CIRCUITS NUMÉRIQUES ET VHDL

Introduction à VHDL

CIRCUITS NUMÉRIQUES ET VHDL

Introduction à VHDL

Édité

par

J.L. MAYORQUIM

MAYORQUIM

France • Cergy

Remerciements

Ce travail pourra sembler quelque chose peu dense pour un nombre d'étudiants. C'est grâce à Dieu, à la richesse de l'environnement humain de l'École Internationale des Sciences du Traitement de l'Information ainsi que du soutien des étudiants des promotions 2000, 2001, 2002 et 2003 que j'ai pu mener à bien ces travaux.

Je tiens à remercier les étudiants *Fernando LAGRANGE* de la promotion 2000, *Jean Gaël ARZUR* de la promotion 2001, *Eddegdag MAJDA* et *Jerôme MORGES* de la promotion 2002, *Romaric BLATRIX* de la promotion 2003 et Monsieur *Patric LECOINTRE* qui m'ont apporté une aide précieuse pendant la phase de correction de ce polycopié.

J. L. MAYORQUIM

Sommaire

Copyright © by J. L. Mayorquim 11 Fevrier 2002 - 12h46

Copyright © by J. L. Mayorquim 11 Fevrier 2002 - 12h46

Copyright © by J. L. Mayorquim 11 Fevrier 2002 - 12h46

PREFACE

Même si un domaine n'est pas complexe, il y a toujours quelque chose à en apprendre. Lorsqu'il est difficile, il est toujours possible d'en comprendre quand même quelque chose.
Antônio Carlos Lucena

Le traitement de l'information n'est pas seulement déplacer la souris et faire un clic.
Jorge Luiz Mayorquim

La nouvelle version du polycopié sera disponible en septembre de 2007 et qui sera un résumé du livre *VHDL : Analyse et Synthèse de Circuits Numériques*.

Naturellement, les étudiants des Écoles d'Informatique posent la question suivante : Pourquoi devons nous étudier la partie électronique si notre objectif est l'informatique. D'abord, il ne faut pas oublier que derrière un clavier, nous avons un système qui est composé d'un ensemble des composants électroniques. Pour profiter au maximum du potentiel de l'ordinateur, nous avons besoin d'une connaissance la plus large possible et encore de façon formelle. Bientôt, le microprocesseur sera *front-end* d'une machine parallèle programmable au niveau de matériel. Nous avons besoin de définir l'architecture d'un système informatique plus adapté au problème pour ensuite définir les instructions spécifiques qui seront utilisée par les microprocesseurs. Donc, dans un futur proche la connaissance électronique sera plus utilisée par l'informaticien.

J. L. MAYORQUIM

• ***ISICO*** • *Ingénierie de Systèmes Informatiques Complexes* •
LAPI - Laboratoire en Processus Intelligents
EISTI - École Internationale des Sciences du Traitement de l'Information

Copyright © by J. L. Mayorquim 11 Fevrier 2002 - 12h46

1 Introduction à VHDL

1.1 *PRÉSENTATION DU LANGAGE VHDL*

VHDL vient du *VHSIC -Very High Speed Integrated Circuit* i.e., Langage de Description de Matériel. *VHDL* est un langage de description et de modélisation de projet pour décrire (en une forme que l'être humain et les machines peuvent lire et entendre) la fonctionnalité et l'organisation de systèmes numériques, cartes de circuits, et composants.

VHDL a été développé comme un langage pour la modélisation et la simulation logique orienté par des événements de systèmes numériques, et actuellement nous l'utilisons aussi pour la synthèse automatique de circuits. *VHDL* a été développé avec une forme similaire au langage *ADA*. Ce langage a été proposé aussi comme un langage qui contient des structures et des éléments de synthèse qui permettent la programmation d'un quelconque système de matériel, sans limitation d'architecture. *ADA* a une orientation vers les systèmes en temps réel et le matériel en général, par conséquent ce modèle de langage a permis de développer le langage *VHDL*.

VHDL est un langage avec une syntaxe varie et flexible qui permet de décrire le modèle structurel, en flux de données et de comportement du matériel. *VHDL* permet de modéliser avec précision le comportement d'un système numérique et de développer le modèle de simulation.

Un des objectifs du langage *VHDL* est la modélisation i.e., la modélisation est le développement d'un modèle par la simulation d'un circuit ou système précédemment implémenté, donc nous connaissons néanmoins le comportement. Donc, l'objectif de la modélisation est la simulation.

Ce langage a comme objectif aussi la synthèse automatique de circuits. En une procédure de synthèse, nous partons d'une spécification d'entrée avec des niveaux d'abstraction déterminés, et nous arriverons à une implémentation plus détaillée, moins abstraite. Pourtant, la synthèse est une tâche verticale entre niveaux d'abstraction i.e., du niveau le plus haut de la hiérarchie de projet vers le niveau le plus bas.

VHDL est un langage qui a été développé initialement pour être utilisé en modélisation de systèmes numériques. C'est pour cette raison que l'utilisation en synthèse n'est pas immédiate, bien que la sophistication des outils actuels de synthèse soit tel qu'elle permettra l'implémentation du projet en un haut niveau d'abstraction.

La synthèse à partir du *VHDL* constitue aujourd'hui une des principales applications du langage comme une grande demande d'utilisation. Les outils de synthèse basés sur le langage permettent enl'actualité gain important en fait un gain important de productivité dans le projet.

Quelques prérogatives de l'utilisation de *VHDL* pour la description de matériel sont :

⇒ *VHDL* permet de projeter, modéliser, et vérifier un système à partir d'un haut niveau d'abstraction vers le niveau de définition structurale de portes.

⇒ Circuits décrits en utilisant *VHDL*, ils peuvent être utilisés comme outils de synthèse pour créer des implementations de projet à niveau de portes.

⇒ *VHDL* est basé sur le standard (*IEEE STD 1076-1987*), les génies de toutes les entreprises de projet peuvent utiliser ce langage pour minimiser les erreurs de communication et les problèmes de compatibilité.

⇒ *VHDL* permet des projets du type *Top-Down*, i.e., il permet de décrire (*modèle*) le comportement des blocs de haut niveau, à partir de l'analyse (*simulation*), et de raffiner les fonctionnalités de haut niveau exigées avant d'arriver aux niveaux plus bas d'abstraction de l'implémentation du projet.

⇒ La modularité du *VHDL* permet de diviser ou de décomposer un projet de matériel et la description en *VHDL* des unités plus élémentaires.

1.2 *NIVEAUX D'ABSTRACTION DU LANGAGE VHDL*

Il existe différentes des formes pour décrire un circuit. Nous pouvons décrire un circuit en indiquant les différentes composantes qui le forment lui ses interconnexions, de cette façon, nous aurons une spécification d'un circuit et nous saurons son fonctionnement ; cela est la façon habituelle de décrire les circuits i.e., les outils utilisent la capture des schémas et les descriptions du *Netlist* (*liste de connexion*).

La seconde forme consiste à décrire un circuit en indiquant ce qu'il fait ou comment il fonctionne, c'est-à-dire, en décrivant son comportement. Naturellement cette forme de décrire un circuit est meilleure pour un utilisateur puisque celui-ci est intéresse par le fonctionnement du circuit plus que les composantes. Par contre, quand nous sommes loin d'un circuit réel, nous pouvons rencontrer des problèmes quant à l'implémentation du circuit à partir de la description de son comportement.

$\Rightarrow$ *VHDL* propose deux niveaux de descriptions différentes :

$\star$ Structure

VHDL peut être utilisé comme une langue de *Netlist* normal et courant où sont spécifiés sur une liste des composants du système et aussi ses interconnexions.

$\star$ Fonctionnelle

Vous pouvez utiliser aussi *VHDL* pour la description fonctionnelle d'un circuit. C'est ce qui le distingue d'un langage de *Netlist*. Sans nécessité de savoir la structure interne d'un circuit, il est possible de décrire et d'expliquer leurs fonctionnalités. C'est spécialement utile dans la simulation. Parce qu'il permet de simuler un système sans connaître sa structure interne, mais ce type de description devient chaque jour plus important parce que les outils de la synthèse courants autorisent la création automatique de circuits qui commencent par une description de son fonctionnement.

VHDL a deux approches pour la description fonctionnelle d'un circuit. Les équations de transfert qui peuvent être spécifiées parmi différents objets dans *VHDL*. À cette possibilité de description d'un circuit s'ajoute la description flot de données ou nommée description au niveau de transfert entre registres - *RTL*.

Une autre forme existe pour décrire des circuits dans un niveau d'abstraction plus haut. Ce niveau de description est connu comme comportementale. Cette

deuxième possibilité est inclue dans la premier et elle permet au dessinateur de circuits de décrire les fonctionnalités dans un haut niveau d'abstraction.

La différence la plus importante entre un style de description et l'autre est que l'exécution ou interprétation de lignes de codée dans le niveau *RTL* est concourante, c'est-à-dire, les lignes de code, plus que commande et indiquent des rapports ou des lois qui sont complétées, comme elles avaient été exécutées continuellement. Dans la description comportementale, il est possible de décrire des parties avec des directives qui exécutées de la même façon en série que les ordres sont exécutés dans un langage comme le *C* ou *Pascal*.

Ce niveau de la synthèse est toujours plus simple pour synthétiser un circuit décrit en *RTL* qu'un autre décrit dans un niveau plus abstrait. C'est parce que la plupart des structures de la description *RTL* a une correspondance presque directe avec sa mise en oeuvre du matériel correspondant. Dans un niveau plus abstrait, la synthèse automatique du circuit est plus complexe, particulièrement à cause de l'exécution des directives qui ont beaucoup de sens dans l'exécution de programmes en série, par contre c'est quelque chose de plus diffus pour le matériel.

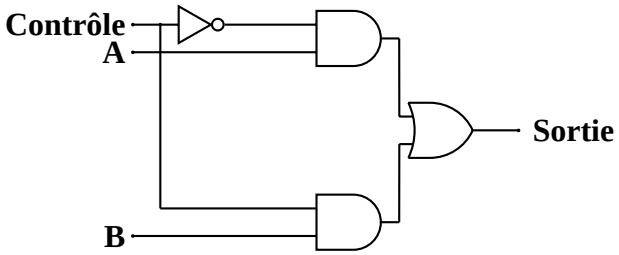
1.3 *DESCRIPTION STRUCTURELLE ET FONCTIONNELLE*

Nous décrirons en *VHDL* un circuit multiplexeur de deux lignes (*A* et *B*) d'un bit, une seule ligne de sortie aussi d'un bit ; le signal contrôle servira pour indiquer que la sortie sera la ligne *A* si le signal de *controle* vaut zéro et la sortie sera la ligne *B* si le signal de contrôle vaut '1', comme illustre la Figure 1.3.1.

Nous développerons la description fonctionnelle du circuit de la Figure 1.3.1. Nous observerons deux types de description fonctionnelle : la comportamentale et *RTL*, et ensuite nous développerons la description structurelle pour analyser les différences. Cette application a comme objectif d'introduire le langage *VHDL* et sa structure.

D'abord, la syntaxe du langage *VHDL* n'est pas sensible aux lettres majuscules et minuscules. Nous essayerons de mettre toujours les mots clé du langage en majuscules et en couleurs pour les distinguer des variables et autres éléments. Ensuite, après chaque bloc de programmation nous décrirons la *Grammaire Formelle* pour illustrer l'aspect formel du langage *VHDL*.

Figure 1.3.1 ***Le multiplexeur.***



Programme 1.3.1 ***L'entité du circuit multiplexeur.***

```
ENTITY mux IS
  PORT (A         : IN  BIT;  -- donnee_d_entree_A
        B         : IN  BIT;  -- donnee_d_entree_B
        Controle  : IN  BIT;  -- donnee_d_entree_de_controle
        Sortie    : OUT BIT); -- sortie
  END mux;
```

Soit une description fonctionnelle ou structurelle, nous devons définir le symbole ou boîte noire pour le circuit. Le symbole dans la langage *VHDL* est nommée d'*entity* du circuit. Donc, nous devons définir les entrées et sorties du circuit comme une partie déclarative du mot clé : *entity*. Cette définition d'*entity* fera partie intégrante de toutes les descriptions en *VHDL*. Dans notre cas, nous devons écrire le symbole du circuit multiplexeur comme :

Le Programme 1.3.1 indique que l'entity du circuit *mux* (qui est le nom que nous avons donnée au circuit) a trois entrées du type *bit*, et une sortie aussi du type *bit*. Le type *bit* indique simplement que chaque ligne peut prendre les valeurs logiques 0 et 1.

Nous pouvons comparer la Grammaire Formelle 1.3.1 avec le Programme 1.3.1. Nous observons que dans notre cas, la partie entity_header et entity_declarative_part ne sont pas utilisées pour cette application. L'entity_statement_part décrit les entrées et sorties du circuit *mux*. Et pour finir, il faut observer que l'entity_simple_name est

Grammaire 1.3.1 ***Déclaration d'entité.***

```
entity_declaration  ::=
ENTITY identifier IS
entity_header
entity_declarative_part
[BEGIN
 entity_statement_part]
END[entity_simple_name] ;
```

égal à l'identifier.

L'entity d'un circuit est unique i.e., un même symbole. Mais, elle peut avoir différentes architectures. Chaque bloc de l'architecture peut être une représentation différente du même circuit. Par exemple, nous pouvons avoir une description structurelle et une autre comportementale, les deux sont des descriptions différentes mais correspondant au même circuit i.e., la même entity ou symbole. Le description comportementale, flot de données et structurelle du circuit multiplexeur sont illustrées par les Programmes 1.3.2, 1.3.3 et 1.3.4, respectivement.

Programme 1.3.3 ***La description flot de données du circuit multiplexeur.***

```
ENTITY mux IS
 PORT (A         : IN  BIT;  -- donnee_d_entree_A
       B         : IN  BIT;  -- donnee_d_entree_B
       Controle  : IN  BIT;  -- donnee_d_entree_de_controle
       Sortie    : OUT BIT); -- sortie_
 END mux;
 ARCHITECTURE flot_de_donnes OF mux IS
 BEGIN
  Sortie <= A WHEN controle ='0' ELSE B;
 END flot_de_donnes;
```

Programme 1.3.4 ***La description structure du circuit multiplexeur.***

```
ENTITY mux IS
 PORT (A         : IN  BIT;  -- donnee_d_entree_A
       B         : IN  BIT;  -- donnee_d_entree_B
       Controle  : IN  BIT;  -- donnee_d_entree_de_controle
       Sortie    : OUT BIT); -- sortie
END mux;
ARCHITECTURE structure OF mux IS
COMPONENT and_2
 PORT(entree_1,entree_2 : IN BIT; S1 : OUT BIT);
END COMPONENT ;
COMPONENT OR_2
 PORT(entree_1,entree_2 : IN BIT; S1 : OUT BIT);
END COMPONENT ;
COMPONENT inv
 PORT(entree_1: IN BIT; S1 : OUT BIT);
END COMPONENT ;
BEGIN
  SIGNAL AX, BX, N_controle : BIT;
  M01: inv   PORT MAP (Controle, N_controle);
  M02: and_2 PORT MAP (A,        N_controle,  AX);
  M03: and_2 PORT MAP (B,        Controle,    BX);
  M04: OR_2  PORT MAP (AX,       BX,          Sortie);
END structure;
```

Programme 1.3.2 ***La description comportamentale du circuit multiplexeur.***

```
 ENTITY mux IS
 PORT (A         : IN  BIT;  -- donnee_d_entree_A
       B         : IN  BIT;  -- donnee_d_entree_B
       Controle  : IN  BIT;  -- donnee_d_entree_de_controle
       Sortie    : OUT BIT); -- sortie
 END mux;
 ARCHITECTURE comoportamentale OF mux IS
 BEGIN
  PROCESS(A,B,controle)
   BEGIN
     IF (controle='0') THEN
       Sortie <= A;
     ELSE
       Sortie <= B;
     END IF;
   END PROCESS;
END comportamentale;
```

Remarque 1.3.1

★ a. *Le niveau structurel ne fait pas intervenir le temps, ce niveau ne peut pas décrire un élément terminal dans la hiérarchie ;*

★ b. *La description dite en flot de données, indique les données sortantes en fonction des données entrantes ;*

★ c. *La description comportementale ne se soucie pas de la structure, le temps peut y intervenir.*

La description s'écrit comme un algorithme, elle peut se révéler très utile pour décrire les systèmes à haut niveau d'abstraction. Dans une réalisation hiérarchisée, nous pourrons trouver un mélange harmonieux de ces trois descriptions. *VHDL* est un langage organisé en bibliothèques, qu'il gère lui-même. Lorsqu'une description est compilée avec succès et est considérée comme correcte par le concepteur, elle est placée dans la bibliothèque de travail, les autres bibliothèques auxquelles la description pourrait faire référence seront considérées comme des bibliothèques de ressource. Une description *VHDL* se décompose en deux parties distinctes: une vue externe qui définit les connexions avec l'univers extérieur et une vue interne qui décrit sa réalisation sous forme de modèles plus simples pour faire l'interconnexions ou plus directement sous forme d'algorithme. Une telle description s'appelle une *entité*, sa vue externe est la *spécification d'entité* et sa vue interne est l'*architecture*. Ces deux dernières sont appelées unités de conception. Par conséquence, c'est l'ensemble de ces unités de conception qui constituent les bibliothèques *VHDL*.

De plus, dans une même description nous pouvons être amené à utiliser à plusieurs reprises le même algorithme. Dans ce cas, l'utilisation de sous-programmes apportera une plus grande évidence à l'ensemble. Une partie de ces sous-programmes sera, le plus souvent, regroupée dans une même description appelée *paquetage* (associée à la *spécification de paquetage*) dans laquelle nous pourrons trouver diverses déclarations profitables. Grâce à ces paquetages différents, un concepteur pourra utiliser des algorithmes dûment éprouvés par leur auteur.

Lorsque nous parlons de conception hiérarchisée, nous pensons souvent à l'utilisation répétée d'un même modèle. L'innovation d'es instances permet d'obtenir des copies similaires du modèle possédant les mêmes caractéristiques. La coexistence instance-modèle sera définie par une unité de conception appelée *configuration*.

