heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Département TIC
Institut REDS

Manuel VHDL

synthèse

et

simulation

Version partielle septembre 2007

(Chapitres : 1, 2, 3, 4, 6, 7, 9, 10, 15)

Etienne Messerli
Version 6-a
Septembre 2007

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale

Auteur et version du manuel

La première version de ce manuel a été écrite par Yves Sonnay dans le cadre du cours à option "Pratique du VHDL" donné par le professeur E.Messerli.

Version :

Initiale :	**Manuel VHDL pour la synthèse automatique** Yves Sonnay, étudiant à l'EIVD, octobre 1998
4ème révision :	E. Messerli, professeur à l'EIVD, mai 2003
5ème révision :	E. Messerli, professeur à la HEIG-VD, septembre 2005 Remise à jour complète (rajout chapitre 8 et 11 sur VHDL avancé)

Remise à jour :

6ème révision :	E. Messerli, professeur à la HEIG-VD, septembre 2007 Complété notion avancée, quelques corrections

Mise à jour de ce manuel

Je remercie tous les utilisateurs de ce manuel de m'indiquer les erreurs qu'il comporte. De même, si des informations semblent manquer ou sont incomplètes, elles peuvent m'être transises, cela permettra une mise à jour régulière de ce manuel.

Contact

Auteur: Etienne Messerli
e-mail : etienne.messerli@heig-vd.ch
Tel: +41 (0)24 / 55 76 302

Institut REDS, Reconfigurable & Embedded Digital Systems
HEIG-VD, Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud
Route de Cheseaux 1
CH-1401 Yverdon-les-Bains
Tel : ++41 (0)24 / 55 76 330 (central)
Fax : ++41 (0)24/ 55 76 404
E-mail : reds@heig-vd.ch
Internet : http://reds.heig-vd.ch/

Autres personnes à contacter en cas d'absence:

M. Corbaz Alexandre	e-mail : alexandre.corbaz@heig-vd.ch	tél : +41 (0)24/55 76 273
M. Graf Yoan	e-mail : yoan.graf@heig-vd.ch	tél : +41 (0)24/55 76 258
M. Perez-Uribe Andres	e-mail : andres.perez-uribe@heig-vd.ch	tél : +41 (0)24/55 76 274
M. Starkier Michel	e-mail : michel.starkier@heig-vd.ch	tél : +41 (0)24/55 76 155

Table des matières

Préambule

Ce manuel a pour but de fournir toutes les connaissances nécessaires à l'utilisation du langage VHDL pour la synthèse automatique. Le manuel inclu aussi la méthodologie pour réaliser des simulations automatiques.

Le manuel est organisé pour un apprentissage progressif du langage. A la fin de la lecture de ce manuel le lecteur sera capable de décrire, en VHDL synthétisable, tout système numérique qu'il souhaite réaliser. Il sera aussi capable d'écrire les bancs de test automatique afin d'utiliser la nouvelle méthodologie de développement.

Ce manuel n'a pas pour objectif de présenter l'ensemble du langage VHDL. Celui-ci est introduit de manière progressive. Certaines notions ont été volontairement écartée afin de facilité l'apprentissage. Le recourt à un ouvrage exhaustif sera nécessaire pour la personne souhaitant connaître l'ensembe du langage. Le manuel fourni toutes les informations nécessaires pour maîtriser les description pour la synthèse automatique et des bancs de test. Le lecteur trouvera en annexe la liste des notions VHDL non traité dans cet ouvrage ()

Nous allons indiquer la suite des chapitres à lire pour un apprentissage progressif du langage. Il est indispensable que le lecteur dispose des exercices liés à ce manuel et qu'il les pratique à chaque étape.

La succession proposée est conseillée pour un débutant. Si vous êtes déjà un utilisateur confirmé, vous pouvez directement lire les chapitres qui vous intéresse.

Le chapitre 1 est, comme son nom l'indique, une introduction au VHDL.

Le chapitre 2 traite des concepts de base du langage VHDL. Cette partie est indispensable pour comprendre le langage. Une bonne compréhension de ces concepts sera obtenue avec leur utilisation dans les chapitres suivants.

Comme nous l'avons déjà dit, l'apprentissage sera progressif. Nous allons donc commencer par décrire des systèmes combinatoires. Nous débuterons avec des descriptions utilisants uniquement des instructions concurrentes. Le lecteur lira donc le chapitre 3 (sans le paragraphe 3-5) puis il appliquera directement ces notions avec le chapitre 9 (9-1 à 9-3 et 9-5) et les exercices associés.

L'étape suivante est d'améliorer l'utilisation du langage, mais toujours pour décrire des systèmes combinatoires. Il s'agira d'utiliser des instructions séquentielles. Le lecteur commence ainsi à entrer dans le monde VHDL en utilisant des descriptions comportementales ou algorithmiques. Le passage incontournable sera l'étude de l'instruction process (paragraphe 3-5). Ensuite, il lira le chapitre 4, la suite du chapitre 9 (paragraphe 9-4) et il pratiquera les exercices.

Dès que le lecteur se sentira assez sûr avec ces thèmes, il pourra étudier l'utilisation du paquetage Numeric_Std, chapitre 7, et l'appliquer aux exercices correspondant.

La prochaine étape consiste à aborder la description de systèmes séquentiels. Le lecteur lira le chapitre 10 et applique les notions avec les exercices.

Les notions de simulation sont décomposées en trois parties. Une partie générale (chapitre 12) et un chapitre pour expliquer la simulation de systèmes combinatoires (chapitre 13) ou séquentiels (chapitre 14). Le lecteur peut commencer l'étude de la simulation après avoir maîtrisé la description des systèmes combinatoires.

Finalement, le lecteur pourra aborder les notions avancées et les descriptions réutilisable (chapitres 8 et 11)

Nous pouvons proposer deux démarches au lecteur. Voici la première démarche qui commence par les descriptions synthétisables:

- Description de systèmes combinatoires
- Description de systèmes séquentiels
- Simulation des systèmes combinatoires
- Simulation des systèmes séquentiels
- Notions avancées et descriptions génériques

La seconde démarche propose d'imbriquer plus intensément la simulation avec l'étude du langage pour la synthèse. Cette démarche à l'avantage d'inté-

grer plus rapidement les deux utilisations du langage. Voici cette seconde démarche:

- Description de systèmes combinatoires
- Simulation des systèmes combinatoires
- Description de systèmes séquentiels
- Simulation des systèmes séquentiels
- Notions avancées et descriptions génériques

Chapitre 1

Introduction

Le langage VHDL est utilisable pour de nombreuses applications. Ce manuel abordera de manière approfondie l'utilisation du langage pour la synthèse automatique. Une introduction sera donnée sur l'utilisation du langage VHDL pour la vérification. Le manuel couvre les parties couramment utilisées du langage VHDL. De nombreux exemples seront à disposition afin de rendre plus aisé la compréhension de ce langage pour la synthèse automatique. Nous donnons plusieurs châblons de bancs de test pour réaliser des simulations automatiques.

A la fin de la lecture de ce document, le lecteur sera capable de décrire le comportement d'un circuit numérique en VHDL synthétisable. Il sera aussi capable d'écrire les bancs de test en VHDL afin de vérifier le comportement de ces descriptions. Finalement, il pourra intégrer sa solution dans un circuit logique en réalisant une synthèse automatique de ces descriptions. Il existe des circuits logiques programmables de type CPLD ou FPGA. Il existe aussi des circuits logiques «Full custom» de type ASIC. Dans le cadre de ce manuel nous parlerons des circuits logiques programmable. L'ensemble du manuel est aussi utilisable pour des circuits de type ASIC sauf les thèmes spécifiques au placement-routage qui sont alors différents.

Avant de plonger directement dans la syntaxe du langage VHDL, il est utile d'avoir une idée du cheminement des opérations qui conduit de la description VHDL à la programmation du circuit en passant par la synthèse automatique.

1-1 Les différentes étapes de la synthèse automatique

Nous allons expliquer les différentes étapes qu'il y a de la description VHDL jusqu'au circuit. Il est important de bien comprendre la fonction de cette étape de synthèse. Cela permettra de mieux appréhender les structures à utiliser pour les descriptions VHDL.

La phase de conception ne fait pas partie de ce manuel. Elle doit être faite préalablement à la description du système en VHDL

Schéma des différentes étapes

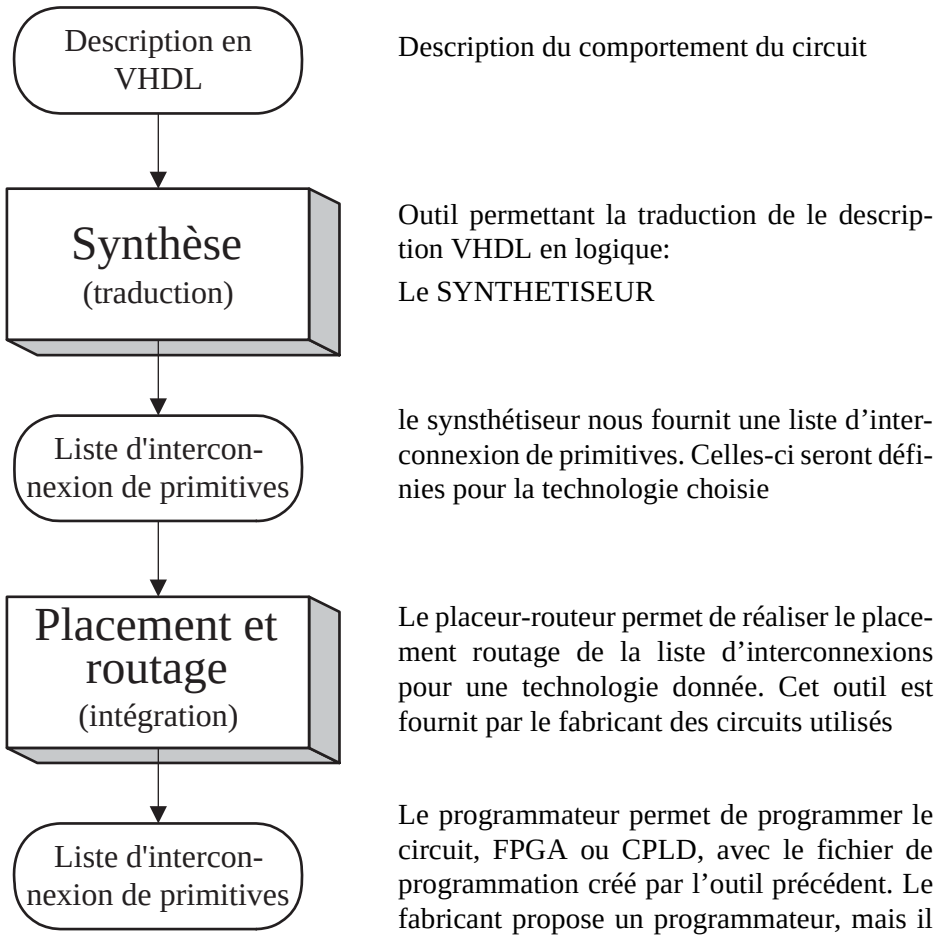


Figure 1- 1 : Schéma des différentes étape pour la réalisation d'un circuit

Le déroulement de la synthèse automatique

Le premier outil utilisé dans les étapes de synthèse automatique est un éditeur de texte avec lequel on décrit le comportement du circuit logique en langage VHDL. Ensuite, la description VHDL est traduite par l'outil de synthèse. Cette étape est très importante, il s'agit de convertir la description de haut niveau (VHDL) en fonctions logiques. Le synthétiseur génère en sortie une liste d'interconnexion de composants logiques de base existant dans la technologie cible choisie. Le placeur-routeur crée, à partir du fichier généré par le synthétiseur, un fichier spécifique à la technologie cible qui décrit les connexions entre les primitives (composants élémentaires) à disposition dans le circuit programmable. Finalement, l'intégration physique de la description est effectuée à partir de ce fichier spécifique dans le circuit programmable.

1-1.1 Mise en garde

Une mise en garde est à faire par rapport à l'outil de synthèse VHDL:

Les descriptions VHDL qui sont données en exemple dans ce manuel utilisent la norme IEEE 1076-1993, souvent abrégé VHDL-93. Elles ont été synthétisées à l'aide de l'outil Leonardo Spectrum de la société Mentor Graphics. Les paquetages Std_Logic_1164 et Numeric_Std ont été utilisés. Les descriptions synthétisables respectent la norme IEEE 1076.6-1999. Celle-ci définit la syntaxe pour le sous-ensemble utilisable pour les descriptions de type *Register Transfer Level* (RTL).

Il est dès lors possible qu'une synthèse correcte des descriptions de ce manuel ne soit pas obtenue lors de l'utilisation d'autres logiciels. Il est possible que ces outils ne disposent pas du paquetage Numeric_Std ou qu'ils ne respectent pas la norme IEEE 1076.6-1999 (règles de description RTL). Il est aussi possible que certain synthétiseur n'accepte pas les possibilités avancées du langage. Nous pouvons citer les génériques, les vecteurs non contraints, etc. Il peut donc être nécessaire d'apporter quelques modifications aux descriptions, afin de les rendre synthétisables par d'autres outils.

Les simulations ont été effectuées avec le logiciel ModelSim de la société Model Technology. L'utilisation d'un autre simulateur ne pose pas de problèmes particuliers si celui-ci supporte la norme VHDL-93 et dispose des paquetages Std_Logic_1164 et Numeric_Std.

1-2 Méthodologie de vérification

L'utilisation d'un langage HDL apporte beaucoup d'avantage. Le plus important est lié à la méthodologie de vérification. L'objectif est de valider le plus tôt possible le fonctionnement d'un système. Nous allons présenter les différentes possibilités de vérifications.

Un outil indispensable actuellement est le simulateur. A toutes les étapes de la réalisation d'un circuit logique, nous disposons d'un fichier en VHDL permettant d'utiliser le mêm banc de test. La figure 1- 2, page 4, nous montre ces différentes possibilités de simulation.

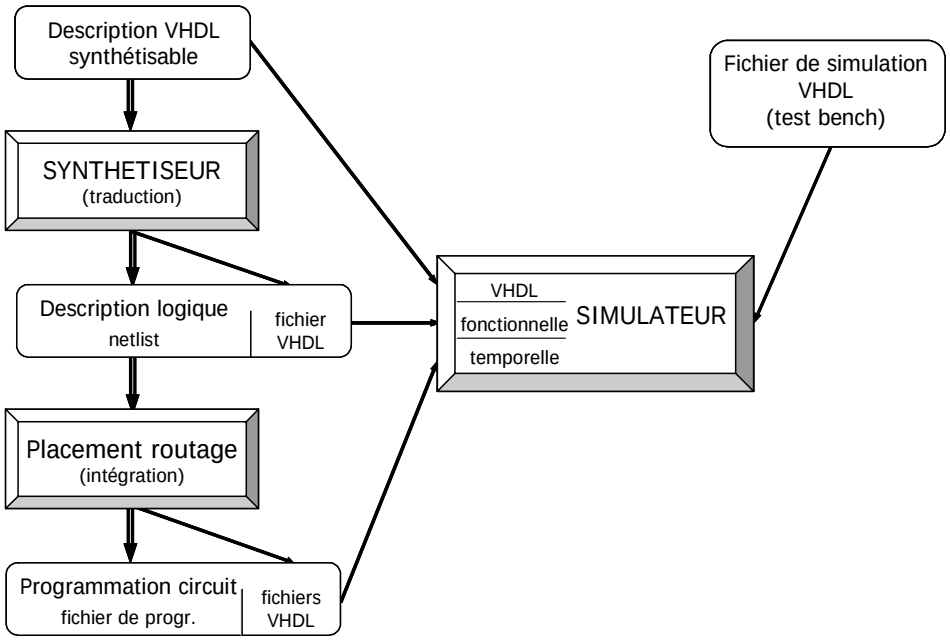


Figure 1- 2 : Etapes de réalisation d'un circuit avec les phases de simulation

La première constatation est que nous utilisons toujours le même fichier de simulation. L'investissement de temps investit sera réutilisé plusieurs fois. Il est important que la simulation soit automatique. C'est à dire que le concepteur reçoive un message final "Go/no Go". Il ne doit pas vérifier le fonctionnement manuellement.

Nous distinquons trois étapes de simulation, soit:

- Simulation VHDL
- Simulation fonctionnelle
- Simulation temporelle

La simulation VHDL est rapidement réalisée car elle ne nécessite pas de synthétise de la description. Elle sera toujours exécutée. Cette simulation est très utile si le fonctionnement de la description VHDL n'est pas modifié par l'étape de synthèse. L'objectif principal de ce manuel est de donner au lecteur la maîtrise de ces description en vue de la synthèse automatique.

Finalement, il sera nécessaire d'exécuter une simulation après synthèse. Dans le cas de projet moyen, la simulation fonctionnelle ne sera pas réalisée. Nous ferons uniquement la simulation temporelle qui utilise le résultat après placement routage. Cela nous permet de valider que toute la chaine s'est correctement déroulée.

Vous trouverez plus d'informations et d'explications dans le chapitre "Simulation et bancs de test", page 129.

1-3 Qu'est ce que le VHDL

Le VHDL est un langage de description de matériel qui est utilisé pour la spécification (description du fonctionnement), la simulation et la preuve formelle d'équivalence de circuits. Ensuite il a aussi été utilisé pour la synthèse automatique. L'abréviation VHDL signifie VHSIC (*Very High Speed Integrated Circuit*) Hardware Description Language (langage de description de matériel pour circuits à très haute vitesse d'intégration).

1-3.1 Historique

Au début des années 80, le département de la défense américaine (DOD) désire standardiser un langage de description et de documentation des systèmes matériels ainsi qu'un langage logiciel afin d'avoir une indépendance vis-à-vis de leurs fournisseurs. C'est pourquoi, le DOD a décidé de définir un langage de spécification. Il a ainsi mandaté des sociétés pour établir un langage. Parmi les langages proposés, le DOD a retenu le langage VHDL qui fut ensuite normalisé par IEEE. Le langage ADA est très proche, car celui-ci a servit de base pour l'établissement du langage VHDL.

La standardisation du VHDL s'effectuera jusqu'en 1987, époque à laquelle elle sera normalisée par l'IEEE (*Institute of Electical and Electronics Engineers*). Cette première normalisation a comme objectif:

- La spécification par la description de circuits et de systèmes.
- La simulation afin de vérifier la fonctionnalité du système.
- La conception afin de tester une fonctionnalité identique mais décrite avec des solutions d'implémentations de différents niveaux d'abstraction.

En 1993, une nouvelle normalisation par l'IEEE du VHDL a permis d'étendre le domaine d'utilisation du VHDL vers:

- La synthèse automatique de circuit à partir des descriptions.
- La vérification des contraintes temporelles.
- La preuve formelle d'équivalence de circuits.

Il existe un autre langage de haut niveau pour la description de matériel. Il s'agit du langage VERILOG. La société Cadence est à l'origine de ce langage. Ce langage est aussi normalisé par IEEE.

