

# EXERCICES

## SOLUTION QUESTION 1

```
entity Multiplexeur is
port (
    in0, in1, in2, in3, in4 : in std_logic_vector(7
downto 0);
    Selecteur                : in std_logic_vector(2
downto 0);
    Sortie                   : out std_logic_vector(7
downto 0));
end Multiplexeur;

architecture CaseProcess of Multiplexeur is
begin
process(all)
```

# EXERCICES

## SOLUTION QUESTION 1

```
begin
  case Selecteur is
    when "000" => Sortie <= in0;
    when "001" => Sortie <= in1;
    when "010" => Sortie <= in2;
    when "011" => Sortie <= in3;
    when "100" => Sortie <= in4;
    when others => Sortie <= (others => '0');
  end case;
end process;
end CaseProcess;
```

# EXERCICES

## SOLUTION QUESTION 3

```
entity Decodeur is
port(
    Entree : in std_logic_vector(1 downto 0);
    Sortie : out std_logic_vector(3 downto 0));
end Decodeur;
architecture ConcSelect of Decodeur is
begin
    process(all)
    begin
        case Entree is
            when "00"    => Sortie <= "0001";
            when "01"    => Sortie <= "0010";
            when "10"    => Sortie <= "0100";
            when "11"    => Sortie <= "1000";
```

# EXERCICES

## SOLUTION QUESTION 3

```
        when others => Sortie <= "0000";  
end process;  
end ConcSelect;
```

# EXERCICES

## SOLUTION QUESTION 4

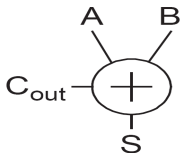
```
entity Encodeur is
port(
    Entree : in  std_logic_vector(2 downto 0);
    Sortie : out std_logic_vector(1 downto 0));
end Encodeur;

architecture ProConc of Encodeur is begin
    Sortie(0) <= Entree(2) or (not Entree(2) and not
        Entree(1) and Entree(0));
    Sortie(1) <= Entree(2) or Entree(1);
end ProConc;
```

# ADDITIONNEUR

## FONCTION ET SPÉCIFICATIONS

### Half adder



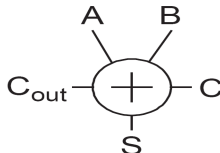
A	B	$C_{OUT}$	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$G = A \cdot B$$

$$P = A \oplus B$$

$$K = \bar{A} \cdot \bar{B}$$

### Full adder



A	B	C	G	P	K	$C_{OUT}$	S
0	0	0	0	0	1	0	0
		1				0	1
0	1	0	0	1	0	0	1
		1				1	0
1	0	0	0	1	0	0	1
		1				1	0
1	1	0	1	0	0	1	0
		1				1	1

