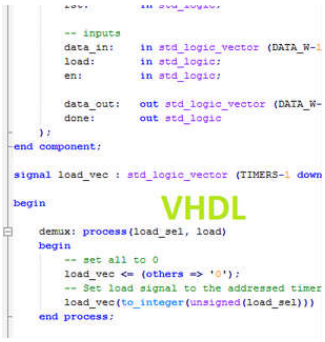
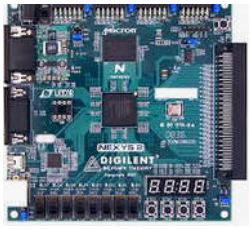
UNIVERSITE MED SEDDIK BENYAHIA - JIJEL
FACULTE DE SCIENCES ET DE TECHNOLOGIE

Master Electronique et Automatique

VHDL et circuits programmables (FPGA) : Cours, TD et TP





Préface

Ce polycopié s'adresse aux étudiants Master (LMD) Electronique/automatique, pour apprendre la simulation et l'implémentation des schémas électroniques numériques en circuits programmables FPGAs (Filed Programmable Gate Arrays). Le langage en question est le langage de description matériel VHDL. En effet, ce manuscrit vise à familiariser l'étudiant avec le langage VHDL, les circuits FPGAs et les outils de simulation ISE et System Generator XSG (Matlab/Simulink).

Les méthodes de conception en matière de composants programmables ont été révolutionnées par l'apparition du langage VHDL**.** En effet, celui-ci représente un intérêt majeur quant à la portabilité des designs et à leurs évolutions. La synthèse logique permet de réaliser l'implémentation physique d'un design, seul un sous-ensemble du jeu d'instructions du VHDL est utilisé. La simulation permet de vérifier le comportement d'un design avant ou après implémentation dans le composant ciblé. Donc c'est une étape indispensable qui vous fera gagner du temps de mise au point sur une carte.

Les langages de description du matériel doivent fournir des constructions pour décrire les attributs d'un design spécifique, etc Les simulateurs utilisent ces descriptions pour «imiter» le comportement du système physique. Les compilateurs de synthèse utilisent de telles descriptions pour synthétiser du matériel manufacturable conforme à une spécification donnée

Les cartes FPGA Digilent disponibles sur le site www.digilentinc.com : la carte FPGA Basys ™ 2 Spartan 3E, la carte FPGA Nexys ™ 2 Spartan-3E, la carte FPGA Nexys ™ 3 Spartan-6, la carte FPGA Nexys ™ 4 Artix-7 ou la carte FPGA Basys3 ™ Artix-7 sont discutées dans ce document. Pour simuler et synthétiser vos conceptions sur une carte FPGA Xilinx, vous devez tout d'abord téléchargé le WebPACK gratuit de Xilinx, Inc. (www.xilinx.com). Chaque carte FPGA aura besoin de son fichier de contraintes utilisateur unique (.ucf), qui identifie les numéros de broche spécifiques pour une carte particulière. Vous pouvez télécharger ces fichiers .**ucf** à partir de http://www.lbebooks.com/downloads.htm.

Ce document est devisé en quatre parties :

- **Cours : Partie 1 - Programmation en langage VHDL**
- **TD : Exercices avec correction**
- **Cours : Partie 2 - Circuits FPGAs**
- **TP et applications**

Contents

Partie I : Cours

PROGRAMMATION EN LANGAGE VHDL

I. 1. INTRODUCTION

VHDL signifie VHSIC Hardware Description Language (VHSIC : Very High Speed Integrated Circuit). Ce langage a été transcrit dans les années 70 pour réaliser la simulation de circuits électroniques. Ensuite il a été développé pour permettre la conception (synthèse) de circuits logiques programmables comme PLD (Programmable Logic Device). 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) intégrée dans les PLDs 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 PLDs. Ils permettent au code écrit d'être portable, c'est à dire qu'une description écrite pour un circuit peut être facilement utilisée pour un autre circuit.

Il faut avoir à l'esprit que ces langages dits de haut niveau permettent de matérialiser les structures électroniques d'un circuit. En effet 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 PLDs, FPGA etc[1].

I.2. HISTORIQUE

VHDL et Verilog sont des concurrents de la première heure, bien que Verilog fût le premier sur le marché [2] (vers 1983-1984, par Gateway Design Automation, société qui fut rachetée par Cadence) - il s'agissait d'un standard fermé, certain allant jusqu'à dire que cette fermeture était la seule raison de la création de VHDL. Dès 1983, IBM, Texas Instruments et Intermetrics ont commencé à développer le VHDL ; sa version 7.2 est

sortie en 1985 et, deux ans après, fut standardisée par l'IEEE en tant que 1076-1987 (un an après en avoir reçu les droits). En tant que standard IEEE, il doit être revu périodiquement, tous les cinq ans au plus (actuellement, les révisions 1993, 2000, 2002 et 2008 sont sorties).

I.3. DOMAINES D'APPLICATION

L'engineering (contrôle, l'imagerie médicale, codage de la parole, télécommunication, les satellites, la surveillance, automate programmable, etc), le développement des applications embarqués pratiquement dans tous les domaines [3].

I.4 ETAPES DE CONCEPTION

La finalité d'une description en langage VHDL est la réalisation d'un ou plusieurs circuits électroniques réalisant les fonctions souhaitées. On distingue alors deux aspects :

- le VHDL synthétisable pour lequel le compilateur sait générer la circuiterie adéquate.
- le VHDL non synthétisable pour lequel le compilateur ne sait pas traduire en circuits les fonctions demandées.

Un des avantages fondamentaux de VHDL est qu'il peut traduire un cahier des charges complexe et en assurer la simulation : il devient donc un outil de spécification et simulation. C'est du ressort de l'électronicien de voir comment une description non synthétisable a priori par le compilateur peut être malgré tout traduite en circuits. C'est également une des raisons pour lesquelles le langage VHDL est avant tout un outil d'électronicien [4].

Remarque: Les plateformes de développement de FPGA (LATTICE, ALTERA, XILINX, ACTEL, INTEL...) n'acceptent que du langage VHDL synthétisable.

Fig. 1 montre les différentes étapes pour la conception et la réalisation d'un projet [3].

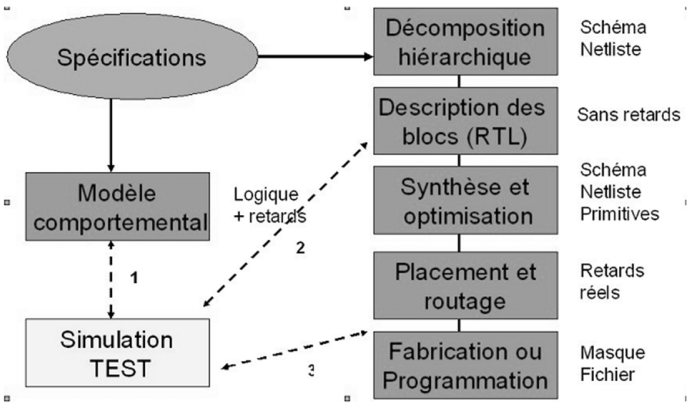


Fig.1. Flot de conception

A partir de la **Fig. 1** on peut clairement observer le rôle fédérateur du VHDL car il intervient au moins sur 3 niveaux, celui de la description comportementale traduisant les spécifications, celui du code RTL (Register Transfert Level) et enfin au niveau technologique post-routage censé représenter le circuit « vrai ». Ces trois types de description seront validés par une même famille de fichiers de test eux-mêmes écrits en VHDL [1].

Le langage est utilisé pleinement dans ses deux déclinaisons : généraliste quand il s'agit de décrire des vecteurs de test ou des comportements abstraits, VHDL synthétisable en vue de générer automatiquement un circuit. Cependant, le VHDL a deux aspects qui peuvent être contradictoires. Lorsqu'il s'agit d'écrire un modèle comportemental qui sera simplement simulé, le langage est compilé puis exécuté par le simulateur. Par contre lorsqu'il s'agit de décrire un circuit qui sera créé par un synthétiseur, la philosophie est sensiblement différente.

L'outil de synthèse, devant transformer l'ensemble du code fourni en une implémentation à base de portes logiques, est conçu pour fonctionner de manière très cadrée. Il est nécessaire de pouvoir lui fournir une description claire (dont la synthèse correspond à l'architecture recherchée) tout en étant le moins spécifique possible (afin de permettre à l'outil d'optimiser au maximum le circuit généré[6]).

Par exemple, si l'on désire générer une fonction de logique combinatoire (indépendante de toute horloge), il faudra affecter l'ensemble des sorties à chaque appel du ***process***, sans quoi l'outil de synthèse, considérant que les sorties non assignées conservent leur ancienne valeur, placera des bascules D en sortie de chaque sortie non affectée. Cette solution est alors très mauvaise, puisqu'elle transforme la fonction en une fonction de logique synchrone, donc dépendant d'une horloge (qui de plus est spécifiée par l'outil de synthèse, hors de contrôle du concepteur).

Cette différence implique un grand travail en amont et en aval du codage, le circuit décrit doit avoir déjà été pensé avant d'être codé et il doit être vérifié après conception, en considérant le nombre de portes et les caractéristiques d'implantation, afin de s'assurer qu'aucune erreur de description n'est présente. Ces contraintes très fortes sur le programmeur entrainent l'adoption de guides de conduites et de méthodes de codage très strictes.

Ces grandes différences avec un langage de programmation comme le C font du VHDL un langage à part, plus proche de l'électronique que de l'informatique. Il n'est d'ailleurs pas rare de voir implémenté sur des FPGA des architectures de microcontrôleurs, eux-mêmes programmés en assembleur ou en C dans la suite du projet[6].

I.5 STRUCTURE D'UNE DESCRIPTION VHDL

Une description VHDL est composée de trois parties [2] :

- **Bibliothèque/ Library**
- **Entité/Entity**
- **Architecteurs/Architecture**

Une conception VHDL décrit un seul système dans un seul fichier. Le fichier a le suffixe * .vhd. Dans le fichier, il y a deux parties qui décrivent le système: l'entité et l'architecture. L'entité décrit le l'interface avec le système (c'est-à-dire les entrées et les

sorties) et l'architecture décrit le comportement. La fonctionnalité de VHDL (par exemple, les opérateurs, les types de signaux, les fonctions, etc.) est définie dans le package. Les différent packages sont regroupés dans une bibliothèque.

IEEE définit l'ensemble de base des fonctionnalités pour VHDL dans la norme paquet (Package). Ce package est contenu dans une bibliothèque appelée IEEE. L'inclusion de la bibliothèque et du package est déclaré au début d'un fichier VHDL avant l'entité et l'architecture [6]. Des fonctionnalités supplémentaires peuvent être ajouté à VHDL en incluant d'autres packages, mais tous les packages sont basés sur la fonctionnalité de base définie dans le package standard. Par conséquent, il n'est pas nécessaire d'indiquer explicitement qu'une conception utilise l'IEEE package standard car il est inhérent à l'utilisation de VHDL. **Fig. 2** montre une représentation générale d'un fichier VHDL.

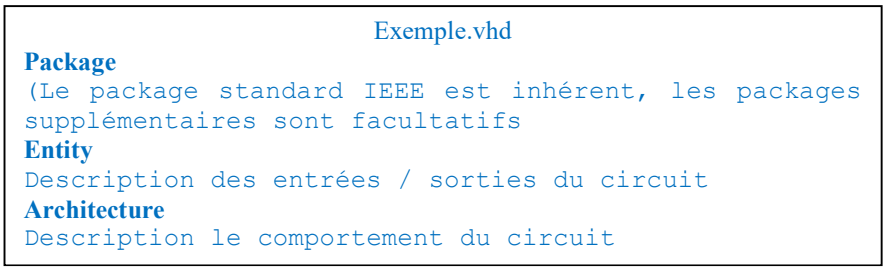


Fig.2. Structure générale d'un fichier VHDL

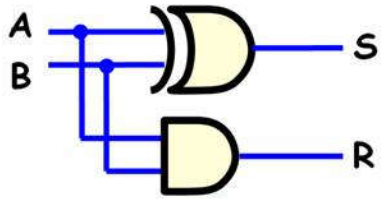
I.5.1 Bibliothèques

Toute description VHDL utilisée pour la synthèse a besoin de bibliothèques. L'IEEE (Institut of Electrical and Electronics Engineers) les a normalisées et plus particulièrement la bibliothèque IEEE1164. Elles contiennent les définitions des types de signaux électroniques, des fonctions et sous programmes permettant de réaliser des opérations arithmétiques et logiques [5].

```
library IEEE;
use IEEE.std_logic_1164.all;          (définition du type bit, vecteur de bit,etc)
use IEEE.numeric_std.all;             (operations numeric)
use IEEE.std_logic_unsigned.all;      (opérations signées ou non signées)
use IEEE.std_logic_signed.all;….
use IEEE.std_logic_textio.all;
use IEEE.std_logic_arith.all;
use IEEE.numeric_bit.all;
use IEEE.math_real.all;
use IEEE.math_complex.all;
```

I.5.2 Entité/Entity

Une entité (**Entity** en Anglais) permet de définir les entrées et sorties de composant(voir **Fig.3**). Exemple d'un demi-additionneur[1] :



(a)

A	B	R	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b)

$$R = AB$$
$$S = \bar{A}B + A\bar{B} = A \oplus B$$

(c)

Fig.3. a)Schéma, b) Table de vérité et c) Equations

```
library ieee;
use ieee.std_logic_1164.all;
entity DA is
        port(A,B : in std_logic; S,R : out std_logic);
end DA;
```

a) **E/S**

E/S or I/O permet de définir le Nom de la description VHDL ainsi que les E/S utilisées, l'instruction qui les définit c'est '**port**' [2] .

