# VHDL 2:

## Types, Généricité et Attributs

Andres Upegui

Laboratoire de Systèmes Numériques
hepia

# Types

type: ensemble des valeurs prises par un objet et regroupées en quatre classes:

- types scalaires (scalar): entier (integer), réel (real), énuméré (enumerated), physique (physical)
- types composites (composite): tableau (array), enregistrement (record)
- type accès (access): pointeur (pointer) permettant d'accéder à des objets d'un type donné
- type fichier (file): séquence de valeurs d'un type donné

# Types scalaires: entiers

## Type prédéfini dans le paquetage standard

```
type INTEGER is range -2'147'483'648 to 2'147'483'647;
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
```

## Exemple

```
type profondeur_de_pile is range 1 to 12;
subtype profondeur_de_pile is POSITIVE range 1 to 12;
```

## Flottants

```
type nouveau_flottant is range 1.23 to 5.43;
```

# Types scalaires: énuméré

Type énuméré: caractérisation d'un objet par la liste complète de ses valeurs

## Types énumérés prédéfinis

```
type bit is ('0','1');          -- caractères
type boolean is (false,true); -- identificateurs
type character is 256 caractères du jeu ISO 8859-1;
type severity_level is (NOTE,WARNING,ERROR,FAILURE);
```

## Exemples de types et sous-types non prédéfinis

```
type logic4 is ('0','1','X',Z'); -- surcharge avec le type bit
type state is (IDLE,INIT,CHECK,DONE);
type mixed is (FALSE,'1','2',IDLE); -- surcharge avec types boolean,
                                    --  bit et states
```

## Attention

```
logic4'('1') ≠ bit'('1')
states'(IDLE) ≠ mixed'(IDLE)
```

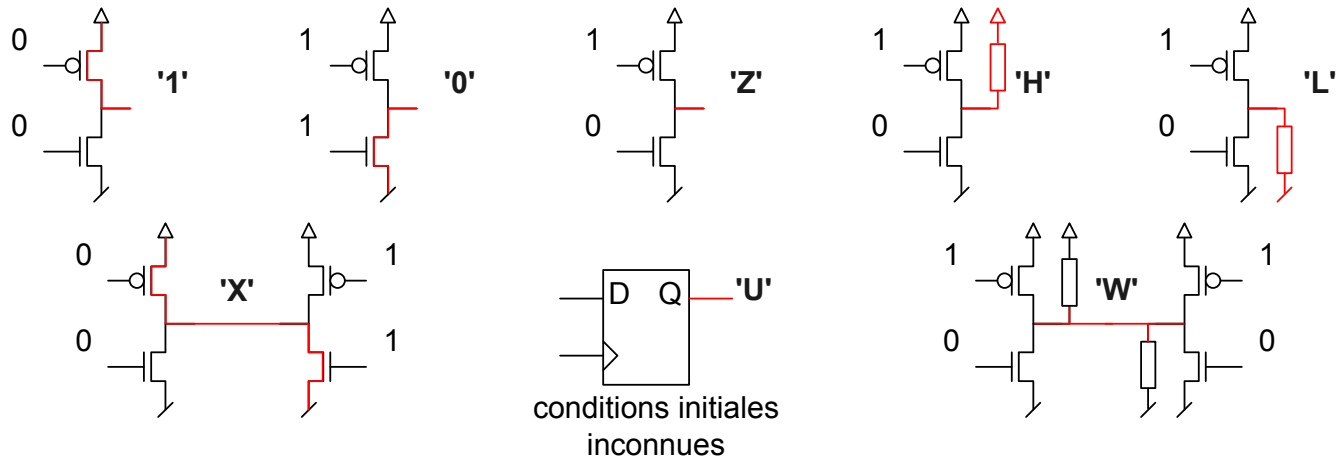
# Types scalaires énuméré: std_logic

Le type `std_logic` est largement utilisé pour la simulation et la synthèse des systèmes matériels

- `std_ulogic:`

```
type std_ulogic is (   'U', -- état non initialisé
                       'X', -- état logique indéfini fort
                       '0', -- état logique 0 fort
                       '1', -- état logique 1 fort
                       'Z', -- état à haute impédance
                       'W', -- état logique indéfini faible
                       'L', -- état logique 0 faible
                       'H', -- état logique 1 faible
                       '-', -- état logique indifférent );
```

- `std_logic:` `subtype std_logic is resolved std_ulogic;`



# Types composites: tableaux

- Type tableau (array type): type composite consistant en un groupe d'objets dont le type est identique et la position indexée.

