Cours de Systèmes numériques

Ch. 2 : Types, Opérateurs & Conversions

Nicolas.Schroeter@hefr.ch

Introduction

Pour écrire des composants VHDL, il est essentiel de connaître:

- Quels sont les types prédéfinis du VHDL ?
- Comment spécifier des nouveaux ?
- Comment les utiliser ?
- Comment convertir la donnée d'un type à un autre ?
- Quels opérateurs sont utilisables ?

Définitions

- VHDL est un langage **typé** où il est obligatoire de spécifier le type des **objets** utilisés.

- Un sous-type (subtype)
  - est contraint, il définit un sous-ensemble des valeurs du type père.
  - hérite des opérateurs du type père.

NB :

L'utilisation des sous-types permet de diminuer la quantité de ressources utilisées dans le circuit programmable.

Compatibilité des types

- En VHDL, tous les types sont incompatibles entre eux:
  - →une expression ou une affectation est acceptée par les outils VHDL uniquement si leur type est compatible.
  - →Les fonctions de conversion permettent de convertir un objet dans un autre type

- Les sous-types sont compatibles avec le type père

Types prédéfinis

- Ils existent des types prédéfinis dans le langage VHDL qui sont spécifiés dans les standards:
  - IEEE 1076
  - IEEE 1164
- Plus précisément, les définitions de ces types sont déclarées dans les packages/library existants;

  → à importer dans vos composants.

Types

VHDL définit quatre classes de types:

- Types scalaires (scalar): **énuméré** (enumerated)
  entier (integer)
  réel (real)
  physique (physical)
- Types composites: **tableau** (array)
  enregistrement (record)
- Type accès (access):
  - pointeur pour accéder à des objets d'un type donné
- Type fichier (file):
  - séquence de valeurs d'un type donné stockées dans un fichier

Types scalaires: entiers

- Types prédéfinis du 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;
```

- Exemples de définition:

```
type short is range -128 to 127;
subtype nat4 is natural range 0 to 15;
subtype offset is nat4 range 14 to 15;
```

NB:

Un sous-type d'un sous-type est un sous-type du type père.

Tous les types/sous-types INTEGER incluent, à cause du codage, la valeur 0, même si l'intervalle ne le contient pas.

Types scalaires: physique

- Type time est défini par son intervalle de valeurs, son unité de base et ses sous-unités:

```
type time is range -2147483647 to 2147483647
-- pour exprimer le temps en femtosecondes
units
    fs; -- unité de base
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr = 60 min;
end units;
```

Types scalaires: énuméré

- Type énuméré: liste ordonnée de valeurs.
- Types énumérés prédéfinis du paquetage standard :

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

- Exemples de définition:

```
type logic is ('0', '1', 'Z', 'X');
type etats is (idle, treat, done); --machines d'états
type mixed is ('0', idle, 'c', done); -- car + ident.
```

NB :

L'utilisation de valeurs énumérées simplifie la lecture du code.

Le synthétiseur est responsable du codage (1 parmi n, binaire, Gray, etc.).

Types scalaires énuméré: std_logic

- Type std_logic: type de base à utiliser pour les signaux logiques défini dans le paquetage 1164.

```
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 );

subtype std_logic is resolved std_ulogic;
```

Types scalaires énuméré: std_logic

- La plupart des valeurs du type std_logic sont prévus pour la simulation uniquement.

- Trois valeurs sont synthétisables sans restriction: '0', '1' and 'Z'

- Lors d'une simulation,
  - 'U' indique des signaux non-initialisés
  - 'X' permet de détecter d'éventuel conflit dans des affectations:

    Si plusieurs valeurs logiques sont connectées sur le même signal, le conflit est résolu grâce à la table suivante

STD_LOGIC: table de résolution

- Lors de la simulation, la table de résolution permet de résoudre d'éventuel conflits, plusieurs valeurs std_logic connectées sur un même signal.