1-3.2 Pourquoi utiliser le VHDL

Le VHDL est un langage normalisé, cela lui assure une pérennité. Il est indépendant d'un fournisseur d'outils. Il devenu un standard reconnu par

tous les vendeurs d'outils EDA. Cela permet aux industriels d'investir sur un outil qui n'est pas qu'une mode éphémère, c'est un produit commercialement inévitable. Techniquement, il est incontournable car c'est un langage puissant, moderne et qui permet une excellente lisibilité, une haute modularité et une meilleure productivité des descriptions. Il permet de mettre en oeuvre les nouvelles méthodes de conception.

Il est à noter toutefois un inconvénient qui est la complexité du langage. En effet, ce langage s'adresse à des concepteurs de systèmes électroniques, qui n'ont pas forcément de grandes connaissances en langages de programmation.

D'autres fausses idées circulent sur le langage VHDL. Celui-ci n'assure pas la qualité du résultat, la portabilité et la synthèse des descriptions. Une méthodologie est indispensable pour combler ces lacunes.

1-3.3 Ensemble synthétisable du VHDL

L'ensemble du VHDL n'est pas utilisable pour la synthèse automatique.

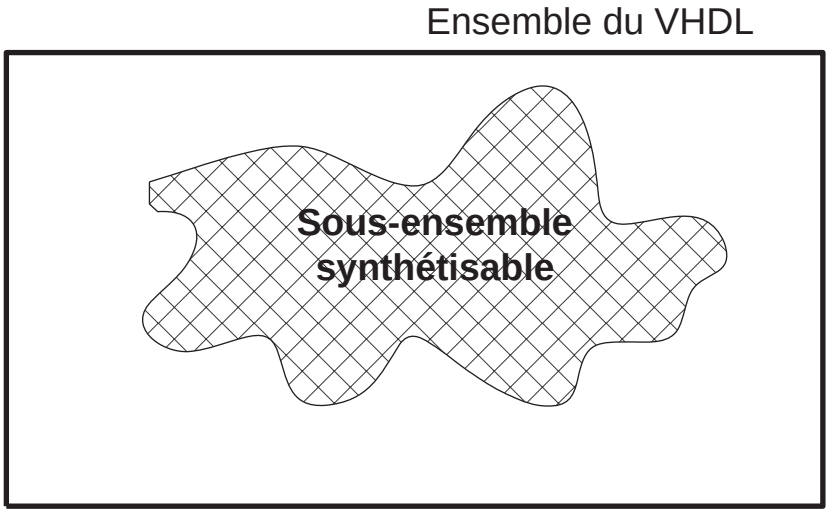


Figure 1- 3 : sous-ensemble synthétisable

Certaines instructions du VHDL sont clairement non synthétisables. Il y a, par exemple, l'instruction "wait for X ns" qui modélise un temps. Mais bien souvent, c'est la manière d'utiliser une instruction qui rend la description non synthétisable.

La connaissance seule des instructions et de leur syntaxe ne suffit pas. Il faut connaître leur utilisation dans le cadre de la synthèse. A l'inverse, la connaissance de l'ensemble du langage n'est pas nécessaire pour écrire des descriptions synthétisables.

En 1999, IEEE a édité la norme 1076.6 qui définit le sous-ensemble synthétisable. Cette norme définit l'ensemble des syntaxes autorisées en synthèse.

Nous pouvons affirmer que depuis l'année 2000 l'ensemble synthétisable du langage VHDL est de mieux en mieux défini. La majorité des outils de synthèse sont compatibles avec cette norme. Un véritable standard est donc établi pour la synthèse automatique avec le langage VHDL.

Chapitre 2

Les concepts du langage VHDL

Le langage VHDL à certaines similitudes avec les langages modernes de programmation structurée. Cela n'est pas surprenant car il en fait partie. La ressemblance se situe surtout sur la syntaxe. Il est surtout très proche du langage ADA. Mais son utilisation est très différente, l'objectif est de décrire le comportement de circuits digitaux. Cela est principalement vrai dans le cadre des descriptions pour la sysnthèse automatique.

2-1 Les éléments de base

2-1.1 Les commentaires

Les commentaires débutent par deux signes moins et se terminent avec la ligne. Voici un exemple de commentaire:

```
-- Ceci est un commentaire sur deux lignes
-- 2eme ligne de commentaire
```

2-1.2 Syntaxe des instructions

Dans le langage VHDL, les instructions sont terminées par un point virgule. Nous donnons ci-après quelques exemples.

```
--Affectation simple
Y <= (not C and B) or (A and C and not D);
--Affectation conditionnelle
A_Eq_B <= '1' when Val_A = Val_B else '0';
--Déclaration d'entité
entity Nom_Du_Module is
        zone pour d'autres instructions
end Nom_Du_Module;
```

Exemple 2- 1 : Syntaxe d'instructions VHDL terminées par un point virgules

2-1.3 Les mots réservés

Les éditeurs actuels affichent les mots réservés du langage en couleur ou en gras. Nous avons choisi, par convention, de les écrire en minuscule. Leur visualisation en gras ou en couleur nous permettant de facilement les reconnaître (voir annexe “Les mots réservés du VHDL.”, page 170).

2-1.4 Les identificateurs

Les identificateurs doivent respecter les règles suivantes:

- Le premier caractère doit être une lettre
- L'identificateur ne doit pas se terminer par un souligné _.
- Dans un identificateur deux soulignés de suite (__) sont interdits.
- Le langage n'est pas sensible à la casse des identificateurs. Il n'y a pas de différence entre deux identificateurs écrits l'un en majuscule et l'autre en minuscule.

Recommandations:

- N'utiliser dans les identificateurs que des lettres et des chiffres.
- Définir si vous écrivez les identificateurs en majuscules ou en minuscules.

Convention utilisée au sein du REDS et dans ce manuel:

La convention de noms utilisée pour les indentificateurs est la suivante:

- $1^{ère}$ lettre en majuscule, le reste en minuscule
- Chaque mot est séparé par un souligné avec la $1^{ère}$ lettre en majuscule
- Voici quelques exemples
  Bus_Donnee, Etat_Present, Adr_Sel, Val_Don

Afin de simplifier la lecture des descriptions VHDL, nous avons défini une convention pour les noms des signaux. Les tableaux ci-après vous donnent les suffixes et préfixes utilisés:

Objet	suffixe
port d'entrée	_i
port de sortie	_o
port entrée/sortie	_io
signal interne architecture textuelle	_s
signal interne schéma bloc	*aucun*
constante	_c
variable	_v
Spécifique pour un banc de test (test-bench)	
signaux de stimuli	_sti
signaux observés	_obs
signaux de référence	_ref

Tableau 2-1 : Liste des suffixes utilisés au REDS

Polarité	préfixe
signaux actif haut	*aucun*
signaux actif bas	'n' exemple: nReset_i
signaux à double nom	'n' devant second nom exemple: Up_nDn_i

Tableau 2-2 : Liste des préfixes utilisés au REDS

2-1.5 Les types d'objets

Le VHDL est un langage fortement typé. Tout objet manipulé doit être déclaré et avoir un type avant sa première utilisation. Indépendamment de son type, un objet appartient à une classe. Nous pouvons résumer cela en disant que le type définit le format de l'objet et que la classe spécifie le comportement de celui-ci.

Le langage distingue quatre catégories de type:

- Les types scalaires. Cela comprend les types énumérés et numériques.
- Les types composés (tableau et enregistrement). Ceux-ci possèdent des sous-éléments.
- Les types accès, qui sont des pointeurs.
- Le type fichier, qui permet de générer des fichiers.

Nous allons uniquement utiliser, dans le cadre du VHDL en vue de la synthèse, les deux premiers types. Les deux autres types ne peuvent pas être utilisés pour la synthèse.

Les types scalaires sont constitués par une liste de valeurs (types énumérés) ou par un domaine de variation (types numériques).

Types scalaires énumérés:

```
type Bit is ('0','1');
type Std_Logic is ('U','X','0','1','Z','W','L','H','-');
type Boolean is (false, true);
```

ou un type défini par un utilisateur:

```
type Etat is (Init, Start, Run, Load, Count, Stop);
```

Types scalaires numériques:

```
type Integer is range -2_147_483_648 to 2_147_483_647;
```

Le type Integer est pré défini pour correspondre à un nombre signé codé en binaire sur 32 bits.

ou un sous types dérivés:

```
subtype Natural is Integer range 0 to 2_147_483_647;
subtype Positive is Integer range 1 to 2_147_483_647;
```

Types scalaires physiques:

```
type Time is range -2_147_483_648 to 2_147_483_647
units fs;  --la femtoseconde !
ps =  1000 fs;
ns =  1000 ps;
us =  1000 ns;
ms =  1000 us;
sec = 1000 ms;
min = 60 sec;
hr =  60 min;
```

Exemple 2- 2 : Types scalaires

Les types composés sont constitués d'éléments scalaires tous de même type dans le cas de tableaux (array) et de types différents dans le cas d'enregistrements (record). Le type enregistrement sera utilisé uniquement pour les bancs de test.

Le type de tableau que nous utiliserons le plus souvent est le Std_Logic_Vector. Celui-ci est un ensemble de Std_Logic.

```
type Std_Logic_Vector is array (Natural range <>)
                                        of Std_Logic;
type Unsigned is array (Natural range <>) of Std_Logic;
```

Exemple 2- 3 : Types composites

L'utilisation des types (Type), sous-types (Subtype) et les articles (record) est présentée dans le chapitre "Notions avancées du langage VHDL", page 63.

2-1.6 Les classes d'objets

Le VHDL définit quatre classes d'objets. Lors de la déclaration d'un objet, il faudra préciser à quelle classe il appartient. Dans le cadre de ce manuel, nous présenterons uniquement les trois premières classes.

Le langage définit les quatre classes suivantes:

- Les constantes.
- Les variables.
- Les signaux.
- Les fichiers.

Les constantes permettent de paramètrer des valeurs non modifiables lors de l'exécution de la description. Elles permettent d'améliorer la lisibilité de la description. Elles sont un peu utilisées dans les descriptions en vue de la synthèse. Leur principale utilisation sera dans les fichiers de simulation. Les fichiers seront uniquement utilisés dans les bancs de test.

```
constant Periode : time  := 100 ns;
constant Faux    : Std_Logic := '0';
constant Vect_Nul: Std_Logic_Vector(3 downto 0)
                                          := "0000";
```

Exemple 2- 4 : Les constantes

Les classes d'objet signal et variable seront présentées dans les paragraphes suivants. Nous pouvons dire qu'une variable appartient au monde *soft*. Celle-ci n'a pas de réalité physique. Par contre, un signal appartient au monde *hard*. Celui-ci représente une équipotentielle, il aura donc une représentation physique dans le circuit décrit.

2-1.7 Les opérateurs

Le langage VHDL dispose de 6 groupes d'opérateurs. Le langage de base ne définit pas ces opérateurs pour tous les types, mais il est possible de surcharger ceux-ci. Nous donnons la liste des opérateurs pour la norme 2000 du langage VHDL. Les opérateurs qui ont été introduit lors de la norme de 1993 seront repérés par l'indication: VHDL-93.

Dans le langage VHDL, les états logiques d'un signal sont définit par un type scalaire énumé. Il s'agit des types Bit et Bit_Vector. Ceux-ci ne sont jamais utilisés en pratique. Nous utiliserons uniquement dans ce manuel les types Std_Logic et Std_Logic_Vector. Les opérateurs sont aussi définies pour les types Std_Logic et Std_Logic_Vector par une surcharge définie dans le paquetage Std_Logic_1164. Il sera nécessaires d'inclure dans

toutes les description VHDL ce paquetage Std_Logic_1164. Voir “Le type Std_Logic”, page 19.

Les opérateurs logiques sont définies pour les types Bit et Bit_Vector. Ceux-ci sont aussi définis pour les types Std_Logic et Std_Logic_Vector par une surcharge définie dans le paquetage Std_Logic_1164.

Opérateur	Description	Résultat
and	ET logique	même type
or	OU logique	même type
nand	Non-ET Logique	même type
nor	Non-OU logique	même type
xor	OU exclusif logique	même type
xnor	Non-OU exclusif logique (VHDL-93)	même type

Tableau 2-3 : Opérateurs logiques

Les opérateurs relationnels sont définis pour tous les types scalaires et les tableaux de scalaires. Mais les deux opérandes doivent être du même type.

Opérateur	Description	Résultat
=	égalité	type booléen
/=	inégalité	type booléen
<	inférieur	type booléen
<=	inférieur ou égal	type booléen
>	supérieur	type booléen
>=	supérieur ou égal	type booléen

Tableau 2-4 : Opérateurs relationnels

Les opérateurs de décalages et de rotation sont définis pour tous les types scalaires et les tableaux de scalaires. Ces opérateurs ont été rajouté dans la norme VHDL-93.

Opérateur	Description	Résultat
sll	Décalage logique gauche	même type opérande gauche
srl	Décalage logique droite	même type opérande gauche
sla	Décalage arithmétique gauche	même type opérande gauche
sra	Décalage arithmétique droite	même type opérande gauche
rol	rotation gauche	même type opérande gauche
ror	rotation droite	même type opérande gauche

Tableau 2-5 : Opérateurs de décalage

Ces opérateurs demandent deux opérandes. Le premier opérande est un tableau, le second est un entier qui indique le nombre de décalage ou de rotation. Le type du résultat est le même que l'opérande de gauche. Voir “Opérateur de décalage (rol)”, page 15.

Les opérateurs de décalage ne seront pas utilisés dans ce manuel. Nous préférons utiliser la concaténation qui permet d’explicité le décalage réalisé (voir “Décalage de vecteurs”, page 48). Voici l’exemple pour la rotation à gauche (rol) de 2 places montré ci-dessus pour un vecteur de 8 bits.

```
--utilisation de l’opérateur rol
Vecteur <= Valeur rol 2;

--utilisation de l’opérateur de concaténation &
Vecteur <= Valeur(5 downto 0) & Valeur(7 downto 6);
```

Exemple 2- 5 : Opérateur de décalage (rol)

Les opérateurs arithmétiques sont définis pour les types numériques. L'utilisation de ces opérateurs pour les types tableaux (exemple: Std_Logic_Vector) nécessitera l'utilisation d'une bibliothèque. Il est recommandé d'utiliser uniquement le paquetage normalisé Numeric_Std.

Opérateur	Description	Résultat
+	Addition	même résultat
-	soustraction	même résultat
&	concaténation	même type, plus large

Tableau 2-6 : Opérateurs arithmétiques

La concaténation fourni un vecteur à partir de deux ou plusieurs vecteurs. Cette fonction est très utilisée dans les descriptions VHDL en vue de la synthèse. Vous trouverez des exemples dans le paragraphe “L’opérateur de concaténation &”, page 47.

Les opérateurs de signe sont définis pour les types numériques.

Opérateur	Description	Résultat
+	signe positif	même résultat
-	signe négatif	même résultat

Tableau 2-7 : Opérateurs de signe

Les opérateurs multiplicatifs sont définis pour les types numériques suivants : entier, signés et non signé, flottants.

Opérateur	Descriptif	Résultat
*	multiplication	dépend
/	division	dépend
mod	modulo, sur entiers	entier
rem	reste, sur entiers	entier

Tableau 2-8 : Opérateurs multiplicatifs

Les opérateurs divers s'effectue sur des types différents selon la fonction. Le tableau ci-dessous donne les types utilisables pour chaque opération.

Opérateur	Descriptif	Résultat
abs	valeur absolue, tous les types	même résultat
**	exponentiel	dépend
not	inversion logique, type scalaire et tableau	même résultat

Tableau 2-9 : Opérateurs divers

Le tableau en annexe définit la priorité entre les groupes d'opérateurs. Celle-ci n'est pas modifiée par une surcharge d'un opérateur. A l'intérieur d'un groupe, tous les opérateurs ont la même priorité. Il est donc nécessaire d'utiliser des parenthèses.

2-2 Signal et variable

Le signal est spécifique au langage de description matériel. Il n'a pas de sens dans un langage informatique. Il est nécessaire pour modéliser les informations qui transitent entre les composants matériels. Le signal est connecté en permanence, par l'intermédiaire d'une ou plusieurs fonctions logiques, à d'autres signaux. En numérique, nous utilisons fréquement la notion de flanc d'un signal. Cela implique qu'il faut connaître la valeur actuelle mais aussi la valeur précédente pour détecter le changment d'état. La notion de signal intégre un pilote qui comprend plusieurs valeurs au cours du temps du signal. Voir paragraphe 2-4, page 22

En informatique, la variable n'a qu'une seule valeur à un instant donné. Celle-ci est mise à jour uniquement lorsque l'instruction d'affectation est exécutée. Ensuite il n'y a plus de lien entre la variable et l'instruction d'affectation.

Nous dirons que le signal appartient au monde *hard*, tandis que la variable appartient au monde *soft*.

2-2.1 Brève définition de la variable

La variable est l'image d'un objet auquel on peut affecter, à tout moment, la valeur que l'on désire.

L'affectation d'une variable se fait à l'aide de l'opérateur : =

```
Nom_Variable := expression ;
```

```
architecture Bidon of Exemple is
  signal N : Integer
begin
  process(N)
    variable A, B : Integer;
  begin
    A := N;
    B := A;
  end process;
end Bidon;
```

Exemple 2- 6 : Affectation d'une variable

La valeur de la variable B est stockée dans la variable A. C'est une opération instantanée, par la suite plus aucun lien n'existe entre ces 2 variables.

L'affectation est aussi valable si B est un signal. Dans tous les cas, le type de l'expression de droite doit être identique à celui de la variable.

2-2.2 Brève définition du signal

Le signal, dans le langage VHDL, est l'image d'une entrée ou d'une sortie d'un composant logique. L'affectation de ce dernier établit un lien définitif entre un ou plusieurs autres signaux.

L'affectation d'un signal se fait à l'aide de l'opérateur: <=

```
Nom_Signal <= expression ;
```

```
architecture Logique of Exemple is
  signal A, B, Y : Std_Logic
begin
  Y <= A or B ;
end Logique;
```

Exemple 2- 7 : Affectation d'un signal

L'exemple ci-dessus modélise l'affectation du signal de sortie d'une porte OU logique.

On remarque que l'affectation ci-dessus a une signification tout à fait différente de celle d'une variable, d'où cette distinction de notation.

2-3 Les types Std_Logic et Std_Logic_Vector

Il y a 2 types principalement utilisés pour les signaux. Ce sont les types:

- Bit et Bit_Vector.
- Std_Logic et Std_Logic_Vector.

Les extensions .._Vector indiquent un élément représentant un vecteur composé d'un ensemble de bits. Ce type correspond à un composite de type tableau (array).Voir paragraphe 2-3.4, page 21

2-3.1 Le type Bit et Bit_Vector

Le type bit est le type standard en VHDL pour un signal. Le bit est définit d'une telle façon qu'il peut uniquement prendre comme valeur 2 états logiques:

- '0', état logique forcé à 0.
- '1', état logique forcé à 1.