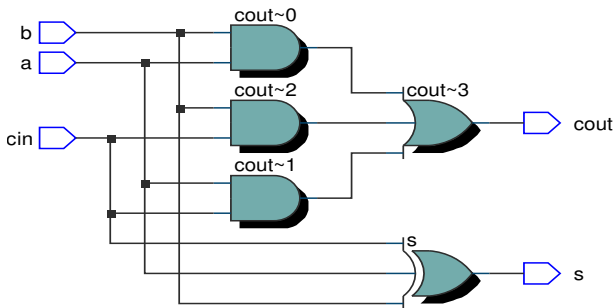
# ADDITIONNEUR

## DESCRIPTION VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity additionneur is
port(
    a,b,cin  : in  std_logic;
    s,cout    : out std_logic);
end additionneur;
--end;
architecture archConc of additionneur is
begin
    s      <= a xor b xor cin;
    cout <= (a and b) or (a and cin) or (b and cin);
end archConc;
```

# ADDITIONNEUR

## VUE DU NIVEAU TRANSFERT DE REGISTRE (RTL)





# SOUSTRACTEUR

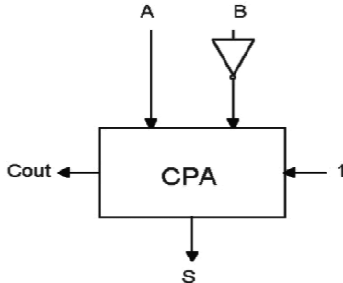
Principe :

- Complémenter à 2 l'opérande à soustraire et réaliser une addition du résultat avec l'autre opérande

Exemple :

Soustracteur

$$A - B = A + (-B) = A + \overline{B} + 1$$

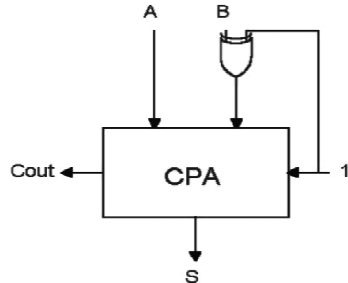


Additionneur/soustracteur

mode addition :  $sub = 0$

mode soustraction :  $sub = 1$

$$A \pm B = A + (B \oplus sub) + sub$$



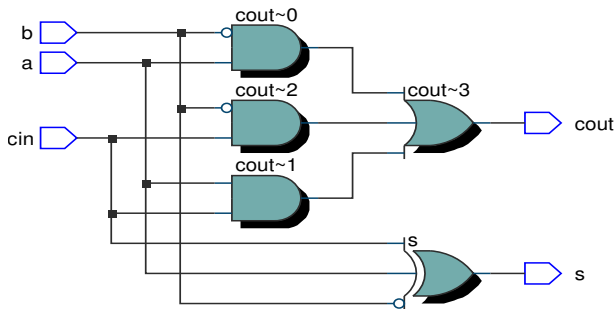
# SOUSTRACTEUR

## DESCRIPTION VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity soustracteur is
port(
    a,b,cin    : in  std_logic;
    s,cout     : out std_logic);
end soustracteur;
--end;
architecture archConc of soustracteur is
    signal bn :std_logic;
begin
    bn    <= not b;
    s     <= a xor bn xor cin;
    cout <= (a and bn) or (a and cin) or (bn and cin);
end archConc;
```

# SOUSTRACTEUR

## VUE DU NIVEAU DE TRANSFERT DE REGISTRES (RTL)



# COMPAREUR

## FONCTIONALITÉ ET SPÉCIFICATIONS

### Principe :

- Calculer si  $A = B$  ou si  $A \geq B$  permet de déterminer les autres résultats de comparaison

$$EQ = (A = B) \quad NE = (A \neq B) = \overline{EQ} \quad GT = (A > B) = \overline{EQ} \cdot GE$$

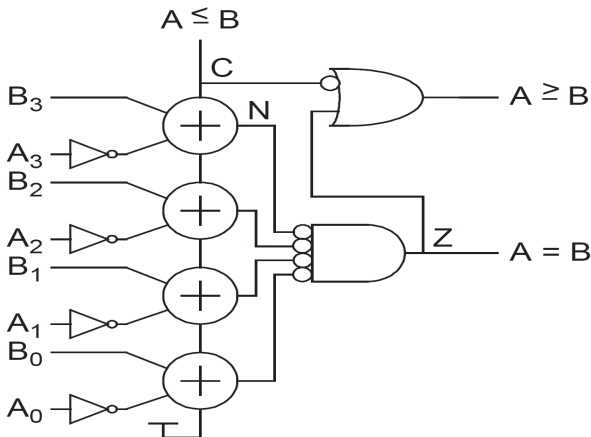
$$GE = (A \geq B) \quad LT = (A < B) = \overline{GE} \quad LE = (A \leq B) = \overline{GE} + EQ$$

- $EQ = (A = B) = (B - A = 0)$
- $EQ_{i+1} = (A_i = B_i) \cdot EQ_i = \overline{(A_i \oplus B_i)} \cdot EQ_i$
- $EQ_0 = 1; EQ_n = EQ$  ou utiliser un soustracteur optimisé (sans sortie  $S_i$ )
- $GE = (A \geq B) = (A - B \geq 0)$
- $GE_{i+1} = (A_i > B_i) + (A_i = B_i) \cdot GE_i = A_i \cdot \overline{B_i} + \overline{(A_i \oplus B_i)} \cdot GE_i$
- $GE_0 = 1; GE_n = GE$  ou utiliser un soustracteur

