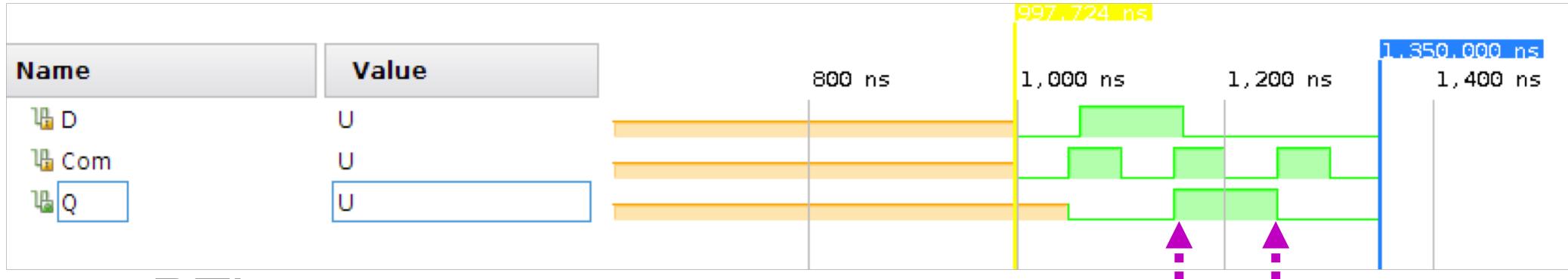
Synthèse de cette mémoire

- Description ambiguë car deux informations se contredisent:
 - Le processus se réveille au changement de COM (et pas D)
→ Action au changement de valeur → **flip-flop**
 - La condition du test spécifie un niveau haut, et pas un flanc avec rising_edge
→ Action sur un niveau logique → **latch**

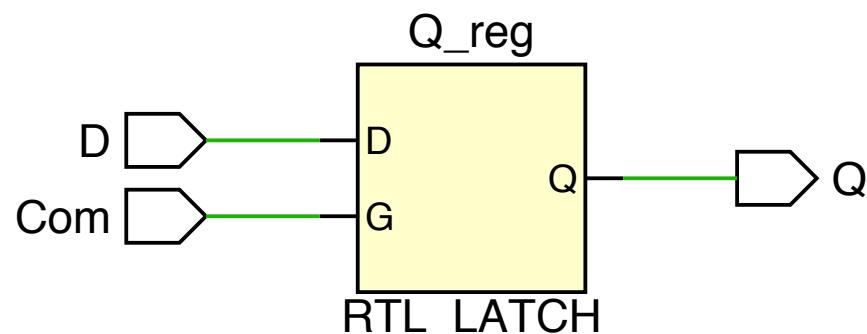


Résultats: Simulation et RTL

❑ Simulation



❑ RTL



Remarque:

La **simulation** indique qu'il s'agit d'une **bascule**. Pour RTL, il s'agit d'un **latch**.



Conclusion de cet exercice

- ❑ Le comportement en simulation du modèle peut être différent du matériel obtenu
- ❑ Cette description est non-synthétisable
 - Interdite par la norme IEEE 1076.6 – 1999
- ❑ Un code qui fonctionne en simulation ne fonctionnera pas forcément dans la FPGA
- ❑ Une méthodologie rigoureuse est nécessaire pour écrire des composants VHDL



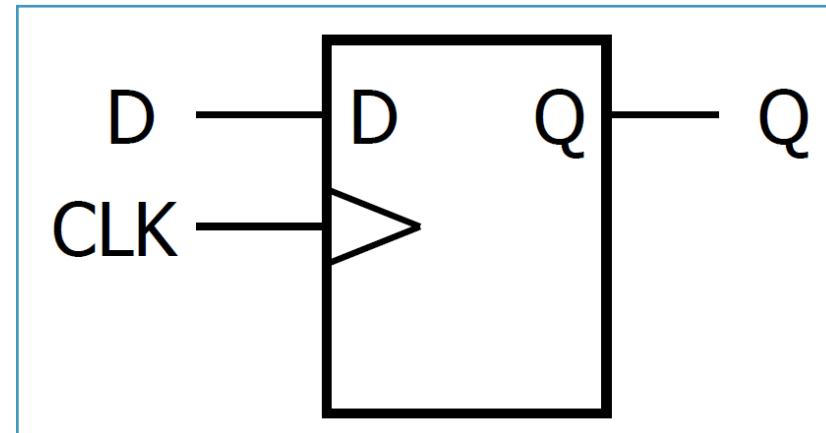
Description simple d'une DFF

- La façon d'écrire une DFF le plus simplement est la suivante:

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity Flip_Flop is
    port (D : in Std_Logic;
          Clock : in Std_Logic;
          Q : out Std_Logic
        );
end Flip_Flop;

architecture Comport of Flip_Flop is
begin
    process(Clock)
    begin
        if Rising_Edge(Clock) then
            Q <= D;
        end if;
    end process;
end Comport;
```



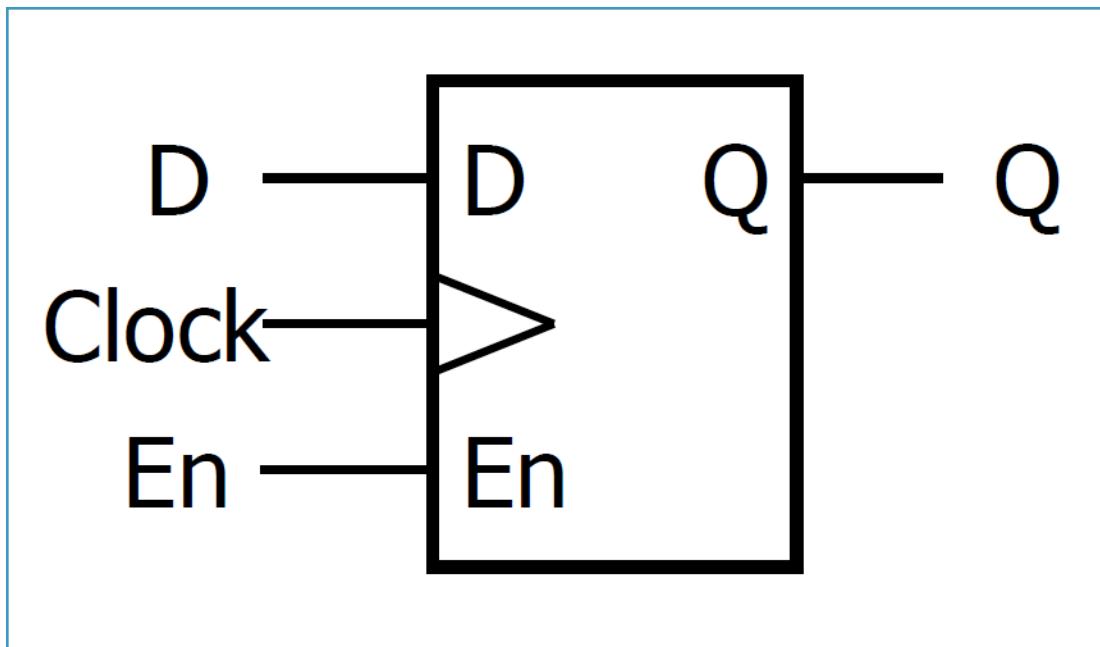
Synthèse du flip-flop

- ❑ Cette description se synthétise parfaitement sur tous les synthétiseurs actuels
- ❑ Deux informations confirment la notion de flip-flop:
 - Le processus ne réagit que sur l'horloge CLK
 - La condition du test spécifie clairement un changement de valeur (`rising_edge`)



Description DFFE

- Description d'un flip-flop avec maintien (Enable)
- Symbole DFFE:



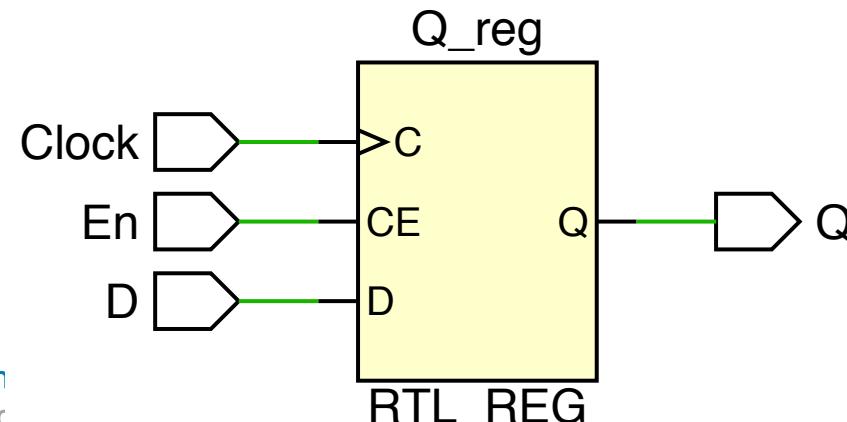
Description DFFE

- Les outils Xilinx génèrent des DFFE lorsque dans le processus séquentiel, selon les instructions séquentielles exécutées dans la portion `rising_edge`, le signal Q n'est pas toujours affecté, d'où le maintien de la valeur actuelle.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DFFE_I is
    Port ( D, Clock, En: in STD_LOGIC;
           Q : out STD_LOGIC);
end DFFE_I;

architecture Behavioral of DFFE_I is
begin
    process(Clock)
    begin
        if rising_edge(clock) then
            if En = '1' then
                Q <= D;
            end if;
        end if;
    end process;
end Behavioral;
```