## Exemple

```
type std_logic_vector is array (NATURAL range <>) of std_logic;
subtype mots16 is std_logic_vector(15 downto 0);
type memoire128 is array (127 downto 0) of mots16;

signal memoire: memoire128;
signal mot: mots16;
signal unbit: std_logic;

mot<=memoire(12);
unbit<=memoire(12)(4);

type BIT_VECTOR is array(NATURAL range <>) of BIT;
signal unvecteur: BIT_VECTOR(5 downto 0);
```

# Manipulation des tableaux

L'opérateur de concaténation "&":

**Exemple**

```
Vect4 <= D & C & B & A;
Vect8 <= Vect10(9 downto 6) & A & B & "00";
```

Décomposition de vecteurs:

**Exemple**

```
Vect4 <= Vect8(5 downto 2);
Signal <= Vect8(4);
Vect8(5 downto 3) <= "010";
```

# Manipulation des tableaux

Le mot clé `others` permet d'indiquer que tous les éléments du tableau sont affectés par une même valeur.

## Exemple 1

```
Vect_A <= (others => '0'); --tout à 0
Vect_C <= (others => 'Z'); --tout à Z

--Ou affecté avec le même signal My_Signal
Vect_4 <= (others => My_Signal);
```

## Exemple 2

```
--affectation : MSB à '1', autres bits à '0'
Vect8 <= (7 => '1', others => '0');

--affectation : MSB à '0', autres bits à '1'
Vect8 <= (7 => '0', others => '1');

--Affectation : LSB à '1', autres bits à '0'
Vect8 <= (0 => '1', others => '0');
```

# Affectation en hexadécimal

- En VHDL93 il est possible de donner des valeurs en hexadécimal.

**Exemple d'affectation de vecteurs:**

```
Vecteur_8Bits  <= x"6A";
Vecteur_24Bits <= x"2A_4F5C"
Vecteur_4Bits  <= x"8";
```

- Attention valable uniquement pour des vecteurs de longueurs : 4, 8, 12, 16 ...
  donc des multiples de 4 !!

# Types composites: enregistrements

Collection d'éléments nommés, ou champs, dont les valeurs peuvent être de types différents

## Exemple

```
type memory_bus is record
    addr: std_logic_vector(15 downto 0);
    data: std_logic_vector(7 downto 0);
    read, write: std_logic;
    enable: std_logic;
end record memory_bus;
```

## Accès aux éléments de l'enregistrement

```
signal MB: memory_bus;
MB.addr                 -- tout le tableau addr
MB.addr(7 downto 0)     -- une partie du tableau addr
MB.data(7)              -- bit de donnée de poids fort
```

# Paquetage Numeric_Std

Le paquetage Numeric_Std definit:

- des types pour les nombres non-signés et signés
- les opérations arithmétiques de bases (+, -)
- la comparaison entre des nombres entiers

# Paquetage Numeric_Std

## Declaration

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

Défini 2 types numériques:

- UNSIGNED : vecteur représentant un nombre non signé
- SIGNED : vecteur représentant un nombre signé

