ELEMENTS DE BASE DU LANGAGE VHDL

I. INTRODUCTION

Auparavant pour décrire le fonctionnement d'un circuit électronique programmable les techniciens et les ingénieurs utilisaient des langages de bas niveau (ABEL, PALASM, ORCAD/PLD, ...) ou plus simplement un outil de saisie de schémas.

Actuellement la densité de fonctions logiques (portes et bascules) intégrées dans les **PLD** est telle (plusieurs milliers de portes voire millions de portes) qu'il n'est plus possible d'utiliser les outils d'hier pour développer les circuits d'aujourd'hui.

Les sociétés de développement et les ingénieurs ont voulu s'affranchir des contraintes technologiques des circuits. Ils ont donc créé des langages dits de haut niveau à savoir VHDL et VERILOG. Ces deux langages font abstraction des contraintes technologies des circuits PLD.

- les instructions écrites dans ces langages se traduisent par une configuration logique de portes et de bascules qui est intégrée à l'intérieur des circuits PLD.

II. LE LANGAGE VHDL

VHDL est le sigle de V_{HSIC} **H**ardware **D**escription **L**anguage ; **V**$_{HSIC}$ vient quant à lui de Very-High-Speed Integrated Circuits.

C'est un langage de programmation dont *la vocation est de donner naissance à un circuit logique et non à un programme executable*. Ce circuit peut se matérialiser dans un CPLD (Complex Programmable Logic Device) ou plus grand encore, dans un FPGA (field programmable gate array).

En tant que standard, VHDL est independant du logiciel utilisé pour la compilation, la programmation des composants, la simulation, etc. Il autorise plusieurs methodologies de conception (comportemental, flot de donnees, structurel) tout en étant d'un très haut niveau d'abstraction en electronique (il est independant de la technologie utilisee : FPGA, CPLD, ASIC, etc.)

Utiliser ce langage évite de dériver des tables de vérité des spécifications, que l'on simplifie alors en équations booléennes à implémenter. Il permet d'expliciter la fonction à implémenter sans se soucier de ces détails, bien que l'implémentation en pratique se fasse à l'aide de portes logiques (le compilateur se chargera de passer du code aux portes).

Ce langage n'est pas dénué d'extensions, il ne se limite pas à la description de circuits numériques. Notamment, VHDL-AMS inclut des extensions pour gérer les signaux analogiques et mixtes (analog and mixed-signal, AMS), en offrant les mêmes avantages (description de

haut niveau, vérification, simulation). Également, VHDL (VHDL Procedural Interface) est une interface pour du code C ou C++. Plus étonnant peut-être, OO-VHDL, qui, comme son nom l'indique, ajoute des fonctionnalités orientées objet au VHDL, pour une abstraction d'encore plus haut niveau, tout en restant utilisable par du code VHDL.

- il n'est pas le seul sur le marché : VERILOG, par exemple, est egalement un langage très utilisé.

III. RELATION ENTRE UNE DESCRIPTION VHDL ET LES CIRCUITS LOGIQUES PROGRAMMABLES

Décrire le fonctionnement d'un circuit logique programmable c'est bien, mais comment faire le lien avec la structure de celui-ci ?

L'implantation d'une ou de plusieurs descriptions VHDL dans un PLD va dépendre de l'affectation que l'on fera des broches d'entrées/sorties et des structures de base du circuit logique programmable.

III.1. SCHEMA FONCTIONNEL D'IMPLANTATION DE DESCRIPTIONS VHDL DANS UN CIRCUIT LOGIQUE PROGRAMMABLE

La figure ci-dessous représente un exemple d'implantation de descriptions VHDL ou de blocs fonctionnels implantés dans un PLD. Lors de la phase de synthèse chaque bloc sera matérialisé par des portes et/ou des bascules. La phase suivante sera d'implanter les portes et les bascules à l'intérieur du circuit logique. Cette tâche sera réalisée par le logiciel placement/routage (« Fitter »), au cours de laquelle les entrées et sorties seront affectées à des numéros de broches.

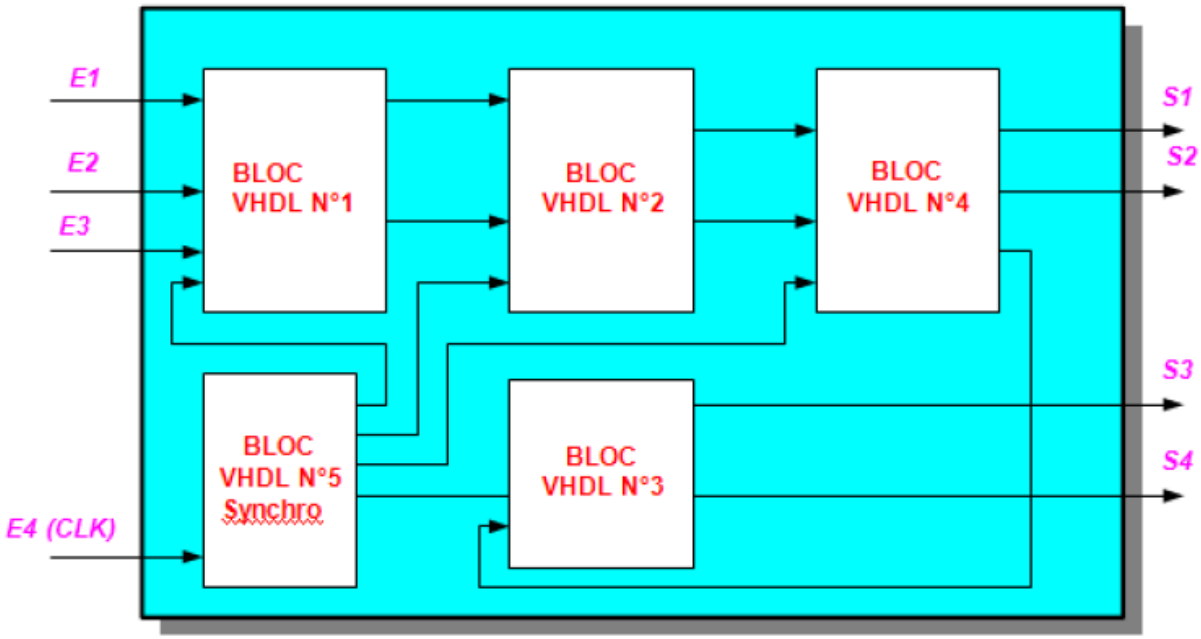


Figure II.1 : Implantation de descriptions VHDL ou de blocs fonctionnels implantés dans un PLD

III.2. L'AFFECTATION DES BROCHES D'ENTREES SORTIES

Elle peut se faire de plusieurs manières :

III.2.1. L'AFFECTATION AUTOMATIQUE

On laisse le **synthétiseur** et le **fitter** du fabriquant (« Fondeur » : Xilinx, Lattice, Altéra, Cypress...») du circuit implanter la structure correspondant à la description VHDL. Les numéros de broches seront choisis de façon automatique.

III.2.2. L'AFFECTATION MANUELLE

On définit les numéros de broches dans la description VHDL ou sur un schéma bloc définissant les liaisons entre les différents blocs VHDL ou dans un fichier texte propre au fondeur. Les numéros de broches seront affectés suivant les consignes données.

III.3. ORGANISATION FONCTIONNELLE DE DEVELOPPEMENT D'UN PLD

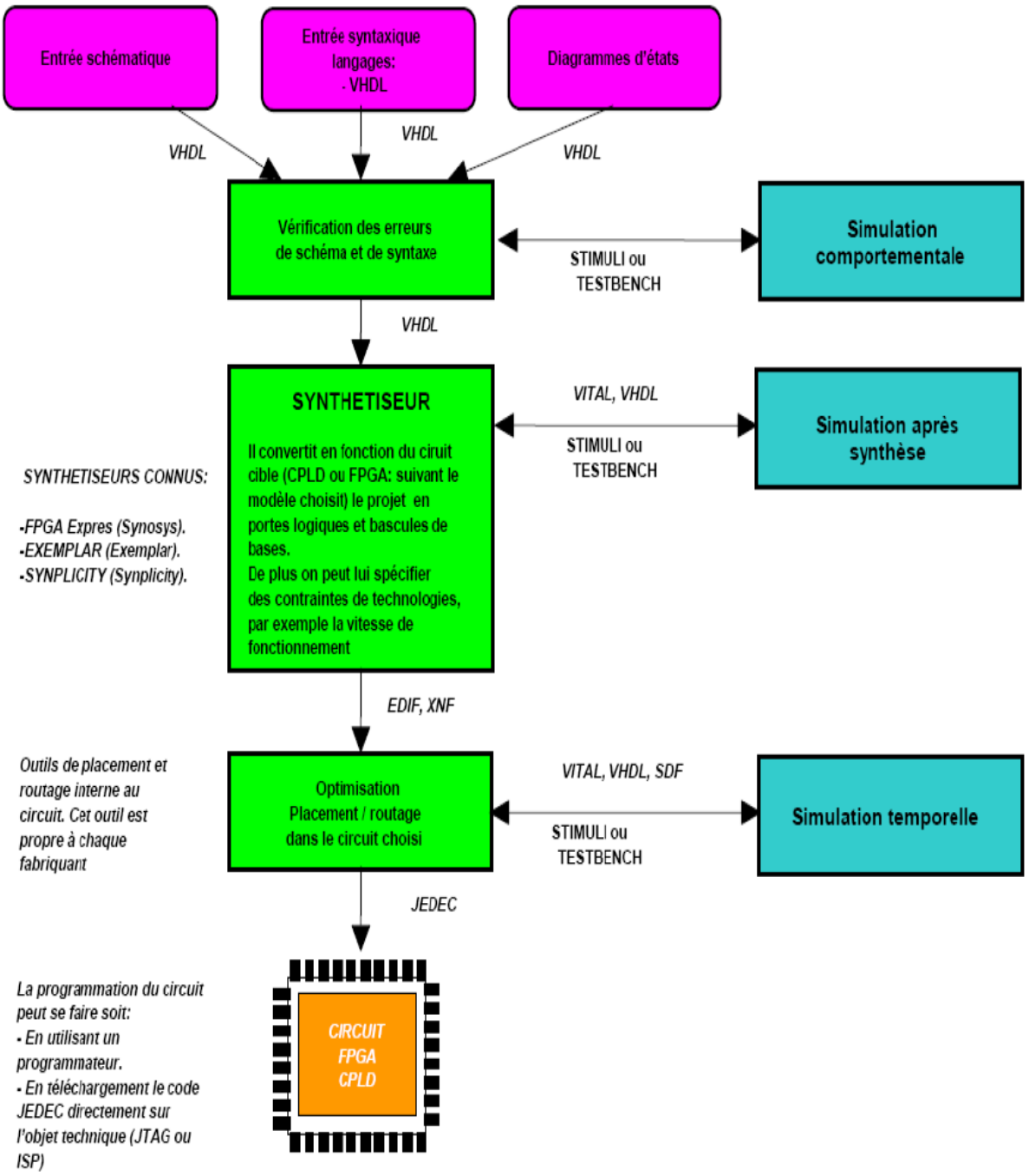


Figure II.2 : Organisation fonctionnelle de développement d'un PLD

IV. OUTILS

Les fabricants des puces programmables fournissent généralement, sur leur site ou avec les puces, tous les logiciels nécessaires pour les utiliser: compilateur ou synthétiseur VHDL, qui *se chargera de convertir le code VHDL en sa version logique*, prête pour la puce; *simulateur, pour tester le code*.

Pour l'édition simple du code, tout éditeur convient également : Notepad++ et jEdit par exemple.