Etat initial d'une bascule

- ❑ Au début d'une simulation, tous les signaux sont à l'état 'U'
 - ❑ A l'enclenchement d'un système numérique, le contenu des bascules n'est pas déterministe.
- Une initialisation est indispensable
- ❑ Comment forcer l'état initial d'une bascule en VHDL ?



Bascule avec état initial

1ère solution:

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity Flip_Flop is
    port (D : in Std_Logic;
          Clock : in Std_Logic;
          Q : out Std_Logic := '0' ←
    );
end Flip_Flop;

architecture Comport of Flip_Flop is
begin
    process(Clock)
    begin
        if Rising_Edge(Clock) then
            Q <= D;
        end if;
    end process;
end Comport;
```

2ème solution:

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity Flip_Flop is
    port (D : in Std_Logic;
          Clock : in Std_Logic;
          Reset : in Std_Logic;
          Q : out Std_Logic
    );
end Flip_Flop;

architecture Comport of Flip_Flop is
begin
    process(Clock, Reset)
    begin
        if Reset = '1' then
            Q <= '0'; ←
        elsif Rising_Edge(Clock) then
            Q <= D;
        end if;
    end process;
end Comport;
```



Discussion initialisation bascule

- ❑ 1^{ère} solution:
 - Fonctionne en simulation VHDL
 - Ne sera pas synthétisé : «soft»
- ❑ 2^{ème} solution:
 - Fonctionne en simulation VHDL
 - Sera correctement synthétisé: «hard»



Reset Synchrone vs Asynchrone

Synchrone

```
process(Clock)
begin
    if rising_edge(Clock) then
        if reset = '1' then
            Q1 <= '0';
        else
            Q1 <= D;
        end if;
    end if;
end process;
```

Asynchrone

```
process(Reset, Clock)
begin
    if Reset = '1' then
        Q1 <= '0';
    elsif rising_edge(Clock) then
        Q1 <= D;
    end if;
end process;
```

NB: Respectez rigoureusement la liste de sensibilité !

- ❑ Choisir la technique au début du projet et à appliquer à tous les composants
- ❑ La technologie peut être optimisée pour l'une ou l'autre technique.