Remarque 1.3.2

⇒ VHDL *est un langage de* type, *il offre quatre types d'objets différents :*

★ a. *Le type* scalaire *comprend les entiers, les flottants, les types physiques et les énumérés ;*

★ b. *Le type* composite *comprend les tableaux et les enregistrements ;*

★ c. *Le type* accès *représente les pointeurs ;*

★ d. *Le type* fichier *se passe de commentaire.*

Remarque 1.3.3

⇒ *A chacun de ces types pourra être associée une classe:*

★ a. *Les constantes dont la valeur sera fixée au plus tard au cours de la phase d'élaboration ;*

★ b. *Les variables dont la valeur est modifiable par affectation ;*

★ c. *Les signaux qui sont remaniés; l'affectation d'un signal est la transformation d'une liaison matérielle.*

Remarque 1.3.4

⇒ *Un signal a une valeur passée, présente et future. Ces deux dernières informations constituent le pilote du signal.*

1.3.1 *Objets et Opérateurs*

Cette partie présente les différents objets et opérateurs manipulés par *VHDL*.

Objets

⋆ *Commentaires*

Comme tout langage évolué, le *VHDL* autorise la présence de commentaires dans vos programmes source. Il s'agit de textes explicatifs destinés aux lecteurs du programme et qui n'ont aucune incidence sur sa compilation. Ils sont liés à la ligne où ils apparaissent, ils commencent par - - et se terminent avec le passage à la ligne suivante. Ils peuvent apparaître à tout endroit du programme où espace est autorisé. En général, cependant, nous nous limiterons à des emplacements propices à une bonne lisibilité du programme.

⋆ *Identificateurs*

Les identificateurs servent à désigner les différents *objets* manipulés par les programmes en *VHDL*: entrée, sortie, variables, fonctions, etc. Comme dans la plupart des langages, ils sont formés d'une suite de caractères choisis parmi les lettres ou les chiffres, le premier d'entre eux étant nécessairement une lettre. En ce qui concerne les lettres :

⇒ Le caractère *souligné* _ est considéré comme une lettre. Nous ne devons pas trouver de _ en début ou en fin d'identificateur, ni deux consécutifs.

⇒ Il n'y a aucune différence entre les majuscules et les minuscules.

Opérateurs

⋆ *Logique*

Ils sont pré-définis pour respectivement réaliser les opérations logiques *ET, OR, NOR et XOR* sur des opérandes de type booléen (*FAUX, VRAI* et *BIT '0', '1'*).

Ils fonctionnent également sur des tableaux à une dimension (aussi nommés *vecteurs* d'éléments de type booléen ou *BIT*), à condition que les opérandes aient la même longueur i.e., le même nombre d'éléments.

Une particularité des opérateurs fait que leur opérande de droite n'est pas forcément évaluée. En effet, si l'opérande de gauche, une fois évaluée, montre une valeur qui détermine le résultat de l'opération (une valeur *FAUX* pour l'opérateur *AND* par exemple), le compilateur négligera l'opérande de droite et gagnera donc, dans certains cas, un temps précieux. Ceci n'est pas une optimisation laissée au bon vouloir des constructeurs de compilateurs *VHDL*, c'est la norme qui le demande explicitement. Les surcharges de ces opérateurs, dont l'algorithme est écrit par les concepteurs, ne sont évidemment pas concernées.

⋆ *Relationnels*

Le résultat de ces opérateurs est de type booléen. Les opérateurs d'égalité et d'inégalité sont définis sur tous les types *VHDL* à l'exception du type fichier (*file*).

Lorsqu'il s'agit de comparer deux types composites (*composées de plusieurs éléments*), tous ces élément entrent en jeu. Lorsqu'il s'agit de comparer deux types accès (*pointeurs*) la comparaison porte exclusivement sur les adresses des objets pointés. Ces opérateurs sont définis sur tous les types énumérés. L'ordre du type énuméré est conservé, le premier élément étant considéré comme le plus petit.

⋆ *Addition*

Ce sont les opérateurs binaires (*ayant deux opérandes*) d'addition et de soustraction auxquels est adjoint celui (*toujours binaire*) de concaténation (*&*).

La concaténation est définie sur les tableaux à une dimension. Une chaîne de caractères (type *string*) n'étant rien d'autre en *VHDL* qu'un vecteur de caractères, la notion normale de concaténation est retrouvée.

Ainsi, l'opération *EI&STI* donne bien la chaîne *EISTI*.

⋆ *De Signe*

Les symboles $+, -$ sont des opérateurs unaire i.e;, ils ont un opérande unique. Si l'on surcharge les opérateurs d'addition et soustraction, les opérateurs de signe ne seront eux pas automatiquement surchargés. Donc, il faudra le faire explicitement. Il faut observer que, les opérateurs de signe sont moins prioritaires que les opérateurs de multiplication, ce n'est pas naturel. Donc, l'utilisation intensive des parenthèses est conseillée, par exemple $0.5/(-0.33)$.

⋆ *Multiplication*

Les mots clés $*$, $/$, mod *et* rem sont des opérateurs binaires. L'opération de division sur types entiers effectue un arrondi en tronquant la partie fractionnaire. Le mot clé rem pour reminder est le reste de la division entière.

⋆ *Divers*

Les autres mots clés sont $**$, abs, not. L'opérande de droite de $**$ doit impérativement être un entier. L'opérateur not est un opérateur logique unaire. Il opère donc sur les booléens, le type *BIT* et sur les tableaux à une dimension de ces deux types.

⋆ *Littéraux*

Les littéraux sont classés en : numériques, énumérés, chaînes de caractères ou chaînes de bits, et *null* qui est un littéral à lui tout seul et qui représente tout objet de type accès non initialisé.

$\Rightarrow 1.$ Les caractères se notent entre apostrophes : 'C'

$\Rightarrow 2.$ Les chaînes de caractères se notent entre guillemets : *chaîne de caractères*.

$\Rightarrow 3.$ Dans la notation des décimaux, l'utilisation des traits bas accroît leur lisibilité. Les nombres réels doivent toujours s'écrire avec un point (même s'ils tombent juste), par exemple *5_000.0 = 5000.0*.

$\Rightarrow 4.$ La notation de base permet d'écrire des entiers ou des réels dans les bases allant de 2 à 16, par exemple *2#0101# ou 8#3563#* . La notation par chaîne permet

de typer une chaîne en la faisant précéder du type correspondant : *B* pour binaire, *O* pour octal et *X* pour hexadécimal. Par exemple, *b"0101", x"fff"*.

1.3.2 *Unités de Conception*

Spécification d'entité

La spécification d'entité décrit l'interface entre les architectures et l'environnement extérieur. Nous noterons qu'à une même entité pourrait correspondre plusieurs architectures. Cette organisation spécification/architecture va permettre des modifications d'architecture sans que cela ne se répercute sur l'ensemble de la hiérarchie (à moins de modifier la spécification elle-même), comme l'illustre le Programme 1.3.5.

Programme 1.3.5 ***Spécification d'une entité.***

```
ENTITY d_fliflop IS
 PORT (D             : IN  BIT;  -- donnee_d_entree
       Reset         : IN  BIT;  -- entree_de_reset
       Clk           : IN  BIT;  -- horloge
       Q             : OUT BIT;  -- donnee_de_sortie
       NQ            : OUT BIT); -- sortie_complementaire
END d_flipflop;
```

Il ne faut surtout pas confondre le nom de la spécification et le nom de l'entité qui lui s'exprime par la Grammaire Formelle [1] 1.3.1.

[1] *The information contained herein inside box is copyrighted information of the IEEE, extracted from IEEE Std 1076-1987, IEEE Standard VHDL Reference Manual copyright ©1987 by the Institute of Electrical and Electronics Engineers, Inc. This information was written written the context of IEEE Std 1076-1987 and the IEEE takes no responsibility for or liability resulting from the reader's misinterpretation of said information resulting from the placement and context. Information is reproduced with the permission of the IEEE.*

Grammaire 1.3.2 ***Déclaration d'une entité.***

```
entity\_declaration  ::=
ENTITY identifier  IS
entity_header
entity_declarative_part
[BEGIN
 entity_statement_part]
END [ entity_simple_name ];
```

Par exemple, une bascule type D développe de façon *structurelle* est différente d'une bascule type *D* développe *comportementale*. La description comportamentale est illustré les Programmes 1.3.6.

Programme 1.3.6 ***Spécification d'une bascule type D de façon comportamentale.***

```
LIBRARY IEEE; USE IEEE.std_logic_1164.all;
ENTITY d_fliflop IS
 PORT (D             : IN  BIT;  -- donnee_d_entree
       Reset         : IN  BIT;  -- entree_de_reset
       Clk           : IN  BIT;  -- horloge
       Q             : OUT BIT;  -- donnee_de_sortie
       NQ            : OUT BIT); -- sortie_complementaire
END d_fliflop;
ARCHITECTURE d_fliflop_arch OF d_fliflop IS
BEGIN
  PROCESS(D, Reset, Clk)
   BEGIN
    IF Reset='0'  THEN -- asynchrone reset
         Q <= '0';
    ELSIF (Clk'event AND Clk='1' THEN -- front montante
        Q <= D;
    END IF;
  END PROCESS;
  NQ <= NOT Q;
END d_flipflop;
```

L'architecture décrit ce qui se passe à l'intérieur de la *boîte* définie par la spécification d'entité. Cette description sera faite en fonction du niveau d'abstraction désiré, c'est à dire que pour une même entité on pourra définir une architecture structurelle, *comportementale* ou flot de données.

L'architecture fait partie du domaine concurrent, comme l'illustre la Grammaire Formelle 1.3.3 par conséquent, elle reste le domaine du signal et ne peut contenir de déclaration de variable.

Grammaire 1.3.3 ***Description de l'architecture.***

```
architecture_body  ::=
ARCHITECTURE identifier OF entity_name IS
 architecture_declarative_part
 BEGIN
  architecture_statement_part
 END [architecture_simple_name] ;
```

1.3.3 *Configuration*

L'utilisation d'un composant dans la description hiérarchisée nécessite sa configuration. Deux possibilités sont offertes :

La configuration immédiate, faite à la suite de la déclaration du composant,précise le modèle dont il est l'instance, ses paramètres éventuels (*génériques*) et fixe la correspondance de ses ports. La configuration différée est utile pour configurer tous les composants au même moment. C'est le rôle de l'unité de conception appelée configuration ; l'autre avantage est la possibilité, en cours d'élaboration, de modifier la configuration du modèle utilisé sans avoir besoin de compiler deux fois l'ensemble de la Grammaire Formelle, comme l'illustre la Grammaire 1.3.4.

Grammaire 1.3.4 ***Description de la configuration.***

```
configuration_declaration  ::=
CONFIGURATION identifier OF entity_name IS
 configuration_declarative_part
 block_configuration
END [configuration_simple_name] ;
```

La partie déclaration ne contient que des clauses *use*, comme par exemple le Programme 1.3.7.

Programme 1.3.7 ***L'utilisation de la clause use.***

```
LIBRARY IEEE;
USE work.EISTI.all;
```

Les spécifications d'attributs du genre, comme par exemple le Programme 1.3.8.

Programme 1.3.8 ***L'utilisation de l'attribut.***

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY Exemple IS
 PORT (D              : IN  BIT;  -- donnee_d_entree
       Reset          : IN  BIT;  -- entree_de_reset
       Clk            : IN  BIT;  -- horloge
       Q              : OUT BIT;  -- donnee_de_sortie
       NQ             : OUT BIT); -- sortie_complementaire
 ATTRIBUTE delay OF Q : SIGNAL IS (8 NS , 10 NS);
END Exemple;
```

La partie configuration se décompose en une suite de blocs imbriqués les uns dans les autres, comme l'illustre la Grammaire Formelle 1.3.5 et le Programme 1.3.9. Un premier bloc englobant désigne l'architecture dans laquelle apparaissent les instances à configurer et pour chacune de ces instances ou liste d'instances, on

précise quel couple entité/architecture est utilisé.

Programme 1.3.9 ***Configuration composée de blocs imbriqués.***

```
LIBRARY IEEE; USE IEEE.std_logic_1164.all;
USE EISTI.ALL;
ENTITY comparateur_nibble IS
 PORT (A,B       : IN  BIT_VECTOR (3 DOWNTO 0);  -- donnee_d_entree
       PG        : IN  BIT;  -- plus_grand
       PP        : IN  BIT;  -- plus_petit
       EG        : IN  BIT;  -- egal
       A_PG_B    : OUT BIT;  -- donnee_de_sortie_A_plus_grand_B
       A_PP_B    : OUT BIT;  -- donnee_de_sortie_A_plus_petit_B
       A_EG_B    : OUT BIT); -- donnee_de_sortie_A_egal_B
END comparateur_nibble;
ARCHITECTURE arch_comparateur_nibble OF comparateur_nibble IS
 COMPONENT comp_niblle
      PORT (A,B      : IN  BIT_VECTOR (3 DOWNTO 0);
            PG       : IN  BIT;
            PP       : IN  BIT;
            EG       : IN  BIT;
            A_PG_B   : OUT BIT;
            A_PP_B   : OUT BIT;
            A_EG_B   : OUT BIT);
 END COMPONENT;
 SIGNAL A, B : BIT_VECTOR (3 DOWNTO 0);
 SIGNAL PG, PP, EG : BIT;
 SIGNAL VU : BIT := '1';
 SIGNAL VZ : BIT := '0';
 BEGIN
  I01: comp_nible PORT MAP (A, B, VZ, VU, VZ, PG, EG, PP);
  BEGIN
   PROCESS(Reset, Clk)
   BEGIN
    IF Reset='0'  THEN -- asynchrone reset
         Q <= '0';
    ELSIF (Clk'event AND Clk='1' THEN -- front montante
        Q <= D;
    END IF;
  NQ <= NOT Q;
 END d_flipflop;
```

Grammaire 1.3.5 ***Configuration composée de blocs imbriqués.***

```
block_configuration  ::=
FOR block_specification
 {use_clause}
 {configuration_item}
END FOR ;
```

1.3.4 *Spécification de Paquetage*

La spécification de paquetage déclare tous les objets qui sont susceptibles d'être exportés par le paquetage. Nous pourrons y trouver la déclaration de signaux, de constantes (les variables sont interdites), de types ou de sous-types ainsi que la déclaration de sous-programmes. De plus, nous pourrons y trouver des déclarations de composants, d'alias, de déclarations ou spécifications d'attributs, spécifications de connexion et des clauses *use*, comme l'illustre la Grammaire Formelle 1.3.6.

Grammaire 1.3.6 ***Description formelle du package.***

```
package_body ::=
PACKAGE BODY  package_simple_name IS
 package_body_declarative_part
END [ package_simple_name ];
```

Si la spécification ne contient aucune déclaration de sous-programme ou de constante à valeur différée, le corps de paquetage n'est pas nécessaire. Dans le cas contraire, cette unité de conception devra suivre la spécification.

1.3.5 *Corps de Paquetage*

Le corps de paquetage constitue la dernière unité de conception, nous y trouverons, les algorithmes des sous-programmes déclarés dans la spécification, la valeur des constantes à valeur différée. Pour un usage local, on pourra y déclarer des types, des sous-types, des constantes et des sous-programmes (*déclaration et corps*), comme l'illustre les Grammaires Formelles 1.3.7 et 1.3.8. Seule restriction, la déclaration de signal ou de variable reste prohibée.

Grammaire 1.3.7 ***Partie déclarative du package.***

```
package_body_declarative_part ::=
{package_body_declarative_item }
```

Grammaire 1.3.8 ***Partie déclaratifve d'item du corps package.***

```
package_body_declarative_item ::=
subprogram_declaration
|subprogram_body
|type_declaration
|subtype_declaration
|constant_declaration
|file_declaration
|alias_declaration
|use_clause
```

1.3.6 *Bibliothèque*

Le choix de la bibliothèque de travail se fait en dehors de la programmation *VHDL*. Toute utilisation d'une unité de conception externe à la bibliothèque (1.3.9) de travail devra être précédée de la référence de la bibliothèque à laquelle elle appartient.

Grammaire 1.3.9 ***Bibliothèque.***

```
library_clause ::=
LIBRARY logical_name_list;
```

Le compilateur *VHDL* fait une déclaration implicite devant toutes les unités de conception, comme l'illustre le Programme 1.3.10.

Programme 1.3.10 ***Unités de conception.***

```
LIBRARY IEEE; USE IEEE.std_logic_1164.all;
USE EISTI.ALL;
```

1.3.7 *Application*

Programme 1.3.11 ***Exemple package de composants GATE_EISTI_2001.***

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;
USE SYNOPSYS.attributes.all;
PACKAGE GATE_EISTI_2001 IS
  COMPONENT inverseur
  PORT (entree   : in  bit;
        sortie   : out bit);
  END COMPONENT;
  COMPONENT nand_2
   PORT(entree_1  : in  bit;
        entree_2  : in  bit;
        sortie    : out bit);
  END COMPONENT;
  COMPONENT nand_3
   PORT(entree_1  : in  bit;
        entree_2  : in  bit;
        entree_3  : in  bit;
        sortie    : out bit);
  END COMPONENT;
  COMPONENT dff
   PORT (d    : in  bit;
         set  : in  bit;
         reset : in  bit;
         clk  : in  bit;
         q    : out bit;
         qc   : out bit);
  END COMPONENT;
END GATE_EISTI_2001;
ENTITY inverseur IS
       PORT (entree   : in  bit;
             sortie   : out bit);
END inverseur;
```

Programme 1.3.12 ***Suite (1) de l'exemple package de composants GATE_EISTI_2001.***

```
ARCHITECTURE arch_inverseur OF inverseur IS
 BEGIN
  sortie <= not entree;
 END arch_inverseur
 ENTITY nand_2 IS
    PORT (entree_1   : in  bit;
          entree_2   : in  bit;
          sortie     : out bit);
 END nand_2;
 ARCHITECTURE arch_nand_2 OF nand_2 IS
 BEGIN
   sortie <= not (entree_1 and entree_2);
 END arch_nand_2 ;
 ENTITY nand_3 IS
  PORT (entree_1   : in  bit;
        entree_2   : in  bit;
        entree_3   : in  bit;
        sortie     : out bit);
END nand_3 ;
ARCHITECTURE arch_nand_3 OF nand_3 IS
BEGIN
  sortie <= not (entree_1 and entree_2 and entree_3);
END arch_nand_3;
 ENTITY dff IS
  PORT ( d     : in  bit;
         set   : in  bit;
         reset : in  bit;
         clk   : in  bit;
         q     : out bit;
         qc    : out bit);
END dff;
ARCHITECTURE arch_dff OF dff IS
signal a : bit_vector(1 downto 0);
BEGIN
```

Programme 1.3.13 ***Suite (2) de l'exemple package de composants GATE_EISTI_2001.***

```
DFF_PRO: PROCESS(clk)
         BEGIN
          IF(clk'event and clk = '1') THEN
          a(0) <= set;
          a(1) <= reset;
          CASE a IS
           WHEN "00" => q  <= '1';
                        qc <= '0';
           WHEN "01" => q  <= '0';
                        qc <= '1';
           WHEN "10" => q  <= '1';
                        qc <= '0';
           WHEN "11" => q  <= d;
                        qc <= not d;
           WHEN OTHERS    => null;
          END CASE;
          END IF;
         END PROCESS DFF_PRO;
 END arch_dff ;
```

2 Les Sous-Programmes

2.1 *INTRODUCTION*

Les sous-programmes permettent d'écrire des algorithmes réutilisables. Les valeurs des paramètres peuvent varier à chaque appel et donc donner lieu à des exécutions différentes.