Il est possible de *s'armer de manière totalement open source : GHDL est un compilateur VHDL basé sur GCC et un simulateur, tandis que GTKWave permet d'en visualiser les résultats.*

V. PREMIERS ELEMENTS DE SYNTAXE

V.1. ELEMENTS DE BASE

Deux types sont à la base de tout en VHDL : bit et bit_vector. Cependant, ils sont très crus, car seules deux valeurs sont possibles (vrai et faux) : comment représenter une haute *impédance*, par exemple ? Comment gérer les don't cares (dans certains cas, la valeur de sortie n'a aucune importance, on peut en profiter pour optimiser encore plus l'implémentation d'une fonction) ?

Le standard IEEE 1164 vient résoudre ce problème avec le types std_logic et std_logic_vector, respectivement ***un bit*** et ***un vecteur de bits***. Les valeurs possibles *sont* principalement des booléens (vrai 1 et faux 0, d'intensité faible au besoin - L et H), le don't care - et la haute impédance Z, utile pour relier plusieurs sorties sans risque d'instabilité du circuit. D'autres valeurs sont également possibles (U pour une valeur non initialisée, X pour une valeur inconnue forte, W pour une valeur inconnue faible), bien que moins utilisées.

- Au niveau de la syntaxe, ***pour indiquer la valeur d'un bit, scalaire, on met le symbole entre apostrophes droites :*** '**1**' ; ***dans le cas d'un vecteur, on utilisera des guillemets droites :*** "**11**".

Ces types sont définis dans la bibliothèque **ieee.std_logic_1164.all** (il faudra le spécifier dans chaque entité vhdl).

V.2. PAIRE ENTITY/ARCHITECTURE

VHDL nous intéresse en tant que langage pour la description, simulation et synthèse des systèmes matériels digitaux ;

Qu'est qu'un système matériel ?

En général, il s'agit d'un schéma mettant en œuvre :

- un certain nombre de composants
- des connexions entre ces composants

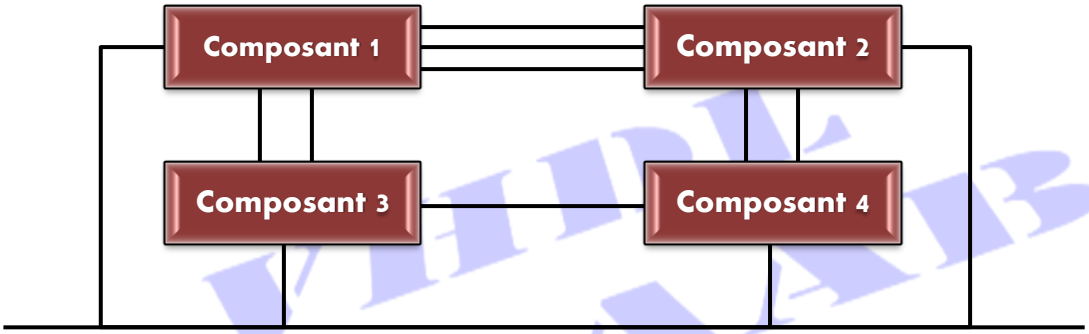


Figure II.3: Composantes d'un système matériel

Au plus haut niveau d'abstraction, un système digital est vu comme une "boîte noire", dont on connaît l'interface avec l'extérieur mais dont on ignore le contenu ;

- En VHDL la boîte noire est nommée entité (entity) ;

Une entity doit toujours être associée avec au moins une description de son contenu, de son implémentation: c'est l'architecture ;

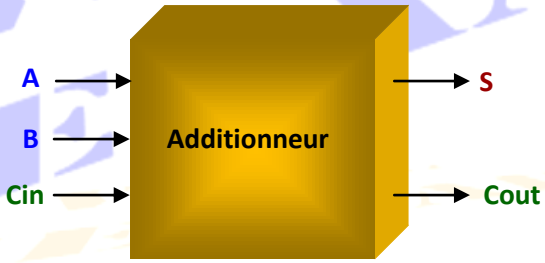


Figure II.4: Entity = Vue externe du composant

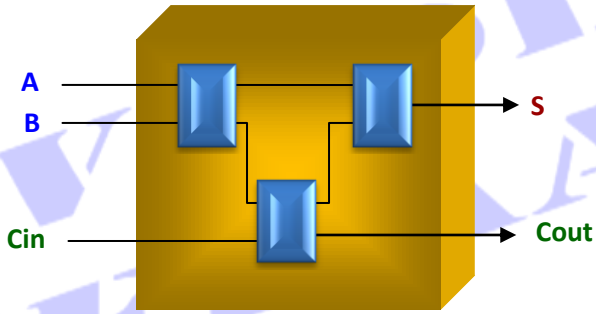


Figure II.5: Architecture = Vue interne du composant

- *Tout programme VHDL contient au moins une paire* entity/architecture.

V.2.1. L'ENTITE (entity)

L'entité (entity) définit *la vue externe du modèle*. On peut la symboliser par un rectangle avec des " broches " d'entrées et sorties. Ce sont les entrées et sorties de l'architecture. *Elle est vue de l'extérieur du composant (interface)* ;

Syntaxe

```
entity  entity_Identifier is
    Zone de déclarations.
begin
     Instructions : cette liste d'instructions peut contenir:
end entity entity_Identifier;
```

- *Une entity peut être assignee à plusieurs architectures.*
- *Toutes les déclarations faites à l'intérieur de l'entity sont visible à l'intérieur de toutes les architectures de cette entity.*

Exemple :

```
entity name is
    port(
        A           : in  std_logic;
        Data_bus    : in  std_logic_vector(7 downto 0)
    );
end name;
```

- Après la dernière définition de signal de l'instruction port il ne faut jamais mettre de point virgule

Dans cette exemple, A est une entrée simple et data_bus est également une entrée mais qui dispose d'un ensemble de 8 bits (donc un bus). Un vecteur de bit déclaré comme ci-dessus aura le bit de poids fort en l'indice le plus élevé et le bit de poids faible en zéro.

- On peut également définir des valeurs par défaut à l'aide d'une expression et de l'opérateur **:=** :

```
entity name is
    port(
        S: in std_logic_vector(1 downto 0) := "10"
    );
end name;
```

CARACTERISTIQUES DES SIGNAUX D'UN PORT

Le signal d'un port est définit par :

1. son ***mode*** qui indique la direction de circulation de l'information et si le port peut ou non être lu à l'intérieur de l'entité. Ces différents modes sont :
   - **MODE ENTREE (in)**

L'entité lit un signal d'entrée fourni par l'extérieur (mais ne peut pas le modifié).

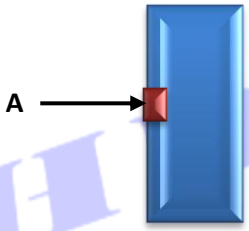


Figure II.8: Signal en mode entrée (in)

- **MODE SORTIE (out)**

  L'entité fournit un signal de sortie, mais ne peut pas relire ce signal.

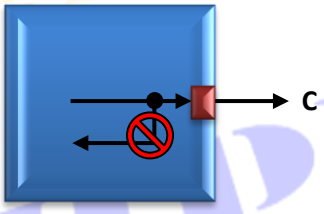


Figure II.9: Signal en mode entrée (out)

- **MODE ENTRE/SORTIE (inout)**

  Le signal peut être lu et modifié à l'intérieur de l'entité, le signal est bidirectionnel : en sortie, il est fourni par l'entité; en entrée, il est fourni par l'extérieur. Ce mode autorise aussi le bouclage interne (ex : bus de données)

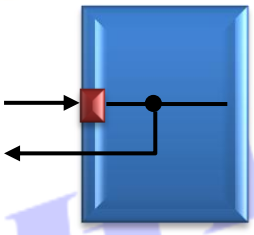


Figure II.10: Signal en mode entrée (inout)

- **MODE buffer**

  Il correspond à une sortie rebouclée en entrée. L'architecture de l'entité fabrique un signal utilisable en sortie, qui peut aussi être relu par l'entité comme un signal interne (ex : sorties d'un compteur dont l'état doit être testé).

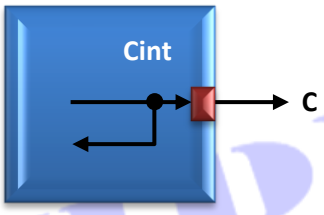


Figure II.11: Signal en mode entrée (buffer)

☞ Par défaut, le mode d'un port est **in**.

2. son *type* std_logic **(***pour un bit***)**, std_logic_vector **(***pour un vecteur de bits***)**, etc...

3. un *intervalle* de variation range, etc...

V.2.2. L'ARCHITECTURE (architecture)

L'architecture définit l'implémentation proprement dite : le contenu du rectangle dont les entrées sorties sont défini par l'entity. Il ya plusieurs façons de décrire le comportement du modèle. Chacune d'elles est une architecture de l'entity. L'architecture est donc *l'intérieur du composant, Elle explicitera les liens entre les entrées et les sorties (implémentation).*

- L'architecture dispose de deux parties :
  - Une partie déclarative : Dans cette partie sont déclarés tous les objets internes à cette architecture;
  - Une partie exécution composée d'un ensemble d'instructions concurrectes;
- L'architecture établit, à travers les instructions, les relations entre les entrées et les sorties.
- Le fonctionnement de l'architecture peut être purement combinatoire, séquentiel voire les deux séquentiel et combinatoire.

Au sein d'un programma VHDL, l'architecture apparaît comme ci dessous :

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

entity entity_Identifier is
    port(
        Déclaration des entrées/sorties
    );
end entity_Identifier;

architecture  architecture_Identifier of entity_Identifier is          -- Declaration of the architecture of the entity.
    Zone de déclarations.
    
begin
    Ensemble d'instructions.
```

```
end architecture_Identifier;
```

- Les entrées/sorties du système sont les **ports** de l'entité
- Chaque composant interne du système sera un processus (process) de l'architecture
- Une architecture est un ensemble de process implicite ou explicite
- Les process s'exécutent en parallèle mais les instructions à l'intérieur d'un process sont exécutées séquentiellement.
- Les process de l'architecture sont interconnectés par le biais des signaux (**signal**: *ils servent à passer les résultats intermédiaires d'un bloc fonctionnel à un autre. On les utilisera en pratique lors de la présentation des architectures comportementales*.

Exemple :

```
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity exemple is                                          -- entity declaration.
    port(
        A, B            : in    std_logic_vector(3 downto 0);
        op, Clk, Reset  : in    std_logic;
        C               : out   std_logic_vector(3 downto 0)
    );
end exemple;
architecture test of exemple is                            -- Declaration of the architecture of the entity.
    signal moinsB, opB, AopB : std_logic_vector(3 downto 0);      -- The key word signal may be igorated
begin                                                      -- because taken by default
    moinsB  <= not b + "0001";                             -- This is is an implicit process
    opB     <= B   when   op='0'                           -- This is an implicit process
               else  moinsB;
    AopB    <= A + opB;                                    -- This is an implicit process
    process(Reset, Clk)                                    -- This is an implicit process
    begin
        if Reset='0' then
            C <= "0000";
        else
            if(clk'event and clk='1') then
                C <= AopB;
            end if;
        end if;
    end process;
end test;
```

☞ Pour modéliser un système complet, on utilisera une série de paires entité-architecture.

Quelques notes sur la syntaxe d'un programme VHDL:

- pas de différentiation entre majuscules et minuscules

- toute phrase termine par un point virgule « ; »
- le début d'un commentaire est signalé par un double trait (“--”). Le commentaire termine avec la fin de ligne

V.3. LES LIBRAIRIES (BIBLIOTHEQUES) PRE DEFINIES

Elles jouent le même rôle que les librairies **lib.h** du langage C. Elles contiennent des déclarations de variables, des sous programmes, des composants génériques, etc..., nécessaires à la compilation des modèles. On peut créer ses propres librairies

- Les Librairies sont déclarées avec le mot clé **library**, elles dépendent des outils utilisés. Elles contiennent des paquetages que l’on déclare vouloir utiliser avec le mot clé **use**:

Syntaxe d’utilisation