il s'agit de nombres ENTIER en binaire ces 2 types sont basés sur le Std_Logic

# Types UNSIGNED et SIGNED

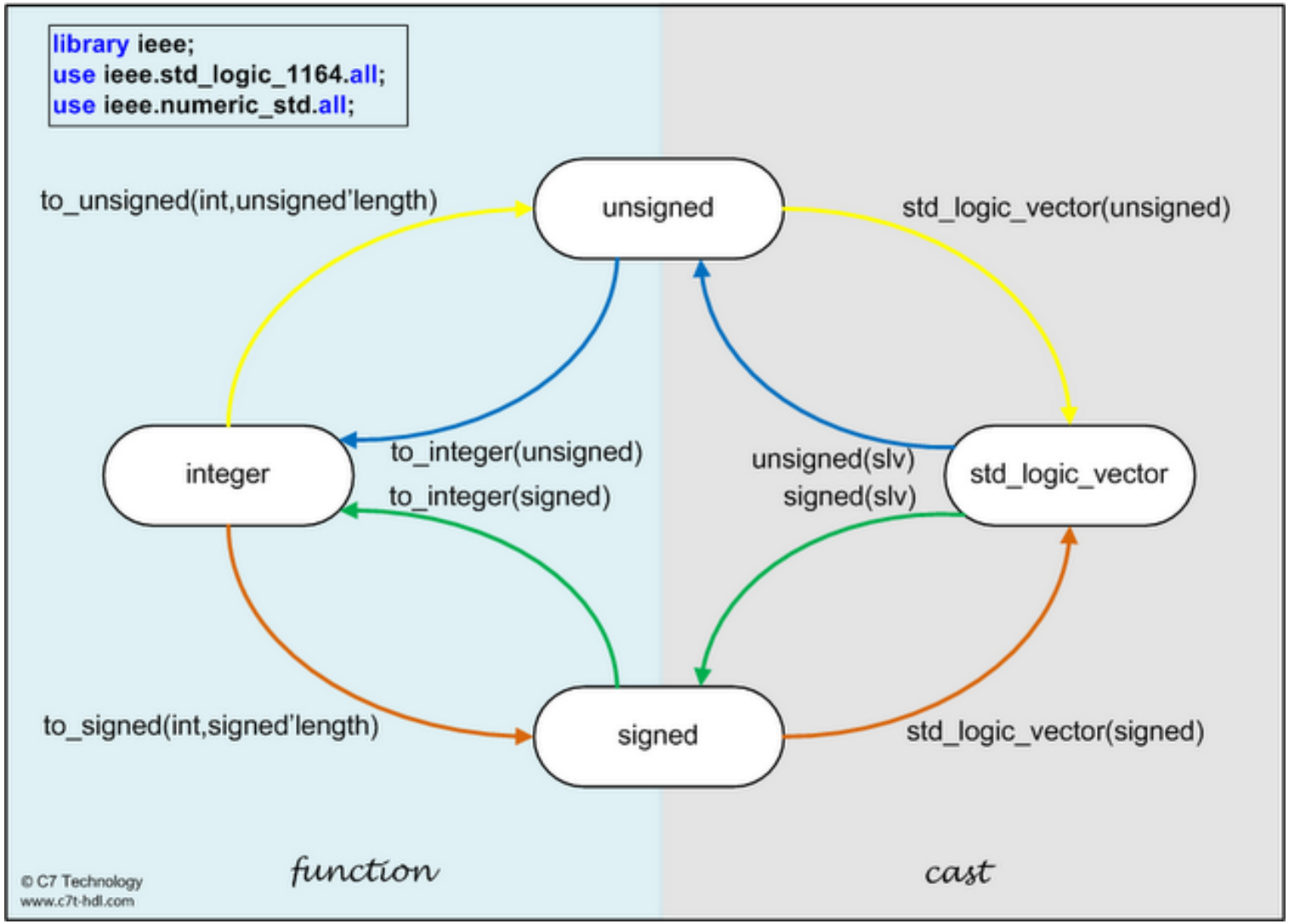
## Définition:

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED   is array (NATURAL range <>) of STD_LOGIC;
```

## Déclaration des signaux:

```
signal Nombre_signe       : Unsigned(7 downto 0);
signal Nombre_non_signe   : Signed(7 downto 0);
signal Vecteur            : Std_Logic_Vector(7 downto 0);
```

# Adaptation de type (cast)



# Adaptation de type (cast)

## Exemple

```
-- Déclaration des signaux :
signal Vecteur : Std_Logic_Vector(7 downto 0);
signal Nombre  : Unsigned(7 downto 0);