Syntaxe

```
entity nom_de_l_entite is
        port (Description des signaux d'entrées /sorties …);
end nom_de_l_entite;
```

Exemple

```
entity exo is
        port ( Clk : in std_logic; Reset : in std_logic; O : out std_logic_vector(1 downto 0) );
end exo;
```

Syntaxe <nom_du_signal> : <sens> <type>;

Exemple

```
Entity mux_Nx1 is
        Generic (d: integer:=4);
        Port (entrée: in bit_vector(d -1 downto 0) ; sel : out integer range 0 to d-1 ) ;
End mux_Nx1;
```

Remarques

-Après la dernière définition de signal de l'instruction **port** il ne faut jamais mettre de ';'
-Chaque signal d'E/S d'une entité est appelé **port**
-Chaque **port** a un nom, une direction (mode) et un type de données

- ✓ Majuscules/minuscules sont équivalents
- ✓ Le premier caractère doit être une lettre
- ✓ Le dernier caractère ne peut être un "underscore"
- ✓ Deux "underscores" successifs sont interdits

-Le nom signal est composé de caractères, le premier caractère doit être une lettre, sa longueur est quelconque, mais elle ne doit pas dépasser une ligne de code. **VHDL** n'est pas sensible à la « casse », c'est à dire qu'il ne fait pas la distinction entre les majuscules et les minuscules

b) Le sens du signal

Le sens d'un signal peut être:

- **in** : pour un signal en entrée.
- **out** : pour un signal en sortie.
- **inout** : pour un signal en entrée sortie
- **buffer** : pour un signal en sortie mais utilisé comme entrée dans la description.

Le **type** utilisé pour les signaux E/S est :

- le **std_logic** pour un signal.
- le **std_logic_vector** pour un bus composé de plusieurs signaux.

Dans le paquetage STD_LOGIC_1164 de la bibliothèque IEEE, le type STD_ULOGIC est défini par :

type std_ulogic is ('U', 'X, 'O', '1', 'Z', 'W', 'L', 'H', '-') .

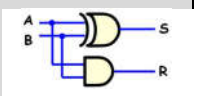
Une donnée de type std_logic possède une valeur parmi neuf possibles :

- Au démarrage les signaux sont dans un état inconnu 'U'.
- 'X' indique un conflit, le signal est affecté d'un côté à '1' et d'un autre à '0'.
- '0' et '1' correspondant aux valeurs booléennes du signal.
- 'Z' correspond à l'état haut 'impédance".
- 'W' est la valeur d'un signal relié à 2 résistances de tirage, une tire à 0 et l'autre à 1.
- 'H' et 'L' sont des valeurs d'un signal relié respectivement à une résistance de tirage à 1 et à 0.
- '-' est un état indifférent. Utile pour décrire les tables de vérité.

I.5.3 Architecture

Une architecture (**architecture en Anglais**) [3] contient les instructions **VHDL** permettant de réaliser le fonctionnement considéré. Elle décrit le fonctionnement d'un circuit ou une partie du circuit électronique. En effet, le fonctionnement d'un circuit est généralement décrit par plusieurs modules **VHDL**. Donc, il faut bien comprendre par module le couple '**Entity/Architecture'**. L'architecture établit à travers les instructions, les relations entre les entrées et les sorties. On peut avoir un fonctionnement purement **combinatoire**, **séquentiel**, les deux **séquentiel et combinatoire**.

```
Architecture arch_DA_flot of DA is
begin
        S <= A xor B;
        C <= A and B;
end arch_DA_flot;
```



Remarques

Une architecture décrit le contenu d'une entité.

• Il est possible de créer plusieurs architectures pour une même entité. Chacune de ces architectures décrira l'entité de façon différente.

• VHDL a trois styles d'architecture qui peuvent être combinée dans le corps d'une architecture :

– Comportementale
– Flot de données
– Structurel

• Le même circuit peut être décrit en utilisant n'importe lequel des trois styles.

a) Description comportementale

Une description comportementale fournit un algorithme qui modélise le fonctionnement du circuit électronique. Une déclaration de processus contient un algorithme. Elle commence par une étiquette (optionnelle), le mot clé "**process**" et une liste des signaux actifs (**sensivity list**). La liste des signaux actifs indique quels signaux provoqueront l'exécution du processus (voir l'exemple ci-après)[4].

Exemple

```
architecture behavioral of eqcomp4 is
begin
        comp: process (a, b)
        begin
        if a = b then  equals <= '1';
        else equals <= '0';
        end if;
        end process comp;
end behavioral;
```

b) Description flot de données)

Une description flot de données (**Data flow**) spécifie comment la donnée est transférée d'un signal à un signal sans utiliser d'affectations séquentielles (comme dans la description comportementale) [4]. Cette description ne nécessite pas de déclaration de processus (**Process**). Toutes les déclarations s'exécutent en même temps (**concurrent signal assignment**), voici un exemple simple en utilisant l'instruction *when…else*.

Exemple

```
architecture dataflow of eqcomp4 is
begin
        equals <= '1' when (a = b) else '0';
end dataflow;
```

c) Descriptions structurelles

Les composants sont instanciés et connectés. Ils doivent être définis dans un package et compilés dans une bibliothèque. Ces dernières sont attachées par une déclaration (voir l'exemple)[4] .

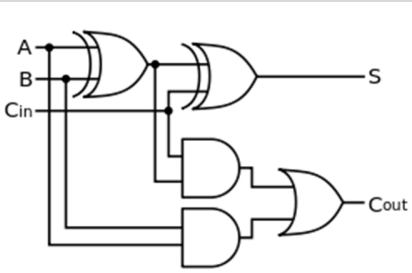
Exemple

```
architecture  archit  of  ma_fct  is
        signal j,k,l : std_logic ;
        component: or3 port ( a1,b1,c1: in std_logic; s1 : out  std_logic); end component;
        component : and3 port(a2,b2,c2: in std_logic; s2: out std_logic); end component;
begin
        Instan1 : or3     port map (a,b,c,j);
        Instan2 : or3             port map (d,e,f,k);
        Instan3:  or3             port map (g,h,i,l);
        Instan4: and3     port map (j,k,l,s);
end archit ;
```

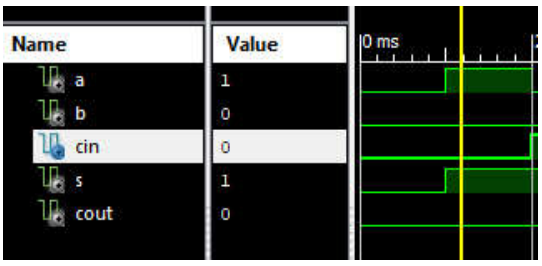
Quelques exemples explicatifs :

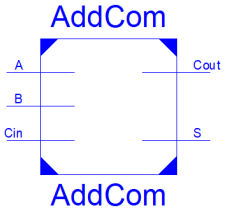
Exemple 1 : Additionneur complet architecture flot de données (data flow architecture)

```
library IEEE;
use IEEE.std_logic_1164.all;
Entity AC is
        Port (A, B, Cin: in std_logic; S , Cout : out std_logic );
End AC;
Architecture V1 of AC is
Begin
        S <= A xor B xor Cin;
        Cout <= (A and B) or (B and Cin) or (A and Cin);
End V1;
```



Résultats de simulation :





Exemple 2 : Additionneur complet: architecture comportementale (behavioral architecture)

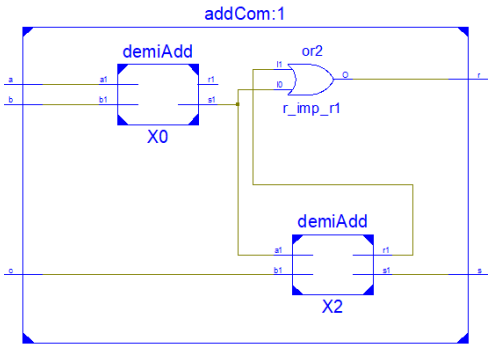
```
 library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity fab is
    Port ( a ,b,c: in std_logic;  s, r : out std_logic);
end fab;
architecture Behavioral of fab is
begin
 process(a,b,c)
 begin
 if(a='0' and b='0' and c='0') then  s<='0';r<='0';
 elsif( a='0' and b='0' and c='1') then  s<='1'; r<='0';
 elsif( a='0' and b='1' and c='0') then s<='1';r<='0';
 elsif( a='0' and b='1' and c='1') then s<='0'; r<='1';
 elsif( a='1' and b='0' and c='0') then  s<='1'; r<='0';
```

a	b	c	s	r
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

```
elsif( a='1' and b='0' and c='1') then s<='0'; r<='1';
elsif( a='1' and b='1' and c='0') then  s<='0';  r<='1';
else  s<='1';  r<='1';
end if;
end process;
end Behavioral;
```

Exemple 3: Additionneur complet architecture structurel (structural architecture)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity addCom is
Port (a, b, c: in STD_LOGIC; r,s: out STD_LOGIC);
end addCom;
architecture Stru of addCom is
signal c1, c2,c3:  std_logic;
component demiAdd is
port (a1, b1 : in std_logic; r1,s1: out std_logic);
end component;
Begin
X0: demiAdd port map (a1=>a,b1=>b, r1=>c1,s1=>c2);
X2: demiAdd port map (a1=>c2,b1=>c, r1=>c3,s1=>s);
r<=c2 or c3;
end Stru ;
```



Composant:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity demiAdd is
    Port ( a1 : in  STD_LOGIC;
           b1 : in  STD_LOGIC;
           r1 : out  STD_LOGIC;
           s1 : out  STD_LOGIC);
end demiAdd;

architecture Behavioral of demiAdd is
begin
s1 <= a1 xor b1;
r1 <= a1 and b1;
end Behavioral;
```

I.5.4 Composants

Un composant (**Component**) permet d'écrire un programme sous la forme structurelle. En effet, il faut déclarer et définir les composants de la fonction puis dans le corps de l'architecture, il suffit de les connecter en suivant un schéma structurel. Cette méthode d'écriture permet aussi la description hiérarchique. Quatre blocs sont à distinguer dans une telle définition[2] :

- recensement de tous les composants du schéma. Ceci se fait par la syntaxe component ... end component ;
- affectation des liaisons du schéma aux broches des composants. On utilise **port map**. Cette phase correspond à réaliser la NETLIST du schéma ;

- attribution des circuits à leur ENTITY. Cette phase permet de contrôler le modèle comportemental qui va déterminer du fonctionnement du composant. Cette phase est réalisée avec **configuration** ;
- définition comportementale de chaque composant associé. Ici on procédera à une simple structure d'écriture basée autour d'**entité** et **d'architecture.**

Syntaxe : Déclaration
Component <nom de component>
    Port(Description des signaux d'entrées /sorties …)
End <nom de component> ;
Syntaxe : Instanciation
    <Nom de l'instance> : <Nom du component>
Port map (Signal1 => SignalInterne1, Signal2 => SignalInterne2);

Remarque:
Le circuit contient quatre déclarations d'instanciation de composants : Elles commencent par une étiquette suivie de '**:**', puis un nom de composant "**port map**" décrit comment chaque composant est connecté au reste du système. Ce sont des déclarations concurrentes : l'ordre n'a pas d'importance. Elles sont toutes exécutées en même temps.

Voici un exemple :

Exemple
```
Entity MAJ is
        port( A_IN, B_IN, C_IN: in std_logic; Z_OUT : out std_logic);
end MAJ;
Architecture struct of MAJ is
        component AND2_OP port (A, B : in std_logic; z : out std_logic); end component;
        component OR3_OP port (A, B, C : in std_logic; z : out std_logic);end component;
        signal INT1, INT2, INT3 : std_logic;
Begin
        A1: AND2_OP port map (A_IN,B_IN,INT1);
        A2: AND2_OP port map (A_IN,C_IN,INT2);
        A3: AND2_OP port map (B_IN,C_IN,INT3);
        A4: OR3_OP   port map (INT1,INT2,INT3,Z_OUT);
End struct;
```

a) Port Map

PORT MAP est utilisé pour associer les broches d'un **component** avec les signaux du circuit. Deux méthodes d'affectation sont possibles[5]:

- Association des noms
- Association des positions (la plus courante).

Syntaxe
PORT MAP (pin_name => signal_name,,,,);
PORT MAP (signal_name)

b) Instanciation

Une instanciation est une instruction concurrente qui est utilisée pour définir la hiérarchie de conception en effectuant une copie d'une entité de conception de niveau

inférieur dans une architecture. L'instruction d'instanciation introduit un sous-système déclaré ailleurs. L'instanciation contient une référence à l'unité instanciée et des valeurs réelles pour les génériques et les ports[5]. Il y a trois formes d'instanciation :

- instanciation d'un composant ;
- instanciation d'une entité de conception ;
- instanciation d'une configuration ;

L'instanciation d'un composant introduit une relation avec une unité définie précédemment en tant que composant (voir **composant**). Le nom du composant instancié doit correspondre au nom du composant déclaré. Le composant instancié est appelé avec les paramètres réels pour les génériques et les ports. La liste d'association peut être positionnelle ou nommée. Il n'est pas nécessaire de définir un composant pour l'instancier : la paire entité / architecture peut être instanciée directement. Dans une telle instanciation directe, l'instruction d'instanciation contient le nom de l'entité de conception et éventuellement le nom de l'architecture à utiliser pour cette entité de conception.

Syntaxe

instance_label: [**component**] component_name
  [**generic map**] [**port map**];
instance_label: **entity** entity_name [(architecture_name)]
  [**generic map**] [**port map**];
instance_label: **configuration** configuration_name
  [**generic map**] [**port map**];

Exemple

```
u1: Nand4 port map(A, B, Q);
u2: entity work.Parity
  generic map(N => 8)
  port map(A => Data,Odd => ParityByte);
```

Remarques :

Une entité, une architecture ou une configuration doit être compilée dans une bibliothèque avant que l'instance correspondante puisse être compilée. Cependant, une instance d'un composant peut être compilée avant même que l'entité de conception correspondante ait été écrite.

Attention !

3 façon de déclarer un composant :

- ~~Toutes les paires entity/architecture sont déclarées dans le même fichier~~
- La paire entity/architecture de chaque composant est déclarée dans un fichier qui lui est propre
- La structure du composant est déclarée dans une bibliothèque via un package

I.5.4 Configurations

Configuration s'utilise en combinaison avec **component**. Une déclaration de configuration définit comment la hiérarchie de conception est liée lors de l'élaboration, en répertoriant les entités et les architectures utilisées à la place de chaque instanciation de composant dans une architecture[1].

Une configuration simple contient une référence à un seul corps d'architecture. Les configurations hiérarchiques permettent d'imbriquer des configurations. Ce mécanisme permet de lier les instanciations de composants avec les entités de conception dans la hiérarchie. Lorsque les ports et les génériques dans la déclaration de composant ne sont pas égaux à leurs équivalents dans la déclaration d'entité, vous pouvez utiliser une notification explicite sur la manière dont les ports et génériques de l'entité doivent être liés aux ports et génériques de l'instance de composant. La carte générique et la carte du port sont utilisées à cette fin.