```
use bibliotheque.paquetage.all;
```

- La bibliothèque par défaut est work. work est aussi le nom symbolique de la bibliothèque dans laquelle sont stockés les résultats ;
- La bibliothèque std est une bibliothèque standard fournie avec le langage, elle contient des définitions des types et des fonctions de base (integer, bit,...).
- Les bibliothèques work et std n'ont pas besoin d'être déclarées. Chaque développement incorpore implicitement les lignes suivantes :

```
library  work;
library  std;
library  std.standard.all;
```

V.3. 1. LE PACKAGE IEEE.STD_LOGIC_1164.ALL

Pour la synthèse, les types de données les plus utilisés sont **std_logic**, pour les données à un bit, et std_logic_vector, pour les bus (vecteur de bits). Ces types ne sont pas prédéfinis et pour les utiliser il faut déclarer le package std_logic_1164, qui fait partie de la bibliothèque (library) ieee:

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

Une donnée de type std_logic possède une valeur parmi neuf possibles. Ces valeurs *décrivent tous les états d’un signal logique*:

* ⇒ Utile pour la synthèse, ** ⇒ Utile pour la simulation

- 'U' : non initialisé (Uninitialized) **

- 'X' : niveau inconnu, forçage fort (forcing unknown) **
- '0' : niveau 0, forçage fort (forcing 0)
- '1' : niveau 1, forçage fort (forcing 1)
- 'Z' : haute impédance high (impedance) *
- 'W' : niveau inconnu, forçage faible (weak unknown) **
- 'L' : niveau 0, forçage faible (weak 0 =pull-down)
- 'H' : niveau 1, forçage faible (weak 1=pull-up)
- '-' : quelconque (don't care)* : le moins de la touche 6 et non pas le underligne de la touche 8

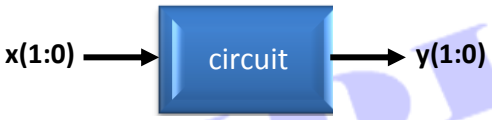
Pour une affectation, les valeurs utilisées sont: 'X', '0', '1', 'Z'

Remarques concernant les synthétiseurs

- '0' et 'L' sont équivalents
- '1' et 'H' sont équivalents
- '-' le moins de la touche 6 et non pas le underligne de la touche 8 : très utile pour la simplification des équations
- **'U', 'X', 'W'** sont interdits

Arretons nous un petit peu sur l'utilité du "Don't Care" "--" dans la simplification des équations. Pour cela considérons l'exemple simple suivant:

Soit la figure (a) ci-dessous qui représente un circuit dont l'entrée (**x**) et la sortie (**y**) sont des signaux à 2 bits pour lesquels deux ensembles de spécifications sont donnés dans les tables de vérité des figures (b) et (c).



(a)

X_1 X_0	Y_1 Y_0
0 0	0 0
0 1	1 0
1 0	0 1
1 1	0 0

(b)

X_1 X_0	Y_1 Y_0
0 0	0 0
0 1	1 0
1 0	0 1
1 1	- -

(c)

Figure II.12: Utilité du don't care

Dans le premier cas, toutes les sorties sont spécifiées avec des "**0**" et des "**1**", tandis que dans le second, il existe une sortie "**Don't Care**" (**y** "--").

En utilisant les tables de Karnaugh, nous obtenons les équations optimales suivantes pour y de la figure b:

$$y = \begin{cases} y_0 = \overline{x_0} \cdot x_1 \\ y_1 = x_0 \cdot \overline{x_1} \end{cases}$$

Par contre, pour y de la figure c, nous obtenons:

$$y = \begin{cases} y_0 = x_1 \\ y_1 = x_0 \end{cases}$$

A partir de ces deux ensemble d'équations de "y", nous pouvons constater déjà la grande simplification apportée par le "**Don't Care**". Est-ce que cette simplification est toujours aussi apparente en VHDL?

Developpons alors le code VHDL correspondant à chaque table de vérité et observons les conséquences de l'utilisation de la valeur logique '-' dans le code.

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

use ieee.std_logic_unsigned.all;

entity dontcare is
    port(
        x      : in    std_logic_vector(1 downto 0);

        y      : out std_logic_vector(1 downto 0)
    );
end dontcare;

architecture arc of dontcare is
begin
    y <= "00" when x = "00" else

          "01" when x = "10" else

          "10" when x = "01" else

          "00" when x = "11";
end arc;
```

programme correspondant à la non utilization du Don't Care "—"

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

use ieee.std_logic_unsigned.all;

entity dontcare is
    port(
        x      : in    std_logic_vector(1 downto 0);

        y      : out std_logic_vector(1 downto 0)
    );
end dontcare;

architecture arc of dontcare is
begin
    y <= "00" when x = "00" else

          "01" when x = "10" else

          "10" when x = "01" else

          "--";              -- Don't Care
end arc;
```

programme correspondant à l'utilization du Don't Care "—"

Les circuits résultants de ces deux codes sont donnés ci-dessous.

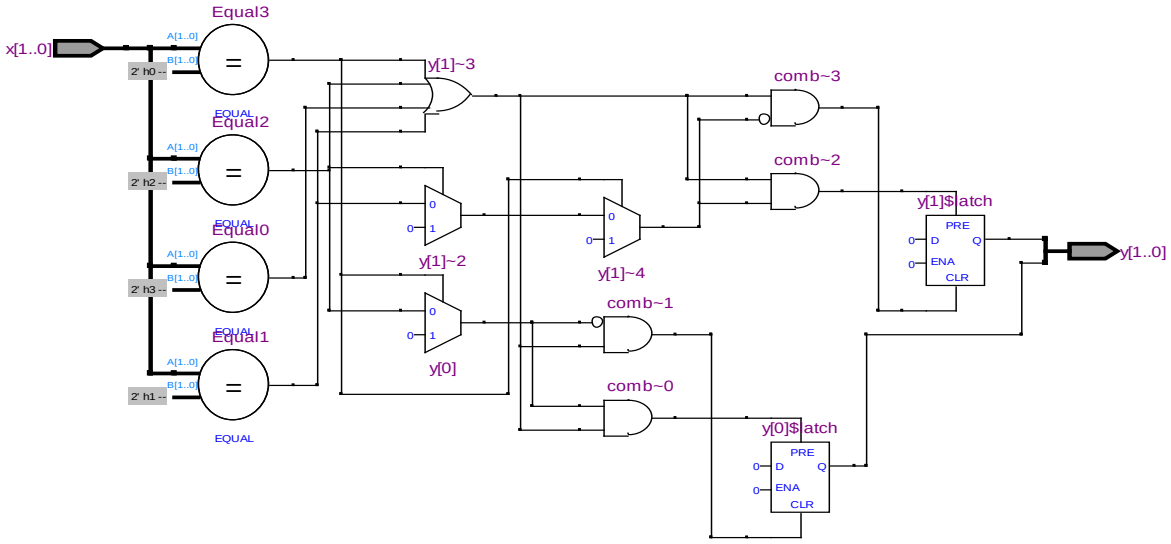


Figure II.13: Circuit résultant à la non utlisation du Don't Care "—"

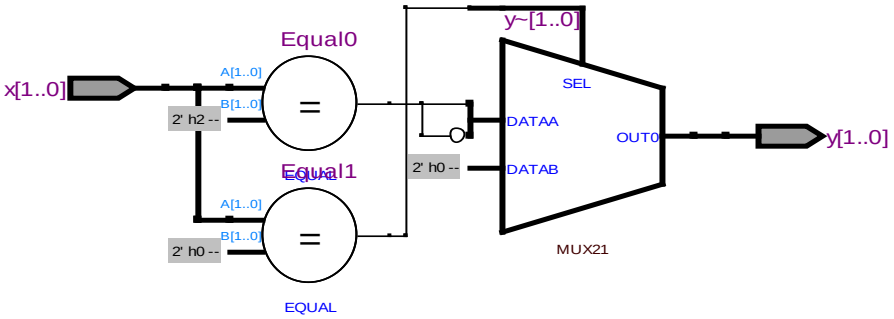


Figure II.14: Circuit résultant de l'utlisation du Don't Care "—"

Les deux circuits confirment bien le résultat déjà avancé, spécifiant que l'utilisation du Don't Care "—" donne lieu à un circuit beaucoup plus simple necessitant moins de matériel donc un coût réduit, une consommation électrique réduite et une vitesse supérieure.

V.3. 2. LE PACKAGE IEEE.NUMERIC_STD.ALL

- Définit les types signed et unsigned. Un vecteur représente alors un nombre signé ou non signé représenté en complément à 2
- Permet d'utiliser les opérateurs arithmétiques + et × sur des vecteurs de bits
- Fournit des fonctions de conversion entre entiers et vecteurs.

VI. LES OPERATEURS

VI. 1. OPERATEUR D'AFFECTATION: <=

Dans une description VHDL, c'est certainement l'opérateur le plus utilisé. En effet il permet de modifier l'état d'un signal en fonction d'autres signaux et/ou d'autres opérateurs.

Exemple avec des portes logiques : S1 <= E2 and E1 ;

Les valeurs numériques que l'on peut affecter à un signal sont les suivantes :

- '1' ou 'H' pour un niveau haut avec un signal de 1 bit.
- '0' ou 'L' pour un niveau bas avec un signal de 1 bit.
- 'z' pour un état haute impédance avec un signal de 1 bit.
- '-' le moins de la touche 6 et non pas le underligne de la touche 8 : pour un état quelconque, c'est à dire '0' ou '1'.

Cette dernière valeur est très utilisée avec les instructions : when ... else et with .... Select ....

Pour les signaux composés de plusieurs bits on utilise les guillemets " ... ", voir les exemples ci dessous :

Les bases numériques utilisées pour les bus peuvent être :

```
BINAIRE :  BUS <= "1001" ;          -- BUS = 9 in Binary
HEXA :     BUS <= X"9" ;            -- BUS = 9 in Hexadecimal
OCTAL :    BUS <= O"11" ;           -- BUS = 9 in Octal
```

Remarque :

La base décimale ne peut pas être utilisée lors de l'affectation de signaux. On peut seulement l'utiliser avec certains opérateurs, comme + et – pour réaliser des compteurs.

Exemple :

```
Library ieee;
Use ieee.std_logic_1164.all;

entity Affec is
    port (
        E1,E2               : in  std_logic;
        BUS1, BUS2, BUS3    : out std_logic_vector(3 downto 0);
        S1, S2, S3, S4      : out std_logic
    );
end Affec;

architecture Description of Affec is
begin
    S1 <= '1';                      -- S1 = 1
    S2 <= '0';                      -- S2 = 0
    S3 <= E1;                       -- S3 = E1
```

```
    S4 <= '1' when (E2 ='1') else 'Z';          -- S4 = 1 if E2=1, else S4 takes the high impedance value
    BUS1 <= "1000";                             -- Bus1 = "1000"
    BUS2 <= E1 & E2 & "10";                     -- Bus2 = E1 & E2 & 10
    BUS3 <= X"A";                               -- A in HEX -> BUS3 = 10 (dec)
end Description;
```

- VHDL est un langage dans leuel il est possible d'étendre le domaine d'utilisation d'un opérateur (types des opérandes) par l'écriture d'une fonction qui porte le nom de l'opérateur, c'est ce que l'on appele *la surcharge*. Cette opération ne modifie pas la classe de priorité à laquelle appartient l'opérateur.

VI. 2. OPERATEUR DE CONCATENATION : &

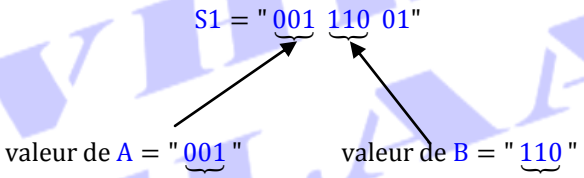
Cet opérateur permet de joindre des signaux entre eux.

Exemple :

Soit A et B de type 3 bits et S1 de type 8 bits : A = "001" et B ="110"

S1 <= A & B & "01" ;

S1 prendra la valeur suivante après cette affectation



VI. 3. LES OPERATEURS LOGIQUES

Tableau II.2: Opérateurs VHDL Logiques

Opérateur	VHDL
et	and
non et	nand
ou	or
non ou	nor
ou exclusif	xor
non ou exclusif	xnor
non	not
decalage à gauche	sll
decalage à droite	srl
rotation à gauche	rol
rotation à droite	ror

Exemples :

```
S1 <= A sll 2 ;                    -- S1 = 2 bits left shifted of A.
S2 <= A rol 3 ;                    -- S2 = 3 bits left rorated of A
S3 <= not (r);                     -- S3 = binary inverse of r
```

Remarque :

Pour réaliser des décalages logiques en synthèse logique, il est préférable d'utiliser les instructions suivantes :

Décalage à droite :

Si A est de type std_logic_vector(7 downto 0)

```
S1 <= '0' & A(7 downto 1);         -- one bit right shift
S1 <= "000" & A(7 downto 3);       -- three bit right shift
```

Décalage à gauche :

Si A est de type std_logic_vector(7 downto 0)

```
S1 <= A(6 downto 0) & '0';         -- one bit shift left
S1 <= A(4 downto 0) & "000";       -- three bit shift left
```

VI. 4. LES OPERATEURS ARITHMETIQUES

Tableau II.3: Opérateurs VHDL arithmétiques

Opérateur	VHDL
addition	+
soustraction	–
multiplication	*
division	/

☞ Les opérateurs +, -, *, et / sont définis pour les entiers et les réels. Les deux opérandes doivent être de même type et le résultat et également du même type que les opérandes.