Nous pouvons utiliser les sous-programmes pour augmenter la lisibilité d'un programme en faisant des découpes fonctionnelles, même si le sous-programmes n'est utilisé qu'une fois. En pratique, les sous-programmes sont principalement utilisés en VHDL lorsque nous avons besoin d'une suite d'instructions séquentielles et donc non concurrentes pour décrire un calcul, une conversion, des morceaux de processus ou des fonctions de résolution opérant sur des signaux.

Il existe deux genres de sous-programmes : les procédures (*procedure*) et les fonctions (*fonction*). Les procédures peuvent agir par effet de bord i.e., une modification (ou une consultation) de l'environnement par un autre moyen que les paramètres de sortie de procédures, les signaux de mode *out* des entités ou les valeurs de retour des fonctions. Par exemple, l'affectation d'un signal global (non déclaré localement mais dans un package) est un effet de bord. Les procédures modifient aussi (éventuellement) la valeur des paramètres transmis à l'appel. Les fonctions rendent un résultat et un seul et n'agissent pas par effet de bord. Elles ne peuvent lire ou écrire des variables ou des signaux que s'ils sont dans la partie déclaration de la fonction elle-même. Il n'y a pas de restriction sur l'usage des constantes par les fonctions. L'appel d'une procédure est une instruction alors que l'appel d'une fonction se met, comme une valeur, à droite du symbole d'affectation. Un sous-programme est constitué de deux parties : sa déclaration (optionnelle) et son corps. La déclaration d'un sous-programme (aussi appelée *spécification de sous-programme*, va indiquer son genre (procédure ou fonction), son nom, la liste

de ses paramètres (dits *paramètres formels*) et le type de la valeur de retour pour une fonction. Pour chaque paramètre formel, la déclaration du sous-programme précise son mode de passage (paramètre d'entrée, de sortie ou d'entrée et sortie) et son type. Le corps du sous-programme contiendra l'algorithme (*comment faire*), tout en étant parfaitement cohérent avec la déclaration. L'analyseur se chargera de garantir cette compatibilité.

2.2 *DÉCLARATION DU SOUS-PROGRAMME*

La déclaration du sous-programme est faite comme l'illustre la Grammaire Formelle 2.2.1.

Grammaire 2.2.1 ***Partie déclaratif d'item du corps package.***

```
subprogram_specification ::=
PROCEDURE designator [(formal_parameter_list)]
|FUNCTION designator [(formal_parameter_list) ] RETURN  type_mark
```

Programme 2.2.1 ***Exemple package de fonctions et processus FONCTION_EISTI_2001 .***

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;
USE SYNOPSYS.attributes.all;
PACKAGE FUNCTION_EISTI_2001 IS
 TYPE integers IS ARRAY (0 to 255) OF integer;
 FUNCTION fpg (a, b, vpg : bit) RETURN bit;
 PROCEDURE bin_2_int (bin : in bit_vector; int : out integer);
END FUNCTION_EISTI_2001 ;
PACKAGE BODY FUNCTION_EISTI_2001 IS
  FUNCTION fpg (a, b, vpg : bit) RETURN bit IS
   BEGIN
       RETURN ((a and vpg) or ((not b) and vpg) or (a and (not b)));
   END fpg;
  PROCEDURE bin_2_int (bin : in bit_vector; int : out integer) IS
  variable resultat : integer;
   begin
     resultat := 0;
       for i in bin'range loop
          if (bin(i) = '1') then
            resultat := resultat + 2**i;
          end if;
        end loop;
       int := resultat;
  END bin_2_int;
END FUNCTION_EISTI_2001;
```

2.3 *CORPS DU SOUS-PROGRAMME*

Les déclarations contenues dans le corps ne sont visibles que par lui-même. Toutes les déclarations sont permises; une restriction demeure, la déclaration de signal appartenant au domaine concurrent est interdite, comme l'illustre la Figure 2.4.1.

2.4 *DÉCLARATION DU SOUS-PROGRAMME*

La déclaration du sous-programme est faite comme l'illustre la Figure 2.2.1.

Grammaire 2.4.1 ***Corps du sous-programme.***

```
subprogram_body ::=
subprogram_specification  IS
subprogram_declarative_part
BEGIN
END [designator];
```

La différence fondamentale entre la fonction et la procédure apparaît dans la partie *instructions* :

$\Rightarrow a$. La fonction renvoie obligatoirement une valeur, grâce à l'instruction *return* suivie de cette valeur. Par contre, si la procédure doit rendre une valeur, elle le fera par l'intermédiaire d'un paramètre (en mode *OUT*) ;

$\Rightarrow b$. L'utilisation du *return* sans valeur est autorisée dans une procédure.

2.5 *APPEL DE SOUS-PROGRAMME*

Bien que les synthèses d'appel d'une procédure soient différentes de celles d'une fonction, l'affectation des paramètres suit la même règle. Il y a deux façons de réaliser ces affectations :

$\Rightarrow a$. Par position, il s'agit d'indiquer les valeurs à passer dans l'ordre où les paramètres ont été déclarés ;

$\Rightarrow b$. Par nom, la position n'est plus importante car le paramètre est nommé et lié à sa valeur par le symbole $\Rightarrow$.

2.6 *MÉCANISME DE LA SURCHARGE*

On dit que deux sous–programmes portant le même nom se surchargent si et seulement si :

$\Rightarrow a$. Ils ont le même nombre de paramètres ;

$\Rightarrow b$. Au moins un type (de base) diffère pour un paramètre correspondant ;

$\Rightarrow c$. *ET/OU* le type du résultat diffère s'il s'agit d'une fonction.

Par contre si une ou plusieurs des spécifications suivantes est différente pour deux sous-programmes du même nom, le compilateur diagnostiquera une double définition :

$\Rightarrow a$. *Nom* de paramètre ;

$\Rightarrow b$.*Classe* de paramètre ;

$\Rightarrow c$. *Mode* de passage ;

$\Rightarrow d$. *Sous-types* ;

$\Rightarrow e$. *Valeur par défaut.*

Le compilateur peut rencontrer des difficultés à l'appel d'un sous-programme surchargé, cette ambiguïté sera levée en nommant les paramètres lors de l'appel.

2.6.1 *Surcharge d'Opérateur*

C'est un cas particulier de la surcharge, il suffit de déclarer une fonction dont le nom est entre deux guillemets et qui possède le même nombre de paramètres que l'opérateur visé. Nous appelons ce *nouvel* opérateur de deux manières différentes :

$\Rightarrow a$. Utilisation identique à l'opérateur *officiel* ;

$\Rightarrow b$. Utilisation à la manière d'une fonction classique avec les paramètres entre parenthèses, le nom entré devra alors être utilisé.

2.7 *FONCTION DE RÉSOLUTION*

La fonction de résolution est une fonction permettant de lever l'ambiguïté sur la valeur d'un signal qui fait l'objet de plusieurs affectations concurrentes. La valeur renvoyée par cette fonction est appelée valeur résolue et du même coup le signal associé sera un signal résolu. Cette fonction est à définir par le concepteur, par contre le moment de l'appel est pris en charge par le simulateur. La déclaration d'une fonction de résolution est classique, et le type de la valeur de retour devra être le même que celui du signal résolu. Nous associons une fonction de résolution à un sous-type ou de façon plus restreinte à un signal.

3 Types de Données

3.1 *INTRODUCTION*

VHDL est un langage fortement typé, tout objet manipulé doit avoir un type défini avant la création. Indépendamment de son type, un objet appartient à une classe. Schématiquement, nous pouvons dire que le type définit le format des données et l'ensemble des opérations légales sur ces données, alors que la classe définit un comportement dynamique et précise la façon dont évolue (ou n'évolue pas dans le cas d'une constante !) une donnée au cours du temps.

Il existe en *VHDL* quatre types différents de données :

3.1.1 *Le Type Scalaire*

Constitué d'un seul élément, il englobe le type énuméré, le type entier, le type flottant et le type physique. Tous les types scalaires sont ordonnés, c'est à dire que toutes leurs valeurs peuvent être comparées à l'aide des opérateurs relationnels. Lors de la déclaration d'un type, il sera possible de spécifier un sous-ensemble d'un type scalaire, comme l'illustre la Grammaire Formelle 3.1.1.

Grammaire 3.1.1 ***Range.***

```
range ::=
range_attribute_name
|simple_expression direction simple_expression
```

Programme 3.1.1 ***Exemple du type scalaire.***

```
TYPE eit_nibble IS ARRAY (3 DOWNTO 0) OF std_logic ;
TYPE eit_byte   IS ARRAY (0 TO 7)     OF std_logic ;
```

La déclaration d'un type commence par le mot clé type.

3.1.2 *Le Type Énuméré*

La définition d'un type énuméré consiste à indiquer une liste d'identificateurs ou de caractères, chacun devant être distinct l'un de l'autre. Si un des constituants de la liste est défini dans plusieurs énumérations à la fois, nous disons qu'il est surchargé. Au moment de l'utilisation de ce littéral, le contexte devra permettre de lever l'ambiguïté, dans le cas contraire, une erreur sera détectée. Comme l'illustre la Grammaire Formelle 3.1.2.

Grammaire 3.1.2 ***Type énuméré.***

```
enumeration_type_definition ::=
(enumeration_literal {,enumeration_literal}
```

Programme 3.1.2 ***Exemple du type énuméré.***

```
TYPE eit  IS ('0','1');
TYPE test IS (OK, NO,GO);
```

3.1.3 *Le Type Entier*

Il suffit de spécifier l'intervalle de variation pour fixer les bornes du sous-ensemble. Les opérateurs arithmétiques s'appliquent à l'ensemble des types entiers. Le type *INTEGER* est un type prédéfini dont les bornes dépendent de la plate-forme sur laquelle est installé le compilateur comme l'illustre le Programmme 3.1.3.

Programme 3.1.3 ***La définition de l'intervalle.***

```
TYPE mesure IS RANGE 1 TO 100;
```

3.1.4 *Le Type Physique*

La valeur d'un type physique représente la mesure d'une quantité. Il est caractérisé par une unité de base, par un intervalle de valeurs et par éventuellement des unités secondaires. Ces dernières seront exprimées en fonction de l'unité de base. Les opérateurs arithmétiques s'appliquent à l'ensemble des types physiques, l'exemple qui suit est contenu dans le paquetage standard, comme l'illustre le Programme 3.1.4.

Programme 3.1.4 ***La définition de l'intervalle de temps.***

```
TYPE time IS RANGE $-2**63$ TO $(2**63)-1$
  units = fs;
  ps  = 1000fs;
  ns  = 1000ps;
  us  = 1000ns;
  ms  = 1000us;
  sec = 1000ms;
  min = 60ms;
  hr  = 60min;
 END units;
```

3.1.5 *Le Type Flottant*

Sa déclaration est identique à celle du type entier, la seule différence étant que les deux bornes de l'intervalle doivent être flottantes.

3.2 *LE TYPE COMPOSITE*

Il regroupe deux types distincts, les tableaux (ensemble d'objets de même type) et les enregistrements (ensemble d'objets qui peuvent être de types différents).

3.2.1 *Les Tableaux*

Un tableau est caractérisé par le nombre de ses indices (dimension du tableau), et d'index. Si la dimension du tableau est précisée à la déclaration, nous disons qu'il est contraint. S'il est non-contraint, sa dimension pourra être fixée lors de son utilisation. L'intervalle de variation d'un tableau contraint est fixé à l'aide des mots clés *TO* et *DOWNTO*, comme l'illustre le Programme 3.2.1.

Programme 3.2.1 ***La définition de l'intervalle avec downto.***

```
TYPE mot IS ARRAY (0 TO 31) of BIT;
TYPE reverse IS ARRAY (8 DOWNTO 1) of BIT;
```

La déclaration avec même sens s'écrit à l'aide du mot clé range associé au symbole <>, comme l'illustre par le Programme 3.2.2.

Programme 3.2.2 ***La définition de l'array.***

```
TYPE chaine IS ARRAY (integer range <>) of CHARACTER;
```

L'utilisation d'un tel type pour la déclaration d'une variable, représentant une chaîne de 32 caractères, se fait comme l'illustre le Programme 3.2.3.

Programme 3.2.3 ***La définition de chaîne.***

```
VARIABLE mot : CHAINE (0 to 31);
```

On verra par la suite que bien qu'un tableau soit non contraint, il sera possible de déterminer sa dimension, son sens de variation et d'autres informations grâce aux attributs prédéfinis associés au type tableau.

3.2.2 *Les Enregistrements*

La déclaration d'un type enregistrement consiste à énumérer les différents éléments qui le composent, comme l'illustre le Programme 3.2.4.

Programme 3.2.4 ***La définition de chaîne.***

```
 TYPE date IS
RECORD
  jour  : integer range 1 TO 31;
  mois  : nom_mois;
  ann\'{e}e : integer range 0 TO 3000;
END RECORD;
```

Un objet de type enregistrement pourra être affecté en une seule fois (*tous les éléments prenant tous une valeur*) ou par morceau, comme l'illustre le Programme 3.2.5. Soit le signal *TEMP* de type date :

Programme 3.2.5 ***La définition de signal.***

```
SIGNAL temp : date;
```

L'élément jour de ce signal sera modifié par l'instruction, comme l'illustre le Programme 3.2.6.

Programme 3.2.6 ***L'élément du record.***

```
temp.jour <= 31;
```

L'affectation complète de temp s'écrit comme l'illustre le Programme 3.2.7.

Programme 3.2.7 ***L'affectation complète.***

```
temp <= (10, fevrier, 1930);
```

Cette dernière affectation utilise la notation par agrégats, elle est applicable aux deux types composites. La notation par agrégats peut être de trois sortes.

3.2.3 *La Notation par Position*

Les valeurs du champ ou des cases sont placées dans l'ordre où elles ont été déclarées, comme l'illustre le Programme 3.2.8.

Programme 3.2.8 ***L'affectation du champ.***

```
(10, fevrier, 1930)
 -- ou
('a','b','c');
```

3.2.4 *La Notation par Nom*

Chaque champ ou indice précède sa valeur, l'ordre n'a plus d'importance, comme l'illustre le Programme 3.2.9.

Programme 3.2.9 ***Le champ ou indice sans ordre.***

```
(mois => fevrier, jour => 10, anneee=>1930)
 -- ou
(2=>'a', 1=>'b', 3=>'c');
```

3.2.5 *Utilisation du Mot Clé OTHERS*

Cela permet de donner, en une seule instruction, une même valeur à un groupe, comme l'illustre le Programme 3.2.10.

Programme 3.2.10 ***Le mot clé OTHERS.***

```
(mois => janvier, others => 10)
 -- ou
('a', others=>'z');
```

3.3 *LE TYPE ACCÈS*

Le type accès définit un pointeur, il représente l'adresse à laquelle est rangé l'objet sur lequel il pointe. Seules les variables peuvent être de type accès. Lors de l'exécution, il sera possible de créer dynamiquement un nouvel objet de type accès en allouant une zone mémoire (mot clé *new*). De la même façon, nous restituerions la zone mémoire qui n'est plus utile (mot clé *deallocate (désaffecter)*). A leur déclaration, les objets de types accès sont tous mis à la valeur initiale pardéfaut nulle, comme l'illustre le Programme 3.3.1 et le Programme 3.3.2.

Programme 3.3.1 ***L'utilisation du pointeur.***

```
TYPE pointeur IS ACCESS integer;
VARIABLE A: pointeur;
```

3.3.1 *Allocation*

Programme 3.3.2 ***L'initialisation du pointeur.***

```
A := NEW integer'(18);
A := NEW integer; -- est = integer'LOW;
```

3.3.2 *Restitution*

La restitution est comme l'illustre le Programme 3.3.3.

Programme 3.3.3 ***Le deadllocate.***

```
DEALLOCATE(A);
```

Grâce au type accès, il sera possible de créer des listes chaînées. La déclaration d'un pointeur sur un enregistrement contenant justement ce même pointeur pose le problème de la déclaration du dit enregistrement, comme l'illustre le Programme 3.3.4.

Programme 3.3.4 ***La déclaration d'un pointeur sur un enregistrement.***

```
TYPE paquet;
TYPE pointeurs sur paquet is access paquet;
TYPE paquet is record
 nom   : STRING;
 poids : real;
 paquet_suivant : pointeur_sur_paquet;
END record;
```

Soit une variable courant de type paquet, comme l'illustre le Programme 3.3.5.

Programme 3.3.5 ***La variable courant de type paquet.***

```
VARIABLE courant : pointeur sur paquet;
```

Courant peut prendre une valeur, comme l'illustre le Programme 3.3.6.

Programme 3.3.6 ***L'initialisation de courant de type paquet.***

```
courant : new paquet'("date",12.0,null);
```

3.4 *LE TYPE FICHIER*

Le type fichier est utile à plus d'un titre. D'abord il permet de répartir des données venant de l'extérieur, mais surtout, il servira de passerelle pour les dialogues console programme. La déclaration d'un type fichier est faite comme l'illustre le Programme 3.4.1.

Programme 3.4.1 ***La déclaration type fichier.***

```
TYPE texte IS FILE of STRING;
```

Le compilateur crée implicitement deux procédures et une fonction accomplissant les accès à ce type de fichier :

$\Rightarrow a$. *READ*, procédure de lecture ;

$\Rightarrow b$. *WRITE*, procédure d'écriture ;

$\Rightarrow c$. *ENDFILE*, fonction booléenne qui renvoie *vrai* si la fin de fichier est atteinte.

La déclaration d'un fichier et son mode d'accès seront vus en détail dans la partie suivante.

3.5 *LES EXPRESSIONS QUALIFIÉES*

Une expression qualifiée aide le compilateur à lever une ambiguïté sur le type d'une expression, notamment pour les sous programmes surchargés, comme l'illustre la Grammaire 3.5.1 et le Programme 3.5.1.

Grammaire 3.5.1 ***Type qualifié.***