Syntaxe

```
configuration configuration_name of entity_name is
  [ configuration_declarations ]
  for architecture_name
    [ configuration_item ]
  end for;
end [ configuration ] [ configuration_name ];

configuration_item = for instance_label : component_name
                       [ use_item
                         [ generic_map ]
                         [ port_map ]; ]
                       [ configuration_item ]
                     end for;
use_item = entity [ library_name. ] entity_name [ ( architecture_name ) ] |
           configuration [ library_name. ] configuration_name
```

Exemple

```
architecture Structure of MicroProcessor is
  component ALU port (...); end component;
  component MUX port (...); end component;
  component LATCH port (...); end component;
begin
  A1: ALU port map(...);
  M1: MUX port map(...);
  M2: MUX port map(...);
  M3: MUX port map(...);
```

```
  L1: LTACH port map(...);
  L2: LATCH port map(...);
end Structure;
-- a configuration of the microprocessor
library TTL, work;
use TTL.all;
use work.all;
configuration Cfg4 of MicroProcessor is
  for Structure
    for A1: ALU
      use configuration TTL.SN74LS181;
    end for;
    for M1, M2, M3: MUX
      use entity Multiplex4(behavior);
    end for;
    for all: LATCH
      use entity Work.Latch;
    end for;
  end for;
end configuration Cfg4;
```

Remarque:
Bien que les configurations soient pertinentes et utiles pour sélectionner quelles entités et architectures constituent la hiérarchie de conception, de nombreux outils de synthèse ne les prennent pas en charge. Une spécification de configuration est une construction qui définit quelle entité et quelle architecture sont utilisées à la place des instances d'un seul composant pendant l'élaboration.

Synthax

```
for instance_label : component_name
  use use_item
    [ generic_map ]
    [ port_map ];
use_item = entity [ library_name. ] entity_name [ ( architecture_name ) ] |
        configuration [ library_name. ] configuration_name
```

Chaque instanciation de composant fait référence à une entité de conception (paire entité / architecture) et l'association est spécifiée par une spécification de configuration. La spécification des composants apparaît dans la partie déclarative de l'unité, où les instances sont utilisées. De cette manière, les composants peuvent être configurés dans l'architecture qui les utilise sans utiliser de déclaration de configuration séparée [6].
Les spécifications de configuration sont inflexibles car la modification de la configuration nécessite l'édition de l'architecture contenant la configuration. Il est généralement préférable d'utiliser des déclarations de configuration distinctes.

Lorsque les ports et les génériques dans la déclaration de composant ne sont pas égaux à leurs équivalents dans la déclaration d'entité, vous pouvez utiliser une notification explicite sur la manière dont les ports et génériques de l'entité doivent être liés aux ports et génériques de l'instance de composant. La carte générique et la carte des ports sont utilisées à cette fin.

Exemlpe

```
architecture Structure of MicroProcessor is
component ALU port (...); end component;
component MUX port (...); end component;
component LATCH port (...); end component;

for A1: ALU use configuration TTL.SN74LS181
port map (Clk, Rst, A1, A2, Q);
for M1, M2, M3: MUX use entity Multiplex4(behavior);
for all: LATCH use entity Work.Latch;

begin
A1: ALU port map(...);
M1: MUX port map(...);
M2: MUX port map(...);
M3: MUX port map(...);
L1: LTACH port map(...);
L2: LATCH port map(...);
end Structure;
```

Remarques :
Bien que les configurations soient pertinentes et utiles pour sélectionner quelles entités et architectures constituent la hiérarchie de conception, de nombreux outils de synthèse ne les prennent pas en charge.

I.6 OBJETS ET DONNEES

I.6.1 Types d'objets

Signal : Fil conducteur.
Syntaxe : **Signal** <nom de signal> : <type> ;
L'affectation se fait avec l'opérateur «<=». Accès à des sous-éléments avec l'opérateur alias.

Variable : Valeur intermédiaire d'une opération complexe.
Syntaxe : **Variable** <nom de variable> : <type> ;

Constant : Valeur fixe.
Syntaxe : Constant <nom de constant> :<type> :=<valeur initial> ;

L'affectation se fait avec l'opérateur (:=)

Exemples

a) Constant

```
Constant state_1 : std_logic_vector:= "01";
Constant state_2 : std_logic_vector:= "10";
Constant addr_max: integer:= 1024;
```

b)Signal

Declaration d'un signal

```
Signal      port_i : std_logic;
Signal      bus_signal: std_logic_vector(15 downto 0);
Signal      count: integer range 0 to 31;
```

Affectation d'un signal

```
std_logic_signal_1 <= notstd_logic_signal_2;
std_logic_signal <= signal_a andsignal_b;
large_vector(15 downto 5) <= small_vector(10 downto 0);
```

c) Variable

Declaration

```
Variable count_v: integer range0 to 15;
Variable data_v: std_logic_vector(7 downto 0);
Variable condition_v: boolean;
```

Affectation d'une variable

```
boolean_v := true;
```

Remarques

Un signal représente une équipotentielle, Il doit être déclaré avant utilisation et peut être déclaré :

- dans un package, il est alors global
- dans une entité, il est alors commun à toutes les architectures de l'entité
- dans l'architecture, il est alors local, utilisé pour des signaux intermédiaires.

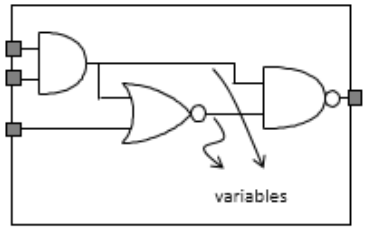
Une constante doit être déclarée avant utilisation. Elle peut être déclarée :

- dans un package, elle est alors globale
- dans une entité, elle est alors commune à toutes les architectures de l'entité
- dans l'architecture, elle est alors locale

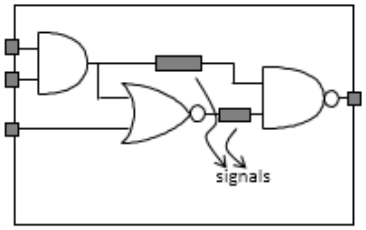
Une variable doit être déclarée avant utilisation, elle ne peut être déclarée que dans un «**Process**»

Attention !

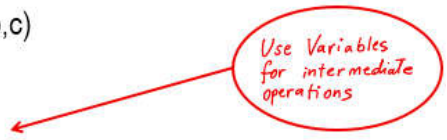
```
-- Process 1 - Correct Coding Style
proc1: process (x, y, z) is
variable var_s1, var_s2: std_logic;
begin
var_s1 := x and y;
var_s2 := var_s1 xor z;
res1 <= var_s1 nand var_s2;
end process;
```




```
Process 2 - Incorrect
proc2: process (x, y, z) is
begin
sig_s1 <= x and y;
sig_s2 <= sig_s1 xor z;
res2   <= sig_s1 nand sig_s2;
end process;
```

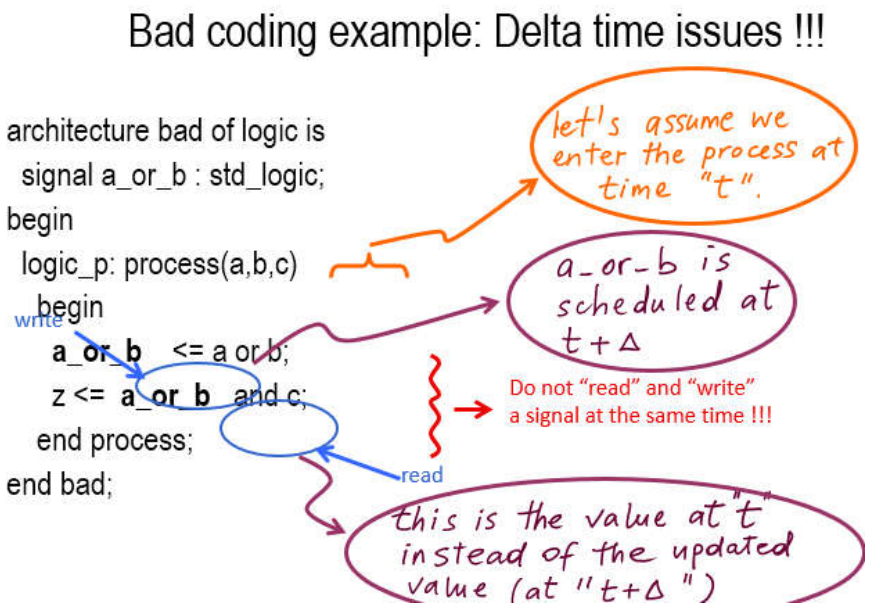



```
architecture good of logic is
  variable a_or_b : std_logic;
begin
  logic_p: process(a,b,c)
    begin
      a_or_b := a or b;
      z <= a_or_b and c;
    end process;
end good;
```



- On ne peut pas lire et écrire en même temps dans un process
- On utilise les variables pour les calculs intermédiaires

Attention !



Remarques

-Il n'y a pas de différence entre les espaces, tabulations et retours de chariot, présents seuls ou en groupe.
-Le langage n'est pas sensible à la casse.
-Un littéral composé d'un caractère unique est placé entre apostrophes : 'a', '5'.
-Un littéral composé d'une chaîne de caractères est placé entre guillemets : "bonjour", "123ABC".
-Un littéral numérique peut être spécifié avec sa base, avec le format suivant : base#chiffres[.chiffres]#[exposant] où les crochets indiquent des éléments facultatifs. La base doit être entre 2 et 16, inclusivement. Par exemple, les nombres suivants ont tous la même valeur : 10#33# = 10#3.3#E1 = 2#10001# = 16#21# = 7#45#. Les chiffres doivent être inférieurs à la base, et peuvent prendre les valeurs 0 à F inclusivement.
-Un littéral composé de bits peut être exprimé en bases 2, 8 ou 16, avec les spécificateurs de base B, O etX, respectivement : B"11111111", O"377", X"FF".
-Tout texte placé après deux tirets et jusqu'à la fin d'une ligne est un commentaire, ce qui est semblable au'//' de C. Il n'y a pas de commentaires en bloc (/* … */) en VHDL.

I.6.2 Types de données

VHDL est un langage fortement typé : tout objet doit être déclaré avant utilisation. Les types prédéfinis sont :

Scalaire :

- integer
- real
- enumerated
- physical

Enregistrement:

- array
- record

Pointeur :

- acces

E/S:

- file

Exemple

```
type COULEURS is (ROUGE, JAUNE, BLEU, VERT, ORANGE);
type bus is array (0 to 31) of std_logic;
type TAB2 is array(0 to 3, 1 to 8) of std_logic;
Type EtatsLogiques is record
valeur : integer range -127 to 128;
force :integer;
end record ;
BINAIRE, exemple : BUS <= "1001" ; -- BUS = 9 en décimal
HEXA, exemple : BUS <= X"9" ;          -- BUS = 9 en décimal
OCTAL, exemple : BUS <= O"11" ;      -- BUS = 9 en décimal
```

I.7 INSTRUCTIONS DU MODE CONCURRENTS

Toutes les déclarations sont exécutées simultanément et en permanence. L'ordre des déclarations dans le code source n'a pas d'influence. Les déclarations possibles sont :

- ✓ Assignation continue : **<=**
- ✓ Instantiation d'un composant : **port map**
- ✓ Assignation conditionnelle : **when … else**
- ✓ Assignation sélective : **with … select … when … when**
- ✓ Appel d'un **process**
- ✓ Instruction **generate**
- ✓ Appel d'une **function**

I.7.1 Equations booléens

La base de la plupart des codes VHDL est les interactions logiques entre les signaux

Syntaxe

SIGNAL <= expression

Exemple :

```
X <= notsignal_1;
Y <= signal_1 andsignal_2;
Z <= (notsignal_1) and (signal_2 xor (signal_3 or notsignal_4))
```

I.7.2 Affectation conditionnelle

Cette instruction modifie l'état d'un signal suivant le résultat d'une condition logique entre un ou des signaux, valeurs, constantes [6].

Syntaxe

SIGNAL <= expression **when** condition
[**else** expression **when** condition]
[**else** expression];

Exemple

```
X <= '1' when b = c else '0';
Y <= signal_1 when state = idle else
signal_2 when state = state_1 else
signal_3 when state = state_2 else
signal_4 when others;
```

I.7.3 Affectation sélective

Cette instruction permet d'affecter différentes valeurs à un signal, selon les valeurs prises par un signal dit de sélection.

Syntaxe

with SIGNAL_DE_SELECTION **select**
SIGNAL <= expression when valeur_de_selection,
[expression when valeur_de_selection,]
[expression when others];

Example:

```
With state select
        X <= "00" when st0;
        "11" when st1 | st2;
sig_vector when others ;
```

Remarques :

l'instruction **[else expression]** n'est pas obligatoire mais elle fortement conseillée, elle permet de définir la valeur du **SIGNAL** dans le cas où la condition n'est pas remplie.
l'instruction **[expression when others]** n'est pas obligatoire mais fortement conseillée, elle permet de définir la valeur du **SIGNAL** dans le cas où la condition n'est pas remplie. **when others** est nécessaire car il faut toujours définir les autres cas du signal de sélection pour prendre en compte toutes les valeurs possibles de celui-ci.

I.8 INSTRUCTIONS DU MODE SEQUENTIEL

Ce mode concerne uniquement les **functions, procedures et processes**

- Les **process** manipulent les variables et les signaux. Au sein de ces descriptions, les déclarations sont exécutées de manière séquentielle, l'une après l'autre. L'ordre des déclarations est donc important. Les déclarations possibles sont :
  - ✓ Assignation continue : **<=(signal)** et **:=(variable)**
  - ✓ Assignation conditionnelle : **if … then … elsif … then … else … end if**;
  - ✓ Assignation sélective : **case…is …when…=>…when…=>…end case**;
  - ✓ Boucles : **for … in … loop … end loop**;
  - ✓ Boucles : **while ... loop … end loop**;

I.8.1 Process

Les processus sont essentiels dans les descriptions comportementales d'une entité de conception. Ils facilitent la spécification des fronts d'horloge ainsi que la synchronisation entre les assignations de signaux. Les processus sont le plus souvent utilisés lorsqu'une assignation de signal dépend de changements dans une autre. La dépendance à cet égard devrait être reflétée dans la liste de sensibilité du processus. L'exécution d'un **process** est déclenchée par un ou des changements d'états de signaux logiques. Le nom de ces signaux est défini dans **la liste de sensibilité** lors de la déclaration du **process[1]**.