```
signal    int1, int2    :   integer := 0;
signal    real1, real2  :   real  := 6.7;
...
int1    <=  int1 + 3;
real1   <=   real2 - 2.2;
int2    <=   int1 * real1;                -- illegal
int2    <=   int1 * integer(real1);
```

```
real2   <=   real1 / 42.3;
```

Remarque

Pour pouvoir utiliser les opérateurs ci-dessus il faut ajouter les bibliothèques suivantes au début du fichier **VHDL**:

```
Use ieee.numeric_std.all ;
Use ieee.std_logic_arith.all ;
```

- Dans l'outil **Xilinx ISE**, l'addition (+), la soustraction (-), la multiplication (∗) et la division (/) des vecteurs sont définies dans le paquetage std_logic_unsigned de la bibliothèque ieee

```
library ieee ;
use ieee.std_logic_unsigned.all;
```

Exemples :

```
S1 <= A - 3 ;                -- S1 = A – 3: subtract 3 from the value of input A
S1 <= S1 + 1 ;               -- increments signal S1 by 1
S1 <= A * B ;                -- S1 = A multiplied by B
S2 <= A / B ;                -- S2 = A divided by B
```

Remarque

Attention, l'utilisation de ces opérateurs avec des signaux comportant un nombre de bits important peut générer de grandes structures électroniques.

VI. 5. LES OPERATEURS RELATIONNELS

Ils permettent de modifier l'état d'un signal ou de signaux suivant le résultat d'un test ou d'une condition. En logique combinatoire, ils sont souvent utilisés avec les instructions :

- when ... else ...
- with .... select ....

Tableau II.4: Opérateurs VHDL relationnels

Opérateur	VHDL
egal	=
non égal	/=
inférieur	<
inférieur ou égal	<=
supérieur	>
supérieur ou égal	>=

Dans une architecture de type flot de données (voir plus loin), on définit explicitement les expressions booléennes reliant les entrées et les sorties. Peu importe le style utilisé, l'implémentation produite par le compilateur sera identique si l'architecture fait exactement la même chose.

- Les **commentaires** sont indiqués par — (double signe moins).

VII. SIGNAUX, VARIABLES ET CONSTANTES

VHDL manipule trois familles d'objets : les ***signaux***, les ***constantes*** et les ***variables***. Chaque objet et repéré par son nom (identificateur). Ce nom doit respecter les conventions suivantes :

- Il n'y a pas de différences entre minuscules et majuscules
- La longueur d'un identificateur ne doit pas excéder une ligne
- Un identificateur est composé de lettres, de chiffres, du symbole _ (soulignement). Il doit commencer par une lettre, on ne doit pas avoir de symboles _ _ consécutifs.
- Le dernier caractère doit être différent de _
- Les mots clés du langage ne peuvent pas être utilisés comme identificateurs.

VII.1. LES SIGNAUX (signal)

VHDL utilise les signaux (signal), équivalent des fils ou câbles d'interconnexions dans le monde réel. Ils sont la base des langages de description de matériel. Les signaux portent les informations des liaisons d'entrée, des liaisons de sortie et des liaisons internes (pour connecter les différents composants d'un système). Un signal représente une équipotentielle ce n'est de ce fait ni une entrée, ni une sortie, ni une entrée-sortie. *Aucun mode n'est donc à lui attribuer*. Un signal :

- Doit être déclaré avant utilisation du mot clé begin.
- Il est géré par un pilote (driver)
- Un signal ne peut être déclaré dans un process ou dans un sous programme (function, procedure). Il peut être déclaré exclusivement dans:
  1. Un package : il est alors global
  2. Une entity : il est alors commun à toutes les architectures de l'entité
  3. Une architecture : il est alors local à cette architecture

Les signaux sont toujours globaux à toute l'architecture. Un signal est donc connu par tous les processus d'une architecture donnée. Toutefois, un signal ne peut pas être modifié par plus

d'un processus: en effet, cela voudrait dire que le signal a plus d'une valeur à un moment donné ;

- *Les phrases à l'intérieur d'un processus sont toujours exécutées en séquence, y compris les affectations de signaux. Toutefois, les signaux* ***sont mis à jour en même temps****, au moment d'un* wait.
- L'affectation d'une valeur à un signal se fait par l'opérateur **<=** et peut être faite à tout moment. Un signal peut être lié à un autre par cet opérateur. Ainsi A **<=** B établit un lien *permanent* entre les signaux A et B, ce qui exige que ces derniers aient le même type.

Exemples :

```
package essai
    signal  CLK   : std_logic ;                    -- Declaration of a signal within a package
    signal  RST   : std_logic;                     -- The keyword signal can be omitted because taken by
                                                   -- default
end essai;

entity essai is
    port(
        a,b    : in    std_logic ;
        c      : out integer
    );
    signal RST : std_logic;                        -- Declaration of a signal within an entity
end essai;

architecture arch of essai is
    signal RST : std_logic;                        -- Declaration of a signal within an architecture
begin
....
end arch;

signal s : std_logic := '0';
•••
S <= '1' after 10 ns, '0' after 18 ns, '1' after 25 ns;
```

Ce qui donne:

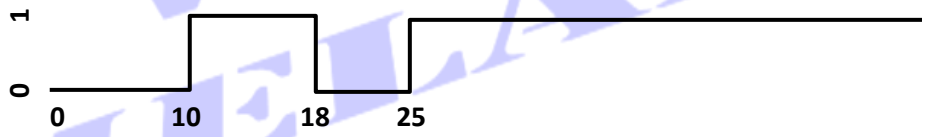


Figure II.15: Génération d'un signal par timing

Il existe deux types de signaux comme le montre la figure suivante :

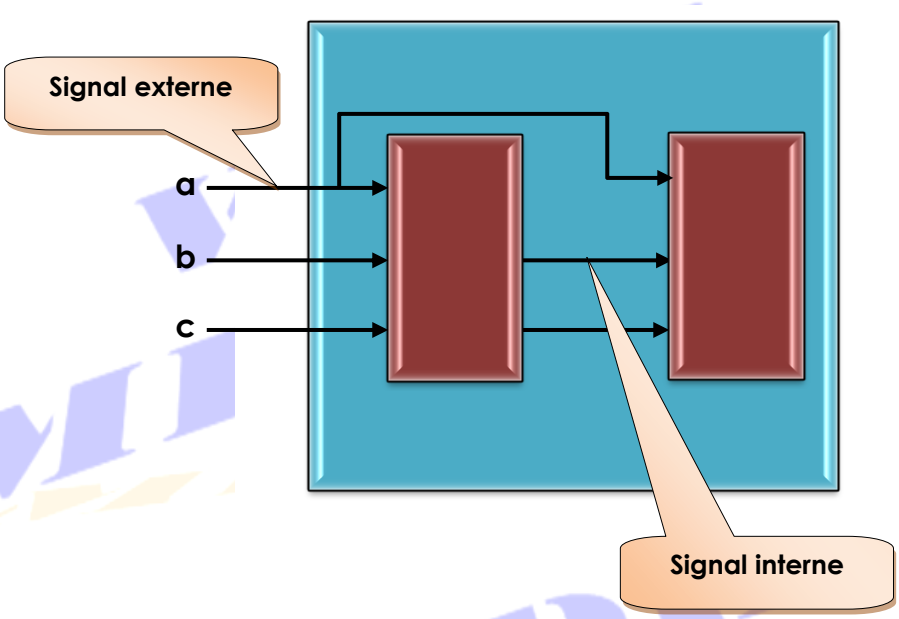


Figure II.16: Signaux internes VS signaux externes

VII.1. 1. ATTRIBUTS D'UN SIGNAL

Un attribut est une caractéristique associée à un type ou un signal. Il est placé juste après ce type ou ce signal mais séparé de lui par une apostrophe. La valeur d'un attribut peut être exploitée dans une expression.

Syntaxe

objet_name'attribut

A. ***Attributs pour les objets de type*** array ***ou scalaire***

Si x est un objet de type array alors :

x'high : retourne l'indice maximum (borne maximale) de x

x'low : retourne l'indice minimum (borne minimal) de x

x'left : retourne l'indice de gauche (borne de gauche) de x

x'right : retourne l'indice de droit (borne de droite) de x

A premier bord la paire x'high, x'low est quivalente respectivement à la paire x'left, x'right. Ceci n'est pas vrai parceque VHDL autorise un rangement en sense inverse des vecteurs. Considérons la déclaration suivante :

Signal slv: std_logic_vector(7 downto 0); ⇒ x'high = x'left =7 et x'low = x'right= 0

Signal slv: std_logic_vector(0 to 7); ⇒ x'high = 7 ≠ x'left=0 et x'low = 0 ≠ x'right = 7

Les deux couples sont complètement différents l'un de l'autre parceque l'ordre des indice dans les deux déclarations est différent.

B. ***Attributs pour les objets de type*** array ***ou un*** subtype

Si X est un objet de type array ou subarray alors :

X'range : retourne le rang de X

X'reverse_range : retourne le rang inverse de X

type Table is array (1 to 8) of Bit; ⇒ Table'range retourne 1 to 8 et Table'reverse_range retourne 8 to 1

Si Table est un tableau à plusieurs dimensions, Table'range(i) Retourne le rang de la dimension '**i**' de Table.

C. ***Attributs pour un SIGNAL***

X'event : retourne true si X change d'état

X'active : retourne true si X a changé durant le dernier interval

X'last_event : retourne une valeur temporelle depuis le dernier changement de X

X'last_active : retourne une valeur temporelle depuis la dernière transition de X

X'last_value : retourne la dernière valeur de X

Ces attributs créent un nouveau SIGNAL

X'delayed(t) : crée un signal du type de X retardé par t

X'stable(t) : retourne true si X n'est pas modifié pendant le temps t

X'quiet(t) : crée un signal logique à true si X n'est pas modifié pendant un temps t

X'transaction : crée un signal logique qui bascule lorsque X change d'état

Si S désigne l'identificateur d'un signal, l'ensemble des attributs qui peuvent lui être attribués sont résumés dans le tableau suivant.

Tableau II.5: Attributs d'un signal

Attribut	Signification
S'event	Retourne true si un évènement vient d'arriver sur S pendant le cycle de simulation en cours (c'est-à-dire, si le signal S change de valeur): S'event **:** prend la valeur true si le signal S change (passe du haut vers le bas ou vice versa)
S'active	Retourne true si une transaction arrive pendant le cycle de simulation en cours

S'quiet(T)	Retourne true si le signal S n'a eu ni transaction ni événement pendant un temps T
S'stable(T)	Retourne true s'il n'y a pas eu d'événement sur le signal S pendant le temps T
S'transaction	genère un signal de type bit qui change d'état pour chaque transaction du signal
S'delayed	genère un signal identique à S mais retardé de T
S'last_event	Time : rend le temps écoulé depuis le dernier événement sur le signal S
S'last_active	Time : rend le temps écoulé depuis la dernière transaction sur le signal S
S'last_value	rend la valeur du signal immédiatement avant le dernier changement de S
rising_edge(S)	retourne true s'il y a un passage, depuis le niveau bas vers le niveau haut, du signal S.
falling_edge(S)	retourne true s'il y a un passage, depuis le niveau haut vers le niveau bas, du signal S.
'pos, 'val, 'succ, 'pred, 'leftof, 'rightof	Attribut fonction sur des types discrets ordonnés
S'range	Placé près d'un signal, il retourne la valeur de l'intervalle spécifié par l'instruction range lors de la déclaration du signal ou du type utilisé. ⇒ Permet de vérifie, lors de l'analyse et de la simulation, que la valeur d'un objet n'est pas en dehors de son intervalle de variation.
S'reverse_range	Placé près d'un signal, il retourne la valeur de l'intervalle inverse spécifié par l'instruction range lors de la déclaration du signal ou du type utilisé.
'high	Placé près d'un nom de tableau, il retourne la valeur du rang le plus haut (la valeur de retour est de type entier).
'low	Placé près d'un nom de vecteur ou tableau, il retourne la valeur de l'indice le plus bas (la valeur de retour est de type entier)
'left	Placé près d'un nom de vecteur ou tableau, il retourne la valeur de l'indice le plus à gauche (la valeur de retour est de type entier).
'right	Placé près d'un nom de vecteur ou tableau, il retourne la valeur de l'indice le plus à droite (la valeur de retour est de type entier).
'length	Retourne X'high - X'low + 1 (sous la forme d'un entier).
S'event	true ou false si le signal auquel il est associé vient de subir un changement de valeur.
S'active	true ou false si le signal auquel il est associé vient d'être affecté.
S'last_value	Retourne la valeur précédente du signal auquel il est associé.

Exemple 1 :