--Exemples d'adaptation de type (cast):
Nombre  <= Unsigned(Vecteur);
Vecteur <= Std_Logic_Vector(Nombre);

--Basé sur le Std_Logic, donc :
Vecteur(i) <= Nombre(i);   --correct
Nombre(i)  <= Vecteur(i);  --correct
```

# Fonctions de conversion ...

## Fonction To_Integer :

```
To_Integer (ARG: Unsigned) return Natural
To_Integer (ARG: Signed) return Integer
```

## Fonction To_Unsigned :

```
To_Unsigned (ARG, SIZE: Natural) return Unsigned
```

## Fonction To_Signed :

```
To_Signed (ARG: Integer; SIZE: Natural) return Signed
```

# Fonctions "+" et "-"

Si les 2 opérandes ont de tailles différentes, le résultat aura la taille du plus grand Result :MAX(L'LENGTH, R'LENGTH)-1 downto 0

## Exemples

```
 -- Déclaration des signaux :
signal N_A, N_B : Unsigned(7 downto 0);
signal Somme    : Unsigned(7 downto 0);

--Exemples d'addition :
Somme <= N_A + N_B;
Somme <= N_A + "0001";
Somme <= N_B + 1;
Somme <= N_A + "00110011";
--Erroné : Somme <= N_B + '1';
--Erroné : Somme <= N_B + "0000110011";
```

# Fonctions de comparaison

Les opérandes peuvent avoir des types différentes Résultat de type booléen

## Exemples

```
--déclaration des signaux
signal nSgn_A, nSgn_B : Unsigned(3 downto 0);

--résultat de la comparaison est un booléen
  nSgn_A =  "1100"  ou  nSgn_A = 12

  nSgn_A /= nSgn_B
```

## Exemples

```
--déclaration des signaux
signal Sgn_A, Sgn_B : Signed(3 downto 0);

--résultat de la comparaison est un booléen
  Sgn_A < "1100"  ou   Sgn_A < -4

  Sgn_A > Sgn_B
```

# Généricité

- La généricité permet de transmettre une information statique à un bloc.
- Un bloc générique est vu de l'extérieur comme un bloc paramétré.
- A l'intérieur du bloc, les paramètres génériques sont identiques à des constantes.
- Une entité peut être générique, mais pas une architecture.

## Syntaxe

```
generic (param1 [, autre_param]: type_param [:=valeur_par_defaut];
         param2 [, autre_param]: type_param [:=valeur_par_defaut];
         ...
         paramN [, autre_param]: type_param [:=valeur_par_defaut]);
```

# Généricité: exemple

- Une porte ET à N entrées

## Porte ET

```
entity porteET is
    generic ( NB_ENTREES: natural :=2);
    port ( entrees: in std_logic_vector(NB_ENTREES downto 1);
           sortie: out std_logic);
end porteET;

architecture comp of porteET is
begin
    process(entrees)
        variable resultat: std_logic;
    begin
        resultat:='1';
        for i in 1 to NB_ENTREES loop
            resultat := resultat and entrees(i);
        end loop;
        sortie <= resultat;
    end process;
end comp;
```

# Généricité: exemple instancié

## Instanciation de la Porte ET

```
architecture struct of quelquechose is

component porteET is
    generic ( NB_ENTREES: natural :=2);
    port ( entrees: in std_logic_vector(NB_ENTREES downto 1);
           sortie: out std_logic);
end component;

signal lesentrees: std_logic_vector(7 downto 0);
signal lasortie: std_logic;
...
begin

porte: porteET
generic map ( NB_ENTREES => 8)
port map ( entrees => lesentrees,
           sortie => lasortie);
...
end struct;
```

# Généricité: exemple 2

## Multiplexeur de N bits

# Généricité: exemple 2

## Multiplexeur de N bits

```
entity multiplexeur is
generic ( SIZE: positive := 1);
port ( entree0: in std_logic_vector(SIZE-1 downto 0);
       entree1: in std_logic_vector(SIZE-1 downto 0);
       sortie:out std_logic_vector(SIZE-1 downto 0);
       sel: in std_logic);
end multiplexeur;
```

# Généricité: exemple 2

## Multiplexeur de N bits