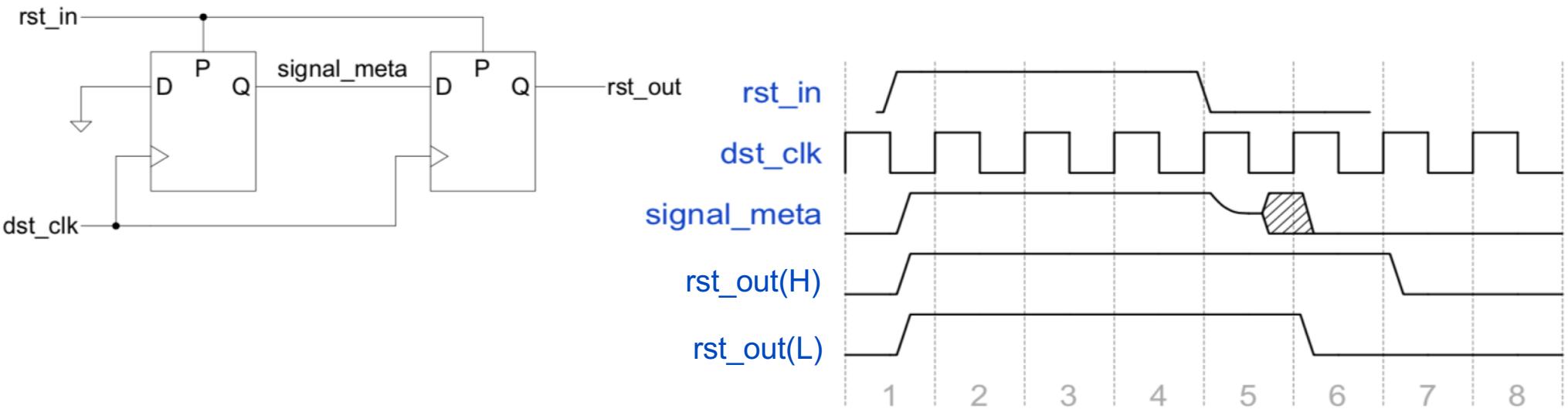
Signal Reset

- ❑ Le signal Reset doit permettre un démarrage stable d'un système en initialisant les machines d'états, compteurs, bascules, etc.
- ❑ Les bascules sont des circuits asynchrones:
 - Les sorties d'un circuit asynchrone ne sont pas déterministes si plusieurs entrées changent simultanément.
 - Si le signal Reset est désactivé en même temps que le flanc actif de l'horloge, les bascules et à fortiori tout le système peuvent entrer dans un état métastable.



Signal Reset (2)

- La solution consiste à rendre le signal Reset synchrone.

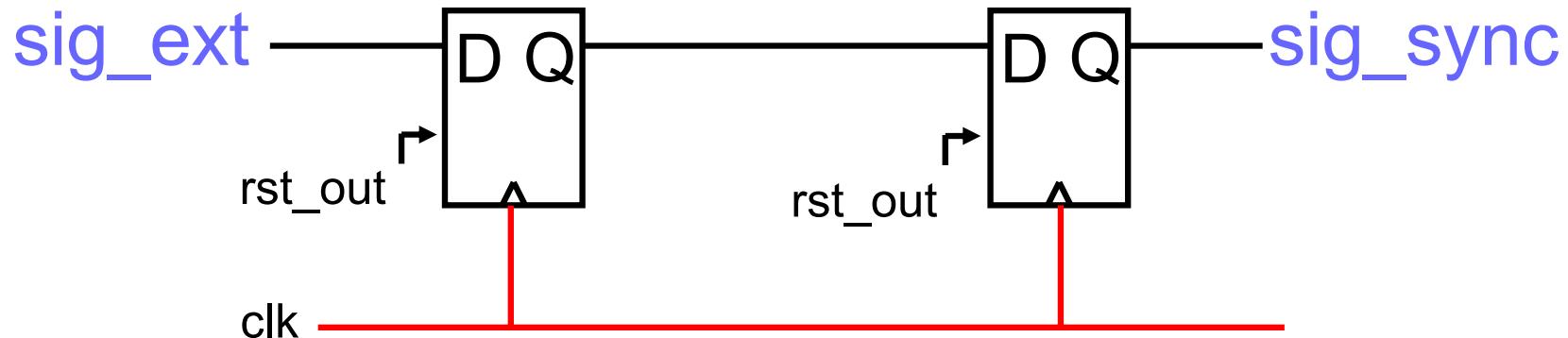


- C'est `rst_out` qui est appliqué au reste du système.



Synchronisation des signaux d'entrée

- Comme dans le cas du Reset, il faut s'assurer que les signaux externes ne produisent pas de métastabilité.



- Cette solution introduit un retard de 2 périodes d'horloge.



Bonnes pratiques

- ❑ Quelques recommandations de Xilinx :
 - Pour des raisons de performance, utiliser la logique positive pour les signaux de reset, clock enable (CE du DFFE) et un flanc actif montant pour l'horloge.
 - Si le design nécessite de la logique négative:
 - décrire en logique positive tous les composants
 - dans le composant TOP-LEVEL, faire les adaptations.
 - Eliminer les latchs sauf si c'est consciemment voulu; les outils Xilinx avertissent de la présence de latchs.
- ❑ Consulter les schémas RTL peut faire économiser beaucoup de temps de développement!