# COMPAREUR

## FONCTIONALITÉ ET SPÉCIFICATIONS

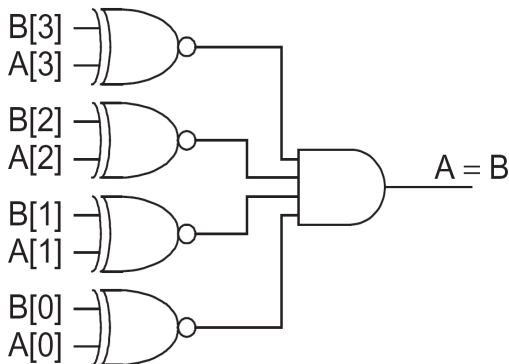
- Utilisation d'un soustracteur modifié



# COMPAREUR

## FONCTIONALITÉ ET SPÉCIFICATIONS

- Si on veut calculer uniquement si  $A=B$ , le circuit est plus simple



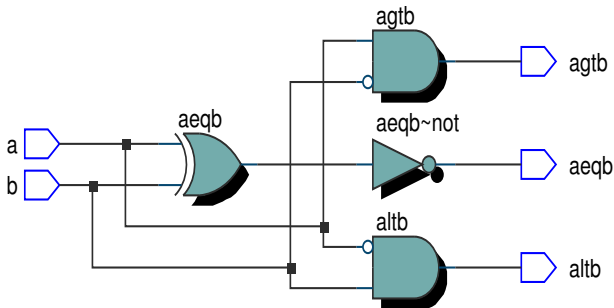
# COMPAREUR

## DESCRIPTION VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity Compareur is
port(
    a,b           : in std_logic;
    agtb, altb, aeqb : out std_logic);
end Compareur;
architecture archConc of Compareur is
begin
    agtb <= '1' when a > b else '0';
    altb <= '1' when a < b else '0';
    aeqb <= '1' when a = b else '0';
end archConc;
```

# COMPAREUR

## DESCRIPTION VHDL





# COMPAREUR

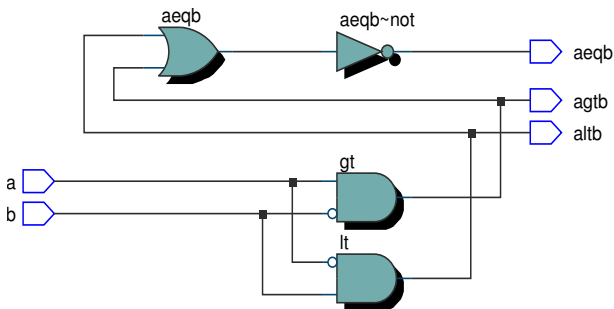
## DESCRIPTION VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity Compareur is
port(
    a,b           : in std_logic;
    agtb, altb, aeqb : out std_logic);
end Compareur;
architecture archConc1 of Compareur is
    signal gt, lt :std_logic;
begin
    gt <= '1' when a > b else '0';
    lt <= '1' when a < b else '0';
    agtb <= gt;
```

# COMPAREUR

## DESCRIPTION VHDL

```
altb <= lt;  
aeqb <= not (gt or lt);  
end archConc1;
```



# COMPAREUR

## DESCRIPTION VHDL

- ❑ Quelles sont les différences au niveau RTL entre ces deux descriptions VHDL ?
- ❑ A votre avis, laquelle des deux est la plus rapide ?

# SOMMAIRE

- ④ VHDL : utilisation du langage
  - Description des fonctions combinatoires usuelles
  - Types de données

# TYPES DE DONNÉES

- Définition d'un type de données
  - ▷ un ensemble de valeurs pouvant être affectées à un objet
  - ▷ un ensemble d'opérations pouvant être appliquées sur les objets d'un même type
- VHDL est un langage très typé
- un objet peut être affecté uniquement avec la valeur du même type
- uniquement les opérations définies pour un type de données peuvent être appliquées sur un objet de même type

Types de données standard :

- entier (**integer**) :
  - ▷ de  $-(2^{31} - 1)$  à  $2^{31} - 1$  par défaut (au maximum)
  - ▷ deux sous-types : **natural** et **positive**

# TYPES DE DONNÉES

- ▷ possibilité de spécifier un domaine avec **range** :  
`variable UnEntier : integer range 0 to 511;`
- ▷ possibilité d'accéder aux limites du domaine avec des attributs :  
`signal limB, limH : natural;`  
`begin limB <= UnEntier'Left; limH <=`  
`UnEntier'Right;`  
⇒ limB vaut 0 et limH vaut 511
- booléen (boolean) : (false, true)
- bit : ('0', '1')
- bit\_vector : un tableau 1D de bits

# TYPES DE DONNÉES

operator	description	data type of operand a	data type of operand b	data type of result
<b>a ** b</b>	exponentiation	integer	integer	integer
<b>abs a</b>	absolute value	integer		integer
<b>not a</b>	negation	boolean, bit, bit_vector		boolean, bit, bit_vector
<b>a * b</b>	multiplication	integer	integer	integer
<b>a / b</b>	division			
<b>a mod b</b>	modulo			
<b>a rem b</b>	remainder			
<b>+ a</b>	identity	integer		integer
<b>- a</b>	negation			
<b>a + b</b>	addition	integer	integer	integer
<b>a - b</b>	subtraction			
<b>a &amp; b</b>	concatenation	1-D array, element	1-D array, element	1-D array