```
-- ---------------------------------------------------------
-- U    X    0    1    Z    W    L    H    - |
-- ---------------------------------------------------------
( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
);
```

STD_LOGIC: fonction de résolution

```
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
        VARIABLE result : std_ulogic := 'Z';  -- weakest state default
    BEGIN
        -- the test for a single driver is essential otherwise the
        -- loop would return 'X' for a single driver of '-' and that
        -- would conflict with the value of a single driver unresolved
        -- signal.
        IF     (s'LENGTH = 1) THEN    RETURN s(s'LOW);
        ELSE
            FOR i IN s'RANGE LOOP
                result := resolution_table(result, s(i));
            END LOOP;
        END IF;
        RETURN result;
    END resolved;
```

Attributs des types scalaires

- Un attribut est une caractéristique pour obtenir de l'information associée à un type ou à un objet.
- Dans le cas des types scalaires, les attributs informent sur l'ordre ou la position des valeurs dans l'énuméré.

NB :

Il est préférable d'utiliser un attribut pour faire référence à une valeur au lieu de la valeur elle-même, car si les valeurs du type changent, la référence pointera toujours vers la bonne valeur.

Attributs des types scalaires

- Attributs selon l'ordre de déclaration dans l'énuméré:
  - LEFT: élément le plus à gauche de l'énuméré
  - RIGHT: élément le plus à droite de l'énuméré
  - LEFTOF(value): élément à gauche de la valeur
  - RIGHTOF(value): élément à droite de la valeur
  - POS(value): position dans l'énuméré, en débutant par 0 pour l'élément le plus à gauche
  - VAL(position): valeur à la position de l'intervalle
- Attributs selon l'ensemble des valeurs:
  - HIGH: valeur la plus grande dans l'énuméré
  - LOW: valeur la plus petite dans l'énuméré
  - PRED(value): valeur précédente
  - SUCC(value): valeur suivante

Attributs des types scalaires

Exercices :

```
type state is (main_green, main_yellow, farm_green, farm_yellow);
type short is range -128 to 127;
type backward is range 127 downto -128;
```

Enoncé	Réponse	Enoncé	Réponse
state'left		state'low	
state'right		state'high	
short'left		short'low	
short'right		short'high	
backward'left		backward'low	
backward'right		backward'high	
state'pos(main_green)		state'val(3)	
state'succ(main_green)		short'pred(0)	
short'leftof(0)		backward'pred(0)	
backward'leftof(0)		state'succ(farm_yellow)	

Types composite: tableaux

- Type Array: un tableau est un ensemble d'objets de même type accessible avec une position indexée, entière ou valeur énumérée.
- Exemples:

```
--déclaration d'un type tableau peut être sans contrainte
type std_logic_vector is array (NATURAL range <>) of std_logic;
subtype mot is std_logic_vector(7 downto 0);
type memoire is array(1023 downto 0) of mot;
signal memory : memoire;

type bit_vector is array (NATURAL range <>) of bit;
--pour la synthèse, un signal doit obligatoirement être contraint
signal unvecteur : bit_vector(15 downto 0);