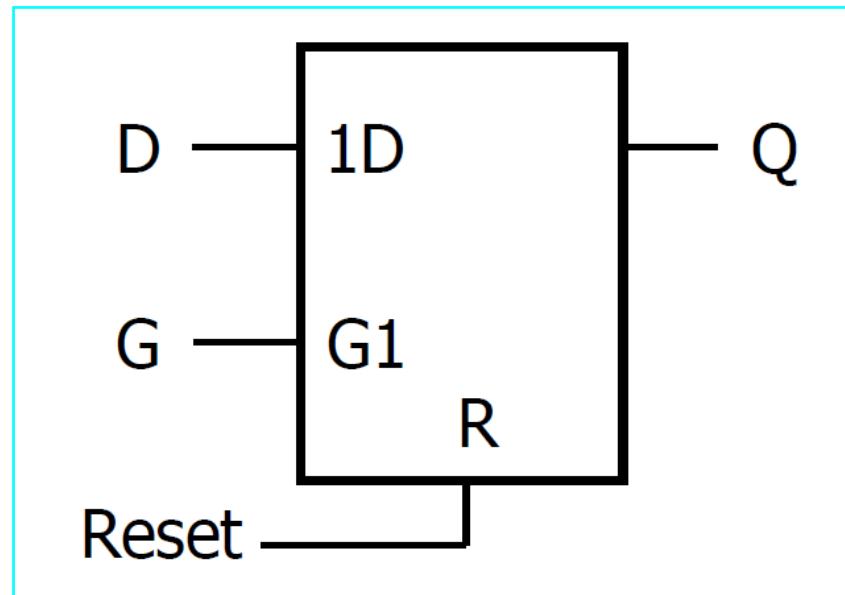
Exercices: Description d'éléments mémoires

- ❑ Pour chaque exercice:
 - Etablir la description du composant VHDL synthétisable de l'élément
 - Produire le schéma RTL, puis
 - Simuler le fonctionnement de l'élément



Exercice 1: Latch D

- Etablir la description VHDL du latch D suivant:

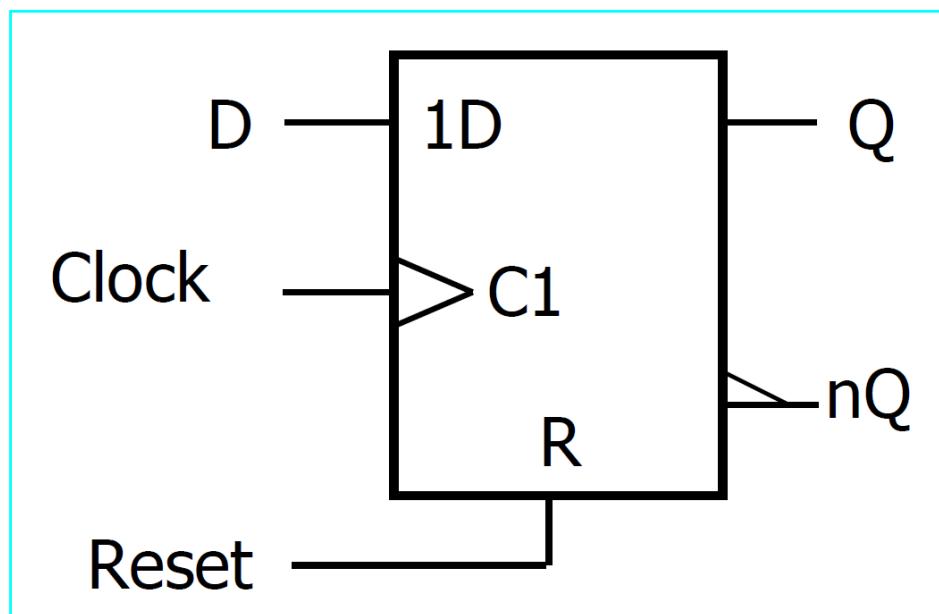


- Quel est le type d'action des signaux suivants:
 - G, Reset



Exercice 2: DFFb

- Etablir la description VHDL du flip-flop D suivant:



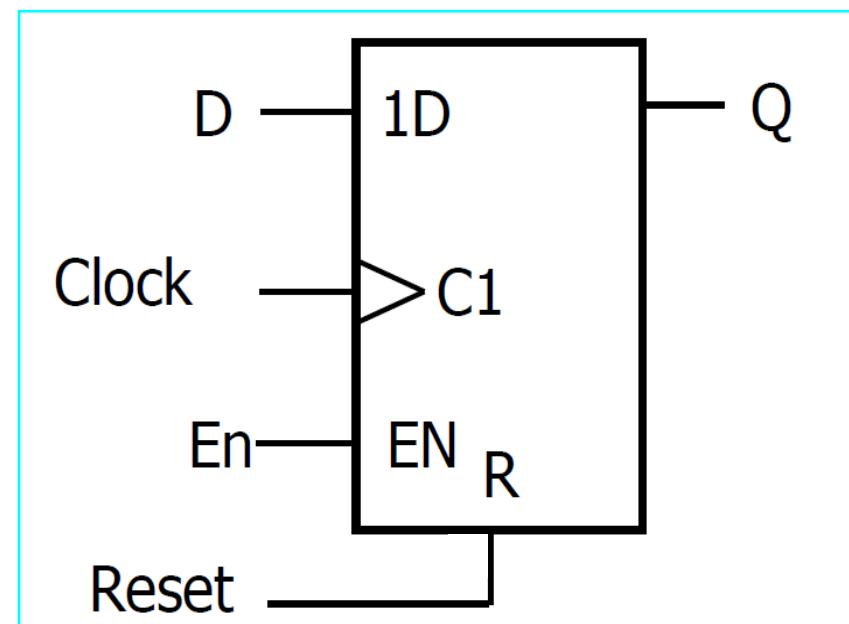
- Quel est le type d'action des signaux suivants:
 - Clock, Reset



Exercice 3: DFF avec Enable

- Modéliser la description VHDL du flip-flop D avec enable suivant, nommée DFFE:
- Fonctionnement:

En	D	Q+
0	x	Q
1	0	0
1	1	1

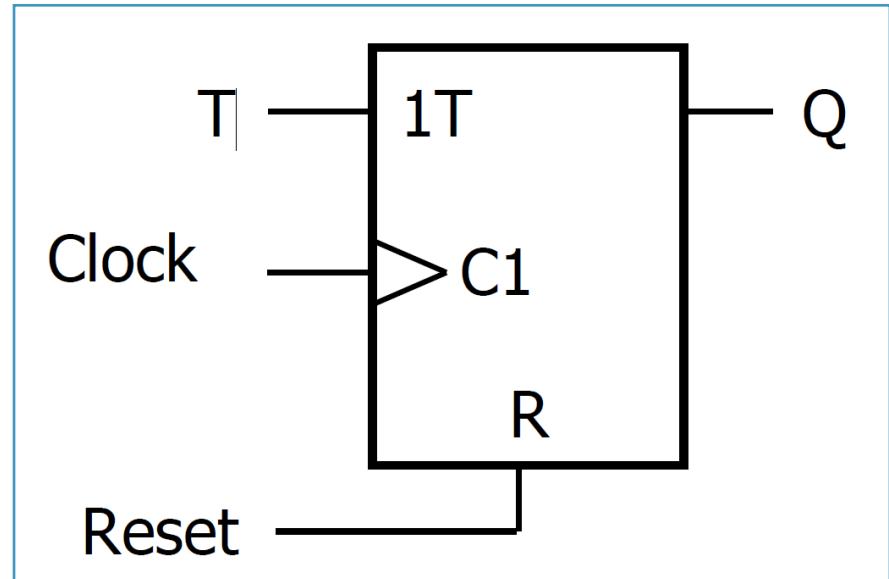


Exercice 4: Flipflop T

- Etablir la description VHDL du flip-flop T suivant:

- Fonctionnement:

T	Q+
0	Q
1	not Q



Exercice 5: Chronogramme

- Compléter le chronogramme suivant pour un Flip-flop D actif au flanc montant et pour un verrou (latch):

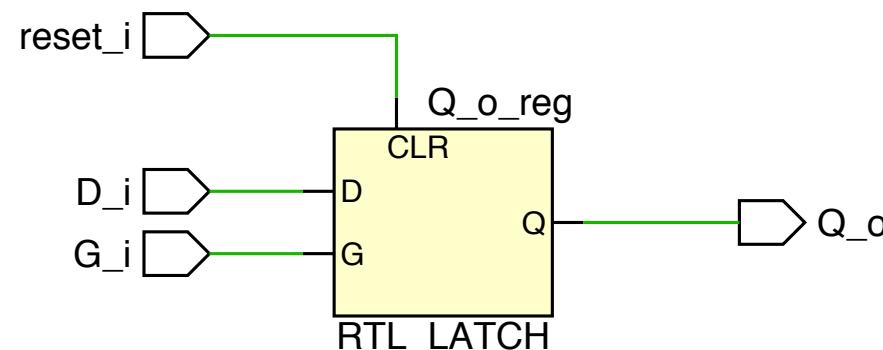
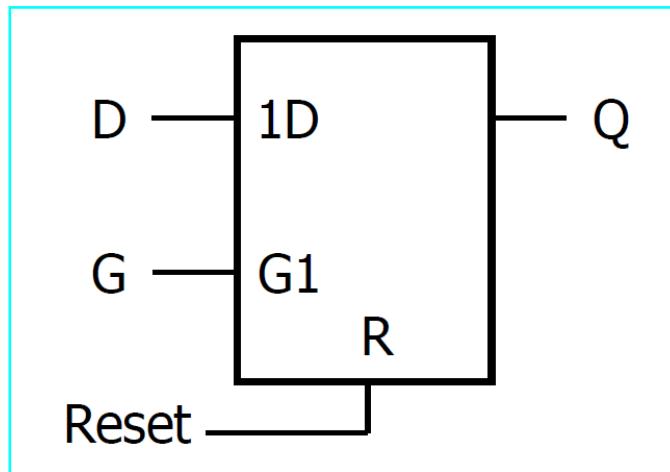