```
type_qualifie ::=  TYPE'(expression)
```

Programme 3.5.1 ***L'expression qualifiée.***

```
WRITE(Ligne_courante, STRING'("101"));
```

3.6 *CONVERSION DE TYPE*

Les conversions autorisées concernent les types qui sont de structures proches et les tableaux (sous certaines conditions), comme l'illustre la Grammaire 3.6.1 et le Programme 3.6.1.

Grammaire 3.6.1 ***Conversion du type.***

```
type_conversion ::=  type_mark (expression)
```

Programme 3.6.1 ***Exemple de conversion du type.***

```
i := INTEGER(A*B);
```

Lors de la conversion d'une expression, il faut s'assurer qu'elle ne contient pas de valeur d'un type non autorisé. En ce qui concerne les tableaux, la conversion sera effectuée avec succès si le tableau conserve sa dimension, si ses éléments restent de même type et si les indices sont convertibles.

Copyright © by J. L. Mayorquim 11 Fevrier 2002 - 12h46

4 *Déclarations et Spécifications*

4.1 *INTRODUCTION*

En langage *VHDL*, tous les objets doivent être déclarés avant toute utilisation. Ces déclarations seront faites à l'intérieur de zones spécifiques, chaque unité de conception en possède une. Il reste la possibilité de faire des déclarations localement dans le corps d'un bloc, d'un processus ou d'un sous-programme.

4.2 *DÉCLARATION DE SOUS-TYPE*

Il peut être quelquefois utile de restreindre l'intervalle de valeur d'un type, tout en conservant la compatibilité avec celui-ci. Le type dont le sous-type est le sous ensemble, s'appelle le type de base, comme l'illustre la Grammaire 4.2.1 et le Programme 4.2.1.

Grammaire 4.2.1 *Subtype.*

```
subtype_declaration ::=
SUBTYPE identifier IS  subtype_indication ;
```

Programme 4.2.1 ***Déclaration de sous-type.***

```
SUBTYPE binaire  IS  std_logic RANGE '0' TO '1' ;
```

Le type de base peut lui-même être un sous type, dans ce cas le sous type déclaré sera compatible avec le type de base du sous type référence. En ce qui concerne la fonction de résolution, elle sera activée sous tous les signaux de ce sous type, les autres objets ne seront pas affectés par la fonction. On notera que la syntaxe mentionnée ci-dessus n'est pas applicable au sous type d'accès et de fichier concernant la fonction de résolution.

4.2.1 *Sous Type Dynamique*

La contrainte, si elle existe, pourra être évaluée lors de l'exécution de façon dynamique. Il suffit d'inclure dans la définition de contrainte un paramètre formel d'architecture ou de sous programme, comme l'illustre le Programme 4.2.2.

Programme 4.2.2 ***Sous-type avec contraintes.***

```
SUBTYPE  mes_entiers IS range 0 TO 255;
```

4.3 *DÉCLARATION DE CONSTANTE*

La valeur d'une constante, une fois déterminée, ne pourra être modifiée par la suite. Cette valeur pourra être le résultat d'une expression complexe ou d'un appel de fonction, comme l'illustre la Grammaire 4.3.1 et le Programme 4.3.1.

Grammaire 4.3.1 ***Constante.***

```
constant_declaration ::=
CONSTANT identifier_list :
subtype_indication [:= expression] ;
```

Programme 4.3.1 ***Déclaration de constante d'un sous-type.***

```
CONSTANT numero : INTEGER := 50 ;
```

Une constante ne pourra être ni de types accès, ni de type fichier. Si la partie :=valeur n'est pas précisée lors de la définition, c'est une constante à valeur différée. Ce genre de déclaration n'est tolérée que dans la spécification de paquetage. La valeur sera fixée dans le corps du paquetage correspondant.

4.4 *DÉCLARATION DE VARIABLE*

La variable est liée au domaine séquentiel, c'est la raison pour laquelle nous ne pourrons l'utiliser que dans les processus et les sous programmes, comme l'illustre la Grammaire 4.4.1 et le Programme 4.4.1.

Grammaire 4.4.1 ***Variable.***

```
variable_declaration ::=
VARIABLE  identifier_list :
subtyp_indication [ := expression ] ;
```

Programme 4.4.1 ***Déclaration de la variable sous-type.***

```
VARIABLE resultat : INTEGER ;
```

La variable est la seule classe d'objets pouvant être de type accès. Quelque soit son type, une variable possède une valeur par défaut :

$\Rightarrow a$. Énumération : valeur la plus à gauche ;

$\Rightarrow b$. Intervalle : valeur la plus petite ;

$\Rightarrow c$. Accès : nul ;

$\Rightarrow d$. Enregistrement : valeur par défaut de tous ses champs.

4.5 *DÉCLARATION DE SIGNAL*

Un signal associé à une fonction de résolution s'appelle un signal résolu. Si ce signal fait l'objet d'une affectation multiple, la fonction de résolution sera chargée par le simulateur de résoudre un éventuel conflit. Pour cette résolution, toutes les connexions du signal seront utilisées à l'exception de celles qui auront été déconnectées à cet instant. Seul un signal gardé peut faire l'objet d'une déconnexion (mot clé bus ou register). Un signal gardé ne peut être affecté qu'au cours d'une affectation gardée, c'est à dire contrôlée par le signal *GUARD*, comme l'illustre la Grammaire 4.5.1 et le Programme 4.5.1.

Grammaire 4.5.1 ***Signal.***

```
signal_declaration ::=
SIGNAL  identifier_list :
subtype_indication  [signal_kind] [:= expression ];
```

Programme 4.5.1 ***Déclaration d'un signal type BIT.***

```
SIGNAL temp_01, temp_02 : BIT ;
```

Au cours de cette affectation, si *GUARD* est vrai, l'affectation se déroule sans problème. Dans le cas contraire, le signal affecté est déconnecté après un délai spécifié par l'instruction de déconnexion. Si celle-ci n'existe pas ce délai sera nul. La différence entre les deux types de signaux gardés apparaît au cours d'une tentative de résolution et lorsque toutes les sources du signal sont déconnectées : Le registre conserve sa dernière valeur.

Le bus fait appel (à vide) à la fonction de résolution : ce cas devra donc être prévu par le concepteur. De plus, un signal ne pourra jamais être de type accès ou fichier.

4.6 *DÉCLARATION DE FICHIER*

Aucune affectation ne peut être faite sur un objet de type fichier, comme l'illustre la Grammaire 4.6.1 et le Programme 4.6.1.

Grammaire 4.6.1 ***Type fichier.***

```
file_declaration ::=
FILE  : subtype_indication IS  [mode]  file_logical_name;
```

Programme 4.6.1 ***Déclaration d'un objet de type fichier.***

```
FILE entree_logique_valeur_fichier :
logique_donnee IS IN "entree.dat";
```

Le mode d'accès à un fichier peut être *IN* ou *OUT*, le nom du fichier est une chaîne de caractères de type *STRING*. Le sous type doit être de type fichier. Le paquetage standard comprend la déclaration des entrées/sorties standards (i.e., la console), ce qui est très utile pour les dialogues console programme.

4.7 *DÉCLARATION D'ALIAS*

L'alias sert à rendre plus explicite le nom d'un objet, comme l'illustre la Grammaire 4.7.1 et le Programme 4.7.1.

Grammaire 4.7.1 ***Alias.***

```
alias_declaration ::=
ALIAS  identifier : subtype_indication IS name;
```

Programme 4.7.1 ***Déclaration d'alias.***

```
ALIAS c_flag : BIT IS flag_register (3);
```

La classe de l'objet est conservée pour l'alias, comme l'illustre le Programme 4.7.2.

Programme 4.7.2 ***La classe de l'objet est conservée pour l'alias.***

```
VARIABLE mot : STRING(0 TO 31);
ALIAS premier_lettre : character IS mot(0);
```

4.8 *DÉCLARATION DE COMPOSANT*

Cette déclaration va définir l'objet qui sera instancié par la suite pour créer l'ensemble des composants réels. Elle comprend le nom du composant et ses paramètres, comme l'illustre la Grammaire 4.8.1 et le Programme 4.8.1.

Grammaire 4.8.1 ***Composant.***

```
component_declaration ::=
COMPONENT   identifier
[locla_generi_clause]
[locla_port_clause]
END COMPONENT;
```

Programme 4.8.1 ***Déclaration de composant.***

```
COMPONENT nand_2
   PORT ( entree_1 : IN  BIT;
          entree_2 : IN  BIT;
          sortie   : OUT BIT);
END COMPONENT;
```

La correspondance entre le composant et le modèle (le contenu) sera faite après cette déclaration par la spécification de configuration et ce dans la partie de la déclaration englobante.

4.9 *DÉCLARATION ET SPÉCIFICATION D'ATTRIBUT*

Lors de la déclaration, une caractéristique est attachée à un type ou à un objet. Les attributs ne peuvent être que de classe constante de tout type sauf accès et

fichier, comme l'illustre la Grammaire 4.9.1 et le Programme 4.9.1.

Grammaire 4.9.1 ***Attribut.***

```
attribute_declaration ::=
ATTRIBUTE identifier : type_mark;
```

Programme 4.9.1 ***Spécification d'attribut.***

```
ATTRIBUTE numero : integer ;
TYPE coordonnees IS record x, y : integer
END record;
ATTRIBUTE location : coordonnees;
```

La spécification d'attribut permet de préciser à quel objet se rapporte un attribut et quelle valeur il va prendre, comme l'illustre la Grammaire 4.9.2 et le Programme 4.9.2.

Grammaire 4.9.2 ***Spécification d'attribut.***

```
attribute_specification ::=
ATTRIBUTE atrribte_deseignator  OF  entity_specification IS
          expression;
```

Programme 4.9.2 ***Exemple de la spécification d'attribut précise.***

```
ATTRIBUTE  retard OF element : SIGNAL IS (8 NS, 10 NS);
```

La partie élément définit l'objet, la liste d'objet ou la classe d'objet à laquelle se rapporte l'attribut, comme l'illustre le Programme 4.9.3.

Programme 4.9.3 ***L'élément définit l'objet par des constantes.***

```
ATTRIBUTE  numero OF B,C    : constant IS 10;
ATTRIBUTE  numero OF others : constant IS 0;
```

4.10 *SPÉCIFICATION DE DÉCONNEXION*

La spécification de déconnexion ne concerne que les signaux gardés. Elle fixe le délai au bout duquel le signal gardé sera déconnecté, comme l'illustre la Grammaire 4.10.1.

Grammaire 4.10.1 ***Spécification d'attribut.***

```
disconnection_specification ::=
DISCONNECT guarded_signal_specification  AFTER
time_expression;
```

A cette syntaxe s'ajoute, comme l'illustre le Programme 4.10.1.

Programme 4.10.1 ***La spécification de déconnexion des autres signaux gardés.***

```
DISCONNECT OTHERS : std_logic AFTER 6 NS;
```

Elle s'applique à tous les signaux gardés du type spécifié et ne faisant l'objet d'aucune spécification de déconnexion.

A cette syntaxe s'ajoute, comme l'illustre le Programme 4.10.2.

Programme 4.10.2 ***La spécification de déconnexion aux autres signaux gardés.***

```
DISCONNECT ALL : std_logic AFTER 6 NS;
```

Elle s'applique à tous les signaux gardés du type précisé : Ces trois spécifications sous entendent que les déclarations des signaux auxquelles elles s'appliquaient soient incluses dans la zone de la déclaration englobante.

4.11 *SPÉCIFICATION DE CONFIGURATION*

La spécification de configuration permet d'associer aux instances de composant une spécification d'entité avec la correspondance entre les ports du composant et ceux de l'entité. Elle sera placée à la suite de la déclaration du composant, comme l'illustre le Programme 4.11.1 ou le Programme 4.11.2.

Programme 4.11.1 ***La spécification de configuration avec entité.***

```
FOR a1 : nand_2
USE ENTITY  WORK.GATE_EISTI(arch_nand_2) ;
GENERIC MAP (thplh => 2 NS, tphl => 3 NS);
PORT ( entree_1 : IN  BIT;
       entree_2 : IN  BIT;
       sortie   : OUT BIT);
END FOR;
```

Programme 4.11.2 ***La spécification de configuration avec configuration.***

```
CONFIGURATION arch_test OF test_complet IS
 FOR a1 : nand_2
 USE ENTITY  WORK.GATE_EISTI(arch_nand_2);
 GENERIC MAP (thplh => 2 NS, tphl => 3 NS);
 PORT ( entree_1 : IN  BIT;
        entree_2 : IN  BIT;
        sortie   : OUT BIT);
 END FOR;
END arch_test;
```

Il reste la possibilité de ne spécifier que la correspondance des ports tout en différant le choix de l'entité, comme l'illustre le Programme 4.11.3.

Programme 4.11.3 ***La spécification de configuration sans choix d' entité.***

```
FOR a1 : nand_2
USE OPEN;
GENERIC MAP (thplh => 2 NS, tphl => 3 NS);
PORT ( entree_1 : IN  BIT;
       entree_2 : IN  BIT;
       sortie   : OUT BIT);
END FOR;
```

On peut fixer une même configuration pour toutes les instances d'un même composant, comme l'illustre le Programme 4.11.4, ou pour un groupe d'instances, comme l'illustre le Programme 4.11.5.

Programme 4.11.4 ***La spécification de configuration pour tous les instanciés.***

```
FOR ALL : nand_2
USE .....;
```

Programme 4.11.5 ***La spécification de configuration pour un groupe.***

```
FOR OTHERS : nand_2
USE .....;
```

Au cours de cette affectation, si *GUARD* est vrai, l'affectation se déroule sans problème. Dans le cas contraire, le signal affecté est déconnecté après un délai spécifié par l'instruction de déconnexion. Si celle-ci n'existe pas ce délai sera nul. La différence entre les deux types de signaux gardés apparaît au cours d'une tentative de résolution et lorsque toutes les sources du signal sont déconnectées : Le registre conserve sa dernière valeur.

Le bus fait appel (à vide) à la fonction de résolution : ce cas devra donc être prévu par le concepteur. De plus, un signal ne pourra jamais être de type accès ou fichier.

4.12 *DÉCLARATION DE FICHIER*

Aucune affectation ne peut être faite sur un objet de type fichier, comme l'illustre la Grammaire 4.12.1 et le Programme 4.12.1.

Grammaire 4.12.1 ***Type fichier.***

```
file_declaration ::=
FILE  : subtype_indication IS  [mode]  file_logical_name;
```

Programme 4.12.1 ***Déclaration d'un objet de type fichier.***

```
FILE entree_logique_valeur_fichier :
logique_donnee IS IN "entree.dat";
```

5 *Instructions Séquentielles*

5.1 *INTRODUCTION*

Bien que le fond d'une description soit généralement concurrent (cela consiste à décrire un fonctionnement global sans donner l'ordre d'affectation des signaux), il est souvent utile de pouvoir revenir au domaine séquentiel pour un certain nombre d'opérations d'algorithmiques.

VHDL possède un jeu d'instructions séquentielles très complet, mais le domaine d'utilisation de ces instructions est restreint : un processus ou un corps de sous-programme.

5.2 *INSTRUCTION WAIT*

Cette instruction, placée dans le bloc *begin-end* d'un processus ou d'une procédure, va permettre de contrôler le processus voire de l'interrompre définitivement. Sa fonction première est de stopper l'exécution d'un processus et de la relancer si toutes les conditions sont remplies, comme l'illustre la Grammaire 5.2.1 et le Programme 5.2.1.

Grammaire 5.2.1 ***Instruction WAIT.***

```
wait_declaration ::=
WAIT  [sensitivity_clause ]
      [condition_clause]
      [ timeout_clause ];
```

Programme 5.2.1 ***L'instruction WAIT.***

```
WAIT on test_01, test_02 UNTIL Cond_01 FOR 5 ms ;
```

Les trois paramètres sont optionnels : wait, utilisé seul, donnera comme résultat l'arrêt définitif du processus. Lorsqu'une liste de signaux est spécifiée, wait attend l'arrivée d'un événement sur un des signaux de la liste. S'il y a une condition booléenne, elle est évaluée et c'est sa valeur qui détermine la reprise. Le mot clé *for* indique que le processus sera interrompu, au plus pendant le temps spécifié. Les signaux *contrôlables* par wait doivent être accessibles en lecture. Cette instruction est interdite dans une fonction.

5.3 *INSTRUCTION D'ASSERTION*

Cette instruction permet de renvoyer un message lorsque la condition devient fausse, comme l'illustre la Grammaire 5.3.1 et le Programme 5.3.1.

Grammaire 5.3.1 ***Instruction ASSERTION.***

```
assertion_declaration ::=
ASSERT  condition
[REPORT expression]
[SEVERETY expression];
```

Programme 5.3.1 ***L'instruction d' assertion.***

```
ASSERT test < 1 min
       REPORT "Fin normale de la simulation" SEVERITY ERROR;
```

Le message doit être une chaîne de caractère, sa valeur par défaut est *assertion violation*. Le niveau de sévérité est du type severity -level, c'est un type pré-défini, comme l'illustre la Figure 5.3.2.

Programme 5.3.2 ***L'instruction type pré-défini.***

```
type severity-level is (NOTE, WARNING, ERROR, FAILURE);
```

La valeur par défaut est error. D'un intérêt non négligeable au cours de la mise au point du programme, le message devra être le plus documenté possible à sa provenance (entité, architecture,...) et son niveau de sévérité.

5.4 *INSTRUCTION D'AFFECTATION DE SIGNAL*

Contrairement à l'affectation de variable, elle ne modifie pas la valeur actuelle du signal. C'est le pilote de ce signal qui est modifié en vue des valeurs futures qu'il pourra prendre en cours d'exécution, comme l'illustre le Programme 5.4.1.

Programme 5.4.1 ***L'instruction d'affectation de signal.***