Syntaxe

```
[<process_name>:] process(<sensitive list>)
{ <type-declaration>
|<constant-declaration>
|<variable-declaration>
| … }
begin
{<statement(s)> }
end process
```

Example:

```
output_process: process(flag_signal)
begin
if flag_signal = '1' then output_vector <= "010";
```

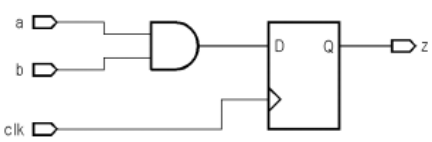
```
else output_vector <= "101";
end if;
end process;
```

Remarques :
Le nom du **process** entre crochet est facultatif, mais il peut être très utile pour repérer un **process** parmi d'autres lors de phases de mise au point ou de simulations.

- L'exécution d'un **process** a lieu à chaque changement d'état d'un signal de la liste de sensibilité.
- Les instructions du **process** s'exécutent séquentiellement.
- Les changements d'état des signaux par les instructions du **process** sont pris en compte à la **fin** du **process**.
- Hors d'un **process**, les affectations sont concurrentes et immédiates. Ainsi, deux **process** peuvent être exécutés en même temps.

Example

```
proc6: process
begin
wait until clk = '1';
z <= a and b;
end process proc6;
```



Attention !

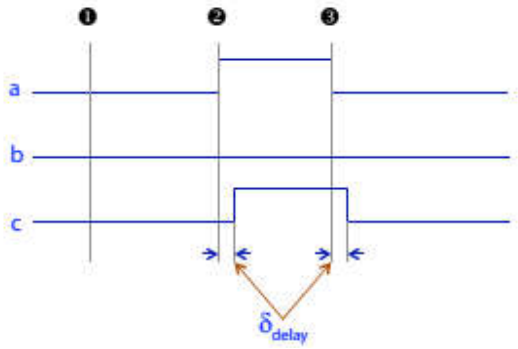
```
signal  S : std_logic;
begin
process(A)
  begin
    S <= A;
    B <= S;
end process;
```

≠

```
signal  S : std_logic;
begin
process(A,S)
  begin
    S <= A;
    B <= S;
end process;
```

Exemple

```
Entity ex is
port (a: instd_logic;  b: outstd_logic);
end ex;
architecture basic of ex is
signalc: std_logic;
begin
process(a)
begin
c <= a;
ifc='1'
then b <= a;
elseb <= '0';
end if;
end process;
end basic;
```



- Le signal ***c*** est affecté à ***a*** au début du processus, mais sa valeur est réellement mise à jour seulement à la fin du processus. Pour cette raison, lors du ***if…then*** la valeur de

c qui est testée est celle qui avait le signal à la fin de l'exécution précédente du processus

- Supposons le cas de la figure suivante, avec *a* et *b* égaux à '0'au début de la simulation (temps 1)
- Au temps 2, *a* a changé. Comme le processus est sensible à a, le simulateur redémarre l'exécution du processus. Le signal c'est affecté à *a*, et la condition (*c*='1') est testée. Le résultat du test sera false, puisque c'est toujours égal à '0': la mise à jour à la valeur '1'n'aura lieu qu'au temps *2+ delta* , c'est-à-dire à la fin du processus. A la fin de la première exécution du processus, *b* sera donc toujours égal à '0'
- Au temps 3, ***a*** passe à '0', ce qui fait redémarrer l'exécution du processus. Comme la valeur de ***c*** est mise à jour seulement au temps 3+ delta, *c* garde sa valeur précédente ('1'). La condition (c='1') est maintenant vraie, et best affecté à la valeur de ***a*** à la fin du processus, c'est-à-dire à '0'. En conclusion, ***b*** reste toujours à '0' pendant la simulation

a) L'instruction wait

Syntaxe
wait on sensitivity_list;
wait until conditional_expresion;
wait for time_expression;

Exemple wait

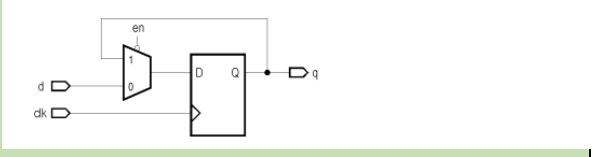
```
proc3: process
begin
x <= a and b and ;
wait on a, b, c;
end process proc3
```

Même fonctionnement.
Example process

```
proc1: process (a, b, c)
begin
x <= a and b and c;
end process proc1
```

wait until signal = value;
wait until signal'event and signal = value;
wait until not signal'stable and signal = value;

```
process
begin
wait until clk = '1';
ifen = '1' then
q <= d;
end if;
end process;
```



I.8.2 Assignation conditionnelle

a) if-then-else conditional statement

Syntaxe

if<condition> then<statement(s)>
{ elsif<condition> then<statement(s)> }
[else<statement(s)>]
end if;

Exemple

```
If cond_v_1 = '1' then
        out_vector <= "001";
elsif cond_v_2 = '1' then
        out_vector <= "110";
else
        out_vector <= "000";
end if;
if (RESET='1') then SORTIE <= "0000";
end if ;
```

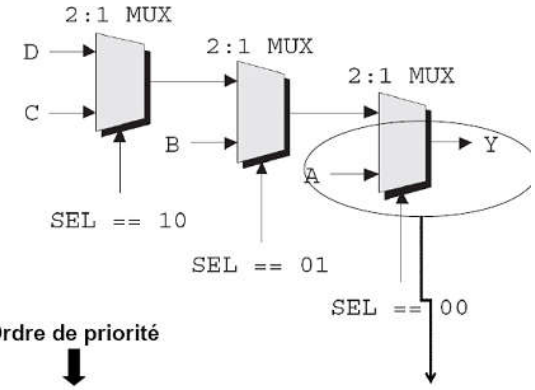
Attention !

```
begin
process(EN,D)
  begin
    if (EN = '1') then
      Q <= D;
    end if;
end process;
```

≠

```
begin
process(EN,D)
  begin
    if (EN = '1') then
      Q <= D;
    else Q <= 0;
    end if;
end process;
```

```
if sel="00" then
   Y<= A;
elsif sel="01" then
   Y<=B;
elsif sel="10" then
   Y<=C;
else
   Y<=D;
end if;
```



Ordre de priorité

Rapidité mais nécessite plus de matériel (ici 2 slices)

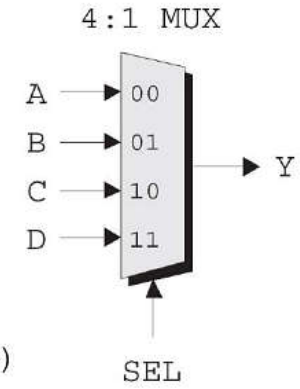
Réduction du chemin critique

b) Boucle case-when

Syntaxe
case<expression> is
{ when<choice(s)> => <assignments>; }
when<choice(s)> => <assignments>;
end case;

Exemple
Case state **is**
when“00" => integer_signal <= 1;
when"11" => integer_signal <= 2;
when others=> integer_signal <= 0;

```
case sel is
   when «  00 » => Y<= A;
   when « 01 » => Y<=B;
   when « 10 » => Y<=C;
   when others =>Y<=D;
end case;
```



Sans priorité ➡ compact (1 slice)

c) Boucle for

Syntax:
[<loop_lavel>:]
for<identifier> in<discrete_range> loop
<statement(s)>;
end loop [<loop_lavel>]

Exemple
for_loop_1:
```
        for i in 3 downto 0 loop
            if reset(i) = '1' then
                out_vector(i) <= ‘0’;
            end if;
end loop for_loop_1;
```

exemple

d) Boucle while

[<loop_lavel>:]
While <condition> loop
<statement(s)>;
end loop [<loop_lavel>];

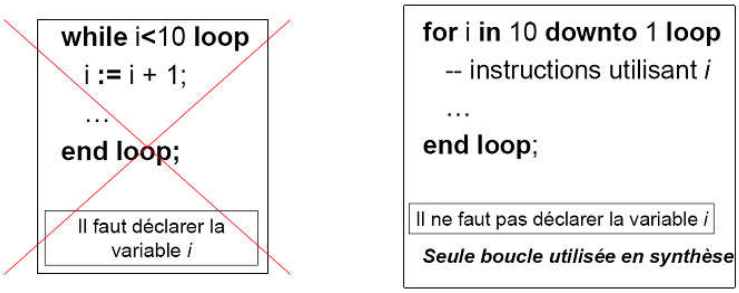
Example

```
while_loop_1:
while(count > 0) loop
 count := count - 1;
result <= result + data_in;
end loop while_loop_1;
```

Attention !

For i **in** val_deb **to** val_fin **loop** … **end loop;**

While condition **loop**… **end loop;**



I.9 OPERATEURS

Addition et soustraction définies pour tous les types numériques multiplication et division s'appliquent à 2 réels ou entiers ou 1 objet physique et un réel entier, MOD et REM définis pour le type entier, ** élève entier ou réel à une puissance entière.

+ Addition
- Subtraction
*** Multiplication**
/ Divide
Mod Modulus
**** Power Operator (i.e. 2**8 returns 256)**

Les opérateurs suivants concatènent plusieurs bits dans un bus ou répliquent un bit ou une combinaison de bits plusieurs fois.

a & b & c Concatenate a, b and c into a bus

Les opérateurs logiques suivants sont utilisés dans les instructions TRUE / FALSE conditionnelles telles qu'une instruction 'if' afin de spécifier la condition de l'opération.
Utilisés pour des objets de type bit, booléen ou tableaux unidimenssionnels de ces types utilisation des opérateurs optimisée par le simulateur

NOT Not True AND (NAND) Both Inputs True
OR (NOR) Either Input True
XOR (XNOR) Only one Input True

Les opérateurs égalité et différent : définis sur tous les types (sauf file)
-inégalités s'appliquent à tous les types scalaires et les vecteurs d'élément entiers ou énumérés
-relation d'ordre sur les types énumérés : gauche->droite

= Inputs Equal
/= Inputs Not Equal
< Less-than
<= Less-than or Equal
> Greater-than
>= Greater-than or Equal

Les opérateurs suivants décalent un bus à droite ou à gauche d'un nombre de bits.
La bibliothèque **IEEE.numeric_std** doit être accessible pour ces fonctions

sll Left shift (i.e. a sll 2 shifts a two bits to the left)
sla Left shift and maintain sign bit
srl Right shift (i.e. b srl 1 shifts b one bits to the right)
sra Right shift and maintain sign bit
rol Rotate left
ror rotate right

Remarques :
Les opérateurs multiplicatifs et l'opérateur d'exponentiation (**) sont soumis à des restrictions, notamment en synthèse où seules les opérations qui se résument à des décalages sont généralement acceptées.
Certaines librairies (numeric_bit et numeric_std de la librairie IEEE) surdéfinissent (au sens des langages objets) les opérateurs d'addition et de soustraction pour les étendre au type bit_vector, par exemple.
On notera que tous les opérateurs logiques ont la même priorité, il est donc plus que conseillé de mettre en parenthèse toutes les expressions qui contiennent des opérateurs différents de cette class.

I.10. SOUS-PROGRAMMES (FONCTIONS ET PROCEDURES)

Le langage VHDL permet l'utilisation et la création de fonctions ou de procédures que l'on peut appeler à partir d'une architecture. Le rôle de ces fonctions ou procédures est de permettre, au sein d'une description, la création d'outils dédiés à certaines tâches pour un type déterminé de signaux (voir chapitre suivant concernant les types de signaux). Cette notion d'objets est typique aux langages évolués de programmation. Permettent de modulariser le code :

- Permettent de regrouper des actions répétitives
- Permettent de réutiliser des fonctions
- Rendent les fichiers plus lisibles
- Permettent d'automatiser des actions (simulation)
- Fonction et procédure

I.10.1 fonctions

Les fonctions (**function**) sont des blocs d'instructions qui retournent une valeur. Les fonctions peuvent être appelées de différents endroits. Les paramètres passés sont du type IN et les objets de class **SIGNAL** ou **CONSTANT**. On peut déclarer des variables locales à l'intérieur d'une fonction. Ces variables perdent leur valeur à chaque sortie de la fonction et sont initialisées à chaque appel.

Syntaxes

```
function function_name{(parameter_list)} RETURN type_name IS;
Corps de la fonction
BEGIN
[block_statement]
[generate_statement]
END [function_name];
```

Exemple

```
Entity exo is
        Port (A1,A2,A3,B1,B2,B3 : in std_logic ; S1,S2 :out std_logic);
End exo;
Architecture arch of exo is
Function exo (A,B,C:std_logic) return std_logic is
Variable r: std_logic;
        Begin
            r:=not(A and B and C);
        Return r;
End;
Begin
        S1<=exo(A1,A2,A3);
        S2<=exo(B1,B2,B3);
End Arch;
```

Remarques

Une fonction peut être déclarée dans un **PACKAGE** ou dans le corps d'une **ARCHITECTURE.**
Une fonction autorise qu'un traitement séquentiel (pas de PROCESS et WAIT) ni d'assignation d'un signal. Une fonction doit comporter au moins une instruction RETURN. L'appel d'une fonction il peut être fait à partir d'instructions **séquentielles** ou **concurrentes**. Dans le cas des instructions concurrentes, la fonction sera toujours vérifiée.

I.10.2 Procédures / Procedure

Les procédures (**Procedure**), elles diffèrent des fonctions par le fait qu'elles acceptent des paramètres dont la direction peut être IN, INOUT et OUT. Une procédure ne possède donc pas un ensemble de paramètres d'entrées et un paramètre de sortie mais un ensemble de paramètres d'entrées-sorties et aucun paramètre spécifique de sortie.

Syntaxe

```
PROCEDURE procedure_name{(parameter_list)};
BEGIN
[block_statement][generate_statement]
END [procedure_name];
```

Exemple

```
Entity test_de_rs is
        port (E1, E2 : in std_logic; S1, S2 inout std_logic);
End test_de_RS;
Architecture test_de_RS of test_de_RS is
Procedure FFrs (signal A, B : in std_logic; signal Q, Qb : out std_logic)
Begin
        Q <= not (A or Qb);
        Qb <= not (B or Q);
End;
Begin
        FFres (E1, E2, S1, S2);
end test_de_RS;
```

On constate que les paramètres A, B, Q et Qb de la procédure ont été explicitement décrits. Ainsi, la liste des paramètres contient :
– le genre de paramètre, variable (valeur par défaut), signal ou constante,
– le nom des paramètres, dans notre cas, A, B, Q et Qb (ces noms ne sont connus en tant que paramètres qu'à l'intérieur de la procédure),
– la direction de chaque paramètre, IN, OUT ou INOUT (dans notre cas INOUT pour Q et Qb qui sont utilisés en lecture IN et écriture OUT au sein de la procédure).