```
type bit is ('0', '1') ;
type boolean is (false, true) ;
```

```
type character is ('A', 'B', 'C', ..., '0', '1',..., '$',...) ;
type severity_level is (note, warning, error, failure) ;
type integer is range -2 147 483 648  to  2 147 483 647 ;
type float is range -16#0.7FFFFF8#E+32  to  16#0.7FFFFF8#E+32 ;
type time is range -9_223_372_036_854_775_808  to  9_223_372_036_854_775_807 ;
units : fs ; ps = 1000 fs ; ns = 1000 ps ; us = 1000 ns ;
ms = 1000 us ; sec = 1000 ms ; min = 60 sec ; hr = 60 min ;
end units ;
type bit_vector is array (natural range <>) of bit ;
type string is array (positive range <>) of character ;
```

Exemple 2 :

Parmi tous ces attributs prédéfinis, les plus couramment utilisés sont certainement les attributs relatifs aux caractéristiques d'un vecteur. Prenons l'exemple d'une fonction nommée FONC que l'on souhaite pouvoir appeler à partir de plusieurs instructions d'une même architecture et qui possède un paramètre de type std_logic_vecteur, la dimension est variable suivant l'instruction appelant. Pour que la dimension du paramètre vecteur s'adapte à l'appel, il faut utiliser les attributs 'left et 'right.

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

entity ex_attrib is
    port (
        in_vecteur1    : in   std_logic_vector(0 to 7);
        in_vecteur2    : in   std_logic_vector(0 to 15);
        out_vecteur1   : out  std_logic_vector(0 to 7);
        out_vecteur2   : out  std_logic_vector(0 to 15)
    );
end ex_attrib;

Architecture Arch of ex_attrib is
    function fonc (vect_A : std_logic_vector) return std_logic_vector is

        variable vect_result : std_logic_vector (vect_A'left to vect_A'right);

    begin
        for i in vect_A'left to vect_A'right loop

            if (vect_A(i) = '1') then
                vect_result(i) := '0';                  -- inverse the bit
            else
                vect_result(i) := '1';                  -- inverse the bit
            end if;

        end loop;

        return vect_result;
    end;
begin
    out_vecteur1 <= fonc (in_vecteur1);
```

```
    out_vecteur2 <= fonc (in_vecteur2);
end Arch;
```

Dans cet exemple, l'utilisation d'attributs lors de la déclaration de la variable vect_result permet d'adapter les dimensions de ce vecteur à 8 ou 16 bits suivant qu'il s'agit du vecteur1 ou du vecteur2. Ainsi, la fonction s'adapte aux dimensions du vecteur paramètre. Cette méthode est très employée au sein des packages pour généraliser un paramètre vecteur de fonction en un vecteur de dimension quelconque.

Exemple 3 :

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

entity Attributs is
    port(
        vector_dwn : in  std_logic_vector(15 downto 0);
        vector_up  : in  std_logic_vector(0 to 7);
        X          : out std_logic_vector(7 downto 0);
        Y          : out std_logic_vector(0 to 7);
        Z          : out std_logic_vector(7 downto 0));
end entity Attributs;

architecture Arch of Attributs is
begin
    X(0) <= vector_dwn(vector_dwn'left);              -- X(0) = vector_dwn(15)
    X(1) <= vector_dwn(vector_dwn'right);             -- X(1) = vector_dwn(0)
    X(2) <= vector_up(vector_up'left);                -- X(2) = vector_up(0)
    X(3) <= vector_up(vector_up'right);               -- X(3) = vector_up(7)
    X(4) <= vector_dwn(vector_dwn'high);              -- X(4) = vector_dwn(15)
    X(5) <= vector_dwn(vector_dwn'low);               -- X(5) = vector_dwn(0)
    X(6) <= vector_up(vector_up'high);                -- X(6) = vector_up(7)
    X(7) <= vector_up(vector_up'low);                 -- X(7) = vector_up(0)
    Y(vector_up'range) <= "00001111";                 -- Y(0 to 7)= "00001111"
    Z(vector_up'reverse_range) <= "00110011";         -- Z(7 to 0)="00110011"
end Arch;
```

Exemple 4 :D Flip Flop

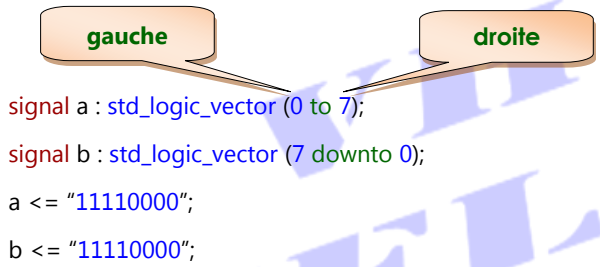
```
library ieee;
use ieee.std_logic_1164.all;

entity D_Flop is
    port (
        D, Clk, Clr, Set : in  std_logic;
        Q                : out std_logic
    );
end D_Flop;

architecture Behavior of D_Flop is
```

```
    constant Clr_it:  std_logic := '0';
    constant Set_it:  std_logic := '1';
begin
    process (Clk, Clr, Set)                    -- sensitivity list
    begin
        if (Clk='1') and (Clk'event) then      -- true on a rising edge of Clk
            if (Clr = '0') then
                Q <= Clr_it;                   -- if Clr 0 then output Q receives 0
            elsif (Set = '0') then
                Q <= Set_it;                   -- -if Set 0 then output Q receives 1
            else
                Q <= D;                        -- if neither clr nor set is active, then Q receives input D
            end if;
        end if;
    end process;
end behavior;
```

VII.1. 2. DECLARATIONS ET ASSIGNATIONS DES SIGNAUX VECTEURS



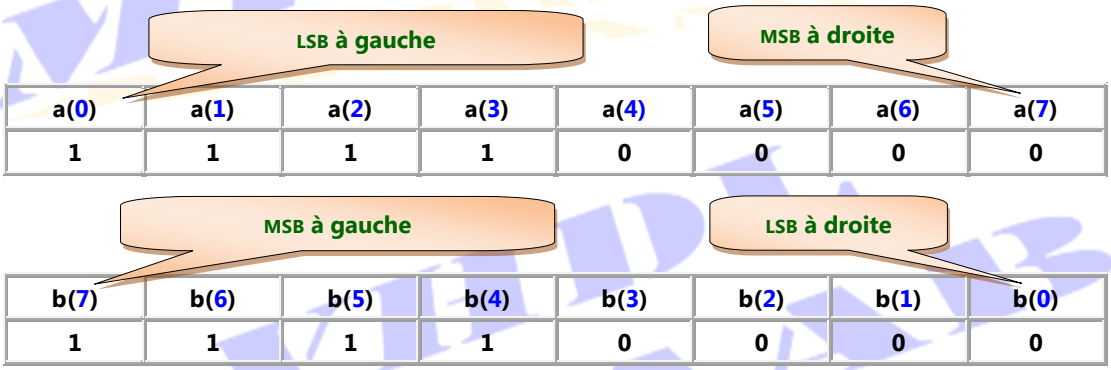


Figure II.17: Disposition des bits d'un vecteur en fonction de la déclaration

VII.1. 3.DECLARATION D'ALIAS

- Un alias est un nom alternatif (un synonyme ou un deuxième nom) d'un objet déjà existant (signal, variable, constant, ou file). *Il ne définit pas un nouvel objet*.
- Le nom de l'objet original, qui a servi à la définition de l'alias, continu à être utilisable,

- La classe de l'objet renommé n'est pas modifier. Ainsi une constant, une variable ou un signal restent respectivement une constant, une variable ou un signal
- L'alias est très utilisé pour assigné un nom spécifique à une partie d'un vecteur dans le but d'améliorer la lisibilité (voir exemple ci dessous).

Syntaxe

```
alias alias_name is name_existing_object_of_same_type;
alias alias_name: [alias_type ] is [name_existing_object_of_same_type];
```

Où :

alias_name : est le nom de l'alias,

alias_type : est le type de l'alias : très souvent c'est l'indication d'un sous type (subtype indication), voir exemple ci-dessous.

name_existing_object_of_same_type: est le nom d'un objet, qui existe déjà, qui va servir à la création de l'alias.

Exemple 1 :

```
signal   Instruction  :   Bit_Vector(15 downto 0);

alias opcode : Bit_Vector(3 downto 0)  is  Instruction(15 downto 12);  -- alias from a signal
alias source  : Bit_Vector(1 downto 0)  is  Instruction(11 downto 10);  -- alias from a signal
alias Destin  : Bit_Vector(1 downto 0)  is  Instruction(9 downto 8);    -- alias from a signal
alias ImmDat : Bit_Vector(7 downto 0)  is  Instruction(7 downto 0);    -- alias from a signal
```

Les quatre alias ci dessus dénotent quatre champs d'une instruction à savoir: le code opération, la source, la destination et la donnée immédiate supporté par certaine instructions. Notez que le nombre de bits dans le sous type qui constitue l'alias et le sous type de l'objet original est le même.

Exemple 2 :

```
signal  DataBus     :  bit_vector(31 downto 0);
alias   FirstNibble  :  bit_vector(0 to 3)  is  DataBus(31 downto 28);
```

DataBus et FirstNibble ont des directions opposées. La référence à FirstNibble(0 to 1) est équivalente à la référence DataBus(31 to 30).

Exemple 3 :