$Q_{flip-flop}$

Q_{verrou}



Solution 1: Latch D

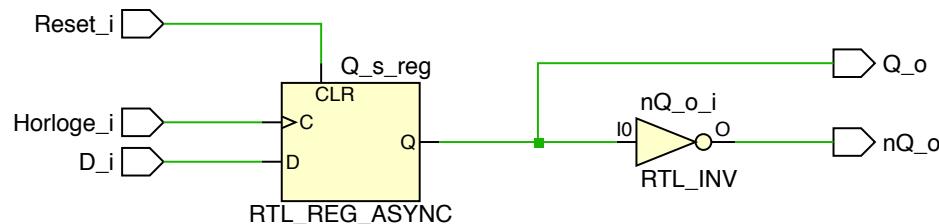
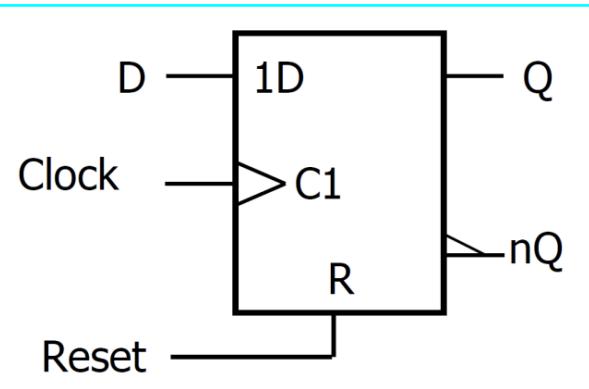


```
library IEEE;
use IEEE.Std_Logic_1164.all;
entity Latch is
    port(d_i, g_i, reset_i: in Std_Logic;
          q_o: out Std_Logic
         );
end Latch;

architecture Comport of Latch is
begin
    process (g_i, d_i, reset_i)
    begin
        if reset_i = '1' then
            q_o <= '0';
        elsif g_i = '1' then
            q_o <= d_i;
        end if;
    end process;
end Comport;
```



Solution 2: DFFb



```
library IEEE;
use IEEE.Std_Loic_1164.all;

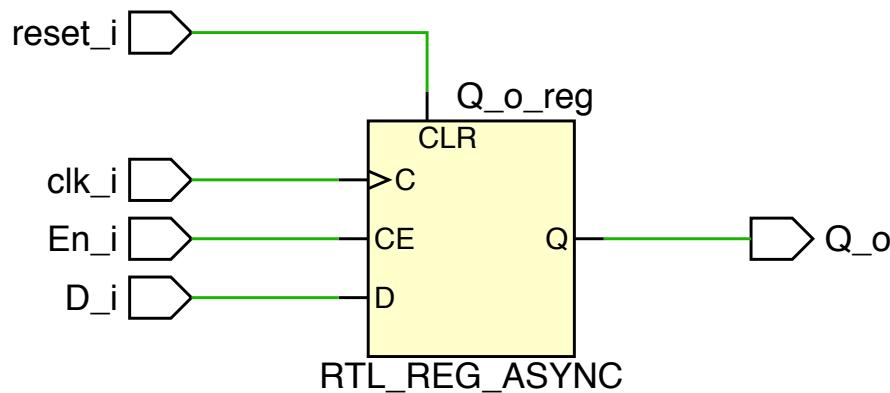
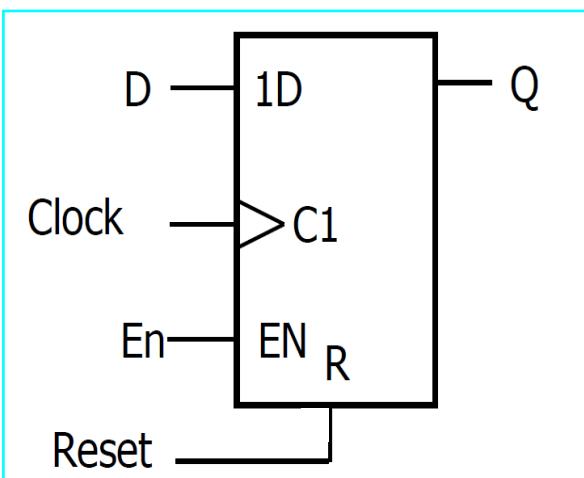
entity DFF_nq_o is
port(d_i,clk_i,reset_i : in Std_Loic;
      q_o, nq_o : out Std_Loic
);
end DFF_nq_o;

architecture behav of DFF_nq_o is

signal q_s : std_logic;
begin
  -- le signal q_s ne doit pas être défini si nq_o n'existe pas
process(clk_i, reset_i)
begin
  if reset_i = '1' then
    q_s <= '0';
  elsif rising_edge(clk_i) then
    q_s <= d_i;
  end if;
end process;
q_o <= q_s;
nq_o <= not(q_s);
end architecture;
```