```
A <= B AFTER 10ns,'1' AFTER 20ns;
```

Dans cet exemple, le signal A prendra la valeur actuelle B dans 10 ns puis passera à "1" 10 ns après . Les temps indiqués doivent impérativement être positifs

et placés dans l'ordre croissant. Une affectation du genre A <= B correspondant à un délai nul (équivalent à A<= B AFTER 0 ns) est autorisée : dans ce cas, le temps au bout duquel la valeur de B sera prise par A est déterminé par le simulateur.

5.5 *MODES DE TRANSMISSION*

Il s'agit de la façon de considérer une modification de la valeur d'un signal.

5.5.1 *Le Mode Inertie*

Ce mode, pris par défaut, ne prend en compte que les événements dont la durée est au moins égale au temps de transmission.

5.5.2 *Le Mode Fréquence*

Ce mode sera précisé lors de l'affectation du signal avec le mot clé transport. Dans ce mode, tous les événements qui surviennent sont pris en compte quelle que soit leur durée, comme l'illustre le Programme 5.5.1.

Programme 5.5.1 ***L'affectation du signal avec le mot clé TRANSPORT.***

```
A <= TRANSPORT B AFTER 10 ns;
```

Remarque 5.5.1

En langage logiciel, toutes les déclarations sont séquentielles. Donc, les déclarations sont écrites dans l'ordre de leur écriture, c'est à dire, top-down*. Nous verrons ensemble des exemples, qui permettra de bien comprendre les modes* inertie *et* fréquence*.*

Exemple 5.5.1

Un retard est mieux représenté par le mode inertie, un exemple de l'architecture en VHDL *est représenté par le Programme 5.5.2.*

Programme 5.5.2 ***La description de transport et inertie.***

```
ARCHITECTURE retard OF rc1 IS
 SIGNAL signal_1, signal_2, signal_3  : BIT;
 BEGIN
   signal_1 <= signal_3 AFTER 5 NS;
   signal_2 <= TRANSPORT signal_3 AFTER 5 NS;
 END retard;
```

5.6 *AFFECTATION DE SIGNAL*

La syntaxe générale de l'affectation de signal prévoit l'affectation par agrégat, comme l'illustre la Figure 5.6.1.

Programme 5.6.1 ***L'affectation par agrégat.***

```
nom_ou_agregat <= {TRANSPORT} valeur{, valeur};
```

où la valeur est définit comme l'illustre le Programme 5.6.2.

Programme 5.6.2 ***L'affectation du signal est définie par la valeur temps.***

```
expression {AFTER temps}
-- ou
agregate   {AFTER temps}
```

Du fait de la difficulté d'identification, l'agrégat de droite ne pourra contenir le mot clé others.

5.7 *AFFECTATION DE VARIABLE*

Nous avons vu dans la partie consacrée aux déclarations qu'il était possible de donner une valeur à une variable dans sa déclaration. Il est bien entendu possible de changer cette valeur à tout moment, ":=" est le symbole de l'affectation de variable. C'est une classe exclusive du type accès. Donc, la variable pourra faire l'objet d'affectation dynamique.

5.8 *APPEL DE PROCÉDURE*

Pris dans son contexte séquentiel, l'appel d'une procédure sera considéré comme une simple instruction et son déroulement devra être terminé pour que l'instruction suivante soit exécutée, comme l'illustre la Grammaire 5.8.1 et le Programme 5.8.1.

Grammaire 5.8.1 ***Procédure.***

```
procedure_call_statement ::=
procedure_name [(actual_parameter_part)];
```

Programme 5.8.1 ***La procédure.***

```
PROCEDURE multi(entree: IN std_logic;
                sortie: OUT std_logic) IS
```

5.9 *STRUCTURE CONDITIONNELLE*

Elle permet d'ajouter une condition à l'exécution d'une séquence d'instructions, comme l'illustre la Grammaire 5.9.1 et le Programme 5.9.1.

Grammaire 5.9.1 ***Instruction IF.***

```
if_statement ::=
IF condition THEN
 sequence_of_statements
 {ELSIF condition  THEN
   sequence_of_statements}
 [ELSE
   sequence_of_statements]
 END IF ;
```

Programme 5.9.1 ***Structure conditionnelle.***

```
IF (ENTREE = '1') THEN
 resultat <= "000111" ;
END IF;
```

5.10 *INSTRUCTION CASE*

Cela permet d'exécuter une séquence d'instructions suivant la valeur d'une expression, comme l'illustre la Grammaire 5.10.1 et le Programme 5.10.1.

Grammaire 5.10.1 ***Instruction CASE.***

```
case\_statement ::=
CASE  expression  IS
case_statement_alternative
{case_statement_alternative}
END CASE ;
```

Programme 5.10.1 ***Exemple de l'instruction CASE.***

```
CASE ENTREE  IS
 WHEN "00"    => resultat <= "000111" ;
 WHEN "10"    => resultat <= "111000" ;
 WHEN OTHERS  => resultat <= "000000" ;
END CASE;
```

Remarque 5.10.1

L'ordre n'a aucune importance, seul le choix OTHERS *doit figurer à la fin de la liste.*

5.11 *BOUCLES*

La boucle sert à exécuter, un nombre de fois déterminé, un ensemble d'instruction, comme l'illustre la Grammaire 5.11.1 et le Programme 5.11.1.

Grammaire 5.11.1 ***Boucle.***

```
loop_statement ::=
[loop_label :]
[iteration_scheme] LOOP
sequence_of_statements
END LOOP [loop_label] ;
```

Programme 5.11.1 ***Les boucles.***

```
FOR i IN 0 TO (bin'LENGTH -1)
 IF (temp MOD 2 =1) HTEN
   bin(i) := '1' ;
 ELSE bin(i) := '0' ;
 END IF;
   temp := temp /2 :
END  LOOP ;
```

Si aucun mode d'itération n'est spécifié, la boucle résultante est infinie et aucune instruction n'est prévue pour en sortir. Nous distinguons deux modes d'itération :

⇒ La boucle *TANT QUE*, comme l'illustre la Grammaire 5.11.2 et le Programme 5.11.2.

Grammaire 5.11.2 ***Instruction WHILE.***

```
iteration_scheme ::=
WHILE condition
| FOR loop_parameter_specification
```

Programme 5.11.2 ***La boucle TANT QUE.***

```
WHILE I > 10 LOOP
     ............
     ............
END LOOP;
```

⇒ La boucle *FOR*, comme l'illustre le Programme 5.11.3.

Programme 5.11.3 ***La boucle FOR.***

```
FOR ENTREE IN 0 TO 32  LOOP
   ..................

   ..................
END LOOP;
```

Exception à la règle, l'indice de boucle ne doit pas être déclaré. L'intervalle de variation peut être de type énuméré ou un intervalle explicite. Les labels en tête de boucle seront utiles lors des instructions next ou exit.

5.12 *INSTRUCTION NEXT*

Cette instruction permet de sortir de l'itération en cours et de passer à la suivante (si elle existe). Elle est :

$\Rightarrow a.$ Imperative *NEXT* label de boucle.

$\Rightarrow b.$ Conditionnelle *NEXT* label de boucle *WHEN* condition.

Dans le mode conditionnel, le saut ne s'exécute que si la condition est vraie. Les labels deviennent très utiles lorsqu'il s'agit de sortir de boucles imbriquées les unes dans les autres. Sans label, le saut concernera la boucle directement englobante.

5.13 *INSTRUCTION EXIT*

Plus forte que *NEXT*, elle permet de sortir définitivement de la boucle. Elle est :

$a.$ Impérative *EXIT* label de boucle. $b.$ Conditionnelle *EXIT* label de boucle *WHEN* conditions(même mécanisme que *NEXT*, comme l'illustre le Programme 5.13.1.

Programme 5.13.1 *L'instruction EXIT.*

```
EXIT WHEN x = 32;
```

5.14 *INSTRUCTION RETURN*

Instruction réservée aux sous programmes, elle exécute un saut au processus appelant. Si cette instruction est obligatoire dans une fonction, elle peut être utilisée dans une procédure pour un retour forcé, comme l'illustre le Programme 5.14.1.

Programme 5.14.1 ***L'instruction EXIT.***

```
FUNCTION "+" (a , b : std_logic) RETURN std_logic IS
BEGIN
 RETURN (a OR b) ;
END "+";
```

5.15 *INSTRUCTION NULL*

Comme son nom l'indique, cette instruction ne fait rien. Elle sera utilisée dans une structure case pour un choix qui ne doit engendrer aucune modification, comme l'illustre le Programme 5.15.1.

Programme 5.15.1 ***L'instruction NULL.***

```
CASE choix IS
 WHEN "00" => null;
 WHEN OTHERS => resultat <= "01010101" ;
END CASE;
```

6 *Instructions Concurrentes*

6.1 *INTRODUCTION*

Une instruction concurrente s'exécute de façon indépendante vis-à-vis des autres instructions. L'ordre d'écriture dans le programme n'est pas déterminant.

On retrouve dans la liste des instructions concurrentes quelques instructions décrites dans la partie dédiée au séquentiel. Nous verrons que leur comportement n'est plus tout à fait le même. Ces instructions se trouveront dans une spécification d'entité, une architecture ou un bloc (qui est lui-même une instruction concurrente).

6.2 *PROCESSUS*

Le processus a déjà été cité dans la partie précédente, il ne peut contenir que des instructions séquentielles. Une fois lancée, un processus ne s'arrêtera qu'en fin de simulation si aucune instruction ne le bloque (*wait sans argument*). Un processus se déroule simplement. Au cours de la première phase, appelée élaboration, il évalue tous les objets présents dans sa partie déclaration. La deuxième phase consiste à exécuter sans cesse les instructions comprises entre *begin* et *process*, comme l'illustre a Grammaire 6.2.1 et le Programme 6.2.1.

Grammaire 6.2.1 ***Process.***

```
process_statement ::=
[proces\_label : ]
PROCESS [(sensitivity_list)]
 process_declarative_part
 BEGIN
  process_statement_part
 END PROCESS [process_label];
```

Programme 6.2.1 ***L'exemple de PROCESS.***

```
TEST: PROCESS {reset, set, clk}
      VARIABLE state : BIT := '0';
      BEGIN
       IF set ='1' THEN
        state := '1';
       ELSIF reset = '1' THEN
        state := '0';
       ELSIF clk = '1' and clk'EVENT THEN
        state := d;
       END IF;
       q  <= state ;
       qc <= not state ;
   END PROCESS TEST;
```

A chaque instruction concurrente, nous pouvons faire correspondre son processus (ou bloc) équivalent. Celui de l'instruction précédente pourrait s'écrire comme dans le Programme 6.2.2.

Programme 6.2.2 ***Les signaux concurrents.***

```
{label :} PROCESS
  ..........................
  partie declaration
  ..........................
  BEGIN
      ..........................
      instructions sequentielles
      ..........................
      {WAIT on liste_de_signaux}
  END PROCESS {label};
```

Lors de la phase d'élaboration, le processus s'exécute au moins une fois puis il s'arrête à la première instruction *WAIT* qu'il trouve (si elle existe !).

6.2.1 *Conventions*

$\Rightarrow a$. Les signaux à surveiller doivent pouvoir être identifiés au cours de la compilation ;

$\Rightarrow b$. L'instruction *wait* ne pourra figurer dans le corps du processus que si la liste est vide; aucune procédure appelée par ce processus ne pourra contenir d'instruction *wait*.

6.2.2 *Processus Passif*

Dans la spécification d'entité, seules les instructions dont le processus équivalent est passif sont tolérées. Un processus est passif si son corps ne contient aucun signal apparaissant à gauche d'une instruction d'affectation. Le processus est passif parce qu'il n'entraîne pas la création de nouveau processus.

6.3 *INSTRUCTION BLOC*

Le bloc a deux objectifs :

$\Rightarrow a$. Réunir un ensemble d'instructions concurrentes en leur faisant partager les déclarations locales du bloc ;

$\Rightarrow b$. Garder, à l'aide d'une condition, des affectations de signaux, comme l'illustre le Programme 6.3.1.

Programme 6.3.1 ***L'instruction bloc.***

```
{label :} BLOCK {(condition_de_garde)}
 {entete_de_selectivite_et_portes}
 partie declarations locales
 ........................
 BEGIN
     ........................
     instructions concurrentes
     ........................
 END BLOCK {label};
```

La condition de garde est une expression booléenne.

6.3.1 *Garde d'Affectation*

Cette instruction revient à conditionner l'affectation d'un signal. Si la condition de garde est vrai toutes les affectations gardées sont évaluées en conséquence. Le mot clé *guarded* à droite d'une affectation signifie qu'il s'agit d'une affectation gardée. Un signal gardé ne peut être affecté qu'au cours d'une telle affectation.

6.4 *APPEL CONCURRENT DE PROCÉDURE*

Étant dans le domaine concurrent, aucun paramètre ne pourra être classé variable. La syntaxe d'appel est la même que pour le séquentiel. A chaque appel de procédure, nous pourrons faire correspondre un processus équivalent, comme l'illustre le Programme 6.4.1.

Programme 6.4.1 ***L'appel de la procédure concurrente.***

```
Calcul(a,b);
```

La procédure concurrente s'écrit, comme dans le Programme 6.4.2.

Programme 6.4.2 ***Insertion de la procédure concurrente.***

```
 PROCESS
BEGIN
  Calcul (a,b);
  WAIT on a,b;
END PROCESS;
```

Si la procédure contient des paramètres en mode *IN* ou *INOUT*, l'instruction *WAIT* réveillera le processus à chaque évènement survenant sur les signaux surveillés. Si la procédure n'en contient aucun, *WAIT* sera sans paramètre et donc cela revient à une exécution unique du processus équivalent.

Remarque 6.4.1

Une procédure sans paramètre en mode IN *ou* INOUT *a un processus équivalent passif.*

6.5 *INSTRUCTION D'ASSERTION CONCURRENTE*

La syntaxe est identique à l'instruction séquentielle, la différence est le processus équivalent, comme l'illustre le Programme 6.5.1.

Programme 6.5.1 ***Instruction d'assertion concurrente.***

```
{label :}ASSERT condition {REPORT message}
{SEVERITY nivea_ de_severite}
```

Le processus est équivalent, comme dans le Programme 6.5.2.

Programme 6.5.2 ***Équivalence de l'assertion pour l'instruction processus.***

```
{label :} PROCESS
          BEGIN
           ASSERT condition {REPORT message}
           {SEVERITY niveau_de_severite}
           WAIT on liste_de_signaux_de_la_condition;
          END PROCESS {label};
```

Un tel processus est également passif. Il faut néanmoins faire attention à la condition car, si elle ne contient aucun signal, ce processus ne tournera qu'une fois !

6.6 *INSTANCE DE COMPOSANT*

Une instance est une copie du composant déclaré, comme l'illustre le Programme 6.6.1.

Programme 6.6.1 ***La déclaration d'une instance de composant.***

```
label : nom_du_composant
        {correspondance_des_parametres_generiques}
        {correspondance_des_ports};
```

Le label sert à nommer l'instance créée, comme l'illustre le Programme 6.6.2.

Programme 6.6.2 ***Le label de l'instance.***

```
SIGNAL A, B, C, D : BIT;
COMPONENT INV
PORT( E  : IN  BIT;
      S  : OUT BIT);
END COMPONENT;
for INV1 : INV use ENTITY INVERSEUR (comportamental);
for INV2 : INV use ENTITY INVERSEUR_2 (comportamental);
          ..................................
          ..................................
INV1 : INV PORT MAP (E=>A; S=>C);
INV2 : INV PORT MAP (E=>B; S=>D);
```

6.7 *INSTRUCTION GENERATE*

Cette instruction permet l'élaboration itérative ou conditionnelle d'instructions concurrentes, comme l'illustre le Programme 6.7.1 et le Programme 6.7.2.

Programme 6.7.1 ***L'instruction generate avec IF.***

```
label : IF condition GENERATE
    ......................
    instructions concurrentes
    ......................
END GENERATE {label};
```

Programme 6.7.2 ***L'instruction generate avec FOR.***

```
label : FOR parametre_generation IN intervalle GENERATE
         ......................
         instructions concurrentes
         ......................
        END GENERATE {label};
```

Condition est une expression booléenne, paramètre-generation est un identificateur qui changera de valeur à chaque intération et l'intervalle est un intervalle discret. Il est impératif que tous les paramètres soient connus au moment de la phase d'élaboration, nous pourrons donc utiliser des constantes ou des paramètres génériques.

6.8 *LA SÉLECTIVITÉ*

La sélectivité permet de transmettre un paramètre sans que ce paramètre soit en mode *IN* ou *INOUT*. A l'intérieur de la structure, ce paramètre sera considéré comme une constante. Deux structures peuvent être génériques, la spécification d'entité et le bloc, comme l'illustre la Figure 6.8.1.

Programme 6.8.1 ***Les paramètres génériques.***

```
GENERIC
  (parametre {, parametre} : TYPE {:= valeur_par_defaut};
  ..................................................
  ..................................................
  ..................................................
  (parametre {, parametre} : TYPE {:= valeur_par_defaut};
  (parametre {, parametre} : TYPE {:= valeur_par_defaut});
```

La spécification du type peut être de base ou une indication de *SOUS-TYPE* (bornes de tableau non-contraint), comme l'illustre le Programme 6.8.2.

Programme 6.8.2 ***La spécification d'identification de type.***