```
signal  Instruction  :  bit_vector(15 downto 0);
alias   opcode      :  bit_vector(3 downto 0) is  Instruction(15 downto 12);
. . .
if opcode = "0101" then                          -- Equivalent to if Instruction(15 downto 12) = "0101"
    . . .
```

Les deux instructions sont exactement les même, mais l'instruction qui utilise l'alias OpCode est beaucoup plus claire.

Exemple 4 :

Si on travaille avec des mots de 16 bits dont le bit de signe est le bit numéro 0, il sera plus agréable de déclarer :

```
variable  Word   :  bit_vector(0 to 15);
alias       signe  :  bit    is   Word(0);                    -- alias from a variable
```

Exemple 5 :

```
entity exemple_alias is
    port(
        Bus : in std_logic_vector(15 downto 0)
    ) ;
end exemple_alias ;

architecture ARC_ exemple_alias is
    alias Bus_Low      :  std_logic_vector          is   Bus(7 downto 0) ;
    alias Bus_Hight    :  std_logic_vector          is   Bus(15 downto 8) ;
    alias Bus_Middle   :  bit_vector(4 downto 0)    is   Bus(9 downto 5);
    alias Bit_Low      :  std_logic                 is   Bus(0);
    alias Bit_Middle   :  std_logic                 is   Bus(8);
    ..................................
    -- suite de l'architecture
    ..................................
end ARC_exemple_alias
```

Dans cette exemple, manipuler bus_low (bus_hight) revient à manipuler les huit bits du poids faible (fort) du signal bus, et manipuler bit_middle revient à manipuler la composante 9 du vecteur bus etc…

- L'operateur alias permet donc, entre autres, d'accèdes à des sous-éléments d'un vecteur.

VII.2. LES VARIABLES (variable)

VHDL est un langage basé sur l'utilisation des signaux. Cependant l'utilisation uniquement de ces signaux présente certaines limites (inconvénients) :

- Les signaux maintiennent uniquement la dernière valeur qui leur est assignée ;
- Les nouvelles valeurs ne sont pas affectées aux signaux lors de l'exécution des instructions d'affectation, mais uniquement après la suspension (wait) de l'exécution du processus.

Pour contourner ces limites et se permettre le stockage des résultats intermédiaires, VHDL propose les variables (variable).

- A l'exception des variables partagées (shared variable), qui peuvent être partagées par plusieurs process, les variables ne peuvent être déclarées et utilisées qu'à l'intérieur:
  - d'un processus (process): elles sont alors ***locales*** *à ce process*
  - un sous programmes (function, procedure): elles sont ***locales*** *à ce sous programme* et elles sont initialisées à chaque fois que le sous programme est appelé

☞ Elles ne peuvent être déclarées ou utilisées directement dans une architecture.

VII.2.1. LES VARIABLES SIMPLE

Comme un signal, une variable doit être déclarée avant son utilisation. La déclaration de la variable est similaire à celle d'un signal, mais le mot clé «variable » est utilisé, selon la syntaxe suivante :

Syntaxe

```
variable Variable_Identifier { , Variable_Identifier} : type [:= Value];
```

Cette déclaration spécifie :

- Le(s) nom(s) de(s) la varaible(s) : Variable_Identifier
- Le type de la variable : type
- Facultativement, pour les variables scalaires, une plage contrainte peut être spécifiée, et pour les variables de type tableau, un index contraint peut être spécifié. Pour les deux types de variables, une valeur initiale peut être spécifiée.
- On peut opérer une initilaisation explicite pendant la création de la variable :

```
variable   X : integer := 5;                    -- Déclaration avec initialisation
```

- Si l'initialisation explicite n'est pas exécutée, la variable sera implicitement initialisée à la valeur la plus à gauche de son type. Ci-dessous, "Y" est implicitement initialisée à 1 (valeur la plus à gauche du type positif) et "Z" est implicitement initialisée à 0 (valeur la plus à gauche du type real)

```
variable    i   : integer range 0 to 3;      -- initialization according to the domain therefore at 0
variable  mem : bit_vector (1 to 15);       -- initialization according to the domain therefore at 1
variable   Y    : positif;                  -- Declaration without initialization: By default the
                                            -- variable is initialized to the left value of its type . So
variable  Z     : real;                     -- « Y » and « Z » are implicitly initialized to 1 and 0
                                             -- respectively
```

Exemple 1 **:** variable declared inside a function

```
function Parity (X : std_ulogic_vector) return std_ulogic is
    variable TMP : std_ulogic := '0';                    -- initialized to 0 for each call to the function parity
begin
      for J in X'range loop
          tmp := tmp xor X(J);
      end loop;
    return tmp;                                          -- if tmp i=0 then parity is even (paire)
end Parity;
```

- Les variables dans les process, hormis pour "for loop", sont initialisées au début de la simulation (time = 0 nS):

Exemple 2 : variable declared within a process

```
process (A)
  variable tmp: std_ulogic := '0';                       -- The variable tmp is initialized each time the process is
begin                                                    -- activated
  tmp:= '0';
  for I in A'low to A'high loop
    tmp := tmp xor A(I);
  end loop;
  odd <= tmp;
end process;
```

- L'affectation d'une variable se fait avec l'opérateur «:=». L'affectation de la valeur d'une variable B à une variable A s'écrit : A := B
- La liaison entre deux variables est temporaire ce qui signifie que si la valeur de B change ensuite, celle de A est inchangée.
- Contrairement aux signaux, *les variables sont mises à jour de façon instantanée*, au moment de l'affectation, sans retard possible.

VII.2.2. UTILITE DES VARIABLES

Comme c'est déjà mentionné plus haut, le plus grand atout des valaurs est qu'elles servent à manipuler des valeurs temporaires/intermédiaires pour faciliter le développement d'un algorithme séquentiel. Pour illustrer ce rôle, extrèmement important, considérons les deux exemples suivants.

Le premier utilise des variables pour stocker des résultats intermédiaires.

Exemple 1 : : utilisation des variables

```
entity importance_variable is
    port (
        a, b, cin  : in    bit;
```

```
        s, cout    : out bit
    );
end importance_variable;

architecture arc of  importance_variable is
begin
    process (a, b, cin)
        variable S1, S2, C1, C2: bit;
    begin
        S1 :=  a   xor   b;
        S2  :=  S1   and cin;
        S1 :=  S2  or    b;
        C1  :=  A   and   b;
        S2 :=  S1  xor   cin;
        C2 :=  S1  and   cin;
        S   :=  S2;
        cout <= c1 or c2;
    end process;
end arc;
```

La synthèse de ce programme donne le circuit ci desous:

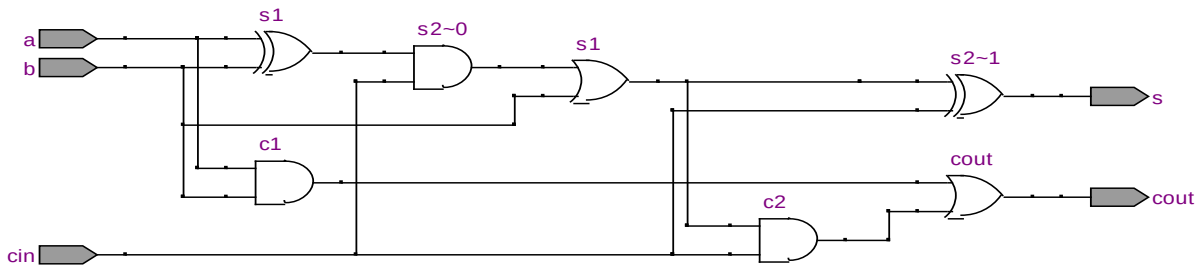


Figure II.18: Synthèse par utilisation des variables

Notez bien que le circuit résultant reflète à la lettre l'exécution séquentielle des instructions du programme: Si une varaible est utilisée dans deux instructions consécutives, elle sera synthétisée deux fois.

Considérons à présent le même exemple par utilisation des signaux au lieu des varaibles.

Exemple 2 : Use of signals

```
entity importance_variable is
    port (
        a, b, cin  : in   bit;
        S, cout    : out bit
    );
end importance_variable;

architecture arc of  importance_variable is
```

```
    signal  S1, S2, C1, C2: bit;
begin
    process (a, b, cin)
    begin
1.      S1 <=   a   xor   b;
2.      S2  <=  S1  and   cin;
3.      S1 <=   S2  or    b;
4.      C1  <=  a   and   b;
5.      S2 <=   S1  xor   cin;
6.      C2  <=  S1  and   cin;
7.      S     <=   S2;
8.    cout <= C1 or C2;
    end process;
end arc;
```

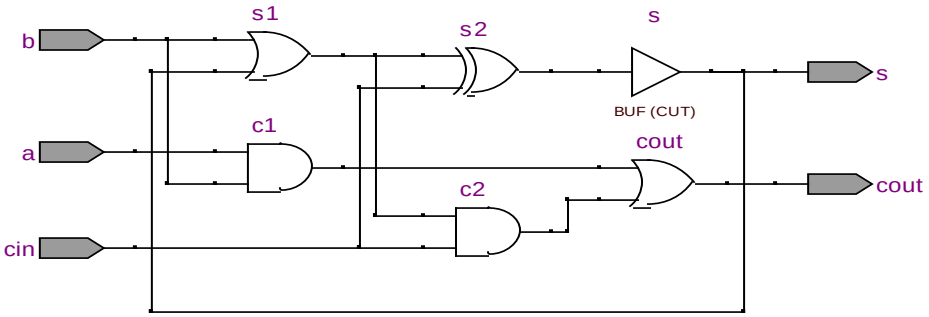


Figure II.19: Synthèse par utilisation des signaux

Le résultat de la synthèse est un circuit complètement différent de celui où les varaibles sont considérées et le rôle, extrèmement important, que joue la notion de variable est bien mis en valeur :

- dans le premier circuit, la synthèse a donnée naissance à deux neuds aux noms de « S1 » et deux neuds aux noms de « S2 », exactement comme le décrit l'exécution séquentielle du processus;
- dans le deuxième circuit cependant, la mise à jour des signaux est faite après l'exécution de toutes les instructions du processus, avec en plus si plusieurs assignements ont eu lieu pour le même signal, c'est uniquement le dernier assignement qui sera considéré. Ainsi dans la suite chronologique considérée: entre les instructions 1 et 3, qui affecte le même signal S1 , c'est la dernière instruction `S1 <= S2 or b;` qui sera considérée et entre les instructions 2 et 5, qui affecte le même signal S2, c'est la dernière instruction `S2 <= S1 xor cin;` qui sera considérée

VII.2.3. LES VARIABLES PARTAGEES (shared variable)

Les variables partagées répondent à la question **Comment utiliser une variable dans plusieurs** process **ou dans plusieurs sous programmes (**function et procedure**) ?** Dans sa version originale, VHDL'87, VHDL limite la portée (scope) d'une variable au process, function ou procedure où elle est déclarée. Les signaux sont le seul moyen de communiquer entre process. Par la suite VHDL'93, introduit cependant les variables partagées (shared variable) qui peuvent être utilisées par plusieurs process, function ou procedure. Comme une variable ordinaire, l'assignement (l'affectation) d'une variable partagée prend effet immédiatement. Cependant, il faut faire attention pendant l'utilisation des variables partagées parceque si plusieurs process procèdent à l'assignation de la même variable, de façon concurrentielle, cela peut conduire à un comportement imprévisible. VHDL'93 ne définit pas la valeur d'une shared variable si plusieurs process procèdent à l'assignation de celle-ci dans le même cycle de simulation. Il faut retenir donc que :

- Les variables partagés sont déclarées à l'extérieur des process, et sous programmes (function, procedure) et elles peuvent être partagées par ceux ci.
- La syntaxe de déclaration d'une variable partagée est la même que celle d'une variable ordinaire à qui il faut ajouter le mot clé shared devant le mot clé variable :

Syntaxe

```
shared variable Variable_Identifier {, Variable_Identifier } : type [:= Value];
```

Exemple :