Le types Bit et ne sont jamais utilisés en pratique. Ils ne permettent pas de représenter tous les états réels d'un signal. Nous devons avoir, en plus des états '0' et '1', la possibilité de spécifier l'état haute impédance, l'état in-

défini (don't care). De plus en simulation, nous avons besoin d'un état non pour indiquer des conflits, un signal non initialisé, ...

2-3.2 Le type Std_Logic

Le type Std_Logic est le type utilisé dans la pratique dans l'industrie. Il comprend tous les états nécessaire pour modiliser le pcmportement des systèmes numériques. Ce type est normalisé par IEEE. Il sera toujours utilisé pour modéliser des signaux dans un circuit. Nous donnons ci-dessous les 9 états de ce type:

Voici la liste des états du type Std_Logic:

- 'U', état non initialisé.
- 'X', état inconnu fort.
- '0', état logique 0 fort.
- '1', état logique 1 fort.
- 'Z', état haute impédance.
- 'W', état inconnu faible.
- 'L', état logique 0 faible.
- 'H', état logique 1 faible.
- '-', état indifférent (don't care).

De par ces différents états possibles le type Std_Logic est utile pour décrire des portes logiques 3 états, des résistances de *pull-up* ou des sorties de circuit à collecteur ouvert, par exemples. L'état '-' (indifférent) permet de spécifier que l'état d'un signal est indiférent. Exemple dans le cas d'une combinaison inutilisée d'une fonction combinatoire. Le synthétiseur à le choix d'utiliser l'état '1' ou '0' afin de simplifier la logique générée. Les états 'U', 'X' et 'W' sont utilisés en simulation. En cas de conflit entre les deux niveaux '0' et '1' pour un signal, il sera possible d'indiquer cette situation par l'état 'X'. Ce type est toujours utilisé pour les descriptions en vue d'une synthèse automatique. Tous les exemples du manuel ainsi que les exercices utiliserons ce type.

Le type Std_Logic est définit par le paquetage normalisé IEEE.Std_Logic_1164. Il sera donc indispensable de déclarer celui-ci au début de son fichier VHDL (pour avoir accès au type Std_logic).

Lors de l'utilisation du type Std_Logic, il est nécessaire d'écrire au début de la description VHDL les lignes suivantes:

```
Library IEEE;
use IEEE.Std_Logic_1164.all;
```

2-3.3 Le type Std_uLogic et Std_Logic

Le paquetage Std_Logic_1164 définit, comme type de base, le type Std_uLogic. Celui-ci est un signal non résolu (unresolved). Il interdit de connecter deux sources (affectations) sur un même signal.

Le paquetage IEEE.Std_Logic_1164 défini le type énuméré Std_uLogic

```
type std_uLogic is (    'U'     -- Uninitialized
                        'X'     -- Forcing Unknown
                        '0'     -- Forcing 0
                        '1'     -- Forcing 1
                        'Z'     High Impedance
                        'W'     -- Weak Unkown
                        'L'     -- Weak 0
                        'H'     -- Weak 1
                        '-'     --Don't care
                        ),
```

et le type vecteur Std_Logic_Vector

```
type Std_uLogic_Vector is array (Natural range <>) of Std_Logic;
```

Le Std_logic, qui est celui utilisé dans ce manuel, est un sous-type du Std_uLogic. Il est associé à une fonction de résolution afin de permettre la connections de deux sources sur un même signal.

```
function resolved (s : Std_uLogic_Vector) return Std_uLogic;
```

Cette fonction utilise une table de résolution pour déterminer l'état d'un signal piloté par deux sources

```
constant resolution_table : stdlogic_table := (
---------------------------------------------------
--| U    X    0    1    Z    W    L    H    -     |   |
---------------------------------------------------
 ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
 ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
 ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
 ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
 ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
 ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
 ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
 ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
 ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
  );
```

Dans l'industrie c'est le type Std_Logic qui est principalement utilisé. C'est celui que nous utilisons dans ce manuel

```
-------------------------------------------------------------------------------------------
-- *** industry standard logic type ***
-------------------------------------------------------------------------------------------
subtype Std_Logic is resolved Std_uLogic;
```

2-3.4 Le sous type vecteur

VHDL offre la possibilité de décrire un ensemble de signaux sous la forme d'un vecteur, il s'agit du type composite (array). Le paquetage Std_Logique_1164 définit celui-ci de la manière suivante pour les deux types Std_uLogic et Std_Logic :

```
type Std_uLogic_Vector is array (Natural range <>) of Std_uLogic;
type Std_Logic_Vector  is array (Natural range <>) of Std_Logic;
```

Nous pouvons donner un exemple de déclaration d'un vecteur de 3 bits.

```
signal Bus_3_Bits : Std_Logic_Vector(2 downto 0);
```

Exemple 2- 8 : Déclaration d'un vecteur 3 bits

Le nombre de bits qui compose le vecteur est spécifié dans la taille du vecteur (2 downto 0). Nous donnerons toujours la taille en descendant (downto). Cela correspond à la représentation normalisée des nombres en binaire. Le bit de poids fort (MSB, Most significant Bit) est toujours à gauche. Inversement, le bit de poids faible (LSB, Least Significant Bit) est placé à droite.

2-3.5 Affectation d'un signal ou d'un vecteur

L'affectation d'un état logique à un signal s'effectue à l'aide des simples guillemets ('état_logique'), alors que l'affectation d'un état logique à un vecteur s'effectue à l'aide des doubles guillemets ("états_logique").

```
Signal_1_Bit  <= '0';
Vecteur_3_Bits <= "001";
```

L'explication est simplement donnée par le type d'élément affecté. Dans le cas d'une affectation d'un bit, il s'agit d'une affectation d'un caractère donc délimité par de simple guillement ('0'). Le type Std_Logic est un type énuméré, soit une liste de caractère. Dans le cas d'une affectation d'un vecteur, il s'agit d'une chaîne de carctère (array de caractère). Dans ce cas, les doubles guillemets sont utilisés ("1000").

Il existe d'autre possibilité d'affecté des vecteurs, voir "Affectation de valeur hexadécimale", page 48 et "Notation d'agrégat et others", page 49.

2-4 Le temps

Comme indiqué précédemment, le signal n'est pas un récipient que l'on peut vider ou remplir à loisir, comme une variable. C'est un objet qui a des liens, il a un passé, un présent et un futur. Nous devons associer à un signal un ensemble de valeurs. La notion de temps est indispensable pour représenter cet ensemble d'informations. Le langage VHDL associe à chaque signal un pilote qui contient toutes ces informations.

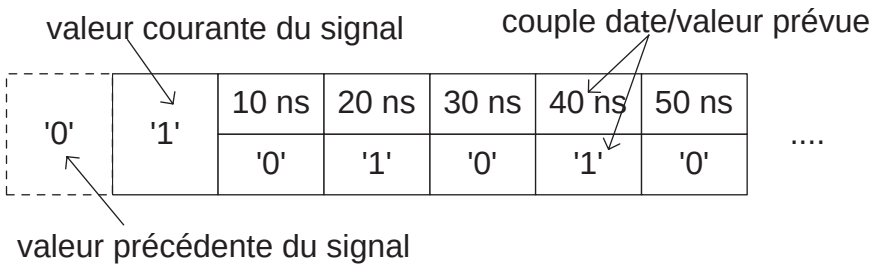


Figure 2- 1 : Représentation du pilote d'un signal

L'exemple ci-dessus correspond au pilote d'un signal d'horloge à 50 MHz.

L'existence du pilote pour un signal va permettre de définir un certain nombre d'attributs et de fonctions pour détecter certaines évolutions du signal. Nous pouvons citer comme exemple: flanc montant. Il s'agit du passage de 0 à 1 d'un signal (Voir “Les attributs pré-définis pour les signaux”, page 54.).

La notion de temps permet également de modéliser les temps de propagation à travers les portes. Il va permettre aussi de décrire des fichiers de simulations (test-bench). Il sera ainsi possible de générer des stimuli pour vérifier le comportement d'un circuits.

2-5 VHDL: un langage typé

Il est important de comprendre que le langage VHDL est typé. Il est interdit d'affecter deux objets de type différent. Nous allons illustrer cela par un exemple.

Le code de l'exemple ci-dessous est **erroné** dans le langage VHDL:

```
architecture Type of exemple
    signal A, B, Y : Std_Logic;
begin
    Y <= (A >= B);  --ERRONE
end;
```

Le résultat de l'opération A >= B est soit vrai, soit faux. Il s'agit d'un résultat de type boolean. Il n'est pas possible d'affecté le résultat au signal Y car celui-ci est de type Std_Logic. Si l'on veut rendre cette partie de description correcte en VHDL, il faut l'écrire comme suit:

```
architecture Type of exemple
    signal A, B, Y : Std_Logic;
begin
    Y <= '1' when (A >= B) else '0';
end;
```

On remarque, cette fois-ci, que les 2 valeurs affectées au signal Y sont bien du même type, soit de type Std_Logic.

2-6 Unité de conception (module VHDL)

L'unité de conception est un ensemble d'éléments VHDL avec lesquels nous allons décrire un système numérique. Celui-ci peut-être constitué d'une simple porte logique jusqu'à un système complexe. Nous parlerons aussi de module VHDL. L'unité de conception est constituée d'une entité (définit l'interface), une ou plusieurs architectures (défini le fonctionnement), des bibliothèques et de la configuration. Les bibliothèques regroupent un ensemble de définition, déclarations, fonctions, etc.. nécessaire à tous les modules. La configuration est optionnelle. Elle ne sera pas utilisée dans ce manuel.

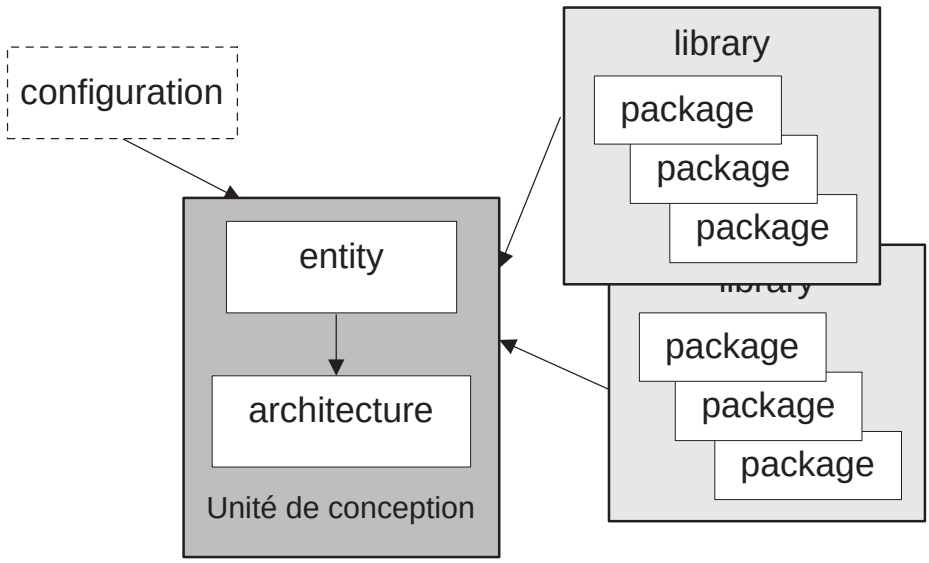


Figure 2- 2 : représentation d'une unité de conception

2-6.1 L'entité

L'entité est une vue externe du module, elle définit toutes les entrées et sorties.

Syntaxe générale de l'entité:

```
entity Nom_Du_Module is
    port ( Nom_Entrée_1  :inType_Du_Signal;
           Nom_Entrée_2  :inType_Du_Signal;
              ...
           Nom_Sortie_1  :outType_Du_Signal;
           Nom_E_S_1     :inoutType_Du_Signal   );
end Nom_Du_Module;
```

Remarque:

Les entrées et les sorties d'un module sont toujours représentées par un port, il en existe 4 différents types, qui sont:

- in : port d'entrée.
- out : port de sortie.
- inout : port bidirectionnel (utilisation pour des bus).
- buffer : port de sortie qui peut être relu (ne pas utiliser)

Un signal de type buffer ne peut pas être connecté directement avec un signal de type out (le VHDL est typé). Il est nécessaire d'utiliser un cast. Cela n'est pas très pratique. Nous déconseillons l'utilisation de ce type ! Nous verrons plus tard comment se passer de ce type en utilisant un signal interne dans l'architecture (Voir "Remplacement du type de port "buffer"", page 50.).

Afin d'être plus concret, nous allons vous présenter, comme exemple, l'entité d'un multiplexeur 8 -> 1 avec 3 entrées de sélection.

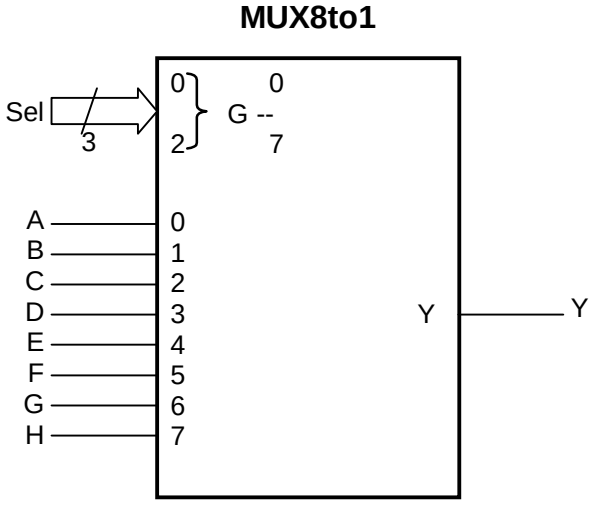


Figure 2- 3 : Schéma bloc du multiplexeur

```
entity MUX8to1 is
  port(Sel : in  Std_Logic_Vector (2 downto 0);
       A,B,C,D,E,F,G,H : in  Std_Logic;
       Y               : out Std_Logic);
end MUX8to1;
```

Exemple 2- 9 : Vue externe du multiplexeur en langage VHDL

2-6.2 L'architecture

L'architecture décrit le comportement du module. Le langage VHDL permet de décrire le comportement de différentes façons. Dans ce manuel nous allons traiter les description synthétisable et les bancs de test. Ceux-ci seront étudié ultérieurement (Voir “Simulation et bancs de test”, page 129.). Les autres types de description ne seront pas présentées ici (exemple: description de spécification)

Nous allons commencer par les descriptions synthétisables. Dans la littérature, ce niveau de description est appelé RTL (Register Transfert Level). Nous distinguerons différentes possibilités de descriptions synthétisables, soit:

- Logique: Equations logiques, description du fonctionnement directement avec des équations logiques.
- TDV: Table de vérité, description du fonctionnement avec une table de vérité (utilisation de l'instruction with ... select).
- Flot_Don: Flot de données, description du fonctionnement très proche du fonctionnement logique (utilisation des instructions concurrentes <=, when ... else ou with ... select)
- Comport: Comportementale, description du fonctionnement en décrivant le comportement (exemple: description avec un process et de l'instruction if ... then ... else)
- M _Etat: Machine d'état, description du fonctionnement à l'aide d'une machine d'état
- Struct: Structurelle, description du module en donnant les interconnexions entre différents sous modules (description hiérarchique avec l'instanciation de composant)

Dans des chapitres ultérieurs, des exemples complets seront donnés pour les 6 types d'architecture décrits ci-dessus.

2-6.3 Syntaxe générale d'une architecture

Voici la syntaxe d'une architecture.

```
architecture Type_D_Architecture of Nom_Du_Module is

--Zone de déclaration

begin

    Instructions_Concurrentes;

    process ( Liste_De_Sensibilité )
    begin
        Instructions_Séquentielles;
    end process;

end Type_D_Architecture;
```

Dans un circuit logique, toutes les portes fonctionnent simultanément. On dit alors que les portes fonctionnent de manière concurrente, c'est à dire que l'ensemble des opérations se déroulent en parallèle. Il est simple de comprendre que toutes les parties d'un circuit fonctionnent simultanément.

Le langage VHDL a été créé dans le but de pouvoir décrire le comportement d'un circuit. Le langage dispose donc d'instructions concurrentes. Cette notion est particulière au langage de description de matériel. Elle n'existe pas dans les langages de programmation conventionnel.

Parfois, il est plus performant de décrire le comportement en utilisant un algorithme en utilisant une instruction tel que le if .. then .. else . C'est la raison de l'existence, dans le langage VHDL, d'une instruction concurrente particulière, l'instruction process. Celle-ci permet de décrire le comportement du circuit avec des instructions séquentielles. A l'intérieur de cette instruction le déroulement des instructions est séquentiel.

Nous allons étudier, dans un premier temps les instructions concurrentes. Ensuite nous aborderons les instructions séquentielles.

Afin d'être plus concret, nous allons donner l'architecture du multiplexeur 8 ->1 avec 3 entrées de sélection déjà présenté au paragraphe précédent "L'entité", page 24.

```
architecture Flot_Don of Mux8_1 is
begin
  with Sel select
    Y <= A  when "000",
         B  when "001",
         C  when "010",
         D  when "011",
         E  when "100",
         F  when "101",
         G  when "110",
         H  when "111",
        'X' when others;
end Flot_Don;
```

Exemple 2- 10 : Multiplexeur 8 to 1

2-6.4 Les paquetages

Le langage VHDL doit son succès grace aux notions de bibliothèque et de paquetage. Il est ainsi possible de fournir un ensemble de fonctionnalités rassemblées dans un paquetage. C'est le cas par exemple des définitions des types standards indispensables en synthèse, soit:

```
Library IEEE;
use IEEE.Std_logic_1164.all;
```

Il sera aussi possible de définir son propre paquetage afin de regrouper des définitions, des sous-programmes, .. pour son projet (voir "Les paquetages", page 84).

Chapitre 3

Les instructions concurrentes

Dans un circuit, toutes les portes fonctionnent simultanément. Le langage VHDL disposent d'instructions concurrentes qui vont nous permettre de décrire le comportement d'un circuit. Une description VHDL est constitué d'un ensemble de processus concurrents. L'ordre dans lequels ces processus apparaissent dans la description est indifférent.

D <= **not** C; C <= A **and** B;	est identique à	C<= A **and** B; D <= **not** C;

Exemple 3- 1 : Processus concurrents

Le langage VHDL disposent de 5 instructions concurrentes, soit:

- l'affectation simple ... <= ....
- l'affectation conditionnelle when ... else
- l'affectation sélectionnée with ... select
- l'instanciation de composant
- l'instruction process. Celle-ci définit une zone pour des instruction séquentielles

Nous allons commencer par décrire les instructions concurrentes simples. Il s'agit de l'affectation simple, l'affectation conditionnelle (when ... else), l'affectation sélectionnée (with ... select) et de l'instanciation de composant. Cette dernière instruction permet de réaliser des descriptions hiérarchiques.