Remarques

Une procédure peut être déclarée dans un **Package** ou dans le corps d'une **Architecture**. On peut déclarer des variables locales à l'intérieur d'une procédure. Ces variables perdent leur valeur à chaque sortie de la procédure et sont initialisées à chaque appel.
-Bien qu'une **Procedure** ne retourne pas de valeur contrairement à une **Fonction**, on peut déclarer dans son entête plusieurs paramètres de type **out** ou **inout** et ainsi récupérés des valeurs de sortie.
-Une procédure peut contenir un **WAIT** sauf si elle est appelée par un **PROCESS** avec une liste de sensibilité ou depuis une **FONCTION**.

I.10.3 Package

PACKAGE déclare une fonction ou une procédure. Le nom défini par PACKAGE BODY doit être le même que celui déclaré par PACKAGE.

Syntaxe

```
PACKAGE package_name IS
[type_declaration]
[subtype_declaration]
[constant_declaration]
[component_declaration]
END [package_name];

Corps du PACKAGE
PACKAGE BODY package_name IS
[type_declaration]
[subtype_declaration]
[constant_declaration]
.
[subprogram_declaration]
END [package_name];
```

Exemple

```
PACKAGE math IS                                  -- déclaration de l'entête du package
FUNCTION minval (CONSTANT a,b : IN integer) RETURN integer;
FUNCTION maxval (CONSTANT a,b : IN integer) RETURN integer;
CONSTANT maxint: integer:=16#FFFF#;
END math;

PACKAGE BODY math IS                                -- déinition du corp du package

FUNCTION minval (CONSTANT a, b : In integer) RETURN integer is
BEGIN
IF a < b THEN RETURN a;
ELSE RETURN b;
END IF;
END minval;

FUNCTION maxval (CONSTANT a, b : In integer) RETURN integer is
BEGIN
        IF a > b THEN RETURN a;
ELSE RETURN b;
```

```
END IF;
END maxval;
END Math ;
```

I.10.4 Fonctions et procédures au sein d'un package

Dans les exemples qui précèdent, les fonctions ou procédures ont été insérées dans des architectures. Or, dans ce cas précis, les fonctions ou procédures en question ne sont accessibles que dans l'architecture dans laquelle elles ont été décrites. Pour les rendre accessibles à partir de plusieurs architectures, il est nécessaire de les déclarer au sein de packages et de déclarer en amont de l'architecture vouloir utiliser ce package. En reprenant l'exemple de la création d'une fonction 'nonet' et en insérant cette fonction au sein d'un package :

```
Example
Package pack_nonet is
Function nonet(A,B,C :std_logic) return std_logic ;
End pack_nonet;

Package body pack_nonet is
Function nonet(A,B,C:std_logic) return std_lgic is
Variable r:std_logic;
Begin
r:=not(A and B and C);
Return r;
End;
End pack_nonet;
Library user;

Use user.pack_nonet.all;
Entity exo is
Port (e1,e2,e3……..);
End exo;
Architecture arch of exo is
Begin
S1<=nonet(e1,e2,e3);
        ,….
End
```

Remarques

La fonction **nonet** déclaré dans le package est décrite dans le package body. Cette procédure est obligatoire est valable aussi pour les procédures. Toute fonction ou procédure déclarée au sein d'un package est décrite dans le package body associe à ce package. En utilisant l'instruction use **work.mypackage.all** pour charger les fonctions, procédure, composants, etc charge dans le fichier le package my_package

I.11 ATTRIBUTS

I.11.1 Présentation des attributs, leurs rôles

Les attributs permettent d'ajouter des informations supplémentaires à un signal, variable ou un composant.

Syntaxe

object_name'attribut_name

Le langage VHDL propose un outil appelé attribut que l'on peut, en quelque sorte, assimiler à des fonctions. Placé auprès d'un signal, l'attribut " 'event " va, par exemple, retourner une information vraie ou fausse (donc de type booléen) sur le fait que le signal en question ait subi ou non un événement (un changement de valeur). Le fonctionnement de cet attribut ressemble au comportement d'une fonction dont le paramètre de retour serait de type booléen, à ceci près que l'attribut 'event en question possède une notion de mémoire. En effet, il détecte un changement d'état qui nécessite d'avoir au préalable mémorisé l'état précédent. Il existe en réalité deux sortes d'attributs.

- Les attributs destinés à retourner des informations concernant le comportement d'un signal (événements, dernière valeur...).
- Les attributs destinés à retourner des informations concernant les caractéristiques d'un vecteur (longueur, dimension...).

Pour ces deux sortes d'attributs, on parle d'attributs prédéfinis, au même titre que l'on parle de types prédéfinis (bit, boolean, etc.). Voici une liste non exhaustive des différents attributs prédéfinis proposés par le langage VHDL.

Syntaxe des attributs pour le type ARRAY ou scalaire

X'HIGH élément le plus grand ou borne maximale
X'LOW élément le plus petit ou borne minimale
X'LEFT élément de gauche ou borne de gauche
X'RIGHT élément de droite ou borne de droite

Syntaxe des attributs pour les objets déclarés dans un ARRAY ou un SUBTYPE

x'RANGE élément range de x
x'REVERSE_RANGE élément range de x dans l'ordre inverse
x'LENGTH x'HIGH - x'LOW + 1 (integer)

Syntaxe des attributs pour 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 est change d'état

Exemple

```
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 -- fonctionnement d'une bascule D
CONSTANT clrit: std_logic := '0';
CONSTANT setit: std_logic := '1';
BEGIN
PROCESS (clk, clr, set) -- liste de sensibilité
BEGIN
IF (clk='1') AND (clk'EVENT) THEN -- TRUE sur un front montant de clk
IF (clr = '0') THEN -- si clr est à 0
q <= clrit; -- clr actif, q à 0
ELSIF (set = '0') THEN -- si set est à 0
q <= setit; -- set actif, q à 1
ELSE
q <= d; -- si ni clr ni set, alors q = d
END IF;
END IF;
END PROCESS;
END behavior;
```

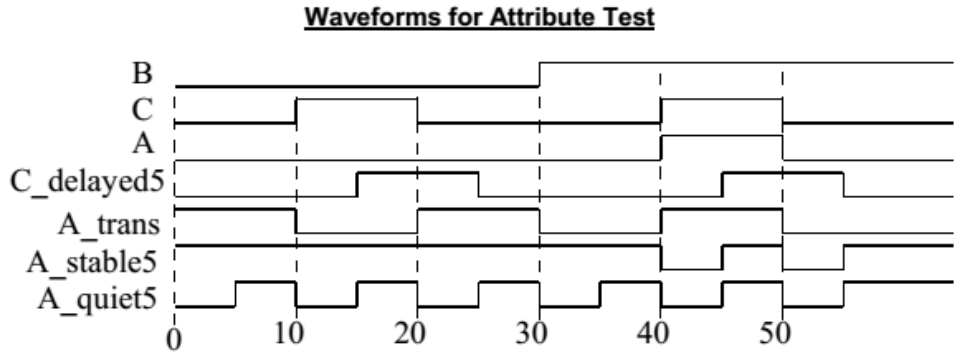
Example: process

```
   variable A,E: boolean;
   begin
      Q <= D after 10 ns;
      A := Q'active;  -- A gets a value of True
      E := Q'event;  -- E gets a value of False
      . . .
   end process;
architecture behavior of shifter is
begin
   reg: process(Rst,Clk)
   begin
      if Rst = '1' then   -- Async reset
         Qreg := (others => '0');
      elsif rising_edge(Clk) then
         Qreg := Data(Data'left+1 to Data'right) & Data(Data'left);
      end if;
   end process;
end behavior;
```

VHDL Code for Attribute Test

```
entityattr_ex is
port(B,C : inbit);
endattr_ex;
architecturetest ofattr_ex is
signalA, C_delayed5, A_trans : bit;
```

```
signalA_stable5, A_quiet5 : boolean;
begin
A <= B andC;
C_delayed5 <= C'delayed(5 ns);
A_trans <= A'transaction;
A_stable5 <= A'stable(5 ns);
A_quiet5 <= A'quiet(5 ns);
endtest;
```



Remarques

Parmi tous ces attributs prédéfinis, les plus couramment utilisés sont certainement les attributs relatifs aux caractéristiques d'un vecteur.

Associés à des types ou à des objets

Valeurs dynamiques prédéfinis mais possibilité d'attributs utilisateurs, Ils portent sur:

- les types scalaires
- les tableaux
- des signaux

Simplification de l'écriture et généricité

Les attributs se note avec une ' après le type ou l'objet

I.11.2 Définition des attributs

Au même titre qu'il est possible de définir des types, il est également possible de définir des attributs. La différence réside dans le fait que le simulateur va ignorer ces nouveaux attributs et ne reconnaître que les attributs de type prédéfinis. La définition d'un nouvel attribut ne sert qu'à ajouter des informations à une description. Ces informations seront par la suite utilisées par d'autres outils. Les outils de synthèses logiques se servent souvent de cette procédure de passage d'informations et profitent de cette opportunité pour introduire des paramètres de synthèse et spécifier, entre autres, le type de boîtier ou le type de composant à utiliser pour la synthèse d'une description[4].

I.12 LOGIQUES COMBINATOIRES ET SEQUENTIELLES

I.12.1. Logique combinatoire

Dans la logique combinatoire la sortie ne dépend pas de l'état passé. Elle peut être décrite dans tous les styles : flot de données, structurelle et comportementale.

Exemple

- **Multiplexeur à n entrées**
- **Décodeur log (n) → n**
- **Additionneur n bits**
- **Comparateur n bits**
- **ALU n bits**
- **etc.**

Exemple : Opérations logiques

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;
entity PORTES is
        port (A,B :in std_logic; Y1,Y2,Y3,Y4,Y5,Y6,Y7:out std_logic);
end PORTES;
architecture ARCH_PORTES of PORTES is
begin
Y1 <= A and B; Y2 <= A or B; Y3 <= A xor B; Y4 <= not A; Y5 <= A nand B; Y6 <= A nor B;
Y7 <= not(A xor B);
end ARCH_PORTES ;
```

Exemple : Décodeur 7 segments hexadécimal

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;
entity DEC7SEG4 is
        port (DEC :in std_logic_vector(3 downto 0);SEG:out std_logic_vector(0 to 6));
end DEC7SEG4;
architecture ARCH_DEC7SEG4 of DEC7SEG4 is
begin
SEG <= "1111110" WHEN DEC = 0 ;
ELSE "0110000" WHEN DEC = 1 ;
ELSE "1101101" WHEN DEC = 2 ;
ELSE "1111001" WHEN DEC = 3 ;
ELSE "0110011" WHEN DEC = 4 ;
ELSE "1011011" WHEN DEC = 5 ;
ELSE "1011111" WHEN DEC = 6 ;
ELSE "1110000" WHEN DEC = 7 ;
ELSE "1111111" WHEN DEC = 8 ;
ELSE "1111011" WHEN DEC = 9 ;
ELSE "-------";
end ARCH_DEC7SEG4 ;
```

I.12.2. Logique séquentielle

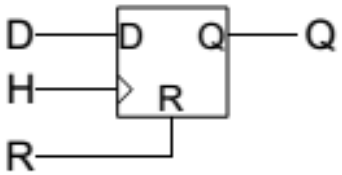
Dans la logique séquentielle la sortie dépend de son état passé. On distingue deux logiques séquentielles : asynchrone et synchrone

a) Logique séquentielle asynchrone

Logique séquentielle asynchrone est activée dès que l'une quelconque de ses entrées change d'état. Styles de description : comportementale, structurelle et flot de données. Voici quelques exemples :

Exemple : Bascule RS Asynchrone

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;
entity RS_ASYNC is
port (R,S :in std_logic;
Q :out std_logic);
end RS_ASYNC;
architecture ARCH_RS_ASYNC of RS_ASYNC is
signal X :std_logic;
begin
X <= '0' when R='1' and S='0'
else '1' when R='0' and S='1'
else X when R='0' and S='0'
else '-';
Q <= X;
end ARCH_RS_ASYNC;
```



Exemple: Bascule D asynchrone

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity D_FF is
port (H,R,D :in std_logic;
Q :out std_logic);
end D_FF;
architecture ARCH_D_FF of D_FF is
signal X :std_logic;
begin
process(H,R)
begin
if R='1' then X <= '0';
elsif (H'event and H='1') then X <= D;
end if;
end process;
Q <= X;
end ARCH_D_FF;
```

b) logique séquentielle synchrone

Logique séquentielle synchrone est activée sur occurrence d'un front d'horloge et non pas sur un changement d'état de l'une quelconque de ses entrées. Conception synchrone signifie une seule horloge et un seul front. Styles de description :

- comportementale (processus avec signal d'horloge)
- structurelle (avec des composants comportant des processus)

Remarques

Toutes les fonctions synchrones sont construites à partir de bascules D maître-esclave car c'est l'élément de base de logique séquentielle qui est présent physiquement dans

chaque cellule de quasiment tous les circuits logiques programmables (PLD, CPLD et FPGA)

Exemples

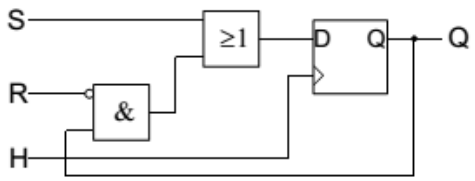
- Registre n bits (avec reset et éventuellement load/ClockEnable)
- Bascules
- Registre à décalage n bits
- Compteur n bits
- Machine à état

Remarques

Pour des raisons de simplification de câblage, on utilisait la logique asynchrone pour réaliser des fonctions complexes mettant en œuvre de nombreux boîtiers. Mais du fait de la difficulté qu'il y a à prévoir les instants de changement d'état des signaux, il apparaissait souvent des impulsions indésirables (Glitch) pouvant générer des résultats non conformes aux attentes. D'où des difficultés souvent importantes dans la mise au point. L'apparition des circuits logiques programmables permet maintenant de réaliser ces mêmes fonctions logiques complexes en fonctionnement complètement synchrone. La bonne prédictibilité des instants de changement d'État permet de réduire considérablement les aléas de fonctionnement. De l'avis unamine de tous les utilisateurs expérimentés de ce type de circuit, il faut utiliser impérativement des méthodes de synthèse synchrone. La synthèse de fonctions logiques doit être effectuée en privilégiant un fonctionnement synchrone. Néanmoins, dans le cadre d'un apprentissage, il est apparaît nécessaire d'étudier aussi quelques fonctions logiques asynchrones.