-- position indexée avec valeurs énumérées
type couleur is (BLANC, BLEU, VERT, ROUGE, NOIR, ARC_EN_CIEL);
type PRIX_PEINTURES is array (couleur range BLANC to NOIR) of natural;
```

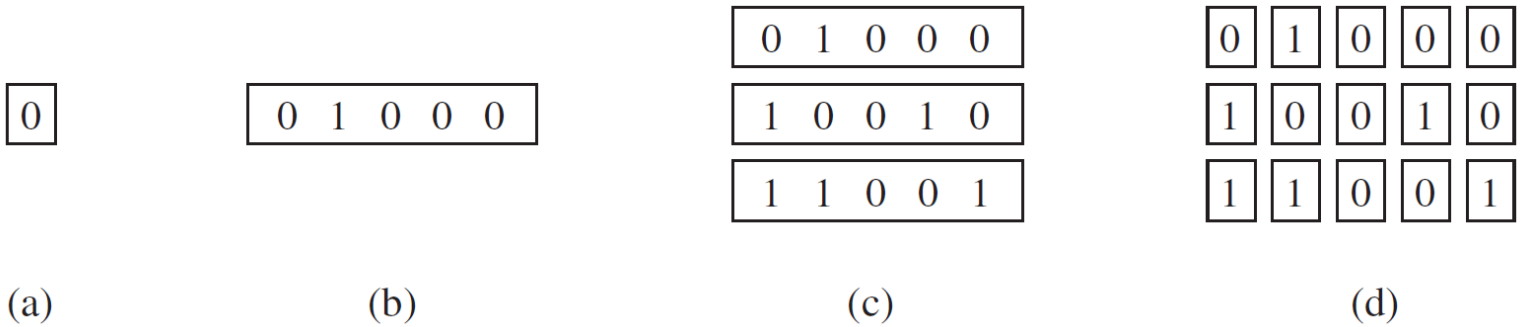
Types composite: tableaux

Cas pratiques d'utilisation:

- Il est possible de créer plusieurs types de tableaux:
  - Une dimension (1D)
  - Deux dimensions (2D)
  - Une dimension x une dimension (1Dx1D)
- Des tableaux peuvent aussi être de dimensions plus élevées, mais ne sont généralement pas synthétisables.

Types composite: tableaux

- Exemple de structure de données:
  a) Une seule valeur: un scalaire
  b) Un vecteur: un tableau 1D
  c) Un tableau de vecteurs (1Dx1D)
  d) Un tableau de scalaires (2D)

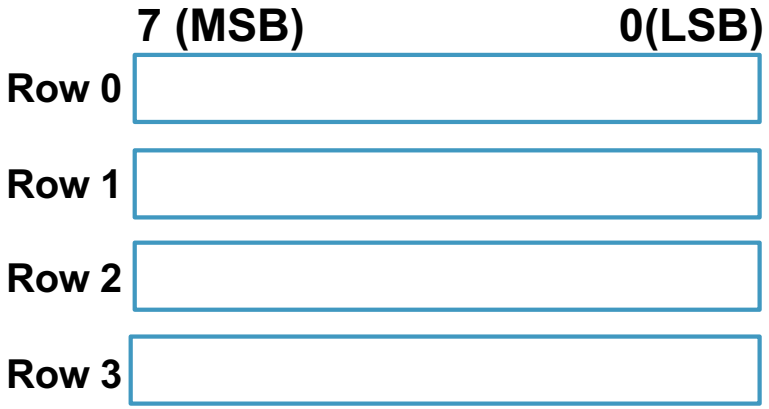


Types composite: Tableau 1Dx1D

- Exemple: Crée un tableau de quatre vecteurs de 8 bits → (1Dx1D)

```
TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC; -- 1D array
TYPE matrix IS ARRAY (0 TO 3) OF row; -- 1Dx1D array
SIGNAL x: matrix; -- 1Dx1D signal
```

- Signal x:



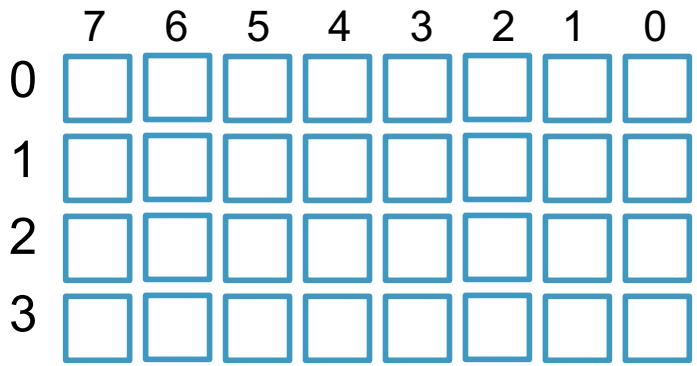
- Une autre façon de créer ce tableau:

```
TYPE matrix IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Types composite: Tableau 2D

- Il s'agit d'un tableau deux dimensions, où sa déclaration est basée sur des scalaires
- Exemple:

```
TYPE matrix2D IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
```



Types composite: exemples

```
TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;
-- 1D array
TYPE array1 IS ARRAY (0 TO 3) OF row;
-- 1Dx1D array
TYPE array2 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
-- 1Dx1D
TYPE array3 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
-- 2D array

SIGNAL x: row;
SIGNAL y: array1;
SIGNAL v: array2;
SIGNAL w: array3;
```

```
--------- Legal scalar assignments: ---------------
-- The scalar (single bit) assignments below are all legal,
-- because the "base" (scalar) type is STD_LOGIC for all signals (x,y,v,w)
x(0) <= y(1)(2); -- notice two pairs of parenthesis(y is 1Dx1D)
x(1) <= v(2)(3); -- two pairs of parenthesis (v is 1Dx1D)
x(2) <= w(2,1);  -- a single pair of parenthesis (w is 2D)
y(1)(1) <= x(6);
y(2)(0) <= v(0)(0);
y(0)(0) <= w(3,3);
w(1,1) <= x(7);
w(3,0) <= v(0)(3);
```

```
--------- Vector assignments: ---------------------
x <= y(0); -- legal (same data types: ROW)
x <= v(1); -- illegal (type mismatch: ROW x STD_LOGIC_VECTOR)
x <= w(2); -- illegal (w must have 2D index)
x <= w(2, 2 DOWNTO 0); -- illegal (type mismatch: ROW x STD_LOGIC)
v(0) <= w(2, 2 DOWNTO 0); -- illegal (mismatch: STD_LOGIC_VECTOR x STD_LOGIC)
v(0) <= w(2); -- illegal (w must have 2D index)
y(1) <= v(3); -- illegal (type mismatch: ROW x STD_LOGIC_VECTOR)
y(1)(7 DOWNTO 3) <= x(4 DOWNTO 0); -- legal (same type, same size)
v(1)(7 DOWNTO 3) <= v(2)(4 DOWNTO 0); -- legal (same type,same size)
w(1, 5 DOWNTO 1) <= v(2)(4 DOWNTO 0); -- illegal (type mismatch)
```