La dernière instruction concurrente est l'instruction process. Cette instruction permet de décrire un comportement avec des instructions séquentielles. Il sera ainsi possible d'utiliser des descriptions algorithmiques. Cette instruction est décrite dans le chapitre suivant "Les instructions séquentielles", page 37.

3-1 L'affectation simple

La syntaxe générique d'une affectation simple est la suivante:

```
Signal_1 <= Signal_2 fonction_logique Signal_3;
```

L'assignation d'un signal de sortie d'une porte logique NAND est un exemple concret d'une affectation simple:

```
architecture Logique of Nand2 is
begin
  Y <= A nand B;
end Logique;
```

Exemple 3- 2 : L'affectation simple

3-2 L'affectation conditionnelle

La syntaxe générique d'une affectation conditionnelle est la suivante:

```
Signal_S <= Signal_1 when Condition_1 else
            Signal_2 when Condition_2 else
      ....                ....
            Signal_X when Condition_X else
            Signal_Y;
```

La description d'un comparateur d'égalité entre 2 signaux, est un exemple concret d'une affectation avec une seule condition.

```
architecture Flot_Don of Comp is
begin
   A_Eq_B <= '1' when Val_A = Val_B else
             '0';
end Flot_Don;
```

Exemple 3- 3 : Affectation conditionnelle avec une seule condition

L'affectation avec condition permet de tester plusieurs conditions qui sont exclusive. La seconde condition ne sera testée uniquement si la première est fausse. La première condition est donc prioritaire sur les suivantes et ainsi de suite. La description d'un détecteur de priorité à 3 entrées nous démontre cette instruction.

```
Priorite <= "11" when Entree3 = '1' else
            "10" when Entree2 = '1' else
            "01" when Entree1 = '1' else
            "00";
```

Exemple 3- 4 : Affectation conditionnelle avec plusieurs conditions

3-3 L'affectation sélectionnée

La syntaxe générique d'une affectation d'un signal sélectionné est la suivante:

```
with Vecteur_De_Commande select

Signal_Sortie <=Valeur_1when Etat_Logique_1,
   Valeur_2when Etat_Logique_2,
      ....              ....
   Valeur_Nwhen others;
```

Remarque:

Le terme **others** affecte au signal l'état logique mentionné, et ceci pour tous les autres états logiques du signal de commande qui n'ont pas été cités précédemment.

La description d'un multiplexeur 8->1 (avec 3 lignes de sélection et 8 entrées) est un exemple concret d'utilisation d'une affectation d'un signal sélectionné. Le symbole du multiplexeur est donné au paragraphe "L'entité", page 24.

```
architecture Flot_Don of Mux8_1 is
begin

  with Sel select
    Y <= A when "000",
         B when "001",
         C when "010",
         D when "011",
         E when "100",
         F when "101",
         G when "110",
         H when "111",
         'X'when others;

end Flot_Don;
```

Exemple 3- 5 : Instruction sélectionnée

Dans l'exemple ci-dessus, le terme others, couvre les cas: "XXX", "ZZZ", "HHH", "UUU", "01U", "-1-", etc. du vecteur Sel, si celui-ci est de type Std_Logic_Vector. Ces états ne sont pas utilisés en synthèse mais existe pour le type Std_Logic (9 états, voir § 3.3).

3-4 L'instanciation de composant

La syntaxe générique d'une instanciation de composant est la suivante:

```
[Label:] Nom_Composant port map(Nom_port => Nom_Signal,
                                ....
                                );
```

Lors de l'utilisation d'un composant, il est nécessaire de déclarer celui-ci et d'indiquer, la paire entité-architecture qui doit être utilisée. Cela s'appelle la configuration. La syntaxe générique de la déclaration d'un composant avec la configuration est la suivante:

```
--Déclaration du composant
component Nom_Composant  [is] --supporté en VHDL93
   port   (Nom_Entree : in   Std_Logic;
                ....
           Nom_Sortie : out Std_Logic );
end component;

for all Nom_Composant use entity work.Nom_Entity(Nom_Arch);
```

Voici la description d'une porte ET par l'utilisation du composant AND pour montrer l'utilisation concrète d'une instanciation de composant.

```
architecture Struct of Porte_And is

  --Déclaration du composant
  component AND2
    port(A, B : in  Std_Logic;
         S    : out Std_Logic );
  end component;
  --configuration
  for all : AND2 use entity work.AND2(Logique);

begin

  --Instanciation du composant
  U1: AND2 port map( A => Entree1,
                     B => Entree2,
                     S => Sortie
                   );

end Struct;
```

Exemple 3- 6 : Instanciation d'un composant

Dans le cas où toutes les connections ne sont pas utilisées il est possible d'utiliser le mot clé open. Voici un exemple d'instanciation d'un comparateur ou une sortie n'est pas utilisée.

```
architecture Struct of Exemple is

  --Déclaration du composant
  component Comp
    port(A, B  : in  Std_Logic_Vector(3 downto 0);
         EG, PP, PG :  out Std_Logic );
  end component;
  --configuration
  for all : Comp use entity work.Comp(Flot_Don);

begin

  --Instanciation du composant
  U1: Comp port map(A =>Nbr_1,
            B =>Nbr_2,
            PP =>open, --non conecte
            EG =>N1_EG_N2,
            PG =>N1_PG_N2
          );

end Struct;
```

Exemple 3- 7 : Instanciation partiel d'un composant

3-5 L'instruction processus

Cette instruction concurrente est très performante. Elle permet de décrire de manière algorithmique le fonctionnement d'un système numérique. Cette instruction sera indispensable pour décrire des éléments mémoires (bascules) en VHDL. Elle est utilisée pour de nombreux types de description. Nous pouvons citer les cas suivants:

- Description algorithmique synthétisable
- Description synthétisable d'éléments mémoires (bascules, registres)
- Description de bancs de tests
- Description de spécification (algorithme)

La syntaxe générique de l'instruction process est la suivante:

```
[Label:] process (Liste_De_Sensibilité)
  --zone de Déclarations
begin

  --Zone pour instructions Séquentielles

end process [Label];
```

Remarques:

- Le processus est traité en un seul bloc. C'est une instruction concurrente. Par contre son contenu va être traité séquentiellement (voir instructions séquentielles).
- Lorsque l'instruction process est utilisée avec une liste de sensibilité, l'instruction wait est interuite à l'intérieur du process.
- Les signaux seront affectés lors de l'endormissement du processus. Cela a lieu soit à la fin (end process) ou lors d'une attente (wait ...). Le temps n'évolue pas lors de l'évaluation d'un processus (exécution des instructions séquentielles).
- A l'intérieur d'un processus, il est possible d'utiliser des variables. Celles-ci seront immédiatement misent à jours.
- Un processus est activé uniquement lorsqu'un des signaux de la liste de sensibilité changent.
- Lorsqu'il y a plusieurs processus dans la même architecture, il est utile de leur donner un nom. Ce qui est possible sous l'appellation label.

L'utilisation de l'instruction processus est difficile. Nous verrons des exemples d'utilisation pour décrire des systèmes combinatoires, voir "Description comportementale", page 94. L'instruction processus est indispensable pour décrire des éléments mémoires. Vous trouverez des exemples

d’utilisation de cette instruction au chapitre “Description de systèmes séquentiels”, page 103.

Chapitre 4

Les instructions séquentielles

Les instructions séquentielles doivent être utilisées uniquement dans une zone de description séquentielle. Nous utiliserons principalement ces instructions à l'intérieur de l'instruction process. Ces instructions peuvent aussi être utilisées dans des procédure et des fonctions.

4-1 L'affectation simple

La syntaxe générique d'une affectation simple est la suivante:

```
Signal_1 <= Signal_2 fonction_logique Signal_3;
```

Remarque:

L'affectation ne modifie pas la valeur actuel du signal. Celle-ci modifie les valeurs futurs. Le temps n'évolue pas lors de l'évaluation d'un process. L'affectation sera donc effective uniquement lors de l'endormissement du process. Voir "L'instruction processus", page 34.

L'instruction when ... else n'est pas utilisable à l'intérieur d'un process. C'est une instruction concurrente. L'instruction séquentielle correspondante est l'instruction conditionnelle (if then else), voir paragraphe suivant.

4-2 L'instruction conditionnelle

Cette instruction est très utile pour décrire à l'aide d'un algorithme le fonctionnement d'un système numérique. La syntaxe générique de l'instruction conditionnelle est la suivante:

```
if Condition_Booléenne_1 then
   --Zone pour instructions séquentielles
elsif Condition_Booléenne_2 then
   --Zone pour instructions séquentielles
elsif Condition_Booléenne_3 then

 ...

else
   --Zone pour instructions séquentielles
end if;
```

Nous reprenons la description du détecteur de priorité (3 entrées et une sortie sur 2 bits) pour démontrer l'utilisation de l'instruction conditionnelle.

```
architecture Comport of Detect_Priorite is
begin

  process (Entree1, Entree2, Entree3)
  begin
    Priorite <= "00";  -- Valeur par defaut
    if (Entree3 = '1') then
      Priorite <= "11";
    elsif (Entree2 = '1') then
      Priorite <= "10";
    elsif (Entree1 = '1') then
      Priorite <= "01";
    --else  pas nécessaire, déjà valeur par défaut
      --Priorite <= "00";
    end if;
  end process;

end Comport;
```

Exemple 4- 1 : Instruction conditionnelle: if ... then ... else

4-3 L'instruction de choix

La syntaxe générique d'une instruction de choix est la suivante:

```
case Expression is
   when Valeur_1 => --Zone pour instructions séquentielles
   when Valeur_2 => --Zone pour instructions séquentielles
     ....
   when others    =>--Zone pour instructions séquentielles
end case;
```

Il est possible de définir plusieurs valeurs de la manière suivante:

```
   when Val_3 | Val_4 | Val_5  =>   --Zone pour instructions séquentielles
```

Si le type utilisé pour l'expression est ordonné (exemple: Integer, Natural, ..), il est possible de définir des plages:

```
   when 0  to 7   =>          --Zone pour instructions séquentielles
ou
   when 31 downto 16  =>  --Zone pour instructions séquentielles
```

Il est important de noté que le type Std_Logic_Vector n'est pas un ensemble de valeurs ordonnées. C'est un type discret. Il est donc impossible d'utiliser les plages to et downto.

Nous allons montrer l'utilisation de l'instruction case avec la description d'un démultiplexeur 1 à 4. Ce circuit dispose d'une entrée de sélection Sel de 2 bits, une entrée à commuter Val_In et de 4 sorties.

```
architecture Comport of DMUX1_4 is
begin

   process (Sel, Val_In)
   begin
      Y0 <= '0'; --Valeur par défaut
      Y1 <= '0'; --Valeur par défaut
      Y2 <= '0'; --Valeur par défaut
      Y3 <= '0'; --Valeur par défaut
      case Sel is
        when "00"   => Y0 <= Val_In;
        when "01"   => Y1 <= Val_In;
        when "10"   => Y2 <= Val_In;
        when "11"   => Y3 <= Val_In;
        when others => Y0 <= 'X';--pour simulation
                       Y1 <= 'X';--pour simulation
                       Y2 <= 'X';--pour simulation
                       Y3 <= 'X';--pour simulation
      end case;
   end process;

end Comport;
```

Exemple 4- 2 : Instruction de choix: case

Remarque:

Le cas others n'est jamais utilisé pour les combinaisons logiques de Sel. Mais le type Std_Logique comprend 9 états ('X', 'U', 'H', etc.). Le vecteur Sel de 2 bits comprend donc 81 combinaisons et pas seulement 4! Il est indispensable de prévoir ces combinaisons dans la description. Lors de la simulation si Sel est différent d'une des 4 combinaisons logiques("00", "01", "10", "11"), les sorties sont affectées avec l'états 'X' pour indiquer qu'il y a un problème dans le fonctionnement du démultiplexeur.

Chapitre 6

Visseries et astuces

Dans ce chapitre, nous allons présenter des éléments facilitant la description en VHDL. Il s'agit soit d'opérateurs, d'attributs ou d'agrégats. Ces éléments forme la boite à outils du langage. L'utilisation de ces outils permet l'écriture de description plus compact et plus portable. Les attribut seront principalement exploités lors du chapitre "Descriptions paramétrables", page 117.

6-1 L'opérateur de concaténation &

Cet opérateur est très utile pour la concaténation de vecteurs de tailles différentes. Nous allons montrer différentes possibilités d'utilisation de cet opérateur.

```
--Soit les signaux suivants:
  signal Val_A, Val_B : Sdt_Logic_Vector(3 downto 0);
  signal Vect8 : Std_Logic_Vector(7 downto 0);
--Exemple de concaténation
  Vect8 <= Val_B & Val_A;
  Vect8 <= "0011" & Val_A;
  Vect8 <= "01" & Val_A & "11";
```

Exemple 6- 1 : Différentes possibilités de concaténation

6-1.1 Décalage de vecteurs

La norme VHDL-93 définit les opérateurs de décalage et rotation. Nous pouvons très facilement réaliser ces fonctions avec l'opérateur de concaténation.

Voici la syntaxe pour un décalage logique à droite:

```
SRL_Vect8 <= "0" & Vect8(7 downto 1);          --décalage logique
```

Nous pouvons aussi montrer la syntaxe d'un décalage arithmétique à droite. Dans ce cas, il est nécessaire d'étendre le bit de signe (MSB). Nous obtenons la syntaxe suivante:

```
SRA_Vect8 <= Vect8(7) & Vect8(7 downto 1);  --décalage arithmetique
```

La rotation est aussi facilement réalisable. La syntaxe d'une rotation à gauche d'une position est la suivante:

```
ROL_Vect8 <= Vect8(6 downto 0) & Vect_8(7);
```

La rotation de plusieurs positions est aussi possible. Voici la syntaxe d'une rotation à droite de 3 position:

```
ROR_Vect8 <= Vect_8(2 downto 0) & Vect8(7 downto 3);
```

6-2 Affectation de valeur hexadécimale

La norme VHDL 93 définit la base hexadécimale pour l'affectation de vecteur. Il est possible dans ce cas d'ajouter un souligner pour séparer les chiffres hexadécimales. Nous allons vous donner la syntaxe de plusieurs affections en hexadécimale:

```
--Soit les signaux suivants:
   signal Vect4  : Sdt_Logic_Vector( 3 downto 0);
   signal Vect8  : Std_Logic_Vector( 7 downto 0);
   signal Vect24 : Std_Logic_Vector(23 downto 0);
   signal Vect32 : Std_Logic_Vector(31 downto 0);

--Nous avons les affectations suivantes:
Vect8   <= x"A5";
Vect4   <= x"D" --double guillemet car c'est un vecteur !
Vect24  <= x"2A_4F5C"
Vect32  <= x"3A79B2E2";
```

Remarque:

L'affectation en hexadécimal ne fonctionne que pour des vecteurs multiples de 4 bits. Si vous avez un vecteur de 30 bits à affecter vous pouvez opérer comme suit:

```
--Soit les signaux suivants:
   signal Vect30a, Vect30b : Std_Logic_Vector(29 downto 0);
   constant Cst32 : Std_Logic_Vector(31 downto 0) := x"3A79B2E2";

--Nous pouvons dés lors affecter les vecteur 30 bits comme suit :

Vect30a  <=  Cst32(29 downto 0);
Vect30b  <= "11" & x"A79B2E2";
```

Remarque:

Il existe aussi la possibiltié de faire des affectation en octal, soit :

```
Vect12    <= o"7426";
```

6-3 Notation d'agrégat et others

La notation par agrégat permet d'indiquer la valeur d'un type composite. Cela est intéressant pour affecter les valeurs d'un tableau (array), et plus spécialement pour les vecteurs. Un agrégat permet d'indiquer la valeur des éléments entre parenthèse et séparé par une virgule.

La syntaxe de l'affectation d'un vecteur avec un agrégat est la suivante:

```
--Soit la déclaration suivante:
   signal Vecteur : Std_Logic_Vector(3 downto 0);

--Nous pouvons dés lors affecter le vecteur comme suit :

Vecteur   <= ('1', '0', '0', '1') ;  --idem que <= "1001"
```

L'agrégat est très utile pour affecter plusieurs bits d'un vecteur à la même valeur avec le mot clé others. Il est dès lors pas nécessaire de connaître la taille de celui-ci. Cette fonction est indispensable pour obtenir des descriptions génériques (taille variable des vecteurs).

La syntaxe de l'affectation d'un vecteur à "00..00" est la suivante:

```
Vecteur <= (others => '0');
```

Voici d'autres exemples d'utilisation de l'agrégat others.

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

--Tout le vecteur affecté avec Entrée
Vect_C <= (others => Entree);
```

Exemple 6- 2 : Utilisation de l'agrégat others

Il est aussi possible grâce à la notion d'agrégat d'affecter une valeur avec une liste de valeur. Nous allons vous donner quelques exemples.

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

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

Exemple 6- 3 : Utilisation de la notion d'agrégat

Nous verrons plus loin que cette notion d'agrégat utilisé avec les attributs permet d'écrire des descriptions paramétrables.

6-4 Remplacement du type de port "buffer"

L'objectif du type buffer est de permettre de pouvoir relire un signal de sortie. Dans le cas du type out, il est interdit de relire l'état de la sortie. Le langage VHDL étant typé, il est interdit ensuite de connecter un signal de type buffer avec un signal de type out. Il est nécessaire d'utiliser un cast entre le type buffer et out. La solution la plus simple est de substituer le type de port buffer par le type out en utilisant un signal interne.

Voici un exemple de description où la sortie est relue avec le type buffer.

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

entity Exemple is
  port( A, B : in     Std_Logic;
        X    : out    Std_Logic;
        Y    : buffer Std_Logic);
end Exemple;
```

```
architecture Logique of Exemple is
begin

  Y <= A nor B;
  X <= Y or B;

end Logique;
```

Exemple 6- 4 : Utilisation d'une sortie avec l'utilisation du type buffer

Nous conseillons dans un tel cas, d'utiliser un type de port out et de créer un signal interne. Celui-ci pourra être relu et le signal de sortie reste avec le type de port out. Nous donnons ci-dessous la modification de l'exemple avec la déclaration d'un signal interne avec le sufixe .._s.