Exemple: Bascule RS synchrone

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity RS_SYNC is
port (H,R,S :in std_logic;
Q :out std_logic);
end RS_SYNC;
architecture ARCH_RS_SYNC of RS_SYNC is
signal X :std_logic;
begin
process(H)
begin
if (rising_edge(H)) then
if R='1' and S='0' then X <= '0';
elsif R='0' and S='1' then X <= '1';
elsif R='0' and S='0' then X <= X;
else X <= '-';
end if;
end if;
end process;
Q <= X;
end ARCH_RS_SYNC;
```

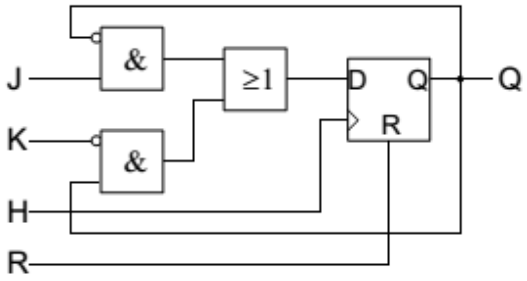


Remarques

Les bibliothèques IEEE possèdent deux instructions permettant de détecter les fronts montants) **rising_edge(CLK) = if (CLK'event and CLK='1')** ou descendants **falling_edge(CLK)= if (CLK'event and CLK='0').**

Exemple: Bascule JK

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity JK_FF is
        port (H,R,J,K :in std_logic; Q :out std_logic);
end JK_FF;
architecture ARCH_JK_FF of JK_FF is
signal X :std_logic;
begin
        process(H,R)
begin
if R='1' then X <= '0';
elsif (H'event and H='1') then
if K='1' and J='0' then X <= '0';
elsif K='0' and J='1' then X <= '1';
elsif K='1' and J='1' then X <= not X;
else X <= X;
end if;
end if;
end process;
Q <= X;
end ARCH_JK_FF;
```



Exemple : Bascule D avec Set et Reset

```
Library ieee;
Use ieee.std_logic_1164.all;
entity BASCULEDSRS is
        port ( D,CLK,SET,RESET : in std_logic;S : out std_logic);
end BASCULEDSRS;
architecture DESCRIPTION of BASCULEDSRS is
begin
        PRO_BASCULEDSRS : process (CLK)
        Begin
        if (CLK'event and CLK ='1') then
        if (RESET =’1’) then
        S <= ‘0’;
        elsif (SET =’1’)then
        S <= ‘1’;
        else
        S <= D;
end if;
end if;
end process PRO_BASCULEDSRS;
end DESCRIPTION;
```

Remarques

L'entrée **RESET** est prioritaire sur l'entrée **SET** qui est à son tour prioritaire sur le front montant de l'horloge **CLK**. Cette description est asynchrone car les signaux d'entrée **SET** et **RESET** sont mentionnés dans la liste de sensibilité du **process**.

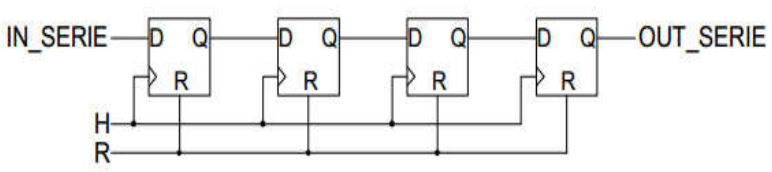
Exemple : Bascule T

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity BASCULET is
port ( D,CLK : in std_logic; S : out std_logic);
end BASCULET;
architecture DESCRIPTION of BASCULET is
signal S_INTERNE : std_logic; -- Signal interne
begin
PRO_BASCULET : process (CLK)
Begin
if (CLK'event and CLK ='1') then
if (D =’1’) then
S_INTERNE <= not (S_INTERNE);
end if;
end if;
end process PRO_BASCULET;
S <= S_INTERNE;
end DESCRIPTION;
```

Remarques: La description ci-dessus fait appel à un signal interne appelé **S_INTERNE**, pourquoi faire appel à celui-ci ? La réponse est que le signal **S** est déclaré comme une sortie dans l'entité, et par conséquent on ne peut pas utiliser une sortie en entrée. Pour contourner cette difficulté on utilise un signal interne qui peut être à la fois une entrée ou une sortie. Avec cette façon d'écrire, les signaux de sortie d'une description ne sont jamais utilisés comme des entrées. Cela permet une plus grande portabilité du code. Si on ne déclare pas de signal interne, le synthétiseur renverra certainement une erreur du type : **Error, cannot read output: s; [use mode buffer or inout].** Le synthétiseur signale qu'il ne peut pas lire la sortie **S** et par conséquent, celle-ci doit être de type **buffer** ou **inout**.

Exemple registre à decalage à droite

```
library ieee;
Use ieee.std_logic_1164.all;
entity DECAL_D is
        port (H,R,IN_SERIE :in std_logic; OUT_SERIE :out std_logic);
end DECAL_D
architecture ARCH_DECAL_D of DECAL_D is
signal Q :std_logic_vector(3 downto 0);
begin
process(H,R)
begin
if R='1' then Q <= "0000";
        elsif (H'event and H='1') then
Q <= Q(2 downto 0) & IN_SERIE;
end if;
```



```
end process;
OUT_SERIE <= Q(3)
end ARCH_DECAL_D;
```

Exemple : compteur simple

Dans ce premier exemple, nous allons créer un compteur allant de 0 à 23 et en même temps, nous allons utiliser ce signal pour générer une horloge qui est 24 fois plus lente que celle à l'entrée. Pour ce faire, il faut commencer par réaliser que, pour compter jusqu'à 23, il faut 5 bits, puisque 2^5=32). Par la suite, on devrait aussi savoir qu'un compteur est compose de flip flops et de logique. Il faut donc avoir un process séquentiel (avec clock). Un compteur devrait augmenter son compte à chaque front montant d'horloge SAUF quand il arrive à 23. Quand il arrive à 23, il doit recommencer à 0. La partie principale du programme devrait donc être :

```
PROCESS (clk)
BEGIN
IF clk'EVENT AND clk = '1' THEN
IF s_compt_sortie >= 23 THEN
        s_compt_sortie <= "00000";
ELSE
        s_compt_sortie <= s_compt_sortie + 1;
END IF;
END IF;
END PROCESS;
```

Il est à remarquer que **s_compt_sortie** n'est pas le nom du signal de sortie. VHDL n'accepte pas qu'on LISE un signal de sortie. A la place de le lire, nous allons créer un fil interne qui sera connecte à cette sortie. Nous avons le droit de lire et d'écrire aux fils internes. Donc, le compteur se fera avec le fil. Par la suite, nous connecterons ce fil interne à la sortie. Pour générer l'horloge de sortie, il faut regarder la valeur du compteur et générer '1' la moitié du temps et générer '0' durant l'autre moitié. Pour cette raison, nous avons :

```
horloge_sortie <= '0' WHEN s_compt_sortie < 12 ELSE  '1';
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY compteur IS
PORT (  clk : IN STD_LOGIC;  compt_sortie : OUT STD_LOGIC_VECTOR(4 DOWNTO 0);  horloge_sortie :
OUT STD_LOGIC  );
END compteur;
ARCHITECTURE rtl OF compteur IS
SIGNAL s_compt_sortie : STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
        PROCESS (clk)
BEGIN
IF clk'EVENT AND clk = '1' THEN
IF s_compt_sortie >= 23 THEN
        s_compt_sortie <= "00000";
ELSE
        s_compt_sortie <= s_compt_sortie + 1;
```

```
END IF;
END IF;
END PROCESS;
        compt_sortie <= s_compt_sortie;
        horloge_sortie <= '0' WHEN s_compt_sortie < 12
ELSE  '1';
END;
```

I.12.3 Machine d'états

Une machine à états finis (FSM) ou simplement une machine d'état, est un modèle de comportement composé d'un nombre fini d'états, de transitions entre ces états et d'actions. C'est comme un «graphe». Les machines d'états permettent d'effectuer la synthèse de systèmes numériques séquentiels, généralement asynchrones. Le passage d'un état à un autre s'effectue si une condition sur les entrées est remplie. Les sorties du système dépendent de l'état courant (machine de Moore) ou de l'état courant et des entrées (machine de Mealy). Dans le cas d'une FSM synchrone, la valeur des entrées est analysée sur front d'horloge.

a) Principe de fonctionnement

La machine d'état s'apparente à un automate ou un grafcet :

- Le système est dans un état stable (équivalent d'une étape pour le grafcet)
- Il peut évoluer vers un autre état du système en fonction des entrées (réceptivité VRAIE)
- A chaque état correspond une/des sorties actives (les actions associées aux étapes)

La représentation graphique est un diagramme à bulle. Ne pas oublier de faire un tableau à côté pour préciser les sorties en fonction de l'état.

b) Contraintes de fonctionnement

- Une machine d'état a besoin d'une horloge (quartz, astable en externe ou PLL embarquée)
- Une transition est activée lorsque l'étape source est active
- La réceptivité est vrai (équation logique sur les entrées)
- Un front d'horloge survient

c) Les différentes types de Finite State Machine (FSM)

- ***Machine de Moore***

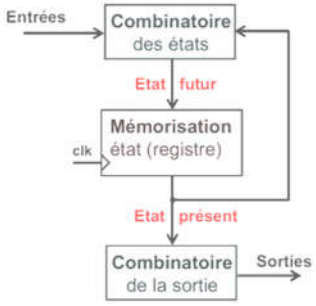


Fig.7a Machine de Moore

- La sortie ne dépend que de l'état courant
- Le temps de réaction dépend donc de l'horloge
- Cette machine peut être qualifiée de synchrone

- ***Machine de Mealy***

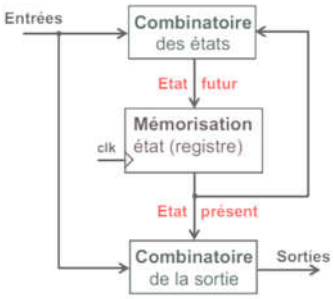


Fig.7.b Machine de Mealy

- La sortie ne dépend de l'état courant mais aussi des entrées
- Le temps de réaction est plus rapide que Moore
- Cette machine peut être qualifiée d'asynchrone
- Inconvénients : la durée minimum d'un état n'ait plus garanti => L'état peut-être interprété par un autre système comme un état parasite

Codage des états

- Création d'un type énuméré
  - Le compilateur affecte automatiquement un entier à chaque état
    T YPE etat IS (debut, etat1, etat2);
    SIGNAL etat_present : etat;
- **Codage à la main**
  - L'utilisateur affecte un entier « à la main »
  - SIGNAL etat_present : integer range 0 to 2; OU
  - SIGNAL etat_present : std_logic_vector(1 down to 0);

Gestion des transitions

- DANS UN PROCESS (car séquentiel)
- Instructions CAS/IS bien adaptée

Généralement utilisée pour décrire des systèmes séquentiels quelconques (state machine). La description du système se fait par un nombre fini d'états. Ci-dessous (Fig.)la représentation schématique d'un système à 4 états (M0 à M3), 2 sorties (S1 et S2), 2 entrées X et Y, sans oublier l'entrée d'horloge qui fait avancer le processus, et celle de remise à zéro qui permet de l'initialiser :

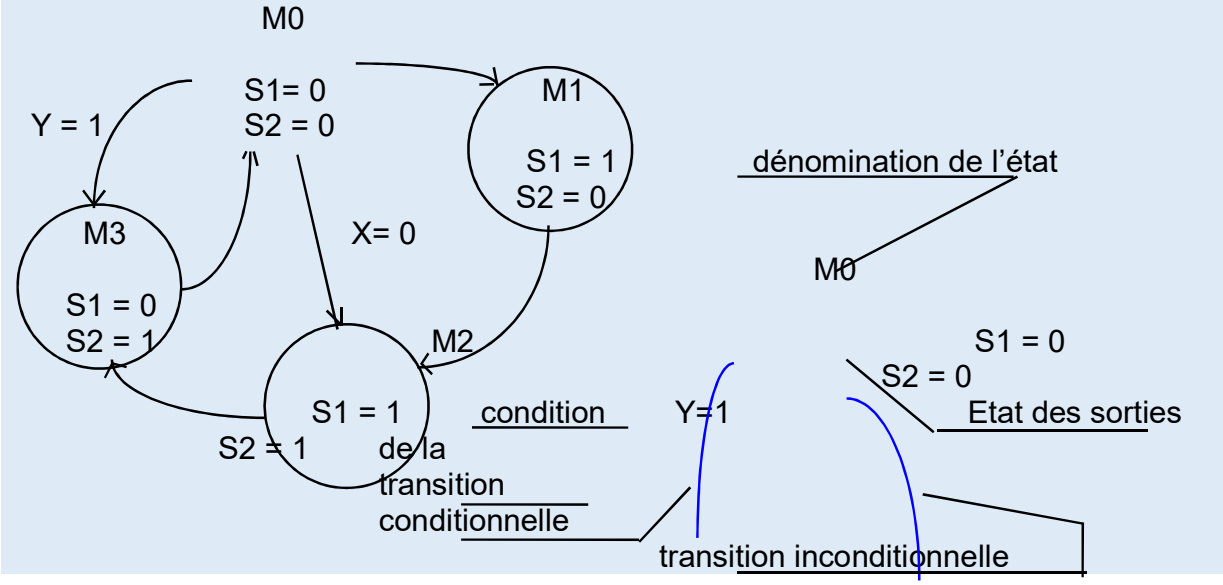


Fig.8 Système séquentiel

L'état initial est M0. Les 2 sorties sont à 0. Au coup d'horloge on passe inconditionnellement à l'état M1 sauf si la condition Y=1 a été vérifiée, ce qui mène à l'état M3 ou si X=0 a été validé ce qui mène à M2. De M3 on revient au coup d'horloge à M0. De M1 on passe à M2, et de M2 on passe à M 3... Dans chaque état on définit les niveaux des sorties.

En langage VHDL une des méthodes conseillée d'écriture des machines d'état est :
* une ENTITE
* une ARCHITECTURE de description d'état avec 2 PROCESS : Un process asynchrone et un process synchrone.