```
entity multiplexeur is
generic ( SIZE: positive := 1);
port ( entree0: in std_logic_vector(SIZE-1 downto 0);
       entree1: in std_logic_vector(SIZE-1 downto 0);
       sortie:out std_logic_vector(SIZE-1 downto 0);
       sel: in std_logic);
end multiplexeur;
architecture flot of multiplexeur is
begin
    sortie<= entree0 when sel='0' else
             entree1;
end flot;
```

# Généricité: exemple 2

## Multiplexeur de N bits

```
entity multiplexeur is
generic ( SIZE: positive := 1);
port ( entree0: in std_logic_vector(SIZE-1 downto 0);
       entree1: in std_logic_vector(SIZE-1 downto 0);
       sortie:out std_logic_vector(SIZE-1 downto 0);
       sel: in std_logic);
end multiplexeur;
architecture flot of multiplexeur is
begin
    sortie<= entree0 when sel='0' else
             entree1;
end flot;
architecture comp of multiplexeur is
begin
    process(entree0,entree1,sel)
    begin
        for i in 0 to SIZE-1 loop
          sortie(i)<=(entree0(i) and (not sel)) or (entree1(i) and sel);
        end loop;
    end process;
end comp;
```

# Attributs d'un tableau

- Un attribut est une caractéristique associée à un type ou à un objet.
- Les attributs d'un tableau T s'utilisent de la façon suivante:
  `T'nom_attribut(numero_dimension)`
- Le numéro de la dimension peut être omis et vaut dans ce cas 1.
- Les attributs suivants sont définis sur n'importe quel type tableau:
  - `LEFT`: élément le plus à gauche de l'intervalle de l'index
  - `RIGHT`: élément le plus à droite de l'intervalle de l'index
  - `HIGH`: élément le plus grand de l'index
  - `LOW`: élément le plus petit de l'index
  - `RANGE`: sous-type des indices, intervalle entre l'attribut `LEFT` et `RIGHT`
  - `REVERSE_RANGE`: intervalle inverse de `RANGE`
  - `LENGTH`: nombre d'éléments du tableau

# Attributs d'un tableau: exemple

Soit le code suivant:

```
type index1 is range 1 to 20;
type index2 is range 19 downto 2;
type vecteur1 is index1 of std_logic;
type vecteur2 is index2 of std_logic;
```

Déterminer les valeurs des attributs suivants:

| Attribut | vecteur1 | vecteur2 |
|---|---|---|
| LEFT | | |
| RIGHT | | |
| HIGH | | |
| LOW | | |
| RANGE | | |
| REVERSE_RANGE | | |
| LENGTH | | |

# Attributs d'un tableau: exemple

Soit le code suivant:

```
type index1 is range 1 to 20;
type index2 is range 19 downto 2;
type vecteur1 is index1 of std_logic;
type vecteur2 is index2 of std_logic;
```

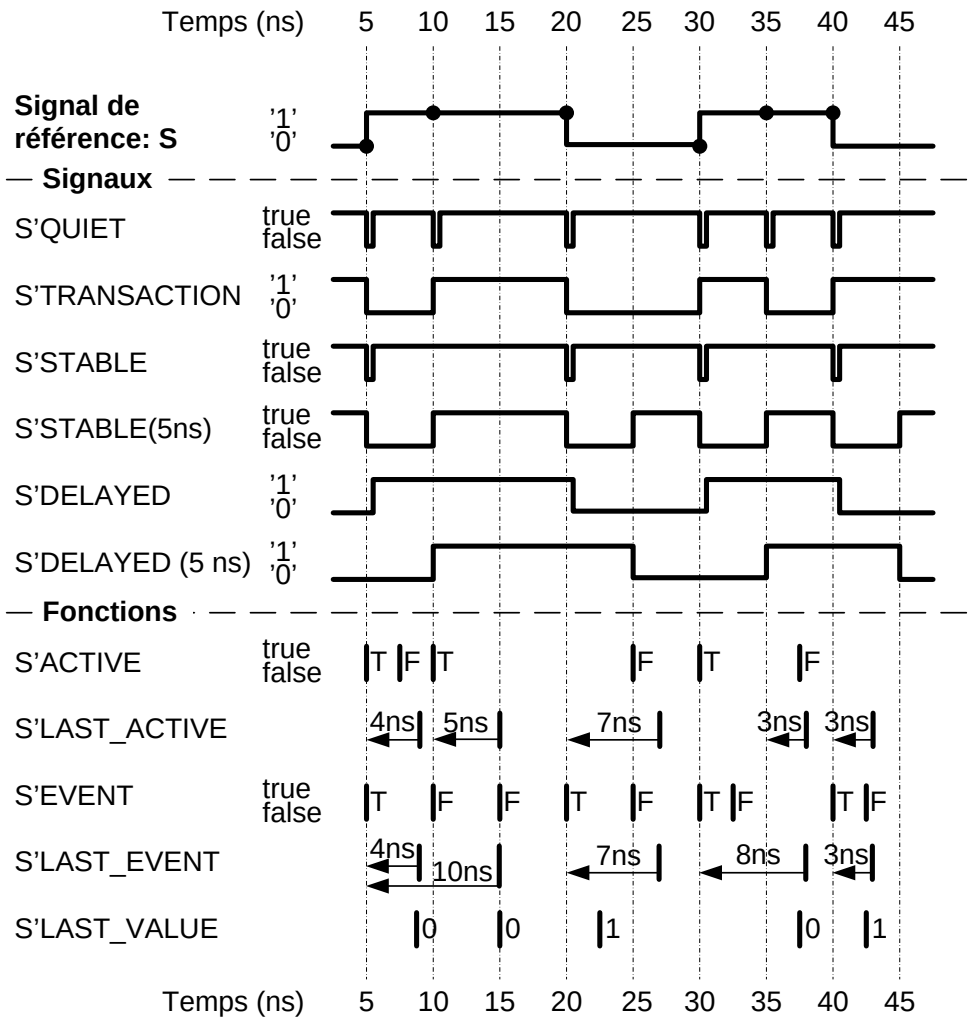
Déterminer les valeurs des attributs suivants:

| Attribut | vecteur1 | vecteur2 |
|---|---|---|
| LEFT | 1 | 19 |
| RIGHT | 20 | 2 |
| HIGH | 20 | 19 |
| LOW | 1 | 2 |
| RANGE | 1 to 20 | 19 downto 2 |
| REVERSE_RANGE | 20 downto 1 | 2 to 19 |
| LENGTH | 20 | 18 |

# Attributs d'un signal

- `S'event`: TRUE si un événement vient d'affecter S, sinon FALSE
- `S'last_event`: Temps écoulé depuis le dernier événement sur S
- `S'last_value`: Valeur de S avant son dernier changement
- `S'delayed(T)`: Image de S décalée dans le temps d'une quantité T
- `S'stable(T)`: TRUE si S n'a pas subit d'événement depuis un temps T
- `S'quiet(T)`: TRUE si S n'a pas été actif depuis un temps T
- `S'transaction`: Retourne un signal de type BIT qui change d'état à chaque transaction sur S
- `S'active`: TRUE si S est actif au temps actuel
- `S'last_active`: Temps écoulé depuis la dernière activation de S
- `S'driving`: FALSE si la transaction dans le process en cours est null
- `S'driving_value`: Valeur de S pour la transaction dans le process en cours

# Attributs d'un signal: exemple



# Attributs d'un signal: exemple

- Deux fonctions de la librairie IEEE.STD_LOGIC_1164

## Exemple

```
function rising_edge  (signal s : std_ulogic) return BOOLEAN is
begin
   return (s'EVENT and (To_X01(s) = '1') and
                       (To_X01(s'LAST_VALUE) = '0'));
end;

function falling_edge (signal s : std_ulogic) return BOOLEAN is
begin
   return (s'EVENT and (To_X01(s) = '0') and
                       (To_X01(s'LAST_VALUE) = '1'));
end;
```