# TYPES DE DONNÉES

a <b>sl</b> b	shift left logical	bit_vector	integer	bit_vector
a <b>sr</b> b	shift right logical			
a <b>sla</b> b	shift left arithmetic			
a <b>srl</b> b	shift right arithmetic			
a <b>rol</b> b	rotate left			
a <b>ror</b> b	rotate right			
a = b	equal to	any	same as a	boolean
a /= b	not equal to			
a < b	less than	scalar or 1-D array	same as a	boolean
a <= b	less than or equal to			
a > b	greater than			
a >= b	greater than or equal to			
a <b>and</b> b	and	boolean, bit,	same as a	boolean, bit,
a <b>or</b> b	or	bit_vector		bit_vector
a <b>xor</b> b	xor			
a <b>nand</b> b	nand			
a <b>nor</b> b	nor			
a <b>xnor</b> b	xnor			



# TYPES DE DONNÉES

## `std_logic`

- Pourquoi le type `bit` n'est pas suffisant ?
- le package `std_logic_1164`
- `std_logic` :
- 9 valeurs possibles :
  - ▷ '0' ou '1'
  - ▷ 'Z' : l'état de haute impédance
  - ▷ 'L' ou 'H' : un faible '0' ou '1'
  - ▷ 'X', 'W' : inconnu ou un faible inconnu
  - ▷ 'U' : non initialisé
  - ▷ '-' : peu importe (indéfini)
- `std_logic_vector` : vecteur de `std_logic`  
→ `std_logic_vector(7 downto 0)`

# TYPES DE DONNÉES

## std\_logic

- utilisation :

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


Opérateurs utilisés avec le type std\_logic

overloaded operator	data type of operand a	data type of operand b	data type of result
<b>not</b> a	std_logic_vector std_logic		same as a
a <b>and</b> b			
a <b>or</b> b			
a <b>xor</b> b	std_logic_vector	same as a	same as a
a <b>nand</b> b	std_logic		
a <b>nor</b> b			
a <b>xnor</b> b			

# TYPES DE DONNÉES

## std\_logic

Les fonctions de conversion disponibles dans le package :

function		data type of operand a	data type of result
to_bit(a)		std_logic	bit
to_stdulogic(a)		bit	std_logic
to_bit_vector(a)		std_logic_vector	bit_vector
to_stdlogicvector(a)		bit_vector	std_logic_vector

Exemple :

# TYPES DE DONNÉES

## std\_logic

```
signal s1, s2, s3: std_logic_vector(7 downto 0);  
signal b1, b2      : bit_vector(7 downto 0);
```

--KO

```
s1 <= b1;  
b2 <= s1 and s2;  
s3 <= b1 or s2;
```

--OK

```
s1 <= to_stdlogicvector(b1);  
b2 <= to_bitvector(s1 and s2);  
s3 <= to_stdlogicvector(b1 or s2);  
-- ou  
s3 <= to_stdlogicvector(b1 or to_bitvector(s2));
```

## OPÉRATEURS SUR LES TABLEAUX D'ÉLÉMENTS

- Les opérandes n'ont pas toujours la même taille
- Lors de la comparaison de deux tableaux n'ayant pas la même taille, tous les éléments sont comparés un par un en partant de gauche (LSB)
- Exemple :  
"011"="011", "011">"010", "011">"00010", "0110">"011"  
Tous les exemples précédents sont vrais
- Opérateur de concaténation  
y<= "00" & a(7 downto 2);  
y<= a(7) & a(7) & a(7 downto 2);  
y<= a(1 downto 0) & a(7 downto 2);

## OPÉRATEURS SUR LES TABLEAUX D'ÉLÉMENTS

### □ Opération d'agrégation

```
a <= "10100000";  
a <= (7=> '1', 6=> '0', 0=> '0', 1=> '0', 5=> '1',  
      4=> '0', 3=> '0', 2=> '1');  
a <= (7|5=> '1', 6|4|3|2|1|0 => '0');  
a <= (7|5=> '1', others => '0');  
a <= "00000000";  
a <= (others => '0');
```

# TYPES DE DONNÉES

## PACKAGE `numeric_std`

- Comment réaliser les opérations arithmétiques avec les `std_logic` ?
- la solution : le package `numeric_std`
- définit un entier comme un tableau d'éléments de type `std_logic`
- Deux types : `unsigned` et `signed`
- utilisation :

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

Les opérateurs définis dans le package :

# TYPES DE DONNÉES

## PACKAGE numeric\_std

overloaded operator	description	data type of operand a	data type of operand b	data type of result
<b>abs</b> a - a	absolute value negation	signed		signed
a * b a / b a <b>mod</b> b a <b>rem</b> b a + b a - b	arithmetic operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	unsigned unsigned signed signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	boolean boolean boolean boolean