```
library ieee;
use ieee.std_logic_1164.all;
library SYNTH ;
use SYNTH.vhdlsynth.all ;
ENTITY machine1 IS
PORT (RAZ, horl :IN STD_LOGIC;                -- Ne pas oublier remise à 0 et horloge !
  X, Y :IN STD_LOGIC;
S1, S2 :OUT STD_LOGIC);
END machine1;
ARCHITECTURE diagramme OF machine1 IS
TYPE etat_4 IS (M0, M1, M2, M3);              --définir une liste des états...
```

```
SIGNAL etat,etat_suiv :etat_4 := M0;                --et 2 signaux: états courant et –-suivant, contenant la
valeur d'un état de la liste (initialisée à M0) .
BEGIN
definir_etat:PROCESS( raz, horl)                    -- "definir_etat":label optionnel
BEGIN
If raz = '1' THEN
etat <= M0;                                 --Le PACKAGE std_logic_1164
ELSIF rising_edge(horl) THEN                        --en permet l'utilisation.
 etat <= etat_suiv ;
END IF;                                             --Mise à jour de la variable
END PROCESS definir_etat;                                   d'état par l'horloge.
sorties : process (etat, x, y)                      --Le PROCESS doit contenir une...
BEGIN
CASE etat IS
WHEN M0 => S1 <= '0'; S2<= '0';
IF Y='1' then etat_suiv <= M3;
elsif X='0' then etat_suiv <= M2;
ELSE etat_suiv <= M1;
END IF;
  --Le signal de l'état suivant n'est   attribué que dans la structure CASE
WHEN M1 => S1 <= '1'; S2<= '0';etat_suiv <= M2;
WHEN M2 => S1 <= '1'; S2<= '1';etat_suiv <= M3;
WHEN M3 => S1 <= '0'; S2<= '1';etat_suiv <= M0;
END CASE;
END process sorties ;
END diagramme ;
```

I. 13. REGLE DE CONCEPTION

I.13.1 Système combinatoire asynchrone

Soit la figure suivante (**Fig.9a**):

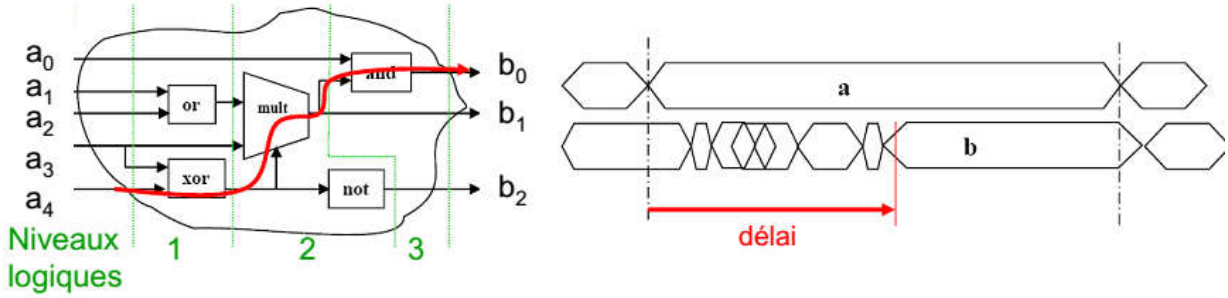


Fig.9a

Remarques :

- Plus le nombre de niveaux logiques est grand, plus le délai augmente
- Mauvaise performance en temps
- Difficile à mettre au point
- Difficile de tester tous les cas possibles

I.13.2 Système combinatoire synchrone

Soit la figure suivante (**Fig.9b**):

Remarques:

- Améliore des performances en vitesse
- Simplifie de la vérification fonctionnelle
- Autorise des analyses statiques du timing
- Assure une parfaite testabilité
- Correspond à l'architecture des composants

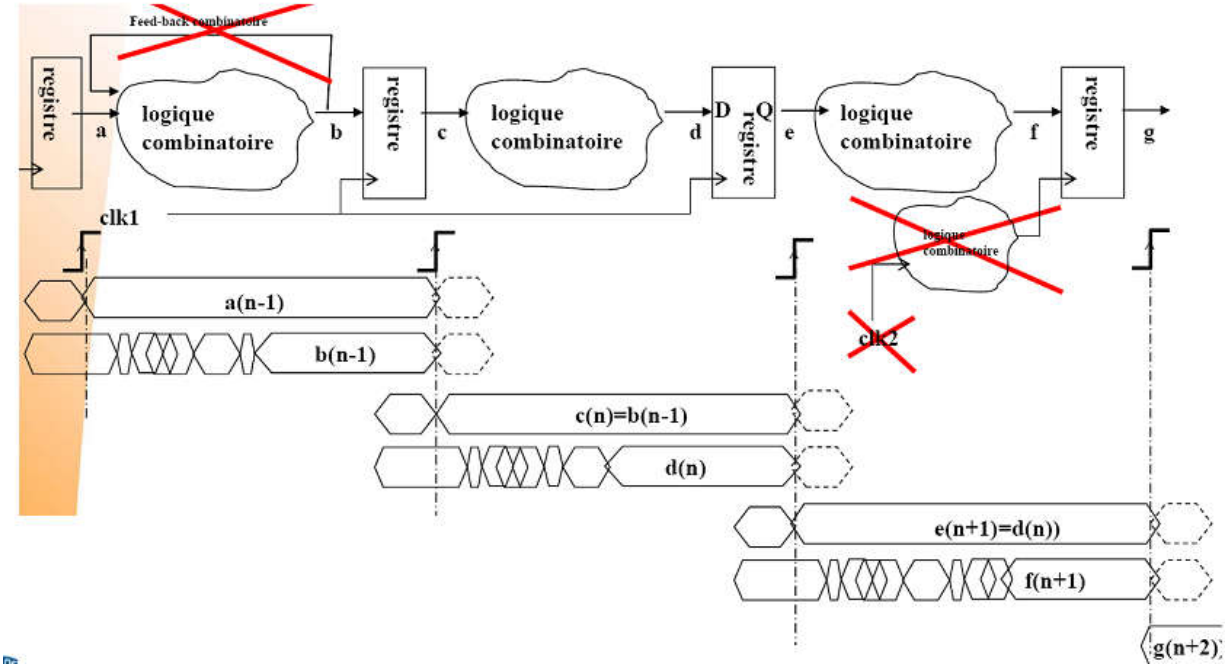


Fig.9b .

Règles à suivre :

- ✓ Une seule horloge
- ✓ Jamais de logique sur un signal d'horloge (utiliser enable)
- ✓ Si plusieurs domaines d'horloge, prévoir des FIFOs tampons

- ✓ Resynchroniser tous les signaux asynchrones pour éviter la métastabilité (2 bascules D à la suite)
- ✓ Adopter une démarche qualité (dénomination, hiérarchisation)
- ✓ Attention aux assignations incomplètes (mémorisation implicite)
- ✓ Attention à l'utilisation des variables
- ✓ Penser à l'implantation (orienter le compilateur)
- ✓ Trouver le bon compromis entre ressources et vitesse en choisissant le bon degré de parallélisme.

I.14 SIMULATION

I.14.1 Assertions

Permettent d'avoir des informations dynamiques sur la simulation :

Syntaxe: **assert** test **report** message **severity** action

ASSERT condition;
ASSERT condition REPORT "message"
ASSERT condition SEVERITY level;
ASSERT condition REPORT "message" SEVERITY level;

Si le test est négatif, on affiche un message avec arrêt ou non de la simulation en fonction de l'action.

Action	Effets	
NOTE	poursuite	néant
WARNING	poursuite	écriture sur fichier .log
ERROR	poursuite	écriture sur écran
FAILURE	arrêt	écriture sur fichier

now = temps de simulation

```
assert (now<10 ms)report "fin de simu" severity failure ;
```

```
architecture comp of dut_tb is
    ...
begin

    -- L'assertion est vérifiée à chaque fois que BCD_obs change
    assert (BCD_obs>9) report "BCD a dépassé sa capacité"
                       severity WARNING;

    process test
    begin
        ...
        -- L'assertion n'est vérifiée que lorsque l'exécution du
        -- processus passe à cette instruction
        assert (A_obs=A_ref) report "résultat incorrect pour A"
                             severity ERROR;
        ...
        assert (A_obs=15) report "La sortie A ne vaut pas 15"
                             severity ERROR;
    end test;
end comp;
```

Il est possible d'afficher la valeur d'un signal ou d'une constante type'image (signal)

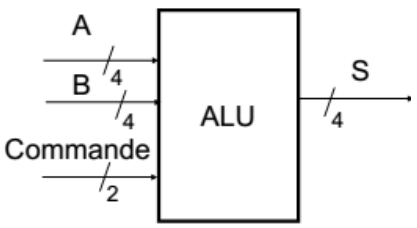
```
signal entier : integer;
signal unbit  : std_logic;
...
process
begin
    ...
    report "La valeur de entier est" & integer'image(entier);

    report "Le signal unbit vaut: " & std_logic'image(unbit);
end process;
```

Attention, pas de fonction prévue pour le type **std_logic_vector**

Exemple

Ecrire le fichier vhdl permettant de simuler une ALU qui peut réaliser les opérations suivantes :

S = A si commande = 00
S = B si commande = 01
S = A + B si commande = 10
S = A – B si commande = 11



Lorsqu'une opération est réalisée, utiliser les assertions pour faire apparaître un message d'erreur s'il y a une erreur de calcul. A la fin de la simulation, faire apparaître «Fin de simulation».

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


entity alu_tb is
end alu_tb ;


architecture A of alu_tb is

component ALU
port(a : in integer range 0 to 15;
    b: in integer range 0 to 15;
    s : out integer rango 0 to 15;
    commande : in std_logic_vector (1 downto 0);
end component;

signal sig_A : integer := 9;
signal sig_B : integer := 3;
signal sig_S : integer;
signal sig_com : std_logic_vector := "00";
constant add : integer := 12;
constant sous : integer := 6;
```

```
begin

    report "A="& integer'image(sig_A);
    report "B="& integer'image(sig_B);

    process (sig_com)
    begin
    sig_com <= "00";
    assert (sig_S=sig_A) report "résultat incorrect pour S=A. S= " & integer'image(sig_S) severity ERROR;
    wait for 20 ns;

    sig_com <= "01";
    assert (sig_S=sig_B) report "résultat incorrect pour S=B. S= " & integer'image(sig_S) severity ERROR;
    wait for 20 ns;

    sig_com <= "10";
    assert (sig_S=add) report "résultat incorrect pour S=A+B. S= " & integer'image(sig_S) severity ERROR;
    wait for 20 ns;

    sig_com <= "11";
    assert (sig_S=sous) report "résultat incorrect pour S=A-B. S= " & integer'image(sig_S) severity ERROR;
    wait for 20 ns;

    assert (now<60 ns) report "Fin de simulation" severity FAILURE;

    end process;


    ALUtest : ALU port map (sig_A, sig_B, sig_S, sig_com);
```

I.14.2. Paramètres génériques

VHDL fournit l'instruction GENERATE pour créer facilement des structures bien structurées. Toute déclaration concurrente VHDL peut être incluse dans un Instruction GENERATE, y compris un autre GENERATE déclaration. Deux façons :

label : FOR identifier IN range GENERATE
concurrent_statements;
END GENERATE [label];

label : IF (boolean_expression) GENERATE
concurrent_statements;
END GENERATE [label];

Permettent de paramétrer des composants

- Se déclarent dans l'entity
- On leur donner une valeur par défaut (:=)
- L'instanciation se fait avec generic map (…)

Exemple

```
entity mux is
 generic (width : integer :=8);
 port (sel :in std_logic;
        a,b : in std_logic_vector (width-1 downto 0);
          c : out std_logic_vector (width-1 downto 0));
end mux;

architecture behav of mux is
begin
c<=a when sel='0' else b;
end behav;

comp1 : mux generic map ( 4)
port map(sel=>seldata, a=> dataa, b=> datab, c=>data);
comp2 : mux port map (sel=>seladr, a=>adra, b=> adrb, c=>adr);
```

a, b et c ont 4 bits de largeur

a, b et c ont 8 bits de largeur (valeur par défaut)

Exemple : Additionneur 4 bits en utilisant l'instruction Generate

```
Entity Adder4 is
port(A, B: inbit_vector(3 downto0); Ci: in bit;  -- Inputs
S: outbit_vector(3 downto0); Co: outbit);  -- Outputs
End Adder4;
Architecture Structure ofAdder4 is
Component FullAdder
port(X, Y, Cin: in bit; -- Inputs
Cout, Sum: outbit); -- Outputs
end component;
signalC: bit_vector(4 downto0);
begin
C(0) <= Ci;
-- generate four copies of the FullAdder
Full Add4: fori in0 to3 generate
begin
FAx: Full Adder port map(A(i), B(i), C(i), C(i+1), S(i));
end generateFullAdd4;
Co <= C(4);
End Structure;
```

ANNEXES

Annexe 1

Reserved Word	Purpose
abs	Arithmetic operator for absolute value. Unary operator, predefined for any numeric type.
access	A variety of data type whose values are pointers to (or links to, or addresses of) dynamically-allocated objects of some other type.
after	Clause used to include delay information in a signal assignment. If there is no after clause, default delay of one simulation delta is assumed.
alias	Declares an alternate name for all or part of an existing object.
all	Suffix for identifying all declarations that are contained within the package or library denoted by the prefix.
and	Logical operator for types bit and Boolean and for one-dimensional arrays of these types.
architecture	Statement that contains description of the design.
array	A composite type in which all values belong to the same data type (e.g., string is an array of the data type character).
assert	Statement that presents a condition to be evaluated. Often used in conjunction with reporting of error messages.
attribute	A named characteristic of items belonging to one of the following classes: Types, subtypes Procedures, functions Signals, variables, constants Entities, architectures, configurations, packages Components Statement labels An