```
architecture behave of Shared_Variable is
    shared variable S : integer;
begin
    process (A, B)
    begin
        S := A + B;
    end process;
    process(A, B)
    begin
        S := A - B;
    end process;
end behave;
```

Dans cet exemple, la variable **S** est déclarée au niveau de l'architecture exactement comme un signal et elle est utilisée (partagée) par les deux process de cette architecture.

- Les mêmes règles qui régissent l'initialisation des variables non partagées restent valables pour les varaibles partagées ainsi :

  shared variable FreeAccess : Boolean := true;.

  Crée une varaible partagée au nom FreeAccess de type Boolean et l'initalise explicitement à true, et :

  shared variable Counter : positif;

  Crée une varaible partagée au nom Counter et l'initalise implicitement à la valeur la plus à gauche du domaine donc à 1.

- Les variables partagées (shared variable) ne sont pas synthétisables.

VII.2.3.1. OU PEUT-ON DECLARER UNE SHARED VARIABLE ?

Si elle est déclarée à l'intérieur d'un process, une function ou une procedure, la shared variable sera locale à l'endroit où elle est déclarée. Le seul moyen pour la rendre accessible par tous est de la déclarer en dehors de tous. On peut donc la déclarer dans :

- Un package,
- La zone déclarative d'une architecture

Déclarée ainsi, une shared variable peut être accédée en lecture/écriture par plusieurs proccess sans restriction, faite attention cependant parceque le standard VHDL'93 ne définit pas le résultat d'un accès concurretiel, par deux process, sur la même shared variable comme c'est déjà mentionné.

Exemple :

```
Architecture SV_example of example is
    shared variable Shared_Variable : bit;                    -- Shared_Variable is a  shared variable
begin
    P1 : process (Clock, In1)
    begin
        if(Shared_Variable) then                              -- use of the Shared_Variable by the first process P1
                ....
        end if;
        Shared_Variable:='0';
    end process P1;
    P2 : process (Clock, In2)
    begin
        if(!(Shared_Variable)) then                           -- use of the Shared_Variable by the second process P2
                ....
        end if;
        Shared_Variable:='1';
```

```
    end process P2;
end SV_example;
```

VII.3. ENTRE SIGNAL ET VARIABLE

En VHDL, les signaux (signal) et les variables (variable) sont très similaires. Ils peuvent être utilisés tous les deux pour le maintien de tout type de données qui leur est affectée.

Le problème est d'éviter la confusion sur la différence entre le comportement d'une affectation de signal et celui d'une variable pendant l'exécution d'un process.

☞ Rappelez-vous qu'une affectation de signal, si tant est qu'il en soit un, programme simplement un événement survenant sur ce signal et n'a pas d'effet immédiat. Lorsque le processus est repris, il s'exécute de haut en bas et aucun événement n'est traité jusqu'à la fin du processus.

Cela signifie que si un événement est programmé sur un signal lors de l'exécution d'un process, cet événement peut être traité après la fin du processus au plus tôt. Examinons un exemple de ce comportement. Dans le processus suivant, deux événements sont programmés sur les signaux X et Z.

```
...
signal X, Y, Z : bit;
...
process (Y)
begin
    X <= Y;
    Z <= not X;
end process;
```

Si le signal Y change, un évènement sera programmé sur X pour le rendre identique à Y. De plus, un événement est programmé sur Z pour en faire le contraire de X. La question est de savoir si la valeur de Z sera l'opposé de Y. Bien sûr, la réponse est non, car lorsque la deuxième instruction est exécutée, l'évènement sur X n'a pas encore été traité et l'évènement planifié sur Z sera l'opposé de la valeur de X avant le début du processus.

Ceci est souligné parce que ce n'est pas nécessairement le comportement intuitif et que les variables fonctionnent différemment. Par exemple, dans :

```
process (Y)
    variable X, Z : bit;
begin
    X:=Y;
    Z:=not X;
end process;
```

La valeur de la variable "Z" serait l'opposé de la valeur de "Y" car la valeur de la variable "X" est modifiée immédiatement.

Les points suivants sont à garder toujours présent à l'esprit :

- Une variable ne peut être ***utilisée*** (et même ***déclarée***) qu'a l'intérieur d'un process, function ou procedure. Un signal peut par contre être ***utilisé*** (et non déclaré) à l'intérieur ou à l'extérieur d'un process, function ou procedure.
- Une variable déclarée dans un process, function ou procedure, ne peut être utilisée qu'à l'intérieur de ce process, function ou procedure. Un signal peut par contre être utilisé dans plusieurs process, function ou procedure. Garder à l'esprit cependant que ce signal ne peut être affecté que dans un seul process.
- Les variables sont assignées par l'utilisation de l'opérateur «**:=**» alors que les signaux sont assignés par l'utilisation de l'opérateur «<=»
- *Une variable ne peut exister que dans un contexte* séquentiel, *elle est affectée immédiatement*. Elle ne peut être déclarée et elle n'est visible qu'à l'intérieur d'un process, function ou procedure

```
X := 1 + 2;                                    -- x prend la valeur 3
```

- Le signal est le seul objet qui peut être affecté soit de façon concurrente, soit de façon séquentielle selon le contexte. Il dépend du code, combinatoire ou séquentiel, qui l'utilise. Il est l'agent de communication entre processus.

Différence dans les affectations

Les exemples suivants illustrent ce point fondamental.

Exemple 1 :

```
entity varsig is
end varsig;

architecture exercise of varsig is
    signal  AA, AAA   :  integer := 3;
    signal  BB, BBB   :  integer := 2;
begin
    P1: process
        variable A : integer := 7;
        variable B  : integer := 6;
    begin
        wait for 10 ns;
        A := 1;                                -- A takes the value 1
        B := A + 8 ;                           -- B takes the value 9
```

```
        A := B - 2 ;                    -- A takes the value 7
        AA <= A;                        -- 7 in driver of AA
        BB <= B ;                       -- 9 in driver of BB
    end process P1;
```

A l'heure H = 10 ns , AA prend la valeur 7 , BB prend la valeur 9. Entre 0 et 10 ns AA vaut 3 et BB vaut 2

```
    P2: process
    begin
        wait for 10 ns;
        AAA <= 1 ;                      -- 1 in driver of AAA
        BBB <= AAA + 8;                 -- 11 in driver of BBB
        AAA <= BBB - 2;                 -- 0 in driver of AAA
    end process P2;
end exercise ;
```

A l'instant t = 10 ns, AAA prend la valeur 0, BBB prend la valeur 11. Entre 0 et 10 ns, AAA vaut 3 et BBB vaut 2;

Remarque:

De deux affectations successives d'un même signal seule la deuxième compte. Le pilote du signal constitue une mémoire associée au signal.

Exemple 2 :

```
entity toto is
end toto;

architecture Var of toto is
    signal Trigger, Sum : integer := 0;
begin
    process
        variable Var1 : integer := 1;
        variable Var2 : integer := 2;
        variable Var3 : integer := 3;
    begin
        wait on Trigger;
        Var1 := Var2 + Var3;
        Var2 := Var1;
        Var3 := Var2;
        Sum <= Var1 + Var2 + Var3;
    end process;
end Var;
```

Si Trigger change à t=10 ns, alors entre 0 et 10 ns Var1=5, Var2=5, Var3=5 et à t=10+Δt, Sum=15

VII.4. LES CONSTANTES (constant)

Les constantes sont utilisées pour référencer une valeur ou un type spécifique. Elles permettent une meilleur lecture et maintenance d'une source. Une constante doit être déclarée avant son utilisation dans:

- Un package : elle est alors globale ;
- Une entity : elle est alors commune *à toutes les architectures de l'entité* ;
- Une architecture : elle est alors locale *à cette architecture*;
- Un process : elle est alors locale *à ce process*

- Les constantes reçoivent leur valeur au moment de leur déclaration. L'affectation se fait avec l'opérateur «:=»

Syntaxe

```
constant Constante_Identifier { , Constante_Identifier} : type := value;
```

Exemple :

```
architecture arc_constante of entity_constante is
    constant   VCC    : std_logic := '1';
    constant   GND    : std_logic := '0';
begin
    ..........................................
    -- Corps de l'architecture
    ..........................................
end arc_constante;
```

VIII. LES FICHIERS file

Les fichiers sont des objets utiles qui peuvent être utilisés pour la simulation ou la gestion des fichiers test benches. En plus, les résultats de la simulation peuvent également être enregistrés dans des fichiers. Leur comportement est similaire à celui où ils sont utilisés dans les langages de programmation tel que le C,

- En VHDL, les fichiers sont traités comme des tableaux de lignes.
- Tous les mots clés de gestion des fichiers ne sont pas synthétisables. Ils sont à utiliser uniquement pour la simulation des tests benches.
- Pour lire/écrire dans un fichier, il faut inclure le package **std.textio** (use std.textio.all). Cette bibliothèque, contient un ensemble d'instructions de lecture/écriture dans un fichier. On en trouve: file_open, file_close, read, write, readline, writeline, flush, endfile, et d'autres.

IX. LES TYPES D'OBJETS

IX.1. LES TYPES PREDEFINIS ET LES OPERATEURS ASSOCIES

Considérons la déclaration :

```
S : out bit;
```

Cette déclaration signifie que le nom du signal est **S**, que sa direction est **out** (sortie) et qu'il est de type **bit**. Nous allons, au cours de ce paragraphe, nous intéresser tout particulièrement aux types d'un signal.

Le type d'un signal définit l'ensemble auquel appartient ce signal au même titre que l'on dit du nombre 3 qu'il appartient à l'ensemble des entiers. Le langage VHDL initial propose des types prédéfinis, c'est-à-dire des types pour lesquels l'ensemble est clairement défini. Voici les types prédéfinis proposés par le langage VHDL initial :

Tableau II.6: Type prédifinis et ensembles associés

Nom du type	Définition de l'ensemble
bit	Deux valeurs possibles '0' ou '1'
Bit_vector	composite tableau représentant un vecteur de bits
Integer	Entiers (nombre positif ou négatif sur 32 bits)
natural	entier positif ou nul
positive	entier positif
real	Réels (flottant) compris entre -1.0E38 et 1.0E38
boolean	énuméré dont les deux valeurs possibles sont false et true
character	Caractères a, b, c ..., 1, 2 ...
time	Nombre réel de temps fs, ps ..., min, hr

On peut donc déclarer un signal de la façon suivante :

```
S : out integer;
```

Dans ce cas précis, le signal **S** est de type entier et peut, par conséquent, prendre l'une des valeurs de l'ensemble des nombres entiers. On peut alors s'interroger sur l'existence d'opérateurs capables de gérer tous ces différents types. En effet, si l'opération `A or B` semble compréhensible lorsque les signaux `A` et `B` sont de types bit, elle devient stupide lorsque les signaux sont de types entiers, sans parler du cas où les signaux sont de types différents.

En fait, le langage VHDL initial propose avec ces types prédéfinis des opérateurs dédiés capables de faire la somme, la soustraction d'entiers ou de réels, etc. Notons que, en l'absence

de fonctions ou de procédures créées par l'utilisateur, il n'existe pas d'opérateur prédéfini capable de manipuler des signaux de types différents.

- **Le langage VHDL initial propose des types prédéfinis ainsi que des opérateurs associés que l'utilisateur peut utiliser au sein des descriptions.**

On peut donc écrire :

A + B ➔ avec A et B de type entiers et "+" une fonction d'addition entre entiers.

A + B ➔ avec A et B de type réels et "+" une fonction d'addition entre réels.

Dans ces deux exemples, les fonctions "+" d'addition ont le même nom mais ne sont pas identiques, car elles concernent des types distincts.

En plus de ces types prédéfinis, VHDL permet de définir d'autres types utilisateur auquel on peut assigner un objet.

IX.2. LE TYPE SCALAIRE

Deux catégories sont groupées sous cette nomination :

IX.2.1. LE TYPE NUMERIQUE

Cette définition de type doit être dans une échelle de valeur. La base par défaut est la décimale.

Syntaxe