Types composite: Attributs d'un tableau

- Les attributs d'un type tableau T s'utilisent de la façon suivante:

  T'nom_attribut(numero_dimension)

- Le numéro de la dimension peut être omis et vaut par défaut 1.

NB :

L'utilisation d'attributs permet de rendre le code générique, pas besoin de connaitre à l'avance la taille d'un tableau.

Types composite: Attributs d'un tableau

- Les attributs suivants sont définis pour n'importe quel type de tableau:
  - LEFT: l'indice le plus à gauche de l'intervalle
  - RIGHT: l'indice le plus à droite de l'intervalle
  - HIGH: l'indice le plus grand dans l'intervalle
  - LOW: l'indice le plus petit dans l'intervalle
  - RANGE: sous-type des indices, intervalle entre l'attribut LEFT et RIGHT
  - REVERSE_RANGE: intervalle inverse de RANGE
  - LENGTH: nombre d'éléments du tableau

NB :

- Les valeurs d'indice retournées sont du type de l'intervalle (range).
- LENGTH retourne une valeur du type INTEGER

Attributs d'un tableau: exemple

- Soit les déclarations suivantes:

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

- Déterminer les valeurs des attributs suivants:

Attribut	Vecteur1	Vecteur2
LEFT		
RIGHT		
HIGH		
LOW		
RANGE		
REVERSE_RANGE		
LENGTH		

Types composites: signed/unsigned

- Types prédéfinis du paquetage numeric_std pour des entiers binaires purs et signés (complément à 2) :

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

Binaire	Unsigned	Signed
0101	5	5
1101	13	-3

- La principale utilisation de ces types est d'effectuer des opérations arithmétiques (+, -, *, /, etc.).
- Syntaxe pour la déclaration est similaire à celle pour la déclaration d'un std_logic_vector:

```
constant unsignVal : unsigned(3 downto 0) := "1011";
```