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

entity Exemple is
  port( A, B : in   Std_Logic;
        Y    : out  Std_Logic;
        X    : out  Std_Logic );
end Exemple;

architecture Logique of Exemple is
  signal X_s : Std_Logic;
begin

  Y_s <= A   nor B;
  X   <= Y_s or  B;
  Y   <= Y_s;

end Logique;
```

Exemple 6- 5 : Utilisation d'une sortie avec l'utilisation du type out et d'un signal interne

6-5 Les attributs

Il existe plusieurs catégories d'attributs. Voici la liste des différents types d'attributs définit en VHDL:

- Les attributs pré-définis pour les types
- Les attributs pré-définis pour les tableaux (array)
- Les attributs pré-définis pour les signaux
- Les attributs définis par l'utilisateur

Cette dernière catégorie comprend principalement des attributs définis par les outils. Nous pouvons donner comme exemple, les attributs définit par le synthétiseur. Ceux-ci permettent de définir les numéro des pins dans la description et de passer l'information à l'outils de placement routage. Nous ne présenterons pas cette catégorie d'attributs dans ce manuel. Il ne

sont pas utile pour l'utilisateur. De plus, ils ne sont pas utilisés au niveau du langage VHDL.

Nous ne mentionnerons pas tous les attributs pré-définis. Nous allons uniquement donner les plus utilisés. Nous laisserons le lecteur ce référer à la norme ou à un livre pour voir la liste complète des attributs pré-définis.

6-5.1 Les attributs pré-définis pour les scalaires

Nous allons présenté uniquement un seul attribut sur les type. Les autres attributs de cette catégorie ne seront pas traités dans ce manuel.

La syntaxe de l'attribut image est le suivant:

```
T'image(X)
```

Cet attribut retourne la chaîne de caractères qui correspond à X pour le type T. Cette attribut est très utile pour afficher une valeur dans la fenêtre du simulateur. Nous donnons ci-dessous un exemple d'utilisation avec un signal de type Integer.

```
report "Affiche la valeur du signal Nbr = " & Interger'image(Nbr);
```

Exemple 6- 6 : Affichage d'un nombre, attribut image

6-5.2 Les attributs pré-définis pour les tableaux (array)

Ces attributs sont très utiles pour rendre les descriptions plus générique. Ils permettent de faire référence à la taille d'un vecteur déclarer précédemment. Ils sont très utiles pour manipuler les vecteurs qui sont des tableaux de Std_Logic.

Les attributs pré-définis pour les tableaux sont indispensables pour rendre les descriptions génériques. Ils vont permettre de s'adapter automatiquement à la taille des vecteurs. Ils sont très utilisés en synthèse, en simulation (test bench) et pour la modélisation (spécification).

Voici les principaux attributs pré-définis pour les array:
A est un signal, une variable, une constante de type tableau (array)

- A'left valeur de gauche
- A'right valeur de droite
- A'high valeur supérieur
- A'low valeur inférieur
- A'length longueur (dimension du tableau)
- A'range portée
- A'reverse_range portée inverse

Nous allons montrer le fonctionnement de ces attributs avec un vecteur qui est un tableau de Std_Logic.

```
--Déclaration d'un vecteur
```

```
signal Data : Std_Logic_Vector(7 downto 0);

--Voici le résultat de l'utilisation de chaque attribut :
  Data'left= Data'high= 7
  Data'right= Data'low= 0
  Data'length= 8
  Data'range= 7 downto 0
  Data'reverse_range= 0 to 7
```

Ces attribut ont de nombreuses utilisations. Nous allons donner quelques exemples ci-dessous. Le lecteur trouvera de multiple exemples d'utilisation dans les descriptions fournies avec ce manuel. Vous pouvez aussi vous référer au chapitre “Descriptions paramétrables”, page 117. Ces attributs y sont un outil indispensable.

```
entity Exemple is
  port( ...
        CPT : out Std_Logic_Vector (7 downto 0);
        ...
end Exemple;

architecture Comport of Exemple is
--Declaration d'un signal interne avec la même taille
--qu'un signal de l'entite
signal CPT_s: Unsigned(Compteur'range);
.....
```

Exemple 6- 7 : Utilisation des attributs sur les tableaux pour les déclarations

Un second exemple montre comment parcourir toutes les combinaisons logiques d'un vecteur d'entrée pour tester toutes les lignes d'une table de vérité d'un système combinatoire

```
library IEEE;
  use IEEE.Std_Logic_1164.all;
  use IEEE.Numeric_Std.all; --Fonction conversion
                                              To_Unsigned

entity Exemple_tb
end Exemple_tb;

architecture Test_Bench of Exemple_tb is
  ....
  signal Entrees_sti : Std_Logic_Vector(Entrees'range);
  ...
begin

  ...

  process
  begin
    ....
    --Boucle pour parcourir toutes les combinaisons
    for I in 0 to (2**(Entrees_sti'length)-1) loop

      --Affecte les stimuli aux entrees du systeme
      Entrees_sti <= Std_Logic_Vector(
                  To_Unsigned(I, Entrees_sti'length) );
      ....
     end loop;
  ...
  end process;

end Test_Bench;
```

Exemple 6- 8 : Utilisation attributs sur les tableaux dans un fichier de simulation avec utilisation de l'instruction for ..loop

Ces attributs sont aussi valable pour des types ou sous-types de tableau. Cette possibilité est plus rarement utilisée. Nous expliquons le fonctionnement avec un sous-type.

```
--déclaration d'un sous-type
subtype T_Bus_Adr : Std_Logic_Vector(11 downto 0);

--Voici le résultat de l'utilisation de chaque attribut :
T_Bus_Adr'left      = Data'high  = 11
T_Bus_Adr'right     = Data'low   = 0
T_Bus_Adr'length  = 12
T_Bus_Adr'range   = 11 downto 0
T_Bus_Adr'reverse_range    = 0 to 11
```

6-5.3 Les attributs pré-définis pour les signaux

Ces attributs permettent de détecter des événements particuliers sur les signaux. Nous pourrons détecter des flancs ou tester l'évolution des signaux.

En synthèse, l'attribut de détection d'évènement sera indispensable pour pouvoir décrire le comportement d'une bascule sensible au flanc (flip-flop). Nous pourrons ainsi assurer une description fiable pour tous les synthétiseurs.

Dans le cas des bancs de test (test bench) et des spécifications, les attributs de test des événements sur des signaux seront très utiles. Nous pourrons tester des évolution très précise sur les signaux.

Nous allons donner les attributs les plus utiles. Nous les donnerons les plus utilisés au début de la liste.

Voici les principaux attributs pré-définis pour les signaux:
S est un signal

- S'event — vrai seulement si un événement se produit sur S
- S'active — vrai seulement si une transaction se produit sur S
- S'transaction — signal de type bit qui change d'état à chaque transaction sur S
- S''last_event — retourne le temps écoulé depuis le dernier évènement
- S'last_active — retourne le temps écoulé depuis la dernière transaction
- S'last_Value — retourne la dernière valeur du signal avant le dernier évènement
- S'delayed(T) — signal correspondant à la valeur du signal S au temps Now-T
- S'stable(T) — vrai si le signal est n'a eu d'évènements (stable) durant le temps T
- S'quiet(T) — vrai si le signal est n'a eu de transaction (tranquille) durant le temps T

Il est important de bien comprendre la différence entre un évènement et une transaction pour l'utilisation des ces attributs. Voici une description de ces deux termes:

- évènement : changement de la valeur du signal
- transaction : affectation d'un signal, cela est vrai même si le signal est affecté par la même valeur

La figure 7-1 nous montre trois affectations d'un signal. L'attribut S'transaction sera vrai pour les trois affectations. Par contre l'attribut S'event sera vrai uniquement à l'instant t = 10 ns et t = 40 ns.

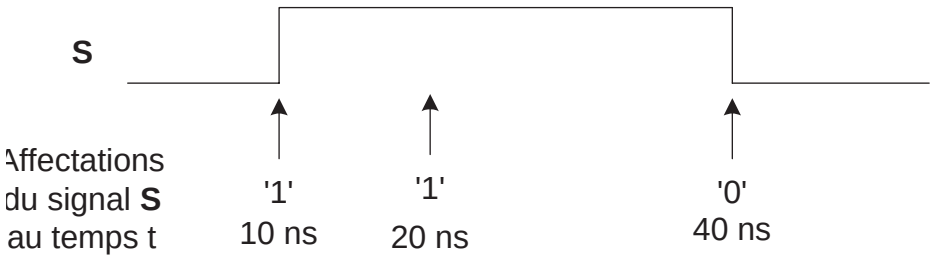


Figure 6- 1 : Différences entre un évènements et une transaction sur un signal

Nous allons vous présenter une utilisation fréquente de l'attribut 'event en synthèse. Il s'agit de la fonction Rising_Edge du paquetage Std_Logic_1164 de IEEE. Il existe une fonction similaire pour la détection du flanc descendant, elle se nomme Falling_Edge.

```
function Rising_Edge(signal S : Std_uLogic)
                                return Boolean is
begin
  return(S'event and (To_X01(S) = '1') and
                  (To_X01(S'last_value) = '0'));
end;
```

Exemple 6- 9 : Utilisation de l'attribut 'event et 'last_value dans la fonction Rising_Edge

Chapitre 7

Paquetage Numeric_Std et opérations arithmétiques

Le paquetage Numeric_Std fait partie de la norme IEEE-1076.3 de 1997. C'est le seul paquetage normalisé pour les opérations arithmétiques. Nous recommandons d'utiliser uniquement celui-ci. C'est le seul qui sera présentaé dans le cadre de ce manuel.

Il existe de nombreux autres paquetages pour réaliser les opérations arithmétique. Nous pouvons cité les paquetages les plus connus, soit:

Std_Logic_Arith, Std_Logic_Unsigned et Std_Logic_Signed

Nous trouvons ces paquetages dans la bibliothèque IEEE. La déclaration de ces paquetages est la suivante:

```
use IEEE.Std_Logic_Arith;
use IEEE.Std_Logic_Unsigned.all;
use IEEE.Std_Logic_Signed;
```

Malgré leur présence dans la bibliothèque IEEE, ces paquetages ne sont pas définit par la norme IEEE. Ils ont été crées par Synopsys. Ils ne font pas partie de la norme VHDL. Nous recommandons au lecteur d'utiliser des outils qui emploie uniquement le paquetage Numeric_Std. Actuellement, la majorité des outils sont compatible avec ce paquetage. Les autres logiciels, pas encore compatibles, seront obligés de suivre la norme à terme.

7-1 Déclaration du paquetage Numeric_Std

Lorsque nous désirons utiliser ce paquetage, nous devons le déclarer dans notre description. La syntaxe de la déclaration de ce paquetage est la suivante:

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

Le paquetage Numeric_Std définit deux nouveaux types pour représenter les nombres. Il s'agit des types suivants:

Unsigned	nombre non signé
Signed	nombre signé en complément à 2

Ces deux types sont basés sur le type Std_Logic. La déclaration de ces deux types est la suivante:

```
type Unsigned is array (Natural range <>) of Std_Logic;
type Signed   is array (Natural range <>) of Std_Logic;
```

Nous devrons utiliser un type ou l'autre en fonction de la représentation que nous souhaitons utiliser pour les nombres

7-2 Fonctions définies dans le paquetage Numeric_Std

Nous n'allons pas mentionner toutes les fonctions définies dans ce paquetage. Nous indiquerons seulement celles qui sont fréquemment utilisées pour la synthèse. Nous renvoyons le lecteur intéressé par l'ensemble du paquetage de la norme IEEE. Vous pouvez aussi consulter le fichier source du paquetage Numeric_Std. Celui-ci se trouve généralement dans un répertoire du programme utilisé (ModelSim, Leonardo, Quartus, ..).

Voici les principales fonctions définies dans le paquetage Numeric_Std:

- Addition et soustraction
  - 2 opérandes de tailles différentes
  - le résultat aura la taille du plus grand des 2 opérandes
    Result :Max(L'Length, R'Length)-1 **downto** 0
  - Combinaisons possibles des types des 2 opérandes (L & R):

(L, R: Unsigned)	return Unsigned
(L: Unsigned; R: Natural)	return Unsigned
(L: Natural; R: Unsigned)	return Unsigned
(L, R: Signed)	return Signed
(L: Integer; R: Signed)	return Signed
(L: Signed; R: Integer)	return Signed

- Comparaison: <, >, <=, >=, = et /=
  L'utilisation de vecteur Unsigned ou Signed est la seule méthode fiable pour décrire des comparaisons en VHDL.
  - 2 opérandes de tailles différentes

- le résultat est de type booléen
- Types des 2 opérandes (L & R):

```
(L, R: Unsigned)              return Boolean
(L: Unsigned; R: Natural)     return Boolean
(L: Natural; R: Unsigned)     return Boolean
(L, R: Signed)                return Boolean
(L, R: Signed)                return Boolean
(L: Integer; R: Signed)       return Boolean
(L: Signed; R: Integer)       return Boolean
```

Remarque:

L'utilisation de la comparaison avec des signaux de type Std_Logic_Vector peut donner des résultats erronés. Dans ce cas, la comparaison est faite élément par élément en commençant par la gauche (chaîne de caractère !). D'où:

"101" > "1000" est vrais !

Cela est évidemment faux au niveau des valeurs binaires. Le résultat sera juste si vous utilisez le type Unsigned.

Voir “Description d'un comparateur”, page 93.

- Conversions: To_Integer, To_Unsigned, To_Signed
  Permet de convertir des nombres entiers (Integer ou Natural) en nombres binaires (Unsigned ou Signed) et inversément. Voici le détail des trois fonctions de conversion:
  - To_Integer (ARG: Unsigned) return Natural
    To_Integer (ARG: Signed) return Integer
  - To_Unsigned (ARG: Natural, SIZE: Natural) return Unsigned
  - To_Signed (ARG: Integer, SIZE: Natural) return Signed

  SIZE : indique le nombre de bits du vecteur

7-3 Vecteurs et nombres

Les vecteurs sont la représentation, définie par le paquetage Sdt_Logic_1164, pour des informations binaires.

Voici la syntaxe pour la déclaration d’un vecteur de type Std_Logic_Vector:

```
signal Vecteur8 : Std_Logic_Vector(7 downto 0);
```

Le paquetage Numeric_Std définit deux types de représentation. Voici la syntaxe pour la déclaration de ces deux types:

```
signal Sgn  : Signed(7 downto 0);  --en complément à 2
signal nSgn : Unsigned(7 downto 0);
```

Le VHDL définit aussi les nombres entiers. Voici la syntaxe pour les déclarations d’un entier positif (Naturel) et d’un entier signé:

```
signal Nbr_P : Natural; --      0     à  (2**31)-1
signal Nbr_E : Integer; --  -2**31  à  (2**31)-1
```

Le langage VHDL étant typé, il est interdit d'affecter ensemble deux signaux de types différents. Il faut donc utiliser des fonctions d'adaptation ou de conversion pour affecter des nombres ayant des différents types de représentation.

Il est à noter que le passage directe entre un Std_Logic_Vector et un entier est impossible. Il est nécessaire de passer par les types Unsigned ou Signed. Nous allons vous présenter dans les prochains paragraphe les différentes possibilités de conversion ou d'adaptation.

Vous trouverez dans les annexes un résumé de ces différentes conversion et adaptation réalisé par la société A.L.S.E. (Advanced Logic Synthesis for Electronics, Paris), voir "Conversion vecteurs et nombres entiers", page 173.

7-3.1 Passage de Std_Logic_Vector à Signed ou Unsigned

Les types Signed et Unsigned sont basés sur le type Std_Logic. Il n'y a donc pas de conversion nécessaire entre un Std_Logic_Vector et un Signed ou un Unsigned. Dans ce cas on parle d'une adaptation de type (cast). Nous allons vous donner ci-dessous quelques exemples en utilisant les signaux déclarés précédement..

```
Vecteur <= Std_Logic_Vector(Sgn);
Vecteur <= Std_Logic_Vector(nSgn);

Sgn  <= Signed(Val);
nSgn <= Unsigned(Val);
```

Exemple 7- 1 : Adaptation entre Std_Logic_Vector et Signed ou Unsigned

7-3.2 Conversion d'un Signed ou Unsigned en un nombre entier

Le passage d'un nombre entier vers une représentation binaire signée ou non signée nécessite une conversion. Il est nécessaire alors d'utiliser les fonctions de conversion définie dans le paquetage Numeric_Std. Nous donnons ci-dessous quelques exemples de conversions.

```
Nbr_E <= To_Integer(Sgn);  --nombre signé
Nbr_P <= To_Integer(nSgn); --nombre non signé

Sgn  <= To_Signed(Nbr_E, 8);
nSgn <= To_Unsigned(Nbr_P, 8);
```

Exemple 7- 2 : Conversion entre un entier et un nombre binaire

Pour la conversion d'un entier vers un vecteur, `unsigned` ou `signed`, il est indispensable d'indiquer la taille de ce dernier.

7-4 Exemple d'additions

Voici quelques exemples d'utilisation du paquetage Numeric_Std pour des additions. Ces exemples sont aussi valable pour la soustraction.

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

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

Exemple 7- 3 : Addition de nombre

Le paquetage `Numeric_Std` définit l'addition entre les type `Unsigned` et `Natural`. Il sera dès lors très pratique d'écrire directement la valeur à ajouter en décimal. Il en de même pour la soustraction avec les types `signed` et `Integer`.

7-5 Exemples de comparaison

Voici quelques exemples de comparaison utilisant le paquetage Numeric_Std et les types Signed et Unsigned.