Solution 3: DFF avec Enable



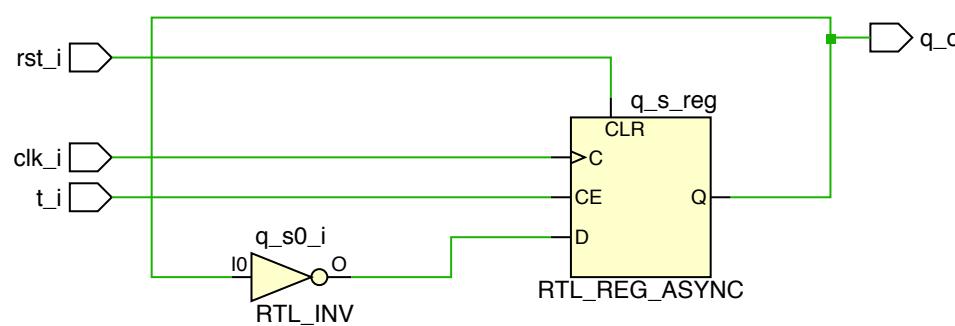
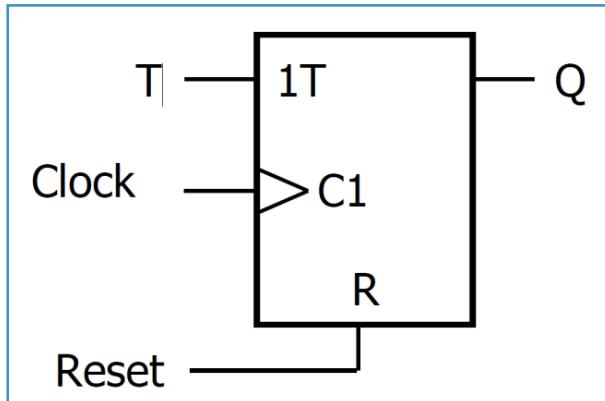
```
library IEEE;
use IEEE.std_logic_1164.all;

entity DFFE is
  port(d_i, en_i, clk_i, reset_i : IN std_logic;
        q_o: out std_logic
      );
end DFFE;

architecture behav of DFFE is
begin
  process(clk_i, reset_i)
  begin
    if reset_i = '1' then
      q_o <= '0';
    elsif rising_edge(clk_i) then
      if en_i = '1' then
        q_o <= d_i;
      end if;
    end if;
  end process;
end architecture;
```



Solution 4: Flipflop T



```

type
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FFT is
  Port ( clk_i : in STD_LOGIC;
         rst_i : in STD_LOGIC;
         t_i : in STD_LOGIC;
         q_o : out STD_LOGIC);
end FFT;

architecture Behavioral of FFT is
  signal q_s : std_logic;
begin
  process(clk_i, rst_i)
  begin
    if rst_i='1' then
      q_s <= '0';
    elsif rising_edge(clk_i)
    then
      if t_i = '1' then
        q_s <= not q_s;
      end if;
    end if;
  end process;
  q_o <= q_s;
end Behavioral;
  
```