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

Conversion entre les types

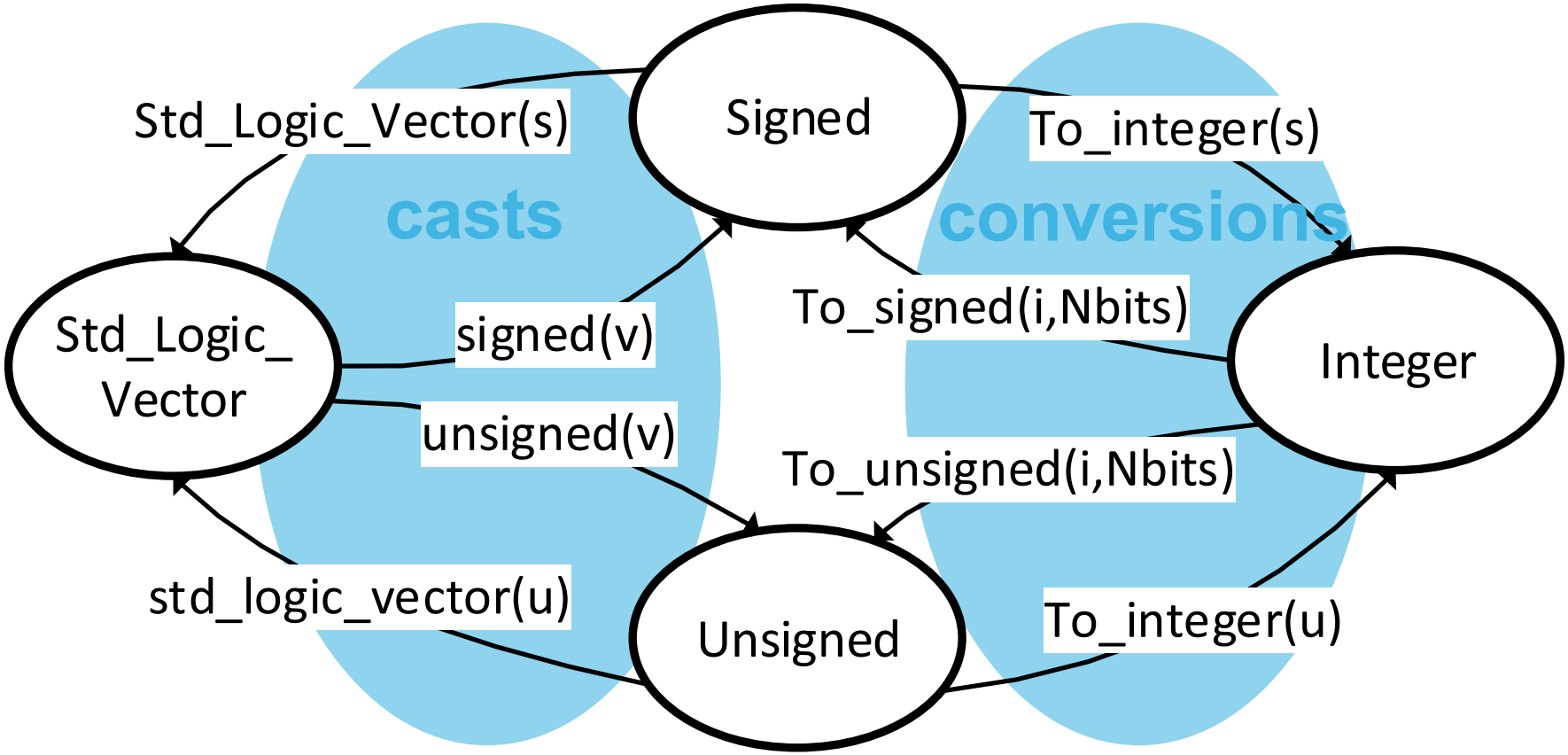
- Il est nécessaire d'effectuer des conversions (cast) entre différents types afin de permettre l'exécution d'opération arithmétique et/ou logique.

- Cependant, il n'est pas possible de convertir un std_logic_vector en integer directement

  →Il faut d'abord convertir en signed (ou unsigned) puis en integer

Conversions

- Ce diagramme résume les principales conversions possibles en VHDL:



Types composites: enregistrements

- Collection de champs dont les valeurs peuvent être de types différents:

```
type memory_bus is record
    addr   : std_logic_vector(15 downto 0);
    data   : std_logic_vector(7 downto 0);
    wr     : std_logic;
end record memory_bus;
```

- Accès aux champs:

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

NB :

Jusqu'à VHDL-2008, tous les types utilisés dans la définition d'un type enregistrement devaient être contraints.

Types composites: enregistrements

- Les enregistrements sont généralement utilisés pour représenter des bus.
- A utiliser aussi comme ports des entités
- Peuvent être utilisés pour les testbenchs
- Simplifie l'écriture et la connectivité

```
entity memory is
port ( bus_in : in memory_bus;
       bus_out : out memory_bus);
end entity memory;
```

Types accès

- Le type access correspond au mécanisme de pointeur du langage C.
- Exemple:

```
type integer_ptr is access integer; -- pointeur sur integer
variable a : integer_ptr;

a := new integer;
a.all := 1234567689; -- .all accède à la valeur
```

NB :

Les pointeurs ne sont pas synthétisables.

Agrégats

- Les agrégats permettent de définir des valeurs pour les types composites tableau et enregistrement.
- Exemples:

```
signal a : std_logic_vector(3 downto 0);
-- affectation de “1000“
Named notation: indice noté explicitement
a <= (3 => '1', 2 => '0', 1 => '0', 0 => '0');
a <= (3 => '1', 2 | 1 | 0 => '0'); -- choix
a <= (3 => '1', 2 downto 0 => '0'); -- subrange
a <= (3 => '1', others => '0'); -- others doit toujours
                                -- être le dernier élément
Positional notation: indices implicites.
a <= ('1', '0', '0', '0');
```

NB:

- VHDL interdit de mélanger les 2 notations lors d’une déclaration.
- Il est conseillé d’utiliser la «named notation» pour augmenter la lisibilité et limiter le risque d’erreurs.