```
signal Sgn_A, Sgn_B : Signed(3 downto 0);
signal nSgn_A, nSgn_B : Unsigned(3 downto 0);

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

  nSgn_A = "1010"  ou  nSgn_A = 10
  nSgn_A /= nSgn_B
```

Exemple 7- 4 : Comparaison avec les types Unsigned et Signed

Le paquetage `Numeric_Std` définit les opérations de comparaison entre les type `Unsigned` et `Natural` ou `signed` et `Integer`. Il sera dès lors très pratique d'écrire directement une valeur de comparaison en décimal.

L'utilisation du paquetage `Numeric_Std` permet de garantir que la comparaison est toujours juste même dans le cas de vecteurs ayant une taille différente. D'autre part la possibilité de donner directement une valeur en décimal facilite la lisibilité de la descritpion. Nous recommandons fortement l'utilisation de ce paquetage pour toutes les comparaisons.

Chapitre 9

Description de systèmes combinatoires

Ayant connaissance de toutes les instructions du langage VHDL, nous pouvons maintenant aborder la description de systèmes combinatoires. Nous allons présenter les différentes formes de descriptions indiquées au § 3.6.2 qui sont utilisable pour des systèmes combinatoires.

Nous allons donner des exemples complet de descriptions VHDL synthétisable. Vous trouverez le fichier complet: déclaration des bibliothèques, entité et architecture.

9-1 Description logique

Le type de description par portes logiques utilise l'instruction d'affectation concurrente. C'est la forme de description la plus proche du matériel. Cette description utilise les équations logiques.

9-1.1 Description d'une porte logique NOR 2 entrées

Voici le schéma logique de la fonction désirée.

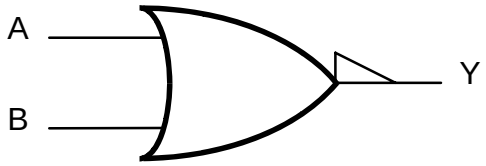


Figure 9- 1 : Schéma d'une porte logique NOR à 2 entrées

La description VHDL correspondante est donnée ci-dessous:

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

entity Non_Ou is
  port(A, B : in  Std_Logic;
       Y    : out Std_Logic);
end Non_Ou;

architecture Logique of Non_Ou is
begin

  Y <= A nor B;

end Logique;
```

Exemple 9- 1 : Description d'une porte logique NOR à 2 entrées

9-1.2 Description d'un schéma logique

Voici un schéma logique d'un système :

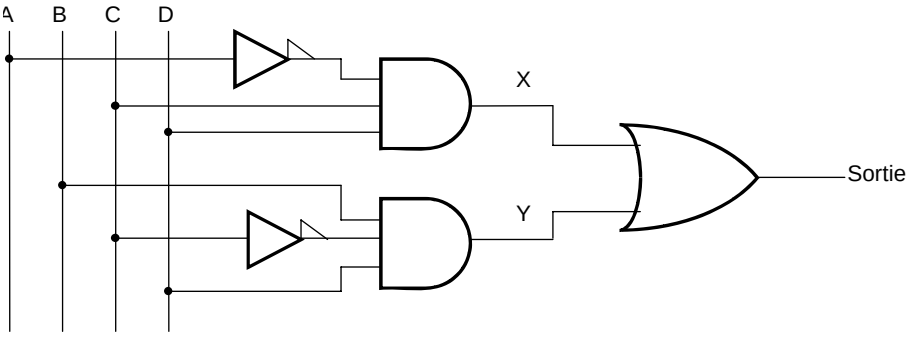


Figure 9- 2 : Schéma logique d'un système

La description VHDL correspondante est donnée ci-dessous :

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

entity Schema is
  port(A, B, C, D : in  Std_Logic;
       Sortie     : out Std_Logic);
end Schema;

architecture Logique of Schema is
  --Signaux internes
  signal X, Y : Std_logic;
begin

  X <= (not A) and C and D;
  Y <= B and (not C) and D;

  Sortie <= X or Y;

end Logique;
```

Exemple 9- 2 : Description d'un schéma logique

9-2 Description par table de vérité (TDV)

Le type de description par table de vérité utilise l'instruction concurrente with ... select. Cette description utilise directement la table de vérité d'un système combinatoire. Cette description est très utile pour des systèmes combinatoire qui ne peuvent pas facilement être décrit par leur comportement.

9-2.1 Description d'un transcodeur: Gray => Binaire pur

Il s'agit de réaliser un transcodeur qui en entrée reçoit une information codée avec le code GRAY sur 3 bits. En sortie, le transcodeur fourni une information en binaire pur sur 3 bits. Le schéma bloc du système est le suivant:

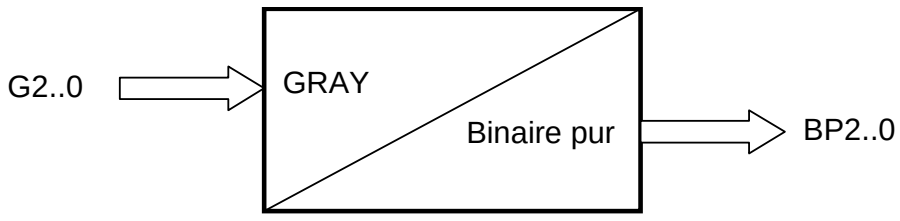


Figure 9- 3 : Schéma bloc d'un transcodeur Gray- binaire pur

La table de vérité correspondante au transcodeur est la suivante:

Gray 3bits			Binaire pur 3 bits		
G2	**G1**	**G0**	**BP2**	**BP1**	**BP0**
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	0	0
1	1	1	1	0	1

Tableau 9-1 : Table de vérité d'un transcodeur Gray - binaire pur

La description VHDL correspondante, utilisant l'instruction with .. select, est donnée ci-dessous:

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

entity Gray_BinPur is
  port(G  : in  Std_Logic_Vector(2 downto 0);
       BP : out Std_Logic_Vector(2 downto 0));
end Gray_BinPur;

architecture TDV of Gray_BinPur is
begin

  with G select
    BP <= "000"  when "000",
          "001"  when "001",
          "011"  when "010",
          "010"  when "011",
          "111"  when "100",
          "110"  when "101",
          "100"  when "110",
          "101"  when "111",
          "XXX"  when others; -- simulation

end TDV;
```

Exemple 9- 3 : Description d'un transcodeur Gray -binair pur

9-3 Description par flot de donnée

Ce type de description est très proche du fonctionnement logique. Le flot des données est clairement explicite dans la description. Elle utilise les instructions concurrentes (affectation <=, when ... else, with ... select). La description n'utilise par les équations logiques.

9-3.1 Description d'un comparateur

Voici le schéma bloc d'un comparateur 4 bits avec sortie égalité:

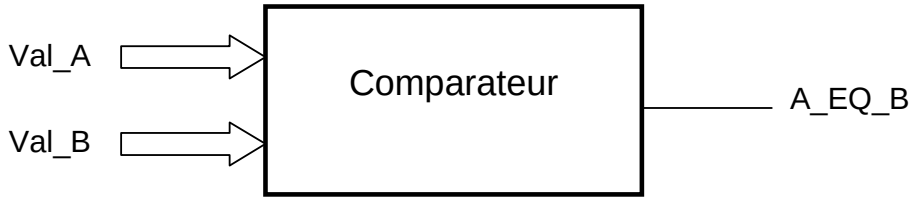


Figure 9- 4 : Schéma bloc d'un comparteur

La description VHDL correspondante, utilisant l'instruction when ... else, est donnée ci-dessous:

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

entity Comp_Eq is
  port(Val_A, Val_B : in  Std_Logic_Vector( 3 downto
0);
       A_Eq_B       : out Std_Logic);
end Comp_Eq;

architecture Flot_Don of Comp_Eq is
begin

  A_Eq_B <= '1' when (Val_A = Val_B) else '0';

end Flot_Don;
```

Exemple 9- 4 : Description d'un comparateur 4 bits

La comparaison de vecteur peut poser certains problèmes. Les opérateurs de comparaison support que les vecteurs soient de taille différente. D'autre part, la comparaison est effectuée bit à bit en commençant par la gauche! Si nous avons deux vecteurs de taille différente, le résultat peut être erroné.

```
signal REG : Std_Logic_Vector(4 downto 0);
signal CNT : Std_Logic_Vector(3 downto 0);
--   REG      CNT    résultat
-- "01111" > "0100"  TRUE     correct
-- "01111" < "1000"  TRUE     ERRONÉ
```

Exemple 9- 5 : Comparaisons de vecteur Std_Logic_Vector!

Nous pouvons garantir une comparaison correcte avec le paquetage `Numeric_Std` (voir “Exemples de comparaison”, page 61). Celui-ci impose que le bit de poids fort est le bit le plus significattif. Le résultat de la comparaison est alors correct pour la représentation choisie (Signed ou Unsigned). Nous donnons ci-dessous une description sûr du comparateur. Le résultat sera toujours même si, cas peu probable, les nombres sont de taille différente.

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

entity Comp_Eq is
  port(Val_A, Val_B : in  Std_Logic_Vector(3 downto 0);
       A_Eq_B       : out Std_Logic);
end Comp_Eq;

architecture Flot_Don of Comp_Eq is
begin
  --Utilise le type Unsigned pour garantir une
  -- comparaison correcte
  A_Eq_B <= '1' when (Unsigned(Val_A)=Unsigned(Val_B))
else
            '0';

end Flot_Don;
```

Exemple 9- 6 : Description du comparateur avec l’utilisation du type Unsigned

9-4 Description comportementale

Ce type de description utilise la notion de langage de haut niveau. L'objectif est de décrire le fonctionnement logique du système sans avoir besoin de déterminer les équations logiques de celui-ci. Ce type de description permet de profiter de la puissance du langage VHDL. Par contre, c'est aussi un des dangers, car parfois la description n'est plus synthétisable. Cela signifie que la description est trop éloignée du matériel et le synthétiseur ne peut plus déterminer le type de logique à utiliser.

Le comportement d’un système peut être défini par un algorithme. Il est possible de reprendre cet algorithme pour décrire en VHDL le comporte-

ment d'un système numérique. Nous montrons cette démarche avec le cas d'un démultiplexeur (voir "Description algorithmique d'un démultiplexeur 1 à N", page 96).

9-4.1 Description d'un démultiplexeur 1 à 4

Voici le schéma bloc d'un démultiplexeur 1 à 4 :

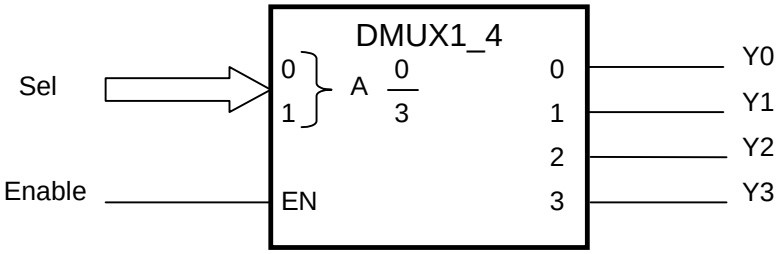


Figure 9- 5 : Schéma bloc d'un démultiplexeur 1 à 4

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

entity DMUX1_4 is
  port ( Sel    : in  Std_Logic_Vector (1 downto 0);
         Enable : in  Std_Logic;
         Y      : out Std_Logic_Vector (3 downto 0) );
end DMUX1_4;

architecture Comport of DMUX1_4 is
begin

  process(Sel ,Enable)
  begin

    Y <= (others => '0'); -- Valeur par défaut

    if Enable = '1' then
      case Sel is
        when "00" => Y(0) <= '1';
        when "01" => Y(1) <= '1';
        when "10" => Y(2) <= '1';
        when "11" => Y(3) <= '1';
        when others => Y <= (others => 'X');
                              -- pour la simulation
      end case;
    end if;
  end process;

end Comport;
```

Exemple 9- 7 : description d'un démultiplexeur 4 to 1

9-4.2 Description algorithmique d'un démultiplexeur 1 à N

Nous pouvons exprimer l'algorithme d'un système sous la forme d'un texte. Voici l'algorithme textuel pour un démultiplexeur 1 à N.

> Activer la sortie correspondant à la sélection,
> les autres sorties sont inactives.

Nous pouvons modifier cet algorithme en inversant les deux lignes, soit:

> Toutes les sorties sont inactives
> Activer la sortie sélectionnée.

Nous pouvons maintenant traduire cet algorithme en VHDL. Nous allons utiliser le concept des instructions séquentiels qui permet d'affecter plusieurs fois le même signal. Nous savons que seul la dernière affectation séquentiel du signal sera effectivement appliquée sur le signal. Voici la traduction de cet algorithme avec des instructions séquentiels:

```
Sorties <= (others => '0'); --toutes les sorties sont désactivées
Sorties(Sel_Integer) <= '1';--sortie sélectionnée activée
```

Nous devons encore convertir la valeur de la sélection en entier et donner la syntaxe exacte en VHDL. Ci-dessous, nous vous donnons la description VHDL du démultiplexeur correspondant à l'algorithme. Nous avons donner l'exemple d'un démultiplexeur 1 à 16. Dans le chapitre "Description de systèmes combinatoires", page 89, nous montrerons comment utiliser cet algorithme pour obtenir une description générique.

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

entity DMUX1_16 is
  port(  Sel    : in  Std_Logic_Vector (3 downto 0);
         Sorties: out Std_Logic_Vector (15 downto 0)
      );
end DMUX1_16;

architecture Comport of DMUX1_16 is
begin

  process(Sel)
  begin
    --toutes les sorties sont desactivees
    Sorties <= (others => '0'); --valeur par defaut
    --active la sortie selectionnee
    Sorties( To_Integer(Unsigned(Sel)) )<= '1';
  end process;

end Comport;
```

Exemple 9- 8 : Description algorithmique d'un démultiplexeur 1 à 16

Nous pouvons vérifier le résultat de l'interprétation de notre description algorithmique par un synthétiseur. Le résultat obtenu, vue RTL, avec le synthétiseur Leonardo (version 2002b) est donné ci-après.

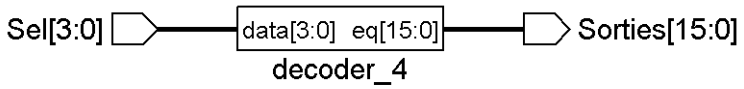


Figure 9- 6 : Vue RTL de la description du démultiplexeur 1 à 16

Nous pouvons aisément introduire une entrée de validation (enable). Nous donnons ci-après la description avec un enable pour activer la sortie sélectionnée.

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

entity DMUX1_16 is
  port(  Sel    : in  Std_Logic_Vector (3 downto 0);
         Enable : in  Std_Logic;
         Sorties: out Std_Logic_Vector (15 downto 0)
       );
end DMUX1_16;

architecture Comport of DMUX1_16 is
begin

  process(Sel, Enable)
  begin
     --toutes les sorties sont desactivees
     Sorties <= (others => '0'); --valeur par defaut
     --active la sortie selectionnee si enable actif
     Sorties( To_Integer(Unsigned(Sel)) )<= Enable;
  end process;

end Comport;
```

Exemple 9- 9 : Description algorithmique d'un démultiplexeur 1 à 16 avec enable

9-4.3 Exemple de description d'un décodeur d'adresse

Voici le schéma bloc d'un décodeur d'adresse :

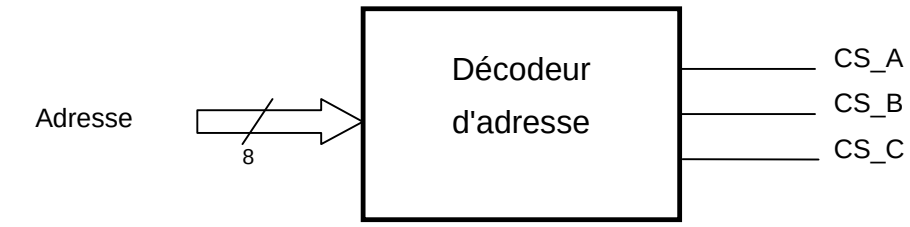


Figure 9- 7 : Schéma bloc d'un décodeur d'adresses

Le tableau ci-dessous nous donne le plan d'adressage de ce décodeur :

Adresses (8 bits)						CS_A	CS_A	CS_A	
en héxa			en décimal						
0	à	2F	0	à	47	1	0	0	
30	à	4F	48	à	79	0	1	0	
50	à	7F	80	à	127	0	0	1	
80	à	FF	128	à	255	0	0	0	Zone libre

Tableau 9-2 : Plan d'aderssage du décodeur

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

entity  Dec_Ad is
   port(Adresse : in Std_Logic_Vector(7 downto 0);
        CS_A, CS_B, CS_C : out Std_Logic );
end Dec_Ad;

architecture Comport of Dec_Ad is
begin

  process(adresse)
  begin

    CS_A <= '0'; --Valeur par defaut
    CS_B <= '0'; --Valeur par defaut
    CS_C <= '0'; --Valeur par defaut
```

```
    case To_Integer(unsigned(Adresse)) is
      when 0   to 47  => CS_A <= '1'; --adresse  0 a 2F
      when 48  to 79  => CS_B <= '1'; --adresse 30 a 4F
      when 80  to 127 => CS_C <= '1'; --adresse 50 a 7F
      when 128 to 255 => null;      --zone libre
      when others  => CS_A <= 'X'; --simulation
                      CS_B <= 'X'; --simulation
                      CS_C <= 'X'; --simulation
    end case;
  end process;
end Comport;
```

Exemple 9- 10 : Description du décodeur d'adresses

9-5 Description structurelle :

Cette architecture est utilisée pour assembler différents modules. Cette architecture fait appelle à l'instruction d'instanciation de composants. Nous pouvons ainsi avoir plusieurs hiérarchies pour décrire un système. Le nombre de niveaux de la hiérarchie n'est pas limité. Au dernier niveau de la hiérarchie, les modules seront décrit en utilisant une des architectures proposées précédemment (Logique, Flot_Don ou Comport).

La syntaxe générique d'une description structurelle est la suivante:

```
entity Hierarchie is
  port(Port_Entrees : in  Type_Port_Entrees;
       Port_Sorties : out Type_Port_Sorties);
end Hierarchie;

architecture Struct of Hierarchie is

    component Bloc1 {is} --accepte en VHDL93
        port(Entrees_Bloc1 : in  Type_Entrees_Bloc1;
             Sorties_Bloc1 : out Type_Sorties_Bloc1);
    end component;
    for all : Bloc1 use entity Work.N_Entity_Bloc1(N_Arch);

    component Bloc2 {is} --accepte en VHDL93
        port(Entrees_Bloc2 : in  Type_Entrees_Bloc2;
             Sorties_Bloc2 : out Type_Sorties_Bloc2);
    end component;
    for all : Bloc2 use entity Work.Entity_Bloc2(Nom_Arch);

    signal Signal_Int1, Signal_Int2 : Std_Logic;

begin

    G0 : Bloc1 port map(Signaux_Bloc1 => Signaux_Int,
                        Signaux_Bloc1 => Signaux_Entity
                        );
```

```
    G1 : Bloc2 port map(Signaux_Bloc2 => Signaux_Int,
                        Signaux_Bloc2 => Signaux_Entity
                        );

end Struct;
```

Remarques:

- Les composants qui sont déclarés dans l'architecture sont les modules VHDL qui sont utilisés lors de l'assemblage du fichier structurel. Lors de cette déclaration, il est indispensable de lister dans l'ordre les signaux d'entrée et sortie du module.
- Les signaux d'interconnexion qui sont déclarés dans l'architecture, sont les signaux internes à l'architecture qui servent à relier les différents composants entre eux.
- Lors de la description de l'assemblage des blocs, il est indispensable de lister les signaux d'interconnexion qui sont connectés en spécifiant la correspondance (.... => ....).

9-5.1 Exemple d'une unité de calcul

Soit une unité de calcul qui permet de soit incrément ou de passer une valeur. La table de vérité ci-dessous nous définit le fonctionnement.

Sel (sélection)	Val_Cal (valeur claculée)
’0’	Valeur (passe)
’1’	Valeur + 1 (incr.)