Reserved Word	Purpose
	attribute declaration declares an attribute name and its type. An attribute specification associates an attribute with a name and assigns a value to the attribute. Predefined attributes exist for types, arrays, and signals.
begin	Marks the beginning of the statement portion (as opposed to the declarative portion) of a process statement or architecture body.
block	Concurrent statement used to partition a design.
body	Conjoined with package. A package body stores the definitions of functions, procedures, and the complete constant declarations for any deferred constants that appear in a corresponding package declaration. The name of the package body is the same as that of the package declaration to which it refers.
buffer	A mode that enables a port to be read and updated within the entity model. A buffer port cannot have more than one source, and can be connected only to another buffer port or to a signal with no more than one source.
bus	A kind of signal that represents a hardware bus. When all drivers to the signal become disconnected, the signal's value is determined by calling the resolution function with all the drivers off. Any previous value is lost. Bus signals may be either a port or locally declared signal.
case	A form of conditional control that selects statements for execution based on the value of a given expression.
component	Declaration made in a top level entity to instantiate lower-level entities.
configuration	Associates particular component instances with specific design entities, and associates entity declarations with specific architectures.
constant	A class of data object. Constants can hold a single value of a given type. If the value is not specified, the constant is a deferred constant, and can appear inside a package declaration only.
disconnect	Specifies the disconnect time for a guarded signal.
downto	Specifies direction in a range.
else	Optional clause in an if statement. An else clause specifies alternative statements when the if clause and any elsif clauses evaluate false.
elsif	Clause in an if statement that poses an alternative condition when the if clause evaluates to false.
end	Marks the end of a statement, subprogram, or declaration of a library unit.
entity	Specifies input and output definitions of the design.
exit	Causes execution to jump out of the innermost loop or the loop whose label is specified.
file	A category of data type. File types provide a way for a VHDL design to communicate with the host environment. File type is declared with a file type definition, while files are declared with a file declaration.
for	Used to iterate a predetermined number of replications in replicated logic, such as generate and loop statements. Also used in specifying blocks, components, and configurations, and in specifying time expression in a timeout clause.
function	A subprogram used for computing a single value. Functions are always terminated by a return statement, which returns a value. Functions are specified with a subprogram specification.
generate	Replicates one or more concurrent statement. Can be in for or if format.
generic	Passes environment information to subcomponents; can be declared in the same constructs in which ports can be declared. Generics are of the object class constant. The declaration of a generic may also include a default value, which will be used if an actual value is missing in the generic map.

Reserved Word	Purpose
guarded	Option for a concurrent signal assignment. The guarded option specifies that the signal assignment statement will execute only when the guard condition of the block statement that contains the assignment is true.
if	Conditional logic statement. Presents a condition to be evaluated as true or false.
impure	Option for a function in a subprogram specification. Use of this reserved word extends the scope of variables and signals declared outside of the function to be available to that function, resulting in the possibility that the function may return different values when called multiple times with the same actual parameter values.
in	Port mode that allows the port to be read only. If no mode is specified, in is assumed.
inertial	An option for specifying delay mechanism in a signal assignment statement. Inertial delay is characteristic of switching circuits: a pulse whose duration is shorter than the switching time of the circuit will not be transmitted or in the case that a pulse rejection limit is specified, a pulse whose duration is shorter than that limit will not be transmitted.
inout	Port mode that allows a bidirectional port to be read and updated within the entity model.
is	Reserved word that equates the identity portion to the definition portion of a declaration.
label	An entity class, to be stated during attribute specification of user-defined attributes.
library	A context clause that makes visible the logical names of design libraries that can be referenced within a design unit. The following library clause is implied for every design unit: library std, work
linkage	A port mode similar to inout used to connect VHDL ports to non-VHDL ports.
literal	An entity class, to be stated during attribute specification of user-defined attributes.
loop	Statement used to iterate through a set of sequential statements.
map	With port or generic, associates port names within a block (local) to names outside a block (external). A port of mode may be left unconnected either by omitting it from the port map, or by connecting it to the reserved word open. In either case, the corresponding port declaration must include a default value.
mod	Arithmetic operator for modulus. Modulus is predefined for any integer type; the operands and the result are of the same type. The result of a mod operator has the sign of the second operand and is defined (for some integer n) as: a mod b = a-b*n
nand	Logical operator for types bit and Boolean and for one-dimensional arrays of these types. Complement of and.
new	An allocator that enables objects of a specific type to be created dynamically. These dynamically-created objects are accessed by access types.
next	Statement that causes the current iteration of the specified loop to be prematurely terminated, resuming execution with the next iteration of the loop.
nor	Logical operator for types bit and Boolean and for one-dimensional arrays of these types. Complement of or.
not	Unary logical operator for types bit and Boolean.
null	Sequential statement that causes no action to take place; execution continues with the next statement.
of	Reserved word used to link an identifier to its entity name, and used when specifying type mark in a file type definition.
on	Used to introduce the sensitivity list in the sensitivity clause of a wait statement.

Reserved Word	Purpose
open	An entity aspect, used as a binding indication to indicate that binding is not yet specified and that it is to be deferred.
or	Logical operator for types bit and Boolean and for one-dimensional arrays of these types.
others	When used as the last branch of case statement, used to cover all values not specified by when statements. Can also be used as part of the right-hand side of a signal or variable assignment statement for array types. This assigns values to array elements not otherwise assigned.
out	Port mode that enables the port to be updated only. It cannot be read.
package	Optional library unit for making shared definitions (usually type definitions). You must issue a use statement to make the package available to other parts of the design.
port	Signals through which an entity communicates with the other models in its external environment.
postponed	Option for a concurrent signal assignment or process statement.
procedure	Subprogram used to partition large behavioral descriptions. Procedures can return zero or more values.
process	A process represents a level of hierarchy in a design. The statements contained in the **process_statement_part**run sequentially (from top to bottom) rather than concurrently. If the process includes the optional **sensitivity_list**, the **process_statement_part** is executed only when there is an event on one or more of the signals listed in the **sensitivity_list**. For simulation, the **process_statement_part** of all processes is executed once when the simulation initializes. Processes with sensitivity lists will not execute again until there is an event on one of the signals in the **sensitivity_list**. Processes without a sensitivity list will continue to re-execute their **process_statement_part** for the remainder of the simulation. This implies that the **process_statement_part** should include at least one wait statement. Otherwise, the simulation time will not advance and the simulator will appear to be frozen. For synthesis, processes may be used to infer either sequential (clocked) or combinational logic. Processes intended to infer combinational logic should include in the **sensitivity_list** all signals that affect the behavior of the **process_statement_part**. This includes not only signals appearing on the right-hand side of signal or variable assignment statements, but also signals or variables appearing as part of conditional statements such as if or case. Processes that infer synchronous logic should include the clock signal and any asynchronous controls (asynchronous resets or presets) in the sensitivity list. In general, processes may or may not be synthesizable, depending on the details of how they are written. See the IEEE VHDL User Manual for details.
pure	Option for a function in a subprogram specification. A pure function will disallow the use of any signals or variables declared outside of the function. All functions are pure unless specified as impure.
range	Parameter used when specifying subtypes in an array type declaration.
record	A composite data type in which the collection of values may belong to the same or different types.
register	A kind of signal which models a latch. If all drivers to such a signal are disconnected, the signal retains its old value.
reject	An option for specifying delay mechanism in a signal assignment statement. Every inertially delayed signal assignment has a pulse rejection limit. If the delay mechanism specifies inertial delay, and if the reserved word reject followed by a time expression is present, then the time expression specifies the pulse rejection limit. In all other cases, the pulse rejection limit is specified by the time expression associated with the first waveform element. Not supported for synthesis.
rem	Arithmetic operator for remainder. Remainder is predefined for any integer type; the operands and the result are of the same type. The result of a rem operator has the sign of the first operand and is defined as: a rem b = a-(a/b)*b
report	Statement for generating report messages. Not supported for synthesis.

Reserved Word	Purpose
return	Statement that causes a subprogram to terminate, returning control back to the calling object. All functions must have a return statement, and the value of the expression in the return statement is returned to the calling program. For procedures, objects of mode out and inout return their values to the calling program.
rol	Shift operator: rotate left. Shift operators are defined for any one-dimensional array type whose element type is either bit or Boolean. The arguments of rol are the array that will be rotated and the amount by a which it will be rotated.
ror	Shift operator: rotate right. Shift operators are defined for any one-dimensional array type whose element type is either bit or Boolean. The arguments of rol are the array that will be rotated and the amount by a which it will be rotated.
select	Expression whose value determines different values for a target signal in a selected signal assignment statement.
severity	A predefined type in the language with values note, warning, error, and failure.
shared	An type of variable that can be declared only in entities, architectures, and generates. A shared variable can be accessed by all three of the subprograms/processes local to the declarative region.
signal	Represents a wire or a placeholder for a value. Signals are assigned in signal assignment statements, and declared in signal declarations. Note that signal assignments always occur with some amount of delay. In the absence of the optional delay_mechanism, signal assignments will occur one delta delay after the signal assignment statement is executed. This fact has major implications when a signal assignment is executed as part of a block of sequential statements within a process. See the IEEE VHDL User Manual for details.
sla	Shift operator: shift left arithmetic. Shift operators are defined for any one-dimensional array type whose element type is either bit or Boolean. The arguments of sla are the array that will be shifted and the amount by a which it will be shifted. This shift operator will fill with the leftmost bit.
sll	Shift operator: shift left logical. Shift operators are defined for any one-dimensional array type whose element type is either bit or Boolean. The arguments of sll are the array that will be shifted and the amount by a which it will be shifted. This shift operator will fill with zeros.
sra	Shift operator: shift right arithmetic. Shift operators are defined for any one-dimensional array type whose element type is either bit or Boolean. The arguments of sra are the array that will be shifted and the amount by a which it will be shifted. This shift operator will fill with the rightmost bit.
srl	Shift operator: shift right logical. Shift operators are defined for any one-dimensional array type whose element type is either bit or Boolean. The arguments of srl are the array that will be shifted and the amount by a which it will be shifted. This shift operator will fill with zeros.
subtype	A declaration that defines a base type and a constraint. The constraint specifies a subset of values for the base type. An object is said to belong to a subtype if it is of the base type and if it satisfies the constraint.
then	Introduces statements to execute when the preceding if or elsif statement evaluates true.
to	Specifies direction in a range.
transport	An option for specifying delay mechanism in a signal assignment statement. Transport delay is characteristic of hardware devices, such as transmission lines, that exhibit nearly infinite frequency response: any pulse is transmitted, no matter how short its duration. Not supported for synthesis.

type	Data type. Each data type has a set of values and a set of operations associated with it. User-defined types are created with type declarations. Predefined types can be divided into several categories: scalar, composite, access, and file. In addition, there are non-predefined types established by the IEEE standard 1164. Each of these types is listed below.
	• Scalar Types
	  ○ Enumerated
	  ○ Character (literals: 128 characters of the ASCII character set)

Reserved Word	Purpose
	○ Bit Boolean (literals: true, false) ○ Severity_level (literals: note, warning, error, failure) ○ Numeric ○ Integer ○ Physical ○ Floating_point • Composite Types ○ Array ○ String (1-dimensional array of type character) ○ Bit_vector(1-dimensional array of type bit) ○ Record ○ Access Types (see access) ○ File Types (see file) • Non-predefined Types: In addition to the predefined types, IEEE standard 1164 adds the following types that are commonly used for modeling digital logic: ○ Std ulogic (an enumerated type with the values 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-') ○ Std_logic (same as std_ulogic except that this is a resolved type) ○ Std_ulogic_vector (an array of std_ulogic) ○ Std_logic_vector (an array of std_logic) • Predefined-Types: Types defined by the IEEE standard 1076.3 are: ○ Unsigned (an array of std_logic) ○ Signed (an array of std_logic) ○ Overloaded arithmetic and conversion operators for types unsigned and signed are defined in the package numeric_std. More information on selecting a data type can be found in the Design Considerations section of the IEEE VHDL Reference Manual.
unaffected	Concurrent statement that causes no action to take place; execution continues with the next statement.
units	An entity class, to be stated during attribute specification of user-defined attributes. Also used in physical type definition statement.
until	Part of the condition clause of a wait statement.
use	Clause that makes the contents of a package visible from inside an entity or an architecture.
variable	Declared inside a process statement with a variable declaration; assigned with a variable assignment statement. Variables are created at the time of elaboration and retain their values throughout the entire simulation run. Note that variable assignments occur without delay (unlike signal assignments). This has major implications when variable assignments are used as part of a block of sequential statements within a process. See the VHDL User Manual for details.
wait	Suspends evaluation of a process. There are three basic forms of a wait statement: wait on sensitivity_list; wait until boolean_expression; wait for time_expression; These forms can be combined: wait on sensitivity_list until boolean_expression for time_expression;
when	Used to present choices for conditional logic in a case statement.
while	Used to iterate replications in replicated a loop statement. Also used for conditional logic in selected waveforms.
with	Introduces the select expression of a selected signal assignment statement.
xnor	Logical operator for types bit and Boolean and for one-dimensional arrays of these types. Logical exclusive nor.
xor	Logical operator for types bit and Boolean and for one-dimensional arrays of these types. Logical exclusive or.

Annexe 2

```
type bit is (‘0’,’1’);
type boolean is (false,true);
type character is (‘A’,’B’,...,’a’,...,’0’,...,’*’,...);
type severity_level is (note,warning,error,failure);
type integer is range -2 147 483 648 to 2 147 483 648;
type real 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_808;
units : fs; ps=1000 fs; ns=1000 ps;...;min=60 sec;hr=60 min;
type bit_vector is array (natural range <>) of bit;
type string is array (natural range <>) of character;
type COULEUR is (R,V,O);
type INDEX is range 0 to 100
```

REFERENCES

1. Perry, D. L. (2002). *VHDL: programming by example* (Vol. 4). New York, NY, USA: McGraw-Hill.
2. Chu, P. P. (2011). *FPGA prototyping by VHDL examples: Xilinx Spartan-3 version*. John Wiley & Sons.
3. Ashenden, P. J. (2010). *The designer's guide to VHDL* (Vol. 3). Morgan Kaufmann.
4. Sjoholm, S., & Lindh, L. (1997). *VHDL for Designers*. Prentice Hall PTR.
5. Airiau, R., Bergé, J. M., Rouillard, J., & Olive, V. (1998). *VHDL: langage, modelisation, synthese*. PPUR presses polytechniques.
6. https://www.academia.edu/30401224/Le_langage_VHDL