```
  GENERIC (T : integer :=2;
  MOT : BIT_VECTOR(1 TO 31));
```

6.9 *ATTRIBUTS*

Les attributs sont des caractéristiques associées à un type ou directement à un objet dont la valeur pourra être connue à tout moment de l'exécution. Cette valeur est représentée par l'identificateur formé du nom de l'attribut suivi du nom de l'objet ou du type et séparé par une *quote*. Nous distinguions deux types d'attributs, les attributs pré-définis et les attributs définis par le concepteur. Ces derniers ont été largement abordés dans la partie sur les déclarations. Voici donc la liste de tous les attributs pré-définis en fonction des objets auxquels ils se rapportent. Attributs se rapportant aux types ou sous-types nommé T, comme l'illustre le Tableau 6.1.

Table 6.1 ***Les attributs se rapportant aux types ou sous-types.***

Attributs	*Résultat*	*Paramètre*	*Contrainte*
T'BASE	*Type de base de T*		*Préfixe d'un autre attribut : T'BASE'LEFT*
T'LEFT	*Borne de gauche de T*		*Type scalaire et sous-type*
T'RIGHT	*Borne de droite de T*		*Type scalaire et sous-type*
T'HIGH	*Borne supérieure de T*		*Type scalaire et sous-type*
T'LOW	*Borne inférieure de T*		*Type scalaire et sous-type*
T'POS(X)	*Position de X*	*Type T*	*Type énumère ou physique et sous-type*
T'VAL(X)	*Expression en position X*	*Entier*	*Type énumère ou physique et sous-type*
T'SUCC(X)	*Prochaine position celle de X*	*Type T*	*Type énumère ou physique et sous-type*
T'PRED(X)	*Position antérieure celle de X*	*Type T*	*Type énumère ou physique et sous-type*

Les attributs se rapportant aux tableaux, sous-types contraints et alias de tableaux, comme l'illustre le Tableau 6.2.

Table 6.2 ***Les attributs se rapportent aux tableaux.***

Attributs	*Résultat*	*Paramètre*
T'LEFT$[(N)]$	*Borne de gauche de l'intervalle N de A. Par défaut N =1.*	*Dimension* $<$ *A.*
T'RIGHT$[(N)]$	*Borne de droite de l'intervalle N de A. Par défaut N =1.*	*Dimension* $<$ *A.*
T'HIGH$[(N)]$	*Borne supérieure*	*Dimension* $<$ *A.*
T'LOW$[(N)]$	*Borne inférieure*	*Dimension* $<$ *A.*
T'RANGE$[(N)]$	*Intervalle définit par RIGHT et LEFT. Par défaut* $= 1$.	*Dimension* $<$ *A.*
T'REVERSE_RANGE$[(N)]$	*Intervalle définit par RIGHT et LEFT. Par défaut* $= 1$.	*Dimension* $<$ *A.*
T'LENGTH$[(N)]$	*Valeur de l'intervalle N. Par défaut* $= 1$.	*Dimension* $<$ *A.*

Les attributs concernant les objets de classe de signal, comme l'illustre le Programme 6.3.

Table 6.3 ***Les attributs concernant les objets de classe de signal.***

Attributs	*Résultat*	*Remarque*
S'DELAYED$[(t)]$	*Délai de t unités de temps.*	*Valeur positive, zéro par défaut.*
S'STABLE$[(t)]$	*Vrai si S n'est pas changé depuis t unités de temps.*	*Valeur positive, zéro par défaut.*
S'QUIET$[(t)]$	*Vrai si S est resté calme depuis t unités de temps.*	*Valeur positive, zéro par défaut.*
S'TRANSACTION$[(t)]$	*Change de valeur chaque fois que S est actif.*	*Type BIT*
S'EVENT$[(t)]$	*Vrai si S change de valeur.*	*Type Booléenne*
S'LAST_EVENT$[(t)]$	*Temps écoulé depuis le dernier événement de S.*	*Type Time*
S'ACTIVE$[(t)]$	*Vrai dès que S est actif.*	*Type Booléenne*
S'LAST_ACTIVET$[(t)]$	*Temps écoulé depuis l'activation de S.*	*Type Time*
S'LAST_VALUE$[(t)]$	*Valeur précédent le dernier changement du signal S.*	*Type de S*

7 Travail Dirigé

7.1 *INTRODUCTION*

L'objectif du Travail Dirigé est d'enseigner la programmation d'un langage complexe nommé *VHDL*. On sait qu'il y a deux de façon d'aborder un langage :

$\Rightarrow a$. Prendre le manuel de référence du langage pour comprendre ;

$\Rightarrow b$. Aborder un langage par analogie de façon intuitive à partir de schémas et d'exemples.

Pour conclure, indépendamment de l'option a choisi, l'apprentissage d'un langage demande de temps devant l'ordinateur et de patience. Parce que il est impossible d'apprendre la programmation du langage à partir seulement de concepts. Donc, la pratique est nécessaire.

7.2 *TRAVAIL DIRIGÉ DE VHDL*

TD 7.2.1

On considère le programme du Programme 7.2.1 :

$\Rightarrow$ *Faire le diagramme de temps.*

Programme 7.2.1 ***L'entité et l'architecture de PEX01.***

```
        library IEEE;
        use IEEE.std_logic_1164.all;
        use IEEE.std_logic_arith.all;
        use IEEE.std_logic_unsigned.all;
        entity ENTITY_PEX01 is
          port ( CLK       : in  STD_LOGIC;
                 ENABLE    : in  STD_LOGIC;
                 DIN1      : in  STD_LOGIC;
                 DIN2      : in  STD_LOGIC;
                 DOUT      : out STD_LOGIC;
                 DOUTC     : out STD_LOGIC;
                 RESULTAT  : out STD_LOGIC);
          end ENTITY_PEX01;
          architecture ARCH_PEX01 of ENTITY_PEX01 is
          signal test :  STD_LOGIC;
          begin
          schem : process (CLK)
           begin
             if CLK'event and CLK='1' then
                 if ENABLE='1' then
                   DOUT  <= DIN1 xor DIN2;
                   DOUTC <= DIN1 AND DIN2;
                   test  <= NOT DIN2;
                 else
                   DOUT  <= DIN1 AND DIN2;
                   DOUTC <= DIN1 xor DIN2;
                   test  <= DIN1;
                 end if;
                end if;
           end process schem;
           RESULTAT <= test XOR ENABLE XOR DIN1;
    end ARCH_PEX01;
```

Copyright © by J. L. Mayorquim 11 Fevrier 2002 - 12h46

TD 7.2.2

On considère le programme de le Programme 7.2.2 :

⇒ a. *Déduire de ce programme, par une construction méthodique, un schéma (portes logiques) ;*

⇒ b. *Faire un diagramme de temps de l'architecture.*

Programme 7.2.2 ***Déclaration d'entité et corps de l'architecture.***