# TYPES DE DONNÉES

PACKAGE numeric\_std

```
signal a,b,c,d: unsigned (7 downto 0);  
...  
a <= b + c;  
d <= b + 1;  
e <= (5 + a + b) - c;
```

# TYPES DE DONNÉES

## PACKAGE numeric\_std

function	description	data type of operand a	data type of operand b	data type of result
shift_left(a,b)	shift left	unsigned, signed	natural	same as a
shift_right(a,b)	shift right			
rotate_left(a,b)	rotate left			
rotate_right(a,b)	rotate right			
resize(a,b)	resize array	unsigned, signed	natural	same as a
std_match(a,b)	compare '-'	unsigned, signed std_logic_vector, std_logic	same as a	boolean
to_integer(a)	data type	unsigned, signed		integer
to_unsigned(a,b)	conversion	natural	natural	unsigned
to_signed(a,b)		integer	natural	signed

# TYPES DE DONNÉES

## PACKAGE numeric\_std

- Les types de données `std_logic_vector`, `unsigned` ou `signed` sont définis comme des tableaux d'éléments `std_logic`
- Ces trois types sont considérés comme des types différents
- Utilisation de fonctions de conversion pour passer d'un type à un autre

# TYPES DE DONNÉES

PACKAGE numeric\_std

data type of a	to data type	conversion function / type casting
unsigned, signed	std_logic_vector	std_logic_vector(a)
unsigned, std_logic_vector	unsigned	unsigned(a)
unsigned, signed	std_logic_vector	std_logic_vector(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

# TYPES DE DONNÉES

## PACKAGE numeric\_std

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3
    downto 0);
signal u1, u2, u3, u4, u6, u7: unsigned(3 downto 0);
signal sg: signed(3 downto 0);

-- OK
u3 <= u2 + u1;    --- ok, operandes non-signés
u4 <= u2 + 1;     --- ok, operandes non-signé et natural
```

# TYPES DE DONNÉES

PACKAGE numeric\_std

--KO

```
u5 <= sg;  -- type mismatch
```

```
u6 <= 5;   -- type mismatch
```

--Solution

```
u5 <= unsigned(sg);  -- type casting
```

```
u6 <= to_unsigned(5,4); -- fonction de conversion
```

--KO

```
s3 <= u3;  -- type mismatch
```

```
s4 <= 5;   -- type mismatch
```

--Solution

```
s3 <= std_logic_vector(u3); -- type casting
```

# TYPES DE DONNÉES

PACKAGE numeric\_std

```
s4 <= std_logic_vector(to_unsigned(5,4));
```

--KO

```
s5 <= s2 + s1; + indefini pour std_logic_vector
```

```
s6 <= s2 + 1; + indefini
```

-- Solution

```
s5 <= std_logic_vector(unsigned(s2) + unsigned(s1));
```

```
s6 <= std_logic_vector(unsigned(s2) + 1);
```

# TYPES DE DONNÉES

## PACKAGE `std_logic_arith`

- package développé par *Synopsys* avant le standard IEEE `numeric_std`
- presque similaire à `numeric_std`
- deux nouveaux types : `unsigned` et `signed`
- les détails d'implémentation sont différents
- manipule les `std_logic_vector` comme des nombres signés ou non-signés
- Dans les outils de simulation, on le trouve souvent dans la library `ieee` (même s'il n'en fait pas partie)



# TYPES DE DONNÉES

## PACKAGE std\_logic\_arith

```
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_arith_unsigned.all;  
...  
signal s1,s2,s3,s4,s5,s6: std_logic_vector(3 downto  
    0);  
...  
s5<=s2+s1; -- ok, l'opérateur + surchargé  
s6<= s2+1; -- ok, l'opérateur + surchargé
```

- un seul des deux packages peut être utilisé à la fois
- leur utilisation remet en cause la réputation du langage VHDL comme un langage très typé

# TYPES DE DONNÉES

PACKAGE std\_logic\_arith

- En conclusion : préférer le package `numeric_std`

# EXERCICES

## UTILISATION DU PACKAGE `numeric_std`

- ❶ Réaliser un additionneur 4 bits en instanciant l'additionneur 1 bit déjà présenté
- ❷ Décrire un additionneur 4 bits en utilisant le package `numeric_std`
- ❸ Décrire un soustracteur 4 bits en utilisant le package `numeric_std`
- ❹ Décrire un comparateur 8 bits en utilisant le package `numeric_std`
- ❺ Pour le comparateur 8 bits, est-il nécessaire de faire des conversions en `unsigned` ?