Tableau 9-3 : Table de vérité d’une unité de calcul

Le schéma bloc de l'unité de calcul est le suivant:

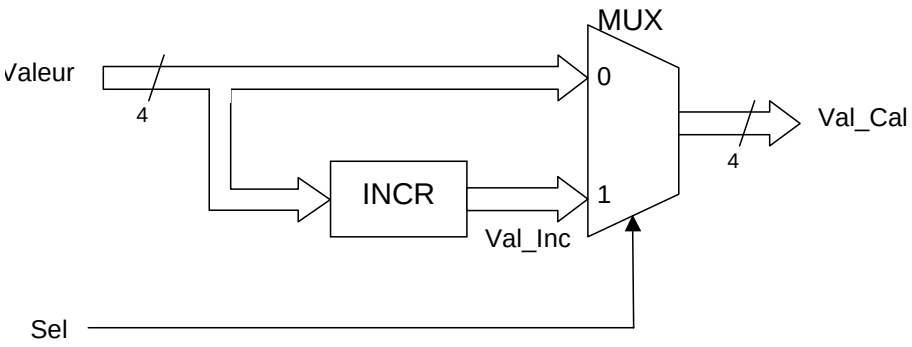


Figure 9- 8 :

Voici la description du circuit d'incrémentation 4 bits:

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

entity  Incr is
  port(Valeur  : in  Std_Logic_Vector(3 downto 0);
       Val_Inc : out Std_Logic_Vector(3 downto 0) );
end Incr;

architecture Comport of Incr is
  signal Val_Int, Result : Unsigned(3 downto 0);
begin

  Val_Int <= Unsigned(Valeur); -- cast

  Result <= Val_Int + 1; --incrementation

  Val_Inc <= std_Logic_Vector(Result); --cast

  --Possible en une affectation sans signaux internes
  --Val_Inc <= std_Logic_Vector( Unsigned(Valeur) + 1
);

end Comport;
```

Exemple 9- 11 : Description d'un incrémenteur

Voici la description du circuit multiplexeur 2 à 1 sur 4 bits :

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity  MUX2_1 is
  port (In0, In1 : in  Std_Logic_Vector(3 downto 0);
        Sel      : in  Std_Logic;
        Y        : out Std_Logic_Vector(3 downto 0) );
end MUX2_1;

architecture Flot_Don of MUX2_1 is
begin

  Y <= In1 when Sel = '1' else
       In0;

end Comport;
```

Exemple 9- 12 : Description d'un multiplexeur 2 à 1 sur 4 bits

La description de l'unité de calcul, nous donne un exemple de description structurelle :

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity Unit_Cal is
  port ( Valeur  : in  Std_Logic_Vector(3 downto 0);
         Sel     : in  Std_Logic;
         Val_Cal : out Std_Logic_Vector(3 downto 0) );
end Unit_Cal;

architecture Struct of Unit_Cal is

  component Incr is --is: accepte en VHDL93
    port ( Valeur  : in  Std_Logic_Vector(3 downto 0);
           Val_Inc : out Std_Logic_Vector(3 downto 0)
);
  end component;
  for all : Incr use entity work.Incr(Comport);

  component  MUX2_1 is -- is: accepte en VHDL93
    port ( In0, In1 : in  Std_Logic_Vector(3 downto 0);
           Sel      : in  Std_Logic;
           Y       : out Std_Logic_Vector(3 downto 0)
);
  end component;
  for all : MUX2_1 use entity work.MUX2_1(Flot_Don);

  signal Val_Inc_Int : Std_logic_Vector(3 downto 0);

begin

  Cal:Incr port map( Valeur  => Valeur,
                     Val_Inc => Val_Inc_Int
                   );

  Mux:MUX2_1 port map( In0 => Valeur,
                       In1 => Val_Inc_Int,
                       Sel => Sel,
                       Y   => Val_Cal
                     );

end Struct;
```

Exemple 9- 13 : Description structurelle de l’unité de calcul

Chapitre 10

Description de systèmes séquentiels

Au chapitre précédent nous avons vu différents types de description avec des systèmes combinatoires. Nous nous sommes ainsi familiarisé avec les instructions du langage VHDL. Maintenant, nous pouvons aborder la description de systèmes séquentiels. Nous serons ensuite capable de décrire tous les types de systèmes numériques. Nous pourrons décrire un système complexe en le décomposant. Nous pouvons utiliser un ou plusieurs niveaux de description structurelle. Dans ce chapitre nous montrerons des exemples de description de différents systèmes séquentiels.

Le langage VHDL ne dispose pas de déclaration explicite d'un signal de type registre. Nous devrons décrire le comportement de l'élément mémoire souhaité. Cette description sera ensuite interprétée par le synthétiseur. L'élément mémoire final obtenu peut-être différent de celui souhaité si la description est ambiguë. Nous dirons dans ce cas que la description non synthétisable. La description en VHDL d'un élément mémoire est très sensible. Nous donnons la description type pour les principaux types d'éléments mémoires. Ces exemples respectent la norme IEEE Std 1076.6 (*Standard for VHDL Register Transfer Level Synthesis*).

Nous allons commencer par la description d'éléments mémoires simples. Ensuite nous donnerons la structure de base pour décrire un système séquentiel synchrone.

10-1 Description d'un 'Flip-Flop D'

Nous allons commencer par la description d'un *Flip-Flop* D (DFF). C'est l'élément mémoire le plus utilisé. Il est à la base de tous les systèmes séquentiels synchrone. Le comportement est celui d'une bascule de type D sensible au flanc. Celle-ci dispose d'une remise à zéro asynchrone (reset).

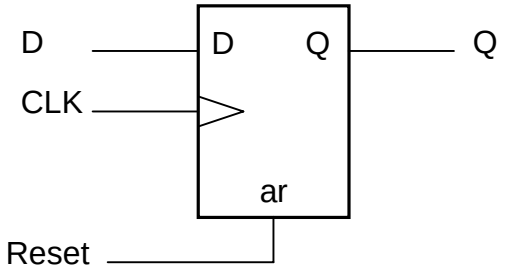


Figure 10- 1 : Symbole du flip-flop D

L'étude du fonctionnement d'un *flip-flop* D nous montre que la sortie ne change que si le signal Reset ou CLK change d'état. Si le signal D change d'état, cela n'influence pas immédiatement le comportement de la bascule. Ce changement sera vu au prochain flanc montant de l'horloge.

La description en VHDL du *flip-flop* D est basé sur cette analyse.

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

entity DFF is
  port(Reset : in  Std_Logic; --reset asynchrone
       CLK   : in  Std_Logic;
       D     : in  Std_Logic;
       Q     : out Std_Logic );
end DFF;

architecture Comport of DFF is
begin

-------------------------------------------------------
-- processus decrivant la memorisation (registre)
-------------------------------------------------------
  Mem: process(CLK, Reset)
  begin
    if (Reset = '1') then
      Q <= '0';
    elsif Rising_Edge(CLK) then
      Q <= D;
   end if;
  end process;
```

```
end Comport;
```

Exemple 10- 1 : Description d'un *flip-flop* D

Remarque:

La description du *flip-flop* D est simple. Il n'y a pas de décodeur d'état futur et de décodeur de sortie. La description comprend uniquement le processus de mémorisation.

10-2 Description d'un latch

Le comportement du *latch* correspond à une bascule de type sensible sur un niveau. Cet élément mémoire n'est pas synchrone. Il est utiliser uniquement pour mémoriser des informations (mémoires, interface).

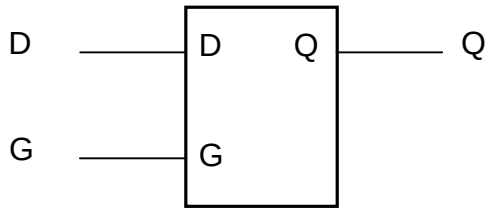


Figure 10- 2 : Symbole d'un latch

L'élément mémoire de type *latch* dispose d'une entrée de commande (G) sensible sur un niveau. Cela signifie que lors l'entrée G est active, la sortie Q suit les variations du signal D. Un changement de cette entrée implique immédiatement un changement sur la sortie Q. Ceci sans que l'entrée de commande G change.

L'étude du fonctionnement d'un *latch* D nous montre que la sortie change si le signal G ou D changent d'état. En conclusion le comportement de la sortie Q dépend des changements de G et de D.

La description en VHDL synthétisable du *latch* D est basée sur cette analyse.

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

entity LatchD is
  port(G : in  Std_Logic; --entrée de commande
       D     : in  Std_Logic;
       Q     : out Std_Logic );
end LatchD;
```

```
architecture Comport of LatchD is
begin

-------------------------------------------------------
-- processus decrivant la memorisation (registre)
-------------------------------------------------------
  Mem: process(G, D)
  begin
    if (G = '1') then
      Q <= D;
   end if;
  end process;

end Comport;
```

Exemple 10- 2 : Description d'un latch D

La description décrit clairement le comportement du *latch* en ayant les signaux G et D dans la liste de sensibilité du process.

10-3 Description d'un système séquentiel synchrone

Nous allons donner la structure de base pour décrire n'importe quel système séquentiel synchrone. Cette structure est rigide afin de garantir une bonne traduction de l'élément mémoire. Il est recommandé de respecter cette structure. Celle-ci garanti une synthèse correcte. Il existe d'autres structures correctes. Notre proposition permet de limiter les risques d'erreurs et d'assurer une bonne lisibilité de la description.

Un système séquentiel se décompose en 3 blocs:

- Le décodeur d'état futur (combinatoire).
- Le registre (séquentiel).
- Le décodeur de sorties (combinatoire).

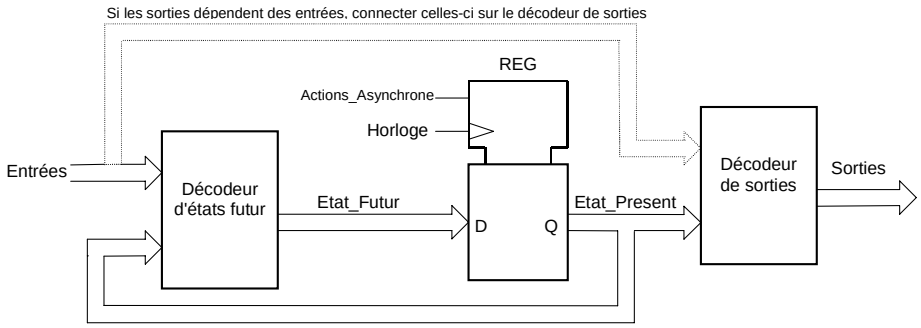


Figure 10- 3 : Schéma bloc d'un système séquentiel

Voici la forme de base de la description en VHDL synthétisable d'un système séquentiel synchrone.

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

entity Nom_Du_Module is
    port ( Nom_Entrée : in  Std_Logic;
           ....
           Nom_Sortie : out Std_Logic);
    end Nom_Du_Module;

architecture Type_De_Description of Nom_De_Module is

    signal Etat_Present, Etat_Futur : Std_Logic_Vector(N-1 downto 0);

begin
--------------------------------------------------------------------------------
-- Description decodeur d'etat futur (combinatoire)
--------------------------------------------------------------------------------
    Etat_Futur <=   .... when Condition_1 else
                    .... when Condition_2 else
                    ...;

--------------------------------------------------------------------------------
-- processus decrivant la memorisation (registre)
--------------------------------------------------------------------------------
    Mem: process(Horloge, Actions_asynchrone )
    begin
        if (Action_asynchrone = '1') then
            Etat_Present <= Etat_Initial;
        elsif Rising_Edge(Horloge) then
            Etat_Present <= Etat_Futur;
        end if;
    end process;

--------------------------------------------------------------------------------
-- Description decodeur de sorties (combinatoire)
--------------------------------------------------------------------------------
    Sortie <=.... when Etat_present = Etat_i else
               .... when Etat_present = Etat_j else
               ....;

end Type_De_Description;
```

Dans la forme de base donnée ci-dessus, les décodeurs sont décrit avec l'instruction concurrente when ... else. Il est possible de décrire ces décodeurs avec des instructions séquentiels (dans un process) dans les cas plus complexes. Le décodeur d'état futur et le décodeur de sorties dépendant des mêmes entrées sont parfois réunis.

10-3.1 Description d'un registre 4 bits:

L'exemple qui suit décrit un registre 4 bits ayant un reset asynchrone, un set synchrone ainsi qu'une entrée de validation (*enable*). Lorsque cette entrée est active, l'état du registre est remis à jour, sinon l'état est maintenu.

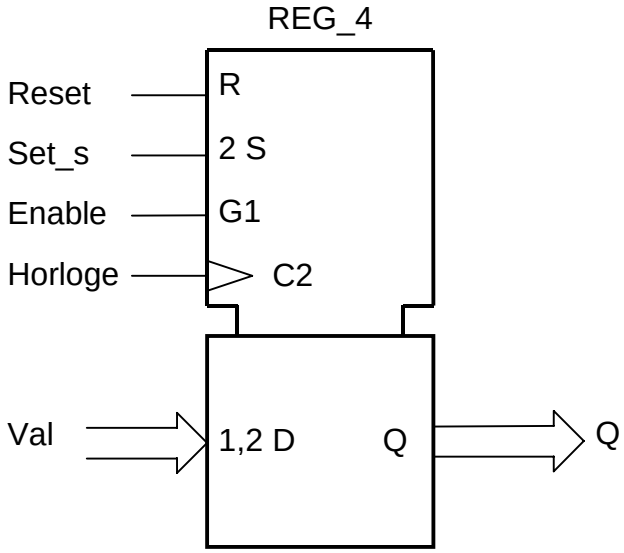


Figure 10- 4 : Schéma CEI du registre 4 bits

Nous donnons ci-dessous la table de fonctionnement de ce registre.

Entrées					**Sorties**
Reset asynchrone	**Set_s** synchrone	**Enable**	**Horloge**	**D**	**Q^+**
H	X	X	X	X	L
L	H	X	↑	X	H
L	L	L	↑	X	Q
L	L	H	↑	Val	Val

Tableau 10-1 : Table de fonctionnement du registre 4 bits

Remarque:

Lorsque la sortie vaut Q, cela veut dire que le registre garde son état présent.

Voici la description synthétisable de ce registre en utilisant la structure proposée précédemment.

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

entity Reg_4 is
  port(Reset   : in  Std_Logic; --reset asynchrone
       Set_s   : in  Std_Logic; --set synchrone
       Enable  : in  Std_Logic;
       Horloge : in  Std_Logic;
       Val     : in  Std_Logic_Vector(3 downto 0);
       Q       : out Std_Logic_Vector(3 downto 0) );
end Reg_4;

architecture Comport of Reg_4 is
  signal Q_Present, Q_Futur : Std_Logic_Vector(3 downto
0);
begin
-------------------------------------------------------
-- Description decodeur d'etat futur
-------------------------------------------------------
  Q_Futur <= "1111" when Set_s = '1'  else
             Val    when Enable = '1' else
             Q_Present;

-------------------------------------------------------
-- processus decrivant la memorisation (registre)
-------------------------------------------------------
  Mem: process(Horloge, Reset)
  begin
    if (Reset = '1') then
      Q_Present <= (others => '0');
    elsif Rising_Edge(Horloge) then
      Q_Present <= Q_Futur;
    end if;
  end process;

-------------------------------------------------------
-- Description decodeur d'etat futur et de sorties
-------------------------------------------------------
  Q <= Q_Present;

end Comport;
```

Exemple 10- 3 : Description d’un registre 4 bits

10-3.2 Description d'un compteur modulo 10:

L'exemple qui suit décrit un compteur modulo 10. Celui-ci dispose d’un reset asynchrone, d’une fonction de chargement et de comptage. Ce dernier mode est actif seulement si l’entrée de validation (*enable*) est active. Dans le cas contraire l’état du compteur est maintenu. Une sortie RCO est prévue pour indiqué que le compteur est à la valeur max, soit 9.

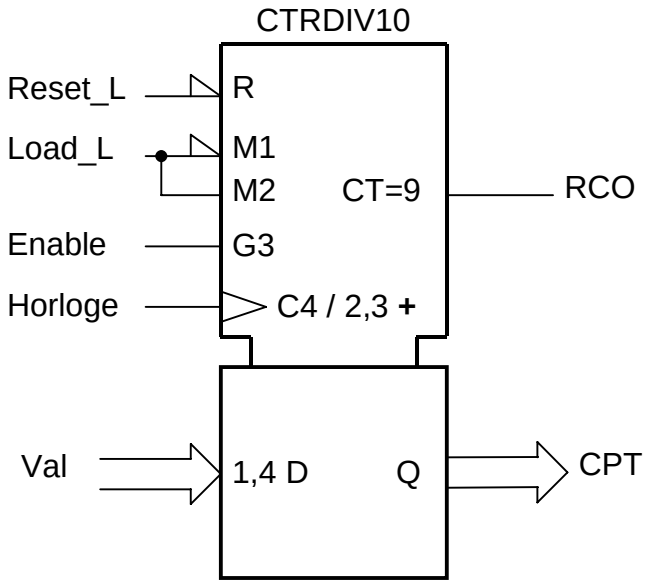


Figure 10- 5 : Symbole CEI d'un compteur modulo 10

Le tableau ci-dessous nous donne la table de fonctionnement du compteur modulo 10. Dans ce tableau toutes les entrées sont indiquées en logique positive.

Entrées					**Sorties**
Reset asynchrone	**Load** synchrone	**Enable**	**Horloge**	**Val**	**CPT$^+$**
H	X	X	X	X	0000
L	H	X	↑	Val	Val
L	L	H	↑	X	CPT + 1
L	L	L	↑	X	CPT

Tableau 10-2 : Table de fonctionnement du compteur modulo 10

La description synthétisable de ce compteur modulo 10 est donnée ci-après. La description utilise la structure donnée précédemment. Nous voyions dans ce cas un exemple de décodeur de sortie. Celui-ci est utilisé pour générer le signal RCO.

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

entity CPTDIV10 is
   port(Horloge, Reset_L : in  Std_Logic;
        Load_L, Enable   : in  Std_Logic;
       Val            : in  Std_Logic_vector(3 downto 0);
       Cpt            : out Std_Logic_Vector(3 downto 0);
         RCO              : out Std_Logic );
end CPTDIV10;

architecture Comport of CPTDIV10 is
  signal Cpt_futur, Cpt_Present : unsigned(3 downto 0);
  signal Reset, Load : Std_Logic;
begin
  --adaptation polarite
  Reset <= not Reset_L;
  Load  <= not Load_L;

  --Decodeur d'etat futur
         --chargement
  Cpt_Futur <= Unsigned(Valeur) when (Load = '1')   else
         --Comptage: si CPT = 9, prochaine valeur=0000
              "0000"           when (Enable = '1')
                              and (Cpt_Present = 9) else
         --          sinon incremente Cpt_present
             Cpt_Present +1   when (Enable = '1') else
         --Maintien
              Cpt_Present;

  Mem: process (Horloge, Reset)
  begin
    if (Reset = '1') then
      Cpt_Present <= "0000";
    elsif Rising_Edge(Horloge) then
      Cpt_Present <= Cpt_Futur;
    end if;
  end process;

  --Calcul du report
  RCO <= '1' when (Cpt_present = 9) else '0';

  --Mise a jour de l'etat du compteur
  Cpt <= Std_Logic_Vector(Cpt_present);

end Comport;
```

Exemple 10- 4 : Description d’un compteur modulo 10

10-4 Les machines d'états

Les machines d'états sont largement utilisées dans les fonctions logiques de contrôle, formant le cœur de nombreux systèmes numériques. Une machine d'états est un système dynamique qui peut se trouver à chaque instant dans un des états parmi un nombre fini d'états possibles. Elle parcourt ces différents états en fonction des valeurs des entrées et modifie éventuellement les valeurs des sorties lors des transitions de l'horloge afin de contrôler l'application.