```
LIBRARY IEEE;
USEE IEEE.std_logic_1164.all;
ENTITY PEX02 IS
    PORT ( In_A  : in  STD_LOGIC;
           In_B  : in  STD_LOGIC;
           In_C  : in  STD_LOGIC;
           In_D  : in  STD_LOGIC;
           In_E  : in  STD_LOGIC;
           S1    : out STD_LOGIC;
           S2    : out STD_LOGIC;
           S3    : out STD_LOGIC);
    END PEX02;
    ARCHITECTURE PEX02_arch of PEX02 IS
    SIGNAL sg01, sg02, sg03, sg04 : STD_LOGIC;
    SIGNAL sg05, sg06, sg07, sg08 : STD_LOGIC;
     BEGIN
        sg01 <= In_A XOR In_C;
        sg02 <= In_C NAND (NOT In_B);
        sg03 <= (NOT In_B) NAND In_A;
        S1   <= NOT ((sg01 AND sg02)) NAND sg03;
        sg04 <= (In_A NAND In_B) XOR In_D;
        sg05 <= (NOT In_A) AND ((NOT In_B) NOR In_D);
        S2   <= sg04 NAND sg05;
        sg06 <= (NOT In_A) NOR In_B;
        sg07 <= (NOT In_A) NAND In_E;
        sg08 <= In_B XOR In_E;
        S3   <= sg06 NAND (sg07 OR sg08);
    END PEX02_arch;
```

TD 7.2.3

Développer un decodeur 3 à 8 synchrone avec enable.

TD 7.2.4

Développer un encodeur 8 à 3 synchrone avec enable.

TD 7.2.5

Développer le composant buffer *trois états SN74S244N et utiliser l'instruction* bloc.

TD 7.2.6

Développer le composant SN74ALS85N qui peut comparer la magnitude de 4 bit.

TD 7.2.7

Développer le composant SN74ALS157N qui est un MUX *2_ligne_pour_1_ligne.*

TD 7.2.8

Décrire par VHDL *l'opération de soustraction avec les portes logiques* ET, Complément, OU *et* XOR.

TD 7.2.9

Développer le composant SN74ALS112N qui est une BASCULE JK.

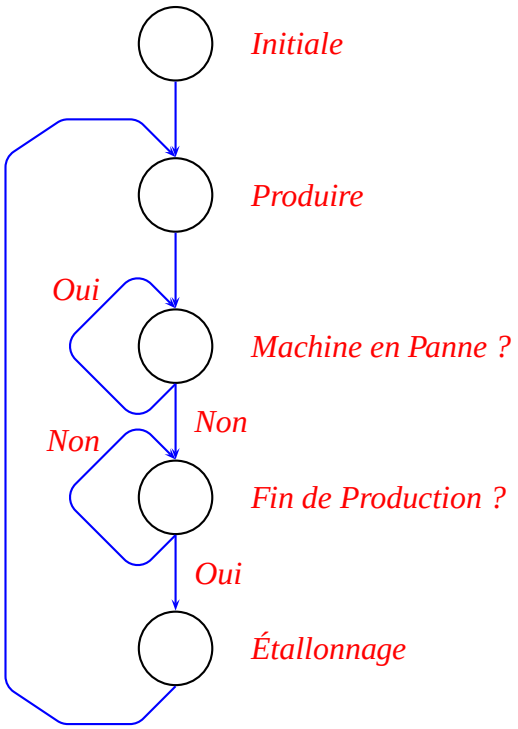
TD 7.2.10

Développer un compteur de 4_bit synchronises avec enable, reset asynchrone et load synchrone SN74ALS161N.

TD 7.2.11

Développer en VHDL *le circuit de contrôle présenté sur la Figure 7.1.*

Figure 7.1 ***Unité de production.***



GLOSSAIRE

Accès (access type)

Voir Type accès.

Accessible en lecture (readable)

Un signal ou une variable sont dits *accessibles en lecture* s'ils peuvent apparaître à droite d'une affectation. Les paramètres de sortie de procédure (mode out) ne sont pas accessibles en lecture.

Affectation (assignment)

C'est l'instruction qui permet de modifier la valeur d'une variable, d'un signal ou d'initialiser des constantes. Pour les constantes et variables le symbole en est $:=$ tandis que pour les signaux, il faut utiliser $<=$.

Affectation gardée (guarded assignment)

C'est une affectation de signal conditionnée par la valeur d'un signal *GUARD* de type booléen. Elle s'utilise principalement dans un bloc gardé (auquel cas le signal *GUARD* est la condition de garde du bloc) et se note par l'utilisation du mot clé guarded.

Alias (alias)

Déclarer un alias sert à donner un nom différent à un objet. Le nom original continue à être utilisable. Le nouveau nom peut désigner une structure différente de celle désignée par le nom original: il est possible de renommer, par exemple, un bit particulier d'un vecteur de bits.

Allocateur (allocator)

Voir Allocation.

Allocation (allocation)

C'est l'opération qui permet d'allouer la place mémoire nécessaire pour stocker les objets pointés: l'instruction particulière new permet de réserver une nouvelle zone mémoire. Cette allocation peut aussi être utilisée pour initialiser la valeur

de l'objet pointé. L'opération inverse de l'allocation est la libération.

Architecture (architecture)

C'est une des cinq unités de conception *VHDL*. Elle se compile séparément. Une architecture décrit la vue interne d'une entité. C'est là qu'est décrit soit le comportement, soit la structure du modèle. Une spécification d'entité peut avoir plusieurs architectures associées, chacune de ces architectures correspond généralement à un niveau de finesse de description différent.

Article (record)

C'est une famille de types qui a la particularité de réunir des éléments de types éventuellement différents. Ces éléments sont désignés par un nom et sont appelés *champs* de l'article. Avec les tableaux, les articles font partie de la famille des types composites.

Assertion (assert statement)

C'est une instruction, existant sous forme d'instruction séquentielle ou concurrente, qui permet d'envoyer un message donné si une condition indiquée n'est pas réalisée.

Association par nom (named association)

Une association de paramètres de sous-programmes, de paramètres génériques ou de ports est dite *faite par nom* si le nom du paramètre ou port formel apparaît explicitement. L'association peut aussi se faire de manière positionnelle ou même mixte (positionnelle et par nom) si l'association positionnelle est en tête.

Association positionnelle

Une association de paramètres de sous-programmes, de paramètres génériques ou de ports est dite *positionnelle* si le nom du paramètre ou port formel n'apparaît pas. L'association est faite en suivant l'ordre d'écriture des paramètres ou des ports. L'association peut aussi se faire par nom ou même mixte (positionnelle et par nom) si l'association positionnelle est en tête.

Attribut (attribute)

C'est un ensemble de propriétés d'un type, d'un label, d'une variable ou d'une entité. Il existe deux sortes d'attributs : ceux qui sont predefines, et ceux définis par le concepteur. Leur référence se fait par le caractère: ' (nommé apostrophe, quote, tick ....) suivi du nom de l'attribut. La valeur d'un attribut défini par le concepteur est calculée à l'élaboration et constante ensuite.

Bibliothèque (library)

C'est l'endroit où le compilateur stocke les unités de conception qu'il a compilées et qui sont donc exactes du point de vue du langage. Il est possible de référencer (c'est-à-dire de *voir,* d'avoir accès) le contenu d'une bibliothèque en utilisant une clause de la bibliothèque.

Bibliothèque de ressources (resource library)

Des unités de conception appartenant aux bibliothèques ressources peuvent être référencées. Pour cela, il est nécessaire que la bibliothèque dans laquelle elles se trouvent sont, soit référencée par une clause library. Cette clause doit précéder l'unité de conception qui se sert de la ressource. Le nombre de ces bibliothèques n'est pas limité par la norme.

Bibliothèque de travail (working library)

A un instant donné, une seule bibliothèque est *bibliothèque de travail*, les autres sont des *bibliothèques de ressources.* Le choix de la bibliothèque de travail est fait hors langage par une commande liée à la plate-forme *VHDL* utilisée. Elle peut être référencée en *VHDL* par le nom logique *WORK*. Concrètement, c'est dans cette bibliothèque que seront rangées les unités de conception compilées avec succès.

Bloc (block)

C'est le support de la hiérarchie en *VHDL*. C'est aussi un moyen d'encapsuler des déclarations et de garder des affectations de signal (bloc gardé). Un bloc peut être ouvert partout où l'on peut mettre une instruction concurrente (c'en est d'ailleurs une). Le bloc (une architecture est un bloc) et le composant sont les deux moyens de hiérarchiser une description structurelle en *VHDL*.

Bloc externe (external block)

C'est une entité, c'est-à-dire un couple spécification d'entité/architecture.

Bloc gardé (guarded block)

La condition de garde d'un bloc est une expression booléenne donnée en tête du bloc. Les affectations gardées intérieures à ce bloc ne seront exécutées que si la condition de garde est vraie.

Bloc générique (generic block)

Un bloc générique est un bloc paramètre (on parlera paramètres génériques). De l'intérieur du bloc, ces paramètres sont vus comme des constantes et peuvent être manipulés comme telles. La possibilité d'utiliser des blocs génériques (souvent

des entités) est extrêmement utilisée en *VHDL*. Les bibliothèques de modèles sont très souvent génériques pour être générales.

Boîte (<>, box)

C'est une notation qui permet de déclarer un tableau non contraint, c'est-à-dire sans préciser l'intervalle de variation des indices et donc la taille du tableau. Celle-ci sera connue plus tard, lors de l'exécution.

Changement de nom (renaming)

Voir Alias et Sous-type.

Classe d'objet (class of object)

Il existe trois classes d'objets en *VHDL*: ce sont les constantes, les variables, les signaux.

Clause (clause)

Une clause est une forme de spécification. Il existe deux clauses en *VHDL*: use et library. Elles servent à indiquer que l'on veut pouvoir accéder à des objets contenus dans des unités de conception (clause use) ou bibliothèques (clause library).

Commentaires (commente)

Les commentaires sont des informations destinées à améliorer la lisibilité d'un programme. Ils sont ignorés par l'analyseur. En *VHDL*, les commentaires commencent par deux tirets – et se finissent avec la ligne. Il est inutile de rappeler le grand intérêt des commentaires pour la maintenabilité d'une description : no comment.

Compatibilité

Voir types compatibles.

Composant (component)

C'est l'idée que l'on se fait d'une entité (spécification d'entité/ architecture) qui servira plus tard à *instancier* des composants réels mais qui peut très bien ne pas (encore) exister. Comme le bloc, c'est un élément de la structuration en *VHDL*.

Concurrence (concurrency)

Les concepts de concurrence et de parallélisme sont souvent confondus. Des instructions parallèles s'exécutent en fonction de priorités données par le système,

sans autre point de synchronisation que ceux que l'on programme explicitement (rendez-vous, sémaphores ...). Le fond de ce mécanisme est non déterministe et la notion de temps vrai est sous-jacente. Pour des instructions concurrentes, l'ordre des exécutions est indifférent. C'est ce qui permet de mettre en oeuvre un parallélisme effectif, mais la notion de temps vrai a disparu.

Condition de garde d'un bloc (guard expression)

C'est une expression booléenne, qui, indiquée en tête d'un bloc (le bloc est alors dit bloc gardé), conditionne les affectations de signaux internes à ce bloc.

Configuration (configuration)

C'est une des cinq unités de conception *VHDL*. Elle sert à indiquer la correspondance entre une instance de composant et un modèle (un couple spécification d'entité/ architecture). Les correspondances entre les ports du composant et ceux du modèle, ainsi qu'entre paramètres génériques locaux et formels peuvent se faire dans cette unité de conception.

Constante (constant)

Les constantes constituent une des trois classes d'objets VHDL Initialisées à une valeur, elles ne peuvent plus être modifiées par la suite. La valeur d'initialisation doit être du même type que celui de la constante. Elle sera calculée à l'exécution et peut être une expression complexe, incluant éventuellement des appels de fonctions.

Constante à valeur différée (deferisico constant)

Une déclaration de constante sans préciser sa valeur se nommera *déclaration de constante à valeur différée*. En pratique, ce type de déclaration de constantes se trouve restreint à la partie spécification d'un paquetage. De l'extérieur du paquetage, la constante peut être utilisée, mais sans voir sa valeur qui est définie par une déclaration complète de constante à l'intérieur du corps du paquetage. C'est un exemple de masquage d'information en *VHDL*.

Conversion de types (type conversion)

Il est possible d'effectuer une conversion explicite entre deux types de structures proches. Cette conversion est très restrictive et n'est autorisée que pour les types entiers ou flottants et les tableaux de même dimension ayant des types d'éléments et des index convertibles. Un type peut toujours être converti en lui-même.

Corps de paquetage (package body)

C'est une des cinq unités de conception *VHDL*. Elle se compile séparément. C'est là où sont écrits les algorithmes des sous-programmes exportés dans la spécification du paquetage. On peut y trouver d'autres sous-programmes ou objets d'usage interne à ce paquetage et qui ne sont pas accessibles de l'extérieur.

Cycle de conception VHDL

Le cycle de conception d'une session *VHDL* est : compilation, élaboration, exécution et exploitation. La compilation s'occupe de tous les aspects statiques de la description, l'élaboration de tous ses aspects paramétrables. L'exécution, selon l'application, simule, synthétise, prouve ... et l'exploitation retrouve le concepteur comme interlocuteur.

Déclaration (declaration)

Une déclaration associe un nom ou identificateur à un objet du langage, qu'il soit une variable, un signal, une constante, un type, un sous-programme, un composant, etc... *VHDL* est un langage où, à de très rares exceptions près (les labels par exemple), on doit déclarer toute chose avant de l'utiliser.

Déclaration complète (full declaration)

Lors de leur déclaration, la plupart des objets sont déclarés complètement. Deux exceptions fréquentes sont les déclarations incomplètes de type (souvent pour les structures récursives) et les constantes à valeurs différées (pour le masquage d'information). Dans ces deux cas, les déclarations classiques de ces objets sont obligatoires et seront faites ultérieurement: elles portent le nom de *déclarations complètes*.

Déclaration de constante à valeur différée (deferisico constant declaration)

Voir constante à valeur différée

Déclaration de sous-programme (subprogram declaration)

Aussi appelée spécification de sous-programme. cette déclaration se trouve souvent dans la partie spécification de paquetage. Elle permet de définir toutes les informations utiles à l'appel du sous-programme: son profil.

Déclaration incomplète de type (incomplete type declaration)

En *VHDL*, tout objet utilisé doit être préalablement déclaré. Cette contrainte peut être critique dans le cas de la déclaration de structures de données récursives, elles s'utilisent elles-mêmes avant d'être complètement définies. La déclaration incomplète de type permet de résoudre ce problème. Le nom du type est donné, mais pas sa structure. Il est alors possible d'utiliser ce type dans une structure de donnée et de ne le définir complètement que plus tard. Cette déclaration tardive mais complète sera exigée par le compilateur.

Délai delta (delta delay)

C'est le délai utilisé par le simulateur pour s'allouer à chaque pas de simulation un nombre variable de tranches infinitésimales de temps et ainsi gérer la succession des affectations de signaux. Il a deux caractéristiques essentielles: il n'est pas nul et le cumul de délais *delta* ne peut jamais atteindre ne serait ce qu'une femtoseconde. Les deux dimensions du temps en *VHDL* sont le temps *réel*, qui se compte en pas de simulation et en unités de type *TIME* (c'est ce temps que voit le concepteur) et le temps *delta*, qui est géré par le simulateur. Le temps *delta* est le reflet de la causalité.

Description flot de données (data-flow description)

C'est une description qui montre les flots de données sortants en fonction des entrants d'un modèle sans se préoccuper de sa structure. Une description *flot de données*, qui n'est qu'un raccourci d'écriture de description comportementale, utilise principalement l'affectation concurrente de signal et l'appel de fonctions pour modéliser.

Description comportementale (behavioural description)

La description de niveau comportemental s'attache à décrire le fonctionnement d'un modèle sans se soucier d'un éventuel découpage proche de la réalisation, donc de la structure. La description prend la forme d'un algorithme. En langage *VHDL*, une description purement comportementale sera un processus au sens général (l'appel d'une procédure concurrente ou la description de niveau flot de données ayant chacun leur traduction en terme de processus) dans lequel se trouvera une algorithmique du genre de celle que l'on utilise dans les langage de programmation classiques. Ce niveau de description, bien que présent dans les *feuilles* de la hiérarchie, peut aussi s'adresser aux concepteurs désirant modéliser un système avec un haut niveau d'abstraction. Le temps peut y intervenir.

Description fonctionnelle (functionnal description)

Voir description comportementale.

Description hybride

C'est une description mélangeant au moins deux des trois genres de descriptions: comportementale, flot de données et structurelle.

Description structurelle (structural description)

C'est le niveau de description le plus simple à appréhender. Il consiste à décrire le modèle par sa structuration, c'est-à-dire par un ensemble d'éléments interconnectés. En langage *VHDL*, une telle description est constituée de déclarations et d'instances de composants. Deux propriétés se déduisent immédiatement: ce niveau de description ne fait pas intervenir le temps et ne peut, en aucun cas, constituer une *feuille* (élément terminal dans la hiérarchie de description) au moment de la simulation. Une *feuille* fera nécessairement l'objet d'une description comportementale ou *flot de données*.

Echéancier

C'est l'agenda du simulateur. Sur la base des événements de la date courante, il y inscrit des événements aux dates futures.

Effet de bord

Se dit d'une modification de l'environnement par un autre moyen que les paramètres de sortie de procédures, les signaux de mode out des entités ou les valeurs de retour des fonctions. Par exemple, l'affectation d'un signal global (non déclaré localement mais dans un paquetage) est un effet de bord.

Elaboration (elaboration)

L'élaboration est la phase d'initialisation de *VHDL* qui consiste à créer les objets qui seront peut-être modifiés par la suite. Voir Cycle de conception *VHDL*.

Elément d'onde (waveform element)

C'est un couple valeur/délai séparé par le mot clé after. Un élément d'onde apparaît dans la partie droite d'une affectation de signal. La valeur est la valeur prévue du signal à la date correspondant à l'instant présent plus le délai. Séparés par des virgules, les éléments d'onde constituent une forme d'onde.

En-tête de sous-programme (subprogram header)

C'est par cela que commence un sous-programme. Ces informations, très proches de celles de la déclaration de sous–programme constituent son profit.

Encapsuler, Encapsulation

C'est le fait de découper en vue externe et vue interne une notion. En *VHDL*, les bibliothèques, les paquetages, les entités, les processus, les blocs et les sous-programmes sont autant de moyens d'encapsuler l'information. Cette encapsulation est une forme de masquage d'information.

Entité (design entity)

C'est un couple spécification d'entité/ architecture, un modèle. Il peut y avoir plusieurs architectures pour la même spécification d'entité. Lors de la simulation, ce sont des entités que l'on simule (une seule architecture par spécification d'entité).

Evénement sur un signal (event)

Un évènement se produit sur un signal si celui-ci change de valeur. La notion d'évènement est à distinguer de la notion de transaction.

Exécution (execution)

Voir Cycle de conception *VHDL*.

Exploitation

Voir Cycle de conception *VHDL*.

Exporter

Les sous-programmes, types et autres objets figurant dans une spécification de paquetage sont dits *exportés par ce paquetage*. Ils sont visibles et utilisables par le monde extérieur au paquetage.

Expression (expression)

Une expression permet d'indiquer la façon de calculer une valeur. Une constante ou un appel de fonction sont des exemples d'expression.

Expression qualifiée (qualifed expression)

Voir Qualification d'expression.

Fichier (file)

Le type fichier (mot clé file) permet d'avoir accès aux fichiers du système sur lequel est installée la plate-forme *VHDL*. Leur utilisation est courante : lecture du contenu d'une *ROM*, écriture de trace, lecture de stimuli, sortie de résultats...

Flot de données (data flow)

Ensemble des informations transitant sur les ports ou signaux internes d'un modèle.

Fonction (function)

Les fonctions sont un des deux genres de sous-programmes en *VHDL*. Elles rendent un résultat (et un seul) et n'agissent pas par effet de bord.

Fonction de résolution (resolution function)

Quand la valeur d'un signal peut provenir de plusieurs sources (collecteurs ouverts, bus trois-états ...), il faut savoir gérer les conflits. On demandera donc à un signal multi-sources de posséder une fonction de résolution. C'est un mécanisme qu'offre *VHDL* et qui permet de résoudre les conflits susceptibles de survenir et de calculer la valeur résultante (on dit valeur résolue) du signal. Un signal possédant une fonction de résolution est dit un signal résolu et cette fonction de résolution est appelée par le simulateur pour calculer sa valeur.

Format intermédiaire (interchange format)

Le résultat de la compilation est appelé *format intermédiair*. De nombreuses discussions sur l'intérêt de normaliser un format intermédiaire ont abouti à la création du groupe de normalisation *VIFASG*. Il est décidé de définir un jeu de sous-programmes permettant le stockage et la relecture de la sémantique d'un texte *VHDL*. Le format physique sous-jacent n'entre pas dans le cadre de cette normalisation.

Forme d'onde (waveform)

Dans une affectation de signal, les différents champs (les couples valeur/délai séparés par le mot clé after) sont appelés des éléments d'onde. Séparés par des virgules, ils constituent une forme d'onde, c'est-à-dire une suite de valeurs prévues du signal à des dates données.

Généricité (genericity)

Voir Bloc générique.

Glossaire (glossary)

C'est une sorte de dictionnaire qui peut être récursif, la preuve...

Groupe de normalisation (standardization groups)

Voir VASG,VDEG,VIFASG,WAVES.

Hold

Voir Temps de maintien

Identificateur (identifier)

C'est le nom d'un objet. L'association d'un nom à un objet se fait lors de la déclaration de cet objet. Un identificateur est une suite de lettres (jeu de caractères *ASCII*), de chiffres et de traits bas (_) commençant par une lettre et qui n'est pas un mot clé.

Inertiel (inertial delay)

Il s'agit d'un mode d'affectation des signaux en *VHDL*. Il filtre les impulsions dont la durée est inférieure au délai de l'affectation qui est indiqué par le mot clé after. C'est ce mode qui est pris par défaut à chaque affectation de signal. L'autre mode s'appelle transport. La figure 8.4 visualise les effets de ces deux modes.

Initialisation (initialization phase)

Cette phase est la première de l'exécution. Tous les processus sont lancés au moins une fois pendant cette phase.

Instance (instantiation)

Une instance de composant est une occurrence du modèle de ce composant. A partir d'un modèle d'inverseur, on peut créer (instancier) plusieurs inverseurs *réels* (*INV1, INV2 et INV3* par exemple) ayant chacun leurs propres ports mais les caractéristiques de leur modèle père. *INV1, INV2 et INV3* sont les instances (on dit aussi instanciations) du composant inverseur.

Instanciation (instantiation)

Voir Instance.

Instancier (to instantiate)

C'est l'action de créer une instance (ou instanciation).

Instruction concurrente (concurrent statement)

Un système matériel se décrit naturellement de façon concurrente. Certaines parties gèrent des accès disque, d'autres des accès mémoire et beaucoup des accès à des bus. Tous ces fonctionnements se passent *en même temps* pour l'esprit humain qui fait la description, c'est cela la notion de *concurrence*. *VHDL* possède tout un jeu d'instructions concurrentes, le chapitre 9 leur est consacré.

Instruction séquentielle (sequential staternent)

Dans une description de matériel, certaines choses se décrivent naturellement de manière séquentielle. Un protocole de lecture d'une mémoire s'exprime par exemple: positionner tel signal, puis tel autre, attendre tant de temps puis faire.... VHDL dispose de tout un jeu d'instructions séquentielles utilisables dans un processus ou dans le corps d'un sous-programme.

Interopérabilité (interoperability)

Peu français, ce terme désigne la faculté de pouvoir réunir au sein d'une même modélisation, des descriptions de composants fournies par des constructeurs différents.

Itération (iteration)

Une boucle (mot clé loop) sert à répéter la séquence d'instructions qui s'y trouve. Chaque déroulement s'appelle une itération.

Label (label)

Certaines instructions peuvent avoir un nom. Ce nom s'appelle un label. Un label est même obligatoire pour certaines instructions concurrentes. Libération (deallocation) Lorsqu'un objet pointé par un type accès n'est plus utile, il est possible de récupérer la zone mémoire qu'il occupe. Cette opération s'appelle une libération mémoire et se fait par appel à la procédure *DEALLOCATE*, déclarée automatiquement pour tout objet pointé. On parle de libérer un pointeur.

Limite de résolution secondaire (secondary unit as resolution limit)

Tout type physique possède une unité de base. Pourtant, si dans une unité de conception, l'unité physique utilisée est très grande (les minutes pour le type *TIME* par exemple), le codage peut poser un problème (surtout s'il est effectué sur 32 bits). La norme admet alors le codage à partir d'une limite de résolution secondaire: c'est l'unité physique la plus petite utilisée dans cette unité de conception qui sert alors d'unité de base. Cette unité secondaire de résolution peut être à l'origine d'incohérences entre plusieurs unités de conception se

partageant (ou s'échangeant) un même type physique.

Liste de sensibilité d'un processus (sensitivity list)

C'est une liste de signaux. Tout événement sur un signal provoque l'activation des processus dans la liste de sensibilité desquels il apparaît.

Littéral (literal)

Les littéraux sont les valeurs *dures* du langage : par exemple 123 est un littéral. Les littéraux sont classés en numériques (123, 3.14), énumérés (*TRUE* en est un), chaînes de caractères (*en voici une*) ou de bits ("101"), et null qui est un littéral à lui tout seul, et qui représente la valeur de tout objet de type accès non initialisé.