Agrégats

- Exemples avec un tableau multidimensionnel:

```
type symbol is ('a', 't', 'd', 'h', digit, cr, other);
type state is range 0 to 3;
type transition_matrix is array (state, symbol) of state;
constant named_state : transition_matrix :=
    ( 0 => ('a' => 1, others => 0),
      1 => ('t' => 2, others => 3),
      2 => ('d' => 3, 'h' => 2, others => 1),
      3 => (digit => 0, others => 2));

constant positional_state : transition_matrix :=
   ((1,0,0,0,0,0,0),
    (3,2,3,3,3,3,3),
    (1,1,3,2,1,1,1),
    (2,2,2,2,0,2,2));
```

Agrégats

- Exemple avec un enregistrement:

```
type pair is record
    first : integer;
    second : integer;
end record;

signal p, a : pair;
p <= (first => 0, second => 0);
p <= (0, 0);

signal c, d : integer;
a <= (first => c, second => c+d); --agregat avec expressions

(c, d) <= p; -- similaire à c <= p.first; d <= p.second;
```

Opérateurs

VHDL définit un ensemble d'opérateurs:

Dans l'ordre de précédence:

- Divers: ** **abs not**
- Multiplication: * / **mod rem**
- Signe (unaire): **+ -**
- Addition: **+ - & (concaténation)**
- Décalage: **sll srl sla sra rol ror**
- Relationnel: **= /= < <= > >=**
- Logiques : **and or nand nor xor xnor**

Types supportés par chaque classe d'opérateurs

- Logiques : Bit, Boolean, std_logic, std_logic_vector
- Relationnel: tous les types, résultat Boolean
- Décalage: Bit_Vector, std_logic_vector
- Concaténation: types Array 1D,
  aussi avec élément de base
- Addition: types entiers (Integer, Natural,…),
  signed, unsigned
- Signe (unaire): types entiers (Integer, Natural,…)
- Multiplication: types entiers (Integer, Natural,…),
  signed, unsigned
- Divers: types entiers (Integer, Natural,…),
  signed, unsigned

Surcharge ou définition d'opérateur

Les opérateurs peuvent être définis pour les nouveaux types déclarés ou on peut surcharger des opérateurs existants:

```
function "+"(a: INTEGER, b: BIT) return INTEGER is
begin
    if b = '1' THEN
      return a+1;
    else
      return a;
    end if;
END "+";
```

```
signal inp1, outp: INTEGER range 0 to 15;
signal inp2: BIT;

outp <= 3 + inp1 + inp2;
```

Exemple d'affectation

- Une simple affection (sans voir la déclaration) peut suggérer plusieurs types pour un même signal

```
x0 <= '0';              -- bit, std_logic, or std_ulogic value '0'
x1 <= "00011111";       -- bit_vector, std_logic_vector,
                        -- std_ulogic_vector, signed, or unsigned
x2 <= "101111"          -- binary representation of decimal 47
x3 <= B"101111"         -- binary representation of decimal 47
x4 <= O"57"             -- octal representation of decimal 47
x5 <= X"2F"             -- hexadecimal representation of decimal 47
n <= 1200;              -- integer
IF ready THEN...        -- Boolean, executed if ready=TRUE
y <= 1.2E-5;            -- real, not synthesizable
q <= d after 10 ns;     -- physical, not synthesizable
```

Exemple d'affection

- Opérations autorisées et interdites entre des signaux de types différents

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR(7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
...
a <= b(5);   -- legal (same scalar type: BIT)
b(0) <= a;   -- legal (same scalar type: BIT)
c <= d(5);   -- legal (same scalar type: STD_LOGIC)
d(0) <= c;   -- legal (same scalar type: STD_LOGIC)
a <= c;      -- illegal (type mismatch: BIT x STD_LOGIC)
b <= d;      -- illegal (type mismatch: BIT_VECTOR x STD_LOGIC_VECTOR)
e <= b;      -- illegal (type mismatch: INTEGER x BIT_VECTOR)
e <= d;      -- illegal (type mismatch: INTEGER x STD_LOGIC_VECTOR)
```

Exercices:

- Série SN-Ch2