L'évolution d'une machine d'état dépend de l'état actuel et de l'état des entrées. Dans ce cas il est préférable d'utiliser des instructions séquentiels pour décrire le fonctionnement. La détection de l'état actuel sera réalisé par une instruction case qui décode l'état présent en cours. Puis le test des entrées sera réalisé par une instruction conditionnelle (if .. then .. elsif .. else). Cela correspondra aux différentes transition du graphe des états.

Voici la structure de base du décodeur d'états futur, en VHDL, pour une machine d'états.

```
 --process combinatoire, calcul etat futur
Fut:process (Run, Send, Etat_Present)
begin
    case Etat_Present is
        when Attent =>
            if Run = '1' then
                Etat_Futur <= Charge;
            else
                Etat_Futur <= Attent;
            end if;
        when Charge =>
            Etat_Futur <= Emet;
        when Emet =>
            if Run = '0' and Send = '0' then
                Etat_Futur <= Attent;
            else
                Etat_Futur <= Emet;
            end if;
        when others =>
            Etat_Futur <= Attent;
    end case;
end process;
```

Dans le paragraphe suivant, nous donnons un exemple simple de machine d'états. Il s'agit d'un détecteur de sens de rotation. Cet exemple va nous permettre de nous familiariser avec la description VHDL d'une machine d'état.

10-4.1 Exemple d'une machine d'états

Nous utilisons, comme exemple, le détecteur de sens de rotation. Nous donnons ci-dessous la description du système.

Un disque peut tourner dans les 2 sens de rotation possibles, le sens horaire et le sens anti-horaire. Deux capteurs A et B sont disposés à 90° l'un de l'autre, autour du disque. Le disque est "polarisé" de façon à ce que les capteurs détectent un état logique haut ('1') sur une demi-circonférence du disque, et un état logique bas ('0') sur l'autre demi-circonférence. Il s'agit à l'aide de la séquence de valeurs lues sur les capteurs A et B, de déterminer dans quel sens tourne le disque.

Nous donnons ci-après un dessin schématique du système permettant la détection du sens de rotation.

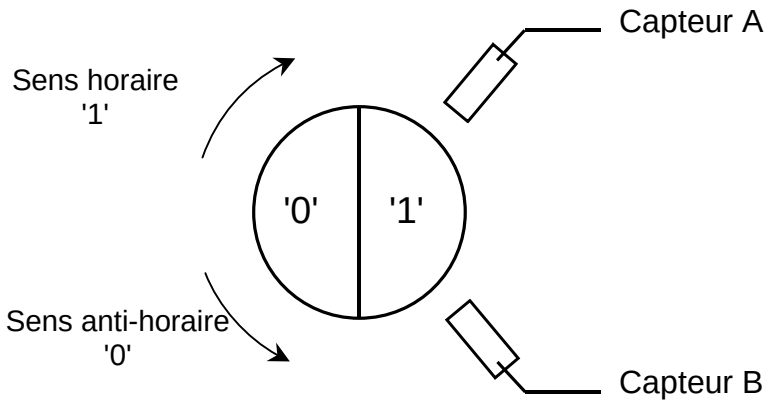


Figure 10- 6 : Schéma d’un détecteur de sens de rotation

Nous pouvons définir le fonctionnement du détecteur à l’aide d’un graphe des états (version de Mealy).

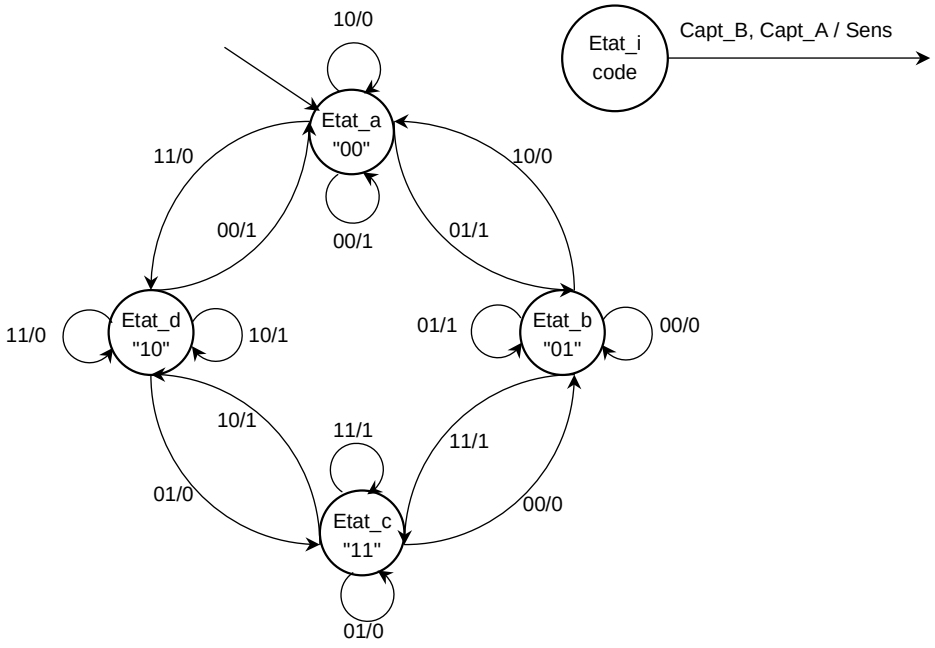


Figure 10- 7 :

Remarque:

Le graphe d'états est directement donné simplifié sans aucun développement ou explication. La conception d'une machine séquentielles ne fait pas partie des objectifs de ce manuel.

Le graphe des états obtenu précédemment est utilisé pour la description en VHDL du système. Le graphe définit quel est l'état futur en fonction de l'état présent et des entrées. Cela correspond au décodeur d'état futur. De même le graphe indique pour chaque état présent la valeur des sorties. Il s'agit dans ce cas du décodeur de sortie.

Dans la description VHDL de la machine d'état, donnée ci-après, nous avons fusionné les deux décodeurs. Nous avons utilisé un seul process pour les deux parties combinatoires. A la fin de la description se trouve la description du registre correspondant aux bits d'états. Nous avons utilisé des constantes pour définir le codage correspondant à chaque état. Cela améliore la lisibilité de la description et permet de modifier le codage de la machine sans réécrire la description.

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

entity Detect_Sens is
  port ( Reset    : in  Std_Logic; --reset asynchrone
         Horloge  : in  Std_Logic;
         Capt_A   : in  Std_Logic;
         Capt_B   : in  Std_Logic;
         Sens     : out Std_Logic );
end Detect_Sens;

architecture M_Etat of Detect_Sens is
  signal Etat_Present : Std_Logic_Vector(1 downto 0);
  signal Etat_Futur   : Std_Logic_Vector(1 downto 0);
  signal Capt_BA : Std_Logic_Vector(1 downto 0);

  constant Etat_a : Std_logic_Vector(1 downto 0) := "00";
  constant Etat_b : Std_logic_Vector(1 downto 0) := "01";
  constant Etat_c : Std_logic_Vector(1 downto 0) := "11";
  constant Etat_d : Std_logic_Vector(1 downto 0) := "10";
begin

  Capt_BA <= Capt_B & Capt_A; --concatenation des en-
trees

-------------------------------------------------------
--Description du decodeur d'etat futur et de sorties
-------------------------------------------------------
  Fut: process(Capt_BA, Etat_Present)
  begin
```

```
case Etat_Present is
  when Etat_a =>
    if (Capt_BA = "00") then
      Etat_Futur <= Etat_a;
      Sens       <= '1';
    elsif (Capt_BA = "01") then
      Etat_Futur <= Etat_b;
      Sens       <= '1';
    elsif (Capt_BA = "10") then
      Etat_Futur <= Etat_a;
      Sens       <= '0';
    else --cas (Capt_BA = "11")
      Etat_Futur <= Etat_d;
      Sens       <= '0';
    end if;
  when Etat_b =>
    if (Capt_BA = "00") then
      Etat_Futur <= Etat_b;
      Sens       <= '0';
    elsif (Capt_BA = "01") then
      Etat_Futur <= Etat_b;
      Sens       <= '1';
    elsif (Capt_BA = "10") then
      Etat_Futur <= Etat_a;
      Sens       <= '0';
    else --cas (Capt_BA = "11")
      Etat_Futur <= Etat_c;
      Sens       <= '1';
    end if;
  when Etat_c =>
    if (Capt_BA = "00") then
      Etat_Futur <= Etat_b;
      Sens       <= '0';
    elsif (Capt_BA = "01") then
      Etat_Futur <= Etat_c;
      Sens       <= '0';
    elsif (Capt_BA = "10") then
      Etat_Futur <= Etat_d;
      Sens       <= '1';
    else --cas (Capt_BA = "11")
      Etat_Futur <= Etat_c;
      Sens       <= '1';
    end if;
  when Etat_d =>
    if (Capt_BA = "00") then
      Etat_Futur <= Etat_a;
      Sens       <= '1';
    elsif (Capt_BA = "01") then
      Etat_Futur <= Etat_c;
      Sens       <= '0';
    elsif (Capt_BA = "10") then
      Etat_Futur <= Etat_d;
      Sens       <= '1';
    else --cas (Capt_BA = "11")
      Etat_Futur <= Etat_d;
      Sens       <= '0';
    end if;
```

```
        when others =>  --pour simulation
          Etat_Futur <= "XX";
          Sens       <= 'X';
      end case;
    end process;

  ------------------------------------------------------
  -- processus decrivant la memorisation (registre)
  ------------------------------------------------------
    Mem: process(Horloge, Reset)
    begin
      if (Reset = '1') then
        Etat_Present <= Etat_a;
      elsif Rising_Edge(Horloge) then
        Etat_Present <= Etat_Futur;
      end if;
    end process;

  end M_Etat;
```

Exemple 10- 5 : Description du discriminateur de sens de rotation

Chapitre 15

Annexes

15-1 Normes IEEE

Voici la liste des normes IEEE pour le langage VHDL. Nous les avons classé selon leur date d'édition.

- 1980 Début du projet financé par le DoD
- 1987 Standard IEEE Std 1076-1987 (VHDL 87)
- 1993 Standard IEEE Std 1164-1993 (Std_Logic_1164)
- 1993 Standard IEEE Std 1076-1993 (VHDL 93)
- 1994 Approbation ANSI (ANSI/IEEE Std 1076-1993)
- 1995 Standard IEEE Std 1076.4 (Vital_Primitive et Vital_Timing)
- 1996 Standard IEEE Std 1076.2 (Mathematical Packages)
- 1997 Standard IEEE Std 1076.3 (Numeric_Bit et Numeric_Std)
- 1999 Standard IEEE Std 1076.6
  (Standard for VHDL Register Transfer Level Synthesis)
- 2000 : Standard IEEE Std 1076-2000 (VHDL 2000)
- 2002 : Standard IEEE Std 1076-2002 (VHDL 2002)

15-2 Les mots réservés du VHDL.

Nous donnons ci-après la liste des mots réservés du langage VHDL. La liste correspond à la norme IEEE Std 1076-1993.

abs
access
after
alias
all
and
architecture
array
assert
attribute

begin
block
body
buffer
bus

case
component
configuration
constant

disconnect
downto

else
elsif
end
entity
exit

file
for
function

generate
generic
group
guarded

if
impure
in
inertial
inout
is

label
library
linkage
loop

map
mod

nand
new
next
nor
not
null

of
on
open
or
others
out

package
port
postponed
procedure
process
pure

range
record
register

reject
rem
report
return
rol
ror

select
severity
signal
shared
sla
sll
sra
srl
subtype

then
to
transport
type

unaffected
units
until
use

variable

wait
when
while
with

xnor
xor

15-3 Les opérateurs du langage VHDL.

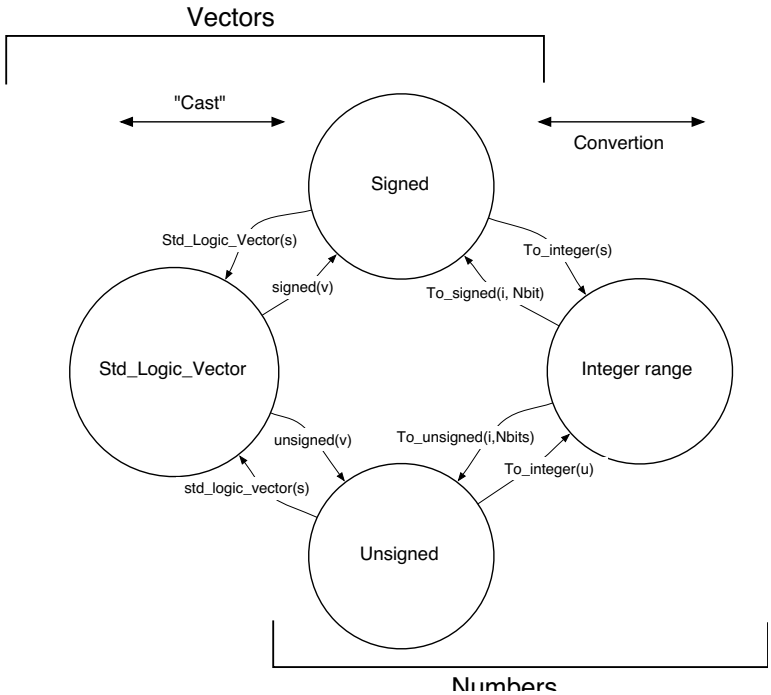
Tous les opérateurs existants en VHDL se trouvent dans la liste ci-dessous dans l'ordre décroissant de leur priorité:

syntaxe VHDL	nom	classe	priorité	exemple
abs	valeur absolue	op. divers	1	abs A
not	non logique	op. divers	1	not A
**	puissance	op. divers	1	A**B
*	multiplication	op. de multiplication	2	A*B
/	division	op. de multiplication	2	A/B
mod	modulo	op. de multiplication	2	A mod B
rem	remainder	op. de multiplication	2	A rem B
+	plus	op. de signe	3	+A
-	moins	op. de signe	3	-A
+	addition	op. d'addition	4	A + B
-	soustraction	op. d'addition	4	A - B
&	concaténation	op. d'addition	4	- D1&D0 - "vh"&"dl"
sll	déc. log. gauche	op. décalages et rotation (VHDL93)	5	Vecteur sll 3
srl	déc. log. droite	op. décalages et rotation (VHDL93)	5	Vecteur srl 1
sla	déc. arith. gauche	op. décalages et rotation (VHDL93)	5	Nombre sla 2
sra	déc. arith. droite	op. décalages et rotation (VHDL93)	5	Nombre sra 4
rol	rotation gauche	op. décalages et rotation (VHDL93)	5	Valeur rol 2
ror	rotation droite	op. décalages et rotation (VHDL93)	5	Valeur ror 4
=	équivalent à	op. relationnels	6	when A=B
/=	différent de	op. relationnels	6	when A/=B
<	plus petit	op. relationnels	6	when A<B
<=	plus petit ou égal	op. relationnels	6	when A<=B
>	plus grand	op. relationnels	6	when A>B

syntaxe VHDL	nom	classe	priorité	exemple
>=	plus grand ou égal	op. relationnels	6	when A>=B
and	et	op. logiques	7	A and B
or	ou	op. logiques	7	A or B
nand	non et	op. logiques	7	A nand B
nor	non ou	op. logiques	7	A nor B
xor	ou exclusif	op. logiques	7	A xor B
xnor	non ou exclusif	op. logiques (VHDL93)	7	A xor B

15-4 Conversion vecteurs et nombres entiers

La société ALSE nous a gracieusement fourni un aide mémoire pour les conversions de vecteurs et de nombres entiers.



Il est important de bien distinguer les cas ou il y a une adaptation de type (*cast*) ou une conversion. Le passage entre le type Std_Logic_Vector et les types Unsigned ou Signed est une simple adaptation (*cast*). Il n'y a pas de conversion de la valeur. Dans les deux cas il s'agit d'un tableau (array) de Std_Logic.

ALSE
Advanced Logic Synthesis for Electronics

La société est active dans les domaines suivants :

- Conception ASIC & FPGA
- Conversion et Portage de Designs existants
- Conception de Modules spécifiques,
- Audit, Conseil, etc…

Contact: Bertrand Cuzeau (direction technique), cuzeau@alse-fr.com

Adresse: ALSE - 166, bd Montparnasse - 75014 PARIS
Tél 01 45 82 64 01 - Fax 01 45 82 67 33
Web : http://www.alse-fr.com

15-5 Bibliographie

[1] Le langage VHDL,
Jacques Weber, Maurice Meaudre,
DUNOD, 2001,
ISBN : 2-10-004755-8

[2] VHDL : méthodologie de design et techniques avancées,
Thierry Schneider,
DUNOD, 2001,
ISBN : 2-10-005371-X

[3] VHDL, Introduction à la synthèse logique,
Phillippe Larcher,
EYROLLES, 1997,
ISBN : 2-212-09584-8

[4] Digital System Design with VHDL,
Mark Zwolinski,
Prentice Hall, 2000

[5] Circuits numériques et synthèse logique, un outil : VHDL,
J. Weber, M. Meaudre,
Masson, 1995,
ISBN : 2-225-84956-0

[6] VHDL du langage à la modélisation,
R. Airiau, J.-M. BergÈ, V. Olive, J. Rouillard,
Presse Polytechniques et Universitaires Romandes, 1998,
ISBN : 2-88074-361-3

[7] VHDL Made Easy!,
D. Pellerin et D. Taylor,
Hardcover, 1996

[8] VHDL for Programmable Logic,
(inclus une version du logiciel Warp de Cypress),
Kevin Skahill de Cypress,
Addison-Wesley, 1996

[9] Introduction au VHDL,
C. Guex,
EIVD, 1997

[10] Version initiale de ce manuel:
Manuel VHDL pour la synthèse automatique,
Yves Sonnay, diplomant de l'EIVD, octobre 1998

15-6 Guides de références

Voici une liste de guides de référence sur le langage VHDL. Nous vous recommandons le guide de Doulos qui est complet avec beaucoups d'exemples et d'explications utiles. Le guide "ACTEL HDL Coding" donne tous les exemples en VHDL et en Vérilog. Il est très pratique pour voir la correspondance entre les deux langages de description.

- The VHDL Golden Reference Guide, version 3.1, 2003
  Disponible chez :Doulos, http://www.doulos.com/
- ACTEL HDL Coding, Style guide, Edition 2003,
  Remis à jour en mai 2004
  Disponible en PDF sur internet : http://www.actel.com/

15-7 Liste des éléments VHDL non traité

Cette annexe indique au lecteur les notions non traitées.

- Acces (allocation dynamique de la mémoire)
- Attribut : déclaration d'un attribut par l'utilisateur
- Alias
- Block
- Configuration
- Exit pour instruction for...loop
- File : le type fichier, les accès fichier et le paquetage TEXTIO (prévu dans une prochaine édition)
- Group
- Guarded
- Inertial (délai pour l'assignement d'un signal)
- Reject (délai pour l'assignement d'un signal)
- type acces
- TEXTIO, paquetage pour les accès fichiers (prévu dans une prochaine édition)
- Transport (délai pour l'assignement d'un signal)