```
type identifier is  range implementation_defined;
```

Exemple :

```
type      byte      is   range 0  to  255;          -- byte varies from 0 to 255
type      index     is   range 7  downto 0;         -- index varies from 7 to 0
subtype   byte      is   integer  range 0 to 255;   -- byte is an integer from 0 to 255
```

IX.2.2. LE TYPE ENUMERATION

Ces types sont utilisés lorsque l'on souhaite créer un nouveau type de signal ou lorsque l'on souhaite créer un signal ou une variable dont les valeurs possibles sont explicites. La déclaration d'un nouveau type peut avoir lieu dans une entity, une architecture ou un process.

Syntaxe

```
type type_name is (element_1, element_2, ...);
```

où element_1, element_2 etc constituent la liste du type

```
type COLOR is (RED, GREEN, YELLOW, BLUE, PURPLE);
```

Le type COLOR définit cinq literal associés, dans l'ordre, à cinq valeurs dans le domaine 0-4. RED a le numéro 0, GREEN a le numéro 1, YELLOW a le numéro 2, BLUE a le numéro 3 et en fin PURPLE (VIOLET) a le numéro 4.

Exemple d'utilisation

```
type COLOR is (RED, GREEN, YELLOW, BLUE, PURPLE);
type_color  COLOR;
process (COLOR)
    case COLOR is
        when RED =>
            next_color <= GREEN;
        when YELLOW =>
            next_color <= RED;
        when GREEN =>
            next_color <= YELLOW;
    end case;
end process;
```

IX.2.3. LES TYPES COMPOSITES

Ca concerne les tableaux (array) et les enregistrements (record)

IX.2.3.1. LES TABLEAUX array

Un tableau est un ensemble d'éléments homogènes (de même type). Le langage VHDL autorise la définition de tableau de valeurs de type quelconque.

Il y a deux sortes de tableaux :

1. **Les tableaux contraint** (Constrained Array): Ces tableaux sont définis avec des dimensions figées (rangées colonnes).

Syntaxe

```
type Array_Identifier is array (Array_Dimension) of Array_type_elements;
```

avec:

Array_Identifier : le nom du tableau,

Array_Dimension : la dimension du tableau,

Array_type_elements : le type des éléments du tableau (bit, integer, real, ...)

Exemples :

```
type Tableau_vecteur_logic is array (1 to 4)          of std_logic_vector(2 downto 0);
type Tableau_entiers       is array (7 downto 0)      of integer ;
type mot                   is array (15 downto 0)     of bit;
type memoire               is array (0 to 255)        of mot;
```

L'utilisation ultérieure de ces nouveaux types peut être:

```
constant   const4 : tableau_vecteur_logic :=  ("000", "011", "100", "100");          -- ou
constant   const4 : tableau_vecteur_logic :=  (('0','0','0'), ('0','1','1'), ...);
signal     Tableau : tableau _entiers      :=  (-3, 7, 11, 5, -2, 58, 42, 19) ;
```

Le signal Tableau comportera donc 8 éléments de type entier que l'on pourra manipuler individuellement ou tous en même temps.

Pour la déclaration de tableaux à plusieurs dimensions, on peut procéder ainsi:

2. Les tableaux non contraint (Unconstrained Array):

- Ces tableaux sont définis sans préciser les dimensions. Le tableau est donc dynamique et la définition de la taille est repoussée à plus tard.
- La manipulation de ce type de tableaux demande l'utilisation d'attributs **range <>**

Syntaxe

```
type  Array_Identiffier   is   array (Arra_type_Index range <>)  of   Array_type_Elements;
```

avec:

Array_Identiffier : le nom du tableau,

Arra_type_Index : le type des indices du tableau, par exemple :

Arra_type_Index = positive ⇒ les indices du tableau sont dans le domaine 1...

Arra_type_Index = natural ⇒ les indices du tableau sont dans le domaine 0...

Array_type_Elements : le type des éléments du tableau (bit, integer, real, ...)

Exemples :

```
type    string          is   array (natural  range <>)   of   character;
type    bus             is   array (natural  range <>)   of   bit;
type    Integer_Array   is   array (positive range <>)   of   integer;
type    Array_bit_vector is  array (natural range <>)    of   bit_vector(2 downto 0);
```

string, bus et Integer_Array sont trois nouveaux objets de type tableau (array) de dimensions non encore définit. Ces déclarations fournissent les informations suivantes :

Pour le nouveau type string : Ses indices sont de type natural, c'est-à-dire qu'ils peuvent être dans le domaine 0,1,.... Ses éléments sont de type caractère ;

Pour le nouveau type bus: Ses indices sont également de type natural, c'est-à-dire qu'ils peuvent être dans le domaine 0,1,.... , cependant, ses éléments sont de type bit ;

Pour le dernier type Integer_Array: Ses indices sont de type positive, c'est-à-dire qu'ils peuvent être dans le domaine 1,.... Alors que ses éléments sont de type entier ;

C’est l’utilisation ultérieure de ces nouveaux types qui en définira les dimensions. Par exemple; les trois lignes suivantes:

```
variable  String_Array    :  string(0 to 15);          -- Is an array (string) of 16 characters
variable  Bits_Array      :  bus (15 downto 0);        -- Is an array (bus) of 16 bits
variable  Integers_Array  :  Integer_Array(1 to 7);
```

Créent trois variables en se servant des nouveaux types qui viennent d’être crées. Ces déclaration fixent en même temps les dimensions des tableaux de chacune des variables qui vient d’être crée.

String_Array : est une variable de type string, donc un tableau de caractères. Ses indices sont de type natural et vont de 0 à 15. Sa dimension est donc fixée à 16 ;

Bits_Array : est une variable de type bus, donc un tableau de bits. Ses indices sont de type natural et vont de 15 à 0. Sa dimension est donc fixée à 16 ;

Integers_Array : est une variable de type Integer_Array, donc un tableau d’entiers. Ses indices vont de 1 à 7. Sa dimension est fixée à 7 ;

La ligne suivante crée une constante en se servant du dernier type crée.

```
constant   const    :  Array_bit_vector (1 downto 0) := ("000", "111");     -- or
constant   const    :  Array_bit_vector (1 downto 0) := (('0','0','0'), ('1','1','1'));
```

Affectation des éléments d’un tableau

a. par défaut : association "positionnelle"

```
type Integers_Array is array (0 to 4) of integer;
....
Integers_Array:= (5, 3, 1, 7, 9);
```

b. association par nom

```
Integers_Array:= (1=>3, 4=>9, 0=> 5, 2=>1, 3=>7 );
```

c. association mixte

```
Integers_Array:= (4=>9, 0=> 5, others => 0);          -- Equivalent to Integers_Array := (5, 0, 0, 0, 9);
```

- les champs affectés par others doivent être du même type

Pour créer un tableau à plusieurs lignes, on precède comme suit:

Exemple 1

```
type Array_8xbit      is  array (7 downto 0) of bit ;
type Array _4x8xbit   is  array (3 downto 0) of Array_8xbit;
```

ou encore :

```
type Array _4x8xbit   is   array (3 downto 0, 7 downto 0) of bit ;
```

Exemple 2

```
type type1 is   array (positive range <>) of integer;
type type2 is   array (0 to 3) of natural;
type type3 is   array (1 to 2) of type2;                --see type2 above
constant const1: type1(1 to 4) := (5, -5, 3, 0);
constant const2: type2 := (2, 0, 9, 4);
constant const3: type3 := ((5, 5, 7, 99), (33, 4, 0, 0));
```

IX.2.3.2. LES ENREGISTREMENT record

Un record est un ensemble de signaux (éléments) de type différents regroupés sous un même nom. Ces éléments peuvent être de types prédéfinis ou de types défini par l'uilisateur. Les éléments ainsi juxtaposés pourront, par la suite, être accessibles grâce à l'utilisation du ".". C'est l'équivalent des structures (struct) dans les langages de programmation évolués tel que le C. Ca permet l'écriture du code plus claire et facile à maintenir.

Syntaxe

```
type  record_type_name  is   record
    Identifier_element_1  :  type element_1;
    Identifier _element_2 :  type element_2;

    suite des déclarations du record

end record ;
```

avec:

Identifier _element_1 : le nom de l'élément 1 du record,
type element_1 : le type de l'élément 1 du record,
Identifier _element_2 : le nom de l'élément 2 du record,
type element_2 : le type de l'élément 2 du record
etc

Exemple 1 :

```
type Bus_micro is record
    nbr_bit_adresse   :integer ;
    nbr_bit_donnee    :integer ;
    vitesse           :time ;
end record ;
```

Si plusieurs éléments sont de même type, ils peuvent être déclarés ensemble (exemple ci-dessous).

Exemple 2 :

```
type  RegName   is   (AX, BX, CX, DX);
type Operation is record
    Mnemonic       :  string (1 to 10);
    OpCode         :  bit_vector(3 downto 0);
    Op1, Op2, Res  :  RegName;                    -- three elements, of the same type, declared together
end record;
```

AFFECTATION DES CHAMPS D'UN RECORD

a. PAR DEFAUT : "positionnelle"

```
type article is record
    champs1 : integer;
    champs2 : bit;
    champs3 : integer;
end record;
.....
article := (12, '1', 9);
```

Le type Operation est declaré dans l'exemple 2 ci-dessus :

```
variable Instr1, Instr2 :  Operation;
. . .
Instr1:= ("ADD AX, BX", "0001", AX, BX, CX);
Instr2:= ("ADD AX, BX", "0010", others => BX);
```

b. PAR NOM

```
article := (champs2=>'1',champs3=> 9, champs1=> 12);
```

c. MIXTE

```
article := (champs2 =>'1', others => 0);          -- Equivalent to article := (0, '1', 0);
```

- les champs affectés par others doivent être du même type
- On peut affecter certains champs uniquement par utilisation de l'opérateur d'adressage '.' :

Le type Operation est declaré dans l'exemple 2 ci dessus

```
variable Instr3 : Operation ;
. . .
Instr3.Mnemonic := "mul AX, BX";
Instr3.Op1 := AX
```

IX.3. LES POINTEURS (access)

Les pointeurs permettent de manipuler des données, crées dynamiquement pendant la simulation et dont la taille exacte n'est pas connue à l'avance. Toute référence à celles-ci est effectuée via des allocateurs (allocators), qui fonctionnent de la même manière que les pointeurs dans les langages de programmation.

Bien que les types access soient très utiles pour modéliser des structures potentiellement volumineuses, telles que des Mémoires ou les structures FIFO, ils ne sont pas pris en charge par les outils de synthèse.

On peut créer des pointeurs vers tous les types: scalar, composite ou même vers d'autres pointeurs.

☞ Le mot clé access sert à créer ces pointeurs vers des objets, et

☞ les mots clés new et deallocate permettent d'allouer respectivement désallouer la mémoire à ces pointeurs. La syntaxe de déclaration d'un pointeur est :

Syntaxe

```
type Pointer_Identifier is access Object_Identifier ;
```

Avec:

Pointer_Identifier : l'identificateur du pointeur;

Object_Identifier : l'identificateur de l'objet sur lequel pointe le pointeur;

- Le type access peut être:
  - Utilisé uniquement *en mode séquentiel*;
  - *Uniquemnt les* variables peuvent être de ce type

Exemple :

```
type Name         is  string(15 downto 0);
type Born_Day     is  integer range 1 to 30;
type Born_Month   is  integer range 1 to 12;
type Born_Year    is  integer range 2000 to 2020;
type Student is record
    Student_Name        : Name;
    Student_Born_Day    : Born_Day;
    Student_Born_Month  : Born_Month;
    Student_Born_Yaer   : Born_Year ;
end record;
type Student_Pointer is access Student ;
```

A partir de cette ligne Student_Pointer est un nouveau type, pointeur sur les objets de type Student de type record.