LRM

C'est de l'anglais, cela signifie *Language Reference Manual* c'est-à-dire *manuel de référence du langage*. Aussi appelé *La norme*, ce document dûment normalisé par *IEEE* contient (théoriquement) toutes les informations relatives à la syntaxe et à la sémantique de VHDL. Destiné avant tout aux constructeurs de compilateurs, ce n'est pas un *manuel utilisateur* et sa lecture est très indigeste pour un néophyte. Cet ouvrage reste néanmoins une référence pour tous les problèmes pointus pouvant survenir lors de modélisations.

Manuel de référence (language reference manual)

Voir *LRM*.

Masquage d'information

C'est un vieux principe de truand qui veut que *moins on en sait, mieux on se porte*. Ce principe est érigé en dogme en *VHDL*. Il consiste à ne donner à voir au concepteur que ce dont il a besoin, en lui en cachant la réalisation. Ainsi, le concepteur n'est ni submergé de détails inutiles, ni soumis à la tentation de modifier ce qui a été fait (et testé) par quelqu'un d'autre, ni à celle d'utiliser les dites informations pour se livrer à du *reverse-engineering*.

Mode d'affectation des signaux

Il existe deux modes d'affectation de signaux différents, le mode inertiel (par défaut) et le mode transport.

Mode inertiel (inertial delay)

Voir Inertiel.

Mode transport (transport delay)

Voir transport.

Modèle (model)

Dans cet ouvrage, modèle désigne un couple spécification d'entité/architecture: une entité. La spécification d'entité est la vue externe du modèle (ce que l'on voit de l'extérieur) et l'architecture est la vue interne (comment marche-t-il ?).

Modèle inertiel (inertial delay)

Voir Inertiel.

Modèle transport (transport delay)

Voir transport.

Mot clé (key word)

Ce sont des mots réservés du langage. On ne peut les utiliser comme identificateurs. Il y en a 82 et il vaut mieux les connaître. Signe particulier: ils sont tous américains.

Niveau de sévérité (severity level)

Le niveau de sévérité de l'erreur d'une instruction d'assertion est une expression du type *SEVERITY-LEVEL*. Ce type est défini dans le paquetage *STANDARD* (détaillé au chapitre 12), et donc connu par défaut. C'est un type énuméré défini par: *type SEVERITY-LEVEL is (NOTE, WARNING, ERROR, FAURE)*;

Niveaux de Description

Voir descriptions structurelle, comportementale, ftot de données et hybride.

Norme VHDL 7.2 (VHDL version 7.2)

Voir *VHDL* 7.2.

Notation basée (based literal)

C'est une notation qui permet d'exprimer les littéraux dans une base autre que la base dix, prise par défaut. Les bases possibles vont de deux à seize.

Notation pointée (selected name)

S'il est visible, un objet se dénote soit directement par son nom, soit plus laborieusement sous forme d'une notation pointée qui traduit le *chemin* pour

l'atteindre.

Objet (object)

D'une manière générale, tout ce qui a un nom en *VHDL* est un objet. Cela inclut tout ce qui explicitement se déclare, plus les labels. Il arrive pourtant de restreindre ce terme aux trois classes d'objets: constantes, variables, signaux.

Objet pointé (object designated by the access value)

C'est un objet qui est créé dynamiquement (à l'exécution) lors de l'allocation. Il disparaît par une opération de libération mémoire.

Opérateur (operator)

Il existe six classes d'opérateurs *VHDL* et un niveau de priorité unique est attribué à chaque classe. Par ordre de priorité croissante, on trouve: les opérateurs logiques, les opérateurs relationnels, les opérateurs d'addition, les opérateurs de signe, les opérateurs de multiplication et les autres. Comme les fonctions, les opérateurs peuvent faire l'objet de surcharges.

Paquetage (package)

C'est le moyen d'encapsuler des objets et des sous-programmes. Il est constitué d'un couple spécification de paquetage/corps de paquetage. Le corps de paquetage est optionnel mais s'il existe, il est unique.

Paramètre d'entrée de sous-programme (parameter of mode in)

C'est un objet transmis à un sous-programme qui ne doit qu'être consulté (lu) et ne peut être modifié (écrit). Le mode in permet de préciser cet état de fait. Le compilateur vérifiera les restrictions d'emploi qui en découlent.

Paramètre de sortie de procédure (parameter of mode out)

C'est une valeur rendue par une procédure. Le mot clé out permet de spécifier ce mode de passage qui n'est qu'un retour.

Paramètre effectif d'un sous-programme (actual parameter of a subprogram)

C'est le paramètre qui est associé au paramètre formel d'un sous-programme lors de l'appel. L'association se fait par position ou par nom du paramètre formel.

Paramètre formel de sous-programme (formal parameter of a subprogram)

C'est le paramètre tel qu'il est donné dans la spécification du sous-programme. C'est ce nom qui sera utilisé lors de l'appel du sous-programme en association par nom.

Paramètre générique effectif (actual generic parameter)

C'est le paramètre avec lequel sera instanciée l'unité de conception générique. L'association du paramètre générique effectif au paramètre générique formel se fait lors de l'instance.

Paramètre générique formel (format generic)

C'est avec ce paramètre que s'écrit l'unité de conception générique. Lors de l'instance de cette description, ce paramètre sera remplacé par la valeur du paramètre générique effectif.

Pas de simulation (step of the simulation cycle)

C'est un instant du temps vrai. Il correspond à une date unique mais peut être divisé en autant de *délais - delta* que nécessaire par le simulateur pour effectuer la succession des instructions concurrentes de cet instant. Phase d'initialisation (initialization phase) voir Initialisation.

Pilote d'un signal (driver)

C'est une caractéristique qui est associée à chaque signal et qui contient sa valeur courante et la liste des valeurs prévues et leurs dates. Un signal n'a qu'un pilote par processus où il apparaît, mais il peut apparaître dans plusieurs processus. Dans ce cas, ce doit être un signal résolu.

Plan mémoire

Une *ROM* ou une *RAM* sont des systèmes comprenant des blocs de contrôle et un bloc où est mémorisé l'information. C'est ce bloc qui est appelé plan mémoire.

Plate-forme VHDL

C'est l'ensemble de l'environnement de développement et de test des modélisations VHDL. Cela comprend généralement un simulateur mais peut aussi s'étendre à un éditeur *langage* textuel, à des éditeurs graphiques (pour entrée de descriptions structurelles) et à tout autre outil du monde *VHDL*.

Pointeur (access type)

Voir 71jpe accès.

Polymorphisme

C'est le moyen (en informatique) de présenter la même information de différentes manières. Association de sous-éléments et conversion de paramètres sont deux façons de pratiquer le polymorphisme en *VHDL*. Ce ne sont pas les seules, les alias par exemple en constituent une autre forme.

Port (port)

Les ports sont les points d'entrées/ sorties des entités ou des composants. Un port est caractérisé par le sens et le type des données qui y transitent.

Port effectif (actual port)

C'est un port *réel* associé à un port formel.

Port formel (formal port)

C'est le nom du port tel qu'il apparaît dans une spécification d'entité. C'est ce nom qui sera utilisé dans le cas d'une association par nom lors de la configuration.

Procédure (procedure)

La procédure est un des deux genres de sous-programmes *VHDL*. Elle peut agir par effet de bord. Elle modifie aussi (éventuellement) la valeur des paramètres transmis à l'appel.

Processus (process)

C'est la base de la concurrence en *VHDL*. Toute instruction concurrente peut être traduite par un processus, c'est le processus équivalent. Un processus ne contient que des instructions séquentielles. Un processus vit toujours arrivé $\grave{a}$ sa fin (mots clés end processus), il s'exécute $\grave{a}$ nouveau depuis son mot clé de début begin. Il est possible, par la spécification d'une liste de sensibilité (des signaux) ou par une instruction wait on, de l'endormir en attendant des événements sur les signaux indiqués.

Processus passif (passive process)

Sont appelés passifs, les processus où aucun signal n'apparaît à gauche d'une affectation. Ces processus sont dits passifs, non parce qu'ils ne s'exécutent pas, mais parce que leur exécution ne va pas entraîner l'exécution d'autres processus. L'instruction concurrente d'assertion et, sous certaines conditions

(pas de paramètres de mode out ni inout), l'appel concurrent de procédures ont des processus équivalents passifs.

Profil (parameter and result type profile)

Le profil d'un sous-programme est un ensemble d'informations qui comprend le nombre, l'ordre et le type des paramètres formels ainsi que, pour une fonction, le type du résultat rendu. Le fait de pouvoir surcharger deux sous-programmes de profils différents est particulièrement pratique et accroît la facilité d'écriture tout en assurant une bonne lisibilité des programmes.

Processus passif (passive process)

Sont appelés passifs, les processus où aucun signal n'apparaît à gauche d'une affectation. Ces processus sont dits passifs, non parce qu'ils ne s'exécutent pas, mais parce que leur exécution ne va pas entraîner l'exécution d'autres processus. L'instruction concurrente d'assertion et, sous certaines conditions (pas de paramètres de mode out ni inout), l'appel concurrent de procédures ont des processus équivalents passifs.

Profil (parameter and result type profile)

Le profil d'un sous-programme est un ensemble d'informations qui comprend le nombre, l'ordre et le type des paramètres formels ainsi que, pour une fonction, le type du résultat rendu. Le fait de pouvoir surcharger deux sous-programmes de profils différents est particulièrement pratique et accroît la facilité d'écriture tout en assurant une bonne lisibilité des programmes.

Qualification d'expression (qualified expression)

La qualification d'expression permet de lever une ambiguïté sur le type d'une expression ou d'un agrégat en indiquant explicitement le nom de ce type.

Rationale

C'est un document édité par le *VASG*. Il explique en détail les raisons des choix qui ont conduit à la norme. Sa lecture (pour personne avertie uniquement) peut être très instructive.

Récursivité (recursivity)

La récursivité est la propriété qu'a un sous-programme de pouvoir s'appeler lui-même. Une structure de donnée s'appelant elle–même est aussi dite *récursive*. La récursivité croisée (le sous–programme A appelle le sous-programme B et réciproquement) est également autorisée en *VHDL*.

Copyright © by J. L. Mayorquim 11 Fevrier 2002 - 12h46

Repository

C'est un ensemble de programmes et de descriptions *VHDL* mis dans le domaine public. Gérée par le *VDEG*, cette banque d'informations permet de faciliter les échanges entre concepteurs *VHDL*.

Reverse-engineering

Procédé d'espionnage industriel tellement laid qu'il n'a pas de traduction en français.

Schéma d'itération (iteration scheme)

Le nombre de fois où sera répétée la séquence d'instructions contenue dans une boucle (mot clé loop) est régi par le schéma d'itération. Il existe deux schémas d'itération distincts correspondant aux mots clés while et for.

Sémantique

C'est le sens, la signification d'une notion. La sémantique complète la syntaxe qui n'est, elle, que la forme, la notation.

Set-up

Voir Temps de pré-positionnement.

Signal (signal)

Les signaux sont spécifiques des langages de description de matériel. Ils modélisent les informations qui passent sur les fils, les bus, ou d'une manière générale qui transitent entre les composants d'une description de matériel. Chacun d'eux possède un pilote (par processus où il apparaît) et est sujet à des transactions et des événements. Les signaux constituent une des trois classes d'objets *VHDL*.

Signal accessible en lecture (readable signal)

Voir Accessible en lecture.

Signal gardé (guarded signal)

C'est un signal qui, lors de sa déclaration, a été classé comme registre (mot clé *register*) ou bus (mot clé *bus*). Il ne peut être affecté que par une affectation gardée (mot clé *guarded*). Cette classification (*registre* ou *bus*) sert à décrire son comportement lors de sa déconnexion.

Signal interne (local signal)

C'est un signal déclaré à l'intérieur d'un bloc ou d'une architecture et qui n'est pas visible de l'extérieur de cette structure.

Signal multi-sources (multiple source signal)

C'est un signal qui est la cible de plusieurs affectations concurrentes de signaux. Un tel signal doit être un signal résolu.

Signal résolu (resolved signal)

Un signal possédant une fonction de résolution dans sa déclaration ou ayant un type en possédant une est dit résolu. Cela signifie qu'il peut avoir plusieurs sources, c'est-à-dire être la cible de plusieurs affectations concurrentes différentes. La fonction de résolution est là pour résoudre les conflits. Durant la simulation, cette fonction est appelée systématiquement pour calculer la valeur que va prendre le signal.

Signal statique (static signal name)

Un signal est statique si son nom est connu à la compilation de l'unité de conception dans laquelle il se trouve. Par exemple, le signal *TAB(1)*, où *TAB* est un tableau et 1 un paramètre de procédure n'est pas statique. Les signaux faisant partie de la liste de sensibilité d'un processus doivent être statiques.

Simulation (simulation)

Ce n'est qu'une des formes que peut prendre l'exécution, mais c'est la plus courante. Voir Cycle de conception *VHDL*.

Source d'un signal (source of a signal)

C'est tous les signaux apparaissant dans une partie droite d'une affectation concurrente de signal.

Sous-programme (subprogram)

Les sous-programmes permettent d'écrire du code réutilisable. Les valeurs des paramètres peuvent varier à chaque appel et donc donner des exécutions différentes. En *VHDL*, les sous-programmes sont principalement utilisés pour décrire une suite d'instructions séquentielles, un calcul, une conversion, des morceaux de processus ou des fonctions de résolution. Il existe deux genres de sous–programmes: les procédures (mot clé *procedure*) et les fonctions (mot clé *function*).

Sous-type (subtype)

C'est une restriction sur les valeurs d'un type ou d'un autre sous-type. Ces restrictions (ou contraintes) peuvent être dynamiques et donc calculées à l'exécution. Sans contrainte, le sous-typage peut servir à renommer un objet.

Sous-type dynamique

C'est un sous-type dont les contraintes sont calculées à l'exécution. C'est une des plus précieuses utilisations du sous-typage.

Spécification (specification)

Une spécification permet de donner des informations complémentaires sur un objet du langage.

Spécification d'entité (entity declaration)

C'est la vue externe d'une entité. On y trouve principalement la description des ports formels et des paramètres formels de généricité de ce modèle. Associée à une architecture, la spécification d'entité forme un modèle (aussi appelé entité). Il peut y avoir plusieurs architectures pour la même spécification d'entité, mais lors de l'exécution (la simulation), un seul couple spécification d'entité/architecture sera sélectionné pour chaque composant. La spécification d'entité est une des cinq unités de conception *VHDL*, elle se compile séparément.

Spécification de paquetage (package declaration)

La spécification de paquetage est une des cinq unités de conception *VHDL*, elle se compile séparément. C'est la vue externe d'un paquetage. On y déclare les objets et les sous-programmes que le paquetage exporte. L'algorithme réalisant ces sous-programmes est, lui, caché dans le corps du paquetage.

Spécification de sous-programme (subprogram declaration)

Voir Déclaration de sous-programme.

Stimulus, stimuli

C'est du latin, d'où le *i* au pluriel. En *VHDL*, un stimulus désigne un couple (valeur, date). Un stimulus est très proche d'un élément d'onde.

Surcharge (overloading)

Deux sous-programmes sont dits surchargés s'ils ont le même nom et que leurs profils diffèrent. Le fait de pouvoir surcharger deux sous-programmes de profils

différents est particulièrement pratique et augmente la facilité d'écriture et la lisibilité d'un programme. On parle aussi de surcharge de symboles de types énumérés pour indiquer que plusieurs types énumérés peuvent se partager le même symbole.

Systèmes matériels (hardware)

Ce sont indifféremment des cartes ou des circuits intégrés en électronique mais ce terme ne se restreint pas à ce domaine. Un carburateur de voiture ou un neurone, par exemple, sont des systèmes matériels et *VHDL* permet de les décrire.

Tableau (array)

C'est une famille de types qui a la particularité de réunir des éléments de type identique. On accède à ces éléments par un (ou plusieurs) indice. Alliés aux articles, les tableaux constituent la famille des types composites. Un tableau à une seule dimension (un seul indice) est appelé vecteur.

Tableau contraint (constrained array)

C'est un tableau dont l'intervalle de variation des indices (et donc la taille) est connu dès l'initialisation (à l'élaboration). Les mots clés *to* et *downto* permettent de préciser le sens de variation des indices.

Tableau non contraint (unconstrained array)

C'est un tableau où le symbole <> (que l'on prononcera box) en indice permet de repousser la définition d'un intervalle d'indiçage et d'une direction de variation à plus tard, lors de l'exécution. Deux exemples de tableaux non contraints sont les vecteurs de type *BIT* (type *BIT-VECTOR*) et les chaînes de caractères (type *STRING*).

Temps de hold

Voir Temps de maintien.

Temps de set-up

Voir Temps de pré-positionnement.

Temps de maintien d'un signal (hold)

Une exigence très courante en électronique est relative à la durée minimale de stabilité d'une donnée après un front. Ce temps minimal s'appelle temps de maintien.

Temps de pré-positionnement d'un signal (set-up)

C'est le délai de pré-positionnement d'un signal par rapport à un autre. Une vérification de pré-positionnement est tout à fait classique pour les éléments de mémorisation. Une donnée DIN devant être mémorisée sur le front montant d'un signal *CLK* se doit d'être stable pendant un certain délai de temps minimum.

Transaction (transaction)

Un transaction est créée à chaque fois que l'on calcule la valeur d'un signal, même si cette valeur calculée est la même que la valeur courante du signal (contrairement à l'événement).

Transport (transport)

Il s'agit d'un mode d'affectation des signaux en VHDL. Ce mode décrit un comportement à réponse fréquentielle infinie de l'affectation de signal. Ce mode transmet intégralement les impulsions, quelles que soient leurs durées. Il est indiqué par le mot clé transport après le symbole d'affectation. L'autre mode s'appelle inertiel et il est pris par défaut à chaque affectation de signal.

Type (type)

VHDL est un langage typé: tout objet du langage doit posséder un type avant d'être utilisé. Un type définit implicitement ou non, (type énuméré) l'ensemble des valeurs que peut prendre un objet, et l'ensemble des opérations disponibles sur cet objet. Ce type permet à l'analyseur/compilateur *VHDL* de faire un grand nombre de vérifications de cohérence.

Type accès (access type)

Un type accès (mot clé *access en VHDL*) est souvent aussi appelé pointeur. Un type accès pointe sur un objet de type précédemment défini. Il permet principalement de créer dynamiquement (c'est-à–dire, lors de l'exécution) des objets nouveaux. Des opérations d'allocation et de libération mémoire sont associées à tout type accès.

Type compatible

Deux types sont compatibles si tout objet appartenant à l'un d'eux peut être mis à la place d'un objet appartenant à l'autre sans erreur de compilation. Néanmoins, des vérifications dynamiques peuvent conduire à une erreur au moment de l'exécution s'il se trouve que la valeur effectivement prise par un objet n'est pas dans l'ensemble des valeurs autorisées pour le type attendu.

Type composite (composite type)

C'est une famille de types qui réunit les types dont la structure est composée d'éléments: les articles et les tableaux.

Type énuméré (enumeration type)

C'est un type dont les valeurs sont des symboles. Par exemple, *BOOLEAN* est un type énuméré. La valeur d'un objet de ce type sera à prendre dans l'ensemble des symboles qui lui sont associés: *FALSE* ou *TRUE* en l'occurrence. Plusieurs types énumérés peuvent se partager les mêmes symboles.

Type fichier (file type)

Voir Fichier.

Type physique (physical type)

Il existe en *VHDL* la notion d'unité de quantité. On définit par exemple dans le paquetage *STANDARD*, le type *TIME* qui est un type physique. C'est ici la notion de tempsvrai que connaît le simulateur et qui sera utile dans bon nombre d'instructions. C'est le seul type physique prédéfini. Un type physique est caractérisé par son unité de base (c'est la femtoseconde pour le type *TIME*), l'intervalle de ses valeurs autorisées et une éventuelle collection de sous-unités ainsi que leur correspondance entre elles.

Type scalaire (scalar type)

Un type scalaire est un type qui ne possède pas de sous–éléments. Les entiers, les flottants, les types physiques et les types énumérés sont des types scalaires. Les tableaux ou les enregistrements ne font pas partie des types scalaires, ils sont appelés types composites.

Unité de base (base unit)

C'est la plus petite unité d'un type physique. Toutes les autres unités s'expriment comme multiples (entiers) de cette unité.

Unité de conception (design unit)

C'est l'élément de base des bibliothèques et l'unité de compilation *VHDL*. Il en existe de cinq sortes : les spécifications de paquetages, les corps de paquetages, les spécifications d'entités, les architectures et les configurations.

Unité de conception générique

Voir Bloc générique.

Valeur courante d'un signal (current value)

Contrairement à l'affectation de variables, l'affectation de signaux ne change pas la valeur courante du signal, mais modifie les valeurs prévues (contenues dans son pilote) que sera susceptible de prendre ce signal. Lorsqu'on affecte un signal à une variable, c'est la valeur courante de ce signal qui est prise par la variable.

Valeur résolue d'un signal (resolved value)

C'est la valeur d'un signal résolu. Ce signal multi-sources est doté d'une fonction de résolution qui est appelée par le simulateur pour calculer la valeur du signal résultant des valeurs des différentes sources. Le résultat de cette fonction de résolution est la valeur résolue du signal.

Valeurs prévues d'un signal (projected output waveforms)

Ce sont des couples valeur/date. Sauf remise en cause par d'autres instructions, elles représentent les valeurs du signal aux dates spécifiées. Cet *agenda du signal*, associé à sa valeur courante constitue le pilote du signal.

Variable (variable)

Les variables sont une des classes d'objets *VHDL* (les deux autres sont les constantes et les signaux). Leurs valeurs peuvent être modifiées tout au long de la simulation. Contrairement aux signaux, on peut modifier (par affectation par exemple), la valeur courante d'une variable. C'est un objet lié à l'algorithmique et à la séquentialité. Son domaine d'utilisation est restreint aux sous-programmes et aux processus.

Variable accessible en lecture

Voir accessible en lecture.

VASG (VHDL Analysis and Group)

C'est le plus important des groupes de normalisation *VHDL*. Il a créé la norme *VHDL*. Son travail de déverminage de cette norme est achevé et l'essentiel de ses activités se recentre sur la définition de la nouvelle norme des années 92.

VDEG (VHDL Design Exchange Group)

Ce groupe de normalisation définit des sous-ensembles de *VHDL* et des paquetages standard. Des discussions sont en cours pour définir les meilleures façons d'intégrer le temps dans une description, ou pour définir des bibliothèques d'usage courant, voire des paquetages d'utilitaires. Le *repository* est maintenu

par ce groupe.

Vecteur (one-dimensional array)

C'est un tableau dont l'indice a une seule dimension. Les chaînes de caractères (type *STRING*) et les vecteurs de bits (type *BIT-VECTOR*) sont parmi les vecteurs les plus utilisés.

VHDL 7.2 (VHDL version 7.2: baseline language)

C'est la norme génératrice du standard *VHDL 1076-1987 IEEE*. Son existence dura six mois et laisse encore quelques traces dans la définition de certaines constructions.

VIFASG (VHDL Intermediate Format and Analysis Standardization Group)

Ce groupe de normalisation *VHDL* travaille à la standardisation d'un format intermédiaire. Ce format se présentera sous la forme d'un modèle de données et d'un jeu de sous–programmes permettant le stockage et la relecture de la sémantique d'un texte *VHDL*. Brancher un outil *X* sur une plate-forme *Y*, sera alors possible par une simple édition de liens.

Visibilité (visibility)

C'est la portion de code *VHDL* où une déclaration est visible, c'est-à-dire dénotable par une notation pointé ou directement par son nom.

RÉFÉRENCES

1. Weber, J. & Meaudre, M. (1997).
   VHDL: Du Langage au Cicrucit, du Circuit au Langage.
   ISBN : 2-225-82957-8, Masson S.A., Paris.

2. Van Den Bout, D. E. (1999).
   The Practical XILINX Designer Lab Book.
   ISBN : 0-13-021617-8, Prentice–Hall, Upper Saddle River, New Jersey 07458.

3. Airiau, R., Bergé, J. M., Olive, V. & Rouillard, J. (1990).
   VHDL : Du Langage à la Modélisation.
   ISBN : 2-88074-191-2, Presses Polytechniques et Universitaires Romandes et CNET-ENST.

Copyright © by J. L. Mayorquim 11 Fevrier 2002 - 12h46